江苏大学

JIANGSU UNIVERSITY

《网络科学基础》

第四次平时作业

学院名称： 计算机学院

专业班级： 物联网 2303 班

学生姓名： 邱佳亮

学生学号： 3230611072

教师姓名： 熊书明

2024 年 11 月

1. 题一代码

```python
import numpy as np #导包
import matplotlib.pyplot as plt
import heapq
#%%
def dijkstra_path(G, start, end):
    # 初始化距离和前驱节点字典
    distances = {node: float('inf') for node in G}
    previous_nodes = {node: None for node in G}
    distances[start] = 0

    # 优先队列，存储（距离， 节点）对
    priority_queue = [(0, start)]

    while priority_queue:
        # 取出当前距离最小的节点
        current_distance, current_node = heapq.heappop(priority_queue)

        # 如果到达终点，提前结束
        if current_node == end:
            break

        # 遍历当前节点的所有邻居
        for neighbor, weight in G[current_node].items():
            distance = current_distance + weight

            # 如果找到更短的路径
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                previous_nodes[neighbor] = current_node
                heapq.heappush(priority_queue, (distance, neighbor))

    # 构建最短路径
    path = []
    current_node = end
    while previous_nodes[current_node] is not None:
        path.insert(0, current_node)
        current_node = previous_nodes[current_node]
    if path:
        path.insert(0, start)

    return distances[end], path
```

```python
# 示例图
G = {
    1: {2: 7, 3: 9, 6: 14},
    2: {1: 7, 3: 10, 4: 15},
    3: {1: 9, 2: 10, 4: 11, 6: 2},
    4: {2: 15, 3: 11, 5: 6},
    5: {4: 6, 6: 9},
    6: {1: 14, 3: 2, 5: 9}
}

# 调用函数
distance, path = dijkstra_path(G, 1, 5)
print(f"最短距离: {distance}")
print(f"最短路径: {path}")
#%%
def bellman_ford(G, start):
    # 初始化距离和前驱节点字典
    distances = {node: float('inf') for node in G}
    previous_nodes = {node: None for node in G}
    distances[start] = 0

    # 获取所有边
    edges = []
    for node in G:
        for neighbor, weight in G[node]:
            edges.append((node, neighbor, weight))

    # 进行 V-1 次松弛操作
    for _ in range(len(G) - 1):
        for u, v, weight in edges:
            if distances[u] + weight < distances[v]:
                distances[v] = distances[u] + weight
                previous_nodes[v] = u

    # 检查负权环
    for u, v, weight in edges:
        if distances[u] + weight < distances[v]:
            raise ValueError("图中存在负权环")

    return distances, previous_nodes

def reconstruct_path(previous_nodes, start, end):
    path = []
    current_node = end
```

```python
    while current_node is not None:
        path.insert(0, current_node)
        current_node = previous_nodes[current_node]
    return path

# 示例图
G = {
    1: [(2, 7), (3, 9), (6, 14)],
    2: [(1, 7), (3, 10), (4, 15)],
    3: [(1, 9), (2, 10), (4, 11), (6, 2)],
    4: [(2, 15), (3, 11), (5, 6)],
    5: [(4, 6), (6, 9)],
    6: [(1, 14), (3, 2), (5, 9)]
}

# 调用函数
distances, previous_nodes = bellman_ford(G, 1)
print(f"最短距离: {distances}")
print(f"前驱节点: {previous_nodes}")

# 重建从起点到终点的最短路径
end = 5
path = reconstruct_path(previous_nodes, 1, end)
print(f"从节点 1 到节点 {end} 的最短路径: {path}")#Bellman-Ford 算法
```

1.1. 运行结果

```
最短距离: 20
最短路径: [1, 3, 6, 5]
```

图 1 Dijkstra 算法运行结果

```
最短距离: {1: 0, 2: 7, 3: 9, 4: 20, 5: 20, 6: 11}
前驱节点: {1: None, 2: 1, 3: 1, 4: 3, 5: 6, 6: 3}
从节点 1 到节点 5 的最短路径: [1, 3, 6, 5]
```

图 2 Bellman-Ford 算法运行结果

1.2. 算法比较

Dijkstra 算法和 Bellman-Ford 算法都是用于求解图中单源最短路径问题的算法，它们都采用贪心策略来逼近最短路径。Dijkstra 算法适用于边权全为非负的

图，时间复杂度为 O((V+E)logV)，实现相对复杂，不能处理负权边，也无法检测负权环。而 Bellman-Ford 算法可以处理含有负权边的图，能检测负权环，时间复杂度为 O(VE)，实现简单，适用于可能包含负权边的图。所以如果图中含有负权边或需要检测负权环，Bellman-Ford 算法效率更高；而当图的边权全为非负且图较为密集时，Dijkstra 算法的效率更高。

2. 题二代码

```
clc,clear;
a=zeros(6)
a(1,[2,3,5])=1;
a(2,[3,4])=1;
a(3,6)=1;
a(4,6)=1;
a=a+a'
[D,L,dist]=myAPL(a)
function[D,L,dist] =myAPL(a)
     A=graph(a)
     dist=distances(A);
     D=max(max(dist));
     Ldist=tril(dist);
     he=sum(nonzeros(Ldist));
     n=length(a);
     L=he/nchoosek(n,2);
end
```

2.1. 运行结果

```
clc,clear;
a=zeros(6)
```

```
a = 6×6
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
```

```
a(1,[2,3,5])=1;
a(2,[3,4])=1;
a(3,6)=1;
a(4,6)=1;
a=a+a'
```

```
a = 6×6
     0     1     1     0     1     0
     1     0     1     1     0     0
     1     1     0     0     0     1
     0     1     0     0     0     1
     1     0     0     0     0     0
     0     0     1     1     0     0
```

```
[D,L,dist]=myAPL(a)
```

```
A =
  graph - 属性:

    Edges: [7×2 table]
    Nodes: [6×0 table]
D = 3
L = 1.6667
dist = 6×6
     0     1     1     2     1     2
     1     0     1     1     2     2
     1     1     0     2     2     1
     2     1     2     0     3     1
     1     2     2     3     0     3
     2     2     1     1     3     0
```

```
function[D,L,dist] =myAPL(a)
    A=graph(a)
    dist=distances(A);
    D=max(max(dist));
    Ldist=tril(dist);
    he=sum(nonzeros(Ldist));
    n=length(a);
    L=he/nchoosek(n,2);
end
```

图 3　运行结果

2.2. 查找帮助文档

2.2.1. tril()

函数功能：返回矩阵 A 的下三角部分

实例：B 提取了 A 的下三角部分，C 仅提取了 A 的主对角线下的部分：

```
A = ones(4)
```

```
A = 4×4
     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1
```

```
B = tril(A)
```

```
B = 4×4
     1     0     0     0
     1     1     0     0
     1     1     1     0
     1     1     1     1
```

```
C = tril(A,-1)
```

```
C = 4×4
     0     0     0     0
     1     0     0     0
     1     1     0     0
     1     1     1     0
```

图 4　函数示例

2.2.2. sparse()

函数功能：将矩阵转换为稀疏矩阵以节省内存；生成 m×n 的全零稀疏矩阵。

实例：通过 sparse 将 A 矩阵转换为稀疏矩阵，节约了内存；也可以建立一个 10000*5000 的稀疏矩阵：

```
A = eye(10000);
whos A
```

```
  Name      Size                  Bytes  Class     Attributes

  A         10000x10000       800000000  double
```

```
S = sparse(A);
whos S
```

```
  Name      Size                Bytes  Class     Attributes

  S         10000x10000        240008  double    sparse
```

```
S = sparse(10000,5000)
```

```
S =
   全零稀疏矩阵: 10000×5000
```

图 5　函数示例

2.2.3. graphallshortestpaths()

函数功能：在图中找到所有最短路径。

实例：建立了一个稀疏矩阵 G，并找到了其最短路径；graphallshortestpaths 已在 R2022b 版本被删除，用 distances 代替。

```
G = sparse([6 1 2 2 3 4 4 5 5 6 1],[2 6 3 5 4 1 6 3 4 3 5],[41 99 51 32 15 45 38 32 36 29 21])
graphallshortestpaths(G)
```

2.2.4. max()

函数功能：返回数组的最大元素。

实例：使用 max 函数返回了数组 A 的最大元素 52：

```
A = [23 42 37 18 52];
M = max(A)
```

```
M = 52
```

图 6　函数示例

2.2.5. sum()

函数功能：返回沿数组一维度的元素之和。

实例：使用 sum 函数返回向量或矩阵 A 的元素之和，可以指定沿列方向或行方向计算：

```
A = 1:10;
S = sum(A)
```

```
S = 55
```

```
A = [1 3 2; 4 2 5; 6 1 4];
S = sum(A,2)
```

```
S = 3×1
     6
    11
    11
```

```
A = [1 3 2; 4 2 5; 6 1 4];
S = sum(A)
```

```
S = 1×3
    11     6    11
```

图 7　函数示例

2.2.6. nonzeros()

函数功能：返回矩阵中非零元素的列向量，v 中的元素按列排序。

实例：建立了 10*10 的稀疏矩阵 A，用 nonzeros 函数返回了权不为 0 的边的权重，并按由小到大排序：

```
A = sparse([1 3 2 1],[1 1 2 3],1:4,10,10)
```

```
A =
   (1,1)        1
   (3,1)        2
   (2,2)        3
   (1,3)        4
```

```
v = nonzeros(A)
```

```
v = 4×1
      1
      2
      3
      4
```

图 8　函数示例

2.2.7. nchoosek()

函数功能：返回二项式系数。

实例：计算了二项式系数C_5^4：

```
b = nchoosek(5,4)
```

```
b = 5
```

图 9　函数示例