Chapter 34
NP-Completeness

The slides for this course are based on the course textbook: Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms*, 2nd edition, The MIT Press, McGraw-Hill, 2001.

• Many of the slides were provided by the publisher for use with the textbook. They are copyrighted, 2001.

• These slides are for classroom use only, and may be used only by students in this specific course this semester. They are NOT a substitute for reading the textbook!

Chapter 34 Topics

- Polynomial time
- Polynomial-time verification
- NP-completeness and reducibility
- NP-completeness proofs
- NP-complete theorems

Computational Complexity

- Algorithmics is the study of the performance characteristics of specific *algorithms*.
- Computational Complexity (also called Computability Theory) looks at the characteristics of *problems*, rather than algorithms.

The Turing Machine

Alan Mathison Turing, b. 1912, d. 1954. Contributed much to the foundations of computing theory.

Published the Turing machine model in 1937.

Church-Turing Thesis - “Any algorithmic procedure that can be carried out by a human, a team of humans, or a machine can be carried out by some Turing machine.”

Unproveable, because we don’t have a precise definition of what “algorithmic procedure” means, but generally accepted as true.

Puts a limit on what can be computed.

The Church-Turing Thesis

- No model of digital computation is more powerful than a Turing machine.
- By “more powerful,” we mean “can recognize languages that a TM cannot recognize.”
- This is not something that can be proved. But everybody believes it because no one has been able to devise a more powerful model of computation.

Solvable and Unsolvable

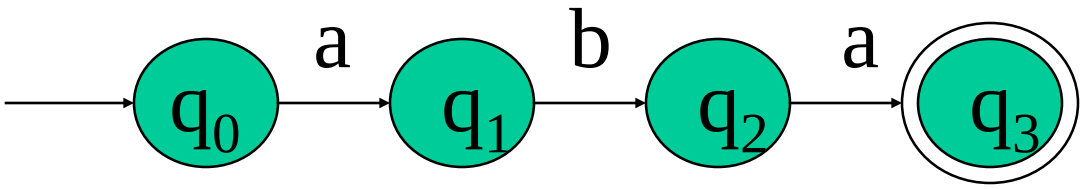
- We make a distinction between **solvable** and **unsolvable** problems. Within the solvable problems, we make a three-way distinction. So the subdivision looks like this:
- ***Unsolvable problems***
- ***Solvable problems***
  - **Provably intractable** (*infeasible*) problems
  - **Probably intractable** problems
  - **Tractable** problems (*feasible* problems)

Unsolvable Problems

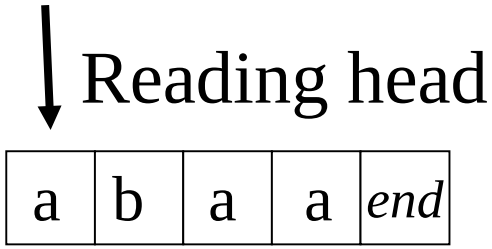
- Some problems exist for which it can be shown that no algorithm exists for solving them. Such problems are called **unsolvable**.

- The first problem proven to be unsolvable was the ***halting problem***. This problem takes as its input a program together with input to this program. The problem asks whether a Turing Machine running the program will halt when executed with the input to the program.

Finite automaton



Finite-state controller



Finite tape with input string.
The tape is read from left to right; no going back.
The Finite Automaton has *no auxiliary memory*.

Definition: Turing Machine

A Turing machine (TM) is a 7-Tuple

$T = (Q, \Sigma, \Gamma, \delta, q_0, \Delta, F)$, where

Q = a finite set of states not containing *h* (the *halt* state)

Σ = a finite set of symbols constituting the *input alphabet*;

$\Sigma \subseteq \Gamma - \{\Delta\}$

Γ = a finite set of symbols constituting the *tape alphabet*

δ = the transition function: $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$

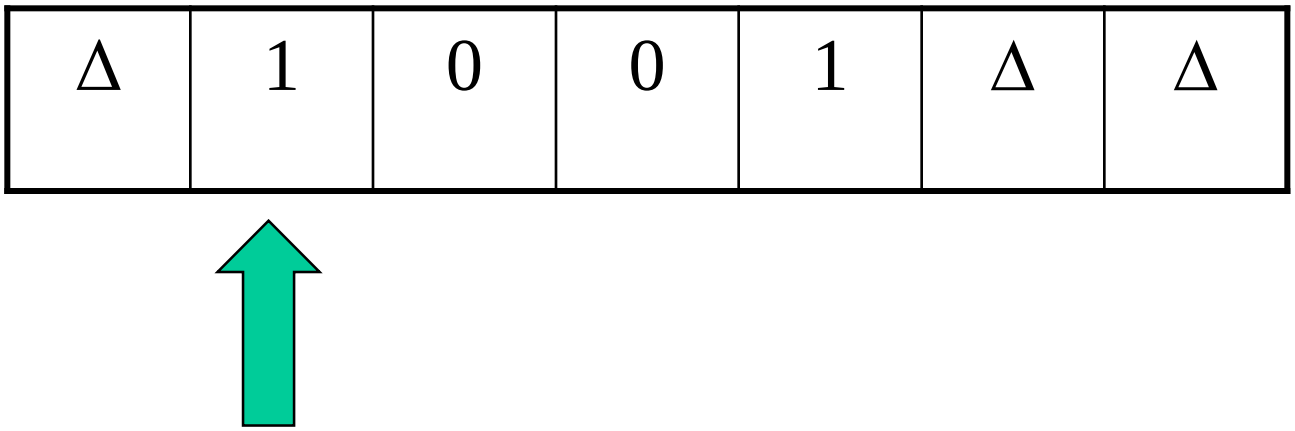
q_0 = the start state ($q_0 \in Q$)

Δ is a special symbol, the *blank* symbol

$F \subseteq Q$ is the set of final states

Definition: Turing Machine

Turing machines can be thought of as a *finite state automaton* with slightly different labels on the arcs, plus a *tape* as auxiliary memory, and a *tape head* (or read-write head).



Definition: Turing Machine

The tape on a Turing machine can be thought of as a linear structure marked off into squares or cells. It is usually defined as infinite in both directions, but may be thought of as bounded on the left side, but infinite on the right.

Each cell on the tape on a Turing machine can hold a symbol from the input alphabet, a symbol from the tape alphabet (which may include some symbols which are not in the input alphabet), or a blank symbol Δ which is distinct from the tape and input alphabet. The cells are numbered, starting with 0 in the left-most cell.

Standard Turing Machine

The standard Turing machine has several features:

The TM has a tape that is *unbounded in both directions*.

The TM is *deterministic* in the sense that δ defines at most one move per configuration

There is no special input file; the input is copied onto the tape.

There is no special output device; the tape contains the output, if any.

The TM tape head

- The Turing machine has a tape head which reads from and writes to the tape during each move.
- Initially, the tape contains all Δ's (blanks).
- The tape head can move right or left, or may stay in the same position.
- The tape head can read a character from the cell under which it is positioned, or it may write a character to the cell.

The TM transition function

The TM transition function has the following form:

$$\delta(q, X) = (r, Y, D)$$

This says that when the TM is in state *q*, and the read-write head reads a certain symbol (X) off the tape, then the TM will change state to state *r*, write a Y onto the tape, and move in direction D.

For example, $\delta(q_1, b) = (q_2, \Delta, R)$ means: “We are currently in state q_1; read the cell; it’s a *b*. OK, change state to q_2, write a blank, and move right.”

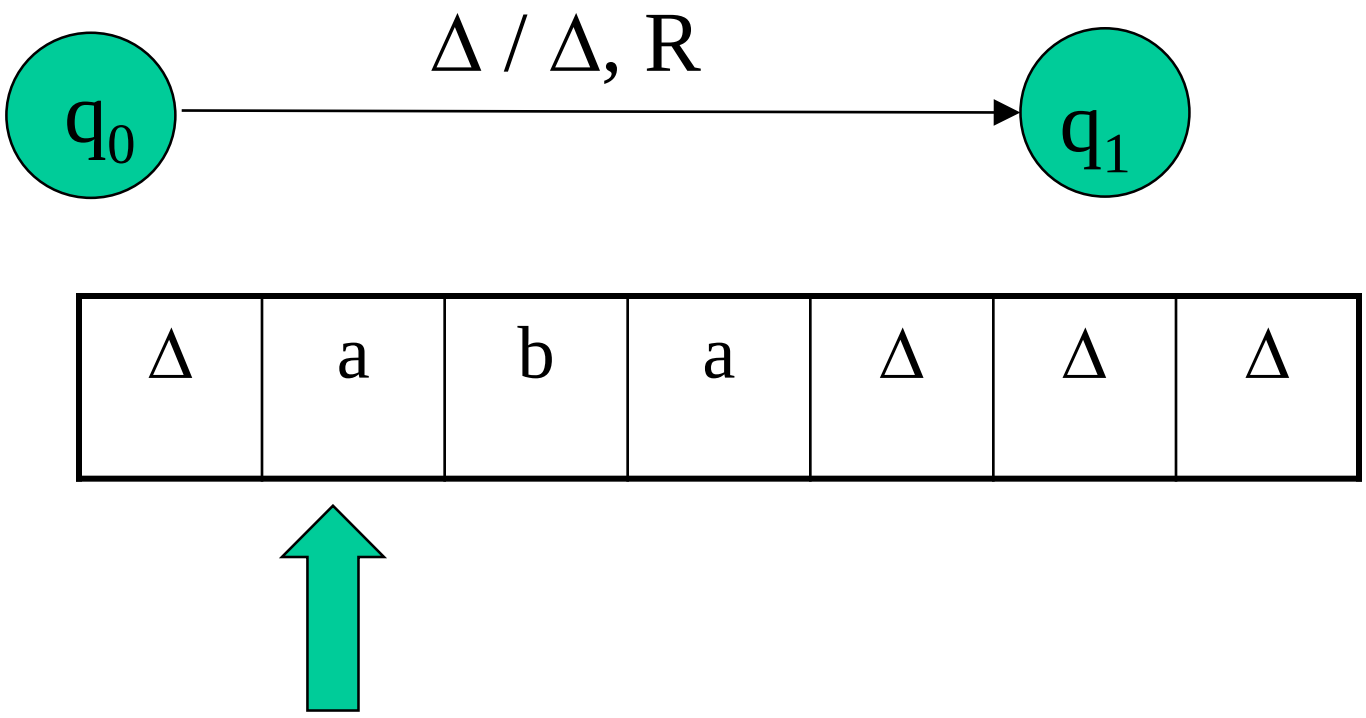
The TM transition function

The TM transition function can be represented in the form of a table:

Current state	Current symbol	New state	New symbol	Direction to move
q0	a	q1	Δ	R
q1	b	q2	a	L

The TM transition function

•We can label the arcs of a finite state machine to indicate the moves of the TM:



Processing a string

To process a string, we place the string onto the tape, where it can be read by the read-write head. By convention, we leave a blank in cell 0, and begin with the left-most character of the string in cell 1, etc.

Once the string is on the tape, then processing can begin. We don’t need a separate input string after that.

Accepting a string

There is only one way that a string may be accepted by a TM: If a move causes the machine to move into the halt state, then we stop and accept the string. (There are no moves out of a halt state.)

This may occur even when we haven't finished processing the string yet.

For example, if our TM is processing all the strings that have *aabb* in them somewhere, the TM may halt after processing the 6th character of this string:

bb*aabb*aaaabbbb

Crashing and Halting

There are 2 ways a string may fail to be accepted by a TM:

1. Crashing
   a. If a symbol is found on the tape for which the transition function is undefined, and the current state is not the halt state, then we crash.
   b. If our TM has a bounded left end, and a move is specified which causes the machine to move off of the left end of the tape, then we crash.
2. Looping
   a. If an input string causes the TM to enter an infinite loop, then we loop forever.

If the TM crashes, the string is *explicitly rejected*.

Model of computation

- A Turing Machine is an *abstract model* of a computer that lets us define in a precise, mathematical way what we mean by computation.
- We use the concept of a “configuration” to define what we mean by computation.

Configuration

A configuration of a TM is a pair,

(q,x<u>a</u>y)

where *q* is a state, *x* and *y* are strings, and *a* is the symbol at the current position of the tape head.

We sometimes use

(q,x<u>w</u>)

To indicate that the tape head is under the first symbol in the string *w*.

Configuration

- A configuration of a TM includes everything we need to know to continue a computation
  - current state
  - symbol currently under the tape head
  - string on tape to left of the head
  - string on tape to right of the head (up to rightmost non-blank symbol)
- Example

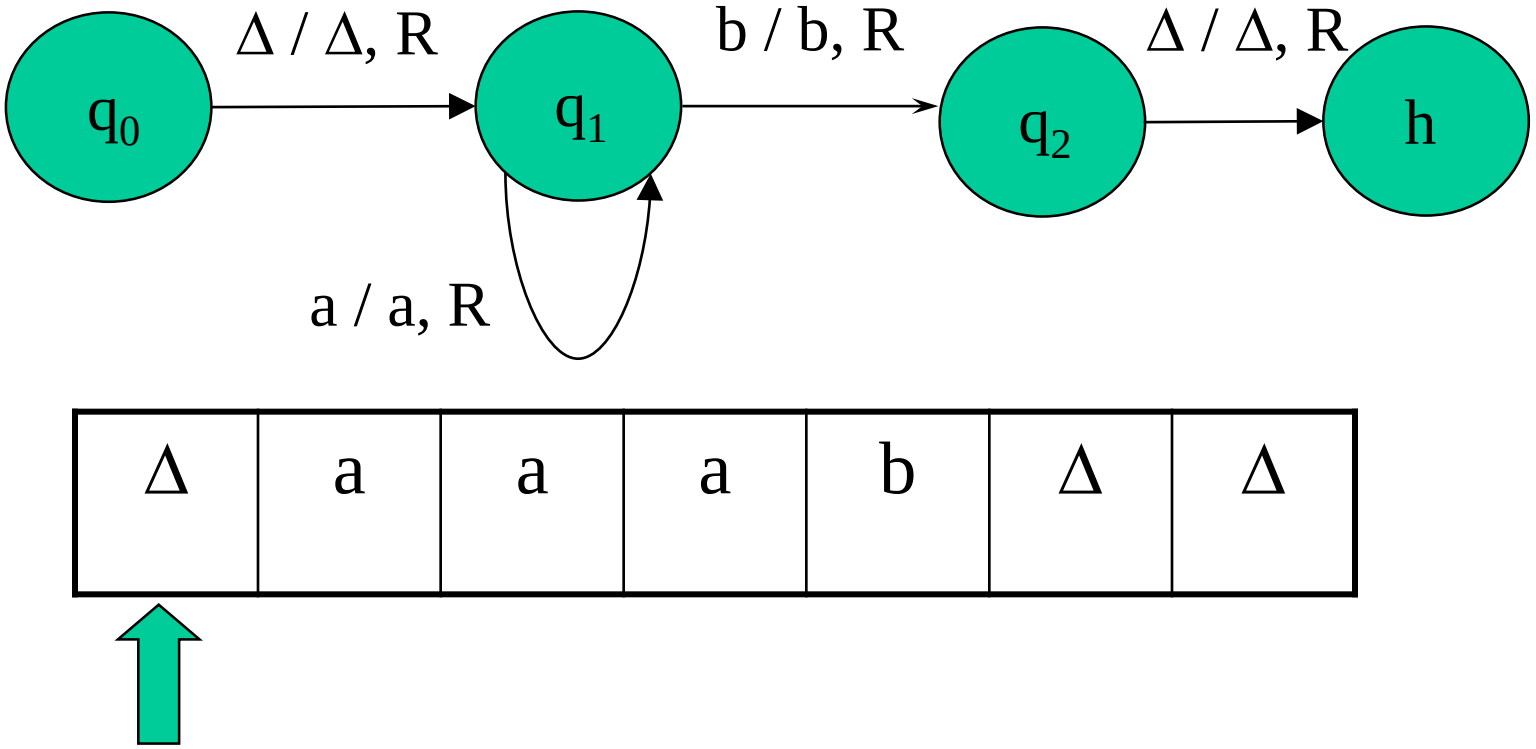
  (q, ∆bb<u>a</u>bbba∆)

Computation

- A computation is a sequence of configurations such that each configuration is obtained from the previous configuration by some transition of the TM.

Example:

Here is the TM to process the language consisting of strings containing any number of *a*'s, with a single *b* at the end.



Crashing versus looping

In a standard TM, strings which are *not accepted* can cause the TM to:

- crash if it enters a state where there are no transitions for the current tape symbol,
- crash if it tries to move left at the left end of the tape, or
- get into an infinite loop,

We can tell that a TM has rejected a string by crashing.

We cannot tell if a TM has rejected a string by looping indefinitely!

The Halting Problem

Given a TM, T, and a string, *w*, will T halt while processing *w*?

Technically, the halting problem is the problem of deciding, for any TM and input, whether the TM halts on that input.

Equivalently, it is the problem of writing an algorithm HALT(T, *w*) that takes as arguments the description of a TM, T, and its input, *w*, and returns 1 if T halts on *w*, and 0 otherwise.

The Halting Problem

Suppose our TM, T, is given a series of strings to process, and the first string is string *w*. Unfortunately, *w* is not $\in$ L(T), and T starts off on an infinite loop to reject it. When should we decide that the TM is going to reject the string? After step 100? 1K? 1M?

Obviously, we don't know how long we have to wait. We could be sitting there forever, thinking, "Maybe the TM will recognize this string on the very next move . . ."

The Halting Problem

So this TM will never halt while processing *w* - but the UTM can't predict that fact in advance.

The poor UTM is sitting there, waiting for T to finish processing *w*, so it can go on to the second string. T is never going to stop, because it is in an infinite loop, but at any given time the UTM can't distinguish that situation from one in which T is going to accept *w* on the very next move. All it can do is wait, and wait, and wait. It is never going to print a 1, but it doesn't know that, so it's never going to print a 0, either.

Obviously, this is an unsolvable problem.

Undecidable problems

- There are many other problems which have been proven to be uncomputable (and thus, according to the Church-Turing thesis, unsolvable).
- For example:
  - Are two context-free grammars equivalent?
  - Is a given CFG ambiguous?
  - Does a TM meet its specification – that is, does it have any “bugs”?
  - Does an arbitrary TM halts on all inputs – that is, does it contain any infinite loops?

Solvable problems

- Now we want to take a look at the class of ***solvable problems***:
  - **Provably intractable** (*infeasible*) problems
  - **Probably intractable** problems
  - **Tractable** problems (*feasible* problems)

Solvable Problems

- **Provably intractable** (*infeasible*) problems
  Example: Generalized chess - Chess is played on a board of size 8 x 8; generalized chess is played on a board of size N x N, with the set of pieces and the set of moves adjusted as needed. It has been proven to have an exponential lower bound.
- **Probably intractable** problems
  Example: Any NP problem, such as Traveling Salesman
- **Tractable** problems (*feasible* problems)
  Example: Any P problem, such as Sorting

Provably intractable Problems

Provably intractable (*infeasible*) problems are problems for which it can be proven that no polynomial-time algorithm can possibly solve the problem.

Suppose we have a set of items and we want to list all possible permutations (a permutation is an *ordered arrangement*) of these items. Can we do this in polynomial time?

No.

In fact, we can prove that this will take us factorial time.

Provably intractable Problems

Consider a 3-element set, S = {a, b, c}.

Then P(3,3), the set of all 3-letter permutations of S, would be:

{abc, acb, bac, bca, cab, cba}.

There are 3 * 2 * 1 = 6 elements in P(3,3).

This is n!, where n = 3.

Similarly, P(4,4) = 24, or 4!, and P(5,5) = 5!, etc.

Each of the permutations occurs once in the list, so obviously it is going to take O(n!) time to list them all.

Tractable Problems

Tractable problems are problems for which we can prove that a polynomial-time solution exists.

The *sorting problem* is a problem for which we can show that a polynomial-time algorithm exists.

In our theoretical discussion of sorting at the beginning of chapter 8, we worked through a proof based on decision trees that showed that any comparison-based sort must take at least $\Omega(n \lg n)$ time.

Tractable Problems

We also established an upper bound of O(n lg n) for comparison-based sorts by:

(1) demonstrating that several sorts (e.g., Merge Sort, Heap Sort, Quicksort) exist that take no more than O(n lg n) time, and

(2) giving a proof-of-correctness (based on the loop invariant) for these sorts.

(For Merge Sort see pp. 30-31; for Heap Sort, see Exercise 6.4-2 on p. 136).

Solvable Problems

- **Provably intractable** (*infeasible*) problems
- **Probably intractable** problems
- **Tractable** problems (*feasible* problems)

The rest of this chapter examines the interesting class of ***probably intractable*** problems.

We sometimes refer to this class of problems as the *NP* class.

Does P = NP?

This chapter concentrates on a question that has been unanswered despite much research:

Does P = NP?

Some related questions:

- What are the meanings of the terms, P, NP, and NP-complete?
- Why do we care whether P = NP?
- What is the best guess as to whether P = NP?
- What are the consequences, if that best guess is right?

Polynomial time algorithms

- **What is a polynomial time algorithm**?
- *Traditional answer:* A polynomial time algorithm is one that can be executed by a standard (deterministic) Turing machine in polynomial time, proportional to the size of the input.

Deterministic Turing Machine

- A Turing machine operates by performing some action (executing an instruction), depending upon what state it is in and what input it receives.
- In a deterministic Turing machine, at each point in time, there is only one possible action that may be performed, given the state that it is currently in and the input it is receiving.

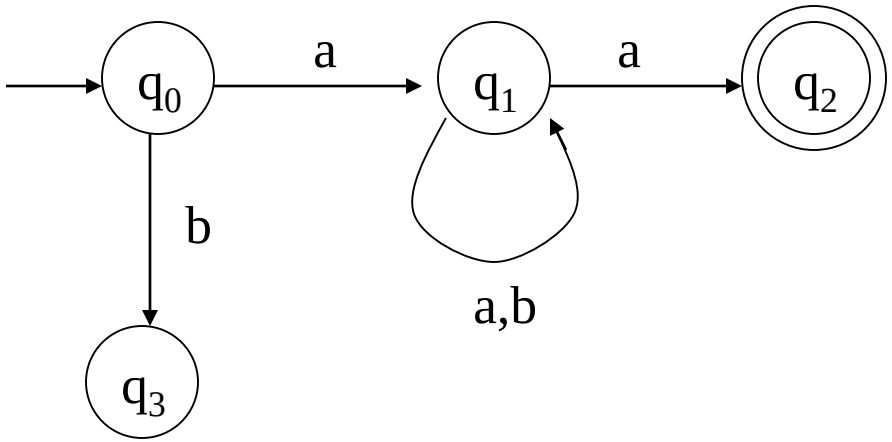
P Class of Problems

- Informally, the P class of problems is the set of all problems that can be solved by a standard deterministic Turing machine in polynomial time.

Nondeterminism

- It is possible for a finite state automaton (FSA) to be either deterministic or nondeterministic.
- In a deterministic FSA, there is exactly one arc with every possible label from one node to another, and there are no lambda (free) moves.
- In a nondeterministic FSA, there may be two or more arcs with the same label emanating from a given node. (In addition, there may be nodes with no arcs emanating from them, and/or lamda moves.)

$L = \{awa : w \in \{a,b\}^*\}$



This FSA accepts all and only the strings of the language given above.

• Note that there are two arcs out of q_1 labeled *a*. How does the FA *know* which path to take on an *a*? It doesn't; it has to magically guess right.

• Also, there are no arcs out of q_2.

• So this FA is *nondeterministic*.

Nondeterministic Turing machine

- A nondeterministic TM (NTM) has more than one possible action that it can take from the same state with the same input.
- In effect, there may be two or more branches out of the same state for a given input.
- How does the TM know which path to follow? Two answers:
  - It “splits” into several machines, each of which follows one of the possible paths, or
  - It always “guesses” the right path

Nondeterministic Turing machine

- A nondeterministic TM (NTM) has more than one transition with the same left-hand part, which means more than one transition can be taken in the same configuration.
- Nondeterminism allows a TM to have different outputs for the same input. A string is accepted if *some* computation leads to the halting state.

NP Class of Problems

- If it always guesses the right path, we can think of a nondeterministic Turing machine as having *guessed* the solution to a problem from the very beginning. Then, when the TM executes its algorithm, it is basically just verifying that the solution is correct.
- Therefore, to show that a problem is in NP, we only need to find a polynomial time algorithm to check that a given solution is valid

NP Class of Problems

- Informally, the NP class of problems is the set of all problems that can be solved by a nondeterministic Turing machine in polynomial time.
- So NP stands for “nondeterministic polynomial”.
- Note that it does **not** stand for “non-polynomial”.

NP Class of Problems

- Any problem in P is automatically in NP, since we may consider the deterministic algorithm to be a nondeterministic algorithm with an empty guessing stage.
- So we know P is a subset of NP.
- However, it is not known whether there is any problem in that is in NP but not in P.
- The question of whether P = NP is perhaps the outstanding unsolved problem of computer science.

Definition of P and NP

- P can be though of as meaning “can be solved by a standard deterministic Turing machine in polynomial time”
- NP can be though of as meaning “can be solved by a non-deterministic Turing machine in polynomial time”

NP-complete: Decision problems

What are decision problems?

In the theory of NP-Completeness, we concentrate on one particular type of problem – decision problems. We think of each instance of a given problem as mapping onto a solution space that consists of {yes, no} or {0,1}. This may seem like a severe restriction on the type of problems that we will consider, but it is not. For example, optimization problems are fairly easy to restate as decision problems. And there exists a proof that if the decision problem is easy, then the optimization problem is easy.

Decision problems

Example optimization problem -A shortest path problem:

- Given a graph G = <V,E>, two vertices, u,v$\in$V and a non-negative integer k, does a path exist in G between u and v whose length is k or less?
- Let i = (G, u, v, k) be an instance of the problem. Then the shortest path "question" with a yes/no answer becomes Path(i) = 1 or Path(i) = 0, where 1 = yes and 0 = no.

Decision problems

To change an optimization problem into a decision problem, we can usually just impose a bound on the value to be optimized.

Examples from path problems include minimization problems (Is the length of the shortest path at most k?) and maximization problems (Is the length of the longest path at least k?).

Decision problems

Usually, if the optimization problem can be solved quickly, the decision problem can also be solved quickly.

Conversely, evidence that the decision problem is hard normally is evidence that the optimization problem is also hard.

NP-Complete Problems

- NP-complete problems are a subset of NP.
- Property of an NP-complete problem: Any problem in NP can be *polynomially reduced* to it.
- This means that any problem in NP can be reduced to an NP-complete problem by an algorithm that runs in polynomial time.

Reducibility

- A problem P_1 can be reduced to P_2 as follows:
  - Provide a mapping so that any instance of P_1 can be transformed to an instance of P_2.
  - Solve P_2 and then map the answer back to the original.
  - All work with the transformation must be performed in polynomial time.

NP-Complete Problems

- A problem that is NP complete can be used as a “subroutine” for any problem in NP.
- If any NP-complete problem has a polynomial time solution, then every problem in NP must have a polynomial time solution.

NP-Complete Problems

How was the first NP-complete problem proven to be NP-complete?

- It was done in 1971 by S. A. Cook.
- The first problem that was proven to be NP-complete was the *satisfiability* problem.
- This problem takes as input a Boolean expression and asks whether the expression has an assignment to the variables that gives a true value.

NP-Complete Problems

- Satisfiability is NP since it is easy to evaluate a Boolean expression and check whether the result is true.
- Cook gave a direct proof that satisfiabiltiy is NP-complete.
- How? The proof is extremely complicated. He showed that satisfiability was NP-complete by directly proving that all problems that are in NP could be transformed to satisfiability.

Class of NP-Complete problems

What is the class of NP-Complete problems?

- In some sense, the NP-complete problems are the “hardest” problems in the class NP
- No polynomial time algorithm is known to solve any of them.
- Every NP-Complete problem is *reducible* to every other NP-complete problem in polynomial time.

Class of NP-Complete problems

What do we mean by *reducible*, in the previous sentence?

- A decision problem Q is polynomial-time reducible to decision problem Q' if a polynomial time algorithm can be developed which changes each instance of problem Q into an instance of problem Q' such that if the answer for an instance of Q' is yes, the answer to the corresponding instance of Q is yes. Then we write:

Class of NP-Complete problems

$$Q \leq_p Q'$$

We say:

Q is polynomial-time reducible to Q’

NP-Complete

Definition of NP-Complete:

A problem Q is NP-Complete if:

- Q is an element of the class NP
- $Q' \leq_p Q$ for every Q' in NP

This means that Q is not more than a polynomial factor harder than Q' for every Q' in NP.

NP-Hard

Definition of NP-Hard:

A problem Q is NP-Hard if:

$Q' \leq_p Q$ for every Q' in NP

NP-Hard vs. NP-Complete

Example:

Problem Q is NP-Hard if and only if the Satisfiability problem (or any other NP problem) is polynomial-time reducible to Q.

Problem Q is NP-Complete if and only if Q is NP-Hard and Q is known to be in the class NP.

NP-Hard vs. NP-Complete

Why can't NP-Hard = NP-Complete?

Remember that we defined NP-Complete in terms of *decision problems*; only decision problems can be proven to be NP-Complete. We can show that the knapsack decision problem can be polynomial-time reduced to the knapsack optimization problem, so the knapsack optimization problem is NP-Hard, but it is not NP-Complete.

P = NP ?

Theorem 34.4

If any NP-complete problem is polynomial-time solvable, then P = NP. If any problem in NP is not polynomial-time solvable, then all NP-complete problems are not polynomial-time solvable

Importance of Theorem

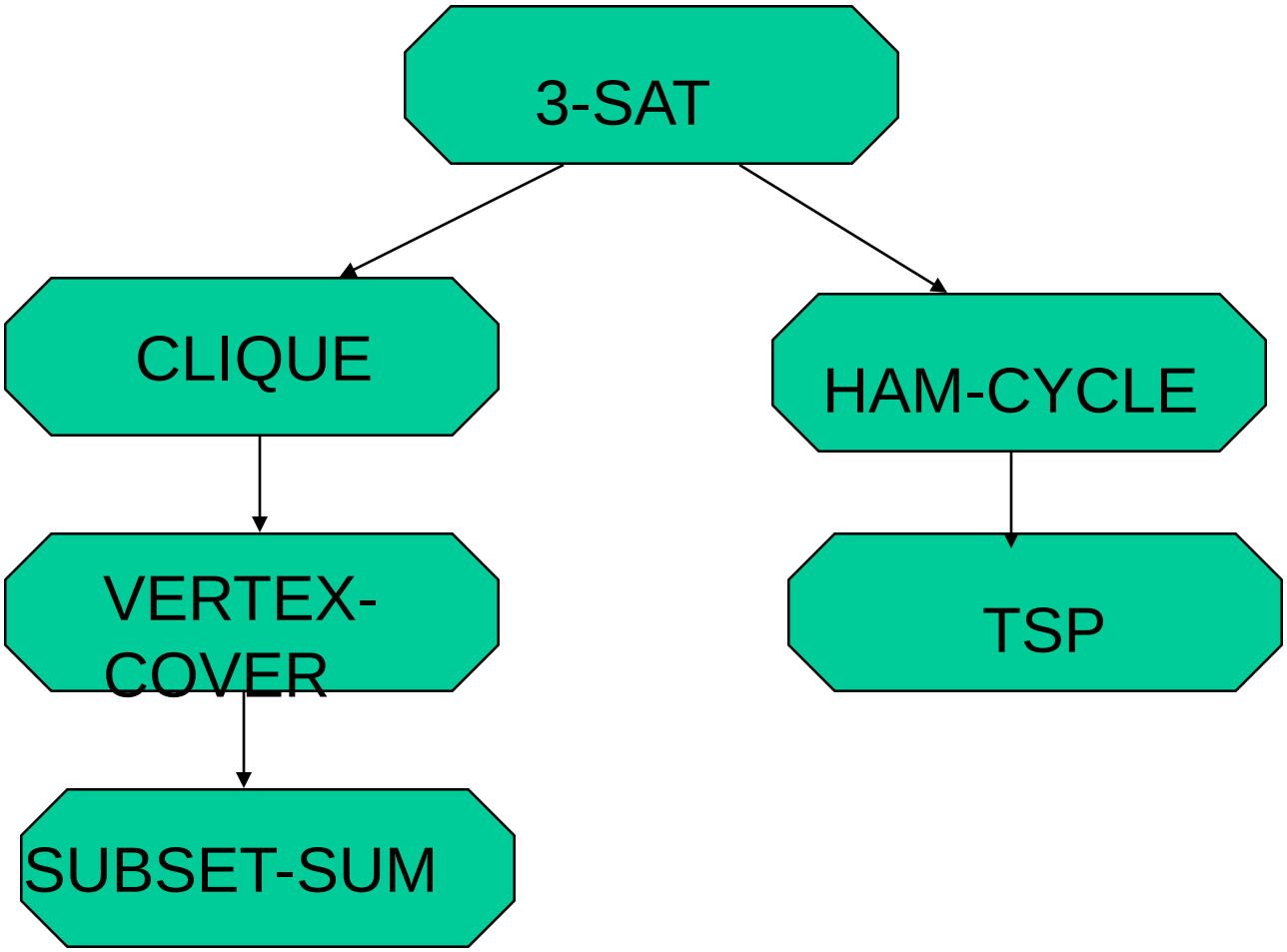
- The theorem make the focus of the P=NP question the NP-complete problems
- Most theoretical computer scientists believe that $P \neq NP$.
- However, if we can find a polynomial time algorithm for any NPC problem, then P = NP.
- No one has been able to do this so far.

NP-Completeness Proofs

- Suppose that we can show that one problem (Q) is in the class NP-complete. Then we can show that another problem Q' is in the class NP-complete by showing that
  - Q' is in NP
  - Q can be reduced to Q' in polynomial time.

3-SAT

- Cook proved that the 3-SAT problem is NP-complete. It is quite difficult to show that a particular problem in NP can be reduced to every other problem in NP. But once one problem is know to be NP-complete, the others are easier.
- Many other problems have since been proven to be NP-complete.


3-SAT
CLIQUE
HAM-CYCLE
VERTEX-COVER
TSP
SUBSET-SUM

Example

- Suppose someone has proven that the problem of determining if a graph G has a Hamilontian cycle has been shown to be NP-complete.
- And we want to show that the Traveling Salesman Problem (TSP) is NP-complete.
- Two steps
  - show that TSP is in the class NP
  - show that HAM reduces to TSP in polynomial time

The Traveling Salesman Problem

- A salesman must visit *n* cities. He must visit each city exactly one time and finish with the city at which he starts.
- We can model possible routes as a directed graph with weights on the edges.
- We can change this to a decision problem by making the problem one in which we find a route of length at least k.

Step 1

- Show that TSP is in the class NP.
  - If we are given a set of n+1 vertices that we claim constitute a tour of length at most k, then we can
    - check to see that each vertex occurs exactly one time
    - check to see that beginning and ending vertices are the same
    - check to see that edges exist between adjacent vertices in the graph
    - see if the sum of the edges is <= k

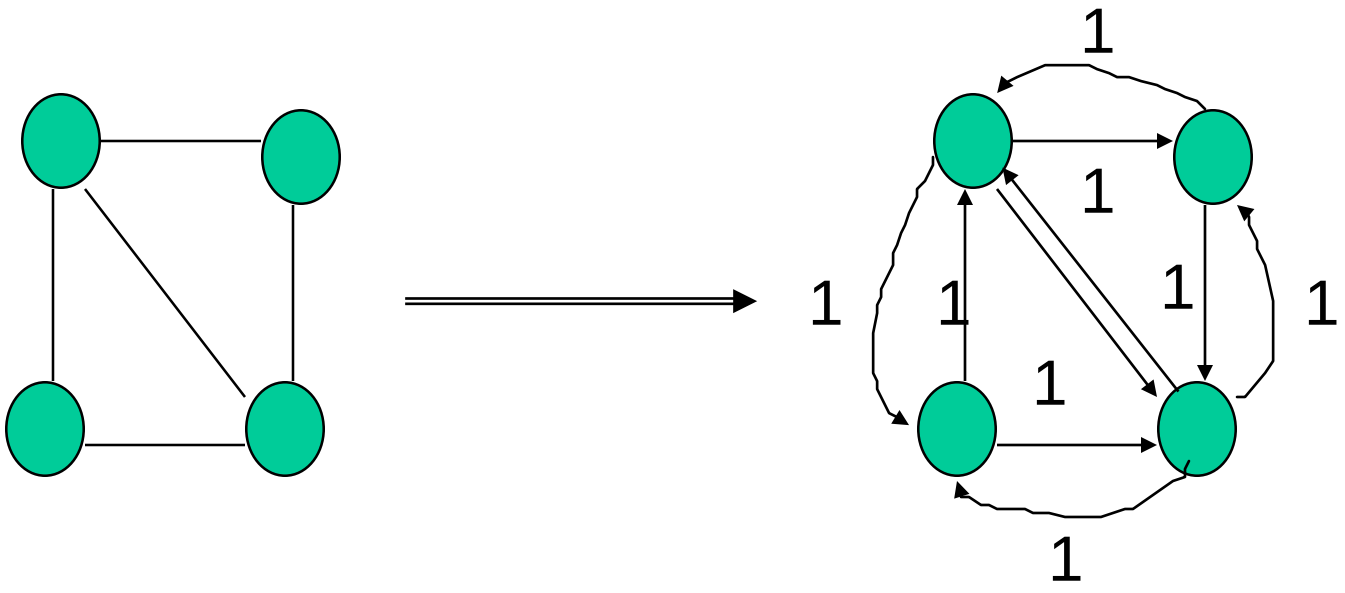
Step 1 continued

- We can show that each of these steps can be done in polynomial time
- Therefore TSP is an element of NP

Step 2

- Now we need to show that any HAM problem can be reduced to a TSP problem.
  - Let G = (V,E) be an instance of HAM
  - Construct an instance of TSP as follows:
    - Form the graph G' = (V,E') where there are two directed edges (one in each direction) in G' corresponding to every undirected edge in G.
    - Define the cost of the edges as
      c(i,j) = 1 if (i,j) is in E, infinity otherwise.
    - This instance can be easily formed in polynomial time

Step 2 Illustrated



G

G’

HAM

TSP

Step 2 continued

- Now we must show that graph G has a Hamiltonian cycle iff G' has a tour of cost at most n.
  - The fact that G has a Hamiltonian cycle implies that G' has a tour of cost at most n.
    - Suppose that graph G has a Hamiltonian cycle h.
    - Each edge in h belongs to E and thus there is an edge from v_i to v_{i+1} with a cost of 1 in G'
    - Thus, h is a tour in G' with a cost of n.

Step 2 continued

- The fact that G' has a tour of cost at most *n* implies that G has a Hamiltonian cycle.
  - Suppose that G' has a tour *h*' of cost at most *n*.
  - Since the cost of the edges is 1 or infinity, the cost of the tour is *n*.
  - Therefore all edges are in E.
  - So *h* is a Hamiltonian cycle in G.

Proof concluded

We have now proven that any HAM problem can be reduced to a TSP problem. An inspection of the proof will show that the reduction can be accomplished in polynomial time, so we have shown that HAM reduces to TSP in polynomial time.

Earlier we showed that TSP is in the class NP.

We know that HAM is NP-Complete.

Therefore TSP is also NP-Complete.

Reduce 3-SAT to Clique

- Pick an instance of 3-SAT, Φ, with k clauses
- Make a vertex for each literal
- Connect each vertex to the literals in other clauses that are not the negation
- Any k-clique in this graph corresponds to a satisfying assignment

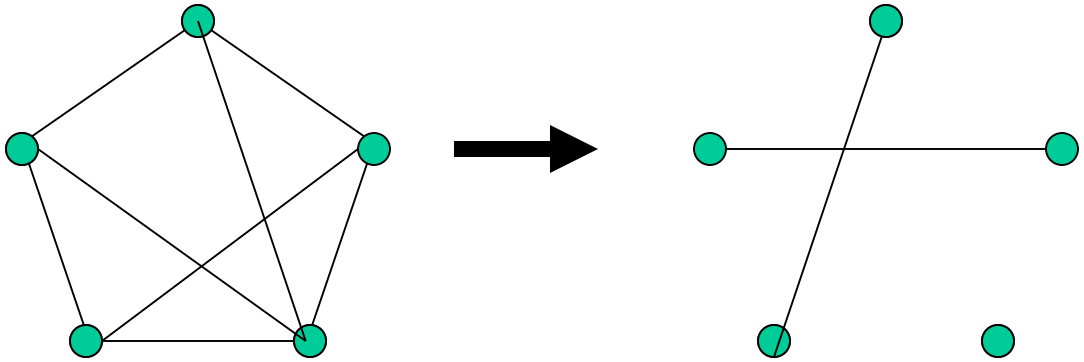
$$\varnothing=(x_1\vee\neg x_2\vee\neg x_3)\wedge(\neg x_1\vee x_2\vee x_3)\wedge(x_1\vee x_2\vee x_3)$$

Example: Independent Set

- INDEPENDENT SET = { <G,k> | where G has an independent set of size k }
- An independent set is a set of vertices that have no edges
- How can we reduce this to clique?

Independent Set to CLIQUE

- This is the *dual problem*!



General advice

- Think hard to understand the structure of both problems
- Come up with a “widget” that exploits the structure

Complexity classes

- A complexity class is a class of problems grouped together according to their time and/or space complexity
- NC: can be solved very efficiently in parallel
- P: solvable by a DTM in poly-time (can be solved efficiently by a sequential computer)
- NP: solvable by a NTM in poly-time (a solution can be checked efficiently by a sequential computer)
- PSPACE: solvable by a DTM in poly-space
- NPSPACE: solvable by a NTM in poly-space
- EXPTIME: solvable by a DTM in exponential time

Relationships between complexity classes

- $NC \subseteq P \subseteq NP \subseteq PSPACE = NPSPACE \subseteq EXPTIME$
- $P \neq EXPTIME$
- Saying a problem is in NP (P, PSPACE, etc.) gives an upper bound on its difficulty
- Saying a problem is NP-hard (P-hard, PSPACE-hard, etc.) gives a lower bound on its difficulty. It means it is at least as hard to solve as any other problem in NP.
- Saying a problem is NP-complete (P-complete, PSPACE-complete, etc.) means that we have matching upper and lower bounds on its complexity

Conclusion

Some problems are impossible to solve algorithmically.

Some are solvable algorithmically, but are intractable.

Some are solvable algorithmically, but are known to be difficult, and probably are intractable.

Some are tractable. These are the ones we can find efficient algorithms for.