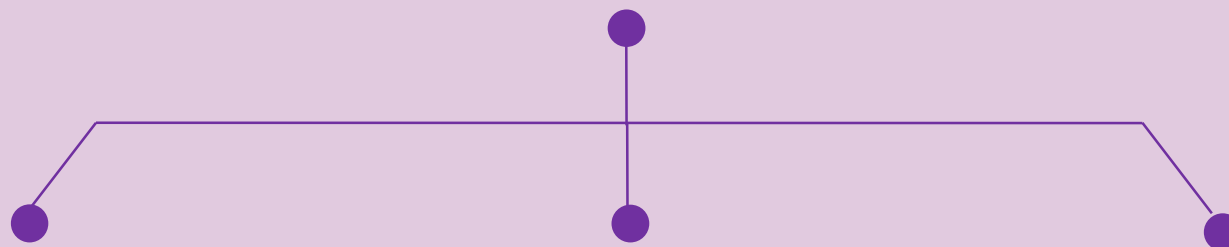


# 搜索算法及应用

---

# 目 录



1

算法概念

2

算法实现

3

算法应用

# PART.1

基本概念

# 搜索算法

---

## 定义

- 搜索算法是有目的的穷举一个问题的部分或所有的可能情况，从而求出问题的解的一种方法。
- 搜索算法实际上是根据初始条件和扩展规则构造一棵“搜索树”并寻找符合目标状态的节点的过程。搜索算法的使用第一步在于搜索树的建立，完成搜索的过程就是找到一条从根结点到目标结点的路径，类似于图或树的遍历。

# 搜索算法

## 分类

- 盲目搜索：无信息搜索，将所有可能性罗列出来，按预定的搜索策略进行搜索，在其中寻找答案。适用于状态空间图是树结构的一类问题。

深度优先搜索depth first search ( DFS )     $\xrightarrow{\text{评估函数}}$     IDA\*算法  
广度优先搜索breadth first search ( BFS )     $\longrightarrow$     A\*算法

- 启发式搜索：选择最有希望的节点加以扩展，在状态空间中对每一个搜索的位置进行评估，得到最好的位置，再从这个位置进行搜索直到目标。搜索中加入了启发性的信息，用于指导搜索。

A\*算法

IDA\*算法

# 搜索算法

---

## 求解过程

- 状态 ( state ) 是对一个问题在某一时刻进展情况的数学描述。
- 状态空间 ( state space ) 是一个表示某问题全部可能状态及其关系的图，包含三个集合，即问题初始状态集合 $S$ ，操作符集合 $F$ 以及目标状态集合 $G$ 。实际上状态空间图由节点及连接节点的边构成，结点所有的状态，边对应状态转移。
- Step1 :  
将问题表示为状态空间图，解表示为目标节点  
Step2 :  
选择合适的搜索算法求解从初始节点到目标节点的路径

# 递归

---

- 递归是一种直接或者间接调用自身函数或者方法的算法。
- 递归算法转换成非递归算法的两种方式
  - ( 1 ) 将递归结构用循环结构来替代
  - ( 2 ) 使用栈保存中间结果
- 求阶乘

```
int factorial(int n){  
    if(n==1)  
        return 1;  
    else  
        return factorial(n-1)*n;  
}
```

```
int factorial(int n){  
    int fac=1;  
    for (int i=1;i<=n;i++)  
        fac=fac*i;  
    return fac;  
}
```

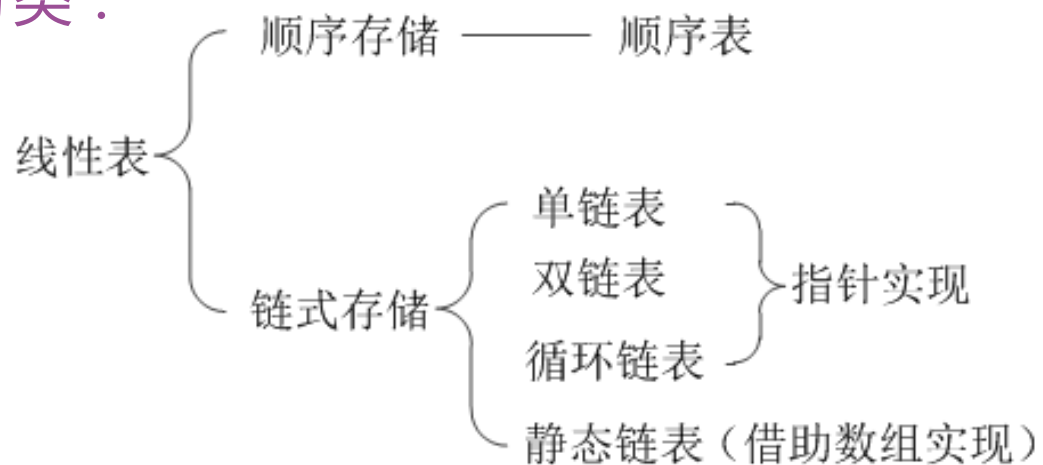
# 线性表

---

- 定义:

n个类型相同数据元素的有限序列，记为： $L=(a_1,a_2,...,a_n)$

- 分类：



- 操作：

InsList ( L , i , e )

GetData ( L , i )

DelList ( L , i , e )

EmptyList ( L )



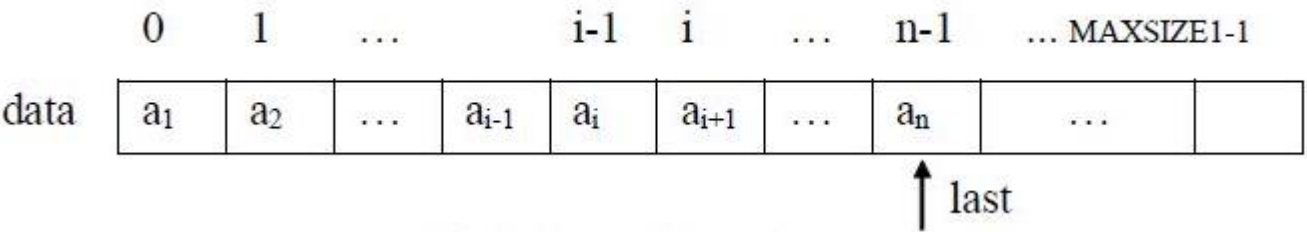
# 线性表

## 顺序表

```
#define MAXLENGTH 20
```

```
struct sequencelist  
{  
    int data[MAXLENGTH];  
    int length;  
};
```

大小可变，连续存储

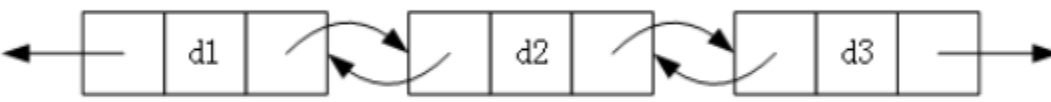


## 链表

单链表



双链表

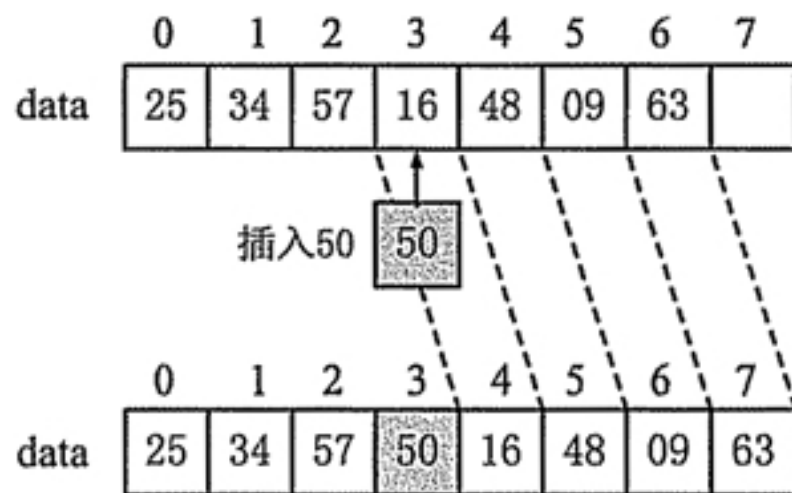


循环链表

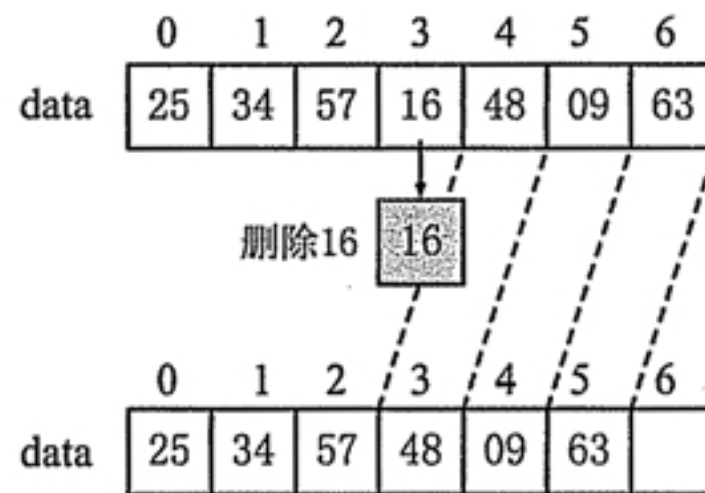


# 线性表

## 顺序表的插入删除



(a) 插入新元素示例



(b) 删除表中元素示例

# 栈

- 定义:

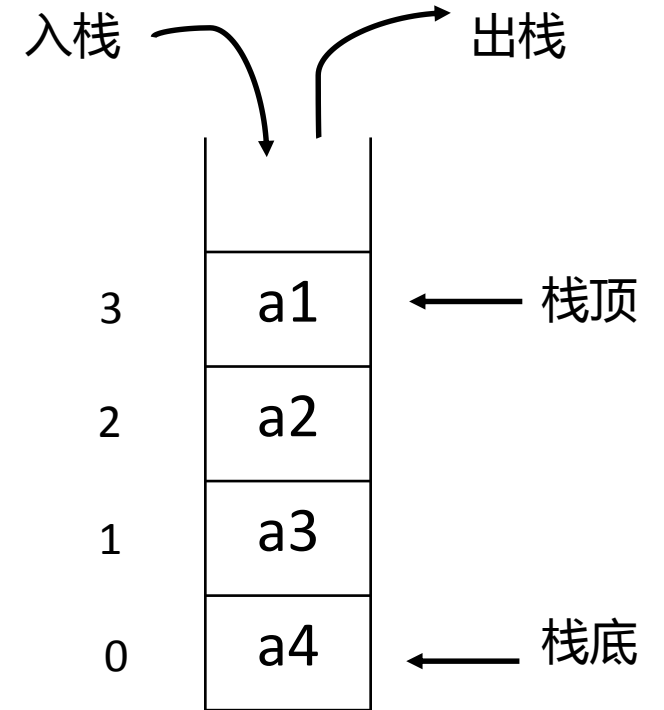
一种限定性线性表，仅允许在表的一端进行插入和删除运算。这一端被称为栈顶，另一端称为栈底。

- 特点：

LIFO，即后进先出（Last in first out）

- 实现

```
#define MAXLENGTH 20
struct Stack
{
    int data[MAXLENGTH];
    int top; //存放栈顶元素下标
};
```



# 栈

---

- 操作：

**push** 在最顶层加入数据

```
void push(int x)//进栈
{
    if (top == MaxSize - 1)
        cout << "栈上溢出！" << endl;
    else
    {
        ++top;
        data[top] = x;
    }
}
```

**pop** 返回并移除最顶层的数据

```
int pop()//出栈
{
    int tmp = 0;
    if (top == -1)
        cout << "栈已空！" << endl;
    else
        tmp = data[top--];
    return tmp;
}
```

# 栈

- 操作：

`top` 返回最顶层数据的值，但不移除它。

`empty` 返回一个布尔值，表示当前stack是否为空栈

```
int top()//获得栈顶元素
```

```
{  
    int tmp = 0;  
    if (top == -1)  
        std::cout << "栈空！" << std::endl;  
    else  
        tmp = data[top];  
    return tmp;  
}
```

```
bool empty()//判断栈为空
```

```
{  
    if (top == -1)  
        return true;  
    else  
        return false;  
}
```

- 使用：

C++、JAVA等语言的标准库已经准备了这一结构，可以使用`#include <stack>`调用栈的方法。

# 队列

- 定义:

一种限定性线性表，仅允许在表的一端插入，在另一端删除。

- 特点：

FIFO，即先进先出（First in first out）

- 使用：

C++、JAVA等语言的标准库已经准备了这一结构，可以使用`#include <queue>`调用栈的方法。

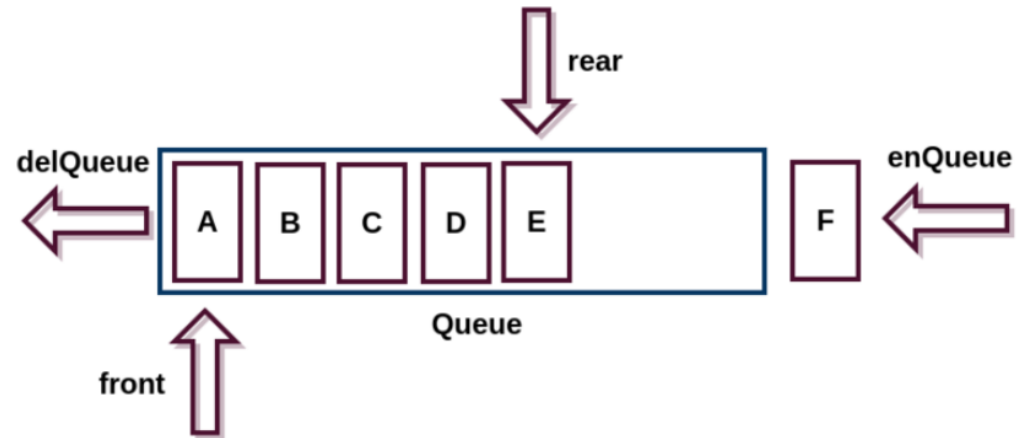
- 操作：

push 入队

pop 出队

front 读取队首元素

empty 判断队列是否为空



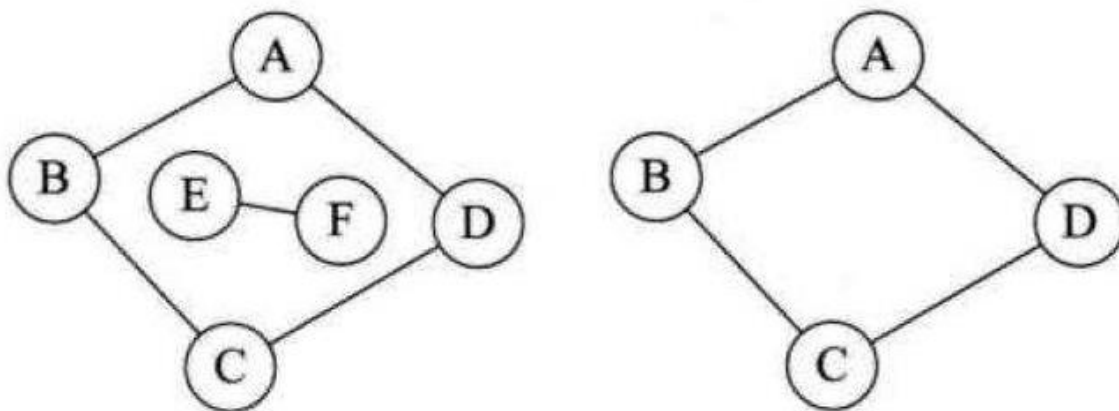
# 连通图

- 连通图

在一个无向图  $G$  中，若从顶点  $v_i$  到顶点  $v_j$  有路径相连，则称  $v_i$  和  $v_j$  是连通的。

- 连通分量

无向图  $G$  的一个极大连通子图称为  $G$  的一个连通分量。连通图只有一个连通分量，即其自身；非连通的无向图有多个连通分量。



# PART.2



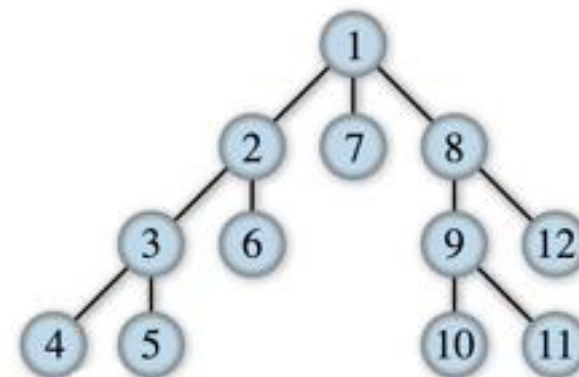
算法实现



# 深度优先搜索

## 思想

- 首先扩展最新产生的节点。
- 从初始节点开始。在其子节点中选择一个节点考察，若不是目标节点，则再在该子节点中选择一个子节点进行考察，一直向下搜索，直到某个节点既不是目标节点又不能继续扩展，之后选择兄弟节点进行考察。



# 深度优先搜索

## 算法描述

- 递归

- (1) 访问顶点 $v$ ;  $visited[v]=1$ ;
- (2)  $w$ =顶点 $v$ 的第一个邻接点;
- (3) while ( $w$ 存在)
  - if ( $w$ 未被访问)
  - 从顶点 $w$ 出发递归执行该算法;
  - $w$ =顶点 $v$ 的下一个邻接点;

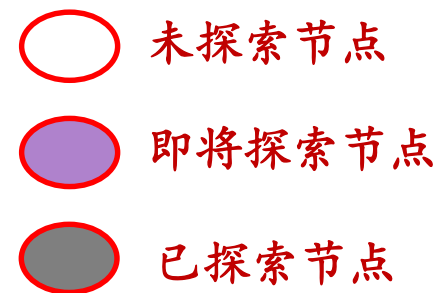
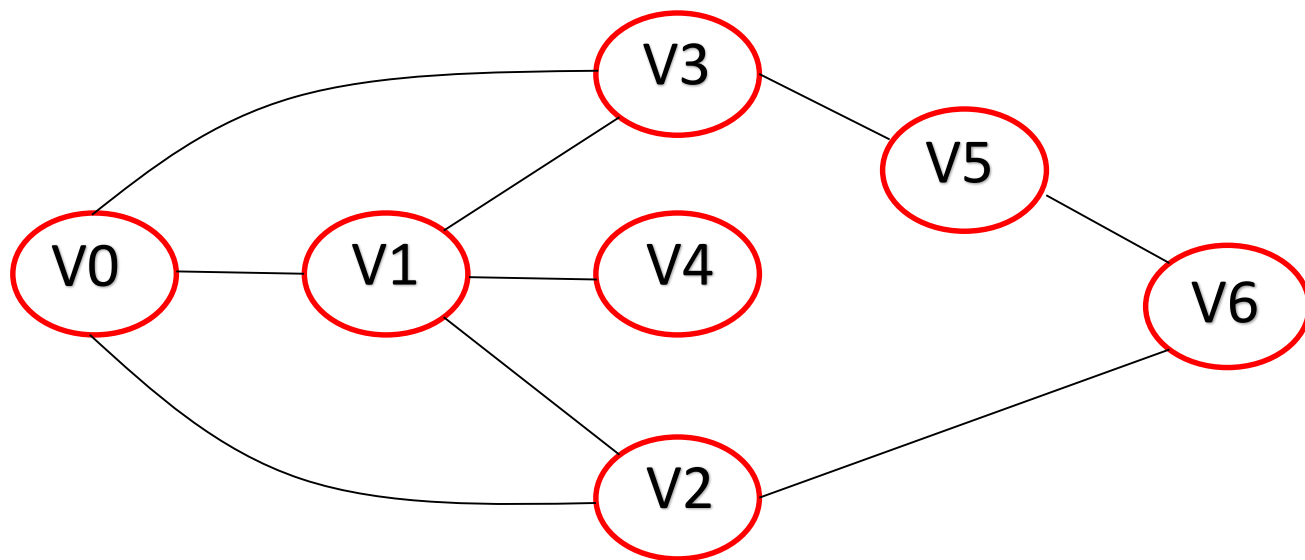
- 非递归

- (1) 栈 $S$ 初始化;  $visited[n]=0$ ;
- (2) 访问顶点 $v$ ;  $visited[v]=1$ ; 顶点 $v$ 入栈 $S$
- (3) while(栈 $S$ 非空)
  - $x$ =栈 $S$ 的顶元素(不出栈);
  - if(存在并找到未被访问的 $x$ 的邻接点 $w$ )
    - 访问 $w$ ;  $visited[w]=1$ ;
    - $w$ 进栈;
  - else
    - $x$ 出栈;

# 深度优先搜索

实现

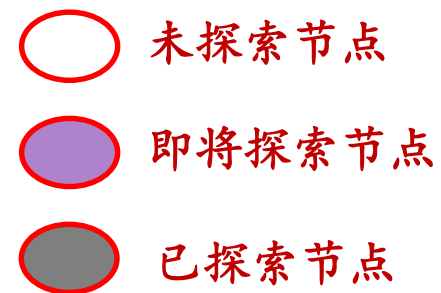
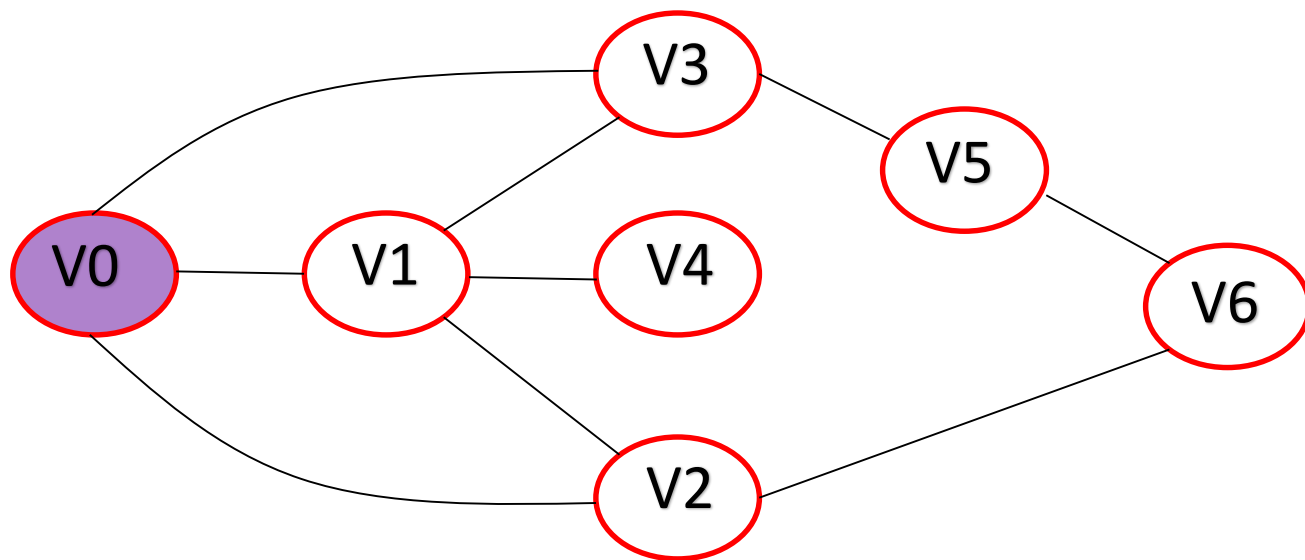
假设某连通图如下，要求寻找一条从V0至V6的路径。



# 深度优先搜索

实现

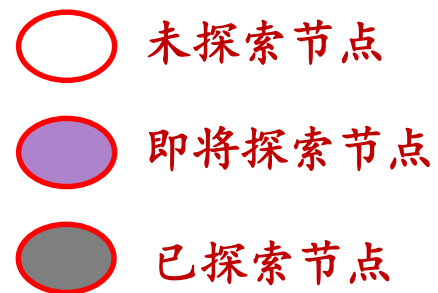
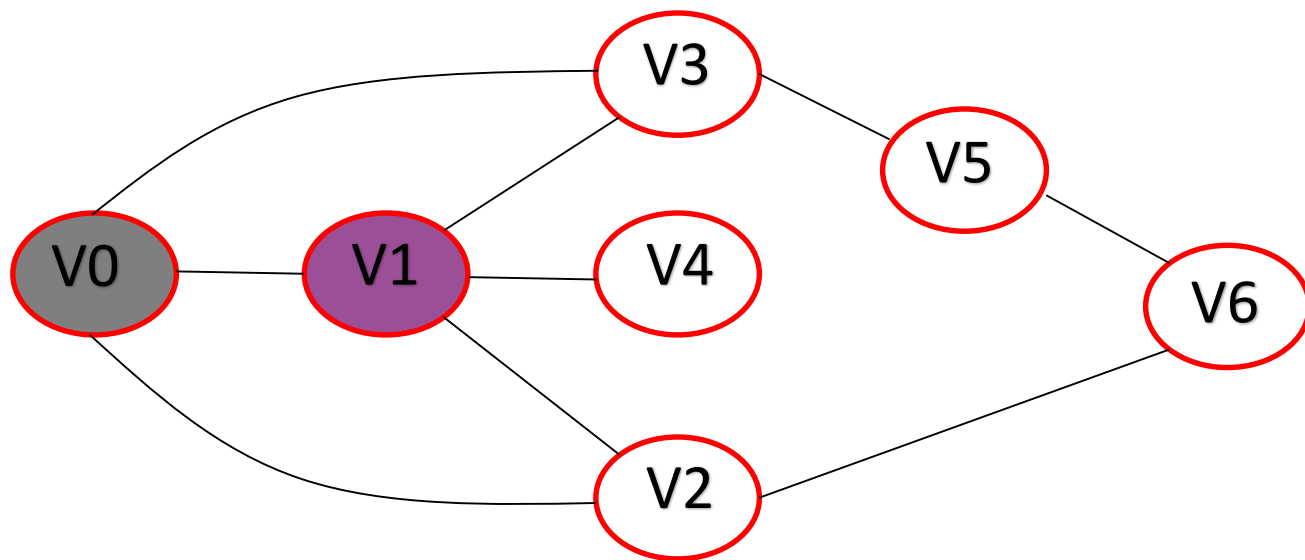
假设某连通图如下，要求寻找一条从V0至V6的路径。



# 深度优先搜索

实现

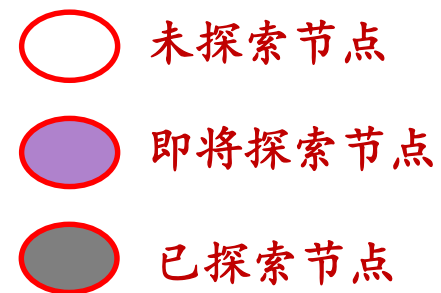
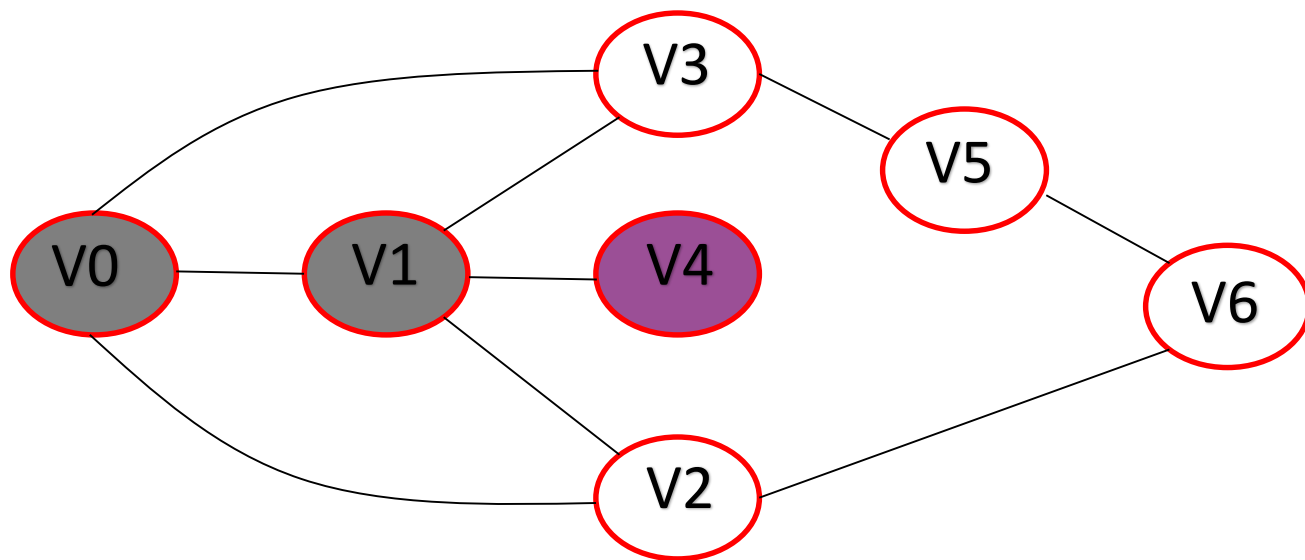
假设某连通图如下，要求寻找一条从V0至V6的路径。



# 深度优先搜索

实现

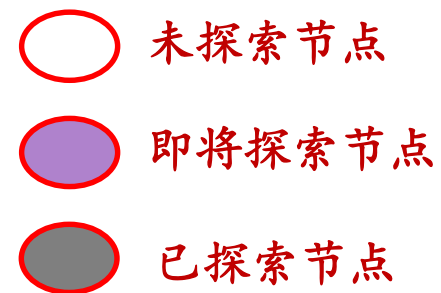
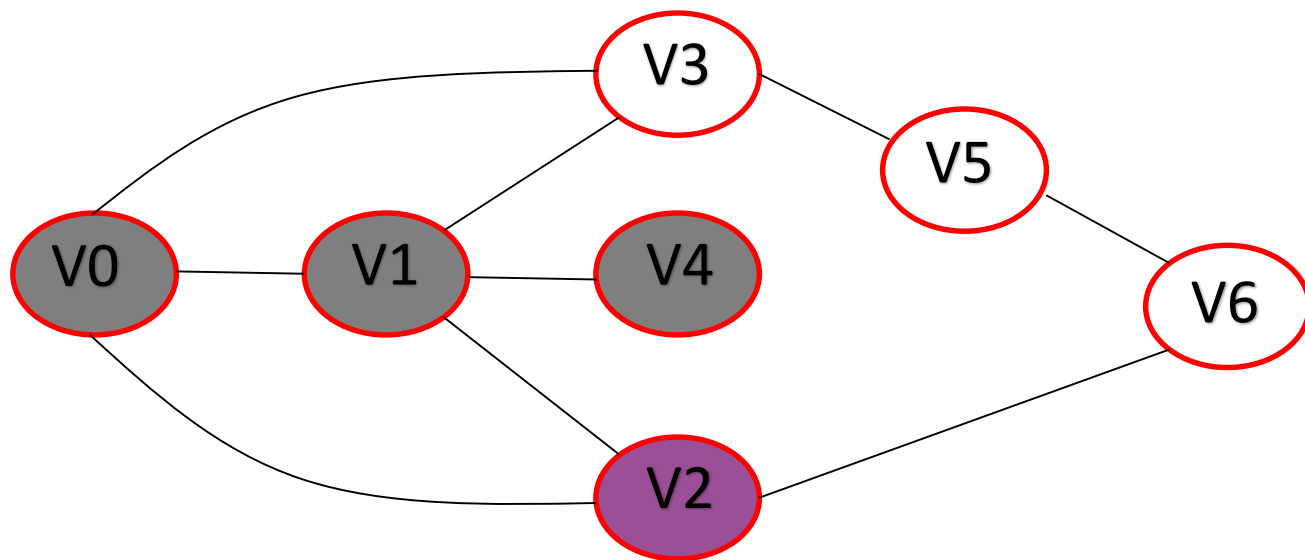
假设某连通图如下，要求寻找一条从V0至V6的路径。



# 深度优先搜索

实现

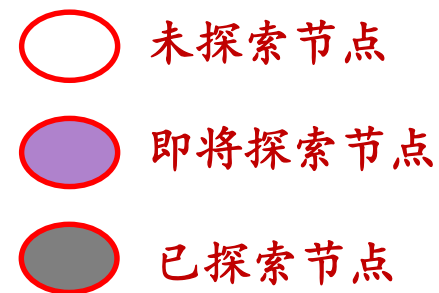
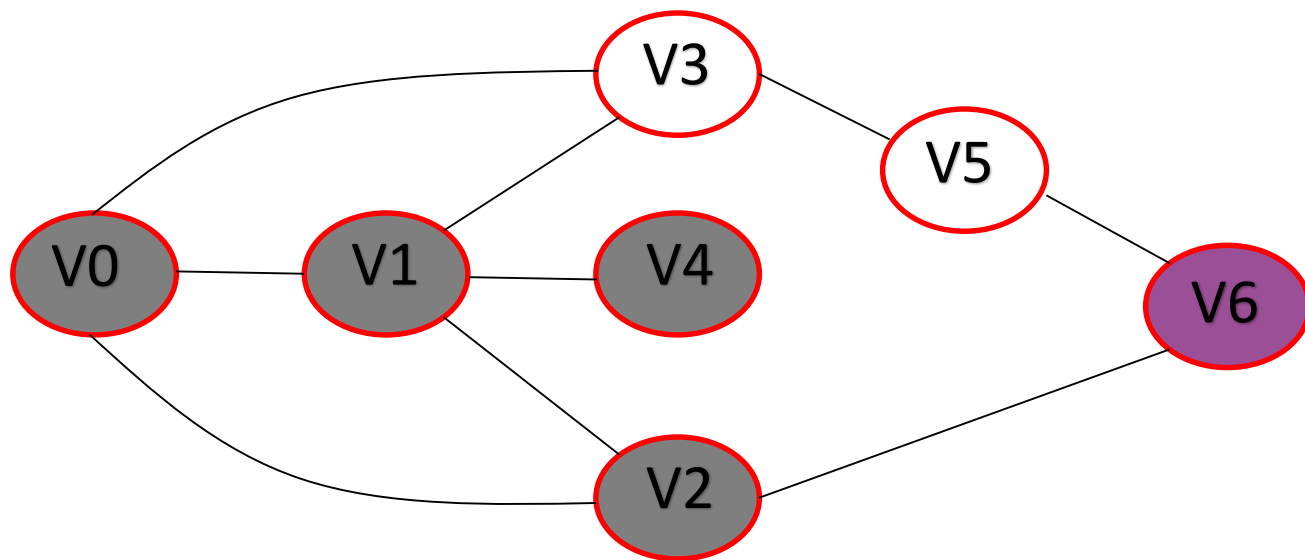
假设某连通图如下，要求寻找一条从V0至V6的路径。



# 深度优先搜索

实现

假设某连通图如下，要求寻找一条从V0至V6的路径。





# 深度优先搜索

---

- 有界深度优先搜索

对深度优先搜索引入搜索深度的界限，当搜索深度达到深度界限，而仍未出现目标节点时，换一个分支进行搜索。

即使应用了深度界限，所求的解并不一定是最短路径。

- 深度定义：

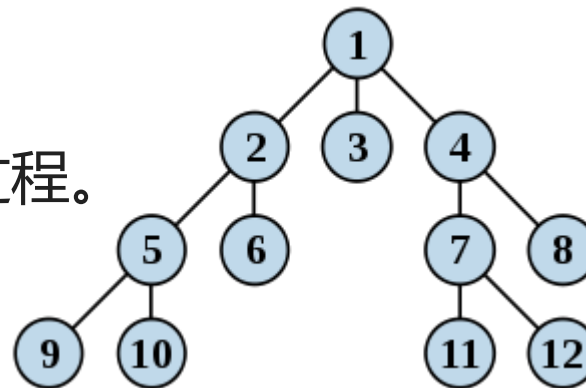
( 1 ) 起始节点深度为0；

( 2 ) 其他节点的深度等于父节点深度加1。

# 广度优先搜索

## 思想

- 搜索以接近起始节点的程度依次扩展，是一个分层搜索的过程。



- 1.访问顶点v0
- 2.依次访问v0的各个未被访问的邻接点
- 3.从这些邻接点出发，再访问它们的邻接点，直到所有的节点均被访问。

# 广度优先搜索

## 算法描述

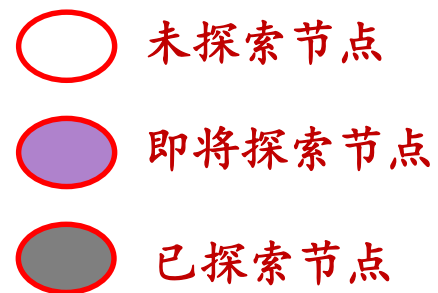
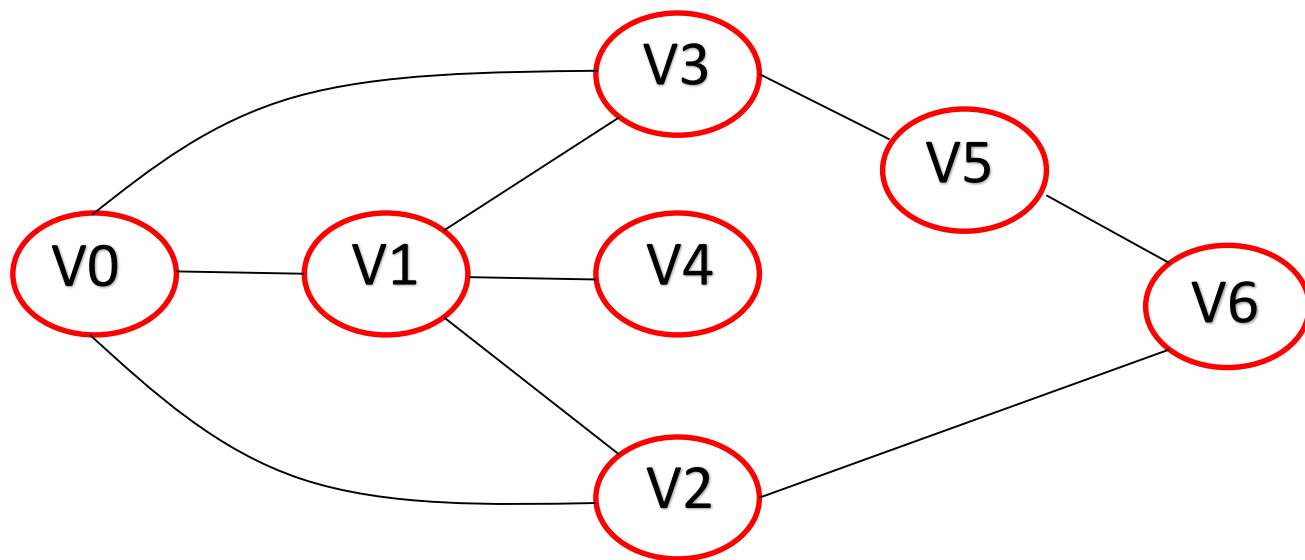
- 使用**队列**以保持遍历过的顶点顺序，以便按出队的顺序再去访问这些顶点的邻接顶点。

```
(1) 初始化队列Q; visited[n]=0;  
(2) 访问顶点v; visited[v]=1; 顶点v入队列Q;  
(3) while (队列Q非空)  
    v=队列Q的队首元素出队;  
    w=顶点v的第一个邻接点;  
    while (w存在)    //将所有邻接点入队  
        如果w未访问，则访问顶点w;  
        visited[w]=1;  
        顶点w入队列Q;  
        w=顶点v的下一个邻接点。
```

# 广度优先搜索

实现

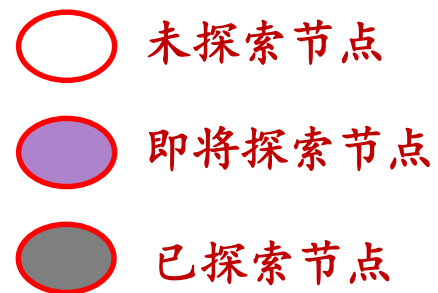
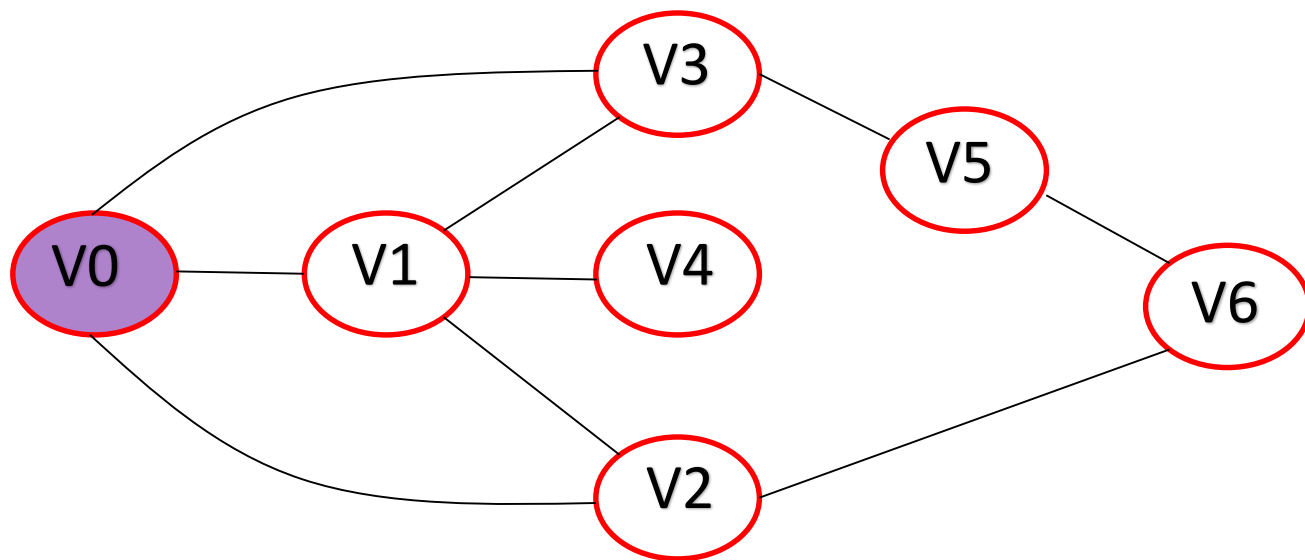
假设某连通图如下，要求寻找一条从V0至V6的最短路径。



# 广度优先搜索

实现

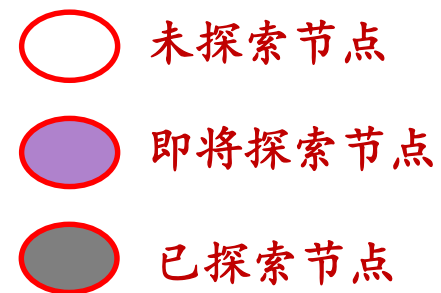
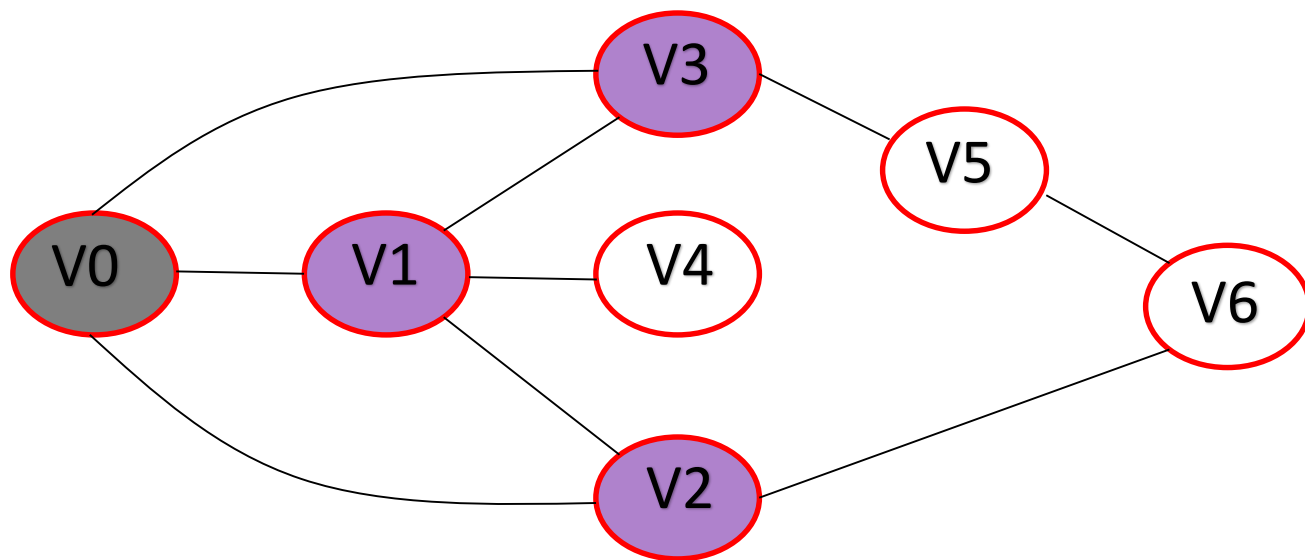
假设某连通图如下，要求寻找一条从V0至V6的最短路径。



# 广度优先搜索

实现

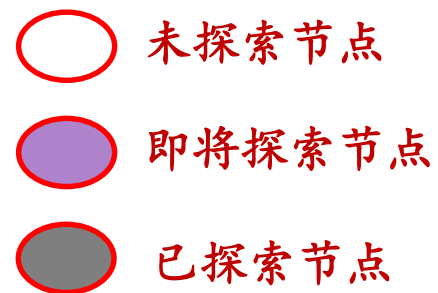
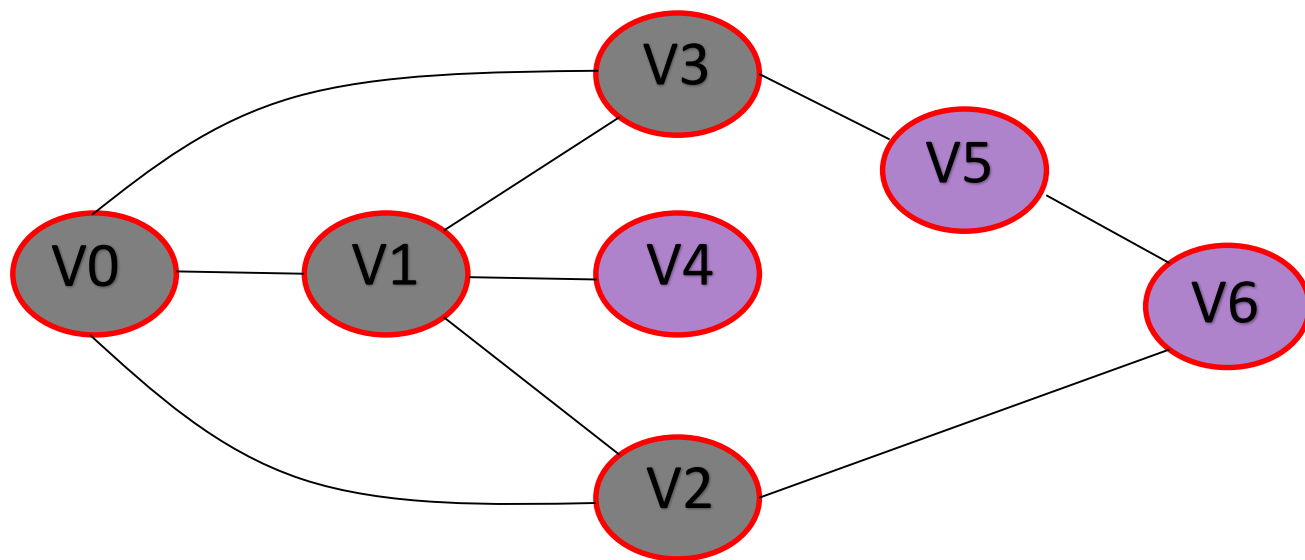
假设某连通图如下，要求寻找一条从V0至V6的最短路径。



# 广度优先搜索

实现

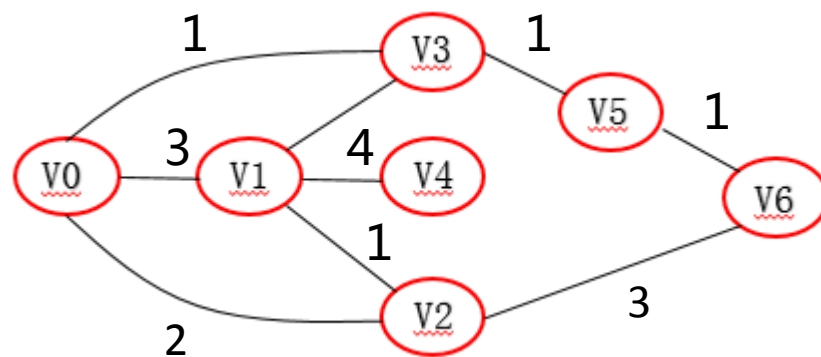
假设某连通图如下，要求寻找一条从V0至V6的最短路径。



# 代价树搜索

## 思想

- 有些问题不要求算符序列为最少的解，而是求具有某些特性的解，如代价最小的解。
- 边上有代价的树（图）称为代价树（图）。
- $g(x)$  表示从初始节点  $S_0$  到节点  $x$  的代价， $c(x_1, x_2)$  表示从父节点  $x_1$  到子节点  $x_2$  的代价，则  $g(x_2) = g(x_1) + c(x_1, x_2)$ 。
- 每次都选择代价最小的节点，也就是代价小的节点总是排在最前面。





# 剪枝

---

## 思想

- 剪枝属于算法优化范畴，通常应用在DFS 和 BFS 搜索算法中
  - 剪枝策略就是寻找过滤条件，提前减少不必要的搜索路径，剪去了搜索树中的某些“枝条”
  - 三原则
    - 1) 正确性 保证不丢失正确的结果
    - 2) 准确性 使不包含最优解的枝条尽可能多的被剪去
    - 3) 高效性 在优化与效率之间寻找一个平衡点
  - 分类
- 可行性剪枝和最优性剪枝

# PART.3



典型应用

# Find The Multiple

## 题目描述

- Given a positive integer  $n$ , write a program to find out a nonzero multiple  $m$  of  $n$  whose decimal representation contains only the digits 0 and 1. You may assume that  $n$  is not greater than 200 and there is a corresponding  $m$  containing no more than 100 decimal digits.
- 给出一个整数 $n$  , ( $1 \leq n \leq 200$ )。求出任意一个它的倍数 $m$  , 要求 $m$ 必须只由十进制的'0'或'1'组成。

### Sample Input

2 6 19 0



输入0结束

分别求2、6、19的倍数 $m$ ，且 $m$ 只由0和1组成

### Sample Output

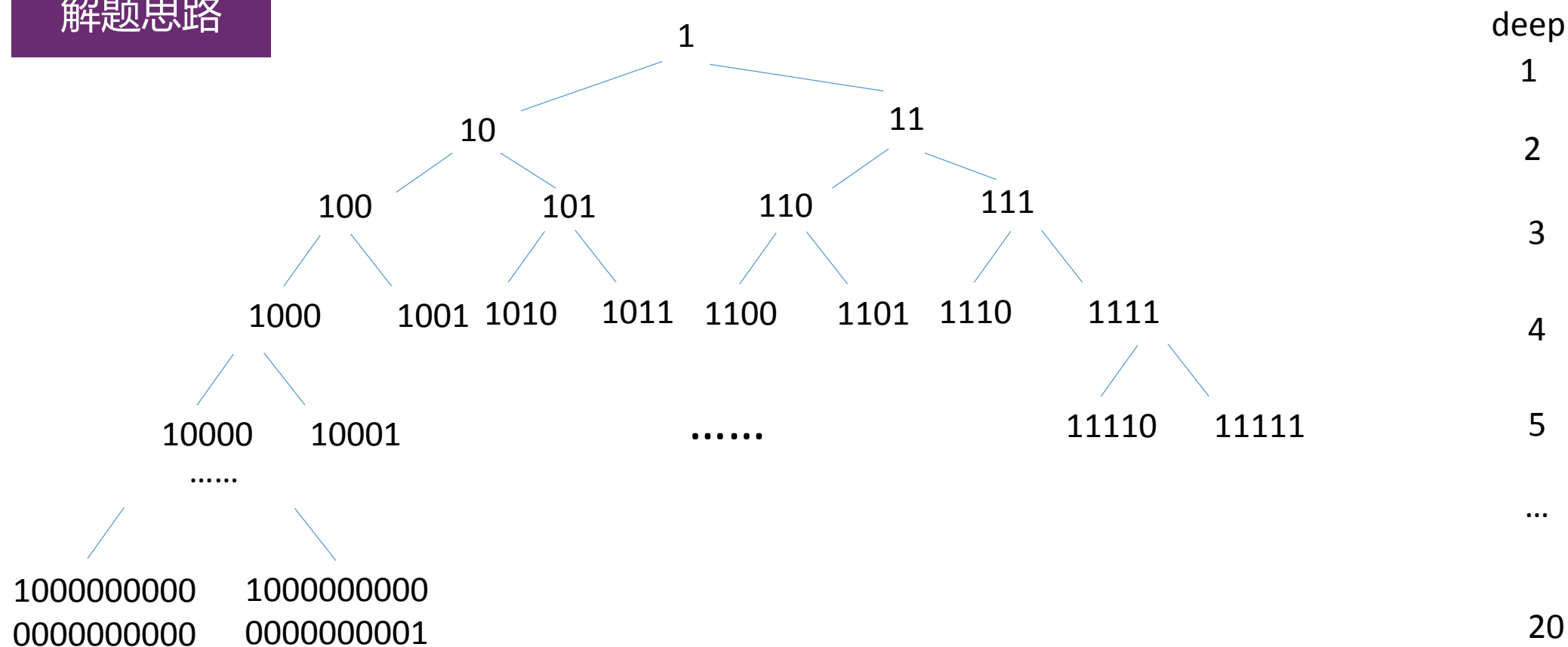
10

100100100100100100

11111111111111111111

# Find The Multiple

## 解题思路



使用深度优先搜索，从1开始，如果搜索到m则输出，否则搜索 $m \times 10$ 和 $m \times 10 + 1$ ，直到得出答案。

# Find The Multiple

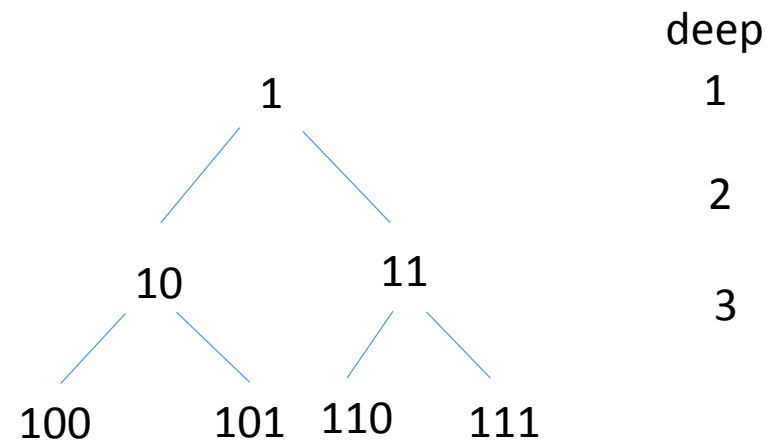
## 实现

//t:表示0和1组成的数, deep:位数, n:给定的数

```
void dfs(unsigned long long m, int deep, int n)
```

```
{  
    剪枝  
    if(ok || deep == 20) //退出的标志  
        return;  
    else if(m%n == 0) //可以整除则输出  
    {  
        cout<<m<<endl;  
        ok = 1;  
        return;  
    }  
    else  
    {dfs(m*10, deep+1, n); //个位是0  
     dfs(m*10+1, deep+1, n); //个位是1  
    }  
}
```

m=1, deep=0, n=4



# 棋盘问题

## 题目描述

- 在一个给定形状的棋盘（形状可能是不规则的）上面摆放棋子。要求摆放时任意两个棋子不能放在棋盘中的同一行或者同一列，请编程求解对于给定形状和大小的棋盘，摆放 $k$ 个棋子的所有可行的摆放方案 $C$ 。输入两个正整数， $n$  将在一个 $n*n$ 的矩阵内描述棋盘， $k$ 是摆放棋子的数目， $k \leq n$

### Sample Input

```
4 4
...#
..#.
.#..
#...
```

有4个待摆放的棋子

输入一个4\*4的棋盘

#表示棋盘 .表示边界

### Sample Output

1

# 棋盘问题

## 解题思路

按行递增的顺序来搜索，因此不可能出现同行的情况；对于同列的情况，设置了一个变量`col[]`，来保存列的访问状态，对于之前访问过的列，棋子是不能再放在这一列上的。

Sample Input

3 2

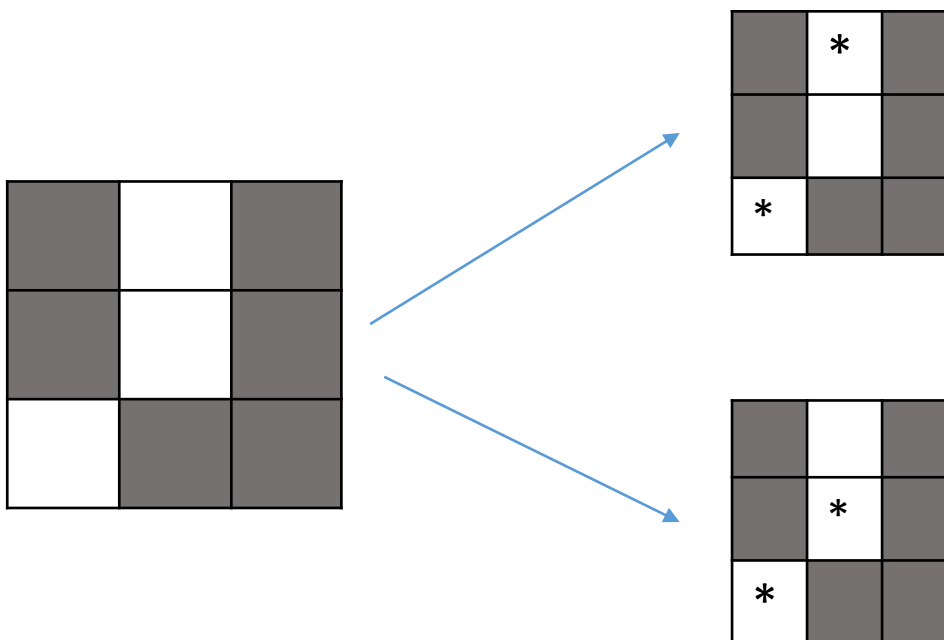
.#.

.#.

#..

Sample Output

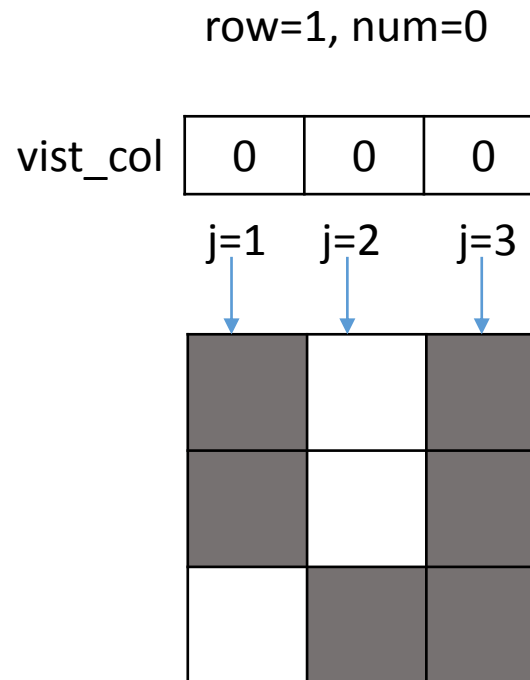
2



# 棋盘问题

## 实现

```
bool vist_col[9]; //列标记
void DFS(int row,int num) //逐行搜索，row为当前搜索行，num为已填充的棋子数
{
    if(num==k){
        status++;
        return;
    }
    if(row>n) return; //n为棋盘的大小
    for(int j=1;j<=n;j++){
        if(chess[row][j]=='#' && !vist_col[j])
        {
            vist_col[j]=true; //放置棋子的列标记
            DFS(row+1,num+1);
            vist_col[j]=false; //回溯后，说明摆好棋子的状态status已记录，将当前的列标记还原
        }
        DFS(row+1,num); //当k<n时，row在等于n之前就可能已经把全部棋子放好
                        //为了处理多余行，保持已放的棋子数不变,搜索在下一行放棋子的情况
    }
    return;
}
```

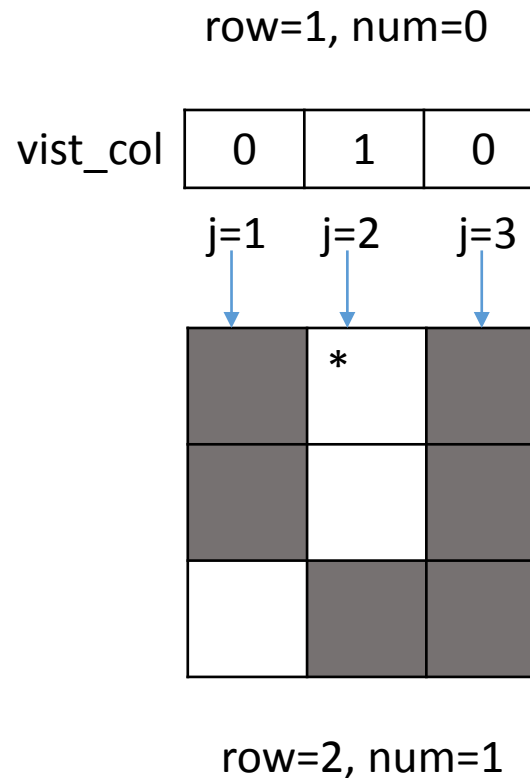




# 棋盘问题

## 实现

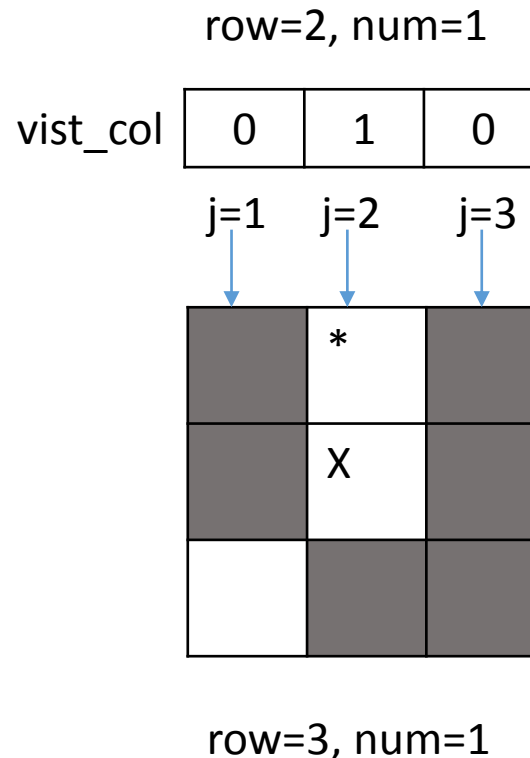
```
bool vist_col[9]; //列标记
void DFS(int row,int num) //逐行搜索，row为当前搜索行，num为已填充的棋子数
{
    if(num==k){
        status++;
        return;
    }
    if(row>n) return; //n为棋盘的大小
    for(int j=1;j<=n;j++){
        if(chess[row][j]!='#' && !vist_col[j])
        {
            vist_col[j]=true; //放置棋子的列标记
            DFS(row+1,num+1);
            vist_col[j]=false; //回溯后，说明摆好棋子的状态status已记录，将当前的列标记还原
        }
    }
    DFS(row+1,num); //当k<n时，row在等于n之前就可能已经把全部棋子放好
                    //为了处理多余行，保持已放的棋子数不变,搜索在下一行放棋子的情况
    return;
}
```



# 棋盘问题

## 实现

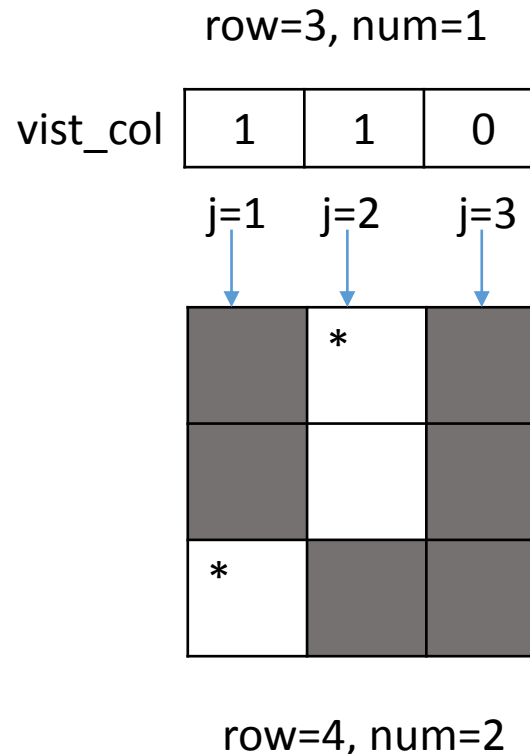
```
bool vist_col[9]; //列标记
void DFS(int row,int num) //逐行搜索，row为当前搜索行，num为已填充的棋子数
{
    if(num==k){
        status++;
        return;
    }
    if(row>n) return; //n为棋盘的大小
    for(int j=1;j<=n;j++){
        if(chess[row][j]!='#' && !vist_col[j])
        {
            vist_col[j]=true; //放置棋子的列标记
            DFS(row+1,num+1);
            vist_col[j]=false; //回溯后，说明摆好棋子的状态status已记录，将当前的列标记还原
        }
        DFS(row+1,num); //当k<n时，row在等于n之前就可能已经把全部棋子放好
        //为了处理多余行，保持已放的棋子数不变,搜索在下一行放棋子的情况
    }
    return;
}
```



# 棋盘问题

## 实现

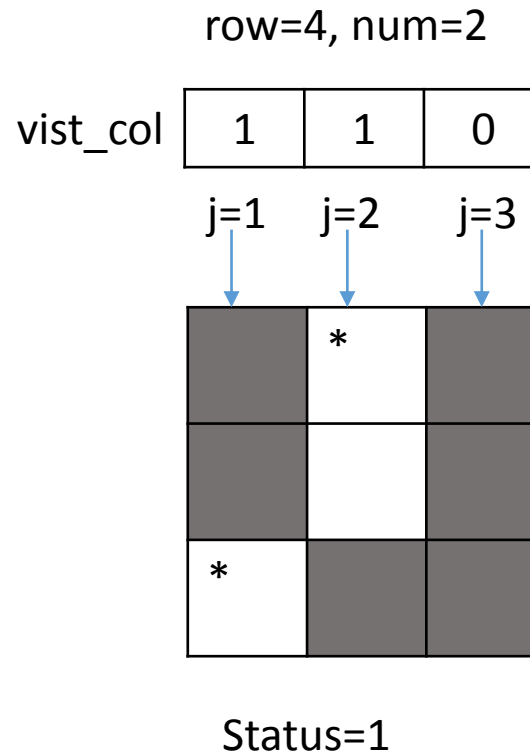
```
bool vist_col[9]; //列标记
void DFS(int row,int num) //逐行搜索，row为当前搜索行，num为已填充的棋子数
{
    if(num==k){
        status++;
        return;
    }
    if(row>n) return; //n为棋盘的大小
    for(int j=1;j<=n;j++){
        if(chess[row][j]!='#' && !vist_col[j])
        {
            vist_col[j]=true; //放置棋子的列标记
            DFS(row+1,num+1);
            vist_col[j]=false; //回溯后，说明摆好棋子的状态status已记录，将当前的列标记还原
        }
    }
    DFS(row+1,num); //当k<n时，row在等于n之前就可能已经把全部棋子放好
                    //为了处理多余行，保持已放的棋子数不变,搜索在下一行放棋子的情况
    return;
}
```



# 棋盘问题

## 实现

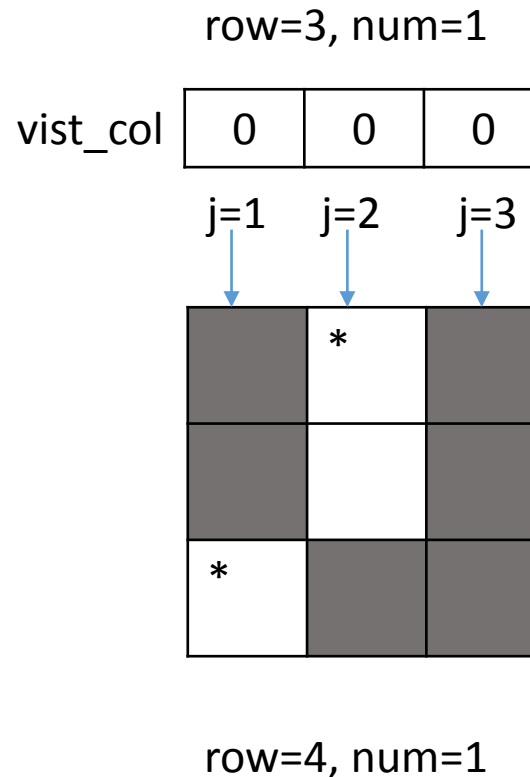
```
bool vist_col[9]; //列标记
void DFS(int row,int num) //逐行搜索，row为当前搜索行，num为已填充的棋子数
{
    if(num==k){
        status++;
        return;
    }
    if(row>n) return; //n为棋盘的大小
    for(int j=1;j<=n;j++){
        if(chess[row][j]!='#' && !vist_col[j])
        {
            vist_col[j]=true; //放置棋子的列标记
            DFS(row+1,num+1);
            vist_col[j]=false; //回溯后，说明摆好棋子的状态status已记录，将当前的列标记还原
        }
    }
    DFS(row+1,num); //当k<n时，row在等于n之前就可能已经把全部棋子放好
                    //为了处理多余行，保持已放的棋子数不变,搜索在下一行放棋子的情况
    return;
}
```



# 棋盘问题

## 实现

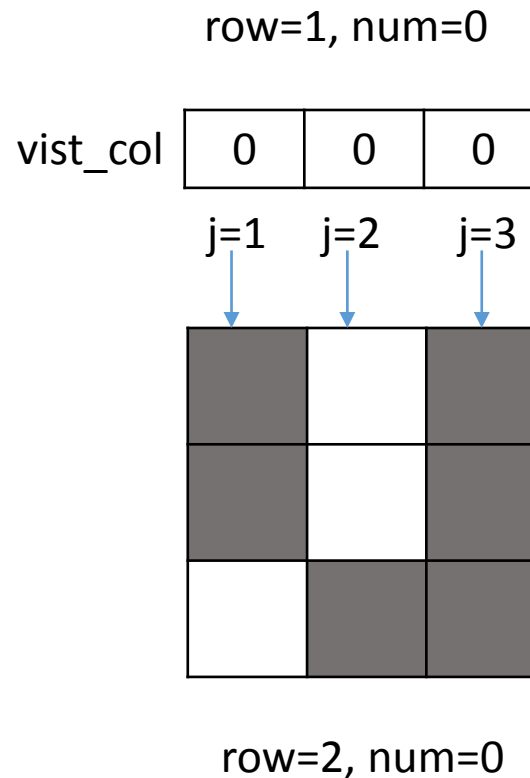
```
bool vist_col[9]; //列标记
void DFS(int row,int num) //逐行搜索，row为当前搜索行，num为已填充的棋子数
{
    if(num==k){
        status++;
        return;
    }
    if(row>n) return; //n为棋盘的大小
    for(int j=1;j<=n;j++){
        if(chess[row][j]!='#' && !vist_col[j])
        {
            vist_col[j]=true; //放置棋子的列标记
            DFS(row+1,num+1);
            vist_col[j]=false; //回溯后，说明摆好棋子的状态status已记录，将当前的列标记还原
        }
        DFS(row+1,num); //当k<n时，row在等于n之前就可能已经把全部棋子放好
        //为了处理多余行，保持已放的棋子数不变,搜索在下一行放棋子的情况
    }
    return;
}
```



# 棋盘问题

## 实现

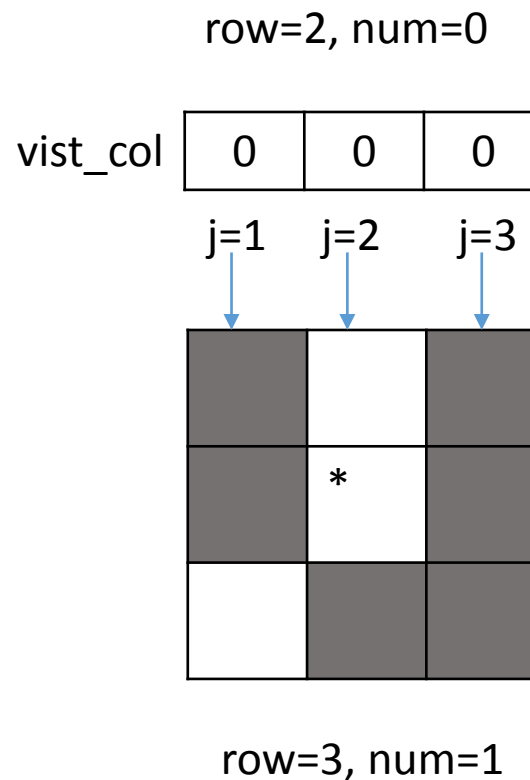
```
bool vist_col[9]; //列标记
void DFS(int row,int num) //逐行搜索，row为当前搜索行，num为已填充的棋子数
{
    if(num==k){
        status++;
        return;
    }
    if(row>n) return; //n为棋盘的大小
    for(int j=1;j<=n;j++){
        if(chess[row][j]=='#' && !vist_col[j])
        {
            vist_col[j]=true; //放置棋子的列标记
            DFS(row+1,num+1);
            vist_col[j]=false; //回溯后，说明摆好棋子的状态status已记录，将当前的列标记还原
        }
        DFS(row+1,num); //当k<n时，row在等于n之前就可能已经把全部棋子放好
        //为了处理多余行，保持已放的棋子数不变,搜索在下一行放棋子的情况
    }
    return;
}
```



# 棋盘问题

## 实现

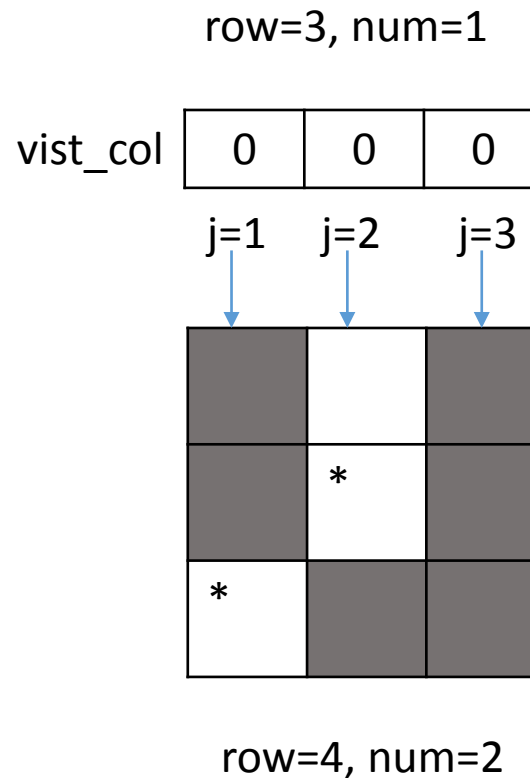
```
bool vist_col[9]; //列标记
void DFS(int row,int num) //逐行搜索，row为当前搜索行，num为已填充的棋子数
{
    if(num==k){
        status++;
        return;
    }
    if(row>n) return; //n为棋盘的大小
    for(int j=1;j<=n;j++){
        if(chess[row][j]=='#' && !vist_col[j])
        {
            vist_col[j]=true; //放置棋子的列标记
            DFS(row+1,num+1);
            vist_col[j]=false; //回溯后，说明摆好棋子的状态status已记录，将当前的列标记还原
        }
        DFS(row+1,num); //当k<n时，row在等于n之前就可能已经把全部棋子放好
        //为了处理多余行，保持已放的棋子数不变,搜索在下一行放棋子的情况
    }
    return;
}
```



# 棋盘问题

## 实现

```
bool vist_col[9]; //列标记
void DFS(int row,int num) //逐行搜索，row为当前搜索行，num为已填充的棋子数
{
    if(num==k){
        status++;
        return;
    }
    if(row>n) return; //n为棋盘的大小
    for(int j=1;j<=n;j++){
        if(chess[row][j]=='#' && !vist_col[j])
        {
            vist_col[j]=true; //放置棋子的列标记
            DFS(row+1,num+1);
            vist_col[j]=false; //回溯后，说明摆好棋子的状态status已记录，将当前的列标记还原
        }
    }
    DFS(row+1,num); //当k<n时，row在等于n之前就可能已经把全部棋子放好
                    //为了处理多余行，保持已放的棋子数不变,搜索在下一行放棋子的情况
    return;
}
```

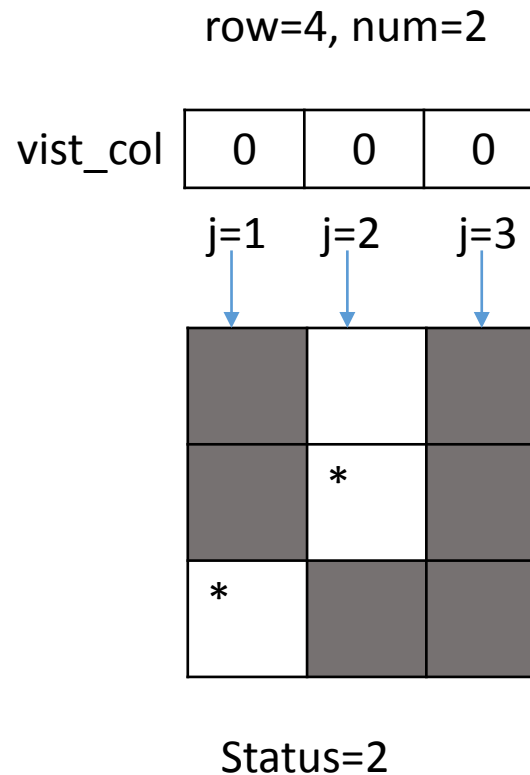




# 棋盘问题

## 实现

```
bool vist_col[9]; //列标记
void DFS(int row,int num) //逐行搜索，row为当前搜索行，num为已填充的棋子数
{
    if(num==k){
        status++;
        return;
    }
    if(row>n) return; //n为棋盘的大小
    for(int j=1;j<=n;j++){
        if(chess[row][j]=='#' && !vist_col[j])
        {
            vist_col[j]=true; //放置棋子的列标记
            DFS(row+1,num+1);
            vist_col[j]=false; //回溯后，说明摆好棋子的状态status已记录，将当前的列标记还原
        }
    }
    DFS(row+1,num); //当k<n时，row在等于n之前就可能已经把全部棋子放好
                    //为了处理多余行，保持已放的棋子数不变,搜索在下一行放棋子的情况
    return;
}
```



# Oil Deposits

---

## 题目描述

- The GeoSurvComp geologic survey company is responsible for detecting underground oil deposits. GeoSurvComp works with one large rectangular region of land at a time, and creates a grid that divides the land into numerous square plots. It then analyzes each plot separately, using sensing equipment to determine whether or not the plot contains oil. A plot containing oil is called a pocket. If two pockets are adjacent, then they are part of the same oil deposit. Oil deposits can be quite large and may contain numerous pockets. Your job is to determine how many different oil deposits are contained in a grid.

# Oil Deposits

---

## 题目描述

- GeoSurvComp地质调查公司负责探测地下石油储量。GeoSurvComp公司在一段时间内在一大块矩形区域内工作，创造出一个网格把土地分成很多方块。对每一方块单独分析，使用感应设备去测定方块中是否含有石油。含有油的方块称为口袋，如果两个口袋是相邻的，那它们属于同一个油床。油床可以相当大，可以包含众多口袋。你的任务就是测定出有多少不同的油床。

# Oil Deposits

## Sample Input

1 1      → 地图为一行一列  
\*      → \*表示不含石油的区域  
1 8      → 地图为一行八列  
@@\*\*\*\*\*@\*      → @表示含石油的区域  
0 0

## Sample Output

0      → 地图中无油床  
1

## 解题思路

- 在地图中找出所有的连通分量
- 先对整个地图进行遍历，找到一个入口，然后用DFS
- 将已找到的@标记为\*，避免重新用visit对遍历过的地图进行标记

# Oil Deposits

5 5

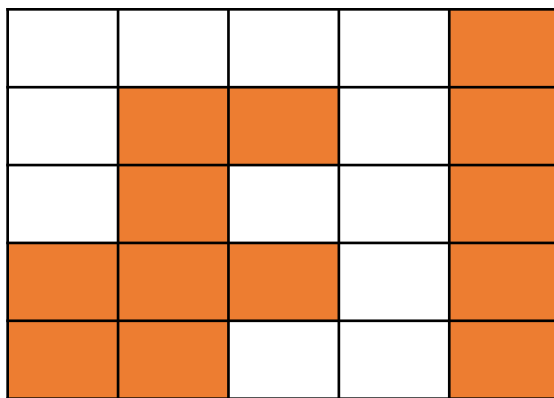
\*\*\*\*\*@

\*@@\*@

\*@\*\*@

@@@\*@

@@\*\*@



```
for (i = 0; i < m; i++) // 分别以每个点为起点扫描一次
{
    for (j = 0; j < n; j++)
    {
        if (s[i][j] == '@') // 直接忽略没有油田的点
        {
            s[i][j] = '*'; // 标记该点已经走过
            ans++;
            dfs(i, j);
        }
    }
}
```

# Oil Deposits

5 5

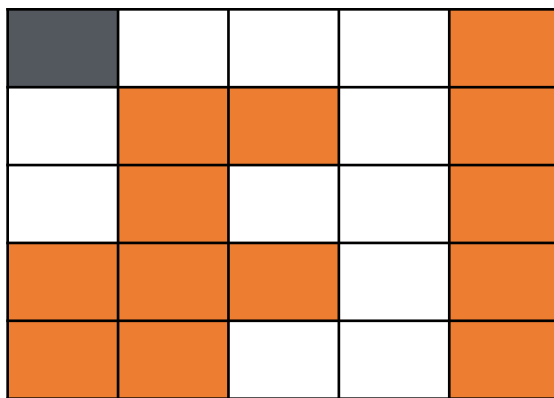
\*\*\*\*\*@

\*@@\*@

\*@\*\*@

@@@\*@

@@\*\*@



```
for (i = 0; i < m; i++) // 分别以每个点为起点扫描一次
{
    for (j = 0; j < n; j++)
    {
        if (s[i][j] == '@') // 直接忽略没有油田的点
        {
            剪枝 s[i][j] = '*'; // 标记该点已经走过
            ans++;
            dfs(i, j);
        }
    }
}
```

# Oil Deposits

5 5

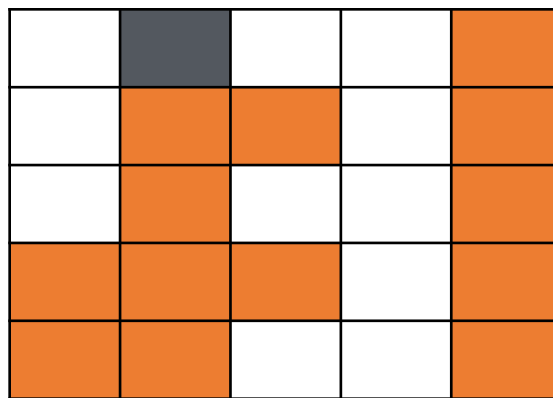
\*\*\*\*\*@

\*@@\*@

\*@\*\*@

@@@\*@

@@\*\*@



```
for (i = 0; i < m; i++) // 分别以每个点为起点扫描一次
{
    for (j = 0; j < n; j++)
    {
        if (s[i][j] == '@') // 直接忽略没有油田的点
        {
            s[i][j] = '*'; // 标记该点已经走过
            ans++;
            dfs(i, j);
        }
    }
}
```

# Oil Deposits

5 5

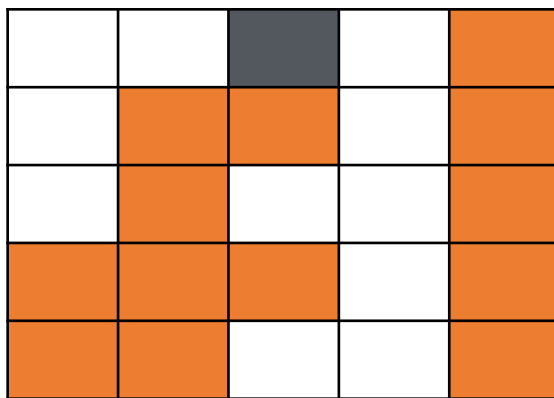
\*\*\*\*\*@

\*@@\*@

\*@\*\*@

@@@\*@

@@\*\*@



```
for (i = 0; i < m; i++) // 分别以每个点为起点扫描一次
{
    for (j = 0; j < n; j++)
    {
        if (s[i][j] == '@') // 直接忽略没有油田的点
        {
            s[i][j] = '*'; // 标记该点已经走过
            ans++;
            dfs(i, j);
        }
    }
}
```



# Oil Deposits

5 5

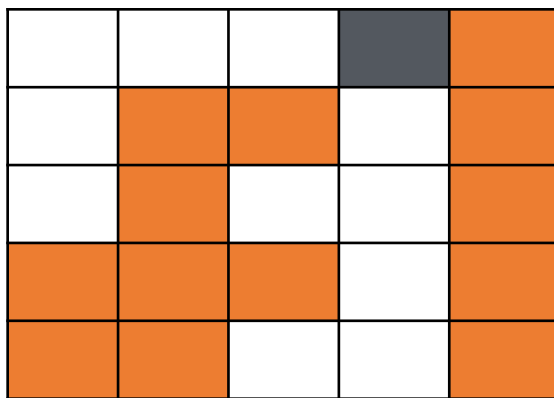
\*\*\*\*\*@

\*@@\*@

\*@\*\*@

@@@\*@

@@\*\*@



```
for (i = 0; i < m; i++) // 分别以每个点为起点扫描一次
{
    for (j = 0; j < n; j++)
    {
        if (s[i][j] == '@') // 直接忽略没有油田的点
        {
            s[i][j] = '*'; // 标记该点已经走过
            ans++;
            dfs(i, j);
        }
    }
}
```

# Oil Deposits

5 5

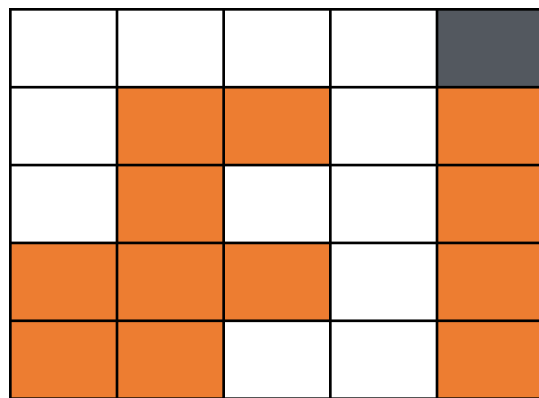
\*\*\*\*@

\*@@\*@

\*@\*\*@

@@@\*@

@@\*\*@



```
void dfs(int x, int y)
```

```
{
```

```
    int i;
```

```
    for (i = 0; i < 8; i++) // 依次扫描八个方向, 寻找可行点
```

```
    {
```

```
        int tx = x + move[i][0];
```

```
        int ty = y + move[i][1];
```

```
        if (tx >= 0 && tx < m && ty >= 0 && ty < n && s[tx][ty] == '@')
```

```
            // 判断是否超出地图边界和是否可行
```

```
        {
```

```
            s[tx][ty] = '*'; // 标记已经走过的点
```

```
            dfs(tx, ty); // 以此点为起点继续搜索
```

```
        }
```

```
    }
```

```
}
```

# Oil Deposits

5 5

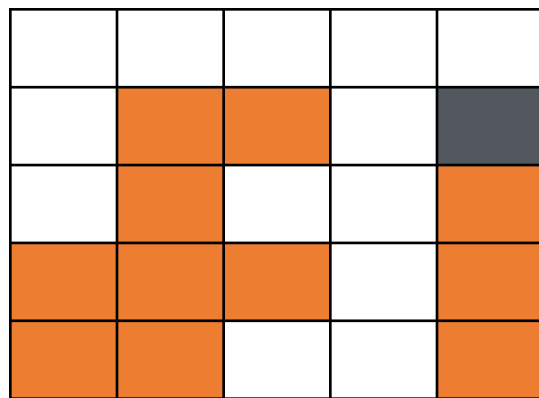
\*\*\*\*@

\*@@\*@

\*@\*\*@

@@@\*@

@@\*\*@



```
void dfs(int x, int y)
```

```
{
```

```
    int i;
```

```
    for (i = 0; i < 8; i++) // 依次扫描八个方向, 寻找可行点
```

```
    {
```

```
        int tx = x + move[i][0];
```

```
        int ty = y + move[i][1];
```

```
        if (tx >= 0 && tx < m && ty >= 0 && ty < n && s[tx][ty] == '@')
```

```
            // 判断是否超出地图边界和是否可行
```

```
        {
```

```
            s[tx][ty] = '*'; // 标记已经走过的点
```

```
            dfs(tx, ty); // 以此点为起点继续搜索
```

```
        }
```

```
    }
```

```
}
```

# Oil Deposits

5 5

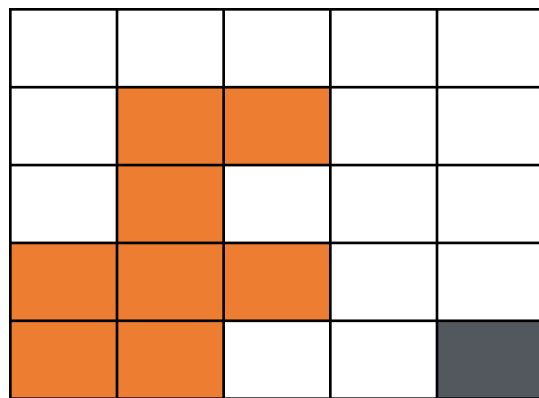
\*\*\*\*@

\*@@\*@

\*@\*\*@

@@@\*@

@@\*\*@



```
void dfs(int x, int y)
```

```
{
```

```
    int i;
```

```
    for (i = 0; i < 8; i++) // 依次扫描八个方向, 寻找可行点
```

```
    {
```

```
        int tx = x + move[i][0];
```

```
        int ty = y + move[i][1];
```

```
        if (tx >= 0 && tx < m && ty >= 0 && ty < n && s[tx][ty] == '@')
```

```
            // 判断是否超出地图边界和是否可行
```

```
        {
```

```
            s[tx][ty] = '*'; // 标记已经走过的点
```

```
            dfs(tx, ty); // 以此点为起点继续搜索
```

```
        }
```

```
    }
```

# Oil Deposits

5 5

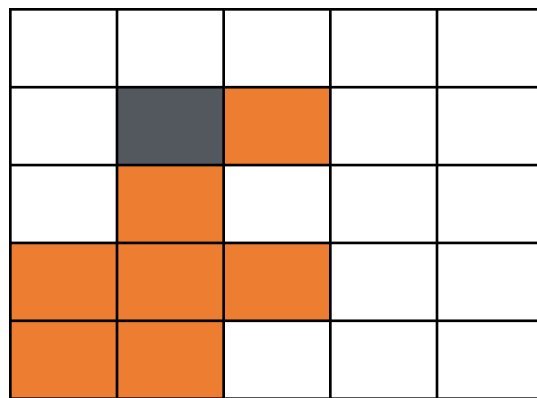
\*\*\*\*@

\*@@\*@

\*@\*\*@

@@@\*@

@@\*\*@



```
for (i = 0; i < m; i++) // 分别以每个点为起点扫描一次
{
    for (j = 0; j < n; j++)
    {
        if (s[i][j] == '@') // 直接忽略没有油田的点
        {
            s[i][j] = '*'; // 标记该点已经走过
            ans++;
            dfs(i, j);
        }
    }
}
```

# Catch That Cow

---

## 题目描述

- Farmer John has been informed of the location of a fugitive cow and wants to catch her immediately. He starts at a point  $N$  ( $0 \leq N \leq 100,000$ ) on a number line and the cow is at a point  $K$  ( $0 \leq K \leq 100,000$ ) on the same number line. Farmer John has two modes of transportation:
  - \*Walking: move from any point  $X$  to the points  $X - 1$  or  $X + 1$  in a single minute
  - \*Teleporting: move from any point  $X$  to the point  $2 \times X$  in a single minuteIf the cow, does not move at all, how long does it take for Farmer John to retrieve it?
- 已知 $n$ 和 $k$ 点的横坐标，求 $n$ 到 $k$ 的最小移动次数

# Catch That Cow

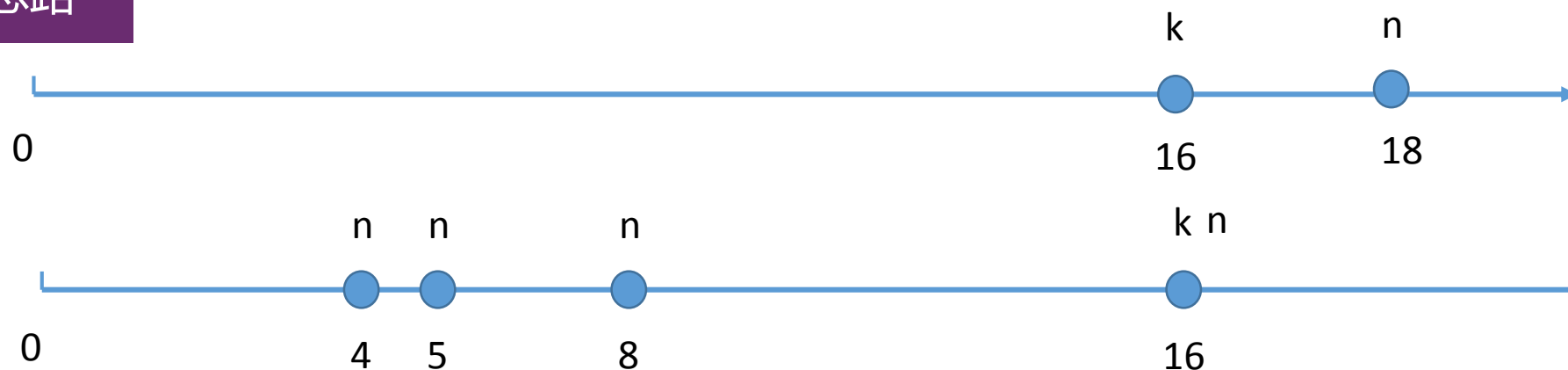
## Sample Input

5 16       $\longrightarrow$  农夫与牛的横坐标分别为5和16

## Sample Output

3       $\longrightarrow$  三分钟后农夫可以抓到牛

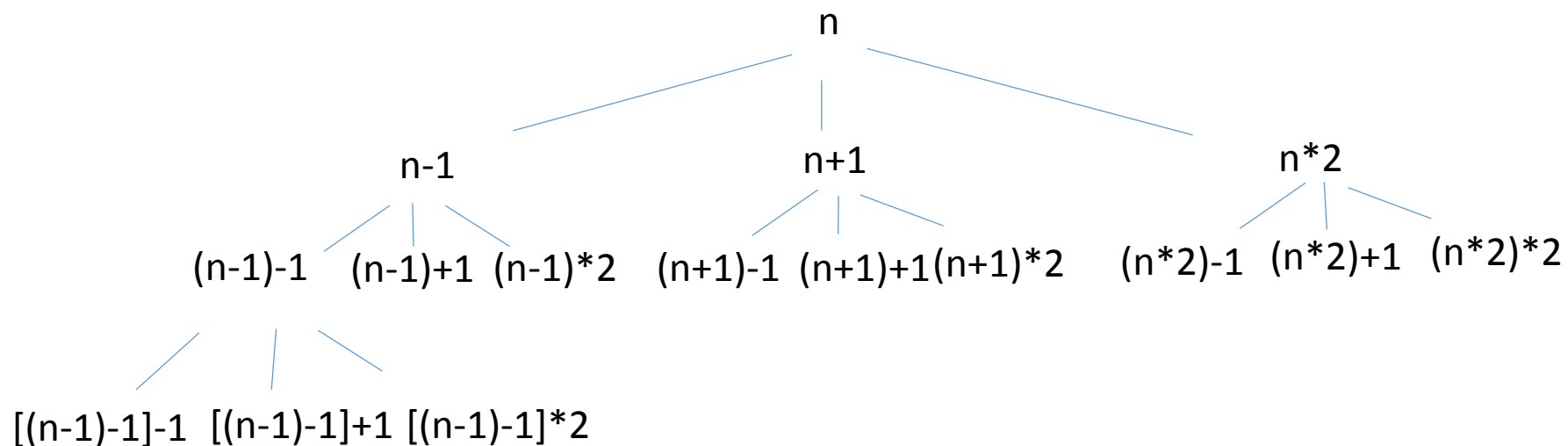
### 解题思路



- 1、如果 $n$ 在 $k$ 后面，只有一步步往前移动到 $k$ 位置，即 $n \geq k$ 时，输出 $n - k$ 即可。
- 2、如果 $n$ 在 $k$ 前面，则**广度优先搜索**进行查找。

# Catch That Cow

---



1. 设置一个队列Q，从顶点n出发，让其进队；
2. 将顶点元素出队，遍历该顶点的所有邻接点（对应于此题即FJ的三种走法n-1、n+1、n\*2），让没有遍历过的邻接点进队；
3. 若邻接点的值等于k，则结束；
4. 若队空停止，队不空时继续第2步。



# Catch That Cow



5

4 6 10

← 1步

6 10 3 8

← 2步

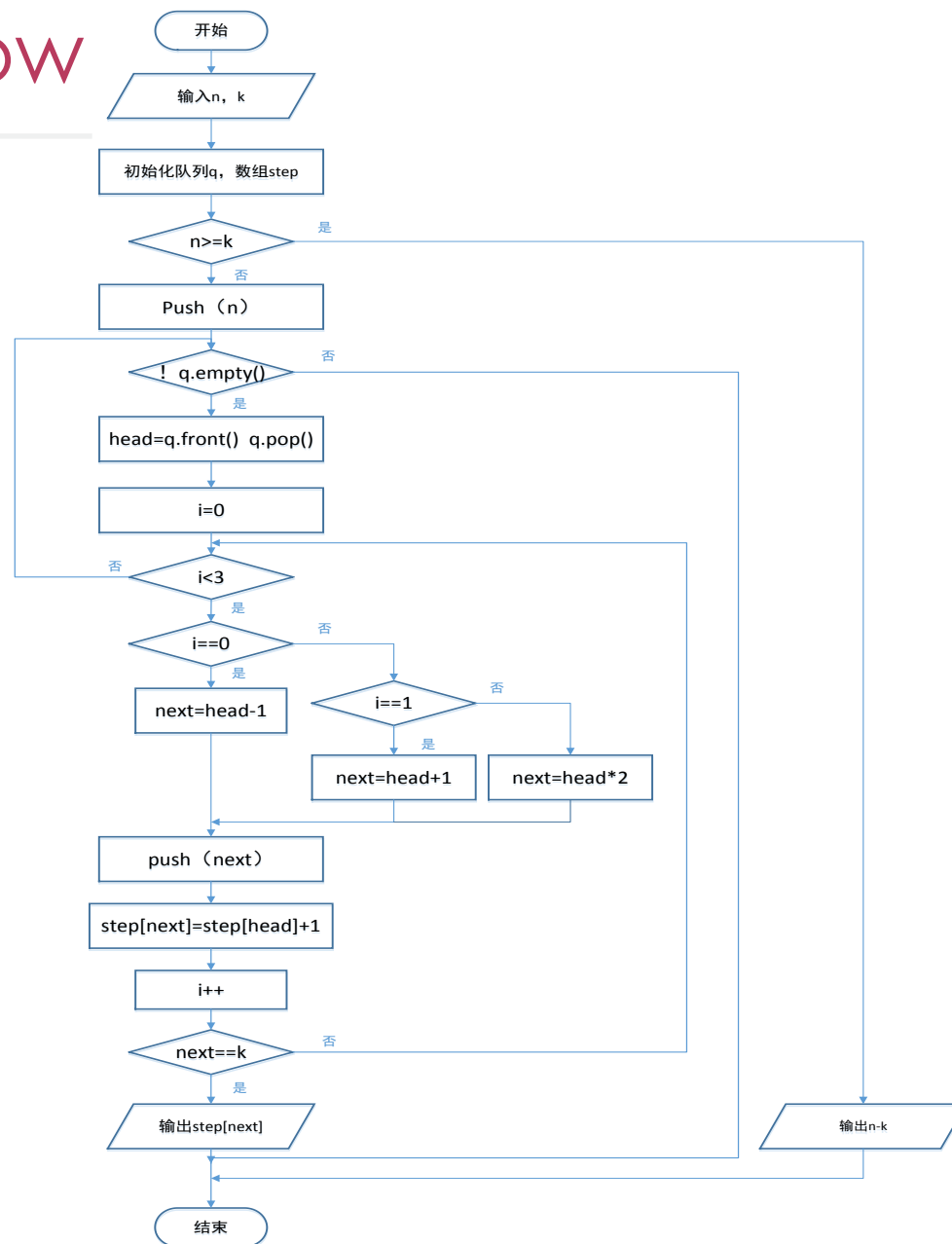
10 3 8 7 12

3 8 7 12 9 11 20

8 7 12 9 11 20 2

← 3步

7 12 9 11 20 2 16



# 迷宫问题

---

## 题目描述

- 定义一个二维数组：

```
int maze[5][5] = {  
    0, 1, 0, 0, 0,  
    0, 1, 0, 1, 0,  
    0, 0, 0, 0, 0,  
    0, 1, 1, 1, 0,  
    0, 0, 0, 1, 0,  
};
```

它表示一个迷宫，其中的1表示墙壁，0表示可以走的路，只能横着走或竖着走，不能斜着走，要求编程找出从左上角到右下角的最短路线。

## 解题思路

## 识别节点和边

节点就是是迷宫路上的每一个非墙格子，根据题意，只能横竖走，因此格子与它横竖方向上的格子是有连通关系的。只要这个格子跟另一个格子是连通的，那么两个格子节点间就有一条边。

0, 1, 0, 0, 0,  
0, 1, 0, 1, 0,  
0, 0, 0, 0, 0,  
0, 1, 1, 1, 0,  
0, 0, 0, 1, 0



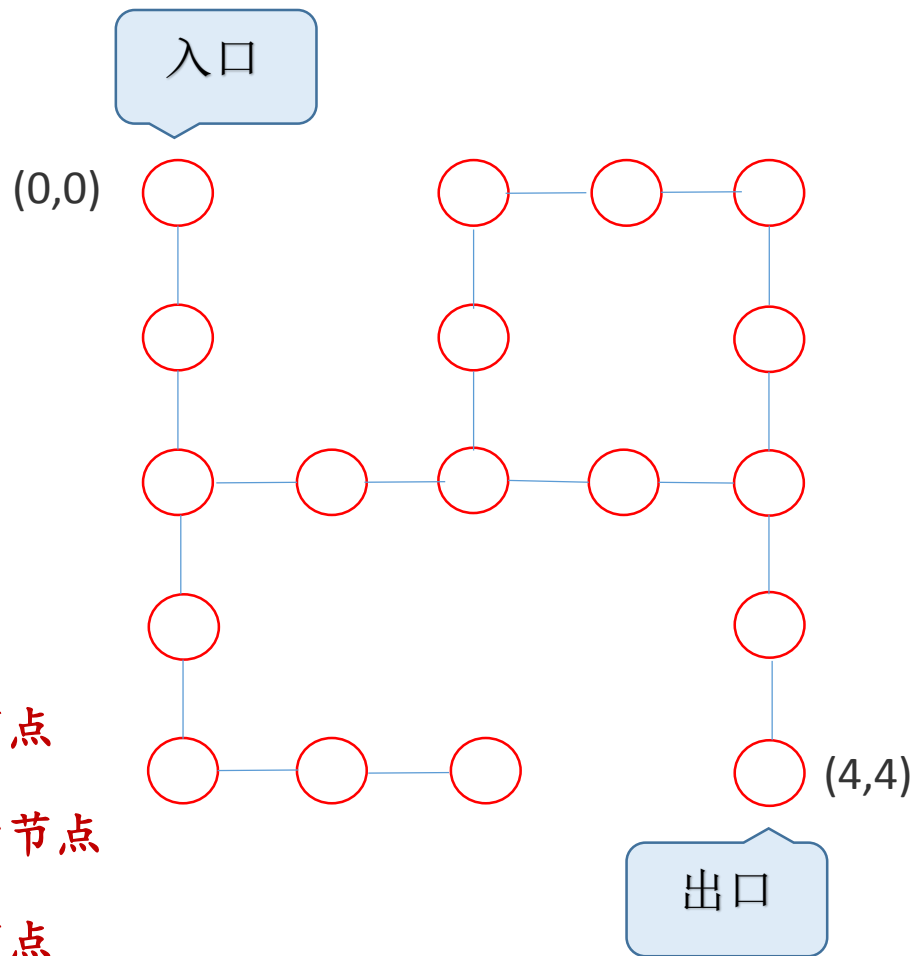
## 未探索节点



## 即将探索节点



## 已探索节点



# 迷宫问题

## 模拟算法流程

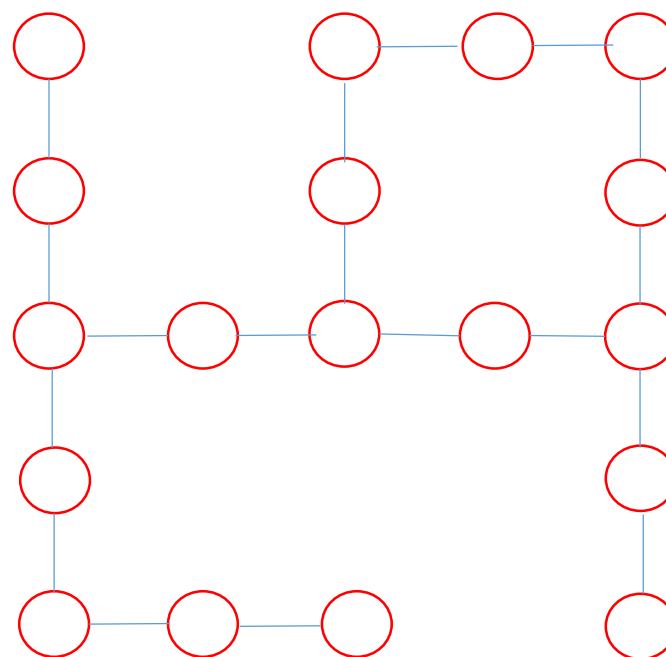
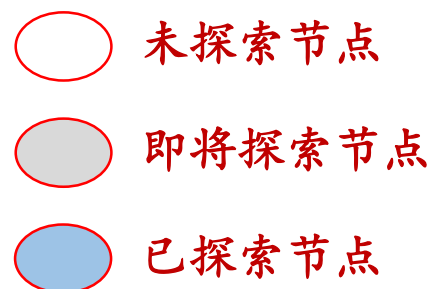
step 1:

起点 $V_s$ 为(0,0)

终点 $V_d$ 为(4,4)

灰色节点集合（队列） $Q=\{\}$

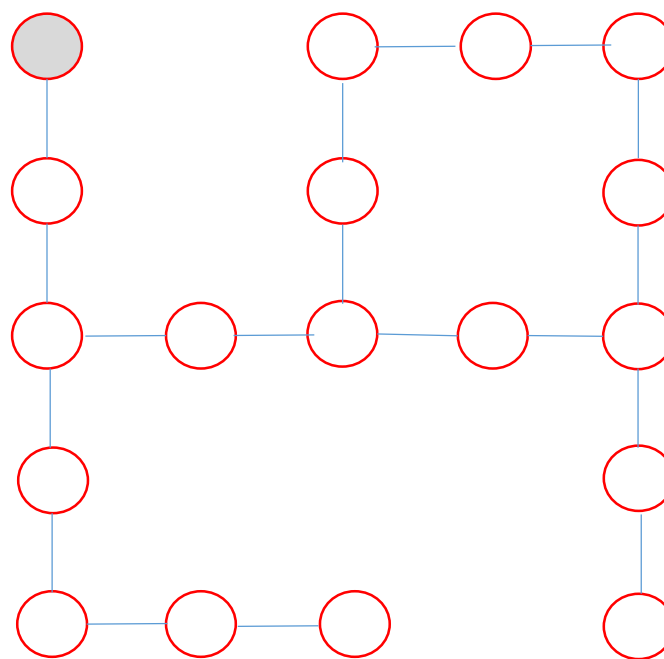
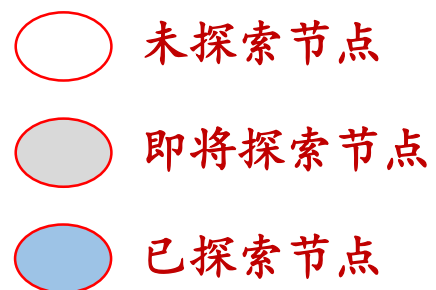
初始化所有节点为白色节点



# 迷宫问题

## 模拟算法流程

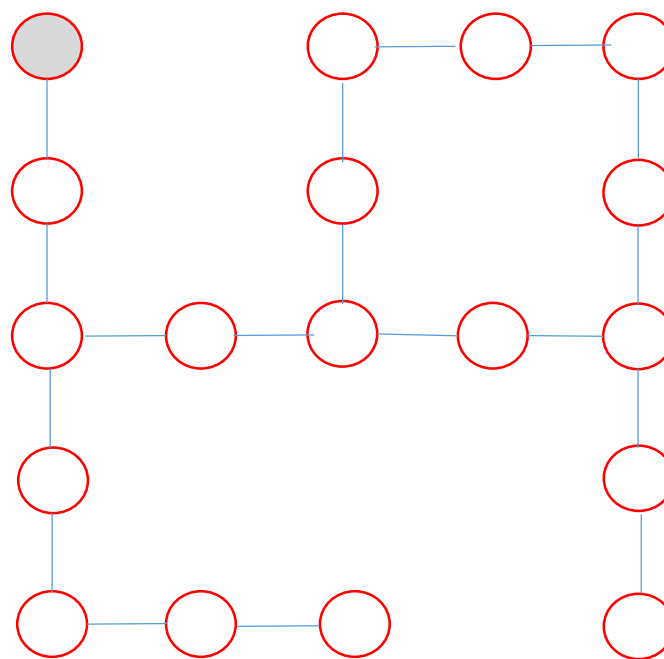
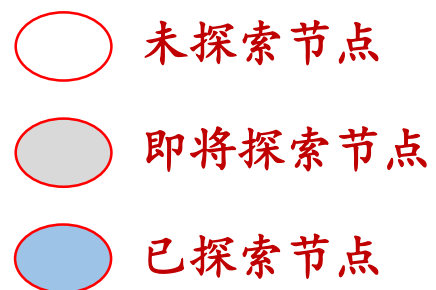
step 2:  
起始节点Vs变成灰色，加入  
队列Q， $Q=\{(0,0)\}$



# 迷宫问题

## 模拟算法流程




step 3:  
Q不包含终点(4,4)，取出队  
列Q的头一个节点 $V_n$ ，  
 $V_n=(0,0)$ ， $Q=\{\}$

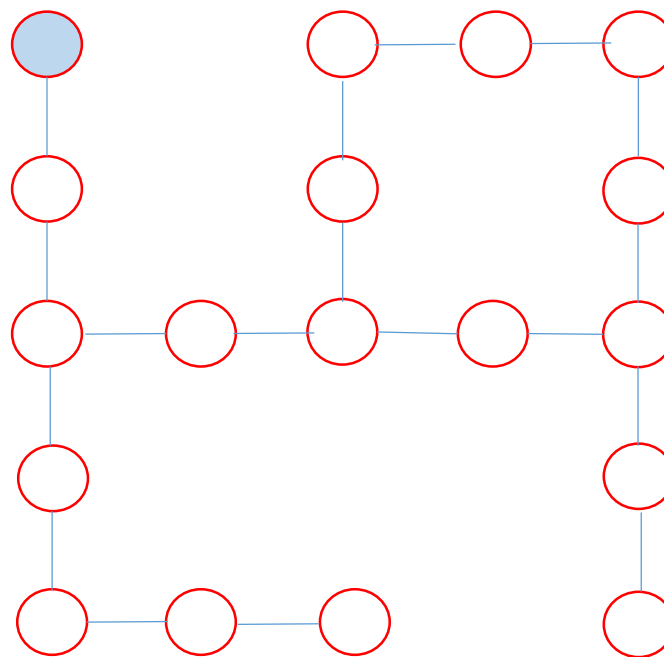


# 迷宫问题

## 模拟算法流程

step 4:  
把 $V_n=(0,0)$ 染成蓝色，取出 $V_n$ 所有相邻的白色节点 $(0,1)$ 。

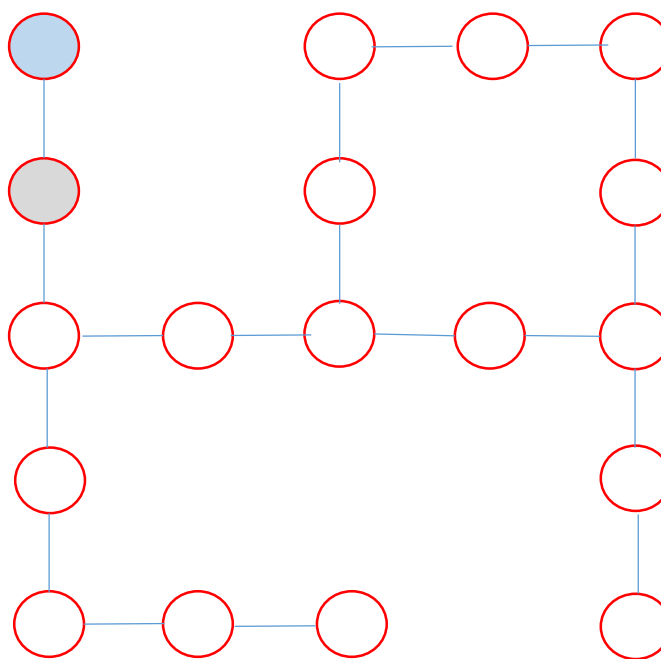
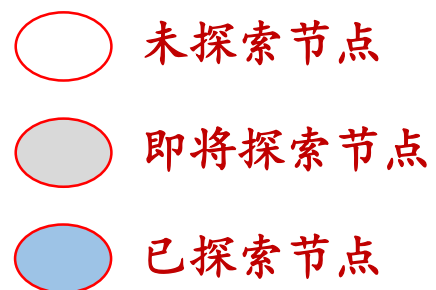
-  未探索节点
-  即将探索节点
-  已探索节点



# 迷宫问题

## 模拟算法流程

step 5:  
将(0,1)加入队列Q, 并染成  
灰色,  $Q=\{(0,1)\}$

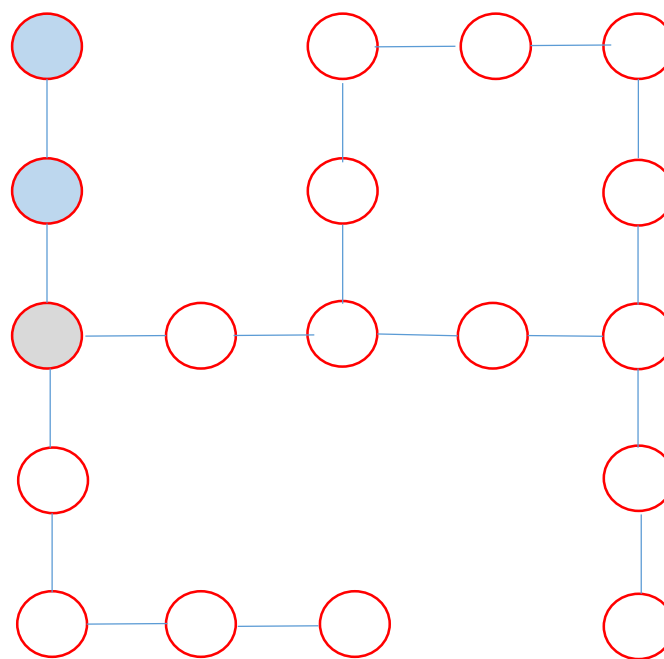
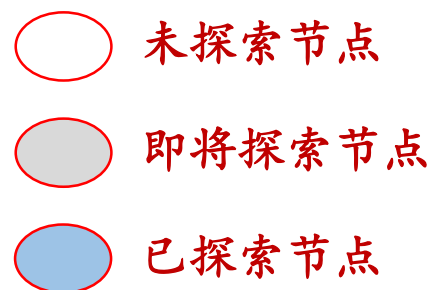




# 迷宫问题

## 模拟算法流程

Q不包含终点(4,4)，将队首元素  
 $V_n = (0,1)$  取出，它的相邻节  
点入队  
 $Q = \{ (0,1) \}$

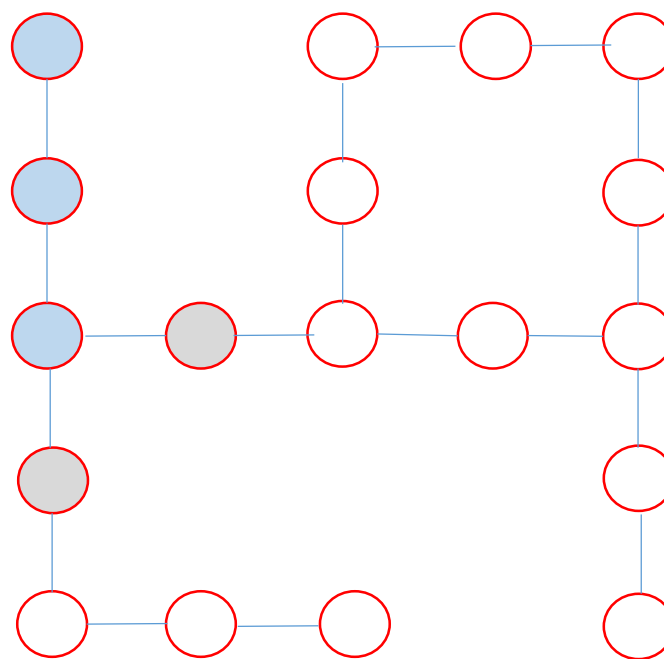
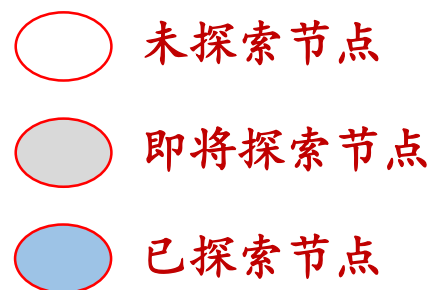


# 迷宫问题

## 模拟算法流程

$V_n = (0, 2)$

$Q = \{ (1, 2), (0, 3) \}$

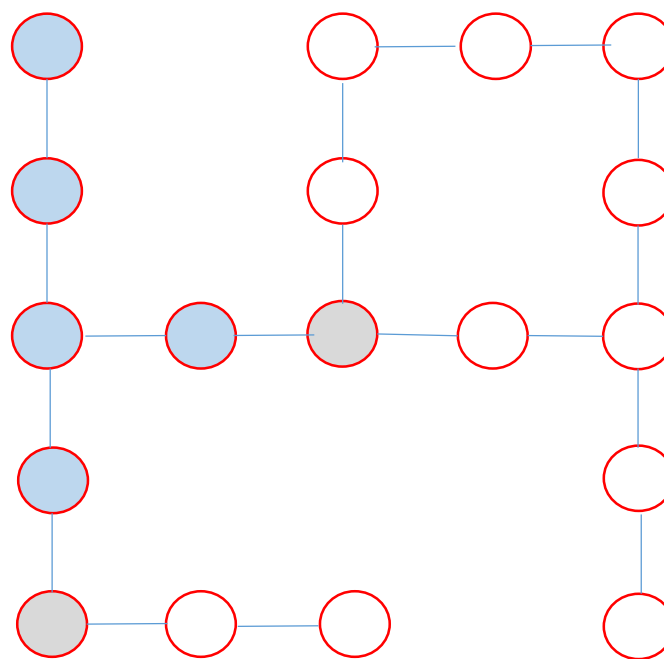
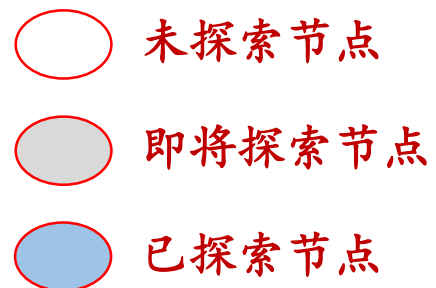


# 迷宫问题

## 模拟算法流程

$V_n = \{ (1,2), (0,3) \}$

$Q = \{ (2,2), (0,4) \}$

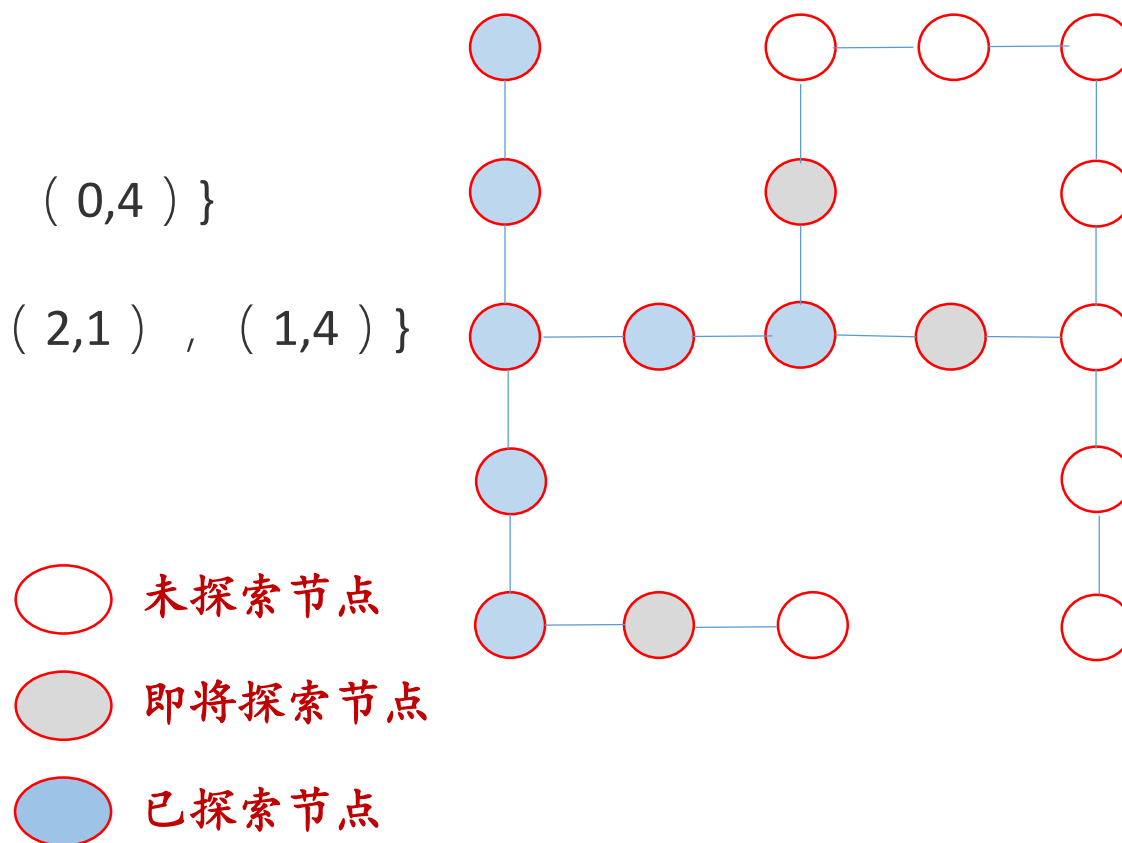


# 迷宫问题

## 模拟算法流程

$V_n = \{ (2,2), (0,4) \}$

$Q = \{ (3,2), (2,1), (1,4) \}$

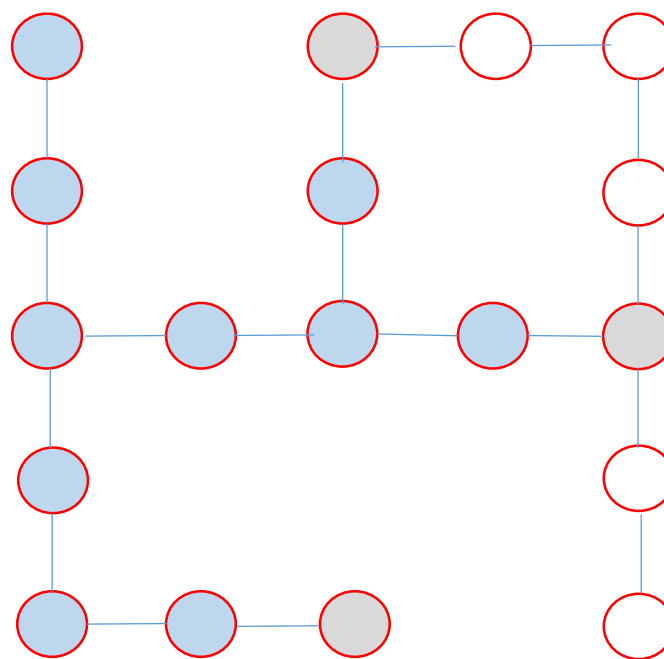
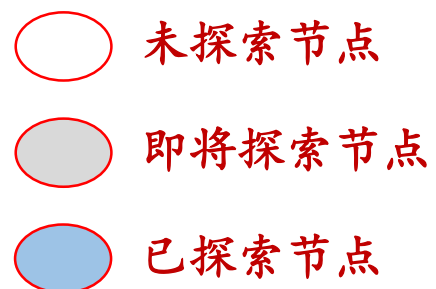


# 迷宫问题

## 模拟算法流程

$V_n = \{ (3,2), (2,1), (1,4) \}$

$Q = \{ (2,0), (4,2), (2,4) \}$

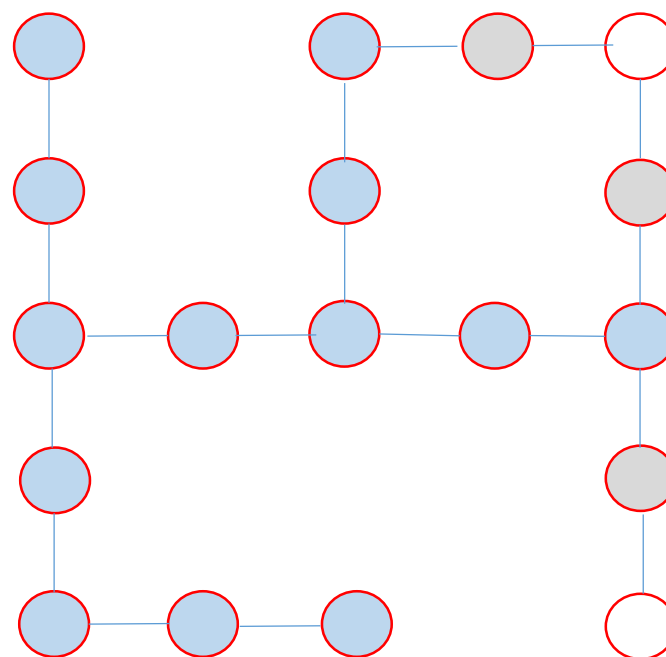
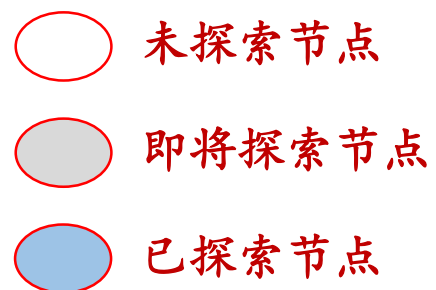


# 迷宫问题

## 模拟算法流程

$V_n = \{ (2,0), (4,2), (2,4) \}$

$Q = \{ (3,0), (4,1), (4,3) \}$

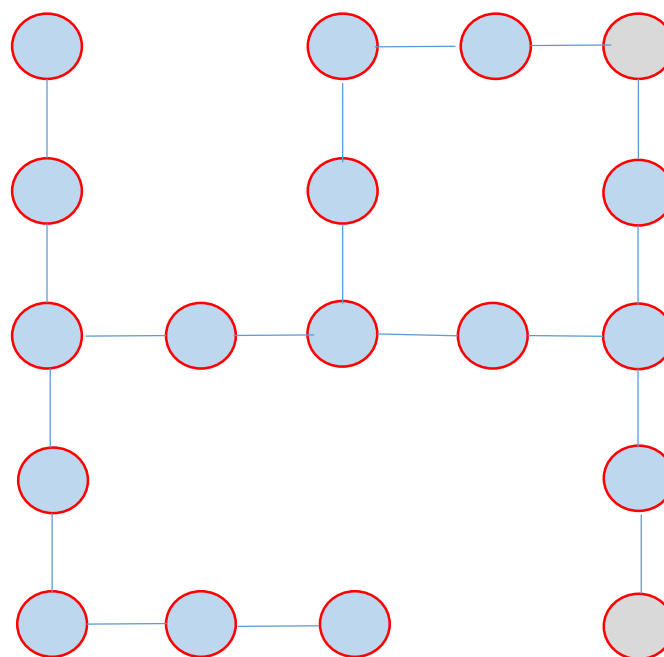
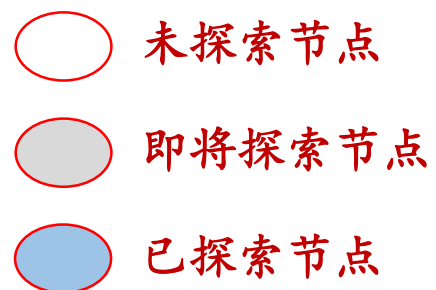


# 迷宫问题

## 模拟算法流程

$V_n = \{ (3,0), (4,1), (4,3) \}$

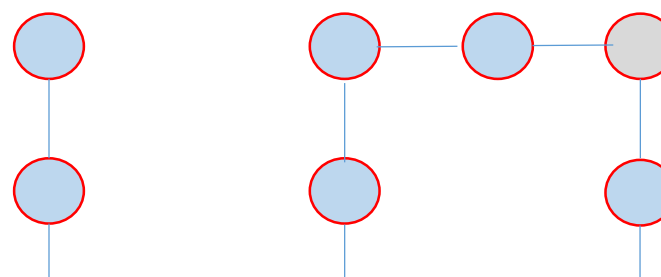
$Q = \{ (4,0), (4,4) \}$



# 迷宫问题

## 模拟算法流程

$V_n = \{ (3,0), (4,1), (4,3) \}$



结论：所以根据广度优先搜索的话，搜索到终点时，该路径一定是最短的。

BFS搜索顺序是第一层->第二层->第三层->第N层。

假设终点在第N层，则搜索到的路径长度肯定是N，且这个N一定是最短的。

我们用简单的反证法来证明：假设终点在第N层上边出现过，例如第M层， $M < N$ ，那么我们在搜索的过程中，肯定是先搜索到第M层的，此时搜索到第M层的时候发现终点出现过了，那么最短路径应该是M，而不是N了。



# 迷宫问题

## 实现

```
void print_ans(int x, int y)
{
    if (a[x][y].px != -1 && a[x][y].py != -1)
        print_ans(a[x][y].px, a[x][y].py);
    printf("(%d, %d)\n", x, y);
}

void bfs()
{
    memset(vis, 0, sizeof(vis));
    vis[0][0] = true;
    a[0][0].px = -1;
    a[0][0].py = -1;

    q.push(site(0, 0));
    while (!q.empty())
    {
        site temp = q.front();

        q.pop();
        int x = temp.x, y = temp.y;
        for (int i = 0; i < 4; i++)
        {
            int nex = x + dx[i];
            int ney = y + dy[i];
            if (nex >= 0 && nex < 5 && ney >= 0 && ney < 5 && vis[nex][ney] == 0)
            {
                a[nex][ney].px = x;
                a[nex][ney].py = y;
                if (nex == 4 && ney == 4) return;

                q.push(site(nex, ney));
                vis[nex][ney] = true;
            }
        }
    }
}
```

```
void bfs()
{
    int i, head, tail;
    int x, y, xx, yy;
    memset(vis, 0, sizeof(vis));
    head = 0;
    tail = 1;
    list[0].x = 0;
    list[0].y = 0;
    pre[0] = -1;
    while (head < tail)
    {
        x = list[head].x;
        y = list[head].y;
        if (x == 4 && y == 4)
        {
            print(head);
            return;
        }
        for (i = 0; i < 8; i += 2)
        {
            xx = x + dir[i];
            yy = y + dir[i + 1];
            if (!vis[xx][yy] && judge(xx, yy))
            {
                vis[xx][yy] = 1;
                list[tail].x = xx;
                list[tail].y = yy;
                pre[tail] = head;
                tail++;
            }
        }
        head++;
    }
    return;
}
```

# N-Find a way

---

## 题目描述

- Yifenfeis's home is at the countryside, but Merceki's home is in the center of city. So yifenfei made arrangements with Merceki to meet at a KFC. There are many KFC in Ningbo, they want to choose one that let the total time to it be most smallest.

Now give you a Ningbo map, Both yifenfei and Merceki can move up, down ,left, right to the adjacent road by cost 11 minutes.

- 宜芬菲的家在农村，但梅尔奇基的家是城市的中心。所以宜芬菲与梅尔奇基安排在肯德基见面。宁波有很多肯德基，他们想选择一个时间最少的一家。现在给你一张宁波地图，伊芬菲和梅切斯基都可以上下左右移动到邻近的道路上。求两人到达所需时间最短肯德基的总时间之和。

# N-Find a way

## Sample Input

5 5

Y, 0, 0, @, 0,  
0, #, 0, 0, 0,  
0, 0, 0, 0, 0,  
@, #, 0, M, 0,  
#, 0, 0, 0, #

→ n行，每行m个元素。

→ 'Y'代表宜芬菲的初始位置。

'M' 代表梅尔奇基的初始位置。

'#' 代表被禁止的路段;

'0' 代表可通行的路段.

'@' 代表KCF

## Sample Output

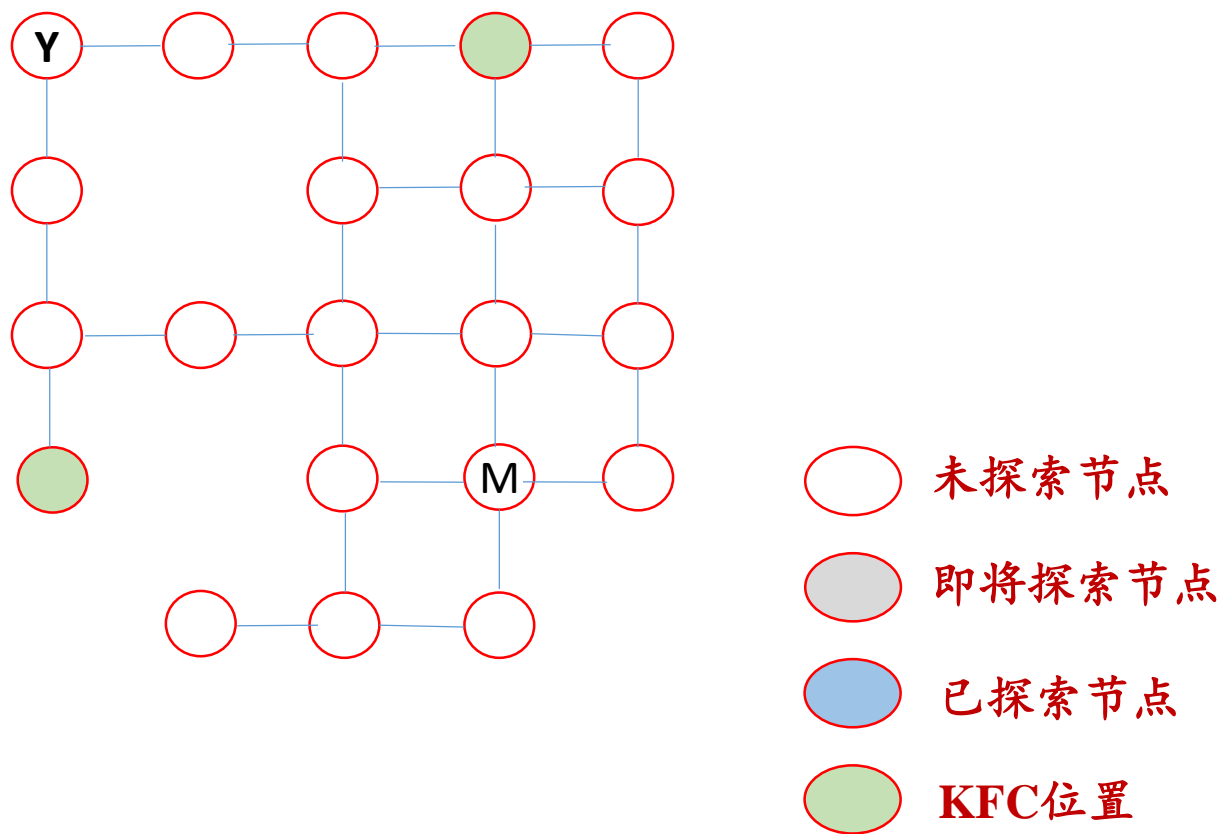
66

## 解题思路

本题为求迷宫最短路径问题，使用BFS下搜索到的第一条路径就是最短路径。  
分别以两个人为起点，对图进行两次的BFS，求出各自到KCF得最短距离，之后比较多个KCF的距离，输出最短的距离。

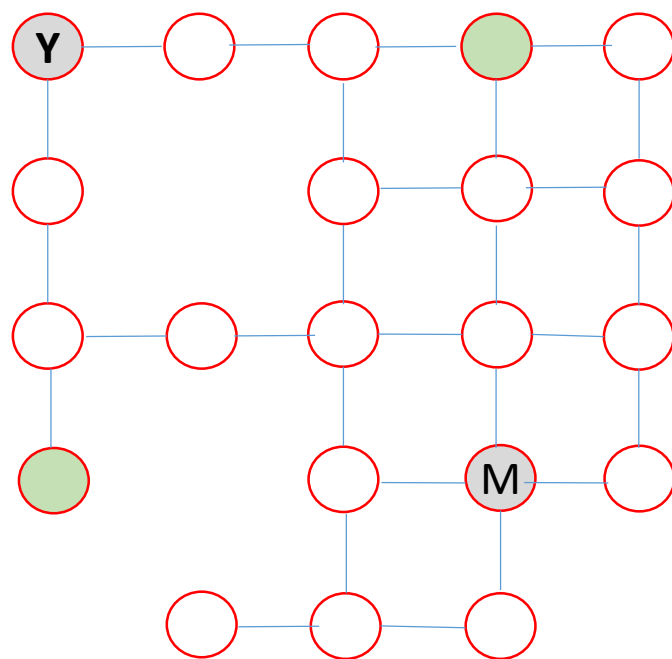
# N-Find a way

## 模拟算法流程



# N-Find a way

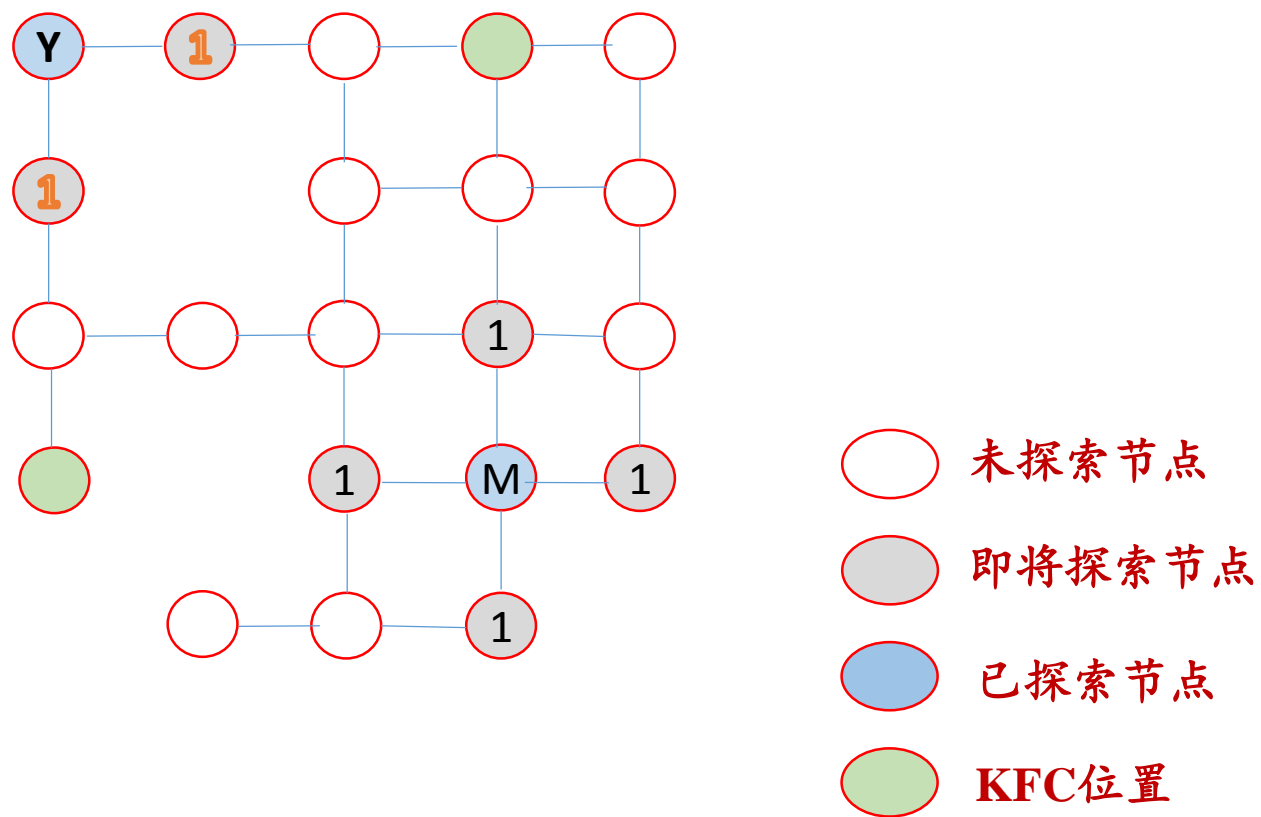
## 模拟算法流程



- 未探索节点
- 即将探索节点
- 已探索节点
- KFC位置

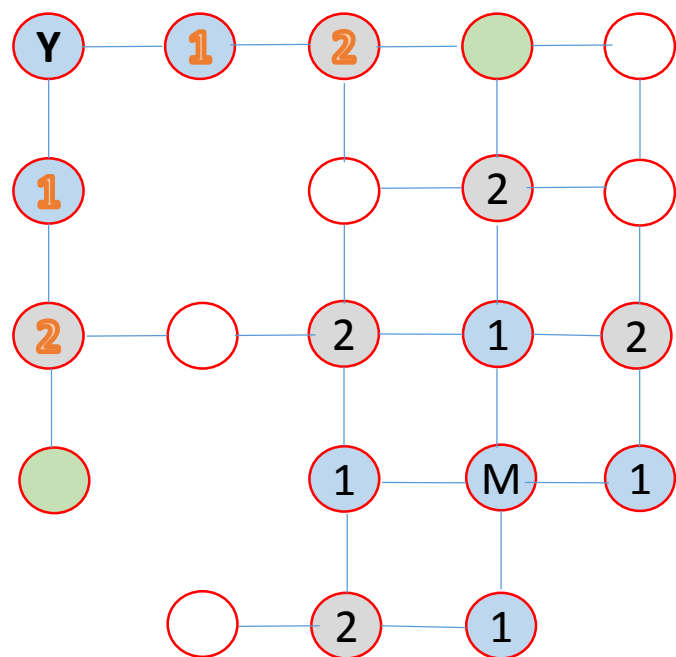
# N-Find a way





## 模拟算法流程



# N-Find a way

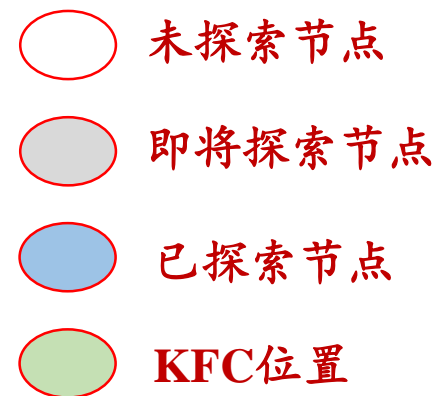
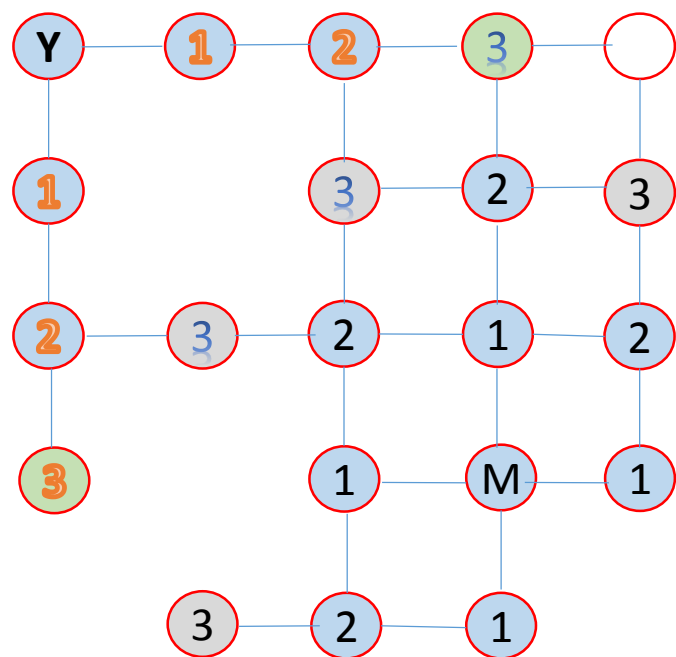
## 模拟算法流程



-  未探索节点
-  即将探索节点
-  已探索节点
-  KFC位置

# N-Find a way

## 模拟算法流程





# N-Find a way

## 实现

```
for (i = 0; i < n; i++)
    for (l = 0; l < m; l++)
        map[i][l] = ori[i][l];
BFS(x_s1, y_s1);

for (i = 0; i < n; i++)
    for (l = 0; l < m; l++)
        map[i][l] = ori[i][l];
BFS(x_s2, y_s2);

Ans = 11111111;
for (i = 0; i < n; i++)
{
    for (l = 0; l < m; l++)
    {
        if (ans[i][l])
        {
            Ans = min(Ans, ans[i][l]);
        }
    }
}
```

```
for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
        if (s[i][j] == '@') /*是KFC，开始计算距离总和*/
            if (dist1[i][j] != 0 && dist2[i][j] != 0)
                if (dist1[i][j] + dist2[i][j] < min) /*比较求距离的最小值*/
                    min = dist1[i][j] + dist2[i][j];
```

# A计划

---

## 题目描述

可怜的公主在一次次被魔王掳走一次次被骑士们救回来之后，而今，不幸的她再一次面临生命的考验。魔王已经发出消息说将在**T时刻**吃掉公主，因为他听信谣言说吃公主的肉也能长生不老。年迈的国王正是心急如焚，告招天下勇士来拯救公主。不过公主早已习以为常，她深信智勇的骑士Acmer肯定能将她救出。 现据密探所报，公主被关在一个两层的迷宫里，迷宫的入口是 $S(0, 0, 0)$ ，**公主的位置用P表示，时空传输机用#表示，墙用\*表示，平地用.表示。**

# A计划

---

## 题目要求

- 骑士一进入时空传输机就会被转到另一层的相对位置，但如果被转到的位置是墙的话，那骑士们就会被撞死。
- 骑士在一层中只能前后左右移动，**每移动一格花1时刻。**
- 层间的移动只能通过时空传输机，**且不需要任何时间。**

# A计划

## Sample Input

1

→ 输入一个测试用例

5 5 14

→ 迷宫大小为5\*5，公主在14时刻被吃掉

S\*#\*.

.#...

→ 第一层

.....

\*\*\*\*.

...#.

..\*.P

#.\*\*\*

\*\*\*..

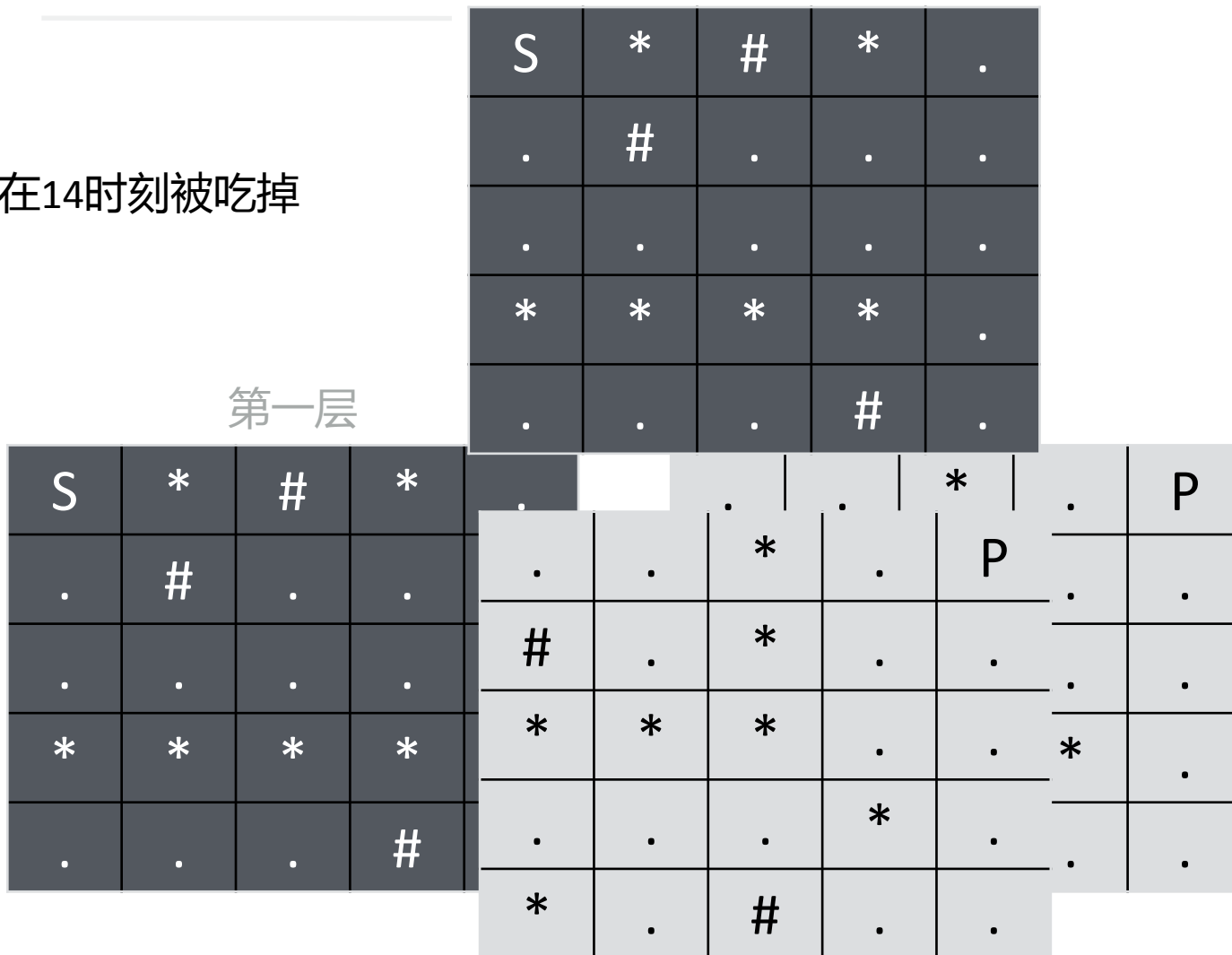
→ 第二层

...\*.

\*.##.

## Sample Input

NO



# A计划

样例输入

x

y

		S	*	#	*	.
.	.	*	.	P	.	.
#	.	*	.	.	.	.
*	*	*	.	.	.	.
.	.	.	*	.	.	.
*	.	#	.	.	.	.

第一层

第二层

# A计划

---

## 思路

- 因为现在每个点多了一个层数信息（Z坐标）。所以不仅仅可以前后左右移动，并且可以上下穿越。第一反应是使用广度优先搜索（使用队列），和以前唯一不同的就是要判断Z坐标的变化。
- 遇到上下两层都是#的，无法移动。

- 双层BFS

```
struct Node
{
    int z;
    int x;
    int y; //记录层数
    int step; //记录步数
};
```

# A计划

## 模拟算法流程

x=0

y=0

S	*	#	*	.
.	#	.	.	.
.	.	.	.	.
*	*	*	*	.
.	.	.	#	.

队列元素(z, x, y, step)

初始队列{(0, 0, 0, 0)}

# A计划

---

## 模拟算法流程

x

y

S	*	#	*	.
.	#	.	.	.
.	.	.	.	.
*	*	*	*	.
.	.	.	#	.

弹出(0, 0, 0, 0)

当前队列

[(0, 0, 1, 1)]



# A计划

---

## 模拟算法流程

	x				
y	S	*	#	*	.
	.	#	.	.	.
	.	.	.	.	.
	*	*	*	*	.
	.	.	.	#	.

当前队列  
[(0, 0, 1, 1)]

# A计划

## 模拟算法流程

x	S	*	#	*	.
y	.	#	.	.	.
.	.	.	.	.	.
*	*	*	*	.	.
.	.	.	#	.	.

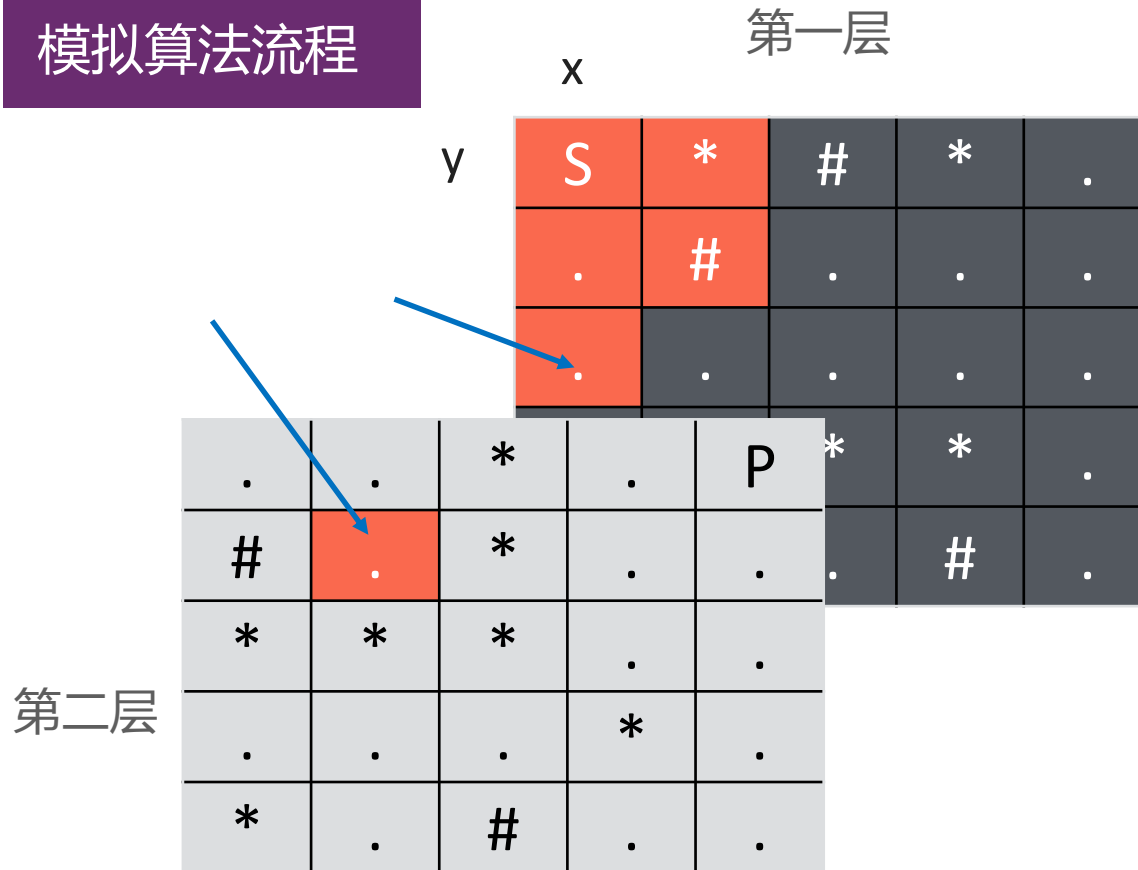
弹出(0, 0, 1, 1)

当前队列

{(0, 1, 1, 2), (0, 0, 2, 2)}

# A计划

## 模拟算法流程



弹出(0, 1, 1, 2)

当前队列

[(0, 0, 2, 2), (**1**, 1, 1, **2**)]

- 穿越时要注意传过去的是不是墙，如果是，则不加入队列。

# A计划


---

## 模拟算法流程

x

y

S	*	#	*	.
.	#	.	.	.
.	.	.	.	.
*	*	*	*	.
.	.	.	#	.



弹出(0, 0, 2, 2)

当前队列

{(1, 1, 1, 2), (0, 1, 2, 3)}

# A计划

## 实现

```
for (int i = 0; i < 4; i++)
{
    int rx = temp.x + dx[i];
    int ny = temp.y + dy[i];
    if (rx >= 0 && ny >= 0 && rx < N && ny < M && maze[temp.l][rx][ny] != '*' && !vis[temp.l][rx][ny]) // 未被访问且不是墙
    {
        if (maze[temp.l][rx][ny] != '#') // 是路 "."
        {
            vis[temp.l][rx][ny] = 1;
            Q.push(Node(temp.l, rx, ny, temp.t + 1));
        }
        else // 是传输机 '#'
        {
            vis[temp.l][rx][ny] = vis[!temp.l][rx][ny] = 1;
            if (maze[!temp.l][rx][ny] != '*' && maze[!temp.l][rx][ny] != '#') // 传输机对应的位置不是墙或传输机 则将节点加入队列
                Q.push(Node(!temp.l, rx, ny, temp.t + 1)); // 设置层数为0和1 用非运算
        }
    }
}
```

# A计划

## 小技巧

为了避免每次穿越时都要注意穿过去的是不是墙，可以在输入数据时做这样的操作：  
如果当前位置可以穿越，则判断穿越过去的是不是墙，**如果是则将当前位置设为墙。**

S	*	#	*	.
.	#	.	.	.
.	.	.	.	.
*	*	*	*	.
.	.	.	#	.

.	.	*	.	P
#	.	*	.	.
*	*	*	.	.
.	.	.	*	.
*	.	#	.	.

# A计划

---

## 小技巧

S	*	*	*	.
.	#	.	.	.
.	.	.	.	.
*	*	*	*	.
.	.	.	#	.

.	.	*	.	P
#	.	*	.	.
*	*	*	.	.
.	.	.	*	.
*	.	#	.	.

# 总结

---

- **BFS**：搜索效率低，占用内存空间大；完备，路径最短  
给定初始状态跟目标状态，要求从初始状态到目标状态的最短路径
- **DFS**：不完备，难以寻找最优解；内存消耗小，每一层只需维护一个节点。  
给定初始状态跟目标状态，要求判断从初始状态到目标状态是否有解。
- 主要差别在于待扩展节点的顺序。

相关题目：

**BFS** POJ3087 POJ3414 POJ3126

**DFS** HDU1241 POJ1416 POJ1564 POJ3411



THANKS

---