Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA: 3.18)

Department of Information Technology

COURSE CODE: DJS23ILPC402

DATE: 21-02-25

COURSE NAME: Design and Analysis of Algorithms Lab

CLASS: S.Y. B.Tech

Experiment No. 4

CO/LO: Solve the problem using appropriate algorithmic design techniques.

Aim: Greedy algorithms for knapsack and coin changing

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

Department of Information Technology

COURSE CODE: DJS23ILPC402 **DATE**: 23/02/25

COURSE NAME: Design and Analysis of Algorithms Lab **CLASS**: SY-I2

NAME: Kiah Monpara **SAP ID**: 60003230249

Experiment No. 4

CO/LO:

Aim: Greedy Algorithms for Knapsack and Coin Change

Theory:

The greedy algorithm is an approach used for optimization problems where the best possible choice is made at each step without considering future consequences. This approach is used in problems where a local optimal solution leads to a globally optimal solution.

Steps:-

1) Selection- Best available option at the current step
2) Feasibility- Ensure the selected choice do not violate the problem's constraints.
3) Solution Check- If the selected choice leads to optimal or near-optimal solution, finalize it.

Common problem using greedy algorithm:-

1) Graph Algorithm - Prim's Algorithm, Kruskal's Algorithm, Dijkstra's Algorithm
2) Optimization Problems - Fractional Knapsack Problem
3) Currency Problems - Coin Change Problem
4) Scheduling Problems - Huffman Coding, Job Scheduling.

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

Department of Information Technology

Coin Change Problem

```
function coinChange (coins[], n,
                     amount){
    coins.sort ((a,b)=>b-a);
    let count=0;
    for (let coin of coins){
        while (amount >= coin){
            amount -= coin;
            count++;
        }
    }
    return amount==0 ?
           count : -1;
}
// Example usage
console.log (coinChange (
    [1,5,10,25], 63));
```

Knapsack Problem

```
function fractionalKnapsack (
    items, capacity){
    items.sort ((a,b) =>
        (b.value / b.weight)
        - (a.value / a.weight));
    let totalValue = 0;
    for (let item of items){
        if (item.weight <= capacity)
        { capacity -= item.weight;
          totalValue += item.value;
        } else {
        let fraction = capacity /
                       item.weight;
        totalValue += item.value *
                      fraction;

        break;
        }
    }
    return totalValue;
}
```

1

Conclusion:

The greedy algorithm makes locally optimal choices at each step, aiming for a globally optimal solution. It works well for problems like fractional knapsack but may fail in cases like 0/1 knapsack, while Dynamic programming is more effective

Theory:

1. **Greedy algorithm for knapsack coin changing**

Code:
```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
   int value;
   int weight;
   double ratio;
} Item;

int compare(const void *a, const void *b) {
   Item *item1 = (Item *)a;
   Item *item2 = (Item *)b;

   if (item1->ratio < item2->ratio) return 1;
   if (item1->ratio > item2->ratio) return -1;
   return 0;
}

double greedy_knapsack(Item items[], int n, int capacity) {
   for (int i = 0; i < n; i++) {
      items[i].ratio = (double)items[i].value / items[i].weight;
   }
   qsort(items, n, sizeof(Item), compare);

   double total_value = 0.0;
   for (int i = 0; i < n; i++) {
      if (capacity == 0) break;

      if (items[i].weight <= capacity) {
         total_value += items[i].value;
         capacity -= items[i].weight;
      } else {
         total_value += items[i].value * ((double)capacity / items[i].weight);
         break;
      }
   }
```

```
    return total_value;
}

int main() {
    int n = 3;
    Item items[] = {
        {60, 10, 0},
        {100, 20, 0},
        {120, 30, 0}
    };
    int capacity = 50;

    double result = greedy_knapsack(items, n, capacity);
    printf("Maximum value in knapsack: %.2f\n", result);
    return 0;
}
```

Output:

```
Maximum value in knapsack: 240.00
```

2. **Greedy algorithm for coin changing**

Code:

```
#include <stdio.h>

int greedy_coin_change(int coins[], int n, int amount) {
    int coin_count = 0;
    for (int i = 0; i < n; i++) {
        if (amount >= coins[i]) {
            coin_count += amount / coins[i];
            amount = amount % coins[i];
        }
    }

    if (amount > 0) {
        return -1;
    }

    return coin_count;
```

```
}

int main() {
    int coins[] = {25, 10, 5, 1};
    int n = sizeof(coins) / sizeof(coins[0]);
    int amount = 63;

    int result = greedy_coin_change(coins, n, amount);
    if (result == -1) {
        printf("Solution not possible with the given coin denominations.\n");
    } else {
        printf("Minimum coins required: %d\n", result);
    }

    return 0;
}
```

Output:

```
Minimum coins required: 6
```

Conclusion:

The greedy algorithms for knapsack and coin changing efficiently solve optimization problems by making locally optimal choices. These approaches are simple and effective but may not always guarantee the best solution for all cases

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA: 3.18)

Department of Information Technology

COURSE CODE: DJS23ILPC402 **DATE: 03-03-2025**

COURSE NAME: Design and Analysis of Algorithms Laboratory **CLASS:** S.Y. B.Tech

EXPERIMENT NO. 5

CO/LO: CO1- Analyse the performance of Algorithms asymptotically

AIM / OBJECTIVE: Implementation of Kruskal algorithm for finding Minimum spanning tree

Department of Information Technology

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

Department of Information Technology

COURSE CODE: DJS23ILPC402 **DATE:**

COURSE NAME: Design and Analysis of Algorithms Lab **CLASS:** SY-I2

NAME: Kiah Monpara **SAP ID:** 60003230249

Experiment No. 5

CO/LO:

Aim: Implement Kruskal's Algorithm for MST

Theory:

→ Kruskal's algorithm is a greedy algorithm used to find the Minimum Spanning Tree (MST) of a connected, weighted and and undirected graph. It selects edges in increasing order of weight, ensuring that no cycles are formed, until all vertices are connected.

→ Minimum Spanning Tree
It is a subgraph that connects all the vertices with the minimum possible total edge weight, without forming cycles.

Steps -
1) Sort all edges in non-decreasing order of their weight
2) Pick smallest edge and check if adding it forms a cycle.
3) If not formed, include the edge in MST
4) Repeat until we have (V-1) edges (V = no of vertices).

Department of Information Technology

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA: 3.18)
Department of Information Technology

→ ~~Function (ex)~~

```
~~public static void main (String[] args){~~
    ~~int V= 4; // vertices~~
    int[][] edges = {
        {0,1,10},{0,2,6},{0,3,5},{1,3,15},{2,3,4}
    };
```

→ Time complexity

Sorting the edges takes $O(E \log E)$ where E is the number of edges.

The union-find operations take almost constant time, $O(\alpha(V))$ where α is inverse Ackermann function which grows very slowly.

Thus overall time complexity of Kruskal's Algorithm is $O(E \log E)$ where E is number edges.

Conclusion:

Kruskal's Algorithm is an efficient method for finding the Minimum Spanning Tree using a greedy approach. It is best suited for sparse graphs and gaurantees optimal MST.

Code:

```
#include <stdio.h>
#include <stdlib.h>

struct Edge {
   int src, dest, weight;
};

// Comparator function for qsort to sort edges by increasing weight
int compareEdges(const void *a, const void *b) {     struct Edge
*edgeA = (struct Edge *)a;     struct Edge *edgeB = (struct Edge *)b;
   return edgeA->weight - edgeB->weight;
}

// DFS check for path
int dfs(int current, int target, struct Edge mstEdges[], int mstCount, int visited[], int vertices) {

   if (current == target)
return 1;
   visited[current] = 1;

   for (int i = 0; i < mstCount; i++) {

      int neighbor = -1;
      // Since the MST is undirected, check both endpoints
      if (mstEdges[i].src == current)
neighbor = mstEdges[i].dest;        else if
(mstEdges[i].dest == current)
neighbor = mstEdges[i].src;        if
(neighbor != -1 && !visited[neighbor]) {
         if (dfs(neighbor, target, mstEdges, mstCount, visited, vertices))
return 1;
      }
   }
   return 0;
}

// Returns 1 if a path exists between src and dest in the current MST, else 0 int
hasPath(int src, int dest, struct Edge mstEdges[], int mstCount, int vertices) {
```

```
    int *visited = (int *)calloc(vertices, sizeof(int));
    int result = dfs(src, dest, mstEdges, mstCount, visited, vertices);
free(visited);
    return result;
}

int main() {

    int vertices, edges;
    printf("Enter number of vertices and edges: ");
scanf("%d %d", &vertices, &edges);

    // Allocate memory for all edges
    struct Edge* edgeList = (struct Edge*)malloc(edges * sizeof(struct Edge));

    printf("Enter each edge details (1-indexed):\n");
    for (int i = 0; i < edges; i++) {

        printf("\nEnter edge %d details:\n", i + 1);
        printf("Source: ");
scanf("%d", &edgeList[i].src);
printf("Destination: ");
scanf("%d", &edgeList[i].dest);
printf("Weight: ");
        scanf("%d", &edgeList[i].weight);
        // Adjust for 0-indexed internal representation
        edgeList[i].src -= 1;
        edgeList[i].dest -= 1;
    }

    // Sort edges by weight
    qsort(edgeList, edges, sizeof(struct Edge), compareEdges);

    // Array to store edges included in the MST
    struct Edge *mstEdges = (struct Edge *)malloc((vertices - 1) * sizeof(struct Edge));
int mstCount = 0;
    int mstCost = 0;

    // Process sorted edges
    for (int i = 0; i < edges && mstCount < vertices - 1; i++) {
```

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA: 3.18)
Department of Information Technology

```
        int src = edgeList[i].src;
int dest = edgeList[i].dest;
        // if there is already a path, skip the edge.
        if (!hasPath(src, dest, mstEdges, mstCount, vertices)) {
mstEdges[mstCount] = edgeList[i];            mstCost +=
edgeList[i].weight;          mstCount++;
        }
    }


    printf("\nEdges in the Minimum Spanning Tree:\n");
    for (int i = 0; i < mstCount; i++) {
        printf("%d -- %d == %d\n", mstEdges[i].src + 1, mstEdges[i].dest + 1, mstEdges[i].weight);
    }
    printf("Total cost of MST: %d\n", mstCost);


    free(edgeList);
free(mstEdges);

    return 0;
}
```

Department of Information Technology

Output:

```
Enter number of vertices and edges: 5
8
Enter each edge details (1-indexed):

Enter edge 1 details:
Source: 1
Destination: 5
Weight: 2

Enter edge 2 details:
Source: 4
Destination: 2
Weight: 6

Enter edge 3 details:
Source: 1
Destination: 2
Weight: 5

Enter edge 4 details:
Source: 3
Destination: 2
Weight: 4

Enter edge 5 details:
Source: 1
Destination: 5
Weight: 2
```

```
Enter edge 6 details:
Source: 3
Destination: 2
Weight: 6

Enter edge 7 details:
Source: 1
Destination: 5
Weight: 3

Enter edge 8 details:
Source: 1
Destination: 5
Weight: 6

Edges in the Minimum Spanning Tree:
1 -- 5 == 2
3 -- 2 == 4
1 -- 2 == 5
4 -- 2 == 6
Total cost of MST: 17
```

CONCLUSION:

Kruskal's algorithm efficiently constructs the Minimum Spanning Tree (MST) by sorting edges and using the Union-Find data structure to ensure cycle prevention. This approach is particularly useful for sparse graphs as it considers edges in increasing order of weight and connects components greedily. The algorithm guarantees an optimal MST solution due to the greedy property and works efficiently with a time complexity of O(E log E), making it well-suited for large graphs with fewer edges.

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA: 3.18)

DEPARTMENT OF INFORMATION TECHNOLOGY

I095

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

Department of Information Technology

COURSE CODE: DJS23ILPC402 **DATE:** 24 - 02 - 25

COURSE NAME: Design and Analysis of Algorithms Lab **CLASS:** SY-I2

NAME: Manav Pathak **SAP ID:** 60003230269

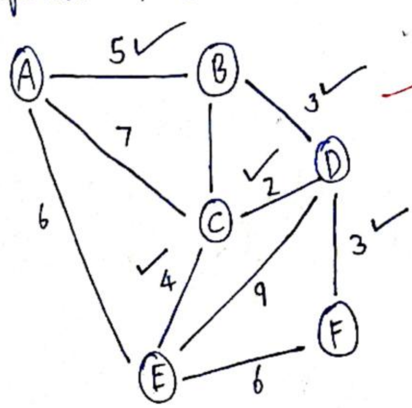
Experiment No. 5

CO/LO:

Aim: To implement MST using Kruskal's algorithm

Theory:

- A Minimum Spanning Tree (MST) of a unidirected and weighted graph is a subset of edges that connect all vertices without forming a cycle and minimum possible total edge weight
- Kruskal algorithm is a greedy approach to find MST



Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA: 3.18)

DEPARTMENT OF INFORMATION TECHNOLOGY

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)
Department of Information Technology

i) Sort all edges in graph in ascending order of weights

ii) Initialise MST as empty set

iii) Iterate : If adding edge does not form cycle, include in MST.
Else discard it

iv) Repeat untill MST has exactly V-1 edges

- For every edge (in increasing order of weight) we check for cycle using DFS
- Since MST is unidirected, both end points have to be checked.
- Neighbouring edge is assigned iteratively and if not present in visited array, DFS is called recursively for it again

Conclusion:

Kruskal's algorithm efficiently constructs an MST by choosing lowest weight edge that doesn't form cycle.

1

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA: 3.18)

DEPARTMENT OF INFORMATION TECHNOLOGY

COURSE CODE: DJS23ILPC402 **DATE: 03-03-2025**

COURSE NAME: Design and Analysis of Algorithms Laboratory **CLASS:** S.Y. B.Tech

EXPERIMENT NO. 5

CO/LO: CO1- Analyse the performance of Algorithms asymptotically

AIM / OBJECTIVE: Implementation of Kruskal algorithm for finding Minimum spanning tree

Code:

```
#include <stdio.h>
#include <stdlib.h>

struct Edge {
   int src, dest, weight;
};

// Comparator function for qsort to sort edges by increasing weight
int compareEdges(const void *a, const void *b) {
   struct Edge *edgeA = (struct Edge *)a;
   struct Edge *edgeB = (struct Edge *)b;
   return edgeA->weight - edgeB->weight;
}

// DFS check for path
int dfs(int current, int target, struct Edge mstEdges[], int mstCount, int visited[], int vertices) {

   if (current == target)
      return 1;
   visited[current] = 1;

   for (int i = 0; i < mstCount; i++) {

      int neighbor = -1;
      // Since the MST is undirected, check both endpoints
      if (mstEdges[i].src == current)
         neighbor = mstEdges[i].dest;
      else if (mstEdges[i].dest == current)
```

```
            neighbor = mstEdges[i].src;
        if (neighbor != -1 && !visited[neighbor]) {
            if (dfs(neighbor, target, mstEdges, mstCount, visited, vertices))
                return 1;
        }
    }
    return 0;
}

// Returns 1 if a path exists between src and dest in the current MST, else 0
int hasPath(int src, int dest, struct Edge mstEdges[], int mstCount, int vertices) {

    int *visited = (int *)calloc(vertices, sizeof(int));
    int result = dfs(src, dest, mstEdges, mstCount, visited, vertices);
    free(visited);
    return result;
}

int main() {

    int vertices, edges;
    printf("Enter number of vertices and edges: ");
    scanf("%d %d", &vertices, &edges);

    // Allocate memory for all edges
    struct Edge* edgeList = (struct Edge*)malloc(edges * sizeof(struct Edge));

    printf("Enter each edge details (1-indexed):\n");
    for (int i = 0; i < edges; i++) {

        printf("\nEnter edge %d details:\n", i + 1);
        printf("Source: ");
        scanf("%d", &edgeList[i].src);
        printf("Destination: ");
        scanf("%d", &edgeList[i].dest);
        printf("Weight: ");
        scanf("%d", &edgeList[i].weight);
        // Adjust for 0-indexed internal representation
        edgeList[i].src -= 1;
        edgeList[i].dest -= 1;
    }

    // Sort edges by weight
    qsort(edgeList, edges, sizeof(struct Edge), compareEdges);
```

DEPARTMENT OF INFORMATION TECHNOLOGY

```
    // Array to store edges included in the MST
    struct Edge *mstEdges = (struct Edge *)malloc((vertices - 1) * sizeof(struct Edge));
    int mstCount = 0;
    int mstCost = 0;

    // Process sorted edges
    for (int i = 0; i < edges && mstCount < vertices - 1; i++) {

        int src = edgeList[i].src;
        int dest = edgeList[i].dest;
        // if there is already a path, skip the edge.
        if (!hasPath(src, dest, mstEdges, mstCount, vertices)) {
            mstEdges[mstCount] = edgeList[i];
            mstCost += edgeList[i].weight;
            mstCount++;
        }
    }


    printf("\nEdges in the Minimum Spanning Tree:\n");
    for (int i = 0; i < mstCount; i++) {
        printf("%d -- %d == %d\n", mstEdges[i].src + 1, mstEdges[i].dest + 1, mstEdges[i].weight);
    }
    printf("Total cost of MST: %d\n", mstCost);


    free(edgeList);
    free(mstEdges);

    return 0;
}
```

DEPARTMENT OF INFORMATION TECHNOLOGY

Output:

```
D:\DSA\Dawg_DAA>cd "d:\DSA\Dawg_DAA\Exp 5\" &
Enter number of vertices and edges: 5
8
Enter each edge details (1-indexed):

Enter edge 1 details:
Source: 1
Destination: 5
Weight: 2

Enter edge 2 details:
Source: 4
Destination: 2
Weight: 6

Enter edge 3 details:
Source: 1
Destination: 2
Weight: 5

Enter edge 4 details:
Source: 3
Destination: 2
Weight: 4

Enter edge 5 details:
Source: 1
Destination: 5
Weight: 2
```

```
Enter edge 6 details:
Source: 3
Destination: 2
Weight: 6

Enter edge 7 details:
Source: 1
Destination: 5
Weight: 3

Enter edge 8 details:
Source: 1
Destination: 5
Weight: 6

Edges in the Minimum Spanning Tree:
1 -- 5 == 2
3 -- 2 == 4
1 -- 2 == 5
4 -- 2 == 6
Total cost of MST: 17
```

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA: 3.18)

DEPARTMENT OF INFORMATION TECHNOLOGY

CONCLUSION:

Kruskal's algorithm efficiently constructs the **Minimum Spanning Tree (MST)** by sorting edges and using the **Union-Find** data structure to ensure cycle prevention. This approach is particularly useful for **sparse graphs** as it considers edges in increasing order of weight and connects components greedily. The algorithm guarantees an optimal MST solution due to the **greedy property** and works efficiently with a time complexity of **O(E log E)**, making it well-suited for large graphs with fewer edges.