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, recòrrer 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 mitjnçant diversos algorismes. En aquesta secció mostraré diverses maneres de recòrrer 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 asociada 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 $\overline{v_0, v_1}, \overline{v_1, v_2}, \cdots, \overline{v_{k-1}, v_k}$. Per definició, també es pot representar un camí p entre v_0 i v_k com a successió de vèrtex $p = v_0 v_1 \cdots v_k$. En auqest 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 $c = v_0, v_1, \cdots, v_n$ no es defineix pel nombre d'arestes per on passa el camí, sinó fent el sumatori dels pesos de les

arestes

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

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. (adjunjtar 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 s'en mostren uns quants d'importants.

1.3.1 BFS

Aquest algorisme serveix per examinar l'estructura d'un graf o fer-ne un recorregut sistemàtic. La recerca per amplada prioritàtia (*breadht-first serch* en anglès, d'aquí **BFS**) fa l'exploració en paral·lel de de totes les alternatives posibles 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 vèrtexs ordenada segons l'ordre en que els ha visitat.

Aquesta és una manera bastant usual de programar el BFS, i encara que ès eficient, s'està desaprofitant propietats de l'algorime. Amb BFS es pot 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 es pot generar un arbre expansiu mínim, agafant les arestes per on passa el BFS. El següent algorisme té en compte aquests detalls. Està pensat per ser implementat en el llengüatge Python, i per aquest motiu utilitza diccionaris (llistes on cada element té un nom i una clau), però en llenguatges basats en C, es poden utilitzar maps de la mateixa manera.

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. Aquest algorisme s'utilitza també per operacions més complexes, com les següents:

- Google l'utilitza per indexar pàgines web noves al seu buscador. Amb BFS pot recòrrer 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).

Algorisme 1: BFS

```
Data: Un graf G i un node inicial v
Result: Seqüència de 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
```

- Les xarxes socials l'utilitzen per suggerir amistats. Amb BFS poden trobar els amics d'una persona (els nodes que estàn a distància 2 d'aquesta), que són susceptibles a ser amics seus. Com més amistats en comú amb la persona a distància 2, més probable és que es coneguin.

- 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 algorime que utilitza uns proncipis 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 prfunditat 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,

```
Data: Un graf G i un node inicial v
Result: Seqüència de nodes visitats, distància de cada node resperce 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)
```

i en presentaré dues. La primera utilitza un contenidor de tipus pila, on només es pot manipular, afegi o treure l'element de dalt de tot de la pila.

Aquest mètode, però, té un problema, i és que només funciona per a grafs no dirigits. Hi ha la possibilitat que, treballant amb un graf dirigit, un node del graf no sigui accessible des del node inicial que hem determinat. Aquest cas excepcional es pot arreglar fent que cada node no visitat per les iteracions anteriors sigui l'inicial. El segon algorisme, a part d'arreglar això, utilitza una funció recursiva (una funció que es crida a si mateixa).

Aquest algorisme no té tantes utilitats pràctiques com el BFS, però també té propuietats utils, com per exemple, que passa per totes les arestes.A través d'ell podem obtenir informació importanat d'un graf:

- Es pot saber si un graf té cicles, comprovant si quan estem a l'iteració d'un

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
```

node (encara no n'hem explorat tots els nodes adjacents) el trobem a ell mateix. En cas afirmatiu voldrà dir que hi ha un cicle.

- Es pot saber si un graf és bipartit assignant un color (entre un total de 2 colors possibles) a cada node mentre es recorre el graf, de manera que un node tingui un color diferent al dels nodes adjacents. Si això és possible voldrà dir que el graf és bipartit.

- Es pot dur a terme una ordenació topològica, si es tracta d'un graf dirigit sence cicles. Un exemple d'ordenció topològica és quan hi ha una llista de tasques a fer però per fer-ne una determinada, cal haver-ne fet primer una altra. Amb el DFS, podem obtenir una de les seqüències vàlides per completar totes les tasques. (Adjuntar exemple d'ordenació topològica i la seva sol·lució donada mitjancant DFS)

1.3.3 Dijkstra

Aquest algorisme, desdenvolupat per Edsger W. Dijkstra el 1956, serveix per trobar el camí més curt entre dos nodes d'un gaf ponderat. De fet, això és el que feia la variant original, però la variant presentada aquí troba el camí més curt entre

```
Data: Un graf G
Result: Seqüència de nodes visitats des de cada node
nou diccionari anterior
nova llista ordre
foreach node u del graf do
    if u no existeix dins anterior then
        imprimeix(u)
        anterior[u] = Nul
        DFSrecursiu(G, u)
    end
end
inverteix ordre
imprimeix(ordre)

/* La funció DFSrecursiu queda determinada pel següent algorisme:       */
Funció DFSrecursiu(G, v)
    foreach node x adjacent a v do
        if x no existeix a anterior then
            imprimeix(x)
            anterior[x] = Nul
            DFSrecursiu(G, x)
        end
    end
    /* Només si es vol obtenir la seqüència de recursió o ordenació
       topològica per a grafs dirigits acíclics,                        */
    afegeix v a ordre
```

un node inicial i tota la resta de nodes dels graf. Tot i això es pot modificar lleugerament el programa perquè pari quan hagi trobat el camí més curt entre dos nodes especificats. El que fa el programa és suposar quines són les distàncies mínimes des del node inicial fins la resta, i va descobrint el graf fins que pot assegurar el camé més curt. Al principi, suposa que el moviment amb menys cost és no moure's (això serà cert si el graf no conté arestes amb pesos negartius, que l'algorisme no pot tractar). Després busca els nodes adjacents al node inicial, i suposa que les distàncies mínimes entre l'inicial i aquests és simplement l'aresta que els uneix. Es pot assegurar que serà cert pel node unit amb l'aresta de menys pes, per la desigualtat triangular (si s'hi pogués accedir per un altre camí, aquest seria més llarg, ja que per força alguna de les arestes amb més pes que s'han descartat ha de formar part del camí alternatiu, i la suma serà sempre superior). Un cop hi ha el primer node amb mínim pes assegurat (i per tant ja sap un camí), es busquen els seus adjacents, el pes dels quals inicialment és infinit. La suposició del pes equivaldrà a el pes del node d'on venen més el de l'aresta que els uneix, i es canviarà pel pes que tenen si aquest és mes gran que el nou. Ara es torna a agafar el node amb el pes menor (que segur que és el mínim) i es torna a mirar els adjacents i assignar pesos. Quan tots els nodes hagin estat visitats, el pes de cada node serà la distància mínima que s'ha de recòrrer per anar del node inicial fins a aquest.

(Adjuntar esquema de procediment de Dijkstra)

Aquest algorisme, encara que no pot treballar amb pesos negatius és molt útil té una gran quantitat d'aplicacions pràctiques:

- Navegadors GPS, on les arestes són carrers i carreteres, els nodes cruïlles i els pesos distàncies. S'utilitza l'algorisme de Dijkstra per trobar els camins més curts entre dues destincaions.

- Problemes de canvis de divisa, on volem trobar la miillor manera de canviar divises i guanyar més diners. Aquí els nodes són les diferents monedes o divises, les arestes les transaccions i els pesos les taxes de canvi. Amb aquest algorisme podem trobar la millor manera de fer els canvis de moneda.

- Els routers utilitzen l'algorisme per portar-te a través d'internet al servidor desitjat amb la menor cantitat de passos possibles.

- En robòtica s'utilitza per fer la planificació de moviment del robot. Cada node és una unitat d'espai, i omplint tot l'espai de nodes excepte els obstacles i executant l'algorisme en el graf resultant, s'obté el camí més òptim per arribar a la posició desitjada.

- En epidemologia es pot utilitzar per modelitzar un grup de persones i els

seus familiars per veure qui és més susceptible a emmalaltir. Això també pot funcionar entre ciutats o col·lectius més grans.

Algorisme 3: Dijkstra

```
Data: Un graf ponderat G i un node inicial s
Result: Distància mínima entre s i la resta de nodes del graf, arbre
        expensiu mínim
nou diccionari dist
nou diccionari Q
foreach node v de G do
    Q[v] = ∞
    dist[v] = ∞
end
Q[s] = 0
while Q no estigui buit do
    u = min{valor de Q}
    dist[u] = Q[u]
    foreach node v adjacent a u do
        if v existeix dins Q then
            if Q[v] > Q[u] + w(u, v) then
                /* w(u, v) és el pes de l'aresta {u, v}          */
                Q[v] = Q[u] + w(u, v)
            end
        end
    end
    elimina(Q[u])
end
imprimeix(dist)
```

1.3.4 Bellman-Ford

Aquest algorisme té un funcionament i utilitats molt seblants a les de l'algorisme de Dijkstra, però té la particularitat de poder tractar sense problemes les arestes amb pesos negatius, mentre que Dijkstra no ho permet. Dijkstra es basa en la desigualtat triangular per tobar el camímés curt, però amb pesos negatius no es pot suposar que la desigualtat triangular es compleixi. A més, permet saber si un graf cinté cicles negatius. Si un camí entre dos nodes conté un cile de pes negatiu, no es pot trobar un camí mínim entre aquests dos nodes. Això es deu

a que recorrent aquest cicle sempre es podria escurçar el camí, i llavors el mínim possible seria de $-\infty$.

Algorisme 4: Bellman-Ford

```
Data: Un graf ponderat G i un node inicia s
Result: Distància mínima entre s i la resta de nodes del graf, arbre
        expansiu mínim
nou diccionari dist
nou diccionari anterior
foreach node v de G do
    dist[v] = ∞
    anterior[v] = Nul
end
dist[s] = 0
for i in range(0, len(Adj)-1) do
    foreach u dins Adj do
        foreach v dins Adj[u] do
            if dist[v] > dist[u] + w(u, v) then
                /* w(u, v) és el pes de l'aresta {u, v}                    */
                dist[v] = dist[u] + w(u, v)
            end
        end
    end
end
foreach u dins Adj do
    foreach v dins Adj[u] do
        if dist[v] > dist[u] + w(u, v) then
            imprimeix("Hi ha cicles de pesos negatius")
        end
    end
end
imprimeix(dist)
imprimeix(anterior)
```

1.3.5 Kruskal

L'algorisme de Kruskal serveix per trobar un arbre expansiu mínim. Aquest algorisme utilitza una estructura de dades especial, anomenada union-find. Aquesta estructura permet fer tres operacions diferents: crear conjunts (Make Set), de-

terminar a quin conjunt està un element (Find) i unir dos subconjunts en un de nou (Union). L'ús d'aquesta estructura especialitzada fa que en grafs petits o poc densos l'algorisme sigui molt ràpid, però en grafs més grans es relantitzi el procés, siguent més eficient utilitzar altres algorismes en aquest cas. L'algorisme, al principi crea un conjunt per a cada node i ordena les arestes de manera creixent segons elsseu pes. Llavors agafa la primera aresta (la de menys pes), que serà una aresta de l'arbre expansiu mínim, i posa en el mateix conjunt els nodes que unia. Després agafa la següent aresta i repeteix el procediment.

Data: Un graf G i un node inicial s
Result: Arbre expansiu mínim de G
nova estructura union-find *subgraf*
nova llista *arbre* ordena G per ordre creixent de pesos
foreach *node u de G* **do**
  foreach *node v adjacent a u* **do**
    if $subgraf[u] != subgarf[v]$ **then**
      afegeix (u, v) a *arbre*
      $union(subgraf[u], subgraf[v])$
    end
  end
end
imprimeix(*arbre*)

1.3.6 Prim

L'algorisme de Prim, tal com del de Kruskal, serveix per trobar l'arbre expansiu mínim d'un graf ponderat no dirigit. Aquest algorisme funciona amb diccionaris, estructures de dades més normalitzades que el Union-Find. Com a conseqüència, Prim és més lent en grafs petits, però més ràpid en grafs molt densos. Prim divideix els nodes en dos grups, els que pot arribar amb les arestes de l'arbre que va construïnt i els que encara no. Sempre selecciona l'aresta de menys pes entre les que surten de nodes del primer grup i van a nodes del segon, i afegeix el node final de l'aresta al primer grup. D'aquesta manera obté l'arbre expansiu mínim del graf. Aquest algorisme funciona per grafs connexos, peró executant-lo per a cada node del graf, trobaria el bosc (conjunt d'arbres) mínim d'un graf no connex.

Tant l'algorisme de Kruskal com del de Prim tenen aplicacions semblants, però segons la mida del graf és més convenient utilitzar-ne un o altre. Entre les aplicacions d'aquests dos algorismes hi ha:

Algorisme 5: Prim

```
Data: Un graf G
Result: Arbre expansiu mínim de G
nou diccionari anterior
nou diccionari Q
foreach node v de G do
    Q[v] = ∞
end
Q[0] = 0
while Q no estigui buit do
    u = min{valor de Q}
    foreach node v adjacent a u do
        if v existeix dins Q then
            if Q[v] > w(u, v) then
                Q[v] = w(u, v)
                anterior[v] = 0
            end
        end
    end
    elimina(Q[u])
end
imprimeix(anterior)
```

- S'utilitzen per dissenyar xarxes de telèfon, aigua, gas, internet... En aquestes xarxes s'ha d'arribar a tots els punts on s'ha de fer la distribució, i amb l'arbre expansiu mínim es pot assegurar que la xarxa és el més curta possible.
- Els arbres expansius mínims es poden utilitzar per generar laberints.
- S'utilitzen com a subrutines (o funcions) d'algorismes més complexes.

1.3.7 Floyd-Warshall

L'algorisme de Floyd-Warshall és un algorisme que permet calcular les distàncies entre tots els nodes d'un graf ponderat. Per fer-ho, compara els pesos de tots els camins possibles. A cada iteració, es defineix el conjunt de nodes que pot tenir cada camí i si ja s'ha trobat un camí amb els mateixos extrems es compara el pes total d'ambdós . El conjunt és de la forma $1, 2, \cdots, k$ i a cada iteració es va incrememntant k (a l'inici $k = 0$). El resultat d'aquest algorisme és una matriu quadrada D de $|V|$x$|v|$ on $D_{i,j} = w(i, j)$.

Algorisme 6: Floyd-Warshall

```
Data: un graf G
Result: una matriu quadrada amb les distàncies entre tots els nodes
nova matriu dist de |V|x|V|
for i in range(0, |V|) do
    for j in range(0, |V|) do
        if i = j then
            dist[i][j] = 0
        else
            dist[i][j] = ∞
        end
    end
end
foreach node u de G do
    foreach node v adjacent a u do
        dist[u][v] = w(u, v)
    end
end
for x in range(0, |V|) do
    for u in range(0, |V|) do
        for v in range(0, |V|) do
            if dist[u][v] > dist[u][x] + dist[x][v] then
                dist[u][v] = dist[u][x] + dist[x][v]
            end
        end
    end
end
```

Aquest algorisme es pot utilitzar per qualsevol de les aplicacions en que s'utilitzaria Dijkstra en més d'un node. S'utilitza sobretot quan es vol mantenir una base de dades de pesos precalculats, per no haver d'executar Dijkstra en cada cas concret. A part d'aquesta aplicació, s'utilitza també d'altres maneres:

- Per detectar cicles de pes negatiu, cosa que passarà quan $D_{i,i} < 0$, quan a la diagonal de la matriu hi hagi un valor negatiu.
- Estudiar la clausura transitiva d'un graf, és a dir, veure quins nodes són accessibles des de cada node. Això es pot veure a la matriu resultant, on els valors ∞ indiquen que no espot accedir a el node concret.