|  |  |
| --- | --- |
| **PRACTICAL COMPONENTS** | |
| **Sl. No** | **Experiments** |
| 1 | Write a Cprogram to sort the elements by using quick sort. |
| 2 | Write a C program to sort the elements using Merge Sort |
| 3 | From a given vertex in a weighted connected graph, Find the shortest path to other vertices using Dijkstra’s algorithm. |
| 4 | Implement 0/1 Knapsack problem using Dynamic Programming. |
| 5 | Find the Minimum cost Spanning Tree of a given undirected graph using Prim’s Algorithm. |
| 6 | Compute the transitive closure of a given directed graph using Warshall’s Algorithm, |
| 7 | Implement all pair shortest path problems using Floyd’s algorithm. |
| 8 | Print all nodes reachable from a given starting node in a digraph using the BFS Method. |
| 9 | Check whether a given graph is connected or not using the DFS Method. |
| 10 | Implement N Queen’s problem using Backtracking. |

**Write a C Program to sort the elements by using quick sort.**

#include <stdio.h>

// Function to swap two elements

void swap(int arr[], int i, int j) {

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

// Function to partition the array

int partition(int arr[], int low, int high) {

int pivot = arr[high]; // Pivot element

int i = low - 1; // Index of smaller element

for (int j = low; j < high; j++) {

// If current element is smaller than the pivot

if (arr[j] < pivot) {

i++; // Increment index of smaller element

swap(arr, i, j);

}

}

swap(arr, i + 1, high);

return i + 1;

}

// QuickSort function

void quickSort(int arr[], int low, int high) {

if (low < high) {

// Partitioning index

int pi = partition(arr, low, high);

// Recursively sort elements before and after partition

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

// Function to print an array

void printArray(int arr[], int size) {

for (int i = 0; i < size; i++)

printf("%d ", arr[i]);

printf("\n");

}

// Main function

int main() {

int arr[] = {10, 7, 8, 9, 1, 5};

int n = sizeof(arr) / sizeof(arr[0]);

printf("Original array: ");

printArray(arr, n);

quickSort(arr, 0, n - 1);

printf("Sorted array: ");

printArray(arr, n);

return 0;

}

**Outpu**t: Original array: 10 7 8 9 1 5

Sorted array: 1 5 7 8 9 10

**Write a C program to sort the elements using Merge Sort**

#include <stdio.h>

// Function to merge two subarrays

void merge(int arr[], int l, int m, int r) {

int n1 = m - l + 1;

int n2 = r - m;

int L[n1], R[n2];

// Copy data to temporary arrays L[] and R[]

for (int i = 0; i < n1; i++)

L[i] = arr[l + i];

for (int j = 0; j < n2; j++)

R[j] = arr[m + 1 + j];

int i = 0, j = 0, k = l;

// Merge the temporary arrays back into arr[l..r]

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

k++;

}

// Copy the remaining elements of L[], if any

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

// Copy the remaining elements of R[], if any

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

// Merge Sort function

void mergeSort(int arr[], int l, int r) {

if (l < r) {

int m = l + (r - l) / 2;

// Recursively sort the first and second halves

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

// Merge the sorted halves

merge(arr, l, m, r);

}

}

// Function to print an array

void printArray(int arr[], int size) {

for (int i = 0; i < size; i++)

printf("%d ", arr[i]);

printf("\n");

}

// Main function

int main() {

int arr[] = {12, 11, 13, 5, 6, 7};

int n = sizeof(arr) / sizeof(arr[0]);

printf("Original array: ");

printArray(arr, n);

mergeSort(arr, 0, n - 1);

printf("Sorted array: ");

printArray(arr, n);

return 0;

}

Output: Original array: 12 11 13 5 6 7

Sorted array: 5 6 7 11 12 13

**From a given vertex in a weighted connected graph, Find the shortest path to other vertices using Dijkstra’s algorithm.**

#include <stdio.h>

#include <limits.h>

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

// Function to find the vertex with the minimum distance value

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 Distance from Source\n");

for (int i = 0; i < n; i++)

printf("%d \t %d\n", i, dist[i]);

}

// Function that implements Dijkstra's single source shortest path algorithm

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 1 if vertex i is included in the shortest path tree or shortest distance from src to i is finalized

// 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++)

// Update dist[v] only if is not in sptSet, there is an edge from u to v, and total weight of path from src to v through u is smaller than current value of dist[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);

}

// Main function

int main() {

// Example graph represented as an adjacency matrix

int graph[V][V] = {

{0, 10, 0, 0, 5},

{10, 0, 1, 0, 2},

{0, 1, 0, 4, 0},

{0, 0, 4, 0, 3},

{5, 2, 0, 3, 0}

};

int src = 0; // Source vertex

dijkstra(graph, src);

return 0;

}

**Output**

Vertex Distance from Source

0 0

1 7

2 8

3 9

4 5

**Implement 0/1 Knapsack problem using Dynamic Programming.**

#include <stdio.h>

// Function to find the maximum of two integers

int max(int a, int b) {

return (a > b) ? a : b;

}

// Function to solve the 0/1 Knapsack problem using Dynamic Programming

int knapsack(int W, int wt[], int val[], int n) {

int K[n + 1][W + 1];

// Build table K[][] in bottom-up manner

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

for (int 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];

}

}

return K[n][W];

}

// Main function

int main() {

int val[] = {60, 100, 120};

int wt[] = {10, 20, 30};

int W = 50;

int n = sizeof(val) / sizeof(val[0]);

printf("Maximum value in Knapsack = %d\n", knapsack(W, wt, val, n));

return 0;

}

Output:

Maximum value in Knapsack = 220

**Find the Minimum cost Spanning Tree of a given undirected graph using Prim’s Algorithm**

#include <stdio.h>

#include <limits.h>

#include <stdbool.h>

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

// Function to find the vertex with the minimum key value

int minKey(int key[], bool mstSet[]) {

int min = INT\_MAX, 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;

}

// Function to print the constructed MST

void printMST(int parent[], int graph[V][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]]);

}

// Function to construct and print the MST using Prim's algorithm

void primMST(int graph[V][V]) {

int parent[V]; // Array to store constructed MST

int key[V]; // Key values used to pick minimum weight edge in cut

bool mstSet[V]; // To represent set of vertices not yet included in MST

// Initialize all keys as INFINITE and mstSet[] as false

for (int i = 0; i < V; i++)

key[i] = INT\_MAX, mstSet[i] = false;

// Always include first 1st vertex in MST.

key[0] = 0; // Make key 0 so that this vertex is picked as first vertex

parent[0] = -1; // First node is always root of MST

// The MST will have V vertices

for (int count = 0; count < V - 1; count++) {

// Pick the minimum key vertex from the set of vertices not yet included in MST

int u = minKey(key, mstSet);

// Add the picked vertex to the MST Set

mstSet[u] = true;

// Update key value and parent index of the adjacent vertices of the picked vertex.

// Consider only those vertices which are not yet included in MST

for (int v = 0; v < V; v++)

// graph[u][v] is non zero only for adjacent vertices of m

// mstSet[v] is false for vertices not yet included in MST

// Update the key only if graph[u][v] is smaller than key[v]

if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])

parent[v] = u, key[v] = graph[u][v];

}

// Print the constructed MST

printMST(parent, graph);

}

// Main function

int main() {

// Example graph represented as an adjacency matrix

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}

};

// Print the solution

primMST(graph);

return 0;

}

Output:

Edge Weight

0 - 1 2

1 - 2 3

0 - 3 6

1 - 4 5

**Compute the transitive closure of a given directed graph using Warshall’s Algorithm,**

**#**include <stdio.h>

#define V 4 // Number of vertices in the graph

// Function to print the transitive closure matrix

void printMatrix(int reach[V][V]) {

printf("Transitive closure matrix:\n");

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

for (int j = 0; j < V; j++) {

printf("%d ", reach[i][j]);

}

printf("\n");

}

}

// Function to compute transitive closure using Warshall's Algorithm

void warshall(int graph[V][V]) {

int reach[V][V];

// Initialize the reachability matrix as the input graph adjacency matrix

for (int i = 0; i < V; i++)

for (int j = 0; j < V; j++)

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

// Apply Warshall's algorithm

for (int k = 0; k < V; k++) {

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

for (int j = 0; j < V; j++) {

reach[i][j] = reach[i][j] || (reach[i][k] && reach[k][j]);

}

}

}

// Print the transitive closure matrix

printMatrix(reach);

}

// Main function

int main() {

// Example graph represented as an adjacency matrix

int graph[V][V] = {

{1, 1, 0, 1},

{0, 1, 1, 0},

{0, 0, 1, 1},

{0, 0, 0, 1}

};

// Compute and print the transitive closure

warshall(graph);

return 0;

}

Output:

Transitive closure matrix:

1 1 1 1

0 1 1 1

0 0 1 1

0 0 0 1

**Implement all pair shortest path problems using Floyd’s algorithm.**

#include <stdio.h>

#define V 4 // Number of vertices in the graph

#define INF 99999 // A large value to represent infinity

// Function to print the solution matrix

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

printf("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");

}

}

// Function to implement Floyd's algorithm

void floydWarshall(int graph[V][V]) {

int dist[V][V], 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

for (k = 0; k < V; k++) {

// Pick all vertices as source one by one

for (i = 0; i < V; i++) {

// Pick all vertices as destination for the above picked source

for (j = 0; j < V; j++) {

// If vertex k is on the shortest path from i to j, then update the value of dist[i][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);

}

// Main function

int main() {

// Example graph represented as an adjacency matrix

int graph[V][V] = {

{0, 3, INF, 7},

{8, 0, 2, INF},

{5, INF, 0, 1},

{2, INF, INF, 0}

};

// Print the solution

floydWarshall(graph);

return 0;

}

OutPut:

Shortest distances between every pair of vertices:

0 3 3 4

5 0 2 3

3 6 0 1

2 5 5 0

**Print all nodes reachable from a given starting node in a digraph using the BFS Method**.

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

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

// Function to add an edge to the adjacency matrix

void addEdge(int graph[V][V], int src, int dest) {

graph[src][dest] = 1;

}

// Function to perform BFS and print reachable nodes

void BFS(int graph[V][V], int start) {

bool visited[V] = {false}; // Array to keep track of visited nodes

int queue[V], front = 0, rear = 0; // Queue for BFS

// Mark the starting node as visited and enqueue it

visited[start] = true;

queue[rear++] = start;

printf("Nodes reachable from node %d:\n", start);

while (front < rear) {

// Dequeue a vertex from the queue

int current = queue[front++];

printf("%d ", current);

// Get all adjacent vertices of the dequeued vertex

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

if (graph[current][i] && !visited[i]) {

// If an adjacent vertex is found and it has not been visited, mark it visited and enqueue it

visited[i] = true;

queue[rear++] = i;

}

}

}

printf("\n");

}

// Main function

int main() {

// Example graph represented as an adjacency matrix

int graph[V][V] = {0};

// Adding edges to the graph

addEdge(graph, 0, 1);

addEdge(graph, 0, 2);

addEdge(graph, 1, 2);

addEdge(graph, 2, 0);

addEdge(graph, 2, 3);

addEdge(graph, 3, 3);

addEdge(graph, 3, 4);

int startNode = 2; // Starting node for BFS

BFS(graph, startNode);

return 0;

}

Output:

Nodes reachable from node 2:

2 0 3 1 4

Check whether a given graph is connected or not using the DFS Method.

#include <stdio.h>

#include <stdbool.h>

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

// Function to add an edge to the adjacency matrix

void addEdge(int graph[V][V], int src, int dest) {

graph[src][dest] = 1;

graph[dest][src] = 1; // Since the graph is undirected

}

// Function to perform DFS and mark visited nodes

void DFS(int graph[V][V], int v, bool visited[]) {

visited[v] = true;

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

if (graph[v][i] && !visited[i]) {

DFS(graph, i, visited);

}

}

}

// Function to check if the graph is connected

bool isConnected(int graph[V][V]) {

bool visited[V] = {false};

// Perform DFS from the first vertex

DFS(graph, 0, visited);

// Check if all vertices were visited

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

if (!visited[i]) {

return false;

}

}

return true;

}

// Main function

int main() {

// Example graph represented as an adjacency matrix

int graph[V][V] = {0};

// Adding edges to the graph

addEdge(graph, 0, 1);

addEdge(graph, 0, 2);

addEdge(graph, 1, 2);

addEdge(graph, 2, 3);

addEdge(graph, 3, 4);

if (isConnected(graph)) {

printf("The graph is connected.\n");

} else {

printf("The graph is not connected.\n");

}

return 0;

}.

Output:

The graph is connected.

**Implement N Queen’s problem using Backtracking**

#include <stdio.h>

#include <stdbool.h>

#define N 8 // Size of the chessboard (N x N)

// Function to print the chessboard

void printSolution(int board[N][N]) {

printf("Solution:\n");

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

for (int j = 0; j < N; j++) {

printf(" %d ", board[i][j]);

}

printf("\n");

}

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 column on previous rows

for (i = 0; i < row; i++) {

if (board[i][col]) {

return false;

}

}

// Check upper diagonal on the left side

for (i = row, j = col; i >= 0 && j >= 0; i--, j--) {

if (board[i][j]) {

return false;

}

}

// Check upper diagonal on the right side

for (i = row, j = col; i >= 0 && j < N; i--, j++) {

if (board[i][j]) {

return false;

}

}

return true;

}

// Recursive function to solve the N-Queens problem

bool solveNQueens(int board[N][N], int row) {

if (row >= N) {

return true; // All queens are placed

}

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

if (isSafe(board, row, i)) {

board[row][i] = 1; // Place the queen

// Recur to place the rest of the queens

if (solveNQueens(board, row + 1)) {

return true;

}

// If placing queen in [row][i] doesn't lead to a solution, remove the queen

board[row][i] = 0;

}

}

return false; // Trigger backtracking

}

// Main function

int main() {

int board[N][N] = {0}; // Initialize the board to 0

if (solveNQueens(board, 0)) {

printSolution(board);

} else {

printf("No solution exists.\n");

}

return 0;

}