

Design and Analysis of Algorithms, Chennai Mathematical Institute
Prof. Madhavan Mukund
Department of Computer Science and Engineering,

Week - 08
Module - 06
Lecture - 55

Intractability: Checking Algorithms

Most of this course has been about identifying efficient solution to problems. But it is also important to realize that there are some situations, where no known efficient solutions exists, and we are able to recognize this, so that we do not fruitlessly try to look for solutions, where the problem is known to be hard to solve. So, let us look at some issues involving intractability.

(Refer Slide Time: 00:24)



Efficient algorithms

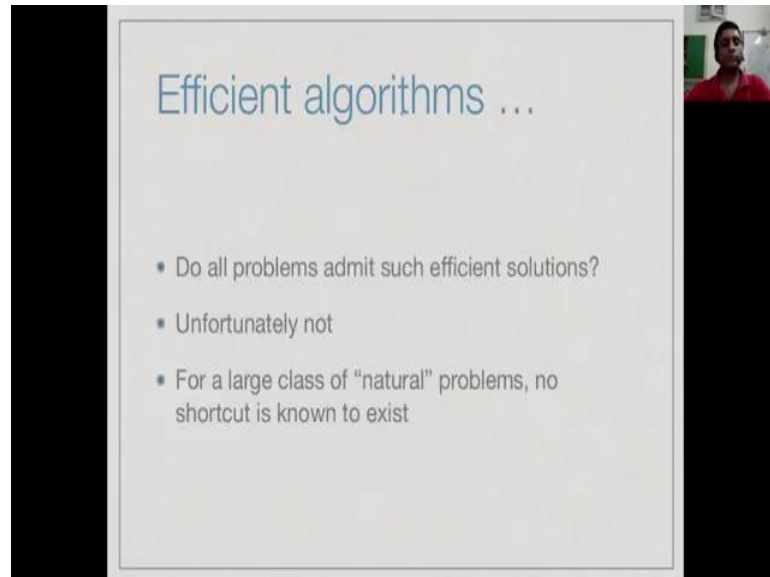
- Shortest path, minimum cost spanning tree, maximum flow, ... have polynomial time algorithms
- Search space for solutions is exponential
 - All possible paths, all possible spanning trees, all possible subsets of edges, ...
 - Brute force: scan exponential possibilities and choose the best

So, many of the problems that we have seen, we are trying to search through a number of possibilities to arrive at some kind of optimum combination. The actual search space is exponential, if we are looking for shortest paths, there are an exponential number of paths, if we are looking for a minimum cost spanning tree, there are an exponential number of such spanning trees to search through.

We are looking for the maximum flow, we have many different ways of adding and subtracting flow along edges. So, if you look at all possible flows, all possible paths, all possible spanning trees and then choose the optimum, this will be a Brute force solution which would take exponential time. When we have a polynomial time algorithm, we are

actually cutting through the exponentials and in some very drastic way, reducing our search space from an exponential one to a polynomial one.

(Refer Slide Time: 01:15)



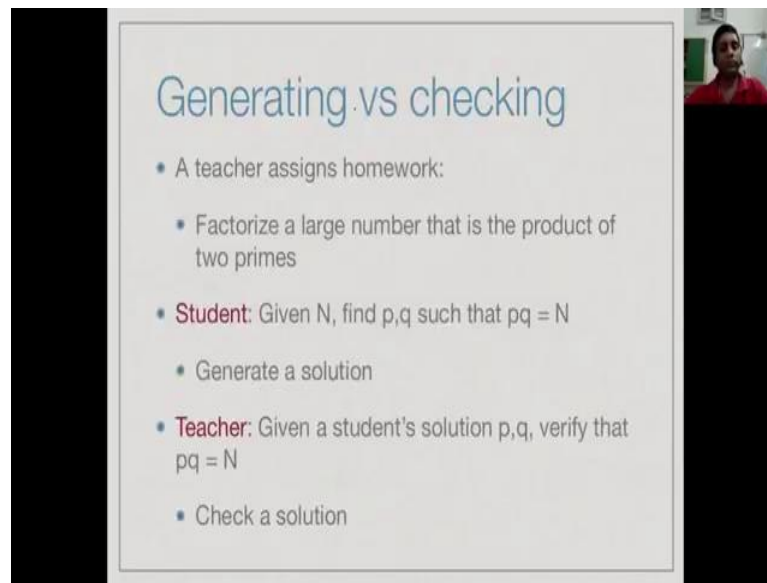
Efficient algorithms ...

- Do all problems admit such efficient solutions?
- Unfortunately not
- For a large class of "natural" problems, no shortcut is known to exist

So, it is tempting to believe that if one just thinks long enough and hard enough, that any such problems, we will always find such an efficient short cut, where we can cut through the exponential space of possibilities and quickly narrow down the space to a polynomial number of realistic ones from which the efficient solution, the actual solution we want will emerge.

Now, unfortunately this ideal world is not actually the way the world is. So, there are many problems, so which efficient algorithm do not exist or are not known to exist and unfortunately also many of these problems are extremely important practical problems.

(Refer Slide Time: 01:56)



Generating vs checking

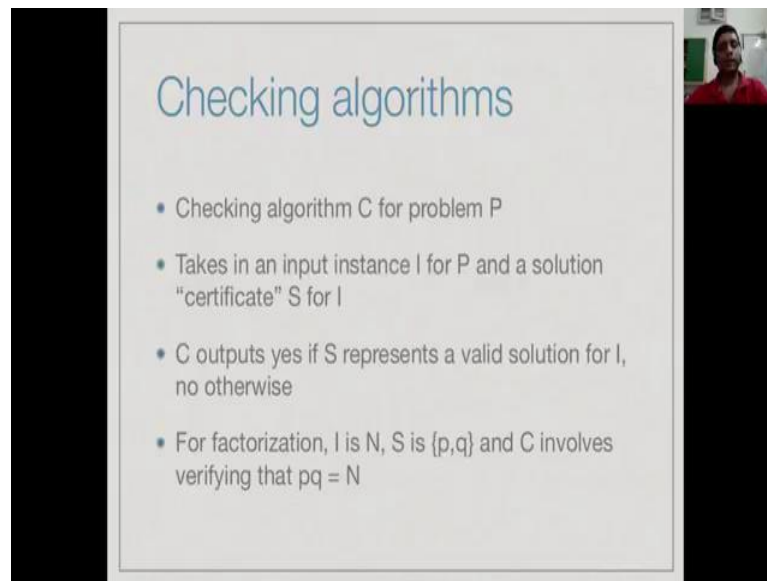
- A teacher assigns homework:
 - Factorize a large number that is the product of two primes
- **Student:** Given N , find p, q such that $pq = N$
 - Generate a solution
- **Teacher:** Given a student's solution p, q , verify that $pq = N$
 - Check a solution

So, to get into this discussion, let us talk about the problem between, the difference between generating a solution and checking a solution. So, supposing a school Math's teacher assigns the following homework, take a large number which is known to be the product of two prime numbers and find these two prime numbers. From the student point of view obviously, the problems is to generate the solution. So, given the large number N , the student is expected to find two prime numbers p and q , such that p times q is equal to N .

Now, the student submits or the students submit their solutions to teacher for evaluation. So, the teacher is in much better respective, the teacher does not have to keep generating p and q , the teacher has does not even need to know the answer. The teacher can just take the answer given by the student, multiply p times q and determine whether p times q is equal to N or not.

So, even without knowing the answer or even if the answer is wrong, the teacher may not know the right answer, the teacher can decide whether or not the student as given the correct answer. So, what the teacher is doing is checking the solution that the student was trying to do or generate the solution.

(Refer Slide Time: 03:12)



Checking algorithms

- Checking algorithm C for problem P
- Takes in an input instance I for P and a solution "certificate" S for I
- C outputs yes if S represents a valid solution for I, no otherwise
- For factorization, I is N, S is {p,q} and C involves verifying that $pq = N$

So, this gives us some notion of a checking algorithm. So, if you have a problem, I can say that I have a checking algorithm, if for every instance I can take a solution to that instance and quickly verify whether or not that solution is actually a valid one. So, checking algorithm takes an input for the problem, a solution which is may be not just the solution, but solution plus some extra information, and then it determine whether or not, this is a valid solutions, if so it says that the solution is correct and outputs S, otherwise it says no.

So, in our example before the factorization example, the input instance is the number N to be factorized. The solution that we get as a candidate to solve the problem is the pair of primes p and q that the student as calculated and the checking algorithm involves verifying that p times q is actually N.

(Refer Slide Time: 04:12)

The slide is titled "Boolean satisfiability" in blue. It contains a bulleted list of definitions: Boolean variables x, y, z, \dots ; $\neg x$ as negation of x ; $x \vee y$ as x or y ; $x \wedge y$ as x and y ; Clause as formula C of the form $(x \vee \neg y \vee z \vee \dots \vee w)$; Disjunction of literals (variables, negated variables); and Formula as conjunction of clauses. Handwritten green notes include "True False" with arrows pointing to the variables, and a diagram showing a clause $C \wedge D \wedge \dots \wedge E$ where C is $x \vee y$ and D is $\neg z \vee \neg y$.

So, in this context, let us look at a very canonical problem which has a checking algorithm called Boolean satisfactory. So, we have some Boolean variables x , y and z . So, Boolean variables can take values, true or false and we have a standard operations on Boolean variables, negation takes a value of true and transit false and vise verses. So, we write that with this exclamation mark using some programming language terminologies, so not x it is used an exclamation mark x .

Then, we have x or y which is true provided at least one of them is true, so if either x is true or y is true or both are true, x or y is true, we write that with this vertical bar or pair of vertical bars and use an ampersand denote AND, x and y is true only if both are true. So, x must be true and y must be true, either one of is false, x and y is false. So, this is the standard Boolean operations that we know about Boolean variables.

Now, we set up Boolean formulas in a very special form, we said first construct what we called are clauses. So, clause is a big disjunction, it is x or not y or z and something. So, what is the inside the disjunction or either variables or their negations. So, these are called literals, so literal is either a variable or a negated variable. So, clause is read as a disclause x or not y or z dot, dot, dot or w . So, this is a clause and in finally, a formula will be a combination of such clauses connected by AND.

So, C will be some x or y or something, D will be some z or not y or something and so on and so each clause will have this structure of big disjunction of literals and then they are combined by an ampersand, so they are all handled together. So, I need to make

clause C true and I need to make clause D true and I need to make clause E true and so on.

(Refer Slide Time: 06:05)

Boolean satisfiability

- Assign suitable values (True, False) to x, y, z, \dots so that the formula evaluates to true

$$(x \parallel y \parallel z) \& (x \parallel !y) \& (y \parallel !z) \& (!x \parallel !y \parallel !z)$$

- $x = \text{True}, y = \text{True}, z = \text{False}$ makes this true

$$(x \parallel y \parallel z) \& (x \parallel !y) \& (y \parallel !z) \& (z \parallel !x) \& (!x \parallel !y \parallel !z)$$

- Now there is no satisfying assignment

So, our goal is to find out whether this given formula can be made true by assigning suitable values to x , y and z ; all the variables in the formula. So, this is called valuation. Valuation is a function that says x is true, y is false, z is true and so on, so it fixes the value of each Boolean variable. Now, having fix that I can evaluate, so for example, if I take x to be true, y to be true and z to be false, then here from since in this clause, x is true so that is enough to make the whole clause true, y is also true, in this clause for instance x is true, so this part is true, x is true.

Here, it says y is true, z is false, so both of these are actually true and now, here it says x is false not x , but x is true, so not x is false, y is true by our evaluations not y is false, but fortunately z is false, so not z is true. So, in each of these clauses at least one of the literals becomes true under this valuation, so this entire formula is actually satisfied. Now, if I had this extra clause here to the same formula, now it terms out, there is no way to find any way of assigning true false to x , y , z make it.

You can check that this particular evaluation does not work, because if I look at this it such that x is true, so this does not work and it says that z is false, so it does not works. So, z is false, so z is not true, x is true, so not x is false, so this is false or false. So, this clause actually valued false. So, this particular evaluation which make the first formula true, it does not make the second formula true.

(Refer Slide Time: 07:50)

Boolean satisfiability

- Assign suitable values {True, False} to x, y, z, \dots so that the formula evaluates to true

$(x \vee y \vee z) \wedge (x \vee \neg y) \wedge (y \vee \neg z) \wedge (\neg x \vee \neg y \vee \neg z)$

- $x = \text{True}, y = \text{True}, z = \text{False}$ makes this true

$(x \vee y \vee z) \wedge (x \vee \neg y) \wedge (y \vee \neg z) \wedge (\neg x \vee \neg y \vee \neg z)$

- Now there is no satisfying assignment

$y \rightarrow x \quad z \rightarrow y \quad x \rightarrow z$

And in particular no valuation will actually make it true and that can actually be checked if you look at these three clauses, this can be said as that y implies x , what this says is that if y is true, then x must be true. And this says that if z is true, then y must be true and this says that if x is true, then z must be true, so this more or less says that x , y and z must all be the same. But if then x , y and z all the same, then either one of the first of the last was going to be false, actually there is no satisfying assignment. So, our goal is to find out whether there is a satisfying assignment or not.

(Refer Slide Time: 08:28)

Boolean satisfiability

- Generating a solution
 - Try each possible assignment to x, y, z, \dots
 - N variables — 2^N possible assignments
 - Is there a better algorithm? Not known
- Checking a solution
 - Given formula F and valuation $V(x)$ for each x , substitute into formula and evaluate

So, to generate a solution, of course the Brute force approach is to try assigning each of x , y and z ; true and false and turn, and then once we assigned x , y and z to be true, false;

we can evaluate and check if the formula is true and we try this for every possible such assignment. If there are N variables, each of them has two possibilities, clearly there are 2^N possible assignments.

Now, the amazing thing is that in general no better algorithm is known for this problem, at the moment as things stand, we do not know an efficient way to take a formula on this form and find out, whether it has a satisfying assignment. However, it is easy to check that it has an algorithm which can check a solution. So, if I give you a formula and I claim that a given way valuation is actually a satisfying assignment, all I have to do, I plug in that assignment, like we did for the earlier case.

I consider ((Refer Time: 09:25)) $\forall x$ is true, let me put true everywhere as C x , you say V y is false, let me put false everywhere at C y , and then evaluate the formula, find out whether the AND's and OR's had a true answer or not. So, it is easy to see that this has the checking algorithm, but it does not have a generating algorithm, at least we do not know a path.

(Refer Slide Time: 09:45)

Boolean satisfiability

- Input format is important
- Suppose a clause is a conjunction of literals ...

$$\begin{matrix} T & T & T & & T \\ (x & \& !y & \& z & \& \dots & \& w) \end{matrix} \quad (!y \dots \& w)$$
- ... and a formula is a disjunction of clauses

$$C \parallel D \parallel \dots \parallel E$$
- Each clause forces a unique valuation
- Try each clause in sequence

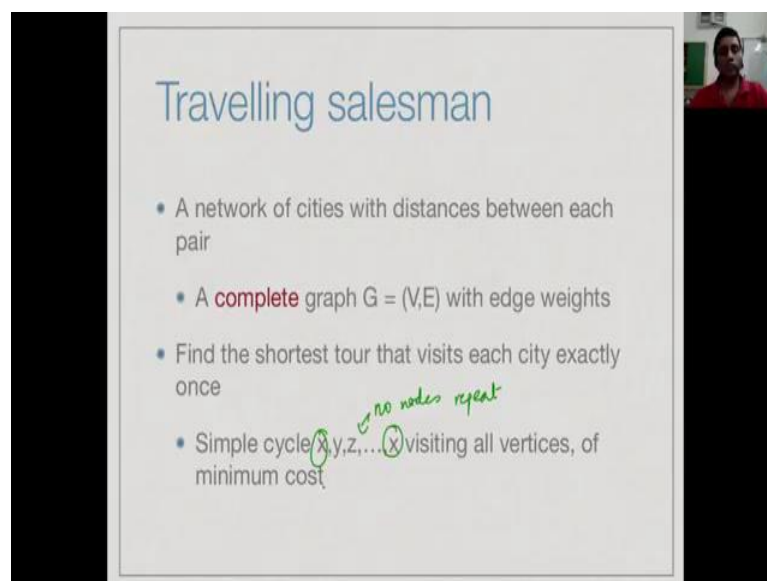
So, in this case, it please turn and other case is also, sometimes your presentation of the problem is important. So, we said that a clause was a disjunction of literals, and then the formula was a conjunction of clauses, what would be a reverse says, what would be say clauses are conjunction of literals. So, in a clause, I connect everything with AND, and then I connect the clauses with OR, this means that I must makes C true or D true or E true.

Earlier, I doing C true and D true and E true, now if I look inside a clause, how do I make a conjunction like this prove, well you sees red, I must make each of this true and else I make everything true, this whole AND will fail. So, if I take a clause, then the settings of the variables inside the clause four sneeze, it says x must be true, y must be not y must be true, so y must be false and z must be true and so on, so I do not have much choice.

Now, inside a clause, I am might see not y and then somewhere else I may see y. So, therefore, such a thing will say I am being asked to say y true and y false, so this clause cannot be true, then I move to the next. So, I look at each clause, I check whether the unique valuations is this on plausible or feasible, if it is at null, because I only need to satisfy one clause, other ways I move to the next one.

So, in a linear scan from left to right, I can basically solve this problem, if it is given to me in this form, whereas in the earlier form I need, there is no efficient algorithm. So, cleared with the presentation the problem is important.

(Refer Slide Time: 11:13)



The slide is titled "Travelling salesman" in blue text. It contains a list of four bullet points:

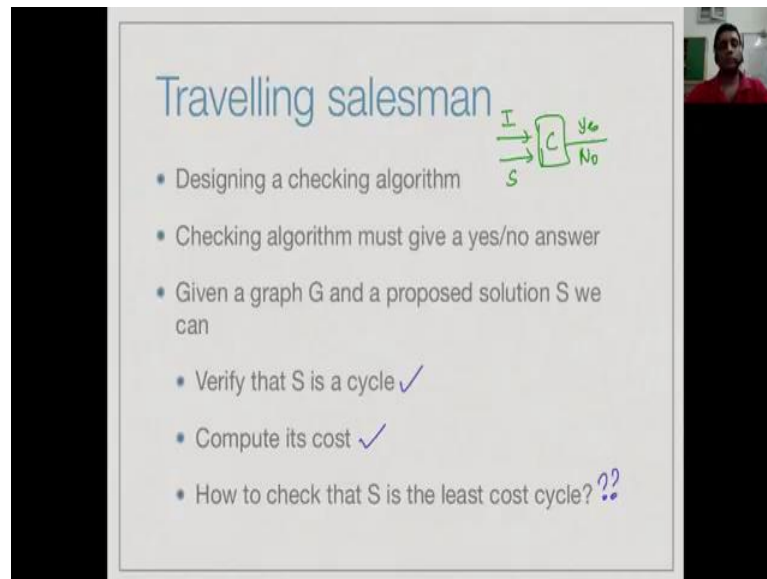
- A network of cities with distances between each pair
- A **complete** graph $G = (V, E)$ with edge weights
- Find the shortest tour that visits each city exactly once
- Simple cycle x, y, z, \dots, x visiting all vertices, of minimum cost

 There is a green handwritten note "no nodes repeat" with an arrow pointing to the sequence x, y, z, \dots, x . The first and last 'x' in the sequence are circled in green. In the top right corner, there is a small video inset showing a man with dark hair wearing a red shirt.

So, let us look at a completely different problem. So, we have this well known traveling salesman problem. So, salesman is supposed to visit a network of cities and between each pair of cities, we have a distance. So, we can think of it is a complete graph, every city connected to every other city and an each edge between two cities, there is a weight indicating the distance of a cause or the time or some quantity which the sales man has to use in order to travel from the one city to next city.

So, the salesman goal is to visit every city on this map. So, the salesman wants to find the shortest tour that visits each city exactly once. Then, a graph theorists sense what it means they were simple cycle, simple cycle means that no nodes repeat, I do not visit the same vertex twice along these things with starts and ends of the same city, that is why as a cycle a starts and ends of the same city, same vertex, visit every vertex in between and of minimum cost.

(Refer Slide Time: 12:19)



Travelling salesman

- Designing a checking algorithm
- Checking algorithm must give a yes/no answer
- Given a graph G and a proposed solution S we can
 - Verify that S is a cycle ✓
 - Compute its cost ✓
 - How to check that S is the least cost cycle? ??

So, one second, there is no simple way to write a generative an algorithm which will actually analyzes and find a good solution. So, now, our question is, is there a checking solution, is there a checking algorithms. So, recall that, what a checking algorithm does is a takes as input, it takes an input instance and it takes a solution, and then it says yes or no, this solution works this solution does not work.

So, now, we have a graph and somebody gives us a cycle, we can verify that is a cycle. So, that partices, we can even compute the cost of the cycle that is also easy, but how do we know that an among all the difference cycles, this is a least cost. So, it is not very clear that there is checking algorithm, because in the end, though we can verify part of this solution, that it is a cycle, that is this all the cities and we know it is cost, we have no way without solving the problem.

Remember the goal over checking algorithm is not to solve the problem, it is to ask whether this given solution is correct for this given instance. So, we may not know how to solve the problem, like the teacher is whose assign the factorization home work, the

teacher does not need know how to factorized, this not even need to know the factors, the teacher only needs to know, how to multiply to potential factors and check whether the answer is the same as the big number.

Similarly, here we just need to check, whether the given nodes form a cycle, but unless we know how to solve the problem, the checking algorithm cannot figure out whether is a least cost cycle or not.

(Refer Slide Time: 13:52)

Travelling salesman

- Transform the problem
- Is there a tour with cost at most K ? upper bound
- Now, given a solution S , we can check it
- For the original problem, cost is at most the sum of all the edge weights in the graph
- Find optimum K — test different values using binary search

So, how do we get a rounds? So, the solution is such optimization problems to convert them into checking algorithms is to transform the problem by giving a bound, so I given upper bound or a lower bound depending on out. So, in this case, we will ask will not just ask if there is a tour, if there is traveling sales men tour of lowest cost, we have says is there are tour with cost at most key. So, we have just given a bound on the cost, we are not asking for optimum tour, we just in only asking for a tour with does not cost more than K .

Now, we have given a solution, we can check it, because we can check it is a cycle, and then we can add up all the edges which form part of the tour and find out, whether it abject to K or S . So, so how to is a help us, because our goal was to find the shortest tour, what we have said now is that if I give you an upper bound size a tour, I can use checking algorithm for yes or no, but now we can try different case.

So, we have the minimum of value of K is clearly 0 and the maximum value is some upper bound. What is a good upper bound? Well, we know that a tour is going to take a

edges from the top, if I take all the edges in the graph and add up all the cost, the total tour cannot be more than that. So, I have a range of values is possible for the cost of tour from 0 to say the sum of all the edge weights in the graph. You might even find in better bound than that, because obviously, we cannot use all the edge weights, we can only use N of them to complete a cycle of size n .

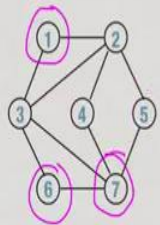
But, this is a very conserve bound, now what we do is, we do in this range, we do binary search. So, we first check is there a tour of whose cost is mid way, if there is then I will now check below is there a tour cost half of that. So, by doing a binary search, we narrow down the thing and finally, will find that this is the level, where we have a tour and about that, we have tours and below that, we do not have tours. So, this is the minimum cost tour.


So, by using a binary search and an upper bound, we can take an optimization problem we have looking for an optimal answer and transform it in to a sequence of checking problems, which after a logarithmic search through this space of the bounds will give me the actual solution or not.

(Refer Slide Time: 16:05)

Independent set

- u, v are independent if there is no edge (u, v)
- U is an independent set if each pair $\{u, v\}$ in U is independent
- Constitute a neutral committee where none of the members know each other
- Find the largest independent set in a given graph
- Checking version: Is there an independent set of size K ?



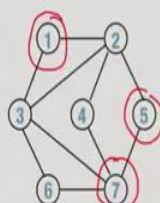


So, here is yet another problem. So, this is the called the independent set bar. So, we said that two vertices are independent; if they are not connected by it is an edge. So, for example, here you good look at say 1 and 7, then there independent together no connections. So, may be 6 and 5 are independent, so we say that two vertices are independent, if there is some edge.

(Refer Slide Time: 16:32)

Independent set

- u, v are independent if there is no edge (u, v)
- U is an independent set if each pair $\{u, v\}$ in U is independent
- Constitute a neutral committee where none of the members know each other
- Find the largest independent set in a given graph
- Checking version: Is there an independent set of size K ?

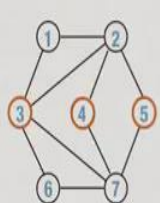


Now, if I take this set by take 1 and 7, this is an independent, but if I add 5 to it is for the instance, now 5 and 7 are connected by an edge, so these are not independent. So, an independent set is one in which every pair is independent. So, 1 and 7 is an independent set, but 1, 5, 7 is not an independent set, because 5, 7 is a neutral.

(Refer Slide Time: 16:55)

Independent set

- u, v are independent if there is no edge (u, v)
- U is an independent set if each pair $\{u, v\}$ in U is independent
- Constitute a neutral committee where none of the members know each other
- Find the largest independent set in a given graph
- Checking version: Is there an independent set of size K ?



So, here for instance, you can check the 3, 4, 5 forms in independent set, because is no in between 3 and 4, there is no edge in between 4 and 5, there is no edge in between 3 and 5. So, for example, if you are trying to say you interpret these nodes as people and you node edges as knowing each other. So, supposing in now you want to set up some kind of a committee, where you want to be usual that all the committee members have

independent opinions are not influence by the facts that they known's at that the before that.

Then you can pick up an independent set, an independent set produce people, who do not mutually do not know each other. So, therefore, when they meet for the first time, hopefully, if they have neutral opinions about each other and about the problem then there... So, this might be one example of why you want pickup an independence set. So, the algorithmic problem is to find than largest independence set.

So, here we have found in independence set of size 3, can I do better and I find one of size 4, maybe, maybe not. So, this my algorithmic problem, given a graph what is the largest independence set, that I can find in it. As we saw before, this is a problem where we have to get an answer. So, if somebody tells me 3, 4, 5 is a maximum independence set, I can verify easily the 3, 4, 5 reason independence set, but I cannot necessarily verify that, if there is no larger set.

This is like that traveling cells one problem, when I can verify that the tour to given to me is a simple cycle, but I may not be able to verify that the cost of the tour is the best among all such. So, once again, if you want to set up which checking version the problem, we will set up off, will say is there an independent set of size K . So, we are trying to find the largest one, so you will say at least size K . So, if I at least size 3 and this produce of this, I will verified at least size 3.

And at least size 4 and this is my witness and this is my solution, I will say no. So, independence set again like traveling salesmen and Boolean satisfies ability, we do not know of a good generative solution. But by using this bounded version were by the provider's number quantities, we are trying to estimate we can produces checking versions.

(Refer Slide Time: 19:04)

Vertex cover

- Node u covers every edge (u,v) incident on u
- U is a vertex cover if each edge in the graph is covered by some vertex in U
- Position surveillance cameras at intersections to watch all roads
- Find the smallest vertex cover in a given graph
- Checking version: Is there an vertex cover of size K ?

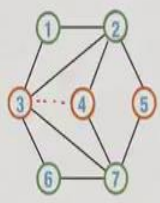
So, the related looking problem, relative problem to be this is called vertex cover, we say that a node you covers every edge that is instance. So, for example, if I take this node, then it covers these three, these four edges, because all these edges started. So, now, vertex cover is something that covers all the edges. So, for instance by take a 2 cover those edges, now these edges are not there, maybe I take 3 to cover it. So, I take 3 in this covers, these edges have a still have some missing edges, so maybe I pickup 7, which is 7 covers these three edges.

So, I have found a collection of 1, 2 3 vertices which cover all the edges in this star. So, now, what we want to do is find this smallest vertex cover in a given graph, an again because we do not know how it is smallest in general, we will say that, we will take any vertex cover which is at most says K , if the remove is smallest one. So, it is like that traveling salesmen, we will in the shortest tour, so it say is there a tour of less than or equal to said me cost, so is there a vertex cover less then equal to set in cost. So, these two problem look connected and in back they are. So, here for example, we have a vertex cover of size 4, which indicated in 3.

(Refer Slide Time: 20:40)

Connecting independent set and vertex cover

- U is an independent set of size K iff $V-U$ is a vertex cover of size $N-K$
- (\Rightarrow)
Every edge (u,v) has at most one end point in U , so at least one end point in $V-U$
- (\Leftarrow)
For any edge (u,v) , at least one endpoint in $V-U$, so no edges (u,v) within U



So, connection if that U is an independent sets of size K , if in only if it is compliments it vertex cover of size N minus K . So, in our case U 3, 4, 5 was a vertex independence of a size 3, when it is compliment must be vertex cover of size N minus K . So, if I could solve the vertex cover problem for the certain size, I can solve the independence set problem for N minus K . So, I can convert one to the other and the proof of this very easy.

So, supposing I know that using independent set, use in independent set, then no just with in U . So, by take any edge $U V$, then if one into inside U , the other in must outside U or both N points are outside. So, therefore, if I take any edge at least one of it is end points lives in V minus U . So, V minus U covers all the edges, so therefore, V minus U some vertices.

And of course, given that this as K vertices, the other must have N minus K vertices, conversely supposing I assume that V minus U is a vertex cover, then every edge starts from there. If every edge starts from there, one of it is ends points is there. So, I cannot have an end which is entirely within the complement, because if have an edge which is like this, then is a an edge which is not covered by vertex cover. So, they cannot be any edges with in U , therefore, U is an independence set.

And once again, the sizes are guaranteed, because their complementary search a vertices. So, this says that independence set reduces to vertices cover with the complementary size, this the bounded vertices version and vertex cover reduces to independence set. So, the reduce to each other.

(Refer Slide Time: 22:25)

The slide is titled "Reductions" in blue. It contains a bulleted list of points and a flowchart. Handwritten notes in green and blue ink are present.

- Independent set and vertex cover reduce to each other
- Recall: if A reduces to B and A is intractable, so is B
- Many pairs of checkable problems are inter-reducible
- All "equally" hard

Handwritten notes above the list:

- in known (green) above A
- not known (green) above B
- $A \rightarrow B$ (red arrow)
- eff (blue) below A
- ← (blue arrow)
- eff (blue) below B

Flowchart for "Algorithm for A":

```
graph LR
    x((x)) --> Preprocess[Preprocess]
    Preprocess -- y --> AlgorithmB[Algorithm for B]
    AlgorithmB -- B(y) --> Postprocess[Postprocess]
    Postprocess -- A(x) --> out(( ))
```

So, when we introduce reducibility in the context of linear programming and network flows we said that one aim is to transfer efficient solution from B to A. So, when I reduce A to B, if B is an efficient, then A is efficient. But in this context, what we are trying to say is that this is not known to be an efficient, then this also is not known to be efficient.

So, since independently we believe that independent it is not efficient then because reduces the vertex cover, vertex cover is also likely to be a not efficient and right sources. So, in terms of it that many pairs of checkable problems or such inter reducible problem, either directly are you may find a cycle or things A reduce to B, B reduce to C and C reduce backward.

So, in that sense, all of them are equally easy or equally hard and since, we tend to believe that nobody has solved any of them yet, if they must all be hard, this gives to the belief that there is a large group of problems which are actually practical in terms of their usefulness. But not known to be efficiently solved notice and in terms of them all notions of algorithmic efficiency. So, we will look at this a little more detail in our last lecture.