图论（2）基本图算法

一、宽度优先搜索（BFS,Breadth First Search）

（一）基本图算法：BFS框架

1.基本定义

广度优先搜索是一种图遍历算法，它在移动到下一个深度级别的顶点之前，先探索图中当前深度的所有顶点。它从指定的顶点开始，先访问其所有邻居，然后再移动到下一层。BFS常用于路径查找、连通分量以及图的最短路径问题等算法中。

BFS的算法如下：

[1] 初始化：将起始节点入队，并标记为已访问。

[2] 探索：当队列不为空时：从队列中出队一个元素并访问它；对于出队节点的每个未访问的邻居，都将邻居入队，并标记为已访问。

[3] 终止：重复步骤2，直到队列为空

该算法确保从起始节点开始，以广度优先的方式访问图中的所有节点

2.用BFS算法访问图中的所有节点

代码实现：

```
from collections import defaultdict, deque
#defaultlist为python内置包，在创建时，采用defaultdict(type)可以创建值的类型为type的字典
#defaultdict相较dict，当访问一个不存在的键时，会返回默认值，而不会发生keyError错误

# Class to represent a graph using adjacency list
class Graph:
    def __init__(self):
        self.adjList = defaultdict(list)

    # Function to add an edge to the graph
    def addEdge(self, u, v):
        self.adjList[u].append(v)

    # Function to perform Breadth First Search on a graph represented using adjacency list
    def bfs(self, startNode):
        # Create a queue for BFS
        queue = deque()
        visited = set()

        # Mark the current node as visited and enqueue it
        visited.add(startNode)
```

```
        queue.append(startNode)

        # Iterate over the queue
        while queue:
            # Dequeue a vertex from queue and print it
            currentNode = queue.popleft()
            print(currentNode, end=" ")

            # Get all adjacent vertices of the dequeued vertex currentNode
            # If an adjacent has not been visited, then mark it visited and enqueue it
            for neighbor in self.adjList[currentNode]:
                if neighbor not in visited:
                    visited.add(neighbor)
                    queue.append(neighbor)

# Create a graph
graph = Graph()

# Add edges to the graph
graph.addEdge(0, 1)
graph.addEdge(0, 2)
graph.addEdge(1, 3)
graph.addEdge(1, 4)
graph.addEdge(2, 4)

# Perform BFS traversal starting from vertex 0
print("Breadth First Traversal starting from vertex 0:", end=" ")
graph.bfs(0)
#Breadth First Traversal starting from vertex 0: 0 1 2 3 4
```

时间复杂度：每个节点和边最多被访问一次，时间复杂度为O(V+E)

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。在最坏情况下，空间复杂度是O(V)

3.BFS在图论中的应用

最短路径查询：BFS可以用于在无权图中查找两个节点之间的最短路径。通过遍历过程中记录每个节点的父节点，可以重构出最短路径。

环检测：BFS可用于检测图中的环。如果在遍历过程中一个节点被访问了两次，那么这表明图中存在环

连通分量：BFS可用于识别图中的连通分量。每个连通分量是可以相互到达的节点集合

拓扑排序：BFS可用于在有向无环图DAG上进行拓扑排序。拓扑排序将节点按照线性顺序排列，使得对于任何边（u，v），u都在v之前

二叉树的层次遍历：BFS可用于进行二叉树的层次遍历。这种遍历方式先访问同一层的所有节点，然后再移动到下一层

网络路由：BFS可用于查找网络中两个节点之间的最短路径，这对于在网络协议中路由数据包非常有用。

（二）经典例题：词梯问题

1.问题描述

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。

2.Step1：构建词梯图

第一个问题是如何用图来表示大的单词集合。如果两个单词之间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。注：无权重的无向图

假设有一个单词列表，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。然而，时间复杂度为O(n^2)，并不是一个好的选择

如下的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。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。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

在python中，采用字典实现上述方法较为便捷。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，我们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。

```
def build_graph(filename):
    buckets = {}
    the_graph = Graph()
    with open(filename, "r", encoding="utf8") as file_in:
        all_words = file_in.readlines()
    # all_words = ["bane", "bank", "bunk", "cane", "dale", "dunk", "foil", "fool", "kale",
    #              "lane", "male", "mane", "pale", "pole", "poll", "pool", "quip",
    #              "quit", "rain", "sage", "sale", "same", "tank", "vain", "wane"
    #              ]

    # create buckets of words that differ by 1 letter
    for line in all_words:
        word = line.strip()
        for i, _ in enumerate(word):
```

```
            bucket = f"{word[:i]}_{word[i + 1:]}"
            buckets.setdefault(bucket, set()).add(word)

    # connect different words in the same bucket
    for similar_words in buckets.values():
        for word1 in similar_words:
            for word2 in similar_words - {word1}:
                the_graph.add_edge(word1, word2)
    return the_graph
#dict用法：mydict.setdefault(key,default_value)，若key存在，返回key对应的value；若不存在则添加key，值
#a.add(num)，向集合a中添加num,-表示作差集
```

3.Step2：利用BFS实现词梯问题

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。

为了记录进度，BFS会将顶点标记成白色、灰色和黑色。在构建时，所有顶点都被初始化为白色，即没有被访问过。当第一次被访问时，就会被标记为灰色；当BFS完成对该顶点的访问之后，就会被标记为黑色。

以下是包括Vertex、Graph类，build_Graph函数和BFS算法主体的整套程序，以及最后的解析函数traverse

```
import sys
from collections import deque

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

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

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

    def __len__(self):
        return self.num_vertices

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

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

```
            nv = self.add_vertex(t)
        self.vertices[f].add_neighbor(self.vertices[t], cost)

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

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


class Vertex:
    def __init__(self, num):
        self.key = num
        self.connectedTo = {}
        self.color = 'white'
        self.distance = sys.maxsize
        self.previous = None
        self.disc = 0
        self.fin = 0
#sys.maxsize表示变量可以取到的最大值
    def add_neighbor(self, nbr, weight=0):
        self.connectedTo[nbr] = weight

    # def __lt__(self,o):
    #     return self.id < o.id

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

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

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

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



def build_graph(filename):
    buckets = {}
    the_graph = Graph()
    with open(filename, "r", encoding="utf8") as file_in:
        all_words = file_in.readlines()
```

```
    # all_words = ["bane", "bank", "bunk", "cane", "dale", "dunk", "foil", "fool", "kale",
    #              "lane", "male", "mane", "pale", "pole", "poll", "pool", "quip",
    #              "quit", "rain", "sage", "sale", "same", "tank", "vain", "wane"
    #              ]

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

    # connect different words in the same bucket
    for similar_words in buckets.values():
        for word1 in similar_words:
            for word2 in similar_words - {word1}:
                the_graph.add_edge(word1, word2)

    return the_graph


#g = build_graph("words_small")
g = build_graph("vocabulary.txt")
print(len(g))


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

# 以FOOL为起点，进行广度优先搜索，从FOOL到SAGE的最短路径，
# 并为每个顶点着色、赋距离和前驱。
bfs(g.get_vertex("FOOL"))


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

    return ans
```

```
ans = traverse(g.get_vertex("SAGE")) # 从SAGE开始回溯，逆向打印路径，直到FOOL
print(*ans[::-1])
```

代码解析:

Vertex类构建key、connectedTo邻居节点，处理状态变量color，前序节点previous。另有加入邻居节点函数add_neighbour

Graph类基本性质为所有点的字典集合（key+node），以及整个graph的大小。另有关键函数：加入节点add_vertex，取得节点get_vertex，构建边关系add_edge

build_graph函数建立桶并完成对于图的连接：建立起整个桶，并将每一个word插入到桶中。最后根据桶（字典：key为桶，value为word的列表）构建neighbour的关系（通过add_edge函数）

接下来做bfs，对于initial word建立起distance与路径的关系。具体地，通过bfs算法确认以initial word作为original point的相对位置关系，通过previous进行链接。

最后进行traverse回溯，从目标word出发，根据previous的链接进行节点回溯

注：BFS算法主体是两个循环的嵌套。while循环对于图中每个节点最多执行一次，时间复杂度O（V）。嵌套在循环中的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，故为O（E）。因此总的时间复杂度为O(V+E)

另外，回溯过程最坏的时间复杂度为O(V)，创建关系图的时间复杂度为O(V+E)

二、深度优先搜索（DFS,Depth First Search）

（一）基本图算法：DFS框架

1.定义

图的深度优先遍历，与树的深度优先遍历类似。与树的不同在于，图中可能包含环。为了避免多次处理同一个节点，可以使用布尔型的已访问数组。一个图可以有多种DFS遍历方式

在DFS中，我们从一个顶点开始，尽可能深地搜索图的分支，直到该分支的尽头。然后，我们回溯前一个顶点，继续搜索其他分支。与BFS不同，DFS使用的是栈而不是队列来存储待访问的节点。

DFS的基本步骤如下：

[1] 初始化：标记起始节点为已访问，并将其压入栈中

[2] 遍历：当栈不为空时，从栈顶弹出一个节点，并访问它；遍历该结点的所有未访问邻居，并将它们标记为已访问，然后压入栈中

[3] 回溯与重复：当当前节点的所有邻居都被访问后，如果没有其他节点在栈中，则遍历结束；否则，继续从栈中弹出节点并重复步骤2

2.DFS遍历序列

从一个key出发，进行遍历。将该key的所有邻居依次弹入栈中（注意反正弹入，确保stack的遍历是依照neighbor list顺序进行解析）

```
from collections import defaultdict


class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def addEdge(self, u, v):
        self.graph[u].append(v)

    def DFS(self, v):
        visited = set()
        stack = [v]

        while stack:
            current = stack.pop()
            if current not in visited:
                print(current, end=' ')
                visited.add(current)
                stack.extend(reversed(self.graph[current]))


# Driver's code
if __name__ == "__main__":
    g = Graph()
    g.addEdge(0, 1)
    g.addEdge(0, 2)
    g.addEdge(1, 2)
    g.addEdge(2, 0)
    g.addEdge(2, 3)
    g.addEdge(3, 3)

    print("Following is Depth First Traversal (starting from vertex 2)")

    # Function call
    g.DFS(2)
```

（二）经典例题：骑士周游问题

1.问题描述

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”。

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dfs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rows × columns-1的路径，满足图中的每一个顶点都只被访问一次。

2.step1：骑士周游图的构建

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。

以下代码构建了n*n棋盘对应的完整图。knight_graph函数遍历整个棋盘，当它访问棋盘上的每一格时，都会调用辅助函数gen_legal_moves创建一个列表，用于记录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线性顶点数

代码实现：

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

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

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

3.step2：实现骑士周游

采用dfs算法解决骑士周游问题。与bfs不同，dfs通过尽可能深地搜索分支来构建搜索树。当dfs遇到死路时（无法找到下一个合理走法），它会退回到树中倒数第二深的顶点，以继续移动

knight_tour函数接受4个参数：n是搜索树的当前深度，path是到当前为止访问过的顶点列表；u是希望在图中访问的顶点；limit是路径上想要实现的顶点总数。该函数为递归函数：被调用时，首先检查基本情况是否返回True；如果路径不够长，则通过选择一个新的访问结点并对其递归调用knight_tour函数来进行更深一层的搜索

dfs也使用颜色来记录已经访问的顶点。未访问白色，已访问灰色。如果一个顶点的所有相邻顶点都已被访问过，但路径长度仍然没有到64，则说明遇到了死路，从而dfs需要回溯（返回False）

注：在bfs中，我们使用队列记录将要访问的顶点。由于dfs是递归的，我们实际上是隐式地使用一个栈来完成回溯。

代码实现：

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

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

备注：一个生成neighbors更好的改进是，采用neightbors = orderes_by_avail(u)函数，保证接下来要访问的顶点有最少的合理走法，原因在于：

选择合理走法最多的顶点，会使得骑士在周游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，而无法到达另一边那些没访问过的格子，保证骑士能够尽早访问难以到达的角落。在这种情况下可以帮助我们更快地找到可行路径。

这种启发式技术（加速算法的知识）被称作Warnsdorff算法，以纪念在1823年提出该算法的数学家H.C. Warnsdorff

4.优化：Warnsdorff算法

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K为一个较小的常量，N为棋盘上的格子数。

我们可以通过树来理解搜索过程。从起点开始，算法生成并且检测骑士能走的每一步，每一步都代表一个子树。若骑士位于四角，则只有两种合理走法；但若在棋盘中央，则有8种合理走法，即有8株子树。

我们使用平均分支因子K估计节点数，则这个算法总执行步长即为k^(N+1)-1的量级（类比二叉树）。尽管我们只需要找到一个可行解，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，并不能改变该问题的指数特性

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，从而尽可能地减少搜索空间

代码实现：

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

注：以下是骑士周游问题的总代码--

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

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

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

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


  class Vertex:
      def __init__(self, num):
          self.key = num
          self.connectedTo = {}
```

```
        self.color = 'white'

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

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

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

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

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

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

```
            return True

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

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

        bdSize = int(input())  # 棋盘大小
        *start_pos, = map(int, input().split())  # 起始位置
        g = knight_graph(bdSize)
        start_vertex = g.get_vertex(pos_to_node_id(start_pos[0], start_pos[1], bdSize))
        if start_vertex is None:
            print("fail")
            exit(0)
            #表示中断程序，程序正常退出
        tour_path = []
        done = knight_tour(0, tour_path, start_vertex, bdSize * bdSize-1)
        if done:
            print("success")
        else:
            print("fail")

        exit(0)

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

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

（三）通用深度优先搜索

1.算法基本介绍

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，目标是尽可能深地搜索，尽可能多地连接图中的顶点，并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）。此外深度优先搜索还会使用Vertex类的两个额外实例变量：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。

我们采用DFSGraph作为Graph的一个子类来实现深度优先搜索算法。该实现继承Graph类，并且增加了time实例变量来记录算法的执行步数。子类中包含dfs和dfs_visit两个方法

在dfs方法中，其遍历图中的所有顶点，并对白色顶点调用dfsvisit方法。为了使得self成为可迭代对象，我们也需要在graph中设置iter函数，返回vertices.values()。

2.算法实现

```
#注：super()就是用来调用父类的方法，super().__init__()就是调用父类的init方法
#子类语法格式：class 子类名(父类名1，父类名2...)
#一个类继承另一个类时，它将自动获得另一个类的所有属性和方法
class DFSGraph(Graph):
    def __init__(self):
        super().__init__() #init括号可以传入父类init方法中的实际参数，当然也需要在def init中写出
        self.time = 0                    #不是物理世界，而是算法执行步数

    def dfs(self):
        for vertex in self:
            vertex.color = "white"       #颜色初始化
            vertex.previous = -1
        for vertex in self:              #从每个顶点开始遍历
            if vertex.color == "white":
                self.dfs_visit(vertex)   #还有未包括的顶点
                                         #则建立森林（另一棵树）

    def dfs_visit(self, start_vertex):
        start_vertex.color = "gray"
        self.time = self.time + 1        #记录算法的步骤
        start_vertex.discovery_time = self.time
        for next_vertex in start_vertex.get_neighbors():
            if next_vertex.color == "white":
                next_vertex.previous = start_vertex
                self.dfs_visit(next_vertex)      #深度优先递归访问
        start_vertex.color = "black"
        self.time = self.time + 1
        start_vertex.closing_time = self.time
```

3.一些注意事项与dfs的性质

（1）与bfs的比较

和dfs_visit相比两者几乎一样。除了for循环的最后一行，dfs通过递归调用自己先进行下一层的搜索，bfs则将顶点添加到队列中，以供后续搜索

（2）括号特性

深度优先搜索树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

（3）深度优先搜索的时间复杂度

在dfs函数中有两个循环，每个都是V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调用，因此循环最多会针对图中的每一条边执行一次。

因此，dfs时间复杂度为O(V+E)

（4）深度优先搜索树（林）的性质

1. **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 有 u.fin < v.fin。
2. **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 t1 和 t2，当且仅当 t1 中的任意节点的结束时间都早于 t2 中的任意节点的开始时间时，称 t1 小于 t2，即符号化表示为：t1 < t2 说明 $\forall$u $\in$ t1, v $\in$ t2 ， 有u < v。
3. **节点间的互斥关系：** 如果 t1 小于 t2，且节点 u 属于 t1，节点 v 属于 t2，则u到v的路径不存在。
4. **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 t1，顶点 v 属于 t 的子树 t2。这种情况下，t1 的结束时间早于 t2 的开始时间。

三、编程题目

（一）sy321：迷宫最短路径

1.问题描述

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，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。

2.代码实现

```
from collections import deque

MAX_DIRECTIONS = 4
dx = [0, 0, 1, -1]
dy = [1, -1, 0, 0]

def is_valid_move(x, y):
```

```
    return 0 <= x < n and 0 <= y < m and maze[x][y] == 0 and not in_queue[x][y]

def bfs(start_x, start_y):
    queue = deque()
    queue.append((start_x, start_y))
    in_queue[start_x][start_y] = True
    while queue:
        x, y = queue.popleft()
        if x == n - 1 and y == m - 1:
            return
        for i in range(MAX_DIRECTIONS):
            next_x = x + dx[i]
            next_y = y + dy[i]
            if is_valid_move(next_x, next_y):
                prev[next_x][next_y] = (x, y)
                in_queue[next_x][next_y] = True
                queue.append((next_x, next_y))

def print_path(pos):
    prev_position = prev[pos[0]][pos[1]]
    if prev_position == (-1, -1):
        print(pos[0] + 1, pos[1] + 1)
        return
    print_path(prev_position)
    print(pos[0] + 1, pos[1] + 1)

n, m = map(int, input().split())
maze = [list(map(int, input().split())) for _ in range(n)]

in_queue = [[False] * m for _ in range(n)]
prev = [[(-1, -1)] * m for _ in range(n)]

bfs(0, 0)
print_path((n - 1, m - 1))
```

（二）sy380：无向图的连通块

1.问题描述

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，分别表示顶点数和边数；

接下来m行，每行两个整数u、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。

2.代码实现

首先创建一个邻接列表来表示图，然后对于每个未访问节点（用visited判断）进行dfs或者bfs。每次搜索都会找到一个连通块。visited帮助标记已经访问的节点，以防止重复访问

以下给出使用dfs的代码：

```
def dfs(node, visited, adjacency_list):
    visited[node] = True
    for neighbor in adjacency_list[node]:
        if not visited[neighbor]:
            dfs(neighbor, visited, adjacency_list)

n, m = map(int, input().split())
adjacency_list = [[] for _ in range(n)]
for _ in range(m):
    u, v = map(int, input().split())
    adjacency_list[u].append(v)
    adjacency_list[v].append(u)

visited = [False] * n
connected_components = 0
for i in range(n):
    if not visited[i]:
        dfs(i, visited, adjacency_list)
        connected_components += 1

print(connected_components)
```

（三）sy381：无向连通图

1.问题描述

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` ），判断其是否是连通图。

输入

第一行两个整数n、m，分别表示顶点数和边数；

接下来m行，每行两个整数u、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` 。

2.代码实现

与题思路保持一致，只遍历一次然后进行判断

```
def dfs(node, visited, adjacency_list):
    visited[node] = True
    for neighbor in adjacency_list[node]:
        if not visited[neighbor]:
```

```
            dfs(neighbor, visited, adjacency_list)

n, m = map(int, input().split())
adjacency_list = [[] for _ in range(n)]
for _ in range(m):
    u, v = map(int, input().split())
    adjacency_list[u].append(v)
    adjacency_list[v].append(u)

visited = [False] * n
dfs(0, visited, adjacency_list)

if all(visited):
    print("Yes")
else:
    print("No")
```

注：python内部的all()函数用于判断给定的可迭代参数iterable中的所有元素是否都为TRUE，如果是返回True，否则返回False

（四）sy382：有向图判环

1.问题描述

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，分别表示顶点数和边数；

接下来m行，每行两个整数u、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` 。

2.代码实现

在判断环的过程中，我们采用dfs方法。我们从一个节点开始，访问其每一个邻居，如果在当前访问过程中，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/状态变量来跟踪每个节点的状态：未访问0，正在访问1，已访问2

```
def has_cycle(n, edges):
    graph = [[] for _ in range(n)]
    for u, v in edges:
        graph[u].append(v)

    color = [0] * n

    def dfs(node):
```

```
        if color[node] == 1:
            return True
        if color[node] == 2:
            return False

        color[node] = 1
        for neighbor in graph[node]:
            if dfs(neighbor):
                return True
        color[node] = 2
        return False

    for i in range(n):
        if dfs(i):
            return "Yes"
    return "No"

# 接收数据
n, m = map(int, input().split())
edges = []
for _ in range(m):
    u, v = map(int, input().split())
    edges.append((u, v))

# 调用函数
print(has_cycle(n, edges))
```

（五）sy383：最大权值连通块

1.问题描述

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，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，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。

2.代码实现

我们需要找到给定无向图中所有连通块的最大权值。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。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，可以计算权值之和，并更新最大权值。

```
def max_weight(n, m, weights, edges):
    graph = [[] for _ in range(n)]
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)

    visited = [False] * n
    max_weight = 0

    def dfs(node):
        visited[node] = True
        total_weight = weights[node]
        for neighbor in graph[node]:
            if not visited[neighbor]:
                total_weight += dfs(neighbor)
        return total_weight

    for i in range(n):
        if not visited[i]:
            max_weight = max(max_weight, dfs(i))

    return max_weight

# 接收数据
n, m = map(int, input().split())
weights = list(map(int, input().split()))
edges = []
for _ in range(m):
    u, v = map(int, input().split())
    edges.append((u, v))

# 调用函数
print(max_weight(n, m, weights, edges))
```

（六）sy384：无向图的顶点层号

1.问题描述

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，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，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。

2.代码实现

由于我们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，因此需要使用bfs搜索。为了输出顶点层号，我们可以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

代码实现：

```
from collections import deque

def bfs(n,s,edges):
    graph = [[] for _ in range(n)]
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)

    distance = [-1] * n
    distance[s] = 0

    queue = deque([s])
    while queue:
        node = queue.popleft()
        for neighbor in graph[node]:
            if distance[neighbor] == -1:
                distance[neighbor] = distance[node] + 1
                queue.append(neighbor)

    return distance

# 接收数据
n, m, s = map(int, input().split())
edges = []
for _ in range(m):
    u, v = map(int, input().split())
    edges.append((u, v))

# 调用函数
distances = bfs(n, s, edges)
print(' '.join(map(str, distances)))
```

（七）sy385：受限层号的顶点数

1.问题描述

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，分别表示顶点数、边数、起始顶点编号和受限层号；

接下来m行，每行两个整数u、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k的顶点个数。

2.代码实现

逻辑上，与无向图的构建方式相同

代码实现：

```
from collections import deque

def bfs(n, s, k, edges):
    graph = [[] for _ in range(n)]
    for u, v in edges:
        graph[u].append(v)  # 只按照输入的方向添加边

    distance = [-1] * n
    distance[s] = 0

    queue = deque([s])
    while queue:
        node = queue.popleft()
        for neighbor in graph[node]:
            if distance[neighbor] == -1:
                distance[neighbor] = distance[node] + 1
                queue.append(neighbor)

    return sum(1 for d in distance if d <= k and d != -1) #注意d不能为-1

# 接收数据
n, m, s, k = map(int, input().split())
edges = []
for _ in range(m):
    u, v = map(int, input().split())
    edges.append((u, v))

# 调用函数
count = bfs(n, s, k, edges)
print(count)
```

总结：DFS与BFS

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。