CS120: Intro. to Algorithms and their Limitations Anshu & Hesterberg

Lecture 20: NP and NP-completeness

Harvard SEAS - Fall 2023 *2023-11-14*

1 Announcements

- PS7 due Nov 15
- PS8 out Nov 14
- Next SRE on Thu Nov 16

Recommended Reading:

- MacCormick §14, 17

2 Recap

Definition 2.1. A computational problem $\Pi = (\mathcal{I}, \mathcal{O}, f)$ is in $\mathsf{NP}_{\mathsf{search}}$ if the following conditions hold:

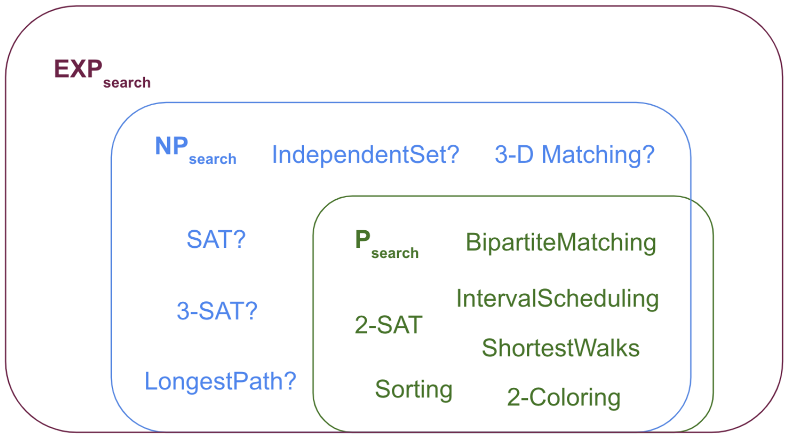
1. All solutions are of polynomial length: There is a polynomial p such that for every $x \in \mathcal{I}$ and every $y \in f(x)$, we have $|y| \le p(|x|)$, where $|z|$ denotes the bitlength of z.
2. All solutions are verifiable in polynomial time: There's a polynomial-time verifier V that, given $x \in \mathcal{I}$ and a potential solution y, decides whether $y \in f(x)$.

Examples are - Satisfiability, Graph Coloring.
Every problem in $\mathsf{NP}_{\mathsf{search}}$ can be solved in exponential time:

Proposition 2.2. $\mathsf{NP}_{\mathsf{search}} \subseteq \mathsf{EXP}_{\mathsf{search}}$.

Proof. Exhaustive search! We can enumerate over all possible solutions and check if any is a valid solution. This has runtime $O(2^{p(n)} \cdot (n + p(n))^c)$ which is bounded by the exponential $O(2^{n^d})$. □

Our diagram of complexity classes looks like this:



Definition 2.3 (NP-completeness, search version)**.** A problem Π is $\mathsf{NP}_{\mathsf{search}}$-complete if:

1. Π is in $\mathsf{NP}_{\mathsf{search}}$
2. Π is $\mathsf{NP}_{\mathsf{search}}$-*hard*: For every computational problem $\Gamma \in \mathsf{NP}_{\mathsf{search}}$, $\Gamma \leq_p \Pi$.

There are natural NP-complete problems. The first one is CNF-Satisfiability:

Theorem 2.4 (Cook–Levin Theorem)**.** *SAT is* $\mathsf{NP}_{\mathsf{search}}$*-complete.*

Once we have one $\mathsf{NP}_{\mathsf{search}}$-complete problem, we can get others via reductions from it.

3 Mapping Reductions

The usual strategy for proving that a problem $\Gamma \in \mathsf{NP}_{\mathsf{search}}$ is also $\mathsf{NP}_{\mathsf{search}}$-hard (and hence $\mathsf{NP}_{\mathsf{search}}$-complete) follows a standard structure:

1. Pick a known $\mathsf{NP}_{\mathsf{search}}$-complete problem Π to try to reduce to Γ. Typically, we might try to pick a problem Π that seems as similar as possible to Γ, or which has been used to prove that problems similar to Γ are $\mathsf{NP}_{\mathsf{search}}$-complete. Otherwise 3-SAT is often a good fallback option.
2. Come up with an algorithm R mapping instances x of Π to instances $R(x)$ of Γ. If Π 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.
3. Show that R runs in polynomial time.
4. Show that if x has a solution, then so does $R(x)$. That is, we can transform valid solutions to x to valid solutions to $R(x)$.
5. Conversely, show that if $R(x)$ has a solution, then so does x. Moreover, we can transform valid solutions to $R(x)$ into valid solutions to x in *polynomial time*. This transformation needs to be efficient (in contrast to Item 4 because it has to be carried out by our reduction).

This is referred to as a *mapping reduction* and used widely in the theory of NP-completeness. Note that the above outline only proves $\mathsf{NP}_{\mathsf{search}}$-hardness; a proof that a problem is $\mathsf{NP}_{\mathsf{search}}$-complete should also check that it's in $\mathsf{NP}_{\mathsf{search}}$.

4 3-SAT is $\mathsf{NP}_{\mathsf{search}}$-complete.

We revisit the theorem from Lec 19:

Theorem 4.1. *3-SAT is* $\mathsf{NP}_{\mathsf{search}}$*-complete.*

Proof.

1. 3SAT is in $\mathsf{NP}_{\mathsf{search}}$.
2. 3SAT is $\mathsf{NP}_{\mathsf{search}}$-hard: Since every problem in NP reduces to SAT, all we need to show is SAT $\leq_p$ 3SAT (since reductions are transitive).

For part (2) we provide a reduction R:

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

```
1 R(φ) :
  Input  : A CNF formula $\varphi$
  Output : A 3-CNF formula $\varphi'$
2 $\varphi' = \varphi$
3 while $\varphi'$ has a clause $C = (\ell_0 \vee \ldots \vee \ell_{k-1})$ of length $k > 3$ do
4     Remove $C$
5     Add clauses $(y \vee \ell_0 \vee \ell_1)$ and $(\neg y \vee \ell_2 \ldots \ell_{k-1})$, where $y$ is a new variable
6 return $\varphi'$
```

R **runs in polynomial time:** At each iteration of the while loop, we take a clause of length k and produce clauses of length 3 and $k-1$. Thus, the total length of a too-large clause goes down by 1 at each step. The number of iterations is bounded by $\sum_{C\in\varphi,|C|>3}|C| \leq nm$ where $|C|$ is the width of the clause.

Claim 4.2. *If φ is satisfiable then $\varphi' = R(\varphi)$ is satisfiable.*

Proof of claim. Assume that φ is satisfiable. Let $\varphi = \varphi_0, \varphi_1, \ldots, \varphi_t = R(\varphi)$ be the formula φ' as it evolves through the t loop iterations. We will prove by induction on i that φ_i is satisfiable for $i = 0, \ldots, t$. 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 φ_{i-1} is satisfiable, and now we need to show that φ_i is satisfiable:

Suppose α_{i-1} is a satisfying assignment to φ_{i-1}, and we obtain φ_i from it by breaking up clause $C = (\ell_0 \vee \ldots \vee \ell_{k-1})$. Then since α_{i-1} satisfies C, it satisfies at least one of $(\ell_0 \vee \ell_1)$ and $(\ell_2 \vee \ldots \vee \ell_{k-1})$. If it satisfies the first, we can set $y = 0$ and obtain an assignment α_i that satisfies both $(y \vee \ell_0 \vee \ell_1)$ and $(\neg y \vee \ell_2 \ldots \ell_{k-1})$ and hence φ_i. In the second case, we can set $y = 1$. Thus, we've maintained that a satisfying assignment exists. □

Finally, we need to show we can transform a satisyfing assignment α' to φ' into a satisfying assignment α to φ. Our transformation simply discards all introduced dummy y variables and takes the assignment to the variables originally in φ.

Claim 4.3. *If α' satisfies $R(\varphi)$, then $\alpha'|_\varphi$ also satisfies φ, where $\alpha'|_\varphi$ is the restriction of the assignment α' to the variables in φ.*

Proof of claim. We prove by "backwards induction" that α' satisfies φ_i for $i = t, \dots, 0$. We can then drop the extra t variables that don't appear in φ without changing the satisfiability. (We call this "backwards induction" since our base cases is $i = t$.)

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

For the induction step: Suppose by induction that α' satisfies φ_i, and now we want to show that it also satisfies φ_{i-1}. φ_i was constructed from φ_{i-1} by breaking up some clause $C = (\ell_0 \vee ... \vee \ell_{k-1})$ into $(y \vee \ell_0 \vee \ell_1) \wedge (\neg y \vee \ell_2 \vee \dots \vee \ell_{k-1})$. By assumption α' satisfies the two new clauses. If $y = 0$ then $\ell_0 \vee \ell_1$ must be 1, which means C is satisfied. Similarly, if $y = 1$ then $\ell_2 \vee \dots \vee \ell_{k-1}$ must be 1, which means C is again satisfied.

□

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

5 Independent Set is $\mathsf{NP}_{\mathsf{search}}$-complete

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

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

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

1. In $\mathsf{NP}_{\mathsf{search}}$: 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. $\mathsf{NP}_{\mathsf{search}}$-hard: We will show 3SAT $\leq_p$ 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 '$\neg z_0$', capturing the fact that only one of the two literals is true.

- Clause gadgets: Clause imposes a constraint between the variables. To enforce this constraint, a clause gadget involves a triangle whose vertices are labelled by the variable number and edges from the vertices to the variable gadgets. For each vertex in a triangle - say labelled by i - we connect it to one of z_i or $\neg z_i$, if the clause has $\neg z_i$ or z_i (respectively). Note that if a clause had both literals, we could discard it by setting it 'true'.

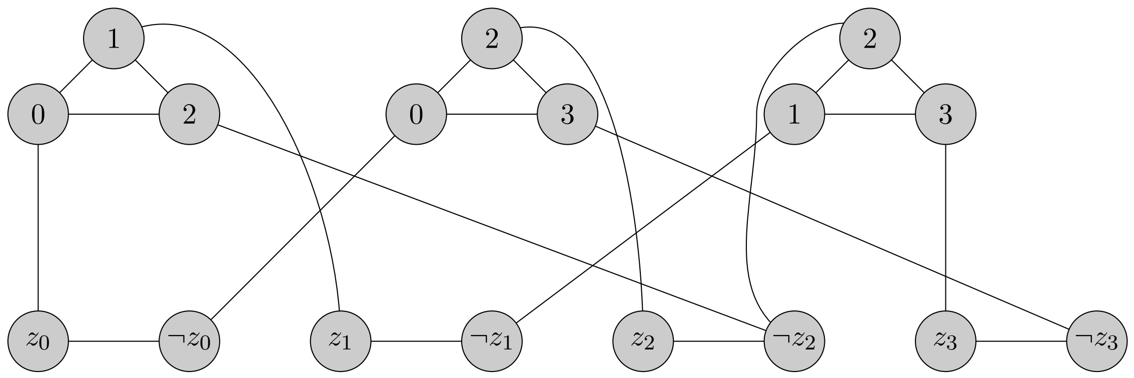


Figure 1: Variable and clause gadgets

We pick $k = m + n$. An algorithm R can create this graph (and k) in polynomial time given φ. The graph for the formula φ is depictedin Figure 1.

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 can go from solutions to one to solutions to the other- see Figure 2.

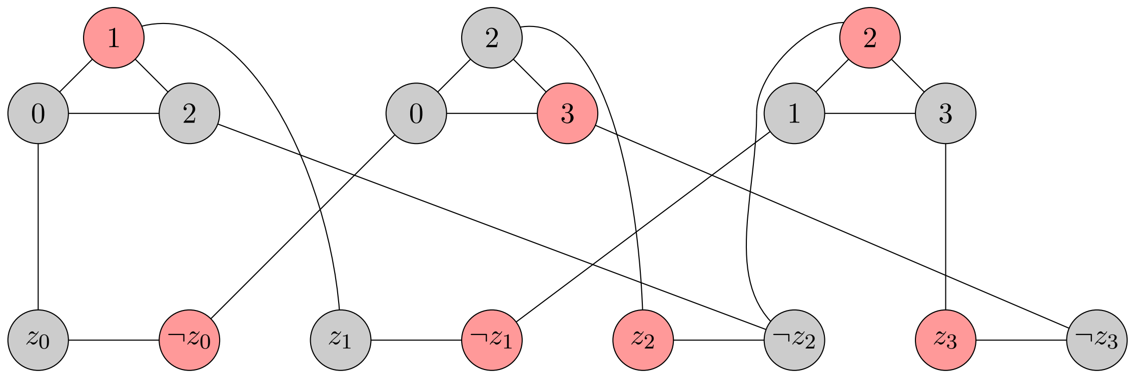


Figure 2: Independent set of size $4 + 3 = 7$ when the satisfiable assignment is $(0, 0, 1, 1)$.

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 φ is satisfiable. Moreover, we can map independent sets of size k to satisfying assignments of φ in polynomial time.*

Proof of claim.

Given a satisfying assignment α to φ, 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 α. For each clause gadget, pick a single vertex that is connected to a literal not satisfied by α. (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.)

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 φ. 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 α 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 $\mathsf{NP}_{\mathsf{search}}$-complete. □

6 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 $\mathsf{NP}_{\mathsf{search}}$-complete, we'll see now that changing "two" to "three" makes that efficiently solvable problem $\mathsf{NP}_{\mathsf{search}}$-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 $\mathsf{NP}_{\mathsf{search}}$-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).

Input	: 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.
Output	: A set of hyperedges which are disjoint and cover all the vertices, if one exists.

Computational Problem ThreeDimensionalMatching

Theorem 6.1. *ThreeDimensionalMatching (AKA 3DM) is* $\mathsf{NP}_{\mathsf{search}}$*-complete.*

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

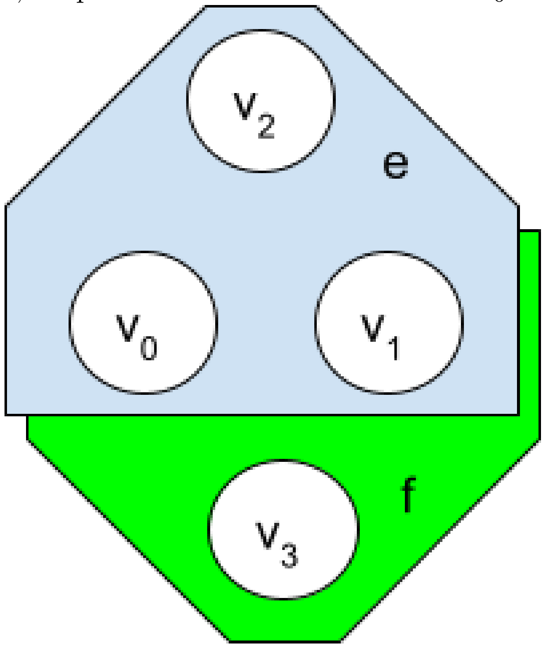
$\mathsf{NP}_{\mathsf{search}}$ membership of 3DM. To show that ThreeDimensionalMatching is in NP, we need to show that there exists a polynomial-time algorithm that, given a potential solution y (that is, a set of triples of vertices denoting a hyperedge), checks whether it's a set of hyperedges which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

$\mathsf{NP}_{\mathsf{search}}$-hardness of 3DM: reduction from 3SAT To show that ThreeDimensionalMatching is $\mathsf{NP}_{\mathsf{search}}$-hard, we need to show that every problem in $\mathsf{NP}_{\mathsf{search}}$ reduces to it in polynomial time. We'll again use the fact that every problem in $\mathsf{NP}_{\mathsf{search}}$ 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 $\mathsf{NP}_{\mathsf{search}}$ 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.

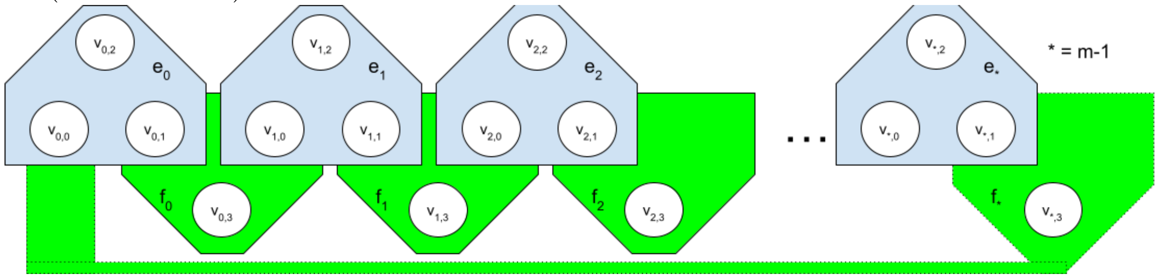
The full reduction is complicated (see the end of these notes), so we'll work our way up to it, building 3DM gadgets that simulate gradually more of 3SAT.

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).

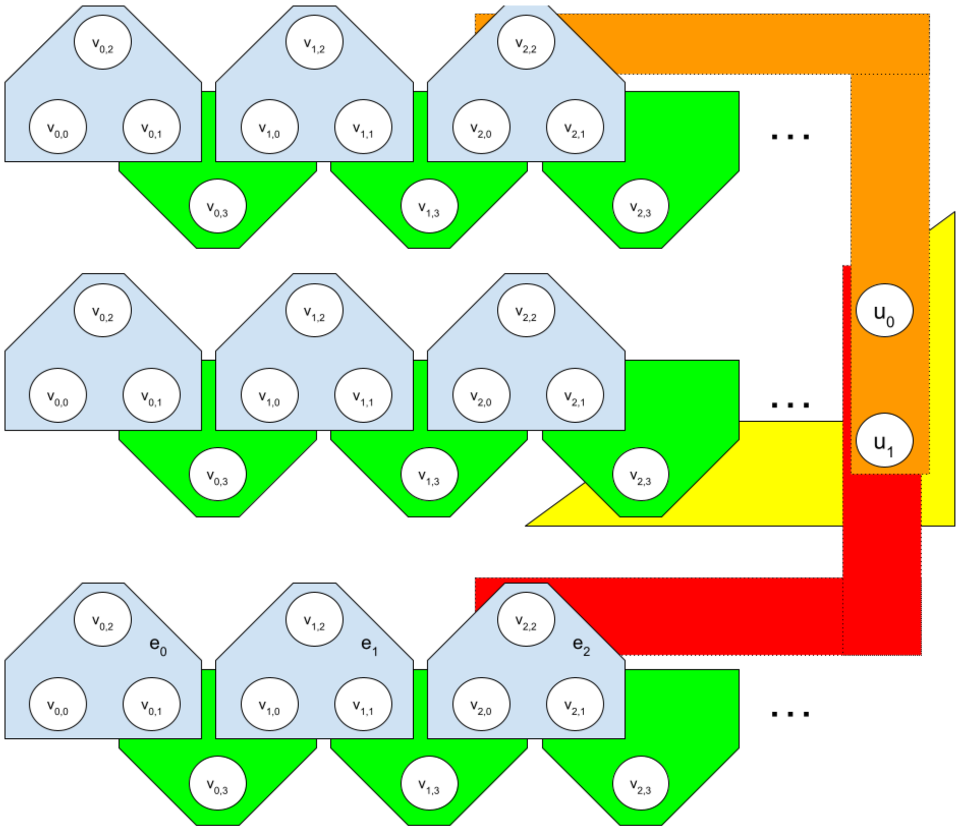


As before, there are two choices a three-dimensional matching can make about hyperedges: if we pick the first hyperedge e_0, we must not pick the second hyperedge f_0 (because it overlaps in $v_{0,1}$), so we must pick the third hyperedge e_1 (because $v_{1,0}$ needs to be covered), and so on. If we 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 jth 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[1] 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}.

[1]A 3SAT clause may have fewer than three literals.



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 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 jth 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 φ with n variables $z_0, z_1, \ldots, z_{n-1}$ that are used in clauses[2] and m clauses $C_0, \ldots, C_{m-1}$, 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 $(v_{i,j+1,0,\ell}, v_{i,j,1,\ell}, v_{i,j,3,\ell})$. (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.

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.

$\mathsf{NP}_{\mathsf{search}}$-hardness of 3DM: runtime of the reduction The reduction from 3SAT to 3DM is an algorithm that takes as input a 3SAT formula φ 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 $O(m^2)$.

[2]If any variables are not used in any clauses, ignore them: they can be set arbitrarily.

$\mathsf{NP}_{\mathsf{search}}$-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 φ that's unsatisfiable, the output is $\bot$, and for every input φ 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.

$\mathsf{NP}_{\mathsf{search}}$-hardness of 3DM: proof of soundness of the reduction To prove soundness, we need to prove that if φ is unsatisfiable, the reduction returns $\bot$. 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.

$\mathsf{NP}_{\mathsf{search}}$-hardness of 3DM: proof of completeness of the reduction To prove completeness, we need to prove that if φ has some satisfying assignment α, the reduction returns a satisfying assignment (not necessarily α). 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 α, pick all the "true" hyperedges in G's variable gadgets for x_i.

2. For each variable that's false in α, pick all the "false" hyperedges in G's variable gadgets for x_i.

3. For each clause C_j, choose α's first true literal in it (one exists because α 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 necesssarily 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. □