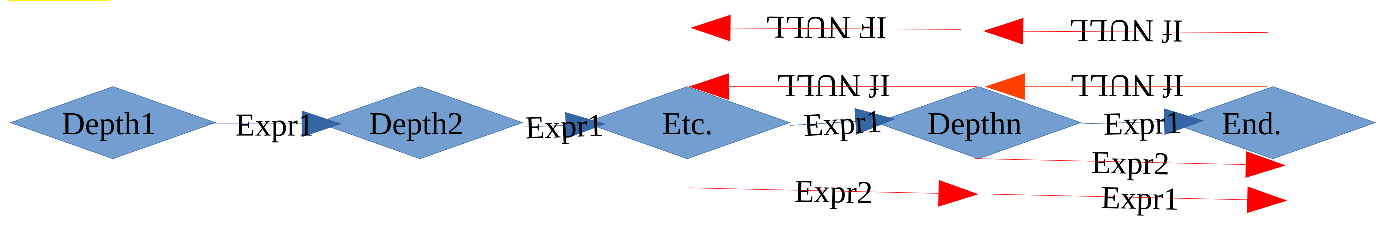
Part 1:
Design:
Four core functions are used to establish whole part 1, which are used to create the linked list, mapping, recursion and compare. The main design is like the logic:

Images



Since the graph shows the logic as follows:

Description

1. Get the initial values from find_derivations_for_strings() by reading from buffers and transmitting given line to Expr* type by using read_expr() function and passing it to find_final_result() function.
2. Since there might be many results during searching for rules, mapping the expression by using find_stage() function and store results into Depth* linked list.
3. Read through the result that the list given, recursively run the program again until the current depth is bigger than max_depth or the expression has tag "isTrue" in first node.
4. Free Depth and Expressions for each loop. Reset minimum number to -1.
5. Read next line and when all recursion finished, print the minimum value.

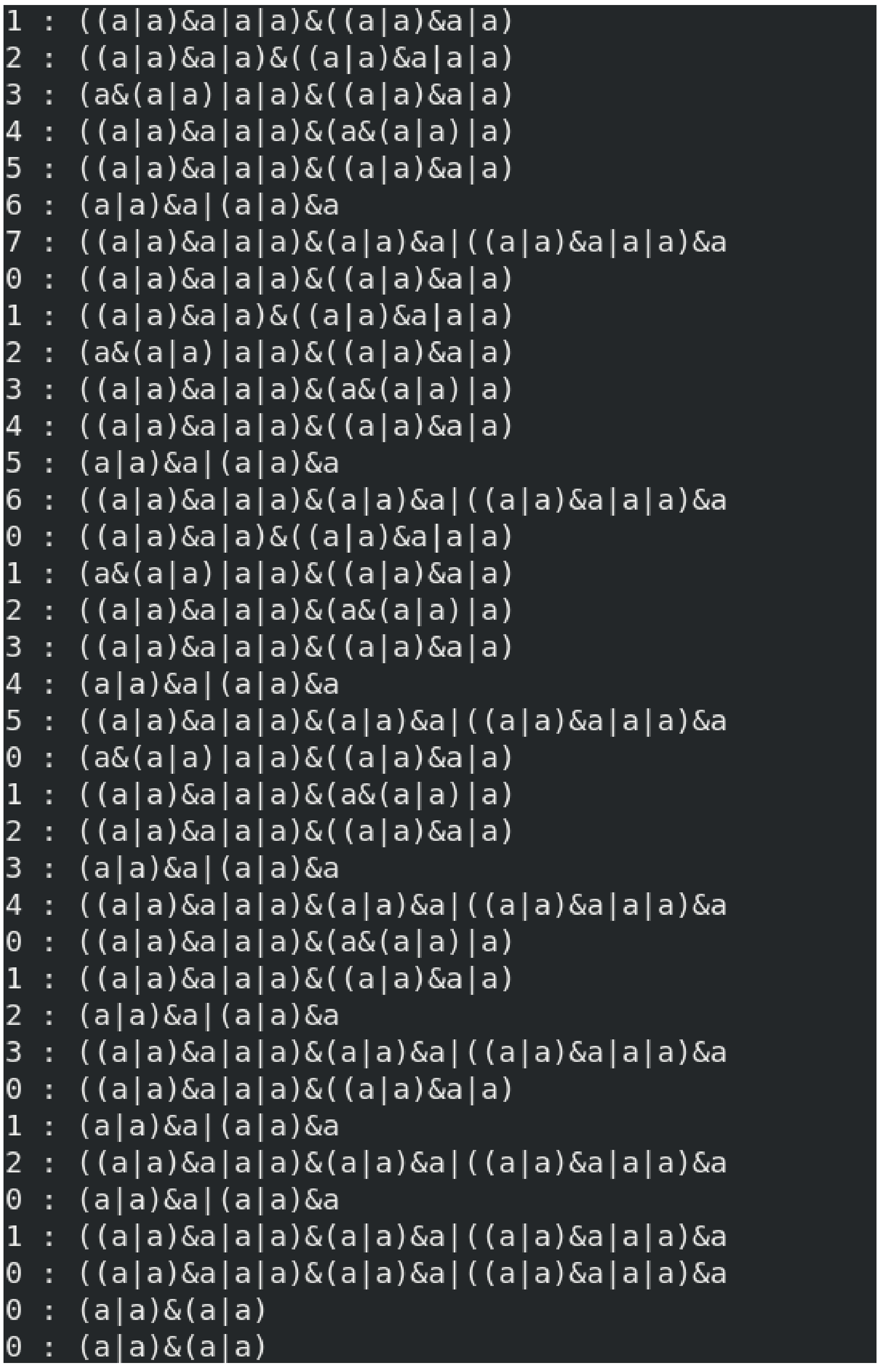
Testing mainly tests the following:
1. Testing whether the pointer points to the list: ln 107
      Result: since I assign ln 85 list_ptr = stage_list, the program does not work since I found that list_ptr assign to NULL instead of the list. So I assigned the list in line 99.

2. Check the Programming error and get some output by using check_stp() and check_ctn.
      Result: Since previously I thought when I assign expr in find_stage() to line 95, I can free the expression in line 111. However the program goes

wrong if I do so (free twice), by checking it I free all nodes in line 138 which prevents the condition happens again and free the initial expression in line 165.

3. Print all elements in Depth:

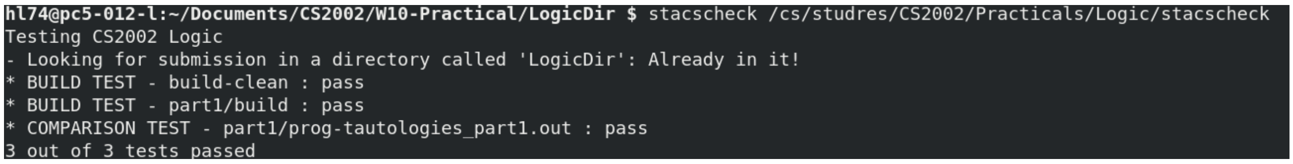
Result: This will be used to check the frame of Depth. The result will shows later. However this test also shows some disadvantages of this program: The program will always mapping the result, which leads to the longer time calculating and more complexity, Even if this will not cause heavy workload to the memory, but the program is lower than expected.



```
1 : ((a|a)&a|a|a)&((a|a)&a|a)
2 : ((a|a)&a|a)&((a|a)&a|a|a)
3 : (a&(a|a)|a|a)&((a|a)&a|a)
4 : ((a|a)&a|a|a)&(a&(a|a)|a)
5 : ((a|a)&a|a|a)&((a|a)&a|a)
6 : (a|a)&a|(a|a)&a
7 : ((a|a)&a|a|a)&(a|a)&a|((a|a)&a|a|a)&a
0 : ((a|a)&a|a|a)&((a|a)&a|a)
1 : ((a|a)&a|a)&((a|a)&a|a|a)
2 : (a&(a|a)|a|a)&((a|a)&a|a)
3 : ((a|a)&a|a|a)&(a&(a|a)|a)
4 : ((a|a)&a|a|a)&((a|a)&a|a)
5 : (a|a)&a|(a|a)&a
6 : ((a|a)&a|a|a)&(a|a)&a|((a|a)&a|a|a)&a
0 : ((a|a)&a|a)&((a|a)&a|a|a)
1 : (a&(a|a)|a|a)&((a|a)&a|a)
2 : ((a|a)&a|a|a)&(a&(a|a)|a)
3 : ((a|a)&a|a|a)&((a|a)&a|a)
4 : (a|a)&a|(a|a)&a
5 : ((a|a)&a|a|a)&(a|a)&a|((a|a)&a|a|a)&a
0 : (a&(a|a)|a|a)&((a|a)&a|a)
1 : ((a|a)&a|a|a)&(a&(a|a)|a)
2 : ((a|a)&a|a|a)&((a|a)&a|a)
3 : (a|a)&a|(a|a)&a
4 : ((a|a)&a|a|a)&(a|a)&a|((a|a)&a|a|a)&a
0 : ((a|a)&a|a|a)&(a&(a|a)|a)
1 : ((a|a)&a|a|a)&((a|a)&a|a)
2 : (a|a)&a|(a|a)&a
3 : ((a|a)&a|a|a)&(a|a)&a|((a|a)&a|a|a)&a
0 : ((a|a)&a|a|a)&((a|a)&a|a)
1 : (a|a)&a|(a|a)&a
2 : ((a|a)&a|a|a)&(a|a)&a|((a|a)&a|a|a)&a
0 : (a|a)&a|(a|a)&a
1 : ((a|a)&a|a|a)&(a|a)&a|((a|a)&a|a|a)&a
0 : ((a|a)&a|a|a)&(a|a)&a|((a|a)&a|a|a)&a
0 : (a|a)&(a|a)
0 : (a|a)&(a|a)
```

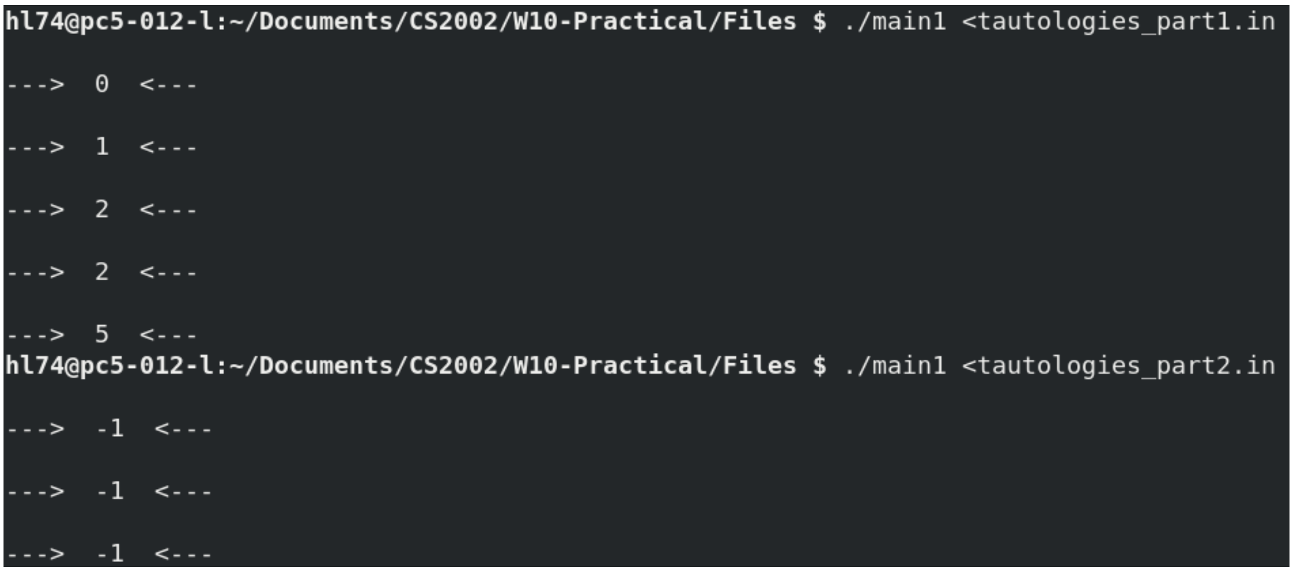
Implementation: I have reprogrammed this program four times, all the backups are stored in back-up folder. The main hardness during my programming is that I have to decide where to free expressions and depth and how can I choose function type to do the recursion. Finally I decide to set a global variable and use global variable to get the minimum values. Another hardness I was facing was that what should be included in my personalised struct variables, after trying put path (simp_2), undefined the variable(simp_3) finally I found that I can just put the last current depth as the depth and compare it with previous smallest number. These are how I implement my program.

Test:



```
hl74@pc5-012-l:~/Documents/CS2002/W10-Practical/LogicDir $ stacscheck /cs/studres/CS2002/Practicals/Logic/stacscheck
Testing CS2002 Logic
- Looking for submission in a directory called 'LogicDir': Already in it!
* BUILD TEST - build-clean : pass
* BUILD TEST - part1/build : pass
* COMPARISON TEST - part1/prog-tautologies_part1.out : pass
3 out of 3 tests passed
```

This is the stacscheck test. However I think the below appearance is more likely to be obvious combining with print_depth before head.



```
hl74@pc5-012-l:~/Documents/CS2002/W10-Practical/Files $ ./main1 <tautologies_part1.in

--->  0  <---

--->  1  <---

--->  2  <---

--->  2  <---

--->  5  <---
hl74@pc5-012-l:~/Documents/CS2002/W10-Practical/Files $ ./main1 <tautologies_part2.in

--->  -1  <---

--->  -1  <---

--->  -1  <---
```

Critical evaluation:
The frame is clear enough but the program has some duplicates. Since the program will map all child nodes which will spend most of time by doing that, such as there are n laws and the map_depth is 6, then there will be n^6 to get the result. By dealing with that, the easiest way is that put some limits in it such as there should not be two same laws happens two times and there should not be laws that do forward first and then do backward next, however this will depends on the order of apply and search function and have to do lots of maths. Despite this the program should be okay.

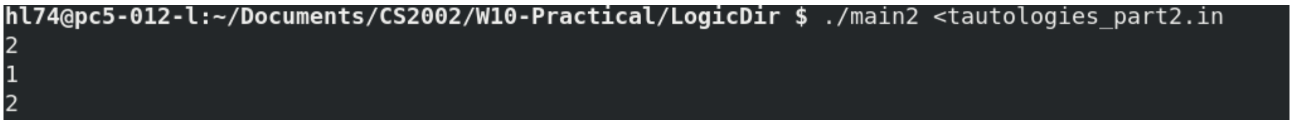
Part 2:
From the axioms before, since when we are calculating T|A, ¬F and ¬¬T, the possible and necessary steps are:
T|A → (A|¬A)|A → A|(¬A|A) , hence there is no axioms that applies this such that it will return -1.
¬F → A&¬A → ¬A&A is the same. So these will always return -1.

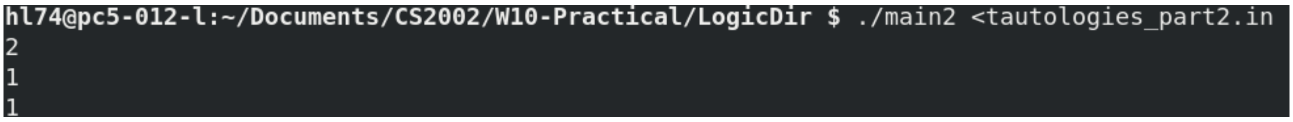
The easiest way to deal with it is apply identity rule, dominant rule and negation of constant rule.
Dominant and Identity rule are axioms that calculating T and variables, which showing that T and any variables are variable itself, T or any variables are T; F and any variables are F, F or any variables are variables. Negation rule can transfer ¬T to F, which can easily solve the issue. As a result, negation constant rule, identity rule and Dominant rule are added to addition law as dominant can only apply forward but cannot apply backward: A | T = T, A & F = F, we cannot get A from right side to left side.

Question e> Apart from these equations and laws, we still cannot solve the equation in De Morgan laws ¬(A ∨ B) = ¬A ∧ ¬B, ¬(A ∧ B) = ¬A ∨ ¬B:
Since this will find A V B but this can only apply commutativity law, we cannot break the bracket. DeMorgan law applies logical negative to distributivity, to solve the question, De Morgan law is in need.

Double negation ¬¬A can not be solved (but ¬¬T can). However double negation only needs forward since we are rarely use A = ¬¬A.

Identity shows that one element conj or disj to itself will get itself. Since Identity backward will severely increase the depth of the tree such as:
a → a|a → (a|a)|(a|a) …
So I only note the forward function in the program.

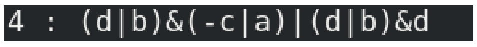
Testing:
By applying the law described before, the result shows as follows:

```
hl74@pc5-012-l:~/Documents/CS2002/W10-Practical/LogicDir $ ./main2 <tautologies_part2.in
2
1
2
```
which obviously successfully solved the question. After applying double negation, the result is like follows:



```
hl74@pc5-012-l:~/Documents/CS2002/W10-Practical/LogicDir $ ./main2 <tautologies_part2.in
2
1
1
```

Both two success, but one step is not worth ten lines of code. So if we do not trying to solve double negation of a variable, double negation is not in need.

Part 3:
The main testing equation is from lecture note:
(a ∨ ¬c ∨ d) ∧ (b ∨ d)
Firstly transmitting it into program readable equations:
(a|-c|d)&(b|d)
Since there are three variables, to get T, we need to transmit these variables into CNF, then calculate to get the result. To transmit variables into T and F, Before choosing what logic should be used. First try to run it through part 2, wants to get depth that this equation will be used.
However, the program takes time so long, so this step failed.
Since what I described before, to get true and false we need complementation, domination and negation of constant; d is the only variable that appeared twice, so we have to move these two d together by using distributivity and commutativity.
By looking at the output of each steps, I found there is one output like this:



```
4 : (d|b)&(-c|a)|(d|b)&d
```

So I also added the De Morgan law to the program.

Unfortunately, the program takes me too much times and there is no enough time for run the program, but I guess this should be goes properly.
As a result, a cnf law needs distribution, commutative, identity, dominating and associativity at least. Run from CNF to T needs complementation, negation constant and domination.