NP Hard Problem

Dr. Bibhudatta Sahoo

Communication & Computing Group

CS215, Department of CSE, NIT Rourkela

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

P and NP

Decision problem: A problem with a **yes** or **no** answer.

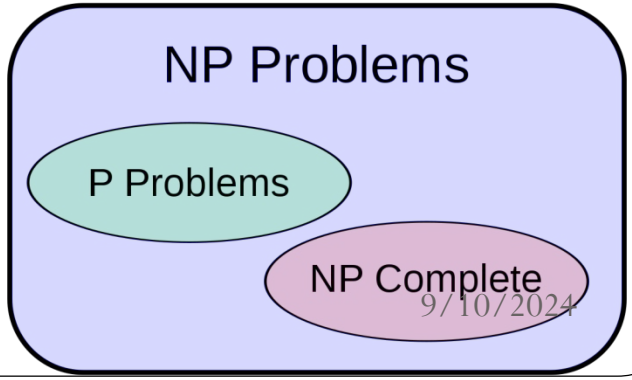
P: P is a complexity class that represents the set of all decision problems that can be **solved in polynomial time**.

- That is, given an instance of the problem, the answer yes or no can be decided in polynomial time.
- If one has a polynomial deterministic algorithm for a problem (e.g., the sorting problem), then the problem is called a **P-class** problem.
- The set of all such problems constitute the **P-class.**

P and NP

NP : NP is a complexity class that represents the set of all decision problems for which the **instances** where the answer is "yes" have proofs that can be verified in polynomial time.

- This means that if someone gives us **an instance** of the problem and a certificate (sometimes called a witness) to the answer being yes, we can check that it is correct in polynomial time.
- If one has a polynomial **non-deterministic** algorithm for a problem, then the problem is called a **NP-class** problem.
- The set of all such problems constitute the N**P-class.**



NP-Complete

- *NP-Complete is a complexity class which represents the set of all problems X in NP for which it is possible to* ***reduce*** *any other NP problem Y to X in polynomial time.*
- Intuitively this means that we can solve Y quickly if we know how to solve X quickly. Precisely, Y is reducible to X, if there is a polynomial time algorithm f to transform instances y of Y to instances x = f(y) of X in polynomial time, with the property that the answer to y is yes, if and only if the answer to f(y) is yes.
- What makes NP-complete problems important is that if a deterministic polynomial time algorithm can be found to solve one of them, every NP problem is solvable in polynomial time (one problem to rule them all).
- The theory of the NP-Completeness does not provide any method of obtaining polynomial time algorithms for the problems of the second group. "Many of the problems for which there is no polynomial time algorithm available are computationally related".

NP-Hard Problems

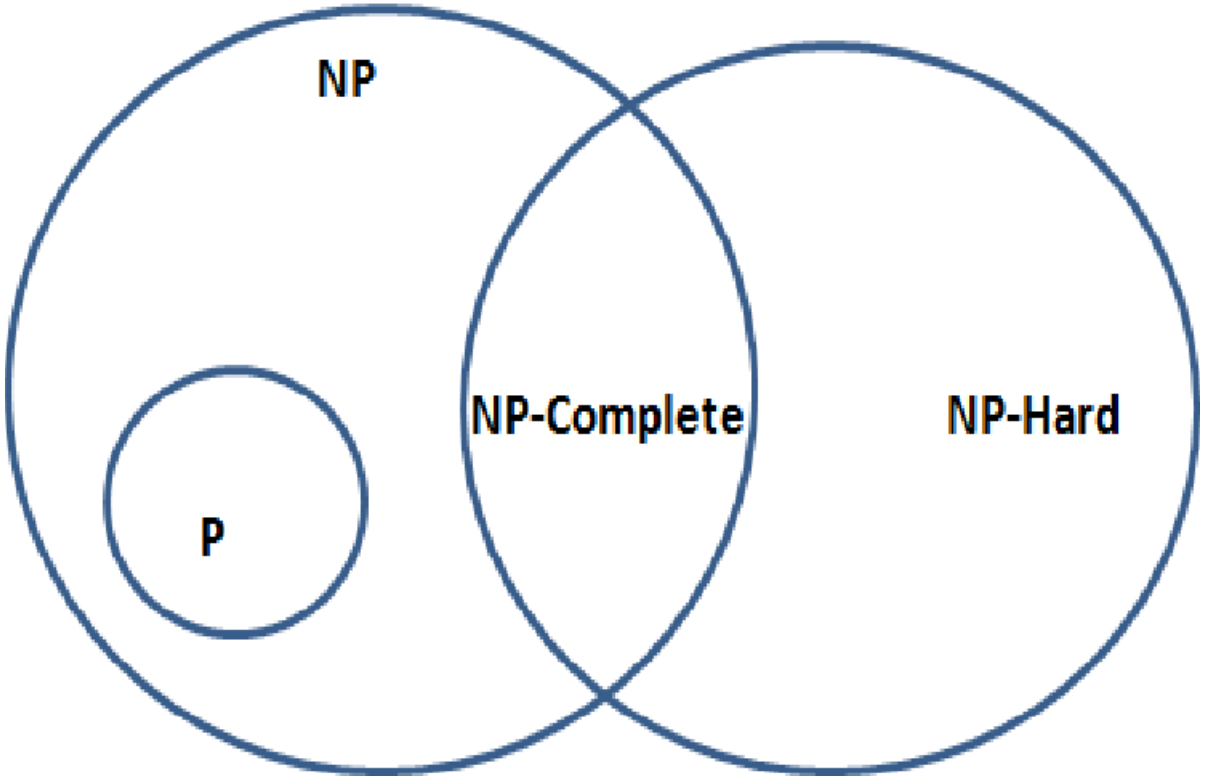
- Intuitively, these are the problems that are *at least as hard as the NP-complete problems*. Note that **NP-hard problems** *do not have to be in NP*, and *they do not have to be decision problems*.
- The precise definition here is that *a* ***problem X*** *is NP-hard, if there is an NP-complete problem Y, such that Y is reducible to X in* ***polynomial time***.
- But since any NP-complete problem can be reduced to any other NP-complete problem in polynomial time, all NP-complete problems can be reduced to any NP-hard problem in polynomial time. Then, if there is a solution to one NP-hard problem in polynomial time, there is a solution to all NP problems in polynomial time.

Example:

- The **halting problem** is an NP-hard problem. This is the problem that given a program P and input I, will it halt? This is a decision problem but it is not in NP. It is clear that any **NP-complete** problem can be reduced to this one. As another example, any **NP-complete problem** is **NP-hard**.

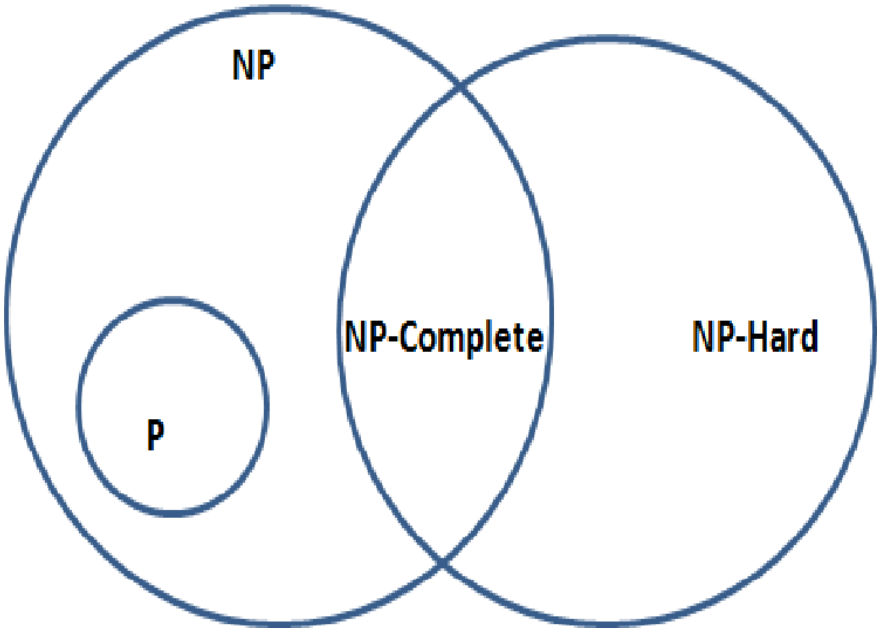
NP-Hard Problems

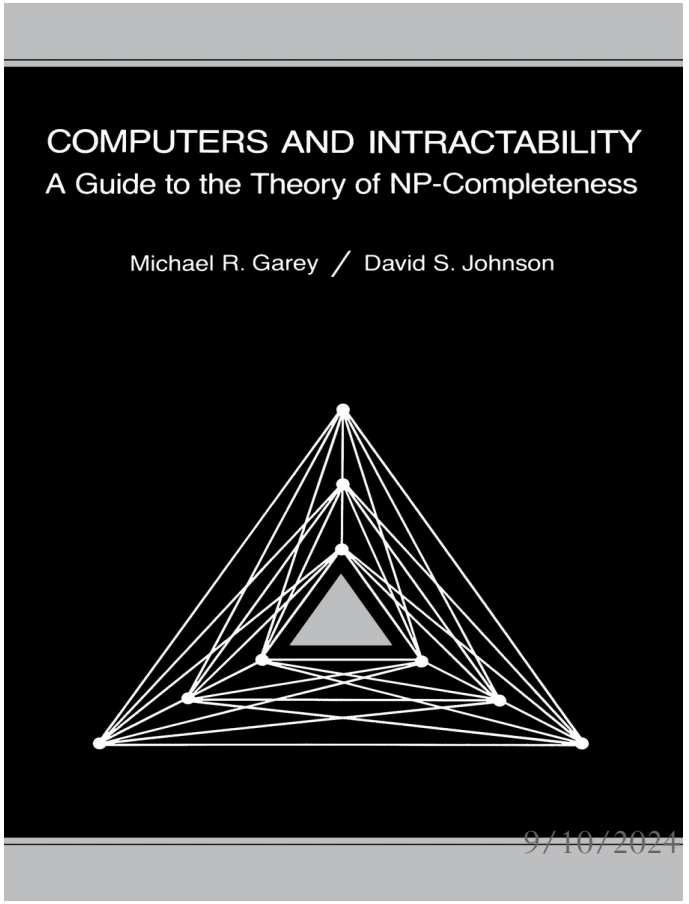
- **NP-Hard Problems**: A problem is classified as NP-Hard when an algorithm for solving it can be translated to solve *any* NP problem. Then we can say, this problem is *at least* as hard as any NP problem, but it could be much harder or more complex.



NP-Hard Problems

- A general list of NP-complete problems can be found in Garey & Johnson's book "Computers and Intractability".
- It contains an appendix that lists **roughly 300** NP-complete problems, and despite its age is often suggested when one wants a list of NP-complete problems.



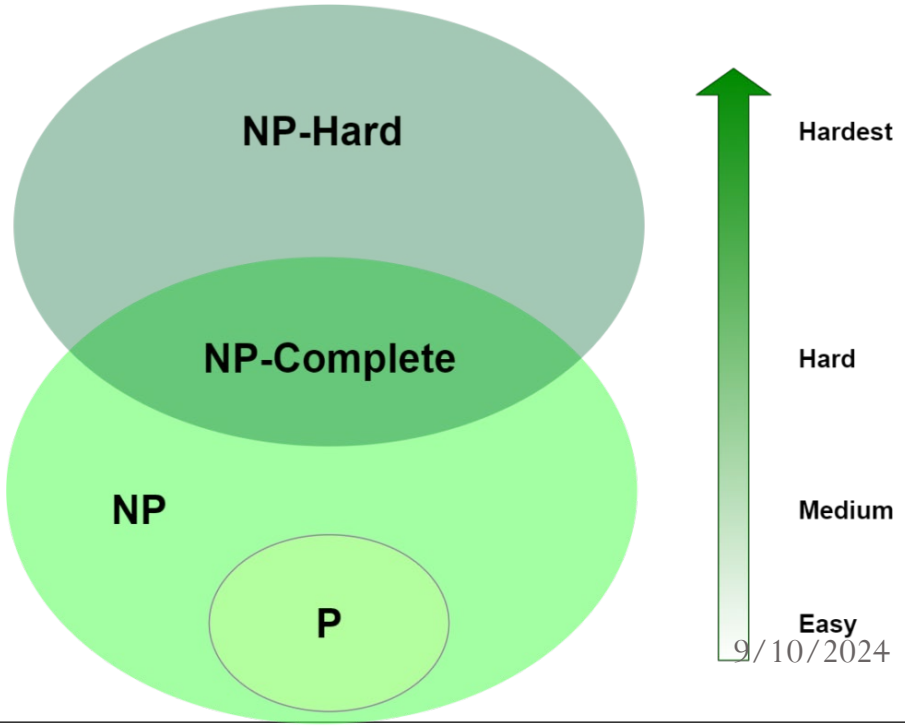


Proof of NP Hard

Problem Classification

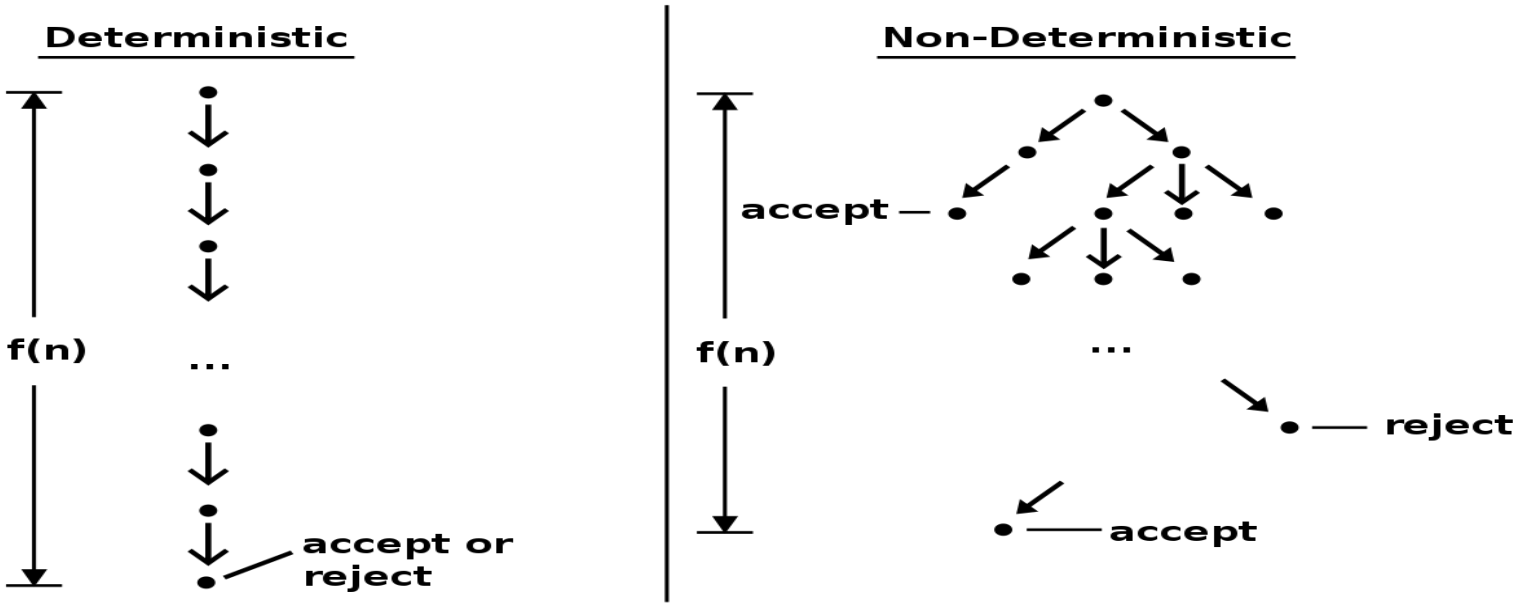
- **In theoretical computer science, the classification and complexity of common problem definitions have two major sets;** P which is "Polynomial" time and NP which "Non-deterministic Polynomial" time. There are also NP-hard and NP-complete sets, which we use to express more sophisticated problems. In the case of rating from easy to hard, we might label these as "easy", "medium", "hard", and finally "hardest":

- Easy → P
- Medium → NP
- Hard → NP-Complete
- Hardest → NP-Hard



Deterministic vs non-deterministic algorithm

- A **deterministic algorithm** represents a single path from an input to an outcome.
- A **nondeterministic algorithm** represents a single path **stemming** into many paths, some of which may arrive at the same output and some of which may arrive at unique outputs.

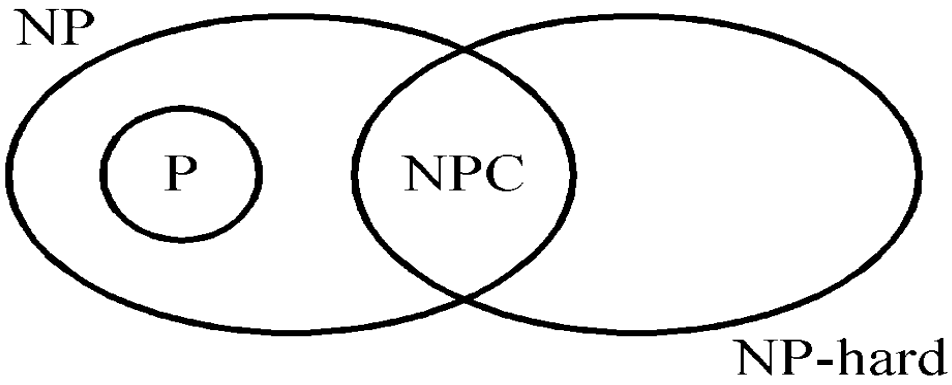


Difference between NP hard and NP complete problem

- **NP Problem:** The NP problems set of problems whose solutions are hard to find but **easy to verify** and are solved by **Non-Deterministic Machine** in **polynomial time**.

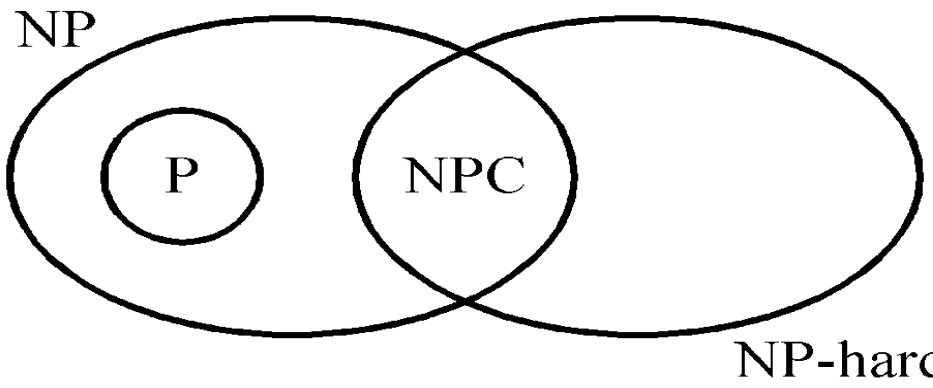
- **NP-Hard Problem:** Any decision problem P_i is called NP-Hard **if and only if** every problem of NP is reducible to P_i in polynomial time.

- **NP-Complete Problem:** Any problem is NP-Complete if it is a part of both NP and NP-Hard Problem.



Difference between NP hard and NP complete problem

- **NP-hard** problems are informally defined as those that can't be solved in polynomial time. In other words, the problems that are **harder** than P.
- **NP-complete** problems are the problems that are both **NP-hard**, and in **NP**.
- Proving that a problem is NP is usually trivial, but proving that a problem is NP-hard is not.
- **Boolean satisfiability (SAT)** is widely believed to be **NP-hard**, and thus the usual way of proving that **a problem** is **NP-complete** is to prove that there's a polynomial time transformation of the problem to SAT.



NP-hard Algorithm

- Among the hardest computer science problems are:
  - **K-means** Clustering
  - **Traveling Salesman Problem**, and
  - **Graph Coloring**
- These algorithms have a property similar to ones in NP-complete – they can all be reduced to any problem in NP. Because of that, these are in NP-hard and are at least as hard as any other problem in NP. A problem can be both in **NP** and **NP-hard**, which is another aspect of being NP-complete.
- This characteristic has led to a debate about whether or not **Traveling Salesman is indeed NP-complete** . Since and problems can be verified in polynomial time, proving that an algorithm cannot be verified in polynomial time is also sufficient for placing the algorithm in NP-hard

Difference between NP-Hard and NP-Complete:

NP-HARD	NP-COMPLETE
• NP-Hard problems(say X) can be solved if and only if there is a NP-Complete problem(say Y) can be reducible into X in polynomial time.	• NP-Complete problems can be solved by deterministic algorithm in polynomial time.
• To solve this problem, it must be a NP problem.	• To solve this problem, it must be both NP and NP-hard problem.
• It is not a Decision problem.	• It is exclusively Decision problem .
• **Example:** Halting problem, Vertex cover problem, Circuit-satisfiability problem, etc.	• **Example:** Determine whether a graph has a Hamiltonian cycle, Determine whether a Boolean formula is satisfiable or not, etc.

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

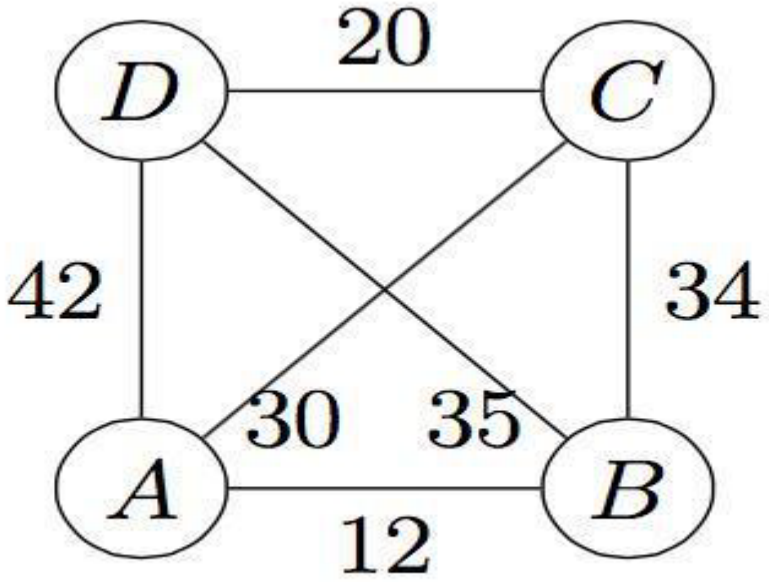
	P	NP	NP-complete	NP-hard
Solvable in polynomial time	✓			
Solution verifiable in polynomial time	✓	✓	✓	
Reduces any NP problem in polynomial time			✓	✓

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.**

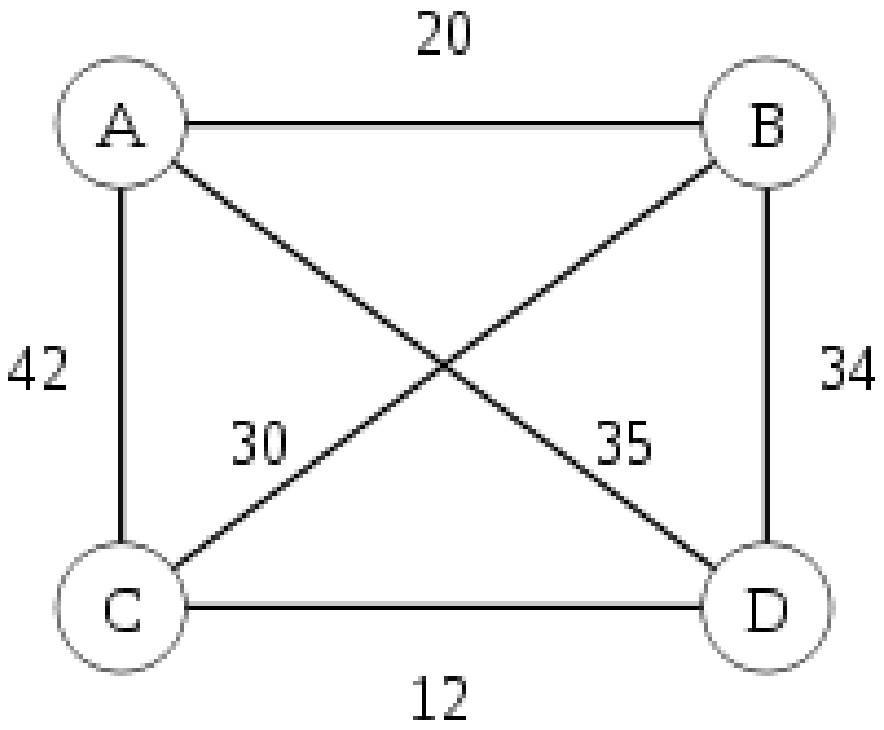
Proof that traveling salesman problem is NP Hard

- Given a set of cities and the distance between each pair of cities, the **travelling salesman problem(TSP)** finds the path between these cities such that it is the shortest path and traverses every city once, returning back to the starting point.
- **Problem –** Given a graph **G(V, E)**, the problem is to determine if the graph has a TSP consisting of cost at most **K**.



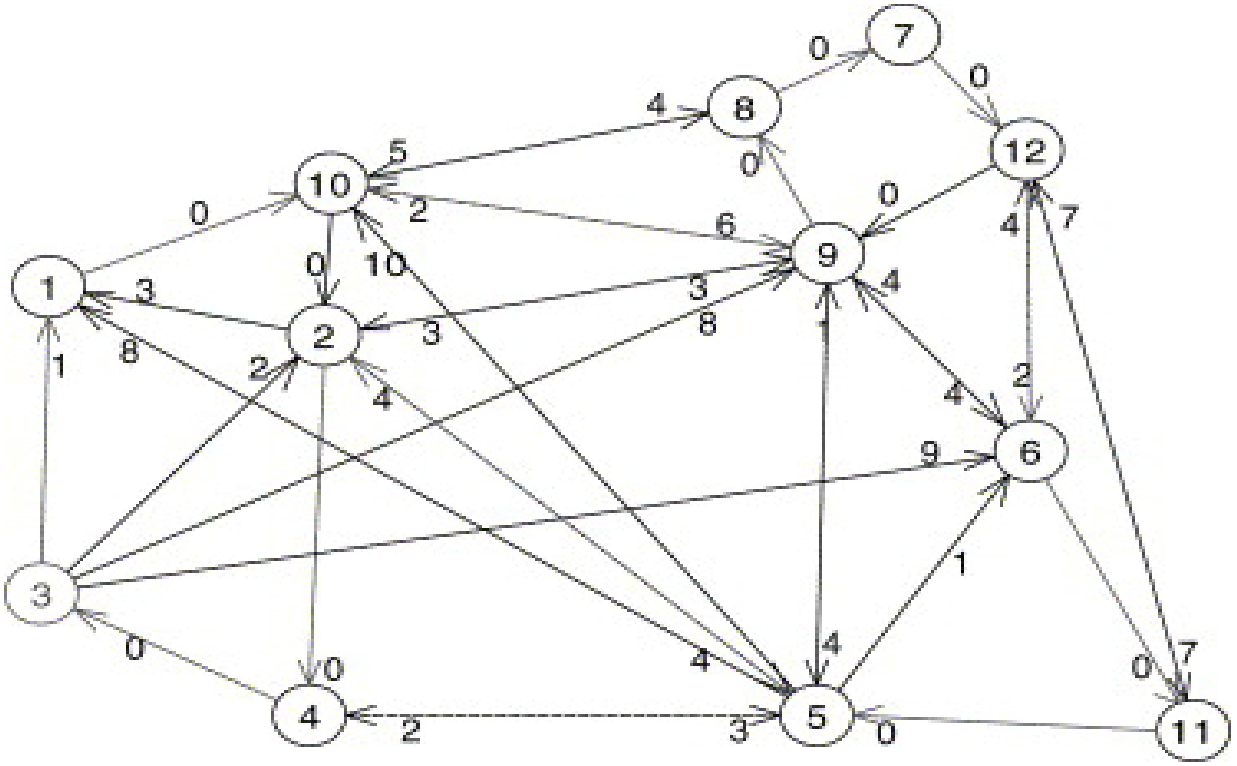
Symmetric TSP

- In the *symmetric TSP*, the distance between two cities is the same in each opposite direction, forming an undirected graph. This symmetry halves the number of possible solutions.



Asymmetric TSP

- In the *asymmetric TSP*, paths may not exist in both directions or the distances might be different, forming a directed graph. Traffic collisions, one-way streets, and airfares for cities with different departure and arrival fees are examples of how this symmetry could break down.



Solution for TSP

- The most direct solution would be to try all permutations (ordered combinations) and see which one is cheapest (using brute-force search).
- The running time for this approach lies within a polynomial factor of , O(n!), the factorial of the number of cities
- For **10** cities, we have **36,28,800** combinations.
- The 20 cities with solutions (5.919012181389928e+36), so this solution becomes impractical even for only 20 cities.
- One of the earliest applications of dynamic programming is the Held–Karp algorithm that solves the problem in time with time complexity $O(n^2 2^n)$.

Miller–Tucker–Zemlin formulation: TSP

Label the cities with the numbers 1, ..., n and define:

$$x_{ij} = \begin{cases} 1 & \text{the path goes from city } i \text{ to city } j \\ 0 & \text{otherwise} \end{cases}$$

For $i = 1, \ldots, n$, let u_i be a dummy variable, and finally take $c_{ij} > 0$ to be the distance from city i to city j. Then TSP can be written as the following integer linear programming problem:

$$\min \sum_{i=1}^{n} \sum_{j \neq i, j=1}^{n} c_{ij} x_{ij}:$$

$$x_{ij} \in \{0,1\} \qquad i,j = 1,\ldots,n;$$

$$u_i \in \mathbf{Z} \qquad i = 2,\ldots,n;$$

$$\sum_{i=1, i \neq j}^{n} x_{ij} = 1 \qquad j = 1,\ldots,n;$$

$$\sum_{j=1, j \neq i}^{n} x_{ij} = 1 \qquad i = 1,\ldots,n;$$

$$u_i - u_j + n x_{ij} \leq n - 1 \qquad 2 \leq i \neq j \leq n;$$

$$1 \leq u_i \leq n - 1 \qquad 2 \leq i \leq n.$$

Dantzig–Fulkerson–Johnson formulation

Label the cities with the numbers 1, ..., n and define:

$$x_{ij} = \begin{cases} 1 & \text{the path goes from city } i \text{ to city } j \\ 0 & \text{otherwise} \end{cases}$$

Take $c_{ij} > 0$ to be the distance from city i to city j. Then TSP can be written as the following integer linear programming problem:

$$\min \sum_{i=1}^{n} \sum_{j \neq i, j=1}^{n} c_{ij} x_{ij}:$$

$$x_{ij} \in \{0,1\} \qquad i,j = 1,\ldots,n;$$

$$\sum_{i=1, i \neq j}^{n} x_{ij} = 1 \qquad j = 1,\ldots,n;$$

$$\sum_{j=1, j \neq i}^{n} x_{ij} = 1 \qquad i = 1,\ldots,n;$$

$$\sum_{i \in Q} \sum_{j \neq i, j \in Q} x_{ij} \leq |Q| - 1 \qquad \forall Q \subsetneq \{1,\ldots,n\}, |Q| \geq 2$$

Why TSP is Not NP-complete

- Why is TSP not NP-complete? The simple answer is that it's NP-hard, but it's not in NP. Since it's not in NP, it can't be NP-complete.
- In TSP you're looking for the shortest loop that goes through every city in a given set of cities.
- Suppose you're given a set of cities, and the solution for the shortest loop among these cities. How would you verify that the solution you're given really is the shortest loop?
- In other words, how do you know there's not another loop that's shorter than the one given to you? The only known way to verify that a provided solution is the shortest possible solution is to actually solve TSP.
- Since it takes exponential time to solve NP, the solution cannot be checked in polynomial time. Thus this problem is NP-hard, but *not* in NP.

Why TSP is Not NP-complete

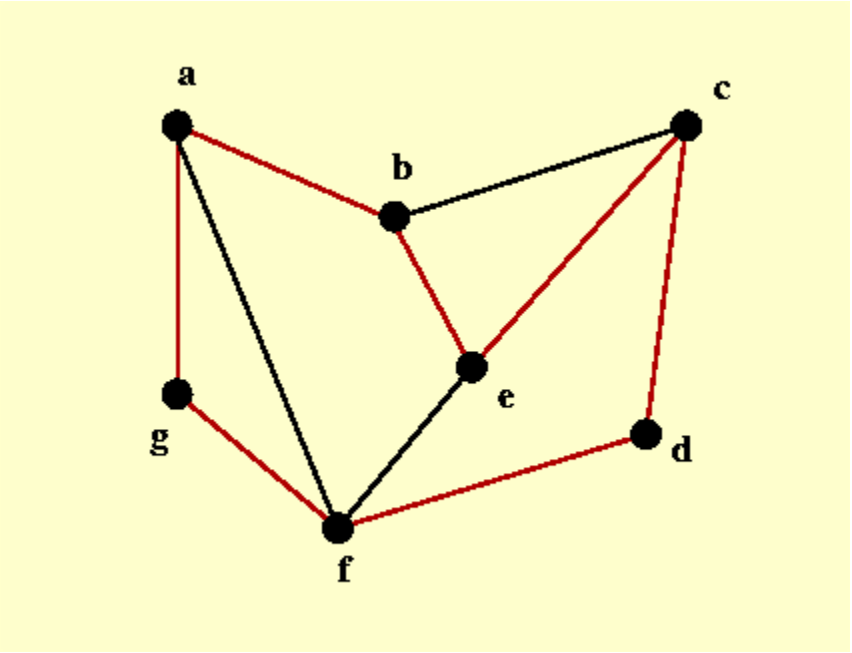
- In general, for a problem to be NP-complete it has to be a "decision problem", meaning that the problem is to decide if something is true or not.
- There's a simple variation of TSP called "decision TSP" that turns it into a decision problem.
- Imagine that instead of finding the shortest loop going through all cities, your goal is to determine if there exists any loop whose total length is less than some fixed number.
- For example, the question might be: is there a loop that goes through all of these cities, whose total distance is less than 100 km? In the negative case this is just as hard as regular TSP, because you'd end up testing all possible paths. But there's an important difference: the solution can be verified in linear time by adding up all of the distances making up the path, and that's what makes this variant part of NP.
- There's a straightforward proof that the decision variant is also NP-complete, by showing that there's an equivalence between the TSP decision problem and the **Hamiltonian path problem**.

What Does NP-hard Really Mean?

- Earlier I mentioned that the definition of NP-hard as "problems harder than P" is informal. In nearly all cases this is sufficient, but it's not technically accurate. Formally, a problem is NP-hard if given an oracle machine for the problem, all other problems in NP could be solved in polynomial time.
- The best known example of a problem that is in NP, but thought not to be NP-hard, is integer factorization. It's trivial to verify that the factorization of a number is correct, simply by taking the product of the factors given to you. This puts integer factorization in NP. However, it is widely believed that integer factorization is not NP-hard (and thus not NP-complete), because there doesn't appear to be any equivalence between integer factorization and other NP-complete problems.
- Integer factorization is therefore in a weird complexity class. We think it's not NP-hard, and intuitively it feels "easier" than NP-hard problems like 3-SAT. A lot of smart people have looked at integer factorization, and no one has found a polynomial time algorithm for integer factorization. Thus it is probably "in-between" P and NP-hard, but not one really knows for sure.

Hamiltonian cycle

- A **Hamiltonian cycle**, also called a **Hamiltonian circuit**, **Hamilton cycle**, or **Hamilton circuit**, is a **graph cycle** (i.e., closed loop) through a **graph** that visits each node exactly once (Skiena 1990, p. 196).
- A **graph** possessing a **Hamiltonian cycle** is said to be a **Hamiltonian graph**.

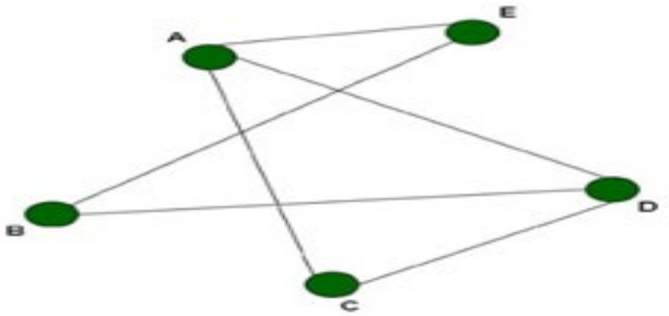


Proof that traveling salesman problem is NP Hard

- **Problem –** Given a graph **G(V, E)**, the problem is to determine if the graph has a TSP consisting of cost at most **K**.

Explanation

- In order to prove the Travelling Salesman Problem is NP-Hard, we will have to reduce a known **NP-Hard problem** to this problem.
- We will carry out a reduction from the **Hamiltonian Cycle** problem to the Travelling Salesman problem.
- Every instance of the Hamiltonian Cycle problem consists of a graph G =(V, E) as the input can be converted to a Travelling Salesman problem consisting of graph G' = (V', E') and the maximum cost, **K**.



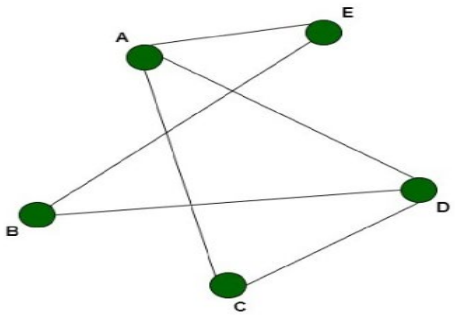
G =
Hamiltonian cycle {EACDBE}

Proof that traveling salesman problem is NP Hard

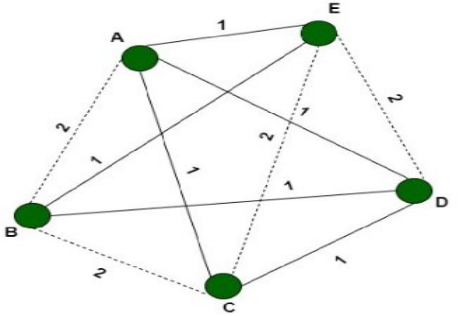
- The new graph G' can be constructed in polynomial time by just converting G to a complete graph G' and adding corresponding costs. This reduction can be proved by the following two claims:
- We will construct the graph G' in the following way: For all the edges e belonging to E, add the cost of edge c(e)=1.
- Connect the remaining edges, **e'** belonging to **E'**, that are not present in the original graph **G**, each with a cost **c(e')= 2**.
- Let us set K=N
- The new graph G' can be constructed in polynomial time by just converting G to a complete graph G' and adding corresponding costs. This reduction can be proved by the following two claims:

Proof that traveling salesman problem is NP Hard

- Let us assume that the graph **G** contains a Hamiltonian Cycle, traversing all the vertices V of the graph. Now, these vertices form a TSP with cost = N.
- Since it uses all the edges of the original graph having cost c(e)=1. And, since it is a cycle, therefore, it returns back to the original vertex.
- We assume that the graph G' contains a TSP with cost K =N.
- The TSP traverses all the vertices of the graph returning to the original vertex. Now since none of the vertices are excluded from the graph and the cost sums to n, therefore, necessarily it uses all the edges of the graph present in **E**, with cost 1, hence forming a Hamiltonian cycle with the graph **G**

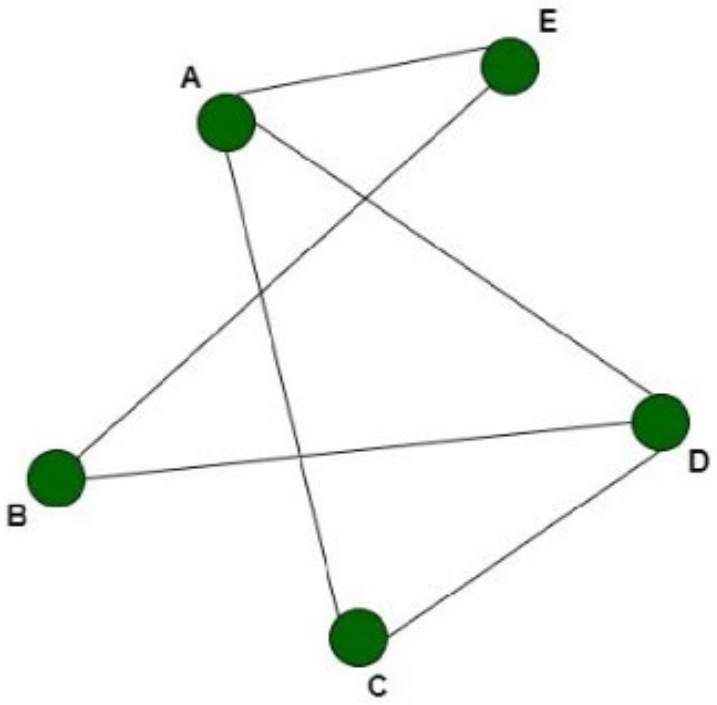


G =
Hamiltonian cycle {EACDBE}

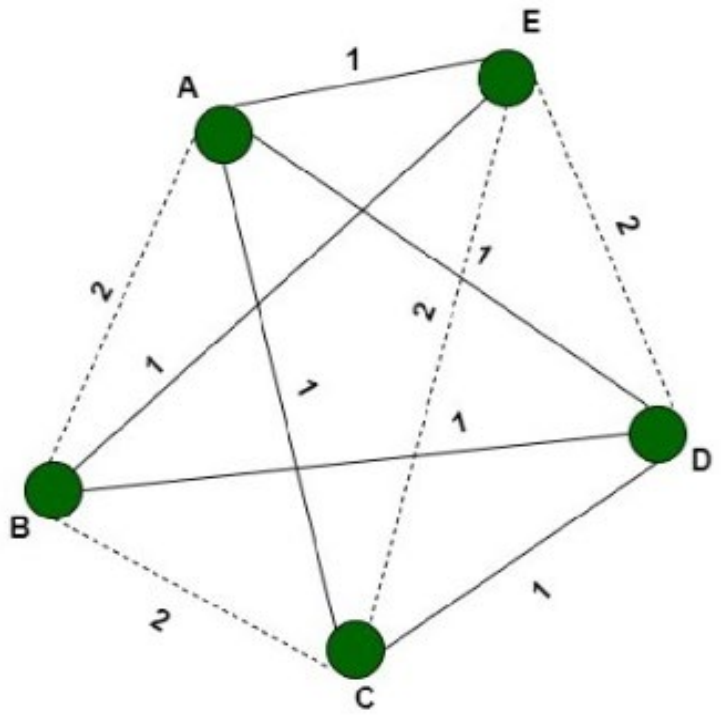


G' =
TSP {EACDBE}
Cost = 5 (=n)

Proof that traveling salesman problem is NP Hard



G =
Hamiltonian cycle {EACDBE}



G' =
TSP {EACDBE}
Cost = 5 (=n)

Proof that traveling salesman problem is NP Hard

$$\begin{matrix} A \\ B \\ C \\ D \\ E \end{matrix}\begin{bmatrix} 00111 \\ 00011 \\ 10010 \\ 11100 \\ 11000 \end{bmatrix}$$

$$\begin{matrix} A \\ B \\ C \\ D \\ E \end{matrix}\begin{bmatrix} 02111 \\ 20211 \\ 12012 \\ 11102 \\ 11220 \end{bmatrix}$$

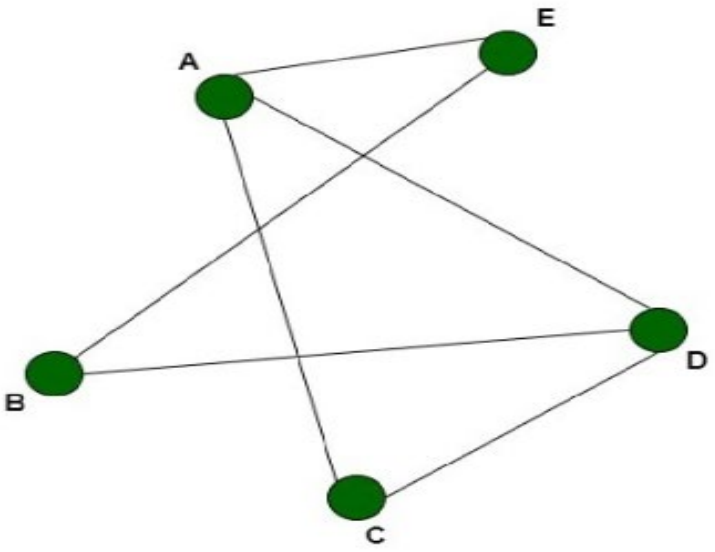
If i ≠ j and A(i,j) =0 then A(i,j) ← 2

$$\begin{matrix} A \\ B \\ C \\ D \\ E \end{matrix}\begin{bmatrix} 00111 \\ 00011 \\ 10010 \\ 11100 \\ 11000 \end{bmatrix}$$

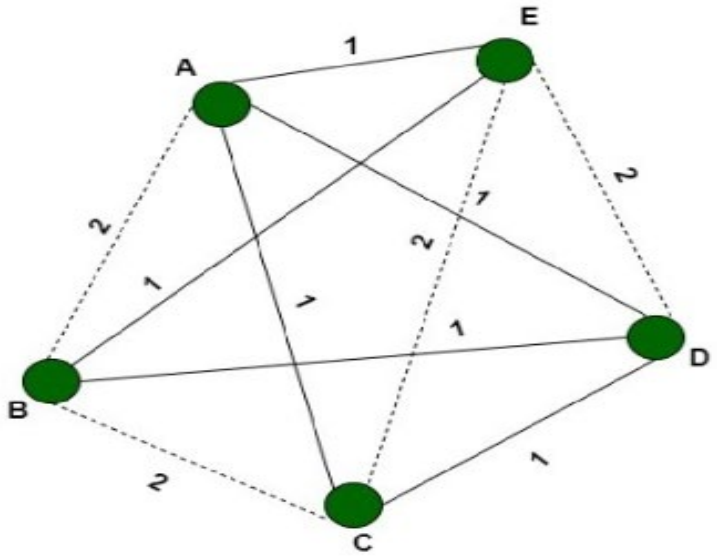
$$\begin{matrix} A \\ B \\ C \\ D \\ E \end{matrix}\begin{bmatrix} 02111 \\ 20211 \\ 12012 \\ 11102 \\ 11220 \end{bmatrix}$$

If i ≠ j and A(i,j) =0 then A(i,j) ← 2

Proof that traveling salesman problem is NP Hard



G =
Hamiltonian cycle {EACDBE}



G' =
TSP {EACDBE}
Cost = 5 (=n)

- Thus we can say that the graph **G'** contains a TSP if graph **G** contains Hamiltonian Cycle.
- Therefore, any instance of the Travelling salesman problem can be reduced to an instance of the hamiltonian cycle problem. Thus, the **TSP** is **NP-Hard**. □

k-means clustering is an NP-hard optimization problem

What is Clustering?

- Clustering is dividing data points into homogeneous classes or clusters:
  - Points in the same group are as similar as possible
  - Points in different group are as dissimilar as possible

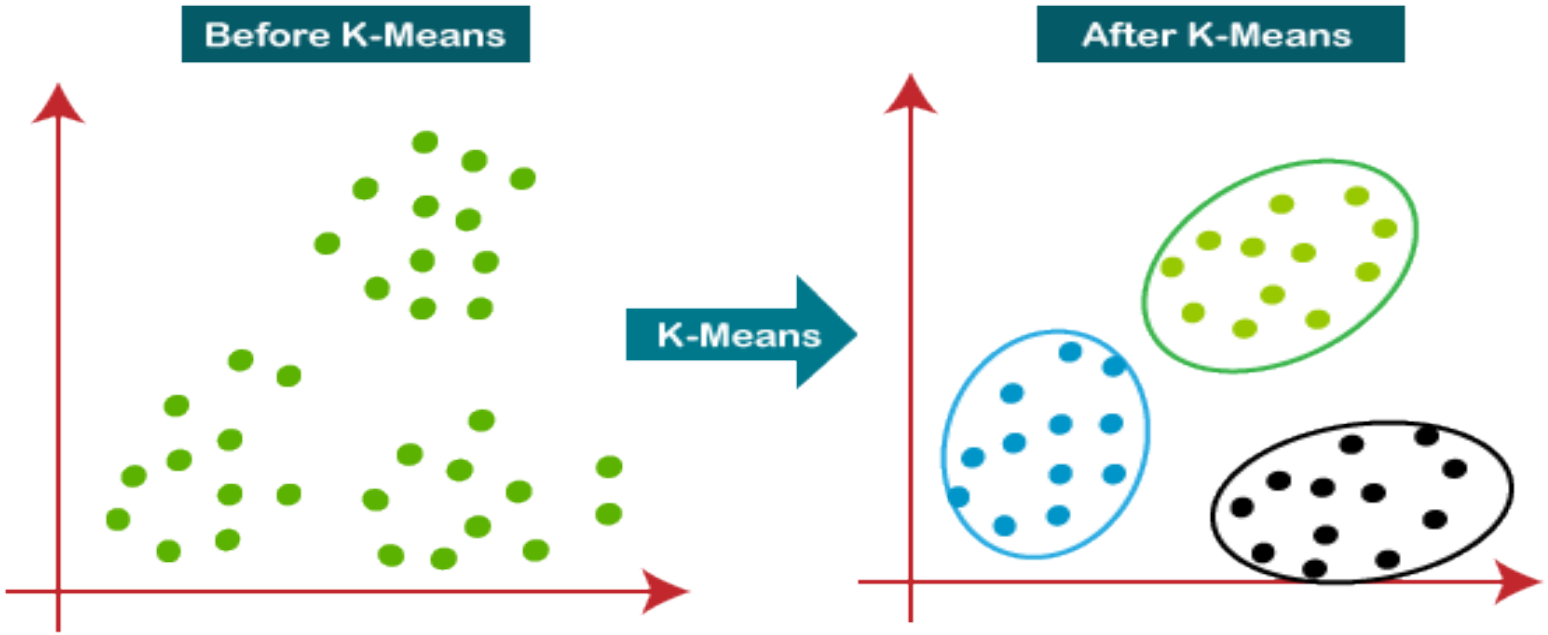
When a collection of objects is given, we put objects into group based on similarity.

Clustering Algorithms:

- A Clustering Algorithm tries to analyze natural groups of data on the basis of some similarity. It locates the centroid of the group of data points. To carry out effective clustering, the algorithm evaluates the distance between each point from the centroid of the cluster.
- The goal of clustering is to determine the intrinsic grouping in a set of unlabelled data.

What is K-means Clustering?

- K-means (Macqueen, 1967) is one of the simplest unsupervised learning algorithms that solve the well-known clustering problem.
- K-means clustering is a method of vector quantization, originally from signal processing, that is popular for cluster analysis in data mining.

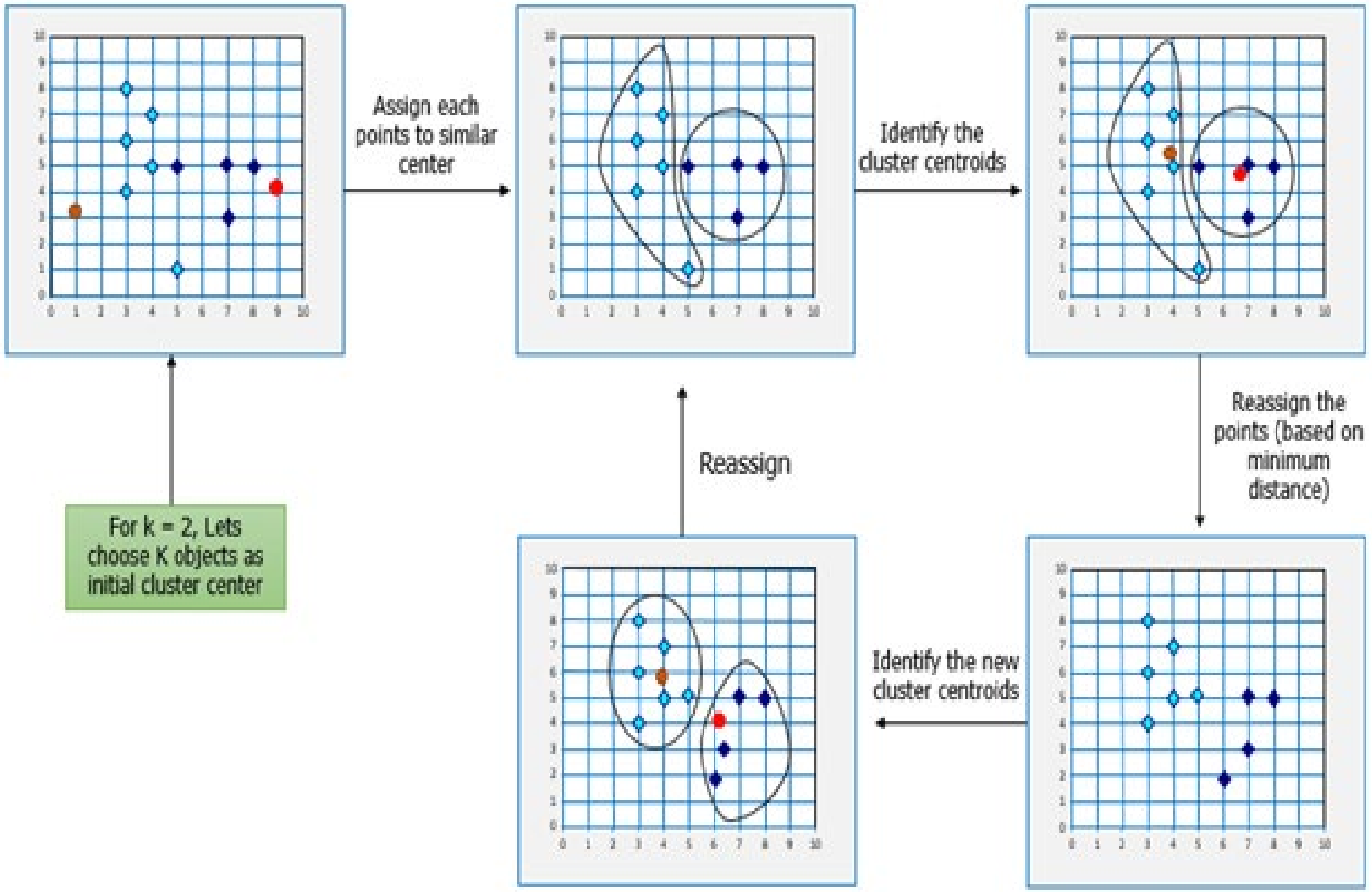


K-means Clustering Method:

If k is given, the K-means algorithm can be executed in the following steps:

1. Partition of objects into k non-empty subsets
2. Identifying the cluster centroids (mean point) of the current partition.
3. Assigning each point to a specific cluster
4. Compute the distances from each point and allot points to the cluster where the distance from the centroid is minimum.
5. After re-allotting the points, find the centroid of the new cluster formed.

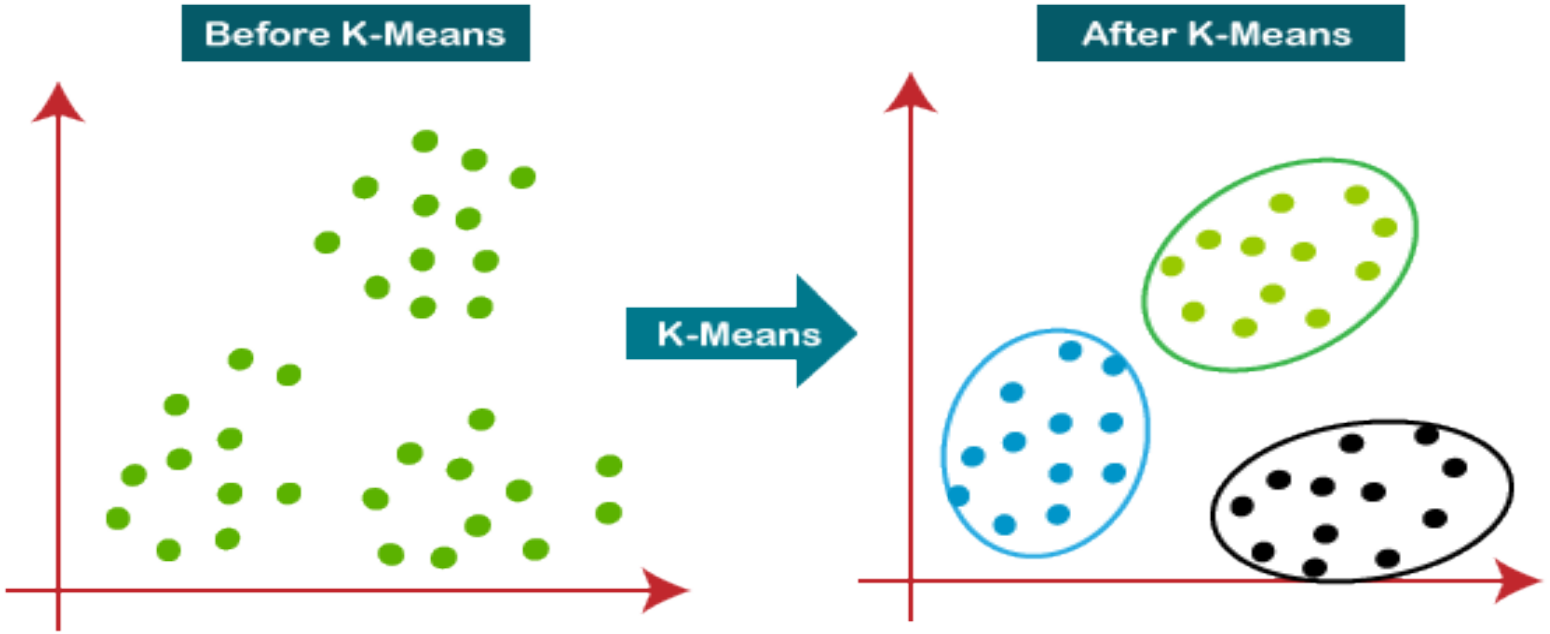
K-means Clustering Method: The step by step process:



k-means clustering is an NP-hard

Algorithm 1 k-means algorithm

1: Specify the number k of clusters to assign.
2: Randomly initialize k centroids.
3: **repeat**
4: **expectation:** Assign each point to its closest centroid.
5: **maximization:** Compute the new centroid (mean) of each cluster.
6: **until** The centroid positions do not change.



Proof of NP Hard

Generalized pseudocode for Traditional k-mean

Step 1: Accept the number of clusters to group data into and the dataset to cluster as input values

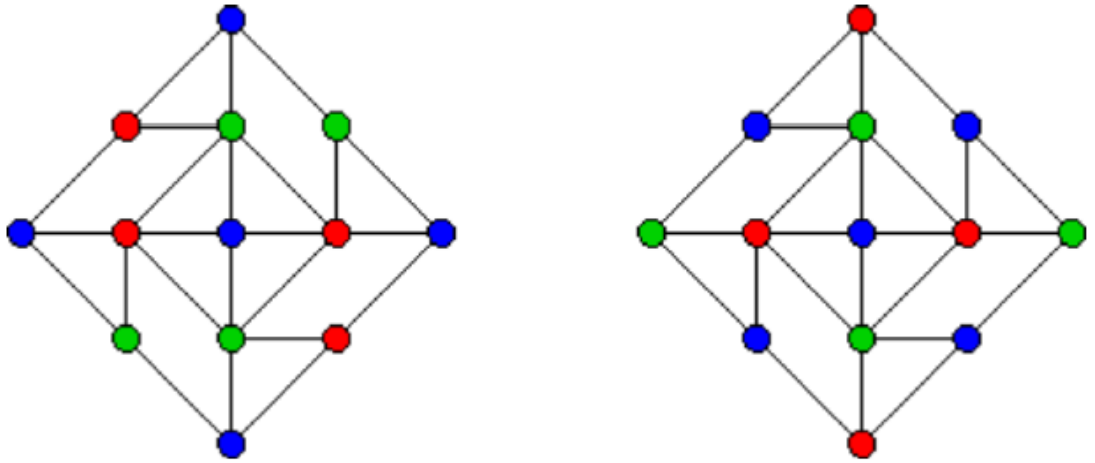
Step 2: Initialize the first K clusters
- Take first k instances or
- Take Random sampling of k elements

Step 3: Calculate the arithmetic means of each cluster formed in the dataset.

Step 4: K-means assigns each record in the dataset to only one of the initial clusters
- Each record is assigned to the nearest cluster using a measure of distance (e.g Euclidean distance).

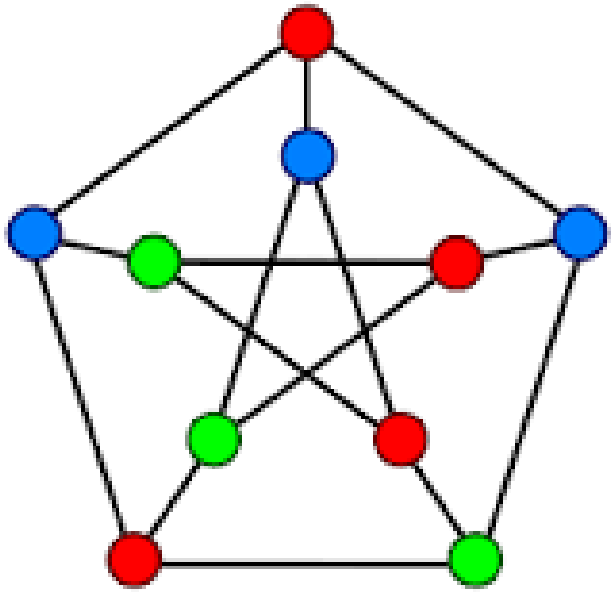
Step 5: K-means re-assigns each record in the dataset to the most similar cluster and re-calculates the arithmetic mean of all the clusters in the dataset.

Graph coloring is computationally hard



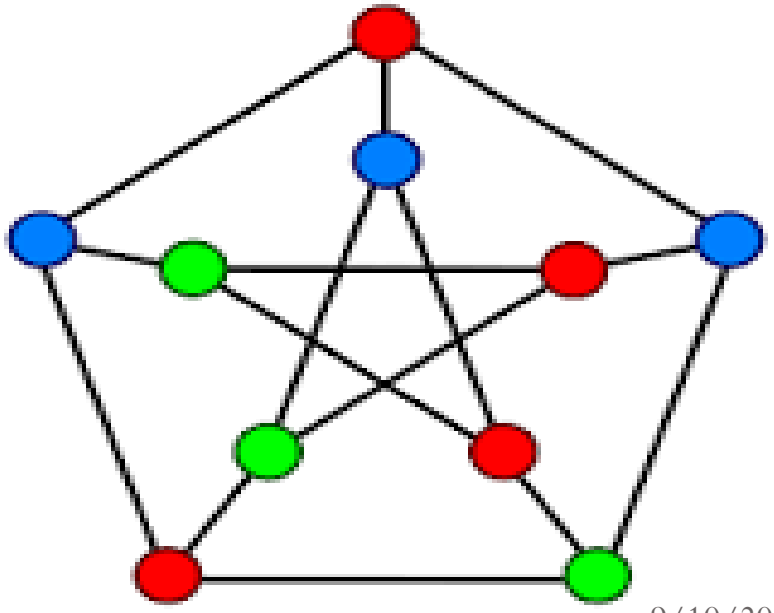
Graph coloring

- "A coloring of a simple graph is **the assignment of a color to each vertex of the graph such that no two adjacent vertices are assigned the same color**."
- A simple solution to this problem is to color every vertex with a different color to get a total of. colors.



Graph coloring

- Graph coloring is **the procedure of assignment of colors to each vertex of a graph G such that no adjacent vertices get same color**.
- The objective is to **minimize the number of colors** while coloring a graph.
- The smallest number of colors required to color a graph G is called its **chromatic number** of that graph.



Graph coloring is computationally hard

- Graph coloring is **computationally hard**.
- It is NP-complete to decide if a given graph admits a k-coloring for a given k except for the cases $k \in \{0,1,2\}$.
- In particular, it is NP-hard to compute the chromatic number.
- 3-COLOURING is in NP and that it is **NP-hard** by giving a reduction from 3-SAT. Therefore 3-COLOURING is **NP-complete**.
- The 3-coloring problem remains NP-complete even on 4-regular planar graphs.

Conclusions

1. NP-Complete- have the property that it can be solved in polynomial time if all other NP-Complete problems can be solved in polynomial time.
2. NP-Hard- if it can be solved in polynomial time then all NP-Complete can be solved in polynomial time.
3. "All NP-Complete problems are NP-Hard but not all NP-Hard problems are not NP-Complete." NP-Complete problems are subclass of NP-Hard.
4. P problems are quick to solve
5. NP problems are quick to verify but slow to solve
6. NP-complete problems are also quick to verify, slow to solve and can be reduced to any other **NP-complete problem**
7. NP-hard problems are slow to verify, slow to solve and can be reduced to any other NP problem
8. As a final note, if P=NP has proof in the future, humankind has to construct a new way of security aspects of the computer era. When this happens, there has to be another complexity level to identify new hardness levels than we have currently.


Thank you for your attention

Exercises

Exercises

1. What are the classes of NP problem?
2. How do you show a problem is NP hard?
3. Is every NP hard problem NP complete?
4. What are the differences between NP, NP-Complete and NP-Hard?