COMS10015 lab. worksheet #1: Boolean algebra

§1. C-class, or core questions

▷ **S1[C].** There is no associated solution for this question, because it is essentially a guided explanation rather than a task or question per se.

▷ **S2[C].** Producing an automated solution to both questions is basically done in the same way: the central ingredient is the `simplify_logic` function, documented here

https://docs.sympy.org/latest/modules/logic.html#sympy.logic.boolalg.simplify_logic

Step-by-step then, each SymPy-produced solution will 1) import the required functionality, 2) introduce some SymPy symbols, e.g., `a` to `c` to represent the variables a to c, 3) define some Boolean functions, e.g., `f` and/or `g`, to match the question, 4) use `simplify_logic` to simplify `f`, then 5) compare the automated (on the LHS) and manual (on the RHS) solutions with each other: if the result of evaluation is `True` (resp. `False`), this indicates their equivalence (resp. non-equivalence). As an aside, you might wonder why the final step does not use the == operator to test equivalence of the LHS and RHS. The reason is that it actually tests *equality* (i.e., whether the LHS and RHS are syntactically the same expressions) not *equivalence* (i.e., whether the LHS and RHS are symbolically equivalent expressions); a more convoluted approach is required, therefore.

- We can attempt to solve this challenge manually, e.g., via

$$
\begin{array}{rlllll}
 & (a \wedge b \wedge c) & \vee & (\neg a \wedge b) & \vee & (a \wedge b \wedge \neg c) \\
= & (a \wedge b \wedge c) & \vee & (a \wedge b \wedge \neg c) & \vee & (\neg a \wedge b) \qquad \text{(commutativity)} \\
= & (a \wedge b) & \wedge & (c \vee \neg c) & \vee & (\neg a \wedge b) \qquad \text{(distribution)} \\
= & (a \wedge b) & \wedge & 1 & \vee & (\neg a \wedge b) \qquad \text{(inverse)} \\
= & (a \wedge b) & & & \vee & (\neg a \wedge b) \qquad \text{(identity)} \\
= & b & \wedge & (a \vee \neg a) & & \qquad \text{(distribution)} \\
= & b & \wedge & 1 & & \qquad \text{(inverse)} \\
= & b & & & & \qquad \text{(identity)}
\end{array}
$$

  which is fairly definitive in the sense there are zero operators! Using SymPy we can obtain an equivalent solution automatically, e.g., via

```
from sympy.logic.boolalg import *

a, b, c = symbols( 'a b c', boolean = True )

f = ( a & b & c ) | ( ~a & b ) | ( a & b & ~c )
r = simplify_logic( f )

isEquivalent = lambda lhs, rhs : simplify( ( lhs & ~rhs ) | ( ~lhs & rhs ) ) == False

print(               r      )
print( isEquivalent( r, b ) )
```

- We can attempt to solve this challenge manually, e.g., via

$$
\begin{array}{rlllll}
 & (\neg(a \vee b) & \wedge & \neg(c \vee d \vee e)) & \vee & \neg(a \vee b) \\
= & \neg(a \vee b) & \vee & (\neg(a \vee b) & \wedge & \neg(c \vee d \vee e)) \qquad \text{(commutativity)} \\
= & \neg(a \vee b) & & & & \qquad \text{(absorption)}
\end{array}
$$

  or

$$
\begin{array}{rlllll}
 & (\neg(a \vee b) & \wedge & \neg(c \vee d \vee e)) & \vee & \neg(a \vee b) \\
= & ((\neg a \wedge \neg b) & \wedge & \neg(c \vee d \vee e)) & \vee & \neg(a \vee b) \qquad \text{(de Morgan)} \\
= & ((\neg a \wedge \neg b) & \wedge & (\neg c \wedge \neg d \wedge \neg e)) & \vee & \neg(a \vee b) \qquad \text{(de Morgan)} \\
= & ((\neg a \wedge \neg b) & \wedge & (\neg c \wedge \neg d \wedge \neg e)) & \vee & (\neg a \wedge \neg b) \qquad \text{(de Morgan)} \\
= & (\neg a \wedge \neg b) & \vee & ((\neg a \wedge \neg b) & \wedge & (\neg c \wedge \neg d \wedge \neg e)) \qquad \text{(commutativity)} \\
= & (\neg a \wedge \neg b) & & & & \qquad \text{(absorption)} \\
= & \neg(a \vee b) & & & & \qquad \text{(de Morgan)}
\end{array}
$$

  where in either case we obtain $\neg(a \vee b)$ as a solution, and hence 2 operators as the minimum. Using SymPy we can obtain an equivalent solution automatically, e.g., via

```
from sympy.logic.boolalg import *

a, b, c, d, e = symbols( 'a b c d e', boolean = True )

f = ( ~( a | b ) & ~( c | d | e ) ) | ~( a | b )
r = simplify_logic( f )

isEquivalent = lambda lhs, rhs : simplify( ( lhs & ~rhs ) | ( ~lhs & rhs ) ) == False

print(               r               )
print( isEquivalent( r, ~( a | b ) ) )
```

Notice that in this case, the automatic solution differs from the manual solution: vs. the above, it should read

$$\neg a \wedge \neg b.$$

We *know* that

$$\neg a \wedge \neg b \quad \equiv \quad \neg(a \vee b)$$

because this is an instance of the de Morgans axiom, but *why* would SymPy prefer the former vs. the latter? The answer is that the former is a valid Sum of Products (SoP) or disjunctive normal form expression, whereas the latter is not: SymPy will, more than likely, manipulate expressions represented in a canonical form of this type.

§2. R-class, or revision questions

▷ **S3[R].** There is a set of solutions available at

https://assets.phoo.org/COMS10015_2025_TB-4/csdsp/sheet/misc-revision_s.pdf

§3. A-class, or additional questions

▷ **S4[A].** We can attempt to solve this challenge manually, e.g., via axiomatic manipulation

$$\begin{aligned}
f(a,b) &= (b \wedge \neg a) && \vee && (a \wedge \neg b) && \\
&= (b \wedge \neg a) \vee 0 && \vee && (a \wedge \neg b) \vee 0 && \text{(identity)} \\
&= (b \wedge \neg a) \vee (b \wedge \neg b) && \vee && (a \wedge \neg b) \vee (a \wedge \neg a) && \text{(inverse)} \\
&= (b \wedge (\neg a \vee \neg b)) && \vee && (a \wedge (\neg b \vee \neg a)) && \text{(distribution)} \\
&= ((\neg a \vee \neg b) \wedge b) && \vee && ((\neg a \vee \neg b) \wedge a) && \text{(commutativity)} \\
&= (\neg a \vee \neg b) \wedge (a \vee b) && && && \text{(distribution)} \\
&= (a \vee b) \wedge (\neg a \vee \neg b) && && && \text{(commutativity)} \\
&= (a \vee b) \wedge \neg(a \wedge b) && && && \text{(de Morgan)} \\
&= g(a,b) && && &&
\end{aligned}$$

or, alternatively, brute-force enumeration

a	b	$b \wedge \neg a$	$a \wedge \neg b$	$f(a,b)$	$a \vee b$	$\neg(a \wedge b)$	$g(a,b)$
0	0	0	0	0	0	1	0
0	1	1	0	1	1	1	1
1	0	0	1	1	1	1	1
1	1	0	0	0	1	0	0

To utilise SAT, the idea is to define

$$h(a,b) = f(a,b) \oplus g(a,b)$$

and use this as a SAT instance. The XOR is central: remember, it produces 1 as output if one and only one of the inputs are 1, and 0 otherwise. So what we are saying is that 1) if h is satisfiable, then an assignment to the variables st. the result of f and g differ was found (i.e., they are definitely not equivalent), whereas 2) if h is not satisfiable, then an assignment to the variables st. the result of f and g differ was not found (i.e., they are probably equivalent). Using SymPy, this approach is implemented as follows:

```
from sympy.logic.inference import *

a, b = symbols( 'a b', boolean = True )

f = ( b & ~a ) | ( a & ~b )
g = ( a | b ) & ~( a & b )
h = f ^ g

print( satisfiable( h ) )
```

An underlying point here is that *all* the approaches are valid, but, equally, may be advantageous or disadvantageous when used in a particular context. For example, use of axiomatic manipulation does not *tell* us anything if f and g are not equivalent. Using SAT does, however: if h is satisfiable then f and g are shown not to be equivalent, *plus* we get an example (or model, i.e., an assignment to the variables) that could, e.g., allow us to "debug" f or g and so identify where or why.