CS 301 – Algorithms

NP–Completeness

Hüsnü Yenigün

So far, it was simple

- The algorithms we have seen so far were all $O(n^k)$ for some $k \geq 0$
- So, they were all easy problems in some sense
- Not easy because the algorithms to solve these problems were not complex
- Easy because they can be solved in polynomial time
- Ever wondered if all the problems are $O(n^k)$, i.e. if all the problems can be solved in at most polynomial time
- What is your guess?

Life is not easy at all

- Life would be so easy (or may be difficult) if all the problems were solvable in polynomial time
- So, the answer is NO!!!

Decision problems

A problem is a *decision problem* is it asks for a “yes” or “no” answer

- Given a number m and an n node RB-tree T, does m occur in T?
- Given two vectors $\overrightarrow{p_1}$ and $\overrightarrow{p_2}$, is $\overrightarrow{p_1}$ ccw from $\overrightarrow{p_2}$?

Converting into decision problems

Some problems are not directly asked in this way, however they can be converted into their decision problem form.

Problem	Corresponding Decision Problem
Given two vectors $\overrightarrow{p_1}$ and $\overrightarrow{p_2}$, which one is ccw from which one?	Given two vectors $\overrightarrow{p_1}$ and $\overrightarrow{p_2}$, is $\overrightarrow{p_1}$ ccw from $\overrightarrow{p_2}$?
Given $G = (V, E)$ and $s, t \in V$, what is the shortest path from s to t in G?	Given $G = (V, E)$ and $s, t \in V$ and an integer k, $\exists$ a path from s to t in G whose length is less than k?

On one extreme : *tractable problems*

- On one extreme we have the easy problems that can solved in $O(n^k)$ time, i.e. the problems that can be solved in polynomial time.
  - Given two vectors $\overrightarrow{p_1}$ and $\overrightarrow{p_2}$ find if $\overrightarrow{p_1}$ is ccw from $\overrightarrow{p_2}$ or not: $\rightarrow O(1)$
  - Check if a number m exists in an n node RB-tree: $\rightarrow O(\lg n)$
  - Find the minimum number in a given set of n numbers: $\rightarrow O(n)$
  - Sort a given list of n numbers $\rightarrow O(n \lg n)$
  - ...
  - Given a number n, is it a prime number? $\rightarrow O(n^{12})$
  - **The class P**: the set of all problems that can be solved in polynomial time

n the other extreme : *undecidable problem*

- On the other extreme, there are problems that can not be solved no matter how much time is used.
  - Halting problem
  - Post Correspondence problem
- These problems are proven to be unsolvable
- In other words, we cannot find an algorithm that solves these problems

Post Correspondence Problem

- Given an alphabet Σ, and two sequences of strings $(v_1, v_2, \ldots, v_n)$ and $(w_1, w_2, \ldots, w_n)$ such that $v_i, w_i \in \Sigma^\star$, does there exist a sequence of positive integers $(i_1, i_2, \ldots, i_k)$ where $k > 0$ such that

$$v_{i_1} v_{i_2} \ldots v_{i_k} = w_{i_1} w_{i_2} \ldots w_{i_k}$$

- An instance of PCP ($\Sigma = \{0, 1\}$):
  - $(v_1, v_2, v_3, v_4) = (1, 0, 010, 11)$
  - $(w_1, w_2, w_3, w_4) = (10, 10, 01, 1)$
- A solution for this instance:
  $(i_1, i_2, i_3, i_4, i_5, i_6) = (1, 2, 1, 3, 3, 4)$
- $\overbrace{1 \quad 0 \quad 1 \quad 010 \quad 010 \quad 11}^{v_1 \; v_2 \; v_1 \; v_3 \; v_3 \; v_4} = \overbrace{10 \quad 10 \quad 10 \quad 01 \quad 01 \quad 1}^{w_1 \; w_2 \; w_1 \; w_3 \; w_3 \; w_4}$

Post Correspondence Problem

- Another instance:
  - $\Sigma = \{0, 1\}$
  - $(v_1, v_2, v_3) = (10, 011, 101)$
  - $(w_1, w_2, w_3) = (101, 11, 011)$
- This instance has no solution.
- We must have $i_1 = 1$ since only v_1 and w_1 starts with the same symbol.
- Then, we must have $i_2 = 3$
- After this point $i_3, i_4, \ldots$ all have to be 3, but we never get the same sequence

Halting Problem

Consider the following programs:

```
program HelloWorld ()
   printf "HelloWorld";
end HelloWorld;
```

```
program Loopy (n :  int)
   while (true)
      print n;
end Loopy;
```

```
program Factorial (n :  int)
   i,j :  int;
   i = 1;
   for (j = n; j > 1; j=j-1)
      i = i * j;
   print j;
end Factorial;
```

```
program CondLoopy (n :  int)
   while (n>0)
      print n;
end CondLoopy;
```

- `HelloWorld` halts (terminates) in a constant amount of time
- `Factorial` halts eventually (but always) depending on the input
- `Loopy` never halts
- `CondLoopy` may or may not halt depending on the input

Halting Problem

- Find an algorithm H whose inputs are
  - a program P
  - some inputs $i_1, i_2, \ldots, i_n$ to P
- such that
  - $H(P, i_1, i_2, \ldots, i_n)$="yes" : $P(i_1, i_2, \ldots, i_n)$ halts
  - $H(P, i_1, i_2, \ldots, i_n)$="no" : $P(i_1, i_2, \ldots, i_n)$ does not halt
- In other words, "Given a program P and some input values for P, determine if P halts when these inputs are applied."
- No such algorithm can be found

Undecidability of the Halting Problem

- Alan Turing, 1936
- Assume that we have found and implemented the algorithm H
- Consider the program

```
N ( W ) {
  if (H(W,W) == yes) // W halts when its source
     while (true);   // is applied to itself

  else          // W does not halt when its
     halt;      // source is applied to itself
}
```

- Consider now the question: What is the outcome of $H(N, N)$?
  - Assume $H(N, N)$ is "yes": This means $N(N)$ halts. But for $N(N)$ to halt, the condition of the if statement in the body of N must be false, i.e. $H(N, N)$ cannot be "yes".
  - Assume $H(N, N)$ is "no": This means $N(N)$ does not halt. But for $N(N)$ not to halt, it must go into the infi nite loop in its body, which is only possible if $H(N, N)$ is "yes".
  - Therefore, no such algorithm can exist

Undecidability Proofs

- Although the proof of undecidability of the Halting Problem was surprisingly simple, it is not the case for most of the undecidable problems
- Reduction and restriction are easy ways to prove the undecidability of problems
- Suppose Q and Q' are two problems, and Q' is known to be undecidable.
- If Q' can be converted into Q, then Q is also undecidable
- If Q' is a special case of Q, then Q is also undecidable

Between the extremes

- On one extreme there are polynomial time solvable problems
- On the other extreme there are undecidable problems
- Are there any problems in between?

Between the extremes

- On one extreme there are polynomial time solvable problems
- On the other extreme there are undecidable problems
- Are there any problems in between?
- Yes: e.g. exponential time problems.
  - Given a boolean formula of the form

    $$p = x_1 \vee x_2 \vee \cdots \vee x_n$$

    find all possible values for $x_1, x_2, \ldots, x_n$ which makes p true.
  - There are 2^{n-1} different solutions
  - To list them all, we need $\Theta(2^n)$ time

Problems with outputs of size $O(n^k)$

- The example exponential time problem given on the previous page is rather aritificial
- Since the output size of the problem is exponential in the input size, any algorithm for this problem needs exponential time
- We will restrict ourselves to the problems where the size of the output of the problem is bounded by a polynomial in the size of the input of the problem.
- e.g.: Given a boolean formula of the form

$$p = x_1 \vee x_2 \vee \cdots \vee x_n$$

find a possible value for $x_1, x_2, \ldots, x_n$ which makes p true.

Polynomial Solvability vs Polynomial Verifiability

- Consider the following problem:

  Given a boolean formula p on variables $x_1, x_2, \ldots, x_n$ (note that we do not restrict the form of the formula), find a value for $x_1, x_2, \ldots, x_n$ so that p becomes true?

- Note that a solution will be of length $n = O(n)$.
- A brute–force algorithm: try all possible values for $x_1, x_2, \ldots, x_n$, and for each combination check if p becomes true
- Looks like an exponential algorithm
- Can we do better?
- Well... Can you?

Polynomial Solvability vs Polynomial Verifiability

- Note that for the previous problem, the time consuming part is generating all the possible solutions
- Once we guess a solution, it takes only polynomial time to verify the correctness of the solution
- Assume that there is an oracle that tells us all possible values
- Also assume that we have an infinite amount of machines on which we can try the guesses simultaneously (non–deterministic machines)
- Under these assumptions, we can solve the problem in polynomial time

The class NP

- If given a guess g as the solution of the problem P, the solution g can be checked for being a solution to P in time $O(n^k)$, then P is said to be an NP problem.
- Note : NP does not stand for non–polynomial
- NP stands for

  “NONDETERMINISTICALLY POLYNOMIAL”
- The class NP: The set of all NP problems

An example NP problem

- Given $G = (V, E)$, two vertices $s, t \in V$ and an integer k, does there exist a path from s to t in G whose length is less than k?
- If we are given a path from s to t (it cannot have more than n edges in it), we can count the number of edges in the path in $O(n)$ time, and compare it with k.
- Hence we can verify a guess in polynomial time

P vs NP

- Note that if a problem is in P, then
  - The size of the solution must be bounded by a polynomial $O(n^k)$ (we can not generate more than one item in a single step)
  - Ignore the guessing part, just use the polynomial algorithm on a deterministic machine to find the result (which is verified by definition)
- Hence P is also in NP
- Therefore P $\subseteq$ NP

NP-P

- What about the set NP-P?
- Are there any problems in this set?

NP-P

- What about the set NP-P?
- Are there any problems in this set?
- The ugly truth : no one really knows!!!
- You can win $1.000.000 if you can prove either
  - Yes, there are problems in between
  - No, there isn't any problem in between

www.claymath.org/Millennium_Prize_Problems/P_vs_NP/

The class NP–Complete

- An interesting set of problems in NP
- The reason a lot of people believe (only believed, not proven) that P $\neq$ NP
- For an NP–Complete problem, no one was able to find a polynomial time algorithm
- The set of NP–Complete problems is growing
- The problems in the class NP–Complete have such a connection with each other that if someone finds a polynomial time algorithm for an NP–Complete problem, then all the NP (not only the NP–Complete) problems will be solvable in polynomial time, i.e. P=NP
- Also, if someone can show that an NP–Complete problem cannot be solved in polynomial time, then P $\neq$ NP

Waste of time?

- Unless you have a lot of time to spare, and you think you are one of the brightest brains in the world, do not try to find an efficient (i.e. polynomial time) algorithm for an NP–Complete problem
- If you face a problem for which you keep failing to come up with an efficient algorithm, immediately suspect that it is an NP–Complete problem
- In such a case, you'll have to prove that your problem is NP–Complete

Proving NP–Completeness

- Informally, a problem is NP–Complete if it is in NP and if it is as “hard” as any other problem in NP.
- The techniques used in NP–Completeness proofs are different than the techniques we used for analyzing algorithms
- Proving a problem to be NP–Complete is making a statement on at least how hard we think the problem is.

Transforming Problems

- Consider two decision problems P and Q.
- e.g. Let P be the decision problem related to maximum bipartite matching: Given a bipartite graph $G = (V, E)$, and an integer k, does there exist a matching $M \subseteq E$ s.t. $|M| > k$?
- e.g. Let Q be the decision problem related to maximum flow in a flow network: Given a flow network $G' = (V', E', s', t', c')$ and an integer k', does there exist a flow f' in G' such that $|f'| > k'$?
- Assume that $\exists$ a polynomial algorithm that solves the decision problem Q
- Finally assume that there exists a procedure that transforms any instance p of the problem P into an instance q of the problem Q satisfying:
  - The transformation procedure takes polynomial time
  - The answers to the instances p and q are the same
- The transformation procedure is called a *reduction algorithm*.

A Notion of Hardness

- If there exists a polynomial time reduction from P to Q, given an instance p of P, we can
  - transform p into a problem q in Q in polynomial time
  - use the known polynomial time algorithm to solve q
  - use the answer to q as the answer to p
- Therefore without fi nding a new polynomial time algorithm for P, we can solve it in polynomial time through the transformation (again in polynomial time)
- By reducing P to Q, we have proved P is no harder than Q
- Or in other words, we have proved P is at least as easy as Q.
- However, proving a problem to be NP–Complete is to show how hard it is, not how easy it is.
- To do this, we follow a similar but the reverse way thinking

Proving Hardness

- Assume we have two decision problems P and Q
- Assume there is a polynomial reduction from P to Q
- Assume no one has been able to fi nd a polynomial solution for P yet (so P is a really hard problem)
- Then Q must also be a hard problem
- If Q were easy, and we could fi nd a polynomial algorithm for Q, then we would simply transform the instances of P into Q, and solve P in polynomial time as well.
- Therefore in order to show that a problem Q is NP–Complete
  - we need fi rst to show that Q is NP
  - fi nd another problem P which is known to be NP–Complete
  - show that P can be transformed into Q in polynomial time

Let's get formal

- Let's use the notation $P \rightsquigarrow Q$ to denote that P is polynomial time reducible to Q.
- So, if $P \rightsquigarrow Q$, then in order to solve an instance p of P, we can transform p into an instance q of Q (in polynomial time), and then solve the instance q which would give us the corresponding solution for the instance p.
- **Definition:** A problem Q is NP–Complete if Q is in NP and for all NP problems P, $P \rightsquigarrow Q$.
- Since all the problems in NP can be transformed into NP–Complete problems, NP–Complete problems are "the hardest problems" in NP.

A useful Theorem

- However, in order to prove that a problem Q is NP–Complete , the defi nition given on the previous slide requires to show that for any problem P in NP, $P \rightsquigarrow Q$.
- How can we consider every problem $P \in$ NP, and show $P \rightsquigarrow Q$?
- If we don't know any NP–Complete problem, then we obviously have to do it.
- However, if we know an NP–Complete problem, then we can use the following theorem.
- **Theorem:** A problem Q is NP–Complete if Q is in NP and there exists an NP–Complete problem P such that $P \rightsquigarrow Q$.
- **Proof:** Since we know that P is NP–Complete , for any problem $P' \in$ NP, we must have $P' \rightsquigarrow P$. We also know that $P \rightsquigarrow Q$, hence $P' \rightsquigarrow P \rightsquigarrow Q$ implies $P' \rightsquigarrow Q$. That is, for any problem $P' \in$ NP, we have $P' \rightsquigarrow Q$. Since also $Q \in$ NP, Q is an NP–Complete problem.

First NP–Complete Problem

- In order to be able to use the theorem on the previous slide in order to show that a problem in NP–Complete , we need to know at least one NP–Complete problem.
- The first NP–Complete problem:

  SATISFIABILITY (SAT for short)
- Due to Stephen A. Cook(1982 Turing Award Winner): Showed SAT is NP–Complete using Definition 2 in the seminal paper:
  “The Complexity of Theorem Proving Procedures”, the 1971 ACM SIGACT Symposium on the Theory of Computing
- The proof is about 5 pages long

The Problem SAT

- Consider a boolean formula over a set of boolean variables $\{x_1, x_2, \ldots, x_n\}$ given in the conjunctive normal form. e.g:

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

- Does there exist a suitable value for $x_1, x_2, \ldots, x_n$ so that the formula evaluates to true?
- By using SAT as a seed, people have been proving lots of other problems to be NP–Complete via the theorem given before.

The Second NP–Complete Problem: 3–SAT

- Consider a boolean formula over a set of boolean variables $\{x_1, x_2, \ldots, x_n\}$ given in the conjunctive normal form, where each conjunct is a disjunction of exactly 3 variable (or their negations). e.g.

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

- Does there exist a suitable value for $x_1, x_2, \ldots, x_n$ so that the formula evaluates to true?
- Since the 3–SAT is more restricted than SAT, there is hope a that it will be solvable in polynomial time.
- However 3–SAT can easily shown to be NP–Complete using the theorem and SAT.

NP–Completeness of 3–SAT

- First of all, is 3–SAT in NP?
- Assume that an oracle guesses a solution: e.g. $x_1 = 1, x_2 = 0, \ldots, x_n = 1$
- We can check whether or not such a guess satisfies the formula in $O(m)$ time, where m is the length of the formula.
- Simply scan the formula from left to right to decide if there exists a conjunct that is not satisfied.
- Knowing that 3–SAT is in NP, we need also to show that an NP–Complete problem can be transformed into 3–SAT in polynomial time.
- Since SAT is the only NP–Complete problem we have, we have to show that SAT $\rightsquigarrow$ 3–SAT.

SAT $\rightsquigarrow$ 3–SAT

- To show SAT $\rightsquigarrow$ 3–SAT, we need to provide a transformation from SAT to 3–SAT such that
  - Transformation takes at most polynomial time
  - If for a formula p in SAT, our translation produces a formula p' in 3–SAT, p must be satisfiable iff p' is satisfiable
- We will consider each conjunct c in p separately and produce a number of conjuncts $c_1, c_2, \ldots, c_k$ to be used in p' instead of c
- We will treat c's differently depending on the number of disjuncts in c

Singleton Conjuncts

- When we see a conjunct of the form

$$\cdots \wedge (x) \wedge \cdots$$

introduce two new variables y_1 and y_2 and replace this conjunct with

$$\cdots \wedge (x \vee y_1 \vee y_2) \wedge (x \vee y_1 \vee y_2') \wedge (x \vee y_1' \vee y_2) \wedge (x \vee y_1' \vee y_2') \wedge \cdots$$

- If x' appears instead of x, just use x' instead.
- Note that, the conjuncts we have produced consist of 3 disjuncts to conform with 3–SAT
- 4 conjuncts for one conjunct in the original formula
- And most importantly

$$x \equiv (x \vee y_1 \vee y_2) \wedge (x \vee y_1 \vee y_2') \wedge (x \vee y_1' \vee y_2) \wedge (x \vee y_1' \vee y_2')$$

Conjuncts with 2 disjuncts

- When we see a conjunct of the form

$$\cdots \wedge (x^1 \vee x^2) \wedge \cdots$$

  introduce a new variable y and replace this conjunct with

$$\cdots \wedge (x^1 \vee x^2 \vee y) \wedge (x^1 \vee x^2 \vee y') \wedge \cdots$$

- If either x^1 or x^2 appears with their primed version (as negated), then use their primed version instead
- 2 conjuncts for one conjunct in the original formula
- Equivalence

$$(x^1 \vee x^2) \equiv (x^1 \vee x^2 \vee y) \wedge (x^1 \vee x^2 \vee y')$$

Conjuncts with 3 disjuncts

- They are already in the form we want
- When we see a conjunct of the form

$$\cdots \wedge (x^1 \vee x^2 \vee x^3) \wedge \cdots$$

  do nothing, just pass this conjunct to the output formula as is

$$\cdots \wedge (x^1 \vee x^2 \vee x^3) \wedge \cdots$$

- 1 conjunct for one conjunct in the original formula
- Equivalence: it is not changed

Conjuncts with more than 3 disjuncts

If there are k disjuncts

$$\cdots \wedge (x^1 \vee x^2 \vee \cdots \vee x^k) \wedge \cdots$$

introduce $k-3$ new variables $y_1, y_2, \ldots, y_{k-3}$, and replace this conjunct with

$$\cdots \wedge (x^1 \vee x^2 \vee y_1) \wedge (y_1' \vee x^3 \vee y_2) \wedge (y_2' \vee x^4 \vee y_3) \wedge \cdots \wedge (y_{k-4}' \vee x^{k-2} \vee y_{k-3}) \wedge (y_{k-3}' \vee x^{k-1} \vee x^k) \wedge \cdots$$

- Use prime version of x^i in the output if necessary
- $k-2$ conjuncts for 1 conjunct
- Equivalence

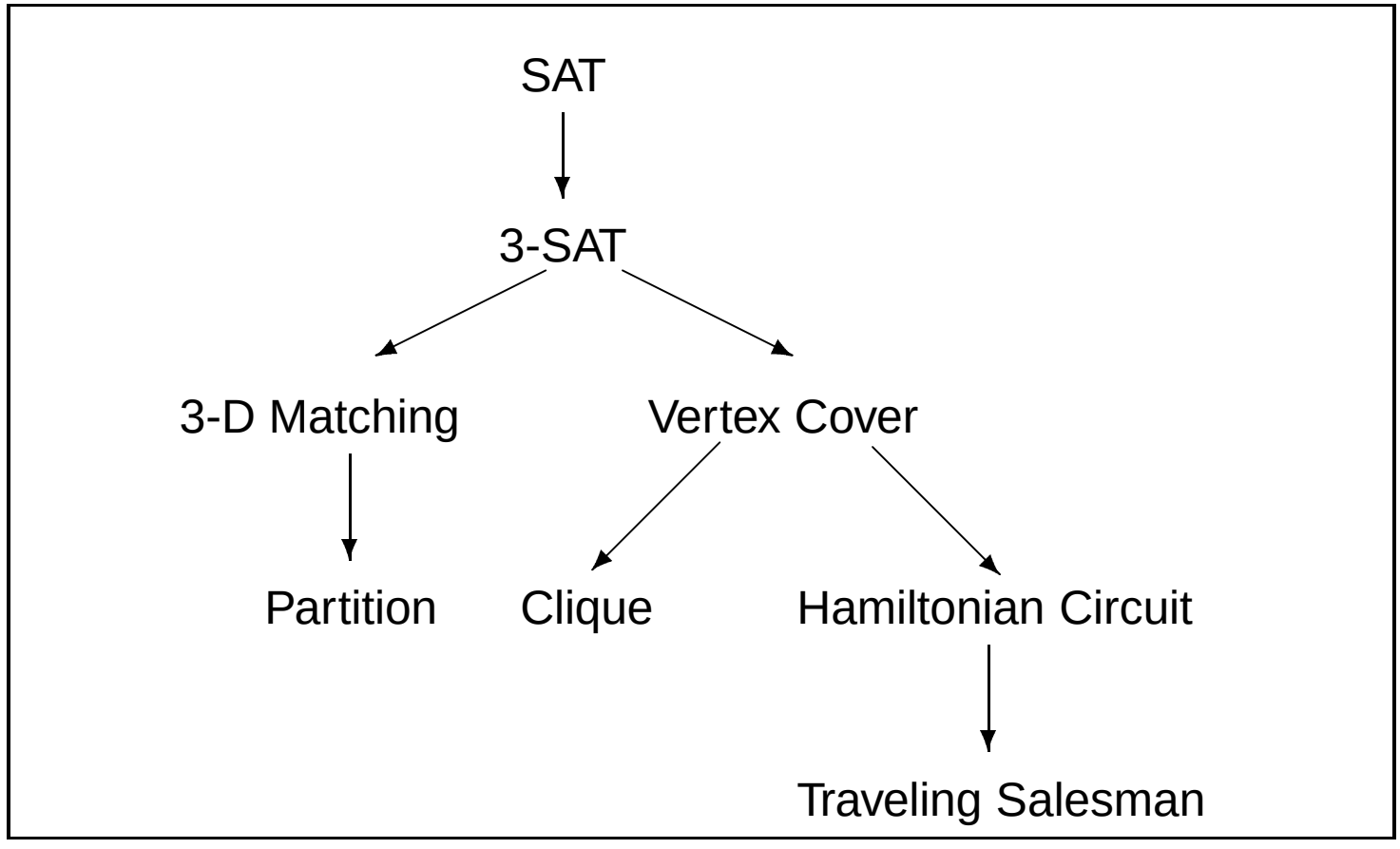
$$(x^1 \vee x^2 \vee \cdots \vee x^k) \equiv$$

$$(x^1 \vee x^2 \vee y_1) \wedge (y_1' \vee x^3 \vee y_2) \wedge (y_2' \vee x^4 \vee y_3) \wedge \cdots \wedge (y_{k-4}' \vee x^{k-2} \vee y_{k-3}) \wedge (y_{k-3}' \vee x^{k-1} \vee x^k)$$

About the transformation & the result

- Correctness: Since we produce an equivalent fragment for each conjunct, the output formula is satisfiable iff the original formula is satisfiable
- Is it a polynomial time transformation?
- In the worst case, we have l conjuncts in the original formula where in each conjunct all n variables appear: the size of the original problem is $l \times n$.
- We will produce $n - 2$ conjuncts for each conjunct in the input
- The size of the output : $l \times (n - 2) \times 3$
- By considering the conjuncts in the input from left–to–right, we can produce $l \times (n - 2)$ disjuncts in polynomial time.
- Hence: 3–SAT is our second NP–Complete problem

Six basic NP–Complete problems



Karp gave a list of 21 NP–Complete problems in "Reducibility among combinatorial problems", Complexity of Computer Computations, 1972

3–D Matching

- Definition: A set $M \subseteq W \times X \times Y$ where W, X, Y are some disjoint sets having the same number q of elements.
- Question: Does there exists $M' \subseteq M$ such that $|M'| = q$ and no two elements in M' agree in any coordinate?

Partition

- Definition: A finite set A and a "size" $s(a) \in \mathbb{Z}^+$ for each $a \in A$.
- Question: Is there a subset $A' \subseteq A$ such that

$$\sum_{a \in A'} s(a) = \sum_{a \in A - A'} s(a)$$

Vertex Cover

- Definition: A graph $G = (V, E)$ and a positive integer $K \leq |V|$.
- Question: Is there a *vertex cover* $V' \subseteq V$ such that $|V'| \leq K$ and for each edge $(u, v) \in E$, at least one of u and v belongs to V'.

Clique

- Definition: A graph $G = (V, E)$ and a positive integer $J \leq |V|$.
- Question: Is there a *clique* $V' \subseteq V$ such that $|V'| \geq J$ and every two vertices in V' are joined by an edge in E?

Hamiltonian Circuit

- Definition: A graph $G = (V, E)$
- Question: Does G contain an *hamiltonian circuit*, that is, an ordering $< v_1, v_2, \ldots, v_n >$ of the vertices of G, where $n = |V|$, such that $(v_n, v_1) \in E$, and $\forall 1 \leq i < n$: $(v_i, v_{i+1}) \in E$?
- Intuitively: Can you find a tour in G by visiting every vertex exactly once?
- Euler Tour: Can you find a tour in G by passing through every edge exactly once?
- Checking if (even finding the edges) G has an Euler tour can be solved in $O(E)$ time, whereas quite a similar problem, namely Hamiltonian Circuit is NP–Complete

Traveling Salesman

- A salesman must visit n cities. Starting from one of the cities, every city will be visited exactly once, and the tour will be finished at the starting city. What is the length of the shortest such tour?
- Definition: A graph $G = (V, E)$, $c : V \times V \to \mathbb{Z}^+$, and a positive integer K.
- Question: Is there a hamiltonian circuit $< v_1, v_2, \ldots, v_n >$ in G such that

$$\sum_{1 \leq i < n} c(v_i, v_{i+1}) + c(v_n, v_1) \leq K$$