GRAPHS ALGORITHMS

ALGORITHMS IN JAVA

Sercan Külcü | Algorithms In Java | 10.05.2023

Contents

Introduction

Graph algorithms are a fundamental part of computer science. They are used to solve a wide variety of problems, such as finding the shortest path between two nodes, finding the connected components of a graph, and finding the maximum flow in a network.

A graph is a mathematical structure that consists of a set of nodes and a set of edges. The nodes represent objects, and the edges represent the relationships between those objects. For example, a graph can be used to represent a road network, where the nodes represent cities and the edges represent roads between those cities.

Graph algorithms are used to solve a wide variety of problems. Some of the most common graph algorithms include:

- Breadth-first search (BFS): BFS is an algorithm for traversing a graph. It starts at a given node and explores all of the nodes that are connected to that node before moving on to the next node.
- Depth-first search (DFS): DFS is another algorithm for traversing a graph. It starts at a given node and explores all of the nodes that are connected to that node, but it does so in a depth-first manner, meaning that it explores all of the nodes that are connected to the current node before moving on to the next node.
- Dijkstra's algorithm: Dijkstra's algorithm is an algorithm for finding the shortest path between two nodes in a graph. It works by iteratively adding nodes to a set of nodes that have already been visited, and it keeps track of the shortest path to each node that has been visited.
- Bellman-Ford algorithm: The Bellman-Ford algorithm is another algorithm for finding the shortest path between two nodes in a graph. It works by iteratively updating the distances to each node in the graph, and it guarantees that it will find the shortest path, even if there are negative weights on the edges.

- Prim's algorithm: Prim's algorithm is an algorithm for finding the minimum spanning tree of a graph. A minimum spanning tree is a tree that connects all of the nodes in the graph, and it has the minimum possible total weight.
- Kruskal's algorithm: Kruskal's algorithm is another algorithm for finding the minimum spanning tree of a graph. It works by iteratively adding edges to the tree, and it guarantees that it will find the minimum spanning tree.
- Ford-Fulkerson algorithm: The Ford-Fulkerson algorithm is an algorithm for finding the maximum flow in a network. A network is a special type of graph where the edges have capacities. The maximum flow is the maximum amount of flow that can be sent through the network.
- Edmonds-Karp algorithm: The Edmonds-Karp algorithm is another algorithm for finding the maximum flow in a network. It works by iteratively adding paths to the network, and it guarantees that it will find the maximum flow.

Graph algorithms are a powerful tool for solving a wide variety of problems. They are used in a variety of applications, such as routing, scheduling, and network analysis.

Graph traversal algorithms

BREADTH-FIRST SEARCH (BFS)

Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key'), and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

The algorithm starts with a queue of nodes to be visited, which initially contains only the start node. While the queue is not empty, the algorithm removes the first node from the queue and adds all of its unvisited neighbors to the queue. This process continues until the queue is empty, which indicates that all of the nodes in the graph have been visited.

BFS is a simple and efficient algorithm for traversing graphs. It is typically used to find the shortest path between two nodes in a graph, or to find all of the nodes that are connected to a given node.

Implementation in Java

BFS can be implemented in Java using the following steps:

- Create a queue to store the nodes that have not yet been visited.
- Add the start node to the queue.
- While the queue is not empty:
- Remove the first node from the queue.
- Mark the node as visited.
- Add all of the node's unvisited neighbors to the queue.

Here is an example of how to implement BFS in Java:

public class BreadthFirstSearch {

```
public static void main(String[] args) {
    // Create a graph
    Graph graph = new Graph();

    // Add nodes to the graph
    graph.addNode("A");
    graph.addNode("B");
    graph.addNode("C");
    graph.addNode("D");

    // Add edges to the graph
    graph.addEdge("A", "B");
    graph.addEdge("A", "C");
    graph.addEdge("B", "D");

    // Start at node A
    Node currentNode = graph.getNode("A");

    // Create a queue to store the nodes that have not yet been visited
    Queue<Node> queue = new LinkedList<>();

    // Add the start node to the queue
```

```
queue.add(currentNode);

// While there are nodes to explore
while (!queue.isEmpty()) {
    // Remove the first node from the queue
    currentNode = queue.remove();

    // Print the node
    System.out.println(currentNode.getName());

    // Get the node's neighbors
    List<Node> neighbors = currentNode.getNeighbors();

    // For each neighbor
    for (Node neighbor : neighbors) {
        // If the neighbor has not been visited
        if (!neighbor.isVisited()) {
            // Mark the neighbor as visited
            neighbor.setVisited(true);

            // Add the neighbor to the queue
            queue.add(neighbor);
```

```
            }
        }
    }
  }
}
```

Advantages and disadvantages

BFS has the following advantages:

- It is simple to implement.
- It is efficient for traversing graphs with few edges.
- It can be used to find the shortest path between two nodes in a graph.

BFS has the following disadvantages:

- It can be inefficient for traversing graphs with many edges.
- It does not work well for graphs with cycles.

Breadth-first search is a powerful algorithm for traversing graphs. It is simple to implement and efficient for graphs with few edges. It can be used to find the shortest path between two nodes in a graph.

DEPTH-FIRST SEARCH (DFS)

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

The algorithm starts with a stack of nodes to be visited, which initially contains only the start node. While the stack is not empty, the algorithm removes the top node from the stack and explores all of its unvisited neighbors. If a neighbor has been visited, the algorithm backtracks to the previous node in the stack. This process continues until the stack is empty, which indicates that all of the nodes in the graph have been visited.

DFS is a simple and efficient algorithm for traversing graphs. It is typically used to find all of the nodes that are connected to a given node, or to find the connected components of a graph.

Implementation in Java

DFS can be implemented in Java using the following steps:

- Create a stack to store the nodes that have not yet been visited.
- Push the start node onto the stack.
- While the stack is not empty:
    - Pop the top node from the stack.
    - Mark the node as visited.
    - Add all of the node's unvisited neighbors to the stack.

Here is an example of how to implement DFS in Java:

```
public class DepthFirstSearch {


   public static void main(String[] args) {

      // Create a graph
```

```
Graph graph = new Graph();

// Add nodes to the graph
graph.addNode("A");
graph.addNode("B");
graph.addNode("C");
graph.addNode("D");

// Add edges to the graph
graph.addEdge("A", "B");
graph.addEdge("A", "C");
graph.addEdge("B", "D");

// Start at node A
Node currentNode = graph.getNode("A");

// Create a stack to store the nodes that have not yet been visited
Stack<Node> stack = new Stack<>();

// Push the start node onto the stack
stack.push(currentNode);
```

```
// While there are nodes to explore
while (!stack.isEmpty()) {
    // Pop the top node from the stack
    currentNode = stack.pop();

    // Print the node
    System.out.println(currentNode.getName());

    // Get the node's neighbors
    List<Node> neighbors = currentNode.getNeighbors();

    // For each neighbor
    for (Node neighbor : neighbors) {
        // If the neighbor has not been visited
        if (!neighbor.isVisited()) {
            // Mark the neighbor as visited
            neighbor.setVisited(true);

            // Push the neighbor onto the stack
            stack.push(neighbor);
        }
    }
```

```
        }
    }
}
```

Advantages and disadvantages

DFS has the following advantages:

- It is simple to implement.
- It can be used to find all of the nodes that are connected to a given node.
- It can be used to find the connected components of a graph.

DFS has the following disadvantages:

- It can be inefficient for traversing graphs with many edges.
- It can get stuck in infinite loops if the graph contains cycles.

Depth-first search is a powerful algorithm for traversing graphs. It is simple to implement and efficient for graphs with few edges. It can be used to find all of the nodes that are connected to a given node, or to find the connected components of a graph.

Shortest path algorithms

DIJKSTRA'S ALGORITHM

Dijkstra's algorithm is an algorithm for finding the shortest path between two nodes in a weighted graph. The algorithm works by iteratively adding nodes to a set of nodes that have already been visited, and it keeps track of the shortest path to each node that has been visited.

Dijkstra's algorithm is a greedy algorithm, which means that it always chooses the node that is closest to the destination node. This makes the algorithm efficient, as it does not need to explore all of the possible paths.

Dijkstra's algorithm can be implemented in Java using the following steps:

- Create a set of nodes that have not yet been visited.
- Add the start node to the set of visited nodes.
- While there are nodes to explore:
    - Find the node in the set of unvisited nodes that has the shortest distance to the destination node.
    - Add the node to the set of visited nodes.
    - Update the distances to the node's neighbors.

Here is an example of how to implement Dijkstra's algorithm in Java:

```
public class DijkstrasAlgorithm {


  public static List<Node> findShortestPath(Graph graph, Node startNode, Node endNode) {

    // Create a set of nodes that have not yet been visited.

    Set<Node> unvisitedNodes = new HashSet<>();
```

```
// Add the start node to the set of visited nodes.
Set<Node> visitedNodes = new HashSet<>();
visitedNodes.add(startNode);

// Create a map to store the distances to each node.
Map<Node, Integer> distances = new HashMap<>();
distances.put(startNode, 0);

// While there are nodes to explore:
while (!unvisitedNodes.isEmpty()) {
    // Find the node in the set of unvisited nodes that has the shortest distance to the destination node.
    Node currentNode = null;
    int minDistance = Integer.MAX_VALUE;
    for (Node node : unvisitedNodes) {
        if (distances.get(node) < minDistance) {
            currentNode = node;
            minDistance = distances.get(node);
        }
    }

    // Add the node to the set of visited nodes.
    visitedNodes.add(currentNode);
```

```
        // Update the distances to the node's neighbors.
        for (Node neighbor : currentNode.getNeighbors()) {
            if (!visitedNodes.contains(neighbor)) {
                int newDistance = distances.get(currentNode) + currentNode.getWeight(neighbor);
                if (newDistance < distances.getOrDefault(neighbor, Integer.MAX_VALUE)) {
                    distances.put(neighbor, newDistance);
                }
            }
        }
    }

    // Return the list of nodes in the shortest path.
    List<Node> shortestPath = new ArrayList<>();
    Node currentNode = endNode;
    while (currentNode != null) {
        shortestPath.add(currentNode);
        currentNode = visitedNodes.get(currentNode);
    }
    Collections.reverse(shortestPath);
    return shortestPath;
```

```
    }
}
```

Advantages and disadvantages

Dijkstra's algorithm has the following advantages:

- It is efficient, as it does not need to explore all of the possible paths.
- It can be used to find the shortest path between any two nodes in a weighted graph.

Dijkstra's algorithm has the following disadvantages:

- It can be inefficient for graphs with many nodes.
- It can be difficult to implement for graphs with negative weights.

Dijkstra's algorithm is a powerful algorithm for finding the shortest path between two nodes in a weighted graph. It is efficient, and it can be used to find the shortest path between any two nodes in a weighted graph.

BELLMAN-FORD ALGORITHM

Bellman-Ford algorithm is an algorithm for finding the shortest paths from a single source vertex to all of the other vertices in a weighted directed graph. It works by iteratively updating the distances to each vertex, starting with the source vertex and then moving on to its neighbors. The algorithm terminates when no more updates can be made, which means that the distances to all of the vertices are correct.

Bellman-Ford algorithm is a simple algorithm to implement, but it can be inefficient for graphs with many vertices. However, it is guaranteed to find the shortest paths, even if the graph contains negative weights.

Bellman-Ford algorithm can be implemented in Java using the following steps:

- Initialize the distances to all vertices to infinity.
- Set the distance to the source vertex to 0.
- Repeat the following steps for each vertex:
    - For each neighbor of the vertex:
        - If the distance to the neighbor is greater than the distance to the vertex plus the weight of the edge between the vertex and the neighbor:
            - Update the distance to the neighbor to the distance to the vertex plus the weight of the edge between the vertex and the neighbor.
- If no updates were made in the previous step, then the algorithm has converged and the distances are correct. Otherwise, the algorithm has not converged and the process must be repeated.

Here is an example of how to implement Bellman-Ford algorithm in Java:

```
public class BellmanFordAlgorithm {
```

```
public static List<Integer> findShortestPaths(Graph graph, int sourceVertex) {
    // Initialize the distances to all vertices to infinity.
    int[] distances = new int[graph.getVertices().size()];
    Arrays.fill(distances, Integer.MAX_VALUE);

    // Set the distance to the source vertex to 0.
    distances[sourceVertex] = 0;

    // Repeat the following steps for each vertex:
    for (int i = 0; i < graph.getVertices().size() - 1; i++) {
        // For each neighbor of the vertex:
        for (Edge edge : graph.getEdges()) {
            // If the distance to the neighbor is greater than the distance to the vertex plus the weight of the edge between the vertex and the neighbor:
            if (distances[edge.getDestination()] > distances[edge.getSource()] + edge.getWeight()) {
                // Update the distance to the neighbor to the distance to the vertex plus the weight of the edge between the vertex and the neighbor.
                distances[edge.getDestination()] = distances[edge.getSource()] + edge.getWeight();
            }
        }
```

```
        }


        // Return the list of vertices in the shortest paths.
        List<Integer> shortestPaths = new ArrayList<>();
        for (int i = 0; i < distances.length; i++) {
            shortestPaths.add(distances[i]);
        }
        return shortestPaths;
    }
}
```

Advantages and disadvantages

Bellman-Ford algorithm has the following advantages:

- It is simple to implement.
- It can be used to find the shortest paths between any two vertices in a weighted directed graph.
- It is guaranteed to find the shortest paths, even if the graph contains negative weights.

Bellman-Ford algorithm has the following disadvantages:

- It can be inefficient for graphs with many vertices.
- It can be difficult to implement for graphs with negative weights.

Bellman-Ford algorithm is a powerful algorithm for finding the shortest paths in a weighted directed graph. It is simple to implement, and it can be used to find the shortest paths between any two vertices in the graph. However, it can be inefficient for graphs with many vertices.

Minimum Spanning Tree algorithms

PRIM'S ALGORITHM

Prim's algorithm is an algorithm for finding the minimum spanning tree of a weighted undirected graph. A minimum spanning tree is a tree that connects all of the vertices in the graph, and it has the minimum possible total weight.

Prim's algorithm works by iteratively adding edges to a set of edges that have already been added, and it keeps track of the minimum spanning tree.

Prim's algorithm is a greedy algorithm, which means that it always chooses the edge that has the minimum weight that connects two vertices that are not already connected. This makes the algorithm efficient, as it does not need to explore all of the possible edges.

Prim's algorithm can be implemented in Java using the following steps:

- Initialize a set of edges that have already been added to the minimum spanning tree.
- Initialize a set of vertices that have not yet been added to the minimum spanning tree.
- Add the first vertex to the set of vertices that have already been added to the minimum spanning tree.
- While there are vertices that have not yet been added to the minimum spanning tree:
    - Find the edge that has the minimum weight that connects a vertex that has not yet been added to the minimum spanning tree to a vertex that has already been added to the minimum spanning tree.
    - Add the edge to the set of edges that have already been added to the minimum spanning tree.

- Add the vertex that is connected to the edge to the set of vertices that have already been added to the minimum spanning tree.

Here is an example of how to implement Prim's algorithm in Java:

```
public class PrimsAlgorithm {


   public static Set<Edge> findMinimumSpanningTree(Graph graph) {

      // Initialize a set of edges that have already been added to the minimum spanning tree.

      Set<Edge> minimumSpanningTree = new HashSet<>();


      // Initialize a set of vertices that have not yet been added to the minimum spanning tree.

      Set<Vertex> unvisitedVertices = new HashSet<>();

      for (Vertex vertex : graph.getVertices()) {

         unvisitedVertices.add(vertex);

      }


      // Add the first vertex to the set of vertices that have already been added to the minimum spanning tree.

      Vertex currentVertex = graph.getVertices().iterator().next();

      unvisitedVertices.remove(currentVertex);
```

```
// While there are vertices that have not yet been added to the minimum spanning tree:

while (!unvisitedVertices.isEmpty()) {

// Find the edge that has the minimum weight that connects a vertex that has not yet been added to the minimum spanning tree to a vertex that has already been added to the minimum spanning tree.

Edge minimumWeightEdge = null;

for (Vertex vertex : unvisitedVertices) {

  for (Edge edge : currentVertex.getEdges()) {

    if (edge.getDestination() == vertex) {

      if (minimumWeightEdge == null || edge.getWeight() < minimumWeightEdge.getWeight()) {

        minimumWeightEdge = edge;

      }

    }

  }

}


// Add the edge to the set of edges that have already been added to the minimum spanning tree.

minimumSpanningTree.add(minimumWeightEdge);


// Add the vertex that is connected to the edge to the set of vertices that have already been added to the minimum spanning tree.
```

```
unvisitedVertices.remove(minimumWeightEdge.getDestination());
        }


        // Return the set of edges that have been added to the minimum spanning tree.
        return minimumSpanningTree;
    }
}
```

Advantages and disadvantages

Prim's algorithm has the following advantages:

- It is efficient, as it does not need to explore all of the possible edges.
- It can be used to find the minimum spanning tree of any weighted undirected graph.

Prim's algorithm has the following disadvantages:

- It can be inefficient for graphs with many vertices.
- It can be difficult to implement for graphs with negative weights.

Prim's algorithm is a powerful algorithm for finding the minimum spanning tree of a weighted undirected graph. It is efficient, and it can be used to find the minimum spanning tree of any weighted undirected graph.

KRUSKAL'S ALGORITHM

Kruskal's algorithm is an algorithm for finding the minimum spanning tree of a weighted undirected graph. A minimum spanning tree is a tree that connects all of the vertices in the graph, and it has the minimum possible total weight.

Kruskal's algorithm works by iteratively adding edges to a set of edges that have already been added, and it keeps track of the minimum spanning tree.

Kruskal's algorithm is a greedy algorithm, which means that it always chooses the edge that has the minimum weight that connects two vertices that are not already connected. This makes the algorithm efficient, as it does not need to explore all of the possible edges.

Kruskal's algorithm can be implemented in Java using the following steps:

- Initialize a set of edges that have already been added to the minimum spanning tree.
- Initialize a set of edges that have not yet been added to the minimum spanning tree.
- Add all of the edges to the set of edges that have not yet been added to the minimum spanning tree.
- Sort the edges in the set of edges that have not yet been added to the minimum spanning tree by weight.
- While there are edges that have not yet been added to the minimum spanning tree:
    - Find the edge that has the minimum weight that connects two vertices that are not already connected.
    - Add the edge to the set of edges that have already been added to the minimum spanning tree.
    - Remove the edge from the set of edges that have not yet been added to the minimum spanning tree.

Here is an example of how to implement Kruskal's algorithm in Java:

```
public class KruskalsAlgorithm {


   public static Set<Edge> findMinimumSpanningTree(Graph graph) {

      // Initialize a set of edges that have already been added to the minimum spanning tree.

      Set<Edge> minimumSpanningTree = new HashSet<>();


      // Initialize a set of edges that have not yet been added to the minimum spanning tree.

      Set<Edge> unvisitedEdges = new HashSet<>();

      for (Edge edge : graph.getEdges()) {

         unvisitedEdges.add(edge);

      }


      // Sort the edges in the set of edges that have not yet been added to the minimum spanning tree by weight.

      Collections.sort(unvisitedEdges, (edge1, edge2) -> edge1.getWeight() - edge2.getWeight());


      // While there are edges that have not yet been added to the minimum spanning tree:

      while (!unvisitedEdges.isEmpty()) {

         // Find the edge that has the minimum weight that connects two vertices that are not already connected.
```

```
            Edge minimumWeightEdge = unvisitedEdges.iterator().next();


            // Add the edge to the set of edges that have already been added to the minimum spanning tree.
            minimumSpanningTree.add(minimumWeightEdge);


            // Remove the edge from the set of edges that have not yet been added to the minimum spanning tree.
            unvisitedEdges.remove(minimumWeightEdge);
        }


        // Return the set of edges that have been added to the minimum spanning tree.
        return minimumSpanningTree;
    }
}
```

Advantages and disadvantages

Kruskal's algorithm has the following advantages:

- It is efficient, as it does not need to explore all of the possible edges.
- It can be used to find the minimum spanning tree of any weighted undirected graph.
- It is deterministic, meaning that it will always find the same minimum spanning tree for a given graph.

Kruskal's algorithm has the following disadvantages:

- It can be inefficient for graphs with many edges.

- It can be difficult to implement for graphs with negative weights.

Kruskal's algorithm is a powerful algorithm for finding the minimum spanning tree of a weighted undirected graph. It is efficient, and it can be used to find the minimum spanning tree of any weighted undirected graph.

Network flow algorithms

FORD-FULKERSON ALGORITHM

The Ford-Fulkerson algorithm is an algorithm for finding the maximum flow in a network. A network is a graph that has a source node, a sink node, and a set of edges that connect the nodes. The maximum flow is the maximum amount of flow that can be sent from the source node to the sink node.

The Ford-Fulkerson algorithm works by iteratively augmenting the flow in the network. An augmentation is a path from the source node to the sink node that can carry more flow. The Ford-Fulkerson algorithm finds an augmentation by repeatedly finding a path from the source node to the sink node that can carry more flow.

The Ford-Fulkerson algorithm can be implemented in Java using the following steps:

- Initialize the flow in the network to 0.
- While there is an augmenting path:
    - Find an augmenting path from the source node to the sink node.
    - Increase the flow in the augmenting path by the minimum capacity of the edges in the augmenting path.

Here is an example of how to implement Ford-Fulkerson algorithm in Java:

```
public class FordFulkersonAlgorithm {


  public static int findMaximumFlow(Network network) {

    int flow = 0;
```

```
        while (true) {
            // Find an augmenting path.
            Path augmentingPath = findAugmentingPath(network);
            if (augmentingPath == null) {
                // There are no more augmenting paths, so we have found the
maximum flow.
                break;
            }

            // Increase the flow in the augmenting path.
            int minCapacity = Integer.MAX_VALUE;
            for (Edge edge : augmentingPath.getEdges()) {
                minCapacity = Math.min(minCapacity, edge.getCapacity());
            }
            for (Edge edge : augmentingPath.getEdges()) {
                edge.addFlow(minCapacity);
            }

            flow += minCapacity;
        }

        return flow;
    }
```

```
private static Path findAugmentingPath(Network network) {
    // Initialize a set of visited nodes.
    Set<Node> visitedNodes = new HashSet<>();

    // Initialize a queue of nodes to be visited.
    Queue<Node> queue = new LinkedList<>();
    queue.add(network.getSourceNode());

    while (!queue.isEmpty()) {
        // Get the next node from the queue.
        Node currentNode = queue.remove();

        // If the current node is the sink node, then we have found an augmenting path.
        if (currentNode == network.getSinkNode()) {
            return new Path(currentNode);
        }

        // For each neighbor of the current node:
        for (Node neighbor : currentNode.getNeighbors()) {
            // If the neighbor has not been visited:
            if (!visitedNodes.contains(neighbor)) {
```

```
                // Add the neighbor to the set of visited nodes.
                visitedNodes.add(neighbor);

                // If the neighbor has residual capacity, then add it to the queue.
                if (neighbor.hasResidualCapacity()) {
                    queue.add(neighbor);
                }
            }
        }
    }

    // There is no augmenting path.
    return null;
  }
}
```

Advantages and disadvantages

The Ford-Fulkerson algorithm has the following advantages:

- It is a simple algorithm to implement.
- It can be used to find the maximum flow in any network.

The Ford-Fulkerson algorithm has the following disadvantages:

- It can be inefficient for networks with many edges.
- It can be difficult to implement for networks with negative weights.

The Ford-Fulkerson algorithm is a powerful algorithm for finding the maximum flow in a network. It is simple to implement, and it can be used to find the maximum flow in any network.

EDMONDS-KARP ALGORITHM

The Edmonds-Karp algorithm is an algorithm for finding the maximum flow in a network. A network is a graph that has a source node, a sink node, and a set of edges that connect the nodes. The maximum flow is the maximum amount of flow that can be sent from the source node to the sink node.

The Edmonds-Karp algorithm works by iteratively augmenting the flow in the network. An augmentation is a path from the source node to the sink node that can carry more flow. The Edmonds-Karp algorithm finds an augmentation by repeatedly finding a path from the source node to the sink node that can carry more flow.

The Edmonds-Karp algorithm can be implemented in Java using the following steps:

- Initialize the flow in the network to 0.
- While there is an augmenting path:
    - Find an augmenting path from the source node to the sink node using breadth-first search.
    - Increase the flow in the augmenting path by the minimum capacity of the edges in the augmenting path.

Here is an example of how to implement Edmonds-Karp algorithm in Java:

```
public class EdmondsKarpAlgorithm {


   public static int findMaximumFlow(Network network) {

      int flow = 0;


      while (true) {

         // Find an augmenting path using breadth-first search.
```

```
        Path augmentingPath = findAugmentingPath(network);
        if (augmentingPath == null) {
            // There are no more augmenting paths, so we have found the
maximum flow.
            break;
        }

        // Increase the flow in the augmenting path.
        int minCapacity = Integer.MAX_VALUE;
        for (Edge edge : augmentingPath.getEdges()) {
            minCapacity = Math.min(minCapacity, edge.getCapacity());
        }
        for (Edge edge : augmentingPath.getEdges()) {
            edge.addFlow(minCapacity);
        }

        flow += minCapacity;
    }

    return flow;
  }

  private static Path findAugmentingPath(Network network) {
```

```
// Initialize a set of visited nodes.
Set<Node> visitedNodes = new HashSet<>();

// Initialize a queue of nodes to be visited.
Queue<Node> queue = new LinkedList<>();
queue.add(network.getSourceNode());

while (!queue.isEmpty()) {
   // Get the next node from the queue.
   Node currentNode = queue.remove();

   // If the current node is the sink node, then we have found an augmenting path.
   if (currentNode == network.getSinkNode()) {
      return new Path(currentNode);
   }

   // For each neighbor of the current node:
   for (Node neighbor : currentNode.getNeighbors()) {
      // If the neighbor has not been visited:
      if (!visitedNodes.contains(neighbor)) {
         // Add the neighbor to the set of visited nodes.
         visitedNodes.add(neighbor);
```

```
                // If the neighbor has residual capacity, then add it to the queue.
                if (neighbor.hasResidualCapacity()) {
                    queue.add(neighbor);
                }
            }
        }
    }

    // There is no augmenting path.
    return null;
  }
}
```

Advantages and disadvantages

The Edmonds-Karp algorithm has the following advantages:

- It is a simple algorithm to implement.
- It can be used to find the maximum flow in any network.
- It is faster than the Ford-Fulkerson algorithm.

The Edmonds-Karp algorithm has the following disadvantages:

- It can be inefficient for networks with many edges.
- It can be difficult to implement for networks with negative weights.

The Edmonds-Karp algorithm is a powerful algorithm for finding the maximum flow in a network. It is simple to implement, and it can be used to find the maximum flow in any network.