SANT DNYANESHWAR SHIKSHAN SANSTHA'S
ANNASAHEB DANGE COLLEGE OF ENGINEERING & TECHNOLOGY, ASHTA

DEPARTMENT OF ARTIFICIAL INTELLIENCE AND DATA SCIENCE

LABORATORY MANUAL

Design and Analysis of Algorithms

Miss. Sonali.S.Malame

THIRD YEAR

Academic Year

2024-25

Experiment List

Lab	Title of Experiment
1	Implement Maximum and Minimum using iterative versions and divide & conquer method. Compare the time complexity of both.
2	Implement Job sequencing using the Greedy problem.
3	Implement Graph coloring using the Greedy method.
4	Program based on minimum-cost spanning trees.
5	Program based on general method of dynamic programming.
6	Program based on Dynamic Programming.
7	Program based on general method of backtracking.
8	Program based on backtracking.
9	Program based on AND/OR graph.
10	Using open MP, implement a parallelized merge sort algorithm to sort a given set of elements and determine the time required to sort the elements.
11	Micro Projects. Work in teams on algorithmic projects to solve real time problems.

Experiment No.1
Maximum and minimum

Aim:- Implement Maximum and Minimum using iterative versions and divide & conquer method. Compare the time complexity of both.

Objective: To write a C program to find out maximum and minimum using the divide and conquer technique.

Theory: The problem is to find the maximum and minimum number using the divide and conquer method.

Using Divide And Conquer Approach:

As we know in the divide and conquer approach, we first divide the problem into small problems and combine the results of these small problems to solve the main problem.

In the divide & conquer approach, you recursively divide the array into smaller subarrays until each subarray has only one or two elements. Then you combine the results to find the maximum and minimum values.

In Divide and Conquer approach:

- Step 1: Find the mid of the array.
- Step 2: Find the maximum and minimum of the left subarray recursively.
- Step 3: Find the maximum and minimum of the right subarray recursively.
- Step 4: Compare the result of step 3 and step 4
- Step 5: Return the minimum and maximum.

Detailed Steps:

1. Divide the array into two halves:

- If the array is represented by indices from low to high, calculate the middle

index mid as: **mid = (low + high) / 2**

- The array is divided into two subarrays:

  Left subarray: from low to mid

  Right subarray: from mid + 1 to high

2. Conquer by recursively finding the minimum and maximum for each subarray.
3. Combine the results from the two halves:

   The overall minimum is the smaller of the minimums from the two halves.

   The overall maximum is the larger of the maximums from the two halves.

ram to find out max and min using divide and conquer rule

#C code:

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

// Structure to hold minimum and maximum values
typedef struct {
    int min;
    int max;
} MinMax;

// Function to find min and max using divide and conquer
MinMax findMaxMinDivideConquer(int arr[], int low, int high) {
    // Divide the array into two halves
    int mid = (low + high) / 2;
    MinMax leftResult = findMaxMinDivideConquer(arr, low, mid);
    MinMax rightResult = findMaxMinDivideConquer(arr, mid + 1, high);

    // Combine the results
    result.min = (leftResult.min < rightResult.min) ? leftResult.min : rightResult.min;
    result.max = (leftResult.max > rightResult.max) ? leftResult.max :
```

```
        MinMax result;


        // Base case: Only one element
        if (low == high) {
        result.min = arr[low];
        result.max = arr[low];
        return result;
        }
// Base case: Two elements
        if (high == low + 1) {
        if (arr[low] > arr[high]) {
        result.max = arr[low];
        result.min = arr[high];
        } else {
        result.max = arr[high];
        result.min = arr[low];
        }
        return result;
        }
```

```
rightResult.max;


        return result;
}


int main() {
        int arr[] = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3,
5};
        int size = sizeof(arr) / sizeof(arr[0]);


        MinMax result =
findMaxMinDivideConquer(arr, 0, size - 1);


   printf("Divide & Conquer Method:\n");
   printf("Maximum: %d\n", result.max);
   printf("Minimum: %d\n", result.min);


        return 0;
}
```

Gate Questions:

1. **Given below are some algorithms, and some algorithm design paradigms.**

1. Dijkstra's Shortest Path	i. Divide and Conquer
2. Floyd-Warshall algorithm to compute all pairs shortest path	ii. Dynamic Programming
3. Binary search on a sorted array	iii. Greedy design
4. Backtracking search on a graph	iv. Depth-first search
	v. Breadth-first search

Match the above algorithms on the left to the corresponding design paradigm they follow.

- **1-i, 2-iii, 3-i, 4-v.**
- **1-iii, 2-iii, 3-i, 4-v**
- **1-iii, 2-ii, 3-i, 4-iv**
- **1-iii, 2-ii, 3-i, 4-v.**

GATE CSE 2015 SET-2 Algorithm

2.The minimum number of comparisons required to find the minimum and the maximum of 100 numbers is _____.

- **100**
- **99**
- **198**
- **148**

GATE CSE 2014 SET-1 Algorithm

3. In quick sort, n numbers the (n/10)th element is selected as pivot using n^2 sortimng time complexity what will be the time complexity of quick sort is.....

a)O(nlogn)

b)O(n^2)

c)O(n^3)

d)O(n)

Extra questions:

1. Write C program to implement binary search using divide and conquer.
2. write C program to implement merge sort.
3. write C program to implement quick sort.

Conclusion: Thus the Program to find out maximum and minimum using divide and conquer rule has been completed successfully.

Experiment No.2

Job sequencing

Aim:- Implement Job sequencing using the Greedy problem.

Objective: To write a C program to Implement Job sequencing using the Greedy problem.

Theory: The problem is to Implement Job sequencing using the Greedy problem.

The Job Sequencing Problem is a classic problem often solved using a greedy approach.

The goal is to schedule jobs to maximize the total profit, given that each job has a

deadline by which it needs to be completed and a profit associated with it.

Approach Using Greedy Method

Sort Jobs:

First, sort the jobs in decreasing order of profit. This ensures that we consider the most profitable jobs first.

Schedule Jobs:

- Use a data structure to keep track of available time slots.
- Iterate through the sorted list and attempt to schedule each job in the latest available slot before its deadline.

Detailed Algorithm

1. **Sort:** Sort all jobs in decreasing order of profit.
2. **Initialize**:
    - Create an array to keep track of which slots are occupied.
    - Initialize the total profit and an array to keep track of the job sequence.

3. **Schedule:**
    - For each job in the sorted list:
    - Find a free slot before the job's deadline.
    - If a free slot is found, assign the job to this slot and update the total profit.
4. **Output**:
    - Print the job sequence and total profit.

#C Code:

```c
#include <stdio.h>

#include <stdlib.h>

// Job structure

typedef struct {

   int id;         // Job ID

   int deadline;   // Deadline of the job

   int profit;   // Profit of the job

} Job;

// Function to compare jobs based on their profit

int compare(const void *a, const void *b) {

   Job *jobA = (Job *)a;

   Job *jobB = (Job *)b;

   return jobB->profit - jobA->profit; // Descending order of profit

}

// Function to find the maximum profit job sequence

void jobSequencing(Job jobs[], int n) {

         // Sort jobs based on profit in descending order

   qsort(jobs, n, sizeof(Job), compare);

         // To keep track of total profit

   int totalProfit = 0;



         // Iterate through all the jobs

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

      // Find a free slot for this job

      for (int j = jobs[i].deadline - 1; j >= 0; j--) {

         // If slot is free

         if (!slotOccupied[j]) {

                  slotOccupied[j] = 1; // Mark this slot as occupied

                  result[j] = i; // Assign this job to the slot

                  totalProfit += jobs[i].profit;

                  break;

         }

      }

         }

         // Print the result

   printf("Job Sequence with Maximum Profit:\n");

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

      if (result[i] != -1) {

         printf("Job ID %d with Profit %d\n", jobs[result[i]].id, jobs[result[i]].profit);
```

```
        // Array to keep track of free time slots
    int result[n];
        // Initialize all slots as free
    for (int i = 0; i < n; i++) {
        result[i] = -1;
            }


        // Array to keep track of whether a slot is occupied
    int slotOccupied[n];
    for (int i = 0; i < n; i++) {
        slotOccupied[i] = 0;  // Initialize all slots as free
            }
        }
            }
    printf("Total Profit: %d\n", totalProfit);
}


int main() {
    Job jobs[] = {
        {1, 4, 100},
        {2, 1, 19},
        {3, 2, 27},
        {4, 1, 25},
        {5, 3, 15}
            };
    int n = sizeof(jobs) / sizeof(jobs[0]);
    jobSequencing(jobs, n);
 return 0;
}
```

Conclusion: Thus the Program to Implement Job sequencing using the Greedy problem has been completed successfully.

Experiment No.3

Graph coloring

Aim:- Implement Graph coloring using the Greedy method.

Objective: To write a C program to Implement Graph coloring using the Greedy method.

Theory: The problem is to Implement Graph coloring using the Greedy method.

Graph Coloring Problem: The Graph Coloring Problem involves assigning colors to vertices of a graph such that no two adjacent vertices share the same color. The goal is to minimize the number of colors used.

Greedy Coloring Algorithm: The Greedy Coloring Algorithm is a heuristic that provides an approximate solution by iteratively assigning colors to vertices in a specific order, ensuring that no two adjacent vertices share the same color. Although this algorithm does not always yield the optimal solution, it is efficient and often used in practice.

Steps of the Greedy Algorithm

1. **Ordering Vertices**:
   - Choose an ordering for the vertices. A common approach is to process vertices one by one, typically in the order they are listed or in a specific order determined by vertex degree or other criteria.
2. **Color Assignment**:
   - For each vertex, assign the smallest possible color that is different from the colors assigned to its adjacent vertices.
3. **Repeat**:
   - Continue this process until all vertices are colored.

Key Points:

- The number of colors used is known as the chromatic number of the graph.

- The greedy algorithm may use more colors than the chromatic number, but it is computationally efficient.

#C Code:

Below is a C implementation of the Greedy Coloring Algorithm. The code uses an adjacency matrix to represent the graph and assigns colors to each vertex.

```
#include <stdio.h>

#include <limits.h>


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


// Function to print the colors assigned to vertices

void printSolution(int color[]) {

   printf("Vertex Coloring:\n");

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

      printf("Vertex %d ---> Color %d\n", i, color[i]);

   }

}


// Function to check if it is safe to color vertex v with color c

int isSafe(int v, int graph[V][V], int color[], int c) {

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

      if (graph[v][i] && color[i] == c) {

         return 0; // Not safe if adjacent vertex has the same color

      }

   }

   // Assign the first color to the first vertex

   color[0] = 0;


   // Assign colors to the remaining vertices

   for (int u = 1; u < V; u++) {

      // Check available colors and assign the smallest valid color

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

         if (isSafe(u, graph, color, c)) {

            color[u] = c;

            break;

         }

      }

   }


   printSolution(color);

}


int main() {

   // Example graph represented as an adjacency matrix

   int graph[V][V] = {
```

```
    }

    return 1; // Safe if no adjacent vertex has the same color

}


// Function to perform greedy coloring of the graph

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

    int color[V];

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

        color[i] = -1;  // Initialize all vertices as uncolored

    }
```

```
        {0, 1, 1, 1},

        {1, 0, 1, 0},

        {1, 1, 0, 1},

        {1, 0, 1, 0}

    };


    greedyColoring(graph);

    return 0;

}
```

Conclusion: Thus the Program to Implement Graph coloring using the Greedy method has been completed successfully.

Experiment No.4
minimum-cost spanning trees

Aim:- Program based on minimum-cost spanning trees.

Objective: To write a C program to Implement minimum-cost spanning trees.

Theory: The problem is to Implement minimum-cost spanning trees.

A minimum-cost spanning tree (MST) is a fundamental concept in graph theory and optimization, used to find a subset of edges in a connected, undirected graph that connects all the vertices together without any cycles and with the minimum possible total edge weight

Key Concepts

Graph Basics:

Graph: A set of vertices (nodes) connected by edges (links).

Weighted Graph: Each edge has an associated weight (cost).

Spanning Tree: A subgraph that includes all the vertices of the original graph, is connected, and has no cycles. It has exactly V−1V-1V−1 edges if there are VVV vertices.

Minimum-Cost Spanning Tree (MST):

Objective: Find a spanning tree that has the minimum sum of edge weights among all possible spanning trees.

Unique MST: A graph can have more than one MST if there are multiple spanning trees with the same minimum total weight.

Algorithms for Finding MSTs

Several algorithms can be used to find the MST of a graph. The most well-known ones include:

Kruskal's Algorithm:

Approach: Greedy algorithm.

Steps:

- Sort all edges in non-decreasing order of their weight.
- Initialize a forest (a collection of trees), where each vertex is a separate tree.
- Iterate through the sorted edges, adding the edge to the MST if it doesn’t form a cycle with the existing edges.
- Stop when you have added V−1V-1V−1 edges.

Prim's Algorithm:

Approach: Greedy algorithm.

Steps:

- Initialize the MST with a single vertex.
- Grow the MST one edge at a time by adding the minimum-weight edge that connects a vertex in the MST with a vertex outside the MST.
- Repeat until all vertices are included in the MST.

 # C code implementation using Kruskal's algorithm:

```
#include <stdio.h>

 #include <conio.h>

   #include <stdlib.h>

   int i, j, k, a, b, u, v, n, ne = 1;

   int min, mincost = 0, cost[9][9],
parent[9];

   int find(int);

   int uni(int, int);

   void main() {

     printf("\n\tImplementation of
Kruskal's Algorithm\n");

     printf("\nEnter the no. of
vertices:");

     scanf("%d", & n);

     printf("\nEnter the cost adjacency
matrix:\n");

     for (i = 1; i <= n; i++) {

       for (j = 1; j <= n; j++) {

         scanf("%d", & cost[i][j]);

         if (cost[i][j] == 0)

           cost[i][j] = 999;

       }

       u = find(u);

       v = find(v);

       if (uni(u, v)) {

         printf("%d edge (%d,%d)
=%d\n", ne++, a, b, min);

         mincost += min;

       }

       cost[a][b] = cost[b][a] = 999;

     }

     printf("\n\tMinimum cost =
%d\n", mincost);

     getch();

   }

   int find(int i) {

     while (parent[i])

       i = parent[i];

     return i;

   }

   int uni(int i, int j) {

     if (i != j) {

       parent[j] = i;
```

```
    }

    printf("The edges of Minimum
Cost Spanning Tree are\n");

    while (ne < n) {

      for (i = 1, min = 999; i <= n; i++) {

        for (j = 1; j <= n; j++) {

          if (cost[i][j] < min) {

            min = cost[i][j];

            a = u = i;

            b = v = j;

          }   }   }
```

```
      return 1;

    }

    return 0;

}
```

Conclusion: Thus the Program to Implement minimum-cost spanning trees has been completed successfully.

Experiment No.5

Dynamic programming

Aim:- Program based on general method of dynamic programming.

Objective: To write a C program to Implement Fibonacci series using a dynamic programming approach.

Theory:

Dynamic programming is an optimization technique used to solve complex problems by breaking them down into simpler subproblems. It is particularly useful for problems with overlapping subproblems and optimal substructure properties.

The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones, typically starting with 0 and 1. The sequence looks like this:

$F(0)=0,\ F(1)=1 \quad F(n)=F(n-1) + F(n-2)$ for $n \geq 2$

Dynamic Programming Approach

The naive recursive solution to calculate Fibonacci numbers has exponential time complexity due to redundant calculations. Dynamic programming optimizes this by storing previously computed Fibonacci numbers, thus avoiding recalculation.

Steps:

1. **Memoization**: Store the results of Fibonacci numbers in an array to reuse them.
2. **Tabulation**: Iteratively fill a table based on previous results.

For the Fibonacci sequence, using a bottom-up approach (tabulation) is efficient in both time and space.

 # C code implementation of Fibonacci series:

```
#include <stdio.h>
void fibonacci(int n)
{
    int fib[n + 2];       // Array to store Fibonacci numbers
    fib[0] = 0;
    fib[1] = 1;
```

```
    for (int i = 2; i <= n; i++) {
        fib[i] = fib[i - 1] + fib[i - 2];
    }

    printf("Fibonacci Series: ");

    for (int i = 0; i < n; i++) {
        printf("%d, ", fib[i]);
    }
}

int main() {
    int n;

    printf("Enter the number of Fibonacci numbers to generate: ");
    scanf("%d", &n);

    fibonacci(n);

    return 0;
}
```

Using Recursionion

Another way to generate Fibonacci series numbers is by using recursion. In this approach, we define a recursive function that returns the nth Fibonacci number.

```
 #include <stdio.h>
int fibonacci(int n) {
    if (n <= 1)
        return n;
    else
        return (fibonacci(n - 1) + fibonacci(n - 2));
}

int main() {
    int n;

    printf("Enter the number of Fibonacci numbers to generate: ");
    scanf("%d", &n);
```

```
    printf("Fibonacci Series: ");

    for (int i = 0; i < n; i++) {
        printf("%d, ", fibonacci(i));
    }

    return 0;
}
```

Conclusion: Thus the Program to Implement Fibonacci series using a dynamic programming approach has been completed successfully.

Experiment No.6

Dynamic programming

Aim:- Program based on Dynamic Programming.

Objective: To write a C program to Implement Knapsack Problem Using Dynamic Programming approach.

Theory:

Dynamic Programming

In the Dynamic Programming,

1. We divide the large problem into multiple subproblems.

2. Solve the subproblem and store the result.

3. Using the subproblem result, we can build the solution for the large problem.

4. While solving the large problem, if the same subproblem occurs again, we can reuse the already stored result rather than recomputing it again. This is also called memoization.

Dynamic Programming Approaches

1. Bottom-Up approach

2. Top-Down approach

Bottom-Up approach

Start computing result for the subproblem. Using the subproblem result solve another subproblem and finally solve the whole problem.

Example

Let's find the nth member of a Fibonacci series.

Fibonacci(0) = 0

Fibonacci(1) = 1

Fibonacci(2) = 1 (Fibonacci(0) + Fibonacci(1))

Fibonacci(3) = 2 (Fibonacci(1) + Fibonacci(2))

We can solve the problem step by step.

1. Find Oth member

2. Find 1st member

3. Calculate the 2nd member using 0th and 1st member

4. Calculate the 3rd member using 1st and 2nd member

5. By doing this we can easily find the nth member.

Top-Down approach

Top-Down breaks the large problem into multiple subproblems.

if the subproblem solved already just reuse the answer.

Otherwise, Solve the subproblem and store the result.

Top-Down uses memoization to avoid recomputing the same subproblem again.

Let's solve the same Fibonacci problem using the top-down approach.

Top-Down starts breaking the problem unlike bottom-up.

Like,

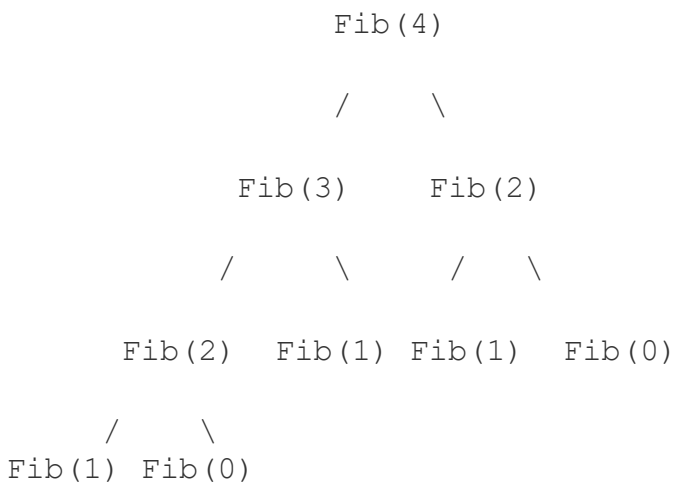
If we want to compute Fibonacci(4), the top-down approach will do the following

Fibonacci(4) -> Go and compute Fibonacci(3) and Fibonacci(2) and return the results.

Fibonacci(3) -> Go and compute Fibonacci(2) and Fibonacci(1) and return the results.

Fibonacci(2) -> Go and compute Fibonacci(1) and Fibonacci(0) and return the results.

Finally, Fibonacci(1) will return 1 and Fibonacci(0) will return 0.



C Program to Solve Knapsack Problem Using Dynamic Programming

The 0/1 Knapsack Problem can be formally defined as follows:

You are given n items, each with a weight wi and a value vi.

You have a knapsack with a maximum capacity W.

The goal is to determine the maximum total value of items that can be put into the knapsack without exceeding its capacity.

```
#include <stdio.h>
 int max(int a, int b)
{ return (a > b)? a : b; }

// Returns the maximum value that can be put in a knapsack of capacity
W
int 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];
        }
    }

    return K[n][W];
}

int main()
{
     int val[] = {60, 100, 120};
     int wt[] = {10, 20, 30};
     int  W = 50;
     int n = sizeof(val)/sizeof(val[0]);
     printf("\nValue = %d", knapsack(W, wt, val, n));
     return 0;
}
```

Conclusion: Thus the Program to Implement Knapsack Problem Using Dynamic Programming approach has been completed successfully.

Experiment No.7
Backtracking

Aim:- Program based on general method of backtracking.

Objective: To write a C program to Implement DFS Using a backtracking approach.

Theory:

Backtracking

Backtracking is a general algorithmic technique used to solve recursive problems, particularly those involving decision making and optimization. The basic idea behind backtracking is to build up a solution incrementally, step-by-step, and remove (or "backtrack") from decisions that lead to no solution. Backtracking is especially useful for solving combinatorial and constraint satisfaction problems, such as the N-Queens problem, Sudoku, maze-solving, and graph traversals.

Key Steps in the Backtracking Method

1. **Define the Problem in Terms of Constraints**:
    - Identify constraints and the conditions to satisfy.
    - Constraints are often represented as conditions that must hold true in order to consider a solution as valid.
2. **Recursive Solution Building**:
    - Use a recursive function to try and build up solutions by making incremental decisions.
    - After making a choice, check if it satisfies all constraints; if it does, move on to the next decision.
3. **Backtrack on Failure**:
    - If a choice doesn't lead to a feasible solution, undo the decision (backtrack) and try another option.
    - The algorithm effectively undoes its last choice and explores alternative paths to the solution.
4. **Pruning**:
    - Early on in the decision-making process, if it becomes clear that no solution is possible (or that it's less optimal), skip (or "prune") that path, reducing unnecessary computation.

Backtracking Pseudocode Outline

Here's a simple pseudocode for backtracking:

```
backtrack(candidate):
    if candidate is a solution:
        output(candidate)
        return
    for choice in choices:
        if choice is valid:
            make the choice
            backtrack(candidate with choice)
            undo the choice
```

C Program for Depth-First Search (DFS) using Backtracking

DFS is a graph traversal technique where you start from a root node and explore each branch as far as possible before backtracking. Here's a C program that performs DFS on a graph using backtracking.

```
#include <stdio.h>

#define MAX 100  // Define maximum number of vertices

int graph[MAX][MAX];  // Adjacency matrix

int visited[MAX];     // Visited array to keep track of visited nodes

void DFS(int vertex, int n) {
    // Mark the current node as visited and print it
    visited[vertex] = 1;
```

```
printf("%d ", vertex);
    // Recursively visit adjacent nodes
    for (int i = 0; i < n; i++) {
        if (graph[vertex][i] == 1 && !visited[i]) {
            DFS(i, n);  // Recursive call for DFS on unvisited adjacent node
        }
    }
}

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

    // Initialize adjacency matrix and visited array
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            graph[i][j] = 0;
        }
```

```
 visited[i] = 0; }

    // Input edges in the adjacency matrix

    printf("Enter the edges (format: source destination):\n");

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

        int src, dest;

        scanf("%d %d", &src, &dest);

        graph[src][dest] = 1;

        graph[dest][src] = 1; // For undirected graph

    }



    // Perform DFS traversal

    printf("DFS traversal starting from vertex 0:\n");

    DFS(0, n);  // Start DFS from vertex 0



    return 0;

}
```

Conclusion: Thus the Program to Implement DFS Using a backtracking approach has been completed successfully.

Experiment No.8

Backtracking

Aim:- Program based on backtracking.

Objective: To write a C program to Implement N-Queens problem using a backtracking approach.

Theory:

Backtracking

Backtracking is a general algorithmic technique that involves searching for a solution to a problem by trying partial solutions and then abandoning ("backtracking") them if they cannot be extended to a full solution. This approach is widely used in constraint satisfaction problems, such as the N-Queens problem, Sudoku solving, and generating combinations or permutations.

Key Concepts in Backtracking

1. Recursive Search: Backtracking is often implemented using recursion, where the function calls itself to try different possibilities.

2. Partial Solution: The algorithm builds the solution incrementally. At each step, it chooses an option, recursively explores further, and backtracks if the partial solution can't be extended.

3. Constraint Checking: Before proceeding with a choice, the algorithm checks if it satisfies the constraints.

4. Backtracking: If adding the current choice does not lead to a solution, the algorithm "backtracks" by undoing the choice and trying a new one.

Common Problems Solved by Backtracking:

1. N-Queens Problem: Placing queens on a chessboard such that no two queens attack each other.

2. Sudoku: Filling a Sudoku grid while adhering to the constraints of the puzzle (each number from 1-9 must appear exactly once in each row, column, and subgrid).

3. Maze Solving: Finding a path through a maze.

4. Subset Sum: Finding subsets of a set of numbers that sum to a particular value.

5. Permutations and Combinations: Generating all permutations or combinations of a set of elements.

6. Graph Coloring: Coloring the vertices of a graph such that no two adjacent vertices share the same color.

Advantages of Backtracking:

Systematic Search: Backtracking systematically explores all possibilities and guarantees finding a solution (if one exists).

Efficiency via Pruning: By abandoning solutions that are not valid early on, backtracking can reduce the number of possibilities that need to be explored.

Disadvantages of Backtracking:

Time-Consuming: The search space for some problems can be very large, leading to slow execution times.

Exponential Growth: In the worst case, backtracking can have exponential time complexity, particularly when the problem space is very large.

Example Problem: N-Queens Problem

The N-Queens problem is a classic example of a backtracking problem. The goal is to

place N queens on an N×N chessboard so that no two queens threaten each other.

Backtracking Algorithm for the N-Queens Problem:

Step-by-Step Explanation:

1. **Start with an empty chessboard**: We need to place N queens on the board, so we will maintain a 2D array to represent the board. Initially, all cells are empty.

2. **Try placing queens row by row**: Starting with the first row, we try to place a queen in each column.

3. **Check if the placement is valid**: For each queen placement in the current row, check:

    - If there's no other queen in the same column.

    - If there's no other queen in the same left diagonal.

    - If there's no other queen in the same right diagonal.

4. **Move to the next row**: If placing the queen is valid, move on to the next row and repeat the process.

5. **Backtrack**: If placing a queen results in an invalid configuration (i.e., no valid columns in the next row), backtrack by removing the last queen and trying a different column in the current row.

6. **Terminate when all queens are placed**: If all N queens are placed successfully, print the solution.

C Program Code for N-Queens Problem

Here's a basic C program that uses backtracking to solve the N-Queens problem.

```
#include <stdio.h>

#include <stdbool.h>

#define MAX 20


// Function to print the board

void printBoard(int board[MAX][MAX], int N) {

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

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

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

        }

       printf("\n");

        }

}

// Check if a queen can be placed on board[row][col]

bool isSafe(int board[MAX][MAX], int row, int col, int N) {

        // Check this column on all rows above this row

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

        if (board[i][col] == 1) {

        return false;
```

```
        }       }


        // Check left diagonal
        for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j] == 1) {
        return false;
        }   }


        // Check right diagonal
        for (int i = row - 1, j = col + 1; i >= 0 && j < N; i--, j++) {
        if (board[i][j] == 1) {
        return false;
        }       }
return true;
 }
 // Solve the N-Queens problem using backtracking
 bool solveNQueens(int board[MAX][MAX], int row, int N) {
        if (row == N) {
```

```
  return true;  // All queens are placed
}


        // Try all columns for the current row
        for (int col = 0; col < N; col++) {
        if (isSafe(board, row, col, N)) {
        board[row][col] = 1;  // Place queen


        // Recur to place queen in the next row
        if (solveNQueens(board, row + 1, N)) {
                return true;
        }


        // Backtrack: remove queen
        board[row][col] = 0;
        }
        }


        return false;  // If no place is found, return false
```

```
}


int main() {
        int N;
        printf("Enter the number of queens: ");
        scanf("%d", &N);

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

        if (solveNQueens(board, 0, N)) {
        printBoard(board, N);
        } else {
       printf("Solution does not exist.\n");
        }

        return 0;
}
```

Conclusion: Thus the Program to Implement N-Queens problem using a backtracking approach has been completed successfully.

Experiment No.9

AND/OR Graph

Aim:- Program based on AND/OR graph..

Objective: To write a C program to implement AND/OR graph.

Theory:

AND/OR Graphs in Design and Analysis of Algorithms (DAA)

An AND/OR graph is a type of graph used in the context of search problems, especially for decision-making or problem-solving tasks that involve multiple conditions and choices. These graphs are commonly used in artificial intelligence (AI), game theory, and operations research to represent decision processes and problems where decisions involve multiple alternatives or conditions.

Structure of an AND/OR Graph:

Nodes: The graph has nodes representing either:

OR-nodes: These nodes represent situations where any one of the possible choices or paths can lead to a solution. For example, if you have to decide between multiple alternatives, an OR-node represents a choice where taking any one of the paths could lead to success.

AND-nodes: These nodes represent situations where all conditions or paths must be satisfied or followed for a solution. For example, if you have multiple conditions that must all be true for success, an AND-node represents a decision where all conditions must be met simultaneously.

Edges: Edges connect the nodes and represent the possible transitions from one state (node) to another. These transitions can represent various actions, decisions, or choices in the problem-solving process.

Purpose of AND/OR Graphs:

AND-nodes are used when all the conditions must be true (like in logical conjunctions).

OR-nodes are used when at least one condition must be true (like in logical disjunctions).

The goal is to traverse the graph from the start node to the goal node, trying to satisfy the conditions (AND-nodes) or making choices (OR-nodes).

Applications of AND/OR Graphs:

Artificial Intelligence: In AI, AND/OR graphs are used to represent decision-making problems where agents or algorithms need to choose paths that satisfy multiple conditions.

Game Trees: Used in game theory, where players make decisions based on a combination of choices (OR) and constraints (AND).

Constraint Satisfaction Problems (CSP): In problems where solutions require satisfying multiple constraints, AND/OR graphs can model the problem as a set of AND and OR relationships between constraints.

Search Algorithms: AND/OR graphs are useful in problems where a solution requires satisfying both alternative and mandatory conditions. For example, in pathfinding problems, you might need to choose paths (OR) that must satisfy certain conditions (AND).

Example of AND/OR Graph:

Consider a decision problem where you need to reach a goal. The problem has the following structure:

OR-node: Decide which path to take. You can either go left, right, or straight.

AND-node: Once you take a path, there are conditions you must meet. For example, if you choose to go left, you must also make sure that you can cross a river (AND

condition).

This structure can be represented as:

Starting from an OR-node, you choose a path.

If you choose the left path, you encounter an AND-node where both the river must be crossed and the path must be clear.

If you choose the right path, you encounter another OR-node and continue making decisions.

Solution Approach Using AND/OR Graphs:

To solve an AND/OR graph, algorithms like Depth First Search (DFS) or Breadth First Search (BFS) can be adapted to handle the nature of AND/OR relationships:

When at an OR-node, you explore all possible paths and proceed if any path leads to a solution.

When at an AND-node, you need to ensure that all conditions are met, so you explore all conditions simultaneously.

C Code for Solving an AND/OR Graph:

Here, we will demonstrate a simple **DFS-based solution** for an AND/OR graph. The solution will explore nodes recursively, checking whether a path can be completed based on OR and AND conditions.

Example: C Code for AND/OR Graph

In this example, we'll create a simple AND/OR graph to solve a basic decision-making problem, where we simulate AND/OR conditions.

```
#include <stdio.h>

#include <stdbool.h>
```

```
#define MAX 20

// Define types for nodes in the AND/OR graph
typedef enum { AND, OR } NodeType;

// Struct to represent a node in the graph
typedef struct {
    NodeType type;     // AND or OR node
    int numChildren;    // Number of children (paths)
    int children[MAX]; // Indices of child nodes (0-based)
} Node;

// Sample AND/OR graph
Node graph[MAX];

// A helper function to simulate the DFS search in the AND/OR graph
bool dfs(int nodeIndex, bool visited[MAX]) {
    if (visited[nodeIndex]) {
```

```
return false; // Avoid revisiting nodes

}



visited[nodeIndex] = true;



// If it's an OR-node, we just need one successful path

if (graph[nodeIndex].type == OR) {

for (int i = 0; i < graph[nodeIndex].numChildren; i++) {

int child = graph[nodeIndex].children[i];

if (dfs(child, visited)) {

        return true;

}

}

return false;

}



// If it's an AND-node, all paths must be successful

if (graph[nodeIndex].type == AND) {

for (int i = 0; i < graph[nodeIndex].numChildren; i++) {
```

```
        int child = graph[nodeIndex].children[i];
        if (!dfs(child, visited)) {
                return false;
        }           }
        return true;
        }
return false;          }


 int main() {
        int N;  // Number of nodes in the graph


        printf("Enter number of nodes in the AND/OR graph: ");
        scanf("%d", &N);


        // Initialize the nodes
        for (int i = 0; i < N; i++) {
        printf("Enter type of node %d (0 for AND, 1 for OR): ", i);
        int type;
       scanf("%d", &type);
```

```
  graph[i].type = (type == 0) ? AND : OR;


  printf("Enter the number of children for node %d: ", i);
scanf("%d", &graph[i].numChildren);


  for (int j = 0; j < graph[i].numChildren; j++) {
    printf("Enter child %d for node %d: ", j + 1, i);
    scanf("%d", &graph[i].children[j]);
  }         }
  // Assume the starting node is node 0
  bool visited[MAX] = {false};
  if (dfs(0, visited)) {
printf("Solution found!\n");
  } else {
  printf("No solution exists.\n");
  }
  return 0;       }
```

Conclusion: Thus the Program to Implement AND/OR graph has been completed successfully.

Experiment No.10

Parallelized merge sort algorithm

Aim:- Using open MP, implement a parallelized merge sort algorithm to sort a given set of elements and determine the time required to sort the elements.

Objective: To write a C program to implement a parallelized merge sort algorithm.

Theory:

Parallelized Merge Sort with OpenMP:

Merge Sort is a classic divide-and-conquer algorithm that divides an array into two halves, recursively sorts each half, and then merges the sorted halves into a single sorted array. The parallelization of Merge Sort is mainly focused on speeding up the recursive divide and conquer steps (splitting and sorting subarrays), while the merge step is usually performed sequentially, as merging is inherently sequential.

1. Theory of Merge Sort

Merge Sort Algorithm:

Merge Sort works by:

Dividing the array into two halves.

Recursively sorting each half.

Merging the two sorted halves into a final sorted array.

Time Complexity:

Sequential Merge Sort: The time complexity is O(Nlogf_0N)O(N \log N)O(NlogN), where NNN is the number of elements in the array. The division of the array takes logf_0N\log NlogN levels of recursion, and at each level, the merge operation takes linear time O(N)O(N)O(N).

Parallelizing Merge Sort:

The challenge in parallelizing Merge Sort lies in how to efficiently split the problem across multiple threads and how to combine the results:

Parallelizing the Divide Step: The recursive nature of Merge Sort allows parallelizing the task of dividing the array into smaller subarrays. Each recursive call on the left and right subarrays can be processed in parallel.

Merge Step: Merging the two sorted halves is generally done sequentially, but there are ways to parallelize the merging process, though this is often more complex. For simplicity, we will leave the merge step sequential in this implementation.

Parallel Merge Sort using OpenMP:

OpenMP (Open Multi-Processing) is a parallel programming model that provides a simple and portable way to parallelize applications.

We will use OpenMP to parallelize the recursive splitting of the array.

2. C Code with OpenMP for Parallel Merge Sort

This C program implements a parallel Merge Sort algorithm using OpenMP to parallelize the recursive sorting of subarrays.

Steps for Parallel Merge Sort:

Divide the array into two halves.

Use #pragma omp parallel sections to recursively sort both halves in parallel.

Merge the sorted halves using a sequential merge operation.

Here's the implementation:

```
#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#include <time.h>


#define MAX_SIZE 1000000  // Define maximum array size for testing


// Function to merge two halves of the array

void merge(int arr[], int left, int mid, int right) {

        int n1 = mid - left + 1;

        int n2 = right - mid;


        int *L = (int*)malloc(n1 * sizeof(int));

        int *R = (int*)malloc(n2 * sizeof(int));


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

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

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

        }
```

```
for (int i = 0; i < n2; i++) {
 R[i] = arr[mid + 1 + i];
}

int i = 0, j = 0, k = left;
while (i < n1 && j < n2) {
if (L[i] <= R[j]) {
arr[k++] = L[i++];
} else {
arr[k++] = R[j++];
}
}

// Copy remaining elements of L[], if any
while (i < n1) {
arr[k++] = L[i++];
}

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

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

        // Free allocated memory
        free(L);
        free(R);
}

// Parallel merge sort function
void parallelMergeSort(int arr[], int left, int right) {
        if (left < right) {
        int mid = left + (right - left) / 2;

        // Recursively sort the two halves in parallel
        #pragma omp parallel sections
        {
        #pragma omp section
          parallelMergeSort(arr, left, mid);
```

```
        #pragma omp section
          parallelMergeSort(arr, mid + 1, right);
        }

        // Merge the sorted halves
        merge(arr, left, mid, right);
        }
}

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

// Function to print the array (optional for debugging)
void printArray(int arr[], int size) {
```

```
    for (int i = 0; i < size; i++) {
    printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[MAX_SIZE];
    int size = MAX_SIZE; // Change this to test different sizes
    generateRandomArray(arr, size);

    // Start the timer
    double start_time = omp_get_wtime();

    // Sort the array using parallel merge sort
    parallelMergeSort(arr, 0, size - 1);

    // Stop the timer
    double end_time = omp_get_wtime();
```

```
    // Optionally print the sorted array (uncomment if needed)

    // printArray(arr, size);



    // Output the time taken for sorting

    printf("Time taken for parallel merge sort: %f seconds\n", end_time - start_time);



    return 0;

}
```

Conclusion: Thus the Program to Implement parallelized merge sort algorithm has been completed successfully.

Experiment No.11

Micro Project

Micro Project Report Format: Design and Analysis of Algorithms Title Page

- Title of the Project
- Your Name
- Course Name
- Date of Submission
- Guide Name :-Prof. Sonali.S.Malame

  M.tech AI & DS

2. Abstract

Brief summary of the project (50-100 words). And the type of algorithm used (e.g., sorting, searching, dynamic programming).

3. Introduction

Introduce the context or domain the algorithm is applied to (e.g., sorting numbers, finding shortest paths).

Why is this problem important?

What is the scope of the micro-project? Is it theoretical, practical, or both? Why did you choose this problem/algorithm? What motivated you to explore it?

4. Problem Statement

Description of the Problem: State the problem the algorithm is solving.

Objective: What does the algorithm aim to achieve or optimize? (e.g., minimize time complexity, optimize resource use).

5. Algorithm Design

Describe the approach/algorithm you have implemented to solve the problem (e.g., divide-and conquer, greedy algorithm, dynamic programming).

- **Steps or Pseudocode:**

o Provide a detailed pseudocode of the algorithm.

o Optionally, you can include a flowchart to visually represent the algorithm steps. · **Explanation of Algorithm:** Provide a step-by-step explanation of the algorithm, explaining each part of the pseudocode and how it contributes to solving the problem.

6. Implementation

Programming Language: Mention the programming language used (e.g., Python, C++, Java). Code: Provide the actual code implementation. Include comments to explain important sections. Input and Output: Show sample inputs and corresponding outputs.

Screenshots: include screenshots of the program running or the output.

7. Conclusion

8. References

· Books, Research Papers, Websites, etc.: Cite all the references used for the report, including textbooks, academic papers, online resources, etc.

9. Appendices (If Applicable)

· Include any additional material that supports your project, such as:

- Additional charts/graphs.
- Extra code snippets or extended algorithms.
- Any detailed mathematical proofs or derivations.