Assignment #C: 图 (2/4)

Updated 2329 GMT+8 Nov 24, 2025

2025 fall, Complied by 林奕妃、环境科学与工程学院

> **说明：**
>
> 1. **解题与记录：**
>
>   对于每一个题目，请提供其解题思路（可选），并附上使用Python或C++编写的源代码（确保已在OpenJudge， Codeforces，LeetCode等平台上获得Accepted）。请将这些信息连同显示“Accepted”的截图一起填写到下方的作业模板中。（推荐使用Typora https://typoraio.cn 进行编辑，当然你也可以选择Word。）无论题目是否已通过，请标明每个题目大致花费的时间。
>
> 2. **提交安排：**提交时，请首先上传PDF格式的文件，并将.md或.doc格式的文件作为附件上传至右侧的“作业评论”区。确保你的Canvas账户有一个清晰可见的本人头像，提交的文件为PDF格式，并且“作业评论”区包含上传的.md或.doc附件。
>
> 3. **延迟提交：**如果你预计无法在截止日期前提交作业，请提前告知具体原因。这有助于我们了解情况并可能为你提供适当的延期或其他帮助。
>
> 请按照上述指导认真准备和提交作业，以保证顺利完成课程要求。

1. 题目

M909.蛇梯棋（40分钟）

bfs, https://leetcode.cn/problems/snakes-and-ladders/

思路：与之前迷宫模型不同的是本次试验只有确定的位置有传送通道，且步数为1-6。棋盘编号是从左下角开始呈之字形交替上升的，所以要根据当前编号计算出对应的矩阵行和列索引。在搜索过程中，我们维护一个队列存储当前位置和步数，并使用一个集合记录已访问过的目的地以避免重复。每次从队列取出当前方格时，模拟掷骰子走出 1 到 6 步，对于每一个潜在的落脚点，先检查其在棋盘对应坐标上是否有蛇或梯子：如果有，则必须强制跳转到指定值；如果没有，则停留在该点。只要这个最终落脚点没有被访问过，就将其加入队列，一旦遇到编号等于总格数的终点，立即返回当前累计的步数，若遍历结束仍未到达则返回 -1。

代码：

```
from collections import deque

class Solution:
    def snakesAndLadders(self, board: list[list[int]]) -> int:
        n = len(board)
        target = n * n

        # 坐标转换函数：将 1~n^2 的编号转换为矩阵的 (r, c)
```

```python
        def get_coordinate(num):
            # 将编号转为 0-indexed
            idx = num - 1
            # 计算从底部数上来是第几行 (0, 1, 2...)
            row_from_bottom = idx // n

            # 实际矩阵的行索引
            r = n - 1 - row_from_bottom

            # 实际矩阵的列索引
            # 如果是偶数行（0, 2...），方向是从左到右
            if row_from_bottom % 2 == 0:
                c = idx % n
            # 如果是奇数行（1, 3...），方向是从右到左
            else:
                c = n - 1 - (idx % n)
            return r, c

        # BFS 初始化
        # 队列存储元素为 (当前位置编号, 当前步数)
        queue = deque([(1, 0)])
        # 记录访问过的位置，避免重复搜索
        visited = {1}

        while queue:
            curr, step = queue.popleft()

            # 尝试掷骰子 1 到 6
            for i in range(1, 7):
                next_val = curr + i

                # 如果超出了棋盘范围，停止后续更大的点数
                if next_val > target:
                    break

                # 获取该位置在矩阵中的坐标
                r, c = get_coordinate(next_val)

                # 确定实际目的地
                # 如果该位置有蛇或梯子，必须跳跃
                if board[r][c] != -1:
                    destination = board[r][c]
                else:
                    destination = next_val

                # 如果到达终点
                if destination == target:
                    return step + 1

                # 如果该目的地没有被访问过，加入队列
                if destination not in visited:
                    visited.add(destination)
                    queue.append((destination, step + 1))

        # 如果队列空了还没到达终点，说明无法到达
        return -1
```

代码运行截图（至少包含有"Accepted"）

sy382: 有向图判环 中等（40分钟）

dfs, topological sort, https://sunnywhy.com/sfbj/10/3/382

思路：采用三色标记法来区分节点的状态。我们需要定义未访问、正在访问和已完成这三种状态，其中正在访问代表节点目前处于当前的递归栈中。算法遍历图中每一个未访问的节点并发起递归搜索，在进入节点时将其标记为正在访问状态，随后检查其所有指向的邻居节点。如果遇到一个状态为正在访问的邻居，说明我们回到了当前路径上的祖先节点，这就构成了环，应立即返回存在环的结论。如果邻居未访问则继续递归，若邻居已完成则跳过。当一个节点的所有邻居都处理完毕后将其标记为已完成状态。若遍历完所有节点都没有发现指向正在访问节点的边，则说明图中不存在环。

代码：

```
import sys
def solve():
    # 使用快速 I/O 读取数据
    input = sys.stdin.read
    data = input().split()

    if not data:
        return

    iterator = iter(data)
    try:
        n = int(next(iterator))
        m = int(next(iterator))
    except StopIteration:
        return

    # 构建邻接表
    adj = [[] for _ in range(n)]
    for _ in range(m):
```

```
        u = int(next(iterator))
        v = int(next(iterator))
        adj[u].append(v)

    # 状态数组: 0=未访问, 1=正在访问(递归栈中), 2=已完成
    state = [0] * n

    def dfs(u):
        state[u] = 1  # 标记为：正在访问

        for v in adj[u]:
            if state[v] == 1:
                # 如果邻居正在访问中，说明遇到了回边，存在环
                return True
            if state[v] == 0:
                # 如果邻居未访问，递归访问
                if dfs(v):
                    return True
            # 如果 state[v] == 2，说明是已完成节点，无需操作

        state[u] = 2  # 标记为：已完成
        return False

    # 图可能不是连通的，需要遍历所有节点
    for i in range(n):
        if state[i] == 0:
            if dfs(i):
                print("Yes")
                return

    print("No")

if __name__ == "__main__":
    solve()
```

代码运行截图 （至少包含有"Accepted"）

M28046: 词梯（40分钟）

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

思路：最短路径采用广度优先搜索。关于解的唯一性，由于 BFS 是严格按照距离起点的步数由近及远遍历的，它保证了第一次到达目标单词时经过的步数最少。

代码：

```
import sys
from collections import deque, defaultdict

def solve():
    # 1. 读取输入
    # 使用 sys.stdin.read 一次性读取，处理多行输入更方便
    input_data = sys.stdin.read().split()

    if not input_data:
        return

    iterator = iter(input_data)
    try:
        n = int(next(iterator))
        words = []
        for _ in range(n):
            words.append(next(iterator))
        start_word = next(iterator)
        end_word = next(iterator)
    except StopIteration:
        return

    # 2. 预处理：构建通配符字典
    # 键为带 * 的模式 (如 "*ool")，值为匹配该模式的单词列表
    combo_dict = defaultdict(list)
    for word in words:
        for i in range(4):
            # 生成 4 个通配模式
            pattern = word[:i] + "*" + word[i+1:]
            combo_dict[pattern].append(word)

    # 3. BFS 初始化
    queue = deque([start_word])
    # visited 用于记录路径：key=当前单词，value=前驱单词
    # 起始单词的前驱为 None
    visited = {start_word: None}

    found = False

    while queue:
        current_word = queue.popleft()

        # 如果找到目标单词
        if current_word == end_word:
            found = True
```

```
                break

            # 寻找邻居
            for i in range(4):
                pattern = current_word[:i] + "*" + current_word[i+1:]

                # 遍历该模式下的所有单词
                for neighbor in combo_dict[pattern]:
                    if neighbor not in visited:
                        visited[neighbor] = current_word
                        queue.append(neighbor)

                # 优化：该模式已经检查过了，之后即使别的单词生成同样的模式，
                # 那些邻居要么已经在队列里，要么已经访问过。
                # 为了防止重复检查，可以清空该模式的列表（可选优化）
                # combo_dict[pattern] = []

    # 4. 输出结果
    if found:
        path = []
        curr = end_word
        while curr is not None:
            path.append(curr)
            curr = visited[curr]
        # 因为是回溯，所以需要反转路径
        print(*(path[::-1]))
    else:
        print("NO")

if __name__ == "__main__":
    solve()
```

代码运行截图 （至少包含有"Accepted"）

#51069793提交状态

查看 提交 统计 提问

状态：Accepted

源代码

```
import sys
from collections import deque, defaultdict

def solve():
    # 1. 读取输入
    # 使用 sys.stdin.read 一次性读取，处理多行输入更方便
    input_data = sys.stdin.read().split()
```

基本信息

#: 51069793
题目: 28046
提交人: 25n2200013554
内存: 5648kB
时间: 40ms
语言: Python3
提交时间: 2025-11-30 14:08:25

M433.最小基因变化（30分钟）

bfs, https://leetcode.cn/problems/minimum-genetic-mutation/description/

思路：把每一个基因序列看作图中的一个节点，如果两个基因序列之间只相差一个字符，并且目标序列在基因库 bank 中，那么这两个节点之间就有一条边。和上一题类似，首先将基因库转换为集合以实现 O(1) 的快速查找，然后从起始基因开始，利用队列逐层向外扩展。在每一层搜索中，我们枚举当前基因序列所有可能的单字符突变（即尝试改变 8 个位置中的每一个字符为 A、C、G 或 T），如果生成的突变

序列存在于基因库中且未被访问过，就将其加入队列继续搜索。只要按照 BFS 的层级顺序遍历，第一次遇到目标基因时所经过的步数一定是最少的，若遍历结束仍未找到则返回 -1。

代码

```python
from collections import deque

class Solution:
    def minMutation(self, start: str, end: str, bank: list[str]) -> int:
        # 将基因库转换为集合，实现 O(1) 的快速查找
        bank_set = set(bank)

        # 如果目标基因不在库中，直接返回 -1
        if end not in bank_set:
            return -1

        # BFS 初始化：队列存储 (当前基因, 当前步数)
        queue = deque([(start, 0)])
        # 记录已访问的基因，防止走回头路
        visited = {start}

        # 基因可能的四个字符
        options = ['A', 'C', 'G', 'T']

        while queue:
            curr, step = queue.popleft()

            # 找到目标基因，返回步数
            if curr == end:
                return step

            # 尝试改变每一个位置的字符
            for i in range(8):
                original_char = curr[i]
                for char in options:
                    # 跳过自身
                    if char == original_char:
                        continue

                    # 拼接生成新基因
                    new_gene = curr[:i] + char + curr[i+1:]

                    # 检查新基因是否有效（在bank中）且未被访问过
                    if new_gene in bank_set and new_gene not in visited:
                        visited.add(new_gene)
                        queue.append((new_gene, step + 1))

        # 队列排空仍未找到目标，返回 -1
        return -1
```

代码运行截图（至少包含有"Accepted"）

M05443: 兔子与樱花（30分钟）

Dijkstra, http://cs101.openjudge.cn/practice/05443/

思路：本题即为求解无向图的加权问题。利用哈希表构建邻接表，将字符串形式的地点名称映射为节点，并存储对应的边权。对于每一组起点和终点的查询，采用 Dijkstra 算法配合优先队列来寻找最短路径。

代码

```
import heapq
import sys

def solve():
    # 使用 sys.stdin.read 一次性读取所有数据，处理多行输入更方便
    input_data = sys.stdin.read().split()

    if not input_data:
        return

    iterator = iter(input_data)

    try:
        # 1. 读取地点信息
        # 虽然可以直接从边信息建立图，但按题目逻辑先读取 P
        P = int(next(iterator))
        locations = set()
        for _ in range(P):
            locations.add(next(iterator))

        # 2. 读取边信息构建图
        Q = int(next(iterator))
        adj = {loc: [] for loc in locations}

        for _ in range(Q):
            u = next(iterator)
            v = next(iterator)
```

```
            w = int(next(iterator))
            # 建立无向图
            if u not in adj: adj[u] = []
            if v not in adj: adj[v] = []
            adj[u].append((v, w))
            adj[v].append((u, w))

        # 3. 处理查询
        R = int(next(iterator))
        for _ in range(R):
            start = next(iterator)
            end = next(iterator)

            # 特殊情况：起点等于终点
            if start == end:
                print(start)
                continue

            # Dijkstra 算法初始化
            # 优先队列: (当前累计距离, 当前节点)
            pq = [(0, start)]
            # 记录最短距离
            dists = {loc: float('inf') for loc in locations}
            dists[start] = 0
            # 记录路径: node -> (前驱节点, 边权)
            parent = {}

            found = False

            while pq:
                d, u = heapq.heappop(pq)

                # 如果当前距离已经大于已知最短距离，跳过
                if d > dists.get(u, float('inf')):
                    continue

                # 找到终点
                if u == end:
                    found = True
                    break

                # 遍历邻居
                if u in adj:
                    for v, weight in adj[u]:
                        if dists[u] + weight < dists.get(v, float('inf')):
                            dists[v] = dists[u] + weight
                            parent[v] = (u, weight)
                            heapq.heappush(pq, (dists[v], v))

            # 路径还原
            if found:
                path_segments = []
                curr = end
                while curr != start:
                    prev_node, cost = parent[curr]
                    # 按照题目要求的格式构造字符串片段
```

```
                    # 比如 "->(80)->Sensouji"
                    segment = f"->({cost})->{curr}"
                    path_segments.append(segment)
                    curr = prev_node

                # 加上起点
                path_segments.append(start)
                # 因为是回溯，所以需要反转列表
                print("".join(path_segments[::-1]))
            else:
                # 题目暗示图是连通的，一般不会走到这里
                pass

    except StopIteration:
        pass

if __name__ == "__main__":
    solve()
```

代码运行截图（至少包含有"Accepted"）

#51071236提交状态

查看 提交 统计 提问

状态: Accepted

源代码

```
import heapq
import sys

def solve():
    # 使用 sys.stdin.read 一次性读取所有数据，处理多行输入更方便
    input_data = sys.stdin.read().split()
```

基本信息

#: 51071236
题目: 05443
提交人: 25n2200013554
内存: 3764kB
时间: 29ms
语言: Python3
提交时间: 2025-11-30 15:14:15

M28050: 骑士周游（40分钟）

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

思路：为了解决普通 DFS 在大棋盘上效率极低的问题，引入Warnsdorff 启发式规则：在选择下一步移动时，并不随机选择，而是计算每个候选位置的“度数”（即从该候选位置出发有多少个合法的下一步），并优先跳转到度数最小的位置。这种贪心策略能让骑士尽早访问那些容易变成死胡同的边缘或角落格子，极大地降低了回溯的概率，从而能在 O(N)级别的时间内找到解，输出 "success"，否则输出 "fail"。

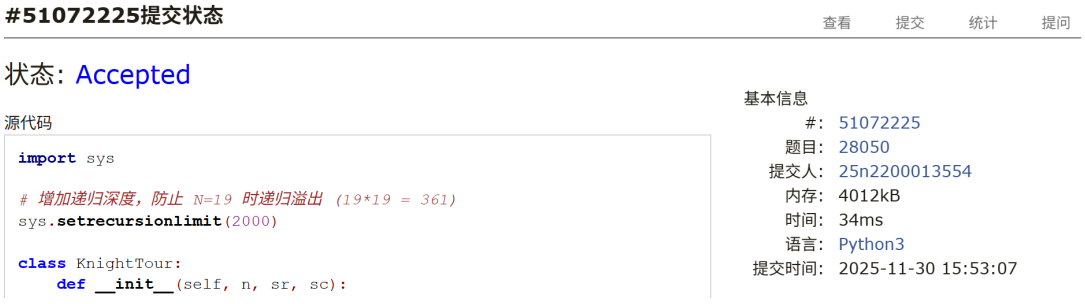
代码:

```
import sys

# 增加递归深度，防止 N=19 时递归溢出 (19*19 = 361)
sys.setrecursionlimit(2000)

class KnightTour:
    def __init__(self, n, sr, sc):
        self.n = n
        self.sr = sr
        self.sc = sc
```

```python
        # 棋盘标记，False 表示未访问
        self.visited = [[False] * n for _ in range(n)]
        # 马走的 8 个方向
        self.moves = [
            (1, 2), (1, -2), (-1, 2), (-1, -2),
            (2, 1), (2, -1), (-2, 1), (-2, -1)
        ]
        self.total_squares = n * n

    def is_valid(self, r, c):
        return 0 <= r < self.n and 0 <= c < self.n and not self.visited[r][c]

    def get_degree(self, r, c):
        """
        计算 (r, c) 位置下一步有多少个合法的走法
        用于 Warnsdorff 规则排序
        """
        count = 0
        for dr, dc in self.moves:
            nr, nc = r + dr, c + dc
            if self.is_valid(nr, nc):
                count += 1
        return count

    def dfs(self, r, c, count):
        # 标记当前点
        self.visited[r][c] = True

        # 成功条件：步数达到总格子数
        if count == self.total_squares:
            return True

        # 生成所有合法的下一步
        next_moves = []
        for dr, dc in self.moves:
            nr, nc = r + dr, c + dc
            if self.is_valid(nr, nc):
                next_moves.append((nr, nc))

        # 【核心优化】Warnsdorff 规则：
        # 对下一步的候选节点进行排序，优先选择“度数”（后续可行步数）最少的节点
        # 这大大减少了回溯的次数
        next_moves.sort(key=lambda x: self.get_degree(x[0], x[1]))

        # 遍历排序后的候选节点
        for nr, nc in next_moves:
            if self.dfs(nr, nc, count + 1):
                return True

        # 回溯：取消标记
        self.visited[r][c] = False
        return False

    def solve(self):
        if self.dfs(self.sr, self.sc, 1):
            print("success")
```

```
        else:
            print("fail")

def main():
    # 读取输入
    try:
        input_data = sys.stdin.read().split()
        if not input_data:
            return

        n = int(input_data[0])
        sr = int(input_data[1])
        sc = int(input_data[2])

        tour = KnightTour(n, sr, sc)
        tour.solve()

    except Exception:
        pass

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

代码运行截图 （至少包含有"Accepted"）



#51072225提交状态

查看 提交 统计 提问

状态: Accepted

源代码

```
import sys

# 增加递归深度，防止 N=19 时递归溢出 (19*19 = 361)
sys.setrecursionlimit(2000)

class KnightTour:
    def __init__(self, n, sr, sc):
```

基本信息

#: 51072225
题目: 28050
提交人: 25n2200013554
内存: 4012kB
时间: 34ms
语言: Python3
提交时间: 2025-11-30 15:53:07

2. 学习总结和个人收获

这是我第一次做晴问的题目，顺便练习了一下题集里面图的遍历的其他题目。求解无向图连通块个数可以使用深度优先搜索配合访问标记数组来实现。判断无向图是否连通的核心在于验证从图中的任意一个节点出发是否能够到达所有其他节点。求解无向图连通块的最大权值问题可以采用深度优先搜索算法来遍历图中的每一个连通分量。求解无向图中各顶点的层号即最短路径长度通常使用广度优先搜索算法来实现。求解受限层号的顶点数问题可以采用广度优先搜索算法结合层级记录来实现。做了这个题集之后再去完成后面的作业变得熟练很多。