CS624 - Analysis of Algorithms

NP completeness

November 26, 2025

Introduction to Complexity Classes

- We have talked about problems that are $O(n)$ or $O(n^2)$, or $O(n \log n)$, and so on.
- When we talk like this, n is some measure of the size of the problem. For instance if we are talking about sorting a set, then n will be the number of elements of the set.
- If we are talking about a graph $G = (V, E)$, then it is reasonable to let n be something like $|V| + |E|$ etc.
- In general, a problem instance a is *encoded* in some way, and n is just another name for $|a|$, which is the length of the encoding.

Introduction to Complexity Classes

Definition

The class P is the class of problems for which there is a number k and an algorithm which solves the problem and whose running time is $O(n^k)$ where n is the size of the instance of the problem.

- This is called the class of *polynomial-time* problems.
- All the problems we have seen in this course so far are in P, almost always with a very small exponent.

Class P – A (semi) Formal Definition

- Class P contains all *decision problems* that can be solved by a *deterministic Turing machine* using a polynomial amount of computation time
- We can semi-formally think of a Turing machine as an algorithm that solves a particular problem (it's not a real machine!)
- In a deterministic Turing machine, at every state of the algorithm (a combination of the input and stage of the computation) we have at most one way to proceed.
- Everything we saw thus far falls into this category. Even "random" numbers.
- The polynomial time is with respect to the space it takes to represent the input.
- We'll define decision problems shortly.

Polynomial Reducibility

- Reducing a problem to an instance of another problem is a common practice – we have also seen it in this course.
- Example – we proved that the average-case running time of building a binary search tree was $\Theta(n \log n)$ by showing that it was really the same algorithm as Quicksort.
- We *reduced* the BuildBinarySearchTree algorithm to the Quicksort algorithm, and also reduced the Quicksort algorithm to the BuildBinarySearchTree algorithm – that is, there was a reduction in both directions

Polynomial Reducibility

- There are other reductions that only go in one direction, but are still quite useful.
- The one we are concerned with here is called *polynomial reduction*.
- We consider a specific class of problems called *decision problems* which are problems for which the answer is "yes" or "no".
- Some problems are naturally decision problems. For instance, there is the Hamiltonian cycle problem: given an undirected graph $G = (V, E)$, is there a simple cycle that contains every vertex in V?
- An *instance* of the Hamiltonian cycle problem is an undirected graph G, together with the question, "Does G have a Hamiltonian cycle?"

Polynomial Reducibility

- Other problems, such as optimization problems, are not naturally decision problems but they can be formulated as such.
- For instance, we say that an *independent set* in an undirected graph $G = (V, E)$ is a subset V_1 of the vertices V such that no two vertices in V_1 are joined by an edge in E.
- The "maximal independent set" problem is, given a graph G, to find the largest independent set in G.
- This is not a decision problem but we can formulate it "Given a positive integer k, is there an independent set V_1 in the graph of size k?
- The pair (G, k) is an *instance* of the independent set problem.

Polynomial Reducibility

- Given two decision problems (like HAMILTONIAN CYCLE and INDEPENDENT SET, for example). Let us call them A and B.
- Suppose we have a function f having the following properties:
  - $f : A \rightarrow B$. That is f maps instances of A into instances of B.
  - f is implemented by an algorithm that runs in polynomial time.
  - If a is an instance of the problem A, then the time to compute $f(a)$ is $O(|a|^k)$ for some k, where $|a|$ is the size of the instance a.
  - For each $a \in A$, a has the same answer ("yes" or "no") as $f(a)$.
- In such a case, we say that A is *polynomial-time reducible* to B, and we write $A \leq_P B$

Polynomial Reducibility – Some Properties

- If $A \leq_P B$, then not only is $f(a)$ computable from a in polynomial time, but $|f(a)|$ is a polynomially bounded function of $|a|$. In other words – $|f(a)| = O(|a|^k)$ for some k.
- This is because for some k,f only runs for $O(|a|^k)$ time and therefore cannot output an encoding for $f(a)$ that is longer than that.
- If $A \leq_P B$, and if B is a problem in P, then A is also in P. We can take any instance a of A, transform it in polynomial time by f to a problem $f(a)$ in B whose size is no more than some fixed power of $|a|$, and then solve that problem by a polynomially bounded algorithm (since B is in P).

Polynomial Reducibility – Some Properties

- In other words, suppose $|f(a)| = O(|a|^p)$, and suppose the algorithm for solving problem instances of B is $O(|b|^q)$.
- Then the cost of solving $f(a)$ is $O\left(\left(|a|^p\right)^q\right) = O(|a|^{pq})$.
- Since $f(a)$ has the same answer ("yes" or "no") as a does, solving $f(a)$ solves a. Therefore the polynomially bounded algorithm for B yields a polynomially bounded algorithm for A.
- In the other direction: if $A \leq_P B$ and A is in some sense hard to solve, then B must be also.
- For instance, if we knew that there was no polynomial-time algorithm for A, then we would also know that there could be no polynomial-time algorithm for B – as such an algorithm for B would immediately yield one for A as well.

The Class NP

- There is a very large class of problems for which no polynomial-time algorithm has been found
- However, they can be *checked* in polynomial time.
- For instance, no polynomial-time algorithm is known for the Hamiltonian cycle problem but if we get a a Hamiltonian cycle as a list of vertices for a graph G, it would be easy to check that it was indeed a Hamiltonian cycle (or wasn't):
  - Check that the list included all the vertices once and none twice, and
  - And that between each two consecutive vertices in the list (and between the first and the last) there was an edge in the graph.

The Class NP

- This can obviously be done in linear time
- The class NP of problems is the the class of problems that can be checked in polynomial time.
- Certainly $P \subseteq NP$
- But there are many problems in NP that are not necessarily "easy" to solve and there are some problems that are as hard as any problems in NP.

The Class NP – (semi) Formal Definition

- Class NP (Nondeterministic Polynomial) contains all *decision problems* that can be decided by a *non-deterministic Turing machine* using a polynomial amount of computation time
- In a non-deterministic Turing machine, at every state of the algorithm (a combination of the input and stage of the computation) we have at least one way to proceed.
- At each step of the computation every choice spawns a set of possible steps.
- We have a tree of computations rather than just one linear sequence.
- If the longest path leading to termination is polynomial in the size of the input, the problem is decided in polynomial time by the machine.
- This is definitely not a very practical model...

Equivalence of the Two Definitions

- If a problem can be decided by a *non-deterministic Turing machine* using a polynomial amount of computation time
- We have a tree of computations rather than just one linear sequence.
- If the longest path leading to termination is polynomial in the size of the input, the problem is decided in polynomial time by the machine.
- Polynomial verification means that if you are given a path in the tree (a possible solution), you can follow it and tell whether it is true or not.

NP-Completeness

Definition

A problem A is *NP-hard* iff for any problem B in NP, B $\leq_P$ A

Definition

A problem A is *NP-complete* iff

- A is in NP, and
- A is NP-hard.

From this it follows that all problems that are NP-complete are polynomially equivalent, in the sense that if A and B are NP-complete, then

$$A \leq_P B$$
$$B \leq_P A$$

NP-Completeness

- To show that a problem A is NP-complete, it is enough to show that
  - A is in NP, and
  - for some problem B that is NP-complete, $B \leq_P A$
- The reason is if C is any problem in NP, we must then have $C \leq_P B \leq_P A$ which shows that A is in fact NP-complete.
- We are now going to give some examples of problems that are NP-complete.

The First NP-Complete Problem

- The following problem is known to be NP-complete.
- In fact, it is historically the first problem that was proved to be NP-complete.
- We will show it is NP-complete later, but for now it suffices to see that it is clearly in NP, and all the very best algorithms experts in the world have tried to find a polynomial-time algorithm for it, and have failed.
- So it is reasonable to assume that it is NP-complete. We will later prove that it is.

SAT

- It is a problem in mathematical logic, which sounds very abstract but closely related to problems in chip design.
- We have a set of Boolean variables. Let us call them $\{v_1, v_2, \ldots, v_n\}$, such that each variable can take on either the value True or False.
- We make Boolean expressions using these variables and three operators:

$\vee$	this means "or"
$\wedge$	this means "and"
$\bar{v}$	this means "not v"

  and parentheses, which we use in the usual way
- An expression such as $a \vee b$ is called a *disjunction*, and an expression of the form $a \wedge b$ is called a *conjunction*

SAT – A CNF Expression

- A *literal* is a very simple expression which is either v or $\bar{v}$ for some variable v.

Definition

A Boolean expression e is in *conjunctive normal form* (CNF) if it is of the following form: $e = c_1 \wedge c_2 \wedge \cdots \wedge c_m$ where each c_k is a *clause*, which by the same definition is of the form $c_k = (z_1^{(k)} \vee z_2^{(k)} \vee \ldots z_{n_k}^{(k)})$ where each $z_j^{(k)}$ is a literal.

- For instance, the expression

$$e = (v_1 \vee \bar{v}_2 \vee \bar{v}_3 \vee v_4) \wedge (v_1 \vee v_2 \vee \bar{v}_5) \wedge (v_3 \vee v_4 \vee v_5) \wedge (v_2 \vee v_4 \vee \bar{v}_5)$$

is an expression in CNF.

Satisfiability

- An expression in CNF is *satisfiable* iff there is an assignment of True and False values to each of the variables v_j which makes the expression True.
- In this example if we set $v_1 = v_4 = True$, then e will be True, regardless of the values of the other variables. So e is satisfiable.
- The problem SAT is, given an expression in conjunctive normal form, to determine if it is satisfiable.
- This is a remarkably difficult problem. There is no known way to definitively solve it other than by exhaustive search.
- On the other hand, it is clearly in NP – if someone tells you a solution, you can check that solution in linear time.

Some Basic NP-Complete Problems – 3-SAT

- 3-SAT is a restricted form of SAT in which all clauses have exactly 3 literals in them.
- It doesn't simplify the problem... We'll show that 3-SAT is just as hard as SAT, and is therefore NP-complete.

Theorem

3-SAT is NP-complete.

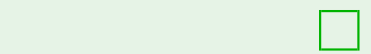
Proof.

3-SAT is in NP. This is straightforward: We can check an assignment to the variables of a 3-SAT expression by substituting them in each clause and verifying that each clause evaluates to True. This is certainly an $O(n)$ operation (where n is the number of literals in the whole expression). □

Cont.

3-SAT is NP-hard. We prove this by showing that SAT $\leq_P$ 3-SAT.

- We start with a SAT formula: $c_1 \wedge c_2 \wedge \cdots \wedge c_m$ where each c_k is a clause of the form $c_k = (z_1^{(k)} \vee z_2^{(k)} \vee \ldots z_{n_k}^{(k)})$ where each $z_j^{(k)}$ is a literal – that is, it is either x or $\bar{x}$, where x is some variable.
- We have to show how to turn this into a 3-SAT expression – an equivalent expression in which all the clauses have exactly 3 literals, and such that the the algorithm that does this runs in polynomial time (in the size of the original expression).
- The size of the final expression will be polynomially bounded in terms of the size of the original expression.



Cont.

- We consider each clause separately. We replace each clause c_j by a set of clauses C_j such that
  - Each of the new clauses will have exactly 3 literals in it.
  - The variables in the clauses in C_j will be the variables in c_j together with possibly some new variables. But the new variables will occur only in the clauses in C_j, and not in any other clauses in any other C_k.
  - c_k will be True iff each clause in C_k is true
    - with the *same* values given to the variables of c_k
    - and with *some* values given to the new variables.

Cont.

There are four cases to consider:

$|c_k| = 1$. That is, $c_k = z_1$. We introduce two new variables y_1 and y_2, and we set

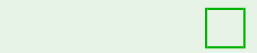
$$\begin{aligned} C_k = \{&(z_1 \vee y_1 \vee y_2) \wedge \\ &(z_1 \vee y_1 \vee \bar{y}_2) \wedge \\ &(z_1 \vee \bar{y}_1 \vee y_2) \wedge \\ &(z_1 \vee \bar{y}_1 \vee \bar{y}_2)\} \end{aligned}$$

Cont.

$|c_k| = 2$. $c_k = (z_1 \vee z_2)$. We introduce one new variable y_1 and set:

$$C_k = \{(z_1 \vee z_2 \vee y_1) \wedge (z_1 \vee z_2 \vee \bar{y}_1)\}$$

$|c_k| = 3$. $c_k = (z_1 \vee z_2 \vee z_3)$. In this case, there is nothing to do: we just set $C_k = \{c_k\}$.



Cont.

$|c_k| \geq 4$. We have

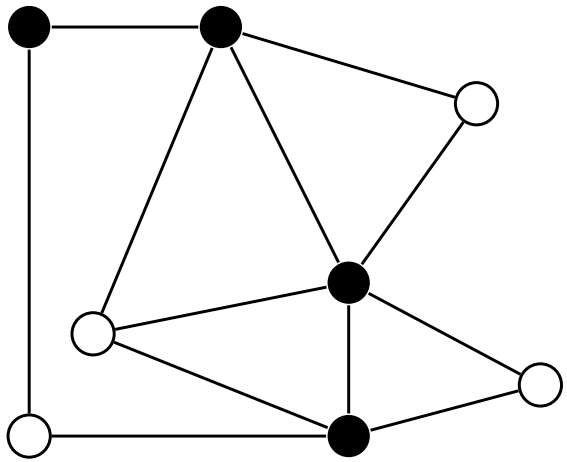
$$c_k = z_1 \vee z_2 \vee \ldots z_{n_k}$$

where $n_k \geq 4$. We introduce new variables $y_1, \ldots, y_{n_k-3}$ and set

$$\begin{aligned} C_k = \{ & (z_1 \vee z_2 \vee y_1) \wedge \\ & (\bar{y}_1 \vee z_3 \vee y_2) \wedge \\ & (\bar{y}_2 \vee z_4 \vee y_3) \wedge \\ & \ldots \\ & (\bar{y}_{n_k-3} \vee z_{n_k-1} \vee z_{n_k}) \} \end{aligned}$$

Vertex Cover (VC)

- A *vertex cover* of an undirected graph $G = (V, E)$ is a subset of vertices $V_1 \subseteq V$ such that every edge $e \in E$ is incident on (at least) one element of V_1.
- In the figure the black vertices constitute a vertex cover.
- Given an undirected graph, the corresponding decision problem is thus "Is there a vertex cover of size k?" (k is as small as possible...)
- An instance of VC is a pair (G, k) where G is a graph, and the question is "Is there a vertex cover of G of size k?"



Vertex Cover (VC)

Theorem

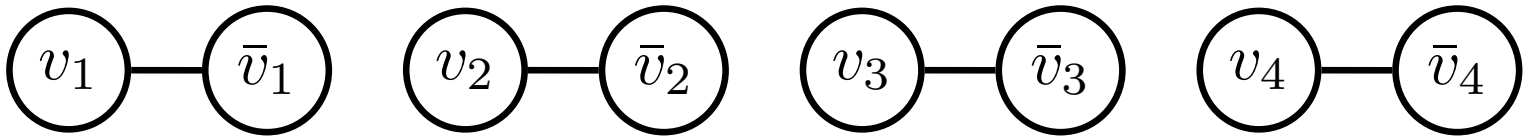
The vertex cover problem is NP-complete.

Proof.

1. **VC is in NP.** Clearly, given a set $V_1 \subseteq V$, we can check that the size of V_1 is k and that each edge $e \in E$ is incident on a vertex in V_1 in $O(E + V)$.
2. **VC is NP-hard.** We prove this by showing that 3-SAT $\leq_P$ VC.
   Let us start with a 3-SAT instance with N variables and C clauses. We will construct a graph with $2N + 3C$ vertices such that
   - The construction can be done in "polynomial time".
   - The original 3-SAT instance is satisfiable iff the graph we construct has a vertex cover with $N + 2C$ vertices.

Vertex Cover (VC)

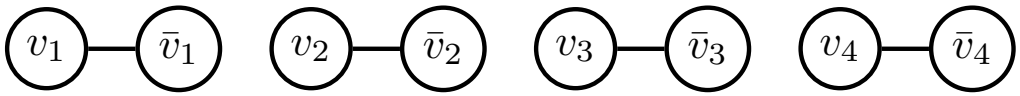
- We show a specific example but the construction will be described in perfectly general terms, so it applies to any 3-SAT instance.
- The graph we will construct consists of three parts.
- The first part consists of pairs of vertices, one pair for each variable in the instance. Each pair is labeled with the variable and its negation, and the pair is connected by an edge, as in the previous figure. This part of the graph consists of the *truth-setting components*.

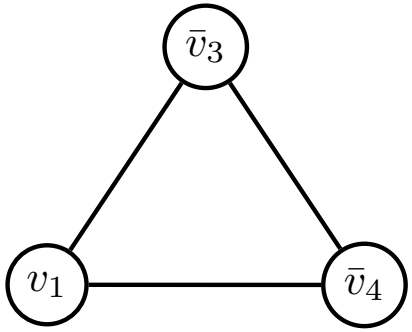


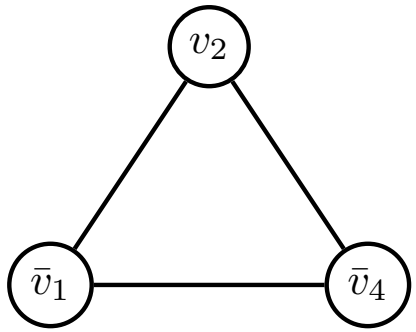
Stage 1 of the construction of the graph corresponding to the 3-SAT instance $(v_1 \vee \bar{v}_3 \vee \bar{v}_4) \wedge (\bar{v}_1 \vee v_2 \vee \bar{v}_4)$

Vertex Cover (VC)

- The second part consists of a triangle of nodes for each clause in the 3-SAT instance.
- The nodes of the triangle are labeled by the literals in the clause.
- This second part of the graph consists of the *satisfaction-testing components*.



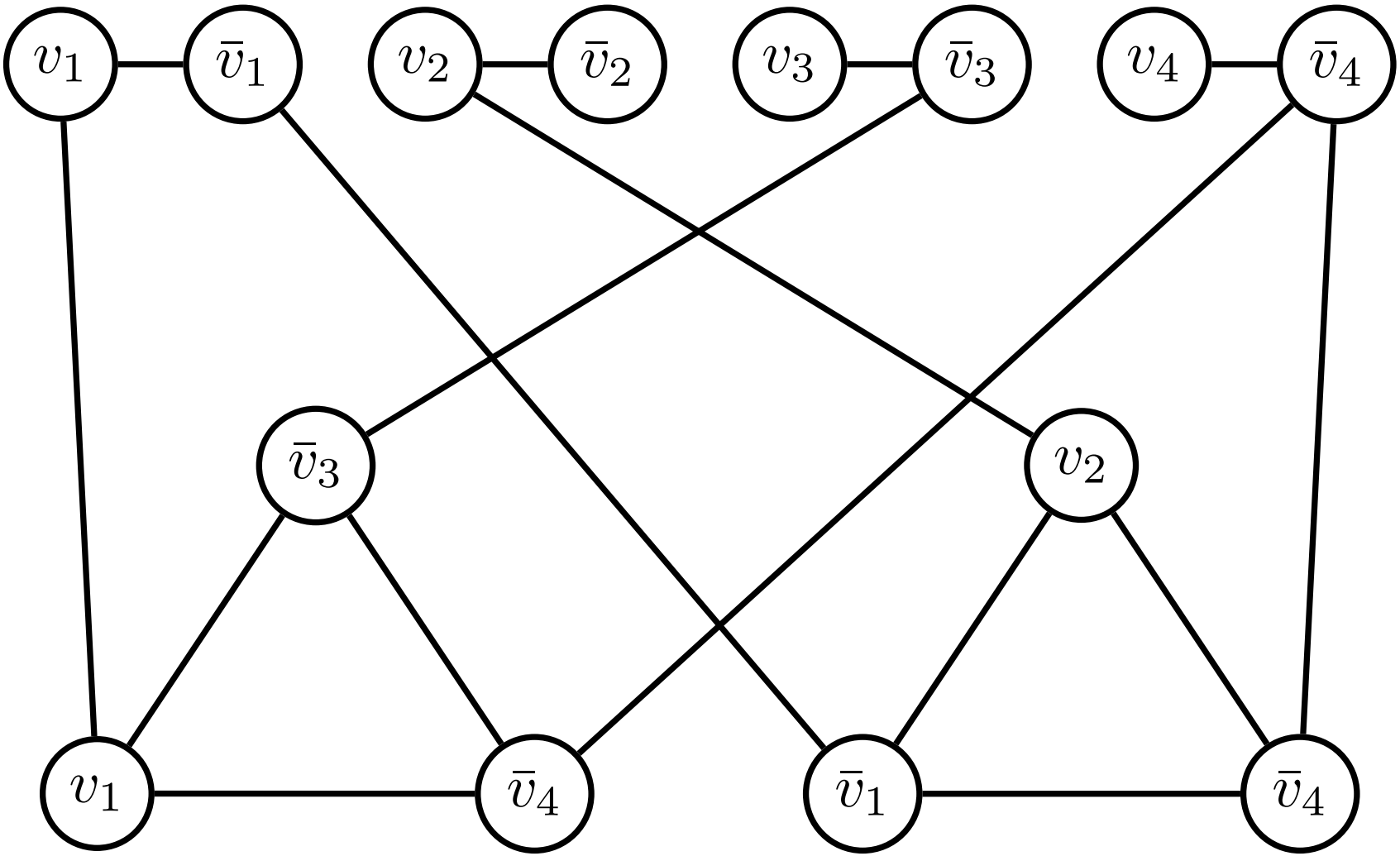




Vertex Cover (VC)

- Finally, we add edges between the truth-setting components on top and the satisfaction-testing components on the bottom.
- These edges encode the literal values in the graph.
- We attach every node on the bottom to its node of the same name on the top.
- We can refer to these edges as *cross edges*.
- Notice that this part is the only one that is specific to the assignment itself

Vertex Cover (VC)



What Does a VC in that Graph Look Like?

- Every one of the truth-setting edges (on the top) must be covered, so a vertex cover must include at least one of every pair of truth-setting vertices on the top. (at least N vertices of this type).
- It must include at least 2 out of the three vertices of each satisfaction-testing triangle on the bottom, because the edges of those triangles can't be covered in any other way (at least $2C$ vertices of that type).
- Any vertex cover of the graph must include at least $N + 2C$ vertices.
- The only remaining issue is whether the cross edges are covered.
- Proving this will prove the reduction is valid.

Validity of the Reduction – Direction 1

Lemma

If the original 3-SAT instance is satisfiable, then the derived graph has a vertex cover of size $N + 2C$

Proof.

We construct our vertex cover as follows:

- For each pair of truth-setting vertices, take the True one
- Since each "triangle" must have at least one vertex corresponding to a True literal, the cross edge coming to that vertex will already be covered by it.
- Pick the other two vertices for the vertex cover. This way all the cross edges to that triangle are also covered.

Thus, a satisfying set of truth values for the N variables in the original 3-SAT instance corresponds to a vertex cover of size $N + 2C$ of the derived graph □

Validity of the Reduction – Direction 2

Lemma

If the derived graph has a vertex cover of size $N + 2C$, then the original 3-SAT instance is satisfiable.

Proof.

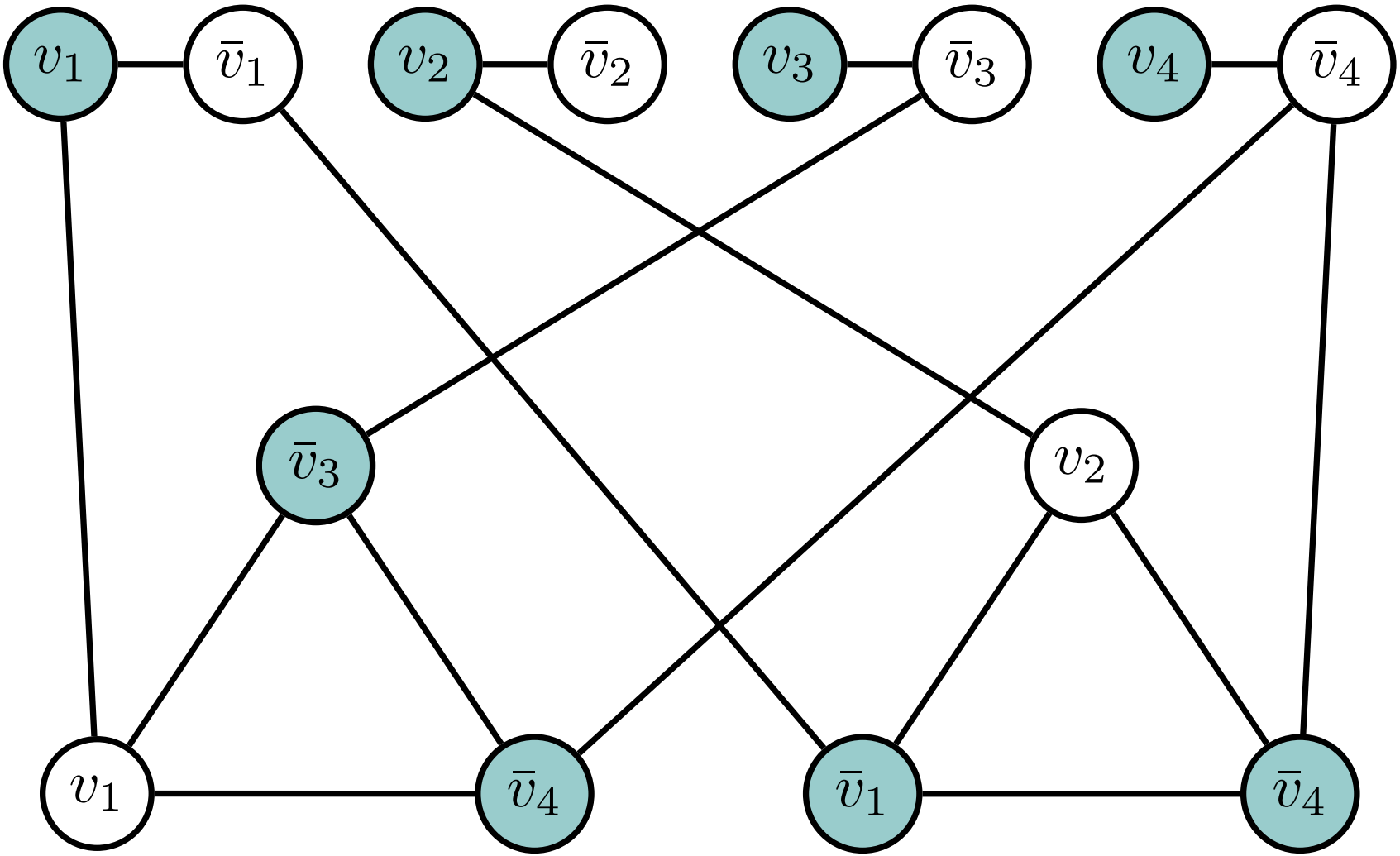
- Suppose we have a vertex cover of the size $N + 2C$ of the derived graph.
- We know that N of the "top" vertices are part of the cover and $2C$ of the "bottom" ones are also – two in every triangle.
- Let the N vertices on the top specify the truth values of each of the N variables.

Validity of the Reduction – Direction 2

Cont.

- The vertex in each triangle that is not part of the cover must be true because it is one endpoint of a cross edge, and since we have a vertex cover, the other endpoint of that cross edge must be part of the cover, and so that literal is True.
- Thus, at least one literal in each clause is True, and so the original 3-SAT instance is satisfiable.
- Notice that it is clearly a polynomial time construction, and this completes the proof.

Example


v_1
$\bar{v}_1$
v_2
$\bar{v}_2$
v_3
$\bar{v}_3$
v_4
$\bar{v}_4$
$\bar{v}_3$
v_2
v_1
$\bar{v}_4$
$\bar{v}_1$
$\bar{v}_4$

CLIQUE and INDEPENDENT SET

- A clique in an undirected graph is a subset of vertices such that each pair of the vertices is joined by an edge in the graph. (Equivalently, a clique is a complete subgraph.)
- Given an undirected graph, we would like to find the largest clique. The corresponding decision problem is "Given a graph G and a number k, does G contain a clique of size k?".
- An instance of CLIQUE is again a pair (G, k) where G is a graph, k is a number, and the associated question is as above.
- An independent set in a graph is a set of vertices such that no two vertices in the set are joined by an edge.
- The INDEPENDENT SET problem is, given a graph, to find the largest independent set. The equivalent decision problem is "Does G contain an independent set of size k?".
- An instance of INDEPENDENT SET is a pair (G, k) where G is an undirected graph, k is a positive integer, and the associated question is as above.

CLIQUE and INDEPENDENT SET

- Clique and independent set are equivalent to vertex cover (and each other). Homework!
- Therefore, clique and independent set are also NP-complete. Definition: Given a graph G, its complement graph G^c is a graph with the same set of vertices such that for any pair of vertices (u,v) there is an edge in G iff there is not an edge in G^c.

INTEGER LINEAR PROGRAMMING (ILP)

- An instance of the ILP problem consists of a set $\{v_1, v_2, \ldots, v_n\}$ of integer variables, a set of linear inequalities (with integer coefficients) over these variables, a function $f(v_1, v_2, \ldots, v_n)$ to maximize, and an integer B.
- The decision problem is, "Does there exist an assignment of integers to the variables such that all the inequalities are true and $f(v_1, v_2, \ldots, v_n) \geq B$?"

ILP – Example

variables: v_1, v_2

inequalities:

$$v_1 \geq 1$$
$$v_2 \geq 0$$
$$v_1 + v_2 \leq 3$$

function: $f(v_1, v_2) = 2v_2$

bound: $B = 3$

A solution to this instance is

$$v_1 = 1$$
$$v_2 = 2$$

ILP – Example

variables: v_1, v_2

inequalities:

$$v_1 \geq 1$$
$$v_2 \geq 0$$
$$v_1 + v_2 \leq 3$$

function: $f(v_1, v_2) = 2v_2$

bound: $B = 5$

This instance has no solution

ILP is NP-Complete

Theorem

Integer linear programming is NP-hard.

Proof.

- We will show that SAT reduces to it.
- Start with some instance of SAT that contains variables $\{v_1, v_2, \ldots, v_n\}$ and clauses.
- We will create an ILP instance as follows:
  - There will be two variables for each variable v_i, named variables V_i and $\bar{V}_i$. They will correspond to the literals v_i and $\bar{v}_i$.
  - Notice that in the integer programming instance they are separate variables, not one variable and its "negation" and they are integers, not Booleans.

ILP is NP-Complete

Cont.

There are three classes of inequalities:

I $0 \leq V_i \leq 1$
$0 \leq \bar{V}_i \leq 1$ This models the Boolean-ness of the variables and amounts to four inequalities – each V_i and $\bar{V}_i$ is either 0 (False) or 1 (True).

II $1 \leq V_i + \bar{V}_i \leq 1$ This is just $V_i + \bar{V}_i = 1$, but we needed to express it in terms of inequalities to make this an ILP instance. This equation says exactly one of V_i and $\bar{V}_i$ is true.

ILP is NP-Complete

Cont.

III Inequalities that encode the clauses in the SAT problem. For each clause $z_1^{(k)} \vee z_2^{(k)} \vee \cdots \vee z_{n_k}^{(k)}$ we create an inequality $W_1 + W_2 + \cdots + W_{n_k} \geq 1$ where

$$W_j = \begin{cases} V_p & \text{if } z_j^{(k)} = v_p \\ \bar{V}_p & \text{if } z_j^{(k)} = \bar{v}_p \end{cases}$$

- For instance, for the clause $v_1 \vee \bar{v}_{19} \vee \bar{v}_7 \vee \cdots \vee v_6$ we introduce the inequality $V_1 + \bar{V}_{19} + \bar{V}_7 + \cdots + V_6 \geq 1$.
- Clearly this inequality is satisfied iff at least one of the variables in it is 1, which corresponds to at least one of the literals in the clause being True.

ILP is NP-Complete

Cont.

- We don't need the function f and the bound B
- We can simply set $f(V_1, \bar{V}_1, \ldots, \bar{V}_n) = 0$ and $B = 0$.
- We see immediately that
  - The integer linear programming instance that we have constructed from the SAT instance has a solution iff the SAT instance is satisfiable.
  - The construction of the integer programming instance from the SAT instance is a polynomial-time algorithm.

And that concludes the proof. □

ILP is NP-Complete – Some Remarks

- We have not really shown that ILP is NP-complete. We *have* shown that it is NP-hard, but it is not quite obvious that it is in NP, because the integers in the solution to the instance (not in the instance itself!) might be too large to even be written out in polynomial time.
- We showed that SAT could be reduced to a more restricted problem: 0-1 ILP, in which each variable can take either 0 or 1, and each coefficient is also 0 or 1.
- This problem is certainly in NP, and so it is NP-complete (so the difficulty does not necessarily lie in big numbers).

SUBSET-SUM

- Also called INTEGER PARTITION.
- An instance of the problem is a set S of integers and a "target" integer t.
- The question is, "Is there a subset of S whose sum is t?"
- For instance, if

$$S = \{1, 4, 16, 64, 256, 1040, 1041, 1093, 1284, 1344\}$$

and $t = 3754$, then the answer is "yes", because

$$1 + 16 + 64 + 256 + 1040 + 1093 + 1284 = t$$

SUBSET-SUM

Theorem

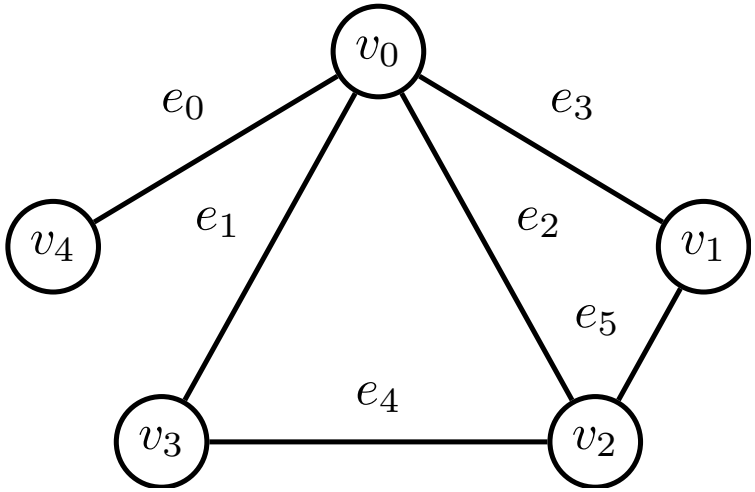
SUBSET SUM is NP-complete.

Proof.

1. **SUBSET SUM is in NP.** This is obvious: checking that a *particular* subset adds up to t can certainly be done in linear time.
2. **SUBSET SUM is NP-hard.** We will prove this by reducing VERTEX COVER to SUBSET SUM:VC $\leq_P$ SUBSET SUM
   - We need to start with a graph in which we are trying to find a vertex cover of size N, and turn this VC instance into an instance of SUBSET SUM.
   - We take our graph, and we construct its *incidence matrix*:

SUBSET-SUM – Example

	e_0	e_1	e_2	e_3	e_4	e_5
v_0	1	1	1	1	0	0
v_1	0	0	0	1	0	1
v_2	0	0	1	0	1	1
v_3	0	1	0	0	1	0
v_4	1	0	0	0	0	0



SUBSET-SUM

- There are exactly two 1's in each column. That will be a key point.
- We will call this matrix b, and in the example above $b[2,1] = 0$.
- Each row can be thought of as a base-4 representation of an integer, only with the low-order digits on the *left* so that the row for v_2 corresponds to $4^2 + 4^4 + 4^5$
- For example – the row corresponding to the vertex v_i corresponds to the number $\sum_{j=0}^{|E|-1} b[i,j] \cdot 4^j$

SUBSET-SUM

We extend the matrix by adding a new row for each edge, and we will put a 1 in the column that corresponds to that edge:

	e_0	e_1	e_2	e_3	e_4	e_5
v_0	1	1	1	1	0	0
v_1	0	0	0	1	0	1
v_2	0	0	1	0	1	1
v_3	0	1	0	0	1	0
v_4	1	0	0	0	0	0
e_0	1	0	0	0	0	0
e_1	0	1	0	0	0	0
e_2	0	0	1	0	0	0
e_3	0	0	0	1	0	0
e_4	0	0	0	0	1	0
e_5	0	0	0	0	0	1

SUBSET-SUM

- Each column has three 1's in it: two from vertex rows and one from an edge row. The top rows of this matrix are just the original matrix b.
- For each vertex row we construct the number (which is just the number above, but with a high-order term added). $V_i = 4^{|E|} + \sum_{j=0}^{|E|-1} b[i,j]4^j$. We will call these the "vertex numbers".
- For each edge row we construct the number (this time *without* the high-order term added) $E_k = 4^k$. We will call these the "edge numbers".
- The subset sum instance is this: the numbers in our set S are just the numbers V_i and E_k we just constructed.
- This is obviously a polynomial construction.

SUBSET-SUM

- The target number is $t = N \cdot 4^{|E|} + 2 \cdot \sum_{j=0}^{|E|-1} 4^j$
- We will show that the graph we started with has a vertex cover of size N iff the subset sum problem we have just constructed is solvable.
- Notice the following facts:
  - If we add up any subset of numbers in S (even if we add up *all* the numbers in S), there will be no "carries" from one column to the next in the base-4 addition. The reason is that each column can contain at most three 1's, and it would take four 1's to produce a carry.
  - It follows from this that for a sum of numbers in S to equal t it must contain exactly N vertex numbers, since that is how many vertex numbers we will need to get the high term $N \cdot 4^{|E|}$ in t.

SUBSET-SUM is NP-Complete – Direction 1

Lemma

If the VC instance is solvable, then the derived SUBSET SUM instance is solvable.

Proof.

- If we have a vertex cover of the graph with N vertices, and if we take the sum of the corresponding vertex numbers, we have a high-order term of $N \cdot 4^{|E|}$.
- As for the other terms, since each edge in the graph is "covered", we will have at least a contribution of $1 \cdot 4^j$ for each edge e_j.
- If we only have $1 \cdot 4^j$ and not $2 \cdot 4^j$, then we can add the edge number E_j.
- This way we have a solution to the SUBSET SUM problem.

SUBSET-SUM is NP-Complete – Direction 2

Lemma

If the derived SUBSET SUM instance is solvable, then the VC instance is solvable.

Proof.

- Take the vertex numbers in the solution of the SUBSET SUM instance, there are exactly N of them.
- The rest of the numbers in the solution (edge numbers) can only contribute at most a 1 in each remaining column.
- The vertex numbers have to contribute either 1 or 2 in each column, so each edge is covered by either 1 or 2 vertices in the subset of vertices that corresponds to the vertex numbers in the solution to the derived SUBSET SUM instance.
- Those vertices constitute a vertex cover of size N.

HAMILTONIAN-CYCLE

- A *Hamiltonian cycle* in a graph G is a simple cycle that visits each vertex.
- There are two variants of this problem, depending on whether the graph is directed or undirected.
- Both problems are NP-complete.
- In what follows we deviate a bit from the proof in the text and prove each one separately.

HAMILTONIAN-CYCLE

Theorem

DIRECTED HAMILTONIAN CYCLE is NP-complete.

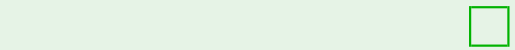
Proof.

1. **DIRECTED HAMILTONIAN CYCLE is in NP.** Clearly it's polynomial-time checkable.
2. **DIRECTED HAMILTONIAN CYCLE is NP-hard.**
3. We will prove this by reducing 3-SAT to it: 3-SAT $\leq_P$ DIRECTED HAMILTONIAN CYCLE Start with a 3-SAT instance that has n variables $\{v_1, v_2, \ldots, v_n\}$ and k clauses $\{c_1, c_2, \ldots, c_k\}$, where each clause is of the form $z_1 \vee z_2 \vee z_3$, each z_j being a literal.
4. We will show produce from it a graph $G = (V, E)$ such that
   - The construction is polynomial in $n + k$.
   - G has a Hamiltonian cycle iff the 3-SAT instance is satisfiable.

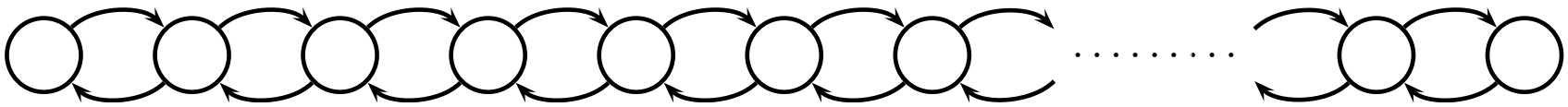
□

Cont.

- We assume that each clause in our 3-SAT instance involves 3 distinct variables.
- If a clause is of the form $v_1 \vee \bar{v}_1 \vee v_2$ then it is automatically true, and we can just eliminate it from the instance.
- A clause such as $v_1 \vee v_1 \vee v_2$ is really just $v_1 \vee v_2$, and we have already seen how to turn this in to a pair of clauses (with a new variable), each clause containing 3 literals.
- So let us assume our 3-SAT instance contains literals corresponding to 3 different variables.
- For each variable v_i we create a set of vertices in G and hook them together in a "doubly linked list".

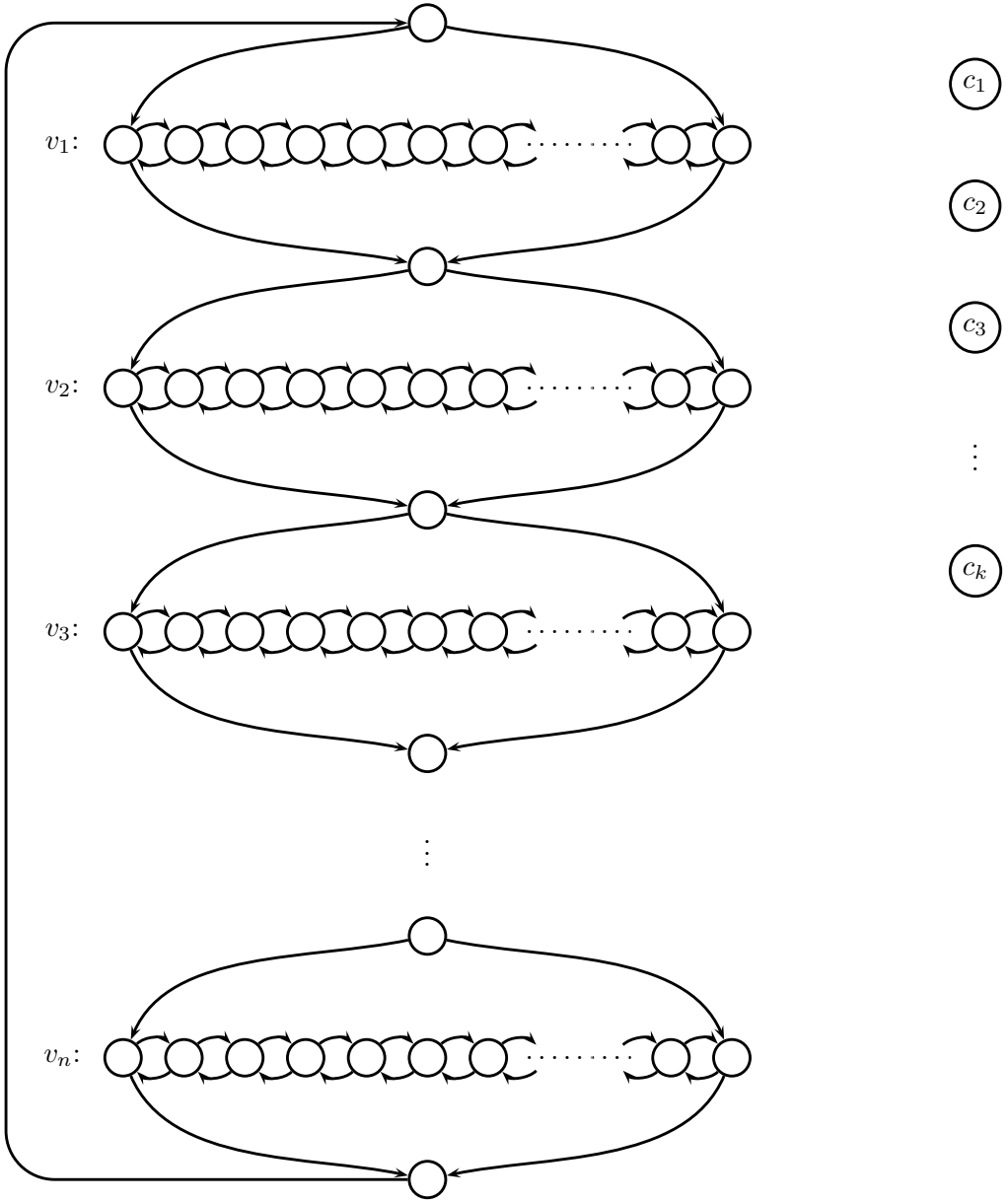


HAMILTONIAN-CYCLE – Example



- We have $3(k+1)$ nodes here.
- We take the list corresponding to each node and connect it to some auxiliary nodes to form a oval-like structure, and we then hook up these oval structures vertically.
- We also add k other nodes, each one corresponding to one of the clauses in the 3-SAT expression.

HAMILTONIAN-CYCLE – Example



HAMILTONIAN-CYCLE

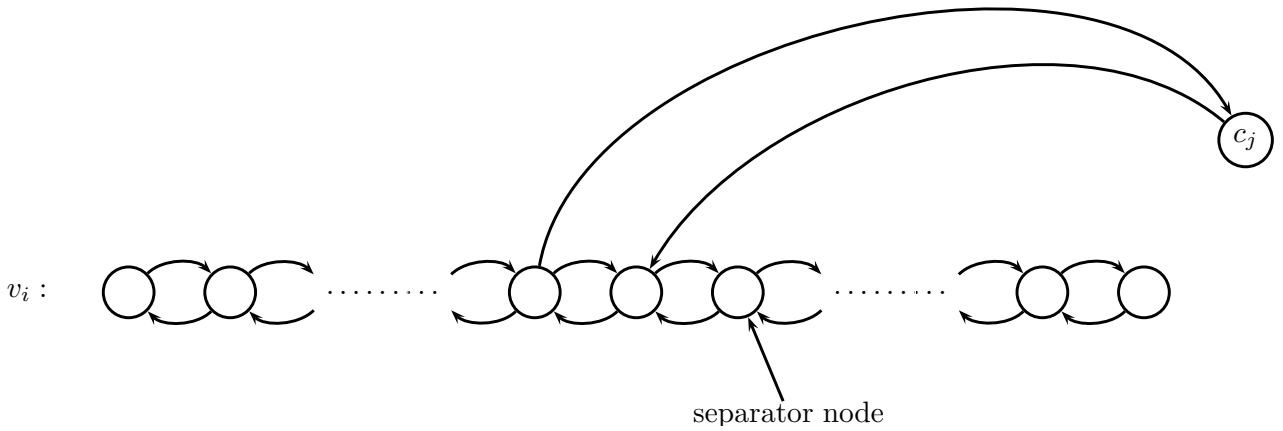
- To form a Hamiltonian cycle, each row will either be traversed left-to-right or right-to-left, the choice for each row being independent of the choice for every other row.
- A traversal of row i left-to-right encodes the value True for the variable v_i, a traversal of row i right-to-left encodes the value False for v_i.
- There are 2^n possible Hamiltonian cycles of the graph and these different cycles correspond exactly to the 2^n different ways of assigning either True or False to the n different variables $\{v_1, v_2, \ldots, v_n\}$.
- Next we hook up the nodes $\{c_1, c_2, \ldots, c_k\}$ to the rest of the graph in such a way that the clause information is encoded.

HAMILTONIAN-CYCLE

- We divide each row (corresponding to each variable v_i as follows:
- An initial node (i.e., the left-most one).
- A "separator node".
- k sets of 3 nodes each. The j^{th} set corresponds to the clause c_j. Actually, the first two nodes in the set correspond to c_j and the third node in each set is another "separator node". We will call the first two nodes in each set the "c_j group in row i".
- A final node (i.e., the right-most one).

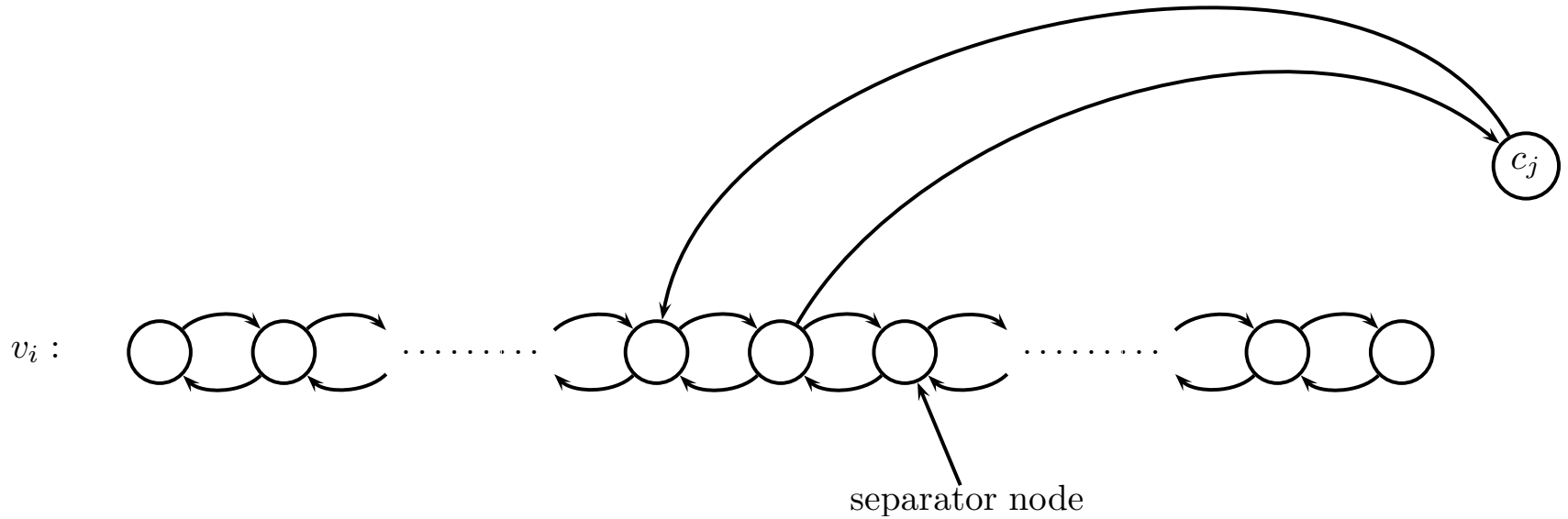
HAMILTONIAN-CYCLE

- Each clause c_j contains three literals ($c_j = z_1^{(j)} \vee z_2^{(j)} \vee z_3^{(j)}$).
- For each of those literals, we add two edges involving c_j.
- A literal z corresponds to v_i or $\bar{v}_i$.
- The two edges we insert will connect c_j with the two nodes in the c_j group in row i, as follows:
- If $z = v_i$, we insert an edge from the left node in the c_j group $\rightarrow c_j$ and an edge from $c_j \rightarrow$ the right node in the c_j group
- If v_i is True, then (since row i is traversed left-to-right), we can use these two edges to make a side trip to c_j, including c_j in the cycle.



HAMILTONIAN-CYCLE

- If $z = \bar{v}_i$, we do things "the other way": we insert an edge from the right node in the c_j group $\rightarrow c_j$ and an edge from $c_j \rightarrow$ the left node in the c_j group
- The reason for doing this is that if v_i has the value False (so $\bar{v}_i$ is True), then (since row i will be traversed right-to-left) we can use these two edges to make a side trip to c_j, thus including c_j in the cycle.



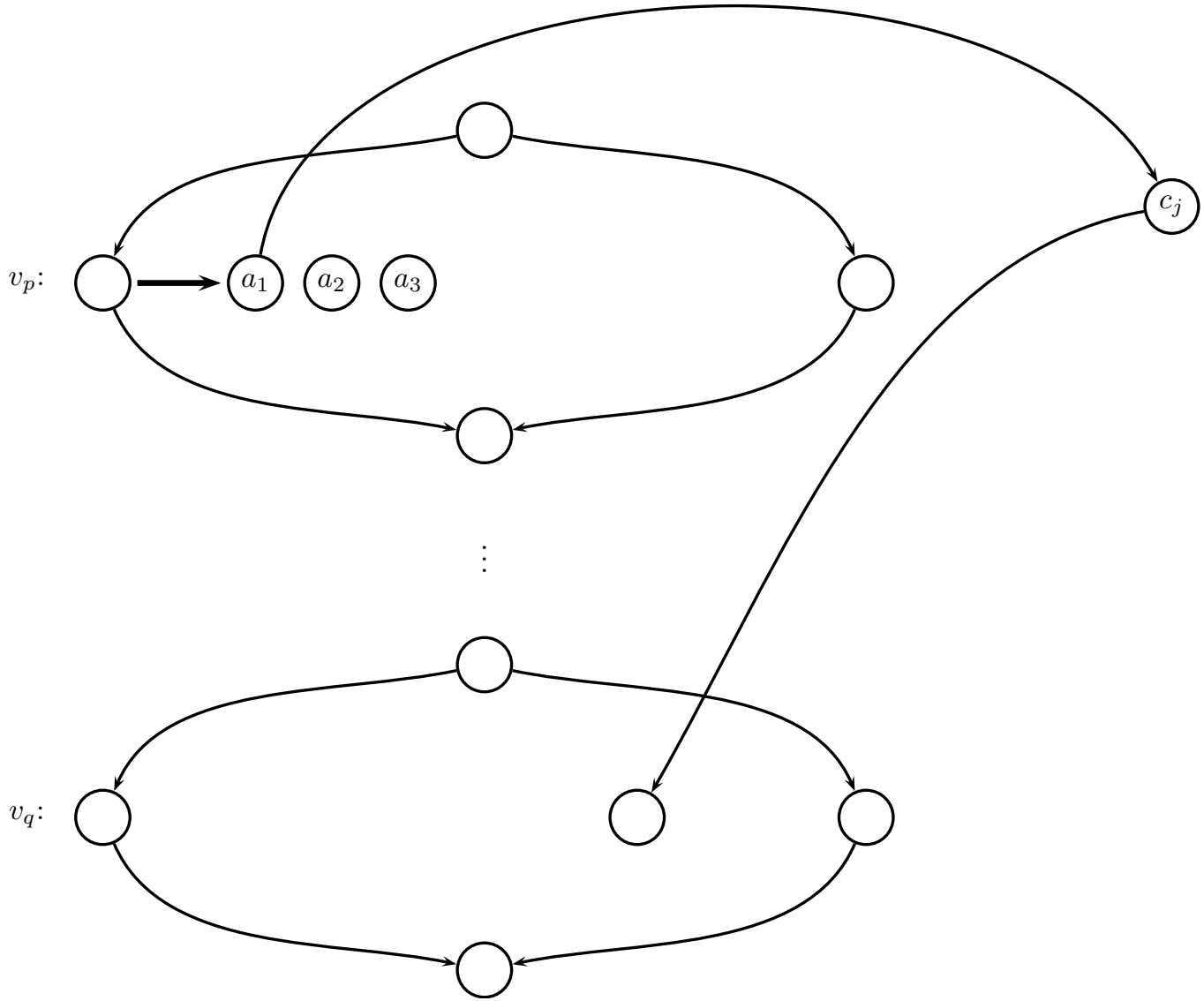
HAMILTONIAN-CYCLE

- Each clause c_j has three pairs of edges that are inserted for it.
- These edges will never "step on each other": the edges from clause c_j will only attach to "column j" of the main part of the graph, so the edges from two different clauses will never coincide.
- We assumed that no clause contains the same variable twice, so each of the three pairs of edges introduced for each clause goes to a different row, so they can't coincide either.
- If the original expression is satisfiable, then G has a Hamiltonian cycle: traverse each edge in the appropriate way (i.e., left-to-right if v_i is True and right-to-left if v_i is False).
- For each clause c_j at least one of the literals in c_j will be True.
- For the row corresponding to variable v_i in that literal, a trip can be made to c_j since the two edges to and from c_j were set up that way. Thus each c_j can be included in the cycle, and so we have a Hamiltonian cycle for G.

HAMILTONIAN-CYCLE – Opposite Direction

- Suppose G has a Hamiltonian cycle, we must show that the original 3-SAT instance is satisfiable.
- This is immediately true if we know that each c_j is reached by a path to and from the same row.
- Then the variable in that row corresponds to a True literal in c_j, and so each c_j is satisfied.
- All we have to prove is that if G has a Hamiltonian cycle, then each c_j is reached by a path to and from the same row.
- Suppose it were not and we had something like the situation in the following figure.

HAMILTONIAN-CYCLE – This Can't Happen!



HAMILTONIAN-CYCLE – This Can't Happen!

- Suppose that a_1 is a node in some row that is reached from the left, and that the edge from a_1 to c_j is not followed by an edge (in the path) from c_j to a_2.
- We know that either a_2 or a_3 must be a separator nodes. Consider these two possibilities separately:

Case I: a_2 is a separator node. It must be attached to the nodes on either side of it, but a_2 cannot be attached to a_1 by an edge in the Hamiltonian cycle, since a_1 already has two Hamiltonian cycle edges attached to it, so it's impossible

Case II: a_3 is a separator node. a_1 and a_2 must both correspond to the same clause (c_j). a_2 must be attached either to c_j or to a_1 by an edge in the Hamiltonian cycle, but neither one is possible, since both those nodes already have two Hamiltonian cycle edges attached to them.

HAMILTONIAN-CYCLE – This Can't Happen!

- If a_1 is approached from the left the above is impossible. If it were approached from the right, then a similar argument (directions switched) would show the same.
- Therefore we showed that a Hamiltonian cycle of G corresponds to an assignment of truth values to the variables $\{v_1, v_2, \ldots, v_n\}$ that satisfies the original 3-SAT instance.
- Finally, we note that the construction of G was polynomial, and that concludes the proof.

UNDIRECTED HAMILTONIAN-CYCLE

Theorem

UNDIRECTED HAMILTONIAN CYCLE is NP-complete.

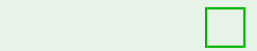
Proof.

1. **UNDIRECTED HAMILTONIAN CYCLE is in NP.** Clearly it's polynomial-time checkable.
2. **UNDIRECTED HAMILTONIAN CYCLE is NP-hard.** We will prove this showing: DIRECTED HAMILTONIAN CYCLE $\leq_P$ UNDIRECTED HAMILTONIAN CYCLE.
3. We start with an instance of DIRECTED HAMILTONIAN CYCLE – a directed graph G – and we will construct an undirected graph H which has a Hamiltonian cycle iff G does.

□

Cont.

- Each vertex v in G corresponds to three vertices v^{in}, v^{mid}, and v^{out} in H.
- They are connected by two (undirected) edges: one between v^{in} and v^{mid}, and the other between v^{mid} and v^{out}.
- The rest of the edges in H mirror the edges in G: If (u, v) is a (directed) edge in G, we create an edge in H from u^{out} to v^{in}.
- Clearly, this is a polynomial time construction.



UNDIRECTED HAMILTONIAN-CYCLE

Lemma

If G has a Hamiltonian cycle, then H does.

Proof.

If $u_1 \to u_2 \to \cdots \to u_n \to u_1$ is a (directed) Hamiltonian cycle in G, then

$$u_1^{\text{in}} \leftrightarrow u_1^{\text{mid}} \leftrightarrow u_1^{\text{out}} \leftrightarrow$$
$$u_2^{\text{in}} \leftrightarrow u_2^{\text{mid}} \leftrightarrow u_2^{\text{out}} \cdots \leftrightarrow$$
$$u_n^{\text{in}} \leftrightarrow u_n^{\text{mid}} \leftrightarrow u_n^{\text{out}} \leftrightarrow u_1^{\text{in}}$$

is an undirected Hamiltonian cycle in H. □

UNDIRECTED HAMILTONIAN-CYCLE

Lemma

If H has a Hamiltonian cycle, then G does.

Proof.

- Since each "mid" node is connected by one edge to an "in" node and one edge to an "out" node, the only way that each "mid" node can be in a cycle is for all three ("in", "mid", "out") nodes to be in that cycle.
- Therefore, a Hamiltonian cycle of H must be of the form

$$u_1^{\text{in}} \leftrightarrow u_1^{\text{mid}} \leftrightarrow u_1^{\text{out}} \leftrightarrow u_2^{\text{in}} \leftrightarrow u_2^{\text{mid}} \leftrightarrow u_2^{\text{out}}$$
$$\cdots \leftrightarrow u_n^{\text{in}} \leftrightarrow u_n^{\text{mid}} \leftrightarrow u_n^{\text{out}} \leftrightarrow u_1^{\text{in}}$$

- But this corresponds exactly to the Hamiltonian cycle $u_1 \to u_2 \to \cdots \to u_n \to u_1$ in G.

The Traveling Salesperson Problem (TSP)

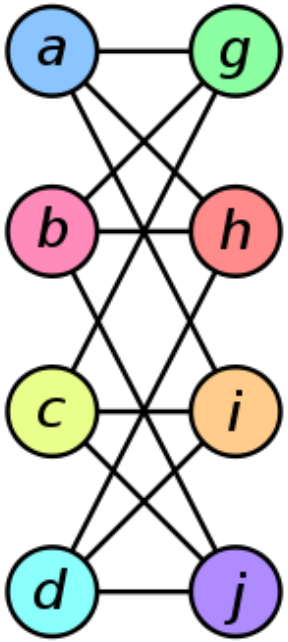
- We have a set of cities, represented as vertices in a graph.
- A salesperson needs to visit each city as cheaply as possible.
- Assume that the cost is the total distance of the trip.
- Between each two vertices there is an edge with an associated weight and we want to find the shortest path that visits each node.
- To make things simple, we may also assume that the path should be a cycle.
- The associated decision problem is "Does this graph have a Hamiltonian cycle of weight $\leq W$?"

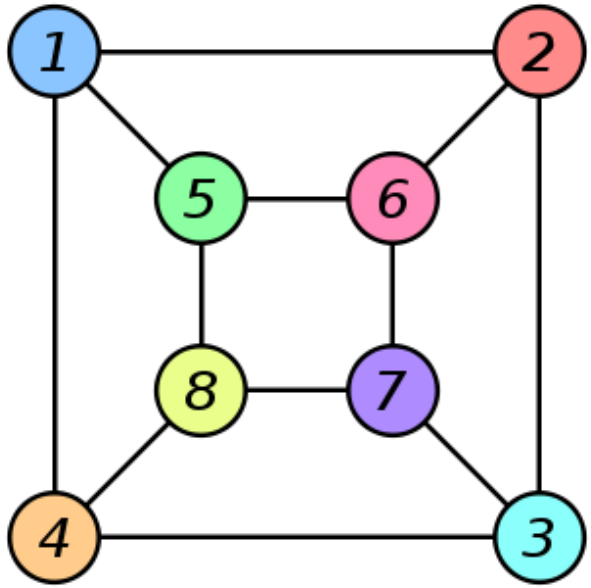
The Traveling Salesperson Problem (TSP)

- We can reduce UNDIRECTED HAMILTONIAN CYCLE to this problem as follows:
- Let G be any undirected graph. This is the of UNDIRECTED HAMILTONIAN CYCLE.
- We will construct a graph H with edge weights that will be an instance of TSP as follows:
  - The vertices of H are just the vertices of G.
  - Every two vertices of H are connected by an edge. (H is a complete graph.)
  - The weight of an edge in H is 0 if that edge is also an edge in G, and is 1 otherwise.
- Then the question "Does H have a Hamiltonian cycle of weight ≤ 0?" has a positive answer iff G has a Hamiltonian cycle. Thus theTSP problem is NP-complete.

Subgraph isomorphism

An isomorphism is a bijection between the vertices of two graphs $f : V(G_1) \rightarrow V(G_2)$ such that any two vertices u and v of G_1 are adjacent in G_1 iff $f(u)$ and $f(v)$ are adjacent in G_2.





From Wikipedia

Subgraph isomorphism

- Given two graphs G and H, is H isomorphic to some subgraph of G?
- Again, the problem is clearly in NP.
- It's NP-hard because we can reduce CLIQUE to it.
- To ask the question "Does G have a clique of size k?" is to ask the question "Does G have a subgraph that is isomorphic to the complete graph on k vertices?" So CLIQUE $\leq_P$ SUBGRAPH ISOMORPHISM, and so SUBGRAPH ISOMORPHISM is NP-complete.

Comments

- Sometimes a small change in the problem definition changes the complexity significantly.
- Graph isomorphism – in NP but not known whether the problem is NP-complete.
- Polynomial time solutions exist for:
  - Eulerian path/cycle – A cycle that goes through every edge once (vertices can be repeated).
  - DNF-SAT.
  - Linear programming (variables are not restricted to integers).
  - etc...