Kozell YON-BROOKS

5727

51319

boolean logic circuit simulator

AQA Computer Science NEA

[Date]

Contents

[Analysis 4](#_Toc101908281)

[The Problem 4](#_Toc101908282)

[End Users 4](#_Toc101908283)

[Overview of Logic Gates 4](#_Toc101908284)

[Features Needed for GCSE Students 5](#_Toc101908285)

[Analysis of existing systems 7](#_Toc101908286)

[Acceptable Limitations 8](#_Toc101908287)

[Platform, Programming Language, and Modules 8](#_Toc101908288)

[General Objectives 9](#_Toc101908289)

[Specific Objectives 9](#_Toc101908290)

[Design 11](#_Toc101908291)

[Overview 11](#_Toc101908292)

[Input, Output, Process, Storage 12](#_Toc101908293)

[File Structure 12](#_Toc101908294)

[User Interface 14](#_Toc101908295)

[Main Interface 14](#_Toc101908296)

[Gate Canvas 15](#_Toc101908297)

[Toolbar 15](#_Toc101908298)

[Menubar for adding components 15](#_Toc101908299)

[Menubar for changing tool 15](#_Toc101908300)

[Truth Table Popup 16](#_Toc101908301)

[Components 17](#_Toc101908302)

[Nodes 18](#_Toc101908303)

[Data Structure of Logic Components WRONG? 19](#_Toc101908304)

[Algorithms 20](#_Toc101908305)

[Truth Table Generator: 20](#_Toc101908306)

# Analysis

## The Problem

GCSE Computer science is where a student’s introduction to logic gates may happen, this is a part of the curriculum but also helps to further a student’s logical thinking which can help in other areas of the subject, such as programming. A way to help me better understand logic gates were logic gate simulators, these simulators allowed me to create simple circuits using basic AND, OR, XOR, and NOT gates virtually. A simulator would allow the students to create circuits that give automatically give outputs without needing to be traced. However, the current simulators are too complex for what is needed for GCSE computer science and lack the teaching tools that my proposed system will include.

## End Users

The primary users of the logic gate simulators will be GCSE students studying computer science. However, as the computer science teachers will most likely introduce the students to this program and use it as part of their lessons, they will be considered as end-users as well.

## Overview of Logic Gates

Logic gates are a model of computation that take one or two inputs and returns a single output based on the gate's logical operation / Boolean function, they are the fundamentals of logical circuits and physical logic gates made of diodes and transistors are what allow computers to work. Logic gates can be combined to produce a certain output based on the inputs of the circuit. An AND gate (Figure 1) for example will take two inputs and return a True output if both inputs are True, and a False output otherwise. Inputs and outputs can be True or False as they are Boolean, this is usually represented as a 1 and 0 for True and False, respectively. Certain gates such as the Not gate will only need one input. There are other parts to the logic circuits besides the gates; switches, constant inputs, and clocks can provide initial inputs. Output can be handled by a simple ‘bulb’ that is on/lit for True and off for False. A more complex output such as a 4-bit digit would produce an integer output based on a binary sequence from 4 Boolean inputs.



Figure 1: AND Logic Gate

## Features Needed for GCSE Students

The truth table below (Figure 2) displays all the possible inputs and outputs in tabular form for the Logic operations AND, OR, XOR, and NOT. These are the only gates that are needed for the AQA course as per the specification (Figure 3).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Input** |  | **Output** |  |  |  |
| A | B | AND | OR | XOR | NOT (Input only A) |
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 |

Figure 2: Example of Truth Table

Table

Description automatically generated

Figure : AQA GCSE Specification for Boolean Algebra 3.4.2

As these are the only gates needed for the AQA course, they will be the only gates included, simplifying the program for the students. My solution would be more focused on teaching, specifically what is needed for the AQA GCSE specification. I had a short conversation with Mr Flynn about what features would make my program relevant to GCSE students. These included:

* Abstracting logic gates that are not needed.
* Including a checklist to determine if the user has used each of the gates at least once.
* A truth table generator to display the results of a circuit as a truth table.
* Converting a written Boolean expression into a truth table.
* Displaying the Boolean expression for the created circuit.
* Allowing individuals to save and load circuits to and from their computer’s local storage.
* Comparing a user-made circuit to a given Boolean expression to check if they have created the circuit correctly

These are all the features I believe are needed for the student to understand and solve GCSE level questions (Example shown in Figure 4).

A picture containing diagram

Description automatically generatedTable

Description automatically generated

Figure : Logic Gate question from AQA GCSE Computer Science June 2019 (8020/1)

## Analysis of existing systems

Figure 5, Figure 6, and Figure 7 below are examples of existing logic gate simulators.

Diagram

Description automatically generated

Figure : <https://academo.org/demos/logic-gate-simulator/>

A screenshot of a computer

Description automatically generated with medium confidence

Figure : <https://logic.ly/demo/>

Graphical user interface

Description automatically generated

Figure : <http://www.richardbowles.co.uk/resources/digital/tools/sim/sim.html>

|  |  |  |
| --- | --- | --- |
|  | **Advantages** | **Disadvantages** |
| **Figure 5** | * Lines between gates clearly show the connections of gates. | * Gates are behind a drop-down menu, making them harder to find. * No function to generate a truth table from the circuit created. |
| **Figure 6** | * Lines connecting gates are coloured to show the value of the gates they are coming out of. | * Gate selection is too complex for a GCSE student. |
| **Figure 7** | * Lines connecting gates are coloured to show the value of the gates they are coming out of. * Gates are clearly shown in the menu bar at top of the page. | * No dragging of gates around the canvas; makes repositioning difficult. * States of components do not dynamically update; the run button must be pressed to update them when a change is made. * Wires are placed manually; this is tedious and makes repositioning components difficult. |

## Acceptable Limitations

Hardware and software constraints – The program must run on school computers and, therefore, not resource-intensive. The school uses Windows 10 as its operating system, so the program does not need to run on any other OS.

The…

## Platform, Programming Language, and Modules

The program would run as either a web application or a windows desktop application. These choices are suitable as Highdown uses windows as the operating system on its computers, and all its computers have a browser and constant access to the internet.

A desktop application is a type of software that is directly installed onto the hard drive of the computer. It can be launched whenever, independent of other applications, i.e., it does not need a web browser to run within, like a web app. They also work regardless of internet connection (unless the program itself requires it).

A Web application is a type of software application that is used through the internet via a web browser. The files are stored on a remote server, the backend processing is done remotely, and the application is only accessible via an internet connection and browser.

As a desktop application will provide all the features needed for the program, I have chosen it over the web app as it does not need a server to be stored/run on nor require a constant internet connection to access.

The programming language I use will somewhat depend on the platform that the program will be run on. For a web app, the programming languages I would use would be Python, JavaScript, and HTML; I would also need to be familiar with CSS. For a windows application, many languages would be suitable, including python. As python is a language that I am already familiar with, it is my top choice for programming language, because the program will be a desktop application only python is needed.

For the GUI, python has an array of frameworks to help build user interfaces. These include PyQt5, tkinter, Kivy, wxPython, and PySimpleGUI. Kivy is currently my choice as it has many notable features and allows for the possibility of deploying the program on multiple platforms (Windows, macOS, iOS, Android). <https://kivy.org/#home>

## General Objectives

The general objective is to create a program that will allow a user to create logical gate circuits using draggable components that will evaluate and return an output based on the circuit. The components will be able to be linked and added/removed. The program will also show a truth table of an expression either taken from a component or input manually. The program should be responsive, easy to use, compatible with the school computers, and have the gates that are on the AQA GCSE computer science specification.

## Specific Objectives

1. The program must create and evaluate logic gate circuits.
   1. Can have gates, switches, and/or outputs (the three referred to as components).
      1. Each component will have a state of either true or false.
      2. The gates will be AND, OR, NOT, and XOR (from GCSE spec).
      3. The components can connect to each other.
         1. The components will have input and output nodes that point to their connected gates.
      4. The components can return their state as an output.
      5. The gates can calculate their state based on the gate’s Boolean operator and its inputs through an ‘evaluate’ method.
      6. The switches can be flipped, changing their state from false to true or vice-versa.
   2. There will be a board class containing the components.
      1. The board will create and destroy components.
      2. The board will store the created components in an array and tree structure.
      3. The board will tell the gates to connect and disconnect to and from each other.
      4. The board will traverse its tree of components, making them evaluate each of their states.
2. There will be a truth table generator­.
   1. It will produce a truth table for a given expression.
      1. It can use the expression of an output component.
      2. It can use an input Boolean expression string.
   2. It will produce a list of input combinations based on the number of inputs.
   3. It will substitute each combination of inputs into the expression to get an output.
   4. It will list the outputs alongside the input combinations in a table.
3. There will be a graphical user interface.
   1. It will allow for the dynamic placing of the components onto a canvas.
      1. It will use mouse inputs to drag the components and move them about the canvas.
   2. It will have a component toolbar that will have buttons that can add components to the canvas.
      1. The gates will have an indicator showing whether they have been used in the current session.
   3. It will have a tool toolbar that will determine what will happen when the components are interacted with depending on the tool selected.
      1. Connect tool: the program will tell the board to connect the selected components.
      2. Disconnect tool: the program will disconnect the selected components from each other.
      3. Move tool: the selected component can be moved about the canvas.
      4. Delete tool: the selected components will be deleted.
      5. Clear tool: All components will be deleted.
      6. Truth Table tool: the program will show a truth table based on the expression of the selected component.
         1. It will show a popup with a truth table and text input box that the user can enter their own expression into
   4. It will have a menu bar with options:
      1. Save, which saves the current circuit to a JSON file.
      2. Load, which loads a circuit from a JSON file.
      3. Quit, which exits the program.
         1. When clicked, a popup will appear asking if you are sure you want to exit.
   5. More…?

# Design

## Overview

This design section will show how the system will look and operate, and demonstrate how the objectives will be implemented.

I have structured the project so that the logic of the circuits are independent of the user interface, this allows the logic code to be used in different project and to be tested without the whole program.

The main window will house everything the user can interact with. The gate canvas will have a board object, this will control the logic component objects. The board will tell them to connect and disconnect, but the components will store who they are connected to themselves. A logic component will be created when a visual component is created and stored in the component array of the board. When the canvas interacts with the visual component, the canvas will ask for its logic component reference and pass that to the board. The board will use that reference to get the logic component from its component array for use.

Diagram

Description automatically generated

## Input, Output, Process, Storage

As Kivy can be used to develop for multiple operating systems, and I would like the program to be easily converted into an Android or iOS app, the user should be able to interact with the program with a single mouse button being pressed or released, and its movement around the screen, as this simulates what can be done on a touch screen.

|  |  |
| --- | --- |
| **Input** | **Output** |
| Mouse Inputs: Left Click Down, Mouse Move, Left Click Up. These inputs will be used for moving the components around a canvas and connecting or disconnecting them through dragging or clicking. (3.1.1)  Button Presses: Buttons around the user interface are what will control most of the program functionality, such as adding gates, selecting tools, and opening the truth table menu. (3.2, 3.3, 3.4)  Boolean expression input box: The truth table menu will display a truth table created for a user-entered Boolean expression; this will be a string entered into a text box which the truth table generator will then use as the input expression. (2.1.2) | Graphical Indicators: Most of the outputs will be done through the GUI, changing colours of components and lines based on their states, buttons showing as depressed/coloured to indicate when they are pressed, and borders around components when they are selected or being interacted with. (3.1)  Truth Table: A string |
| **Process** | **Storage** |
| Truth Table Generator: Will process a Boolean expression into a truth table.  Boolean Evaluation: Each gate will use the values of their inputs with their respective Boolean operators to produce their output value. | I do not think I will be able to implement the saving and loading in time ☹  Cannot think of any other storage? |

## File Structure

| CODE ¬

| Logic ¬

| TruthTable.py

| Components.py

| Board.py

| Gate Icons ¬

| (All component icon PNGs)

| GUI.py

Graphical user interface

Description automatically generated

Figure : Class diagram for circuit components

Figure 8 above shows the class diagram for the logic side of the program. The 4 gates (and, or, xor, not) will inherit from a parent class called Gate, the subclasses will contain the same attributes as the parent class but will overwrite some to fit the gates purpose, e.g. And\_Gate will overwrite the Gate classes currently empty attribute *type* with the string “and”. The gate subclasses will also overwrite the evaluate method so that it uses the correct boolean operator. As the not gate is different to the other 3 gates because of its single input rather than 2, it will overwrite other methods to account for only having 1 input.

There are also classes for switch and output objects, these have methods similar to the gate class, but with slight changes as they have only one node and no boolean evaluation method. Switch also has a flip method which will *flip* its state.

The switch, output and all gate classes all have a relationship with the Board class through composition. The board class will have 2 attributes, board\_id; an integer, and gates; an array of all components that currently exist. It will also have methods to add, remove, connect, and disconnect gates, a method to clear the board which will remove all components.

## User Interface

### Main Interface

When the program is initially opened, the first thing seen will be the full interactable program, allowing the user to start creating their circuits right away. The user interface should be simple and intuitive to use, no features should be hidden away or unclear on what their use is. Figure 9 shows a draft of how the main interface will look and the sections it will be split into.

Diagram

Description automatically generated

Figure : Main Graphical User Interface Design

### Gate Canvas

This is the canvas that will display Gates and Connection Lines. It will allow the gates two be placed anywhere within the canvas and the connection lines will dynamically update their positions to match where the components they are connected to are positioned.

### Toolbar

This will house the buttons to save, load, and open the truth table popup interface. It also has a quit button to quit the program, this will also bring up a popup asking for confirmation that the user wants to quit and that anything unsaved will be lost.

### Menubar for adding components

This sidebar will contain the components that the user can add to the canvas, it will have buttons that when pressed adds the component to the canvas. Each button will also have an indicator that shows whether the component has been used in the current session. There will also be labels alongside the components to show what they are but this isn’t shown in Figure 9.

### Menubar for changing tool

This is a sidebar with 5 buttons, the three top buttons are toggles that the user can choose between (one and only one of the three will always be on). The bottom two buttons, delete and clear, are not toggles and will immediately have an effect. Below is an explanation of each tool.

|  |  |
| --- | --- |
| **Tool** | **What it does.** |
| **Connect** | When toggled, the gate canvas will show the nodes of each component, the user may then drag from a node of one component to the node of another to connect them. This will create a connection line on the canvas linking the two components. The order of the connecting nodes (OUT 🡪 IN or IN 🡪 OUT) should not matter, the program will handle connecting them correctly. Connecting an out-node to an out-node or in-node to an in-node will not work. The components should not allow multiple connections to their in-nodes but should allow multiple connections from their out-nodes. |
| **Disconnect** | When toggled, the gate canvas will display the nodes of each component, the user may then click a node of a component and it will disconnect that component from the component that its node is connected to. When the components are disconnected, the connection lines connected to the clicked node will be removed. This means if the user clicks an out-node with multiple connections out of it, it will remove all connections. |
| **Move** | When toggled, the nodes are hidden, the user may click a component to select it, multiple components can be selected, click-and-drag a component to move it around the canvas, and double click the switch to flip its state. |
| **Delete** | When pressed, this will remove any selected components from the canvas. It will only work if the move tool is toggled. If not, it will toggle the move tool but will not remove the components. |
| **Clear** | When pressed, all components on the canvas will be deleted. Will toggle move tool when gate canvas is cleared. |

### Truth Table Popup

Truth Table Popup: This is where the user can generate truth tables. The expression input box will allow the user to input a Boolean expression, when the user presses the enter key or clicks the generate button a truth table for that expression will display in the truth table area. If a component is selected when the truth table button is pressed in the toolbar of the main interface, the expression box will automatically be populated with the expression of that component. Only one component must be selected, or it will default to empty on opening. There is also a cancel button to close the popup.

A picture containing diagram

Description automatically generated

Figure : Truth Table popup UI

### Components

These are all the icons for the components. The menu icons for switch and output will only appear in the menu bar for adding components. The gate components will only have one icon as they do not need to show their state. Switch and output will have Green and Red versions to show their states, output will also have an extra white icon for when its state is None (when the circuit is incomplete). They are all PNGs as they benefit from lossless compression and have transparency.

|  |  |
| --- | --- |
| **Component** | **Icons** |
| **Output:**  **(None, On, Off, Menu)** | **Shape, circle  Description automatically generated Shape, circle  Description automatically generated Shape, circle  Description automatically generated Chart, pie chart  Description automatically generated** |
| **Switch:**  **(On, Off, Menu)** | **Shape  Description automatically generated** |
| **XOR Gate** |  |
| **OR Gate** | **A picture containing icon  Description automatically generated** |
| **NOT Gate** | **Icon  Description automatically generated** |
| **AND Gate** | **Shape  Description automatically generated** |

Nodes: Component inputs and outputs are split into nodes, an AND gate for example will have 3 nodes, input\_node\_1, input\_node\_2, and out\_node, whereas a SWITCH or OUTPUT will have only an out\_node or in\_node\_1, respectively. The nodes are how the user will connect and disconnect components. The nodes will only appear when either the connect or disconnect tool is selected.

|  |  |
| --- | --- |
| **And Gate** | **And Gate with visible Nodes** |
| Shape  Description automatically generated | Icon  Description automatically generated |

### Nodes

Component nodes will be visually linked by connection lines; these lines will be drawn onto the canvas and will move with the components so that they always start on the out\_node of one component and end on the correct in\_node of the other component. Below shows how the line’s colour will show the state of the gate it’s exiting, green for true, red for false, and black/white for an incomplete or invalid circuit. The lines will also shape themselves to always be made up of only horizontal or vertical parts. This will help with following the lines and organising the canvas.

Diagram, schematic

Description automatically generated

Saving, Loading… :/

## Data Structures

### Logic Components

The logic components in a circuit will be treated as a binary tree. The outputs or last component in a circuit would act as the root node, each node would represent a component and would have either 2 children (AND, OR, XOR), 1 child (NOT) or no children (SWITCH). This conforms to the binary tree structure.

Because each component can have multiple output connections and therefore each node can have multiple parents, the overall structure does not fully match a binary tree. However when the logic is evaluated the logic will work from the root component and will not need access to any output components of the components.

A preorder traversal recursive method will check if each component has all its input connections, if not it will return its current boolean output. If the component does have both input connections it will call a preorder traversal method on the left child/1st input. When the left child has returned some data, it will store it and then repeat for the right child/2nd input if there is one. Once all of its children have returned data, the component will use it in an evaluate method to get a boolean value and return that.

Below is a pseudocode representation of a node that has two inputs.

CLASS Node2:

instance.data = [None, None]

instance.left = None

instance.right = None

FUNCTION set\_data(data):

instance.data = data

FUNCTION set\_left(left\_node):

instance.left = left\_node

FUNCTION set\_right(right\_node):

instance.right = right\_node

FUNCTION preorder\_traverse():

IF (instance has no left or right) THEN:

RETURN instance.data

ELSE

instance.data[0] 🡨 instance.left.preorder\_traverse()

instance.data[1] 🡨 instance.right.preorder\_traverse()

RETURN instance.data

### Truth Table

The truth table will be stored using a dictionary. The dictionary will have keys that are strings, which is valid as strings are hashable, and values which are arrays of integers.

Example\_Dictionary: dict[

## Algorithms

### Truth Table Generator:

FUNCTION generateTruthTable(expression):

input\_expression 🡨 expression

expression 🡨 expression.lower()

expression 🡨 expression.replace("not", "2+~")

expression 🡨 expression.replace("and", "&")

expression 🡨 expression.replace("xor", "^")

expression 🡨 expression.replace("or", "|")

operators 🡨 ["2+~", "&", "^", "|", "(", ")"]

variables 🡨 []

temp\_exp 🡨 expression.replace("(", "")

temp\_exp 🡨 temp\_exp.replace(")", "")

temp\_exp 🡨 temp\_exp.split(" ")

FOR(i IN temp\_exp):

IF (i NOT IN operators) AND (i NOT IN variables):

variables.append(i)

num\_variables 🡨 len(variables)

temp 🡨 []

FOR(i 🡨 0 TO num\_variables):

temp.append(0)

FOR(i 🡨 0 TO num\_variables):

temp.append(1)

p 🡨 get\_permutations(temp, num\_variables)

variable\_permutations 🡨 []

FOR(i IN p):

IF i NOT IN variable\_permutations:

variable\_permutations.append(i)

dictionary = DICTIONARY()

FOR i,j IN ENUMERATE(variables):

FOR ii IN variable\_permutaions:

dictionary[j] = i[ii]

final\_dictionary 🡨 getTruthTableOutputs(expression, dictionary, len(variable\_permutations))

Explanation:

The function is given a Boolean expression as a string. The first 7 lines will format the string into an expression that can be used by the getTruthTableOutputs function.

An array called variables is created and populated with each variable in the expression, they are found by removing any brackets and splitting the string by its spaces into an array. Then each item is checked if it’s in the list of operators and if not, added to variables.

The number of variables is found from the length of the variables array and a temporary array will be created with that amount of 0s followed by the same amount of 1s. This is for the get\_permutations function to work correctly.

The get\_permutations will return a list of the permutations of the temp array as tuples of size num\_variables. E.g., for an expression with 2 variables:

get\_permutations([0,0,1,1], 3) will return

[(0, 0), (0, 1), (0, 1), (0, 0), (0, 1), (0, 1), (1, 0), (1, 0), (1, 1), (1, 0), (1, 0), (1, 1)]

The duplicated tuples are then removed to make the variable\_permuations array:

[(0, 0), (0, 1), (1, 0), (1, 1)]

This method was used as it produced a nicely ordered list of all possible value combinations used in a truth table.

A dictionary is then made with the variables as keys, and an array of their ordered values, found by combining the values with the same index in the variable\_permutations array tuples.

E.g. with variables A and B:

{‘A’:[0,0,1,1], ‘B’:[0,1,0,1]}

The formatted expression, dictionary, and the number of tuples in the variable\_permutations will be given to the getTruthTableOutputs function which will return the same dictionary but with an extra OUT key with its values.

A tuple of the final dictionary and the original expression will then be returned.

FUNCTION getTruthTableOutputs(expression, dictionary, num\_permutations):

truth\_dictionary 🡨 dictionary

list\_of\_outputs 🡨 []

FOR (i 🡨 0 TO num\_permutations):

expression\_with\_variables 🡨 expression

FOR (variable in truth\_dictionary):

value 🡨 INT\_TO\_STRING(dictionary[variable][i]))

expression\_with\_variables 🡨 expression\_with\_variables.replace(variable, value)

TRY:

output 🡨 EVALUATE(expression\_with\_variables)

EXCEPT SyntaxError as e:

OUTPUT(e, expression\_with\_variables)

return "Error"

list\_of\_outputs.append(output)

truth\_dictionary['OUT'] 🡨 list\_of\_outputs

RETURN truth\_dictionary

Explanation:

The function is given a formatted expression, a dictionary with a key-value pair of each variable and the ordered array of its values, and num\_permutations which is the number of values the variables will have.

The function creates a copy of the given dictionary called truth\_dictionary and an array called list\_of\_outputs. Within a for loop counting from i = 0 to the num\_permutations, a copy of the expression called expression will be created, a for loop iterating through the truth\_dictionary variables will then produce a value by converting the value at index i of the dictionary value with the variable as key to a string. All occurrences of that variable in the expression\_with\_variables string are replaced with

Some more things

# Technical Solution

## Some things to note <- rename the heading

Kivy has its own declarative language which makes creating the widget tree for the program easier. I used this mainly to make designing the static parts of the interface easier...

## Truth Table

from itertools import permutations

from os import system as sys

from typing import Union # For type hinting

def get\_tt\_outputs(expression: str, dictionary: dict[str, list], num\_permutations: int) -> dict[str, list]:

    truth\_dictionary = dictionary

    list\_of\_outputs = []

    for i in range(num\_permutations):

        expression\_with\_variables = expression

        for variable in truth\_dictionary:

            expression\_with\_variables = expression\_with\_variables.replace(variable, str(dictionary[variable][i]))

        try:

            output = eval(expression\_with\_variables)

        except SyntaxError as e:

            print(e, expression\_with\_variables)

            return "Error"

        list\_of\_outputs.append(output)

    truth\_dictionary['OUT'] = list\_of\_outputs

    return truth\_dictionary

def generate\_truth\_table(expression: str) -> Union[Tuple[dict[str, list], str], str]:

    '''Returns a truth table in dictionary form'''

    input\_expression = expression

    expression = expression.lower()

    expression = expression.replace("not ", "2+~ ")

    expression = expression.replace("not", "2+~ ")

    expression = expression.replace("and", "&")

    expression = expression.replace("xor", "^")

    expression = expression.replace("or", "|")

    operators = ["2+~", "&", "^", "|", "(", ")"]

    variables = []

    temp\_exp = expression.replace("(", "").replace(")", "")

    for var in temp\_exp.split(" "):

        if var not in operators and var not in variables:

            variables.append(var)

    num\_variables = len(variables)

    temp = [0 for \_ in range(num\_variables)] + [1 for \_ in range(num\_variables)]

    perms = permutations(temp, num\_variables)

    variable\_permutations = []

    for perm in perms:

        if perm not in variable\_permutations:

            variable\_permutations.append(perm)

    dictionary = dict()

    for i, key in enumerate(variables):

        dictionary[key] = [perm[i] for perm in variable\_permutations]

    final\_dictionary = get\_tt\_outputs(expression, dictionary, len(variable\_permutations))

    if final\_dictionary == "Error":

        return "Invalid Input"

    else:

        return (final\_dictionary, input\_expression)

if \_\_name\_\_ == '\_\_main\_\_':

    '''For Testing'''

    sys('clear')

    exp = input('Here: ')

    print(generate\_truth\_table(exp))

    print()

## Logic Board

from logic.gates import \*

from logic.truth\_table import \*

class Board:

    '''Logic Board class.'''

    ID = 0

    def \_\_init\_\_(self) -> None:

        self.id = Board.ID

        Board.ID += 1

        self.gates = []

    def add\_gate(self, gate: Gate) -> None:

        '''Adds given component to gate array.'''

        self.gates.append(gate)

    def remove\_gate(self, gate: Gate) -> None:

        '''Removes given component from gate array.'''

        gate.disconnect\_all()

        self.gates.remove(gate)

    def connect\_gate(self, in\_gate: Gate, out\_gate: Gate, node: int) -> bool:

        '''Connects two given components together'''

        if node == 1:

            if in\_gate.connect\_node(out\_gate, 1):

                return True

        if node == 2:

            if in\_gate.get\_gate\_type() != "not" or in\_gate.get\_gate\_type() != "output":

                if in\_gate.connect\_node(out\_gate, 2):

                    return True

        return False

    def disconnect\_gate(self, in\_gate: Gate, out\_gate: Gate) -> bool:

        '''Disconnects two given components from eachother'''

        return in\_gate.disconnect\_node(out\_gate)

    def clear\_board(self) -> None:

        '''Removes all components from gate array'''

        for gate in self.gates[:]:

            self.remove\_gate(gate)

## Logic Components

from typing import Union # For type hinting

class Gate:

    '''Logic Gate component class. Parent class for logic gates.'''

    ID = 0

    def \_\_init\_\_(self) -> None:

        self.\_input\_nodes = [None, None]

        self.\_output\_nodes = []

        self.\_output = None

        self.\_type = ''

        self.\_expression = None

        self.name = ''

        self.id = Gate.ID

        Gate.ID += 1

    def evaluate(self) -> None:

        '''Not implemented in parent class.'''

        pass

    def has\_input(self, node=0) -> bool:

        '''Checks if nodes are connected to other components.'''

        if node == 0:

            return bool(self.\_input\_nodes[0] and self.\_input\_nodes[1])

        elif node == 1:

            return bool(self.\_input\_nodes[0])

        elif node == 2:

            return bool(self.\_input\_nodes[1])

    def connect\_node(self, gate, node) -> bool:

        '''Connects given component to given node of self'''

        if gate in self.\_input\_nodes:

            return False

        if node == 1:

            if not self.has\_input(1):

                self.\_input\_nodes[0] = gate

                gate.connect\_node(self, -1)

                self.update\_expression()

                return True

            else:

                return False

        elif node == 2:

            if not self.has\_input(2):

                self.\_input\_nodes[1] = gate

                gate.connect\_node(self, -1)

                self.update\_expression()

                return True

            else:

                return False

        elif node == -1:

            self.\_output\_nodes.append(gate)

            return True

    def disconnect\_node(self, gate) -> bool:

        '''Disconnects given component from self.'''

        if self.\_input\_nodes[0] is gate:

            self.\_input\_nodes[0].disconnect\_node(self)

            self.\_input\_nodes[0] = None

            self.update\_expression()

            return True

        elif self.\_input\_nodes[1] is gate:

            self.\_input\_nodes[1].disconnect\_node(self)

            self.\_input\_nodes[1] = None

            self.update\_expression()

            return True

        elif gate in self.\_output\_nodes:

            self.\_output\_nodes.remove(gate)

            return True

        else:

            return False

    def disconnect\_all(self) -> None:

        '''Disconnects all inputs and outputs from self.'''

        if self.\_input\_nodes[0] != None:

            self.\_input\_nodes[0].disconnect\_node(self)

            self.\_input\_nodes[0] = None

            self.update\_expression()

        if self.\_input\_nodes[1] != None:

            self.\_input\_nodes[1].disconnect\_node(self)

            self.\_input\_nodes[1] = None

            self.update\_expression()

        for gate in self.\_output\_nodes:

            gate.disconnect\_node(self)

    def \_process(self) -> None:

        '''Uses evaluate method to set output of self. If both input nodes empty, or any input nodes output is None, output is set to None.'''

        if self.has\_input():

            var1 = self.\_input\_nodes[0].getOutput()

            var2 = self.\_input\_nodes[1].getOutput()

            if var1 == None or var2 == None:

                self.\_output = None

            else:

                self.\_output = self.evaluate(var1, var2)

        else:

            self.\_output = None

    def update\_expression(self) -> None:

        '''Updates the boolean expression of self. Used when connecting or disconnecting components to/from self.'''

        if self.has\_input():

            if self.\_input\_nodes[0].getGateType() == 'switch':

                exp1 = self.\_input\_nodes[0].getExpression()

            else:

                exp1 = f"({self.\_input\_nodes[0].getExpression()})"

            if self.\_input\_nodes[1].getGateType() == 'switch':

                exp2 = self.\_input\_nodes[1].getExpression()

            else:

                exp2 = f"({self.\_input\_nodes[1].getExpression()})"

            self.\_expression = f"{exp1} {self.\_type} {exp2}"

        else:

            self.\_expression = None

    def get\_expression(self) -> str:

        '''Returns boolean expression of self.'''

        return self.\_expression

    def get\_gate\_type(self) -> str:

        '''Returns the type of component that self is.'''

        return self.\_type

    def get\_name(self) -> str:

        '''Returns the name of self. Name is string: type\_id.'''

        return self.name

    def get\_output(self) -> Union[int, None]:

        '''Calls process method, then return output of self.'''

        self.\_process()

        return self.\_output

class And\_Gate(Gate):

    '''Logic And\_Gate component class, child of Gate class.'''

    def \_\_init\_\_(self) -> None:

        super().\_\_init\_\_()

        self.\_type = 'and'

        self.name = f"{self.\_type}\_{str(self.id)}"

    def evaluate(self, var1, var2) -> int:

        '''Return the result of an and comparison bewtween two variables.'''

        return int(var1 and var2)

class Or\_Gate(Gate):

    '''Logic Or\_Gate component class, child of Gate class.'''

    def \_\_init\_\_(self) -> None:

        super().\_\_init\_\_()

        self.\_type = 'or'

        self.name = f"{self.\_type}\_{str(self.id)}"

    def evaluate(self, var1, var2) -> int:

        '''Return the result of an or comparison bewtween two variables.'''

        return int(var1 or var2)

class Xor\_Gate(Gate):

    '''Logic Xor\_Gate component class, child of Gate class.'''

    def \_\_init\_\_(self) -> None:

        super().\_\_init\_\_()

        self.\_type = 'xor'

        self.name = f"{self.\_type}\_{str(self.id)}"

    def evaluate(self, var1, var2) -> int:

        '''Return the result of an xor comparison bewtween two variables.'''

        return int((var1 and not var2) or (not var1 and var2))

class Not\_Gate(Gate):

    '''Logic Not\_Gate component class, child of Gate class.'''

    def \_\_init\_\_(self) -> None:

        super().\_\_init\_\_()

        self.\_type = 'not'

        self.name = f"{self.\_type}\_{str(self.id)}"

    def evaluate(self, var1) -> int:

        '''Return the result of a not operation on a variable.'''

        return int((not var1))

    def has\_input(self) -> bool:

        '''Checks if input node is connected to another component.'''

        return bool(self.\_input\_nodes[0])

    def \_process(self) -> None:

        '''Uses evaluate method to set output of self. If has no input node, or input nodes output is None, output is set to None.'''

        if self.has\_input():

            var1 = self.\_input\_nodes[0].getOutput()

            if var1 == None:

                self.\_output = None

            else:

                self.\_output = self.evaluate(var1)

        else:

            self.\_output = None

    def connect\_node(self, gate, node) -> bool:

        '''Connects given component to given node of self.'''

        if node == 1:

            if not self.has\_input():

                self.\_input\_nodes[0] = gate

                gate.connect\_node(self, -1)

                self.update\_expression()

                return True

            else:

                return False

    def disconnect\_node(self, gate) -> bool:

        '''Disconnects given component from self.'''

        if self.\_input\_nodes[0] is gate:

            self.\_input\_nodes[0].disconnect\_node(self)

            self.\_input\_nodes[0] = None

            self.update\_expression()

            return True

        elif gate in self.\_output\_nodes:

            self.\_output\_nodes.remove(gate)

            return True

        else:

            return False

    def disconnect\_all(self) -> None:

        '''Disconnects all inputs and outputs from self.'''

        if self.\_input\_nodes[0] != None:

            self.\_input\_nodes[0].disconnect\_node(self)

            self.\_input\_nodes[0] = None

            self.update\_expression()

    def update\_expression(self) -> None:

        '''Updates the boolean expression of self. Used when connecting or disconnecting components to/from self.'''

        if self.has\_input():

            self.\_expression = str(f"{self.\_type}({self.\_input\_nodes[0].getExpression()})")

        else:

            self.\_expression = None

class Switch:

    '''Logic Switch component class.'''

    ID = 0

    def \_\_init\_\_(self) -> None:

        self.\_output = 0

        self.\_output\_nodes = []

        self.\_type = 'switch'

        self.id = chr(Switch.ID+97)

        Switch.ID += 1

        self.name = (f"{self.\_type}\_{self.id}")

    def connect\_node(self, gate, node) -> bool:

        '''Connects given component to given node of self.'''

        if gate not in self.\_output\_nodes and node == -1:

            self.\_output\_nodes.append(gate)

            return True

        return False

    def disconnect\_node(self, gate) -> bool:

        '''Disconnects given component from self.'''

        if gate in self.\_output\_nodes:

            self.\_output\_nodes.remove(gate)

            return True

        return False

    def disconnect\_all(self) -> None:

        '''Disconnects all inputs and outputs from self.'''

        for gate in self.\_output\_nodes:

            gate.disconnect\_node(self)

    def flip(self) -> None:

        '''Changes output of self from 0 to 1 or 1 to 0.'''

        self.\_output = int(not(self.\_output))

    def get\_name(self) -> str:

        '''Returns the name of self. Name is string: type\_id.'''

        return self.name

    def get\_gate\_type(self) -> str:

        '''Returns the type of self that self is.'''

        return self.\_type

    def get\_expression(self) -> str:

        '''Return character id of self.'''

        return self.id

    def get\_output(self) -> Union[int, None]:

        '''Returns output of self.'''

        return self.\_output

class Output:

    '''Logic Output component class.'''

    ID = 65

    def \_\_init\_\_(self) -> None:

        self.\_output = None

        self.\_input\_nodes = [None]

        self.\_type = 'output'

        self.id = chr(Output.ID)

        Output.ID += 1

        self.name = (f"{self.\_type}\_{self.id}")

    def \_process(self) -> None:

        '''Sets output of self to output of input node. If input node is empty, output is set to None.'''

        try:

            self.\_output = self.\_input\_nodes[0].getOutput()

        except AttributeError as e:

            print(e, len(self.\_input\_nodes))

            self.\_output = None

    def connect\_node(self, gate, node) -> bool:

        '''Connects given component to given node of self.'''

        if self.\_input\_nodes[0] == None:

            self.\_input\_nodes[0] = gate

            gate.connect\_node(self, -1)

            return True

        else:

            return False

    def disconnect\_node(self, gate) -> bool:

        '''Disconnects given component from self.'''

        if self.\_input\_nodes[0] is gate:

            self.\_input\_nodes[0] = None

            gate.disconnect\_node(self)

            return True

        return False

    def disconnect\_all(self) -> None:

        '''Disconnects all inputs and outputs from self.'''

        if self.\_input\_nodes[0] == None:

            pass

        else:

            self.\_input\_nodes[0].disconnect\_node(self)

            self.\_input\_nodes[0] = None

        self.\_process()

    def get\_expression(self) -> str:

        '''Return boolean expression of input component.'''

        return self.\_input\_nodes[0].getExpression()

    def get\_name(self) -> str:

        '''Returns the name of self. Name is string: type\_id.'''

        return self.name

    def get\_gate\_type(self) -> str:

        '''Returns the type of component that self is.'''

        return self.\_type

    def get\_output(self) -> Union[int, None]:

        '''Calls process method, then return output of self.'''

        self.\_process()

        return self.\_output

## GUI (Python)

'''Doctrings Incomplete'''

## Kivy GUI ##

import sys

import platform

import kivy

kivy.require('2.0.0')

from kivy.app import App

from kivy.clock import Clock

from kivy.config import Config

from kivy.core.window import Window

from kivy.graphics import Color, Line

from kivy.lang import Builder

from kivy.properties import ListProperty, StringProperty

from kivy.uix.behaviors import DragBehavior

from kivy.uix.floatlayout import FloatLayout

from kivy.uix.image import Image

from kivy.uix.popup import Popup

from kivy.uix.widget import Widget

from tabulate import tabulate

from logic.board import Board

from logic.gates import \*

from logic.truth\_table import \*

Window.maximize()

class ExitPopup(Popup):

    '''Popup that opens when quit button is pressed.'''

    def close\_window(self):

        '''Exits program.'''

        sys.exit()

class TruthPopup(Popup):

    '''Popup for truth table generator.'''

    FONT = StringProperty()

    def \_\_init\_\_(self, \*args, \*\*kwargs):

        super().\_\_init\_\_(\*args, \*\*kwargs)

        if platform.system() == "Linux":

            self.FONT = "FreeMonoBold"

        else:

            self.FONT = "LUCON"

    def open(self, root, \*args, \*\*kwargs):

        '''Overwrite of kivy method, inserts selected components expression into input field.'''

        super().open(\*args, \*\*kwargs)

        selected\_gate = root.ids["gateCanvas"].get\_selected\_gates()

        if len(selected\_gate) != 1:

            text = 'Enter Boolean Expression Here'

        else:

            text = selected\_gate[0].logic\_gate.getExpression()

        self.ids["truth\_input"].text = text

        self.generate()

    def generate(self):

        '''Using expression in input field, generates a truth table and diplays it.'''

        input\_expression = self.ids["truth\_input"].text

        out = generate\_truth\_table(input\_expression)

        if out == "Invalid Input":

            if input\_expression == "Enter Boolean Expression Here":

                self.ids["truth\_label"].text = "Truth table will appear here"

            else:

                self.ids["truth\_label"].text = "Invalid Input."

        else:

            out2 = tabulate(out[0], headers=out[0].keys(), tablefmt="pretty")

            self.ids["truth\_label"].text = f"{out[1]}\n{out2}"

class ConnectionLine(Widget):

    '''Class for lines that connects components together'''

    color\_state = ListProperty()

    def \_\_init\_\_(self, out\_gate, in\_gate, in\_node, \*\*kwargs):

        super().\_\_init\_\_(\*\*kwargs)

        self.color\_states = {None:(1,1,1,1), 0:(1,0,0,1), 1:(0,1,0,1)}

        self.color\_state = (1,1,1,1)

        self.out\_gate = out\_gate

        self.in\_gate = in\_gate

        self.in\_node = in\_node

        self.points = []

        self.state = None

        self.inline = Line(points=self.points, width=1.5, cap='round', joint='round')

        self.outline = Line(points=self.points, width=4, cap='round', joint='round')

        self.canvas.add(self.inline)

        self.canvas.before.add(Color(rgba=(0,0,0,1)))

        self.canvas.before.add(self.outline)

        self.update\_pos()

        self.update\_state()

    def get\_gates(self):

        '''Returns tuple: (component line is exiting, component line is entering)'''

        return (self.out\_gate, self.in\_gate)

    def get\_nodes(self):

        '''Returns tuple: (component line is exiting, node its exiting), (component line is entering, node its entering))'''

        return ((self.out\_gate, -1), (self.in\_gate, self.in\_node))

    def get\_turn\_points(self, out\_node\_pos, in\_node\_pos):

        '''Returns the extra points to make the line the shape it is <<<<<'''

        x1 = out\_node\_pos[0]

        x2 = in\_node\_pos[0]

        y1 = out\_node\_pos[1]

        y2 = in\_node\_pos[1]

        x\_mid = (x1 + x2)/2

        turn1 = [x\_mid, y1]

        turn2 = [x\_mid, y2]

        return turn1 + turn2

    def update\_state(self):

        '''Updates colour of line to match the output of the component its exiting.'''

        self.state = self.out\_gate.get\_state()

        self.color\_state = self.color\_states[self.state]

    def update\_pos(self):

        '''Updates the points of the line to match the position of the components its connect to.'''

        out\_node\_pos = self.out\_gate.get\_node\_pos(-1)

        in\_node\_pos = self.in\_gate.get\_node\_pos(self.in\_node)

        self.points = out\_node\_pos + self.get\_turn\_points(out\_node\_pos, in\_node\_pos) + in\_node\_pos

        self.outline.points = self.points

        self.inline.points = self.points

class GateCanvas(FloatLayout):

    '''Canvas where components and connection lines are placed onto'''

    def \_\_init\_\_(self, \*\*kwargs):

        super().\_\_init\_\_(\*\*kwargs)

        self.connection\_lines = []

        self.in\_connection = None

        self.out\_connection = None

        self.gates = []

        self.tool = "move"

        self.board = Board()

        self.gate\_dict = {

            "and":DragAndGate,

            "or":DragOrGate,

            "not":DragNotGate,

            "xor":DragXorGate,

            "switch":DragSwitch,

            "output":DragOutput,

            "clock":DragClock

        }

    def add\_connection\_line(self, out\_gate, in\_gate, in\_node):

        '''Creates connection line.'''

        line = ConnectionLine(out\_gate, in\_gate, in\_node)

        self.connection\_lines.append(line)

        self.add\_widget(line)

    def update\_connection\_lines(self):

        '''Updates the postions of all connection lines on the canvas'''

        for line in self.connection\_lines[:]:

            line.update\_pos()

    def update\_states(self):

        '''Updates the states of all components and connection lines on the canvas'''

        for gate in self.gates:

            gate.update\_state()

        for line in self.connection\_lines:

            line.update\_state()

    def set\_tool(self, tool):

        '''Sets the tool, can be move, connect, disconnect'''

        self.tool = tool

        if tool is "move":

            for gate in self.gates[:]:

                gate.hide\_nodes()

        else:

            for gate in self.gates[:]:

                gate.show\_nodes()

                self.deselect\_gates()

    def on\_touch\_down(self, touch):

        '''Checks if there is a touch down within the canvas. If touch is double click, set tool to move. Calls a method based on the current tool if single click.'''

        if self.collide\_point(\*touch.pos):

            if touch.is\_double\_tap and self.tool != "move":

                self.root.set\_tool("move")

                return True

            if self.tool == "connect":

                return self.connect\_down(touch)

            if self.tool == "disconnect":

                return self.disconnect\_down(touch)

            if self.tool == "move":

                return self.move\_down(touch)

    def on\_touch\_up(self, touch):

        '''Checks if there is a touch up within the canvas, Calls a method based on the cuurent tool if there is'''

        if self.collide\_point(\*touch.pos):

            if self.tool == "connect":

                return self.connect\_up(touch)

            if self.tool == "disconnect":

                pass

            if self.tool == "move":

                pass

        return super().on\_touch\_up(touch)

    def connect\_down(self, touch):

        '''Checks for node collision on touch down. If collision replace either out\_connection or in\_connection attribute with (node, gate) tuple'''

        self.deselect\_gates()

        for gate in self.gates[:]:

            if node := gate.get\_node\_collide(touch):

                if node == -1:

                    self.out\_connection = (node, gate)

                else:

                    self.in\_connection = (node, gate)

                gate.select\_node(node)

                return True

        return True

    def connect\_up(self, touch):

        '''Checks for node collision on touch up. If collision, replace either out\_connection or in\_connection attribute with (node, gate) tuple.

        If out\_connection and in\_connection both have values, connects the component nodes together'''

        for gate in self.gates[:]:

            if node := gate.get\_node\_collide(touch):

                if node == -1:

                    self.out\_connection = (node, gate)

                else:

                    self.in\_connection = (node, gate)

                gate.select\_node(node)

                if self.in\_connection is None:

                    print("Can't connect output to output")

                elif self.out\_connection is None:

                    print("Cant connect input to input")

                else:

                    in\_gate = self.in\_connection[1]

                    out\_gate = self.out\_connection[1]

                    in\_node = self.in\_connection[0]

                    if self.board.connect\_gate(in\_gate.get\_logic\_gate(), out\_gate.get\_logic\_gate(), node=in\_node):

                        self.add\_connection\_line(out\_gate, in\_gate, in\_node)

                    self.update\_states()

        for gate in self.gates[:]:

            gate.deselect\_nodes()

        self.out\_connection = None

        self.in\_connection = None

        return True

    def move\_down(self, touch):

        '''Calls on\_touch\_down method for children. If canvas clicked instead of a child, deselects all gates.'''

        for child in self.gates[::-1]:

            if child.dispatch('on\_touch\_down', touch):

                return True

        if not self.collide\_point(\*touch.pos):

            return False

        self.deselect\_gates()

        return True

    def disconnect\_down(self, touch):

        '''Checks for node collision on touch down. If collision, disconnect clicked node from any nodes its connected to and remove connection lines.'''

        self.deselect\_gates()

        for gate in self.gates[:]:

            if node := gate.get\_node\_collide(touch):

                if node == -1:

                    for line in self.connection\_lines:

                        gate\_node\_tuple = line.get\_nodes()

                        if gate\_node\_tuple[0][0] == gate:

                            gate2 = gate\_node\_tuple[1][0]

                            self.board.disconnect\_gate(gate2.get\_logic\_gate(), gate.get\_logic\_gate())

                            self.connection\_lines.remove(line)

                            self.remove\_widget(line)

                elif node == 1 or node == 2:

                    for line in self.connection\_lines:

                        gate\_node\_tuple = line.get\_nodes()

                        if gate\_node\_tuple[1] == (gate,node):

                            gate2 = gate\_node\_tuple[0][0]

                            self.board.disconnect\_gate(gate.get\_logic\_gate(), gate2.get\_logic\_gate())

                            self.connection\_lines.remove(line)

                            self.remove\_widget(line)

        self.update\_states()

        return True

    def get\_selected\_gates(self):

        '''Returns a list of all components that are selected.'''

        return [child for child in self.gates[:] if child.is\_selected()]

    def deselect\_gates(self):

        '''Deselects all selected components.'''

        [child.deselect() for child in self.gates[:]]

    def add\_gate(self, gate\_type):

        '''Creates new visual component and adds it to canvas, adds logic component to logic board.'''

        new\_gate = self.gate\_dict[gate\_type](parent\_rect=(self.x, self.y, self.width, self.height))

        self.add\_widget(new\_gate)

        self.gates.append(new\_gate)

        new\_gate.root = self.root

        self.board.add\_gate(new\_gate.get\_logic\_gate())

        new\_gate.update\_state()

    def delete\_gate(self):

        '''Deletes selected components from canvas and logic board, also removes any connection lines they are connected to.'''

        if not self.tool == "move":

            self.root.set\_tool("move")

        else:

            for gate in self.get\_selected\_gates():

                #This is probably a terrible way to do this, so find a better one

                for line in self.connection\_lines[:]:

                    if gate in line.get\_gates():

                        self.connection\_lines.remove(line)

                        self.remove\_widget(line)

                        del line

                self.board.remove\_gate(gate.get\_logic\_gate())

                self.remove\_widget(gate)

                self.gates.remove(gate)

                del gate

                self.update\_states()

    def clear\_canvas(self):

        '''Deletes all components from canvas and logic board'''

        self.board.clear\_board()

        self.clear\_widgets()

        self.gates = []

class DragGate(DragBehavior, FloatLayout):

    '''Visual components parent class. DOCSRINGS NOT DONE'''

    def \_\_init\_\_(self, parent\_rect, \*\*kwargs):

        super().\_\_init\_\_(\*\*kwargs)

        '''Sets kivy properties'''

        self.drag\_rectangle = parent\_rect

        self.drag\_timeout = 500

        self.drag\_distance = 5

        self.allow\_stretch = True

        self.size\_hint = None, None

        self.size = (100,100)

        self.pos = (parent\_rect[0]+parent\_rect[2])/2, (parent\_rect[1]+parent\_rect[3])/2

        '''Sets image of component'''

        self.img = Image(pos = self.pos, size\_hint = (1,1))

        self.add\_widget(self.img)

        '''Initialises the nodes of the component and adds them to node array'''

        self.nodes = []

        self.nodes\_init()

        '''Creates the selection border and adds to component.'''

        self.border = Line(rounded\_rectangle = (self.x, self.y, self.width, self.height, 10))

        self.canvas.add(Color(0,0,0,1))

        self.canvas.add(self.border)

        '''Creates a parallel logic version of the component. None in parent class.'''

        self.logic\_gate = None

        self.state = None

        '''Creates dragged attribute and selects the component.'''

        self.dragged = False

        self.select()

    def nodes\_init(self):

        '''Initialises the nodes of the component, positions them and adds them to the component widget.'''

        node\_source="GateIcons/node.png"

        self.in\_node\_1 = Image(source=node\_source, size\_hint=(None, None), size=(20,20))

        self.in\_node\_2 = Image(source=node\_source, size\_hint=(None, None), size=(20,20))

        self.out\_node = Image(source=node\_source, size\_hint=(None, None), size=(20,20))

        self.nodes.append(self.in\_node\_1)

        self.nodes.append(self.in\_node\_2)

        self.nodes.append(self.out\_node)

        self.add\_widget(self.in\_node\_1)

        self.add\_widget(self.in\_node\_2)

        self.add\_widget(self.out\_node)

        self.hide\_nodes()

    def get\_node\_pos(self, node):

        '''Return the coordinates of the given node.'''

        return self.get\_node(node).center

    def update\_nodes(self):

        '''Updates the positions of the components nodes.'''

        self.in\_node\_1.center=(self.x, self.center\_y+26)

        self.in\_node\_2.center=(self.x, self.center\_y-26)

        self.out\_node.center=(self.right, self.center\_y+2)

    def get\_node\_collide(self, touch):

        '''Returns node index if the user has clicked a node.'''

        for node in self.nodes:

            if node.collide\_point(\*touch.pos):

                if node == self.in\_node\_1:

                    return 1

                elif node == self.in\_node\_2:

                    return 2

                elif node == self.out\_node:

                    return -1

        return False

    def select\_node(self, node\_index):

        '''Selects the given node.'''

        node = self.get\_node(node\_index)

        node.color = (0.3, 0.7, 0.7, 1)

    def deselect\_node(self, node\_index):

        '''Deselects the given node.'''

        node = self.get\_node(node\_index)

        node.color = (1, 1, 1, 1)

    def deselect\_nodes(self):

        '''Deselects all node of component.'''

        for node in self.nodes:

            node.color = (1, 1, 1, 1)

    def get\_node(self, index):

        '''Returns the node object for given node index.'''

        if index == 1:

            return self.in\_node\_1

        if index == 2:

            return self.in\_node\_2

        if index == -1:

            return self.out\_node

    def on\_pos(self, \*args, \*\*kwargs):

        '''Listens for change of kivy pos property. Updates position of image, nodes and selection border. Also tells board to update position of connection lines.'''

        try:

            self.border.rounded\_rectangle = (self.x, self.y, self.width, self.height, 10)

            self.img.pos = self.pos

            self.update\_nodes()

            self.parent.update\_connection\_lines()

        except AttributeError as e:

            print("Gate not finished initalising", e)

    def is\_selected(self):

        '''Returns True if component is currently selected.'''

        return self.selected

    def show\_nodes(self):

        '''Shows the components nodes.'''

        for node in self.nodes:

            node.opacity = 1

            node.disabled = False

        self.update\_nodes()

    def hide\_nodes(self):

        '''Hides the components nodes.'''

        for node in self.nodes:

            node.opacity = 0

            node.disabled = True

    def select(self):

        '''Selects the component.'''

        self.selected = True

        self.border.width = 2

    def deselect(self):

        '''Deselects the component'''

        self.selected = False

        self.border.width = 0.0001

    def on\_touch\_down(self, touch):

        '''Checks if user has clicked node, sets dragged attribute to false.'''

        if self.collide\_point(\*touch.pos):

            self.dragged = False

        return super().on\_touch\_down(touch)

    def on\_touch\_move(self, touch):

        '''Checks if the user drags the component when clicked. Sets dragged attribute to true, selectes component updates position if user drags within the drag\_timeout time and a distance greater than the drag\_distance distance.'''

        if touch.grab\_current is not self:

            return super().on\_touch\_move(touch)

        self.select()

        self.dragged = True

        if self.x < self.parent.x+5:

            self.x = self.parent.x+5

        elif self.y < self.parent.y+5:

            self.y = self.parent.y+5

        elif self.right > self.parent.right-5:

            self.right = self.parent.right-5

        elif self.top > self.parent.top-5:

            self.top = self.parent.top-5

        else:

            return super().on\_touch\_move(touch)

    def on\_touch\_up(self, touch):

        '''When user releases click, checks dragged attribute, if true component is deselected, if false and the component is already selected the component is also deselected, if both false component is selected.'''

        if not self.collide\_point(\*touch.pos):

            return super().on\_touch\_up(touch)

        if self.dragged:

            self.deselect()

        elif self.selected:

            self.deselect()

        else:

            self.select()

        return super().on\_touch\_up(touch)

    def get\_logic\_gate(self):

        '''Returns the logic component object of this visual component object.'''

        return self.logic\_gate

    def update\_state(self):

        '''Updates the state of component to match state of logic component.'''

        x = self.logic\_gate.get\_output()

        self.state = x

    def get\_state(self):

        '''Returns the components state.'''

        return self.state

class DragSwitch(DragGate):

    '''Switch visual component. User sets state of component'''

    def \_\_init\_\_(self, \*\*kwargs):

        super().\_\_init\_\_(\*\*kwargs)

        self.logic\_gate = Switch()

        self.states = {1:"GateIcons/switch\_on.png", 0:"GateIcons/switch\_off.png"}

        self.img.source = self.states[self.logic\_gate.get\_output()]

    def nodes\_init(self):

        '''Initialises the nodes of the component, positions them and adds them to the component widget.'''

        node\_source="GateIcons/node.png"

        self.out\_node = Image(source=node\_source, size\_hint=(None, None), size=(20,20))

        self.add\_widget(self.out\_node)

        self.nodes.append(self.out\_node)

        self.hide\_nodes()

    def update\_nodes(self):

        '''Updates the positions of the components nodes.'''

        self.out\_node.center=(self.right, self.y+(self.height//2))

    def get\_node\_collide(self, touch):

        '''Returns node index if the user has clicked a node.'''

        for node in self.nodes:

            if node.collide\_point(\*touch.pos):

                if node == self.out\_node:

                    return -1

        return False

    def on\_touch\_up(self, touch):

        '''When user releases click, checks dragged attribute, if false and the touch is within the set area, the switch is flipped and dragged is set as true, then super of method is called.'''

        if not self.dragged:

            if (touch.pos[0] < self.right - 30) and (touch.pos[0] > self.x + 10) and (touch.pos[1] < self.top - 20) and (touch.pos[1] > self.y + 20):

                self.logic\_gate.flip()

                self.update\_state()

                self.parent.update\_states()

                self.dragged = True

        return super().on\_touch\_up(touch)

    def update\_state(self):

        '''Updates the state of component to match state of logic component.'''

        x = self.logic\_gate.get\_output()

        self.state = x

        self.img.source = self.states[self.state]

class DragClock(DragGate):

    '''Clock visual component. Kivy scheduler flips state after an interval. (Uses switch logic component.'''

    def \_\_init\_\_(self, \*\*kwargs):

        super().\_\_init\_\_(\*\*kwargs)

        self.logic\_gate = Switch()

        self.states = {1:"GateIcons/clock\_on.png", 0:"GateIcons/clock\_off.png"}

        self.img.source = self.states[self.logic\_gate.get\_output()]

        Clock.schedule\_interval(self.clock\_flip, 1)

    def clock\_flip(self, \*args):

        '''Flips state of logic component, if AttributeError, the clock has been deleted so it stop the kivy schedule.'''

        self.logic\_gate.flip()

        self.update\_state()

        try:

            self.parent.update\_states()

        except AttributeError as e:

            print("Clock must have been deleted")

            Clock.unschedule(self.clock\_flip)

    def nodes\_init(self):

        '''Initialises the nodes of the component, positions them and adds them to the component widget.'''

        node\_source="GateIcons/node.png"

        self.out\_node = Image(source=node\_source, size\_hint=(None, None), size=(20,20))

        self.add\_widget(self.out\_node)

        self.nodes.append(self.out\_node)

        self.hide\_nodes()

    def update\_nodes(self):

        '''Updates the positions of the components nodes.'''

        self.out\_node.center=(self.right, self.y+(self.height//2))

    def get\_node\_collide(self, touch):

        '''Returns node index if the user has clicked a node.'''

        for node in self.nodes:

            if node.collide\_point(\*touch.pos):

                if node == self.out\_node:

                    return -1

        return False

    def update\_state(self):

        '''Updates the state of component to match state of logic component.'''

        x = self.logic\_gate.get\_output()

        self.state = x

        self.img.source = self.states[self.state]

class DragOutput(DragGate):

    '''Output visual component. Displays its state.'''

    def \_\_init\_\_(self, \*\*kwargs):

        super().\_\_init\_\_(\*\*kwargs)

        self.logic\_gate = Output()

        self.states = {1:"GateIcons/output\_on.png", 0:"GateIcons/output\_off.png", None:"GateIcons/output\_empty.png"}

        self.update\_state()

    def nodes\_init(self):

        '''Initialises the nodes of the component, positions them and adds them to the component widget.'''

        node\_source="GateIcons/node.png"

        self.in\_node\_1 = Image(source=node\_source, size\_hint=(None, None), size=(20,20))

        self.add\_widget(self.in\_node\_1)

        self.nodes.append(self.in\_node\_1)

        self.hide\_nodes()

    def update\_nodes(self):

        '''Updates the positions of the components nodes.'''

        self.in\_node\_1.center=(self.x, self.center\_y)

    def get\_node\_collide(self, touch):

        '''Returns node index if the user has clicked a node.'''

        for node in self.nodes:

            if node.collide\_point(\*touch.pos):

                if node == self.in\_node\_1:

                    return 1

        return False

    def update\_state(self):

        '''Updates the state of component to match state of logic component.'''

        x = self.logic\_gate.get\_output()

        self.state = x

        self.img.source = self.states[self.state]

class DragAndGate(DragGate):

    '''And Gate visual component.'''

    def \_\_init\_\_(self, \*\*kwargs):

        super().\_\_init\_\_(\*\*kwargs)

        self.img.source = "GateIcons/and.png"

        self.logic\_gate = And\_Gate()

class DragOrGate(DragGate):

    '''And Or visual component.'''

    def \_\_init\_\_(self, \*\*kwargs):

        super().\_\_init\_\_(\*\*kwargs)

        self.img.source = "GateIcons/or.png"

        self.logic\_gate = Or\_Gate()

class DragXorGate(DragGate):

    '''And Xor visual component.'''

    def \_\_init\_\_(self, \*\*kwargs):

        super().\_\_init\_\_(\*\*kwargs)

        self.img.source = "GateIcons/xor.png"

        self.logic\_gate = Xor\_Gate()

class DragNotGate(DragGate):

    '''Not Gate visual component.'''

    def \_\_init\_\_(self, \*\*kwargs):

        super().\_\_init\_\_(\*\*kwargs)

        self.img.source = "GateIcons/not.png"

        self.logic\_gate = Not\_Gate()

    def nodes\_init(self):

        '''Initialises the nodes of the component, positions them and adds them to the component widget.'''

        node\_source="GateIcons/node.png"

        self.in\_node\_1 = Image(source=node\_source, size\_hint=(None, None), size=(20,20))

        self.out\_node = Image(source=node\_source, size\_hint=(None, None), size=(20,20))

        self.add\_widget(self.in\_node\_1)

        self.add\_widget(self.out\_node)

        self.nodes.append(self.in\_node\_1)

        self.nodes.append(self.out\_node)

        self.hide\_nodes()

    def update\_nodes(self):

        '''Updates the positions of the components nodes.'''

        self.in\_node\_1.center=(self.x, self.center\_y+2)

        self.out\_node.center=(self.right, self.center\_y+2)

    def get\_node\_collide(self, touch):

        '''Returns node index if the user has clicked a node.'''

        for node in self.nodes:

            if node.collide\_point(\*touch.pos):

                if node == self.in\_node\_1:

                    return 1

                elif node == self.out\_node:

                    return -1

        return False

class MainWindow(Widget):

    '''Main Window class'''

    def \_\_init\_\_(self, \*\*kwargs):

        super().\_\_init\_\_(\*\*kwargs)

        self.ids["gateCanvas"].root = self

        self.toggles = ["connectToggle", "moveToggle", "disconnectToggle"]

    def set\_tool(self, tool):

        '''Calls Gate Canvases set\_tool method.'''

        self.ids["gateCanvas"].set\_tool(tool)

        toggle = f"{tool}Toggle"

        if self.ids[toggle].state == 'down':

            pass

        else:

            for tog in self.toggles:

                if tog == toggle:

                    self.ids[tog].state = 'down'

                else:

                    self.ids[tog].state = 'normal'

    def clear\_canvas(self):

        '''Calls Gate Canvases clear\_canvas method.'''

        self.ids["gateCanvas"].clear\_canvas()

        self.set\_tool("move")

    def add\_gate(self, gate\_type):

        '''Calls Gate Canvases add\_gate method.'''

        self.ids["gateCanvas"].add\_gate(gate\_type)

    def delete\_gate(self):

        '''Calls Gate Canvases delete\_gate method.'''

        self.ids["gateCanvas"].delete\_gate()

        self.set\_tool("move")

kv = Builder.load\_file("LogicSim.kv")

'''Loads the kivy file'''

class LogicGateSimulator(App):

    '''Kivy App class.'''

    def build(self):

        '''Builds the kivy app.'''

        self.icon = "GateIcons/and.png"

        return MainWindow()

if \_\_name\_\_ == '\_\_main\_\_':

    LogicGateSimulator().run()

## GUI (kvlang)

#:kivy 2.0

#:import Factory kivy.factory.Factory

<ConnectionLine>:

    canvas:

        Color:

            rgba: self.color\_state

<ActionButton>:

    font\_size:15

    color: 0,0,0,1

    background\_normal: ''

    background\_color: 0.8,0.8,0.8,1

<GateButton@Button>:

    size\_hint: None, None

    used: False

    center: self.parent.center

    background\_normal: ''

    background\_color: 0.8,0.8,0.8,1

    canvas:

        Color:

            rgba: (0.9,0.2,0.2,1) if not self.used else (0,0,0,1)

        Line:

            width: 2

            rectangle: self.x, self.y, self.width, self.height

<ToolToggleButton@ToggleButton>:

    font\_size: 15

    color: 0,0,0,1

    background\_normal: ''

    background\_color: 0.8,0.8,0.8,1

    size\_hint\_y: None

    size: 100,100

    pos\_hint: {"center\_x":0.5}

    allow\_no\_selection: False

<ToolButton@Button>:

    font\_size: 15

    color: 0,0,0,1

    background\_normal: ''

    background\_color: 0.8,0.8,0.8,1

    size\_hint\_y: None

    size: 100,100

    pos\_hint: {"center\_x":0.5}

<GateLabel@Label>:

    font\_size: 25

    color: 1,1,1,1

<ExitPopup>:

    auto\_dismiss: True

    title\_color: 0,0,0,1

    title\_size: 18

    title: "Are you sure you want to quit?\nAny unsaved progress will be lost"

    size\_hint: None, None

    size: 350,150

    BoxLayout:

        Button:

            text: 'Cancel'

            on\_release: root.dismiss()

        Button:

            text: 'Quit'

            on\_release: root.close\_window()

<TruthPopup>:

    auto\_dismiss: True

    title\_color: 0,0,0,1

    title\_size: 18

    title: "Truth Table Generator"

    size\_hint: 0.6, 0.8

    BoxLayout:

        orientation: "vertical"

        padding: 2

        ScrollView:

            size\_hint: 1, 0.6

            do\_scroll\_x: False

            effect\_cls: "ScrollEffect"

            Label:

                id: truth\_label

                size\_hint\_y: None

                text\_size: self.width, None

                height: self.texture\_size[1]

                halign: "center"

                bold: True

                font\_name: root.FONT

                color: (0,0,0,1)

        GridLayout:

            spacing: 2

            padding: 2

            size\_hint: 1, None

            height: 50

            rows: 1

            columns: 3

            TextInput:

                id: truth\_input

                text: "Enter Boolean Expression"

                size\_hint: 1, 1

                multiline: False

                on\_text\_validate: root.generate()

            Button:

                text: 'Cancel'

                on\_release: root.dismiss()

                size\_hint: None, 1

                width: 100

            Button:

                text: 'Generate'

                on\_release: root.generate()

                size\_hint: None, 1

                width: 100

<GridLayout>:

    canvas.before:

        Color:

            rgba: (0.8,0.8,0.8,1)

        Rectangle:

            pos: self.pos

            size: self.size

<MyGridLayout@GridLayout>:

    padding: 2

    spacing: 2

    canvas.before:

        Color:

            rgba: (0.2,0.2,0.2,1)

        Rectangle:

            pos: self.pos

            size: self.size

<MainWindow>:

    GridLayout:

        size: root.width, root.height

        rows:2

        orientation: "tb-lr"

        ActionBar:

            height: 40

            pos\_hint: {"top":1}

            ActionView:

                spacing: 2

                padding: 2

                ActionPrevious:

                    with\_previous: False

                    font\_size: 40

                    markup: True

                    title: " [b]Logic Gate Simulator[/b]"

                    app\_icon: ""

                ActionButton:

                    text: "Truth Table"

                    on\_release: Factory.TruthPopup().open(root)

                ActionButton:

                    text: "Save"

                ActionButton:

                    text: "Load"

                ActionButton:

                    text: "Help"

                ActionButton:

                    text: "Quit"

                    on\_release: Factory.ExitPopup().open()

        MyGridLayout:

            size: self.parent.width, self.parent.height

            orientation: "tb-rl"

            cols: 3

            padding: 0

            MyGridLayout:

                orientation: "tb-lr"

                cols:1

                size\_hint: None, 1

                width: 100

                Label:

                    color: 1,1,1,1

                    markup: True

                    font\_size: 30

                    size\_hint\_y: None

                    text: "[b]Tools[/b]"

                ToolToggleButton:

                    id: connectToggle

                    group: "tools"

                    text: "Connect"

                    on\_press: root.set\_tool("connect")

                ToolToggleButton:

                    id: disconnectToggle

                    group: "tools"

                    text: "Disconnect"

                    on\_press: root.set\_tool("disconnect")

                ToolToggleButton:

                    id: moveToggle

                    state: "down"

                    group: "tools"

                    text: "Move"

                    on\_press: root.set\_tool("move")

                ToolButton:

                    text: "Delete"

                    on\_press: root.delete\_gate()

                ToolButton:

                    text: "Clear"

                    on\_press: root.clear\_canvas()

            GateCanvas:

                id: gateCanvas

                canvas.before:

                    Color:

                        rgba: (0.8,0.8,0.8,1)

                    Rectangle:

                        pos: map(lambda x: x+5, self.pos)

                        size: map(lambda x: x-10, self.size)

                canvas.after:

                    Color:

                        rgba: (0,0,0,1)

                    Line:

                        width: 4

                        rectangle: (self.x+4, self.y+4, self.width-8, self.height-8)

            ScrollView:

                size\_hint: None, 1

                width: 200

                effect\_cls: "ScrollEffect"

                MyGridLayout:

                    size\_hint\_y: None

                    height: 900

                    cols: 1

                    rows: 2

                    Label:

                        size\_hint: 1,0.1

                        markup: True

                        font\_size: 40

                        color: 1,1,1,1

                        text: "[b]Gates[/b]"

                    MyGridLayout:

                        cols: 2

                        spaceing: 1

                        padding: 1

                        size: self.parent.size

                        GateLabel:

                            text: "And"

                        GateButton:

                            id: And

                            on\_press:

                                root.add\_gate("and")

                                self.used = True

                            Image:

                                source: 'GateIcons/and.png'

                                width: self.parent.width \* 0.9

                                height: self.parent.height \* 0.9

                                center\_x: self.parent.center\_x

                                center\_y: self.parent.center\_y

                        GateLabel:

                            text: "Or"

                        GateButton:

                            id: Or

                            on\_press:

                                root.add\_gate("or")

                                self.used = True

                            Image:

                                source: 'GateIcons/or.png'

                                width: self.parent.width \* 0.9

                                height: self.parent.height \* 0.9

                                center\_x: self.parent.center\_x

                                center\_y: self.parent.center\_y

                        GateLabel:

                            text: "Not"

                        GateButton:

                            id: Not

                            on\_press:

                                root.add\_gate("not")

                                self.used = True

                            Image:

                                source: 'GateIcons/not.png'

                                width: self.parent.width \* 0.9

                                height: self.parent.height \* 0.9

                                center\_x: self.parent.center\_x

                                center\_y: self.parent.center\_y

                        GateLabel:

                            text: "Xor"

                        GateButton:

                            id: Xor

                            on\_press:

                                root.add\_gate("xor")

                                self.used = True

                            Image:

                                source: 'GateIcons/xor.png'

                                width: self.parent.width \* 0.9

                                height: self.parent.height \* 0.9

                                center\_x: self.parent.center\_x

                                center\_y: self.parent.center\_y

                        GateLabel:

                            text: "Switch"

                        GateButton:

                            id: Input

                            on\_press:

                                root.add\_gate("switch")

                                self.used = True

                            Image:

                                source: 'GateIcons/switch\_menu.png'

                                width: self.parent.width \* 0.9

                                height: self.parent.height \* 0.9

                                center\_x: self.parent.center\_x

                                center\_y: self.parent.center\_y

                        GateLabel:

                            text: "Output"

                        GateButton:

                            id: Output

                            on\_press:

                                root.add\_gate("output")

                                self.used = True

                            Image:

                                source: 'GateIcons/output\_menu.png'

                                width: self.parent.width \* 0.9

                                height: self.parent.height \* 0.9

                                center\_x: self.parent.center\_x

                                center\_y: self.parent.center\_y

                        GateLabel:

                            text: "Clock"

                        GateButton:

                            id: Clock

                            on\_press:

                                root.add\_gate("clock")

                                self.used = True

                            Image:

                                source: 'GateIcons/clock\_menu.png'

                                width: self.parent.width \* 0.9

                                height: self.parent.height \* 0.9

                                center\_x: self.parent.center\_x

                                center\_y: self.parent.center\_y