UNIT -2

Greedy Method in Algorithm Design – Detailed Explanation

✅ Overview

The **Greedy Method** is an algorithmic paradigm used for solving optimization problems. It builds up a solution **piece by piece**, always choosing the **next piece** that offers the **most immediate benefit** (i.e., the "greedy" choice), **without considering the global consequences**.

It does not always produce the optimal solution for all problems, but for **some problems**, it does lead to the **globally optimal** solution.

🔑 Key Characteristics of Greedy Algorithms

1. **Greedy Choice Property**: A global optimum can be arrived at by selecting a local optimum.
2. **Optimal Substructure**: A problem has an optimal substructure if an optimal solution can be constructed from optimal solutions of its subproblems.

If a problem has both properties, a greedy algorithm is often an appropriate solution.

📌 General Approach of Greedy Method:

1. **Start** with an empty solution.
2. **Repeat**:
    - Choose the best option **based on a local decision** (e.g., max profit, min weight, etc.).
    - Add it to the solution.
3. **Stop** when you reach the goal or no more choices are possible.

4. **Return** the solution.

🧠 General Structure of a Greedy Algorithm

text

Copy code

```
1. Initialize a solution set (often empty).
2. While the solution is not complete:
   a. Choose the best possible option based on the current state (greedy choice).
   b. Add this choice to the solution set.
   c. Update the state.
3. Return the solution set.
```

📈 Greedy vs Dynamic Programming

Feature	**Greedy Method**	**Dynamic Programming**
Decision Making	Local optimal choice	Optimal choice based on subproblems
Solution Guarantee	Not always optimal	Always optimal (for problems with optimal substructure)
Time Complexity	Often faster	Generally more time-consuming
Space Complexity	Low	Higher due to memoization

❗ When Not to Use Greedy Algorithms

- When the problem **lacks optimal substructure**.
- When the **local optimum** choice leads to a **globally suboptimal** solution.

- Example: **0/1 Knapsack Problem** – Greedy fails to give optimal result.

Simple Example: Coin Change Problem (Greedy Version)

🧾 Problem:

You need to make change for ₹30 using the minimum number of coins.
Available coins: ₹1, ₹2, ₹5, ₹10

✅ Greedy Strategy:

Always take the highest denomination coin that doesn't exceed the remaining amount.

💡 Step-by-step Solution:

text

Copy code

```
Remaining Amount = 30


- Take ₹10 → Remaining = 20

- Take ₹10 → Remaining = 10

- Take ₹10 → Remaining = 0


Coins used: [10, 10, 10]
```

✅ Minimum number of coins = 3

✅ Advantages of Greedy Algorithms

- Simple to design and implement.
- Generally efficient and fast.
- Uses less memory compared to dynamic programming.

❌ Disadvantages

- Doesn't always give the optimal solution.
- Not suitable for all problem types.

📘 Definition of Fractional Knapsack Problem

The **Fractional Knapsack Problem** is an optimization problem in which you are given a set of items, each with a **value** and a **weight**, and a knapsack with a **maximum weight capacity**.

The goal is to **maximize the total value** in the knapsack by selecting items. Unlike the 0/1 Knapsack problem, you **can take fractions of an item** rather than having to take it entirely or not at all.

🔑 Key Point:

You are **allowed to break items** and take only a part of them to fill the knapsack optimally.

🧠 Example:

If an item has:

- Value = ₹100
- Weight = 10 kg
  And the knapsack has only 5 kg space left,
  You can take **half** of the item, gaining **₹50** in value.

✅ Problem Statement

You are given:

- A knapsack with capacity W
- n items, each with:

    - value[i] – value of item i
    - weight[i] – weight of item i

- **Goal**: Maximize total value placed into the knapsack.
- **Condition**: You **can take fractions** of items (unlike the 0/1 knapsack).

🧠 Greedy Approach

The greedy strategy:

1. Calculate **value per unit weight** (value/weight) for each item.
2. Sort items by this ratio in **descending order**.
3. Pick items in this order:
    - If the entire item fits, take it.
    - If not, take the **fraction that fits**.

📈 Algorithm Steps

Step-by-step procedure:

1. Calculate value/weight ratio for all items.
2. Sort items based on this ratio in descending order.
3. Initialize total_value = 0 and remaining_capacity = W.
4. For each item:
    - If item weight ≤ remaining capacity:
        - Take the full item
        - Add its value to total_value
        - Subtract weight from remaining_capacity
    - Else:
        - Take the fractional part that fits
        - Add corresponding value

        - Break (knapsack is full)

5. Return total_value.

💻 Algorithm (Pseudo-Code)

plaintext

Copy code

```
function fractionalKnapsack(W, items):
    for each item:
        compute value_per_weight = value / weight

    sort items by value_per_weight in descending order

    total_value = 0

    for item in sorted_items:
        if item.weight ≤ W:
            W = W - item.weight
            total_value += item.value
        else:
            fraction = W / item.weight
            total_value += item.value * fraction
            break  // knapsack is full

    return total_value
```

🧪 Example

Let:

- Capacity W = 50
- Items:

Item	Value	Weight	Value/Weight
A	60	10	6.0
B	100	20	5.0
C	120	30	4.0

Sorted Order:

A > B > C

Fill Knapsack:

- Take A (10) → Remaining = 40 → Value = 60
- Take B (20) → Remaining = 20 → Value = 160
- Take 20/30 of C → Value = 120 × (20/30) = 80

✅ **Total Value = 60 + 100 + 80 = 240**

✅ Advantages

Feature	Description
🔹 Simplicity	Easy to understand and implement
🔹 Fast	Efficient with time complexity O(n log n) (due to sorting)
🔹 Optimal Result	Always gives optimal result for **fractional version**
🔹 Useful in real life	Applies to problems like cutting, packing, budgeting

---

❌ Disadvantages

Limitation	**Explanation**
❌ Doesn't work for 0/1 knapsack	Greedy may miss better combinations
❌ Assumes divisibility	Not all items in real-life can be split (e.g., laptop)
❌ May be misleading	For certain coin or resource problems, greedy fails

---

🕒 Time Complexity

- Sorting items by ratio: **O(n log n)**
- Iterating through items: **O(n)**
  ✅ **Total Time Complexity: O(n log n)**
  ✅ **Space Complexity: O(1)** (or O(n) if storing additional data)

📌 Use Cases

- Budget planning with divisible resources
- Resource allocation (e.g., allocating computing power)
- Industrial cutting or material use where splitting is allowed
- Bandwidth allocation problems

✅ Summary

Aspect	**Detail**
Problem Type	Optimization
Strategy	Greedy – pick based on value/weight
Can take fractions?	✅ Yes
Optimal solution?	✅ Yes for fractional, ❌ No for 0/1 version
Time Complexity	O(n log n)

Would you like a **Python, C++, or Java implementation**, or a **visual diagram** for this example?

You said:

defination

ChatGPT said:

🧩 Job Sequencing with Deadlines

✅ Definition

The **Job Sequencing with Deadlines** problem is an optimization problem where:

- You are given a list of **jobs**.
- Each job has a **deadline** and a **profit**.
- Each job takes **one unit of time** to complete.
- **Only one job can be done at a time**.

Your goal is to **schedule jobs** in such a way that the **total profit is maximized** and **no job is scheduled after its deadline**.

💡 Approach (Greedy Algorithm)

The Greedy approach schedules the **most profitable jobs first**, ensuring they are done **before their deadlines**, if possible.

🔄 Steps:

1. **Sort** all jobs in **descending order of profit**.
2. Create a time slot array of size = max deadline.
3. For each job:
    - Try to place it in the latest **available time slot ≤ its deadline**.
    - If a slot is found, schedule the job.
4. Repeat until all jobs are checked or all slots are filled.

📌 Algorithm (Pseudocode)

plaintext

Copy code

```
function jobSequencing(jobs):
    sort jobs by descending profit
    maxDeadline = max(jobs.deadline)
    slots = array of size maxDeadline, initialized to None

    for job in jobs:
        for j in range(min(job.deadline, maxDeadline) - 1, -1, -1):
            if slots[j] is None:
                slots[j] = job
                break

    return slots, total profit
```

✅ Advantages

Advantage	Description
⏱️ Fast and efficient	O(n log n) due to sorting
🧠 Simple logic	Easy to implement using greedy
💰 Maximizes profit	Selects best jobs based on returns
📦 Good for scheduling	Used in OS task scheduling, manufacturing

---

❌ Disadvantages

Disadvantage	Explanation
❌ Assumes unit job time	Doesn’t handle variable-duration jobs
❌ Needs sorting	Sorting overhead for large inputs
❌ Not always optimal	For jobs with dependencies or setup times
❌ Deadlines must be known	Can’t work in real-time dynamic scenarios

---

🧑‍💻 Real-Life Applications

- Job/task scheduling in operating systems
- Manufacturing & assembly line optimization
- CPU time slot allocation
- Advertising slot scheduling

📝 Summary

Feature	Description
Problem Type	Greedy – maximize profit with constraints
Time Complexity	O(n log n + n×d) where d = max deadline
Output	Job order that gives maximum profit
Input Assumption	Jobs take 1 unit time and no overlap

Let me know if you want the **code implementation** in a specific language, a **flowchart**, or more **practice questions**!

You said:

question n=4,[p1,p2,p3,p4]=[70,12,18,35]and[d1,d2,d3,d4]=[2,1,2,1]

ChatGPT said:

Let's solve the **Job Sequencing with Deadlines** problem using the given values:

🧮 Given:

- **Number of Jobs, n = 4**
- **Profits = [70, 12, 18, 35]**
- **Deadlines = [2, 1, 2, 1]**

We can denote the jobs as:

Job	Profit (P)	Deadline (D)
J1	70	2
J2	12	1
J3	18	2
J4	35	1

✅ Step 1: Sort Jobs by Profit (Descending)

Sorted list:

Job	Profit	Deadline
J1	70	2
J4	35	1
J3	18	2
J2	12	1

✅ Step 2: Initialize Time Slots

- Max deadline = **2**
- Create slots = [None, None] → represents slot 1 and slot 2

✅ Step 3: Assign Jobs to Latest Possible Slot ≤ Deadline

🔹 **Try J1 (Profit 70, Deadline 2):**

- Slot 2 is free → assign J1 to slot 2
  ➡️ slots = [None, J1]

🔹 **Try J4 (Profit 35, Deadline 1):**

- Slot 1 is free → assign J4 to slot 1
  ➡️ slots = [J4, J1]

🔹 **Try J3 (Profit 18, Deadline 2):**

- Slot 2 is taken → check slot 1 → also taken
  ➡️ Cannot schedule J3

🔹 **Try J2 (Profit 12, Deadline 1):**

- Slot 1 is taken
  ➡️ Cannot schedule J2

🏁 Result

- **Scheduled Jobs:** J4, J1 (at slot 1 and 2)
- **Total Profit:** 35 + 70 = **₹105**

✅ Final Job Schedule:

Slot	Job	Profit
1	J4	35
2	J1	70

🌲 Minimum-Cost Spanning Tree (MST)

✅ Definition:

A **Minimum-Cost Spanning Tree (MST)** is a subset of the edges of a **connected, undirected, weighted graph** that:

- Connects **all the vertices** together,
- **Has no cycles** (i.e., it's a tree),
- And has the **minimum possible total edge weight** among all such spanning trees.

✨ Key Properties of an MST:

- Includes **all vertices** (V) in the graph.
- Contains **V - 1 edges** (no cycles).
- Total edge weight is **minimized**.
- There may be **multiple MSTs** if weights allow ties.

🧠 Why MST is Important (Applications)

- Network design (e.g., wiring, computer networks, roads).
- Reducing cost in connecting all components.
- Clustering data in machine learning.
- Image processing (e.g., segmentation).

🔧 Two Main MST Algorithms:

Algorithm	Basis	Best for
Prim’s	Grow a single tree	Dense graphs
Kruskal’s	Build forest, then merge	Sparse graphs

🌲 Prim’s Algorithm (Greedy Tree Expansion)

✅ Definition

Prim’s Algorithm is a **greedy algorithm** used to find the **Minimum Spanning Tree** (MST) of a **connected, undirected, weighted graph** by **growing the MST one vertex at a time**, always choosing the **minimum weight edge** that connects a visited node to an unvisited node.

💡 Key Idea

- Start from any vertex.
- Repeatedly add the **cheapest edge** connecting the tree to a vertex not in the tree.
- Continue until **all vertices** are included in the MST.

🔄 Step-by-Step Approach

1. Initialize:

- Start from any arbitrary vertex.
- Mark it as visited.
- Use a **min-heap** (priority queue) to keep track of the **cheapest edge** to an unvisited vertex.

2. Loop:

- Pick the **minimum-weight edge** from the heap that connects to an unvisited vertex.
- Add that vertex to the MST.
- Mark it as visited.
- Add all its outgoing edges (to unvisited vertices) to the heap.
- Repeat until all vertices are included.

📦 Data Structures Used:

Structure	Purpose
Min-Heap / Priority Queue	To efficiently select the next minimum edge
Visited Array	To track which nodes are already in MST
Adjacency List	To represent the graph efficiently

🧠 Time Complexity

Version	Time Complexity
Naive (adj. matrix)	$O(V^2)$
With Min-Heap & List	$O(E \log V)$

🧮 Example

Graph:

mathematica

Copy code

Vertices: A, B, C, D

Edges:

A-B (1)

A-C (3)

B-C (1)

B-D (6)

C-D (4)

🧭 Execution (Start from A):

1. A is added to MST
2. Add edges A-B(1), A-C(3) to heap → pick A-B
3. Add B to MST

4. Add B-C(1), B-D(6) to heap → pick B-C
5. Add C to MST
6. Add C-D(4) → pick it
7. Add D to MST

✅ Final MST Edges:

- A-B (1), B-C (1), C-D (4)
- **Total Cost = 6**

🔧 Prim's Algorithm - Pseudocode

plaintext

Copy code

```
function PrimMST(graph):
  Initialize: visited = [False] * n
              minHeap = []
              MST = []

  Add (0, source_node) to minHeap

  while minHeap is not empty:
    (weight, u) = minHeap.pop()

    if u is already visited:
      continue

    mark u as visited
    add edge to MST
```

```
        for all neighbors v of u:
            if v not visited:
                push (weight(u, v), v) to minHeap

    return MST
```

✅ Advantages of Prim's Algorithm

Advantage	Description
✅ Efficient on dense graphs	Because it avoids sorting all edges
✅ Builds tree incrementally	Easy to track and control growth
✅ Deterministic result	Produces consistent MST with same start

---

❌ Disadvantages

Disadvantage	Explanation
❌ Needs extra data structures	Requires priority queue & visited tracking
❌ Slower on sparse graphs	Compared to Kruskal in some cases
❌ Only for connected graphs	Doesn't handle disconnected graphs

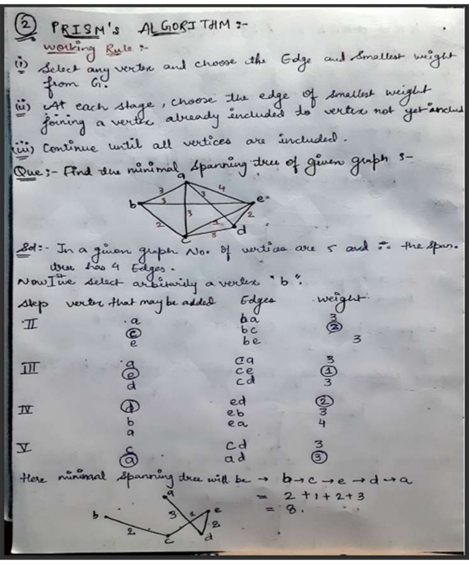
---

📘 Use Cases

- Designing least-cost communication networks
- Laying cables and pipes efficiently
- Road or electrical wiring planning
- Image segmentation

Summary

Feature	Description
Algorithm Type	Greedy
Graph Type	Undirected, connected, weighted
Data Structures Used	Min-heap (priority queue), visited array
Time Complexity	O(E log V) with binary heap
Output	Minimum-cost spanning tree



② PRISM's ALGORITHM :-

working Rule :-

(i) Select any vertex and choose the Edge and Smallest weight from G.

(ii) At each stage, choose the edge of smallest weight joining a vertex already included to vertex not yet includ

(iii) Continue until all vertices are included.

Que :- Find the minimal spanning tree of given graph :-

a
b
c
d
e
3
3
4
3
3
2
1
2
3

Sol :- In a given graph No. of vertices are 5 and ∴ the span. tree has 4 Edges.

Now Ist we select arbitarily a vertex "b".

step	vertex that may be added	Edges	weight
II	a ⓒ e	ba bc be	3 ② 3
III	a ⓔ d	ca ce cd	3 ① 3
IV	ⓓ b a	ed eb ea	② 3 4
V	c ⓐ	cd ad	3 ③

Here minimal spanning tree will be → b → c → e → d → a

= 2 + 1 + 2 + 3

= 8.

Definition

Kruskal's Algorithm is a **greedy algorithm** used to find the **Minimum Spanning Tree (MST)** of a **connected, undirected, weighted graph** by **selecting the lowest-weight edges first** and ensuring that no cycles are formed.

💡 Key Idea

- Sort all edges in **increasing order of weight**.
- Use a **Disjoint Set (Union-Find)** data structure to check for cycles.
- Repeatedly add the **smallest edge** that doesn't form a cycle until MST is complete.

🧱 Concept: Spanning Tree

A **spanning tree**:

- Connects all vertices.
- Has no cycles.
- Has exactly **V - 1** edges for **V** vertices.

🔄 Step-by-Step Approach

1. Sort all edges in non-decreasing order by weight.

2. Initialize:

- Each vertex is its own **disjoint set** (component).
- Create an empty list for MST.

3. Iterate over sorted edges:

- For each edge (u, v):
    - If u and v belong to **different sets** (no cycle), add edge to MST.
    - Merge the sets using **Union-Find**.

4. Repeat until MST contains V - 1 edges.

🧠 Time Complexity

Step	Complexity
Sorting edges	O(E log E)

Step	Complexity
Union-Find operations	O(E log V) (with path compression + union by rank)
Overall	**O(E log V)**

📦 Data Structures Used

Structure	Purpose
Edge List	To sort and iterate edges by weight
Union-Find (DSU)	To detect cycles efficiently
MST Edge List	To store result edges

🔄 Example

Graph:

mathematica

Copy code

Vertices: A, B, C, D

Edges:

A-B (1)

A-C (3)

B-C (1)

B-D (6)

C-D (4)

✅ Step 1: Sort edges

Sorted by weight:

- A-B (1)
- B-C (1)

- A-C (3)
- C-D (4)
- B-D (6)

✅ **Step 2: Initialize DSU**

✅ **Step 3: Add edges to MST**

- A-B → ✅ (no cycle)
- B-C → ✅ (no cycle)
- A-C → ❌ (would form cycle)
- C-D → ✅ (no cycle)
- B-D → ❌ (already connected)

✅ **Final MST:**

- Edges: A-B (1), B-C (1), C-D (4)
- **Total cost: 6**

📝 Pseudocode of Kruskal's Algorithm

plaintext

Copy code

```
function KruskalMST(graph):
    result = []
    sort all edges in increasing weight
    create disjoint sets for all vertices

    for each edge (u, v) in sorted edge list:
        if find(u) ≠ find(v):
            add edge to result
        union(u, v)

    return result
```

✅ Advantages of Kruskal's Algorithm

Advantage	**Description**
✅ Efficient for sparse graphs	Fewer edges to check, especially in graphs with V >> E
✅ Simple logic	Straightforward to understand and implement
✅ Handles disconnected graphs	Can find **minimum spanning forest**

❌ Disadvantages

Disadvantage	**Explanation**
❌ Requires edge sorting	Sorting edges adds O(E log E) overhead
❌ Needs Union-Find structure	More complex implementation than Prim's basic form
❌ Slower on dense graphs	More edges = more sorting and comparisons

---

📘 Applications of Kruskal's Algorithm

- Network design (LAN, road, power grid)
- Image segmentation
- Clustering in machine learning
- Finding connected components

🧾 Kruskal's vs. Prim's Algorithm

Feature	**Kruskal’s Algorithm**	**Prim’s Algorithm**
Strategy	Edge-based	Vertex-based
Suitable for	Sparse graphs	Dense graphs
Data Structure	Disjoint Set (Union-Find)	Priority Queue (Min Heap)
Time Complexity	O(E log V)	O(E log V)
Cycle Detection	Union-Find	Visited array

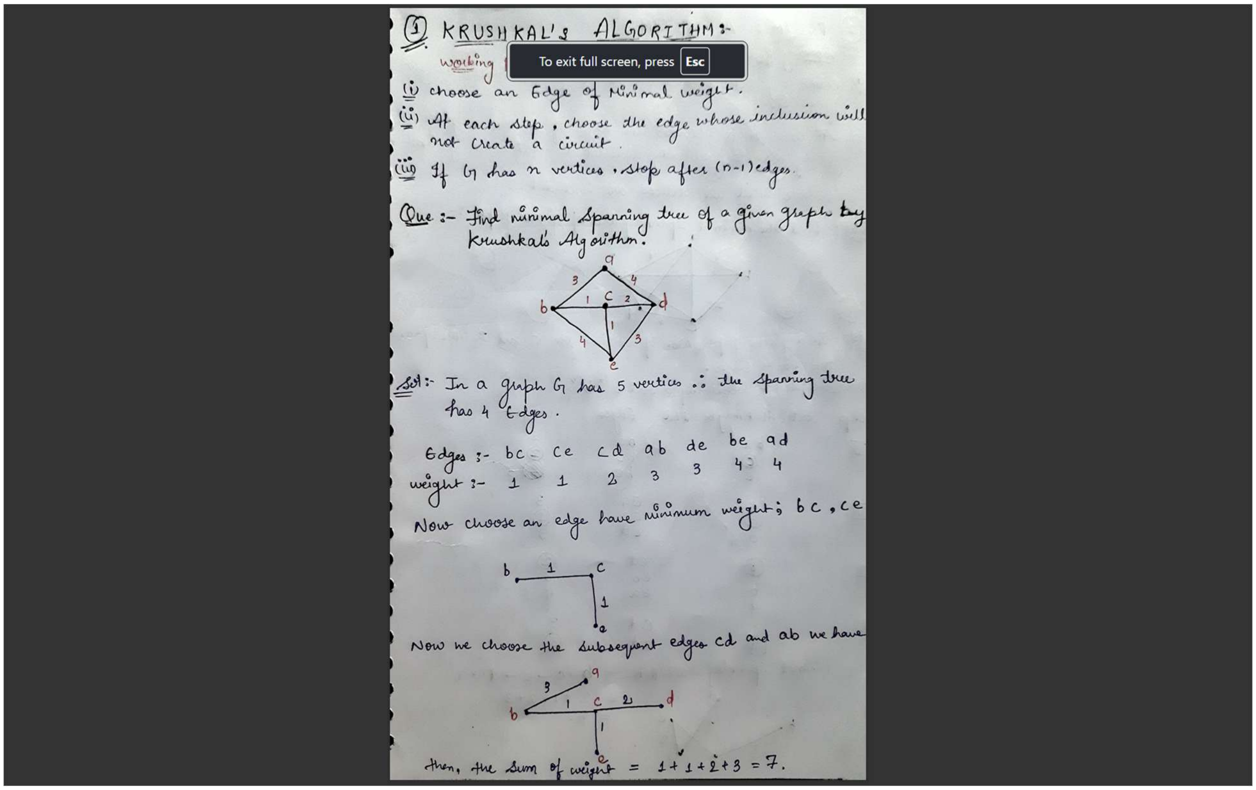
📎 Summary

- **Kruskal’s Algorithm** builds MST by **selecting the smallest edges** while **avoiding cycles**.
- It is **efficient for sparse graphs** and often easier to implement than Prim’s for edge list representations.
- Uses **Union-Find** to maintain disjoint sets for cycle checking.

Would you like:

- ✅ Python / C++ / Java code?
- ✅ A visual diagram for Kruskal’s algorithm?
- ✅ Practice problems with solutions?

Let me know how you'd like to continue!



① KRUSHKAL'S ALGORITHM :-

working

(i) choose an Edge of Minimal weight.

(ii) At each step, choose the edge whose inclusion will not create a circuit.

(iii) If G has n vertices, stop after (n-1) edges.

Que :- Find minimal Spanning tree of a given graph by Krushkal's Algorithm.

Sol :- In a graph G has 5 vertices ∴ the spanning tree has 4 Edges.

Edges :-	bc	ce	cd	ab	de	be	ad
weight :-	1	1	2	3	3	4	4

Now choose an edge have minimum weight ; bc , ce

Now we choose the subsequent edges cd and ab we have

then, the sum of weight = 1 + 1 + 2 + 3 = 7.

✅ Definition

Dijkstra's Algorithm is a **greedy algorithm** that finds the **shortest path from a single source vertex** to **all other vertices** in a **weighted graph** (with **non-negative edge weights**).

💡 Key Idea

- Continuously select the **closest unvisited node** and update the shortest distances to its neighbors.
- Repeat until all vertices are visited.

📋 Assumptions

- Works on **directed or undirected graphs**
- All **edge weights must be ≥ 0** (non-negative)
- Input: Graph G(V, E), source vertex S

🔧 Data Structures Used

Structure	Purpose
Distance Array	Store shortest known distances
Priority Queue	Get the next closest node efficiently
Visited Set	Keep track of already processed nodes

🧠 Algorithm Steps

Step 1: Initialize

- dist[source] = 0
- dist[others] = ∞
- Use a **min-priority queue** to pick the next vertex with the **minimum distance**.

Step 2: Loop until the queue is empty

- Extract the node u with the smallest dist[u]
- For each neighbor v of u:
    - If dist[v] > dist[u] + weight(u, v):
        - Update dist[v]
        - Push (dist[v], v) into the priority queue

Step 3: Repeat until all nodes have been visited.

💻 Python Pseudocode

python

Copy code

```
import heapq


def dijkstra(graph, source):
    n = len(graph)
    dist = [float('inf')] * n
    dist[source] = 0
    pq = [(0, source)]  # (distance, node)

    while pq:
        d, u = heapq.heappop(pq)
        if d > dist[u]:  # Already found better path
            continue
        for v, w in graph[u]:
            if dist[v] > dist[u] + w:
                dist[v] = dist[u] + w
                heapq.heappush(pq, (dist[v], v))
    return dist
```

🔄 Example

📊 Graph

Let’s say we have the following graph with 5 nodes (A–E = 0–4):

mathematica

Copy code

```
    A ---2--- B
```

```
  |      / |
  4    1   3
  |  /    |
  C ---5--- D
   \     /
    --1-- E
```

Edges (with weights):

A–B (2), A–C (4), B–C (1), B–D (3), C–D (5), C–E (1), D–E (2)

Step-by-step from source A (0):

Iteration	**Node**	**Distance Updates**	**Final dist[]**
Init	—	A=0, others=∞	[0, ∞, ∞, ∞, ∞]
1	A	B=2, C=4	[0, 2, 4, ∞, ∞]
2	B	C=3, D=5	[0, 2, 3, 5, ∞]
3	C	E=4	[0, 2, 3, 5, 4]
4	E	D=4	[0, 2, 3, 4, 4]
5	D	—	Final distances

✅ Final shortest distances from A:

mathematica

Copy code

A → A = 0

A → B = 2

A → C = 3

A → D = 4

A → E = 4

📈 Time Complexity

Implementation Type	**Time Complexity**
Simple array (no heap)	$O(V^2)$
Binary heap + adjacency list	**$O((V + E) \log V)$**
Fibonacci heap (advanced)	$O(E + V \log V)$

✅ Advantages

- Efficient for **sparse graphs** with **non-negative weights**
- Always finds the **shortest path**
- Simple and intuitive

❌ Disadvantages

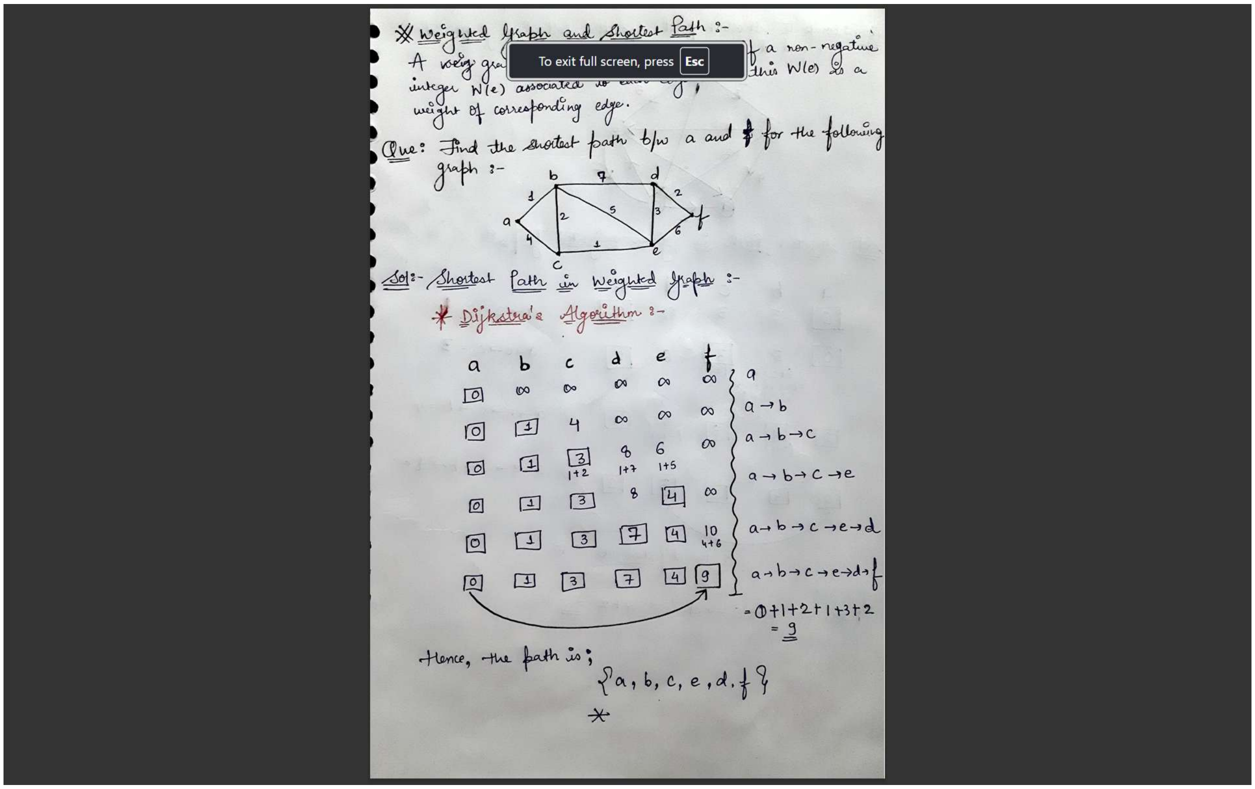
- **Does not work with negative edge weights**
- Slightly **slower than A*** in some heuristic problems
- Not suitable for **dynamic graphs** unless recomputed

📦 Applications

- GPS / Navigation systems
- Network routing (e.g., OSPF, RIP)
- Game AI pathfinding
- Robot motion planning
- Real-time traffic apps

📌 Summary

Feature	Description
Input	Graph + source vertex
Output	Shortest path from source to all others
Type	Greedy algorithm
Limitation	Cannot handle negative weights
Efficiency	O((V + E) log V) with a min-heap


To exit full screen, press Esc
b
7
d
1
2
a
2
5
3
f
4
6
1
e
c

※ **Weighted Graph and Shortest Path :-**

A weig gra f a non-negative
integer W(e) associated this W(e) is a
weight of corresponding edge.

Que: Find the shortest path b/w a and f for the following graph :-

Sol:- **Shortest Path in Weighted Graph :-**

* **Dijkstra's Algorithm :-**

a	b	c	d	e	f	
$\boxed{0}$	∞	∞	∞	∞	∞	a
$\boxed{0}$	$\boxed{1}$	4	∞	∞	∞	a → b
$\boxed{0}$	$\boxed{1}$	$\boxed{3}$ 1+2	8 1+7	6 1+5	∞	a → b → c
$\boxed{0}$	$\boxed{1}$	$\boxed{3}$	8	$\boxed{4}$	∞	a → b → c → e
$\boxed{0}$	$\boxed{1}$	$\boxed{3}$	$\boxed{7}$	$\boxed{4}$	10 4+6	a → b → c → e → d
$\boxed{0}$	$\boxed{1}$	$\boxed{3}$	$\boxed{7}$	$\boxed{4}$	$\boxed{9}$	a → b → c → e → d → f

= ⓪ + 1 + 2 + 1 + 3 + 2
= 9

Hence, the path is;

{a, b, c, e, d, f}

*