Two methods for counting the number of gates are presented. One that Bryan came up with and one I came up with:

Bryan’s version - Take the first open parenthesis and find its closing parenthesis. Then if a token hasn’t been made that has that as a string already, add that to the list of tokens. The number of tokens in the end should equal the number of gates used.

My Version - Find a matching set of parenthesis with no other parentheses inside it. Replace all occurrences of those matching parentheses and what’s inside them with a number that represents how many gates are found so far. At the end you will have a length one string that is the number of gates.

I will write a program for both of these and see if they return the same number for a number of test cases. The next step would be to write a method that generates the truthValues from a string input of the circuit. Then create a method that creates the new levels of circuits (using two inputs and check) using only strings rather than having them be individual objects. This will save space. I will make two methods, one that creates a HashMap at the end to determine the which logic has the smallest number of gates for a given truth value sequence, and another that creates the HashMap as you go and throws out values for things already found to save memory space . This should also be significantly faster to run. Finally I will compare the two HashMaps to see if the circuits found are the same in each case. If not, it should check the number of gates in each case and print out both circuits and the number of gates for that circuit.

I made Bryan’s version (along with several helper functions) and it seems to work well with the exception that it cannot tell the difference between (a.b) and (b.a) whereas with my version, it would be easy to take this into account. This problem should not come up with the way new levels are made. I will code my version later on. For now, I will move on to creating a method that can determine the truthTable produced by a given circuit in string form.

~~The replaceAll function was being glitchy so while making the method to find the truthTables I had to manually remove parentheses from the string after putting in the value derived from nor.~~

The replace function was used instead of replaceAll. The truthTable derivation was tested for three inputs and two inputs and proved to be functional for the test cases. Next step is to make the function that creates the different levels.

I was able to create a level finder using the same method as for the previous one but changing the object type from Circuit to String. It worked on two inputs. It required at least 4 levels to find all 16 inputs. The greatest number of gates needed is 5 for 0110 and the logic is (((a.0).(b.0)).(a.b)). The numbe of gates needed for each logic and its circuit are as followed:

1 gates used to build 1000 using this circuit: (a.b)

3 gates used to build 1011 using this circuit: (((a.b).a).0)

4 gates used to build 1110 using this circuit: (((a.0).(b.0)).0)

1 gates used to build 1010 using this circuit: (b.0)

2 gates used to build 0010 using this circuit: ((a.b).b)

0 gates used to build 0101 using this circuit: b

0 gates used to build 0011 using this circuit: a

0 gates used to build 0000 using this circuit: 0

3 gates used to build 1101 using this circuit: (((a.b).b).0)

5 gates used to build 0110 using this circuit: (((a.0).(b.0)).(a.b))

1 gates used to build 1100 using this circuit: (a.0)

2 gates used to build 0100 using this circuit: ((a.b).a)

2 gates used to build 0111 using this circuit: ((a.b).0)

3 gates used to build 0001 using this circuit: ((a.0).(b.0))

4 gates used to build 1001 using this circuit: (((a.b).a).((a.b).b))

3 gates used to build 1111 using this circuit: (((a.0).a).0)

There are still problems with making the 6th level even with two inputs. There is not enough memory to handle it. The number of items in a set is given by the following equations:

Or

Where n= number of levels and

For x inputs n1 = x+1

For three inputs, since we can only go up to the 5th level using this method, we can still only find 202/256 of the circuits. The next step is to make a method that doesn’t add repeats to levels as we move along. This will dramatically cut the amount of space used and time consumed.

I have created the method that will create a HashMap as the circits are built and will only add the circuit to the level if a smaller version of it has not yet been found. I also made a method that given the number of inputs and the number of levels to make for each method, will compare the two HashMaps produced and print out any differences. I expect it to run into problems at the same logics as problems were found yesterday. These are the highlighted cases in the 6.18.15\_TestCases document.

Using 2 inputs and 4 levels for both methods, the method with elimination was unable to find a circuit for 1111 while the other could. In fact, it is impossible to find a circuit for this using the elimination method because the minimum circuits were found for everything else and they won’t be replaced so nothing will be added to new levels. Actually, I forgot that finding a circuit for 1111 is not necessary.

I ran it with 5 levels for without elimination (the max) and with 20 levels for with elimination. **The method without elimination runs extremely slow.** I compared only the 202 gates that were produced. There were some cases were the circuits were different but used the same number of gates, and there were some cases where the version without elimination used less gates. But the strange thing is that there were cases were the version with elimination used less gates. This means that it is possible for something in a higher level to be a simpler form of something in a lower level. Of the differences, these were the summarized results

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Which was better | w/ Elimination | w/o Elimination | Tie | Total |
| How many cases | 6 | 7 | 20 | 33 |

Number of circuits made by no elimination: 202

Number of circuits made with elimination: 256

00000001

Version 1 used 6 to make ((((a.0).b).((a.0).(c.0))).((a.0).b))

Version 2 used 6 to make ((((a.0).(b.0)).0).(c.0))

BREAK

11100110

Version 1 used 8 to make ((((a.b).c).(b.0)).(((a.b).(b.c)).(a.b)))

Version 2 used 7 to make (((((a.b).b).c).((b.c).b)).0)

BREAK

10101110

Version 1 used 5 to make (((b.0).(c.0)).((c.0).a))

Version 2 used 5 to make ((((a.b).b).(c.0)).0)

BREAK

10101011

Version 1 used 4 to make (((c.0).a).((c.0).b))

Version 2 used 5 to make (((a.c).a).((b.c).b))

BREAK

00001001

Version 1 used 7 to make ((((a.0).b).((a.0).(c.0))).((c.0).b))

Version 2 used 7 to make ((((b.0).(c.0)).(b.c)).(a.0))

BREAK

11001101

Version 1 used 4 to make (((b.0).a).((b.0).c))

Version 2 used 5 to make (((a.b).a).((b.c).c))

BREAK

11110100

Version 1 used 5 to make (((a.0).c).((a.0).(b.0)))

Version 2 used 5 to make ((((b.c).b).(a.0)).0)

BREAK

11001110

Version 1 used 5 to make (((b.0).a).((b.0).(c.0)))

Version 2 used 5 to make ((((a.c).c).(b.0)).0)

BREAK

01101100

Version 1 used 5 to make ((((a.c).b).b).(((a.c).b).(a.c)))

Version 2 used 7 to make ((((a.b).c).c).((a.c).(b.0)))

BREAK

10111100

Version 1 used 8 to make ((((a.b).a).(b.0)).(((a.b).0).(a.c)))

Version 2 used 7 to make (((((a.c).a).b).((a.b).a)).0)

BREAK

01101110

Version 1 used 6 to make ((((a.b).c).c).(((a.b).c).(b.0)))

Version 2 used 6 to make ((((a.b).c).((a.c).b)).0)

BREAK

11011010

Version 1 used 8 to make ((((a.b).c).(a.0)).(((a.b).(a.c)).(a.b)))

Version 2 used 7 to make (((((a.b).a).c).((a.c).a)).0)

BREAK

10100111

Version 1 used 6 to make ((((a.c).(b.c)).(a.c)).((a.c).a))

Version 2 used 6 to make ((((a.0).(b.c)).(a.c)).0)

BREAK

11110010

Version 1 used 5 to make (((a.0).b).((a.0).(c.0)))

Version 2 used 5 to make ((((b.c).c).(a.0)).0)

BREAK

11000110

Version 1 used 8 to make ((((a.c).c).(b.0)).(((a.c).(b.c)).(a.c)))

Version 2 used 8 to make (((((a.b).b).c).((b.c).b)).((a.b).a))

BREAK

11110001

Version 1 used 4 to make (((a.0).b).((a.0).c))

Version 2 used 5 to make (((a.b).b).((a.c).c))

BREAK

10111010

Version 1 used 5 to make (((a.0).(c.0)).((c.0).b))

Version 2 used 5 to make ((((a.b).a).(c.0)).0)

BREAK

01101010

Version 1 used 5 to make ((((a.b).c).c).(((a.b).c).(a.b)))

Version 2 used 6 to make ((((a.b).c).c).((a.b).(c.0)))

BREAK

10000001

Version 1 used 7 to make ((((a.0).(b.0)).(a.c)).((b.0).c))

Version 2 used 8 to make ((((a.0).(b.0)).(a.c)).((a.b).c))

BREAK

00100001

Version 1 used 7 to make ((((a.0).(b.0)).((b.0).c)).((a.0).c))

Version 2 used 7 to make ((((a.0).(c.0)).(a.c)).(b.0))

BREAK

11011100

Version 1 used 5 to make (((a.0).(b.0)).((b.0).c))

Version 2 used 5 to make ((((a.c).a).(b.0)).0)

BREAK

11010010

Version 1 used 8 to make ((((a.c).0).(b.c)).(((b.c).c).(a.0)))

Version 2 used 8 to make (((((a.b).a).c).((a.c).a)).((a.b).b))

BREAK

10011011

Version 1 used 6 to make ((((a.c).0).(b.c)).((b.c).b))

Version 2 used 6 to make ((((a.c).(b.0)).(b.c)).0)

BREAK

10110111

Version 1 used 6 to make ((((a.b).(b.c)).(a.b)).(((a.b).(b.c)).(b.c)))

Version 2 used 6 to make ((((a.b).(b.c)).(a.c)).0)

BREAK

10011111

Version 1 used 6 to make ((((a.b).(a.c)).(a.b)).(((a.b).(a.c)).(a.c)))

Version 2 used 6 to make ((((a.b).(a.c)).(b.c)).0)

BREAK

10000110

Version 1 used 9 to make ((((a.b).b).(c.0)).(((a.c).(b.c)).(a.b)))

Version 2 used 8 to make (((((a.b).b).c).((b.c).b)).((b.c).a))

BREAK

10110100

Version 1 used 8 to make ((((a.b).0).(b.c)).(((b.c).b).(a.0)))

Version 2 used 8 to make (((((a.c).a).b).((a.b).a)).((a.c).c))

BREAK

01111010

Version 1 used 6 to make ((((a.b).c).c).(((a.b).c).(a.0)))

Version 2 used 6 to make ((((a.b).c).((a.c).a)).0)

BREAK

01111000

Version 1 used 5 to make ((((b.c).a).a).(((b.c).a).(b.c)))

Version 2 used 7 to make ((((a.b).c).c).((a.0).(b.c)))

BREAK

10011101

Version 1 used 6 to make ((((a.b).0).(b.c)).((b.c).c))

Version 2 used 6 to make ((((a.b).(c.0)).(b.c)).0)

BREAK

10010010

Version 1 used 9 to make ((((a.b).a).(c.0)).(((a.c).(b.c)).(a.b)))

Version 2 used 8 to make (((((a.b).a).c).((a.c).a)).((a.c).b))

BREAK

10010100

Version 1 used 9 to make ((((a.b).(a.c)).(b.c)).(((b.c).b).(a.0)))

Version 2 used 8 to make (((((a.c).a).b).((a.b).a)).((a.b).c))

BREAK

00000110

Version 1 used 6 to make ((((a.0).b).((a.0).c)).(b.c))

Version 2 used 6 to make ((((b.c).b).((b.c).c)).(a.0))

BREAK

I added to the method with elimination to keep the circuit in the list if it is the same size, but don’t add it to the dictionary. This severely decreases the speed of the program. The method with elimination gives an upper bound to the number of levels the method without elimination would need to find all the circuits and an upper bound to how many gates are needed for each truth set. This tells that at 6 levels, we will not need to make more levels to find a circuit for each truth set. If we do check higher levels, there is still the possibility that something will simplify and use less circuits than something in a previous level and give the same truth set.

The next step would be to find a way to selectively eliminate the circuits that we know will not be useful in later circuits.