<https://leetcode.com/explore/learn/card/queue-stack/231/practical-application-queue/1372/>

Previously, we have already introduced two main scenarios of using BFS: do traversal or find the shortest path. Typically, it happens in a tree or a graph. As we mentioned in the chapter description, BFS can also be used in more abstract scenarios.

In this article, we will provide you with a template. Then, we provide some exercise after this article for practice.

It will be important to determine the nodes and the edges before doing BFS in a specific question. Typically, the node will be an actual node or a status while the edge will be an actual edge or a possible transition.

### *Template I*

Here we provide a pseudo code for you as a template:

/\*\*

\* Return the length of the shortest path between root and target node.

\*/

int BFS(Node root, Node target) {

Queue<Node> queue; // store all nodes which are waiting to be processed

int step = 0; // number of steps neeeded from root to current node

// initialize

add root to queue;

// BFS

while (queue is not empty) {

// iterate the nodes which are already in the queue

int size = queue.size();

for (int i = 0; i < size; ++i) {

Node cur = the first node in queue;

return step if cur is target;

for (Node next : the neighbors of cur) {

add next to queue;

}

remove the first node from queue;

}

step = step + 1;

}

return -1; // there is no path from root to target

}

1. As shown in the code, in each round, the nodes in the queue are the nodes which are waiting to be processed.
2. After each outer while loop, we are one step farther from the root node. The variable step indicates the distance from the root node and the current node we are visiting.

### *Template II*

Sometimes, it is important to make sure that we never visit a node twice. Otherwise, we might get stuck in an infinite loop, *e.g.* in graph with cycle. If so, we can add a hash set to the code above to solve this problem. Here is the pseudocode after modification:

/\*\*

\* Return the length of the shortest path between root and target node.

\*/

int BFS(Node root, Node target) {

Queue<Node> queue; // store all nodes which are waiting to be processed

Set<Node> visited; // store all the nodes that we've visited

int step = 0; // number of steps neeeded from root to current node

// initialize

add root to queue;

add root to visited;

// BFS

while (queue is not empty) {

// iterate the nodes which are already in the queue

int size = queue.size();

for (int i = 0; i < size; ++i) {

Node cur = the first node in queue;

return step if cur is target;

for (Node next : the neighbors of cur) {

if (next is not in visited) {

add next to queue;

add next to visited;

}

}

remove the first node from queue;

}

step = step + 1;

}

return -1; // there is no path from root to target

}

There are some cases where one does not need keep the visited hash set:

1. You are absolutely sure there is no cycle, for example, in tree traversal;
2. You do want to add the node to the queue multiple times.

**When we need calculate queue size in BFS ? Why do we need inner size loop in case of BFS when we have queue which processes in FIFO basis ?**

<https://stackoverflow.com/questions/65542101/why-do-we-need-inner-size-loop-in-case-of-bfs-when-we-have-queue-which-processes>

Q: This is the algorithmic template for BFS. In this we have a while loop which checks emptiness of queue. My question is why we need inner for loop which loops till size of queue. Whereas when queue processes elements in FIFO manner.

int BFS(Node root, Node target) {

Queue<Node> queue; // store all nodes which are waiting to be processed

int step = 0; // number of steps neeeded from root to current node

// initialize

add root to queue;

// BFS

while (queue is not empty) {

step = step + 1;

// iterate the nodes which are already in the queue

int size = queue.size();

for (int i = 0; i < size; ++i) { // --> why need this for loop ?

Node cur = the first node in queue;

return step if cur is target;

for (Node next : the neighbors of cur) {

add next to queue;

}

remove the first node from queue;

}

}

return -1; // there is no path from root to target

}

**A: Because otherwise step will not be correct, so when our purpose relate to step calculation, we have to calculate the size of queue to make sure iterate all current level elements then move to next and depth + 1**

**Another use case is level order traversal, we have to record sequence strictly based on each level**

**e.g L102/P8.1.Binary Tree Level Order Traversal**

class Solution {

public List<List<Integer>> levelOrder(TreeNode root) {

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

if(root == null) {

return result;

}

Queue<TreeNode> q = new LinkedList<TreeNode>();

q.offer(root);

while(!q.isEmpty()) {

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

int size = q.size();

for(int i = 0; i < size; i++) {

TreeNode node = q.poll();

list.add(node.val);

if(node.left != null) {

q.offer(node.left);

}

if(node.right != null) {

q.offer(node.right);

}

}

result.add(list);

}

return result;

}

}