Prove of NP Completeness

Dr. Bibhudatta Sahoo

Communication & Computing Group

CS215, Department of CSE, NIT Rourkela

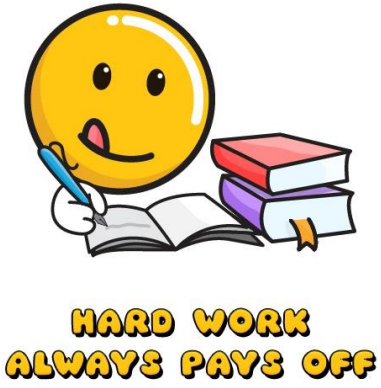
Email: bdsahu@nitrkl.ac.in, 9937324437, 2462358

What is the definition of P, NP, NP-complete and NP-hard?

	P	NP	NP-complete	NP-hard
Solvable in polynomial time	✓			
Solution verifiable in polynomial time	✓	✓	✓	
Reduces any NP problem in polynomial time			✓	✓

How do we prove a problem is NP complete

- Given a problem U, the steps involved in proving that it is **NP-complete** are the following:
- Step 1: Show that U is in NP.
- Step 2: Select a known NP-complete problem V.
- Step 3: Construct a **reduction from** V to U.
- Step 4: Show that the reduction requires **polynomial time.**



Hamiltonian Path

- Hamiltonian Path is a path in a directed or undirected graph that visits each vertex exactly once. The problem to check whether a graph (directed or undirected) contains a Hamiltonian Path is NP-complete, so is the problem of finding all the Hamiltonian Paths in a graph.

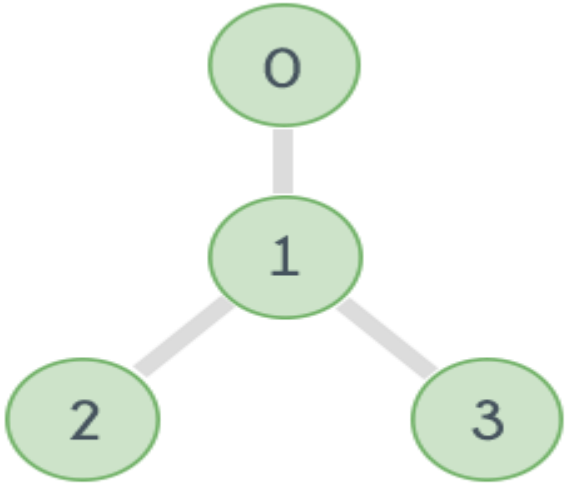


Fig. 1

Graph shown in Fig.1 does not contain any Hamiltonian Path

Hamiltonian Path

- A Hamiltonian Path in a graph having N vertices is nothing but a permutation of the vertices of the graph $[v_1, v_2, v_3, \ldots\ldots v_{N-1}, v_N]$, such that there is an edge between v_i and v_{i+1} where $1 \leq i \leq N-1$. So it can be checked for all permutations of the vertices whether any of them represents a Hamiltonian Path or not.

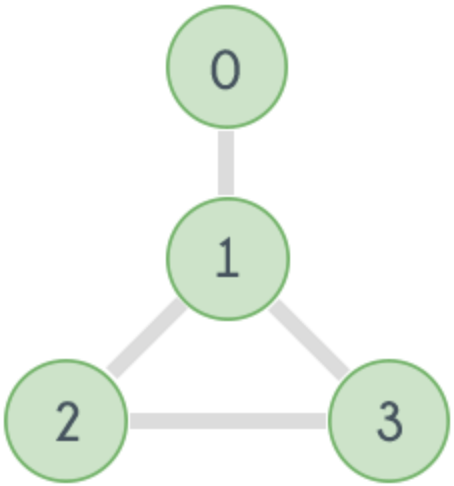


Fig. 2

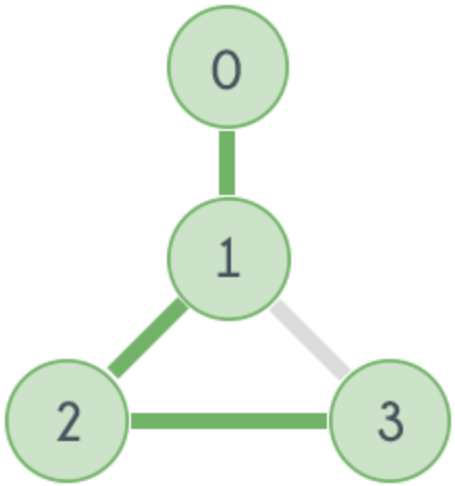


Fig. 3

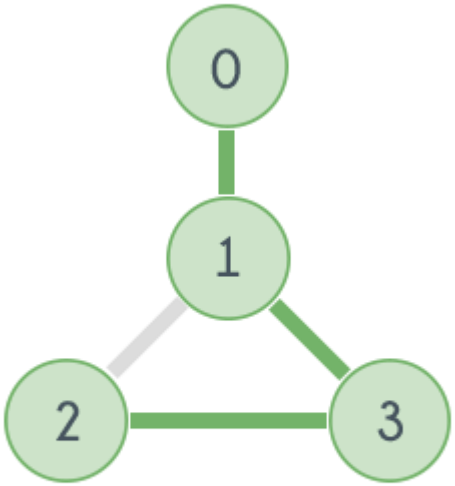


Fig. 4

Hamiltonian Path

- For example, for the graph given in Fig. 2 there are 4 vertices, which means total 24 possible permutations, out of which only following represents a Hamiltonian Path

0-1-2-3

3-2-1-0

0-1-3-2

2-3-1-0

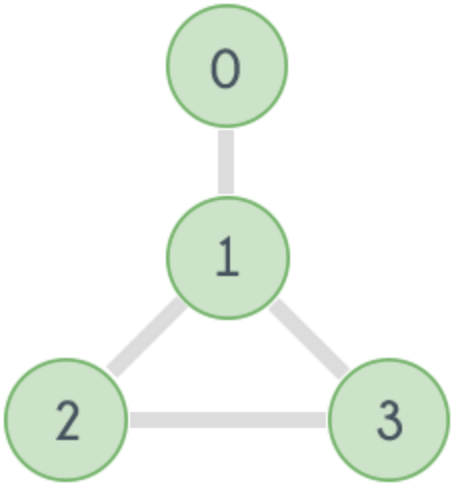


Fig. 2

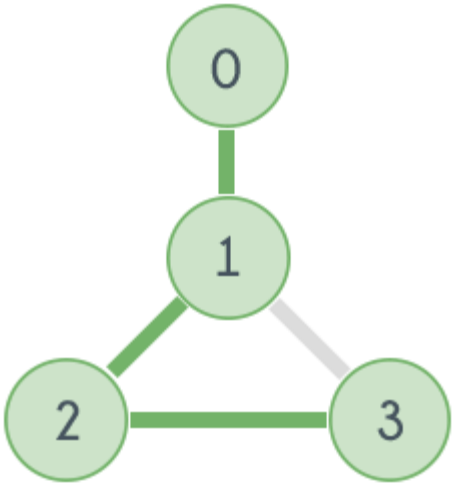


Fig. 3

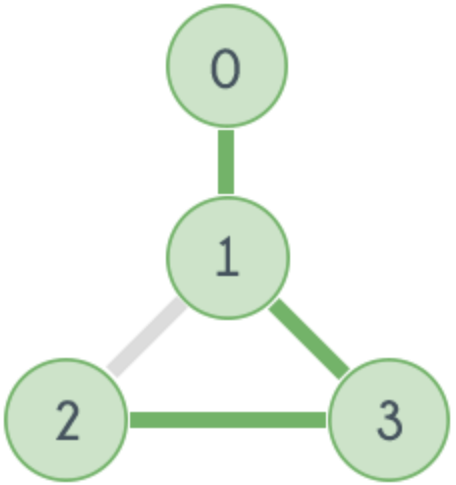


Fig. 4

Hamiltonian Path

- Time complexity of the following method can be easily derived.
- For a graph having N vertices it visits all the permutations of the vertices, i.e. N! iterations and in each of those iterations it traverses the permutation to see if adjacent vertices are connected or not i.e. N iterations, so the complexity is **O(N * N!).**

```
function check_all_permutations(adj[][], n)
    for i = 0 to n
        p[i]=i
    while next permutation is possible
        valid = true
        for i = 0 to n-1
            if adj[p[i]][p[i+1]] == false
                valid = false
                break
        if valid == true
            return true
        p = get_next_permutation(p)
    return false
```

Hamiltonian Path: The the C++ implementation

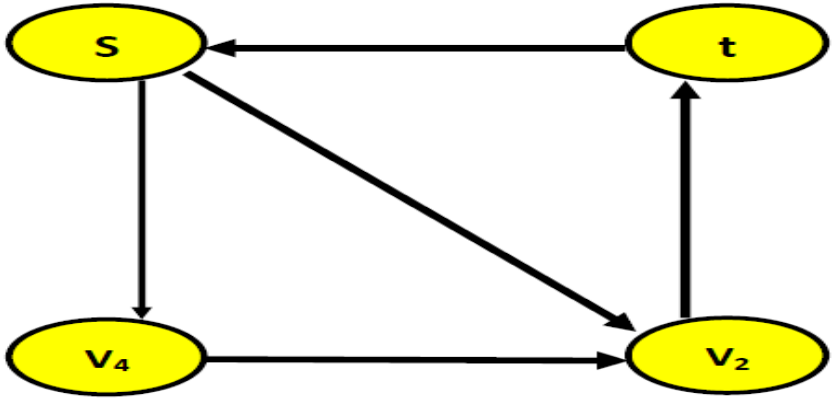
```
bool check_all_permutations(bool adj[][MAXN], int n){
        vector<int>v;
        for(int i=0; i<n; i++)
            v.push_back(i);
        do{
            bool valid=true;
            for(int i=0; i<v.size()-1; i++){
                if(adj[v[i]][v[i+1]] == false){
                    valid=false;
                    break;
                }


            }
            if(valid)
                return true;
        }while(next_permutation(v.begin(), v.end()));
        return false;
}
```

Theorem: The Hamiltonian path (HP) Problem is NP-Complete

Proof:

Step 1: Given a graph G=(V, E) and two distinguished vertices s and t.

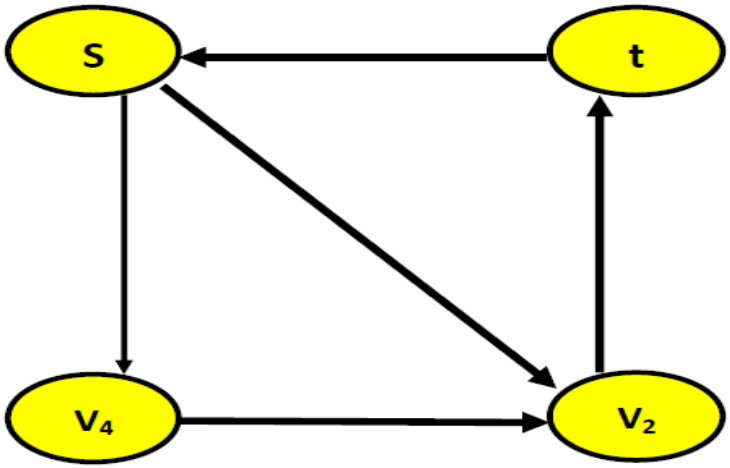
- Hamiltonian path problem: Finding a simple path in G with end points **s** and **t**, that passes all other vertices .
- We have to prove that this problem is belongs to the NP class of problem.
- Given a path, we have to verify whether the path begins at s and end at t and all other vertices of V appears only once on path.
- This verification is possible in polynomial time



G= (V, E)

The Hamiltonian path (HP) Problem is NP-Complete

Step 2: Let us select **Directed Hamiltonian Path** (DHP) problem as the NP-complete problem , which is to be reduced to the **Hamiltonian Path Problem.**

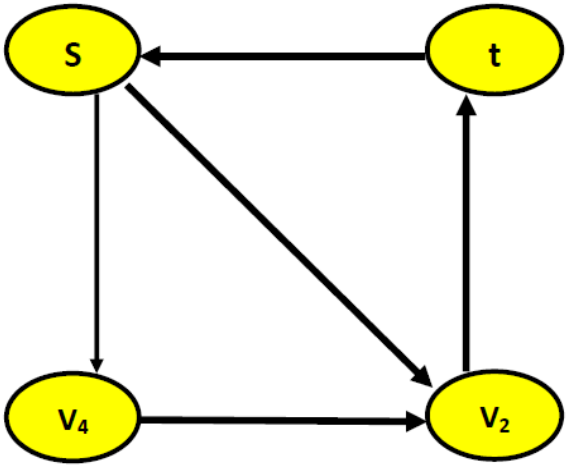


G= (V, E)

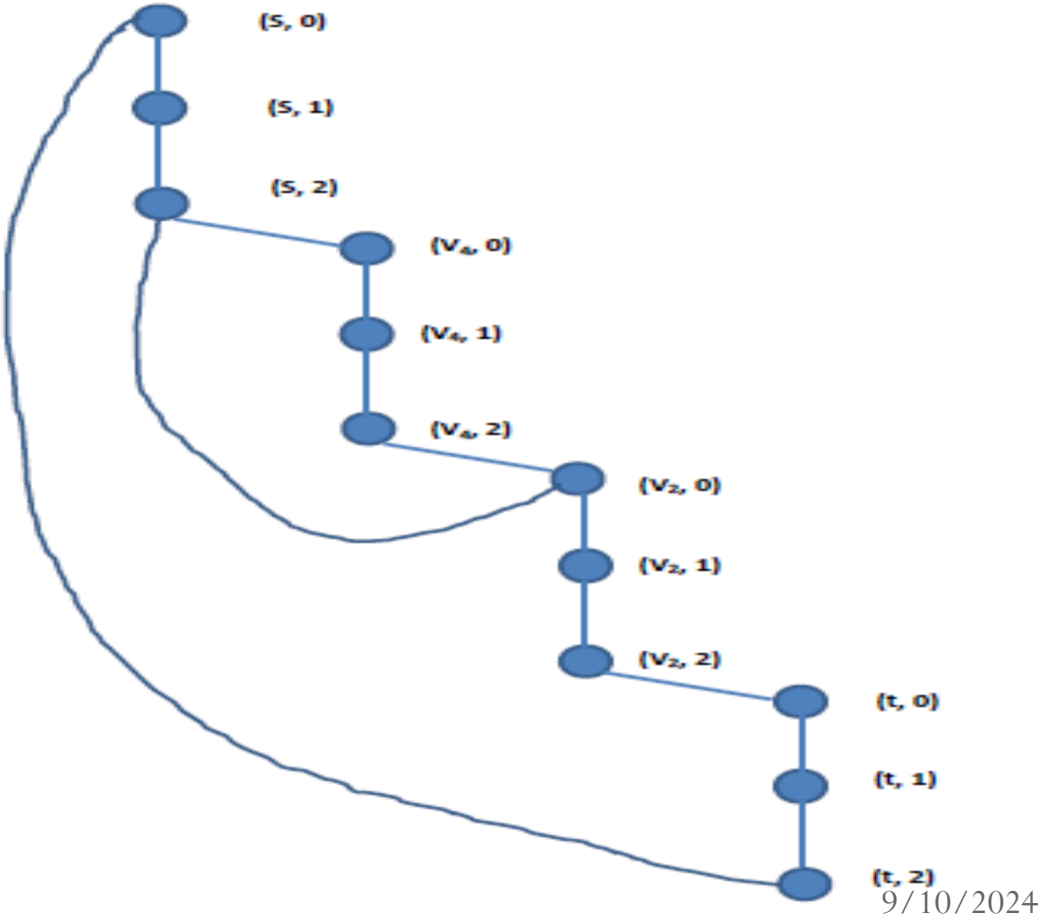
Step 3: Reduction step: To design an algorithm that reduces DHP problem to HP problem

The Hamiltonian path (HP) Problem is NP-Complete

- Let us take an instance I of DHP, that is consists of the digraph *G(V, E) and* two vertices s & t
- Let us define and instance f(I) as G' = (V', E')

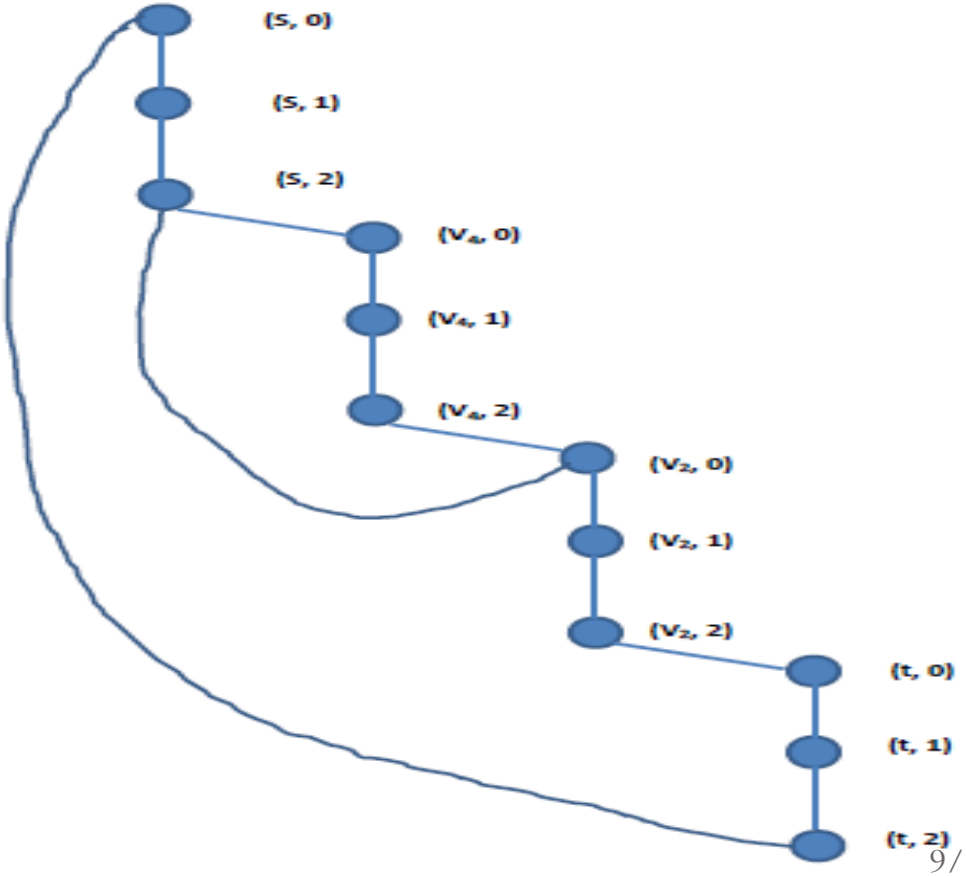


G= (V, E)



The Hamiltonian path (HP) Problem is NP-Complete

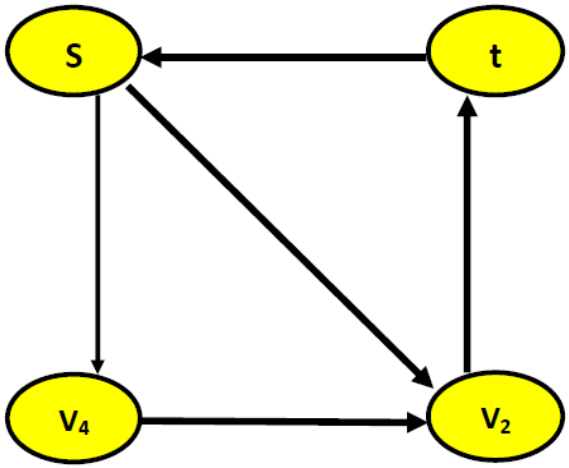
- Let us define and instance f(I) as G' = (**V'**, **E'**)
- Where **V'** = {(v,0), (v,1), (v,2) | v ∈ V} and
- **E'** = { **((v,0), (v,1)), ((v,1), (v,2)) | v ∈ V} ∪ {((u, 2), (v,0)) | (u,v) ∈ E}**



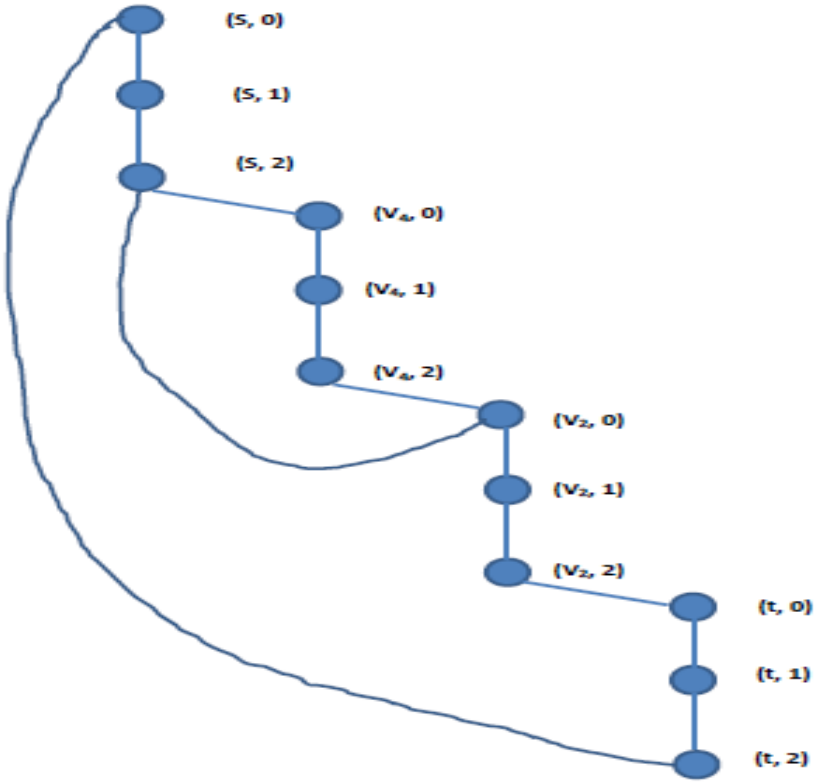
The two end points are (s,0) and (t,2)

The Hamiltonian path (HP) Problem is NP-Complete

- There is a directed Hamiltonian path **P** from **s** to **t** in **G** , and a Hamiltonian path **P'** from **(s,0)** to **(t,2)** in **G'**.
- **P'** consists of all the edges in the first part definition of **E' ,** and if (u, v) is in **P**, then ((u,2), (v,0)) is included in **P'**



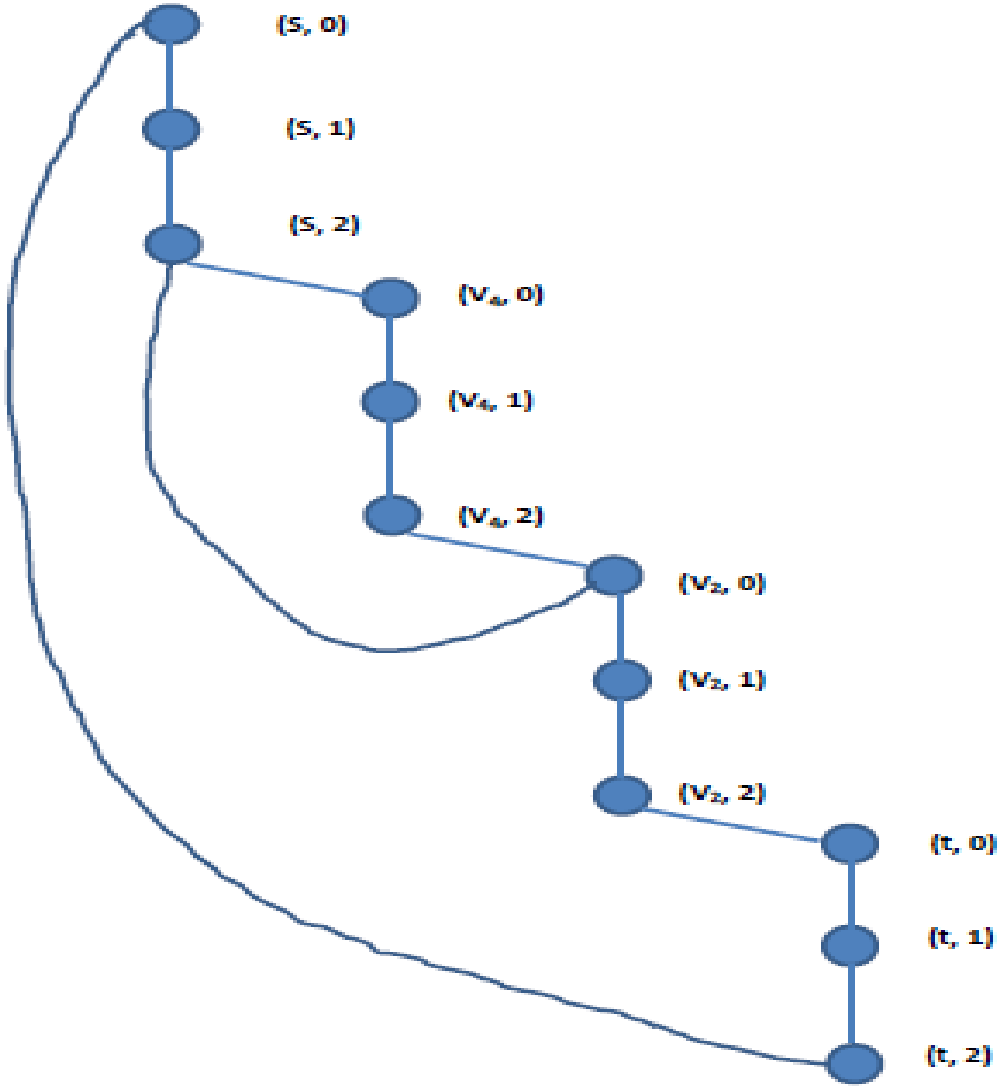
G= (V, E)



The Hamiltonian path (HP) Problem is NP-Complete

- Now assume that P' is a solution to the Hamiltonian Path Problem specified by f(I). Clearly al the edges of the first part of the defination of E' appears in P' or there would not be any way to pass through some (v, 1).
- Now by considering path P' , that starts with ((s,0), (s,1)), ((s,1),(s,2)) and ends with ((t, 0), (t,1)), ((t,1),(t,2)).
- Whenever ((u,2), (v,0)) is used in P', we can use (u,v) in G.
- Thus, the resulting directed path is simple, starts at s and ends at t and passes through all other vertices of G.
- ***Step 4:*** The reduction process in step-3 can be carried out in polynomial time. Hence the **Hamiltonian path** problem is NP-complete.□

The Hamiltonian path (HP) Problem is NP-Complete

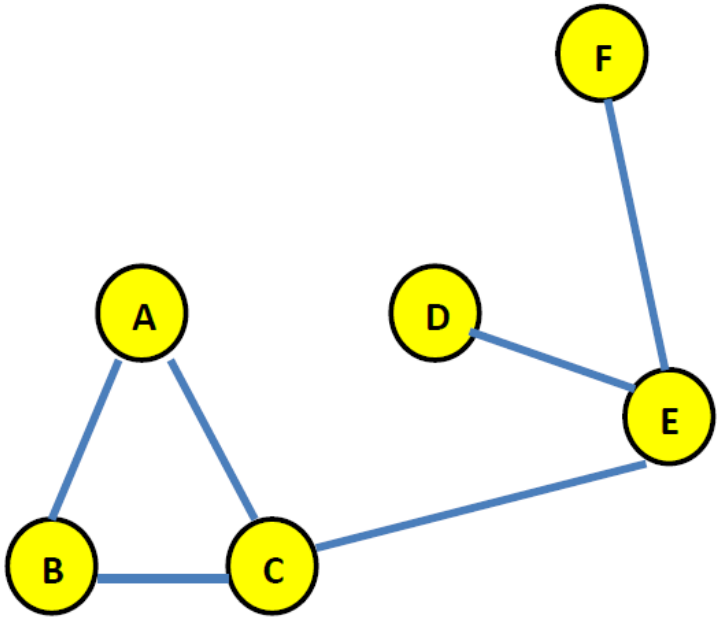


The vertex-cover problem is NP-complete

Cliques in Graph

- A clique in an undirected graph is a complete subgraph of the given graph.
- A complete sub-graph is one in which all of its vertices are linked to all of its other vertices.
- A **clique**, **CG**, in an undirected graph G = (V, E) is **a subset of the vertices, X ⊆ V, such that every two distinct vertices are adjacent**.

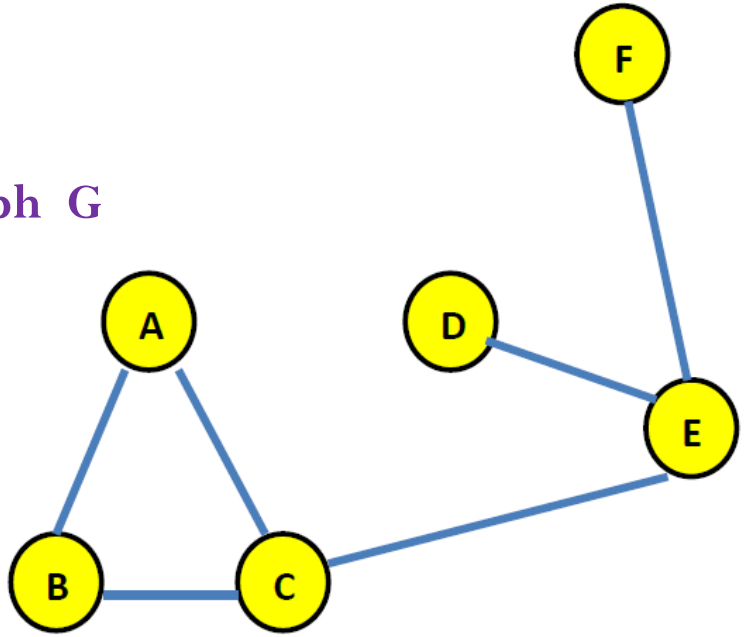
X={A,B,C}

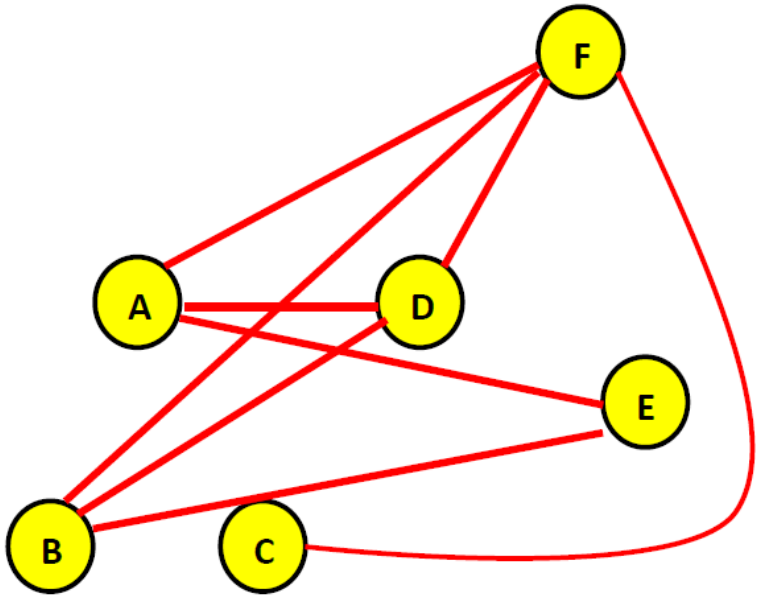


The complement of a graph: CG

- An undirected graph is described by a pair (V, E) where V is set of vertices and E,
- If $G = (V, E)$ is an undirected graph, the **complement of graph G** is denoted as CG and defined as $CG = (V, E')$, where E' is set of all edges $\{<u,v>) | <u,v>$ is not an edge of $G(V,E)$, *i.e.* $<u,v> \notin E\}$

Graph G

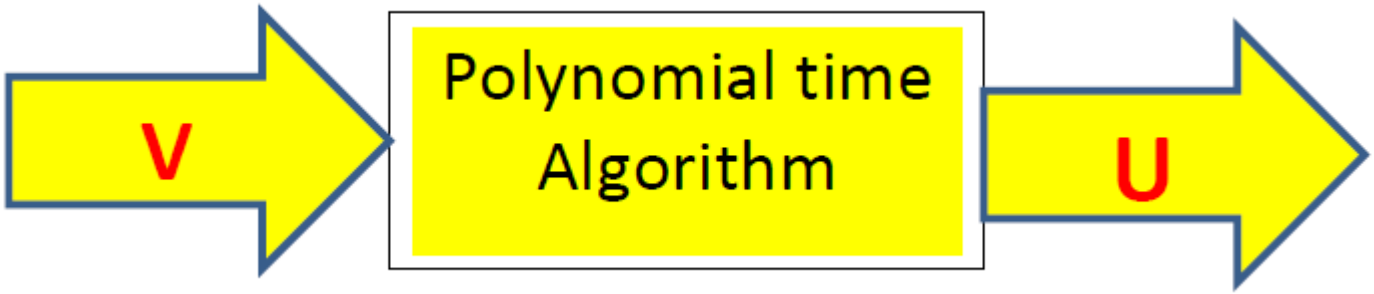




Complement of Graph G

How do we prove a problem is NP complete

- Given a problem U, the steps involved in proving that it is **NP-complete** are the following:
- Step 1: Show that U is in NP.
- Step 2: Select a **known NP-complete** problem V.
- Step 3: Construct a **reduction** from **V** to **U**.
- Step 4: Show that the reduction requires **polynomial time.**



The vertex-cover problem is NP-complete

Theorem: The vertex-cover problem is NP-complete

Proof:

Step 1: Let consider a graph G(V, E) and an integer k.

Let X be the vertex cover of G; $X \subseteq V$, which is determined non-deterministically.

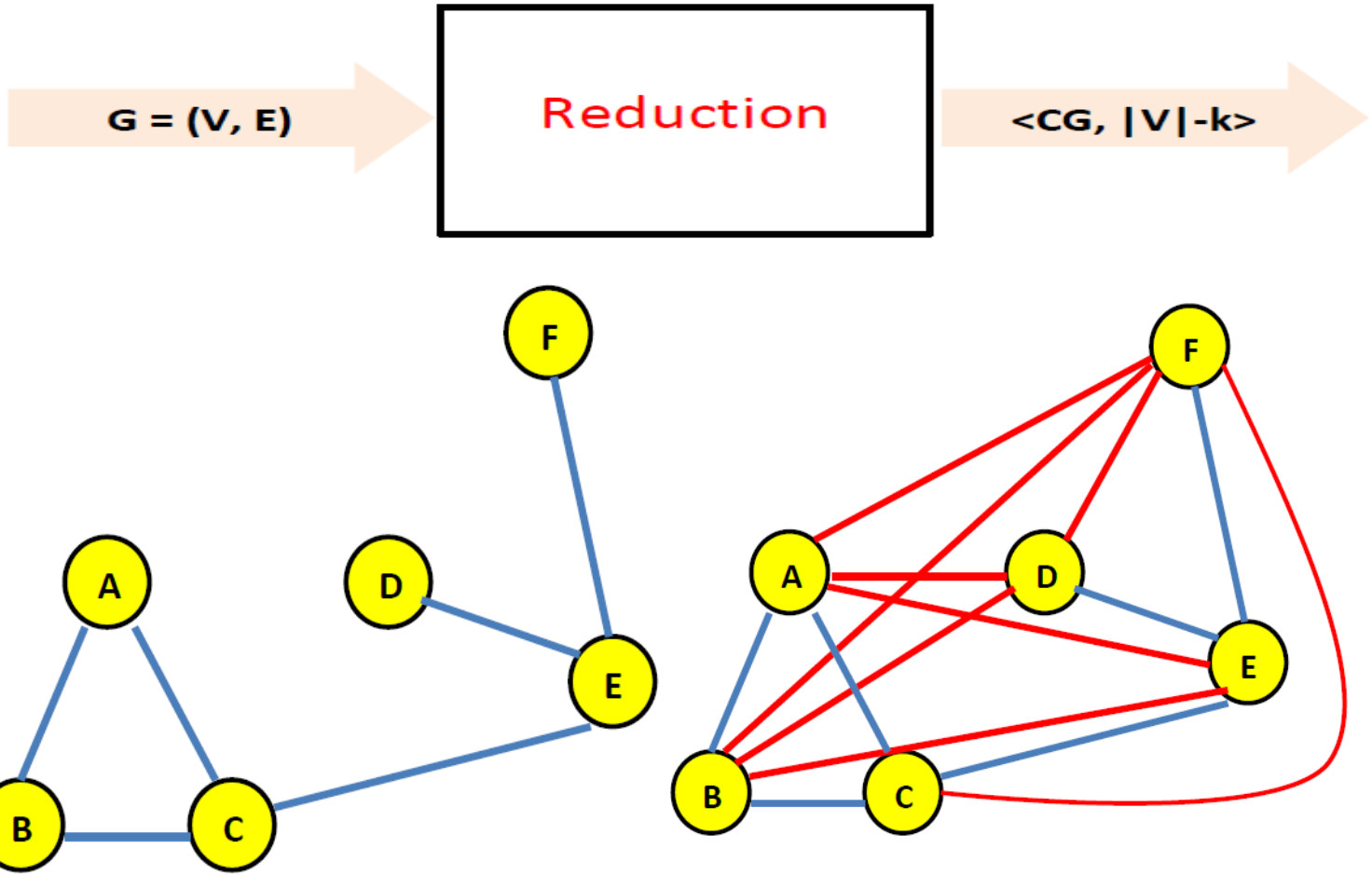
The verification algorithm checks (i) whether $|X| \leq k$, (ii) for each edge $(u,v) \in E$, whether $u \in X$ or $v \in X$.

This can be checked by an non-deterministic algorithm in polynomial time.

Step 2: Let Clique is an NP-complete problem. This implies the NP-complete problem **Clique** to be reduced to **vertex-cover problem**.

The vertex-cover problem is NP-complete

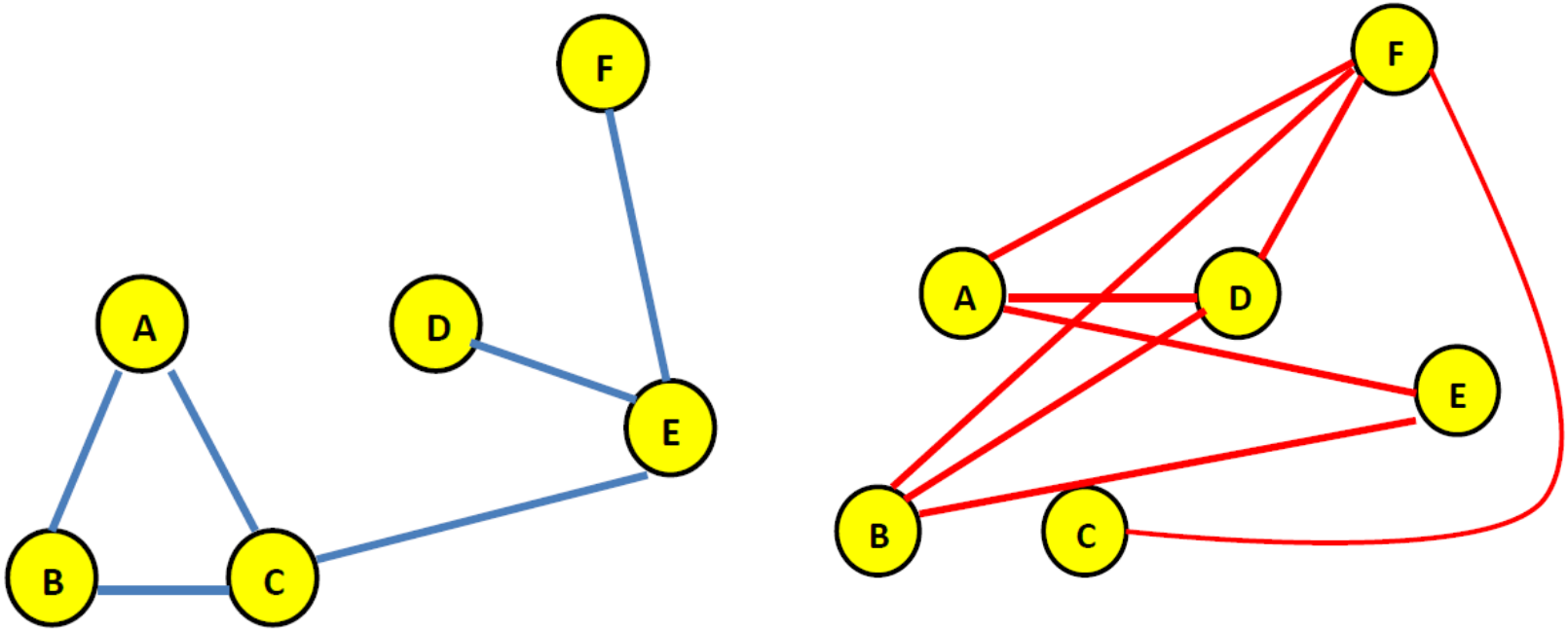
- *Step 3:* ***Reduction Algorithm***



The vertex-cover problem is NP-complete

Step 3: ***Reduction Algorithm***

- Let G(V,E) is an undirected graph with |V| = n vertices. The complement of graph G is denoted as CG, and defined as follows:
- CG = (*V*, *E'*), where E' is set of all edges {<u,v>)|<u,v> is not an edge of *G(V,E), i.e.* <u,v> ∉ E }



The vertex-cover problem is NP-complete

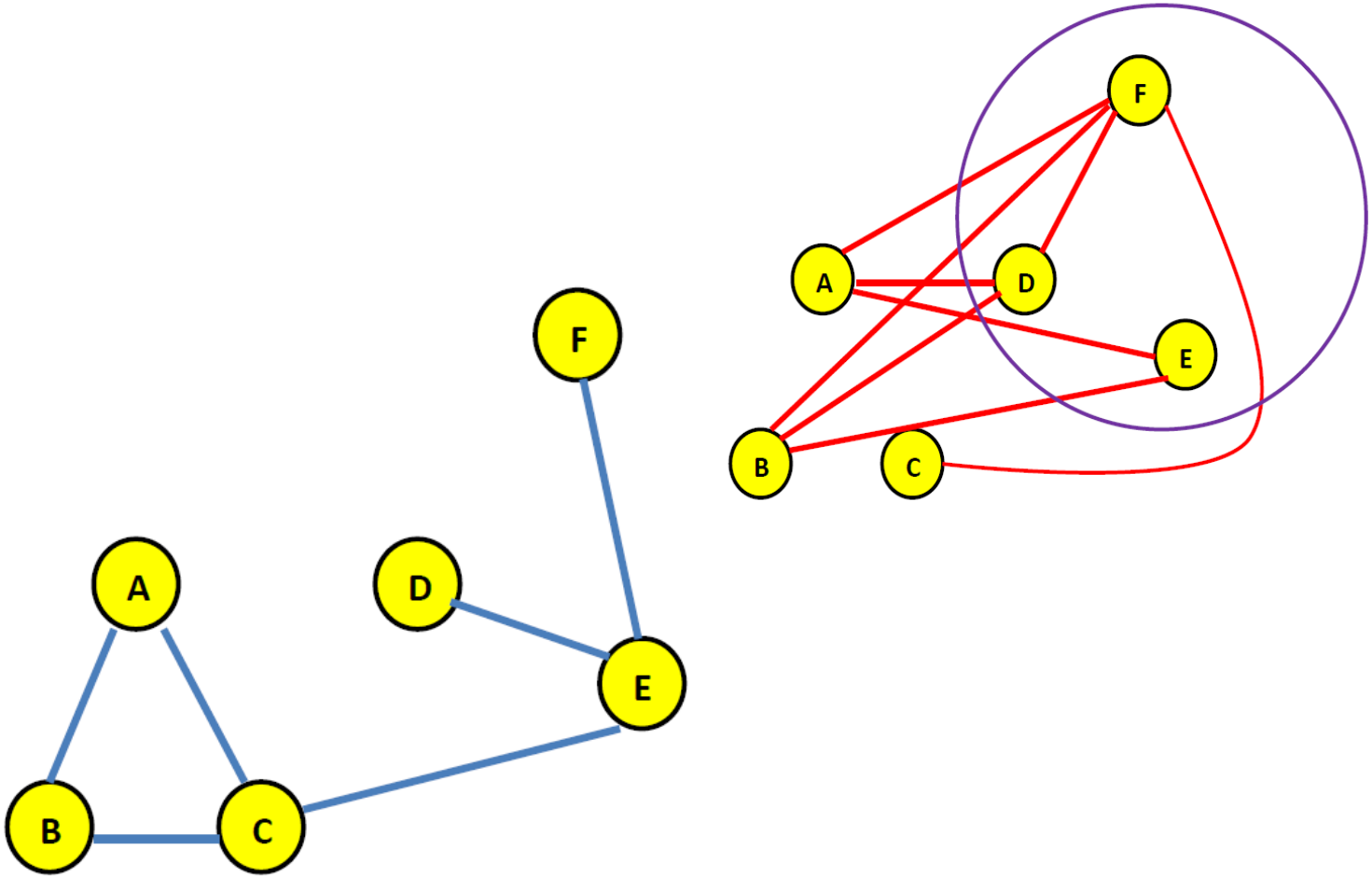
- Let us take an instance <G, k> of the given **clique** problems as input and computes its corresponding CG.
- The output of the reduction algorithm is <CG, |V|- k> of the vertex-cover problem.

- This is to be proved that the creation of <CG, |V|- k> is a reduction.
- That is to prove G=(V, E) has a clique of size k if and only if the graph CG has a vertex cover of size **|V|- k.**

The vertex-cover problem is NP-complete

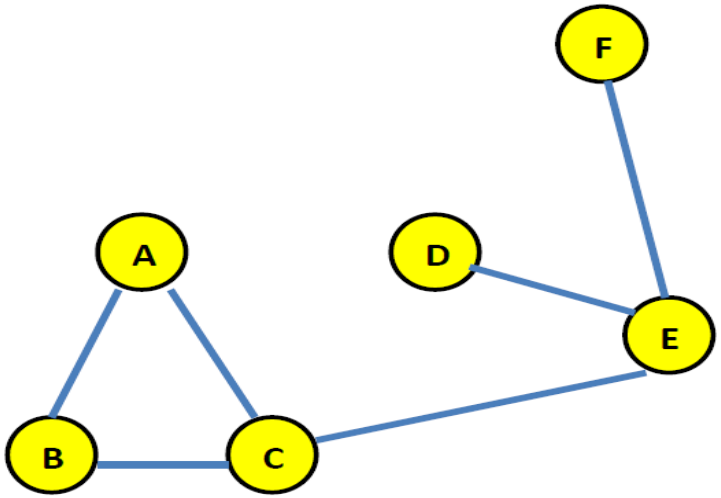
- Let us assume that the graph G has a clique X of size k. It is claimed that V-X is a vertex-cover in CG.
- If edge (u, v) $\in$ E', then (u, v) $\notin$ E and this implies that at least one of the u or v does not belong to X, since every pair of vertex in X is connected by an edge of E.
- At least one of u or v is in V-X, and this means the edge (u, v) is covered by V-X.
- Since (u, v) is chosen arbitrarily from E' , every edge of E' is covered by a vertex V-X.
- Hence , the set V-X having size $|\mathbf{V}| - \mathbf{k}$ from a vertex-cover for CG

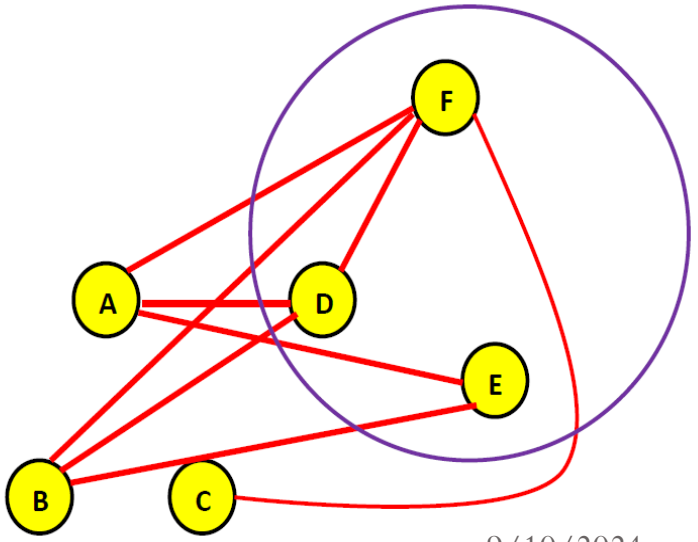
The vertex-cover problem is NP-complete



The vertex-cover problem is NP-complete

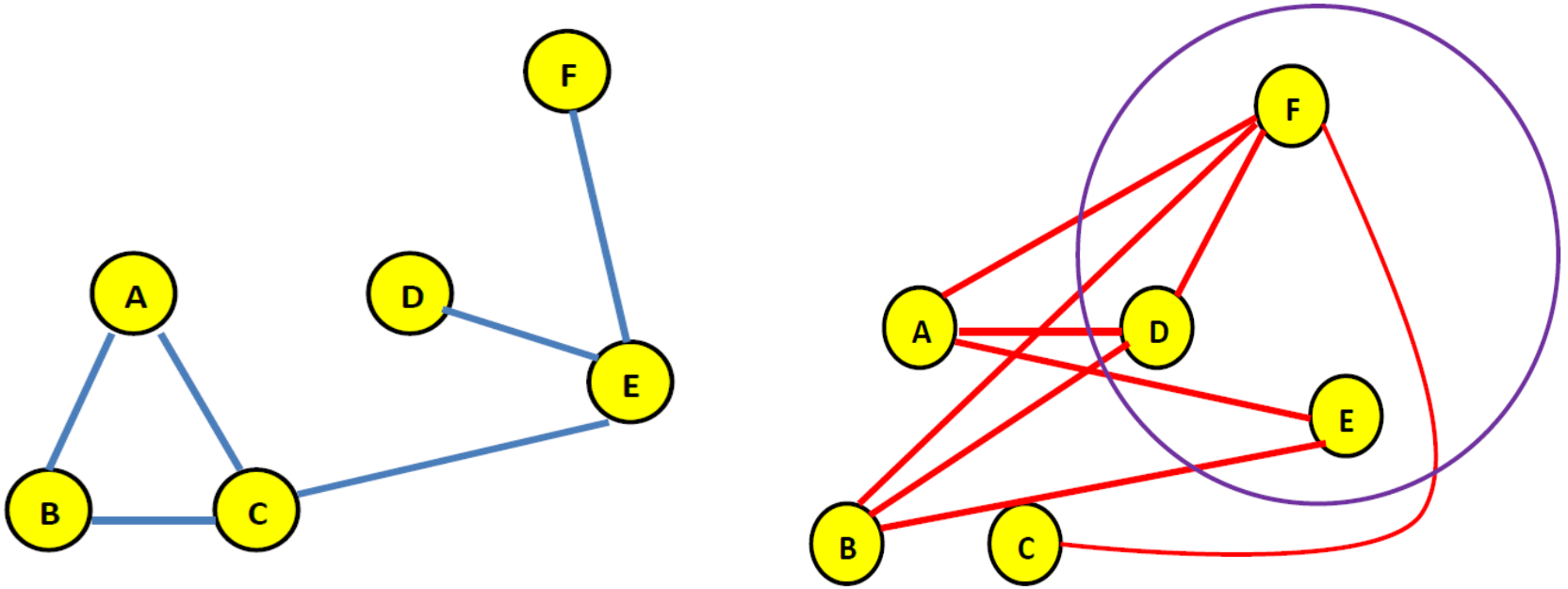
- Now to prove in the reverse direction.
- Suppose CG has a vertex-cover $VC \subseteq V$, where $|VC| = |V| - k$. For all $u, v \in V$, if $\langle u, v \rangle \in E'$, than $u \in VC$ or $v \in VC$ or both.
- The **contrapositive** of this is that for all $u, v \in V$, if $u \notin VC$ and $v \notin VC$, then $\langle u, v \rangle \in E$.
- In other words, V-VC is a clique, and it has size $|V| - |VC| = k$





The vertex-cover problem is NP-complete

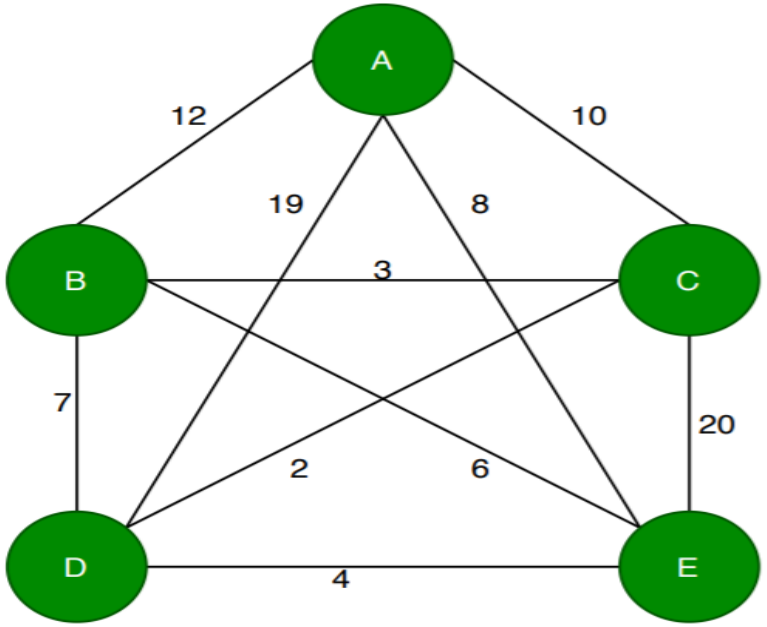
- *Step 4*: The above reduction process can be carried out in polynomial time. Hence the vertex cover problem is NP-Complete.□



TSP is NP-Complete

The Travelling Salesman Problem (TSP)

- Suppose a salesman needs to give sales pitches in four cities. He looks up the airfares between each city, and puts the costs in a graph. In what order should he travel to visit each city once then return home with the lowest cost?
- To answer this question of how to find the lowest cost **Hamiltonian circuit**, we will consider some possible approaches. The first option that might come to mind is to just try all different possible circuits.



Unique Hamiltonian Circuits

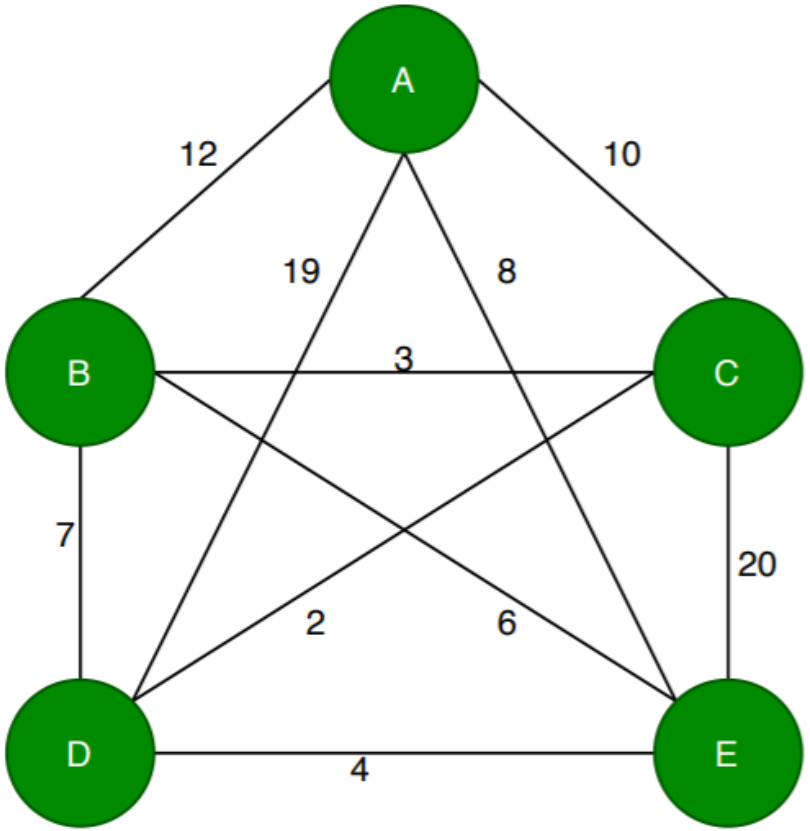
NUMBER OF POSSIBLE CIRCUITS

- For *n* vertices in a **complete graph**, there will be $(n-1)!=(n-1)(n-2)(n-3)\ldots 3\cdot 2\cdot 1(n-1)!=(n-1)(n-2)(n-3)\ldots 3\cdot 2\cdot 1$ routes.
- Half of these are duplicates in reverse order, so there are $(n-1)!/2$ unique circuits.
- The exclamation symbol, !, is read "factorial" and is shorthand for the pro

Cities	Unique Hamiltonian Circuits
9	8!/2 = 20,160
10	9!/2 = 181,440
11	10!/2 = 1,814,400
15	14!/2 = 43,589,145,600
20	19!/2 = 60,822,550,204,416,000

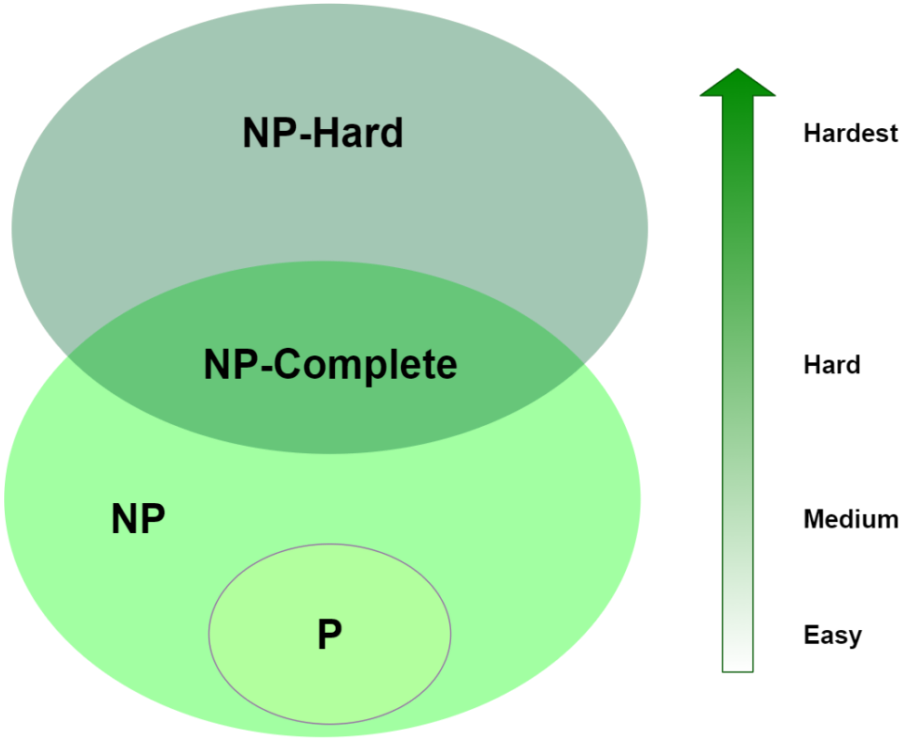
The Travelling Salesman Problem (TSP)

- Given a complete Graph G with 5nodes, is having (n−1)!/2 = 4!/ = 12 unique Hamiltonian Circuits



The Travelling Salesman Problem (TSP)

- TSP belongs to the class of combinatorial optimization problems known as **NP-complete**. This means that TSP is classified as **NP-hard** because it has no "quick" solution and the complexity of calculating the best route will increase when you add more destinations to the problem.



The Travelling Salesman Problem (TSP)

- The problem can be solved by analyzing every round-trip route to determine the shortest one. However, as the number of destinations increases, the corresponding number of roundtrips surpasses the capabilities of even the fastest computers.
- With 10 destinations, there can be more than **300,000** roundtrip permutations and combinations.
- With 15 destinations, the number of possible routes could exceed 87 billion.
- In the dynamic algorithm for TSP, the number of possible subsets can be at most $N \times 2^N$. Each subset can be solved in $O(N)$ times. **Therefore, the time complexity of this algorithm would be** $O(N^2 \times 2^N)$.

Prove that Travelling Salesman Problem (TSP) NP Complete

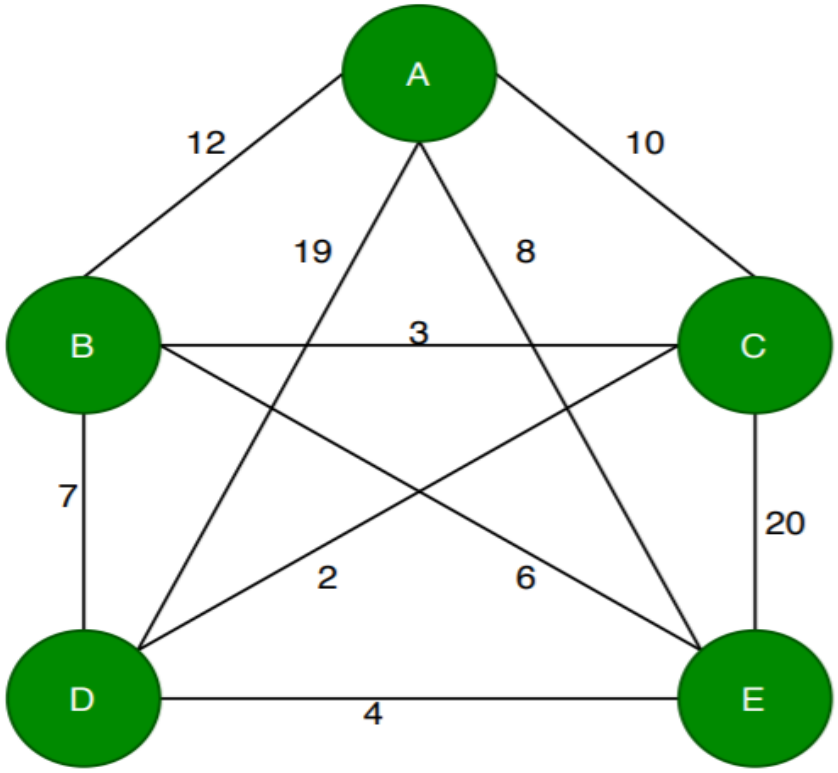
The proof that the Traveling Salesman Problem (TSP) is NP-complete involves two main steps: showing that TSP is in NP, and then showing it is NP-hard.

TSP is in NP:

- For a decision problem to be in NP (Non-deterministic Polynomial time), there must be a way to verify a given solution in polynomial time.
- In the case of TSP, given a sequence of cities and a proposed total distance, we can easily check in polynomial time whether the sequence forms a valid tour (i.e., visits each city exactly once and returns to the starting city) and whether the total distance of this tour is less than or equal to the proposed distance.

Prove that Travelling Salesman Problem (TSP) NP Complete

- A sequence cites (A, D, B, E, C)forms a valid tour can be verified in polynomial time



Prove that Travelling Salesman Problem (TSP) NP Complete

TSP is NP-hard:

- To show that a problem is NP-hard, it needs to be at least as hard as any problem in NP. This is typically done by a polynomial-time reduction from a known NP-hard problem to the problem in question.
- One common approach is to reduce from the Hamiltonian Cycle problem, which is known to be NP-complete. The Hamiltonian Cycle problem asks whether there is a cycle in a given graph that visits each vertex exactly once.

Prove that Travelling Salesman Problem (TSP) NP Complete

- The reduction works as follows:Given a graph for the Hamiltonian Cycle problem, construct a graph for TSP where each edge that was in the original graph has a weight of 1, and each edge that was not in the original graph has a weight of 2 (or any large number).
- Now, if the original graph has a Hamiltonian Cycle, there will be a tour in the constructed graph with total weight equal to the number of vertices (since each edge in the cycle has weight 1).
- Conversely, if the constructed graph has a tour with a total weight equal to the number of vertices, then the original graph must have a Hamiltonian Cycle (since any tour with a larger weight must use an edge not in the original graph).

Prove that Travelling Salesman Problem (TSP) NP Complete

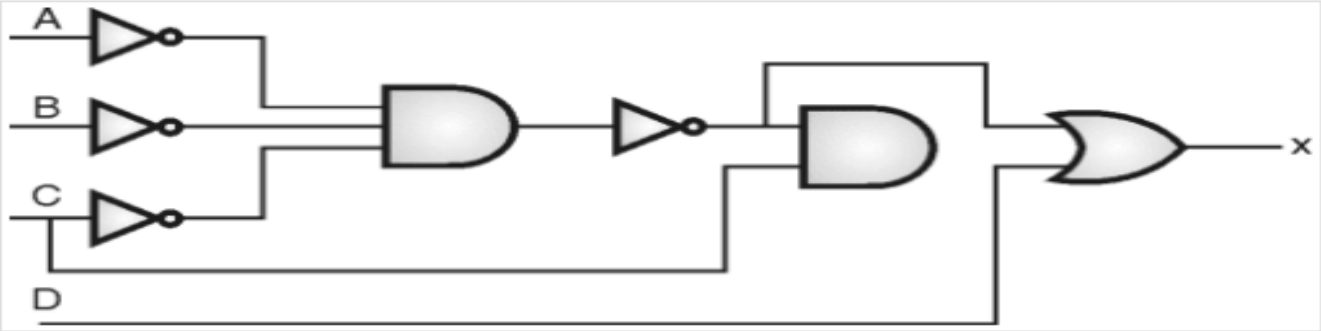
- Since we can transform any instance of the Hamiltonian Cycle problem into an instance of TSP in polynomial time, and the existence of a solution in one implies a solution in the other, this reduction shows that TSP is at least as hard as Hamiltonian Cycle, and thus is NP-hard.

- By showing that TSP is both in NP and NP-hard, we conclude that TSP is NP-complete. This means that there is no known polynomial-time algorithm to solve all instances of the TSP unless P=NP, and verifying a given solution can be done in polynomial time.□

3-SAT (3-Satisfiability Problem) is NP-complete

Circuit Satisfiability (CIRCUIT-SAT) Problem

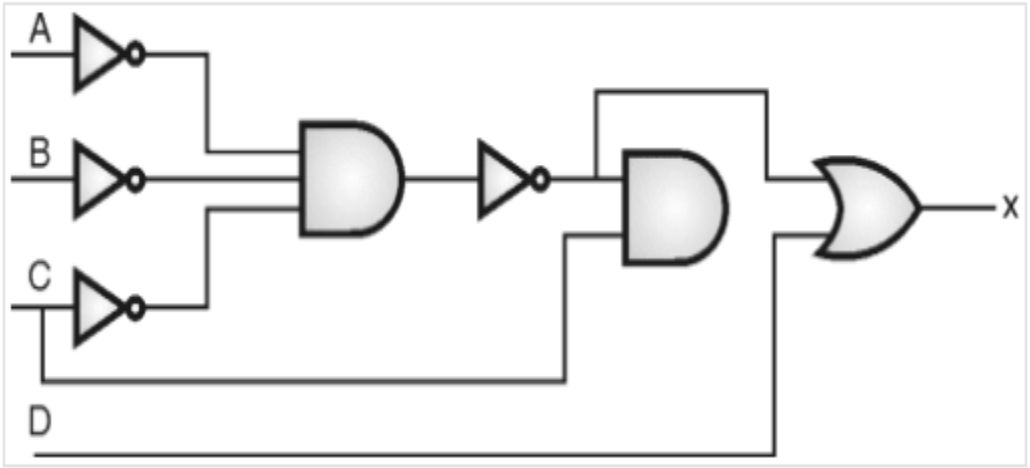
- *A combinational circuit is a collection of AND, OR and NOT gates.* ***Circuit satisfiability problem*** *is to determine, given the combinational circuit, is it satisfiable?*
- Given the assignment of the Boolean value for the input variable of the circuit, we can trace the output by assigning 0/1 to each input.
- If the final value of the circuit results in 1, the algorithm outputs "yes", otherwise algorithm outputs "no".
- This checking is done in polynomial time, more formally in the linear order of circuit size.



Satisfiability Problem

Circuit Satisfiability (CIRCUIT-SAT) Problem

- If the circuit has k inputs, 2^k different assignments are possible. If the size of the circuit is polynomial in k, each assignment leads to a super-polynomial time algorithm.
- This proves that the polynomial time algorithm does not exist for circuit satisfiability problems and hence it is in NP.



Satisfiability Problem

SAT(Boolean Satisfiability Problem)

- **SAT Problem:** SAT(Boolean Satisfiability Problem) is the problem of determining if there exists an interpretation that satisfies a given boolean formula.
- It asks whether the variables of a given boolean formula can be consistently replaced by the values **TRUE or FALSE** in such a way that the formula evaluates to **TRUE**.
- If this is the case, the formula is called *satisfiable*. On the other hand, if no such assignment exists, the function expressed by the formula is **FALSE** for all possible variable assignments and the formula is *unsatisfiable*.
- A given boolean expression, determining if there exists a truth assignment to its variables for which the expression is *true* is defined as the satisfiability problem.

SAT(Boolean Satisfiability Problem)

a	b	c	$(a \wedge b) \vee c$	$(a \wedge \neg a) \vee (c \wedge \neg c)$
1	1	1	1	0
0	1	1	1	0
0	0	1	1	0
0	1	0	0	0
1	0	1	1	0
1	0	0	0	0
1	0	1	1	0
0	0	0	0	0

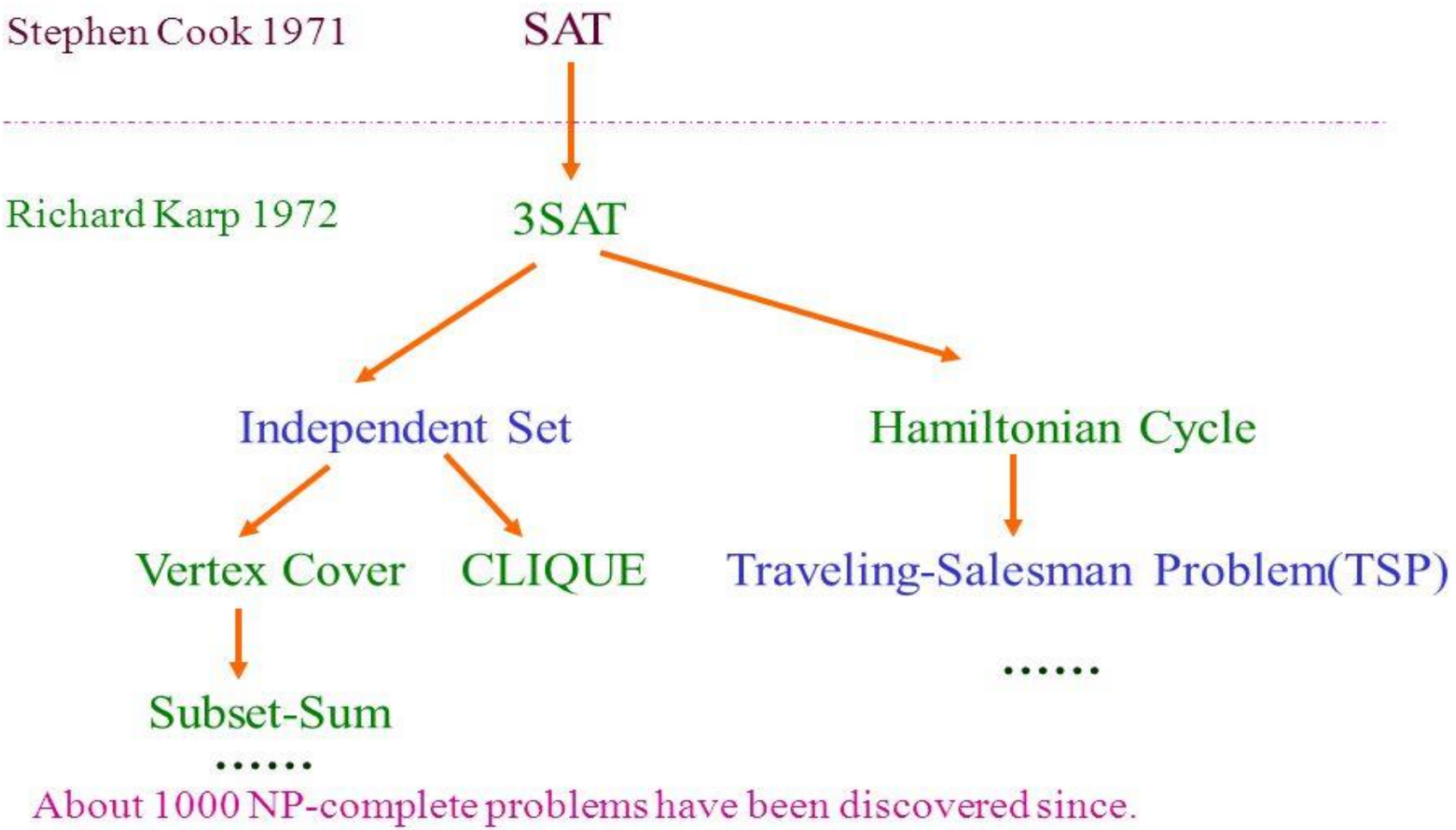
- Let there are two expressions:

$$(a \wedge b) \vee c$$

$$(a \wedge \neg a) \vee (c \wedge \neg c)$$

- There are 5 satisfying truth assignments for the expression (i) , hence it is **satisfiable**.

NP-complete Problems



About 1000 NP-complete problems have been discovered since.

SAT(Boolean Satisfiability Problem)

- To test whether there exists a **truth assignment** to variables a, b, c, we've tested all possible assignments.
- There are 5 satisfying truth assignments for the expression above and hence it is **satisfiable.**
- But the expression is *false* for all possible truth assignments and hence it is **unsatisfiable**.
- If there are n variables in the boolean expression, then **the truth table method** would take time to determine if there is a truth assignment that makes the expression *true*.
- This takes $\mathbf{O(2^n)}$. This is called the **truth table method**. If there are **n** variables in the boolean expression, then the truth table method would take $\mathbf{O(2^n)}$ time to determine if there is a truth assignment that makes the expression *true*.

3-SAT (3-Satisfiability Problem) is NP-complete

- The term NP-complete refers to the fact that 3-SAT is among the hardest problems in NP (nondeterministic polynomial time) in the sense that any problem in NP can be reduced to it in polynomial time.
- The proof that 3-SAT (3-Satisfiability Problem) is NP-complete is a fundamental result in computational complexity theory and involves two main components:
- (i) demonstrating that 3-SAT is in NP, and
- (ii) showing that it is **NP-hard**.

- https://www.baeldung.com/cs/cook-levin-theorem-3sat

3-SAT (3-Satisfiability Problem) is NP-complete

3-SAT is in NP:

- A problem is in NP if you can verify a solution in polynomial time.
- For 3-SAT, given a set of variables and a Boolean formula in conjunctive normal form where each clause has exactly three literals (variables or their negations), if you're given a truth assignment for the variables, you can easily check if this assignment satisfies the formula in polynomial time. Simply substitute the truth values into the formula and evaluate it.

3-SAT (3-Satisfiability Problem) is NP-complete

3-SAT is NP-hard:

- A problem is NP-hard if every problem in NP can be reduced to it in polynomial time.
- The proof that 3-SAT is NP-hard typically involves a reduction from another NP-complete problem, such as SAT or the Clique problem.
- The most common reduction is from SAT (Satisfiability Problem) to 3-SAT.
- Since SAT was the first problem to be proven NP-complete (by Stephen Cook), showing that SAT reduces to 3-SAT would prove 3-SAT is NP-hard.

3-SAT (3-Satisfiability Problem) is NP-complete

3-SAT is NP-hard:

- The reduction works by taking a general Boolean formula (which may have any number of literals in its clauses) and transforming it into one that has exactly three literals per clause.
- This transformation is done in such a way that the original formula is satisfiable if and only if the transformed formula is satisfiable.
- This transformation can be done in polynomial time.
- Once you have established that 3-SAT is in NP and that it is NP-hard, by definition, it is NP-complete □.

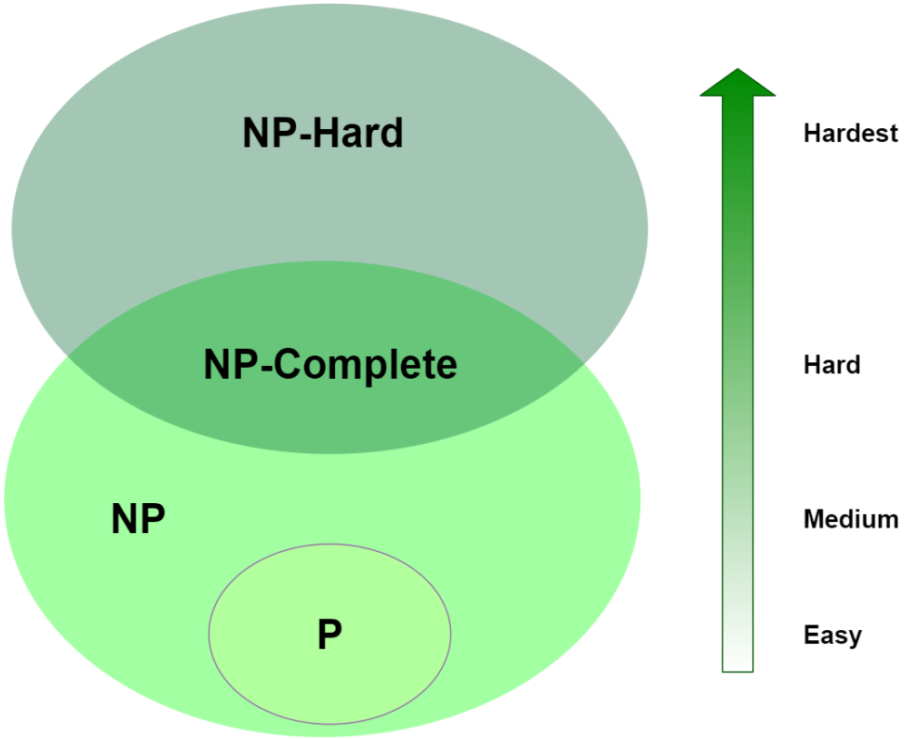
0/1 Knapsack Problem NP Hard

The 0/1 Knapsack problem itself is NP-hard

- 0/1 Knapsack Problem NP Complete." This is a common misconception.
- **The 0/1 Knapsack problem itself is NP-hard**, not NP-complete.
- **NP-hard:** A problem is NP-hard if it is at least as hard as the hardest problems in NP. NP-hard problems may not necessarily be in NP (that is, their solution may not be verifiable in polynomial time), and they include decision problems, optimization problems, and search problems.
- **NP-complete:** A problem is NP-complete if it is in NP and it is as hard as any problem in NP. NP-complete problems are a subset of NP problems that are both in NP and NP-hard.

The 0/1 Knapsack problem itself is NP-hard

- **NP-complete:** A problem is NP-complete if it is in NP and it is as hard as any problem in NP.
- NP-complete problems are a subset of NP problems that are both in NP and NP-hard.



The 0/1 Knapsack problem itself is NP-hard

- The decision version of the Knapsack problem (deciding whether there is a subset of items that fits exactly into the knapsack without exceeding its capacity) is NP-complete because it is in NP (you can verify a solution quickly by checking the total weight and value of the chosen subset of items) and it's NP-hard.
- However, when we refer to the "0/1 Knapsack Problem" without further context, it usually means the optimization problem where the goal is to maximize the value within the knapsack capacity, and this version is not NP-complete because it is **not a decision problem**; it is NP-hard.

Proof: The 0/1 Knapsack problem itself is NP-hard

- The proof of NP-hardness for the 0/1 Knapsack problem typically involves a reduction from another NP-complete problem.
- Let us use the Subset Sum problem, which is known to be **NP-complete**.

Proof:

Subset Sum Problem: The Subset Sum problem asks whether a given set of integers has a subset whose sum is exactly equal to a specified sum, *S*. This problem is known to be NP-complete.

Subset Sum Problem

- Given a set of positive integers, and a value sum, determine that the sum of the subset of a given set is equal to the given sum.

OR

- Given an array of integers and a sum, the task is to have all subsets of given array with sum equal to the given sum.

Example:

Input: set[] = {2, 3, 5, 6, 8, 10}, sum = 10

Output = true

There are three possible subsets that have the sum equal to 10.

Subset1: {5, 2, 3}

Subset2: {2, 8}

Subset3: {10}

The 0/1 Knapsack problem itself is NP-hard

- **Reduction to 0/1 Knapsack:** To prove that the 0/1 Knapsack problem is NP-hard, we show that the Subset Sum problem can be reduced to it. We do this by constructing an instance of the 0/1 Knapsack problem from an instance of the Subset Sum problem.
- Suppose we have a Subset Sum problem with a set of integers $\{a_1, a_2, \ldots, a_n\}$ and a target sum S.
- We construct an instance of the 0/1 Knapsack problem by setting the weight and value of each item to be the integers a_i from the Subset Sum problem, and we set the knapsack's capacity to S.

The 0/1 Knapsack problem itself is NP-hard

- **Proof by Reduction:** The key point of the reduction is that any subset of items that fits exactly into the knapsack (i.e., the total weight is equal to the capacity *S*) will also have a total value that is equal to *S* because the weights and values of items are the same.
- Therefore, if there is a subset of items that maximizes the value without exceeding the capacity, it must also solve the Subset Sum problem.

The 0/1 Knapsack problem itself is NP-hard

- **Conclusion:** Because we can transform any instance of the Subset Sum problem into an instance of the 0/1 Knapsack problem in polynomial time, and a solution to the 0/1 Knapsack problem gives us a solution to the Subset Sum problem, we have shown that the 0/1 Knapsack problem is at least as hard as Subset Sum.
- Since Subset Sum is **NP-complete,** the 0/1 Knapsack problem is **NP-hard**.

Graph Coloring Problem is NP-Hard

Graph Coloring Problem: Given a graph and a number of colors, the task is to determine if the vertices of the graph can be colored with the given colors such that no two adjacent vertices share the same color.

Graph Coloring Problem is NP-Hard

- To prove that Graph Coloring is NP-hard, a common approach is to use a reduction from another NP-complete problem, such as the 3-SAT problem.

Overview of how the proof through reduction works:

Step-1:

Start with a known NP-complete problem: Select a known NP-complete problem, like 3-SAT, which is a problem of determining if there exists a truth assignment to variables in a Boolean formula in conjunctive normal form (with three literals per clause) that makes the formula true.

Graph Coloring Problem is NP-Hard

Step-2

Construct an instance of Graph Coloring from 3-SAT: Construct a graph in such a way that the graph can be colored with k colors if and only if the original 3-SAT formula is satisfiable.

The construction of the graph involves creating vertices and edges that correspond to the literals and clauses of the 3-SAT instance.

Graph Coloring Problem is NP-Hard

Step 3:

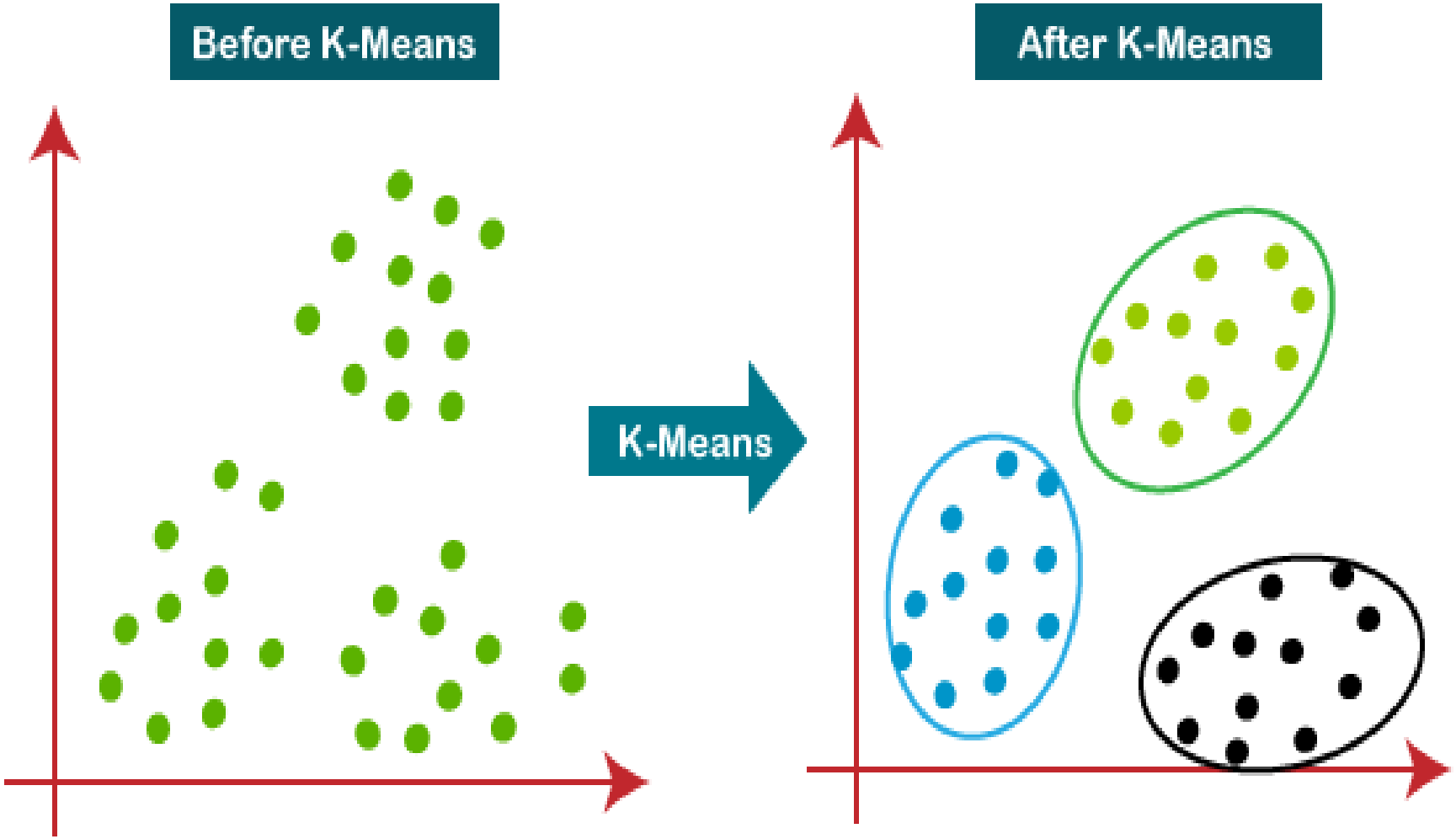
- **Reduction**: If one can show that any instance of the 3-SAT problem can be polynomially transformed into an instance of the Graph Coloring problem, and a k-coloring of this graph exists if and only if the original 3-SAT formula is satisfiable, then you have shown that Graph Coloring is at least as hard as 3-SAT. Since 3-SAT is NP-complete, this implies that Graph Coloring is NP-hard.
- The above is a simplified explanation. The actual construction of the graph from a 3-SAT instance is quite technical and involves creating gadgets that enforce the rules of 3-SAT within the structure of the graph.

K-Means is NP-hard

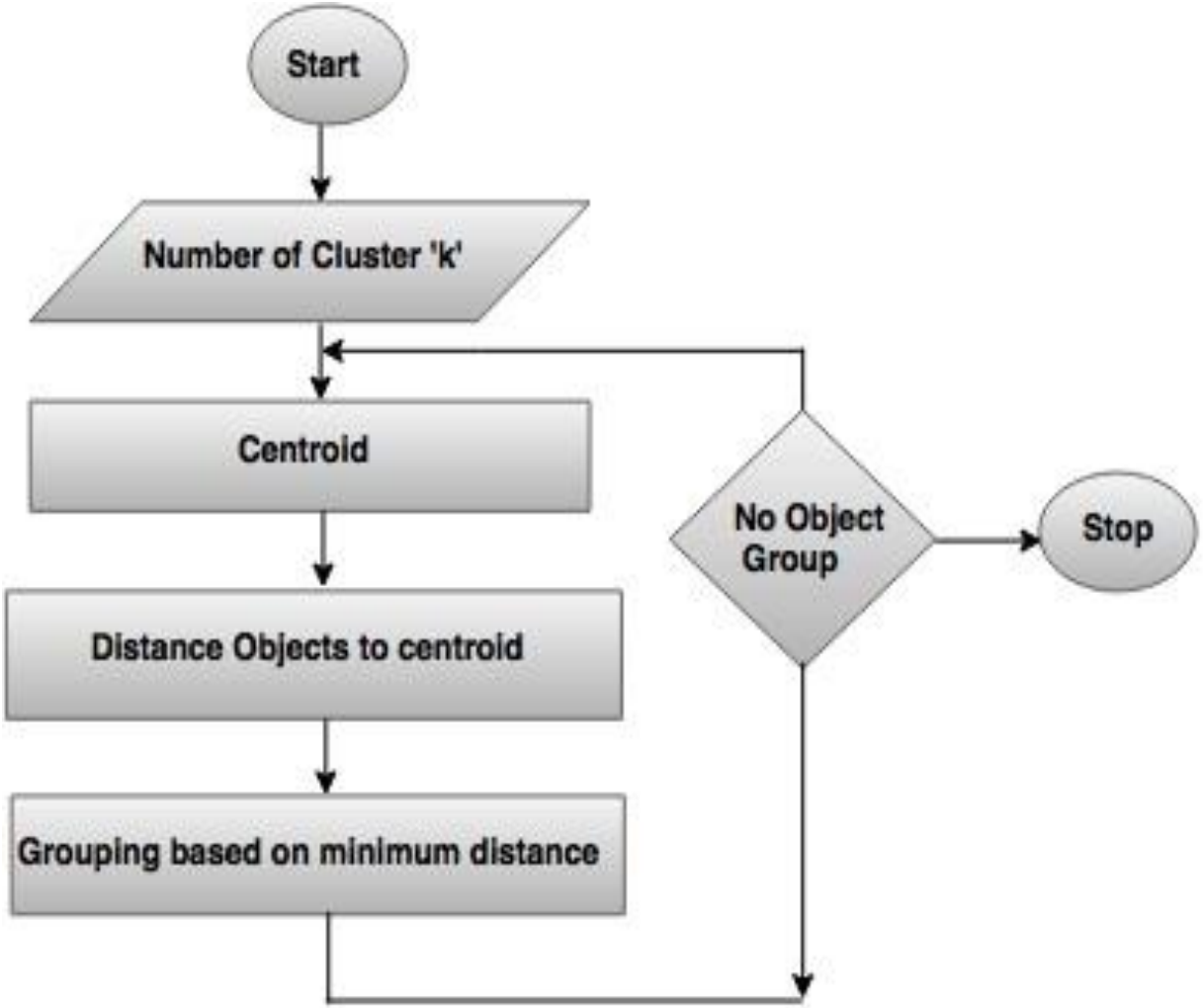
K-Means

- The K-means problem is an unsupervised learning algorithm that groups unlabeled data into clusters. The algorithm's goal is to find groups in data, where the number of groups is represented by K.
- The K-means algorithm is also known as clusterization.
- K-means algorithm is a centroid-based algorithm that calculates the distance between each data point and a centroid to assign it to a cluster.
- The grouping is done by minimizing the sum of the distances between each object and the group or cluster centroid.
- How K is determined ?

K-Means



K-means algorithm



K- mean Algorithm

K-means algorithm

- K-Means Clustering is an **Unsupervised Learning algorithm**, which groups the unlabeled dataset into different clusters. Here K defines the number of pre-defined clusters that need to be created in the process, as if K=2, there will be two clusters, and for K=3, there will be three clusters, and so on.
- It allows us to cluster the data into different groups and a convenient way to discover the categories of groups in the unlabeled dataset on its own without the need for any training.
- It is a centroid-based algorithm, where each cluster is associated with a centroid. The main aim of this algorithm is to minimize the sum of distances between the data point and their corresponding clusters.

How does the K-Means Algorithm Work?

Step-1: Select the number K to decide the number of clusters.

Step-2: Select random K points or centroids. (It can be other from the input dataset).

Step-3: Assign each data point to their closest centroid, which will form the predefined K clusters.

Step-4: Calculate the variance and place a new centroid of each cluster.

Step-5: Repeat the third steps, which means reassign each data point to the new closest centroid of each cluster.

Step-6: If any reassignment occurs, then go to step-4 else go to FINISH.

Step-7: The model is ready.

K-means problem: Objective function

Objective Function:

The objective of K-Means is to minimize the within-cluster sum of squares (WCSS), which is essentially the sum of squared distances between each point and its centroid in a cluster, summed over all clusters. Mathematically, it is represented as:

$$\text{Minimize} \sum_{i=1}^{K} \sum_{x \in S_i} ||x - \mu_i||^2$$

Here, S_i is the set of points in the i-th cluster, μ_i is the mean of points in S_i, and $||x - \mu_i||$ is the distance between a point x and the mean μ_i of the cluster it belongs to.

The solution space of the K-Means

- **Discrete and Non-Convex**: The solution space is discrete and non-convex. Each cluster assignment represents a distinct point in the solution space, and there's no smooth, continuous path from one solution to another. The non-convex nature of the space means that local optimization techniques, like the standard K-Means algorithm, can easily get trapped in local minima.
- **Dependent on the Number of Clusters (K)**: The size and complexity of the solution space are directly dependent on the chosen value of K (the number of clusters). For a fixed set of data points, the larger the K, the more potential cluster combinations there are, exponentially increasing the solution space.
- **Size of Solution Space**: For N data points and K clusters, there are KN possible ways to assign each point to one of the K clusters. However, many of these assignments might be redundant or invalid (e.g., a cluster with no points), especially as K gets close to N.

The solution space of the K-Means

- **High-Dimensional Spaces**: In high-dimensional spaces, the behavior of distances and density can become counterintuitive, complicating the clustering process. This phenomenon, often referred to as the "curse of dimensionality," can make it more challenging to find meaningful clusters and can affect the geometry of the solution space.
- **Influence of Initial Centroids**: The solution obtained is highly influenced by the initial choice of centroids. Different initializations can lead to different local minima, meaning that the algorithm might converge to different solutions on different runs.
- **Sensitivity to Data Distribution**: The structure and distribution of the data also play a significant role. For example, K-Means tends to work well if the clusters are spherical and of similar size. If the actual clusters in the data are elongated or of different sizes, this can distort the solution space and make it hard for K-Means to find a good solution.
- **Global Optimization is Challenging**: Finding the global optimum in such a vast and complex solution space is computationally expensive and often impractical, especially for large datasets.

This is why K-Means is considered NP-hard in its most general form.

Proving that K-Means is NP-hard

- Proving that K-Means is NP-hard involves demonstrating that it is at least as hard as the hardest problems in NP (Nondeterministic Polynomial time).
- The NP-hardness of K-Means clustering, particularly for a general case where the number of clusters *K* is part of the input, can be established using a reduction from a known NP-hard problem.
- One common approach is to use a reduction from the NP-hard **Graph Partitioning problem**.

Steps to prove K-Means is NP-hard

1.Choose a Known NP-Hard Problem for Reduction

The Graph Partitioning problem is often chosen for this purpose. In this problem, the goal is to partition the vertices of a graph into *K* groups of specified sizes such that the number of edges between different groups is minimized.

2. Construct an Instance of K-Means from the NP-Hard Problem

Translate an instance of the Graph Partitioning problem into an instance of the K-Means problem. This involves constructing a set of points in a high-dimensional space such that the optimal partition of these points into *K* clusters corresponds to an optimal solution of the original Graph Partitioning problem.

3. Ensure Polynomial Time Transformation

The transformation itself must be done in polynomial time to satisfy the requirements of NP-hardness proofs. This means the time it takes to convert the Graph Partitioning problem to a K-Means problem must be polynomial in the size of the input.

Proving that K-Means is NP-hard

4. Prove Equivalence of Solutions

Show that a solution to the K-Means problem provides a solution to the original Graph Partitioning problem, and vice versa. This involves demonstrating that if you can solve the K-Means problem efficiently, you can also solve the Graph Partitioning problem efficiently, which is known to be NP-hard.

5. Address the Specifics of K-Means

Address the specific characteristics of the K-Means problem, such as the Euclidean distance metric and the mean computation of clusters, to ensure they align with the requirements of the Graph Partitioning problem.

6. Conclude NP-Hardness

Once you have established that any instance of the Graph Partitioning problem can be polynomially transformed into an instance of the K-Means problem, and that solving the K-Means problem would also solve the Graph Partitioning problem, you can conclude that K-Means is at least as hard as Graph Partitioning, and therefore is NP-hard.

Questions and answers related to NP-completeness

Questions and answers related to NP-completeness

[1] What does NP stand for?

NP stands for "Nondeterministic Polynomial time." Problems in NP are those for which a proposed solution can be checked for correctness in polynomial time.

[2] What does it mean for a problem to be NP-complete?

An NP-complete problem is one that is both in NP and NP-hard. This means that not only can a solution to the problem be verified quickly, but the problem is also at least as hard as the hardest problems in NP.

[3] How is the reduction from SAT to 3-SAT constructed?

The reduction involves taking a clause with more than three literals and breaking it down into a series of clauses with exactly three literals that are collectively equivalent to the original clause. This is done using additional "helper" variables to ensure equivalence.

Questions and answers related to NP-completeness

[4] Why is the concept of reduction so important in proving NP-completeness?

Reductions show that a problem is at least as hard as another problem. If you can reduce an NP-complete problem to another problem in polynomial time, it implies the second problem is at least as hard as all problems in NP.

[5] What was the significance of Cook's Theorem?

Cook's Theorem was the first to show that there exists an NP-complete problem, which was the SAT problem. It established the foundation for the theory of NP-completeness.

[6] Does P = NP?

The question of whether P, the class of problems that can be solved quickly (in polynomial time), is equal to NP is one of the most important open problems in computer science. It is unknown whether a polynomial-time algorithm can be found for all problems in NP.

Questions and answers related to NP-completeness

[7] Can NP-complete problems be solved quickly?

As of my last update in April 2023, no polynomial-time algorithm is known for solving NP-complete problems in general. However, specific instances of these problems can sometimes be solved quickly, and heuristic algorithms can provide solutions that are good enough for practical purposes.

[8] Why are NP-complete problems so important in computer science?

NP-complete problems are crucial because they are representative of a class of problems that are computationally intensive. Understanding the nature of these problems can lead to breakthroughs in computational theory and the development of new algorithms.

Questions and answers related to NP-completeness

[9] What is a literal in the context of SAT and 3-SAT problems?

- A literal is a variable or the negation of a variable. In the context of SAT (Satisfiability) problems, a literal is an expression that can take on a value of true or false. In a 3-SAT problem, each clause in the formula must have exactly three literals.

[10] What does it mean to reduce one problem to another?

- In computational complexity theory, reducing one problem to another means transforming instances of the first problem into instances of the second problem in such a way that the solutions to the instances of the second problem correspond directly to solutions of the instances of the first problem. If this transformation can be done in polynomial time, then we say that the first problem reduces to the second.

Questions and answers related to NP-completeness

[11] How do we know that a problem is in NP?

- A decision problem is in NP if the solution to any instance of the problem can be verified as correct or incorrect in polynomial time given the correct certificate (or witness), even if finding the certificate itself might be hard.

[12] What is a certificate (or witness) in NP problems?

- A certificate or witness for an instance of an NP problem is a piece of information that, if provided alongside the instance, allows for a quick (polynomial-time) verification of the instance's solution. For example, in the context of the 3-SAT problem, a certificate would be a particular assignment of truth values to variables that satisfies the formula.

Questions and answers related to NP-completeness

[13] Why do we care about NP-complete problems outside of theoretical computer science?

- NP-complete problems occur in many practical fields, including operations research, economics, biology, and more. Often, these problems must be solved to make real-world decisions. While exact solutions may be computationally infeasible, understanding the structure of NP-complete problems can lead to better heuristics and approximation algorithms that work well in practice.

[14] Are there any problems that are harder than NP-complete problems?

- Yes, there are problem classes that are believed to be even harder than NP-complete problems, such as NP-hard problems that are not in NP, and problems in the complexity classes PSPACE or EXPTIME, which include problems that require polynomial space or exponential time, respectively, even to verify a solution.

Questions and answers related to NP-completeness

[15] Has anyone proved that P ≠ NP or P = NP?

- As of my last update, the P vs NP question remains unresolved. It is one of the seven Millennium Prize Problems for which the Clay Mathematics Institute has offered a million-dollar prize for a correct proof.

[16] What is the significance of proving a problem is NP-complete?

- Proving a problem is NP-complete helps in understanding its computational difficulty. It also means that if someone finds a polynomial-time algorithm for any NP-complete problem, it would solve all problems in NP quickly, which would be a groundbreaking achievement in computer science.

[17] Can NP-complete problems be solved using quantum computers?

- Quantum computers hold promise for certain computational problems, but it is not yet clear whether they can solve NP-complete problems efficiently. Some problems that are hard for classical computers are more tractable on quantum computers due to quantum superposition and entanglement, but a general solution for NP-complete problems using quantum computing has not been discovered.


Colourbox
Thank you for your attention

References

- https://www.hackerearth.com/practice/algorithms/graphs/hamiltonian-path/tutorial/
- https://courses.lumenlearning.com/math4liberalarts/chapter/introduction-euler-paths/

Exercises

Exercise:

1. What is the difference between NP-hard and NP-complete?
2. How can you show a problem is NP-hard?
3. Why is the 3-SAT problem often used in reductions to prove NP-hardness?
4. What is the significance of polynomial-time reductions in proving NP-hardness?
5. Can an optimization problem be NP-complete, or are only decision problems classified as NP-complete?
6. How does the concept of a verifier fit into the definition of NP?
7. What would be the implications for computer science if P were equal to NP?
8. Are there any real-world applications of NP-hard problems?
9. What is a "gadget" in the context of reductions for NP-completeness proofs?
10. Is it possible for an NP-hard problem to be solvable in polynomial time?
11. Can you provide an example of a reduction from an NP-complete problem to the Graph Coloring problem?
12. How does the proof of NP-hardness for the Graph Coloring problem differ when the number of colors is fixed compared to when it is part of the input?

Exercise:

- Explain the process of reducing one problem to another to prove NP-hardness. How does the notion of reduction provide a framework for understanding the relative difficulty of computational problems?
- Discuss the historical context and significance of Cook's Theorem in establishing the foundation of NP-completeness. How did Cook's Theorem change the landscape of computational complexity?
- Analyze the role of the P vs NP problem in the development of computational complexity theory and its importance in both theoretical and practical aspects of computer science.
- Analyze the interplay between heuristics and NP-hardness. How do heuristics contribute to the practical solution of NP-hard problems, and what are their limitations?

Exercise:

- Describe the significance of approximation algorithms in dealing with NP-hard problems. How do they provide a practical approach to solving problems for which we currently lack polynomial-time solutions?
- Examine the concept of NP-hardness in the context of optimization problems. How does the NP-hardness of an optimization problem relate to the complexity of its decision version?
- Critically evaluate the statement: "If an algorithm can solve an NP-complete problem in polynomial time, it can solve all problems in NP in polynomial time." What are the assumptions and implications behind this statement?
- Discuss the impact of quantum computing on the classification of problems as NP-hard. What are the potential breakthroughs and limitations of quantum algorithms in addressing NP-hard problems?