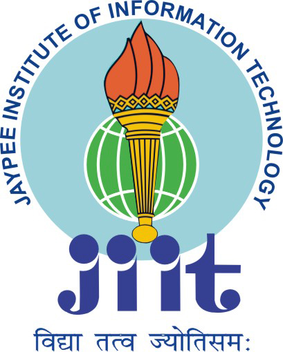
**JAYPEE INSTITUTE OF INFORMATION TECHNOLOGY**

SECTOR 62, NOIDA



DATA STRUCTURE AND ALGORITHM PROJECT

**Project Title – Hamiltonian Path and Circuit Finder**

Submitted by –

Ajaydeep 22104005

Sanskar Srivastava 22104014

Lokendra Singh Shekhawat 22104006

Submitted to –

**Mr. Kashav Ajmera**

**DECLARATION BY CANDIDATES**

We, the undersigned students, hereby declare our commitment and collaboration in undertaking the project titled "HAMILTONIAN PATH AND CIRCUIT FINDER" As a team, we are dedicated to the successful planning, execution, and completion of this project, pooling our collective skills, knowledge, and resources.

1. Ajaydeep (22104005)
2. Lokendra Singh Shekhawat (22104006)
3. Sanskar Srivastava (22104014)

We acknowledge the role of our project supervisor, MR. KASHAV AJMERA SIR, and appreciate the guidance and feedback he gave us throughout the project’s lifecycle.

This is an authentic piece of work and in case there is any query regarding the same, we should be held responsible for answering the queries in this regard.

**Problem Statement and Objective:**

The "Hamiltonian Path Finder" is a software application designed to solve the Hamiltonian Path Problem for graphs. The Hamiltonian Path Problem is a classic graph theory problem that involves finding a path in a graph that visits each vertex exactly once. This project aims to create an efficient and user-friendly program to determine the existence of Hamiltonian Paths in arbitrary graphs.

WHAT IS HAMILTONIAN PATH?

Hamiltonian path (or traceable path) is a path in an undirected or directed graph that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a cycle that visits each vertex exactly once. A Hamiltonian path that starts and ends at adjacent vertices can be completed by adding one more edge to form a Hamiltonian cycle, and removing any edge from a Hamiltonian cycle produces a Hamiltonian path.

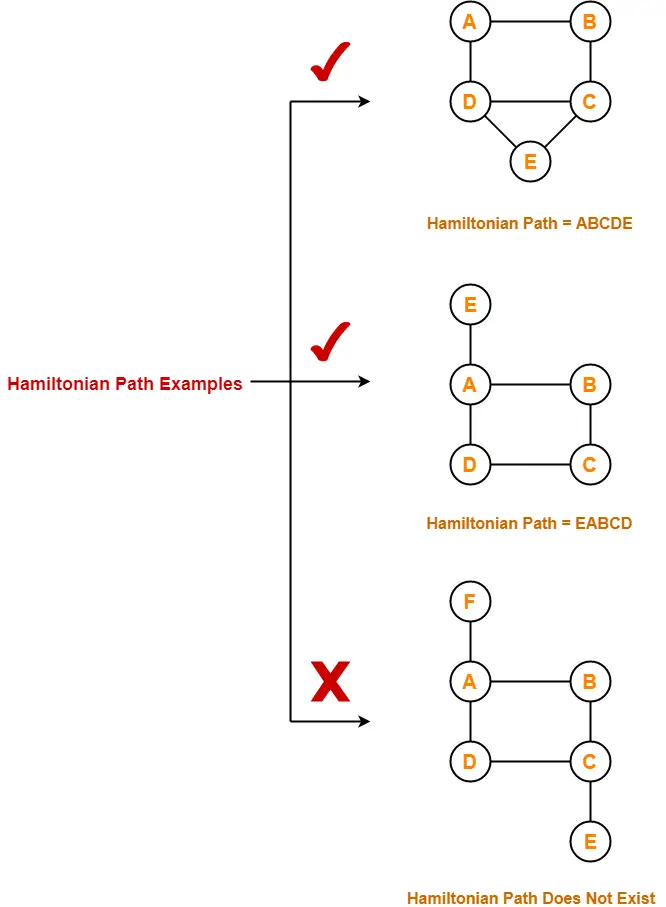
Key Features:

1. Graph Representation: The program allows users to input the graph's vertices and edges. It uses an adjacency matrix or adjacency list data structure to represent the graph.

2. Hamiltonian Path Checker: The core functionality of the program is the Hamiltonian Path checker. It implements various algorithms to check whether a Hamiltonian Path exists in the given graph. This includes backtracking, exhaustive search, or potentially more optimized approaches.

3. Path Visualization: When a Hamiltonian Path is found, the program displays the path to the user. This feature helps users visualize the order in which vertices are visited in the Hamiltonian Path.

4. Error Handling: The program includes robust error handling to deal with scenarios where the input graph is not valid (e.g., not connected) or where a Hamiltonian Path does not exist.



**Code:**

class HamiltonianPathCircuit {

private:

int vertices;

vector<vector<int>> graph;

bool isDirected;

public:

HamiltonianPathCircuit(int v, bool directed): vertices(v), graph(v, vector<int>(v, 0)), isDirected(directed) {}

// Function to add an edge to the graph

void addEdge(int u, int v)

{

graph[u][v] = 1;

if (!isDirected) {

graph[v][u] = 1; // For undirected graphs, add the reverse edge

}

}

// Function to check if it's safe to add vertex v at position pos in the path

bool isSafe(int v, vector<int>& path, int pos)

{

if (graph[path[pos - 1]][v] == 0)

return false;

if (find(path.begin(), path.end(), v) != path.end())

return false;

return true;

}

// Recursive utility function to find Hamiltonian path

bool hamiltonianPathUtil(vector<int>& path, int pos)

{

if (pos == vertices)

{

// All vertices are included in the path

return true;

}

for (int v = 0; v < vertices; ++v)

{

if (isSafe(v, path, pos))

{

path[pos] = v;

// Recur to try the next vertex in the path

if (hamiltonianPathUtil(path, pos + 1))

return true;

path[pos] = -1; // Backtrack

}

}

return false;

}

// Function to find and print Hamiltonian path

bool findHamiltonianPath()

{

vector<int> path(vertices, -1);

path[0] = 0; // Starting from vertex 0

if (!hamiltonianPathUtil(path, 1))

{

cout << "Hamiltonian path does not exist." << endl;

return false;

}

cout << "Hamiltonian path exists. Path: ";

for (int vertex : path)

{

cout << vertex << " ";

}

cout << endl;

return true;

}

// Function to check if the graph has a Hamiltonian circuit

bool isHamiltonianCircuit() {

vector<int> path(vertices, -1);

// Try different starting vertices to find a Hamiltonian circuit

for (int v = 0; v < vertices; ++v)

{

path[0] = v;

// If a Hamiltonian path exists and the last vertex is connected to the starting vertex, it's a circuit

if (hamiltonianPathUtil(path, 1) &&

((isDirected && graph[path[vertices - 1]][path[0]] == 1) || (!isDirected))) {

cout << "Hamiltonian circuit exists. Circuit: ";

for (int vertex : path)

{

cout << vertex << " ";

}

cout << path[0] << endl;

return true;

}

}

cout << "Hamiltonian circuit does not exist." << endl;

return false;

}

};

int main()

{

int vertices, edges;

bool isDirected;

cout << "Enter the number of vertices in the graph: ";

cin >> vertices;

cout << "Is the graph directed? (0 for undirected, 1 for directed): ";

cin >> isDirected;

HamiltonianPathCircuit graph(vertices, isDirected);

cout << "Enter the number of edges: ";

cin >> edges;

cout << "Enter the edges (format: u v):" << endl;

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

int u, v;

cin >> u >> v;

graph.addEdge(u, v);

}

int choice;

do {

cout << "\nMenu:\n1. Find Hamiltonian Path\n2. Find Hamiltonian Circuit\n3. Exit\nEnter your choice: ";

cin >> choice;

switch (choice)

{

case 1:

graph.findHamiltonianPath();

break;

case 2:

graph.isHamiltonianCircuit();

break;

case 3:

cout << "Exiting the program.\n";

break;

default:

cout << "Invalid choice. Please enter a valid option.\n";

}

} while (choice != 3);

return 0;

}

**Approach-**

* 1.The class **HamiltonianPathCircuit** has private members for the number of vertices, the adjacency matrix representation of the graph, and whether the graph is directed or not.
* 2.The class constructor initializes the number of vertices, the adjacency matrix, and the directed flag.
* 3.The **addEdge** method is used to add edges to the graph.
* 4.The **isSafe** method checks whether it's safe to add a vertex to a given position in the path.
* 5. The **hamiltonianPathUtil** method is a recursive utility function to find Hamiltonian paths using backtracking.
* 6.The **findHamiltonianPath** method initializes a path and calls the utility function to find and print a Hamiltonian path.
* 7.The **isHamiltonianCircuit** method attempts to find a Hamiltonian circuit by trying different starting vertices.
* 8.The **main** function takes user input for the number of vertices, whether the graph is directed, the number of edges, and the edges themselves.
* 9.The main menu allows the user to choose between finding a Hamiltonian path, finding a Hamiltonian circuit, or exiting the program.
* 10.The program runs in a loop until the user chooses to exit
* .
* **WORKFLOW-**

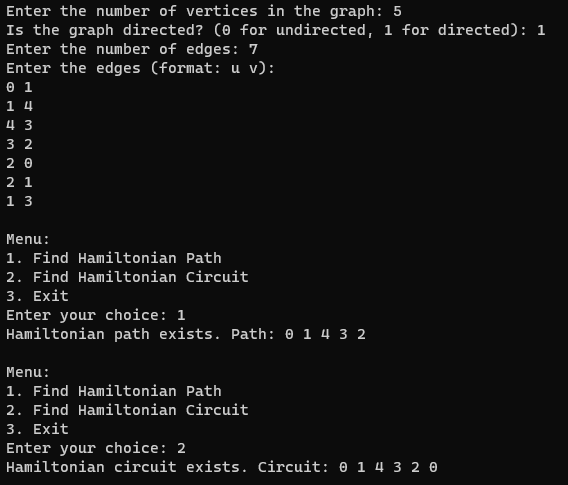
Start of the Program

* Initialize graph with vertices
* y taking input
* Enter whether directed
* or undirected
* Function to add edges to the graph
* User input for edges
* Menu Loop
* User selects operation:
* 1. Find Ham. Path
* 2. Find Ham. Circ

3. Exit

* Perform selected operation
* Display result
* End of the Program

**Illustration:**

* Let a directed graph G with 5 vertices and 7 edges.
* 0 1
* 2 4
* 3
* Hamiltonian Path will be = 0 > 1 > 4 > 3 > 2
* Hamiltonian Circuit will be = 0 > 1 > 4 > 3 > 2 > 0
* Output snapshot from the program:
* 

Here are a few suggestions for future work:

* Input Validation: Add input validation to ensure that the user inputs are within valid ranges and types.
* Documentation: Consider adding comments to explain the purpose and logic of each method for better code readability.
* Error Handling: Implement error handling for invalid inputs or unexpected situations.
* Print Functionality: Instead of printing directly in the class methods, consider returning values and printing in the main function for better separation of concerns.
* Graph Representation: The code uses an adjacency matrix to represent the graph. Depending on the characteristics of the graphs you expect, an adjacency list might be more memory-efficient.
* Create functionality to generate random graphs based on certain properties (E.G., random graphs, complete graphs, sparse graphs).
* Allow user to choose graph generation algorithms and parameters.

**Implementation Details:**

1. Graph Data Structure: The project employs C++ data structures like arrays, vectors, or linked lists to represent the graph. Users can input graph details via a user-friendly interface.

2. Algorithmic Approach: The heart of the project lies in the Hamiltonian Path checking algorithm, which uses C++ functions and logic to determine the presence of a Hamiltonian Path within the graph.

3. Result Display: After analyzing the graph, the program displays the result, indicating whether a Hamiltonian Path exists. If it does, the program also presents the sequence of vertices in the Hamiltonian Path.

**Possible Enhancements:**

- Graph Visualization: Utilize C++ libraries such as Graphviz to create graphical representations of the input graph and Hamiltonian Path.

- File I/O: Implement functionality to save and load graphs from files, making it easier to work with larger datasets.

- Algorithm Optimization: Optimize the Hamiltonian Path checking algorithm to handle larger graphs more efficiently.

**Conclusion:**

The "Hamiltonian Path Finder" project provides a valuable opportunity to delve into graph theory, algorithm design, and data structures using the C++ programming language. It aims to create a practical tool for solving the Hamiltonian Path Problem while offering room for customization and extension based on the user's preferences and requirements.