

## Lecture 20: NP-completeness

Harvard SEAS - Fall 2024

2024-11-12

## 1 Announcements

- Anurag OH today moved to Zoom, 11:30-12:30.
- Salil OH Thu 11:15-12:00 in SEC 3.327.
- PS7 due tomorrow.
- PS8 out by Thursday, due 2024-11-20.
- Next SRE on Thursday.

Recommended Reading:

- MacCormick §14, 17
- Sipser §7.5

## 2 Recap

- Def of  $\text{NP}_{\text{search}}$ .
- Def of  $\text{NP}_{\text{search}}$ -completeness.
- Cook-Levin Theorem and significance.
- Transitivity of  $\leq_p$ .

In today's lecture, we will prove that several different problems are  $\text{NP}_{\text{search}}$ -complete.

## 3 3-SAT

Once we have one  $\text{NP}_{\text{search}}$ -complete problem, we can get others via reductions from it. Consider the computational problem 3-SAT, which is obtained when we restrict the number of literals in each clause of SAT.

<b>Input</b>	: A CNF formula $\varphi$ on $n$ variables $z_0, \dots, z_{n-1}$ in which each clause has width at most 3 (i.e. contains at most 3 literals)
<b>Output</b>	: An $\alpha \in \{0, 1\}^n$ such that $\varphi(\alpha) = 1$ (if one exists)

### Computational Problem 3-SAT

**Theorem 3.1.** *3-SAT is  $\text{NP}_{\text{search}}$ -complete.*

*Proof.* The proof follows in two steps.

1. 3SAT is in  $\text{NP}_{\text{search}}$ : Our verifier can check if an assignment  $\alpha$  satisfies the 3CNF formula (the same verifier as for SAT).
2. 3SAT is  $\text{NP}_{\text{search}}$ -hard: Since every problem in  $\text{NP}_{\text{search}}$  reduces to SAT (Cook–Levin Theorem), all we need to show is  $\text{SAT} \leq_p 3\text{SAT}$  (since reductions compose).

The reduction algorithm from SAT to 3SAT has the following components (Figure 1). First, we give an algorithm R which takes a SAT instance  $\varphi$  to a 3SAT instance  $\varphi'$ .

$$\text{SAT instance } \varphi \xrightarrow{\text{polytime R}} \text{3SAT instance } \varphi'$$

Then we feed the instance  $\varphi'$  to our 3SAT oracle and obtain a satisfying assignment  $\alpha'$  to  $\varphi'$  or  $\perp$  if none exists. If we get  $\perp$  from the oracle, we return  $\perp$ , else we transform  $\alpha'$  into a satisfying assignment to  $\varphi$  using another algorithm S.

$$\text{SAT assignment } \alpha \xleftarrow{\text{polytime S}} \text{3SAT assignment } \alpha'$$

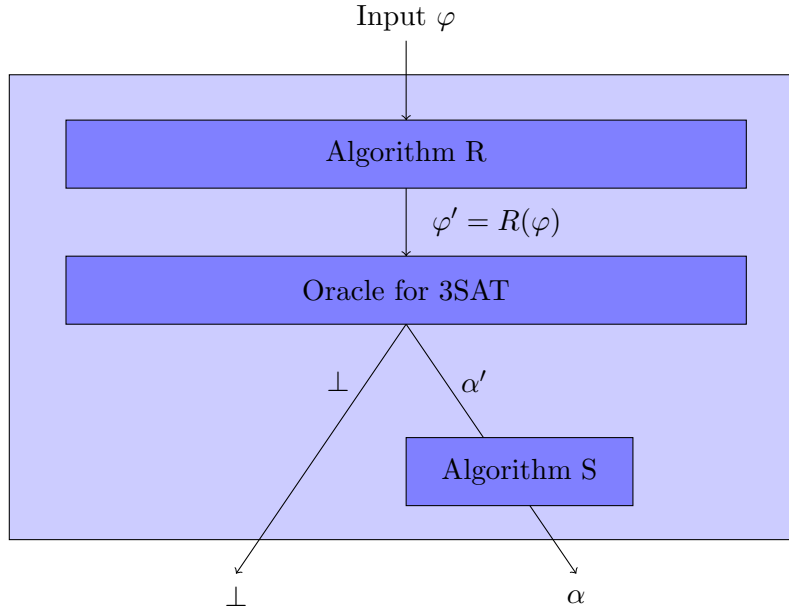


Figure 1: Reduction algorithm from SAT to 3SAT.

**Algorithm R:** The intuition behind this algorithm is that when we have a clause  $(\ell_0 \vee \ell_1 \vee \dots \vee \ell_{k-1})$  in the SAT instance  $\phi$  (with large width  $k > 3$ ), we want to break it into multiple clauses of width 3. But simply breaking it up doesn't preserve information about  $\varphi$  being satisfiable. Instead, we introduce the clauses  $(y \vee \ell_0 \vee \ell_1)$  and  $(\neg y \vee \ell_2 \dots \ell_{k-1})$ , where  $y$  is a new boolean variable. We repeat this process until all the clauses are of width 3. Note that this way of reducing the width of

the clauses is the reverse of the resolution procedure. Formally, the algorithm is as follows:

```

1  $R(\varphi)$  :
   Input      : A CNF formula  $\varphi$ 
   Output    : A CNF formula  $\varphi'$  with each clause of width 3
2  $\varphi' = \varphi$ 
3  $i = 0$ 
4 while  $\varphi'$  has a clause  $C = (\ell_0 \vee \dots \vee \ell_{k-1})$  of width  $k > 3$  do
5   | Remove  $C$ 
6   | Add clauses  $(y_i \vee \ell_0 \vee \ell_1)$  and  $(\neg y_i \vee \ell_2 \dots \ell_{k-1})$ , where  $y_i$  is a new variable
7   |  $i = i + 1$ 
8 return  $\varphi'$ 

```

Note that  $\varphi'$  is **not** an equivalent formula to  $\varphi$ . While  $\varphi$  is on variables  $z_0, \dots, z_{n-1}$ , the formula  $\varphi'$  is on variables  $z_0, \dots, z_{n-1}, y_0, \dots, y_{t-1}$ , where  $t$  is the number of iterations of the while loop.

**Algorithm S:** Given an assignment  $\alpha' = (\alpha_0, \dots, \alpha_{n-1}, \beta_0, \dots, \beta_{t-1})$  to  $\varphi'$ , the algorithm simply takes the first  $n$  bits, i.e.  $\alpha = (\alpha_0, \dots, \alpha_{n-1})$

Next we consider the runtime and correctness of the overall reduction algorithm.

**Runtime of the reduction algorithm:** We first consider the runtime of the algorithm R: At each iteration of the while loop, we take a clause of width  $k$  and produce clauses of width 3 and  $k-1$ . Thus, the total width of clauses (with width greater than 3) goes down by 1 at each step, so the procedure terminates. In fact, the number of iterations is bounded by  $\sum_{C \in \varphi, |C| > 3} |C| = O(nm)$ , where  $|C|$  is the width of the clause  $C$ ,  $n$  is the number of variables in  $\varphi$  and  $m$  is the number of clauses.

Then, we consider the runtime of the algorithm  $S$ , which is simply  $O(n)$ . Overall, the runtime of the reduction algorithm is  $O(nm)$ .

**Proof of correctness:** We will show that if  $\varphi$  is satisfiable, then the reduction algorithm produces a satisfying assignment and if  $\varphi$  is unsatisfiable, the reduction algorithm will output  $\perp$ . This is based on the following two claims. (In lecture, we combined the claims into one to streamline the proof.)

**Claim 3.2.** *If  $\varphi$  is satisfiable then  $\varphi' = R(\varphi)$  is satisfiable.*

*Proof of claim.* Assume that  $\varphi$  is satisfiable. Let  $\varphi = \varphi_0, \varphi_1, \dots, \varphi_t = R(\varphi)$  be the formula as it evolves through the  $t$  loop iterations. We will prove by induction on  $i$  that  $\varphi_i$  is satisfiable for  $i = 0, \dots, t$ , constructed through the  $t$  loop iterations.

**Base case ( $i = 0$ ):**  $\varphi_0 = \varphi$ , which is satisfiable by hypothesis.

**Induction step:** By the induction hypothesis, we can assume that  $\varphi_{i-1}$  is satisfiable, and now we need to show that  $\varphi_i$  is satisfiable:

Suppose  $\gamma = (\alpha_0, \dots, \alpha_{n-1}, \beta_0, \dots, \beta_{i-1})$  is a satisfying assignment to  $\varphi_{i-1}$ , and we obtain  $\varphi_i$  from it by breaking up clause  $C = (\ell_0 \vee \dots \vee \ell_{k-1})$ . Then since  $\gamma$  satisfies  $C$ , it satisfies at least one of  $(\ell_0 \vee \ell_1)$  and  $(\ell_2 \vee \dots \vee \ell_{k-1})$ . If it satisfies the first, we can set  $\beta_i = 0$  and obtain an assignment  $\gamma' = (\alpha_0, \dots, \alpha_{n-1}, \beta_0, \dots, \beta_{i-1}, \beta_i)$  that satisfies both  $(y_i \vee \ell_0 \vee \ell_1)$  and  $(\neg y_i \vee \ell_2 \dots \ell_{k-1})$  and hence  $\varphi_i$ . In the second case, we can set  $\beta_i = 1$ . Thus, we've maintained that a satisfying assignment exists.  $\square$

**Claim 3.3.** *If  $\alpha'$  satisfies  $R(\varphi)$ , then  $\alpha = S(\alpha')$  also satisfies  $\varphi$ .*

*Proof of claim.* We prove by “backwards induction” that  $\alpha'$  satisfies  $\varphi_i$  for  $i = t, \dots, 0$ . We can then drop the extra  $t$  variables that don’t appear in  $\varphi$  without changing the satisfiability. (We call this “backwards induction” since our base cases is  $i = t$ .)

The base case ( $i = t$ ) follows because  $\alpha'$  satisfies  $R(\varphi) = \varphi_t$  by assumption.

For the induction step: Suppose by induction that  $\alpha'$  satisfies  $\varphi_i$ , and now we want to show that it also satisfies  $\varphi_{i-1}$ .  $\varphi_i$  was constructed from  $\varphi_{i-1}$  by breaking up some clause  $C = (\ell_0 \vee \dots \vee \ell_{k-1})$  into  $(y_i \vee \ell_0 \vee \ell_1) \wedge (\neg y_i \vee \ell_2 \vee \dots \vee \ell_{k-1})$ . By assumption  $\alpha'$  satisfies the two new clauses. By the validity of resolution,  $\alpha'$  also satisfies their resolution,  $(y_i \vee \ell_0 \vee \ell_1) \diamond (\neg y_i \vee \ell_2 \vee \dots \vee \ell_{k-1})$ , which equals  $C$ . Thus  $\alpha'$  satisfies  $\varphi_{i-1}$ .  $\square$

To finish the correctness proof, suppose  $\varphi$  is satisfiable. Then from Claim 3.2,  $\varphi'$  is also satisfiable. The 3SAT oracle returns a satisfying assignment  $\alpha'$ , which is turned into a satisfying assignment for  $\varphi$  via the algorithm  $S$  (Claim 3.3). If  $\varphi$  is unsatisfiable, then by Claim 3.3,  $\varphi'$  is also unsatisfiable. In this case, the 3SAT oracle returns  $\perp$  - as a result the reduction algorithm also returns  $\perp$ .

This completes the proof that 3-SAT is  $\text{NP}_{\text{search}}$ -complete.  $\square$

## 4 Mapping Reductions

The usual strategy for proving that a problem  $\Gamma$  in  $\text{NP}_{\text{search}}$  is also  $\text{NP}_{\text{search}}$ -hard (and hence  $\text{NP}_{\text{search}}$ -complete) follows a structure similar to the proof of Theorem 3.1.

1. Pick a known  $\text{NP}_{\text{search}}$ -complete problem  $\Pi$  to try to reduce to  $\Gamma$ . Typically, we might try to pick a problem  $\Pi$  that seems as similar as possible to  $\Gamma$ , or which has been used to prove that problems similar to  $\Gamma$  are  $\text{NP}_{\text{search}}$ -complete. Otherwise 3-SAT is often a good fallback option.
2. Come up with an algorithm  $R$  mapping instances  $x$  of  $\Pi$  to instances  $R(x)$  of  $\Gamma$ . If  $\Pi$  is 3-SAT, this will often involve designing “variable gadgets” that force solutions to  $R(x)$  to encode true/false assignments to variables of  $x$  and “clause gadgets” that force these assignments to satisfy each of the clauses of  $x$ . We will see an example in the next lecture.
3. Show that  $R$  runs in polynomial time.
4. Show that if  $x$  has a valid answer, then so does  $R(x)$ . That is, we can transform valid answers to  $x$  to valid answers to  $R(x)$ .
5. Conversely, show that if  $R(x)$  has an answer, then so does  $x$ . Moreover, we can transform valid answers to  $R(x)$  into valid answers to  $x$  through a *polynomial time* algorithm  $S$ . This transformation needs to be efficient (in contrast to Item 4 because it has to be carried out by our reduction).

Reductions with the structure outlined above are called *mapping reductions*, and they are what are typically used throughout the theory of  $\text{NP}$ -completeness. A formal definition follows (but we won’t expect you to use this formalism, you can stick with the general definition of polynomial-time reductions):

**Definition 4.1.** Let  $\Pi = (\mathcal{I}, \mathcal{O}, f)$  and  $\Gamma = (\mathcal{J}, \mathcal{P}, g)$  be search problems. A *polynomial-time mapping reduction* from  $\Pi$  to  $\Gamma$  consists of two polynomial-time algorithms  $R$  and  $S$  such that for every  $x \in \mathcal{I}$ :

1.  $R(x) \in \mathcal{J}$ .
2. If  $f(x) \neq \emptyset$ , then  $g(R(x)) \neq \emptyset$ . That is, if an input  $x$  has some correct answer in  $\mathcal{O}$ , then the transformed  $R(x)$  has corresponding a correct answer in  $\mathcal{P}$ .
3.  $\forall y \in g(R(x))$ , we have  $S(x, y) \in f(x)$ .

Note that the above outline only proves  $\text{NP}_{\text{search}}$ -hardness; a proof that  $\Gamma$  is  $\text{NP}_{\text{search}}$ -complete should also check that it's in  $\text{NP}_{\text{search}}$ .

## 5 Independent Set

Next we turn to IndependentSet. (Formally the IndependentSet-ThresholdSearch version.)

**Theorem 5.1.** *IndependentSet is  $\text{NP}_{\text{search}}$ -complete.*

*Proof.* We'll do this proof less formally than we did the proof of  $\text{NP}_{\text{search}}$ -completeness of 3SAT.

1. In  $\text{NP}_{\text{search}}$ : The verifier checks if the set  $S \subseteq V(G)$  (claimed to be a solution, an independent set of size at least  $k$  in  $G$ ) is actually (a) of size at least  $k$  so  $|S| \geq k$  and (b) independent so  $\forall e \in E(G), |e \cap S| \leq 1$ . Both of these checks can be completed in polynomial time.
2.  $\text{NP}_{\text{search}}$ -hard: We will show  $3\text{SAT} \leq_p \text{IndependentSet}$ .

We've previously encoded many other problems in SAT, but here we're going in the other direction and showing a graph problem can encode SAT.

Our reduction  $R(\varphi)$  takes in a CNF and produces a graph  $G$  and a size  $k$ . We'll use as an example the formula

$$\varphi(z_0, z_1, z_2, z_3) = (\neg z_0 \vee \neg z_1 \vee z_2) \wedge (z_0 \vee \neg z_2 \vee z_3) \wedge (z_1 \vee z_2 \vee \neg z_3).$$

Our graph  $G$  consists of:

- Variable gadgets: these are pairs of vertices connected by an edge, labelled by ' $z_0$ ' and ' $\neg z_0$ '. An independent set can contain only one of these two vertices, capturing the fact that only one of the two literals can be made true. We will set  $k$  so that any independent set of size  $k$  must also contain at least one vertex from each variable gadget, thus representing a full boolean assignment to the  $n$  variables.
- Clause gadgets: For each clause  $C = (\ell_0 \vee \ell_1 \vee \ell_2)$ , we add a "clause gadget," which consists of a triangle of three vertices, labelled by the literals  $\ell_0$ ,  $\ell_1$ , and  $\ell_2$ . Note that an independent set can contain at most one of these three vertices. We will set  $k$  so that any independent of size  $k$  *must* contain at least one vertex from each clause gadget, (hopefully) corresponding to a literal that is satisfied in the assignment given by the choices made in the variable gadgets.

- **Conflict edges:** We add *conflict edges* between clause-gadget-vertices and variable-gadget-vertices that have opposite labels. This ensures that the literal we choose to satisfy each clause is consistent with the selection among the variable gadgets.

We pick  $k = m + n$ , which implies that every independent set of size  $k$  must contain exactly one vertex from each variable gadget and one vertex from each clause gadget. An algorithm  $R$  can create this graph (and  $k$ ) in polynomial time given  $\varphi$ . The graph for the formula  $\varphi$  is depicted in Figure 2.

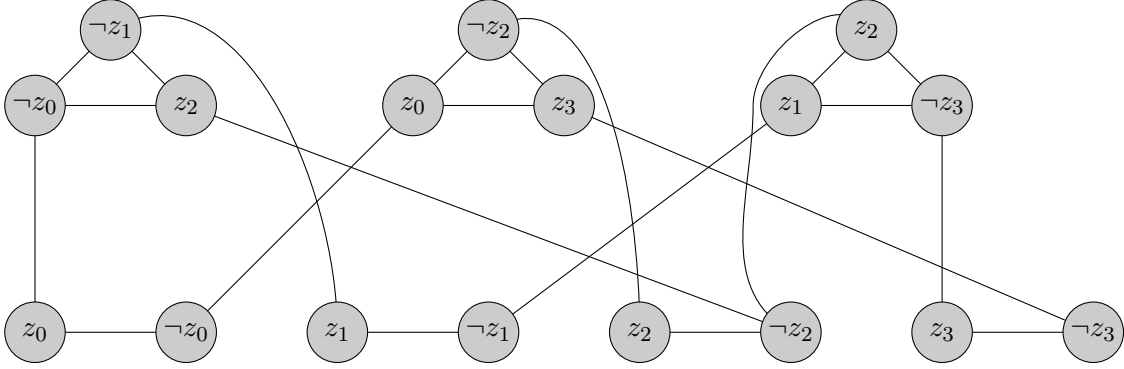


Figure 2: Variable and clause gadgets

Note that (analogously to the SAT to 3SAT case) the correspondence between 3SAT and ISET does not exactly preserve the set of satisfying solutions (they aren't even the same problem) but we will see how we can go from solutions to one to solutions to the other.

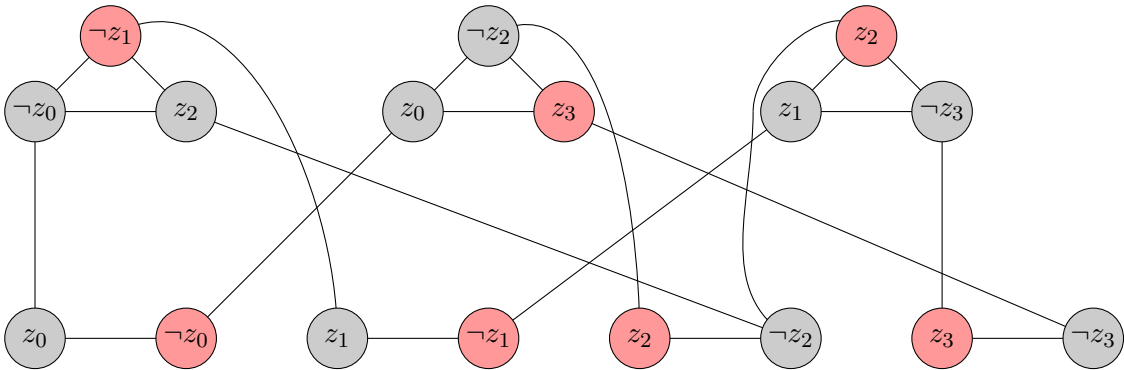


Figure 3: Independent set of size  $4 + 3 = 7$  when the satisfying assignment is  $(0, 0, 1, 1)$ .

Remember that an IndependentSet problem consists of 1) a graph  $G$  and 2) a minimum size  $k$  of an independent set. How can we choose the size  $k$  for this reduction? Intuitively, we might think about assigning True to the variables whose corresponding vertices are selected as part of the independent set. Then, we'll choose  $k = n + m$ , where  $n$  is the number of variables and  $m$  is the number of clauses in the original Boolean formula. The hope is that the clause edges will force

exactly  $n$  of the variable gadgets to be set to True, and at least one vertex in each of the  $m$  clause gadget is also True. We'll now prove that this claim is true.

**Claim 5.2.**  *$G$  has an independent set of size  $k = n + m$  if and only if  $\varphi$  is satisfiable. Moreover, we can map independent sets of size  $k$  to satisfying assignments of  $\varphi$  in polynomial time.*

*Proof of claim.*

Given a satisfying assignment  $\alpha$  to  $\varphi$ , we can pick one vertex in each variable and clause gadget and have them all be independent. For each variable gadget, pick the vertex corresponding to the assignment in  $\alpha$ . For each clause gadget, pick a single vertex that is connected to a literal not satisfied by  $\alpha$ . (If there are multiple vertices that can be chosen, we can pick one arbitrarily. We can't pick more than one since an independent set can only have one vertex from any triangle.) See Figure 3 for an example.

A similar proof in the other direction shows that given an independent set of size  $n + m = k$  in  $G$ , we can recover a satisfying assignment to  $\varphi$ . Specifically, if we have an independent set of size  $n + m$  in  $G$ , it must contain exactly one vertex from each variable gadget and exactly one vertex from each clause gadget (else it would be of size smaller than  $n + m$ ). Then we take our assignment  $\alpha$  according to the vertices chosen from the variable gadget. The vertices chosen from the clause gadgets certify that at least one literal is satisfied in each clause.

□

This completes the proof that IndependentSet is  $\text{NP}_{\text{search}}$ -complete.

□

## 6 Longest Path

Finally, we consider the problem from SRE5:

<b>Input</b>	: A digraph $G = (V, E)$ , two vertices $s, t \in V$ , and a path-length $k \in \mathbb{N}$
<b>Output</b>	: A path from $s$ to $t$ in $G$ of length $k$ , if one exists
<b>Computational Problem</b> LongPath	

In the Sipser text (optional reading), it is proven that LongPath is  $\text{NP}_{\text{search}}$ -complete, even in the special case where  $k = n - 1$ , i.e. the path visits all vertices in the graph:

<b>Input</b>	: A digraph $G = (V, E)$ , two vertices $s, t \in V$
<b>Output</b>	: A path from $s$ to $t$ in $G$ that includes every vertex in $G$ , if one exists
<b>Computational Problem</b> HamiltonianPath	

**Theorem 6.1.** *HamiltonianPath is  $\text{NP}_{\text{search}}$ -complete.*

In fact, the HamiltonianCycle problem (where  $s = t$ ) is also  $\text{NP}_{\text{search}}$ -complete, even in undirected graphs. (The Sipser text has a reduction from HamiltonianPath to UndirectedHamiltonianPath; the reduction from HamiltonianPath to HamiltonianCycle is a good exercise.) HamiltonianCycle is a special case of the Travelling Salesperson Problem (TSP), where a salesperson wants to visit a set of cities and return to their start city in the minimum amount of travel time. If the possible trips between cities are given by a digraph  $G$  and every possible trip takes the same amount time, then the shortest possible route is of length  $n$  and is given by a HamiltonianCycle.

In contrast, *EulerianWalk*, where we seek a walk from  $s$  to  $t$  that uses every *edge* in  $G$  exactly once is known to be in  $\text{P}_{\text{search}}$ .

## 7 Optional read: Three-Dimensional Matching

A month ago, we saw algorithms to find maximum matchings in a bipartite graph: that is, given a graph whose vertices are in two sets  $V_0$  and  $V_1$ , and a set  $E$  of edges each of which contains exactly one vertex from  $V_0$  and one vertex from  $V_1$ , we can find (in polynomial time) a maximum-size matching, a subset of  $E$  in which no edges overlap (no edges share an endpoint).

Just as changing 2SAT to 3SAT turns a polynomial-time solvable problem  $\text{NP}_{\text{search}}$ -complete, we'll see now that changing "two" to "three" makes that efficiently solvable problem  $\text{NP}_{\text{search}}$ -complete. (A similar example which we won't prove in CS 120: changing 2-coloring to 3-coloring also turns a polynomial-time solvable problem  $\text{NP}_{\text{search}}$ -complete). We note here that a "hypergraph" is like a graph, but its (hyper)edges can consist of any number of vertices (not necessarily exactly two).

<b>Input</b>	: A hypergraph $G = (V, E)$ where $V$ is partitioned into three sets $V_0$ , $V_1$ , and $V_2$ , and each hyperedge contains exactly three vertices, one from each of $V_0$ , $V_1$ , and $V_2$ .
<b>Output</b>	: A set of hyperedges which are disjoint and cover all the vertices, if one exists.

**Computational Problem** ThreeDimensionalMatching

**Theorem 7.1.** *ThreeDimensionalMatching (AKA 3DM) is  $\text{NP}_{\text{search}}$ -complete.*

*Proof.* There are two requirements for a problem to be  $\text{NP}_{\text{search}}$ -complete: (1) it's in  $\text{NP}_{\text{search}}$  (2) other problems in  $\text{NP}_{\text{search}}$  reduce to it in polynomial time.

**$\text{NP}_{\text{search}}$  membership of 3DM.** To show that ThreeDimensionalMatching is in  $\text{NP}$ , we need to show that there exists a polynomial-time algorithm that, given a potential solution  $y$  (that is, a set of triples of vertices denoting a hyperedge), checks whether it's a set of hyperedges which is disjoint and covers all the vertices.

To do so, make an array  $A$  with one entry per vertex of  $G$ , initialized to all 0s. Read through the triples of vertices in  $y$ . For each one, check whether it's a hyperedge (if not, return  $\perp$ ), and if so, add one to  $A$ 's entries for each of those three vertices. After reading all of  $y$ , check that every entry of  $A$  is 1, and return  $\perp$  if not. If so, every vertex was covered exactly once, so we can return True.

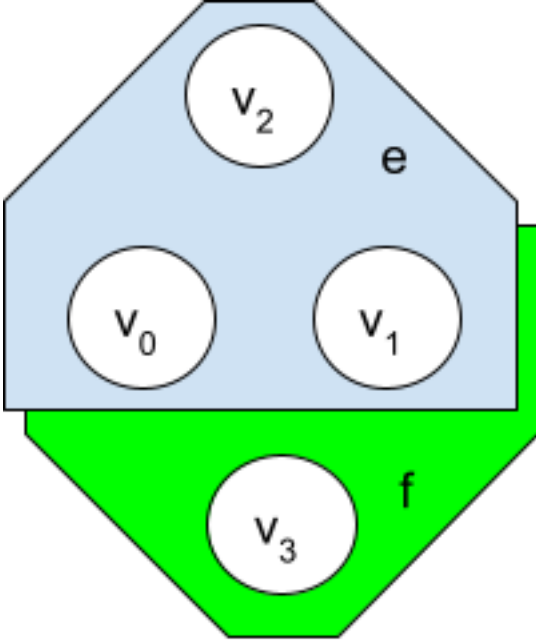
**$\text{NP}_{\text{search}}$ -hardness of 3DM: reduction from 3SAT** To show that ThreeDimensionalMatching is  $\text{NP}_{\text{search}}$ -hard, we need to show that every problem in  $\text{NP}_{\text{search}}$  reduces to it in polynomial time. We'll again use the fact that every problem in  $\text{NP}_{\text{search}}$  reduces to 3SAT in polynomial time, so if we can reduce from 3SAT to ThreeDimensionalMatching, transitivity of polynomial-time reductions means that every problem in  $\text{NP}_{\text{search}}$  reduces to ThreeDimensionalMatching in polynomial time. We'll use the reductions framework mentioned earlier: we'll take an input to 3SAT (that is, a 3CNF formula), make it an input to 3DM (that is, a graph), call a 3DM oracle, and use the resulting matching to make a satisfying assignment to 3SAT.

The full reduction is complicated (see the end of these notes), so we'll work our way up to it, building 3DM gadgets that simulate gradually more of 3SAT.

**Simple variable gadget:** In 3SAT, each variable can be set to true or false. To simulate that in 3DM, we need some ability to make a binary choice. The simplest way we could make a choice

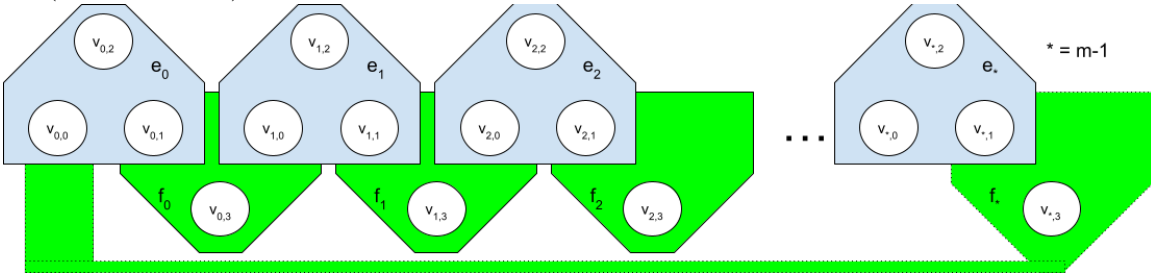


is to make two hyperedges  $e = \{v_0, v_1, v_2\}$  and  $f = \{v_0, v_1, v_3\}$  which overlap in vertices  $v_0$  and  $v_1$ . Also, we promise in 3DM not to include  $v_0$  or  $v_1$  in any other hyperedges.



Why is this a binary choice? The 3DM solution can't pick both  $e$  and  $f$  (because they overlap, and 3DM requires the set of chosen hyperedges to be disjoint), but to cover  $v_0$  and  $v_1$ , we must pick at least one of  $e$  and  $f$ ; therefore, picking  $e$  or  $f$  can simulate picking true or false for a variable. If we pick hyperedge  $e$  (representing picking true for the variable), then vertex  $v_3$  is not yet matched; if we pick hyperedge  $f$  (representing false), then vertex  $v_2$  is not yet matched. These four vertices,  $v_k$  for  $k \in [4]$ , constitute a simple variable gadget.

**Expanded variable gadget** In the simple variable gadget, the rest of the graph we're constructing can only interface with the gadget at two vertices,  $v_2$  and  $v_3$ . It may be useful to have more than two vertices to interface with, so we create a bigger variable gadget: make a chain of  $2m$  hyperedges ( $m$  is the number of clauses), each of which overlaps with the next in one shared vertex (like  $v_0$  and  $v_1$ ).

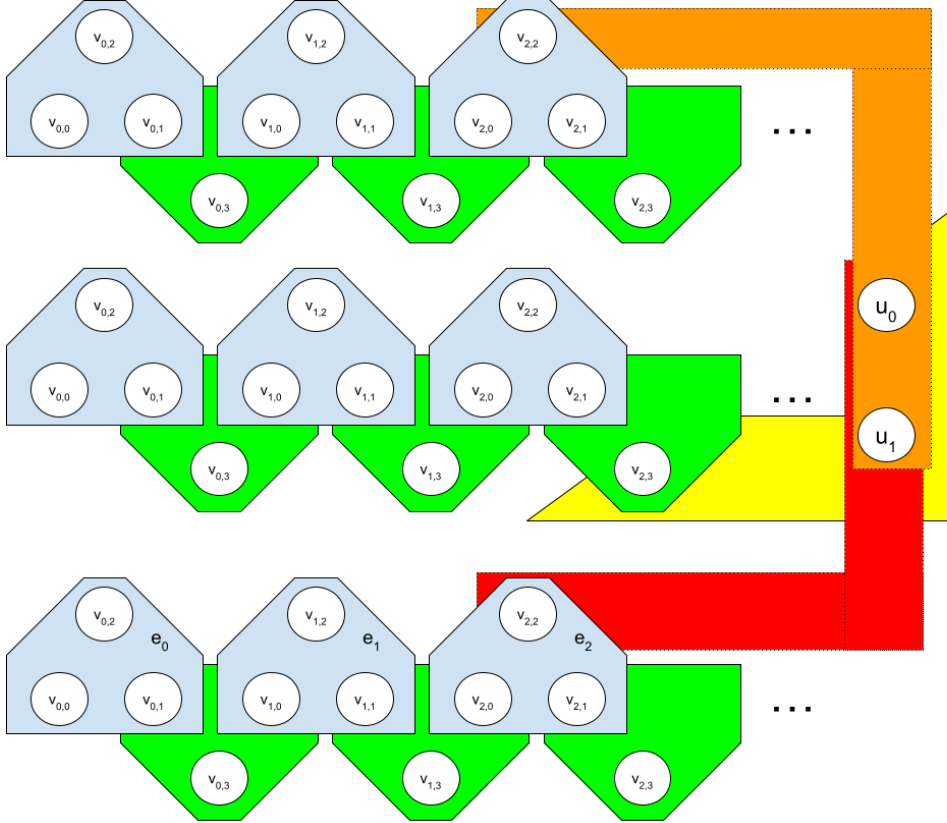


As before, there are two choices a three-dimensional matching can make about hyperedges: if we pick the first hyperedge  $e_0$ , we must not pick the second hyperedge  $f_0$  (because it overlaps in  $v_{0,1}$ ), so we must pick the third hyperedge  $e_1$  (because  $v_{1,0}$  needs to be covered), and so on. If we don't pick the first hyperedge, we must pick the second, so we must not pick the third, and so on. We'll say that picking all the odd ( $e$ ) hyperedges corresponds to setting a variable true, and picking all the even ( $f$ ) hyperedges corresponds to setting a variable false. As before, we would like to call

the shared variables  $v_0$  and  $v_1$ , the vertices that are left uncovered by the “true” choice  $v_3$ , and the vertices that are left uncovered by the “false” choice  $v_2$ . However, since there are now  $m$  copies of each of these, they need to be distinguished from each other, so we call them  $v_{j,k}$ :  $j \in [m]$  for the  $j$ th copy of each of the four vertices,  $k \in [4]$  as in the simple variable gadget.

**Assignment gadget:** An assignment to 3SAT consists of an assignment of all  $n$  variables, each of which can be independently set either true or false. To simulate this, we make  $n$  separate copies of the expanded variable gadget above and label them with numbers  $i \in [n]$ ; that is, we now have vertices labeled  $v_{i,j,k}$  for  $i \in [n]$ ,  $j \in [m]$ , and  $k \in [4]$ .

**Clause gadget** A clause like  $C = (\neg z_{120} \vee z_{121} \vee \neg z_{124})$  can be satisfied in at most<sup>1</sup> three ways: in that case, by having  $z_{120}$  set false, by having  $z_{121}$  set true, or by having  $z_{124}$  set false. To simulate this, we make two vertices  $u_0$  and  $u_1$  that are each only in (at most) three hyperedges: one hyperedge with  $u_0$ ,  $u_1$ , and one of the variables that’s uncovered if  $z_{120}$  is set false (that is,  $v_{120,j,2}$  for some value of  $j$ ); another hyperedge with  $u_0$ ,  $u_1$ , and one of the variables that’s uncovered if  $z_{121}$  is set true (that is,  $v_{121,j,3}$  for some value of  $j$ ), and another edge  $(u_0, u_1, v_{124,j,2})$  for  $z_{124}$  false. In the figure below, the first pictured extended variable gadget is for  $z_{120}$ , the second is for  $z_{121}$ , and the third is for  $z_{124}$ .



Therefore, to cover  $u_0$  and  $u_1$ , we must use one of those three hyperedges, so at least one of  $v_{120,j,2}$ ,  $v_{121,j,3}$ , and  $v_{124,j,2}$  must have been left uncovered, so either  $x_{120}$  must have been set false,  $x_{121}$  must have been set true, or  $x_{124}$  must have been set false.

We want one clause gadget for each clause  $C_j$ . Each clause gadget should use its own vertices

<sup>1</sup>A 3SAT clause may have fewer than three literals.

$u_0$  and  $u_1$ , which we distinguish by additionally labeling them with  $j$ .

To make sure that vertex  $v_{120,j,2}$  is available if we need it (and isn't used by some other clause gadget), we'll have clause  $j$  use only the  $j$ th copies of vertices from the assignment gadget. So, in full generality, for each positive literal  $x_i \in C_j$ , we add the hyperedge  $(u_{j,0}, u_{j,1}, v_{i,j,3})$ , and for each negative literal  $\neg x_i \in C_j$ , we add the hyperedge  $(u_{j,0}, u_{j,1}, v_{i,j,2})$ .

**Cleanup gadget** The gadgets above guarantee that if the formula is not satisfiable, there's no matching that covers all the vertices. In the other direction, we have some cleanup to do: if the formula is satisfiable, our described use of the gadgets covers most of the vertices: the clause vertices  $u_{j,k}$  and some of the variable-gadget vertices  $v_{i,j,0}$  and  $v_{i,j,1}$ . However, vertices  $v_{i,j,2}$  and  $v_{i,j,3}$  may not be covered yet, even if we have a valid solution to 3SAT: maybe variable  $i$  isn't used in clause  $j$ , or clause  $j$  had more than one true literals so a perfect matching doesn't use some vertices corresponding to some literals that satisfied it. To use up any extra vertices  $v_{i,j,2}$  and  $v_{i,j,3}$ , we make three copies of the whole construction so far, which we label with  $\ell = 0$ ,  $\ell = 1$ , or  $\ell = 2$ , and add hyperedges connecting the three copies of  $v_{i,j,2}$ . We also add hyperedges connecting the three copies of  $v_{i,j,3}$ . If the 3SAT formula is satisfiable, we can pick the same hyperedges on each level, and cover any unpicked vertices  $v_{i,j,2}$  and  $v_{i,j,3}$  on all three levels with these cleanup hyperedges. On the other hand, we proved above that the only ways to cover the vertices  $v_{i,j,0}$ ,  $v_{i,j,1}$  and  $u_{j,k}$  on even one level correspond to satisfying assignments to the 3SAT formula, so if there's a perfect 3DM, we can look only at level  $\ell = 0$  to get a solution to the original 3SAT problem.

So, all together, the reduction is:

- Given a 3SAT problem  $\varphi$  with  $n$  variables  $z_0, z_1, \dots, z_{n-1}$  that are used in clauses<sup>2</sup> and  $m$  clauses  $C_0, \dots, C_{m-1}$ , we'll make a graph  $R(\varphi)$  with  $12nm + 6m$  vertices.
  - Name  $12nm$  of the vertices  $v_{i,j,k,\ell}$ , where  $i \in [n]$ ,  $j \in [m]$ ,  $k \in [4]$ , and  $\ell \in [3]$ .
  - Name the other  $6m$  vertices  $u_{j,k,\ell}$  where  $j \in [m]$ ,  $k \in [2]$ , and  $\ell \in [3]$ .
- We include the following hyperedges:
  - For each  $i \in [n]$ ,  $j \in [m]$ , and  $\ell \in [3]$ , add the hyperedge  $(v_{i,j,0,\ell}, v_{i,j,1,\ell}, v_{i,j,2,\ell})$ . (Call these “True hyperedges”.)
  - For each  $i \in [n]$ ,  $j \in [m]$ , and  $\ell \in [3]$ , add the hyperedge  $(v_{i,j+1,0,\ell}, v_{i,j+1,1,\ell}, v_{i,j+1,2,\ell})$ . (Call these “False hyperedges”.) Consider  $j \bmod m$ : that is,  $j + 1$  should wrap back around to 0.
  - For each  $i \in [n]$ ,  $j \in [m]$ , and  $k \in \{2, 3\}$ , add the hyperedge  $(v_{i,j,k,0}, v_{i,j,k,1}, v_{i,j,k,2})$ . (Call these “cleanup hyperedges”.)
  - For each  $j \in [m]$  and  $\ell \in [3]$  and positive literal  $x_i \in C_j$ , add the hyperedge  $(u_{j,0,\ell}, u_{j,1,\ell}, v_{i,j,3,\ell})$ . (Call these “positive clause-satisfying edges”.)
  - For each  $j \in [m]$  and  $\ell \in [3]$  and negative literal  $\neg x_i \in C_j$ , add the hyperedge  $(u_{j,0,\ell}, u_{j,1,\ell}, v_{i,j,2,\ell})$ . (Call these “negative clause-satisfying hyperedges”.)
- After we generate the graph  $R(\varphi)$  as above, call the 3DM oracle on it. If it returns  $\perp$ , return  $\perp$ . If it returns a 3DM, assign each variable  $z_i$  to be true if the hyperedge  $(v_{i,0,0,0}, v_{i,0,1,0}, v_{i,0,2,0})$  was picked, and false otherwise.

---

<sup>2</sup>If any variables are not used in any clauses, ignore them: they can be set arbitrarily.

**3-partition** Note that the definition of 3DM requires the graph's vertices to be divisible into three sets, where each hyperedge contains one vertex from each set. This is true for the graph we've constructed by putting the vertices  $v_{i,j,k,\ell}$  or  $u_{j,k,\ell}$  where  $\ell + \min(k, 2) \in \{0, 3\}$  into one set  $V_0$ , the vertices where  $\ell + \min(k, 3) \in \{1, 4\}$  into another set  $V_1$ , and the vertices where  $\ell + \min(k, 3) = 2$  into another set  $V_2$ . You can check that every hyperedge defined in the reduction has one of each.

**NP<sub>search</sub>-hardness of 3DM: runtime of the reduction** The reduction from 3SAT to 3DM is an algorithm that takes as input a 3SAT formula  $\varphi$  and produces a hypergraph  $R(\varphi)$  as above, then does some faster steps (calls the oracle and reads out an answer). To produce the graph, the algorithm does nothing more complicated than run some loops (over  $i \in [n]$ ,  $j \in [m]$ , etc., or over clauses in the input), adding one thing to the graph in each instance of the loop, so the runtime of the algorithm is just proportional to the size of the graph it outputs.

That graph has size polynomial in the size of the input: the size of the input formula is  $\Theta(m)$ , and the size of the produced graph is  $O(nm)$ . Since we threw out variables not in any clauses,  $n < m$  so  $O(nm) = O(m^2)$ , so the runtime is  $O(m^2)$ .

**NP<sub>search</sub>-hardness of 3DM: proof of correctness of the reduction** As we built up the reduction gadget by gadget, we proved properties of each gadget which, combined, constitute a proof of correctness. However, we'll write a proof of correctness here separately for two reasons:

1. To make clear the distinction between a reduction (just the algorithm described in bullet points above) and a proof of correctness (statements like “a perfect matching must/can pick certain edges”).
2. To see how all the things we proved about the gadgets fit together into a full proof of correctness.

To prove the reduction is correct, we need to prove it's correct on all inputs: that is, for every input 3SAT formula  $\varphi$  that's unsatisfiable, the output is  $\perp$ , and for every input  $\varphi$  that's satisfiable, the output of the reduction is a satisfying assignment. When a proof of correctness is divided into those two pieces, they're called “soundness” and “completeness”, respectively. In the reductions framework from the start of lecture, Item 4 is a proof of completeness and Item 5 is a proof of soundness.

**NP<sub>search</sub>-hardness of 3DM: proof of soundness of the reduction** To prove soundness, we need to prove that if  $\varphi$  is unsatisfiable, the reduction returns  $\perp$ . It's easier (here and often) to prove the equivalent contrapositive statement: if the reduction returns an assignment, then it satisfies the 3SAT formula. If the reduction returns an assignment, it did so in the last bullet point, and the 3DM oracle found a matching; in particular, each clause vertex  $u_{j,0,0}$  is covered. Only (at most) three hyperedges contain that vertex, each of which also uses a vertex like  $v_{i,j,3,0}$  or  $v_{i,j,2,0}$  from some expanded variable gadget corresponding to a literal in the clause. We proved when we described the expanded variable gadgets that the variable gadget leaves  $v_{i,j,3,0}$  or  $v_{i,j,2,0}$ , respectively, uncovered only if it picked the “True hyperedges” or “False hyperedges”, respectively, for the variable gadget representing  $x_i$ . The last step of the reduction sets  $x_i$  to be true or false, respectively, in those cases, so clause  $C_j$  is satisfied by that value of  $x_i$ . That's true for each clause, so the whole formula is satisfied.

**NP<sub>search</sub>-hardness of 3DM: proof of completeness of the reduction** To prove completeness, we need to prove that if  $\varphi$  has some satisfying assignment  $\alpha$ , the reduction returns a satisfying assignment (not necessarily  $\alpha$ ). This is mostly a matter of saying that our gadgets can be used as intended:

1. For each variable  $x_i$  that's true in  $\alpha$ , pick all the “true” hyperedges in  $G$ 's variable gadgets for  $x_i$ .
2. For each variable that's false in  $\alpha$ , pick all the “false” hyperedges in  $G$ 's variable gadgets for  $x_i$ .
3. For each clause  $C_j$ , choose  $\alpha$ 's first true literal in it (one exists because  $\alpha$  is a satisfying assignment), and pick the corresponding hyperedges.
4. Finally, choose whatever cleanup hyperedges are unused.

Together, these show that the 3DM problem has a solution. So the oracle returns a solution (not necessarily the one described above), so the reduction returns an assignment, and the soundness proof above guarantees that the returned assignment is in fact a satisfying assignment.  $\square$