# 1 Grafs

**Definició 1.1** (Graf). *Un graf és un conjunt $V$ de vèrtexs o nodes i un conjunt $E$ de d'arestes o arcs relacionats entre aquests vèrtexs. El graf és simple si el graf és no dirigit, no té cap bucle i no més d'una aresta entre dos vèrtexs diferents qualssevol.*

**Definició 1.2** (Grau d'un vèrtex). *El grau d'un vèrtex és el nombre d'arestes que hi incideixen. El grau d'entrada (indegree) és el nombre d'arestes d'entrada i el grau de sortida (outdegree) és el nombre d'arestes de sortida.*

**Propietat 1.3** (Representació d'un graf).

| Representació | Espai | Afegir aresta | Aresta entre dos nodes | Iterar sobre els veïns | Best |
|---|---|---|---|---|---|
| `array` | $E$ | 1 | $E$ | $E$ | |
| Matriu d'adjacència | $V$ | 1 | 1 | $V$ | dens |
| Llista d'adjacència | $E+V$ | 1 | degree($v$) | degree($v$) | *sparse* |

**Observació 1.4.** *Un camí és un recorregut sense vèrtexs repetits. Un cicle és un camí amb primer i últim iguals.*

**Teorema 1.5.** $\mathcal{O}(|V|+|E|) \implies \mathcal{O}(2|E|) \implies \mathcal{O}(|E|)$.

```
visited = set()
def dfs(visited, graph, node):
    if node not in visited:
        print(node)
        visited.add(node)
        for n in graph[node]:
            dfs(visited, graph, n)
#    ARRAY      MATR      LIST
#   O(|E|^2)  O(|V|^2)   O(|E|)
# DFS: pila
# BFS: cua, arbres de camí mínim
```

La diferència entre *Dijkstra* i `dfs` és que el primer usa una cua amb prioritats en lloc d'una regular; prioritza nodes en funció dels costos de les arestes.

```
def bfs(graph, start):
    explored = []
    queue = [start]
    while queue:
        node = queue.pop(0)
        if node not in explored:
            explored.append(node)
            ns = graph[node]
            for n in ns:
                queue.append(n)
    return explored
```

El `bfs` només té en compte els nodes que estan connectats a un node $s$, la resta són ignorats.

**Definició 1.6** (Component connexa). *És un subgraf connex maximal.*

**Definició 1.7** (Graf connex). *Existeix un camí des de qualsevol vèrtex a qualsevol altre en el graf. Cada vèrtex i cada aresta pertany a una única component connexa.*

**Definició 1.8** (Graf fortament connex). *Donats dos vèrtexs qualssevol $u, v$, conté un camí dirigit de $u$ a $v$ i un camí dirigit de $v$ a $u$.*

**Definició 1.9** (Graf feblement connex). *Connex com a graf no dirigit.*

**Definició 1.10** (Graf bipartit). *Diem que un graf és bipartit si els nodes d'un grup tenen arestes cap el segon grup i viceversa.*

## Dijkstra

```
def dijkstra(G, origin, destination):
    length = len(G.nodes); minheap = [];
    dist = {}; prev = {}; counter = 0;
    h.heapify(minheap);

    for node in G.nodes:
        if(node != origin):
            dist[node] = float('inf');
            h.heappush(minheap,(float('inf'),node));
        else:
            dist[node] = 0; h.heappush(minheap,(0,node));
        prev[node] = -1;

    while(bool(minheap)):
        actual = h.heappop(minheap)[1];
        if actual == destination: break; counter += 1;
        for dest in G.neighbors(actual):
            val = 1; # si G[actual][dest]['weight'] = ø
            if(dist[dest] > dist[actual]+val):
                dist[dest] = dist[actual]+val;
                prev[dest] = actual;
                h.heappush(minheap,(dist[dest],dest));
    path = [destination];
    while prev[path[-1]] != -1: path.append(prev[path[-1]]);
    path.reverse();
    return { 'path': path, 'exp': counter, 'd': len(path) }
```

| | minpop | decrkey | TOTAL |
|---|---|---|---|
| Array | $\mathcal{O}(V)$ | $\mathcal{O}(1)$ | $\mathcal{O}(\|V\|^2)$ |
| Heap | $\mathcal{O}(\log \|V\|)$ | $\mathcal{O}(\log \|V\|)$ | $\mathcal{O}(\|V\| + \|E\| \log \|V\|)$ |

Generalitzant, tenim que la complexitat de Dijkstra per a qualsevol estructura de dades és

$$\mathcal{O}(|E| \cdot T_{decrkey} + |V| \cdot T_{minpop}). \tag{1.1}$$

Amb Dijkstra sempre arribem de $s$ a $t$ amb camí mínim independentment de l'ordre dels pesos de les arestes si aquests són positius, no si són negatius.

```
def bellman_ford(G, origin, destination):
    n = len(G.nodes); dist = {}; prev = {};
    for node in G.nodes:
        dist[node] = float('inf') if node != origin else 0;
        prev[node] = -1;
    for _ in range(n-1):
        copycat = dist;
        for edge in G.edges:
            w = 1;
            if dist[edge[1]] > dist[edge[0]]+w:
                dist[edge[1]] = dist[edge[0]]+w;
                prev[edge[1]] = edge[0];
        if dist == copycat: break;
    path = [destination];
    while prev[path[-1]] != -1:
        path.append(prev[path[-1]]);
        path.reverse();
    return path;
```

**Proposició 1.11.** *En un graf general ponderat es pot calcular el camí més curt entre dos nodes en temps $\mathcal{O}(|V||E|)$ utilitzant l'algorisme de Bellman-Ford.*

**Observació 1.12.** *En un graf ponderat sense pesos negatius, podem fer-ho millor i calcular el camí més curt entre dos nodes en temps $\mathcal{O}(|E| + |V| \log |V|)$ utilitzant l'algorisme de Dijkstra.*

Per resoldre el camí mínim en grafs acíclics negatius es linealitza usant `dfs` en temps lineal. Si posem els negatius dels pesos podem trobar els camins de longitud màxima.

**Definició 1.13** (Ordenació topològica)**.** *Ordenació dels nodes en un graf dirigit on per a cada aresta dirigida del node $A$ al node $B$ apareix el node $A$ abans del node $B$ a la seva ordenació.*

**Definició 1.14.** *Només determinats tipus de grafs tenen un ordre topològic. Aquests grafs son anomenats Grafs Dirigits Acíclics (DAGs).*

**Definició 1.15** (Algorisme de Kahn)**.** *És un algorisme d'ordenació topològica simple. Té una complexitat de $\mathcal{O}(V + E)$.*

**Proposició 1.16.** *Trobar els camins mínims entre totes les parelles de nodes d'un mateix graf utilitzant Dijkstra té una complexitat de $\mathcal{O}(|V|^2 + |V||E| \log |V|)$.*

```
# let dist be a |V|x|V| array of minimum distances
# let next be a |V|x|V| array of vertex indexes to null
def floyd_warshall():
    for each edge (u,v):
        dist[u][v] <- w(u,v)
        next[u][v] <- v
    for each vertex v:
        dist[v][v] <- 0
        next[v][v] <- v
    for k in range(|V|):
        for i in range(|V|):
            for j in range(|V|):
                if dist[i][j] > dist[i][k]+dist[k][j]:
                    dist[i][j] <- dist[i][k]+dist[k][j]
                    next[i][j] <- next[i][k]
```

La complexitat d'aquest algorisme és de $\mathcal{O}(n^3)$, on $n$ és el nombre de vèrtexs $|V|$ en el graf.

$$D_k[i,j] = \min\{D_{k-1}[i,j], D_{k-1}[i,k], D_{k-1}[k,j]\}. \quad (1.2)$$

**Definició 1.17** (Floyd-Warshall)**.** *L'algorisme de Floyd-Warshall proporciona les longituds dels camins mínims entre tots els parells de vèrtexs.*

**Definició 1.18** (Max-Flow)**.** *Un graf dirigit on tenim un node inicial ($s$) i un node objectiu ($t$) amb l'objectiu de passar el màxim flux possible. $G = (V, E, s, t, u)$.*

1. *$(V, E)$ és un graf dirigit sense arcs.*
2. *$u(e)$ és la capacitat de l'aresta $e$.*
3. *Cada aresta té una capacitat màxima finita ($\geq 0$).*
4. *Conservació de flux: $\sum f_{input} = \sum f_{output}$.*

**Definició 1.19** (Tall)**.** *Un tall és una partició dels nodes $(S, T)$ tals que $s \in S$ i $t \in T$.*

**Definició 1.20** (Xarxa residual)**.** *Donada una xarxa flux $G = (V, E)$ amb inici $s$ i destinació $t$. Sigui $f$ el flux en $G$, i consideri un parell de vèrtexs $u, v \in V$, la quantitat de flux addicional que es pot abocar sobre $u, v$ és la capacitat residual.*

**Teorema 1.21** (Teorema de flux màxim tall mínim)**.** *En una xarxa de flux, la quantitat màxima de flux que pot passar d'una font a un pou és igual a la capacitat mínima que necessitem treure-li a la xarxa perquè no pugui passar més flux de la font al pou.*

**Definició 1.22** (Camí i circuit eulerià)**.** *Un camí eulerià és aquell camí que recorre totes les arestes d'un graf passant una i només una vegada per a cadascuna d'elles. Un circuit eulerià és un camí eulerià que comença i acaba al mateix node.*

| Graf | Camí eulerià | Circuit eulerià |
|---|---|---|
| No dirigit | $grau(v) \equiv 0 \pmod 2$ o $u, v$ amb grau senar | Tots els vèrtexs del graf tenen un grau parell |
| Dirigit | $v$ amb outdegree$(v)$ − indegree$(v) = 1$<br>$u$ amb indegree$(u)$ − outdegree$(u) = 1$<br>indegree$(\omega)$ = outdegree$(\omega)$, $\forall \omega \setminus \{u, v\}$ | indegree$(v)$ = outdegree$(v)$, $\forall v$. |

```
# n = vertices; m = edges; g = adjacency list
# in = out = [0]*n; path = empty integer linked list
function findEulerianPath():
    countInOutDegrees()
    if not graphHasEulerianPath(): return null
    dfs(findStartNode())
    if path.size() == m+1: return path
    else: return null
function countInOutDegrees():
    for edges in g:
        for edge in edges:
        out[edge.from]++
        in[edge.to]++
function graphHasEulerianPath():
    start_nodes, end_nodes = 0,0
    for i in range(n):
        if(out[i]-in[i])>1 or (in[i]-out[i])>1:
            return false
        else if out[i]-in[i] == 1: start_nodes++
        else if in[i]-out[i] == 1: end_nodes++
    return (end_nodes == 0 and start_nodes == 0) or
            (end_nodes == 1 and start_nodes == 1)
```

```
function findStartNode():
    start = 0
    for i in range(n):
        if out[i]-in[i] == 1: return i
        if out[i]>0: start = i
    return start
function dfs(at):
    while(out[at]!=0):
        next_edge = g[at].get(--out[at])
        dfs(next_edge.to)
    path.insertFirst(at)
```

Com trobar el camí eulerià:

1. Determinar si existeix un camí eulerià.
2. Si hi ha dos vèrtexs senars, comencem per un dels dos; si no hi ha cap, per qualsevol.
3. Executem el `dfs` modificat que visita totes les arestes.

No oblidem Ford-Fulkerson (*max-flow*, $\mathcal{O}(Ef)$):

1. $f(u,v) \longleftarrow 0,\ \forall (u,v)$.
2. `while` $\exists$ camí $p$ entre $s$ i $p$ tal que $c_f(u,v) > 0$:
    1. $c_f(p) = \min\{c_f(u,v) \mid (u,v) \in p\}$.
    2. `foreach` $(u,v) \in p$:
        1. $f(u,v) \longleftarrow f(u,v) + c_f(p)$,
        2. $f(v,u) \longleftarrow f(v,u) - c_f(p)$.

## 2 Greedy

**Definició 2.1** (Algorisme voraç)**.** *És un algorisme que, per resoldre un problema d'optimització, fa una seqüència d'eleccions, prenent en cada pas un òptim local, amb l'esperança d'arribar a un òptim global. L'algorisme greedy no torna mai enrere per revaluar les eleccions ja preses. Per tant, aquells problemes que millor s'adapten al paradigma greedy són aquell que escollir l'òptim localment condueixen a una solució global.*

**Observació 2.2.** *Els algorismes voraços no garanteixen una solució, ni que la solució obtinguda sigui la més òptima.*

**Definició 2.3** (Minimum Spanning Tree (MST))**.** *És el resultat de fer un càlcul a partir d'un graf inicial que ens proporciona un arbre amb tots els nodes i que que minimitza la suma dels pesos de totes les arestes sel·leccionades.*

```
def kruskal(g):
    for v in g.v:
        MAKE-SET(v)
    for (u,v) in g.e ordered by increasing w(u,v):
        if FIND-SET(u) != FIND-SET(v):
            A = A U {(u,v)}
            UNION(FIND-SET(u),FIND-SET(v))
    return A
def MAKESET(): A(x) = x; rank(x) = 0
def FIND-SET(x):
    while x != A(x): x = A(x); return x
def union(x,y):
    rx = FIND-SET(x); ry = FIND-SET(y)
    if rx == ry: return
    if rank(rx)>rank(ry): A(ry) = rx
    else: A(rx) = ry
```

Kruskal comença ordenant les arestes per ordre creixent de cost. Itera sobre elles i mirem si els nodes formen part del mateix conjunt. Si no, introduïm l'aresta a la solució i unifiquem els conjunts. L'algorisme finalitza quan hem processat totes les arestes del nostre graf o bé tots els nodes han estat unificats en un mateix conjunt. Podem implementar `UNION-FIND` a través d'un `array`.

1. `MAKE-SET`: nou conjunt amb un node. $\mathcal{O}(1)$.
2. `FIND-SET`: buscar el conjunt que conté l'element $u$. Per trobar a quin component pertany un element concret hem de trobar l'arrel d'aquest component seguint els nodes pares fins que s'arriba a un bucle (un node on el seu parent és el mateix node).
3. `UNION`: crea un nou conjunt amb la unió de conjunts $S_1, S_2$. Per unificar dos elements buscarem quin és el l'arrel de cada component i si l'arrel es diferent farem que el node arrel d'un del dos nodes es el pare de l'altre.

$\mathcal{O}(V) + \mathcal{O}(E \log E) + \mathcal{O}(E \log V) \approx \mathcal{O}(E \log E)$.

```
def prim(G, origin):
    length = len(G.nodes); minheap = []; cost = {}; prev = {}; counter = 0;
    h.heapify(minheap);
    for node in G.nodes:
        if(node != origin): cost[node] = float('inf'); h.heappush(minheap,(float('inf'),node));
        else: cost[node] = 0; h.heappush(minheap,(0,node));
        prev[node] = NULL;
    while(bool(minheap)):
        actual = h.heappop(minheap)[1]; counter += 1;
        for dest in G.neighbors(actual): val = 1; # si G[actual][dest]['weight'] = ø
            if(cost[dest] > val): cost[dest] = val; prev[dest] = actual; h.heappush(minheap,(cost[dest],dest));
```

**Observació 2.4.** *L'algorisme de Prim és anàleg al de Dijkstra, amb l'única variació que en el segon ens guardem les distàncies de cada node al node inicial. En el primer, ens guardem la distància al predecessor. El de Prim i Kruskal també tenen diferències fonamentals: així com en el de Kruskal les arestes eren afegides o descartades de la solució, pel cas de Prim una aresta pertany a la solució fins que no n'aparegui una de millor.*

# 3 Exercicis

**Exercici 3.1.** *Llista i nombre de components.*

```
def aux_dfs(G, node, visited, cc):
    if node not in visited:
        visited.add(node)
        cc.append(node)
        for nei in G.neighbors(node):
            # Crida recursiva
            visited, cc = aux_dfs(G, nei, visited, cc)
    return visited, cc

def connected_components(G):
    visited = set()
    connected_lst = []
    for n in G.nodes():
        if n not in visited:
            cc = []
            visited, cc = aux_dfs(G, n, visited, cc)
            connected_lst.append(cc)

    return len(connected_lst), connected_lst
```

**Exercici 3.2.** *Detectar cicles.*

```
def dfs_cycles(G, visited, current_node, previous_node):
    if current_node not in visited:
        visited.add(current_node)
        for nei in G.neighbors(current_node):
            if nei not in visited:
                if dfs_cycles(G, visited, nei, current_node):
                    return True
            elif previous_node != nei: return True
    return False
def cycles(G):
    n = list(G.nodes())[0]
    visited = set()
    return dfs_cycles(G, visited, n, None)
```

**Exercici 3.3** (Problema de tornar canvi). *Disposant d'un sistema monetari amb monedes d'n valors diferents, $1 = v_1 < \cdots < v_n$, trobar el mínim nombre de monedes per tornar un canvi de certa quantitat $K$.*

*Resolució.* Es tracta d'un algorisme voraç caracteritzat per utilitzar que es torna sempre la moneda més gran que es pugui. Això només és així si $v_i \geq 2v_{i-1}, \forall i = 2, \ldots, n$. ■

**Exercici 3.4** (Problema de la motxilla). *D'entre $n$ objectes que tenen pesos $w_i \in \mathbb{R}$, per $i = \{1, \ldots, n\}$ i valors $v_i \in \mathbb{R}$, aconseguir el màxim valor possible, sempre que el pes total no superi la capacitat de pes $W$ de la motxilla.*

*Resolució.* Escollim l'element de valor més elevat sense tenir en compte el pes o bé escollim l'element que té un millor equilibri entre pes i valor. Inicialitzem el pes actual de la motxilla, el valor acumulat fins el moment i els ítems que hi hem carregat. Escollim el millor element. Mentre existeixin elements que puguem afegir (I) eliminem l'element de la llista, (II) modifiquem els valors que emmagatzemen la informació i (III) busquem un nou element. ■

```
def union(parent, rank, node1, node2):
    if rank[node1] <= rank[node2]: parent[node1] = node2
        if rank[node1]==rank[node2]: rank[node2]+=1
    else: parent[node2] = node1
def find(parent, node):
    while parent[node] != node: node = parent[node]
    return node
def find_compressed(parent, node):
    if parent[node]!=node: parent[node] = find_compressed(pare
    return parent[node]
def union_find(lst):
    list1, list2 = zip(*lst); unique_nodes = set(list1+list2)
    rank = defaultdict(int); parent = {n: n for n in unique_no
    parent_states = []
    for node1, node2 in lst:
        parent1 = find(parent, node1); parent2 = find(parent,
        if parent1 == parent2: continue
        union(parent, rank, parent1, parent2)
        parent_states.append(parent.copy())
    return parent, parent_states
```