Laboratory Manual

Design and Analysis of Algorithms Lab

Paper Code: PCC-CS494

Prepared by

Dr. Dilip Kumar Maity and Prof. Prasenjit Das

Department of Computer Science and Engineering

Academy of Technology

College Code: 169

Department of Computer Science and Engineering
Academy of Technology, G. T. Road, Adisaptagram, Aedconagar, Hooghly-712121
West Bengal, India

Contents

Introduction

An algorithm is a set of steps of operations to solve a problem performing calculation, data processing, and automated reasoning tasks. It is an efficient method that can be expressed within finite amount of time and space. An algorithm is the best way to represent the solution of a particular problem in a very simple and efficient way. If we have an algorithm for a specific problem, then we can implement it in any programming language, meaning that the algorithm is independent from any programming languages.

The important aspects of algorithm design include creating an efficient algorithm to solve a problem in an efficient way using minimum time and space. To solve a problem, different approaches can be followed. Some of them can be efficient with respect to time consumption, whereas other approaches may be memory efficient. However, one has to keep in mind that both time consumption and memory usage cannot be optimized simultaneously.

Course outcome

Upon the completion of Design and Analysis of Algorithms practical course, the student will be able to:

CO No.	Outcome	Bloom's Taxonomy Level
PCC-CS494.01	Understand and implement graph traversal algorithms like BFS and DFS	Understand/ Apply
PCC-CS494.02	Implement Binary search, Max-Min, Quick Sort, Merge Sort using Divide and conquer approach and analyze the time complexity.	Apply/ Analyze
PCC-CS494.03	Apply greedy algorithms to solve optimization problems such as fractional knapsack and job sequencing with deadlines	Apply
PCC-CS494.04	Implement algorithms for finding minimum spanning trees using Prim's and Kruskal's algorithms and Dijkstra's algorithm for single source shortest path	Apply/ Analyze
PCC-CS494.05	Explore dynamic programming techniques through implementations such as matrix chain multiplication and shortest path algorithms like Bellman-Ford and Floyd-Warshall.	Apply/ Analyze
PCC-CS494.06	Implement n-Queen problem and m-Coloring problem using backtracking algorithm	Apply

DOs and DON'Ts in Laboratory

1. Make entry in the Log Book as soon as you enter the Laboratory.
2. All the students should sit according to their roll numbers starting from their left to right.
3. All the students are supposed to enter the terminal number in the log book.
4. Do not change the terminal on which you are working.
5. All the students are expected to get at least the algorithm of the program/concept to be implemented.
6. Strictly observe the instructions given by the teacher/Lab Instructor.

Platform used in the Lab

Linux (Also known as GNU/Linux) is an operating system. It is one of the most prominent examples of open source software. Its underlying source code is available for anyone to use and modify freely. The coding is implemented through either C or C++. Linux system supports "gcc" compiler for C and "g++" compiler for C++.

Format of the lab record preparation

The students are required to maintain the lab records as per the following instruction:

1. All the records of the lab programs must be written in cover page Lab Note Book.
2. All records should have a index.
3. All records should be maintained in the following format:
    - Date
    - Program Name
    - Problem Definition and Explanation
    - Algorithm
    - Code
    - Output
    - Discussion

Evaluation System

There is two parts of the evaluation system:

1. Internal marks (40) The Internal marls will be evaluated through daily performance basis depending on the following criteria:
    - Attendance
    - Lab note book
    - Performing program in the lab
    - Viva
2. University Laboratory Examination (60) The university examination marks will be evaluated as follows:
    - Problem Definition/Algorithm (10)
    - Coding (20)
    - Output (10)
    - Viva (20)

Course Pre-Requisite

Expected Prior Knowledge and Skills: Proficiency in a C & C++ programming language, basic program design concepts (e.g, pseudo code), proof techniques, familiarity with trees and graph data structures, familiarity with basic algorithms such as those for searching, and sorting, knowledge of Discrete Structures as minimum cost spanning trees.

List of Assignments

Sl No	Assignment
1	Graph representation adjacency matrix and adjacency list.
2	Divide and conquer: Max-Min, Binary Search, Merge Sort
3	Divide and conquer: Quick sort, Randomized Quick Sort
4	Graph Traversal: BFS, DFS, Topological Sorting
5	Heap sort, Priority Queue using binary heap.
6	Greedy Algorithm: Fractional Knapsack, Job Sequencing with Deadline
7	Greedy Algorithm: Minimum Spanning tree using Kruskal Algorithm
8	Greedy Algorithm: Minimum Spanning tree using Prims Algorithm
9	Greedy Algorithm: Single source shortest path using Dijkstra Algorithm
10	Dynamic Programming: Matrix Chain Multiplication
11	Dynamic Programming: Single source shortest path using Bellman-Ford Algorithm
12	Dynamic Programming: All Pair of shortest path using Floyd-Warshall Algorithm
13	Backtracking Algorithm: n-Queen problem, m-coloring problem and Hamiltonian Cycle Problem.

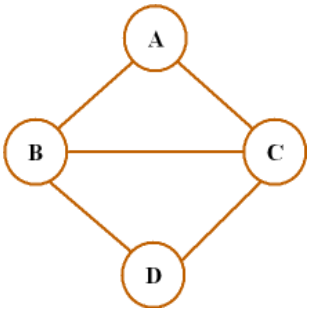
Chapter 1

Assignment 1: Graph Representation

1.1 Problem 1: Adjacency Matrix

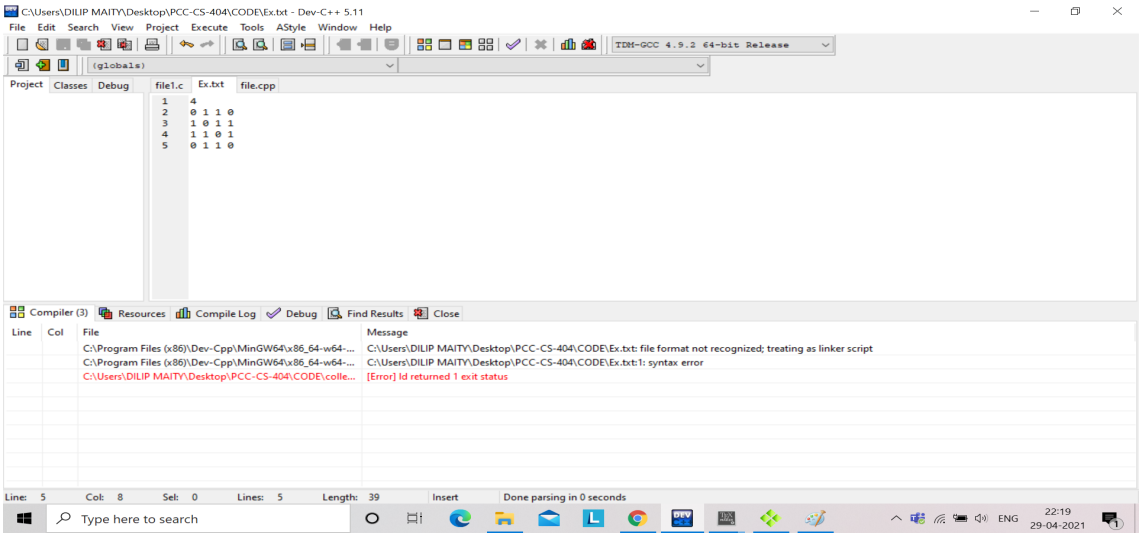
Write a program in C or C++ to read a graph from file and to store the graph in adjacency matrix. Implement the following operations.

1. Find the number of edges of the graph.
2. Find the total degree of the graph.
3. Display the adjacent of a given vertex.
4. Display the graph.



1.1.1 Representation of the graph

The information of the above graph is stored in a file ***Ex.txt***. The first line indicates the number of vertices of the graph. Rest four lines show the adjacency information of the graph.



1.1.2 Representation of the graph using adjacency matrix

The following C and CPP codes read the graph from the file and store the graph in adjacency matrix.

```c
//C code
#include <stdio.h>
#include <stdlib.h>

int main () {
   FILE *fp;     //create a pointer to a file
   fp= fopen ("Ex.txt", "r");   //open a file in read mode, ex.txt must exist in the same directory
                  //oherwise it is requir to mention the absolute path
   if (fp == NULL) {            //if file open failed then fopen() returns NULL
      printf("\nError to open the file\n");
      exit (1);
   }

   int n;
   fscanf(fp,"%d",&n); //fscanf function read a data (in specified format as scanf())
              //from file pointed by the pointer fp
   int graph[10][10];
   int i,j;
   for(i=0;i<n;i++){
      for(j=0;j<n;j++){
        fscanf(fp,"%d",&graph[i][j]); //read a graph from file
    }
   }
   for(i=0;i<n;i++){
```

```
25        for(j=0;j<n;j++){
26          printf("%3d",graph[i][j]); //read a graph from file
27       }
28       printf("\n");
29     }
30     fclose (fp);     //to close the file
31     return 0;
32  }
```

```cpp
//C++ code

#include<iostream>
#include<fstream>
using namespace std;
int main () {
   fstream infile;     //create aa object of fstream class
   infile.open("Ex.txt", ios::in);   //open a file in read mode
   (ios::in), ex.txt must exist in the same directory
                  //oherwise it is require to mention the absolute path
   if (!infile) {            //to check whether the file is opened
   successfully
      printf("\nError to open the file\n");
      exit (1);
   }
   int n;
   infile>>n; //to read number of vertices from file
   int graph[10][10];
   int i,j;
   for(i=0;i<n;i++){
      for(j=0;j<n;j++){
        infile>>graph[i][j]; //read a graph from file
    }
   }
   for(i=0;i<n;i++){
      for(j=0;j<n;j++){
        printf("%3d",graph[i][j]); //read a graph from file
      }
    printf("\n");
   }
   infile.close ();    //to close the file
   return 0;
}
```

1.2 Problem 2: Adjacency list

Write a program in C or C++ to read a graph from file and store it in adjacency list. Then implement the following operations.

1. Find the number of edges of the graph.
2. Find the total degree of the graph.
3. Display the adjacent of a given vertex.
4. Display the graph.

1.2.1 Representation of the graph using adjacency list

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

struct Node{      //structure for a node
  int vertex;
  struct Node*next;
};

struct Node *adjList[10]={NULL};        //Make array of pointers
int n;                    //number of vertices

void insertList(int i,int j){       //insert adjecent at the end of the
    list
  struct Node *curr=(struct Node*) malloc(sizeof(struct Node));
  curr->next=NULL;
  curr->vertex=j;             //insert j as adjecent
  if(adjList[i]==NULL){
    adjList[i]=curr;
    return;
  }
  struct Node *temp=adjList[i];
  while(temp->next!=NULL) temp=temp->next;
  temp->next=curr;
  return;
}

void makeAdjacencyList(){
  FILE *fp;
  fp=fopen("abc.txt","r");          //open the file
  if(fp==NULL){             //check whether the file is opened
    successfully or not
    printf("File open failed\n");
    return;
  }
  fscanf(fp,"%d",&n);    //read number of vertices from file
  int i,j;
  int data;
  for(i=0;i<n;i++){
    for(j=0;j<n;j++){
      fscanf(fp,"%d",&data);
      if(data==1) insertList(i,j);  //j is adjacent of i
    }
  }
  fclose(fp);         //close the file
}

```

```
45 void displayGraph(){
46   int i;
47   for(i=0;i<n;i++){
48     if(adjList[i]!=NULL){
49       struct Node*temp=adjList[i];
50       printf("adjacent of %c::",'A'+i);
51       while(temp!=NULL){
52         printf("%4c",temp->vertex+'A');
53         temp=temp->next;
54       }
55       printf("\n");
56     }
57   }
58   return;
59 }
60
61 int main(){
62   makeAdjacencyList();
63   displayGraph();
64   return;
65 }
```

1.3 Practice Questions

1.3.1 General Practice Questions

1. Explain what an adjacency matrix is in the context of graph representation.
2. How do you represent a directed graph using an adjacency matrix?
3. Discuss the advantages and disadvantages of using an adjacency matrix to represent a graph.
4. Explain the space complexity of an adjacency matrix.
5. Given an adjacency matrix, how would you determine if there is an edge between two vertices?
6. What is an adjacency list and how does it represent a graph?
7. Compare and contrast the adjacency list with the adjacency matrix representation.
8. How do you represent a weighted graph using an adjacency list?
9. Discuss the space complexity of an adjacency list compared to an adjacency matrix.
10. How would you find all the neighbors of a vertex using an adjacency list?

1.3.2 Advanced Practice Questions

1. Explain the process of multiplying two adjacency matrices.
2. Discuss how you would implement depth-first search (DFS) and breadth-first search (BFS) using an adjacency matrix.
3. How would you represent a weighted graph with negative weights using an adjacency matrix?
4. Explain the process of removing an edge from an adjacency list representation.
5. Discuss the advantages of using an adjacency list over an adjacency matrix when dealing with sparse graphs.
6. How would you implement Dijkstra's algorithm using an adjacency list?

Chapter 2

Assignment 2: Divide and Conquer

2.1 Max-Min Algorithm

The objective of this laboratory assignment is to implement and analyze the Min-Max Algorithm using the Divide and Conquer approach. The Min-Max Algorithm aims to find the minimum and maximum elements in an array using a more efficient approach than scanning the array linearly.

2.1.1 Task

1. Implement the Min-Max Algorithm using the Divide and Conquer approach in a C/C++ programming language.
2. Test your implementation with various input arrays of different sizes and values. Ensure that your implementation handles edge cases such as empty arrays and arrays with a single element.
3. Analyze the time complexity of your implementation and compare it with the time complexity of a linear scan approach for finding the minimum and maximum elements.

2.1.2 Algorithm

Algorithm 1: **Max-Min** $(x[], i, j, max, min)$

```
// x[0 : n − 1] is a an array of n elements. Parameters i
   and j are integers, 1 ≤ i ≤ j ≤ n. The effect is to
   set max and min to the largest and smallest values
   in x[i : j], respectively.
if i = j then max := min := x[i]; // Small P
else if i = j − 1 then
    // Another case of Small P
    if a[i] < a[j] then
        max := x[j]; min = x[i];
    end
    else
        max := x[i]; min = x[j];
    end
end
else
    // If P is not small, divide P into sub-problems
    mid := ⌊(i+l)/2⌋; // Find where to split the set
    // Solve the sub-problems
    MAX-MIN(x[], i, mid, max, min);
    MAX-MIN(x[], mid + 1, j, max1, min1);
    // Combine the solutions
    if max < max1 then max := max1;
    if min > min1 then min := min1;
end
```

2.1.3 General practice questions on viva

1. What is the max-min algorithm using divide and conquer, and how does it work?
2. How does the algorithm divide the array into subproblems, and how are the results combined?
3. What is the time complexity of the max-min algorithm? Is it efficient for large arrays?
4. Can you explain the base case of the max-min algorithm? How does it handle arrays with different lengths?
5. Compare the max-min algorithm with other methods for finding the maximum and minimum elements of an array.
6. Discuss any practical scenarios where you might need to find the maximum and minimum elements of an array.
7. Explain any difficulties you faced while implementing or understanding the max-min algorithm.

8. Can you think of any ways to optimize the max-min algorithm to make it more efficient?
9. Implement the max-min algorithm in a programming language of your choice and test it with different array sizes.
10. How does the divide and conquer strategy contribute to the efficiency of the max-min algorithm compared to other approaches?

2.1.4 Advanced practice questions on viva

1. Describe the basic idea behind the max-min algorithm using the divide and conquer approach.
2. Can you explain how the divide and conquer strategy is applied in finding the maximum and minimum elements of an array?
3. Discuss the time complexity of the max-min algorithm. How does it perform in the best, average, and worst-case scenarios?
4. Walk me through the recursive implementation of the max-min algorithm. How do you divide the problem into subproblems and combine the results?
5. How does the max-min algorithm handle arrays with even and odd lengths differently?
6. Explain any optimizations or improvements you could make to the standard max-min algorithm using divide and conquer.
7. Implement the max-min algorithm in a programming language of your choice and analyze its performance with various input sizes.
8. Compare the max-min algorithm using divide and conquer with other methods for finding the maximum and minimum elements of an array.
9. Can you discuss any real-world applications or scenarios where the max-min algorithm is particularly useful?
10. Describe any challenges you encountered while implementing or understanding the max-min algorithm using divide and conquer.

2.2 Merge Sort

2.2.1 Objective

The objective of this lab assignment is to implement the Merge Sort algorithm using C++ and analyze its time complexity. Students will learn how to create a program to implement Merge Sort, perform sorting on a given dataset, and evaluate its performance.

2.2.2 Task

Design a C++ program to implement Merge Sort on an array of integers. The program should include the following tasks:

1. Implement Merge Sort: Write a function to sort the array using the Merge Sort algorithm.
2. Generate Random Array: Write a function to generate a random array of integers.
3. Display the Array: Write a function to display the elements of the array.
4. Perform Sorting: In the main function, generate a random array, display the original array, sort it using Merge Sort, and display the sorted array.
5. Time Complexity Analysis: Measure the execution time of Merge Sort for arrays of different sizes and discuss its time complexity.

2.2.3 Algorithm

Algorithm 2: **MergeSort** ***(low,high)***

Input: An array $arr[low : high]$ is a global array to be sorted.
Output: Sorted array such that $arr[i] \leq arr[i+1]$ for all $1 \leq i \leq n$

```
// Small(P) is true if there is only one element to sort.
if low < high then
    mid := ⌊(low+high)/2⌋; // divide P into sub problems.
    MergeSort(low, mid); // Solve the first sub problem.
    MergeSort(mid + 1, high);
    // Solve the second sub problem.
    Merge(low, mid, high); // Combine the solutions.
end
```

Algorithm 3: **Merge*(low, mid, high)***

Input: An array $arr[low : high]$ is a global array containing two sorted subsets $arr[low : mid]$ and in $arr[mid + 1 : high]$.

Output: The goal is to merge these two sets into a single set residing in $arr[low : high]$. $b[]$ is an auxiliary array.

```
k := low; i := low; j := mid + 1;
while i ≤ mid and j ≤ high do
    if arr[i] < arr[j] then
        b[k] := arr[i]; i := i + 1;
    end
    else
        b[k] := arr[j]; j := j + 1;
    end
    k := k + 1;
end
while i ≤ mid do
    b[k] := arr[i]; i := i + 1; k := k + 1;
end
while j ≤ high do
    b[k] := arr[j]; j := j + 1; k := k + 1;
end
for i := low to high do arr[i] := b[i];
```

2.2.4 General practice questions on viva

1. What is merge sort and how does it work?
2. How do you merge two sorted arrays in merge sort? Explain the process.
3. Can you describe the steps involved in implementing merge sort?
4. What is the time complexity of merge sort? Is it efficient for large datasets?
5. How do you ensure that merge sort maintains stability in sorting?
6. Compare merge sort with another sorting algorithm you are familiar with. What are the differences?
7. Explain a scenario where you might prefer using merge sort over other sorting techniques.
8. Discuss any challenges you faced while implementing merge sort in a programming language.
9. Can you think of any real-life examples where sorting algorithms like merge sort are used?
10. What are some advantages of using merge sort compared to simpler sorting methods like bubble sort or selection sort?

2.2.5 Advanced practice questions on viva

1. Explain the basic idea behind merge sort algorithm.
2. Can you describe the divide-and-conquer strategy employed in merge sort? How does it contribute to its efficiency?
3. Discuss the time complexity of merge sort. How does it perform in the best, average, and worst-case scenarios?
4. Could you walk me through the recursive implementation of merge sort? How do you handle the merging step?
5. What are the advantages and disadvantages of merge sort compared to other sorting algorithms such as quicksort or insertion sort?
6. How does merge sort ensure stability in sorting?
7. Describe any optimizations or improvements you could make to the standard merge sort algorithm.
8. Implement merge sort in a programming language of your choice and analyze its performance with various input sizes.
9. Explain the concept of in-place merge sort. How does it differ from the standard merge sort?
10. Can you discuss any real-world applications or scenarios where merge sort is particularly useful?

2.3 Quick Sort

2.3.1 Objective

The objective of this lab assignment is to implement the Quick Sort algorithm using C++ and analyze its time complexity. Students will learn how to create a program to implement Quick Sort, perform sorting on a given dataset, and evaluate its performance.

2.3.2 Task

Design a C++ program to implement Quick Sort on an array of integers. The program should include the following tasks:

1. Implement Quick Sort: Write a function to sort the array using the Quick Sort algorithm.
2. Generate Random Array: Write a function to generate a random array of integers.
3. Display the Array: Write a function to display the elements of the array.

4. Perform Sorting: In the main function, generate a random array, display the original array, sort it using Quick Sort, and display the sorted array.
5. Time Complexity Analysis: Measure the execution time of Quick Sort for arrays of different sizes and discuss its time complexity.

2.3.3 How to generate random number?

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void generateRandomArray(int arr[], int length, int min_value, int
    max_value) {
    for (int i = 0; i < length; i++) {
        arr[i] = min_value + rand() % (max_value - min_value + 1);
    }
}
int main() {
    // Seed the random number generator with the current time
    srand(time(NULL));
    int length = 10;  // Adjust the length of the array as needed
    int min_value = 1;
    int max_value = 100;  // Adjust the range of random integers as
    needed
    int random_array[length];
    generateRandomArray(random_array, length, min_value, max_value);
    printf("Random Array: ");
    for (int i = 0; i < length; i++) {
        printf("%d ", random_array[i]);
    }
    printf("\n");
    return 0;
}
```

2.3.4 Algorithm

Algorithm 4: **QuickSort *(arr,low,high)***

Input: An array $arr[low : high]$ is a global array to be sorted.
Output: Sorted array such that $arr[i] \leq arr[i+1]$ for all $1 \leq i \leq n$
// Here low $\Longrightarrow$ Starting index and high $\Longrightarrow$ Ending index
if $low < high$ **then**
    // j is partitioning index, arr[j] is now at right place
    $j :=$ PARTITION(arr, low, $high$);
    QUICKSORT(arr, low, $j-1$); // Before $arr[j]$
    QUICKSORT(arr, $j+1$, $high$); // after $arr[j]$
end

Algorithm 5: **Hoare-Partition-Left** ***(arr,low,high)***

```
// This function takes first element as pivot, places the pivot element at
   its correct position in sorted array, and places all smaller (smaller than
   pivot) to left of pivot and all greater elements to right of pivot
i := low;
j := high + 1;
pivot := arr[low];
while i < j do
    do i := i + 1; while (i ≤ j and arr[i] < pivot);
    do j := j − 1; while (arr[j] > pivot);
    if i < j then INTERCHANGE(arr, i, j)
end
INTERCHANGE(arr, low, j);
return j;
```

Algorithm 6: **Lomuto-Partition-Right** ***(arr,low,high)***

```
// This function takes last element as pivot, places the pivot element at its
   correct position in sorted array, and places all smaller (smaller than
   pivot) to left of pivot and all greater elements to right of pivot
pivot := arr[high];
i := low − 1; // Temporary pivot index
for j := low to high − 1 do
    // If the current element is less than or equal to the pivot
    if arr[j] ≤ pivot then
        i := i + 1; // Move the temporary pivot index forward
        INTERCHANGE (arr, i, j); // Swap the current element with the element at
            the temporary pivot index
    end
end
// Move the pivot element to the correct pivot position (between the smaller
   and larger elements
i := i + 1;
INTERCHANGE (arr, i, high);
return i;
// the pivot index
```

2.3.5 General practice Questions for Viva

1. What is Quicksort, and why is it important in computer science?
2. Can you outline the basic idea behind Quicksort?
3. How does Quicksort achieve sorting? Describe the high-level approach.
4. What is the pivot element in Quicksort, and why is its selection crucial for the efficiency of the algorithm?
5. Explain different strategies for selecting the pivot element in Quicksort.
6. Describe the partitioning process in Quicksort.
7. How does Quicksort handle duplicate elements during partitioning?

8. What is the worst-case time complexity of Quicksort? When does it occur?
9. Discuss the space complexity of Quicksort.
10. What are the advantages of Quicksort over other sorting algorithms?
11. How does Quicksort perform on nearly sorted or already sorted input arrays?
12. Can you suggest optimizations to improve the performance of Quicksort?
13. Can Quicksort be implemented as an in-place sorting algorithm?
14. Discuss the role of recursion in Quicksort. Can it be implemented iteratively?
15. Compare Quicksort with other sorting algorithms like Merge Sort or Heap Sort.
16. What are the main drawbacks or limitations of Quicksort, and how can they be mitigated?

2.3.6 Advanced practice Questions for viva

1. Can you prove the average-case time complexity of Quicksort using mathematical induction or recurrence relations?
2. Discuss the impact of choosing different pivot selection strategies on the average-case time complexity of Quicksort.
3. What is randomized Quicksort, and how does it differ from the standard Quicksort algorithm?
4. Explain the randomized pivot selection strategy and its effects on the performance of Quicksort.
5. Describe the dual-pivot Quicksort algorithm and its advantages over traditional Quicksort implementations.
6. Compare the partitioning process in dual-pivot Quicksort with standard Quicksort.
7. How can Quicksort be parallelized to leverage multi-core processors or distributed computing environments?
8. Discuss the challenges and techniques involved in implementing parallel Quicksort.
9. Can you explain the role of memory hierarchy (cache, RAM) in influencing the performance of Quicksort?
10. Discuss cache-conscious variants of Quicksort designed to minimize cache misses and improve overall performance.
11. Is Quicksort inherently stable? Explain why or why not.
12. Describe stability-aware Quicksort algorithms designed to preserve the stability property while sorting.
13. What are some real-world applications where Quicksort is particularly well-suited?
14. Explore variants of Quicksort tailored to specific data distributions or hardware architectures, such as cache-oblivious Quicksort or GPU-based Quicksort.

Chapter 3

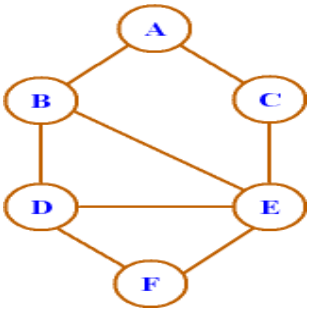
Assignment 3: Graph Traversal

3.1 Problem 1: Breadth-First-Search (BFS)

The objective of this laboratory assignment is to implement the Breadth-First Search (BFS) algorithm in C or C++ and apply it to various graph traversal problems. BFS is a fundamental graph traversal algorithm used to explore nodes in a graph level by level.

3.1.1 Task

1. Implement the Breadth-First Search algorithm in C or C++.
2. Test your implementation on different types of graphs, including directed and undirected graphs with various sizes and structures.
3. Apply the BFS algorithm to solve the following representation of the graph:
   (a) adjacency matrix
   (b) adjacency list
4. Analyze the time complexity of your BFS implementation and discuss its efficiency for different graph sizes and structures.
5. Write a report documenting your implementation, including code snippets, algorithmic details, test cases used, and results of your analysis. Discuss any optimizations applied and potential extensions or improvements to the BFS algorithm.



3.1.2 Algorithm of BFS

Algorithm 7: **BFS(v)**

```
// A breadth first search of G is carried out beginning at
   vertex v.  All vertices visited are marked as
   visited[i] = 1.  The graph G and array visited[] are global
   and visited[] is initialized to zero.
for u := 1 to |G(V)| do visited[u] := 0;
Insert v into Q; // Q is a queue of unexplored vertices
visited[v] := 1;
while Q is not empty do
    Delete u from Q; // get first unexplored vertex
    for all vertices w adjacent to u do
        if visited[w] = 0 then
            Insert w into Q; // w is unexplored
            visited[w] := 1;
        end
    end
end
```

3.1.3 General Practice Questions

1. Explain the basic idea behind BFS algorithm.
2. What data structure is typically used to implement BFS?
3. How does BFS guarantee to find the shortest path in an unweighted graph?
4. Discuss the time complexity of BFS.
5. How would you implement BFS for a directed graph?

3.1.4 Advanced Practice Questions

1. Explain how you would modify BFS to find the shortest path in a weighted graph.
2. Discuss applications of BFS in real-world scenarios.
3. How would you detect cycles in an undirected graph using BFS?

3.2 Problem 2: Depth-First-Search (DFS)

The objective of this laboratory assignment is to implement the Depth-First Search (DFS) algorithm in C or C++ and apply it to various graph traversal problems. DFS is a fundamental graph traversal algorithm used to explore nodes in a graph depth by depth.

3.3 Task

1. Implement the Depth-First Search algorithm in C or C++.
2. Test your implementation on different types of graphs, including directed and undirected graphs with various sizes and structures.
3. Apply the DFS algorithm to solve the following representation of the graph:
   (a) adjacency matrix
   (b) adjacency list
4. Analyze the time complexity of your DFS implementation and discuss its efficiency for different graph sizes and structures.
5. Write a report documenting your implementation, including code snippets, algorithmic details, test cases used, and results of your analysis. Discuss any optimizations applied and potential extensions or improvements to the DFS algorithm.

3.3.1 Iterative Algorithm of DFS

Algorithm 8: **DFS(v)**

```
// A depth first search of G is carried out beginning at
   vertex v.  All vertices visited are marked as
   visited[i] = 1.  The graph G and array visited[] are global
   and visited[] is initialized to zero.
Push v into S; // S is a stack of unexplored vertices
visited[v] := 1;
while S is not empty do
    Pop u from S; // get first unexplored vertex
    for all vertices w adjacent to u do
        if visited[w] = 0 then
            Push w into S; // w is unexplored
            visited[w] := 1;
        end
    end
end
```

3.3.2 Recursive Algorithm of DFS

Algorithm 9: **DFS-REC(v)**

```
// Given an undirected (directed) graph G = (V, E) with n
   vertices and an array visited[] initially set to zero, this
   algorithm visits all vertices reachable from v. G and
   visited[] are global.
visited[v] := 1;
for for each vertex w adjacent from v do
    if visited[w] = 0 then
        DFS-REC(w);
    end
end
```

3.3.3 General Practice Questions

1. Explain the basic idea behind DFS algorithm.
2. What data structure is typically used to implement DFS?
3. Discuss the difference between recursive and iterative implementations of DFS.
4. How does DFS handle cycles in a graph?
5. What is the time complexity of DFS?

3.3.4 Advanced Practice Questions

1. Explain how you would modify DFS to detect strongly connected components in a directed graph.
2. Discuss applications of DFS in real-world scenarios.
3. How would you determine whether a graph is bipartite using DFS?

Chapter 4

Assignment 4: Heap Sort and Priority Queue

4.1 Problem 1: Heap Sort

The objective of this laboratory assignment is to implement the Heap Sort algorithm in C or C++ and analyze its time complexity and performance.

4.1.1 Task

1. Implement the Heap Sort algorithm in C or C++.
2. Test your implementation with various input arrays of different sizes and values. Ensure that your implementation correctly sorts the input arrays in ascending order.
3. Analyze the time complexity of your implementation. Compare the theoretical time complexity of Heap Sort with its actual performance on different input sizes.
4. Compare the performance of Heap Sort with other sorting algorithms, such as Quick Sort and Merge Sort, for various input sizes.
5. Write a report documenting your implementation, including code snippets, algorithmic details, test cases used, and results of your analysis. Discuss any optimizations applied and potential improvements to the Heap Sort algorithm.

4.1.2 Algorithm

Algorithm 10: **Heap-Adjust(a, i, n)**

```
// The complete binary trees with roots 2i and 2i + 1 are combined
   with i to form a heap rooted at i, No node has an address
   greater than n or less than 1
j := 2 * i; key := a[i];
while j ≤ n do
    // compare left and right child, j points to the larger child
    if j < n and a[j] < a[j + 1] then j := j + 1;
    if key ≥ a[j] then break; // a position for key is found
    a[⌊j/2⌋] := a[j]; j := 2 * j;
    // move the larger child up a level
end
a[⌊j/2⌋] := key;
```

Algorithm 11: **Make-Heap(a, n)**

```
// Readjust the elements in A[1 : n] to form a heap
for i := ⌊n/2⌋ to 1 step −1 do
    HEAP-ADJUST(a, i, n);
end
```

Algorithm 12: **Heap-Sort(a, n)**

```
// a[1 : n] contains n elements to be sorted. Heap-Sort rearranges
   them in-place into non-decreasing order.
MAKE-HEAP(a, n); // first transform the elements into a heap
// interchange the new maximum with the element at the end of the
   tree
for i := n to 2 step −1 do
    t := a[i]; a[i] := a[1]; a[1] := t;
    HEAP-ADJUST(a, 1, i − 1);
end
```

4.1.3 General Questions

Q1. What is heap sort?

Q2. Explain the basic idea behind heap sort.

Q3. How does heap sort compare to other sorting algorithms like bubble sort and quicksort?

Q4. Describe the time complexity of heap sort.

Q5. What is the space complexity of heap sort?

Q6. Can you explain the process of building a heap in heap sort?

Q7. What is the significance of the heap property in heap sort?

Q8. How do you maintain the heap property during the heap sort algorithm?

Q9. What is the role of the max-heapify procedure in heap sort?

Q10. How do you extract the maximum element from a heap in heap sort?

4.1.4 Advanced Questions

Q11. Discuss the applications of heap sort in real-world scenarios.

Q12. Can you explain the difference between a min-heap and a max-heap?

Q13. What are the advantages and disadvantages of using heap sort?

Q14. How do you implement heap sort efficiently in practice?

Q15. Can you compare the performance of heap sort with other sorting algorithms for large datasets?

Q16. Explain the concept of heapify and its role in heap sort.

Q17. How do you handle duplicate elements in heap sort?

Q18. Discuss any optimizations that can be applied to improve the performance of heap sort.

Q19. What is the impact of input data distribution on the performance of heap sort?

Q20. Can you describe any variations or extensions of heap sort algorithm?

4.2 Problem 2: Priority Queue

Write a program in C or C++ to implement minimum priority queue using Heap. And perform the following operation.

a) Get-Minimum() to get the minimum element.

b) Extract-Min()to removes the minimum element from Min Heap.

c) Decrease-Key()to decreases value of key.

d) Insert-Key()to add a new key.

e) Delete-Key()to delete a key.

4.2.1 Algorithm

Algorithm 13: **Get-Minimum()**

return $A[1]$;

Algorithm 14: **Extract-Min()**

if $heapSize \leq 0$ **then return** ∞;
if $heapSize = 1$ **then**
  $heapSize := heapSize - 1$;
  return $A[1]$;
end
// Store the minimum value, and remove it from heap
$min := A[1]$;
$A[1] := A[heapSize]$;
$heapSize := heapSize - 1$;
HEAP-ADJUST(A, 1, $heapSize$);
return min;

Algorithm 15: **Decrease-Key(i, $newVal$)**

$A[i] := newVal$;
while $i \geq 1$ *and* $A[\text{PARENT}(i)] > A[i]$ **do**
  EXCHANGE($A[i]$, $A[\text{PARENT}(i)]$);
  $i :=$ PARENT(i);
end

Algorithm 16: **Insert-Key(k)**

if $heapSize = heapCapacity$ **then**
  WRITE"Overflow: Could not insert Key";
  return;
end
$heapSize := heapSize + 1$;
$A[heapSize] := k$; // First insert the new key at the end
$i := heapSize$;
// Fix the min heap property if it is violated
while $i \geq 1$ *and* $A[\text{PARENT}(i)] > A[i]$ **do**
  EXCHANGE($A[i]$, $A[\text{PARENT}(i)]$);
  $i :=$ PARENT(i);
end

Algorithm 17: **Delete-Key(i)**

```
if i > heapSize then
    WRITE"Delete key is not possible";
    return;
end
DECREASE-KEY (i, −∞);
EXTRACT-MIN ();
```

Chapter 5

Assignment 5: Greedy Algorithms

5.1 Problem 1: Fractional Knapsack

The objective of this laboratory assignment is to implement the Fractional Knapsack algorithm in C or C++ and apply it to solve a specific knapsack instance.

5.1.1 Task

1. Implement the Fractional Knapsack algorithm in C or C++.
2. Test your implementation using the example knapsack instance:

$$\boldsymbol{n = 7, \quad W = 15}$$

$$\boldsymbol{(p_1, p_2, \ldots, p_7) = (10, 5, 15, 7, 6, 18, 3)}$$
$$\boldsymbol{(w_1, w_2, \ldots, w_7) = (2, 3, 5, 7, 1, 4, 1)}$$

3. Apply your Fractional Knapsack algorithm to find an optimal solution to the given knapsack instance.
4. Analyze the time complexity of your algorithm and discuss its efficiency for solving the given knapsack instance.
5. Write a report documenting your implementation, including code snippets, algorithmic details, the process used to solve the given knapsack instance, and the results of your analysis.

5.1.2 Algorithm

Algorithm 18: **Knapsack(m, n)**

```
// p[1 : n] and w[1 : n] contain the profits and weights
   respectively of the n objects ordered such that
   p[i]/w[i] ≥ p[i + 1]/w[i + 1]. m is the knapsack size
   and x[1 : n] is the solution vector.
for i := 1 to n do x[i] := 0.0; // Initialize x
U := m;
for i := 1 to n do
    if w[i] > U then break;
    x[i] := 1.0; U := U − w[i];
end
if i ≤ n then x[i] := U/w[i];
```

5.1.3 General Questions

Q1. What is the fractional knapsack problem?

Q2. Explain the basic idea behind solving the fractional knapsack problem.

Q3. How does the fractional knapsack problem differ from the 0/1 knapsack problem?

Q4. Describe the greedy approach used to solve the fractional knapsack problem.

Q5. What is the time complexity of the greedy algorithm for fractional knapsack?

Q6. Can you provide an example to illustrate how the greedy approach works for fractional knapsack?

Q7. Discuss the importance of the "greedy-choice property" in the fractional knapsack algorithm.

Q8. How do you select items to include in the knapsack using the greedy approach?

Q9. What happens if the items in the fractional knapsack problem have both weights and values associated with them?

Q10. Can you explain how to handle items with non-integer weights in the fractional knapsack problem?

5.1.4 Advanced Questions

Q11. Discuss the applications of the fractional knapsack problem in real-world scenarios.

Q12. How do you handle situations where the knapsack has a weight limit in the fractional knapsack problem?

Q13. Can you compare the performance of the greedy approach with other approaches for solving the fractional knapsack problem?

Q14. What are the limitations of the greedy algorithm for fractional knapsack?

Q15. Explain any modifications or variations of the fractional knapsack problem.

Q16. How do you extend the fractional knapsack problem to include constraints other than weight limits?

Q17. Discuss any optimizations that can be applied to improve the efficiency of the greedy algorithm for fractional knapsack.

Q18. Can you describe dynamic programming approaches to solve the fractional knapsack problem?

Q19. What is the impact of the input data distribution on the performance of the greedy algorithm for fractional knapsack?

Q20. Are there any practical considerations or real-world challenges when implementing the fractional knapsack algorithm?

5.2 Problem 2: Job Sequencing with deadline

The objective of this laboratory assignment is to implement the Job Sequencing with Deadline algorithm in C or C++ and apply it to solve a specific job scheduling instance.

5.2.1 Task

1. Implement the Job Sequencing with Deadline algorithm in C or C++.
2. Test your implementation using the example job scheduling instance:

$$n = 7$$

$$(p_1, p_2, \ldots, p_7) = (3, 5, 20, 18, 1, 6, 30)$$
$$(d_1, d_2, \ldots, d_7) = (1, 3, 4, 3, 2, 1, 2)$$

3. Apply your Job Sequencing with Deadline algorithm to generate a solution for the given job scheduling instance.
4. Analyze the time complexity of your algorithm and discuss its efficiency for solving the given job scheduling instance.
5. Write a report documenting your implementation, including code snippets, algorithmic details, the process used to solve the given job scheduling instance, and the results of your analysis.

position is ————————. Explain.

5.2.2 Algorithm

Algorithm 19: **Job-scheduling*(n)***

```
// d[i] ≥ 1, 1 ≤ i ≤ n are the deadlines, n ≥ 1.
// The jobs are order such that p[1] ≥ p[2] ≥ ... ≥ p[n].
// x[j] store result (Sequence of jobs)
// slot[n] keep track of free time slots
for i := 1 to n do
    slot[i] := false; // Initialize all slots to be free
    x[i] := 0;
end
// Iterate through all given jobs
for i := 1 to n do
    // Find a free slot for this job (Note that we start
       from the last possible slot)
    for j = min(n, d[i]) to 1 step −1 do
        // Free slot found
        if slot[j] = false then
            x[j] := i; // Add this job to result
            slot[j] := true; // Make this slot occupied
            break;
        end
    end
end
```

5.2.3 General Questions

Q1. What is the job sequencing problem with deadline?

Q2. Explain the basic idea behind solving the job sequencing problem using the greedy approach.

Q3. How does the job sequencing problem differ from other scheduling problems?

Q4. Describe the greedy algorithm used to solve the job sequencing problem with deadline.

Q5. What is the objective of the greedy algorithm in job sequencing?

Q6. Can you provide an example to illustrate how the greedy approach works for job sequencing with deadline?

Q7. Discuss the importance of the "greedy-choice property" in the greedy algorithm for job sequencing.

Q8. How do you select jobs to include in the sequence using the greedy approach?

Q9. What happens if there are more jobs than available time slots in the job sequencing problem?

Q10. Can you explain how to handle jobs with varying profits and deadlines in the job sequencing problem?

5.2.4 Advanced Questions

Q11. Discuss the applications of the job sequencing problem with deadline in real-world scenarios.

Q12. How do you handle situations where jobs have dependencies in the job sequencing problem?

Q13. Can you compare the performance of the greedy approach with other approaches for solving the job sequencing problem with deadline?

Q14. What are the limitations of the greedy algorithm for job sequencing with deadline?

Q15. Explain any modifications or variations of the job sequencing problem with deadline.

Q16. How do you extend the job sequencing problem to include constraints other than deadlines?

Q17. Discuss any optimizations that can be applied to improve the efficiency of the greedy algorithm for job sequencing.

Q18. Can you describe dynamic programming approaches to solve the job sequencing problem with deadline?

Q19. What is the impact of the input data distribution on the performance of the greedy algorithm for job sequencing?

Q20. Are there any practical considerations or real-world challenges when implementing the greedy algorithm for job sequencing?

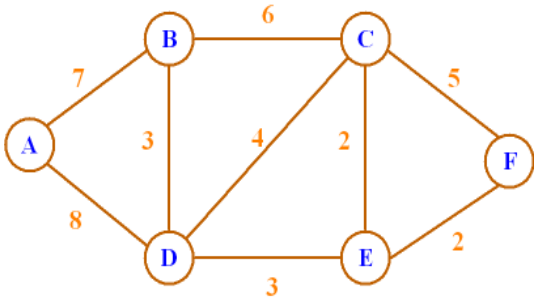
5.3 Problem 3: Kruskals Algorithm

The objective of this laboratory assignment is to implement Kruskal's Algorithm in C or C++ and apply it to find a minimum spanning tree in an undirected weighted graph.

5.3.1 Task

1. Implement Kruskal's Algorithm in C or C++.
2. Test your implementation using various undirected weighted graphs.
3. Apply your Kruskal's Algorithm implementation to find a minimum spanning tree for a given undirected weighted graph.
4. Analyze the time complexity of your algorithm and discuss its efficiency for finding a minimum spanning tree.

5. Write a report documenting your implementation, including code snippets, algorithmic details, the process used to find the minimum spanning tree, and the results of your analysis.



5.3.2 Algorithm

Algorithm 20: **MST-Kruskal*(G,w)***

$A := \Phi$;
for *each vertex* $v \in G(V)$ **do**
  MAKE-SET(V);
end
Sort the edges of E in non-decreasing order by weight w;
for *each edge* $(u, v) \in E$, *taken in non-decreasing order by weight* **do**
  if FIND-SET(u)$\neq$ FIND-SET(v) **then**
    $A := A \cup \{u, v\}$;
    UNION(u, v);
  end
end

5.3.3 General Questions

Q1. What is Kruskal's algorithm?

Q2. Describe the basic idea behind Kruskal's algorithm for finding minimum spanning trees.

Q3. How does Kruskal's algorithm differ from Prim's algorithm?

Q4. Explain the role of the greedy approach in Kruskal's algorithm.

Q5. What is a minimum spanning tree (MST), and why is it important?

Q6. What is the time complexity of Kruskal's algorithm?

Q7. How do you represent a graph in the context of Kruskal's algorithm?

Q8. Can you describe the process of selecting edges in Kruskal's algorithm?

Q9. How does Kruskal's algorithm ensure that the selected edges form a spanning tree?

Q10. Discuss the importance of the "safe edge" concept in Kruskal's algorithm.

5.3.4 Advanced Questions

Q11. Discuss the applications of Kruskal's algorithm in real-world scenarios.

Q12. How do you handle disconnected components in Kruskal's algorithm?

Q13. Can you explain the Union-Find (Disjoint Set Union) data structure and its role in Kruskal's algorithm?

Q14. What are the advantages and disadvantages of using Kruskal's algorithm?

Q15. Explain any modifications or variations of Kruskal's algorithm.

Q16. How do you handle weighted graphs with negative edge weights in Kruskal's algorithm?

Q17. Can you compare the performance of Kruskal's algorithm with other algorithms for finding minimum spanning trees?

Q18. Discuss any optimizations that can be applied to improve the efficiency of Kruskal's algorithm.

Q19. What is the impact of the input graph structure on the performance of Kruskal's algorithm?

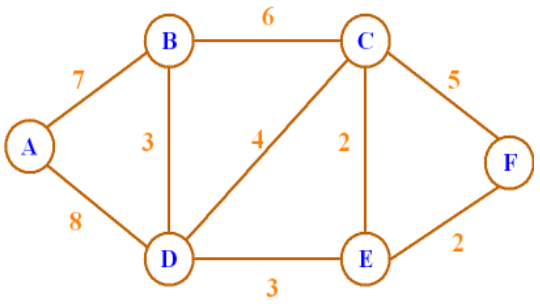
Q20. Are there any practical considerations or real-world challenges when implementing Kruskal's algorithm?

5.4 Problem 4: Prims Algorithm

The objective of this laboratory assignment is to implement Prim's Algorithm in C or C++ and apply it to find a minimum spanning tree in an undirected weighted graph.

5.4.1 Task

1. Implement Prim's Algorithm in C or C++.
2. Test your implementation using various undirected weighted graphs.
3. Apply your Prim's Algorithm implementation to find a minimum spanning tree for a given undirected weighted graph.
4. Analyze the time complexity of your algorithm and discuss its efficiency for finding a minimum spanning tree.
5. Write a report documenting your implementation, including code snippets, algorithmic details, the process used to find the minimum spanning tree, and the results of your analysis.



5.4.2 Algorithm

Algorithm 21: **Prim** $(G,\ w,\ r)$

for *each vertex* $u \in G(V)$ **do**
    $key[u] := \infty$; // Initialization
    $parent[u] := nil$;
    $mstSet[u] := false$;
end
$key[r] := 0$; // Start from root
Make a min priority Queue Q; // min-heap is used for min priority Queue
while Q *is not empty* **do**
    // Until all vertices in MST
    $u := ExtractMin(Q)$; // Delete a minimum valued vertex from heap
    $mstSet[u] := true$;
    for *each* v *adjacent from* u **do**
        if $mstSet[v] = false$ *and* $w[u, v] < key[v]$ **then**
            $DecreaseKey(v, w[u][v])$; // Update heap accordingly
            $parent[v] := u$;
        end
    end
end

5.4.3 General Questions

Q1. What is Prim's algorithm used for?

Q2. Describe the problem that Prim's algorithm aims to solve.

Q3. Explain the basic idea behind Prim's algorithm.

Q4. How does Prim's algorithm differ from other algorithms for finding minimum spanning trees, such as Kruskal's algorithm?

Q5. Discuss the role of the greedy approach in Prim's algorithm.

Q6. What is the time complexity of Prim's algorithm for finding a minimum spanning tree?

Q7. How does Prim's algorithm select the next edge to include in the minimum spanning tree?

Q8. Can you explain the significance of the "cut property" in Prim's algorithm?

Q9. What data structure is commonly used to implement Prim's algorithm efficiently?

Q10. Can you provide an example to illustrate how Prim's algorithm works step by step?

5.4.4 Advanced Questions

Q11. Discuss the applications of minimum spanning trees in real-world scenarios.

Q12. How does Prim's algorithm handle graphs with weighted edges?

Q13. Can you explain the difference between Prim's algorithm and Dijkstra's algorithm?

Q14. What are the limitations of Prim's algorithm?

Q15. Describe any modifications or variations of Prim's algorithm.

Q16. How do you handle disconnected graphs in Prim's algorithm?

Q17. Discuss any optimizations that can be applied to improve the efficiency of Prim's algorithm.

Q18. What is the impact of the input data distribution on the performance of Prim's algorithm?

Q19. Can you compare the performance of Prim's algorithm with other algorithms for finding minimum spanning trees?

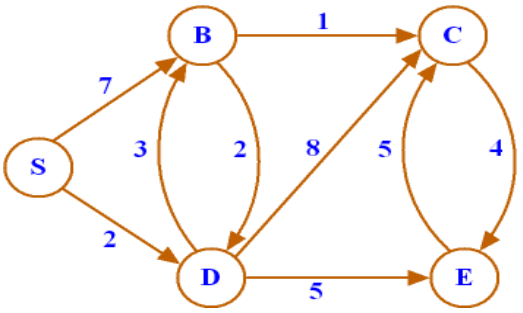
Q20. Are there any practical considerations or real-world challenges when implementing Prim's algorithm?

5.5 Problem 5: Dijkstra's Algorithm

The objective of this laboratory assignment is to implement Dijkstra's Algorithm in C or C++ and apply it to find the single source shortest paths in a weighted graph.

5.6 Task

1. Implement Dijkstra's Algorithm in C or C++.
2. Test your implementation using various weighted graphs.
3. Apply your Dijkstra's Algorithm implementation to find single source shortest paths for a given weighted graph.
4. Analyze the time complexity of your algorithm and discuss its efficiency for finding single source shortest paths.
5. Write a report documenting your implementation, including code snippets, algorithmic details, the process used to find single source shortest paths, and the results of your analysis.



5.6.1 Algorithm

Algorithm 22: **Dijkstra*(G, w, s)***

for *each vertex* $u \in G(V)$ **do**
  $dist[u] := \infty$; // Initialization
  $pred[u] := nil$;
  $sptSet[u] := false$;
end
$dist[s] := 0$; // Start from root
Make a min priority Queue Q;
// Make a min priority Queue
while Q *is not empty* **do**
  // Until all vertices in MST
  $u := extractMin(Q)$; // Delete a minimum valued vertex from heap
  $sptSet[u] := true$;
  for *each* v *adjacent from* u **do**
    if $sptSet[v] = false$ *and* $dist[u] + cost[u, v] < dist[v]$ **then**
      $decreaseKey(v, dist[u] + cost[u][v])$; // Update heap accordingly
      $pred[v] := u$;
    end
  end
end

5.6.2 General Questions

Q1. What is the shortest path problem?

Q2. Explain the basic idea behind Dijkstra's algorithm.

Q3. How does Dijkstra's algorithm differ from other algorithms for solving the shortest path problem, such as Bellman-Ford algorithm?

Q4. Describe the role of the greedy approach in Dijkstra's algorithm.

Q5. What is the time complexity of Dijkstra's algorithm for finding the shortest path?

Q6. How does Dijkstra's algorithm handle graphs with weighted edges?

Q7. Can you explain the significance of the "greedy-choice property" in Dijkstra's algorithm?

Q8. What data structure is commonly used to implement Dijkstra's algorithm efficiently?

Q9. How does Dijkstra's algorithm select the next vertex to include in the shortest path?

Q10. Can you provide an example to illustrate how Dijkstra's algorithm works step by step?

5.6.3 Advanced Questions

Q11. Discuss the applications of shortest path algorithms in real-world scenarios.

Q12. What are the limitations of Dijkstra's algorithm?

Q13. Describe any modifications or variations of Dijkstra's algorithm.

Q14. How does Dijkstra's algorithm handle negative edge weights?

Q15. Can you explain the difference between Dijkstra's algorithm and A* search algorithm?

Q16. Discuss any optimizations that can be applied to improve the efficiency of Dijkstra's algorithm.

Q17. What is the impact of the input data distribution on the performance of Dijkstra's algorithm?

Q18. Can you compare the performance of Dijkstra's algorithm with other algorithms for solving the shortest path problem?

Q19. Are there any practical considerations or real-world challenges when implementing Dijkstra's algorithm?

Q20. How does Dijkstra's algorithm handle graphs with cycles?

Chapter 6

Assignment 6: Dynamic Programming

6.1 Problem 1: 0/1 Knapsack Problem

The objective of this laboratory assignment is to implement the 0/1 Knapsack Problem using dynamic programming in C or C++ and apply it to find the maximum profit.

6.1.1 Task

1. Implement the dynamic programming approach for the 0/1 Knapsack Problem in C or C++.
2. Test your implementation using the given example:

$$\text{profit}[] = \{100, 20, 40, 60, 50, 30, 80\}$$
$$\text{weight}[] = \{5, 4, 3, 2, 1, 3, 2\}$$
$$M = 10$$

3. Apply your dynamic programming implementation to find the maximum profit and the selected items for the given knapsack instance.
4. Analyze the time complexity of your algorithm and discuss its efficiency for solving the 0/1 Knapsack Problem.
5. Write a report documenting your implementation, including code snippets, algorithmic details, the process used to find the maximum profit and the selected items, and the results of your analysis.

6.1.2 Algorithm

Algorithm 23: **Knapsack(n, M)**

```
// Let profit[1 : n] and weight[1 : n] are the profit and weight of
   n items. Algorithm returns the maximum profit value that can
   be put in a knapsack of capacity M. X[n + 1 : M + 1] is a
   temporary array. Build table X in bottom up manner
for i = 0 to n do X[i, 0] = 0; // Initialize first column
for w = 0 to M do X[0, w] = 0; // Initialize first row
for i = 0 to n do
    for w = 0 to M do
        if weight[i] ≤ w then
            X[i, w] = max(profit[i] + X[i − 1, w − weight[i]], X[i − 1, w]);
        end
        else X[i, w] = X[i − 1, w];
    end
end
return X[n, M];
```

Algorithm 24: **PrintItems(n, M)**

```
// To find the items that make this maximum profit, trace back through the
   table
// X[0..n, 0..M] is global array evaluated by Knapsack(n, M)
total_profit := X[n, M];
w := M;
for i := n to 1 step −1 do
    if total_profit = X[i − 1][w] then
        continue;
    end
    else
        write i; // This item i is included
        w = w − weight[i]; // Weight of i is deducted
        total_profit := total_profit − profit[i]; // Profit of i is deducted
        if total_profit = 0 then break;
    end
end
```

6.2 Problem 2: Matrix Chain Multiplication

The objective of this laboratory assignment is to implement the Matrix Chain Multiplication problem using Dynamic Programming in C or C++ and determine the minimum number of multiplications needed to multiply the chain along with the optimal parenthesization.

6.3 Task

1. Implement the Matrix Chain Multiplication algorithm using Dynamic Programming in C or C++.
2. Test your implementation using the given array $p[] = \{2, 3, 5, 2, 4\}$ representing the chain of matrices.
3. Apply your algorithm to determine the minimum number of multiplications needed to multiply the chain.
4. Find the optimal parenthesization.
5. Analyze the time and space complexity of your algorithm and discuss its efficiency for solving the Matrix Chain Multiplication problem.
6. Write a report documenting your implementation, including code snippets, algorithmic details, the process used to find the minimum number of multiplications and the optimal parenthesization, and the results of your analysis.

6.3.1 Algorithm

Algorithm 25: **Matrix-Chain-Order(p)**

```
// Matrix A[i] has dimension p[i − 1] × p[i] for i = 1..n
// Let M[i, j] is equal to Minimum number of scalar multiplications (i.e.,
   cost) m[i,j] needed to compute the matrix A[i] × A[i + 1] × ... × A[j] = A[i..j].
n := length(p) − 1; // length(p) = n + 1
// The cost is zero when multiplying one matrix
for i := 1 to n do M[i, i] := 0;
// Subsequence lengths
for len := 2 to n do
    for i := 1 to n − len + 1 do
        j := i + len − 1;
        M[i, j] := ∞;
        for k := i to j − 1 do
            cost := M[i, k] + M[k + 1, j] + p[i − 1] × p[k] × p[j];
            if cost < M[i, j] then
                M[i, j] := cost;
                S[i, j] := k; // Index of the subsequence split that achieved
                   minimal cost
            end
        end
    end
end
```

Algorithm 26: **Print-Optimal-Parens(S, i, j)**

```
if i = j then print "A" i;
else
    print "(";
    PRINT-OPTIMAL-PARENS(S, i, S[i, j]);
    PRINT-OPTIMAL-PARENS(S, S[i, j] + 1, j);
    print ")";
end
```

6.3.2 General Questions

Q1. What is matrix chain multiplication?

Q2. Explain the problem statement of matrix chain multiplication.

Q3. How does dynamic programming help in solving matrix chain multiplication efficiently?

Q4. Discuss the importance of matrix dimensions in matrix chain multiplication.

Q5. What is the time complexity of the dynamic programming approach for matrix chain multiplication?

Q6. Can you explain the recursive formulation of the matrix chain multiplication problem?

Q7. Describe any optimizations that can be applied to improve the efficiency of the dynamic programming approach for matrix chain multiplication.

Q8. How does matrix chain multiplication relate to other problems in computer science?

6.3.3 Advanced Questions

Q9. Discuss the applications of matrix chain multiplication in real-world scenarios.

Q10. Can you compare the performance of dynamic programming with other approaches for solving matrix chain multiplication?

Q11. What are the limitations of the dynamic programming approach for matrix chain multiplication?

Q12. Describe any modifications or variations of the dynamic programming approach for matrix chain multiplication.

Q13. How does the choice of data representation impact the efficiency of the dynamic programming approach for matrix chain multiplication?

Q14. What is the impact of the input data distribution on the performance of the dynamic programming approach for matrix chain multiplication?

Q15. Can you explain any practical considerations or real-world challenges when implementing the dynamic programming approach for matrix chain multiplication?

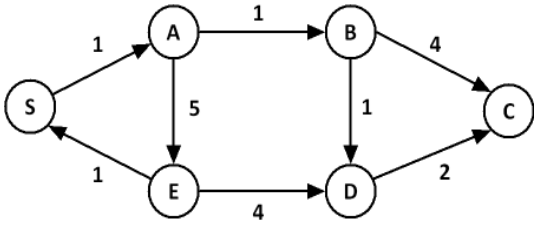
Q16. How does the dynamic programming approach handle corner cases, such as matrices with different dimensions?

6.4 Problem 3: Bellman ford

The objective of this laboratory assignment is to implement the Bellman-Ford Algorithm in C or C++ and apply it to find the single source shortest paths from a given source node in a graph.

6.4.1 Task

1. Implement the Bellman-Ford Algorithm in C or C++.
2. Test your implementation using the given graph and source node $\boldsymbol{S}$.
3. Apply your Bellman-Ford Algorithm implementation to find the single source shortest paths from the given source node.
4. What happens if we change the weight of the edge $\boldsymbol{(A, E)}$ from $\mathbf{5}$ to $\mathbf{-5}$?
5. Analyze the time complexity of your algorithm and discuss its efficiency for finding single source shortest paths.
6. Write a report documenting your implementation, including code snippets, algorithmic details, the process used to find single source shortest paths, and the results of your analysis.



6.4.2 Algorithm

Algorithm 27: **Bellman-Ford**(G, s)

```
// This implementation takes in a graph G(V, E), represented as
   lists of vertices and edges, and fills two arrays (distance
   dist[] and predecessor pred[]) about the shortest path from the
   source s to each vertex.
// initialize graph
for each vertex v ∈ G(V) do
    d[v] := ∞; // Initialize the distance to all vertices to
       infinity
    p[v] := nil; // And having a nil predecessor
end
d[s] := 0; // The distance from the source to itself
// relax edges repeatedly
for i := 1 to |V|−1 do
    for each edge (u, v) ∈ G(E) do
        if d[u] + w[u, v] < d[v] then
            d[v] := d[u] + w[u, v];
            p[v] := u;
        end
    end
end
// check for negative-weight cycles
for each edge (u, v) ∈ G(E) do
    if d[u] + w[u, v] < d[v] then return false; // Graph contains a
       negative-weight cycle
end
return true;
```

6.4.3 General Questions

Q1. What is the single-source shortest path problem?

Q2. Explain the basic idea behind the Bellman-Ford algorithm.

Q3. How does dynamic programming approach help in solving the single-source shortest path problem with the Bellman-Ford algorithm?

Q4. Describe the main steps involved in the Bellman-Ford algorithm.

Q5. What is the significance of relaxation in the Bellman-Ford algorithm?

Q6. What is the time complexity of the Bellman-Ford algorithm?

Q7. How does the Bellman-Ford algorithm handle negative edge weights?

Q8. Can you provide an example to illustrate how the Bellman-Ford algorithm works step by step?

Q9. Discuss the importance of detecting negative cycles in the Bellman-Ford algorithm.

Q10. What are the limitations of the Bellman-Ford algorithm?

6.4.4 Advanced Questions

Q11. Discuss the applications of the Bellman-Ford algorithm in real-world scenarios.

Q12. How does the Bellman-Ford algorithm compare to other algorithms for finding single-source shortest paths, such as Dijkstra's algorithm?

Q13. Can you explain any variations or extensions of the Bellman-Ford algorithm?

Q14. What optimizations can be applied to improve the efficiency of the Bellman-Ford algorithm?

Q15. Describe any practical considerations or real-world challenges when implementing the Bellman-Ford algorithm.

Q16. How does the choice of data representation impact the efficiency of the Bellman-Ford algorithm?

Q17. What is the impact of the input data distribution on the performance of the Bellman-Ford algorithm?

Q18. Can you compare the performance of the Bellman-Ford algorithm with other algorithms for finding single-source shortest paths in different scenarios?

Q19. How does the Bellman-Ford algorithm handle graphs with cycles?

Q20. What are some strategies for efficiently detecting negative cycles in the graph using the Bellman-Ford algorithm?

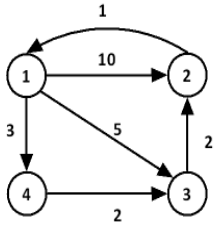
6.5 Problem 4: Floyd-Warshall

The objective of this laboratory assignment is to implement the Floyd-Warshall Algorithm in C or C++ and apply it to find the all pairs shortest paths in a graph.

6.5.1 Task

1. Implement the Floyd-Warshall Algorithm in C or C++ for the All Pairs Shortest Path problem.
2. Test your implementation using various graphs and also given graph.
3. What happens if we change the weight of the edge $(1, 2)$ from 10 to -10 of the given graph?

4. Apply your Floyd-Warshall Algorithm implementation to find all pairs shortest paths for a given graph.
5. Analyze the time complexity of your algorithm and discuss its efficiency for finding all pairs shortest paths.
6. Write a report documenting your implementation, including code snippets, algorithmic details, the process used to find all pairs shortest paths, and the results of your analysis.



6.5.2 Solution:

$$D^0 = \begin{bmatrix} 0 & 10 & 5 & 3 \\ 1 & 0 & \infty & \infty \\ \infty & 2 & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix}$$

$$D^1 = \begin{bmatrix} 0 & 10 & 5 & 3 \\ 1 & 0 & 6 & 4 \\ \infty & 2 & 0 & \infty \\ \infty & \infty & 2 & 0 \end{bmatrix} \quad D^2 = \begin{bmatrix} 0 & 10 & 5 & 3 \\ 1 & 0 & 6 & 4 \\ 3 & 2 & 0 & 6 \\ \infty & \infty & 2 & 0 \end{bmatrix} \quad D^3 = \begin{bmatrix} 0 & 7 & 5 & 3 \\ 1 & 0 & 6 & 4 \\ 3 & 2 & 0 & 6 \\ 5 & 4 & 2 & 0 \end{bmatrix} \quad D^4 = \begin{bmatrix} 0 & 7 & 5 & 3 \\ 1 & 0 & 6 & 4 \\ 3 & 2 & 0 & 6 \\ 5 & 4 & 2 & 0 \end{bmatrix}$$

Using Print-All-Pairs-Shortest-Path algorithm find the shortest path from a given vertex to another given vertex.

Question 2: What happens if we change the weight of the edge $(1, 2)$ from 10 to -10?

6.5.3 Algorithm

Algorithm 28: **Floyd-Warshall(G)**

```
// let G(V, E) be a directed graph with n (n = |V|) vertices and W be the
   weight matrix.
// let d[1 : n, 1 : n] be a n × n array of minimum distances initialized to ∞.
// let p[1 : n, 1 : n] be a n × n array of vertex indices initialized to NIL.
// Initialization
for i = 1 to n do
    for j = 1 to n do
        d[i, j] := w[i, j];
        if w[i, j] = 0 or w[i, j] = ∞ then p[i, j] := NIL;
        else p[i, j] := i;
    end
end
// Floyd-Warshall implementation
for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            if d[i, j] > d[i, k] + d[k, j] then
                d[i, j] := d[i, k] + d[k, j];
                p[i, j] := p[k, j];
            end
        end
    end
end
```

Algorithm 29: **Shortest-Path(i, j)**

```
if i = j then print i;
else if p[i, j] = NIL then print "no path from" i "to" j "exists";
else
    SHORTEST-PATH(i, p[i, j]);
    print j;
end
```

6.5.4 General Questions

Q1. What is the Floyd-Warshall algorithm used for?

Q2. Explain the basic idea behind the Floyd-Warshall algorithm.

Q3. How does the Floyd-Warshall algorithm differ from other algorithms for finding all pairs of shortest paths, such as Dijkstra's algorithm and Bellman-Ford algorithm?

Q4. Describe the role of dynamic programming in the Floyd-Warshall algorithm.

Q5. What is the time complexity of the Floyd-Warshall algorithm for finding all pairs of shortest paths?

Q6. How does the Floyd-Warshall algorithm handle graphs with negative edge weights?

Q7. Can you explain the significance of the "transitive closure" property in the Floyd-Warshall algorithm?

Q8. What data structure is commonly used to implement the Floyd-Warshall algorithm efficiently?

Q9. Can you provide an example to illustrate how the Floyd-Warshall algorithm works step by step?

6.5.5 Advanced Questions

Q10. Discuss the applications of the Floyd-Warshall algorithm in real-world scenarios.

Q11. How does the Floyd-Warshall algorithm handle disconnected graphs?

Q12. What are the limitations of the Floyd-Warshall algorithm?

Q13. Describe any modifications or variations of the Floyd-Warshall algorithm.

Q14. How does the Floyd-Warshall algorithm handle graphs with cycles?

Q15. Discuss any optimizations that can be applied to improve the efficiency of the Floyd-Warshall algorithm.

Q16. What is the impact of the input data distribution on the performance of the Floyd-Warshall algorithm?

Q17. Can you compare the performance of the Floyd-Warshall algorithm with other algorithms for finding all pairs of shortest paths?

Q18. Are there any practical considerations or real-world challenges when implementing the Floyd-Warshall algorithm?

Q19. How does the choice of data representation impact the efficiency of the Floyd-Warshall algorithm?

Chapter 7

Assignment 8: Backtracking

7.1 Problem 1: n-Queen

The objective of this laboratory assignment is to implement the n-Queen Problem using Backtracking in C or C++ and find all possible solutions for placing n queens on an $n \times n$ chessboard without any two queens threatening each other.

7.1.1 Task

1. Implement the n-Queen Problem using Backtracking in C or C++.
2. Test your implementation for various values of n, starting from small values and gradually increasing like $n = 1$, 2, 3, 4 and 8.
3. Apply your implementation to find all possible solutions for placing n queens on an $n \times n$ chessboard.
4. Analyze the time complexity of your algorithm and discuss its efficiency for finding all possible solutions.
5. Write a report documenting your implementation, including code snippets, algorithmic details, the process used to find all possible solutions, and the results of your analysis.

7.1.2 Algorithm

Algorithm 30: **NQueen**(k, n)

```
// Using backtracking, this algorithm prints all possible placement of n
   Queens on an n × n chessboard so that they are non attacking.
for i := 1 to n do
    if Place (k, i) then
        x[k] := i;
        if k = n then Write (x[1 : n]);
        else NQueen (k + 1, n);
    end
end
```

Algorithm 31: **Place(k, i)**

```
// Returns true if a queen can be placed in k^th row and i^th column. Other it
   returns false. x[1 : n] is a global array whose first (k − 1) values have
   been set. Abs(r) returns the absolute value of r.
for j := 1 to k − 1 do
    // (x[j] = i) indicates two queens are in the same column
    // |x[j] − i)|= |j − k| indicates two queens are in the same diagonal
    if (x[j] = i) or (Abs(x[j] − i) = Abs(j − k)) then return false;
end
return true;
```

7.1.3 General Questions

Q1. What is the n-Queen Problem?

Q2. Explain the problem statement of the n-Queen Problem.

Q3. How does the Backtracking algorithm help in solving the n-Queen Problem?

Q4. Discuss the significance of the Backtracking approach in solving combinatorial problems like the n-Queen Problem.

Q5. What is the time complexity of the Backtracking algorithm for solving the n-Queen Problem?

Q6. Can you explain the general structure of the Backtracking algorithm for the n-Queen Problem?

Q7. Describe any optimizations that can be applied to improve the efficiency of the Backtracking algorithm for the n-Queen Problem.

Q8. How does the n-Queen Problem relate to other problems in computer science?

7.1.4 Advanced Questions

Q9. Discuss the applications of the n-Queen Problem in real-world scenarios.

Q10. Can you compare the performance of the Backtracking algorithm with other approaches for solving the n-Queen Problem?

Q11. What are the limitations of the Backtracking approach for solving the n-Queen Problem?

Q12. Describe any modifications or variations of the Backtracking algorithm for the n-Queen Problem.

Q13. How does the choice of data representation impact the efficiency of the Backtracking algorithm for the n-Queen Problem?

Q14. What is the impact of the input data distribution on the performance of the Backtracking algorithm for the n-Queen Problem?

Q15. Can you explain any practical considerations or real-world challenges when implementing the Backtracking algorithm for the n-Queen Problem?

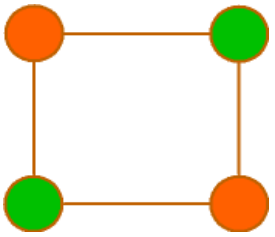
Q16. How does the Backtracking algorithm handle corner cases, such as boards of different sizes or additional constraints?

7.2 Problem 2: m-Coloring

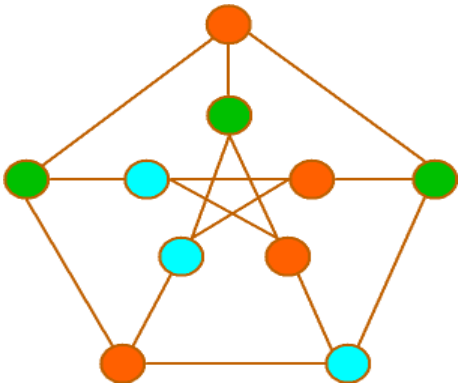
The objective of this laboratory assignment is to implement the m-Coloring Problem using Backtracking in C or C++ and find a valid coloring of a given graph with at most m colors.

7.2.1 Task

1. Implement the m-Coloring Problem using Backtracking in C or C++.
2. Test your implementation using various graphs and values of m.
3. Apply your implementation to find valid colorings of given graphs with at most m colors.
4. Test your code for $m = 1$, $m = 2$ and $m = 3$ for the following graph (one solution has been shown for $m = 2$). Print all possible solution.



5. Test your code for $m = 2$ and $m = 3$ for the following graph (one solution has been shown for $m = 3$). Print all possible solution.



6. Analyze the time complexity of your algorithm and discuss its efficiency for finding valid colorings.
7. Write a report documenting your implementation, including code snippets, algorithmic details, the process used to find valid colorings, and the results of your analysis.

7.2.2 Algorithm

Algorithm 32: **mColoring(k)**

```
// This program was formed using the recursive backtracking schema. The graph
   is represented by its boolean adjacency matrix G[1 : n, 1 : n]. All
   assignments of 1, 2, ..., m to the vertices of the graph such that adjacent
   vertices are assigned distinct integers are printed. k is the index of the
   next vertex to color.
repeat
    // generate all legal assignments for x[k]
    NextValue (k); // assign to x[k] a legal color
    if x[k] = 0 then return; // no new color possible
    // if at most m colors are assigned to n vertices
    if k = n then Write (x[1 : n]);
    else mColoring (k + 1);
until false;
```

Algorithm 33: **NextValue(k)**

```
// x[1], ..., x[k − 1] have been assigned integer values in the range [l, m] such
   that adjacent vertices have distinct integers. A value for x[k] is
   determined in the range [0, m]. x[k] is assigned the next highest numbered
   color while maintaining distinctness from the adjacent vertices of vertex
   k. If no such color exists then x[k] = 0.
repeat
    x[k] := (x[k] + 1) mod (m + 1); // next highest color
    if x[k] = 0 then return; // all colors have been exhausted
    for j := 1 to k − 1 do
        // check if this color is distinct from adjacent colors
        // if (k, j) is an edge and if adjacent vertices have identical colors
        if graph[k, j] = 1 and x[j] = x[k] then break;
    end
    if j = k then return; // new color found
    // otherwise try to find another color
until false;
```

7.2.3 General Questions

Q1. What is the m-Coloring Problem?

Q2. Explain the problem statement of the m-Coloring Problem.

Q3. How does the Backtracking algorithm help in solving the m-Coloring Problem?

Q4. Discuss the significance of the Backtracking approach in solving graph coloring problems like the m-Coloring Problem.

Q5. What is the time complexity of the Backtracking algorithm for solving the m-Coloring Problem?

Q6. Can you explain the general structure of the Backtracking algorithm for the m-Coloring Problem?

Q7. Describe any optimizations that can be applied to improve the efficiency of the Backtracking algorithm for the m-Coloring Problem.

Q8. How does the m-Coloring Problem relate to other problems in computer science?

7.2.4 Advanced Questions

Q9. Discuss the applications of the m-Coloring Problem in real-world scenarios.

Q10. Can you compare the performance of the Backtracking algorithm with other approaches for solving the m-Coloring Problem?

Q11. What are the limitations of the Backtracking approach for solving the m-Coloring Problem?

Q12. Describe any modifications or variations of the Backtracking algorithm for the m-Coloring Problem.

Q13. How does the choice of data representation impact the efficiency of the Backtracking algorithm for the m-Coloring Problem?

Q14. What is the impact of the input data distribution on the performance of the Backtracking algorithm for the m-Coloring Problem?

Q15. Can you explain any practical considerations or real-world challenges when implementing the Backtracking algorithm for the m-Coloring Problem?

Q16. How does the Backtracking algorithm handle corner cases, such as graphs with different sizes or additional constraints?