Teoria de Grafs

Revisió segona

Aniol Garcia i Serrano

23/9/2016

Índex

Part I
Introducció a la teoria de grafs

Aquest primer apartat consisteix en la part més teòrica del treball. Explicaré breument la història d'aquesta branca de la matemàtica i posteriorment ens endinsarem en la teoria de grafs com a tal. La teoria de grafs pot semblar bastant abstracta i, a vegades, complicada d'entendre, però estarà acompanyada de demostracions i exemples propis que pretenen facilitar el seguiment del treball.

(Reescriure la introducció)

1 Història del grafs

1.1 Els primers passos

Tot sovint, les noves branques de la matemàtica sorgeixen de cercar solucions a problemes. Problemes que no poden ser resolts ni demostrats amb el que coneixem i que forcen a desenvolupar nous mètodes i teories. La teoria de grafs no n'és una excepció i tot seguit presentaré els problemes que van forjar la creació d'aquesta branca.

Euler i els ponts de Königsberg

La teoria de grafs neix a partir de la solució d'un problema curiós: el problema dels ponts de Königberg (l'actual Kaliningrad, Rússia). El planteig del problema és el següent:

> *"El riu Pregel divideix Königsberg en quatre parts separades, i connectades per set ponts. És possible caminar per la ciutat passant per tots els ponts tan sols una vegada?"*

Els ciutadans de Königsberg sabien que no era possible, però mai ningú ho havia demostrat fins que Leonhard Euler ho va fer. La demostració de que això no era possible queda recollida en el *"Solutio problematis ad geometriam situs pertinentis"* publicat el 1736, i l'article també va ser inclòs en el volum 8 de *"Commentarii Academiae Scientiarum Imperialis Petropolitanae"* publicat el 1741 [EUL41]. En aquest text, Euler diu el següent:

> *"A més d'aquella part de la geometria que s'ocupa de les quantitats, la qual sempre ha generat un interés preferent, hi ha una altra part -encara pràcticament desconeguda- que ja fou esmentada per Leibniz amb el nom de geometria de posició. Aquesta part de la geometria*

estudia tot allò que pot ésser determinat únicament per la posició, així com les propietats de la posició; en aquest camp hom no hauria de preocupar-se de quantitats ni de com obtenir-les. No obstant això, els tipus de problemes que pertanyen a aquesta geometria de posició i els mètodes usats per resoldre'ls encara no són suficientment definits. Així doncs, quan darrerement se'm va plantejar un problema que semblava bàsicament geomètric, però que no demanava l'obtenció de quantitats ni admetia una sol·lució basada en el càlcul de quantitats, em vaig adonar que pertanyia a la geometria de posició, sobretot perquè només es podia usar la posició per resolde'l, mentre que el càlcul es mostrava inútil. Així, he decidit exposar ací, com a mostra de la geometria de posició, el mètode que he deduït per a resoldre problemes d'aquesta mena."

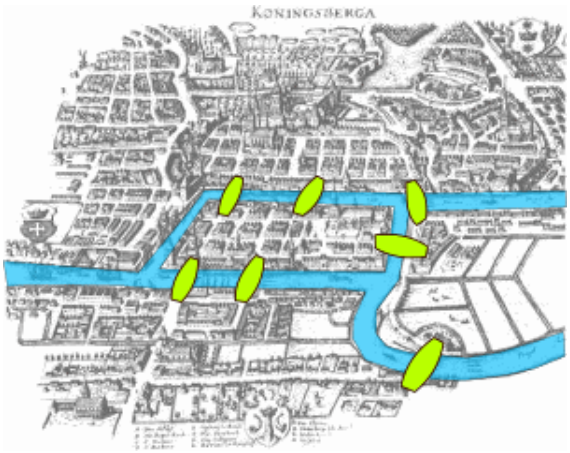


Figura 1: Representació dels ponts de Köingsberg [1]

Per aconseguir demostrar que el problema no tenia sol·lució, Euler va haver de representar el problema com un mapa topològic, posant les masses de terra com a punts i els ponts com a segments que unien aquests punts, i creant d'aquesta manera el primer graf de la història. Euler va observar que, exceptuant el punt inicial i final del trajecte, quan s'arriba a un punt per un pont, s'ha de sortir per un altre. Això significa que el nombre de vegades que s'entra a un punt equival al nombre de vegades que cal sortir-ne (exceptuant, tal com abans, el punt inicial i final), cosa que no és possible si no hi ha un nombre parell de ponts que vaigin a un punt. Cal observar que en el problema original hi ha 5 masses de terra, i totes elles tenen un nombre imparell de ponts o connexions, i per tant no és possible

[1] Autor: Bogdan Giuşcă. Imatge de domini públic sota la llicència CC BY-SA 3.0 i GNU Free Documentation License (versió 1.2)

passar per tots els ponts tan sols una vegada. Euler va arribar a la conclusió de que es pot recòrrer un graf passant només una vegada per les seves arestes només si aquest tenia 0 o 2 arestes de grau senar, corresponents al punt final i inicial. Aquest resulat es considera el primer en teoria de grafs ja que conté un important teorema d'aquesta branca. A més d'iniciar la teoria de grafs, amb aquest resultat també comença l'estudi dels grafs planars, introdueix el concepte de característica d'Euler de l'espai i el teorema de poliedres d'Euler (teorema que després va utilitzar per demostrar que no hi havia mes sòlids regulars que els sòlids platònics). Amb tot això, Euler posa les bases, no tan sols de l'estudi dels grafs, sinó també de la topologia.

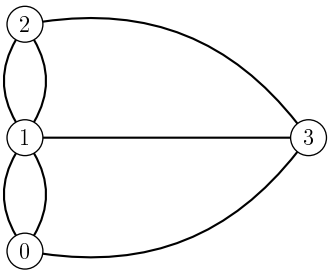


Figura 2: Representació topològica dels ponts de Königsberg

Vandermonde i el tour del cavall d'escacs

A partir de l'article d'Euler, diversos matemàtics van començar a interessar-se pel camp de la topologia (o geometria de la posició, com li deien en aquell moment). Un d'ells fou Alexandre-Théophile Vandermonde. Vandermonde va treballar i estudiar el problema dels cavalls, que pregunta quins moviments hem de fer per tal que un cavall passi per totes les caselles del tauler d'escacs, problema que també va tractar Euler. Els estudis que Vandermonde va fer sobre aquest problema van ser publicats el 1771 en el "*Remarques sur des problèmes de situation*" [**VAN71**], i per la relativa proximitat als treballs d'Euler, encara no parlava de grafs, tot i que en l'actualitat el problema es resol mitjançant aquests. Aquest treball inicia l'estudi de la teoria de nusos, una altra branca de la topologia.

1.2 Les primeres descobertes i aplicacions

Un cop establert l'inici, és difícil veure com va continuar desesenvolupant-se la teoria, ja que en els primers treballs es volien resoldre problemes concrets sense establir relacions entre ells. Tot i així, durant el segle XIX es van plantejar una gran qüantitat de problemes i es van desenvolupar molts teoremes referents els grafs.

Francis Guthrie

El 1852, Francis Guthrie, matemàtic britànic, es planteja el següent problema mentre intenta pintar un mapa del Regne Unit:

> *"És possible pintar qualsevol mapa de països de tal manera que un país tingui un color diferent al de tots els seus veïns, utilitzant tan sols quatre colors?"*

D'aquest problema en surt el teorema a partir del qual s'sestableix que qualsevol mapa pot ser pintat únicament amb quatre colors diferents, de tal manera que dues regions adjacents (entenem com a adjacents dues regions que comparteixiin frontera, no tan sols un punt) no tinguin colors iguals. Aquest problema, que pot semblar tan trivial, no va ser demostrat fins l'any 1976. Va passar per mans de pioners com De Morgan, Hamilton, Cayley, Kempe (que va fer una demostració publicada el 1879 [KEM79]), Heawood (que va concloure que la demostració de Kempe no era correcta)... Finalment el 1976 Appel i Hanken van demostrar, mitjançant un programa d'ordinador, que tot mapa es podia pintar només amb quatre colors [AH77]. Pel fet de basar-se en un programa d'ordinador, no va acabar de convèncer la demostració. Així doncs, aquest problema no va ser solucionat de manera formal fins l'any 1996 quan, recorrent a la teoria de grafs ja desenvolupada, Neil Robertson, Daniel P. Anders, Paul Seymour i Robin Thomas van publicarne un demostració. En els treballs d'Appel i Hanken es van definir alguns dels conceptes i fonaments de l'actual teoria de grafs.

Arthur Cayley

Arthur Cayley, matemàtic que treballava en la teoria de grups, topologia i combintòria, també va aportar una gran quantitat de coneixement a la teoria de grafs. Va treballar amb grafs de tipus arbre i va desenvolupar la fòrmula n^{n-2}, que determina el nombre d'arbres expansius que té un graf complet de n vèrtex (veure apartat 3.7). Una fórmula semblant apareixia en treballs de Carl Wilhelm Borchardt, en els quals Cayley es va basar i va extendre, tot i que el que actualment dóna nom a la fórmula és el mateix Cayley.

També va treballar en el desenvolupament d'una representació de l'estructura abstracta d'un grup, creant els grafs de Cayley i el teorema de Cayley. Finalment va contribuir també, l'any 1857, en la representació i ennumeració dels isòmers alcans (composts químics que comparteixen fórmula o composició però tenen diferent estructura molecular), representant cada compost mitjançant un graf de tipus arbre. Tot i això, Cayley no només va ser actiu en teoria de grafs, sinó que també va desenvolupar teoremes en àlgebra lineal, topologia i geometria.

William Hamilton i Thomas Kirkman

William Rowan Hamilton va plantejar un problema el 1859 que consistia a trobar un camí que passés pels 20 vèrtex d'un dodecàedre una sola vegada a través de les seves arestes. Hamilton va comercialitzar el joc sota el nom de "The Icosian game" (és important dir que el nom de icosian no va ser degut a que utilitzés un icosaedre, sinó que feia referència als 20 vèrtexs del dodecaedre per on s'havia de passar). Entorn aquest joc existeix un gran controvèrsia, ja que Euler anteriorment havia plantejat un problema semblant mentre estudiava el problema dels cavalls d'escacs, i Kirkman va plantejar exactament el mateix problema que Hamilton a la Royal Society un temps abans.

Gustav Kirchhoff

Gustav Kirchhoff, conegut majoritàriament en el camp de l'electrotècnia per les lleis de Kirchhoff, també va fer aportacions importants a la teoria de grafs. Les seves lleis, publicades el 1874, es basen en la teoria de grafs, però a més, va ser el primer d'utilitzar els grafs en aplicacions industrials. Va estudiar sobretot els grafs de tipus arbre i, amb l'investigació que va dur a terme sobre aquest tipus de grafs, va formular el teorema de Kirchhoff, referent al nombre d'arbres d'expansió que es poden trobar en un graf. Aquest teorema es considera una generalització de la fórmula de Cayley.

1.3 Teoria de grafs moderna

Durant el segle XX, la teoria de grafs es va continuar desenvolupant. Amb les bases ja establertes durant el segle XIX, els matemàtics hi van començar a treballar i el 1936 Dénes König va escriure el primer llibre de teoria de grafs. Frank Harary va escriure un altre llibre el 1969, fent més accessible la teoria de grafs. El desenvolupament de l'informàtica i les noves tècniques de computació van permetre treballar amb grafs a molt més gran escala, fent possible, per exemple, la ja citada primera demostració del teorema dels quatre colors per Appel i Hanken.

Actualment la teoria de grafs és una part molt important de la matemàtica discreta i està relacionada amb molts àmbits diferents, com per exemple la topologia, la combinatòria, la teoria de grups, la geometria algebraica... Des del seu desenvolupament s'han utilitzat els grafs per resoldre i representar de manera visual problemes en aquests camps. Té aplicacions en molts altres àmbits com per exemple la computació, la informàtica, la física, la química, l'electrònica, les telecomunicacions, la biologia, la logística i fins i tot en l'àmbit econòmic.

2 Principis bàsics

Un *graf* $G = (V, E)$ es defineix com un conjunt de *vèrtexs* (o nodes) $V = \{v_1, v_2, ..., v_n\}$ i un conjunt d'*arestes* $E = \{e_1, e_2, ..., e_m\}$, cadascuna de les quals uneix dos vèrtex de V. Si v_i i v_j, amb $v_i, v_j \in V$, estan units per l'aresta e_k escriurem $e_k = \{v_i, v_j\}$. Quan volguem especificar el graf concret empararem les notacions $V(G)$ i $E(G)$. Així doncs un graf està format per un conjunt de punts i un conjunt d'arestes que uneixen alguns d'aquests punts. El nombre de vèrtexs d'un graf queda determinat pel nombre d'elements que hi ha en el conjunt V, per tant ens referirem a ell com a $|V|$ (cardinal de V). Amb les arestes passa el mateix, i també utilitzarem $|E|$ per indicar el nombre d'arestes d'un graf. Definim també que dos vèrtexs són *adjacents* si estan units per una aresta i, com a conseqüència, són *incidents* a l'aresta.
La idea intuitiva de graf convida a utilitzar representacions gràfiques. Així, en la figura 3 es mostra un graf simple G format per:

- $V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$
- $E = \{\{v_1, v_2\}, \{v_1, v_4\}, \{v_1, v_5\}, \{v_2, v_5\}, \{v_3, v_4\}, \{v_3, v_5\}, \{v_3, v_6\}, \{v_4, v_6\}\}$

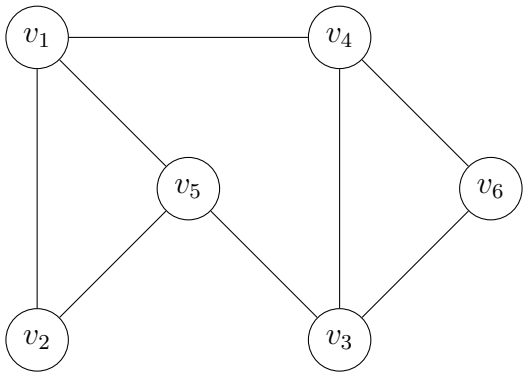


Figura 3: Representació gràfica del graf $G = (V, E)$ que s'ha presentat

Si una aresta comença i acaba en el mateix vèrtex (per exemple $e_m = \{v_i, v_i\}$ s'anomena llaç (figura 4, cas (a)). També pot ser que hi hagi dues arestes idèntiques, és a dir, dues arestes que uneixin v_i i v_j (figura 4, cas (b)). En qualsevol d'aquests dos casos anteriors, el graf s'anomena *multigraf* o *pseudograf*. En cas contrari, el graf serà anomenat *simple*. Amb el que hem vist fins ara, podem dir que $e_1 = \{v_1, v_2\}$ és equivalent a $e_2 = \{v_2, v_1\}$ (ja que es tracta de parells no ordenats). Tanmateix, existeixen grafs en els quals les arestes han de ser recorregudes en una direcció determinada. S'anomenen grafs *dirigits* i, en aquest cas, si $e_1 = (v_1, v_2)$ i

$e_2 = (v_2, v_1)$, $e_1 \neq e_2$, ja que es tracta de parells ordenats (figura 4, cas (c)). En la següent imatge es mostren els grafs esmentats anteriorment:

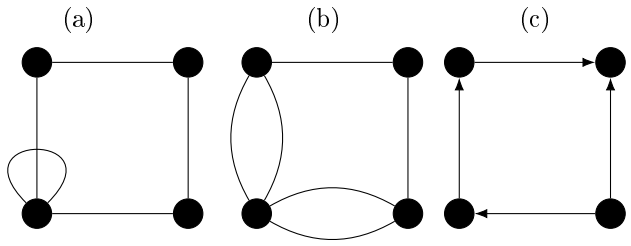


Figura 4: Graf amb un llaç (a), graf amb arestes múltiples (b) i graf dirigit (c)

Nota de l'autor: a partir d'ara, i si no s'indica el contrari, quan es parli de grafs, s'exclouran els multigrafs i grafs dirigits.

Direm que un graf no dirigit $G = (V, E)$ és *connex* si per a qualsevol $v_i, v_j \in V$ $v_i \neq v_j$ existeix un *camí* (successió d'arestes) que els uneix.
El nombre d'arestes que són incidents a un vèrtex v (comptant els llaços com a dues arestes) determinen el que anomenarem *grau* de v, que es representa amb $g(v)$. La successió de graus d'un graf serà la successió que s'obté a l'ordenar de manera creixent els graus dels seus vèrtex. Cal dir que no és un problema gens trivial saber si una successió de nombres naturals donada pot ser una successió de graus d'un graf. El grau mínim d'un graf $G = (V, E)$ queda determinat de la següent manera: $\delta(G) = min\{g(v) : v \in V(G)\}$. De manera similar, el grau màxim de G, $\Delta(G) = max\{g(v) : v \in V(G)\}$.
Si d'un graf connex en treiem una aresta o un node, en resulta un altre (o més d'un) graf connex. Si $v \in V(G)$, indicarem per $G - v$ al graf que s'obté al suprimir el vèrtex v i totes les seves arestes incidents. De la mateixa manera, si $e \in E(G)$, $G - e$ indicarà el graf que s'obté a l'eliminiar la aresta e.

Amb tots aquests conceptes ja podem veure el teorema d'Euler, un dels primers teoremes en teoria de grafs i un dels més importants.

Teorema 1 (Euler)

En tot graf $G = (V, E)$, la suma dels graus dels vèrtex és igual al doble del nombre d'arestes.

$$\sum_{v \in V(G)} g(v) = 2|E(G)|$$

Demostració: Només cal veure que cada aresta té dos extrems i que suma dos en el total dels graus. La demostració formal és més complicada, i comporta desenvolupar una inducció respecte del nombre d'arestes:

- Si $|E| = 0$, no cal considerar el cas. Si $|E| = 1$, o $|V| = 2$ i cada vèrtex té grau 1 o la aresta és un llaç i hi ha un sol vèrtex de grau 2. En qualsevol d'aquests dos casos, el teorema es verifica.
- Ara suposem que el teorema està demostrat per a $|E| \leq k$ i que G és un graf amb $|E| = k + 1$. Si e és una aresta de G prenem $H = G - e$.
- Llavors tots els vèrtexs de H tenen el mateix grau a H que a G excepte 2 que tenen un grau menys o un que té dos graus menys (només en el cas que e sigués un llaç). En tots dos casos obtenim que:

$$\sum_{v \in V(G)} g(v) = \sum_{v \in V(H)} g(v) + 2 = 2(|E| - 1) + 2 = 2|E|$$

D'aquesta demostració en treiem una altra afirmació:

En tot graf, el nombre de vèrtex amb grau imparell, és parell.

3 Tipus de grafs

Fins ara hem vist els conceptes bàsics en teoria de grafs i algunes de les propietats que compleixen tots els grafs. No obstant, existeixen diversos tipus de grafs que tenen propietats especials. A continuació se'n presenten alguns.

(Cal ordenar es apartats segons la ordenació topològica del graf de la figura 21)

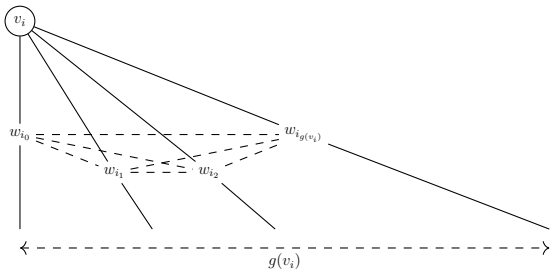
3.1 Graf lineal

Un graf lineal $L(G)$ d'un graf G és un graf que representa les adjacències entre les arestes de G. Formalment, donat un graf G, el graf lineal $L(G)$ és aquell en el qual cada vèrtex correspon a una aresta de G i dos vèrtexs són adjacents només si les arestes corresponents a G comparteixen un vèrtex.

Propietat 1 El graf lineal d'un graf amb n nodes, e arestes i amb vèrtexs de graus $g(v_i)$ té $n' = e$ nodes i

$$e' = \frac{1}{2} \sum_{i=1}^{n} g(v_i)^2 - e$$

arestes.
Demostració: Cada node v_i amb grau $g(v_i)$ del graf original generarà un graf complet de $g(v_i)$ nodes $(K_{g(v_i)})$ [2].



Un graf complet té $\binom{n}{k} = \frac{n(n-1)}{2}$, per tant en aquest cas se'n generen $\frac{g(v_i)(g(v_i)-1)}{2}$. Però això es compleix per a cada vèrtex, i llavors podem escriure

$$\sum_{i=1}^{n} \frac{1}{2} g(v_i)(g(v_i)-1) = \frac{1}{2}\sum_{i=1}^{n}(g(v_i)^2 - g(v_i)) = \frac{1}{2}\sum_{i=1}^{n} g(v_i)^2 - \underbrace{\frac{1}{2}\underbrace{\sum_{i=0}^{n} g(v_i)}_{2|E|}}_{|E|} = \frac{1}{2}\sum_{i=1}^{n} g(v_i)^2 - e$$

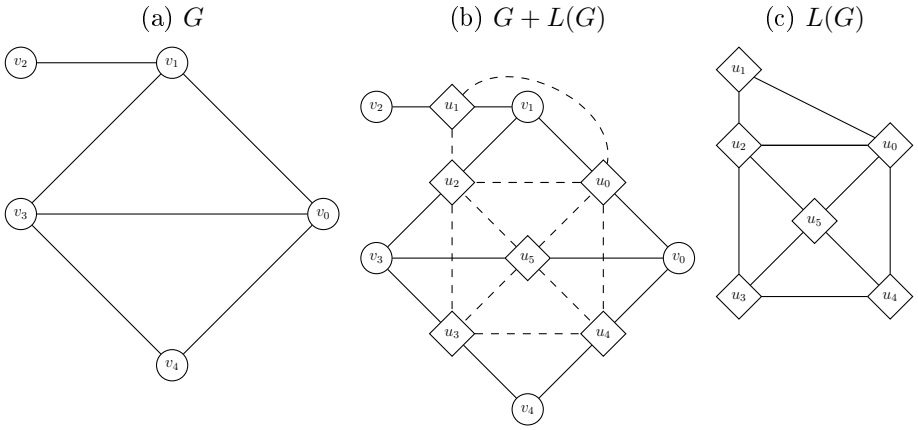


Figura 5: Procés de construcció d'un graf lineal

3.2 Cicles

Els *cicles* són grafs 2-regulars amb n vèrtexs i n arestes, i s'anomenen C_n.

[2] Veure apartat 3.7

Propietat 1 El graf lineal d'un cicle és ell mateix.

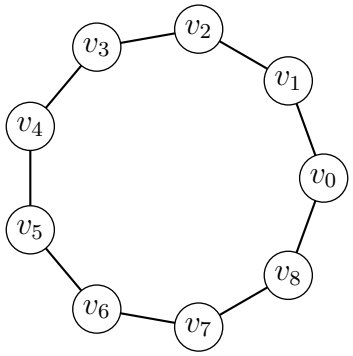


Figura 6: Cicle C_9

3.3 Grafs bipartits

Els *grafs bipartits* són aquells en els quals els vèrtexs es poden separar en dos conjunts disjunts U i W (és a dir, que els dos conjunts no tinguin elements comuns) de tal manera que un vèrtex d'un conjunt no sigui mai adjacent a un altre vèrtex del mateix conjunt. Es pot dir també que un graf és bipartit si i sols si no conté cicles de longitud senar.

Propietat 1 Tots els grafs C_n amb n parell, són també grafs bipartits.

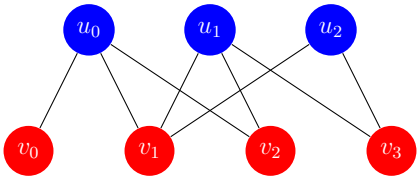


Figura 7: Graf bipartit

3.4 Grafs bipartits complets

Els *grafs bipartits complets* són grafs bipartits en els quals cada element del conjunt U està unit a tots els elements dels del conjunt W. S'anomenen $K_{m,n}$, on $m = |U|$ i $n = |W|$.

Propietat 1 En els grafs bipartits $K_{m,n} = K_{n,m}$, $|V| = n + m$ i $|E| = mn$.

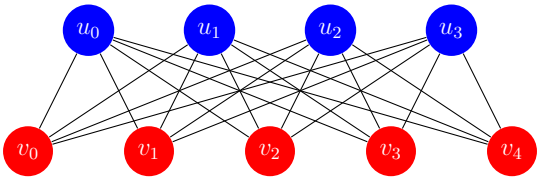


Figura 8: Graf bipartit complet $K_{4,5}$

3.5 Xarxes

Els *grafs xarxes* bidimensionsals $G_{m,n}$ són grafs bipartits que formen un entramaten forma de cuadrícula de mn vèrtexs.

Propietat 1 Es pot greneralitzar per a xarxes de més dimensions com a $G_{m,n,o,\dots}$.

Propietat 2 Un graf xarxa té mn vèrtexs i $(m-1)n+(n-1)m$ arestes.

Demostració: Tal com es pot veure a la figura **??**, un graf xarxa té

$$(n-2)(m-2)+2(m-2)+2(n-2)+4 = nm-\cancel{2m}-\cancel{2n}+\cancel{4}+\cancel{2m}-\cancel{4}+\cancel{2n}-\cancel{4}+\cancel{4} = mn$$

vèrtexs. D'altra banda, la suma dels graus de tots els vèrtexs és

$$\frac{4(m-2)(n-2)+3\times 2(m-2+n-2)+4\times 2}{2} = 2(m-2)(n-2)+3(m-2)+3(n-2)+4$$

Si es desenvolpua, s'obté

$$(n-2)(m-2)+\boxed{(n-2)(m-2)+2(m-2)}+(m-2)+\boxed{2(n-2)}+(n-2)+\boxed{4}$$

on la suma dels elements marcats equival a mn. Si es continua desenvolupant tinguent en compte aquest detall, s'obté

$$mn-2m-2n+4+m-2+n-2+mn = 2mn-m-n = \boxed{m(n-1)+n(m-1)}$$

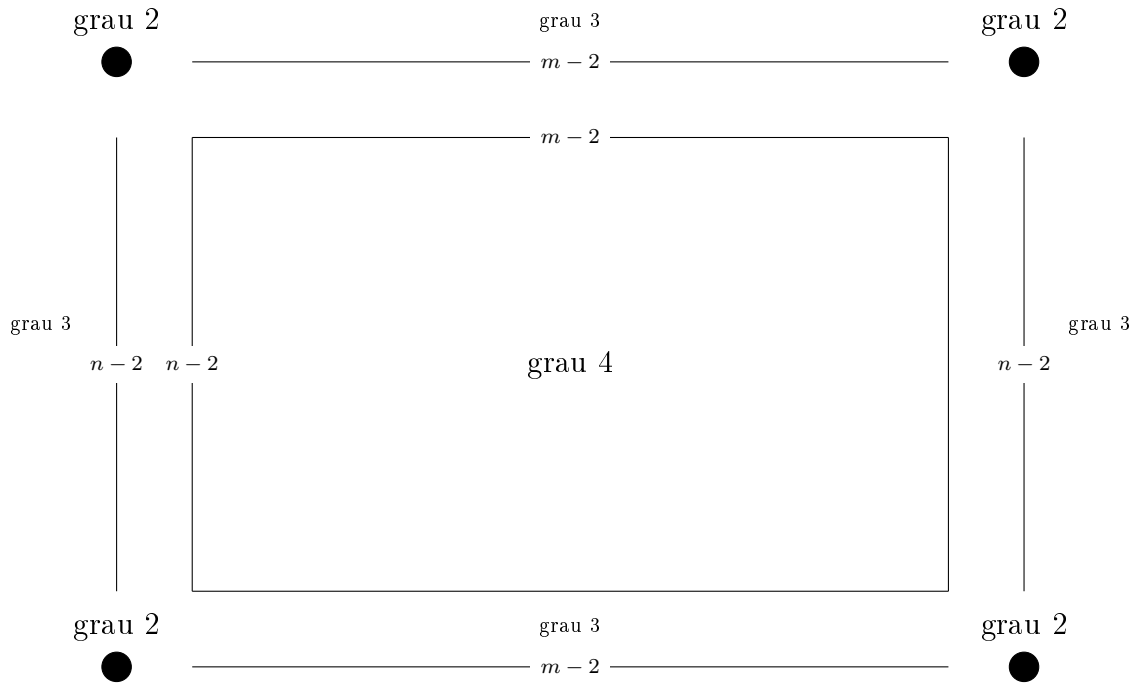


Figura 9: Representació dels graus dels diferents nodes d'un grau xarxa

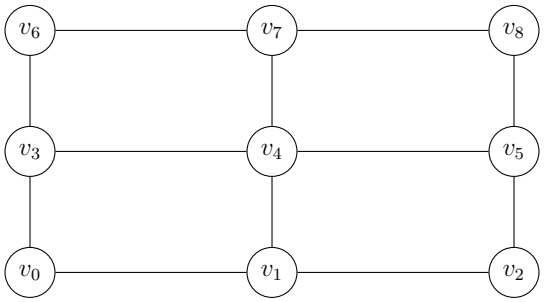


Figura 10: Graf xarxa $G_{3,3}$

3.6 Arbres

Els *arbres* són un tipus molt important de grafs: són grafs connexos sense cicles, de manera que existeix un únic camí entre dos vèrtexs.

Propietat 1 Si a un arbre se li afegeix una aresta, es genera un cicle, i se s'en treu una, el graf deixa de ser connex.

Propietat 2 Hi ha un tipus especial d'arbres anomenats *elementals o camins*, que són els arbres amb $|V| = 1$, $|V| = 2$ i en general tots aquells on $\delta = 1$ i $\Delta = 2$.

S'anomenen P_n, on $n = |V|$. També es pot pensar en grafs elementals com a $G_{n,1}$.

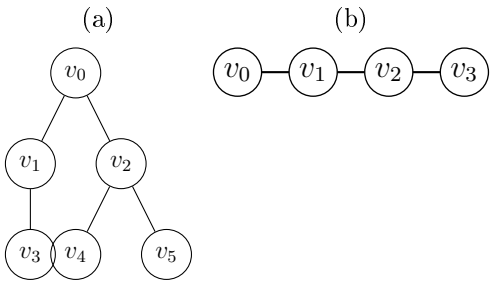


Figura 11: Arbre (a) i graf elemental P_4 (b)

Teorema 2

Un arbre amb n nodes té $n-1$ arestes.

Demostració: Si comprovem el cas on un arbre T té $n = 1$ vèrtexs, veiem que no té cap aresta, per tant el teorema es compleix. Si treiem una aresta de l'arbre,en sorgiran 2 arbres més T_1 i T_2. Per hipòtesi, T_1 tindrà $v_1 = |V(T_1)|$ vèrtexs i $n_1 - 1 = |E(T_1)|$ arestes, i T_2 tindrà $n_2 = |V(T_2)|$ vèrtexs i $n_2 - 1 = |E(T_2)|$ arestes. Llavors el nombre d'arestes de T és $(n_1-1)+(n_2-1)+1 = (n_1+n_2)-1$. Deduïm llavors que el nombre de nodes de T és $n_1 + n_2$. Dit d'una altra manera, $|E(T)| = |V(T)| - 1$.

Com a conseqüència d'aquest teorema, en podem arribar a un altre:

Teorema 3

En un arbre T amb $|V| \geq 2$, hi ha com a mínim dos vèrtexs de grau 1 (anomenats fulles).

Demostració: Fem un raonament per reducció a l'absurd. Suposem que no es compleix el teorema que volem demostrar, és a dir, que $g(v_i) > 1$ per a tots els vèrtexs. Llavors es compleix que

$$\sum_{v \in V} g(v) > \sum_{v \in V} 2 = 2|V|$$

Però això no és correcte, ja que, com a conseqüència del Teorema 1 i Teorema 2 podem dir que en un arbre

$$\sum_{v \in V} g(v) = 2|E| = 2|V| - 2$$

i estem dient que $2|V|-2 \geq 2|V|$. Amb això ja podem veure que necessitem treure com a mínim dos graus, però demostrem també que amb un sol node de grau 1 tampoc és suficient. Suposem ara que l'arbre té un sol node tal que $g(v) = 1$ i elsaltres tenencom a mínim grau 2. Llavors

$$2|V| - 2 = \sum_{v \in V} g(v) = 1 + \sum_{\substack{v \in V \\ \text{si } g(v) \neq 1}} g(v) \geq 1 + \sum_{\substack{v \in V \\ \text{si } g(v) \neq 1}} 2 = 1 + 2(|V| - 1) = 2|V| - 1$$

Ara estem dient que $2|V| - 2 \geq 2|V| - 1$, que torna a ser una contradicció. Sabem que si treiem un grau més, el teorema es complirà, i hem demostrat que el mínim nombre de vèrtexs de grau 1 és 2.

Hi ha una altra manera de demostrar-ho, més directa: Suposem que a l'arbre hi ha k vèrtex de grau 1, i $|V| - k$ de grau $g(v_i) \geq 2$. D'aquesta manera

$$2|V| - 2 = \sum_{v \in V} g(v) \geq k + 2(|V| - k) = 2|V| - k$$

Llavors podem dir que $-2 \geq -k$ i per tant $2 \leq k$.
Hi ha un altre teorema que relaciona el nombre de fulles amb el grau màxim d'un arbre:

Teorema 4

El nombre de fulles d'un arbre T és més gran o igual a Δ.

Demostració: Si eliminem un node de grau Δ de l'arbre, juntament amb totes les seves arestes incidents, obtenim un conjunt de Δ grafs disconnexos. Si alguns d'aquests grafs consisteixen en tan sols un node, vol dir que abans eren adjacents al node que hem eliminat, per tant, només tenien grau 1. Si, pel contrari, formen nous arbres, pel teorema 3 podem dir que hi haurà com a mínim dues fulles. Encara que hi ha la possibilitat que un dels nodes amb grau 1 sigués l'adjacent al node que hem tret, sempre podem garantir que hi ha com a mínim una fulla. Per tant, també podem garantir que que hi haurà com a mínim Δ fulles.

3.7 Grafs complets

Un *graf complet* és un graf on cada vèrtex està unit a tots els altres una sola vegada. Un graf complet amb n nodes és un graf simple i $(n-1)$-regular i la seva nomenclatura és K_n.

Propietat 1 Els grafs complets tenen $\binom{n}{2} = \frac{n(n-1)}{2}$ arestes.

Propietat 2 El nombre de cicles que conté un graf complet queda determinat per la següent igualtat:

$$C_n = \sum_{K=3}^{n} \frac{1}{2} \binom{n}{k} (k-1)!$$

On:

$\frac{1}{2}$ es multiplica perquè es conten dues vegades els cicles, ja que no són dirigits.

$\binom{n}{k}$ és el nombre de grups de k elements que es poden agafar.

$(k-1)!$ és el nombre de permutacions circulars de k elements.

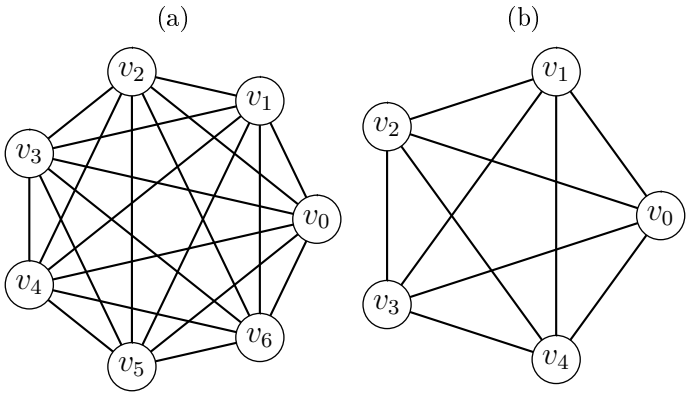


Figura 12: Grafs complets K_7(a) i K_5(b)

3.8 Rodes

Un *graf roda* amb n vèrtex ès un graf que conté un cicle de longitud $n-1$, on tots els vèrtex del cicle estan connectats a un vèrtex fora del cicle, anomenat node central. S'escriu W_n, i a vegades simplement s'estudia com a $C_{n-1} + K_1$.

Propietat 1 El node central té grau $n-1$, i la resta de nodes tenen grau 3.

Propietat 2 El nombre de cicles que conté un graf roda amb n vètrexs està determinat per $n^2 - 3n + 3$.
Demostració: El nombre de cicles que conté un graf roda és la suma del nombre de cicles de longitud des de 3 fins a n ($\sum_{i=3}^{n} C_i$). Amb l'exepció de C_{n-1}, el nombre de cicles per a cada i és $n-1$. En el cas de C_{n-1} n'hi ha n de possibles, però ho representem de la manera $n-1+1$, per tal que sigui més còmode operar. Com

que els cicles són possibles a partir de longitud 3, en total hi ha $n-2$ longituds possibles. D'aquesta manera podem escriure $(n-2)(n-1)+1 = \boxed{n^2-3n+3}$.

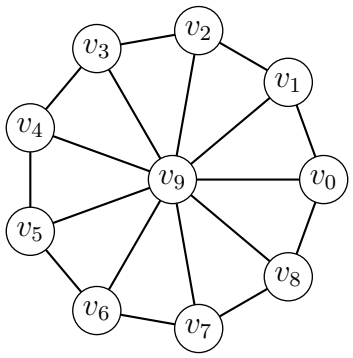


Figura 13: Roda W_{10}

3.9 Grafs estrella

Un *graf estrella* de grau n és aquell que conté un vèrtex amb grau $n-1$ i els $n-1$ vèrtexs restants de grau 1. La seva nomenclatura és S_n.

Propietat 1 Són estrelles tots els grafs bipartits de la forma $K_{1,n-1}$ o $K_{n-1,1}$.

Propietat 2 El graf lineal de S_n és K_{n-1}.

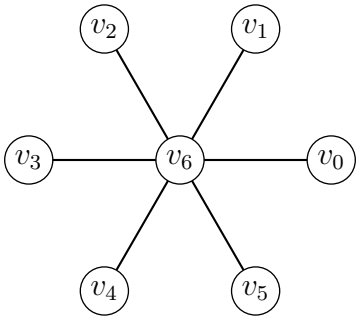


Figura 14: Graf estrella S_7

3.10 Grafs complementaris

El graf complementari del graf G és el graf H amb els mateixos vèrtexs que G, de manera que dos vèrtexs d'H seran adjacents si i només si a G no ho són. Formalment, si $G=(V,E)$ és un graf simple i $K=E(K_n)$ és el conjunt d'arestes derivades de totes les possibles combinacions de dos elements de V, essent $n=$

$|V(G)|$, llavors $H = (V, K \setminus E)$ [3]. Per indicar el complementari de G s'escriu $\bar{G}$ o G'.

Propietat 1 Per obtenir el graf complementari de G tan sols s'han de posar les arestes que falten per obtenir un graf complet i treure totes les que hi eren inicialment (ja que $|E(G)| + |E(\bar{G})| = |K|$). Veure figura 15.

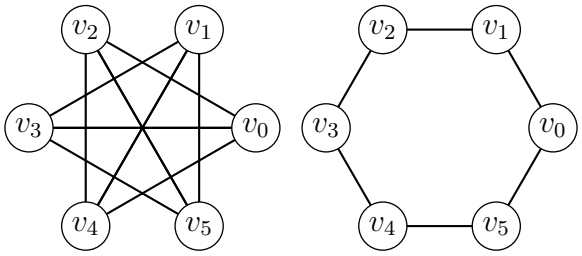


Figura 15: Un graf G i el seu complementari $\bar{G}$

3.11 Grafs regulars

Es diu que un graf $G = (V, E)$ és *regular de grau* r quan tots els seus vèrtexs tenen grau r. Formalment, un graf és r-regular quan $\Delta(G) = \delta(G) = r$. Un graf 0-regular és un graf nul [4], un graf 1-regular consisteix en arestes separades entre elles i un graf 2-regular consisteix en un o més cicles [5] separats. A partir d'aquí, els grafs regulars més importants tenen noms propis, com per exemple els 3-regulars, que són els *cúbics*; els 4-regulars, *quàrtics*; els 7-regulars, *grafs de Witt truncats dobles*; els 8-regulars, *grafs de 24 cel·les*...

A la imatge 16 es poden veure diversos exemples de grafs regulars.

Propietat 1 No necessàriament existeix un únic graf r-regular, sinó que sovint s'en poden fer amb diferent nombre de vèrtexs.

Propietat 2 Com a conseqüència del teorema 2, per a tots els grafs r-regulars amb n vèrtexs es compleix que

$$|E(G)| = \frac{1}{2}nr$$

on $|E(G)|$ és el nombre d'arestes.

[3] La notació $K \setminus E$ indica el conjunt dels elements de K que no són a E

[4] Veure apartat 3.12

[5] Veure apartat 3.2

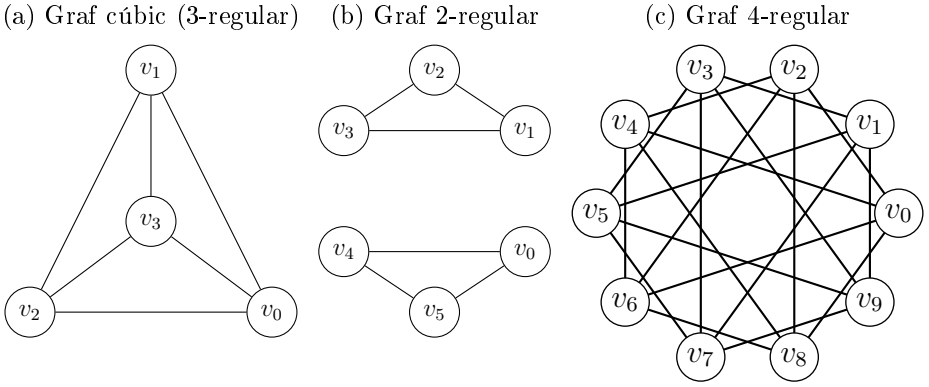


Figura 16: Exemples de grafs regulars

3.12 Graf nul i grafs buits

Els *grafs buits* són grafs sense arestes, és a dir, conjunts d'n vértexs. Són els complementaris dels grafs complets K_n [6], i per tant la seva nomenclatura és $\bar{K}_n$ o, simplement, N_n. Estrictament s'anomena *graf nul* a N_0 i buits a la resta, però com que normalment no s'utilitza N_0, convencionalment es diuen nuls tots els elements del conjunt dels buits.

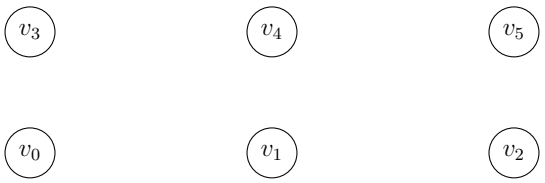


Figura 17: Graf buit N_6

Part II
Camins i algorismes

"Simplicity is a prerequisite for reliability"
Edsger W. Dijkstra

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

[6] Veure apartat 3.7

altre. Per aquest motiu utilitzem els camins, que trobarem o generarem mitjnçant diversos algorismes.

En aquesta secció es mostraran i s'implementaran diverses maneres de recòrrer un graf, torbant la manera més eficient per a cada cas.

4 Grafs ponderats i dirigits

4.1 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 $e_0 = (v_0, v_1) \neq e_1 = (v_1, v_0)$ [7]. 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.

5 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 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*.

[7] Observi's la diferència de notació per a les arestes dels grafs dirigits i no dirigits, atenent al fet que l'ordre dels vèrtexs en un cas importa i en l'altre no

En els grafs, ponderats, la *longitud* d'un camí $p = v_0 v_1 \cdots v_n$ 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 a l'agafar la menor longitud d'entre tots els camins elementals entre v i u. (adjuntar exemple de distància)

5.1 Camins Eulerians

Un camí Eulerià d'un graf és un camí que recorre totes les arestes d'un graf una sola vegada. De la mateixa manera, un cicle Eulerià consisteix en un camí que utilitza totes les arestes d'un graf una sola vegada i el punt inicial i final és el mateix. Un graf connex té un camí Eulerià si i només si té 0 o 2 vèrtexs de grau imparell, i té un cicle Eulerià si i només sí tots els vèrtex del graf tenen grau parell.

5.2 Camins Hamiltonians

Un camí Hamiltonià és un camí que passa una sola vegada per cadascún dels vèrtexs d'un graf. Un cicle Hamiltonià és un camí Hamiltonià en el qual el vèrtex inicial és igual al vèrtex final.

6 Estructures de dades dels grafs

En els grafs, normalement s'ha de tractar una gran quantitat d'informació: nodes, arestes, pesos, sentits... Tota aquesta informació no és difícil de gestionar si el graf que s'estudia és petit, ja que fins i tot es pot dibuixar. El problema sorgeix quan el graf en qüestió és més gran, com per exemple podria ser una xarxa de clavagueram o de metro. Llavors la informació a tractar és molta, i és convenient organitzar-la en estructures de dades. Les estructures de dades solen ser utilitzades en la programació, però en aquest cas sol ser beneficiós usar-ne fins i tot quan no es treballa amb informàtica.

Matrius

Matriu d'adjacència

La matriu d'adjacència d'un graf és una matriu quadrada que conté informació respecte el nombre d'arestes que uneixen qulsevol parella de nodes, i s'organitza

de la següent manera:

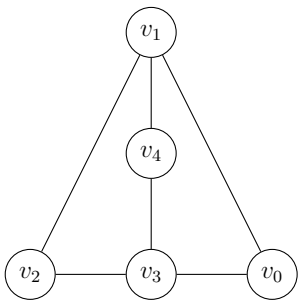
Si G és un graf amb $|V|$ nodes, $A(G) = (a_{ij})_{ij=1,\dots,|V|}$ és la seva matriu d'adjacència de $|V| \times |V|$, on a_{ij} correspon al nombre d'arestes que uneixen els nodes i i j, contant com a 2 els llaços, que sempre es trobaràn a la diagonal. La matriu serà simètrica si el graf ho ès, i podrem conèixer el grau d'un node i fent el sumatori de les caselles de la i-èsima fila. A vegades, quan s'utilitzen grafs ponderats, les matrius d'adjacència s'omplen amb els pesos de les arestes, per no haver d'utilitzar altres estructures per emmagatzemar la resta d'informació. En aquest tipus de grafs, els llaços no necessàriament valdran 2, però es podran diferenciar perquè estaràn a la diagonal. En aquest cas, el problema serà que no es podràn representar arestes múltiples amb pesos diferents. Tot i això, en termes de programació, utilitzar aquesta estructura de dades (sobretot en grafs grans) no és el més eficient, ja que la matriu creix molt ràpidament. La matriu d'un graf de $|V|$ nodes té $|V|^2$ espais, i en grafs poc densos, la majoria d'espais estan ocupats per 0, de tal manera que s'està utilitzant molta memòria que no conté informació útil. De la mateixa manera, si afegim un nou node al graf, que serà l'n-èssim, s'hauràn d'afegir $2n - 1$ espais a la matriu.

Matriu d'incidència

La matriu d'incidència d'un graf G sense llaços, $I(G) = (b_{i,j})_{i=1,\dots,|V(G)|,j=1,\dots,|E(G)|}$, és la matriu binaria de $|V(G)| \times |E(G)|$ on $b_{i,j}$ indica si la aresta j és incident al node i.

(afegir exemple de graf amb les seves dues matrius)

(a) Graf G



(b) A(G)

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

(c) I(G)

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}$$

Llistes d'adjacència

Aquesta estructura de dades és molt utilitzada per tractar grafs, ja que ocupa menys memòria i només inclou l'informació necessària. Les llistes d'adjacència d'un graf G consisteixen en un conjunt de $|V(G)|$ llistes. Cada llista correspon a

un node del graf, i conté els nodes al quals és adjacent. En informàtica s'acostumen a fer mitjançant apuntadors, de tal manera que, a partir de cada element de la llista, es pugui accedir a la seva pròpia llista, siguent així molt més senzill fer iteracions. Amb l'esquema següent es pot veure més clarament:

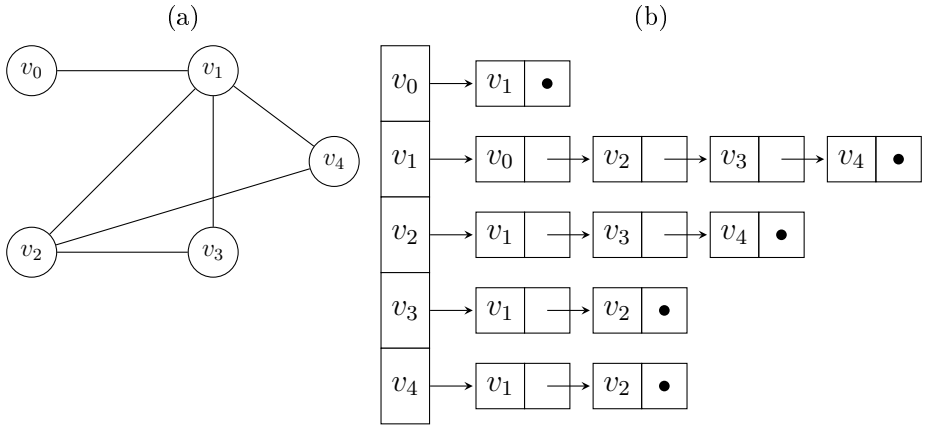


Figura 18: Un graf G (a) i la seva llista d'adjacència (b)

7 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.

Per automatitzar diversos càlculs sobre grafs, molts científics han treballat desenvolupant processos universals que poden ser duts a terme per ordinadors. Aquests processos, anomenats *algorismes*, consisteixen en conjunts específics d'ordres que s'executen sobre unes estructures de dades que contenen el graf.

Per fer l'anàlisi dels algorismes, s'haurà de fer l'anàlisi de la cota superior asimptòtica, és a dir, analitzar el temps d'execució de l'algorisme quan l'entrada és la més gran i complexa possible. Per a aquest propòsit s'utiltizarà la notació de Landau, també anomenada notació de la gran "O". Es temps majoritariament es calcularan en funció del nombre d'atestes o de nodes. Cal tenir present que tots els logarismes que apareguin en funcions de Landau són logaritmes en base 2, sempre que no s'indiqui el contrari.

A continuació se'n mostren uns quants d'importants per fer operacions sobre grafs.

7.1 BFS

Aquest algorisme serveix per examinar l'estructura d'un graf o fer-ne un recorregut sistemàtic. La recerca per amplada prioritària (*breadht-first serch* en anglès, d'aquí **BFS**) fa l'exploració en paral·lel de totes les alternatives posibles per nivells des del vèrtex inicial. A la imatge 19 es pot veure com funcionaria aquest algorisme en un graf. 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. El que farà el programa serà imprimir per pantalla la seqüència de vèrtexs ordenada segons l'ordre en que els ha visitat.

7.1.1 Funcionament

Donat un node incial v, es busquen tots els nodes no visitats que estiguin a distància i de v. Inicialment $i = 0$ i l'únic a distància 0 de v és ell mateix. Després, $i = 1$ i per tant, en aquesta iteració es visitaran tots els nodes adjacents a v. Quan $i = 2$ es visitaran els nodes adjacents als adjacent de v que no s'hagin visitat, i així consecutivament fins a visitar tots els nodes.

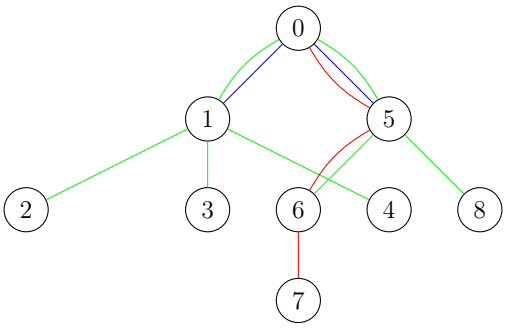


Figura 19: Exemple del recorregut que realitzaria l'algorisme

7.1.2 Pseudocodi

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
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 l'algorisme BFS, i encara que és eficient, s'està desaprofitant propietats de l'algorisme. Amb l'algorisme 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' l'algorisme. 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 una clau i un valor), però en llenguatges basats en C, es poden utilitzar maps de la mateixa manera.

7.1.3 Aplicacions

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:

- Buscadors com Google l'utilitzen per indexar pàgines web noves. Amb l'al-

```
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)
```

gorisme 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.

- Les xarxes socials l'utilitzen per suggerir amistats. Amb aquest algorisme es poden trobar els amics d'una persona (els nodes que estan 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 sol·lucionar un cub de rubik. 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 es pot arribar a l'estat resolt amb els mínims moviments possibles.

7.2 DFS

La recerca per profunditat prioritària (*depth-first search* en anglès, d'aquí **DFS**) és un algorime que utilitza uns proncipis semblants als de l'algorisme 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.

7.2.1 Funcionament

L'algorisme DFS arriba el mes lluny possible abans de retrocedir. Això significa que quan arriba a un node se'n va a un d'adjacent no visitat i va emmagatzemant l'ordre amb el qual els visita. Quan un node no té nodes adjacents no visitats, torna al node anterior i continua a partir d'allà. A la figura 19 es mostra un graf amb els nodes numerats segons l'ordre en el qual es visiten i amb el recorregut que faria l'algorisme:

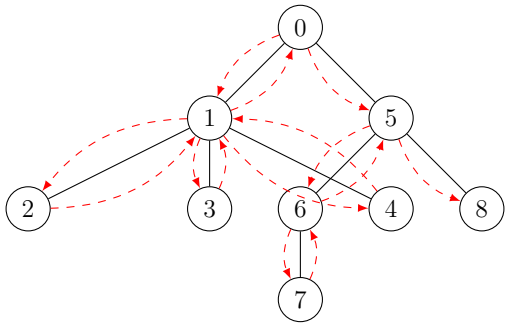


Figura 20: Exemple del recorregut que realitzaria l'algorisme

7.2.2 Pseudocodi

Tal com en el BFS, també hi ha diverses maneres d'implementar l'algorisme, i 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 d'aquest.

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

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 [8].

[8] S'anomena funció recursiva a aquella que es crida a si mateixa

```
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
```

7.2.3 Propietats

Propietat 1 En grafs dirigits es passa una vegada per cada aresta, mentre que en grafs no dirigits es passa dues vegades (una des de cada vèrtex).

Propietat 2 El segon algorisme, implementat en Python amb llistes d'adjacència, té un temps d'execució de $O(|V| + |E|)$, i per tant, un templs lineal.
Demostració: El bucle principal s'executa $|V|$ vegades (d'aquí $O(|V|)$). D'aquesta manera, s'executa la funció recursiva una vegada per vèrtex, i dins d'aquest es miren tots els nodes adjacents. El temps total que triguen totes les funcions recursives és de $O(\sum_{v \in V} |adjacents[v]|)$ que equival a $O(|E|)$.

Propietat 3 A través de l'algorisme es podrien classificar les arestes en 4 tipus

diferents:

- Arestes de l'arbre: Són les arestes que pertanyen a l'arbre expansiu del graf.
- Arestes cap endavant: Són les que van d'un node cap a un dels seus descendents de l'arbre.
- Arestes cap enrere: Són les que van d'un node cap a un dels seus antecessors de l'arbre.
- Arestes creuades: Són les que no pertanyen a cap de les categories anteriors.

En el cas de grafs no dirigits, només hi ha arestes de l'arbre i arestes cap enrere.

7.2.4 Aplicacions

Aquest algorisme no té tantes utilitats pràctiques com l'algorisme BFS, però també té propietats útils, 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 node (encara no n'hem explorat tots els nodes adjacents) el trobem a ell mateix. De la mateixa manera, es pot dir que un graf conté un cicle si i només si conté una aresta cap enrere.
- 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 l'algorisme DFS, podem obtenir una de les seqüències vàlides per completar totes les tasques. A continuació es presenta un exemple d'ordenació topològica utilitzat dirant la creació d'aquest treball:

  Resulta que, a la secció 3, quan explicava un tipus concret de graf, sovint utilitzava altres tipus de graf. Per determinar en quin ordre havien d'estar els apartats, de tal manera que quan apareixès un tipus de graf (a una altra definició) aquest ja s'hagués explicat, es va utilitzar una ordenació topològica. Es va construïr un graf dirigit representant les dependències de cada apartat (veure figura 21), i es va executar l'algorisme d'ordenació topològica.

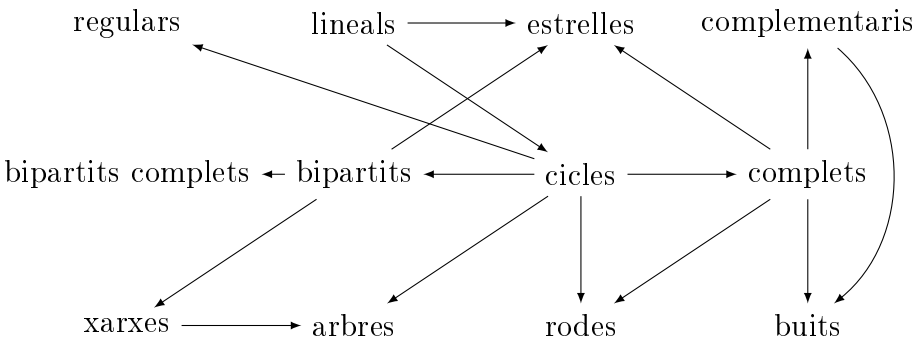


Figura 21: Graf de les dependències de cada apartat

7.3 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 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.

7.3.1 Funcionament

L'algorisme de Dijkstra es basa en dos principis:

- La desigualtat triangular: La desigualtat triangular com a tal diu que per a qualsevol triangle, la suma de dos dels seus costats serà igual o superior que al costat restant. Si a, b i c són les longituds dels costats d'unt triangle i c és la més gran, llavors $c \leq a + b$. Aplicant la desigualtat a un graf ponderat (sense pesos nogatius) s'obté que per tots els vèrtex $v, u, x \in V$, $d(v, u) \leq d(v, x) + d(x, u)$ (cal recordar que la distància entre dos vèrtexs ve donada pel camí més curt entre aquests vèrtexs).

- La subestructura òptima: En porgramació dinàmica, el concepte de subestructura òptima significa que es poden fer servir les solucions òptimes de problemes més petits per solucionar de manera òptima un problema més gran. En aquest cas en particular diem que un subcamí d'un camí mínim és també un camí mínim.

Donat un node inicial s, l'algorisme suposa que la distància mínima per arribar als seus nodes adjacents és simplement el pes de l'arsta que els uneix i per la resta de nodes és ∞. Aquesta serà una distància provisional, ja que posteriorment

potser es trobarà un camí més curt. No obstant, hi ha una única distància que es pot determinar de manera definitiva: la del node v que està unit a s amb l'aresta de menys pes. Si existís un camí més curt entre s i v, aquest per força hauria de passar per a algun node x adjacent a s, però per la desigualtat triangular $d(s,v) \leq d(s,x) + d(x,v)$.

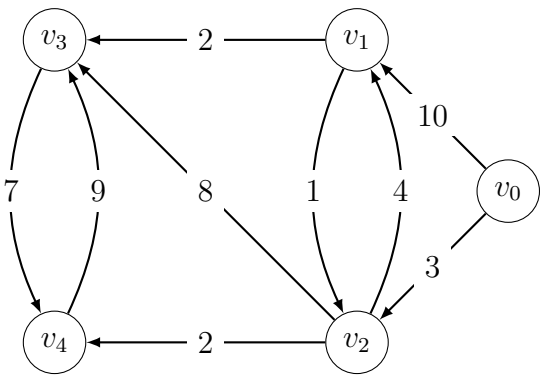
Això també es pot veure d'una manera més intuituva: si v està unit a s amb l'aresta de menys pes incident a s, per força l'aresta que uneix s i x ja té un pes superior a la primera, i per tant el camí no podrà ser més curt.

Un cop fet aquest pas s'obtenen les distàncies provisionals des del node s fins a tots els seus adjacents i el camí més curt entre dos nodes que, com a conseqüència de la subestructura òptima, forma part de l'arbre expansiu mínim.

A partir del node v, es busquen els seus adjacents i es calcula la distància a la qual estan respecte s (és a dir, la suma del camí acumulat i el pes de l'aresta). Si la nova distància calculada és menor a la provisional (cosa que passarà quan es trobi un camí més curt), la provisional passarà a ser la calculada. Un cop s'hagi fet tots els canvis necessaris, es buscarà el node no visitat que estigui a menor distància de s (i que per tant formarà part de l'arbre expansiu mínim), que passarà a ser v i es tornarà a fer el mateix procediment. Això s'anirà repetint fins que s'hagi visitat tots els vèrtexs.

En la figura 22 es mostra un graf i la taula amb les distàncies calculades per a cada iteració del programa. En aquest cas el node inicial és v_0. A cada iteració s'agafa la distància més petita (marcada amb un quadre) i es calcularà la distància provisional dels seus nodes, sumant la distància acomulada fins al node més el pes de l'aresta.

(a)



(b)

Distàncies provisionals

Iteració	v_0	v_1	v_2	v_3	v_4
1	0	10	[3]	∞	∞
2	|	7	|	11	[5]
3	|	[7]	|	11	|
4	|	|	|	[9]	|

Figura 22: Graf G (figura a) i taula amb les distàncies provisionals per als nodes de G en cada iteració (figura b)

7.3.2 Pseudocodi

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 = minvalordeQ
    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)
```

7.3.3 Propietats

Propietat 1 L'algorisme de Dijkstra és el que s'anomena un *algorisme voraç*. Això significa que sempre agafa la millor opció en aquell moment, suposant que les opcions que es trobin posteriorment no la podràn millorar. Aquest és el motiu per al qual l'algorisme de Dijkstra no funciona amb pesos negatius. En un graf amb pesos negatius això no necessàriament es complirà.

Propietat 2 L'algorisme de Dijkstra no troba únicament el camí mínim entre un node i un altre perquè la millor manera de fer-ho que es coneix és precisament trobar els camins mínims des d'un node fins a tota la resta.

Propietat 3 L'algorisme té un temps d'execució de $O(|E| + |V|log(|V|))$.

Demostració: Les operacions que es fan són les següents:

- S'inicialitzen la cua i les distàncies per a cada node, operació que triga $O(|V|)$.
- Es busca el valor mínim de la cua que, en cas d'estar optimitzada, triga $O(log|V|)$. En cas de no estar optimitzada, s'hauia de recòrrer tota la cua i anar comparant tots els elements, cosa que suposaria un temps de $O(|V|)$.
- Es calcula la ditància provisional per a cada node, que en principi tan sols ha de trigar $O(1)$.

La iteració principal acaba quan no queden nodes a la pila. Inicialment tots els nodes són a la pila i cada node es treu només una vegada. Per tant, per a cada node es buscarà una vegada el valor mínim de la pila. Això és un total de $O(|V|log(|V|))$ amb la cua optimitzada i $O(|V|^2)$ sense. El procediment per calcular la nova distància s'executarà, com a màxim, una vegada per aresta, i com que el temps de cada operació és de $O(1)$, el total és de $O(|E|)$.

D'aquesta manera el temps d'execució total és de $O(|E|+|V|log(|V|))$ per cues optimitzades i de $O(|E|+|V|^2)$ per a cues normals.

7.3.4 Aplicacions

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 destinacions.
- Problemes de canvis de divisa, on volem trobar la millor manera de canviar divises successives 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. En aquest context prenen sentit les arestes amb pesos negatius, encara que l'algorisme de Dijkstra no els pot tractar.
- 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 posició i representa 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 aquest algorisme 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.

7.4 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 l'algorisme de Dijkstra no ho permet.L'algorisme de Dijkstra es basa en la desigualtat triangular per tobar el camí més curt, però amb pesos negatius no es pot suposar que la desigualtat triangular es compleixi. A més, permet saber si un graf conté cicles negatius. Si un camí entre dos nodes conté un cicle 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 $-\infty$.

7.4.1 Funcionament

L'algorisme de Bellman-Ford utilitza una part de l'algorisme de Dijkstra, concretament la part on comprova si la distància suposada és major a la distància calculada en aquell moment. La diferència amb l'algorisme de Dijkstra és que, en lloc de sel·leccionar el node amb menor distància i fer l'operació sobre totes lse seves arestes incidents, simplement fa l'operació sobre totes les arestes del graf. Aquesta operació sobre les arestes es fa un total de $|V| - 1$ vegades, i a cada iteració el nombre de nodes amb distàncies mínimes augmenta, fins que tots els nodes tenen la distància correcta.

7.4.2 Pseudocodi

Algorisme 4: Bellman-Ford

```
Data: Un graf ponderat G i un node inicial 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);
```

7.4.3 Propietats

Propietat 1 L'algorisme de Bellman-Ford pot treballar amb pesos negatius ja que no es basa en la desigualtat triangular.

En el graf de la figura 23, l'algorisme de Dijkstra calcularia malament el camí mínim entre s i v: la distància provisional seria de 3 per arribar a v i de 4 per arribar a x. Llavors, com que 3 és la distància més petita que ha trobat (i suposa

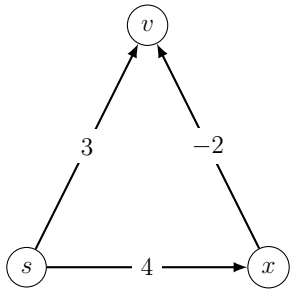


Figura 23: Graf on la desigualtat triangular no es compleix

que és la correcta), visita v. Però v no té nodes adjacents, per tant acaba l'iteració del node v. El següent node no visitat amb mínim valor és x. Encara que v sigui adjacent a x, com que v ja ha estat visitat i per la desigualtat trianglar s'assegurava que era el mínim, no el pot canviar.

En canvi, si s'executés l'algorisme de Bellman-Ford en aquest graf, el procediment que faria seria el següent: tal com l'algorisme de Dijkstra, posaria una distància provisional de 3 per arribar a v i de 4 per arribar a x. Ara visitaria v, però com que no té nodes adjacents, no hi faria res. A continuació visitaria x, que té v com a node adjacent. Com que la distància calculara des de x és de $4-2=2$ i $2<3$, simplement canviaria el valor de la distància entre s i v.

Propietat 2 Es pot aconseguir reduïr el temps d'execució amb una implementació diferent de l'algorisme, que funciona per a grafs dirigits sense ccles. En aquesta nova implementació es fa una ordenació topològica del graf i es visiten de manera ordenada tots els nodes. A cada node es comprova si el pes suposat per arribar a cadascun dels seus adjacentses pot millorar. Aquesta implementació permtet reduïr el temps d'execució fins a $O(|V|+|E|)$.

Propietat 3 L'algorisme de Bellman-Ford té un temps d'execució de $O(|E|\times|V|)$. Es fan $|V|-1$ iteracions, i a cada iteració es comproven totes les arestes. A tot això cal afegir-hi el temps de l'inicialització, que és de $O(|V|)$. Per simplificar-ho, es sol dir que té un temps d'execució de $O(|E|\times|V|)$, un temps que sempre serà lleugerament superior al temps real.

7.5 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*), determinar 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.

7.5.1 Funcionament

L'algorisme es basa en les següents propietats:

- Si un graf ponderat conté un cicle C i e és l'aresta de més pes de C, e no forma part de l'arbre expansiu mínim del graf. *Demostració*: Suposem que e pertany a l'arbre expansiu mínim. Llavors, si s'elimina e, l'arbre queda trencat en dos arbres, cadascun dels quals conté un extrem de e. Però la resta del cicle fa que els dos arbres siguin connexos, i en particular hi haurà una aresta f que tindrà un extrem a cadascún del arbres i que tindrà un pes inferior que e. Concretament f formarà part de l'arbre expansiu mínim.

- Si en un graf $G = (V, E)$ es fa un *tall*, es paretixen els vèrtex del graf en dos conjunts disjunts $(S, V(G) - S)$. D'enrte les arestes que tenen un extrem a S i l'altre a $V(G) - S$, e és la de menor pes i forma part de l'arbre expansiu mínim. *Demostració*: Suposem que e no forma part de l'arbre expansiu mínim. Llavors, si s'afegeix e a aquest arbre expansiu, es genera un cicle C. Això comporta que hi ha una altra aresta f amb un un extrem a S i l'altre a $V(G) - S$. D'aquesta manera, si es canvia f per e es genera també un arbre expansiu que és mínim, ja que $w(e) < w(f)$.

El que fa l'algorisme és anar construïnt arbres expansius separats per acabar unintlos si el graf és connex. Per aconseguir-ho, l'algorisme crea un conjunt per a cada node del graf (representant cadascun dels arbres), ordena les arestes amb ordre creixent segons el seu pes i les visita en aquest ordre. Quan es visita una aresta, s'uneixen els conjunts que les contenien (és a dir, es connecten els arbres). Si una aresta uneix nodes que formen part del mateix conjunt (és a dir, es crea un cicle), vol dir que l'aresta en concret no forma part de l'arbre expansiu del graf, tal com estableix la primera propietat.

En la figura 24 es mostra el procés que duu a terme l'algorisme de Kruskal. En aquesta figura cada color representa un conjunt diferent i a cada pas s'ajunten diversos grup dins d'un més gran, tinguent en compte quines arestes els uneuxen. Al final tan sols hi ha un únic grup i totes le arestes marcades formen part de l'arbre expansiu mínim.

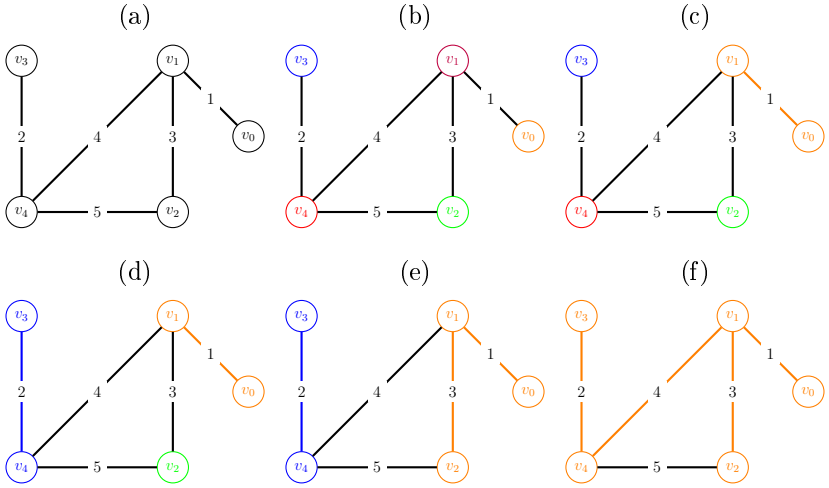


Figura 24: Passos corresponents a les diverses unions de grups que fa l'algorisme

7.5.2 Pseudocodi

Data: Un graf G i un node inicial s
Result: Arbre expansiu mínim de G
nova estructura union-find *subgraf*
nova llista *arbre*
ordena $E(G)$ 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*)

7.5.3 Propietats

Propietat 1 Com que l'algorisme construeix l'arbre expansiu mínim a partir de diversos components inicialment disconnexos, encara que el graf no sigui connex funcionarà correctament. En particular, trobarà l'arbre expansiu mínim de cadascun dels conjunts de nodes connexos.

Propietat 2 El temps d'execució aproximat de l'algorisme de Kruskal és de $O(|E| \times log(|V|))$. L'operació d'ordenar les arestes triga $O(|E| \times log(|E|))$ i el total de les operacions find i union és de $O(|E| \times log(|V|))$. La suma total és de $O(|E| \times log(|V|) + |E| \times log(|E|))$. Tot i això, com que $|E|$ pot ser com a màxim $|V|^2$ podem escriure que $log(|E|) = log(|V|)$ ($log(|E|) = log(|V|^2) = 2 \times log(|V|)$, però en la notació de Landau $2 \times log(|V|)$ és essencialment $log(|V|)$). D'aquesta manera el total és de $O(2 \times (|E| \times log(|V|))) = O(|E| \times log(|V|))$.

7.6 Prim

L'algorisme de Prim, tal com el 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 les Union-Find. Com a conseqüència, l'algorisme de Prim és més lent en grafs petits, però més ràpid en grafs molt densos.

7.6.1 Funcionament

L'algorisme de Prim es basa en la mateixa propietat dels talls que l'algorisme de Kruskal. L'algorisme de Prim divideix els nodes en dos grups S i $V(G) - S$, 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 (ja que per la propietat de tall sabem que formarà part de l'arbre expansiu mínim), i afegeix el node final de l'aresta al primer grup (el que seria euivalent a la funció union de l'algorisme de Kruskal). D'aquesta manera obté l'arbre expansiu mínim del graf. Aquest algorisme funciona per grafs connexos, però executant-lo per a cada node del graf, trobaria el *bosc* (conjunt d'arbres) mínim d'un graf no connex.

A la figura 25 es pot veure un exemple del funcionament de l'algorisme de Prim sobre un graf poderat. Els dos conjunts es representen mitjançant els colors taronja i negre, i les arestes que formin part de l'arbre expansiu mínim seràn les que estiguin pintades.

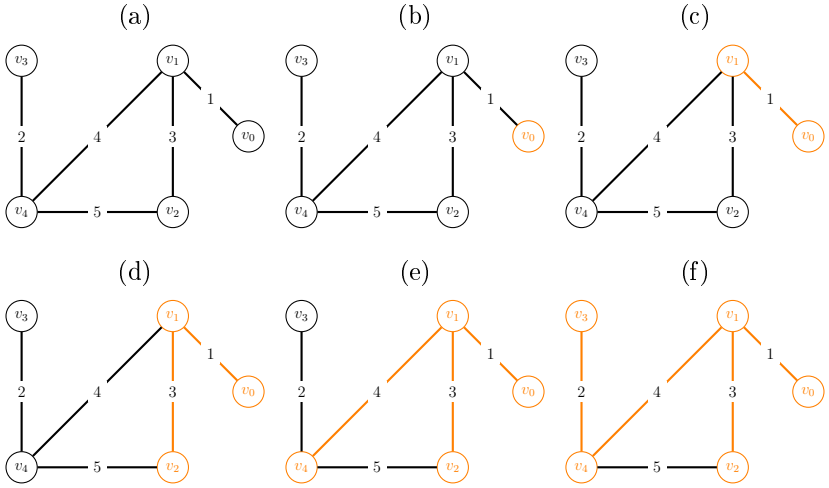


Figura 25: Passos corresponents a les fases d'expansió de l'arbre expansiu mínim

7.6.2 Pseudocodi

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**
  $Q[v] = \infty$
end
$Q[0] = 0$
while Q *no estigui buit* **do**
  $u = min\{\text{valor de } Q\}$
  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$(anterior)$

7.6.3 Propietats

Propietat 1 En l'algorisme de Prim tan sols es necessiten dos conjunts, un que contindrà els nodes de l'arbre expansiu mínim i l'altre els que encara no en formen part. D'aquesta manera l'arbre creix en només un component, mentre que a l'algorisme de Kruskal l'arbre es forma enmés components. Això fa que en grafs molt densos l'algorisme de Prim sigui més ràpid, però, com a contrapartida, no pot tractar amb grafs disconnexos.

Propietat 2 El temps d'execució aproximat d'aquesta implementació de l'algorisme de Prim és de $O(|E| + |V| \times log(|V|))$, però hi ha implementacions que aconseguiexen temps de $O(|V|^2)$ o $O(|V| \times log(|V|))$. En aquest cas es busca el mínim valor de Q un total de $|V|$ vegades, i es fan $|E|$ comparacions de cost 1.

7.6.4 Aplicacions

Tant l'algorisme de Kruskal com el 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:

- El disseny de 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 la més curta possible.
- Els arbres expansius mínims es poden utilitzar per generar laberints.
- S'utilitzen com a subrutines (o funcions) d'algorismes més complexos.

7.7 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 $|V| \times |V|$ on $D_{ij} = w(i, j)$.

7.7.1 Funcionament

Per funcionar, l'algorisme de Floyd-Warshall manté una matriu $D^{(x)}$ de mida $|V| \times |V|$, on $D^{(x)}_{vu}$ és la logitid del camí mínim entre v i u que està passa com a màxim

per x nodes. A cada iteració principal s'afegeix un node al conjunt de nodes que poden ser utilitzats, i es recalculen els valors de la matriu de la següent manera:

$$D_{vu}^{(x)} = min(D_{vu}^{(x-1)}, D_{vk}^{(x-1)} + D_{ku}^{(x-1)})$$

on $D^{(x)}$ és la matriu on hi ha les distàncies amb camins de com a màxim x nodes intermitjos.

7.7.2 Pseudocodi

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
```

7.7.3 Propietats

Propietat 1 Assumint que l'entrada de l'algorisme és $D^{(0)}$, el temps d'execució de l'algorisme és de $O(|V|^3)$. Tan sols cal veure que hi ha tres bucles un dins l'altre, i cadascun fa $|V|$ iteracions.

7.7.4 Aplicacions

Aquest algorisme es pot utilitzar per a qualsevol de les aplicacions en que s'utilitzaria Dijkstra amb 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 $D_{ii} < 0$, 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 es pot accedir a el node concret.

7.8 Coloració de grafs

La coloració de grafs és un problema que consisteix en "pintar"els nodes d'un graf de manera que dos nodes adjacents no tinguin el mateix color. El primer problema plantejat en aquesta branca ser plantejat per Francis Guthrie [9] i consistia en veure quans colors eren necessaris per pintar qualsevol mapa de manera que es complissin aquests mateixos requisits.

La majoria d'algorismes que realitzen aquest procediment són lents i relativament poc eficients. Per aquest motiu vaig decidir no utilitzar els que vaig trobar i dissenyar-ne un jo mateix.

7.8.1 Funcionament

El que fan la majoria d'algorismes és posar un color a un node i assignar-ne de diferents als seus adjacents, però aquest procediment no sempre troba una distribució eficient, ja que escull arbitràriament entre els diversos colors al assignar-los a un node. A més, el resultat que dóna l'alorisme depèn del node en el qual es comenci a executar. Amb l'algorisme que he dissenyat s'utilitzen estrictament els colors necessàris, establint un ordre específic en els grafs, de manera que hi ha una única distribució possible per a cada graf.

[9] Veure apartat 1.2

Per establir aquest ordre específic s'ordenen els nodes en funció del nombre de nodes adjacents. D'aquesta manera es comença assignant un color als nodes amb més grau, i per tant més complicats de pintar en cas que ja hi hagi nodes adjacents pintats. Un cop els nodes estan ordenats, s'agafa un color c (representat per un enter que inicialment és 0) i es pinta el node amb grau més alt. A continuació i per nombre decreixent de grau, es comprova per a tots els nodes del graf si són adjacents a un altre pintat amb color c. En cas de ser-ho, es pintaran també amb c i la resta es deixaràn igual. Un cop s'haigi recorregut tots els nodes, s'establirà un altre color i es repetirà tot el procediment fins que tots els nodes tinguin un color assignat.

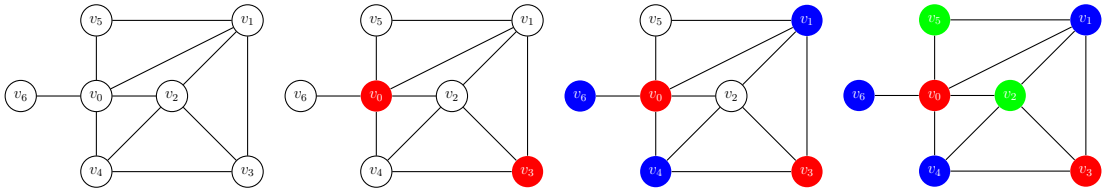


Figura 26: Procés de coloració que segueix l'algorisme

7.8.2 Pseudocodi

Algorisme 7: Coloració

```
Data: un graf G
Result: conjunt de nodes de G amb el color corresponent
nova llista nodes
nova llista colors
nou enter color_actual
nou booleà usat
usat = False
actual = 0
for i in range(0, |V|) do
    colors[i] = Nul
end
nodes =vèrtexs de G ordenats de manera decreixent segons el seu grau
color[nodes[0]] = 0
while no tots els nodes tinguin assignat un color do
    foreach node v de G do
        if colors[v] == Nul then
            foreach node u adjacent a v do
                if colors[u] == color_actual then
                    usat = True
                    break
                end
            end
            if usat == False then
                colors[v] = color_actual
            end
            usat = False
        end
    end
    color_actual = color_actual + 1
end
```

7.8.3 Propietats

Propietat 1 En el pitjor dels casos, l'algorisme triga $O(|V|^3)$. *Demostració*: El pitjor cas que pot rebre aquest algorisme és un graf complet K_n [10]. En aquest cas,

[10] Cal observar que en un graf K_n tots els nodes estan connectats amb la resta, i que per tant es necessiten n colors

caldrà executar $n = |V|$ vegades el bucle principal, on cada vegada recorrerà tots els nodes del graf i els seus adjacents. D'aquesta manera es recorrerà 3 vegades el graf sencer, trgant un temps total de $O(|V|^3)$.

Propietat 2 Si el que es vol és pintar les arestes d'un graf en lloc dels vèrtexs, només cal executar l'algorisme sobre el graf lineal d'aquest i un cop pintat tornar a muntar el graf original.

7.8.4 Aplicacions

- Per la manera com està fet l'algorisme, es pot comprovar si un graf és bipartit. Tota execució de l'algorisme en un graf bipartit utilitzarà tan sols dos colors.
- S'utilitza la coloració de grafs per a problemes que requereixen separar els nodes en grups separats de manera que dos nodes del mateix grup no siguin adjacents.

Part III
Disseny de grafs

Fins ara, s'han mostrat i estudiat grafs que ja estan definits sobre els quals s'ha de fer alguna operació. Ara bé, en aplicacions reals de teoria de grafs, sovint no hi ha un graf determinat, sinó que s'ha de generar.

8 Punt de Fermat i l'arbre de Steiner

Normalment, quan s'utilitzen els grafs com a mdels matemàtics de situacions reals, els nodes ja estaran determinats i el problema consistirà a trobar les arestes. Si això és així, segurament es podràn afegir altres nodes. Un exemple d'aquest cas consistiria a haver d'unir tres ciutats amb carreteres de tal manera que des d'una es pugui arribar directament a les altres dues. La primera sol·lució en que sol pensar una persona és fer un triangle en el qual les ciutats siguin els vèrtexs (l'equivalent a un graf K_3). Aquesta sol·lució és òptima si es vol anar d'una ciutat a l'altra amb la mínima distància possible, però si es vol construïr el mínim de carreteres possibles per a la xarxa conjunta, aquesta sol·lució no és la millor. En aquest cas, el millor és posar un altre node en el punt de Fermat del triangle que formen i obtenir un graf S_4. Donat un triangle acutangle, el punt de Fermat (també anomenat $X(13)$) és el punt tal que la suma de les distàncies des de cada vèrtex fins a aquest punt

és la mínima. El punt de Fermat es determina gràficament amb el procediment següent:

- Donat un triangle T, es generen dos triangles equlàters a partir de dos costats arbitraris de T.
- S'uneixen els nous vèrtexs externs dels triangles equilàters amb els vèrtexs oposats de T.
- L'intersecció d'aquests dos segments dóna la posició del punt de Fermat.

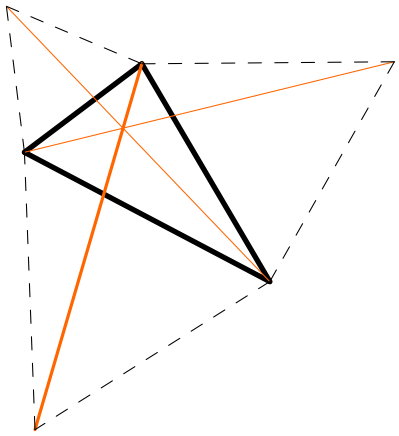


Figura 27: Procés de construcció del punt de Fermat per a un triangle

Aquest procediment és vàlid per a triangles amb angles menors a 120º, però en cas contrari, el punt de Fermat s'identificarà amb vèrtex corresponent a l'angle que els superi.

El punt de Fermat es pot generalitzar per a altres polígons convexos a partir de triangulacions d'aquests. Un exemple senzill és el del quadrilàter, on en fer les diagonals ja es generen quatre triangles. En aquest cas, trobant els punts de Fermat en dos dels triangles oposats i unint-los ja es connecten tots els punts del quàdrilàter amb la menor distància possible.

El problema de connectar amb la mínima distància un nombre determinat de punts essent possible afegir nodes és conegut com el problema de l'arbre de Steiner. Per aquest problema s'ha demostrat que l'arbre òptim té com a màxim $n-2$ nodes afegits (siguent n el nombre inicial de nodes), que aquests tenen sempre grau 3, i que formen sempre angles de 120º.

(Afegir un apartat per parlar de les bombolles de sabó)

9 Arbres expansius

Quan hi ha una situació on no es pot afegir cap node, el millor sol ser utilitzar els algorismes per trobar arbres expansius que ja s'han vist. Una opció és construïr el graf complet amb totes les arestes i els seus corresponents pesos i executar algun algorisme que en trobi l'arbre expansiu mínim.

Part IV
Topologia

La topologia és la branca de les matemàtiques que estudia les propietats de l'espai i si aquestes es conserven després de deformacions. Des d'un punt de vista teòric,

10 Isomorfismes

Es diu que dos grafs són isomorfs si existeix una funció bijectiva φ entre els seus vèrtexs que conservi les arestes. Formalment dos grafs G i H són isomorfs si

$$\exists \varphi : V(G) \to V(H) \text{ tal que } \overline{\varphi(v)\varphi(u)} \in E(H) \text{ si i només si } \overline{vu} \in E(G)$$

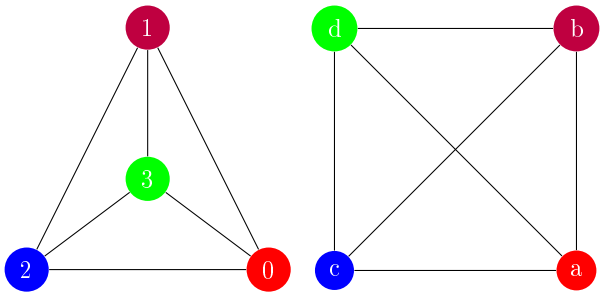


Figura 28: Grafs isomorfs

(S'haurà d'eliminar l'apartat de topologia, encabir els siomorfismes amb l'exemple a algun altre apartat [segurament a conceptes bàsics] i fer l'apartat d'aplicaicions pràctiques)

Part V
Aplicacion pràctiques de la teoria de grafs

11 Algorisme PageRank

Una de les aplicacions de teoria de grafs que utilitzem cada dia és l'algorisme PageRank de Google. A través d'aquest algorisme, aquest cercador pot saber quines pàgines web són més importants i, per tant, en quin ordre s'han de presentar els resultats d'una cerca.

Aquest algorisme conta el nombre de links cap a una pàgina web, així com l'importància de la pàgina web on estan posats, per saber quina importància té una web. La presuposició que fan és que quans més links cap a una web concreta, més important és.

S'ha de pensar el conjunt de pàgines web com un graf dirigit i poderat: cada vèrtex representa una pàgina i hi ha una aresta dirigida de del vèrtexv_i fins a v_j si hi ha un enllaç a v_i que porta a v_j. El pes de l'aresta $e = (v_i, v_j)$, $w(e)$, serà la probabilitat de passar del vèrtex v_i al vèrtex v_j (és a dir, $\frac{1}{n}$, on n és el nombre d'arestes que surten de v_i).

Un cop es té la matriu de probabilitats (la matriu d'adjacència del graf) es fa que un programa o bot navegui de manera aleatòria per aquest graf, d'acord amb les probabilitats de passar per cada aresta. Quan el programa ha estat un temps recorrent es fa el recompte de quantes vegades ha passat per a cada pàgina, i a paritr d'aquest valor es determina la posició en els resultats.

12 Productes relacionats

Quan es visita una botiga online i s'està mirant un producte, sovint apareix una secció de productes recomenats o relacionats. Hi ha diverses maneres de saber quins productes estàn relacionats a aquell que s'està mirant, i una de les que més s'utilitza es basa en teoria de grafs.

El que es fa és anar construïnt un graf a mesura que els usuaris van visitant la web: cada producte és un vèrtex, i es genera una aresta dirigida cap a un altre producte quan aquest és visitat immediatament després del primer. Per exemple, si un usuari busca un ordinador portàtil, el més probable és que al acabar de mirar-ne un en veigi un altre. En aques cas es generaria una aresta des del primer ordinador fins al segon. A mesura que els usuaris van utilitzant la web, es van

generant arestes i el graf va creixent. Tot i això, és possible que l'usuari, al acabar de mirar l'ordinador es posi a buscar un altre article que no hi té res a veure, com per exemple un paraigües. Però això no és habitual, i hi haurà molt poques arestes que uneixin l'ordinador i el paraigües.

Sobre aquest graf s'executarà un algorisme de clusterització, que buscarà parts del graf que siguin més denses o altament conexes. Auquestes parts segurament seràn d'una temàtica concreta, com per exemple el grup dels ordinadors i hi haurà molt poques arestes que surtin d'un grup cap a un altre.

13 Instal·lació de càmeres de videovigilància

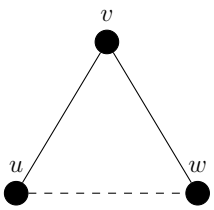
Un altre exemple és la optimització de sistemes de videovigilància. Amb teoria de grafs es pot saber quin és el màxim nombre de càmeres necessàries per poder vigilar un local, tingui la forma que tingui. Aquest problema és conegut com el problema de la galeria d'art

La intució diu que, per una sala poligonal amb n vértexs, es pot posar una càmera a cada cantonada (que resulta amb un total de n càmeres) o de manera més òptima, posant una càmera cada dos vèrtexs (que comporta tenir $\frac{n}{2}$ càmeres). Tot i això, Václav Chvátal afirma en el seu teorema que sempre s'hauràn d'utilitzar com a màxim $\frac{n}{3}$.

13.1 Demostració i procediment

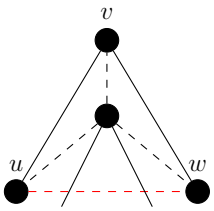
La resolució d'aquest problema es basa en el fet de que qualsevol polígon pot ser descompost en triangles, i més concretament que sempre es pot traçar una corda en el polígon. *Demostració*: S'agafa un vértex v qualsevol del polígon. Poden passar dues coses:

- La corda que formen els vértexs adjacents a v, que anomenarem u i w, està completament dins el polígon. En aquest cas es pot traçar la corda sense problemes.



- El propi polígon creua la corda entre u i w . Això voldrà dir que a l'espai delimitat pel triangle que formaríen v,u i w hi haurà com a mínim un altre

vèrtex del polígon. En aquest cas també es podràn traçar diverses cordes amb aquest vértex.



El resultat de descompondre el polígon en triangles és un graf els vèrtex dels quals poden ser pintats en tres colors, de tal manera que cada triangle tingui un vèrtex de cada color (d'aquí l'aproximació de $\frac{n}{3}$). Qualsevol conjunt de vértex d'un matiex color és un conjunt de posicions vàlides per les càmeres de seguretat. Tan sols cal veure quin és el conjunt de menys elements per poder determinar les posicions vàides de les càmeres.

Cal puntualitzar que hi ha casos on amb aquest procediment no es troba la solució òptima al problema, és tan sols una bona aproximació.

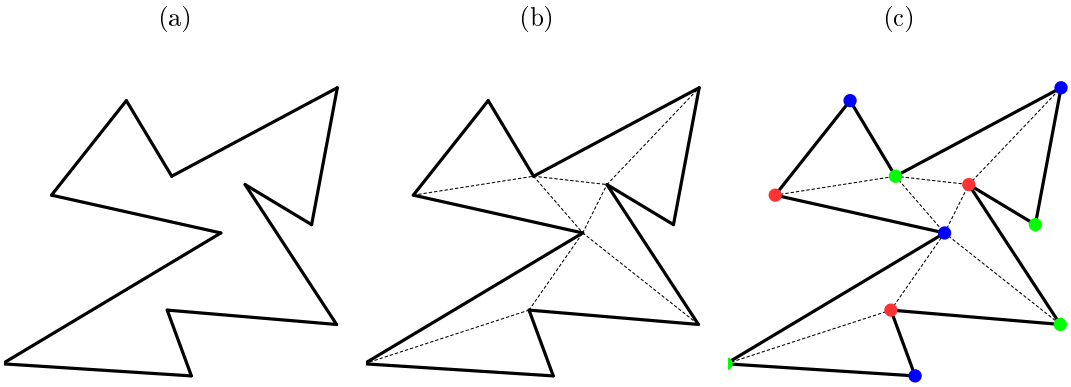


Figura 29: Exemple del procés de triangulació i coloració d'un polígon irregular

14 Metro

Una xarxa de metro és un altre bon exemple de graf: es pot pensar com un conjunt de grafs elementals P_n amb la propietat de que alguns d'aquests comparteixen nodes. Així doncs, com a una última aplicació pràctica, em vaig proposar fer un programa que sigués capaç de calcular un trajecte entre dues estacions qualsevols

i el temps aproximat de recorregut que es trigaria, incloent-hi els possibles transbords entre línies.

14.1 Metodologia de treball

El primer gran repte consistia en representar la basta xarxa de metro de Barcelona en forma de graf. Cada estació correspondria a un node i dos nodes serien adjacents si i només si les estacions estan connectades, ja sigui a través del propi metro o a través de passadissos (cas que tan sols apareix quan diverses línies comparteixen estació). Això comporta representar totes les adjacències de cadascuna de les estacions. Per fer-ho, es va dividir la xarxa total en cadascuna de les línies. Per poder tenir en compte els temps de transbord entre línies d'una mateixa estació, es va crear una estació diferent per a cada línia.

Un cop l'adjacència del graf va estar completa, calia posar pesos a cadascuna de les arestes, corresponents al temps de viatge entre estacions mitjançant el metro o, en cas dels transbords, el temps caminant entre les andanes. Per aconseguir aquests temps, es va anar un dia a recòrrer la xarxa sencera, tot cronometrant els intervals de temps necessaris. Tots els temps van ser introduïts al graf en segons, per tal de facilitar el posterior càlcul del temps total.

14.2 Consideracions

Cal tenir remarcar que els temps introuïts al graf són els temps que va trigar el metro el duimenge 30 d'octubre de 2016, i que aquests no són constants, sinó que poden variar en funció del dia, l'hora i molts altres factors que no es poden tenir en compte. El mateix passa amb el temps de les parades. Es va observar que el temps que el metro estava parat en una estació incrementava en 5 segons a estacions concretes i a mesura que s'anava apropant al centre de la ciutat. Com que és un factor que també varia en funció de molts paràmetres i massa comlex per tenir-lo en compte, s'ha establert que el temps de parada és d'uns 25 segons de mitjana. Aquest temps s'afegeix al total multiplicat pel nombre de parades menys dues (la inicial i la final). Així doncs, els valors de temps que calculi el programa seràn relatius i aproximats.

El format de les estacions en el graf és el següent:

(número de línia)_(nom d'estació)

Els noms de les estacions estan posats d'acord el plànol de la xarxa de metro que proporciona TMB (Veure figura30).

Figura 30: Plànol de la xarxa de metro de Barcelona [11]

14.3 Algorisme

El programa que fa els càlculs funciona a partir d'una execució de l'algorisme de Dijkstra (en realitat en una versió modificada que pot tractar amb paraules com a noms de nodes).

[11] Autor: Ferrocarril Metropolità de Barcelona, S.A. Tots els drets reservats a l'autor

```
Funció DijkstraOrdenat(G, s)
    nou diccionari dist amb les mateixes claus que G
    nou diccionari Q amb les mateixes claus que G
    nou diccionari arbre
    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
                    Q[v] = Q[u] + w(u, v)
                    arbre[v] = u
                end
            end
        end
        elimina(Q[u])
    end
    return dist, arbre
```

Dijkstra retorna un diccionari amb els temps des del node inicial s fins a la resta de nodes i un segon diccionari amb l'arbre expansiu que té s com a arrel. Per saber el recorregut a fer, només cal seguir l'arbre de manera inversa, és a dir des del punt final fins a s. Al ser un arbre, només hi haurà un camí possible a seguir.

Per obtenir el temps caldrà buscar l'entrada corresponent al node final al diccionari dels temps. A aquest temps se li afegiran els temps per a cada parada i es farà la conversió de segons a minuts i segons.

Data: Graf G del metro, un node inicial *inici* i un node final $final$
Result: Temps total del trajecte, trajecte
nou vector *recorregut*
imprimeix("Punt inicial:", *inici*)
imprimeix("Punt final:", $final$)
dist, arbre = DijkstraOrdenat(G, *inici*)
$i = final$
while *arbre*[i] *sigui diferent a inici* **do**
    afegeix *arbre*[i] a *recorregut*
    $i = arbre[i]$
end
afegeix *inici* a *recorregut*
inverteix l'ordre dels elements de *recorregut*
$TempsTotal = dist[final] + (25 \times (len(recorregut) - 2)$
$minuts$ = part entera de $TempsTotal/60$
$segons$ = (residu de $TempsTotal/60) \times 0'60$
imprimeix("Temps total del recorregut:", $minuts$, "minuts i", $segons$, "segons")
imprimeix("Recorregut:")
for i *in* $range(0, len(recorregut))$ **do**
    imprimeix($recorregut[i]$)
end