Decision Problems

- For rather technical reasons, most NPcomplete problems that we will discuss will be phrased as decision problems.
- A problem is called a ***decision problem*** if its output is a simple ``yes" or ``no" (or you may think of this as True/False, 0/1, accept/reject).
- We will phrase many optimization problems in terms of decision problems. For example, the minimum spanning tree decision problem might be: Given a weighted graph *G* and an integer *k*, does *G* have a spanning tree whose weight is at most *k*?
- This may seem like a less interesting formulation of the problem. It does not ask for the weight of the minimum spanning tree, and it does not even ask for the edges of the spanning tree that achieves this weight.
- However, our job will be to show that certain problems *cannot* be solved *efficiently.*
- If we show that the simple decision problem cannot be solved efficiently, then the more general optimization problem certainly cannot be solved efficiently either.

Language Recognition Problems

- Observe that a decision problem can also be thought of as a ***language recognition problem.*** We could define a language L

$L = \{(G, k) : G$ has a MST of weight at most $k\}$:

- This set consists of pairs, the first element is a graph (e.g. the adjacency matrix encoded as a *string*) followed by an integer *k* encoded as a binary number.
- At first it may seem strange expressing a graph as a string, but obviously anything that is represented in a computer is broken down somehow into a string of bits.
- When presented with an input string *(G, k),* the algorithm would answer ``yes" if $(G, k) \in L$ implying that *G* has a spanning tree of weight at most *k*, and ``no" otherwise.
- In the first case we say that the algorithm ``*accepts*" the input and otherwise it ``*rejects*" the input.
- Given any language, we can ask the question of how hard it is to ***determine whether a given string is in the language***.
- For example, in the case of the MST language *L*, we can determine membership easily in polynomial time. We just store the graph internally, run Kruskal's algorithm, and see whether the final optimal weight is at most *k*. If so we accept, and otherwise we reject.

Complexity Classes: Definitions

- Define **P** to be the set of all languages for which ***membership can be tested in polynomial time.*** (Intuitively, this corresponds to the set of all decisions problems that can be solved in polynomial time.)

- Note that languages are sets of strings, and **P** is a set of languages. **P** is defined in terms of how hard it is computationally to recognized membership in the language.

- A set of languages that is defined in terms of how hard it is to determine membership is called a ***complexity class***.

- Since we can compute minimum spanning trees in polynomial time, we have $L \in$ **P**.

- Here is a harder one. $M = \{(G, k)$: G has a simple path of length at least $k\}$

- Given a graph G and integer k how would you ``recognize" whether it is in the language M?

- You might try searching the graph for a simple paths, until finding one of length at least k.

- If you find one then you can accept and terminate. However, if not then you may spend a lot of time searching (especially if k is large, like $n-1$, and no such path exists). So is $M \in$ **P**? No one knows the answer. In fact, we will see that M is **NP**-complete.

- In what follows, we will be introducing a number of classes. We will jump back and forth between the terms ``language" and ``decision problems", but for our purposes they mean the same things.

Polynomial Time Verification and Certificates

- Before talking about the class of **NPcomplete** problems, it is important to introduce the notion of ***a verification algorithm***.

- Many language recognition problems that may be very hard to solve, but they have the property that it is easy to ***verify*** whether a string is in the language.

- Consider the following problem, called ***the Hamiltonian cycle problem***.

- Given an undirected graph G, does G have a cycle that visits every vertex exactly once. (There is a similar problem on directed graphs, and there is also a version which asks whether there is a path that visits all vertices.)

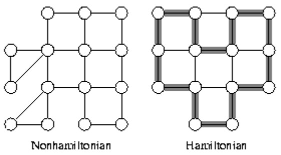
- We can describe this problem as a language recognition problem, where the language is
  $HC = \{(G)$: G has a Hamiltonian cycle$\}$,
  where (G) denotes an encoding of a graph G as a string.

- The Hamiltonian cycle problem seems to be much harder, and there is no known polynomial time algorithm for this problem.

- For example, the figure below shows two graphs, one which is Hamiltonian and one which is not. However, suppose that a graph did have a Hamiltonian cycle. Then it would be a very easy matter for someone to convince us of this. They would simply say ``the cycle is $\langle v_3, v_7, v_1, \dots, v_{13} \rangle$“.

- We could then inspect the graph, and check that this is indeed a legal cycle and that it visits all the vertices of the graph exactly once.

- Thus, even though we know of no efficient way to solve the Hamiltonian cycle problem, there is a very efficient way to *verify* that a given graph is in HC.



Hamiltonian cycle.

Polynomial Time Verification and Certificates (cont.)

- The given cycle is called a ***certificate***. This is some piece of information which allows us to verify that a given string is in a language.

- More formally, given a language L, and given $x \in L$, a ***verification algorithm*** is an algorithm which given x and a string y called the *certificate,* can verify that x is in the language L using this certificate as help. If x is not in L then there is nothing to verify.

- Note that *not all* languages have the property that they are easy to verify. For example, consider the following languages:

  $UHC = \{(G) : G$ has a unique Hamiltonian cycle$\}$,
  $\overline{HC} = \{(G) : G$ has no Hamiltonian cycle$\}$.

- Suppose that a graph G is in the language UHC. What information would someone give us that would allow us to verify that G is indeed in the language? They could give us an example of the unique Hamiltonian cycle, and we could verify that it is a Hamiltonian cycle, but what sort of certificate could they give us to convince us that this is the only one?

- They could give another cycle that is NOT Hamiltonian, but this does not mean that there is not another cycle somewhere that is Hamiltonian. They could try to list every other cycle of length n, but this would not be at all efficient, since there are $n!$ possible cycles in general. Thus, it is hard to imagine that someone could give us some information that would allow us to efficiently convince ourselves that a given graph is in the language.

The Class **NP**

Definition: Define **NP** to be the set of all languages that *can be verified by a polynomial time algorithm.*

- Why is the set called ``**NP**" rather than ``**VP**"? The original term **NP** stood for ``non-deterministic polynomial time". This referred to a program running on a *non-deterministic computer* that can make guesses. Basically, such a computer could non-deterministically guess the value of certificate, and then verify that the string is in the language in polynomial time.

- We have avoided using non-determinism here.

- Like **P**, **NP** is a set of languages based on some complexity measure (the complexity of verification). Observe that **P**$\subseteq$ **NP**. In other words, if we can solve a problem in polynomial time, then we can certainly verify membership in polynomial time. (More formally, we do not even need to see a certificate to solve the problem, we can solve it in polynomial time anyway).

- However it is not known whether **P = NP**. It seems unreasonable to think that this should be so. In other words, just being able to verify that you have a correct solution does not help you in finding the actual solution very much.

- Most experts believe that **P** $\neq$ **NP**, but no one has a proof of this.

- Next we will define the notions of **NPhard** and **NPcomplete**.

Summary

•The following concepts are important.

• ***Decision Problems:*** are problems for which the answer is either *yes* or *no*. NP-complete problems are expressed as decision problems, and hence can be thought of as language recognition problems, assuming that the input has been encoded as a string. We encode inputs as strings.

• For example: HC = {G : G has a Hamiltonian cycle}

MST = {($G; x$): G has a MST of cost at most x}.

• **P:** is the class of all decision problems which can be solved in polynomial time, $O(n^k)$ for some constant k. For example MST∈**P** but HC is not known (and suspected not) to be in **P**.

• ***Certificate:*** is a piece of evidence that allows us to verify in polynomial time that a string is in a given language. For example, suppose that the language is the set of Hamiltonian graphs. To convince someone that a graph is in this language, we could supply the certificate consisting of a sequence of vertices along the cycle. It is easy to access the adjacency matrix to determine that this is a legitimate cycle in G. Therefore HC∈**NP**.

• **NP:** is defined to be the class of all languages that can be verified in polynomial time. Note that since all languages in **P** can be solved in polynomial time, they can certainly be verified in polynomial time, so we have **P**⊆ **NP**. However, **NP** also seems to have some pretty hard problems to solve, such as HC.

NP-Completeness: Reductions

• The class of **NPcomplete** problems consists of a set of decision problems (languages) (a subset of the class **NP**) that no one knows how to solve efficiently, but if there were a polynomial time solution for even a single **NPcomplete** problem, then every problem in **NP** would be solvable in polynomial time.

• To establish this, we need to introduce the concept of a ***reduction.***

• Before discussing reductions, let us just consider the following question. Suppose that there are two problems, *A* and *B*. You know (or you strongly believe at least) that it is impossible to solve problem *A* in polynomial time. You want to prove that *B* cannot be solved in polynomial time. How would you do this?

• We want to show that (*A*∉ **P**) ➔ (*B*∉ **P**).

• To do this, we could prove the contrapositive, (*B* ∈**P**) ➔ (*A* ∈ **P**):

• In other words, to show that *B* is not solvable in polynomial time, we will suppose that there is an algorithm that solves *B* in polynomial time, and then derive a contradiction by showing that *A* can be solved in polynomial time.

• How do we do this? Suppose that we have a subroutine that can solve any instance of problem *B* in polynomial time. Then all we need to do is to show that we can use this subroutine to solve problem *A* in polynomial time. Thus we have ``reduced" problem *A* to problem *B*.

• It is important to note here that this supposed subroutine is really a *fantasy*. We know (or strongly believe) that *A* cannot be solved in polynomial time, thus we are essentially proving that the subroutine cannot exist, implying that *B* cannot be solved in polynomial time.

• Be sure that you understand this, this is the basis behind all reductions.

Example: 3-Colorability and Clique Cover

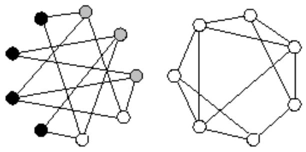
• Let us consider an example to make this clearer. The following problem is well-known to be **NPcomplete**, and hence it is strongly believed that the problem cannot be solved in polynomial time.

3coloring (3Col): Given a graph *G*, can each of its vertices be labeled with one of 3 different ``colors", such that no two adjacent vertices have the same label.

• Coloring arises in various partitioning problems, where there is a constraint that two objects cannot be assigned to the same set of the partition. The term ``coloring" comes from the original application which was in map drawing. Two countries that share a common border should be colored with different colors. It is well known that planar graphs can be colored with 4 colors, and there exists a polynomial time algorithm for this. But determining whether 3 colors are possible (even for planar graphs) seems to be hard and there is no known polynomial time algorithm.

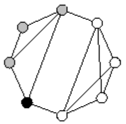
• In the figure below we give two graphs. One which can be colored with 3 colors, and one that cannot.

• The ***3Col*** problem will play the role of problem *A*, which we strongly suspect to not be solvable in polynomial time.



• For our problem *B*, consider the following problem.

• Given a graph $G = (V, E)$, we say that a subset of vertices $V' \subseteq V$ forms a ***clique*** if for every pair of vertices $u, v \in V'$, $(u, v) \in E$. That is, the subgraph induced by V' is a complete graph.



Clique Cover (CCov): Given a graph *G* and an integer *k*, can we find *k* subsets of vertices $V_1, V_2, \ldots, V_k$, such that, $\bigcup_i V_i = V$ and that each V_i is a clique of *G*.

• The clique cover problem arises in applications of clustering. We put an edge between two nodes if they are similar enough to be clustered in the same group. We want to know whether it is possible to cluster all the vertices into *k* groups.

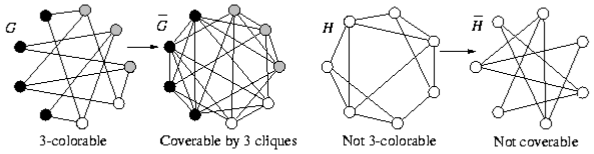
• Suppose that you want to solve the ***CCov*** problem, but after a while of fruitless effort, you still cannot find a polynomial time algorithm for the ***CCov*** problem. How can you prove that ***CCov*** is likely to not have a polynomial time solution?

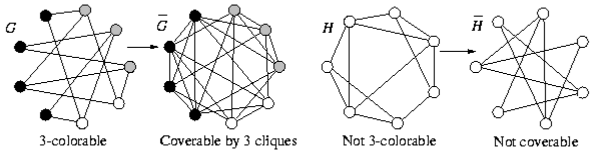
• You know that ***3Col*** is **NPcomplete**, and hence experts believe that ***3Col*** ∉ **P**. You feel that there is some connection between the ***CCov*** problem and the ***3Col*** problem.

• Thus, you want to show that (***3Col*** ∉ **P**) ➔ (***CCov*** ∉ **P**), which you will show by proving the contrapositive (***CCov*** ∈ **P**) ➔ (***3Col*** ∈ **P**).

• To do this, you assume that you have access to a subroutine ***CCov(G, k).*** Given a graph *G* and an integer *k*, this subroutine returns *true* if *G* has a clique cover of size *k* and *false* otherwise, and furthermore, this subroutine runs in polynomial time.

- How can we use this ``alleged" subroutine to solve the wellknown hard ***3Col*** problem?
- We want to write a polynomial time subroutine for ***3Col***, and this subroutine is allowed to call the subroutine ***CCov(G, k)*** for any graph *G* and any integer *k*.
- Both problems involve partitioning the vertices up into groups. The only difference here is that in one problem the number of cliques is specified as part of the input and in the other the number of color classes is fixed at 3.
- In the clique cover problem, for two vertices to be in the same group they must be adjacent to each other. In the 3coloring problem, for two vertices to be in the same color group, they must not be adjacent.
- In some sense, the problems are almost the same, but the requirement adjacent/ nonadjacent is exactly reversed.
- We claim that we can reduce the 3coloring problem to the clique cover problem as follows.
- Given a graph *G* for which we want to determine its 3colorability, output the pair ($\overline{G}$, *3*) where $\overline{G}$ denotes the complement of *G*. We can then feed the pair ($\overline{G}$, *3*) into a subroutine for clique cover. This is illustrated in the figure below.





Claim: A graph *G* is 3colorable if and only if its complement $\overline{G}$ has a cliquecover of size 3.

In other words, *G*∈ ***3Col*** iff ($\overline{G}$, *3*)∈ ***CCov.***

Proof: (➔) If *G* 3colorable, then let V_1, V_2, V_3 be the three color classes. We claim that this is a clique cover of size 3 for $\overline{G}$, since if *u* and *v* are distinct vertices in V_i , then *(u, v)*∉ *E(G)* (since adjacent vertices cannot have the same color) which implies that *(u, v)*∈ *E($\overline{G}$).* Thus every pair of distinct vertices in V_i are adjacent in $\overline{G}$.

(🡐) Suppose $\overline{G}$ has a clique cover of size 3, denoted V_1, V_2, V_3 . For *i =1,2,3* give the vertices of V_i color *i*. We assert that this is a legal coloring for *G*, since if distinct vertices *u* and *v* are both in V_i , then *(u, v)* ∈ *E(G)* (since they are in a common clique), implying that *(u, v)* ∉ *E($\overline{G}$).* Hence, two vertices with the same color are not adjacent.

- We now take this intuition of reducing one problem to another through the use of a subroutine call, and place it on more formal footing.
- Notice that in the example above, we converted an instance of the 3coloring problem (*G*) into an equivalent instance of the Clique Cover problem ($\overline{G}$, *3*).

Polynomial – Time Reduction

Definition: We say that a language (i.e. decision problem) *L1* is ***polynomialtime reducible to*** language *L2* (written $L1 \prec_P L2$) if there is a polynomial time computable function *f* , such that for all *x*, $x \in L1$ if and only if $f(x) \in L2$.

- In the previous example we showed that ***3Col*** $\prec_P$ ***CCov.***
- In particular we have $f(G) = (\overline{G}, 3)$. Note that it is easy to complement a graph in $O(n^2)$ (i.e. polynomial) time (e.g. flip *0*'s and *1*'s in the adjacency matrix). Thus *f* is computable in polynomial time.
- Intuitively, saying that $L1 \prec_P L2$ means that ``if *L2* is solvable in polynomial time, then so is *L1* ." This is because a polynomial time subroutine for *L2* could be applied to *f(x)* to determine whether $f(x) \in L2$, or equivalently whether $x \in L1$.
- Thus, in sense of polynomial time computability, *L1* is ``no harder" than *L2* .
- The way in which this is used in **NPcompleteness** is exactly the converse. We usually have strong evidence that *L1* is not solvable in polynomial time, and hence the reduction is effectively equivalent to saying ``since *L1* is not likely to be solvable in polynomial time, then *L2* is also not likely to be solvable in polynomial time."
- Thus, this is how polynomial time reductions can be used to show that problems are as hard to solve as known difficult problems.

Lemma : If $L1 \prec_P L2$ and $L2 \in$ P then $L1 \in$ P.
Lemma : If $L1 \prec_P L2$ and $L1 \notin$ P then $L2 \notin$ P.

NP-Completeness

- One important fact about reducibility is that it is transitive. In other words

Lemma : If $L1 \prec_P L2$ and $L2 \prec_P L3$ then $L1 \prec_P L3$.

- The reason is that if two functions *f(x)* and *g(x)* are computable in polynomial time, then their composition *f(g(x))* is computable in polynomial time as well.

NPcompleteness: The set of **NPcomplete** problems are all problems in the complexity class **NP**, for which it is known that if any one is solvable in polynomial time, then they all are, and conversely, if any one is not solvable in polynomial time, then none are.

- This is made mathematically formal using the notion of polynomial time reductions.

Definition: A language *L* is **NPhard** if: $L' \prec_P L$ for all $L' \in$ **NP**.

Definition: A language *L* is **NPcomplete** if: $L \in$ **NP**, and *L* is **NPhard**.

- An alternative (and usually easier way) to show that a problem is NPcomplete is to use transitivity.

Lemma: *L* is **NPcomplete** if (1) $L \in$ **NP** and
(2) $L' \prec_P L$ for some known **NPcomplete** language *L'*.

- The reason is that all $L'' \in$ **NP** are reducible to *L'* (since *L'* is **NPcomplete** and hence **NPhard**) and hence by transitivity *L''* is reducible to *L*, implying that *L* is **NPhard**.

NP-Completeness (cont.)

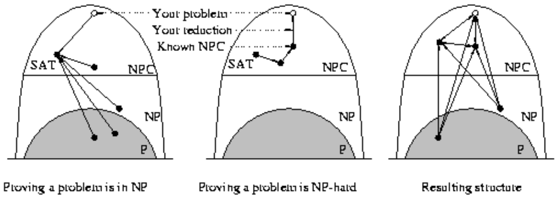
- This gives us a way to prove that problems are **NPcomplete**, once we know that one problem is **NPcomplete**.

- Unfortunately, it appears to be almost impossible to prove that one problem is **NP-complete**, because the definition says that we have to be able to reduce every problem in **NP** to this problem.

- There are infinitely many such problems, so how can we ever hope to do this?

- We will talk about this next time with **Cook's theorem**. Cook showed that there is one problem called **SAT** (short for **boolean satisfiability**) that is **NPcomplete**.

- To prove a second problem is **NPcomplete**, all we need to do is to show that our problem is in **NP** (and hence it is reducible to **SAT**), and then to show that we can reduce **SAT** (or generally some known **NPC** problem) to our problem. It follows that our problem is equivalent to **SAT** (with respect to solvability in polynomial time). This is illustrated in the figure below.



Proving a problem is in NP

Proving a problem is NP-hard

Resulting structure

Structure of NPC and reductions.