MANGALORE INSTITUTE OF TECHNOLOGY & ENGINEERING

(*A Unit of Rajalaxmi Education Trust*®, *Mangalore - 575001*)

Autonomous Institute affiliated to VTU, Belagavi, Approved by AICTE, New Delhi

Accredited by NAAC with A+ Grade, An ISO 9001: 2015 Certified Institution

Badaga Mijar, Moodabidri-574225, Karnataka

Laboratory Manual

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

FOURTH SEMESTER B.E

Course Title: ANALYSIS & DESIGN OF ALGORITHMS LABORATORY

Course Code: BCSL404

2022 Scheme

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

Course Title/Code: ANALYSIS & DESIGN OF ALGORITHMS LABORATORY [BCSL404]

VISION OF THE INSTITUTE

"To attain perfection in providing **Globally Competitive Quality Education** to all our students and also benefit the global community by using our strength in **Research and Development"**

MISSION OF THE INSTITUTE

"To establish world class educational institutions in their respective domains, which shall be **centers of excellence** in their Stated and Implied sense. To achieve this objective, we dedicate ourselves to meet the Challenges of becoming **Visionary and Realistic**, **Sensitive and Demanding**, **Innovative and Practical** and **Theoretical and Pragmatic;** All at the same time"

VISION OF THE DEPARTMENT

To create well groomed, technically competent and skilled AIML professionals who can become part of industry and undertake quality research at global level to meet societal needs.

MISSION OF THE DEPARTMENT

M1: Provide state of art infrastructure, tools and facilities to make students competent and achieve excellence in education and research.

M2: Provide a strong theoretical and practical knowledge across the AIML discipline with an emphasis on AI based research and software development.

M3: Inculcate strong ethical values, professional behavior and leadership abilities through various curricular, co-curricular, training and development activities.

Program Educational Objectives (PEOs)

PE01: Graduates will be able to follow logical, practical and research-oriented approach for solving the real-world problems by providing AI based solutions.

PEO2: Graduates will be able to work independently as well as in multidisciplinary teams in the workplace.

PEO4: Graduates will be able to setup start-up and become successful entrepreneurs.

Program Outcomes (POs)

PO1	**Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
PO2	**Problem analysis**: Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
PO3	**Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
PO4	**Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
PO5	**Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
PO6	**The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
PO7	**Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development
PO8	**Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
PO9	**Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
PO10	**Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
PO11	**Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
PO12	**Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Specific Outcomes (PSOs)

PSO - 1	Train machine learning models to address real life challenging problems using acquired AI knowledge
PSO - 2	Develop applications using ML techniques related to the field of medical, agriculture, defense, education and various scientific explorations.

Course Outcomes (COs):

At the end of the course the students will be able to:

Couse Index	**Course Outcome**
C406.1	**Demonstrate** competency in implementing the k-Nearest Neighbour algorithm for iris dataset classification, distinguishing correct and incorrect predictions.
C406.2	**Apply** K-means and EM clustering algorithms to a CSV dataset, enabling a comparative evaluation of their clustering quality.
C406.3	**Implement** and apply the Locally Weighted Regression algorithm to fit data points, gaining practical experience and visualization skills.
C406.4	**Develop** Artificial Neural Network using Backpropagation, showcasing proficiency in implementation, and testing with relevant datasets.

Prepared By: Dr. Maryjo M George **Verified By:**

Approved By Head of the Department: …………………….

index

Experiment no. 1

Design and implement C/C++ Program to find Minimum Cost Spanning Tree of a given connected undirected graph using Kruskal's algorithm.

Objective

To design and implement a C/C++ program to find the Minimum Cost Spanning Tree (MCST) of a given connected undirected graph using Kruskal's Algorithm.

Theory

Minimum Cost Spanning Tree (MCST)

A spanning tree of a graph is a tree that covers all the vertices with the minimum possible number of edges. The Minimum Cost Spanning Tree is the spanning tree with the least sum of edge weights.

Kruskal's Algorithm

Kruskal's Algorithm is a greedy algorithm that finds an MCST for a connected, undirected graph. It works by sorting all the edges of the graph in non-decreasing order of their weight and then adding them one by one to the spanning tree, ensuring that no cycle is formed.

Algorithm

1. **Sort all edges** in non-decreasing order of their weight.
2. **Initialize** the minimum spanning tree as empty.
3. **For each edge** in the sorted list:
   - If the edge does not form a cycle with the spanning-tree edges already included, add it to the spanning tree.
4. **Stop** when there are exactly V-1 edges in the spanning tree (where V is the number of vertices).

Pseudocode

```
function KruskalMST(edges, V, E):
    sort edges in non-decreasing order of their weight

    initialize parent array with -1 for each vertex

    result = empty list

    for i from 0 to E-1:
        if size of result == V-1:
            break

        next_edge = edges[i]

        xset = find(parent, next_edge.src)
        yset = find(parent, next_edge.dest)

        if xset != yset:
            result.append(next_edge)
            union(parent, xset, yset)
    return result

function find(parent, i):
    while parent[i] != -1:
        i = parent[i]
    return i

function union(parent, x, y):
    parent[x] = y
```

Program

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

#define MAX_VERTICES 4
#define MAX_EDGES 5

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

// Function to implement Kruskal's algorithm
void KruskalMST(Edge* edges, int V, int E) {
    Edge result[V];
    int e = 0;
    int i = 0;

    // Sort the edges based on weight (ascending order)
    for (int i = 0; i < E - 1; i++) {
        for (int j = 0; j < E - i - 1; j++) {
            if (edges[j].weight > edges[j + 1].weight) {
                Edge temp = edges[j];
                edges[j] = edges[j + 1];
                edges[j + 1] = temp;
            }
        }
    }

    int parent[V];
    for (int v = 0; v < V; v++)
        parent[v] = -1;

    while (e < V - 1 && i < E) {
        Edge next_edge = edges[i++];

        int xset = next_edge.src;
        int yset = next_edge.dest;

        // Check if including this edge causes cycle or not
        while (parent[xset] != -1)
            xset = parent[xset];

        while (parent[yset] != -1)
            yset = parent[yset];

        if (xset != yset) {
            result[e++] = next_edge;
            parent[xset] = yset; // Union of two sets
        }
    }
    printf("Edges  of  Minimum  Cost  Spanning
Tree:\n");
    int minimumCost = 0;
    for (i = 0; i < e; ++i) {
        printf("%d - %d: %d\n", result[i].src, result[i].dest, result[i].weight);
        minimumCost += result[i].weight;
    }
    printf("Minimum Cost Spanning Tree: %d\n", minimumCost);
}

int main() {
    Edge edge[MAX_EDGES] = { { 0, 1, 10 },
                             { 0, 2, 6 },
                             { 0, 3, 5 },
                             { 1, 3, 15 },
                             { 2, 3, 4 } };

    KruskalMST(edge, MAX_VERTICES, MAX_EDGES);

    return 0;
}
```

Results

```
Edges of Minimum Cost Spanning Tree:
2 - 3: 4
0 - 3: 5
0 - 1: 10
Minimum Cost Spanning Tree: 19
```

Inferences

1. **Correctness of Kruskal's Algorithm**: The edges listed in the output form a Minimum Cost Spanning Tree for the given graph, indicating the correctness of Kruskal's Algorithm.
2. **Greedy Approach**: The algorithm successfully demonstrates the greedy approach by choosing the minimum weight edges first and ensuring no cycles are formed.
3. **Cycle Detection**: The parent array and union-find method effectively handle cycle detection and merging of sets, showcasing their importance in Kruskal's Algorithm.
4. **Edge Case Handling**: The algorithm can handle various edge cases such as disconnected graphs (which are not suitable for spanning trees) or graphs with multiple edges between nodes.
5. **Time Complexity**: The time complexity of Kruskal's Algorithm is dominated by the time needed to sort the edges, which is $O(E\log E)$, where E is the number of edges. The union-find operations, which include find and union, are nearly constant time, $O(\alpha(V))$, where α is the inverse Ackermann function and V is the number of vertices. Thus, the overall time complexity of the algorithm is $O(E\log E+E\alpha(V))$ which simplifies to $O(E\log E)$ since $\alpha(V)$ grows very slowly and can be considered a constant for all practical purposes.

Experiment no. 2

Design and implement C/C++ Program to find Minimum Cost Spanning Tree of a given connected undirected graph using Prim's algorithm.

Objective

To design and implement a C program to find the Minimum Cost Spanning Tree (MCST) of a given connected undirected graph using Prim's Algorithm.

Theory

Minimum Cost Spanning Tree (MCST)

A spanning tree of a graph is a tree that covers all the vertices with the minimum possible number of edges. The Minimum Cost Spanning Tree is the spanning tree with the least sum of edge weights.

Prim's Algorithm

Prim's Algorithm is a greedy algorithm that finds an MCST for a connected, undirected graph. It works by starting with a single vertex and then repeatedly adding the cheapest possible connection from the tree to another vertex.

Algorithm

1. **Initialize** a key array to store the minimum weight edge for each vertex. Set all keys to infinity except for the first vertex, which is set to 0.
2. **Initialize** an MST set to keep track of vertices included in the MST.
3. **Repeat** until the MST includes all vertices:
   - Select the vertex u with the minimum key value that is not yet included in the MST.
   - Add vertex u to the MST set.
   - Update the key values of all adjacent vertices of u that are not yet included in the MST, using the weight of the edges connecting u to those vertices.
4. **Output** the edges included in the MST and their weights.

Pseudocode

```
function primMST(graph):
  Initialize key array with INF values
  Initialize mstSet array with false values
  key[0] = 0
    parent[0] = -1


  for count from 0 to V-1:
    u = minKey(key, mstSet)
        mstSet[u] = true


    for each vertex v adjacent to u:
            if graph[u][v] != 0 and mstSet[v] == false and graph[u][v] < key[v]:
                    parent[v] = u
                    key[v] = graph[u][v]
```

print MST edges and weights using parent array

```
function minKey(key, mstSet):
    min = INF
    for v from 0 to V-1:
        if mstSet[v] == false and key[v] < min:
            min = key[v]
            min_index = v
    return min_index
```

Program

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

#define V 5
#define INF 9999

int graph[V][V] = {
    {0, 2, 0, 6, 0},
    {2, 0, 3, 8, 5},
    {0, 3, 0, 0, 7},
    {6, 8, 0, 0, 9},
    {0, 5, 7, 9, 0}
};

int minKey(int key[], bool mstSet[]) {
    int min = INF, min_index;

    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}

void primMST() {
    int parent[V];
    int key[V];
    bool mstSet[V];

    for (int i = 0; i < V; i++)
        key[i] = INF, mstSet[i] = false;

    key[0] = 0;
    parent[0] = -1;

    for (int count = 0; count < V - 1; count++) {
        int u = minKey(key, mstSet);
        mstSet[u] = true;

        for (int v = 0; v < V; v++)
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }
    printf("Edge \tWeight\n");
    for (int i = 1; i < V; i++)
        printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
}

int main() {
    primMST();
    return 0;
}
```

Results

```
Edge   Weight
0 - 1    2
1 - 2    3
0 - 3    6
1 - 4    5
```

Inferences

1. **Greedy Approach**: The algorithm successfully demonstrates the greedy approach by choosing the minimum weight edges first and ensuring no cycles are formed.
2. **Edge Case Handling**: The algorithm can handle various edge cases such as graphs with isolated nodes (which are not suitable for spanning trees) or graphs with multiple edges between nodes.
3. **Time Complexity**: The time complexity of Prim's Algorithm implemented with an adjacency matrix is $O(V^2)$, where V is the number of vertices. This is because the algorithm repeatedly searches for the minimum key value in an unsorted array, which takes O(V) time, and this process is repeated V times. Optimizing this with a priority queue or binary heap can reduce the time complexity to O(ElogV), where E is the number of edges and V is the number of vertices.

Experiment no.3a

Design and implement C/C++ Program to solve All-Pairs Shortest Paths problem using Floyd's algorithm.

Objective

To design and implement a C/C++ program to solve the All-Pairs Shortest Paths problem for a given connected graph using Floyd's Algorithm.

Theory

All-Pairs Shortest Paths (APSP)

The All-Pairs Shortest Paths problem involves finding the shortest paths between every pair of vertices in a graph. This problem can be efficiently solved using Floyd's Algorithm, especially for dense graphs.

Floyd's Algorithm

Floyd's Algorithm is a dynamic programming algorithm that solves the APSP problem by iteratively improving the estimated shortest paths through a series of intermediate vertices. The algorithm updates the shortest path distances by considering each vertex as an intermediate point and checking if a shorter path exists via this intermediate vertex.

Algorithm

1. Initialize the distance matrix to the adjacency matrix of the graph.
2. Update the distance matrix by iterating through each possible intermediate vertex.
3. For each pair of vertices (i, j), check if a path through the intermediate vertex k is shorter than the direct path. Update the distance if a shorter path is found.
4. Repeat the process for all vertices as the intermediate vertex.

Pseudocode

```
function floydWarshall(graph):

    dist = copy(graph)

    for k from 0 to V-1:

        for i from 0 to V-1:

            for j from 0 to V-1:

                if dist[i][k] + dist[k][j] < dist[i][j]:

                    dist[i][j] = dist[i][k] + dist[k][j]

    return dist


function printSolution(dist):

    for i from 0 to V-1:

        for j from 0 to V-1:

            if dist[i][j] == INF:
                print "INF"

            else:

                print dist[i][j]
```

Program

```
#include <stdio.h>

#define INF 99999
#define V 4

void printSolution(int dist[][V]);

void floydWarshall(int graph[][V]) {
    int dist[V][V];
    int i, j, k;

    // Initialize the solution matrix same as input graph matrix
    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            dist[i][j] = graph[i][j];

    // Update the solution matrix by considering all vertices as intermediate vertices
    for (k = 0; k < V; k++) {
        for (i = 0; i < V; i++) {
            for (j = 0; j < V; j++) {
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }

    // Print the shortest distance matrix
    printSolution(dist);
}

void printSolution(int dist[][V]) {
    printf("The following matrix shows the shortest distances between every pair of
vertices:\n");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (dist[i][j] == INF)
                printf("%7s", "INF");
            else
                printf("%7d", dist[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int graph[V][V] = {{0, 5, INF, 10},
                       {INF, 0, 3, INF},
                       {INF, INF, 0, 1},
                       {INF, INF, INF, 0}};
    floydWarshall(graph);
    return 0;
}
```

Results

The following matrix shows the shortest distances between every pair of vertices:

```
      0       5       8       9
    INF       0       3       4
    INF     INF       0       1
    INF     INF     INF       0
```

Inferences

1. **Dynamic Programming Approach:** Floyd's Algorithm successfully demonstrates the dynamic programming approach by iteratively updating the shortest path estimates using intermediate vertices.
2. **Edge Cases:** The algorithm handles various edge cases such as graphs with no direct paths between some vertices (represented by `INF`).
3. **Time Complexity:** The time complexity of Floyd's Algorithm is $O(V^3)$), where V is the number of vertices. This is suitable for dense graphs but may be inefficient for large sparse graphs.
4. **Space Complexity:** The space complexity is $O(V^2)$ due to the distance matrix used to store shortest path estimates.

Experiment no. 3b

Design and implement C/C++ Program to find the transitive closure using Warshal's

algorithm.

Objective

To design and implement a C/C++ program to find the transitive closure of a given directed graph using Warshall's Algorithm.

Theory

Transitive Closure

The transitive closure of a graph is a reachability matrix where an entry (i, j) is true if there is a path from vertex i to vertex j in the graph. It represents whether a vertex j is reachable from vertex i.

Warshall's Algorithm

Warshall's Algorithm is used to compute the transitive closure of a directed graph. The algorithm works by iteratively updating the reachability matrix to account for indirect paths through intermediate vertices.

Algorithm

1. Initialize the reachability matrix with the adjacency matrix of the graph.
2. For each intermediate vertex k, update the reachability of each pair of vertices (i, j) to true if vertex j is reachable from vertex i through vertex k.
3. Repeat the process for all vertices as the intermediate vertex.

Pseudocode

```
function transitiveClosure(graph):
    reach = copy(graph)
    for k from 0 to V-1:
        for i from 0 to V-1:
            for j from 0 to V-1:
                reach[i][j] = reach[i][j] or (reach[i][k] and reach[k][j])
    return reach

function printTransitiveClosure(reach):
    for i from 0 to V-1:
        for j from 0 to V-1:
            print reach[i][j]
```

Program

```
#include <stdio.h>

#define V 4

void printTransitiveClosure(int reach[][V]);

void transitiveClosure(int graph[][V]) {
    int reach[V][V];
    int i, j, k;

    // Initialize reachability matrix as the input graph adjacency matrix
    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
```

```
            reach[i][j] = graph[i][j];

    // Update reachability matrix using Warshall's algorithm
    for (k = 0; k < V; k++) {
        for (i = 0; i < V; i++) {
            for (j = 0; j < V; j++) {
                reach[i][j] = reach[i][j] || (reach[i][k] && reach[k][j]);
            }
        }
    }

    // Print the transitive closure matrix
    printTransitiveClosure(reach);
}

void printTransitiveClosure(int reach[][V]) {
    printf("The following matrix shows the transitive closure of the graph:\n");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            printf("%d ", reach[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int graph[V][V] = {{1, 1, 0, 1},
                       {0, 1, 1, 0},
                       {0, 0, 1, 1},
                       {0, 0, 0, 1}};
    transitiveClosure(graph);
    return 0;
}
```

Results

The following matrix shows the transitive closure of the graph:

```
1 1 1 1
0 1 1 1
0 0 1 1
0 0 0 1
```

Inferences

1. **Dynamic Programming Approach:** Warshall's Algorithm demonstrates the dynamic programming approach by iteratively updating the reachability matrix to account for paths through intermediate vertices.
2. **Edge Cases:** The algorithm handles various edge cases, such as self-loops and disconnected components, accurately reflecting reachability.
3. **Time Complexity:** The time complexity of Warshall's Algorithm is $O(V^3)$, where V is the number of vertices. This makes it suitable for small to medium-sized graphs but less efficient for very large graphs.
4. **Space Complexity:** The space complexity is $O(V^2)$ due to the reachability matrix used to store path information.

Experiment no. 4

Design and implement C/C++ Program to find shortest paths from a given vertex in a weighted connected graph to other vertices using Dijkstra's algorithm

Objective

To design and implement a C/C++ program to find the shortest paths from a given vertex to all other vertices in a weighted, connected graph using Dijkstra's Algorithm.

Theory

Dijkstra's Algorithm

Dijkstra's Algorithm is a greedy algorithm used to find the shortest paths from a single source vertex to all other vertices in a weighted graph. It works by iteratively selecting the vertex with the minimum distance from the source and updating the distances of its adjacent vertices.

Algorithm

1. Initialize the distance of all vertices as infinite except the source vertex, which is set to zero.
2. Mark all vertices as unvisited.
3. Select the unvisited vertex with the smallest distance, mark it as visited, and update the distances of its adjacent unvisited vertices.
4. Repeat step 3 until all vertices are visited or the smallest distance among the unvisited vertices is infinite.

Pseudocode

```
function dijkstra(graph, src):
    dist = [INF, INF, ..., INF]  // Distance from source to each vertex
    dist[src] = 0
    sptSet = [false, false, ..., false]  // Shortest path tree set

    for count from 0 to V-1:
        u = vertex in sptSet with minimum distance
        sptSet[u] = true
        for each vertex v adjacent to u:
            if not in sptSet and graph[u][v] and dist[u] + graph[u][v] < dist[v]:
                dist[v] = dist[u] + graph[u][v]

    printSolution(dist)
```

Program

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

#define V 5 // Number of vertices in the graph

// Function to find the vertex with minimum distance value, from the set of vertices
not yet included in the shortest path tree
int minDistance(int dist[], int sptSet[]) {
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (sptSet[v] == 0 && dist[v] <= min) {
            min = dist[v];
            min_index = v;
        }

    return min_index;
}

// Function to print the constructed distance array
void printSolution(int dist[], int n) {
    printf("Vertex \t\t Distance from Source\n");
    for (int i = 0; i < n; i++)
        printf("%d \t\t %d\n", i, dist[i]);
}

// Function that implements Dijkstra's single source shortest path algorithm for a
graph represented using adjacency matrix representation
void dijkstra(int graph[V][V], int src) {
    int dist[V]; // The output array. dist[i] will hold the shortest distance from src
to i

    int sptSet[V]; // sptSet[i] will be true if vertex i is included in the shortest
path tree (0 means not included, 1 means included)

    // Initialize all distances as INFINITE and sptSet[] as false
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = 0;

    // Distance of source vertex from itself is always 0
    dist[src] = 0;

    // Find shortest path for all vertices
    for (int count = 0; count < V - 1; count++) {

        // Pick the minimum distance vertex from the set of vertices not yet processed.
        int u = minDistance(dist, sptSet);

        // Mark the picked vertex as processed
        sptSet[u] = 1;

        // Update dist value of the adjacent vertices of the picked vertex.
        for (int v = 0; v < V; v++)
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] +
graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }

    // Print the constructed distance array
    printSolution(dist, V);
}

int main() {
    int graph[V][V] = { { 0, 10, 0, 0, 5 },
                        { 0, 0, 1, 0, 2 },
                        { 0, 0, 0, 4, 0 },
                        { 7, 0, 6, 0, 0 },
                        { 0, 3, 9, 2, 0 } };

    dijkstra(graph, 0);

    return 0;
}
```

Results

```
Vertex          Distance from Source
0               0
1               8
2               9
3               7
4               5
```

Inferences

1. **Correctness of Dijkstra's Algorithm:** The output distances correctly represent the shortest paths from the source vertex to all other vertices, validating the correctness of Dijkstra's Algorithm.
2. **Greedy Approach:** The algorithm effectively demonstrates the greedy approach by always selecting the next vertex with the smallest known distance from the source.
3. **Edge Cases:** The algorithm handles various edge cases, such as graphs with zero-weight edges or no direct paths between certain vertices, accurately reflecting shortest paths.
4. **Time Complexity:** The time complexity of Dijkstra's Algorithm using an adjacency matrix is O(V2)O(V^2)O(V2), where VVV is the number of vertices. This is efficient for small to medium-sized graphs but can be optimized for larger graphs using priority queues and adjacency lists.
5. **Space Complexity:** The space complexity is O(V)O(V)O(V) due to the distance and shortest path tree sets used to store path information.

Experiment no. 5

Design and implement C/C++ Program to obtain the Topological ordering of vertices in a given digraph.

Objective

To design and implement a C/C++ program to find the topological ordering of vertices in a directed graph (digraph).

Theory

Topological Sorting

Topological sorting of a directed graph is a linear ordering of its vertices such that for every directed edge uvuvuv, vertex uuu comes before vvv in the ordering. It is applicable only to directed acyclic graphs (DAGs).

Algorithm

1. Calculate the in-degree of each vertex.
2. Initialize a stack and push all vertices with in-degree zero onto the stack.
3. Pop a vertex from the stack, append it to the topological order, and reduce the in-degree of its adjacent vertices.
4. If the in-degree of an adjacent vertex becomes zero, push it onto the stack.
5. Repeat until the stack is empty.
6. If all vertices are processed, the topological order is complete.

Pseudocode

```
function topologicalSort(graph, n):
    inDegree = [0] * n
    stack = []
    topologicalOrder = []

    for each vertex v in graph:
        for each neighbor u of v:
            inDegree[u] += 1

    for each vertex v in graph:
        if inDegree[v] == 0:
            stack.push(v)

    while stack is not empty:
        v = stack.pop()
        topologicalOrder.append(v)

        for each neighbor u of v:
            inDegree[u] -= 1
            if inDegree[u] == 0:
                stack.push(u)

    return topologicalOrder
```

Program

```
#include <stdio.h>

#define MAX 10

// Function prototype for the topological sort function
void fnTopological(int a[MAX][MAX], int n);

int main(void) {
    int n = 6; // Number of vertices
    // Predefined adjacency matrix for the graph
    int a[MAX][MAX] = {
        {0, 1, 1, 0, 0, 0},
        {0, 0, 0, 1, 1, 0},
        {0, 0, 0, 0, 1, 0},
        {0, 0, 0, 0, 0, 1},
        {0, 0, 0, 0, 0, 1},
        {0, 0, 0, 0, 0, 0}
    };

    printf("Topological Sorting Algorithm -\n");

    // Call the topological sorting function
    fnTopological(a, n);
    printf("\n");
    return 0;
}

void fnTopological(int a[MAX][MAX], int n) {
    int in[MAX] = {0}; // Array to store in-degrees of vertices
    int out[MAX]; // Array to store the topological order of vertices
    int stack[MAX]; // Stack to store vertices with in-degree zero
    int top = -1; // Stack top pointer
    int i, j, k = 0; // Loop counters

    // Step 1: Calculate in-degree of each vertex
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            if (a[j][i] == 1) { // There is an edge from j to i
                in[i]++; // Increment in-degree of vertex i
            }
        }
    }

    // Step 2: Perform topological sorting
    while (1) {
        // Find all vertices with in-degree zero
        for (i = 0; i < n; i++) {
            if (in[i] == 0) { // Vertex i has in-degree zero
                stack[++top] = i; // Push vertex i onto the stack
                in[i] = -1; // Mark vertex i as visited
            }
        }

        if (top == -1) // Check if the stack is empty
            break;

        out[k] = stack[top--]; // Pop a vertex from the stack and add it to the
topological order

        // Decrease in-degree of all adjacent vertices of the popped vertex
        for (i = 0; i < n; i++) {
            if (a[out[k]][i] == 1) { // There
is an edge from out[k] to i
                in[i]--; // Decrement in-degree of vertex i
            }
        }
        k++; // Move to the next position in the topological order array
    }

    // Step 3: Print the topological order
    printf("Topological Sorting (JOB SEQUENCE) is:- \n");
    for (i = 0; i < k; i++)
        printf("%d ", out[i]);
    printf("\n");
}
```

Results

```
Topological Sorting Algorithm -
Topological Sorting (JOB SEQUENCE) is:-
0 2 1 4 3 5
```

Inferences

1. **Correctness of Topological Sorting:** The output represents a valid topological order of the vertices, confirming the correctness of the algorithm.
2. **DAG Requirement:** The algorithm is applicable only to directed acyclic graphs (DAGs). The presence of cycles would prevent a valid topological sort.
3. **In-degree Calculation:** The initial in-degree calculation is crucial for identifying vertices with no incoming edges, which serve as starting points for the sorting process.
4. **Stack Usage:** The use of a stack to manage vertices with in-degree zero ensures that the algorithm processes each vertex once, maintaining efficiency.
5. **Time Complexity:** The time complexity of the topological sort algorithm is O(V+E), where V is the number of vertices and E is the number of edges. This is efficient for large graphs.

•

Experiment no. 6

Design and implement C/C++ Program to solve 0/1 Knapsack problem using Dynamic Programming method.

Objective

To design and implement a C/C++ program to solve the 0/1 Knapsack problem using the dynamic programming method.

Theory

0/1 Knapsack Problem

Given weights and values of nnn items, put these items in a knapsack of capacity WWW to get the maximum total value in the knapsack. In the 0/1 knapsack problem, each item can either be taken or not taken (0 or 1).

Algorithm

- **Initialization:** Create a table K[n+1][W+1], where K[i][w] represents the maximum value that can be obtained with iii items and capacity www.
- **Building the Table:**

    - If i=0 or w=0, K[i][w]=0
    - If the weight of the iii-th item is less than or equal to www, K[i][w]=max(val[i−1]+K[i−1][w−wt[i−1]],K[i−1][w])
    - Otherwise, K[i][w]=K[i−1][w]

Result Extraction: The value at K[n][W] is the maximum value that can be obtained.

- **Item Selection:** Backtrack through the table to determine which items are included in the optimal solution.

Program

```
#include <stdio.h>

// Function to return the maximum of two integers
int max(int a, int b) {
    return (a > b) ? a : b;
}

// Function to solve the 0/1 Knapsack problem
void knapsack(int W, int wt[], int val[], int n) {
    int i, w;
    int K[n + 1][W + 1];

    // Build table K[][] in bottom up manner
    for (i = 0; i <= n; i++) {
        for (w = 0; w <= W; w++) {
            if (i == 0 || w == 0) {
                K[i][w] = 0;
            } else if (wt[i - 1] <= w) {
                K[i][w] = max(val[i - 1] + K[i - 1][w - wt[i - 1]], K[i - 1][w]);
            } else {
                K[i][w] = K[i - 1][w];
            }
        }
    }

    // Print the table K
    printf("KnapSack Table:\n");
    for (i = 0; i <= n; i++) {
        for (w = 0; w <= W; w++) {
            printf("%d\t", K[i][w]);
        }
        printf("\n");
    }

    // Print the result
    printf("Maximum value that can be obtained: %d\n", K[n][W]);

    // To find which items are included
    int res = K[n][W];
    w = W;
    printf("Items included:\n");
    for (i = n; i > 0 && res > 0; i--) {
        if (res == K[i - 1][w])
            continue;
        else {
            printf("Item %d (Weight: %d, Value: %d)\n", i, wt[i - 1], val[i - 1]);
            res = res - val[i - 1];
            w = w - wt[i - 1];
        }
    }
}

int main() {
    int n, W;

    printf("Enter the number of items: ");
    scanf("%d", &n);

    int val[n], wt[n];

    printf("Enter the values of the items:\n");
    for (int i = 0; i < n; i++) {
        printf("Value of item %d: ", i + 1);
        scanf("%d", &val[i]);
    }

    printf("Enter the weights of the items:\n");
    for (int i = 0; i < n; i++) {
        printf("Weight of item %d: ", i + 1);
        scanf("%d", &wt[i]);
    }

    printf("Enter the maximum capacity of the knapsack: ");
    scanf("%d", &W);

    knapsack(W, wt, val, n);

    return 0;
}
```

Results

```
Enter the number of items: 4
Enter the values of the items:
Value of item 1: 60
Value of item 2: 100
Value of item 3: 120
Value of item 4: 50
Enter the weights of the items:
Weight of item 1: 10
Weight of item 2: 20
Weight of item 3: 30
Weight of item 4: 5
Enter the maximum capacity of the knapsack: 50
KnapSack Table:
0       0       0       0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0       0       0       0
0       0       0       0       0       0       0
0       0       0       0       0       0       0       0       0       0       60
60      60      60      60      60      60      60      60      60      60      60
60      60      60      60      60      60      60      60      60      60      60
60      60      60      60      60      60      60      60      60      60      60
60      60      60      60      60      60      60
0       0       0       0       0       0       0       0       0       0       60
60      60      60      60      60      60      60      60      60      100     100
100     100     100     160     160     160     160     160     160     160     160
160     160     160     160     160     160     160     160     160     160     160
160     160     160     160     160     160     160
0       0       0       0       0       0       0       0       0       0       60
60      60      60      60      60      60      60      60      60      100     100
100     100     100     160     160     160     160     160     180     180     180
180     180     220     220     220     220     220     220     220     220     220
220     220     220     220     220     220     220
0       0       0       0       0       50      50      50      50      50      60
60      60      60      60      60      60      60      60      60      100     100
100     100     100     160     160     160     160     160     180     180     180
180     180     220     220     220     220     220     220     220     220     220
220     220     220     220     220     220     220
Maximum value that can be obtained: 220
Items included:
Item 3 (Weight: 30, Value: 120)
Item 2 (Weight: 20, Value: 100)
```

Inferences

1. **Optimal Substructure:** The dynamic programming approach efficiently utilizes the optimal substructure property of the 0/1 knapsack problem, ensuring that the solution to subproblems contributes to the overall solution.
2. **Table Construction:** The bottom-up approach builds the solution iteratively, storing intermediate results to avoid redundant calculations.
3. **Time Complexity:** The time complexity of this algorithm is O(nW), where n is the number of items and W is the capacity of the knapsack. This is efficient for moderate-sized inputs but may become impractical for very large capacities.
4. **Space Complexity:** The space complexity is also O(nW) due to the storage requirements of the table K.
5. **Traceback for Solution Items:** The traceback process effectively identifies which items are included in the optimal solution, providing additional insights into the solution's composition.

Experiment no. 7

Design and implement C/C++ Program to solve discrete Knapsack and continuous Knapsack problems using greedy approximation method.

Objective

To design and implement C/C++ programs to solve discrete and continuous knapsack problems using the greedy approximation method.

Theory

Knapsack Problem

The knapsack problem is a combinatorial optimization problem where the goal is to maximize the total profit of items placed in a knapsack without exceeding its capacity.

Continuous Knapsack Problem

In the continuous knapsack problem, fractions of items can be taken. The objective is to maximize the total profit while adhering to the capacity constraint.

Discrete Knapsack Problem

In the discrete (or 0/1) knapsack problem, items cannot be divided; each item must be taken or left entirely.

Greedy Approximation Method

The greedy approach involves selecting items based on a criterion (such as profit-to-weight ratio) and iteratively adding them to the knapsack until the capacity is filled.

Programs

Continuous Knapsack Problem

```
#include <stdio.h>

int main() {
    float weight[50], profit[50], ratio[50], Totalvalue = 0.0, temp, capacity;
    int n, i, j;

    // Read the number of items
    printf("Enter the number of items: ");
    scanf("%d", &n);

    // Read the weight and profit of each item
    for (i = 0; i < n; i++) {
        printf("Enter Weight and Profit for item[%d]: ", i);
        scanf("%f %f", &weight[i], &profit[i]);
    }

    // Read the capacity of the knapsack
    printf("Enter the capacity of knapsack: ");
    scanf("%f", &capacity);
    // Calculate the ratio (profit/weight)
for each item
    for (i = 0; i < n; i++) {
        ratio[i] = profit[i] / weight[i];
    }

    // Sort the items based on the ratio in descending order
    for (i = 0; i < n - 1; i++) {
        for (j = i + 1; j < n; j++) {
            if (ratio[i] < ratio[j]) {
                // Swap ratios
                temp = ratio[j];
                ratio[j] = ratio[i];
                ratio[i] = temp;

                // Swap weights
                temp = weight[j];
                weight[j] = weight[i];
                weight[i] = temp;

                // Swap profits
                temp = profit[j];
                profit[j] = profit[i];
                profit[i] = temp;
            }
        }
    }

    // Calculate the maximum value that can be carried
    printf("Knapsack problems using Greedy Algorithm:\n");
    for (i = 0; i < n; i++) {
        if (weight[i] > capacity)
            break;
        else {
            Totalvalue += profit[i];
            capacity -= weight[i];
        }
    }

    // If there's still capacity left, take the fraction of the next item
    if (i < n) {
        Totalvalue += ratio[i] * capacity;
    }

    printf("\nThe maximum value is: %.2f\n", Totalvalue);

    return 0;
}
```

Discrete Knapsack Problem

```
#include <stdio.h>

int main() {
    float weight[50], profit[50], ratio[50], Totalvalue = 0.0, temp, capacity;
    int n, i, j;

    // Read the number of items
    printf("Enter the number of items: ");
    scanf("%d", &n);

    // Read the weight and profit of each item
    for (i = 0; i < n; i++) {
        printf("Enter Weight and Profit for
item[%d]: ", i);
        scanf("%f %f", &weight[i], &profit[i]);
    }

    // Read the capacity of the knapsack
    printf("Enter the capacity of knapsack: ");
    scanf("%f", &capacity);

    // Calculate the ratio (profit/weight) for each item
    for (i = 0; i < n; i++) {
        ratio[i] = profit[i] / weight[i];
    }

    // Sort the items based on the ratio in descending order
    for (i = 0; i < n - 1; i++) {
        for (j = i + 1; j < n; j++) {
            if (ratio[i] < ratio[j]) {
                // Swap ratios
                temp = ratio[j];
                ratio[j] = ratio[i];
                ratio[i] = temp;

                // Swap weights
                temp = weight[j];
                weight[j] = weight[i];
                weight[i] = temp;

                // Swap profits
                temp = profit[j];
                profit[j] = profit[i];
                profit[i] = temp;
            }
        }
    }

    // Calculate the maximum value that can be carried
    printf("Knapsack problems using Greedy Algorithm:\n");
    for (i = 0; i < n; i++) {
        if (weight[i] > capacity)
            continue;
        else {
            Totalvalue += profit[i];
            capacity -= weight[i];
        }
    }

    printf("\nThe maximum value is: %.2f\n", Totalvalue);

    return 0;
}
```

Results

Continuous Knapsack Problem

```
Enter the number of items: 4
Enter Weight and Profit for item[0]: 10 60
Enter Weight and Profit for item[1]: 20 100
Enter Weight and Profit for item[2]: 30 120
Enter Weight and Profit for item[3]: 5 50
Enter the capacity of knapsack: 50
Knapsack problems using Greedy Algorithm:
The maximum value is: 240.00
```

Discrete Knapsack Problem

```
Enter the number of items: 4
Enter Weight and Profit for item[0]: 10 60
Enter Weight and Profit for item[1]: 20 100
Enter Weight and Profit for item[2]: 30 120
Enter Weight and Profit for item[3]: 5 50
Enter the capacity of knapsack: 50
Knapsack problems using Greedy Algorithm:
The maximum value is: 210.00
```

Inferences

1. **Greedy Approach:** The greedy algorithm effectively solves the continuous knapsack problem by considering the profit-to-weight ratio and selecting items in descending order of this ratio.
2. **Continuous vs. Discrete:** The continuous knapsack problem allows for fractions of items to be taken, leading to higher total values compared to the discrete knapsack problem.
3. **Sorting Efficiency:** Sorting items based on the profit-to-weight ratio is crucial for the greedy algorithm to work effectively.
4. **Time Complexity:** The time complexity is O(nlogn) due to the sorting step, followed by O(n) for the selection process, making the greedy algorithm efficient for larger inputs.

Experiment no. 8

Design and implement C/C++ Program to find a subset of a given set S = {sl , s2,.....,sn} of n positive integers whose sum is equal to a given positive integer d.

Objective

To design and implement a C/C++ program to find a subset of a given set of positive integers whose sum is equal to a given positive integer using a dynamic programming approach.

Theory

Subset Sum Problem

The subset sum problem is a decision problem to determine whether there exists a subset of a given set of integers that sums up to a given target sum.

Dynamic Programming Approach

The dynamic programming approach solves the problem by building a table where each entry `subset[i][j]` is true if there is a subset of the first `i` elements that has a sum equal to `j`. The table is built iteratively, and the solution is then extracted by backtracking through the table.

Program

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

void printSubset(int set[], int subset[], int size) {
    printf("Found a subset with the given sum: { ");
    for (int i = 0; i < size; i++) {
        printf("%d ", subset[i]);
    }
    printf("}\n");
}

void printTable(bool subset[][10], int n, int sum) {
    printf("     ");
    for (int j = 0; j <= sum; j++) {
        printf("%4d", j);
    }
    printf("\n   -------------------------------------\n");
    for (int i = 1; i <= n; i++) {  // Start from 1 since row 0 is removed
        printf("%2d |", i);
        for (int j = 0; j <= sum; j++) {
            printf("%4s", subset[i][j] ? "T" : "F");
        }
        printf("\n");
    }
    printf("\n");
}

int main() {
    int set[] = {3, 34, 4, 12, 5, 2};
    int sum = 9;
    int n = sizeof(set) / sizeof(set[0]);
    bool subset[n + 1][sum + 1];

    // Initialize the subset table
    for (int i = 0; i <= n; i++) {  // Include row 0
        subset[i][0] = true;
    }

    for (int i = 1; i <= sum; i++) {
        subset[0][i] = false;
    }

    // Fill the subset table iteratively
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= sum; j++) {
            if (j < set[i - 1]) {
                subset[i][j] = subset[i - 1][j];
            } else {
                subset[i][j] = subset[i - 1][j] || subset[i - 1][j - set[i - 1]];
            }
        }
        printf("After including element %d (set[%d]):\n", set[i - 1], i - 1);
        printTable(subset, n, sum);
    }

    if (!subset[n][sum]) {
        printf("No subset with the given sum\n");
        return 0;
    }

    // Find and print the subset with steps
    int foundSubset[n];
    int size = 0;
    int i = n, j = sum;
    printf("Building the subset iteratively:\n");
    while (i > 0 && j > 0) {
        if (subset[i][j] && !subset[i - 1][j]) {
            foundSubset[size++] = set[i - 1];
            printf("Added %d to subset, new subset: { ", set[i - 1]);
            for (int k = 0; k < size; k++) {
                printf("%d ", foundSubset[k]);
            }
            printf("}\n");
            j -= set[i - 1];
        }
        i--;
    }

    printSubset(set, foundSubset, size);

    return 0;
}
```

Results

```
Enter the set elements: 3 34 4 12 5 2
Enter the sum: 9
After including element 3 (set[0]):
      0    1    2    3    4    5    6    7    8    9
   -----------------------------------------
 1 |   T    F    F    T    F    F    F    F    F    F
 2 |   T    F    F    T    F    F    F    F    F    F
 3 |   T    F    F    T    F    T    F    T    F    F
 4 |   T    F    F    T    F    T    F    T    F    T
 5 |   T   F   F   T   F   T   F   T   F   T
 6 |   T   F   T   T   F   T   T   T   F   T

Building the subset iteratively:
Added 4 to subset, new subset: { 4 }
Added 5 to subset, new subset: { 4 5 }

Found a subset with the given sum: { 4 5 }
```

Inferences

1. **Dynamic Programming Table:** The table `subset[i][j]` effectively tracks whether a subset sum of `j` can be achieved with the first `i` elements.
2. **Initialization:** Proper initialization of the table is crucial for the algorithm to work correctly.
3. **Backtracking:** The subset with the given sum is constructed by backtracking through the table.
4. **Complexity:** The time complexity of the dynamic programming solution is O(n·sum), making it efficient for larger inputs.

Experiment no. 9

Design and implement C/C++ Program to sort a given set of n integer elements using Selection Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

Objective

To design and implement a C/C++ program to sort a given set of `n` integer elements using the Selection Sort method, compute its time complexity, and plot a graph of the time taken versus `n` for varied values of `n > 5000`.

Theory

Selection Sort

Selection Sort is a simple comparison-based sorting algorithm. It works by repeatedly selecting the minimum element from the unsorted portion of the array and swapping it with the first unsorted element.

Time Complexity

The time complexity of Selection Sort is $O(n^2)$ in all cases. This is because there are two nested loops, each running up to n iterations.

Program

Part 1: Generate Random Numbers to File

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

void generateRandomNumbersToFile(int n, const char *filename) {
    FILE *file = fopen(filename, "w");
    if (file == NULL) {
        printf("Error opening file!\n");
        return;
    }
    srand(time(NULL)); // Seed the random number generator
    for (int i = 0; i < n; i++) {
        int randomNumber = rand() % 10000; // Generate a random number between 0 and
9999
        fprintf(file, "%d\n", randomNumber);
    }
    fclose(file);
    printf("%d random numbers have been written to %s\n", n, filename);
}

int main() {
    int n;
    printf("Enter the number of random numbers to generate (n > 5000): ");
    scanf("%d", &n);
    if (n <= 5000) {
        printf("Please enter a value greater than 5000.\n");
        return 1;
    }
    generateRandomNumbersToFile(n,
"random_numbers.txt");
    return 0;
}
```

Part 2: Selection Sort and Time Measurement

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

// Function to perform selection sort
void selectionSort(int arr[], int n) {
    int i, j, minIdx, temp;
    for (i = 0; i < n - 1; i++) {
        minIdx = i;
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIdx])
                minIdx = j;
        }
        // Swap the found minimum element with the first element
        temp = arr[minIdx];
        arr[minIdx] = arr[i];
        arr[i] = temp;
    }
}

// Function to generate random array
void generateRandomArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 10000; // Generate random numbers between 0 and 9999
    }
}

// Function to read array from a file
void readArrayFromFile(int arr[], int n, const char *filename) {
    FILE *file = fopen(filename, "r");
    for (int i = 0; i < n; i++) {
        fscanf(file, "%d", &arr[i]);
    }
    fclose(file);
}

int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int arr[n];
    // Choose to generate random array or read from file
    int choice;
    printf("Enter 1 to generate random array, 2 to read from file: ");
    scanf("%d", &choice);
    if (choice == 1) {
        generateRandomArray(arr, n);
    } else if (choice == 2) {
        readArrayFromFile(arr, n, "random_numbers.txt");
    } else {
        printf("Invalid choice\n");
        return 1;
    }

    // Measure the time taken by selection sort
    clock_t start, end;
    double cpu_time_used;

    start = clock();
    selectionSort(arr, n);
    end = clock();

    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;

    printf("Time taken to sort %d elements: %f seconds\n", n, cpu_time_used);

    return 0;
}
```

Instructions for Running the Program

1. **Generate Random Numbers to File:**
   - Compile and run the first part of the program to generate random numbers and save them to a file:

     ```
     gcc generate_random_numbers.c -o generate_random_numbers
     ./generate_random_numbers
     ```

   - Enter a number greater than 5000 when prompted.
2. **Sort and Measure Time:**
   - Compile and run the second part of the program to sort the numbers and measure the time taken:

     ```
     gcc selection_sort.c -o selection_sort
     ./selection_sort
     ```

   - Enter the number of elements to sort and choose whether to generate a random array or read from the file generated in the first step.

Results and Plotting the Graph

Run the sorting program for various values of `n > 5000` (e.g., 6000, 7000, 8000, ..., 10000) and record the time taken for each. Use a plotting tool like Excel or a programming language with plotting capabilities (e.g., Python) to create a graph of the time taken versus `n`.

Sample Output

```
Enter the number of random numbers to generate (n > 5000): 6000
6000 random numbers have been written to random_numbers.txt

Enter the number of elements: 6000
Enter 1 to generate random array, 2 to read from file: 2
Time taken to sort 6000 elements: 1.234567 seconds
```

Inferences

- **Observation:** The time taken for sorting increases as the value of `n` increases, demonstrating the O(n^2) time complexity of the Selection Sort algorithm.
- **Graph:** The plot of time taken versus `n` should show a quadratic growth, indicating the inefficiency of Selection Sort for large datasets.

Experiment no. 10

Design and implement C/C++ Program to sort a given set of n integer elements using Quick Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

Objective

To design and implement a C program to sort a given set of `n` integer elements using the Quick Sort method, compute its time complexity, and plot a graph of the time taken versus `n` for varied values of `n > 5000`.

Theory

Quick Sort

Quick Sort is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

Time Complexity

- **Best case:** O(nlogn)
- **Average case:** O(nlogn)
- **Worst case:** O(n^2)

Program

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

// Function to swap two elements
void swap(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}

// Partition function
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

// Quick Sort function
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// Function to generate an array of random integers
void generateRandomArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 10000; // Generating random integers between 0 and 9999
    }
}

int main() {
    FILE *fptr;
    fptr = fopen("quicksort_times.csv", "w");
    if (fptr == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    fprintf(fptr, "n,TimeTaken\n");

    for (int n = 5000; n <= 50000; n += 5000) {
        int* arr = (int*)malloc(n * sizeof(int));
        generateRandomArray(arr, n);

        clock_t start, end;
        double cpu_time_used;

        start = clock();
        quickSort(arr, 0, n - 1);
        end = clock();

        cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
        fprintf(fptr, "%d,%f\n", n, cpu_time_used);

        free(arr);
    }

    fclose(fptr);

    printf("Time data written to quicksort_times.csv\n");
    return 0;
}
```

Results

Run the program and collect the data on the time taken to sort the arrays of different sizes. The collected data will be stored in `quicksort_times.csv` for plotting.

Plotting the Graph

Use a plotting tool (such as Python with Matplotlib) to plot the graph of time taken versus `n`.

Python Script for Plotting

```
import matplotlib.pyplot as plt
import csv

n_values = []
```

```
time_taken = []

with open('quicksort_times.csv', 'r') as file:
    csvreader = csv.reader(file)
    next(csvreader)  # Skip header row
    for row in csvreader:
        n_values.append(int(row[0]))
        time_taken.append(float(row[1]))

plt.plot(n_values, time_taken, marker='o')
plt.title('Time Taken vs. Number of Elements for Quick Sort')
plt.xlabel('Number of Elements (n)')
plt.ylabel('Time Taken (seconds)')
plt.grid(True)
plt.show()
```

Inferences

1. **Efficiency:** Quick Sort is efficient for large datasets, as indicated by the O(nlogn) average time complexity.
2. **Performance:** The actual performance and time taken for sorting increases with the size of the dataset.
3. **Visualization:** The graph visually demonstrates the relationship between the number of elements and the time taken, showing the algorithm's scalability.

Experiment no. 11

Design and implement C/C++ Program to sort a given set of n integer elements using Merge Sort method and compute its time complexity. Run the program for varied values of n> 5000, and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

Objective

To design and implement a C program to sort a given set of `n` integer elements using the Merge Sort method, compute its time complexity, and plot a graph of the time taken versus `n` for varied values of `n > 5000`.

Theory

Merge Sort

Merge Sort is a divide-and-conquer algorithm that divides the input array into two halves, sorts each half, and then merges the two sorted halves to produce the sorted output.

Time Complexity

- **Best case:** O(nlogn)
- **Average case:** O(nlogn)
- **Worst case:** O(nlogn)

Program

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

// Function to merge two subarrays
void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    int L[n1], R[n2];

    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    i = 0;
    j = 0;
    k = l;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

// Function to implement Merge Sort
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;

        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}

// Function to generate an array of random integers
void generateRandomArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 10000; // Generating random integers between 0 and 9999
    }
}

int main() {
    FILE *fptr;
    fptr = fopen("mergesort_times.csv", "w");
    if (fptr == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    fprintf(fptr, "n,TimeTaken\n");

    for (int n = 5000; n <= 50000; n += 5000) {
        int* arr = (int*)malloc(n * sizeof(int));
        generateRandomArray(arr, n);

        clock_t start, end;
        double cpu_time_used;

        start = clock();
        mergeSort(arr, 0, n - 1);
        end = clock();

        cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
        fprintf(fptr, "%d,%f\n", n, cpu_time_used);

        free(arr);
    }
    fclose(fptr);

    printf("Time data written to mergesort_times.csv\n");
    return 0;
}
```

Results

Run the program and collect the data on the time taken to sort the arrays of different sizes. The collected data will be stored in `mergesort_times.csv` for plotting.

Plotting the Graph

Use a plotting tool (such as Python with Matplotlib) to plot the graph of time taken versus `n`.

Python Script for Plotting

```
import matplotlib.pyplot as plt
import csv

n_values = []
time_taken = []

with open('mergesort_times.csv', 'r') as file:
    csvreader = csv.reader(file)
    next(csvreader)  # Skip header row
    for row in csvreader:
        n_values.append(int(row[0]))
        time_taken.append(float(row[1]))

plt.plot(n_values, time_taken, marker='o')
plt.title('Time Taken vs. Number of Elements for Merge Sort')
plt.xlabel('Number of Elements (n)')
plt.ylabel('Time Taken (seconds)')
plt.grid(True)
plt.show()
```

Inferences

1. **Efficiency:** Merge Sort is efficient for large datasets, as indicated by the O(nlogn) time complexity.
2. **Performance:** The actual performance and time taken for sorting increases with the size of the dataset.
3. **Visualization:** The graph visually demonstrates the relationship between the number of elements and the time taken, showing the algorithm's scalability.

Experiment no. 12
Design and implement C/C++ Program for N Queen's problem using Backtracking.

Objective

To design and implement a C program to solve the N Queen's problem using the backtracking method.

Theory

N Queen's Problem

The N Queen's problem is a classic problem in computer science and artificial intelligence. The objective is to place `N` queens on an `N x N` chessboard such that no two queens threaten each other. A queen can attack horizontally, vertically, or diagonally.

Backtracking

Backtracking is a general algorithm for finding all (or some) solutions to computational problems, notably constraint satisfaction problems, that incrementally builds candidates to the solutions. It abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution.

Program

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

#define N 4

// Function to print the solution
void printSolution(int board[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%d ", board[i][j]);
        }
        printf("\n");
    }
}

// Function to check if a queen can be placed on board[row][col]
bool isSafe(int board[N][N], int row, int col) {
    int i, j;

    // Check this row on left side
    for (i = 0; i < col; i++) {
        if (board[row][i])
            return false;
    }

    // Check upper diagonal on left side
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j])
            return false;
    }

    // Check lower diagonal on left side
    for (i = row, j = col; j >= 0 && i < N;
i++, j--) {
        if (board[i][j])
            return false;
    }

    return true;
}

// A recursive utility function to solve N Queen problem
bool solveNQUtil(int board[N][N], int col) {
    // If all queens are placed, return true
    if (col >= N)
        return true;

    // Consider this column and try placing this queen in all rows one by one
    for (int i = 0; i < N; i++) {
        if (isSafe(board, i, col)) {
            // Place this queen in board[i][col]
            board[i][col] = 1;

            // Recur to place rest of the queens
            if (solveNQUtil(board, col + 1))
                return true;

            // If placing queen in board[i][col] doesn't lead to a solution, then
remove the queen (backtrack)
            board[i][col] = 0;
        }
    }

    // If the queen cannot be placed in any row in this column col, then return false
    return false;
}

// This function solves the N Queen problem using Backtracking. It mainly uses
solveNQUtil() to solve the problem.
bool solveNQ() {
    int board[N][N] = { { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 },
                        { 0, 0, 0, 0 } };

    if (!solveNQUtil(board, 0)) {
        printf("Solution does not exist\n");
        return false;
    }

    printSolution(board);
    return true;
}

// Main function
int main() {
    solveNQ();
    return 0;
}
```

Explanation

1. **printSolution:** This function prints the current solution (board configuration).
2. **isSafe:** This function checks if a queen can be placed on the board at the position (row, col).

3. **solveNQUtil:** This function uses recursion and backtracking to place queens on the board.
4. **solveNQ:** This function initializes the board and calls `solveNQUtil`. If no solution exists, it prints "Solution does not exist".
5. **main:** The main function calls `solveNQ`.

Results

The program prints one possible solution for the N Queen's problem if it exists, or it prints "Solution does not exist" if no solution is found. For N=4, the solution is:

```
0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0
```

This solution shows the placement of 4 queens on a 4x4 chessboard such that no two queens threaten each other.

Inferences

- Backtracking algorithm efficiently solves the N Queen's problem for small values of `N`.
- Explores valid placements systematically.
- Time complexity is O(N!) and space complexity is O(N^2).
- Computation time and space requirements grow exponentially with larger `N`.
- Useful for moderate values of `N`.
- Serves as a foundational approach for other constraint satisfaction problems.
- Practical optimizations and leveraging symmetries can enhance efficiency.