# Lab 4: Dinitz' Algorithm

20226758 - Jingqi Fan

May 29, 2024

# 1 Algorithm

## 1.1 Calculate Max Flow

My algorithm firstly try to identify augmenting paths and adjust flows along these paths until no more paths can be augmented. After determining the maximum flow, a traversal such as depth-first search (DFS) is used on the residual graph to identify nodes reachable from the start node. The residual graph includes edges with remaining capacities, reflecting the flow that was used during the algorithm.

```java
    public int calculateMaxFlow(Node start, Node finish) {
//        throw new RuntimeException("You need to implement this...");
        int totalFlow = 0;
        while (true) {
            Map<Node, Integer> levels = bfs(start, finish);
            if (!levels.containsKey(finish)) {
                break;
            }
            Map<Node, Iterator<FlowEdge>> edgeIterators = new HashMap<>();
            for (Node node : nodeList) {
                edgeIterators.put(node, adjacencyList.get(node).iterator());
            }
            int flow;
            do {
                flow = dfs(start, Integer.MAX_VALUE, finish, levels, edgeIterators);
                totalFlow += flow;
            } while (flow > 0);
        }
        return totalFlow;
    }
```

Figure 1: calculateMaxFlow(·)

## 1.2 Find Min Cut

After calculate max flow, my algorithm has established which nodes are reachable. The minimum cut is determined by examining the edges of the original graph. Edges that connect a reachable node to a non-reachable node are considered part of the cut. This effectively defines the boundary where the network is separated into two, with one set containing the start node and another the finish node. The nodes on the side of the start node that are connected across these edges are returned as the minimum cut set. This approach leverages the Max-Flow Min-Cut Theorem, which equates the maximum flow in the network to the capacity of the smallest set of edges. And if it is removed, it would disconnect the start from the finish.

```java
public Set<Node> findMinCut(Node start, Node finish) {
    Set<Node> reachable = new HashSet<>();
    Queue<Node> queue = new LinkedList<>();
    queue.add(start);
    reachable.add(start);
    while (!queue.isEmpty()) {
        Node node = queue.poll();
        for (FlowEdge edge : adjacencyList.get(node)) {
            if (edge.getCapacity() > edge.getFlow() && !reachable.contains(edge.getEndNode())) {
                reachable.add(edge.getEndNode());
                queue.add(edge.getEndNode());
            }
        }
    }
    Set<Node> minCut = new HashSet<>();
    for (Node u : reachable) {
        for (FlowEdge edge : adjacencyList.get(u)) {

            if (!reachable.contains(edge.getEndNode())) {
                minCut.add(edge.getEndNode());
            }
        }
    }
    return minCut;
```

Figure 2: findMinCut($\cdot$)

## 2 Complexity Analysis

**Theorem 1** *The complexity of my algorithm is $\mathcal{O}(mn^2)$.*

*Proof* The initialization of each phase typically involves constructing or reconstructing the level graph using BFS, which takes $O(m)$ time, where $m$ is the number of edges. This graph reflects the shortest paths from the source node in terms of the number of edges in the residual graph.

During each phase, the algorithm performs at most $m$ augmentations because each augmentation increases the flow and decreases the residual capacity along at least one edge, preventing that edge from being used in the same manner again within the same phase. The time complexity for all augmentations in a single phase is $O(mn)$, assuming each augmentation might involve traversing all nodes and edges in the worst case.

The retreat process can occur up to $n$ times per phase. A retreat is triggered when no augmenting path can be made from a particular node. Handling retreats involves potentially updating the level graph and the BFS tree, contributing an additional $O(m+n)$ to the complexity per phase. The number of advances per phase is limited by the product of the number of edges and nodes, $O(mn)$, since each node can attempt to advance along each edge in the residual graph.

Note that there can be at most $n-1$ phases in the algorithm. Therefore, the total time complexity of the algorithm is $O(mn^2)$. $\square$