**2017**

**Question:**

Describe an approximate solution to an NP-complete problem. Why would you prefer an approximation method to solve the Travelling Salesman Problem (TSP)?

**Answer:**

1. **Approximate Solution to an NP-Complete Problem**: An NP-complete problem is a type of computational problem for which no known polynomial-time solution exists. Solving such problems exactly can be very time-consuming, especially as the size of the input grows. An approximate solution to an NP-complete problem provides a solution that is close to the optimal one but may not be exact. Approximation algorithms are used to find solutions within a guaranteed bound of the optimal solution in polynomial time.

In approximate solutions, the algorithm is designed to deliver a solution that is within a certain factor of the optimal solution. For example, if an algorithm for a minimization problem is within a factor of 2, it means the algorithm's output will be at most twice as bad as the optimal solution.

1. **Why Use an Approximation Method for the Travelling Salesman Problem**: The Travelling Salesman Problem (TSP) is an NP-hard problem where the objective is to find the shortest possible route that visits each city once and returns to the starting city. Finding the exact optimal solution for TSP requires checking all possible routes, which is computationally infeasible for large instances due to factorial time complexity.

Approximation methods are preferred for TSP because:

* + They provide a solution in polynomial time, making it feasible to handle larger problem instances.
  + They produce a near-optimal solution that is often close enough for practical applications, such as logistics and route planning.
  + Exact algorithms for TSP are too slow for real-world applications involving a large number of cities.

Approximation algorithms, such as the Nearest Neighbor and Christofides’ algorithm (which guarantees a solution within 1.5 times the optimal for metric TSP), are commonly used for TSP to strike a balance between solution quality and computational efficiency.

**Question 10 (c): Discuss NP-Completeness and NP-Hardness with the help of a hypothetical example.**

**NP-Completeness and NP-Hardness**

* **NP (Nondeterministic Polynomial time)**: A class of problems for which a solution can be verified in polynomial time.
* **NP-Complete**: A problem is NP-complete if it is in NP and as hard as any problem in NP, meaning any NP problem can be reduced to it in polynomial time. If an NP-complete problem can be solved in polynomial time, all problems in NP can also be solved in polynomial time.
* **NP-Hard**: A problem is NP-hard if it is at least as hard as the hardest problems in NP. However, NP-hard problems are not required to be in NP, which means they may not have verifiable solutions in polynomial time.

**Hypothetical Example:**

Consider the **Traveling Salesman Problem (TSP)**:

* **Problem Statement**: Given a set of cities and distances between them, find the shortest route that visits each city exactly once and returns to the starting city.
* **NP-Completeness**: TSP is NP-hard, and for the variant where all distances are symmetric and satisfy the triangle inequality, it is NP-complete.
* **NP-Hardness**: Even if finding the optimal solution is difficult, an approximate solution can be found. Solving TSP exactly for large instances is infeasible, but approximate algorithms can provide near-optimal solutions in reasonable time.

The complexity of TSP illustrates both NP-hardness and NP-completeness concepts.

**Question 3:**

**What are approximation algorithms? What is meant by a P(n)-approximation algorithm? Give an approximation algorithm for the Travelling Salesman Problem.**

**Answer:**

* **Approximation Algorithms**: Approximation algorithms are algorithms used for NP-hard optimization problems, where finding an exact solution in polynomial time is infeasible. They provide solutions that are close to the optimal solution within a known factor.
* **P(n)-Approximation Algorithm**: A P(n)-approximation algorithm for an optimization problem guarantees that the solution it provides is within a factor of P(n) of the optimal solution. For a minimization problem, if the optimal solution has a value OPTOPTOPT, the approximation algorithm will yield a solution with a value SOLSOLSOL such that: OPT≤SOL≤P(n)×OPTOPT \leq SOL \leq P(n) \times OPTOPT≤SOL≤P(n)×OPT
* **Approximation Algorithm for the Travelling Salesman Problem (TSP)**:
  + For the metric TSP (where the distance satisfies the triangle inequality), Christofides' algorithm is a common approximation algorithm.
  + **Christofides' Algorithm** guarantees a solution within 1.5 times the optimal for metric TSP by constructing a minimum spanning tree, finding an Eulerian circuit, and creating a Hamiltonian path from it.

Let's go through each question from the image, convert them to text, and provide solutions.

**Question 3:**

**What are approximation algorithms? What is meant by a P(n)-approximation algorithm? Give an approximation algorithm for the Travelling Salesman Problem.**

**Answer:**

* **Approximation Algorithms**: Approximation algorithms are algorithms used for NP-hard optimization problems, where finding an exact solution in polynomial time is infeasible. They provide solutions that are close to the optimal solution within a known factor.
* **P(n)-Approximation Algorithm**: A P(n)-approximation algorithm for an optimization problem guarantees that the solution it provides is within a factor of P(n) of the optimal solution. For a minimization problem, if the optimal solution has a value OPTOPTOPT, the approximation algorithm will yield a solution with a value SOLSOLSOL such that: OPT≤SOL≤P(n)×OPTOPT \leq SOL \leq P(n) \times OPTOPT≤SOL≤P(n)×OPT
* **Approximation Algorithm for the Travelling Salesman Problem (TSP)**:
  + For the metric TSP (where the distance satisfies the triangle inequality), Christofides' algorithm is a common approximation algorithm.
  + **Christofides' Algorithm** guarantees a solution within 1.5 times the optimal for metric TSP by constructing a minimum spanning tree, finding an Eulerian circuit, and creating a Hamiltonian path from it.

**Question 4:**

**Explain various asymptotic methods used to represent the rate of growth of running time of an algorithm. Show that 2n2+3n+1=Θ(n2)2n^2 + 3n + 1 = \Theta(n^2)2n2+3n+1=Θ(n2).**

**Answer:**

* **Asymptotic Notations**:
  1. **Big O Notation (O)**: Represents an upper bound on the growth rate of a function. f(n)=O(g(n))f(n) = O(g(n))f(n)=O(g(n)) means that f(n)f(n)f(n) grows at most as fast as g(n)g(n)g(n).
  2. **Omega Notation (Ω)**: Represents a lower bound on the growth rate. f(n)=Ω(g(n))f(n) = \Omega(g(n))f(n)=Ω(g(n)) means that f(n)f(n)f(n) grows at least as fast as g(n)g(n)g(n).
  3. **Theta Notation (Θ)**: Represents a tight bound on the growth rate. f(n)=Θ(g(n))f(n) = \Theta(g(n))f(n)=Θ(g(n)) means that f(n)f(n)f(n) grows asymptotically as fast as g(n)g(n)g(n).
* **Proof** that 2n2+3n+1=Θ(n2)2n^2 + 3n + 1 = \Theta(n^2)2n2+3n+1=Θ(n2):
  1. Let f(n)=2n2+3n+1f(n) = 2n^2 + 3n + 1f(n)=2n2+3n+1.
  2. For large nnn, the n2n^2n2 term dominates, so f(n)f(n)f(n) grows at the same rate as n2n^2n2.
  3. We can find constants c1,c2c\_1, c\_2c1​,c2​ and n0n\_0n0​ such that: c1n2≤2n2+3n+1≤c2n2c\_1 n^2 \leq 2n^2 + 3n + 1 \leq c\_2 n^2c1​n2≤2n2+3n+1≤c2​n2 For sufficiently large nnn, constants c1=2c\_1 = 2c1​=2 and c2=3c\_2 = 3c2​=3 work, proving that f(n)=Θ(n2)f(n) = \Theta(n^2)f(n)=Θ(n2).

**Question 5:**

**Show that the lower bound for any comparison-based sorting algorithm is Ω(nlog⁡n)\Omega(n \log n)Ω(nlogn). Draw a valid decision tree for sorting three items.**

**Answer:**

* **Lower Bound for Comparison-Based Sorting**:
  + For a list of nnn elements, there are n!n!n! possible permutations (arrangements).
  + A comparison-based sorting algorithm must be able to distinguish between each of these permutations.
  + In a decision tree representing the sorting process, each leaf corresponds to a permutation.
  + The height of this tree must be at least log⁡2(n!)\log\_2(n!)log2​(n!).
  + Using Stirling’s approximation, log⁡2(n!)≈nlog⁡n\log\_2(n!) \approx n \log nlog2​(n!)≈nlogn, so the minimum height is Ω(nlog⁡n)\Omega(n \log n)Ω(nlogn), which is the lower bound.
* **Decision Tree for Sorting Three Items (A, B, C)**:
  + Compare A and B:
    - If A < B, compare B and C:
      * If B < C: A < B < C
      * If C < B: A < C < B
    - If B < A, compare A and C:
      * If A < C: B < A < C
      * If C < A: B < C < A
  + Continue similarly for all cases.

**Question 6:**

**Differentiate between BFS and DFS traversal with the help of an example.**

**Answer:**

* **BFS (Breadth-First Search)**:
  + Explores nodes level by level, starting from a source node and visiting all neighbors before moving to the next level.
  + Useful for finding the shortest path in an unweighted graph.
  + **Example**: For a graph with nodes A → B, A → C, B → D:
    - BFS traversal starting from A: A, B, C, D.
* **DFS (Depth-First Search)**:
  + Explores as far as possible along a branch before backtracking.
  + Useful for topological sorting, detecting cycles, and solving maze problems.
  + **Example**: For the same graph:
    - DFS traversal starting from A: A, B, D, C.

**Question 7:**

**Differentiate between P, NP, NP-complete, and NP-hard problems. Give at least five problems that can be classified as NP problems. Discuss the approximation algorithms for NP-hard problems.**

**Answer:**

* **P**: Class of problems that can be solved in polynomial time by a deterministic Turing machine. Example: Sorting algorithms.
* **NP**: Class of problems whose solutions can be verified in polynomial time. Example: Hamiltonian cycle.
* **NP-Complete**: Problems that are both in NP and as hard as any problem in NP. If an NP-complete problem has a polynomial-time solution, all NP problems do. Example: Traveling Salesman Problem (TSP).
* **NP-Hard**: Problems that are at least as hard as NP problems but may not be in NP. They do not have to be decision problems. Example: Halting Problem.
* **Examples of NP Problems**:
  1. Traveling Salesman Problem (TSP)
  2. Knapsack Problem
  3. Hamiltonian Path Problem
  4. Subset Sum Problem
  5. Graph Coloring Problem
* **Approximation Algorithms for NP-Hard Problems**:
  1. Approximation algorithms provide near-optimal solutions in polynomial time.
  2. For example, **Greedy algorithms** for the Knapsack Problem and **Christofides' algorithm** for TSP offer solutions within a known factor of the optimal.

4o

**2018**

**Question 7:**

**Discuss the difference between P, NP, NPC, and NPH problems. Explain with examples.**

**Answer:**

* **P (Polynomial Time)**:
  + Class of problems that can be solved in polynomial time by a deterministic Turing machine.
  + Example: Sorting algorithms like QuickSort or MergeSort.
* **NP (Nondeterministic Polynomial Time)**:
  + Class of problems for which a solution can be verified in polynomial time.
  + Example: The Hamiltonian Path problem (finding a path that visits each vertex exactly once).
* **NP-Complete (NPC)**:
  + Problems that are both in NP and as hard as any problem in NP, meaning any NP problem can be reduced to it in polynomial time.
  + Example: The Traveling Salesman Problem (TSP).
* **NP-Hard (NPH)**:
  + Problems that are at least as hard as the hardest problems in NP but are not necessarily in NP themselves.
  + Example: The Halting Problem (determining if a program will halt or run indefinitely).

**Question 7 (a):**

**Compare NP-Hard with NP-Complete problems with suitable examples. (5 Marks)**

**Answer:**

* **NP-Hard Problems**:
  + Definition: A problem is NP-Hard if it is at least as hard as the hardest problems in NP. However, NP-Hard problems are not required to be in NP, meaning they may not have verifiable solutions in polynomial time.
  + Characteristics:
    - Solving an NP-Hard problem in polynomial time would solve all NP problems in polynomial time.
    - NP-Hard problems do not have to be decision problems.
  + Example: The **Halting Problem** is NP-Hard. It involves determining if a program will halt or run indefinitely, which cannot be solved by any algorithm (undecidable).
* **NP-Complete Problems**:
  + Definition: A problem is NP-Complete if it is both in NP and as hard as any problem in NP. NP-Complete problems are a subset of NP problems.
  + Characteristics:
    - Solutions for NP-Complete problems can be verified in polynomial time.
    - If any NP-Complete problem has a polynomial-time solution, then all NP problems can be solved in polynomial time.
  + Example: The **Traveling Salesman Problem (TSP)** is NP-Complete when formulated as a decision problem (e.g., "Is there a tour shorter than a given length?").

**Summary**: NP-Complete problems are decision problems in NP that are as hard as any other problem in NP, while NP-Hard problems are at least as hard as NP-Complete problems but may not be in NP.

**Question 7 (b):**

**If any problem in NP cannot be solved by a polynomial-time deterministic algorithm, then NP-Complete problems are not in P. Justify the given statement.**

The statement given is an assertion about the relationship between classes **P**, **NP**, and **NP-Complete** in computational complexity theory, and it relates to the famous **P vs NP problem**. Here’s the reasoning behind the statement:

**Understanding P, NP, and NP-Complete**

1. **P**: The class **P** consists of problems that can be solved in polynomial time by a deterministic algorithm. In other words, problems in P can be solved "efficiently."
2. **NP**: The class **NP** consists of problems for which a given solution can be verified in polynomial time by a deterministic algorithm, even if finding that solution might not be easy. All problems in P are also in NP because if a problem can be solved in polynomial time, its solution can certainly be verified in polynomial time as well.
3. **NP-Complete**: An **NP-Complete** problem is one that:
   * Is in NP (solutions can be verified in polynomial time).
   * Is as hard as any other problem in NP, meaning every problem in NP can be transformed into this problem in polynomial time (it’s at least as hard as the hardest problems in NP).

If any NP-Complete problem can be solved in polynomial time, then all problems in NP can be solved in polynomial time, meaning **P = NP**.

**Justification of the Statement**

The statement says that if **any problem in NP cannot be solved by a polynomial-time deterministic algorithm**, then **NP-Complete problems are not in P**. Let’s break this down step-by-step:

1. **Assumption**: If any problem in NP cannot be solved by a polynomial-time deterministic algorithm, it means there exists at least one problem in NP that is not in P. This implies that **P ≠ NP**, as not all NP problems would be solvable in polynomial time.
2. **Implication for NP-Complete Problems**: If **P ≠ NP**, then NP-Complete problems cannot be in P either. This is because if any NP-Complete problem could be solved in polynomial time (i.e., if any NP-Complete problem were in P), then all problems in NP would also be in P (since any NP problem can be reduced to an NP-Complete problem). This would contradict our assumption that **P ≠ NP**.
3. **Conclusion**: Therefore, if there exists any problem in NP that cannot be solved in polynomial time, then NP-Complete problems, being the hardest problems in NP, also cannot be solved in polynomial time. In other words, **NP-Complete problems are not in P**.

**Summary**

The statement is essentially describing the logical implications of **P ≠ NP**. If there is even a single problem in NP that cannot be solved in polynomial time, then **P ≠ NP**, which implies that no NP-Complete problem can be in P, as solving one NP-Complete problem in polynomial time would mean that all NP problems are in P. Thus, the statement is justified as a logical consequence of the assumption that **P ≠ NP**.

**Question 10 (c):**

**Prove that the SAT problem is NP-Complete. (5 Marks)**

**Answer:**

* **Definition of SAT (Boolean Satisfiability Problem)**:
  + The SAT problem involves determining if there exists an assignment of truth values to variables such that a given Boolean formula evaluates to true.
* **Proving NP-Completeness**:
  + **SAT is in NP**: Given an assignment of values to variables, we can evaluate the Boolean formula in polynomial time to verify if it satisfies the formula. Therefore, SAT is in NP.
  + **SAT is NP-Hard**:
    - Every problem in NP can be reduced to SAT in polynomial time. This was proven by Stephen Cook in 1971 (Cook’s Theorem).
    - Any problem in NP can be transformed into an instance of SAT, meaning that SAT is at least as hard as any problem in NP.
  + **Conclusion**:
    - Since SAT is both in NP and NP-Hard, it is NP-Complete

1 q screenshot

2021

**Question 5:**

**Differentiate between P, NP, NP-Complete, and NP-Hard Problems. Also, give the relationship between each of the classes using a suitable diagram.**

**Answer:**

1. **P (Polynomial Time)**:
   * Class of problems that can be solved by a deterministic Turing machine in polynomial time.
   * Example: Sorting algorithms, such as Merge Sort and QuickSort.
2. **NP (Nondeterministic Polynomial Time)**:
   * Class of problems for which a solution can be verified in polynomial time by a deterministic Turing machine.
   * Example: The Hamiltonian Path problem (finding a path that visits each vertex exactly once).
3. **NP-Complete**:
   * A problem is NP-Complete if it is both in NP and as hard as any problem in NP. That is, every problem in NP can be reduced to an NP-Complete problem in polynomial time.
   * Example: The Traveling Salesman Problem (when formulated as a decision problem).
4. **NP-Hard**:
   * A problem is NP-Hard if it is at least as hard as the hardest problems in NP. NP-Hard problems do not have to be in NP, meaning they may not be decision problems.
   * Example: The Halting Problem.

**Relationship Diagram:**

A typical relationship diagram is as follows:

lua

Copy code

+---------------------+

| NP |

| +--------+ |

| | P | |

| +--------+ |

| |

| +---------------------+

| | NP-Complete |

| +---------------------+

+---------------------+

+---------------------+

| NP-Hard |

+---------------------+

* P is a subset of NP.
* NP-Complete is a subset of NP and NP-Hard.
* NP-Hard includes problems that are as hard as or harder than NP problems, but they may not be verifiable in polynomial time.

**Question 9 (c):**

**Explain Cook’s Theorem for NP-complete problems. Show that the Clique Decision Problem (CDP) is NP-complete.**

**Answer:**

1. **Cook's Theorem**:
   * Cook's Theorem states that the Boolean satisfiability problem (SAT) is NP-complete. This theorem was the first to establish the concept of NP-completeness and showed that every problem in NP can be reduced to SAT in polynomial time.
   * This means that if SAT can be solved in polynomial time, then all problems in NP can also be solved in polynomial time.
2. **Proving Clique Decision Problem (CDP) is NP-complete**:
   * **Definition of CDP**: The Clique Decision Problem asks if there exists a subset of vertices of a given size kkk in a graph such that each pair of vertices in this subset is connected by an edge (i.e., it forms a complete subgraph or clique).
   * **Step 1: CDP is in NP**:
     + A solution to CDP can be verified in polynomial time by checking if the subset of vertices forms a clique of size kkk.
   * **Step 2: Reduction from SAT to CDP**:
     + It can be shown that SAT can be transformed (reduced) to CDP in polynomial time, meaning any instance of SAT can be converted to an instance of CDP.
   * **Conclusion**: Since CDP is in NP and any NP problem can be reduced to it, CDP is NP-complete.

2022 hamiltonian path proving in option

**Question (b):**

**Discuss the relationship between the classes P, NP, NP-complete, and NP-hard problems with suitable examples of each class.**

**Answer:**

1. **P (Polynomial Time)**:
   * Class P consists of problems that can be solved by a deterministic Turing machine in polynomial time.
   * **Example**: Sorting algorithms (e.g., Merge Sort), finding the shortest path in a graph (e.g., Dijkstra’s algorithm).
2. **NP (Nondeterministic Polynomial Time)**:
   * Class NP consists of problems for which a solution can be verified in polynomial time by a deterministic Turing machine.
   * **Example**: Hamiltonian Path, where given a path, we can verify if it visits every vertex exactly once.
3. **NP-Complete**:
   * A problem is NP-Complete if it is both in NP and as hard as any problem in NP. In other words, every problem in NP can be reduced to an NP-complete problem in polynomial time.
   * **Example**: The Traveling Salesman Problem (in its decision form), where we ask if there exists a tour of a given maximum length.
4. **NP-Hard**:
   * A problem is NP-Hard if it is at least as hard as the hardest problems in NP, but it doesn’t have to be in NP (i.e., it may not be verifiable in polynomial time).
   * **Example**: The Halting Problem, which cannot be verified or solved in polynomial time.

**Relationship**:

* **P** is a subset of **NP**.
* **NP-Complete** problems are a subset of **NP** and are the hardest problems in **NP**.
* **NP-Hard** problems are as hard as or harder than **NP-Complete** problems, but they may not be in **NP**.

Differentiate between P, NP, NP-Complete, and NP-Hard problems with a suitable diagram. Let L be an NP-complete problem, and Q and R be two other problems not known to be in NP. Q is polynomial-time reducible to L, and L is polynomial-time reducible to R. What can you say about problem R?

**Question 8:**

**(a) Prove that the CLIQUE problem is NP-Complete.**

* **Proof Outline**: Show that CLIQUE is in NP and reduce a known NP-complete problem (e.g., Vertex Cover) to CLIQUE in polynomial time.

2023

**Differentiation between P, NP, NP-Complete, and NP-Hard Classes of Problems**

1. **P (Polynomial Time)**:
   * Class **P** includes problems that can be solved in polynomial time by a deterministic algorithm. These problems are considered "tractable" and efficiently solvable.
   * **Examples**:
     + **Sorting**: Sorting a list of numbers (e.g., using merge sort or quicksort).
     + **Shortest Path Problem**: Finding the shortest path in a graph with non-negative weights (e.g., using Dijkstra’s algorithm).
2. **NP (Nondeterministic Polynomial Time)**:
   * Class **NP** includes problems for which a given solution can be verified in polynomial time by a deterministic algorithm.
   * Problems in P are also in NP, as a problem that can be solved in polynomial time can also have its solution verified in polynomial time.
   * **Examples**:
     + **Hamiltonian Path Problem**: Determining if there exists a path in a graph that visits each vertex exactly once.
     + **Subset Sum Problem**: Determining if there is a subset of a set of integers that adds up to a specific target.
3. **NP-Complete**:
   * A problem is **NP-Complete** if:
     + It is in NP (solutions can be verified in polynomial time).
     + It is as hard as any problem in NP, meaning that every problem in NP can be reduced to it in polynomial time.
   * If any NP-Complete problem can be solved in polynomial time, then all NP problems can also be solved in polynomial time, implying **P = NP**.
   * **Examples**:
     + **SAT (Boolean Satisfiability Problem)**: Determining if there exists a truth assignment to variables in a Boolean formula that makes the formula true.
     + **Traveling Salesman Problem (Decision Version)**: Determining if there exists a tour of cities with a total distance less than or equal to a given limit.
4. **NP-Hard**:
   * A problem is **NP-Hard** if it is at least as hard as any problem in NP, meaning that every NP problem can be reduced to it in polynomial time.
   * NP-Hard problems do not need to be in NP; they may not even have solutions that can be verified in polynomial time (e.g., optimization problems).
   * **Examples**:
     + **Traveling Salesman Problem (Optimization Version)**: Finding the shortest possible tour that visits each city exactly once and returns to the starting city.
     + **Knapsack Problem (Optimization Version)**: Finding the most valuable subset of items that fit within a weight limit.

**Diagram: Relationships Between P, NP, NP-Complete, and NP-Hard**

A Venn diagram can help illustrate these relationships:

lua

Copy code

NP

+-------------------------+

| +---------+ |

| | P | |

| +---------+ |

| NP-Complete |

+-------------------------+ NP-Hard

\\_\_\_\_\_\_\_\_\_\_\_/

* **P** is a subset of **NP** (problems that can be both solved and verified in polynomial time).
* **NP-Complete** problems are within **NP** and are the hardest problems in NP.
* **NP-Hard** problems include NP-Complete problems as well as other problems that may not be in NP (e.g., optimization problems).

Show that the Vertex-Cover Problem (VCP) is NP-complete.

**Design a framework for P, NP, NP-Complete, and NP-Hard problems that measure complexity.**

**Solution:**

1. **P (Polynomial Time)**:
   * Problems that can be solved in polynomial time.
   * Example: Sorting, finding the maximum of a list.
2. **NP (Non-deterministic Polynomial Time)**:
   * Problems for which a solution can be verified in polynomial time.
   * Example: Traveling Salesman Problem (verification of a solution path).
3. **NP-Complete**:
   * Problems that are both in NP and as hard as any problem in NP.
   * Example: Traveling Salesman Problem (finding the optimal path).
4. **NP-Hard**:
   * Problems that are at least as hard as NP-Complete problems but may not be in NP.
   * Example: Halting Problem.

Also generated on separate chat

**(b) Design and analyze the algorithm to solve the Vertex Cover problem using the approximation algorithm.**

**Solution:**

1. **Algorithm**:
   * Start with an empty cover.
   * While there are edges in the graph:
     + Pick an arbitrary edge (u,v)(u, v)(u,v).
     + Add both uuu and vvv to the vertex cover.
     + Remove all edges incident to uuu and vvv.
2. **Complexity**:
   * Time complexity: O(E)O(E)O(E), where EEE is the number of edges.
   * This algorithm gives a 2-approximation, meaning the cover size is at most twice the optimal.