CS5110: Complexity Theory

Shreyas Havaldar
CS18BTECH11042

October 20, 2020

1 Assignment 1

1.1 Let $U = \{\langle M, x, \#^t\rangle |$ M accepts input x within t steps on at least one computing path$\}$. To show that U is NP-complete.

For U to be NP-complete, we need to show U $\in$ NP and U is NP-Hard.

To prove U $\in$ NP: Let there be a non-deterministic turing machine N, such that N outputs the result of running M on x, when both M and x are provided as input to N, in time polynomial in the time taken by M to run on input x.

Now we merely simulate N on $\langle M, x\rangle$ for time p(t) where t is the time taken by M to run on x and p(n) is a fixed polynomial function via a poly time non-det turing machine TM. Note TM is poly time, as we have defined N such that it runs in poly time and thus we have the simulation of non-det N, to output the result on TM in poly time This polynomial function p(n) is fixed for a M, and does not change with the change in the input, and thus we know irrespective of the input config, the maximum time in which M will accept x as a function of input size and can simulate running of M on x for that many steps necessarily.

TM accepts $\langle M, x, \#^t\rangle$ if and only if, N accepts $\langle M, x\rangle$ in p(t) steps, which occurs if and only if M accepts x within t steps. Therefore by using a non-det TM, we have decided U in poly time, therefore U $\in$ NP.

To prove U is NP-Hard. Therefore to prove: $\forall L \in NP, L \leq_P U$. Therefore need to show a poly time reduction to U, from every L in NP exists. Therefore, $\forall x, x \in L \iff f(x) \in U$.

As L $\in$ NP, there exists a non-det turing machine that decides L in poly time say p'(n), where the size of the input is n. We define f(x) as the polynomial function such that, f(x) = $\langle M, x, \#^{p'(|x|)}\rangle$. As here x is our input, $|x|$ is our input size, and the non-det poly time turing machine M, decides x in time within $p'(|x|)$ steps on some branch. For the reduction to be poly time, f(x) needs to be poly in x. This is obvious as M is independent of x and thus need not be calculated and p'(n) is a poly function in n, and thus can be computed in poly time on input, for a given p'(n).

We know 3-SAT is NP-Complete, thus NP-Hard. Here showing a fixed upper bounding p'(n) is obtainable for any input given to the 3-SAT problem, we know a p'(n) irrespective of the input is obtainable for every L in NP. It might be different for every L, but there will indeed exist a poly time function as defined above.

Now to show a p'(n), such that 3-SAT can be decided by a non-det turing machine M in atmost p'(n) steps irrespective of the input. So for a 3-CNF formula ϕ, with m variable literals and c clauses, we randomly choose an assignment for the variable literal, and for every branch of non-determinism we have a certain assignment made to each of the m variable literals. We would have 2^m branches, one corresponding to a particular unique assignment. Now our M, has an input x = $\langle\phi\rangle$, and $|x|$ = $|\phi|$ and while evaluating our assignment, we are simply replacing the the variable literals with an arbitrary assignment and thus the time taken to evaluate the assignment is linear in input, we maintain the current truth value of the clause and the overall truth value of the 3-CNF on a work tape and keep modifying them according to the assignment made to the literal read. Thus $|x| = \Omega(3c)$ and our evaluation takes time $O(3c)$. (Can ignore constants thus same.) If the evaluation on any branch evaluates to TRUE, we return TRUE, else we return FALSE. Thus, we have decided 3-SAT in poly time using a non-det turing machine. Thus the polynomial p'(x), is p'(x) = kx for 3-SAT, where k is a constant. (We simply take a large arbitrary constant say 777 here.)

So we have indeed shown that a fixed p'(x) would exist irrespective of M, and we can encode this function in poly time itself on the tape by printing t #'s, along with M and x.

Thus U is NP-Hard.

Thus U is NP-complete.

Hence Proved!

1.2 To show that KNAPSACK is NP-complete while KNAPSACK-INT is in P.

$KNAPSACK = \{\langle a_1, a_2, \ldots a_n, b\rangle | \exists x_1, x_2, \ldots x_n \in \{0,1\}$ such that $\Sigma_{i=i}^{n} a_i x_i = b\}$

For KNAPSACK to be NP-complete, we need to show KNAPSACK $\in$ NP and KNAPSACK is NP-Hard.

To prove KNAPSACK $\in$ NP: Let there be a non-deterministic turing machine N, such that N calculates the sum of the arbitrarily chosen elements from $A = \{a_1, a_2, \ldots a_n\}$. Each branch of non-determinism has a different subset, S of elements chosen from A ($S \subseteq A$). We simply calculate the sum of the elements on this branch in poly time of input as the input must be $\Omega(A)$, where A is the set of all elements that can be chosen to sum, therefore our arbitrarily chosen set of elements, $|S|$ is $O(|A|)$, and the sum can be performed while reading the elements on another tape which would also likewise not take more than poly time in input. If for any branch of non determinism, the set of elements chosen sums up to b, we return true, else we return false. (Choosing an element a_i of the set is equivalent to making $x_i = 1$).

Thus we have decided KNAPSACK in poly time using a non-det TM, thus KNAPSACK $\in$ NP.

Now to show KNAPSACK is NP-Hard: We know 3-SAT is NP-Complete, thus NP-Hard. If we show a polynomial reduction from 3-SAT to KNAPSACK, then KNAPSACK is NP-Hard. If $3-SAT \leq_p KNAPSACK$, we would be able to reduce all languages in NP, to $KNAPSACK$, by using 2 successive poly time reductions, first to 3-SAT and then from the above found 3-SAT to KNAPSACK. Therefore $\forall L, L \in NP, L \leq_p KNAPSACK$.

Now to show the reduction: Let our 3-CNF formula ϕ consist of m variable literals called v_i and c clauses called C_i.

Note: Indexing starting from 1, and moving left to right in the number.

$\forall v_i, i \in \{1, 2 \dots m\}$: we construct t_i and f_i, both of number of digits = m+c, initialized with m+c 0's each, such that:

- $\forall i \in \{1, 2 \dots m\}, t_{i,i} = f_{i,i} = 1$ (j^{th} index in i^{th} number is represented by $t_{i,j}$)
- If clause C_{j-m} contains $v_i, \forall j \in \{m+1, m+2 \dots m+c\}, t_{i,j} = 1$
- If clause C_{j-m} contains $\neg v_i, \forall j \in \{m+1, m+2 \dots m+c\}, t_{i,j} = 1$
- All other positions in all t_i and f_i, which have not be value 1 by any of the above three procedures remain 0.

Likewise $\forall C_i, i \in \{1, 2 \dots c\}$: we construct p_i and q_i, both of number of digits = m+c, initialized with m+c 0's each, such that:

- $\forall i \in \{1, 2 \dots c\}, p_{i,i+m} = q_{i,i+m} = 1$ (j^{th} index in i^{th} number is represented by $p_{i,j}$)
- All other positions in all t_i and f_i, which have not be value 1 by any of the above procedure remain 0.

Now, we create a number s of m+c digits, again initialized to m+c 0's, such that:

- $\forall i \in \{1, 2 \dots m\}, s_i = 1$ (i^{th} index in s is represented by s_i)
- $\forall i \in \{m+1, m+2 \dots m+c\}, s_i = 3$

Note: Our set $A = \{a_1, a_2, \dots a_n\}$ from KNAPSACK is basically the union of all the numbers we have constructed above in decimal notation, all m each of t_i, f_i and all c each of p_i, q_i that we have constructed above. Our b in KNAPSACK is the sum s we created above in decimal notation.

Now we have constructed KNAPSACK from starting out from 3-SAT input configuration. We demonstrate that ϕ is satisfiable iff on some assignment of value to $x_1, x_2, \dots x_n \in \{0, 1\}$, $\Sigma_{i=i}^{n} a_i x_i = s$. To show the reduction indeed is correct:

If 3-SAT is satisfiable, KNAPSACK must also be satisfied:

- If v_i is TRUE in our satisfying assignment, we have $x_j = 1$ for $j, a_j = t_i$.
- If v_i is FALSE in our satisfying assignment, we have $x_j = 1$ for $j, a_j = f_i$.

- If number of TRUE variable literals in clause C_i in our satisfying assignment is atmost 2, we have $x_j = 1$ for $j, a_j = p_i$.
- If number of TRUE variable literals in clause C_i in our satisfying assignment is exactly 1, we have $x_j = 1$ for $j, a_j = q_i$.
- All other x_j that were not assigned 1 by any of the above 4 procedures remain 0.

This indeed would add up to b = s, as we have a 1 in each of the first m digits because $\forall i$ we have in the sum necessarily one of t_i or f_i. Also, each of the last c digits is a necessarily a number between 1 and 3 as every clause is satisfied and thus must contain between 1 and 3 variable literals that have been assigned value TRUE. For each of the rightmost c positions in the sum b=s, we would have digit 3 and by our construction we would choose both p_i or q_i thus adding 2 at that position, when only 1 is being added at that position by the one literal in the clause is TRUE, would choose only p_i thus adding 1, when two literals added have value 1 at that position(thus total 2) as two literals in that clause are assigned TRUE, would choose none of p_i or q_i to add when all three literal add one each, when each in the clause is assigned TRUE. And each clause must be TRUE for the satisfying assignement, so we won't have the case when every literal in the clause is assigned value FALSE. Thus when 3-CNF is satisfiable, our mapping is correct!

Now when the 3-CNF is not satisfiable, KNAPSACK would also not be satisfiable, or by contrapositive, when KNAPSACK has a valid assignment, 3-CNF is also satisfiable. Each number added in the KNAPSACK has only digits 0 or 1, and thus there are no issues like carry-forward as each index would contain maximum 3 1's from the literal variables (t_i or f_i) and atmost 2 1's from the clause variables (p_i or q_i) from our construction and thus can worst case be summed up to 5. Our sum is b=s.

- If we have $x_j = 1$ for $j, a_j = t_i$, v_i is TRUE in our satisfying assignment.
- If we have $x_j = 1$ for $j, a_j = f_i$, v_i is FALSE in our satisfying assignment.

This indeed would be a satisfying assignment to ϕ, each of the rightmost c digits must add up to 3. And at most 2 additions can come from p_i and q_i for i^{th} digit from our construction, thus atleast 1 addition at this position must come from having either f_i or t_i a part of the sum we make in KNAPSACK to add all up to 3 at this position. So either one of v_i or $\neg v_i$ must exist in the clause to make up for the 1 addition and thus one of them would necessarily be TRUE, thus making the entire clause TRUE. This would be the case for every clause and thus every clause would be TRUE if KNAPSACK condition is indeed satisfied and thus we would have a satisfying assignment for our 3-CNF, thus making the 3-SAT satisfiable.

Thus we have shown both possible possibilities!

We know that the reduction is poly time as we make assignments to $m+c$ digits of (2m+2c+1) numbers in constant time operation for every assignment. So the total time is $O((2m + 2c + 1) * (m + c))$ which is polynomial in our input size, the 3-CNF provided to us, which is $\Omega(3 * c)$. (Number of variables cannot exceed 3c even if we assign 3 unique variables to each clause, so $m \leq 3c$) and assuming no variables that are not present in the clause either in as it is or as a negation, need to be considered.

Thus reduction takes $O(|Inp|^2)$ time and is poly time in input size $|Inp|$.

Thus, 3-SAT $\leq_p$ KNAPSACK, and KNAPSACK is NP-Hard.

Thus KNAPSACK is NP-Complete.

Hence Proved!

To show KNAPSACK-INT is in P. $KNAPSACK - INT = \{\langle a_1, a_2, \dots a_n, b\rangle | \exists x_1, x_2, \dots x_n \in \mathbb{Z}$ such that $\Sigma_{i=i}^{n} a_i x_i = b\}$

We know from Bézout's identity:

$\gcd(a_1, a_2, \dots, a_n) = d \iff \exists x_1, x_2, \dots, x_n \in \mathbb{Z}$ such that $d = a_1x_1 + a_2x_2 + \cdots + a_nx_n$

For a given input to be accepted, if b is a multiple of the Greatest Common Divisor(d) of all n elements from the fixed set $A = \{a_1, a_2, \dots a_n\}$, then $\exists x_1, x_2, \dots x_n \in \mathbb{Z}$ such that $\Sigma_{i=i}^{n} a_i x_i = b\}$ necessarily exist. (We can multiply each element by a constant k, to achieve b=kd).

So to check if the given input belongs to KNAPSACK-INT, we simply need to calculate the GCD of the set A and find if b is divisible by the GCD, if it is then we return true, else we return false.

We calculate the GCD of n elements by calculating it pairwise, starting from $d = GCD(a_1, a_2)$, and then $\forall i \in \{3, 4 \dots n\}, d = GCD(d, a_i)$. We know that the calculation of GCD takes $\log(min(a, b))$ where a and b are the two numbers whose GCD is to be found by using Euclid's algorithm. We can denote the integers in binary notation and thus input size $|a| = \log a$ for all a. Thus our algorithm to find GCD runs in $2^{\log |b|} = |b|$ time for a pair where b is the smaller integer. Thus for our (n-1) pairwise calculations of GCD to find the GCD for the entire set A, the time taken would be $O(\Sigma_{i=i}^{n} |a_i|)$, as we would do lesser than n comparisons necessarily. Our set A would also be of the size $\Omega(\Sigma_{i=i}^{n} |a_i|)$ where m is the smallest integer. Thus the GCD calculation is $O(Inp)$, where Inp is the Input size.

Further to check if b is divisible by the GCD d we found, we would repeatedly subtract d from b until b becomes strictly lesser than d on our work tape. If at this point the value of b is equal to 0, we know that b is divisible by d. This algorithm would run in $\frac{b}{d}$ steps which we know would take $|b|^2$ time to run, when b is represented in binary notation. If b is divisible by d, we return true, else we return false.

Thus we need a sequence of linear time and quadratic time algorithms in the input size to determine KNAPSACK-INT. Thus we need only a sequence of poly time algorithms with no non determinism involved at any step of the algorithm to check if the given input say x $\in$ KNAPSACK-INT, thus KNAPSACK-INT $\in$ P.

Hence Proved!

1.3 If 3-SAT is reducible in polynomial time to $\overline{3-SAT}$, to show that polynomial hierarchy collapses to NP.

We know that 3-SAT is NP-complete and $\overline{3-SAT}$ is coNP-complete as we proved in an earlier assignment that A is coNP-complete $\iff \overline{A}$ is NP-complete, and we know 3-SAT is NP-complete.

As we know 3-SAT is NP-hard, if we show a polynomial reduction from 3-SAT to $\overline{3-SAT}$, then $\overline{3-SAT}$ is NP-hard. If $3-SAT \leq_p \overline{3-SAT}$, we would be able to reduce all languages in NP, to $\overline{3-SAT}$, by using 2 successive poly time reductions, first to 3-SAT and then using the above found reduction 3-SAT to $\overline{3-SAT}$, thus $\overline{3-SAT}$ would be NP-hard.

Therefore $\forall L, L \in NP, L \leq_p \overline{3-SAT}$. This can be seen from the definition of NP-hardness, our $\overline{3-SAT}$ is necessarily harder than every language in NP, and thus the class NP must be a subset of the class coNP, to which our $\overline{3-SAT}$ belongs. Therefore $NP \subseteq coNP$ and $\forall L, L \in NP \implies L \in coNP$.

And from the fact that $\overline{3-SAT}$ is coNP-complete, $\overline{3-SAT}$ is coNP-hard and thus $\forall L, \overline{L} \in NP, L \leq_p \overline{3-SAT}$ (As we define $coNP = \{L|\overline{L} \in NP\}$).

Also, let the polynomial function reducing 3-SAT to $\overline{3-SAT}$ be f. $x \in 3{-}SAT \iff f(x) \in \overline{3-SAT}$ and as $x \notin 3-SAT \iff x \in \overline{3-SAT}$ and $x \notin 3-SAT \iff f(x) \notin \overline{3-SAT}$. Combining the two we get, $f(x) \notin \overline{3-SAT} \iff f(x) \in 3-SAT$ and we can thus use the same function f to reduce $\overline{3-SAT}$ to 3-SAT. $x \in \overline{3-SAT} \iff f(x) \in 3-SAT$.

Now we can show a similar proof as shown above for all languages in NP reducing to $\overline{3-SAT}$ in polytime using two successive reductions. Here we first reduce $L \in coNP$ to $\overline{3-SAT}$ and then from $\overline{3-SAT}$ to 3-SAT using f(x), to show all languages in coNP reduce to $3-SAT$ in polytime. Thus $\forall L, L \in coNP, L \leq_p 3-SAT$ and 3-SAT is coNP-hard. Again by the definition of hardness, 3-SAT is harder than all problems in coNP and thus $coNP \subseteq NP$, and $\forall L, L \in coNP \implies L \in NP$.

Therefore, $NP \subseteq coNP$ and $coNP \subseteq NP$ which implies NP=coNP. Thus if 3-SAT is reducible in poly time to $\overline{3-SAT}$, NP=coNP.

We know, if NP=coNP, then $\Sigma_2 = \Pi_2 = NP = coNP$. From generalizing the proof we have done in class to show if $\Sigma_i = \Pi_i$, $\Sigma_{i+1} = \Pi_{i+1} = \Sigma_i = \Pi_i$:

First $\Sigma_2 = \Pi_2 = \Sigma_1 = \Pi_1$, where $\Sigma_1 = \Pi_1$ is $NP = coNP$ respectively as $\Sigma_1 = NP$ and $\Pi_1 = coNP$. Now as $\Sigma_2 = \Pi_2$, $\Sigma_3 = \Pi_3 = \Sigma_2 = \Pi_2 = NP = coNP$ using our class proof. Thus like a chain reaction, sequentially all levels in the polynomial hierarchy will collapse to their lower ones, which would have themselves collapsed to NP=coNP and thus the entire polynomial hierarchy would collapse to NP=coNP. Thus the entire polynomial hierarchy would collapse to NP if 3-SAT is reducible in polynomial time to $\overline{3-SAT}$.

Hence Proved!

1.4 Suppose polynomial hierarchy has a complete problem with respect to polynomial time reductions. To prove that the polynomial hierarchy collapses.

Let the complete problem be C. Therefore as C is PH-complete, C $\in$ PH and C is PH-hard. Now we know PH $= \bigcup_{\forall i} \Sigma_i$. (Also $= \bigcup_{\forall i} \Pi_i$).

Note: As $\Pi_i \subseteq \Sigma_{i+1}$ and $\Sigma_i \subseteq \Pi i + 1$, so we can proceed using only the union of either one type of the classes, Σ or Π and need not do union over both types of classes for all i. I proceed with the result PH $= \bigcup_{\forall i} \Sigma_i$, without loss of generality.

Now, as our C $\in$ PH, $\exists i$, C $\in \Sigma_i$. Now as C is PH-hard, $\forall L, L \in PH, L \leq_p C$. Therefore, $\forall L, L \in PH \implies L \in \Sigma_i$. This means that PH $\subseteq \Sigma_i$. This can be seen from the definition of K-hardness of a problem, our complete problem is necessarily harder than every language in K, and thus the K must be a subset of the class K', to which our complete problem C belongs. Thus as C is harder than every language in the polynomial hierarchy, every language in the polynomial hierarchy would belong to the class Σ_i to which C belongs.

Thus the entire PH has collapsed to the level i. Thus we have shown that the polynomial hierarchy collapses to a finite level if we have a complete problem with respect to polynomial time reductions.

Hence Proved!

1.5 To show that 2-SAT is in P.

2-SAT $= \{\langle\phi\rangle | \phi$ is a satisfiable 2-CNF formula$\}$

We know $a \vee b \equiv (\neg a \to b) \wedge (\neg b \to a)$.

Let there be an implication graph G with directed edges, of 2m nodes, where m are the number of variable literals in the 2-CNF formula. For each literal, we have two nodes in the graph corresponding to the literal and the negation of the literal. So for every clause $C_i = a \vee b$, we create 2 directed edges in the implication graph G, one from node $\neg a$ to b and the other from $\neg b$ to a. Thus for a 2-CNF formula with c clauses, we have 2c edges in our graph G.

Due to the propagation property of implication, $(a \to b) \wedge (b \to c) \equiv a \to c$, and therefore in G, having a path from a to c is sufficient to show that $a \to c$ and there need not necessarily be a direct edge between them.

As we know from the construction of our graph, for every edge a to b, we also have an edge from $\neg b$ to $\neg a$, and thus for every path from i to j, we also have an path from $\neg j$ to $\neg i$. Let the path be $i \to v_1 \to v_2 \ldots \to v_k \to j$. Then as edges $\neg v_1 \to \neg i, \neg v_2 \to \neg v_1 \ldots \neg j \to \neg v_k$ would also exist, a path from $\neg j$ to $\neg i$ would necessarily exist.

If there is an path from a to b, for $a \to b$ to be true: if a is true, b has to be true as well (along with each of the intermediate vertices); if a is false, then b can take any value, likewise for the intermediate

vertices. We simply propagate the truth value via the sequential implication between two vertices with a directed edge between them such that they lie on this path.

Assuming the 2-CNF is satisfiable and thus each variable is assigned a value. If there exists a path from v to $\neg v$ and from $\neg v$ to v, we cannot choose an assignment for the literal v. If we assign TRUE to v, then $\neg v$ must also be TRUE due to the path from v to $\neg v$, which means v must be FALSE, which is a contradiction. If we assign FALSE to v, then $\neg v$ is TRUE, and due to the path from $\neg v$ to v, we must assign v, the value TRUE, which is a contradiction. Thus if we can say that 2-CNF is unsatisfiable iff there exists a path from v to $\neg v$ and from $\neg v$ to v, because we would find no assignment for the literal v that would satisfy the 2-CNF by our construction. Or the 2-CNF is satisfiable, iff no such pair of paths exist.

For the case when 2-CNF is satisfiable, no such pair of paths must exist, or by contrapositive if such a pair of paths exists, 2-CNF is unsatisfiable. Thus if both v and $\neg v$ belong to the same strongly connected component, we know there exists a path from both v to $\neg v$ and from $\neg v$ to v. Thus we just need to find if there is a variable literal such that it and its negation both belong to the same strongly connected component in our construction of the graph G. This can be done in $O(|V| + |E|)$ via the simply performing a simple Depth First Search algorithm in time $O(|V| + |E|)$ for every vertex in G and seeing if the vertex corresponding to the negation of the variable is reachable. Thus we would perform $|V|$ DFS runs, and each DFS would be worst case $O(|V|^2)$ as any graph can have no more than $|V|^2$ edges. So $|E| = O(|V|^2)$ and thus total operation of checking for a path existence from every vertex to its negation, would be $O(|V|^3)$. (We can also directly apply Kosaraju's Algorithm which basically performs DFS twice in a smart manner, for finding the strongly connected components in G and then ensure a variable and its negation don't lie in the same SCC in $O(|V| + |E|)$). I need the graph to be separated into strongly connected components to provide an assignment when the 2-CNF is satisfiable. Here we only care about decidability and thus I did not perform these operations via our turing machine.

Now for the other possibility, that 2-CNF is unsatisfiable and thus for no v, v and $\neg v$ belong to the same strongly connected component. By contrapositive, if there is no variable v $\in$ V, such that there exists a path from both v to $\neg v$ and from $\neg v$ to v, we can say that the 2-CNF is satisfiable.

We can prove this direction by giving an algorithm for providing a satisfying assignment independent of the input configuration given to us by following these steps:

1. We topologically sort the strongly connected components. A vertex u, such that there is a path from u to w, lies before w in the sorting. Each vertex is addressed by the strongly connected component it belongs to henceforth.

2. We assign value TRUE to all variable vertices such that they lie after their negation variable vertices in the topological sort and FALSE to the negations of all these vertices.

3. We assign value FALSE to all variable vertices such that they lie before their negation variable vertices in the topological sort and TRUE to the negations of all these vertices.

This kind of assignment is bound to work as both a variable and its negation don't lie in the same SCC and thus occupy different positions in the topological sort. Suppose a vertex v is assigned value TRUE, then it must lie after $\neg v$ in the sorted order and thus no path from v to $\neg v$ would exist, so we would not be forced to assign TRUE to $\neg v$ causing a contradiction and we can safely assign $\neg v$ value FALSE. Likewise if we assign FALSE to a vertex u, then it must lie to the left of $\neg u$, and we would have no contradiction.

Also, there won't exist a w such that there would have a path to both w and $\neg w$ from the vertex v, because by the construction of our implication graph, it would cause a contradiction if we assign TRUE to v, thus making w and $\neg w$ both TRUE. If there indeed exists such a variable w, then due to the property we proved earlier, there would exist a path from $\neg w$ to $\neg v$ and w to $\neg v$. And thus there would be a path from v to $\neg v$, which would contradict our assumption made above that v could be assigned value TRUE. Thus no such vertex w can exist.

This algorithm would take $O(|V|+|E|)$ time for topological sorting as it is basically DFS with additional space usage. Whenever we encounter a variable that is yet unassigned, we assign a value according to the procedure defined above which takes a single traversal of all the sorted vertices, we assign the variable v(can be a negation of a literal as well) we encounter first in the sorted order value FALSE, if $\neg v$ has not been encountered yet, thus v lies to the left of $\neg v$.

It is not included in the time taken by the TM to decide 2-SAT as this algorithm is only providing a proof for our equivalence. We need not show it every-time for the 2-CNF formula we are given as input and the implication graph we create, it has been proved TRUE irrespective of the input. We only need to decide 2-SAT via our implication graph and not provide an assignment for it. Therefore our only calculation done on the tape of the deterministic turing machine is the DFS operation from each of the 2m vertices in the graph.

Thus we have shown both possibilities, 2-CNF being satisfiable and 2-CNF not being satisfiable, and proved the equivalence between our implication graph constructed and the 2-SAT problem.

Note: We know $|V| = 2m$ and $|E| = 2c$. I'm assuming we are given a list of all the variable literals that can be used in the 2-CNF formula, thus input being $|V|/2$ (if there is a situation such that a large number of literals are not used as literals in clauses, thus creating a sparse implication graph). In general $|E| = O(|V|^2)$ and in the case when all literal variables are indeed used in the clauses but not listed, our literal variables m can be atmost 2c. (each clause has 2 unique literals). So $m \leq 2c$ And then as each clause creates 2 edges, we have input size $(|E| = 2c) \geq (|V|/2 = 2m/2)$, thus we can express our algorithm in poly time of $|V|$ and the algorithm is polytime in the input.

The input given to us is the 2-CNF formula, which is necessarily $\Omega(|V|)$ at both extremes cases, and thus all cases. And as we have show we only perform $O(|V|)$ DFS operations and thus, our we know our construction and checking algorithm would run in polytime of $O(|Inp|)$, or $O(|Inp|^3)$ always, where $|Inp|$ is the input size. irrespective of the type of implication graph we create and thus we have a deterministic algorithm to decide 2-SAT in poly time of input, and thus 2-SAT is in P.

Hence Proved!