第八章 搜索

文章目录

搜索

深度优先搜索和宽度优先搜索

DFS基本算法

（1）算法思想

1. 利用DFS遍历图 graph，从任意结点 start 出发
2. 通过栈存储中间结点，每次弹出一个结点，并遍历该结点的邻接结点，若该结点未访问，则加入遍历结果列表。

（2）程序代码

```
// DFS
    public List<Integer> DFS(List graph, GraphNode start){
        // 利用DFS遍历图 graph，从任意结点 start 出发
        // 通过栈存储中间结点，每次弹出一个结点，并遍历该结点的邻接结点，若该结点未访问，则加入遍历结果列表
        // 本质就是将BFS中的队列用栈表示

        List result = new ArrayList<>(); // 存储遍历结果列表
        Stack<GraphNode> stack = new Stack<>(); // 通过队列存储中间处理数据
        //初始化
        stack.push(start); // 初始结点加入栈
        result.add(start.key); // 对初始结点进行访问

        while(!stack.isEmpty()) {
            // 队列不为空时，弹出队首结点，寻找该结点的邻接结点
            GraphNode head = stack.pop();

            List<neighborNode> neighbors = head.neighbors; // 获取该结点的邻结点
            for(int i=0;i<neighbors.size();i++) {
                // 对该结点的邻结点进行遍历，若没有访问过，则加入队列
                char neighborValue = neighbors.get(i).key;
                if(!result.contains(neighborValue)) { // 结果列表中不包含该结点，则对该结点进行访问
                    stack.push(findGraphNodeByKey(graph,neighborValue)); // 将该结点加入队列
                    result.add(neighborValue); // 已访问的结点加入结果列表
                }
            }
        }

        return result;
    }
```

BFS基本算法

（1）算法思想

1. 利用BFS遍历图 graph，从任意结点 start 出发。
2. 通过队列存储中间结点，每次弹出一个结点，并遍历该结点的邻接结点，若该结点未访问，则加入遍历结果列表。

（2）程序代码

```
    // BFS
    public List BFS(List graph, GraphNode start){
        // 利用BFS遍历图 graph，从任意结点 start 出发
        // 通过队列存储中间结点，每次弹出一个结点，并遍历该结点的邻接结点，若该结点未访问，则加入遍历结果列表
        // 特别的，对于队列而言，BFS的结果就是队列的层序遍历

        List result = new ArrayList<>(); // 存储遍历结果列表
        Queue<GraphNode> queue = new LinkedList<>(); // 通过队列存储中间处理数据
        //初始化
        queue.offer(start); // 初始结点加入队列
        result.add(start.key); // 对初始结点进行访问

        while(!queue.isEmpty()) {
            // 队列不为空时，弹出队首结点，寻找该结点的邻接结点
            GraphNode head = queue.poll();

            List<neighborNode> neighbors = head.neighbors; // 获取该结点的邻结点
            for(int i=0;i<neighbors.size();i++) {
                // 对该结点的邻结点进行遍历，若没有访问过，则加入队列
                char neighborValue = neighbors.get(i).key;
                if(!result.contains(neighborValue)) { // 结果列表中不包含该结点，则对该结点进行访问
                    queue.offer(findGraphNodeByKey(graph,neighborValue)); // 将该结点加入队列
                    result.add(neighborValue); // 已访问的结点加入结果列表
                }
            }
        }

        return result;
    }
```

leetcode

例1：岛屿数量（200）

题目描述

给定一个由 ‘1’（陆地）和 ‘0’（水）组成的的二维网格，计算岛屿的数量。一个岛被水包围，并且它是通过水平方向或垂直方向上相邻的陆地连接而成的。你可以假设网格的四个边均被水包围。

```
示例 1:
输入:
11110
11010
11000
00000
输出: 1

示例 2:
输入:
11000
11000
00100
00011
输出: 3
```

算法思路

采用DFS搜索：

1. 标记当前搜索位置已被搜索（标记当前位置的mark数组为1）
2. 按照方向数组的4个方向，扩展4个新位置newx、newy
3. 若新位置不在地图范围内，则忽略
4. 若新位置未曾到达过（mark[newx][newy]为0）、且是陆地（grid[newx][newy]为"1"），继续DFS该位置。

程序代码

```
    // 200.岛屿数量
    // 给定一个由 '1'（陆地）和 '0'（水）组成的的二维网格，
    // 计算岛屿的数量。一个岛被水包围，并且它是通过水平方向或垂直方向上相邻的陆地连接而成的。
    // 你可以假设网格的四个边均被水包围。
    public int numIslands(char[][] grid) {
        int number = 0; // 记录最终岛屿的数量
        // char[][] grid // 记录岛屿上可达区域与不可达区域（陆地1/水0）
        if(grid == null || grid.length == 0)return number;
        Integer height = grid.length;
        Integer width = grid[0].length;

        boolean[][] visited = new boolean[height][width]; // 记录当前位置是否已访问
        // 初始化所有位置均未访问
        for(int i=0;i<height;i++)
                for(int j=0;j<width;j++)
                    visited[i][j] = false;
        // 从初始位置(0,0)开始遍历
        for(int i=0;i<height;i++)
                for(int j=0;j<width;j++) {
                    // 若当前位置（i,j）可达grid[i][j]='1' && 未访问visiyed[i][j] = false
                    // 岛屿数量+1，并开始DFS访问该岛屿上所有的陆地
                    if(grid[i][j] == '1' && visited[i][j] == false) {
                        number++;
                        islandDFS(grid,visited,new Pair(i,j));
                    }
                }
        return number;
    }

    public void islandDFS(char[][] grid,boolean[][] visited,Pair<Integer,Integer> start) {
        // 以start 为起始点开始DFS
            List result = new ArrayList<>(); // 存储遍历结果列表
        Stack<Pair> stack = new Stack<Pair>(); // 通过队列存储中间处理数据
        //初始化
        stack.push(start); // 初始结点加入栈
        visited[start.getFirst()][start.getSecond()] = true;

        while(!stack.isEmpty()) {
            // 队列不为空时，弹出队首结点，寻找该结点的所有邻接结点
            // 若邻接结点可访问且未访问，则访问并入栈
            Pair head = stack.pop();
            visitNeighbours(stack,grid,visited,head);
        }
    }

    public void visitNeighbours(Stack stack,char[][] grid,boolean[][] visited,Pair<Integer,Integer> cur_node) {
        final int dx[] = {-1,1,0,0};
        final int dy[] = {0,0,-1,1};
        int x = cur_node.getFirst();
        int y = cur_node.getSecond();

        for(int i=0;i<4;i++) {
            int new_x = x + dx[i];
            int new_y = y + dy[i];

            if(new_x >= 0 && new_x < grid.length && new_y >= 0 && new_y < grid[0].length)
                if(grid[new_x][new_y] == '1' && visited[new_x][new_y] == false){// 若该位置可达且未访问
                    stack.push(new Pair(new_x,new_y));
                    visited[new_x][new_y] = true;
            }
        }
    }
```

例2：词语阶梯（127）

题目描述

给定两个单词（beginWord 和 endWord）和一个字典，找到从 beginWord 到 endWord 的最短转换序列的长度。转换需遵循如下规则：

每次转换只能改变一个字母。

转换过程中的中间单词必须是字典中的单词。

说明:

如果不存在这样的转换序列，返回 0。

所有单词具有相同的长度。

所有单词只由小写字母组成。

字典中不存在重复的单词。

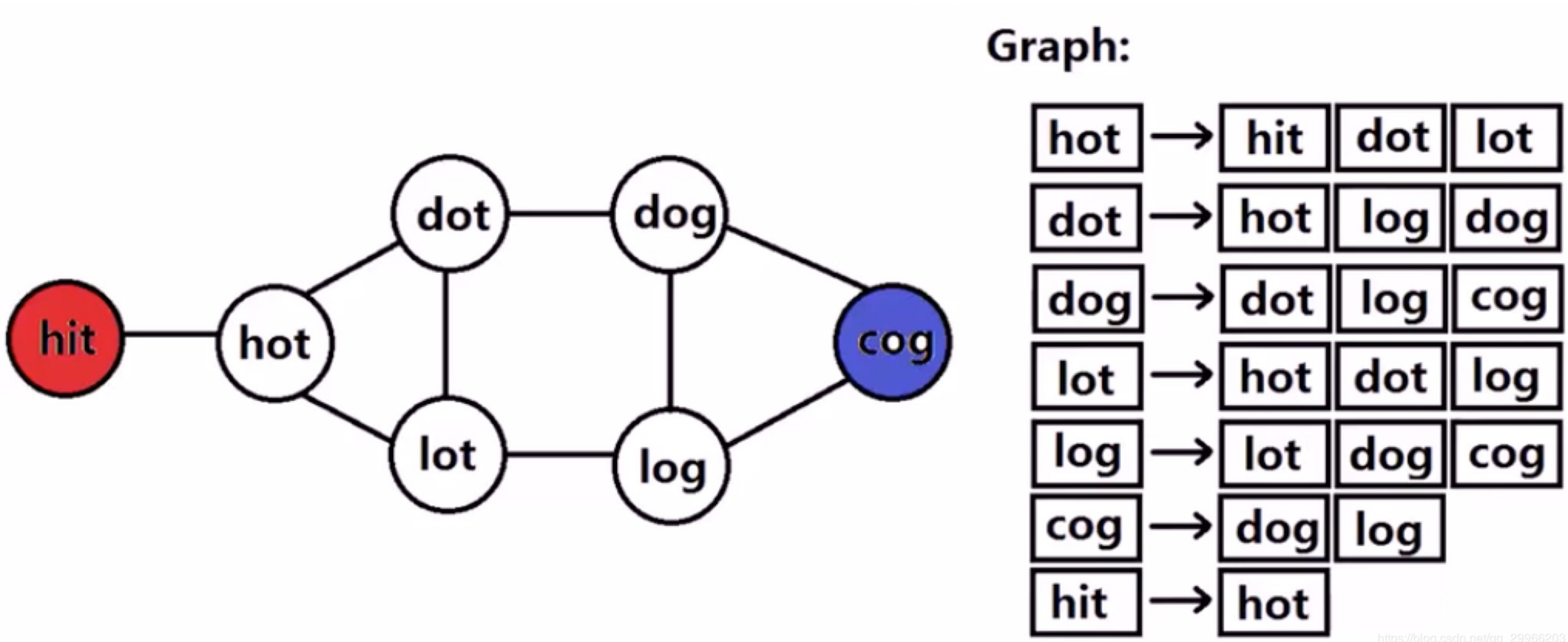
你可以假设 beginWord 和 endWord 是非空的，且二者不相同。

```
示例 1:

输入:
beginWord = "hit",
endWord = "cog",
wordList = ["hot","dot","dog","lot","log","cog"]

输出: 5

解释: 一个最短转换序列是 "hit" -> "hot" -> "dot" -> "dog" -> "cog",
     返回它的长度 5。
示例 2:

输入:
beginWord = "hit"
endWord = "cog"
wordList = ["hot","dot","dog","lot","log"]

输出: 0

解释: endWord "cog" 不在字典中，所以无法进行转换。
```

算法思路

（1）图的表示与构造

使用map构造邻接表表示的图，map定义为以string为key（代表图的顶点），vector为value(代表图的各个顶点邻接顶点)，如下图所示：

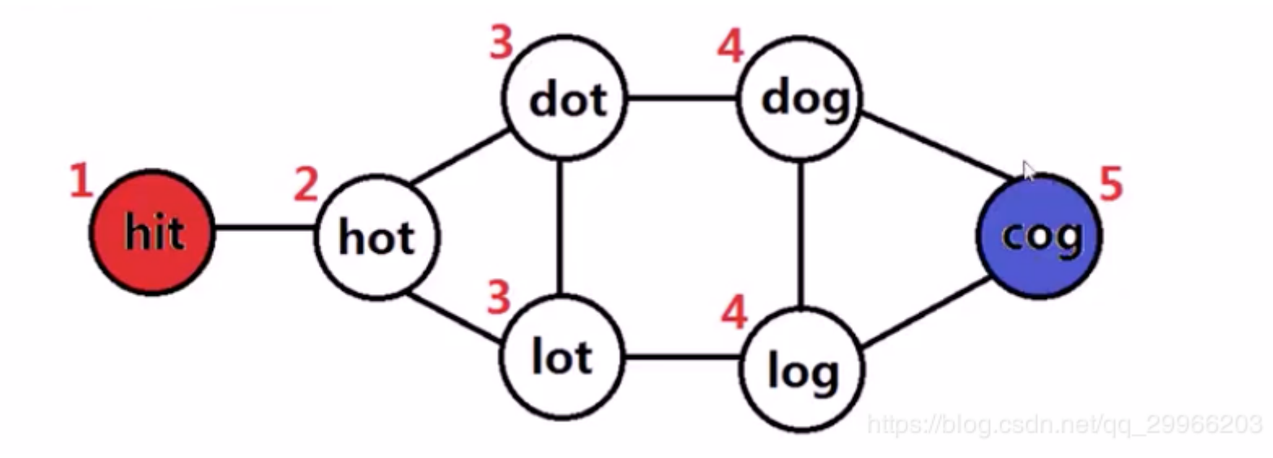


将beginWord push进入wordList.遍历wordList，对任意两个单词wordList[i]与wordList[j]，如果wordList[i]与wordList[j]只相差1个字符，则将其相连。

（2）图的宽度遍历

给定图的起点beginWord，终点endWord，图graph，从beginWord开始宽度优先搜索图graph，搜索过程中记录到达步数；

1. 设置搜索队列Q，队列结点为pair<顶点，步数>，设置集合visit，记录搜索过的顶点；将<beginWord,1>添加至队列；
2. 只要队列不空，取出队列头部元素：
   （1）若取出队列头部元素为endWord,返回到达当前节点的步数；
   （2）否则扩展该节点，将与节点相邻的且未添加到visit中的节点与步数同时添加至队列Q，并将扩展节点加入visit；
3. 若最终都无法搜索到endWord，返回0.



程序代码

```
    // 针对超出时间限制（下采用构造邻接表尽兴预处理：空间换时间思想）
    // 算法中最重要的步骤是找出相邻的节点，也就是只差一个字母的两个单词。为了快速的找到这些相邻节点，我们对给定的 wordList 做一个预处理。
    // 这步预处理找出了单词表中所有单词改变某个字母后的通用状态，并帮助我们更方便也更快的找到相邻节点。
```

```
    // 否则，对于每个单词我们需要遍历整个字母表查看是否存在一个单词与它相差一个字母，这将花费很多时间。
    // 预处理操作在广度优先搜索之前高效的建立了邻接表。
    public int ladderLength(String beginWord, String endWord, List<String> wordList) {
        if(!wordList.contains(endWord))return 0;
        // 先创建邻接表（接龙单词对应邻接表）——空间换时间策略
        List<GraphNode>graphList = initGraph(beginWord,wordList);

        List<Integer> lengthList = new ArrayList(); // 存放各个可行单词接龙的长度
        Queue<Pair<GraphNode,Integer>> queue = new LinkedList(); // 存放每次接龙的单词的字符结点及接龙次数
        List<String> visited = new ArrayList(); // 存放已访问的单词，防重复
        Pair<GraphNode, Integer> beginPair = new Pair<GraphNode,Integer>(graphList.get(0),1); // 图头结点，即 beginWord 所在头结点
        queue.offer(beginPair);
        while(!queue.isEmpty()) {
                // 采用BFS进行层序遍历（记录接龙次数）
                Pair<GraphNode,Integer> curPair = queue.poll();
                GraphNode<String> curNode = curPair.getFirst();
                Integer curRow = curPair.getSecond();
                List neighbors = curNode.neighbors; // 获取该结点的邻结点
                for(int i=0;i<neighbors.size();i++) {
                    // 对该结点的邻结点进行遍历，若没有访问过，则加入队列
                    String neighbor = (String) neighbors.get(i);
                    if(neighbor.equals(endWord))lengthList.add(curRow+1);
                    if(!visited.contains(neighbor)) { // 结果列表中不包含该结点，则对该结点进行访问
                        queue.offer(new Pair(findGraphNodeByKey(neighbor,graphList),curRow+1)); // 将该结点加入队列
                        visited.add(neighbor); // 已访问的结点加入结果列表
                    }
                }
        }
        lengthList.sort(null); // sort默认升序排序

        if(lengthList.size()>0)return lengthList.get(0);
        else return 0;
    }

        public List<GraphNode> initGraph(String beginWord,List<String> wordList) {
            // 返回头结点为beginWord的图（邻接表表示）
            Set<GraphNode> graphList = new HashSet<>();// 采用Set存储，防重复
            wordList.add(0,beginWord);// 头结点加入WordList
        // 单词列表中每个结点处理
            for(int i=0;i<wordList.size();i++) {
                String ref_word = wordList.get(i);
                GraphNode node = new GraphNode(ref_word);
                // 对于每个单词结点，将满足接龙条件的单词插入邻接结点
                for(int j=0;j<wordList.size();j++) {
                    String cur_word = wordList.get(j);
                    if(isSatisfyLadder(ref_word,cur_word))node.neighbors.add(cur_word);
                }
                graphList.add(node);
            }

            return new ArrayList<>(graphList);
    }

        public boolean isSatisfyLadder(String ref_word,String cur_word) {
            // 是否满足接龙单词(只有一个位置字符不同的单词)
                int connect = 0; // 匹配字符的数目
            if(ref_word.length() != cur_word.length())return false;
            for(int i=0;i<ref_word.length();i++) {
                if(ref_word.charAt(i) == cur_word.charAt(i)) connect++;
            }
            if(connect == ref_word.length()-1)return true;// 若匹配字符的数目为字符长度-1，则满足接龙单词的规律
            else return false;
    }

        public GraphNode findGraphNodeByKey(String key,List<GraphNode> graphList) {
                for(int i=0;i<graphList.size();i++) {
                    GraphNode node = graphList.get(i);
                    if(node.label.equals(key))return node;
                }
                return null;
        }
```

改进（超时）

```
    public int ladderLength(String beginWord, String endWord, List<String> wordList) {
        Integer min_length = 0;
        HashMap<String,ArrayList<String>> combineNode = new HashMap<String,ArrayList<String>>();    // 利用哈希表存储单词及其邻接节点（仿图结构）
        List<String> visited = new ArrayList<String>(); // 存储已经访问过的单词节点
        initCombineNode(combineNode, beginWord, wordList);
        // 对图结构进行BFS
        Queue<Pair<String, Integer>> Q = new LinkedList<Pair<String, Integer>>();
        Q.offer(new Pair<String,Integer>(beginWord,1));
        while(!Q.isEmpty()) {
                Pair<String,Integer> cur_pair = Q.poll();
                String cur_word = cur_pair.getFirst();  // 单词
                Integer cur_step = cur_pair.getSecond();    // 步数
                if(cur_word.equals(endWord) && (cur_step<min_length || min_length == 0)) min_length = cur_step;
                ArrayList<String> neighbours = combineNode.get(cur_word);
                for(int i=0;i<neighbours.size();i++) {
                    String neighbourWord = neighbours.get(i);
                    if(!visited.contains(neighbourWord)) {
                        Q.offer(new Pair<String,Integer>(neighbourWord,cur_step+1));
                        visited.add(neighbourWord);
                    }
                }
        }
        return min_length;
    }

    public void initCombineNode(HashMap<String,ArrayList<String>> combineNode, String beginWord, List<String> wordList) {
            if(!wordList.contains(beginWord))wordList.add(beginWord);
```

```
            for(int i=0;i<wordList.size();i++) {
                String ref_word = wordList.get(i);
                ArrayList<String> combineList = new ArrayList<String>();
                // 对于每个单词结点，将满足接龙条件的单词插入邻接结点
                for(int j=0;j<wordList.size();j++) {
                    String cur_word = wordList.get(j);
                    if(isSatisfyLadder(ref_word,cur_word))combineList.add(cur_word) ;
                }
                combineNode.put(ref_word, combineList);
            }
    }

    public boolean isSatisfyLadder(String ref_word,String cur_word) {
        // 是否满足接龙单词(只有一个位置字符不同的单词)
            int connect = 0; // 匹配字符的数目
        if(ref_word.length() != cur_word.length())return false;
        for(int i=0;i<ref_word.length();i++) {
            if(ref_word.charAt(i) == cur_word.charAt(i)) connect++;
        }
        if(connect == ref_word.length()-1)return true;// 若匹配字符的数目为字符长度-1，则满足接龙单词的规律
        else return false;
}
```

官方解法

```
import javafx.util.Pair;

class Solution {
  public int ladderLength(String beginWord, String endWord, List<String> wordList) {

    // Since all words are of same length.
    int L = beginWord.length();

    // Dictionary to hold combination of words that can be formed,
    // from any given word. By changing one letter at a time.
    HashMap<String, ArrayList<String>> allComboDict = new HashMap<String, ArrayList<String>>();

    wordList.forEach(
        word -> {
          for (int i = 0; i < L; i++) {
            // Key is the generic word
            // Value is a list of words which have the same intermediate generic word.
            String newWord = word.substring(0, i) + '*' + word.substring(i + 1, L);
            ArrayList<String> transformations =
                allComboDict.getOrDefault(newWord, new ArrayList<String>());
            transformations.add(word);
            allComboDict.put(newWord, transformations);
          }
        });

    // Queue for BFS
    Queue<Pair<String, Integer>> Q = new LinkedList<Pair<String, Integer>>();
    Q.add(new Pair(beginWord, 1));

    // Visited to make sure we don't repeat processing same word.
    HashMap<String, Boolean> visited = new HashMap<String, Boolean>();
    visited.put(beginWord, true);

    while (!Q.isEmpty()) {
      Pair<String, Integer> node = Q.remove();
      String word = node.getKey();
      int level = node.getValue();
      for (int i = 0; i < L; i++) {

        // Intermediate words for current word
        String newWord = word.substring(0, i) + '*' + word.substring(i + 1, L);

        // Next states are all the words which share the same intermediate state.
        for (String adjacentWord : allComboDict.getOrDefault(newWord, new ArrayList<String>())) {
          // If at any point if we find what we are looking for
          // i.e. the end word - we can return with the answer.
          if (adjacentWord.equals(endWord)) {
            return level + 1;
          }
          // Otherwise, add it to the BFS Queue. Also mark it visited
          if (!visited.containsKey(adjacentWord)) {
            visited.put(adjacentWord, true);
            Q.add(new Pair(adjacentWord, level + 1));
          }
        }
      }
    }

    return 0;
  }
}

```

例3：词语阶梯2（126）

题目描述

给定两个单词（beginWord 和 endWord）和一个字典 wordList，找出所有从 beginWord 到 endWord 的最短转换序列。转换需遵循如下规则：

每次转换只能改变一个字母。
转换过程中的中间单词必须是字典中的单词。
说明:

如果不存在这样的转换序列，返回一个空列表。
所有单词具有相同的长度。
所有单词只由小写字母组成。

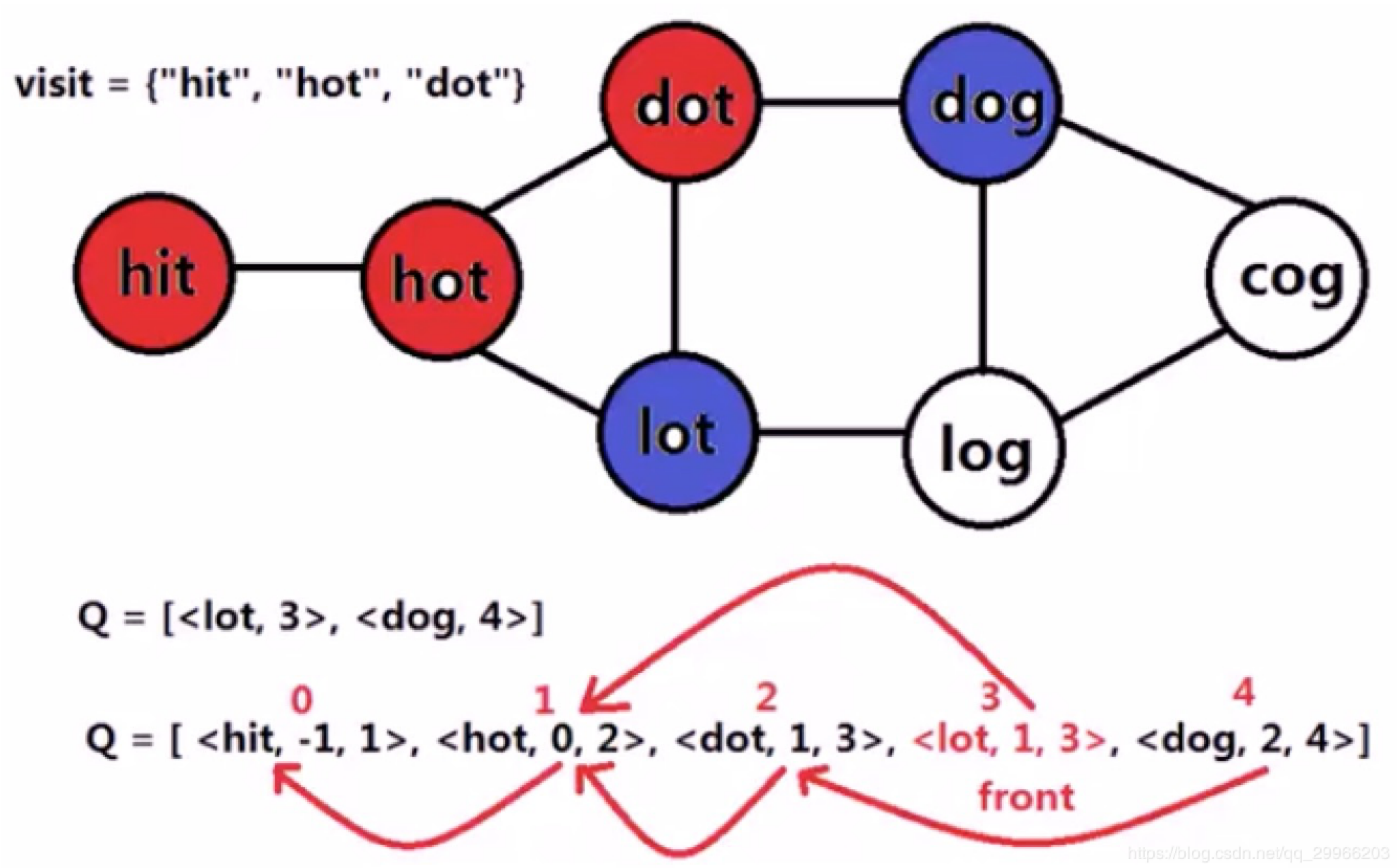
字典中不存在重复的单词。
你可以假设 beginWord 和 endWord 是非空的，且二者不相同。

```
示例 1:
输入:
beginWord = "hit",
endWord = "cog",
wordList = ["hot","dot","dog","lot","log","cog"]

输出:
[
  ["hit","hot","dot","dog","cog"],
  ["hit","hot","lot","log","cog"]
]
示例 2:
输入:
beginWord = "hit"
endWord = "cog"
wordList = ["hot","dot","dog","lot","log"]

输出: []

解释: endWord "cog" 不在字典中，所以不存在符合要求的转换序列。
```

算法思路

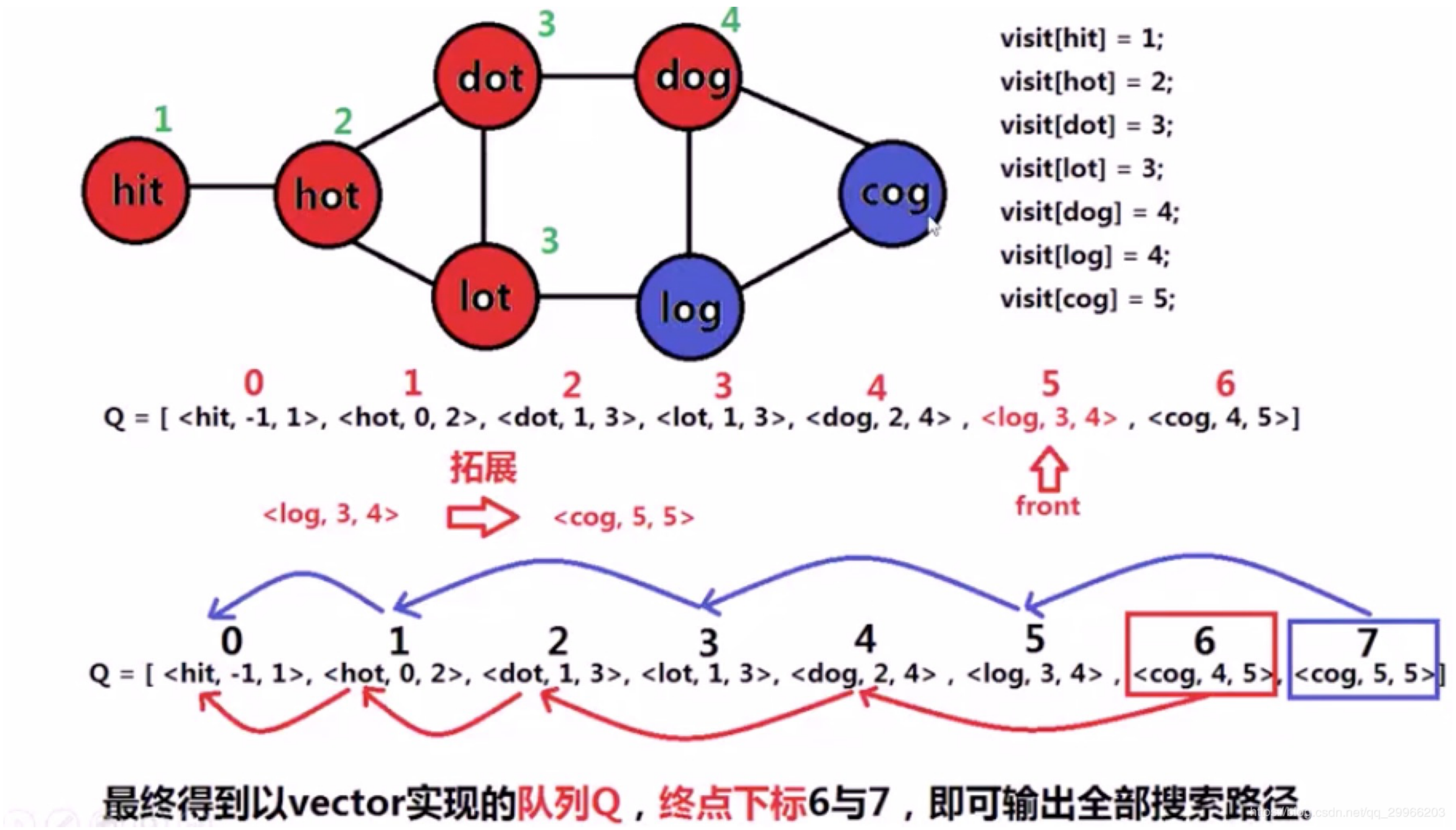
1. 记录路径的宽度搜索
   （1）将普通队列更换为vector实现队列，保存所有的搜索节点，即在pop节点时不会丢弃队头元素，只是移动front指针。
   （2）在队列节点中增加该节点的前驱节点在队列中的下标信息，可通过该下标找到是队列中的哪个节点搜索到地当前节点。

```
    class BFSItem{
            String word; // 搜索节点
            Integer parent_pos; //前驱节点在队列中的位置
            Integer step; //到达当前节点的步数

            BFSItem(String word, Integer parent_pos, Integer step){
                this.word = word;
                this.parent_pos = parent_pos;
                this.step = step;
            }
    }
```

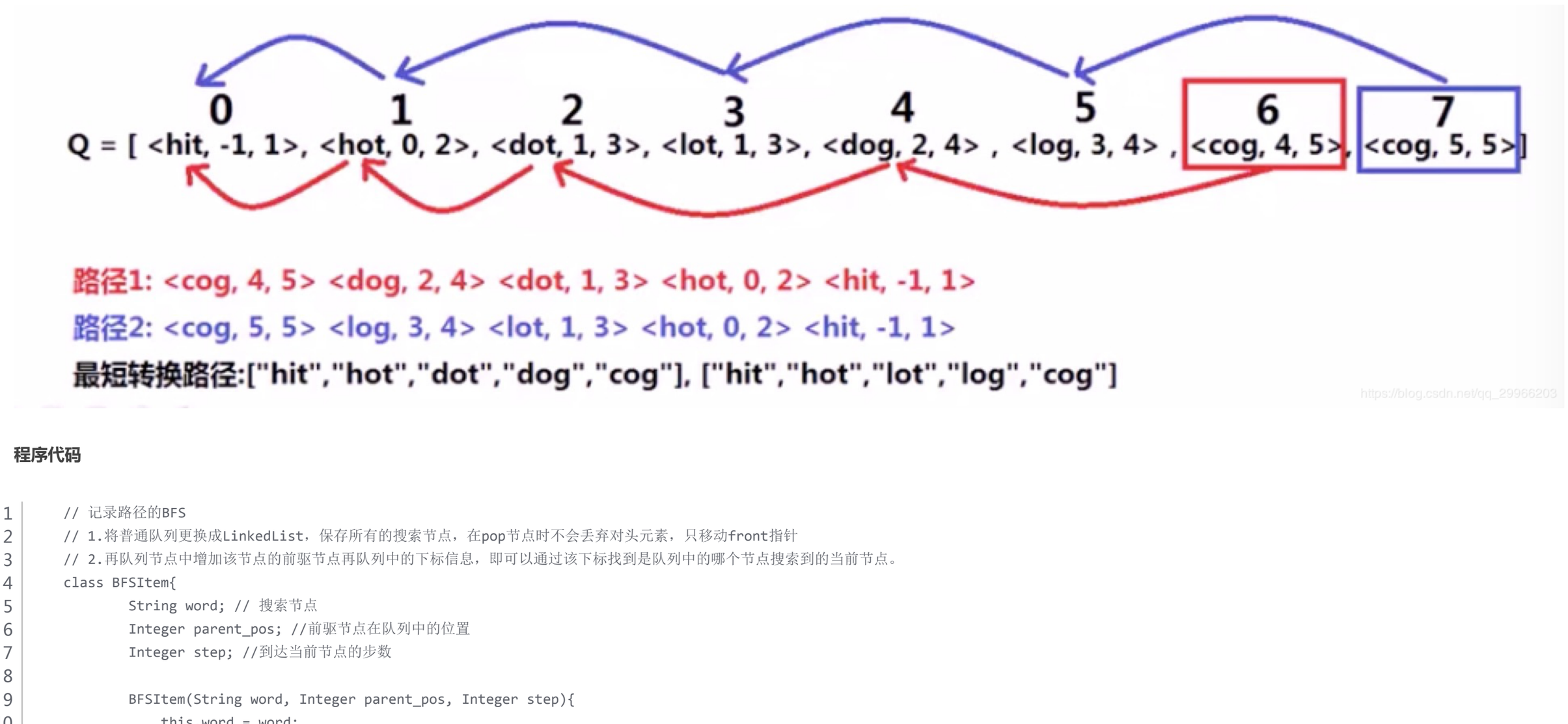


2. 多条路径的保存
   到达某一位置可能存在多条路径，使用映射记录到达每个位置的最短需要步数，新扩展到的位置只要未曾到达或到达步数与最短步数相同，即将该位置添加到队列中，从而存储了从不同前驱到达该位置的情

况。



3. 遍历搜索路径
   从所有结果（endWord）所在的队列位置（end_word_pos），向前遍历直到起始单词（beginWord），遍历过程中，保存路径上的单词。如此遍历得到的路径为endWord到beginWord的路径，将其按从尾到头的顺序存储到最终结果中即可。



程序代码

```
// 记录路径的BFS
// 1.将普通队列更换成LinkedList，保存所有的搜索节点，在pop节点时不会丢弃对头元素，只移动front指针
// 2.再队列节点中增加该节点的前驱节点再队列中的下标信息，即可以通过该下标找到是队列中的哪个节点搜索到的当前节点。
class BFSItem{
        String word; // 搜索节点
        Integer parent_pos; //前驱节点在队列中的位置
        Integer step; //到达当前节点的步数

        BFSItem(String word, Integer parent_pos, Integer step){
            this.word = word;
            this.parent_pos = parent_pos;
            this.step = step;
        }
}

// 126. 单词接龙 II
// 给定两个单词（beginWord 和 endWord）和一个字典 wordList，找出所有从 beginWord 到 endWord 的最短转换序列。
// 转换需遵循如下规则：
// 每次转换只能改变一个字母。
// 转换过程中的中间单词必须是字典中的单词。
public List<List<String>> findLadders(String beginWord, String endWord, List<String> wordList) {
        List<List<String>> result = new ArrayList<List<String>>();  // 存储所有最短转换序列
        HashMap<String,ArrayList<String>> combineNode = new HashMap<String,ArrayList<String>>();    // 利用哈希表存储单词及其邻接节点列表（图结构）
    HashMap<String,Integer> visited = new HashMap<String, Integer>();   // 存储已经访问过的单词节点的单词及步数
    List<Integer> end_word_pos = new ArrayList<Integer>(); // 终点endWord所在队列位置下标
    initCombineNode(combineNode, beginWord, wordList); // 构造邻接图
    Integer min_step = 0;//到达endWord的最小步数
    // 对图结构进行BFS
    ArrayList<BFSItem> Q = new ArrayList<BFSItem>();    // 构造自定义队列，存储BFS节点（节点单词，父结点下标，步数）
    Q.add(new BFSItem(beginWord,-1,1)); // 起始单词的前驱为-1
    visited.put(beginWord, 1); // 标记起始单词的步数为1
    Integer front = 0; // 队列指针front指向队列Q的队列头
    while(front != Q.size()) {
            BFSItem cur_item = Q.get(front); // 队列头元素
```

```
            String cur_word = cur_item.word;    // 当前元素单词
            Integer cur_step = cur_item.step; // 当前元素步数
            if(min_step!=0 && cur_step>min_step)break; // step>min_step时，代表所有到达终点的路径都搜索完成
            if(cur_word == endWord) {min_step = cur_step;end_word_pos.add(front);} // 搜索到结果时，记录到达终点的最小步数

            ArrayList<String> neighbours = combineNode.get(cur_word);
            for(int i=0;i<neighbours.size();i++) {
                String neighbourWord = neighbours.get(i);
                if(!visited.containsKey(neighbourWord) || visited.get(neighbourWord) == cur_step+1) {   // 节点未访问，或是另一条最短路径
                    Q.add(new BFSItem(neighbourWord,front,cur_step+1));
                    visited.put(neighbourWord, cur_step+1);//标记到达邻接点neighbourWord的最小步数
                }
            }
            front++;
        }
        // 从endWord到beginWord将路径上的节点值push进入path
        for(int i=0;i<end_word_pos.size();i++) {    // 作为一条路径
            int pos = end_word_pos.get(i);
            List<String> path = new ArrayList<String>();
            while(pos!=-1) {
                path.add(Q.get(pos).word);
                pos = Q.get(pos).parent_pos;    // 根据前置节点找路径
            }
            List<String> result_item = new ArrayList<String>();
            for(int j=path.size()-1;j>=0;j--) result_item.add(path.get(j));
            result.add(result_item);
        }
        return result;
    }
```

剑指offer