grafos.py

```python
import heapq, math
from collections import deque
class Graph:# Classe base para representar um grafo
    def __init__(self, vertices, edges, direct=False):
        self.adj = [[] for _ in range(vertices)]  # Lista de adjacências
        for edge in edges:
            v1, v2 = edge[:2]# Peso padrão 1 se não especificado
            w = edge[2] if len(edge) == 3 else 1
            self.adj[v1].append((v2, w))
            if not direct:# Aresta reversa para grafo não-direcionado
                self.adj[v2].append((v1, w))
class dfs(Graph):# Classe para busca em profundidade (DFS)
    def path_rec(self, visited, e, f, size):
        visited[e] = True
        if e == f: return size  # Caminho encontrado
        for i, w in self.adj[e]:
            if not visited[i]:
                r = self.path_rec(visited, i, f, size+w)
                if r is not None: return r
    def path(self, e, f):
        visited = [False] * len(self.adj)
        return self.path_rec(visited, e, f, 0)
class bfs(Graph):# Classe para busca em largura (BFS)
    def path_rec(self, visited, e, f, caminho):
        visited[e] = True
        for i, w in self.adj[e]:
            if i == f: return caminho + [i]  # Caminho encontrado
            if not visited[i]:# Adiciona estado à pilha
                self.pilha.append((visited + [], i, caminho + [i]))
    def path(self, e, f):
        self.pilha = [([False] * len(self.adj), e, [e])]
        while self.pilha:
            visited, i, caminho = self.pilha.pop()
            r = self.path_rec(visited, i, f, caminho)
            if r is not None: return r
class Dijkstra(Graph): # Classe para o algoritmo de Dijkstra
    def path(self, e, f):
        V = len(self.adj)
        dist = [float('inf')] * V
        dist[e] = 0
        pai = [None] * V
        pilha = [(0, e)]  # Heap de prioridades
        while pilha:
            size, i = heapq.heappop(pilha)
            if i == f:  # Caminho encontrado
                path = []
                while i is not None:
                    path.append(i)
                    i = pai[i]
                return path[::-1], dist[f]
            for j, w in self.adj[i]:
                if size + w < dist[j]:
                    dist[j], pai[j] = size + w, i
                    heapq.heappush(pilha, (dist[j], j))
        return None, float('inf')
class Tarjan(Graph):# Classe para encontrar CFC (conexos forte)
    def __init__(self, vertices, edges, direct=True):
```

```
        super().__init__(vertices, edges, direct)
        self.index = 0
        self.stack = []
        self.on_stack = [False] * vertices
        self.indices = [-1] * vertices
        self.low_link = [-1] * vertices
        self.sccs = []
    def strongconnect(self, v): # Inicializa índice e low-link
        self.indices[v] = self.low_link[v] = self.index
        self.index += 1
        self.stack.append(v)
        self.on_stack[v] = True
        for w, _ in self.adj[v]:
            if self.indices[w] == -1:
                self.strongconnect(w)
                self.low_link[v] = min(self.low_link[v], self.low_link[w])
            elif self.on_stack[w]:
                self.low_link[v] = min(self.low_link[v], self.indices[w])
        if self.low_link[v] == self.indices[v]:  # Vértice de raiz de SCC
            scc = []
            while True:
                w = self.stack.pop()
                self.on_stack[w] = False
                scc.append(w)
                if w == v: break
            self.sccs.append(scc)
    def find_sccs(self):
        for v in range(len(self.adj)):
            if self.indices[v] == -1:
                self.strongconnect(v)
        return self.sccs
    def scc_to_graph(self): # Primeiro, identificamos todos os SCCs
        sccs = self.find_sccs()
        scc_map = {v: idx for idx, scc in enumerate(sccs) for v in scc}
        new_edges = [(scc_map[v], scc_map[w], weight)
            for v in range(len(self.adj)) for w, weight in self.adj[v]
            if scc_map[v] != scc_map[w]]
        return Graph(len(sccs), new_edges, direct=True)
class Dinic():# Classe para o algoritmo de fluxo máximo de Dinic
    def __init__(self, vertices, edges):
        self.vertices = vertices
        self.adj = [[] for _ in range(vertices)]
        for v1, v2, *w in edges:
            w = w[0] if w else 1
            self.adj[v1].append([v2, w, len(self.adj[v2])])
            self.adj[v2].append([v1, 0, len(self.adj[v1]) - 1])
        self.level = [-1] * vertices
    def bfs_level_graph(self, source, sink):
        """Realiza BFS para construir o grafo de nível."""
        self.level = [-1] * len(self.adj)
        self.level[source] = 0
        queue = deque([source])
        while queue:
            u = queue.popleft()
            for v, capacity, _ in self.adj[u]:
                if self.level[v] < 0 and capacity > 0:
                    self.level[v] = self.level[u] + 1
                    queue.append(v)
        return self.level[sink] != -1
    def dfs_blocking_flow(self, u, flow, sink, start):
```

```
        if u == sink: return flow
        while start[u] < len(self.adj[u]):
            v, capacity, rev_index = self.adj[u][start[u]]
            if self.level[v] == self.level[u] + 1 and capacity > 0:
                pushed = self.dfs_blocking_flow(v, min(flow, capacity),
                                                sink, start)
                if pushed > 0:
                    self.adj[u][start[u]][1] -= pushed
                    self.adj[v][rev_index][1] += pushed
                    return pushed
            start[u] += 1
        return 0
    def max_flow(self, source, sink):
        max_flow = 0
        while self.bfs_level_graph(source, sink):
            start = [0] * self.vertices
            while (flow := self.dfs_blocking_flow(source, float('inf'),
                                                  sink, start)):
                max_flow += flow
        return max_flow
class Dijkstrapar(Graph):# Variante do Dijkstra para grafos modificados
    def path(self, e, f):
        V = len(self.adj)
        self.nadj = [[] for _ in range(V)]
        for u in range(V):  # Construir o grafo G2
            for v1, w1 in self.adj[u]:
                for v2, w2 in self.adj[v1]:
                    if u != v2:
                        self.nadj[u].append((v2, w1 + w2))
        self.adj = self.nadj
        dist = [float('inf')] * V
        dist[e] = 0
        pai = [None] * V
        pilha = [(0, e)]
        while pilha:
            size, i = heapq.heappop(pilha)
            if i == f:  # Caminho encontrado
                path = []
                while i is not None:
                    path.append(i)
                    i = pai[i]
                return path[::-1], dist[f]
            for j, w in self.adj[i]:
                if size + w < dist[j]:
                    dist[j], pai[j] = size + w, i
                    heapq.heappush(pilha, (dist[j], j))
        return None, float('inf')
class dfsimposto(Graph):# DFS customizado com cálculo de custo de imposto
    def path_rec(self, visited, e, size):
        visited[e] = True
        custo = self.imposto[e]
        for i, w in self.adj[e]:
            if not visited[i]:
                custo += self.path_rec(visited, i, w)
        return math.ceil(size * custo / 4)
    def path(self, e, f):
        visited = [False] * len(self.adj)
        self.imposto = []
        return self.path_rec(visited, e, 0)
```