Maximum Matching by Augmenting Path Algorithm

Team Members:

- Rana Wahaj Ahmed 21k-3281
- Muhammad Hassan 21k-4577
- Ibad ur Rehman 21k-4652

Introduction:

In this project, the augmenting path algorithm has been implemented in both C++ and Python to address the challenges of finding maximum matching in a graph. The transition to Python allows for a more intuitive and visual representation of the graph, overcoming the limitations of console-based visualization. By seamlessly combining algorithmic efficiency with graphical clarity, this project aims to provide a comprehensive solution to optimization problems involving resource allocation and task assignment. The unique representation of maximum matching edges in the visual output enhances the project's impact and accessibility, demonstrating the practical significance of the augmenting path algorithm in real-world scenarios.

Why this Problem?

- Optimizing Resource Allocation:
  Explore the augmenting path algorithm to enhance resource allocation and task assignment efficiency.

- Streamlining Network Flows:
  Address network flow challenges by leveraging the augmenting path algorithm for improved resource movement.

- Bipartite Graphs in Action:
  Investigate practical applications of maximum matching in bipartite graphs, focusing on real-world scenarios.

- Efficiency in Large-scale Graphs:
  Assess the augmenting path algorithm's suitability for handling large-scale graphs, emphasizing computational efficiency.

- Contributions to Graph Theory:
  Contribute insights to graph theory and combinatorial optimization by studying the augmenting path algorithm's role in finding maximum matching.

Optimization:

In an effort to enhance the efficiency of the augmenting path algorithm, we implemented an optimized version that employs a Depth-First Search (DFS) approach to identify augmenting paths. By integrating DFS into the algorithm, we streamline the path-finding process, improving the overall runtime of the

algorithm. This modification aligns with our commitment to algorithmic optimization, ensuring a more robust and scalable solution for

Why DFS:

Used DFS to find augmenting paths in the bipartite graph. It explores unmatched vertices in set X, recursively searching for alternating paths to unmatched vertices in set Y. Augmenting paths are used to update the matching, maximizing the number of edges in the bipartite graph. The overall algorithm employs DFS to iteratively improve the matching until no more augmenting paths can be found.

Solution And Implementation:

The C++ implementation employs the augmenting path algorithm to find the maximum matching in a bipartite graph. The program allows user-defined or hard-coded input, showcasing the algorithm's flexibility. Utilizing Depth-First Search, the algorithm efficiently identifies augmenting paths for optimal matching. The output includes the original bipartite graph, the maximum matching, and the corresponding edges. The concise yet comprehensive code enables a clear understanding of the algorithm's application in solving bipartite graph matching problems.

- Code:

```cpp
#include <iostream>
#include <cstring>

using namespace std;

const int MAX = 100; // Maximum number of vertices

class BipartiteGraph {
public:
    int X, Y; // Number of vertices in sets X and Y
    int edges[MAX][MAX]; // Adjacency matrix to represent the graph
    int matchX[MAX], matchY[MAX]; // Arrays to store the matching
    bool visitedX[MAX], visitedY[MAX]; // Arrays to track whether a vertex is already visited
in sets X and Y

    // Default constructor
    BipartiteGraph() {
    }

    // Parameterized constructor
    BipartiteGraph(int x, int y) : X(x), Y(y) {
        // Initialize arrays using loops
        for (int i = 0; i < MAX; ++i) {
            for (int j = 0; j < MAX; ++j) {
                edges[i][j] = 0;
            }
            matchX[i] = matchY[i] = -1;
            visitedX[i] = visitedY[i] = false;
        }
    }
```

```
    BipartiteGraph(const BipartiteGraph& other) {
        X = other.X;
        Y = other.Y;
        for (int i = 0; i < MAX; ++i) {
            for (int j = 0; j < MAX; ++j) {
                edges[i][j] = other.edges[i][j];
            }
            matchX[i] = other.matchX[i];
            matchY[i] = other.matchY[i];
            visitedX[i] = other.visitedX[i];
            visitedY[i] = other.visitedY[i];
        }
    }

    // Function to add an edge to the graph
    void addEdge(int x, int y) {
        // Avoid self-loops in the matching
        if (x != y) {
            edges[x][y] = 1;
        }
    }

    // Depth-First Search to find augmenting paths
    bool dfs(int x, bool visitedY[]) {
        for (int y = 0; y < Y; ++y) {
            if (edges[x][y] && !visitedY[y]) {
                visitedY[y] = true;
                if (matchY[y] == -1 || (visitedX[matchY[y]] == false && dfs(matchY[y],
visitedY))) {
                    matchX[x] = y;
                    matchY[y] = x;
                    return true;
                }
            }
        }
        return false;
    }
```

```
    // Augment the matching along a path
    void augmentPath(int x) {
        bool visitedY[MAX];  // Create a local array for visitedY
        for (int i = 0; i < MAX; ++i) {
            visitedY[i] = false;
        }

        if (dfs(x, visitedY)) {
            // Augment the matching
            int y = matchX[x];
            while (y != -1) {
                int nextX = matchY[y];
                matchX[nextX] = -1;
                matchY[y] = -1;
                y = matchX[nextX];
            }
        }
    }
```

```
    // Find the maximum matching using augmenting path algorithm
    int maxMatching() {
        int result = 0;
        for (int i = 0; i < MAX; ++i) {
            visitedX[i] = false;
            visitedY[i] = false;
        }

        for (int x = 0; x < X; ++x) {
            if (!visitedX[x] && matchX[x] == -1) {
                bool visitedY[MAX];  // Create a local array for visitedY
                for (int i = 0; i < MAX; ++i) {
                    visitedY[i] = false;
                }

                if (dfs(x, visitedY)) {
                    ++result;
                    visitedX[x] = true;
                }
            }
        }
        return result;
    }

    // Print the edges in the maximum matching
    void printMaximumMatchingEdges() {
        cout << "Edges in Maximum Matching:" << endl;
        for (int x = 0; x < X; ++x) {
            if (matchX[x] != -1) {
                cout << "(" << x << ", " << matchX[x] << ")" << endl;
            }
        }
    }
```

```
    // Run augmenting path algorithm for all unmatched vertices
    void augmentingPathAlgorithm() {
        for (int x = 0; x < X; ++x) {
            if (!visitedX[x] && matchX[x] == -1) {
                augmentPath(x);
            }
        }
    }

    // Print the original bipartite graph
    void printGraph() {
        cout << "Bipartite Graph (Edges):" << endl;
        for (int x = 0; x < X; ++x) {
            for (int y = 0; y < Y; ++y) {
                if (edges[x][y] == 1) {
                    cout << "(" << x << ", " << y << ")" << endl;
                }
            }
        }
    }
};
```

```
BipartiteGraph& userInputGraph() {
    cout << "\nEnter Vertices In X Graph: ";
    int x;
    cin >> x;
    cout << "\nEnter Vertices In Y Graph: ";
    int y;
    cin >> y;

    // Create bipartite graph
    BipartiteGraph graph(x, y);
    return graph;
}
```

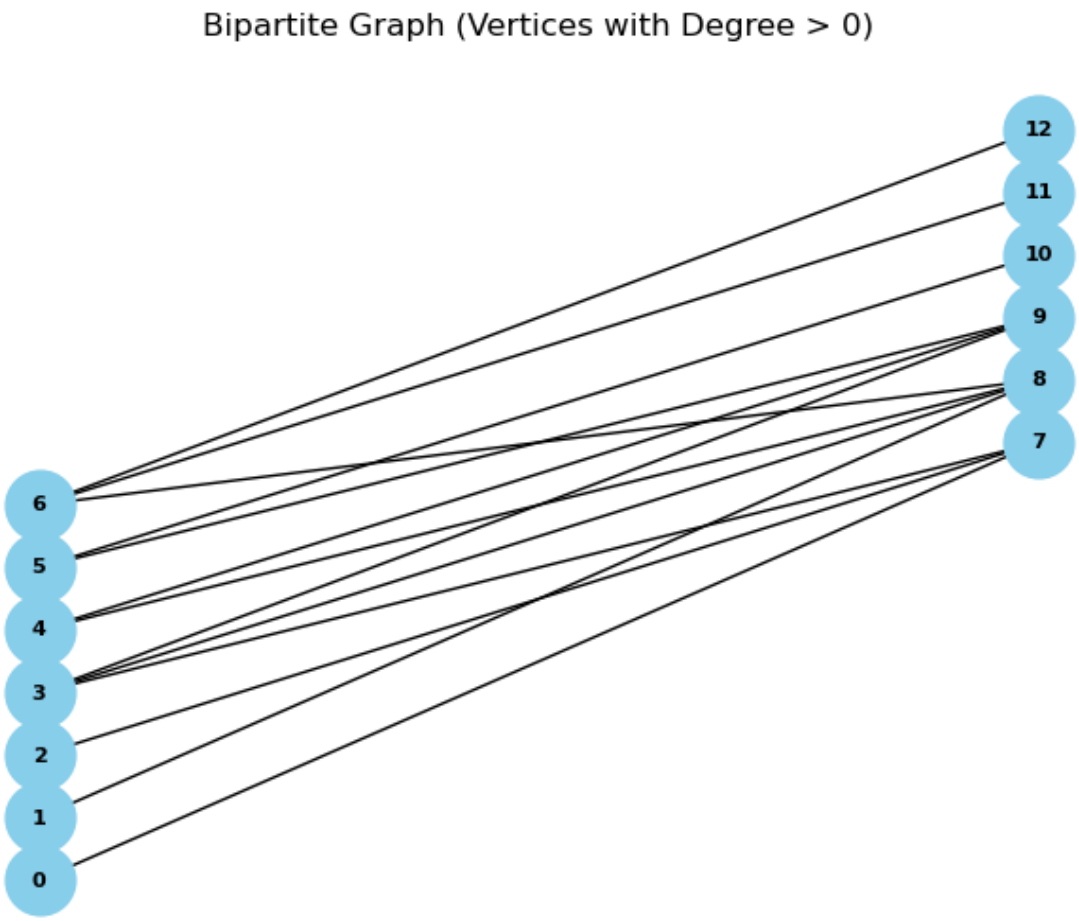
```
int main() {
    char ch;
    BipartiteGraph graph;
  here:
    cout << "\nEnter Your Choice:\n\t1. User-Input\n\t2. Hard-Coded Example";
    cin >> ch;

    if (ch == '1') {
        graph = userInputGraph();
        cout << "\nHow many Edges in Your graph?";
        int x;
        cin >> x;

        if (x > 0) {
            for (int i = 0; i < x; i++) {
                cout << "\nEnter Source from X (0...X-1 vertices) :";
                int s;
                cin >> s;
                cout << "\nEnter Target from Y (0...Y-1 vertices) :";
                int t;
                cin >> t;
                graph.addEdge(s, t);
            }
        }
    } else if (ch == '2') {
        // Hard-coded example
        int X = 13; // Number of vertices in set X
        int Y = 13; // Number of vertices in set Y
        BipartiteGraph graph1(X, Y);
        graph = graph1;
```

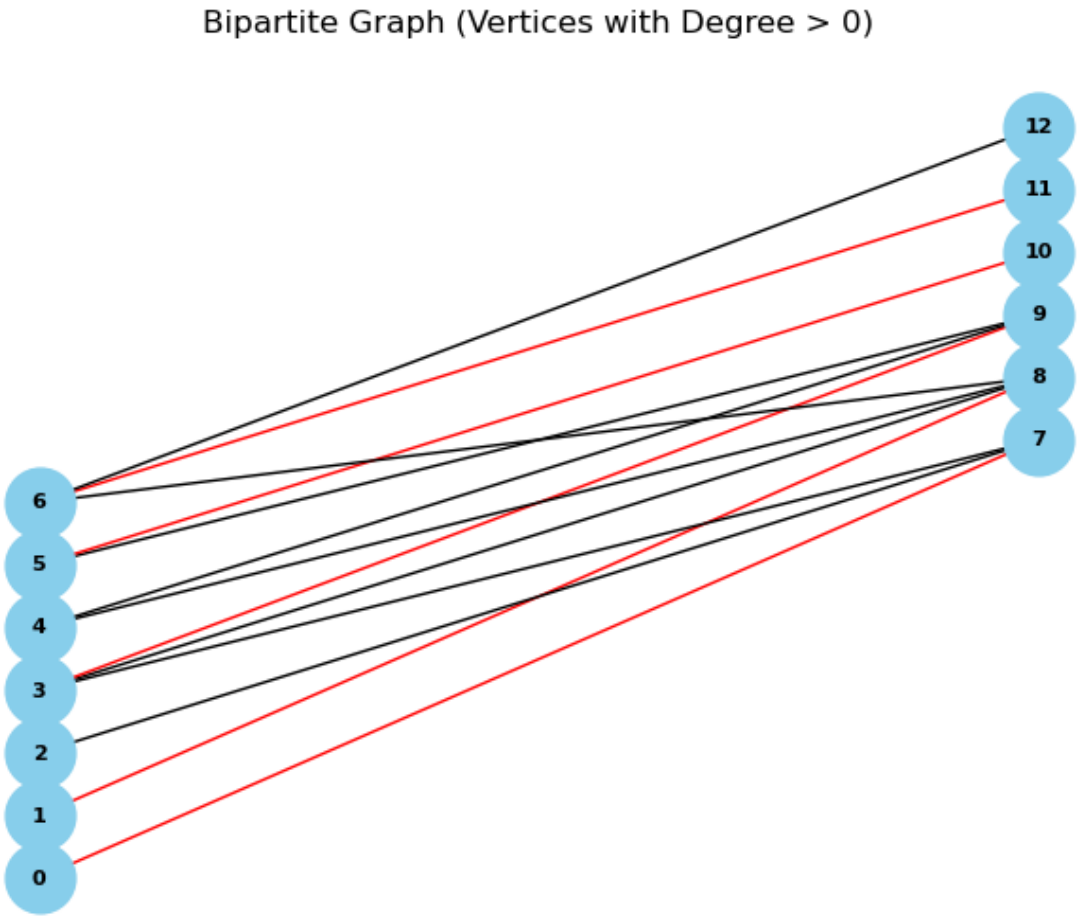
```
        // Add edges to the bipartite graph
        graph.addEdge(0, 7);
        graph.addEdge(1, 8);
        graph.addEdge(2, 7);
        graph.addEdge(3, 7);
        graph.addEdge(3, 8);
        graph.addEdge(3, 9);
        graph.addEdge(4, 8);
        graph.addEdge(4, 9);
        graph.addEdge(5, 9);
        graph.addEdge(5, 10);
        graph.addEdge(6, 8);
        graph.addEdge(6, 11);
        graph.addEdge(6, 12);
    }
    else{
      cout<<"\nWrong Input....";
      goto here;
  }

    graph.printGraph();

    // Run the augmenting path algorithm
    graph.augmentingPathAlgorithm();
    int maxMatching = graph.maxMatching();
    cout << "Maximum Matching after Augmenting Path Algorithm: " << maxMatching << endl;

    // Print the edges in the maximum matching
    graph.printMaximumMatchingEdges();

    return 0;
}
```

Output:

```
Enter Your Choice:
        1. User-Input
        2. Hard-Coded Example2
Bipartite Graph (Edges):
(0, 7)
(1, 8)
(2, 7)
(3, 7)
(3, 8)
(3, 9)
(4, 8)
(4, 9)
(5, 9)
(5, 10)
(6, 8)
(6, 11)
(6, 12)
Maximum Matching after Augmenting Path Algorithm: 5
Edges in Maximum Matching:
(0, 7)
(1, 8)
(3, 9)
(5, 10)
(6, 11)
```

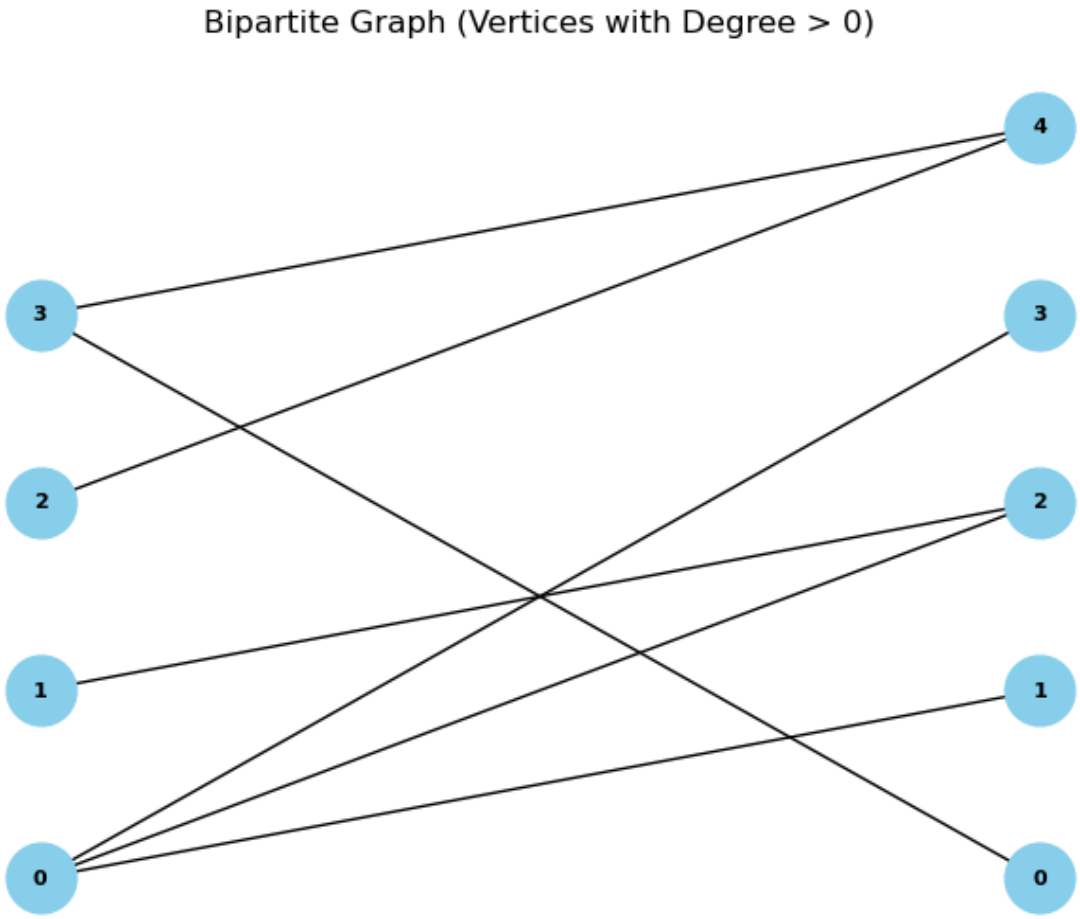
- **Visually by python of Hard-Coded Graph:**

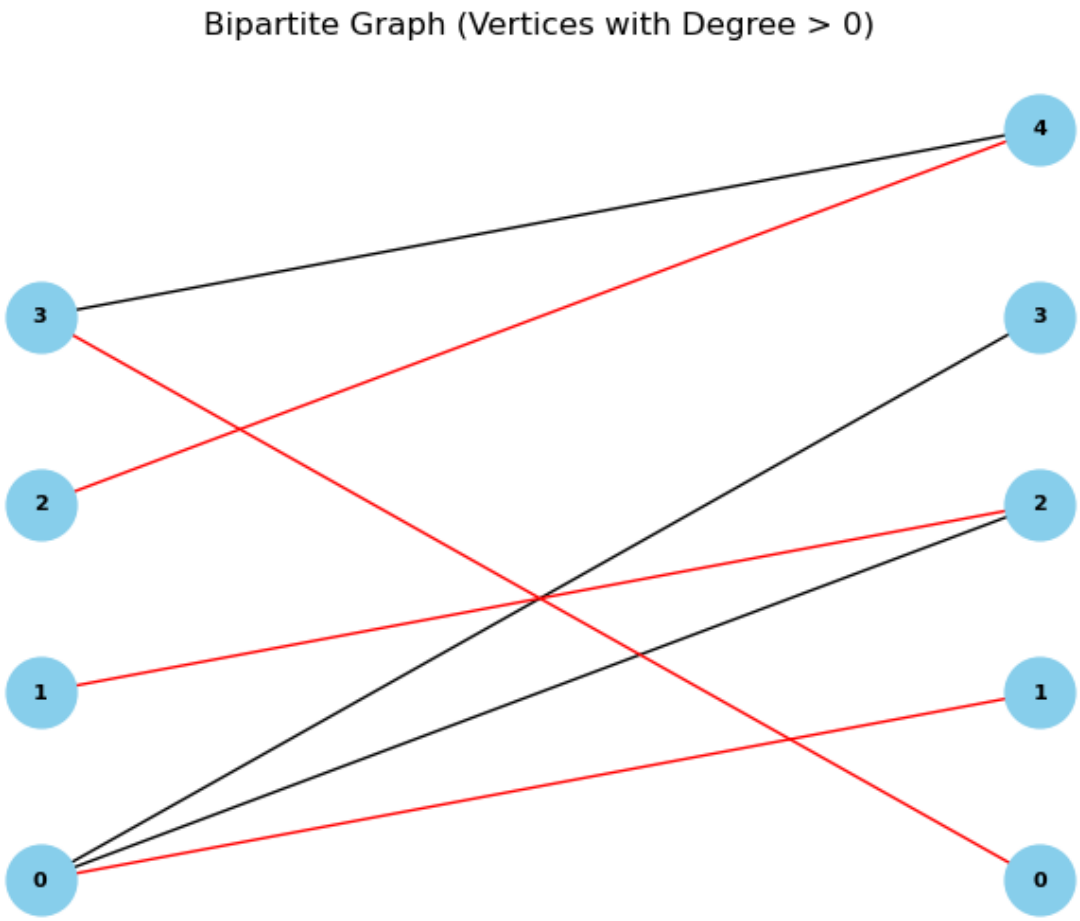



Bipartite Graph (Vertices with Degree > 0)
12
11
10
9
8
7
6
5
4
3
2
1
0

- **Visually by python of User-Input Graph:**

```
Enter Your Choice:
        1. User-Input
        2. Hard-Coded Example: 1

Enter Vertices In X Graph: 4

Enter Vertices In Y Graph: 5

How many Edges in Your graph? 7

Enter Source from X (0...X-1 vertices): 0

Enter Target from Y (0...Y-1 vertices): 1

Enter Source from X (0...X-1 vertices): 0

Enter Target from Y (0...Y-1 vertices): 2

Enter Target from Y (0...Y-1 vertices): 2

Enter Source from X (0...X-1 vertices): 1

Enter Target from Y (0...Y-1 vertices): 2

Enter Source from X (0...X-1 vertices): 3

Enter Target from Y (0...Y-1 vertices): 4

Enter Source from X (0...X-1 vertices): 3

Enter Target from Y (0...Y-1 vertices): 0

Enter Source from X (0...X-1 vertices): 2

Enter Target from Y (0...Y-1 vertices): 4

Enter Source from X (0...X-1 vertices): 0

Enter Target from Y (0...Y-1 vertices): 3
```


Bipartite Graph (Vertices with Degree > 0)
3
2
1
0
4
3
2
1
0



Complexity Analysis:

- **<u>Space Complexity:</u>**

-<u>Adjacency Matrix:</u>

The space complexity of the adjacency matrix is O(X * Y), where X and Y are the number of vertices in sets X and Y. This matrix represents the bipartite graph and is used to store information about the edges.

-<u>Matching Arrays:</u>

The arrays `matchX`, `matchY`, `visitedX`, and `visitedY` contribute O(MAX) space complexity.

- Local Variables:

Local variables within functions, such as the `visitedY` array in the `augmentPath` and `maxMatching` functions, incur additional space during function execution.

- **Time Complexity:**

- Initialization:

The initialization of arrays in the constructor has a time complexity of O(MAX^2), as it involves nested loops iterating over MAX.

- DFS Algorithm:

The Depth-First Search (DFS) algorithm used to find augmenting paths has a time complexity of O(X * Y), where X and Y are the number of vertices in sets X and Y.

-Augmenting Path Algorithm:

The `augmentingPathAlgorithm` function runs the DFS algorithm for each unmatched vertex, resulting in a total time complexity of O(X * Y^2).

- Maximum Matching:

The `maxMatching` function iterates over all unmatched vertices and runs DFS, contributing to the overall time complexity of O(X * Y^2).

- Printing:

The printing functions, such as `printGraph` and `printMaximumMatchingEdges`, have a time complexity proportional to the number of edges in the graph.

- **Overall:**

The overall space complexity is dominated by the adjacency matrix, while the time complexity is primarily influenced by the DFS-based augmenting path algorithm. The algorithm's efficiency in finding augmenting paths and updating the matching arrays contributes to the overall performance of the code

Conclusion:

In conclusion, this project implemented the augmenting path algorithm in C++ and Python, emphasizing efficient resource allocation and task assignment. The transition to Python improved visual representation, overcoming console limitations. Depth-First Search (DFS) optimization enhanced path-finding efficiency, streamlining the algorithm. Real-world applications included network flow optimization and bipartite graph scenarios. The project's contributions extend to graph theory and combinatorial optimization. Space complexity is dominated by the adjacency matrix, while time complexity is influenced by DFS-based path finding. The optimized algorithm demonstrates scalability and computational efficiency in handling large-scale graphs. Practical significance was showcased through a visual output of maximum matching edges. Overall, this project successfully blends algorithmic efficiency with graphical clarity, providing a comprehensive solution to optimization challenges in diverse applications.