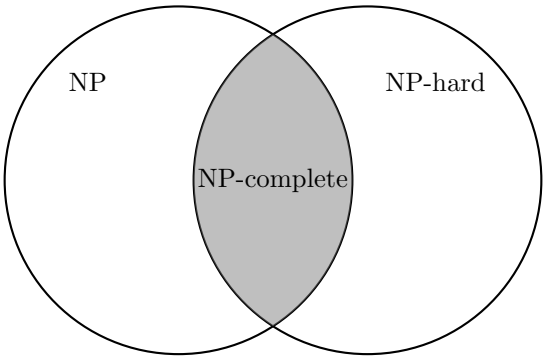
CS760 Topics in Computational Complexity **2024-25 Sem I**

Lecture 1: 02-08-2024

Scribe: 22B0923, 22B0912 *Lecturer: Rohit Gurjar*

1 NP-complete/NP-hard

A language L is said to be NP-hard if $\forall Q \in \text{NP}$, $Q \leq_p L$. L is NP-complete if $L \in \text{NP}$ and L is NP-hard.



Example:

Input : $M, x, 1^n, 1^t$

Output : Yes iff $\exists u \in \{0,1\}^n$ such that M outputs 1 on input $\langle x, u \rangle$ within t steps.

HW : Prove that the above is in NP and is NP-hard

1.1 SAT

SAT or Boolean satisfiability problem is the problem of determining whether a given boolean formula has a satisfying assignment. Its easy to see that SAT is in NP as any correct satisfying assignment serves as a certificate for true instance of the problem, and the certificate can be verified in polynomial time.

Cook-Levin Theorem- SAT is NP complete.
Proof:
Reduction: We already showed that SAT is in NP. To prove that it is also NP-hard we show a polynomial time conversion of any non-deterministic Turing machine M with an input x to a SAT instance $\phi_{M,x}$. That means if we can solve SAT then we can solve any problem in NP, as NP is defined by the problems solvable in polynomial time by a non deterministic Turing machine.

Let $p(n)$ be the max time required (polynomial) for an input of size n (as the problem is in NP there must exist a $p(n)$ such that any valid input can be solved in at most $p(n)$ time on an NTM). For $i = 0$ to $p(n)$ (time step) we define variables encoding the state of the system. Note that since the machine runs for time $p(n)$, it can only use the tape cells between $-p(n)$ to $+p(n)$ (so polynomial). For each time step we have a variable encoding the value of the tape ($2p(n)$ variables), we also create $2p(n)$ variables stating whether the head is at this position or not. We also create variables for the state the machine is currently in. The total number of such variables is polynomial.

Variables	Intended interpretation	How many? [6]
$T_{i,j,k}$	True if tape cell i contains symbol j at step k of the computation.	$O(p(n)^2)$
$H_{i,k}$	True if M's read/write head is at tape cell i at step k of the computation.	$O(p(n)^2)$
$Q_{q,k}$	True if M is in state q at step k of the computation.	$O(p(n))$

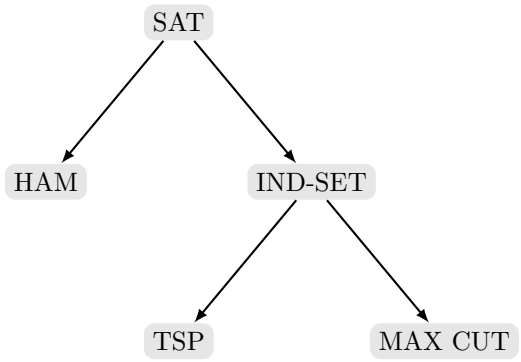
Figure 1: Encoding of variables. Image source: `https://en.wikipedia.org/wiki/Cook-Levin_theorem`

Expression	Conditions	Interpretation	How many?
$T_{i,j,0}$	Tape cell i initially contains symbol j	Initial contents of the tape. For $i > n-1$ and $i < 0$, outside of the actual input I, the initial symbol is the special default/blank symbol.	$O(p(n))$
$Q_{s,0}$		Initial state of M.	1
$H_{0,0}$		Initial position of read/write head.	1
$\neg T_{i,j,k} \vee \neg T_{i,j',k}$	$j \neq j'$	At most one symbol per tape cell.	$O(p(n)^2)$
$\bigvee_{j \in \Sigma} T_{i,j,k}$		At least one symbol per tape cell.	$O(p(n)^2)$
$T_{i,j,k} \wedge T_{i,j',k+1} \rightarrow H_{i,k}$	$j \neq j'$	Tape remains unchanged unless written by head.	$O(p(n)^2)$
$\neg Q_{q,k} \vee \neg Q_{q',k}$	$q \neq q'$	At most one state at a time.	$O(p(n))$
$\bigvee_{q \in Q} Q_{q,k}$		At least one state at a time.	$O(p(n))$
$\neg H_{i,k} \vee \neg H_{i',k}$	$i \neq i'$	At most one head position at a time.	$O(p(n)^3)$
$\bigvee_{-p(n) \leq i \leq p(n)} H_{i,k}$		At least one head position at a time.	$O(p(n)^2)$
$(H_{i,k} \wedge Q_{q,k} \wedge T_{i,\sigma,k}) \rightarrow \bigvee_{((q,\sigma),(q',\sigma',d)) \in \delta} (H_{i+d,\ k+1} \wedge Q_{q',\ k+1} \wedge T_{i,\ \sigma',\ k+1})$	$k < p(n)$	Possible transitions at computation step k when head is at position i.	$O(p(n)^2)$
$\bigvee_{0 \leq k \leq p(n)} \bigvee_{f \in F} Q_{f,k}$		Must finish in an accepting state, not later than in step $p(n)$.	1

Figure 2: Encoding of clauses(rules). Image source: `https://en.wikipedia.org/wiki/Cook-Levin_theorem`

intuition: In the above reduction we basically have stored all tapes values, state and head position at all times $< p(n)$. We can get a satisfying assignment of the above SAT formula for an input of NTM by simply running the NTM for $p(n)$ time steps on an accepting computation path and putting in the corresponding symbols on the NTM tape, head and state for every step in $p(n)$ as the assignment of the variables $T_{i,j,k}$, $H_{i,k}$ and $Q_{q,k}$, respectively according to the encoding. Due to how we have made our clauses this has to be a satisfying assignment of the SAT formula. Also a satisfying assignment of SAT is a valid run of the NTM that accepts in at most $p(n)$ steps. Therefore, we conclude that the NTM has an accepting path on input x if and only if the formula $\phi_{M,x}$ is satisfiable. Moreover, if we can find the solution of SAT efficiently then we can find a polynomial time accepting run of any NTM with just polynomial time overhead. This, combined with the fact that SAT is in NP, completes the proof of Cook-levin theorem that says that SAT is NP-complete.

Reductions are transitive.



2 co-NP complete

A decision problem C is co-NP-complete if it is in co-NP and if every problem in co-NP is polynomial-time many-one reducible to it. Each co-NP problem is the complement of some NP problem.
Example : Given a CNF formula, is it a tautology?
Examples which are in $co - NP \cap NP$ but not known to be in P:

- Integer factoring
- Discrete log: Given a, b, and p, find r such that

$$a^r \equiv b \pmod{p}.$$

Next, we are going to show another problem to be NP-complete.

Hamiltonian Path : Given a directed graph, does there exists a path from v_1 to v_n that covers all vertices.

Reduction of SAT to Hamiltonian path ($SAT \leq_p HAM$)
We'd like to create a graph where there is a path corresponding to each of the 2^n possible assignments to variables $x_1, \ldots, x_n$. And if the assignment satisfies the given Boolean formula then the corresponding path covers all vertices. How we do this is as follows, we make a path for each x_i, which can be traversed forwards and backwards. If moving from left to right, it's like assuming x_i is 1, and right to left denotes assigning x_i as 0. We will have connecting edges to move from 1 variable to the next, and one source and sink vertex. The length of each path is going to be twice the number of clauses (we'll see why we need this length later, just assume we choose some random length for now).

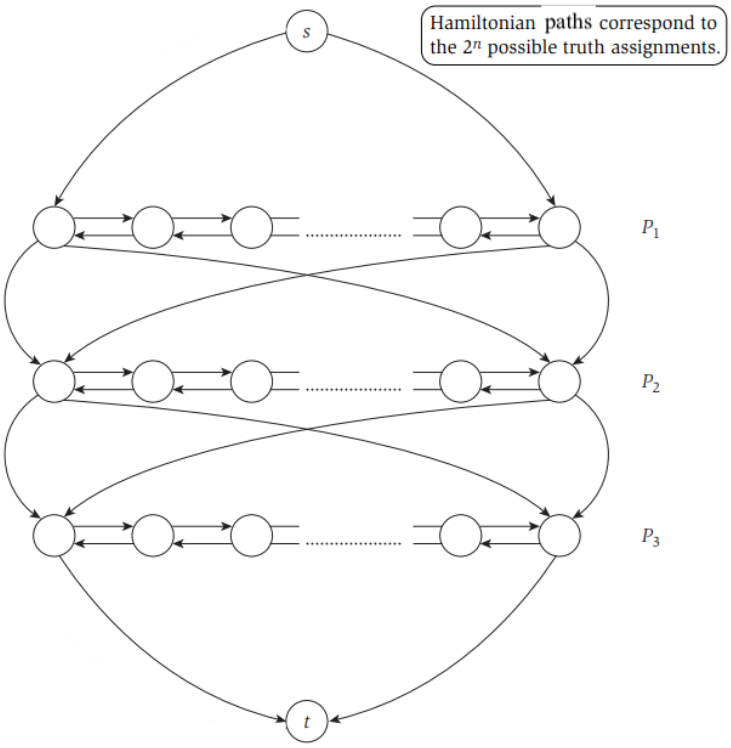
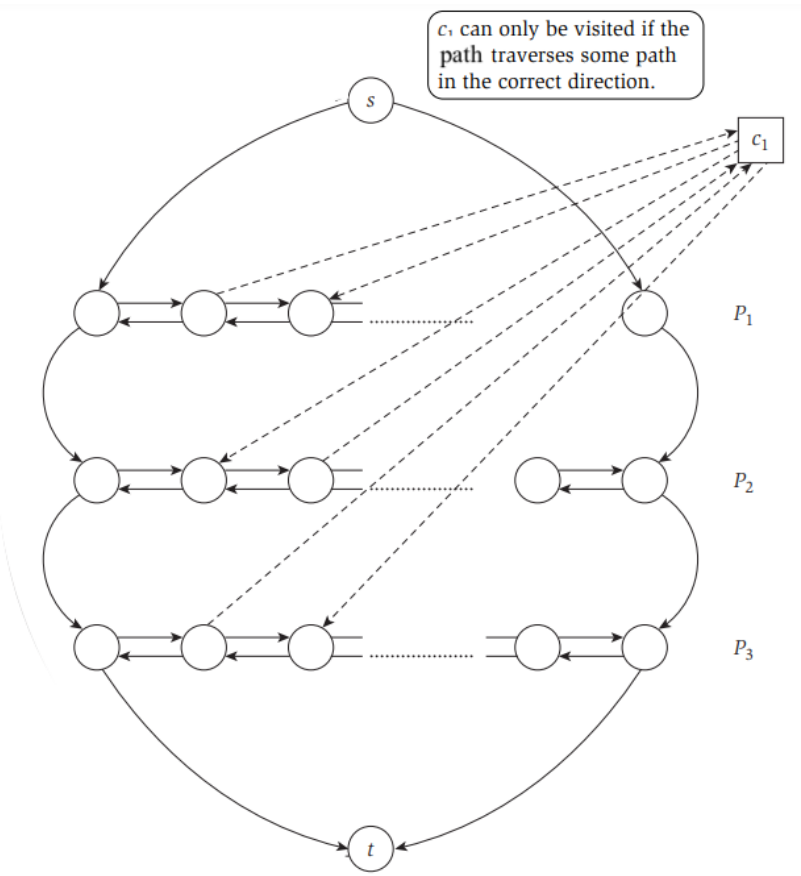


Figure 3: Image source: Kleinberg and Tardos, Algorithm Design, Chapter 8

For example, if wehave 3 variables, x_1, x_2, x_3 then the graph looks like the one in Figure 3. In this graph, there will be 2^3 Hamiltonian paths from s to t, because for each row, we can either go left to right, or right to left. We can't backtrack on any path because each vertex has to be visited exactly once, and each can correspond to some assignment.

Now we have to bring in the clauses. We add a node for each clause, and need to make sure it is visited only if the clause is satisfied. Let's say our first clause C_1 is $x_1 \vee x_2 \vee \neg x_3$. So whenever x_1 is true i.e we move left to right in row 1, we should be able to visit C_1. So what we do is add an edge from vertex 1 in row1 to C_1, and an edge from C_1 to vertex 2 in row 1. So instead of going straight from 1 to 2, we can divert to C_1 and come back. Since we have $\neg x_3$, we need to be able to visit C_1 while going from right to left. So we add an edge from vertex 2 in row 3 to C_1, and an edge from C_1 to vertex 1 in row 3. We're only going to use the first and second nodes in each row for C_1, these are allocated for C_1. Similarly third and fourth nodes are for C_2, fifth and sixth nodes are for C_3. Why are we keeping different nodes for each clause? This is because the same variable can make multiple clauses true, maybe even all clauses so we want to have the option of allowing a variable path to visit any number of clauses while moving along the row. Below is how the graph will look after filling everything for C_1, and also we have the full graph if C_2 is $\neg x_2 \vee x_3 \vee x_4$, and C_3 is $x_1 \vee \neg x_2 \vee x_4$.



Graph showing one clause node c_1. Image source: Kleinberg and Tardos, Algorithm Design, Chapter 8

Now if we find a Hamiltonian path in the graph, we can easily find out the assignment which works. Just see in which direction we move in each row, and correspondingly assign each variable. This will satisfy the assignment as the Hamiltonian path visits every clause, meaning at least one variable assignment is there to make every clause true. Similarly, if we have a satisfying assignment we'll definitely have a Hamiltonian path. We can find out in which direction to move through each row from the assignment. Since the assignment is satisfying, there will be at least 1 variable assignment making each clause true, and whatever that is, we can take a detour there and visit that clause. Since we allocated different sections for each variable connecting to each clause, this is feasible.