Algorithms for Problem Solving

Problem Representation

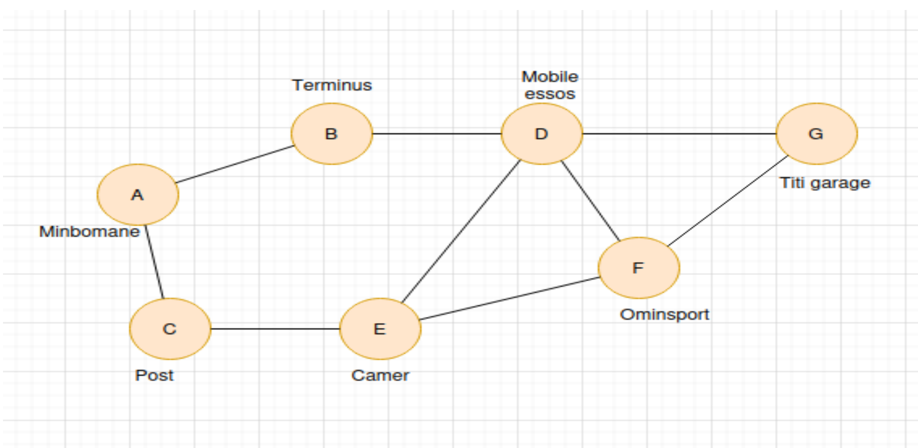
Binary Search

Given a sorted list of integers, efficiently search for a target value.
Example:
Input: [1, 3, 5, 7, 9], Target: 5
Output: Index 2

Graph Traversal (BFS and DFS)

Given a graph representation of a city map, explore connectivity and shortest paths.

Example of graph:



Dynamic Programming (Knapsack Problem)

Maximize total value of items packed in a container with a weight limit.

Example:
Items: [(value: 60, weight: 10), (value: 100, weight: 20), (value: 120, weight: 30)]
Weight Limit: 50
Output: Max Value = 220, object_selected = [(value: 100, weight: 20), (value: 120, weight: 30)]

Merge Intervals

Given a list of time intervals, merge overlapping intervals.

Example:
Input: [(1,3), (2,6), (8,10), (15,18)]
Output: [(1,6), (8,10), (15,18)]

Maximum Subarray Sum (Kadane's Algorithm)

Find the contiguous subarray with the maximum sum.

Example:
Input: [-2, 1, -3, 4, -1, 2, 1, -5, 4]
Output: [4, -1, 2, 1] (Max Sum = 6)

Solution

Binary Search

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

Graph Traversal (BFS and DFS)

```
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])
    while queue:
        node = queue.popleft()
        if node not in visited:
            visited.add(node)
```

```
        queue.extend(graph[node] - visited)
    return visited

def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    for neighbor in graph[start] - visited:
        dfs(graph, neighbor, visited)
    return visited
```

Knapsack Problem

```
def knapsack(values, weights, W):
    n = len(values)
    dp = [[0] * (W + 1) for _ in range(n + 1)]
    for i in range(1, n + 1):
        for w in range(W + 1):
            if weights[i - 1] <= w:
                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]] + values[i - 1])
            else:
                dp[i][w] = dp[i - 1][w]
    return dp[n][W]
```

Merge Intervals

```
def merge_intervals(intervals):
    intervals.sort()
    merged = []
    for interval in intervals:
        if not merged or merged[-1][1] < interval[0]:
            merged.append(interval)
        else:
            merged[-1] = (merged[-1][0], max(merged[-1][1], interval[1]))
    return merged
```

Maximum Subarray Sum (Kadane’s Algorithm)

```
def max_subarray_sum(arr):
    max_sum = float('-inf')
    current_sum = 0
```

```
    start = end = s = 0

    for i in range(len(arr)):
        current_sum += arr[i]
        if current_sum > max_sum:
            max_sum = current_sum
            start, end = s, i
        if current_sum < 0:
            current_sum = 0
            s = i + 1

    return max_sum, arr[start:end+1]
```

Results

```
# Binary Search
print(binary_search([1, 3, 5, 7, 9], 5))  # Output: 2

# BFS
graph = {'A': {'B', 'D'}, 'B': {'A', 'C', 'E'}, 'C': {'B'}, 'D': {'A', 'E'}, 'E': {'B', 'D'}}
print(bfs(graph, 'A'))  # Output: {'A', 'B', 'C', 'D', 'E'}

# Knapsack
print(knapsack([(60, 10), (100, 20), (120, 30)], 50))  # Output: 160, [(100, 20), (120, 30)] #output: 220, [(100, 20), (120, 30)]

# Merge Intervals

items
print(merge_intervals([(1,3), (2,6), (8,10), (15,18)]))  # Output: [(1,6), (8,10), (15,18)]

# Maximum Subarray Sum
arr = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
print(max_subarray_sum(arr))  # Output: (6, [4, -1, 2, 1])
```

Conclusion

Each algorithm provides an optimal solution to its problem:
- Binary Search is efficient with O(log n) complexity.
- Graph Traversal (BFS/DFS) explores connected components in O(V + E).

- Knapsack Problem optimally selects items using O(n * W).
- Merge Intervals sorts and merges efficiently in O(n log n).
- Kadane’s Algorithm finds the max subarray sum in O(n).

These algorithms are fundamental in problem-solving and real-world applications.