作业6 杨镕争 2021K8009929022

T1 最基础的寻找最大流的模型，由于图比较简单，选用什么算法都可以，这里选用了Edmond-Karp算法，每次通过BFS优先选择从源点到汇点的边最少的增广路径。在最坏的情况下，每次找到的增广路径长度为O(V+E)，需要找O(VE)次，算法复杂度为$O(VE^2)$ 伪代码：

```
function Edmonds_Karp(Graph, source, sink):
        residualGraph = copy(Graph)
        maxFlow = 0
        while True:
            path, capacity = findAugmentingPath(residualGraph, source, sink)
            if not path:
                break
            updateResidualGraph(residualGraph, path, capacity)
            maxFlow += capacity
        return maxFlow
function findAugmentingPath(graph, source, sink):
        visited = set()
        queue = [(source, [source], float('inf'))]
        while queue:
            current, path, capacity = queue.pop(0)
            visited.add(current)
            for neighbor, residualCapacity in graph[current]:
                if neighbor not in visited and residualCapacity > 0:
                    if neighbor == sink:
                        return path + [neighbor], min(capacity, residualCapacity)
                    queue.append((neighbor, path + [neighbor], min(capacity,
residualCapacity)))
                    visited.add(neighbor)

        return None, 0
function updateResidualGraph(residualGraph, path, capacity):
        for i in range(len(path) - 1):
            u, v = path[i], path[i + 1]
            residualGraph[u][v] -= capacity
            residualGraph[v][u] += capacity

graph = {
            1: {1: 30},
        2: {1: 30, 3: 20},
        3: {},
        4: {2: 30,3: 20}
}
source = 4
sink = 3
maxFlow = Edmonds_Karp(graph, source, sink)
print("Maximum Flow:", maxFlow)
```

T2 分批次转移学生，为了更少的批次，每次要转移尽可能多的学生，因此要找到图的最大流。算法采用Edmonds-Karp算法，时间复杂度为$O(VE^2)$

```
from collections import deque
def max_students_evacuated(graph, source, sink):
        batches = 0
        max_evacuated = 0

        while True:
            visited = set()
            queue = deque([(source, float('inf'))])
            while queue:
                current, min_capacity = queue.popleft()
                if current == sink:
                    max_evacuated += min_capacity
                    break
                visited.add(current)
                for neighbor, capacity in graph[current]:
                    if neighbor not in visited and capacity > 0:
                        min_capacity = min(min_capacity, capacity)
                        queue.append((neighbor, min_capacity))
            if min_capacity == float('inf'):
                break
            batches += 1
            update_graph(graph, source, sink, min_capacity)
        return max_evacuated, batches

def update_graph(graph, source, sink, min_capacity):
        current = source
        while current != sink:
            neighbor, capacity = graph[current].pop(0)
            reverse_edge = next((edge for edge in graph[neighbor] if edge[0] ==
current), None)

            graph[current].append((neighbor, capacity - min_capacity))

            if reverse_edge:
                graph[neighbor].remove(reverse_edge)
                graph[neighbor].append((current, reverse_edge[1] + min_capacity))

            current = neighbor

graph = {
            'A': [('B', 1), ('D', 2)],
        'B': [('C', 1)],
        'C': [('F', 1)],
        'D': [('C', 1), (‘E’, 1)],
        'E': [('F', 1)],
        'F': []
}
source = 'A'
sink = 'F'
max_evacuated, batches = max_students_evacuated(graph, source, sink)
print(f"每次运送最多学生: {max_evacuated}")
print(f"运送次数: {batches}")
```

T3 由于只有三个人，因此这里可以讨论一下：如果三个人中有两个或三个人是来访人员，不可能安排的下；如果没有来访人员，一定能安排下，不回家的住在自己床上就行。因此只讨论有一个来访人员的情况。对于来访人员A，查看他的关系表，对于他认识的人B，如果B回家了，他住在B的床上即可；否则，查询B的关系表，查询除了A之外B认识的其他人，如果查询到了其他人C，检查C是否回家，如果回家，B住C床上，A住B床上，解决，不回家的话就无法安排开；如果查询不到其他人，返回false。在这种情况下，时间复杂度是常数级的，因为只有三个人。 对于更多人的情况，可以这样建模：如果x认识y，在x和y之间建立联系，如果x回家，y到x方向的边流量为1，y回家同理。对于所有来访人员，视为流的source，而回家的学生作为sink，这是一个多源多汇的问题，实际解决上可能会有更多的问题，。伪代码只给出三人的情况。 伪代码：

```
def can_accommodate_one_visitor(students, home_status, knows):
        num_students = len(students)
        visitors = [i for i in range(num_students) if students[i] == 0]
        if not visitors:
            return True
        if len(visitors) == 1:
            visitor = visitors[0]
            for friend in range(num_students):
                if knows[visitor][friend] == 1:
                    if home_status[friend] == 1:
                        return True
                    else:
                        for other_friend in range(num_students):
                        if knows[friend][other_friend] == 1 and other_friend !=
visitor:
                            if home_status[other_friend] == 1:
                                return True
        return False
students = [1, 1, 0]
home_status = [0, 1, 0]
knows = [
            [0, 1, 1],
        [1, 0, 0],
        [1, 0, 0]
]
result = can_accommodate_one_visitor(students, home_status, knows)
if result:
        print("Available")
else:
        print("Unavailable")
```

T4 最大匹配问题，将外国飞行员和英国飞行员作为二分图的两个集合，利用匈牙利算法即可求解。对每个店都要进行一次DFS算法，DFS算法在邻接矩阵的情况下复杂度为$O(V^2)$，总时间复杂度为$O(V^3)$ 伪代码：

```
def hungarian_algorithm(graph):
        matching = [-1] * len(graph)
        for u in range(len(graph)):
            visited = [False] * len(graph)
            dfs(u, graph, matching, visited)
```

```
        return matching
def dfs(u, graph, matching, visited):
        for v in range(len(graph[u])):
            if graph[u][v] and not visited[v]:
                visited[v] = True
                if matching[v] == -1 or dfs(matching[v], graph, matching,
visited):
                    matching[v] = u
                    return True
        return False
def find_best_pairing_scheme(edges):
        num_foreign_pilots = 5
        num_british_pilots = 5
        num_pilots = num_foreign_pilots + num_british_pilots
        graph = [[0] * num_pilots for _ in range(num_pilots)]
        for u, v in edges:
            graph[u][v] = 1
        matching = hungarian_algorithm(graph)
        max_aircraft_dispatched = sum(1 for pilot in matching if pilot != -1)
        return max_aircraft_dispatched, matching
edges = [(1, 7), (1, 8), (2, 6), (2, 9), (2, 0), (3, 7), (3, 8),(4, 7), (4, 8),
(5, 0)]
max_aircraft, pairing_scheme = find_best_pairing_scheme(edges)

print(f"max_plane: {max_aircraft}")
for i, j in enumerate(pairing_scheme):
        if j != -1:
            print(f"Foreign Pilot {i} with British Pilot {j}")
```

T5 假设矩阵的大小为m×n，构建这样一个图：一共有m+n+2条边，其中节点0为源点，节点m+n+1为汇点，节点1,2,3...m代表矩阵的每一行，m+1,m+2...m+n代表矩阵的每一列，图的边流量和方向如下：源点流向行节点，行节点流向列节点，列节点流向汇点。其中源点和行节点之间的流量为矩阵该行之和，共m条边；列节点和汇点之间的流量为矩阵该列之和，共n条边；行节点和列节点之间的流量为1，每一个行和列节点之间都有边，共m×n条边。接着，运行最大流算法，如果可以找到一个符合题意的矩阵，那么从源点到行节点的流量和从列节点到汇点的流量一定相等且都为满。行列节点直接的边实际流量为1代表矩阵的这个位置为1.如果找不到这样的矩阵，从源点到行节点的流量和列节点到汇点的流量一定有至少一方不满。

这里也尝试给出一个贪心算法解决这个问题：对于给定的矩阵形状(m×n)与每一行每一列的和(row_num,column_num)，更新权重矩阵，(i,j)点的权重等于row_num[i]+column_num[j].将矩阵中每个点按照权重大小降序排列，按照顺序进行以下判断：如果这个点被访问过，访问下一个点；如果没被访问过，查看此时这个点所在位置的行与列的和的剩余值，如果均有剩余，将这个点置为1，所在行与列的和减1，标记为已访问，继续访问下一个点。矩阵初始化为全0.在进行操作之前，先检查一下输入是否合法：首先检查行/列的和有没有超过列/行数，其次检查row_num的和与column_num的和是否相等