

Vertex Vertusa

INTRODUCTION

1.1. Need Identification

This project focuses on implementing algorithms to find the **Maximal Independent Set (MIS)** of vertices in a graph. The **Maximal Independent Set** is a subset of vertices such that no two vertices in the set are adjacent, and no additional vertices can be added without breaking this property. Given the NP-hard nature of the problem, the project explores both a brute-force approach, which checks all possible subsets of vertices to guarantee the correct solution, and an approximation method, which provides a faster, near-optimal solution. The project will implement these algorithms in Java and analyze their performance on different types of graphs, aiming to balance accuracy and computational efficiency in the solution.

1.2. Identification of Problem

The problem addressed in this project is the challenge of finding the Maximal Independent Set (MIS) in a graph. In a graph, an independent set is a subset of vertices where no two vertices are adjacent, and the MIS is the largest such set that cannot be expanded by adding more vertices.

The MIS problem is classified as NP-hard, meaning it is computationally intractable for large graphs using traditional methods. This project aims to implement and compare two approaches:

brute-force method, which guarantees the exact solution but is inefficient for large graphs, and an approximation algorithm, which provides a faster, near-optimal solution.

The goal is to explore efficient ways to solve this graph-theoretical problem, which has practical applications in areas like network design, scheduling, and resource allocation.

1.3. Identification of Tasks

The tasks identified for this project on finding the **Maximal Independent Set (MIS)** in a graph are as follows:

1. **Graph Representation:** Design and implement a suitable data structure (e.g., adjacency list) to represent a graph in Java.
2. **Brute-Force Algorithm Implementation:** Develop the brute-force algorithm that generates all possible subsets of vertices and checks for independence, identifying the maximal independent set.

3. **Approximation Algorithm Implementation:** Explore and implement an efficient approximation algorithm to find a near-optimal maximal independent set for large graphs.
4. **Performance Analysis:** Test the algorithms on various types of graphs (e.g., sparse, dense, random) and compare their performance in terms of time complexity, space complexity, and solution accuracy.
5. **Optimization and Refinement:** Optimize the implementation for better performance, reducing runtime without compromising solution accuracy, especially in the approximation algorithm.
6. **Documentation:** Document the entire process, including the code, methodology, results, and comparisons, to provide a comprehensive project report.
7. **Presentation and Conclusion:** Summarize findings, highlight key insights, and present the project's results, conclusions, and potential future work or improvement.

Implementation

Brute Force Approach for MIS(Minimum Independent Set)

The brute force approach for finding the maximal independent set (MIS) involves checking every possible subset of vertices and determining which of these subsets forms an independent set. Among all valid independent sets, we select the largest one.

Steps for Brute Force Algorithm:

1. Generate all possible subsets of vertices.
2. Check each subset to see if it forms an independent set (i.e., no two vertices in the subset are connected by an edge).
3. Keep track of the largest independent set.
4. Return the maximal independent set.

Program:

```
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set; class
Graph { private int
vertices;
private List<List<Integer>>> adjList;
```

// Constructor to initialize the graph

```
public Graph(int vertices) {
    this.vertices = vertices; adjList = new
    ArrayList<>(); for (int i = 0; i <
    vertices; i++) { adjList.add(new
    ArrayList<>());
}
}
```

// Function to add an edge between two vertices

```
public void addEdge(int u, int v) {
    adjList.get(u).add(v); adjList.get(v).add(u);
}
```

// Brute-force function to find the largest independent set public

```
Set<Integer> findMaximalIndependentSet() {
    Set<Integer> maxSet = new HashSet<>(); // Stores the largest independent set
```

```
// Iterate through all subsets of vertices for (int i = 0; i < (1 << vertices);
i++) { // 1 << vertices means 2^vertices Set<Integer> currentSet = new
HashSet<>(); // Current subset of vertices boolean isIndependent = true; //
To check if the current subset is independent
```

// Iterate through each vertex to see if it is in the current subset for

```
(int v = 0; v < vertices; v++) {
    if ((i & (1 << v)) != 0) { // Check if vertex 'v' is in the subset
```

// Check if this vertex forms an edge with any other vertex in the subset

```
for (int neighbor : currentSet) { if (adjList.get(v).contains(neighbor)) {
isIndependent = false; // If connected, it's not independent
break; } }
```

```
if (isIndependent) {
    currentSet.add(v); // Add vertex to the current independent set
} else { break; // No need to check further if the set is not
independent
} }
}
```

// If the current subset is independent and larger than the maxSet, update maxSet

```
if (isIndependent && currentSet.size() > maxSet.size()) {
    maxSet = new HashSet<>(currentSet);
} }
```

```

return maxSet; // Return the largest independent set found
}
}

public class MaximalIndependentSetBruteForce { public
static void main(String[] args) {
// Create a graph with 5 vertices (example graph) Graph
graph = new Graph(5);

// Add edges
graph.addEdge(0, 1);
graph.addEdge(0, 2);
graph.addEdge(1, 3);
graph.addEdge(2, 4); //
Find and print the
Maximal Independent
Set using brute-force
Set<Integer> mis =
graph.findMaximalIndepe
ndentSet();
System.out.println("Maximal Independent Set (Brute Force): " + mis);
}
}

```

Explanation:

```

public Graph(int vertices) {
    this.vertices = vertices;
    adjList = new ArrayList<>();
    for (int i = 0; i < vertices; i++) {
        adjList.add(new ArrayList<>());
    }
}

```

Figure 1

Constructor: Initializes the graph with a specified number of vertices.

- The adjList is initialized as an empty list of lists. Each vertex gets its own list to hold the vertices it connects to.
- The for loop creates an empty list for each vertex, which will later hold its adjacent vertices.

```
public void addEdge(int u, int v) {
    adjList.get(u).add(v);
    adjList.get(v).add(u);
}
```

Figure 2

addEdge method: Adds an edge (connection) between two vertices **u** and **v**.

- Adds vertex **v** to the **adjacency list of vertex u**.
- Adds vertex **u** to the **adjacency list of vertex v**. • This creates an undirected edge between **u** and **v**, meaning both vertices are connected.

```
public Set<Integer> findMaximalIndependentSet() {
    Set<Integer> maxSet = new HashSet<>();
```

Figure 3

findMaximalIndependentSet method: This method will find the maximal independent set in the graph. • A **maxSet** variable is initialized to store the largest independent set found.

```
int totalSubsets = 1 << vertices; // 2^vertices
for (int i = 0; i < totalSubsets; i++) {
    Set<Integer> currentSet = new HashSet<>();
    boolean isIndependent = true;

    // Check the subset corresponding to the binary representation of i
    for (int v = 0; v < vertices; v++) {
        if ((i & (1 << v)) != 0) { // Check if vertex v is in the subset
            for (int u : currentSet) {
                if (adjMatrix[u][v]) { // Check adjacency
                    isIndependent = false;
                    break;
                }
            }
        }
    }
}
```

Figure 4

This loop iterates through all possible **subsets of vertices**.

- **(1 << vertices)** is the same as 2^{vertices} , representing the total number of subsets (since each vertex can either be included or not).
- Each subset is represented as a binary number, where each bit indicates whether a vertex is **included in the subset (1) or not (0)**.
 - **currentSet:** Stores the current subset of vertices being checked.

- **isIndependent:** A boolean flag used to track whether the current subset is an independent set (i.e., no two vertices are connected by an edge).

This loop iterates over each vertex v and checks if it is included in the current subset.

- $(i \& (1 \ll v)) \neq 0$ checks if the v -th bit of i is set to 1, meaning vertex v is included in the subset represented by i .
- This checks if the current vertex v forms an edge with any other vertex already in the current set (`currentSet`).
- The loop goes through each vertex already in `currentSet` and checks if it is connected to vertex v using the adjacency list.
- If they are connected, `isIndependent` is set to false, and the loop breaks since the current subset is not independent.

If the subset remains independent after checking, vertex v is added to the `currentSet`.

- Otherwise, the loop breaks, and we move on to check the next subset

```
if (isIndependent && currentSet.size() > maxSet.size()) {  
    maxSet = new HashSet<>(currentSet);  
}
```

Figure 5

After checking the entire subset, if it's independent and larger than the current `maxSet`, we update `maxSet` to be the new largest independent set.

```
public class MaximalIndependentSetBruteForce {  
    public static void main(String[] args) {  
        Graph graph = new Graph(5);  
  
        graph.addEdge(0, 1); // A -- B  
        graph.addEdge(0, 2); // A -- C  
        graph.addEdge(1, 3); // B -- D  
        graph.addEdge(2, 4); // C -- E
```

Figure 6

Adding edges: These lines add edges between specific vertices, forming an undirected graph:

- Vertex 0 (A) is connected to vertex 1 (B) and vertex 2 (C).
- Vertex 1 (B) is connected to vertex 3 (D).
- Vertex 2 (C) is connected to vertex 4 (E).

Approximation method (Greedy Approach)

Since finding the maximal independent set using brute force is computationally expensive, approximation algorithms are often used, especially for large graphs. Approximation algorithms aim to find an independent set that is close to maximal but do so more efficiently.

Greedy Approximation Algorithm:

One of the simplest approximation algorithms for finding an independent set is the greedy algorithm.

Algorithm (Greedy Approximation):

1. Initialize an empty set: Start with an empty set SSS to store the independent set.
2. Sort vertices by degree: Sort all vertices based on some heuristic (commonly by degree, i.e., the number of edges connected to the vertex).
3. Iterate over vertices: For each vertex v :
 - If v is not adjacent to any vertex already in the independent set SSS, add v to SSS.
4. Continue until no more vertices can be added.
5. Return the set: The set SSS is the approximate independent set.

Example:

- Start with an empty set $S = \emptyset$.
- Sort vertices by degree (or use a random order).
- Add the first vertex to SSS, and then skip any vertices adjacent to this vertex.
- Continue until all vertices are considered.

Program

```
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

class Graph { private
int vertices;
private List<List<Integer>> adjList;

// Constructor to initialize the graph
public Graph(int vertices) {
this.vertices = vertices; adjList = new
ArrayList<>(); for (int i = 0; i <
vertices; i++) { adjList.add(new
ArrayList<>());
```

```
}
}
```

// Function to add an edge between two vertices

```
public void addEdge(int u, int v) {
    adjList.get(u).add(v); adjList.get(v).add(u);
}
```

// Function to find a maximal independent set using a greedy approach

```
public Set<Integer> findMaximalIndependentSet() {
    Set<Integer> mis = new HashSet<>(); // Stores the maximal independent set
    boolean[] selected = new boolean[vertices]; // Tracks if a vertex is selected
```

// Iterate through each vertex for

```
(int v = 0; v < vertices; v++) {
```

// If the vertex is not selected and none of its neighbors are selected

```
if (!selected[v]) {
```

```
    mis.add(v); // Add the vertex to the independent set
    selected[v]
```

```
= true; // Mark it as selected
```

// Mark all adjacent vertices as selected

```
for (int neighbor : adjList.get(v)) {
```

```
    selected[neighbor] = true;
```

```
}
```

```
} } return
```

```
mis; }
```

```
}
```

```
public class MaximalIndependentSet { public
```

```
    static void main(String[] args) {
```

// Create a graph with 5 vertices (example graph)

```
    Graph graph = new Graph(5);
```

// Add edges (example: A=0, B=1, C=2, D=3, E=4)

```
    graph.addEdge(0, 1); // A -- B
    graph.addEdge(0, 2);
```

```
    // A -- C
    graph.addEdge(1, 3); // B -- D
```

```
    graph.addEdge(2, 4); // C -- E
```

// Find and print the Maximal Independent Set

```
    Set<Integer> mis = graph.findMaximalIndependentSet();
```

```
    System.out.println("Maximal Independent Set: " + mis);
```



```
}  
}
```

Explanation

```
class Graph {  
    private int vertices;  
    private List<List<Integer>> adjList;
```

Figure 7

vertices: This variable stores the number of vertices (nodes) in the graph.

adjList: An **adjacency list** representation of the graph, where each vertex has a list of its adjacent vertices. This is a common way to represent graphs efficiently in terms of memory.

```
public Graph(int vertices) {  
    this.vertices = vertices;  
    adjList = new ArrayList<>();  
    for (int i = 0; i < vertices; i++) {  
        adjList.add(new ArrayList<>());  
    }  
}
```

Figure 8

Purpose: Initializes the graph with the given number of vertices. It also creates an empty adjacency list for each vertex to store its neighbors.

Explanation: When the graph is created, it sets the number of vertices and creates a list for each vertex in the adjacency list.

```
public void addEdge(int u, int v) {  
    adjList.get(u).add(v);  
    adjList.get(v).add(u);  
}
```

Figure 9

Purpose: Adds an edge between two vertices, u and v, in an undirected graph.

Explanation: It updates the adjacency list by adding each vertex to the other's list of neighbors. Since this is an **undirected graph**, the edge is added in both directions: $u \rightarrow v$ and $v \rightarrow u$.

```

public Set<Integer> findMaximalIndependentSet() {
    Set<Integer> mis = new HashSet<>(); // Stores the maximal independent set
    boolean[] selected = new boolean[vertices]; // Tracks if a vertex is selected

    // Iterate through each vertex
    for (int v = 0; v < vertices; v++) {
        // If the vertex is not selected and none of its neighbors are selected
        if (!selected[v]) {
            mis.add(v); // Add the vertex to the independent set
            selected[v] = true; // Mark it as selected

            // Mark all adjacent vertices as selected
            for (int neighbor : adjList.get(v)) {
                selected[neighbor] = true;
            }
        }
    }
    return mis;
}

```

Figure 10

This method returns a set of integers representing the Maximal Independent Set of a graph. **mis**: A HashSet to store the vertices that are part of the Maximal Independent Set. **selected**: A boolean array that tracks whether a vertex or any of its neighbors has been selected. Initially, all vertices are unselected (false).

- Iterates over each vertex v from 0 to the total number of vertices (vertices).
- If the vertex v has not been selected yet, it means that neither the vertex nor its neighbors have been included in the MIS.
- Mark all the adjacent vertices (neighbors of v) as selected because if vertex v is part of the MIS, none of its neighbors can be part of it.
- Finally, return the `mis` set which contains the vertices forming the Maximal Independent Set.

RESULTS ANALYSIS AND VALIDATION

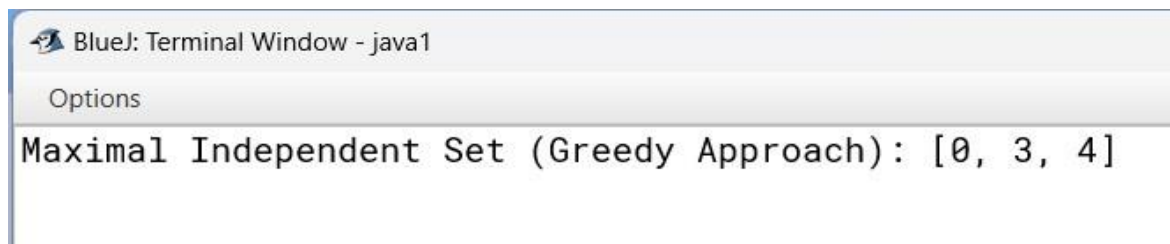
Outputs

1. Brute Force Approach



```
BlueJ: Terminal Window - java1
Options
Maximal Independent Set (Brute Force): [0, 3, 4]
```

2. Approximation Method (Greedy Approach)



```
BlueJ: Terminal Window - java1
Options
Maximal Independent Set (Greedy Approach): [0, 3, 4]
```

Figure 2

Complexities

1. Brute Force Approach

1. Time Complexity:

- The brute force algorithm explores all possible subsets of vertices in a graph.
- For a graph with n vertices, the number of subsets is 2^n .
- For each subset, it checks if it is independent by examining all pairs of vertices in that subset, which takes $O(n^2)$ time in the worst case.
- Therefore, the overall **time complexity is $O(n^2 \cdot 2^n)$** .

2. Space Complexity:

- The space complexity primarily depends on the storage of the adjacency list, which takes $O(n+m)$, where m is the number of edges. ○ Additionally, the current subset and the maximum independent set can also take space proportional to n . ○ Hence, **the overall space complexity is $O(n+m)$.**

Summary

- **Time Complexity:** $O(n^{2.2^n})$. •
- Space Complexity:** $O(n+m)$.

2. Approximation Method(Greedy Approach)

Time Complexity of Greedy Approach

The time complexity of the greedy approach depends on two factors:

1. **Iteration through vertices:** The algorithm iterates over all vertices once, giving a time complexity of $O(V)$ where V is the number of vertices.
2. **Checking neighbours:** For each vertex added to the independent set, we mark its neighbors, and this takes $O(D)$ time per vertex where D is the degree (number of neighbors) of the vertex.

Since the sum of the degrees of all vertices in a graph is $2E$ (where E is the number of edges), iterating through all neighbours gives a time complexity of $O(V + E)$.

Thus, the time complexity of the greedy algorithm is:

$O(V + E)$.

This is much more efficient compared to the brute-force approach, which has an exponential time complexity of $O(2^V * V^2)$.

Space Complexity of Greedy Approach

The space complexity of the greedy approach is determined by:

1. **Adjacency list:** Storing the graph's edges requires $O(V + E)$ space for the adjacency list representation.
2. **Selected array:** An additional $O(V)$ space is used to keep track of which vertices and their neighbors are excluded.

Thus, the space complexity is: **$O(V + E)$.**

Summary

- **Time Complexity: $O(V + E)$** where V is the number of vertices, and E is the number of edges.
- **Space Complexity: $O(V + E)$** mainly due to the adjacency list and additional data structures.

The greedy approach is significantly faster than the brute-force method, making it practical for large graphs, though it may not always find the maximum independent set, but only a maximal one.

Comparison of the two Approaches

Aspect	Brute-Force Approach	Greedy Approach
Time Complexity	$O(2^V * V^2)$ (Exponential)	$O(V + E)$ (Linear for sparse graphs)
Space Complexity	$O(V + E)$ (Adjacency list, subsets)	$O(V + E)$ (Adjacency list, selected array)
Optimality	Guaranteed to find the maximum independent set	Only finds a maximal independent set (not always the largest)
Performance on Large Graphs	Infeasible due to exponential complexity	Efficient for large graphs
Approach	Exhaustively checks all subsets of vertices	Adds vertices greedily based on local decisions
Use Case	Useful for small graphs where exact solutions are needed	Suitable for large graphs where approximate solutions are acceptable