关键知识点1 图中的广度优先搜索

伪代码

```
  广度优先搜索 BFS(start_node)
1、 queue = {start_node}, initialize visited to all False
2、 visited[start_node] = True
3、 while(!queue.empty()){
4、   node = queue.front()
5、   for neighbor_node in adj(node){ //遍历node的邻居
6、     if(!visited[neighbor_node]){
7、       visited[neighbor_node] = True
8、       queue.push_back(neighbor_node)
9、       //do something here
10、    }
11、  }
12、}
```

关键知识点2 图中的深度优先搜索

伪代码

```
  深度优先搜索 DFS(node)
1、 visited[node] = True
2、 for neighbor_node in adj(node){ //遍历node的邻居
3、   if(!visited[neighbor_node]){
4、     //do something here
5、     DFS(neighbor_node)
6、   }
7、 }
```

基本解法

Java代码

```
 public boolean findWhetherExistsPath(int n, int[][] graph, int start, int
target) {
        //建立邻接表 adj
        ArrayList<ArrayList<Integer> > adj  = new ArrayList<ArrayList<Integer>>
(n);
        for(int i=0; i < n; i++)  adj.add(new ArrayList<Integer>());
        for(int i=0; i < graph.length; i++)  adj.get(graph[i][0]).add(graph[i]
[1]);
        //队列node_queue存储当前以及访问过尚未扩展的节点
        Queue<Integer> node_queue = new LinkedList<Integer>();
        boolean [] visited = new boolean[n];
        node_queue.add(start);
        while(!node_queue.isEmpty()){
            int cur_vert = node_queue.poll();
            //扩展当前节点的邻居
```

```
            for(int j=0;j<adj.get(cur_vert).size();j++){
                int vert=adj.get(cur_vert).get(j);
                //如果该邻居没有被访问过
                if(!visited[vert]){
                    visited[vert] = true;
                    if(vert == target) return true;
                    node_queue.add(vert);
                }
            }
        }
        return false;
    }
```

优化解法

Java代码

```
class Solution {
    public ArrayList<ArrayList<Integer> > adj;
    public boolean findWhetherExistsPath_recursive(int n, int start, int target,
boolean[] visited) {
        if(start == target)  return true;
        //遍历start出发所有边，进行dfs
        for(int j=0;j<adj.get(start).size();j++){
            int vert=adj.get(start).get(j);
            if(!visited[vert]){
                visited[vert] = true;
                boolean result = findWhetherExistsPath_recursive(n, vert,
target, visited);
                if(result) return true;
            }
        }
        return false;
    }
    public boolean findWhetherExistsPath(int n, int[][] graph, int start, int
target) {
        boolean []visited = new boolean[n];
        visited[start]=true;
        //建立邻接表 adj
        adj  = new ArrayList<ArrayList<Integer>>(n);
        for(int i=0; i < n; i++)  adj.add(new ArrayList<Integer>());
        for(int i=0; i < graph.length; i++)  adj.get(graph[i][0]).add(graph[i]
[1]);
        //开始递归
        return findWhetherExistsPath_recursive(n, start, target, visited);
    }
}
```

C++代码

```
class Solution {
public:
    bool findWhetherExistsPath_recursive(vector<vector<int>>& adj, vector<bool>&
visited, int start, int target){
        if (start == target) return true;
```

```
        visited[start] = true;
        for (auto vertex : adj[start])
            // 若当前顶点未遍历且以当前开始递归到返回真
            if (!visited[vertex] && findWhetherExistsPath_recursive(adj,
visited, vertex, target))
                return true;
        return false;
    }
    bool findWhetherExistsPath(int n, vector<vector<int>>& graph, int start, int
target) {
        vector<vector<int>> adj(n); //建立邻接表
        for (auto verPair : graph)
            adj[verPair[0]].push_back(verPair[1]);
        vector<bool> visited(n);
        return findWhetherExistsPath_recursive(adj, visited, start, target);
    }
};
```

Python代码

```
class Solution:
    def findWhetherExistsPath(self, n: int, graph: List[List[int]], start: int,
target: int) -> bool:
        #建立邻接表，使用set删除平行边
        adj = collections.defaultdict(set)
        for u, v in graph:
            adj[u].add(v)
        visited = set()

        def dfs(i):
            if i == target:
                return True
            visited.add(i)
            for j in adj[i]:
                if j not in visited:
                    if dfs(j):
                        return True
            return False

        return dfs(start)
```