The Theory of NP-Completeness

Adnan YAZICI
Dept. of Computer Engineering
Middle East Technical Univ.
Ankara - TURKEY

The Theory of NP-Completeness

- We are concerned with intractable problems whose complexity may be described by exponential functions.
- The best known algorithms for them would require many years or centuries of computer time for moderately large inputs.
- These algorithms are useless except for very small inputs.
- Here we present definitions aimed at distinguishing between the tractable (i.e., "not-so-hard") problems and intractable (i.e., "hard", or very time-consuming) ones.
- No reasonable fast algorithms for these problems have been found, but no one has been able to prove that the problems must require a lot of time.

The Theory of NP-Completeness

- Because many of these problems are optimization problems that arise frequently in applications, the lack of efficient algorithms is of real importance.
- The theory of NP-Completeness does not provide a method for obtaining polynomial time algorithms for intractable problems; nor does it say that such algorithms do not exist.
- What it does say is that many problems for which there are no known polynomial time algorithms are computationally related.
- Before we can proceed to define the properties of algorithms that are known to be in P and that are computationally related, we must introduce the concepts of decision problems and nondeterminism.

Non-deterministic Algorithms

- For deterministic algorithms, the outcome of every operation is uniquely defined.
- For nondeterministic algorithms, the outcome of each operation is not uniquely defined, but restricted to a specific set of possibilities.
- The (as yet non-existant and hypothetical) machine executing such operations is allowed to choose any one of the allowed outcomes subject to a terminating condition.
- A non-deterministic algorithm has two phases and an output step:
  1. The non-deterministic "guessing" phase. Some completely arbitrary string of characters, s, is written beginning at some designated place in memory. Each time the algorithm is run, the string written may differ. This string is the certificate; it may be thought of as a guess at a solution for the problem, so this phase may be called the guessing phase.
  2. The deterministic "verifying" phase. Eventually, this phase returns a value "true" or "false"- or it may get in an infinite loop and never halt.
  3. The output step. If the verifying phase returned "true", the algorithm's output is "success". Otherwise it is "failure."

Non-deterministic Algorithms

Non-deterministic Algorithms: We can describe a non-deterministic algorithm with an explicit procedure. Assume *genCertif* generates an arbitrary certificate.

```
void nondetA(String input)
        String s = genCertif();
        Boolean CheckOK = verifyA(input,s)
        If (checkOK)
                Ouput "success"
    Return;
```

- To make it easier to conceive of nondeterministic algorithms, it is useful to introduce the following three hypothetical procedures which we cannot actually process them on any known machine.
  Choose (S): arbitrarily chooses one of the elements of the set S
  Failure: signals an unsuccessful completion (a *no* answer)
  Success: signals a successful completion (a *yes* answer.
- *Choose* can be thought of as knowing the criterion for success, generating all of the possible choices in parallel, so is O(1).

Nondeterministic Algorithms: We can now formally define a nondeterministic algorithm as follows:

- *A non-deterministic algorithm is one that terminates unsuccessfully if and only if there exist no set of choices leading to a successful signal.*
- A machine capable of executing a nondeterministic algorithm is not known to exist in practice. However, they can provide strong intuitive reasons to conclude that fast deterministic algorithms cannot solve certain problems.
- Example: Consider the following nondeterministic algorithm for searching for a target x in an unordered array A of n elements. It is required to determine an index j such that A(j) = x or j = 0 if x is not in A.

```
j ← Choose (1:n)     // Choose elements from A
If A(j) = x then
     begin           // if x was in A, then Choose found it
     Print j; Success
     end
//  If the algorithm did not terminate on success, then //
print '0', Failure
```

- From the way a nondeterministic algorithm is defined, the number '0' can be output if and only if there is no j such that A(j) = x.
- The complexity of this algorithm is O(1).

Nondeterministic Algorithms

- Note that *Choose* function does not itself indicate success or failure. The algorithm must take the result of *Choose* function and check for success or failure.
- Another useful way to view *Choose* is as an infinitely wise guesser (or oracle) – if a correct guess can be made, *Choose* will make it.
- A deterministic interpretation of a nondeterministic algorithm can be made by allowing unbounded parallelism in computation.
- We now combine the concept of decision problems and nondeterministic algorithms. Consider the following nondeterministic algorithm for the 0-1 Knapsack Decision Problem.

```
Procedure Non-Bknap (p,w,n,m,Q,X);
    For j = 1 to n do
        X(j) = Choose (0,1)
    If ∑1≤j≤n wixi > M or ∑1≤j≤n pixi < Q  Then  Failure
    Else  Success
```

- A successful termination is possible if and only if the answer to the decision problem is "yes." The complexity of the algorithm is O(n).

Non-deterministic Algorithms

Non-deterministic Sorting:

```
Procedure Non-Sort (A,n);
// sort n positive integers in increasing order  //
  B(n) ← 0                          // initialize B to zero //
  For j = 1 to n do
      k = Choose (1:n)
      If B(k) ≠0 Then  Failure
      B(k) ← A(j)
  repeat
  For j = 1 to n-1 do               // verify order //
      If B(j) > B(j+1) Then Failure
  repeat
      print (B),   Success
```

The complexity of non-deterministic sorting is O(n).

The Class P and NP

Defn: P is the set of all decision problems solvable by a deterministic algorithm in polynomial time.

Defn: NP is the set of all decision problems solvable by a non-deterministic algorithm in polynomial time. The abbreviation NP stands for Nondeterministic Polynomial.

- Since a deterministic algorithm is just a special case of a nondeterministic one, it is clear from the definitions that $P \subseteq NP$.
- What we do not know and what is considered by many to be the most famous open problem in CS is whether $P = NP$ or $P \neq NP$.
- Is it possible that for all of the problems in NP there exist polynomial time deterministic algorithms that have remained undiscovered?
- This seems rather unlikely because of the amount of effort that has been expended to find such algorithms.
- Nonetheless, a proof that $P \neq NP$ has been just as elusive.

NP-Completeness

Definition: A decision D problem is NP-Complete if the following two properties hold:

1. $D \in NP$
2. $\forall D' \in NP, D' \propto D$

Property (2) requires that any problem $D' \in NP$ can be polynomially transformed to the specific problem D.

- Thus, NP-Complete problems can be viewed as "the hardest" problems in NP. If any single NP-Complete problem can be solved in polynomial time, then so can all problems in NP.

Cook's Theorem: While considering the P = NP question, S.A. Cook (Proc. of the Third ACM Symposium on Theory of Computing, 1971, pp. 151-158) formulated the following question:

- *Is there a single problem in NP such that if we showed it to be in P, then that would imply that P=NP?*
- Cook answered his own question with his famous theorem:
  The Satisfiability problem (SAT) is in P if and only if P = NP.

Satisfiability Problem (SAT)

- SAT is the following decision problem: Let x_1, x_2,…, x_n be Boolean (true and false) variables.
  - Let $\sim x_i$ denote the negation of x_i.
  - A literal is then a variable or its negation.
  - A clause is a disjunction,"or", or conjunction,"and", of the literals.
  - A formula is a series of disjunctions and/or conjunctions of clauses.
- If only disjunctions are used, the formulas are said to be in disjunctive normal form (DNF) and if only conjunctions are used the formula is in CNF.
- In mathematical terms, if c_i are clauses and l_{ij} are literals, then CNF can be represented as:

  $$c_i = \vee\, l_{ij}$$

  and the formula is

  $$\text{CNF} = \wedge\, c_i \qquad \text{(product of elementary sums)}$$
- For DNF, the $\vee$ and $\wedge$ are interchanged.

NP-Completeness

Question: Is there a satisfying truth assignment? In other words, for CNF, is there a choice of variables such that at least one literal is true in each clause?

- An alternative statement of his theorem is:
  SAT is NP-Complete.
  - That is, in the process of proving his theorem, Cook showed that all problems in NP could be polynomially transformed to SAT.

Example: Variable x_1 x_2 x_3 and the formula

$$(x_1 \vee x_2) \wedge (\sim x_1 \vee \sim x_2) \wedge (x_1 \vee x_3) \wedge (\sim x_1 \vee \sim x_3)$$

- For this instance, if x_1 = true, x_2 = false, and x_3 = false, then the formula (or proposition) is true and the answer is "yes"
- Cook's theorem has truly remarkable consequences in that it delineates one simple problem as being archetypal of all problems in NP. SAT is the common denominator!

NP-Completeness

Outline of Proof of Cook Theorem: We will not show the proof of Cook's theorem because it is a number of typeset pages long. The general idea used by Cook in the proof is as follows:

- First, Cook gives a full mathematical definition of a Turing machine, which can read inputs, perform certain operations, and write outputs. When endowed with the power of nondeterminism, a Turing machine can solve any problem in NP.
- The next step in the proof is to describe each feature of the machine in terms of logical formulas such as appear in SAT, including the way that instructions are executed. In this way, a correspondence is established between every problem in NP (which can be expressed as a program on the nondeterministic Turing machine) and some instance of SAT (the translation of that program into a logical formula).
- Now, the solution to SAT corresponds to a simulation of the Turing machine running the program on the given input, so it produces a solution to an instance of the given problem.
- The textbook by Garey and Johnson is a good source of the proof of Cook's theorem.

Proving NP-Completeness

- If every proof of NP-Completeness was as complicated as Cook's proof, not many problems would have been shown by now to be NP-Complete.
- The process for proving a decision problem D to be NP-Complete:
    1. Show that D is in NP
    2. Select a known NP-Complete problem D_k
    3. Construct a transformation $D_k \propto D$
    4. Prove that the transformation is O(p)
- Starting with SAT as the common denominator, researchers have proven many decision problems to be NP-Complete by this process.
- The most famous collection of these results are by Richard Karp ("Reducibility among combinatorial problems," in Complexity of Computer Computations, 1972), where he proved 21 problems to be NP-Complete.
- Many researchers begin their search for a suitable NP-Complete problem to start from these basic NP-Complete Problems.

Proving NP-Completeness

3-SAT: Same as SAT except that the number of variables is limited to 3.

3-Dimensional Matchning (3DM): Three sets U,V, W, each of cardinality m and a collection M of sets (called triples), each of which contains three elements, one each from U, V, and W.

Question: Do there exist m triples in M, say M_1, M_2, …, M_n, such that

$$M = \cup_{1 \le j \le m} M_j = U \cup V \cup W$$

In other words, does M contain a matching?

Vertex Cover (VC): A graph G having n vertices and an integer k; 0<k<n.

Question: Is there a subset S of k vertices in G such that every edge of G has at least one endpoint in S? In other words, is there a set of less than n vertices that touches all the edges.

Clique: A graph G with n vertices and an integer k; 0<k<=n.

Question: Does G contain a clique of size k or more? A clique is the maximal subgraph of a graph that is complete (all vertices are connected to all other vertices).

Hamiltonian Circuit (HC): A graph G

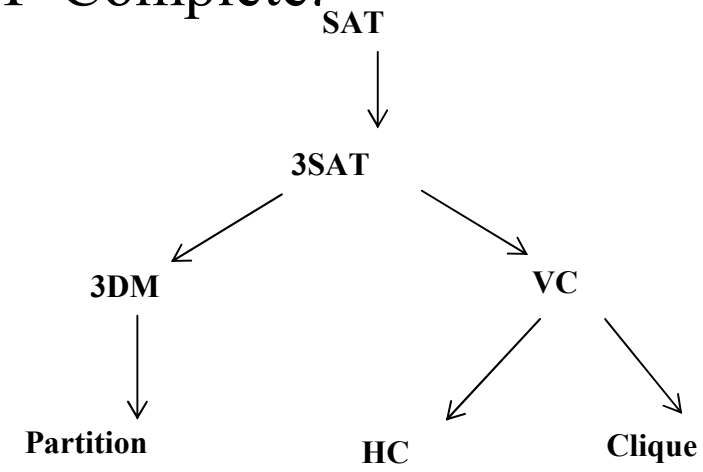
Question: Does G contain a Hamiltonian circuit?

Partition: A set of integers A.

Question: Can A be divided into two subsets whose sums are equal?

Proving NP-Completeness

- The following diagram shows how the six basic problems have been proved NP-Complete:



• We now show the first of these proofs, namely that 3SAT is NP-Complete.

• The Garey and Johnson text is the most comprehensive source of NP-Complete proofs as well as techniques for proving NP-Completeness.

• Johnson has an on-going discussion of this topic in the Journal of Algorithms.

Examples for the Proofs for NP-Completeness

Example-1: 3-SAT is the following decision problem.

Instance: A collection C of disjunctive clauses, where each clause contains exactly three literals.

Question: Is there a satisfying truth assignment for C?

- 3-SAT is a special case of CNF-SAT. It is important to note that special cases of an NP-Complete problem are not necessarily NP-Complete.

- For example, 2-SAT (where each clause has exactly 2 literals) can be solved in polynomial time by a deterministic algorithm and therefore, is in P.

Examples for the Proofs for NP-Completeness

Theorem: *3-SAT is NP-Complete.*

- We will use CNF-SAT as the known NP-Complete problem to be reduced to the 3-SAT problem.
- First we need to show that the problem is non-deterministic.
- The following procedure is a polynomial time non-deterministic algorithm that terminates successfully iff a given prepositional formula $E(x_1,\ldots,x_n)$.

```
Procedure EVAL-3SAT (E,n)
// non-deterministic 3-satisfiablity //
// Determine if the propositional formula E is satisfiable. The
     variables are x_j, 1≤ j ≤ n //
   Boolean x(n)
   For j ← 1 to n do              // choose a truth value assignment //
     x_j ← Choice (true,false)
    repeat
    If E(x_1,…,x_n) is true then Success    // satisfiable //
     Else Failure
    Endif
End EVAL-3SAT
```

Examples for the Proofs for NP-Completeness

Theorem: *3-SAT is NP-Complete.*

2. Select a known NP-Complete problem D_k
- We choose SAT as the known NP-Complete problem.

3. Construct a transformation $D_k \propto D$
- As the third step of the proof, every instance of SAT problem (a known NP-Complete problem) must be transformed into that of 3-SAT problem (to be an NP-Complete problem), where “yes” and “no” answers are preserved.
- It turns out that CNF-SAT can be reduced to 3-SAT.
- Given an instance I of SAT, we can construct an instance f(I) of 3-SAT as follows:

Examples for the Proofs for NP-Completeness

Theorem: *3-SAT is NP-Complete.*

3. Construct a transformation D' $\propto$ D

- Replace a clause {z} by four clauses {z,a,b}, {z,~a,b}, {z,a,~b}, {z,~a,~b}, where a and b are new Boolean variables.

- Replace a clause $\{z_1,z_2\}$ by two clauses $\{z_1,z_2,c\},\{z_1,z_2,\sim c\}$, where c is a new Boolean variable.

- Leave any clause $\{z_1,z_2,z_3\}$ unchanged.

- For any clause $\{z_1,z_2,\ldots,z_k\}$, where $k > 3$, replace it by k-2 clauses $\{z_1,z_2,c_1\}$, $\{\sim c_1,z_3,c_2\}$, $\{\sim c_2,z_4,c_3\}$, $\{\sim c_3,z_5,c_4\}$, …, $\{\sim c_{k-4},z_{k-2},c_{k-3}\}$, $\{\sim c_{k-3},z_{k-1},z_k\}$, where c_i are new Boolean variables.

Examples for the Proofs for NP-Completeness

- We must now show that "yes" and "no" instances are preserved. Suppose that I is a "yes" instance of CNF-SAT.
- We now show that f(I) is a "yes" instance of 3-SAT.
- {z} is satisfied only if z is true. If z is true, all four clauses constructed in 3-SAT are also true since they all contain z, since the clauses are disjunctive.
- $\{z_1,z_2\}$ is satisfied if either z_1 or z_2 is true. Since the two clauses constructed in 3-SAT contain both z_1 and z_2, they must both be satisfied.
- This (3^{rd}) case is trivial
- Suppose z_i is true literal in $\{z_1,z_2,\ldots,z_k\}$, k>3. Then define:
  c_j = true, if $j \leq i-2$
  c_m = false, if $m > i-2$
- The first i-2 clauses are satisfied since the c_j are true.
- The $(i-1)^{st}$ clause is satisfied because z_j is true.
- The remaining clauses are satisfied since $\sim c_m$ is true.
- Hence f(I) has a satisfying truth assignment if I of SAT does.

Examples for the Proofs for NP-Completeness

- In order to prove that "no" instances are preserved, we have two choices;
  (1) we can show that if an individual clause is false then the transformed one is false, or
  (2) we can show that "yes" instances are preserved in the reverse direction (3-SAT → SAT). [Consider contra-positive rule, p → q ⇔ ~q → ~p.]
- This is a sufficient proof since the sequence D_k(no) → D(yes) → D_k(yes), that is, SAT(no) → 3-SAT(yes) → SAT(yes) is ruled out by contradiction.
- In other words, if we show that D(yes) → D_k(yes), that is, 3-SAT(yes) → SAT(yes), then "no" instances are also preserved.
- The latter method is the one usually followed.

Examples for the Proofs for NP-Completeness

- The proof that "yes" instances of 3-SAT transform to "yes" instances of CNF-SAT is as follows:
- If the four clauses arising from {z} in 3-SAT are satisfied, then z must be true. To see this, pick any arbitrary values for a and b; the only way all four clauses can be true if {z} is true.
- If the two clauses $\{z_1,z_2,c\}$ and $\{z_1,z_2,\sim c\}$ are true, then at least one of z_1 or z_2 must be true and thus $\{z_1,z_2\}$ is satisfied.
- This is trivial again, $\{z_1,z_2,z_3\}\leftrightarrow\{z_1,z_2,z_3\}$
- Suppose that the k-2 clauses arising from a clause $\{z_1,z_2,\ldots,z_k\}$, k >3, are satisfied. We must prove that at least one z_i is true.
- Assume that all z_i are false. Since $\{z_1,z_2,c_1\}$ is satisfied, c_1 must be true. Then, since $\{\sim c_1,z_3,c_2\}$ is satisfied, c_2 must be true.
- By continuing this way, we find that all c_i must be true. However, then the last clause $\{\sim c_{k-3},z_{k-1},z_k\}$ is not satisfied. This contradiction proves that at least one z_i must be true and hence the clause $\{z_1,z_2,\ldots,z_k\}$, k >3, is satisfied.
- In summary, the way in which the instance of CNF-SAT was transformed into 3-SAT preserves "yes" and "no" instances.

Examples for the Proofs for NP-Completeness

4. Prove that the transformation is O(p)

- This transformation is a polynomial time algorithm, which is $\Theta(n)$ and is therefore a polynomial transformation.

This completes the proof.

Examples for the Proofs for NP-Completeness

Example-2: *A clique* in an undirected graph G = (V,E) is a subset V' $\subseteq$ V of vertices, each pair of which is connected by an edge in E.

- That is, clique = {<G,k> : G is a graph with a clique of size k}.
- In other words, a clique is a complete subgraph of G.
- The size of a clique = the number of vertices that it contains.
- The Clique problem is the optimization problem of finding a clique of maximum size in a graph. That is, a clique is the maximal subgraph of a graph that is complete (all vertices are connected to all of the other vertices).
- As a decision problem, is there a clique of a given size k exists in the graph?
- Exhaustive algorithm is $\Omega(k^2\ C(\ |V|,\ k))$, where C is used for combination, and means that list all k-subsets of |V| and check each one to see whether it forms a clique.

Examples for the Proofs for NP-Completeness

Theorem: *The Clique problem is NP-Complete.*

- We will use 3-CNF-SAT as the known NP-Complete problem to be reduced to the Clique decision problem (CLIQUE).

1. Show that CLIQUE $\in$ NP.

```
Procedure CLIQUE (G,n,k)
    S ← ∅                          // S is an initially empty set //
                                   // the following is O(k)  //
    for j ← 1 to k do                      // select k distinct vertices //
            t ← Choose(1:n)
            If t ∈ S then Failure          // not distinct k //
               S ← S ∪ t                   // add t to set S //
    repeat                         // at this point S contains k distinct
                                                vertex indices //
                                   // The following is O(k²)  //
    for all pairs (j,m) such that j ∈S, m ∈S and j ≠ m do
            if (j,m) is not an edge of the graph then Failure
    repeat
    Success
End // CLIQUE
```

Overall, this algorithm has time complexity $O(\max\{n,k^2\}) = O(k^2)$.

Examples for the Proofs for NP-Completeness

Theorem: *The Clique problem is NP-Complete.*

2. 3-CNF-SAT $\propto$ CLIQUE ?

- Let $F = C_1 \wedge C_2 \wedge \ldots \wedge C_k$ be a Boolean formula in 3-CNF with k clauses, $l_1,\ldots,l_n$ be literals in F, r = 1, 2, .., k, and each clause C_r has exactly three distinct literals l^r_1, l^r_2, l^r_3.
- We shall construct a graph G = (V,E) such that F is satisfiable iff G has a clique of size k.
- For any F, the graph G = (V,E) is constructed as follows:
- For each clause $C_r = (l^r_1, l^r_2, l^r_3)$ in F, we place a triple of vertices v^r_1, v^r_2, and v^r_3 in V. We put an edge between two vertices v^r_i, and v^s_m if the following rules hold:

1. v^r_i and v^s_m are in different triples, that is, $r \neq s$, and
2. The corresponding literals are *consistent*, so, l^r_i is not negation of l^s_m.

- This graph can easily be computed from F in polynomial time, O(k), when the length of F is k.

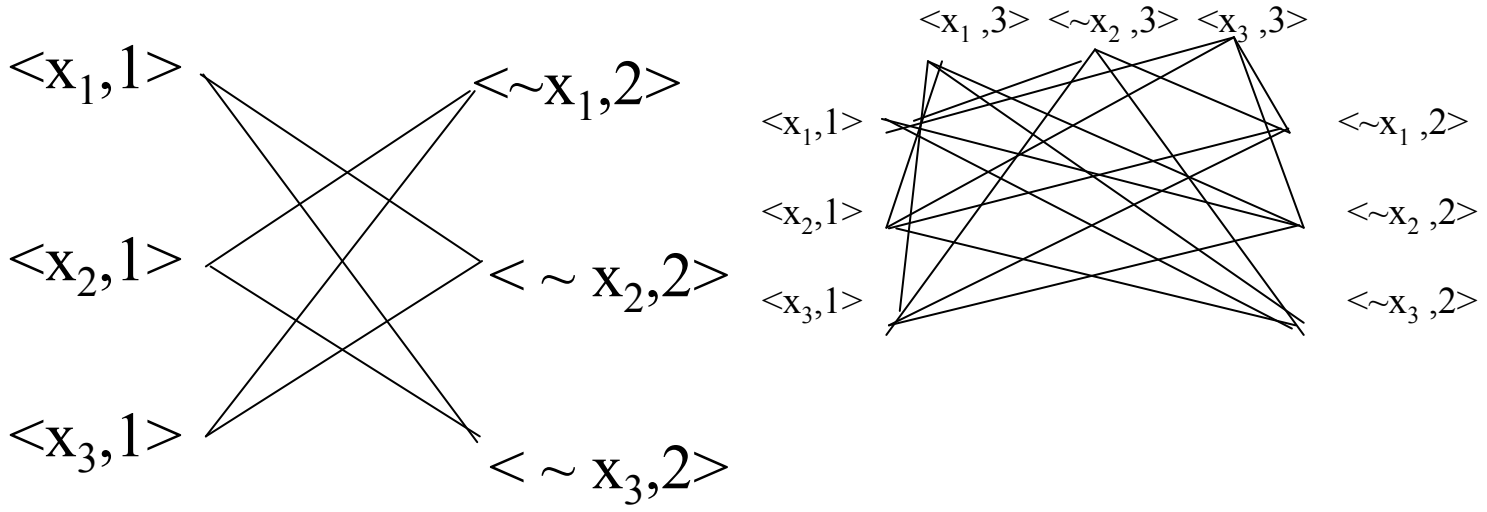
Examples for the Proofs for NP-Completeness

Theorem: *The Clique problem is NP-Complete.*

Example: Construction examples of G are shown below:

$F_1 = (x_1 \vee x_2 \vee x_3) \wedge (\sim x_1 \vee \sim x_2 \vee \sim x_3)$

$F_2 = (x_1 \vee x_2 \vee x_3) \wedge (\sim x_1 \vee \sim x_2 \vee \sim x_3) \wedge (x_1 \vee \sim x_2 \vee x_3).$



This graph contains six cliques of size two.

Examples for the Proofs for NP-Completeness

- We must show that this transformation of F into G is a reduction. So, if F is 3-CNF-SAT, G has a clique of size at least k.
- Suppose that F has a satisfying assignment. Then each clause C_r contains at least one literal l^r_i that is assigned 1 (true) and each such literal corresponds to a vertex v^r_i.
- Picking one such "true" literal from each clause yields a set of V' of k vertices.
- We claim that V' is a clique.
- For any two vertices, $v^r_i, v^s_m \in V'$, where $r \neq s$, both corresponding literals l^r_i and l^s_m are mapped to 1 by the given satisfying assignment, thus the literals cannot be complements.
- Thus, by the construction of G, the edge (v^r_i, v^s_m) belongs to E.

Examples for the Proofs for NP-Completeness

2. Conversely, we now show that if G has a clique V' of size at least k, then F is 3-CNF-Satisfiable.

- There cannot be an edge in G connect vertices in the same triple, and so V' contains exactly one vertex per triple.
- If G has a clique (V',E) of size at least k, the number of vertices in V' must be exactly k.
- We can assign 1 to each literal l^r_i such that $v^r_i \in V'$ without fear of assigning 1 to both a literal and its complement, since G contains no edges between inconsistent literals. Each clause is satisfied, and so F is satisfied.
- In the example, a satisfying assignment of F is $<x_1 = 1, x_3 = 0>$, and a corresponding clique of size k = 2.
- Therefore, S is 3-CNF- Satisfiable. Hence, S is satisfiable iff G has a clique of size at least k.

NP-Hard problems

- Garey and Johnson extended the theory of NP-Completeness to a theory of NP-Hardness.
- They give a rigorous definition of NP-Hardness in terms of reducibility by a special non-deterministic Turing Machine, called an Oracle Turing Machine.

  Defn: A problem L is NP-Hard iff $\forall$ problems $\in$ NP $\propto$ L.

  Defn: A problem L' is NP-Complete iff L' is NP-hard and L' $\in$ NP.
- It is easy to see that there are NP-Hard problems that are not NP-Complete.
- Although only decision problems can be NP-Complete (according to the definition), we usually relax the definition and refer to the corresponding optimization problem as being NP-Complete.
- The NP-Hard refers to a problem that all problems in NP can be reduced to, but which itself is not in NP (remember that only decision problems can be in NP).
- NP-Hard problems are at least as hard as NP-Complete problems.
- An optimization problem may be NP-Hard. There also exist NP-Hard decision problems that are not NP-Complete.
- For example, *Halting* problem is an NP-Hard decision problem, but not NP-Complete.

NP-Hard problems

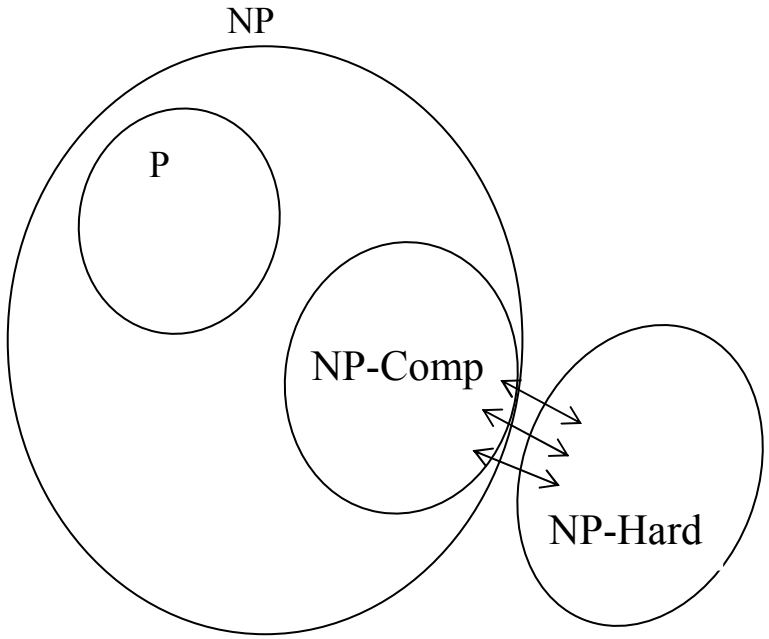
Halting problem: The halting problem is to determine for an arbitrary deterministic algorithm A and input X whether an algorithm A with input X ever terminates (or enters an infinite loop).

- It is known that this problem is undecidable. Hence there exist no algorithm (of any complexity) to solve it. So, it is clearly cannot be in NP.

Idea of the proof: To show *SAT* $\propto$ *Halting* problem, simply construct an algorithm A that tries out all 2^n possible truth assignments and verifies if X is satisfiable.

- If it is, then A stops. If X is not satisfiable, then A enters an infinite loop. Hence A halts on input X iff X is satisfiable.
- If we had a polynomial time algorithm for the halting problem, then we could solve the Satisfiability problem in polynomial time algorithm using A and X as input to the algorithm for the Halting problem.
- Hence, the Halting problem is an NP-Hard problem that is not in NP.

NP-Complete and NP-Hard problems



Relationships between *P, NP, NP-Complete,* and *NP-Hard.*

Methods for Proving NP-Completeness

There are no truly standard approaches to prove a problem to be NP-Complete.

1. **Restriction**: This is the easiest NP-Completeness proof method, when it is applicable. This method consists of showing that a problem to be proved NP-Complete contains a special case of a known NP-Complete problem. For example, the Directed Hamiltonian Cycle problem is proved to be the special case of Undirected Hamiltonian Cycle, where each directed arc is complemented by its opposite direction counterpart, which is identical to the Undirected Hamiltonian Cycle problem.
2. **Local Replacement**: In this approach, we pick some aspect of the known NP-Complete problem instance to make up a collection of basic units, and then we obtain the corresponding instance of the target problem by replacing each basic unit, in a uniform way, with a different structure. The transformation of SAT to 3-SAT follows this approach. The basic units of an instance of SAT were the clauses, and each clause was replaced by a collection of clauses of 3-SAT clauses according to the same general rule.
3. **Component design:** This is the most difficult of the three popular proof methods. The basic idea is to use the constituents (or components, parts) of the target problem instance to design certain "components" that can be combined to "realize" instances of the known NP-Complete problem.

Using NP-Completeness to Analyze Problems

- When confronted with a new problem, if there is no obvious polynomial time algorithm to solve it, the theory of NP-Completeness allows us to use a two-pronged approach.
- On one hand, we should try to formulate a polynomial time algorithm.
- Concurrently, we should try to construct a proof of NP-Completeness.
- Unfortunately, the ultimate result might be elusive; the problem might be in that group where, if P ≠ NP (as most of us believe), it is neither NP-Complete nor is it polynomially solvable.
- This is not an effort to be taken on without luck, skill, and perseverance.

Methods for Dealing with NP-Complete Problems

- If a problem is known to be NP-Complete, do we just give up since we know it is improbable that a polynomial algorithm can be found? Of course no! First, it is important to understand that solving particular instances of an NP-Complete problem need not be difficult.
- NP-Completeness says that we do not expect to find a polynomial time algorithm for all instances. Often, special instances of an NP-Complete problem can be solved readily. Recall, for example, the task scheduling problem, which is NP-Complete. The special instances where all tasks lengths are 1 (one) can be solved by a greedy algorithm in $\Theta(n^2)$ time.
- Another approach to solving NP-Complete problems is to try to find a polynomial time approximation algorithm. For example, recall the multiprocessor-scheduling problem. A greedy algorithm is guaranteed to produce a solution that is at most 1/3 longer than optimal.
- Methods that seek a "good" but not necessarily optimal solution in an acceptable amount of time are referred to as "heuristic" algorithms.
- Other approaches to approximation algorithms use the following definition:
- Let ε be a positive real number. An algorithm A for a combinatorial optimization problem is said to be an *ε-approximation* algorithm if A always finds a feasible solution having relative error at most ε.

Example for approximation algorithms

Example: A special case of TSP is the Euclidean TSP, where the triangle inequality is satisfied. That is, for all vertices u, v, and w, the following inequality is satisfied:

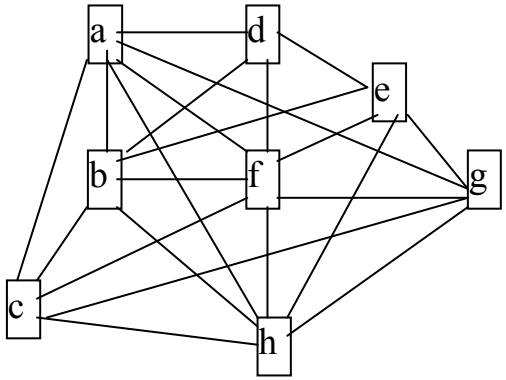
$$c(u,v) \leq c(u.w) + c(w,v)$$

- This problem can be shown to be NP-Complete. An approximation algorithm based on minimum spanning trees is as follows:

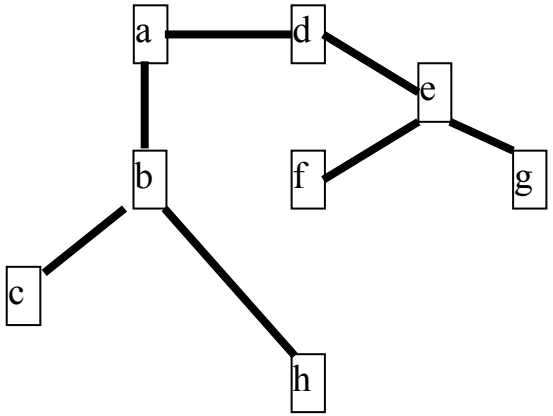
APPROX-TSP-TOUR (G,c)

- Select a vertex r $\in$ V[G] to be a "root" vertex
- Grow a MST T for G from root r using MST-PRIM (G,c,r)
- Let L be the list of vertices visited in a preorder tree walk of T
- **Return** Hamiltonian cycle H visiting the vertices in the order L

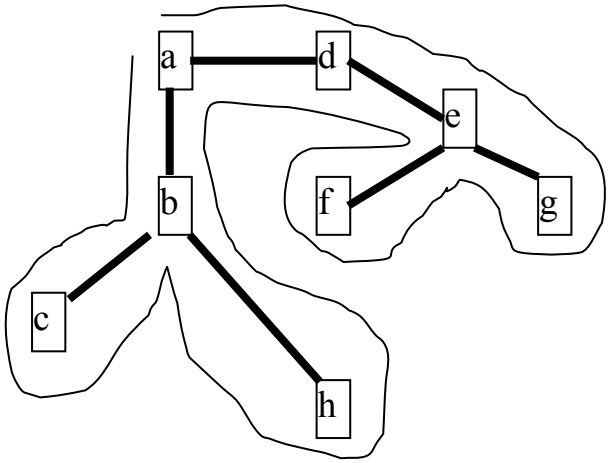
Example for approximation algorithms



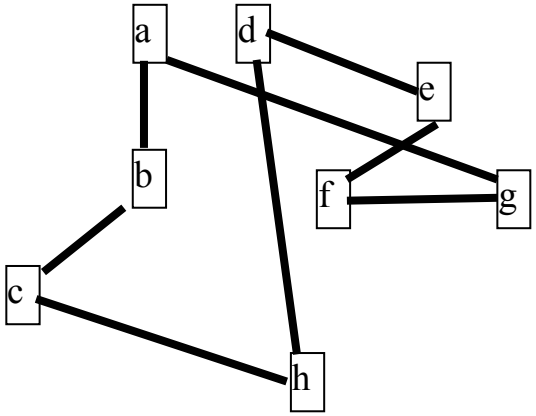
Given graph G



Result of MST-Prim



A preorder tree walk of MST T



The tour H returned by Approx-TSP-Tour algorithm

Example for approximation algorithms

- This approximation algorithm based on using minimum spanning trees can be shown to have a maximum relative error of 1 (ɔ is 1.0).
- More particularly, if H is the approximation of Hamiltonian cycle and H_{opt} is the optimum one, then it can be shown that $C(H) \leq 2.c(Hopt)$

Example for approximation algorithms

Example: For the 0/1 Knapsack problem, an even more interesting approximation algorithm has been developed by defining an approximation problem as follows:

- The original problem with n objects, weights w_i, profits p_i, and capacity M is approximated as a problem with n objects, weights w_i, profits $\lfloor p_i/2^k \rfloor$ and capacity M, where k is a positive integer. Thus, all we are doing is truncating the last k bits from each p_i.
- By using a modification of the dynamic programming algorithm for BKP, it can be shown that the complexity of the computations is $\Theta(n^3/э)$, where $э \geq n.2^k/pmax$, where pmax is the largest of the p_i.
- Note that this does not imply a polynomial time algorithm for BKP.