```
std::pair<std::vector<int>, std::vector<int>> Graph::backwardsDijkstra(int source, int end) {
    // Checks if the nodes are valid
    if (!isNode(source) || !isNode(end)) {
        throw std::exception();
    }

    // Start from the end
    source = end;

    // Declare a priority queue that will hold a pair (cost, node)
    std::priority_queue<std::pair<int, int>, std::vector<std::pair<int, int>>, std::greater<>>
priorityQueue;

    // Declare the vector that will hold the smallest distance between the source and every node
    std::vector<int> dist(nrNodes(), INT32_MAX);

    // Declare the vector that will hold the path for all vertices
    std::vector<int> paths(nrNodes(), -1);

    // Initialise the queue and the distance vector
    priorityQueue.push(std::make_pair(0, source));
    dist[source] = 0;

    while (!priorityQueue.empty()) {
        int x = priorityQueue.top().second;  // x will hold the vertex number
        priorityQueue.pop();
        for (int y: inboundMap[x]) {  // Cycle through all inbound edges of x
            int cost = costMap[std::make_pair(y, x)];  // get the cost of the y->x edge

            if (dist[y] > dist[x] + cost) {  // Check if it is smaller than the current smallest distance
                dist[y] = dist[x] + cost;  // change the smallest cost
                paths[y] = x;  // update the shortest path
                priorityQueue.push(std::make_pair(dist[y], y));  // add the new node and the total cost to
the queue
            }
        }
    }

    return std::make_pair(dist, paths);
}
```