Experiment Title: Implementation of Programs on NCDP as NP Hard problem,

Aim/Objective: To understand the concept and implementation of Basic program on NCDP as NP Hard problem

Description: The students will understand and able to implement programs on NCDP as NP Hard problem.

Pre-Requisites:

Knowledge: NCDP as NP Hard problem in C/C++/PythonTools: Code Blocks/Eclipse IDE

Pre-Lab:

Given a graph G = (V, E) and an integer k, determine if there exists a vertex cover of size k or less.

Input: Number of vertices, edges, adjacency matrix, and k.

Output: Yes/No, whether a vertex cover of size k exists.

- **Procedure/Program:**

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

#define MAX_VERTICES 100

int covers_all_edges(int graph[MAX_VERTICES][MAX_VERTICES], int n, int subset[], int
subset_size) {
  for (int u = 0; u < n; u++) {
    for (int v = u + 1; v < n; v++) {
      if (graph[u][v] == 1) {
        int found = 0;
        for (int i = 0; i < subset_size; i++) {
          if (subset[i] == u || subset[i] == v) {
            found = 1;
            break;
          }
        }
        if (!found) {
          return 0;
        }
      }
    }
  }
```

```
    return 1;
}

int is_vertex_cover(int graph[MAX_VERTICES][MAX_VERTICES], int n, int k) {
    int *subset = (int *)malloc(n * sizeof(int));

    for (int subset_size = 0; subset_size <= k; subset_size++) {
        for (int i = 0; i < (1 << n); i++) {
            int count = 0;
            for (int j = 0; j < n; j++) {
                if (i & (1 << j)) {
                    subset[count++] = j;
                }
            }

            if (count == subset_size && covers_all_edges(graph, n, subset, count)) {
                free(subset);
                return 1;
            }
        }
    }

    free(subset);
    return 0;
}

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

    int graph[MAX_VERTICES][MAX_VERTICES] = {0};
    printf("Enter the adjacency matrix:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &graph[i][j]);
        }
    }

    printf("Enter the value of k: ");
```

```
    scanf("%d", &k);

    if (is_vertex_cover(graph, n, k)) {
        printf("Yes\n");
    } else {
        printf("No\n");
    }

    return 0;
}
```

In-Lab:

Find the minimum vertex cover of a given graph. A vertex cover is a set of vertices such that every edge in the graph has at least one endpoint in the set.

Input: Number of vertices, edges, and adjacency matrix.

Output: Minimum vertex cover and its vertices.

- **Procedure/Program:**

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

void findMinVertexCover(int vertices, int adj[vertices][vertices]) {
    bool visited[vertices];
    for (int i = 0; i < vertices; i++) {
        visited[i] = false;
    }

    printf("Minimum vertex cover: ");

    for (int u = 0; u < vertices; u++) {
        for (int v = u + 1; v < vertices; v++) {
            if (adj[u][v] == 1 && !visited[u] && !visited[v]) {
                visited[u] = visited[v] = true;
                printf("%d %d ", u, v);
            }
        }
    }
```

```
    printf("\n");
}

int main() {
    int vertices = 4;
    int adj[4][4] = {
        {0, 1, 0, 1},
        {1, 0, 1, 0},
        {0, 1, 0, 1},
        {1, 0, 1, 0}
    };

    findMinVertexCover(vertices, adj);
    return 0;
}
```

Post-Lab

Determine whether there exists a Hamiltonian cycle in a graph that visits each vertex exactly once and returns to the starting point.

Input: Number of vertices and the adjacency matrix of the graph.

Output: Yes/No and the cycle if it exists.

- **Procedure/Program:**

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

bool is_safe(int v, int pos, int path[], int graph[][5], int n) {
    if (graph[path[pos - 1]][v] == 0) {
        return false;
    }
    for (int i = 0; i < pos; i++) {
        if (path[i] == v) {
            return false;
        }
    }
    return true;
```

```
}

bool hamiltonian_cycle_util(int graph[][5], int path[], int pos, int n) {
    if (pos == n) {
        if (graph[path[pos - 1]][path[0]] == 1) {
            return true;
        }
        return false;
    }
    for (int v = 1; v < n; v++) {
        if (is_safe(v, pos, path, graph, n)) {
            path[pos] = v;
            if (hamiltonian_cycle_util(graph, path, pos + 1, n)) {
                return true;
            }
            path[pos] = -1;
        }
    }
    return false;
}

void hamiltonian_cycle(int graph[][5], int n) {
    int path[n];
    for (int i = 0; i < n; i++) {
        path[i] = -1;
    }
    path[0] = 0;

    if (!hamiltonian_cycle_util(graph, path, 1, n)) {
        printf("No\n");
        return;
    }

    printf("Yes\nCycle: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", path[i]);
    }
    printf("%d\n", path[0]);
}
```

```
int main() {
   int graph[5][5] = {
      {0, 1, 1, 1, 0},
      {1, 0, 1, 1, 0},
      {1, 1, 0, 1, 0},
      {1, 1, 1, 0, 0},
      {0, 0, 0, 0, 0}
   };
   int n = 4;

   hamiltonian_cycle(graph, n);
   return 0;
}
```

- **Data and Results:**

Data:

Graph with 4 vertices and the following adjacency matrix provided.

Result:

Yes, Hamiltonian cycle exists: 0 1 2 3 0.

- **Analysis and Inferences:**

Analysis:

Graph allows visiting each vertex exactly once, returning to start.

Inferences:

Hamiltonian cycle successfully detected using backtracking and adjacency matrix validation.

- **Sample VIVA-VOCE Questions:**

1. Differentiate between CDP and NCDP?

- **CDP (Centralized Decision Problem)**: Solved by a central authority or algorithm.
- **NCDP (Non-Centralized Decision Problem)**: Solved by distributed processes or multiple agents.

2. List the NP-Hard Graph problems?

- Hamiltonian Cycle
- Graph Coloring
- Clique Problem
- Vertex Cover

3. What is Reducibility?

- The process of transforming one problem into another, ensuring that a solution to the second problem solves the first.

4. Identify one difference between Satisfiability and Reducibility?

- **Satisfiability**: Determines if a logical formula can be satisfied by some assignment.
- **Reducibility**: Involves converting one problem to another to solve it.

5. What is Hamiltonian Graph Cycle?

- A cycle that visits each vertex exactly once and returns to the starting point.

Evaluator Remark (if Any):	**Marks Secured ___ out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.