Module 3

(TA ASSIGNMENT)

Contents

Greedy Method: 0/1 knapsack Problem, Job Sequencing with Deadlines, Minimum Spanning Trees, Optimal Sub-Structure.

Name: TANISH CHOURASIA

Roll No.: 230101144

Branch: CSE

Submitted To: Dr. Ujjwal Biswas

Topic-1

1. The Greedy Method
    1. Definition and Key Idea

The greedy method is an algorithmic strategy that makes a series of choices, each of which looks the best at the moment (i.e., locally optimal). The main principle is to make a choice that seems best immediately, without worrying about the consequences of future choices. The hope is that by selecting local optima, a global optimum will eventually be reached.

1.2 Characteristics of Greedy Algorithms

-
    - **Local Optimization**: Each step in the algorithm takes the best immediate or local solution.
    - **Feasibility**: Ensure that the selected option is a feasible solution.
    - **Optimal Substructure**: The optimal solution to a problem contains optimal solutions to subproblems.
    - **Non-retractable**: Once a choice is made, it can't be undone.

1.3 Common Greedy Problems

-
    - **Activity Selection Problem**: Selecting the maximum number of non-overlapping activities from a set.
    - **Huffman Coding**: Building an optimal prefix code for lossless data compression.
    - **Fractional Knapsack**: Choosing parts of items to maximize the profit-to-weight ratio.

1.4 Example: Activity Selection Problem

Given n activities with start and finish times, find the maximum number of activities that can be performed by a single person or machine. The greedy choice is to select the activity with the earliest finish time and recursively schedule the remaining activities.

Algorithm:

Sort the activities by their finishing times.

Select the activity that finishes first.

Select the next activity that starts after the current activity finishes.

Repeat until no more activities are left.

2. The 0/1 Knapsack Problem

2.1 Problem Definition

The 0/1 Knapsack Problem is a combinatorial optimization problem where you are given a set of items, each with a weight and a value, and a knapsack with a capacity limit. The goal is to maximize the total value by selecting a subset of items, such that the total weight does not exceed the capacity of the knapsack. The 0/1 nature implies that each item can either be taken entirely or left behind (no fractions).

2.2 Problem Statement

Given:

- A set of items {i1, i2, ..., in} where each item i has a weight wi and value vi.
- A knapsack with a maximum weight capacity W.

Objective:

- Maximize the sum of the values of the selected items, ensuring the sum of their weights doesn't exceed W.

2.3 Algorithm (Dynamic Programming Approach)

The 0/1 Knapsack Problem can't be solved by the greedy method due to the integer nature of the problem. Instead, a dynamic programming solution is typically used:

1. Create a DP table where dp[i][j] represents the maximum value obtainable with the first i items and a knapsack capacity of j.
2. Recurrence relation:
    - If item i is not included: dp[i][j] = dp[i-1][j]
    - If item i is included: dp[i][j] = max(dp[i-1][j], dp[i-1][j-wi] + vi)
3. The answer is in dp[n][W] where n is the number of items and W is the capacity.

2.4 Example

Suppose you have three items with the following weights and values:

- Item 1: Weight = 1, Value = 1
- Item 2: Weight = 3, Value = 4
- Item 3: Weight = 4, Value = 5
- Knapsack capacity: 7

Using dynamic programming, you can calculate the maximum value as 9.

3. Job Sequencing with Deadlines

3.1 Problem Definition

Job sequencing with deadlines is a problem where you are given a set of jobs, each associated with a deadline and a profit. The goal is to schedule jobs to maximize the total profit, ensuring that each job is completed before its deadline.

3.2 Problem Statement

Given:

- A set of jobs {J1, J2, ..., Jn} where each job Ji has a deadline di and profit pi.

Objective:

- Schedule the jobs in such a way that the total profit is maximized, and each job Ji is completed within its deadline di.

3.3 Greedy Algorithm

A common greedy approach for this problem is to:

1. Sort all jobs in decreasing order of their profits.
2. Iterate through the sorted jobs, trying to schedule each one as close to its deadline as possible.
3. If a time slot is already occupied, move to an earlier available slot.

3.4 Example

Suppose you have four jobs with the following deadlines and profits:

- Job 1: Deadline = 4, Profit = 100
- Job 2: Deadline = 1, Profit = 19
- Job 3: Deadline = 2, Profit = 27
- Job 4: Deadline = 1, Profit = 25

By following the greedy approach, you can schedule the jobs to maximize total profit, ensuring jobs are completed before their deadlines.

4. Minimum Spanning Trees (MST)

4.1 Problem Definition

A Minimum Spanning Tree is a subset of the edges of a connected, undirected graph that connects all the vertices with the minimum possible total edge weight and without any cycles.

4.2 Problem Statement

Given:

- A graph G = (V, E) where V is a set of vertices and E is a set of edges with weights.

Objective:

- Find a subset of edges that form a tree connecting all vertices, such that the sum of the weights is minimized.

4.3 Algorithms for MST

Two popular algorithms are used to find an MST:

- **Prim's Algorithm**: A greedy algorithm that starts with a single vertex and repeatedly adds the minimum-weight edge that connects a vertex in the tree to a vertex outside the tree.
- **Kruskal's Algorithm**: A greedy algorithm that sorts all the edges in increasing order of their weights and repeatedly adds the smallest edge, ensuring no cycles are formed.

4.4 Example

Consider a graph with 4 vertices and 5 edges with the following weights:

- Edge 1-2: Weight = 1
- Edge 1-3: Weight = 4
- Edge 2-3: Weight = 2
- Edge 2-4: Weight = 3
- Edge 3-4: Weight = 5

By using Kruskal's or Prim's algorithm, you can construct the MST with the minimum total weight of 6.

5. Optimal Sub-Structure

5.1 Definition

Optimal substructure means that an optimal solution to a problem can be constructed efficiently from optimal solutions to its subproblems. This property is crucial for dynamic programming and greedy algorithms.

5.2 Example of Optimal Substructure

Consider the **shortest path problem** in a graph:

- If the shortest path from vertex A to vertex C passes through vertex B, then the subpath from A to B and from B to C must also be the shortest paths for those segments.

Thus, dynamic programming (like in Dijkstra's algorithm) can exploit this optimal substructure property to solve problems efficiently.

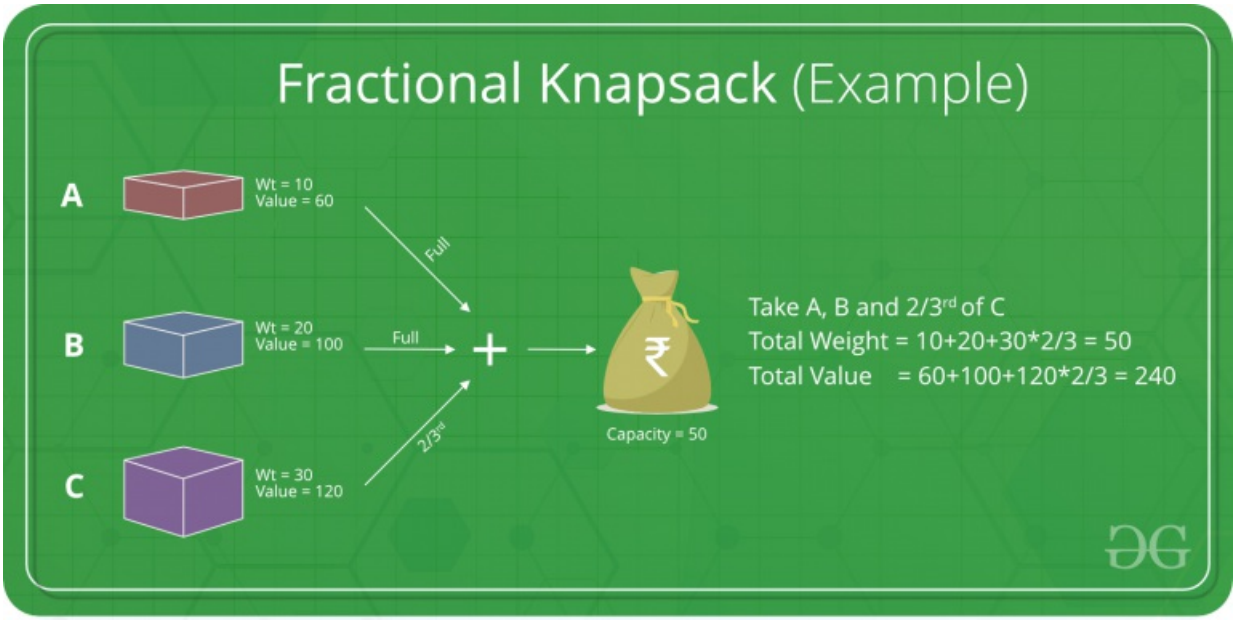
5.3 Applications

- **Dynamic Programming**: Uses optimal substructure to break problems down into smaller overlapping subproblems (e.g., 0/1 Knapsack, Fibonacci sequence).
- **Greedy Algorithms**: Exploits local optimal choices that lead to a globally optimal solution (e.g., Activity Selection, Huffman Coding.

Greedy Algorithms

Greedy algorithms are a class of algorithms that make **locally optimal** choices at each step with the hope of finding a **global optimum** solution. In these algorithms, decisions are made based on the information available at the current moment without considering the consequences of these decisions in the future. The key idea is to select the best possible choice at each step, leading to a solution that may not always be the most optimal but is often good enough for many problems.

In this article, we will understand greedy algorithms with examples. We will also look at problems and their solutions using the greedy approach.



What is Greedy Algorithm?

A **greedy algorithm** is a type of optimization algorithm that makes locally optimal choices at each step to find a globally optimal solution. It operates on the principle of "taking the best option now" without considering the long-term consequences.

To learn what is greedy method and how to use the greedy approach, read the given tutorial on the Greedy Algorithm:

Steps for Creating a Greedy Algorithm

The steps to define a greedy algorithm are:

1. **Define the problem:** Clearly state the problem to be solved and the objective to be optimized.
2. **Identify the greedy choice:** Determine the locally optimal choice at each step based on the current state.
3. **Make the greedy choice:** Select the greedy choice and update the current state.
4. **Repeat:** Continue making greedy choices until a solution is reached.

Following the given steps, one can learn how to use greedy algorithms to find optimal solutions.

Greedy Algorithm Examples

Examples of greedy algorithms are the best way to understand the algorithm. Some greedy algorithm real-life examples are:

- **Fractional Knapsack:** Optimizes the value of items that can be fractionally included in a knapsack with limited capacity.
- **Dijkstra's algorithm:** Finds the shortest path from a source vertex to all other vertices in a weighted graph.
- **Kruskal's algorithm:** Finds the minimum spanning tree of a weighted graph.
- **Huffman coding:** Compresses data by assigning shorter codes to more frequent symbols.

Applications of Greedy Algorithm

There are many **applications of the greedy method in DAA**. Some important greedy algorithm applications are:

- Assigning tasks to resources to minimize waiting time or maximize efficiency.
- Selecting the most valuable items to fit into a knapsack with limited capacity.
- Dividing an image into regions with similar characteristics.
- Reducing the size of data by removing redundant information.

Disadvantages/Limitations of Using a Greedy Algorithm

Below are some disadvantages of the Greedy Algorithm:

- Greedy algorithms may not always find the best possible solution.
- The order in which the elements are considered can significantly impact the outcome.
- Greedy algorithms focus on local optimizations and may miss better solutions that require considering a broader context.
- Greedy algorithms are not applicable to problems where the greedy choice does not lead to an optimal solution.

0/1 Knapsack Peoblem / Fractional Knapsack Problem

1. Problem Definition

The 0/1 Knapsack Problem is a classic problem in combinatorial optimization. The problem is defined as follows: given a set of n items, each with a weight and a value, and a knapsack that can carry a limited weight capacity W, determine the maximum value that can be obtained by selecting a subset of the items without exceeding the weight capacity. The 0/1 nature of the problem implies that each item can either be included entirely (1) or excluded (0); fractional parts of items cannot be included.

2. Problem Statement

Let:

- n be the number of items.
- w_i be the weight of item i.
- v_i be the value of item i.
- W be the total weight capacity of the knapsack.

The objective is to maximize the total value V:

$$V = \sum_{i=1}^{n} v_i \times x_i$$

Subject to the constraint:

$$\sum_{i=1}^{n} w_i \times x_i \leq W$$

Where `xi ∈ {0,1}` (either take the item `i` or leave it).

3. Algorithm to Solve the 0/1 Knapsack Problem

The 0/1 Knapsack Problem cannot be solved efficiently by a greedy method because the greedy approach doesn't always lead to an optimal solution. Instead, it is typically solved using **dynamic programming (DP)** or **branch and bound**.

3.1 Dynamic Programming Approach

The dynamic programming solution to the 0/1 knapsack problem builds a DP table where:

- dp[i][w] represents the maximum value obtainable with the first i items and a knapsack capacity of w.

Steps:

1. Initialize a 2D array dp[i][w], where i is the item index and w is the current capacity.
2. For each item i and for each capacity w from 0 to W, use the following recurrence relation:
    - If the weight of item i is greater than the current capacity w, skip the item:

$$dp[i][w] = dp[i-1][w]$$

Otherwise, choose the maximum value between including and excluding the item:

$$dp[i][w] = \max(dp[i-1][w], dp[i-1][w-w_i] + v_i)$$

After processing all items, the value at dp[n][W] will be the maximum value that can be carried in the knapsack.

Example:

Consider 3 items with the following weights and values:

- Item 1: Weight = 1, Value = 1
- Item 2: Weight = 3, Value = 4
- Item 3: Weight = 4, Value = 5
- Knapsack capacity: W = 7

The DP table will compute the optimal solution, and the maximum value that can be carried in the knapsack will be 9.

3.2 Time Complexity:

The time complexity of the dynamic programming solution is O(n * W), where n is the number of items and W is the maximum weight capacity.

4. Example

Suppose you have the following items:

- Item 1: Weight = 2, Value = 3
- Item 2: Weight = 3, Value = 4
- Item 3: Weight = 4, Value = 5
- Item 4: Weight = 5, Value = 6
- Knapsack capacity = 8

Using dynamic programming, the maximum value that can be obtained in the knapsack is 9 (by selecting items 2 and 3).

3. Greedy Algorithm to Solve the Fractional Knapsack Problem

Unlike the 0/1 Knapsack Problem, the fractional knapsack problem can be solved efficiently using a greedy algorithm. The key idea is to prioritize items based on their value-to-weight ratio, often called "profit per unit weight."

Steps:

1. **Calculate the value-to-weight ratio (vi/wi) for each item.**
2. **Sort the items in decreasing order of their value-to-weight ratio.**
3. **Iterate through the sorted items:**
    - **If the current item can fit entirely in the knapsack, take it.**
    - **If it can't fit entirely, take a fraction of the item that can fit in the remaining capacity of the knapsack.**
4. **Stop when the knapsack reaches its weight capacity.**

3.1 Example

Consider a knapsack with a capacity of W = 50 and three items:

- **Item 1: Weight = 10, Value = 60 → Ratio = 6**
- **Item 2: Weight = 20, Value = 100 → Ratio = 5**
- **Item 3: Weight = 30, Value = 120 → Ratio = 4**

Sort the items by their value-to-weight ratio. The solution would be:

1. **Take all of Item 1 (10 units), contributing value 60.**
2. **Take all of Item 2 (20 units), contributing value 100.**
3. **Take 2/3 of Item 3 (since only 20 units can fit in the remaining capacity), contributing value 80.**

The total value in the knapsack is:

60+100+80=24060 + 100 + 80 = 24060+100+80=240

4. Time Complexity:

The time complexity of the greedy algorithm for the fractional knapsack problem is O(n log n) due to the sorting step.

Fractional Knapsack Problem

1. Problem Definition

The **fractional knapsack problem** is a variant of the knapsack problem where items can be broken down into smaller parts. In this version of the problem, you can take fractions of items rather than just entire items. The goal remains the same: maximize the total value without exceeding the weight capacity of the knapsack.

2. Problem Statement

Given:

- n items, each with a weight wi and a value vi.
- A knapsack with a maximum weight capacity W.

The objective is to maximize the total value by choosing parts or entire items such that the sum of their weights does not exceed the capacity W. **3. Greedy Algorithm to Solve the Fractional Knapsack Problem**

Unlike the 0/1 Knapsack Problem, the **fractional knapsack problem can be solved efficiently using a greedy algorithm**. The key idea is to prioritize items based on their value-to-weight ratio, often called "profit per unit weight."

Steps:

1. **Calculate the value-to-weight ratio** (vi/wi) for each item.
2. **Sort the items** in decreasing order of their value-to-weight ratio.
3. **Iterate through the sorted items**:
    - If the current item can fit entirely in the knapsack, take it.
    - If it can't fit entirely, take a fraction of the item that can fit in the remaining capacity of the knapsack.
4. Stop when the knapsack reaches its weight capacity.

3.1 Example

Consider a knapsack with a capacity of W = 50 and three items:

- Item 1: Weight = 10, Value = 60 → Ratio = 6
- Item 2: Weight = 20, Value = 100 → Ratio = 5
- Item 3: Weight = 30, Value = 120 → Ratio = 4

Sort the items by their value-to-weight ratio. The solution would be:

1. Take all of Item 1 (10 units), contributing value 60.
2. Take all of Item 2 (20 units), contributing value 100.
3. Take 2/3 of Item 3 (since only 20 units can fit in the remaining capacity), contributing value 80.

The total value in the knapsack is:

60+100+80=24060 + 100 + 80 = 24060+100+80=240

4. Time Complexity:

The time complexity of the greedy algorithm for the fractional knapsack problem is O(n log n) due to the sorting step.

5. Difference Between 0/1 Knapsack and Fractional Knapsack

Feature	0/1 Knapsack	Fractional Knapsack
Nature of Items	Items cannot be divided, either taken or not.	Items can be divided and taken in fractions.
Approach	Dynamic Programming or Branch and Bound.	Greedy Algorithm.
Optimal Solution	Greedy may not always give the optimal solution.	Greedy gives the optimal solution.
Time Complexity	O(n * W)	O(n log n)

6. Conclusion

The 0/1 Knapsack Problem and the Fractional Knapsack Problem are two well-known variations of the knapsack problem. The 0/1 Knapsack Problem is solved using dynamic programming due to its combinatorial nature, while the Fractional Knapsack Problem is solved using a greedy approach, which is more efficient when fractional solutions are allowed. Both are crucial in understanding different optimization techniques in algorithm design.

Job Sequencing Problem with Deadlines

1. Introduction

The **Job Sequencing Problem with Deadlines** is a classic optimization problem in computer science, typically framed within the context of scheduling and resource allocation. The problem involves selecting and scheduling a subset of jobs, each with a defined deadline and profit, such that the total profit is maximized. The jobs must be completed before their respective deadlines, and only one job can be performed at a time.

This problem is frequently encountered in real-world applications like task scheduling in operating systems, project management, and maximizing return on investments where deadlines and limited resources are involved.

2. Problem Statement

Given a set of n jobs where each job i has:

- **Profit (p_i)**: The profit earned by completing the job.
- **Deadline (d_i)**: The time by which the job must be completed.

The goal is to schedule these jobs in such a way that:

- The total profit is maximized.
- Each job is completed before its deadline.
- No two jobs overlap, meaning only one job can be performed at any given time.

In other words, the objective is to maximize the sum of profits from a subset of jobs while ensuring that the scheduled jobs are completed before their respective deadlines.

Example Problem:

Consider 5 jobs with the following profits and deadlines:

Job	Profit (p_i)	Deadline (d_i)
1	100	2
2	19	1
3	27	2
4	25	1
5	15	3

The objective is to select and schedule jobs to maximize total profit while respecting their deadlines.

Job Sequencing Problem with Deadlines

1. Introduction

The **Job Sequencing Problem with Deadlines** is a classic optimization problem in computer science, typically framed within the context of scheduling and resource allocation. The problem involves selecting and scheduling a subset of jobs, each with a defined deadline and profit, such that the total profit is maximized. The jobs must be completed before their respective deadlines, and only one job can be performed at a time.

This problem is frequently encountered in real-world applications like task scheduling in operating systems, project management, and maximizing return on investments where deadlines and limited resources are involved.

2. Problem Statement

Given a set of n jobs where each job i has:

- **Profit (p_i)**: The profit earned by completing the job.
- **Deadline (d_i)**: The time by which the job must be completed.

The goal is to schedule these jobs in such a way that:

- The total profit is maximized.
- Each job is completed before its deadline.
- No two jobs overlap, meaning only one job can be performed at any given time.

In other words, the objective is to maximize the sum of profits from a subset of jobs while ensuring that the scheduled jobs are completed before their respective deadlines.

Example Problem:

Consider 5 jobs with the following profits and deadlines:

Job	Profit (p_i)	Deadline (d_i)
1	100	2
2	19	1
3	27	2
4	25	1
5	15	3

The objective is to select and schedule jobs to maximize total profit while respecting their deadlines.

3. Greedy Approach to Solve the Job Sequencing Problem

The **Job Sequencing Problem with Deadlines** can be solved efficiently using a **greedy algorithm**. The greedy strategy for this problem works by making a locally optimal choice at each step in the hope of finding a globally optimal solution. In this case, the greedy approach involves scheduling jobs in decreasing order of profit and assigning them to the latest possible time slot before their respective deadlines.

Steps of the Greedy Algorithm:

1. **Sort the Jobs by Profit**:
   - First, sort the jobs in descending order of profit. The intuition behind this is to attempt to maximize profit by scheduling higher-profit jobs first.
2. **Schedule the Jobs**:
   - For each job in the sorted order, assign the job to the latest available time slot that is on or before its deadline.
   - If a time slot is already occupied, move to the next available earlier time slot.
   - If no time slot is available, skip the job.
3. **Return the Maximum Profit**:
   - Once all the jobs are scheduled (or skipped if necessary), sum up the profits of the scheduled jobs to get the maximum profit.

Greedy Algorithm in Detail:

Job Sequencing Problem with Deadlines

1. Introduction

The **Job Sequencing Problem with Deadlines** is a classic optimization problem in computer science, typically framed within the context of scheduling and resource allocation. The problem involves selecting and scheduling a subset of jobs, each with a defined deadline and profit, such that the total profit is maximized. The jobs must be completed before their respective deadlines, and only one job can be performed at a time.

This problem is frequently encountered in real-world applications like task scheduling in operating systems, project management, and maximizing return on investments where deadlines and limited resources are involved.

2. Problem Statement

Given a set of n jobs where each job i has:

- **Profit (p_i)**: The profit earned by completing the job.

- **Deadline (d_i)**: The time by which the job must be completed.

The goal is to schedule these jobs in such a way that:

- The total profit is maximized.
- Each job is completed before its deadline.
- No two jobs overlap, meaning only one job can be performed at any given time.

In other words, the objective is to maximize the sum of profits from a subset of jobs while ensuring that the scheduled jobs are completed before their respective deadlines.

Example Problem:

Consider 5 jobs with the following profits and deadlines:

Job	Profit (p_i)	Deadline (d_i)
1	100	2
2	19	1
3	27	2
4	25	1
5	15	3

The objective is to select and schedule jobs to maximize total profit while respecting their deadlines.

3. Greedy Approach to Solve the Job Sequencing Problem

The **Job Sequencing Problem with Deadlines** can be solved efficiently using a **greedy algorithm**. The greedy strategy for this problem works by making a locally optimal choice at each step in the hope of finding a globally optimal solution. In this case, the greedy approach involves scheduling jobs in decreasing order of profit and assigning them to the latest possible time slot before their respective deadlines.

Steps of the Greedy Algorithm:

1. **Sort the Jobs by Profit**:
   - First, sort the jobs in descending order of profit. The intuition behind this is to attempt to maximize profit by scheduling higher-profit jobs first.
2. **Schedule the Jobs**:
   - For each job in the sorted order, assign the job to the latest available time slot that is on or before its deadline.
   - If a time slot is already occupied, move to the next available earlier time slot.
   - If no time slot is available, skip the job.
3. **Return the Maximum Profit**:
   - Once all the jobs are scheduled (or skipped if necessary), sum up the profits of the scheduled jobs to get the maximum profit.

Greedy Algorithm in Detail:

Given n jobs, each with a deadline d_i and profit p_i, the greedy algorithm can be outlined as follows:

1. **Sort** all jobs in decreasing order of profit.
2. Initialize an array to keep track of time slots, where each slot corresponds to one unit of time.
3. For each job:
   - Find a free time slot from d_i down to 1.
   - If a free slot is found, schedule the job at that time slot.
4. Sum the profits of the scheduled jobs to get the maximum profit.

4. Example of Job Sequencing with Deadlines

Let's revisit the example with the following jobs:

Job	Profit (`p_i`)	Deadline (`d_i`)
1	100	2
2	19	1
3	27	2
4	25	1
5	15	3

Step-by-step Execution:

1. **Step 1: Sort Jobs by Profit**: First, sort the jobs by their profits in decreasing order:

Job	Profit (p_i)	Deadline (d_i)
1	100	2
3	27	2
4	25	1
2	19	1
5	15	3

1. **Step 2: Initialize Time Slots**:
    - Let's assume we have 3 time slots available (t1, t2, t3), where each slot represents a time unit (since the maximum deadline is 3).
    - Initialize an empty array to keep track of the job scheduled for each time slot.
2. **Step 3: Schedule the Jobs**:
    - **Job 1** (Profit = 100, Deadline = 2): Schedule it in time slot 2 (the latest available slot before its deadline).
    - **Job 3** (Profit = 27, Deadline = 2): Slot 2 is already taken, so schedule it in time slot 1 (the next available earlier slot).
    - **Job 4** (Profit = 25, Deadline = 1): Slot 1 is already taken, so skip this job.
    - **Job 2** (Profit = 19, Deadline = 1): Slot 1 is already taken, so skip this job.
    - **Job 5** (Profit = 15, Deadline = 3): Schedule it in time slot 3.
3. **Step 4: Calculate the Maximum Profit**: The jobs scheduled are:
    - Job 1 in slot 2.
    - Job 3 in slot 1.
    - Job 5 in slot 3.

The total profit is:

100+27+15=142100 + 27 + 15 = 142100+27+15=142

Solution:

The maximum profit is 142 by scheduling jobs 1, 3, and 5.

5. Time Complexity of the Greedy Algorithm

The time complexity of the greedy algorithm can be broken down as follows:

- Sorting the jobs by profit takes **O(n log n)**.
- Scheduling each job requires checking the available slots, which takes **O(n)** in the worst case.

Thus, the overall time complexity of the greedy algorithm is **O(n log n)**.

6. Limitation of the Greedy Approach

While the greedy approach works efficiently for the Job Sequencing Problem with Deadlines, it has limitations in more complex variations of the problem. The greedy algorithm doesn't always guarantee an optimal solution for all types of scheduling problems, especially when the constraints or objectives change (such as when the jobs have varying lengths or dependencies).

For more complex scheduling problems, other optimization techniques like **dynamic programming**, **branch and bound**, or **backtracking** may be required.

Job Sequencing Problem with Deadlines

1. Introduction

The **Job Sequencing Problem with Deadlines** is a classic optimization problem in computer science, typically framed within the context of scheduling and resource allocation. The problem involves selecting and scheduling a subset of jobs, each with a defined deadline and profit, such that the total profit is maximized. The jobs must be completed before their respective deadlines, and only one job can be performed at a time.

This problem is frequently encountered in real-world applications like task scheduling in operating systems, project management, and maximizing return on investments where deadlines and limited resources are involved.

2. Problem Statement

Given a set of n jobs where each job i has:

- **Profit (p_i)**: The profit earned by completing the job.
- **Deadline (d_i)**: The time by which the job must be completed.

The goal is to schedule these jobs in such a way that:

- The total profit is maximized.
- Each job is completed before its deadline.
- No two jobs overlap, meaning only one job can be performed at any given time.

In other words, the objective is to maximize the sum of profits from a subset of jobs while ensuring that the scheduled jobs are completed before their respective deadlines.

Example Problem:

Consider 5 jobs with the following profits and deadlines:

Job	Profit (p_i)	Deadline (d_i)
1	100	2
2	19	1
3	27	2
4	25	1
5	15	3

The objective is to select and schedule jobs to maximize total profit while respecting their deadlines.

3. Greedy Approach to Solve the Job Sequencing Problem

The **Job Sequencing Problem with Deadlines** can be solved efficiently using a **greedy algorithm**. The greedy strategy for this problem works by making a locally optimal choice at each step in the hope of finding a globally optimal solution. In this case, the greedy approach involves scheduling jobs in decreasing order of profit and assigning them to the latest possible time slot before their respective deadlines.

Steps of the Greedy Algorithm:

1. **Sort the Jobs by Profit**:
   - First, sort the jobs in descending order of profit. The intuition behind this is to attempt to maximize profit by scheduling higher-profit jobs first.
2. **Schedule the Jobs**:
   - For each job in the sorted order, assign the job to the latest available time slot that is on or before its deadline.
   - If a time slot is already occupied, move to the next available earlier time slot.
   - If no time slot is available, skip the job.
3. **Return the Maximum Profit**:
   - Once all the jobs are scheduled (or skipped if necessary), sum up the profits of the scheduled jobs to get the maximum profit.

Greedy Algorithm in Detail:

Given n jobs, each with a deadline d_i and profit p_i, the greedy algorithm can be outlined as follows:

1. **Sort** all jobs in decreasing order of profit.
2. Initialize an array to keep track of time slots, where each slot corresponds to one unit of time.
3. For each job:
   - Find a free time slot from d_i down to 1.
   - If a free slot is found, schedule the job at that time slot.
4. Sum the profits of the scheduled jobs to get the maximum profit.

4. Example of Job Sequencing with Deadlines

Let's revisit the example with the following jobs:

Job	Profit (p_i)	Deadline (d_i)
1	100	2
2	19	1
3	27	2
4	25	1
5	15	3

Step-by-step Execution:

1. **Step 1: Sort Jobs by Profit**: First, sort the jobs by their profits in decreasing order:

Job	Profit (p_i)	Deadline (d_i)
1	100	2
3	27	2
4	25	1
2	19	1
5	15	3

1. **Step 2: Initialize Time Slots**:
    - Let's assume we have 3 time slots available (t1, t2, t3), where each slot represents a time unit (since the maximum deadline is 3).
    - Initialize an empty array to keep track of the job scheduled for each time slot.
2. **Step 3: Schedule the Jobs**:
    - **Job 1** (Profit = 100, Deadline = 2): Schedule it in time slot 2 (the latest available slot before its deadline).
    - **Job 3** (Profit = 27, Deadline = 2): Slot 2 is already taken, so schedule it in time slot 1 (the next available earlier slot).
    - **Job 4** (Profit = 25, Deadline = 1): Slot 1 is already taken, so skip this job.
    - **Job 2** (Profit = 19, Deadline = 1): Slot 1 is already taken, so skip this job.
    - **Job 5** (Profit = 15, Deadline = 3): Schedule it in time slot 3.
3. **Step 4: Calculate the Maximum Profit**: The jobs scheduled are:
    - Job 1 in slot 2.
    - Job 3 in slot 1.
    - Job 5 in slot 3.

The total profit is:

100+27+15=142100 + 27 + 15 = 142100+27+15=142

Solution:

The maximum profit is 142 by scheduling jobs 1, 3, and 5.

5. Time Complexity of the Greedy Algorithm

The time complexity of the greedy algorithm can be broken down as follows:

- Sorting the jobs by profit takes **O(n log n)**.
- Scheduling each job requires checking the available slots, which takes **O(n)** in the worst case.

Thus, the overall time complexity of the greedy algorithm is **O(n log n)**.

6. Limitation of the Greedy Approach

While the greedy approach works efficiently for the Job Sequencing Problem with Deadlines, it has limitations in more complex variations of the problem. The greedy algorithm doesn't always guarantee an optimal solution for all types of scheduling problems, especially when the constraints or objectives change (such as when the jobs have varying lengths or dependencies).

For more complex scheduling problems, other optimization techniques like **dynamic programming**, **branch and bound**, or **backtracking** may be required.

7. Applications of Job Sequencing with Deadlines

The Job Sequencing Problem with Deadlines has several practical applications, particularly in areas where maximizing profit or value under time constraints is critical. Some of the common applications include:

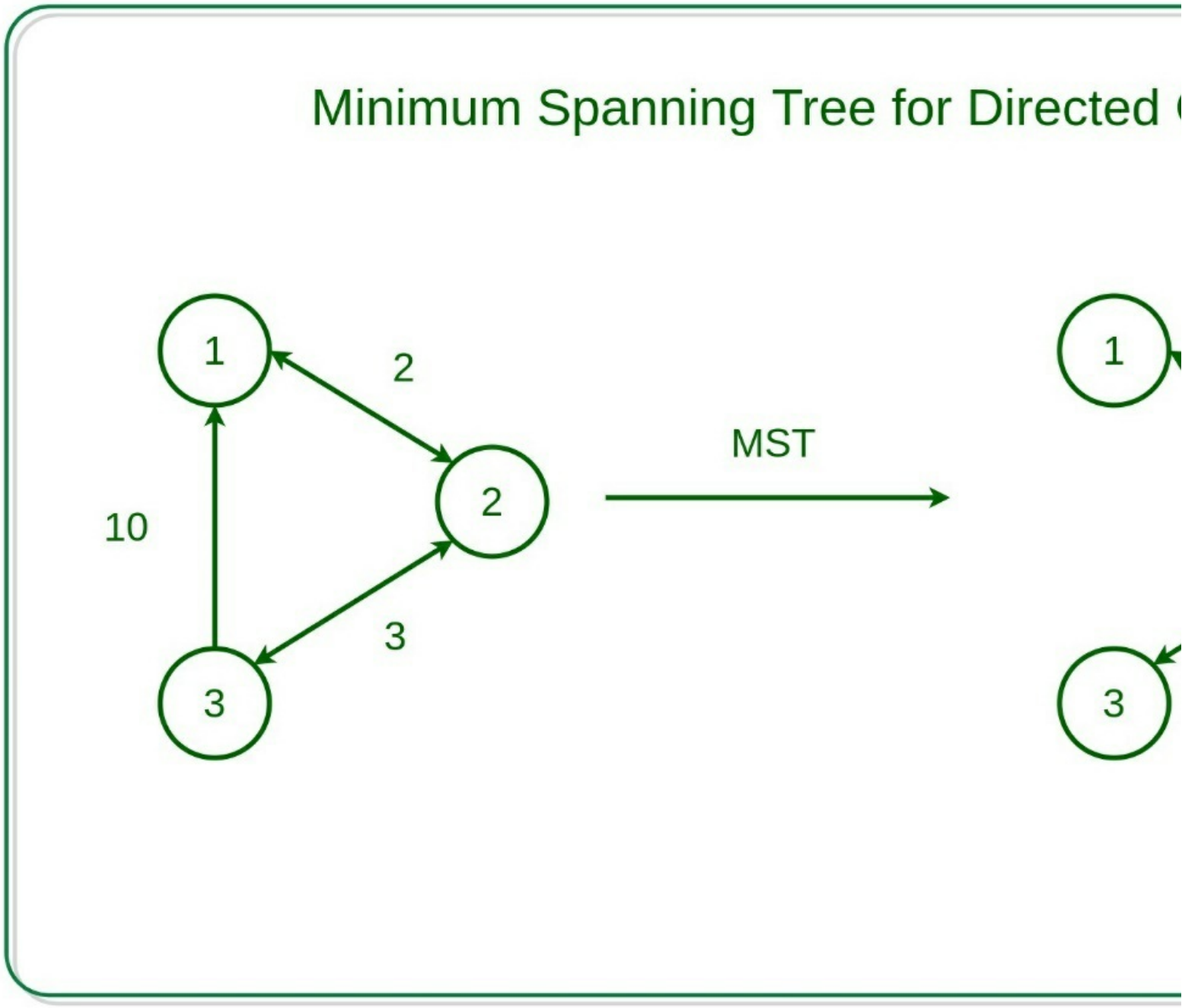
1. **Task Scheduling in Operating Systems**:
    - In modern operating systems, various tasks and processes compete for limited CPU time. Some tasks may have deadlines (e.g., real-time processes), and the system needs to schedule tasks in such a way that higher-priority tasks are completed before their deadlines.
2. **Project Management and Scheduling**:
    - In project management, multiple tasks must be completed within given deadlines. Proper scheduling can ensure that high-value tasks are prioritized, thereby maximizing the overall project value.
3. **Manufacturing and Production Planning**:
    - In manufacturing, several jobs may need to be completed within specified deadlines to maximize profit or minimize production costs. The Job Sequencing Problem helps determine the optimal order of job execution.
4. **Maximizing Return on Investment**:
    - In investment scenarios where certain investment opportunities have deadlines (e.g., limited-time offers), the Job Sequencing Problem can help investors choose the best set of investments to maximize returns.
5. **Cloud Computing and Resource Allocation**:
    - In cloud computing environments, tasks from different users or clients need to be scheduled on limited resources (e.g., CPUs or virtual machines). The Job Sequencing Problem can help in determining which tasks to prioritize to maximize system efficiency and profit.

Minimum Spanning Trees (MST)

*A **minimum spanning tree (MST)** is defined as a **spanning tree** that has the minimum weight among all the possible spanning trees*

A **spanning tree** is defined as a tree-like subgraph of a connected, undirected graph that includes all the vertices of the graph. Or, to say in Layman's words, it is a subset of the edges of the graph that forms a tree (**acyclic**) where every node of the graph is a part of the tree.

The minimum spanning tree has all the properties of a spanning tree with an added constraint of having the minimum possible weights among all possible spanning trees. Like a spanning tree, there can also be many possible MSTs for a graph.



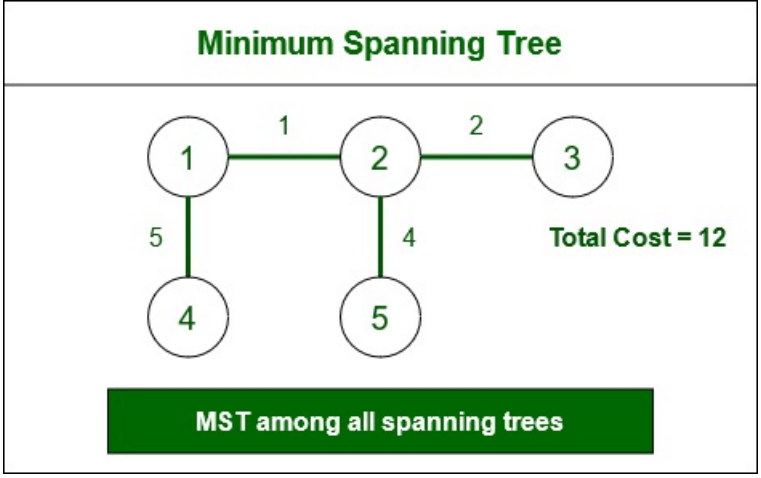
Properties of a Spanning Tree:

The spanning tree holds the **below-mentioned principles**:

- The number of vertices (**V**) in the graph and the spanning tree is the same.
- There is a fixed number of edges in the spanning tree which is equal to one less than the total number of vertices (**E** = **V-1**).
- The spanning tree should not be **disconnected**, as in there should only be a single source of component, not more than that.
- The spanning tree should be **acyclic,** which means there would not be any cycle in the tree.
- The total cost (or weight) of the spanning tree is defined as the sum of the edge weights of all the edges of the spanning tree.
- There can be many possible spanning trees for a graph.

The minimum spanning tree has all the properties of a spanning tree with an added constraint of having the minimum possible weights among all possible spanning trees. Like a spanning tree, there can also be many possible MSTs for a graph.

- Let’s look at the MST of the above example Graph,



Algorithms to find Minimum Spanning Tree:

There are several algorithms to find the minimum spanning tree from a given graph, some of them are listed below:

Kruskal's Minimum Spanning Tree Algorithm:

This is one of the popular algorithms for finding the minimum spanning tree from a connected, undirected graph. This is a greedy algorithm. The algorithm workflow is as below:

- First, it sorts all the edges of the graph by their weights,
- Then starts the iterations of finding the spanning tree.
- At each iteration, the algorithm adds the next lowest-weight edge one by one, such that the edges picked until now does not form a cycle.

This algorithm can be implemented efficiently using a DSU (Disjoint-Set) data structure to keep track of the connected components of the graph. This is used in a variety of practical applications such as network design, clustering, and data analysis.

Prim's Minimum Spanning Tree Algorithm:

This is also a greedy algorithm. This algorithm has the following workflow:

- It starts by selecting an arbitrary vertex and then adding it to the MST.
- Then, it repeatedly checks for the minimum edge weight that connects one vertex of MST to another vertex that is not yet in the MST.
- This process is continued until all the vertices are included in the MST.

To efficiently select the minimum weight edge for each iteration, this algorithm uses priority_queue to store the vertices sorted by their minimum edge weight currently. It also simultaneously keeps track of the MST using an array or other data structure suitable considering the data type it is storing.

This algorithm can be used in various scenarios such as image segmentation based on color, texture, or other features. For Routing, as in finding the shortest path between two points for a delivery truck to follow.

Boruvka's Minimum Spanning Tree Algorithm:

This is also a graph traversal algorithm used to find the minimum spanning tree of a connected, undirected graph. This is one of the oldest algorithms. The algorithm works by iteratively building the minimum spanning tree, starting with each vertex in the graph as its own tree. In each iteration, the algorithm finds the cheapest edge that connects a tree to another tree, and adds that edge to the minimum spanning tree. This is almost similar to the Prim's algorithm for finding the minimum spanning tree. The algorithm has the following workflow:

- Initialize a forest of trees, with each vertex in the graph as its own tree.
- For each tree in the forest:
  - Find the cheapest edge that connects it to another tree. Add these edges to the minimum spanning tree.
  - Update the forest by merging the trees connected by the added edges.
- Repeat the above steps until the forest contains only one tree, which is the minimum spanning tree.

The algorithm can be implemented using a data structure such as a priority queue to efficiently find the cheapest edge between trees. Boruvka's algorithm is a simple and easy-to-implement algorithm for finding minimum spanning trees, but it may not be as efficient as other algorithms for large graphs with many edges.

To knowing more about the properties and characteristics of Minimum Spanning Tree, click here.

Applications of Minimum Spanning Trees:

- **Network design**: Spanning trees can be used in network design to find the minimum number of connections required to connect all nodes. Minimum spanning trees, in particular, can help minimize the cost of the connections by selecting the cheapest edges.
- **Image processing**: Spanning trees can be used in image processing to identify regions of similar intensity or color, which can be useful for segmentation and classification tasks.
- **Biology**: Spanning trees and minimum spanning trees can be used in biology to construct phylogenetic trees to represent evolutionary relationships among species or genes.
- **Social network analysis**: Spanning trees and minimum spanning trees can be used in social network analysis to identify important connections and

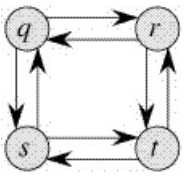
relationships among individuals or groups.

Optimal Sub-Structure

A given problem is said to have **Optimal Substructure Property** if the optimal solution of the given problem can be obtained by using the optimal solution to its subproblems instead of trying every possible way to solve the subproblems.
Example:

The Shortest Path problem has the following optimal substructure property:

If node x lies in the shortest path from a source node **U** to destination node **V** then the shortest path from **U** to **V** is a combination of the shortest path from **U** to **X** and the shortest path from **X** to **V**. The standard All Pair Shortest Path algorithm like Floyd–Warshall and **Single Source Shortest path** Optimal Substructure property. Here by Longest Path, we mean the longest simple path (path without cycle) between two nodes. Consider the following unweighted graph given in the **CLRS** book. There are two longest paths from q to t: q?r?t and q?s?t. Unlike shortest paths, these longest paths do not have the optimal substructure property. **For example,** The longest path q?r?t is not a combination of the longest path from q to r and the longest path from r to t, because the longest path from q to r is q?s?t?r and the longest path from r to t is r?q?s?t. algorithm for negative weight edges like Bellman–Ford are typical examples of **Dynamic Programming**.
On the other hand, the Longest Path problem doesn't have the



Shortest Path Problems

Shortest Path Problems involve finding the shortest path between two or more nodes in a weighted graph. These problems are commonly used in routing, navigation systems, and network optimization.

Methods to Solve Shortest Path Problems:

1. **Dijkstra's Algorithm**:
   - **Description**: Dijkstra's algorithm finds the shortest path from a source node to all other nodes in a graph with non-negative edge weights. It uses a priority queue to explore the nearest unvisited node.
   - **Time Complexity**:
     - Using a simple array: O(V2)O(V^2)O(V2), where VVV is the number of vertices.
     - Using a priority queue (min-heap): O((V+E)log⁡V)O((V + E) \log V)O((V+E)logV), where EEE is the number of edges.
   - **Space Complexity**: O(V)O(V)O(V), where VVV is the number of vertices.
2. **Bellman-Ford Algorithm**:
   - **Description**: Bellman-Ford can handle graphs with negative edge weights. It relaxes all edges repeatedly for V−1V-1V−1 iterations, where VVV is the number of vertices.
   - **Time Complexity**: O(V×E)O(V \times E)O(V×E), where EEE is the number of edges.
   - **Space Complexity**: O(V)O(V)O(V).
3. **Floyd-Warshall Algorithm**:
   - **Description**: Floyd-Warshall is a dynamic programming algorithm that computes shortest paths between all pairs of vertices. It's commonly used for dense graphs.
   - **Time Complexity**: O(V3)O(V^3)O(V3).
   - **Space Complexity**: O(V2)O(V^2)O(V2).

Need for Dijkstra's Algorithm (Purpose and Use-Cases)

The need for Dijkstra's algorithm arises in many applications where finding the shortest path between two points is crucial.

For example:

It can be used in the routing protocols for computer networks and also used by map systems to find the shortest path between starting point and the Destination.

Can Dijkstra's Algorithm work on both Directed and Undirected graphs?

Yes, Dijkstra's algorithm can work on both directed graphs and undirected graphs as this algorithm is designed to work on any type of graph as long as it meets the requirements of having non-negative edge weights and being connected.

- **In a directed graph,** each edge has a direction, indicating the direction of travel between the vertices connected by the edge. In this case, the algorithm follows the direction of the edges when searching for the shortest path.

1. **In an undirected graph,** the edges have no direction, and the algorithm can traverse both forward and backward along the edges when searching for the shortest path.

Algorithm for Dijkstra's Algorithm:

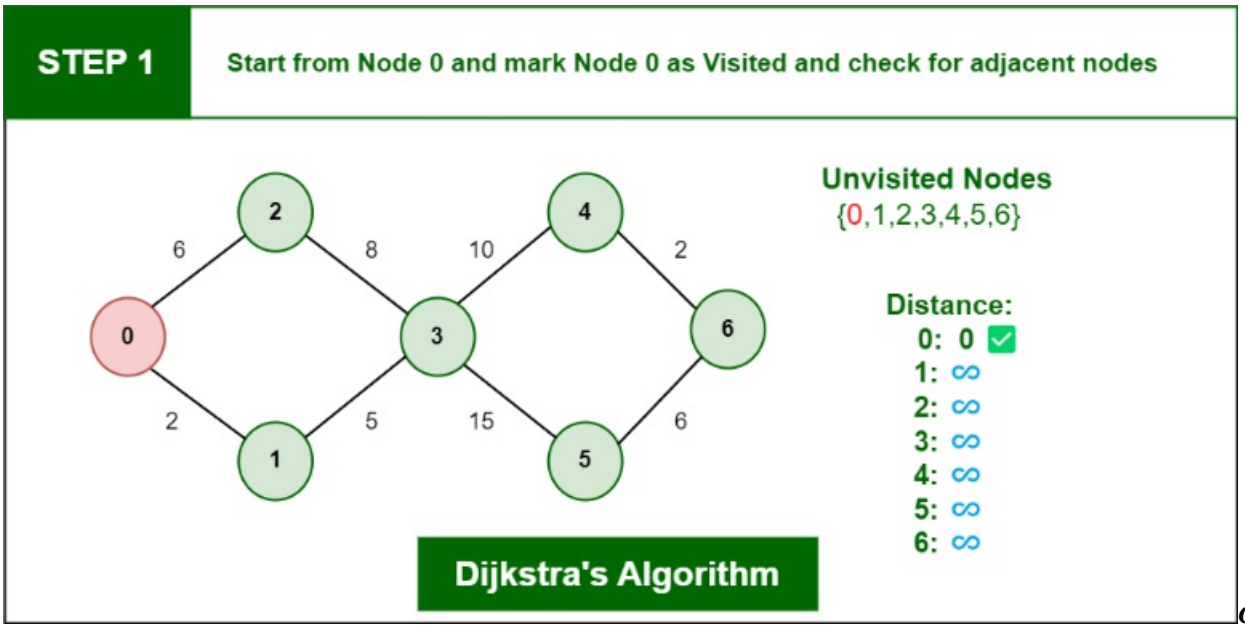
1. Mark the source node with a current distance of 0 and the rest with infinity.
2. Set the non-visited node with the smallest current distance as the current node.

3. For each neighbor, N of the current node adds the current distance of the adjacent node with the weight of the edge connecting 0->1. If it is smaller than the current distance of Node, set it as the new current distance of N.
4. Mark the current node 1 as visited.
5. Go to step 2 if there are any nodes are unvisited.

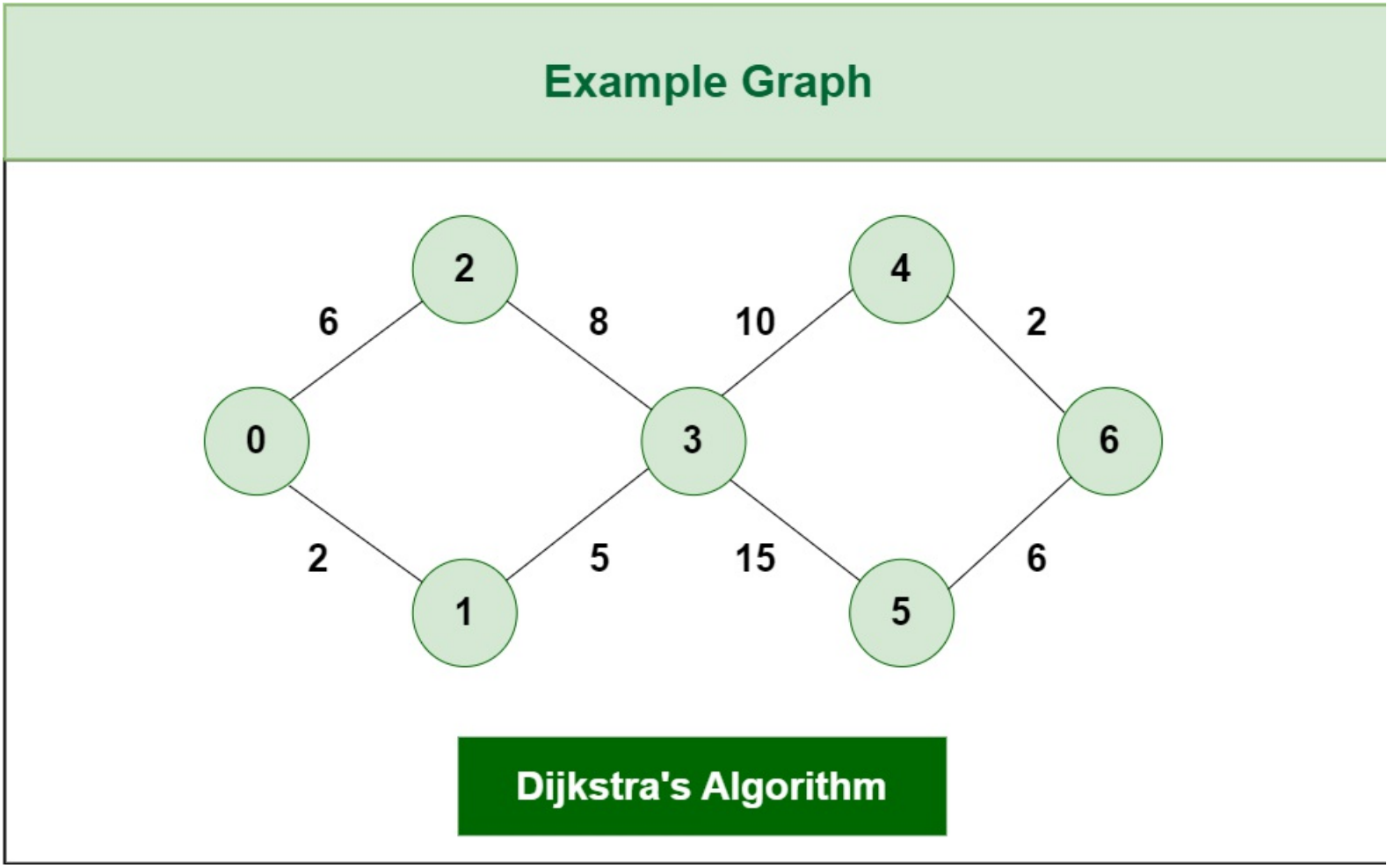
How does Dijkstra's Algorithm works?

Let's see how Dijkstra's Algorithm works with an example given below:

Dijkstra's Algorithm will generate the shortest path from Node 0 to all other Nodes in the graph.



Consider the below graph:



Dijkstra's Algorithm

The algorithm will generate the shortest path from node 0 to all the other nodes in the graph.

For this graph, we will assume that the weight of the edges represents the distance between two nodes.

As, we can see we have the shortest path from,
Node 0 to Node 1, from

Node 0 to Node 2, from
Node 0 to Node 3, from
Node 0 to Node 4, from
Node 0 to Node 6.

Initially we have a set of resources given below :

- *The Distance from the source node to itself is 0. In this example the source node is 0.*

1. *The distance from the source node to all other node is unknown so we mark all of them as infinity.*

Example: 0 -> 0, 1-> ∞,2-> ∞,3-> ∞,4-> ∞,5-> ∞,6-> ∞.

- *we'll also have an array of unvisited elements that will keep track of unvisited or unmarked Nodes.*

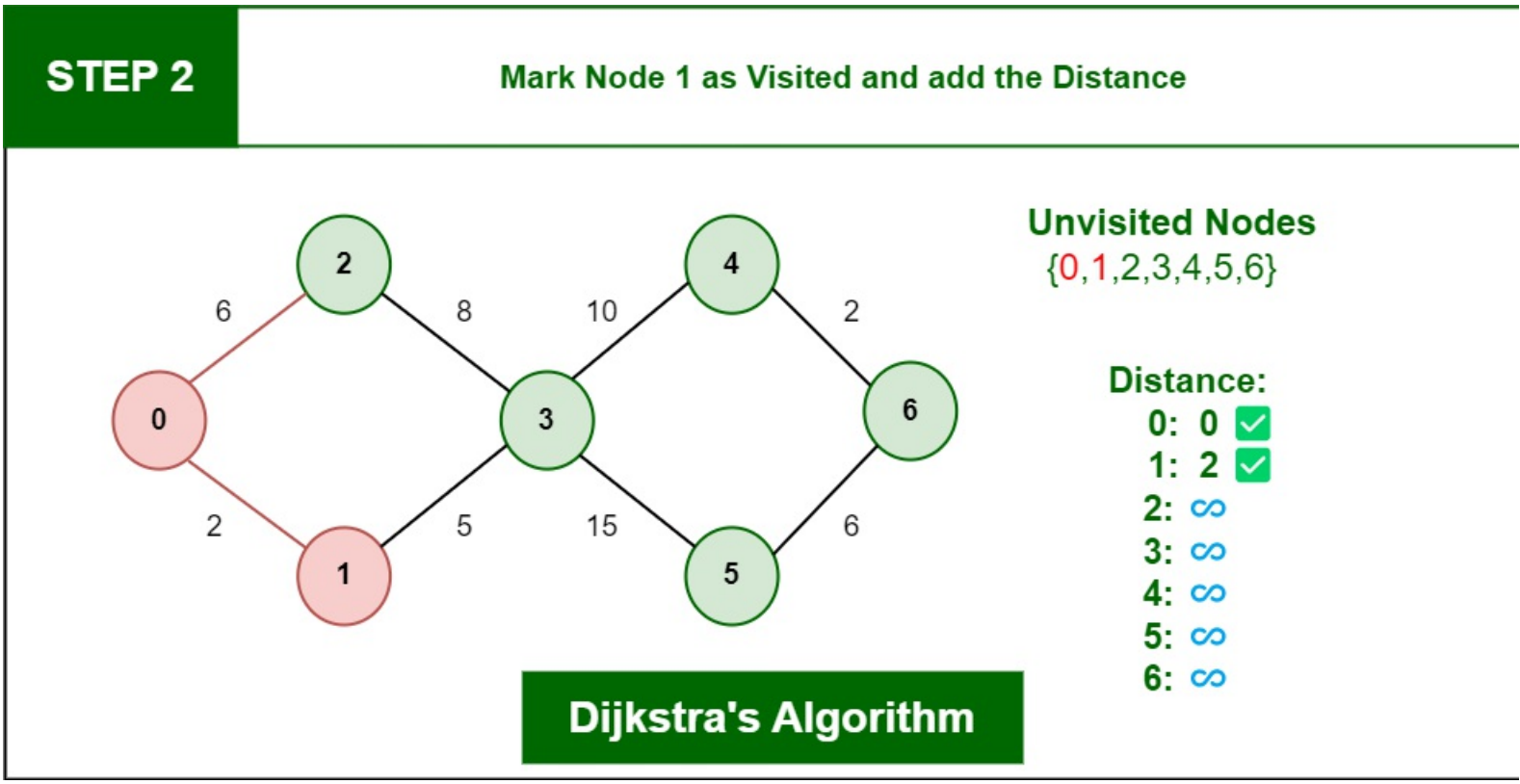
1. *Algorithm will complete when all the nodes marked as visited and the distance between them added to the path.* ***Unvisited Nodes:- 0 1 2 3 4 5 6.***

Step 1: *Start from Node 0 and mark Node as visited as you can check in below image visited Node is marked red.*

Dijkstra's Algorithm

Step 2: *Check for adjacent Nodes, Now we have to choices (Either choose Node1 with distance 2 or either choose Node 2 with distance 6) and choose Node with minimum distance. In this step* ***Node 1*** *is Minimum distance adjacent Node, so marked it as visited and add up the distance.*

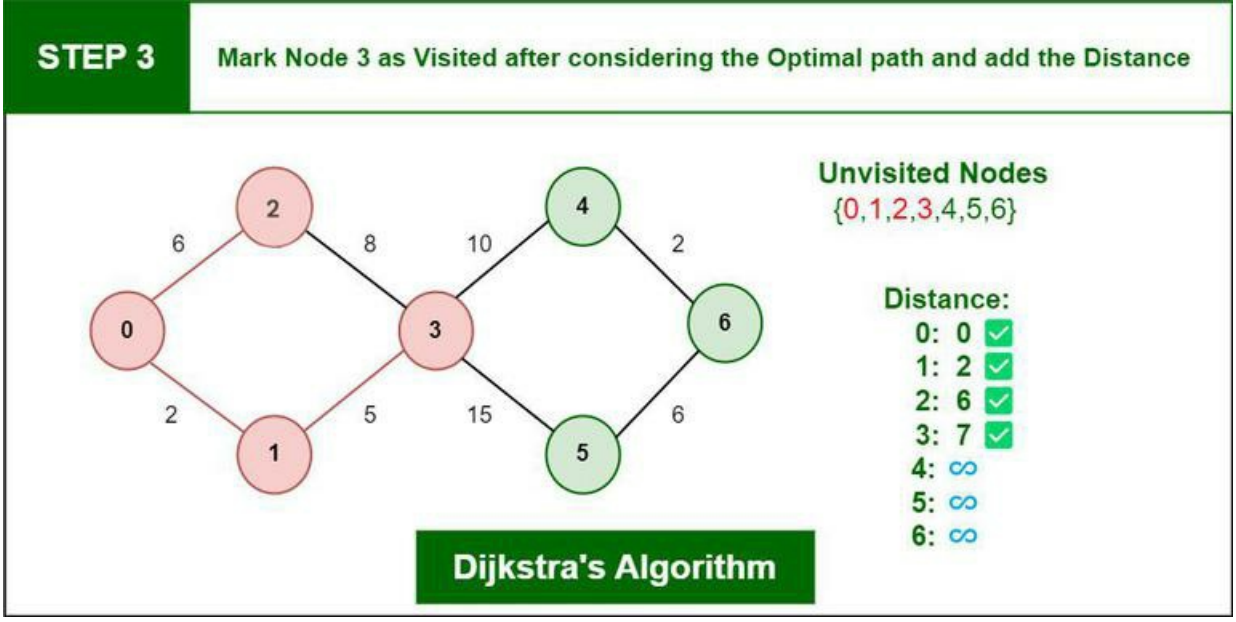
Distance: Node 0 -> Node 1 = 2



Dijkstra's Algorithm

Step 3: *Then Move Forward and check for adjacent Node which is Node 3, so marked it as visited and add up the distance, Now the distance will be:*

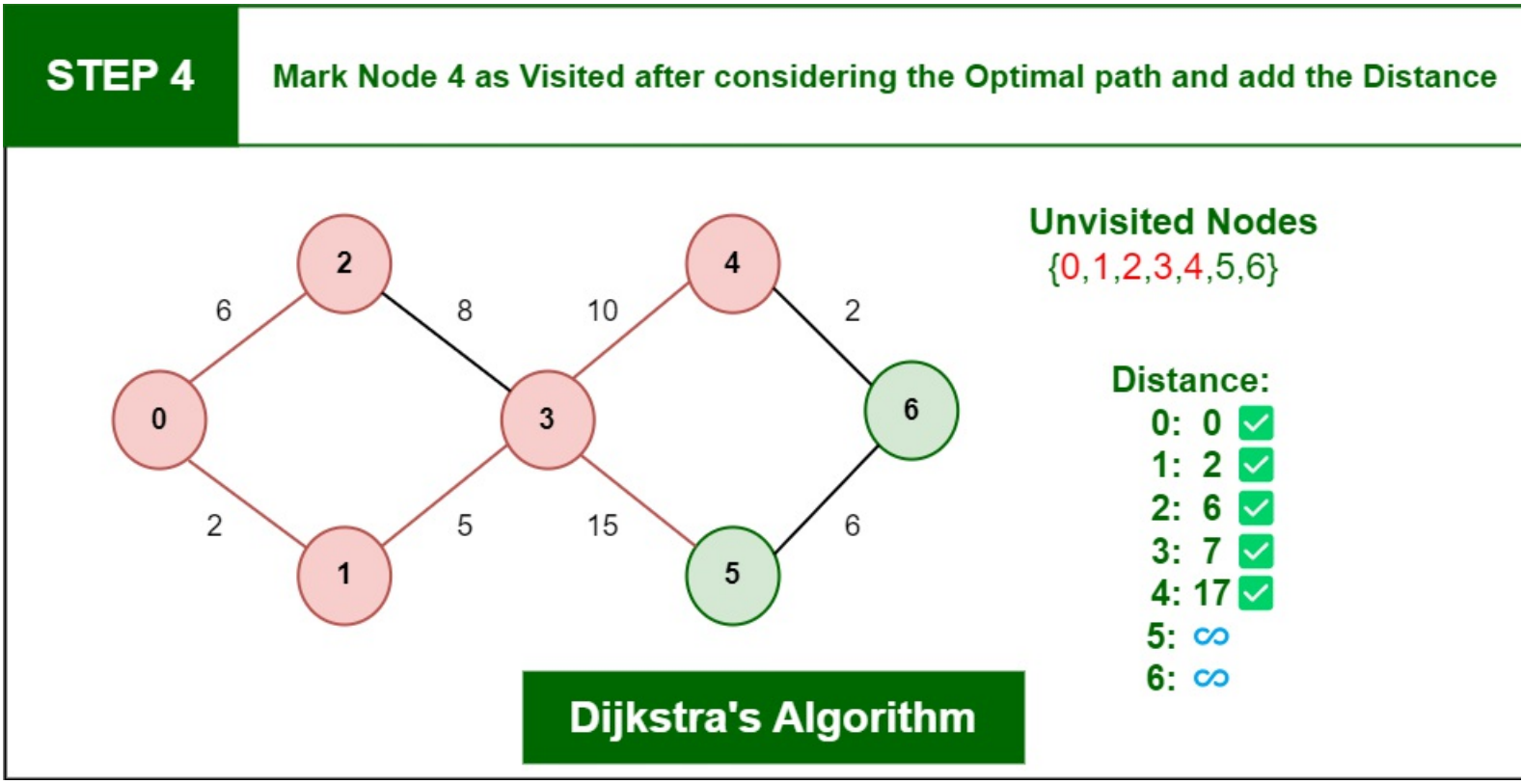
Distance: Node 0 -> Node 1 -> Node 3 = 2 + 5 = 7



Dijkstra's Algorithm

Step 4: *Again we have two choices for adjacent Nodes (Either we can choose Node 4 with distance 10 or either we can choose Node 5 with distance 15) so choose Node with minimum distance. In this step* ***Node 4*** *is Minimum distance adjacent Node, so marked it as visited and add up the distance.*

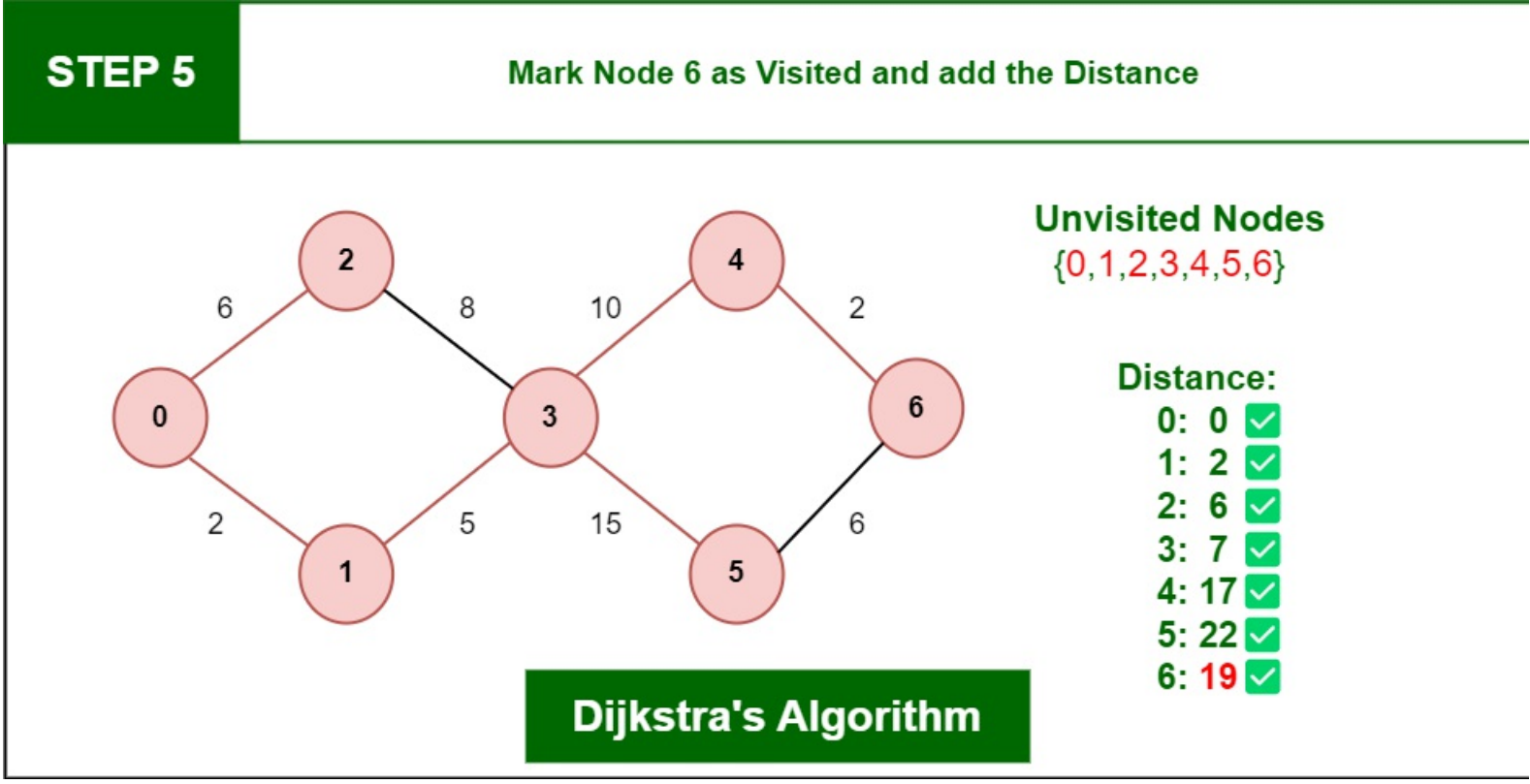
Distance: Node 0 -> Node 1 -> Node 3 -> Node 4 = 2 + 5 + 10 = 17



Dijkstra's Algorithm

Step 5: *Again, Move Forward and check for adjacent Node which is* ***Node 6****, so marked it as visited and add up the distance, Now the distance will be:*

Distance: Node 0 -> Node 1 -> Node 3 -> Node 4 -> Node 6 = 2 + 5 + 10 + 2 = 19



Dijkstra's Algorithm

So, the Shortest Distance from the Source Vertex is 19 which is optimal one

function Dijkstra(Graph, source):

 # Step 1: Initialization

dist[] = array of distances from source to all vertices, initialized to infinity

dist[source] = 0

visited[] = array of booleans, initialized to false for all vertices

 # Step 2: Create a priority queue (min-heap) and insert the source node with distance 0

priorityQueue = min-heap

priorityQueue.insert(source, 0)

while priorityQueue is not empty:

 # Step 3: Extract the vertex u with the minimum distance

u = priorityQueue.extractMin()

if visited[u] is true:

continue # Skip the node if it is already visited

visited[u] = true

 # Step 4: For each neighbor v of u

for each neighbor v of u in Graph:

weight = edge weight from u to v

if visited[v] is false:

 # Step 5: Relaxation step

if dist[u] + weight < dist[v]:

dist[v] = dist[u] + weight

 # Step 6: Update the priority queue

priorityQueue.insert(v, dist[v])

return dist[] # Array of shortest distances from the source to all vertices

Implemention of Dijkstra's Algorithm:

There are several ways to Implement Dijkstra's algorithm, but the most common ones are:

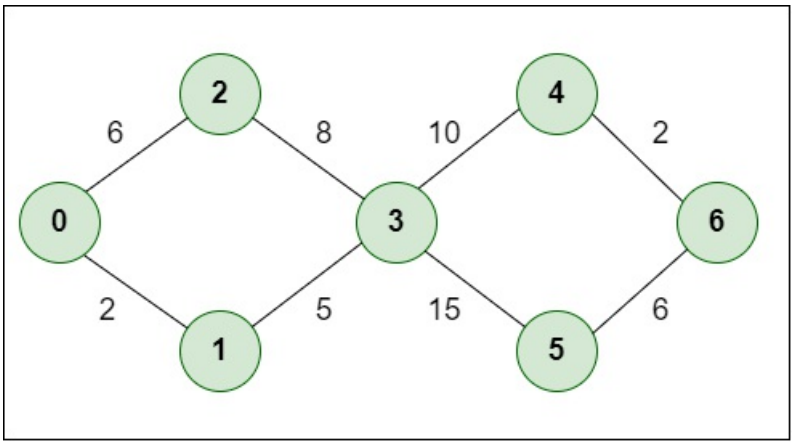
1. Priority Queue (Heap-based Implementation):
2. Array-based Implementation:

1. Dijkstra's Shortest Path Algorithm using priority_queue (Heap)

In this Implementation, we are given a graph and a source vertex in the graph, finding the shortest paths from the source to all vertices in the given graph.

Example:

Input: *Source = 0*



Example

Output: *Vertex*

Vertex Distance from Source

0 -> 0
1 -> 2
2 -> 6
3 -> 7
4 -> 17
5 -> 22
6 -> 19

Below is the algorithm based on the above idea:

- Initialize the distance values and priority queue.

1. Insert the source node into the priority queue with distance 0.
2. While the priority queue is not empty:
    - Extract the node with the minimum distance from the priority queue.
    1. Update the distances of its neighbors if a shorter path is found.
    2. Insert updated neighbors into the priority queue.

Code:

```
#include <bits/stdc++.h>

using namespace std;

#define INF 0x3f3f3f3f

typedef pair<int, int> iPair;

class Graph {

int V;

list<pair<int, int> >* adj;

public:

Graph(int V);
```

```
void addEdge(int u, int v, int w);
void shortestPath(int s);
};
Graph::Graph(int V)
{
this->V = V;
adj = new list<iPair>[V];
}
void Graph::addEdge(int u, int v, int w)
{
adj[u].push_back(make_pair(v, w));
adj[v].push_back(make_pair(u, w));
}
void Graph::shortestPath(int src)
{ priority_queue<iPair,vector<iPair>,greater<iPair> > pq;
vector<int> dist(V, INF);
pq.push(make_pair(0, src));
dist[src] = 0;
while (!pq.empty()) {
int u = pq.top().second;
pq.pop();
list<pair<int, int> >::iterator i;
for (i = adj[u].begin(); i != adj[u].end(); ++i) {
int v = (*i).first;
int weight = (*i).second;
if (dist[v] > dist[u] + weight) {
dist[v] = dist[u] + weight;
pq.push(make_pair(dist[v], v));
}
}
}
printf("Vertex Distance from Source\n");
for (int i = 0; i < V; ++i)
printf("%d \t\t %d\n", i, dist[i]);
}
int main()
{
int V = 7;
Graph g(V);
```

g.addEdge(0, 1, 2);

g.addEdge(0, 2, 6);

g.addEdge(1, 3, 5);

g.addEdge(2, 3, 8);

g.addEdge(3, 4, 10);

g.addEdge(3, 5, 15);

g.addEdge(4, 6, 2);

g.addEdge(5, 6, 6);

g.shortestPath(0);

return 0;

}

Complexity Analysis of Dijkstra Algorithm:

- **Time complexity:** O((V + E) log V)**,** where V is the number of vertices and E is the number of edges.

1. **Auxiliary Space:** O(V), where V is the number of vertices and E is the number of edges in the graph.

Bellman-Ford Algorithm

Bellman-Ford Algorithm

The **Bellman-Ford algorithm** is used to find the shortest paths from a single source vertex to all other vertices in a weighted graph. Unlike Dijkstra's algorithm, the Bellman-Ford algorithm works for graphs with **negative weight edges**. It can also detect **negative weight cycles**—a cycle in which the sum of the edge weights is negative. If such a cycle exists, the algorithm reports that no solution exists.

Key Features:

- **Handles Negative Weights**: Unlike Dijkstra's algorithm, Bellman-Ford can handle graphs where some edges have negative weights.
- **Detects Negative Weight Cycles**: It can detect if there is a cycle whose total weight is negative, which means there's no shortest path (the distances keep getting smaller).

Time Complexity:

The time complexity of Bellman-Ford is **O(V * E)**, where:

- V is the number of vertices.
- E is the number of edges.

This complexity arises because the algorithm iterates V - 1 times over all edges to ensure that no shorter paths exist, and each iteration checks all edges (thus, V * E operations).

While Bellman-Ford is slower than Dijkstra's algorithm (which has a time complexity of **O(E + V log V)** using a priority queue), it remains useful for graphs that contain negative edge weights.

How Bellman-Ford Works:

The algorithm follows these steps:

1. **Initialization**: Set the distance to the source vertex as 0 and all other vertices as infinity.
2. **Relaxation**: For each edge in the graph, if the distance to the destination vertex can be shortened by taking the edge, update the distance to the destination.
3. **Repeat Relaxation**: Repeat the relaxation step V - 1 times, where V is the number of vertices. This guarantees that all shortest paths (even those requiring V - 1 edges) are found.
4. **Negative Cycle Check**: After V - 1 iterations, repeat the process one more time. If any edge can still be relaxed, a negative weight cycle exists.

Pseudocode for Bellman-Ford Algorithm:

sql

Copy code

function BellmanFord(Graph, V, E, source):

dist[] = array of size V, initialize all values to infinity

dist[source] = 0

```
# Step 1: Relax all edges V-1 times

for i from 1 to V-1:

for each edge (u, v) with weight w in Graph:

if dist[u] + w < dist[v]:

dist[v] = dist[u] + w

# Step 2: Check for negative weight cycles

for each edge (u, v) with weight w in Graph:

if dist[u] + w < dist[v]:

return "Graph contains negative weight cycle"

return dist[]
```

Explanation:

- **Initialization**: The dist[] array holds the shortest distance from the source to all other vertices. Initially, all distances are set to infinity, except the source vertex, which is set to 0.
- **Relaxation Step**: In each iteration, the algorithm checks all edges and relaxes the distances. Relaxing an edge means checking if there is a shorter path to the destination vertex through the source vertex.
- **Cycle Detection**: After the V - 1 iterations, the algorithm checks one more time if it can relax any edge. If any edge can still be relaxed, this indicates a negative weight cycle.

Bellman-Ford Algorithm Implementation in C++:

```
#include <bits/stdc++.h>

using namespace std;

struct Edge {

int u, v, weight;

};

void BellmanFord(vector<Edge>& edges, int V, int E, int source) {

vector<int> dist(V, INT_MAX);

dist[source] = 0;

// Step 1: Relax all edges V-1 times

for (int i = 1; i <= V - 1; ++i) {

for (int j = 0; j < E; ++j) {

int u = edges[j].u;

int v = edges[j].v;

int weight = edges[j].weight;

if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {

dist[v] = dist[u] + weight;

}

}

}

// Step 2: Check for negative weight cycles

for (int j = 0; j < E; ++j) {

int u = edges[j].u;

int v = edges[j].v;
```

```
int weight = edges[j].weight;

if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {

cout << "Graph contains negative weight cycle\n";

return;

}

}

// Print shortest distances

cout << "Vertex Distance from Source\n";

for (int i = 0; i < V; ++i)

cout << i << "\t\t" << dist[i] << "\n";

}

int main() {

int V = 5; // Number of vertices in the graph

int E = 8; // Number of edges in the graph

vector<Edge> edges = { {0, 1, -1}, {0, 2, 4}, {1, 2, 3}, {1, 3, 2},

{1, 4, 2}, {3, 2, 5}, {3, 1, 1}, {4, 3, -3} };

int source = 0;

BellmanFord(edges, V, E, source);

return 0;

}
```

Explanation of the Implementation:

1. **Edge Representation**: The graph is represented as an array of edges, where each edge has a start vertex u, an end vertex v, and a weight.
2. **Distance Array**: The dist[] array holds the minimum distance from the source to each vertex, initialized to infinity (INT_MAX) except for the source, which is set to 0.
3. **Relaxation**: Each edge is relaxed V - 1 times to ensure the shortest paths are calculated.
4. **Negative Cycle Check**: If any edge can still be relaxed after V - 1 iterations, the graph contains a negative weight cycle, which is detected and reported.
5. **Output**: The shortest distances from the source to each vertex are printed.

Advantages of Bellman-Ford:

- **Handles Negative Weights**: It can work with graphs that have negative weight edges, unlike Dijkstra's algorithm.
- **Detects Negative Cycles**: It can detect negative weight cycles in a graph, which is useful for identifying scenarios where there is no solution.

Disadvantages of Bellman-Ford:

- **Time Complexity**: The algorithm's time complexity of **O(V * E)** makes it less efficient than Dijkstra's algorithm, especially for large graphs with many edges.
- **Not Optimal for Non-Negative Graphs**: If all edge weights are non-negative, Dijkstra's algorithm is faster and more efficient.

Applications of Bellman-Ford Algorithm:

- **Detecting Arbitrage**: In financial markets, negative cycles can be used to detect arbitrage opportunities (making a profit with zero net investment).
- **Routing Algorithms**: It is used in networking for distance-vector routing algorithms like RIP (Routing Information Protocol), where networks can have negative weights.
- **Solving Linear Programming Problems**: Some linear programming problems can be represented using graphs with negative weights, where Bellman-Ford can be useful.

In summary, the Bellman-Ford algorithm is a fundamental algorithm in graph theory that efficiently handles negative weights and detects negative cycles. It is an important tool for various applications, though slower than other algorithms like Dijkstra's in cases where negative weights are not involved.

Floyd Warshall Algorithm

The Floyd-Warshall algorithm is an all-pairs shortest path algorithm, meaning it finds the shortest paths between all pairs of vertices in a weighted graph. It is particularly useful for dense graphs and can handle negative weight edges (but not negative weight cycles). Unlike Dijkstra's or Bellman-Ford algorithms, which find shortest paths from a single source, Floyd-Warshall computes the shortest paths between all vertex pairs.

Key Features:

- All-Pairs Shortest Path: It calculates the shortest path between all pairs of vertices.
- Dynamic Programming Approach: It uses dynamic programming to iteratively update the shortest paths.
- Negative Weights: The algorithm can handle graphs with negative weight edges but cannot handle graphs with negative weight cycles (it will fail to produce correct results in this case).
- Space Efficiency: The algorithm operates in $O(V^2)$ space, where V is the number of vertices.

Time Complexity:

The time complexity of Floyd-Warshall is $O(V^3)$, where V is the number of vertices. This is because the algorithm uses three nested loops, each running V times.

While this makes the algorithm slower than Dijkstra's for sparse graphs, it is useful for dense graphs or when the number of edges approaches V^2.

How Floyd-Warshall Works:

The Floyd-Warshall algorithm works by considering each possible pair of vertices (i, j) and checking if introducing an intermediate vertex k improves the shortest path between them. The core idea is to incrementally check if a shorter path can be found by including each vertex as an intermediate point between two other vertices.

Steps:

1. Initialization: Create a distance matrix where dist[i][j] represents the shortest distance between vertex i and vertex j. Initially, the matrix is filled with the direct edge weights between vertices (or infinity if there is no direct edge), and 0 for the diagonal (distance from a vertex to itself).
2. Iterative Improvement: For each vertex k, update the distance matrix by checking whether using k as an intermediate vertex provides a shorter path between two vertices i and j.
3. Result: After iterating through all possible intermediate vertices, the matrix will contain the shortest distances between all pairs of vertices.

Pseudocode for Floyd-Warshall Algorithm:

function FloydWarshall(Graph, V):

dist = array[V][V]

Step 1: Initialize the distance matrix

for i from 1 to V:

for j from 1 to V:

if i == j:

dist[i][j] = 0

else if there is an edge from i to j:

dist[i][j] = weight of edge (i, j)

else:

dist[i][j] = infinity

Step 2: Update distances with intermediate vertices

for k from 1 to V:

for i from 1 to V:

for j from 1 to V:

if dist[i][k] + dist[k][j] < dist[i][j]:

dist[i][j] = dist[i][k] + dist[k][j]

return dist

Explanation:

1. Initialization: The distance matrix is initialized with the direct edge weights between vertices. If there is no direct edge between two vertices, the distance is set to infinity, except for the diagonal where dist[i][i] = 0 (distance to itself).
2. Updating with Intermediate Vertices: For each possible intermediate vertex k, the algorithm checks whether the path from i to j can be shortened by going through k. If dist[i][k] + dist[k][j] is smaller than dist[i][j], the distance is updated.
3. Final Matrix: After the three nested loops finish, the distance matrix contains the shortest distances between all pairs of vertices.

Floyd-Warshall Algorithm Implementation in C++:

```
#include <bits/stdc++.h>
using namespace std;
#define INF 99999
void FloydWarshall(int graph[][4], int V) {
int dist[V][V];
// Step 1: Initialize distance matrix with given graph
for (int i = 0; i < V; i++) {
for (int j = 0; j < V; j++) {
dist[i][j] = graph[i][j];
}
}
// Step 2: Update the distance matrix using intermediate vertices
for (int k = 0; k < V; k++) {
for (int i = 0; i < V; i++) {
for (int j = 0; j < V; j++) {
if (dist[i][k] + dist[k][j] < dist[i][j]) {
dist[i][j] = dist[i][k] + dist[k][j];
}
}
}
}
// Step 3: Print the shortest distance matrix
cout << "Shortest distances between every pair of vertices:\n";
for (int i = 0; i < V; i++) {
for (int j = 0; j < V; j++) {
if (dist[i][j] == INF)
cout << "INF ";
else
cout << dist[i][j] << " ";
}
cout << endl;
}
}
int main() {
int V = 4;
int graph[4][4] = { {0, 5, INF, 10},
{INF, 0, 3, INF},
{INF, INF, 0, 1},
{INF, INF, INF, 0} };
```

```
FloydWarshall(graph, V);

return 0;

}
```

Explanation of the Implementation:

1. **Distance Matrix Initialization**: The distance matrix dist[][] is initialized with the given graph's edge weights. If there is no edge between two vertices, the distance is set to a large value (INF), except for the diagonal where the distance is 0.
2. **Update Step**: For each possible intermediate vertex k, the algorithm checks if there is a shorter path between any pair of vertices i and j through vertex k. If the shorter path is found, it updates the distance matrix.
3. **Final Output**: The program prints the matrix of shortest distances between all pairs of vertices. If no path exists between two vertices, it prints INF (representing infinity).

Time Complexity:

The time complexity of Floyd-Warshall is $\mathbf{O(V^3)}$, where V is the number of vertices. This comes from the three nested loops used to update the distance matrix. It is relatively expensive for large graphs, but for smaller graphs, it is quite efficient and simple to implement.

Space Complexity:

The space complexity is $\mathbf{O(V^2)}$ due to the use of a 2D distance matrix to store the shortest paths between all pairs of vertices.

Advantages of Floyd-Warshall Algorithm:

- **Simplicity**: The algorithm is easy to understand and implement.
- **Handles Negative Weights**: It can handle negative edge weights (but no negative weight cycles).
- **All-Pairs Shortest Paths**: It finds the shortest paths between all pairs of vertices, making it more comprehensive than single-source shortest path algorithms like Dijkstra or Bellman-Ford.

Disadvantages of Floyd-Warshall Algorithm:

- **Time Complexity**: With a time complexity of $\mathbf{O(V^3)}$, the Floyd-Warshall algorithm is less efficient than other algorithms (like Dijkstra's) for sparse graphs.
- **No Negative Cycle Detection**: While Floyd-Warshall can handle negative edge weights, it does not explicitly detect negative weight cycles. If a negative cycle exists, it may produce incorrect results by continuing to reduce path costs in every iteration.

Applications of Floyd-Warshall Algorithm:

- **Shortest Path Problems in Dense Graphs**: The algorithm is particularly well-suited for dense graphs where the number of edges approaches V^2.
- **Routing Algorithms**: It can be used in network routing to find the shortest paths between all pairs of routers.
- **Transitive Closure of Graphs**: Floyd-Warshall can also be adapted to compute the transitive closure of a graph, determining whether a path exists between every pair of vertices.

Handling Negative Weight Cycles:

While Floyd-Warshall cannot explicitly detect negative weight cycles, a simple modification can be made. After running the algorithm, if $dist[i][i] < 0$ for any vertex i, it means that a negative weight cycle exists, as the distance from a vertex to itself has become negative.

Conclusion:

The Floyd-Warshall algorithm is an elegant and simple solution for finding the shortest paths between all pairs of vertices in a graph. Though it is not the most efficient algorithm for large sparse graphs, it excels in dense graphs and scenarios where negative weights are present. Its simplicity, versatility, and ability to compute all-pairs shortest paths make it a fundamental tool in graph theory and computer science.