Green University of Bangladesh
Department of Computer Science and Engineering (CSE)

Faculty of Sciences and Engineering

Semester: (Spring, Year:2024), B.Sc. in CSE (Day)

Lab Report NO 03

Course Title: Artificial Intelligence

Course Code: CSE 316 Section: 212 – D3

Lab Experiment Name: Combining lab manuals 4,5 and 6 which are Iterative Deepening depth-first search(IDDFS),Implement Graph Coloring Algorithm and Solve N-Queen Problem Using Backtracking Algorithm.

Student Details

	Name	ID
1.	Sajid Rahman Rifan	212902017

Submission Date : 13 - 05 - 2024

Course Teacher's Name : Tasnim Tayiba Zannat

Lab Report Status

Marks: …… **Signature:.....**

Comments:..... **Date:.....**

1. TITLE OF THE LAB REPORT

EXPERIMENT Combining lab manuals 4,5 and 6

1. Iterative Deepening depth-first search(IDDFS) → Lab Manual 4
   Lab Exercise : Write a program to perform topological search using IDDFS.

2. Implement Graph Coloring Algorithm → Lab Manual 5
   Lab Exercise : Write a program to perform graph coloring algorithms which take input as text files from the computer.

3. Solve N-Queen Problem Using Backtracking Algorithm → Lab Manual 6
   Lab Exercise : Find all distinct solutions to the N-queen problem.

Other question answer in procedure part :

- Basic properties of IDDFS
- Why should it be used over other searching strategies like BFS and DFS and its implementation?
- What is CSP?
- How can we model a problem as CSP?
- Different strategies to solve CSP?

2. OBJECTIVES/AIM

In this lab report, I have gained a deeper understanding of search algorithms, constraint satisfaction problems, and their applications in solving combinatorial problems. By implementing and analyzing IDDFS, Graph Coloring Algorithm, and N-Queen Problem using Backtracking Algorithm, we explored fundamental concepts in artificial intelligence and their practical implications.

3. PROCEDURE / ANALYSIS / DESIGN

- **Basic properties of IDDFS →**
  IDDFS, or Iterative Deepening Depth-First Search, is a variant of depth-first search (DFS) algorithm. It combines the benefits of both breadth-first search (BFS) and depth-first search (DFS) algorithms while avoiding their respective limitations. Here are some of its basic properties:
  - IDDFS is complete, meaning it is guaranteed to find a solution if one exists.
  - Like DFS, IDDFS does not guarantee optimality in finding the shortest path to the goal.
  - IDDFS has a space complexity of O(bd).

- The time complexity of IDDFS is O(b^d).
- IDDFS uses an incremental depth limit strategy.
- IDDFS trades space for time compared to BFS
- IDDFS can be implemented recursively or iteratively.

❖ **Why should it be used over other searching strategies like BFS and DFS and its implementation?**

IDDFS offers a practical compromise between memory efficiency and completeness in searching large state spaces.

- Advantages: Memory-efficient, complete, applicable to large state spaces with unknown depths.
- Optimality: Finds the shallowest solution, not necessarily the shortest path.
- Implementation: Can be implemented recursively or iteratively.
- Usage: Suitable for scenarios where memory is a concern, and depth of solution is unknown.

❖ **What is CSP?**

CSP stands for Constraint Satisfaction Problem. It's a computational problem where the goal is to find a solution that satisfies a set of constraints. In a CSP, you have a set of variables, each with a domain of possible values, and a set of constraints that specify allowable combinations of values for subsets of variables.

❖ **How can we model a problem as CSP?**

To model a problem as a Constraint Satisfaction Problem (CSP), first, identify the entities needing values, which become variables. Define the possible values for each variable as its domain. Then, establish the relationships among variables as constraints. Express these constraints mathematically and translate the problem into a formal CSP representation. Choose an appropriate CSP solving technique, such as backtracking or constraint propagation. Apply the chosen solver to find a solution satisfying all constraints. Finally, validate and interpret the solution to ensure it meets the problem's requirements.

❖ **Different strategies to solve CSP?**

Various strategies for solving Constraint Satisfaction Problems (CSPs) include backtracking, constraint propagation, variable and value ordering heuristics, forward checking, local search methods, Integer Linear Programming (ILP), tree decomposition, and learning and adaptive techniques. These methods aim to efficiently explore the search space, enforce constraints, optimize variable assignments, and improve solver efficiency.

4. IMPLEMENTATION

1. Iterative Deepening depth-first search(IDDFS) → Manual 4

```
from collections import defaultdict, deque

class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)

    def dfs_limit(self, node, limit, visited):
        if visited[node]:
                return False
        visited[node] = True
        if limit == 0:
            return True

        for neighbor in self.graph[node]:
            if self.dfs_limit(neighbor, limit - 1, visited):
                return True

        visited[node] = False
        return False

    def topological_sort_IDDFS(self):
        max_depth = len(self.graph)
        for depth in range(max_depth):
            visited = {node: False for node in self.graph}
            stack = []
            for node in self.graph:
                if self.dfs_limit(node, depth, visited):
                    stack.append(node)
            if stack:
                return stack[::-1]
        return []

if __name__ == "__main__":
```

```
    g = Graph()

    graph_matrix = [
        [0, 0, 1, 0, 1],
        [0, 1, 1, 1, 1],
        [0, 1, 0, 0, 1],
        [1, 1, 0, 1, 1],
        [1, 0, 0, 0, 1]
    ]

    for i in range(len(graph_matrix)):
        for j in range(len(graph_matrix[0])):
            if graph_matrix[i][j] == 1:
                g.add_edge(i, j)

    top_order = g.topological_sort_IDDFS()
    print("Topological Order:", top_order)
```

Figure 1 : Topological search using IDDFS (Code)

2. Implement Graph Coloring Algorithm

```
class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for _ in range(vertices)] for _ in
range(vertices)]

     def is_safe(self, v, color, c):
        for i in range(self.V):
            if self.graph[v][i] == 1 and color[i] == c:
                return False
        return True

    def graph_coloring(self, m):
        color = [-1] * self.V
```

```
        color[0] = 0

        for v in range(1, self.V):
            for c in range(m):
                if self.is_safe(v, color, c):
                    color[v] = c
                    break

        print("Vertex Color")
        for i in range(self.V):
            print(f"{i}\t{color[i]}")


def read_graph_from_file(filename):
    with open(filename, 'r') as file:
        lines = file.readlines()
        vertices = int(lines[0])
        graph = Graph(vertices)
        for line in lines[1:]:
            edge = line.split()
            graph.graph[int(edge[0])][int(edge[1])] = 1
            graph.graph[int(edge[1])][int(edge[0])] = 1
        return graph


def main():
    filename = input("Enter the path to the graph file: ")
    graph = read_graph_from_file(filename)
    m = int(input("Enter the number of colors: "))
    graph.graph_coloring(m)


if __name__ == "__main__":
    main()
```

Figure 2 : Graph coloring algorithm which take input as text file from computer (Code)

3. Solve N-Queen Problem Using Backtracking Algorithm → Manual 6

```
matrix = []

  def checkLeftSide(row, col):
     for i in range(col):
         if matrix[row][i] == 1:
             return False
     return True

def checkLowerLef(row, col, n):
     i = row + 1
     j = col - 1
     while i < n and j >= 0:
         if matrix[i][j] == 1:
             return False
         i += 1
         j -= 1
     return True

def checkUpperLeft(row, col):
     i = row - 1
     j = col - 1
     while i >= 0 and j >= 0:
         if matrix[i][j] == 1:
             return False
         i -= 1
         j -= 1
     return True

def solve(col, n, solutions):
     if col == n:
         solutions.append([row[:] for row in matrix])
         return 1
     count = 0
     for i in range(n):
         if checkLeftSide(i, col) and checkLowerLef(i, col, n) and
checkUpperLeft(i, col):
             matrix[i][col] = 1
```

```
            count += solve(col+1, n, solutions)
            matrix[i][col] = 0
    return count

n = int(input("Give the value of n: "))

def r():
    print("dddd")
    return True


for i in range(n):
    row = []
    for j in range(n):
        row.append(0)
    matrix.append(row)

solutions = []
total_solutions = solve(0, n, solutions)

print("Number of solutions:", total_solutions)
print("Solutions:")
for i, solution in enumerate(solutions, 1):
    print(f"Solution {i}:")
    for row in solution:
        print(row)
```

Figure 3 : Find all distinct solutions to the N-queen problem. (Output)

5. TEST RESULT / OUTPUT

```
"F:\Pythone File\newproject\ver
Enter the number of edges:
6
Enter the edge (u v):
5 2
Enter the edge (u v):
5 0
Enter the edge (u v):
4 0
Enter the edge (u v):
4 1
Enter the edge (u v):
3 1
Enter the edge (u v):
2 3
Topological Sort Order:
4 5 0 2 3 1
Enter the source node:
5
Enter the target node:
3
Enter the maximum depth limit:
4

Path from 5 to 3: [5, 2, 3]
```

Topological Search using IDDFS

```
Enter the number of queens: 4
Distinct solutions for 4 queens:
Solution 1
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0

Solution 2
0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0
```

N-queen Problem

```
PS F:\Green University\8th Semester\CSE-316> &
Enter the filename of the graph: graph.txt
Vertex : Color
0 : 0
1 : 1
2 : 2
3 : 0
4 : 1
```

Graph Coloring Algorithm which takes input as text file

6. ANALYSIS AND DISCUSSION

In this lab report, I investigate three fundamental topics in artificial intelligence: Iterative Deepening Depth-First Search (IDDFS), Graph Coloring Algorithm, and the N-Queen Problem solved using Backtracking Algorithm. We begin by discussing the basic properties and implementation of IDDFS, followed by an exploration of CSP and its modeling. We then delve into strategies to solve CSPs before implementing a Graph Coloring Algorithm and solving the N-Queen Problem using Backtracking Algorithm.

IDDFS offers a memory-efficient approach to exploring large state spaces, making it advantageous over BFS and DFS in certain scenarios. CSP provides a versatile framework for modeling and solving various combinatorial problems, including graph coloring and the N-Queen Problem. By applying appropriate CSP-solving strategies, we can efficiently find solutions to complex problems.

In our implementations of the Graph Coloring Algorithm and N-Queen Problem, we observed the effectiveness of backtracking in systematically searching for valid solutions. The total number of solutions varied based on problem size and complexity, highlighting the scalability of thesealgorithms.

7. SUMMARY

Overall, the exploration of IDDFS, CSP, and related algorithms provides valuable insights into problem-solving strategies in artificial intelligence, with practical applications in various domains.