深度优先搜索算法
（DFS）

广度优先搜索算法
（BFS）

在整个算法知识点中占比非常大；应用最多的地方是对图进行遍历（树也是图的一种）。

深度优先搜索 / DFS

力扣 LeetCode × 拉

DFS 解决什么问题

DFS解决的是连通性的问题，即给定两一个起始点（或某种起始状态）和一个终点（或某种最终状态），判断是否有一条路径能从起点连接到终点。

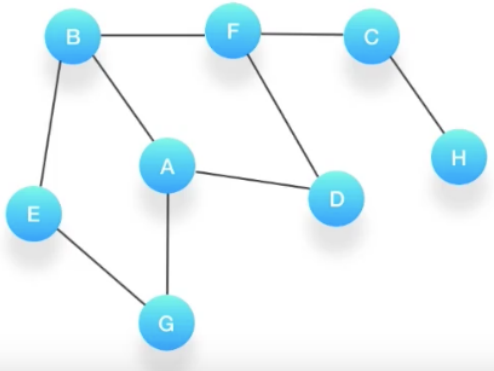
很多情况下，连通的路径有很多条，只需要找出一条即可，DFS 只关心路径存在与否，不在乎其长短。

算法的思想

从起点出发，选择一个可选方向不断向前，直到无法继续为止

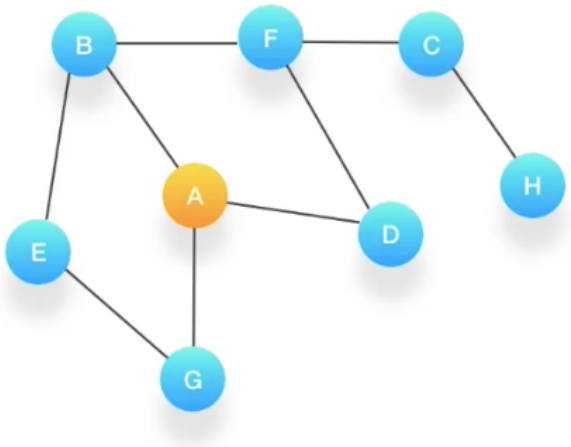
然后尝试另外一种方向，直到最后走到终点

假设我们有这么一个图，里面有A, B, C, D, E, F, G, H 8个顶点，
点和点之间的联系如下图所示：



如何对这个图进行深度优先的遍历呢?

1. 深度优先遍历必须依赖栈（Stack）这个数据结构
2. 栈的特点是后进先出（LIFO）



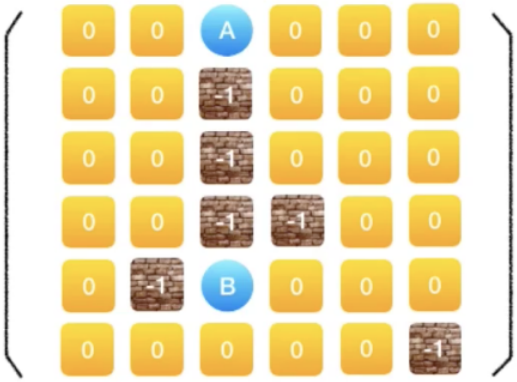
结果： A

栈

例题分析

1. 给定一个二维矩阵代表一个迷宫
2. 迷宫里面有通道，也有墙壁，有墙壁的地方不能通过
3. 通道由数字0表示，而墙壁由-1表示
4. 现在问能不能从A点走到B点?

从A开始走的话，可以有很多条路径通往B，例如下面两种。




```
boolean dfs(int maze[][], int x, int y) {

  if (x == B[0] && y == B[1]) {
    return true;
  }

  maze[x][y] = -1;

  for (int d = 0; d < 4; d++) {
    int i = x + dx[d], j = y + dy[d];

    if (isSafe(maze, i, j) && dfs(maze, i, j)) {
      return true;
    }
  }

  return false;
}
```

Step 1：判断是否抵达了目的地B，是则立即返回

Step 2：标记当前点已经被访问过了

Step 3：在规定的四个方向上进行尝试

Step 4：如果有一条路径被找到了，则返回true

Step 5：尝试了所有可能还没找到B，则返回false

```java
boolean dfs(int maze[][], int x, int y) {

  Stack<Integer[]> stack = new Stack<>();
  stack.push(new Integer[] {x, y});
  maze[x][y] = -1;

  while (!stack.isEmpty()) {
    Integer[] pos = stack.pop();
    x = pos[0]; y = pos[1];

    if (x == B[0] && y == B[1]) {
      return true;
    }

    for (int d = 0; d < 4; d++) {
      int i = x + dx[d], j = y + dy[d];

      if (isSafe(maze, i, j)) {
        stack.push(new Integer[] {i, j});
        maze[i][j] = -1;
      }
    }
  }
  return false;
}
```

Step 1：创建一个Stack，用来将要被访问的点压入以及弹出

Step 2：将起始点压入Stack，并标记它被访问过

Step 3：只要Stack不为空，就不断地循环，处理每个点

Step 4：从堆栈取出当前要处理的点

Step 5：判断是否抵达了目的地B，是则返回true

Step 6：如果不是目的地，就从四个方向上尝试

Step 7：将各个方向上的点压入堆栈，并把标记为访问过

Step 8：尝试了所有可能还没找到B，则返回false

DFS复杂度分析

由于DFS是图论里的算法，分析利用DFS解题的复杂度时，应当借用图论的思想，图有两种表示方式：

- 邻接表（图里有V个顶点，E条边）

  访问所有顶点的时间为O(V)，查找所有顶点的邻居的时间为O(E)，所以总的时间复杂度是O(V+E)

- 邻接矩阵（图里有V个顶点，E条边）

  查找每个顶点的邻居需要O(V)的时间，所以查找整个矩阵的时候需要O(V^2)的时间

利用DFS在迷宫里找一条路径

由于迷宫是用矩阵表示，所以假设它是一个M行N列邻接矩阵

- 时间复杂度为O(M × N)

  因为一共有M × N个顶点，所以时间复杂度就是O(M × N)

- 空间复杂度为O(M × N)

  DFS需要堆栈来辅助，在最坏情况下所有顶点都被压入堆栈，所以它的空间复杂度是O(V)，即O(M × N)

暴力解题法

找出所有路径，然后比较它们的长短，找出最短的那个
如果硬要使用DFS去找最短的路径，我们必须尝试所有的可能
DFS解决的只是连通性问题，不是用来求解最短路径问题的

优化解题思路

一边寻找目的地，一边记录它和起始点的距离（也就是步数）
当发现从某个方向过来所需要的步数更少，则更新到这个点的步数
如果发现步数更多，则不再继续尝试

情况一：从某方向到达该点所需要的步数更少则更新

情况二：从各方向到达该点所需要的步数都更多则不再尝试

```
void solve(int maze[][]) {

  for (int i = 0; i < maze.length; i++) {
    for (int j = 0; j < maze[0].length; j++) {
    if (maze[i][j] == 0 && !(i == A[0] && j == A[1])) {
        maze[i][j] = Integer.MAX_VALUE;
      }
    }
  }

  dfs(maze, A[0], A[1]);

  if (maze[B[0]][B[1]] < Integer.MAX_VALUE) {
    print("Shortest path count is: " +
maze[B[0]][B[1]]);
  } else {
    print("Cannot find B!");
  }
}
```

Step 1： 除A之外的所有0都用Max替换

Step 2： 对矩阵进行DFS遍历

Step 3： 判读是否抵达B点

```
void dfs(int maze[][], int x, int y) {

  if (x == B[0] && y == B[1]) return;

  for (int d = 0; d < 4; d++) {
    int i = x + dx[d], j = y + dy[d];

    if (isSafe(maze, i, j) && maze[i][j] > maze[x][y] + 1) {

      maze[i][j] = maze[x][y] + 1;
      dfs(maze, i, j);
    }
  }
}
```

Step 1： 判断是否找到到目的地B

Step 2： 从四个方向上进行尝试

Step 3： 下个点步数是否大于当前点步数+1

Step 4： 是则更新下个点步数，并继续DFS

矩阵的最终结果：

```
2,  1,  A,  1,  2,  3
3,  2, -1,  2,  3,  4
4,  3, -1,  3,  4,  5
5,  4, -1, -1,  5,  6
6, -1,  8,  7,  6,  7
7,  8,  9,  8,  7, -1
```

广度优先搜索 / BFS

广度优先搜索简称BFS

广度优先搜索一般用来解决最短路径的问题

广度优先的搜索是从起始点出发，一层一层地进行

每层当中的点距离起始点的步数都是相同的

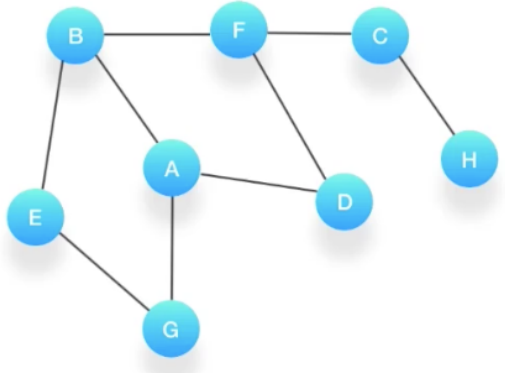
双端BFS

同时从起始点和终点开始进行的 广度优先的搜索称为双端BFS

双端BFS可以大大地提高搜索的效率

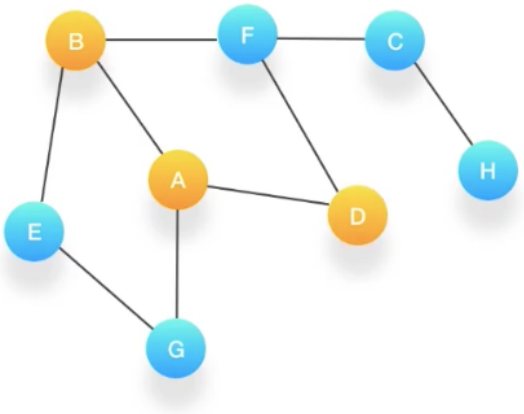
例如，想判断社交应用程序中两个人之间需要经过多少朋友介绍才能互相认识

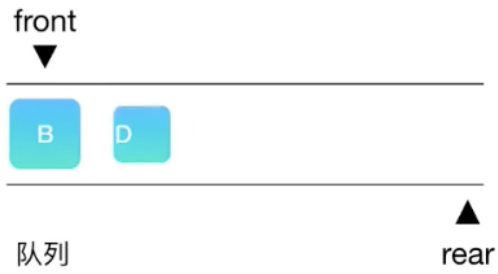
假设我们有这么一个图，里面有A, B, C, D, E, F, G, H 8个顶点，
点和点之间的联系如下图所示：



如何对这个图进行广度优先的遍历呢?

1. 广度优先遍历需要借用的数据结构是队列（Queue）
2. 队列特点是先进先出（FIFO）





结果：A

如何运用广度优先搜索在迷宫中寻找最短的路径?

```
void bfs(int[][] maze, int x, int y) {

  Queue<Integer[]> queue = new LinkedList<>();
  queue.add(new Integer[] {x, y});

  while (!queue.isEmpty()) {

    Integer[] pos = queue.poll();
    x = pos[0]; y = pos[1];

    for (int d = 0; d < 4; d++) {
      int i = x + dx[d], j = y + dy[d];

      if (isSafe(maze, i, j)) {
        maze[i][j] = maze[x][y] + 1;
        queue.add(new Integer[] {i, j});

        if (i == B[0] && j == B[1]) return;
      }
    }
  }
}
```

Step 1：创建一个队列，将起始点加入队列中

Step 2：只要队列不为空，就一直循环下去

Step 3：从队列中取出当前要处理的点

Step 4：在四个方向上进行BFS搜索

Step 5：判断一下该方向上的点是否已经访问过了

Step 6：被访问过，则记录步数，并加入队列中

Step 7：找到目的地后立即返回

BFS复杂度分析

由于BFS是图论里的算法，分析利用BFS解题的复杂度时，应当借用图论的思想，图有两种表示方式：

- 邻接表（图里有V个顶点，E条边）

  每个顶点都需要被访问一次，因此时间复杂度是O(V)，在访问每个顶点的时候，与它相连的顶点（也就是每条边）也都要被访问一次，所以加起来就是O(E)，因此整体时间复杂度就是O(V+E)。

- 邻接矩阵（图里有V个顶点，E条边）

  由于有V个顶点，每次都要检查每个顶点与其他顶点是否有联系，因此时间复杂度是O(V^2)

利用BFS在迷宫里找一条路径

由于迷宫是用矩阵表示，所以假设它是一个M行N列邻接矩阵

- 时间复杂度为O(M × N)

  因为一共有M × N个顶点，所以时间复杂度就是O(M × N)

- 空间复杂度为O(M × N)

  BFS需要借助一个队列，所有顶点都要进入队列一次，从队列弹出一次

  在最坏的情况下，空间复杂度是O(V)，即O(M × N)

从A走到B最多允许打通3堵墙，求最短路径的步数

暴力解题法

首先枚举出所有拆墙的方法，假设一共有K堵墙在当前的迷宫里，现在最多允许拆3堵墙，意味着可以选择不拆、只拆一堵墙、两堵墙或三堵墙，那么一共有这么多种组合方式：

$$C_k^0 + C_k^1 + C_k^2 + C_k^3 = 1 + k + \frac{K(K-1)}{2} + \frac{K(K-1)(K-2)}{2}$$

在这么多种情况下分别进行BFS，整体的时间复杂度就是O($n^2 \times K^w$)，从中找到最短的那条路径

如何将BFS的数量减少?

在不允许打通墙的情况下，只有一个人进行BFS搜索，时间复杂度是n^2

允许打通一堵墙的情况下，分身为两个人进行BFS搜索，时间复杂度是$2n^2$

允许打通两堵墙的情况下，分身为三个人进行BFS搜索，时间复杂度是$3n^2$

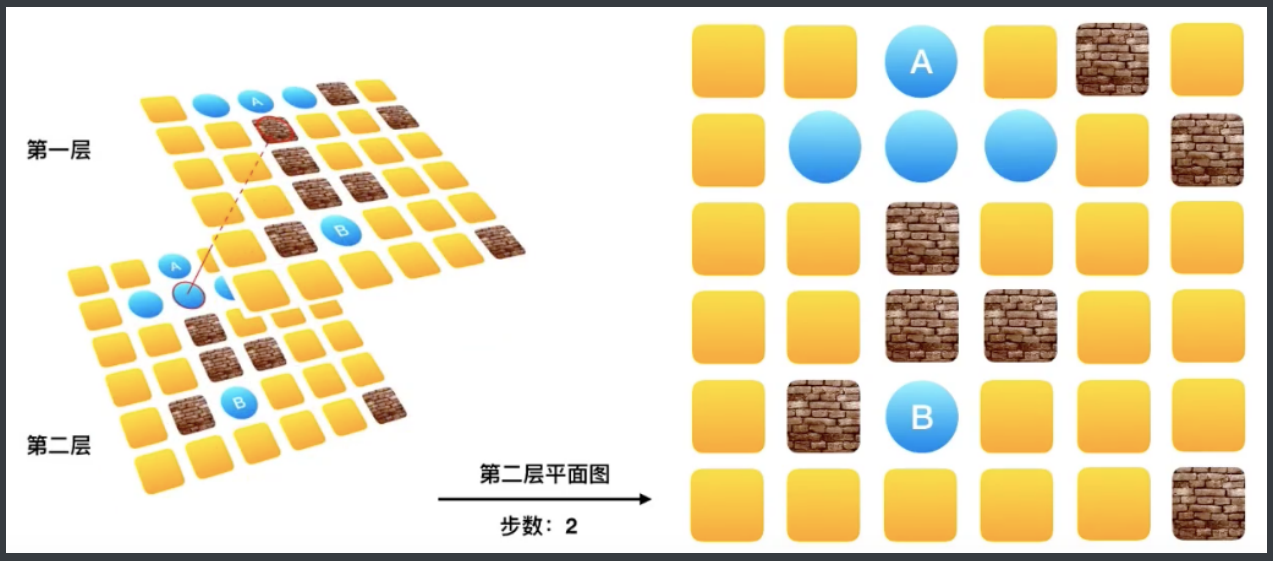
允许打通三堵墙的情况下，分身为四个人进行BFS搜索，时间复杂度是$4n^2$

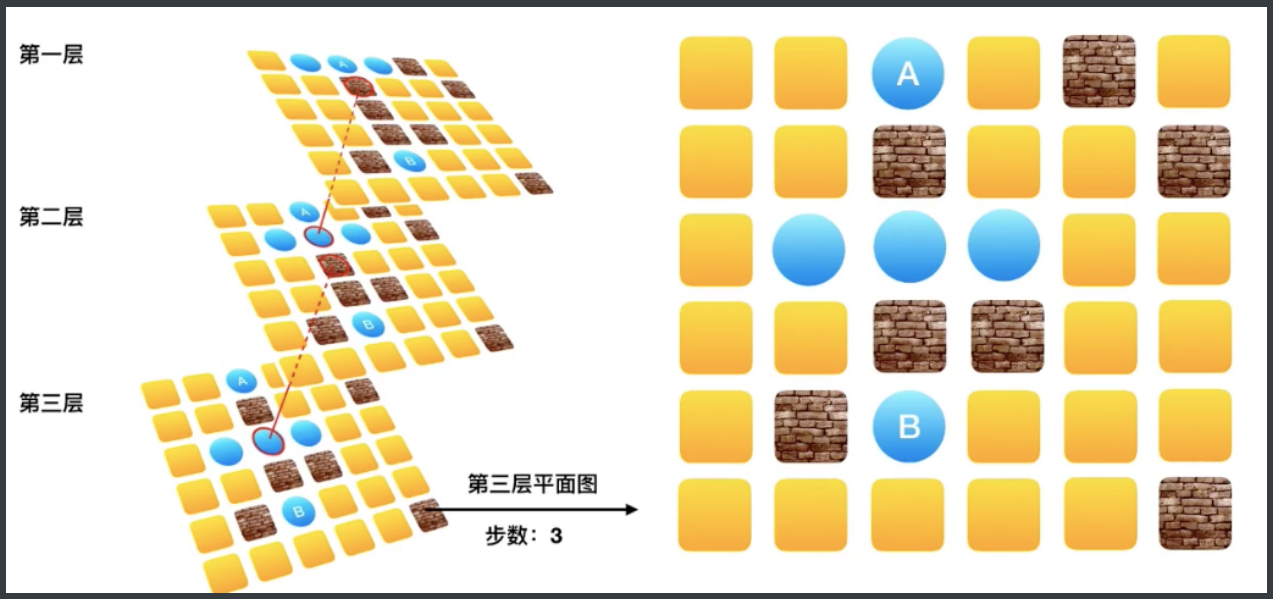
关键问题

如果第一个人又遇到了一堵墙，那么他是否需要再次分身呢？不能

第一个人怎么告诉第二个人可以去访问这个点呢？把这个点放入到队列中就好了

如何让4个人在独立的平面里搜索呢？利用一个三维矩阵记录每个层面里的点即可






```
int bfs(int[][] maze, int x, int y, int w) {

  int steps = 0, z = 0;

  Queue<Integer[]> queue = new LinkedList<>();
  queue.add(new Integer[] {x, y, z});
  queue.add(null);

  boolean[][][] visited = new boolean[N][N][w + 1];
  visited[x][y][z] = true;

  while (!queue.isEmpty()) {
    Integer[] pos = queue.poll();

    if (pos != null) {
      x = pos[0]; y = pos[1]; z = pos[2];

      if (x == B[0] && y == B[1]) {
        return steps;
      }
```

Step 1： 初始化变量，steps记录步数，z记录Level
用一个队列来存储要处理的各个层面的点
三维布尔数组记录各层面点的被访问情况

Step 2： 只要队列不为空就一直循环下去

Step 3： 取出当前点，如遇目的地则返回步数

```
      for (int d = 0; d < 4; d++) {
        int i = x + dx[d], j = y + dy[d];
        if (!isSafe(maze, i, j, z, visited)) {
          continue;
        }
        int k = getLayer(maze, w, i, j, z);
        if (k >= 0) {
       visited[i][j][k] = true;
          queue.add(new Integer[] {i, j, k});
        }
      }
    } else {
      steps++;
      if (!queue.isEmpty()) {
        queue.add(null);
      }
    }
  }

  return -1;
}
```

Step 1： 初始化变量，steps记录步数，z记录Level
用一个队列来存储要处理的各个层面的点
三维布尔数组记录各层面点的被访问情况

Step 2： 只要队列不为空就一直循环下去

Step 3： 取出当前点，如遇目的地则返回步数

Step 4： 如果不是，则朝四个方向尝试下一步

Step 5： getLayer函数判断是否遇到可打通的墙壁

Step 6： 若当前点是null，则继续下一步

getLayer的思想

如果当前遇到的是一堵墙，判断所打通的墙壁个数是否已经超出规定，如果没有，就将其打通，否则返回-1。

```
int getLayer(int[][] maze, int w, int x,
int y, int z) {
    if (maze[x][y] == -1) {
      return z < w ? z + 1 : -1;
    }

    return z;
  }
```