TEORIA DE GRAFS I ALGORISMES:
BUSCANT EL CAMÍ MÉS CURT

Treball de recerca 2019-2020

Autor: Jacobi

Resum

Trobar la ruta per anar des d'un lloc a un altre és una informació crucial que necessitem si no coneixem el lloc a on anem. A més, sovint ens interessa no només obtenir una ruta que ens porti cap a la nostra destinació, sinó que volem la ruta que, de totes les rutes possibles, és la més curta. En aquest treball de recerca, descobrirem com funcionen els algorismes per trobar la ruta més curta entre dues ubicacions d'un mapa i utilitzarem diverses tècniques per optimitzar-los, com l'ús de funcions heurístiques o la cerca bidireccional. La branca matemàtica de la teoria de grafs, una part de les matemàtiques generalment desconeguda pels estudiants de secundària, és la base en què operen els algorismes que farem servir. Explorarem els conceptes fonamentals de la teoria de grafs i algunes aplicacions que tenen els grafs en el món real. Finalment, gràcies als grafs i als algorismes, desenvoluparem un planificador de rutes pel poble de Calldetenes. A part de poder trobar el camí més curt, el nostre planificador de rutes inclourà addicionalment un visualitzador d'algorismes innovador i únic que mostra com funcionen els diversos algorismes durant la cerca. També compararem els mètodes utilitzats en el nostre planificador de rutes amb els mètodes utilitzats per Google Maps, el planificador de rutes més utilitzat i avançat del món.

Abstract

Finding the route to go from one place to another is a piece of crucial information we need if we do not know the site where we are going. Moreover, we are often interested not only in obtaining a route to reach our destination but also on getting the route that, from all the possible routes, is the shortest. In this research project, we will discover how algorithms work to find the shortest path between two locations on a map, and we will use various techniques to optimize them, such as the use of heuristic functions or bidirectional search. The mathematical branch of graph theory, a part of mathematics generally unknown to high school students, is the base on which the algorithms we will use operate. We will explore the fundamental concepts of graph theory and some applications that graphs have in the real world. Finally, thanks to graphs and algorithms, we will develop our route planner for the town of Calldetenes. Besides finding the shortest path, our route planner will additionally include an innovative and unique algorithm visualizer that shows how the different algorithms work during the search. We will also compare the methods used in our route planner to the methods used by Google Maps, the most used and advanced route planner in the world.

Agraïments

M'agradaria donar les gràcies, primer de tot, a la tutora del meu treball de recerca per l'orientació i els consells que m'ha donat i la constant revisió al llarg de tot el procés de la confecció d'aquest treball. També voldria mostrar el meu agraïment a Albert Graells Rovira, que treballa a Google, per donar-me recursos per fer recerca i per haver-me respost totes les preguntes sobre el funcionament intern de Google Maps. De la mateixa manera, també vull agrair a totes les persones que voluntàriament han fet realitat el projecte OpenStreetMap, que lliurement m'ha proporcionat les dades per fer possible la cerca de camins sobre un mapa real. Igualment, estic agraït als professors de matemàtiques, tecnologia i informàtica de l'institut per ensenyar-me i fer-me interessar en les matemàtiques i la programació. Així mateix, m'agradaria agrair a la meva família pel suport que m'han donat i per repassar el treball. I finalment, a tota la gent que de manera directa o indirecta han fet possible aquest treball, i a tothom que s'hagi interessat per ell.

Índex

Presentació

Els humans sempre hem viatjat. Sigui a peu, en cotxe, en tren, en vaixell, amb avió o amb coets, les persones ens hem desplaçat d'un lloc a un altre des de temps immemorials. Però trobar com anar des d'un lloc a un altre sense conèixer com és la zona per on hem de passar no és una tasca fàcil. Això encara es complica més si a sobre volem anar-hi de la forma més òptima possible, és a dir, passant pel camí més curt.

En aquest treball de recerca exploraré quins són alguns dels algorismes i les tècniques que es poden fer servir per resoldre la tasca de trobar el camí més curt entre dos punts en un mapa. A partir d'aquests algorismes, el meu objectiu final serà desenvolupar un planificador de rutes informàtic pel meu poble, Calldetenes. Aquest planificador de rutes permetrà, com qualsevol programa similar, trobar el camí més curt. A part, també tindrà la singularitat de poder visualitzar el funcionament dels algorismes que ho fan possible.

Tots els algorismes que treballaré se sustenten per la teoria de grafs, una interessant branca de les matemàtiques desconeguda per la gran majoria d'alumnes de secundària, però que té moltíssimes aplicacions en camps tan diversos com l'enginyeria, la ciència i l'economia.

Des de petit m'han agradat les matemàtiques i la informàtica. El meu interès per aquests dos camps em va portar a participar en diferents concursos de programació competitiva. Arran d'això vaig començar a aprendre de forma autodidàctica com programar algorismes per solucionar els problemes que es plantejaven en aquests concursos. Per resoldre molts d'aquests problemes es necessiten algorismes que treballen amb grafs. Per això, vaig pensar que el tema que he triat per fer el treball de recerca seria una molt bona oportunitat per aprendre més sobre els grafs i alguns dels algorismes més importants que els tracten. I així, alhora que aprenc sobre grafs, algorismes i a fer recerca, puc treballar en quelcom que m'agrada i m'interessa especialment.

Per iniciar amb els grafs, començarem amb el problema que va fer-los originar: el problema dels set ponts de Königsberg.

"Aquesta qüestió és molt banal, però em va semblar digna d'atenció, ja que ni la geometria, ni l'àlgebra, ni tan sols l'art de comptar eren suficients per resoldre-la."

— Leonhard Euler, en una carta a Giovanni Marinoni parlant del problema dels set ponts de Königsberg, 1736.

1

El problema dels set ponts de Königsberg

Heus aquí una vegada, a principis del segle XVIII, una pintoresca ciutat anomenada Königsberg, es trobava al nord-est de Prússia, a banda i banda del riu Pregel. La ciutat tenia dues illes al mig del riu, les quals estaven connectades amb set ponts.

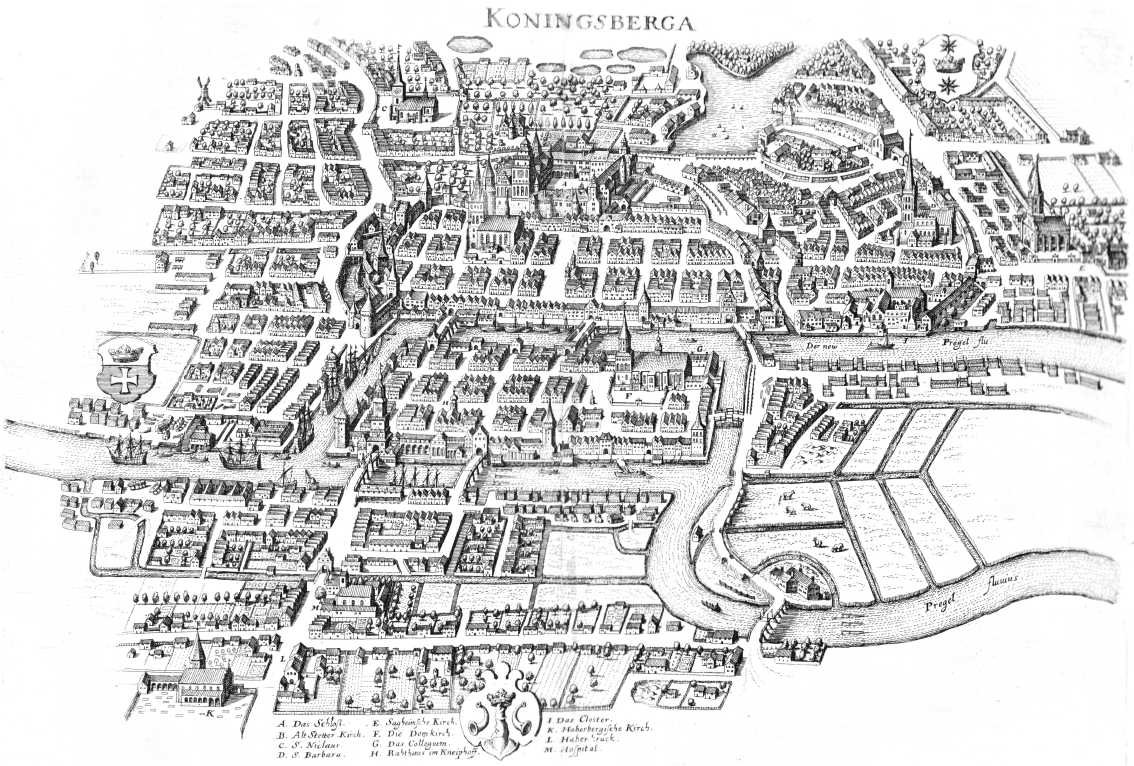

Figura 1.1: Mapa del Königsberg del segle XVIII.

Els habitants de Königsberg solien passar les tardes del diumenge passejant per la seva bella ciutat. Un dia, se'ls hi va acudir fer un joc: el seu objectiu era idear una manera de poder caminar per la ciutat creuant cadascun dels set ponts una sola vegada. Tot i que cap dels ciutadans de Königsberg aconseguia trobar tal ruta, cap d'ells tampoc podia demostrar que fos impossible.

El matemàtic i alcalde de Danzig (ciutat propera a Königsberg), Carl Gottlieb Ehler, es va obsessionar amb trobar una ruta que complís l'objectiu del joc, fins al punt que, l'any 1735, Ehler va escriure una carta demanant ajuda per resoldre el problema

al famós matemàtic Leonhard Euler, que en aquell temps vivia a Sant Petersburg. Euler va respondre la carta l'any següent dient-li que creia que aquell problema era una qüestió banal amb poca relació amb les matemàtiques. Però com més voltes li donava, més veia que, en realitat, aquell problema era bastant complicat i interessant, així que va decidir resoldre'l. El mateix any 1736, Euler va presentar l'article *Solutio problematis ad geometriam situs pertinentis* en què soluciona el problema.

Al principi de l'article, Euler exposa que no creu que sigui una bona solució fer una llista exhaustiva de totes les rutes possibles i després mirar si alguna satisfà les condicions del joc, ja que aquest mètode seria molt difícil i laboriós, i impossible amb problemes amb més ponts. Un cop rebutjat aquest mètode, explica la manera en què ha resolt el problema.

Primer de tot, representa les quatre àrees amb les lletres A, B, C i D i els set ponts amb les lletres a, b, c, d, e, f i g com es pot veure en la figura 1.2.

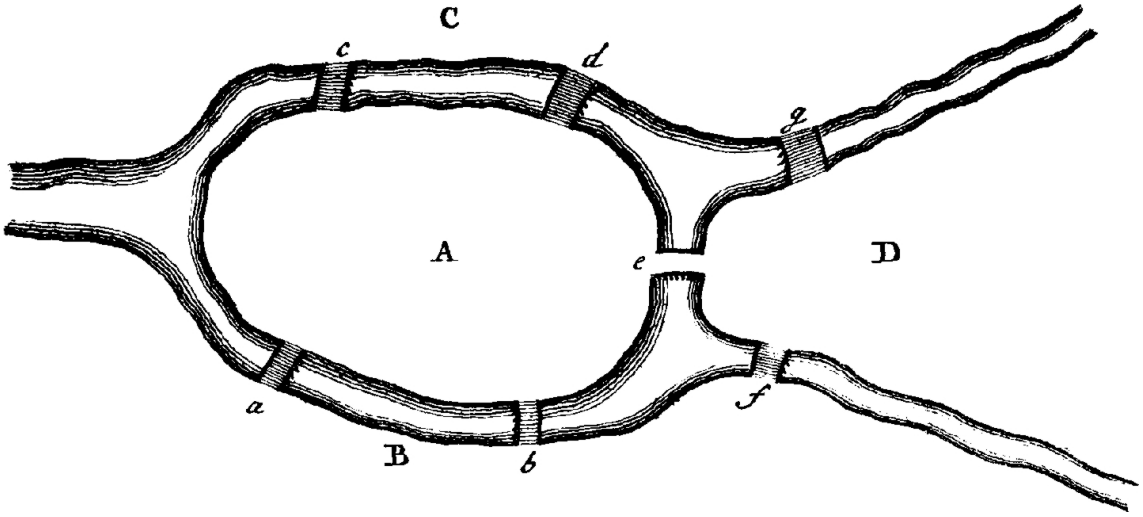

Figura 1.2: Dibuix original d'Euler del problema.

A continuació, explica el mètode que fa servir, basat en una manera especialment convenient de representar el pas per un pont:

Si algú creua des d'A fins a B pel pont a o b, escriu AB, on la primera lletra es refereix a l'àrea de la qual se surt, i la segona a l'àrea a la qual s'arriba després de creuar el pont. Així mateix, si se surt de B i es creua a D pel pont f, representa aquest creuament amb BD, i dos moviments AB i BD combinats els denota amb les tres lletres ABD, on la lletra B del mig es refereix tant a l'àrea entrada pel primer pont com a la sortida pel segon pont.

Similarment, si ara es va des de D fins a C pel pont g, es representen els quatre creuaments successius amb les quatre lletres $ABDC$, representant que es comença a A, es creua a B, es continua cap a D i finalment s'arriba a C. Com que cada àrea està separada de les altres pel riu, s'han d'haver travessat tres ponts. De la mateixa manera, el creuament successiu de quatre ponts seria representat per cinc lletres i, de

forma general, el camí és indicat per un nombre de lletres igual al nombre de ponts creuats més u. Així doncs, per travessar set ponts un sol cop haurem d'escriure vuit lletres per representar la ruta.

En aquest mètode de representació no es té en compte el pont pel qual es passa, sinó només les àrees. Si el pas d'una àrea a una altra es pot fer per diversos ponts, es pot fer servir qualsevol pont que arribi a l'àrea requerida. A partir d'això podem veure que si un camí pels set ponts de Königsberg pot ser dissenyat per tal que cada pont només es creui un cop, la ruta podrà ser representada per vuit lletres que tindran un ordre tal que les lletres A i B estaran de costat dues vegades, ja que hi ha dos ponts, a i b, connectant les àrees A i B; de la mateixa manera, A i C també hauran de ser adjacents dos cops en la sèrie de vuit lletres, i finalment els parells A i D, B i D, i C i D hauran d'estar junts un cop cada un.

El problema es redueix, doncs, a trobar una seqüència de vuit lletres, formades a partir de les quatre A, B, C i D en què els diversos parells de lletres apareguin el nombre requerit de vegades descrit en el paràgraf anterior. Però, abans de començar a buscar tal seqüència, seria útil veure si és possible o no ordenar les lletres d'aquesta manera, ja que si fos possible mostrar que no existeix tal disposició ja no caldria buscar-la. Per això, Euler tracta de buscar una regla que sigui útil en aquest cas i en altres per determinar si tal disposició pot existir.

Per trobar la regla, Euler imagina una sola àrea A on hi ha diversos ponts a, b, c, d, etc. com es pot veure en la figura 1.3. Considerem que una persona travessa tan sols el pont a: si algú travessa aquest pont, per força ha d'haver-se trobat a A abans de creuar o ha d'haver arribat a A després de creuar. Si considerem que travessa tres ponts (com a, b i c), llavors en la representació de la ruta la lletra A hi serà dues vegades sense importar si va començar des d'A o no. Similarment, si cinc ponts condueixen a A, la representació d'una ruta que passi per tots tindrà tres ocurrències de la lletra A. De manera general, si el nombre de ponts és un nombre senar n, els cops que apareixerà la lletra A serà $(n+1)/2$.

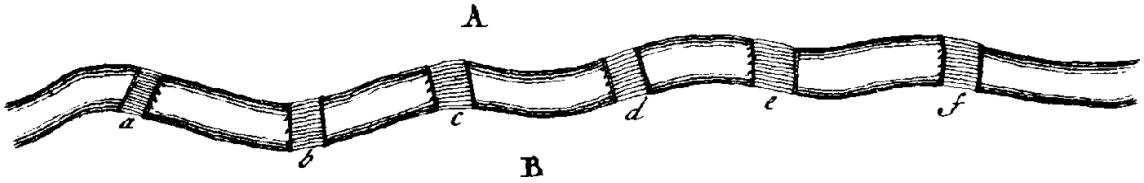

Figura 1.3: Dibuix original d'Euler del cas considerat per trobar la regla.

Per tant, en el cas dels set ponts de Königsberg (figura 1.2) hi ha d'haver tres vegades la lletra A en la representació de la ruta, ja que cinc ponts (a, b, c, d i e) condueixen cap a l'àrea A. A més, com que hi ha tres ponts que porten a B, la lletra B haurà de ser-hi dos cops. De la mateixa manera, hi haurà d'haver dues vegades la lletra D i la C. És a dir, en una seqüència de vuit lletres la lletra A haurà de ser-hi tres vegades i les lletres B, C i D hauran de ser-hi dos cops cadascuna. Si sumem $(3+2+2+2=9)$ veiem que no és possible fer-ho en una seqüència de només vuit lletres, arribant així a la conclusió que és impossible crear una ruta que passi per

tots els set ponts de Königsberg un sol cop.

Però Euler no s'atura aquí, ja que també troba en quins altres casos sí que és possible fer un camí que travessi tots els ponts una sola vegada. Les regles que troba són les següents:

1. Si hi ha més de dues àrees a les quals hi arriben un nombre senar de ponts, és impossible crear un camí que compleixi les condicions.
2. Tanmateix, si el nombre de ponts és senar per a exactament dues àrees, el camí és possible si comença en qualsevol d'aquestes dues àrees.
3. Finalment, si no hi ha àrees cap a les quals hi arribi un nombre senar de ponts, es pot fer el camí començant des de qualsevol àrea.

La demostració de per què aquestes regles funcionen per a qualsevol cas es troba en el següent capítol, en l'apartat sobre camins eulerians.

Veiem que en el cas original dels set ponts de Königsberg el camí no és possible perquè incompleix la primera regla, ja que a totes quatre àrees hi arriben un nombre senar de ponts: a l'àrea A n'arriben cinc i a les altres, tres.

I doncs, com podríem crear un camí perquè els habitants de Königsberg poguessin aconseguir guanyar el joc? És fàcil, simplement hem d'eliminar algun pont perquè es compleixi alguna de les regles. Resulta que això mateix és el que va passar el cap de més de dos-cents anys. Durant la Segona Guerra Mundial, la força aèria soviètica va bombardejar la ciutat de Königsberg, que en aquell temps pertanyia a l'Alemanya nazi. Les bombes van destruir dos dels ponts de la ciutat (el b i el d), fent així que l'àrea A passés a ser connectada per tres ponts, les àrees B i C per dos ponts i l'àrea D seguís amb tres ponts com abans.

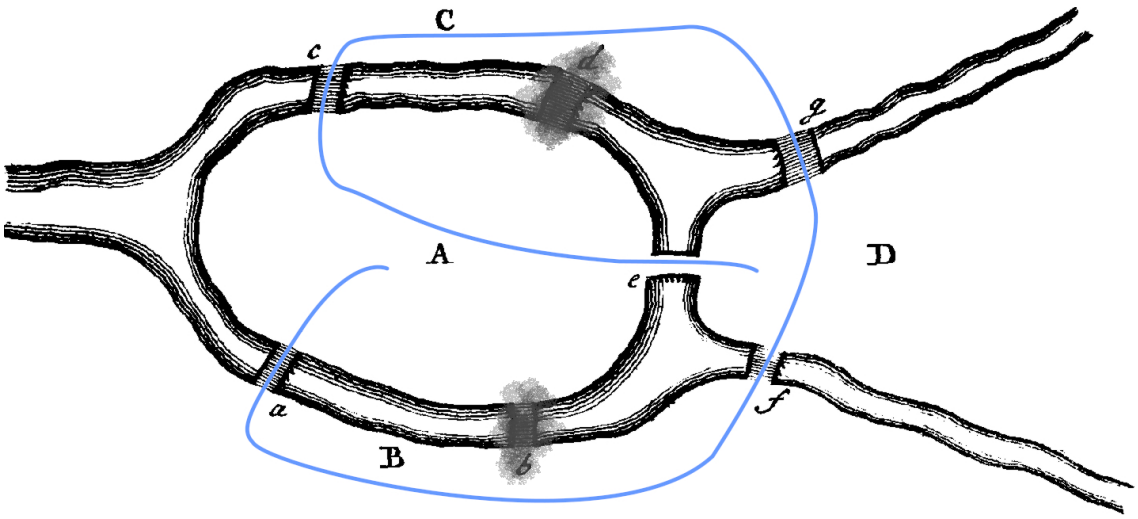

Figura 1.4: Camí $ABDCAD$, el qual passa per tots els cinc ponts una sola vegada.

Aquesta situació, en què hi ha exactament dues àrees connectades per un nombre senar de ponts, és justament la que fa que es compleixi la segona regla, permetent

així un camí com el que volem! Tot i això, hi ha la condició que hem de començar en una de les dues àrees amb nombre senar de ponts, és a dir, el camí ha de començar des de l'àrea A o la D. Un exemple de camí que passi per tots els cinc ponts encara en peu és el camí $ABDCAD$, representat en la figura 1.4.

Els bombardejos de Königsberg, a part de destruir els ponts, pràcticament també van esborrar la ciutat del mapa i van permetre que fos ocupada pels soviètics. Els següents anys es va reconstruir com la ciutat russa actualment coneguda com a Kaliningrad.

Figura 1.5: Königsberg després de la batalla, 1945.

Així doncs, encara que Königsberg i els seus set ponts ja no existeixin, seran recordats al llarg de la història per ser un enigma aparentment trivial que va conduir a l'aparició d'una nova branca de les matemàtiques: la teoria de grafs.

"Siguin quines siguin les voltes d'un o més fils en l'espai, sempre es pot obtenir una expressió per al càlcul de les seves dimensions, però aquesta expressió no serà gaire útil en la pràctica. L'artesà que confecciona una trena, una xarxa o nusos, no es preocuparà per qüestions de mesura, sinó de posicions; allò que li importarà és com els fils estan entrellaçats. Per tant, seria útil tenir un sistema de càlcul més rellevant, una notació que representés aquesta manera de pensar i que es podria utilitzar per a la reproducció d'objectes similars per sempre més."
— Alexandre-Théophile Vandermonde, *Notes sobre problemes de posició*, 1771.

2

Introducció a la teoria de grafs

Un *graf* $G = (V, E)$ és una estructura matemàtica que consisteix en un conjunt de *vèrtexs* $V = \{v_1, v_2, v_3, ...\}$ i un conjunt d'*arestes* $E = \{e_1, e_2, e_3, ...\}$[1]. Cada aresta e consisteix en el parell de vèrtexs (v_x, v_y), que poden ser diferents o iguals.

2.1 Representació visual

La forma habitual de representar un graf és dibuixant un cercle per a cada vèrtex i una línia que enllaça els vèrtexs connectats per una aresta. Com es dibuixen aquests cercles i línies es considera irrellevant: l'únic important és la informació de quins parells de vèrtexs formen una aresta i quins no ho fan.

Per exemple, un graf

$G = (\{v_1, v_2, v_3, v_4\}, \{(v_1, v_2), (v_1, v_2), (v_1, v_3), (v_1, v_3), (v_1, v_4), (v_2, v_4), (v_3, v_4)\})$

es pot representar de la següent manera:

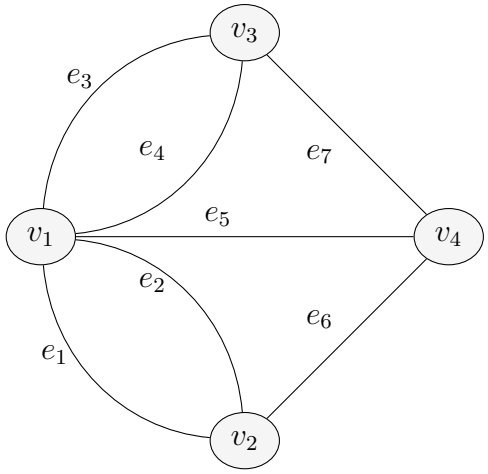

Figura 2.1: Diagrama del graf G.

El diagrama del graf G de la figura 2.1 justament representa la situació presentada en el capítol anterior sobre el problema dels set ponts de Königsberg. En el dibuix original d'Euler de la figura 1.2, les àrees A, B, C i D serien els vèrtexs v_1, v_2, v_3

[1] E i e provenen de l'anglès *edge*, que en català vol dir *aresta*.

i v_4 respectivament i, de la mateixa manera, els ponts a, b, c, d, e, f i g serien les arestes e_1, e_2, e_3, e_4, e_5, e_6 i e_7.

Euler no va utilitzar aquesta representació en el seu article perquè com que va ser el primer article relacionat amb la teoria de grafs encara no hi havia una manera establerta de representar grafs. No va ser fins a finals del segle XIX que van començar a aparèixer aquest tipus de diagrames que avui en dia s'utilitzen habitualment per representar grafs.

2.2 Definicions

El nombre de vèrtexs d'un graf s'anomena *ordre* i es denota $|V|$. El nombre d'arestes d'un graf s'anomena *mida* i es denota $|E|$. Pel graf G representat $|V| = 4$ i $|E| = 7$.

Els vèrtexs que formen una aresta s'anomenen els *extrems* de l'aresta. Per exemple, els extrems de l'aresta e_1 són els vèrtexs v_1 i v_2. Alternativament, també es pot dir que e_1 és *incident* en v_1 i v_2, que v_1 *connecta* amb v_2 (i viceversa) o que v_1 i v_2 són *adjacents*.

Si dues arestes tenen els mateixos extrems, es diu que són *paral·leles*. En el graf G hi ha dos parells d'arestes paral·leles: e_1 i e_2 perquè les dues són incidents en v_1 i v_2, i e_3 i e_4 perquè les dues són incidents en v_1 i v_3.

S'anomena *bucle* a una aresta que connecta un vèrtex amb si mateix, és a dir, si els dos extrems són iguals. En el graf G no hi ha cap bucle, ja que totes les arestes tenen els extrems diferents. Un exemple de bucle és l'aresta e_5 del graf H de la figura 2.3.

El *grau* d'un vèrtex, $g(v)$, és el nombre d'arestes incidents en v. Si mirem el graf G, veiem que $g(v_1) = 5$ i que $g(v_2) = g(v_3) = g(v_4) = 3$. Un vèrtex de grau zero, com v_6 d'H, es diu vèrtex isolat.

Si el grau d'un vèrtex és un nombre parell, s'anomena un *vèrtex parell*. De la mateixa manera, si el grau d'un vèrtex és senar, s'anomena *vèrtex senar*. En el cas del graf G, tots els vèrtexs són senars.

Un *subgraf* S d'un graf G és un altre graf format a partir d'un subconjunt dels vèrtexs i les arestes de G. Per exemple, el graf de la figura 2.2 és un subgraf S del graf G.

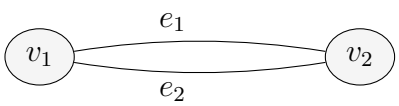

Figura 2.2: Diagrama del subgraf $S = (\{v_1, v_2\}, \{(v_1, v_2), (v_1, v_2)\})$.

Un graf es diu que és *connex* si tots els vèrtexs són accessibles des de tots els altres vèrtexs. El graf G és un graf connex. Un exemple de graf no connex és el graf H de la figura 2.3.

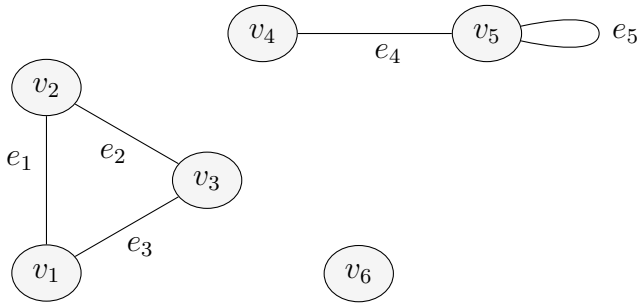

Figura 2.3: Diagrama del graf H.

Cada subgraf connex d'un graf s'anomena *component*. El graf G té un sol component i el graf H té tres components.

2.3 Camins

Un *camí* és una seqüència de vèrtexs i arestes d'un graf en què les arestes connecten els vèrtexs del camí.

Per exemple, podem crear un camí P[2] en el graf de la figura 2.4 que comenci en el vèrtex v_1, passi per l'aresta e_6 per arribar al vèrtex v_4, després continuï per l'aresta e_5 per arribar a v_3 i finalment vagi per e_2 per acabar en el vèrtex v_2. Aquest camí el podem escriure $P : v_1 \overset{e_6}{\text{—}} v_4 \overset{e_5}{\text{—}} v_3 \overset{e_2}{\text{—}} v_2$.

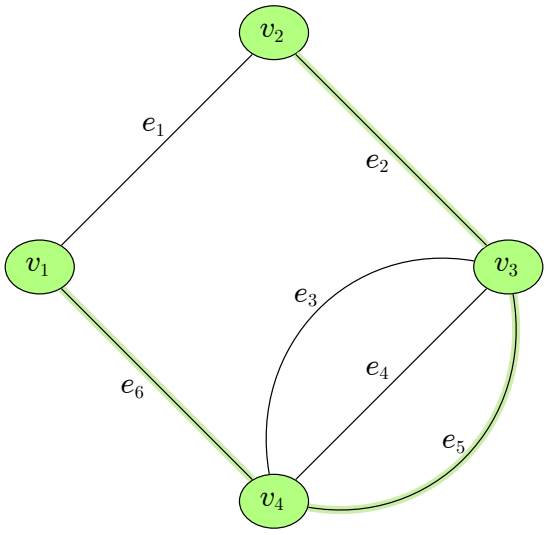

Figura 2.4: Graf amb el camí P pintat de color verd.

El nombre d'arestes que té un camí P s'anomena *longitud* i es denota $l(P)$. Per exemple, en el camí anterior $l(P) = 3$, ja que passa per tres arestes.

[2]La P prové de l'anglès *path*, que en català vol dir *camí*. No s'utilitza C perquè C es fa servir per denotar circuits (explicats a continuació).

Si un camí comença i acaba en el mateix vèrtex se l'anomena *circuit*. S'anomena *cicle* a un circuit en què els únics vèrtexs que es repeteixen són l'inicial i el final. Quan un graf no conté cap cicle s'anomena *acíclic*.

2.3.1 Camins eulerians

Un *camí eulerià* és un camí que passa per totes les arestes d'un graf un sol cop. Es diu així en honor a Leonhard Euler, que va solucionar el problema dels set ponts de Königsberg (explicat en el capítol anterior).

Quan un camí eulerià és un circuit, s'anomena *circuit eulerià*.

Si un graf conté un camí eulerià, s'anomena *graf eulerià*. Perquè un graf sigui eulerià s'han de complir les tres regles exposades en el capítol anterior, que ara podem reescriure en forma dels següents dos teoremes amb la terminologia dels grafs introduïda en aquest capítol.

Teorema 1. *Un graf G té un circuit eulerià si i només si tots els vèrtexs de G són parells.*

Demostració.
($\Longrightarrow$): Primer de tot demostrem que si G té un circuit eulerià, llavors tots els vèrtexs de G han de ser parells. Quan fem un circuit eulerià C_E, visitem cada vèrtex un cert nombre de vegades. Sigui v_a un vèrtex diferent de l'inicial de C_E, i assumim que passem per v_a exactament a vegades. Això voldrà dir que haurem entrat en v_a exactament a vegades, i sortit del vèrtex v_a exactament a vegades també. Com que no podem tornar a passar per la mateixa aresta més d'un cop, ja que es tracta d'un camí eulerià, haurem fet servir $2a$ arestes. Per altra banda, C_E conté totes les arestes de G, és a dir, v_a no pot tenir cap aresta addicional; per tant, $g(v_a) = 2a$. Això demostra que qualsevol vèrtex excepte el vèrtex inicial v_i de C_E és parell. Finalment, ens hem de fixar que v_i no només és el vèrtex inicial, sinó també el final perquè es tracta d'un circuit; per tant, si visitem v_i exactament n vegades entre el principi i el final de C_E, haurem passat per $1 + 2n + 1 = 2n + 2 = 2(n + 1)$ arestes, de manera que $g(v_i) = 2(n + 1)$, és a dir, v_i també serà parell, la qual cosa fa que l'afirmació quedi demostrada.

($\Longleftarrow$): Ara assumim que tots els vèrtexs de G són parells i demostrem que G conté un circuit eulerià. Prenem qualsevol vèrtex v_i, i comencem el camí P_1 per una aresta e_1. Continuem fent servir les arestes que encara no s'han utilitzat en els passos anteriors fins que es forma un circuit C_1 que revisita un vèrtex ja visitat. Com que el nombre de vèrtexs i arestes de G és finit, aquest circuit sempre es formarà. No ens podem quedar encallats en un vèrtex abans d'haver completat el circuit perquè com que tots els vèrtexs són parells cada cop que n'entrem un també en podem sortir, excepte possiblement si retornem al vèrtex inicial. Si C_1 passa per totes les arestes de G, hem acabat. Si no, podem triar un vèrtex v de C_1 tal que C_1 no contingui totes les arestes adjacents a v.

A continuació, eliminem totes les arestes que formen C_1 de G. Obtenim un graf en què un altre cop tots els vèrtexs són parells. Començant a v, tornem a iniciar un

camí fins que es formi un altre circuit C_2. Llavors, podem unir C_1 i C_2 en un circuit dins de G. Si encara queda part del graf sense recórrer, podem seguir amb el mateix procediment. Traiem les arestes de C_1 i C_2 del graf, tornem a buscar un altre circuit i repetim fins que el circuit format passi per totes les arestes de G. Quan això passi haurem passat per totes les arestes de G un sol cop, demostrant així que el graf G conté un circuit eulerià. □

Per acabar d'entendre la demostració anterior, en les figures que hi ha a continuació es pot veure il·lustrat el procediment amb un graf d'exemple.

En la següent figura 2.5 tenim un graf G format només per vèrtexs parells, condició que com hem demostrat és necessària per obtenir un circuit eulerià.

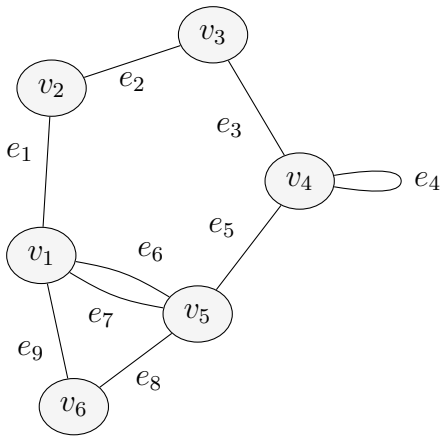

Figura 2.5: Diagrama del graf G el qual conté únicament vèrtexs parells.

Podem començar un camí $P_1 : v_1 \overset{e_1}{—} v_2 \overset{e_2}{—} v_3 \overset{e_3}{—} v_4 \overset{e_4}{—} v_4$, el qual al final té un circuit $C_1 : v_4 \overset{e_4}{—} v_4$. Eliminem C_1 de G i obtenim el graf de la figura 2.6.

Ara hem d'agafar un vèrtex v de C_1 tal que C_1 no contingui totes les arestes adjacents a v. L'única opció que tenim és agafar a v_4 com a v. Comencem de nou des de v, i fins que ens trobem que hem repetit un vèrtex, a fer un camí, per exemple $P_2 : v_4 \overset{e_5}{—} v_5 \overset{e_6}{—} v_1 \overset{e_7}{—} v_5$. Ens aturem perquè hem repetit el vèrtex v_5, és a dir, tenim un altre circuit $C_2 : v_5 \overset{e_6}{—} v_1 \overset{e_7}{—} v_5$. Eliminem C_1 i C_2 de G i ens queda el graf de la figura 2.7.

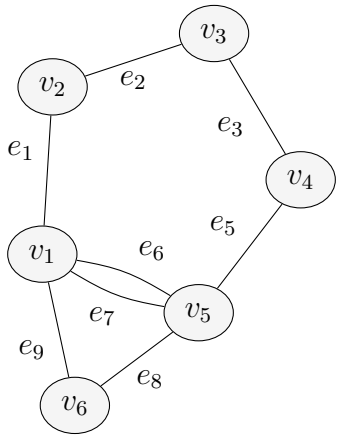

Figura 2.6: G sense les arestes de C_1.

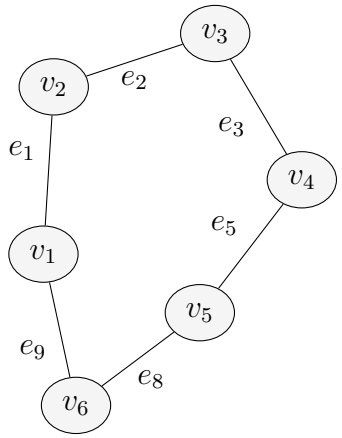

Figura 2.7: G sense les arestes de C_1 i C_2.

Tornem a agafar un vèrtex v de C_2 tal que C_2 no contingui totes les arestes adjacents a v. Podríem agafar v com a v_5 o v_1, agafem v_1. Tornem a fer un camí $P_3 : v_1 \overset{e_1}{—} v_2 \overset{e_2}{—} v_3 \overset{e_3}{—} v_4 \overset{e_5}{—} v_5 \overset{e_8}{—} v_6 \overset{e_9}{—} v_1$. Com que tornem a repetir un vèrtex, v_1, veiem que hem obtingut un altre cop un circuit $C_3 : v_1 \overset{e_1}{—} v_2 \overset{e_2}{—} v_3 \overset{e_3}{—} v_4 \overset{e_5}{—} v_5 \overset{e_8}{—} v_6 \overset{e_9}{—} v_1$. Com és fàcil de veure, el circuit C_3, que és el mateix que el camí P_3, passa per totes les arestes del graf.

Finalment, si eliminem totes les arestes dels circuits C_1, C_2 i C_3 del graf G ens queda un graf sense arestes, és a dir, haurem passat per totes les arestes un sol cop, fent possible que hi hagi un circuit eulerià.

Si unim C_1, C_2 i C_3 reordenant les arestes perquè quedi un camí continu podem obtenir el circuit eulerià $C_E : v_1 \overset{e_1}{—} v_2 \overset{e_1}{—} v_3 \overset{e_3}{—} v_4 \overset{e_4}{—} v_4 \overset{e_5}{—} v_5 \overset{e_6}{—} v_1 \overset{e_7}{—} v_5 \overset{e_8}{—} v_6 \overset{e_9}{—} v_1$.

Com a conseqüència directa del teorema 1, obtenim el següent teorema.

Teorema 2. *Un graf G té un camí eulerià que comença en el vèrtex v_i i acaba en un vèrtex diferent v_f si i només si v_i i v_f són senars, i la resta de vèrtexs de G són parells.*

Demostració.
($\Longrightarrow$): Afegim una aresta que uneixi v_i i v_f, aleshores tenim un graf nou que, pel teorema 1, tindrà un circuit eulerià, ja que tots els seus vèrtexs seran parells. Per tant, en G tots els vèrtexs són parells excepte v_i i v_f, que han de ser senars.

($\Longleftarrow$): Ara suposem que tots els vèrtexs de G són parells menys v_i i v_f. Si unim v_i i v_f per una aresta, aleshores tots els vèrtexs del nou graf són parells i pel teorema 1 aquest graf conté un circuit eulerià, que necessàriament ha de passar per l'aresta que uneix v_i i v_f. Si eliminem l'aresta que uneix v_i i v_f, el circuit eulerià es converteix en un camí eulerià. Per tant, G té un camí eulerià. □

Per exemplificar la demostració anterior, utilitzem el graf G de la figura 2.8, que conté exactament dos vèrtexs senars, $v_i = v_1$ i $v_f = v_4$.

Si afegim una aresta e_6 que uneix v_i amb v_f obtenim el graf H de la figura 2.9.

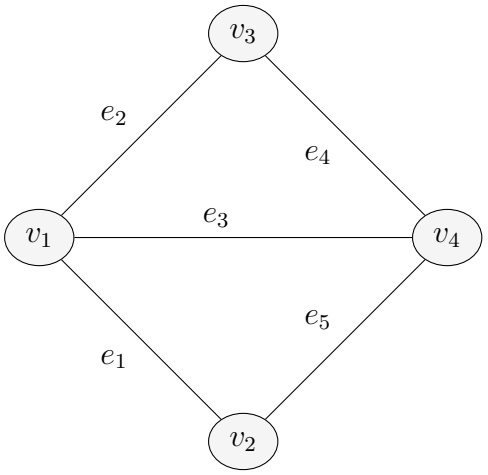

Figura 2.8: Diagrama del graf G.

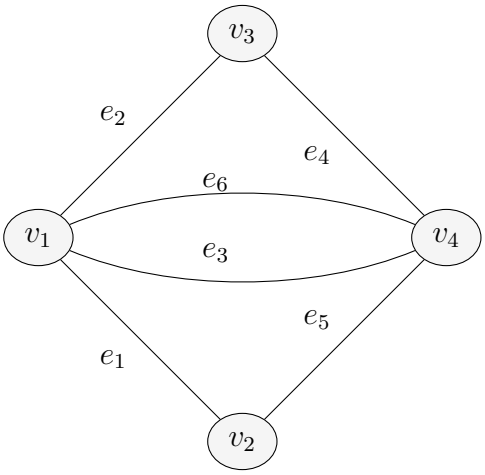

Figura 2.9: Diagrama del graf H.

Com que la nova aresta e_6 incideix en v_1 i v_4, el grau de cadascun dels dos vèrtexs incrementa en un, la qual cosa fa que passin a ser parells. Com segueix del teorema 1, és possible crear un circuit eulerià en un graf on tots els vèrtexs són parells com en el graf H. És fàcil trobar en el graf H un circuit eulerià, per exemple $C_E : v_1 \overset{e_1}{—} v_2 \overset{e_5}{—} v_4 \overset{e_4}{—} v_3 \overset{e_2}{—} v_1 \overset{e_3}{—} v_4 \overset{e_6}{—} v_1$. Si ara volem obtenir el camí eulerià P_E que està en el graf G, hem de treure l'aresta e_6 (i, per tant, de C_E també el vèrtex v_1 del final el qual l'aresta e_6 connecta) que havíem afegit abans i ens queda que en el graf G hi ha el camí eulerià $P_E : v_1 \overset{e_1}{—} v_2 \overset{e_5}{—} v_4 \overset{e_4}{—} v_3 \overset{e_2}{—} v_1 \overset{e_3}{—} v_4$. Com podem veure P_E no és un circuit perquè comença en els vèrtexs v_i i v_f que són diferents.

El graf G precisament representa la situació de Königsberg després dels bombardejos, en què només hi havia cinc ponts. El graf G i el camí obtingut P_E aquí es poden comparar amb la figura 1.4 del primer capítol.

2.3.2 Camins hamiltonians

Un *camí hamiltonià* és un camí que passa per tots els vèrtexs d'un graf un sol cop (a diferència d'un eulerià, que passa per totes les arestes). Si un camí hamiltonià és un cicle, s'anomena *cicle hamiltonià*.

Si un graf conté un camí hamiltonià, s'anomena *graf traçable*. Si conté un cicle hamiltonià, s'anomena *graf hamiltonià*.

Aquests grafs, camins i cicles s'anomenen així en honor al conegut matemàtic William Rowan Hamilton, que va inventar un joc matemàtic que consistia en trobar una manera de traçar un camí que comencés i acabés en el mateix vèrtex i passés per

tots els vèrtexs d'un dodecaedre (un poliedre format per dotze cares pentagonals) un sol cop. És a dir, consistia en trobar un cicle hamiltonià.

Les arestes i vèrtexs d'un dodecaedre es poden representar en forma de graf, com es mostra en la figura 2.10. Aquest graf s'anomena *graf dodecaèdric*.

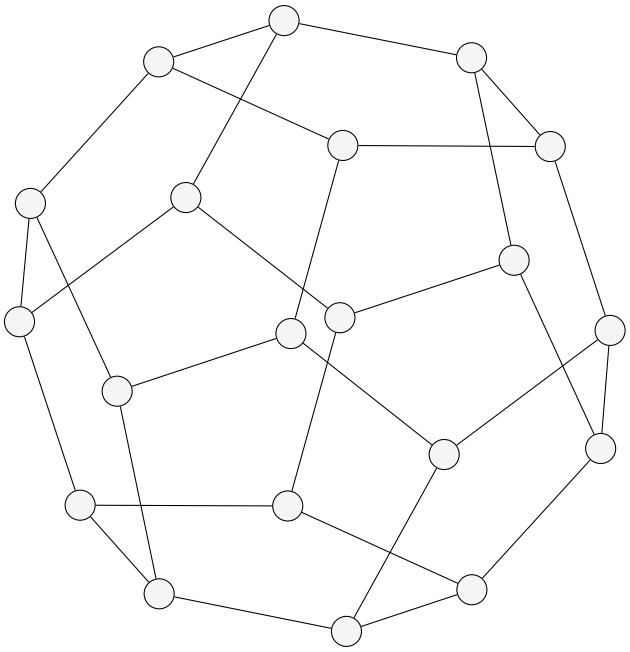

Figura 2.10: Graf dodecaèdric.

Evidentment, el graf dodecaèdric és hamiltonià, per això Hamilton va dissenyar el joc. Tot i això, determinar si un graf és hamiltonià, és molt més difícil que determinar si és eulerià, i no es coneix una bona caracterització dels grafs que ho són. A més, tampoc es coneix cap algorisme que sigui ràpid que pugui determinar si un graf és hamiltonià o no.

2.4 Arbres

Si un graf és acíclic, s'anomena *bosc*. S'anomena *arbre* a un graf acíclic i connex. Cada component d'un bosc és un arbre.

Un vèrtex d'un arbre de grau u s'anomena *fulla*. L'arbre de la figura 2.11 té quatre fulles: els vèrtexs v_5, v_3, v_7 i v_8.

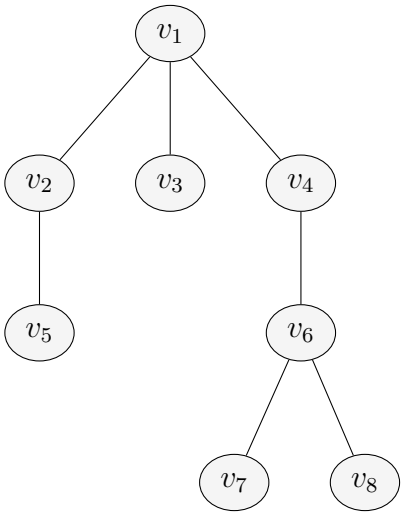

Figura 2.11: Un arbre.

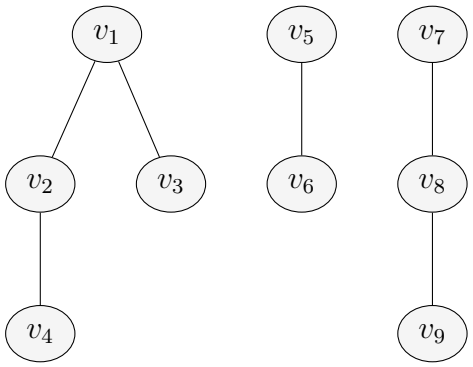

Figura 2.12: Un bosc.

Els següents teoremes recullen diverses propietats importants sobre arbres.

Teorema 3. *Un graf G és un arbre si i només si hi ha exactament un camí entre dos vèrtexs qualssevol.*

Demostració.
($\Longrightarrow$): Sigui G un arbre. Com que G és un arbre, G és connex i, per tant, hi ha almenys un camí entre cada parell de vèrtexs. Suposem que hi ha dos camins diferents entre dos vèrtexs u i v de G. La unió d'aquests dos camins forma un cicle, fet que contradiu que G sigui un arbre; per tant, hi ha exactament un camí entre cada parell de vèrtexs d'un arbre.

($\Longleftarrow$): Per altra banda, sigui G un graf en què hi ha exactament un camí entre cada parell de vèrtexs; per tant, G és connex. G no té cap cicle perquè si tingués un cicle, per exemple entre els vèrtexs u i v, llavors hi hauria dos camins diferents entre u i v, la qual cosa és una contradicció. És a dir, G és connex i no té cap cicle; per tant, és un arbre. □

Teorema 4. *Si un arbre té dos o més vèrtexs, aleshores té almenys dues fulles.*

Demostració. Considerem qualsevol camí P de longitud màxima en l'arbre. Els vèrtexs u i v, extrems del camí, han d'acabar en fulles. De fet, si un dels extrems, per exemple v, no fos una fulla, voldria dir que podríem estendre P per l'aresta adjacent a v que encara no forma part de P, és a dir, aquest no seria el camí de longitud màxima. □

Teorema 5. *Si un graf $G = (V, E)$ és un arbre, llavors $|E| = |V| - 1$.*

Demostració. Fem servir inducció per provar el teorema.

Per un arbre amb $n = 1$ vèrtexs, l'afirmació és clarament correcta ja que un graf acíclic connex amb un sol vèrtex té $n - 1 = 1 - 1 = 0$ arestes.

Ara suposem que el teorema és cert per arbres amb n vèrtexs. Vegem-ho per arbres amb $n+1$ vèrtexs.

Sigui G un arbre amb $n+1$ vèrtexs. Anomenem G' l'arbre que s'obté si traiem una de les fulles de G (el teorema anterior assegura l'existència d'almenys dues fulles) i l'única aresta adjacent a la fulla treta. L'arbre G' obtingut té n vèrtexs; per tant té exactament $n-1$ arestes. Aleshores G té $n-1+1=n$ arestes i es verifica doncs que per qualsevol arbre $G=(V,E)$, $|E|=|V|-1$. □

Un *arbre d'expansió* E és un subgraf d'un graf connex G el qual és un arbre que conté tots els vèrtexs de G.

La figura 2.13 mostra un graf G i la figura 2.14 mostra l'arbre d'expansió E d'aquest graf G.

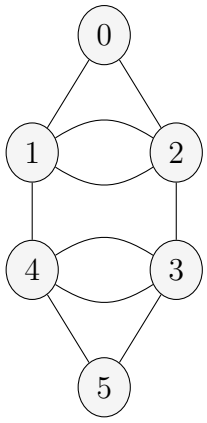

Figura 2.13: Un graf G.

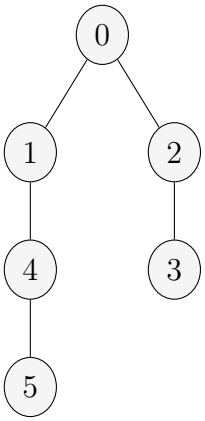

Figura 2.14: Arbre d'expansió E del graf G.

2.5 Grafs dirigits

Un *graf dirigit* es defineix de forma similar que un graf normal, amb l'excepció que cada aresta del graf consisteix en un parell ordenat de vèrtexs (a diferència d'un graf normal, en què l'ordre del parell de vèrtexs que formen una aresta no importa). Això vol dir que en un graf dirigit les arestes només es poden recórrer en un sentit determinat.

2.5.1 Representació visual

Per representar un graf dirigit es fan servir fletxes en comptes de línies. Les fletxes assenyalen la direcció[3] de l'aresta, establerta per l'ordre del parell de vèrtexs que la descriu.

[3]En la teoria de grafs es poden utilitzar les paraules *direcció* i *sentit* per dir el mateix.

Per exemple, el diagrama del graf dirigit

$$D = (\{v_1, v_2, v_3, v_4\}, \{(v_1, v_2), (v_2, v_1), (v_2, v_2), (v_1, v_3), (v_1, v_3), (v_3, v_4)\})$$

és el següent:

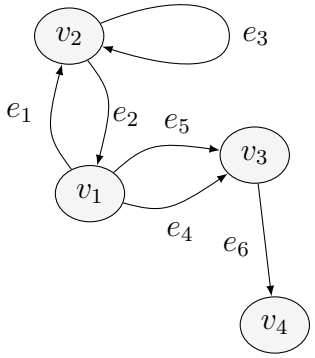

Figura 2.15: Diagrama del graf dirigit D.

2.5.2 Definicions

En el graf dirigit D de la figura 2.15 observem que $e_1 \neq e_2$ perquè l'ordre dels parells de vèrtexs importa. En aquest cas, en què dues arestes connecten els mateixos dos vèrtexs diferents en sentits oposats, es diu que són arestes *antiparal·leles*. En canvi, sí que es compleix que $e_5 = e_4$. En aquest últim cas, en què dues arestes són iguals perquè connecten dos vèrtexs diferents en el mateix sentit, podem dir que són arestes *paral·leles*.

L'aresta e_3 és un *bucle* perquè connecta el vèrtex amb si mateix.

En els grafs dirigits els vèrtexs tenen *grau entrant*, $g_e(v)$, el nombre d'arestes que arriben al vèrtex, i *grau sortint*, $g_s(v)$, el nombre d'arestes que surten del vèrtex. Per exemple, $g_e(v_3) = 2$ i $g_s(v_3) = 1$

En general, en un graf dirigit no sempre és possible crear un camí des de qualsevol vèrtex a qualsevol altre. Per exemple, en el graf dirigit D no podem anar de v_4 a v_3 perquè l'única aresta de v_4 és entrant, és a dir, si ens trobem al vèrtex v_4, no podem anar a cap altre lloc.

Si en un graf dirigit sí que és possible fer un camí que vagi des de qualsevol vèrtex a qualsevol altre, s'anomena graf dirigit *fortament connex*. Un exemple de graf dirigit fortament connex és el graf E de la figura 2.16.

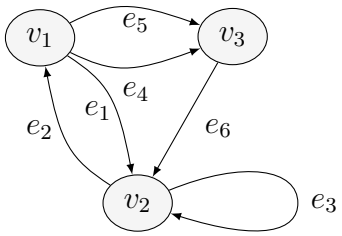

Figura 2.16: Diagrama del graf dirigit fortament connex E.

2.6 Grafs ponderats

Quan volem utilitzar grafs per modelar problemes del món real molts cops necessitem tenir en compte altres factors, com és el cas del cost associat en passar per una aresta. Per això podem utilitzar *grafs ponderats*.

Un graf ponderat consisteix en un graf G, el qual té associat un nombre real $p(e)$ a cada aresta, anomenat *pes de l'aresta*. La suma dels pesos de totes les arestes de G es denota $p(G)$ i s'anomena *pes del graf*. Si un camí P està format per arestes ponderades, podem denotar *el pes del camí* (la suma dels pesos de totes les arestes que formen el camí) amb $p(P)$.

En els següents capítols veurem diverses aplicacions d'aquest tipus de graf.

2.7 Tractament informàtic dels grafs

Per la gran majoria de les aplicacions dels grafs s'utilitzen grafs molt grans i processats automàticament. Per això, hi ha la necessitat de poder representar un graf de manera informàtica en la memòria de l'ordinador.

Per representar un graf s'utilitza una estructura de dades bastant simple anomenada *llista d'adjacència*. La llista d'adjacència Adj d'un graf $G = (V, E)$ consisteix en una llista de $|V|$ llistes, una per cada vèrtex de V. Per cada vèrtex $u \in V$, la llista $Adj[u]$ conté tots els vèrtexs v tals que hi ha una aresta $(u, v) \in E$. És a dir, $Adj[u]$ consisteix en tots els vèrtexs adjacents al vèrtex u en el graf G.

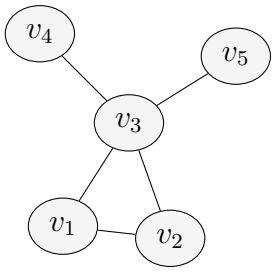

Figura 2.17: Diagrama del graf G.

```
Adj = {
       {2, 3},
       {1, 3},
       {1, 2, 4, 5},
       {3},
       {3}
      }
```

Figura 2.18: Llista d'adjacència del graf G.

En cas que el graf sigui dirigit la definició anterior també es manté.

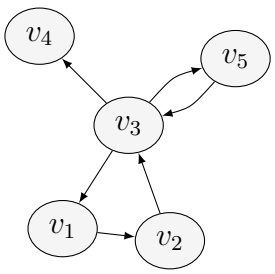

Figura 2.19: Diagrama del graf dirigit D.

```
Adj = {
       {2},
       {3},
       {1, 4, 5},
       {},
       {3}
      }
```

Figura 2.20: Llista d'adjacència del graf dirigit D.

També podem adaptar la llista d'adjacència pels grafs ponderats, és a dir, un graf on cada aresta té un pes associat, donat per una funció de pes $p : E \to \mathbb{R}$. Per exemple, en un graf ponderat $G = (V, E)$ amb funció de pes p, només hem de guardar el pes $p(u, v)$ de l'aresta $(u, v) \in E$ amb el vèrtex v en la llista d'adjacència del vèrtex u.

Existeixen altres estructures de dades per representar grafs, però en aquest treball només utilitzarem les llistes d'adjacència.

"Cal considerar tres coses: problemes, teoremes i aplicacions."
— Gottfried Wilhelm Leibniz, *Dissertació sobre l'art combinatori*, 1666

3

Aplicacions dels grafs

Els grafs són molt útils per modelar connexions i relacions del món real, fet que fa que s'utilitzin en molts camps de la ciència i l'enginyeria. En les següents seccions s'expliquen algunes aplicacions destacables i diverses dels grafs.

3.1 Cerca web

Amb l'enorme quantitat d'informació disponible a la xarxa, trobar alguna cosa que necessitem seria gairebé impossible sense eines que ordenen aquesta informació. Els motors de cerca web estan dissenyats per ordenar bilions de pàgines web per trobar els resultats més rellevants i útils en tan sols una fracció de segon. Alguns exemples de motors de cerca web són Google, Bing, Baidu, Yahoo! o DuckDuckGo.

Aquestes eines, més enllà de fer coincidir les paraules del terme de cerca amb documents rellevants al web, també intenten prioritzar les fonts més fiables disponibles. Per fer això, aquests serveis utilitzen algorismes dissenyats per identificar senyals que poden ajudar a determinar quines pàgines demostren domini, autenticitat i fiabilitat en un tema determinat. Un dels elements utilitzats per aconseguir això és mirar si altres llocs web destacats enllacen amb la pàgina que s'està considerant. En el cas de Google, el motor de cerca més utilitzat del món, aquesta tasca s'aconsegueix a través de l'algorisme *PageRank*[1].

El resultat donat per PageRank és el resultat d'un algorisme matemàtic que processa un graf web. Un graf web és un tipus de graf que consisteix en un graf dirigit on els vèrtexs consisteixen en pàgines web i les arestes són els enllaços d'una pàgina a una altra. Per exemple, si considerem una part d'Internet formada només per les pàgines X, Y i Z, i sabem que X i Y tenen un enllaç a Z i que Z enllaça amb Y, el graf web seria el següent graf W de la figura 3.1.

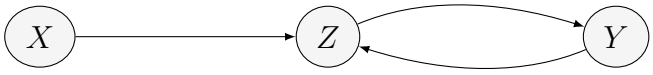

Figura 3.1: Graf web W de les pàgines X, Y i Z.

[1] Actualment, PageRank no és l'únic algorisme utilitzat per Google per ordenar resultats de cerca, però és el primer que l'empresa va fer servir i el més conegut.

El rànquing d'una pàgina es defineix recursivament i depèn del rànquing de totes les pàgines que l'enllacen. Això aconsegueix dues coses: la primera, és que considera que si una pàgina web important enllaça amb una pàgina, aquesta pàgina haurà de rebre un rànquing millor que si l'enllacés una pàgina amb un rànquing més baix i, la segona, és que fa que sigui més difícil manipular el sistema creant moltes pàgines web amb tan sols enllaços per influenciar a l'algorisme a fer pujar en el rànquing una pàgina web que realment no és important.

L'ús dels grafs en els motors de cerca va ser un dels factors claus pel triomf de Google sobre els altres serveis, ja que abans que es fundés Google el 1998, els motors de cerca més populars utilitzaven principalment dos mètodes. El primer consistia en llistes mantingudes per humans que incloïen temes populars de forma eficaç, però eren subjectives, costoses de construir i mantenir, lentes de millorar i no podien cobrir tots els temes més específics i amb menys interès general. Aquest mètode, a més, seria impossible actualment amb el creixement massiu d'Internet que hi ha hagut en els últims vint anys. El segon mètode consistia en motors de cerca automatitzats que es basaven en la coincidència de paraules clau del terme de cerca en pàgines web. Aquest mètode solia retornar molt pocs resultats i de baixa qualitat. A més, eren fàcils de manipular a partir de la inclusió de termes populars en les pàgines, encara que no estiguessin relacionats amb els continguts dels quals realment tractava.

3.2 Xarxes socials

A part dels motors de cerca, un dels tipus de pàgines web més utilitzades són les xarxes socials com Facebook, Instagram o Twitter. Aquests serveis també utilitzen grafs, en aquest cas per representar les relacions entre els usuaris. Un usuari es representa com un vèrtex d'un graf i una aresta representa un amistat o un seguiment.

Depenent de la xarxa social, el graf pot ser dirigit o no. Per exemple, en el cas de Facebook, perquè un usuari A sigui amic d'un altre usuari B, primer A ha d'enviar-li una sol·licitud d'amistat a B i després B l'ha d'acceptar. Si B accepta la sol·licitud, A i B es converteixen en amics mutus, és a dir, A és amic de B i B és amic d'A. Això es representaria com un graf no dirigit. En el cas d'Instagram o Twitter, un usuari A pot seguir a qualsevol altre usuari B sense que l'usuari B hagi de seguir a l'usuari A, és a dir, es tractaria d'un graf dirigit.

Gràcies a la utilització dels grafs, els sistemes utilitzats per implementar les xarxes socials són més simples i més fàcils de processar per algorismes que llavors són usats per fer suggeriments rellevants i personalitzats de publicacions, altres usuaris, anuncis, etc. Això fa que l'usuari tingui una millor experiència i, per tant, també utilitzi el servei més i més, creant així més ingressos a l'empresa propietària de la xarxa social.

Una de les maneres per trobar coses noves que puguin interessar a un usuari, és mirar què li interessa als amics d'aquest usuari. Per exemple, tenim la xarxa social representada en el graf de la figura 3.2 i volem fer un suggeriment a l'usuari A. Com que la majoria d'amics d'A són amics de l'usuari X, és probable que A també estigui interessat en X.

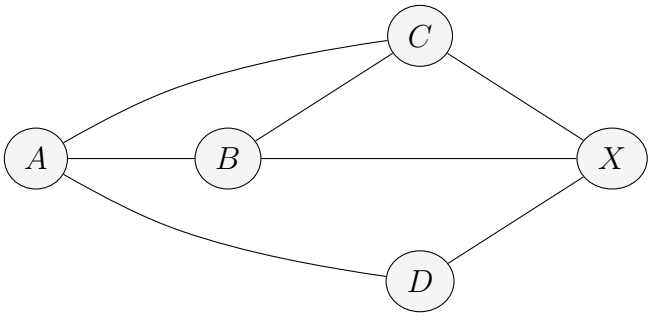

Figura 3.2: Graf de la xarxa social formada pels usuaris A, B, C, D i X.

Per aconseguir veure què interessa a l'usuari A, s'analitzen els vèrtexs adjacents dels vèrtexs que representen els amics de l'usuari A. Veiem que el vèrtex A està connectat amb els vèrtexs B, C i D, és a dir, són amics. Llavors el que hem de fer és analitzar els vèrtexs adjacents d'aquests amics. A partir d'això trobem que els tres amics d'A també són amics de l'usuari X; per tant, probablement l'usuari A, també voldrà ser amic de l'usuari X i viceversa.

La teoria de grafs també s'utilitza en eines que analitzen xarxes socials per fer estudis sociològics, per exemple, per mesurar el prestigi i la influència d'una persona, explorar la propagació de notícies o rumors, etc.

3.3 Conectòmica

Una de les aplicacions més recents dels grafs és en l'estudi de les connexions cerebrals. És un vell somni crear un *graf cerebral* en què els vèrtexs són les neurones i les arestes les connexions entre elles. El graf cerebral del cuc *Caenorhabditis elegans*, que només té 302 neurones, es va aconseguir crear fa més de 30 anys. Tot i això, els grafs cerebrals d'animals una mica més complexos com d'una mosca, les quals tenen unes 100.000 neurones, segueixen sense haver-se pogut crear, encara que s'hi han invertit molts recursos i esforços arreu del món. La creació del graf cerebral d'un cervell humà a nivell neuronal avui en dia és inassolible, principalment perquè, de mitjana, el cervell humà té 86.000 milions de neurones.

Tanmateix, construir grafs cerebrals (o *conectomes*) en què els vèrtexs no són neurones individuals, sinó àrees molt més grans del cervell anomenades *regions d'interès* és possible i actualment és objecte de moltes investigacions. Dos vèrtexs, corresponents a regions d'interès, són connectats amb una aresta si al processar una imatge de ressonància magnètica del cervell es troben trams de fibres neuronals entre aquestes dues àrees. A la figura 3.3 es pot observar una imatge de ressonància magnètica d'un cervell humà que ha estat processada per obtenir-ne el seu conectoma.

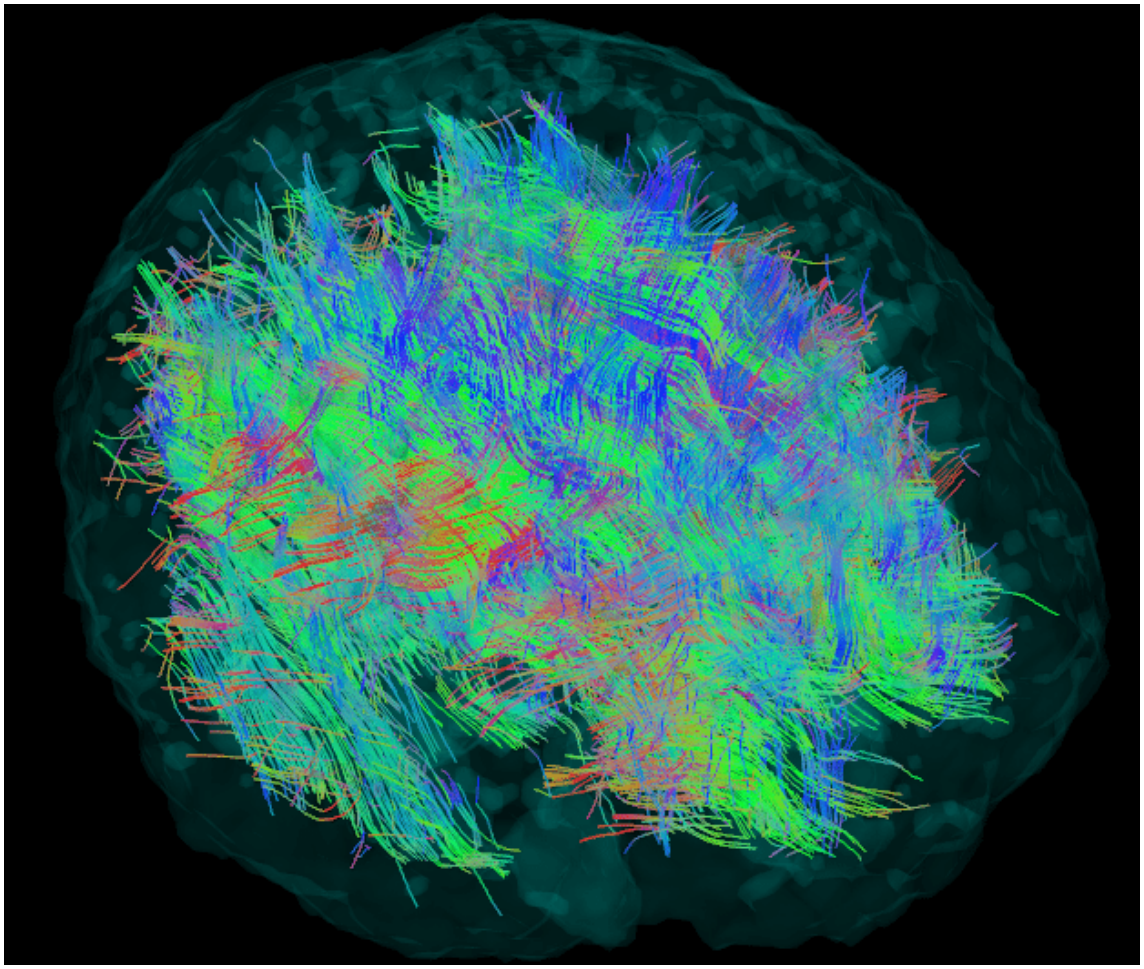

Figura 3.3: Imatge de ressonància magnètica d'un cervell humà que ha estat processada per obtenir-ne el seu conectoma.

La *conectòmica*, l'estudi dels conectomes, està permetent avanços importants en la investigació de malalties cognitives com la dislèxia, l'autisme, l'esquizofrènia, el Parkinson i l'Alzheimer. Un dels símptomes principals d'aquestes malalties és que es redueix la connectivitat (bàsicament el nombre d'arestes) del conectoma.

Per exemple, en el cas de l'Alzheimer, l'any 2018 un estudi de l'IDIBAPS, de Barcelona, va demostrar que a partir d'un graf cerebral es podia detectar de forma precoç deficiències en la connectivitat cerebral abans que apareguin els primers símptomes de la malaltia d'Alzheimer.

Altres estudis de conectòmica també han obtingut resultats interessants. Per exemple, s'han comparat diversos conectomes de dones i homes sans i s'ha trobat que els conectomes des les dones estan significativament més ben connectats que els dels homes.

3.4 Microxips i sistemes de distribució

A l'hora de dissenyar un circuit integrat d'un microxip ens interessa minimitzar la quantitat de cablejat que fem servir. Evidentment, com menys material fem servir, més barata serà la fabricació. A més, el circuit també serà més ràpid, ja que l'electricitat no haurà de recórrer tanta distància. Per tot això serà important minimitzar la longitud dels conductors utilitzats.

Per aconseguir-ho, podem fer servir la teoria de grafs. En primer lloc representarem cada pin que volem connectar amb un vèrtex. Llavors, ens faltarà connectar els vèrtexs amb arestes, formant un graf en què cada vèrtex estigui connectat amb la resta perquè formin una xarxa connectada entre si. Aquest graf en què cada vèrtex està connectat directament amb una aresta amb la resta de vèrtexs s'anomena *graf complet*.

Ara el problema rau en quines arestes fem servir per connectar els vèrtexs. El primer que ens interessaria, és que a cada vèrtex només s'hi pogués arribar per un camí, per així no tenir camins repetits i, per tant, innecessaris, els quals gastin més material de fabricació i espai. Per això, podem arribar a la conclusió que necessitarem crear un arbre, ja que crearem un graf en què entre cada vèrtex només hi haurà un sol camí.

Tanmateix, es poden trobar molts arbres que connectin tots els vèrtexs, però ens interessa el que utilitzi menys cablejat. Per això, podem representar la longitud dels cables amb arestes ponderades. Aleshores, el problema que hem de solucionar és trobar un arbre ponderat que connecti tots els vèrtexs i tingui el menor pes possible.

En la teoria de grafs, un arbre així s'anomena *arbre d'expansió de pes mínim*, ja que és un arbre que passa per tots els vèrtexs, és a dir, és un arbre d'expansió, i té el menor pes possible. Existeixen diversos algorismes, com l'algorisme de Prim o el de Kruskal, que donats un graf complet van traient arestes fins formar un arbre d'expansió de pes mínim d'un graf.

Una situació semblant ens la podem trobar en alguns sistemes de distribució de recursos com aigua, gas, electricitat, etc., en què ens podria interessar utilitzar el mínim material en la construcció. Tot i això, normalment en sistemes de distribució majors això no ens interessa, ja que sol ser més important que el sistema segueixi funcionant si hi ha una avaria (recordem que si es treu un aresta d'un arbre, el graf deixa de ser connex), que no pas el cost de construcció o l'espai que ocupa. Per avaluar com de robusta és una xarxa de distribució i trobar punts febles també s'utilitza la teoria de grafs i altres algorismes més complexos.

3.5 Xarxes de transport

Una altra aplicació dels grafs és la de la modelització de molts tipus de xarxes de transport, com xarxes ferroviàries, aèries, marítimes i de carreteres.

En el cas de les xarxes ferroviàries, aèries i marítimes, podem representar una xarxa

en forma de graf on els vèrtexs són els ports o estacions i les arestes les rutes que es fan entre vèrtex i vèrtex. Normalment s'utilitza un graf dirigit, ja que els mitjans de transport van en una sola direcció. En el cas que una línia de transport tingui rutes en les dues direccions, podem utilitzar dues arestes dirigides antiparal·leles.

En el cas de les xarxes de carreteres, és una mica més complicat perquè no hi ha parades establertes en llocs en concret. Per això, si volem representar una xarxa que inclogui tots els tipus de carreteres, es fa utilitzant les arestes per les carreteres i els vèrtexs per les interseccions entre carreteres. Si un carrer és d'una sola direcció, utilitzarem una sola aresta dirigida en la direcció del carrer. En el cas que sigui de dues direccions, utilitzarem dues arestes dirigides antiparal·leles.

Normalment s'utilitzen grafs dirigits i ponderats, on el pes de cada aresta és la distància entre els seus extrems o el temps que es tarda a recórrer aquesta distància.

A partir d'un graf d'una xarxa de transport, podem obtenir molta informació fent servir diversos algorismes. Una de les dades més útils que podem aconseguir és el camí més curt entre dos vèrtexs. Aquesta informació clarament és de gran utilitat per la majoria de persones del món, ja que tots ens hem de desplaçar d'algun lloc a un altre en algun moment i no sempre sabem com arribar-hi. A partir dels algorismes que troben el camí més curt, no només podem aconseguir saber quina ruta fer per arribar a un lloc determinat, sinó que també aconseguim saber quina, de totes les rutes possibles, és la més curta. Passar pel camí més curt també té els avantatges evidents de ser normalment més ràpid que altres camins possibles, més barat i més sostenible amb el medi ambient. Exemples de programes que duen a terme aquesta tasca, anomenats planificador de rutes, són els que van incorporats en els sistemes GPS dels cotxes o aplicacions com Google Maps, Bing Maps o Yandex.Maps.

Diversos algorismes per buscar el camí més curt entre dos vèrtexs seran explicats en el capítol 4 i implementats en el capítol 5 per desenvolupar un planificador de rutes pel meu poble. En el capítol 6, també compararem el planificador de rutes desenvolupat amb el Google Maps.

En el transport, també s'utilitza la teoria de grafs per problemes encara més complexos, com el problema del viatjant ambulant o el problema del carter xinès.

Per una banda, el problema del viatjant ambulant consisteix en un viatjant ambulant que vol sortir de casa seva, anar a vendre a uns quants pobles i tornar a casa. Donats els temps que es tarda a viatjar entre cada poble, quin seria el millor itinerari per visitar cada poble un cop tardant el mínim de temps possible? Amb termes de la teoria de graf, això ho podem expressar si representem els pobles i la casa del venedor amb vèrtexs d'un graf. Llavors, donat un graf complet, on el pes de les arestes és el pes del camí més curt[2] entre cada poble, l'objectiu és trobar un cicle hamiltonià de pes mínim en el graf.

Els algorismes que solucionen aquest problema tenen una aplicació clara a la realitat en dissenyar les rutes per una empresa de repartiment de paquets. Tanmateix,

[2]Prèviament ja hem de saber quin és el camí més ràpid entre cada poble. Si no els sabem, podem fer servir els algorismes per trobar el camí més curt que també es basen en la teoria de grafs.

existeixen aplicacions d'aquest problema més complexes com en el disseny de circuits impresos (PCBs), l'anàlisi d'estructures cristal·lines a través de rajos X i més.

Per altra banda, el problema del carter xinès[3], consisteix en un carter que surt de l'oficina de correus, passa per tots els carrers del poble per repartir les cartes i torna de nou a l'oficina de correus. El carter vol caminar el mínim possible per acabar abans i no cansar-se tant. En teoria de grafs, si representem cada carrer amb una aresta ponderada, on el pes representi el temps que es tarda a transcórrer el carrer, podem expressar el problema com el que, donat un graf ponderat, troba un cicle eulerià de pes mínim.

Aquest problema té aplicacions en la neteja de carrers, la recollida d'escombraries, el repartiment de cartes que s'han de fer arribar a tothom, conduir una llevaneu...

[3]S'anomena xinès perquè el problema va ser estudiat originalment pel matemàtic xinès Kwan Mei-Ko, i quan es van traduir els seus estudis es va posar xinès en honor seu.

"Quin és el camí més curt per viatjar de Rotterdam a Groningen? Ho podem saber amb l'algorisme per al camí més curt, que vaig dissenyar en uns vint minuts. Un matí feia compres a Amsterdam amb la meva jove promesa i, cansats, ens vam asseure a la terrassa d'una cafeteria per beure una tassa de cafè i només pensava si podia fer-lo i llavors vaig dissenyar l'algorisme per al camí més curt. Com he dit, va ser una invenció de vint minuts."

— Edsger W. Dijkstra, pioner en ciències de la computació i dissenyador de l'algorisme homònim per trobar al camí més curt.

4

Buscant el camí més curt

Un camí P^* en un graf G que comença en un vèrtex inicial i i acaba en un vèrtex final f s'anomena *el camí més curt* si no hi ha cap altre camí P' des de i fins a f tal que $l(P') < l(P^*)$. La *distància* $d(i, f)$ des de i fins a f en G és la longitud del camí més curt des de i fins a f o ∞ si no hi ha cap camí que vagi des de i fins a f.

En el cas dels grafs ponderats, el camí més curt P^* és el camí que va des de i fins a f de manera que no hi ha cap altre camí P' tal que $p(P') < p(P^*)$, és a dir, en comptes d'estar interessats en la longitud del camí en termes del nombre d'arestes pel que passa, ens interessa el pes del camí.

Una manera possible de trobar el camí més curt entre dos punts seria enumerar totes les rutes possibles, calcular-ne la distància que recorren i agafar el camí amb la distància més curta. Tot i això, és fàcil veure que hauríem d'examinar una enorme quantitat de possibilitats, la majoria de les quals no valdria la pena considerar-les. Per exemple, una ruta des de Vic fins a Barcelona que passi per París és possible, però òbviament és una mala elecció si el que volem és passar pel camí més curt.

Aquest capítol se centra en alguns dels algorismes que es fan servir per trobar el camí més curt en grafs no ponderats i ponderats de manera eficient.

4.1 Algorisme de cerca en amplada

L'algorisme de cerca en amplada, també anomenat BFS, de l'anglès *Breadth-first search*, va ser descobert l'any 1959 per Edward F. Moore, que el va utilitzar per trobar el camí més curt a través d'un laberint. Aquest algorisme és segurament el més utilitzat per buscar el camí més curt en un graf no ponderat, ja que és eficient i simple.

4.1.1 Procediment

L'algorisme no troba directament el camí més curt entre un vèrtex inicial i i un vèrtex final f, sinó que troba el camí més curt des de i a tots els altres vèrtexs accessibles des de i. L'execució acaba quan tots els vèrtexs accessibles des de i han estat processats. Llavors si f és un vèrtex accessible des de i, en podrem trobar el camí més curt.

L'algorisme comença des del vèrtex inicial i i explora tots els vèrtexs a distància 1, és a dir, els vèrtexs adjacents a l'inicial. Llavors continua amb els vèrtexs a distància 2, i va fent així fins que tots els vèrtexs accessibles des del vèrtex inicial han estat explorats. Durant l'execució a cada vèrtex v se li assigna una distància $d(i, v)$, i un predecessor, el vèrtex per al qual s'ha arribat a v. El predecessor del vèrtex inicial i serà nul, per indicar que no té predecessor, ja que és per on es comença.

En primer lloc, l'algorisme crea una llista on guardarà la distància a cada vèrtex des del vèrtex inicial. Com que al principi de tot encara no se saben aquestes distàncies, la llista s'inicialitza amb infinits. Inicialitzar la llista amb infinits té dos objectius. El primer és que l'algorisme podrà saber si un vèrtex adjacent no ha estat visitat si la distància a ell és encara infinit. La segona cosa que aconsegueix és que els vèrtexs no accessibles des del vèrtex inicial tindran, quan l'algorisme acabi, la distància assignada correctament, és a dir, la seva distància serà infinit. Tota aquesta llista de distàncies s'inicialitza amb infinits amb excepció del primer element que representa la distància al primer vèrtex, al qual se li assigna distància zero, ja que evidentment sempre tindrem que $d(i, i) = 0$.

En segon lloc, també es crea una llista que ens servirà per guardar el vèrtex anterior al vèrtex al qual som, per així poder aconseguir el camí que s'ha seguit per arribar a cada vèrtex. Al principi quan es crea la llista tots els seus elements són elements nuls, ja que encara no s'ha explorat res. De manera similar amb la llista de distàncies, els vèrtexs que quan l'algorisme acabi encara no s'hagin explorat, seguiran amb valor nul, cosa que serà correcta perquè significarà que no hi ha camí que porti a aquells vèrtexs. El vèrtex inicial sempre tindrà adientment com a predecessor un valor nul.

En tercer lloc, s'inicialitza una *cua*. Una cua, o *queue* en anglès, és una estructura de dades similar a una llista però que per afegir, treure i llegir elements, funciona d'una manera diferent. Té dues funcions: *encuar un element*, que afegeix un element al final de la cua, i *desencuar*, que treu el primer element i el retorna. Bàsicament, en el món real seria com una filera de persones esperant per ser ateses, on la primera de sortir-ne i ser atesa serà la primera que s'ha posat a la fila i la gent que vagi arribant anirà al final. En el cas de l'algorisme de cerca en amplada, els elements de la cua seran vèrtexs. Aquesta cua s'utilitzarà per guardar els vèrtexs adjacents al vèrtex que estan pendents a explorar. Al principi s'afegeix a la cua el vèrtex inicial, ja que és pel que es comença a explorar.

Un cop inicialitzades les dues llistes i la cua, es comença un bucle on s'explorarà el graf i s'aniran omplint les llistes que al final ens permetran trobar el camí més curt al vèrtex que desitgem. Aquest bucle funcionarà mentre la cua no estigui buida, cosa que voldrà dir que encara queden vèrtexs accessibles des de i per explorar.

En cada iteració del bucle es desencua un vèrtex de la cua, que anomenarem u. Llavors, per cada vèrtex v adjacent a u el qual no ha estat explorat ja (això es pot saber mirant si la distància a ell és encara infinit), se li assigna com a predecessor a u i se li assigna com a distància $d(i, v) = d(i, u) + 1$. Això ho fem perquè com que haurem arribat a v des de u, el predecessor de v serà u i, com que v està una aresta més lluny, la distància serà la distància a u incrementada una unitat. A continuació, també encuem v a la cua perquè sigui explorat més tard.

Gràcies a la utilització d'una cua, sempre es desencuaran abans els vèrtexs que estiguin més a prop del vèrtex inicial, cosa que garantirà que es trobi el camí més curt.

Un cop tots els vèrtexs accessibles des de i s'han explorat, la cua quedarà buida i el bucle s'acabarà. Aquí és on també s'acaba l'algorisme que finalment retornarà la llista de predecessors, la qual ens permetrà trobar el camí més curt.

El pseudocodi[1] de l'algorisme de cerca en amplada és el següent:

```
algorisme cerca_en_amplada(G, i):
  distàncies = {∞} * G.|V|
  distàncies[i] = 0

  predecessors = {Nul} * G.|V|

  Q = cua buida
  encuar i a Q

  mentre Q no és buida:
    u = desencuar de Q

    per cada v ∈ G.Adj[u]:
      si distàncies[v] == ∞:
        predecessors[v] = u
        distàncies[v] = distàncies[u] + 1
        encuar v a Q

  retornar predecessors
```

Si ara volem aconseguir el camí més curt a un determinat vèrtex, podem aconseguir-ho a partir de la llista de predecessors que ha retornat l'algorisme de cerca en amplada. Per fer-ho, fem una llista buida i afegim al principi el vèrtex al qual volem arribar. Aquest vèrtex serà el final del camí i, per això, l'anomenarem f. Després, afegim al principi de la llista el predecessor de f, a continuació el predecessor del predecessor de f... Anem fent així fins que el predecessor sigui nul, cosa que voldrà dir que hem arribat al vèrtex inicial, que recordem que sempre té com a predecessor un valor nul.

El procediment descrit en el paràgraf anterior el podem escriure en pseudocodi de la següent manera:

```
algorisme reconstruir_camí(predecessors, f):
  camí = {}
  mentre f ≠ Nul:
    inserir f al principi de camí
    f = predecessors[f]
  retornar camí
```

[1] Alguns aclariments sobre el pseudocodi utilitzat a partir d'ara es troben a l'annex A.

4.1.2 Exemple

Per acabar d'entendre el funcionament de l'algorisme de cerca en amplada, vegem un exemple de com funcionaria si volguéssim fer-lo servir en el graf G de la figura 4.1, començant pel vèrtex $i = 0$. Això es faria executant `cerca_en_ampladda(G, 0)`, on G hauria de ser un graf que haguéssim definit per representar el de la figura.

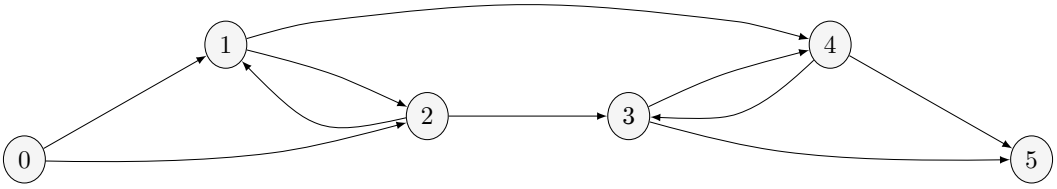

Figura 4.1: Graf G.

En la següent figura 4.2 veiem com comença l'algorisme de cerca en amplada. Primer inicialitza les distàncies a cada vèrtex (indicades en els quadrats grisos del costat de cada vèrtex) amb infinit menys la del vèrtex inicial, a la qual se li assigna zero. També s'assigna nul com a predecessor (representat pel color de les arestes) de tots els vèrtexs i s'encua 0 (els vèrtexs dins la cua es pinten de color vermell), que és el vèrtex inicial.

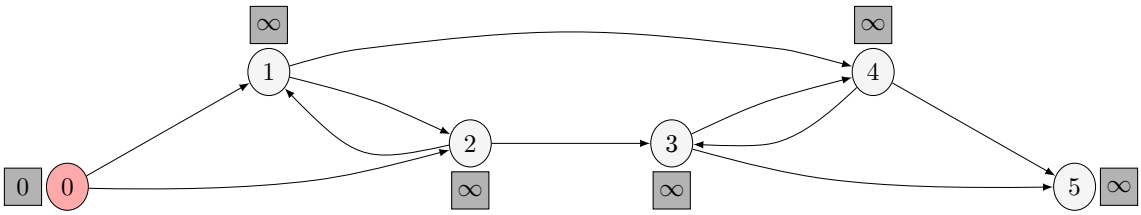

Figura 4.2: Abans de començar el bucle. $Q = \{0\}$

En la primera iteració (figura 4.3) es desencua de la llista un vèrtex u (pintat en groc), el qual en la primera iteració sempre serà l'inicial i, que en aquest cas és el vèrtex 0. A continuació, també s'assigna com a predecessor dels vèrtexs adjacents a u (els vèrtexs 1 i 2), el valor de u, que és el vèrtex 0. També s'assigna als vèrtexs adjacents la distància $d(i, u) + 1 = d(0, 0) + 1 = 0 + 1 = 1$. Finalment, s'encuen aquests dos vèrtex adjacents.

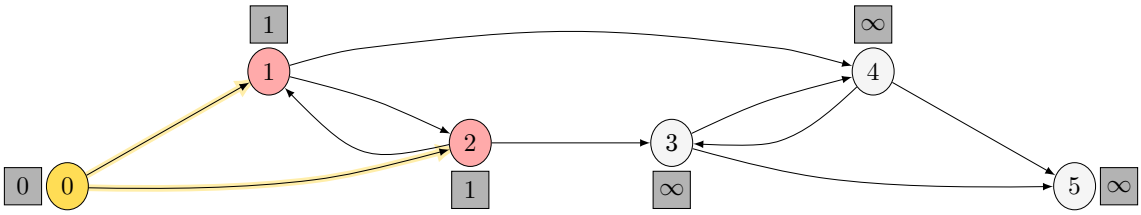

Figura 4.3: Al final de la primera iteració del bucle. $Q = \{1, 2\}$

En les següents iteracions (de la figura 4.4 a la 4.8) es va seguint el mateix procediment i s'acaba quan tots els vèrtexs accessibles des del vèrtex inicial han estat

explorats. En aquest cas tots els vèrtexs del graf són accessibles des de qualsevol vèrtex; per tant, s'acabarà quan tot el graf hagi sigut explorat.

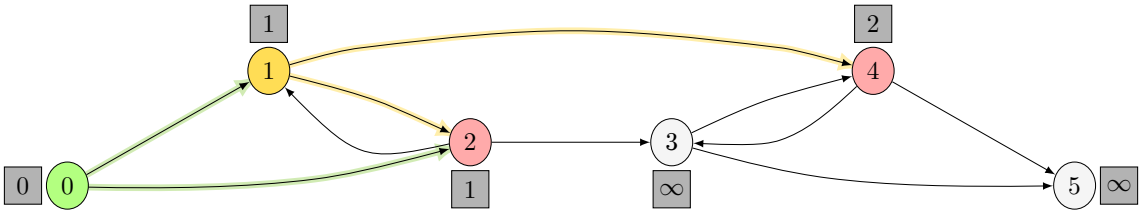

Figura 4.4: Al final de la segona iteració del bucle. $Q = \{2, 4\}$

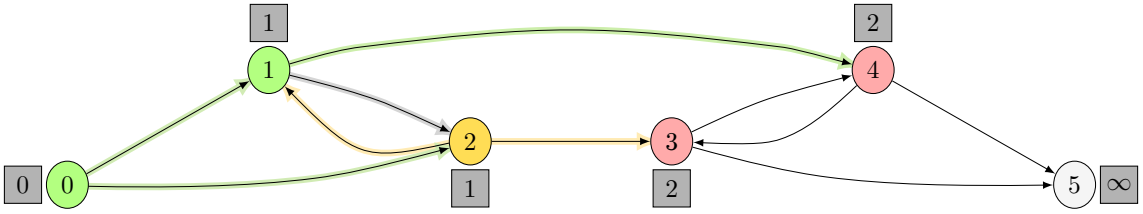

Figura 4.5: Al final de la tercera iteració del bucle. $Q = \{4, 3\}$

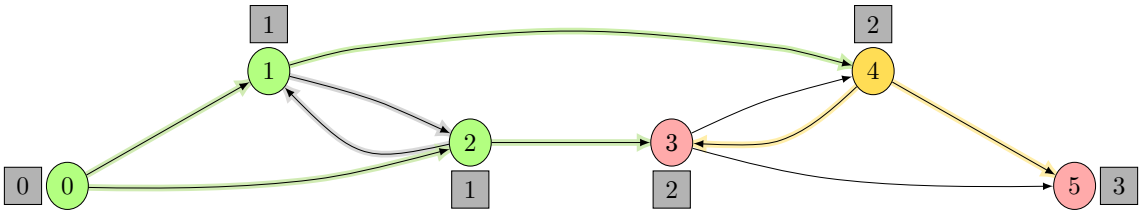

Figura 4.6: Al final de la quarta iteració del bucle. $Q = \{3, 5\}$

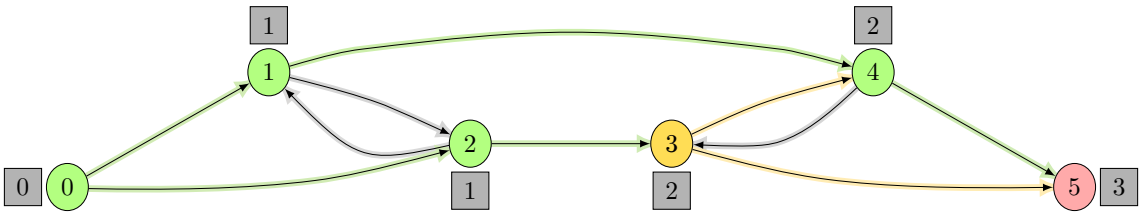

Figura 4.7: Al final de la cinquena iteració del bucle. $Q = \{5\}$

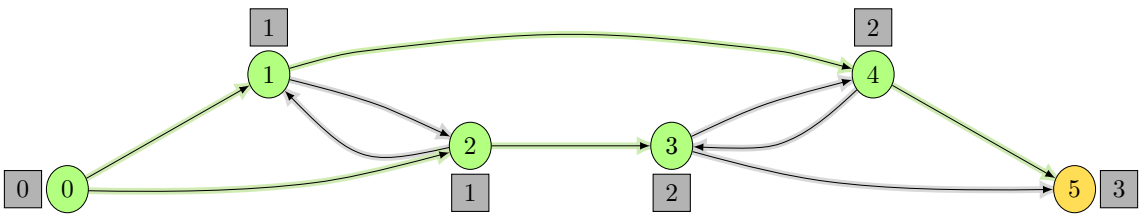

Figura 4.8: Al final de la sisena iteració del bucle. $Q = \{\}$

En la sisena iteració la cua Q queda buida; per tant, el bucle s'acaba.

Finalment, queda com a la figura 4.9.

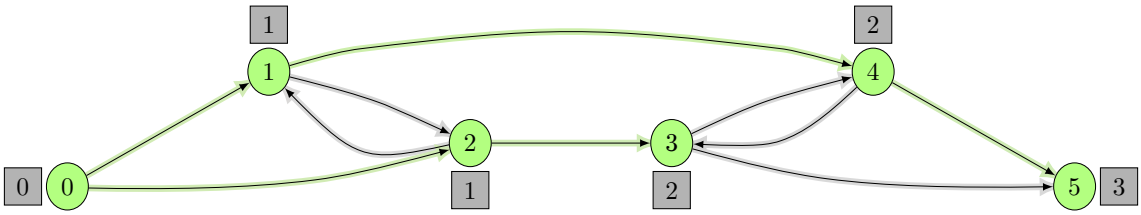

Figura 4.9: Al final de l'execució de l'algorisme.

La llista de predecessors retornada és $\{\text{Nul}, 0, 0, 2, 1, 4\}$, ja que el vèrtex 0 no té predecessor, els vèrtexs 1 i 2 tenen com a predecessor el vèrtex 0, el vèrtex 3 té com a predecessor el vèrtex 2, el 4 l'1 i, finalment, el 5 el 4.

Ara podem trobar el camí més curt que vagi des del vèrtex inicial $i = 0$ a qualsevol vèrtex del graf. Per exemple, si volem arribar al vèrtex $f = 4$, hauríem d'executar `reconstruir_camí({Nul, 0, 0, 2, 1, 4}, 4)`. Això retornaria correctament el camí $\{0, 1, 4\}$, que és el camí més curt del vèrtex $i = 0$ al vèrtex $f = 4$.

És interessant veure que l'algorisme de cerca en amplada sempre trobarà un únic camí per cada un dels vèrtexs del graf accessibles des del vèrtex inicial. És a dir, ens retornarà un subgraf de G del component on es troba el vèrtex inicial i el qual contindrà tots els vèrtexs d'aquest component i les arestes representaran el camí únic a cada vèrtex des de l'inicial; per tant, com que aquest subgraf serà un graf acíclic (ja que entre cada parell de vèrtexs només hi haurà un sol camí) i connex (perquè té un sol component), haurem obtingut un arbre d'expansió del component on es troba el vèrtex inicial. De fet, es pot veure molt bé com en la figura 4.9 els vèrtexs i arestes verdes formen un arbre d'expansió del component on es troba el vèrtex $i = 0$, que en aquest cas és tot el graf sencer.

Si en el graf G hi hagués hagut altres components, al final de l'execució de l'algorisme tots els vèrtexs d'aquests altres components haurien quedat amb una distància infinita i amb predecessors nuls. Això és adient perquè no és possible trobar un camí que connecti dos components. El bucle també hauria fet sis iteracions, ja que els vèrtexs dels altres components no haurien estat mai encuats i, per tant, no haurien influït en l'execució del bucle.

4.1.3 Una optimització

Com hem vist, l'algorisme de cerca en amplada troba el camí més curt des d'un vèrtex inicial a la resta de vèrtexs del graf. Tanmateix, si només ens interessa el camí més curt entre un vèrtex inicial i i un únic vèrtex final f, podem fer una petita optimització que farà que molts cops l'algorisme no hagi de processar tot el graf, cosa que el farà anar més de pressa.

L'optimització que podem fer és posar una condició dins del bucle que sigui que si el vèrtex desencuat u és igual a f, s'acabi el bucle i es trobi el camí. Això ho podem

fer perquè un cop desencuem un vèrtex, ja sabem tots els seus predecessors; per tant, en podem saber el camí que hi porta. Pot ser que f coincideixi amb l'últim vèrtex que s'hauria processat sense l'optimització. En aquest cas l'optimització no serviria, però en la majoria de casos es trobarà abans f.

Els canvis que hauríem de fer en el pseudocodi són els marcats en groc a continuació:

```
algorisme cerca_en_amplada(G, i, f):
  distàncies = {∞} * G.|V|
  distàncies[i] = 0

  predecessors = {Nul} * G.|V|

  Q = cua buida
  encuar i a Q

  mentre Q no és buida:
    u = desencuar de Q

    si u == f:
      retornar reconstruir_camí(predecessors, f)

    per cada v ∈ G.Adj[u]:
      si distàncies[v] == ∞:
        predecessors[v] = u
        distàncies[v] = distàncies[u] + 1
        encuar v a Q

  retornar Nul
```

A la primera línia l'únic que canvia és que afegim f com a paràmetre, que és el vèrtex final. Dins del bucle, afegim la condició descrita anteriorment. Com podem veure, ara l'algorisme retornarà directament el camí més curt, no la llista de predecessors. Finalment, en l'última línia posem que retorni nul. Això ho fem perquè si arriba a l'última línia, voldrà dir que el vèrtex f no s'ha trobat i que, per tant, el vèrtex f no és accessible des de i, la qual cosa vol dir que no existeix un cap camí de i a f.

En el graf de l'exemple de la secció anterior si haguéssim fet servir aquesta optimització per trobar el camí més curt des de $i = 0$ fins a $f = 4$, només s'hauria hagut d'arribar a la quarta iteració (figura 4.6), on s'hauria desencuat el vèrtex 4 i s'hauria pogut trobar ja el camí més curt.

4.2 Algorisme de Dijkstra

L'algorisme de Dijkstra és un algorisme molt similar al de cerca en amplada, però que permet trobar el camí més curt en un graf ponderat. Va ser descobert l'any 1956 per Edsger W. Dijkstra, científic de la computació que dona nom a l'algorisme.

Dijkstra explica en una entrevista que quan va inventar l'algorisme estava treballant al Centre de Matemàtiques d'Amsterdam, on s'estava desenvolupant un dels primers ordinadors dels Països Baixos, l'ARMAC[2]. Ell era l'encarregat de fer una

[2]Sigles en neerlandès de *Automatische Rekenmachine van het MAthematische Centrum* (Calculadora automàtica del Centre de Matemàtiques).

demostració del poder que tenia aquest nou ordinador, on hi havia d'assistir gent sense molts coneixements matemàtics. Per això, havia d'ensenyar com l'ordinador solucionava un problema que els assistents poguessin entendre'n tant l'enunciat com la solució. Se li va acudir dissenyar un programa que trobés el camí més curt entre dues ciutats dels Països Baixos, fent servir un mapa de carreteres reduït que incloïa 64 ciutats. Uns dies abans, prenent el cafè, estava pensant en com solucionar el problema i en vint minuts va dissenyar l'algorisme (vegeu la cita del principi del capítol). Avui en dia, aquest algorisme inventat en vint minuts fa més de 50 anys segueix sent un dels algorismes més utilitzats per trobar el camí més curt en grafs ponderats.

Una limitació que té l'algorisme de Dijkstra és que no funciona amb grafs ponderats que tinguin alguna aresta amb pes negatiu. Tanmateix, en la majoria d'aplicacions en què es vol buscar el camí més curt, el pes de les arestes no sol ser negatiu. En el cas d'un planificador de rutes, el pes de les arestes mai serà negatiu, ja que aquí el pes es fa servir per representar la distància al món real, la qual mai pot ser negativa. Per casos on sí que hi hagi arestes amb pesos negatius, existeixen altres algorismes que sí que poden trobar-ne el camí més curt. Per exemple, és el cas de l'algorisme de Bellman-Ford. Tot i això, aquests altres algorismes són més lents que l'algorisme de Dijkstra.

No obstant això, cap algorisme pot trobar el camí més curt en un graf ponderat amb cicles de pes negatiu, ja que no és possible trobar el camí més curt en un graf d'aquest tipus. Això és perquè el camí començaria a donar voltes en el cicle per reduir cada cop més el pes del camí, però no acabaria mai. La distància d'un camí així seria menys infinit.

4.2.1 Procediment

De la mateixa manera que l'algorisme de cerca en amplada, l'algorisme de Dijkstra comença inicialitzant dues llistes per assignar a cada vèrtex la distància des del vèrtex inicial, i el predecessor de cada vèrtex. La llista de distàncies també s'inicialitza amb infinits i la llista de predecessors amb valors nuls. També s'assigna zero a la distància al vèrtex inicial.

Però, a diferència de l'algorisme de cerca en amplada, en l'algorisme de Dijkstra no es fa servir una cua normal, sinó que es fa servir una *cua de prioritat*. Una cua de prioritat, o en anglès *priority queue*, és com una cua però cada element té un valor associat, anomenat *prioritat*. Aquest valor determinarà la posició en què es col·loca l'element. Com menor sigui el valor de prioritat, més endavant anirà. Les cues de prioritat tenen dues funcions: *encuar un element amb prioritat* x, que afegeix un element a la cua, amb un valor de prioritat associat x, i *desencuar*, que treu l'element amb la menor prioritat de la cua i el retorna. Com en la cerca en amplada, al principi encuem el vèrtex inicial, però aquí ho fem amb prioritat 0, ja que així serà el vèrtex pel qual començarem a explorar el graf.

Una vegada inicialitzades les dues llistes i la cua de prioritat, es comença un bucle, el qual com en la cerca en amplada s'executarà mentre la cua no estigui buida. Quan

la cua estigui buida voldrà dir que haurem processat tots els vèrtexs accessibles des de l'inicial; per tant, podrem trobar el camí més curt si existeix (no existirà un camí si el vèrtex inicial i el final es troben en components diferents).

El primer que fem dins del bucle és desencuar de la cua de prioritat un vèrtex, que anomenarem u. Aquest vèrtex per la manera en què funciona la cua de prioritat serà sempre el que tingui una prioritat assignada menor. En densencuar el vèrtex u també es garanteix que l'algorisme ja sabrà el valor $d(i, u)$, és a dir, la distància del camí més curt de i fins a u. Després de desencuar u, per cada vèrtex v adjacent a u, si la suma $d(i, u)+p(u \xrightarrow{e} v)$[3] és menor que la distància que teníem calculada per anar a v, voldrà dir que hem trobat un camí més curt per arribar a v. Per això, reassignarem al predecessor de v com a u, la distància a v com la suma $d(i, u)+p(u \xrightarrow{e} v)$ i, finalment, encuarem v a la cua també amb aquesta nova distància a v com a prioritat.

Com que al principi tots els vèrtexs (menys l'inicial) tenen una distància assignada d'infinit, tots els vèrtexs que siguin accessibles des de l'inicial obtindran una nova distància, ja que qualsevol nombre que doni la suma del pes de u més el pes de l'aresta de u a v serà menor que ∞. Tot i això, la primera distància assignada pot ser que canviï per una de més curta si durant l'execució de l'algorisme es troba que hi ha un camí més curt. De la mateixa manera també canviaríem el predecessor de v per reflectir el nou camí. També es tornaria a encuar v, ja que com que és un possible camí més curt, ens interessarà saber si més camins més curts passen per v.

D'altra banda, cal notar que, com en l'algorisme de cerca en amplada, es comença a processar el graf des del vèrtex inicial del camí, ja que serà el que es desencuarà en la primera iteració del bucle. El vèrtex inicial mai rebrà una nova distància, ja que des del principi ja té distància 0 (recordem que no treballem amb pesos negatius). Lògicament si un camí té longitud 0, no se'n pot trobar un d'encara més curt.

El pseudocodi de l'algorisme de Dijkstra és el següent:

```
algorisme dijkstra(G, i):
  distàncies = {∞} * G.|V|
  distàncies[i] = 0

  predecessors = {Nul} * G.|V|

  PQ = cua de prioritat buida
  encuar i a PQ amb prioritat 0

  mentre PQ no és buida:
    u = desencuar de PQ

    per cada v ∈ G.Adj[u]:
      si distàncies[u] + pes(u, v) < distàncies[v]:
        predecessors[v] = u
        distàncies[v] = distàncies[u] + pes(u, v)
        encuar v a PQ amb prioritat distàncies[v]

  retornar predecessors
```

[3] $p(u \xrightarrow{e} v)$ és el pes de l'aresta e que connecta de u fins a v

Acabat el bucle es retorna la llista amb els predecessors de cada vèrtex, la qual ens permetrà trobar el camí més curt entre el vèrtex inicial i i qualsevol altre vèrtex del graf utilitzant la funció `reconstruir_camí`, la mateixa que s'utilitza en l'algorisme de cerca en amplada.

4.2.2 Exemple

Utilitzarem el mateix graf que en l'exemple de cerca en amplada, però ara amb pesos en les seves arestes, cosa que fa que sigui probable que el camí més curt no sigui el que passi pel menor nombre d'arestes. Vegem si és així.

El graf ponderat G utilitzat és el mostrat en la figura 4.10. Els nombres al costat de les arestes marquen els pesos de cada aresta.

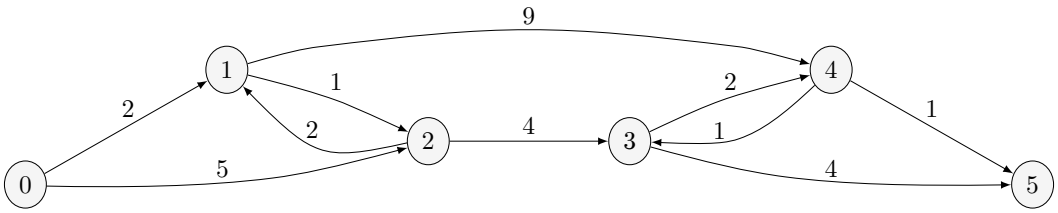

Figura 4.10: Graf G.

Fem servir el vèrtex 0 com a vèrtex inicial i. Conseqüentment, per fer-ho s'executa `dijkstra(G, 0)`, on G hauria de ser un graf ponderat que haguéssim definit per representar el de la figura.

L'algorisme de Dijkstra, com l'algorisme de cerca en amplada, comença assignant al vèrtex inicial distància 0 i l'encua a la cua de prioritat amb prioritat 0. A la resta de vèrtexs se'ls hi assigna distància ∞. A tots els vèrtexs també se'ls hi assigna com a predecessor un valor nul.

En el peu de les següents figures s'indica el contingut de la cua de prioritat PQ, la qual conté parells de nombres (v, p), on v és el vèrtex i p la prioritat. La cua de prioritat sempre s'ordenarà automàticament fent que com menys prioritat tinguin els elements, més endavant vagin.

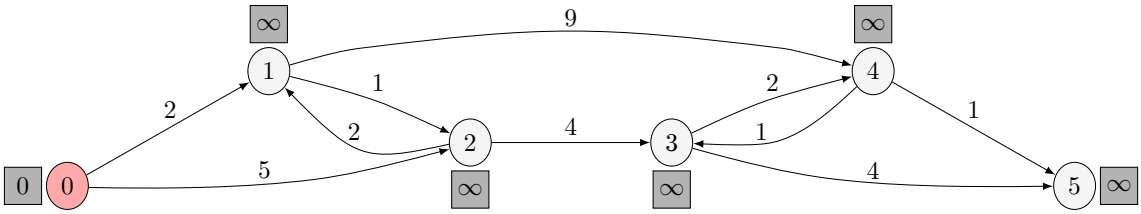

Figura 4.11: Abans de començar el bucle. $PQ = \{(0, 0)\}$

Ara comença el bucle. En la primera iteració es desencua el vèrtex 0, ja que és el que té menor prioritat de la llista (és l'únic que hi ha i té prioritat 0). A continuació es processa cada vèrtex adjacent a 0. Primer, es comprova si la distància

assignada al vèrtex 0 ($d(i,u) = d(0,0) = 0$) més el pes de l'aresta que va de 0 a 1 ($p(u \xrightarrow{e} v) = p(0 \xrightarrow{e} 1) = 2$) és menor que la distància que té actualment el vèrtex 1 (∞). Evidentment, $0+2 = 2 < \infty$. Com que es compleix aquesta condició, s'assigna al vèrtex 1 una distància 2 i com a predecessor el vèrtex 0. Passa de forma similar amb el vèrtex 2, que també és adjacent al 0. Com que $0 + 5 = 5 < \infty$, la nova distància del vèrtex 2 serà 5 i el seu predecessor serà el vèrtex 0. Podem veure tot això en la figura 4.12.

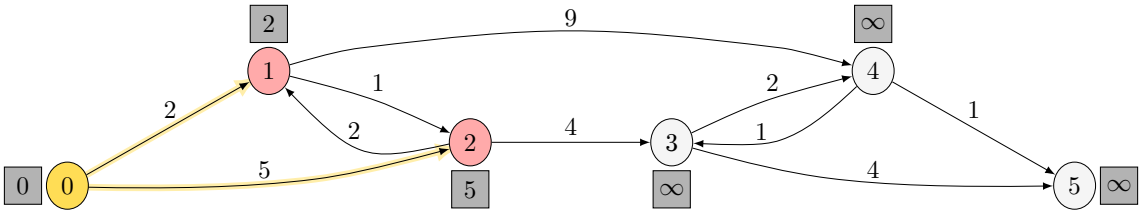

Figura 4.12: Al final de la primera iteració del bucle. $PQ = \{(1,2),(2,5)\}$

En la segona iteració del bucle (figura 4.13) es desencua el vèrtex 1, ja que és el que tenia menor prioritat. Observem que ocorre amb el vèrtex 4 el mateix que passava amb els vèrtexs 1 i 2 en la primera iteració. El vèrtex 4 es processa perquè és adjacent al vèrtex 1 i com que es compleix que $d(i,u) + p(u \xrightarrow{e} v) = d(0,1) + p(1 \xrightarrow{e} 4) = 2 + 9 = 11 < \infty$, s'assignarà al vèrtex 4 una distància de 11 i a 1 com el seu predecessor. No obstant això, en el vèrtex 2, que es processa un altre cop perquè també és adjacent al vèrtex 1, veiem que es reassigna una nova distància. Això és a causa que s'ha trobat un camí més curt que l'anterior. Al final de la primera iteració l'algorisme creia que el camí més curt de $i = 0$ fins al vèrtex 2 era passar per l'aresta de pes 5 que anava directament de 0 a 2. Tanmateix, ara en la segona iteració, es comprova que $d(i,u) + p(u \xrightarrow{e} v) = d(0,1) + p(1 \xrightarrow{e} 2) = 2 + 1 = 3 < 5$; per tant, es reassigna 3 com la distància del vèrtex 2 i ara tindrà com a predecessor el vèrtex 1 en comptes del vèrtex 0.

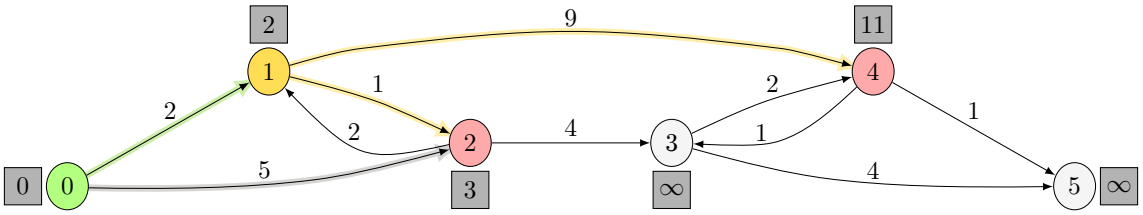

Figura 4.13: Al final de la segona iteració del bucle. $PQ = \{(2,3),(2,5),(4,11)\}$

En la tercera iteració es desencua el vèrtex 2 i, per tant, es processen els seus vèrtexs adjacents 1 i 3. Per una banda, amb el vèrtex 1 no passa res perquè la distància del vèrtex 2 ($d(0,2) = 3$) més el pes de l'aresta que connecta de 2 a 1 ($p(2 \xrightarrow{e} 1) = 2$) no és menor que la distància que ja té el vèrtex 1 ($3 + 2 = 5 \not< 2$). A més això crearia un cicle que no tindria sentit com a part del camí més curt. Per altra banda, es processa el vèrtex 3 i se li assigna la distància i el predecessor de manera semblant a com s'ha fet amb els vèrtexs que tenien distància ∞ en iteracions anteriors. En aquesta iteració s'encuarà el vèrtex 3 però no l'1, ja que en el cas del vèrtex 1 no s'haurà satisfet la condició.

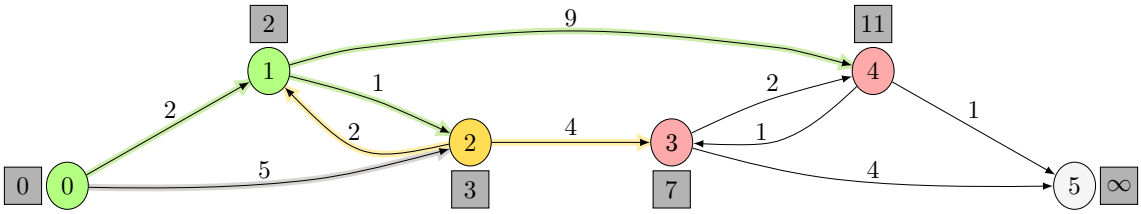

Figura 4.14: Al final de la tercera iteració del bucle. $PQ = \{(2, 5), (3, 7), (4, 11)\}$

En la quarta iteració es torna a desencuar el vèrtex 2, però no es troben nous camins més curts; per tant, seguiria quedant com en la figura 4.14, menys que ara $PQ = \{(3, 7), (4, 11)\}$.

En les següents tres iteracions (de la figura 4.15 a la 4.17) es procedeix de manera similar a les iteracions anteriors.

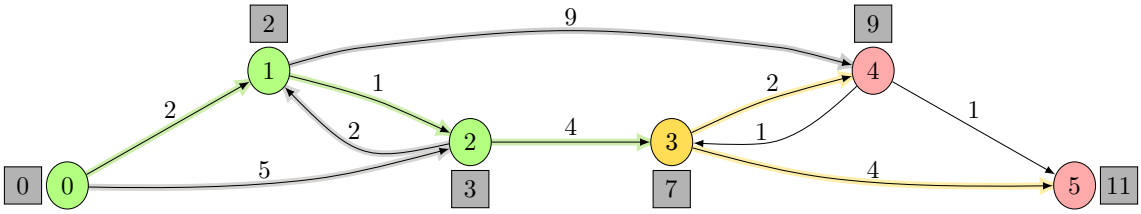

Figura 4.15: Al final de la cinquena iteració del bucle. $PQ = \{(4, 9), (4, 11), (5, 11)\}$

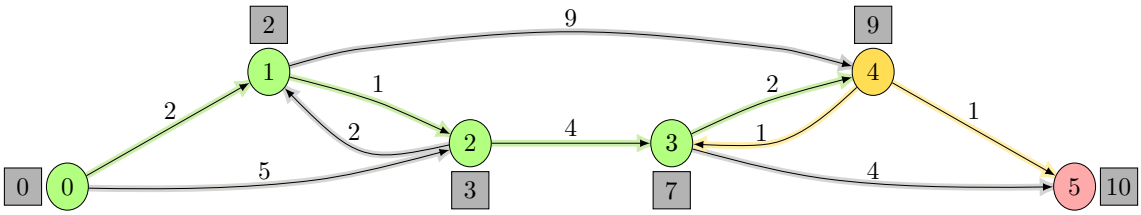

Figura 4.16: Al final de la sisena iteració del bucle. $PQ = \{(5, 10), (5, 11), (4, 11)\}$

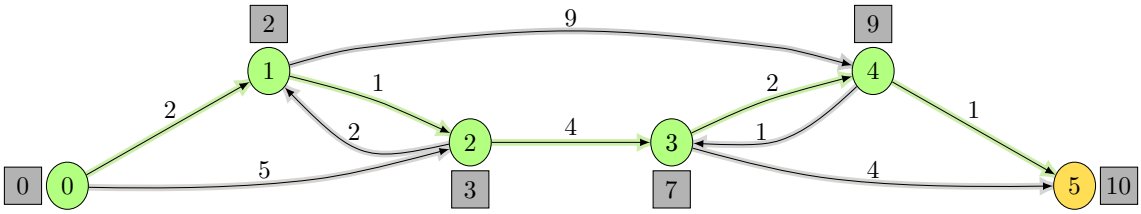

Figura 4.17: Al final de la setena iteració del bucle. $PQ = \{(4, 11), (5, 11)\}$

En la vuitena iteració es desencua el vèrtex 4 però no es troben camins més curts; per tant, tot segueix igual, menys que ara $PQ = \{(5, 11)\}$.

En la novena iteració passa igual que en l'anterior. Es desencua el vèrtex 5 però no es troben nous camins més curts. No obstant això, al final d'aquesta iteració la cua

de prioritat serà $PQ = \{\}$, és a dir, quedarà buida. Això farà que s'acabi el bucle, i que, per tant, ja puguem trobar el camí més curt a qualsevol vèrtex des del vèrtex inicial $i = 0$.

Finalment, queda com a la figura 4.18.

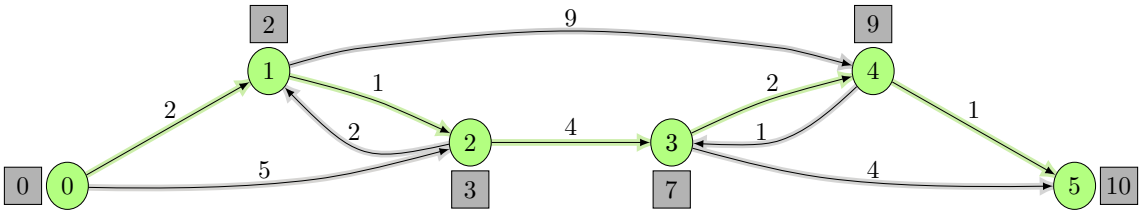

Figura 4.18: Al final de l'execució de l'algorisme.

L'algorisme de Dijkstra al final retornarà la llista de predecessors, que serà $\{\text{Nul}, 0, 1, 2, 3, 4\}$. Això indicarà que el vèrtex 0 no té predecessor (atès que és l'inicial), que el vèrtex 1 té com a predecessor el 0, el 2 l'1, el 3 el 2, el 4 el 3 i, en últim terme, el 5 el 4.

De la mateixa manera que en l'algorisme de cerca en amplada, ara podem trobar el camí més curt a qualsevol vèrtex f de G que comenci des de i. Per exemple, si volem que el camí acabi a $f = 4$, executem `reconstruir_camí({Nul, 0, 1, 2, 3, 4}, 4)`. Això retornaria el camí $\{0, 1, 2, 3, 4\}$, que efectivament és el camí més curt des de $i = 0$ fins a $f = 4$.

Com es pot veure, tot i utilitzar un graf amb els mateixos vèrtexs i les mateixes arestes en l'exemple de l'algorisme de cerca d'amplada i en aquest de l'algorisme de Dijkstra, en afegir pesos a les arestes, el camí més curt a la majoria dels vèrtexs del graf ha canviat. Tanmateix, l'algorisme de Dijkstra, igual que el de cerca en amplada i qualsevol altre algorisme per buscar el camí més curt, ens retornarà sempre un arbre d'expansió del component on es troba el vèrtex inicial, com el que podem veure en la figura 4.18, o en la figura 4.9 en el cas de l'exemple de l'algorisme de cerca en amplada.

4.2.3 Una optimització

En l'algorisme de Dijkstra es pot fer la mateixa optimització explicada per l'algorisme de cerca en amplada si només ens interessa el camí d'un vèrtex i a un altre vèrtex f. Al desencuar un vèrtex l'algorisme de Dijkstra ja haurà trobat el camí més curt a aquest, encara que no s'hagin explorat tots els vèrtexs del graf.

Aquesta optimització és de gran utilitat per aplicacions com el planificador de rutes, ja que en trobar la destinació on es vol arribar l'algorisme, ja pot acabar. Com a exemple, suposem que volem trobar el camí més curt de Calldetenes a Vic. Si utilitzem l'algorisme de Dijkstra sense l'optimització, segurament s'acabarien processant tots els vèrtexs d'Eurafràsia[4], que seria el component on es troba el vèrtex inicial

[4]Eurafràsia és el supercontinent format per Europa, Àfrica i Àsia (Europa i Àsia estant connectats per carreteres terrestres i Àsia i Àfrica estan connectats pels ponts del canal de Suez).

(algun indret de Calldetenes). Evidentment el camí més curt de Vic a Calldetenes no passa ni per Singapur, ni per Ciutat del Cap; per tant no ens interessa analitzar tot el component, només fins que trobem la destinació final.

L'optimització en el pseudocodi quedaria de la següent manera (els únics canvis que s'han de fer són les línies marcades en groc):

```
algorisme dijkstra(G, i, f):
  distàncies = {∞} * G.|V|
  distàncies[i] = 0

  predecessors = {Nul} * G.|V|

  PQ = cua de prioritat buida
  encuar i a PQ amb prioritat 0

  mentre PQ no és buida:
    u = desencuar de PQ

    si u == f:
      retornar reconstruir_camí(predecessors, f)

    per cada v ∈ G.Adj[u]:
      si distàncies[u] + pes(u, v) < distàncies[v]:
        predecessors[v] = u
        distàncies[v] = distàncies[u] + pes(u, v)
        encuar v a PQ amb prioritat distàncies[v]

  retornar Nul
```

4.3 Algorisme de cerca A*

L'algorisme de cerca A* (pronunciat *a-estrella* o *a-star*), o simplement A*, és una modificació de l'algorisme de Dijkstra amb l'optimització descrita en la secció 4.2.3, la qual millora encara més el seu rendiment. Aquest increment en l'eficàcia s'aconsegueix a partir de l'ús d'heurístiques que guien la cerca cap al vèrtex final.

L'algorisme va ser publicat l'any 1968 per Peter Hart, Nils Nilsson i Bertram Raphael, investigadors del Centre d'Intel·ligència Artificial de l'Institut de Recerca de Stanford. Aquests tres investigadors formaven part del projecte Shakey, que tenia l'objectiu de construir el primer robot mòbil que pogués planificar les seves pròpies accions. La part que s'encarregava de trobar per on havia de passar el robot Shakey funcionava gràcies a l'algorisme de cerca A*, el qual van inventar com a part del projecte.

Figura 4.19: Robot Shakey.

L'"A" d'"A*" prové de la inicial d'*algorithm*, algorisme en anglès, i "*" és per indicar que troba el camí més curt, que sovint es denota $P*$.

4.3.1 Procediment

L'únic que canvia en l'algorisme de cerca A* respecte de l'algorisme de Dijkstra amb l'optimització són les dues línies marcades en groc:

```
algorisme a_star(G, i, f):
  distàncies = {∞} * G.|V|
  distàncies[i] = 0

  predecessors = {Nul} * G.|V|

  PQ = cua de prioritat buida
  encuar i a PQ amb prioritat 0

  mentre PQ no és buida:
    u = desencuar de PQ

    si u == f:
      retornar reconstruir_camí(predecessors, f)

    per cada v ∈ G.Adj[u]:
      si distàncies[u] + pes(u, v) < distàncies[v]:
        predecessors[v] = u
        distàncies[v] = distàncies[u] + pes((u, v))
        encuar v a PQ amb prioritat distàncies[v] + h(v, f)

  retornar Nul
```

En la primera línia l'únic que varia és el nom de l'algorisme. Els paràmetres requerits segueixen sent el graf G en què volem trobar el camí més curt, el vèrtex inicial i per on volem que comenci el camí i el vèrtex final f, que és al vèrtex al qual volem arribar de la manera més òptima des de i.

Llavors, l'altre únic canvi és dins del bucle, on encuem v a la cua de prioritat. En comptes d'encuar v amb una valor de prioritat corresponent a només la suma de $d(i, u) + p(u \xrightarrow{e} v)$, també hi sumem el valor de $h(v, f)$. La funció h és una funció heurística, el truc que fa que A* funcioni encara més de pressa que l'algorisme de Dijkstra.

4.3.2 Heurístiques

Una *funció heurística*, també anomenada simplement heurística, és una funció que ordena les diferents opcions que té un algorisme de cerca en cada pas a partir d'informació extra amb la finalitat de trobar més ràpidament l'objectiu final.

Les heurístiques són la base del camp de la intel·ligència artificial, ja que també es poden utilitzar per arribar a una solució aproximada per tasques en les quals no es coneix un algorisme que trobi una solució en un temps d'execució raonable.

L'heurística farà que en cada iteració del bucle s'examini primer un vèrtex que és més propens a formar part del camí més curt cap al vèrtex final. Per fer-ho, en cada iteració la prioritat del vèrtex v serà $f(v) = g(v) + h(v)$.

Per una banda, $g(v)$, com en l'algorisme de Dijkstra, serà $d(i,u) + p(u \xrightarrow{e} v)$. Recordem que $d(i,u)$ és la distància del camí més curt del vèrtex inicial i a u, u és el predecessor de v, i $p(u \xrightarrow{e} v)$ és el pes de l'aresta e que va de u a v.

Per l'altra banda, $h(v)$ representa la distància heurística estimada des de v fins al vèrtex final f. És a dir, $f(v)$ serà una estimació de la distància del camí més curt de i a f que passa per v.

Per això, serà molt raonable voler processar primer els vèrtexs que minimitzin el valor de $f(v)$. Es processaran abans els vèrtexs amb menor valor de $f(v)$ gràcies a utilitzar una cua de prioritat.

La funció heurística $h(v)$ que s'utilitza en l'algorisme de cerca A* depèn de l'aplicació en què es vulgui buscar el camí més curt. Tanmateix, hem de tenir en compte que la funció heurística que dissenyem sigui *admissible*. S'anomena admissible a una funció heurística si mai sobreestima la distància real entre v i f. És a dir, en el cas del camí més curt, una funció heurística serà admissible sempre que $h(v) \leq d(v,f)$.

En el cas que $h(v)$ no sigui admissible, és a dir, si sobreestima, l'algorisme A* funcionarà igualment, però no es garanteix que el camí que trobi sigui el més curt. Tot i això, si $h(v)$ sobreestima, A* normalment es dirigirà en menys iteracions al vèrtex final. Això pot ser útil en situacions on hi ha grafs gegants els quals tardarien massa a processar-se per trobar la millor solució. En moltes ocasions una bona solució, encara que no sigui la més òptima, també pot ser d'utilitat.

Si sempre passa que $h(v) = d(v,f)$, A* farà el mínim nombre d'iteracions possibles, ja que en cada iteració triarà la millor opció. No obstant això, per fer una heurística així prèviament hauríem de saber el camí més curt, cosa que faria que ja no haguéssim de buscar-lo un altre cop.

Si $h(v) < d(v,f)$, es garanteix que A* trobi el camí més curt. Com més bona sigui l'estimació, és a dir, com menor sigui la diferència $d(v,f) - h(v)$, més aviat s'arribarà al vèrtex final i, per tant, més ràpid s'anirà a trobar el camí.

Finalment, si $h(v) = 0$, l'algorisme de cerca A* equivaldrà a l'algorisme de Dijkstra i $g(v)$ serà l'únic encarregat de controlar la cerca. Evidentment, en aquesta situació també es garanteix que es trobi el camí més curt, tot i que com que l'heurística és nul·la l'algorisme tardarà més.

Una possible funció heurística utilitzada en planificadors de rutes s'explicarà en el següent capítol, on també s'explicarà la implementació completa del planificador de rutes, que a més tindrà un visualitzador que permetrà veure com funcionen els algorismes.

4.4 Cerques bidireccionals

Aprofitant que només volem buscar el camí més curt entre dos vèrtexs i i f, i no des de l'inicial i a tots els altres del graf, encara podem fer una altra millora als algorismes presentats: començar a buscar des dels dos vèrtexs i i f, és a dir, fer una cerca bidireccional.

L'algorisme de Dijkstra sol expandir-se per tot el graf creant una forma circular, ja que s'expandeix cap a totes bandes perquè no utilitza heurístiques.

Si representem l'arbre que l'algorisme de Dijkstra genera quan visita vèrtexs en un graf, sol quedar quelcom semblant al representat en la figura 4.20[5].

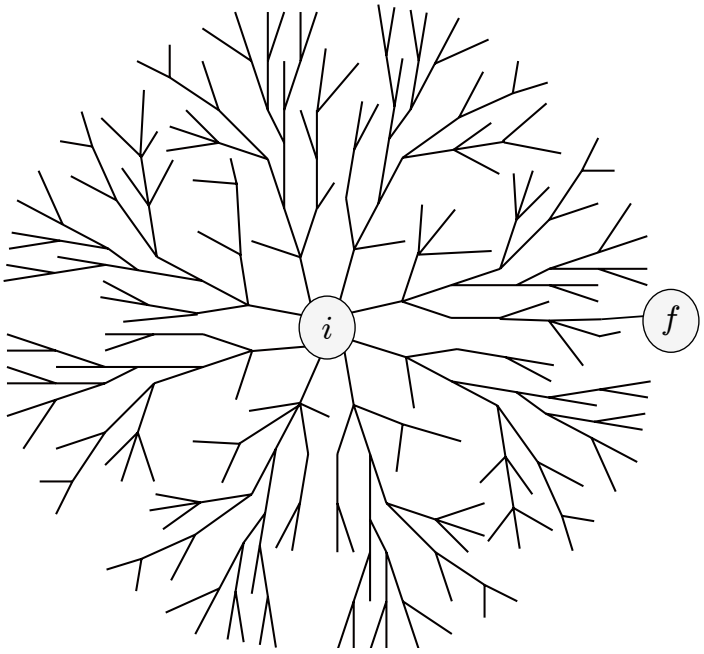

Figura 4.20: Representació de com sol quedar l'arbre quan Dijkstra va visitant vèrtexs començant pel vèrtex inicial i i fins que troba f.

Tanmateix, si comencem la cerca des dels dos vèrtexs, veurem que passarà diferent, ja que tindrem dos cercles, com es veu a la figura 4.21.

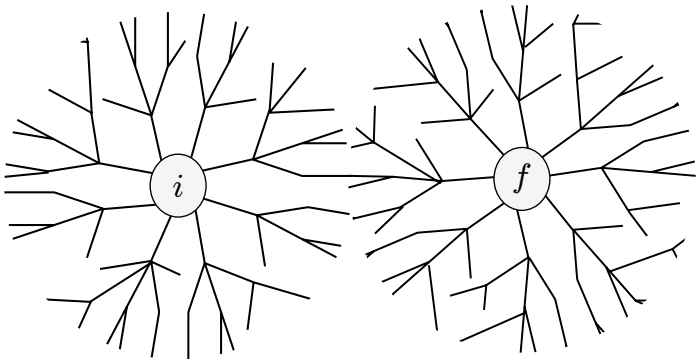

Figura 4.21: Representació de com sol quedar l'arbre quan Dijkstra va visitant vèrtexs començant pels dos vèrtexs i i f, és a dir, quan és bidireccional. Convé ressaltar que la distància entre els dos vèrtexs és la mateixa en les dues figures.

En la cerca bidireccional, pararem quan els dos cercles es trobin, ja que en aquest moment ja podrem trobar el camí. La cerca bidireccional serà més eficient, ja que el nombre de vèrtexs visitats serà menor.

[5]Els vèrtexs de la figura representen vèrtexs del graf, encara que no hi hagi cercles dibuixats. La longitud de cada aresta representa el seu pes.

Tot i això, crear l'algorisme no és tan fàcil com podria semblar a primera vista. El primer que haurem de fer és aconseguir que la cerca des de f sigui possible i respecti com és el graf. Per cercar al revés, haurem d'anar en contra les direccions de les arestes. Per fer-ho, a partir de la llista d'adjacència, crearem el que s'anomena *llista d'incidència*. Una llista d'incidència guardarà els vèrtexs que incideixen a cada vèrtex, és a dir els que entren (en contraposició a una llista d'adjacència que guarda els que surten). A les següents figures es pot veure un exemple:

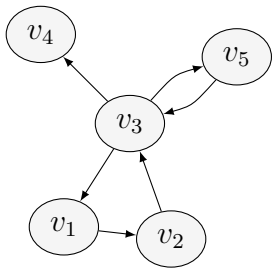

Figura 4.22: Diagrama de G.

```
Adj = {
       {2},
       {3},
       {1, 4, 5},
       {},
       {3}
      }
```

Figura 4.23: Llista d'adjacència Adj de G.

```
Inc = {
       {3},
       {1},
       {2, 5},
       {3},
       {3}
      }
```

Figura 4.24: Llista d'incidència Inc de G.

Es veu bastant clar en les figures que si en el primer element de la llista d'adjacència (v_1) hi havia un 2, en la llista d'incidència hi haurà un 1 en el segon element (v_2), i així per tots els vèrtexs del graf. Aquest algorisme el podem escriure en pseudocodi de la següent manera:

```
algorisme generar_llista_incidència(G):
  G.Inc = {{}} * G.|V|
  per cada u ∈ G.V:
    per cada v ∈ G.Adj[u]:
      inserir u a G.Inc[v]
  retornar G.Inc
```

Un cop tinguem la llista d'incidència, la podrem fer servir com a llista d'adjacència per la cerca que comenci des de f.

Llavors, farem el mateix que fa l'algorisme de Dijkstra però en els dos vèrtexs. Tot i això, la condició que farà acabar l'algorisme és quan un vèrtex s'hagi processat per les dues cerques, la que comença a i i va endavant, i la que comença a f i va endarrere. A aquest vèrtex mitjà, l'anomenarem m.

A part de necessitar la llista d'incidència, també haurem de fer servir dues llistes de distàncies, dues llistes de predecessors i dues cues de prioritat.

Abans de començar el bucle principal també haurem d'inicialitzar les llistes de distàncies amb infinits per cada vèrtex menys 0 pel vèrtex i en el cas de la cerca que va endavant i pel vèrtex f en el cas de la llista que va endarrere. Les llistes de predecessors, com en l'algorisme de Dijkstra unidireccional, al principi seran inicialitzades amb valors nuls. Finalment, encuarem amb prioritat 0 a la cua de prioritat de la cerca que va endavant el vèrtex i i a la cua de prioritat de la cerca que va endarrere el vèrtex f.

A diferència de l'algorisme de Dijkstra, encara necessitarem un parell de llistes més: la de vèrtexs processats. Aquesta llista l'utilitzarem en un principi per poder saber quan les dues cerques han processat un mateix vèrtex (l'anomenat vèrtex m) i, per tant, podem aturar les cerques i trobar el camí més curt. Al començament estaran buides i anirem afegint els vèrtexs que anem processant en cada cerca.

Com en l'algorisme de Dijkstra unidireccional, hi haurà un bucle principal que s'anirà executant, però en aquest cas mentre alguna de les dues cues de prioritat no estigui buida.

El pseudocodi del mètode descrit fins ara és el següent:

```
algorisme dijkstra_bidireccional(G, i, f):
  distàncies_i = {∞} * G.|V|
  distàncies_i[i] = 0
  distàncies_f = {∞} * G.|V|
  distàncies_f[f] = 0

  predecessors_i = {Nul} * G.|V|
  predecessors_f = {Nul} * G.|V|

  processats_i = {}
  processats_f = {}

  PQ_i = cua de prioritat buida
  encuar i a PQ_i amb prioritat 0
  PQ_f = cua de prioritat buida
  encuar f a PQ_f amb prioritat 0

  mentre PQ_i no és buida i PQ_f no és buida:
    u = desencuar de PQ_i
    per cada v ∈ G.Adj[u]:
      si distàncies_i[u] + pes(u, v) < distàncies_i[v]:
        predecessors_i[v] = u
        distàncies_i[v] = distàncies_i[u] + pes(u, v)
        encuar v a PQ_i amb prioritat distàncies_i[v]
    si u ∈ processats_f:
      processats = procesats_i ∪ processats_f
      retornar reconstruir_camí(i, distàncies_i, predecessors_i,
                                f, distàncies_f, predecessors_f,
                                processats)

    u = desencuar de PQ_f
    per cada v ∈ G.Inc[u]:
      si distàncies_f[u] + pes(v, u) < distàncies_f[v]:
        predecessors_f[v] = u
        distàncies_f[v] = distàncies_f[u] + pes(v, u)
        encuar v a PQ_f amb prioritat distàncies_f[v]
    si u ∈ processats_i:
      processats = procesats_i ∪ processats_f
      retornar reconstruir_camí(i, distàncies_i, predecessors_i,
                                f, distàncies_f, predecessors_f,
                                processats)

  retornar Nul
```

Només ens queda concretar què fa la funció `reconstruir_camí` per obtenir el camí més curt. Intuïtivament podríem pensar que per obtenir el camí més curt entre i i f és tan fàcil com fer la unió del camí més curt des del vèrtex inicial i al vèrtex mitjà m (que haurà trobat la cerca que va endavant) i el camí més curt des del mateix vèrtex mitjà m fins al vèrtex final f (que haurà trobat la cerca que va endarrere).

Tot i això, l'anterior no és correcte, ja que pot ser que el vèrtex m no formi part del camí més curt entre i i f.

Com a exemple, suposem que tenim el graf G de la figura 4.25.

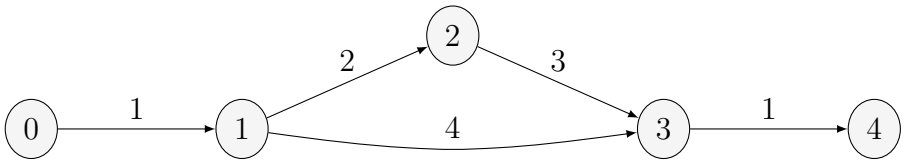

Figura 4.25: Graf G.

És fàcil veure sense la necessitat de cap algorisme que el camí més curt entre el vèrtex $i = 0$ i el vèrtex $f = 4$ és el que va del 0 a l'1, de l'1 al 3 i, finalment, del 3 al 4. En total aquest camí més curt té pes $1 + 4 + 1 = 6$.

No obstant això, si anem alternant l'algorisme de Dijkstra des de les dues bandes, passarà el que es mostra a la llista que hi ha a continuació. Els ítems senars són de la cerca que va endavant des de i i els parells els de la cerca que va endarrere des de f. Al final de cada línia s'indica l'estat de cada cua de prioritat (al començament s'encuen els vèrtexs inicials de cada cerca: $PQ_i = \{(0, 0)\}$ i $PQ_f = \{(4, 0)\}$):

1. Es desencua i es processa el vèrtex 0. $PQ_i = \{(1, 1)\}$
2. Es desencua i es processa el vèrtex 4. $PQ_f = \{(3, 1)\}$
3. Es desencua i es processa el vèrtex 1. $PQ_i = \{(2, 3), (3, 5)\}$
4. Es desencua i es processa el vèrtex 3. $PQ_f = \{(2, 4), (1, 5)\}$
5. Es desencua i es processa el vèrtex 2. $PQ_i = \{(3, 5)\}$
6. Es desencua i es processa el vèrtex 2. $PQ_f = \{(1, 5)\}$

Com que les dues cerques han processat el vèrtex 2, aturarem el bucle en aquest moment. Tot i això, com es pot veure, el camí que passa pel vèrtex 2 no és el més curt, ja que té pes $1 + 2 + 3 + 1 = 7$, i $6 < 7$.

Per tant, com que el vèrtex m no sempre (a vegades sí) formarà part del camí més curt, haurem de buscar quin vèrtex realment és el millor per unir els dos camins, és a dir, haurem de comprovar quin és el vèrtex que realment minimitza la distància dels dos camins. Això ho podrem aconseguir comprovant quin vèrtex u, de tots els processats (com es veu en el pseudocodi, això serà la unió de `processats_i` amb `processats_f`), és el que té menor la suma `distàncies_i[u] + distàncies_f[u]`.

Per exemple, si executéssim `dijkstra_bidireccional(G, 0, 4)` pel graf G anterior de la figura 4.25, la funció `reconstruir_camí` rebria els següents paràmetres:

- $i = 0$
- `distàncies_i` $= \{0, 1, 3, 5, \infty\}$
- `predecessors_i` $= \{\text{Nul}, 0, 1, 1, \text{Nul}\}$
- $f = 4$
- `distàncies_f` $= \{\infty, 5, 4, 1, 0\}$
- `predecessors_f` $= \{\text{Nul}, 3, 3, 4, \text{Nul}\}$
- `processats` $= \{0, 1, 2\} \cup \{4, 3, 2\} = \{0, 1, 2, 4, 3, 2\}$

Com podem observar tots els vèrtexs estan a la llista de processats (no en tots els casos passa); per tant, haurem de comprovar amb tots quin minimitza la distància total. Calculem per cada vèrtex u (marcat pel nombre d'ítem de la llista) el resultat de la suma `distàncies_i[u] + distàncies_f[u]`:

0. $0 + \infty = \infty$
1. $1 + 5 = 6$
2. $3 + 4 = 7$
3. $5 + 1 = 6$
4. $\infty + 0 = \infty$

Podem veure que els vèrtexs que minimitzen la distància són l'1 i el 3. Pot ser que, com en aquest cas, hi hagi més d'un vèrtex que minimitzi la distància però és indistint agafar un o l'altre. En aquest cas agafarem l'1 perquè l'haurem comprovat abans, ja que estava més endavant a la llista de processats.

Ara que ja hem obtingut el vèrtex correcte que unirà les dues cerques, només ens falta reconstruir el camí a partir de la llista de predecessors i obtindrem el camí més curt. El vèrtex que uneix les dues cerques l'anomenarem c. Per reconstruir el camí que ha trobat la cerca que comença des de i, inserirem al principi c, llavors el predecessor de c, després el predecessor del predecessor de c i així fins a arribar al vèrtex inicial. En l'exemple del graf G, aquest camí serà $\{0, 1\}$, ja que com es pot veure per anar des de $i = 0$ fins $c = 1$, només hi ha una aresta.

Llavors farem el mateix per trobar el camí des de c fins a f, tot i que haurem d'inserir els elements en ordre invers (al final de la llista en comptes del principi) perquè aquesta cerca anava endarrere. En l'exemple, s'inserirà el 3 al final de la llista, ja que és el predecessor de $c = 1$, i finalment el 4, ja que és el predecessor del 3. Així, haurem trobat el camí més curt per anar del vèrtex $i = 0$ al vèrtex $f = 4$ en el graf G: $\{0, 1, 3, 4\}$.

El pseudocodi per fer tot això és el següent:

```
algorisme reconstruir_camí(i, distàncies_i, predecessors_i,
                           f, distàncies_f, predecessors_f,
                           processats):
  distància_mínima = ∞
  millor = Nul
  per cada u ∈ processats:
    si distàncies_i[u] + distàncies_f[u] < distància_mínima:
      millor = u
      distància_mínima = distàncies_i[u] + distàncies_f[u]

  camí = {}
  c = millor
  mentre c ≠ Nul:
    inserir c al principi de camí
    c = predecessors_i[c]

  c = millor
  mentre c ≠ f:
    c = predecessor_f[c]
    inserir c al final de camí

  retornar camí
```

Per acabar, podem ajuntar el millor dels dos mons, unir la cerca bidireccional que hem dissenyat per l'algorisme de Dijkstra i la cerca informada a través d'heurístiques. Amb la combinació d'aquestes dues tècniques obtenim un algorisme altament eficient: l'algorisme de cerca A* bidireccional.

Tanmateix, la implementació de l'heurística per l'A* bidireccional és més complexa que la de l'A* unidireccional, ja que perquè es garanteixi que es trobi el camí més curt l'heurística per la cerca que comença a i ha de ser $h_i(v) = (\pi_i(v) - \pi_f(v))/2$ i per la cerca que comença en f ha de ser $h_f(v) = (\pi_f(v) - \pi_i(v))/2 = -h_i(v)$, on $\pi_i(v)$ és una funció que estima la distància entre v i f, i $\pi_f(v)$ és una funció que estima la distància entre v i i. Això farà que també s'hagi d'utilitzar una altra llista per guardar els valors calculats per cada vèrtex, per així fer coincidir que $h_i(v) = -h_f(v)$. En últim lloc, com en l'A* unidireccional, hauríem de sumar en cada iteració els valors de les heurístiques a les prioritats en què s'encuen els vèrtexs a les cues de prioritats.

La justificació de les fórmules utilitzades per les heurístiques queda fora de l'abast d'aquest treball, però es pot trobar en la bibliografia utilitzada [Ikeda et al. 1994].

Gràcies a les heurístiques, la cerca que comença pel vèrtex i anirà ràpidament en direcció el vèrtex f, mentre que passarà al revés per la cerca que comença des de f, la qual anirà de pressa en direcció el vèrtex i. D'aquesta manera, l'algorisme de cerca A* bidireccional trobarà abans el vèrtex mitjà, cosa que farà que en la gran majoria dels casos es processin un menor nombre de vèrtexs que en la resta d'algorismes presentats; per tant, sigui molt més ràpid a trobar el camí més curt.

"Un viatge de mil milles comença amb una sola passa i, si aquesta és la correcta, esdevé l'última."

— Laozi, pensador xinès, segle VI aC.

5

Desenvolupant un planificador de rutes pel meu poble

A partir dels algorismes exposats en el capítol anterior he desenvolupat un planificador de rutes pel meu poble, Calldetenes, un municipi de la comarca d'Osona. Un planificador de rutes és un programa informàtic dissenyat per trobar la millor ruta, normalment per carretera, entre dos punts del món. En el meu planificador, considerarem que la millor ruta és la que passa pel camí més curt. El planificador de rutes que desenvoluparé, com la majoria de programes semblants, també proporciona un mapa interactiu amb la ruta suggerida marcada.

A part, també he implementat un visualitzador integrat en el programa que permet veure el funcionament intern de l'algorisme mentre busca la ruta més curta. Aquest visualitzador permetrà veure molt bé el progrés dels algorismes mentre exploren el graf.

El resultat final ha estat una aplicació web a la qual tothom pot accedir obrint en un navegador web l'adreça `caminscalldetenes.cat`. Tot el codi font que he escrit per aquest projecte és obert i està disponible a `github.com/salcc/CaminsCalldetenes` sota una llicència de programari lliure. He comentat bastant el codi font amb la intenció que qualsevol persona sense gaires coneixements sobre programació pugui entendre quin és el seu funcionament.

El codi font de la part interna del programa que s'encarrega de processar el mapa i trobar el camí més curt està escrit en el llenguatge de programació Python. La part de la interfície visible a través de la web està dissenyada amb HTML[1] i CSS[2], i el funcionament del mapa interactiu i els botons estan programats en JavaScript.

5.1 Classe per representar grafs en Python

Abans de poder implementar els algorismes per trobar el camí més curt, la part principal del programa, he hagut d'implementar una classe per representar grafs, ja que com hem vist en el capítol anterior tots aquests algorismes treballen amb grafs. En programació, una *classe* s'utilitza per crear estructures de dades personalitzades,

[1] Sigles en anglès de *HyperText Markup Language* (llenguatge de marques d'hipertext).

[2] Sigles en anglès de *Cascading Style Sheets* (fulls d'estil en cascada). Serveix per descriure l'aspecte d'un document escrit en HTML.

com és el cas d'un graf. Una classe també pot contenir mètodes per operar amb les dades que emmagatzema.

Internament la classe que he dissenyat per representar grafs està formada per una llista d'adjacència i un diccionari que es pot fer servir per guardar atributs dels vèrtexs i les arestes.

Amb aquesta classe es poden crear grafs dirigits. Tot i això, també es pot crear un graf no dirigit si a l'hora de crear una aresta, en realitat afegim dues arestes antiparal·leles. És a dir, si volem crear un graf no dirigit a l'hora d'afegir una aresta (u, v) haurem d'afegir una aresta (u, v) i una aresta (v, u).

La classe també té diversos mètodes per poder modificar el graf i obtenir-ne informació. Els mètodes que té són els següents:

`afegir_vertex(**atributs)`
: Afegeix un vèrtex al graf.

  El paràmetre `atributs` és opcional i serveix per assignar atributs al vèrtex que s'està afegint.

`vertexs()`
: Retorna una llista de tots els vèrtexs del graf.

`ordre()`
: Retorna l'ordre (el nombre de vèrtexs) del graf.

`afegir_aresta(aresta, **atributs)`
: Afegeix una aresta al graf.

  El paràmetre `aresta` serveix per indicar quins dos vèrtexs connecta l'aresta. Per exemple pot ser `(0, 1)`, per indicar que l'aresta que s'està afegint va del vèrtex 0 al vèrtex 1.

  El paràmetre `atributs` és opcional i serveix per donar un atribut a l'aresta que s'està afegint.

`arestes()`
: Retorna una llista de totes les arestes del graf.

`mida()`
: Retorna la mida (el nombre d'arestes) del graf.

`llegir_atributs(vertex_o_aresta)`
: Retorna el diccionari d'atributs del vèrtex o aresta indicat en el paràmetre `vertex_o_aresta`.

`assignar_atributs(vertex_o_aresta, **atributs)`
: Assigna els atributs especificats en el paràmetre `atributs` al diccionari d'atributs del vèrtex o aresta indicat en el paràmetre `vertex_o_aresta`.

A continuació es pot veure un exemple de com es pot utilitzar la classe per crear un graf com el de la figura 5.1.

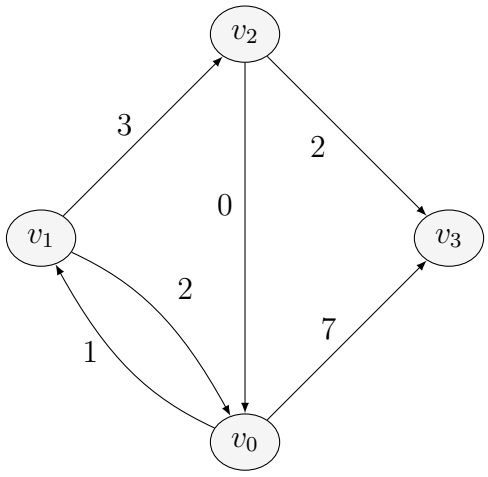

Figura 5.1: Diagrama del graf dirigit G.

```python
# Creem el graf, incialment buit
G = GrafDirigit()

# Afegim els quatre vèrtexs
for i in range(4):
  G.afegir_vertex()

# Afegim les arestes amb
# l'atribut "pes"
G.afegir_aresta((0, 1), pes=1)
G.afegir_aresta((1, 0), pes=2)
G.afegir_aresta((1, 2), pes=3)
G.afegir_aresta((2, 0), pes=0)
G.afegir_aresta((2, 3), pes=2)
G.afegir_aresta((0, 3), pes=7)
```

Figura 5.2: Codi per crear el graf dirigit G de la figura de l'esquerre.

Ara podem utilitzar els altres mètodes de la classe per obtenir informació sobre el graf:

```python
print("L'ordre del graf G és", G.ordre())
print("El graf G conté els següents vèrtexs:", G.vertexs())

print("La mida del graf G és", G.mida())
print("El graf G conté les següents arestes:", G.arestes())

print("El pes de l'aresta (0, 3) és", G.llegir_atributs((0, 3))["pes"])
```

Sortida:

L'ordre del graf G és 4

El graf G conté els següents vèrtexs: $[0, 1, 2, 3]$

La mida del graf G és 6

El graf G conté les següents arestes: $[(0, 1), (0, 3), (1, 0), (1, 2), (2, 0), (2, 3)]$

El pes de l'aresta $(0, 3)$ *és* 7

El codi complet de la classe per representar grafs en Python està disponible a `github.com/salcc/CaminsCalldetenes/blob/master/graf.py`.

5.2 Processament de dades d'OpenStreetMap

Ara que ja podem crear grafs falta tenir les dades per crear un graf dels carrers i carreteres que formen Calldetenes. Crear un graf des de zero que representi les ubicacions del món de forma precisa és una feina bastant complicada i que requereix molt temps, sobretot per una sola persona. Faria falta recórrer tots els carrers i

anotar les coordenades geogràfiques de les interseccions amb ajuda d'algun aparell de geolocalització, o bé utilitzar alguna imatge aèria del territori on tots els elements presentessin la mateixa escala (una ortofoto) i amb ajuda d'algun programa informàtic o manualment marcar les interseccions dels carrers i obtenir-ne les coordenades. A partir de les coordenades de les interseccions es podrien crear els vèrtexs del graf. Llavors es crearien les arestes a partir dels carrers que connecten les interseccions. A part també s'haurien de tenir en compte si un carrer és d'una sola direcció o si només és per a vianants.

Per sort, existeix OpenStreetMap[3], un projecte col·laboratiu per crear un mapa gratuït i editable de tot el món. Aquest projecte permet que qualsevol persona modifiqui el mapa de la terra per afegir informació, carrers, etc. L'objectiu del projecte és proporcionar les dades del mapa sota una llicència oberta a qualsevol que les necessiti.

OpenStreetMap proporciona un editor visual del mapa que tothom pot utilitzar i fa molt fàcil i ràpid actualitzar dades que han canviat, per exemple si s'ha construït una nova carretera. Gràcies a això, hi ha zones del mapa que estan més detallades i actualitzades que en mapes d'altres serveis no oberts com és el cas de Google Maps, Bing Maps o Yandex.Maps. Aquest és el cas de la zona de Calldetenes i els seus voltants, ja que algunes coses que estaven desactualitzades les he pogut actualitzar jo mateix. Una contribució important que he fet és corregir diverses coordenades i direccions errònies que hi havia a causa que recentment s'ha construït una variant de Calldetenes, s'han creat rotondes i s'han modificat les direccions d'alguns carrers.

Tot i això, OpenStreetMap no proporciona directament un graf amb les dades de tot el mapa del món sinó que permet exportar una àrea del mapa en format OSM XML. A la figura 5.3 es mostra un fragment d'un fitxer OSM XML.

El que ens interessa del fitxer OSM XML són els elements `node` i els elements `way`.

Els elements `node` defineixen una intersecció o un punt d'inflexió d'un carrer. Cada element `node` té, entre altres coses, un identificador únic guardat en l'atribut `id` i unes coordenades geogràfiques en els atributs `lat` i `lon`, que són la latitud i la longitud respectivament.

Per altra banda, els elements `way` defineixen les vies (carrers, carreteres, etc.). Cada element `way` té una llista de subelements `nd` que referencien amb el seu identificador als nodes que formen aquesta via. Els elements `way` també contenen altres subelements `tag` que especifiquen si és d'una sola direcció, el tipus de via (si és un corriol, un carrer, una autopista...), el nom, etc.

Per convertir d'OSM XML a un graf he escrit un programa en Python. El codi complet d'aquest convertidor es troba a `github.com/salcc/CaminsCalldetenes/blob/master/generar_graf.py`.

El graf obtingut per al planificador de rutes de Calldetenes, té 16.975 vèrtexs connectats per 31.643 arestes dirigides.

[3]URL: `www.openstreetmap.org`.

```
<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.6" generator="CGImap 0.7.5 (32275 thorn-02.
   ↪ openstreetmap.org)" copyright="OpenStreetMap and contributors"
   ↪ attribution="http://www.openstreetmap.org/copyright" license="
   ↪ http://opendatacommons.org/licenses/odbl/1-0/">
  <bounds minlat="41.8981000" minlon="2.2347000" maxlat="41.9520000"
     ↪ maxlon="2.3554000"/>
  <node id="33960681" visible="true" version="3" changeset="73211125"
     ↪ timestamp="2019-08-10T01:08:27Z" user="SerV" uid="708973" lat=
     ↪ "41.8925787" lon="2.3622115"/>
  <node id="33960683" visible="true" version="3" changeset="73211125"
     ↪ timestamp="2019-08-10T01:08:27Z" user="SerV" uid="708973" lat=
     ↪ "41.8935040" lon="2.3519968"/>
  <node id="33960684" visible="true" version="5" changeset="22052793"
     ↪ timestamp="2014-04-30T21:42:14Z" user="EliziR" uid="605366"
     ↪ lat="41.8940870" lon="2.3491751"/>
  <node id="33960685" visible="true" version="3" changeset="73211125"
     ↪ timestamp="2019-08-10T01:08:27Z" user="SerV" uid="708973" lat=
     ↪ "41.8954133" lon="2.3461026"/>
  ...
  <way id="43770743" visible="true" version="4" changeset="70830052"
     ↪ timestamp="2019-06-01T08:03:11Z" user="monyrojasv" uid="
     ↪ 9339152">
    <nd ref="554510845"/>
    <nd ref="554510840"/>
    <nd ref="554510850"/>
    <tag k="highway" v="residential"/>
    <tag k="name" v="Carrer de Carles Riba"/>
    <tag k="surface" v="asphalt"/>
  </way>
  <way id="43770876" visible="true" version="5" changeset="72863858"
     ↪ timestamp="2019-07-31T16:08:06Z" user="salcc" uid="10135230">
    <nd ref="554537030"/>
    <nd ref="6666326647"/>
    <nd ref="554537076"/>
    <tag k="highway" v="living_street"/>
    <tag k="name" v="Passeig del Canigó"/>
    <tag k="oneway" v="yes"/>
    <tag k="surface" v="asphalt"/>
  </way>
  ...
</osm>
```

Figura 5.3: Fragment d'un fitxer OSM XML.

El programa crea un graf i afegeix a aquest graf un vèrtex per cada element `node` del fitxer OSM XML. Cada vèrtex afegit també guarda amb un atribut les coordenades on es troba en el món real. Llavors, per cada parell d'elements `nd` de les vies que són transitables en cotxe, s'afegeix al graf una aresta si el carrer és d'una sola direcció o, si no, dues arestes antiparal·leles. Per exemple, si una via està formada pels nodes amb identificadors 1, 2, i 3, s'afegeixen dues arestes $(1, 2)$ i $(2, 3)$ en cas que sigui d'una sola direcció, o quatre arestes $(1, 2)$, $(2, 3)$, $(3, 2)$, $(2, 1)$ en cas contrari.

Finalment només queda afegir a les arestes creades un atribut que guardi la longitud de l'aresta al món real, que s'utilitzarà com a pes perquè els algorismes puguin trobar el camí més curt. Per fer això, ho fem amb la fórmula del semiversinus, explicada a continuació.

5.3 La fórmula del semiversinus

La *fórmula del semiversinus* és una fórmula matemàtica per determinar la distància ortodròmica entre dos punts d'una esfera donades les seves longituds i latituds. La distància ortodròmica és la distància que s'obté quan es mesura al llarg de la superfície de l'esfera (en contraposició a una línia recta que travessés per l'interior de l'esfera).

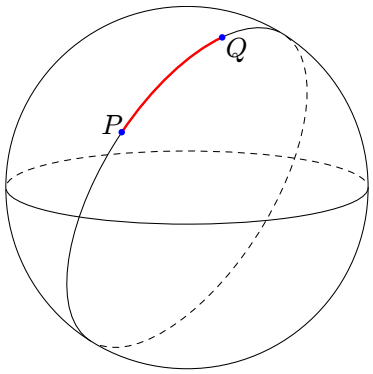

Figura 5.4: Distància ortodròmica entre dos punts P i Q.

És una de les fórmules més utilitzades per calcular distàncies entre dos punts de la Terra, ja que dona una bona aproximació de la distància real. Tot i això, no és del tot exacta perquè la Terra no és una esfera perfecta, ja que el valor del seu radi oscil·la entre 6.378 km a l'equador i 6.357 km en els pols i, a més, té accidents geogràfics que creen relleu.

La fórmula s'anomena així perquè utilitza una funció trigonomètrica que es diu *semiversinus*, que és la meitat d'una altra funció trigonomètrica anomenada *versinus*. Les definicions d'aquestes dues funcions són les següents:

$$\operatorname{versin}(\alpha) = 1 - \cos(\alpha) = 2\sin^2\left(\frac{\alpha}{2}\right)$$

$$\operatorname{semiversin}(\alpha) = \frac{\operatorname{versin}(\alpha)}{2} = \sin^2\left(\frac{\alpha}{2}\right)$$

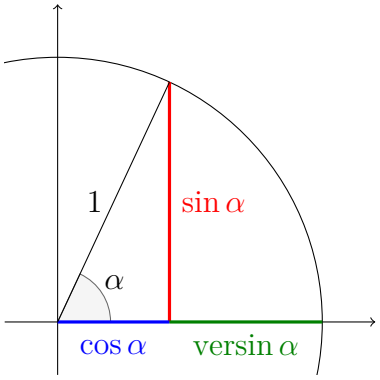

Figura 5.5: Sinus, cosinus i versinus de l'angle α en un cercle (retallat) de radi 1.

Com totes les funcions trigonomètriques, el versinus i el semiversinus també tenen funcions inverses:

$$\operatorname{arcversin}(y) = \arccos(1 - y)$$

$$\operatorname{arcsemiversin}(y) = 2\arcsin(\sqrt{y})$$

Sigui θ l'angle entre dos punts en una esfera:

$$\theta = \frac{d}{r}$$

on d és la distància ortodròmica i r el radi de l'esfera.

La fórmula del semiversinus permet calcular el semiversinus de l'angle θ directament a partir de la latitud i la longitud dels dos punts:

$$\operatorname{semiversin}(\theta) = \operatorname{semiversin}(\phi_2 - \phi_1) + cos(\phi_1)\cos(\phi_2)\operatorname{semiversin}(\lambda_2 - \lambda_1)$$

on ϕ i λ denoten en radians la latitud i la longitud de cada punt respectivament.

Per trobar la distància, s'utilitza l'arcsemiversinus per aïllar d:

$$d = r\,\text{arcsemiversin}(\text{semiversin}(\phi_2 - \phi_1) + cos(\phi_1)\cos(\phi_2)\,\text{semiversin}(\lambda_2 - \lambda_1))$$

La fórmula del semiversinus va facilitar molt la feina dels navegants abans que s'inventessin les calculadores electròniques. Com que les funcions trigonomètriques usades tenen un domini limitat (el cosinus i el semiversinus entre $-\pi$ i π, i l'arcsemiversinus entre 0 i 1), es podien fer taules de les funcions trigonomètriques que donessin valors precalculats aproximats fent així el càlcul molt més simple. Per poder calcular la distància d, un navegant només havia de buscar en la taula dos valors del cosinus i dos valors del semiversinus, sumar-los i multiplicar-los de la manera correcta, buscar el valor de l'arcsemiversinus en la taula i finalment multiplicar-ho per r (el radi de la Terra).

Tot i això, si volem programar la fórmula en un llenguatge de programació com Python, que no implementa les funcions semiversinus i arcsemiversinus, és convenient expressar la fórmula només utilitzant sinus, cosinus, arcsinus i arrel quadrada, funcions més usades actualment que Python sí que implementa:

$$d = 2r\arcsin\left(\sqrt{\sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + cos(\phi_1)\cos(\phi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$$

La fórmula del semiversinus, a part d'utilitzar-la per calcular quant mesura cada aresta en el món real, també l'utilitzarem per implementar la funció heurística de l'algorisme de cerca A*.

5.4 Implementació dels algorismes per trobar el camí més curt

Els algorismes per trobar el camí més curt són la base per fer funcionar el programa. Pel planificador he implementat l'algorisme de Dijkstra, el de cerca A*, el de Dijkstra bidireccional i el d'A* bidireccional. El codi font de la implementació dels algorismes es pot trobar al repositori de GitHub a `github.com/salcc/CaminsCalldetenes/blob/master/cami_mes_curt.py` i a `github.com/salcc/CaminsCalldetenes/blob/master/utils.py` on hi ha les funcions `reconstruir_camí`. També he implementat l'algorisme de cerca en amplada, tot i que no s'utilitza en el planificador, ja que no funciona en grafs ponderats com el que tenim.

L'únic que no estava especificat en el capítol anterior sobre l'algorisme de cerca A* és quina heurística s'utilitza per al planificador de rutes. Cal notar que la funció heurística varia depenent de l'aplicació i aquí s'explica l'heurística utilitzada pel meu planificador de rutes.

En el cas d'un planificador de rutes pel món real, podem crear una heurística si ens adonem que el camí que va cap al vèrtex final normalment va en direcció cap al vèrtex final en el món físic. És a dir, normalment una carretera que va des d'una

ciutat A a una ciutat B de la manera més curta sol anar més o menys recta des de A fins a B com es pot veure en la següent figura:

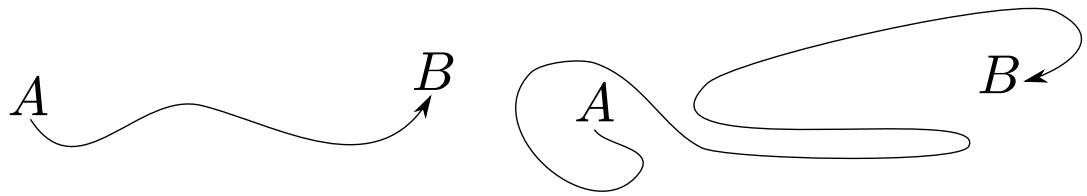

Figura 5.6: A l'esquerra, representació de com sol ser la trajectòria de la carretera més curta possible entre dues ciutats A i B en el món real. A la dreta, com no sol ser (és clar que pot existir una carretera així, però segurament no serà la més curta).

A partir d'aquesta conclusió podem crear una heurística que faci que es processin abans els vèrtexs que són més propers al vèrtex final.

És a dir, l'heurística serà $h(v) = d$, on d serà la distància ortodròmica entre el vèrtex v i el vèrtex final del camí f. Aquesta distància la podrem calcular gràcies a la fórmula del semiversinus. Convé ressaltar que aquesta distància no és la distància que fa el camí més curt entre v i f (ja que aquí el camí més curt és el que estem buscant; per tant, encara no el sabem), sinó que és la distància mínima, com si passéssim per una carretera totalment recta que s'anés adaptant a la curvatura terrestre.

Així, si $h(v) = d$ és l'heurística que fem servir, haurem aconseguit crear una heurística que estimi la distància al vèrtex final. A més, com que la distància ortodròmica entre v i f sempre serà menor o igual que la distància del camí més curt entre v i f, ja que el més curt que pot ser un camí és una línia recta (és a dir, la distància ortodròmica), l'heurística mai sobreestimarà. Gràcies a això, haurem obtingut una heurística admissible i es garantirà que sempre trobem el camí més curt.

En el cas de la cerca bidireccional, farem servir el mateix: $\pi_i(v)$ serà igual a la distància ortodròmica entre les coordenades de v i les de f, i π_f serà igual a la distància ortodròmica entre les coordenades de v i les de i.

5.5 Creació de l'aplicació web

Per crear l'aplicació web he utilitzat el *micro web framework* de codi obert Flask. Un *web framework* permet que el codi que s'està executant en el navegador web es pugui comunicar amb el codi que s'està executant al servidor. S'anomena *micro* perquè és molt lleuger i només proporciona les funcions més bàsiques.

En el cas del planificador de rutes que he desenvolupat, el codi que fa funcionar la interfície gràfica s'executa en el navegador web de l'usuari que accedeix a la pàgina. Tanmateix, en interaccionar amb la pàgina, el codi envia al servidor web les dades perquè es pugui computar el camí més curt. Llavors, en el servidor és on s'executaran els algorismes per buscar el camí més curt, els quals finalment retornaran el resultat al navegador web perquè es mostri en la pàgina.

Que el camí més curt es computi en el servidor web té el gran avantatge que el graf, que ha d'estar carregat en la memòria de l'ordinador perquè els algorismes puguin funcionar, només s'hagi de carregar en el servidor web i no en l'ordinador de cada usuari que accedeix a la pàgina. Si tot passés al navegador, tot i que el servidor gairebé no necessitaria potència, l'usuari que visités la web rebria una experiència lenta perquè tot s'estaria processant en el seu ordinador.

En la mateixa línia, un altre avantatge és que d'aquesta manera el graf només s'ha de guardar en el servidor, fet que fa que la pàgina es carregui més de pressa, ja que no s'ha d'enviar el graf. Això també fa que el servidor i l'usuari no requereixin tanta amplada de banda.

Finalment, un últim avantatge d'utilitzar un web framework és que en el servidor no hi ha la necessitat d'utilitzar un llenguatge de programació web com JavaScript. En el meu cas, gràcies a això he pogut implementar els algorismes per trobar el camí més curt en Python, cosa que els fa més entenedors i senzills d'implementar.

Pel que fa al servidor, he fet servir el servei de Microsoft Azure. Azure és un servei de computació en el núvol de Microsoft que permet, entre moltes altres coses, allotjar i executar aplicacions web en el núvol. Una aplicació web es distingeix d'una pàgina web tradicional en el fet que una aplicació web també executa codi en el núvol. Això era una necessitat que tenia pels motius explicats anteriorment.

Vaig triar fer servir Microsoft Azure perquè proporcionava totes les funcions que necessitava, que bàsicament és que pogués executar codi en Python i, principalment, perquè el servei, amb un servidor bastant bo, és gratuït per estudiants.

Una cosa que sí que vaig haver de pagar és el domini de `caminscalldetenes.cat`, que vaig comprar a una empresa anomenada DonDominio. També vaig decidir comprar un certificat SSL que fa que la web sigui més segura, ja que s'utilitza HTTPS. A més, les pàgines que utilitzen HTTPS també solen sortir més amunt als resultats dels cercadors web. Realment, HTTPS no era del tot necessari per la meva pàgina perquè no processa dades personals, però m'interessava veure com funcionava l'activació d'un certificat SSL per saber-ho pel futur.

Vaig triar l'empresa DonDominio per comprar el domini i el certificat SSL perquè era una de les més barates i oferia el domini .cat.

Les despeses totals estan resumides a continuació:

Registre del domini	9,95€
Correu electrònic	0,99€
Certificat SSL	5,95€
Subtotal	16,89€
IVA (21%)	3,55€
Total	20,44€

Finalment, pel que fa a la interfície gràfica d'usuari de la pàgina web, la vaig desenvolupar utilitzant HTML, CSS i JavaScript. La interfície consta de dues parts: un panell lateral a l'esquerra, amb els controls, i un visor pel mapa i el camí.

Per mostrar el mapa vaig utilitzar una llibreria de JavaScript, també de codi obert, anomenada Leaflet. Aquesta llibreria proporciona el visor de mapes interactiu que permet interaccionar amb ell, per exemple moure's, apropar-se, etc. També proporciona les funcions que permeten dibuixar línies en el mapa, les quals utilitzo per dibuixar el camí més curt.

La capa topogràfica que es veu és d'OpenStreetMap, el mateix projecte que proporciona les dades que m'han permès crear un graf. Per altra banda, també es pot canviar la capa per veure una capa d'ortofoto en comptes de la topogràfica. La capa ortofoto, que ha estat proporcionada per l'Institut Cartogràfic i Geològic de Catalunya (ICGC), mostra com és a la realitat des de l'aire sense cap text o iconografia. La capa es pot canviar clicant el botó de la part superior dreta del visor.

Ara que ja sabem tots els components implicats en el planificador, podem explicar què passa des que un usuari accedeix a la web fins que veu el camí més curt entre dos punts en el mapa:

1. L'usuari entra a la pàgina web a través del navegador accedint pel domini `caminscalldetenes.cat`.
2. L'usuari col·loca els dos marcadors en el mapa, que es visualitza gràcies a Leaflet i les capes d'OpenStreetMap i l'ICGC. El marcador verd marca l'inicial i el vermell el final.
3. El navegador web de l'usuari envia al servidor les coordenades dels marcadors.
4. El servidor, gràcies a Flask, rep les coordenades enviades pel navegador.
5. En el servidor, es busca el vèrtex del graf més proper a les coordenades del marcador inicial i el més proper a les del final. Aquests dos vèrtexs seran l'inicial i i el final f.
6. Seguint en el servidor, l'algorisme de cerca A* bidireccional busca el camí més curt des de i fins a f.
7. El servidor, utilitzant Flask, retorna al navegador web les coordenades de cada parell de vèrtexs que formen una aresta.
8. En el navegador, Leaflet dibuixa línies entre cada parell de coordenades. El resultat final serà que quedarà el camí més curt dibuixat amb una línia blava.
9. L'usuari pot veure quin és el camí més curt entre els dos marcadors.

Al final del procés, l'usuari pot canviar la posició dels marcadors i es tornaria al tercer punt.

Prèviament, en el servidor s'haurà d'haver processat un fitxer OSM XML per obtenir el graf. Aquest graf no cal que el toquem més, excepte si volem actualitzar les dades.

També cal notar que aquí només s'utilitza la cerca A* bidireccional, ja que sol ser l'algorisme més ràpid. Tanmateix, en la part que visualitza els algorismes, podrem utilitzar els quatre algorismes i veure com operen.

5.6 Desenvolupament del visualitzador d'algorismes

El visualitzador d'algorismes es pot activar a través del panell lateral de Camins Calldetenes. Si s'activa, es pot triar quin algorisme utilitzar per computar el camí més curt, veure el nombre de vèrtexs que es visita i, el més interessant, visualitzar com funciona l'algorisme mentre busca el camí més curt en el mapa.

Quan el visualitzador d'algorismes està activat, el navegador també enviarà al servidor que desitja totes les coordenades necessàries per fer la visualització, juntament amb l'algorisme que es vol utilitzar.

En el servidor, s'utilitza l'algorisme que l'usuari ha triat en el navegador i en cada iteració de l'algorisme, es guarden les coordenades del vèrtex que es desencua i les coordenades dels vèrtexs que es visiten des del vèrtex desencuat. Connectant les coordenades, obtindrem les arestes que podrem dibuixar amb línies en el mapa.

En el cas dels algorismes de Dijkstra i A* unidireccionals, el visualitzador fa servir, per una part, línies verd clar per marcar les arestes que porten als vèrtexs u, cap als quals ja s'ha trobat el camí més curt. Com que ja s'ha trobat el camí més curt, les línies verd clar seran permanents. Per altra part, fa servir línies verd fosc per marcar les arestes que van als vèrtexs v que es visiten i s'encuen perquè són adjacents a u.

En el cas dels algorismes de Dijkstra i A* bidireccionals, es fa el mateix que en els unidireccionals per la cerca que comença pel vèrtex inicial i es marca la cerca que comença pel vèrtex final del camí amb línies vermelles clar i fosc, que tenen el mateix significat que en els casos anteriors però es refereixen a la cerca que va cap endarrere.

Finalment, quan el camí més curt entre els dos marcadors es troba, aquest es dibuixa amb línies blaves. Aquestes línies blaves tenen una opacitat del 70% per no tapar del tot les línies verdes o vermelles per les quals passa el camí més curt quan es troba.

Realment, la visualització no és a temps real, ja que primer es computa tot el camí més curt entre els dos punts i tots els vèrtexs que s'exploren en el procés, tot guardant totes les coordenades per poder dibuixar les arestes, i llavors s'envien aquestes coordenades al servidor. Això té el desavantatge que pot arribar a passar un segon entre que es col·loquen els marcadors i es comença la visualització, però té l'avantatge que només s'ha de fer una petició al servidor i un cop es rep la resposta amb totes les dades per la visualització, aquesta es pot fer de manera fluida i amb la possibilitat de controlar la velocitat en què es mostra.

El codi de la modificació que he fet als algorismes per guardar la visualització es pot trobar a `github.com/salcc/CaminsCalldetenes/blob/master/cami_mes_curt_visual.py` i el codi que s'encarrega de dibuixar els camins forma part del codi en JavaScript que fa funcionar la pàgina completa en la part del navegador i es pot veure a `github.com/salcc/CaminsCalldetenes/blob/master/static/script.js`.

"La majoria, si no totes, de les grans idees de les matemàtiques modernes han tingut el seu origen en l'observació."

— James Joseph Sylvester, matemàtic anglès, 1870.

6

Anàlisi del programa desenvolupat

Primer de tot, aquest capítol explica el funcionament fonamental de Camins Calldetenes[1], el planificador de rutes desenvolupat. A continuació s'explica el funcionament del visualitzador d'algorismes i algunes coses que es poden veure en el visualitzador. Finalment, es compara Camins Calldetenes amb Google Maps[2], que és el planificador de rutes més avançat i utilitzat del món. Aquesta última part ha estat possible gràcies a l'ajuda d'un matemàtic de Google que ha treballat en el desenvolupament de Google Maps.

6.1 Funcionament bàsic

La pàgina de Camins Calldetenes té una estructura bastant senzilla. Consta de dues parts, el panell lateral a l'esquerra i el visor de mapes que mostra el camí més curt i les visualitzacions. El visor també té un botó a la part superior dreta que permet canviar entre la capa topogràfica i la capa d'ortofoto del mapa.

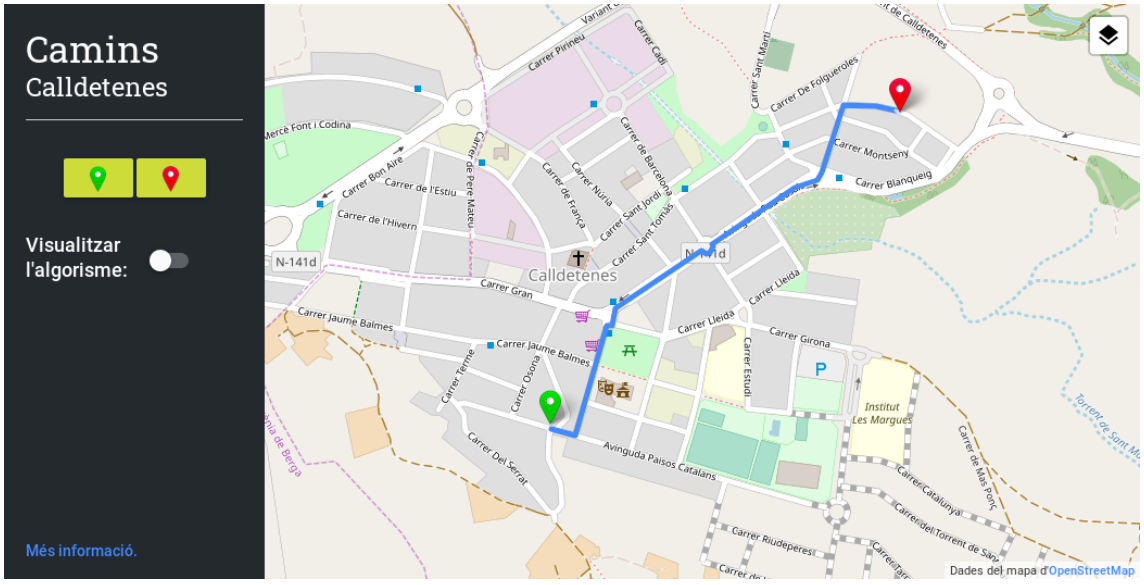

Figura 6.1: Camins Calldetenes mostrant el camí més curt que va des de l'avinguda dels Països Catalans, 17 fins al carrer Jacint Verdaguer, 28.

[1]URL: `caminscalldetenes.cat`
[2]URL: `maps.google.com`

En el panell lateral, hi ha els dos botons principals: el marcador inicial, de color verd, i el marcador final, de color vermell. En clicar-los, permetrà col·locar els marcadors en el visor de mapes. Tan bon punt es detecti que hi ha dos marcadors establerts en el mapa, es buscarà el camí més curt entre els dos punts i es mostrarà. Finalment, en el panell lateral hi ha un altre control, que permet mostrar o no els controls per la visualització d'algorismes i farà que s'utilitzi un algorisme en concret i es mostri en el visor de mapes el procés que fa servir per buscar el camí més curt.

6.2 Funcionament del visualitzador d'algorismes

Si activem l'interruptor per visualitzar els algorismes, s'obren més controls i es visualitzaran els algorismes que busquen el camí més curt en el mapa. La visualització mostrarà com funciona l'algorisme que busca el camí més curt pas a pas en el mapa. És un tipus de visualització que no he vist en cap altre lloc i que ens permet observar diverses coses interessants.

Quan s'activa el visualitzador, el panell lateral permet escollir quin dels quatre algorismes implementats utilitzar. En les quatre figures següents (de la figura 6.2 a la 6.5) es mostra com es visualitzen els algorismes quan han acabat la cerca pel camí més curt que va des de l'avinguda dels Països Catalans, 17 fins al carrer Jacint Verdaguer, 28.

En el panell lateral (no inclòs en les imatges perquè es vegi millor la visualització), també es mostra el nombre de vèrtexs visitats per cada algorisme mentre s'està buscant el camí i quan ha acabat. Per buscar el camí més curt entre els dos marcadors establerts, obtenim els següents nombres:

Algorisme utilitzat	**Vèrtexs visitats**
Dijkstra	539
Cerca A*	124
Dijkstra bidireccional	296
Cerca A* bidireccional	88

Podem veure clarament en els nombres de vèrtexs visitats i també mirant les visualitzacions que les heurístiques i la cerca bidireccional redueixen considerablement el nombre de vèrtexs visitats, cosa que indica que els algorismes són més eficients.

En les visualitzacions també es veu com en les dues versions de Dijkstra s'explora cap a totes bandes indistintament, ja que com que no s'utilitzen heurístiques, l'algorisme no té cap informació de cap a on ha de buscar. En canvi, en les versions de cerca A*, es veu com, tot i que no es va directament cap al vèrtex final, sí que es busca cap en la seva direcció i gairebé no s'explora cap a les altres.

També podem veure que gràcies a la cerca bidireccional, l'àrea explorada per l'algorisme de Dijkstra es redueix molt, gairebé a la meitat. En el cas de l'A*, fer servir la cerca bidireccional fa que el nombre de vèrtexs, que ja és molt menor que en l'algorisme de Dijkstra, encara es redueixi més.

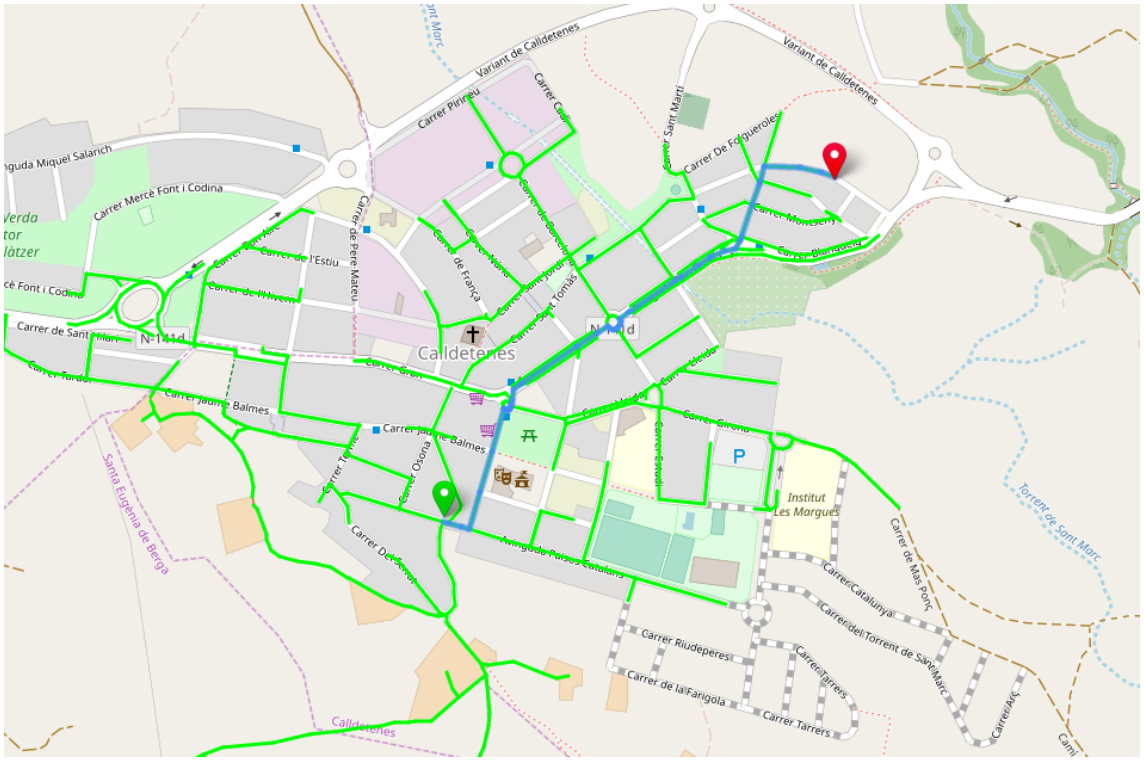

Figura 6.2: Visualització de la cerca amb l'algorisme de Dijkstra.

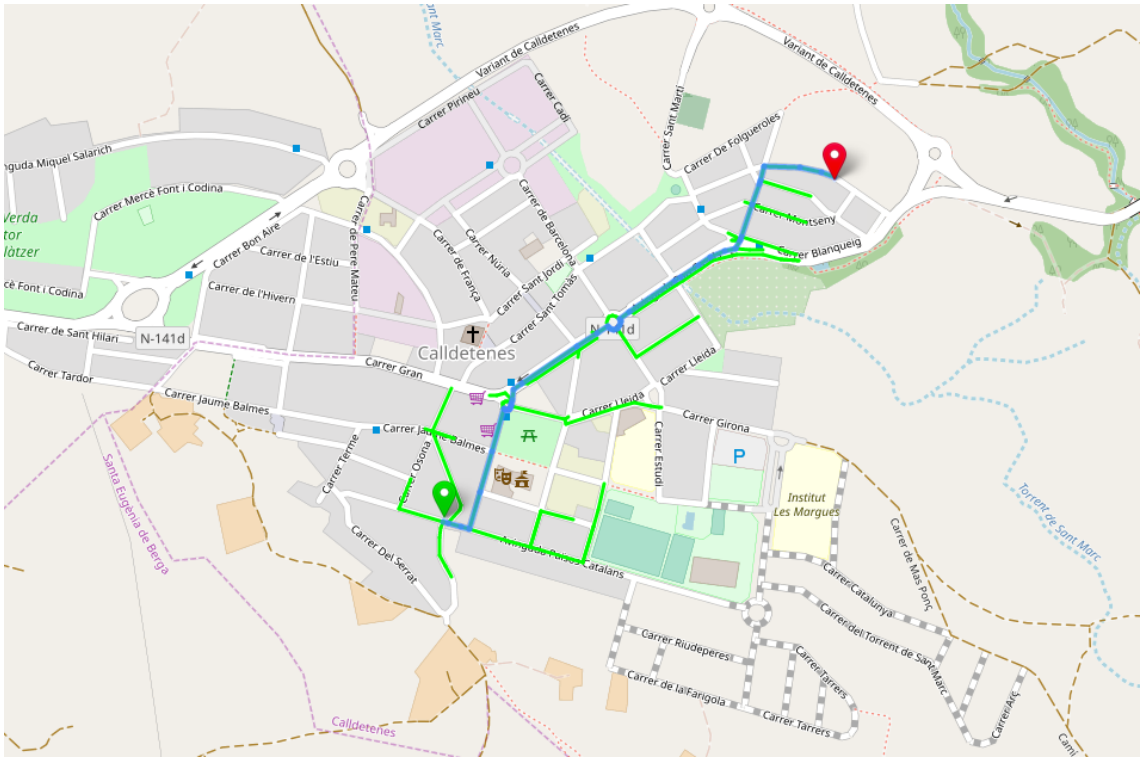

Figura 6.3: Visualització de la cerca amb l'algorisme de cerca A*.

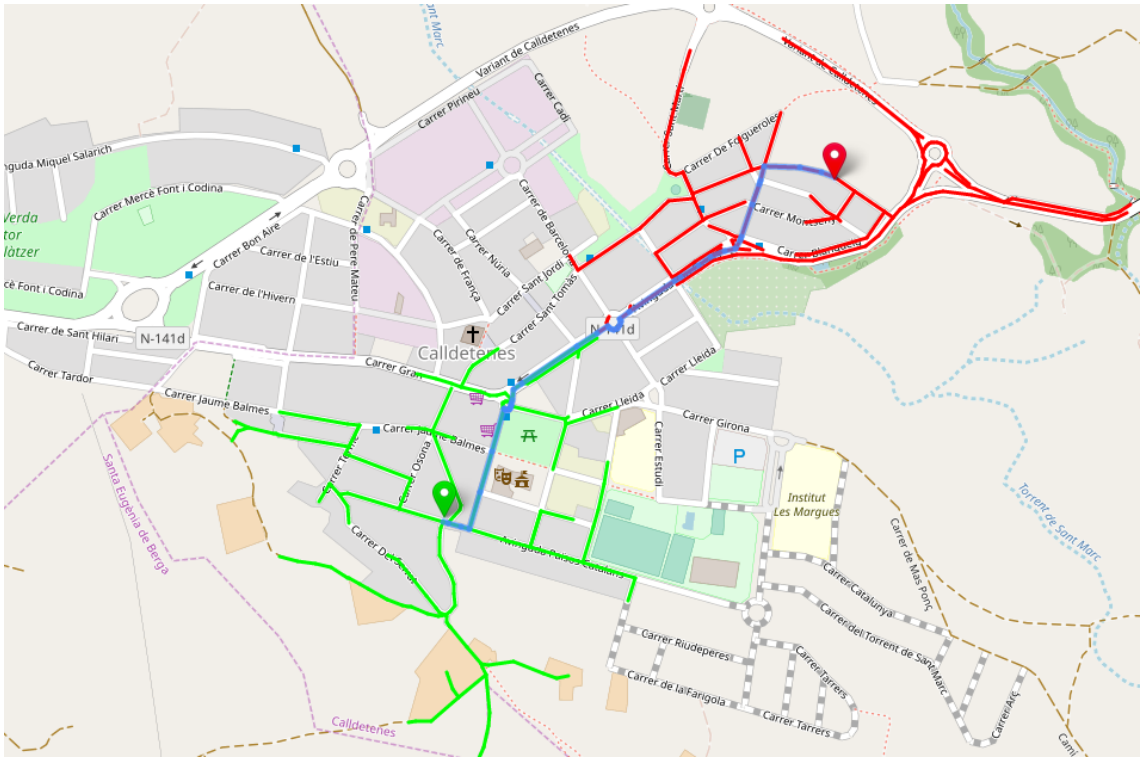

Figura 6.4: Visualització de la cerca amb l'algorisme de Dijkstra bidireccional.

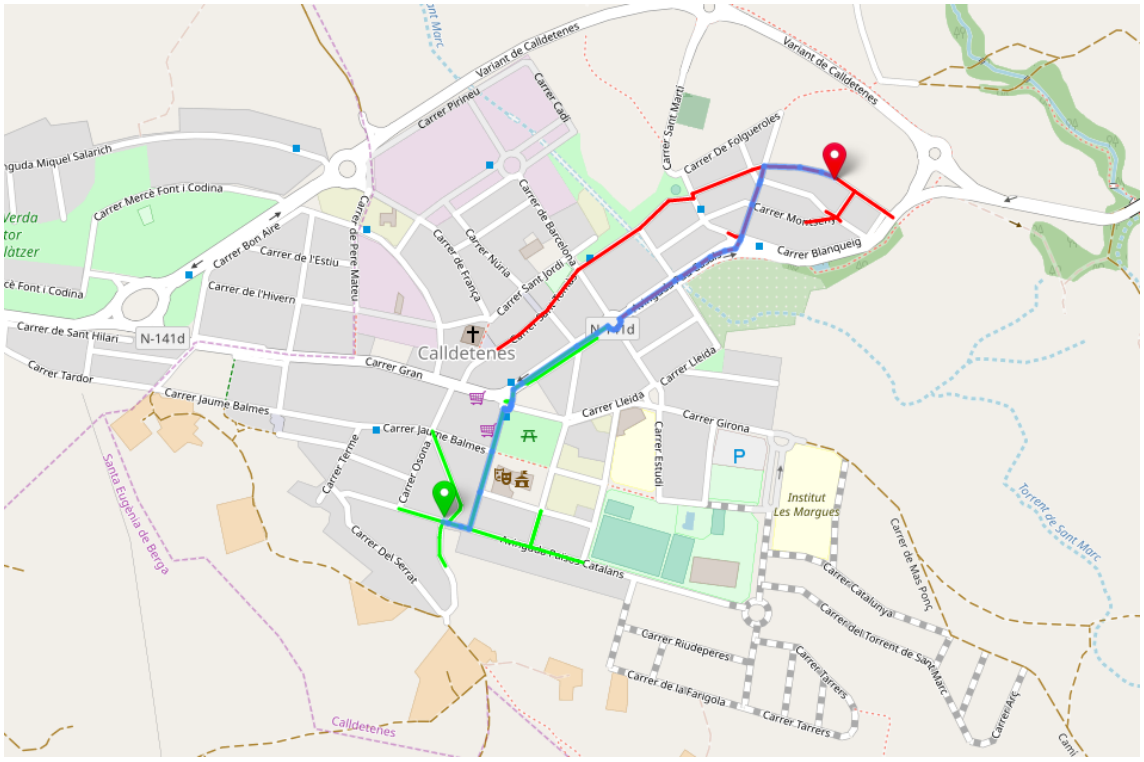

Figura 6.5: Visualització de la cerca amb l'algorisme de cerca A* bidireccional.

En les visualitzacions també podem observar aspectes sobre com funcionen els algorismes que busquen el camí més curt. Per exemple, si ens apropem a les rotondes, es pot apreciar com mai s'acaben de girar del tot, ja que això crearia un cicle, cosa que faria que hi hagués dos camins, a més, segurament un més curt que l'altre. És a dir, veiem que com està explicat en el capítol 4, cada cerca produeix un arbre que es van expandint per tot el mapa, que en realitat és el graf.

També es pot veure com els algorismes respecten les direccions dels carrers, ja que estaven en les dades del mapa i s'utilitza un graf dirigit, fet que fa que les arestes del graf només es puguin recórrer en una direcció. En el cas dels carrers bidireccionals, en realitat hi ha dues arestes antiparal·leles.

Finalment, una altra cosa que podem veure és com les cerques unidireccionals van buscant fins que es troba el vèrtex corresponent al marcador final, mentre que les bidireccionals funcionen fins que les dues cerques es troben. També es pot observar com en algunes cerques bidireccionals, el camí més curt representat per la línia blava passa per algunes arestes que no s'havien marcat en verd o vermell. Això voldrà dir que el camí més curt no passa pel vèrtex en què s'han trobat les dues cerques, ja que com s'ha explicat en la secció sobre cerques bidireccionals del capítol 4, aquest vèrtex no és sempre el que fa que el camí sigui el més curt.

6.3 Comparació amb Google Maps

En darrer lloc, compararem com funciona Camins Calldetenes envers Google Maps tant des de l'exterior, és a dir, el que qualsevol usuari pot veure visitant la pàgina de Google Maps, com des de l'interior, amb l'ajuda d'Albert Graells Rovira, matemàtic que ha treballat a Google Maps i m'ha facilitat informació sobre quins mètodes s'utilitzen per trobar els camins que proporciona.

Aquí tractaré només sobre la cerca de camins. Evidentment Camins Calldetenes no té tantes funcionalitats com un cercador de llocs, diferents modes de transport, vista de carrer (Street View), etc., ja que m'interessava centrar-me en la cerca del camí més curt. Per altra banda, però, Google Maps no té un visualitzador dels algorismes que utilitza.

En primer lloc, s'ha de dir que Google no intenta trobar el camí més curt en distància, sinó que el camí més bo. Els algorismes es basen en el mateix, però el que canvia són els pesos de les arestes del graf que Google Maps també utilitza. En el cas de Camins Calldetenes, el pes de les arestes representa la distància, mentre que en el Google Maps, s'utilitza el temps que es tarda a recórrer-les, tot i que s'afegeix artificialment una mica de cost per girar cap a l'esquerra (perquè cal canviar de carril), per canviar de carrer, etc.

Per aconseguir la velocitat, Google Maps fa servir dades com la velocitat màxima dels carrers però la gran quantitat de dades que obté, que a més encara són més útils, provenen de la gent que tenen sempre la navegació engegada (transportistes, taxistes...). Google Maps a partir de l'aplicació per mòbils, pot saber com es mouen els cotxes i, per tant, obtenir dades sobre el trànsit i la velocitat en què se circula.

Si en un lloc no hi ha dades en directe, s'utilitzen dades mitjanes. També hi ha organismes nacionals que publiquen dades sobre el trànsit i incidències a la carretera.

Tot això ho podem observar lleugerament en la següent figura 6.6 que mostra el camí que troba Google Maps entre els dos punts també utilitzats en les figures anteriors per Camins Calldetenes.

Figura 6.6: Google Maps mostrant el camí que va des de l'avinguda dels Països Catalans, 17 fins al carrer Jacint Verdaguer, 28.

Es pot veure com Google Maps troba un camí diferent del trobat per Camins Calldetenes. Observem que Google Maps prefereix passar més tros per la carretera en comptes de passar per carrers i estalviar-se un gir respecte de la ruta trobada per Camins Calldetenes, ja que en cotxe segurament serà més ràpid. Tot i això, si es busca la ruta per anar a peu amb Google Maps, et dona la mateixa ruta trobada per Camins Calldetenes, ja que anant a peu no es tarda més temps fent girs però si fent més distància.[3]

6.3.1 Algorisme utilitzat per Google Maps

Pel que fa a l'algorisme utilitzat per Google Maps, també difereix de Camins Calldetenes. Camins Calldetenes, tot i que pot utilitzar en el visualitzador l'algorisme de Dijkstra, el de cerca A* i les seves versions bidireccionals respectives, de normal quan no s'especifica, s'utilitza l'algorisme de cerca A* bidireccional, ja que sol ser el més ràpid.

Per la seva part, Google Maps utilitza un algorisme propi, és a dir, personalitzat per les seves necessitats. Tot i això, aquest algorisme està basat en l'algorisme de

[3]He comprovat la ruta sobre el terreny i no és que Google Maps doni una ruta diferent perquè Camins Calldetenes vagi contra direcció algun cop.

jerarquies de contracció, o més conegut pel seu nom anglès, *contraction hierarchies algorithm*.

A més, el codi de l'algorisme que fa servir Google Maps no és públic, o sigui, no és de codi obert. Malgrat tot, l'Albert Graells Rovira m'ha pogut explicar a grans trets el funcionament d'aquest algorisme.

L'algorisme de jerarquies de contracció encara va més de pressa que els algorismes presentats, ja que aprofita la natura jeràrquica dels sistemes de carreteres que existeixen al món. Per exemple, vegem la següent ruta trobada per Google Maps des de l'església de Calldetenes fins a la Torre Eiffel de París.

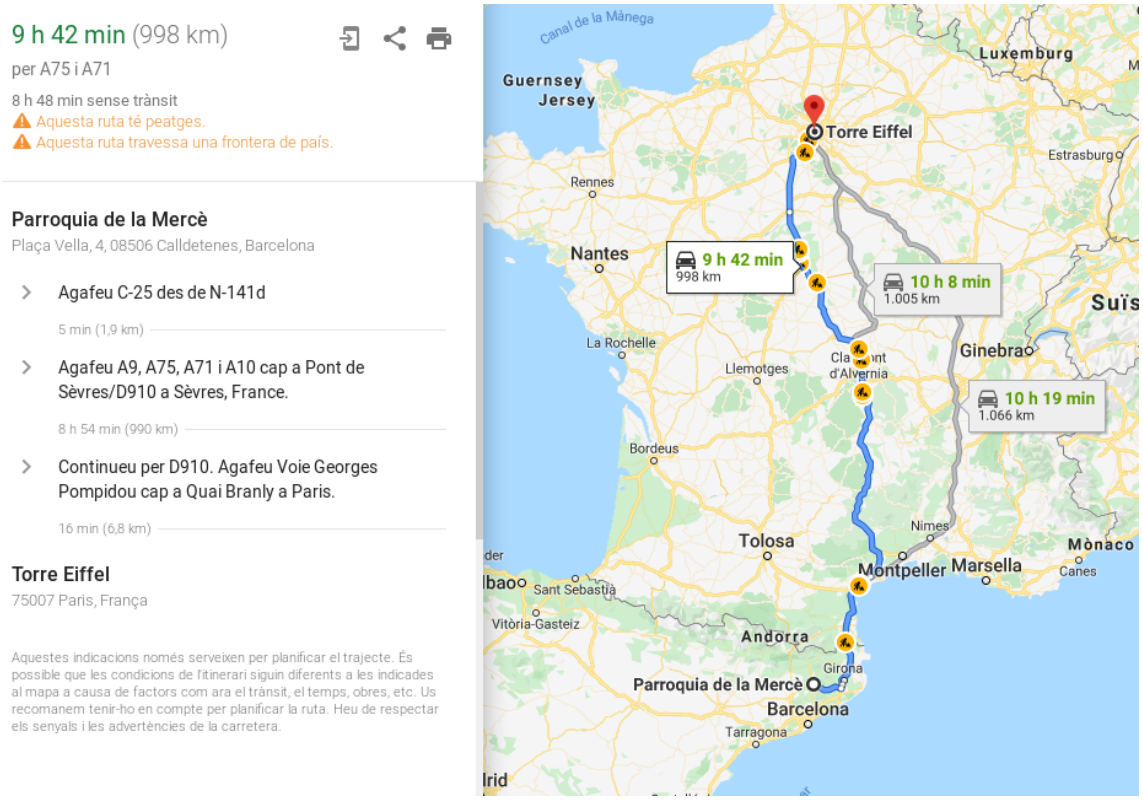

Figura 6.7: La ruta proporcionada per Google Maps per anar des de l'església de Calldetenes fins a la Torre Eiffel de París.

Podem observar que el trajecte comença en un carrer, continua cap a una carretera terciària (l'N-141d), tot seguit s'incorpora a una carretera més gran (la C-25 o Eix Transversal) i després segueix per diverses autopistes (A9, A75, A71...), que són encara de més importància. Al cap de molts quilòmetres per autopista, es torna a una carretera de menor importància (la D910), llavors cap a una encara menor (la Voie Georges-Pompidou) fins que, finalment, de nou es torna als carrers. Així doncs, en un trajecte es comença des dels tipus de vies menys importants, s'ascendeix a la jerarquia de vies fins a arribar per exemple a una autopista, i llavors es torna a descendir de la jerarquia fins a arribar de nou a les vies de menor importància com els carrers.

Per aprofitar aquesta natura jeràrquica de les carreteres, l'algorisme de jerarquies de contracció consta d'una etapa de preprocessament del graf. En aquesta etapa, els vèrtexs s'ordenen per importància. Obtindran major importància els vèrtexs com els que representen la incorporació o la desincorporació a una carretera gran, ja que seran vèrtexs per on molts camins òptims passaran. També hi ha alguns vèrtexs que són inevitables, com per exemple un pont que connecta dues parts de terra que no estan connectades per altres ponts propers. Els vèrtexs de menor importància seran aquells que no canvien entre tipus de vies, per exemple, el vèrtex entre dos carrers. En el graf en si, què representa cada aresta no està determinat (si és un carrer o una autopista), però a partir de l'ús d'heurístiques, es descobrirà quines són les arestes més importants, cosa que permetrà fer l'ordenació dels vèrtexs.

Un cop ordenats els vèrtexs, es genera la jerarquia *contraient* repetidament el vèrtex menys important. Contraure un vèrtex v vol dir que se substitueixen els camins més curts que passen per v per *dreceres*. Per exemple, si hi ha el camí més curt entre u i w és $\{u, v, w\}$, el vèrtex v se substituiria per una aresta de drecera que fes $\{u, w\}$. Per calcular els camins més curts aquí, sí que s'utilitzen algorismes similars a la cerca A* bidireccional. Un cop s'han contret tots els vèrtexs, s'acaba l'etapa de preprocessament i haurem obtingut un graf on serà més ràpid obtenir el camí més curt. Aquest graf tindrà les mateixes arestes que el graf original més les arestes de drecera que haurem afegit. També guardarem l'ordenació dels vèrtexs que hem fet en aquest preprocessament.

En el graf es farà una cerca també bidireccional que aprofitarà l'ordenació dels vèrtexs de tal manera que la cerca que va endavant només utilitzarà arestes que porten a vèrtexs de major importància, mentre que la cerca que va endarrere només utilitzarà arestes que vénen de vèrtexs de major importància. L'algorisme de cerca llavors haurà d'explorar molts menys vèrtexs gràcies a l'ordenació i sobretot també al fet que el graf tindrà moltes dreceres. Totes aquestes optimitzacions faran que la cerca arribi a ser milers de vegades més ràpida que si utilitzéssim l'algorisme de Dijkstra, que si el féssim servir per trajectes tan llargs com el de la figura 6.7 tardaria molt.

Tot i que un cop es té el graf preprocessat les cerques són molt ràpides, el preprocessament sol ser molt costós computacionalment i pot arribar a tardar més d'un dia amb grafs molt grans com un graf amb totes (o gairebé totes) les carreteres del món com el que fa servir Google Maps. A més, constantment hi ha canvis al món; per exemple, carrers nous o talls, cosa que fa que el graf s'actualitzi i s'hagi de reprocessar. En el cas de Google Maps, el graf del món sencer es reprocessa diversos cops per setmana.

Google va modificar l'algorisme de jerarquies de contracció perquè pogués funcionar amb informació a temps real sobre embussos, carrers tallats, obres... També el va adaptar perquè pogués trobar *rutes alternatives*. Les rutes alternatives són rutes que no solen ser les més òptimes però segueixen sent basant bones i molts cops també són d'interès per a l'usuari. La majoria d'algorismes coneguts només computen el camí òptim, i si els modifiques perquè calculin el segon (o tercer o quart) camí òptim, els resultats són pràcticament iguals, cosa que no ens interessa. Calcular

rutes alternatives que siguin suficientment diferents és molt complex i Google va adaptar l'algorisme perquè fos possible. En la figura 6.7 es pot veure com Google Maps suggereix dues rutes alternatives.

Una altra diferència entre Google Maps i Camins Calldetenes és que Google no té tot el graf en memòria mentre es busca el camí, ja que, per l'enormitat del graf del món sencer seria infactible. Per això, el graf s'ha de partir en trossos utilitzant dues tècniques: la partició geogràfica i la partició jeràrquica. Per una banda, la partició geogràfica consisteix a carregar les parts del graf per on ha de passar la ruta de forma aproximada, per exemple per estats. Per altra banda, la partició jeràrquica parteix el graf per nivells d'importància dels vèrtexs.

El nivell més baix d'importància, on hi ha la gran majoria de vèrtexs, no cap sencer en la memòria; per tant, es parteix geogràficament. Llavors, es comença la cerca en el graf del nivell més baix de la zona on comença la ruta (o on acaba en el cas de la cerca que va endarrere) i si es detecta que es puja a una importància major es carrega el graf amb els vèrtexs d'aquesta major importància. Els grafs amb vèrtexs més importants tenen un ordre menor, cosa que fa que no s'hagin de partir geogràficament.

En conclusió, Google Maps cerca rutes en grafs d'abast mundial ponderats amb pesos més complexos que la distància i que també estan influenciats per dades dinàmiques a temps real. Una altra funcionalitat és la de buscar rutes alternatives. Tot això va produir la necessitat de desenvolupar un algorisme personalitzat eficient, el qual però, també és complex i necessita un preprocessament molt costós computacionalment. L'algorisme utilitzat per Google Maps es pot considerar com l'algorisme més complet per trobar rutes òptimes en un mapa, ja que a més és utilitzat per milions d'usuaris, la qual cosa li permet obtenir una millor constant.

"Tot camí té un final."

— Sèneca, filòsof romà, segle I.

7 Conclusions

En aquest treball hem començat el camí passant pels ponts de Königsberg i veient com un problema aparentment banal va ser la primera pedra dels fonaments de la teoria de grafs. Hem observat com aquesta branca de les matemàtiques, amb aquest origen peculiar i sense gaire utilitat, ha acabat sent aplicada a multitud de qüestions quotidianes, com utilitzar una xarxa social o fer una cerca web, o també per a tasques avançades, com dissenyar un microxip o estudiar el cervell. Però una de les aplicacions més interessants de la teoria de grafs i la que ens ha concernit, és la de trobar el camí més curt per anar d'un lloc a un altre.

Hem vist com podem abstreure les xarxes de carreteres de la realitat i convertir-les en grafs. A partir d'algorismes, llavors podem trobar la ruta més curta entre dos punts del mapa de carreteres o, en el graf, trobar el camí de menys pes entre dos vèrtexs.

Hem examinat l'algorisme de cerca en amplada, que soluciona el problema de trobar el camí més curt però en un graf no ponderat; per tant, encara no soluciona el problema de buscar el camí entre dos punts d'un mapa, ja que en aquest cas el graf és ponderat. Per això, hem vist el funcionament de l'algorisme de Dijkstra, que sí que soluciona el problema.

No obstant això, també hem examinat diverses tècniques per obtenir el camí més curt de manera més eficient. En primer lloc, hem vist que no calia processar tot el graf i que podíem aturar l'algorisme un cop ja haguéssim arribat al vèrtex final. A partir d'aquesta optimització, també hem examinat l'algorisme de cerca A*, que només amb una petita modificació a l'algorisme de Dijkstra, aconsegueix grans millores de rendiment. L'A* també ens ha permès aprendre sobre les funcions heurístiques i quan s'utilitzen. Una altra tècnica treballada és la cerca bidireccional, que l'hem combinat amb l'algorisme de Dijkstra i amb l'algorisme d'A*, obtenint així algorismes encara més eficients.

Aquests algorismes, basats també en la teoria de grafs, m'han permès assolir el meu objectiu final de desenvolupar un planificador de rutes per Calldetenes. A més, també he aconseguit crear un visualitzador que permet veure com funcionen els algorismes examinats anteriorment sobre el mapa de Calldetenes mentre busquen el camí més curt. El visualitzador d'algorismes també ens ha permès confirmar que les millores descrites realment eren millores.

Tanmateix, el camí no ha acabat aquí, ja que també he tingut la sort de poder parlar

amb un treballador de Google que m'ha proporcionat detalls sobre els algorismes que són utilitzats per un dels planificadors de rutes més avançats, Google Maps. Hem vist així que encara més tècniques són necessàries per optimitzar la cerca de camins òptims, a causa de l'enormitat del graf que conformen totes les carreteres de la Terra i que moltes dades són dinàmiques.

Finalment, en l'àmbit personal, penso que la realització d'aquest treball de recerca m'ha resultat molt interessant i positiva. A part d'aconseguir l'objectiu del treball, també he après molt sobre grafs i algorismes; no només els presentats en el treball, sinó també d'altres, i sobre com es dissenyen. També he après tot el procés de creació d'una aplicació web, com fer servir una plataforma com Microsoft Azure i com finalment publicar-ho a internet. Així mateix, aquest treball m'ha exigit fer molta recerca i fer ús intensiu de la bibliografia, la qual en major part és en anglès, i considero que durant el transcurs he anat aprenent on i com buscar. De cara al futur, voldria aconseguir implementar encara més algorismes i executar-los amb el visualitzador d'algorismes desenvolupat. I en general, m'agradaria seguir fent recerca i estudiant sobre els algorismes i les matemàtiques.

Fonts d'informació

Bibliografia

- BIGGS, Norman L.; LLOYD, E. Keith; WILSON, Robin J. *Graph Theory, 1736-1936*. Oxford University Press, 1998.
- HOPKINS, Brian; WILSON, Robin J. "The Truth about Königsberg". *The College Mathematics Journal*, vol. 35 (2004), pàg. 198-207.
- BÓNA, Miklós. *A Walk Through Combinatorics: An Introduction to Enumeration and Graph Theory*. World Scientific Publishing Co. Pte. Ltd., 2017.
- DIESTEL, Reinhard. *Graph Theory*. Springer, 2017.
- EVEN, Shimon. *Graph Algorithms*. Cambridge University Press, 2011.
- BRIN, Sergey; PAGE, Lawrence. "The Anatomy of a Large-Scale Hypertextual Web Search Engine" (1998).
- SZALKAI, Balázs; VARGA, Bálint; GROLMUSZ, Vince. "The graph of our mind" (2016).
- RIVEST, Ronald L.; CORMEN, Thomas H.; LEISERSON, Charles E.; STEIN, Clifford. *Introduction to Algorithms*. MIT Press, 2009.
- AHUJA, Ravindra K.; MAGNATI, Thomas L.; ORLIN, James B. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc., 1993.
- RUSSELL, Stuart; NORVIG, Peter. *Artificial Intelligence: A Modern Approach*. Pearson, 2010.
- FRANA, Philip L.; MISA, Thomas J. "An interview with Edsger W. Dijkstra". *Communications of the ACM*, vol. 53 (2010), pàg 41-47.
- NILSSON, Nils J. *The Quest for Artificial Intelligence*. Cambridge University Press, 2009.
- IKEDA, Takahro; HSU, Min-Yao; IMAI, Hiroshi; NISHIMURA, Shigeki; SHIMOURI, Hiroshi; HASHIMOTO, Takeo; TENMOKU, Kenji; MITOH, Kunihiko. "A fast algorithm for finding better routes by AI search techniques". Proceedings of VNIS'94 - 1994 Vehicle Navigation and Information Systems Conference, Yokohama, Japan (1994), pàg. 291-296.
- GEISBERGER, Robert; SANDERS, Peter; SCHULTES, Dominik; DELLING, Daniel. "Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks". Proceedings of the 7th International Conference on Experimental Algorithms (2008), pàg 319-333.

Recursos electrònics

- Google. “How Google Search works”.
  URL: google.com/search/howsearchworks.
- CCMA. “Nou avenç en la detecció precoç de l’Alzheimer gràcies a La Marató”. 324, 2018.
  URL: ccma.cat/324/nou-aven-en-la-detecci-preco-de-lalzheimer-grcies-a-la-marat/noticia/2849363/.
- GOLDBERG, Andrew V.; HARRELSON, Chris; KAPLAN, Haim; WERNECK, Renato F. *Efficient Point-to-Point ShortestPath Algorithms*. Princeton University, 2006.
  URL: cs.princeton.edu/courses/archive/spr06/cos423/Handouts/EPP shortest path algorithms.pdf
- OpenStreetMap. “OpenStreetMap Wiki.”
  URL: wiki.openstreetmap.org.
- Python Software Foundation. “Python 3 Documentation”.
  URL: docs.python.org/3.
- Pallets. “Flask Documentation”.
  URL: flask.palletsprojects.com.
- Microsoft. “Microsoft Azure Documentation”.
  URLs: azure.com i docs.microsoft.com/azure/app-service.
- WEISSTEIN, Eric W. “MathWorld”. Wolfram Research, Inc.
  URL: mathworld.wolfram.com.
- CHAMBERLAIN, Robert G. “What is the best way to calculate the distance between 2 points?”. *United States Census Bureau Geographic Information Systems FAQ*, 1996.
  URL: movable-type.co.uk/scripts/gis-faq-5.1.html.
- FREIBERGER, Marianne. “Lost but lovely: The haversine”. *Plus Magazine*, 2014. URL: plus.maths.org/content/lost-lovely-haversine.

Annexos

A

Notes sobre el pseudocodi

El pseudocodi és una representació simplificada del codi real d'un algorisme. Utilitza les convencions estructurals d'un llenguatge de programació normal, però està destinat a la lectura humana en lloc de la lectura automàtica feta per una màquina.

En aquest annex s'aclareixen els significats d'alguns símbols i estructures utilitzades en el pseudocodi usat al llarg del treball:

- Els espais a principi de línia (també anomenat la *indentació*) serveixen per indicar els blocs de codi. Si diverses línies tenen el mateix nombre d'espais, formaran part del mateix bloc. El funcionament seria el mateix que en Python.
- "`=`" serveix per assignació. Per exemple, quan s'executa `x = 42`, la variable x tindria el valor 42.
- "$=$" serveix per comparar dues variables per igualtat. Per exemple, si les dues variables x i y tenen el valor 42, el bloc a continuació de "`si x = y:`" s'executaria. El funcionament és anàleg per "$\neq$", "$\leqslant$"...
- "`.`" s'utilitza per accedir a les propietats d'un objecte. Per exemple, "`G.|V|`" vol dir accedeix a l'ordre de G, seria equivalent a dir "$|V|$ de G".
- Les claus ("`{`" i "`}`") s'utilitzen per marcar les llistes. Com ara "`{0, 4, 2}`" és una llista amb el nombre 0 en primera posició, el 4 en segona posició i el 2 en tercera.
- L'estructura "`l = {x} * y`" serveix per assignar a la variable l una llista de y elements x. Com a mostra, executar "`l = {0} * 5`" seria el mateix que fer "`l = {0, 0, 0, 0, 0}`". Evidentment, com es fa durant el treball podem fer servir variables en comptes del nombre literal. També es pot fer "`{{}} * x`" per crear llista que dins contingui x llistes buides.

B

Eines utilitzades per la realització del treball

Per realitzar aquest treball he hagut de fer un ús intensiu de diferents eines informàtiques, les quals també d'alguna manera o altra, l'han fet possible i n'han facilitat la confecció de la part escrita i la del desenvolupament del planificador de rutes.

B.1 Sistema operatiu

Per escriure aquest treball i desenvolupar el planificador de rutes he utilitzat el sistema operatiu Ubuntu, una distribució de GNU/Linux. Treballar en sistemes GNU/Linux molts cops facilita el desenvolupament de programari, ja que té moltes eines útils per la programació. A més, Ubuntu i la majoria d'eines que proporciona, són de codi obert i gratuïtes.

El sistema operatiu utilitzat en el servidor de Microsoft Azure que allotja la pàgina web de `caminscalldetenes.cat` també està basat en Linux.

B.2 Confecció del document escrit

B.2.1 LaTeX i TeXstudio

Per redactar la part escrita d'aquest treball (això), he utilitzat el sistema de tipografia LaTeX. LaTeX es diferencia d'altres programes com Microsoft Word o LibreOffice Writer en el fet que només s'escriu text amb comandes per donar format, crear fórmules matemàtiques, inserir figures..., en comptes de posar-ho tot barrejat en un mateix document.

Utilitzar LaTeX permet un major control de com queda el document, facilita la inserció de fórmules matemàtiques i figures, i la generació d'elements complexos com gràfics, grafs o codi.

LaTeX es pot estendre a partir de l'ús de paquets, que proporcionen funcionalitats extres. Per exemple, hi ha un paquet anomenat *dot2texi*, que permet generar els diagrames dels grafs fàcilment, un altre que es diu *listings* que permet escriure codi que automàticament es pinta de colors per millorar la comprensió... En total he utilitzat 25 paquets de LaTeX: *lipsum*, *geometry*, *babel*, *xcolor*, *url*, *hyperref*, *tocbibind*,

quotchap, *parskip*, *multicol*, *amsthm*, *amsfonts*, *amssymb*, *mathtools*, *caption*, *wrapfig*, *float*, *graphicx*, *dot2texi*, *tikz*, *fontspec*, *listings*, *lstfiracode*, *changepage* i *appendix*. Alguns paquets al seu torn utilitzen altres paquets no mencionats.

Un cop escrit el LaTeX, es compila i es genera el PDF. Per fer-ho, es pot fer directament o utilitzant un programa com TeXstudio, que és el que he utilitzat, que et permet veure com va quedant el document mentre escrius el LaTeX. TeXstudio també intenta corregir errors ortogràfics i gramaticals a partir de l'ús del corrector ortogràfic LanguageTool.

Vaig decidir aprendre a fer servir LaTeX i utilitzar-lo per redactar el treball de recerca per diversos motius. Primerament, perquè considero que ofereix una qualitat tipogràfica superior a la d'altres programes tradicionals. En segon lloc, perquè facilita la inserció d'elements que necessitava per explicar millor el treball, com fórmules matemàtiques i diagrames de grafs. Un altre motiu és que LaTeX és l'estàndard de facto per la comunicació i publicació de documents científics, és a dir, que en el futur segurament l'hauré de fer servir més. Finalment, també perquè tant LaTeX de la mateixa manera que tots els paquets que he utilitzat i TeXstudio, són programari lliure, de codi obert i gratuït.

B.2.2 Graphviz, Inkscape i GIMP

He utilitzat diverses eines per realitzar les diferents figures que apareixen al llarg del treball, que, menys algunes fotos, són totes d'elaboració pròpia.

Per crear els diagrames de grafs, he utilitzat el paquet de LaTeX *dot2texi* que permet inserir grafs molt fàcilment en el document. Per exemple, per crear el graf de la figura 2.16 (pàgina 21), només s'ha d'escriure el següent codi en el document de LaTeX:

```
\begin{dot2tex}[neato,mathmode]
  digraph E {
    nodesep=0.5;
    node [shape=circle, style=filled, fillcolor=WhiteSmoke];

    v_1 -> v_3 [label="e_5"];
    v_3 -> v_2 [label="e_6"];
    v_1 -> v_2 [label="e_1"];
    v_1 -> v_3 [label="e_4"];
    v_2 -> v_1 [label="e_2"];
    v_2 -> v_2 [label="e_3"];
  }
\end{dot2tex}
```

El paquet *dot2texi* al seu torn utilitza un programa extern anomenat Graphviz, que serveix per generar diagrames de grafs, una tasca que seria molt més complicada si s'hagués de fer amb un editor d'imatges tradicional.

Per crear la resta de diagrames, he fet servir Inkscape, un programa d'edició de gràfics vectorials. Finalment, per editar algunes imatges he utilitzat GIMP.

Tots aquests programes també són programari lliure, de codi obert i gratuït.

B.3 Desenvolupament del planificador de rutes

B.3.1 PyCharm

Per escriure el codi font del planificador de rutes vaig utilitzar principalment l'entorn de desenvolupament integrat PyCharm, que proporciona moltes funcions. Algunes de les més importants i útils són el ressaltat en color de sintaxi, la inspecció del codi per detectar errors de programació, la formatació automàtica, la depuració del programa per veure com funciona internament o el suport pel desenvolupament d'aplicacions web amb Flask. Tot i que està principalment enfocat al desenvolupament de Python, PyCharm també permet escriure codi en JavaScript, HTML i CSS, com el que he utilitzat per a la part web del planificador.

PyCharm és un programa de pagament desenvolupat per l'empresa JetBrains, però que és gratuït per estudiants i pel desenvolupament de projectes de codi obert.

B.3.2 Firefox

També he fet servir el navegador web Mozilla Firefox per provar el funcionament del planificador de rutes i editar alguns paràmetres. Firefox té eines de desenvolupament integrades en el navegador que permeten depurar el codi de JavaScript i modificar paràmetres d'estil escrits en CSS. Aquest tipus d'eines també es poden trobar en altres tipus de navegadors.

Firefox, a part, també és un dels pocs navegadors web que és totalment de codi obert i lliure.

B.3.3 Git i GitHub

Git és un sistema de control de versions que guarda les diverses versions dels fitxers que conformen un projecte al llarg del seu desenvolupament. És una eina útil per si es necessita veure els canvis que s'han anat fent i per si se n'ha de revertir algun. Per projectes desenvolupats per diverses persones, també és útil, ja que pot ajuntar diversos canvis en un mateix repositori central. És una eina utilitzada en molts projectes de desenvolupament de programari. En el meu cas, l'he utilitzat durant el desenvolupament del planificador de rutes Camins Calldetenes.

Pel que fa a GitHub, és un servei ofert per Microsoft que proporciona allotjament gratuït a repositoris de codi que utilitzen Git. És utilitzat per molts projectes de codi obert com Camins Calldetenes per no només guardar el codi, sinó també per mostrar-lo al públic. El repositori de codi a GitHub de Camins Calldetenes és accessible a través de l'enllaç `github.com/salcc/CaminsCalldetenes`.