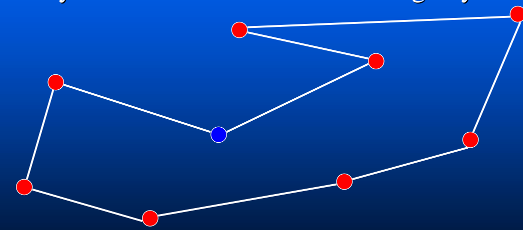
Topic 20

NP-Completeness
1. Polynomial time algorithm
2. Polynomial time reduction
3. P vs NP
4. NP-completeness

(some slides by P.T. Uma University of Texas at Dallas are used)

Traveling Salesperson Problem

- Find minimum length tour that visits each city once and returns to the starting city.



Given a Problem

- Polynomial time algorithm ☺
  – $O(n^k)$ (n is the input size; k is a constant)
- Super-polynomial time algorithm ☹ (some people call it non-polynomial)
  – $O(2^n)$, O(n!)

What's the Big Deal?

-  2.20 GHz (Jan 2002)
  (2.20 billion cycles per second)
- N = 30 cities N! = 30! = 265 * 10^{30} tours
- Super-fast machine: compute 1 TSP tour in 1 cycle.
- 2.20 billion TSP tours in 1 second.
- 120 * 10^{21} seconds
- 1 year = 31536000 s
- 38 * 10^{14} years!
- 3800 trillion years!!!
- Age of Earth = 4.6 billion years!

What's the Big Deal?

-  2.53 GHz (Aug 2002)
  (2.53 billion cycles per second)
- N = 30 cities N! = 30! = 265 * 10^{30} tours
- Super-fast machine: compute 1 TSP tour in 1 cycle.
- 2.53 billion TSP tours in 1 second.
- 104 * 10^{21} seconds
- 1 year = 31536000 s
- 32 * 10^{14} years!
- 3200 trillion years!!!
- Age of Earth = 4.6 billion years!

What's the Big Deal?

-  3.06 GHz (Jan 2003)
  (3.06 billion cycles per second)
- N = 30 cities N! = 30! = 265 * 10^{30} tours
- Super-fast machine: compute 1 TSP tour in 1 cycle.
- 3.06 billion TSP tours in 1 second.
- 86 * 10^{21} seconds
- 1 year = 31536000 s
- 27 * 10^{14} years!
- 2700 trillion years!!!
- Age of Earth = 4.6 billion years!

Progress in Technology

- N = 30 cities N! = 30! = 265 * 10^{30} tours
- Super-fast machine: compute 1 TSP tour in 1 cycle.
- Age of Earth (earth) = 4.6 billion years.
- Age of the Milky Way Galaxy (mwg) = 13 billion years.
- Age of the Universe (univ) = 15 billion years.

Jan 2002
3800 trillion years
826, 086 earth
292, 307 mwg
253, 333 univ

→

Aug 2002
3200 trillion years
695, 652 earth
246, 153 mwg
213, 333 univ

→

Jan 2003
2700 trillion years
586, 956 earth
207, 692 mwg
180, 000 univ

Given a Problem

- Tractable or intractable?
- Tractable – give a polynomial time algorithm.
- Intractable – show the problem is NP-complete and explore other means of solving the problem.

Given a Problem

- Tractable or intractable?
- Tractable – give a polynomial time algorithm.
- Intractable – show the problem is NP-complete and explore other means of solving the problem.

Given a Problem

- ***Give an efficient polynomial time algorithm.***
- 3 GHz ; 3 billion cycles/s ; 0.33 ns/cycle
- N = 1, 000, 000
- $O(n) = 330\ \mu s$
- $O(n^2) = 330\ s = 5.5$ minutes
- $O(n^3) = 330$ million $s = 10$ years

1. NP-Completeness

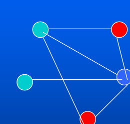
Before defining *P* and *NP,* let's understand the differences between problem that require to

1) give a YES or NO answer (***decision problem***)
2) find the cost of the optimal solution
3) find the optimal solution.

for example
1) Does the graph contain a spanning tree with weight at most 40?
2) What is the weight of the minimum spanning tree.
3) Find the minimum spanning tree

More Example: colouring

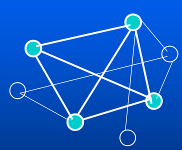
Given a graph *G=<V,E>*, we want to paint the vertices so that for any (*v,u*) in *E*, the colour of *v* is different from the colour of *u.*



1) **COLD:** On input *G* and an integer *k,* is *G* is *k*-colourable? (a special case when *k*=3, is know as the 3-colour problem, **3COL**).
2) **COLO:** On input *G* what is the minimum number of colours required to paint the graph?
3) **COLC:** On input *G,* find the way to paint the graph with minimum colours.

More Example: ***k*-clique**

Given a graph *G=<V,E>*, we want to find a subgraph of *G* that is a complete graph. A graph is complete if any two vertices are connected by an edge. A complete graph with *k* vertices is also known as a *k*-clique.



this graph contains a 4-clique

1) **CLQD:** On input *G* and an integer *k,* determine whether *G* contains a *k*-clique.
2) **CLQO:** On input *G* find the size of the largest clique.
3) **CLQC:** Find the largest clique in *G.*

1. Polynomial time algorithm

A algorithm is polynomial time if its worst-case running time is in $O(n^k)$ where n is the size of the input, and *k* is a constant independent of *n.*

For example,

quicksort is polynomial time $O(n^2)$.
mergesort is polynomial time O(n log n) which is also in $O(n^2)$.
Prim's algorithm is polynomial time O(|V| log |V| + |E|).
if we take the (n=|V|+|E|) as the size of the input, then Prim's algorithm is in O(n log n).

The following is not a polynomial time algorihtm.

1) input an n-bit integer M.
2) for i=1 to M; print i; end;

Note that the size of the problem is n.
The running time is $O(2^n)$, which is not a polynomial.

2. Polynomial time reduction

Suppose we have an algorithm, known as the *oracle*, that can determine whether a graph has a k-clique in O(1) worst case running time, can we find the k-clique easily?

In other words, if we can solve the decision problem, can we solve the other 2 forms of problem?

CLQO: On input *G* find the size of the largest clique.

To find the size of the largest clique, we can ask the oracle in the following way,

For i=n down to 1
If the graph contains a *i*-clique, return (i).
end

The worst case running time is O(n), which is a polynomial.
(the running time can be improved to O(log n)).

Suppose we can solve **CLQO** in O(1) time, can we solve **CLQC** efficiently?

3) **CLQC:** Find the largest clique in *G.*

1. Let T be the set of all vertices. Let G' be the graph G.
2. Ask the oracle the size of the largest clique in G. Let k be the size.
2. Select a vertex v from T. Remove v and all edges incident to v from G'.
3. Ask the oracle about G'. Let k' be the size of the largest clique in G'.
4. If k not equal k', then put v and all edges remove in step 2 back to G'.
5. Repeat step 2 to step 4 until T is empty.
6. Output G'.

The running time of the above is $O(n^2)$ where n is the number of vertices in *G.*

Definition:
Let A and B be two problems.
We say that A is polynomially Turing reducible to B if
there exists a polynomial time algorithm for solving A if we could solve B in O(1) time.

If A is polynomially Turing reducible to B, we write
$A \le B$ (or $B \ge A$)

If $A \le B$ and $B \le A$, we say that A and B are polynomially Turing equivalent, written as A = B.

If $A \le B$, we can view "B is more difficult or equal to A, because if we can solve B, then we can solve A".

Properties of reduction:
1) If $A \ge B$ and $B \ge C$, then $A \ge C$.
2) $A \ge A$.

Recall that **CLQD** ≥ **CLQO** and **CLQO** ≥ **CLQC**
Furthermore, it is clear that **CLQC** ≥ **CLQD.**

Thus we have **CLQD = CLQO = CLQC.**

So, the 3 problems are actually equivalent.

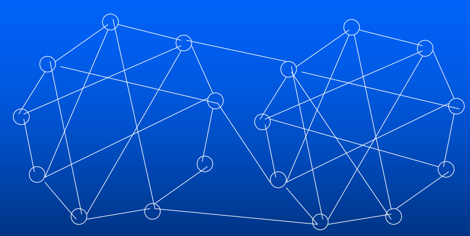
Tutorial:
Show that COLD = COLO = COLC.

Remark

This lecture note taking "short-cut" in defining polynomial Turing reducible. The notation used for *polynomial Turing reducible* is usually this : $\ge_T$,
which is to be distinguish from *polynomial many-to-one reducible*, usually denote as: $\ge_m$.

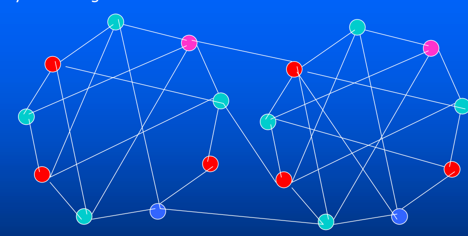
For polynomial many-to-one reducible, we can only call the oracle once. In polynomail Turing reducible, we can use it polynomial number of times.

If you tell me that this graph is 3-colourable,



it is very difficult for me to check whether you are right.

But if you tell me that this graph is 3-colorable and give me a solution, it is very easy for me to verify whether you are right.



Loosely speaking, problems that are difficult to compute, but easy to verify are known as Non-deterministic Polynomial.

P vs NP

Definition: P
P is the set of decision problems that can be solved by a polynomial time algorithm.

Recall that an algorithm is polynomial time if its worst-case running time is in $O(n^k)$ where n is the size of the input, and *k* is a constant independent of *n*.

We can represent a decision problem using a set, say K. An instance *x* is in K iff on input *x*, the output is YES.

For example, let K_1 be the problem where given an input, output YES if the input is already sorted in increasing order.
Then, K_1 is the set of sequences which are sorted,
$K_1 = \{\langle\rangle, \langle 1,2\rangle, \langle 1,3\rangle, \langle 2,3\rangle, \langle 1,2,3\rangle, \langle 1,4\rangle .... \}$

It is easy to write a linear time algorithm for K_1, thus, K_1 is in P (or we simply write $K_1 \in P$).

More examples:

1. Let K_2 be the set of binary sequence whose binary representation is dividisible by 3.
$K_2 = \{ 11, 110, 1001, 1100, 1111, ... \}$ $=\{3,6,9,12,15,...\}$
K_2 is in P.
(the length of the input "15" is 4, because $15 = 1111_2$)

2. Let K_3 be the set of binary sequence whose binary representation is a prime.
$K_3 = \{10,11,101, 111, 1011, 1101,...\}$ $=\{2,3,5,7,11,13,...\}$

For many hundreds of years, we don't have an algorithm that can solve K_3 in polynomial time, although people believe that there should be one. Recently, researchers from India find a polynomial time algorithm, i.e. they prove that $K_3 \in P$.

3. Let K_4 be the set of weighted graphs whose Minimum Spanning Tree have weight less than 30. Then $K_4 \in P$.

Definition: NP (non-deterministic polynomial)
A decision problem K is NP iff, there exists a Q∈P s.t.
x∈K if and only if there exists a y s.t. ⟨x,y⟩∈Q.

For example, let Q be the set of <x,y>, where x is dividisible by y , where (y>1) and (x>y). Here x and y are represented as binary sequences.
Q={ <100,10>,<110,10>,...... }
={ <4,2>,<6,2>,<6,3>,<8,2>,<8,4>,<9,3>,<10,2>,<10,5>,..}
Note that Q∈P.

Let K be the set of binary sequences, which represent a non-prime number that is greater than 1. (For many years, no one know whether K∈P. Recently, researchers from India prove that K∈P).
By the above definition, clearly, K∈NP.
This is because a number x is non-prime iff there exists a y>1 and x>y s.t. x is dividisible by y.

For eg., 135 is not a prime because <135,5>∈Q.
13 is a prime because there don't exists any y s.t. <13,y>∈Q.

The y in the definition is known as the *witness* or *certificate* or *proof* that x∈K.

The problem *Q* is known as the *proof system*.

So, non-deterministic polynomial are problems that have a proof system that can be solved in polynomial time.

In other words, non-deterministic polynomial are problems that can be easily verified in polynomial time.

We say that a decision problem is *decision-reducible* if, given an oracle that solves the decision problem in O(1) time, we can find the witness in polynomial time.

More examples of NP problems.

1. 3COL (3-colorability) is in NP.

2. CLQD (k-clique) is in NP.

3. Given a sequence of integer, $a_1,a_2,a_3,....,a_n$, and an integer *k*, can we group them into *k* groups s.t. the sum of each group is less than 50.

4. **Partition problem**. Given a sequence of integers, $a_1,a_2,a_3,....,a_n$, can we group them into 2 groups s.t. the sum of each group is the same.

5. **Packing**: Given a set of template for the *n* parts in a jean, and *k* pieces of standard sized cloth.
Can we cut them out from *k* pieces of standard sized cloth.

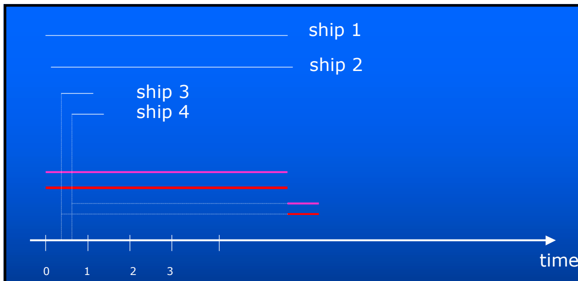
In the optimization version, we want to know how to cut them from minimum number of standard sized cloth.

can we pack these into 2 standard sized cloth?

6. Ship **parking**..
We have *n* ships, where each ship is represented as a circle with certain radius. (the radius depend on the size of the ship). Could we park these ships in the port? That is, pack the circles with no overlap?

7. **Scheduling**.

Suppose our harbor has 2 docking facilities A and B. Thus we can serve 2 ship concurrently.
Given a list of *n* ships, and the expected docking time and arrival time of each ship. Can we assign the ship to either A or B so that each ship will not wait for more than 3 hours, and the average waiting time is less than 1 hour?



8. Scheduling. Given the modules taken by *n* students, can we plan a two-week exam so that there is no crash, and each student will take at most two exams in any 3 consecutive days.

Hundreds of "NP-hard" problems are listed in
M.R.Garey, D.S. Johnson, *Computers and Intractability: a guide to the theory of NP-Completeness*, 1979, Freeman.

So, NP are important problems.

Theorem $P \subseteq NP$

This theorem states that any problem that can be solved in polynomial time, can also be verified in polynomial time.

(this is so obviously true....)
In proof, let K be a problem in P. Let us consider this problem Q which is defined as
$Q = \{ <x,0> \mid x \in K \}$.
Now, Q can be a proof system for K, and thus $K \in NP$.
Since for any $K \in P$, we have $K \in NP$, therefore $P \subseteq NP$.

Now, the million dollars open problem is,

is **$NP \subseteq P$?**

if this is true (i.e, NP=P) , then any problem in NP can be solved efficiently.
A lot of researchers have worked on some NP problems but get no progress.

So, there might be some problems in NP that cannot be solved efficiently. (i.e. $NP \neq P$).

Unfortunately, we still don't know the answer. Most people strongly believe that $NP \neq P$.

4. NP-complete

Definition: NP-hard
A decision problem K is NP-hard if
1) $K \geq Y$ for every $Y \in NP$.

Definition: NP-complete
A decision problem K is NP-complete if
1) $K \in NP$, and
2) K is NP-hard.

The first definition can be viewed as: K is more difficult or equal to any problem in NP.

Note that a NP-complete problem K is the "ticket" to all NP problems. If we can solve K in polynomial time, then we can solve **ALL** NP problems in polynomial time, and thus NP=P.

Conversely, if indeed $NP \neq P$, then a NP-complete problem can not be solved in polynomial time.

Now the question is to find these NP-complete problems.

Theorem

If $K \in NP$, and $K \geq Y$ where Y is NP-complete. Then K is NP-complete.

Another NP problem **SAT-3-CNF**

definition: A *literal* is a Boolean or its negation or 1 or 0.
A 3-*clause* is a disjunction ("or") of 3 literals.
A 3-*CNF* of is a conjunction ("and") of 3-clauses.

e.g. $A = (\overline{a}+\overline{b}+c)(a+\overline{c}+d)(a+0+0)$

$B = (\overline{a}+\overline{b}+c)(a+b+0)(c+0+0)$

The input is a 3-CNF with *n* variables. Is there a way to assign 0/1 (TRUE/FALSE) to the variables so that the formula is 1 (TRUE).

in the above eg. by assigning a=1, b=0, c=0, d=0, then A is 1. Equation B is always 0.

Theorem
SAT-3-CNF is NP-complete

Cook shows that SAT-3-CNF is NP-complete (actually, he shows that another problem SAT-CNF is NP-complete, and it is not difficult to show that SAT-3-CNF > SAT-CNF). for details, see text.

So, we have a NP-Complete problem. Starting from here, researchers found that in fact most interesting problems (include the packing, scheduling problems) are NP-complete. This is done using the theorem in the previous slide.

In this lecture, we will describe one reduction.

Lemma
3COL > SAT-3-CNF.

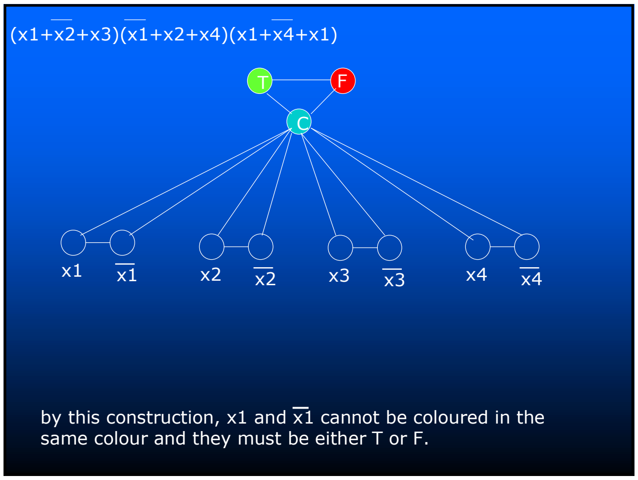
Proof (3COL>SAT-3-CNF)

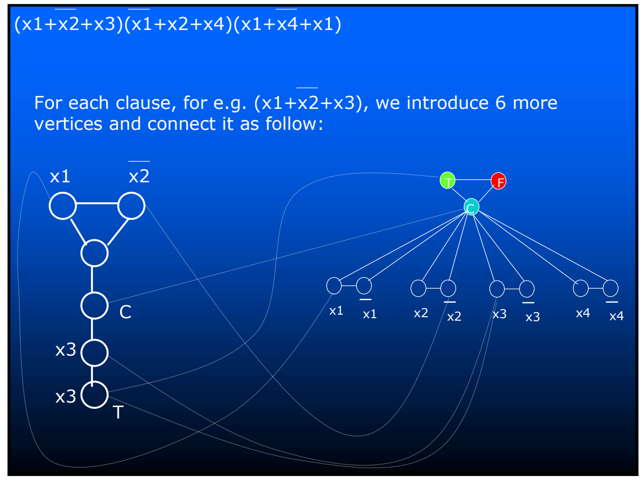
Given an input x of SAT-3-CNF, we want to transform it into the input y of the 3COL. The transformation is done in a way that $x \in$ SAT-3-CNF if and only if $y \in$ 3COL

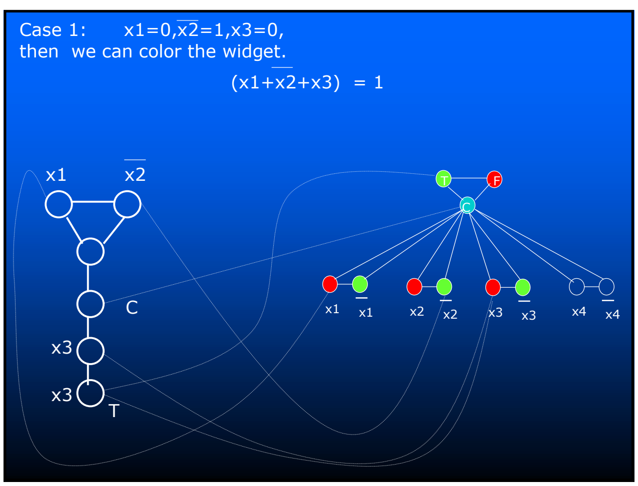
Given the an input of SAT-3-CNF. Let k to be the number of clauses, number of variables is t. We want to build a graph G with 3+2t+k vertices and 3+3t+12 k edges.

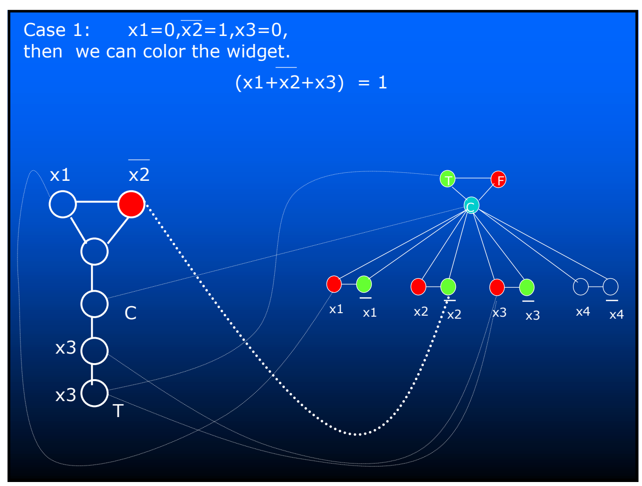
e.g.
$(x1+\overline{x2}+x3)(\overline{x1}+x2+x4)(x1+\overline{x4}+x1)$

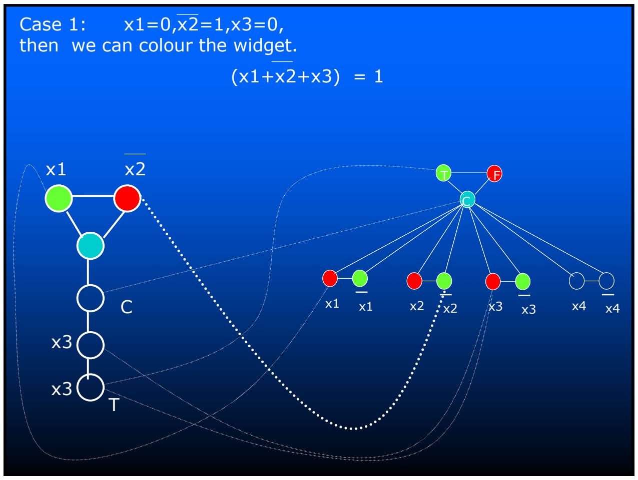
For the variables, we build this:

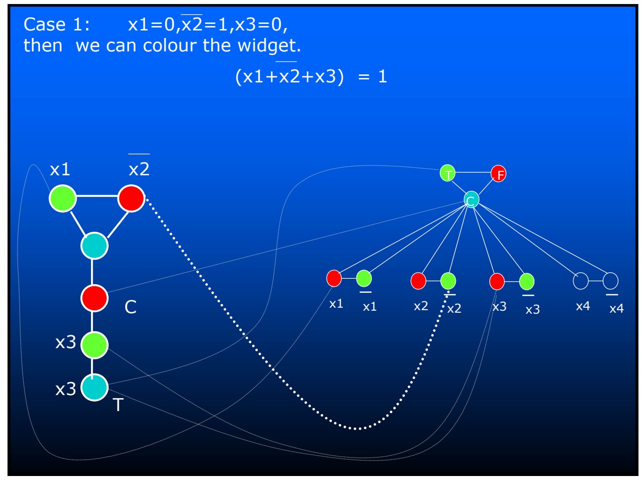

(x1+x2+x3)(x1+x2+x4)(x1+x4+x1)
T
F
C
x1
x1
x2
x2
x3
x3
x4
x4
by this construction, x1 and x1 cannot be coloured in the same colour and they must be either T or F.

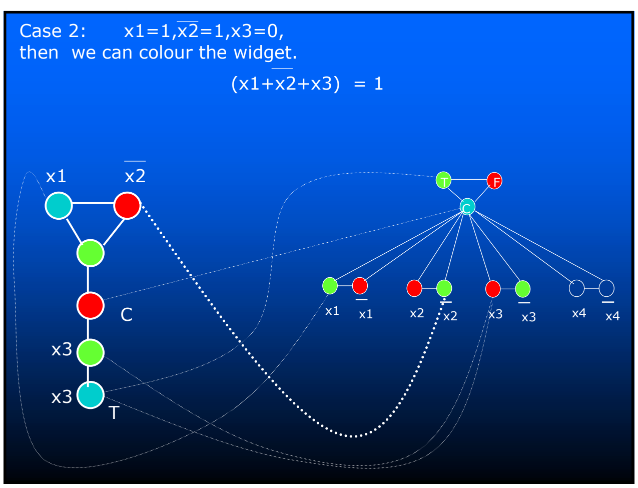

(x1+x2+x3)(x1+x2+x4)(x1+x4+x1)
For each clause, for e.g. (x1+x2+x3), we introduce 6 more vertices and connect it as follow:
x1
x2
C
x3
x3
T

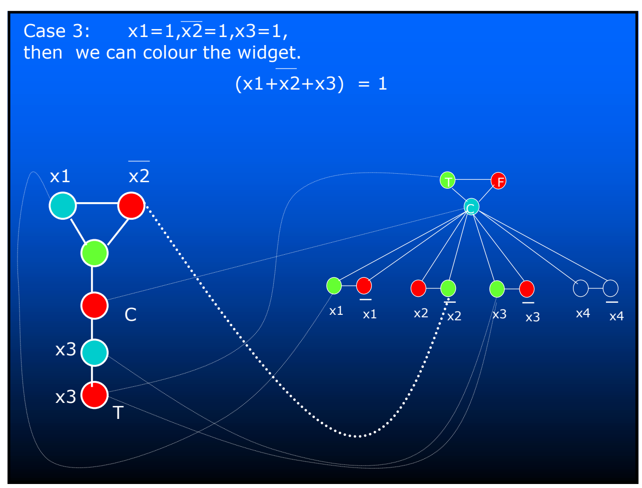

Case 1: x1=0,x2=1,x3=0,
then we can color the widget.
(x1+x2+x3) = 1
x1
x2
C
x3
x3
T

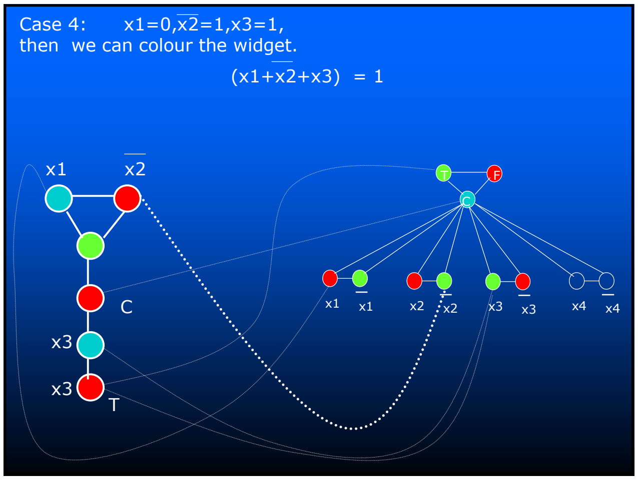

Case 1: x1=0,x2=1,x3=0,
then we can color the widget.
(x1+x2+x3) = 1
x1
x2
C
x3
x3
T

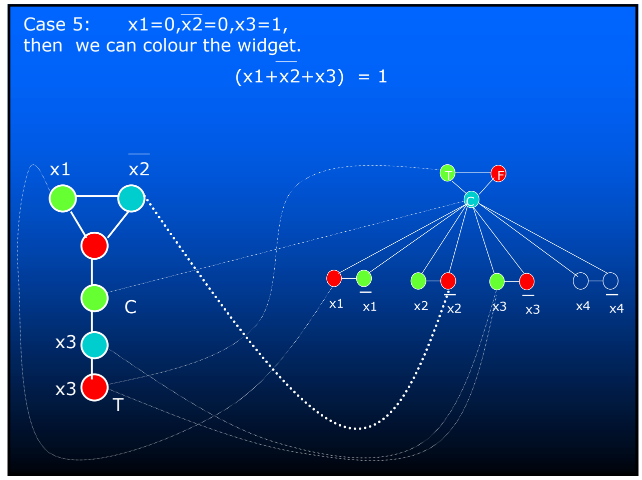

Case 1: x1=0,$\overline{x2}$=1,x3=0,
then we can colour the widget.
(x1+$\overline{x2}$+x3) = 1
x1
$\overline{x2}$
C
x3
x3
T
T
F
C
x1
$\overline{x1}$
x2
$\overline{x2}$
x3
$\overline{x3}$
x4
$\overline{x4}$

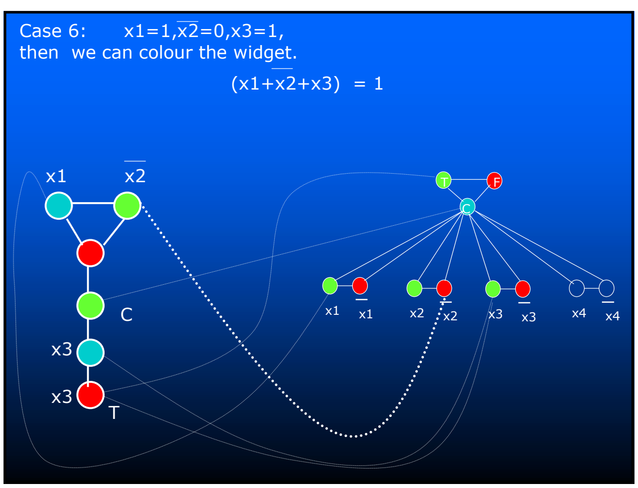

Case 1: x1=0,$\overline{x2}$=1,x3=0,
then we can colour the widget.
(x1+$\overline{x2}$+x3) = 1
x1
$\overline{x2}$
C
x3
x3
T
T
F
C
x1
$\overline{x1}$
x2
$\overline{x2}$
x3
$\overline{x3}$
x4
$\overline{x4}$

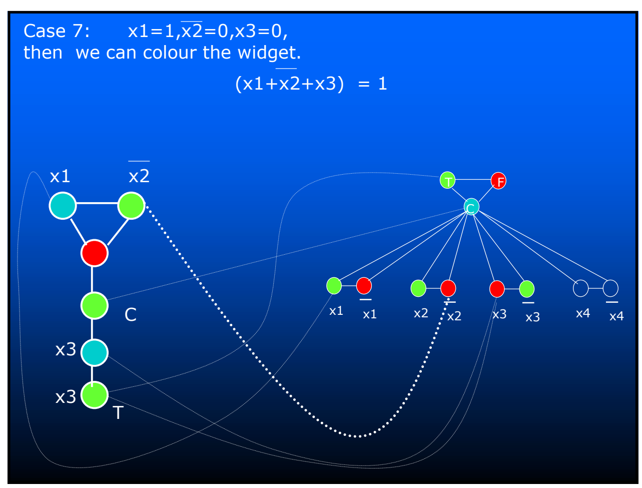

Case 2: x1=1,$\overline{x2}$=1,x3=0,
then we can colour the widget.
(x1+$\overline{x2}$+x3) = 1
x1
$\overline{x2}$
C
x3
x3
T
T
F
C
x1
$\overline{x1}$
x2
$\overline{x2}$
x3
$\overline{x3}$
x4
$\overline{x4}$


Case 3: x1=1,$\overline{x2}$=1,x3=1,
then we can colour the widget.
(x1+$\overline{x2}$+x3) = 1
x1
$\overline{x2}$
C
x3
x3
T
T
F
C
x1
$\overline{x1}$
x2
$\overline{x2}$
x3
$\overline{x3}$
x4
$\overline{x4}$


Case 4: x1=0,$\overline{x2}$=1,x3=1,
then we can colour the widget.
(x1+$\overline{x2}$+x3) = 1
x1
$\overline{x2}$
C
x3
x3
T

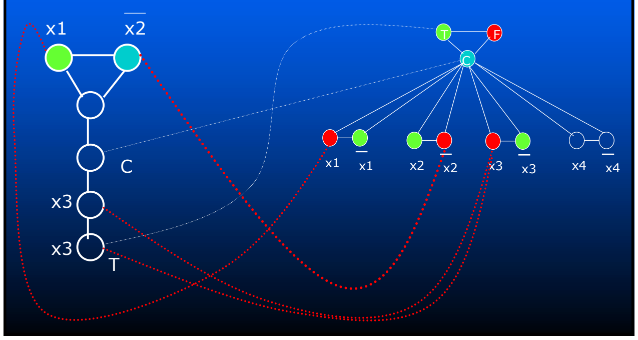

Case 5: x1=0,$\overline{x2}$=0,x3=1,
then we can colour the widget.
(x1+$\overline{x2}$+x3) = 1
x1
$\overline{x2}$
C
x3
x3
T


Case 6: x1=1,$\overline{x2}$=0,x3=1,
then we can colour the widget.
(x1+$\overline{x2}$+x3) = 1
x1
$\overline{x2}$
C
x3
x3
T


Case 7: x1=1,$\overline{x2}$=0,x3=0,
then we can colour the widget.
(x1+$\overline{x2}$+x3) = 1
x1
$\overline{x2}$
C
x3
x3
T

Case 8: $x1=0, \overline{x2}=0, x3=0$,
then we can colour the widget.

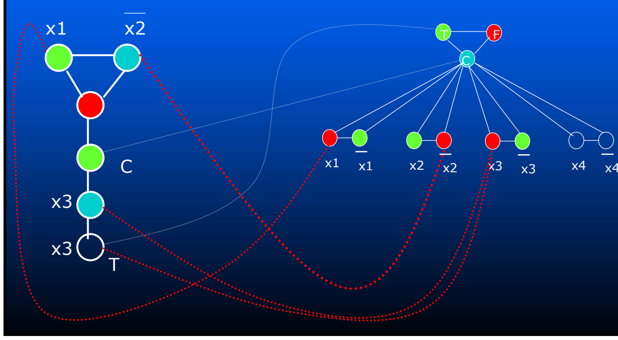
$(x1+\overline{x2}+x3) = 0$



Case 8: $x1=0, \overline{x2}=0, x3=0$,
then we can colour the widget.

$(x1+\overline{x2}+x3) = 0$

CAN NOT BE COLOURED



To summarize, if one of the literals is assigned as T (thus the clause is true), then we can 3-coloured the widget. Otherwise, we cannot 3-coloured the widget.

The Boolean equation is true iff all the clauses are true. Thus, the Boolean equation is true iff we can 3-coloured all the widgets.

Thus, 3COL $\geq$ SAT-3-CNF.

What can we do if a problem is NP-hard

1. Fast algorithm that find the solution for small input.
2. Algorithm that find approximate solution.
3. Algorithm that find solution for special type of instances.

Approximation Solution/Algorithm

Let OPT be the cost of the optimal solution.
If we can find a solution with cost APR, such that

$APR < \varepsilon\, OPT$, where ε is a constant greater than 1.

then we say that the solution is a ε-approximation solution, and the algorithm that find the approximation solution is called the ε-approximation algorithm.

Examples (approximation algo)

Traveling Salesman problem.

Input: A complete undirected graph G=<V,E> that has nonnegative cost c(u,v) associated to each edge (u,v).

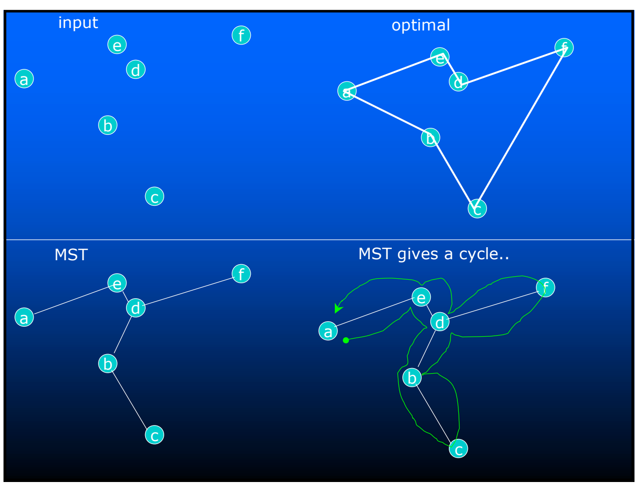
Output: A cycle (that is, a path that start and end at the some vertex) that visits each vertex once, with minimum cost.

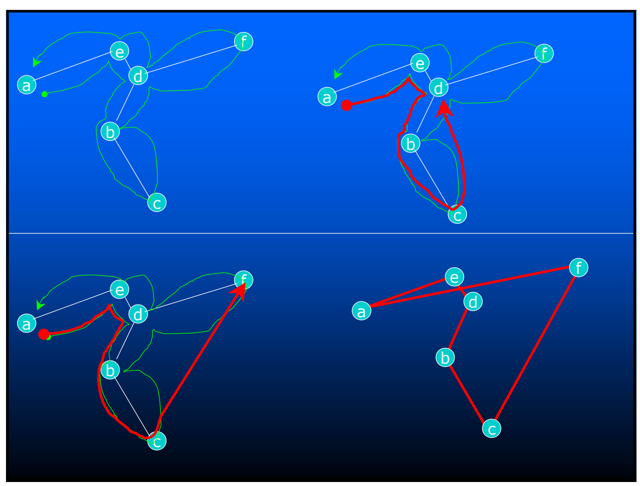
This is a NP-hard problem. Let us now look at a special case where the graph is an **Euclidean Graph**, and give a 2-approximation algorithm.

Traveling salesman on Euclidean Graph (triangluar inequality)

Algorithm Approx-TSP

1. Find the minimum spanning tree.
   (Note that the MST can be converted into a cycle.)
2. Randomly select a vertex and designate it as the root.
3. Do a preorder traversal of the tree.
4. Return the cycle that visits the vertices in the order computed in step 3.





Claim: Approx-TSP is a 2-approximation algorithm.

Let H* be the optimal cycle, and let T be the MST.
By removing any edge from H*, it become a spanning tree. Thus

cost (T) ≤ cost (H*).

The cycle obtained from T in step 1 traverses every edges in T twice.
Let W be this cycle. Clearly
cost (W) = 2 cost (T).

Note that W is not a solution, because vertices are visited twice.
Now, just remove the repeating vertices. If W visits the vertices in this order..

...... v_1, v_2, v_3, ...

By removing v_2, we will visit v_3 after visiting v1,. That is, the edge from (v_1,v_2) and (v_2,v_3) will be replace by the edge (v_1,v_3).
By triangular inequality,

the length of (v_1,v_3) ≤ lenght of (v_1,v_2) + lenght of (v_2,v_3) .

Let H be the cycle obtained by removing all repeating vertices.
We have cost (H) ≤ cost (W) = 2 cost (T) < 2 cost (H*).
Thus cost (H) ≤ 2 cost (H*)

Remark on Approximation algorithm

- Note, however, that there are problem that does not have approximation algorihtm (unless P=NP).

For example, we can prove that, unless P=NP, TSP on general graph does not have an c-approximation algorithm, where c is a constant.

List of problems described in this course:

Decision problem.

- **COLD**
- **CLQD**
- **3COL**
- **Partition Problem**
- **SAT-3-CNF**
- **SAT-CNF**

Finding the optimal cost

- **COLO**
- **CLQO**

Finding the optimal solution

- **COLC**
- **CLQC**
- **TSP**