Lab 03: Greedy Algorithms

ICT 2202: Algorithm Analysis Design and Lab

Submitted By

Group 1

Tahshin Islam Tuhin(ID: 1957)

Nobin Chandra(ID: 1958)

Md. Shaon Khan(ID: 1984)

Submitted To

Professor Fahima Tabassum, PhD

Institute of Information Technology

Jahangirnagar University

Submission Date: 14/10/2025

Overview:

Algorithms like 0/1 Knapsack, Activity Selection, and graph algorithms such as Prim's, Kruskal's, and Dijkstra's are essential tools for optimization and efficient problem-solving in computer science. The 0/1 Knapsack Problem focuses on selecting items with given costs and values to maximize total value without exceeding a budget, and it can be solved using dynamic programming for exact solutions or greedy approaches for approximations. Similarly, Activity Selection employs a greedy strategy to schedule the maximum number of non-overlapping tasks by prioritizing activities with the earliest finish times, making it useful for scheduling jobs, meetings, or classroom assignments.

In graph theory, Prim's and Kruskal's algorithms are widely used to determine a Minimum Spanning Tree (MST), connecting all vertices in a weighted graph with the minimum total edge weight. Prim's algorithm grows the MST by repeatedly adding the minimum-weight edge from the existing tree to a new vertex, whereas Kruskal's algorithm sorts all edges by weight and adds them sequentially while avoiding cycles. On the other hand, Dijkstra's algorithm finds the shortest path from a source vertex to all other vertices in a weighted graph with non-negative edges by iteratively updating the shortest tentative distances. Together, these techniques are applied in areas such as resource allocation, project planning, network design, and routing, demonstrating the practical value of greedy strategies, dynamic programming, and graph traversal methods in solving real-world optimization problems.

Problem -01: Re-scheduling a presentation list from N-proposed Talk list according to Priority value given by authority for a Academic Conference.

Algorithm-
Algorithm Steps:

1. Sort the talks in descending order according to their value.
   This ensures that higher-value talks are considered first.
2. Initialize an empty list selected to store chosen talks.
3. For each talk in the sorted list:
   - Check if it is compatible with all previously selected talks.
     (Two talks are compatible if their time intervals do not overlap.)
   - If compatible, add it to the selected list.
4. After checking all talks, display the selected talks and calculate the total value.

Time Complexity: O(n^2) in worst and average cases, O(nlogn) best-case

Space Complexity: O(n^2)

Code:

```
#include<bits/stdc++.h>
using namespace std;

struct Talk {
   char id;
   int start;
   int finish;
   int value;
};
bool isCompatible(Talk a, Talk b) {
   return (a.finish <= b.start || b.finish <=
a.start);
}

bool compareByValue(Talk a, Talk b) {
   return a.value > b.value;
}

int main() {

   vector<Talk> talks = {
      {'A', 1, 4, 10},
      {'B', 3, 5, 5},
      {'C', 0, 6, 15},
      {'D', 5, 7, 12},
      {'E', 3, 8, 8},
      {'F', 5, 9, 15},
      {'G', 8, 11, 7}
   };

   sort(talks.begin(), talks.end(),
   compareByValue);

   vector<Talk> selected;

  for (auto &talk : talks) {
      bool canSelect = true;
      for (auto &sel : selected) {
         if (!isCompatible(talk, sel)) {
            canSelect = false;
            break;
         }

      }
      if (canSelect) {
         selected.push_back(talk);
      }
   }


   cout << "(ID, Start, Finish, Value):\n";
   int totalValue = 0;
   for (auto &talk : selected) {

      cout << talk.id << "  " << talk.start <<
     "  " << talk.finish << "  " << talk.value
     << endl;
      totalValue += talk.value;

   }
   cout << "Total Priority: " << totalValue <<
    endl;

   return 0;
}
```

1. Output

```
(ID,    Start,   Finish, Value):
C        0        6        15
G        8        11       7
Total Priority:  22
```

Problem- 2.1.(a) : Use Kruskal Algorithm to determine the set of edges that form the Minimum Spanning Tree (MST). What is the total minimum cost for the network construction?

Algorithm:

1. List all edges of the graph along with their weights.
2. Sort all edges in non-decreasing order of weight.
3. Initialize each vertex as an independent set (using Union-Find).
4. Pick the smallest edge.
   - If it connects two different sets (no cycle), include it in the MST.
   - Otherwise, discard it.
5. Repeat Step 4 until (V – 1) edges are included in the MST.
6. The selected edges form the Minimum Spanning Tree with minimum total cost.

Time Complexity: O(E log E) — sorting dominates the process.

Space Complexity: O(V + E) — for storing parent array and edge list.

Code:

```
#include <bits/stdc++.h>
using namespace std;

struct Edge {
   char city1, city2;
   int weight;
};

int findParent(vector<int>& parent, int i) {
   if (parent[i] == -1)
      return i;
   return parent[i] = findParent(parent,
parent[i]);
}

void unionSets(vector<int>& parent, int u, int
v) {
   int parentU = findParent(parent, u);
   int parentV = findParent(parent, v);
   if (parentU != parentV)
      parent[parentU] = parentV;
}

void kruskalMST(const vector<Edge>&
edges, int numCities) {

   vector<Edge> sortedEdges = edges;
   sort(sortedEdges.begin(),
sortedEdges.end(),
       [](const Edge& a, const Edge& b) {
return a.weight < b.weight; });

   vector<int> parent(numCities, -1);
   int totalCost = 0;

   cout << "\nEdges in Minimum Spanning
Tree (Kruskal's):\n";

   for (auto& edge : sortedEdges) {
      int u = edge.city1 - 'A';
      int v = edge.city2 - 'A';

      if (findParent(parent, u) !=
findParent(parent, v)) {
         cout << " " << edge.city1 << " - " <<
edge.city2
             << " : " << edge.weight << endl;
         totalCost += edge.weight;
         unionSets(parent, u, v);
      }
   }

   cout << "\nTotal Minimum Construction
Cost: "
        << totalCost << " BDT Lacs\n";
}

int main() {

   const int numCities = 5;

   vector<Edge> edges = {
      {'A', 'B', 10},
      {'A', 'C', 40},
      {'B', 'C', 25},
      {'B', 'D', 50},
      {'C', 'D', 15},
      {'C', 'E', 30},
      {'D', 'E', 20}
   };

   cout << "Minimum Spanning Tree using
Kruskal's Algorithm\n";
   kruskalMST(edges, numCities);

   return 0;
}
```

2.1.(a) Output:

```
Minimum Spanning Tree using Kruskal's Algorithm
Edges in Minimum Spanning Tree (Kruskal's):
 A - B : 10
 C - D : 15
 D - E : 20
 B - C : 25
Total Minimum Construction Cost: 70 BDT Lacs
```

Problem - 2.1.(b): b. Starting from City A, use Prim's Algorithm to confirm the set of edges and the total minimum cost found in (a).

Prim's Algorithm:

1. Start with any vertex as the initial node.
2. Mark it as visited.
3. Find the smallest weighted edge that connects a visited vertex to an unvisited vertex.
4. Include this edge in the Minimum Spanning Tree (MST) and mark the new vertex as visited.
5. Repeat Step 3 and 4 until all vertices are included in the MST.
6. The set of selected edges forms the MST with the minimum total cost.

Time Complexity:
$O(V^2)$ — because we check all edges between visited and unvisited vertices.

Space Complexity:
$O(V^2)$ — for storing the adjacency matrix.

Code:

```
#include <bits/stdc++.h>
using namespace std;

const int N = 5;
int getIndex(char city) {
    return city - 'A';
}

void addEdge(int adj[N][N], char u, char v,
int w) {
    int i = getIndex(u);
    int j = getIndex(v);
    adj[i][j] = w;
    adj[j][i] = w;
}

int main() {
    int adj[N][N] = {0};
    addEdge(adj, 'A', 'B', 10);
    addEdge(adj, 'A', 'C', 40);
    addEdge(adj, 'B', 'C', 25);
    addEdge(adj, 'B', 'D', 50);
    addEdge(adj, 'C', 'D', 15);
    addEdge(adj, 'C', 'E', 30);
    addEdge(adj, 'D', 'E', 20);

    bool visited[N] = {false};
    visited[0] = true;
    int totalCost = 0;

    cout << "Minimum Spanning Tree using
    Prim's Algorithm:\n";
    cout << "\nEdges included:\n";
    for (int k = 0; k < N - 1; k++) {
        int minDist = 1e9;
        int a = -1, b = -1;

        for (int i = 0; i < N; i++) {
            if (visited[i]) {
                for (int j = 0; j < N; j++) {
                    if (!visited[j] && adj[i][j] &&
adj[i][j] < minDist) {
                        minDist = adj[i][j];
                        a = i;
                        b = j;
                    }
                }
            }
        }

        visited[b] = true;

        cout << " " << char('A' + a) << " - " <<
char('A' + b)
             << " : " << minDist << endl;

        totalCost += minDist;
    }

    cout << "\nTotal Minimum Construction
Cost: " << totalCost << " BDT Lacs\n";
    return 0;
}
```

2.1.(b) Output:

```
Minimum Spanning Tree using Prim's Algorithm:
Edges included:
 A - B : 10
 B - C : 25
 C - D : 15
 D - E : 20
Total Minimum Construction Cost: 70 BDT Lacs
```

Problem – 2.2: Use Dijkstra's Algorithm to find the shortest (least expensive) path from City A to City E.

● What is the sequence of cities in this shortest path?

● What is the total minimum cost for this path?

Dijkstra's Algorithm:

1. Assign a tentative distance value to every vertex:
    - 0 for the starting vertex.
    - ∞ (infinity) for all other vertices.
2. Mark all vertices as unvisited.
3. Select the unvisited vertex with the smallest tentative distance.
4. Update the distance values of its adjacent vertices:
    - If the current path offers a smaller distance, update it.
5. Mark the selected vertex as visited (a shortest path to it is now finalized).
6. Repeat Steps 3–5 until all vertices are visited or the shortest path to the destination is found.

Time Complexity:
$O(V^2)$ — when implemented using an adjacency matrix.

Space Complexity:
$O(V^2)$ — for storing the graph and distance arrays.

Code:

```
#include <bits/stdc++.h>
using namespace std;
const int N = 5;
const int INF = 1e9;
int getIndex(char city) {
   return city - 'A';
}
void addEdge(vector<vector<int>>& graph,
char u, char v, int w) {
   int i = getIndex(u);
   int j = getIndex(v);
   graph[i][j] = w;
   graph[j][i] = w;
}
int minDistance(vector<int>& dist,
vector<bool>& visited) {
   int minVal = INF, minIndex = -1;
   for (int i = 0; i < N; i++) {
      if (!visited[i] && dist[i] < minVal) {
         minVal = dist[i];
         minIndex = i;
      }
   }
   return minIndex;
}
void dijkstra(vector<vector<int>>& graph,
char startCity, char endCity) {
   vector<int> dist(N, INF);
   vector<bool> visited(N, false);
   vector<int> parent(N, -1);
   int start = getIndex(startCity);
   int end = getIndex(endCity);
   dist[start] = 0;
   for (int count = 0; count < N - 1; count++){
      int u = minDistance(dist, visited);
      visited[u] = true;
      for (int v = 0; v < N; v++) {
        if (!visited[v] && graph[u][v] &&
           dist[u] + graph[u][v] < dist[v]) {
             dist[v] = dist[u] + graph[u][v];
             parent[v] = u;
        }
      }
   }
   vector<char> path;
   int current = end;

   while (current != -1) {
      path.push_back('A' + current);
      current = parent[current];
   }
   reverse(path.begin(), path.end());

   cout << "\nShortest Path from " <<
startCity << " to " << endCity << ": ";
   for (int i = 0; i < path.size(); i++) {
      cout << path[i];
      if (i != path.size() - 1)
         cout << " -> ";
   }
   cout << "\nTotal Minimum Cost: " <<
dist[end] << " BDT Lacs\n";
}
int main() {
   vector<vector<int>> graph(N,
vector<int>(N, INF));
   addEdge(graph, 'A', 'B', 10);
   addEdge(graph, 'A', 'C', 40);
   addEdge(graph, 'B', 'C', 25);
   addEdge(graph, 'B', 'D', 50);
   addEdge(graph, 'C', 'D', 15);
   addEdge(graph, 'C', 'E', 30);
   addEdge(graph, 'D', 'E', 20);

   dijkstra(graph, 'A', 'E');

   return 0;
}
```

2.2 Output:

Shortest Path from A to E: A -> B -> C -> E

Total Minimum Cost: 65 BDT Lacs

Problem – 3.1 : Resource Acquisition (0/1 Knapsack with a Greedy Strategy)

Greedy Knapsack Algorithm:

1. Sort items in descending order of impact-to-cost ratio (impact / cost).
2. Initialize:
   - totalCost = 0
   - totalImpact = 0
3. Iterate through items in sorted order:
   - If adding the current item does not exceed the budget:
     - Add its cost to totalCost.
     - Add its impact to totalImpact.
   - Otherwise, skip the item.
4. Output: totalCost and totalImpact.

Time Complexity: O(n log n) — due to sorting the items.

Space Complexity: O(n) — for storing the list of items and temporary variables.

Code:

```
#include <bits/stdc++.h>
using namespace std;
struct Item {
    string id;
    int cost, impact;
};
void greedyKnapsack(vector<Item> items, int budget) {
    sort(items.begin(), items.end(), [](Item &a, Item &b) {
        return (double)a.impact / a.cost > (double)b.impact / b.cost;
    });
    int totalCost = 0, totalImpact = 0;
    cout << "\nGreedy Approach:\n";
    for (auto &it : items) {
        if (totalCost + it.cost <= budget) {
            totalCost += it.cost;
            totalImpact += it.impact;
        }
    }
    cout << "Total Cost: " << totalCost << " BDT\n";
    cout << "Total Impact: " << totalImpact << "\n";
}
int main() {
    int budget = 500000;
    vector<Item> items = {
        {"P1", 300000, 40},
        {"P2", 100000, 25},
        {"P3", 350000, 50},
        {"P4", 100000, 18},
        {"P5", 50000, 12}};
    greedyKnapsack(items, budget);
}
```

Output:

```
Greedy Approach:
Total Cost: 250000 BDT
Total Impact: 55
```

Conclusion:

In this report, several important problems were solved using the Greedy Algorithm approach and related optimization techniques. The Activity Selection Problem demonstrated how greedy choice ensures optimal scheduling by selecting non-overlapping tasks based on earliest finish time. The Minimum Construction Cost Problem, solved using both Prim's and Kruskal's algorithms, showed how spanning tree methods can efficiently connect all points in a network with minimal total cost. The Fastest Route Problem, implemented through Dijkstra's Algorithm, illustrated how shortest paths can be determined in weighted graphs using iterative distance updates. Finally, the 0/1 Knapsack Problem applied a greedy strategy based on the impact-to-cost ratio to achieve a near-optimal selection within budget constraints.

Through these implementations, it becomes clear that greedy and graph-based algorithms play a crucial role in achieving efficient and practical solutions to optimization problems. They not only reduce computational effort but also provide systematic methods for making locally optimal choices that often lead to globally effective outcomes.