Extra Notes: Polytime Reductions II

There are many interesting $\mathcal{NP}$-complete problems that we can find across a wide range of study areas. You've gotten a list of starter examples already, and here are a few more languages for you to play with.

Contents

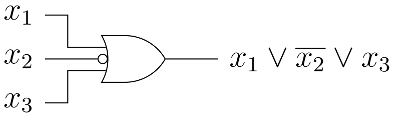
1 Another Useful 3SAT Lemma **1**
1.1 Embedding a 3-SAT Formula into a Grid 1

2 Examples **6**
2.1 Boolean Logic and Satisfiability 7
2.1.1 MAX-2SAT 7
2.2 Mathematical Programming 9
2.2.1 0-1-INTEGER PROGRAMMING 9
2.3 Graph Theory 11
2.3.1 UHAM-PATH 11
2.3.2 UHAM-CYCLE 14
2.3.3 MAX-CUT 16
2.3.4 INDUCED-PATH 19
2.4 Sets, Partitions, and Orderings 22
2.4.1 PARTITION 22
2.4.2 X3C 25
2.4.3 BETWEENNESS 26
2.4.4 GROUPING-BY-SWAPPING 28
2.5 Timetables and Scheduling 34
2.5.1 TIMETABLE-SCHEDULING 34
2.5.2 OPEN-SHOP-SCHEDULING 42
2.6 Puzzles and Games 47
2.6.1 CROSSWORD 47
2.6.2 NONOGRAM 52
2.6.3 SPIRAL-GALAXIES 62

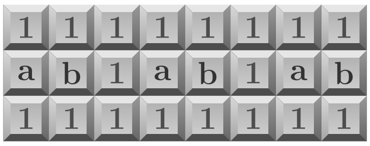
1 Another Useful 3SAT Lemma

1.1 Embedding a 3-SAT Formula into a Grid

A number of our reductions (in particular in the Games section) are originally reduced from some version of 3SAT, where we view the 3CNF ϕ as defining a circuit. That is, we can view the single clause $(x_1 \vee \overline{x_2} \vee x_3)$ as the following gate:



Most of these reductions are straightforward in their constructions – we simply need to build widgets to encode the different parts of the circuit (e.g., the wires and gates) using the rules of the game we are working with. As an example of what we mean by this, consider the game of Minesweeper. In (Kaye, 2000) we see the following construction:



1	1	1	1	1	1	1	1
a	b	1	a	b	1	a	b
1	1	1	1	1	1	1	1

Now, if we see this pattern in a Minesweeper game – and if we are able to solve said game at all – we will have either a solution in which all of the tiles marked "a" are mined and the tiles marked "b" are not, or we will have the opposite: a solution in which all of the tiles marked "b" are mined and the tiles marked "a" are not. We can think of this construction as a sort of wire, then, that transmits a signal down its length: either the "a"s or the "b"s are mined. This signal is binary, and so can be used to represent the truth value for a binary variable.
Given a ciruit diagram for a 3CNF ϕ, we can build a Minesweeper in which we replace the wires of the circuit with the pattern we see above. If we can similarly replace the other elements of the circuit with, say, Minesweeper patterns, then we could hope to be able to take any circuit and replace its elements with an appropriate set of Minesweeper widgets.

> *In general, we'll set up our circuit so that its output has to be* true. *So the game will be solvable iff there is a satisfying truth assignment for the variables in the circuit. We'd end up with a game that would have a solution iff our original ϕ could be satisfied.*

In order to get this to work we need to be able to encode the wires of our circuit, but we'd also need a few more widgets – we'd need:

- Endpoints for the wires: we should be able to emulate a signal input in which the wire can either be set to *true* or *false*, and we'd also want to be able to set up inputs in which the signal is hard-coded to be one of the two values.

  In the above example, and in many of the reductions we see, the wire widget is symmetric. So we can build a wire output by taking an input widget and flipping it around.

- *Not* gates.

- *Or* gates.

  If we have not *gates and* or *gates, we can also simulate* and *gates, and likewise if we can code an* and *gate and a* not *gate we can also simulate an* or*gate. Basically, we just need to be able to simulate any complete set of connectives.*

We'll also need to worry about aligning and connecting all of our widgets. So in addition to the above widgets, we'll also need to encode mechanismes that allow us to:

- Bend wires – we'll sometimes need to change directions.

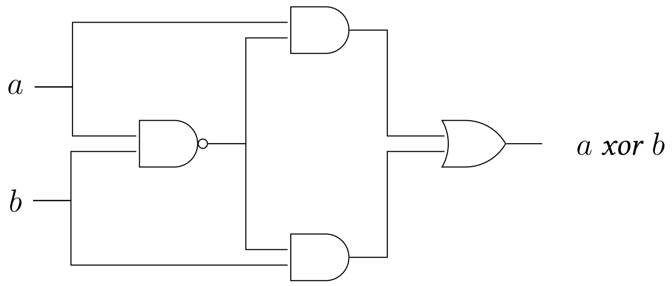
- Shift wires – in the above wire pattern for Minesweeper we have a pattern that takes 3×3 squares. If we need to shift it up, down, or sideways by one squre to get it to fit into the rest of the circuit we should be able to do so.

- Split wires – In general the variables in our 3SAT instances are used more than once in the formula, and so we have to be able to connect our wires to multiple targets.

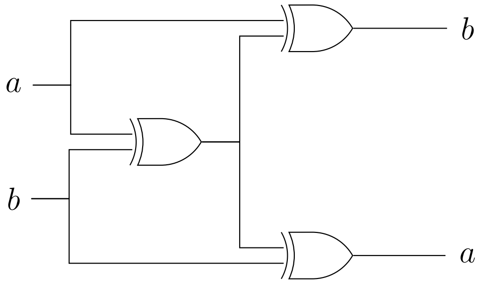
Finally, we need to be able to have wires cross over each other.

> *In many reductions in the literature the need for this particular widget is avoided by reducing from a* 3SAT *variant whose instances form planar ciruits, but the reduction from (Kaye, 2000) shows us that we really don't need the special* 3SAT *variant in the first place – if we have the rest of the widgets we can also simulate a crossover widget as well.*

Let's see how we can build a crossover widget from the other pices we've listed. Firstly, we can use multiple *not* and *and* widgets to build a planar *xor* circuit:

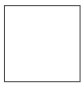


We can then use a series of these *xor* gates to program a planar crossover widget:



So let's suppose that we know how to build these widgets in, say, Minesweeper. We'll also want to be able to set up areas of the game board that have no components in them – this is easy in Minesweeper, since we can just place no mines anywhere but where we build our components. We can then build a set of tiles with the following patterns:

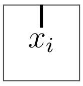
Empty Squares:



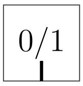
Straight wires:

Bent wires:

Signal inputs:

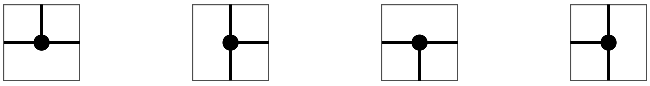


Signal outputs:

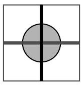


Note that we can force these outputs to be 1 *or* 0*, or we can allow them to take either value.*

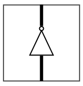
Signal splitters:



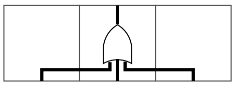
Wire crossovers:



Not gates:

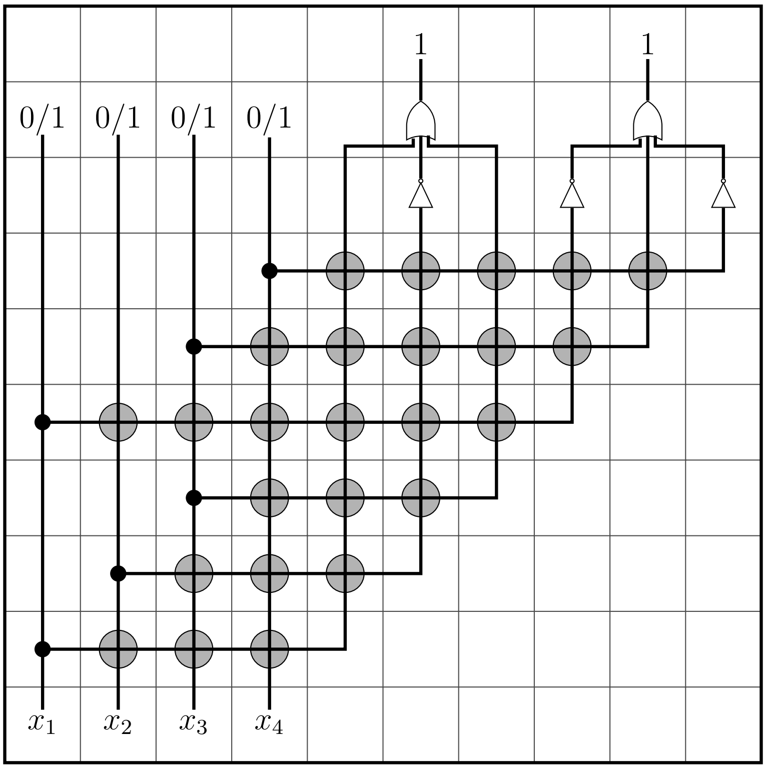


Or gates:



We're going to set up these tiles so that each square is the same size (e.g., takes up the same number of Minesweeper squares). We'll require that the wires enter and leave the tiles in the center of the square, and that they do so at the beginning/end of the repeated pattern that we use to transmit the signal over the wire. It is clear that if we can shift the wires up, down, or to the sides by an arbitrary number of cells then we can ensure this for suitably large tiles.

Once we've set up the tiles to these specifications, we can line them up to form circuits, and in particular, 3CNF circuits. For example, if we have a 3CNF formula $\phi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4)$, then we can write the circuit like so:



That is, suppose that ϕ is any 3CNF formula with n clauses and k variables. In order to embed ϕ into a grid by using these tiles, we follow these steps:

1) We start by setting up $k+3n$ columns in our grid. On the bottom row, the first k columns contain an input for each of the k variables. The rest of the row is empty. So the signal for x_i can be found on column i of our grid.

2) We build another row for our grid such that:

    - The first k columns have a $0/1$ output (an output that can take either value). These outputs will terminate the signals for the x_i wires.
    - The remaining $3n$ columns will contain n *or* widgets. These will correspond to the clauses of ϕ, and in general clause C_j of ϕ will be found from columns $k+3j-2$ to $k+3j$. For $a \in \{1,2,3\}$, the input a for clause C_j will be found in the column $k+3j+a-3$.

    This row will be the second-from-top row of our grid.

3) In the top row of our grid, we'll attach a 1 output to the outputs of every *or* widget. Every other column will be empty.

4) For each $1 \leqslant j \leqslant n$ and $1 \leqslant a \leqslant 3$, we will insert a row at height $1+3j+a-3$ into our grid by using the following steps:

    - Find the variable x_i corresponding to the input a of clause C_j. (note that the literal may be negated, but the variable is not).
    - Add a right-facing splitter to column i of our new row.
    - For each column between column i and column $k+3j+a-3$:

    - If there is an output on the top of the tile in the current column of row $3j+a-3$ (i.e., one tile below), then add a crossover tile.
    - Otherwise, add a horizontal wire.
  - Add an upwards-bending wire in column $k+3j+a-3$.
  - For each remaining column
    - If there is an output on the top of the tile in the current column of row $3j+a-3$, then add an upwards-facing wire.
    - Otherwise, add an empty cell.
- We will add one more row to our grid at height $2+3n$ in which a *not*-gate is added to the column of every negated literal. Every other column has an upwards-facing wire.

Now, we can see that we can follow all of these steps in polynomial time in the size of the grid, and that the grid required to encode this circuit has a height of $4+3n$ and a width of $k+3n$. So the total grid size is $C(9n^2+3nk+12n+4k)$, where C is the size of the square needed to encode our tiles. We can therefore conclude:

Lemma 1.1 *Suppose that we have tiles for the following circuit components:*

- *Straight and bent wires.*
- *Signal inputs and outputs.*
- *Signal splitters.*
- *Signal crossovers.*
- *Not gates.*
- *Or gates.*

  Note that the Or gates will have three inputs and one output.

Suppose further that we can set these components into square tiles of a set size, with the or gates taking three tiles, and that the wires enter and leave at the center of the tiles.

Then for any 3CNF ϕ with n clauses and k variables we can encode, in a grid of size $\mathcal{O}(n^2+nk)$, a circuit representing ϕ. If each tile can be printed in a constant time, then the whole grid will also be printable in polynomial time.

Moreover, if we have every tile except for the crossover tile, we can build a pattern simulating the crossover. By padding all of our tiles with extra blank tiles we will still be able to embed ϕ into a grid, and the overall size of the grid will still be $\mathcal{O}(n^2+nk)$.

□

2 Examples

The examples in the following section will be reduced from either other languages in this section, or from languages in our first polytime reduction handout. As a reminder, the languages in the previous handout were: 3SAT,SAT, X1-3SAT, NAE-3SAT, CLIQUE, INDEPENDENT-SET, VERTEX-COVER, FEEDBACK-ARC-SET, HAM-PATH, HAM-CYCLE, CUBIC-SUBGRAPH, 3COL, PARTITION-INTO-TRIANGLES, 3DM, and SUBSET-SUM.

2.1 Boolean Logic and Satisfiability

2.1.1 MAX-2SAT

Definition of MAX-2SAT (from (Garey et al., 1974), via (Garey and Johnson, 1979)):

> **INPUT:** A 2CNF ϕ with k variables and n clauses, a positive integer $m \leqslant n$.
>
> **QUESTION:** Does ϕ have a truth assignment that satisfies at least m clauses?

Reduced from: 3SAT.

Note that the subproblem in which $m = n$ is in $\mathcal{P}$!

Firstly, we can see 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. $count = 1$.
     $\mathcal{O}(1)$ time
  2. For $i = 1$ to $n$ (where $n$ is the number of clauses in $\phi$):
     $\mathcal{O}(n)$ iterations.
  3.    Use $\tau$ to assign truth values to the literals in the clauses of the clause $c_i$.
     $\mathcal{O}(n)$ time per iteration.
  4.    If the clause $c_i$ is satisfied, $count += 1$.
     $\mathcal{O}(1)$ time per iteration after amortization.
  5. If $count \leqslant m$, reject.
     $\mathcal{O}(n)$ 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 ϕ.

So MAX-2SAT is in $\mathcal{NP}$.

Reduction from 3SAT:

Suppose we are given a 3SAT instance $\langle \phi \rangle$, where ϕ has n clauses and k variables.

For each clause $c_i = (\ell_{i,1}, \ell_{i,2}, \ell_{i,3})$ of ϕ, we add one variable d_i, and output the following nine clauses, which we'll refer to as $c'_{i,j}$, for $1 \leqslant j \leqslant 10$:

$c_{i,1}$	$(\ell_{i,1} \vee \ell_{i,1})$	$c_{i,2}$	$(\ell_{i,2} \vee \ell_{i,2})$	$c_{i,3}$	$(\ell_{i,3} \vee \ell_{i,3})$	$c_{i,4}$	$(\neg\ell_{i,1} \vee \neg\ell_{i,2})$	$c_{i,5}$	$(\neg\ell_{i,1} \vee \neg\ell_{i,3})$
$c_{i,6}$	$(\neg\ell_{i,2} \vee \neg\ell_{i,3})$	$c_{i,7}$	$(\ell_{i,1} \vee \neg d_i)$	$c_{i,8}$	$(\ell_{i,2} \vee \neg d_i)$	$c_{i,9}$	$(\ell_{i,3} \vee \neg d_i)$	$c_{i,10}$	$(d_i \vee d_i)$

We'll refer to these clauses as .

For every clause c_i in ϕ we add each of these ten clauses to ϕ', and we set $m' = 7n$.

To see why this works, consider the following truth table for the clauses:

$(\ell_{i,1}, \ell_{i,2}, \ell_{i,3})$	d_i	$c'_{i,1}$	$c'_{i,2}$	$c'_{i,3}$	$c'_{i,4}$	$c'_{i,5}$	$c'_{i,6}$	$c'_{i,7}$	$c'_{i,8}$	$c'_{i,9}$	$c'_{i,10}$	# Satisfied
(T, T, T)	T	T	T	T	F	F	F	T	T	T	T	7
(T, T, T)	F	T	T	T	F	F	F	T	T	T	F	6
(T, T, F)	T	T	T	F	F	T	T	T	T	F	T	7
(T, T, F)	F	T	T	F	F	T	T	T	T	T	F	7
(T, F, T)	T	T	F	T	T	F	T	T	F	T	T	7
(T, F, T)	F	T	F	T	T	F	T	T	T	T	F	7
(T, F, F)	T	T	F	F	T	T	T	T	F	F	T	6
(T, F, F)	F	T	F	F	T	T	T	T	T	T	F	7
(F, T, T)	T	F	T	T	T	T	F	F	T	T	T	7
(F, T, T)	F	F	T	T	T	T	F	T	T	T	F	7
(F, T, F)	T	F	T	F	T	T	T	F	T	F	T	6
(F, T, F)	F	F	T	F	T	T	T	T	T	T	F	7
(F, F, T)	T	F	F	T	T	T	T	F	F	T	T	6
(F, F, T)	F	F	F	T	T	T	T	T	T	T	F	7
(F, F, F)	T	F	F	F	T	T	T	F	F	F	T	4
(F, F, F)	F	F	F	F	T	T	T	T	T	T	F	6

So if at least one of $\ell_{i,1}$, $\ell_{i,2}$, or $\ell_{i,3}$ is satisfied, we can set d_i so that seven of these clauses will also be satisfied. If none of $\ell_{i,1}$, $\ell_{i,2}$, or $\ell_{i,3}$ is satisfied, the best we can do is six. Furthermore, we can never satisfy more than seven of these ten clauses at a time.

We claim that $\langle \phi', m' \rangle \in$ MAX-2SAT iff $\langle \phi \rangle \in$ 3SAT:

> Suppose $\langle \phi \rangle \in$ 3SAT.
>
> Then there is some truth assignment τ that satisfies every clause in ϕ.
>
> Since, for any clause c_i, there is at least one literal $\ell_{i,1}$, $\ell_{i,2}$, or $\ell_{i,3}$ that is satisfied, we can set d_i so that seven of the clauses $c_{i,j}$ are satisfied (as per our discussion above).
>
> Since this is true for every clause, and since we can set the d_i independently, we can extend τ to the d_i to get a τ' such that seven out of ten of all of the $c_{i,j}$ are satisfied.
>
> Since there are n values that i can take, and since we satisfy seven clauses per value of i, we can see that τ' satisfies $7n = m$ clauses in ϕ'.
>
> $\Rightarrow \langle \phi', m' \rangle \in$ MAX-2SAT.
>
> Suppose $\langle \phi', m' \rangle \in$ MAX-2SAT.
>
> Then there is a truth assignment τ' that satisfies $m = 7n$ clauses in ϕ'.
>
> Let τ be the truth assignment we get if we restrict τ' to the variables in ϕ (so it's τ' without the d_is).
>
> We can see that ϕ' has $10n$ clauses: one set of ten $c_{i,j}$ for every clause c_i in ϕ.
>
> Furthermore, none of these sets of ten $c_{i,j}$ can ever have more than seven clauses satisfied at once, as per our discussion above.

So if any one of these sets had fewer than seven clauses satisfied, we would not be able to satisfy all $7n$ clauses.

Therefore, for every i, seven of the clauses $c_{i,j}$ must be satisfied.

As we saw in our table above, though, this can onl happen if at least one of the literals $\ell_{i,1}$, $\ell_{i,2}$, or $\ell_{i,3}$ of c_i is satisfied.

So τ satisfies c_i.

Since this is true for every clause c_i, τ satisfied ϕ.

$\Rightarrow \langle \phi \rangle \in$ 3SAT.

Finally, we can see that we have simply written our a 2CNF with $10n$ clauses, where each clause can be determined in polynomial time given the corresponding clause in ϕ. So this is a polytime reduction.

So MAX-2SAT is $\mathcal{NP}$-complete, as required.

■

2.2 Mathematical Programming

2.2.1 0-1-INTEGER PROGRAMMING

Definition of 0-1-INTEGER PROGRAMMING *(as defined in (KARP, 1972))*:

Instance: An $m \times n$ matrix C of integers and an integer vector $\overline{d}$.

Question: Is there a 0-1 vector $\overline{x}$ such that $C\overline{v} = \overline{d}$?

Reduced from: 3SAT.
Note: This is one of the original 21 $\mathcal{NP}$-complete problems from (KARP, 1972).

Firstly, we can see that this is in $\mathcal{NP}$:

1) It is a yes/no problem.
2) Its certificate is the 0-1 vector $\overline{v}$.
3) The size of this certificate is $\mathcal{O}(n)$, where n is the number of columns in C. This certificate is polynomial in the size of C.
4) A verifier for this certificate would be:

```
Let $\mathcal{V}$ ="On $\langle C, \overline{d}, \overline{v} \rangle$:"
  1. Reject if any of the following checks fails:
     $\mathcal{O}(1)$ time.
  2. Check that $\overline{v}$ is a length-$n$ 0-1 vector.
     $\mathcal{O}(n)$ time.
  3. Check that $C\overline{v} = \overline{d}$.
     $\mathcal{O}(mn^2)$ time.
  4.Accept.
     $\mathcal{O}(1)$ time.
```

5) We can see that this verifier runs in $\mathcal{O}(mn^2)$ time, and so is polytime in the size of $\langle C, \overline{d}\rangle$.

So 0-1-INTEGER PROGRAMMING is in $\mathcal{NP}$.

Reduction from 3SAT:

Suppose we are given a 3SAT instance ϕ with ℓ variables and k clauses. Note that we can assume using our Helpful 3SAT Lemma from the previous handout that the clauses in ϕ each have three distinct variables.

We'll set the vector $\overline{d}$ to be an $\ell + k$-dimensional vector $[\underbrace{1, 1, \ldots, 1}_{\ell \text{ times}}, \underbrace{3, 3, \ldots 3}_{k \text{ times}}]^T$.

The matrix C will be a size $(\ell + k) \times 2(\ell + k)$ matrix, where the j^{th} row $C_{j,\cdot}$ of C is as follows:

- If $1 \leqslant j \leqslant \ell$, then $C_{j,2j-1} = C_{j,2j} = 1$. All other entries in this row are 0.

  You can see that since every element of $\overline{v}$ must be 0 or 1, exactly one of the elements v_{2j-1} and v_{2j} must be 1, and the othe must be 0. We'll use this to represent the variables x_i from ϕ: If $v_{2i-1} = 1$, this corresponds to $x_i = T$. Similarly, if $v_{2i} = 1$, this corresponds to $x_i = F$. So there's a one-to-one correspondence between the truth assignments to ϕ and the ways of setting the first 2ℓ elements of $\overline{v}$ to satisfy the top ℓ rows of $\overline{d}$.

- If $j > \ell$, let $j' = j - \ell$ and consider the clause j' of ϕ.

  If x_i occurs as a variable in this clause, we set $C_{j,(2i-1)} = 1$.

  If $\neg x_i$ occurs as a variable in this clause, we set $C_{j,2i} = 1$.

  We always set the elements $C_{j,2\ell+(2j'-1)}$ and $C_{j,2\ell+2j'} = 1$.

  Note that these are the columns not corresponding to any of the x_i or $\neg x_i$. They're the slack variables for clause j'. Note that every slack variable is used in exactly one row of C.

  All other entries in this row are set to 0.

This gives us the matrix C, and so we can return the output $\langle C, \overline{d}\rangle$.

We argue that $\langle C, \overline{d}\rangle \in$ 0-1-INTEGER PROGRAMMING$\langle \phi\rangle \in$ 3SAT:

Suppose $\langle \phi\rangle \in$ 3SAT.

Then it has a satisfying truth assignment τ.

Recall from the above argument that we can associate tau with an assignment to the first 2ℓ rows of $\overline{v}$ that will satisfy the top ℓ rows of $C\overline{v} = \overline{d}$.

For the remaining row, note that for any $\overline{v}$ that satisfies the first ℓ rows of $C\overline{v} = \overline{d}$ corresponds to a truth assignment τ, and so if $j = \ell + j'$ the product $C_{j,\cdot} \cdot \overline{v}$ gives the number of literals in the clause j' that are satisfied by τ, plus the number of slack variables we set to true.

Since there are three literals per clause, the partial sum due to the literals will be between 1 and 3 if the clause is satisfied, and will be 0 if it is not.

So there will be some way of setting the slack variables to make the sum go to 3 iff the clause is satisfied, but it will be at most 2 if the clause is not satisfied.

Since there are separate slack variables for every clause, we can repeat this process for each clause.

So there is some way of setting the slack variables in $\overline{v}$ so that every row of $C\overline{v} = \overline{d}$ is satisfied.
$\Rightarrow \langle C, \overline{d} \rangle \in$ 0-1-INTEGER PROGRAMMING.

Suppose $\langle C, \overline{d} \rangle \in$ 0-1-INTEGER PROGRAMMING.

Then there is some 0-1 vector $\overline{v}$ that satisfies $C\overline{v} = \overline{d}$.

As have have previously discussed, the first 2ℓ elements of $\overline{v}$ must correspond to a truth assignment τ for ϕ.

If we consider the clause j' of ϕ, we see that the j^{th} row of $C\overline{v}$ is the number of literals in the clause j' that are satisfied by τ, plus the number of slack variables we set to true.

Since there are no more than two slack variables per clause, at least one literal must be true, and so this clause is satisfied by τ.

Since this is true for every clause, ϕ is satisfied by τ.
$\Rightarrow \langle \phi \rangle \in$ 3SAT.

Finally, we can see that this reduction simply builds a size-$(\ell + k) \times 2(\ell + k)$ matrix C and a size-$(\ell + k)$ vector $\overline{d}$. So the size of the output is polynomial in the size of ϕ. Since each element of C and $\overline{d}$ can also be founf in a time polynomial in the size of ϕ, we can see that this entire reduction is polytime.

So 0-1-INTEGER PROGRAMMING is $\mathcal{NP}$-complete, as required.

■

2.3 Graph Theory

2.3.1 UHAM-PATH

Definition of UHAM-PATH *(Undirected Hamiltonian Path)*:

Instance: A graph $G = (V, E)$ with specified vertices s and t.

Question: Does there exist a Hamiltonian path in G from s to t (i.e., a path from s to t that passes through every $v \in V$ exactly once)?

Reduced from: HAM-PATH.

Firstly, we can see that this is in $\mathcal{NP}$:

1) It is a yes/no problem.

2) Its certificate is a Hamiltonian 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 $\mathcal{O}(n)$, 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 exactly $n = |V|$ distinct elements.
     $\mathcal{O}(n^2)$ time.
  3. For $i = 1$ to $n - 1$:
     $\mathcal{O}(n)$ 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 $P[n] = t$.
     $\mathcal{O}(1)$ 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 UHAM-PATH is in $\mathcal{NP}$.

Reduction from HAM-PATH:

Suppose we are given an instance $\langle G = (V, E), s, t \rangle$ of HAM-PATH.

We'll adapt G to a new undirected graph $G' = (V', E')$ as follows:

- For every node $v_i \notin \{s, t\}$ in V we'll create three nodes on V': $v_{i,\text{in}}$, $v_{i,\text{mid}}$, and $v_{i,\text{out}}$.

  We'll add the edges $\{v_{i,\text{in}}, v_{i,\text{mid}}\}$ and $\{v_{i,\text{mid}}, v_{i,\text{out}}\}$ to E'. These will be the only edges connecting to $v_{i,\text{mid}}$.

  As a result, any Hamiltonian path in G' will have to pass through $v_{i,\text{mid}}$, and so (if none of these vertices is s' or t') must have a subpath of the form $(v_{i,\text{in}}, v_{i,\text{mid}}, v_{i,\text{out}})$ (or its inverse).

- We'll add the edges s' and t' to V'. These will be the start and end points of our Hamiltonian path (if one exists).

  If there is an edge $(s, t) \in E$ we add an edge $\{s', t'\}$ to E' (although we won't use that edge except in the case of a two-node graph).

- For every $v_i \neq s, t$ in V, if there is an edge (s, v_i) in E, we add the edge $\{s', v_{i,\text{in}}\}$ to E'.

- For every $v_i \neq s, t$ in V, if there is an edge (v_i, t) in E, we add the edge $\{v_{i,\text{out}}, t'\}$ to E'.

- For every $v_i \neq s, t$ and $v_j \neq s, t$ in V, if there is an edge (v_i, v_j) in E, we add the edge $\{v_{i,\text{out}}, v_{j,\text{in}}\}$ to E'.

We argue that $\langle G' = (V', E'), s', t' \rangle \in$ UHAM-PATH iff $\langle G = (V, E), s, t \rangle \in$ HAM-PATH:

Suppose $\langle G = (V, E), s, t\rangle \in$ HAM-PATH.

Then there is a Hamiltonian path $P = (v_{i_1} = s, v_{i_2}, \ldots, v_{i_n} = t)$ in G.

We let $P' = (s', v_{i_2,\text{in}}, v_{i_2,\text{mid}}, v_{i_2,\text{out}}, \ldots, v_{i_{n-1},\text{in}}, v_{i_{n-1},\text{mid}}, v_{i_{n-1},\text{out}}, t')$.

Clearly, P' covers all of the nodes in V', starts at s', and ends at t', and so it is a Hamiltonian path iff every pair of sequential nodes in P' has an edge between them. But we can see:

- For every v_{i_k}, $1 < k < n$, the edges $\{v_{i_k,\text{in}}, v_{i_k,\text{mid}}\}$ and $\{v_{i_k,\text{mid}}, v_{i_k,\text{out}}\}$ are in E'.
- We know the edge $(s, v_{i_2}) \in E$, since P is a Hamiltonian path. So $\{s', v_{i_2,\text{in}}\} \in E'$.

  Similarly, the edge $(v_{i_{n-1}}, t) \in E$, since P is a Hamiltonian path. So $\{v_{i_{n-1},\text{out}} t'\} \in E'$.
- Finally, we can see that if $2 \leqslant k < k + 1 < n$, then $(v_{i_k}, v_{i_{k+1}}) \in E$, and so $\{v_{i_k,\text{out}}, v_{i_{k+1},\text{in}}\} \in E'$.

So P' is a Hamiltonian path from s' to t' in G'.

$\Rightarrow \langle G' = (V', E'), s', t'\rangle \in$ UHAM-PATH.

Suppose $\langle G' = (V', E'), s', t'\rangle \in$ UHAM-PATH.
Then there is a Hamiltonian path P' from s' to t' in G'.

Since P' starts at s', its second vertex must be some $v_{i_2,\text{in}} \in V'$.

Unless $|V| = 2$, in which case it goes directly to t'. But this only happens if $|V| = 2$.

We can see that (s, v_{i_2}) is an edge in E, since $\{s', v_{i_2,\text{in}}\} \in E'$.

By our discussion above, P' must contin the subpath $(v_{i_2,\text{in}}, v_{i_2,\text{mid}}, v_{i_2,\text{out}})$.

By repeating the same argument, we see that $v_{i_2,\text{out}}$ must connect to some $v_{i_3,\text{in}}$ in P' (or to t').

As before, (v_{i_2}, v_{i_3}) must be an edge in E. Moreover, $v_{i_3,\text{in}}$ must be followed be $v_{i_3,\text{mid}}$ and $v_{i_3,\text{out}}$.

Continuing inductively we see that we can obtain from P' a sequence of vertices $P = (v_{i_1} = s, v_{i_2}, v_{i_3}, \ldots, v_{i_r} = t)$ in G, where $(v_{i_k}, v_{i_{k+1}}) \in E$ for all $1 \leqslant k < r$. That is, P is a path in G from s to t.

Note that the vertices $v_{i_k,\text{in}}$, $v_{i_k,\text{mid}}$, and $v_{i_k,\text{out}}$ must always occur contiguously in P', and so P is, in fact, a simple path.

But the only way that we can get to t in P is after we've exhausted all of the vertices in P', and thus all of the vertices in V' as well.

This means that we have to have included the vertices of the form $v_{i,\text{in}}$ in V' in our costruction of P, and so every vertex $v_i \neq s, t$ occurs in P.

So P is a Hamiltonian path from s to t in G.

$\Rightarrow \langle G = (V, E), s, t\rangle \in$ HAM-PATH.

Finally, we can see that when constructing G', $|V'| < 3|V|\mathcal{O}(|V|)$, and $|E'| < |E| + 2|V| = \mathcal{O}(|V| + |E|)$. Since we can decide what these vertices are and what their edges are in polynomial time, the entire reduction is polytime.

So UHAM-PATH is $\mathcal{NP}$-complete, as required.

■

2.3.2 UHAM-CYCLE

Definition of UHAM-CYCLE *(Undirected Hamiltonian Cycle)*:

> **Instance:** A graph $G = (V, E)$.
>
> **Question:** Does there exist a Hamiltonian cycle in G (i.e., a cycle that passes through every $v \in V$ exactly once)?

Reduced from: HAM-CYCLE.

Note: This is one of the original 21 $\mathcal{NP}$-complete problems from (KARP, 1972).

Firstly, we can see that this is in $\mathcal{NP}$:

1) It is a yes/no problem.
2) Its certificate is a Hamiltonian cycle C (we can treat this as a list of the vertices of V).
3) The size of this certificate is $\mathcal{O}(n)$, 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 $\langle G = (V, E), C\rangle$:"
  1. Reject if any of the following checks fails:
     $\mathcal{O}(1)$ time.
  2. Check that $C$ has exactly $n = |V|$ distinct elements.
     $\mathcal{O}(n^2)$ time.
  3. For $i = 1$ to $n - 1$:
     $\mathcal{O}(n)$ 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 $\{P[n], P[1]\} \in E$.
     $\mathcal{O}(n)$ or $\mathcal{O}(1)$ time, depending on your implementation.
  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 UHAM-CYCLE is in $\mathcal{NP}$.

Reduction from HAM-CYCLE:

Suppose we are given an instance $\langle G = (V, E)\rangle$ of HAM-CYCLE.

We'll adapt G to a new undirected graph $G' = (V', E')$ as follows:

- For every node $v_i \in V$ we'll create three nodes on V': $v_{i,\text{in}}$, $v_{i,\text{mid}}$, and $v_{i,\text{out}}$.

  We'll add the edges $\{v_{i,\text{in}}, v_{i,\text{mid}}\}$ and $\{v_{i,\text{mid}}, v_{i,\text{out}}\}$ to E'. These will be the only edges connecting to $v_{i,\text{mid}}$.

  As a result, any Hamiltonian cycle in G' will have to pass through $v_{i,\text{mid}}$, and so must have a subpath of the form $(v_{i,\text{in}}, v_{i,\text{mid}}, v_{i,\text{out}})$ (or its inverse).

- For every v_i and $v_j \in V$, if there is an edge (v_i, v_j) in E, we add the edge $\{v_{i,\text{out}}, v_{j,\text{in}}\}$ to E'.

We argue that $\langle G' = (V', E') \rangle \in$ UHAM-CYCLE iff $\langle G = (V, E) \rangle \in$ HAM-CYCLE:

Suppose $\langle G = (V, E) \rangle \in$ HAM-CYCLE.

Then there is a Hamiltonian cycle $C = (v_{i_1}, v_{i_2}, \ldots, v_{i_n})$ in G.

We let $C' = (v_{i_1,\text{in}}, v_{i_1,\text{mid}}, v_{i_1,\text{out}}, v_{i_2,\text{in}}, v_{i_2,\text{mid}}, v_{i_2,\text{out}}, \ldots, v_{i_n,\text{in}}, v_{i_n,\text{mid}}, v_{i_n,\text{out}})$.

Clearly, C' covers all of the nodes in V', and so it is a Hamiltonian cycle iff every pair of sequential nodes in C' has an edge between them. But we can see:

- For every v_{i_k}, $1 \leqslant k \leqslant n$, the edges $\{v_{i_k,\text{in}}, v_{i_k,\text{mid}}\}$ and $\{v_{i_k,\text{mid}}, v_{i_k,\text{out}}\}$ are in E'.
- We know the edge $(v_{i_n}, v_{i_1}) \in E$, since C is a Hamiltonian cycle. So $\{v_{i_n,\text{out}}, v_{i_1,\text{in}}\} \in E'$.
- Finally, we can see that if $1 \leqslant k < k+1 < n$, then $(v_{i_k}, v_{i_{k+1}}) \in E$, and so $\{v_{i_k,\text{out}}, v_{i_{k+1},\text{in}}\} \in E'$.

So C' is a Hamiltonian cycle in G'.

$\Rightarrow \langle G' = (V', E') \rangle \in$ UHAM-CYCLE.

Suppose $\langle G' = (V', E'), s', t' \rangle \in$ UHAM-CYCLE.
Then there is a Hamiltonian cycle C' in G'.

We know that C' must pass through $v_{i_1,\text{in}}$, and so wlog. we will treat this as its first vertex (we can just cycle through a list of the elements in C' to get this).

By our arguments above we also know that $\{v_{i_1,\text{in}}, v_{i_1,\text{mid}}\}$ and $\{v_{i_1,\text{mid}}, v_{i_1,\text{out}}\}$ are in C'. So either $v_{i_1,\text{mid}}$ is the second vertex in C' or it is the last. Since G' is undirected, though, we can flip the order of its vertices and still have a Hamiltonian cycle. So we can still assume wlog. that $v_{i_1,\text{in}}$ is the first vertex, and that it is followed by $v_{i_1,\text{mid}}$ and $v_{i_1,\text{out}}$.

Since the only elements connecting to $v_{i_1,\text{in}}$ are $v_{i_1,\text{mid}}$ or vertices of the form $v_{i_k,\text{in}}$, the vertex in C' following $v_{i_1,\text{out}}$ must be some $v_{i_2,\text{in}} \in V'$.

We can see that (v_{i_1}, v_{i_2}) is an edge in E, since $\{v_{i_1,\text{out}}, v_{i_2,\text{in}}\} \in E'$.

By our discussion above, C' must contin the subpath $(v_{i_2,\text{in}}, v_{i_2,\text{mid}}, v_{i_2,\text{out}})$.

By repeating the same argument, we see that $v_{i_2,\text{out}}$ must connect to some $v_{i_3,\text{in}}$ in C'.

As before, (v_{i_2}, v_{i_3}) must be an edge in E. Moreover, $v_{i_3,\text{in}}$ must be followed be $v_{i_3,\text{mid}}$ and $v_{i_3,\text{out}}$.

Continuing inductively we see that we can obtain from C' a sequence of vertices $C = (v_{i_1}, v_{i_2}, v_{i_3}, \ldots, v_{i_r})$ in G, where $(v_{i_k}, v_{i_{k+1}}) \in E$ for all $1 \leqslant k < r$ and where $(v_{i_k}, v_{i_1}) \in E$. That is, C is a cycle in G.

Note that the vertices $v_{i_k,in}$, $v_{i_k,mid}$, and $v_{i_k,out}$ must always occur contiguously in P', and so C is, in fact, a simple cycle.

But the only way that we can finish C is after we've exhausted all of the vertices in C', and thus all of the vertices in V' as well.

This means that we have to have included the vertices of the form $v_{i,\text{in}}$ in V' in our costruction of C, and so every vertex v_i occurs in C.

So C is a Hamiltonian cycle in G.

$\Rightarrow \langle G = (V, E) \rangle \in$HAM-CYCLE.

Finally, we can see that when constructing G', $|V'| = 3|V|\mathcal{O}(|V|)$, and $|E'| = |E| + 2|V| = \mathcal{O}(|V| + |E|)$. Since we can decide what these vertices are and what their edges are in polynomial time, the entire reduction is polytime.

So UHAM-CYCLE is $\mathcal{NP}$-complete, as required.

■

2.3.3 MAX-CUT

Definition of MAX-CUT:

Instance: A graph $G = (V, E)$, a weighting function $w : E \mapsto \mathbb{Z}_0^+$, and a number k.

Question: Is there a cut in G of weight at least k, i.e., a set $S \subseteq V$ such that

$$\sum_{\substack{u \in S, \\ v \notin S, \\ \{u, v\} \in E}} w(\{u, v\}) \geqslant k?$$

Reduced from: PARTITION.

Note: This is one of the original 21 $\mathcal{NP}$-complete problems from (KARP, 1972).

Firstly, we can see that this is in $\mathcal{NP}$:

1) It is a yes/no problem.
2) Its certificate is the cut S (we treat this as a set of vertices in V).
3) The size of this certificate is $\mathcal{O}(n)$, 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 $\langle G = (V, E), w, k, S\rangle$:"
  1. Reject if any of the following checks fails:
     $\mathcal{O}(1)$ time.
  2. Check that $S \subseteq V$.
     $\mathcal{O}(n^2)$ time.
  3. Weight = 0.
     $\mathcal{O}(1)$ time.
  4. For all edges $\{u, v\} \in E$:
     $\mathcal{O}(n^2)$ iterations.
  5.   If $u \in S$ and $v \notin S$ or $v \in S$ and $u \notin S$:
     $\mathcal{O}(n)$ time per iteration.
  6.       Weight += $w(\{u, v\})$.
     $\mathcal{O}(N)$ time per iteration, where $N$ is the size of $\langle w, k\rangle$.
  7. Check that Weight$\geqslant k$.
     $\mathcal{O}(N)$ time, where $N$ is the size of $\langle w, k\rangle$.
  8.Accept.
     $\mathcal{O}(1)$ time.
```

5) We can see that this verifier runs in $\mathcal{O}(n^3 + n^2 N)$ time, and so is polytime in the size of G.

So MAX-CUT is in $\mathcal{NP}$.

Reduction from PARTITION:

Suppose we are given a PARTITION instance $X = \big\{x_1, x_2, \ldots, x_n\big\}$.

- We let $V = \{v_i | x_i \in S\}$ (so we have one vertex per element of X).

  We let $E = \{v_i, v_j\}, 1 \leqslant i < j \leqslant n$ (so there is an edge between every pair of vertices).
- We define our weight function as: $w(\{v_i, v_j\}) = x_i x_j$, nd we let $k = \frac{1}{4} \sum_{x_i \in S} x_i^2$.

Note that under this construction, if we choose any set $S \subseteq V$,
the sum $\sum_{\substack{v_i \in S, \\ v_j \notin S, \\ \{v_i, v_j\} \in E}} w(\{v_i, v_j\})$ evaluates to

$$
\begin{aligned}
\sum_{\substack{v_i \in S, \\ v_j \notin S, \\ \{v_i, v_j\} \in E}} w(\{v_i, v_j\}) &= \sum_{v_i \in S} \sum_{\substack{v_j \notin S, \\ \{v_i, v_j\} \in E}} w(\{v_i, v_j\}) \\
&= \sum_{v_i \in S} \sum_{v_j \notin S} w(\{v_i, v_j\}) \\
&\quad \textit{(Since, for any } v_i \neq v_j, \{v_i, v_j\} \in E\textit{).} \\
&= \sum_{v_i \in S} \sum_{v_j \notin S} x_i x_j \\
&\quad \textit{(This is how we've defined } w\textit{).} \\
&= \sum_{v_i \in S} x_i \sum_{v_j \notin S} x_j \\
&= \Big[\sum_{v_i \in S} x_i \Big] \Big[\sum_{v_j \notin S} x_j \Big]
\end{aligned}
$$

Now, we can see that every set $S \subseteq V$ corresponds to a set $S' = \{x_i | v_i \in S\} \subseteq X$ (and vice-versa). So our sum becomes $\left[\sum_{x \in S'} x\right]\left[\sum_{x \in X \setminus S'} x\right]$.

If we set $s = \sum_{x \in X} x$, then clearly $\left[\sum_{x \in S'} x\right] = rs$ for some $0 \leqslant r \leqslant 1$. Similarly, $\left[\sum_{x \in S'} x\right] = (1 - r)s$.

So the sum $\left[\sum\limits_{v_i \in S} x_i\right]\left[\sum\limits_{v_j \notin S} x_j\right] = r(1-r)s^2, 0 \leqslant r \leqslant 1$. From calculus we know that this takes a maximum value exactly when $r = 1/2$, at which point the maximum value for our sum is $\frac{1}{4}s^2$. You can see that this is precisely the value we have chosen for k.

We argue that $\langle G = (V, E), w, k\rangle \in$ MAX-CUT iff $\langle S\rangle \in$ PARTITION:

> Suppose $\langle S\rangle \in$ PARTITION.
>
> So there is some $S' \subseteq X$ such that $\sum_{x_i \in S'} x_i = \frac{1}{2}\sum_{x_i \in X} x_i = (1/2)s$.
>
> We let $S = \{v_i | x_i \in S'\}$., By our above discussion we see that $\sum\limits_{\substack{v_i \in S, \\ v_j \notin S, \\ \{v_i, v_j\} \in E}} w(\{v_i, v_j\}) = \frac{1}{4}s^2 = k$.
>
> So S is a maximum cut of G.
>
> $\Rightarrow \langle G = (V, E), w, k\rangle \in$ MAX-CUT.
>
> Suppose $\langle G = (V, E), w, k\rangle \in$ MAX-CUT.
> Then it has some $S \subseteq V$ such that $\sum\limits_{\substack{v_i \in S, \\ v_j \notin S, \\ \{v_i, v_j\} \in E}} w(\{v_i, v_j\}) \geqslant k = \frac{1}{4}s^2$.
>
> If we let $S' = \{x_i | v_i \in S\}$, then, by our above discussion, we know that $\sum_{x_i \in S'} x_i = (1/2)s = (1/2)\sum_{x_i \in X} x_i$.
>
> So S' is a partitions X into two sets whose sum is the same.
>
> $\Rightarrow \langle S\rangle \in$ PARTITION.

Finally, we can see that $|V| = |S|$ and that $|E| = \mathcal{O}(|S|^2)$. Each of the weights in w and the cutoff k are polynomial in the size of the numbers $x_i \in S$, and can be calculated in polynomial time. So our enitre reduction is polytime.

So MAX-CUT is $\mathcal{NP}$-complete, as required.

■

2.3.4 INDUCED-PATH

Definition of INDUCED-PATH(Problem from (Garey and Johnson, 1979)):

> **Instance:** A graph $G = (V, E)$ and a number $k \in \mathbb{N}$.
>
> **Question:** Is there a subset $V' \subseteq V$ such that $|V'| \geqslant k$ and such that the subgraph induced by V' is a simple path on $|V'|$ vertices?

Reduced from:3SAT.

Firstly, we can see that this is in$\mathcal{NP}$:

1) It is a yes/no problem.
2) Its certificate is the set $V' \subseteq V$.
3) The size of this certificate is $\mathcal{O}(V)$, and so this certificate is polynomial in the size of the input.
4) A verifier for this certificate would be:

```
Let $\mathcal{V}$ ="On $\langle G = (V, E), k, V' \rangle$:"
  1. Reject if any of the following checks fails:
     *$\mathcal{O}(1)$ time.*
  2. Check that $V' \subseteq V$.
     *$\mathcal{O}(|V|^2)$ time.*
  3. Check that $|V'| \geqslant k$.
     *$\mathcal{O}(|V|)$ time.*
  4. Let $G'$ be the subgraph of $G$ induced by $V'$.
     *$\mathcal{O}(|V|^2)$ or $\mathcal{O}(|V|^3)$ time to construct, depending on your implementation.*
   5. Check that $G'$ is connected and has no vertex of degree more than two
(i.e., that it's a path).
     *$\mathcal{O}(|V|^2)$ time.*
  6.*Accept.*
     *$\mathcal{O}(1)$ time.*
```

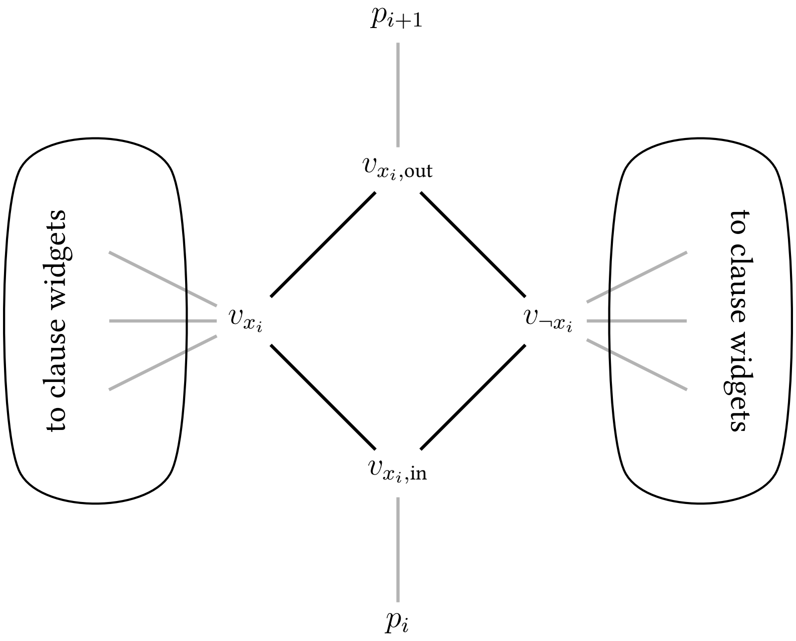
5) We can see that this verifier runs in $\mathcal{O}(|V|^3)$ time, and so is polytime in the size of G.

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

Reduction from 3SAT:
Let an input 3CNF ϕ be given, where ϕ has m variables and n clauses. Note that we can assume using our Helpful 3SAT Lemma from the previous handout that the clauses in ϕ each have three distinct variables.

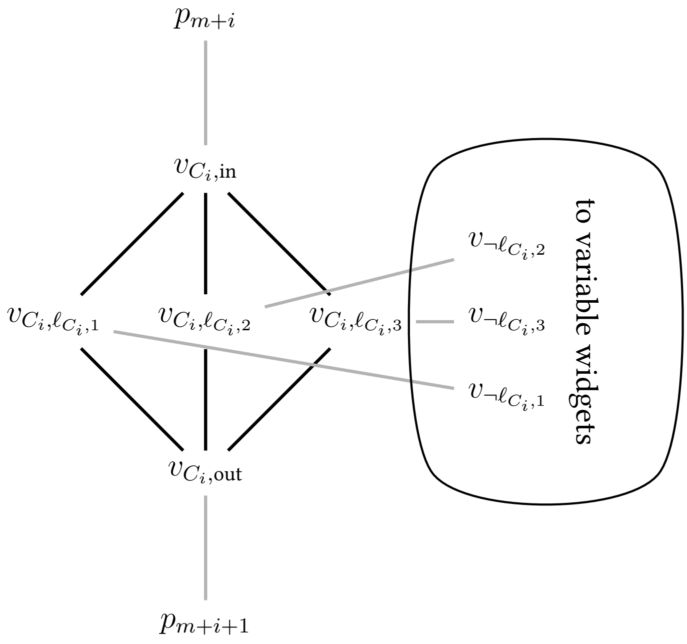
We build G as follows: we start by building $m + n + 1$ copies of a single vertex. We'll label these vertices as p_1 to p_{m+n+1}.

We'll attach p_i, $1 \leqslant i \leqslant m$, with the following widget for the m vertices in ϕ:



Note that we attach p_{i+1} to the output (the top) of each variable widget.

Once we're done with the variable widgets, we'll start on the clause widgets. As before, we'll attach p_{m+i} to the input of the clause widget for clause i, and we'll attach the output of the clause widget to p_{m+i+1}. The clause widget for clause $C_i = (\ell_1 \vee \ell_2 \vee \ell_3)$ will be as follows:



So we connect the vertex for each literal $\ell_{C_i,a}, a \in \{1,2,3\}$ in C_i with the input and output of the clause widget, but also to the *negation* of $\ell_{C_i,a}$ in the varibale widgets: so if one of the literals was x_1, we would connect to $v_{\neg x_1}$ in the first variable widget. Similarly, if one of the literals is $\neg x_2$, we would connect to the v_{x_2} in the second variable widget.

Once we've connected all of our variable and clause widgets we have our graph G. We'll set $k = 3m + 3n + (m + n + 1) = 4m + 4n + 1$.

We argue that $\langle G, k\rangle \in$ INDUCED-PATH iff $\langle \phi \rangle \in$ 3SAT:

Suppose $\langle \phi \rangle \in$ 3SAT.

Then ϕ has a satisfying truth assignment τ.

So we can pick, for any clause C_i, a literal $\ell_{C_i,a}$, $a \in \{1, 2, 3\}$, which is satisfied by τ.

Consider the following V':

- Every $p_{i'}$, $1 \leqslant i' \leqslant m + n + 1$ is contained in V'.
- For every variable x_i, the nodes $v_{x_i,\text{in}}$ and $v_{x_i,\text{out}}$ are in V'.

  If τ sets x_i to true, the vertex $v_{x_i} \in V'$, while if $x_i = F$ under τ the vertex $v_{\neg x_i} \in V'$.
- For every clause C_j, we include the vertices $v_{C_j,\text{in}}$ and $v_{C_j,\text{out}}$ in V'.

  Moreover, we add the vertex $v_{C_i,\ell_{C_i,a}}$ to V' for one literal set to true by τ.

We can see that $|V'| = 3m + 3n + (m + n + 1) = 4m + 4n + 1 = k$, and so it satisfies the size requirement for INDUCED-PATH.

Moreover, we can see that the graph G' induced by V' is connected – there is a path through every vertex and clause widget, as well as through all of the $p_{i'}$ In fact, G' has a path through all of these widgets.

We can also see that G' has no possible branching points or cycles: the $p_{i'}$ connect in G only to their neighbouring widgets, all of which contain simple paths. No two clause widgets are adjacent to each other in G, nor are any variable widgets adjacent.

The only possible adjacencies are between the clause and the vertex widgets. And we can see that we can only get an adjacency between nodes in these widgets if we choose a literal in some C_j that is not satisfied by τ (e.g., if $C_j = (x_1 \vee x_2 \vee x_3)$ and we set $x_1 = F$ and then choose $v_{C_j,\ell_{C_{j,1}}}$ in our V').

Since we have avoided these adjacencies in our construction of V', we can see that V' induces a simple path on G.

$\Rightarrow \langle G, k \rangle \in$ INDUCED-PATH.

Suppose $\langle G, k \rangle \in$ INDUCED-PATH.

Then there is some V' of size at least $k = 3m + 3n + (m + n + 1)$ that induces a path P on G.

Note that we cannot use more than three vertices in any one vertex widget without inducing a cycle, and that we cannot use more than three vertices in any clause widget without inducing either a cycle or a claw in our graph. So these widgets can add at most three vertices each to P.

So at most $3m+3n$ vertices are in these widgets, and the remaining $(m+n+1)$ vertices are in the vertices p_i. Since there are exactly $(m+n+1)$ vertices of this type, they must all be contained in P, and so the clause and variable widgets must contain three vreices each in P.

Now, we can see that p_1 and p_{m+n+1} have a degree of one even in G, and so they must be the beginning and end of P. This path is undirected, but we'll treat p_1 as its beginning.

Now, each p_i must connect to either one or two other vertices in V', and since p_1 and p_{m+n+1} are the beginning and end points in P, they must be the two p_i of degree one. All other p_i must connect to two other vertices in V'.

But the only way these vertices can connect to anything is through the $v_{x_i,\text{in}}$, $v_{x_i,\text{out}}$, $v_{C_j,\text{in}}$, and $v_{C_j,\text{out}}$ vertices in the variable and clause widgets. Moreover, the only way we can have a degree of one for p_1 and p_{m+n+1} and a degree of two for all other p_i is if all of the $v_{x_i,\text{in}}$, $v_{x_i,\text{out}}$, $v_{C_j,\text{in}}$ are included in V'.

Since each variable widget has exactly one more vertex in V', and since these vertices are either of the form v_{x_i} or $v_{\neg x_i}$, we can use these vertices to determine a truth assignment τ: τ sets $x_i = T$ if $v_{x_i} \in V'$, otherwise it sets it to F.

We also see that there must be some $v_{C_j,\ell_{C_j,a}}$ variable included in V' for every clause C_j. As we have previously noted, if the literal $\ell_{C_j,a}$ were not satisfied by τ, then there would be an edge between $v_{C_j,\ell_{C_j,a}}$ and $v_{\neg \ell_{C_j,a}}$. This edge would induce a cycle on G when paired with the rest of V'.

So $\ell_{C_j,a}$ must be satisfied by τ.

Since this is true for every clause C_j, τ must set at least one variable in each clause to true.

$\Rightarrow \langle \phi \rangle \in$ 3SAT.

Finally, we see that this graph uses $4m + 5n + (m + n + 1) = \mathcal{O}(m + n)$ vertices, and that the number of edges is also $\mathcal{O}(m + n)$. The values for these vertices and edges can be calculated in polynomial time, and so this entire reduction is polytime.

So INDUCED-PATH is $\mathcal{NP}$-complete, as required.

■

2.4 Sets, Partitions, and Orderings

2.4.1 PARTITION

Definition of PARTITION :

INPUT: A set $S = \{x_1, x_2, \ldots, x_n\}$ of non-negative integers.

QUESTION: Is there some subset $X \subseteq S$ such that $\sum_{x \in X} x = \sum_{x \in S \setminus X} x$?

Reduced from: SUBSET-SUM.

Note: This is one of the original 21 $\mathcal{NP}$-complete problems from (KARP, 1972).

More appropriately, from the fragment of SUBSET-SUM *that requires S to be a proper set (without repition), as opposed to a multiset (which allows repition). Basically, we need this because the* PARTITION definition calls for set, not a multiset. The reduction we give for SUBSET-SUM *in the parsimonious reductions tutorial also produces a proper set, and so we can see that* SUBSET-SUM *is still $\mathcal{NP}$-complete under this restriction.*
Proof that PARTITION is in $\mathcal{NP}$:

1) This is a decision problem.

2) A certificate for an instance $\langle S \rangle$ of this problem would be a subset $X \subseteq S$ such that $\sum_{x \in X} x = \sum_{x \in S \setminus X} x$.

3) Since this is a subset of S, we can see that $|X| \subseteq |S|$, and so the certificate is polysize in the size of the input.

4) Our verifier will be:

   $\mathcal{V}$ = "On input $\langle S, X \rangle$:

   1. Check that $X \subseteq S$.
   2. Let $s = \sum_{x \in X} x$.
   3. Let $s' = \sum_{x \in S} x$.
   4. Return $2s == s'$."

5) We can see that step 1. takes $\mathcal{O}(|\langle S \rangle|^2)$ time to run, where $|\langle S \rangle|$ is the space need to write out the elements of S. Steps 2. and 3. take $\mathcal{O}(|\langle S \rangle|)$ time, and that step 4. takes $\mathcal{O}(1)$ time. So the entire process takes $\mathcal{O}(|\langle S \rangle|^2)$ time.

Proof that SUBSET-SUM $\leqslant_p$ PARTITION:

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

> *Note that $s + t + 1 > s$ and that $2s - t + 1 > s$ unless $t > s$. Also note that $s \geqslant x$ for all $x \in S$. So unless we have a trivial instance of* SUBSET-SUM *we'll end up with a valid instance of* PARTITION.

Proof that $\langle S, t \rangle$ ∈SUBSET-SUM iff $\langle S' \rangle$ ∈PARTITION:

⇒) Suppose that $\langle S, t \rangle$ ∈SUBSET-SUM:

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

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

   Then,

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

On the other hand, we can also see that

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

So X' is a certificate for S'.

$\Rightarrow S' \in$ PARTITION.

$\Leftarrow$) Suppose that $\langle S' \rangle \in$ PARTITION:

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

Now, we can see that

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

In particular, we can see that

$$\sum_{x \in X'} x = \sum_{x \in S' \setminus X'} x = \Big(\sum_{x \in S'} x\Big)/2 = 2s+1.$$

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

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

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

Therefore, one of the two numbers is in X', and the other is in $S' \setminus X'$. We can say wlog. that $(2s-t+1) \in X'$.

Let $X = X' \setminus \{2s-t+1\}$.

Then,

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

and so

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

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

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

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

Note: The $+1$ in the reduction is there just to get around the corner case where $\sum_{x \in S} x = 0$.

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. So the entire process takes polynomial time.

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

■

2.4.2 X3C

Definition of X3C (Problem and reduction are from (Garey and Johnson, 1979)):

> **Instance:** A set X, where $|X| = 3q$, and a collection C of 3-element subsets of X.
>
> **Question:** Does C contain an exact cover of X, i.e., a $C' \subseteq C$ such that every element of X occurs in exactly one member of C'?

Reduced from: Reduced from:3DM.

Firstly, we can see that this is in $\mathcal{NP}$:

1) It is a yes/no problem.
2) Its certificate is the set C'.
3) Since $C' \subseteq C$, and C is part of the input, C' is polynomial in the size of the input.
4) A verifier for this certificate would be:

```
Let $\mathcal{V}$ =“On $\langle X, C, C' \rangle$:”
  1. Reject if any of the following checks fails:
     $\mathcal{O}(1)$ time.
  2. Check that $C' \subseteq C$.
     $\mathcal{O}(|C|^2)$ time.
  3. Check that every element of $C$ (and so also $C'$) has exactly three elements, all from $X$.
     $\mathcal{O}(|X||C|)$ time.
  4. For every $x \in X$:
     $\mathcal{O}(|X|)$ iterations.
  5.   Check that $x$ occurs in exactly one triple in $C'$.
     $\mathcal{O}(|X|)$ time per iteration.
  6. Accept.
     $\mathcal{O}(1)$ time.
```

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

So X3C is in $\mathcal{NP}$.

Reduction from 3DM:

Let an instance $\langle A, B, C, T \rangle$ of 3DM be given.

We set $X = A \cup B \cup C$, and let $T' = \{\{a, b, c\} | (a, b, c) \in T\}$. We return the tuple $\langle X, T' \rangle$.

Since A, B, and C are disjoint, it is trivial to see that there is an $M \subseteq T$ that covers $A \times B \times C$ iff there is a $C' \subseteq T'$ that covers triples of elements in $X = A \cup B \cup C$.

So $\langle X, T' \rangle \in$ X3C iff $inptA, B, C, T \in$ 3DM.

Finally, we can see that this reduction simply consists of copying out the elements in A, B, C, and T, and so this reduction is polytime.

So X3C is $\mathcal{NP}$-complete.

■

2.4.3 BETWEENNESS

Definition of BETWEENNESS (Problem from (Opatrny, 1979) via (Garey and Johnson, 1979)):

> **INPUT:** A set A of size n, a set T of triples $(a, b, c) \in A^3$, where a, b, and c are distinct.
>
> **QUESTION:** Is there a map $f : A \mapsto \{1, 2, \ldots, n\}$ such that for every $(a, b, c) \in T$, either $f(a) < f(b) < f(c)$ or $f(c) < f(b) < f(a)$?

Reduced from: 3SAT.

Firstly, we can see that this is in $\mathcal{NP}$:

1) It is a yes/no problem.
2) Its certificate is simply the function f (we can treat this as either a dictionary mapping A to $\{1, \ldots, n\}$, or as a litst ordering the values of A).
3) The size of this certificate is $\mathcal{O}(n)$. So this certificate is polynomial in the size of $\langle A, T \rangle$.
4) A verifier for this certificate would be:

```
Let $\mathcal{V}$ =“On $\langle A, T, f \rangle$:”
  1. Reject if any of the following checks fails:
     $\mathcal{O}(1)$ time.
  2. Check that $f$ is a one-to-one function from $A$ to $\{1, n\}$.
     $\mathcal{O}(n^2)$ time.
  3. For $(a, b, c) \in T$:
     $\mathcal{O}(n^3)$ iterations.
  4.     Check that either $f(a) < f(b) < f(c)$ or $f(c) > f(b) > f(a)$.
     $\mathcal{O}(n)$ time per iteration.
  5. Accept.
     $\mathcal{O}(1)$ time.
```

5) We can see that this verifier runs in $\mathcal{O}(n^4)$ time, and so is polytime in the size of $\langle A, T \rangle$.

So BETWEENNESS is in $\mathcal{NP}$.

The reduction 3SAT $\leqslant_p$ BETWEENNESS**:**

Let an input 3CNF ϕ be given. Note that we can assume using our Helpful 3SAT Lemma from the previous handout that the clauses in ϕ each have three distinct variables.

We will define an A and a series T of triples as follows:

- We will add two elements $\bot$ and $\top$ to A. For every element $x \in A, x \notin \{\bot, \top\}$, we will add the triple $(\bot, x\,\top)$ to T.

  > *This will make $\bot$ and $\top$ the “bottom” and “top” elements in our ordering. Note that given any valid ordering f, $f' = n + 1 - f$ is also a valid ordering, so what $\bot$ and $\top$ really are is a pair of elements that will always be on the outside of any valid ordering. Keeping this in mind, though, we’ll act like $f(\bot) = 1$ and $f(\top) = n$, since any valid ordering can be made into a valid ordering of this form.*

- We will also add the element s_+ to A.

  So, using our assumption that $f(\bot) < f(\top)$, any valid ordering will have the form $1 = f(\bot) < f(s_+) < f(\top)$.

- For every variable x_i in ϕ, we will add the variables $s_{i,T}$, and $s_{i,F}$ to A, along with the triple $(s_{i,F}, s_+, s_{i,T})$.

  So, using our assumption that $f(\bot) < f(\top)$, any valid ordering will have the form $1 = f(\bot) < f(s_{i,F}) < f(s_+) < f(s_{i,T}) < f(\top)$ or $1 = f(\bot) < f(s_{i,T}) < f(s_+) < f(s_{i,F}) < f(\top)$

  The $s_{i,T}$ element corresponds to the literal x_i, and the $s_{i,F}$ element corresponds to $\neg x_i$. So the ordering $1 = f(\bot) < f(s_{i,F}) < f(s_+) < f(s_{i,T}) < f(\top)$ corresponds to the truth assignment $x_i = T$, and the ordering $1 = f(\bot) < f(s_{i,T}) < f(s_+) < f(s_{i,F}) < f(\top)$ to $x_i = F$.

- For every clause $C_j = (\ell_1 \vee \ell_2 \vee \ell_3)$, we add three elements to A: $c_{j,1}$, $c_{j,2}$, and $c_{j,T}$. We then add the following triples to T:

  - $(a_1, c_{j,1}, a_2)$ and $(a_1, c_{j,2}, a_3)$, where a_1 is the element of A corresponding to ℓ_1, a_2 is the element of A corresponding to ℓ_2, and a_3 is the element of A corresponding to ℓ_3.
  - $(c_{j,1}, c_{j,T}, c_{j,2})$.
  - $(s_+, c_{j,T}, \top)$.

  You can see that if both ℓ_1 and ℓ_2 are false under a truth assignment τ, then their corresponding elements have an oder that is less than s_+. Thus, $c_{j,1}$, which is between these elements, satisfies $f(c_{j,1}) < f(s_+)$. If at least one of ℓ_1 or ℓ_2 is true, however, we can set $f(c_{j,1}) > f(s_+)$. The same argument applies to ℓ_1 and ℓ_3.

  So if τ satisfies C_j, it is possible to set either $f(c_{j,1}) > f(s_+)$ or $f(c_{j_2}) > f(s_+)$ (or both). Since $c_{j,T}$ is between these elements, it is possible to set $f(c_{j,T}) > f(s_+)$ iff C_j is satisfied by τ. So we can satsfy the triple $(s_+, c_{j,T}, \top)$ iff C_j is satisfied by τ.

  Similarly, if f is a valid ordering of A, then $f(s_+) < f(c_{j,T}) < f(\top)$. This means that at least one of $f(c_{j,1})$ or $f(c_{j,2})$ is greater than $f(s_+)$, and so at least one of the literals ℓ_1, ℓ_2, and ℓ_3 must correspond to an element with an order greater than $f(s_+)$. So the truth assignment corresponding to the orderings of the $s_{i,T}$ and $s_{i,F}$ elements must satisfy C_j.

After applying this construction to every variable x_i and clause C_j, we return the resulting sets $\langle A, T\rangle$.

We can see that $\langle A, T\rangle \in \text{BETWEENNESS}$ iff $\langle\phi\rangle \in \text{3SAT}$:

> If $\langle\phi\rangle \in \text{3SAT}$, then there is a truth assignment τ that satisfies ϕ. We set the $s_{i,T}$ and $s_{i,F}$ orderings as described above.
>
> By our arguments above, we know that the orderings for each $c_{j,1}$, $c_{j,2}$, and $c_{j,T}$ can be completed iff C_j is satisfied by τ. But τ satisfies every clause, so we can find an order for

each of these elements ()note that there are no inter-clause constraints between these elements, so the can't interfere with each other.

So there is a valid ordering f of A, and so $\langle A, T\rangle \in$ BETWEENNESS.

On the other hand, if $\langle A, T\rangle \in$ BETWEENNESS, then there is some valid ordering f on A.

We can find a truth assignment τ by reading the orderings of the elements $s_{i,F}$ and $s_{i,T}$ (as described above).

By our discussion, we know that every $c_{j,T}$ has an ordering greater than that of s_+, and so (also by our discussion) at least one of the literals ℓ_1, ℓ_2, or ℓ_3 is true. So τ satisfies C_j. Since this is true for every clause C_j, τ satisfies ϕ.

So $\langle \phi\rangle \in$ BETWEENNESS.

Finally, we can see that this construction will take polynomial time: if ϕ has m clauses and k variables, then $|A| = 4 + 2k + 3m$. If we say $|A| = r$, then T has:

- $r - 2$ triples of the form $(\bot, s, \top), s \in A$.
- k triples of the form $(s_{i,F}, s_+, s_{i,T})$.
- $2m$ triples of the form $(a_1, c_{j,1}, a_2)$ or $(a_1, c_{j,1}, a_3)$, where a_1, a_2, and a_3 are elements of A corresponding to literals.
- $2m$ triples of the form $(c_{j,1}, c_{j,T}, c_{j,2})$ or $(s_+, c_{j,T}, \top)$.

So $|T| = r + 4m + k - 2$.

So $\langle A, T\rangle$ is polynomial in the size of ϕ. Furthermore, we can find every element of A and T in polynomial time, and our construction is polytime.

So 3SAT $\leqslant_p$ BETWEENNESS, and so BETWEENNESS is $\mathcal{NP}$-complete.

■

2.4.4 GROUPING-BY-SWAPPING

Definition of GROUPING-BY-SWAPPING (from (Garey and Johnson, 1979), unpublished reference):

Instance: A finite alphabet Σ, a string $s \in \Sigma^*$, and a positive integer K.

Question: Is there a sequence of K or fewer adjacent symbol interchanges that converts s into a string y in which all occurances of each symbol $a \in \Sigma$ are in a single block, i.e., y has no subsequences of the form aba for $a \in \Sigma$, $b \in \Sigma^*$, and $b \neq a^i, i \in \mathbb{N}$?

Reduced from:FEEDBACK-ARC-SET.

Firstly, we can see that this is in $\mathcal{NP}$:

1) It is a yes/no problem.

2) Its certificate is the sequence $S = (\sigma_{i_1}, \sigma_{i_2}, \ldots, \sigma_{i_m})$ of swaps, where each swap σ_i is a simple transposition in $\mathfrak{S}_{|s|}$.

   > *Note: for the purposes of this discussion, $\mathfrak{S}_n$ can be thought of as the set of permutations of n distinct elements.*

3) The simple transpositions on n elements generate all of $\mathfrak{S}_n$. Moreover, we need at most $[n(n+1)]/2$ of these transpositions to generate any $\sigma \in \mathfrak{S}_n$, since we need at most n transpositions to fix the first element, and then at most $n-1$ to fix the second, and so on. So we have at most $\mathcal{O}(n^2)$ transpositions, and so the whole sequence is polynomial inthe size of $n = |s|$.

4) A verifier for this certificate would be:

```
Let $\mathcal{V}$ ="On $\langle s, K, S \rangle$:"
  1. Reject if any of the following checks fails:
     $\mathcal{O}(1)$ time.
  2. Check that $|S| \leqslant K$.
     $\mathcal{O}(|s|^2)$ time.
  3. For $i = 1$ to $|S|$:
     $\mathcal{O}(|s|^2)$ iterations.
  4.     Swap the elements $s[j]$ and $s[j+1]$, where $\sigma_i = [j, j+1]$.
     $\mathcal{O}(1)$ or $\mathcal{O}(|s|)$ time, depending on your implementation.
  5. For $i = 1$ to $|s|$:
     $\mathcal{O}(|s|)$ iterations.
  6.     Check that there are no $i', i''$ such that $i < i' < i''$, and $s[i] = s[i''] \neq s[i']$.
     $\mathcal{O}(|s|)$ time.
  7. Accept.
     $\mathcal{O}(1)$ time.
```

5) We can see that this verifier runs in $\mathcal{O}(|s|^3)$ time, and so is polytime in the size of $\langle s, K \rangle$.

So GROUPING-BY-SWAPPING is in $\mathcal{NP}$.

Reduction from FEEDBACK-ARC-SET:

Suppose we are given a FEEDBACK-ARC-SET instance $\langle G = (V, E), k \rangle$, where $V = \{v_1, v_2, \ldots, v_n\}$ and $E = \{e_1, e_2, \ldots, e_m\}$ (if not order is given to the edges or vertices we'll just choose one to work with – for example, we can order the edges by reading an adjacency matrix top to bottom and left to right).

We note, using the claim from our FEEDBACK-ARC-SET reduction that G has a size-k feedback arc set iff V has an ordering $\pi \in \mathfrak{S}_{|V|}$ that has at most k backward edges (i.e., edges (u, v) such that $\pi(u) > pi(v)$). We'll proceed using this observation.

As a bit of terminology, suppose we have a string s over an alphabet Σ. Let π be an ordering of Σ.

For example, if $\Sigma = \{a, b, c\}$ we could have $s = aabcbcca$, and we could have π be the ordering $\{a : 2, b : 1, c : 3\}$ – that is, where $\pi(b) < \pi(a) < \pi(c)$.

Let $S(s, \pi)$ be the string we get if we order s according to the order π, and let $N(s, \pi)$ be the smallest number of simple transpositions we need to derive $S(s, \pi)$ from s.

In our example, if we order $s = aabcbcca$ according to the order $\{a : 2, b : 1, c : 3\}$, then we would have to put the bs forst, then the as, then the cs, so we'd get $S(s, \pi) = bbaaaccc$. We'd also get $N(s, \pi) = 8$ – the reason why will be clear in a moment.

Claim 1: *Given Σ, s, and π, the value of $N(s, \pi) = |\{(i, j)|\ 1 \leqslant i < j < |s|$ and $\pi(s[j]) < \pi(s[i])\}|$.*

Proof:

To start, we can see that the elements (i, j) in the set $\{(i, j)|\ 1 \leqslant i < j < |s|$ and $\pi(s[j]) < \pi(s[i])\}$ correspond to distinct indices in the string s. In our above example, one such pair (i, j) would be $(2, 5)$, and this would correspond to the characters a and b in s: $a\underbrace{a}_{2}bc\underbrace{b}_{5}cca$.

We can see that according to our ordering π we'll need to move the b in index 5 to somewhere before the a in index 2, and so we'll need at least one transposition to swap the order of these two character tokens – and this is in addition to the swaps we'll need to use get the two characters next to each other, and to the other swaps we'll need as we order s.

On the other hand, if we had $(i, j) = (2, 7)$, then we'd be looking at these characters: $a\underbrace{a}_{2}bcbc\underbrace{c}_{7}a$. We might or might not need to swap these two characters.

Now, this observations means that if we have an (i, j) such that $i < j$, we can say for certain that we need to swap i and j if the character at j (i.e., $s[j]$) goes before the character $s[i]$ at index i uder the ordering π. Since $\pi(s[j])$ tells us where on in π the character $s[j]$ will go, we can write this as $i < j$ and $\pi(s[j]) < \pi(s[i])$.

So we can see that the value $|\{(i, j)|\ 1 \leqslant i < j < |s|$ and $\pi(s[j]) < \pi(s[i])\}|$ just counts the pairs of characters that we know we must transpose to order s according to π, and so it is certainly a lower bound for $N(s, \pi)$. To show that the two values are equal, then, all we have to do is give a sequence of transpositions that uses exactly $|\{(i, j)|\ 1 \leqslant i < j < |s|$ and $\pi(s[j]) < \pi(s[i])\}|$ to order s.

But the method for doing this is straightforward – we swap characters until the first character is fixed, then the second, and so on.

So in our example, to order $aabcbcca$, we'd forstly move the bs to the beginning, then fix the as, then the cs. This would give us the transposition sequence

$$
\begin{aligned}
& aabcbcca \\
(1)\quad & a\mathbf{ba}cbcca \\
(2)\quad & \mathbf{ba}acbcca \\
(3)\quad & baa\mathbf{bc}cca \\
(4)\quad & ba\mathbf{ba}ccca \\
(5)\quad & b\mathbf{ba}accca \\
(6)\quad & bbaacc\mathbf{ac} \\
(7)\quad & bbaac\mathbf{ac}c \\
(8)\quad & bbaa\mathbf{ac}cc
\end{aligned}
$$

If the first character in our final ordering starts at index j_0, then we'll need j_0 swaps to get it to the beginning of the string – and by our choice of j_0, we see that for every $i < j_0$, the character $s[i]$ must go later than $s[j_0]$ in the ordering π. So the number of swaps we'll need to move j_0 to its place in the ordered string is exactly equal to the number of (i, j_0) such that $i < j_0$ and $\pi(s[i]) > \pi(s[j_0])$.

Once this character is fixed, we siumply repeat the process for the second charcter, then the third, and so on. Continuing inductively, we see that the total number of swaps we need for any character is exactly the number of (i, j), where $i < j$, such that $\pi(s[i]) > \pi(s[j])$, as required.

□

With this in mind, we'll build a string s using the alphabet $\Sigma = V$. To start, for every edge $e_i = (v_{e_i,1}, v_{e_i,2})$ we'll let x_i be the two-character string $v_{e_i,1}v_{e_i,2}$.

Let $s_0 = x_1x_2 \ldots x_m$, and consider the string $s_1 = s_0 s_0^{\mathcal{R}}$ (recall that $s_0^{\mathcal{R}}$ is the reverse of s_0, so s_1 is an even-length palindrome).

Let's look at what happens if we try to group the characters in s_1 together through simple transpositions. Let π be any ordering of V.

By **Claim 1**, we can see that the value $N(s_1, \pi)$ will be equal to the number of $1 \leqslant i < j \leqslant 4m$ such that $\pi(s_1[i]) > \pi(s_1[j])$. But since s_1 is a palindrome, we can actually look at the characters i, j, $4m - i + 1$, and $4m - j + 1$, where $1 \leqslant i < j \leqslant 2m$ – so we find an i and j in the first half of s_1, we take their reflections in the second half, and we look at all six pairs we can get of these four indices.

Now, since s_1 is a palindrome, we can see that $s_1[i] = s_1[4m-i+1]$ and $s_1[j] = s_1[4m-j+1]$. So it will never be true that $\pi(s_1[i]) > \pi(s_1[4m - i + 1])$ or $\pi(s_1[j]) > \pi(s_1[4m - j + 1])$.

As for the other four pairs:

- If $s_1[i] = s_1[j]$, then all four characters are equal, and so $\pi(s_1[a]) = \pi(s_1[b])$ for $a, b \in \{i, j, 4m - i + 1, 4m - j + 1\}$.

- If $s_1[i] \neq s_1[j]$ and $\pi(s_1[i]) < \pi(s_1[j])$, then $\pi(s_1[i]) < \pi(s_1[4m-j+1])$, $\pi(s_1[4m-i+1]) < \pi(s_1[j])$, and $\pi(s_1[4m-i+1]) < \pi(s_1[4m-j+1])$. Since $i < j < 4m-j+1 < 4m-i+1$ by assumption, we see that there are exactly two trasnpositions needed to order these elements.
- Similarly, if $s_1[i] \neq s_1[j]$ and $\pi(s_1[i]) > \pi(s_1[j])$, then $\pi(s_1[i]) > \pi(s_1[4m-j+1])$, $\pi(s_1[4m-i+1]) > \pi(s_1[j])$, and $\pi(s_1[4m-i+1]) > \pi(s_1[4m-j+1])$. Since $i < j < 4m-j+1 < 4m-i+1$ by assumption, we see that there are also exactly two trasnpositions needed to order these elements.

Since every pair of indices $i < j$ in s_1 is in one of these quadruples of indices, and since none of these outcomes depends on the order in π itself, and since the pairs $s_1[i]$ and $s_1[]4m-i+1$ never contribute any transpositions (so we don't need to worry about double-counting), we can see The following:

> **Claim 2:** *The value $N(s, \pi)$ is the same for every permutation π.*

□

In fact, this claim will hold for any even-length palindrome.

Let $N_1 = N(s_1, \pi)$ for any permutation – say, the identity permutation. As we've said, we'll get the same result no matter which permutation we choose.

Now, let's consider what happens to $N(s_1, \pi)$ if we reverse the direction of any of the $x_{i'}^{\mathcal{R}}$ in s_1 – e.g., if we work with the string $s_0 x_m x_{m-1}^{\mathcal{R}} \dots x_1^{\mathcal{R}}$ instead of $s_0 x_m^{\mathcal{R}} x_{m-1}^{\mathcal{R}} \dots x_1^{\mathcal{R}}$.

If we try ordering this string according to a permutation π we see that, for any $i, j \neq 4m + 1 - 2i'$, the arguments from **Claim 2** still hold (i.e., when neither $4m-i+1$ nor $4m-j+1$ is in x_i). The only difference will be when on the i or j is in the string x_i – in fact, since the only pair of characters whose order has been switched has been the $x_{i'}^{\mathcal{R}}$ string, the only change in the number of swaps will be when $j = i+1$ and $s_1[i : i+2]$ is the $x_{i'}$.

When this happens, we can see that we're counting the number of transpositions needed for a sequence of characters $s_1[i], s_1[i+1], s_1[4m-i], s_1[4m-i+1] = s_1[i], s_1[i+1], s_1[i], s_1[i+1]$.

- Since $s_1[i : i+2] = x_{i'}$, and since we do not allow self-loops in simple graphs, we can see that we'll need at least one transposition.
- If $\pi(s_1[i]) < \pi(s_1[i+1])$, then we'll need one transposition to change $s_1[i], s_1[i+1], s_1[i], s_1[i+1]$ into $s_1[i], s_1[i], s_1[i+1], s_1[i+1]$.
- If $\pi(s_1[i]) > \pi(s_1[i+1])$, then we'll need three transpositions to change $s_1[i], s_1[i+1], s_1[i], s_1[i+1]$ into $s_1[i+1], s_1[i+1], s_1[i], s_1[i]$.

Note that we would initially have needed two transpositions.

As we've said, the number of transpositions for any other quadruple will not change, and so the value of $N(s_1, \pi)$ will decrease by one if $\pi(s_1[i]) < \pi(s_1[i+1])$ in π, and will increase by one if $\pi(s_1[i]) > \pi(s_1[i+1])$. But $s_1[i : i+2]$ is an edge string $x_{i'}$, and so we see that this change decreases the number of transpositions needed by one if $x_{i'}$ is a forward edge in our ordering, and increases our number of transpositions by one if it is a backward edge.

We can see that we can continue this process inductively until no $x_{i'}^{\mathcal{R}}$ remains in s_1. So if we let s_2 be the result of this process, then:

Claim 3: *The value of $N(s_2, \pi)$ is equal to $N_1 - f_\pi + b_\pi$, where f_π is the number of forward edges in G according to π, and where b_π is the number of backward edges. Note that $f_\pi + b_\pi = m$, and so we can write this value as $N_1 - m + 2b_\pi$.*

□

We cna now finish the reduction: Given $\langle G, k\rangle$, we retrun the value $\langle s_2, N_0 - m + 2k\rangle$

We argue that $\langle s_2, N_0 - m + 2k\rangle \in$ GROUPING-BY-SWAPPING iff $\langle G, k\rangle \in$ FEEDBACK-ARC-SET:

Suppose $\langle G, k\rangle \in$ FEEDBACK-ARC-SET.

Then there is a size-k feedback arc set S.

By the claim in our FEEDBACK-ARC-SET reduction, we see that this means that there is an ordering π on V in which G has at most k backward edges.

By **Claim 3**, $N(s_2, \pi) = N_0 - m + 2b_\pi < N_0 - m + 2k$.

$\Rightarrow \langle s_2, N_0 - m + 2k\rangle \in$ GROUPING-BY-SWAPPING. Suppose $\langle s_2, N_0 - m + 2k\rangle \in$ GROUPING-BY-SWAPPING.

Then there is some sequence S' of at most $N_0 - m + 2k$ simple transpositions that group s_2 into blocks of similar characters.

The order of these blocks induces one or more permutations π such that $N(s_2, \pi) < N_0 - m = 2k$.

Note that we only get more than one such π if there is some character that doesn't show up in s_2. This occurs when G has an independent vertex. On the other hand, we can see that it offers no real impediment to our argument, we'll just choose one of the available π.

By **Claim 3**, we can see that this π has at most k backward edges.

By the claim in our FEEDBACK-RC-SET reduction, this means that there is a size-k feedback arc set $S \in E$.

$\Rightarrow \langle G, k\rangle \in$ FEEDBACK-ARC-SET.

Finally, we see that s_2 is a length-$2m$ string that just lists the edges in G twice. Furthermore, note that we can calculate N_0 in $\mathcal{O}(n^2)$ time, and that the entire value $N_0 - m + 2k$ can be calculated in $\mathcal{O}(n^2 + \log k)$ time. So both elements of our instance can be calculated in polynomial time, and therefore the entire reduction is polytime.
So GROUPING-BY-SWAPPING is $\mathcal{NP}$-complete, as required.

■

2.5 Timetables and Scheduling

2.5.1 TIMETABLE-SCHEDULING

Definition of TIMETABLE-SCHEDULING *Problem and reduction from (Even et al., 1975), via (Garey and Johnson, 1979).*:

Instance: We are given the following sets:

- A finite set H of available hours $\{1, \ldots, k\}$.
- A series $\mathcal{T}$ of subsets $\{T_1, \ldots T_n\}$, where each $T_i \subseteq H$.

  In our problem, there are n teachers available. The set T_i is the set of hours in which teacher i is available to teach.
- A series $\mathcal{C}$ of subsets $\{C_1, \ldots C_m\}$, where each $C_j \subseteq H$.

  In our problem, there are m classes to be taught. The set C_j is the set of hours in which class j can be taught.
- An $n \times m$ matrix R of non-negative integers.

  In our problem, the entry r_{ij} of R is the number of hours that teacher i is required to teach class j.

Question: Is there a possible timetable $f(i, j, h)$ for the teachers and classes that satisfies the following constraints?

a) $f(i, j, h) = 1 \to h \in T_i \cap C_j$,

   The function $f(i, j, h)$ takes the values 0 and 1, and it takes the value 1 iff teacher i is assigned to class j during the hour h. This constraint ensures that we only assign teachers to classes in times that both the teacher and class are avaiable.

b) $\sum_{h=1}^{k} f(i, j, h) = r_{ij}$ for all $1 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant m$.

   This constraint ensures that teacher i is assigned all of the hours required to teach the class j.

c) $\sum_{i=1}^{n} f(i, j, h) \leqslant 1$ for all $1 \leqslant j \leqslant m$ and $1 \leqslant h \leqslant k$.

   This constraint ensures that at most one teacher is assigned to teach class j in any one time slot.

d) $\sum_{j=1}^{m} f(i, j, h) \leqslant 1$ for all $1 \leqslant i \leqslant n$ and $1 \leqslant h \leqslant k$.

   This constraint ensures that no teacher is assigned more than one class in any one time slot.

Reduced from:3SAT.

Firstly, we can see that this is in $\mathcal{NP}$:

1) It is a yes/no problem.

2) Its certificate is the timetable function f. We'll treat this as an $n \times m \times k$ table.

3) The size of f is $\mathcal{O}(i \times j \times k)$. The input includes lists of length i, j, and k, and so this certificate is polynomial in the size of the input.

4) A verifier for this certificate would be:

```
Let $\mathcal{V}$ =“On $\langle H, \mathcal{T}, \mathcal{C}, R, f \rangle$:”
  1. Check that $f$ is a function from $\mathcal{T} \times \mathcal{C} \times H$ to $\{0, 1\}$.
     $\mathcal{O}(nmk)$ time.
  2. Check that $f(i, j, h) = 1$ only if $h \in T_i \cap C_j$.
     $\mathcal{O}(nmk[n + m])$ time.
     Here we're treating the table lookups fpor $f$ (and $R$) as constant time, but searching through $T_i$ and $C_j$ as linear. We can do a bit better in the searches, and are likely a bit worse in the lookups, but this is an upper bound to within a logarithmic factor.
  3. Check that for all $i$ and $j$, $\sum_{h=1}^{k} f(i, j, h) = r_{ij}$.
     $\mathcal{O}(nmk)$ time.
     Recall that the addition of $k$ $1$s amortizes to $\mathcal{O}(1)$ time, even if general inteeger additon isn't necessarily constant time.
  4. Check that for all $j$ and $h$, $\sum_{i=1}^{n} f(i, j, h) \leqslant 1$.
     $\mathcal{O}(nmk)$ time.
  5. Check that for all $i$ and $h$, $\sum_{j=1}^{n} f(i, j, h) \leqslant 1$.
     $\mathcal{O}(nmk)$ time.
  6. If all of these checks pass, accept, otherwise reject.
     $\mathcal{O}(1)$ time.
```

5) We can see that this verifier runs in $\mathcal{O}(nmk[n + m])$ time, and so is polytime in the size of the input.

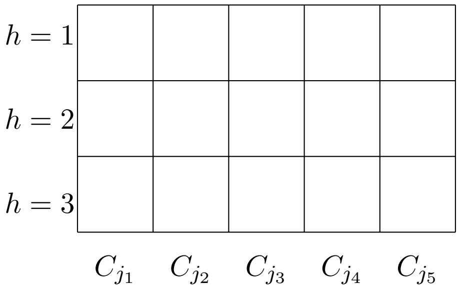
So TIMETABLE-SCHEDULING is in $\mathcal{NP}$.

Reduction from 3SAT:

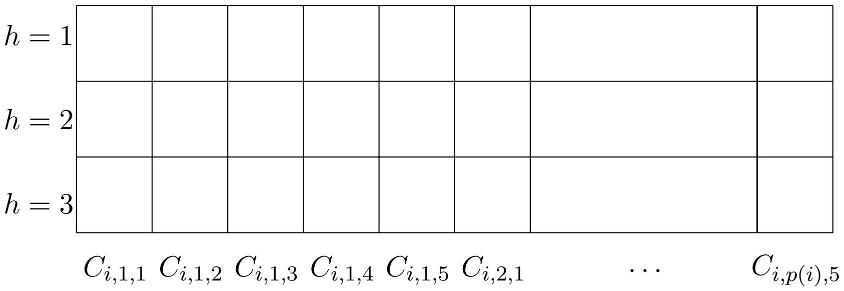
Let an instance $\langle \phi \rangle$ of 3SAT be given, where ϕ has n clauses and k variables. Note that we can assume using our Helpful 3SAT Lemma from the previous handout that the clauses in ϕ each have three distinct variables.

To start, we set $H = \{1, 2, 3\}$. We'll use the classes to represent whether a variable x_i occurs in clause D_j of ϕ, with the understanding that we'll add extra classes to help enforce consistency. We'll use the teachers to indicate our truth assignments, also with the understanding that we'll add extra teachers to help enforce consistency. Every class will be avaiable on all three hours (i.e., $C_j = H$ for all j).
Our teachers will be kept on a *tight* schedule in the sense that for all i, $\sum_{j=1}^{m} r_{ij} = |T_i|$. That is, every teacher will have to teach whenever avaiable. Also, every teacher will be required to teach either two or three hours.

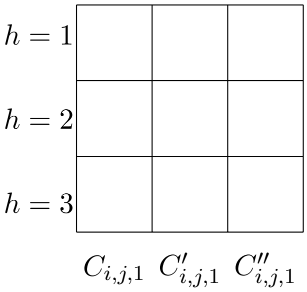
To (hopefully) help make our reduction more clear, we'll show the classes and times graphically, with the columns of a grid representing our classes, and the rows representing the times:



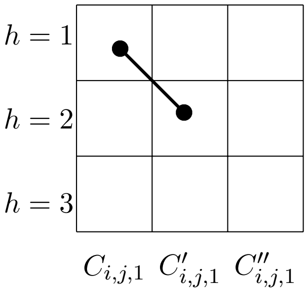
Now, for any variable x_i, let $p(i)$ be the number of clauses of ϕ in which either x_i or $\neg x_i$ occurs.
For each of these variables we will create $5p(i)$ classes, which we will label as $C_{i,1,1}, C_{i,1,2}, C_{i,1,3}, C_{i,1,4}, C_{i,1,5}, C$


$h=1$
$h=2$
$h=3$
$C_{i,1,1}$ $C_{i,1,2}$ $C_{i,1,3}$ $C_{i,1,4}$ $C_{i,1,5}$ $C_{i,2,1}$ $\cdots$ $C_{i,p(i),5}$

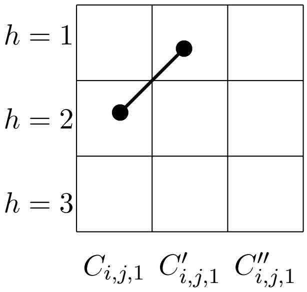
Now, we'll want to add some restrictions to which of our time slots are available. We'll do so by adding two copies of each class $C_{i,j,1}$:



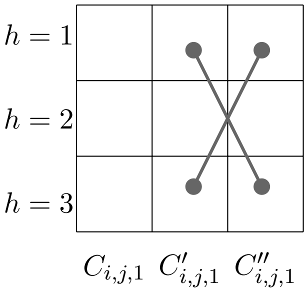
We'll add a teacher $T_{i,j,1}$ who is available in hours $h=1$ and $h=2$, and who must teach one hour each of $C_{i,j,1}$ and $C'_{i,j,1}$. This teacher has two possible schedules:



or

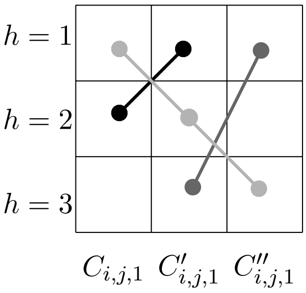


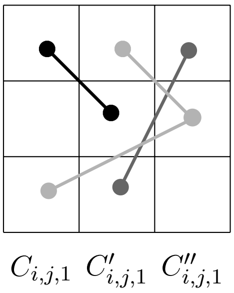
We'll add a second teacher $T'_{i,j,1}$ who is available in time slots $h = 1$ and $h = 3$, and who must teach classes $C'_{i,j,1}$ and $C''_{i,j,1}$. So we have one of the two following possible schedules for this teacher (we'll write both as lines on the grid):

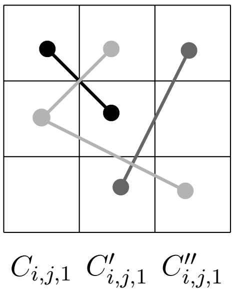


Finally, we'll add a third teacher $T''_{i,j,1}$ who must teach all three of these classes, and is avaiable in all three time slots. We can see that this gives us four possible schedules:

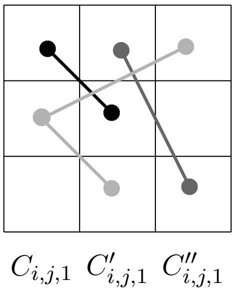
either or or





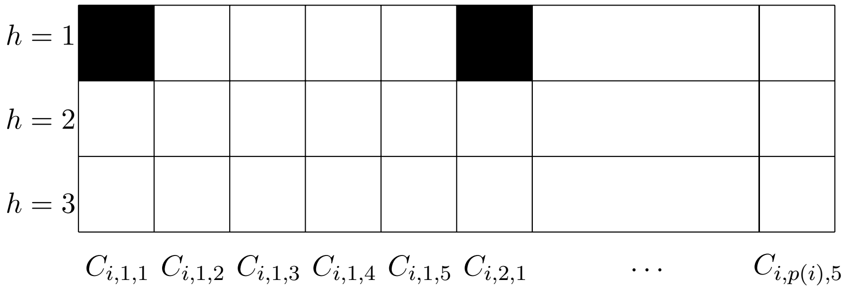


or



In any of these cases, we see that class $C_{i,j,1}$ is taken in time slot $h = 1$. One of its other two time slots is also taken, but we get to choose which one.

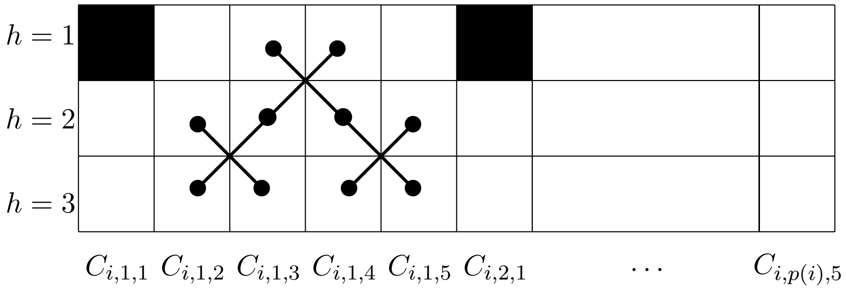
With this in mind, we'll remove the time slot $h = 1$ from our timetable for the classes $C_{i,j,1}$:



For each $1 \leqslant j \leqslant p(i)$, we'll add three teachers: $T_{i,j,3/4}$, $T_{i,j,2/3}$, and $T_{i,j,4/5}$ each of whom will teach exactly two hours:

- The first, $T_{i,j,3/4}$, will be available in time slots $h = 1$ and 2, and will teach the classes $C_{i,j,3}$ and $C_{i,j,4}$.
- The second, $T_{i,j,2/3}$, will be available in time slots $h = 2$ and 3, and will teach the classes $C_{i,j,2}$ and $C_{i,j,3}$.
- The third, $T_{i,j,4/5}$, will be available in time slots $h = 2$ and 3, and will teach the classes $C_{i,j,4}$ and $C_{i,j,5}$.

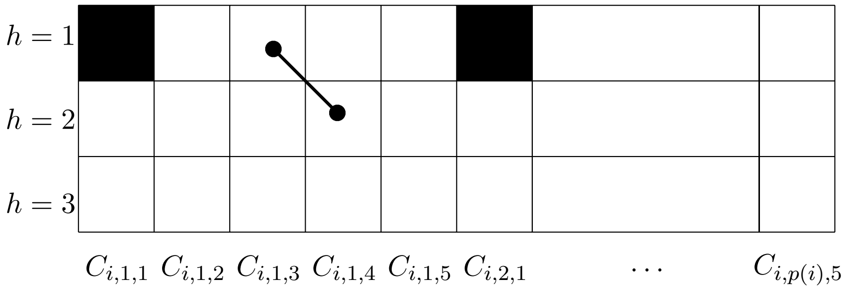
So the time slots used for $j = 1$ will be some combination of:



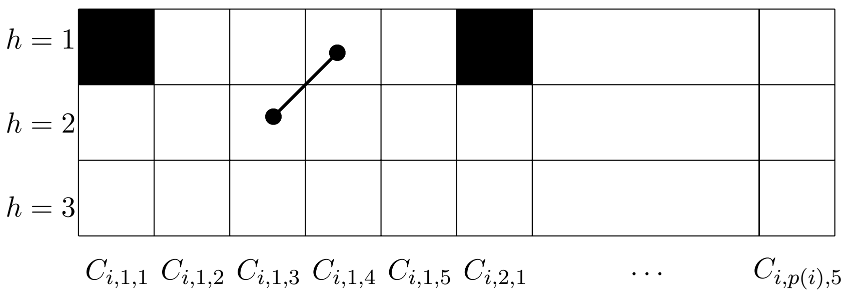
Now, we can see that there are two ways of setting the assignment of the teacher

$$T_{i,j,3/4}$$

:



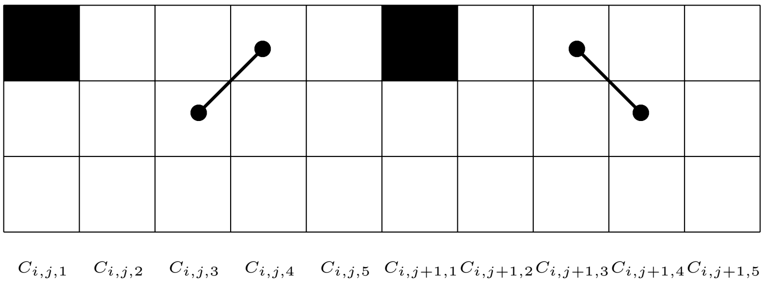
or


$h = 1$
$h = 2$
$h = 3$
$C_{i,1,1}$ $C_{i,1,2}$ $C_{i,1,3}$ $C_{i,1,4}$ $C_{i,1,5}$ $C_{i,2,1}$ $\cdots$ $C_{i,p(i),5}$

We can force this choice to be consistent across all $1 \leqslant j \leqslant i$ for this i by adding a fourth teacher $T_{i,j,c}$ for each j: This teacher will be avaiable for all three time slots, and will teach the classes $C_{i,j,4}$, $C_{i,j+1,1}$, and $C_{i,j+1,3}$.

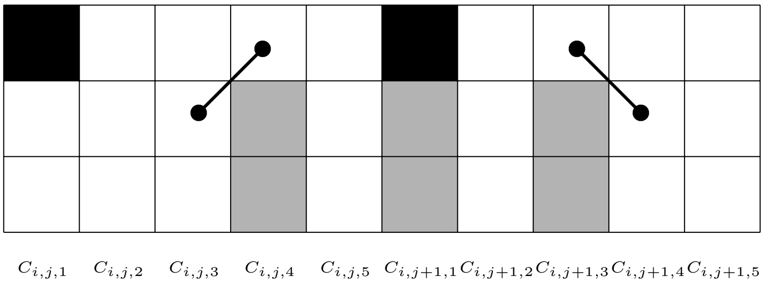
> *Note that we wrap around when we hit the end of the table: if $j = p(i)$, then the index referred to by $j + 1$ is just 1 – i.e., our additions are modulo $p(i)$.*

The reason this forces consistency is as follows: if the teachers $T_{i,j,3/4}$ are not assigned the classes $C_{i,j,3}$ consistently, then there must be some point for which their assignments are:


$C_{i,j,1}$ $C_{i,j,2}$ $C_{i,j,3}$ $C_{i,j,4}$ $C_{i,j,5}$ $C_{i,j+1,1}$ $C_{i,j+1,2}$ $C_{i,j+1,3}$ $C_{i,j+1,4}$ $C_{i,j+1,5}$

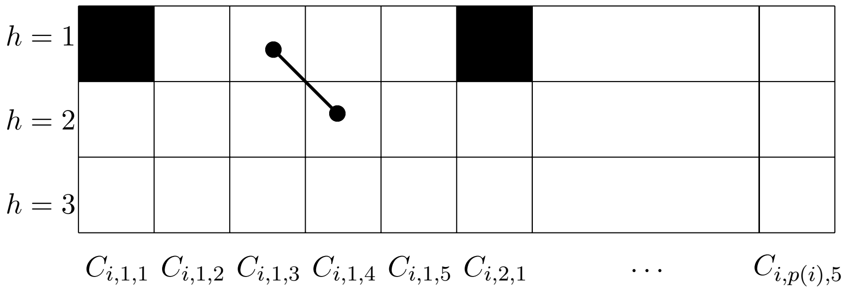
> *Note again that we wrap this table around at its end point – we may have $j = p(i)$ and $j + 1 = 1$!*

If this is the case we have to assign our new teacher $T_{i,j,c}$ to the classes $C_{i,j,4}$, $C_{i,j+1,1}$, and $C_{i,j+1,3}$. The time slots available to us for our assignment would be the gray squares in the following timetable:

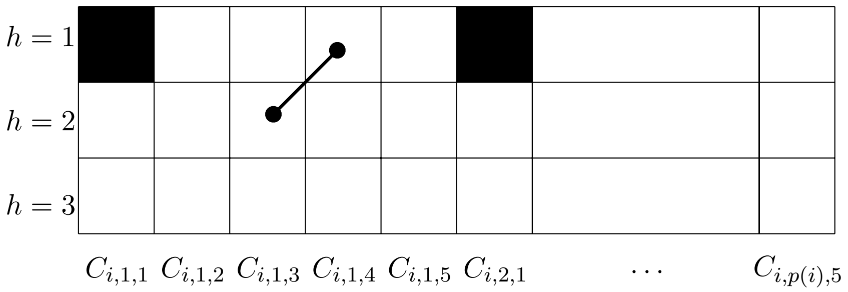

$C_{i,j,1}$ $C_{i,j,2}$ $C_{i,j,3}$ $C_{i,j,4}$ $C_{i,j,5}$ $C_{i,j+1,1}$ $C_{i,j+1,2}$ $C_{i,j+1,3}$ $C_{i,j+1,4}$ $C_{i,j+1,5}$

We would have to assign this teacher to three classes, but only have two available timeslots to do it in. We can see by the pigeonhole principle that this is not possible.

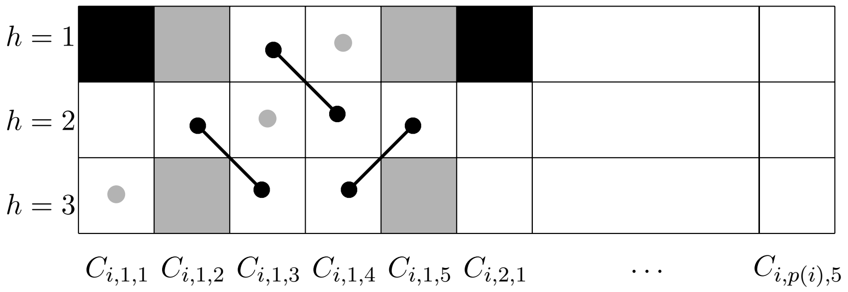
Now, once we know that our assignments for the teachers $T_{i,j,3/4}$ are consistent, we can use their scheduling assignments to encode the truth assignment for the variable x_i into our schedule. The following schedule will correspond to $x_i = T$:


$h = 1$
$h = 2$
$h = 3$
$C_{i,1,1}$ $C_{i,1,2}$ $C_{i,1,3}$ $C_{i,1,4}$ $C_{i,1,5}$ $C_{i,2,1}$ $\cdots$ $C_{i,p(i),5}$

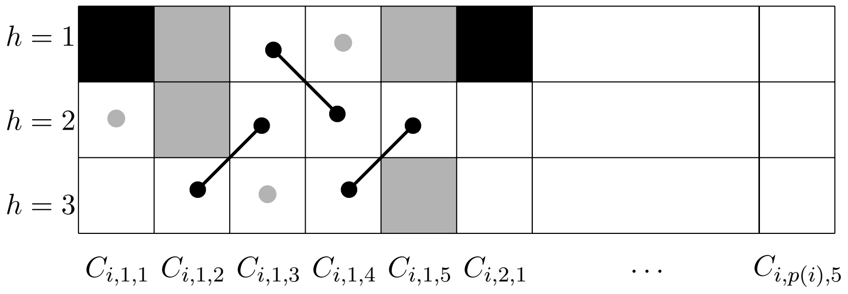
While the following schedule represents $x_i = F$:


$h = 1$
$h = 2$
$h = 3$
$C_{i,1,1}$ $C_{i,1,2}$ $C_{i,1,3}$ $C_{i,1,4}$ $C_{i,1,5}$ $C_{i,2,1}$ $\cdots$ $C_{i,p(i),5}$

In order to use these truth values to evaluate our clauses, we will make use of the classes $C_{i,j,2}$ and $C_{i,j,5}$. Note that each of these classes is taught in exactly one timeslot by the teachers currently at our disposal. If $x_i = T$, the times available for $C_{i,1,2}$ and $C_{i,1,5}$, for example would be the gray squares in the following timetables:


$h = 1$
$h = 2$
$h = 3$
$C_{i,1,1}$ $C_{i,1,2}$ $C_{i,1,3}$ $C_{i,1,4}$ $C_{i,1,5}$ $C_{i,2,1}$ $\cdots$ $C_{i,p(i),5}$

or


$h = 1$
$h = 2$
$h = 3$
$C_{i,1,1}$ $C_{i,1,2}$ $C_{i,1,3}$ $C_{i,1,4}$ $C_{i,1,5}$ $C_{i,2,1}$ $\cdots$ $C_{i,p(i),5}$

The grayed-out circles represent the timeslots taken by our consistency-enforcing teachers $T_{i,p(i),c}$ and $T_{i,1,c}$ (the first two dots are for $T_{i,p(i),c}$, the third is for $T_{i,1,c}$). Note that these teachers are only necessarily consistent in the $h = 1$ timeslot: the other two timeslots can change from clause to clause, and are determined by our choice as to whether $C_{i,1,2}$ is taught in the timeslot $h = 2$ or $h = 3$.

Note that we are forced to take the time slot $h = 2$ for the class $C_{i,1,5}$, but we have a choice for $C_{i,1,2}$. As we've said, this choice is independent across clauses.

If $x_i = F$, the situation is reversed: we can choose the timeslot taken for the $C_{i,j,5}$, but are forced to use the time slot $h = 2$ for the classes $C_{i,j,2}$.

We repeat this construction for all variables. As we've stated, each valid timetable corresponds to a truth assignment to ϕ, and every truth assignment to ϕ can generate at least one timetable under our current restrictions.

We're almost done the reduction: we simply add one more set of teachers – one for each clause $D_j = (\ell_1 \vee \ell_2 \vee \ell_3)$ of ϕ. We'll call these teachers T_j. Each T_j will teach for three hours.

Now, if $\ell_1 = x_i$ for some i, and if D_j is the q^{th} clause that makes use of x_i, we assign T_j to teach one hour of $C_{i,q,2}$. If $\ell_1 = \neg x_i$, we assign T_j to teach one hour of $C_{i,q,5}$.

We assign T_j to classes for ℓ_2 and ℓ_3 in the same fashion.

Once we are finished, the set $\mathcal{T}$ be br the set of teachers we have defined above, $\mathcal{C}$ to be the classes, R to be the teacher/class assignments, and $H = \{1, 2, 3\}$, and return $\langle H, \mathcal{T}, \mathcal{C}, R\rangle$.

We argue that $\langle H, \mathcal{T}, \mathcal{C}, R\rangle \in$ TIMETABLE-SCHEDULING iff $\langle \phi \rangle \in$ 3SAT:

> Suppose $\langle \phi \rangle \in$ 3SAT.
>
> Then it has a satisfying truth assignment τ.
>
> As we have described above, we can encode this τ into an assignemt for the teachers $T_{i,j,3/4}$. This assignment will force the class assigned to the $T_{i,j,c}$ teachers for the first hour $h = 1$, and will also determine the class assignment for one of the two teachers $T_{i,j,2/3}$ or $T_{i,j,4/5}$ (i.e., if $x_i = T$, the teachers $T_{i,j,4/5}$ will all teach the class $C_{i,j,4}$ in the timeslot $h = 3$ and the class $C_{i,j,5}$ when $h = 2$. We would be able to choose the assignment for each of the $C_{i,j,2}$. If $x_i = F$, the situation is reversed.)
>
> Since τ is a satisfying truth assignment, at least one literal in every clause is satisfied. If, say, in clause D_j, a satisfying literal is x_i, then we can assign the teacher T_j to teach the class $C_{i,q,2}$ in time $h = 2$. This teacher assignment forces the assignment of teachers $T_{i,q,2/3}$ and $T_{i,q-1,c}$, as we have seen above, but this assignment is possible. The teaching assignment for $T_{i,q-1,c}$ in class $T_{i,q,1}$ can be extended to a choice of the teachers $T_{i,q,1}$, $T'_{i,q,1}$, and $T''_{i,q,1}$ in the classes $C_{i,q,1}$, $C'_{i,q,1}$, and $C''_{i,q,1}$, as we have seen above.
>
> If the satisfing literal is $\neg x_i$, then T_j can be assigned to the class $C_{i,q,5}$ when $h = 2$. The class assignments for $T_{i,q,4/5}$ and $T_{i,q,c}$ are then forced, and this assignment can be extended to $T_{i,q+1,1}$, $T'_{i,q+1,1}$, and $T''_{i,q+1,1}$ in the same fashion as in the previous case.
>
> The remaining classes taught by T_j can be arranged as needed in time slots $h = 1$ and $h = 3$. As we have noted, the class assignments for different clauses in the same variable section of the timetable must be consistent as to their $T_{i,j,3/4}$ assignment, but can otherwise vary from clause to clause. The fact that τ is also consistent in its truth assignment from clause to clause ensures the consistency of the $T_{i,j,3/4}$ assignments. The remaining assignments can be extended to the $T_{i,q,2/3}$, $T_{i,q,4/5}$, $T_{i,q,c}$, $T_{i,q+1,1}$, $T'_{i,q+1,1}$, and $T''_{i,q+1,1}$ teachers, again following the argument above.
>
> This teaching assignment will cover all of the required teachers and classes:
>
>> We have ensured that ever teaching assignment is in a time when both the teacher and class are avaiable, and so property a) is satisfied.

> We have arranged that every teacher uses every available time slot (and so teaches all required classes). So property b) is satisfied.
>
> We have ensured that no two teachers are assigned the same class at the same time, so propert c) is satisfied.
>
> We have only assigned one class to any one teacher at a time, and so property d) is satisfied.

So gives us a valid timetable f.

$\Rightarrow \langle H, \mathcal{T}, \mathcal{C}, R\rangle \in$ TIMETABLE-SCHEDULING.

Suppose $\langle H, \mathcal{T}, \mathcal{C}, R\rangle \in$ TIMETABLE-SCHEDULING.

Then there is a valid timetable f satisfying properties a), b), c), and d).

As we have argued above, the teachers $T_{i,j,3/4}$ must be assigned consistently for any fixed i, and so we can use this assignment to read a truth assignment τ: if the $T_{i,j,3/4}$ is assigned to the classes $C_{i,j,3}$ when $h = 1$, then $x_i = T$. Otherwise it is false.

Consider any clause D_j in ϕ. Since f is a valid timetable, the teacher T_j must teach some class in the timeslot $h = 2$.

If this class is $C_{i,q,2}$ for some i, then we know that x_i is a literal in D_j. Moreover, if T_j teaches $C_{i,q,2}$ when $h = 2$, then $T_{i,q,2/3}$ teaches $C_{i,q,2}$ when $h = 3$. We then know that $T_{i,q,2/3}$ teaches $C_{i,q,3}$ when $h = 2$, and so $T_{i,q,3/4}$ teaches $C_{i,q,3}$ when $h = 1$. So the variable x_i is set to true, and therefore satisfies D_j.

Similarly, if T_j teaches $C_{i,q,5}$ in timeslot $h = 2$ for some i, then $\neg x_i$ is a literal in D_j and x_i is set to false. Either way, D_j is satisfied.

Since this is true for every clause D_j, τ satisfies every clause in ϕ.

$\Rightarrow \langle \phi\rangle \in$ 3SAT.

Finally, we can see that if ϕ has n clauses and k variables we have created $21n$ classes and $22n$ teachers. There are only three time slots avaiable, and we can determine the values of $\mathcal{T}$, $\mathcal{C}$, and R in polynomial time. So this entire reduction is polytime.

So TIMETABLE-SCHEDULING is $\mathcal{NP}$-complete, as required.

■

2.5.2 OPEN-SHOP-SCHEDULING

Definition of OPEN-SHOP-SCHEDULING (from (Garey and Johnson, 1979)):

> **Instance:** A number $m \in \mathbb{Z}^+$ of processors, a set J of jobs, each job $j \in J$ consisting of m tasks $t_1[j], t_2[j], \dots, t_m[j]$ (with $t_i[j]$ to be executed by processor i), a length $L = \{\ell(t_i[j]) | 1 \leqslant i \leqslant m, j \in J\}$ for each such task, and an overall deadline $D \in \mathbb{Z}^+$.
>
> **Question:** Is there an open-shop scedule for J that meets the overall deadline, i.e., a collection of one-processor schedules $\sigma_i : J \mapsto \mathbb{Z}_0^+, 1 \leqslant i \leqslant m$, such that $\sigma_i(j) > \sigma_i(k)$ implies $\sigma_i(j) \geqslant \sigma_i(k) + \ell(t_i[k])$, such that for each $j \in J$ the intervals $[\sigma_i(j), \sigma_i(j) + \ell(t_i[j]))$ are all disjoint, and such that $\sigma_i(j) + \ell(t_i[j]) \leqslant D$ for all $1 \leqslant m$, $1 \leqslant j \leqslant |J|$?

Reduced from:PARTITION.

Firstly, we can see that this is in $\mathcal{NP}$:

1) It is a yes/no problem.
2) Its certificate is the set $S = \{\sigma_i | 1 \leqslant i \leqslant m\}$ of schedules.
3) Since we list out every job j in J as part of the input, and since each job lists a time $\ell_{t_i[j]}$ for each $1 \leqslant i \leqslant m$, we can see that there are $|J| \times m$ times listed in the input. Since there are also $|J| \times m$ times listed in the certificate, this part is polynomial – we only have to worry about the sizes of the times given. But since we have a deadline D, which is written in $\mathcal{O}(\log D)$ bits, as part of the input, and since, for a schedule to meet this deadline we must have $s_i(j) < D$ for every i and j, we can see that the whole certificate must have a size of $\mathcal{O}(|J| m \log D)$, which is polynomial in the size of the input.
4) A verifier for this certificate would be:

```
Let $\mathcal{V}$ ="On $\langle mJ, L, D, S \rangle$:"
  1. Reject if any of the following checks fails:
     *$\mathcal{O}(1)$ time.*
  2. For $j = 1$ to $|J|$:
     *$\mathcal{O}(|J|)$ iterations.*
  3.   For $i = 1$ to $m$:
     *$\mathcal{O}(m)$ iterations.*
  4.     For $i' = 1$ to $m$:
         *$\mathcal{O}(m)$ iterations.*
  5.       Check that $\sigma_i(j) + \ell(t_i[j])$ and $\sigma_{i'}(j) + \ell(t_{i'}[j])$ are at most $D$
           *$\mathcal{O}(\log D)$ time.*
     *This'll actually check each job multiple times,*
     *so it's not as efficent as we'd like it to be.*
     *But it's still polytime, and it makes the code easier.*
  6.       Check that $[\sigma_i(j), \sigma_i(j) + \ell(t_i[j]))$ and $[\sigma_{i'}(j) + \ell(t_{i'}[j]))$ are disjoint.
           *$\mathcal{O}(\log D)$ time.*
  7. For $i = 1$ to $m$:
     *$\mathcal{O}(m)$ iterations.*
  8.   For $j = 1$ to $|J|$:
       *$\mathcal{O}(|J|)$ iterations.*
  9.     For $j' = 1$ to $|J|$:
         *$\mathcal{O}(|J|)$ iterations.*
  10.      If $\sigma_i(j) < \sigma_i(j')$:
           *$\mathcal{O}(\log D)$ time.*
  11.        Check that $\sigma_i(j') \geqslant \sigma_i(j) + \ell(t_i(j))$.
             *$\mathcal{O}(\log D)$ time.*
  12. *Accept.*
     *$\mathcal{O}(1)$ time.*
```

5) We can see that this verifier runs in $\mathcal{O}(m^2 |J|^2 \log D)$ time, and so is polytime in the size of $\langle mJ, L, D \rangle$.

So OPEN-SHOP-SCHEDULING is in $\mathcal{NP}$.

Reduction from PARTITION:

Suppose we are given a PARTITION instance $\langle S = \{a_1, a_2, \ldots, a_n\}\rangle$. If S contains only zeroes, it is trivially a yes-instance, and we can output a yes-instance of OPEN-SHOP-SCHEDULING. So we'll assume that S does not contain only zeroes. If S contains any zero, this zero can be placed into any set without changing its sum, so they do not change whether S is a yes or a no-instance of PARTITION. So we can assume wlog. that S contains no zero element.

> *This assumption is actually not necessary, but it removes the headache of dealing with length-zero jobs.*

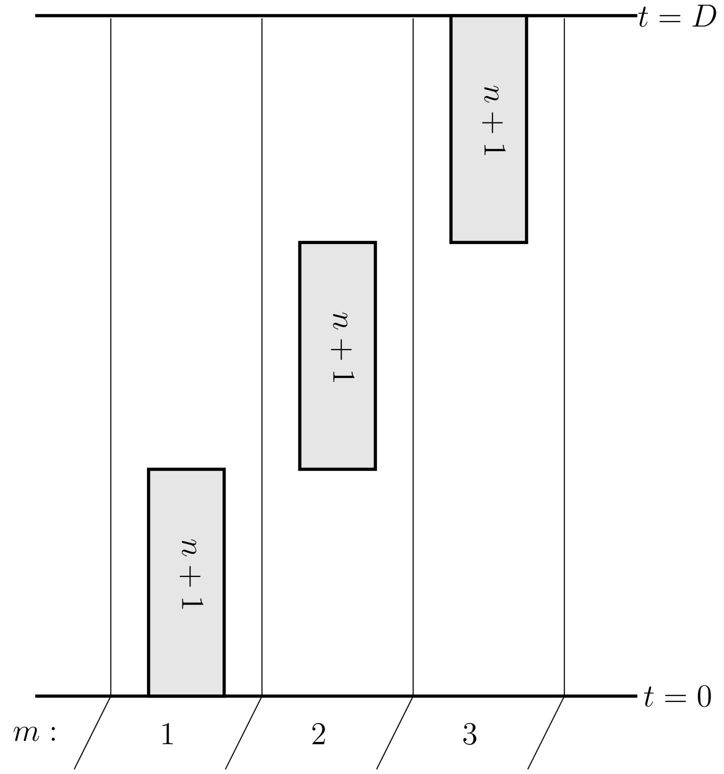
We'll set $m = 3$, and for every $a_j \in S, 1 \leqslant jleqslantn$, we build a job j in J. For each of these jobs, and for $i \in \{1, 2, 3\}$, $\ell(t_i[j] = 2a_j\}$.

We'll also add a job $n + 1$ to J. For $i \in \{1, 2, 3\}$, $\ell(t_i[n + 1])$ will be $\left(\sum_{a \in S} a\right)$. We'll put all of these lengths together into a list L.

We'll set he overall deadline D to be $3\left(\sum_{a \in S} a\right)$, and we'll return the instance $\langle m, J, L, D\rangle$.

Claim:

Before we get to the iff proof, we'll note the following property of our OPEN-SHOP-SCHEDULING instance: The job $(j + 1)$ must be run on all three processors, and the sum of the times for each of its three subprocesses is exactly D. So one of the three processors must always be running a subprocess of job $j + 1$ if we are to meet the deadline:



In fact, if we have a valid schedule, we can assume wlog. that it looks exactly like this – one of the three processors must immediately start work on job $n + 1$ at time $t = 0$, and by swapping the names of the processors we can argue that this machine is processor 1.

Similarly, we can argue that processor 2 is the second to work on $n + 1$, and processor 3 is the third.

We note that processors 1 and 3 have one large time slot remaining for all of the jobs 1 to n, while each of these jobs must be split into two equally-sized time slots in the schedule for processor 2.

We can see that $\langle m, J, L, D\rangle \in$ OPEN-SHOP-SCHEDULING iff $\langle S\rangle \in$ PARTITION:

Suppose $\langle S\rangle \in$ PARTITION.

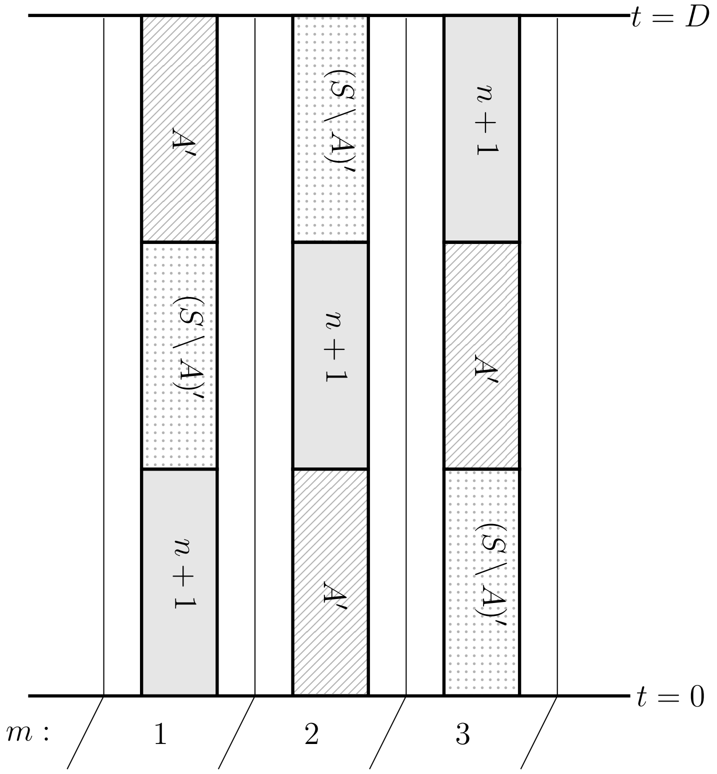
Then there is some set $A \subseteq S$ such that $\sum)a \in Aa = \sum_{a\in S\setminus A} a$.

Let A' be the set $\{i|a_i \in A\}$, taken as a set of jobs. Similarly, let $(S \setminus A)' = \{i|a_i \in A'\}$, also taken as a set of jobs.

Now, if we sort the jobs in A' and place them all together, we can run them in a block of time $\sum_{a\in A} 2a = (1/2)\sum_{a\in S} 2a = \sum_{a\in S} a$. We note that this block of jobs has the same total running time as the job $n+1$. We'll run this block at time $t = 0$ on processor 2, at time $t = \sum_{a\in S} a$ on processor 3, and at time $t = 2\sum_{a\in S} a$ on processor 1.

Now, if we sort the jobs in $(S \setminus A)'$ and place them all together, we can run them in a block of time $\sum_{a\in S\setminus A} 2a = (1/2)\sum_{a\in S} 2a = \sum_{a\in S} a$. This block of jobs also has the same total running time as the job $n+1$. We'll run this block at time $t = 0$ on processor 3, at time $t = \sum_{a\in S} a$ on processor 1, and at time $t = 2\sum_{a\in S} a$ on processor 2.

If we do this, our overall schedule will look something like this:



We note that these processor schedules meet the deadline D.

Moreover, we can see that by running each job in A' and $(S \setminus A)'$ is order, the schedule for each processor will not have any jobs that are run at the same time. So this schedule will satisfy the property that for each $j \neq j' \in J$ and each $i \in 1, 2, 3$, $\sigma_i(j) \leqslant \sigma_i(j') \to \sigma_i(j) + \ell(t_i[j]) \leqslant \sigma_i(j')$.

In addition, we have divided our jobs into three disjoint bloks of jobs, each of which can be finished in the same amount of time: the set A', the set $(S \setminus A)'$, and the singleton set $\{n+1\}$. Each of these blocks is run in a different time for each processor, and so the intervals $[\sigma_i(j), \sigma_i(j) + \ell(t_i[j]))$ are all disjoint.

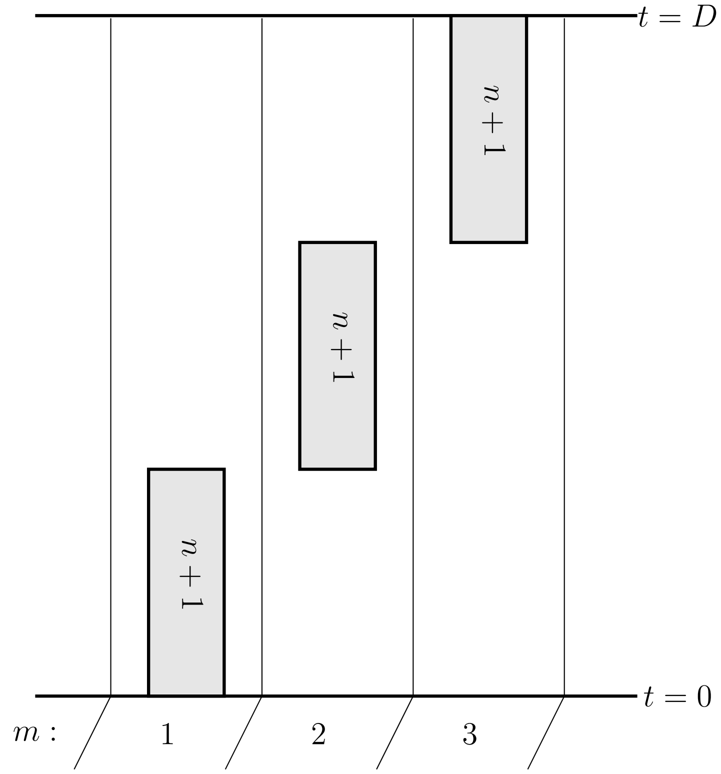
This means that our set of schedules is valid.

$\Rightarrow \langle m, J, L, D\rangle \in$ OPEN-SHOP-SCHEDULING.

Suppose $\langle m, J, L, D\rangle \in$ OPEN-SHOP-SCHEDULING.

Then there is some valid schedule for the jobs in J that meets the deadline D.

By our claim above, we can assume wlog. that the schedules for the job $n+1$ look like:



Specifically, we can assume that processor 2 has a schedule in which job $n+1$ takes the time slot from $t = \sum_{a \in S} a$ to $2\sum_{a \in S} a$.

This means that there are two free blocks of time available for our jobs: once from $t = 0$ to $t = \sum_{a \in S} a$, and another from $t = 2\sum_{a \in S} a$ to $t = 2\sum_{a \in S} a$ to $t = 3\sum_{a \in S} a = D$.

Let A' be the set of jobs that are run by processor 2 anytime in the time interval $[0, \sum_{a \in S} a)$ (note that we do not include the endpoint, so job $n+1$ is not in this set).

Since job $n+1$ is not in A', A' contains some subset of the jobs 1 to n. So we can build a set $A \subseteq S$, where $A = \{a_j | j \in A'\}$. Note that $\sum_{j \in A'} \ell(t_i[j]) = 2\sum_{a \in A} a$ by our definition of $\ell(t_i[j])$.

Since we cannot run any job before $t = 0$, and since we cannot run any job in A' after $t = \sum_{a \in S} a$ without conflicting with the job $n+1$, we can see that $\sum_{j \in A'} \ell(t_i[j]) \leqslant \sum_{a \in S} a$ (i.e., $\sum_{a \in A} a \leqslant (1/2)\sum_{a \in S} a$).

Similarly, if we let B' be the set of jobs $J \setminus (A' \cup \{n+1\})$, then the set B' must contain exactly the jobs that are run in the time interval $[2\sum_{a\in S} a, D)$. Note that the set $S \setminus A$ is exactly the set $\{a_i | i \in B'\}$. So $\sum_{j\in B'} \ell(t_i[j]) = 2\sum_{a\in S\setminus A} a$ by our definition of $\ell(t_i[j])$.

Since we cannot run any of our jobs after D, and since we cannot run any job in B' before $t = 2\sum_{a\in S} a$ without conflicting with the job $n+1$, we can see that $\sum_{j\in B'} \ell(t_i[j]) \leqslant \sum_{a\in S} a$ (i.e., $\sum_{a\in S\setminus A} a \leqslant (1/2)\sum_{a\in S} a$).

But since $\sum_{a\in A} a + \sum_{a\in S\setminus A} a = \sum_{a\in S} a$, and since $\sum_{a\in A} a \leqslant (1/2)\sum_{a\in S} a$ and $\sum_{a\in S\setminus A} a \leqslant (1/2)\sum_{a\in S} a$, we can see that $\sum_{a\in A} a = \sum_{a\in S\setminus A} a = (1/2)\sum_{a\in S} a$.

So A is a valid partiton of S.

$\Rightarrow \langle S\rangle \in$ PARTITION.

Finally, we can see that this reduction simply makes three copies of S to form the job list, sums up the times in S, and uses a linear multiple of these sums to build job $n+1$ and the deadline D. All of these operations can be performed in polynomial time, and so this entire reduction is polytime.

So OPEN-SHOP-SCHEDULING is $\mathcal{NP}$-complete.

■

2.6 Puzzles and Games

2.6.1 CROSSWORD

Definition of CROSSWORD (Problem is from (Garey and Johnson, 1979) under the name CROSSWORD-PUZZLE-CONSTRUCTION, the reduction is entirely my own fault):

> **Instance:** An alphabet Σ, a finite set $W \subseteq \Sigma^*$ and an $n \times n$ matrix A of elements in $\{0, 1\}$.
>
> **Question:** Can an $n \times n$ crossword puzzle C be built in such a way that the blank squares of C occur exactly at the 1-entries of A, in such a way that every non-blank character is an element of Σ, and such that every maximal contiguous horizontal or vertical sequence of non-blank characters in C, taken in order, gives a word in W?
>
> *Note that we can safely ignore any characters in Σ not used in W, and we can safely ignore any words in W of length greater than n. So we'll assume that every character in Σ is used in W, and that W has only words of length at most n.*

Reduced from: UHAM-CYCLE.

Firstly, we can see that this is in $\mathcal{NP}$:

1) It is a yes/no problem.
2) Its certificate is the crossword C.
3) The size of this certificate is $\mathcal{O}(n^2 \times |\Sigma|)$ Since every character in Σ occurs in W, which is part of the input, and since A is an $n \times n$ matrix, we can see that C is polynomial in the size of the input.

4) A verifier for this certificate would be:

```
Let $\mathcal{V}$ =“On $\langle W, A, C\rangle$:”
  1. Reject if any of the following checks fails:
     $\mathcal{O}(1)$ time.
  2. Check that $C$ is an $n \times n$ matrix with characters in $\Sigma \cup \{\sqcup\}$).
     $\mathcal{O}(n^2 \times |\Sigma|)$ time.
  3. For all rows $C_{i,\cdot}$of $C$:
     $\mathcal{O}(n)$ iterations.
  4. For all maximal words $w$ in the $i^{th}$ row $C_{i,\cdot}$ of $C$:
     $\mathcal{O}(n)$ iterations.
  5.     Check that $w \in W$.
     $\mathcal{O}(n|W|)$ time per iteration.
  6. For all columns $C_{\cdot,j}$of $C$:
     $\mathcal{O}(n)$ iterations.
  4. For all maximal words $w$ in the $j^{th}$ column $C_{\cdot,j}$ of $C$:
     $\mathcal{O}(n)$ iterations.
  5.     Check that $w \in W$.
     $\mathcal{O}(n|W|)$ time per iteration.
  6.Accept.
     $\mathcal{O}(1)$ time.
```

5) We can see that this verifier runs in $\mathcal{O}(n^3|W|)$ time, and so is polytime in the size of G.

So CROSSWORD is in $\mathcal{NP}$.

Reduction from UHAM-CYCLE:

Suppose we are given a graph $G = (V, E)$. We will assume there is an ordering $v_1, \dots, v_k, k = |V|$ of the vertices in G (this can just be the order of the vertices in an adjacency list or matrix). If $k = 2$ there cannot be a Hamiltonian cycle, and so we'll just output a no-insntance of CROSSWORD, say, $W = \varnothing$ and $A = 0^{2\times 2}$. Otherwise,

We'll set $\Sigma = \{a, b, c, d, e, f, g, h, 0, 1, 2\}$, and we'll set $A = 0^{(k+2)\times(k+2)}$ (that is, A is just the $(k+2) \times (k+2)$ zero matrix). This means that every word in our crossword will have to be of length exactly n.

We'll start by adding four words to our dictionary W: “$a\overbrace{ee\dots e}^{k \text{ times}}b$” “$a\overbrace{ff\dots f}^{k \text{ times}}c$”, “$b\overbrace{gg\dots g}^{k \text{ times}}d$”, and “$c\overbrace{hh\dots h}^{k \text{ times}}d$”.

Every other word in W will either begin in e and end in h, or it will begin in f and end with g. The internal characters of each of these words will be elements of $\{0, 1, 2\}$.

- Note that this means that the top-left character in any valid C must be a - that the top and the left words must be “$a\overbrace{ee\dots e}^{k \text{ times}}b$” and “$a\overbrace{ff\dots f}^{k \text{ times}}c$”, in some order:
  - If either the top or the left word were “$b\overbrace{gg\dots g}^{k \text{ times}}d$” or “$c\overbrace{hh\dots h}^{k \text{ times}}d$”, then the righmost or the bottommost word would have to begin in d. But no word will be allowed to begin with the letter d.

- If any of the other words in W were to begin C, they would have to be of the form "esh" or "fsg", where s is some string in $\{0, 1, 2\}^k$. But no word in W will start with 0, 1, 2, g, or h. So we would not be able to complete our puzzle.

- In fact, we can see that either the top word is "$a\overbrace{ee\ldots e}^{k\text{ times}}b$" and the leftmost word is "$a\overbrace{ff\ldots f}^{k\text{ times}}c$", or vice-versa.

  - If, say, both the top word and the leftmost word were "$a\overbrace{ee\ldots e}^{k\text{ times}}b$", then the rightmost word and the bottom word would have to be the one word starting in b: "$b\overbrace{gg\ldots g}^{k\text{ times}}d$". But then every other word in the puzzle would have to start with e and end with g. Since we will write no words of this form, we would be unable to complete the puzzle.

  The "$c\overbrace{hh\ldots h}^{k\text{ times}}d$" case is similar.

So the outer boundary of any valid C must be of the form

a	e	e	e	$\cdots$					$\cdots$	e	b
f	$\ddots$			$\cdots$							g
f											g
$\vdots$											$\vdots$
$\vdots$											$\vdots$
f	$\cdots$									$\cdots$	g
c	h	h	h	$\cdots$					$\cdots$	h	d

(or the transpose of this).

We'll work with C of the form above (that is, with "$a\overbrace{ee\ldots e}^{k\text{ times}}b$" as the topmost word), and keep in mind that if there is any valid C, we can turn it into one of this form by using a transpose operation.

This means that we can split the remaining words in W into two categories: the horizontal words that we can put in the rows of C, and the vertical words that we can put into the columns. The horizontal words will be the ones of the form "fsg", and the vertical ones will be of the form "esh".

The horizontal words in W will correspond to the edges in E: for any edge $\{u, v\} \in E$:

- If $\{u, v\} = \{v_1, v_i\}$, $i > 1$, we'll add the string "fsg" to W, where s is 0 on all but its first and i^{th} characters. The first character will be 2, and the i^{th} character will be 1. So s is of the form $20^{i-2}10^{k-i}$.

- If $\{u, v\} = \{v_i, v_j\}$, $j > i > 1$, we'll add the string "fsg" to W, where s is 0 on all but its first and i^{th} and j^{th} characters, both of which will be 1. So s is of the form $0^{i-1}10^{j-i-1}10^{k-j}$.

The vertical words in W will Give us constriants on how often we can use nodes in C. These words will be of the form "$e20^{k-2}2h$" or of the form $e0^i110^{k-i-2}h, 0 \leqslant i \leqslant k$.

After adding these words to W, we output $\langle W, A\rangle$.

We can see that $\langle W, A\rangle \in$ CROSSWORD iff $\langle G\rangle \in$ UHAM-CYCLE:

> Suppose that $\langle G\rangle \in$ UHAM-CYCLE.
>
> Then there is some Hamiltonian cycle through G.
>
> Since this cycle goes through every vertex, we can treat it aas a list $v_1 = v_{i_i}, v_{i_2}, \ldots, v_{i_k}$, where $\forall 1 \leqslant r \leqslant k-1, \{v_{i_r}, v_{i_{r+1}}\} \in E$ and $\{v_{i_1}, v_{i_k}\} \in E$.
>
> We build a C as follows:
>
>> Firstly, we set the outer boundary as we have described above, with the topmost word being "ae^kb", the leftmost word being "af^kc", the rightmost word being "bg^kd", and the bottom word being ch^kd.
>>
>> Secondly, the second horizontal word (the first beneath the top word) is $f20^{i_2-2}10^{k-i_2}f$. This row corresponds to the edge $\{v_1, v_{i_2}\}$.
>>
>> The second-to-bottom row is similar: it is the string $f20^{i_k-2}10^{k-i_k}f$, and it corresponds to the edge $\{v_1, v_{i_k}\}$.
>>
>> The rows from row 3 to row k follow the edges in out cycle: the string for row $r+1, 1 < r < k$ is either of the form $f0^{i_r-1}10^{i_{r+1}-i_r-1}10^{k-i_{r+1}}g$ or $f0^{i_{r+1}-1}10^{i_r-i_{r+1}-1}10^{k-i_r}g$, depending on whether $i_r < i_{r+1}$ or $i_r > i_{r+1}$.
>
> By doing this, we have set up C so that all of its rows, as well as its first and last column, are elements of W. But the second to $(k+1)^{st}$ column are also in W:
>
>> The second column of C is clearly $e20^{k-2}2h$, which is an element of C.
>>
>> Every vertex v_i in G is seen exactly once in our Hamiltonian cycle, and so shows up in exactly two edges – once when we go into v_i, and once when we leave it. So, if the vertex $v_i = v_{i_r}, 1 < r \leqslant k$, we can see that the $(i+1)^{st}$ column of C is just $e0^{r-2}110^{k-r}h$, which is also an element of C.
>
> So C satisfies our definition of a crossword.
>
> $\Rightarrow \langle W, A\rangle \in$ CROSSWORD.

Suppose that $\langle W, A\rangle \in$ CROSSWORD.

Then it has a valid crossword C.

From our discussion above, we know that both C and its transpose are valid crosswords, and that either C or its transpose have a topmost word of "ae^kb", a leftmost word of "af^kc", a rightmost word of "bg^kd", and a bottom word of ch^kd. We'll assume wlog. that C is the one of this form.

We argue that the second column of C must be $e20^{k-2}2h$: We can see that every horizontal word (every word starting with "f") either has the prefix "$f2$" or "$f0$". So the second column must be of the form "$e\{0,2\}^kh$". But the only $w \in W$ of this form is "$e20^{k-2}2h$", and so that is what this column must be.

So the second row of C must be of the form "$f20^{i_2-2}10^{k-i_2}f$" for some $1 < i_2 \leqslant k$. Note that $\{v_1 v_{i_2}\} \in E$ by construction.

But this means that the $(i_2+1)^{st}$ column of C must be of the form "$e110^{k-2}h$", since that is the only string in W with a prefix of "$e1$".

So the third row of C must be of the form "$f0^{i_3-1}10^{i_2-i_3-1}10^{k-i_2}$" or "$f0^{i_2-1}10^{i_3-i_2-1}10^{k-i_3}$", where $\{v_{i_2,v_{i_3}}\} \in E$. Note that $i_3 \neq 1$, since $k > 2$ by assumption.

We see that we can continue this process inductively to get a sequence $i_2, i_3, \ldots$. By construction, each $\{v_{i_r}, v_{i_{r+1}}\} \in E$, and so this sequence describes a path in G.

Furthermore, we can see that no i_r can occur more than once in this sequence, since the $(i_r+1)^{st}$ column of C must be "$e0^{r-2}110^{k-r}h$".

But this means that the process must terminate, since there are only $k-1$ values for the i_r. We can see that the only way for this to happen is if the final row is of the form "$f20^{i_k-2}10^{k-i_k}g$".

Note that this means that $\{v_1, v_{i_k}\} \in E.$, and that furthermore, the length of the sequence $(1, i_2, \ldots, i_k)$ must be k.

So the sequence $(v_1, v_{i_2}, \ldots, v_{i_k})$ must describe a Hamiltonian cycle in G.

$\Rightarrow \langle G\rangle \in$ UHAM-CYCLE.

Finally, we can see that A just the $(k+2) \times (k+2)$ zero matrix, where $k = |V|$. So A is polynomial in the size of G. Furthermore W has $4+k+|E|$ strings, all of which are of length $k+2$. So $\langle W, A\rangle$ is polynomial in the size of G. Since every element of A and W ca n also be found in polynomial time, we can see that this construction is polytime, as required.

So CROSSWORD is $\mathcal{NP}$-complete.

■

2.6.2 NONOGRAM

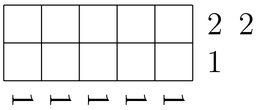
Definition of NONOGRAM (Problem and reduction from (Ueda and Nagao, 1996).):

> **Input:** Two numbers $h, w \in \mathbb{N}$, and two sequences $R = \langle \overline{r_1}, \ldots, \overline{r_h} \rangle$ and $C = \langle \overline{c_1}, \ldots, \overline{c_w} \rangle$ of lists of non-negative integers.
>
> *Note that each $\overline{r_i}$ and $\overline{c_j}$ is itself a series of non-negative integers. That's why we're denoting them as verctors. The lengths of these sequences may vary, though.*
>
> **Question:** Is there an $h \times w$ $\{0, 1\}$ matrix P that satisfies the following properties:?
>
> - For every row $R_{i,\cdot}$ of P, the lengths of the maximal contiguous blocks of 1s match the numbers in $\overline{r_i}$.
> - For every column $C_{\cdot,j}$ of P, the lengths of the maximal contiguous blocks of 1s match the numbers in $\overline{c_j}$.

Let's look at a simple instance of this problem as an example. Suppose we have the following problem instance:

$h = 2, w = 5.$
$R = \langle (2, 2), (1) \rangle$
$C = \langle (1), (1), (1), (1), (1) \rangle$

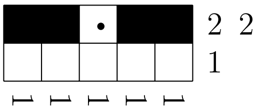
This corresponds to the puzzle



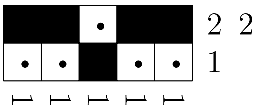
Now, this puzzle is asking us to fill in the top row of the table using two blocks of two ones, and the bottom row with one blaock of a single one. The columns should each have a sincegle filled block.

We'll indicate a filled block (a 1 in P) as a filled black square. If there's a block that we know is a 0 in P, we'll indicate it with a dot ("•"). If we don't know if a square is 0 or 1 it'll be left blank.

Now, in out example puzzle we see that the top row needs two blocks of contiguous ones. There are five spaces, and the two blocks must be seaparate. So they have to be separated by a space in the middle. The only way to do this is to fill it in this way:



Now, each column has exactly one block of one, so the leftmost two columns and the rightmost two columns must have a zero on the bottom, while middle must have a one:



And at this point we've filled the table, and so we've solved the puzzle. So this is a *yes*-instance of NONOGRAM. If there were no way of finishing the puzzle, it would be a *no*-instance.

Reduced from: 3DM.

Note: This is a logic puzzle that can show up under several names including LOGIC ART and PICROSS.

Proof that NONOGRAM is in $\mathcal{NP}$:

1) It is a yes/no problem.

2) A certificate would be the $\{0, 1\}$ matrix P.

3) Since $|R| = h$ and $|C| = w$, and since P is an $h \times w$ $\{0, 1\}$ matrix, this certificate is polynomial in the size of $\langle R, C\rangle$.

4) A verifer V for this language would be:

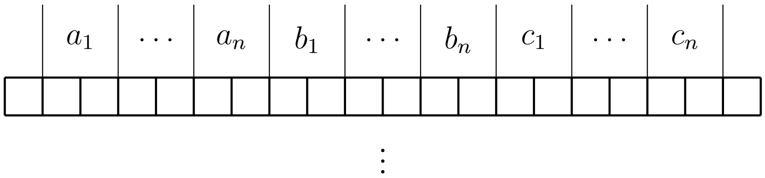
```
$\mathcal{V}$ = "On input $\langle h, w, R, C, P\rangle$:
1. Check that $P$ is an $h \times w$ $\{0, 1\}$ matrix.
   *$\mathcal{O}(hw)$ time.*
2. For $i = 1$ to $h$:
   *$\mathcal{O}(h)$ iterations.*
3.     Loop over $P_{i,\cdot}$ and find the sizes $s_i$ of the contiguous blocks of 1s.
   *$\mathcal{O}(w)$ time after amortization.*
4. Check that $s_i == r_i$, and *reject* if they are different.
   *$\mathcal{O}(w)$ time per iteration.*
5. For $j = 1$ to $w$:
   *$\mathcal{O}(w)$ iterations.*
6.     Loop over $P_{\cdot,j}$ and find the sizes $s'_j$ of the contiguous blocks of 1s.
   *$\mathcal{O}(h)$ time after amortization.*
7. Check that $s'_j == c_j$, and *reject* if they are different.
   *$\mathcal{O}(h)$ time per iteration.*
8. *Accept.*
```

5) We can see that this verifier runs in $\mathcal{O}(hw)$ time, and so is polytime in the size of $\langle h, w, R, C\rangle$ *(again, recall that $|R| = h$ and $|C| = w$).*

Proof that 3DM $\leqslant_p$ NONOGRAM:

Let an instance $\langle A, B, C, T\rangle$ of 3DM be given. We'll construct an instance of NONOGRAM as follows:

- The columns of the nonogram puzzle will roughly correspond to the elements of $A \cup B \cup C$: There will be a set of columns corresponding to the elements of A, then a set of columns for B, and finally a set for C:



- The rows will roughly correspond to the triples in T. Each row will be made up of four "bars" of 1s: One at the left hand side of the table, one separating the A and the B element, one separating the B and the C elements, and one on the right hand side of the table.

  As an example, suppose we have the 3DM instance

  $$A = \{a_1, a_2, a_3\}, B = \{b_1, b_2, b_3\}, C = \{c_1, c_2, c_3\}$$

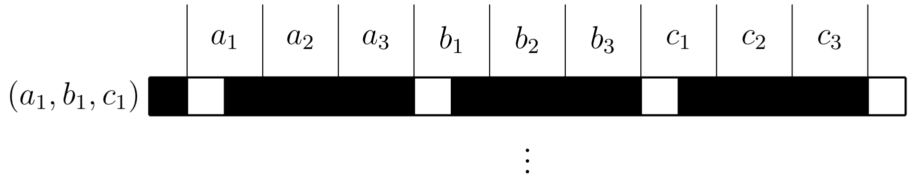
  with the triples

  $$T = \{(a_1, b_1, c_1), (a_1, b_2, c_3), (a_2, b_2, c_3), (a_2, b_3, c_2), (a_3, b_2, c_3), (a_3, b_3, c_2)\}.$$

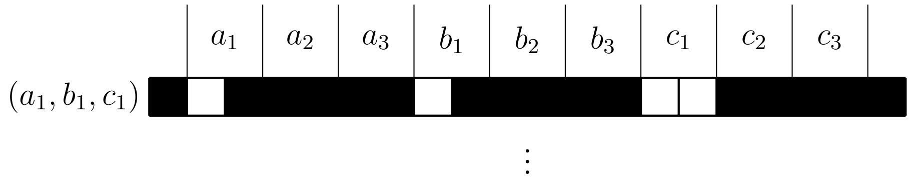
  Note that this is also the example used in (Ueda and Nagao, 1996).

  Then, the first row of our puzzle would correspond to the triple (a_1, b_1, c_1). This row would take the numbers $(1, 5, 5, 5)$ (we'll see why we use those numbers in a minute).

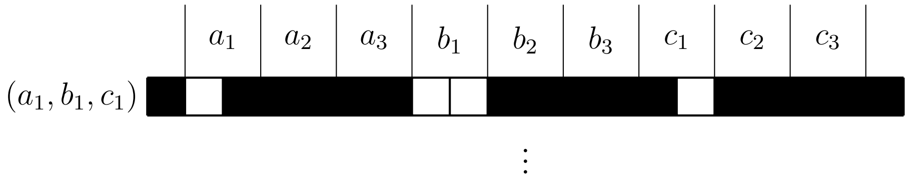
  If we do this, there are five ways of filling out the top row:



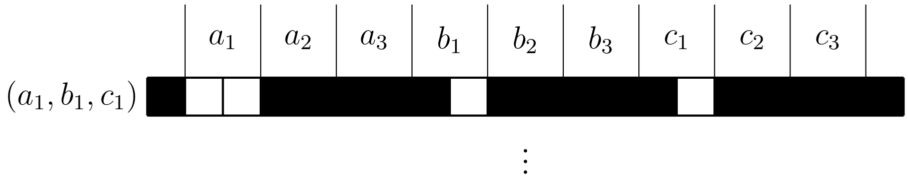
  or



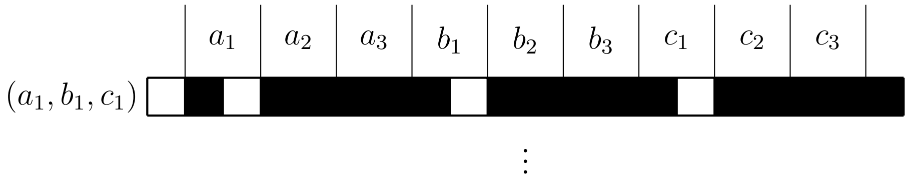
  or



or



or



We'll use the numbers on the columns to ensure that only the second or the fourth configuration is ever possible.

Furthermore, we'll set up our numbers so that the second configuration (the bars shifted to the left) corresponds to a choice of $(a_1, b_1, c_1) \in M$, while the fourth configuration (the bars shifted to the right) corresponds to a choice of $(a_1, b_1, c_1) \notin M$.

You'll notice we've used two columns each for the elements in $A \cup B \cup C$. We've done this so we have just enough room for these configurations, but not enough room for anything else. We've also added two columns on each side. You'll see soon that they act as anchors for our blocks: we'll use the numbers $\overline{c_1}$ and $\overline{c_{6n+2}}$ for these columns to enusre that neither the left-hand block nor the right-hand block can move from the ends of the matrix.

You can now see where the numbers $(1, 5, 5, 5)$ come from: If we're looking at a triple (a_i, b_j, c_k) (in this example (a_1, b_1, c_1)), then:

- The first number should encode a block that extends from the left of the matrix to the start of a_i (a_1 in this example). So the length of the first block is $1 + 2(i - 1) = 2i - 1$ (in our example, this gives $2 \times 1 - 1 = 1$).
- The second number should fill in every column between a_i and b_j, and add one extra square so it intersects exactly one of the a_i or b_j columns. There are n elements in each of A, B, and C, and so the number of elements between a_i and b_j is $n + j - i$.

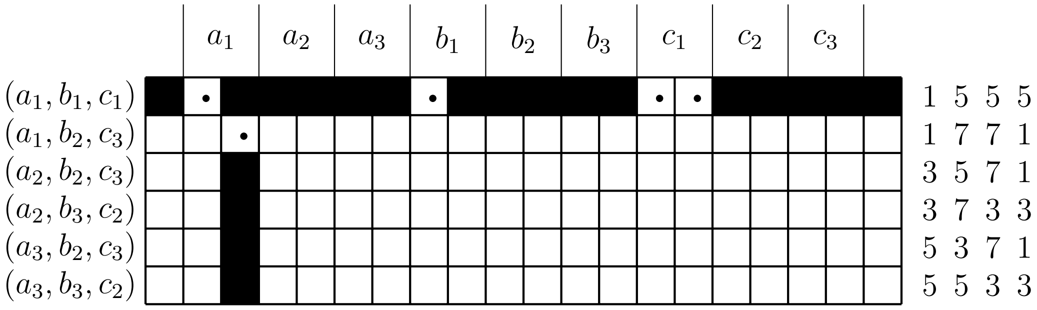
  So in total the second block is of length $2(n + (j - 1) - i) + 1 = 2(n + j - i) - 1$. In our example this gives $2(3 + 1 - 1) - 1 = 5$.

- Similarly, the third number should fill in every column between b_j and c_k, and add one extra square so it intersects exactly one of the b_j or c_k columns. Using a similar argument to the pprevious one, the number of elements between b_j and c_k is $n + k - j$.

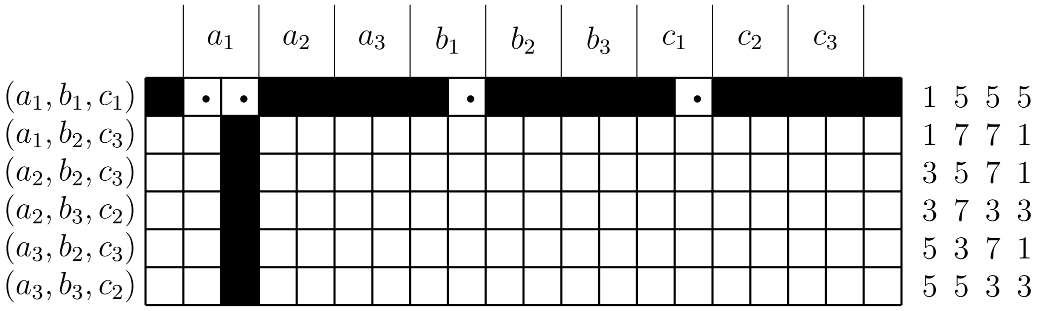
  So in total the third block is of length $2(n + (k - 1) - j) + 1 = 2(n + k - j) - 1$. In our example this gives $2(3 + 1 - 1) - 1 = 5$.
- The final number should encode a block from the end of the c_k columns to the right side of the matrix. So the number here is $2(n - k) + 1$. In our example this gives $2(3 - 1) + 1 = 5$.

You can see that the numbers for the other triples are: $(1, 7, 7, 1)$, $(3, 5, 7, 1)$, $(3, 7, 3, 3)$, $(5, 3, 7, 1)$, and $(5, 5, 3, 3)$.

Now, we can't just stack the rows corresponding to the triples on top of each other: If we did, we'd end up having to encode the value of M into the C vectors. For example, you can see that in our example instance, the element a_1 occurs in two triples: (a_1, b_1, c_1) and (a_1, b_2, c_3). If we use only one row for each triple, the top row and third column would end up being either:


a_1
a_2
a_3
b_1
b_2
b_3
c_1
c_2
c_3
(a_1, b_1, c_1) 1 5 5 5
(a_1, b_2, c_3) 1 7 7 1
(a_2, b_2, c_3) 3 5 7 1
(a_2, b_3, c_2) 3 7 3 3
(a_3, b_2, c_3) 5 3 7 1
(a_3, b_3, c_2) 5 5 3 3

or


a_1
a_2
a_3
b_1
b_2
b_3
c_1
c_2
c_3
(a_1, b_1, c_1) 1 5 5 5
(a_1, b_2, c_3) 1 7 7 1
(a_2, b_2, c_3) 3 5 7 1
(a_2, b_3, c_2) 3 7 3 3
(a_3, b_2, c_3) 5 3 7 1
(a_3, b_3, c_2) 5 5 3 3

Remember that we intend to use the column numbers to force this behaviour!

If we did this, the numbers in $\overline{c_3}$ would either be $(1, 4)$ or (5), and which one we chose would completely determine whether we could choose (a_1, b_1, c_1) or (a_1, b_2, c_3) as out element in M. So we need to fix this behaviour.

The way to do this is to put a one-line break between every triple in our puzzle:

		a_1		a_2		a_3		b_1		b_2		b_3		c_1		c_2		c_3			
(a_1, b_1, c_1)																					1 5 5 5
																					0
(a_1, b_2, c_3)																					1 7 7 1
																					0
(a_2, b_2, c_3)																					3 5 7 1
																					0
(a_2, b_3, c_2)																					3 7 3 3
																					0
(a_3, b_2, c_3)																					5 3 7 1
																					0
(a_3, b_3, c_2)																					5 5 3 3

Now, the choices we get for the top row and third column are:

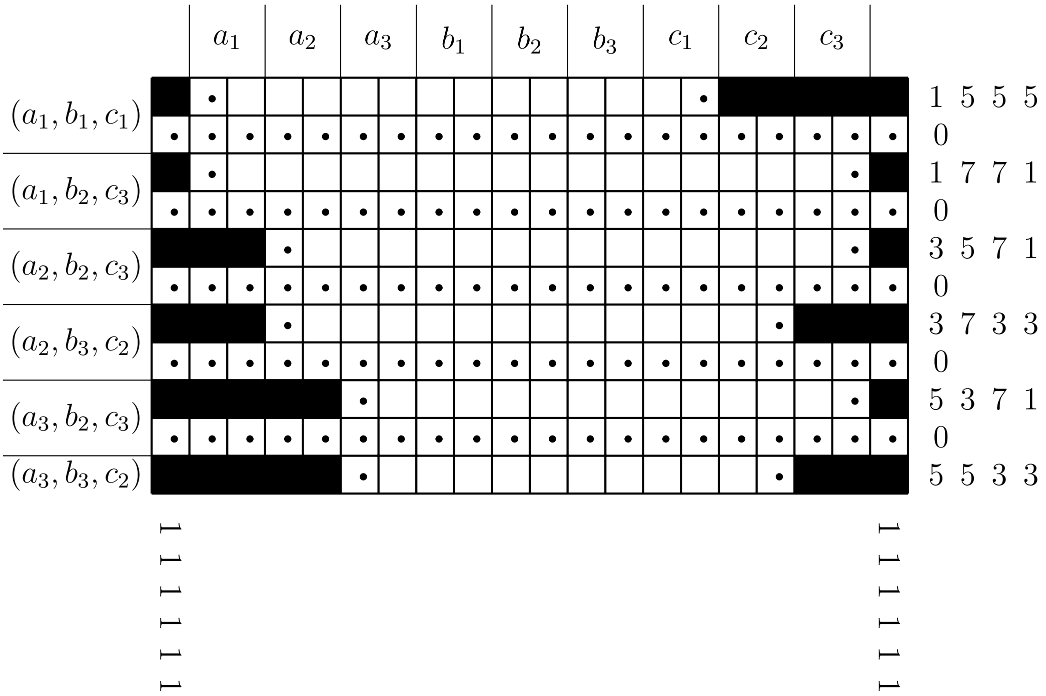
		a_1		a_2		a_3		b_1		b_2		b_3		c_1		c_2		c_3			
(a_1, b_1, c_1)	■	•	■	■	■	■	■	•	■	■	■	■	■	•	•	■	■	■	■	■	1 5 5 5
	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	0
(a_1, b_2, c_3)			•																		1 7 7 1
	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	0
(a_2, b_2, c_3)			■																		3 5 7 1
	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	0
(a_2, b_3, c_2)			■																		3 7 3 3
	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	0
(a_3, b_2, c_3)			■																		5 3 7 1
	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	0
(a_3, b_3, c_2)			■																		5 5 3 3

or

		a_1		a_2		a_3		b_1		b_2		b_3		c_1		c_2		c_3			
(a_1, b_1, c_1)	■	•	•	■	■	■	■	■	•	■	■	■	■	■	•	■	■	■	■	■	1 5 5 5
	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	0
(a_1, b_2, c_3)			■																		1 7 7 1
	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	0
(a_2, b_2, c_3)			■																		3 5 7 1
	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	0
(a_2, b_3, c_2)			■																		3 7 3 3
	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	0
(a_3, b_2, c_3)			■																		5 3 7 1
	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	0
(a_3, b_3, c_2)			■																		5 5 3 3

And in either case we get a $\overline{c_3}$ of $(1, 1, 1, 1, 1)$. So we don't accidentally force a choice of M.

Of course, we've said that we want to force only one of these two possibilities to occur. The first thing we can do to enforce this is to disallow the far-left and the far-right blocks from moving. We can do this by making the lefmost and rightmost column numbers $\overline{c_1}$ and $\overline{c_{20}}$ both be set to $(1, 1, 1, 1, 1, 1)$ (a single 1 for each triple in T). Now, there are six triples in our examples, and six 1s in the leftmost and rightmost columns, and the only rows that can contain 1s are the rows corresponding to the triples. So we have to fill out the ends of our puzzle like so:



To fill in the rest of the $\overline{c_i}$, we'll define a couple of variables.

- Let $t = |t|$ (so $\overline{c_1} = \overline{c_{20}} = (\overbrace{1\ 1\ \dots\ 1}^{t\text{times}})$ in our example).
- For $v \in A \cup B\ cupC$, let $N(v)$ be the number of triples in T that contain v. So in our example, $N(a_1 = 2)$ and $(N(c_3)) = 3$.

Now, we want every a_i to be in exactly one triple in M, and we've said that this corresponds to the choice of pushing all of the bars on a row to the left.

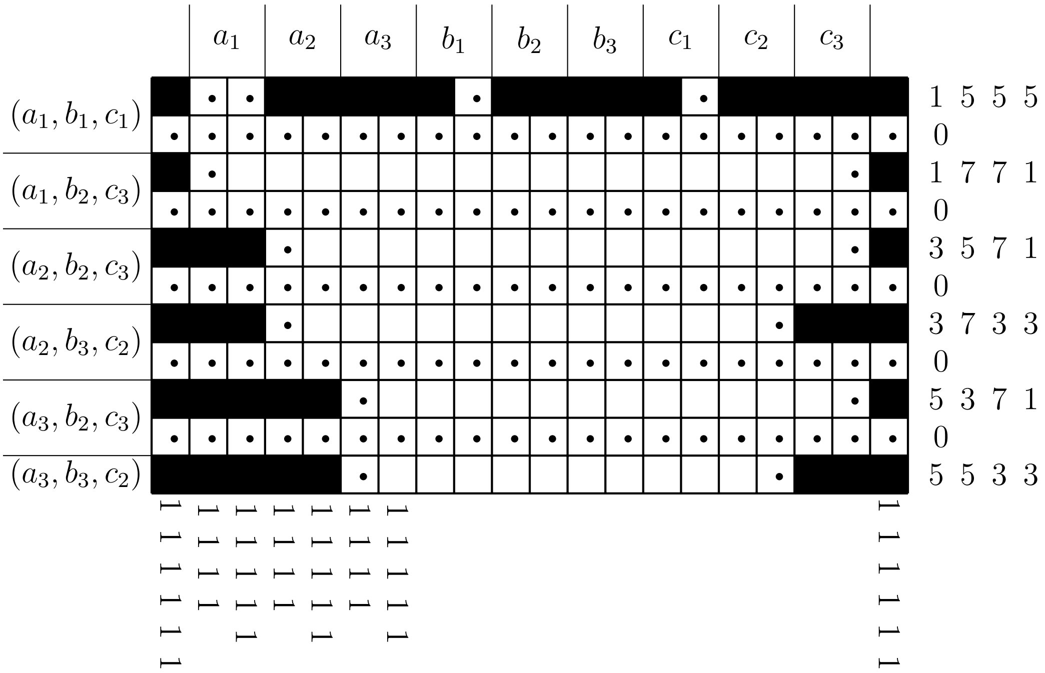
So suppose that, for some triple (a_i, b_j, c_k), the bars will be pushed to the left. Then on the corresponding row the first of the a_i columns will be 0 (since this is where the break after the leftmost row block is), but the second a_i column will be set to 1. Remember that only one row per a_i will satisfy this property.

On the other hand, if the bars are pushed to the right, both a_i columns will be set to 0.

Finally, the a_i columns for every non-zero row corresponding to a triple not containing a_i will both be 1.

So the value given by C for the first a_i column will be $\overline{c_{2i}} = (\overbrace{1, 1, \ldots, 1}^{n-N(a_i \text{ times}})$ (i.e., 1 repeated $n - N(a_i)$ times). The second column will have the numbers $\overline{c_{2i+1}} = (\overbrace{1, 1, \ldots, 1}^{n-N(a_i)+1 \text{ times}})$. In our example this will give us the values $\overline{c_2} = (1, 1, 1, 1)$, $\overline{c_3} = (1, 1, 1, 1, 1)$, $\overline{c_4} = (1, 1, 1, 1)$, $\overline{c_5} = (1, 1, 1, 1, 1)$, $\overline{c_6} = (1, 1, 1, 1)$, and $\overline{c_7} = (1, 1, 1, 1, 1)$.

Note: there are n elements in A, and each one has exactly one row not *pushed to the right, and so the total number of rows pushed to the right is $t - n$. These rows can only be completed in one way, given the constraints we've already given: If we push the first row to the right by setting both a_1 columns to 0, we'd have to fill in the whole row as*



If we chose to set the bars to the left we'd still have a couple of choices. Regardless of the choices we make, though, the current constraints will require n such rows.

Similarly, we want every c_k to be in exactly one triple in M.

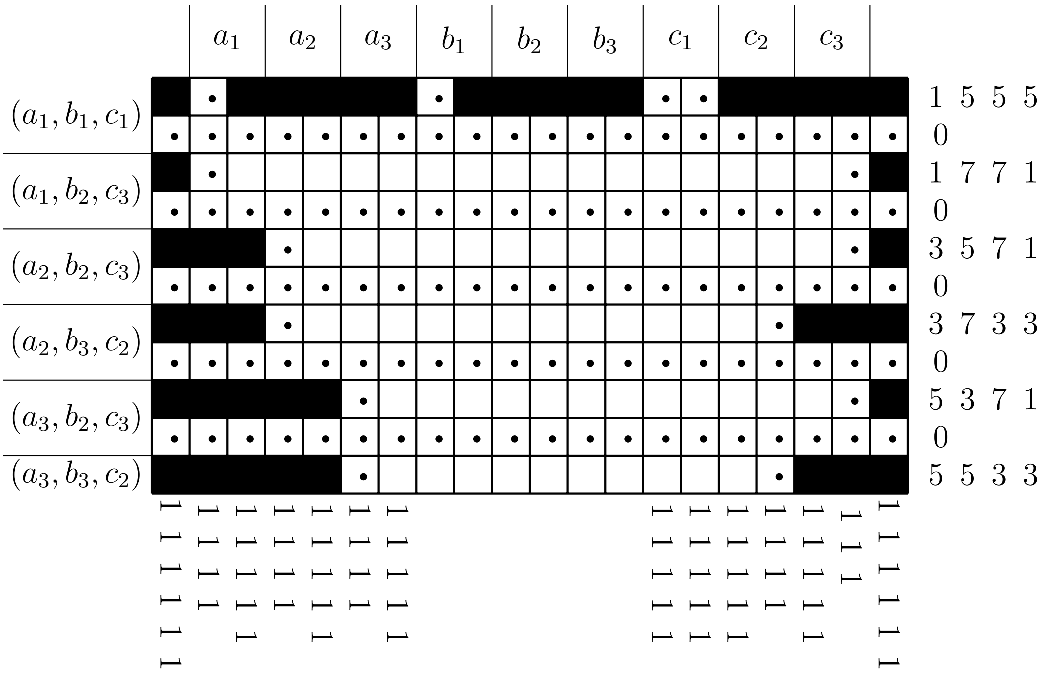
Suppose that, for some triple (a_i, b_j, c_k), the bars will be pushed to the right. Then on the corresponding row the first of the a_i columns will be 1, but the second c_k column will be set to 0 (since this is where the break before the rightmost row block is).

On the other hand, if the bars are pushed to the left, both c_k columns will be set to 0. Remember that only one row per c_k will satisfy this property.

Finally, the c_k columns for every non-zero row corresponding to a triple not containing c_k will both be 1.

So the value given by C for the first c_k column will be $\overline{c_{4n+2k}} = (\overbrace{1, 1, \ldots, 1}^{n-1 \text{ times}})$. The second column will have the numbers $\overline{c_{4n+2k+1}} = (\overbrace{1, 1, \ldots, 1}^{n-N(c_k \text{ times}})$. In our example this will give us the values $\overline{c_{14}} = (1,1,1,1,1)$, $\overline{c_{15}} = (1,1,1,1,1)$, $\overline{c_{16}} = (1,1,1,1,1)$, $\overline{c_{17}} = (1,1,1,1)$, $\overline{c_{18}} = (1,1,1,1,1)$, and $\overline{c_{19}} = (1,1,1)$.

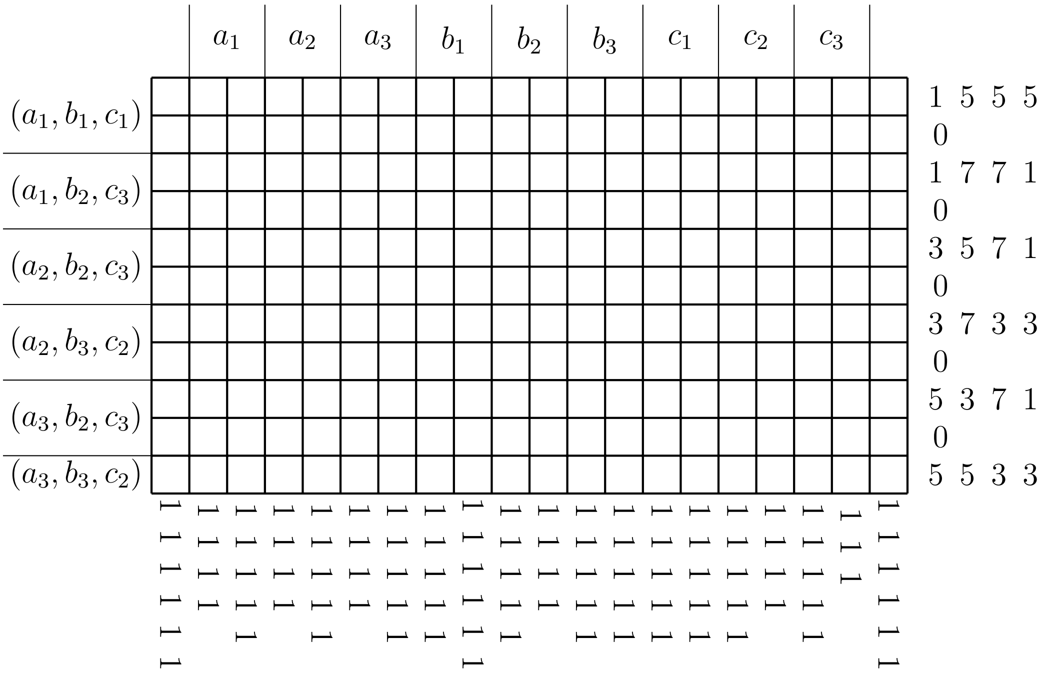
Note: there are also n elements in C, and each one has exactly one row pushed to the left, and so the total number of rows pushed to the left is n. These rows can only be completed in one way, given the constraints we've already given: If we push the first row to the left by setting both c_1 columns to 0, we'd have to fill in the whole row as



So we see that the current constriants on pur puzzle will force us to choose n rows to be pushed to the left, and $t - n$ rows to be pushed to the right, and that every element of each of these rows is determined by this choice. Furthermore, each a_i can occur in exactly one of our left choices, and each c_k occurs in exactly one of our left choices as well. So every way of completing this puzzle corresponds to a size-n subset $M \subseteq T$ in which each element of A and each element of C occurs exactly once. We just need to ensure that every element of B occurs ecactly once in our choice as well.

Now, for any b_j and triple (a_i, b_j, c_k), we know that choosing to include (a_i, b_j, c_k) in M corresponds to bushing its bars to the left. This will leave the leftmost column of b_j as a 0 and the rightmost column as a 1. A choice not to include (a_i, b_j, c_k) in M will likewise leave the leftmost column of b_j as a 1 and the rightmost column of b_j as a 0.

We want to choose exactly one triple with b_j in it, so if we set the left b_j column to $\overline{c_{2n+2j}} = (\overbrace{1, 1, \ldots, 1}^{n-1 \text{ times}})$ and the right b_j column to be $\overline{c_{4n+2k+1}} = (\overbrace{1, 1, \ldots, 1}^{n-N(b_j)+1 \text{ times}})$ we'll get this result. So the final puzzle in our example would be



With this puzzle, every solution will correspond to a choice of n rows to be set to the left, and these rows will correspond to a set of triples. By our choices of the numbers in C, we see that every a_i, b_j, and c_k occurs exactly once in the triples set to the left. So every solution to our NONOGRAM problem corresponds to a solution $M \subseteq T$ to our 3DM instance.

Similarly, if $M \subseteq T$ is a solution to our 3DM instance, then we can set the rows of P corresponding to the triples in M to the left, and the others to the right. We can see that each a_i, b_j, and c_k will occur in exactly one triple, and from our arguments above this means that the columns numbers will be satisfied. So this 3DM solution gives us a solution to our puzzle. This means that the solutions for our NONOGRAM puzzle correspond to solutions to our 3DM instance.

Moreover, we can see that different solutions to our 3DM will correspond to a different choice of rows in our puzzle to be pushed left (i.e., to different puzzle solutions), and vice-versa.

In fact, there is exactly one way to fill in the values of P for every certificate for our original 3DM isntance. If we accept two certificates as equivalent if they encode the same matrix P, then this is a parsimonious reduction.

We set $h = 2t$ (the number of rows in our table) and $w = 2n + 2$ (the number of columns), and R and C to be the row and column numbers. From our discussion we can see that $\langle A, B, C, T\rangle \in$ 3DM iff $h, w, \langle R, C\rangle \in$ NONOGRAM.

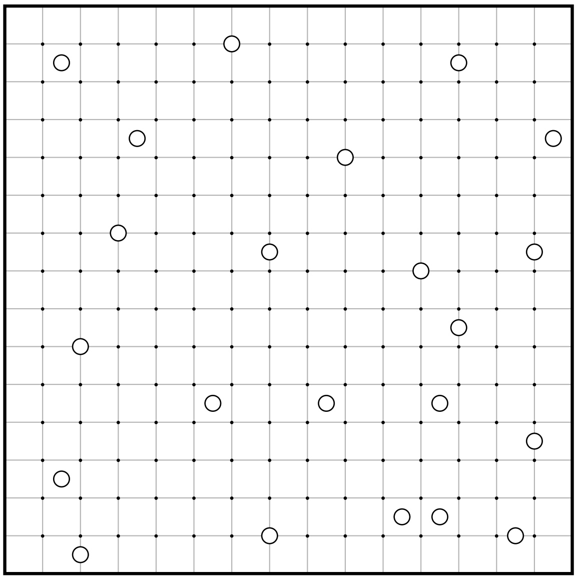
Finally, we can see that our puzzle will contain $w = 2n + 2$ columns and $h = 2t - 1$ rows. So $|R| = 2t - 1$ and $|C| = 2n + 2$. Each $\overline{c_i} \in C$ will contain numbers that sum up to at most h, and so will certainly take up $\mathcal{O}(h)$ space. Similarly, Each $\overline{r_i} \in R$ will contain numbers that sum up to at most w, and so will certainly take up $\mathcal{O}(w)$ space. Determining the values of these numbers takes polynomial time per number, and so the entire reduction is polytime, as required.

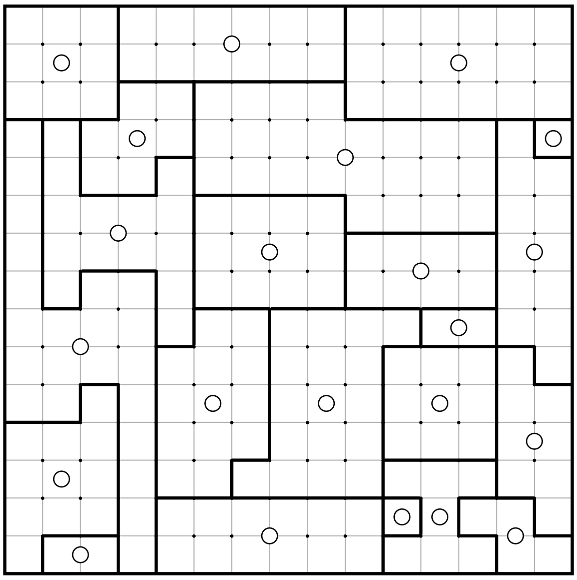
So NONOGRAM is $\mathcal{NP}$-complete.

■

2.6.3 SPIRAL-GALAXIES

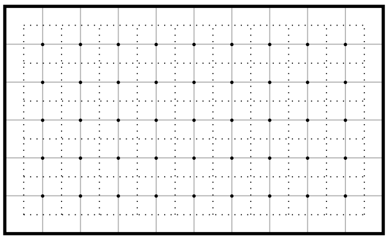
Definition of SPIRAL-GALAXIES (problem and reduction from (Friedman, 2002)):

> **Instance:** An $(m+1) \times (n+1)$ grid G, plus a set $C = \{(x_1, y_1), (x_2, y_2), \dots, (x_r, y_r)\}$ of points, where $\forall 1 \leqslant k \leqslant r, 2x_k \in \{1, \dots, 2n-1\}$ and $y_k \in \{1, \dots, 2m-1\}$.
>
> Formally, G is the graph whose vertices consist of the points (x_i, y_i), where $0 \leqslant x_i \leqslant n$ and $0 \leqslant y_i \leqslant m$. For $0 \leqslant i < n$, and for all j, the pair (i, j) has an edge to $(i+1, j)$. Similarly, for $0 \leqslant j < m$ and for all i, (i, j) has an edge to $(i, j+1)$. The set C is a collection of points either on this grid, or midway between the vertices (but not on the border).
>
> We'll visualize this input as follows:
>
> 
>
> *The grid is as we've described it, and the dots represent the locations of the elements of C.*
>
> **Question:**
>
> Is it possible to choose a subset S of the lines in our grid (the edges in G), where the S partitions the cells in the grid into sections S_1, S_2, ..., S_r such that:
>
> 1) Every elelemt of C is in exactly one section, and every section has exactly one element of C.
> 2) Every element of C is a center of rotational symmetry of its section: If we take an element $(x_i, y_i) \in C$ and rotate its section of the grid and rotate it by 180° around (x_i, y_i), we'll get the same section of the grid.
> 3) No grid section can be a neighbour to itself (i.e., if we choose an edge e to be in S, then the cells on either sides of e are in different grid sections).
>
> If we write the edges we choose in bold, then here would be a solution for our example:



Note that we'll always include the outer border of G in S.

Formally, we can refer to the cells in our grid using a new graph G', where the vertices in G' are of the form (x_i, y_i), $x_i - 0.5 \in \{0, \ldots, n-1\}$ and $y_i - 0.5 \in \{0, \ldots, m-1\}$. We'll connect these vertices in the same way as we do for G:



We'll also include the elements of C in G', and connect every $(x_i, y_i) \in C$ to every grid vertex $(x_{i'}, y_{i'})$ in G' that has an L^∞ distance from (x_i, y_i) of at most 0.5 (i.e., $\max(|x_i - x_{i'}|, |y_i - y_{i'}|) \leqslant 0.5$).

If we choose to include an element $e = \{(x_i, y_i), (x_i + 1, y_i)\}$ in S, then we remove the edge $\{(x_i + 0.5, y_i + 0.5), (x_i + 0.5, y_i - 0.5)\}$ from G' (if that edge indeed exists in G'). Similarly, if we include $e = \{(x_i, y_i), (x_i, y_i + 1)\}$ in S, we remove $\{(x_i + 0.5, y_i + 0.5), (x_i - 0.5, y_i + 0.5)\}$ from G', again if that edge exists.

In this construction, given S we can check:

- Is any element of C on any edge in S?

  If so, it belongs to the sector on both sides of that edge, and so S is an invalid partiton.

- If not, we can find the sections of G induced by S by finding the connected components of G'. We'll then call the sections S_1 to S_ℓ.

  Once we know these components we can ensure that each one contains exactly one element of C.

- Once we know that every section S_k contains exactly one $(x_k, y_k) \in C$, we check that for every element $(x_k + a, y_k + b)$ in S_k, $(x_k - a, y_k - b)$ is also an element of S_k.
- We can also ensure that no section is a neighbour to itself: for any section S_k, if two elements (x, y) and $(x + 1, y)$ are elements of S_k, we check to see if there is an edge between them.

  Similarly, if (x, y) and $(x, y + 1)$ are elements of S_k, we check to see if there is an edge between them.

We'll use these ideas to write our verifier for SPIRAL-GALAXIES.

Reduced from:3SAT.

Firstly, we can see that this is in NP:

1) It is a yes/no problem.
2) Its certificate is the set of paths S, of edges in G.
3) Since each element in S is an edge in G, and since G is part of the input, this certificate is polynomial in the size of the input.
4) A verifier for this certificate would be:

```
Let $\mathcal{V}$ =“On $\langle G = (V, E), C, S\rangle$:”
  1. Reject if any of the following checks fails:
     $\mathcal{O}(1)$ time.
  2. Construct the grid of cells $G' = (V', E')$ described in the question.
     $\mathcal{O}(|V|^2|C|)$ time.
     Note that since $G$ is a grid and $G'$ is a grid with the elements of $C$ added, $|E| = \mathcal{O}(|V|)$
     and $|E'| = \mathcal{O}(|V'|) = \mathcal{O}(|V| + |C|)$.
  3. Remove the edges of $G'$ that cross an edge in $S$.
     $\mathcal{O}(|V|^2)$ time.
  4. Use a BFS or DFS to find the connected components $S_1$, . . ., $S_\ell$.
     $\mathcal{O}(|V| + |E| + |C|) = \mathcal{O}(|V| + |C|)$ time.
  5. For $k = 1$ to $\ell$:
     $\mathcal{O}(|V|)$ iterations.
  6.    Check that $S_k$ contains exactly one element $(x_\ell, y_\ell)$ of $C$, and that this element is not on an edge from $S$.
           $\mathcal{O}(|V| \times |C|)$ time per iteration.
  7.    For every elelent $(x_\ell + a, y_\ell + b)$ in $S_\ell$:
           $\mathcal{O}(|S_\ell|)$ iterations.
  8.       Check that $(x_\ell - a, y_\ell - b) \in S_\ell$.
           $\mathcal{O}(|S_\ell|)$ or $\mathcal{O}(1)$ time per iteration, depending on your implementation.
  9.       If $(x_\ell + a, y_\ell + b)$ is adjacent to an edge in $S$, check that the cell on the other side is not in $S_\ell$.
              $\mathcal{O}(|S_\ell| \times |S|) = \mathcal{O}(|S_\ell| \times |V|)$ time per iteration.
  10.Accept.
     $\mathcal{O}(1)$ time.
```

5) Keeping in mind that the loop from lines 5 to 9 amortizes to $\mathcal{O}(|V|^3 \times |C|)$ time, the total time required by our verifier is $\mathcal{O}(|V|^3 \times |C|)$. So this verifier is polytime in the input $\langle G, C\rangle$.

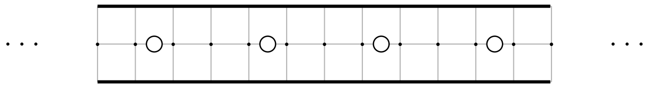
So SPIRAL-GALAXIES is in NP.

Reduction from 3SAT:

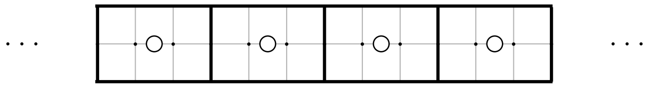
Let an input 3CNF ϕ be given, where ϕ has m variables and n clauses. Note that we can assume using the Helpful 3SAT Lemma from the previous handout that the clauses in ϕ each have three distinct variables.

We'll proceed using the ideas from 1.1: We'll construct a series of wire widgets to transmit a true/false value in SPIRAL-GALAXIES, and we'll create a series of widgets to allow us to do Boolean logic on these values.

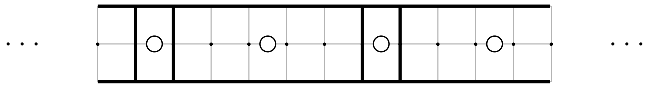
Firstly, we'll introduce our wire widget: it's just a width-2 section of our grid, with periodic dots in C:



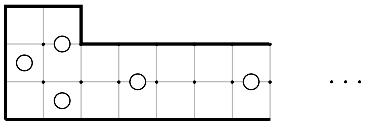
We'll set up our input/output widgets so that there will be two ways of filling these wires: a value of *true* will correspond to the solution



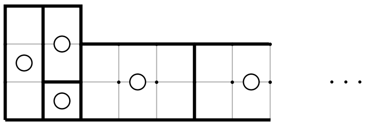
While a value of *false* will correspond to the solution



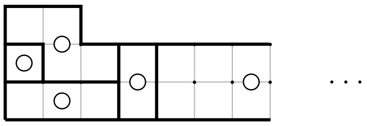
An input/output widget that allows us to choose the value of the wires can be constructed as follows:



The reader can verify that the only two ways of filling out this widget are:

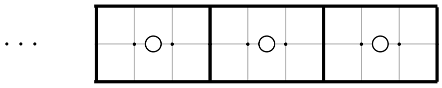


and

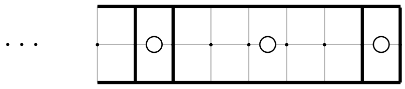


and that these solutions indeed initiate the *true*/*false* sequence we expect to see in our wires.

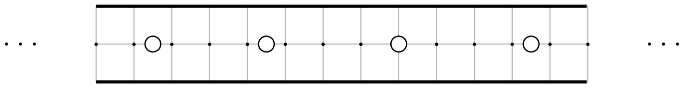
An input or output that forces the wire to take a given value is easy: we just cut off the wire at the appropriate point:



or

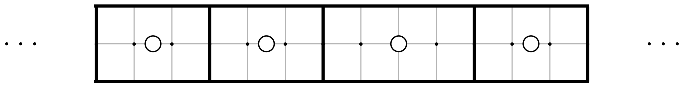


We can see that the patterns that this wire transmits have a period of six cells. In order to connect our wires to other widgets we may need to shift this pattern. We'll do this by adding a half-cell difference to the dots on the wire:

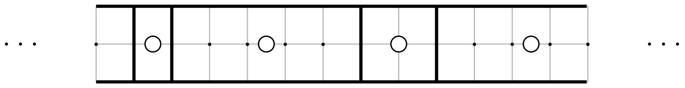


> *Note that we add the extra cell in the section of the pattern that makes a short block when we transmit a* false *signal.*

If the value we're transmitting is *true*, our solution becomes

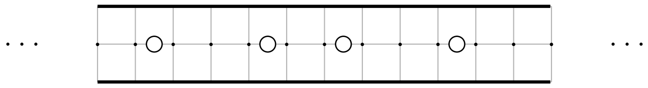


While a value of *false* will correspond to the solution



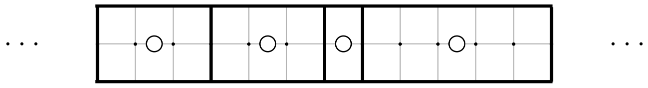
Whichever pattern is being transmitted in the wire is shifted down by one cell. We can repeat this construction to get the desired phase (i.e., if we need to shift the pattern back by one phase, we really shift it forward five times).

On a similar note, we can implement a *not*-gate by moving a dot in our pattern once cell to the left:

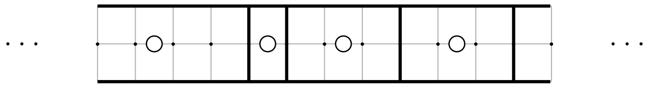


Note: If the signal is false*, there is a pattern of a long and a short box. We move the long box down by one, not the short one. So this picture is shifted from the above examples.*

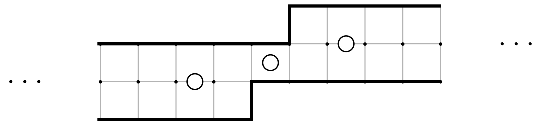
So if we feed in a value of *true*, the output is:



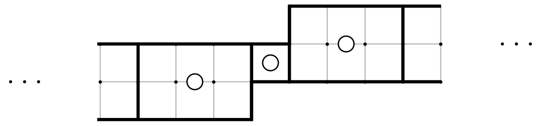
While if we feed in a value of *false* we will get the output



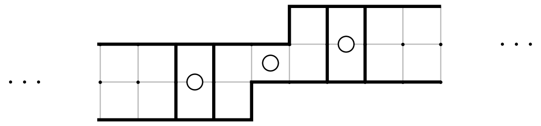
We can furthermore shift our wires one cell up or down as needed with the follwoing widget:



We can see that this widget can be filled in the following two ways:

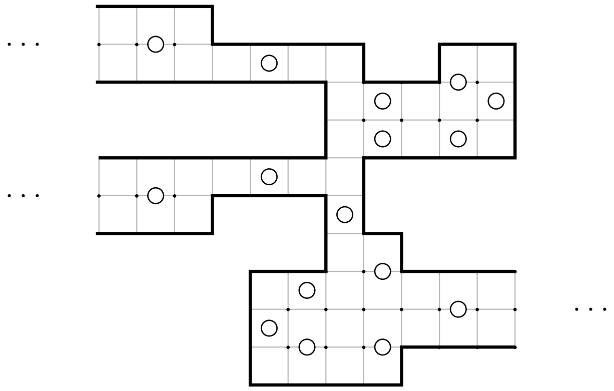


or

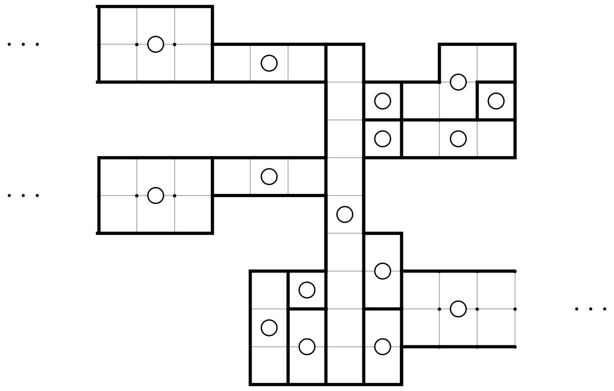


In fact, we can see that this construction is not limited to only taking one side-step at a time – the middle column can be made as wide as we would like.

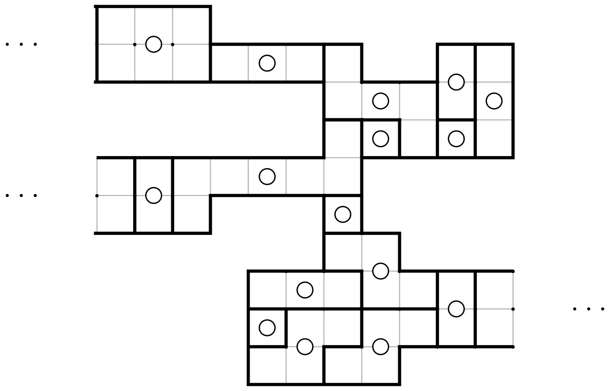
Here is a widget to encode an *And* gate: the inputs are on the left, and the output is on the right.



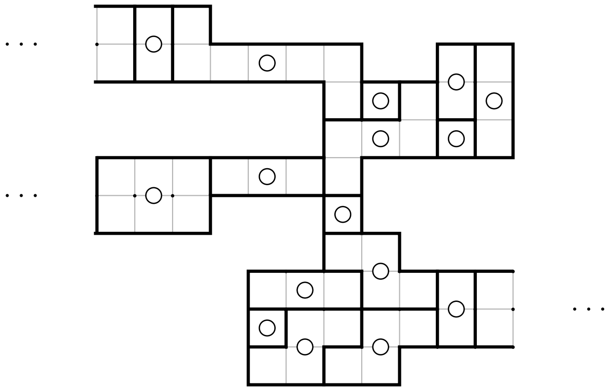
To demonstrate that this widget indeed encodes an *And*-gate, we see that if we feed two values of true into the widget, the rest must be filled out as so:



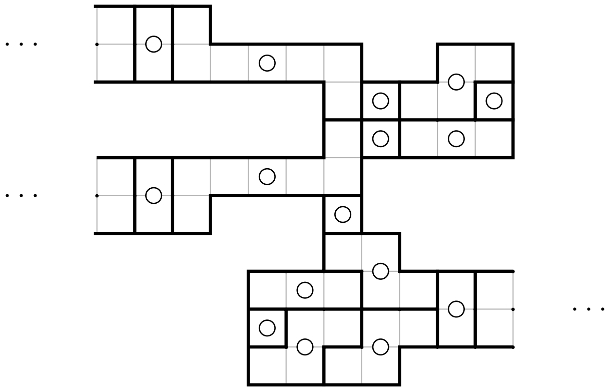
If we fill in a true and a false, the solution must be:



If we fill in a false and a true, the solution must be:



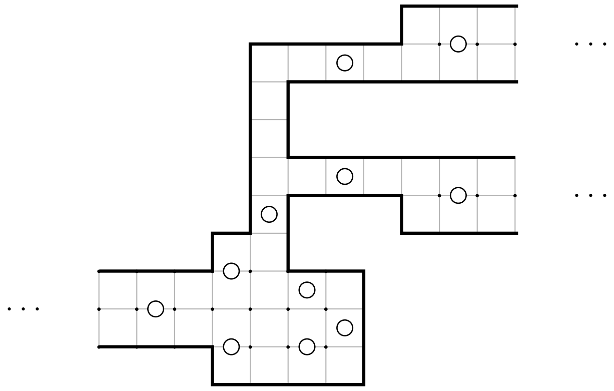
And finally, if both inputs are false, the solution must be:



The reader can verify that these are the only ways to fill out this widget.

In fact, we can see something more: the top right side of the widget must follow one pattern if both inputs are true, and another if both are false. In either case, the output is the same aas the (repeated) inputs.

This suggests that if we remove the top right of this widget and switch the input and the outputs, we'll have a splitter widget as well:

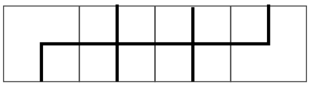


The reader can use the solutions from the *And* widget to verify that if we feed a signal into the bottom left of this widget, the same signal must be repeated on the two output wires on the right.

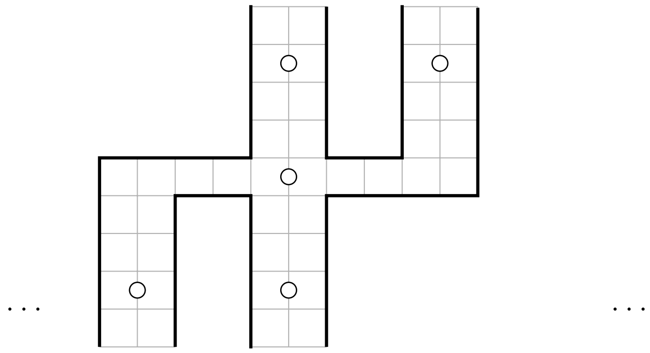
> *In order to bring this construction in line with the construction from 1.1 we'll need to turn these widgets into* or*-gates. But this can be done by adding* not *gates to the input and outputs of the gate.*

The last things we need to do to be able to build circuits as we do in 1.1 would be to show how to allow wires to cross and to allow wired to bend.

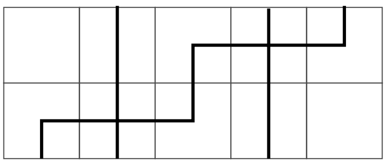
Unfortunately, it is tricky to implement a bend in the wires we use in the SPIRAL-GALAXIES language. A somewhat simpler standby, though, can be built when we note that the bent sections of the wires in the circuits of 1.1only occur when we pss a wire to the right-hand side of the circuit – that is, only in the form (usually with more crossings)



Now, if we cannot bend our wires way from vertical this construction will not be possible. But we can build the following crossover without using bends in our wires:

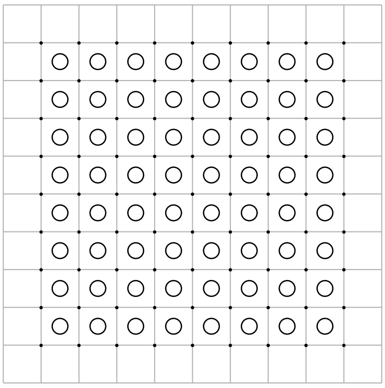


We can then extend each of these each of these sections using a series of staggered crossings:

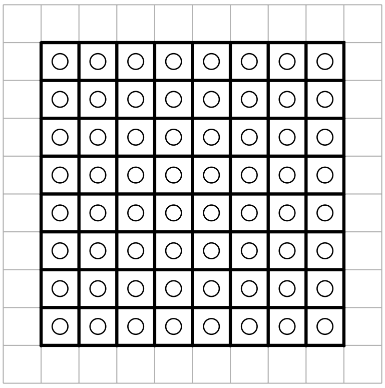


We note that if we modify the construction in 1.1 to allow this sort of grid the overall grid size is still $\mathcal{O}(n^2 + nk)$, where ϕ has n clauses and k variables.

Finally, we note that mny of our widgets are odd shapes surrounded by blank spaces, while the actual puzzle is always given on a rectangular grid. So we'll need to fill in the remaining squares. But we note that every space in the SPIRAL-GALAXIES grid that we don't use in our widgets is contained in a block of height and width at least two. And if we fill all of these grid cells with their own centers:



We can see that the only way to fill these dense blocks is to fence off every cell within them:



So if we use the above widgets to encode our circuit and fill the remaining cells with centers, any solution to the puzzle must fill the remaining widgets in the ways described above. And since the ways of filling the widgets we have correspond to the values on a Boolen circuit, we can see that once we've used these widgets to encode the 3CNF ϕ using the construction from 1.1, the resulting SPIRAL-GALAXIES game will have a solution iff the initial ϕ was satisfiable.

Moreover, we can see that the largest of the widgets that we have described is the *and*-gate, and that this gate has a size of 10×11 cells. Attaching a *not*-gate to each of the inputs and outputs will increase the size of this widget by 7 cells, while adding the sideways shift and phase change widgets will add no more than 42 more cells of width. So we can fit all of our widgets in square tiles of 60×60 cells.

Since, as we have noted bove, we need $\mathcal{O}(n^2 + nk)$ of these tiles to encode ϕ, and since we can costruct and place each tile in polynomial time, the overall reduction is polytime.

So 3SAT $\leqslant_p$ SPIRAL-GALAXIES, and so SPIRAL-GALAXIES is $\mathcal{NP}$-complete.

■

References

Shimon Even, Alon Itai, and Adi Shamir. On the complexity of time table and multi-commodity flow problems. In *16th annual symposium on foundations of computer science (sfcs 1975)*, pages 184–193. IEEE, 1975.

Erich Friedman. Spiral galaxies puzzles are np-complete. *Unpublished manuscript, March*, 2002.

Michael R Garey and David S Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. WH Freeman, 1979.

Michael R Garey, David S Johnson, and Larry Stockmeyer. Some simplified np-complete problems. In *Proceedings of the sixth annual ACM symposium on Theory of computing*, pages 47–63, 1974.

R KARP. Reducibility among combinatorial problems. *Complexity of Computer Computation*, pages 85–104, 1972.

Richard Kaye. Minesweeper is np-complete. *Mathematical Intelligencer*, 22(2):9–15, 2000.

Jaroslav Opatrny. Total ordering problem. *SIAM Journal on Computing*, 8(1):111–114, 1979.

Nobuhisa Ueda and Tadaaki Nagao. Np-completeness results for nonogram via parsimonious reductions. *preprint*, 1996.