BFS 在图中的应用

BFS 在图论和计算机科学中有多种应用，包括：

**最短路径查找**： BFS 可用于查找未加权图中两个节点之间的最短路径。通过在遍历过程中跟踪每个节点的父节点，可以重建最短路径。

**循环检测**： BFS 可用于检测图形中的循环。如果在遍历过程中访问了一个节点两次，则表示存在一个循环。

连通分量： BFS 可用于识别图中的连通分量。每个连通的分量都是一组可以相互访问的节点。

**拓扑排序（topological sort）**： BFS 可用于对有向无环图 （DAG） 执行拓扑排序。拓扑排序按线性顺序排列节点，以便对于任何边 （u， v），u 按顺序显示在 v 之前。

**二叉树的级别顺序遍历**： BFS 可用于执行二叉树的级别顺序遍历。此遍历在移动到下一个级别之前访问同一级别的所有节点。

权限分层型Visited：

如你在一个迷宫里最多杀死k只蛇，此时的visited可以按照已经杀死的蛇数和x,y坐标排布，然后用deque和bfs去做（visited=[[[False ]\*(k)for j in range(m) ]for c in range(n)]

在dfs下的回溯写法（骑士周游）：如果⼀个顶点的所有相邻顶点都已被访问过，但是路径⻓度仍然没有达到64，就说明遇到了死路。如果遇到死路，就必须回溯。当从 knight\_tour 返回False时，就会发⽣回溯。在宽度优先搜索中，我们使⽤了队列来记录将要访问的顶点。由于深度优先搜索是递归的，因此我们隐式地使⽤⼀个栈来回溯。当从 knight\_tour 调⽤返回False时，仍然在循环中，并且会查看 neighbors 中的下⼀个顶点。



颜色标志法（white未解决，grey解决中，black已解决）解决是否存在循环向环的问题：  
def has\_cycle(n, edges):

# 输入验证

if n <= 0 or m < 0:

return "Invalid input"

# 构建图

graph = [[] for \_ in range(n)]

for u, v in edges:

if u >= n or v >= n:

return "Invalid vertex id"

graph[u].append(v)

color = [0] \* n # 0:未访问, 1:访问中, 2:已访问

def dfs(node):

if color[node] == 1:#在正在访问的路径上形成闭环

return True

if color[node] == 2:#访问到了之前检查过的死亡路径，无法和现在的1标志路径形成闭环

return False

color[node] = 1

for neighbor in graph[node]:

if dfs(neighbor):#嵌套方法让内部的小死路先标志成2，之后继续筛查其他分路为1的路径

return True

color[node] = 2#标志死亡

return False

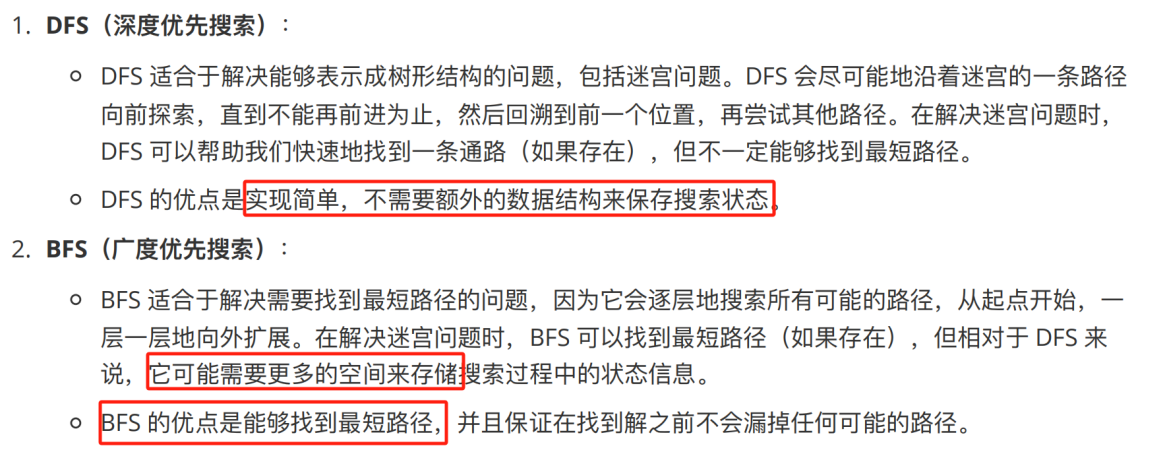
for i in range(n):

if color[i] == 0: # 只处理未访问节点

if dfs(i):

return "Yes"

return "No"



## 记忆化+颜色标记在同一个列表判环

class Solution:

def largestPathValue(self, colors: str, edges: List[List[int]]) -> int:

n = len(colors)

g = [[] for \_ in range(n)]

for x, y in edges:

if x == y: # 自环

return -1

g[x].append(y)

memo = [None] \* n#memo同时承担**判环和记录**两个责任

def dfs(x: int) -> Dict[str, int]:

if memo[x] is not None: # x 计算中或者计算过

return memo[x] # 如果是 0，表示有环

memo[x] = 0 # 用 0 表示计算中#dfs的函数表示更容易用标记颜色法，计算完之后记忆化，在最后的标注处写上dp值（二维dp，即该处为止各颜色的最大长度）

res = defaultdict(int)

for y in g[x]:

cy = dfs(y)

if not cy: # 有环#如果是没处理，在dfs它的时候for这一步就不会进行，直接返回一个列表其中自己颜色为1，这也是res[colors[x]] += 1写在循环外的好处所在，此时not cy只有可能是0

return cy

for ch, c in cy.items():

res[ch] = max(res[ch], c)

res[colors[x]] += 1

memo[x] = res # 记忆化，同时也表示 x 计算完毕

return res

ans = 0

for x, c in enumerate(colors):

res = dfs(x)

if not res: # 有环

return -1

ans = max(ans, res[c])

return ans

**自写：dfs的用stack保留从start到end路径的回溯算法：**

Example：  
mazelist = [

"++++++++++++++++++++++",

"+ + ++ ++ +",

"E + ++++++++++",

"+ + ++ ++++ +++ ++",

"+ + + + ++ +++ +",

"+ ++ ++ + +",

"+++++ + + ++ + +",

"+++++ +++ + + ++ +",

"+ + + S+ + +",

"+++++ + + + + + +",

"++++++++++++++++++++++",

]

stack=[]

dire=[(-1,0),(0,1),(0,-1),(1,0)]

havegone=[[False]\*len(mazelist[0]) for \_ in range(len(mazelist))]

def dfs(i,j):

if mazelist[i][j] == "E":

stack.append((i,j))

return True

havegone[i][j] = True

stack.append((i,j))

for m,k in dire:

if 0<=i+m<len(mazelist) and 0<=j+k<len(mazelist[0]) and mazelist[i+m][j+k]!="+":

if not havegone[i+m][j+k]:

if not dfs(i+m,j+k):

havegone[i+m][j+k] = False

stack.pop()

else:

return True

return False

print(dfs(8,15))

print(\*stack)

print(mazelist[8][15])

## 拓扑排序（topological sort）

算法：对有向无环图（DAG）进⾏排序的⼀种算法。它将图中的顶点按照⼀种线性顺序进⾏排列，使得对于任意的有向边 (u, v)，顶点 u 在排序中出现在顶点 v 的前⾯

Bfs适合从多个出发点发起的最短距离

如<http://cs101.openjudge.cn/practice/22508/>

Kahn排序（bfs，拓扑排序，DAG有向无环图）

def topological\_sort(graph):

indegree = defaultdict(int)

result = []

queue = deque()

# 计算每个顶点的⼊度

for u in graph:

for v in graph[u]:

indegree[v] += 1

# 将⼊度为 0 的顶点加⼊队列

for u in graph:

if indegree[u] == 0:

queue.append(u)

# 执⾏拓扑排序

while queue:

u = queue.popleft()

result.append(u)

for v in graph[u]:

indegree[v] -= 1

if indegree[v] == 0: #此时，距根部距离最远的树枝已经伸展到了v点，此时就是他入deque之时！

queue.append(v)

# 检查是否存在环

if len(result) == len(graph):

return result

else:

return None

from collections import deque, defaultdict

def kahn\_check(adj, nodes):

"""

改进的 Kahn 算法，用来判断**当前图**是否：

- 有环，返回 0

- 唯一拓扑序，返回该序列 list

- 多解，返回 None

"""

in\_deg = {u: 0 for u in nodes}

for u in adj:

for v in adj[u]:

in\_deg[v] += 1

q = deque(u for u in nodes if in\_deg[u] == 0)

topo = []

unique = True

while q:

if len(q) > 1:

unique = False

u = q.popleft()

topo.append(u)

for v in adj[u]:

in\_deg[v] -= 1

if in\_deg[v] == 0:

q.append(v)

if len(topo) < len(nodes):

return 0 # 有环

return topo if unique else None

def forward\_reachable(adj, start, pos, hi):

"""

从 start 沿出边做 BFS，只收集那些 pos[w] ≤ hi 的节点。

"""

seen = {start}

q = deque([start])

while q:

u = q.popleft()

for v in adj[u]:

if v not in seen and pos[v] <= hi:

seen.add(v)

q.append(v)

return seen

def solve():

import sys

for line in sys.stdin:

line = line.strip().split()

if not line:

continue

n, m = map(int, line)

if n == 0 and m == 0:

break

# 节点 A, B, ..., chr(ord('A')+n-1)

nodes = [chr(ord('A') + i) for i in range(n)]

adj = defaultdict(list)

# 初始序列就是 A, B, C, ...

order = nodes[:]

pos = {u:i for i,u in enumerate(order)}

def insert\_edge(u, v):

"""

插入 u->v：

1) 如果 pos[u] < pos[v]，无需调整

2) 否则

F = 所有能从 v 出发（沿原图）到达且 pos[...] ≤ pos[u] 的节点

若 u∈F 则成环，返回 False

否则把 F 从 order 中摘出，插到 u 之后，更新 pos，返回 True

"""

adj[u].append(v)

if pos[u] < pos[v]:

return True

F = forward\_reachable(adj, v, pos, pos[u])

if u in F:

return False

# 从 order 中删掉 F

new\_order = [w for w in order if w not in F]

# 找 u 在删完 F 后的新位置

idx = new\_order.index(u)

# 保持 F 在旧 order 中的相对次序

F\_list = [w for w in order if w in F]

# 重建 order：u 之后插入 F\_list

order.clear()

order.extend(new\_order[:idx+1])

order.extend(F\_list)

order.extend(new\_order[idx+1:])

# 更新 pos

for i,w in enumerate(order):

pos[w] = i

return True

# 读入每条关系，在线处理

rels = [sys.stdin.readline().strip() for \_ in range(m)]

done = False

for i, rel in enumerate(rels, start=1):

u, v = rel.split('<')

ok = insert\_edge(u, v)

if not ok:

print(f"Inconsistency found after {i} relations.")

done = True

break

chk = kahn\_check(adj, nodes)

if isinstance(chk, list):

print(f"Sorted sequence determined after {i} relations: {''.join(chk)}.")

done = True

break

if not done:

print("Sorted sequence cannot be determined.")

if \_\_name\_\_ == "\_\_main\_\_":

solve()

## 最小生成树（Kruskal算法，Prim算法）

1. Kruskal算法

Kruskal算法是一种贪心算法，它按照边的权重从小到大依次选择边，如果这条边连接的两个顶点不在同一个连通分量中，就将这条边加入最小生成树。

算法步骤

将图中的所有边按权重从小到大排序

初始化一个空集合T，用于存放最小生成树的边

按顺序考虑每条边(u,v):

如果u和v不在同一个连通分量中(即加入这条边不会形成环)

则将这条边加入T

合并u和v所在的连通分量

当T中的边数等于顶点数减1时，算法结束

通常使用并查集(Disjoint Set Union)数据结构来高效地判断和合并连通分量

1. prim算法

Prim算法

基本思想

Prim算法从**一个顶点开始，逐步扩展最小生成树**，每次选择连接树与非树顶点且权重最小的边加入树中。

算法步骤

选择任意一个顶点作为**起始点，加入集合S(已包含在MST中的顶点)**

初始化一个优先队列(最小堆)，存放所有连接S与非S顶点的边

从优先队列中**取出权重最小的边(u,v)**，其中u∈S，v∉S

将v加入S，边(u,v)加入MST

将与v相连的所有边加入优先队列

def prim(graph, n):

"""Prim 算法求解最小生成树"""

visited = [False] \* n

min\_heap = [(0, 0)] # (weight, node)

min\_weight = 0

while min\_heap:

weight, u = heapq.heappop(min\_heap)

if not visited[u]:

visited[u] = True

min\_weight += weight

for v, w in graph[u]:

if not visited[v]:

heapq.heappush(min\_heap, (w, v))

return min\_weight