All in One - CS701 - Final Papers - Spring 2015 [Solutions]

By: Muhammad Asif Mansoor – MSCS VU

[CS701 - Paper 1 - Spring 2015]

Question 1:

Let SET-SPLITTING = {hS,Ci| S is a finite set and C = {C1, . . . ,Ck} is a collection of subsets of S, for some k > 0, such that elements of S can be colored red or blue so that no Ci has all its elements colored with the same color}. Show that SET-SPLITTING is NP-complete.

Solution:

SET-SPLITTING in in NP because a coloring of the elements can be guesses and verified to have the desired properties in polynomial time. We give a polynomial time reduction f from *3SAT* to *SET-SPLITTING* as follows. Given a 3cnf ϕ, we set $S = \{x_1, \overline{x_1}, \ldots, x_m, \overline{x_m}, y\}$. In other words, S contains an element for each literal (two for each variable) and a special element y. For every clause c_i in ϕ, let C_i be a subset of S containing the elements corresponding to the literals in c_i and the special element $y \in S$. We also add subsets C_{x_i} for each literal, containing elements x_i and $\overline{x_i}$. Then $C = \{C_1, \ldots, C_l, C_{x_1}, \ldots, C_{x_l}\}$.

If ϕ is satisfiable, consider a satisfying assignment. If we color all the true literals red, all the false ones blue, and y blue, then every subset C_i of S has at least one red element (because c_i of ϕ is "turned on" by some literal) and it also contains one blue element (y). In addition, every subset C_{x_i} has exactly one element red and one blue, so that the system $\langle S, C\rangle \in$ *SET-SPLITTING*.

If $\langle S, C\rangle \in$ *SET-SPLITTING*, then look at the coloring for S. If we set the literals to true which are colored differently from y, and those to false which are the same color as y we obtain a satisfying assignment to ϕ.

Question 2:

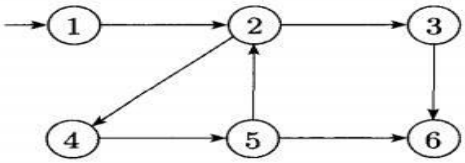
For a cnf-formula - with m variables and c clauses, show that you can construction polynomial time an NFA with O(cm) states that accepts all non-satisfying assignments, represented as Boolean strings of length m. Conclude that P = NP implies that NFAs cannot be minimized in polynomial time.

SOLUTION OUTLINE: On input ϕ, construct a NFA N that nondeterministically picks one of the c clauses (via ϵ-transitions), reads the input of length m, and accepts if it does not satisfy the clause, and rejects otherwise. In addition, N also accepts all inputs of length not equal to m. For each clause, we need $O(m)$ states, so N has $O(cm)$ states. It is clear that N can be computed in polynomial time. In addition, for any nonsatisfying assignment a, at least one clause is not satisfied, so N accepts a. Conversely, if N accepts a, some clause is not satisfied, so a is a nonsatisfying assignment. Hence, N accepts all the nonsatisfying assignments of ϕ.

Next, suppose the problem of minimizing NFAs can be done in polynomial time. Then, consider the polynomial-time algorithm that on input a 3cnf formula ϕ with m clauses, constructs a NFA N that accepts all the nonsatisfying assignments of ϕ. Observe that N accepts all binary strings iff ϕ is not satisfiable. Now, run the NFA minimizing algorithm to produce a new NFA N'. If N' contains exactly one state and accepts all binary strings, reject ϕ; otherwise, accept ϕ. This yields a polynomial-time algorithm for $3SAT$, and hence $\mathbf{P = NP}$.

Question 3:

8.3 Consider the following generalized geography game wherein the start node is the one with the arrow pointing in from nowhere. Does Player I have a winning strategy? Does Player II? Give reasons for your answers.



Player 1 start the game from node 1 and then pass to player 2 this game win the player 2 because he have more nodes than player 1 coz player 1 stuck on 2 but 2 have nodes 3 to 6 and 4 to 5,6. (Explain More)

Question 4:

Let A be the language of properly nested parentheses. For example, (()) and
((()(()))() are in A, but)(is not. Show that A is in L.

Solution:

We know that the class L contains all problems requiring only a fixed number of counters/pointers to solve them. The language A requires one counter. Initially, the counter is 0, and, as the head moves to the right along the tape, the counter is incremented by 1 if the symbol read is "(", and it is decremented by 1 if the symbol read is ")". If the counter ever becomes negative we reject the string (more closing parentheses then opening ones). If the counter is positive after reading the last symbol in the string, we reject (more opening parentheses then closing ones). Otherwise, we accept.

[CS701 - Paper 2 - Spring 2015]

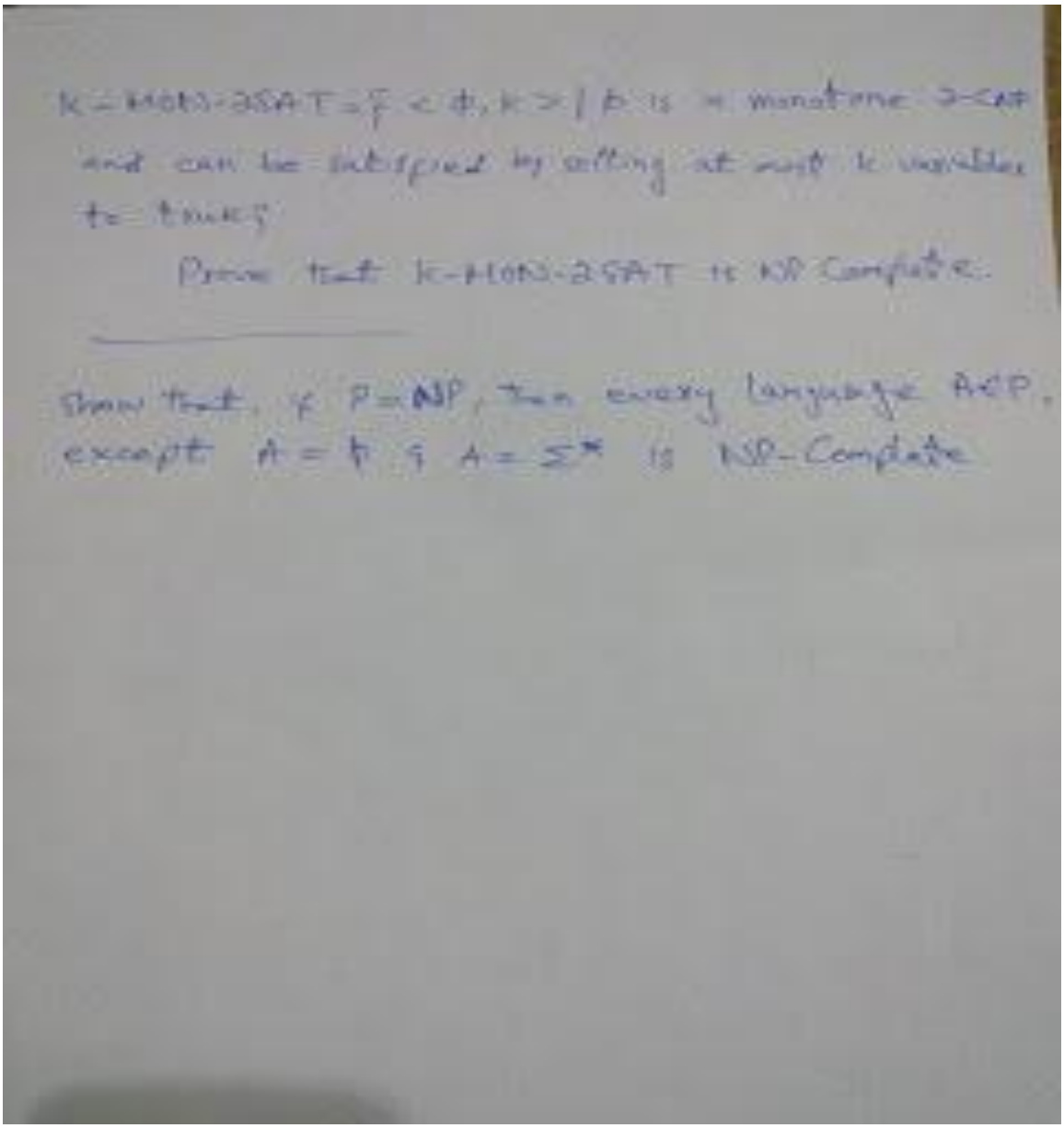
Q.1

prove that MIN-Form is NP-Complete

Can be solved as of the solution for MIN_FORMULA is in PSPACE

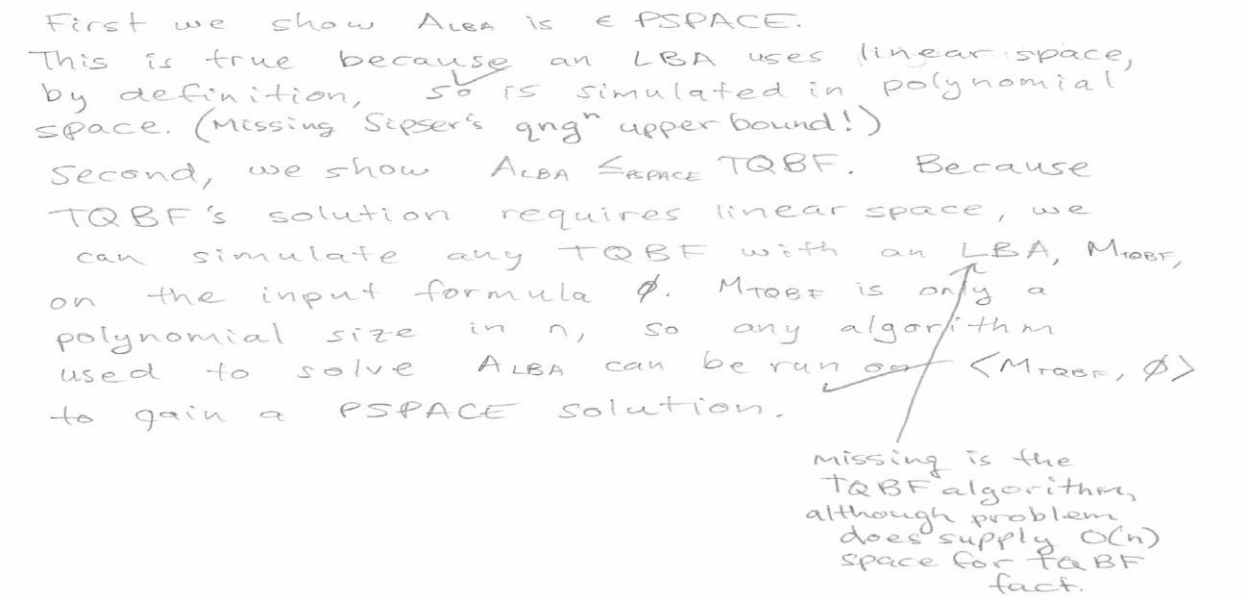
Q.2

Prove that k-MON-2SAT is in P.



Q.3

ALBA{fi such that ...} prove that ALBA is in PSPACE



Q.4

Prove that EQNFA{....} prove that EQNFA is PSPACE-complete if P=NP

- Because of the potential change in input size when the NFA is converted to a DFA, the NL algorithm for EQDFA uses NPSPACE in terms of the original input. So EQNFA is not shown to be in NL by this argument and there is no contradiction with its being PSPACE-complete.

[CS701 - Paper 3 - Spring 2015]

Question 1: Reduction through vertex cover.

7.34 A subset of the nodes of a graph G is a ***dominating set*** if every other node of G is adjacent to some node in the subset. Let

$$DOMINATING\text{-}SET = \{\langle G, k\rangle \mid G \text{ has a dominating set with } k \text{ nodes}\}.$$

Show that it is NP-complete by giving a reduction from *VERTEX-COVER*.

Solution:

Clearly Dominating Set is in **NP**. Given a dominating set, one can verify in polynomial time if that is a dominating set. This can be done by taking each vertex and checking if it is either in the given set or one of its edges travel into the set.

To show that is **NP**-complete, first of all notice that a dominating set has to include all isolated vertices (those which have no edges from them). So let us assume that our graph does not have any isolated vertices. We will show that Dominating Set is **NP**-complete using a reduction from Vertex Cover. Given a graph G, we will construct a graph G' as follows. G' has all edges and vertices of G. Also, for every edge $\{u, v\} \in G$, we add intermediate node on a parallel path in G'. Keeping $\{u, v\}$ intact in G', we add vertex w and edges $\{u, w\}$ and $\{w, v\}$ in G'. Now we will show that G has a vertex cover of size k **if and only if** G' has a dominating set of the same size.

If S is a vertex cover in G, we will show that S is a dominating set for G'. S is a vertex cover, this means that every edge in G has atleast one of its end points in S. Consider $v \in G'$. If v is an original node in G, then either $v \in S$ or there must be some edge connecting v to some other vertex u. Since S is a vertex cover, is $v \notin S$, then u must be in S, and hence there is an adjacent vertex of v in S. So v is covered by some element in S. However, if w is an additional node in G', then w has two adjacent vertices $u, v \in G$ and using the above argument at least one of them is in S. So the additional nodes are also covered by S. So if G has a vertex cover, then G' has a dominating set of at most the same size (in fact the same set itself would do).

If G' has a dominating set D of size k, then look at all the additional vertices $w \in D$. Notice that w must be connected to exactly 2 vertices $u, v \in G$. Now see that we can safely replace w by one of u or v. w in D will help us dominate only $u, v, w \in G'$. But these three edges form a 3-cycle, and we can as well pick u or v and still dominate all the vertices that w used to dominate. So we can eliminate all the additional vertices as above. Since all the additional vertices correspond to one of the edges in G, and since all of the additional vertices are covered by the modified D, this means that all the edges in G are covered by the set. So if G' has a dominating set of size k, then G has a vertex cover of size atmost k.

Question 2: 3CNF NP complete.

7.23 Let $CNF_k = \{\langle\phi\rangle | \ \phi$ is a satisfiable cnf-formula where each variable appears in at most k places$\}$.

a. Show that $CNF_2 \in$ P.

b. Show that CNF_3 is NP-complete.

Solution:

As 3SAT is NP-complete, it might be convenient for us if we are able to reduce 3SAT to CNF3. So, we need to make each variables of 3SAT appear only as most as thrice in the formula. So, we need to introduce new variables which will replace the variables appearing more than three times. Let us do this this way. If a variable appears more then three times, let us introduce a range of variables (x1, x2, ..., xn), xi for each i-th occurrence. Now to make this new version work, we need to add the translation clauses to the formula. Here is the clause,

(x1 -> x2) ^ (x2 -> x1) ^ ..... ^ (xn-1 -> xn) (xn -> x1)

We can write any xi-> xj as not(xi) OR xj.

So, we convert the translation clause as follows.
(-x1 OR x2) ^ (x2 -> x1) ^ ..... ^ (xn-1 -> xn) (xn -> x1)

When we add the formula to our original formula with all variables, appearing more than 3 times, replaced with the new variables, we get the formula in CNF3 form. The new formula has exactly the same satisfiable condition as 3SAT. As we have reduced 3SAT to a generic CNF3, it is a representative reduction over the whole set.

But as the original formula, i.e. 3SAT, is NP-complete, CNF3 is also NP-complete.

Question 3: min_formula PSPACE

MIN-FORMULA = { <F> : F is a propositional formula that is *not* equivalent to any shorter propositional formula }

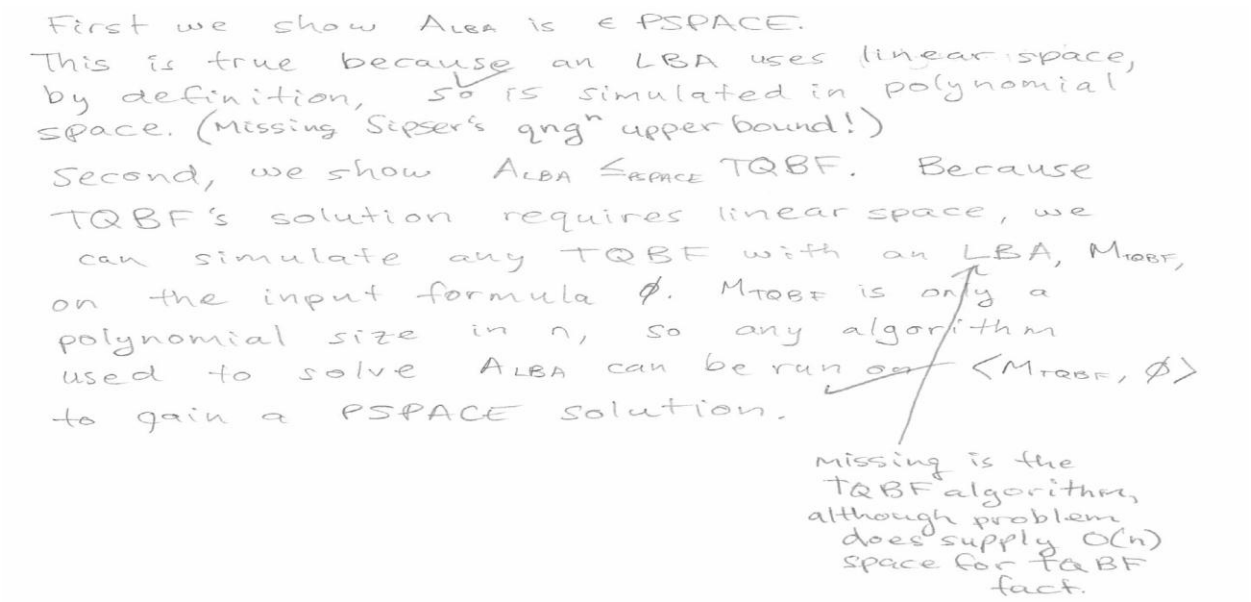
Algorithm: On input <F>:

1. Iterate over all strings s such that |s| < |<F>|, in lexicographic order, and for each one...
2. Reject if s = <F'> for some propositional formula F' such that F and F' share the same variables and F and F' are equivalent -- this can be checked by evaluating both F and F' over all possible assignments to their variables.
3. Accept if all strings of length less than |<F>| have been checked and none of them encode a formula equivalent to F.

Correctness: This rejects if F is equivalent to some shorter propositional formula (because it will find such a formula) and it accepts otherwise, i.e., it decides MIN-FORMULA.

Space: This requires linear space to store the current string s being examined, linear space to store assignments to the variables of F and F', and linear space to evaluate F and F'. All of this space can be reused from one string s to the next, so the total space usage is linear.

Question 4: ALBA PSPACE complete.



NEAR-TAUT is in co-NP. The complement of NEAR-TAUT consists of all inputs that are not well-formed expressions and inputs that are well-formed expressions such that there exist at least two non-satisfying assignments. To decide the complement in polynomial time we need only guess and check two assignments that do not satisfy the given expression. We will show that the complement is NP-complete, so it is unlikely that NEAR-TAUT is in NP.

To show the complement of NEAR-TAUT is NP-complete, we reduce SAT to it. Given an expression E, we convert it to E0 as follows:

E0 = :(E ^ (y _ :y))

If E is in SAT, then E0 must have at least two non-satisfying assignments, since we can take some satisfying assignment of E and apply it to E0, setting y to either true or false and E0 will not be satis_ed. If E0 is in the complement of NEAR-TAUT then it must have at least two non-satisfying assignments, thus E must have at least one satisfying assignment in order for the expression E ^ (y _ :y) to ever be true.

Question 1:-

Let $T = \{<M> \mid M$ is a TM that accepts wR whenever it accepts $w\}$. Show that T is undecidable. wR: w written backwards (e.g. if $w = 010111$ then $wR = 111010$)

Solution:-

We solve the problem by contradiction. Let T is decidable and R be a decider for T. We construct a TM S that decides A_{TM}.

S = "On input $<M, x>$ where M is a TM and x be a string:

1. Construct a TM M_1 as follows:
   M_1 = "On input w,
   i. If $w = 01$, accept
   ii. If $w \neq 10$, reject
   iii. If $w = 10$, run M on x, if M accepts, accept."
2. Run R on $<M_1>$
3. if R accepts, accept, if R rejects, reject."

We can observe R decides T iff S decides A_{TM}, but A_{TM} is undecidable. Hence, T is also undecidable.

7.17 Show that, if P = NP, then every language $A \in \mathrm{P}$, except $A = \emptyset$ and $A = \Sigma^*$, is NP-complete.

Let A be any language in P except $A = \emptyset$ and $A = \Sigma^*$. To show that A is NP-complete, we show that $A \in$ NP and that every $B \in$ NP is polynomial time reducible to A. The first condition holds because $A \in$ P so $A \in$ NP. To demonstrate that the second condition is true, let $x_{\text{in}} \in A$ and $x_{\text{out}} \notin A$ be two strings that are guaranteed to exist by virtue of our assumptions about A. The assumption that P = NP implies that B is recognized by a polynomial time TM M. A polynomial time reduction from B to A simulates M to determine whether its input w is a member of B, then outputs x_{in} or x_{out} accordingly.

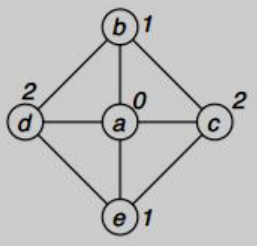
7.29 Consider the following scheduling problem. You are given a list of final exams $F_1, \ldots, F_k$ to be scheduled, and a list of students $S_1, \ldots, S_l$. Each student is taking some specified subset of these exams. You must schedule these exams into slots so that no student is required to take two exams in the same slot. The problem is to determine if such a schedule exists that uses only h slots. Formulate this problem as a language and show that this language is NP-complete.

Proof. SCHEDULE is in NP because we can guess a schedule with h slots and verify in polynomial time that no student is taking two exams in the same slot. We prove that SCHEDULE is NP-hard by showing that 3COLOR $\leq_P$ SCHEDULE. Given a graph $G = (V, E)$ we simply create an instance $\langle F, S, h \rangle$ where $F = V$, $S = E$, and $h = 3$ (which can obviously be done in polynomial time).

Given a 3-coloring for G, we can get a schedule for $\langle V, E, 3 \rangle$ by assigning finals that correspond to nodes colored with color i to slot i for $i = 0, 1, 2$. Adjacent nodes are colored by different colors, so corresponding exams taken by the same student are assigned to different slots.

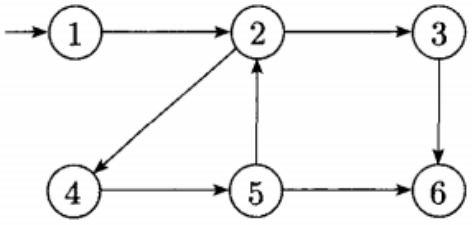
Given a schedule for $\langle V, E, 3 \rangle$, we can get a 3-coloring for the original graph by assigning colors according to the slots of corresponding nodes. Since exams taken by the same student are scheduled in different slots, adjacent nodes are assigned different colors. □

Below is an example 3-colored graph which divide the 5 finals into 3 slots for 8 students. Final a is scheduled into slot 0, b and e into slot 1, and c and d into slot 2.



F = {a, b, c, d, e}
S = {{a,b}, {a,c}, ... {e,c}}
h = 3

8.3 Consider the following generalized geography game wherein the start node is the one with the arrow pointing in from nowhere. Does Player I have a winning strategy? Does Player II? Give reasons for your answers.



Player 1 start the game from node 1 and then pass to player 2 this game win the player 2 because he have more nodes than player 1 coz player 1 stuck on 2 but 2 have nodes 3 to 6 and 4 to 5,6.

Isko thora r explain kr dena ap.

8.13 Define $A_{\text{LBA}} = \{\langle M, w\rangle | \ M \text{ is an LBA that accepts input } w\}$. Show that A_{LBA} is PSPACE-complete.

8.22

a. Let $ADD = \{\langle x, y, z\rangle | \ x, y, z > 0 \text{ are binary integers and } x + y = z\}$. Show that $ADD \in$ L.

b. Let $PAL\text{-}ADD = \{\langle x, y\rangle | \ x, y > 0 \text{ are binary integers where } x + y \text{ is an integer whose binary representation is a palindrome}\}$. (Note that the binary representation of the sum is assumed not to have leading zeros. A palindrome is a string that equals its reverse). Show that $PAL\text{-}ADD \in$ L.

Add wala question mje aya tha solve ni hy r Alba wale ka b ni hy

PLANROME SOL:

Palindrome

M1 = "On input string w:
1. Scan across the tape and look for 0s and 1s
2. Count number of 0s separately

3. Count number of 1s separately
4. compare both the counters if both are equal accept, otherwise reject

the machine counts the number of 0s and, separately, the number of 1s in binary on the work tape. The only space required is that used to record the two counters. In binary, each counter uses only logarithmic space and hence the algorithm runs in O(log n) space. Therefore, A ϵ L.

Q: STRONGLY-CONNECTED

Proof:
A language B is NL-complete if
1. B ϵ NL, and
2. every A in NL is log space reducible to B.
We know that PATH is NL-Complete. So we will construct a log space reduction from Strongly-connected to PATH. i.e
STRONGLY-CONNECTED <L PATH
Let say NTM M decides STRONGLY-CONNECTED in O(logn) space.
Functionality of M is as under
The nodes of G are the configurations of M on w.
For configuration c1 and c2 of M on w , (c1,c2) are the edges if c2 is the next possible configuration of M starting from c1.
Node s is the start configuration of M on w.
If machine reaches configuration t then accept.
Now to show that this operation has been performed in logarithmic time, we introduce a log space transducer from <G,s,t> on input w. We describe G by listing its nodes and edges. Listing of nodes is easy because each node is configuration of M on w and can be represented in clogn space for some constant c. The transducer possibally goes through all possible strings of length clogn.
Thus the STRONGLY-CONNECTED <L PATH.

Q: Pallindrome

M1 = "On input string w:
1. Scan across the tape and look for 0s and 1s
2. Count number of 0s separately
3. Count number of 1s separately
4. compare both the counters if both are equal accept, otherwise reject

the machine counts the number of 0s and, separately, the number of 1s in binary on the work tape. The only space required is that used to record the two counters. In binary, each counter uses only logarithmic space and hence the algorithm runs in O(log n) space. Therefore, A ϵ L.

Q: Mult-SAT

On input ϕ a non-deterministic TM can guess two assignments and accept both if both satisfy ϕ. Thus Double-SAT is NP. DOUBLE-SAT is mapping reducible to MULT_SAT. MULT-SAT ≤double_SAT. The following Turing machine R computes the assignments in polynomial time reduction.

R= "On input <ϕ>, a Boolean formula with variables x1,x2,. . ., xm

Let ϕ' be the ϕ □ (x v $\overline{x}$), where x is a new variable.

Output <ϕ'>"

If ϕ ϵ Double-SAT, ϕ' has at least two satisfying assignments because two satisfying conditions from the original documents of ϕ by changing the values of x.

Q: Clique reducibility to Half-clique

We give a polynomial time mapping reduction from CLIQUE to HALF-CLIQUE. The input to the reduction is a pair (G, k) and the reduction produces the graph (H) as output where H is as follows. If G has m nodes and k = m/2, then H =G. If k < m/2, then H is the graph obtained from G by adding j nodes, each connected to every one of the original nodes and to each other, where j = m − 2k.
Thus, H has m + j = 2m − 2k nodes. Observe that G has a k-clique iff H has a clique of size k + j = m − k, and so hG, ki 2CLIQUE iff hHi2HALF-CLIQUE.

If k > m/2, then H is the graph obtained by adding j nodes to G without any additional edges, where j = 2k − m. Thus, H has m + j = 2k nodes, and so G has a k-clique iff H has a clique of size k. Therefore, hG, ki 2 CLIQUE iff hHi2HALF-CLIQUE. We also need to show HALF-CLIQUE 2NP. The certificate is simply the clique.

Q: ISOMORPHIC…..

A nondeterministic polynomial time algorithm for *ISO* operates as follows:

"On input $\langle G, H \rangle$ where G and H are undirected graphs:

1. Let m be the number of nodes of G and H. If they don't have the same number of nodes, *reject*.
2. Nondeterministically select a permutation π of m elements.
3. For each pair of nodes x and y of G check that (x, y) is an edge of G iff $(\pi(x), \pi(y))$ is an edge of H. If all agree, *accept*. If any differ, *reject*."

Stage 2 can be implemented in polynomial time nondeterministically. Stage 3 takes polynomial time. Therefore $ISO \in$ NP.

In this question replace 1 with #....

U is in NP because on input $\langle d, x, 1^t \rangle$ a nondeterministic algorithm can simulate M_d on x (making nondeterministic branches when M_d does) for t steps and accept if M_d accepts. Doing so takes time polynomial in the length of the input because t appears in unary.

To show $3SAT \leq_P U$, let M be a NTM that decides $3SAT$ in time cn^k for some constants c and k. Given a formula ϕ, transform it into $w = \langle \langle M \rangle, \phi, 1^{c|\phi|^k} \rangle$. By the definition of M, $w \in U$ if and only if $\phi \in 3SAT$.

LPATH and UHMPATH question.

First, $LPATH \in$ NP because a nondeterministic machine can guess and verify a simple path of length at least k from a to b. Next, $UHAMPATH \leq_P LPATH$, because the following TM F computes the reduction f.

F = "On input $\langle G, a, b \rangle$, where G is an undirected graph, and a and b are nodes of G.

1. Let k be the number of nodes of G.
2. Output $\langle G, a, b, k \rangle$."

If $\langle G, a, b \rangle \in UHAMPATH$, G contains a Hamiltonian path of length k from a to b, thus $\langle G, a, b, k \rangle \in LPATH$. If $\langle G, a, b, k \rangle \in LPATH$, G contains a simple path of length k from a to b. Since G has k nodes, the path is Hamiltonian. Thus $\langle G, a, b \rangle \in UHAMPATH$.

The following is a nondeterministic Turing machine (NTM) that decides the *HAMPATH* problem in nondeterministic polynomial time. Recall that in Definition 7.9, we defined the time of a nondeterministic machine to be the time used by the longest computation branch.

N_1 = "On input $\langle G, s, t \rangle$, where G is a directed graph with nodes s and t:

1. Write a list of m numbers, $p_1, \ldots, p_m$, where m is the number of nodes in G. Each number in the list is nondeterministically selected to be between 1 and m.
2. Check for repetitions in the list. If any are found, *reject*.
3. Check whether $s = p_1$ and $t = p_m$. If either fail, *reject*.
4. For each i between 1 and $m - 1$, check whether (p_i, p_{i+1}) is an edge of G. If any are not, *reject*. Otherwise, all tests have been passed, so *accept*."

To analyze this algorithm and verify that it runs in nondeterministic polynomial time, we examine each of its stages. In stage 1, the nondeterministic selection clearly runs in polynomial time. In stages 2 and 3, each part is a simple check, so together they run in polynomial time. Finally, stage 4 also clearly runs in polynomial time. Thus, this algorithm runs in nondeterministic polynomial time.

CIS 511 Spring 2007: Homework 5 Solutions
Problem 1
We construct a formula ' that selects the k edges which are in the cut in such a way that ' is satis_able if and only if G has a cut of size k. The size of ' will be polynomial in jGj and k.
The variables pv correspond to vertices in V and pv set to 1 corresponds to v 2 U. Furthermore we use variables pie, for i 2 1; 2; : : :k and e 2 E. pie set to 1 indicates that e is the i-th edge in the cut.
Now we will construct formulas that say that there are exactly k edges in the cut.
'1 =^
i
(_
e
pie)
'2 =^
i
(^
e
(pie ! ^
e0 6=e
:pie0 ^ ^

j6=i
:pje))
'1 implies there are at least k edges in the cut and '2 implies that there are at most k edges in the cut.
Now we must ensure that the edges selected by '1 and '2 are in the cut, i.e. they have one endpoint in U and the other not in U.
'3 = ^
i
^
e=(u;v)
(pie ! (pu _ pv))
'4 = ^
e=(u;v)
((pu _ pv) ! _
i
pie)
'3 means that if pie is set then one of its endpoints is in U while the other is not. '4 means that, for each edge, if one of its endpoints is in U while the other is not, it must be the i-th edge in our cut for some i.
The conjunction '1 ^ '2 ^ '3 ^ '4 is thus satis_able if and only if there is a cut of size k.
The size of the formula is in O(k _ jEj2), since the size of '2 is in O(k _ jEj2) and the other formulas are smaller in size.

Problem 2

It can be easily seen that DOM-SET is in NP. The nondeterministic algorithm can guess which vertices are in the dominating set and then verify its guess was correct.
To show that DOM-SET is NP-hard, we reduce Node Cover to this problem. Let (G; k) be an instance Node Cover. We suppose without loss of generality that there are no isolated vertices in G, since these vertices do not inuence whether (G; k) is a \yes" instance. We show a polynomial algorithm that produces a graph G0 such that G has a node cover of size k i_ G0 has a dominating
1
set of size k. If G = (V;E), we de_ne G0 as follows:
G0 = (V 0 = V [fvab j (a; b) 2 Eg; E0 = E [f(a; vab); (b; vab) j (a; b) 2 Eg)
i.e. we add to G a vertex for each edge (a; b) and connect it to the vertices a and b.
First, consider the case when G has a node cover U of size k. We show that it is also a dominating set for G0. Consider a node in V . It is either in U or adjacent to a node in U, since U is a node cover in G. Consider a node in vab 2 V 0 n V . Either a or b must be in U, thus vab is adjacent to a node in U.
Second, consider the case when G0 has a dominating set W of size k. We start by constructing a dominating set W0 for G0 of size k such that it contains only nodes from V . If W contains a node vab, we can replace it by a (or b) and the set of vertices will remain a dominating set, since all the vertices adjacent to vab (i.e. a and b) will either be in W0 or adjacent to a vertex in W0. Now

we can prove that W0 is a vertex cover for G. Consider an edge (a; b). Either a or b are in W0 (since vab is adjacent to a node in W0), thus the edge (a; b) is covered.

Problem 3

First, we prove that the problem of equivalence of two formulas is in co-NP. We show an NP algorithm for the complement of the problem. It checks whether the input represents two well-formed formulas ' and . If not, the algorithm accepts immediately. If it is the case, the algorithm guesses an assignment and veri_es that ' is satis_ed while is not (or vice-versa). If P=NP, then co-NP=co-P=P. Thus P=NP implies that the problem of equivalence of two formulas is in P.

If equivalence is in P, we prove that MIN-FORM is in co-NP. Consider the following NP algorithm for the complement of MIN-FORM. It checks whether the input is a well-formed formula '. If it is not, the algorithm accepts immediately. If it is, the algorithm non-deterministically chooses a smaller formula and veri_es the equivalence of _ and and accepts if the two formulas are equivalent. The veri_cation of equivalence can be done in P (under the assumption P=NP). Thus if P=NP, MIN-FORM is in co-NP and we can conclude that P=NP implies **MIN-FORM** is in P.

Problem 4a

TRUE-SAT is in NP. We need only to test whether the expression is true when all variables are true (a polynomial-time, deterministic step) and then guess and check some other assignment. The complement of TRUE-SAT consists of all inputs that are not well-formed expressions, inputs that are well-formed expressions but that are false when all variables are true, and well-formed expressions that are only true when all variables are true. We will show TRUE-SAT is NP-complete, so it is unlikely that the complement is in NP.

2

To show TRUE-SAT is NP-complete, we reduce SAT to it. Suppose we are given an expression E with variables x1, x2, ..., xn. Convert E to E0 as follows:

1. First, test if E is true when all variables are true. If so, we know is E is satis_able, and so set E0 = E + y where y is a variable not already in E so that E0 will be accepted by TRUE-SAT.
2. Otherwise, let E0 = E + x1x2:::xn, a polynomial-time reduction. Note that E0 will always be true when all variables are true. If E is in SAT then it is satis_ed by some truth assignment other than all-true, thus E0 is in TRUE-SAT. Conversely, if E0 is in TRUE-SAT, then since x1x2:::xn is true only for the all-true assignment then E must be satis_able.

Thus we have a polynomial-time reduction from SAT to TRUE-SAT, therefore TRUE-SAT is NP-complete.

Problem 4b

FALSE-SAT is in NP. We can test whether the expression is false when all variables are false and then guess and check some other assignment. The complement of FALSE-SAT consists of all inputs that are not well-formed expressions, inputs that are well-formed expressions that are true when all variables are false, and well-formed expressions that are only false when all variables are false. We

will show FALSE-SAT is NP complete, so it is unlikely that the complement is in NP.

To show FALSE-SAT is NP-complete, we reduce TRUE-SAT to it. Suppose we are given an expression E with variables x1, x2, ..., xn. Convert E to E0 as follows:

1. Replace every instance of xi in E with :xi to obtain E00.
2. Set E0 = :E00. If E is in TRUE-SAT then E00 is true for the all-false assignment and some other assignment, thus E0 is false when all variables are false and for some other assignment. Conversely, if E0 is in FALSE-SAT then E00 is false for the all-true assignment and some other assignment, thus E is true for the all-true assignment and some other assignment.

Thus we have a polynomial-time reduction from FALSE-SAT to TRUE-SAT (which we proved above is NP-complete), therefore FALSE-SAT is NP-complete.

Problem 4c

DOUBLE-SAT is NP. We need only guess and check two assignments that satisfy the given expression. The complement of DOUBLE-SAT consists of all inputs that are not well-formed expressions and inputs that are well-formed expressions but have no more than one satisfying assignment. We will show that DOUBLE-SAT is NP complete, so it is unlikely that the complement is in NP.

To show DOUBLE-SAT is NP-complete, we reduce SAT to it. Given an expression E, we convert it to E0 as follows:

E0 = E ^ (y _ :y)

If E is in SAT then E0 must have at least two satisfying assignments, since we can take some satisfying assignment of E and apply it to E0, setting y to either true or false and E0 will be satis_ed. If E0 is in DOUBLE-SAT then E must have at least one satisfying assignment to allow E0 to be satis_ed. Since we have a polynomial-time reduction from SAT to DOUBLE-SAT then DOUBLE-SAT must be NP-complete.

Problem 4d

NEAR-TAUT is in co-NP. The complement of NEAR-TAUT consists of all inputs that are not well-formed expressions and inputs that are well-formed expressions such that there exist at least two non-satisfying assignments. To decide the complement in polynomial time we need only guess and check two assignments that do not satisfy the given expression. We will show that the complement is NP-complete, so it is unlikely that NEAR-TAUT is in NP.

To show the complement of NEAR-TAUT is NP-complete, we reduce SAT to it. Given an expression E, we convert it to E0 as follows:

E0 = :(E ^ (y _ :y))

If E is in SAT, then E0 must have at least two non-satisfying assignments, since we can take some satisfying assignment of E and apply it to E0, setting y to either true or false and E0 will not be satis_ed. If E0 is in the complement of NEAR-TAUT then it must have at least two non-satisfying assignments, thus E must have at least one satisfying assignment in order for the expression E ^ (y _ :y) to ever be true.

Q1. Dominating set through vertex cover
Q2. 3cnf
Q3. Min_formula in Pspace
Q4. ALBA in pspace complete

7.28) Let *SET-SPLITTING = {(S,C)}* *S* is a finite set and C = *{C1, . . ,Ck}* is a collection of subsets of *S,* for some k >0, such that elements of S can be colored *red* or *blue* so that no *Ci* has all its elements colored with the same color.} Show that *SET-SPLITTING* is NP-complete.

7.27 ***SET-SPLITTING*** **in in NP because a coloring of the elements can be guesses and verified to have the desired properties in polynomial time. We give a polynomial time reduction f from *3SAT* to *SET-SPLITTING* as follows.**
Given a 3cnf ϕ, we set $S = \{x_1, \overline{x_1}, \ldots, x_m, \overline{x_m}, y\}$. In other words, S contains an element for each literal (two for each variable) and a special element y. For every clause c_i in ϕ, let C_i be a subset of S containing the elements corresponding to the literals in c_i and the special element $y \in S$. We also add subsets C_{x_i} for each literal, containing elements x_i and $\overline{x_i}$. Then $C = \{C_1, \ldots, C_l, C_{x_1}, \ldots, C_{x_l}\}$.
If ϕ is satisfiable, consider a satisfying assignment. If we color all the true literals red, all the false ones blue, and y blue, then every subset C_i of S has at least one red element (because c_i of ϕ is "turned on" by some literal) and it also contains one blue element (y). In addition, every subset C_{x_i} has exactly one element red and one blue, so that the system $\langle S, C\rangle \in$ *SET-SPLITTING*.
If $\langle S, C\rangle \in$ *SET-SPLITTING*, then look at the coloring for S. If we set the literals to true which are colored differently from y, and those to false which are the same color as y we obtain a satisfying assignment to ϕ.

7.29)

4. (7.29) Consider the following scheduling problem. You are given a list of final exams $F_1, \ldots, F_k$ to be scheduled, and a list of students $S_1, \ldots S_l$. Each student is taking some specified subset of these exams. You must schedule these exams into slots so that no student is required to take two exams in the same slot. The problem is to determine if such a schedule exists and only uses h slots. The language is

$$\text{SCHEDULE} = \{\langle F, S, h\rangle \mid F \text{ is a list of finals, } S \text{ is a list of subsets specifying the finals each student is taking, and finals can be scheduled in to } h \text{ slots so that no student is taking two exams in the same slot}\}.$$

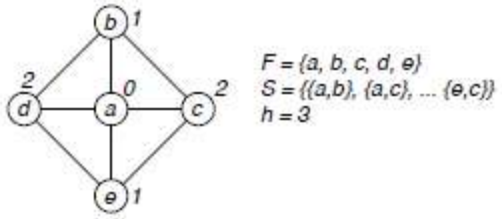
For example, if $F = \{1, 2, 3, 4\}$ and $S = \{\{1, 3\}, \{1, 2\}, \{2, 3, 4\}\}$, then $\langle F, S, 2\rangle \notin$ SCHEDULE, but $\langle F, S, 3\rangle \in$ SCHEDULE. Show that SCHEDULE is NP-complete.

Proof. SCHEDULE is in NP because we can guess a schedule with h slots and verify in polynomial time that no student is taking two exams in the same slot. We prove that SCHEDULE is NP-hard by showing that 3COLOR $\leq_P$ SCHEDULE. Given a graph $G = (V, E)$ we simply create an instance $\langle F, S, h\rangle$ where $F = V$, $S = E$, and $h = 3$ (which can obviously be done in polynomial time).

Given a 3-coloring for G, we can get a schedule for $\langle V, E, 3\rangle$ by assigning finals that correspond to nodes colored with color i to slot i for $i = 0, 1, 2$. Adjacent nodes are colored by different colors, so corresponding exams taken by the same student are assigned to different slots.

Given a schedule for $\langle V, E, 3\rangle$, we can get a 3-coloring for the original graph by assigning colors according to the slots of corresponding nodes. Since exams taken by the same student are scheduled in different slots, adjacent nodes are assigned different colors. □

Below is an example 3-colored graph which divide the 5 finals into 3 slots for 8 students. Final a is scheduled into slot 0, b and e into slot 1, and c and d into slot 2.



7.41)

7.41 For a cnf-formula ϕ with m variables and c clauses, show that you can construct in polynomial time an NFA with $O(cm)$ states that accepts all nonsatisfying assignments, represented as Boolean strings of length m. Conclude that NFAs cannot be minimized in polynomial time unless P = NP.

Sol:

SOLUTION OUTLINE: On input ϕ, construct a NFA N that nondeterministically picks one of the c clauses (via ϵ-transitions), reads the input of length m, and accepts if it does not satisfy the clause, and rejects otherwise. In addition, N also accepts all inputs of length not equal to m. For each clause, we need $O(m)$ states, so N has $O(cm)$ states. It is clear that N can be computed in polynomial time. In addition, for any nonsatisfying assignment a, at least one clause is not satisfied, so N accepts a. Conversely, if N accepts a, some clause is not satisfied, so a is a nonsatisfying assignment. Hence, N accepts all the nonsatisfying assignments of ϕ.

Next, suppose the problem of minimizing NFAs can be done in polynomial time. Then, consider the polynomial-time algorithm that on input a 3cnf formula ϕ with m clauses, constructs

a NFA N that accepts all the nonsatisfying assignments of ϕ. Observe that N accepts all binary strings iff ϕ is not satisfiable. Now, run the NFA minimizing algorithm to produce a new NFA N'. If N' contains exactly one state and accepts all binary strings, reject ϕ; otherwise, accept ϕ. This yields a polynomial-time algorithm for $3SAT$, and hence $\mathbf{P} = \mathbf{NP}$.

7.27 A *coloring* of a graph is an assignment of colors to its nodes so that no two adjacent nodes are assigned the same color. Let

3COLOR = { *(G)* I the nodes of *G* can be colored with three colors such that no two nodes joined by an edge have the same color}.

Show that *3COLOR* is NP-complete. (Hint: Use the following three subgraphs.)

Sol:

Problem 9.4

A ***coloring*** of a graph is an assignment of colors to its nodes so that no two adjacent nodes are assigned the same color. Let

$$3COLOR = \{\langle G \rangle \mid \text{the nodes of } G \text{ can be colored with three colors such that no two nodes joined by an edge have the same color}\}.$$

Show that $3COLOR$ is NP-complete. (Hint: Use the following three subgraphs.)

Sipser Problem 7.27

Don't forget that there are two parts to showing a language is NP-complete. You have to show:

1. $3COLOR \in \text{NP}$
2. Every $L \in NP$ is polynomial-time reducible to $3COLOR$.

Try showing the reduction $3SAT \leq_p 3COLOR$!

Corollary 7.42 (page 282) states that $3SAT$ is NP-complete. This means all languages are polynomial-time reducible to $3SAT$. Therefore, if we can show that $3SAT \leq_p 3COLOR$, then:

$$\forall L \in \text{NP}, \quad L \leq_p 3SAT \leq_p 3COLOR$$

Therefore, if you show $3SAT \leq_p 3COLOR$, you satisfy the second requirement of NP-completeness.

Why $3SAT$? It might be easier to find a mapping, since it has three elements just like $3COLOR$.

Tidbits: This is related to the **four color theorem**. From Wikipedia:

> "The four color theorem (also known as the four color map theorem) states that given any plane separated into regions, such as a political map of the states of a country, the regions may be colored using no more than four colors in such a way that no two adjacent regions receive the same color.. . .
>
> . . . The conjecture was first proposed in 1852 when Francis Guthrie, while trying to color the map of counties of England, noticed that only four different colors were needed."

Just in case you were wondering where all of these weird "problems" were coming from. Many of them are based on real-world problems. (There are other applications of the four-color theorem, this is just where it actually came from.)

8.17

Let A be the language of properly nested parentheses. For example, (()) and (()(()))() are in A, but)(is not. Show that A is in L.

We know that the class L contains all problems requiring only a fixed number of counters/pointers to solve them. The language A requires one counter. Initially, the counter is 0, and, as the head moves to the right along the tape, the counter is incremented by 1 if the symbol read is "(", and it is decremented

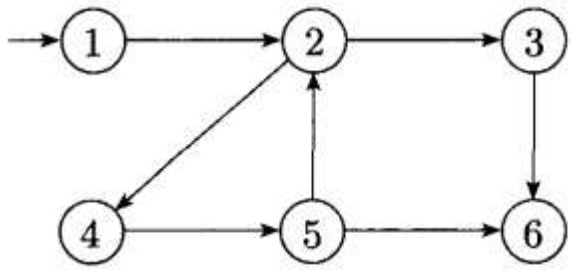
by 1 if the symbol read is ")". If the counter ever becomes negative we reject the string (more closing parentheses then opening ones). If the counter is positive after reading the last symbol in the string, we reject (more opening parentheses then closing ones). Otherwise, we accept.

Question 8.33

Define $CYCLE = \{\langle G \rangle \mid G \text{ is a directed graph that contains a directed cycle}\}$. Show that $CYCLE$ is NL-complete.

Reduce *PATH* to *CYCLE*. The idea behind the reduction is to modify the *PATH* problem instance $\langle G, s, t \rangle$ by adding an edge from t to s in G. If a path exists from s to t in G, a directed cycle will exist in the modified G. However, other cycles may exist in the modified G because they may already be present in G. To handle that problem, first change G so that it contains no cycles. A ***leveled directed graph*** is one where the nodes are divided into groups, $A_1, A_2, \ldots, A_k$, called *levels*, and only edges from one level to the next higher level are permitted. Observe that a leveled graph is acyclic. The *PATH* problem for leveled graphs is still NL-complete, as the following reduction from the unrestricted *PATH* problem shows. Given a graph G with two nodes s and t, and m nodes in total, produce the leveled graph G' whose levels are m copies of G's nodes. Draw an edge from node i at each level to node j in the next level if G contains an edge from i to j. Additionally, draw an edge from node i in each level to node i in the next level. Let s' be the node s in the first level and let t' be the node t in the last level. Graph G contains a path from s to t iff G' contains a path from s' to t'. If you modify G' by adding an edge from t' to s', you obtain a reduction from *PATH* to *CYCLE*. The reduction is computationally simple, and its implementation in logspace is routine. Furthermore, a straightforward procedure shows that $CYCLE \in \text{NL}$. Hence *CYCLE* is NL-complete.

8.3 Game



Player 1 start the game from node 1 and then pass to player 2 this game win the player 2 because he have more nodes than player 1 coz player 1 stuck on 2 but 2 have nodes 3 to 6 and 4 to 5,6.

8.**30**

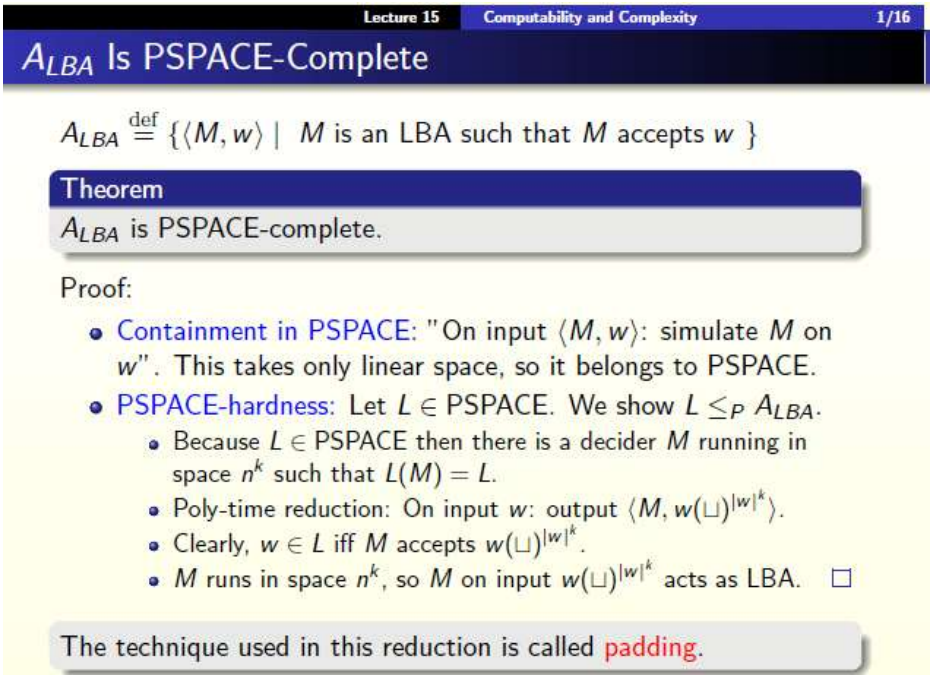
Show that EDFA is NL-complete.

Answer: We show that $N = \overline{E_{DFA}}$ is NL-complete and then use the fact that NL = co-NL to conclude that E_{DFA} is NL-complete.
Part I: N is in NL.
We can non-deterministically guess the string accepted by N and then verify that string is indeed accepted in log-space as all we need to keep track of is current state of N and current position in the string.

Part II: N is NL-complete.
We reduce PATH to N as follows. Given instance $< G(V, E), s, t >$ of PATH, Let k denote maximum outdegree of any node in G. So we construct a DFA $D = \{Q, \Sigma, \delta, q_0, F\}$ as follows. $Q = V \cup \{d\}$, where d is new dead state. $\Sigma = \{a_1, a_2, .., a_k\}$ be arbitrary set of input alphabet. Each outgoing edge from every state of D is labeled with arbitrary symbol from Σ. Note that we always have enough symbols to unquiely label outgoing edges from given state. If any state has less than k outgoing edges, we add transition from that state to dead state for all symbols not used in any transition from that state. Finally we set $q_0 = s$ and $F = \{t\}$. It is easy to see that graph G has a path from s to t if and only if DFA D accepts at least one string.

8.13 **Define ALBAacceptence of linera boundd automata = {(M, w) I M is an LBA that accepts input w}. Show that ALBA is**

PSPACE-complete.



Q1 Show that A is NP-Complete.

Sol:

A language A is called NP-Complete ,

1 if A is NP-Hard

2 And A is in NP.

NP-Hard:- A languages A is NP-Hard, if every language L ϵ NP and $L \leq_p P$.

If both conditions are satisfied, then the language A is in NP-Complete.

Q2 Prove that Double SAT is NP-Complete (by reducing from 3 SAT)

Sol:

DOUBLE-SAT is in NP because the following is a polynomial time verifier for

DOUBLE-SAT.

V is verifier

V = On input << Φ >,c>: where c is certificate

- Test whether c contains two different assignments.
- Test whether both of the two assignments satisfy Φ .
- If both satisfied accept, otherwise reject.

We take 3SAT problem as known NP-Complete problem and reduce to the DSAT. 3SATcontains Φ and Φ is in CNF form and Φ is satisfiable.

We show that $3SAT \leq_p DSAT$. Given a formula Φ and define f(<Φ >) = Φ V Φ', where Φ' is a formula with every literal in Φ complemented(e.g. negation of x is a complement of x). if Φ is satisfiable, then Φ V Φ' has at least two satisfying assignments, one is the same as one satisfying assignment for Φ, and another one is by complementing each

variable's assignment in the previous satisfying assignment. In the reverse direction, if Φ V Φ' has at least two satisfying assignments, then either Φ or Φ' is satisfied. If Φ is satisfied then we are done. If Φ' is satisfied then we can complement the assignment of every variable to get a satisfying assignment for Φ and the time taken by this reduction is polynomial to the number of variables in Φ.

Q3: Prove that SET SPLITTING is NP-Complete.

Sol:

7.27 *SET-SPLITTING* in in NP because a coloring of the elements can be guesses and verified to have the desired properties in polynomial time. We give a polynomial time reduction f from *3SAT* to *SET-SPLITTING* as follows. Given a 3cnf ϕ, we set $S = \{x_1, \overline{x_1}, \ldots, x_m, \overline{x_m}, y\}$. In other words, S contains an element for each literal (two for each variable) and a special element y. For every clause c_i in ϕ, let C_i be a subset of S containing the elements corresponding to the literals in c_i and the special element $y \in S$. We also add subsets C_{x_i} for each literal, containing elements x_i and $\overline{x_i}$. Then $C = \{C_1, \ldots, C_l, C_{x_1}, \ldots, C_{x_l}\}$.

If ϕ is satisfiable, consider a satisfying assignment. If we color all the true literals red, all the false ones blue, and y blue, then every subset C_i of S has at least one red element (because c_i of ϕ is "turned on" by some literal) and it also contains one blue element (y). In addition, every subset C_{x_i} has exactly one element red and one blue, so that the system $\langle S, C\rangle \in$ *SET-SPLITTING*.

If $\langle S, C\rangle \in$ *SET-SPLITTING*, then look at the coloring for S. If we set the literals to true which are colored differently from y, and those to false which are the same color as y we obtain a satisfying assignment to ϕ.

Q 4: Prove Multi SAT is NP-Complete

Sol:

On input ϕ a non-deterministic TM can guess two assignments and accept both if both satisfy ϕ. Thus Double-SAT is NP. DOUBLE-SAT is mapping reducible to MULT_SAT. MULT-SAT ≤double_SAT. The following Turing machine R computes the assignments in polynomial time reduction.

R= "On input <ϕ>, a Boolean formula with variables x1,x2,. . ., xm

Let ϕ' be the ϕ □ (x v $\overline{x}$), where x is a new variable.

Output <ϕ'>"

If ϕ ϵ Double-SAT, ϕ' has at least two satisfying assignments because two satisfying conditions from the original documents of ϕ by changing the values of x.

Q5: Show that directed hamiltonian cycle is NP-Complete

Sol:

23.0.0.4 Directed Hamiltonian Cycle

Input Given a directed graph $G = (V, E)$ with n vertices

Goal Does G have a *Hamiltonian cycle*?

- A Hamiltonian cycle is a cycle in the graph that visits every vertex in G exactly once

23.0.0.5 Directed Hamiltonian Cycle is NP-complete

- Directed Hamiltonian Cycle is in NP
  - *Certificate:* Sequence of vertices
  - *Certifier:* Check if every vertex (except the first) appears exactly once, and that consecutive vertices are connected by a directed edge
- *Hardness:* We will show 3-SAT $\leq_P$ Directed Hamiltonian Cycle

Q6: Show that half clique is NP-Complete

Sol:

7.22 We give a polynomial time mapping reduction from *CLIQUE* to *HALF-CLIQUE*. The input to the reduction is a pair $\langle G, k \rangle$ and the reduction produces the graph $\langle H \rangle$ as output where H is as follows. If G has m nodes and $k = m/2$ then $H = G$. If $k < m/2$, then H is the graph obtained from G by adding j nodes, each connected to every one of the original nodes and to each other, where $j = m - 2k$. Thus H has $m + j = 2m - 2k$ nodes. Observe that G has a k-clique iff H has a clique of size $k + j = m - k$ and so $\langle G, k \rangle \in$ *CLIQUE* iff $\langle H \rangle \in$ *HALF-CLIQUE*. If $k > 2m$, then H is the graph obtained by adding j nodes to G without any additional edges, where $j = 2k - m$. Thus H has $m + j = 2k$ nodes, and so G has a k-clique iff H has a clique of size k. Therefore $\langle G, k_k \rangle \in$ *CLIQUE* iff $\langle H \rangle \in$ *HALF-CLIQUE*. We also need to show *HALF-CLIQUE* $\in$ NP. The certificate is simply the clique.

Q7: Let CNF-k = {<ϕ> | ϕ is satisfiable CNF-formula where each variable appears in at most k places}. Show that CNF3 is NP-Complete

Sol:

(b) We leave it to the reader to show that CNF_3 is in NP. We show that $3SAT \leq_P \text{CNF}_3$.

First an example: $\varphi = (p \vee q) \wedge (p \vee r) \wedge (\neg p \vee s) \wedge (p \vee t)$. This formula is not in 3nmf, but the general case will be treated below. This formula is replaced by

$$\psi = (p \vee q) \wedge (\neg p \vee p_1) \wedge (p_1 \vee r) \wedge (\neg p_1 \vee p_2) \wedge (\neg p_2 \vee s) \wedge (\neg p_2 \vee p_3) \wedge (p_3 \vee t) \wedge (\neg p_3 \vee p).$$

Note that the $(\neg p \vee p_1), (\neg p_1 \vee p_2), (\neg p_2 \vee p_3), (\neg p_3 \vee p)$ are expressing that $p \to p_1, p_1 \to p_2, p_2 \to p_3, p_3 \to p$, that is, the p, p_1, p_2, p_3 are all equivalent. Hence in this formula $(p_1 \vee r)$ expresses the same as $(p \vee r)$, etc. And thus φ is satisfiable when ψ is, but in ψ every variable occurs at most 3 times.

The general case. We define a reduction f from $3SAT$ to CNF_3 as follows. $f(\langle \varphi \rangle)$ is the formula that is the result of the following procedure:

1. pick the first propositional variable (reading from left to right) in φ that occurs more than 3 times in the formula, suppose it is p, and suppose it occurs at n places: $(x_1 \vee A_1), \ldots, (x_n \vee A_n)$, where the x_i are p or $\neg p$. If no variable occurs more than 3 times, output φ.

2. Choose fresh variables $p_1, \ldots, p_n$, remove the conjuncts $(x_i \vee A_i)$ from the formula and add as a conjunct the formula

$$(p_1 \vee A_1) \wedge (\neg p_1 \vee p_2) \wedge (p_2 \vee A_2) \wedge (\neg p_2 \vee p_3) \ldots (p_n \vee A_n) \wedge (\neg p_n \vee p_1).$$

Q8: A subset of nodes of a graph is a dominating set if every other node of G is adjacent to some node in the subset. Let DOMINATING-SET={<G, k> | G has a dominating set with k nodes}. Show that it is NP-Complete by giving a reduction from VERTEX-COVER.

Sol:

Clearly Dominating Set is in **NP**. Given a dominating set, one can verify in polynomial time if that is a dominating set. This can be done by taking each vertex and checking if it is either in the given set or one of its edges travel into the set.

To show that is **NP**-complete, first of all notice that a dominating set has to include all isolated vertices (those which have no edges from them). So let us assume that our graph does not have any isolated vertices. We will show that Dominating Set is **NP**-complete using a reduction from Vertex Cover. Given a graph G, we will construct a graph G' as follows. G' has all edges and vertices of G. Also, for every edge $\{u, v\} \in G$, we add intermediate node on a parallel path in G'. Keeping $\{u, v\}$ intact in G', we add vertex w and edges $\{u, w\}$ and $\{w, v\}$ in G'. Now we will show that G has a vertex cover of size k **if and only if** G' has a dominating set of the same size.

If S is a vertex cover in G, we will show that S is a dominating set for G'. S is a vertex cover, this means that every edge in G has atleast one of its end points in S. Consider $v \in G'$. If v is an original node in G, then either $v \in S$ or there must be some edge connecting v to some other vertex u. Since S is a vertex cover, is $v \notin S$, then u must be in S, and hence there is an adjacent vertex of v in S. So v is covered by some element in S. However, if w is an additional node in G', then w has two adjacent vertices $u, v \in G$ and using the above argument at least one of them is in S. So the additional nodes are also covered by S. So if G has a vertex cover, then G' has a dominating set of at most the same size (in fact the same set itself would do).

9. Let TRUE-SAT; given Boolean expression E that is true when all the variables are made true, is there some other truth assignment besides all true that make E true. Show that TURE-SAT is NP-Complete by reducing SAT to it.

Sol:

To show TRUE-SAT is NP-complete, we reduce SAT to it. Suppose we are given an expression E with variables x1, x2, ..., xn. Convert E to E0 as follows:
1. First, test if E is true when all variables are true. If so, we know is E is satis_able, and so set E0 = E + y where y is a variable not already in E so that E0 will be accepted by TRUE-SAT.
2. Otherwise, let E0 = E + x1x2:::xn, a polynomial-time reduction. Note that E0 will always be true when all variables are true. If E is in SAT then it is satis_ed by some truth assignment other than all-true, thus E0 is in TRUE-SAT. Conversely, if E0 is in TRUE-SAT, then since x1x2:::xn is true only for the all-true assignment then E must be satis_able.
Thus we have a polynomial-time reduction from SAT to TRUE-SAT, therefore TRUE-SAT is NP-complete.

10. Let NEAR TAUT; given Boolean expression E all the variables are made f, is there some other truth assignment besides all f that make E true. Show that the NEAR TAUT is NP-Complete by reducing SAT to it.
Sol:

11. Let NEAR-TAUT: E is a Boolean expression having when at most one true makes it false. Show that complement of NEAT-TAUT is in NP-Complete using reduction it to SAT.

Sol:

NEAR-TAUT is in co-NP. The complement of NEAR-TAUT consists of all inputs that are not well-formed expressions and inputs that are well-formed expressions such that there exist at least two non-satisfying assignments. To decide the complement in polynomial time we need only guess and check two assignments that do not satisfy the given expression. We will show that the complement is NP-complete, so it is unlikely that NEAR-TAUT is in NP.
To show the complement of NEAR-TAUT is NP-complete, we reduce SAT to it. Given an expression E, we convert it to E0 as follows:
E0 = :(E ^ (y _ :y))
If E is in SAT, then E0 must have at least two non-satisfying assignments, since we can take some satisfying assignment of E and apply it to E0, setting y to either true or false and E0 will not be satis_ed. If E0 is in the complement of NEAR-TAUT then it must have at least two non-satisfying assignments, thus E must have at least one satisfying assignment in order for the expression E ^ (y _ :y) to ever be true.
4

12. Let G represent and undirected graph. Also let LPATH = {<G,a,b,k>|G contains a simple path of length k for a to b>}. Show that LPATH is NP complete. You may assume that NP completeness of UHAMPTH the HAMPATH problem of undirected graph.

Let LONG-PATH = {<G, a, b, k> | G is an undirected graph and contains a simple path of length at least k from a to b}

G is graph is graph, a and b are vertex and k is a number of edges.

V is verifier.

A is start vertex and b is a end vertex.

V=on input <<G,a,b,k>,c> where G is a graph, a and b are vertex.

- Longest path contain at least k edges.
- Starts from vertex a, ends with vertex b.
- C test whether G contains all the nodes from a to b.
- If both vertex (a,b) does not pass, reject otherwise accept.(visit exactly once).

Reduce the known NP-Complete problem to unknown NP-Complete problem.

Suppose we take Hamiltonian path as know NP-Complete problem and reduce the Hamiltonian path to longest path. If we reduce longest path problem from Hamiltonian path its means longest path is also in NP complete.

Given an instance of Hamiltonian path on the graph G=(V,E) where V is vertex and E is a edges. And we create a instance of longest path G' that contains a simple path of length at least k. We utilize precisely the same graph, e.g. G'=G and we set K=|V|-1.At that point there exist a clear method for length k in G' if and if Contain Hamiltonian path. This method shows that longest path is in NP Complete.

13. A directed Graph is STRONGLY-CONNECTED of every two nodes are connected by a directed graph in each direction. Let STRONGLY-CONNECTED = {<G>|G is strongly connected graph}. Show that STRONGLY-CONNECTED is NP-Complete.

14. Let A = {<M, x, # $_t$ >|NTM, M accepts input x within t steps on at least one branch}. Show that A is NP-Complete.

Here A= U

SOLUTION: Given any NP language L, we have an NDTM M_L such that $\forall x \in L$, M_L accepts x on at least one branch in at most $p_L(|x|)$ steps, where $p_L()$ is a fixed polynomial depending on the machine. Also, M_L does not accept any $x \notin L$. Then, given x, we create $y = \langle M_L, x, \#^{p_L(|x|)} \rangle$ in polynomial time. By the previous argument, $x \in L$ iff $y \in U$. Thus, U is NP-hard.

To show that U is also in NP, we can create an NDTM M_U, which given an input $u = \langle M, x, \#^t \rangle$, simulates M on x for t steps. M_U nondeterministically guesses all the branches of M and accepts u iff M accepts u. Since the input has length at least t and we simulate M for at most t steps, the running time is polynomial in the length of the input (note this is the reason we need t in unary). It is easy to see that M_U accepts exactly the language U, thus proving $U \in NP$. Hence, U is NP-complete.

15. Prove cycle- length problem is NL-Complete.

8.33 Reduce *PATH* to *CYCLE*. The idea behind the reduction is to modify the *PATH* problem instance $\langle G, s, t \rangle$ by adding an edge from t to s in G. If a path exists from s to t in G, a directed cycle will exist in the modified G. However, other cycles may exist in the modified G because they may already be present in G. To handle that problem, first change G so that it contains no cycles. A ***leveled directed graph*** is one where the nodes are divided into groups, $A_1, A_2, \ldots, A_k$, called *levels*, and only edges from one level to the next higher level are permitted. Observe that a leveled graph is acyclic. The *PATH* problem for leveled graphs is still NL-complete, as the following reduction from the unrestricted *PATH* problem shows. Given a graph G with two nodes s and t, and m nodes in total, produce the leveled graph G' whose levels are m copies of G's nodes. Draw an edge from node i at each level to node j in the next level if G contains an edge from i to j. Additionally, draw an edge from node i in each level to node i in the next level. Let s' be the node s in the first level and let t' be the node t in the last level. Graph G contains a path from s to t iff G' contains a path from s' to t'. If you modify G' by adding an edge from t' to s', you obtain a reduction from *PATH* to *CYCLE*. The reduction is computationally simple, and its implementation in logspace is routine. Furthermore, a straightforward procedure shows that $CYCLE \in$ NL. Hence *CYCLE* is NL-complete.

16. Let ADD = {<x, y, z> | x, y, z >0 are binary integers and x + y =z}.
Show that ADD ϵ LSPACE.
Sol:

17. Let PAL-ADD = {<x, y> | x, y >0 are binary integers where x + y is an integer whose binary representation is palindrome}. Show that PAL-ADD ϵ LSPACE.

1) Hamiltonian cycle asks if there is a simple cycle that visits every node once in an undirected graph G(V,E). Prove that directed-HC directed Hamiltonian cycle in np-complete?. Where DHC we are given as input a directed graph G'(V,E) and we ask if there is a simple directed cycle that visits every node once.

2) let φ be 3cnf, as ≠- assignment to variables of $\varphi$is one ehere each clause contains 2 literals with unequal truth values. In other words an ≠- assignment satisfies φwithout 3 true literals in any clause.

let ≠SAT be a collection of 3cnf formula that have an ≠-assignmnt . show that we obtain a pol-time reduction from 3SAT to ≠SAT by replacing each clause ci(y1 v y2 v y3) with 2 clauses (y1 v y2 v zi) and (zi' v y3 v b). zi new variable for each clause, ci and b is a single additional new variable.

3) let EQ_{DFA} = {<N, N'>| N, N' are NFAs with same alphabets and L(N)= L(N')}. Show that EQ_{DFA} =PSPACE

4) Define A_{LBA}acceptence of linera boundd automata = {(M, w) I M is an LBA that accepts input w}. Show that A_{LBA} is
PSPACE-complete.

CS 482 Summer 2004

Proving that Problems are NP-Complete

To prove that a problem X is NP-Complete, you need to show that it is both in NP and that it is NP-Hard. Steps 2 through 5 seek to accomplish the latter.

Step 1: Show that X is in NP. We want to argue that there is a polytime verifier for X. In other words, for any yes instance of X, there exists a certificate that the verifier will accept, and for any no instance of X, there is no certificate that the verifier will accept. Both the size of the certificate and the running time of the verifier must be polynomial. Often this step is very brief, but necessary.

Step 2: Pick a known NP-Complete problem. State what problem Y you are reducing to X. You need to show that $Y \leq_p X$. You may use any problem Y which we have proved in class to be NP-Complete, as well as any problem you have proved to be NP-Complete on the homework assignments. Some problems will be far easier to use than others in your proof.

Step 3: Construct an algorithm to solve Y given an algorithm to solve X. You need to show that any instance of Y can be solved using a polynomial number of operations, and a polynomial number of calls to a black box that can solve X. It is very easy to get mixed up and instead prove that $X \leq_p Y$. Unfortunately, this is not what you want to show (we already know that Y is NP-Complete).

Step 4: Prove the correctness of your algorithm. This has 2 parts: You want to show that given a yes instance of Y your algorithm returns "yes". Furthermore, you want to show that given a no instance of Y, your algorithm returns "no". It is always trivial to come up with an algorithm that satisfies just one of these two conditions. We want something that satisfies both.

Step 5: Polytime and wrap-up. Finally, you need to conclude that since your algorithm runs in polynomial time, $Y \leq_p X$. Since Y is NP-Complete, X is NP-Hard, and since we also have shown that X is in NP, X is in fact NP-Complete.

An Example: INDEPENDENT SET (IS) is NP-Complete

First, IS is in NP, since given any set S we can check in polytime that S is independent and that $|S| = k$. So for a yes instance, we simply use an independent set of size k.

And for a no instance, it is clear that no such set exists. Therefore IS is in NP. Now we will show that IS is NP-Hard via reduction to 3-SAT.

Suppose we have an instance $F = C_1 \wedge C_2 \wedge \ldots \wedge C_m$, where C_i is the disjunction of 3 variables, drawn from $x_1, x_2, \ldots, x_n$ and their negations, $\overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}$. We create the graph G as follows:

For each variable in each clause, create a node, which we will label with the name of the variable. Therefore there may be multiple nodes with the label x_i or $\overline{x_i}$, if these variables appear in multiple clauses.For each clause, add an edge between the three nodes corresponding the variables from that clause. We will call these three nodes and three edges a "clause gadget". Finally, for all i, add an edge between every pair of nodes with one labeled x_i and the other labeled $\overline{x_i}$. Now use our black box for IS to determine whether or not there is an independent set of size m in G. If there is, return yes. Otherwise, return no.

We must now show that this algorithm correctly determines whether or not F has a satisfying assignment. Suppose F has a satisfying assignment A. Then at least one variable in each clause is satisfied by A. Define S to be a set of nodes in G found by selecting one of the satisfied variable nodes in each clause gadget. Since we picked one node for each clause, there are clearly m nodes in S. Furthermore, since we only select a single node for each gadget, independence is not violated within any clause gadget. Finally, independence is not violated between clauses, since the only edges between clauses go between nodes with labels x_i and $\overline{x_i}$, and A can not have satisfied both of these, so we never would have selected both. Therefore S is an independent set of size k, so our algorithm will return yes.

Now suppose that our algorithm returns yes. We now need to show that there is a satisfying assignment A. Since our algorithm returned yes, there must be an independent set S of size m on G. Since S is independent, at most one node in each clause gadget must be used by S. But in fact, since there are exactly m clause gadgets, S must contain exactly one node from each clause gadget. Since S is independent, no pair of nodes x_i and $\overline{x_i}$ are ever both selected for S. Consider the following assignment. Set $A(x_i) = T$ if $x_i \in S$ and set $A(x_i) = F$ otherwise. Observe that this is a truth assignment, since all variables are assigned either T or F, but not both. Furthermore, A satisfies F, since by our construction, A satisfies each clause.

Therefore we have shown that 3-SAT $\leq_p$ IS. Thus IS is NP-Hard, and since we have shown IS to be in NP, IS is NP-Complete.

Computability and Complexity

Lecture 15

Test
PSPACE-completeness
Summary and Exam Info

given by Jiri Srba

PSPACE-Completeness

Definition (PSPACE-Completeness)

A language B is PSPACE-complete iff

1. $B \in$ PSPACE (containment in PSPACE), and
2. for every $A \in$ PSPACE we have $A \leq_P B$ (PSPACE-hardness).

Theorem

If B is PSPACE-complete, $B \leq_P C$, and $C \in$ PSPACE, then C is PSPACE-complete.

Proof: Because $\leq_P$ is transitive.

A_{LBA} Is PSPACE-Complete

$A_{LBA} \stackrel{\text{def}}{=} \{\langle M, w\rangle \mid M \text{ is an LBA such that } M \text{ accepts } w\}$

Theorem

A_{LBA} is PSPACE-complete.

Proof:

- Containment in PSPACE: "On input $\langle M, w\rangle$: simulate M on w". This takes only linear space, so it belongs to PSPACE.
- PSPACE-hardness: Let $L \in$ PSPACE. We show $L \leq_P A_{LBA}$.
  - Because $L \in$ PSPACE then there is a decider M running in space n^k such that $L(M) = L$.
  - Poly-time reduction: On input w: output $\langle M, w(\sqcup)^{|w|^k}\rangle$.
  - Clearly, $w \in L$ iff M accepts $w(\sqcup)^{|w|^k}$.
  - M runs in space n^k, so M on input $w(\sqcup)^{|w|^k}$ acts as LBA. □

The technique used in this reduction is called padding.

TQBF Is PSPACE-Complete

Quantified Boolean formula (QBF):

$$\psi \stackrel{\text{def}}{=} \forall x_1 \exists x_2 \forall x_3 \exists x_4 \ldots \forall x_{k-1} \exists x_k . \phi$$

where ϕ is a Boolean formula over the variables $x_1, \ldots, x_k$.

Any given QBF ψ is either true or false.

$$TQBF \stackrel{\text{def}}{=} \{\langle \psi \rangle \mid \psi \text{ is a true QBF}\}$$

Theorem

TQBF is PSPACE-complete.

Proof: See the book (not part of the syllabus).

ALL_{NFA} Is PSPACE-Complete

Problem: "Does a given NFA accept all strings from Σ^*?"

Theorem

ALL_{NFA} is PSPACE-complete.

Proof: Not part of the syllabus.

The Course Is Over

The course is now over.

Thank you for your participation!

Decision Problems

Decision Problem

"Given an instance of the problem, is it a positive instance or a negative instance?"

Language Formulation

$$L = \{\langle P \rangle \mid P \text{ is a positive instance of the problem } \}$$

Questions:

- Computability Theory: Is a given problem algorithmically solvable? (Is the corresponding language decidable?)
- Complexity Theory: How difficult is to solve a given problem? (What is the time/space complexity of the corresponding language?)

Turing Machine — A Model of a Computer

Church-Turing Thesis

"The Turing machine model captures exactly the informal notion of algorithm."

Polynomial Time Equivalence of Deterministic Models (Thesis)

"All reasonable deterministic models of computation are polynomial time equivalent to deterministic single-tape Turing machine."

Variants of TMs and Time vs. Space Complexity

Let $t(n)$ be a function s.t. $t(n) \geq n$.

Theorem (Multi-Tape TM)

Every k-tape TM M has an equivalent 1-tape TM M'.

If M is running in time $t(n)$ then M' is running in time $O(t^2(n))$.

Theorem (Nondeterministic TM)

Every nondeterministic TM M has an equivalent determin. TM M'.

If M is running in time $t(n)$ then M' is running in time $2^{O(t(n))}$.

If M is running in space $t(n)$ then M' is running in space $O(t^2(n))$.

Theorem (Time vs. Space Complexity)

Every TM running in time $t(n)$ is running in $O(t(n))$ space.

Every TM running in space $t(n)$ is running in time $2^{O(t(n))}$.

Classes of Languages

- P: class of all languages decidable in polynomial time on deterministic TMs.
- NP: class of all languages decidable in polynomial time on nondeterministic TMs.
- co-NP: class of all languages which complements belong to NP.
- PSPACE: class of all languages decidable in polynomial space on (deterministic or nondeterministic) TM.
- EXPTIME: class of all languages decidable in exponential time on deterministic TMs.
- Decidable: class of all languages that are recognized by TMs which are deciders.
- Recognizable: class of all languages that are recognized by TMs.
- Co-recognizable: class of all languages which complements are recognized by TMs.

Crucial Results

Theorem (Turing — Undecidability of A_{TM})

The acceptance problem of a Turing machine is undecidable.

Theorem (Cook-Levin — NP-Completeness of SAT)

The satisfiability problem for Boolean formulae is NP-complete.

Other undecidable or computationally hard problems were derived using reductions:

- for undecidability we used mapping reductions, and
- for NP-hardness we used polynomial time reductions.

Languages Studied in Computability Theory

- Decidable:
  A_{DFA}, A_{NFA}, E_{NFA}, EQ_{NFA}, A_{CFG}, E_{CFG}, A_{LBA}.
- Recognizable but not decidable:
  A_{TM}, $HALT_{TM}$, $\overline{E_{TM}}$, $\overline{E_{LBA}}$, $\overline{ALL_{CFG}}$, $\overline{EQ_{CFG}}$, PCP, $BPCP$.
- Co-recognizable but not decidable:
  Complements of all languages from the above category.
- Neither recognizable nor co-recognizable:
  EQ_{TM}, $REGULAR_{TM}$, $TOTAL_{TM}$.

Languages Studied in Complexity Theory

- In P:
  PATH, *RELPRIME*, any context-free language.
- NP-complete:
  SAT, *CNF-SAT*, *3SAT*, *HAMPATH*, *UHAMPATH*, *CLIQUE*, *SUBSET-SUM*, *VERTEX-COVER*.
- PSPACE-complete:
  A_{LBA}, *TQBF*, ALL_{NFA}.

Closure Properties

Class of Languages	$\cap$	$\cup$	$\circ$	$*$	$-$
decidable	YES	YES	YES	YES	YES
recognizable	YES	YES	YES	YES	NO
P	YES	YES	YES	YES	YES
NP	YES	YES	YES	YES	???
PSPACE	YES	YES	YES	YES	YES
EXPTIME	YES	YES	YES	YES	YES

- Intersection: Run two Turing machines in sequence.
- Union: Run two Turing machines in sequence, in parallel, or nondeterministically choose one.
- Concatenation: Try all possible splitting points, or guess the point nondeterministically.
- Kleene star: Try all possible splittings (exponentially many, or use dynamic programming), or guess them.
- Complement: Swap accept and reject state (works only for deterministic TMs that never loop).

Exam Information

- Individual, written, graded exam.
- On the exam day bring only a pen (or rather two).
- No notes, no books, no slides, no calculator, ... :-(
- No empty sheets of paper (they will be provided to you).
- Read very carefully the tasks.
- Answer all you are asked to do, but don't do anything more!
- Work in the order from the easiest to the most difficult task.
- Double check your solutions.
- Your hand-writing must be readable (make drafts on the provided paper, do not hand them in).
- Return only the sheets of paper you want us to consider for the exam. On each sheet write your full name and question number you solve there.

How to Prepare for the Exam

Assumption

You already read the suggested reading in the book and solved all compulsory exercises during the semester.
If not: do it as soon as possible!!!

- Go lecture by lecture through the slides. If something is unclear, read the corresponding text in the book.
- Go through allcompulsory exercises, recall the solutions on your own and then compare them with the provided ones.
- Go through the checklist on Course Info page, make sure you
  - can precisely write down all the main definitions,
  - formulate in written and understand the main theorems
  - recall definitions of the languages (write them down) and refresh into which class they belong.
- Try to solve the tests 1 to 4 one more time.

You are now ready for the exam, good luck and all the best!

CMSC 451: SAT, Coloring, Hamiltonian Cycle, TSP

Slides By: Carl Kingsford

Department of Computer Science
University of Maryland, College Park

Based on Sects. 8.2, 8.7, 8.5 of *Algorithm Design* by Kleinberg & Tardos.

Boolean Formulas

Boolean Formulas:

Variables: x_1, x_2, x_3 (can be either **true** or **false**)

Terms: $t_1, t_2, \ldots, t_\ell$: t_j is either x_i or $\bar{x}_i$
(meaning either x_i or **not** x_i).

Clauses: $t_1 \vee t_2 \vee \cdots \vee t_\ell$ ($\vee$ stands for "OR")
A clause is **true** if any term in it is **true**.

Example 1: $(x_1 \vee \bar{x_2}), (\bar{x_1} \vee \bar{x_3}), (x_2 \vee \bar{v_3})$

Example 2: $(x_1 \vee x_2 \vee \bar{x_3}), (\bar{x_2} \vee x_1)$

Boolean Formulas

Def. A truth assignment is a choice of **true** or **false** for each variable, ie, a function $v : X \longrightarrow \{\textbf{true}, \textbf{false}\}$.

Def. A CNF formula is a conjunction of clauses:

$$C_1 \wedge C_2, \wedge \cdots \wedge C_k$$

Example: $(x_1 \vee \bar{x_2}) \wedge (\bar{x_1} \vee \bar{x_3}) \wedge (x_2 \vee \bar{v_3})$

Def. A truth assignment is a satisfying assignment for such a formula if it makes every clause **true**.

SAT and 3-SAT

Satisfiability (SAT)

Given a set of clauses $C_1, \ldots, C_k$ over variables $X = \{x_1, \ldots, x_n\}$ is there a satisfying assignment?

Satisfiability (3-SAT)

Given a set of clauses $C_1, \ldots, C_k$, each of length 3, over variables $X = \{x_1, \ldots, x_n\}$ is there a satisfying assignment?

Cook-Levin Theorem

Theorem (Cook-Levin)

3-SAT is NP-complete.

Proven in early 1970s by Cook. Slightly different proof by Levin independently.

Idea of the proof: encode the workings of a Nondeterministic Turing machine for an instance I of problem $X \in \mathbf{NP}$ as a SAT formula so that the formula is satisfiable if and only if the nondeterministic Turing machine would accept instance I.

We won't have time to prove this, but it gives us our first hard problem.

Reducing 3-SAT to Independent Set

Thm. 3-SAT $\leq_P$ Independent Set

Proof. Suppose we have an algorithm to solve Independent Set, how can we use it to solve 3-SAT?

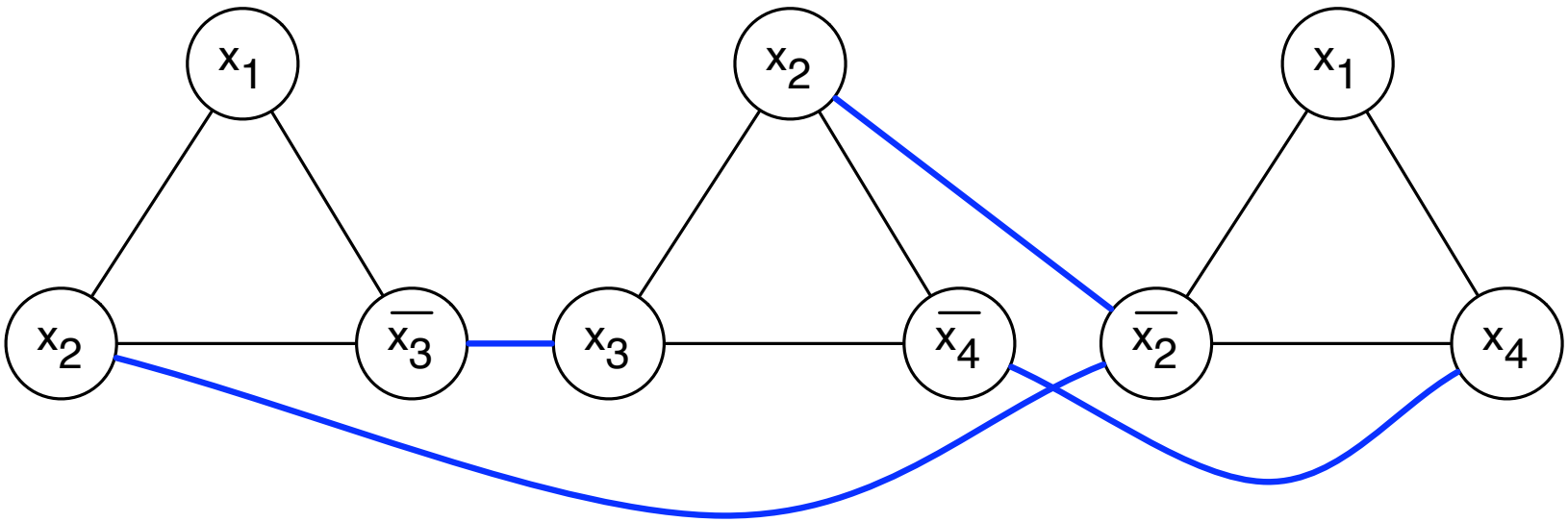
To solve 3-SAT:

- you have to choose a term from each clause to set to **true**,
- but you can't set both x_i and $\bar{x}_i$ to **true**.

How do we do the reduction?

3-SAT $\leq_P$ Independent Set

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



Proof

Theorem

This graph has an independent set of size k iff the formula is satisfiable.

Proof. $\implies$ If the formula is satisfiable, there is at least one true literal in each clause. Let S be a set of one such true literal from each clause. $|S| = k$ and no two nodes in S are connected by an edge.

$\impliedby$ If the graph has an independent set S of size k, we know that it has one node from each "clause triangle." Set those terms to **true**. This is possible because no 2 are negations of each other. $\square$

Graph Coloring

Graph Coloring Problem

Graph Coloring Problem

Given a graph G, can you color the nodes with $\leq k$ colors such that the endpoints of every edge are colored differently?

Notation: A k-coloring is a function $f : V \to \{1, \ldots, k\}$ such that for every edge $\{u, v\}$ we have $f(u) \neq f(v)$.

If such a function exists for a given graph G, then G is k-colorable.

Special case of $k = 2$

How can we test if a graph has a 2-coloring?

Special case of $k = 2$

How can we test if a graph has a 2-coloring?

Check if the graph is bipartite.

Unfortunately, for $k \geq 3$, the problem is NP-complete.

Theorem

3-Coloring is NP-complete.

Graph Coloring is NP-complete

3-Coloring $\in$ **NP**: A valid coloring gives a certificate.

We will show that:

$$\text{3-SAT} \leq_P \text{3-Coloring}$$

Let $x_1, \ldots, x_n, C_1, \ldots, C_k$ be an instance of 3-SAT.

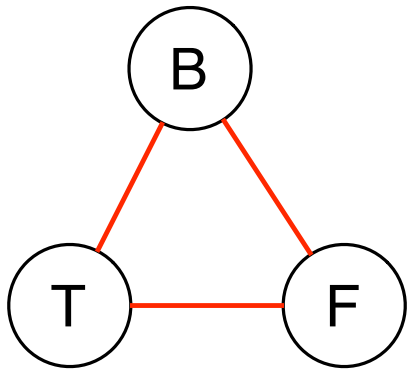
We show how to use 3-Coloring to solve it.

Reduction from 3-SAT

We construct a graph G that will be 3-colorable iff the 3-SAT instance is satisfiable.

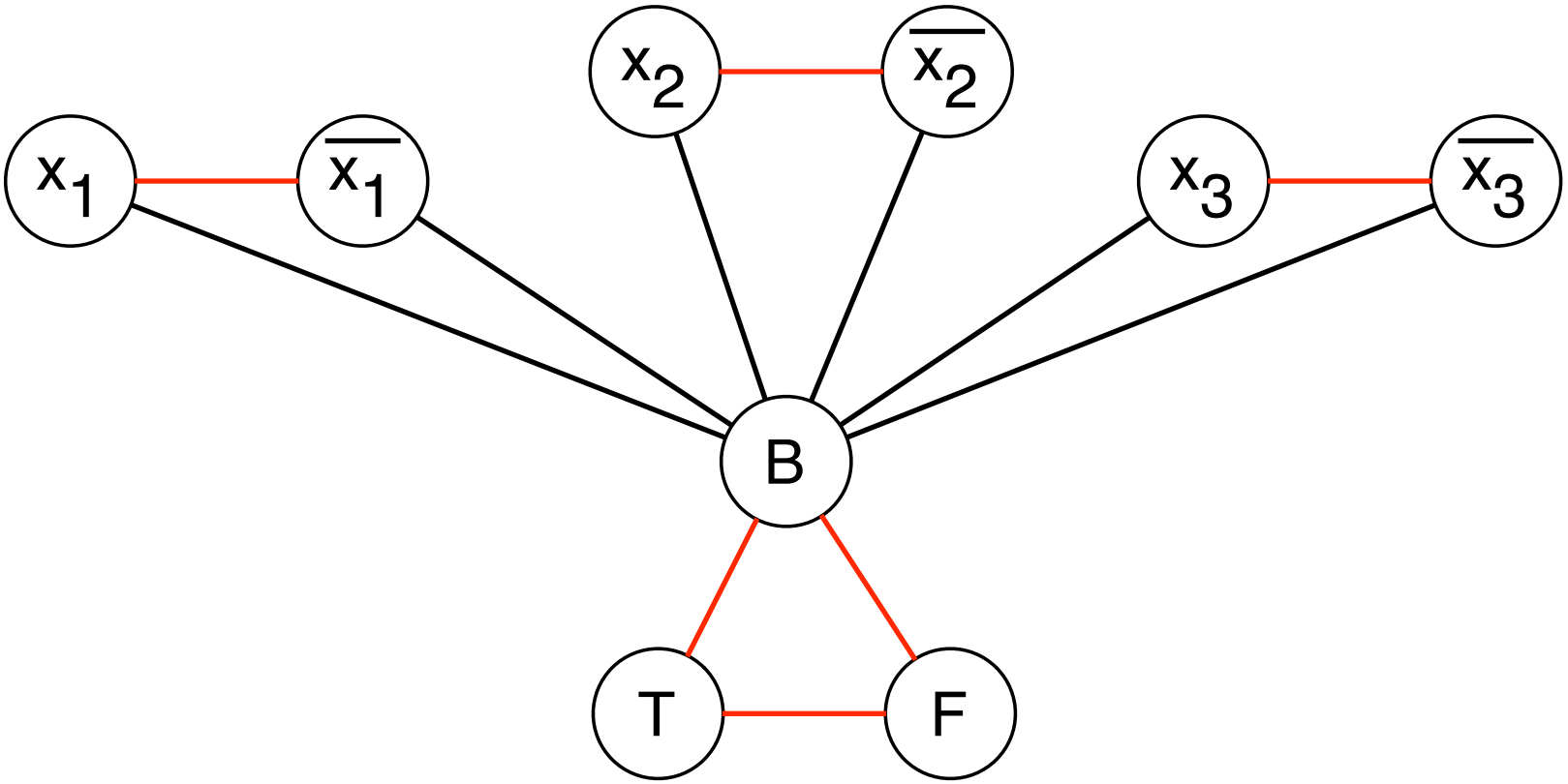
For every variable x_i, create 2 nodes in G, one for x_i and one for $\bar{x}_i$. Connect these nodes by an edge:

Create 3 *special nodes* T, F, and B, joined in a triangle:



Connecting them up

Connect every variable node to B:



Properties

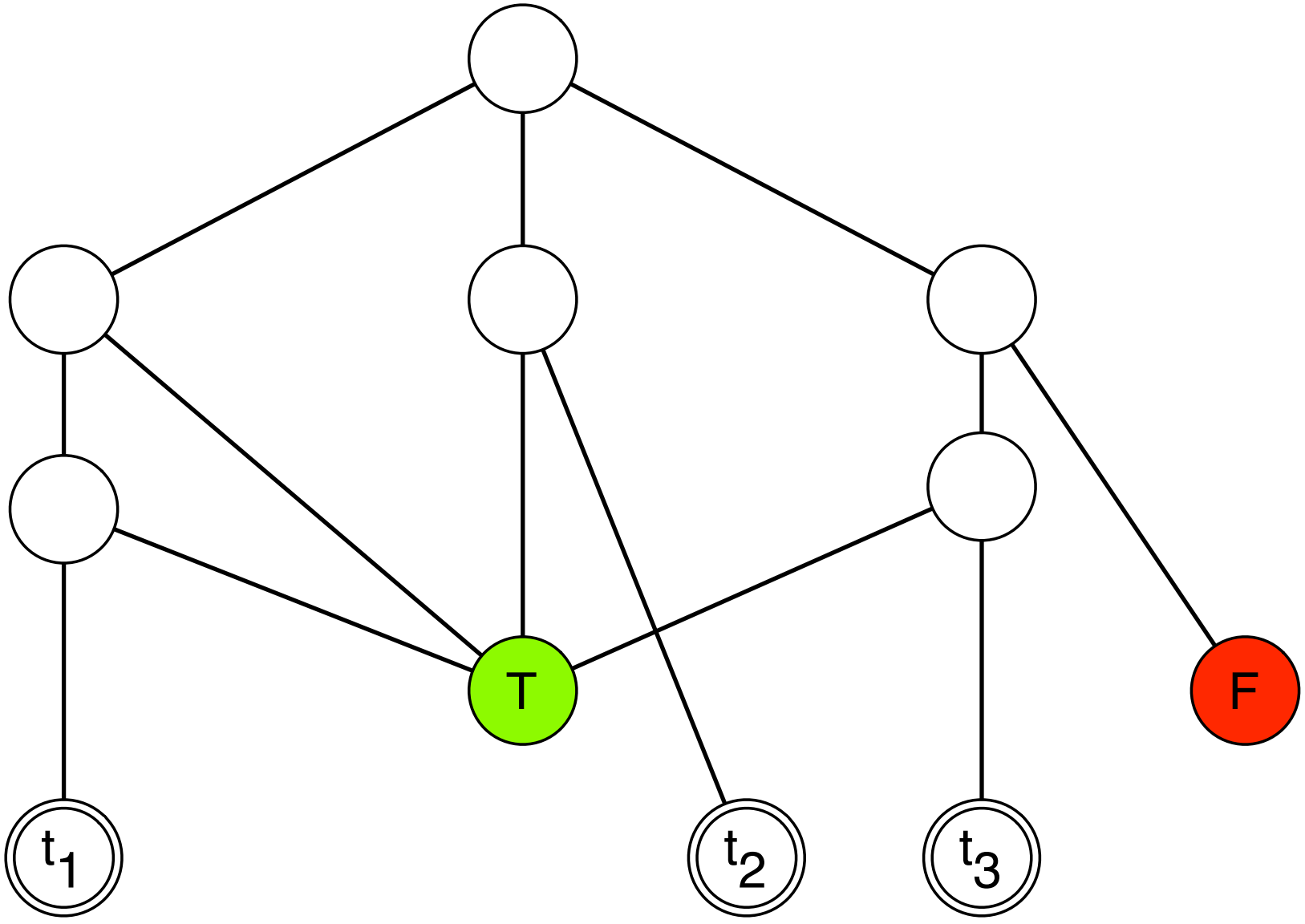
Properties:

- Each of x_i and $\bar{x}_i$ must get different colors
- Each must be different than the color of B.
- B, T, and F must get different colors.

Hence, any 3-coloring of this graph defines a valid truth assignment!

Still have to constrain the truth assignments to satisfy the given clauses, however.

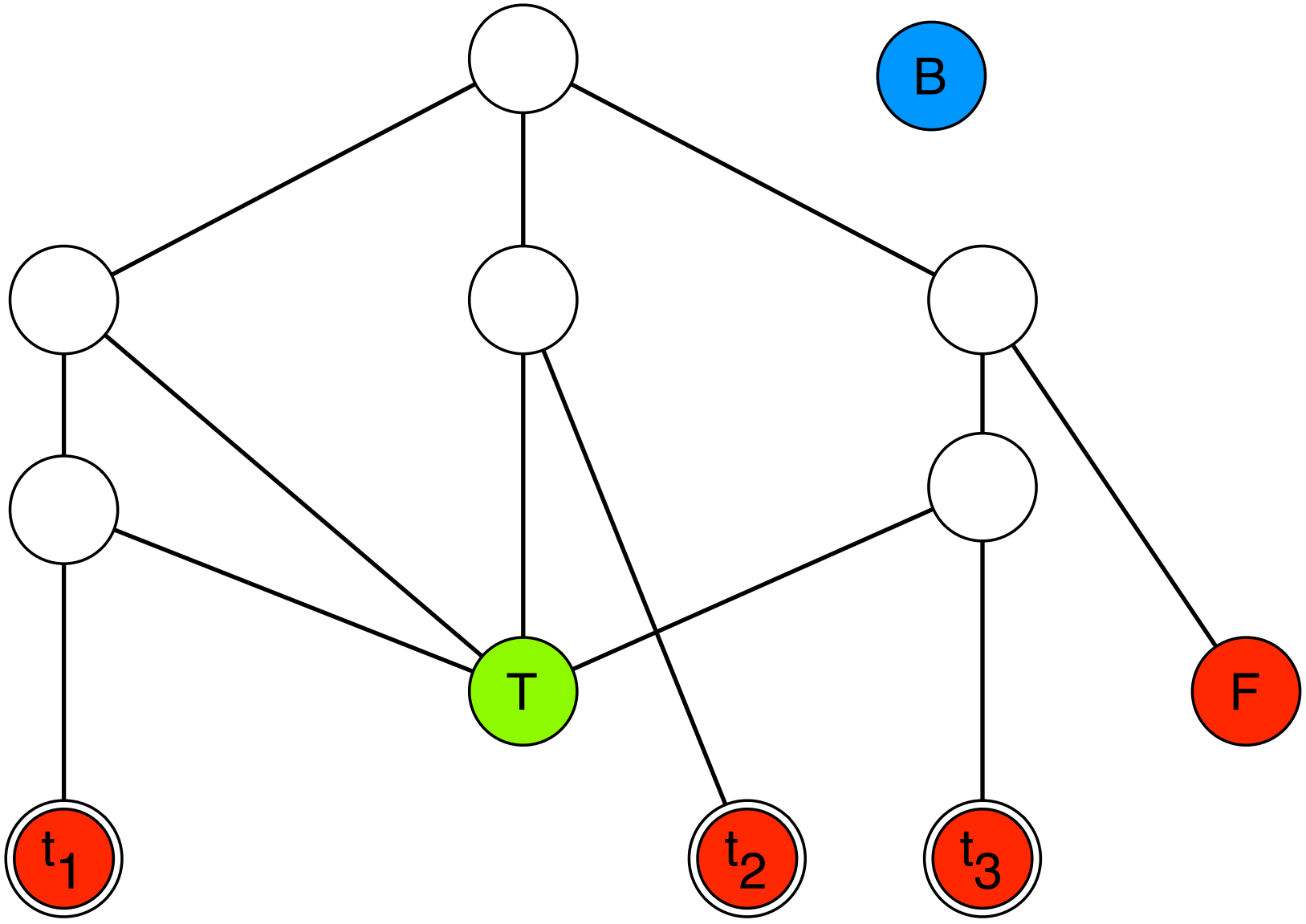
Connect Clause (t_1, t_2, t_3) up like this:



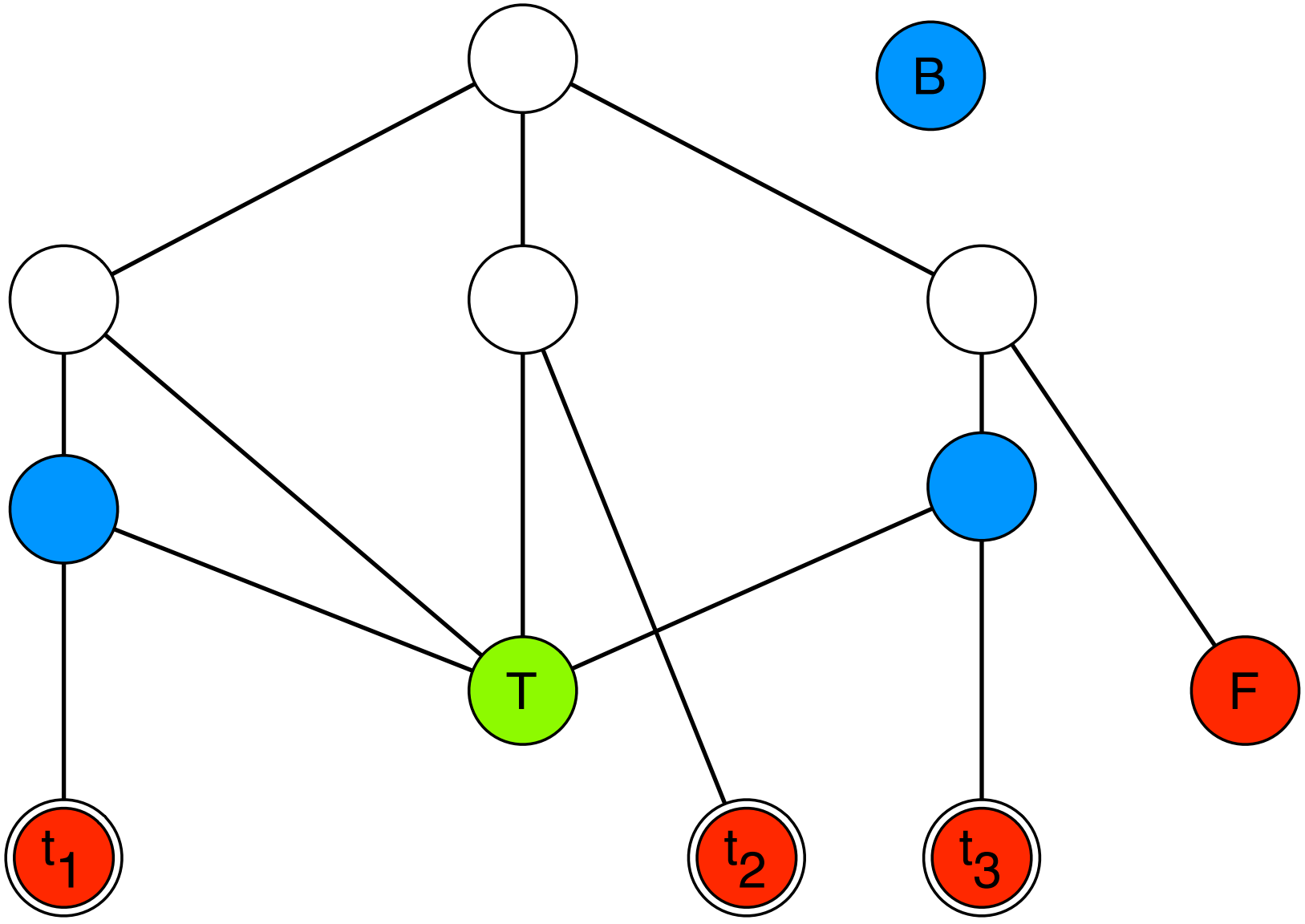
Suppose Every Term Was False

What if every term in the clause was assigned the **false** color?

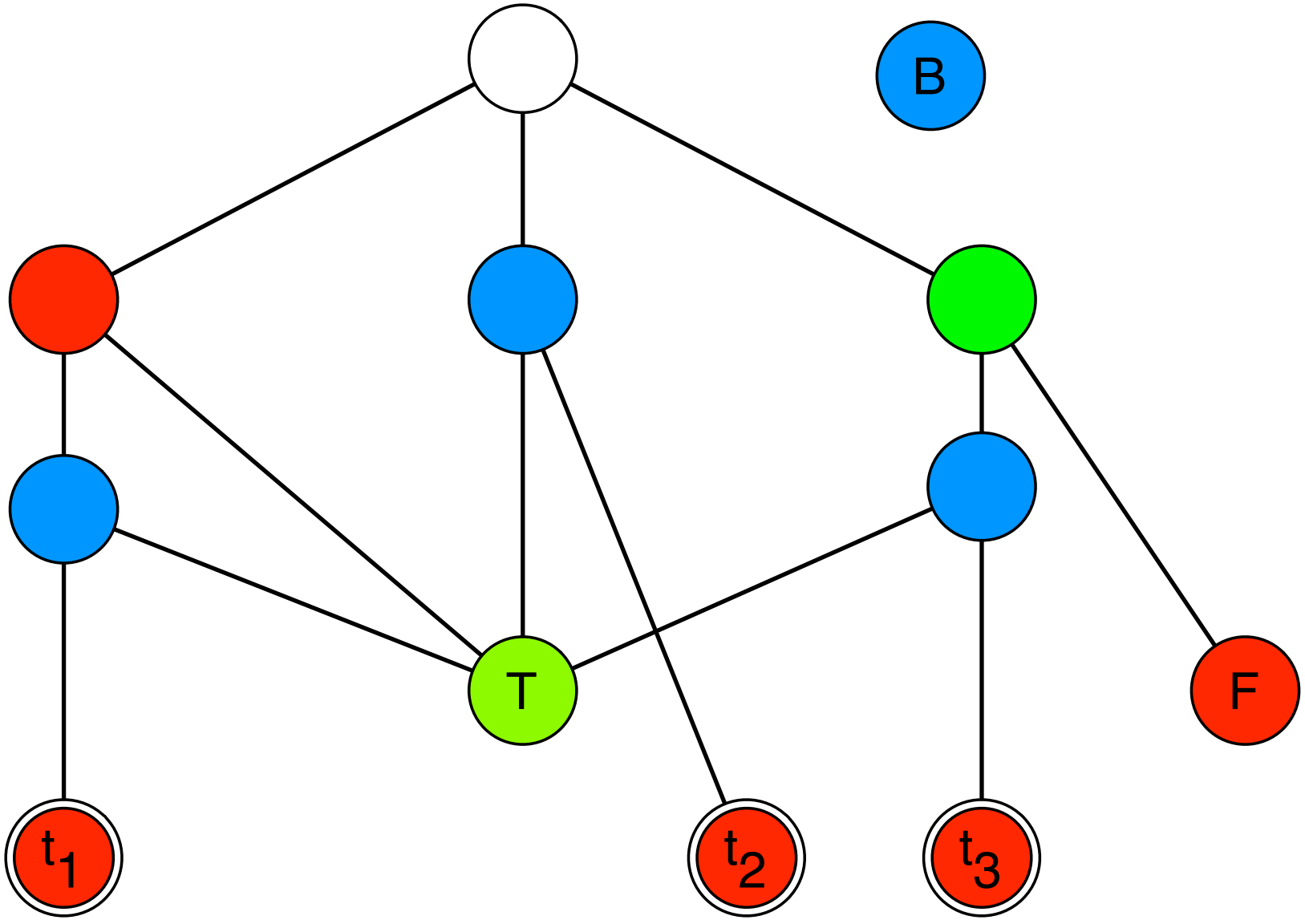
Connect Clause (t_1, t_2, t_3) up like this:



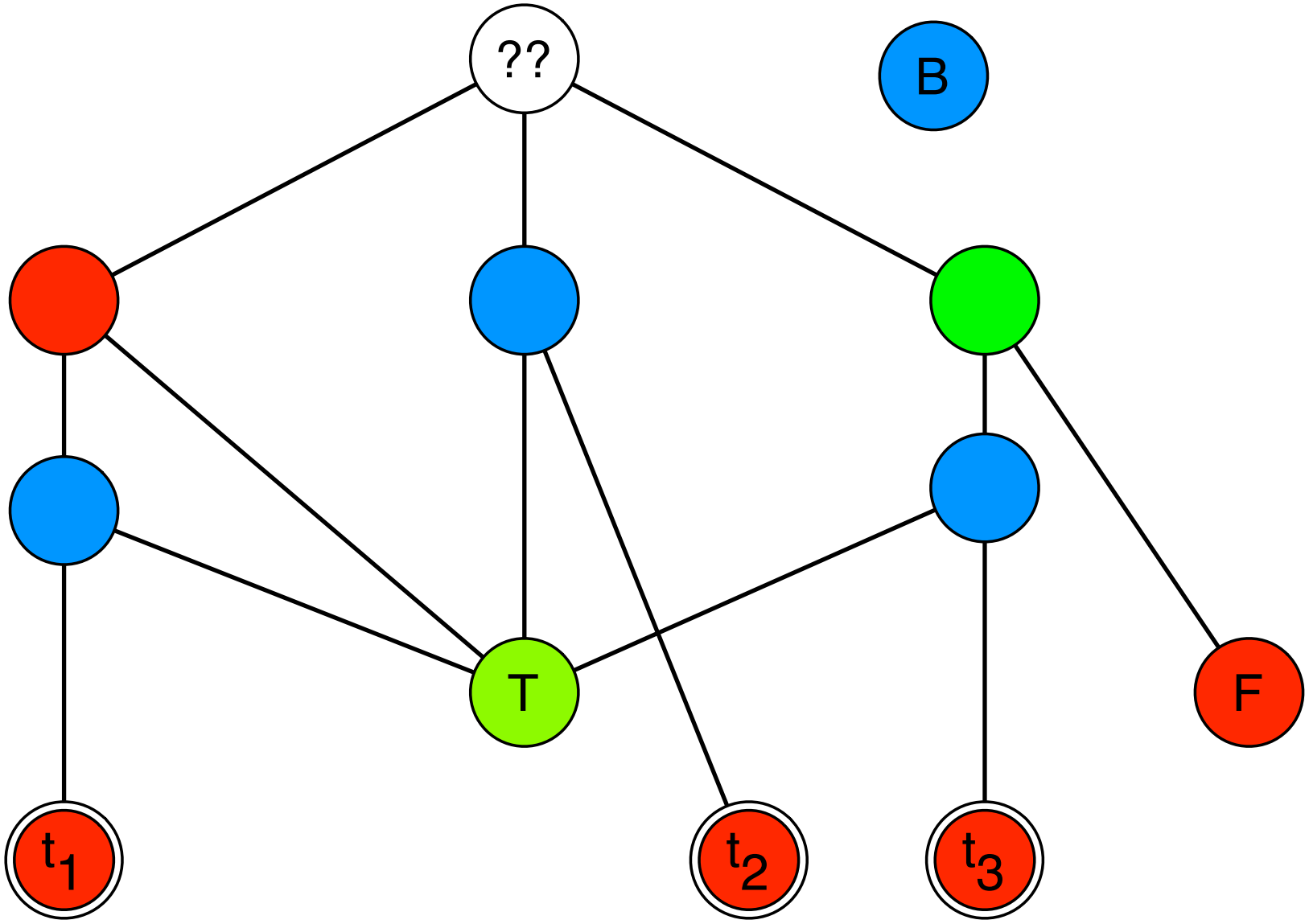
Connect Clause (t_1, t_2, t_3) up like this:



Connect Clause (t_1, t_2, t_3) up like this:



Connect Clause (t_1, t_2, t_3) up like this:



Suppose there is a 3-coloring

Top node is colorable iff one of its terms gets the **true** color.

Suppose there is a 3-coloring.

We get a satisfying assignment by:

- Setting $x_i =$ **true** iff v_i is colored the same as T

Let C be any clause in the formula. At least 1 of its terms must be true, because if they were all false, we couldn't complete the coloring (as shown above).

Suppose there is a satisfying assignment

Suppose there is a satisfying assignment.

We get a 3-coloring of G by:

- Coloring T, F, B arbitrarily with 3 different colors
- If $x_i =$ **true**, color v_i with the same color as T and $\bar{v}_i$ with the color of F.
- If $x_i =$ **false**, do the opposite.
- Extend this coloring into the clause gadgets.

Hence: the graph is 3-colorable iff the formula it is derived from is satisfiable.

General Proof Strategy

General Strategy for Proving Something is NP-complete:

1. Must show that $X \in \mathbf{NP}$. Do this by showing there is an certificate that can be efficiently checked.

2. Look at some problems that are known to be NP-complete (there are thousands), and choose one Y that seems "similar" to your problem in some way.

3. Show that $Y \leq_P X$.

Strategy for Showing $Y \leq_P X$

One strategy for showing that $Y \leq_P X$ often works:

1. Let I_Y be any instance of problem Y.
2. Show how to construct an instance I_X of problem X in polynomial time such that:
   - If $I_Y \in Y$, then $I_X \in X$
   - If $I_X \in X$, then $I_Y \in Y$

Hamiltonian Cycle

Hamiltonian Cycle Problem

Hamiltonian Cycle

Given a directed graph G, is there a cycle that visits every vertex exactly once?

Such a cycle is called a Hamiltonian cycle.

Hamiltonian Cycle is NP-complete

Theorem

Hamiltonian Cycle is NP-complete.

Proof. First, HamCycle $\in$ **NP**. Why?

Second, we show 3-SAT $\leq_P$ Hamiltonian Cycle.

Suppose we have a black box to solve Hamiltonian Cycle, how do we solve 3-SAT?

In other words: how do we encode an instance I of 3-SAT as a graph G such that I is satisfiable exactly when G has a Hamiltonian cycle.

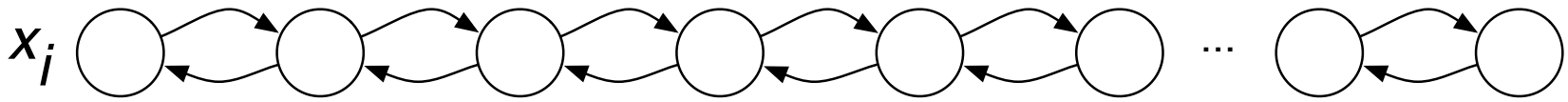
Consider an instance I of 3-SAT, with variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_k$.

Reduction Idea

Reduction Idea (very high level):

- Create some graph structure (a "gadget") that represents the variables
- And some graph structure that represents the clauses
- Hook them up in some way that encodes the formula
- Show that this graph has a Ham. cycle iff the formula is satisfiable.

Gadget Representing the Variables



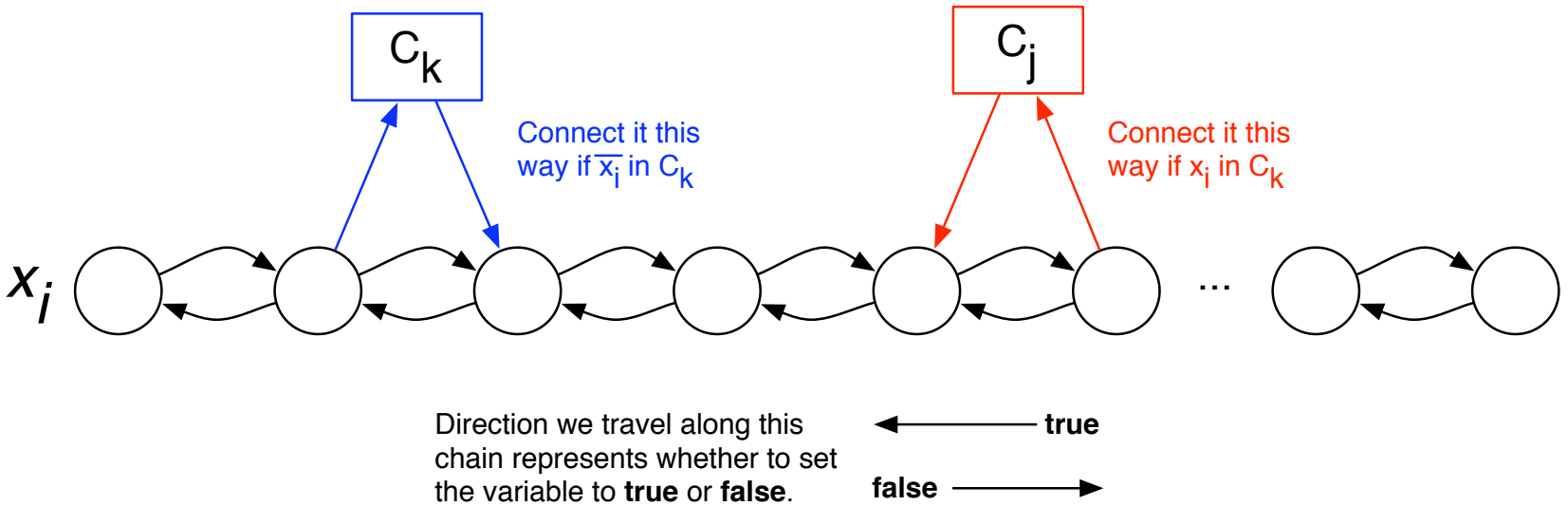
Direction we travel along this chain represents whether to set the variable to **true** or **false**.

true ←

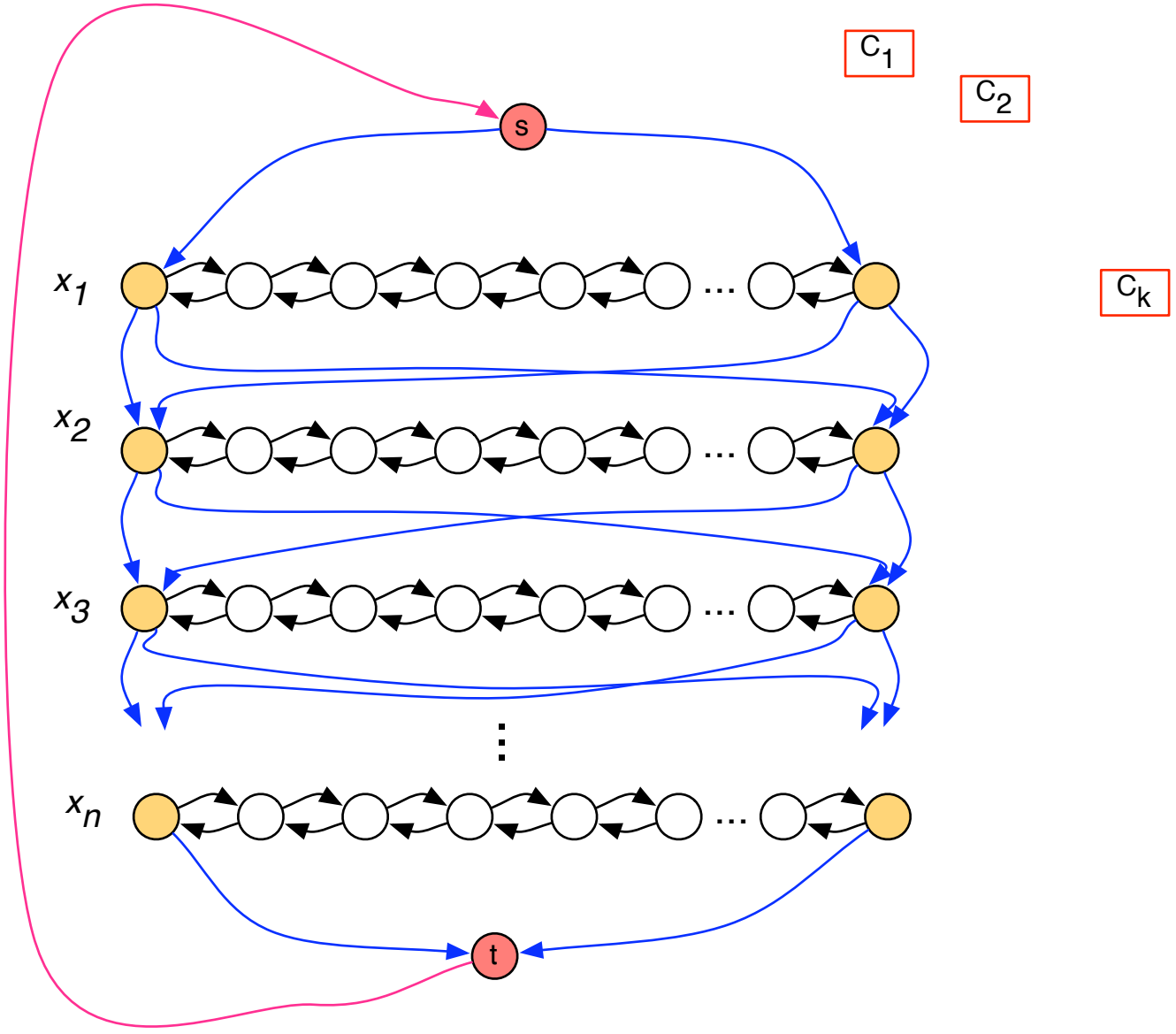
false →

Hooking in the Clauses

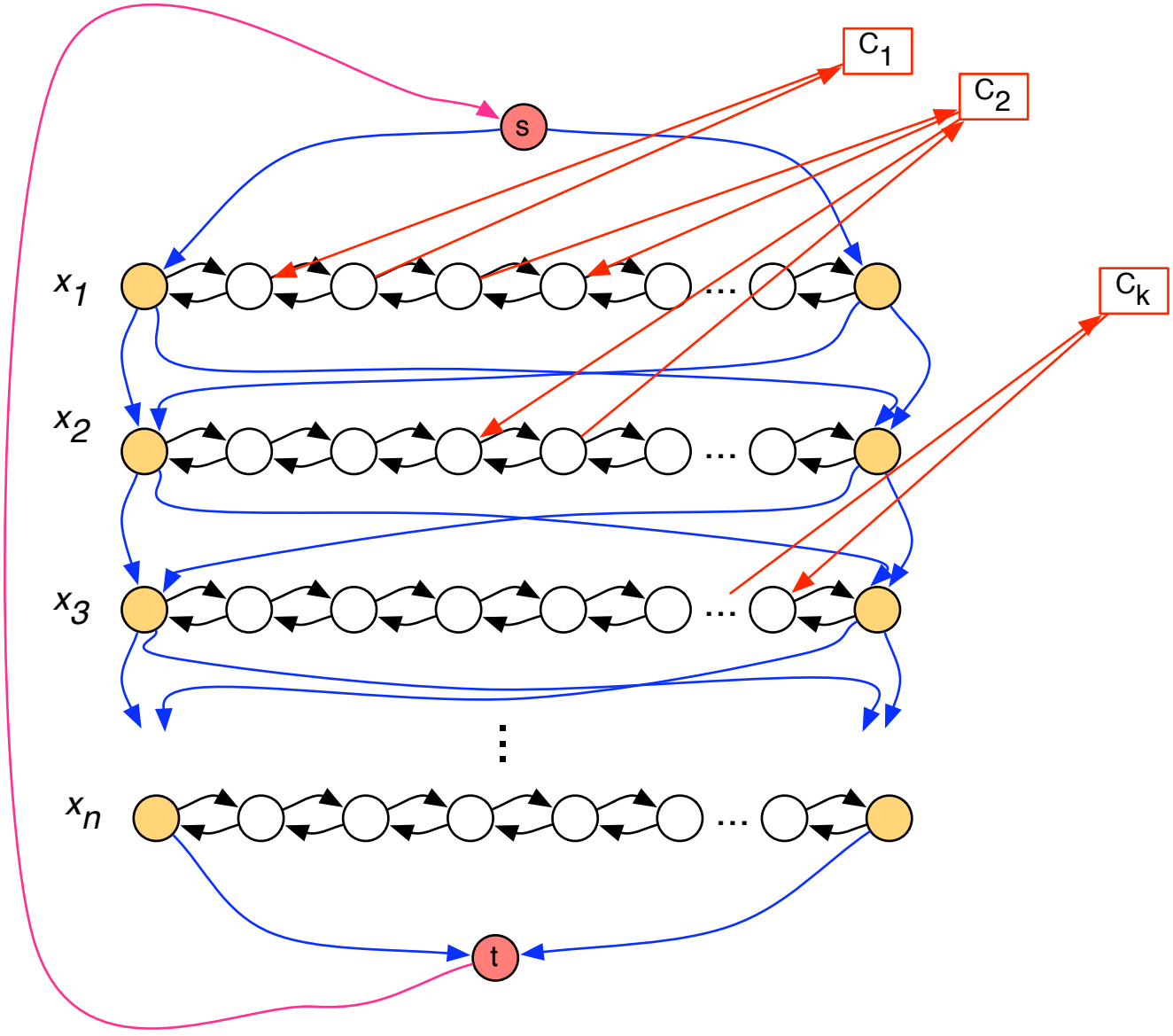
Add a new node for each clause:



Connecting up the paths



Connecting up the paths



Hamiltonian Cycle is NP-complete

- A Hamiltonian path encodes a truth assignment for the variables (depending on which direction each chain is traversed)
- For there to be a Hamiltonian cycle, we have to visit every clause node
- We can only visit a clause if we satisfy it (by setting one of its terms to true)
- Hence, if there is a Hamiltonian cycle, there is a satisfying assignment

Hamiltonian Path

Hamiltonian Path: Does G contain a path that visits every node exactly once?

How could you prove this problem is NP-complete?

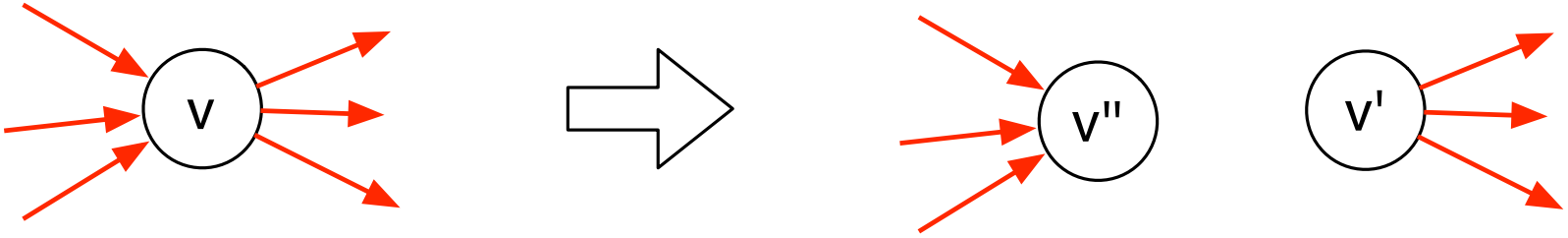
Hamiltonian Path

Hamiltonian Path: Does G contain a path that visits every node exactly once?

How could you prove this problem is NP-complete?

Reduce Hamiltonian Cycle to Hamiltonian Path.

Given instance of Hamiltonian Cycle G, choose an arbitrary node v and split it into two nodes to get graph G':



Now any Hamiltonian Path must start at v' and end at v''.

Hamiltonian Path

G'' has a Hamiltonian Path $\iff$ G has a Hamiltonian Cycle.

$\implies$ If G'' has a Hamiltonian Path, then the same ordering of nodes (after we glue v' and v'' back together) is a Hamiltonian cycle in G.

$\impliedby$ If G has a Hamiltonian Cycle, then the same ordering of nodes is a Hamiltonian path of G' if we split up v into v' and v''. $\square$

Hence, Hamiltonian Path is NP-complete.

Traveling Salesman Problem

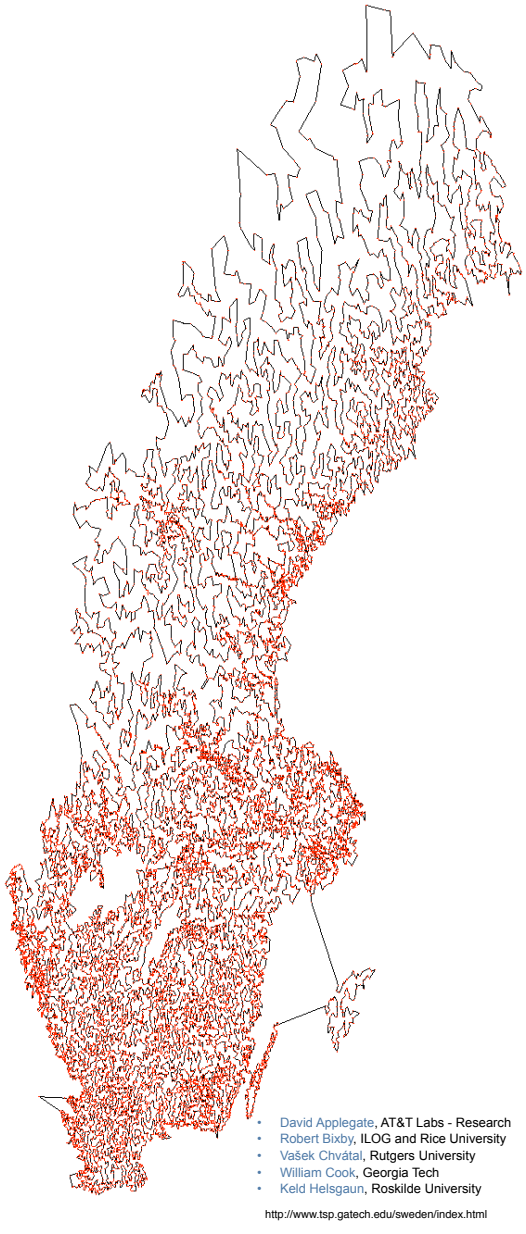
Traveling Salesman Problem

Given n cities, and distances $d(i, j)$ between each pair of cities, does there exist a path of length $\leq k$ that visits each city?

Notes:

- We have a distance between every pair of cities.
- In this version, $d(i, j)$ doesn't have to equal $d(j, i)$.
- And the distances don't have to obey the triangle inequality ($d(i, j) \leq d(i, k) + d(k, j)$ for all i, j, k).

TSP large instance



- TSP visiting 24,978 (all) cities in Sweden.
- Solved by David Applegate, Robert Bixby, Vašek Chvátal, William Cook, and Keld Helsgaun
- `http://www.tsp.gatech.edu/sweden/index.html`
- Lots more cool TSP at `http://www.tsp.gatech.edu/`

Traveling Salesman is NP-complete

Thm. Traveling Salesman is NP-complete.

TSP seems a lot like Hamiltonian Cycle. We will show that

$$\textsc{Hamiltonian Cycle} \leq_P \textit{TSP}$$

To do that:

Given: a graph $G = (V, E)$ that we want to test for a Hamiltonian cycle,

Create: an instance of TSP.

Creating a TSP instance

A TSP instance D consists of n cities, and $n(n-1)$ distances.

Cities We have a city c_i for every node v_i.

Distances Let $d(c_i, c_j) = \begin{cases} 1 & \text{if edge } (v_i, v_j) \in E \\ 2 & \text{otherwise} \end{cases}$

TSP Redcution

Theorem

G has a Hamiltonian cycle $\iff$ D has a tour of length $\leq n$.

Proof. If G has a Ham. Cycle, then this ordering of cities gives a tour of length $\leq n$ in D (only distances of length 1 are used).

Suppose D has a tour of length $\leq n$. The tour length is the sum of n terms, meaning each term must equal 1, and hence cities that are visited consecutively must be connected by an edge in G. $\square$

Also, TSP $\in$ **NP**: a certificate is simply an ordering of the n cities.

TSP is NP-complete

Hence, TSP is NP-complete.

Even TSP restricted to the case when the $d(i, j)$ values come from actual distances on a map is NP-complete.

CS 701 - Final Term 2015

1. Let CNFk = {<ϕ> | ϕ is satisfiable CNF-formula where each variable appears in at most k places}. Show that CNF3 is NP-Complete.

Ans:

(a) We give an informal description of a polynomial time decider M for CNF_2. On input φ M does the following:

1. Consider the first clause of φ. If it is of the form x, and there is a clause $\neg x$ in φ, reject,
2. otherwise the first clause is of the form $x \vee A$. If x does not appear negated in the other clauses, remove every clause of the form $x \vee B$ of φ, and call the result φ, if there remain no clauses in φ, accept,
3. if there are two clauses $x \vee A$ and $\neg x \vee B$ in φ, remove them from φ and add $A \vee B$ to φ and call the result φ, and go to 1.

It is clear that M is a decider. We leave it to the reader to show that its running time is polynomial and that it decides CNF_2.

(b) We leave it to the reader to show that CNF_3 is in NP. We show that $3SAT \leq_P \mathrm{CNF}_3$.

First an example: $\varphi = (p \vee q) \wedge (p \vee r) \wedge (\neg p \vee s) \wedge (p \vee t)$. This formula is not in 3nmf, but the general case will be treated below. This formula is replaced by

$$\psi = (p \vee q) \wedge (\neg p \vee p_1) \wedge (p_1 \vee r) \wedge (\neg p_1 \vee p_2) \wedge (\neg p_2 \vee s) \wedge (\neg p_2 \vee p_3) \wedge (p_3 \vee t) \wedge (\neg p_3 \vee p).$$

Note that the $(\neg p \vee p_1), (\neg p_1 \vee p_2), (\neg p_2 \vee p_3), (\neg p_3 \vee p)$ are expressing that $p \to p_1, p_1 \to p_2, p_2 \to p_3, p_3 \to p$, that is, the p, p_1, p_2, p_3 are all equivalent. Hence in this formula $(p_1 \vee r)$ expresses the same as $(p \vee r)$, etc. And thus φ is satisfiable when ψ is, but in ψ every variable occurs at most 3 times.

The general case. We define a reduction f from $3SAT$ to CNF_3 as follows. $f(\langle\varphi\rangle)$ is the formula that is the result of the following procedure:

1. pick the first propositional variable (reading from left to right) in φ that occurs more than 3 times in the formula, suppose it is p, and suppose it occurs

2. A subset of nodes of a graph is a dominating set if every other node of G is adjacent to some node in the subset. Let DOMINATING-SET={<G, k> | G has a dominating set with k nodes}. Show that it is NP-Complete by giving a reduction from VERTEX-COVER

Solution:

Clearly Dominating Set is in **NP**. Given a dominating set, one can verify in polynomial time if that is a dominating set. This can be done by taking each vertex and checking if it is either in the given set or one of its edges travel into the set.

To show that is **NP**-complete, first of all notice that a dominating set has to include all isolated vertices (those which have no edges from them). So let us assume that our graph does not have any isolated vertices. We will show that Dominating Set is **NP**-complete using a reduction from Vertex Cover. Given a graph G, we will construct a graph G' as follows. G' has all edges and vertices of G. Also, for every edge $\{u, v\} \in G$, we add intermediate node on a parallel path in G'. Keeping $\{u, v\}$ intact in G', we add vertex w and edges $\{u, w\}$ and $\{w, v\}$ in G'. Now we will show that G has a vertex cover of size k **if and only if** G' has a dominating set of the same size.

If S is a vertex cover in G, we will show that S is a dominating set for G'. S is a vertex cover, this means that every edge in G has atleast one of its end points in S. Consider $v \in G'$. If v is an original node in G, then either $v \in S$ or there must be some edge connecting v to some other vertex u. Since S is a vertex cover, is $v \notin S$, then u must be in S, and hence there is an adjacent vertex of v in S. So v is covered by some element in S. However, if w is an additional node in G', then w has two adjacent vertices $u, v \in G$ and using the above argument at least one of them is in S. So the additional nodes are also covered by S. So if G has a vertex cover, then G' has a dominating set of at most the same size (in fact the same set itself would do).

3. Let ADD = {<x, y, z> | x, y, z >0 are binary integers and x + y =z}.
   Show that ADD ϵ LSPACE.

4. Let PAL-ADD = {<x, y> | x, y >0 are binary integers where x + y is an integer whose binary representation is palindrome}. Show that PAL-ADD ϵ LSPACE.

5. Let NEAR-TAUT: E is a Boolean expression having when at most one true make it false. Show that complement of NEAT-TAUT is in NP-Complete using reduction it to SAT.

6. A directed Graph is STRONGLY-CONNECTED of every tow nodes are connected by a directed graph in each direction. Let STRONGLY-CONNECTED = {<G>|G is strongly connected graph}.

7. Show that STRONGLY-CONNECTED is NP-Complete.

8. Let PAL-ADD = {x,y>0, are binary integers whose x+y is an integer is a Palindrome}. Show PAL-ADD ε LSPACE.

9. Let TRUE-SAT; given Boolean expression E that is true when all the variables are made true, is there some other truth assignment besides all true that make E true. Show that TURE-SAT is NP complete by reducing SAT to it.

10. Let A = {<M, x, #t>|NTM, M accepts input x within t steps on at least one brach}. Showt hat A is NP Completeness.

11. Let G represent and undirected graph. Also let LPATH = {<G,a,b,k>|G contains a simple path of length k for a to b>}. Show that LPATH is NP complete. You may assume that NP completeness of UHAMPTH the HAMPATH problem of undirected graph.

12. Prove that Double SAT is NP complete by reducing from 3 SAT.

13. Prove that SET SPLITTING is NP complete.

14. Let NEAR TAUT; given Boolean expression E all the variables are made f, is there some other truth assignment besides all f that make E true. Show that NEAR TAUT is NP complete by reducing SAT to it.
15. Let PAL-ADD = {x,y>0, are binary integers whose x+y is an integer is a Palindrome}. Show PAL-ADD ε LSPACE.

16. Nim game question to prove unbalance and balance

17. See the Questions of Sipser Book 2013 edition 3 Excercises
18.

    7.22 7.30 8.19 8.22

*7.48 Call a regular expression ***star-free*** if it does not contain any star operations. Let $EQ_{\mathsf{SF-REX}} = \{\langle R, S\rangle \mid R \text{ and } S \text{ are equivalent star-free regular expressions}\}$. Show that $EQ_{\mathsf{SF-REX}}$ is in coNP. Why does your argument fail for general regular expressions?

Answer: To show that $\overline{\mathrm{EQ}_{SF\_RFX}}$ is in NP, we observe that a string x is in $\overline{\mathrm{EQ}_{SF\_RFX}}$ if and only if it is one of the following forms:

(a) x does not represent a valid encoding of two regular expressions;

(b) x is of the correct form $\langle R, S\rangle$, but either R, or S, or both are not star-free;

(c) x is of the correct form $\langle R, S\rangle$, R and S are both star-free, but $L(R) \neq L(S)$.

If x is in the first and the second form, x can be accepted by a DTM easily in polynomial time. If x is in the third form, there exists a string y that is in exactly one of the $L(R)$ or $L(S)$. As R and S are star-free, the length of y must be polynomial in the size of $|R| + |S|$ (why?). Thus, there is an NTM N that guesses such a string y in polynomial time whenever x is of the third form (but will never find such a string y when $L(R) = L(S)$). Thus, there exists an NTM that recognizes $\overline{\mathrm{EQ}_{SF\_RFX}}$ in polynomial time. This completes the proof.

*7.46 Let $MAX\text{-}CLIQUE = \{\langle G, k\rangle \mid$ the largest clique in G is of size exactly $k\}$. Use the result of Problem 7.45 to show that *MAX-CLIQUE* is DP-complete.

If P = NP, then *CLIQUE* is recognizable in polynomial time. We show how to compute *MAX-CLIQUE* using *CLIQUE*. Let m be the number of nodes in the input graph. For each i from 1 to m, test whether there exists a clique of size i using the polynomial time algorithm for *CLIQUE*. Output the largest such i for which a clique exists.

To find the maximum clique, we start with i, the maximum clique size. Remove one node and see if there is still a clique of size i. If not, restore that node and remove another. If so, iterate the process until we are left with a graph of i nodes, which must be a clique. This process takes at most m trials to find which node to remove, and at most m nodes need to be removed. The total running time is therefore polynomial.

WWW.VUMULTAN.COM