# 广度优先搜索算法

## 1.算法简介

广度优先搜索算法(Breadth First Search)，又称为"宽度优先搜索"或"横向优先搜索"，简称BFS； BFS是用于图的查找算法(要求能用图表示出问题的关联性)。

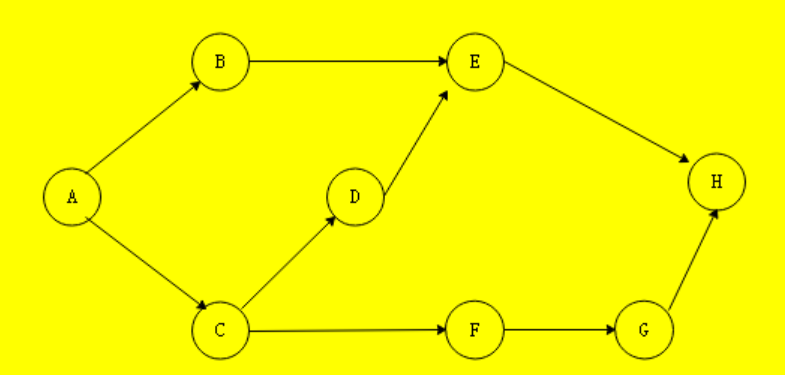
BFS可用于解决2类问题：

从A出发是否存在到达B的路径；

从A出发到达B的最短路径(这个应该叫最少步骤合理)；

其思路为从图上一个节点出发，访问先访问其直接相连的子节点，若子节点不符合，再问其子节点的子节点，按级别顺序依次访问，直到访问到目标节点。

## 2.图问题



如上图所示，找出从A到H的最短路径(步骤最少的，假设每一段距离相等),此时就可以使用广域搜索算法，原理步骤为:

假设存在一个空的搜索队列Queue,首先将节点A添加进队列Queue

判断队列第一个节点是否是需要查找的目标节点，若不是，则将第一个节点的直接子节点添加进队列，并移除第一个节点

重复判断，直到第一个节点为目标节点，或者队列为空(即代表没有合适的)

## 3.实验代码

**import** java.util.ArrayList;

**import** java.util.HashMap;

**import** java.util.LinkedList;

**import** java.util.List;

**public** **class** BFS {

**public** **static** **void** main(String[] args){

//初始化先建立起各个节点信息，以及对应的直接子节点列表

HashMap<String,String[]> map = **new** HashMap<>();

map.put("A", **new** String[] {"B","C"});

map.put("B", **new** String[] {"E"});

map.put("C", **new** String[] {"D","F"});

map.put("D", **new** String[] {"E"});

map.put("E", **new** String[] {"H"});

map.put("F", **new** String[] {"E","G"});

map.put("G", **new** String[] {"H"});

map.put("H", **new** String[] {});

//获取从A到H的最短路径节点链表

Node target = *findTarget*("A","H",map);

//打印出最短路径的各个节点信息

*printSearPath*(target);

}

/\*\*

\* 打印出到达节点target所经过的各个节点信息

\* **@param** target

\*/

**static** **void** printSearPath(Node target) {

**if** (target != **null**) {

System.***out***.print("找到了目标节点:" + target.id + "\n");

List<Node> searchPath = **new** ArrayList<Node>();

searchPath.add(target);

Node node = target.parent;

**while**(node!=**null**) {

searchPath.add(node);

node = node.parent;

}

String path = "";

**for**(**int** i=searchPath.size()-1;i>=0;i--) {

path += searchPath.get(i).id;

**if**(i!=0) {

path += "-->";

}

}

System.***out***.print("步数最短："+path);

} **else** {

System.***out***.print("未找到了目标节点");

}

}

/\*\*

\* 从指定的开始节点 startId ，查询到目标节点targetId 的最短路径

\* **@param** startId

\* **@param** targetId

\* **@param** map

\* **@return**

\*/

**static** Node findTarget(String startId,String targetId,HashMap<String,String[]> map) {

List<String> hasSearchList = **new** ArrayList<String>();

LinkedList<Node> queue = **new** LinkedList<Node>();

queue.offer(**new** Node(startId,**null**));

**while**(!queue.isEmpty()) {

Node node = queue.poll();

**if**(hasSearchList.contains(node.id)) {

//跳过已经搜索过的，避免重复或者死循环

**continue**;

}

System.***out***.print("判断节点:" + node.id +"\n");

**if** (targetId.equals(node.id)) {

**return** node;

}

hasSearchList.add(node.id);

**if** (map.get(node.id) != **null** && map.get(node.id).length > 0) {

**for** (String childId : map.get(node.id)) {

queue.offer(**new** Node(childId,node));

}

}

}

**return** **null**;

}

/\*\*

\* 节点对象

\* **@author** Administrator

\*

\*/

**static** **class** Node{

//节点唯一id

**public** String id;

//该节点的直接父节点

**public** Node parent;

//该节点的直接子节点

**public** List<Node> childs = **new** ArrayList<Node>();

**public** Node(String id,Node parent) {

**this**.id = id;

**this**.parent = parent;

}

}

}

