# 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 02**

**Course Title: Artificial Intelligence Lab**

**Course Code: 316 Section: 213_D4**

**Lab Experiment Name:** Lab report by combining lab manuals 4,5 and 6.

**Student Details**

| | Name | ID |
|---|---|---|
| 1. | Md.Rabby Khan | 213902037 |

**Lab Date** : 15-05-24

**Submission Date** : 13-06-24

**Course Teacher's Name** : Md Fahimul Islam

**Lab Report Status**

**Marks:** ………………………………… **Signature:**.....................

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

**1. TITLE OF THE LAB REPORT EXPERIMENT**

Lab report by combining lab manuals 4,5 and 6.

**2. OBJECTIVES/AIM**

This lab report aims to explore three search algorithms commonly used for Constraint Satisfaction Problems (CSPs):

- **Iterative Deepening Depth-First Search (IDDFS):** Investigate its properties, advantages over BFS and DFS, and provide a basic implementation.
- **Graph Coloring Algorithm:** Understand how to model graph coloring as a CSP, analyze the number of solutions, and explore backtracking to solve and print all solutions.
- **N-Queens Problem with Backtracking Algorithm:** Model the problem as a CSP, implement a backtracking solution to find all solutions for the N-Queens problem.

  By combining these explorations, this report demonstrates the application of search algorithms to solve various constraint satisfaction problems.

**3. PROCEDURE / ANALYSIS / DESIGN**

## Algorithm

**IDDFS:**

**Graph:**

- Stores connections with adjacency lists.
- Adds edges with add_edge(u, v).

**IDDFS Topological Sort:**

- Tries depths 1 to max_depth to avoid cycles.
- Tracks visited nodes (overall and within depth) for cycle detection.
- Runs limited-depth DFS for unvisited nodes at each depth.

**DFS:**

- Explores neighbors recursively, stopping at depth 0.
- Marks and checks for cycles within current depth.
- Adds itself to order on valid path found at that depth (backtracking)

**Main:**

- Runs IDDFS sort and prints order or "Cycle detected".

## GRAPH_COLORING:

**File Reading**: read_adjacency_list(file_path): Reads vertex and its neighbors from a text file.

**Graph Coloring Function**: graph_coloring(adj_list, num_of_colors): Backtracking-based coloring algorithm.

- solve(v): Recursively assigns colors, backtracks on conflicts.
- is_possible(v, c): Checks if c can be assigned to v without conflicts.
- next_node(v): Determines next vertex to color.
- display(): Prints colored vertices upon finding a solution.

## NQUEEN:

- **Initialization (__init__)**: Initializes with N and solutions.
- **print_solution(board)**: Formats and stores board configurations.
- **is_safe(board, row, col)**: Ensures no conflicts for placing queens.
- **solve_nq_util(board, col)**: Recursively attempts queen placements.
- **solve_nq()**: Finds all solutions for placing n queens.
- **Main Function (__main__)**: Handles user input and outputs solutions.

**4. IMPLEMENTATION**

**IDDFS:**

```
#  Write a program to perform topological search using IDDFS.
from collections import defaultdict

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

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

    def iddfs_topological_sort(self, max_depth):
        visited = set()
        topo_order = []
        for node in self.graph:
            if node not in visited:
                if not self.dfs(node, visited, topo_order, max_depth):
                    return "Graph has cycles, topological sorting not possible"
        return topo_order[::-1]

    def dfs(self, node, visited, topo_order, depth):
        if depth == 0:
            return False
        if node in visited:
            return True
        visited.add(node)
        for neighbor in self.graph[node]:
            if not self.dfs(neighbor, visited, topo_order, depth - 1):
                return False
        topo_order.append(node)
        return True
```

```python
def main():
    graph = Graph()

    # Input the number of nodes and edges
    num_nodes = int(input("Enter the number of nodes: "))
    num_edges = int(input("Enter the number of edges: "))

    # Input edges
    print("Enter the edges (source destination):")
    for _ in range(num_edges):
        u, v = map(int, input().split())
        graph.add_edge(u, v)

    # Set maximum depth for IDDFS
    max_depth = int(input("Enter the maximum depth for IDDFS: "))

    # Perform topological sorting using IDDFS
    sorted_nodes = graph.iddfs_topological_sort(max_depth)

    # Print the topologically sorted nodes
    if isinstance(sorted_nodes, str):
        print(sorted_nodes)
    else:
        print("Topologically sorted nodes:")
        print(sorted_nodes)

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

- **OUTPUT_1**

```
Enter the number of nodes: 5
Enter the number of edges: 5
Enter the edges (source destination):
1 2
1 3
2 4
3 4
4 5
Enter the maximum depth for IDDFS: 3
Graph has cycles, topological sorting not possible
```

**GRAPH COLORING:**

```
# Write a program to perform graph coloring algorithm which take input as text file from
computer.
def read_adjacency_list(file_path):
    adjacency_list = {}
    with open(file_path, 'r') as file:
        for line in file:
            nodes = line.strip().split()
            vertex = nodes[0]
            neighbors = nodes[1:]
            adjacency_list[vertex] = neighbors
    return adjacency_list

def graph_coloring(adj_list, num_of_colors):
    colors = {}


    def solve(v):
        if v not in adj_list:
            raise Exception("Solution found")
```

```
for c in range(1, num_of_colors + 1):
        if is_possible(v, c):
            colors[v] = c


         solve(next_node(v))
            colors[v] = 0

    def is_possible(v, c):
        for neighbor in adj_list[v]:
            if colors.get(neighbor) == c:
                return False
        return True

    def next_node(v):
        for node in adj_list:
            if node not in colors:
                return node
        return None

    def display():
        text_color = ["", "RED", "GREEN", "BLUE", "YELLOW", "ORANGE", "PINK",
                    "BLACK", "BROWN", "WHITE", "PURPLE", "VIOLET"]
        print("Colors:")
        for node, color_index in colors.items():
            print(f"{node}: {text_color[color_index]}")

    try:
        solve(next(iter(adj_list)))
        print("No solution")
    except Exception as e:
        print("\nSolution exists")
        display()
```

```python
if __name__ == "__main__":
    file_path = input("Enter the path to the text file containing the adjacency list: ")
    num_colors = int(input("Enter the number of colors: "))

    adj_list = read_adjacency_list(file_path)
    graph_coloring(adj_list, num_colors)
```

- **OUTPUT_2:**

```
Enter the path to the text file containing the adjacency list: graph.txt
Enter the number of colors: 3

Solution exists
Colors:
7: RED
0: RED
1: GREEN
```

**NQUEEN:**

```
class NQueen:
    def __init__(self, N):
        self.N = N
        self.solutions = []

    def print_solution(self, board):
        solution = []
        for row in board:
            solution.append(" ".join(map(str, row)))
        self.solutions.append(solution)

    def is_safe(self, board, row, col):
        for i in range(col):
            if board[row][i] == 1:
                return False

        for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
            if board[i][j] == 1:
                return False

        for i, j in zip(range(row, self.N, 1), range(col, -1, -1)):
            if board[i][j] == 1:
                return False

        return True

    def solve_nq_util(self, board, col):
        if col >= self.N:
            self.print_solution(board)
            return True
```

```python
        res = False
        for i in range(self.N):
            if self.is_safe(board, i, col):
                board[i][col] = 1
                res = self.solve_nq_util(board, col + 1) or res
                board[i][col] = 0  # BACKTRACK

        return res

    def solve_nq(self):
        board = [[0] * self.N for _ in range(self.N)]
        self.solve_nq_util(board, 0)
        return self.solutions


if __name__ == "__main__":
    #
    n = int(input("Number of queens to place: "))
    queen = NQueen(n)
    solutions = queen.solve_nq()
    for idx, solution in enumerate(solutions):
        print(f"Solution {idx + 1}:")
        for row in solution:
            print(row)
        print()
```

➢ **OUTPUT_3:**

```
Number of queens to place: 4
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
```

## 5. ANALYSIS AND DISCUSSION

The Python code encompasses three essential algorithms that demonstrate key principles of graph theory and combinatorial optimization:

1. **IDDFS-based Topological Sorting for Directed Graphs:**This algorithm uses Iterative Deepening Depth-First Search (IDDFS) to perform topological sorting. It handles directed acyclic graphs (DAGs) by progressively increasing the depth limit, ensuring that nodes are processed in a linear order that respects their dependencies. This method is particularly useful in scheduling tasks where certain tasks must precede others.

2. **Graph Coloring Using Backtracking:**This algorithm attempts to color the vertices of a graph using a given number of colors such that no two adjacent vertices share the same color. By reading an adjacency list from a text file, the algorithm employs backtracking to explore all possible color assignments. This technique is significant in resource allocation problems where conflicts need to be minimized, such as register allocation in compilers and frequency assignment in wireless networks.

3. **N-Queens Problem Solving via Backtracking:**The N-Queens problem involves placing N queens on an N×N chessboard such that no two queens threaten each other. The algorithm uses backtracking to explore possible configurations, ensuring that each queen is placed safely. This problem exemplifies combinatorial optimization and can be extended to various constraint satisfaction problems, including puzzle-solving and layout design.