Combinational logic with 3-colouring

To k-colour a graph is to assign a colour to each of its vertices, using at most k colours altogether, in such a way that no two adjacent vertices have the same colour. It is well-known that it can be determined in polynomial time whether a given graph can be 2-coloured, but determining whether a given graph can be 3-coloured is an NP-complete problem. So unless $P = NP$, the ability to 3-colour graphs is in some sense greater than the ability to 2-colour them. To illustrate how large an ability 3-colouring is, we will show that if we can 3-colour graphs, then we can compute any boolean function.

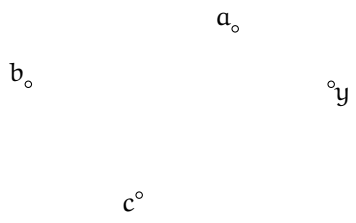
We will use the colours 0, 1, and Z. The colour 0 represents the boolean value "false"; the colour 1 represents the boolean value "true"; the colour Z has no boolean meaning. A boolean expression such as

$$a \wedge (\neg b \vee c)$$

will be implemented as a suitable 3-colourable graph. Three of that graph's vertices represent the variables a, b, and c; these are called the *input vertices*. One of its vertices represents the value of the expression; this is the *output vertex*. To evaluate the boolean expression for a given truth assignment to the variables, we first colour the input vertices 0 and 1 to show the desired truth assignment; then we 3-colour the graph (by some unspecified method); then we observe the colour of the output vertex, which will (due to the shape of the graph) be either 0 or 1, according to the value of the expression for the given truth assignment.

To make this work, we require not only that the graph be 3-colourable, but that the set of its 3-colourings have a certain shape. First, for every way to colour the input vertices with 0s and 1s, there must exist a 3-colouring of the graph which colours the input vertices that way. (Otherwise there are truth assignments for which we cannot evaluate our boolean expression.) Second, every 3-colouring of the graph in which the input vertices are assigned 0s and 1s must also assign the output vertex the appropriate colour, according to the value of the boolean expression for that truth assignment. (Otherwise our 3-colouring method might miscalculate the boolean expression.)

To construct such a graph for our example boolean expression $a \wedge (\neg b \vee c)$, we will begin with a completely disconnected graph containing just the input vertices a, b, c and an output vertex y.

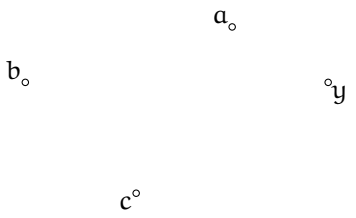


It is now, in a silly sense, easy to evaluate the boolean expression by 3-colouring this graph; once the input vertices a, b, c are assigned 0s and 1s, we need simply do this:

> 3-colour the graph, satisfying the constraint $y = a \wedge (\neg b \vee c)$.

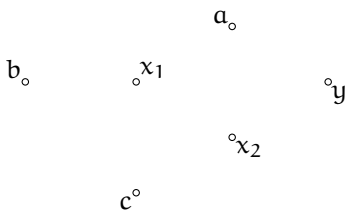
Our task is now to move the constraint into the graph, so that this instruction can be replaced by "3-colour the graph".

We will begin with the familiar step of breaking the constraint down into more primitive constraints, one for each subexpression. (This process resembles how a compiler might implement a large expression using temporary variables for intermediate results.) Thus from the current situation



Constraint: $y = a \wedge (\neg b \vee c)$

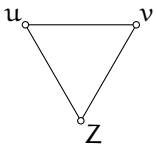
we pass to the situation



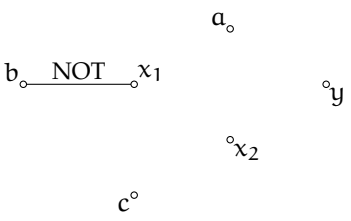
Constraints: $y = a \wedge x_2$, $x_2 = x_1 \vee c$, $x_1 = \neg b$

Next, we wish to move, for example, the constraint $x_1 = \neg b$ into the graph; to do so we connect the vertices x_1 and b with some gadget, which of course will be called "NOT". Note that it is not quite adequate just to join b and x_1 with an edge, since that would allow one or the other to be coloured Z, while we want them to be coloured with the boolean colours $0, 1$. But it is easy to see that a triangle will do the trick:

$u, v \in \{0, 1\}$
$u = \neg v$



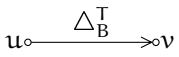
(The bottom vertex in this NOT gadget is always coloured Z.[1]) For simplicity in the following diagrams, the NOT gadget will be denoted simply as an edge with "NOT" written by it; thus we are now in this situation:



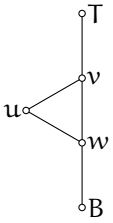
Constraints: $y = a \wedge x_2$
$x_2 = x_1 \vee c$

It remains only to implement two-argument boolean functions such as $\vee$ and $\wedge$. As is well-known, we can implement any such function using NOT and OR, so all we need is an OR gadget.

The OR gadget is rather complicated, so we'll build up to it. First consider this triangular gadget:



$(u = B \text{ and } v = X)$
$\quad \text{or } (u \neq B \text{ and } v = B)$



On the left is the abbreviated notation for the gadget, and below that, the constraint it implements;[2] on the right is the full implementation. The top vertex T and the bottom vertex B are constant vertices of different colours; the third colour is X. Note that, unlike in the NOT gadget, the vertices u and v here have different roles; thus the abbreviated notation has an arrow to show which is which.

[1] The ability to specify such constant vertices is necessary: 3-colouring is symmetrical under permutation of colours, but we wish to implement boolean functions such as AND and OR which are not symmetric under exchange of 0 and 1. Constant vertices are the simplest way to introduce colour asymmetry.

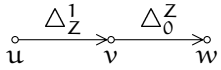
[2] Note that we write out "and" and "or" in the statement of the constraint instead of using the symbols $\wedge$ and $\vee$. We reserve $\wedge$ and $\vee$ for the functions on the colours 0 and 1, which are distinct from the boolean values "true" and "false". Of course the former represent the latter during the evaluation of boolean expressions via 3-colouring — that's the whole point — but identifying them introduces many opportunities for confusions and subtle errors.

Now, to show that this gadget implements the constraint stated, we must show two things: first, that if vertices u and v are connected as shown and the graph is 3-colourable, then the constraint is satisfied; second, that if the constraint is satisfied, then the rest of this gadget is 3-colourable. (It may well be that the graph as a whole is not 3-colourable, or is only 3-colourable under additional contraints on u and v; but we are concerned now only with the effect of this gadget.)

So suppose the gadget is 3-colourable. Since u, v, w are all adjacent, they represent all three colours T, B, and X. If $u = B$ then v is adjacent to B and to T, so $v = X$. If $u \neq B$, then either $v = B$ or $w = B$; we cannot have $w = B$, so $v = B$. Thus the stated constraint is satisfied.

Now suppose the constraint is satisfied. If $u = B$ and $v = X$ then we can complete the colouring of the gadget by taking $w = T$; if $u \neq B$ and $v = B$ then we can complete the colouring of the gadget by taking w to be either T or X, whichever differs from u. Thus the gadget is 3-colourable.

This completes the proof that $\triangle_B^T$ works as advertised. Putting two of them together yields our next gadget, which in a sense tests whether a specified vertex is coloured Z:[3]



$(u = Z \text{ and } w = 1)$
$\quad$ or $(u \neq Z \text{ and } w = 0)$

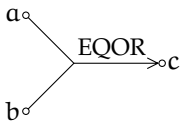
The two $\triangle$ gadgets together enforce the constraint

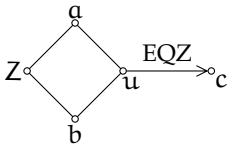
$$
\begin{aligned}
&((u = Z \text{ and } v = 0) \text{ or } (u \neq Z \text{ and } v = Z)) \\
&\text{and } ((v = 0 \text{ and } w = 1) \text{ or } (v \neq 0 \text{ and } w = 0)) \\
\equiv\ & \{\text{and distributes over or}\} \\
&(u = Z \text{ and } v = 0 \text{ and } v = 0 \text{ and } w = 1) \\
&\text{or } (u = Z \text{ and } v = 0 \text{ and } v \neq 0 \text{ and } w = 0) \\
&\text{or } (u \neq Z \text{ and } v = Z \text{ and } v = 0 \text{ and } w = 1) \\
&\text{or } (u \neq Z \text{ and } v = Z \text{ and } v \neq 0 \text{ and } w = 0) \\
\equiv\ & \{\text{second and third clauses contain contradictions on } v\} \\
&(u = Z \text{ and } v = 0 \text{ and } v = 0 \text{ and } w = 1) \\
&\text{or } (u \neq Z \text{ and } v = Z \text{ and } v \neq 0 \text{ and } w = 0) \\
\equiv\ & \{v = 0 \Rightarrow v = 0; \text{ also, } v = Z \Rightarrow v \neq 0\} \\
&(u = Z \text{ and } v = 0 \text{ and } w = 1) \text{ or } (u \neq Z \text{ and } v = Z \text{ and } w = 0)
\end{aligned}
$$

[3] Note again that writing the constraint in the simpler form "$w = (u = Z)$" would conflate the boolean type of the expression "$u = Z$" with the colour type of the expression "w".

The conditions on v are certainly satisfiable; removing them yields the condition claimed for EQZ.

With EQZ we can build EQOR:

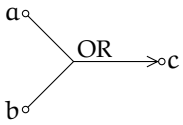




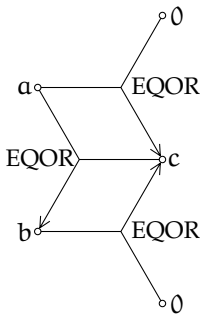
$a, b, c \in \{0, 1\}$
$a = b$ or $c = 1$

We have the constraint $a, b \in \{0, 1\}$ due to the Z at the left, and the constraint $c \in \{0, 1\}$ because c is at the endpoint of EQZ, which only allows these values there. By the operation of EQZ there are then two cases: first, $u = Z$ and $c = 1$, in which case a and b are unconstrained; second, $u \neq Z$ and $c = 0$, in which case the gadget is 3-colourable exactly when $a = b = \neg u$. Thus this gadget implements the constraint stated.

Three EQORs yield, finally, an OR gadget:



$a, b, c \in \{0, 1\}$
$c \equiv a \vee b$



The top EQOR enforces that either $a = 0$ or $c = 1$, that is, $a \Rightarrow c$. Similarly the bottom one enforces $b \Rightarrow c$; together they enforce $a \vee b \Rightarrow c$. The left EQOR enforces that either $b = c$ or $a = 1$, which since $a, b, c \in \{0, 1\}$ means

$$
\begin{aligned}
& (b \equiv c) \vee a \\
\equiv \quad & \{\text{bottom EQOR establishes } b \Rightarrow c\} \\
& (c \Rightarrow b) \vee a \\
\equiv \quad & \{(p \Rightarrow q) \equiv (\neg p \vee q), \text{ with } p, q := c, b\} \\
& \neg c \vee b \vee a \\
\equiv \quad & \{\text{the same, with } p, q := c, a \vee b\} \\
& c \Rightarrow a \vee b
\end{aligned}
$$

Thus the three EQORs together enforce $c \equiv a \vee b$, as desired.