DFS+Kosaraju

**007: 深度优先遍历一个无向图**

输出无向图深度优先遍历序列

第一行是整数n和m(0 < n <=16)，表示无向图有n个顶点，m条边，顶点编号0到n-1。接下来m行，每行 两个整数a,b，表示顶点a,b之间有一条边。

输出 任意一个深度优先遍历序列

9 9

0 1

0 2

3 0

2 1

1 5

1 4

4 5

6 3

8 7

样例输出

0 1 2 4 5 3 6 8 7

def dfs(graph, visited, node): visited[node] = True print(node, end=" ")

for neighbor in graph[node]:

if not visited[neighbor]: dfs(graph, visited, neighbor)

def main():

n, m = map(int, input().split()) graph = [[] for \_ in range(n)] visited = [False] \* n

for \_ in range(m):

a, b = map(int, input().split()) graph[a].append(b) graph[b].append(a)

for i in range(n):

if not visited[i]: dfs(graph, visited, i)

if name == " main ": main()

**04123: 马走日**

马在中国象棋以日字形规则移动。

请编写一段程序，给定n\*m大小的棋盘，以及马的初始位置(x，y)，要求不能重复经过棋盘上的同一个 点，计算马可以有多少途径遍历棋盘上的所有点。

### 输入

第一行为整数T(T < 10)，表示测试数据组数。 每一组测试数据包含一行，为四个整数，分别为棋盘的大

小以及初始位置坐标n,m,x,y。(0<=x<=n-1,0<=y<=m-1, m < 10, n < 10)

### 输出

每组测试数据包含一行，为一个整数，表示马能遍历棋盘的途径总数，0为无法遍历一次。

1

5 4 0 0

32

maxn = 10;

sx = [-2,-1,1,2, 2, 1,-1,-2]

sy = [ 1, 2,2,1,-1,-2,-2,-1]

ans = 0;

def Dfs(dep: int, x: int, y: int):

#是否已经全部走完 if n\*m == dep:

global ans ans += 1 return

#对于每个可以走的点

for r in range(8): s = x + sx[r] t = y + sy[r]

if chess[s][t]==False and 0<=s<n and 0<=t<m : chess[s][t]=True

Dfs(dep+1, s, t) chess[s][t] = False; #回溯

for \_ in range(int(input())):

n,m,x,y = map(int, input().split())

chess = [[False]\*maxn for \_ in range(maxn)] #False表示没有走过 ans = 0

chess[x][y] = True Dfs(1, x, y) print(ans)

# **02754:** 八皇后

甚至没用二维列表表示棋盘，好新奇的思路，整体只用了一个记录某行的棋子在哪一列的思路，然后就 是正常八皇后的思路dfs

代码

def is\_safe(board,h,l): for i in range(h):

if board[i]==l: return False

i=h-1 j=l-1

while i>=0 and j>=0: if board[i]==j:

return False

i-=1 j-=1

i=h-1

j=l+1

while i>=0 and j>=0: if board[i]==j:

return False

i-=1 j+=1

return True

def queen(board,h): if h==8:

ans.append(''.join([str(x+1) for x in board])) return

for l in range(8):

if is\_safe(board,h,l): board[h]=l queen(board,h+1) board[h]=None

ans=[] queen([None]\*8,0)

for \_ in range(int(input())): print(ans[int(input())-1])

#答案转置棋盘输出

def is\_safe(board, row, col):

# 检查当前位置是否安全

# 检查同一列是否有皇后

for i in range(row):

if board[i][col] == 1: return False

#

i j

检查左上方是否有皇后

= row

= col while i

- 1

- 1

>= 0 and j >= 0:

if board[i][j] == 1: return False

i -= 1

j -= 1

# 检查右上方是否有皇后

i = row - 1 j = col + 1

while i >= 0 and j < 8:

if board[i][j] == 1: return False

i -= 1

j += 1

return True

def

solve\_n\_queens(board, row,

# 递归回溯求解八皇后问题

if row == 8:

solutions):

# 找到一个解，将解添加到结果列表

solutions.append([board[i].copy() for i in range(8)]) return

for col in range(8):

if is\_safe(board, row, col):

# 当前位置安全，放置皇后

board[row][col] = 1

# 继续递归放置下一行的皇后

solve\_n\_queens(board, row + 1, solutions)

# 回溯，撤销当前位置的皇后

board[row][col] = 0

# 初始化棋盘

board = [[0] \* 8 for \_ in range(8)] solutions = []

# 求解八皇后问题

solve\_n\_queens(board, 0, solutions)

# 输出结果

def transpose\_matrix(matrix):

return [[matrix[j][i] for j in range(len(matrix))] for i in range(len(matrix[0]))]

for i, solution in enumerate(solutions): print(f"No. {i+1}")

for row in transpose\_matrix(solution): print(' '.join(map(str, row)))

**28170: 算鹰**

注意看，这个男人叫小帅。他今天在云顶之弈中有所感悟，决定和捷豹在围棋领域扳扳手腕。他注意 到，对于自己的棋子(如下O)，其上下左右四个位置（如下'!'所示）极其关键。如果四个位置都被对方棋 子占领（如下） 那么自己的棋子就会因为没有“气”而被提子。

#!#

!O!

#!#

而如果这四个位置有己方的子，则可以连成整体，整体的“气”更多，更难被提子。小帅决定称自己通过上 下左右四个方向连成一起的整体为“鹰”，以表示其蓬勃的生机。现在他有一个10x10的小棋盘，其中'-'表 示空位，'.'表示自己的落子，他想写一个程序，看看自己有多少“鹰”。

---.--.-..

-..-.-....

...--....-

----......

--.---....

-.-..-.---

....-.-..-

-..-----..

-.......-.

.....--.--

def dfs(x,y):

graph[x][y] = "-"

for dx,dy in [(1,0),(-1,0),(0,1),(0,-1)]:

if 0<=x+dx<10 and 0<=y+dy<10 and graph[x+dx][y+dy] == ".": dfs(x+dx,y+dy)

graph = []

result = 0

for i in range(10): graph.append(list(input()))

for i in range(10):

for j in range(10):

if graph[i][j] == ".": result += 1 dfs(i,j)

print(result)

## **sy380:** 无向图的连通块 简单

现有一个共n个顶点、m条边的无向图（假设顶点编号为从 0 到 n-1 ），求图中的连通块个数。

### 输入

第一行两个整数n、m，分别表示顶点数和边数； 接下来m行，每行两个整数u、v，表示一条边的两个端点的编号。数据保证不会有重边。 **输出**

输出一个整数，表示图中的连通块个数。

6 5

0 1

0 3

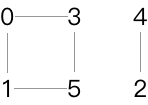
3 5

2 4

1 5

2

对应的无向图如下图所示，共有两个连通块。



以下是使用DFS的Python代码：

def dfs(node, visited, adjacency\_list): visited[node] = True

for neighbor in adjacency\_list[node]: if not visited[neighbor]:

dfs(neighbor, visited, adjacency\_list)

n, m = map(int, input().split()) adjacency\_list = [[] for \_ in range(n)] for \_ in range(m):

u, v = map(int, input().split())

adjacency\_list[u].append(v) adjacency\_list[v].append(u)

visited = [False] \* n connected\_components = 0 for i in range(n):

if not visited[i]:

dfs(i, visited, adjacency\_list) connected\_components += 1

print(connected\_components)

## **sy382:** 有向图判环 中等

现有一个共n个顶点、m条边的有向图（假设顶点编号为从 0 到 n-1 ），如果从图中一个顶点出发，沿着 图中的有向边前进，最后能回到这个顶点，那么就称其为图中的一个环。判断图中是否有环。

### 输入

第一行两个整数n、m，分别表示顶点数和边数； 接下来m行，每行两个整数u、v，表示一条边的起点和终点的编号。数据保证不会有重边。 **输出**

如果图中有环，那么输出 Yes ，否则输出 No 。

在这个问题中，需要检查给定的有向图是否包含一个环。可以使用深度优先搜索（DFS）来解决这个问 题。在DFS中，从一个节点开始，然后访问它的每一个邻居。如果在访问过程中，遇到了一个已经在当前 路径中的节点，那么就存在一个环。**可以使用一个颜色数组来跟踪每个节点的状态：未访问（0），正在 访问（1），已访问（2）。**

以下是解决这个问题的Python代码：

def has\_cycle(n, edges):

graph = [[] for \_ in range(n)] for u, v in edges:

graph[u].append(v) color = [0] \* n

def dfs(node):

if color[node] == 1: return True

if color[node] == 2: return False

color[node] = 1

for neighbor in graph[node]: if dfs(neighbor):

return True color[node] = 2 return False

for i in range(n):

if dfs(i):

return "Yes" return "No"

n, m = map(int, input().split()) edges = []

for \_ in range(m):

u, v = map(int, input().split()) edges.append((u, v))

print(has\_cycle(n, edges))

在这个函数中，我们首先构建了一个邻接列表来表示图。然后，我们对每个节点执行深度优先搜索。如 果在搜索过程中，我们遇到了一个正在访问的节点，那么就存在一个环。如果我们遍历完所有的节点都 没有找到环，那么就返回"No"。

## **sy383:** 最大权值连通块 中等

def max\_weight(n, m, weights, edges): graph = [[] for \_ in range(n)] for u, v in edges:

graph[u].append(v)

graph[v].append(u)

visited = [False] \* n max\_weight = 0

def dfs(node): visited[node] = True

total\_weight = weights[node] for neighbor in graph[node]:

if not visited[neighbor]: total\_weight += dfs(neighbor)

return total\_weight

for i in range(n):

if not visited[i]:

max\_weight = max(max\_weight, dfs(i))

return max\_weight

# 接收数据

n, m = map(int, input().split())

weights = list(map(int, input().split())) edges = []

for \_ in range(m):

u, v = map(int, input().split()) edges.append((u, v))

# 调用函数

print(max\_weight(n, m, weights, edges))

**27635: 判断无向图是否连通有无回路(同23163)**

<http://cs101.openjudge.cn/practice/27635/>

思路：用一个DFS同时判断是否连通以及是否成环，到过的非父节点又到一遍必然成环 代码

n, m = list(map(int, input().split())) edge = [[]for \_ in range(n)]

for \_ in range(m):

a, b = list(map(int, input().split())) edge[a].append(b)

edge[b].append(a)

cnt, flag = set(), False def dfs(x, y):

global cnt, flag cnt.add(x)

for i in edge[x]:

if i not in cnt: dfs(i, x)

elif y != i:

flag = True for i in range(n):

cnt.clear()

dfs(i, -1)

if len(cnt) == n: break

if flag:

break

print("connected:"+("yes" if len(cnt) == n else "no")) print("loop:"+("yes" if flag else 'no'))

# 有向图整张强连通

def

dfs(graph, node, visited, stack): visited[node] = True

for neighbor in graph[node]:

if not visited[neighbor]: dfs(graph, neighbor, visited,

stack.append(node)

stack)

def transpose(graph): transposed = {} for u in graph:

for v in graph[u]:

if v not in transposed: transposed[v] = set()

transposed[v].add(u)

return transposed

def kosaraju(graph): n = len(graph)

visited = [False] \* n stack = []

# Step 1: DFS on original graph to compute finishing times for i in range(n):

if not visited[i]:

dfs(graph, i, visited, stack)

# Step 2: Compute the transpose of the graph transposed\_graph = transpose(graph)

# Step 3: DFS on transposed graph to find SCCs visited = [False] \* n

scc\_count = 0 while stack:

node = stack.pop()

if not visited[node]: scc\_count += 1

dfs(transposed\_graph, node, visited, [])

return scc\_count

# Example usage graph = {

0: {1},

1: {2},

2: {0, 3},

3: {4},

4: {5},

5: {3}

}

sccs = kosaraju(graph) if sccs == 1:

print("Yes") else:

print("No")

**有向图强连通分量**

以下是Kosaraju算法的Python实现：

def dfs1(graph, node, visited, stack): visited[node] = True

for neighbor in graph[node]:

if not visited[neighbor]:

dfs1(graph, neighbor, visited, stack) stack.append(node)

def dfs2(graph, node, visited, component):

visited[node] = True component.append(node)

for neighbor in graph[node]:

if not visited[neighbor]:

dfs2(graph, neighbor, visited, component) def kosaraju(graph):

# Step 1: Perform first DFS to get finishing times

stack = []

visited = [False] \* len(graph) for node in range(len(graph)):

if not visited[node]:

dfs1(graph, node, visited, stack)

# Step 2: Transpose the graph

transposed\_graph = [[] for \_ in range(len(graph))] for node in range(len(graph)):

for neighbor in graph[node]: transposed\_graph[neighbor].append(node)

# Step 3: Perform second DFS on the transposed graph to find SCCs visited = [False] \* len(graph)

sccs = [] while stack:

node = stack.pop()

if not visited[node]: scc = []

dfs2(transposed\_graph, node, visited, scc) sccs.append(scc)

return sccs

# Example

graph = [[1], [2, 4], [3, 5], [0, 6], [5], [4], [7], [5, 6]]

sccs = kosaraju(graph)

print("Strongly Connected Components:") for scc in sccs:

print(scc)

Strongly Connected Components: [0, 3, 2, 1]

[6, 7]

[5, 4]