Computer Language Theory

Chapter 7: Time Complexity

Last Modified 4/2/20

Complexity

- A decidable problem is computationally solvable
  - But what resources are needed to solve the problem?
    - How much time will it require?
    - How much memory will it require?
  - In this chapter we study time complexity
  - Chapter 8 covers the space complexity of a problem
    - Space corresponds to memory
    - We do not cover space complexity (this topic is rarely covered in introductory theory courses)

Goals

- Basics of time complexity theory
  - Introduce method for the measuring time to solve a problem
  - Show how to classify problems according to the amount of time required
  - Show that certain classes of problems require enormous amounts of time
    - We will see how to determine if we have this type of problem

Chapter 7.1

Measuring Complexity

An Example of Measuring Time

- Take the language $A = \{0^k 1^k \mid k \geq 0\}$
  - A is clearly decidable
    - we can write a program to recognize it
  - How much time does a single-tape Turing machine, M1, need to decide A?
  - Do you recall the main steps involved?

Turing Machine M1

M1 = On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1
2. Repeat if both 0s and 1s remain on the tape
    3. Scan across the tape, crossing off a single 0 and a single 1
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain, *accept*.

Time Analysis of a TM

- The number of steps that an algorithm/TM requires may depend on several parameters
  - If the input is a graph, then the number of steps may depend on:
    - Number of nodes, number of edges, maximum degree of the graph
    - For this example, what does it depend on?
      - The length of the input string
      - But also the value of the input string $(1 \bullet 0^{10,000} \bullet 1^{10,000})$
  - For simplicity, we compute the running time of an algorithm as a function of the length of the string that represents the input
    - In worst-case analysis, we consider the longest running time of all inputs of a particular length (that is all we care about here)
    - In average-case analysis, we consider the average of all running times of inputs of a particular length

Running Time/Time Complexity

- Definition: Running time or time complexity:
  - If M is a TM that halts on all inputs, the running time of M is the function $f: N \rightarrow N$, where $f(n)$ is the maximum number of steps that M uses on any input of length n.
    - We say that M runs in time $f(n)$ and that M is an $f(n)$ time Turing machine.
    - Custom is to use n to represent the input length

Big-O and Small-O Notation

- We usually just estimate the running times
- In asymptotic analysis, we focus on the algorithm's behavior when the input is large
  - We consider only the highest order term of the running time complexity expression
    - The high order term will dominate low order terms when the input is sufficiently large
    - We also discard constant coefficients
      - For convenience, since we are estimating

Examples of using Big-O Notation

- If the time complexity is given by the f(n) below:
  - $f(n) = 6n^3 + 2n^2 + 20n + 45$ then
  - Then f(n) = O(?)
  - An answer, using what was just said, is:
    - $F(n) = O(n^3)$

Definition of Big-O

- Let f and g be functions f, g: $N \rightarrow R^+$. Say that f(n) = O(g(n)) if positive integers *c* and n_o exist such that for every integer $n \geq n_o$
  - $f(n) \leq c\ g(n)$
  - When f(n) = O(g(n)) we say that g(n) is an asymptotic upper bound for f(n), to emphasize that we are suppressing constant factors
- Intuitively, f(n) = O(g(n)) means that f is less than or equal to g if we disregard differences up to a constant factor.

Example

- From earlier, we said if the time complexity is:
  - $f(n) = 6n^3 + 2n^2 + 20n + 45$, then
  - $f(n) = O(n^3)$
- Is f(n) always less than c g(n) for some n?
  - That is, is $f(n) \leq c\ g(n)$?
  - Try n=0; we get $45 \leq 0$ (c x 0)
  - Try n=10; we get $6000 + 200 + 200 + 45 \leq c\ 1000$
    - $6445 \leq 1000c$. Yes, if c is 7 or more.
    - So, if $c = 7$, then true whenever $n_o \geq 10$

Example continued

- What if we have:
  - $f(n) = 6n^3 + 2n^2 + 20n + 45$, then
  - $f(n) = O(n^2)$?
    - Let's pick n = 10
    - $6445 \leq 100c$
      - True if c = 70. But then what if n is bigger, such as 100
      - Then we get $6{,}022{,}045 \leq 10{,}000c$
        - Now c has to be 603
        - So, this fails and hence f(n) is not $O(n^2)$.
        - Note that you must fix c and n_o. You cannot keep changing c. If you started with a larger c, then you would have to get a bigger n, but eventually you would always cause the inequality to fail.
        - If you are familiar with limits (from calculus), then you should already understand this

Another way to look at this

- Imagine your graph $y1=(x^2)$ and $y2=(x^3)$
  - Clearly y2 would be bigger than y1 if $x > 0$.
  - However, we could change y1 so that it is equal to cx^2 and then y1 could be bigger than y2 for some values of x
    - However, once x gets sufficiently large, for any value of c, then y2 will be bigger than y1

Example Continued

- Note that since $f(n) = O(n^3)$, then it would trivially hold for $O(n^4)$, $O(n^5)$, and 2^n
- The big-O notation does not require that the upper bound be "tight" (i.e., as small as possible)
- For the perspective of a computer scientist who is interesting in characterizing the performance of an algorithm, a tight bound is better

A Brief Digression on Logarithms

- As it turns out, $\log_2 n$ and $\log_{10} n$ and $\log_x n$ differ from each other by a constant amount
  - So if we use logarithms inside the O(log n), we can ignore the base of the logarithm
- Note that log n grows much more slowly than a polynomial like n or n^2. An exponential like 2^n or 5^n or 7^{2n} grows much faster than a polynomial.
  - Recall from "basic" math that a logarithm is essentially the opposite of an exponential
    - Example, $\log_2 16 = 4$ since $2^4=16$. In general, $\log_2 2^n = n$

Small-O Notation

- Big-O notation says that one function is asymptotically no more than another
  - Think of it as $\leq$.
- Small-O notation says that one function is asymptotically less than another
  - This of it as $<$
  - Examples: Come up with small-o
    - If $f(n) = n$, then $f(n) = o(?)$
      - $f(n) = o(n \log n)$ or $o(n^2)$
    - If $f(n) = n^2$ then $f(n) = o(?)$
      - $f(n) = o(n^3)$
  - A small-O bound is always also a big-O bound (not vice-versa)
    - Just as if $x < y$ then $x \leq y$ ($x \leq y$ does not mean that $x < y$)

Formal Definition of Small-O

- Let f and g be functions f, g: N → R+. Say that f(n) = o(g(n)) if:
  - Limit as $n \to \infty$ f(n)/g(n) = 0
  - This is equivalent to saying that f(n) = o(g(n)) then, for any real number c > 0, a number n_o exists where f(n) < cg(n) for all n ≥ n_o.

Now Back to Analyzing Algorithms

- Take the language $A = \{0^k 1^k \mid k \geq 0\}$

M1 = On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1
2. Repeat if both 0s and 1s remain on the tape
    3. Scan across the tape, crossing off a single 0 and a single 1
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain, *accept*.

How to Analyze M1

- To analyze M1, consider each of the four stages separately
- How many steps does each take?

Analysis of M1

M1 = On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1
2. Repeat if both 0s and 1s remain on the tape
    3. Scan across the tape, crossing off a single 0 and a single 1
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain, *accept*.

- Stage 1 takes: ? steps
    - n steps so O(n)
- Stage 3 takes: ? Steps
    - n steps so O(n)
- Stage 4 takes: ? Steps
    - n steps so O(n)
- How many times does the loop in stage 2 cause stage 3 to repeat?
    - n/2 so O(n) steps
- What is the running time of the entire algorithm?
    - $O(n) + n/2 \times O(n) + O(n) = O(n^2/2) = O(n^2)$

Time Complexity Class

- Let t: N $\rightarrow$ R+ be a function. We define the time complexity class, TIME(t(n)), to be the collection of all languages that are decidable by an O(t(n)) time Turing machine.
  - Given that language A was accepted by M1 which was $O(n^2)$, we can say that A $\in$ $TIME(n^2)$ and that $TIME(n^2)$ contains all languages that can be decided in $O(n^2)$ time.

Is There a Tighter Bound for A?

- Is there a machine that will decide A asymptotically more quickly?
  - That is, is A in TIME(t(n)) for $t(n) = o(n^2)$?
    - If it is, then there should be a tighter bound
  - The book gives an algorithm for doing it in O(nlogn) on page 252 by crossing off every other 0 and 1 at each step (assuming total number of 0s and 1s is even)
    - Need to look at the details to see why this works, but not hard to see why this is O(n log n).
      - Similar to M1 that we discussed except that the loop is not repeated n/2 times.
      - How many times is it repeated?
        - Answer $\log_2 n$ (but as we said before the base does not matter so log n)
      - So, that yields nlogn instead of n^2

Can we Recognize A even Faster?

- If you had to program this, how long would it take (don't worry about using a Turing machine)?
  - I can do it in O(n) time by counting the number of 0s and then subtracting each 1 from that sum as we see it
- Can you do it in O(n) with a Turing Machine?
  - Not with a single tape Turing Machine
  - But you can with a 2-tape Turing Machine
  - How?

A TM M3 to Recognize A in O(n)

- M3 = On input string w:
  1. Scan across the tape and reject if a 0 is found to the right side of a 1.
  2. Scan across the 0s on tape 1 until the first 1 is seen. As each 0 is seen, copy a 0 to tape 2 (unary counting)
  3. Scan across the 1s on tape 1 until the end of the input. For each 1 read on tape 1, cross off a 0 on tape 2. If all 0s are crossed off before all 1s are read, *reject*.
  4. If all the 0s have now been crossed off, *accept*. If any 0s remain, *reject*.
- What is the running time:
  - 1: O(n) 2: O(n) 3: O(n) 4: O(n)
- Is it possible to do even better?
  - No, since the input is O(n). Could only do better for a problem where it is not necessary to look at all of the input, which isn't the case here.

Implications

- On a 1-tape TM first we came with a $O(n^2)$ algorithm and then a O(nlogn) algorithm
  - So, the algorithm you choose helps to decide the time complexity
    - Hopefully no surprise there
    - As it turns out, O(nlogn) is optimal for a 1-tape TM
- On a 2-tape TM, we came up with a O(n) algorithm
  - So, the computational model does matter
    - A 2-tape and 1-tape TM are of equivalent computational power with respect to what can be computed, but have different time complexities for same problem
    - How can we ever say anything interesting if everything is so model dependent?
    - Recall that with respect to computability, TMs are incredibly robust in that almost all are computationally equivalent. That is one of the things that makes TMs a good model for studying computability

A Solution

- One solution is to come up with a measure of complexity that does not vary based on the model of computation
  - That necessarily means that the measure cannot be very fine-grained
- We will briefly study the relationships between the models with respect to time complexity
  - Then we can try to come up with a measure that is not impacted by the differences in the models

Complexity Relationships between Models

- We will consider three models:
  - Single-tape Turing machine
  - Multi-tape Turing machine
  - Nondeterministic Turing machine

Complexity Relationship between Single and Multi-tape TM

- Theorem:
  - Let t(n) be a function, where $t(n) \geq n$. Then every t(n) time multitape Turing machine has an equivalent $O(t^2(n))$ time single-tape TM
  - Proof Idea:
    - We previously showed how to convert any multi-tape TM into a single-tape TM that simulates it. We just need to analyze the time complexity of the simulation.
    - We show that simulating each step of the multitape TM uses at most O(t(n)) steps on the single-tape machine. Hence the total time used is $O(t^2(n))$ steps.

Proof of Complexity Relationship

- Review of simulation:
  - The single tape of machine S must represent all of the tapes of multi-tape machine M
    - S does this by storing the info consecutively and the positions of the tape heads is encoded with a special symbol
    - Initially S puts its tape into the format that represents the multi-tape machine and then simulates each step of M
  - A single step of the multi-tape machine must then be simulated

Simulation of Multi-tape step

- To simulate one step:
  - S scans tape to determine contents under tape heads
  - S then makes second pass to update tape contents and move tape heads
    - If a tape heads moves right into new previously unused space (for the corresponding tape in M), then S must allocate more space for that tape
      - It does this by shifting the rest of the contents one cell to the right
- Analysis of this step:
  - For each step in M, S makes two passes over active portion of the tape. First pass gets info and second carries it out.
  - The shifting of the data, when necessary, is equivalent to moving over the active portion of the tape
  - So, equivalent to three passes over the active contents of the tape, which is same as one pass. So, O(length of active contents).
    - We now determine the length of the active contents

Length of Active Contents on Single-Tape TM

- Must determine upper bound on length of active tape
  - Why an upper bound?
    - Answer: we are looking a worst-case analysis
  - Active length of tape equal to sum of lengths of the k-tapes being simulated
    - What is the maximum length of one such active tape?
      - Answer: t(n) since can at most make t(n) moves to the right in t(n) steps
  - Thus a single scan of the active portion takes O(t(n)) steps
    - Why not O(k x t(n)) since k tapes in k-multitape machine M?
      - Answer: k is a constant so drops out
- Putting it all together:
  - O(n) to set up tape initially and then t(n) steps to simulate each of the O(t(n)) steps.
  - This yields $O(n) + O(t^{2(n)}) = O(t^{2(n)})$ given that $t(n) \geq n$. Proof done!

Complexity Relationship between Single-tape DTM and NDTM

- Definition:
  - Let N be a nondeterministic Turing machine that is a decider. The running time of N is:
    - Function f: N → N, where f(n) is the maximum number of steps that N uses on any branch of its computation on any input of length n
- Doesn't correspond to real-world computer
  - Except maybe a quantum computer?*
  - Rather a mathematical definition that helps to characterize the complexity of an important class of computational problems
  - As I said before, non-determinism is like always guessing correctly (as if we had an oracle). Given this, the running time result makes sense.

* This is more related to this course than you might think. Lately many new methods for computation have arisen in research, such as quantum and DNA computing. These really do 1) show that computing is not about "computers" and 2) that it is useful to study different models of computation. Quantum computing is radically different.

Proof of Complexity Relationship

- Theorem:
  - Let t(n) be a function, where $t(n) \geq n$. Then every t(n) time nondeterministic single-tape Turing machine has an equivalent $2^{O(t(n))}$ time deterministic single-tape TM
- Note that $2^{O(t(n))}$ is exponential, which means it grows very fast
  - Exponential problems considered "computationally intractable"

Definition of NTM Running Time

- Let N be a Nondeterministic Turing Machine that is a decider. The running time is the maximum number of steps that N takes on any branch of its computation on any input of length n.
  - Essentially the running time assumes that we always guess correctly and execute only one branch of the tree. The worst case analysis means that we assume that correct branch may be the longest one.

Proof

- Proof:
  - This is based on the proof associated with Theorem 3.16 (one of the few in Chapter 3 that we did not do)
  - Construct a deterministic TM D that simulates N by searching N's nondeterministic computation tree
    - Finds the path that ends in an accept state.
    - On input n the longest branch of computation tree is t(n)
    - If at most b transitions, then number of leaves in tree is at most $b^{t(n)}$
    - Explore it breadth first
    - Total number of nodes in tree is less than twice number of leaves (basic discrete math) so bounded by $O(b^{t(n)})$

Proof continued

- The way the computation tree is searched in the original proof is very inefficient, but it does not impact the final result, so we use it
  - It starts at the root node each time
  - So, it goes to $b^{t(n)}$ nodes and must travel $O(t(n))$ each time.
    - This yields $b^{t(n)}\ O(t(n))$ steps $= 2^{O(t(n))}$
      - Note that b is a constant ≥ 2 and for running time purposes can be listed as 2 (asymptotically equivalent).
      - The $O(t(n))$ does not increase the overall result, since exponential dominates the polynomial
      - If we traversed the tree intelligently, still must visit $O(b^{t(n)})$ nodes
    - The simulation that we did not cover involves 3 tapes. But going from 3 tapes to 1 tape at worst squares the complexity, which has no impact on the exponential

Chapter 7.2

The Class P

The Class P

- The two theorems we just proved showed an important distinction
  - The difference between a single and multi-tape TM is at most a square, or polynomial, difference
  - Moving to a nondeterministic TM yields an exponential difference
    - A non-deterministic TM is not a valid real-world model
- So we can perhaps focus on polynomial time complexity

Polynomial Time

- From the perspective of time complexity, polynomial differences are considered small and exponential ones large
  - Exponential functions do grow incredibly fast and do grow much faster than polynomial function
  - However, different polynomials do grow much faster than others and in an algorithms course you would be crazy to suggest they are equivalent
    - O(n log n) sorting much better than $O(n^2)$.
    - O(n) and $O(n^3)$ radically different
- As we shall see, there are some good reasons for nonetheless assuming polynomial time equivalence

Background

- Exponential time algorithms arise when we solve problems by exhaustively searching a space of possible solutions using brute force search
- Polynomial time algorithms require something other than brute force (hence one distinction)
- All reasonable computational models are polynomial-time equivalent
  - So if we view all polynomial complexity algorithms as equivalent then the specific computational model does not matter

The Definition of the Class P

- Definition:
  - P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. We can represent this as:
    - $P = \cup_k TIME(n^k)$
      - The k should be under the U, but what we mean is that the language P is the union over all languages that can be recognized in time polynomial in the length of the input, n
        - That is, n^2, n^3, …

The Importance of P

- P plays a central role in the theory of computation because
  - P is invariant for all models of computation that are polynomial equivalent to the deterministic single-tape Turing machine
  - P roughly corresponds to the class of problems that are realistically solvable on a computer
    - Okay, but take this with a grain of salt as we said before
      - And even some exponential algorithms are okay

Another Way of Looking at P

- If something is decidable, then there is a method to compute/solve it
  - It can be in P, in which case there is an "intelligent" algorithm to solve it, where all I mean by intelligent is that it is smarter than brute force
  - It may not be in P in which case it can be solved by brute force
  - If it is not in P then it can only be solved via brute force searching (i.e. trying all possibilities)
  - Note: NP does *not* mean "not in P" (as we shall soon see). In fact every problem in P is in NP (but not vice versa).
    - NP means can be solved in nondeterministic polynomial time (polynomial time on a non-deterministic machine).

Examples of Problems in P

- When we present a polynomial time algorithm, we give a high level description without reference to a particular computational model
- We describe the algorithms in stages
- When we analyze an algorithm to show that it belongs to P, we need to:
  - Provide a polynomial upper bound, usually using big-O notation, on the number of stages in terms of an input of length n
  - We need to examine each stage to ensure that it can be implemented in polynomial time on a reasonable deterministic model
  - Note that the composition of a polynomial with a polynomial is a polynomial, so we get a polynomial overall
    - We choose the stages to make it easy to determine the complexity associated with each stage

The Issue of Encoding the Problem

- We need to be able to encode the problem in polynomial time into the internal representation
  - We also need to decode the object in polynomial time when running the algorithm
  - If we cannot do these two things, then the problem cannot be solved in polynomial time
- Methods for encoding:
  - Standard methods for encoding things like graphs can be used
    - Use list of nodes and edges or an adjacency matrix where there is an edge from i to j if the cell (i,j) equals 1
  - Unary encoding of numbers not okay (not in polynomial time)
    - The decimal number 1,000 has string length 4 but in unary (i.e., 1,000 1's) would be of length 1,000. 10,000 would be 10,000 instead of 5,..
      - The relationship between the two lengths is exponential (in this case 10^n)

Example: Path Problem ∈ P

- The PATH problem is to determine if in a directed graph G whether there is a path from node s to node t
  - PATH = {<G,S,t> | G is a directed graph that has a directed path from s to t}
  - Prove PATH ∈ P
  - Brute force method is exponential and doesn't work
    - Assuming m is the number of nodes in G, the path cannot be more than m, since no cycle can be required
    - Total number of possible paths is around m^m
      - Think about strings of length m. Each symbol represents a node. Actually less than m^m since no node can repeat.
      - This is actually silly since tries paths that are not possible given the information in G

Path Problem ∈ P

- A breadth first search will work
  - Text does not consider this brute-force, but is close
  - M = On input <G, s, t> where G is a directed graph with nodes s and t (and there are m nodes):
    1. Place a mark on node s
    2. Repeat until no additional nodes are marked
       3. Scan all edges of G. If an edge (a,b) is found from a marked node a to an unmarked node b, mark node b
    4. If t is marked, accept. Otherwise, reject.
  - Analysis:
    - Stages 1 and 4 executed exactly 1 time each. Stage 3 runs at most m times since each time a node is marked. There are at most m+2 stages, which is polynomial in the size of G.
    - Stages 1 and 4 are easily implemented in polynomial time on any reasonable model. Stage 3 involves a scan of the input and a test of whether certain nodes are marked, which can easily be accomplished in polynomial time. Proof complete and PATH ∈ P.

RELPRIME ∈ P

- The RELPRIME problem
  - Two numbers are relatively prime if 1 is the largest integer than evenly divides them both.
    - For example, 10 and 21 are relatively prime even though neither number is prime
    - 10 and 22 are not relatively prime since 2 goes into both
  - Let RELPRIME be the problem of testing whether two numbers are relatively prime
    - RELPRIME = {<x,y> | x and y are relatively prime}

A Simple Method for Checking RELPRIME

- What is the most straightforward method for determining if x and y are relatively prime?
  - Answer: Search through all possible divisors starting at 2. The largest divisor to check should be max(x,y)/2.
- Does this straightforward method $\in$ P?
  - Answer: No.
    - Earlier we said that the input must be encoded in a reasonable way, in polynomial time. That means that unary encoding is no good. So, the encoding must be in something like binary (ternary, decimal, etc.)
    - The straightforward method is exponential in terms input string length
      - Example: let the larger of x and y be 1024. Then the binary encoding is length 10. The straightforward method requires 1024/2 steps.
      - The difference is essentially $O(\log_2 n)$ vs. $O(n)$ which is an exponential difference

A More Efficient Algorithm

- We can use the Euclidean algorithm for computing the greatest common divisor
  - If the gcd(x,y) = 1, then x and y are relatively prime
  - Example: gcd(18, 24) = 6
  - The Euclidean algorithm E uses the mod function, where x mod y is the remainder after integer division of x by y.

RELPRIME using Euclidean Algorithm

- We will not prove this algorithm. It is well known.
- E = On input <x,y>, where x and y are natural numbers in binary:
  1. Repeat until y = 0
  2. Assign x ← x mod y
  3. Exchange x and y
  4. Output x
- R solves RELPRIME using E as a subroutine
  - R = On input <x,y> where *x* and *y* are a natural numbers in binary:
    1. Run E on <x,y>
    2. If the result is 1, accept. Otherwise reject.

An Example

- Try E on <18,24> (x=18, y=24)
  - x = 18 mod 24 = 18 [line 2]
  - x=24, y = 18 [line 3]
  - x = 24 mod 18 = 6 [line 2]
  - x = 18, y = 6 [line 3]
  - x = 18 mod 6 = 0 [line 2]
  - y = 0, x = 6 [line 3]
  - Because y = 0, loop terminates and we output x value which is 6.

Continued

- x = x mod y will always leave x < y
- After line 2 the values are switched so next time thru the loop x >y before the mod step is performed
- So, except for the first time thru the loop, the x mod y will always yield a smaller value for x and that value will be < y
  - If x is twice y or more, then it will be cut by at least half (since the new x must be < y)
  - If x is between y and 2y, then it will also be cut in at least half
    - Why? Because x mod y in this case will be x – y
      - If x = 2y-1 this gives y-1 which cuts it in about half
        - Example y=10 so 2y=20 so x = 19. 19 mod 10 = 9. At least in half!!
      - If x = y+1, this gives 1
        - Example y = 10 so x = 11 so 11 mod 10 = 1. At least in half.

Analysis of Running Time of E

- Given the algorithm for E, each time thru the loop x or y is cut in half
  - Every other time thru the loop both are cut in half
  - This means that the maximum number of times the loop will repeat is:
    - $2\log_2 \max(x,y)$ which is $O(\log_2 \max(x,y))$
    - But the input is encoded in binary, so the input length is also of the same order (technically $\log_2 x + \log_2 y$)
    - So, the number of total steps is of the same order as the input string, so it is O(n), where n is the length of the input string.

Chapter 7.3

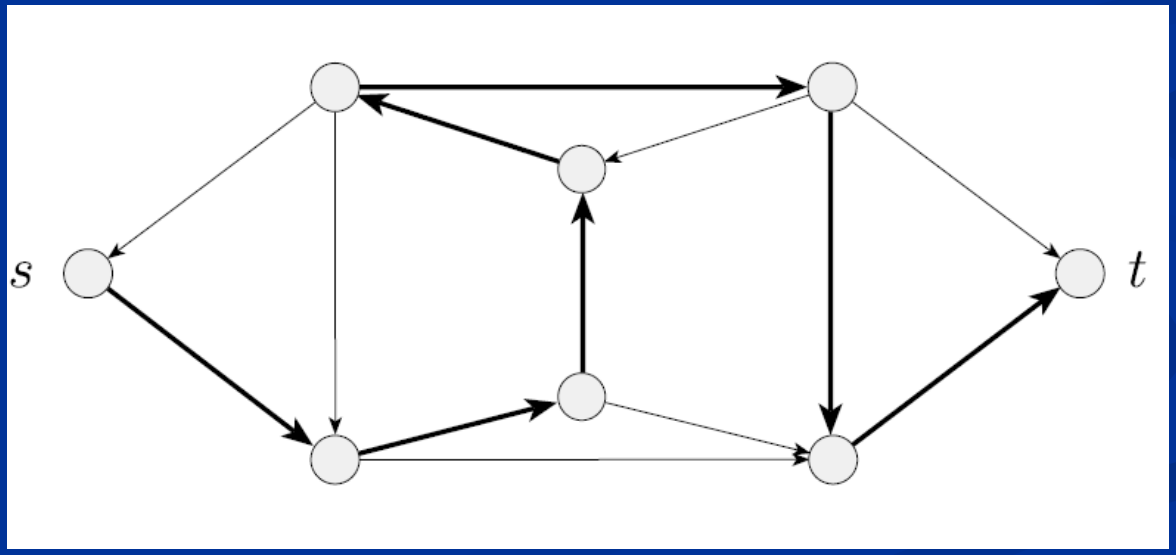
The Class NP

The Class NP

- We have seen that in some cases we can do better than brute force and come up with polynomial-time algorithms
- In some cases polynomial time algorithms are not known (e.g., Traveling Salesman Problem)
  - Do they exist but we have not yet found the polytime solution? Or does one not exist? Answer this and you will be famous (and probably even rich).
- The complexities of many problems are linked
  - If you solve one in polynomial time then many others are also solved

Hamiltonian Path Example

- A Hamiltonian path in a directed graph G is a directed path that goes thru each node exactly once
  - We consider the problem of whether two specific nodes in G are connected with a Hamiltonian path
  - HAMPATH = {<G,s,t>|G is a directed graph with a Hamiltonian path from s to t}



Brute Force Algorithm for HamPath

- Use the brute-force path algorithm we used earlier in this chapter
  - List all possible paths and then check whether the path is a valid Hamiltonian path.
  - How many paths are there to check for n nodes? Pick one node and start enumerating from there.
    - The total would be n! which is actually exponential
    - Of course can do better that this with just a little smarts: you know starting and ending node so (n-2)!

Polynomial Verifiability

- The HAMPATH problem does have a feature called polynomial verifiability
  - Even though we don't know of a fast (polytime) way to determine if a Hamiltonian path exists, if we discover such a path (e.g., with exponential brute-force method), then we can verify it easily (in polytime)
    - The book says we can verify if by just presenting it, which in this case means presenting the Hamiltonian path
    - Clearly you can verify this in polytime from the input
    - If you are not trying to be smart, how long will the check take you?
      - At worst $O(n^3)$ since path at most n long and graph has at most n^2 edges.
- Verifiability is often much easier than coming up with a solution

Another Polynomial Verifiable Problem

- A natural number is a composite if it is the product of two integers greater than 1
  - A composite number is not a prime number
  - COMPOSITES = $\{x | x = pq$, for integers $p,q > 1\}$
- Is it hard to check that a number is a composite if we are given the solution?
  - No, we must multiply the numbers. This is in polytime
- Interesting mathematical trivia:
  - Is there a brute-force method for checking primality of m?
    - Yes, see if x from 2 to n goes in evenly. Better: try 2 to square root of n.
  - Is there a polytime algorithm for checking primality?
    - Not when I took this course!
    - Proven in 2002 (AKS primality test, 2006 Gödel prize)

Some Problems are Not Polynomial Time Verifiable

- Consider HAMPATH', the complement of HAMPATH
- Even if we tell you there is not a Hamiltonian path between two nodes, we don't know how to verify it without going thru the same number of exponential steps to determine whether one exists

Definition of Verifier

- A verifier for a language A is an algorithm V, where:
  - A = {w|V accepts <w,c> for some string c}
    - A verifier uses additional information, represented by the symbol c, to verify that a string w is a member of A.
      - This information is called a certificate (or proof) of membership in A
  - We measure the time of the verifier only in terms of the length of w, so a polynomial time verifier runs in polytime of w.
    - A language A is polytime verifiable if it has a polytime verifier

Definition Applied to Examples

- For HAMPATH, what is the certificate for a string $\langle G,s,t \rangle \in$ HAMPATH?
  - Answer: it is the Hamiltonian path
- For the COMPOSITES problem, what is the certificate?
  - Answer: it is one of the two divisors
- In both cases we can check that the input is in the language in polytime given the certificate

Definition of NP

- NP is class of languages that have polytime verifiers
  - NP comes from Nondeterministic Polynomial time
  - Alternate formulation: nondeterministic TM accepts language
    - That is, if have nondeterminism (can always guess solution) then can solve in polytime
- The class NP is important because it contains many practical problems
  - HAMPATH and COMPOSITES $\in$ NP
  - COMPOSITES $\in$ P, but the proof is difficult
- Clearly if something is in P it is in NP. Why?
  - Because if you can find the solution in polytime then certainly can verify it in polytime (just run same algorithm)
  - So, P $\subseteq$ NP

Nondeterministic TM for HAMPATH

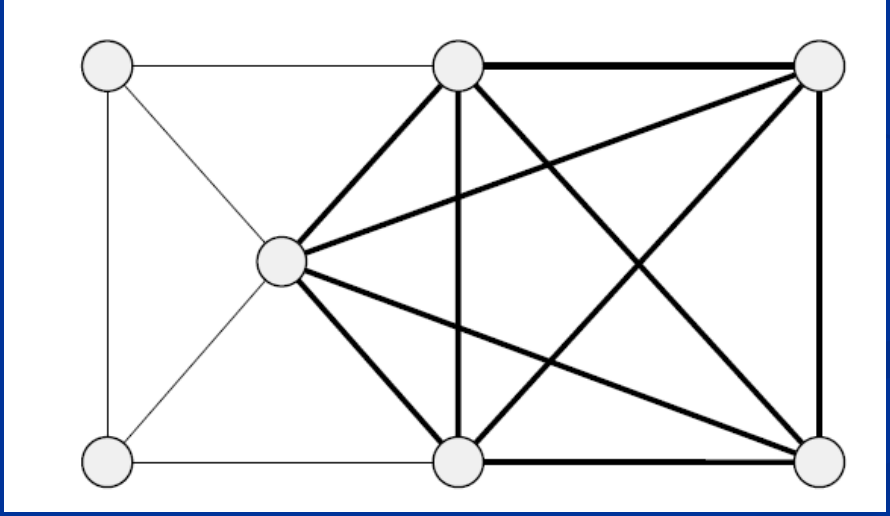
- Given input <G,s,t> and m nodes in G
  1. Write a list of m numbers, p_1, …, p_m. Each number is nondeterministically selected
  2. Check for repetitions in the list. If exists, reject.
  3. Check if s=p1 and t = pm. If either fails, reject.
  4. For each I between 1 and m-1, check that (p_i, p_{i+1}) is an edge in G. If not, reject; otherwise accept.
- Running time analysis:
  - In stage 1 the nondeterministic selection runs in polytime
  - Stages 2, 3, and 4 run in polytime.
  - So, the entire algorithm runs in nondeterministic polynomial time.

Definition: NTIME

- Nondeterministic time complexity, NTIME(t(n)) is defined similarly to the deterministic time complexity class TIME(t(n))
- Definitions:
  - NTIME(t(n)) = {L|L is a language decided by a O(t(n)) time nondeterministic Turing machine}
  - NP = $\cup_k$NTIME(n^k)
    - The k is under the union. NP is the union of all languages where the running time on a NTM is polynomial

Example of Problem in NP: CLIQUE

- Definitions:
  - A clique in an undirected graph is a subgraph, where every two nodes are connected by an edge.
    - A k-clique is a clique that contains k-nodes
    - The graph below has a 5 clique



The Clique Problem

- The clique problem is to determine whether a graph contains a clique of a specified size
  - CLIQUE: {<G,k>|G is an undirected graph with a k-clique}
  - Very important point not highlighted in the textbook:
    - Note that k is a parameter. Thus the problem of deciding whether there is a 3-clique or a 10-clique is not the CLIQUE problem
    - This is important because we will see shortly that CLIQUE is NP-complete but HW problem 7.9 asks you to prove 3-clique $\in$ P

Prove that CLIQUE ∈ NP

- We just need to prove that the clique is a certificate
- Proofs:
  - The following is a verifier V for CLIQUE:
    - V = On input <<G,k>, c>:
      1. Test whether c is a set of k nodes in G
      2. Test whether G contains all edges connected nodes in c
         1. This requires you to check at most k^2 edges
      3. If both pass, accept; else reject
  - If you prefer to think of NTM method:
    - N = On input <G, k> where G is a graph
      1. Nondeterministically select a subset c of k nodes of G
      2. Test whether G contains all edges connected nodes in c
      3. If yes, accept, otherwise reject
  - Clearly these proofs are nearly identical and clearly each step is in polynomial time (in second case in nondeterministic polytime to find a solution)

Example of Problem in NP: Subset-Sum

- The SUBSET-SUM problem:
  - Given a collection of numbers $x_1, \ldots, x_n$ and a target number t
  - Determine whether a collection of numbers contains a subset that sums to t
  - SUBSET-SUM = $\{<S,t> \mid S = \{x_1, \ldots, x_n\}$ and for some $\{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_n\}$, we have $\sum y_i = t\}$
    - Note that these sets are actually multi-sets and can have repeats
  - Does $<\{4,11,16,21,27\}, 25> \in$ SUBSET_SUM?
    - Yes since 21+4 = 25

Prove that SUBSET-SUM ∈ NP

- Show that the subset is a certificate
- Proof:
  - The following is a verified for SUBSET-SUM
    - V = On input <<S,t>,c>:
      1. Test whether c is a collection of numbers that sum to t
      2. Test whether S contains all the numbers in c
      3. If both pass, accept; otherwise, reject.
    - Clearly in polynomial time

Class coNP

- The complements of CLIQUE and SUBSET-SUM are not obviously members of NP
  - Verifying that something is not present seems to be more difficult than verifying that it is present
  - The class coNP contains the languages that are complements of languages in NP
  - We do not know if coNP is different from NP

The P Versus NP Question

- Summary:
  - P = the class of languages for which membership can be *decided* quickly
  - NP = the class of languages for which membership can be *verified* quickly
- I find the book does not explain this well
  - In P, you must be able to essential determine whether a certificate exists or not in polynomial time
  - In NP, you are given the certificate and must check it in polynomial time

P Versus NP cont.

- HAMPATH and CLIQUE $\in$ NP
- We do not know if they belong to P
- Verifiability seems much easier than decidability, so we would probably expect to have problem in NP but not P
- No one has ever found a single language that has proved to be in NP but not P
- We believe P $\neq$ NP
  - People have tried to prove that many problems belong to P (e.g., Traveling Salesman Problem) but have failed
  - Best methods for solving some languages in NP deterministically use exponential time
  - NP $\subseteq$ EXPTIME = $\cup_k$TIME(2^{n^k})
    - We do not know if NP is contained in a smaller deterministic time complexity class

Chapter 7.4

NP-Completeness

NP-Completeness

- An advance in P vs. NP came in 1970's
  - Certain problems in NP are related to that of the entire class
  - If a polynomial time algorithm exists for any of these problems, then all problems in NP would be polynomial time solvable (i.e., then P = NP)
  - These problems are called NP-complete
  - They are the hardest problems in NP
    - because if they have a polytime solution, then so do all problems in NP

Importance of NP Complete

- Theoretical importance of NP-complete
  - If any problem in NP requires more than polytime, then so do the NP-complete problems.
  - If any NP-complete problem has a polytime solution, then so do all problems in NP
- Practical importance of NP-Complete
  - Since no polytime solution has been found for an NP-complete problem, if we determine a problem is NP-complete it is reasonable to give up finding a general polytime solution

Satisfiability: An NP-complete Problem

- Background for Satisfiability Problem
  - Boolean variables can take on values of TRUE or FALSE (0 or 1)
  - Boolean operators are $\wedge$ (and), $\vee$ (or) and $\neg$ (not)
  - A Boolean formula is an expression with Boolean variables and operators
    - A Boolean formula is satisfiable if some assignment of 0s and 1s to the variables makes the formula evaluate to 1 (TRUE).
    - Example: $(\neg x \wedge y) \vee (x \wedge \neg z)$.
      - This is satisfiable in several ways, such as x=0, y=1, z=0

Cook-Levin Theorem

- The Cook-Levin Theorem links the complexity of the SAT problem to the complexities of all problems in NP
  - SAT is essentially the hardest problem in NP since if it is solved in P then all problems in NP are solved in P
  - We will need to show that solution to SAT can be used to solve all problems in NP
- Theorem: SAT $\in$ P iff P = NP

Polynomial Time Reducibility

- If a problem A reduces to problem B, then a solution to B can be used to solve A
  - Note that this means B is at least as hard as A
    - B could be harder but not easier. A cannot be harder than B.
- When problem A is efficiently reducible to problem B, an efficient solution to B can be used to solve A efficiently
  - "Efficiently reducible" means in polynomial time
  - If A is polytime reducible to B, we can convert the problem of testing for membership in A to a membership test in B
  - If one language is polynomial time reducible to a language already known to have a polynomial time solution, then the original language will have a polynomial time solution

Polynomial Time Reducibility cont.

- Note that we can chain the languages
  - Assume we show that A is polytime reducible to B
  - Now we show that C is polytime reducible to A
  - Clearly C is polytime reducible to B
  - We can build up a large class of languages such that if one of the languages has a polynomial time solution, then all do.

3SAT

- Before demonstrating a polytime reduction, we introduce 3SAT
  - A special case of the satisfiability problem
  - A literal is a Boolean variable or its negation
  - A clause is several literals connected only with $\vee$s
  - A Boolean formula is in conjunctive normal form (cnf) if it has only clauses connected by $\wedge$s
  - A 3cnf formula has 3 literals in all clauses
  - Example:
    - $(x1 \vee \neg x2 \vee \neg\ x3) \wedge (x3 \vee \neg\ x5 \vee x6) \wedge (x3 \vee \neg\ x6 \vee x4) \wedge (x4 \vee x5 \vee x6)$
  - Let 3SAT be the language of 3cnf formulas that are satisfiable
    - This means that each clause must have at least one literal with a value of 1

Polytime Reduction from 3SAT to CLIQUE

- Proof Idea: convert any 3SAT formula to a graph, such that a clique of the specified size corresponds to satisfying assignments of the formula
  - The structure of the graph must mimic the behavior of the variables in the clauses
- How about you take a try at this now, assuming you have not seen the solution. Try the example on the previous page.

Polytime Reduction Procedure

- Given a 3SAT formula create a graph G
  - The nodes in G are organized into k groups of three nodes (triples) called $t_1, \ldots, t_k$
  - Each triple corresponds to one of the clauses in the formula and each node in the triple is labeled with a literal in the clause
  - The edges in G connect all but two types of pairs of nodes in G
    - No edge is between nodes in the same triple
    - No edge is present between nodes with contradictory labels (e.g., x1 and ¬x1)
  - See example on next slide

The Reduction of 3SAT to CLIQUE

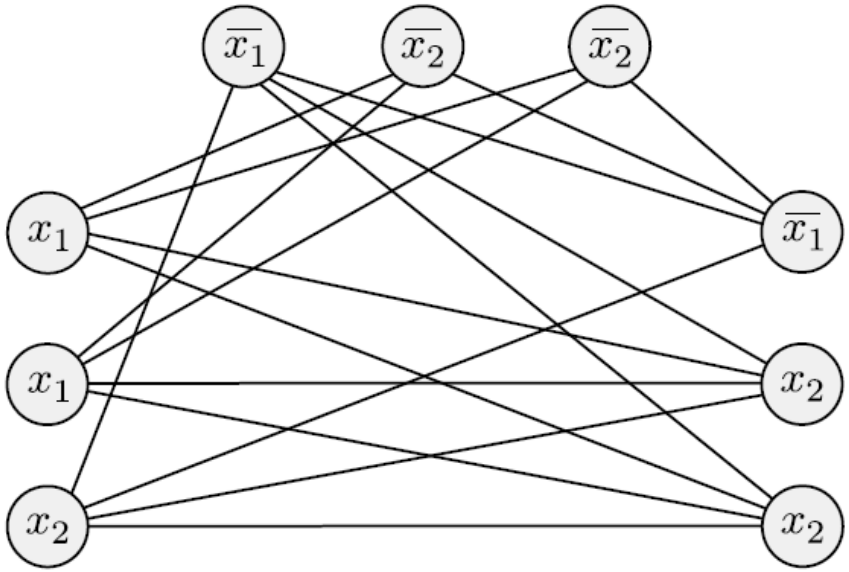


FIGURE 7.33
The graph that the reduction produces from
$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$

Formula is satisfiable if graph has a k-clique (k=3 for the 3 clauses)

If CLIQUE solvable in polytime so is 3SAT

Why does this reduction work?

- The satisfiability problem requires that each clause has at least one value that evaluates to true
  - In each triple in G we select one node corresponding to a true literal in the satisfying assignment
    - If more than one is true in a clause, arbitrarily choose one
    - The nodes just selected will form a k-clique
      - The number of nodes selected is k, since k clauses
      - Every pair of nodes will be connected by an edge because they do not violate one of the 2 conditions for not having an edge
  - This shows that if the 3cnf formula is satisfiable, then there will be a k-clique
  - The argument the other way is essentially the same
    - If k-clique then all in different clauses and hence will be satisfiable, since any logically inconsistent assignments are not represented in G

Definition of NP-Completeness

- A language B is NP-complete it if satisfies two conditions:
  1. B is in NP and
  2. Every A in NP is polynomial time reducible to B
  - Note: in a sense NP-complete are the hardest problems in NP (or equally hard)
- Theorems:
  - if B is NP-complete and B $\in$ P, then P = NP
  - If B is NP-complete and B is polytime reducible to C for C in NP, then C is NP-complete
    - Note that this is useful for proving NP-completeness.
    - All we need to do is show that a language is in NP and polytime reducible from any known NP-complete problem
    - Other books use the terminology NP-hard. If an NP-complete problem is polytime reducible to a problem X, then X is NP-hard, since it is at least as hard as the NP-complete problem, which is the hardest class of problems in NP. We then show X $\in$ NP to show it is NP-complete. Note that it is possible that X $\notin$NP (in which case it is not NP-complete).

The Cook Levin Theorem

- Once we have one NP-complete problem, we may obtain others by polytime reduction from it.
  - Establishing the first NP-complete problem is difficult
  - The Cook-Levin theorem proves that SAT is NP-complete
    - To do this, we must prove than every problem in NP reduces to it
    - This proof is 5 pages long (page 277 – 282) and quite involved. Unfortunately we don't have time to go thru it in this class.
      - Proof idea: we can convert any problem in NP to SAT by having the Boolean formula represent the simulation of the NP machine on its input
      - Given the fact that Boolean formulas contain the Boolean operators used to build a computer, this may not be too surprising.
    - We can do the easy part, though. The first step to prove NP-complete is to show the language $\in$ NP. How?
      - Answer: guess a solution (or use a NTM) and can easily check in polytime.

Reminder about CLIQUE Problem

- Definitions:
  - A clique in an undirected graph is a subgraph, where every two nodes are connected by an edge.
    - A k-clique is a clique that contains k-nodes
- The clique problem is to determine whether a graph contains a clique of a specified size
  - CLIQUE: {<G,k>|G is an undirected graph with a k-clique}
  - Very important point not highlighted in the textbook:
    - Note that k is a parameter. Thus the problem of deciding whether there is a 3-clique or a 10-clique is not the CLIQUE problem
    - This is important because we will see shortly that CLIQUE is NP-complete but HW problem 7.9 asks you to prove 3-clique $\in$ P

Proving CLIQUE NP-complete

- To prove CLIQUE is NP-complete we need to show:
  - Step 1: CLIQUE is in NP
  - Step 2: An NP-Complete problem is polytime reducible to it
    - The book proved SAT is NP-complete and also that 3SAT is NP-complete, so we can just use 3SAT
    - We need to prove that 3-SAT is polytime reducible to CLIQUE
      - We already did this!
- Step 1: CLIQUE is in NP because its certificate can easily be checked in polytime
  - Given a certificate, we need to check that there is an edge between all k nodes in the clique
  - Simply look at each node in turn an make sure it has an edge to each of the k-1 other nodes. Clearly this is $O(n^2)$ at worst (assuming k=n).
- When trying to prove a problem NP-complete, you can "use" any problem you know is already NP-complete

The Hard Part

- The hard part is always coming up with the polynomial time reduction
  - The one from 3SAT to CLIQUE was actually not that bad
  - Experience and seeing lots of problems can help
  - But it certainly can be tricky
  - Chapter 7.5 has several examples. All are harder than the reduction of 3SAT to CLIQUE
    - We will do one of them
  - You are responsible for knowing how to prove CLIQUE NP-complete

Chapter 7.5

Additional NP-Complete Problems

The Vertex-Cover Problem

- If G is an undirected graph, a vertex cover of G is a subset of the nodes where every edge of G touches one of the nodes.
- The vertex cover problem asks whether a graph contains a vertex cover of a specified size
  - I think if we just ask if it has a vertex cover or not, w/o specifying a size, that is easy to answer. Why?
    - Just select all nodes in G. Since an edge must be between two nodes, it must be a vertex cover. At worst we just check that we have a valid graph.

Vertex Cover is NP-complete

- What is the first thing we must do?
  - Show that Vertex Cover $\in$ NP
- How do we do this?
  - A certificate for a k-vertex cover is a set of k nodes.
  - We can verify this is a valid vertex-cover in polytime
- Now we need to show that an NP-complete problem can be reduced to vertex-cover in polytime
  - What would you choose?
    - 3SAT is a reasonable choice
    - Will anything else work?
      - Assuming VERTEX-COVER is NP-complete, all other NP-complete problems will have a polytime reduction to it. But the reduction may be difficult

Reduction from 3SAT to Vertex Cover

- We start with a 3CNF expression, expressed as:
  - $U = (u_{1,1} \vee u_{1,2} \vee u_{1,3}) \wedge \ \ldots \wedge (u_{m,1} \vee u_{m,2} \vee u_{m,3})$
  - Where we have m clauses over the set $V = \{x_1, ..., x_n\}$ Boolean variables (i.e., n variables)
  - The reduction will work so that the vertex cover is for $k = n + 2m$
    - That is, k equals the number of variables plus 2 times the number of clauses
  - Now let us work on the reduction

Reduction continued

- First we need to decide on what the nodes in the graph we construct will represent
  - Imagine we have a CNF expression such as:
    - $(x1 \vee x1 \vee x2) \wedge (\neg x1 \vee \neg x2 \vee \neg x2) \wedge (\neg x1 \vee x2 \vee x2)$
  - What might we label the each node with?
    - Probably makes sense to use the Boolean variables with both possible truth values
    - So in the above example we will have nodes labeled as x1, $\neg$x1, x2 and $\neg$x2
    - We still need to decide how to arrange them and then how to connect them with edges

Reduction continued

- We need to think about making consistent variable assignments and also dealing with the clause structure
- We will start with the consistent variable assignments
- For each Boolean variable, add it to the graph and connect it by an edge to its complement
- This leads to the following graph for our example

Reduction continued

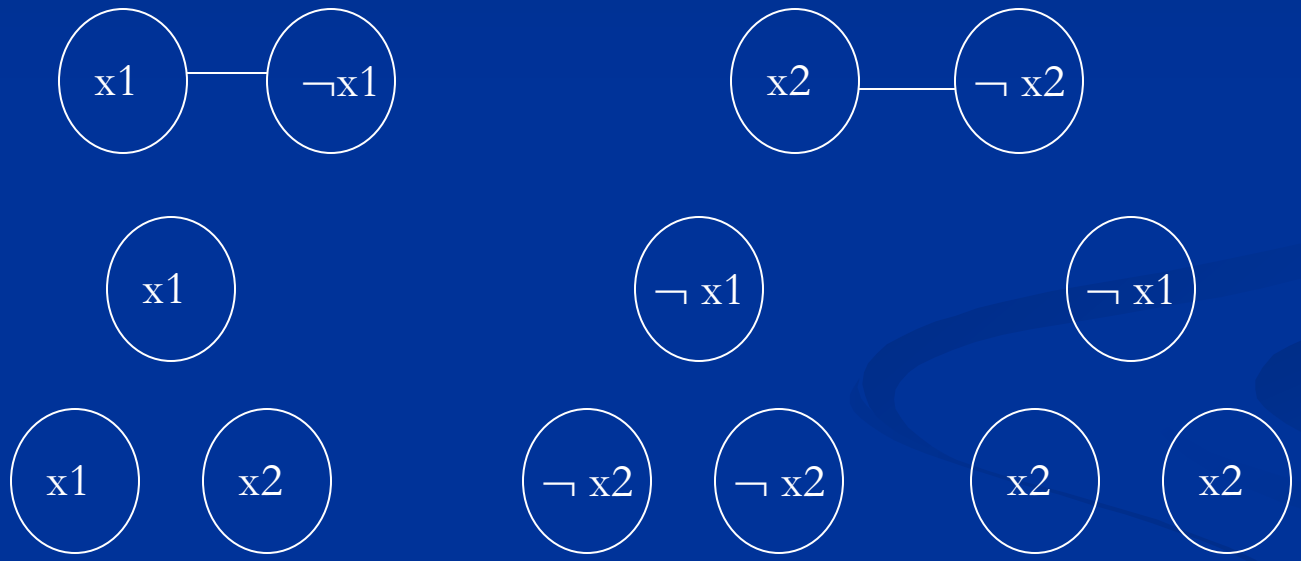
3CNF formula: $(x1 \vee x1 \vee x2) \wedge (\neg x1 \vee \neg x2 \vee \neg x2) \wedge (\neg x1 \vee x2 \vee x2)$

Idea: we will pick k for the vertex cover so that one has to pick the vertices very carefully. We will pick k so that we must limit the nodes in this part of the graph to n nodes, given n Boolean variables. Thus, for the partial graph above, we can only pick two nodes, which corresponds to a variable assignment.

But now we need to add the clauses in. Lets start by putting the vertices together to mimic the clauses

Reduction continued

3CNF formula: $(x1 \vee x1 \vee x2) \wedge (\neg x1 \vee \neg x2 \vee \neg x2) \wedge (\neg x1 \vee x2 \vee x2)$

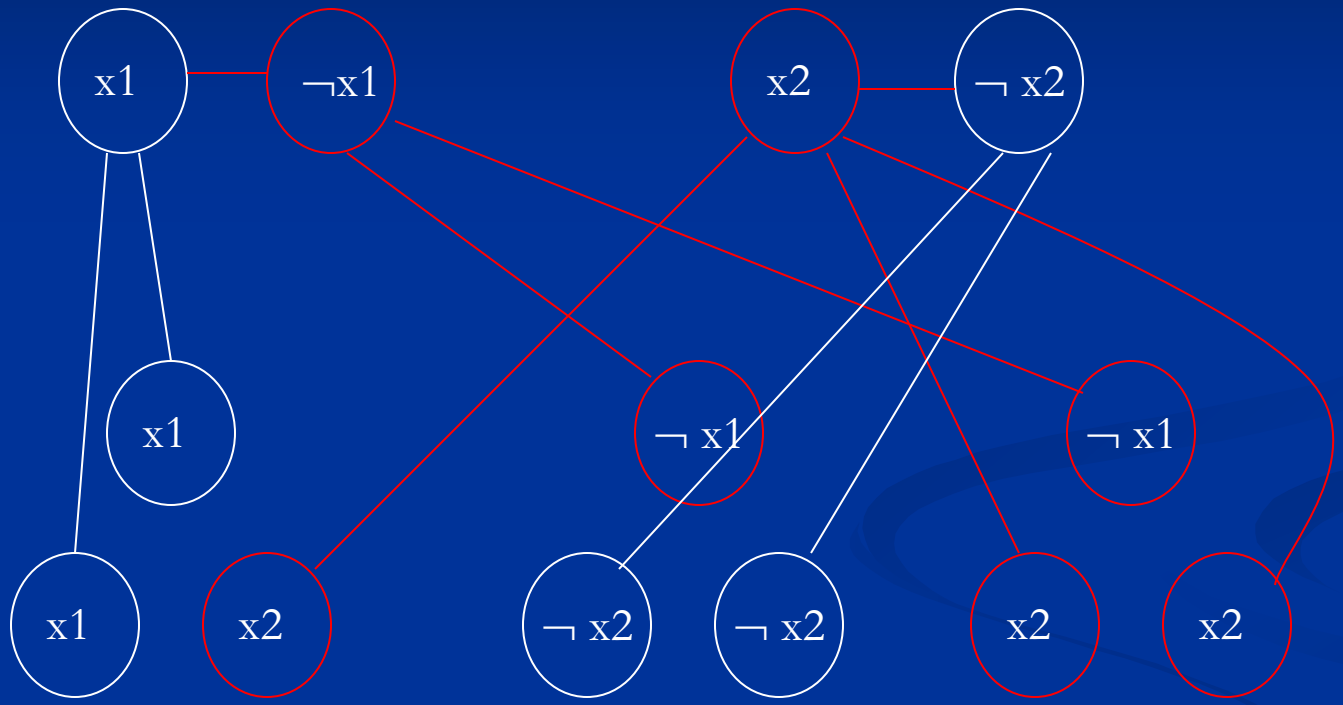


Now what? Any idea what nodes you might want to connect to begin with?

Hint: how about indicating what variables in the clauses are satisfied given the variable assignments at top

Reduction continued

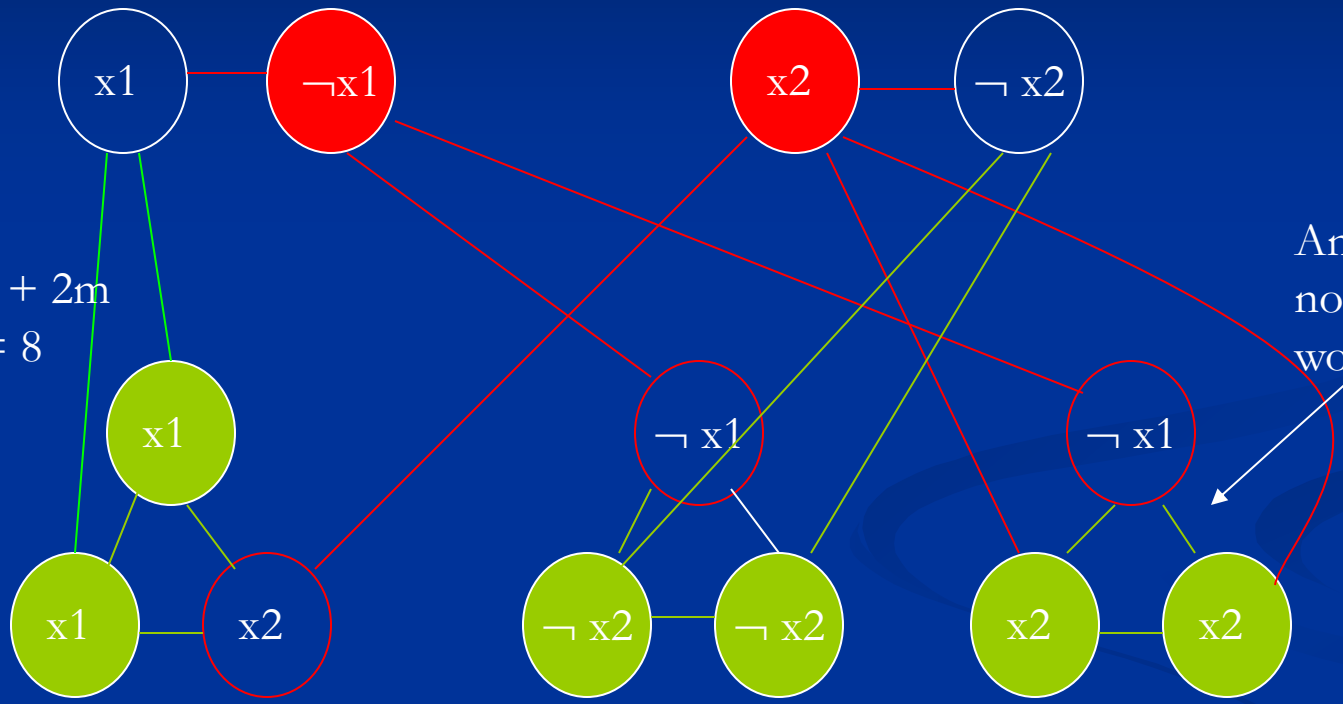
3CNF formula: (x1 ∨ x1 ∨ x2) ∧ (¬x1 ∨ ¬ x2 ∨ ¬ x2) ∧ (¬ x1 ∨ x2 ∨ x2)



Now where are we? If we pick good variable assignments, then at least one variable in each clause is satisfied. That is good, but it is not a vertex cover problem yet. Note that x1=False and x2=True satisfies the formula but does not cover every edge.

Reduction Continued

3CNF formula: $(x1 \vee x1 \vee x2) \wedge (\neg x1 \vee \neg x2 \vee \neg x2) \wedge (\neg x1 \vee x2 \vee x2)$



Note: k = n + 2m
= 2 + 2x3 = 8

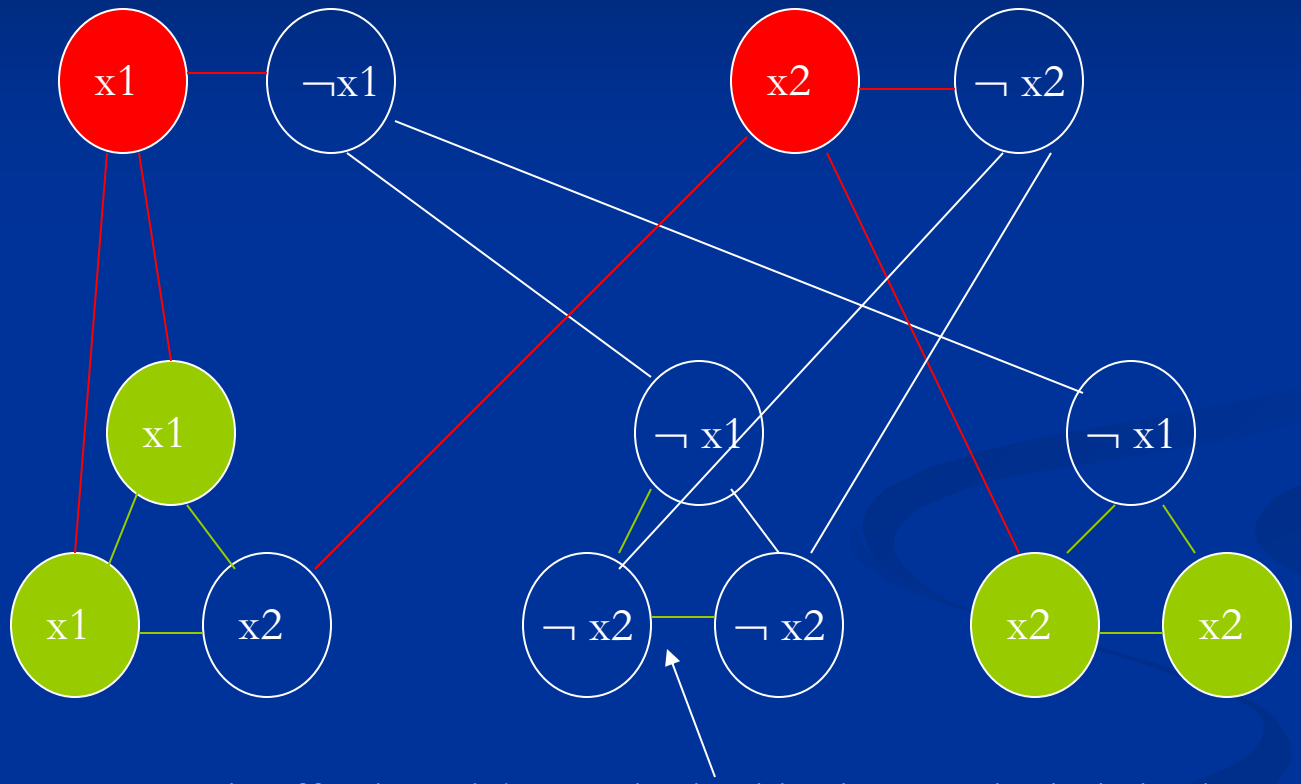
Any 2 of the 3 nodes here will work

If we link all 3 variables in each clause, now we can cover all edges if we require that the variables in the clause not associated with the satisfied variables are included in the vertex cover. This covers the edges in the clause and the ones to the top vertices. This only works if the 3CNF was satisfiable

Reduction Continued

- We will not formally prove this, but if the original 3CNF expression is satisfiable then there will be a vertex cover and only in this case.
  - First, if we pick variable assignments for the top such that they satisfy the 3CNF formula, then we can satisfy the covering by picking the clause variables that cover the edges not linked to the satisfied variables
    - Based on the way we constructed things, this will be a vertex cover
      - So, 3SAT ➔ Vertex cover
    - If it is not 3SAT, then there will be no vertex cover
      - If it is not satisfiable, then one of the clauses would require 3 nodes to be marked, not 2 and to handle the value of k, we can only mark two of the clause nodes. See the following graph:

Reduction Continued



We can only afford to pick 2 nodes in this clause to include in the cover. Any two will cover the 3 edges connecting the clauses. However, each of the 3 nodes is connected by an edge to unsatisfied variables at the top. Whichever node is not picked will have that associated edge uncovered

Vertex Cover NP-Complete

- So Vertex Cover is NP-Complete
  - It is in NP
  - An NP-Complete problem, 3-SAT, is reducible to it