Assignment #9: 图论：遍历，及 树算

Updated 1739 GMT+8 Apr 14, 2024

2024 spring, Complied by ==同学的姓名、院系==

说明：

1）请把每个题目解题思路（可选），源码Python, 或者C++（已经在Codeforces/Openjudge上AC），截图（包含Accepted），填写到下面作业模版中（推荐使用 typora https://typoraio.cn ，或者用word）。AC 或者没有AC，都请标上每个题目大致花费时间。

2）提交时候先提交pdf文件，再把md或者doc文件上传到右侧“作业评论”。Canvas需要有同学清晰头像、提交文件有pdf、"作业评论"区有上传的md或者doc附件。

3）如果不能在截止前提交作业，请写明原因。

编程环境

==（请改为同学的操作系统、编程环境等）==

操作系统：macOS Ventura 13.4.1 (c)

Python编程环境：Spyder IDE 5.2.2, PyCharm 2023.1.4 (Professional Edition)

C/C++编程环境：Mac terminal vi (version 9.0.1424), g++/gcc (Apple clang version 14.0.3, clang-1403.0.22.14.1)

1. 题目

04081: 树的转换

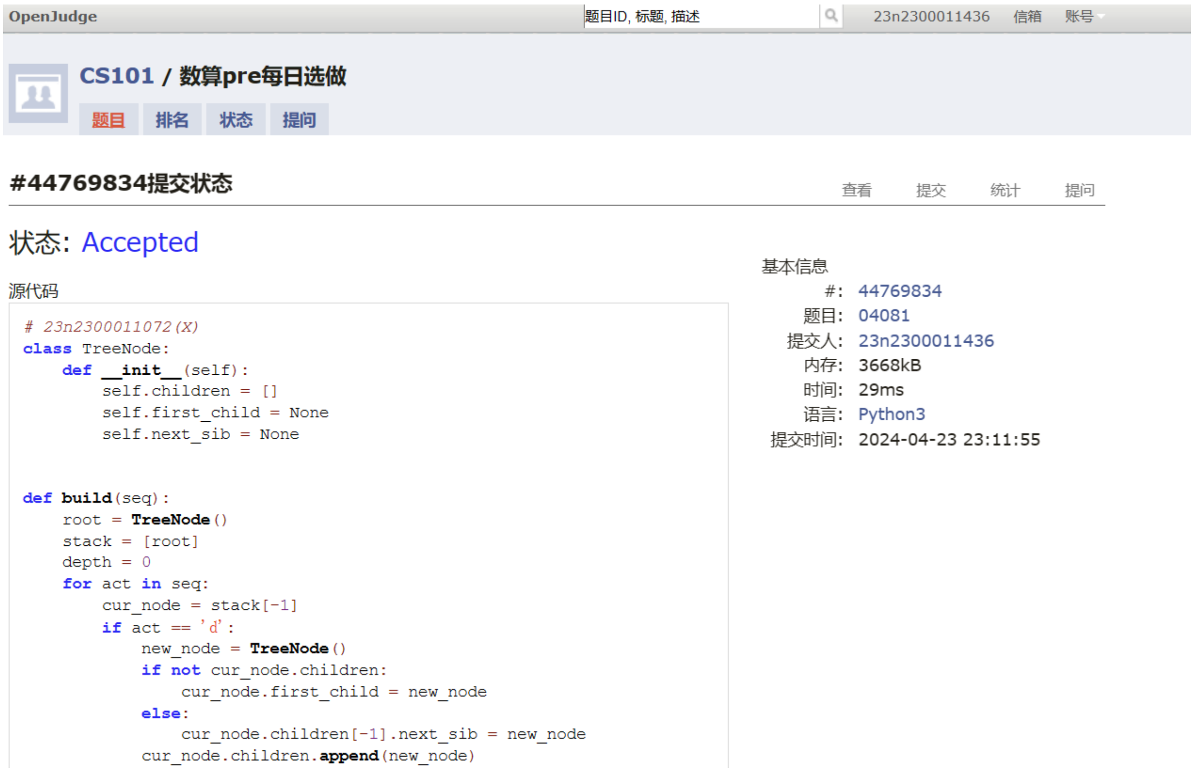
http://cs101.openjudge.cn/dsapre/04081/

思路：

代码

```
# 23n2300011072(X)
class TreeNode:
    def __init__(self):
        self.children = []
        self.first_child = None
        self.next_sib = None


def build(seq):
    root = TreeNode()
    stack = [root]
    depth = 0
    for act in seq:
        cur_node = stack[-1]
        if act == 'd':
            new_node = TreeNode()
            if not cur_node.children:
                cur_node.first_child = new_node
            else:
                cur_node.children[-1].next_sib = new_node
            cur_node.children.append(new_node)
            stack.append(new_node)
            depth = max(depth, len(stack) - 1)
        else:
            stack.pop()
    return root, depth


def cal_h_bin(node):
    if not node:
         return -1
    return max(cal_h_bin(node.first_child), cal_h_bin(node.next_sib)) + 1


seq = input()
root, h_orig = build(seq)
h_bin = cal_h_bin(root)
print(f'{h_orig} => {h_bin}')xxxxxxxxxx # 23n2300011072(X)class TreeNode:    def
__init__(self):        self.children = []        self.first_child = None
 self.next_sib = Nonedef build(seq):    root = TreeNode()    stack = [root]
 depth = 0    for act in seq:        cur_node = stack[-1]        if act == 'd':
        new_node = TreeNode()            if not cur_node.children:
 cur_node.first_child = new_node            else:
 cur_node.children[-1].next_sib = new_node
 cur_node.children.append(new_node)            stack.append(new_node)
 depth = max(depth, len(stack) - 1)        else:            stack.pop()    return
root, depthdef cal_h_bin(node):    if not node:         return -1    return
max(cal_h_bin(node.first_child), cal_h_bin(node.next_sib)) + 1seq = input()root,
h_orig = build(seq)h_bin = cal_h_bin(root)print(f'{h_orig} => {h_bin}')#
```

代码运行截图 ==（至少包含有"Accepted"）==



#44769834提交状态

状态: Accepted

源代码

```
# 23n2300011072(X)
class TreeNode:
    def __init__(self):
        self.children = []
        self.first_child = None
        self.next_sib = None


def build(seq):
    root = TreeNode()
    stack = [root]
    depth = 0
    for act in seq:
        cur_node = stack[-1]
        if act == 'd':
            new_node = TreeNode()
            if not cur_node.children:
                cur_node.first_child = new_node
            else:
                cur_node.children[-1].next_sib = new_node
            cur_node.children.append(new_node)
```

基本信息

#:	44769834
题目:	04081
提交人:	23n2300011436
内存:	3668kB
时间:	29ms
语言:	Python3
提交时间:	2024-04-23 23:11:55

08581: 扩展二叉树

http://cs101.openjudge.cn/dsapre/08581/

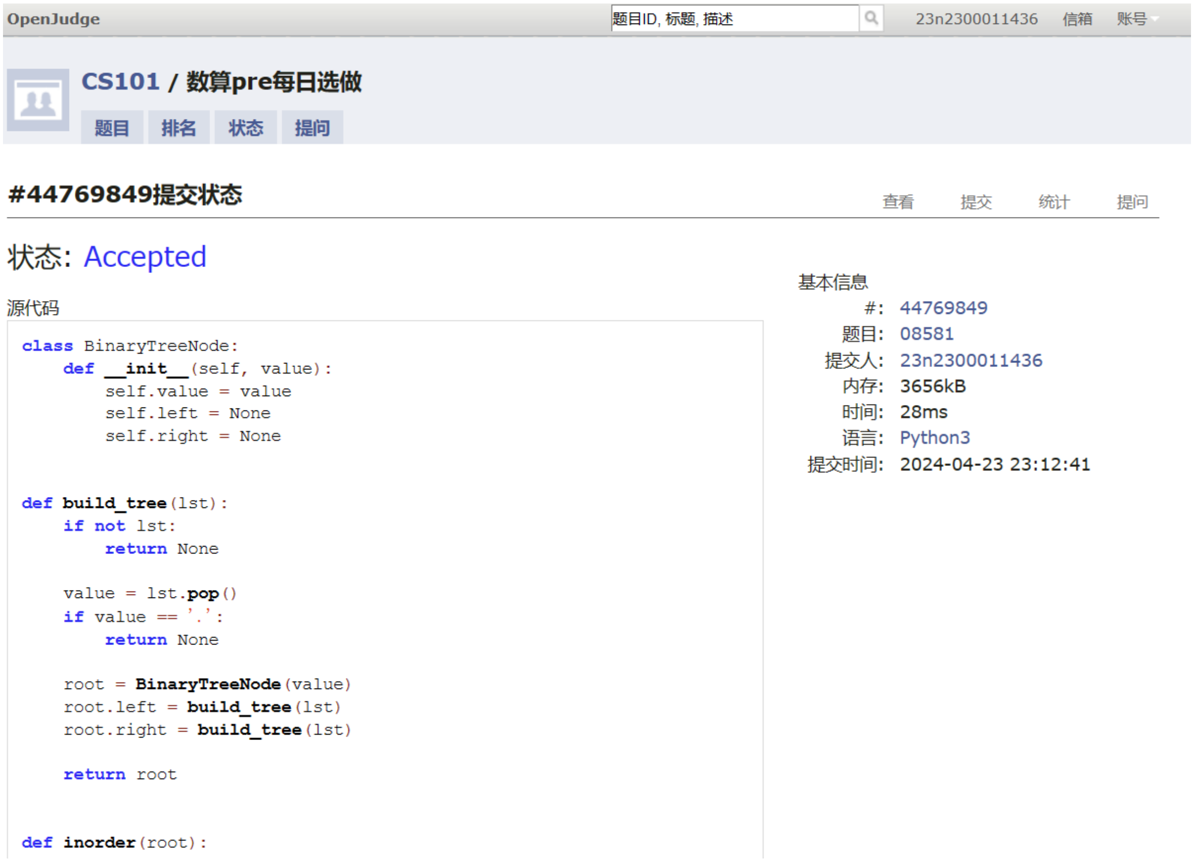
思路：

代码

```
class BinaryTreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None


def build_tree(lst):
    if not lst:
        return None

    value = lst.pop()
    if value == '.':
        return None

    root = BinaryTreeNode(value)
    root.left = build_tree(lst)
    root.right = build_tree(lst)
```

```python
    return root


def inorder(root):
    if not root:
        return []

    left = inorder(root.left)
    right = inorder(root.right)
    return left + [root.value] + right


def postorder(root):
    if not root:
        return []

    left = postorder(root.left)
    right = postorder(root.right)
    return left + right + [root.value]


lst = list(input())
root = build_tree(lst[::-1])
in_order_result = inorder(root)
post_order_result = postorder(root)
print(''.join(in_order_result))
print(''.join(post_order_result))
```

代码运行截图 == （至少包含有"Accepted"） ==



CS101 / 数算pre每日选做

#44769849提交状态

状态: Accepted

源代码

```python
class BinaryTreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None


def build_tree(lst):
    if not lst:
        return None

    value = lst.pop()
    if value == '.':
        return None

    root = BinaryTreeNode(value)
    root.left = build_tree(lst)
    root.right = build_tree(lst)

    return root


def inorder(root):
```

基本信息

#:	44769849
题目:	08581
提交人:	23n2300011436
内存:	3656kB
时间:	28ms
语言:	Python3
提交时间:	2024-04-23 23:12:41

22067: 快速堆猪

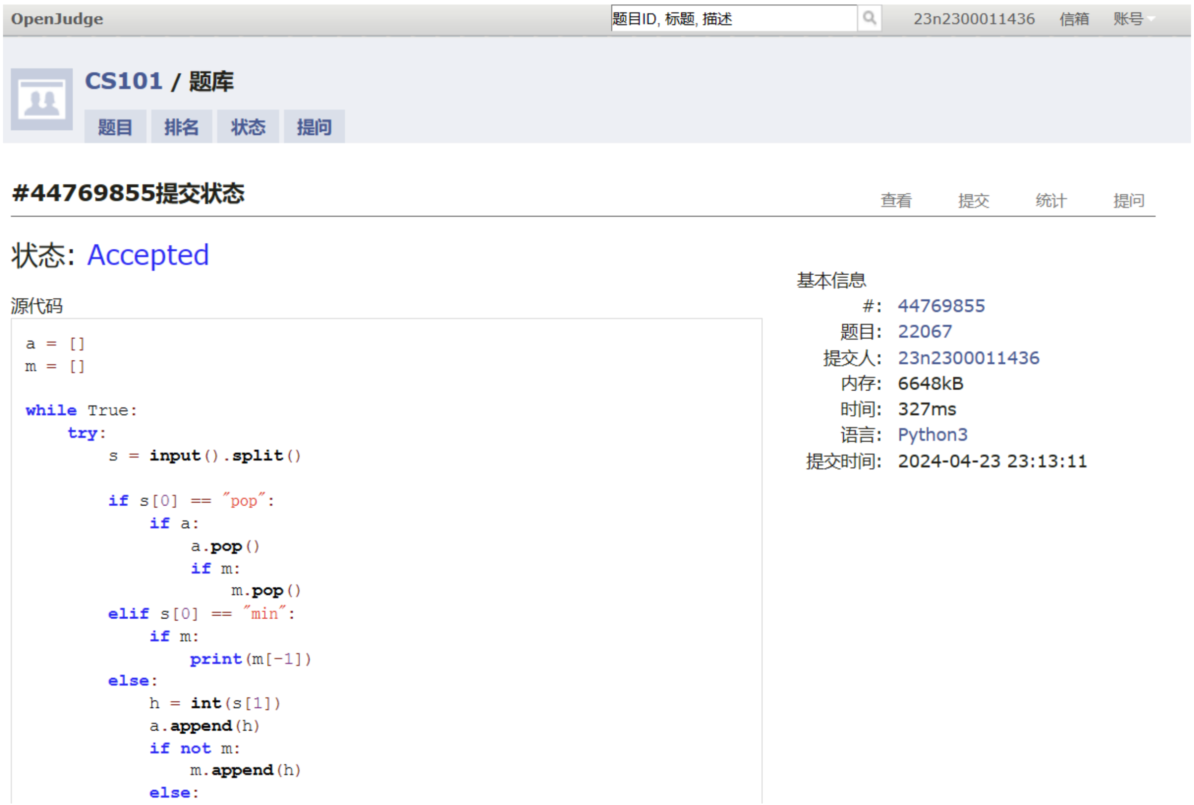
http://cs101.openjudge.cn/practice/22067/

思路：

代码

```python
a = []
m = []

while True:
    try:
        s = input().split()

        if s[0] == "pop":
            if a:
                a.pop()
                if m:
                    m.pop()
        elif s[0] == "min":
            if m:
                print(m[-1])
        else:
            h = int(s[1])
            a.append(h)
            if not m:
                m.append(h)
            else:
                k = m[-1]
                m.append(min(k, h))
    except EOFError:
        break
```

代码运行截图 ==（AC代码截图，至少包含有"Accepted"）==



#44769855提交状态

状态: Accepted

源代码

```
a = []
m = []

while True:
    try:
        s = input().split()

        if s[0] == "pop":
            if a:
                a.pop()
                if m:
                    m.pop()
        elif s[0] == "min":
            if m:
                print(m[-1])
        else:
            h = int(s[1])
            a.append(h)
            if not m:
                m.append(h)
            else:
```

基本信息

#:	44769855
题目:	22067
提交人:	23n2300011436
内存:	6648kB
时间:	327ms
语言:	Python3
提交时间:	2024-04-23 23:13:11

04123: 马走日

dfs, http://cs101.openjudge.cn/practice/04123

思路：

代码

```
#
```

代码运行截图 ==（AC代码截图，至少包含有"Accepted"）==

28046: 词梯

bfs, http://cs101.openjudge.cn/practice/28046/

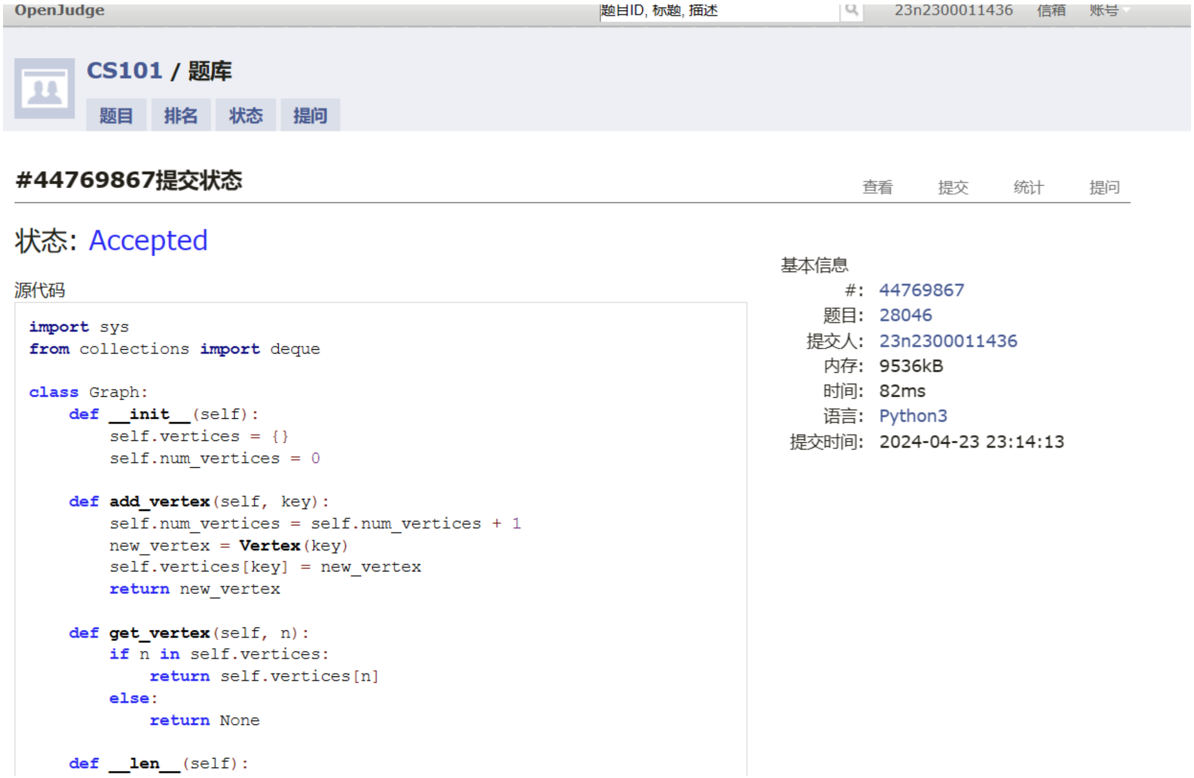
思路：

代码

```python
import sys
from collections import deque

class Graph:
    def __init__(self):
        self.vertices = {}
        self.num_vertices = 0

    def add_vertex(self, key):
        self.num_vertices = self.num_vertices + 1
        new_vertex = Vertex(key)
        self.vertices[key] = new_vertex
        return new_vertex

    def get_vertex(self, n):
        if n in self.vertices:
            return self.vertices[n]
        else:
            return None

    def __len__(self):
        return self.num_vertices

    def __contains__(self, n):
        return n in self.vertices

    def add_edge(self, f, t, cost=0):
        if f not in self.vertices:
            nv = self.add_vertex(f)
        if t not in self.vertices:
            nv = self.add_vertex(t)
        self.vertices[f].add_neighbor(self.vertices[t], cost)

    def get_vertices(self):
        return list(self.vertices.keys())

    def __iter__(self):
        return iter(self.vertices.values())


class Vertex:
    def __init__(self, num):
        self.key = num
        self.connectedTo = {}
        self.color = 'white'
        self.distance = sys.maxsize
        self.previous = None
        self.disc = 0
        self.fin = 0

    def add_neighbor(self, nbr, weight=0):
        self.connectedTo[nbr] = weight

    # def setDiscovery(self, dtime):
    #     self.disc = dtime
    #
    # def setFinish(self, ftime):
    #     self.fin = ftime
    #
    # def getFinish(self):
    #     return self.fin
    #
    # def getDiscovery(self):
    #     return self.disc

    def get_neighbors(self):
        return self.connectedTo.keys()

    # def getWeight(self, nbr):
    #     return self.connectedTo[nbr]

    # def __str__(self):
    #     return str(self.key) + ":color " + self.color + ":disc " +
str(self.disc) + ":fin " + str(
    #         self.fin) + ":dist " + str(self.distance) + ":pred \n\t[" +
str(self.previous) + "]\n"



def build_graph(all_words):
    buckets = {}
    the_graph = Graph()

    # 创建词桶 create buckets of words that differ by 1 letter
    for line in all_words:
        word = line.strip()
        for i, _ in enumerate(word):
            bucket = f"{word[:i]}_{word[i + 1:]}"
            buckets.setdefault(bucket, set()).add(word)

    # 为同一个桶中的单词添加顶点和边
    for similar_words in buckets.values():
        for word1 in similar_words:
            for word2 in similar_words - {word1}:
                the_graph.add_edge(word1, word2)

    return the_graph


def bfs(start, end):
    start.distnce = 0
    start.previous = None
    vert_queue = deque()
    vert_queue.append(start)
    while len(vert_queue) > 0:
        current = vert_queue.popleft()  # 取队首作为当前顶点

        if current == end:
            return True
```

```python
        for neighbor in current.get_neighbors():  # 遍历当前顶点的邻接顶点
            if neighbor.color == "white":
                neighbor.color = "gray"
                neighbor.distance = current.distance + 1
                neighbor.previous = current
                vert_queue.append(neighbor)
        current.color = "black"  # 当前顶点已经处理完毕，设黑色

    return False

"""
BFS 算法主体是两个循环的嵌套: while-for
    while 循环对图中每个顶点访问一次，所以是 O(|V|)；
    嵌套在 while 中的 for，由于每条边只有在其起始顶点u出队的时候才会被检查一次，
    而每个顶点最多出队1次，所以边最多被检查次，一共是 O(|E|)；
    综合起来 BFS 的时间复杂度为 0(V+|E|)

词梯问题还包括两个部分算法
    建立 BFS 树之后，回溯顶点到起始顶点的过程，最多为 O(|V|)
    创建单词关系图也需要时间，时间是 O(|V|+|E|) 的，因为每个顶点和边都只被处理一次
"""



def traverse(starting_vertex):
    ans = []
    current = starting_vertex
    while (current.previous):
        ans.append(current.key)
        current = current.previous
    ans.append(current.key)

    return ans


n = int(input())
all_words = []
for _ in range(n):
    all_words.append(input().strip())

g = build_graph(all_words)
# print(len(g))

s, e = input().split()
start, end = g.get_vertex(s), g.get_vertex(e)
if start is None or end is None:
    print('NO')
    exit(0)

if bfs(start, end):
    ans = traverse(end)
    print(' '.join(ans[::-1]))
else:
    print('NO')
```

代码运行截图 ==（AC代码截图，至少包含有"Accepted"）==



#44769867提交状态

状态: Accepted

源代码

```
import sys
from collections import deque

class Graph:
    def __init__(self):
        self.vertices = {}
        self.num_vertices = 0

    def add_vertex(self, key):
        self.num_vertices = self.num_vertices + 1
        new_vertex = Vertex(key)
        self.vertices[key] = new_vertex
        return new_vertex

    def get_vertex(self, n):
        if n in self.vertices:
            return self.vertices[n]
        else:
            return None

    def __len__(self):
```

基本信息

#:	44769867
题目:	28046
提交人:	23n2300011436
内存:	9536kB
时间:	82ms
语言:	Python3
提交时间:	2024-04-23 23:14:13

28050: 骑士周游

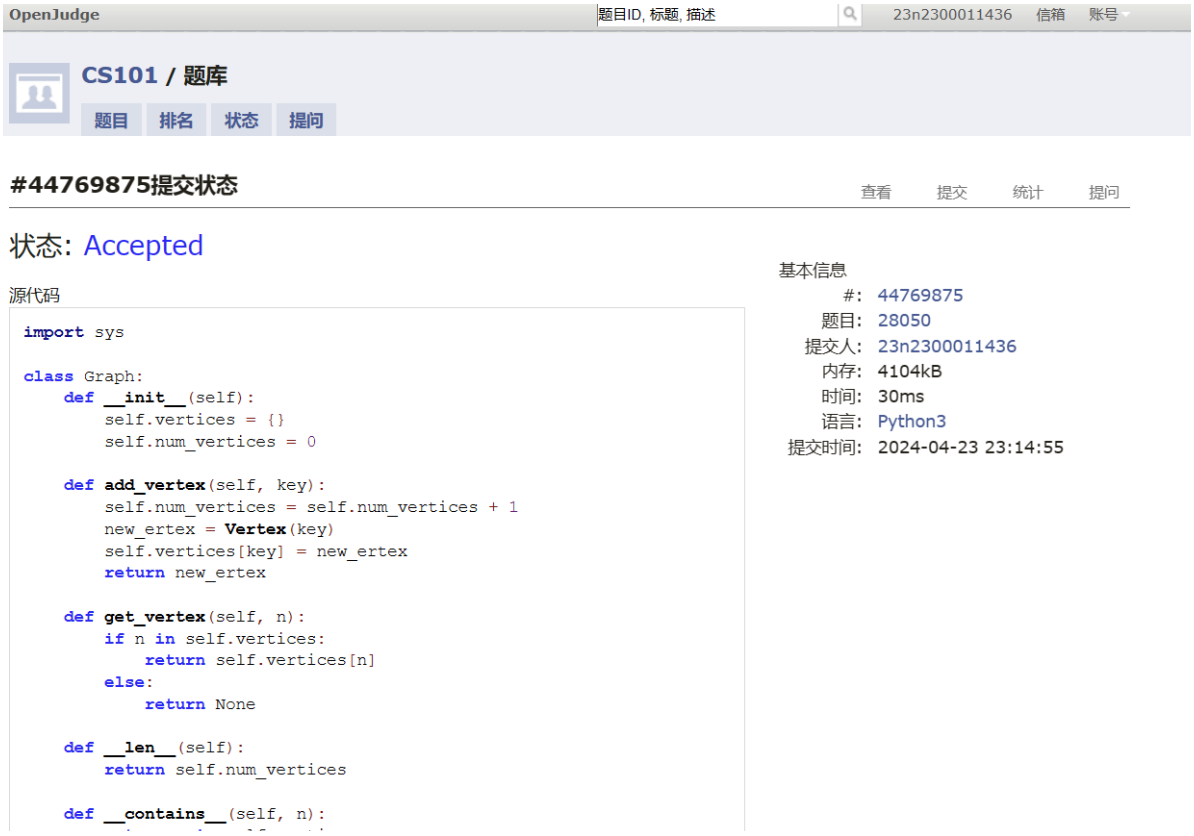
dfs, http://cs101.openjudge.cn/practice/28050/

思路：

代码

```
import sys

class Graph:
    def __init__(self):
        self.vertices = {}
        self.num_vertices = 0

    def add_vertex(self, key):
        self.num_vertices = self.num_vertices + 1
        new_ertex = Vertex(key)
        self.vertices[key] = new_ertex
        return new_ertex

    def get_vertex(self, n):
        if n in self.vertices:
            return self.vertices[n]
        else:
            return None

    def __len__(self):
        return self.num_vertices

    def __contains__(self, n):
        return n in self.vertices

    def add_edge(self, f, t, cost=0):
        if f not in self.vertices:
            nv = self.add_vertex(f)
        if t not in self.vertices:
            nv = self.add_vertex(t)
        self.vertices[f].add_neighbor(self.vertices[t], cost)
        #self.vertices[t].add_neighbor(self.vertices[f], cost)

    def getVertices(self):
        return list(self.vertices.keys())

    def __iter__(self):
        return iter(self.vertices.values())


class Vertex:
    def __init__(self, num):
        self.key = num
        self.connectedTo = {}
        self.color = 'white'
        self.distance = sys.maxsize
        self.previous = None
        self.disc = 0
        self.fin = 0

    def __lt__(self,o):
        return self.key < o.key

    def add_neighbor(self, nbr, weight=0):
        self.connectedTo[nbr] = weight


    # def setDiscovery(self, dtime):
    #     self.disc = dtime
    #
    # def setFinish(self, ftime):
    #     self.fin = ftime
    #
    # def getFinish(self):
    #     return self.fin
    #
    # def getDiscovery(self):
    #     return self.disc

    def get_neighbors(self):
        return self.connectedTo.keys()

    # def getWeight(self, nbr):
    #     return self.connectedTo[nbr]
    def __str__(self):
        return str(self.key) + ":color " + self.color + ":disc " + str(self.disc) + ":fin " + str(
            self.fin) + ":dist " + str(self.distance) + ":pred \n\t[" + str(self.previous) + "]\n"



def knight_graph(board_size):
    kt_graph = Graph()
    for row in range(board_size):           #遍历每一行
        for col in range(board_size):       #遍历行上的每一个格子
            node_id = pos_to_node_id(row, col, board_size) #把行、列号转为格子ID
            new_positions = gen_legal_moves(row, col, board_size) #按照 马走日，返回下一步可能位置
            for row2, col2 in new_positions:
                other_node_id = pos_to_node_id(row2, col2, board_size) #下一步的格子ID
                kt_graph.add_edge(node_id, other_node_id) #在骑士周游图中为两个格子加一条边
    return kt_graph

def pos_to_node_id(x, y, bdSize):
    return x * bdSize + y

def gen_legal_moves(row, col, board_size):
    new_moves = []
    move_offsets = [                        # 马走日的8种走法
        (-1, -2),  # left-down-down
        (-1, 2),  # left-up-up
        (-2, -1),  # left-left-down
        (-2, 1),  # left-left-up
        (1, -2),  # right-down-down
        (1, 2),  # right-up-up
        (2, -1),  # right-right-down
        (2, 1),  # right-right-up
    ]
    for r_off, c_off in move_offsets:
        if (                                # #检查，不能走出棋盘
            0 <= row + r_off < board_size
            and 0 <= col + c_off < board_size
        ):
            new_moves.append((row + r_off, col + c_off))
    return new_moves

# def legal_coord(row, col, board_size):
#     return 0 <= row < board_size and 0 <= col < board_size


def knight_tour(n, path, u, limit):
    u.color = "gray"
    path.append(u)              #当前顶点涂色并加入路径
    if n < limit:
        neighbors = ordered_by_avail(u) #对所有的合法移动依次深入
        #neighbors = sorted(list(u.get_neighbors()))
        i = 0

        for nbr in neighbors:
            if nbr.color == "white" and \
                knight_tour(n + 1, path, nbr, limit):   #选择“白色”未经深入的点，层次
加一，递归深入
                return True
        else:                           #所有的“下一步”都试了走不通
            path.pop()              #回溯，从路径中删除当前顶点
            u.color = "white"       #当前顶点改回白色
            return False
    else:
        return True

def ordered_by_avail(n):
    res_list = []
    for v in n.get_neighbors():
        if v.color == "white":
            c = 0
            for w in v.get_neighbors():
                if w.color == "white":
                    c += 1
            res_list.append((c,v))
    res_list.sort(key = lambda x: x[0])
    return [y[1] for y in res_list]

# class DFSGraph(Graph):
#     def __init__(self):
#         super().__init__()
#         self.time = 0                   #不是物理世界，而是算法执行步数
#
#     def dfs(self):
#         for vertex in self:
#             vertex.color = "white"      #颜色初始化
#             vertex.previous = -1
#         for vertex in self:             #从每个顶点开始遍历
#             if vertex.color == "white":
#                 self.dfs_visit(vertex)  #第一次运行后还有未包括的顶点
#                                         #  则建立森林
#
#     def dfs_visit(self, start_vertex):
#         start_vertex.color = "gray"
#         self.time = self.time + 1       #记录算法的步骤
#         start_vertex.discovery_time = self.time
#         for next_vertex in start_vertex.get_neighbors():
#             if next_vertex.color == "white":
#                 next_vertex.previous = start_vertex
#                 self.dfs_visit(next_vertex)     #深度优先递归访问
#         start_vertex.color = "black"
#         self.time = self.time + 1
#         start_vertex.closing_time = self.time


def main():
    def NodeToPos(id):
       return ((id//8, id%8))

    bdSize = int(input())  # 棋盘大小
    *start_pos, = map(int, input().split())  # 起始位置
    g = knight_graph(bdSize)
    start_vertex = g.get_vertex(pos_to_node_id(start_pos[0], start_pos[1],
bdSize))
    if start_vertex is None:
        print("fail")
        exit(0)

    tour_path = []
    done = knight_tour(0, tour_path, start_vertex, bdSize * bdSize-1)
    if done:
        print("success")
    else:
        print("fail")

    exit(0)

    # 打印路径
    cnt = 0
    for vertex in tour_path:
        cnt += 1
        if cnt % bdSize == 0:
            print()
        else:
            print(vertex.key, end=" ")
            #print(NodeToPos(vertex.key), end=" ")   # 打印坐标

if __name__ == '__main__':
    main()
```

代码运行截图 ==（AC代码截图，至少包含有"Accepted"）==



#44769875提交状态

状态: Accepted

源代码

```
import sys

class Graph:
    def __init__(self):
        self.vertices = {}
        self.num_vertices = 0

    def add_vertex(self, key):
        self.num_vertices = self.num_vertices + 1
        new_ertex = Vertex(key)
        self.vertices[key] = new_ertex
        return new_ertex

    def get_vertex(self, n):
        if n in self.vertices:
            return self.vertices[n]
        else:
            return None

    def __len__(self):
        return self.num_vertices

    def __contains__(self, n):
```

基本信息

#:	44769875
题目:	28050
提交人:	23n2300011436
内存:	4104kB
时间:	30ms
语言:	Python3
提交时间:	2024-04-23 23:14:55

2. 学习总结和收获

学习了树和图的遍历