# Route Between Nodes (4.1)

Constraints:

* Input: directed graph, source node, target node
* The route must only go from the source node to the target node (not vice versa)
* Is the graph connected? Not necessarily
* Does the graph has cycles? Possibly
* What does the algorithm have to return? Only true or false, or the path (if exists)? True and false

Idea:

* Breadth-first search from the source node
* When the target node is among the neighbours of the current node, stop and return true
* When the BFS is exhausted, return false

Implementation:

import java.util.List;

import java.util.ArrayList;

import java.util.Queue;

import java.util.LinkedList;

import java.util.Arrays;

class Route {

public static void main(String[] args) {

List<List<Integer>> graph = new ArrayList<>();

graph.add(Arrays.asList(1));

graph.add(Arrays.asList(2));

graph.add(Arrays.asList(0, 3));

graph.add(Arrays.asList(2));

graph.add(Arrays.asList(6));

graph.add(Arrays.asList(4));

graph.add(Arrays.asList(5));

int source = Integer.parseInt(args[0]);

int target = Integer.parseInt(args[1]);

if (hasPath(graph, source, target))

System.out.println("yes");

else

System.out.println("no");

}

private static boolean hasPath(List<List<Integer>> graph, int source, int target) {

if (source == target) return true;

boolean[] marked = new boolean[graph.size()];

Queue<Integer> queue = new LinkedList<>();

queue.add(source);

while (!queue.isEmpty()) {

int current = queue.remove();

marked[current] = true;

for (int v : graph.get(current)) {

if (v == target) return true;

if (!marked[v]) queue.add(v);

}

}

return false;

}

}

Complexity:

* in the worst case, we have to visit every node, and the target node is the last one
* Thus, the complexity is O(n)