CSCC63 Assignment 2

Reductions, Polytime Reductions, and $\mathcal{NP}$

Due 11:59pm, July 7

Warning: *For this assignment you may work either alone or in pairs. Your electronic submission of a PDF to Crowdmark affirms that this assignment is your own work and that of your partner, and no one else's, and is also in accordance with the University of Toronto Code of Behaviour on Academic Matters, the Code of Student Conduct, and the guidelines for avoiding plagiarism in CSCC63. Note that using Google or any other online resource is considered plagiarism.*

1. (10 marks) Consider the language

$$L_1' = \left\{ \langle G_1 = (V, \Sigma, P, S) \rangle \,\middle|\, G_1 \text{ is a CFGs and there are strings } w, w' \in \Sigma^* \text{ such that } w, w', \text{ and } ww' \in L(G_1) \right\}.$$

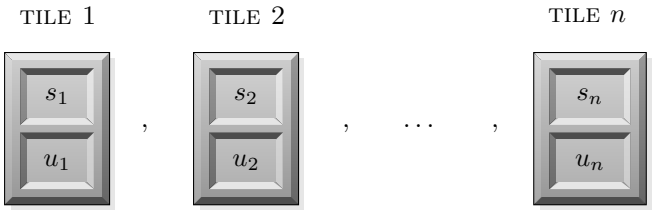
Show that L_1' is undecidable.

Note that any CFG G and string x, it is possible to decide whether $x \in L(G)$ in $\mathcal{O}(|G|^2|x|^3)$ time.

Solution:

We'll follow the example from class nd reduce from the PCP.

With this in mind, suppose we are given a PCP instance



Note that s_i is the string on the top of the i^{th} tile, and u_i is the string on the bottom.

We'll build the grammar G_1 using the alphabet of the strings s_i and u_i, along with:

- The n extra characters $t_1, \ldots, t_n$, and
- One extra character #.

Here is the grammars G_1:

$$\begin{aligned} G_1: \quad & S \to |S_1\#|\#S_2\#\#S_2\# \\ \\ & S_1 \to s_1A_1t_1 \,|\, s_2A_1t_2 \,|\, \ldots \,|\, s_nA_1t_n \\ & A_1 \to \# \,|\, s_1A_1t_1 \,|\, s_2A_1t_2 \,|\, \ldots \,|\, s_nA_1t_n \\ \\ & S_2 \to u_1A_2t_1 \,|\, u_2A_2t_2 \,|\, \ldots \,|\, u_nA_2t_n \\ & A_2 \to \# \,|\, u_1A_2t_1 \,|\, u_2A_2t_2 \,|\, \ldots \,|\, u_nA_2t_n \end{aligned}$$

The t_i and the # characters are all new characters that do not occur in any of the s_is or t_is.

Suppose that our PCP instance has match:
Then there is some tile sequence $t_{i_1}, t_{i_2}, \ldots, t_{i_k}$ of tiles such that $s_{i_1}s_{i_2}\ldots s_{i_k} = u_{i_1}u_{i_2}\ldots u_{i_k}$.
If we use this tile sequence to generate an S_1, we get the string

$$w = w' = \#s_{i_1}s_{i_2}\ldots s_{i_k}\#\#t_{i_k}\ldots t_{i_2}t_{i_1}\#.$$

On the other hand, we can also use S_2 to generate the string

$$w = \#u_{i_1}u_{i_2}\ldots u_{i_k}\#\#t_{i_k}\ldots t_{i_2}t_{i_1}\#.$$

This means that we can use the $S \to \#S_2\#\#S_2\#$ rule to generate the string

$$\#u_{i_1}u_{i_2}\ldots u_{i_k}\#\#t_{i_k}\ldots t_{i_2}t_{i_1}\#\#u_{i_1}u_{i_2}\ldots u_{i_k}\#\#t_{i_k}\ldots t_{i_2}t_{i_1}\# = ww'.$$

$\Rightarrow \langle G_1\rangle \in L_1'$.
Suppose that our $\langle G_1\rangle \in L_1'$:
Then there are strings w and w' such that G_1 generates w, $'$, and ww'.
Now, we can see that the strings generated by S come in two forms:

- If we expand S using the rule $S \to \#S_1\#$, then we have a string of the form

  $$\#s_{i_1}s_{i_2}\ldots s_{i_k}\#t_{i_1}t_{i_2}\ldots t_{i_k}\#.$$

  This string contains exactly three "#" characters. Note that the only characters between the final two #s are "t_i"s, and that there is no such character between the first two "#"s.
- If we expand S using the rule $S \to \#S_2\#\#S_2\#$, then we have a string of the form

  $$\#u_{i_1}u_{i_2}\ldots u_{i_k}\#t_{i_1}t_{i_2}\ldots t_{i_k}\#\#u_{i'_1}u_{i'_2}\ldots u_{i'_{k'}}\#t_{i'_1}t_{i'_2}\ldots t_{i'_k}\#.$$

  This string contains exactly six "#" characters. Note that the t_i characters are exactly the characters occuring between the second pair of "#"s and final pair of "#"s.

This means that the only way for G_1 to generate w, w', and ww' is if both w and w' are of the form $\#S_1\#$, while ww' is of the form $\#S_2\#\#S_2\#$.
In particular, we see that w must match the first $\#S_2\#$, while w' must match the second $\#S_2\#$.
Now, if w can be generated by both $\#S_1\#$ and $\#S_2\#$, then it can be expressed as both

$$\#s_{i_1}s_{i_2}\ldots s_{i_k}\#t_{i_1}t_{i_2}\ldots t_{i_k}\#$$

and

$$\#u_{i'_1}u_{i'_2}\ldots u_{i'_{k'}}\#t_{i'_1}t_{i'_2}\ldots t_{i'_k}\#.$$

This can only happen if $t_{i_1}t_{i_2}\ldots t_{i_k} = t_{i'_1}t_{i'_2}\ldots t_{i'_k}$ and $s_{i_1}s_{i_2}\ldots s_{i_k} = u_{i'_1}u_{i'_2}\ldots u_{i'_{k'}}$.
But if $t_{i_1}t_{i_2}\ldots t_{i_k} = t_{i'_1}t_{i'_2}\ldots t_{i'_k}$, they represent the same tile sequence.
Moreover, if $s_{i_1}s_{i_2}\ldots s_{i_k} = u_{i'_1}u_{i'_2}\ldots u_{i'_{k'}}$, then the top and bottom strings from this tile sequence must be the same.
$\Rightarrow$ Our PCP instance has match.

■

2. (10 marks) Let $B = \{0^n 1^n \mid n \in \mathbb{N}\}$. Then we can write the languge L_5 from assignment 1 as

$$L_5 = \{\langle M \rangle \mid \overline{L(M)} \subseteq B\}.$$

Let $L_2' = \{\langle M \rangle \mid |\overline{L(M)} \cap \overline{B}| < \infty\}$.

Show that $L_5 \leqslant_m L_2'$.

Solution:

Let $w_0, w_1, \ldots$ be an effective enumeration of Σ^*.

Let $P =$ "On input $\langle M \rangle$:

1. Let $M_0 =$ "On input x:
    1. For $i = 0$ to ∞:
    2. If $w_i == x$, *accept*.
    3. Otherwise, if w_i is *not* of the form $0^n 1^n$ for any $n \in \mathbb{N}$, run M on w_i.
    4. If it accepts, *accept*.
    5. Otherwise *loop*."
2. Return $\langle M_0 \rangle$."

Proof that $\langle M \rangle \in L_5$ iff $\langle M_0 \rangle \in L_2'$:

Suppose that $\langle M \rangle \in L_5$.
Then, using our characterization of L_5 from the assignment 1 solutions, M accepts every string $w \notin B$.
So suppose that we run M_0 on some input x.
When we do so, there will be no loop iteration i in which M_0 can loop in lines 3. or 5., since we will either not run M on w_i or M will ccept w_i.
So every loop iteration of M_0 will eventually halt.
This means that it will eventually reach the loop itertion in which $w_i = x$, at which point it will accept.
So M_0 accepts every string, and therefore $\overline{L(M_0)} = \varnothing$.
But then $\overline{L(M_0)} \cap \overline{B} = \varnothing$, and so $|\overline{L(M_0)} \cap \overline{B}| = 0 < \infty$.
$\Rightarrow \langle M_0 \rangle \in L_2'$.
Suppose that $\langle M \rangle \notin L_5$.
Then there is some i' such that $w_{i'} \notin B$ and such that M does not accept $w_{i'}$.
If we run M_0 on ny x, then, and if M_0 reaches the loop iteration i', it will either loop on line 3. or on line 5.
So if we run M_0 on any w_j for $j > i'$, either M_0 will either loop on iteration i' or on nearlier iteration.
Since there are only a finite number of strings w_j for $j \leqslant i'$, M_0 accepts only a finite number of strings.
In particular, it accepts only a finite number of strings in $\overline{B}$.
But since $|\overline{B}| = \infty$, this mens that $|\overline{L(M_0)} \cap \overline{B}| = \infty$.
$\Rightarrow \langle M_0 \rangle \notin L_2'$.

■

3. (10 marks) Show that $\overline{L_5} \leqslant_m L_2'$.

Solution:

Let $w_0, w_1, \ldots$ be an effective enumeration of Σ^*.

Let P = "On input $\langle M \rangle$:

1. Let N = "On input w:
    1. *Flag* $= T$.
    2. *Flag* $= F$.
    3. For $i = 0$ to ∞:
    4. If w_i is *not* of the form 0^n1^n for any $n \in \mathbb{N}$, run M on w_i.
    5. If it accepts:
    7. *Flag* $= T$.
    8. *Flag* $= F$.
    9. Otherwise, *loop forever*."
2. Let M_0 = "On input x:
    1. Let $i = |x|$.
    2. Run N on ε for i steps.
    3. If $N.Flag == T$ after i steps, *reject*.
    4. Otherwise, *Accept*."
3. Return $\langle M_0 \rangle$."

Claim: $\langle M \rangle \in L_5$ iff, when we run the TM N on the input ε, there will be an infinite number of timesteps in which *Flag* is T.

Proof:

$\Rightarrow$) If $\langle M \rangle \in L_5$, then M accepts every string not in B.
Suppose that we run N on the input ε.
Since we only run M on w_i if $w_i \notin B$, and since M accepts every such w_i, no loop iteration i of N can loop on lines 4. or 9.
So every loop iteration eventually finishes, and so N will pass through an infinite number of loop iterations.
But every loop iteration has step in which *Flag* is T, and so there are an infinite number of steps in which *Flag* is T.

$\Leftarrow$) If $\langle M \rangle \notin L_5$, then there is some $w_{i'} \notin B$ such that M does not accept $w_{i'}$.
If we run N on the input ε, then, and if N reaches the loop iteration i', it will loop in either lines 4. or 9.
But then N will never reach any loop itertion i for $i > i'$.
So N only reaches a finite number of loop iterations.
Since there is only one step per loop itertion in which *Flag* is T, there are only a finite number of timesteps in which *Flag* is T.

Proof that $\langle M \rangle \in \overline{L_5}$ iff $\langle M_0 \rangle \in L_2'$:

Suppose that $\langle M \rangle \in \overline{L_5}$.
Then, by our claim, when we run N on ε there are only a finite number of timesteps in which *Flag* is T.
So there are only a finite number of i such that, if we run M_0 an x of length i, it will reject on line 3.
Since there are only finite number of strings of any given size i, this means tht there are only finite number of strings x such tht, if we run M_0 on x, it will reject on line 3.
Since M_0 always halts, it will accept every other string.
So $|\overline{L(M_0)}| < \infty$.
In particular, $|\overline{L(M_0)} \cap \overline{B}| < \infty$.
$\Rightarrow \langle M_0 \rangle \in L_2'$.

Suppose that $\langle M \rangle \notin \overline{L_5}$.
Then, by our claim, when we run N on ε there are an infinite number of timesteps in which *Flag* is T.
So there are an infinite number of i such that, if we run M_0 an x of length i, it will reject on line 3.
Therefore, there are an infinite number of strings $x = 1^i$ that M_0 will reject.
Since $1^i \notin B$ for any $i > 0$, there are an infinite number of strings in $\overline{B}$ that are not accepted by M_0.
$\Rightarrow \langle M_0 \rangle \notin L_2'$.

■

4. (5 marks) Consider the language

$$\text{FACT-RANGE} = \big\{ \langle n, a, b \rangle \,\big|\, n, a, b \in \mathbb{N}, \text{ and } \exists k \in \mathbb{N}, a \leqslant k \leqslant b \text{ and } k \text{ divides } n \big\}.$$

Now, consider the following program to solve FACT-RANGE:

FIND-FACT= "On $\langle n, a, b \rangle$:
1. If $(0 < n < a)$ or $(b < a)$:
2. *Reject.*
3. For $i = \max(a, 1)$ to b:
4. If $n \,\%\, i == 0$:
5. *Accept.*
6. *Reject.*

Is this a polytime algorithm? Why or why not?

Solution:

This is not a polytime algorithm, since the inputs n, a, and b are numbers that are not given in unary. So the size of these numbers is exponential in the input size, and so the loop from lines 3-5 will run an exponential number of times in the worst case.

□

5. (15 marks) Let $\text{STORAGE-BOX} = \left\{ \langle U, b, k \rangle \;\middle|\; \begin{array}{l} U \text{ is a finite set of elements such that} \\ \text{each } u \in U \text{ has a size } s(u) \in \mathbb{N}, \text{ where } b, k \in \mathbb{N}, \text{ and} \\ \text{where the elements in } U \text{ can placed into } k \text{ boxes,} \\ \text{each of which has size at most } b. \end{array} \right\}$.

(a) (5 marks) Show that $\text{STORAGE-BOX} \in \mathcal{NP}$.

Solution:

To show that this is in $\mathcal{NP}$:
1) It is a yes/no problem.
2) Its certificate is the box assignment $A = (U_1, U_2, \ldots, U_k)$.
3) The size of this certificate is at most $\mathcal{O}(|\langle U \rangle|)$, since A is simply a list of subsets of U without repitition. This certificate is polynomial in the size of $\langle U, b, k, A \rangle$.

4) A verifier for this certificate would be:
Let $\mathcal{V}$ ="On $\langle U, b, k, A\rangle$:"
1. Reject if any of the following checks fails:
$\mathcal{O}(1)$ *time.*
2. Check that $k \leqslant |U|$.
$\mathcal{O}(|U|)$ *time.*
3. Check that $\bigcup_{i=1}^{k} U_i = U$.
$\mathcal{O}(|U|^2)$ *time.*
4. Check that for all $1 \leqslant i < j \leqslant k$, $\cap = U_i \cap U_j = \varnothing$.
$\mathcal{O}(|U|^4)$ *time.*
5. For $i = 1$ to k:
$\mathcal{O}(|U|)$ *iterations.*
6. Let $sum = 0$.
$\mathcal{O}(1)$ *time per iteration.*
7. For $u \in U_i$:
$\mathcal{O}(|U_i|)$ *iterations.*
8. Let $sum+ = s(u)$.
$\mathcal{O}(\log s(u))$ *time per iteration.*
9. Check that $sum \leqslant b$.
$\mathcal{O}(\log b)$ *time per iteration.*
10. *Accept.*
$\mathcal{O}(1)$ *time.*

5) We can see that we can amortize over the loops to get a running time of $\mathcal{O}(|\langle U\rangle|^4 + |U| \log b)$ time, and so is polytime in the size of G.

So STORAGE-BOX is in $\mathcal{NP}$.

■

(b) (10 marks) Assuming that SUBSET-SUM is $\mathcal{NP}$-complete, show that STORAGE-BOX is also $\mathcal{NP}$-complete.

Solution:

Let $\langle S, t\rangle$, be a SUBSET-SUM instance, where S is a finite set of non-negative integers. Set $s = \sum_{x \in S} x$, let $w = s + t$, and let $S' = S \cup \{w - s + t + 1, 2w - t + 1\}$.

We initilize $U = \varnothing$, and for each $x \in S'$ we add an element u_x to U with a size $s(u_x) = x$.

We set $b = 2w + 1$ and $k = 2$.

Note that $w + t + 1 > s$ and that $2w - t + 1 > s$. Also note that $s \geqslant x \geqslant 0$ for all $x \in S$. So we'll end up with a valid instance of STORAGE-BOX.

Proof that $\langle S, t\rangle \in$SUBSET-SUM **iff** $\langle U, b, k\rangle \in$STORAGE-BOX**:**

$\Rightarrow$) Suppose that $\langle S, t\rangle \in$SUBSET-SUM:

Then, there is some subset $X \subseteq S$ such that $\sum_{x \in X} = t$.

Let $X' = X \cup \{2w - t + 1\}$.

Then,

$$X' \subseteq S', \text{ and } \sum_{x \in X'} x = \sum_{x \in X} x + (2w - t + 1) = t + (2w - t + 1) = 2w + 1.$$

So if we set $U_1 = \left\{u_x \,\middle|\, x \in X'\right\}$, the contents of U_1 will take up exactly $2w + 1 = b$ space.

On the other hand, we can also see that

$$\sum_{x \in S' \setminus X'} x = \sum_{x \in S \setminus X} x + (w - s + t + 1) = (s - t) + (w - s + t + 1) = 2w + 1.$$

So when we assign u_x to box number 2 for all $x \notin X'$, the contents of box 2 will also take up exactly $2w + 1 = b$ space.
So if $U_2 = \{u_x \mid x \in U \setminus X'\}$, then $A = (U_1, U_2)$ is a certificate for $\langle U, b, k\rangle$.

$\Rightarrow U, b, k' \in$ STORAGE-BOX.

$\Leftarrow$) Suppose that $\langle U, b, k\rangle \in$ STORAGE-BOX:

Then, there is some box assignment $A = (U_1, U_2)$, where U_i is the subset of elements in U that are sent to box i.

Now, the element u_x for $x = (2w - t + 1)$ must be in one of these two sets. Since we can swap the two sets and still have a valid box assignment, we'll say wlog. that it is in U_1.

We'll let $X' = \{x \in S' \mid u_x \in U_1\}$.
Recall that $S' = S \cup \{(2w - t + 1), (w - s + t + 1)\}$.

Now, we can see that

$$\sum_{x \in S'} x = \sum_{x \in S} +(2w - t + 1) + (w - s + t + 1) = s + (2w - t + 1) + (w - s + t + 1) = 4w + 2.$$

Note that $4w + 2 = 2(2w + 1) = 2b$, and that

$$\sum_{x \in X'} x + \sum_{x \in S' \setminus X'} x = \sum_{x \in X'} s(u_x) + \sum_{x \in S' \setminus X'} s(u_x) \leqslant b + b = 2b,$$

with equality only when

$$\sum_{x \in X'} x = \sum_{x \in S' \setminus X'} x = \Big(\sum_{x \in S'} x\Big)/2 = b.$$

In particular, we know that we cannot have both $(2w - t + 1)$ and $(w - s + t + 1) \in X'$, since then we would have

$$\sum_{x \in X'} \geqslant (2w - t + 1) + (w - s + t + 1) = 3w - s + 2 \geqslant 2w + 2 > 2w + 1.$$

Similarly, we can't have both $(2w - t + 1)$ and $(w - s + t + 1) \notin X'$.

Therefore, since we hve asserted that $(2w - t + 1) \in X'$, we also know that $(w - s + t + 1) \in S' \setminus X'$.

Let $X = X' \setminus \{(2w - t + 1)\}$.

Then,

$$\sum_{x \in X'} x = \sum_{x \in X} x + (2w - t + 1) = 2w + 1,$$

and so

$$\sum_{x \in X} x = 2w + 1 - (2w - t + 1) = t.$$

Moreover, we can see that, since $(w - s + t + 1) \notin X'$, $(w - s + t + 1) \notin X$, and so $X \subseteq S$.

So X is a subset of S that sums to t.

$\Rightarrow \langle S, t\rangle \in$ SUBSET-SUM.

Finally, we can see that the reduction takes $\mathcal{O}(|\langle S \rangle|)$ time, since it just involves copying out the original input with two new numbers, and in setting the numbers b and k, which also takes polytime. So the entire process takes polynomial time.

So STORAGE-BOX is $\mathcal{NP}$-complete, as required.

Note: If you check you'll see that this reduction is essentially a copy of the reduction for PARTITION *from the second Polytime Reductions handout. This is a very natural reduction for this language, but you can get another fairly natural reduction from a language called* 3-PARTITION*. We're not covering this language in this course – the $\mathcal{NP}$-completeness proof takes up bit more space than I'd want – but it's an interesting variant of the partition problem because, unlike* SUBSET-SUM *and* PARTITION*, it can't be solved using a quasipolytime algorithm unless $\mathcal{P} = \mathcal{NP}$. We haven't talked bout quasipolytime algorithms yet, but you might see them in other courses, such as CSCC73. So* 3-PARTITION *and* STORAGE-BOX *would remain $\mathcal{NP}$-complete even if we gave the numbers in the input in unary.*

■

6. (15 marks) Let HALF-PATH $= \left\{ \langle G = (V, E), s, t \rangle \;\middle|\; \begin{array}{l} G \text{ is a directed graph, } s, t \in V, \\ \text{and } G \text{ has a simple path from } s \text{ to } t \\ \text{that passes through at least } |V|/2 \text{ vertices.} \end{array} \right\}$.

   (a) (5 marks) Show that HALF-PATH $\in \mathcal{NP}$.

   Solution:

   1) It is a yes/no problem.
   2) Its certificate is a half-path P from s to t (we can treat this as a list of the vertices of V).
   3) The size of this certificate is at most $\mathcal{O}(n)$, where n is the number of vertices in G. This certificate is polynomial in the size of G.
   4) A verifier for this certificate would be:

      Let $\mathcal{V}$ ="On $\langle G = (V, E), s, t, P \rangle$:"

      1. Reject if any of the following checks fails:
         $\mathcal{O}(1)$ *time.*
      2. Check that P has at least $n/2 = |V|/2$ distinct elements and no repeats.
         $\mathcal{O}(n^2)$ *time.*
      3. For $i = 1$ to $len(P) - 1$:
         $\mathcal{O}(n)$ *iterations.*
      4. Check that $(P[i], P[i+1]) \in E$.
         $\mathcal{O}(n)$ *or* $\mathcal{O}(1)$ *time per iteration, depending on your implementation.*
      5. Check that $P[1] = s$ and $P[len(P)] = t$.
         $\mathcal{O}(1)$ *time.*
      6. *Accept.*
         $\mathcal{O}(1)$ *time.*

   5) We can see that this verifier runs in $\mathcal{O}(n^2)$ time, and so is polytime in the size of G.

   So HALF-PATH is in $\mathcal{NP}$.

   ■

   (b) (10 marks) Assuming that HAM-PATH is $\mathcal{NP}$-complete, show that HALF-PATH is also $\mathcal{NP}$-complete.

   Solution:

   Let $\langle G = (V, E), s, t \rangle$ be a HAM-PATH instance, and let $n = |V|$.

   Let $G' = (V', E)$ be a copy of G with n disconnected vertices added.

Proof that $\langle G, s, t\rangle \in$HAM-PATH **iff** $\langle G', s, t\rangle \in$HALF-PATH**:**

$\Rightarrow$) Suppose that $\langle G, s, t\rangle \in$ HAM-PATH.
Then G has Hamiltonian path $P = (v_{i_1} = s, v_{i_2}, \ldots, v_{i_n} = t)$.
Since P is path in G, it is also path in G'.
Now, the total number of nodes in G' is $2n$, nd the number of nodes in P is n.
So P passes through at least $|V'/2 = 2n/2 = n$ vertices of G'.
Moreover, this is a simple path, since every Hamiltonian path only passes through each vertex once.
$\Rightarrow \langle G', s, t\rangle \in$ HALF-PATH.

$\Leftarrow$) Suppose that $\langle G', s, t\rangle \in$ HALF-PATH.
Then G' has a simple path $P' = (v_{i_1} = s, v_{i_2}, \ldots, v_{i_k} = t)$, where $k \geqslant |V'|/2 = 2n/2 = n$.
Since the vertices in $V' \backslash V$ are all disconnected, they cannot be part of any path containing s and t.
So the only vertices in P must be vertices in V (and so P is a path in G).
Since there are at least n nodes in P, and since there are exactly n nodes in V, P must be a simple path in G containing exactly $n = |V|$ vertices.
So P is a Hamiltonian path in G.
$\Rightarrow \langle G, s, t\rangle \in$ HAM-PATH.

■

7. (15 marks) Let MONOTONE-3SAT be the language

$$\left\{ \langle \phi \rangle \;\middle|\; \begin{array}{l} \phi \text{ is a satisfiable } 3CNF \text{ Boolean formula in which} \\ \text{every clause either only contains three distinct} \\ \text{negated variables or three distinct un-negated variables.} \end{array} \right\}.$$

(a) (5 marks) Show that MONOTONE-3SAT $\in \mathcal{NP}$.

Solution:

To show that this is in $\mathcal{NP}$:

1) It is a yes/no problem.
2) Its certificate is simply a truth assignment τ to the variables in ϕ.
3) The size of this certificate is $\mathcal{O}(k)$, where k is the number of variables in ϕ. Since every variable occurs at least one in the clauses, this certificate is polynomial in the size of ϕ.
4) A verifier for this certificate would be:
   Let $\mathcal{V}$ ="On $\langle \phi, \tau\rangle$:"
   1. Check that every clause of ϕ has either only true or only false literals. *Reject* if this check fails.
      $\mathcal{O}(n)$ *time.*
   2. For $i = 1$ to n (where n is the number of clauses in ϕ):
      $\mathcal{O}(n)$ *iterations.*
   3. Use τ to assign truth values to the literals in the clauses of the clause c_i.
      $\mathcal{O}(n)$ *time per iteration.*
   4. If τ does not set exactly one literal in c_i to true, *reject.*
      $\mathcal{O}(1)$ *time per iteration.*
   4. *Accept.*
      $\mathcal{O}(1)$ *time.*
5) We can see that this verifier runs in $\mathcal{O}(n^2)$ time, and so is polytime in the size of ϕ.

So MONOTONE-3SAT is in $\mathcal{NP}$.

■

(b) (10 marks) Assuming that 3SAT is $\mathcal{NP}$-complete, show that MONOTONE-3SAT is also $\mathcal{NP}$-complete.

Solution:

We'll give a polytime reduction from 3SAT, so suppose we are given a 3CNF ϕ. Using the helpful 3SAT lemma from the Polytime Reductions handout, we see that we can assume wlog. that each clause of ϕ has three distinct variables.

Let's say that ϕ has n clauses and k variables. So the variables are x_1, x_2, …, x_k.

To start, we'll add six new variables: x_a, x_b, x_c, x_d, x_e, nd x_f. We'll now build a ϕ' using the following clauses:

$$(x_a \vee x_b \vee x_c) \wedge (x_a \vee x_b \vee x_d) \wedge (x_a \vee x_c \vee x_d) \wedge (x_b \vee x_c \vee x_d).$$

Claim 1: In any truth assignment tht satisfies these clauses, at least two of the variables x_a, x_b, x_c, and x_d must be true.

If this claim were not true, then there would be at least three of these variables that were false. But since every set of three of these variables is a clause of ϕ', we would then have an unsatisfied clause of ϕ'.

We now extend ϕ' by adding the following clauses:

$$\begin{aligned}&(\neg x_a \vee \neg x_b \vee \neg x_e) \wedge (\neg x_a \vee \neg x_c \vee \neg x_e) \wedge (\neg x_a \vee \neg x_d \vee \neg x_e)\\ \wedge&(\neg x_b \vee \neg x_c \vee \neg x_e) \wedge (\neg x_b \vee \neg x_d \vee \neg x_e) \wedge (\neg x_c \vee \neg x_d \vee \neg x_e)\\ \wedge&(\neg x_a \vee \neg x_b \vee \neg x_f) \wedge (\neg x_a \vee \neg x_c \vee \neg x_f) \wedge (\neg x_a \vee \neg x_d \vee \neg x_f)\\ \wedge&(\neg x_b \vee \neg x_c \vee \neg x_f) \wedge (\neg x_b \vee \neg x_d \vee \neg x_f) \wedge (\neg x_c \vee \neg x_d \vee \neg x_f).\end{aligned}$$

Claim 2: In any truth assignment that satisfies these clauses, the two variables x_e, and x_f must be false.

Indeed, by **Claim 1** we know that in any satisfying truth assignment to ϕ' there must be some pair of varibles in x_a, x_b, x_c, and x_d that are both true. If, say, the two true variables are x_a and x_b, then the only way to satisfy the clauses $(\neg x_a \vee \neg x_b \vee \neg x_e)$ and $(\neg x_a \vee \neg x_b \vee \neg x_f)$ is to set x_e and x_f to false. The same arguemnt applies to the other five pairs.

Next, we add to ϕ' the clauses

$$(x_a \vee x_e \vee x_f) \wedge (x_{ba} \vee x_e \vee x_f) \wedge (x_c \vee x_e \vee x_f) \wedge (x_d \vee x_e \vee x_f).$$

Claim 3: In any truth assignment that satisfies these clauses, the four variables x_a, x_b, x_c, and x_d must all be true.

By **Claim 2** we know that in any satisfying truth assignment to ϕ' x_e and x_f are both false. So the only way to satisfy these four clauses is to set the four variables x_a, x_b, x_c, and x_d to true.

We'll now encode the variables in ϕ into ϕ'. As we've stated, the variables in ϕ are $x_1, x_2, \ldots, x_n$, and so we'll add these variables to ϕ' as well. We'll lso add the variables $y_1, y_2, \ldots, y_k$. For every $1 \leqslant i \leqslant k$, we'll add the following two clauses to ϕ':

$$(x_i \vee y_i \vee x_e) \wedge (\neg x_i \vee \neg y_i \vee \neg x_a).$$

Claim 4: In any truth assignment that satisfies these clauses, each of the variables x_i and y_i take opposite truth values.

By **Claim 2** we know that in any satisfying truth assignment to ϕ' x_e must be false, and by **Claim 3** we know that x_a must be true. So the clause $(x_i \vee y_i \vee x_e)$ ensures that at least one of x_i and y_i must be true, and the clause $(\neg x_i \vee \neg y_i \vee \neg x_a)$ ensures that at least one is false.

Finally, for any clause $C_j = (\ell_{j_1} \vee \ell_{j_2} \vee \ell_{j_3})$ of ϕ, we add a clause $C'_j = (\ell'_{j_1} \vee \ell'_{j_2} \vee \ell'_{j_3})$ to ϕ', where for $r \in \{1, 2, 3\}$,

$$\ell'_{j_r} = \begin{cases} x_i, & \ell_{j,r} = x_i, \\ y_i, & \ell_{j,r} = \neg x_i \end{cases}$$

E.g., we would replace the clause $(x_1 \vee \neg x_2 \vee x_3)$ *with the clause* $(x_1 \vee y_2 \vee x_3)$.

We note that every clause that we have added to ϕ' contains only positive or only negative literals,and that each has three distinct variables. So $\langle \phi' \rangle$ is a valid instance of MONOTONE-3SAT.

Proof that $\langle \phi \rangle \in$3SAT **iff** $\langle \phi' \rangle \in$MONOTONE-3SAT**:**

$\Rightarrow$) Suppose that $\langle \phi \rangle \in$ 3SAT.
Then it has satisfying truth assignment τ.
We'll extend τ into truth assignment τ' for ϕ' by setting

$$x_a = x_b = x_c = x_d = T,$$

$$x_e = x_f = F,$$

and by setting $y_i = \neg x_i$ for every $1 \leqslant i \leqslant k$.
As we have seen in our claims, this truth assignment will satisfy the clauses

$$(x_a \vee x_b \vee x_c) \wedge (x_a \vee x_b \vee x_d) \wedge (x_a \vee x_c \vee x_d) \wedge (x_b \vee x_c \vee x_d),$$

the clauses

$$\begin{array}{c} (\neg x_a \vee \neg x_b \vee \neg x_e) \wedge (\neg x_a \vee \neg x_c \vee \neg x_e) \wedge (\neg x_a \vee \neg x_d \vee \neg x_e) \\ \wedge (\neg x_b \vee \neg x_c \vee \neg x_e) \wedge (\neg x_b \vee \neg x_d \vee \neg x_e) \wedge (\neg x_c \vee \neg x_d \vee \neg x_e) \\ \wedge (\neg x_a \vee \neg x_b \vee \neg x_f) \wedge (\neg x_a \vee \neg x_c \vee \neg x_f) \wedge (\neg x_a \vee \neg x_d \vee \neg x_f) \\ \wedge (\neg x_b \vee \neg x_c \vee \neg x_f) \wedge (\neg x_b \vee \neg x_d \vee \neg x_f) \wedge (\neg x_c \vee \neg x_d \vee \neg x_f), \end{array}$$

the clauses

$$(x_a \vee x_e \vee x_f) \wedge (x_{ba} \vee x_e \vee x_f) \wedge (x_c \vee x_e \vee x_f) \wedge (x_d \vee x_e \vee x_f),$$

and

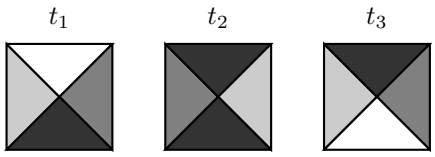
$$(x_i \vee y_i \vee x_e) \wedge (\neg x_i \vee \neg y_i \vee \neg x_a)$$

from ϕ'. So the only remaining claues we need to worry about are the clauses $C'_j = (\ell'_{j_1} \vee \ell'_{j_2} \vee \ell'_{j_3})$, where $C_j = (\ell_{j_1} \vee \ell_{j_2} \vee \ell_{j_3})$ is in ϕ.
But we can see tht since every $y_i = \neg x_i$, the literals ℓ'_{j_1}, ℓ'_{j_2}, and ℓ'_{j_3} take the sme values as ℓ_{j_1}, ℓ_{j_2}, and ℓ_{j_3}. Since at least one of these values must be T, we can conclude that C'_j must be satisfied.
Since this is true for every C'_j in ϕ', τ' is a satisfying truth assignment to ϕ'.
$\Rightarrow \langle \phi' \rangle \in$ MONOTONE-3SAT.

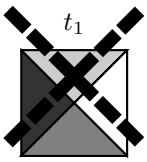
$\Leftarrow$) Suppose that $\langle \phi' \rangle \in$ MONOTONE-3SAT.
Then ϕ' has satisfying truth assignment τ'.
We let τ be the restriction of τ' to the variables x_i, where $1 \leqslant i \leqslant k$.
We'll argue that τ is a satifying truth assignment for ϕ.
Indeed, we know from *Claim 4* that $y_i = \neg x_i$ for all $1 \leqslant i \leqslant k$.
So consider any clause $C_j = (\ell_{j_1} \vee \ell_{j_2} \vee \ell_{j_3})$ in ϕ.
The corresponding clause $C'_j = (\ell'_{j_1} \vee \ell'_{j_2} \vee \ell'_{j_3})$ in ϕ' must be satified by τ', and the literals ℓ_{j_1}, ℓ_{j_2}, and ℓ_{j_3} take the same values as ℓ'_{j_1}, ℓ'_{j_2}, and ℓ'_{j_3}.
So at least one of ℓ_{j_1}, ℓ_{j_2}, and ℓ_{j_3} mut be T, and so C_j must be satisfied.
Since this is true for every C_j in ϕ, τ is a satisfying truth assignment for ϕ.
$\Rightarrow \langle \phi \rangle \in$ 3SAT.

■

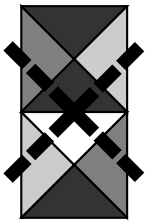
8. (20 marks) In this question we'll see an undecidable problem that is somewhat similar to the PCP. Suppose that I give you a finite set of tiles:



Our aim will be to cover the entire plain with copies of these tiles. We can re-use the tiles as many times as we want, but we cannot rotate them:

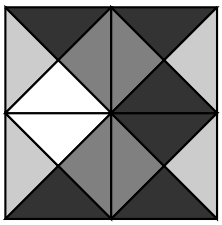


We'll have to place the copies of our tiles side-to-side, and they have to match in their colours:



Here we've tried placing t_2 dirctly above t_1 but, as you can see, the bottom colour of t_2 does not match the top of t_1.

We could, however, cover the entire plain with the tiles in our example – we'd be able to build the following 2×2 block and repeat it ad infinitum:



That is, we could use block of tiles of the form

$$\begin{matrix} t_3 & t_2 \\ t_1 & t_2 \end{matrix}$$

Note that not all tilings of the plane have to be periodic!

It turns out that it is undecidable to determine, given a set of tiles, whether it can cover the whole plane. The proof looks a bit like the PCP proof that we've gone over in class, but there are extra mechanisms we'd need which we won't go into detail about here. What we can do with the tools from class is work with a restricted version of the tiling problem.

Consider the following languge:

Definition: MULTI-TILE

Instance: Finite set T of four-colour tiles of the sort seen above, along with two distinguished tiles t_a nd t_b.

Question: Is there a way to cover the entire plain with the tiles in T in such a way that:

1) The tile t_a is used at least once, and
2) the tile t_b is used an infinite number of times.

There must be eactly one time on every point in the plain, and the colouring and rotation rules we've described above must be followed.

Show that MULTI-TILE is neither recognizable nor co-recognizable.

Solution:

You'll see that this solution is a bit on the long side. That's to be expected for a 20-mark question, but it'll also be taken into account while marking. There'll be about 10 marks given for showing the TM to tile encoding, and 10 for tying that encoding into one or more reductions. A TM to tile encoding that misses few of the required constraints, such as the different colours on the two sides of the TM head tiles, can still get most of the marks – so long as the underlying idea is sound.

We'll do this in two parts:

1) We'll demonstrte how to encode a TM operation into the MULTI-TILE problem.
2) We'll use this encoding to give a mpping reduction $\text{ALL}_{\text{TM}} \leqslant_m$ MULTI-TILE (note that this will mean that MULTI-TILE is in fact Π_2-hard).

So let's look at the two parts.

1) Our first task is to encode a TM operation into the plne tiling model. The most straightforward way to do this is for a deterministic TM/input pair $\langle M, w\rangle$ is to set up a set of tiles that will tile the plne if M *loops*, nd which will fail to cover the plne otherwise.

   We can modify this encoding so that if M halts then we can use the tile t_b to finish the covering of the plane, but the ALL_{TM} reduction will give us a stronger result for less work.

   We can also get this basic idea to work for non-deterministic TM, but it'll be more work.

   So to start, let's look at what happens if we run M on w. When we do, we can write up the configurations that we make in a table:

#	q_0	w_0	w_1	...	w_{n-1}	␣	␣	...
#	$t_{1,0}$	q_1	w_1	...	w_{n-1}	␣	␣	...
#								
#								
⋮	⋮	⋮				⋮		⋮

   This is the same idea as we use in the Cook-Levin reduction, actually, but this time we don't bound the size of the table. As before, the "#" character is a special charcter that we use to delimit the left hand side of the tape.

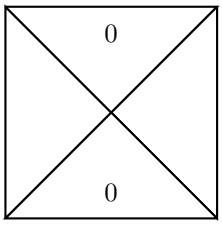
We'll set up our encoding so that each cell of this table correponds to single tile, and so that the colour matching constraints ensure that the tile order matches that of the TM operation.

The top left tile will more or less correpond to the initial tile t_a. We'll assume wlog. that the t_a tile occurs on the plane at the position $(0, 0)$, nd that the TM configurations take up the lower right quadrant of the plane.

> *Techncally, this approach will only let us cover the lower right quadrant of the plane. We'll just colour the other three quadrants black. Once we've seen how to build the tiles for the overall TM encoding, we'll be able to connect them to the boundary between the sections.*

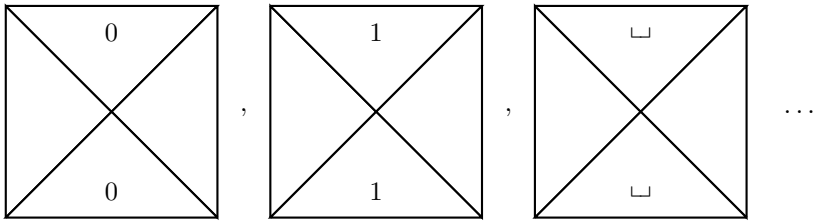
So let's look at the different tiles we'll use in out encoding. The first item we'll note is that we represnt colours in our computers using strings. So we'll use different strings to represent the colours in our tiles.

Now, the first tiles we'll build will represent the cells in the middle of our table *that are not adjacent to TM head*. We'll deal with the TM head soon. These tiles have esy constraints: the tiles above and below must contain the same character, but we don't care what the tiles to the sides are. We can represnt this using the following type of tile:

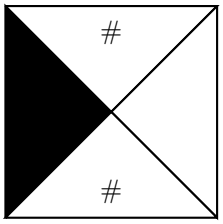


This tile will represent cell containing the chracter 0: generally, the top colour indicates the contents of the current cell, and the bottom indicates the contents of the next cell down in the table. The cells that don't have strings we'll colour white.

We'll have one such tile for every character in the tape alphabet Γ of M:



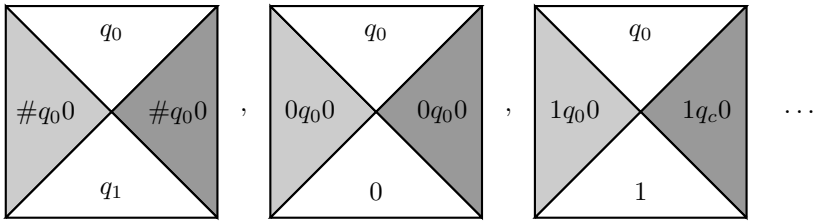
It's easy enough to adapt this pproach to the "#" character that delimits the left-hand side of the tape, as well – we'll just colour its left side black to indicate tht the TM encoding stops there:



And this will give us enough power to encode all of the inner tiles not adjacent to the TM head. But the TM head gives us bit of a problem, since it can change the tiles on either of its sides. We'll use the side colours to encode these constraints. So suppose, for example, that our TM M contains the transition rule

$$(q_0, 0) \to (q_1, x, L).$$

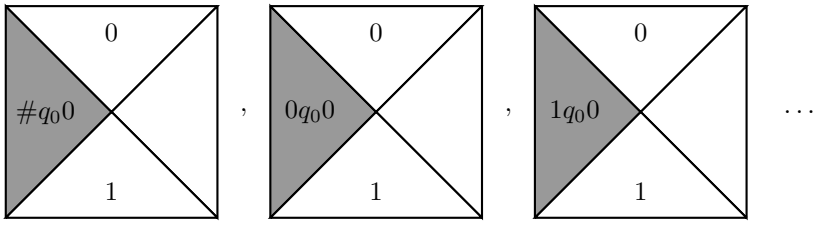
We'll need to encode this rule into our tiles, and we can see that this rule will modify the tiles on either side. So we'll colour both sides of the head tiles with a non-white colour, and we'll include the TM state and both side tiles in the colour string. So we'd build a set of tiles:



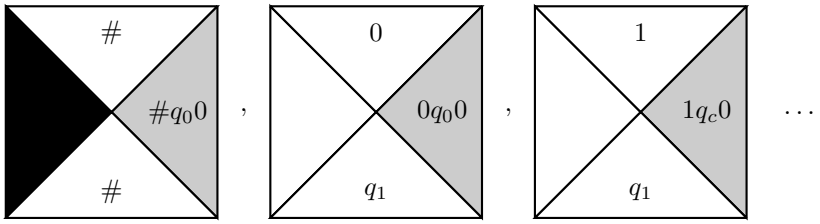
Note that if the left hand character is the "#" character then the TM head can't move (it's the left side of the tape), so the lower character has to be the TM head again. Otherwise we set it to the character from the left. We'll do this for every character that could be to the left of the tape head.

Also note that we've used different greyscal for the left and right rules. This is to ensure that the two character tiles on the side can't connect directly to each other without a TM head in between. We're depicting the difference as greyscale difference, but in relity it will be an addition to the string for the colour.

We'll couple these tiles on the left and right with modified versions of the character tiles. For this particular rule the tiles on the right will be:



While the tiles on the left will be:

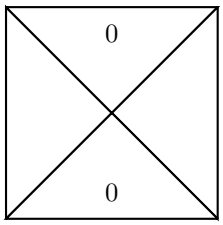


The process for TM rules that move the head right is similar.

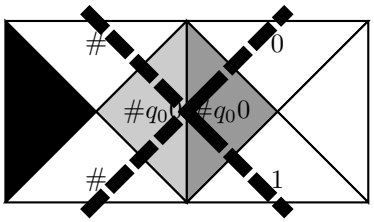
Note that the side triples in TM head tiles all contain a state consistent with the TM head, that the first character in the triple is consistent with the character in the left tile, and that the last character in the triple is consistent with the character in the right tile. We'll build a set of tiles that follow these constraints for each possible TM rule in δ.

Now, we can see that if we have the right initial configuration on the top row of the quadrant (followed by an infinite number of ␣s), then every subequent row must encode the correct configuration:

- If a character (e.g., 0) is not next to a TM head, then it must be encoded by a tile of the form:



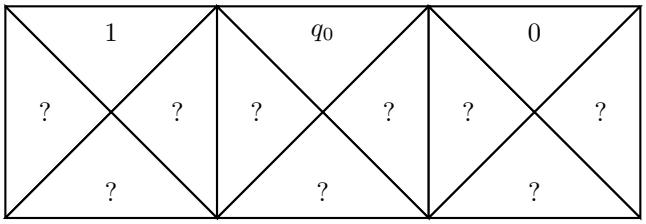
That is, the top string of the tile has to be the character (e.g., 0). On the other hand, we cannot choose any of the tiles that connect to the TM head, since these tiles can only connect to the (non-adjacent) TM head. The different greyscales mean that we we cannot match, e.g.:



So every character tile not adjacent to a TM head will force the next row to repeat the same character, which is what we want.

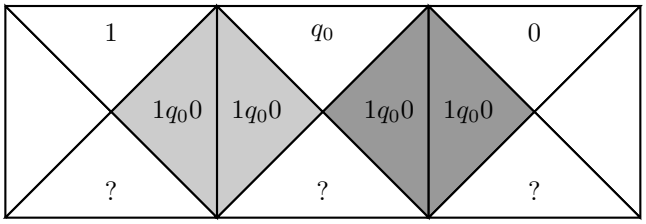
- On the other hand, we can see that we can always place an appropriate set of head rule tiles for the head and the characters to its sides (if there is a TM transition at all for the current configuration), and that the characters on the next row down as given by these tiles will match the correct set of characters in the TM operation. Moreover, we see that we will hve to chooe a set of tiles consistent with the TM operation.

  For example, if we use the TM rule $(q_0, 0) \to (q_1 1, L)$ and the current configuration has the sequence $1q_00$, then we have to represent these characters with three tiles starting with:

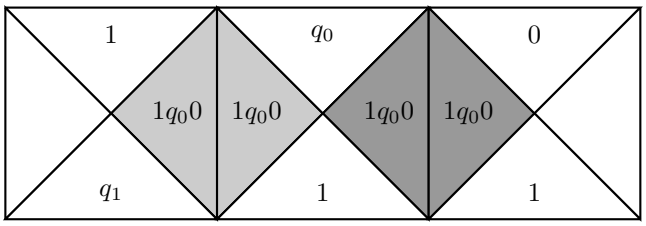


  Now, the colours on the outer sides of thi triple cannot connect to a head tile (there's only the one head), and so they must be white.

  But the only triples that the 1 on the left can hold are the ones that start with a 1. Similarly, the rightmost 0 can only hold triples that end in a 0, and the q_0 can only hold triples using a q_0. And since the q_0 tile must encode some triple, the triple it holds is $1q_00$. So our colouring is



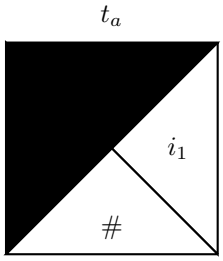
  Since we have assumed that our TM is deterministic, the only tiles that can finish this pattern are



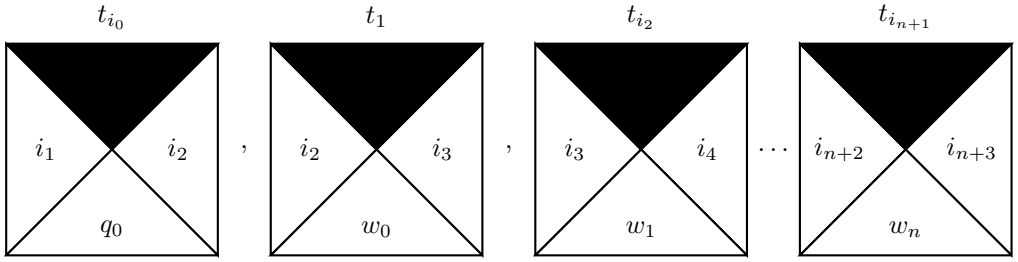
  And so we see that the next configuration will hve the sequence q_111, which is the correct sequence given the TM operation. So the entire next configuration will be correct, and since this will work for every configuration, we can see that the whole configurtion history will be correct.

  This last section is where the assumption that M is deterministic really matters: if it is not then we need to enforece constraints to ensure that the left and right tiles in the triple use the same rule. This wouldn't be insurmountable, but it would take more work.

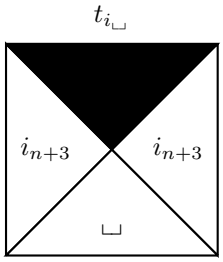
So as long as we caan get the initial configuration set up right, then we'll hve the tile sequence we need. If the initial configuration of M is $q_0 w_0 w_1 \dots w_n$, then we'll start the left hand side of this configuration with the tile t_a:



We'll follow this tile with:



And finally we end the configuration with the repeating tile

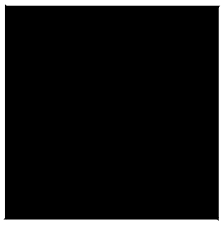


Now, if we start our plane covering with the tile t_a, we can see that the only way to extend the row to the right is to exactly use the sequence of tiles

$$t_a, t_{i_0}, t_{i_1}, \dots, t_{i_{n+1}}, t_{i_\sqcup}, t_{i_\sqcup}, t_{i_\sqcup} \dots.$$

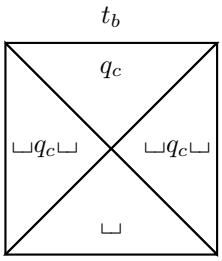
The row immeditately beneath this row must exactly correpond to the initial configuration of M. So the only way to use t_a and to tile the quadrant is to follow the TM operation of M, as desired.

Since the allowable tile sequences match the configurtions of the TM operation, we can see that this set of tiles can cover the lower right quadrant of the plane iff the TM M does not halt on w. We can extend this to a cover of the whole plane by adding a single all-black tile:



So we have our encoding.

2) As we've said, let's do a reduction from ALL_{TM}.
Let $w_0, w_1, \ldots$ be an effective enumeration of Σ^*.
Let $P =$ "On input $\langle M \rangle$:
1. Let $M' =$ "On input x:
    1. Write "$" to the head of the tape.
    The "$" character will be a special charcter not otherwise used.
    2. For $i = 0$ to ∞:
    3. Run M on w_i.
    4. If it accepts:
    5. Move the head one step beyond the right side of the tape.
    6. Switch the TM into special state q_c.
    The "q_c" state will be a special state not otherwise used by the TM.
    7. Leave the q_c state and move the head to the left side of the tape.
    We'll use the "$" character to find the left hand side.
    8. Otherwise, *loop*."
2. Let $\langle T \rangle$ be the set of tiles used to encode M' run on ε.
3. Let t_a be the initial tile of T, and let t_b be the tile



Note that this tile will be used once in our encoding per string accepted by M: It'll be the tile representing the tape head at the beginning of line 7.

We'll use T, along with the t_a from the encoding and the t_b specified above to build our MULTI-TILE instance $\langle T, t_a, t_b \rangle$.

We argue that $\langle T, t_a, t_b \rangle \in$ MULTI-TILE iff $\langle M \rangle \in \text{ALL}_{\text{TM}}$:

$\Leftarrow$) Suppose that $\langle M \rangle \in \text{ALL}_{\text{TM}}$.
Then M accepts every input w.
This means that M' will run M on every string $w_{i'}$ in Σ^*, and that every time it does so M will accept and M' will enter the q_c state.
If we build a tile sequence from the run of M', then, it will cover the plane starting from the t_a tile. This covering of the plane will use the tile t_b an infinite number of times.
So $\langle T, t_a, t_b \rangle$ is a *yes*-instance of MULTI-TILE.

$\Rightarrow$) Suppose that $\langle M \rangle \notin \text{ALL}_{\text{TM}}$.
Then there is some string $w_{i'}$ that M does not accept.
If we try to run M', then, it will loop on (or before) the loop iteration $i = i'$.
Since M' only reaches the state q_c once per main loop iteration, M' will never again reach the state q_c.
So it will visit the state q_c at most a finite number of times.
Now, suppose that $\langle T, t_a, t_b \rangle$ admits a covering of the plane that uses the tile t_a.
We've argued that the only way that we can build such a tile covering is to emulate the TM operation of M'.
So any covering of the plane that uses T and the tile t_a will use the tile t_b once for every step that M' uses the state q_c.
This covering can therefore only use the tile t_b a finite number of times.
So $\langle T, t_a, t_b \rangle$ is a *no*-instance of MULTI-TILE.

■

9. **Bonus** (10 marks — your mark will be rounded to the nearest multiple of 2.5)

Consider the following languge:

Definition: 3COL

Instance: A graph $G = (V, E)$.

Question: Does G have a vlid 3-colouring?
(i.e., a map c from V to {Red, Green, Blue} such that if $\{u, v\} \in E$, then $c(u) \neq c(v)$).

Now, suppose that, instead of asking whether such a colouring c exists, we want to know *how many* such c exist. If $|V| = n$, then the number of certificates c for 3COL is somewhere between 0 and 3^n, inclusive.

Suppose you have an oracle that will tell you whether a graph G with n nodes has at least $3^{n/2}$ colourings. Give polytime algorithm using this oracle to count the eact number of 3-colourings that G has.

Solution:

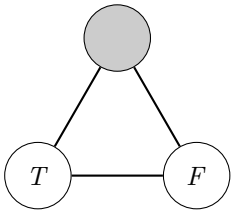
Suppose we are given a graph G with n vertices, and that we have an oracle that will tell us whether an n' graph G' has at least $3^{n'/2}$ 3-colourings.

Let N_G represent the number of 3-colourings of the graph G. Note that $0 \leqslant N_G \leqslant 3^n$. In fact, N_G can only be 3^n if it is totally disconnected, and we can test this. So if G is totally diconnected we return 3^n, and of not we continue with the process. This means that we'll assume in the following that G has at least one edge.

Broadly speaking, we'll find N_G by running a binary search, but the details are going to be tricky. So students will be graded by how effective their binary searches are. Our particular binary search will be implemented by attaching G to a graph that acts as a sort of counter. To do this we'll need to know a colouring for G.

So to start, we want to determine whether G has any 3-colourings at all, and to find a colouring if one exists. Let's do this by building a graph G_1 containing three disconnected sections:

- The first section will be copy of G.
- The second is a copy of the palette widget from our 3COL reduction:



- The third section is made up of $n + 3$ disconnected vertices.

Now, the number of colourings of G_1 is the product of the number of 3-colourings of its sections, since those sections are disconected. There are N_G colourings of G, 6 colourings of the palette, and 3^{n+3} colourings of the extra vertices. So we have $6N_G \times n^3$ colourings.

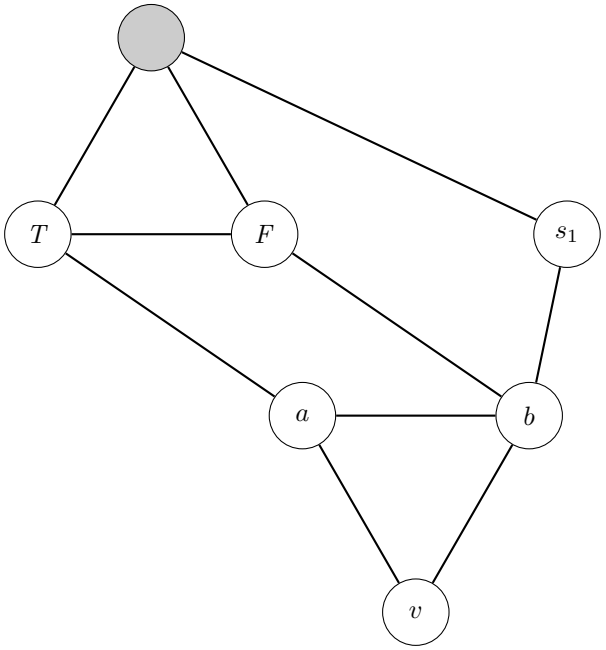
Note that G_1 has $2(n + 3)$ vertices, and so if we feed it into the oracle it will return *true* iff $6N_G \times n^3 \geqslant 3^{n+3}$. This will be the case so long as $N_G > 0$. So if the orcle returns *flse* we know that re no 3-colourings of G, and we'll return 0. Otherwise we want to find a colouring. We do so as follows:

For each vertex $v \in G$:
  For each colour c in {*red, true, false*}:
    "Colour" v to c by connecting it to the two opposing colours in the palette.
    So we'd colour v red, for example, by connecting it to the true and false nodes

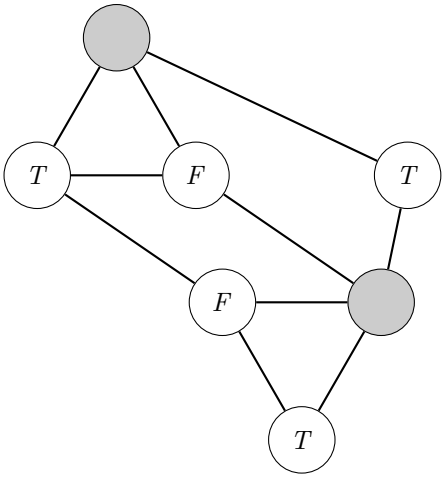
in the palette.
Feed the modified G_1 into the oracle.
If the oracle returns *yes*:
Record the colour and move to the next vertex (break the inner loop).
Otherwise, remove the colouring on v by removing the two new edges.

We can see that this process will allow us to find a colouring of G: We can show using induction that G remains 3-colourable after ever loop iteration, even with the colour retrictions that we place. This implies that one of the three colours will always work, as well, so we'll end up with a 3-colouring for G at the conclusion. Finally, we can see that there are a maximum of $3n$ loop iterations, and that each loop iteration involves adding (and possibly removing) two edges from G_1, feeding G_1 into the oracle, and recording colours as we go. So the whole process is polytime.

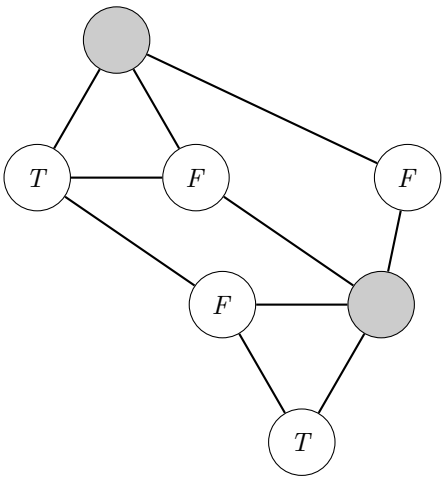
So if G is 3-colourable we can find a 3-colouring for G in polytime by using our oracle. Let's call this colouring C.
Now, consider the following widget:



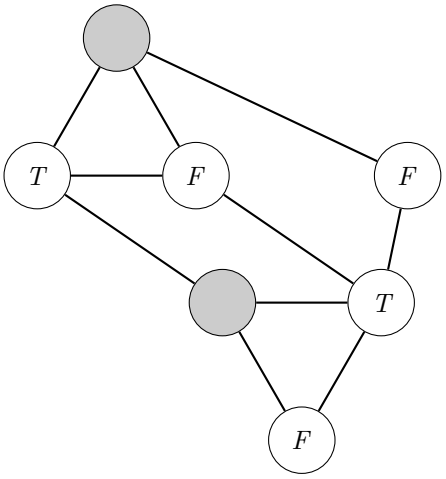
We see that we can colour the node s_1 as T or F. If it is coloured T the only way to colour the rest of the graph is:



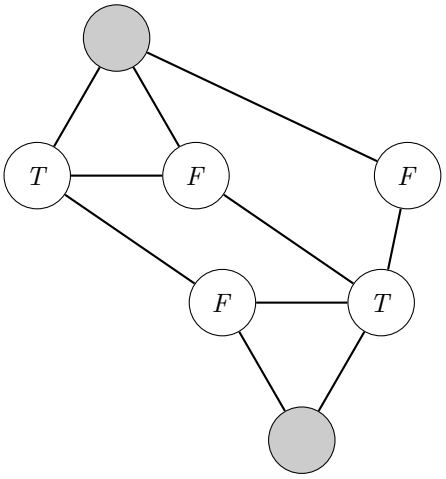
On the other hand, if we colour the node s_1 as F we hve three possible colourings:



and

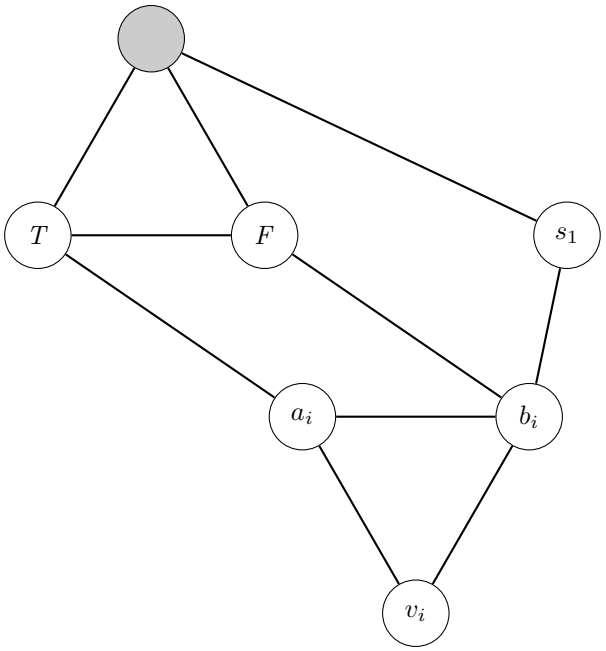


and



So the node labelled as v must be coloured T if s_1 is T, but if s_1 is F then there is exactly one colouring that sets v to each of the three colours - it can be set freely.

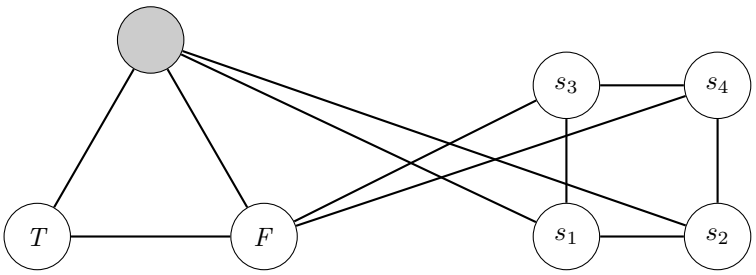
Now, consider the colouring C of G. In this colouring, some of the nodes are coloured T. For every such node v_i we can attach v_i to two new nodes, and attach everything to our s_1:



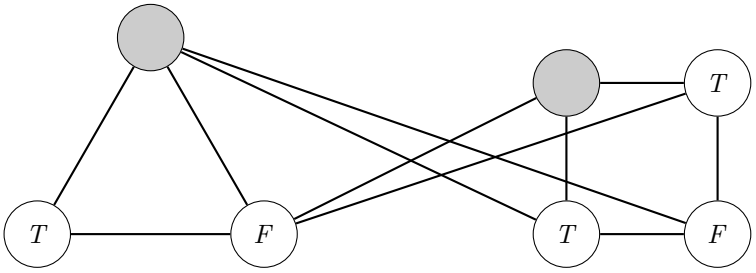
Note that the a_i and b_i nodes are different for every v_i, but the s_1 node is cxonstant.

If we do this, then when s_1 is set to T the nodes of G that are T under C will be forced to take the colour T, but if s_1 is F these nodes (and so all of G) can be coloured freely.

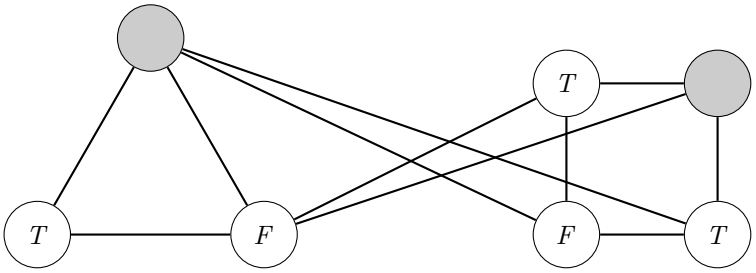
We'll repeat this pattern for the other two colours. We'll etend the s_1 switch to four nodes:



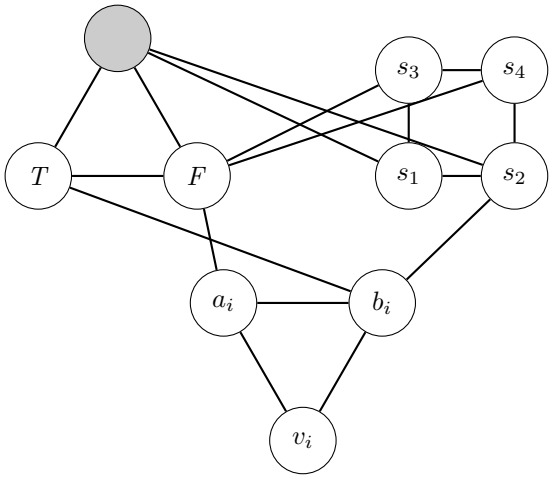
We can see that there re two posible colourings given the palette:



and

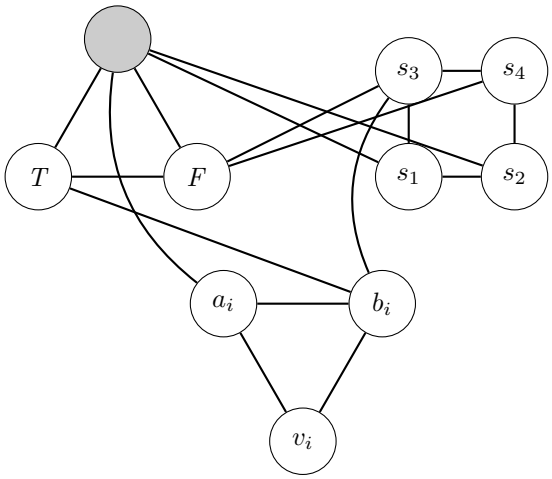


We can repeat the s_1 switch for the different colours using these new nodes:



The reader can verify that if s_1 is T, then v_i must be F, while if s_1 is F then v_i has one colouring for any value we choose. As before, we identify these v_i with the vertices of G that are F under the colouring C.

Next, we consider the widget:

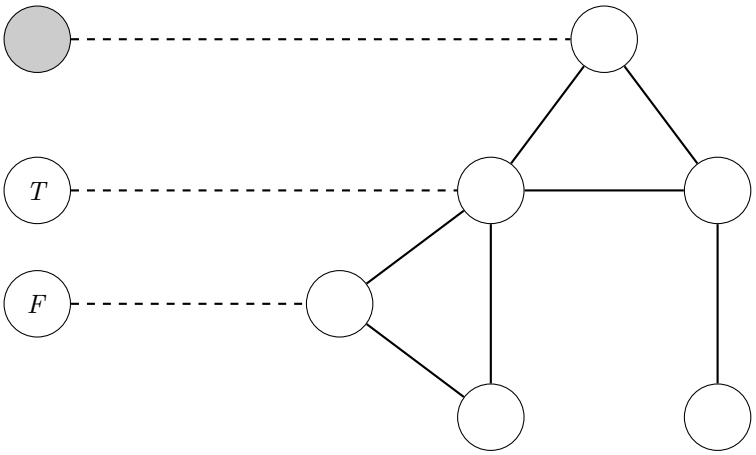


The reader can verify that if s_1 is T, then v_i must be *red*, while if s_1 is F then v_i has one colouring for any value we choose. As before, we identify these v_i with the vertices of G that are *red* under the colouring C.

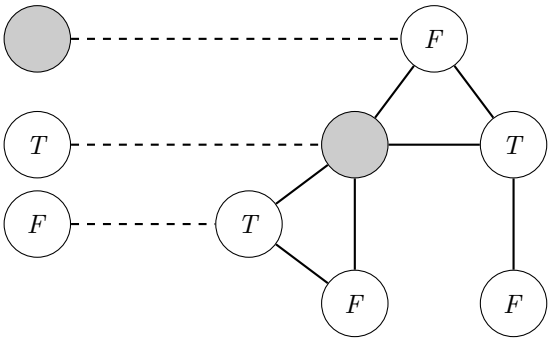
All told, this gives us a graph G' that has, if the palette is coloured in the way we have here, $1 + N_G$ colourings: one in which s_1 is T and N_G in which it is F.
But if we had a second graph H with a 3-colouring C', we could use the sametechnique to connect it to the s_1-switch in such a way that it would have one colouring if s_1 were F but could be coloured freely if s_1 were T. If N_H is the number of colourings of H, then the total number of colourings of G' given the palette would become $N_G + N_H$. If we let the palette take any colour, then, the total number of colourings would be $6(N_G + N_H)$.

> *In fact, we've assumed that G has at leat one edge. If H also has at least one edge then we can fix the two endpoints of this edge to the palette so that they always take the colours in their known colourings C and C'. Since every colouring of G and H is equivalent to such a colouring under a permutation of colours, and since there are six permutations of three colours, this modification reduces the number of colourings to $(N_G + N_H)$.*

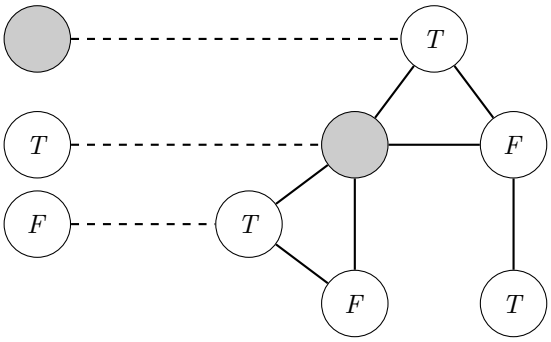
So the question is, what sort of graph should we use for H? Wht we'll do to nswer this is note that if we dapt the *or*-widget from our 3COL reduction into:



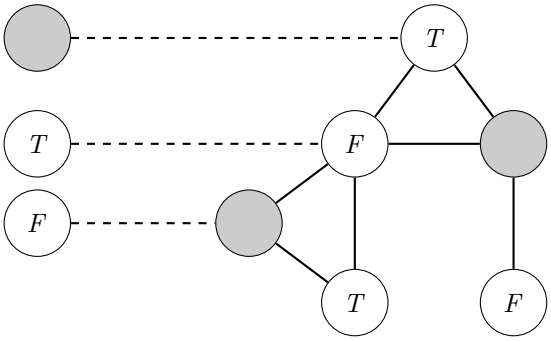
Observe that once the inputs are fixed, there is only one way to colour the remaining vertices:



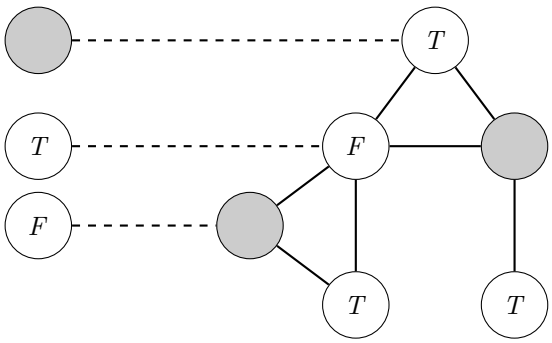
or



or



or



Also note that if a node u is either T or F, and if v connects to both u and *red*, then v takes the colour $\neg u$. So we can build *not*-gates and *or*-gates in 3COL. We can use these gates to encode a circuit into H that takes as input two $2n$-bit numbers x and y, and which outputs T iff $y < x$ when read as a binary number. We'll let H_i be the same graph with the x hard-coded to the number i (we can hard-code x by attaching each bit of its input to two out of three of the vertices in the palette). Note that each H_i has the same number of vertices.

> *Note that if $i > 0$ then H_i can always be coloured by setting its y to 0, and that in general H_i will have i colourings once the palette is fixed.*
>
> *Also note that any graph that reasonably lets the students choose the number of 3-colourings will be accepted here: this one is just quick if you know the or-gate trick.*

Let G'_i be the construction that we obtain if we attach G and H_i to the switch above, and note that G'_i has $(N_G + i)$ colourings. We note that we can construct this graph in time polynomial in n, and so it has number k of vertices that is polynomial in n.

Let G''_i be G'_i with k new vertices added. Of these new vertices, $k - n$ will be disconnected, and n will be connected to both the *red* and F vertices of the palette (so their colour is fixed). Note that G''_i has $2k$ vertices, so our oracle will return true iff it has at least 3^k colourings.

The total number of colourings for G''_i will be $3^{k-n} \times (N_G + i)$, and so our oracle will return true iff we choose an i such that $(N_G + i) \geqslant 3^n$. Note that this happens iff $(N_G + i) \geqslant 3^n$, which happens iff $(N_G + i) \geqslant 3^n$.

Note that the values of N_G can vary from 1 to $3^n - 1$ (we've dealt with the $N_G = 0$ and the $N_G = 3^n$ possibilities), and that the values of i can vary from 1 to $4^n - 1$ (we're using $2n$ bits in our register, so the maximum number is $2^{2n} - 1 = 4^n - 1$). For $n > 1$ we can see that $4^n - 1 > 3^n$, and so for any possible value of N_G we can find an i such that $(N_G + i) \geqslant 3^n$ and an i such that $(N_G + i) \leqslant 3^n$. In fact, since we have assumed that G has at least one edge, N_G is a multiple of 6, and so $3^n \leqslant 3^n - 3$. So we can in fact find an i such that $(N_G + i) < 3^n$

This means that we can run a binary search to find the smallest i_0 such that $(N_G + i_0) \geqslant 3^n$. For this i_0, we see that $(N_G + i_0) \geqslant 3^n$ and $(N_G + i_0 - 1) < 3^n$, and so $(N_G + i_0) = 3^n$. Hence, $N_G = 3^n - i_0$. If we return this i_0 we will have the number N_G of 3-colourings of G.

Now, if we run a binary search to find i, then the total number of steps we'll need to take for the search will be at most $\log_2(3^n) = \mathcal{O}(n)$. Each step of the search will require us to build a graph with $2k$ nodes, where k is the number of nodes in G'_i. Note that G'_i requires three nodes for the palette, four nodes for the switch, $3n$ nodes to copy G, and the nodes required to build H_i.

But we can also see that each H_i is an encoding of a binary comprison circuit uing $4n$ bits, nd that such a circuit can be built using a number of gates that is polynomial in n. We note that encoding our improved *or*-gate into 3COL requires us to add four nodes (plus the inputs), and the *not*-gate requires us to add one. If we use these gates to implement De Morgan's law we can build an *and*-gte with seven nodes, and so the total number of nodes we'll per gate in H_i is at

most seven. So H_i has a number of nodes that is polynomial in n, and therefore G_i' and G_i'' have sizes that are polynomial in n.

Since the vertices and edges of G_i' (and G_i'') can be determined in polynomial time, the total process taken in this procedure also takes polynomial time, as required.

■