CSC373: Algorithm Design, Analysis and Complexity
Winter/Spring 2020

Allan Borodin and Sara Rahmati

Week of February 24-28, 2020

Announcements

- Midterm Monday, March 2. There is a TA office hour Thursday, Feb 27, 2-3PM in BA 3289. I am holding an additional office hour this Friday, Feb 28 1:30-2:30 in my office.
  Comments on the midterm:
- The test during the 5-7 time slot will take place in Bahen 1130.
- The test will cover divide and conquer, greedy algorithms, dynmaic programming and network flows. Complexity theory will not be on the midterm.
- You are allowed one page (two sides) of *handwritten* notes.
- No other aids are allowed. In particular, cell phones or smart watches that can communicate should be accessible.
- Students taking the test in the 4-6 time slot *must* remain in the test room until 5:20. Students taking the test in the 5-7 time slot must arrive by 5:20.

More announcements

- Requests for regrading are normally within one week of solutions being discussed or posted. For assignment 1, we will accept regrading requests up to March 2. Regrading requests for assignments must be made on Markus. All requests must have a one or two paragraph explanation as to why you think a question was not given the proper grade.
- Assignment 2 is due March 5 at 4:59. There is a correction for Q1 on A2. The desired time bound is $O(n^2 \cdot \max_i v_i)$

This weeks agenda

- Quick review of basic concepts relating to NP decisions problems, NP hardness and NP completeness.
- Polynomial time reductions and polynomial time transformations.
- Proving 3SAT is NP complete.
- How important is the P vs NP question?

What is *NP*?

- The class *NP* (Nondeterministic Polynomial Time)
- *NP* consists of all decision problems for which a solution y (to the associated search problem) can be verified in in polynomial time.
- More specifically, a set (or language) L is in the class *NP* if there is a polynomial time computable relation $R(x, y)$ and a polynomial time computable function f such that for all $x \in L$, there is a certificate y such that $|y| \leq f(|x|)$ and $R(x, y)$.

Examples in *NP* (besides everything in *P*)

- Given an integer x (in say binary of decimal representation), is it a composite number? (This is in fact a polynomial time computable decision problem.)
- Given a graph G, can it be vertex colored in 3 colors?
- Given a set $S = \{a_i\}$ of integers can it partitioned into two subsets S_1 and S_2 such that $\sum_{a_i \in S_1} a_i = \sum_{a_i \in S_2} a_i$?

P versus NP

- P: Problems for which solutions/certificates can be efficiently found
- NP: Problems for which solutions/certificates can be efficiently verified
- Simple fact: $P \subseteq NP$.

Conjecture

$$P \neq NP$$

- Most computer scientists believe this conjecture.
- But is seems to be incredibly hard to prove.

Why is proving $P \neq NP$ difficult?

- One reason is that some search problems in NP turn out to be relatively easy. An example is the maximum bipartite matching problem introduced in Week 4). More generally, matching in any graph is polynomial time solvable but this is not an easy result.

The matching problem for undirected graphs

Given a large group of people, we want to pair them up to work on projects. We know which pairs of people are compatible, and (if possible) we want to put them all in compatible pairs.

- If there are 50 or more people, a brute force approach of trying all possible pairings would take billions of years.
- However in 1965 Jack Edmonds found an ingenious efficient algorithm. So this problem is in P.
- There is often a "fine line" between what is and what is not known to be efficiently solvable (e.g. polynomial time 2SAT vs NP-hard 3SAT).

NP-Complete Problems

- These are (in a certain sense) the hardest NP problems.
- A problem A is p-reducible to a problem B if an "oracle" (i.e. a subroutine) for B can be used to efficiently solve A.
- If A is p-reducible to B, then any efficient procedure for solving B can be turned into an efficient procedure for A.
- If A is p-reducible to B and B is in P, then A is in P.

Definition

A problem B is NP-complete if B is in NP and every problem A in NP is p-reducible to B.

Theorem

If A is NP-complete and A is in P, then $P = NP$.

To show $P = NP$ you just need to find a fast (polynomial-time) algorithm for any one NP-complete problem!!!

Conjunctive normal form propositional formulas

The SAT problem is defined in terms of inputs given as conjunctive normal form (CNF) formulas.

Here are the standard definitions regarding CNF propositional formulas:

- A literal is a propositional variable x or its negation $\bar{x}$.
- A clause $C = \ell^1 \vee \ell^2 \ldots \vee \ell^r$ is a disjunction of literals.
- A CNF formula $F = C_1 \wedge C_2 \ldots \wedge C_m$ is a conjunction of clauses.
- A CNF formula is a kCNF formula if every clause has at most k literals. For our purposes we will abuse terminology and say that every clause has exactly k literals. It is always assumed that no variable appears twice in any clause.

$$(\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$$

Figure: An example of a 3CNF formula

Sat and 3SAT

A formula is *satisfiable* if there is an assignment of truth values (i.e. TRUE, FALSE) to the propositional variables such that the formula evaluate to TRUE. For CNF formulas, this means that there is an assignment of truth values such that every clause is TRUE.

SAT (resp. 3SAT) is the decision problem that determines if a CNF (resp. 3CNF) formula is satisfiable.

Following the initial results of Cook and Karp, our development of *NP* complete problems rests on showing that SAT is *NP* complete. We have to show how to reduce *any NP* decision problem to SAT.

We delay the proof of that Theorem until after some examples of reductions. Some reductions will be relatively easy and some not.

Many research papers were written in the early 1970's establishiing NP-completeness for specific problems. The monograph by Garey and Johnson was a valued reference keeping track of many examples. There are web sites devoted to particular domains listing NP complete problems.

- A great many (literally, thousands) of problems have been shown to be *NP*-complete.
- Most scheduling related problems (delivery trucks, exams etc) are *NP*-complete.
- The following simple exam scheduling problem is *NP*-complete:

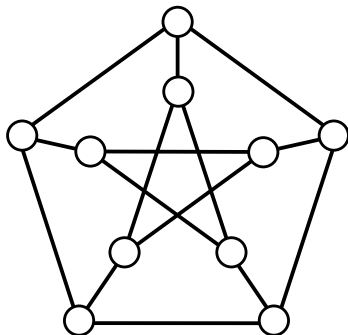
Example

- We need to schedule N examinations, and only three time slots are available.
- We are given a list of exam conflicts: A conflict is a pair of exams that cannot be offered at the same time, because some student needs to take both of them.
- **Problem:** Determine if there is a way of assigning each exam to one of the time slots T_1, T_2, T_3, so that no two conflicting exams are assigned to the same time slot.
- This problem is known as the graph 3-colourability problem.

Graph 3-Colourability

Problem

Given a graph, determine whether each node can be coloured red, blue, or green, so that the endpoints of each edge have different colours.

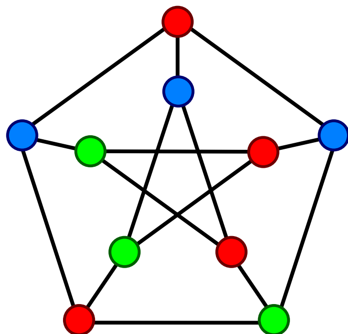


Imagine trying to decide this when there are say hundreds or thousands of nodes.

Graph 3-Colourability

Problem

Given a graph, determine whether each node can be coloured red, blue, or green, so that the endpoints of each edge have different colours.



Imagine trying to decide this when there are say hundreds or thousands of nodes.

Some more remarks on graph coloring

- The natural graph coloring optimization problem is to color a graph with the fewest number of colors.
- We can phrase it as a search or decision problem by saying that the input is a pair (G, k) and then
  1. The search problem is to find a k-coloring of the graph G if one exists.
  2. The decision problem is to determine whether or not G has a k coloring.
  3. Clearly, solving the optimization problem solves the search problem which in turn solves the decision problem.
  4. Conversely, if we can efficiently solve an NP complete decision problem then we can efficiently solve the search and optimization problems. This can be shown in general for all NP-complete problems but we can also show it explicitly for specific problems (e.g., for graph coloring as you should try to show).
- Formally it is the graph coloring decision problem which is NP-complete. More precisely, the graph coloring decision problem for any fixed $k \geq 3$ is NP-complete. However, 2-Colorability is in P.
- Search or optimization problems to which any NP-complete decision problem can be reduced are then called NP-hard.

Reducing Graph 3-Colourability to 3SAT

We begin our examples of reductions between *NP* decision problems with a reduction that is implied by the fact that graph 3-colouring is in *NP* and hence must reduce to 3SAT which is *NP*-complete. This reduction would now be considered a relatively easy reduction (certainly in hindsight) but it illustrates how reductions can be between problems coming from what are traditionally thought of as different research areas.

- We are given a graph G with nodes, say $V = \{v_1, v_2, \ldots, v_n\}$
- We are given a list of edges, say $(v_3, v_5), (v_2, v_6), (v_3, v_6), \ldots$
- We need to find a 3CNF formula F which is satisfiable if and only if G can be colored with 3 colors (say red, blue, green). **Note:** Any permutation or renaming of the colors does the change what follows.
- We use three different types of Boolean VARIABLES
  $r_1, r_2, \ldots, r_n$ (r_i means node i is colored red)
  $b_1, b_2, \ldots, b_n$ (b_i means node i is colored blue)
  $g_1, g_2, \ldots, g_n$ (g_i means node i is colored green)

Here we are abusing terminology as "means" is really "intended meaning"

- Here are the CLAUSES for the formula F:
  - We need one clause for each node:
    $(r_1 \vee b_1 \vee g_1)$ (node 1 gets at least one color)
    $(r_2 \vee b_2 \vee g_2)$ (node 2 gets at least one color)
    ...
    $(r_n \vee b_n \vee g_n)$ (node n gets at least one color)
  - We could put in clauses saying that no node gets colored with more than one color but coloring a node with more than one color can only make it more difficult to color so we really don't need these clauses.
  - We need 3 clauses for each edge: For the edge (v_3, v_5) we need
    $(\overline{r_3} \vee \overline{r_5})$ (v_3 and v_5 not both red)
    $(\overline{b_3} \vee \overline{b_5})$ (v_3 and v_5 not both blue)
    $(\overline{g_3} \vee \overline{g_5})$ (v_3 and v_5 not both green)

- The size of the formula F is comparable to the size of the graph G.

- **Check:** G is 3-colorable if and only if F is satisfiable.

What the previous reduction *does not* show

NOTE: Assuming that 3-SAT is *NP*-complete and showing that 3-colourability reduces to 3SAT does not prove that 3-colourability is NP complete.

This is the *wrong* direction of the reduction. Given the NP-completness of 3SAT, the reduction of 3SAT to 3-colourability would show that 3-colourability is *NP*-hard.

It is easy to see that 3-colorability is in *NP* so then 3-colourability becomes *NP*-complete.

Summarizing: To show that a problem Y is *NP*-hard, given that a problem X is known to be *NP*-hard, we show that X can be reduced to Y.

On the nature of the previous polynomial time reduction

- If we consider the previous reduction of 3-coloring to 3-SAT, it can be seen as a very simple type of reduction.
- Namely, given an input w to the 3-coloring problem, it is transformed (in polynomial time) to say $h(w)$ such that

$$w \in \{G \mid G \text{ can be 3-colored}\} \text{ iff}$$
$$h(w) \in \{F \mid F \text{ is a satisfiable 3CNF formula}\}.$$

- If we express the problems as decision probems, the reduction of bipartite matching to maximum flows is also a transformation.

Polynomial time transformations

- We say that a language L_1 is polynomial time transformable to L_2 if there exists a polynomial time computable function h such that

$$w \in L_1 \text{ iff } h(w) \in L_2.$$

- The function h is called a polynomial time transformation.

Polynomial time reductions and transformations

- In practice, when we are reducing one NP problem to another NP problem, it will be a polynomial time transformation.
- We will use the same notation $\leq_p$ to denote a polynomial time reduction and polynomial time transformation.
- As we have observed before if $L_1 \leq_p L_2$ and $L_2 \in P$, then $L_1 \in P$.
- The contrapositive says that if $L_1 \leq_p L_2$ and $L_1 \notin P$, then $L_2 \notin P$.

$\leq_p$ is transitive

- An important fact that we will use to prove NP completeness of problems is that polynomial time reductions are transitive.
- That is $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$ implies $L_1 \leq_p L_3$.

- The proof for transformations is easy to see. For say that $L_1 \leq_p L_2$ via g and $L_2 \leq_p L_3$ via h, then $L_1 \leq_p L_3$ via $h \circ g$; that is, $w \in L_1$ iff $h(g(w) \in L_3$.

Polynomial reductions/transformations continued

- One fact that holds for polynomial time transformation but is believed not to hold for polynomial time reductions is the following:

NP closed under polynomial time transformation

If $L_1 \leq_p L_2$ and $L_2 \in NP$ then $L_1 \in NP$.

- The closure of NP under polynomial time transformations is also easy to see. Namely,

Suppose

- $L_2 = \{w \mid \exists y, |y| \leq q(|w|) \text{ and } R(w, y)\}$ for some polynomial time relation R and polynomial q, and
- $L_1 \leq_p L_2$ via h.

Then

$L_1 = \{x \mid \exists y', |y'| \leq q(|h(x)| \text{ and } R'(x, y')\}$ where $R'(x, y') = R(h(x), y')$

Polynomial reductions/transformations continued

- On the other hand we do not believe that NP is closed under general polynomial time reductions.
- Specifically, for general polynomial time transformations we have $\bar{L} \leq_p L$. Here $\bar{L} = \{w | w \notin L\}$ is the language complement of L.
- We do not believe that NP is closed under language complementation.
- For example, how would you provide a short verification that a propositional formula F is *not* satisfiable? Or how would you efficiently verify that a graph G *cannot* be 3-coloured?
- While we will use polynomial time transformations between decision problems/languages we need to use the more general polynomial time reductions to say reduce a search or optimization problem to a decision problem.

So how do we show that a problem is NP complete?

- Showing that a language (i.e. decision problem) L is NP complete involves establishing two facts:

1. L is in NP
2. Showing that L is NP-hard; that is showing

$$L' \leq_p L \text{ for every } L' \in NP$$

- Usually establishing $L \in NP$ is relatively easy and is done directly in terms of the definition of $L \in NP$.
  - That is, one shows how to verify membership in L by exhibiting an appropriate certificate. (It could also be established by a polynomial time transformation to a known $L \in NP$.)
- Establishing that L is $NP - hard$ is usually done by reducing some known NP complete problem L' to L.

But how do we show that there are any *NP* complete problems?

How do we get started?

- Once we have established that there exists at least one *NP* complete problem then we can use polynomial time reductions and transitivity to establish that many other *NP* problems are *NP* hard.
- Following Cook's original result, we will show that *SAT* (and even 3*SAT*) is NP complete "from first principles".

- It is easy to see that *SAT* is in *NP*.
- We will (later) show that *SAT* is *NP* hard by showing how to encode an arbitrary "non-deterministic" polynomial time (Turing) computation by a *CNF* formula. We can simply think of such a computation as one that "guesses" a certificate (i.e. makes non-deterministic Turing machine operations) and then verifies the certificate.

A tree of reductions/transformations

Polynomial-Time Reductions

constraint satisfaction

3-SAT

3-SAT reduces to INDEPENDENT SET

Dick Karp (1972)
1985 Turing Award

INDEPENDENT SET → VERTEX COVER → SET COVER

DIR-HAM-CYCLE → HAM-CYCLE → TSP

GRAPH 3-COLOR → PLANAR 3-COLOR

SUBSET-SUM → SCHEDULING

packing and covering | sequencing | partitioning | numerical

A little history of NP-completenes

- In his original 1971 seminal paper, Cook was interested in theorem proving. Stephen Cook won the Turing award in 1982
- Cook used the general notion of polynomial time reducibility which is called polynomial time Turing reducibility and sometimes called Cook reducibility.
- Cook established the NP completeness of 3SAT as well as a problem that includes CLIQUE $= \{(G, k) | G$ has a k clique $\}$.
- Independently, in the (former) Soviet Union, Leonid Levin proved an analogous result for SAT (and other problems) as a search problem.
- Following Cook's paper, Karp exhibited over 20 prominent problems that were also NP-complete.
- Karp showed that polynomial time transformations (sometimes called polynomial many to one reductions or Karp reductions) were sufficient to establish the NP completness of these problems.

Another historical note

We should note that the concepts of polynomial time reducibility (Cook reducibility) and polynomial time transformation (Karp reduction) have precedents in computability theory (also called reucrsive function theory).

In computability theory, we are concerned with what is and isn't computable without any consideration of complexity issues.

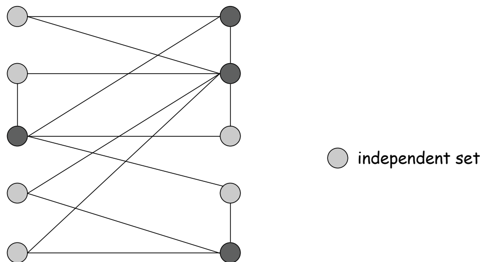
The analogue of P are the computable decision problems (also called recursive sets) and the analogue of NP is called the recursively enumerable (also called semi-computable).

The analogue of Cook reduction is called Turing reduction and the analogue of Katp reduction is called many-to-one reduction. This work started with Turing in his 1936 paper and conituned as an active field into the 40's and 50's.

Independent Set is *NP* complete

The independent set problem

- Given a graph $G = (V, E)$ and an integer k.
  Note that for every fixed k, there is a brute force $|V|^k$ time algorithm.
- Is there a subset of vertices $S \subseteq V$ such that $|S| \geq k$, and for each edge at most one of its endpoints is in S?

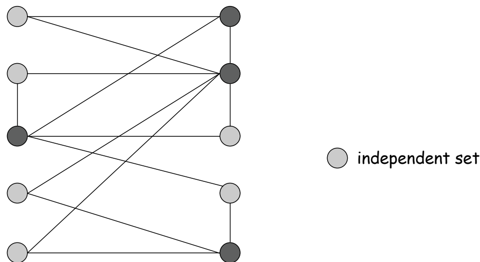


- Question: Is there an independent set of size 6?

Independent Set is NP complete

The independent set problem

- Given a graph $G = (V, E)$ and an integer k.
  Note that for every fixed k, there is a brute force $|V|^k$ time algorithm.
- Is there a subset of vertices $S \subseteq V$ such that $|S| \geq k$, and for each edge at most one of its endpoints is in S?

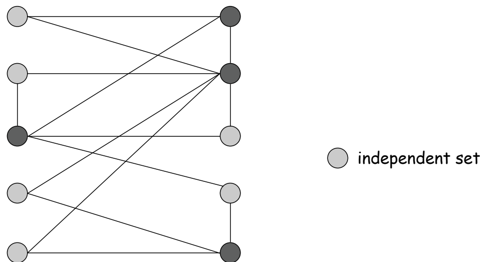


- Question: Is there an independent set of size 6? Yes.

Independent Set is *NP* complete

The independent set problem

- Given a graph $G = (V, E)$ and an integer k.
  Note that for every fixed k, there is a brute force $|V|^k$ time algorithm.
- Is there a subset of vertices $S \subseteq V$ such that $|S| \geq k$, and for each edge at most one of its endpoints is in S?

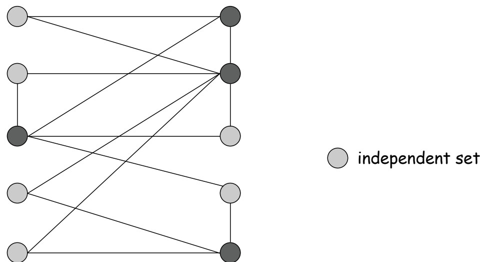


- Question: Is there an independent set of size 6? Yes.
- Question: Is there an independent set of size 7?

Independent Set is *NP* complete

The independent set problem

- Given a graph $G = (V, E)$ and an integer k.
  Note that for every fixed k, there is a brute force $|V|^k$ time algorithm.
- Is there a subset of vertices $S \subseteq V$ such that $|S| \geq k$, and for each edge at most one of its endpoints is in S?



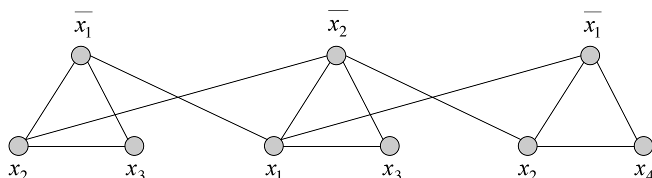
- Question: Is there an independent set of size 6? Yes.
- Question: Is there an independent set of size 7? No.

3SAT reduces to Independent Set

Claim

3SAT $\leq_p$ Independent Set

- Given an instance F of 3SAT with k clauses, we construct an instance (G, k) of Independent Set that has an independent set of size k iff F is satisfiable.
- G contains 3 vertices for each clause; i.e. one for each literal.
- Connect 3 literals in a clause in a triangle.
- Connect literal to each of its negations.



$$\Phi = (\overline{x_1} \lor x_2 \lor x_3) \land (x_1 \lor \overline{x_2} \lor x_3) \land (\overline{x_1} \lor x_2 \lor x_4)$$

Subset Sum

Subset Sum

- Given a set of integers $S = \{w_1, w_2, \ldots, w_n\}$ and an integer W.
- Is there a subset $S' \subseteq S$ that adds up to exactly W?

Example

- Given $S = \{1, 4, 16, 64, 256, 1040, 1041, 1093, 1284, 1344\}$ and $W = 3754$.
- Question: Do we have a solution?
- Answer: Yes. $1 + 16 + 64 + 256 + 1040 + 1093 + 1284 = 3754$.

3SAT reduces to Subset Sum

Claim

3SAT $\leq_p$ Subset Sum

- Given an instance F of 3SAT, we construct an instance of Subset Sum that has solution iff F is satisfiable.
- In the following array (next slide), rows represent integers represented in decimal. The colun entry in each row represents one digit of the integer. For each propositional variable we have a column specifying that each variable has just one truth assignment (i.e., $true = 1$) and for each clause we have a column saying that the clause is satisfiable. The "dummy rows" make it possible to sum each column to 4 if and only if there is at least one literal set to true. Note that the decimal representation insures that addition in each column will not carry over to the next column. As mentioned in class, we could have inserted 3 dummy rows for each clause where each dummy row would contain a '1' for that clause.

3SAT reduces to Subset Sum continued

The figure illustrates how a specific 3CNF formula is transformed into a set of integers and a target (bottow row).

$$C_1 = \bar{x} \vee y \vee z$$
$$C_2 = x \vee \bar{y} \vee z$$
$$C_3 = \bar{x} \vee \bar{y} \vee \bar{z}$$

	x	y	z	C_1	C_2	C_3
x	1	0	0	0	1	0
¬ x	1	0	0	1	0	1
y	0	1	0	1	0	0
¬ y	0	1	0	0	1	1
z	0	0	1	1	1	0
¬ z	0	0	1	0	0	1
dummies to get clause columns to sum to 4	0	0	0	1	0	0
	0	0	0	2	0	0
	0	0	0	0	1	0
	0	0	0	0	2	0
	0	0	0	0	0	1
	0	0	0	0	0	2
W	1	1	1	4	4	4

Reviewing how to show some L is NP complete.

- We must show $L \in NP$. To do so, we provide a polynomial time verification predicate $R(x, y)$ and polynomial length certificate y for every $x \in L$; that is, $L = \{x | \exists y, R(x, y) \text{ and } |y| \leq q(|x|)\}$.

- We must show that L is NP hard (say with respect to polynomial time tranformations); that is, for some known NP complete L', there is a polynomial time transducer function h such that $x \in L'$ iff $h(x) \in L$. This then establishes that $L' \leq_p L$.

- Warning The reduction/transformation $L' \leq_p L$ must be in the correct direction and h must be defined for every input x; that is, one must also show that if $x \notin L'$ then $h(x) \notin L$ as well as showing that if $x \in L'$ then $h(x) \in L$.

Some transformations are easy, some are not

- Tranformations are (as we have been arguing) algorithms computing a function and hence like any algorithmic problem, sometimes there are easy solutions and sometimes not.

- In showing NP-completeness it certainly helps to choose the right known NP-complete problem to use for the transformation.

- In the Karp tree, there are some transformations that are particularly easy such as :
  - IndependentSet $\leq_p$ VertexCover
  - VertexCover $\leq_p$ SetCover

- A transforrmation of moderate difficulty is 3SAT $\leq_p$ 3-COLOR

- I am using Kevin Wayne's slides to illustrate the transformation. See slides for "Poly-time reductions" in

  `http://www.cs.princeton.edu/courses/archive/spring05/cos423/lectures.php`

3CNF $\leq_p$ 3-COLOR: Outline of Transformation

Claim. 3-SAT $\leq_P$ 3-COLOR.

Pf. Given 3-SAT instance Φ, we construct an instance of 3-COLOR that is 3-colorable iff Φ is satisfiable.

Construction.

i. For each literal, create a node.
ii. Create 3 new nodes T, F, B; connect them in a triangle, and connect each literal to B.
iii. Connect each literal to its negation.
iv. For each clause, add gadget of 6 nodes and 13 edges.

↑ to be described next

If ϕ is a 3CNF formula in n variables and m clauses, then $h(\phi) = G_\phi$ will have $2n + 6m + 3$ nodes and $3n + 13m + 3$ edges.

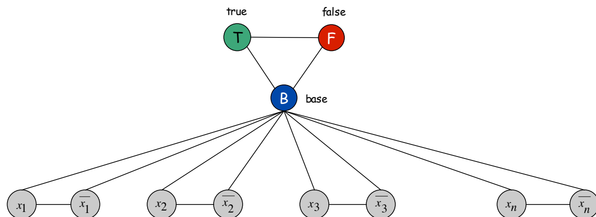
3CNF $\leq_p$ 3-COLOR: Consistent literals

This slide and the next slide are perhaps better thought of as illustrating that if Φ is not satisfiable then then G is not 3 colorable.

Claim. Graph is 3-colorable iff Φ is satisfiable.

Pf. $\Rightarrow$ Suppose graph is 3-colorable.

- Consider assignment that sets all T literals to true.
- (ii) ensures each literal is T or F.
- (iii) ensures a literal and its negation are opposites.

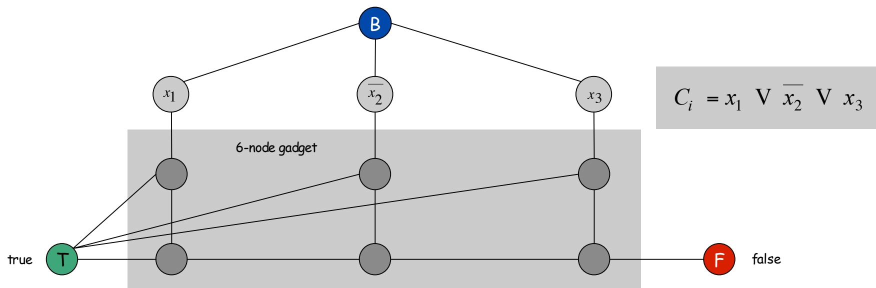


3CNF $\leq_p$ 3-COLOR: The clause gadget

Claim. Graph is 3-colorable iff Φ is satisfiable.

Pf. $\Rightarrow$ Suppose graph is 3-colorable.

- Consider assignment that sets all T literals to true.
- (ii) ensures each literal is T or F.
- (iii) ensures a literal and its negation are opposites.
- (iv) ensures at least one literal in each clause is T.

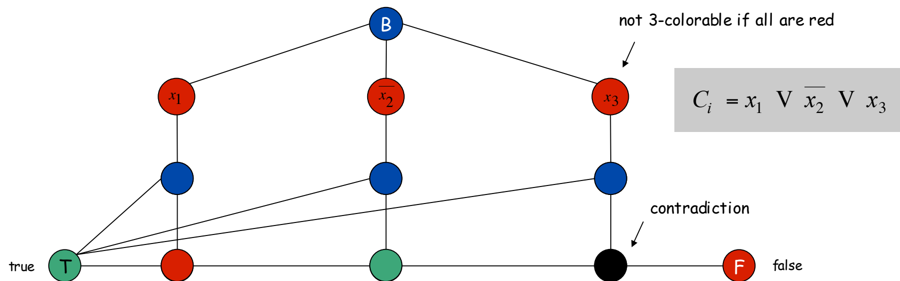


G_ϕ is 3-colourable $\Rightarrow$ ϕ satisfiable

Claim. Graph is 3-colorable iff Φ is satisfiable.

Pf. $\Rightarrow$ Suppose graph is 3-colorable.

- Consider assignment that sets all T literals to true.
- (ii) ensures each literal is T or F.
- (iii) ensures a literal and its negation are opposites.
- (iv) ensures at least one literal in each clause is T.

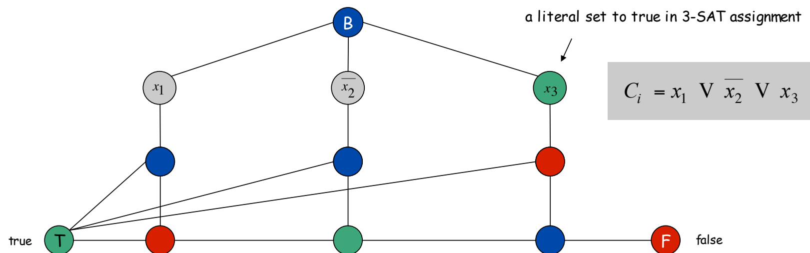


ϕ satisfiable $\Rightarrow G_\phi$ is 3-colourable

Claim. Graph is 3-colorable iff Φ is satisfiable.

Pf. $\Leftarrow$ Suppose 3-SAT formula Φ is satisfiable.

- Color all true literals T.
- Color node below green node F, and node below that B.
- Color remaining middle row nodes B.
- Color remaining bottom nodes T or F as forced. ▪



- Note we are choosing precisely one green node in each clause to force a proper colouring.

Finally, how does one show that 3SAT is *NP*-complete

We now know how to generate more and more *NP*-compplete problems **assumming** that 3SAT is *NP*-complete. But how do we know that 3SAT (or any problem if we don't use 3SAT) is *NP*-complete. Without at least one *NP*-hard problem to begin, we cannot generate other *NP*-hard problems.

Following Cook's original proof, we can show how we can encode Turing machine computations in the language of propositional logic. This will yield the desired fresult that 3SAT is *NP*-hard (and therefore *NP*-complete).

CLRS uses a circuit value problem relying on our understanding of Boolean circuits. Although perhaps more intuitive, there is some hand-waving in that approach whereas the TM encoding shows how we can make everything precise. Note: We may not be proving all the claims but argue that there is enough detail to be convinced that this encoding of TM computations is sufficient to obtain the dsesired result.

Turing machines

- We are using the classical one tape TM. This is the simplest variant to formalize which will enable the proof for the NP completeness of SAT. In the proof, we are assuming (without loss of generality) that all time bounds $T(n)$ are computable in polynomial time.
- Claim Any reasonable (classical) computing model algorithm running in time $T(n)$, can be simulated by a TM in time $T(n)^k$ for some k. Hence we can use the TM model in the definition of P and NP.
- Since we are only considering decision problems we will view TMs that are defined for decision problems and hence do not need an output other than a reject and accept state.
- Following standard notation, formally, a specific TM is defined by a tuple $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$
- Next week, we will briefly explain (using the board) the model and notation. Note that Q, Σ, Γ are all finite sets.