Greedy Technique

Varun Kumar

July 5, 2025

1. Logic

Greedy algorithms build up a solution step by step by choosing the best available option at each stage. They don't backtrack or revise previous choices.

Key Idea

At each step, choose the best local option with the hope that it leads to a globally optimal solution.

2. When to Use

- The problem has the **Greedy Choice Property**.
- The problem exhibits **Optimal Substructure**.
- Fast approximation is acceptable.

3. General Pseudocode

```
# Generic structure of a greedy algorithm
def greedyAlgorithm(problem):
    solution = []
    while not problem.complete():
        candidate = select_best_candidate(problem)
        if is_feasible(candidate):
            solution.append(candidate)
    return solution
```

4. Example 1: Activity Selection

Problem

Given n activities with start and finish times, select the maximum number of non-overlapping activities.

Greedy Strategy

Always pick the activity that finishes earliest.

Example

Given: `[(1, 3), (2, 5), (4, 7), (1, 8), (5, 9), (8, 10), (9, 11), (11, 14), (13, 16)]`

- Sort by finish time: `(1, 3), (2, 5), (4, 7), (1, 8), (5, 9), (8, 10), (9, 11), (11, 14), (13, 16)`
- Select (1, 3) $\rightarrow$ End time = 3
- Next valid = (4, 7) $\rightarrow$ End = 7
- Next = (8, 10) $\rightarrow$ End = 10
- Next = (11, 14) $\rightarrow$ End = 14

Selected activities: `[(1, 3), (4, 7), (8, 10), (11, 14)]`

Python Code

```python
def activity_selection(activities):
    # Sort activities by their end time
    activities.sort(key=lambda x: x[1])

    selected = []
    end_time = 0

    for start, finish in activities:
        # If current activity starts after or at the end of last selected
        if start >= end_time:
            selected.append((start, finish)) # Select activity
            end_time = finish # Update end time

    return selected
```

How to Give Input

The input should be a list of tuples, where each tuple represents an activity with its start and end times.

```python
activities = [(1, 3), (2, 5), (4, 7), (1, 8),
              (5, 9), (8, 10), (9, 11),
              (11, 14), (13, 16)]
result = activity_selection(activities)
print(result)
```

5. Example 2: Huffman Coding

Problem

Given characters and their frequencies, build an optimal prefix code that minimizes total encoding length.

Greedy Strategy

Merge the two least frequent nodes at each step to build the tree.

Example

Characters and Frequencies:

Character	Frequency
A	5
B	9
C	12
D	13
E	16
F	45

- Step 1: Combine A(5) + B(9) $\rightarrow$ Node(14)
- Step 2: Combine C(12) + D(13) $\rightarrow$ Node(25)
- Step 3: Combine Node(14) + E(16) $\rightarrow$ Node(30)
- Step 4: Combine Node(25) + Node(30) $\rightarrow$ Node(55)
- Step 5: Combine Node(55) + F(45) $\rightarrow$ Node(100)

Result: A binary tree with optimal prefix codes based on traversal.

Python Code

```
import heapq

# Define a tree node
class Node:
    def __init__(self, char, freq):
        self.char = char # Character (None for internal nodes)
        self.freq = freq # Frequency of the character
        self.left = None # Left child
        self.right = None # Right child

    # Comparison operator for priority queue (min-heap)
    def __lt__(self, other):
        return self.freq < other.freq

def huffman_coding(char_freq):
    # Create a heap with one node per character
    heap = [Node(c, f) for c, f in char_freq.items()]
    heapq.heapify(heap)

    # Build the Huffman Tree
    while len(heap) > 1:
        left = heapq.heappop(heap) # Least frequent node
        right = heapq.heappop(heap) # Next least frequent

        # Merge both nodes
        merged = Node(None, left.freq + right.freq)
        merged.left = left
        merged.right = right

        heapq.heappush(heap, merged) # Insert merged node back

    return heap[0] # Return the root of the Huffman Tree
```

How to Give Input

The input should be a dictionary where the keys are characters and values are their corresponding frequencies.

```
char_freq = {
    'A': 5,
    'B': 9,
    'C': 12,
    'D': 13,
    'E': 16,
    'F': 45
}

root = huffman_coding(char_freq)
# To generate the codes, perform a traversal on the tree.
```

6. Example 3: Fractional Knapsack

Problem

Given n items with value and weight, and knapsack capacity W, maximize value with possible fractional items.

Greedy Strategy

Pick items in descending order of value-to-weight ratio.

Example

Items:

Item	Value	Weight
1	60	10
2	100	20
3	120	30

Knapsack capacity = 50

- Compute value/weight ratios: Item1 = 6, Item2 = 5, Item3 = 4
- Sort by ratio: Item1, Item2, Item3
- Take all of Item1 $\rightarrow$ Remaining = 40
- Take all of Item2 $\rightarrow$ Remaining = 20
- Take 20/30 of Item3 = $120 \times \frac{2}{3} = 80$

Total value: $60 + 100 + 80 = 240$

Python Code

```
def fractional_knapsack(capacity, items):
    # items: list of tuples (value, weight)

    # Sort items by value/weight ratio in descending order
    items.sort(key=lambda x: x[0]/x[1], reverse=True)

    total_value = 0.0 # Store maximum total value

    for value, weight in items:
        if capacity >= weight:
            # Take full item
            total_value += value
            capacity -= weight
        else:
            # Take fractional part
            total_value += value * (capacity / weight)
            break # Knapsack is full

    return total_value
```

How to Give Input

The input should be a list of tuples where each tuple is (value, weight), and an integer for knapsack capacity.

```
items = [(60, 10), (100, 20), (120, 30)]
capacity = 50

max_value = fractional_knapsack(capacity, items)
print(max_value)
```

7. Example 4: Job Sequencing with Deadline

Problem Statement

You are given n jobs. Each job has:

- A unique identifier (Job ID)
- A deadline by which it must be finished
- A profit associated with completing it on or before the deadline

Each job takes 1 unit of time. The goal is to schedule jobs to maximize total profit such that no two jobs overlap and each is finished by its deadline.

Greedy Strategy

- Sort all jobs in descending order of profit
- Iterate through each job and try to place it in the latest available slot before its deadline
- Use a time slot array to track which time units are occupied

Key Idea

Choose the most profitable jobs first and place them as late as possible (before or on their deadline) to keep earlier slots open for other jobs.

Example

Jobs:

Job ID	Deadline	Profit
J1	2	100
J2	1	19
J3	2	27
J4	1	25
J5	3	15

- Sort by profit: J1, J3, J4, J2, J5
- Schedule J1 at slot 2
- Schedule J3 at slot 1
- J4 and J2 can't be scheduled (slots full)
- Schedule J5 at slot 3

Total Profit: $100 + 27 + 15 = 142$ **Selected Jobs:** `[J3, J1, J5]` in slots `[1, 2, 3]`

Python Code

```
# Job class to store each job's properties
class Job:
    def __init__(self, job_id, deadline, profit):
        self.id = job_id
        self.deadline = deadline
        self.profit = profit

def job_sequencing(jobs):
    # Sort jobs by descending profit
    jobs.sort(key=lambda x: x.profit, reverse=True)

    # Find maximum deadline to size the time slot array
    max_deadline = max(job.deadline for job in jobs)
    slots = [False] * (max_deadline + 1) # 1-based indexing
    result = [None] * (max_deadline + 1)

    total_profit = 0

    for job in jobs:
        # Try to find a free slot from job.deadline down to 1
        for t in range(job.deadline, 0, -1):
            if not slots[t]:
                slots[t] = True
                result[t] = job.id
                total_profit += job.profit
                break

    scheduled_jobs = [job_id for job_id in result if job_id is not None]
    return scheduled_jobs, total_profit
```

How to Give Input

You need to define a list of 'Job' objects with their IDs, deadlines, and profits.

```
# Define jobs
jobs = [
    Job("J1", 2, 100),
    Job("J2", 1, 19),
    Job("J3", 2, 27),
    Job("J4", 1, 25),
    Job("J5", 3, 15)
]

# Run algorithm
scheduled, profit = job_sequencing(jobs)

# Output
print("Scheduled Jobs:", scheduled)
print("Total Profit:", profit)
```

8. Example 5: Minimal Cost Spanning Tree (MCST)

Problem Statement

Given a connected, undirected, weighted graph, find a spanning tree (subset of edges) that:

- Connects all the vertices (no cycles)
- Has the minimum possible total edge weight

This is known as a **Minimum Spanning Tree (MST)**.

Prim's Algorithm

Greedy Strategy

- Start from any node
- Always pick the minimum-weight edge that connects a visited vertex to an unvisited vertex
- Use a priority queue (min-heap) to efficiently find the next edge

Key Idea

Build the MST incrementally by always adding the smallest edge connecting the tree to a new vertex.

Example

Graph with 5 vertices and following weighted edges:

From	To	Weight
0	1	2
0	3	6
1	2	3
1	3	8
1	4	5
2	4	7
3	4	9

Output: Total cost = 16 Edges: (0-1), (1-2), (1-4), (0-3)

Python Code

```
import heapq
from collections import defaultdict

def prims_mst(graph, start=0):
    visited = set()
    min_heap = [(0, start)] # (cost, node)
    total_cost = 0

    while min_heap:
        cost, u = heapq.heappop(min_heap)
        if u in visited:
            continue
        visited.add(u)
        total_cost += cost
        for v, weight in graph[u]:
            if v not in visited:
                heapq.heappush(min_heap, (weight, v))

    return total_cost
```

How to Give Input (Prim's)

```
# Define graph as an adjacency list
graph = {
    0: [(1, 2), (3, 6)],
    1: [(0, 2), (2, 3), (3, 8), (4, 5)],
    2: [(1, 3), (4, 7)],
    3: [(0, 6), (1, 8), (4, 9)],
    4: [(1, 5), (2, 7), (3, 9)]
}

# Run Prim's algorithm
cost = prims_mst(graph, start=0)
print("Total cost of MST:", cost)
```

Kruskal's Algorithm

Greedy Strategy

- Sort all edges by weight
- Use Disjoint Set Union (DSU) to check whether adding an edge creates a cycle
- Add edge only if it connects two different components

Key Idea

Always pick the smallest edge that doesn't form a cycle with the edges already chosen.

Example

Same graph as in Prim's Example.

Sorted Edges: (0-1), (1-2), (1-4), (0-3), (1-3), (2-4), (3-4)
Selected Edges: (0-1), (1-2), (1-4), (0-3)
Total Cost: 16

Python Code

```python
class DSU:
    def __init__(self, n):
        self.parent = list(range(n))

    def find(self, u):
        if self.parent[u] != u:
            self.parent[u] = self.find(self.parent[u])
        return self.parent[u]

    def union(self, u, v):
        pu, pv = self.find(u), self.find(v)
        if pu == pv:
            return False
        self.parent[pu] = pv
        return True

def kruskal_mst(n, edges):
    edges.sort(key=lambda x: x[2]) # Sort by weight
    dsu = DSU(n)
    total_cost = 0

    for u, v, weight in edges:
        if dsu.union(u, v):
            total_cost += weight

    return total_cost
```

How to Give Input (Kruskal's)

```
# Define number of nodes
n = 5

# Define edges as (u, v, weight)
edges = [
    (0, 1, 2),
    (0, 3, 6),
    (1, 2, 3),
    (1, 3, 8),
    (1, 4, 5),
    (2, 4, 7),
    (3, 4, 9)
]

# Run Kruskal's algorithm
cost = kruskal_mst(n, edges)
print("Total cost of MST:", cost)
```

9. Example 6: Dijkstra's Shortest Path Algorithm

Problem Statement

Given a graph with non-negative edge weights, find the shortest distance from a source vertex to all other vertices in the graph.

Input: A connected, weighted, directed/undirected graph with n vertices and m edges.

Output: Shortest distance from the source vertex to all other vertices.

Greedy Strategy

- Initialize distances from the source to all vertices as infinity, except the source itself (0).
- Use a priority queue (min-heap) to always expand the vertex with the smallest known distance.
- Update distances to its neighbors if a shorter path is found via the current node.
- Repeat until all vertices are processed.

Key Idea

Always expand the node with the smallest known distance. Once a node is finalized (visited), its shortest path is guaranteed.

Example

Graph:

From	To	Weight
0	1	4
0	2	1
2	1	2
1	3	1
2	3	5
3	4	3

Source: 0
Distances from 0: $0 \to 0$, $1 \to 3$ (via 2), $2 \to 1$, $3 \to 4$, $4 \to 7$

Python Code

```
import heapq
from collections import defaultdict

def dijkstra(graph, source):
    # Initialize distance to all nodes as infinity
    dist = {node: float('inf') for node in graph}
    dist[source] = 0

    # Min-heap to get the node with the smallest distance
    heap = [(0, source)]

    while heap:
        current_dist, u = heapq.heappop(heap)

        # Skip if we already found a better path
        if current_dist > dist[u]:
            continue

        for v, weight in graph[u]:
            if dist[u] + weight < dist[v]:
                dist[v] = dist[u] + weight
                heapq.heappush(heap, (dist[v], v))

    return dist
```

How to Give Input

```
# Define graph as adjacency list
graph = {
    0: [(1, 4), (2, 1)],
    1: [(3, 1)],
    2: [(1, 2), (3, 5)],
    3: [(4, 3)],
    4: []
}

source = 0
shortest_paths = dijkstra(graph, source)

# Output the result
for node in sorted(shortest_paths):
    print(f"Distance␣from␣{source}␣to␣{node}␣is␣{shortest_paths[node]}")
```

Python Code (Adjacency Matrix)

```
import heapq

def dijkstra_matrix(adj_matrix, source):
    n = len(adj_matrix)
    dist = [float('inf')] * n
    visited = [False] * n
    dist[source] = 0

    min_heap = [(0, source)] # (distance, node)

    while min_heap:
        d, u = heapq.heappop(min_heap)

        if visited[u]:
            continue
        visited[u] = True

        for v in range(n):
            weight = adj_matrix[u][v]
            if weight != 0 and not visited[v]:
                if dist[u] + weight < dist[v]:
                    dist[v] = dist[u] + weight
                    heapq.heappush(min_heap, (dist[v], v))

    return dist
```

How to Give Input (Adjacency Matrix)

```
# 0 means no direct edge between the nodes
adj_matrix = [
    [0, 4, 1, 0, 0],
    [0, 0, 0, 1, 0],
    [0, 2, 0, 5, 0],
    [0, 0, 0, 0, 3],
    [0, 0, 0, 0, 0]
]

source = 0
distances = dijkstra_matrix(adj_matrix, source)
# Output the result
for i, d in enumerate(distances):
    print(f"Distance from {source} to {i} is {d}")
```

Comparison: Adjacency List vs Adjacency Matrix

Feature	**Adjacency List**	**Adjacency Matrix**
Memory Efficiency	Efficient for sparse graphs	Wastes space in sparse graphs due to many 0s
Edge Lookup Time	Slower — requires looping through neighbors	Faster — direct access in $O(1)$ time
Implementation	Easier for dynamic graphs	Easier for fixed-size graphs
Best Use Case	Sparse graphs (fewer edges)	Dense graphs (many edges)
Time Complexity	$O((V+E)\log V)$ using heap	$O(V^2)$ in worst case

10. Understanding Edge Relaxation in Dijkstra's Algorithm

Relaxation is the process of updating the shortest known distance to a vertex if a shorter path is found.

What is Edge Relaxation?

For an edge (u, v) with weight w, relaxation checks if going from source $\to u \to v$ is shorter than the currently known distance to v:

if `dist[u] + w < dist[v]` then update `dist[v] = dist[u] + w`

This ensures that we always maintain the shortest known distance to each node.

Relaxation Line in Adjacency List Code

```
if dist[u] + weight < dist[v]:
    dist[v] = dist[u] + weight # <-- Relaxation happens here
    heapq.heappush(heap, (dist[v], v))
```

Relaxation Line in Adjacency Matrix Code

```
if dist[u] + weight < dist[v]:
    dist[v] = dist[u] + weight # <-- Relaxation happens here
    heapq.heappush(min_heap, (dist[v], v))
```

> **Note**
>
> Relaxation is the heart of Dijkstra's algorithm. It is applied every time a shorter path to a neighbor is found via the current node.

11. Time & Space Complexity Summary

Problem	**Time**	**Space**	**Optimal**
Activity Selection	$O(n \log n)$	$O(1)$	Yes
Huffman Coding	$O(n \log n)$	$O(n)$	Yes
Fractional Knapsack	$O(n \log n)$	$O(1)$	Yes
Job Sequencing with Deadline	$O(n \log n)$	$O(n)$	Yes
Prim's Algorithm (List)	$O((V + E) \log V)$	$O(V + E)$	Yes
Prim's Algorithm (Matrix)	$O(V^2)$	$O(V^2)$	Yes
Kruskal's Algorithm	$O(E \log E)$	$O(V)$ (DSU)	Yes
Dijkstra (Adjacency List)	$O((V + E) \log V)$	$O(V + E)$	Yes
Dijkstra (Adjacency Matrix)	$O(V^2)$	$O(V^2)$	Yes

12. Points to Remember

- Greedy algorithms make a locally optimal choice at each step with the hope of reaching a global optimum.
- They do not always work — correctness depends on satisfying the **Greedy Choice Property** and **Optimal Substructure**.
- Sorting is often the first step (e.g., by profit, deadline, frequency, ratio, etc.).
- Greedy algorithms are usually more efficient than dynamic programming.
- Many problems (e.g., Fractional Knapsack) can be solved using greedy methods, but not all (e.g., 0/1 Knapsack).
- Activity Selection and Job Sequencing both rely on choosing based on deadline or end time.
- Huffman Coding builds a binary tree using a priority queue (min-heap).
- Fractional Knapsack allows taking parts of an item — hence greedy works perfectly.
- In Job Sequencing, use a time slot array to track available deadlines.
- Prim's Algorithm grows a tree from any starting vertex by choosing the lightest connecting edge.
- Kruskal's Algorithm sorts all edges and builds the MST by adding edges that don't form a cycle.
- Dijkstra's Algorithm uses a min-heap to expand the shortest known node first — it only works for non-negative weights.
- Greedy algorithms typically have lower space complexity ($O(1)$ to $O(n)$).
- Always prove or validate the correctness of a greedy algorithm — don't assume it will always work.
- Common applications include scheduling, compression, spanning trees, and pathfinding.

Final Summary

What You Should Know About Greedy Algorithms

- Greedy algorithms make **locally optimal** choices at each step, aiming for a **globally optimal** solution.
- They are efficient both in time and space, often outperforming dynamic programming in simple problems.
- Correctness depends on:
  - **Greedy Choice Property**
  - **Optimal Substructure**
- Not all problems are solvable using greedy — always analyze if the greedy conditions are met.

Key Problems Solved Using Greedy

- Activity Selection
- Fractional Knapsack
- Huffman Coding
- Job Sequencing with Deadline
- Prim's Algorithm (MST)
- Kruskal's Algorithm (MST)
- Dijkstra's Algorithm (Shortest Path)

Tips

- Sort data first based on the problem requirement (e.g., profit, deadline, ratio).
- Use data structures like heaps or DSU to improve performance.
- Greedy is ideal for real-time and approximation problems.
- Validate greedy correctness via proof or counter-example testing.