

20240409~23-Week8~10 图论

Updated 2013 GMT+8 Apr 17, 2024

2024 spring, Complied by Hongfei Yan

Logs:

2024/4/17 重构了目录，主要是把图算法分成了 基本图算法、更多图算法

2024/4/7 打*的章节，可以跳过，可能超纲了。

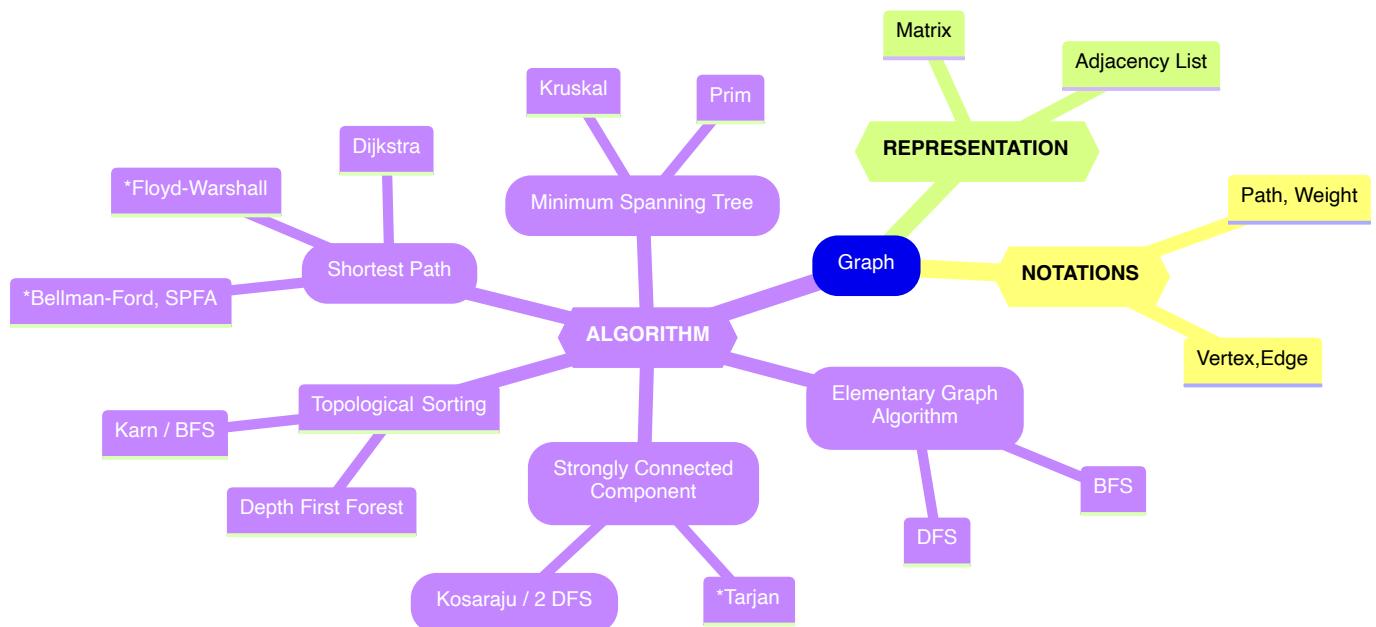
2024/4/5 数算重点是树、图、和算法。图这部分重点是算法，因为图的主要表示方式——邻接表，树也使用。

其中涉及到矩阵存储的图遍历，基本都是计概题目。

这可以是笔试判断题：

计算机存储器线性编址的简单性与程序逻辑的复杂性之间的鸿沟，导致了数据结构的必要性。

取自：李晓明. 为什么会有“数据结构”？计算机教育，2019-01-18



图的知识图谱

一、(Week8) 图的概念、表示方法

图论是数学的一个分支，主要研究图的性质以及图之间的关系。在与数据结构和算法相关的内容中，图论涵盖了以下几个方面：

1. **图的表示**：图可以用不同的数据结构来表示，包括邻接矩阵、邻接表、关联矩阵等。这些表示方法影响着对图进行操作和算法实现的效率。
2. **图的遍历**：图的遍历是指从图中的某个顶点出发，访问图中所有顶点且不重复的过程。常见的图遍历算法包括深度优先搜索（DFS）和广度优先搜索（BFS）。
3. **最短路径**：最短路径算法用于找出两个顶点之间的最短路径，例如 Dijkstra 算法和 Floyd-Warshall 算法。这些算法在网络路由、路径规划等领域有广泛的应用。
4. **最小生成树**：最小生成树算法用于在一个连通加权图中找出一个权值最小的生成树，常见的算法包括 Prim 算法和 Kruskal 算法。最小生成树在网络设计、电力传输等领域有着重要的应用。
5. **图的匹配**：图的匹配是指在一个图中找出一组边，使得没有两条边有一个公共顶点。匹配算法在任务分配、航线规划等问题中有着广泛的应用。
6. **拓扑排序**：拓扑排序算法用于对有向无环图进行排序，使得所有的顶点按照一定的顺序排列，并且保证图中的边的方向符合顺序关系。拓扑排序在任务调度、依赖关系分析等领域有重要的应用。
7. **图的连通性**：图的连通性算法用于判断图中的顶点是否连通，以及找出图中的连通分量。这对于网络分析、社交网络分析等具有重要意义。
8. **图的颜色着色**：图的着色问题是指给图中的顶点赋予不同的颜色，使得相邻的顶点具有不同的颜色。这在调度问题、地图着色等方面有应用。

这些内容是图论在数据结构与算法领域的一些重要内容，它们在计算机科学和工程领域有广泛的应用。

1 术语和定义

图是更通用的结构；事实上，可以把树看作一种特殊的图。图可以用来表示现实世界中很多有意思的事物，包括道路系统、城市之间的航班、互联网的连接，甚至是计算机专业的一系列必修课。图一旦有了很好的表示方法，就可以用一些标准的图算法来解决那些看起来非常困难的问题。

尽管我们能够轻易看懂路线图并理解其中不同地点之间的关系，但是计算机并不具备这样的能力。不过，我们也可以将路线图看成是一张图，从而使计算机帮我们做一些非常有意思的事情。用过互联网地图网站的人都知道，计算机可以帮助我们找到两地之间最短、最快、最便捷的路线。

计算机专业的学生可能会有这样的疑惑：自己需要学习哪些课程才能获得学位呢？图可以很好地表示课程之间的依赖关系。图1展示了要获得计算机科学学位，所需学习课程的先后顺序。

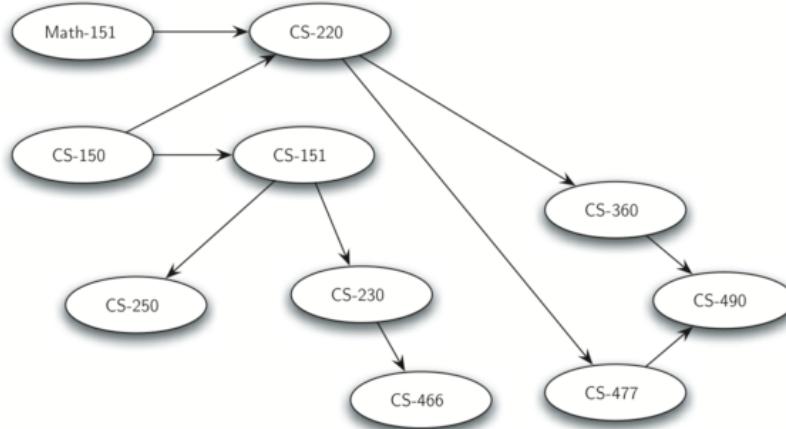


图1 计算机课程的学习顺序

抽象出来看，图 (Graph) 由顶点 (Vertex) 和边 (Edge) 组成，每条边的两端都必须是图的两个顶点(可以是相同的顶点)。而记号 $G(V,E)$ 表示图 G 的顶点集为 V 、边集为 E 。图 2 是一个抽象出来的图。

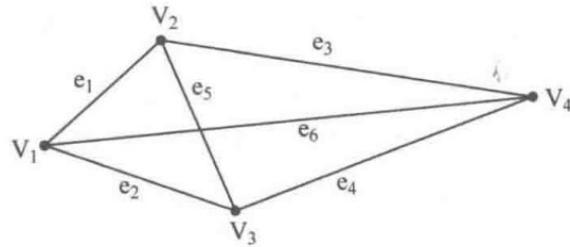


图2 抽象出来的图

一般来说，图可分为有向图和无向图。有向图的所有边都有方向，即确定了顶点到顶点的一个指向；而无向图的所有边都是双向的，即无向边所连接的两个顶点可以互相到达。在一些问题中，可以把无向图当作所有边都是正向和负向的两条有向边组成，这对解决一些问题很有帮助。图 3是有向图和无向图的举例。

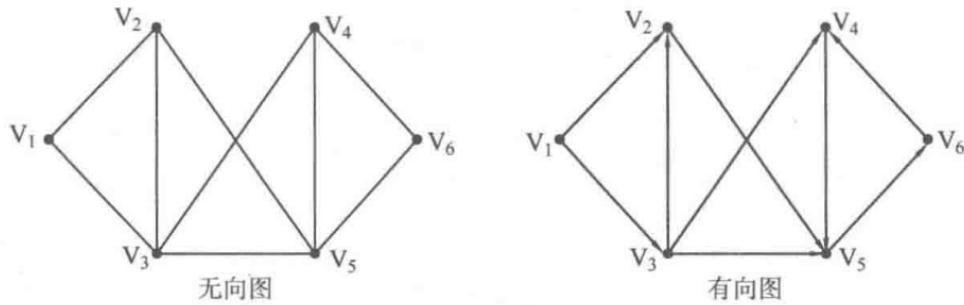


图3 无向图与有向图

顶点 Vertex

顶点又称节点，是图的基础部分。它可以有自己的名字，我们称作“键”。顶点也可以带有附加信息，我们称作“有效载荷”。

边Edge

边是图的另一个基础部分。两个顶点通过一条边相连，表示它们之间存在关系。边既可以是单向的，也可以是双向的。如果图中的所有边都是单向的，我们称之为有向图。图1明显是一个有向图，因为必须修完某些课程后才能修后续的课程。

度Degree

顶点的度是指和该顶点相连的边的条数。特别是对于有向图来说，顶点的出边条数称为该顶点的出度，顶点的入边条数称为该顶点的入度。例如图3的无向图中，V1的度为2，V5的度为4；有向图例子中，V2的出度为1、入度为2。

权值Weight

顶点和边都可以有一定属性，而量化的属性称为权值，顶点的权值和边的权值分别称为点权和边权。权值可以根据问题的实际背景设定，例如点权可以是城市中资源的数目，边权可以是两个城市之间来往所需要的时间、花费或距离。

有了上述定义之后，再来正式地定义图**Graph**。图可以用 G 来表示，并且 $G = (V, E)$ 。其中， V 是一个顶点集合， E 是一个边集合。每一条边是一个二元组 (v, w) ，其中 $w, v \in V$ 。可以向边的二元组中再添加一个元素，用于表示权重。子图 s 是一个由边 e 和顶点 v 构成的集合，其中 $e \in E$ 且 $v \in V$ 。

图4展示了一个简单的带权有向图。我们可以用6个顶点和9条边的两个集合来正式地描述这个图：

$$V = \{V_0, V_1, V_2, V_3, V_4, V_5\}$$

$$E = \left\{ (v_0, v_1, 5), (v_1, v_2, 4), (v_2, v_3, 9), (v_3, v_4, 7), (v_4, v_0, 1), \right. \\ \left. (v_0, v_5, 2), (v_5, v_4, 8), (v_3, v_5, 3), (v_5, v_2, 1) \right\}$$

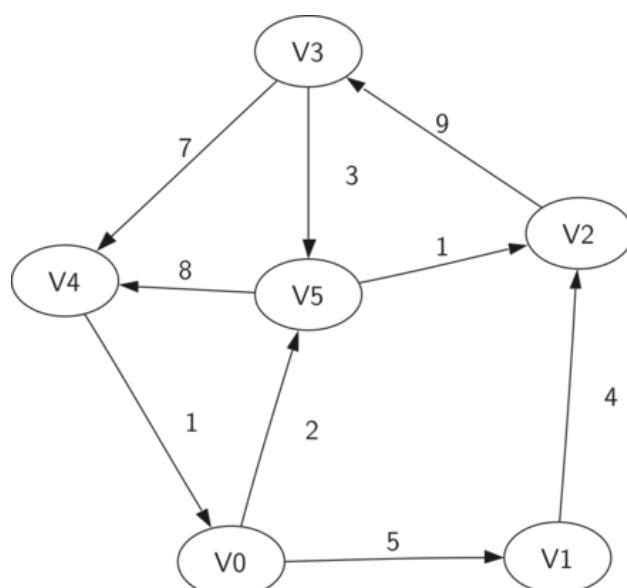


图4 简单的带权有向图

图4中的例子还体现了其他两个重要的概念。

路径Path

路径是由边连接的顶点组成的序列。路径的正式定义为 w_1, w_2, \dots, w_n , 其中对于所有的 $1 \leq i \leq n-1$, 有 $(w_i, w_{i+1}) \in E$ 。无权重路径的长度是路径上的边数, 有权重路径的长度是路径上的边的权重之和。以图4为例, 从 V3 到 V1 的路径是顶点序列(V3, V4, V0, V1), 相应的边是 $\{(v3, v4, 7), (v4, v0, 1), (v0, v1, 5)\}$ 。

环Cycle

环是有向图中的一条起点和终点为同一个顶点的路径。例如, 图4中的路径(V5, V2, V3, V5)就是一个环。没有环的图被称为无环图, 没有环的有向图被称为有向无环图, 简称为DAG。接下来会看到, DAG能帮助我们解决很多重要的问题。

1.1 编程题目

啃题目练习网址, <https://sunnywhy.com/sfbj>

sy374: 无向图的度 简单

<https://sunnywhy.com/sfbj/10/1/374>

现有一个共n个顶点、m条边的无向图 (假设顶点编号为从 0 到 $n-1$) , 求每个顶点的度。

输入

第一行两个整数n、m ($1 \leq n \leq 100, 0 \leq m \leq \frac{n(n-1)}{2}$) , 分别表示顶点数和边数;

接下来m行, 每行两个整数u、v ($0 \leq u \leq n-1, 0 \leq v \leq n-1, u \neq v$) , 表示一条边的两个端点的编号。数据保证不会有重边。

输出

在一行中输出n个整数, 表示编号为从 0 到 $n-1$ 的顶点的度。整数之间用空格隔开, 行末不允许有多余的空格。

样例1

输入

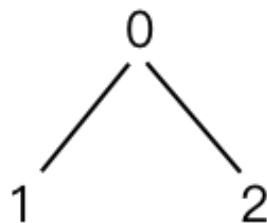
1	3 2
2	0 1
3	0 2

输出

1	2 1 1
---	-------

解释

对应的无向图如下图所示，0号顶点的度为2，1号和2号顶点的度为1。



为了求解每个顶点的度，我们可以创建一个列表来存储每个顶点的度，初始值都为0。然后，对于每条边，我们将边的两个端点的度都加1。

以下是实现这个过程的Python代码：

```
1 n, m = map(int, input().split())
2 degrees = [0] * n
3 for _ in range(m):
4     u, v = map(int, input().split())
5     degrees[u] += 1
6     degrees[v] += 1
7
8 print(' '.join(map(str, degrees)))
```

这段代码首先读取输入，然后创建一个列表来存储每个顶点的度。然后，它遍历每条边，将边的两个端点的度都加1。最后，它输出每个顶点的度。

sy375: 有向图的度 简单

<https://sunnywhy.com/sfbj/10/1/375>

现有一个共n个顶点、m条边的有向图（假设顶点编号为从0到n-1），求每个顶点的入度和出度。

输入

第一行两个整数n、m ($1 \leq n \leq 100, 0 \leq m \leq n(n - 1)$)，分别表示顶点数和边数；

接下来m行，每行两个整数u、v ($0 \leq u \leq n - 1, 0 \leq v \leq n - 1, u \neq v$)，表示一条边的两个端点的编号。
数据保证不会有重边。

输出

输出行，每行为编号从0到n-1的一个顶点的入度和出度，中间用空格隔开。

样例1

输入

```
1 | 3 3  
2 | 0 1  
3 | 0 2  
4 | 2 1
```

输出

```
1 | 0 2  
2 | 2 0  
3 | 1 1
```

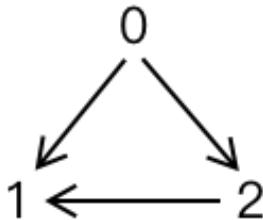
解释

对应的有向图如下图所示。

0号顶点有0条入边，2条出边，因此入度为0，出度为2；

1号顶点有2条入边，0条出边，因此入度为2，出度为0；

2号顶点有1条入边，1条出边，因此入度为1，出度为1。



为了求解每个顶点的入度和出度，我们可以创建两个列表来分别存储每个顶点的入度和出度，初始值都为0。然后，对于每条边，我们将起点的出度加1，终点的入度加1。

以下是实现这个过程的Python代码：

```
1 n, m = map(int, input().split())  
2 in_degrees = [0] * n  
3 out_degrees = [0] * n  
4 for _ in range(m):  
5     u, v = map(int, input().split())  
6     out_degrees[u] += 1  
7     in_degrees[v] += 1  
8  
9 for i in range(n):  
10    print(in_degrees[i], out_degrees[i])
```

这段代码首先读取输入，然后创建两个列表来存储每个顶点的入度和出度。然后，它遍历每条边，将边的起点的出度加1，终点的入度加1。最后，它输出每个顶点的入度和出度。

2 图的表示方法

图的抽象数据类型由下列方法定义。

- Graph() 新建一个空图。
- addVertex(vert) 向图中添加一个顶点实例。
- addEdge(fromVert, toVert) 向图中添加一条有向边，用于连接顶点fromVert和toVert。
- addEdge(fromVert, toVert, weight) 向图中添加一条带权重weight的有向边，用于连接顶点fromVert和toVert。
- getVertex(vertKey) 在图中找到名为vertKey的顶点。
- getVertices() 以列表形式返回图中所有顶点。
- in 通过 vertex in graph 这样的语句，在顶点存在时返回True，否则返回False。

根据图的正式定义，可以通过多种方式在Python中实现图的抽象数据类型（ADT）。在使用不同的表达方式来实现图的抽象数据类型时，需要做很多取舍。有两种非常著名的图实现，它们分别是邻接矩阵 **adjacency matrix** 和邻接表**adjacency list**。本节会解释这两种实现，并且用 Python 类来实现邻接表。

dict套list, dict套queue, dict套dict, 真好用啊！

确实，Python 中的字典（`dict`）是非常灵活和强大的数据结构，可以用于构建各种复杂的数据结构。

1. Dict套List:

- 可以用字典来表示多个对象的集合，其中每个对象都有一个唯一的键，而值则是一个列表，存储相关的数据。
- 例如，可以使用字典来表示学生和他们的成绩列表：`{"张三": [90, 85, 95], "李四": [88, 92, 90]}`。

2. Dict套Queue:

- 可以用字典来实现队列的功能，其中字典的键可以表示队列中的元素，而值则是用于实现队列操作的相关信息。
- 例如，可以使用字典来实现一个任务队列，其中键是任务名称，值是任务的优先级或其他相关信息。
- 例如，27925:小组队列，<http://cs101.openjudge.cn/practice/27925/>

3. Dict套Dict:

- 可以用字典来表示更复杂的数据结构，例如树、图等。其中外层字典的键可以表示节点或顶点，而值则是另一个字典，存储该节点或顶点的属性以及与之相关联的其他节点或顶点。
- 例如，可以使用字典来表示图，其中外层字典的键是节点，内层字典的键是该节点与其他节点之间的边，值是边的权重或其他相关信息。

使用字典来构建这些复杂的数据结构，使得数据的组织和操作更加方便和高效，同时也提高了代码的可读性和可维护性。因此，dict套list、dict套queue、dict套dict等形式的数据结构在Python中确实是非常好用的。

dict的value如果是list/set, 是邻接表。dici嵌套dict 是 字典树/前缀树/Trie

是的，你提到的两种数据结构分别是邻接表和字典树（前缀树， Trie）。

1. 邻接表：在图论中，邻接表是一种表示图的常见方式之一。如果你使用字典（dict）来表示图的邻接关系，并且将每个顶点的邻居顶点存储为列表（list），那么就构成了邻接表。例如：

```
1 graph = {
2     'A': ['B', 'C'],
3     'B': ['A', 'D'],
4     'C': ['A', 'D'],
5     'D': ['B', 'C']
6 }
```

27928: 遍历树

<http://cs101.openjudge.cn/practice/27928/>

请你对输入的树做遍历。遍历的规则是：遍历到每个节点时，按照该节点和所有子节点的值从小到大进行遍历，例如：

```
1           7
2       /   |   \
3     10   3   6
```

对于这个树，你应该先遍历值为3的子节点，然后是值为6的子节点，然后是父节点7，最后是值为10的子节点。

本题中每个节点的值为互不相同的正整数，最大不超过9999999。

输入

第一行：节点个数n (n<500)

接下来的n行：第一个数是此节点的值，之后的数分别表示它的所有子节点的值。每个数之间用空格隔开。如果没有子节点，该行便只有一个数。

输出

输出遍历结果，一行一个节点的值。

样例输入

```
1 sample1 input:
2 4
3 7 10 3 6
4 10
5 6
6 3
7
8 sample1 output:
9 3
10 6
11 7
12 10
```

样例输出

```
1 sample2 input:  
2 6  
3 10 3 1  
4 7  
5 9 2  
6 2 10  
7 3 7  
8 1  
9  
10 sample2 output:  
11 2  
12 1  
13 3  
14 7  
15 10  
16 9
```

来源

2024spring zht

```
1 # 李思哲 物理学院  
2 class TreeNode:  
3     def __init__(self, value):  
4         self.value = value  
5         self.children = []  
6  
7  
8     def traverse_print(self, nodes):  
9         if self.children == []:  
10             print(self.value)  
11             return  
12         pac = {self.value: self}  
13         for child in self.children:  
14             pac[child] = nodes[child]  
15         for value in sorted(pac.keys()):  
16             if value in self.children:  
17                 traverse_print(pac[value], nodes)  
18             else:  
19                 print(self.value)  
20  
21  
22     n = int(input())  
23     nodes = {}  
24     children_list = []  
25     for i in range(n):  
26         info = list(map(int, input().split()))  
27         nodes[info[0]] = TreeNode(info[0])
```

```

28     for child_value in info[1:]:
29         nodes[info[0]].children.append(child_value)
30         children_list.append(child_value)
31     root = nodes[[value for value in nodes.keys() if value not in children_list][0]]
32     traverse_print(root, nodes)
33

```

2. 字典树（前缀树，Trie）：字典树是一种树形数据结构，用于高效地存储和检索字符串数据集中的键。如果你使用嵌套的字典来表示字典树，其中每个字典代表一个节点，键表示路径上的字符，而值表示子节点，那么就构成了字典树。例如：

```

1  trie = {
2      'a': {
3          'p': {
4              'p': {
5                  'l': {
6                      'e': {'is_end': True}
7                  }
8              }
9          }
10     },
11     'b': {
12         'a': {
13             'l': {
14                 'l': {'is_end': True}
15             }
16         }
17     },
18     'c': {
19         'a': {
20             't': {'is_end': True}
21         }
22     }
23 }

```

这样的表示方式使得我们可以非常高效地搜索和插入字符串，特别是在大型数据集上。

例如，04089:电话号码，<http://cs101.openjudge.cn/practice/04089/>

神奇的dict

字典（dict）是Python中非常强大和灵活的数据结构之一，它可以用来存储键值对，是一种可变容器模型，可以存储任意数量的Python对象。

字典在Python中被广泛用于各种场景，例如：

1. **哈希映射**：字典提供了一种快速的键值查找机制，可以根据键快速地检索到相应的值。这使得字典成为了哈希映射（Hash Map）的理想实现。

2. 符号表：在编程语言的实现中，字典常常被用作符号表，用来存储变量名、函数名等符号和它们的关联值。
3. 配置文件：字典可以用来表示配置文件中的键值对，例如JSON文件就是一种常见的字典格式。
4. 缓存：字典常常用于缓存中，可以将计算结果与其输入参数关联起来，以便后续快速地检索到相同参数的计算结果。
5. 图的表示：如前文所述，字典可以用来表示图的邻接关系，是一种常见的图的表示方式。

由于其灵活性和高效性，字典在Python中被广泛应用于各种场景，并且被称为是Python中最常用的数据结构之一。

2.1 邻接矩阵

要实现图，最简单的方式就是使用二维矩阵。在矩阵实现中，每一行和每一列都表示图中的一个顶点。第v行和第w列交叉的格子中的值表示从顶点v到顶点w的边的权重。如果两个顶点被一条边连接起来，就称它们是相邻的。图5展示了图4对应的邻接矩阵。格子中的值表示从顶点v到顶点w的边的权重。

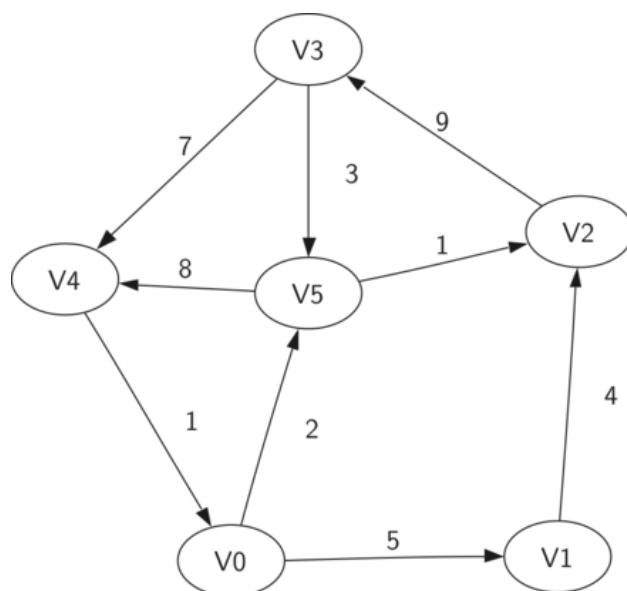


图4 简单的带权有向图

	v0	v1	v2	v3	v4	v5
v0		5				2
v1			4			
v2				9		
v3					7	3
v4	1					
v5			1		8	

Figure 5: An Adjacency Matrix Representation for a Graph

邻接矩阵的优点是简单。对于小图来说，邻接矩阵可以清晰地展示哪些顶点是相连的。但是，图5中的绝大多数单元格是空的，我们称这种矩阵是“稀疏”的。对于存储稀疏数据来说，矩阵并不高效。

邻接矩阵适用于表示有很多条边的图。但是，“很多条边”具体是什么意思呢？要填满矩阵，共需要多少条边？由于每一行和每一列对应图中的每一个顶点，因此填满矩阵共需要 $|V|^2$ 条边。当每一个顶点都与其他所有顶点相连时，矩阵就被填满了。在现实世界中，很少有问题能够达到这种连接度。

2.2 邻接表

为了实现稀疏连接的图，更高效的方式是使用邻接表。在邻接表实现中，我们为图对象的所有顶点保存一个主列表，同时为每一个顶点对象都维护一个列表，其中记录了与它相连的顶点。在对Vertex类的实现中，我们使用字典（而不是列表），字典的键是顶点，值是权重。图6展示了图4所对应的邻接表

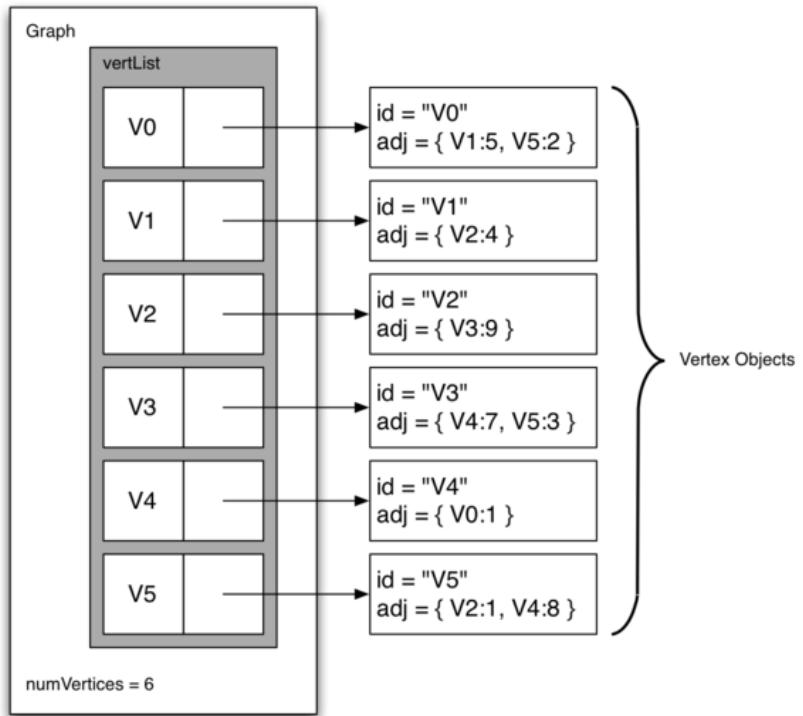


Figure 6: An Adjacency List Representation of a Graph

邻接表的优点是能够紧凑地表示稀疏图。此外，邻接表也有助于方便地找到与某一个顶点相连的其他所有顶点。

*关联矩阵

关联矩阵是一种图的表示方法，通常用于表示有向图。在关联矩阵中，行代表顶点，列代表边，如果顶点与边相连，则在对应的位置填上1，否则填上0。

虽然关联矩阵和邻接表都可以用来表示图，但它们在内存消耗和操作效率上有所不同。关联矩阵的存储空间复杂度为 $O(V^2)$ ，其中V是顶点数，E是边数。但是，邻接表的存储空间复杂度为 $O(V+E)$ ，它通常比关联矩阵更节省空间。

另外，对于某些图的操作，如查找两个顶点之间是否存在边，邻接表的操作效率更高。而对于其他操作，如计算图的闭包或者判断两个图是否同构等，关联矩阵可能更方便。

因此，关联矩阵和邻接表都有各自的优势和适用场景，选择使用哪种数据结构取决于具体的应用需求和对内存和操作效率的考虑。

图的闭包是指对于一个有向图或无向图，将所有顶点对之间的可达路径都加入到图中的过程。闭包可以用于分析图中的传递性关系。

判断两个图是否同构是指确定两个图是否具有相同的结构，即它们的顶点和边的连接关系是否一致。同构性是图的一个重要性质，可以用于在不同图之间进行匹配、比较和分类。

图的闭包和判断两个图是否同构是两个不同的概念，它们在图论中具有不同的应用和研究方向。

图的闭包可以通过图的邻接矩阵或邻接表来计算。对于有向图，可以使用传递闭包算法，通过矩阵乘法或深度优先搜索来确定顶点对之间的可达性。对于无向图，可以使用传递闭包算法或者深度优先搜索来计算。

判断两个图是否同构是一个复杂的问题，没有简单的算法可以解决。通常使用图同构性的定义和算法来进行判断，其中一种常用的方法是通过图的特征向量或特征值来比较两个图的结构。其他方法包括基于图的同构性的哈希算法和子图同构性的算法。

判断两个图是否同构是一个重要的图论问题，它在许多领域中都有应用，包括网络分析、化学和生物信息学等。

当涉及到图的闭包和判断两个图是否同构时，以下是一些示例：

1. 图的闭包：

考虑一个有向图，表示人际关系网络。假设有以下关系：A认识B，B认识C，C认识D。闭包操作将在图中添加所有可达的路径。在这种情况下，闭包操作将在图中添加A到D的边，因为A通过B和C可以到达D。闭包操作后的图将包含A、B、C、D四个顶点，并存在A到D的边。

2. 判断两个图是否同构：

考虑两个无向图G1和G2。G1有三个顶点{A, B, C}和两条边{A-B, B-C}。G2有三个顶点{X, Y, Z}和两条边{X-Y, Y-Z}。通过比较两个图的边连接关系，我们可以发现G1和G2具有相同的结构。因此，可以判断G1和G2是同构的。

尽管图的闭包和同构问题在理论上具有重要性，并且在某些应用领域中有实际意义，但在常见的编程题目中，它们并不是经常出现的主题。更常见的图相关编程题目包括广度优先搜索、深度优先搜索、拓扑排序、最短路径算法（如Dijkstra算法和Floyd-Warshall算法）、最小生成树算法（如Prim算法和Kruskal算法）等。

闭包和最小生成树是两个不同的概念？

闭包是指对于一个有向图或无向图，将所有顶点对之间的可达路径都加入到图中的过程。闭包操作后的图包含了原图中所有顶点和它们之间的可达路径。闭包操作可以用于分析图中的传递性关系。

最小生成树是指在一个连通无向图中，选择一棵包含所有顶点且边权重之和最小的树。最小生成树可以用来找到连接图中所有顶点的最短路径。

闭包和最小生成树是针对不同类型的图的不同操作。闭包适用于有向图和无向图，而最小生成树适用于无向图。闭包操作是为了添加路径，以展示顶点之间的可达性，而最小生成树是为了找到一棵最小权重的树，以连接所有顶点。

因此，闭包和最小生成树是不同的概念，在图论中具有不同的应用和意义。

下面是一个用 Python 实现的关联矩阵示例：

```
1 class Graph:
2     def __init__(self, vertices, edges):
3         self.vertices = vertices
4         self.edges = edges
5         self.adj_matrix = self.create_adj_matrix()
6
7     def create_adj_matrix(self):
8         # Create an empty adjacency matrix
```

```

9     adj_matrix = [[0] * len(self.edges) for _ in range(len(self.vertices))]
10
11     # Fill adjacency matrix based on edges
12     for i, vertex in enumerate(self.vertices):
13         for j, edge in enumerate(self.edges):
14             if vertex in edge:
15                 adj_matrix[i][j] = 1
16
17     return adj_matrix
18
19 def display_adj_matrix(self):
20     for row in self.adj_matrix:
21         print(row)
22
23
24 # Example usage
25 if __name__ == "__main__":
26     vertices = ['A', 'B', 'C', 'D']
27     edges = [('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'A')]
28
29     graph = Graph(vertices, edges)
30     print("Adjacency Matrix:")
31     graph.display_adj_matrix()

```

在这个示例中，定义了一个 `Graph` 类来表示图。在初始化时，传入顶点列表和边列表。然后，使用 `create_adj_matrix` 方法来创建关联矩阵。最后，调用 `display_adj_matrix` 方法来显示关联矩阵。

注意：这个示例假设了顶点和边的表示是字符串形式，可以根据需要调整表示方式。

2.3 编程题目

sy376: 无向图的邻接矩阵 简单

<https://sunnywhy.com/sfbj/10/2/376>

现有一个共n个顶点、m条边的无向图（假设顶点编号为从0到n-1），将其按邻接矩阵的方式存储（存在边的位置填充1，不存在边的位置填充0），然后输出整个邻接矩阵。

输入

第一行两个整数n、m ($1 \leq n \leq 100, 0 \leq m \leq \frac{n(n-1)}{2}$)，分别表示顶点数和边数；

接下来m行，每行两个整数u、v ($0 \leq u \leq n - 1, 0 \leq v \leq n - 1, u \neq v$)，表示一条边的两个端点的编号。数据保证不会有重边。

输出

输出n行n列，表示邻接矩阵。整数之间用空格隔开，行末不允许有多余的空格。

样例1

输入

1	3	2
2	0	1
3	0	2

输出

1	0	1	1
2	1	0	0
3	1	0	0

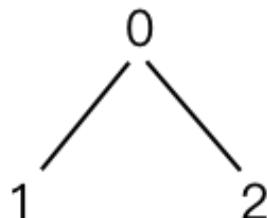
解释

对应的无向图如下图所示。

0号顶点有2条出边，分别到达1号顶点和2号顶点；

1号顶点有1条出边，到达0号顶点；

2号顶点有1条出边，到达0号顶点。



为了将无向图按邻接矩阵的方式存储，我们可以创建一个 $n \times n$ 的二维列表，初始值都为0。然后，对于每条边，我们将边的两个端点对应的位置填充为1。

以下是实现这个过程的Python代码：

```
1 n, m = map(int, input().split())
2 adjacency_matrix = [[0]*n for _ in range(n)]
3 for _ in range(m):
4     u, v = map(int, input().split())
5     adjacency_matrix[u][v] = 1
6     adjacency_matrix[v][u] = 1
7
8 for row in adjacency_matrix:
9     print(' '.join(map(str, row)))
```

这段代码首先读取输入，然后创建一个 $n \times n$ 的二维列表来存储邻接矩阵。然后，它遍历每条边，将边的两个端点对应的位置填充为1。最后，它输出整个邻接矩阵。

sy377: 有向图的邻接矩阵 简单

<https://sunnywhy.com/sfbj/10/2/377>

现有一个共n个顶点、m条边的有向图（假设顶点编号为从0到n-1），将其按邻接矩阵的方式存储（存在边的位置填充1，不存在边的位置填充0），然后输出整个邻接矩阵。

输入

第一行两个整数n、m ($1 \leq n \leq 100, 0 \leq m \leq n(n - 1)$)，分别表示顶点数和边数；

接下来m行，每行两个整数u、v ($0 \leq u \leq n - 1, 0 \leq v \leq n - 1, u \neq v$)，表示一条边的起点和终点的编号。数据保证不会有重边。

输出

输出n行n列，表示邻接矩阵。整数之间用空格隔开，行末不允许有多余的空格。

样例1

输入

1	3	3
2	0	1
3	0	2
4	2	1

输出

1	0	1	1
2	0	0	0
3	0	1	0

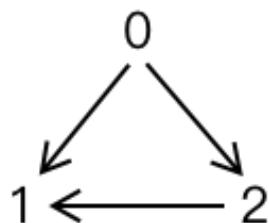
解释

对应的有向图如下图所示。

0号顶点有2条出边，分别到达1号顶点和2号顶点；

1号顶点有0条出边；

2号顶点有1条出边，到达1号顶点。



为了将有向图按邻接矩阵的方式存储，我们可以创建一个 $n \times n$ 的二维列表，初始值都为0。然后，对于每条边，我们将边的起点和终点对应的位置填充为1。

以下是实现这个过程的Python代码：

```
1 n, m = map(int, input().split())
2 adjacency_matrix = [[0]*n for _ in range(n)]
3 for _ in range(m):
4     u, v = map(int, input().split())
5     adjacency_matrix[u][v] = 1
6
7 for row in adjacency_matrix:
8     print(' '.join(map(str, row)))
```

这段代码首先读取输入，然后创建一个 $n \times n$ 的二维列表来存储邻接矩阵。然后，它遍历每条边，将边的起点和终点对应的位置填充为1。最后，它输出整个邻接矩阵。

sy378: 无向图的邻接表 简单

<https://sunnywhy.com/sfbj/10/2/378>

现有一个共 n 个顶点、 m 条边的无向图（假设顶点编号为从0到 $n-1$ ），将其按邻接表的方式存储，然后输出整个邻接表。

输入

第一行两个整数 n 、 m ($1 \leq n \leq 100, 0 \leq m \leq \frac{n(n-1)}{2}$)，分别表示顶点数和边数；

接下来 m 行，每行两个整数 u 、 v ($0 \leq u \leq n-1, 0 \leq v \leq n-1, u \neq v$)，表示一条边的两个端点的编号。数据保证不会有重边。

输出

输出行，按顺序给出编号从0到 $n-1$ 的顶点的所有出边，每行格式如下：

```
1 | id(k) v_1 v_2 ... v_k
```

其中表示当前顶点的编号，表示该顶点的出边数量， \dots 表示条出边的终点编号（按边输入的顺序输出）。行末不允许有多余的空格。

样例1

输入

```
1 | 3 2
2 | 0 1
3 | 0 2
```

输出

```
1 | 0(2) 1 2
2 | 1(1) 0
3 | 2(1) 0
```

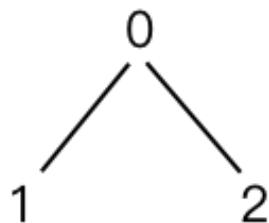
解释

对应的无向图如下图所示。

0号顶点有2条出边，分别到达1号顶点和2号顶点；

1号顶点有1条出边，到达0号顶点；

2号顶点有1条出边，到达0号顶点。



为了将无向图按邻接表的方式存储，我们可以创建一个列表，其中每个元素都是一个列表，表示一个顶点的所有邻接顶点。然后，对于每条边，我们将边的两个端点添加到对方的邻接列表中。

以下是实现这个过程的Python代码：

```
1 n, m = map(int, input().split())
2 adjacency_list = [[] for _ in range(n)]
3 for _ in range(m):
4     u, v = map(int, input().split())
5     adjacency_list[u].append(v)
6     adjacency_list[v].append(u)
7
8 for i in range(n):
9     num = len(adjacency_list[i])
10    if num == 0:
11        print(f"{i}({num})")
12    else:
13        print(f"{i}({num})", ' '.join(map(str, adjacency_list[i])))
```

这段代码首先读取输入，然后创建一个列表来存储邻接表。然后，它遍历每条边，将边的两个端点添加到对方的邻接列表中。最后，它输出整个邻接表。

sy379: 有向图的邻接表 简单

<https://sunnywhy.com/sfbj/10/2/379>

现有一个共n个顶点、m条边的有向图（假设顶点编号为从0到n-1），将其按邻接表的方式存储，然后输出整个邻接表。

输入

第一行两个整数n、m ($1 \leq n \leq 100, 0 \leq m \leq n(n - 1)$)，分别表示顶点数和边数；

接下来m行，每行两个整数u、v ($0 \leq u \leq n - 1, 0 \leq v \leq n - 1, u \neq v$)，表示一条边的起点和终点的编号。数据保证不会有重边。

输出

输出行，按顺序给出编号从0到n-1的顶点的所有出边，每行格式如下：

1		id(k) v_1 v_2 ... v_k
---	--	-----------------------

其中id表示当前顶点的编号，k表示该顶点的出边数量，v1、v2、...、vk表示k条出边的终点编号（按边输入的顺序输出）。行末不允许有多余的空格。

样例1

输入

1		3 3
2		0 1
3		0 2
4		2 1

输出

1		0(2) 1 2
2		1(0)
3		2(1) 1

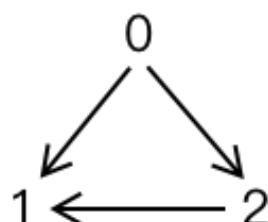
解释

对应的有向图如下图所示。

0号顶点有2条出边，分别到达1号顶点和2号顶点；

1号顶点有0条出边；

2号顶点有1条出边，到达1号顶点。



为了将有向图按邻接表的方式存储，我们可以创建一个列表，其中每个元素都是一个列表，表示一个顶点的所有邻接顶点。然后，对于每条边，我们将边的终点添加到起点的邻接列表中。

以下是实现这个过程的Python代码：

```
1 n, m = map(int, input().split())
2 adjacency_list = [[] for _ in range(n)]
3 for _ in range(m):
4     u, v = map(int, input().split())
5     adjacency_list[u].append(v)
6
7 for i in range(n):
8     num = len(adjacency_list[i])
9     if num == 0:
10         print(f'{i}({num})')
11     else:
12         print(f'{i}({num})', ' '.join(map(str, adjacency_list[i])))
```

这段代码首先读取输入，然后创建一个列表来存储邻接表。然后，它遍历每条边，将边的终点添加到起点的邻接列表中。最后，它输出整个邻接表。

2.4 图的类实现（笔试）

在Python中，通过字典可以轻松地实现邻接表。我们要创建两个类：Graph类存储包含所有顶点的主列表，Vertex类表示图中的每一个顶点。

Vertex使用字典connectedTo来记录与其相连的顶点，以及每一条边的权重。代码清单7-1展示了Vertex类的实现，其构造方法简单地初始化id（它通常是一个字符串），以及字典connectedTo。addNeighbor方法添加从一个顶点到另一个的连接。getConnections方法返回邻接表中的所有顶点，由connectedTo来表示。getWeight方法返回从当前顶点到以参数传入的顶点之间的边的权重。

代码清单7-1 Vertex类

```
1 class Vertex:
2     def __init__(self, key):
3         self.id = key
4         self.connectedTo = {}
5
6     def addNeighbor(self, nbr, weight=0):
7         self.connectedTo[nbr] = weight
8
9     def __str__(self):
10        return str(self.id) + ' connectedTo: ' + str([x.id for x in
11 self.connectedTo])
12
13     def getConnections(self):
14         return self.connectedTo.keys()
```

```

15     def getId(self):
16         return self.id
17
18     def getWeight(self,nbr):
19         return self.connectedTo[nbr]

```

Graph类的实现如代码清单7-2所示，其中包含一个将顶点名映射到顶点对象的字典。在图6中，该字典对象由灰色方块表示。Graph类也提供了向图中添加顶点和连接不同顶点的方法。getVertices方法返回图中所有顶点的名字。此外，我们还实现了`__iter__`方法，从而使遍历图中的所有顶点对象更加方便。总之，这两个方法使我们能够根据顶点名或者顶点对象本身遍历图中的所有顶点。

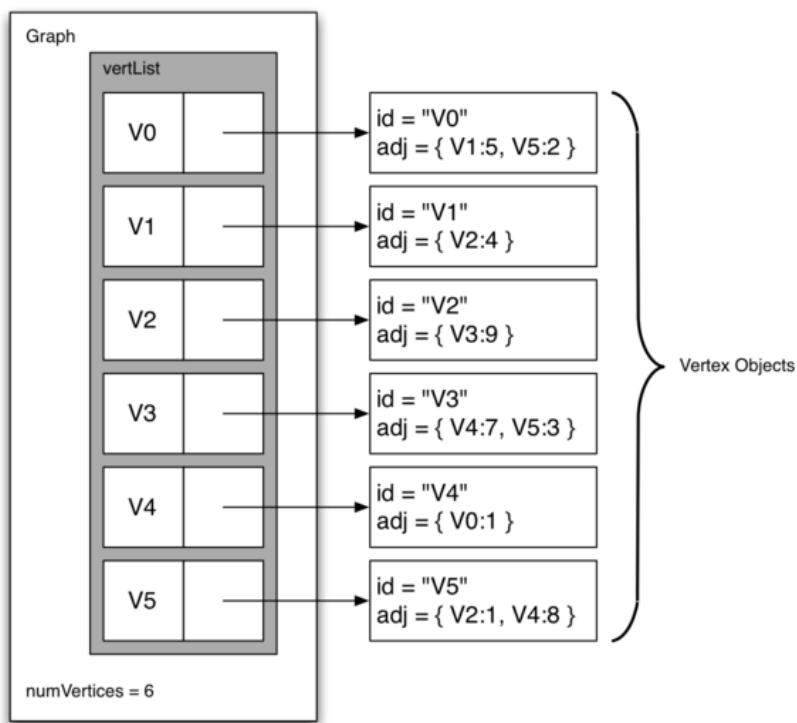


Figure 6: An Adjacency List Representation of a Graph

代码清单7-2 Graph类

```

1  class Graph:
2      def __init__(self):
3          self.vertList = {}
4          self.numVertices = 0
5
6      def addVertex(self,key):
7          self.numVertices = self.numVertices + 1
8          newVertex = Vertex(key)
9          self.vertList[key] = newVertex
10         return newVertex
11

```

```

12     def getVertex(self,n):
13         if n in self.vertList:
14             return self.vertList[n]
15         else:
16             return None
17
18     def __contains__(self,n):
19         return n in self.vertList
20
21     def addEdge(self,f,t,weight=0):
22         if f not in self.vertList:
23             nv = self.addVertex(f)
24         if t not in self.vertList:
25             nv = self.addVertex(t)
26         self.vertList[f].addNeighbor(self.vertList[t], weight)
27
28     def getVertices(self):
29         return self.vertList.keys()
30
31     def __iter__(self):
32         return iter(self.vertList.values())

```

下面的Python会话使用Graph类和Vertex类创建了如图6所示的图。首先创建6个顶点，依次编号为0~5。然后打印顶点字典。注意，对每一个键，我们都创建了一个Vertex实例。接着，添加将顶点连接起来的边。最后，用一个嵌套循环验证图中的每一条边都已被正确存储。请按照图6的内容检查会话的最终结果。

```

1  >>> g = Graph()
2  >>> for i in range(6):
3      ...     g.addVertex(i)
4  >>> g.vertList
5  {0: <adjGraph.Vertex instance at 0x41e18>,
6  1: <adjGraph.Vertex instance at 0x7f2b0>,
7  2: <adjGraph.Vertex instance at 0x7f288>,
8  3: <adjGraph.Vertex instance at 0x7f350>,
9  4: <adjGraph.Vertex instance at 0x7f328>,
10 5: <adjGraph.Vertex instance at 0x7f300>}
11 >>> g.addEdge(0,1,5)
12 >>> g.addEdge(0,5,2)
13 >>> g.addEdge(1,2,4)
14 >>> g.addEdge(2,3,9)
15 >>> g.addEdge(3,4,7)
16 >>> g.addEdge(3,5,3)
17 >>> g.addEdge(4,0,1)
18 >>> g.addEdge(5,4,8)
19 >>> g.addEdge(5,2,1)
20 >>> for v in g:
21     ...     for w in v.getConnections():
22     ...         print("( %s , %s )" % (v.getId(), w.getId()))
23 ...

```

```

24 ( 0 , 5 )
25 ( 0 , 1 )
26 ( 1 , 2 )
27 ( 2 , 3 )
28 ( 3 , 4 )
29 ( 3 , 5 )
30 ( 4 , 0 )
31 ( 5 , 4 )
32 ( 5 , 2 )

```

上面类方式定义顶点和图，要求掌握，数算B-2021笔试出现在算法部分。在机考中，也可以直接使用二维列表或者字典来表示邻接表。

19943: 图的拉普拉斯矩阵

<http://cs101.openjudge.cn/practice/19943/>

在图论中，度数矩阵是一个对角矩阵，其中包含的信息为的每一个顶点的度数，也就是说，每个顶点相邻的边数。邻接矩阵是图的一种常用存储方式。如果一个图一共有编号为0,1,2, ...,n-1的n个节点，那么邻接矩阵A的大小为n*n，对其中任一元素Aij，如果节点i, j直接有边，那么Aij=1；否则Aij=0。

将度数矩阵与邻接矩阵逐位相减，可以求得图的拉普拉斯矩阵。具体可见下图示意。

$$L := D - A$$

$$D = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix} \quad A = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix} \quad L = \begin{pmatrix} 2 & -1 & -1 & 0 \\ -1 & 3 & -1 & -1 \\ -1 & -1 & 3 & -1 \\ 0 & -1 & -1 & 2 \end{pmatrix}$$

现给出一个图中的所有边的信息，需要你输出该图的拉普拉斯矩阵。

输入

第一行2个整数，代表该图的顶点数n和边数m。

接下m行，每行为空格分隔的2个整数a和b，代表顶点a和顶点b之间有一条无向边相连，a和b均为大小范围在0到n-1之间的整数。输入保证每条无向边仅出现一次（如1 2和2 1是同一条边，并不会在数据中同时出现）。

输出

共n行，每行为以空格分隔的n个整数，代表该图的拉普拉斯矩阵。

样例输入

```
1 4 5  
2 2 1  
3 1 3  
4 2 3  
5 0 1  
6 0 2
```

样例输出

```
1 2 -1 -1 0  
2 -1 3 -1 -1  
3 -1 -1 3 -1  
4 0 -1 -1 2
```

来源

cs101 2019 Final Exam

```
1 class Vertex:  
2     def __init__(self, key):  
3         self.id = key  
4         self.connectedTo = {}  
5  
6     def addNeighbor(self, nbr, weight=0):  
7         self.connectedTo[nbr] = weight  
8  
9     def __str__(self):  
10        return str(self.id) + ' connectedTo: ' + str([x.id for x in  
self.connectedTo])  
11  
12    def getConnections(self):  
13        return self.connectedTo.keys()  
14  
15    def getId(self):  
16        return self.id  
17  
18    def getWeight(self, nbr):  
19        return self.connectedTo[nbr]  
20  
21 class Graph:  
22     def __init__(self):  
23         self.vertList = {}  
24         self.numVertices = 0  
25  
26     def addVertex(self, key):  
27         self.numVertices = self.numVertices + 1  
28         newVertex = Vertex(key)  
29         self.vertList[key] = newVertex  
30         return newVertex
```

```

31
32     def getVertex(self, n):
33         if n in self.vertList:
34             return self.vertList[n]
35         else:
36             return None
37
38     def __contains__(self, n):
39         return n in self.vertList
40
41     def addEdge(self, f, t, weight=0):
42         if f not in self.vertList:
43             nv = self.addVertex(f)
44         if t not in self.vertList:
45             nv = self.addVertex(t)
46         self.vertList[f].addNeighbor(self.vertList[t], weight)
47
48     def getVertices(self):
49         return self.vertList.keys()
50
51     def __iter__(self):
52         return iter(self.vertList.values())
53
54     def constructLaplacianMatrix(n, edges):
55         graph = Graph()
56         for i in range(n): # 添加顶点
57             graph.addVertex(i)
58
59         for edge in edges: # 添加边
60             a, b = edge
61             graph.addEdge(a, b)
62             graph.addEdge(b, a)
63
64         laplacianMatrix = [] # 构建拉普拉斯矩阵
65         for vertex in graph:
66             row = [0] * n
67             row[vertex.getId()] = len(vertex.getConnections())
68             for neighbor in vertex.getConnections():
69                 row[neighbor.getId()] = -1
70             laplacianMatrix.append(row)
71
72         return laplacianMatrix
73
74
75     n, m = map(int, input().split()) # 解析输入
76     edges = []
77     for i in range(m):
78         a, b = map(int, input().split())
79         edges.append((a, b))
80
81     laplacianMatrix = constructLaplacianMatrix(n, edges) # 构建拉普拉斯矩阵
82

```

```
83 |     for row in laplacianMatrix: # 输出结果
84 |         print(' '.join(map(str, row)))
```

二、图算法

3 (Week9) 基本图算法

3.1 宽度优先搜索 (BFS)

我们先给出BFS搜索框架，进而学习用BFS实现词梯问题。

3.1.1 基本图算法：BFS框架

Breadth First Search or BFS for a Graph

<https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/>

Breadth First Search (BFS) is a graph traversal algorithm that explores all the vertices in a graph at the current depth before moving on to the vertices at the next depth level. It starts at a specified vertex and visits all its neighbors before moving on to the next level of neighbors. **BFS** is commonly used in algorithms for pathfinding, connected components, and shortest path problems in graphs.

The algorithm for the BFS:

1. **Initialization:** Enqueue the starting node into a queue and mark it as visited.
2. **Exploration:**

While the queue is not empty:

- o Dequeue a node from the queue and visit it (e.g., print its value).
- o For each unvisited neighbor of the dequeued node:
 - Enqueue the neighbor into the queue.
 - Mark the neighbor as visited.

3. **Termination:** Repeat step 2 until the queue is empty.

This algorithm ensures that all nodes in the graph are visited in a breadth-first manner, starting from the starting node.

```
1  from collections import defaultdict, deque
2
3  # Class to represent a graph using adjacency list
4  class Graph:
5      def __init__(self):
6          self.adjList = defaultdict(list)
7
8      # Function to add an edge to the graph
9      def addEdge(self, u, v):
10         self.adjList[u].append(v)
11
12     # Function to perform Breadth First Search on a graph represented using adjacency
13     list
14     def bfs(self, startNode):
```

```

14     # Create a queue for BFS
15     queue = deque()
16     visited = set()
17
18     # Mark the current node as visited and enqueue it
19     visited.add(startNode)
20     queue.append(startNode)
21
22     # Iterate over the queue
23     while queue:
24         # Dequeue a vertex from queue and print it
25         currentNode = queue.popleft()
26         print(currentNode, end=" ")
27
28         # Get all adjacent vertices of the dequeued vertex currentNode
29         # If an adjacent has not been visited, then mark it visited and enqueue
30         it
31         for neighbor in self.adjList[currentNode]:
32             if neighbor not in visited:
33                 visited.add(neighbor)
34                 queue.append(neighbor)
35
36     # Create a graph
37     graph = Graph()
38
39     # Add edges to the graph
40     graph.addEdge(0, 1)
41     graph.addEdge(0, 2)
42     graph.addEdge(1, 3)
43     graph.addEdge(1, 4)
44     graph.addEdge(2, 4)
45
46     # Perform BFS traversal starting from vertex 0
47     print("Breadth First Traversal starting from vertex 0:", end=" ")
48     graph.bfs(0)
49     """
50     Breadth First Traversal starting from vertex 0: 0 1 2 3 4
51     Process finished with exit code 0
52     """

```

我们来分析一下时间复杂度和空间复杂度：

- 时间复杂度：BFS算法的时间复杂度取决于图的顶点数和边数。在最坏情况下，每个节点和边都会被访问一次，因此时间复杂度为 $O(V + E)$ ，其中 V 是顶点数， E 是边数。
- 空间复杂度：在BFS算法中，我们使用了一个队列来存储待访问的节点，以及一个集合来存储已经访问过的节点。因此，空间复杂度取决于队列的大小和集合的大小。在最坏情况下，队列的大小可以达到图的顶点数，集合的大小也可以达到图的顶点数。因此，空间复杂度为 $O(V)$ ，其中 V 是顶点数。

综上所述，这个程序的时间复杂度为 $O(V + E)$ ，空间复杂度为 $O(V)$ ，其中 V 是顶点数， E 是边数。

Time Complexity of BFS Algorithm: $O(V + E)$

- BFS explores all the vertices and edges in the graph. In the worst case, it visits every vertex and edge once. Therefore, the time complexity of BFS is $O(V + E)$, where V and E are the number of vertices and edges in the given graph.

Space Complexity of BFS Algorithm: $O(V)$

- BFS uses a queue to keep track of the vertices that need to be visited. In the worst case, the queue can contain all the vertices in the graph. Therefore, the space complexity of BFS is $O(V)$, where V and E are the number of vertices and edges in the given graph.

Applications of BFS in Graphs:

BFS has various applications in graph theory and computer science, including:

- **Shortest Path Finding:** BFS can be used to find the shortest path between two nodes in an unweighted graph. By keeping track of the parent of each node during the traversal, the shortest path can be reconstructed.
- **Cycle Detection:** BFS can be used to detect cycles in a graph. If a node is visited twice during the traversal, it indicates the presence of a cycle.
- **Connected Components:** BFS can be used to identify connected components in a graph. Each connected component is a set of nodes that can be reached from each other.
- **Topological Sorting:** BFS can be used to perform topological sorting on a directed acyclic graph (DAG). Topological sorting arranges the nodes in a linear order such that for any edge (u, v) , u appears before v in the order.
- **Level Order Traversal of Binary Trees:** BFS can be used to perform a level order traversal of a binary tree. This traversal visits all nodes at the same level before moving to the next level.
- **Network Routing:** BFS can be used to find the shortest path between two nodes in a network, making it useful for routing data packets in network protocols.

3.1.2 词梯问题

我们从词梯问题开始学习图算法。考虑这样一个任务：将单词FOOL转换成SAGE。在解决词梯问题时，必须每次只替换一个字母，并且每一步的结果都必须是一个单词，而不能是不存在的词。词梯问题由《爱丽丝梦游仙境》的作者刘易斯·卡罗尔于1878年提出。下面的单词转换序列是样例问题的一个解。

FOOL
POOL
POLL
POLE
PALE
SALE
SAGE

词梯问题有很多变体，例如在给定步数内完成转换，或者必须用到某个单词。在本节中，我们研究从起始单词转换到结束单词所需的最小步数。

由于主题是图，因此我们自然会想到使用图算法来解决这个问题。以下是大致步骤：

- 用图表示单词之间的关系；
- 用一种名为宽度优先搜索的图算法找到从起始单词到结束单词的最短路径。

1 构建词梯图

第一个问题是如何用图来表示大的单词集合。如果两个单词的区别仅在于有一个不同的字母，就用一条边将它们相连。如果能创建这样一个图，那么其中的任意一条连接两个单词的路径就是词梯问题的一个解。图1展示了一个小型图，可用于解决从FOOL到SAGE的词梯问题。注意，它是无向图，并且边没有权重。

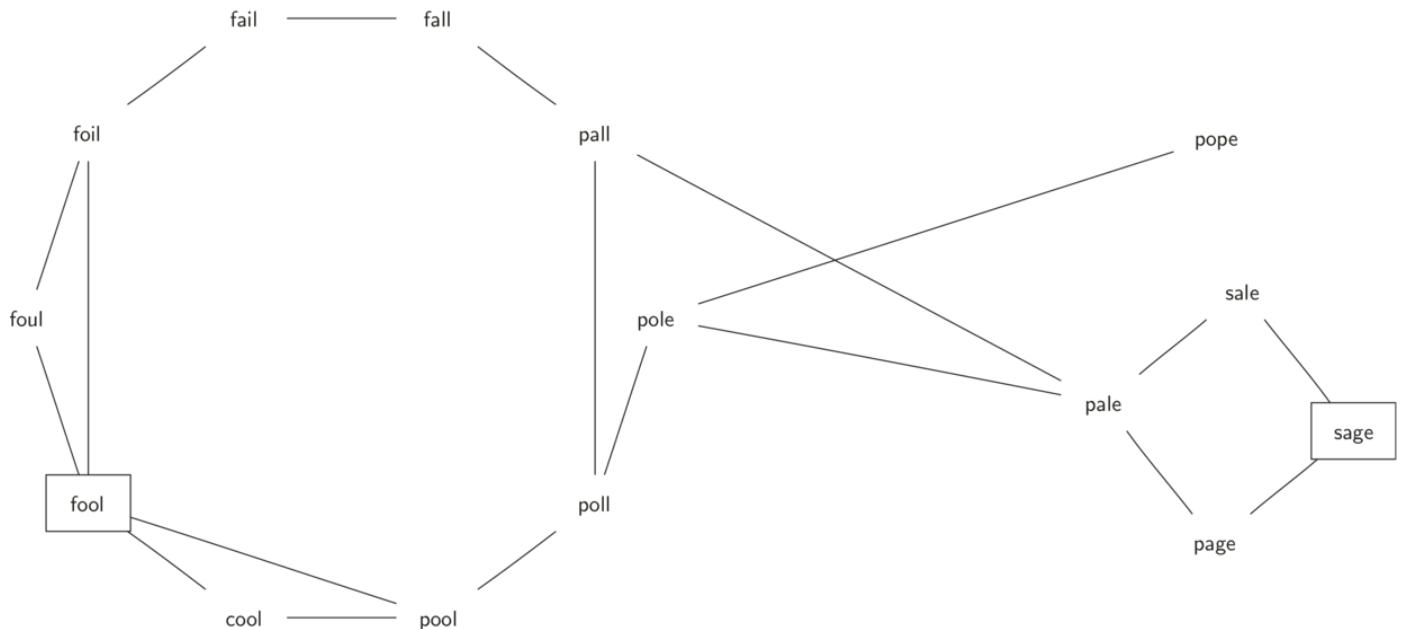


Figure 1: A Small Word Ladder Graph

创建这个图有多种方式。假设有一个单词列表，其中每个单词的长度都相同。首先，为每个单词创建顶点。为了连接这些顶点，可以将每个单词与列表中的其他所有单词进行比较。如果两个单词只相差一个字母，就可以在图中创建一条边，将它们连接起来。对于只有少量单词的情况，这个算法还不错。但是，假设列表中有3933个单词，将一个单词与列表中的其他所有单词进行比较，时间复杂度为 $O(n^2)$ 。对于3933个单词来说，这意味着要进行1500多万次比较。

采用下述方法，可以更高效地构建这个关系图。假设有数目巨大的桶，每一个桶上都标有一个长度为4的单词，但是某一个字母被下划线代替。图2展示了一些例子，如POP_。当处理列表中的每一个单词时，将它与桶上的标签进行比较。使用下划线作为通配符，我们将POPE和POPS放入同一个桶中。一旦将所有单词都放入对应的桶中之后，我们就知道，同一个桶中的单词一定是相连的。

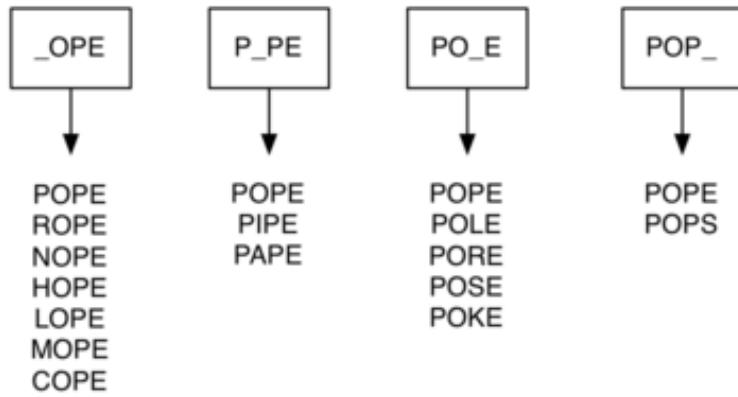


Figure 2: Word Buckets for Words That are Different by One Letter

在Python中，可以通过字典来实现上述方法。字典的键就是桶上的标签，值就是对应的单词列表。一旦构建好字典，就能利用它来创建图。首先为每个单词创建顶点，然后在字典中对应同一个键的单词之间创建边。代码清单1展示了构建图所需的Python代码。

代码清单1 为词梯问题构建单词关系

```

1  from pythonds3.graphs import Graph
2
3  def build_graph(filename):
4      buckets = {}
5      the_graph = Graph()
6      with open(filename, "r", encoding="utf8") as file_in:
7          all_words = file_in.readlines()
8          # all_words = ["bane", "bank", "bunk", "cane", "dale", "dunk", "foil",
9          "kale",
10         #           "lane", "male", "mane", "pale", "pole", "poll", "pool", "quip",
11         #           "quit", "rain", "sage", "sale", "same", "tank", "vain", "wane"
12         #           ]
13
14      # create buckets of words that differ by 1 letter
15      for line in all_words:
16          word = line.strip()
17          for i, _ in enumerate(word):
18              bucket = f"{word[:i]}_{word[i + 1:]}"
19              buckets.setdefault(bucket, set()).add(word)
20
21      # connect different words in the same bucket
22      for similar_words in buckets.values():
23          for word1 in similar_words:
24              for word2 in similar_words - {word1}:
25                  the_graph.add_edge(word1, word2)
26
27
28  return the_graph

```

这是我们在本节中遇到的第一个实际的图问题，你可能会好奇这个图的稀疏程度如何。本例中的单词列表文件 vocabulary.txt 包含 3,933 个单词。如果使用邻接矩阵表示，就会有 $15,468,489$ 个单元格 ($3933 \times 3933 = 15,468,489$)。用 buildGraph 函数创建的图一共有 42,600 条边（用邻接表来表示图）。因此，只有 0.27% 的单元格被填充。这显然是一个非常稀疏的矩阵。

2 BFS实现词梯问题

完成图的构建之后，就可以编写能帮我们找到最短路径的图算法。我们使用的算法叫作宽度优先搜索（breadth first search，以下简称BFS）。BFS是最简单的图搜索算法之一，也是后续要介绍的其他重要图算法的原型。

给定图 G 和起点 s ，BFS 通过边来访问在 G 中与 s 之间存在路径的顶点。BFS 的一个重要特性是，它会在访问完所有与 s 相距为 k 的顶点之后再去访问与 s 相距为 $k+1$ 的顶点。为了理解这种搜索行为，可以想象 BFS 以每次生成一层的方式构建一棵树。它会在访问任意一个孙节点之前将起点的所有子节点都添加进来。

为了记录进度，BFS 会将顶点标记成白色、灰色或黑色。在构建时，所有顶点都被初始化成白色。白色代表该顶点没有被访问过。当顶点第一次被访问时，它就会被标记为灰色；当 BFS 完成对该顶点的访问之后，它就会被标记为黑色。这意味着一旦顶点变为黑色，就没有白色顶点与之相连。灰色顶点仍然可能与一些白色顶点相连，这意味着还有额外的顶点可以访问。

来看看 bfs 函数如何构建对应于图 1 的宽度优先搜索树。从顶点 fool 开始，将所有与之相连的顶点都添加到树中。相邻的顶点有 pool、foil、foul，以及 cool。它们都被添加到队列中，作为之后要访问的顶点。

```
1 import sys
2 from collections import deque
3
4 class Graph:
5     def __init__(self):
6         self.vertices = {}
7         self.num_vertices = 0
8
9     def add_vertex(self, key):
10        self.num_vertices = self.num_vertices + 1
11        new_vertex = Vertex(key)
12        self.vertices[key] = new_vertex
13        return new_vertex
14
15    def get_vertex(self, n):
16        if n in self.vertices:
17            return self.vertices[n]
18        else:
19            return None
20
21    def __len__(self):
22        return self.num_vertices
23
```

```

24     def __contains__(self, n):
25         return n in self.vertices
26
27     def add_edge(self, f, t, cost=0):
28         if f not in self.vertices:
29             nv = self.add_vertex(f)
30         if t not in self.vertices:
31             nv = self.add_vertex(t)
32         self.vertices[f].add_neighbor(self.vertices[t], cost)
33
34     def get_vertices(self):
35         return list(self.vertices.keys())
36
37     def __iter__(self):
38         return iter(self.vertices.values())
39
40
41 class Vertex:
42     def __init__(self, num):
43         self.key = num
44         self.connectedTo = {}
45         self.color = 'white'
46         self.distance = sys.maxsize
47         self.previous = None
48         self.disc = 0
49         self.fin = 0
50
51     def add_neighbor(self, nbr, weight=0):
52         self.connectedTo[nbr] = weight
53
54     # def __lt__(self,o):
55     #     return self.id < o.id
56
57     # def setDiscovery(self, dtime):
58     #     self.disc = dtime
59     #
60     # def setFinish(self, ftime):
61     #     self.fin = ftime
62     #
63     # def getFinish(self):
64     #     return self.fin
65     #
66     # def getDiscovery(self):
67     #     return self.disc
68
69     def get_neighbors(self):
70         return self.connectedTo.keys()
71
72     # def getWeight(self, nbr):
73     #     return self.connectedTo[nbr]
74
75     # def __str__(self):

```

```

76     #      return str(self.key) + ":color " + self.color + ":disc " + str(self.disc)
77     + ":fin " + str(
78         #          self.fin) + ":dist " + str(self.distance) + ":pred \n\t[" +
79     str(self.previous) + "]\n"
80
81
82 def build_graph(filename):
83     buckets = {}
84     the_graph = Graph()
85     with open(filename, "r", encoding="utf8") as file_in:
86         all_words = file_in.readlines()
87     # all_words = ["bane", "bank", "bunk", "cane", "dale", "dunk", "foil", "fool",
88     "kale",
89     #           "lane", "male", "mane", "pale", "pole", "poll", "pool", "quip",
90     #           "quit", "rain", "sage", "sale", "same", "tank", "vain", "wane"
91     #           ]
92
93     # create buckets of words that differ by 1 letter
94     for line in all_words:
95         word = line.strip()
96         for i, _ in enumerate(word):
97             bucket = f"{word[:i]}_{word[i + 1:]}"
98             buckets.setdefault(bucket, set()).add(word)
99
100    # connect different words in the same bucket
101    for similar_words in buckets.values():
102        for word1 in similar_words:
103            for word2 in similar_words - {word1}:
104                the_graph.add_edge(word1, word2)
105
106
107
108 #g = build_graph("words_small")
109 g = build_graph("vocabulary.txt")
110 print(len(g))
111
112
113 def bfs(start):
114     start.distnace = 0
115     start.previous = None
116     vert_queue = deque()
117     vert_queue.append(start)
118     while len(vert_queue) > 0:
119         current = vert_queue.popleft() # 取队首作为当前顶点
120         for neighbor in current.get_neighbors(): # 遍历当前顶点的邻接顶点
121             if neighbor.color == "white":
122                 neighbor.color = "gray"
123                 neighbor.distance = current.distance + 1
124                 neighbor.previous = current

```

```

125             vert_queue.append(neighbor)
126             current.color = "black" # 当前顶点已经处理完毕，设黑色
127
128 """
129 BFS 算法主体是两个循环的嵌套：while-for
130     while 循环对图中每个顶点访问一次，所以是 O(|V|)；
131     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，
132     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|)；
133     综合起来 BFS 的时间复杂度为 O(V+|E|)
134
135 词梯问题还包括两个部分算法
136     建立 BFS 树之后，回溯顶点到起始顶点的过程，最多为 O(|V|)
137     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
138 """
139
140
141
142 #bfs(g.getVertex("fool"))
143
144 # 以FOOL为起点，进行广度优先搜索，从FOOL到SAGE的最短路径，
145 # 并为每个顶点着色、赋距离和前驱。
146 bfs(g.get_vertex("FOOL"))
147
148
149 # 回溯路径
150 def traverse(starting_vertex):
151     ans = []
152     current = starting_vertex
153     while (current.previous):
154         ans.append(current.key)
155         current = current.previous
156     ans.append(current.key)
157
158     return ans
159
160
161 # ans = traverse(g.get_vertex("sage"))
162 ans = traverse(g.get_vertex("SAGE")) # 从SAGE开始回溯，逆向打印路径，直到FOOL
163 print(*ans[::-1])
164 """
165 3867
166 FOOL TOOL TOLL TALL SALL SALE SAGE
167 """

```

代码及数据在，<https://github.com/GMyhf/2024spring-cs201/tree/main/code>

Q: "vocabulary.txt"是 3933行单词，但是**build_graph**，只有3867个顶点？检查了vocabulary.txt，没有重复的。

A: 建图过程中，如果桶里只有一个单词，就没有加入顶点集合。

3.1.3 分析宽度优先搜索

在学习其他图算法之前，让我们先分析BFS的性能。

```
1 def bfs(start):
2     start.distance = 0
3     start.previous = None
4     vert_queue = deque()
5     vert_queue.append(start)
6     while len(vert_queue) > 0:
7         current = vert_queue.popleft() # 取队首作为当前顶点
8         for neighbor in current.get_neighbors(): # 遍历当前顶点的邻接顶点
9             if neighbor.color == "white":
10                 neighbor.color = "gray"
11                 neighbor.distance = current.distance + 1
12                 neighbor.previous = current
13                 vert_queue.append(neighbor)
14         current.color = "black" # 当前顶点已经处理完毕，设黑色
```

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最多只执行一次，时间复杂度是 $O(|V|)$ ，因为只有白色顶点才能被访问并添加到队列中。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1次，一共是 $O(|E|)$ 。因此两个循环总的时间复杂度为 $O(|V|+|E|)$ 。

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 $O(|V|)$ ，因为最坏情况是整个图是一条长链。另外，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。

3.2 深度优先搜索（DFS）

我们先给出DFS搜索框架，进而学习用DFS实现骑士周游问题。

3.2.1 基本图算法：DFS框架

Depth First Search or DFS for a Graph

<https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/>

Depth First Traversal (or DFS) for a graph is similar to [Depth First Traversal of a tree](#). The only catch here is, that, unlike trees, graphs may contain cycles (a node may be visited twice). To avoid processing a node more than once, use a boolean visited array. A graph can have more than one DFS traversal.

递归实现DFS

```

1  from collections import defaultdict
2
3
4  class Graph:
5      def __init__(self):
6          self.graph = defaultdict(list)
7
8      def addEdge(self, u, v):
9          self.graph[u].append(v)
10
11     def DFS(self, v, visited=None):
12         if visited is None:
13             visited = set()
14         visited.add(v)
15         print(v, end=' ')
16         for neighbour in self.graph[v]:
17             if neighbour not in visited:
18                 self.DFS(neighbour, visited)
19
20
21 # Driver's code
22 if __name__ == "__main__":
23     g = Graph()
24     g.addEdge(0, 1)
25     g.addEdge(0, 2)
26     g.addEdge(1, 2)
27     g.addEdge(2, 0)
28     g.addEdge(2, 3)
29     g.addEdge(3, 3)
30
31     print("Following is Depth First Traversal (starting from vertex 2)")
32
33     # Function call
34     g.DFS(2)
35 """
36 Following is Depth First Traversal (starting from vertex 2)
37 2 0 1 3
38 """

```

在这个递归实现的深度优先搜索（DFS）算法中，让我们来分析一下时间复杂度和空间复杂度：

- 时间复杂度：DFS算法的时间复杂度取决于图的顶点数和边数。在最坏情况下，每个节点和边都会被访问一次，因此时间复杂度为 $O(V + E)$ ，其中 V 是顶点数， E 是边数。
- 空间复杂度：在递归实现的DFS中，我们使用了一个集合来存储已经访问过的节点。在最坏情况下，集合的大小可能会达到图的顶点数，因此空间复杂度为 $O(V)$ ，其中 V 是顶点数。此外，由于递归调用会产生函数调用栈，因此在最坏情况下，递归调用栈的深度可能会达到图的最大深度，所以空间复杂度还会受到递归深度的影响，但通常情况下，递归深度不会超过图的顶点数，因此我们仍然将空间复杂度视为 $O(V)$ 。

综上所述，这个递归实现的DFS算法的时间复杂度为 $O(V + E)$ ，空间复杂度为 $O(V)$ 。

How stack is implemented in DFS:-

1. Select a starting node, mark the starting node as visited and push it into the stack.
2. Explore any one of adjacent nodes of the starting node which are unvisited.
3. Mark the unvisited node as visited and push it into the stack.
4. Repeat this process until all the nodes in the tree or graph are visited.
5. Once all the nodes are visited, then pop all the elements in the stack until the stack becomes empty.

```
1  from collections import defaultdict
2
3
4  class Graph:
5      def __init__(self):
6          self.graph = defaultdict(list)
7
8      def addEdge(self, u, v):
9          self.graph[u].append(v)
10
11     def DFS(self, v):
12         visited = set()
13         stack = [v]
14
15         while stack:
16             current = stack.pop()
17             if current not in visited:
18                 print(current, end=' ')
19                 visited.add(current)
20                 stack.extend(reversed(self.graph[current]))
21
22
23 # Driver's code
24 if __name__ == "__main__":
25     g = Graph()
26     g.addEdge(0, 1)
27     g.addEdge(0, 2)
28     g.addEdge(1, 2)
29     g.addEdge(2, 0)
30     g.addEdge(2, 3)
31     g.addEdge(3, 3)
32
33     print("Following is Depth First Traversal (starting from vertex 2)")
34
35     # Function call
36     g.DFS(2)
37 """
38 Following is Depth First Traversal (starting from vertex 2)
39 2 0 1 3
40 """
```

如果需要记录路径的DFS，可以这样写

Algorithm for DFS

<https://www.codespeedy.com/depth-first-search-algorithm-in-python/>

This algorithm is a recursive algorithm which follows the concept of backtracking and implemented using stack data structure. But, what is backtracking.

Backtracking:-

It means whenever a tree or a graph is moving forward and there are no nodes along the existing path, the tree moves backwards along the same path which it went forward in order to find new nodes to traverse. This process keeps on iterating until all the unvisited nodes are visited.

How stack is implemented in DFS:-

1. Select a starting node, mark the starting node as visited and push it into the stack.
2. Explore any one of adjacent nodes of the starting node which are unvisited.
3. Mark the unvisited node as visited and push it into the stack.
4. Repeat this process until all the nodes in the tree or graph are visited.
5. Once all the nodes are visited, then pop all the elements in the stack until the stack becomes empty.

Implementation of DFS in Python

Source Code: DFS in Python

```
1 import sys
2
3 def ret_graph():
4     return {
5         'A': {'B':5.5, 'C':2, 'D':6},
6         'B': {'A':5.5, 'E':3},
7         'C': {'A':2, 'F':2.5},
8         'D': {'A':6, 'F':1.5},
9         'E': {'B':3, 'J':7},
10        'F': {'C':2.5, 'D':1.5, 'K':1.5, 'G':3.5},
11        'G': {'F':3.5, 'I':4},
12        'H': {'J':2},
13        'I': {'G':4, 'J':4},
14        'J': {'H':2, 'I':4},
15        'K': {'F':1.5}
16    }
17
18 start = 'A'
19 dest = 'J'
20 visited = []
21 stack = []
22 graph = ret_graph()
```

```

23 path = []
24
25
26 stack.append(start)
27 visited.append(start)
28 while stack:
29     curr = stack.pop()
30     path.append(curr)
31     for neigh in graph[curr]:
32         if neigh not in visited:
33             visited.append(neigh)
34             stack.append(neigh)
35         if neigh == dest:
36             print("FOUND:", neigh)
37             print(path)
38             sys.exit(0)
39 print("Not found")
40 print(path)
41 """
42 FOUND: J
43 ['A', 'D', 'F', 'G', 'I']
44 """

```

Explanation:

1. First, create a graph in a function.
2. Initialize a starting node and destination node.
3. Create a list for the visited nodes and stack for the next node to be visited.
4. Call the graph function.
5. Initially, the stack is empty. Push the starting node into the stack (stack.append(start)).
6. Mark the starting node as visited (visited.append(start)).
7. Repeat this process until all the neighbours are visited in the stack till the destination node is found.
8. If the destination node is found exit the while loop.
9. If the destination node is not present then "Not found" is printed.
10. Finally, print the path from starting node to the destination node.

3.2.2 骑士周游问题

骑士周游问题是经典问题，用它来说明DFS算法。为了解决骑士周游问题，取一块国际象棋棋盘和一颗骑士棋子（马）。目标是找到一系列走法，使得骑士对棋盘上的每一格刚好都只访问一次。这样的一个移动序列被称为“周游路径”。多年来，骑士周游问题吸引了众多棋手、数学家和计算机科学家。对于 8×8 的棋盘，周游数的上界是 1.305×10^{35} ，但死路更多。很明显，解决这个问题需要聪明人或者强大的计算能力，抑或兼具二者。

尽管人们研究出很多种算法来解决骑士周游问题，但是图搜索算法是其中最好理解和最易编程的一种。我们再一次通过两步来解决这个问题：

- 用图表示骑士在棋盘上的合理走法；
- 使用图算法找到一条长度为 $\text{rows} \times \text{columns}-1$ 的路径，满足图中的每一个顶点都只被访问一次。

1 构建骑士周游图

为了用图表示骑士周游问题，将棋盘上的每一格表示为一个顶点，同时将骑士的每一次合理走法表示为一条边。图1展示了骑士的合理走法以及在图中对应的边。

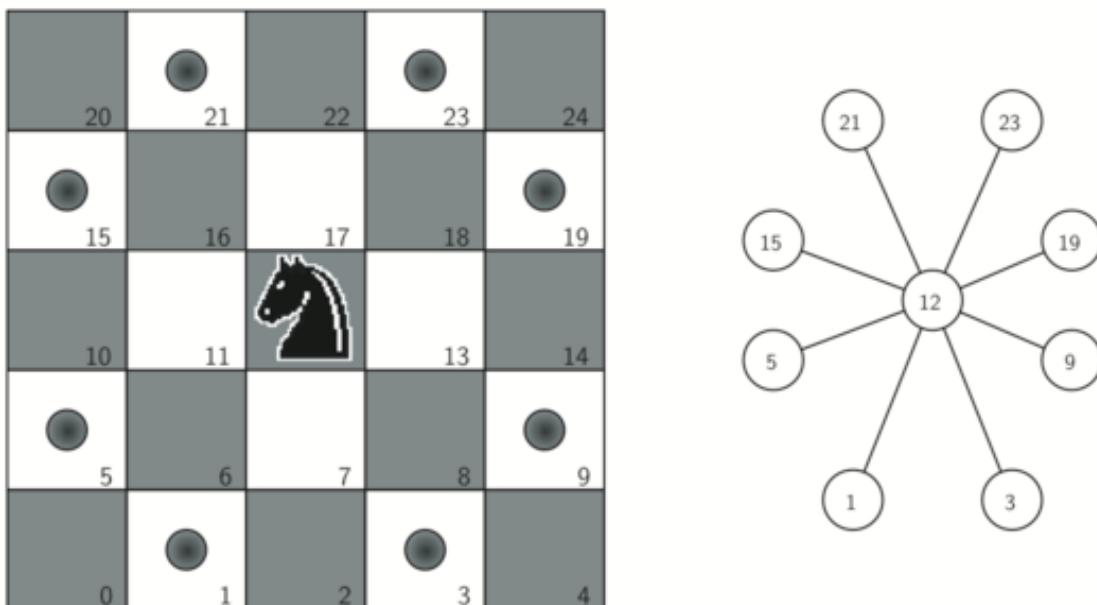


图1 骑士的合理走法以及在图中对应的边

用List 1中的函数来构建 $n \times n$ 棋盘对应的完整图。`knight_graph` 函数将整个棋盘遍历了一遍。当它访问棋盘上的每一格时，都会调用辅助函数 `gen_legal_moves` 来创建一个列表，用于记录从这一格开始的所有合理走法。之后，所有的合理走法都被转换成图中的边。另一个辅助函数 `pos_to_node_id` 将棋盘上的行列位置转换成与图1中顶点编号相似的线性顶点数。

Listing 1 knight_graph函数

```
1  from pythonds3.graphs import Graph
2
3  def knight_graph(board_size):
4      kt_graph = Graph()
5      for row in range(board_size):          #遍历每一行
6          for col in range(board_size):      #遍历行上的每一个格子
7              node_id = pos_to_node_id(row, col, board_size) #把行、列号转为格子ID
8              new_positions = gen_legal_moves(row, col, board_size) #按照 马走日，返回下一步可能位置
9              for row2, col2 in new_positions:
10                  other_node_id = pos_to_node_id(row2, col2, board_size) #下一步的格子ID
11                  kt_graph.add_edge(node_id, other_node_id)
```

```

11         kt_graph.add_edge(node_id, other_node_id) #在骑士周游图中为两个格子加一条
12     边
13
14     return kt_graph
15
16 def pos_to_node_id(x, y, bdSize):
17     return x * bdSize + y

```

在List2中，`gen_legal_moves` 函数接受骑士在棋盘上的位置，并且生成8种可能的走法。并确认走法是合理的。

Listing 2 gen_legal_moves函数

```

1 def gen_legal_moves(row, col, board_size):
2     new_moves = []
3     move_offsets = [                                # 马走日的8种走法
4         (-1, -2),    # left-down-down
5         (-1, 2),    # left-up-up
6         (-2, -1),    # left-left-down
7         (-2, 1),    # left-left-up
8         (1, -2),    # right-down-down
9         (1, 2),    # right-up-up
10        (2, -1),    # right-right-down
11        (2, 1),    # right-right-up
12    ]
13    for r_off, c_off in move_offsets:
14        if (                                # #检查，不能走出棋盘
15            0 <= row + r_off < board_size
16            and 0 <= col + c_off < board_size
17        ):
18            new_moves.append((row + r_off, col + c_off))
19
20
21 # def legal_coord(row, col, board_size):
22 #     return 0 <= row < board_size and 0 <= col < board_size

```

图2展示了在8×8的棋盘上所有合理走法所对应的完整图，其中一共有336条边。注意，与棋盘中间的顶点相比，边缘顶点的连接更少。可以看到，这个图也是非常稀疏的。如果图是完全相连的，那么会有4096条边。由于本图只有336条边，因此邻接矩阵的填充率只有8.2%。还是稀疏图。

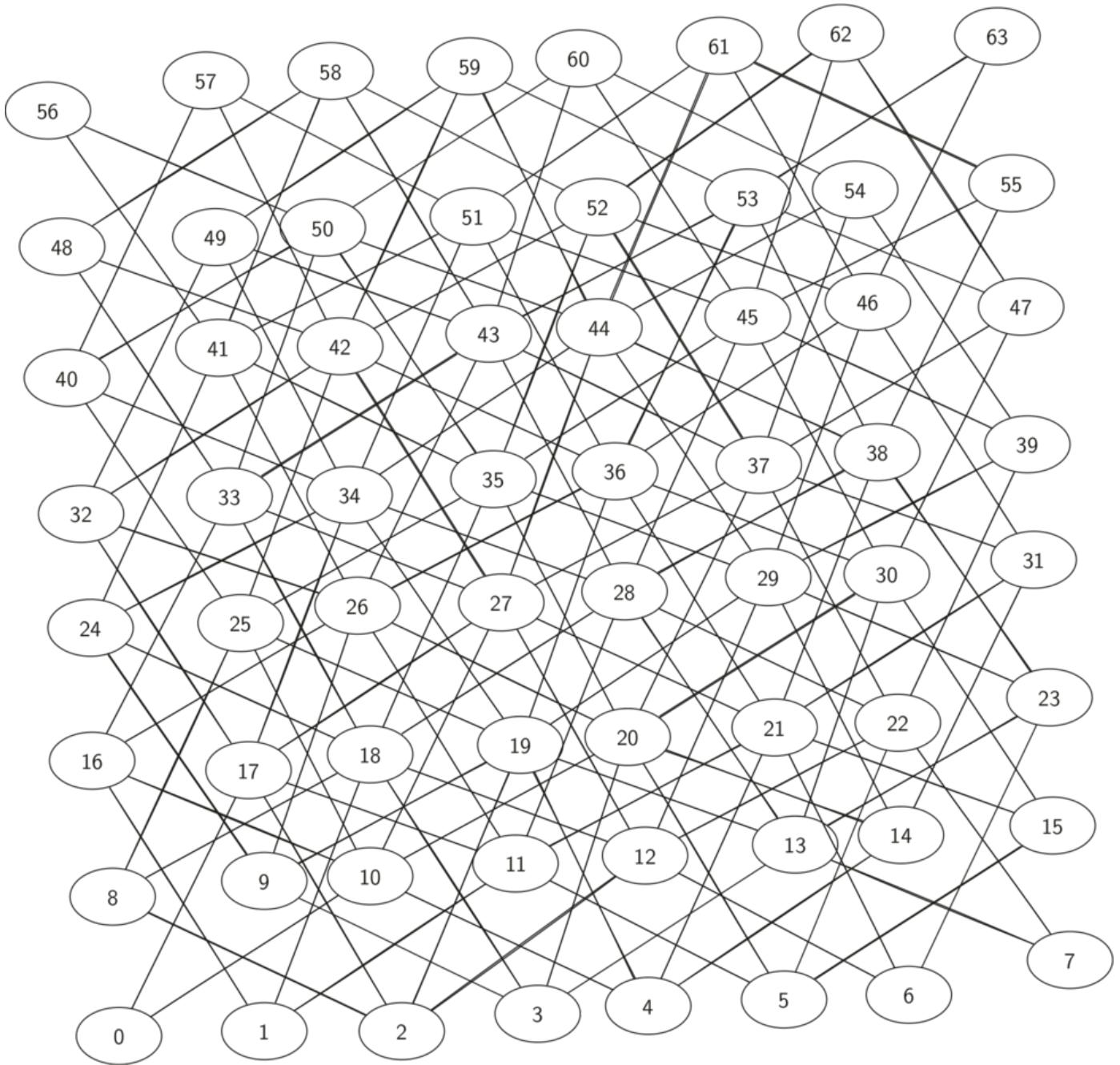


Figure 2: All Legal Moves for a Knight on an 8×8 Chessboard

2 实现骑士周游

用来解决骑士周游问题的搜索算法叫作深度优先搜索（depth first search，以下简称DFS）。与BFS每次构建一层不同，DFS通过尽可能深地探索分支来构建搜索树。下面介绍DFS的两种实现：第1种每个顶点仅访问一次，用于解决骑士周游问题；第2种更通用，它在构建搜索树时允许其中的顶点被多次访问。可以作为其他的图算法的基础。

DFS正是为找到由63条边构成的路径所需的算法。我们会看到，当DFS遇到死路时（无法找到下一个合理走法），它会回退到树中倒数第2深的顶点，以继续移动。

在list 3中，`knight_tour` 函数接受4个参数：n是搜索树的当前深度；path是到目前为止访问过的顶点列表；u是希望在图中访问的顶点；limit是路径上的顶点总数。`knight_tour` 函数是递归的。当被调用时，它首先检查基本情况。如果有一条包含64个顶点的路径，就返回True，以表示找到了一次成功的周游。如果路径不够长，则通过选择一个新的访问顶点并对其递归调用 `knightTour` 来进行更深一层的探索。

DFS也使用颜色来记录已被访问的顶点。未访问的顶点是白色的，已被访问的则是灰色的。如果一个顶点的所有相邻顶点都已被访问过，但是路径长度仍然没有达到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。

List 3 knight_tour函数

```
1 def knight_tour(n, path, u, limit):
2     u.color = "gray"
3     path.append(u)          #当前顶点涂色并加入路径
4     if n < limit:
5         neighbors = ordered_by_avail(u) #对所有的合法移动依次深入
6         #neighbors = sorted(list(u.get_neighbors()))
7         i = 0
8
9         for nbr in neighbors:
10            if nbr.color == "white" and \
11                knight_tour(n + 1, path, nbr, limit):    #选择“白色”未经深入的点，层次加
12                return True
13            else:                                #所有的“下一步”都试了走不通
14                path.pop()                      #回溯，从路径中删除当前顶点
15                u.color = "white"              #当前顶点改回白色
16                return False
17        else:
18            return True
```

第5行是最重要的一行。这一行保证接下来要访问的顶点有最少的合理走法。你可能认为这样做非常影响性能；为什么不选择合理走法最多的顶点呢？

选择合理走法最多的顶点作为下一个访问顶点的问题在于，它会使骑士在周游的前期就访问位于棋盘中间的格子。当这种情况发生时，骑士很容易被困在棋盘的一边，而无法到达另一边的那些没访问过的格子。首先访问合理走法最少的顶点，则可使骑士优先访问棋盘边缘的格子。这样做保证了骑士能够尽早访问难以到达的角落，并且在需要的时候通过中间的格子跨越到棋盘的另一边。我们称利用这类知识来加速算法为启发式技术。人类每天都在使用启发式技术做决定，启发式搜索也经常被用于人工智能领域。本例用到的启发式技术被称作 **Warnsdorff 算法**，以纪念在 1823 年提出该算法的数学家 H. C. Warnsdorff。

我们通过一个例子来看看 `knight_tour` 的运行情况，可以参照图3来追踪搜索的变化。这个例子假设在list 3中第6行对 `getConnections` 方法的调用将顶点按照字母顺序排好。首先调用 `knight_tour(0, path, A, 6)`。

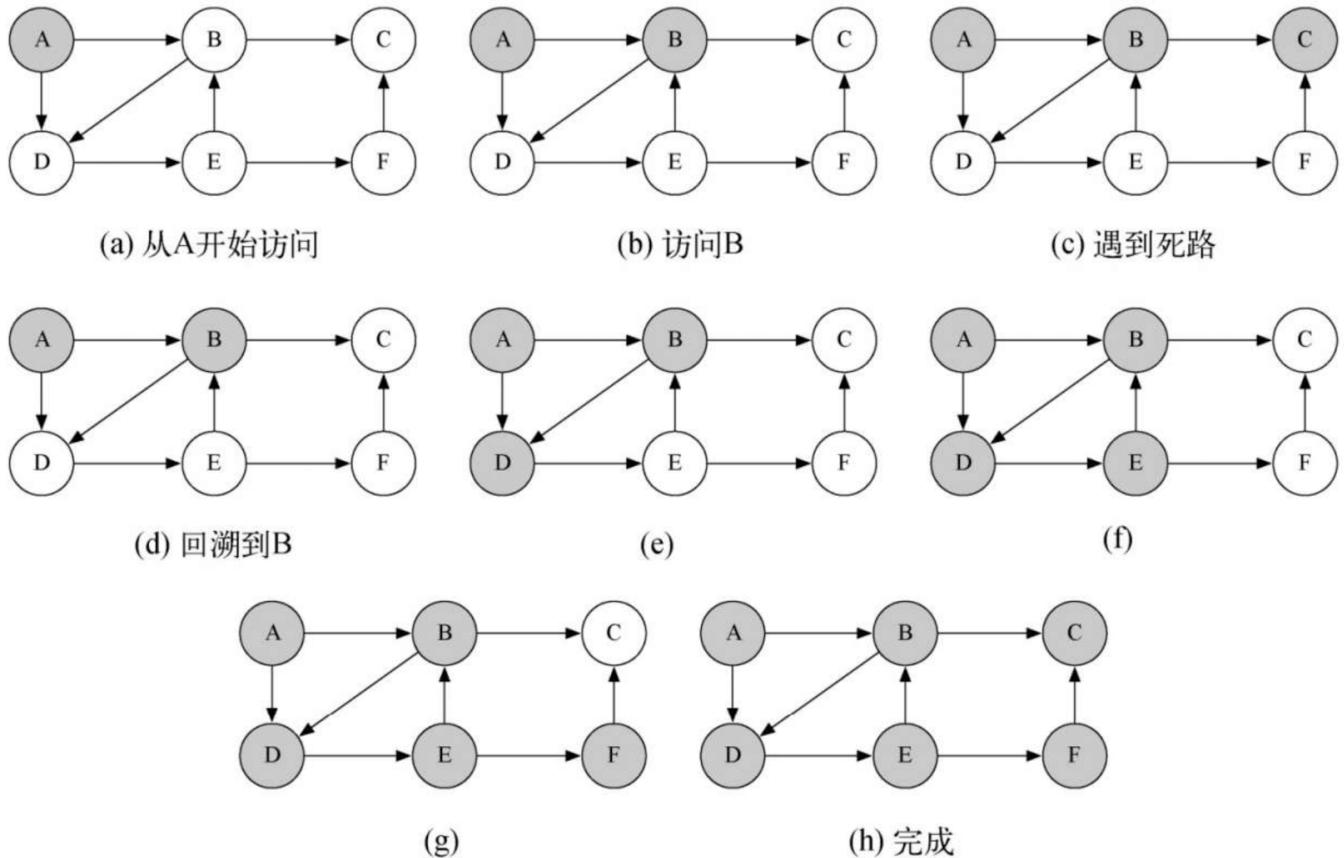


图3 利用knightTour函数找到路径

`knight_tour` 函数从顶点A开始访问。与A相邻的顶点是B和D。按照字母顺序，B在D之前，因此DFS选择B作为下一个要访问的顶点，如图3b所示。对B的访问从递归调用 `knight_tour` 开始。B与C和D相邻，因此 `knight_tour` 接下来会访问C。但是，C没有白色的相邻顶点（如图3c所示），因此是死路。此时，将C的颜色改回白色。

`knight_tour` 的调用返回False，也就是将搜索回溯到顶点B，如图3d所示。接下来要访问的顶点是D，因此 `knight_tour` 进行了一次递归调用访问它。从顶点D开始，`knight_tour` 可以继续进行递归调用，直到再一次访问顶点C。但是，这一次，检验条件 $n < \text{limit}$ 失败了，因此我们知道遍历完了图中所有的顶点。此时返回True，以表明对图进行了一次成功的遍历。当返回列表时，path包含[A, B, D, E, F, C]。其中的顺序就是每个顶点只访问一次所需的顺序。

图4展示了在8×8的棋盘上周游的完整路径。存在多条周游路径，其中有一些是对称的。通过一些修改之后，可以实现循环周游，即起点和终点在同一个位。

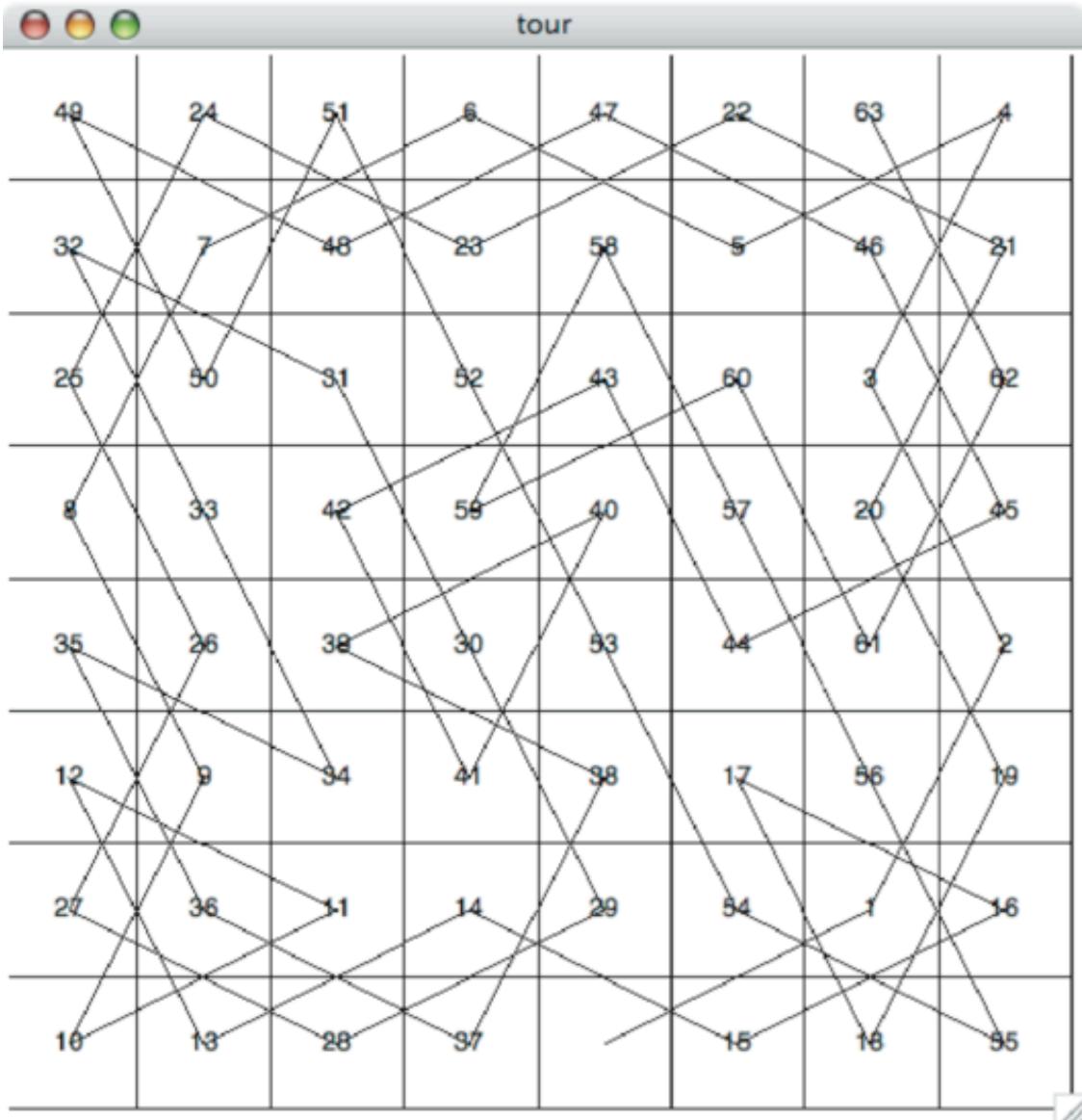


Figure 4: A Complete Tour of the Board

3 分析骑士周游 (Warnsdorff 算法)

在学习深度优先搜索的通用版本之前，我们探索骑士周游问题中的最后一个有趣的话题：性能。具体地说，`knight_tour` 对用于选择下一个访问顶点的方法非常敏感。例如，利用速度正常的计算机，可以在1.5秒之内针对 5×5 的棋盘生成一条周游路径。但是，如果针对 8×8 的棋盘，会怎么样呢？可能需要等待半个小时才能得到结果！

如此耗时的原因在于，目前实现的骑士周游问题算法是一种 $O(k^N)$ 的指数阶算法，其中 N 是棋盘上的格子数， k 是一个较小的常量。图5有助于理解搜索过程。树的根节点代表搜索过程的起点。从起点开始，算法生成并且检测骑士能走的每一步。如前所述，合理走法的数目取决于骑士在棋盘上的位置。若骑士位于四角，只有2种合理走法；若位于与四角相邻的格子中，则有3种合理走法；若在棋盘中央，则有8种合理走法。图5展示了棋盘上的每一格所对应的合理走法数目。在树的下一层，对于骑士当前位置来说，又有2~8种不同的合理走法。待检查位置的数目对应搜索树中的节点数目。

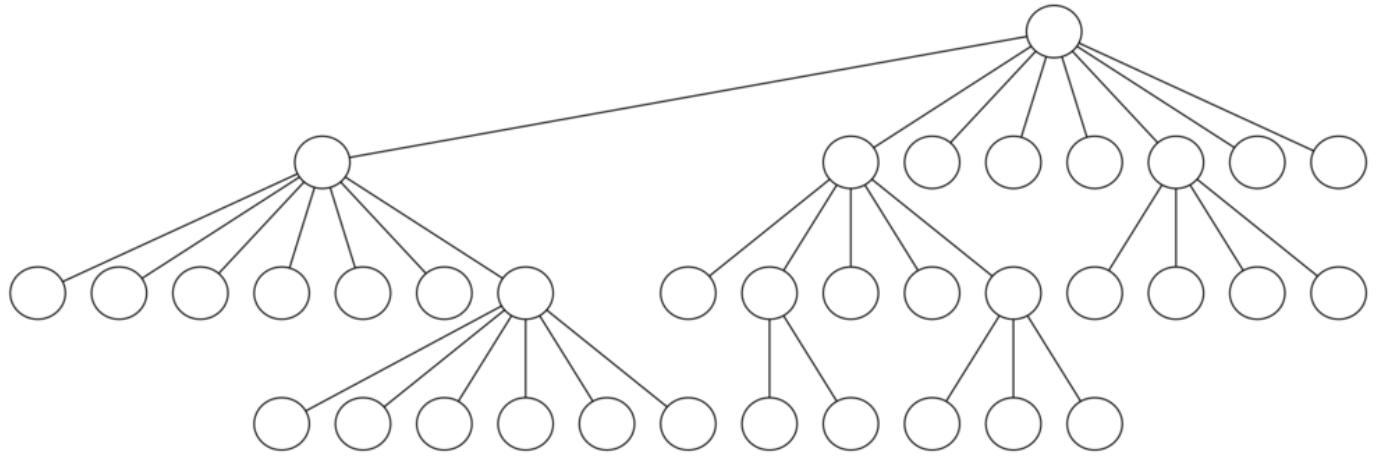


Figure 5: A Search Tree for the Knight's Tour

2	3	4	4	4	4	3	2
3	4	6	6	6	6	4	3
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
3	4	6	6	6	6	4	3
2	3	4	4	4	4	3	2

图6 每个格子对应的合理走法数目

我们已经看到，在高度为N的二叉树中，节点数为 $2^{N+1} - 1$ ；至于子节点可能多达8个而非2个的树，其节点数会更多。由于每一个节点的分支数是可变的，因此可以使用平均分支因子来估计节点数。需要注意的是，这个算法是指数阶算法： $k^{N+1} - 1$ ，其中k是棋盘的平均分支因子。让我们看看它增长得有多快。对于5×5的棋盘，搜索树有25层（若把顶层记为第0层，则N = 24），平均分支因子k = 3.8。因此，搜索树中的节点数是 $3.8^{25} - 1$ 或者 3.12×10^{14} 。对于6×6的棋盘，k = 4.4，搜索树有 1.5×10^{23} 个节点。对于8×8的棋盘，k = 5.25，搜索树有 1.3×10^{46} 个节点。由于这个问题有很多个解，因此不需要访问搜索树中的每一个节点。但是，需要访问的节点的小数部分只是一个常量乘数，它并不能改变该问题的指数特性。

幸运的是，有办法针对8×8的棋盘在1秒内得到一条周游路径。list 4展示了加速搜索过程的代码。在order_by_avail函数中，第10行是最重要的一行。这一行保证接下来要访问的顶点有最少的合理走法。你可能认为这样做非常影响性能；为什么不选择合理走法最多的顶点呢？

List4 选择下一个要访问的顶点至关重要

```

1 def ordered_by_avail(n):
2     res_list = []
3     for v in n.get_neighbors():
4         if v.color == "white":
5             c = 0
6             for w in v.get_neighbors():
7                 if w.color == "white":
8                     c += 1
9             res_list.append((c,v))
10    res_list.sort(key = lambda x: x[0])
11    return [y[1] for y in res_list]

```

选择合理走法最多的顶点作为下一个访问顶点的问题在于，它会使骑士在周游的前期就访问位于棋盘中间的格子。当这种情况发生时，骑士很容易被困在棋盘的一边，而无法到达另一边的那些没访问过的格子。首先访问合理走法最少的顶点，则可使骑士优先访问棋盘边缘的格子。这样做保证了骑士能够尽早访问难以到达的角落，并且在需要的时候通过中间的格子跨越到棋盘的另一边。我们称利用这类知识来加速算法为启发式技术。人类每天都在使用启发式技术做决定，启发式搜索也经常被用于人工智能领域。本例用到的启发式技术被称作Warnsdorff算法，以纪念在1823年提出该算法的数学家H. C. Warnsdorff。

Warnsdorff 算法是一种用于解决骑士周游问题的启发式算法。该算法的主要思想是优先选择下一步可行路径中具有最少可选路径的顶点（优先选择子节点中可行节点少的），从而尽可能地减少搜索空间。

骑士周游程序在，<https://github.com/GMyhf/2024spring-cs201/blob/main/code/KnightTour.py>

```

1 import sys
2
3 class Graph:
4     def __init__(self):
5         self.vertices = {}
6         self.num_vertices = 0
7
8     def add_vertex(self, key):
9         self.num_vertices = self.num_vertices + 1
10        new_ertex = Vertex(key)
11        self.vertices[key] = new_ertex
12        return new_ertex
13
14     def get_vertex(self, n):
15         if n in self.vertices:
16             return self.vertices[n]
17         else:
18             return None
19
20     def __len__(self):
21         return self.num_vertices
22
23     def __contains__(self, n):

```

```

24         return n in self.vertices
25
26     def add_edge(self, f, t, cost=0):
27         if f not in self.vertices:
28             nv = self.add_vertex(f)
29         if t not in self.vertices:
30             nv = self.add_vertex(t)
31         self.vertices[f].add_neighbor(self.vertices[t], cost)
32         #self.vertices[t].add_neighbor(self.vertices[f], cost)
33
34     def getVertices(self):
35         return list(self.vertices.keys())
36
37     def __iter__(self):
38         return iter(self.vertices.values())
39
40
41 class Vertex:
42     def __init__(self, num):
43         self.key = num
44         self.connectedTo = {}
45         self.color = 'white'
46         self.distance = sys.maxsize
47         self.previous = None
48         self.disc = 0
49         self.fin = 0
50
51     def __lt__(self, o):
52         return self.key < o.key
53
54     def add_neighbor(self, nbr, weight=0):
55         self.connectedTo[nbr] = weight
56
57
58     # def setDiscovery(self, dtime):
59     #     self.disc = dtime
60     #
61     # def setFinish(self, ftime):
62     #     self.fin = ftime
63     #
64     # def getFinish(self):
65     #     return self.fin
66     #
67     # def getDiscovery(self):
68     #     return self.disc
69
70     def get_neighbors(self):
71         return self.connectedTo.keys()
72
73     # def getWeight(self, nbr):
74     #     return self.connectedTo[nbr]
75

```

```

76     def __str__(self):
77         return str(self.key) + ":color " + self.color + ":disc " + str(self.disc) +
78         ":fin " + str(
79             self.fin) + ":dist " + str(self.distance) + ":pred \n\t[" +
80             str(self.previous) + "]\\n"
81
82 def knight_graph(board_size):
83     kt_graph = Graph()
84     for row in range(board_size):          #遍历每一行
85         for col in range(board_size):      #遍历行上的每一个格子
86             node_id = pos_to_node_id(row, col, board_size) #把行、列号转为格子ID
87             new_positions = gen_legal_moves(row, col, board_size) #按照 马走日，返回下
一步可能位置
88             for row2, col2 in new_positions:
89                 other_node_id = pos_to_node_id(row2, col2, board_size) #下一步的格子ID
90                 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
91     return kt_graph
92
93 def pos_to_node_id(x, y, bdSize):
94     return x * bdSize + y
95
96 def gen_legal_moves(row, col, board_size):
97     new_moves = []
98     move_offsets = [                      # 马走日的8种走法
99         (-1, -2), # left-down-down
100        (-1, 2), # left-up-up
101        (-2, -1), # left-left-down
102        (-2, 1), # left-left-up
103        (1, -2), # right-down-down
104        (1, 2), # right-up-up
105        (2, -1), # right-right-down
106        (2, 1), # right-right-up
107    ]
108    for r_off, c_off in move_offsets:
109        if (                                # #检查，不能走出棋盘
110            0 <= row + r_off < board_size
111            and 0 <= col + c_off < board_size
112        ):
113            new_moves.append((row + r_off, col + c_off))
114    return new_moves
115
116 # def legal_coord(row, col, board_size):
117 #     return 0 <= row < board_size and 0 <= col < board_size
118
119
120 def knight_tour(n, path, u, limit):
121     u.color = "gray"
122     path.append(u)                      #当前顶点涂色并加入路径
123     if n < limit:

```

```

124         neighbors = ordered_by_avail(u) #对所有的合法移动依次深入
125         #neighbors = sorted(list(u.get_neighbors()))
126         i = 0
127
128         for nbr in neighbors:
129             if nbr.color == "white" and \
130                 knight_tour(n + 1, path, nbr, limit):    #选择“白色”未经深入的点，层次加
131             # 递归深入
132                 return True
133             else:                                #所有的“下一步”都试了走不通
134                 path.pop()                      #回溯，从路径中删除当前顶点
135                 u.color = "white"            #当前顶点改回白色
136                 return False
137             else:
138                 return True
139
140     def ordered_by_avail(n):
141         res_list = []
142         for v in n.get_neighbors():
143             if v.color == "white":
144                 c = 0
145                 for w in v.get_neighbors():
146                     if w.color == "white":
147                         c += 1
148                     res_list.append((c,v))
149             res_list.sort(key = lambda x: x[0])
150             return [y[1] for y in res_list]
151
152     # class DFSGraph(Graph):
153     #     def __init__(self):
154     #         super().__init__()
155     #         self.time = 0          #不是物理世界，而是算法执行步数
156     #
157     #     def dfs(self):
158     #         for vertex in self:
159     #             vertex.color = "white"      #颜色初始化
160     #             vertex.previous = -1
161     #         for vertex in self:        #从每个顶点开始遍历
162     #             if vertex.color == "white":
163     #                 self.dfs_visit(vertex) #第一次运行后还有未包括的顶点
164     #                               # 则建立森林
165     #
166     #     def dfs_visit(self, start_vertex):
167     #         start_vertex.color = "gray"
168     #         self.time = self.time + 1    #记录算法的步骤
169     #         start_vertex.discovery_time = self.time
170     #         for next_vertex in start_vertex.get_neighbors():
171     #             if next_vertex.color == "white":
172     #                 next_vertex.previous = start_vertex
173     #                 self.dfs_visit(next_vertex)    #深度优先递归访问
174     #             start_vertex.color = "black"
175     #             self.time = self.time + 1

```

```

175     #         start_vertex.closing_time = self.time
176
177
178 def main():
179     def NodeToPos(id):
180         return ((id//8, id%8))
181
182     bdSize = int(input()) # 棋盘大小
183     *start_pos, = map(int, input().split()) # 起始位置
184     g = knight_graph(bdSize)
185     start_vertex = g.get_vertex(pos_to_node_id(start_pos[0], start_pos[1], bdSize))
186     if start_vertex is None:
187         print("fail")
188         exit(0)
189
190     tour_path = []
191     done = knight_tour(0, tour_path, start_vertex, bdSize * bdSize - 1)
192     if done:
193         print("success")
194     else:
195         print("fail")
196
197     exit(0)
198
199     # 打印路径
200     cnt = 0
201     for vertex in tour_path:
202         cnt += 1
203         if cnt % bdSize == 0:
204             print()
205         else:
206             print(vertex.key, end=" ")
207             #print(NodeToPos(vertex.key), end=" ") # 打印坐标
208
209     if __name__ == '__main__':
210         main()
211

```

3.3 通用深度优先搜索

骑士周游是深度优先搜索的一种特殊情况，它需要创建没有分支的最深深度优先搜索树。通用的深度优先搜索其实更简单，它的目标是尽可能深地搜索，尽可能多地连接图中的顶点，并且在需要的时候进行分支。

有时候深度优先搜索会创建多棵深度优先搜索树，称之为深度优先森林（**Depth First Forest**）。和宽度优先搜索类似，深度优先搜索也利用前驱连接来构建树。此外，深度优先搜索还会使用Vertex类中的两个额外的实例变量：`发现时间`记录算法在第一次访问顶点时的步数，`结束时间`记录算法在顶点被标记为黑色时的步数。在学习之后会发现，顶点的`发现时间`和`结束时间`提供了一些有趣的特性，后续算法会用到这些特性。

深度优先搜索的实现如代码list 5所示。由于`dfs`函数和`dfsvisit`辅助函数使用一个变量来记录调用`dfsvisit`的时间，因此我们选择将代码作为Graph类的一个子类中的方法来实现。该实现继承Graph类，并且增加了time实例变量，以及`dfs`和`dfsvisit`两个方法。注意第10行，`dfs`方法遍历图中所有的顶点，并对白色顶点调用`dfsvisit`方法。之所以遍历所有的顶点，而不是简单地从一个指定的顶点开始搜索，是因为这样做能够确保深度优先森林中的所有顶点都在考虑范围内，而不会有被遗漏的顶点。`for vertex in self`这条语句可能看上去不太正确，但是此处的self是DFSGraph类的一个实例，遍历一个图实例中的所有顶点其实是一件非常自然的事情。

List5 实现通用深度优先搜索

```
1 # https://github.com/psads/pythonds3
2 from pythonds3.graphs import Graph
3
4 class DFSGraph(Graph):
5     def __init__(self):
6         super().__init__()
7         self.time = 0 #不是物理世界，而是算法执行步数
8
9     def dfs(self):
10        for vertex in self:
11            vertex.color = "white" #颜色初始化
12            vertex.previous = -1
13        for vertex in self: #从每个顶点开始遍历
14            if vertex.color == "white":
15                self.dfs_visit(vertex) #第一次运行后还有未包括的顶点
16                # 则建立森林
17
18    def dfs_visit(self, start_vertex):
19        start_vertex.color = "gray"
20        self.time = self.time + 1 #记录算法的步骤
21        start_vertex.discovery_time = self.time
22        for next_vertex in start_vertex.get_neighbors():
23            if next_vertex.color == "white":
24                next_vertex.previous = start_vertex
25                self.dfs_visit(next_vertex) #深度优先递归访问
26            start_vertex.color = "black"
27        self.time = self.time + 1
28        start_vertex.closing_time = self.time
```

在接下来的两个例子中，我们会看到为何记录深度优先森林十分重要。

从`start_vertex`开始，`dfs_visit`方法尽可能深地探索所有相邻的白色顶点。如果仔细观察`df_svisit`的代码并且将其与bfs比较，应该注意到二者几乎一样，除了内部for循环的最后一行，`dfs_visit`通过递归地调用自己来继续进行下一层的搜索，bfs则将顶点添加到队列中，以供后续搜索。有趣的是，bfs使用队列，`dfsvisit`则使用栈。我们没有在代码中看到栈，但是它其实隐式地存在于`dfs_visit`的递归调用中。

图7展示了在小型图上应用深度优先搜索算法的过程。图中，虚线表示被检查过的边，但是其一端的顶点已经被添加到深度优先搜索树中。在代码中，这是通过检查另一端的顶点是否不为白色来完成的。

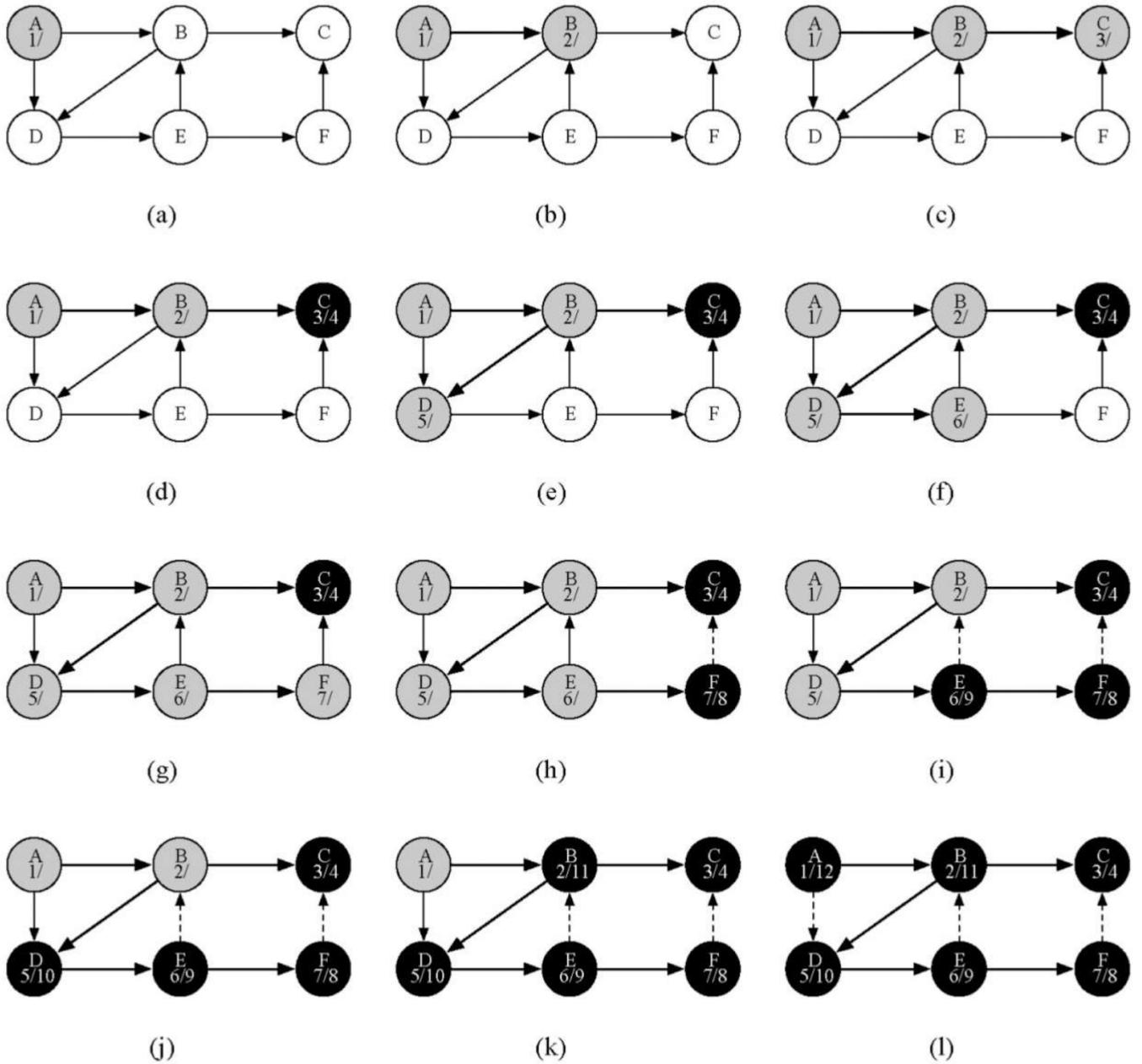


图7 构建深度优先搜索树

搜索从图中的顶点A开始。由于所有顶点一开始都是白色的，因此算法会访问A。访问顶点的第一步是将其颜色设置为灰色，以表明正在访问该顶点，并将其发现时间为1。由于A有两个相邻顶点（B和D），因此它们都需要被访问。我们按照字母顺序来访问顶点。

接下来访问顶点B，将它的颜色设置为灰色，并把发现时间设置为2。B也与两个顶点（C和D）相邻，因此根据字母顺序访问C。

访问C时，搜索到达某个分支的终点。在将C标为灰色并且把发现时间设置为3之后，算法发现C没有相邻顶点。这意味着对C的探索完成，因此将它标为黑色，并将完成时间设置为4。图7d展示了搜索至这一步时的状态。

由于C是一个分支的终点，因此需要返回到B，并且继续探索其余的相邻顶点。唯一的待探索顶点就是D，它把搜索引到E。E有两个相邻顶点，即B和F。正常情况下，应该按照字母顺序来访问这两个顶点，但是由于B已经被标记为灰色，因此算法自知不应该访问B，因为如果这么做就会陷入死循环。因此，探索过程跳过B，继续访问F。

F只有C这一个相邻顶点，但是C已经被标记为黑色，因此没有后续顶点需要探索，也即到达另一个分支的终点。从此时起，算法一路回溯到起点，同时为各个顶点设置完成时间并将它们标记为黑色，如图7h~图7l所示。

每个顶点的发现时间和结束时间都体现了括号特性，这意味着深度优先搜索树中的任一节点的子节点都有比该节点更晚的发现时间和更早的结束时间。图8展示了通过深度优先搜索算法构建的树。

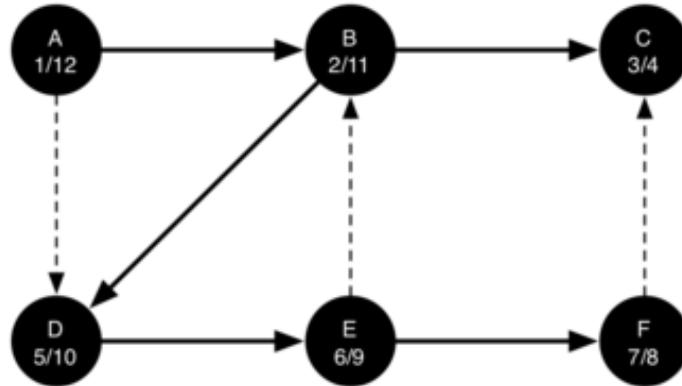


图8 最终的深度优先搜索

分析深度优先搜索

一般来说，深度优先搜索的运行时间如下。在 `dfs` 函数中有两个循环，每个都是 $|V|$ 次，所以是 $O(|V|)$ ，这是由于它们针对图中的每个顶点都只执行一次。在 `dfs_visit` 中，循环针对当前顶点的邻接表中的每一条边都执行一次，且仅在顶点是白色时被递归调用，因此循环最多会对图中的每一条边执行一次，也就是 $O(|E|)$ 。因此，深度优先搜索算法的时间复杂度是 $O(|V| + |E|)$ ，与 BFS一样。

*深度优先搜索树（林）的性质

- 顶点大小的定义：**对于图 G 中的任意两个顶点 u 和 v ，当且仅当顶点 u 的结束时间 (fin) 小于顶点 v 的结束时间 (fin) 时，称顶点 u 小于顶点 v ，即符号化表示为： $\forall u, v \in G; u < v \equiv u.fin < v.fin$ 。

Page 608, introduction to algorithms 3rd Edition

Corollary 22.8 (Nesting of descendants' intervals)

Vertex v is a proper descendant of vertex u in the depth-first forest for a (directed or undirected) graph G if and only if $u.d < v.d < v.f < u.f$.

- 子树大小的定义：**对于深度优先搜索树（林）中的任意两棵不相交的子树 t_1 和 t_2 ，当且仅当 t_1 中的任意节点的结束时间都早于 t_2 中的任意节点的开始时间时，称 t_1 小于 t_2 ，即符号化表示为： $t_1 < t_2 \equiv \forall u \in t_1, v \in t_2; u < v$ 。
- 节点间的互斥关系：**如果 t_1 小于 t_2 ，且节点 u 属于 t_1 ，节点 v 属于 t_2 ，则 $u \rightarrow v$ 不存在。
- 节点间的关系限定：**如果顶点 u 小于顶点 v ，并且它们在同一棵树中，则只有两种可能情况：
 - 顶点 v 是顶点 u 的祖先。
 - 顶点 u 和 v 具有共同的祖先 t 。其中，顶点 u 属于 t 的子树 t_1 ，顶点 v 属于 t 的子树 t_2 。这种情况下， t_1 的结束时间早于 t_2 的开始时间。

3.4 编程题目

28046: 词梯

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

28050: 骑士周游

<http://cs101.openjudge.cn/practice/28050/>

04123: 马走日

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

02754: 八皇后

dfs and similar, <http://cs101.openjudge.cn/practice/02754>

02698: 八皇后问题解输出

dfs and similar, <http://cs101.openjudge.cn/practice/02698>

sy321: 迷宫最短路径

<https://sunnywhy.com/sfbj/8/2/321>

现有一个 $n*m$ 大小的迷宫，其中 1 表示不可通过的墙壁，0 表示平地。每次移动只能向上下左右移动一格，且只能移动到平地上。假设左上角坐标是(1,1)，行数增加的方向为增长的方向，列数增加的方向为增长的方向，求从迷宫左上角到右下角的最少步数的路径。

输入

第一行两个整数 n, m ($2 \leq n \leq 100, 2 \leq m \leq 100$)，分别表示迷宫的行数和列数；

接下来 n 行，每行 m 个整数（值为 0 或 1），表示迷宫。

输出

从左上角的坐标开始，输出若干行（每行两个整数，表示一个坐标），直到右下角的坐标。

数据保证最少步数的路径存在且唯一。

样例1

输入

```
1 | 3 3  
2 | 0 1 0  
3 | 0 0 0  
4 | 0 1 0
```

输出

```
1 | 1 1  
2 | 2 1  
3 | 2 2  
4 | 2 3  
5 | 3 3
```

解释

假设左上角坐标是(1,)，行数增加的方向为增长的方向，列数增加的方向为增长的方向。

可以得到从左上角到右下角的最少步数的路径为：(1,1)=>(2,1)=>(2,2)=>(2,3)=>(3,3)。

in_queue 数组，结点是否已入过队

```
1 | from collections import deque  
2 |  
3 | MAX_DIRECTIONS = 4  
4 | dx = [0, 0, 1, -1]  
5 | dy = [1, -1, 0, 0]  
6 |  
7 | def is_valid_move(x, y):  
8 |     return 0 <= x < n and 0 <= y < m and maze[x][y] == 0 and not in_queue[x][y]  
9 |  
10 | def bfs(start_x, start_y):  
11 |     queue = deque()  
12 |     queue.append((start_x, start_y))  
13 |     in_queue[start_x][start_y] = True  
14 |     while queue:  
15 |         x, y = queue.popleft()  
16 |         if x == n - 1 and y == m - 1:  
17 |             return  
18 |         for i in range(MAX_DIRECTIONS):  
19 |             next_x = x + dx[i]  
20 |             next_y = y + dy[i]  
21 |             if is_valid_move(next_x, next_y):  
22 |                 prev[next_x][next_y] = (x, y)  
23 |                 in_queue[next_x][next_y] = True  
24 |                 queue.append((next_x, next_y))  
25 |  
26 | def print_path(pos):  
27 |     prev_position = prev[pos[0]][pos[1]]  
28 |     if prev_position == (-1, -1):
```

```

29         print(pos[0] + 1, pos[1] + 1)
30     return
31     print_path(prev_position)
32     print(pos[0] + 1, pos[1] + 1)
33
34 n, m = map(int, input().split())
35 maze = [list(map(int, input().split())) for _ in range(n)]
36
37 in_queue = [[False] * m for _ in range(n)]
38 prev = [[(-1, -1)] * m for _ in range(n)]
39
40 bfs(0, 0)
41 print_path((n - 1, m - 1))
42

```

sy380: 无向图的连通块 简单

<https://sunnywhy.com/sfbj/10/3/380>

现有一个共n个顶点、m条边的无向图（假设顶点编号为从0到n-1），求图中的连通块个数。

输入

第一行两个整数n、m ($1 \leq n \leq 100, 0 \leq m \leq \frac{n(n-1)}{2}$)，分别表示顶点数和边数；

接下来m行，每行两个整数u、v ($0 \leq u \leq n - 1, 0 \leq v \leq n - 1, u \neq v$)，表示一条边的两个端点的编号。
数据保证不会有重边。

输出

输出一个整数，表示图中的连通块个数。

样例1

输入

```

1 6 5
2 0 1
3 0 3
4 3 5
5 2 4
6 1 5

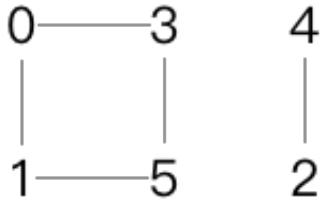
```

输出

1	2
---	---

解释

对应的无向图如下图所示，共有两个连通块。



要解决这个问题，我们可以使用深度优先搜索（DFS）或广度优先搜索（BFS）来找出图中的所有连通块。我们首先创建一个邻接列表来表示图，然后对每个未访问的节点进行DFS或BFS，每次搜索都会找到一个连通块。搜索过程中，我们需要标记已访问的节点，以防止重复访问。

以下是使用DFS的Python代码：

```

1 def dfs(node, visited, adjacency_list):
2     visited[node] = True
3     for neighbor in adjacency_list[node]:
4         if not visited[neighbor]:
5             dfs(neighbor, visited, adjacency_list)
6
7 n, m = map(int, input().split())
8 adjacency_list = [[] for _ in range(n)]
9 for _ in range(m):
10    u, v = map(int, input().split())
11    adjacency_list[u].append(v)
12    adjacency_list[v].append(u)
13
14 visited = [False] * n
15 connected_components = 0
16 for i in range(n):
17     if not visited[i]:
18         dfs(i, visited, adjacency_list)
19         connected_components += 1
20
21 print(connected_components)

```

在这段代码中，我们首先读取顶点数和边数，然后读取每条边的信息并在邻接列表中添加相应的边。然后我们初始化一个布尔数组 `visited` 来跟踪已访问的节点，并初始化连通块的数量为0。接着，我们遍历每个节点，如果它还未被访问，我们就从它开始进行DFS，并将连通块的数量加1。最后，我们输出连通块的数量。

sy381: 无向连通图 简单

<https://sunnywhy.com/sfbj/10/3/381>

现有一个共 n 个顶点、 m 条边的无向图（假设顶点编号为从0到 $n-1$ ），判断其是否是连通图。

输入

第一行两个整数 n 、 m ($1 \leq n \leq 100, 0 \leq m \leq \frac{n(n-1)}{2}$)，分别表示顶点数和边数；

接下来m行，每行两个整数u、v ($0 \leq u \leq n - 1, 0 \leq v \leq n - 1, u \neq v$)，表示一条边的起点和终点的编号。数据保证不会有重边。

输出

如果是连通图，那么输出 yes，否则输出 no。

样例1

输入

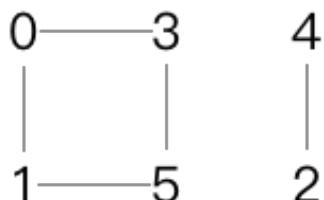
1	6 5
2	0 1
3	0 3
4	3 5
5	2 4
6	1 5

输出

1 | No

解释

对应的无向图如下图所示，不是连通图。



样例2

输入

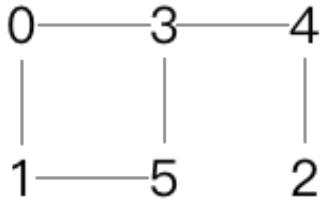
1	6 6
2	0 1
3	0 3
4	3 5
5	2 4
6	1 5
7	3 4

输出

1 | Yes

解释

对应的无向图如下图所示，是连通图。



要判断一个图是否是连通图，我们可以使用深度优先搜索（DFS）或广度优先搜索（BFS）来遍历图。如果在遍历结束后，所有的节点都被访问过，那么这个图就是连通图。否则，就不是连通图。

以下是使用DFS的Python代码：

```

1 def dfs(node, visited, adjacency_list):
2     visited[node] = True
3     for neighbor in adjacency_list[node]:
4         if not visited[neighbor]:
5             dfs(neighbor, visited, adjacency_list)
6
7 n, m = map(int, input().split())
8 adjacency_list = [[] for _ in range(n)]
9 for _ in range(m):
10    u, v = map(int, input().split())
11    adjacency_list[u].append(v)
12    adjacency_list[v].append(u)
13
14 visited = [False] * n
15 dfs(0, visited, adjacency_list)
16
17 if all(visited):
18     print("Yes")
19 else:
20     print("No")

```

在这段代码中，我们首先读取顶点数和边数，然后读取每条边的信息并在邻接列表中添加相应的边。然后我们初始化一个布尔数组 `visited` 来跟踪已访问的节点，并从第一个节点开始进行DFS。最后，我们检查是否所有的节点都被访问过，如果是，那么输出 `yes`，否则输出 `no`。

sy382: 有向图判环 中等

<https://sunnywhy.com/sfbj/10/3/382>

现有一个共n个顶点、m条边的有向图（假设顶点编号为从0到n-1），如果从图中一个顶点出发，沿着图中的有向边前进，最后能回到这个顶点，那么就称其为图中的一个环。判断图中是否有环。

输入

第一行两个整数n、m ($1 \leq n \leq 100, 0 \leq m \leq n(n - 1)$)，分别表示顶点数和边数；

接下来m行，每行两个整数u、v ($0 \leq u \leq n - 1, 0 \leq v \leq n - 1, u \neq v$)，表示一条边的起点和终点的编号。数据保证不会有重边。

输出

如果图中有环，那么输出 yes，否则输出 no。

样例1

输入

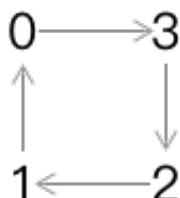
1	4 4
2	1 0
3	0 3
4	3 2
5	2 1

输出

1	Yes
---	-----

解释

对应的有向图如下图所示，存在 $0 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 0$ 的环。



样例2

输入

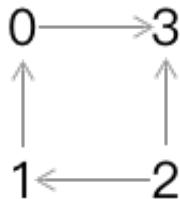
1	4 4
2	1 0
3	0 3
4	2 3
5	2 1

输出

1	No
---	----

解释

对应的有向图如下图所示，图中不存在环。



在这个问题中，需要检查给定的有向图是否包含一个环。可以使用深度优先搜索（DFS）来解决这个问题。在DFS中，从一个节点开始，然后访问它的每一个邻居。如果在访问过程中，遇到了一个已经在当前路径中的节点，那么就存在一个环。可以使用一个颜色数组来跟踪每个节点的状态：未访问（0），正在访问（1），已访问（2）。

以下是解决这个问题的Python代码：

```

1 def has_cycle(n, edges):
2     graph = [[] for _ in range(n)]
3     for u, v in edges:
4         graph[u].append(v)
5
6     color = [0] * n
7
8     def dfs(node):
9         if color[node] == 1:
10             return True
11         if color[node] == 2:
12             return False
13
14         color[node] = 1
15         for neighbor in graph[node]:
16             if dfs(neighbor):
17                 return True
18         color[node] = 2
19         return False
20
21     for i in range(n):
22         if dfs(i):
23             return "Yes"
24     return "No"
25
26 # 接收数据
27 n, m = map(int, input().split())
28 edges = []
29 for _ in range(m):
30     u, v = map(int, input().split())
31     edges.append((u, v))
32
33 # 调用函数
34 print(has_cycle(n, edges))

```

在这个函数中，我们首先构建了一个邻接列表来表示图。然后，我们对每个节点执行深度优先搜索。如果在搜索过程中，我们遇到了一个正在访问的节点，那么就存在一个环。如果我们遍历完所有的节点都没有找到环，那么就返回"No"。

sy383: 最大权值连通块 中等

<https://sunnywhy.com/sfbj/10/3/383>

现有一个共个顶点、条边的无向图（假设顶点编号为从 0 到 $n-1$ ），每个顶点有各自的权值。我们把一个连通块中所有顶点的权值之和称为这个连通块的权值。求图中所有连通块的最大权值。

输入

第一行两个整数 n 、 m ($1 \leq n \leq 100, 0 \leq m \leq \frac{n(n-1)}{2}$)，分别表示顶点数和边数；

第二行个用空格隔开的正整数（每个正整数不超过 100），表示个顶点的权值。

接下来 m 行，每行两个整数 u 、 v ($0 \leq u \leq n-1, 0 \leq v \leq n-1, u \neq v$)，表示一条边的起点和终点的编号。数据保证不会有重边。

输出

输出一个整数，表示连通块的最大权值。

样例1

输入

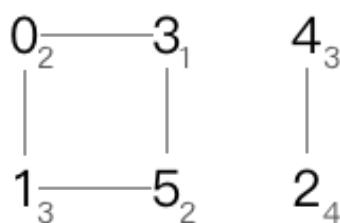
```
1 6 5
2 2 3 4 1 3 2
3 0 1
4 0 3
5 3 5
6 2 4
7 1 5
```

输出

```
1 | 8
```

解释

对应的无向图如下图所示，左边连通块的权值为，右边连通块的权值为，因此最大权值为 8。



需要找到给定无向图中所有连通块的最大权值。使用深度优先搜索（DFS）来解决这个问题。在DFS中，从一个节点开始，然后访问它的每一个邻居。可以使用一个visited数组来跟踪每个节点是否已经被访问过。对于每个连通块，可以计算其权值之和，并更新最大权值。

以下是解决这个问题的Python代码：

```
1 def max_weight(n, m, weights, edges):
2     graph = [[] for _ in range(n)]
3     for u, v in edges:
4         graph[u].append(v)
5         graph[v].append(u)
6
7     visited = [False] * n
8     max_weight = 0
9
10    def dfs(node):
11        visited[node] = True
12        total_weight = weights[node]
13        for neighbor in graph[node]:
14            if not visited[neighbor]:
15                total_weight += dfs(neighbor)
16        return total_weight
17
18    for i in range(n):
19        if not visited[i]:
20            max_weight = max(max_weight, dfs(i))
21
22    return max_weight
23
24 # 接收数据
25 n, m = map(int, input().split())
26 weights = list(map(int, input().split()))
27 edges = []
28 for _ in range(m):
29     u, v = map(int, input().split())
30     edges.append((u, v))
31
32 # 调用函数
33 print(max_weight(n, m, weights, edges))
```

在这段代码中，首先通过 `input()` 函数接收用户输入的顶点数 `n`、边数 `m` 和每个顶点的权值，然后在一个循环中接收每条边的起点和终点，并将它们添加到 `edges` 列表中。然后，我们调用 `max_weight` 函数并打印结果。

sy384: 无向图的顶点层号

<https://sunnywhy.com/sfbj/10/3/384>

现有一个共n个顶点、m条边的无向连通图（假设顶点编号为从0到n-1）。我们称从s号顶点出发到达其他顶点经过的最小边数称为各顶点的层号。求图中所有顶点的层号。

输入

第一行三个整数n、m、s ($1 \leq n \leq 100, 0 \leq m \leq \frac{n(n-1)}{2}, 0 \leq s \leq n - 1$)，分别表示顶点数、边数、起始顶点编号；

接下来m行，每行两个整数u、v ($0 \leq u \leq n - 1, 0 \leq v \leq n - 1, u \neq v$)，表示一条边的起点和终点的编号。数据保证不会有重边。

输出

输出n个整数，分别为编号从0到n-1的顶点的层号。整数之间用空格隔开，行末不允许有多余的空格。

样例1

输入

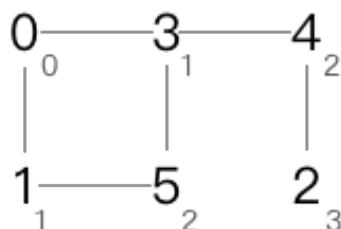
1	6 6 0
2	0 1
3	0 3
4	3 5
5	2 4
6	1 5
7	3 4

输出

1	0 1 3 1 2 2
---	-------------

解释

对应的无向图和顶点层号如下图所示。



需要找到从给定的起始顶点到图中所有其他顶点的最短路径长度，这也被称为顶点的层号。可以使用广度优先搜索(BFS)来解决这个问题。在BFS中，从起始节点开始，然后访问它的所有邻居，然后再访问这些邻居的邻居，依此类推。我们可以使用一个队列来跟踪待访问的节点，并使用一个距离数组来记录从起始节点到每个节点的最短距离。

以下是解决这个问题的Python代码：

```

1  from collections import deque
2
3  def bfs(n, m, s, edges):
4      graph = [[] for _ in range(n)]
5      for u, v in edges:
6          graph[u].append(v)
7          graph[v].append(u)
8
9      distance = [-1] * n
10     distance[s] = 0
11
12     queue = deque([s])
13     while queue:
14         node = queue.popleft()
15         for neighbor in graph[node]:
16             if distance[neighbor] == -1:
17                 distance[neighbor] = distance[node] + 1
18                 queue.append(neighbor)
19
20     return distance
21
22 # 接收数据
23 n, m, s = map(int, input().split())
24 edges = []
25 for _ in range(m):
26     u, v = map(int, input().split())
27     edges.append((u, v))
28
29 # 调用函数
30 distances = bfs(n, m, s, edges)
31 print(' '.join(map(str, distances)))

```

在这段代码中，我们首先通过 `input()` 函数接收用户输入的顶点数 `n`、边数 `m` 和起始顶点 `s`，然后在一个循环中接收每条边的起点和终点，并将它们添加到 `edges` 列表中。然后，我们调用 `bfs` 函数并打印结果。

sy385: 受限层号的顶点数 中等

<https://sunnywhy.com/sfbj/10/3/385>

现有一个共 n 个顶点、 m 条边的有向图（假设顶点编号为从 0 到 $n-1$ ）。我们称从 s 号顶点出发到达其他顶点经过的最小边数称为各顶点的层号。求层号不超过的顶点个数。

输入

第一行四个整数 n 、 m 、 s 、 k ($1 \leq n \leq 100, 0 \leq m \leq \frac{n(n-1)}{2}, 0 \leq s \leq n-1, 0 \leq k \leq 100$)，分别表示顶点数、边数、起始顶点编号；

接下来 m 行，每行两个整数 u 、 v ($0 \leq u \leq n-1, 0 \leq v \leq n-1, u \neq v$)，表示一条边的起点和终点的编号。数据保证不会有重边。

输出

输出一个整数，表示层号不超过的顶点个数。

样例1

输入

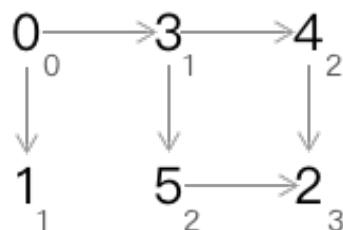
```
1 | 6 6 0 2
2 | 0 1
3 | 0 3
4 | 3 5
5 | 4 2
6 | 3 4
7 | 5 2
```

输出

```
1 | 5
```

解释

对应的有向图和顶点层号如下图所示，层号不超过 2 的顶点有 5 个。



需要找到从给定的起始顶点到图中所有其他顶点的最短路径长度（也被称为顶点的层号），并计算层号不超过k的顶点个数。可以使用广度优先搜索（BFS）来解决这个问题。在BFS中，从起始节点开始，然后访问它的所有邻居，然后再访问这些邻居的邻居，依此类推。可以使用一个队列来跟踪待访问的节点，并使用一个距离数组来记录从起始节点到每个节点的最短距离。

以下是解决这个问题的Python代码：

```
1 | from collections import deque
2 |
3 | def bfs(n, m, s, k, edges):
4 |     graph = [[] for _ in range(n)]
5 |     for u, v in edges:
6 |         graph[u].append(v) # 只按照输入的方向添加边
7 |
8 |     distance = [-1] * n
9 |     distance[s] = 0
10 |
```

```

11     queue = deque([s])
12     while queue:
13         node = queue.popleft()
14         for neighbor in graph[node]:
15             if distance[neighbor] == -1:
16                 distance[neighbor] = distance[node] + 1
17                 queue.append(neighbor)
18
19     return sum(1 for d in distance if d <= k and d != -1)
20
21 # 接收数据
22 n, m, s, k = map(int, input().split())
23 edges = []
24 for _ in range(m):
25     u, v = map(int, input().split())
26     edges.append((u, v))
27
28 # 调用函数
29 count = bfs(n, m, s, k, edges)
30 print(count)

```

在这段代码中，首先通过 `input()` 函数接收用户输入的顶点数 `n`、边数 `m`、起始顶点 `s` 和层号上限 `k`，然后在一个循环中接收每条边的起点和终点，并将它们添加到 `edges` 列表中。然后，调用 `bfs` 函数并打印结果。

3.5 笔试题目（类图 + dfs）

1 数算B-2021笔试最后一个算法题目（8分）

图的深度优先周游算法实现的迷宫探索。图采用邻接表表示，给出了Graph类和Vertex类的基本定义。从题面看基本上与书上提供的G Graph的ADT实现一样，稍作更改。但是这样不一定得到的是最短路径？

在走迷宫时，通常会使用深度优先搜索（DFS）或广度优先搜索（BFS），具体选择哪种搜索算法取决于问题的要求和对性能的考虑。

1. DFS（深度优先搜索）：

- DFS 适合于解决能够表示成树形结构的问题，包括迷宫问题。DFS 会尽可能地沿着迷宫的一条路径向前探索，直到不能再前进为止，然后回溯到前一个位置，再尝试其他路径。在解决迷宫问题时，DFS 可以帮助我们快速地找到一条通路（如果存在），但不一定能够找到最短路径。
- DFS 的优点是实现简单，不需要额外的数据结构来保存搜索状态。

2. BFS（广度优先搜索）：

- BFS 适合于解决需要找到最短路径的问题，因为它会逐层地搜索所有可能的路径，从起点开始，一层一层地向外扩展。在解决迷宫问题时，BFS 可以找到最短路径（如果存在），但相对于 DFS 来说，它可能需要更多的空间来存储搜索过程中的状态信息。
- BFS 的优点是能够找到最短路径，并且保证在找到解之前不会漏掉任何可能的路径。

综上所述，如果你只需要找到一条通路而不关心路径的长度，可以选择使用 DFS。但如果你需要找到最短路径，或者需要在可能的路径中选择最优解，那么应该选择 BFS。

2. 阅读下列程序，完成图的深度优先周游算法实现的迷宫探索。已知图采用邻接表表示，
Graph 类和 Vertex 类基本定义如下：

```
class Graph:  
    def __init__(self):  
        pass  
    def addVertex(self, key, label): # 添加节点, id 为 key, 附带数据 label  
    def getVertex(self, key): # 返回 id 为 key 的节点  
    def __contains__(self, key): # 判断 key 节点是否在图中  
    def addEdge(self, f, t, cost=0): # 添加从节点 id==f 到 id==t 的边  
    def getVertices(self): # 返回所有的节点 key  
    def __iter__(self): # 迭代每一个节点对象  
  
class Vertex:  
    def __init__(self, key, label=None): # 缺省颜色为"white"  
    def addNeighbor(self, nbr, weight=0): # 添加到节点 nbr 的边  
    def setColor(self, color): # 设置节点颜色标记  
    def getColor(self): # 返回节点颜色标记  
    def getConnections(self): # 返回节点的所有邻接节点列表  
    def getId(self): # 返回节点的 id  
    def getLabel(self): # 返回节点的附带数据 label  
  
mazelist = [  
    "+++++++",  
    "+ + ++ +",  
    "E + ++++++",  
    "+ + ++ +++ ++",  
    "+ + + + + + +",  
    "+ + + + + + +",  
    "+ + + + + + +",  
    "+ + + + + + +",  
    "+ + + + + + +",  
    "+ + + + S + + +",  
    "+ + + + + + + +",  
    "+++++++",  
]  
  
def mazeGraph(mlist, rows, cols): # 从 mlist 创建图, 迷宫有 rows 行 cols 列  
    mGraph = Graph()  
    vstart = None  
    for row in range(rows):  
        for col in range(cols):  
            if mlist[row][col] != "+":  
                mGraph.addVertex((row, col), mlist[row][col])  
            if mlist[row][col] == "S":  
                vstart = _____ (1 分)  
                break  
        if vstart: break  
    return mGraph
```

8 / 9

```

        for v in mGraph:
            row, col = v.getId()
            for i in [(-1, 0), (1, 0), (0, -1), (0, +1)]:
                if (0 <= row + i[0] < rows) and (0 <= col + i[1] < cols):
                    if (row + i[0], col + i[1]) in mGraph:
                        mGraph.addEdge(_____(1分))

    return mGraph, vstart # 返回图对象, 和开始节点

def searchMaze(path, vcurrent): # 从 vcurrent 节点开始 DFS 搜索迷宫
    vcurrent.setColor('gray')
    path.append(vcurrent.getId())
    if vcurrent.getLabel() != "E":
        done = False
        for nbr in _____(2分):
            if nbr.getColor() == "white":
                done = searchMaze(_____(2分))
            if done:
                break
        if not done:
            _____(2分)
            vcurrent.setColor("white")
    else:
        done = True
    return done # 返回是否成功找到通路

g, vstart = mazeGraph(mazelist, len(mazelist), len(mazelist[0]))
path = []
searchMaze(path, vstart)
print(path)

```

2 参考答案

下面是我给的答案, 请同学自己完成上面算法填空, 然后来对照, 看答案是否正确。

阅读下列程序, 完成图的深度优先周游算法实现的迷宫探索。已知图采用邻接表表示, Graph 类和 Vertex 类基本定义如下:

```

1 class Graph:
2     def __init__(self):
3         self.vertices = {}
4
5     def addVertex(self, key, label): #添加节点, id 为key, 附带数据 label
6         self.vertices[key] = Vertex(key, label)
7
8     def getVertex(self, key): # 返回 id 为 key 的节点
9         return self.vertices.get(key)
10

```

```

11     def __contains__(self, key): # 判断 key 节点是否在图中
12         return key in self.vertices
13
14     def addEdge(self, f, t, cost=0): # 添加从节点 id==f 到 id==t 的边
15         if f in self.vertices and t in self.vertices:
16             self.vertices[f].addNeighbor(t, cost)
17
18     def getVertices(self): # 返回所有的节点 key
19         return self.vertices.keys()
20
21     def __iter__(self): # 迭代每一个节点对象
22         return iter(self.vertices.values())
23
24
25 class Vertex:
26     def __init__(self, key, label=None): # 缺省颜色为"white"
27         self.id = key
28         self.label = label
29         self.color = "white"
30         self.connections = {}
31
32     def addNeighbor(self, nbr, weight=0): # 添加到节点 nbr 的边
33         self.connections[nbr] = weight
34
35     def setColor(self, color): # 设置节点颜色标记
36         self.color = color
37
38     def getColor(self): # 返回节点颜色标记
39         return self.color
40
41     def getConnections(self): # 返回节点的所有邻接节点列表
42         return self.connections.keys()
43
44     def getId(self): # 返回节点的 id
45         return self.id
46
47     def getLabel(self): # 返回节点的附带数据 label
48         return self.label
49
50 #https://github.com/Yuqiu-
51 Yang/problem_solving_with_algorithms_and_data_structures_using_python/blob/master/ch
52 /ch4_maze2.txt
53 mazelist = [
54     "++++++++++++++",
55     "+    +    ++ ++      +",
56     "E    +    +++++++",
57     "+ +    ++ ++++ +++ ++",
58     "+ +    + + ++      +++ +",
59     "+           ++ ++ + +",
60     "+++++ + +    ++ + +",
61     "+++++ +++    + + ++ +",
62     "+           + + S+ + +",

```

```

61     "+++++ + + + + +",
62     "++++++++++++++",
63 ]
64
65 def mazeGraph(mlist, rows, cols): # 从 mlist 创建图, 迷宫有 rows 行 cols 列
66     mGraph = Graph()
67     vstart = None
68     for row in range(rows):
69         for col in range(cols):
70             if mlist[row][col] != "+":
71                 mGraph.addVertex((row, col), mlist[row][col])
72             if mlist[row][col] == "S":
73                 vstart = mGraph.getVertex((row, col)) # 等号右侧填空 (1分)
74
75     for v in mGraph:
76         row, col = v.getId()
77         for i in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
78             if 0 <= row + i[0] < rows and 0 <= col + i[1] < cols:
79                 if (row + i[0], col + i[1]) in mGraph:
80                     mGraph.addEdge((row, col), (row + i[0], col + i[1])) #括号中两个参
数填空 (1分)
81
82     return mGraph, vstart # 返回图对象, 和开始节点
83
84
85 def searchMaze(path, vcurrent, mGraph): # 从 vcurrent 节点开始 DFS 搜索迷宫, path 保存路
径
86     path.append(vcurrent.getId())
87     vcurrent.setColor("gray")
88     if vcurrent.getLabel() != "E":
89         done = False
90         for nbr in vcurrent.getConnections(): # in 后面部分填空 (2分)
91             nbr_vertex = mGraph.getVertex(nbr)
92             if nbr_vertex.getColor() == "white":
93                 done = searchMaze(path, nbr_vertex, mGraph) # 参数填空 (2分)
94                 if done:
95                     break
96             if not done:
97                 path.pop() # 这条语句空着, 填空 (2分)
98                 vcurrent.setColor("white")
99             else:
100                 done = True
101     return done # 返回是否成功找到通路
102
103
104 g, vstart = mazeGraph(mazelist, len(mazelist), len(mazelist[0]))
105 path = []
106 searchMaze(path, vstart, g)
107 print(path)
108

```

```
# [(8, 15), (7, 15), (7, 14), (6, 14), (5, 14), (4, 14), (4, 13), (5, 13), (6, 13),
(6, 12), (6, 11), (6, 10), (5, 10), (5, 9), (4, 9), (3, 9), (2, 9), (2, 8), (2, 7),
(1, 7), (1, 6), (1, 5), (2, 5), (3, 5), (4, 5), (5, 5), (5, 4), (4, 4), (3, 4), (2,
4), (2, 3), (1, 3), (1, 2), (2, 2), (2, 1), (2, 0)]
```

4 (Week10) 更多图算法

接下来我们介绍更多图算法，多是构建于基本宽度优先搜索（BFS）或者深度优先搜索（DFS）之上。图算法可以根据其功能和应用领域进行分类。以下是一些常见的图算法分类：

1. 最短路径算法：

- Dijkstra算法：用于找到两个顶点之间的最短路径。
- Bellman-Ford算法：用于处理带有负权边的图的最短路径问题。
- Floyd-Warshall算法：用于找到图中所有顶点之间的最短路径。

2. 最小生成树算法：

- Prim算法：用于找到连接所有顶点的最小生成树。
- Kruskal算法 / 并查集：用于找到连接所有顶点的最小生成树，适用于边集合已经给定的情况。

3. 拓扑排序算法：

- DFS：用于对有向无环图（DAG）进行拓扑排序。
- Kahn算法 / BFS：用于对有向无环图进行拓扑排序。

4. 强连通分量算法：

- Kosaraju算法 / 2 DFS：用于找到有向图中的所有强连通分量。
- Tarjan算法：用于找到有向图中的所有强连通分量。

这只是图算法的一些常见分类，还有很多其他类型的图算法，包括最大流量（Ford-Fulkerson算法）、最小割（Stoer-Wagner算法）、图着色（贪心算法）、二分图最大匹配（匈牙利算法）、社区检测、网络分析、路径规划等。每个算法都有其独特的特点和应用场景，选择适当的算法取决于具体的问题和需求。

4.1 拓扑排序

拓扑排序（Topological Sorting）是对有向无环图（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。

拓扑排序可以用于解决一些依赖关系的问题，例如任务调度、编译顺序等。

4.1.1 煎松饼

This section shows how we can use depth-first search to perform a topological sort of a directed acyclic graph, or a “dag” as it is sometimes called. A **topological sort** of a dag $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v) , then u appears before v in the ordering. (If the graph contains a cycle, then no linear ordering is possible.) We can view a topological sort of a graph as an ordering of its vertices along a horizontal line so that all directed edges go from left to right. Topological sorting is thus different from the usual kind of “sorting” studied in Part II. Many applications use directed acyclic graphs to indicate precedences among events.

为了展示计算机科学家可以将几乎所有问题都转换成图问题，让我们来考虑如何制作一批松饼。配方十分简单：一个鸡蛋、一杯松饼粉、一勺油，以及3/4杯牛奶。为了制作松饼，需要加热平底锅，并将所有原材料混合后倒入锅中。当出现气泡时，将松饼翻面，继续煎至底部变成金黄色。在享用松饼之前，还会加热一些枫糖浆。图7-18用图的形式展示了整个过程。

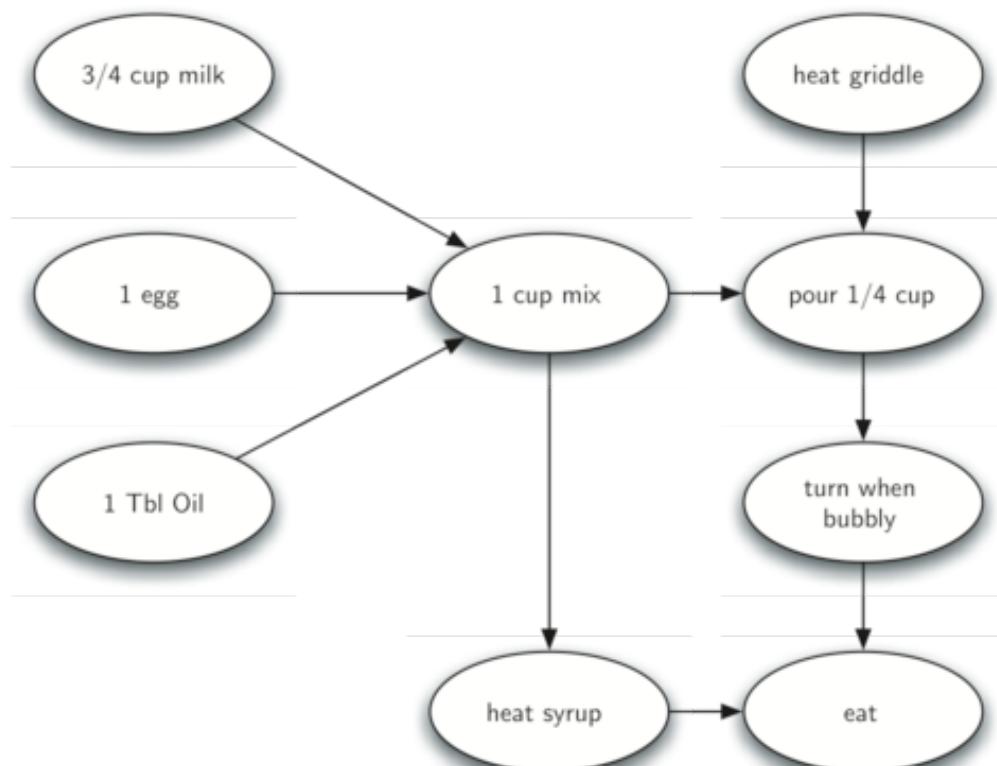


图1 松饼的制作步骤

制作松饼的难点在于知道先做哪一步。从图1可知，可以首先加热平底锅或者混合原材料。我们借助拓扑排序这种图算法来确定制作松饼的步骤。

拓扑排序根据有向无环图生成一个包含所有顶点的线性序列，使得如果图 G 中有一条边为 (v, w) ，那么顶点 v 排在顶点 w 之前。在很多应用中，有向无环图被用于表明事件优先级。制作松饼只是其中一个例子，其他例子还包括软件项目调度、优化数据库查询的优先级表，以及矩阵相乘。

拓扑排序是对深度优先搜索的一种简单而强大的改进，其算法如下。

- (1) 对图 g 调用 `dfs(g)`。之所以调用深度优先搜索函数，是因为要计算每一个顶点的结束时间。
- (2) 基于结束时间，将顶点按照递减顺序存储在列表中。
- (3) 将有序列表作为拓扑排序的结果返回。

图2 展示了dfs根据如图1所示的松饼制作步骤构建的深度优先森林。

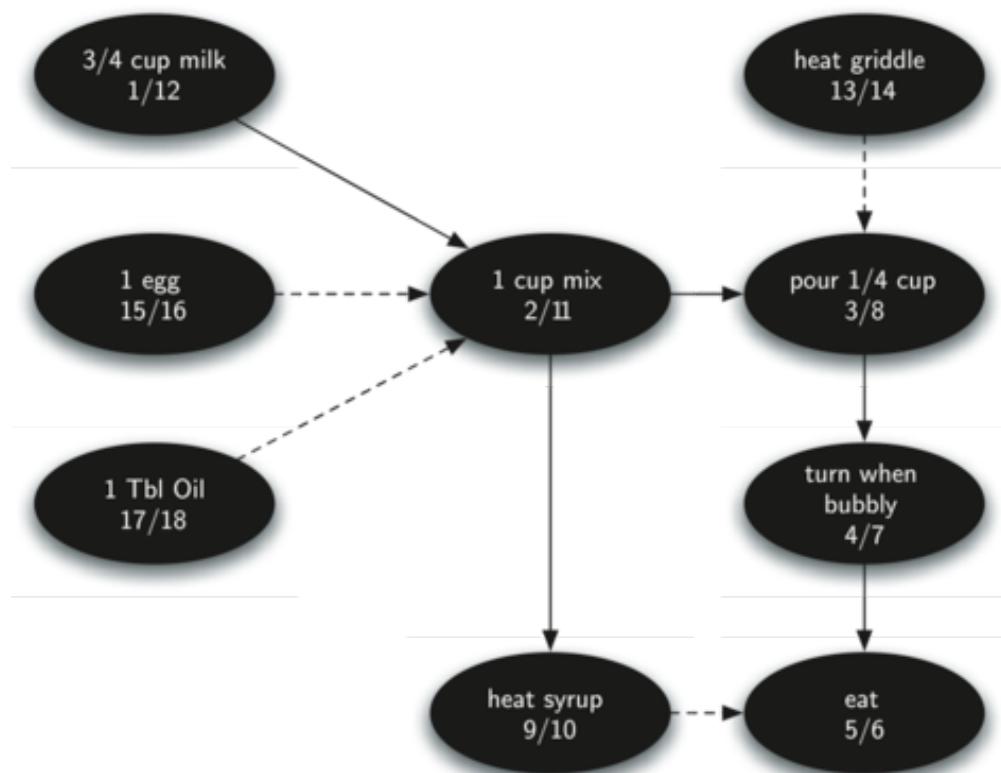


图2 根据松饼制作步骤构建的深度优先森林

图3 展示了拓扑排序结果。现在，我们明确地知道了制作松饼所需的步骤

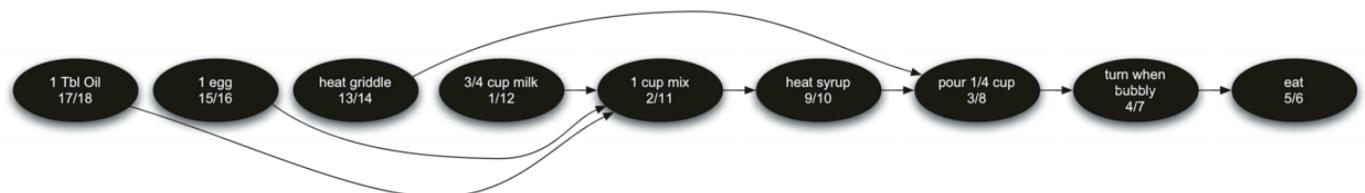


图3 对有向无环图的拓扑排序结果

4.1.2 实现煎松饼 with DFS

最新MakingPancake_DepthFirstForest.py 在 <https://github.com/GMyhf/2024spring-cs201/tree/main/code>

```
1 import sys
2
3 class Graph:
4     def __init__(self):
5         self.vertices = {}
6         self.num_vertices = 0
```

```

7
8     def add_vertex(self, key):
9         self.num_vertices = self.num_vertices + 1
10        new_ertex = Vertex(key)
11        self.vertices[key] = new_ertex
12        return new_ertex
13
14    def get_vertex(self, n):
15        if n in self.vertices:
16            return self.vertices[n]
17        else:
18            return None
19
20    def __len__(self):
21        return self.num_vertices
22
23    def __contains__(self, n):
24        return n in self.vertices
25
26    def add_edge(self, f, t, cost=0):
27        if f not in self.vertices:
28            nv = self.add_vertex(f)
29        if t not in self.vertices:
30            nv = self.add_vertex(t)
31        self.vertices[f].add_neighbor(self.vertices[t], cost)
32        #self.vertices[t].add_neighbor(self.vertices[f], cost)
33
34    def getVertices(self):
35        return list(self.vertices.keys())
36
37    def __iter__(self):
38        return iter(self.vertices.values())
39
40
41    class Vertex:
42        def __init__(self, num):
43            self.key = num
44            self.connectedTo = {}
45            self.color = 'white'
46            self.distance = sys.maxsize
47            self.previous = None
48            self.discovery = 0
49            self.finish = None
50
51        def __lt__(self, o):
52            return self.key < o.key
53
54        def add_neighbor(self, nbr, weight=0):
55            self.connectedTo[nbr] = weight
56
57        def setDiscovery(self, dtime):
58            self.discovery = dtime

```

```

59
60     def setFinish(self, ftime):
61         self.finish = ftime
62
63     def getFinish(self):
64         return self.finish
65
66     def getDiscovery(self):
67         return self.discovery
68
69     def get_neighbors(self):
70         return self.connectedTo.keys()
71
72     # def getWeight(self, nbr):
73     #     return self.connectedTo[nbr]
74
75     def __str__(self):
76         return str(self.key) + ":color " + self.color + ":disc " +
77             str(self.discovery) + ":fin " + str(
78                 self.finish) + ":dist " + str(self.distance) + ":pred \n\t[ " +
79             str(self.previous) + "] \n"
80
81
82 class DFSGraph(Graph):
83     def __init__(self):
84         super().__init__()
85         self.time = 0
86         self.topologicalList = []
87
88     def dfs(self):
89         for aVertex in self:
90             aVertex.color = "white"
91             aVertex.predecessor = -1
92         for aVertex in self:
93             if aVertex.color == "white":
94                 self.dfsvisit(aVertex)
95
96     def dfsvisit(self, startVertex):
97         startVertex.color = "gray"
98         self.time += 1
99         startVertex.setDiscovery(self.time)
100        for nextVertex in startVertex.get_neighbors():
101            if nextVertex.color == "white":
102                nextVertex.previous = startVertex
103                self.dfsvisit(nextVertex)
104                startVertex.color = "black"
105                self.time += 1
106                startVertex.setFinish(self.time)
107
108    def topologicalSort(self):
109        self.dfs()
110        temp = list(self.vertices.values())

```

```

109     temp.sort(key = lambda x: x.getFinish(), reverse = True)
110     print([(x.key,x.finish) for x in temp])
111     self.topologicalList = [x.key for x in temp]
112     return self.topologicalList
113
114 # Creating the graph
115 g = DFSGraph()
116
117 g.add_vertex('cup_milk')      # 3/4杯牛奶
118 g.add_vertex('egg')          # 一个鸡蛋
119 g.add_vertex('tbl_oil')       # 1勺油
120
121 g.add_vertex('heat_griddle')   # 加热平底锅
122 g.add_vertex('mix_ingredients') # 混合材料—1杯松饼粉
123 g.add_vertex('pour_batter')    # 倒入1/4杯松饼粉
124 g.add_vertex('turn_pancake')    # 出现气泡时翻面
125 g.add_vertex('heat_syrup')     # 加热枫糖浆
126 g.add_vertex('eat_pancake')    # 开始享用
127
128 # Adding edges based on dependencies
129 g.add_edge('cup_milk', 'mix_ingredients')
130 g.add_edge('mix_ingredients', 'pour_batter')
131 g.add_edge('pour_batter', 'turn_pancake')
132 g.add_edge('turn_pancake', 'eat_pancake')
133
134 g.add_edge('mix_ingredients', 'heat_syrup')
135 g.add_edge('heat_syrup', 'eat_pancake')
136
137 g.add_edge('heat_griddle', 'pour_batter')
138 g.add_edge('tbl_oil', 'mix_ingredients')
139 g.add_edge('egg', 'mix_ingredients')
140
141
142
143 # Getting topological sort of the tasks
144 topo_order = g.topologicalSort()
145 print("Topological Sort of the Pancake Making Process:")
146 print(topo_order)
147
148 """
149 Output:
150 函数 topologicalSort 中的调试信息
151 [('eat_pancake', 18), ('heat_syrup', 16), ('turn_pancake', 14), ('pour_cup', 12),
152 ('cup_mix', 10), ('cup_milk', 8), ('heat_griddle', 6), ('tbl_oil', 4), ('egg', 2)]
153 Topological Sort of the Pancake Making Process:
154 ['heat_griddle', 'tbl_oil', 'egg', 'cup_milk', 'cup_mix', 'heat_syrup', 'pour_cup',
155 'turn_pancake', 'eat_pancake']
"""

```

4.1.3 Kahn算法 / BFS

Kahn算法是基于广度优先搜索（BFS）的一种拓扑排序算法。

Kahn算法的基本思想是通过不断地移除图中的入度为0的顶点，并将其添加到拓扑排序的结果中，直到图中所有的顶点都被移除。具体步骤如下：

1. 初始化一个队列，用于存储当前入度为0的顶点。
2. 遍历图中的所有顶点，计算每个顶点的入度，并将入度为0的顶点加入到队列中。
3. 不断地从队列中弹出顶点，并将其加入到拓扑排序的结果中。同时，遍历该顶点的邻居，并将其入度减1。如果某个邻居的入度减为0，则将其加入到队列中。
4. 重复步骤3，直到队列为空。

Kahn算法的时间复杂度为 $O(V + E)$ ，其中 V 是顶点数， E 是边数。它是一种简单而高效的拓扑排序算法，在有向无环图（DAG）中广泛应用。

如果 `result` 列表的长度等于图中顶点的数量，则拓扑排序成功，返回结果列表 `result`；否则，图中存在环，无法进行拓扑排序。

下面是一个使用 Kahn 算法进行拓扑排序的示例代码：

```
1  from collections import deque, defaultdict
2
3  def topological_sort(graph):
4      indegree = defaultdict(int)
5      result = []
6      queue = deque()
7
8      # 计算每个顶点的入度
9      for u in graph:
10         for v in graph[u]:
11             indegree[v] += 1
12
13     # 将入度为 0 的顶点加入队列
14     for u in graph:
15         if indegree[u] == 0:
16             queue.append(u)
17
18     # 执行拓扑排序
19     while queue:
20         u = queue.popleft()
21         result.append(u)
22
23         for v in graph[u]:
24             indegree[v] -= 1
25             if indegree[v] == 0:
26                 queue.append(v)
27
```

```

28     # 检查是否存在环
29     if len(result) == len(graph):
30         return result
31     else:
32         return None
33
34 # 示例调用代码
35 graph = {
36     'A': ['B', 'C'],
37     'B': ['C', 'D'],
38     'C': ['E'],
39     'D': ['F'],
40     'E': ['F'],
41     'F': []
42 }
43
44 sorted_vertices = topological_sort(graph)
45 if sorted_vertices:
46     print("Topological sort order:", sorted_vertices)
47 else:
48     print("The graph contains a cycle.")
49
50 # Output:
51 # Topological sort order: ['A', 'B', 'C', 'D', 'E', 'F']

```

在上述代码中，`graph` 是一个字典，用于表示有向图的邻接关系。它的键表示顶点，值表示一个列表，表示从该顶点出发的边所连接的顶点。

你可以将你的有向图表示为一个邻接矩阵或邻接表，并将其作为参数传递给 `topological_sort` 函数。如果存在拓扑排序，函数将返回一个列表，按照拓扑排序的顺序包含所有顶点。如果图中存在环，函数将返回 `None`，表示无法进行拓扑排序。

4.1.4 实现煎松饼 with Kahn

最新 `MakingPancake_Kahn.py` 在 <https://github.com/GMyhf/2024spring-cs201/tree/main/code>

```

1 from collections import deque, defaultdict
2
3 def topological_sort(graph):
4     indegree = defaultdict(int)
5     result = []
6     queue = deque()
7
8     # 计算每个顶点的入度
9     for u in graph:
10         for v in graph[u]:
11             indegree[v] += 1
12
13     # 将入度为 0 的顶点加入队列
14     for u in graph:

```

```

15     if indegree[u] == 0:
16         queue.append(u)
17
18     # 执行拓扑排序
19     while queue:
20         u = queue.popleft()
21         result.append(u)
22
23         for v in graph[u]:
24             indegree[v] -= 1
25             if indegree[v] == 0:
26                 queue.append(v)
27
28     # 检查是否存在环
29     if len(result) == len(graph):
30         return result
31     else:
32         return None
33
34 # 示例调用代码
35 graph = {
36     'cup_milk': ['mix_ingredients'],
37     'mix_ingredients': ['pour_batter', 'heat_syrup'],
38     'pour_batter': ['turn_pancake'],
39     'turn_pancake': ['eat_pancake'],
40     'heat_syrup': ['eat_pancake'],
41     'heat_griddle': ['pour_batter'],
42     'tbl_oil': ['mix_ingredients'],
43     'egg': ['mix_ingredients'],
44     'eat_pancake': []
45 }
46
47
48 sorted_vertices = topological_sort(graph)
49 if sorted_vertices:
50     print("Topological sort order:", sorted_vertices)
51 else:
52     print("The graph contains a cycle.")
53
54 """
55 #Depth First Forest ouput:
56 #['heat_griddle', 'tbl_oil', 'egg', 'cup_milk', 'mix_ingredients', 'heat_syrup',
57 # 'pour_batter', 'turn_pancake', 'eat_pancake']
58
59 # Kahn ouput:
60 Topological sort order: ['cup_milk', 'heat_griddle', 'tbl_oil', 'egg',
61 'mix_ingredients', 'pour_batter', 'heat_syrup', 'turn_pancake', 'eat_pancake']
62
63 """

```

4.2 强连通单元 (SCCs)

Page 615, introduction to algorithms 3rd Edition

We now consider a classic application of depth-first search: decomposing a directed graph into its strongly connected components. This section shows how to do so using two depth-first searches. Many algorithms that work with directed graphs begin with such a decomposition. After decomposing the graph into strongly connected components, such algorithms run separately on each one and then combine the solutions according to the structure of connections among components.

接下来将注意力转向规模庞大的图。我们将以互联网主机与各个网页构成的图为例，学习其他几种算法。首先讨论网页。

在互联网上，各种网页形成一张大型的有向图，谷歌和必应等搜索引擎正是利用了这一事实。要将互联网转换成一张图，我们将网页当作顶点，将超链接当作连接顶点的边。图1展示了以路德学院计算机系的主页作为起点的网页连接图的一小部分。由于这张图的规模庞大，因此我们限制网页与起点页之间的链接数不超过10个。

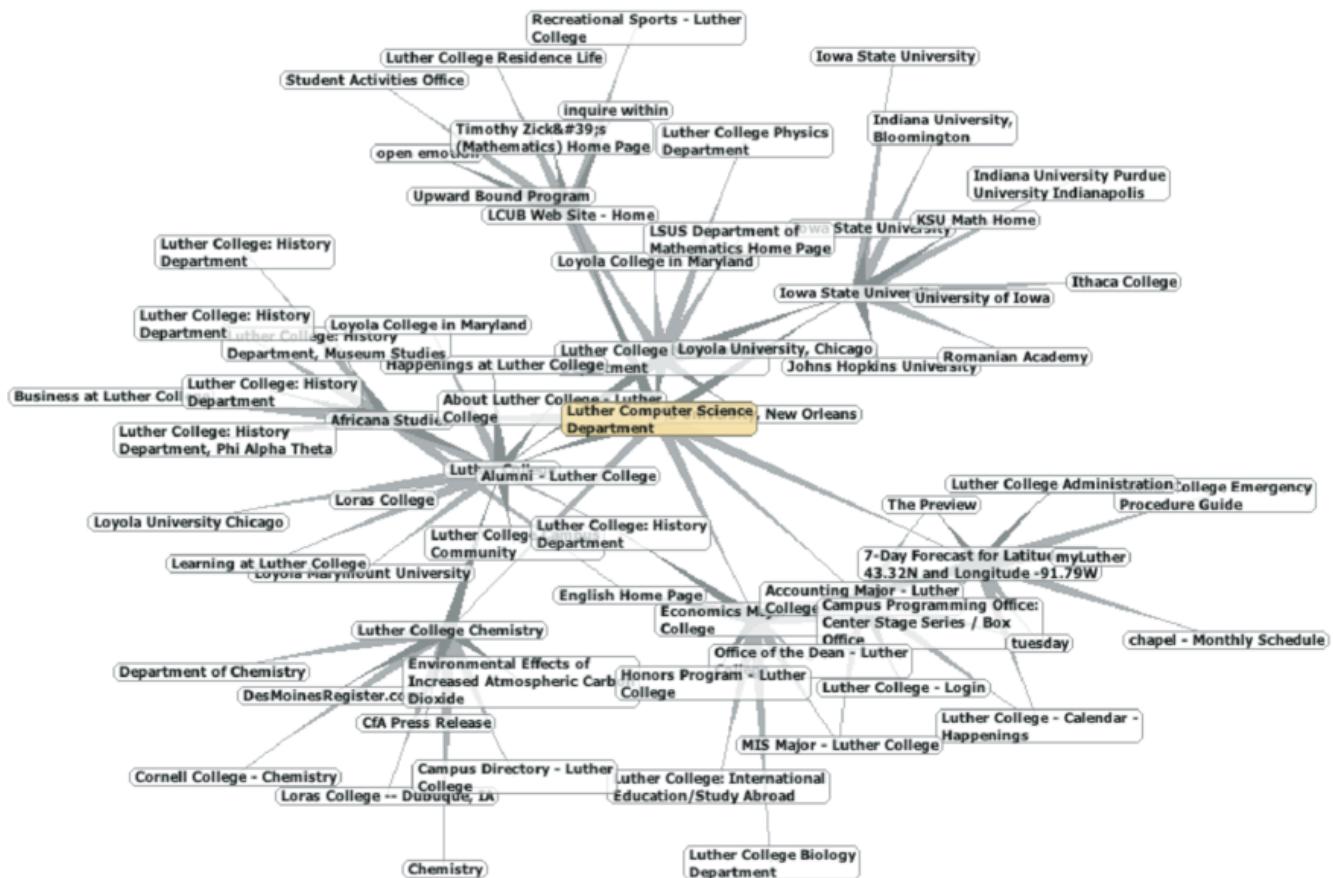


图1 以路德学院计算机系的主页作为起点的网页连接图

仔细研究图1，会有一些非常有趣的发现。首先，图中的很多网页来自路德学院的其他网站。其次，一些链接指向爱荷华州的其他学校。最后，一些链接指向其他文理学院。由此可以得出这样的结论：网络具有一种基础结构，使得在某种程度上相似的网页相互聚集。

通过一种叫作强连通单元的图算法，可以找出图中高度连通的顶点簇。对于图G，强连通单元C为最大的顶点子集 $C \subset V$ ，其中对于每一对顶点 $v, w \in C$ ，都有一条从 v 到 w 的路径和一条从 w 到 v 的路径。

We formally define a **strongly connected component (SCC)**, C , of a graph G , as the largest subset of vertices $C \subset V$ such that for every pair of vertices $v, w \in C$ we have a path from v to w and a path from w to v .

图2展示了一个包含3个强连通单元的简单图。不同的强连通单元通过不同的阴影来表现。

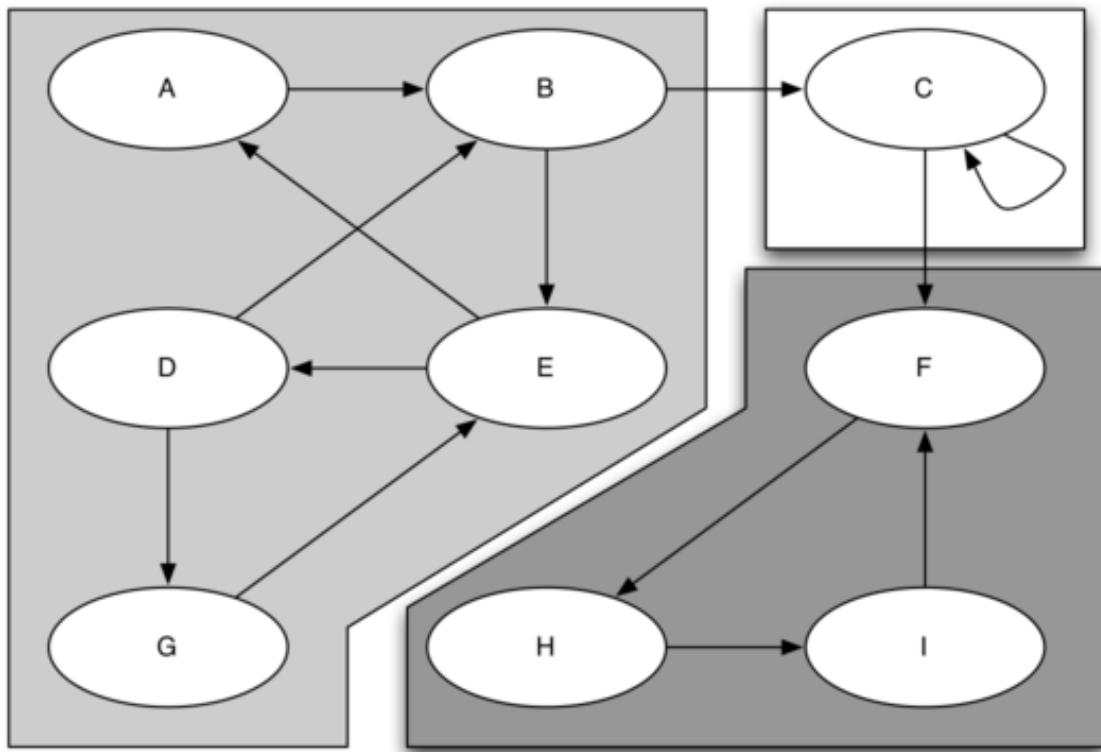


图2 含有3个强连通单元的有向图

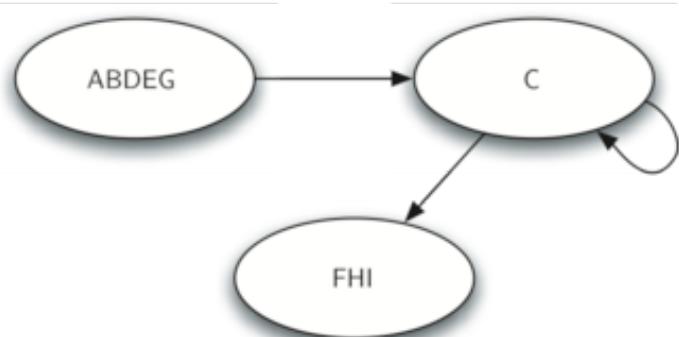
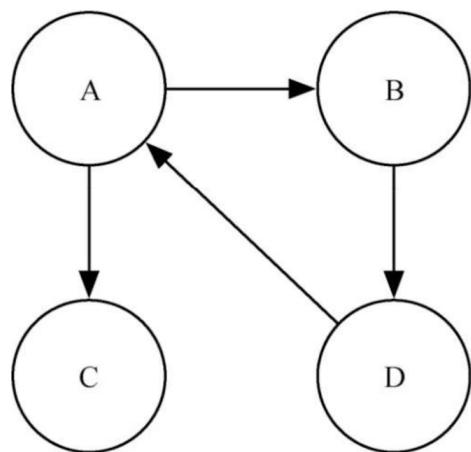


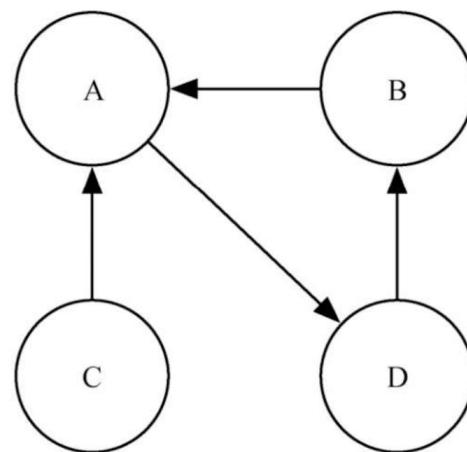
图3 简化后的有向图

定义强连通单元之后，就可以把强连通单元中的所有顶点组合成单个顶点，从而将图简化。图3是图2的简化版。

利用深度优先搜索，我们可以再次创建强大高效的算法。在学习强连通单元算法之前，还要再看一个定义。图G的转置图被定义为 G^T ，其中所有的边都与图G的边反向。这意味着，如果在图G中有一条由A到B的边，那么在 G^T 中就会有一条由B到A的边。图4展示了一个简单图及其转置图。



(a) 图G



(b) 图G的转置图

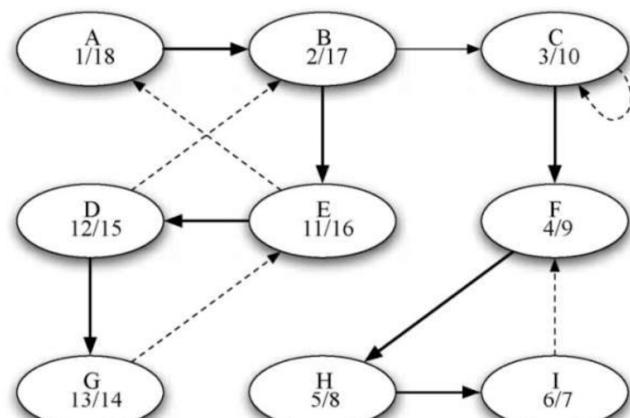
图4 图G及其转置图

再次观察图4。注意，图4a中有2个强连通单元，图4b中也是如此。

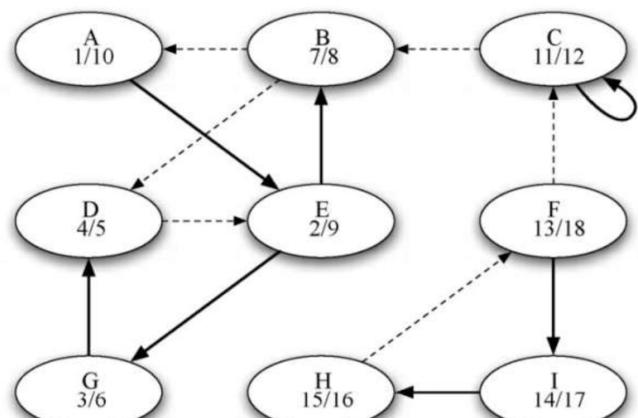
以下是计算强连通单元的算法。

- (1) 对图G调用dfs，以计算每一个顶点的结束时间。
- (2) 计算图 G^T 。
- (3) 对图 G^T 调用dfs，但是在主循环中，按照结束时间的递减顺序访问顶点。
- (4) 第3步得到的深度优先森林中的每一棵树都是一个强连通单元。输出每一棵树中的顶点的id。

以图2为例，让我们来逐步分析。图5a展示了用深度优先搜索算法对原图计算得到的发现时间和结束时间，图5b展示了用深度优先搜索算法在转置图上得到的发现时间和结束时间。



(a) 图G



(b) 图G的转置图

图5 计算强连通单元

最后，图6展示了由强连通单元算法在第3步生成的森林，其中有3棵树。我们没有提供强连通单元算法的Python代码

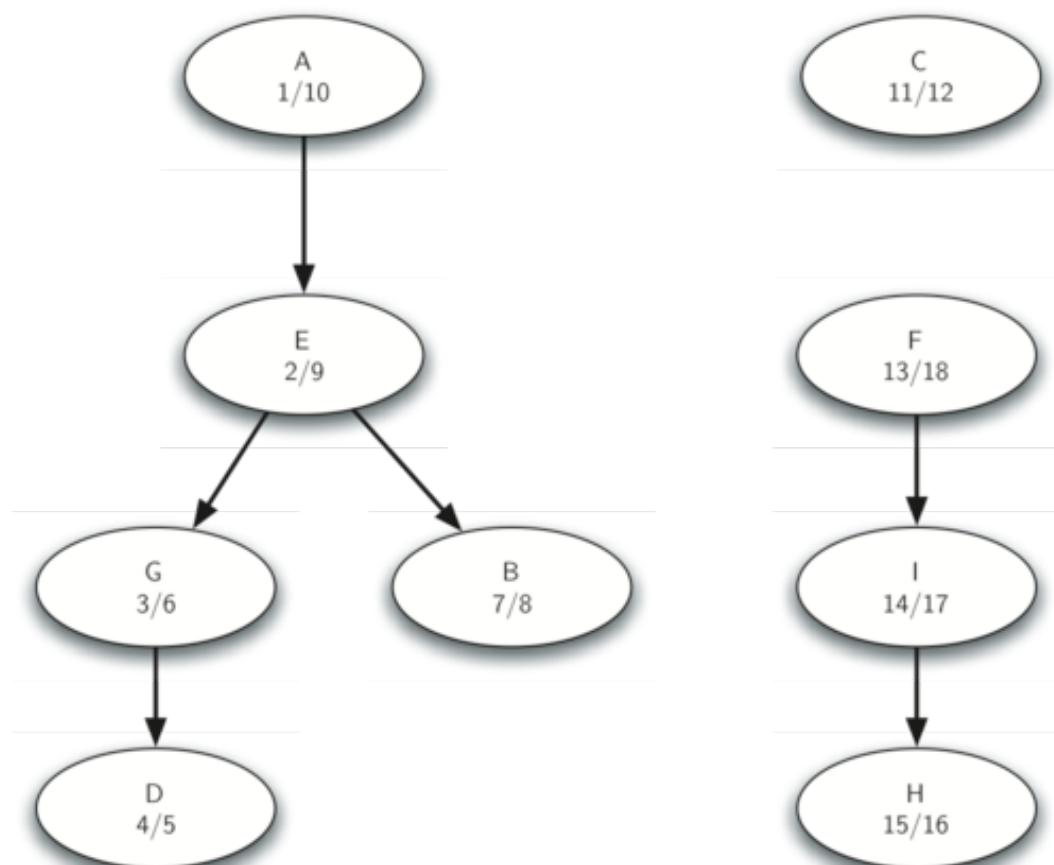


图6 由强连通单元算法生成的森林 SCCs

4.2.1 Kosaraju / 2 DFS

Kosaraju算法是一种用于在有向图中寻找强连通分量（Strongly Connected Components, SCC）的算法。它基于深度优先搜索（DFS）和图的转置操作。

Kosaraju算法的核心思想就是两次深度优先搜索（DFS）。

1. **第一次DFS**: 在第一次DFS中，我们对图进行标准的深度优先搜索，但是在此过程中，我们记录下顶点完成搜索的顺序。这一步的目的是为了找出每个顶点的完成时间（即结束时间）。
2. **反向图**: 接下来，我们对原图取反，即将所有的边方向反转，得到反向图。
3. **第二次DFS**: 在第二次DFS中，我们按照第一步中记录的顶点完成时间的逆序，对反向图进行DFS。这样，我们将找出反向图中的强连通分量。

Kosaraju算法的关键在于第二次DFS的顺序，它保证了在DFS的过程中，我们能够优先访问到整个图中的强连通分量。因此，Kosaraju算法的时间复杂度为 $O(V + E)$ ，其中 V 是顶点数， E 是边数。

以下是Kosaraju算法的Python实现：

```
1 | def dfs1(graph, node, visited, stack):
```

```

2     visited[node] = True
3     for neighbor in graph[node]:
4         if not visited[neighbor]:
5             dfs1(graph, neighbor, visited, stack)
6     stack.append(node)
7
8 def dfs2(graph, node, visited, component):
9     visited[node] = True
10    component.append(node)
11    for neighbor in graph[node]:
12        if not visited[neighbor]:
13            dfs2(graph, neighbor, visited, component)
14
15 def kosaraju(graph):
16     # Step 1: Perform first DFS to get finishing times
17     stack = []
18     visited = [False] * len(graph)
19     for node in range(len(graph)):
20         if not visited[node]:
21             dfs1(graph, node, visited, stack)
22
23     # Step 2: Transpose the graph
24     transposed_graph = [[] for _ in range(len(graph))]
25     for node in range(len(graph)):
26         for neighbor in graph[node]:
27             transposed_graph[neighbor].append(node)
28
29     # Step 3: Perform second DFS on the transposed graph to find SCCs
30     visited = [False] * len(graph)
31     sccs = []
32     while stack:
33         node = stack.pop()
34         if not visited[node]:
35             scc = []
36             dfs2(transposed_graph, node, visited, scc)
37             sccs.append(scc)
38     return sccs
39
40 # Example
41 graph = [[1], [2, 4], [3, 5], [0, 6], [5], [4], [7], [5, 6]]
42 sccs = kosaraju(graph)
43 print("Strongly Connected Components:")
44 for scc in sccs:
45     print(scc)
46
47 """
48 Strongly Connected Components:
49 [0, 3, 2, 1]
50 [6, 7]
51 [5, 4]
52 """

```

这段代码首先定义了两个DFS函数，分别用于第一次DFS和第二次DFS。然后，Kosaraju算法包含了三个步骤：

1. 第一次DFS：遍历整个图，记录每个节点的完成时间，并将节点按照完成时间排序后压入栈中。
2. 图的转置：将原图中的边反转，得到转置图。
3. 第二次DFS：按照栈中节点的顺序，对转置图进行DFS，从而找到强连通分量。

最后，输出找到的强连通分量。

*Tarjan算法

Kosaraju's algorithm is similar to Tarjan's algorithm in that both are used to find strongly connected components of a graph. While Tarjan's and Kosaraju's algorithms have a linear time complexity, their SCC computing methodologies differ. Tarjan's technique partitions the graph using nodes' timestamps in a DFS, but Kosaraju's method partitions the graph using two DFSs and is comparable to the method for discovering topological sorting.

Tarjan算法是一种图算法，用于查找有向图中的强连通分量。强连通分量是指在有向图中，存在一条路径可以从任意一个顶点到达另一个顶点的一组顶点。

Tarjan算法使用了一种称为深度优先搜索（DFS）的技术来遍历图，并在遍历的过程中标记和识别强连通分量。算法的基本思想是，通过在深度优先搜索的过程中维护一个栈来记录已经访问过的顶点，并为每个顶点分配一个“搜索次序”（DFS编号）和一个“最低链接值”。搜索次序表示顶点被首次访问的次序，最低链接值表示从当前顶点出发经过一系列边能到达的最早的顶点的搜索次序。

Tarjan算法的步骤如下：

1. 从图中选择一个未访问的顶点开始深度优先搜索。
2. 为当前顶点分配一个搜索次序和最低链接值，并将其入栈。
3. 对当前顶点的每个邻接顶点进行递归深度优先搜索，如果邻接顶点尚未被访问过，则递归调用。
4. 在递归回溯的过程中，更新当前顶点的最低链接值，使其指向当前顶点和其邻接顶点之间较小的搜索次序。
5. 如果当前顶点的最低链接值等于其自身的搜索次序，那么将从当前顶点开始的栈中的所有顶点弹出，并将它们构成一个强连通分量。

通过这样的深度优先搜索和回溯过程，Tarjan算法能够识别出图中的所有强连通分量。算法的时间复杂度为 $O(V+E)$ ，其中 V 是顶点数， E 是边数。

总结起来，Tarjan算法是一种用于查找有向图中强连通分量的算法，它利用深度优先搜索和回溯的技术，在遍历图的过程中标记和识别强连通分量。

以下是Tarjan算法的Python实现：

```
1 def tarjan(graph):  
2     def dfs(node):  
3         nonlocal index, stack, indices, low_link, on_stack, sccs  
4         index += 1  
5         indices[node] = index  
6         low_link[node] = index  
7         stack.append(node)  
8         on_stack[node] = True
```

```

9
10     for neighbor in graph[node]:
11         if indices[neighbor] == 0: # Neighbor not visited yet
12             dfs(neighbor)
13             low_link[node] = min(low_link[node], low_link[neighbor])
14         elif on_stack[neighbor]: # Neighbor is in the current SCC
15             low_link[node] = min(low_link[node], indices[neighbor])
16
17     if indices[node] == low_link[node]:
18         scc = []
19         while True:
20             top = stack.pop()
21             on_stack[top] = False
22             scc.append(top)
23             if top == node:
24                 break
25         sccs.append(scc)
26
27     index = 0
28     stack = []
29     indices = [0] * len(graph)
30     low_link = [0] * len(graph)
31     on_stack = [False] * len(graph)
32     sccs = []
33
34     for node in range(len(graph)):
35         if indices[node] == 0:
36             dfs(node)
37
38     return sccs
39
40 # Example
41 graph = [[1],           # Node 0 points to Node 1
42           [2, 4],        # Node 1 points to Node 2 and Node 4
43           [3, 5],        # Node 2 points to Node 3 and Node 5
44           [0, 6],        # Node 3 points to Node 0 and Node 6
45           [5],          # Node 4 points to Node 5
46           [4],          # Node 5 points to Node 4
47           [7],          # Node 6 points to Node 7
48           [5, 6]]       # Node 7 points to Node 5 and Node 6
49
50 sccs = tarjan(graph)
51 print("Strongly Connected Components:")
52 for scc in sccs:
53     print(scc)
54
55 """
56 Strongly Connected Components:
57 [4, 5]
58 [7, 6]
59 [3, 2, 1, 0]
60 """

```

在这个实现中，`tarjan` 函数接收一个有向图（使用邻接表表示）作为参数，并返回图中的强连通分量。算法使用深度优先搜索（DFS）来遍历图，并使用一个栈来记录遍历过程中的节点。在DFS过程中，算法通过记录每个节点的搜索次序（即DFS序号）和能够回溯到的最早的节点的DFS序号（即low值），来确定强连通分量。最后，算法输出找到的强连通分量。

4.3 最短路径（Shortest Paths）

当我们浏览网页、发送电子邮件，或者从校园的另一处登录实验室里的计算机时，在后台发生了很多事，信息从一台计算机传送到另一台计算机。深入地研究信息在多台计算机之间的传送过程，是计算机网络课程的主要内容。本节将适当地讨论互联网的运作机制，并以此介绍另一个非常重要的图算法。

图1从整体上展示了互联网的通信机制。当我们使用浏览器访问某一台服务器上的网页时，访问请求必须通过路由器从本地局域网传送到互联网上，并最终到达该服务器所在局域网对应的路由器。然后，被请求的网页通过相同的路径被传送回浏览器。在图1中，标有“互联网”的云图标中有众多额外的路由器，它们的工作就是协同将信息从一处传送到另一处。如果你的计算机支持traceroute命令，可以利用它亲眼看到许多路由器。List 1展示了traceroute命令的执行结果：在路德学院的Web服务器和明尼苏达大学的邮件服务器之间有13个路由器。

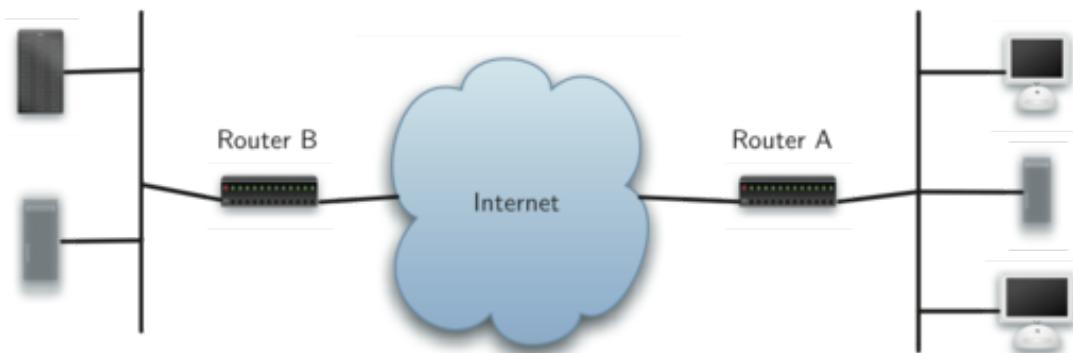


图1 互联网通信概览

List1 服务器之间的路由器

```
1 1 192.203.196.1
2 2 hilda.luther.edu (216.159.75.1)
3 3 ICN-Luther-Ether.icn.state.ia.us (207.165.237.137)
4 4 ICN-ISP-1.icn.state.ia.us (209.56.255.1)
5 5 p3-0.hsa1.chi1.bbnplanet.net (4.24.202.13)
6 6 ae-1-54.bbr2.Chicago1.Level3.net (4.68.101.97)
7 7 so-3-0-0.mpls2.Minneapolis1.Level3.net (64.159.4.214)
8 8 ge-3-0.hsa2.Minneapolis1.Level3.net (4.68.112.18)
9 9 p1-0.minnesota.bbnplanet.net (4.24.226.74)
10 10 TelecomB-BR-01-V4002.ggnet.umn.edu (192.42.152.37)
11 11 TelecomB-BN-01-Vlan-3000.ggnet.umn.edu (128.101.58.1)
12 12 TelecomB-CN-01-Vlan-710.ggnet.umn.edu (128.101.80.158)
13 13 baldrick.cs.umn.edu (128.101.80.129)(N!) 88.631 ms (N!)
14
```

互联网上的每一个路由器都与一个或多个其他的路由器相连。如果在不同的时间执行traceroute命令，极有可能看到信息在不同的路由器间流动。这是由于一对路由器之间的连接存在着一定的成本，成本大小取决于流量、时间段以及众多其他因素。至此，你应该能够理解为何可以用带权重的图来表示路由器网络。

图2展示了一个小型路由器网络对应的带权图。我们要解决的问题是为给定信息找到权重最小的路径。这个问题并不陌生，因为它和我们之前用宽度优先搜索解决过的问题十分相似，只不过现在考虑的是路径的总权重，而不是路径的长度。需要注意的是，如果所有的权重都相等，那么两个问题就没有区别。

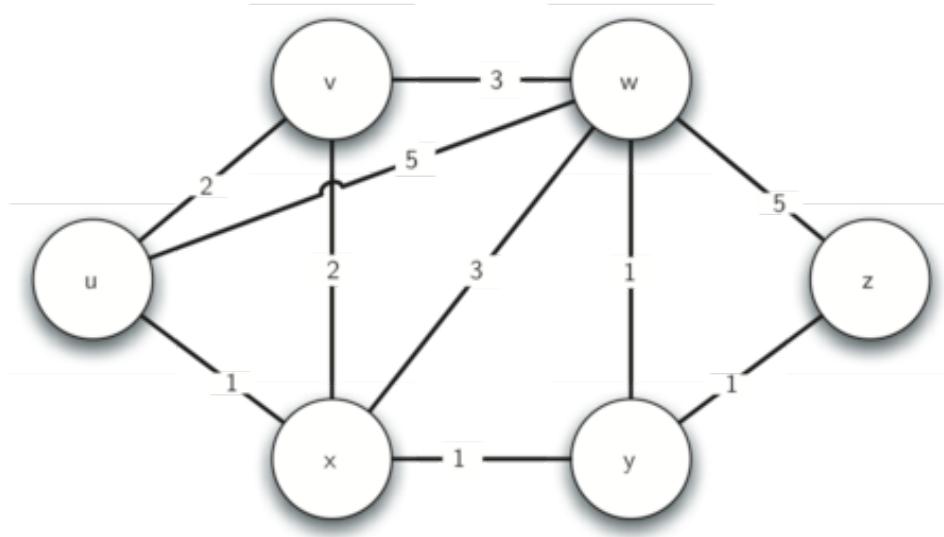


图2 小型路由器网络对应的带权图

4.3.1 有权图Dijkstra, 无权图BFS

Dijkstra算法与BFS（广度优先搜索）有相似之处，但它们有一些关键的区别。

1. 相似性：

- Dijkstra算法和BFS都是用于图的遍历。
- 它们都从一个起始顶点开始，逐步扩展到邻居顶点，并以某种方式记录已经访问过的顶点。

2. 不同之处：

- BFS是一种无权图的最短路径算法，它以层次遍历的方式遍历图，并找到从起始顶点到所有其他顶点的最短路径。
- Dijkstra算法是一种有权图的最短路径算法，它通过贪心策略逐步确定从起始顶点到所有其他顶点的最短路径。
- BFS使用队列来保存待访问的顶点，并按照顺序进行遍历。它不考虑权重，只关注路径的长度。

- Dijkstra算法使用优先队列（通常是最小堆）来保存待访问的顶点，并按照顶点到起始顶点的距离进行排序。它根据路径长度来决定下一个要访问的顶点，从而保证每次都是选择最短路径的顶点进行访问。

虽然Dijkstra算法的实现方式和BFS有些相似，但是它们解决的问题和具体实现细节有很大的不同。BFS适用于无权图的最短路径问题，而Dijkstra算法适用于有权图的最短路径问题。

Breadth First Search explores equally in all directions. This is an incredibly useful algorithm, not only for regular path finding, but also for procedural map generation, flow field pathfinding, distance maps, and other types of map analysis.

Dijkstra's Algorithm (also called Uniform Cost Search) lets us prioritize which paths to explore.

Instead of exploring all possible paths equally, it favors lower cost paths. We can assign lower costs to encourage moving on roads, higher costs to avoid enemies, and more. When movement costs vary, we use this instead of Breadth First Search.

Dijkstra算法可用于确定最短路径，它是一种循环算法，可以提供从一个顶点到其他所有顶点的最短路径。这与宽度优先搜索非常像。

为了记录从起点到各个终点的总开销，要利用Vertex类中的实例变量distance。该实例变量记录从起点到当前顶点的最小权重路径的总权重。Dijkstra算法针对图中的每个顶点都循环一次，但循环顺序是由一个优先级队列控制的。用来决定顺序的正是dist。在创建顶点时，将distance设为一个非常大的值。理论上可以将distance设为无穷大，但是实际一般将其设为一个大于所有可能出现的实际距离的值。

Dijkstra算法的实现如代码清单List 2所示。当程序运行结束时，distance和previous都会被设置成正确的值。

2024/4/14 说明：教材目前已经有第3版了。我尽量按照第3版做课件，例程与老版相比有稍微改动。

List1 Dijkstra算法的Python实现

```

1 # https://github.com/psads/pythonds
2 from pythonds3.graphs import PriorityQueue
3 def dijkstra(graph,start):
4     pq = PriorityQueue()
5     start.setDistance(0)
6     pq.buildHeap([(v.getDistance(),v) for v in graph])
7     while pq:
8         distance, current_v = pq.delete()
9         for next_v in current_v.getneighbors():
10             new_distance = current_v.distance + current_v.get_neighbor(next_v) # +
11             get_weight
12             if new_distance < next_v.distance:
13                 next_v.distance = new_distance
14                 next_v.previous = current_v
15                 pq.change_priority(next_v,new_distance)

```

```

1 from pythonds3.trees.binary_heap import BinaryHeap

```

```

2 class PriorityQueue(BinaryHeap):
3     def change_priority(self, search_key: Any, new_priority: Any) -> None:
4         key_to_move = -1
5         for i, ( _, key) in enumerate(self._heap):
6             if key == search_key:
7                 key_to_move = i
8                 break
9         if key_to_move > -1:
10            self._heap[key_to_move] = (new_priority, search_key)
11            self._perc_up(key_to_move)
12
13    def __contains__(self, search_key: Any) -> bool:
14        for _, key in self._heap:
15            if key == search_key:
16                return True
17        return False

```

Dijkstra算法使用了优先级队列。你应该记得，“树”那一章讲过如何用堆实现优先级队列。不过，当时的简单实现和用于Dijkstra算法的实现有几个不同点。首先，PriorityQueue类存储了键-值对的二元组。这对于Dijkstra算法来说非常重要，因为优先级队列中的键必须与图中顶点的键相匹配。其次，二元组中的值被用来确定优先级，对应键在优先级队列中的位置。在Dijkstra算法的实现中，我们使用了顶点的距离作为优先级，这是因为我们总希望访问距离最小的顶点。另一个不同点是增加了`change_priority`方法（第14行）。当到一个顶点的距离减少并且该顶点已在优先级队列中时，就调用这个方法，从而将该顶点移向优先级队列的头部。

让我们对照图3来理解如何针对每一个顶点应用Dijkstra算法。从顶点u开始，与u相邻的3个顶点分别是v、w和x。由于到v、w和x的初始距离都是`sys.maxint`，因此从起点到它们的新开销就是直接开销。更新这3个顶点的开销，同时将它们的前驱顶点设置成u，并将它们添加到优先级队列中。我们使用距离作为优先级队列的键。此时，算法运行的状态如图3a所示。

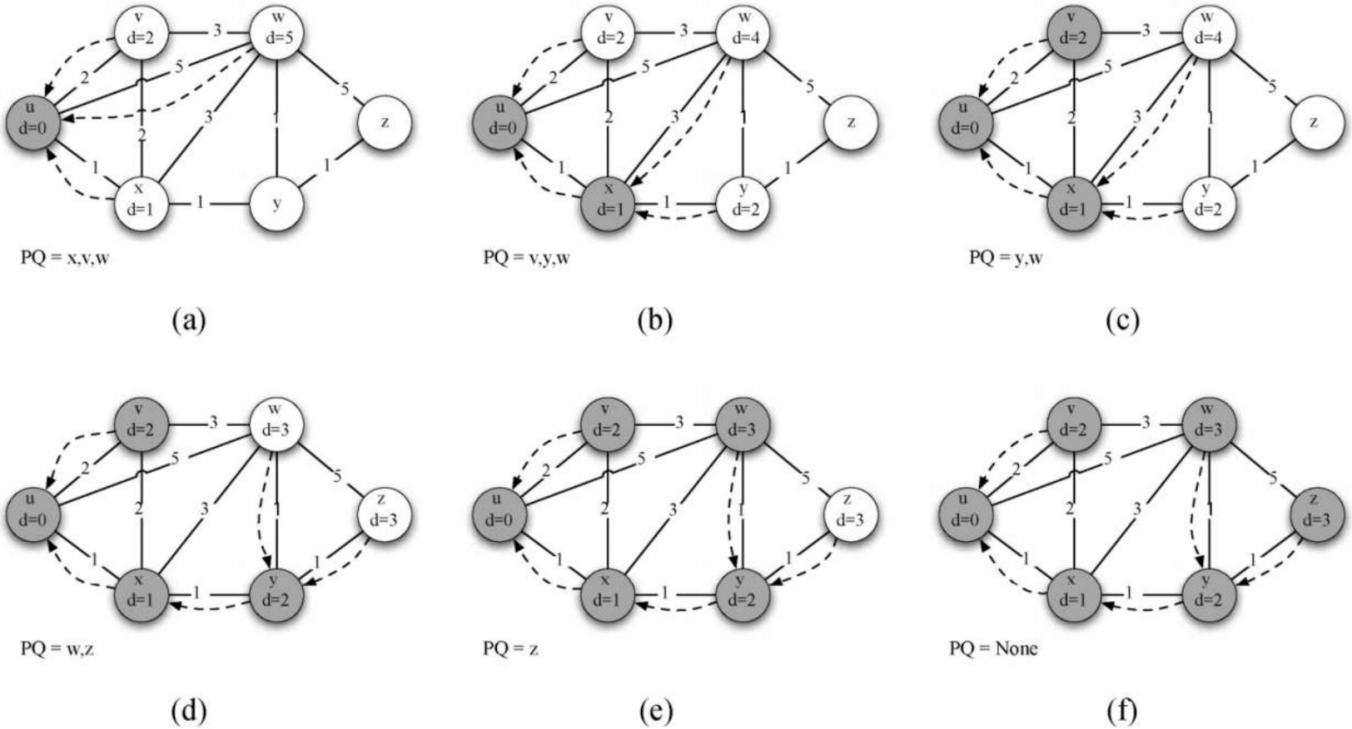


图3 Dijkstra算法的应用过程

下一次while循环检查与x相邻的顶点。之所以x是第2个被访问的顶点，是因为它到起点的开销最小，因此排在了优先级队列的头部。与x相邻的有u、v、w和y。对于每一个相邻顶点，检查经由x到它的距离是否比已知的距离更短。显然，对于y来说确实如此，因为它的初始距离是`sys.maxsize`；对于u和v来说则不然，因为它们的距离分别为0和2。但是，我们发现经过x到w的距离比直接从u到w的距离要短。因此，将到达w的距离更新为更短的值，并且将w的前驱顶点从u改为x。图3b展示了此时的状态。

下一步检查与v相邻的顶点。这一步没有对图做任何改动，因此我们继续检查顶点y。此时，我们发现经由y到达w和z的距离都更短，因此相应地调整它们的距离及前驱顶点。最后检查w和z，发现不需要做任何改动。由于优先级队列为空，因此退出。

非常重要的一点是，Dijkstra算法只适用于边的权重均为正的情况。如果图2中有一条边的权重为负，那么Dijkstra算法永远不会退出。

除了Dijkstra算法，还有其他一些算法被用于寻找最短路径。Dijkstra算法的问题是需要有完整的图，这意味着每一个路由器都要知道整个互联网的路由器连接情况，而事实并非如此。Dijkstra算法的一些变体允许每个路由器在运行时才发现图，例如“距离向量”路由算法。

4.3.2 分析Dijkstra算法

最后，我们来分析Dijkstra算法的时间复杂度。开始时，要将图中的每一个顶点都添加到优先级队列中，这个操作的时间复杂度是 $O(|V|)$ 。优先级队列构建完成之后，`while`循环针对每一个顶点都执行一次，这是由于一开始所有顶点都被添加到优先级队列中，并且只在循环时才被移除。在循环内部，每次对`pq.delete`的调用都是 $O(|V| \log(|V|))$ 。综合起来考虑，循环和`delMin`调用的总时间复杂度是 $O(|V| \log(|V|))$ 。`for`循环对图中的每一条边都执行一次，并且循环内部的`change_priority`调用为 $O(|E| \log |V|)$ 。因此，总的时间复杂度为 $O((|V| + |E|) \log(|V|))$ 。

*Bellman-Ford, SPFA算法

在图论中，有两种常见的方法用于求解最短路径问题：**Dijkstra算法**和**Bellman-Ford算法**。这两种算法各有优劣，选择哪种算法取决于图的特性和问题要求。如果图中没有负权边，并且只需要求解单源最短路径，Dijkstra算法通常是一个较好的选择。如果图中存在负权边或需要检测负权回路，或者需要求解所有节点对之间的最短路径，可以使用Bellman-Ford算法。

Bellman-Ford算法：Bellman-Ford算法用于解决单源最短路径问题，与Dijkstra算法不同，它可以处理带有负权边的图。算法的基本思想是通过松弛操作逐步更新节点的最短路径估计值，直到收敛到最终结果。具体步骤如下：

- 初始化一个距离数组，用于记录源节点到所有其他节点的最短距离。初始时，源节点的距离为0，其他节点的距离为无穷大。
- 进行 $V-1$ 次循环（ V 是图中的节点数），每次循环对所有边进行松弛操作。如果从节点 u 到节点 v 的路径经过节点 u 的距离加上边 (u, v) 的权重比当前已知的从源节点到节点 v 的最短路径更短，则更新最短路径。
- 检查是否存在负权回路。如果在 $V-1$ 次循环后，仍然可以通过松弛操作更新最短路径，则说明存在负权回路，因此无法确定最短路径。

Bellman-Ford算法的时间复杂度为 $O(V^2E)$ ，其中 V 是图中的节点数， E 是图中的边数。

SPFA是"Shortest Path Faster Algorithm"的缩写，中文名称为最短路径快速算法。它是一种用于解决带有负权边的图中单源最短路径问题的算法。

SPFA算法是对Bellman-Ford算法的一种优化，旨在减少算法的时间复杂度。Bellman-Ford算法的时间复杂度为 $O(V^2E)$ ，其中 V 是顶点数， E 是边数。而SPFA算法通过引入一个队列来避免对所有边进行松弛操作，从而减少了不必要的松弛操作，提高了算法的效率。

SPFA算法的基本思想如下：

1. 初始化源节点的最短距离为0，其他节点的最短距离为正无穷大。
2. 将源节点加入队列中，并标记为已访问。
3. 循环执行以下步骤直到队列为空：
 - 从队列中取出一个节点作为当前节点。
 - 遍历当前节点的所有邻接节点：
 - 如果经过当前节点到达该邻接节点的路径比当前记录的最短路径更短，则更新最短路径，并将该邻接节点加入队列中。
4. 当队列为空时，算法结束，所有节点的最短路径已计算出来。

SPFA算法在实际应用中通常表现出良好的性能，尤其适用于稀疏图（边数相对较少）和存在负权边的情况。然而，需要注意的是，如果图中存在负权环路，SPFA算法将无法给出正确的结果。

*多源最短路径Floyd-Warshall算法

求解所有顶点之间的最短路径可以使用**Floyd-Warshall算法**，它是一种多源最短路径算法。Floyd-Warshall算法可以在有向图或无向图中找到任意两个顶点之间的最短路径。

算法的基本思想是通过一个二维数组来存储任意两个顶点之间的最短距离。初始时，这个数组包含图中各个顶点之间的直接边的权重，对于不直接相连的顶点，权重为无穷大。然后，通过迭代更新这个数组，逐步求得所有顶点之间的最短路径。

具体步骤如下：

1. 初始化一个二维数组 `dist`，用于存储任意两个顶点之间的最短距离。初始时，`dist[i][j]` 表示顶点*i*到顶点*j*的直接边的权重，如果*i*和*j*不直接相连，则权重为无穷大。
2. 对于每个顶点*k*，在更新 `dist` 数组时，考虑顶点*k*作为中间节点的情况。遍历所有的顶点对(*i, j*)，如果通过顶点*k*可以使得从顶点*i*到顶点*j*的路径变短，则更新 `dist[i][j]` 为更小的值。

```
dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
```

3. 重复进行上述步骤，对于每个顶点作为中间节点，进行迭代更新 `dist` 数组。最终，`dist` 数组中存储的就是所有顶点之间的最短路径。

Floyd-Warshall算法的时间复杂度为 $O(V^3)$ ，其中V是图中的顶点数。它适用于解决稠密图（边数较多）的最短路径问题，并且可以处理负权边和负权回路。

以下是一个使用Floyd-Warshall算法求解所有顶点之间最短路径的示例代码：

```
1 def floyd_marshall(graph):
2     n = len(graph)
3     dist = [[float('inf')] * n for _ in range(n)]
4
5     for i in range(n):
6         for j in range(n):
7             if i == j:
8                 dist[i][j] = 0
9             elif j in graph[i]:
10                dist[i][j] = graph[i][j]
11
12     for k in range(n):
13         for i in range(n):
14             for j in range(n):
15                 dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
16
17     return dist
```

在上述代码中，`graph` 是一个字典，用于表示图的邻接关系。它的键表示起始顶点，值表示一个字典，其中键表示终点顶点，值表示对应边的权重。

你可以将你的图表示为一个邻接矩阵或邻接表，并将其作为参数传递给 `floyd_marshall` 函数。函数将返回一个二维数组，其中 `dist[i][j]` 表示从顶点*i*到顶点*j*的最短路径长度。

4.4 最小生成树(MSTs)

4.4.1 有权图Prim, 无权图BFS

Prim算法与BFS（广度优先搜索）有一些相似之处，但它们解决的问题和具体实现方式有所不同。

1. 相似性:

- Prim算法和BFS都是图的遍历算法。
- 它们都从一个起始顶点开始，逐步扩展到邻居顶点，并以某种方式记录已经访问过的顶点。

2. 不同之处:

- Prim算法是一种用于解决最小生成树（MST）问题的贪心算法，它会逐步构建一个包含所有顶点的树，并且使得树的边权重之和最小。
- BFS是一种用于无权图的遍历算法，它按照层次遍历的方式访问图的所有节点，并找到从起始顶点到其他所有顶点的最短路径。
- Prim算法通过选择具有最小权重的边来扩展生成树，并且只考虑与当前生成树相邻的顶点。
- BFS通过队列来保存待访问的顶点，并按照顺序进行遍历，不考虑边的权重。

虽然Prim算法和BFS都是一种逐步扩展的遍历算法，但是它们解决的问题和具体实现方式有所不同。Prim算法用于构建最小生成树，而BFS用于寻找无权图中的最短路径。

在学习最后一个图算法之前，先考虑网络游戏设计师和互联网广播服务提供商面临的问题。他们希望高效地把信息传递给所有人。这在网络游戏中非常重要，因为所有玩家都可以据此知道其他玩家的最近位置。互联网广播也需要做到这一点，以让所有听众都接收到所需数据。图1展示了上述广播问题。

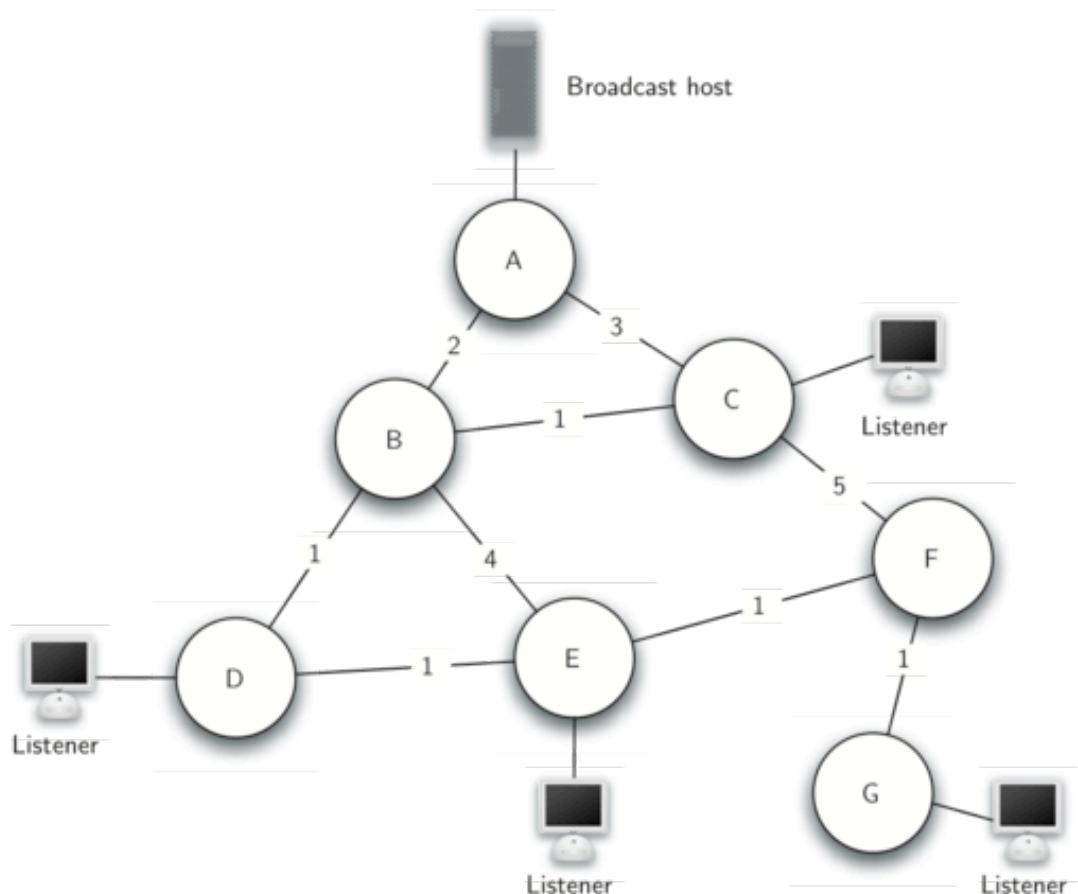


图1 广播问题

为了更好地理解上述问题，我们先来看看如何通过蛮力法求解。你稍后会看到，本节提出的解决方案为何优于蛮力法。假设互联网广播服务提供商要向所有收听者播放一条消息，最简单的方法是保存一份包含所有收听者的列表，然后向每一个收听者单独发送消息。以图1为例，若采用上述解法，则每一条消息都需要有4份副本。假设使用开销最小的路径，让我们来看看每一个路由器需要处理多少次相同的消息。

```
1 A -> C: 3
2 A -> D: 2 + 1
3 A -> E: 2 + 1 + 1
4 A -> G: 2 + 1 + 1 + 1 + 1
5
6 总费用: 3 + 3 + 4 + 6 = 16
```

从广播服务提供商发出的所有消息都会经过路由器A，因此A能够看到每一条消息的所有副本。路由器C只能看到一份副本，而由于路由器B和D在收听者1、2、3的最短路径上，因此它们能够看到每一条消息的3份副本。考虑到广播服务提供商每秒会发送数百条消息，这样做会导致流量剧增。

一种蛮力法是广播服务提供商针对每条消息只发送一份副本，然后由路由器来正确地发送。最简单的方法就是不受控泛洪法 (uncontrolled flooding)，策略如下：每一条消息都设有存活时间 (time to live, ttl)，它大于或等于广播服务提供商和最远的收听者之间的距离；每一个路由器都接收到消息的一份副本，并且将消息发送给所有的相邻路由器。在消息被发送时，它的ttl递减，直到变为0。不难发现，不受控泛洪法产生的不必要的消息比第一种方法更多。

解决广播问题的关键在于构建一棵权重最小的生成树。我们对最小生成树的正式定义如下：对于图 $G=(V, E)$ ，最小生成树T是E的无环子集，并且连接V中的所有顶点，并且T中边集合的权重之和最小。

图2展示了简化的广播图，并且突出显示了形成最小生成树的所有边。为了解决广播问题，广播服务提供商只需向网络中发送一条消息副本。每一个路由器向属于生成树的相邻路由器转发消息，其中不包括刚刚向它发送消息的路由器。在图2的例子中，需要广播的消息只需要从A开始，沿着树的路径层次向下传播，就可以达到每个路由器只需要处理1次消息，消息的传输有成本，消息的复制没有成本。A把消息转发给B，B把消息转发给C和D，D转发给E，E转发给F，F转发给G。每一个路由器都只看到任意消息的一份副本，并且所有的收听者都接收到了消息。

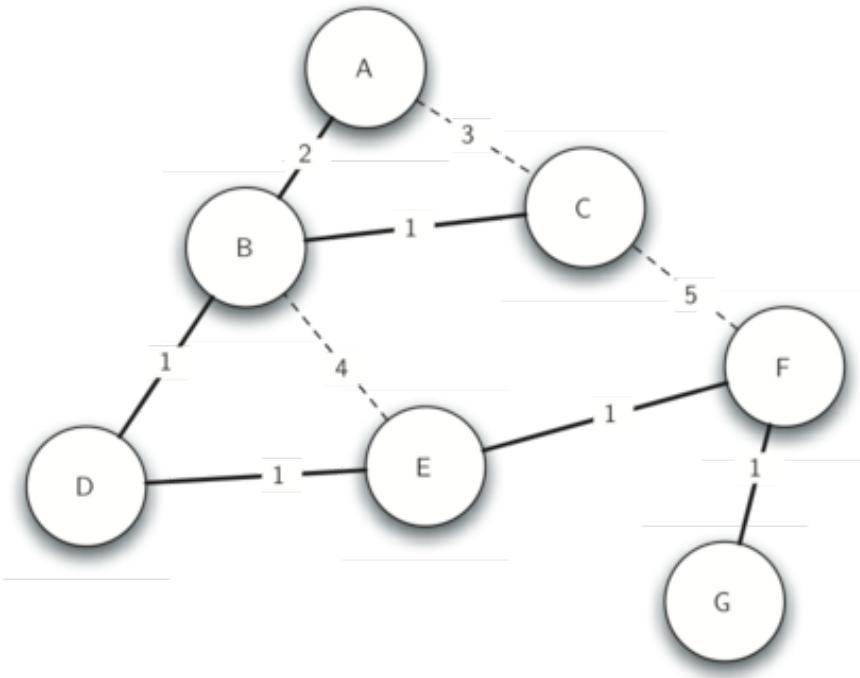


图2 广播图中的最小生成树

总权重（成本、费用）最小： $2 + 1 + 1 + 1 + 1 + 1 = 7$

上述思路对应的算法叫作Prim算法。由于每一步都选择代价最小的下一步，因此Prim算法属于一种“贪婪算法”。在这个问题中，代价最小的下一步是选择权重最小的边。接下来实现Prim算法。

构建生成树的基本思想如下：

```

1 while T is not yet a spanning tree
2   Find an edge that is safe to add to the tree
3   Add the new edge to T

```

难点在于，如何找到“可以安全添加到树中的边”。我们这样定义安全的边：它的一端是生成树中的顶点，另一端是还不在生成树中的顶点。这保证了构建的树不会出现循环。

Prim算法的Python实现如代码List1所示。与Dijkstra算法类似，Prim算法也使用了优先级队列来选择下一个添加到图中的顶点。

List1 Prim算法的Python实现

```

1 # https://github.com/psads/pythonds3
2 from pythonds3.graphs import PriorityQueue
3
4 def prim(graph,start):
5     pq = PriorityQueue()
6     for vertex in graph:

```

```
7     vertex.distance = sys.maxsize
8     vertex.previous = None
9     start.distance = 0
10    pq.buildHeap([(v.distance,v) for v in graph])
11    while pq:
12        distance, current_v = pq.delete()
13        for next_v in current_v.get_eighbors():
14            new_distance = current_v.get_neighbor(next_v)
15            if next_v in pq and new_distance < next_v.distance:
16                next_v.previous = current_v
17                next_v.distance = new_distance
18                pq.change_priority(next_v,new_distance)
```

图3展示了将Prim算法应用于示例生成树的过程。以顶点A作为起点，将A到其他所有顶点的距离都初始化为无穷大。检查A的相邻顶点后，可以更新从A到B和C的距离，因为实际的距离小于无穷大。更新距离之后，B和C被移到优先级队列的头部。并且，它们的前驱顶点被设置为A。注意，我们还没有把B和C添加到生成树中。只有在从优先级队列中移除时，顶点才会被添加到生成树中。

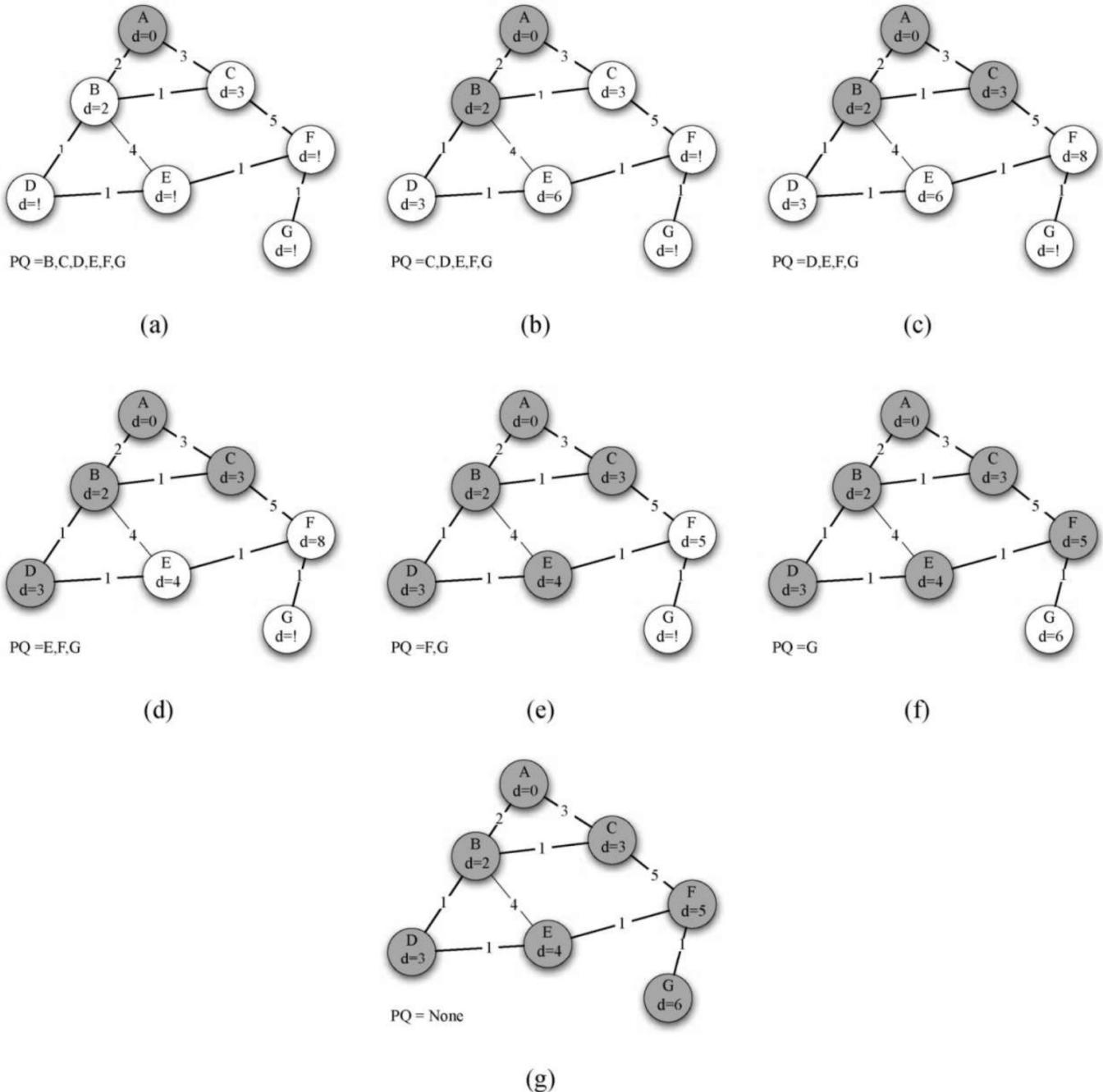


图3 Prim算法的应用过程

由于到B的距离最短，因此接下来检查B的相邻顶点。检查后发现，可以更新D和E。接下来处理优先级队列中的下一个顶点C。与C相邻的唯一一个还在优先级队列中的顶点是F，因此更新到F的距离，并且调整F在优先级队列中的位置。

现在检查与D相邻的顶点，发现可以将到E的距离从6减少为4。修改距离的同时，把E的前驱顶点改为D，以此准备将E添加到生成树中的另一个位置。Prim算法正是通过这样的方式将每一个顶点都添加到生成树中。

4.4.2 Kruskal and Disjoint Set

Kruskal算法通常与并查集（Disjoint Set）结构一起使用，但它们并不是同一种算法。

1. Kruskal算法：

- Kruskal算法是一种用于解决最小生成树（MST）问题的贪心算法。它通过不断选择具有最小权重的边，并确保选择的边不形成环，最终构建出一个包含所有顶点的最小生成树。
- 在Kruskal算法中，通常会使用并查集来维护图中顶点的连通性信息。当选择一条边时，通过并查集判断该边的两个端点是否属于同一个连通分量，以避免形成环。

2. 并查集（Disjoint Set）：

- 并查集是一种数据结构，用于管理元素的不相交集合。它通常支持两种操作：查找（Find）和合并（Union）。查找操作用于确定某个元素属于哪个集合，合并操作用于将两个集合合并为一个集合。
- 在Kruskal算法中，我们可以使用并查集来快速判断两个顶点是否属于同一个连通分量。当我们遍历边并选择加入最小生成树时，可以通过并查集来检查该边的两个端点是否已经在同一个连通分量中，以避免形成环。

因此，尽管Kruskal算法通常与并查集结合使用，但它们是两个不同的概念。Kruskal算法是解决最小生成树问题的一种算法，而并查集是一种数据结构，用于管理元素的不相交集合。

- Approach: Kruskal's algorithm sorts all the edges in the graph by their weights and then iteratively adds the edges with the minimum weight as long as they do not create a cycle in the MST.
- Suitable for: Kruskal's algorithm is often used when the graph is sparse or when the number of edges is much smaller than the number of vertices. It is efficient for finding the MST in such cases.
- Connectivity: Kruskal's algorithm may produce a forest of MSTs initially, and then it merges them into a single MST.

Key similarities and connections between Prim's and Kruskal's algorithms:

- Both algorithms find the minimum spanning tree of a graph.
- They are both greedy algorithms that make locally optimal choices in each step to achieve the overall minimum weight.
- The resulting MSTs produced by both algorithms have the same total weight.

In summary, you can choose between Prim's algorithm and Kruskal's algorithm based on the characteristics of the graph, such as density or sparsity, and the specific requirements of your problem.

Kruskal算法是一种用于解决最小生成树（Minimum Spanning Tree, 简称MST）问题的贪心算法。给定一个连通的带权无向图，Kruskal算法可以找到一个包含所有顶点的最小生成树，即包含所有顶点且边权重之和最小的树。

以下是Kruskal算法的基本步骤：

1. 将图中的所有边按照权重从小到大进行排序。
2. 初始化一个空的边集，用于存储最小生成树的边。
3. 重复以下步骤，直到边集中的边数等于顶点数减一或者所有边都已经考虑完毕：
 - 选择排序后的边集中权重最小的边。
 - 如果选择的边不会导致形成环路（即加入该边后，两个顶点不在同一个连通分量中），则将该边加入最小生成树的边集中。
4. 返回最小生成树的边集作为结果。

Kruskal算法的核心思想是通过不断选择权重最小的边，并判断是否会造成环路来构建最小生成树。算法开始时，每个顶点都是一个独立的连通分量，随着边的不断加入，不同的连通分量逐渐合并为一个连通分量，直到最终形成最小生成树。

实现Kruskal算法时，一种常用的数据结构是并查集（Disjoint Set）。并查集可以高效地判断两个顶点是否在同一个连通分量中，并将不同的连通分量合并。

下面是一个使用Kruskal算法求解最小生成树的示例代码：

```
1 class DisjointSet:
2     def __init__(self, num_vertices):
3         self.parent = list(range(num_vertices))
4         self.rank = [0] * num_vertices
5
6     def find(self, x):
7         if self.parent[x] != x:
8             self.parent[x] = self.find(self.parent[x])
9         return self.parent[x]
10
11    def union(self, x, y):
12        root_x = self.find(x)
13        root_y = self.find(y)
14
15        if root_x != root_y:
16            if self.rank[root_x] < self.rank[root_y]:
17                self.parent[root_x] = root_y
18            elif self.rank[root_x] > self.rank[root_y]:
19                self.parent[root_y] = root_x
20            else:
21                self.parent[root_x] = root_y
22                self.rank[root_y] += 1
23
24
25    def kruskal(graph):
26        num_vertices = len(graph)
27        edges = []
28
29        # 构建边集
30        for i in range(num_vertices):
31            for j in range(i + 1, num_vertices):
32                if graph[i][j] != 0:
33                    edges.append((i, j, graph[i][j]))
34
35        # 按照权重排序
36        edges.sort(key=lambda x: x[2])
37
38        # 初始化并查集
39        disjoint_set = DisjointSet(num_vertices)
40
41        # 构建最小生成树的边集
42        minimum_spanning_tree = []
```

```

44     for edge in edges:
45         u, v, weight = edge
46         if disjoint_set.find(u) != disjoint_set.find(v):
47             disjoint_set.union(u, v)
48             minimum_spanning_tree.append((u, v, weight))
49
50     return minimum_spanning_tree

```

在上述代码中，`graph` 是一个二维矩阵，表示带权无向图的邻接矩阵。`graph[i][j]` 表示顶点 i 和顶点 j 之间的边的权重。

Kruskal 算法的时间复杂度为 $O(E \log E)$ ，其中 E 是边的数量。排序边集的时间复杂度为 $O(E \log E)$ ，并查集操作的时间复杂度为 $O(\alpha(V))$ ，其中 α 是 Ackermann 函数的反函数，近似为常数。因此，总体上来说，Kruskal 算法的时间复杂度可以近似为 $O(E \log E)$ 。

4.5 Dijkstra 和 Prim 实现

4.5.1 通常的Dijkstra实现

使用 `heapq` 来实现 Dijkstra 算法的完整 Python 代码。这个实现包括了图的类表示，顶点类，以及 Dijkstra 算法的具体逻辑。

通过维护一个 `visited` 集合，我们可以确保每个顶点只被处理一次。

这个实现也包括了输出每个顶点的最短距离和从起始点到每个顶点的具体路径。

使用 `heapq` 模块确实是一个好选择，因为它通常比 `PriorityQueue` 更为高效。

`PriorityQueue`，即 Python 标准库中 `queue.PriorityQueue`，其底层实际上是基于 `heapq` 实现的。就像 `heapq`，`PriorityQueue` 也没有直接支持修改队列中元素优先级的内置方法。`PriorityQueue` 提供线程安全的队列操作，适用于多线程程序，但它并不支持如 `decrease-key`` 操作，这种操作在很多图算法中非常有用。

要实现修改优先级的功能，你可以采用与 `heapq` 类似的策略：将新的优先级作为一个新的条目加入到队列中，并通过一种机制（比如标记或记录）忽略或移除旧的条目。这种方法在 `PriorityQueue` 中同样适用，但要注意它的线程安全性和性能影响。

Dijkstra 算法：Dijkstra 算法用于解决单源最短路径问题，即从给定源节点到图中所有其他节点的最短路径。算法的基本思想是通过不断扩展离源节点最近的节点来逐步确定最短路径。具体步骤如下：

- 初始化一个距离数组，用于记录源节点到所有其他节点的最短距离。初始时，源节点的距离为 0，其他节点的距离为无穷大。
- 选择一个未访问的节点中距离最小的节点作为当前节点。
- 更新当前节点的邻居节点的距离，如果通过当前节点到达邻居节点的路径比已知最短路径更短，则更新最短路径。
- 标记当前节点为已访问。
- 重复上述步骤，直到所有节点都被访问或者所有节点的最短路径都被确定。

Dijkstra 算法的时间复杂度为 $O(V^2)$ ，其中 V 是图中的节点数。当使用优先队列（如最小堆）来选择距离最小的节点时，可以将时间复杂度优化到 $O((V+E)\log V)$ ，其中 E 是图中的边数。

Dijkstra.py 程序在 <https://github.com/GMyhf/2024spring-cs201/tree/main/code>

```
1 import heapq
2 import sys
3
4 class Vertex:
5     def __init__(self, key):
6         self.id = key
7         self.connectedTo = {}
8         self.distance = sys.maxsize
9         self.pred = None
10
11    def addNeighbor(self, nbr, weight=0):
12        self.connectedTo[nbr] = weight
13
14    def getConnections(self):
15        return self.connectedTo.keys()
16
17    def getWeight(self, nbr):
18        return self.connectedTo[nbr]
19
20    def __lt__(self, other):
21        return self.distance < other.distance
22
23 class Graph:
24     def __init__(self):
25         self.vertList = {}
26         self.numVertices = 0
27
28     def addVertex(self, key):
29         newVertex = Vertex(key)
30         self.vertList[key] = newVertex
31         self.numVertices += 1
32         return newVertex
33
34     def getVertex(self, n):
35         return self.vertList.get(n)
36
37     def addEdge(self, f, t, cost=0):
38         if f not in self.vertList:
39             self.addVertex(f)
40         if t not in self.vertList:
41             self.addVertex(t)
42         self.vertList[f].addNeighbor(self.vertList[t], cost)
43
44 def dijkstra(graph, start):
45     pq = []
46     start.distance = 0
47     heapq.heappush(pq, (0, start))
48     visited = set()
49
50     while pq:
```

```

51     currentDist, currentVert = heapq.heappop(pq)      # 当一个顶点的最短路径确定后（也
就是这个顶点
52                                         # 从优先队列中被弹出时），它的最
53                                         # 短路径不会再改变。
54     if currentVert in visited:
55         continue
56     visited.add(currentVert)
57
58     for nextVert in currentVert.getConnections():
59         newDist = currentDist + currentVert.getWeight(nextVert)
60         if newDist < nextVert.distance:
61             nextVert.distance = newDist
62             nextVert.pred = currentVert
63             heapq.heappush(pq, (newDist, nextVert))
64
65 # 创建图和边
66 g = Graph()
67 g.addEdge('A', 'B', 4)
68 g.addEdge('A', 'C', 2)
69 g.addEdge('C', 'B', 1)
70 g.addEdge('B', 'D', 2)
71 g.addEdge('C', 'D', 5)
72 g.addEdge('D', 'E', 3)
73 g.addEdge('E', 'F', 1)
74 g.addEdge('D', 'F', 6)
75
76 # 执行 Dijkstra 算法
77 print("Shortest Path Tree:")
78 dijkstra(g, g.getVertex('A'))
79
80 # 输出最短路径树的顶点及其距离
81 for vertex in g.vertList.values():
82     print(f"Vertex: {vertex.id}, Distance: {vertex.distance}")
83
84 # 输出最短路径到每个顶点
85 def printPath(vert):
86     if vert.pred:
87         printPath(vert.pred)
88         print(" -> ", end="")
89     print(vert.id, end="")
90
91 print("\nPaths from Start Vertex 'A':")
92 for vertex in g.vertList.values():
93     print(f"Path to {vertex.id}: ", end="")
94     printPath(vertex)
95     print(", Distance: ", vertex.distance)
96
97 """
98 Shortest Path Tree:
99 Vertex: A, Distance: 0
100 Vertex: B, Distance: 3
101 Vertex: C, Distance: 2

```

```

101 Vertex: D, Distance: 5
102 Vertex: E, Distance: 8
103 Vertex: F, Distance: 9
104
105 Paths from Start Vertex 'A':
106 Path to A: A, Distance: 0
107 Path to B: A -> C -> B, Distance: 3
108 Path to C: A -> C, Distance: 2
109 Path to D: A -> C -> B -> D, Distance: 5
110 Path to E: A -> C -> B -> D -> E, Distance: 8
111 Path to F: A -> C -> B -> D -> E -> F, Distance: 9
112 """

```

4.5.2 通常的Prim实现

Prim's algorithm and Kruskal's algorithm are both used to find the minimum spanning tree (MST) of a connected, weighted graph. However, they have different approaches and are suitable for different scenarios. Here are the key differences and the typical use cases for each algorithm:

Prim's Algorithm:

- Approach: Prim's algorithm starts with a single vertex and gradually grows the MST by iteratively adding the edge with the minimum weight that connects a vertex in the MST to a vertex outside the MST.
- Suitable for: Prim's algorithm is often used when the graph is dense or when the number of edges is close to the number of vertices. It is efficient for finding the MST in such cases.
- Connectivity: Prim's algorithm always produces a connected MST.

通过维护一个 `visited` 集合，我们可以确保每个顶点只被处理一次。

```

1 import sys
2 import heapq
3
4 class Vertex:
5     def __init__(self, key):
6         self.id = key
7         self.connectedTo = {}
8         self.distance = sys.maxsize
9         self.pred = None
10
11     def addNeighbor(self, nbr, weight=0):
12         self.connectedTo[nbr] = weight
13
14     def getConnections(self):
15         return self.connectedTo.keys()
16
17     def getWeight(self, nbr):
18         return self.connectedTo[nbr]

```

```

19
20     def __lt__(self, other):
21         return self.distance < other.distance
22
23 class Graph:
24     def __init__(self):
25         self.vertList = {}
26         self.numVertices = 0
27
28     def addVertex(self, key):
29         newVertex = Vertex(key)
30         self.vertList[key] = newVertex
31         self.numVertices += 1
32         return newVertex
33
34     def getVertex(self, n):
35         return self.vertList.get(n)
36
37     def addEdge(self, f, t, cost=0):
38         if f not in self.vertList:
39             self.addVertex(f)
40         if t not in self.vertList:
41             self.addVertex(t)
42         self.vertList[f].addNeighbor(self.vertList[t], cost)
43         self.vertList[t].addNeighbor(self.vertList[f], cost)
44
45 def prim(graph, start):
46     pq = []
47     start.distance = 0
48     heapq.heappush(pq, (0, start))
49     visited = set()
50
51     while pq:
52         currentDist, currentVert = heapq.heappop(pq)
53         if currentVert in visited:
54             continue
55         visited.add(currentVert)
56
57         for nextVert in currentVert.getConnections():
58             weight = currentVert.getWeight(nextVert)
59             if nextVert not in visited and weight < nextVert.distance:
60                 nextVert.distance = weight
61                 nextVert.pred = currentVert
62                 heapq.heappush(pq, (weight, nextVert))
63
64 # 创建图和边
65 g = Graph()
66 g.addEdge('A', 'B', 4)
67 g.addEdge('A', 'C', 3)
68 g.addEdge('C', 'B', 1)
69 g.addEdge('C', 'D', 2)
70 g.addEdge('D', 'B', 5)

```

```

71 g.addEdge('D', 'E', 6)
72
73 # 执行 Prim 算法
74 print("Minimum Spanning Tree:")
75 prim(g, g.getVertex('A'))
76
77 # 输出最小生成树的边
78 for vertex in g.vertList.values():
79     if vertex.pred:
80         print(f'{vertex.pred.id} -> {vertex.id} Weight:{vertex.distance}')
81
82 """
83 Minimum Spanning Tree:
84 C -> B Weight:1
85 A -> C Weight:3
86 C -> D Weight:2
87 D -> E Weight:6
88 """

```

4.5.3 书上Dijkstra实现，不敢恭维

在 `heapq` 中，直接改变一个元素的优先级并重新排序堆并不是直接支持的功能，因为 `heapq` 模块提供的是一个简单的堆实现，而不是一个优先级队列。然而，你可以通过一种变通的方法来模拟这个功能。

一个常用的技巧是将被更新优先级的元素重新插入到堆中，但标记原有的元素为失效。具体到 Dijkstra 算法中，这意味着当你发现到某个顶点的更短路径时，你将这个顶点与新的距离一起推入堆中，并通过一个字典或集合来跟踪最新的有效状态。

下面的代码展示了这种技术的具体实现。Book_Dijkstra.py 在<https://github.com/GMyhf/2024spring-cs201/tree/main/code>

```

1 import sys
2 from heapq import heappop, heappush, heapify
3
4 class Vertex:
5     def __init__(self, key):
6         self.key = key
7         self.neighbors = {}
8         self.distance = sys.maxsize
9         self.previous = None
10        self.color = None
11
12    def get_neighbor(self, other):
13        return self.neighbors.get(other, None)
14
15    def set_neighbor(self, other, weight=0):
16        self.neighbors[other] = weight
17
18    def __repr__(self):
19        return f"Vertex({self.key})"

```

```

20
21     def __str__(self):
22         return (
23             f"{self.key} connected to: "
24             + f"\n{[x.key for x in self.neighbors]}"
25         )
26
27     def get_neighbors(self):
28         return self.neighbors.keys()
29
30     def get_key(self):
31         return self.key
32
33     def __eq__(self, other):
34         if isinstance(other, Vertex):
35             return self.key == other.key
36         return False
37
38     def __lt__(self, other):
39         if isinstance(other, Vertex):
40             return self.distance < other.distance
41         return False
42
43     def __hash__(self):
44         return hash(self.key)
45
46
47 class Graph:
48     def __init__(self):
49         self.vertices = {}
50
51     def set_vertex(self, key):
52         self.vertices[key] = Vertex(key)
53
54     def get_vertex(self, key):
55         return self.vertices.get(key, None)
56
57     def __contains__(self, key):
58         return key in self.vertices
59
60     def add_edge(self, from_vert, to_vert, weight=0):
61         if from_vert not in self.vertices:
62             self.set_vertex(from_vert)
63         if to_vert not in self.vertices:
64             self.set_vertex(to_vert)
65         self.vertices[from_vert].set_neighbor(
66             self.vertices[to_vert], weight
67         )
68
69     def get_vertices(self):
70         return self.vertices.keys()
71

```

```

72     def __iter__(self):
73         return iter(self.vertices.values())
74
75
76 # print("\n---Graph---\n")
77 # g = Graph()
78 # for i in range(6):
79 #     g.set_vertex(i)
80 # print(g.vertices)
81 # g.add_edge(0, 1, 5)
82 # g.add_edge(0, 5, 2)
83 # g.add_edge(1, 2, 4)
84 # g.add_edge(2, 3, 9)
85 # g.add_edge(3, 4, 7)
86 # g.add_edge(3, 5, 3)
87 # g.add_edge(4, 0, 1)
88 # g.add_edge(5, 4, 8)
89 # g.add_edge(5, 2, 1)
90 # for v in g:
91 #     for w in v.get_neighbors():
92 #         print(f"({v.get_key()}), {w.get_key()})")
93
94
95 """
96
97 这个实现使用了一个字典 visited 来跟踪每个顶点的最短已知距离。
98 如果在优先队列中发现一个顶点的旧记录，通过检查 visited 中记录的距离来决定是否忽略它。
99 这样可以确保每个顶点的最新距离总是被正确处理，即便它被多次推入堆中。
100 """
101 def dijkstra(graph, start):
102     pq = [(v.distance, v) for v in graph]
103     start.distance = 0
104     heapify(pq)
105     visited = {}
106     while pq:
107         distance, current_v = heappop(pq)
108         if current_v in visited and visited[current_v] < distance:
109             continue
110         for next_v in current_v.get_neighbors():
111             new_distance = (
112                 current_v.distance
113                 + current_v.get_neighbor(next_v)
114             )
115             if new_distance < next_v.distance:
116                 next_v.distance = new_distance
117                 next_v.previous = current_v
118                 heappush(pq, (next_v.distance, next_v))
119                 print("".join(f"{v.distance % 1000:<5d}" for v in graph))
120
121 print("\n---Dijkstra's---\n")
122 g = Graph()
123 vertices = ["u", "v", "w", "x", "y", "z"]

```

```

124 for v in vertices:
125     g.set_vertex(v)
126 g.add_edge("u", "v", 2)
127 g.add_edge("u", "w", 5)
128 g.add_edge("u", "x", 1)
129 g.add_edge("v", "u", 2)
130 g.add_edge("v", "w", 3)
131 g.add_edge("v", "x", 1)
132 g.add_edge("w", "u", 5)
133 g.add_edge("w", "v", 3)
134 g.add_edge("w", "x", 3)
135 g.add_edge("w", "y", 1)
136 g.add_edge("w", "z", 5)
137 g.add_edge("x", "u", 1)
138 #g.add_edge("x", "v", 2)
139 g.add_edge("x", "v", 1)
140 g.add_edge("x", "w", 3)
141 g.add_edge("x", "y", 1)
142 g.add_edge("y", "w", 1)
143 g.add_edge("y", "x", 1)
144 g.add_edge("y", "z", 1)
145 g.add_edge("z", "w", 5)
146 g.add_edge("z", "y", 1)
147 print("".join(f"{v:5s}" for v in vertices))
148 dijkstra(g, g.get_vertex("u"))
149 print(
150     "".join(
151         f"{g.get_vertex(v).distance:<5d}"
152         for v in vertices
153     )
154 )
155 """
156 """
157 ---Dijkstra's---
158
159 u    v    w    x    y    z
160 0    2    807  807  807  807
161 0    2    5    807  807  807
162 0    2    5    1    807  807
163 0    2    4    1    807  807
164 0    2    4    1    2    807
165 0    2    3    1    2    807
166 0    2    3    1    2    3
167 0    2    3    1    2    3
168 """
169

```

4.5.4 书上Prim实现，不敢恭维

Book_Prim.py在<https://github.com/GMyhf/2024spring-cs201/tree/main/code>

```

1 import sys
2 import heapq
3
4 class Vertex:
5     def __init__(self, key):
6         self.id = key
7         self.connectedTo = {}
8         self.distance = sys.maxsize
9         self.pred = None
10
11    def addNeighbor(self, nbr, weight=0):
12        self.connectedTo[nbr] = weight
13
14    def getConnections(self):
15        return self.connectedTo.keys()
16
17    def getWeight(self, nbr):
18        return self.connectedTo[nbr]
19
20    def __lt__(self, other):
21        return self.distance < other.distance
22
23 class Graph:
24    def __init__(self):
25        self.vertList = {}
26        self.numVertices = 0
27
28    def addVertex(self, key):
29        newVertex = Vertex(key)
30        self.vertList[key] = newVertex
31        self.numVertices += 1
32        return newVertex
33
34    def getVertex(self, n):
35        return self.vertList.get(n)
36
37    def addEdge(self, f, t, cost=0):
38        if f not in self.vertList:
39            self.addVertex(f)
40        if t not in self.vertList:
41            self.addVertex(t)
42        self.vertList[f].addNeighbor(self.vertList[t], cost)
43        self.vertList[t].addNeighbor(self.vertList[f], cost)
44
45    def prim(graph, start):
46        pq = []
47        start.distance = 0
48        heapq.heappush(pq, (0, start))
49        visited = set()
50
51        while pq:
52            currentDist, currentVert = heapq.heappop(pq)

```

```

53     if currentVert in visited:
54         continue
55     visited.add(currentVert)
56
57     for nextVert in currentVert.getConnections():
58         weight = currentVert.getWeight(nextVert)
59         if nextVert not in visited and weight < nextVert.distance:
60             nextVert.distance = weight
61             nextVert.pred = currentVert
62             heapq.heappush(pq, (weight, nextVert))
63
64 # 创建图和边
65 g = Graph()
66 g.addEdge('A', 'B', 4)
67 g.addEdge('A', 'C', 3)
68 g.addEdge('C', 'B', 1)
69 g.addEdge('C', 'D', 2)
70 g.addEdge('D', 'B', 5)
71 g.addEdge('D', 'E', 6)
72
73 # 执行 Prim 算法
74 print("Minimum Spanning Tree:")
75 prim(g, g.getVertex('A'))
76
77 # 输出最小生成树的边
78 for vertex in g.vertList.values():
79     if vertex.pred:
80         print(f'{vertex.pred.id} -> {vertex.id} Weight:{vertex.distance}')
81
82 """
83 Minimum Spanning Tree:
84 C -> B Weight:1
85 A -> C Weight:3
86 C -> D Weight:2
87 D -> E Weight:6
88 """

```

4.6 对3.1~4.5小结

介绍了图的抽象数据类型，以及一些实现方式。如果能将一个问题用图表示出来，那么就可以利用图算法加以解决。对于解决下列问题，图非常有用。

- 利用宽度优先搜索找到无权重的最短路径。
- 利用Dijkstra算法求解带权重的最短路径。
- 利用深度优先搜索来探索图。
- 利用强连通单元来简化图。
- 利用拓扑排序为任务排序。
- 利用最小生成树广播消息。

*关键路径

在数据结构中，关键路径算法通常与有向加权图（有向图中每条边都有一个权重）相关。一种常用的关键路径算法是AOV 网络关键路径算法（Activity On Vertex Network Critical Path Algorithm），它适用于没有环路的有向加权图。

以下是 AOV 网络关键路径算法的基本步骤：

1. 根据项目的活动和依赖关系，构建有向加权图。图的顶点表示活动，边表示活动之间的依赖关系，边的权重表示活动的持续时间。
2. 对图进行拓扑排序，以确定活动的执行顺序。拓扑排序可以使用 Kahn 算法来实现。
3. 初始化两个数组：`earliest_start_time` 和 `latest_finish_time`，分别用于存储每个顶点的最早开始时间和最晚完成时间。
4. 从拓扑排序的第一个顶点开始，按照拓扑排序的顺序遍历每个顶点。
 - 对于当前顶点 u ，计算其最早开始时间 `earliest_start_time[u]`，即前面所有依赖顶点的最晚完成时间中的最大值加上 u 的持续时间。
5. 从拓扑排序的最后一个顶点开始，按照逆拓扑排序的顺序遍历每个顶点。
 - 对于当前顶点 v ，计算其最晚完成时间 `latest_finish_time[v]`，即后面所有依赖顶点的最早开始时间中的最小值减去 v 的持续时间。
6. 对于每条边 (u, v) ，计算其总时差（Total Float）：
 - 总时差等于 `latest_finish_time[v] - earliest_start_time[u] - edge_weight(u, v)`。
7. 找到总时差为 0 的边，这些边构成了关键路径。关键路径上的活动是项目的关键活动，任何关键活动的延迟都会导致项目延迟。

关键路径算法可以使用图的邻接表或邻接矩阵来表示有向加权图，并以此作为输入进行计算。通过计算关键路径，可以确定项目的关键活动和项目的最长完成时间，有助于项目管理和资源分配。

请注意，这里介绍的是一种常见的关键路径算法，其他算法和技术也可用于求解关键路径问题，具体选择取决于实际情况和需求。

4.7 编程题目

sy403: 先导课程 中等

<https://sunnywhy.com/sfbj/10/6/403>

现有 n 门课程（假设课程编号为从 0 到 $n-1$ ），课程之间有依赖关系，即可能存在两门课程，必须学完其中一门才能学另一门。现在给出个依赖关系，问能否把所有课程都学完。

注：能同时学习多门课程时总是先学习编号最小的课程。

输入

第一行两个整数n、m ($1 \leq n \leq 100, 0 \leq m \leq n(n - 1)$)，分别表示顶点数、边数；

接下来m行，每行两个整数u、v ($0 \leq u \leq n - 1, 0 \leq v \leq n - 1, u \neq v$)，表示一条边的起点和终点的编号。数据保证不会有重边。

输出

如果能学完所有课程，那么输出一行 yes，然后在第二行输出学习课程编号的顺序，编号之间用空格隔开，行末不允许有多余的空格；如果不能学完所有课程，那么输出一行 no，然后在第二行输出不能学习的课程门数。

样例1

输入

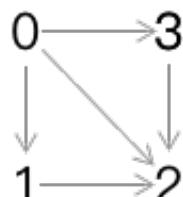
1	4 5
2	0 1
3	0 2
4	0 3
5	1 2
6	3 2

输出

1	yes
2	0 1 3 2

解释

对应的依赖关系如下图所示。由于每次选择编号最小的顶点，因此学习顺序为 0 1 3 2。



样例2

输入

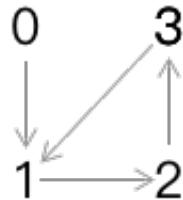
1	4 4
2	0 1
3	1 2
4	2 3
5	3 1

输出

1	No
2	3

解释

对应的依赖关系如下图所示。编号为 0 的课程可以直接学习；由于编号 1、2、3 的课程互相依赖，因此无法学习的课程数为 3。



这个问题可以通过使用拓扑排序来解决。拓扑排序是对有向无环图的顶点的一种排序，它使得如果存在一条从顶点 u 到顶点 v 的路径，那么在排序中 u 一定在 v 的前面。如果在进行拓扑排序的过程中，发现存在没有被访问的顶点，但是已经没有入度为 0 的顶点，那么就说明图中存在环，也就是存在不能学习的课程。

以下是使用拓扑排序判断有向图是否存在环并输出学习课程顺序的Python代码：

```
1 from collections import defaultdict
2
3 def courseSchedule(n, edges):
4     graph = defaultdict(list)
5     indegree = [0] * n
6     for u, v in edges:
7         graph[u].append(v)
8         indegree[v] += 1
9
10    queue = [i for i in range(n) if indegree[i] == 0]
11    queue.sort()
12    result = []
13
14    while queue:
15        u = queue.pop(0)
16        result.append(u)
17        for v in graph[u]:
18            indegree[v] -= 1
19            if indegree[v] == 0:
20                queue.append(v)
21        queue.sort()
22
23    if len(result) == n:
24        return "Yes", result
25    else:
26        return "No", n - len(result)
27
28 n, m = map(int, input().split())
29 edges = [list(map(int, input().split())) for _ in range(m)]
30 res, courses = courseSchedule(n, edges)
31 print(res)
```

```
32 if res == "Yes":  
33     print(*courses)  
34 else:  
35     print(courses)
```

在这段代码中，我们首先构建了一个图，并计算了每个顶点的入度。然后我们将所有入度为0的顶点添加到队列中，并对队列进行排序。然后在循环中，我们每次从队列中取出一个顶点，将其添加到结果中，然后遍历其所有的邻接顶点，将邻接顶点的入度减1，如果邻接顶点的入度变为0，那么就将其添加到队列中。最后，我们检查访问的顶点数是否等于总的顶点数，如果等于，那么说明可以学完所有课程，否则说明存在不能学习的课程。

sy386: 最短距离 简单

<https://sunnywhy.com/sfbj/10/4/386>

现有一个共n个顶点（代表城市）、m条边（代表道路）的无向图（假设顶点编号为从0到n-1），每条边有各自的边权，代表两个城市之间的距离。求从s号城市出发到达t号城市的最短距离。

输入

第一行四个整数n、m、s、t ($1 \leq n \leq 100, 0 \leq m \leq \frac{n(n-1)}{2}, 0 \leq s \leq n - 1, 0 \leq t \leq n - 1$)，分别表示顶点数、边数、起始编号、终点编号；

接下来m行，每行三个整数u、v、w ($0 \leq u \leq n - 1, 0 \leq v \leq n - 1, u \neq v, 1 \leq w \leq 100$)，表示一条边的两个端点的编号及边权距离。数据保证不会有重边。

输出

输出一个整数，表示最短距离。如果无法到达，那么输出-1。

样例1

输入

```
1 6 6 0 2  
2 0 1 2  
3 0 2 5  
4 0 3 1  
5 2 3 2  
6 1 2 1  
7 4 5 1
```

输出

```
1 | 3
```

解释

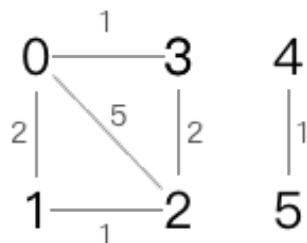
对应的无向图如下图所示。

共有3条从0号顶点到2号顶点的路径：

1. 0->3->2： 距离为3；

2. `0->2` : 距离为 5 ;
3. `0->1->2` : 距离为 3 。

因此最短距离为 3 。



样例2

输入

1	6	6	0	5
2	0	1	2	
3	0	2	5	
4	0	3	1	
5	2	3	2	
6	1	2	1	
7	4	5	1	

输出

1	-1
---	----

解释

和第一个样例相同的图，终点换成了 5 号顶点，显然从 0 号无法到达 5 号。

需要找到从给定的起始城市到目标城市的最短距离。可以使用 Dijkstra 算法来解决这个问题。Dijkstra 算法是一种用于在图中找到最短路径的算法。它从起始节点开始，然后逐步扩展到所有可达的节点，每次选择当前最短的路径进行扩展。

以下是使用 Python 实现 Dijkstra 算法来解决这个问题的示例代码：

```

1 import heapq
2
3 def dijkstra(n, edges, s, t):
4     graph = [[] for _ in range(n)]
5     for u, v, w in edges:
6         graph[u].append((v, w))
7         graph[v].append((u, w))
8
9     pq = [(0, s)] # (distance, node)
10    visited = set()
11    distances = [float('inf')] * n
12    distances[s] = 0
  
```

```

13
14     while pq:
15         dist, node = heapq.heappop(pq)
16         if node == t:
17             return dist
18         if node in visited:
19             continue
20         visited.add(node)
21         for neighbor, weight in graph[node]:
22             if neighbor not in visited:
23                 new_dist = dist + weight
24                 if new_dist < distances[neighbor]:
25                     distances[neighbor] = new_dist
26                     heapq.heappush(pq, (new_dist, neighbor))
27
28
29 # Read input
30 n, m, s, t = map(int, input().split())
31 edges = [list(map(int, input().split())) for _ in range(m)]
32
33 # Solve the problem and print the result
34 result = dijkstra(n, edges, s, t)
35 print(result)

```

这段代码实现了 Dijkstra 算法来求解从起点到终点的最短路径。首先构建了一个图，然后使用优先队列来选择下一个要探索的节点，并在探索过程中更新最短距离。最后返回从起点到终点的最短距离。

这个版本的Dijkstra算法使用了一个集合 `visited` 来记录已经访问过的节点，这样可以避免对同一个节点的重复处理。当我们从优先队列中取出一个节点时，如果这个节点已经在 `visited` 集合中，那么我们就跳过这个节点，处理下一个节点。这样可以提高算法的效率。

此外，这个版本的Dijkstra算法还在找到目标节点 `t` 时就立即返回结果，而不是等到遍历完所有节点。这是因为 Dijkstra 算法保证了每次从优先队列中取出的节点就是当前距离最短的节点，所以当我们找到目标节点 `t` 时，就已经找到了从起始节点 `s` 到 `t` 的最短路径，无需再继续搜索。

这个版本的Dijkstra算法的时间复杂度仍然是 $O(V+E)\log V$ ，其中 V 是顶点数， E 是边数。这是因为每个节点最多会被加入到优先队列中一次（当找到一条更短的路径时），并且每条边都会被处理一次（在遍历节点的邻居时）。优先队列的插入和删除操作的时间复杂度都是 $O(\log V)$ ，所以总的时间复杂度是 $O((V+E)\log V)$ 。

Dijkstra 算法是一种经典的图算法，它综合运用了多种技术，包括邻接表、集合、优先队列（堆）、贪心算法和动态规划的思想。例题：最短距离，<https://sunnywhy.com/sfbj/10/4/386>

- 邻接表：Dijkstra 算法通常使用邻接表来表示图的结构，这样可以高效地存储图中的节点和边。
- 集合：在算法中需要跟踪已经访问过的节点，以避免重复访问，这一般使用集合（或哈希集合）来实现。
- 优先队列（堆）：Dijkstra 算法中需要选择下一个要探索的节点，通常使用优先队列（堆）来维护当前候选节点的集合，并确保每次都能快速找到距离起点最近的节点。
- 贪心算法：Dijkstra 算法每次选择距离起点最近的节点作为下一个要探索的节点，这是一种贪心策略，即每次做出局部最优的选择，期望最终能达到全局最优。

- 动态规划：Dijkstra 算法通过不断地更新节点的最短距离来逐步得到从起点到各个节点的最短路径，这是一种动态规划的思想，即将原问题拆解成若干子问题，并以最优子结构来解决。

综合运用这些技术，Dijkstra 算法能够高效地求解单源最短路径问题，对于解决许多实际问题具有重要意义。

sy396: 最小生成树-Prim算法 简单

<https://sunnywhy.com/sfbj/10/5/396>

现有一个共 n 个顶点、 m 条边的无向图（假设顶点编号为从 0 到 $n-1$ ），每条边有各自的边权。在图中寻找一棵树，使得这棵树包含图上所有顶点、所有边都是图上的边，且树上所有边的边权之和最小。使用Prim算法求出这个边权之和的最小值。

输入

第一行两个整数 n 、 m ($1 \leq n \leq 100, 0 \leq m \leq \frac{n(n-1)}{2}$)，分别表示顶点数、边数；

接下来 m 行，每行三个整数 u 、 v 、 w ($0 \leq u \leq n-1, 0 \leq v \leq n-1, u \neq v, 1 \leq w \leq 100$)，表示一条边的两个端点的编号及边权距离。数据保证不会有重边。

输出

输出一个整数，表示最小的边权之和。如果不存在这样的树，那么输出 -1。

样例1

输入

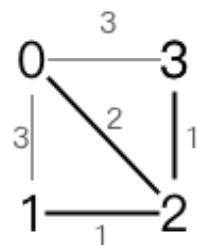
1	4 5
2	0 1 3
3	0 2 2
4	0 3 3
5	2 3 1
6	1 2 1

输出

1	4
---	---

解释

对应的无向图如下图所示。加粗的部分即为最小生成树，其边权之和为 $1+1+2=4$ 。



样例2

输入

1	3 1
2	0 1 1

输出

1	-1
---	----

解释

由于此图不连通，因此不存在最小生成树。

以下是使用 Prim 算法求解的 Python 代码：

```
1 import heapq
2
3 def prim(graph, n):
4     visited = [False] * n
5     min_heap = [(0, 0)] # (weight, vertex)
6     min_spanning_tree_cost = 0
7
8     while min_heap:
9         weight, vertex = heapq.heappop(min_heap)
10
11        if visited[vertex]:
12            continue
13
14        visited[vertex] = True
15        min_spanning_tree_cost += weight
16
17        for neighbor, neighbor_weight in graph[vertex]:
18            if not visited[neighbor]:
19                heapq.heappush(min_heap, (neighbor_weight, neighbor))
20
21    return min_spanning_tree_cost if all(visited) else -1
22
23 def main():
24     n, m = map(int, input().split())
25     graph = [[] for _ in range(n)]
26
27     for _ in range(m):
28         u, v, w = map(int, input().split())
29         graph[u].append((v, w))
30         graph[v].append((u, w))
31
32     min_spanning_tree_cost = prim(graph, n)
33     print(min_spanning_tree_cost)
34
```

```
35 if __name__ == "__main__":
36     main()
37
```

sy397: 最小生成树-Kruskal算法 简单

<https://sunnywhy.com/sfbj/10/5/397>

现有一个共n个顶点、m条边的无向图（假设顶点编号为从0到n-1），每条边有各自的边权。在图中寻找一棵树，使得这棵树包含图上所有顶点、所有边都是图上的边，且树上所有边的边权之和最小。使用Kruskal算法求出这个边权之和的最小值。

输入

第一行两个整数n、m ($1 \leq n \leq 10^4, 0 \leq m \leq 10^5$)，分别表示顶点数、边数；

接下来m行，每行三个整数u、v、w ($0 \leq u \leq n - 1, 0 \leq v \leq n - 1, u \neq v, 1 \leq w \leq 100$)，表示一条边的两个端点的编号及边权。数据保证不会有重边。

输出

输出一个整数，表示最小的边权之和。如果不存在这样的树，那么输出-1。

样例1

输入

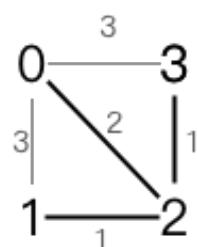
```
1 4 5
2 0 1 3
3 0 2 2
4 0 3 3
5 2 3 1
6 1 2 1
```

输出

```
1 | 4
```

解释

对应的无向图如下图所示。加粗的部分即为最小生成树，其边权之和为 $1+1+2=4$ 。



Kruskal算法是一种用于寻找最小生成树的算法。它的基本思想是按照边的权值从小到大的顺序选择边，如果这条边连接的两个顶点不在同一连通分量中，则选择这条边，否则放弃这条边。重复这个过程，直到图中所有的顶点都在同一连通分量中。

在实现Kruskal算法时，我们需要使用并查集来维护图中的连通分量，以便于快速判断两个顶点是否在同一连通分量中。

以下是Kruskal算法的Python实现：

```
1 class UnionFind:
2     def __init__(self, n):
3         self.parent = list(range(n))
4         self.rank = [0] * n
5
6     def find(self, x):
7         if self.parent[x] != x:
8             self.parent[x] = self.find(self.parent[x])
9         return self.parent[x]
10
11    def union(self, x, y):
12        px, py = self.find(x), self.find(y)
13        if self.rank[px] > self.rank[py]:
14            self.parent[py] = px
15        else:
16            self.parent[px] = py
17            if self.rank[px] == self.rank[py]:
18                self.rank[py] += 1
19
20    def kruskal(self, n, edges):
21        uf = UnionFind(n)
22        edges.sort(key=lambda x: x[2])
23        res = 0
24        for u, v, w in edges:
25            if uf.find(u) != uf.find(v):
26                uf.union(u, v)
27                res += w
28        if len(set(uf.find(i) for i in range(n))) > 1:
29            return -1
30        return res
31
32    n, m = map(int, input().split())
33    edges = []
34    for _ in range(m):
35        u, v, w = map(int, input().split())
36        edges.append((u, v, w))
37    print(kruskal(n, edges))
```

这段代码首先定义了一个并查集类 `UnionFind`，然后定义了 `kruskal` 函数来实现 Kruskal 算法。在 `kruskal` 函数中，我们首先创建一个并查集实例，然后按照边的权值对边进行排序。然后遍历每一条边，如果这条边连接的两个顶点不在同一连通分量中，就将这两个顶点合并到同一连通分量中，并将这条边的权值加到结果中。最后，如果图中的所有顶点都在同一连通分量中，就返回结果，否则返回-1。

可以看到，kruskal 算法的时间复杂度主要来源于对边进行排序，因此其时间复杂度是 $O(E \log E)$ ，其中 E 为图的边数。显然 kruskal 适合顶点数较多、边数较少的情况，这和 prim 算法恰好相反。于是可以根据题目所给的数据范围来选择合适的算法，即如果是稠密图(边多)，则用 `prim` 算法；如果是稀疏图(边少)，则用 `kruskal` 算法。

20106: 走山路

bfs + heap, Dijkstra, <http://cs101.openjudge.cn/routine/20106/>

04115: 鸣人和佐助

bfs, <http://cs101.openjudge.cn/practice/04115/>

04116: 拯救行动

bfs, <http://cs101.openjudge.cn/practice/04116/>

04129: 变换的迷宫

bfs, <http://cs101.openjudge.cn/practice/04129>

02802: 小游戏

bfs, <http://cs101.openjudge.cn/practice/02802/>

19930: 寻宝

bfs, <http://cs101.openjudge.cn/practice/19930>

三、笔试题目

2023年有考到KMP，冒泡排序的优化。

2022年5个大题：图Dijkstra，二叉树，排序，单链表，二叉树。

2021年6个大题：森林dfs、bfs，哈夫曼树，二叉树建堆，图prim，二叉树遍历，图走迷宫。

选择（30分，每题2分）

Q:

判断（10分，每题1分）

对填写"Y"，错填写"N"

Q: (Y)

填空（20分，每题2分）

Q:

简答（24分，每题6分）

Q:

参考

Problem Solving with Algorithms and Data Structures using Python, 3ed.

<https://runestone.academy/ns/books/published/pythonds3/index.html?mode=browsing>

<https://github.com/psads/pythonds3>

https://github.com/Yuqiu-Yang/problem_solving_with_algorithms_and_data_structures_using_python

<https://github.com/wesleyjtann/Problem-Solving-with-Algorithms-and-Data-Structures-Using-Python>

晴题目练习网址, <https://sunnywhy.com/sfbj>

Applications of the 20 Most Popular Graph Algorithms

<https://memgraph.com/blog/graph-algorithms-applications>