Unit-5

NP-COMPLETE & NP-HARD

Polynomial

Linear search- O(n)

Binary search- O(log n)

O(1)

Bubble Sorting- O(n^2)

Merge sort- O(nlogn)

Matric chain multiplication- O(n^3)

Deterministic Algorithms

Non-polynomial (Exponential time algorithms)

0/1 knapsack problem – 2^n

TSP- n^2 2^n

Graph coloring – 2^n

Hamiltonian cycles- 2^n

Non-deterministic polynomial time algorithms

```
A,n,key
For i=1 to n
If (a[i]==key)
Successful
If (i==n+1)
unsuccessful
```

Choice(S)- randomly choose one of the elements from S – O(1)

Failure()- signals unsuccessful completion– O(1)

Success()- signal unsuccessful completion– O(1)

```
Algorithm Nsearch(A,n,key)
{
j=Choice() //
If (key == A[j]);
{
  Write(j);
 Success();
}
Write(0);
Failure();
}
```

Non-deterministic Sorting

```
B ← 0
/* guessing */
 for i = 1 to n do
j ← choice(1 : n)
 if B[j] ≠ 0 then
failure B[j] = A[i]
/* checking */
 for i = 1 to n-1 do
 if B[i] > B[i+1] then failure
success
```

3, 4 , 2, 1

i=1, 2

j=Choice()

b[j]NE 0

b[j]=a[i]

b[3]=3

b[4]=4

b[2]=2

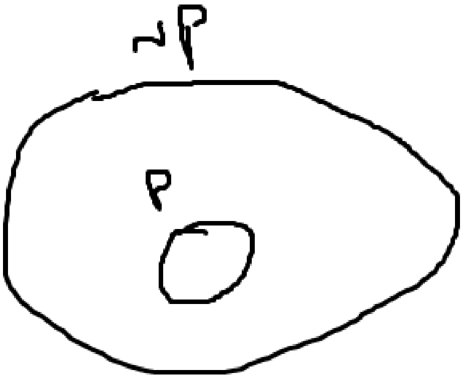
b[1]=1

P: set of problems for which deterministic polynomial time algorithms are available

Eg: linear search, sorting

NP: set of problems for which non-deterministic polynomial time algorithms are available

Eg: Satisfiability problem



Framework or guidelines provided to solve exponential time algorithms in polynomial time NDA is called NP-Hard and NP-Complete.

NP-Hard: set of hard problems with exponential time complexity

NP-Complete: NP-Hard problems for which non-deterministic polynomial time algorithm is available.

To solve Exponential time algorithms, we follow a method to relate the problems.

This is done by reduction (α) operation

If we already know an exponential time problem for which non deterministic polynomial time algorithm is available, then we can prove that other exponential time problems reduce to the known problem.

For this we need a base problem with exponential time complexity and for which already non deterministic algorithm is designed

Example: CNF Satisfiability problem

SAT is a CNF with n Boolean variables. We need to find the assign truth values to xi’s , so that CNF is true.

```
Algorithm EVAL (E, n)
{
for i=1 to n do     O(n)
xi=choice (true, false) O(1)
if E(x1,x2,…..xn) then Success();
else Failure();
}
```

SAT- O(n)

CNF-SAT

3   3

$x_1\ x_2\ x_3 \cdots x_n$ 2

n

$(x_1 \vee \bar{x}_2) \wedge (x_1 \vee \bar{x}_3)$

$\wedge (x_2 \vee x_3) \Rightarrow T$

$\boxed{2^n} \rightarrow$ ETA

1. NP – 1

2. NDA

x_1	x_2	x_3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

← L_1 — NP-C

SAT α L_1 L_2 — ??

SAT α L_1

L_1 α L_2

L_1 α L_2

∴ L_2 NP-C

I_1 I_2

polynomial

Reduction

L1 NP-Complete

L2 ???

L1 α L2

Steps to prove a problem is NP-Complete

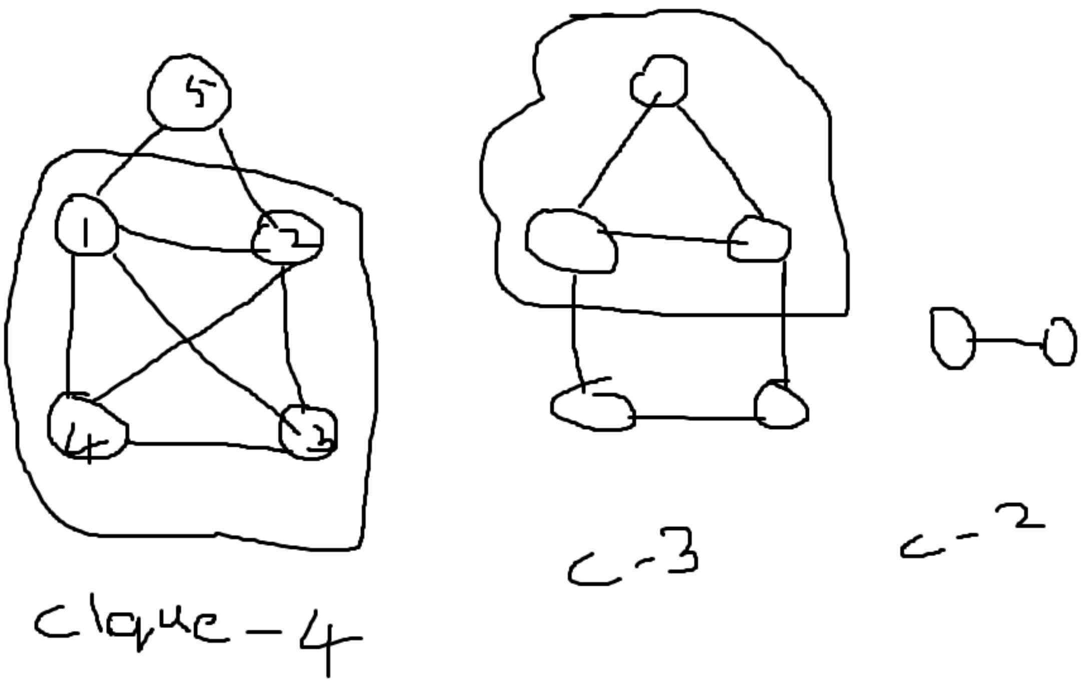
1. Pick a problem L1 already known to be NP-Complete
2. Show how to obtain an instance I2 of L2 from any instance I1 of L1
3. Conclude from step 2 that L1 α L2
4. Conclude from step 1 and 3 and transitivity of α that L2 is NP-Complete

1. Clique Decision Problem

Clique is a complete subgraph of a graph G= (V,E). CDP is to find weather a clique of size K exists or not in a graph G.

n-vertices comple graph no.of edges= n (n-1) /2

Eg:



Proof: We know that SAT is NP-Complete problem, to prove CDP as NP-Complete, we need to transform an instance of SAT into an instance of CDP in polynomial time. i.e.

CNF-SAT α CDP

I1   I2

Consider a CNF formula F given below

$$F = (x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2) \wedge (x_1 \vee x_3)$$

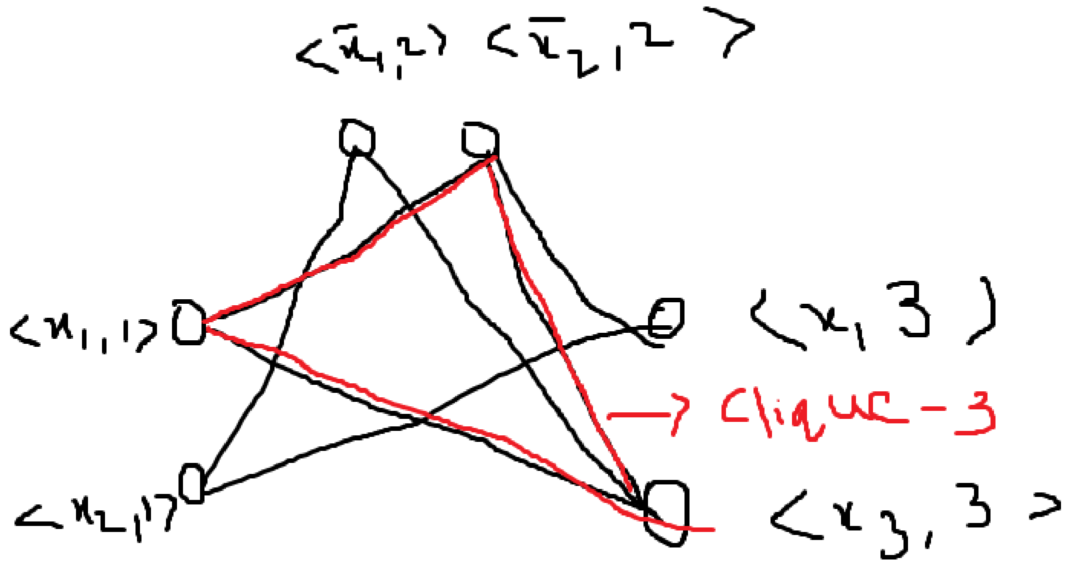
$$F = \bigwedge_{i=1}^{k} C_i$$

Prepare a graph G from the formula F and show that the graph contains a clique of size K.

In G = (V,E), the vertices and edges are defined by using the following equations.

V= { <a,i> | a €Ci }

E = { <a,i> <b,j> | i ≠j and b ≠ a }



Graph G for the CNF Formula F with three clauses has a Clique of size 3. i.e. an instance of CNF-SAT is transformed to an instance of CDP and we already know that SAT is NP-Complete. Hence, it is proved that CDP is also NP-Complete.

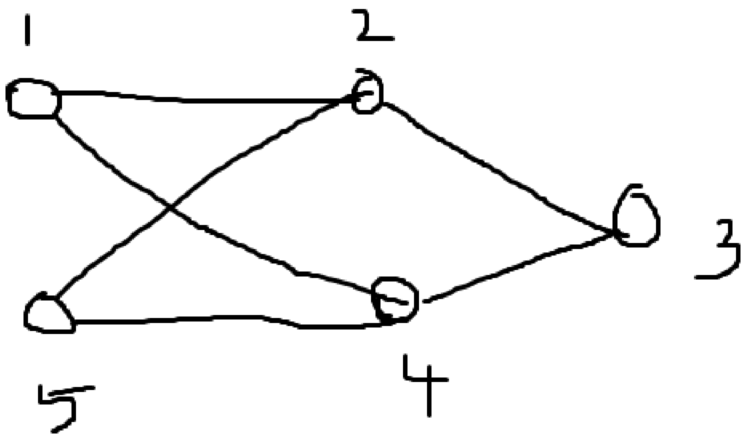
To find the truth assignments for which SAT is true is obtained from the graph G as given below.

x1	x2'	x3
1	0	1

This is one assignment which is a clique in the transformed graph G for which SAT is true.

2. Node Cover Decision Problem (NCDP)

A set $S \subseteq V$ is a node cover for a graph G= (V,E) iff all edges in E are incident on at least one vertex in S. The size |S| is the number of vertices in S.



Eg:

Node cover S = { 2, 4 } , |S| =2

S = { 1,3,5 } , |S| =3

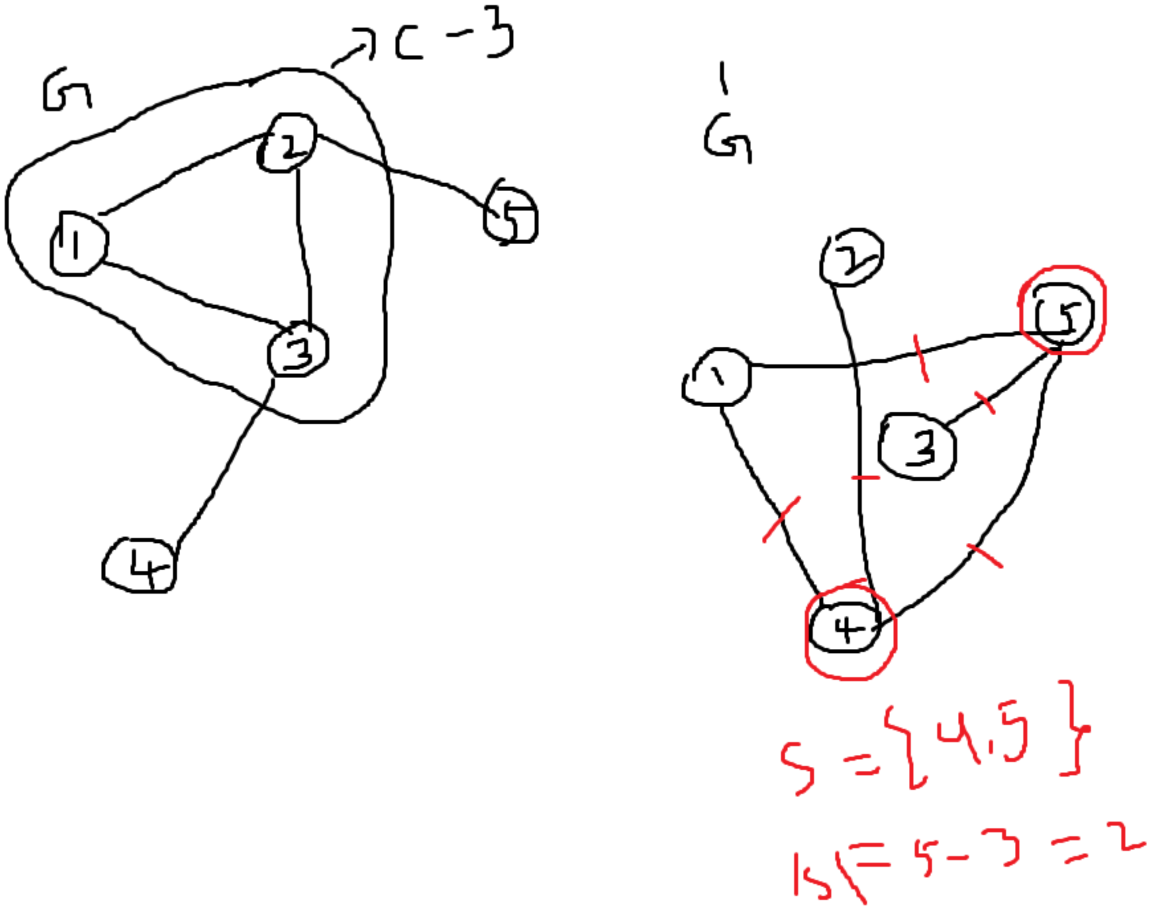
Problem: NCDP problem is to determine that whether a graph has a node cover of size at most K.

Proof: If any existing NP-Complete problem is reduced to NCDP problem then we can say that NCDP is also NP-Complete.

We consider CDP which is already proven as NP-Complete to prove NCDP as NP-Complete.

Let G=(V,E) and graph has a Clique of size K iff the complement of G will have a node cover of size at most (n-k).

Construct a graph G^1 such that G^1 has a node cover of size (n-k) iff G has a Clique of size at most K.



Hence proved that NCDP is also NP-Complete problem.

3. Hamiltonian Cycle decision problem (HCDP)

Hamiltonian Cycle: A cycle in an undirected graph G =(V, E) which traverses every vertex exactly once.

Problem Statement: Given a graph G(V, E), the problem is to determine if the graph contains a Hamiltonian cycle consisting of all the vertices belonging to V.

Proof: . First, HamCycle ∈ NP.

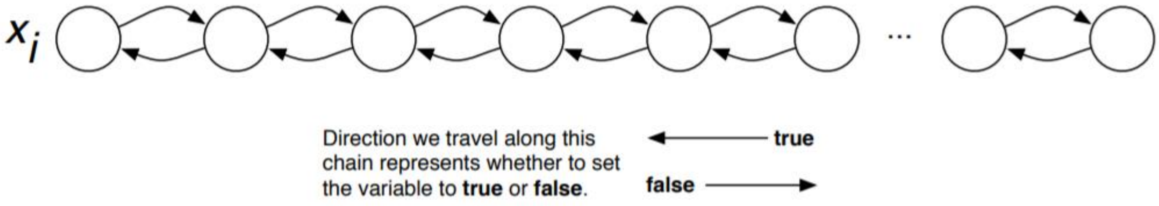
Second, we show 3-SAT ≤P Hamiltonian Cycle

In other words: how do we encode an instance I of 3-SAT as a graph G such that I is satisfiable exactly when G has a Hamiltonian cycle.

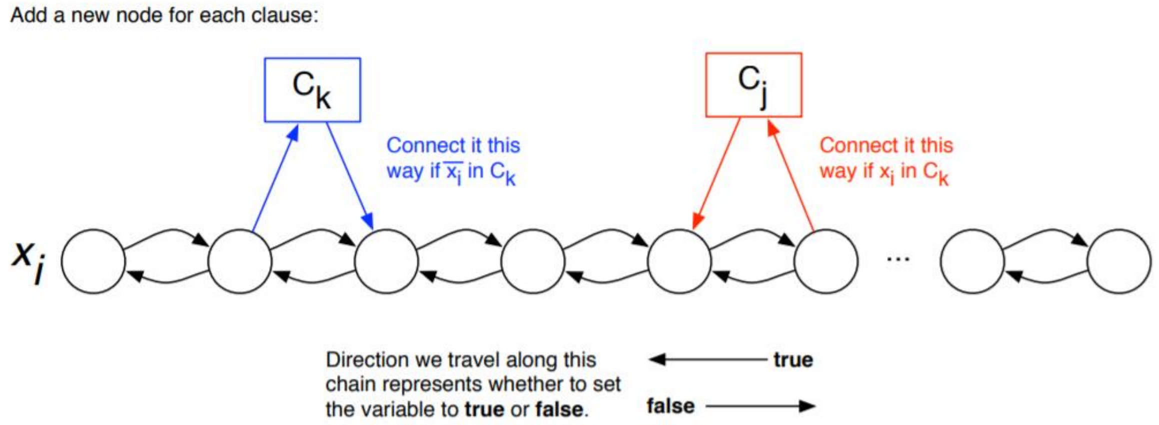
Consider an instance I of 3-SAT, with variables x1, . . . , xn and clauses C1, . . . , Ck .

- Create some graph structure (a “gadget”) that represents the variables
- And some graph structure that represents the clauses
- Hook them up in some way that encodes the formula
- Show that this graph has a Ham. cycle iff the formula is satisfiable.

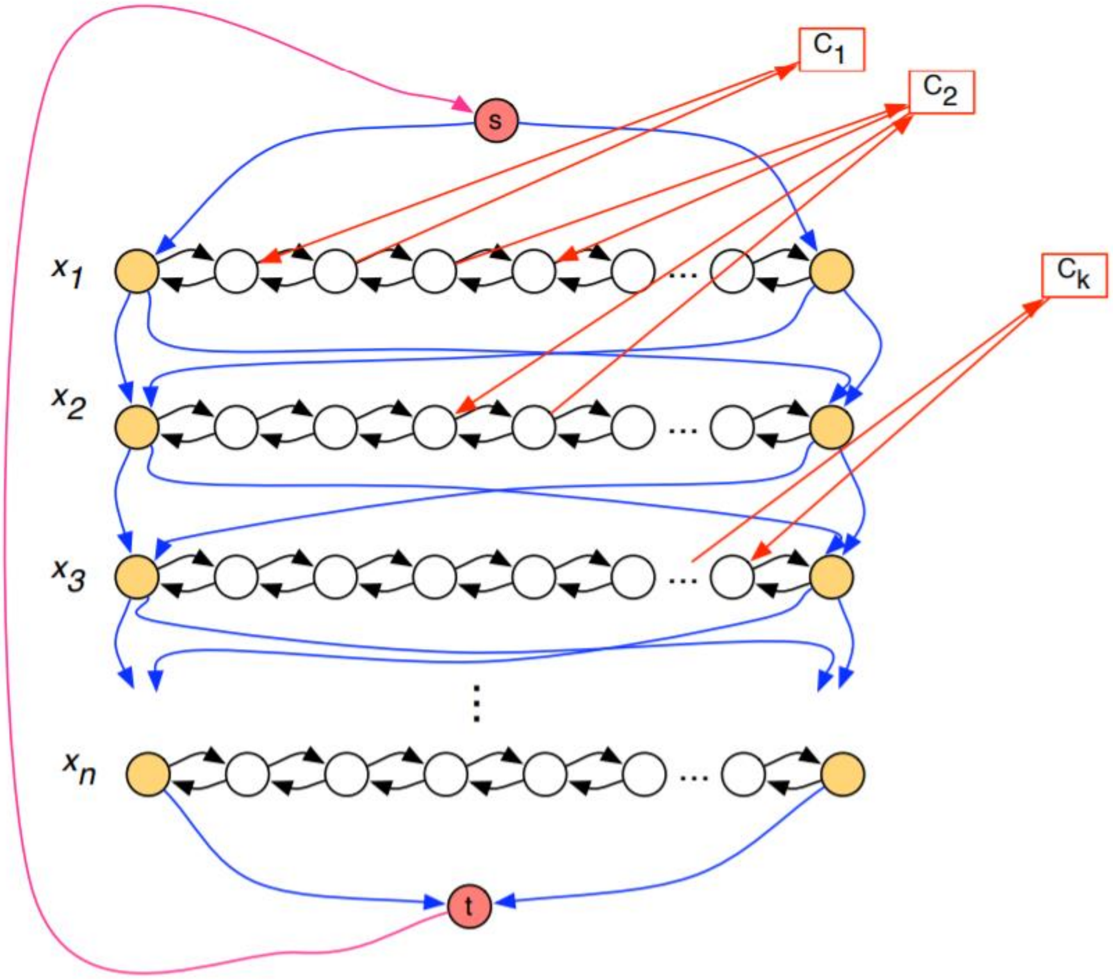
Gadget representing variables:


x_i
...
Direction we travel along this chain represents whether to set the variable to **true** or **false**.
true
false

Hooking the clauses:


Add a new node for each clause:
C_k
C_j
Connect it this way if $\overline{x_i}$ in C_k
Connect it this way if x_i in C_k
x_i
...
Direction we travel along this chain represents whether to set the variable to **true** or **false**.
true
false

Connecting the paths:

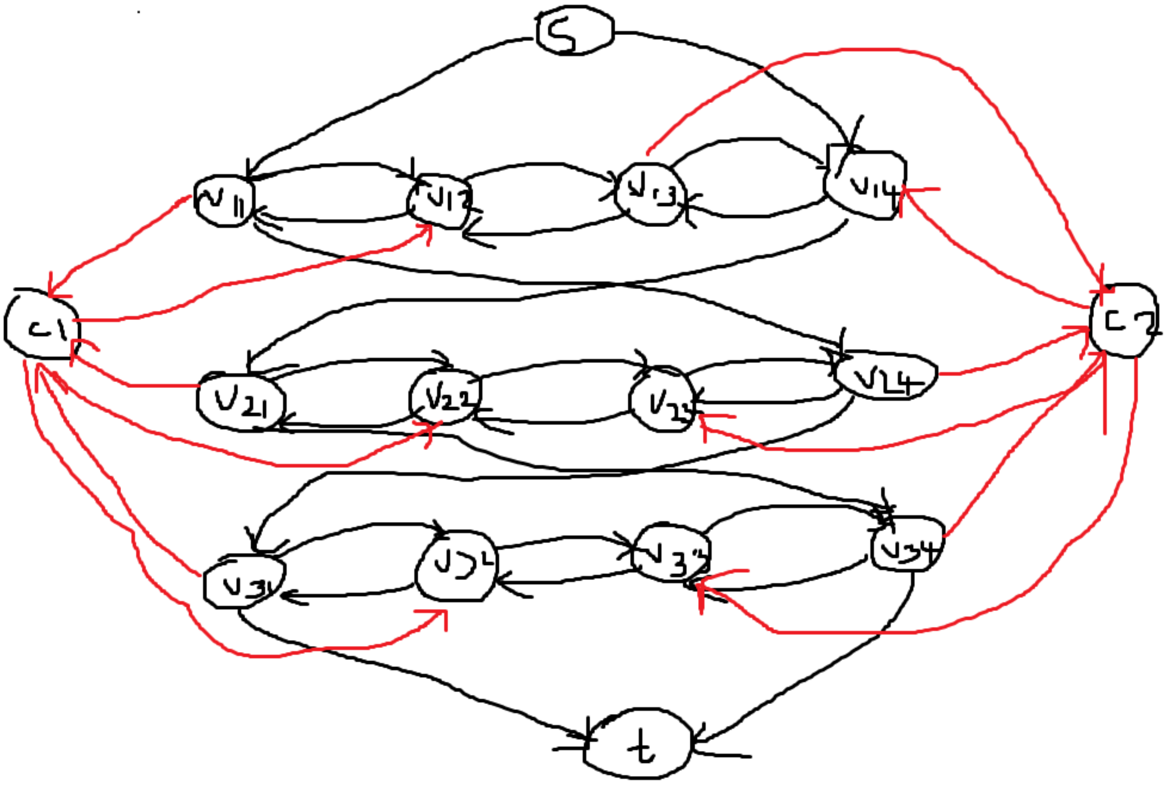


Hamiltonian is NP-Complete

- A Hamiltonian path encodes a truth assignment for the variables (depending on which direction each chain is traversed)
- For there to be a Hamiltonian cycle, we have to visit every clause node
- We can only visit a clause if we satisfy it (by setting one of its terms to true)
- Hence, if there is a Hamiltonian cycle, there is a satisfying assignment

Eg:

$$F = (x \vee y \vee z) \wedge (x \vee \bar{y} \vee \bar{z})$$



Find the Hamiltonian Cycle that is present in the graph shown above by traversing all vertices and clauses. Hence, proved that
3-CNF SAT α HAM

4. TSP is NP-Complete

The traveling salesman problem consists of a salesman and a set of cities. The salesman must visit each one of the cities starting from a certain one and returning to the same city. The challenge of the problem is that the traveling salesman wants to minimize the total length of the trip

Proof:

To prove ***TSP is NP-Complete***, first we must prove that ***TSP belongs to NP***. In TSP, we find a tour and check that the tour contains each vertex once. Then the total cost of the edges of the tour is calculated. Finally, we check if the cost is minimum. This can be completed in polynomial time. Thus ***TSP belongs to NP***.

Secondly, we have to prove that ***TSP is NP-hard***. To prove this, one way is to show that ***Hamiltonian cycle*** $\leq_p$ ***TSP*** (as we know that the Hamiltonian cycle problem is NP-complete).

Assume ***G = (V, E)*** to be an instance of Hamiltonian cycle.

Hence, an instance of TSP is constructed. We create the complete graph ***G′ = (V, E′)***, where

$$E′=\{(i,j): i,j∈V \text{ and } i≠j$$

Thus, the cost function is defined as follows −

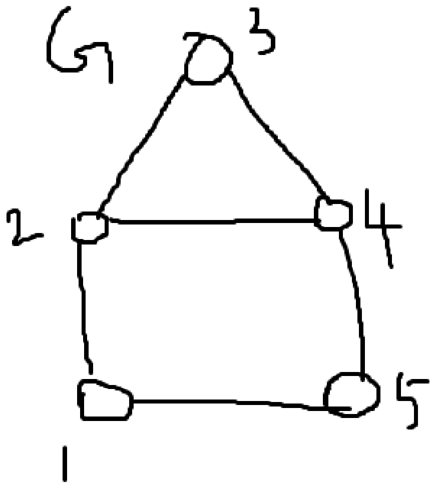
$$t(i,j)=\{0 \ \text{ if } (i,j)∈E$$

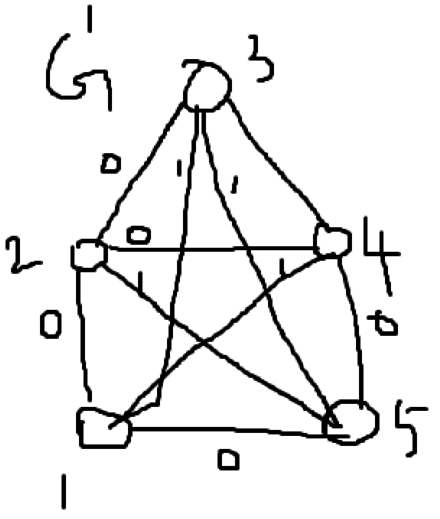
$$1 \text{ otherwise}\}$$

Now, suppose that a Hamiltonian cycle ***h*** exists in ***G***. It is clear that the cost of each edge in ***h*** is **0** in ***G′*** as each edge belongs to ***E***. Therefore, ***h*** has a cost of **0** in ***G′***. Thus, if graph ***G*** has a Hamiltonian cycle, then graph ***G′*** has a tour of **0** cost.

Conversely, we assume that ***G′*** has a tour ***h′*** of cost at most **0**. The cost of edges in ***E′*** are **0** and **1** by definition. Hence, each edge must have a cost of **0** as the cost of ***h′*** is **0**. We therefore conclude that ***h′*** contains only edges in ***E***.

We have thus proven that ***G*** has a Hamiltonian cycle, if and only if ***G′*** has a tour of cost at most **0**. TSP is NP-complete.





G^1 has a tour of sum 0, hence proved that TSP is a NP-Complete problem.

5. Chromatic Number Decision Problem (CNDP)

A Graph G= (V, E) is K-vertex colourable iff the graph G is not (k-1) colourable. Then K is called the Chromatic number of G.

To prove CNDP is an NP-Complete problem, we use 3-CNF-SAT (SATY) which is known as NP-Complete and show that

SATY α CNDP

Proof:

Let F be a CNF formula having at most three literals and r clauses C1, C2, …. Cr.

A k-coloring of an undirected graph G is an assignment of colors to nodes
such that each node is assigned a different color from all its neighbors,
and at most k colors are used.

k-COLORING: Given an undirected graph G and an integer k.
QUESTION: Is there a k-coloring of G?

The search problem is poly-time reducible to the decision problem: Why???
[Can use the decision problem in a binary search for the minimum k.]

We will show 3-COLORING is NP-complete.
Does this imply k-COLORING is NP-complete???
[yes. If could solve in poly-time for arbitrary k, then could solve
in poly-time for k=3.]

Theorem: 3-COLORING is NP-Complete.

Proof: (1) In NP: witness is a 3-coloring.

(2) Reduce 3-SAT to 3-COLORING.
(3) Given a 3-SAT formula of m clauses on n variables x1,x2,...,xn,
we construct a graph G as follows. We have
(a) a vertex vi for each variable xi,
(b) a vertex vi' for the negation of each variable xi,
(c) 5 vertices j1-j5 for each clause j,
(d) 3 special vertices: T, F, R

We would like T, F, and R to be forced to different colors,

so we will add edges between them to form a triangle.
For the remaining nodes, and node that is colored the same color
as T/F/R will be called colored TRUE/FALSE/RED, respectively.

We would like the edges to enforce the constraints on
satisfying assignments.

Constraint: For all i, exactly one of vi and vi' is colored TRUE
and one is colored FALSE.

Edges: for each i, form a triangle between vi, vi', and R.

Constraint: For each clause j, at least one of the literals in the
clause is colored TRUE.

Edges: for each clause j, say = (xi or not(xj) or xk), we have the
following gadget

```
vi ---j1
      |\
      | j3---j4
      |/     | \
vj'---j2     |  T
             | /
vk ----------j5
```

Claim: If each of vi, vj', and vk is colored TRUE or FALSE, then gadget
is 3-colorable iff at least one of vi, vj', and vk is colored TRUE.

Proof: If vi, vj', and vk are all colored false, then we are forced to
the following colors:

```
F  ---j1
      |\
      | j3--- F
      |/     | \
F  ---j2     |  T
             | /
F  ---------- R
```

But then j1, j2, j3 all must be colored different colors and NONE can be colored
F, so there is no legal coloring.

The remainder of the proof considers the 7 possible combinations of coloring vi, vj', and vk such that at least one is colored TRUE and the rest are colored FALSE, and shows that a 3 coloring exists in each case.
As an example, if vk is colored TRUE but vi and vj' are colored FALSE, we have
the following legal 3-coloring:

```
F  --- T
       |\
       | F --- R
       |/     | \
F  --- R       |  T
              | /
T  ----------- F
```

The other cases are similar and were presented in class.

The construction takes polynomial time.

(4) Follows from the above arguments.

Thus 3-COLORING is NP-complete.

Cook’s Theorem

Cook’s Theorem states that

Any NP problem can be converted to SAT in polynomial time.

In order to prove this, we require a uniform way of representing NP problems. Remember that what makes a problem NP is the existence of a polynomial-time algorithm—more specifically, a Turing machine—for checking candidate certificates. What Cook did was somewhat analogous to what Turing did when he showed that the Entscheidungsproblem was equivalent to the Halting Problem. He showed how to encode as Propositional Calculus clauses both the relevant facts about the problem instance and the Turing machine which does the certificate-checking, in such a way that the resulting set of clauses is satisfiable if and only if the original problem instance is positive. Thus the problem of determining the latter is reduced to the problem of determining the former.

Assume, then, that we are given an NP decision problem D. By the definition of NP, there is a polynomial function P and a Turing machine M which, when given any instance I of D, together with a candidate certificate c, will check in time no greater than P(n), where n is the length of I, whether or not c is a certificate of I.

Let us assume that M has q states numbered 0, 1, 2, . . . , q − 1, and a tape alphabet a1, a2, . . . , as. We shall assume that the operation of the machine is governed by the functions T, U, and D as described in the chapter on the Entscheidungsproblem. We shall further assume that the initial tape is inscribed with the problem instance on the squares 1, 2, 3, . . . , n, and the putative certificate on the squares −m, . . . , −2, −1. Square zero can be assumed to contain a designated separator symbol. We shall also assume that the machine halts scanning square 0, and that the symbol in this square at that stage will be a1 if and only if the candidate certificate is a true certificate. Note that we must have m ≤ P(n). This is because with a problem instance of length n the computation is completed in at most P(n) steps; during this process, the Turing machine head cannot move more than P(n) steps to the left of its starting point.

We define some atomic propositions with their intended interpretations as follows:

1. For i = 0, 1, . . . , P(n) and j = 0, 1, . . . , q − 1, the proposition Qij says that after i computation steps, M is in state j.
2. For i = 0, 1, . . . , P(n), j = −P(n), . . . , P(n), and k = 1, 2, . . . , s, the proposition Sijk says that after i computation steps, square j of the tape contains the symbol ak.
3. i = 0, 1, . . . , P(n) and j = −P(n), . . . , P(n), the proposition Tij says that after i computation steps, the machine M is scanning square j of the tape.

Next, we define some clauses to describe the computation executed by M:

1. *At each computation step, M is in at least one state.* For each $i = 0, \ldots, P(n)$ we have the clause

   $$Q_{i0} \vee Q_{i1} \vee \cdots \vee Q_{i(q-1)},$$

   giving $(P(n)+1)q = O(P(n))$ literals altogether.

2. *At each computation step, M is in at most one state.* For each $i = 0, \ldots, P(n)$ and for each pair j, k of distinct states, we have the clause

   $$\neg(Q_{ij} \wedge Q_{ik}),$$

   giving a total of $q(q-1)(P(n)+1) = O(P(n))$ literals altogether.

3. *At each step, each tape square contains at least one alphabet symbol.* For each $i = 0, \ldots, P(n)$ and $-P(n) \leq j \leq P(n)$ we have the clause

   $$S_{ij1} \vee S_{ij2} \vee \cdots \vee S_{ijs},$$

   giving $(P(n)+1)(2P(n)+1)s = O(P(n)^2)$ literals altogether.

4. *At each step, each tape square contains at most one alphabet symbol.* For each $i = 0, \ldots, P(n)$ and $-P(n) \leq j \leq P(n)$, and each distinct pair a_k, a_l of symbols we have the clause

   $$\neg(S_{ijk} \wedge S_{ijl}),$$

   giving a total of $(P(n)+1)(2P(n)+1)s(s-1) = O(P(n)^2)$ literals altogether

5. *At each step, the tape is scanning at least one square.* For each $i = 0, \ldots, P(n)$, we have the clause

   $$T_{i(-P(n))} \vee T_{i(1-P(n))} \vee \cdots \vee T_{i(P(n)-1)} \vee T_{iP(n)},$$

   giving $(P(n)+1)(2P(n)+1) = O(P(n)^2)$ literals altogether.

6. *At each step, the tape is scanning at most one square.* For each $i = 0, \ldots, P(n)$, and each distinct pair j, k of tape squares from $-P(n)$ to $P(n)$, we have the clause

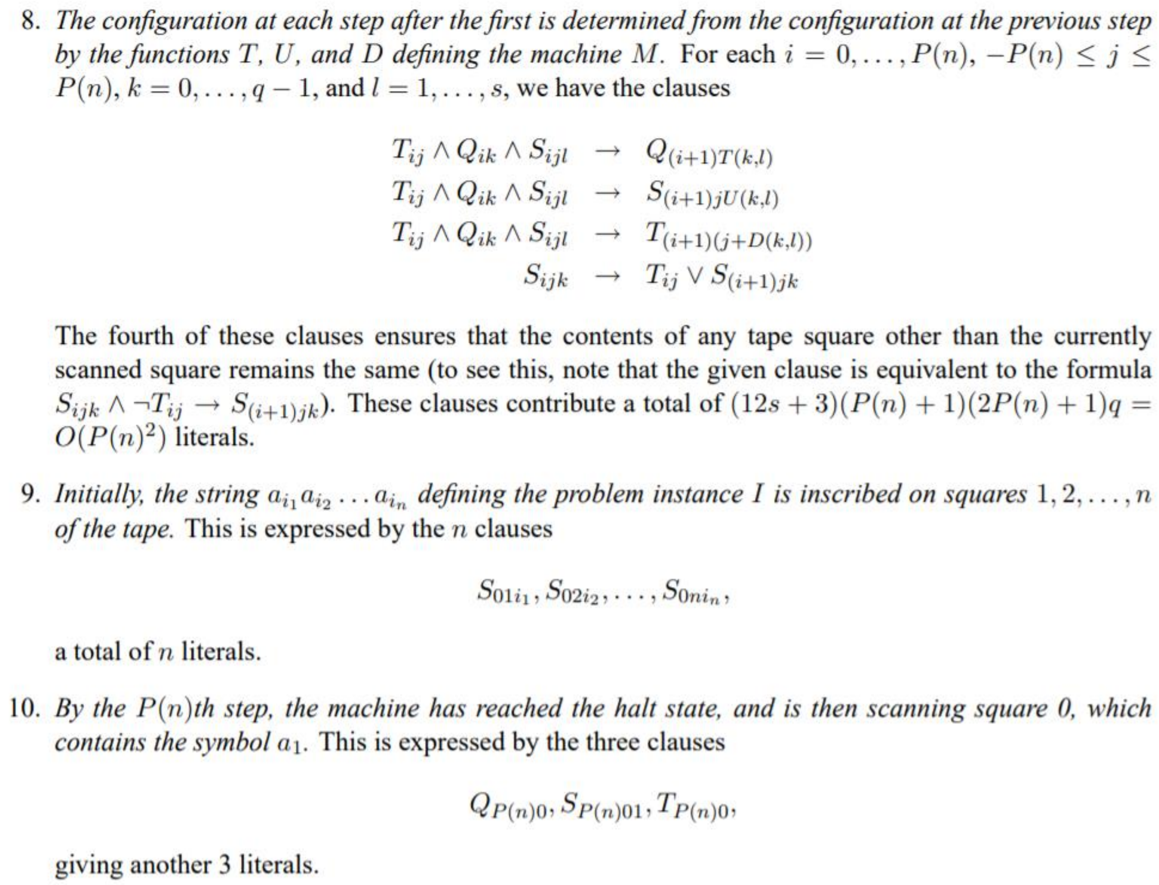
   $$\neg(T_{ij} \wedge T_{ik}),$$

   giving a total of $2P(n)(2P(n)+1)(P(n)+1) = O(P(n)^3)$ literals.

7. *Initially, the machine is in state 1 scanning square 1.* This is expressed by the two clauses

   $$Q_{01}, T_{01},$$

   giving just two literals.



8. *The configuration at each step after the first is determined from the configuration at the previous step by the functions T, U, and D defining the machine M.* For each $i = 0, \ldots, P(n)$, $-P(n) \leq j \leq P(n)$, $k = 0, \ldots, q-1$, and $l = 1, \ldots, s$, we have the clauses

$$\begin{aligned} T_{ij} \wedge Q_{ik} \wedge S_{ijl} &\rightarrow Q_{(i+1)T(k,l)} \\ T_{ij} \wedge Q_{ik} \wedge S_{ijl} &\rightarrow S_{(i+1)jU(k,l)} \\ T_{ij} \wedge Q_{ik} \wedge S_{ijl} &\rightarrow T_{(i+1)(j+D(k,l))} \\ S_{ijk} &\rightarrow T_{ij} \vee S_{(i+1)jk} \end{aligned}$$

The fourth of these clauses ensures that the contents of any tape square other than the currently scanned square remains the same (to see this, note that the given clause is equivalent to the formula $S_{ijk} \wedge \neg T_{ij} \rightarrow S_{(i+1)jk}$). These clauses contribute a total of $(12s+3)(P(n)+1)(2P(n)+1)q = O(P(n)^2)$ literals.

9. *Initially, the string $a_{i_1}a_{i_2}\ldots a_{i_n}$ defining the problem instance I is inscribed on squares $1, 2, \ldots, n$ of the tape.* This is expressed by the n clauses

$$S_{01i_1}, S_{02i_2}, \ldots, S_{0ni_n},$$

a total of n literals.

10. *By the $P(n)$th step, the machine has reached the halt state, and is then scanning square 0, which contains the symbol a_1.* This is expressed by the three clauses

$$Q_{P(n)0}, S_{P(n)01}, T_{P(n)0},$$

giving another 3 literals.

Altogether the number of literals involved in these clauses is O(P(n) 3) (in working this out, note that q and s are constants, that is, they depend only on the machine and do not vary with the problem instance; thus they do not contribute to the growth of the the number of literals with increasing problem size, which is what the O notation captures for us). It is thus clear that the procedure for setting up these clauses, given the original machine M and the instance I of problem D, can be accomplished in polynomial time.

We must now show that we have succeeded in converting D into SAT. Suppose first that I is a positive instance of D. This means that there is a certificate c such that when M is run with inputs c, I, it will halt scanning symbol a1 on square 0. This means that there is some sequence of symbols that can be placed initially on squares −P(n), . . . , −1 of the tape so that all the clauses above are satisfied. Hence those clauses constitute a positive instance of SAT.

Conversely, suppose I is a negative instance of D. In that case there is no certificate for I, which means that whatever symbols are placed on squares −P(n), . . . , −1 of the tape, when the computation halts the machine will not be scanning a1 on square 0. This means that the set of clauses above is not satisfiable, and hence constitutes a negative instance of SAT.

Thus from the instance I of problem D we have constructed, in polynomial time, a set of clauses which constitute a positive instance of SAT if and only I is a positive instance of D. In other words, we have converted D into SAT in polynomial time. And since D was an arbitrary NP problem it follows that any NP problem can be converted to SAT in polynomial time.