

# 1 Camins i algorismes

Sovint, quan utilitzem un graf per modelitzar quelcom, ens interessa poder-hi fer algunes operacions. Podem, per exemple, voler trobar un camí entre dos punts, recórrer el graf sencer o trobar el camí més curt per anar d'un vèrtex a un altre. Per aquest motiu utilitzem els camins, que trobarem o generarem mitjançant diversos algorismes. En aquesta secció mostraré diverses maneres de recórrer un graf, torbant la manera més eficient per a cada cas.

## 1.1 Grafs ponderats i dirigits

### Grafs ponderats

Els grafs ponderats són grafs on cada aresta  $e$  està associada a un nombre  $w(e)$  anomenat pes o cost, tal que  $w(e) \in \mathbb{R}$ . El pes pot representar diverses quantitats, segons el que es vulgui modelitzar. Moltes vegades s'utilitza per representar distàncies, però si per exemple modelitzem una xarxa de distribució d'aigua, ens pot interessar representar el cabal de les canonades, o en una xarxa de bus, la densitat de trànsit de cada tram.

### Grafs dirigits

Els grafs dirigits són grafs les arestes dels quals només admeten una direcció. D'aquesta manera, una aresta  $e_0 = (v_0, v_1) \neq e_1 = (v_1, v_0)$ , contràriament als grafs no dirigits. DE fet, no necessàriament ha d'existir una aresta contrària a una altra. Aquest tipus de grafs poden ser útils per representar carreteres, o moviments vàlids en algun joc.

## 1.2 Camins

Un camí  $p$  és una seqüència finita i ordenada d'arestes que connecta una seqüència ordenada de vèrtexs. Un camí  $p$  de longitud  $k$  (expressat com a  $l(p) = k$ ) entre el vèrtex inicial  $v_0$  i el vèrtex final  $v_k$  sempre que  $v_0 \neq v_k$ ) és una successió de  $k$  arestes i  $k + 1$  vèrtexs de la forma  $\overline{v_0, v_1}, \overline{v_1, v_2}, \dots, \overline{v_{k-1}, v_k}$ . Per definició, també es pot representar un camí  $p$  entre  $v_0$  i  $v_k$  com a successió de vèrtex  $p = v_0 v_1 \dots v_k$ . En aquest cas, pot ser tractat com un graf elemental  $P_n$ . Un cas especial és quan el camí comença i acaba al mateix vèrtex ( $v_0 = v_k$ ). Llavors el camí és un cicle, i és l'equivalent a un graf cicle  $C_n$ . Quan un camí té totes les arestes diferents, s'anomena simple, i si a més té tots els vèrtexs diferents, s'anomena elemental.

En els grafs, ponderats, la longitud d'un camí  $c = v_0, v_1, \dots, v_n$  no es defineix pel nombre d'arestes per on passa el camí, sinó fent el sumatori dels pesos de les arestes

$$\text{longitud}_w(c) = \sum_{i=0}^{n-1} w(\overline{v_i, v_{i+1}})$$

La distància entre dos vèrtexs  $v$  i  $u$ ,  $d_w(v, u)$ , és la que s'obté al agafar la menor longitud d'entre tots els camins elementals entre  $v$  i  $u$ . (adjuntar exemple de distància)

### 1.3 Algorismes

Un algorisme és un conjunt d'instruccions precises i ben definides que, donada una entrada, calculen la sortida corresponent segons les instruccions que té. A continuació

#### 1.3.1 BFS

Aquest algorisme serveix per examinar l'estructura d'un graf o fer-ne un recorregut sistemàtic. La recerca per amplada prioritària (*breadth-first search* en anglès, d'aquí **BFS**) fa l'exploració en paral·lel de totes les alternatives possibles per nivells des del vèrtex inicial. A la següent imatge es pot veure com funcionaria aquest algorisme en un graf: (Adjuntar imatge de BFS)

Per programar aquest algorisme s'acotuma a utilitzar un contenidor de tipus cua, que només permet afegir elements al final de la cua i treure'n de l'inici, sense poder accedir a elements del mig de la cua. El que farà això és bàsicament imprimir per pantalla la seqüència de vèrtex ordenada segons l'ordre en que els ha visitat.

---

#### Algorisme 1: BFS

---

**Data:** Un graf  $G$  i un node inicial  $v$   
**Result:** Seqüència de nodes visitats  
 nova cua  $Q$ ;  
 marca  $v$  com a visitat;  
 imprimeix( $v$ );  
 afegeix  $v$  a la cua  $Q$ ;  
 nou node *auxiliar*;  
 nou node *següent*;  
**while** la cua no estigui buida **do**  
     *auxiliar* = primer element de  $Q$ ;  
     imprimeix(*auxiliar*);  
     elimina(primer element de  $Q$ );  
     **while** hi hagi nodes adjacents a *auxiliar* i aquests no s'hagin visitat  
         **do**  
             marca adjacent(*auxiliar*) com a visitat;  
             afegeix adjacent(*auxiliar*) a la cua;  
         **end**  
     **end**  
**foreach** node de  $G$  **do**  
     marca'l com a no visitat  
**end**

---

Aquesta és una manera bastant usual de programar el BFS, i encara que és eficient, estem desaprofitant propietats de l'algorisme. Amb BFS podem saber a quina distància del punt inicial està cada node, el camí més curt per anar del node inicial a qualsevol altre i fins i tot podem generar un arbre expansiu mínim. El següent algorisme té en compte aquests detalls. Està pensat per ser implementat en el llenguatge Python, i per aquest motiu utilitza diccionaris (l·listes on cada element té un nom i una clau), però en llenguatges basats en C, es poden utilitzar maps de la mateixa manera.

---

```

Data: Un graf  $G$  i un node inicial  $v$ 
Result: Seqüència de nodes visitats, distància de cada node respecte  $v$ 
nou diccionari  $dist$ ;
 $dist[v] = 0$ ;
nou diccionari  $anterior$ ;
 $anterior[v] = Nul$ ;
 $i = 0$ ;
nova llista  $frontera$  afegeix  $v$  a  $frontera$ ;
imprimeix( $v$ );
while  $frontera$  no estigui buida do
    nova llista  $següent$ ;
    foreach node  $x$  de  $frontera$  do
        /* A cada iteració,  $x$  agafarà un valor diferent de  $frontera$  */
        foreach node  $y$  adjacent a  $x$  do
            if  $y$  no existeix dins  $dist$  then
                 $dist[y] = i$ ;
                 $anterior[y] = x$ ;
                afegeix  $y$  a  $següent$ ;
                imprimeix( $y$ );
            end
        end
    end
     $frontera = següent$ ;
     $i = i + 1$ ;
end
imprimeix( $dist$ );

```

---

Encara que aquest algorisme sembli molt senzill, ens pot aportar informació important, i fins i tot permet resoldre problemes senzills on haguem de trobar distàncies o el camí més curt entre dos nodes. Tot i això aquest algorisme s'utilitza també per operacions més complexes:

- Google l'utilitza per indexar pàgines web noves al seu buscador. Amb BFS es pot recórrer tota la xarxa d'internet sencera, i si cada pàgina web és un node i cada enllaç és una aresta, si es posa un link d'una pàgina no indexada a una que sí ho està, l'algorisme trobarà el nou node).

- Les xarxes socials l'utilitzen per suggerir-te amistats. Amb BFS poden trobar els amics dels teus amics (els nodes que estàn a distància 2 de tu), que són susceptibles a ser amics teus. Com més amics teus tingui la persona a distància 2 de tu, més probable és que també la coneguis.
- Es pot sil·lucionar un cub de rubik amb aquest algorisme. Si s'aconsegueix generar un graf on cada node sigui un estat diferent del cub i les arestes siguin un moviment d'una cara, donat un estat inicial, amb BFS arribes a l'estat resolt amb els mínims moviments possibles.

### 1.3.2 DFS

La recerca per profunditat prioritària (*depth-first search* en anglès, d'aquí **DFS**) és un algorisme que utilitza uns principis semblants al BFS, però en lloc de cobrir tota l'amplada d'un nivell abans de passar al següent, el que fa és cobrir tota la profunditat possible (arribar el més lluny possible) abans de tornar enrere. En la següent imatge es pot veure un esquema del funcionament de l'algorisme: (adjuntar imatge de DFS) Tal com en el BFS, també hi ha diverses maneres de fer l'algorisme, i jo en presentaré dues. La primera utilitza un contenidor de tipus pila, on només es pot manipular, afegir o treure l'element de dalt de tot de la pila.

---

#### Algorisme 2: DFS

---

**Data:** Un graf  $G$  i un node inicial  $v$   
**Result:** Seqüència de nodes visitats  
nova pila  $S$ ;  
nou node *següent*;  
marca  $v$  com a visitat;  
imprimeix( $v$ );  
afegeix  $v$  a la pila  $S$ ;  
**while** la pila no estigui buida **do**  
    *següent* = node adjacent no visitat de l'element superior de  $S$ ;  
    /\* En cas que no n'hi hagi cap, *següent* = *Nul* \*/  
    **if** *següent* = *Nul* **then**  
        elimina(element superior de  $S$ );  
    **else**  
        marca *següent* com a visitat;  
        imprimeix(*següent*) afegeix *següent* a  $S$   
    **end**  
**end**

---