PROTOTIPAT D'UNA GRAPH NEURAL NETWORK

TREBALL DE FINAL DE GRAU (3 DE JULIOL DE 2019)

AUTOR: ALBERT CANYELLES RUIZ

DIRCETOR: ALBERT CABELLOS APARICIO

DEPARTAMENT: ARQUITECTURA DE COMPUTADORS

ESPECIALITAT: COMPUTACIÓ

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Índex

Índex de figures

Índex de taules

1. Abast del projecte i contextualització

Abans de definir la finalitat d'aquest projecte creiem necessari una breu explicació sobre en quin context es troba situat, i aquelles entitats que es veuen afectades pel mateix.

1.1. Contextualització del projecte

1.1.1. Context actual

Suposem que li venen donades el mapa d'una ciutat i les dades de tot allò que influeix en el tràfic d'aquesta (com ara el nombre de vehicles que s'hi mouen, la seva posició original, el seu destí, la configuració dels semàfors, els vianants que hi caminen, etc.), i li demanen que calculi quan tardaran en arribar un conjunt de vehicles a la seva destinació i per on ho faran. És un problema especialment complex, ja que els vehicles no es desplacen amb un patró fàcilment predictible ja que, per exemple, un conductor pot canviar la seva ruta cap a un camí més llarg per tal d'evitar una retenció en comptes de seguir amb el camí més curt en el plànol.

En un problema semblant es troba en gestió de xarxes, en les quals la topologia[1] de la xarxa i les dades que hi viatgen ja ens venen donades, i s'han d'escollir les polítiques d'encaminament més eficients per a la xarxa.

De la mateixa forma que passava en l'exemple de la ciutat, no es coneix encara cap forma d'eficient per a computar el problema en un temps raonable, però existeixen múltiples aproximacions que permeten obtenir resultats en temps molt

[1] Topologia: configuració dels nodes, usualment routers i switches, i la seves respectives connexions en una xarxa, usualment representada en forma de graf.

acceptable a canvi, òbviament, de relaxar les condicions del problema. Exemples d'aquestes aproximacions són OpenFlow, DEFO i RouteNet. En aquesta última solució serà allà on ens centrarem en aquest projecte.

La proposta de solució de RouteNet, feta per un grup de recerca d'Arquitectura de Computadors de la FIB, consisteix en utilitzar una *Graph Neural Network* (GNN) [1], [2], capaç d'associar la latència d'una xarxa amb la seva configuració.

No obstant l'algorisme que complementa el model de RouteNet encara és molt rudimentari, ja que es basa en generar solucions aleatòries, i quedar-se amb la millor d'elles.

1.1.2.Actors implicats

A continuació detallarem els actors implicats en el projecte:

- **El tutor del projecte:** el tutor d'aquest projecte és l'Albert Cabellos. El seu paper ha estat el de supervisar el correcte compliment del objectius marcats i guiar desenvolupament del projecte a través dels seus coneixements sobre la matèria en qüestió.
- **El grup de recerca de RouteNet:** apart de implementar la GNN de RouteNet, han proporcionat suport i coneixements per tal de facilitar l'acoblament amb el projecte.
- **Els Internet Service Providers (ISP):** els ISP estan interessats especialment la finalització del projecte RouteNet en general, ja que els proporciona escalabilitat, flexibilitat i robustesa contra fallades de la xarxa. I pel que respecte el nostre projecte, els proporcionaria menors temps de computació, i per tant, millors temps de resposta a la xarxa i reduccions de costos.

- **Els Usuaris de ISP:** és el conjunt d'actors que presentarem més gran. Aquests es veurien beneficiats amb latències més baixes i connexions més estables.

1.2. Estat de l'art

Tradicionalment, les grans xarxes com les dels ISP fan servir polítiques d'encaminament poc flexibles, incapaces d'adaptar-se a fallades i sobrecàrregues de la xarxa ràpidament. Això és degut a que aquestes polítiques estan programades a nivell de node i es troben, per tant, codificades en els diferents *routers* i *switches* que formen la xarxa [3]. Aquests tenen una potència computacional molt limitada, i usualment utilitzen algorismes d'una de les següents classes: *distance vector, link state* i *path vector* [4].

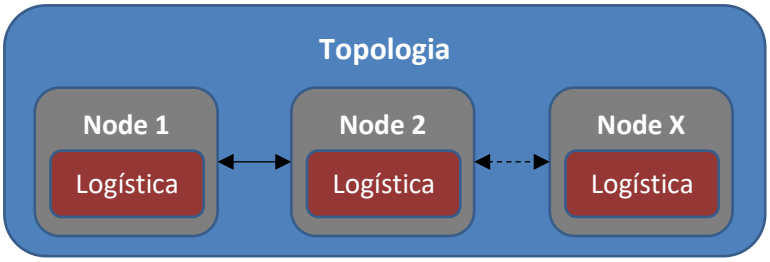

Fig. 1.1 Esquematització d'una xarxa tradicional d'un ISP.

La classe d'algorismes *distance vector* es basen en l'intercanvi d'informació dels nodes entre veïns directes. La idea és que perquè un node mantingui actualitzada la taula de distàncies a la resta de nodes d'una xarxa, aquest ha de consultar les taules dels seus veïns i comprovar els canvis produïts, si en fos el cas. Conseqüentment, aquests algorismes tenen l'inconvenient necessitar enviar cada vegada més informació entre dos nodes a mesura que augmenta el nombre de nodes que conformen de la xarxa. Un exemple d'aquesta classe d'algorismes és el protocol *Routing Information Protocol* (RIP).

Per altre banda tenim la classe *link state*. En aquesta, quan un node s'inicialitza, envia la seva informació a propagar per tota la resta nodes de la xarxa . Els nodes llavors, a mesura que van rebent les dades del node inicial, actualitzen la seva taula de distàncies i la comuniquen al seus veïns. Un representant d'aquesta classe és l'algorisme *Open Shortest Path First* (OSPF).

Finalment tenim els *path vector*. Aquests es basen en anar controlant que els camins pels quals viatge la informació no contén cicles. Per a fer-ho, cada node al qual arriba la informació afegeix la seva identificació en un vector, el qual s'emmagatzema el camí recorregut fins al moment, abans de continuar retransmetent. En el cas en que un node rebi un paquet d'informació que ja contingui la seva identificació no la continuarà retransmetent. Un exemple d'aquesta classe és el *Border Gateway Protocol* (BGP).

Degut a les restriccions que tenen aquest algorismes a l'hora controlar a quins nodes s'encamina la informació en casos de fallada, sobrecàrrega de la xarxa o per altres interessos, es va iniciar la recerca de nous models de xarxes.

Les Software-Defined Networks (SDN) separen la topologia de la xarxa de la seva logística. Aquesta separació atorga a la logística la capacitat de posseir una visió global de l'estat de la xarxa, permetent una major capacitat d'expressivitat a l'hora de posar requisits i restriccions a les diferents demandes[2] de la xarxa. Tal i com està il·lustrat en la figura Fig. 1.2, les SDN queden dividides en tres capes: la topologia, la logística i la d'aplicació. En la capa de logística es troba el controlador de la SDN, el qual fa d'intermediari entre la topologia i les aplicacions. Així doncs, quan un node necessita realitzar una nova connexió, transmet la seva

[2] Demanda: informació que ha de circular des d'un node d'origen a un de destí d'una xarxa.

sol·licitud al controlador, el qual la passa a les aplicacions que cregui necessàries, i li retorna el camí en la topologia que haurà de recórrer la sol·licitud. Cal remarcar que no es necessari realitzar aquest procediment cada vegada que un node necessiti realitzar la mateixa sol·licitud, ja que es poden emmagatzemar en diferents nivells de memòries cau sempre i quan es refresquin periòdicament.

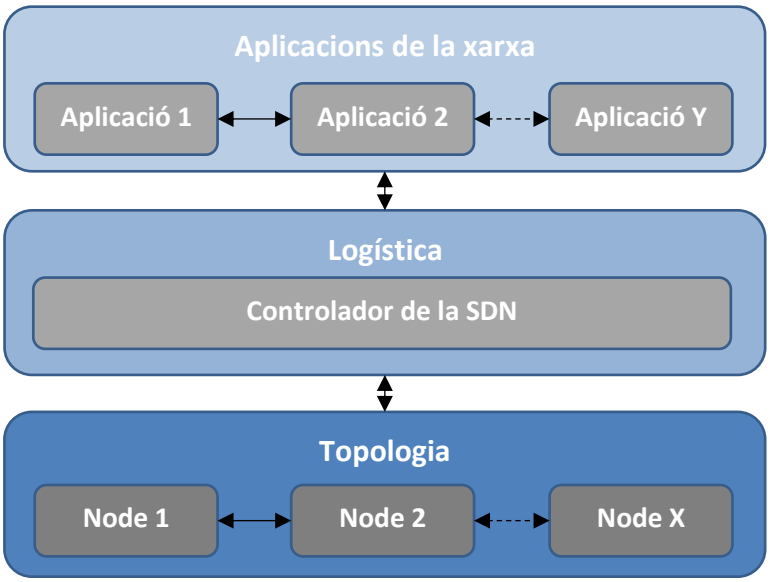

Fig. 1.2 Esquematització d'un SDN.

El problema que de les SDN és que a dia d'avui és que encara no és prou potent com per a ser usat en les xarxes tant grans com les dels ISP, no obstant sí que està sent utilitzada actualment en el món empresarial [5]. Exemples d'aquesta tecnologia en són OpenFlow, ONOS i CORD [6].

Una altre aproximació al problema plantejat és la *Declarative and Expressive Forwarding Optimizer* (DEFO) [7]. DEFO està basat en la tècnica de *Segment Routing*, i té com a objectiu minimitzar la saturació[3] dels enllaços d'una xarxa. A diferència de les tècniques vistes fins ara, on el camí que realitzarà un paquet es genera a mesura que va navegat pels nodes de la xarxa o dins del controlador

[3] Saturació d'un enllaç: és la reducció de la qualitat del servei en la que es troba un enllaç quan a través d'ell s'intenta fer circular més informació de la que és capaç de suportar[16].

de la SDN, en *Segment Routing* el camí es calcula en el node inicial. En aquest s'emmagatzema una llista amb cada node pel qual el paquet ha de passar i, a mesura que ho va fent, cada node s'elimina a si mateix d'ella fins a arribar al destí. La forma en que DEFO afronta el problema de Segment Routing és una combinació de les tècniques de *Constraint Programming* i *Large Neighborhood Search* (LNS) per a generar un camí per a totes les demandes d'una xarxa [8].

Constraint Programming consisteix en expressar les relacions i restriccions entre les dades d'un problema com a una sèrie d'expressions lògiques, per tal reduir el problema original al problema de satisfacció booleana (SAT) [9].

LNS és una metaheurística[4] que, donada una solució inicial a un problema és millorada progressivament a través desfer-la i refer-la parcialment múltiples vegades [10].

Així doncs, amb la combinació de les dues DEFO és capaç d'expressar *Segment Routing* com a una seqüència de restriccions lògiques, enviar-les a un solucionador de SAT, fixar part de les variables de la solució obtinguda per SAT i recomputar-la fins a tenir una solució satisfactòria. La seva efectivitat el fa capaç de poder ser usat en xarxes tant grans com les d'un ISP.

Finalment, una altra aproximació totalment diferent és RouteNet [11], la qual està basada en l'ús d'aprenentatge automàtic per tal de preveure la latència de les demandes d'una xarxa. Aquesta solució està sent desenvolupada per un grup de recerca de la FIB, i està basada en una tecnologia en recent expansió, les *Graph Neural Networks* (GNN), capaces d'aprendre a través d'informació estructurada

[4] Una metaheurística és un mètode heurístic per a resoldre un tipus de problema computacional general, utilitzant els paràmetres donats per l'usuari sobre uns procediments genèrics i abstractes d'una manera que s'espera eficient [17].

en forma de graf. La GNN ha estat implementada amb l'API de TensorFlow en Python3, i entrenada amb resultats precisos obtinguts mitjançant simulacions. Amb aquesta tecnologia, a partir de informació sobre la topologia, les demandes dins la xarxa i la matriu d'encaminament[5] de la xarxa es pot obtenir una predicció sobre quina latència tindrà cada demanda.

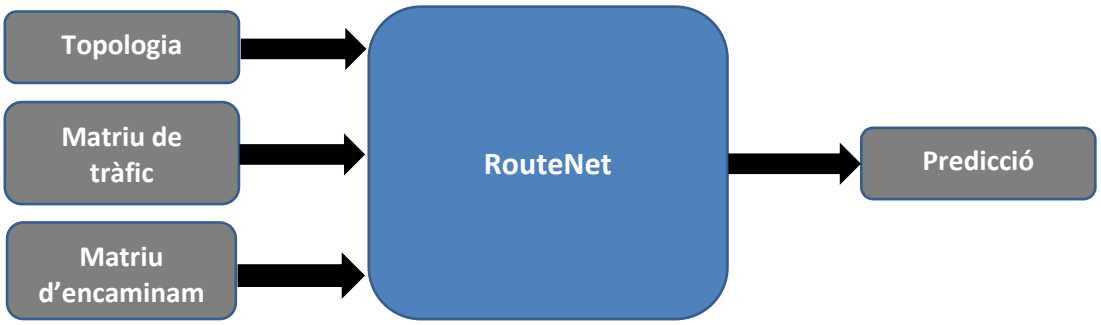

Fig. 1.3 Dades d'entrada i de sortida del model de RouteNet.

No obstant, perquè aquesta solució sigui funcional dins d'un controlador de SDN cal anar un pas més enllà, i no només avaluar a partir de les tres dades d'entrada, sinó arribar a ser capaç de generar matrius d'encaminament pròpies per tal de poder buscar la configuració dels ports amb la latència més baixa. Serà en aquest punt on es centrarem durant la resta del projecte.

1.3. Formulació del problema

En el següent apartat ens disposarem a especificar els objectius que es pretenen assolir a través d'aquest projecte, juntament amb l'abast del mateix.

1.3.1. Objectius

L'objectiu d'aquest projecte del consisteix en, com ja havíem deixat entreveure al final de l'estat de l'art, prototipar dos generadors de matrius de tràfic que

[5] Matriu d'encaminament: matriu que conté per a cada parella de nodes *N*,*M* d'una xarxa, per quin port de *N* ha s'ha de transmetre la informació per tal que arribi a *M*.

donades una topologia, un tràfic en la xarxa i unes restriccions de latència màximes per a cada demanda de la xarxa, minimitzin la latència de la xarxa satisfent les restriccions imposades. Un cop implementats, s'avaluarà l'eficàcia entre els ells.

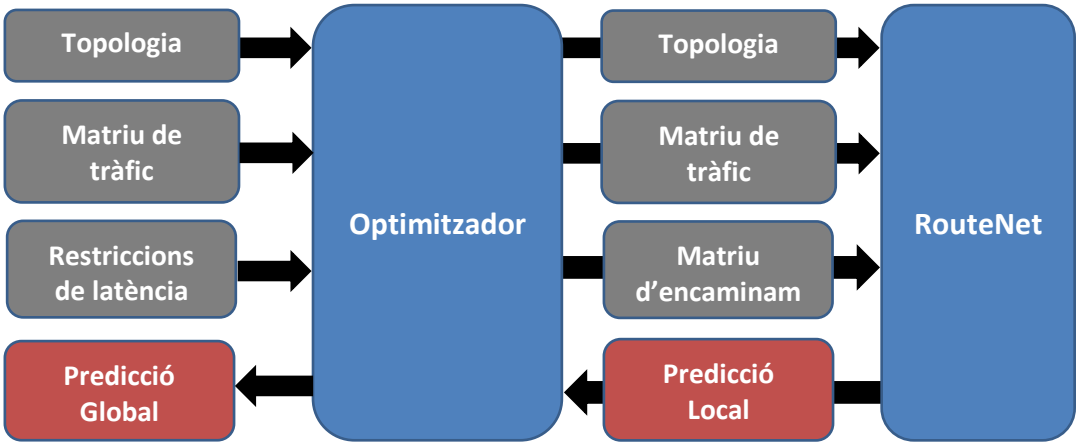

Fig. 1.4 Esquema del funcionament esperat dels optimitzadors.

1.3.2. Abast

L'abast d'aquest projecte el podem definir els següents apartats:

- **La implementació del l'optimitzador**: aquest serà programat en Python3, i es plantejaran tres possibles implementacions amb una lectura de dades comuna:
  - *L'adaptació de l'optimitzador implementat per DEFO*: consistirà en adaptar l'algorisme emprat per DEFO, el qual optimitzava càrregues del enllaços, per tal de que funcioni amb un model entrenat per a RouteNet.
  - *Un optimitzador basat en una cerca per profunditat*: un cop implementat l'optimitzador l'anterior, ens disposarem a implementar un nou codi per tal que heurístic del optimitzador passi a fer servir RouteNet en comptes de capacitats.
  - *Un optimitzador basat en una cerca per amplada*.

- **L’estudi dels resultats**: en aquest apartat final ens disposarem a analitzar els resultats obtinguts pels nostres optimitzadors respecte si mateixos. Es realitzarà a partir d’un anàlisis de les latències retornades i el temps d’execució del mateixos.

1.3.3. Possibles obstacles

És ben sabut que el temps atenua les complicacions que poden sorgir en un projecte. En el nostre cas, el temps no és un recurs abundant. La quantitat de feina a realitzar durant aquests quatre mesos és notable, fet que accentua la gravetat dels possibles imprevistos que puguin aparèixer.

Durant la realització d’aquest projecte els obstacles als que ens trobarem amb major probabilitat seran els relacionats amb el codi, i especialment aquells que estiguin relacionat amb *Constraint Programming*. Aquest és l’encarregat de posar restriccions al model resultat, i en cas de patir retards severs en el desenvolupament del projecte es poder relaxar el nombre de restriccions a implementar al mínim perquè l’algorisme sigui viable.

Fins i tot, en el pitjor dels casos, podria ser necessari l’abandonament de la implementació d’un tercer optimitzador, ja sigui per falta de temps o deguda la naturalesa del projecte, que sent de recerca, es podria descobrir en mig del procés que l’enfocament plantejat no podria ser beneficiós pel nostre objectiu.

1.4. Metodologia

Tenint en compte els límits de durada del projecte, es requerirà una metodologia de treball estricte. Aquesta consistirà una metodologia d’enfocament de projectes clàssica, basada així en un estudi previ rigorós de la matèria en qüestió

pel disseny i desenvolupament final del projecte. Es plantejaran també fites setmanals, revisades amb el director del projecte, per tal d'assegurar-ne un progrés constant.

2. Descripció de les tasques

Com ja havíem comentat ja en la metodologia, farem servir una gestió tradicional del projecte. Per aquest motiu el diagrama de Gantt del projecte es pot dividir principalment en: la recerca de documentació i informació que permeti plantejar realísticament els objectius i possibles obstacles del problema; el disseny i implementació del codi; la supervisió dels resultats obtinguts; i finalment el tancament del projecte. El diagrama de Gantt inicial per aquest projecte és el següent:

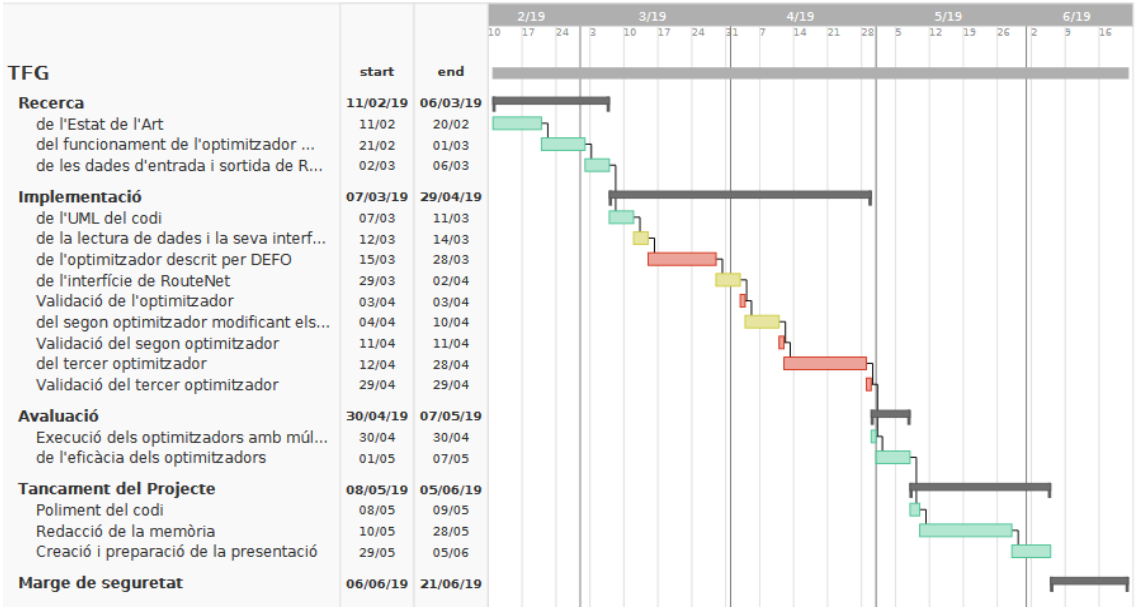

Fig. 2.1 Diagrama de Gantt inicial.

2.1. Recerca

En aquesta secció s'hi comprèn la major part de la investigació i estudi de la teoria del projecte (23 dies en total). Aquesta fase té un nivell de risc molt baix, degut al seu elevat nivell conceptual.

- *Recerca de l'Estat de l'Art (10 dies):* aquest és el moment del projecte on es buscarà quines tècniques d'optimització s'utilitzen en els algorismes d'encaminament en les grans xarxes.
- *Recerca del funcionament de l'optimitzador DEFO (9 dies):* un cop escollit l'optimitzador en el que es basarà el projecte, aprofundirem sobre l'algorisme i les estructures de dades que suporten l'optimitzador en qüestió.
- *Recerca de les dades d'entrada i sortida de RouteNet (5 dies):* es procedirà després en comprendre superficialment com funciona el model de RouteNet, i quins són els seus requeriments d'entrada i sortida de les dades.

2.2. Implementació

Un cop finalitzada l'etapa de recerca sobre el projecte, ens disposarem ja a implementar el codi de l'optimitzador (53 dies en total). Aquesta és la fase més arriscada del projecte, degut a possible existència de errades de disseny o implementació del codi, que poden causar desviacions en els terminis del projecte.

- *Implementació de l'UML del codi (5 dies)*: ja amb els coneixements necessaris dels dos factors crítics del projecte, l'optimitzador i el model, ens dirigirem a dissenyar com haurà de ser el codi per tal que llegeixi l'estat inicial de la xarxa, generi un model, el transmeti a RouteNet perquè l'avaluï, i torni a iterar en funció del resultat. Es centrarà la modularitat de la solució com a objectiu clau, degut a la necessitat de reutilitzar el codi per als diferents optimitzadors.

- *Implementació de la lectura de les dades i la seva interfície (3 dies)*: en tant que tots els optimitzadors hauran de llegir les mateixes dades d'entrada, es plantejarà una interfície perquè pugui ser utilitzada per cadascun d'ells.
- *Implementació de l'optimitzador descrit per DEFO (14 dies)*.
- *Implementació de la interfície de RouteNet (5 dies)*: amb mateixa idea en ment que amb la lectura de dades es crearà una única interfície per on els 3 optimitzadors consultaran les latències dels models generats.
- *Validació de l'optimitzador (1 dia)*: un cop implementat tot allò que l'optimitzador necessita per executar-se, s'avaluarà el seu correcte funcionament en xarxes de prova en la cerca de resultats insatisfactoris.
- *Implementació del segon optimitzador modificant els heurístics (7 dies)*: en tant que arribat a aquest punt l'optimitzador DEFO ja és funcional, el risc de complicacions en la seva modificació és molt més baix que durant el seu primer desenvolupament.
- *Validació del segon optimitzador (1 dia).*
- *Implementació del tercer optimitzador (17 dies)*: en cas d'anar bé de temps, s'implementaria un optimitzadors basat en una cerca per amplada, el qual s'allunyaria del mode de navegació del graf que representa la xarxa, i per tant, el codi seria difícil de reutilitzar.
- *Validació del tercer optimitzador (1 dia).*

2.3. Avaluació

Durant aquesta fase posarà en rellevància els resultats obtinguts de la comparació entre l'algorisme d'optimització anterior amb el recent implementat (7 dies en total) . Aquesta fase també té pocs riscos implicats en ella.

- *Execució dels optimitzadors amb múltiples xarxes (1 dia)*: s'executaran els 2 o 3 optimitzadors implementats per a la posterior avaluació.
- *Avaluació de l'eficàcia dels optimitzadors (7 dies).*

2.4. Tancament del Projecte

Un cop arribada a aquesta fase del projecte, l'única feina restant és, principalment, enllestir la documentació (28 dies en total).

- *Poliment del codi (2 dies):* consistirà en acabar d'eliminar elements no usats i afegir els comentaris adients per a la fàcil comprensió del mateix.
- *Redacció de la memòria (19 dies).*
- *Creació i preparació de la presentació (8 dies).*

Cal remarcar que, com ja s'havia comentat en algun cas durant aquest apartat, hi ha tasques on el risc és moderat o elevat a nivell temporal. Aquestes tasques són aquelles en les que s'implementa el codi o es valida la seva correctesa. Per aquest motiu s'ha fixat un marge de seguretat de 16 dies, els quals serviran en el pitjor dels casos com a coixí per a imprevistos.

2.5. Diagrama definitiu

Les complicacions patides durant el treball, han causat l'abandó de la implementació del tercer optimitzador, i la modificació del diagrama de Gantt de la forma següent:

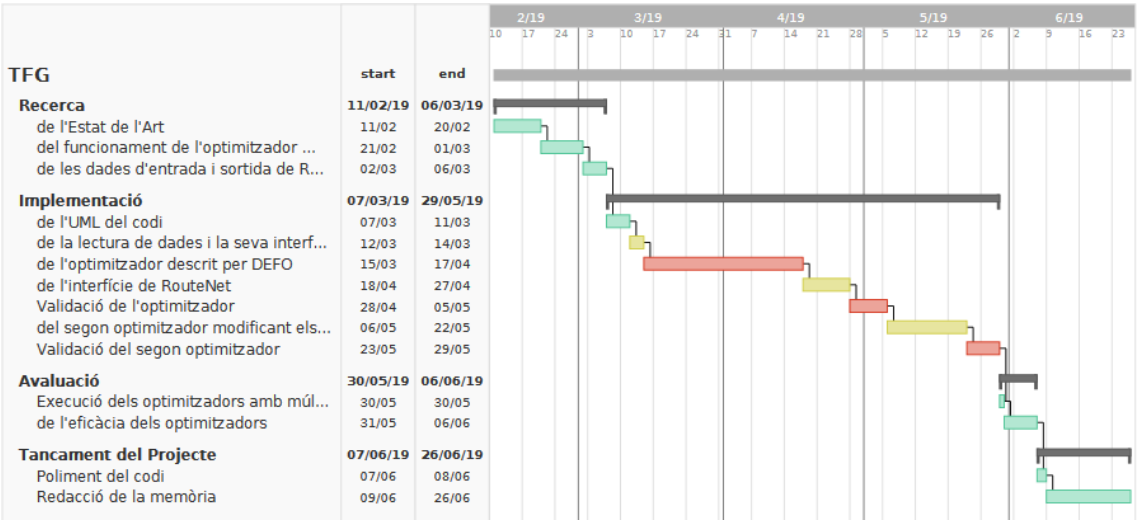

Fig. 2.2 Diagrama de Gantt final.

Tot i l'existència de dues llibreries de *Constraint Programming* per a Python3 ("python-constraint" [12] i "OR-Tools" [13]), la inexperiència amb el llenguatge amb la falta de documentació de les mateixes ha impossibilitat l'ús d'aquestes per a la resolució del projecte. Aquest fet ha implicat modificar el nucli mateix de l'algorisme de DEFO, passat de girar al voltant del Constraint Programming a un rudimentari *Depth First Search* (DFS).

Juntament amb senzilles complicacions durant l'acoblament del codi a RouteNet, han fet que el diagrama es distorsioni fins al diagrama final, presentat en la figura anterior.

3. Gestió econòmica i sostenibilitat

A continuació procedirem a estimar el cost del desenvolupament del projecte tant a nivell econòmic com a nivell mediambiental.

3.1. Pressupost del projecte

Per estimar el pressupost del projecte tindrem en compte els costos del hardware el software, els recursos humans, els imprevistos i els costos indirectes que aquest treball implica.

3.1.1.Hardware

Els recursos destinats en el hardware del projecte es mostren en la taula següent:

Producte	Preu	Unitats	Vida útil	Amortització
Lenovo IdeaPad320	500€	1	3 anys	166,67€
Pantalla Asus vc239	60 €	1	6 anys	10€
Total	**560€**			**176,67€**

Taula 3.1 Cost del hardware.

3.1.2.Software

De la mateixa forma que en la taula anterior hi teníem les despeses en hardware, en la taula següent hi tenim les despeses en software del projecte:

Producte	Preu	Vida útil	Amortització
Github	0€		0€
GitKraken	0€		0€
Mozilla Firefox	0€		0€
Ubuntu on Windows	0€		0€
Python3	0€		0€
Visual Studio 2017	0€		0€
TensorFlow library	0€		0€
Office 365	126€	1 any	126
Total	**126€**		**126€**

Taula 3.2 Cost del software.

3.1.3.Recursos Humans

En la taula següent es mostren les despeses destinades als recursos humans del desenvolupament del projecte:

Rol	Hores	Preu per hora	Preu total
Cap de projecte	50	50€	2.500€
Enginyer de Software	70	35€	2.450€
Programador	250	30€	7.500€
Analista	100	35€	3.500€
Total	470		**15.950€**

Taula 3.3 Cost dels recursos humans.

3.1.4.Imprevistos

A continuació hi ha la taula dels recursos reservats a problemes o desviacions de temps que hi puguin arribar a haver durant la programació i testejat del projecte:

Rol	Hores	Preu per hora	Preu total
Cap de projecte	10	50€	500€
Enginyer de Software	10	35€	350€
Programador	25	30€	750€
Analista	10	35€	350€
Total	55		**1.950€**

Taula 3.4 Cost dels imprevistos.

3.1.5.Indirectes

Les següents despeses que deriven indirectament del procés de desenvolupament del projecte i que, per tant, també s'han de tenir en compte:

Producte	**Preu**	**Unitats**	**Cost**
Consum elèctric del portàtil	0,12€/kWh	470h * 0.06W	3,38€
Internet	30/mes €	4 mesos	10€
Aperitius i altres productes de consum	5€/setmana	16 setmanes	80€
Total			**93,38€**

Taula 3.5 Costos indirectes.

3.1.6.Pressupost total

Amb la suma dels pressupostos anteriors podem concloure que el pressupost total del projecte serà el següent:

Concepte	**Cost aproximat**
Hardware	176,67€
Software	126€
Recursos humans	15.950€
Imprevistos	1.950€
Indirectes	93,38€
Total	**18.296,05€**

Taula 3.6 Cost total del projecte.

3.1.7.Control de desviacions

El principal problema que pot afectar al pressupost final prové d'una desviació temporal en la implementació del codi. En tal cas, es reorganitzarien les tasques en el diagrama de Gantt per tal d'assegurar la fi del projecte el termini pactat, i conseqüentment, en el seu pressupost. Tot i així, cal remarcar que degut a la naturalesa del projecte, un cop acabada l'etapa d'implementació del codi, la probabilitat de noves desviacions en les etapes finals és pràcticament inexistent.

3.2. Informe de sostenibilitat

En aquesta secció ens disposarem a analitzar l'impacte social, econòmic i ambiental que suposa el nostre projecte per a la societat.

3.2.1.Dimensió social

A nivell personal, aquest projecte em permet conèixer el funcionament intern d'un grup de recerca, a part de practicar la compressió i implementació d'algorismes complexos descrits en articles científics.

A nivell social, aquest projecte permetrà millorar l'eficiència d'un nou controlador de SDN, la qual cosa permetria aprofitar millor les infraestructures actuals que suporten aquesta tecnologia. Reflectint-se en els usuaris d'aquestes xarxes en latències més baixes.

3.2.2.Dimensió econòmica

Com ja s'havia comentat anteriorment, aquest projecte permet optimitzar l'ús de les infraestructures de les SDN, i per tant, allargar-ne la vida útil abans de que sigui necessari el seu reemplaçament per tal de satisfer la creixent demanda de la xarxa.

3.2.3.Dimensió ambiental

Tal i com havíem calculat anteriorment, durant la realització del projecte el portàtil haurà consumit 28,2 kWh, lo que suposa la producció de 10,43 Kg de Co2. No obstant, aquesta quantitat és molt petita en relació al efecte que pot arribar a tenir el controlador si és adoptat pel mercat, ja el reduiria nombre de hardware que haurien d'adquirir els propietaris de grans xarxes, i amb aquests, l'impacte mediambiental que té la seva producció.

4. Implementació

En aquest apartat ens disposarem a implementar el codi dels optimitzadors. Primerament s'exposaran les parts comunes en ambdós optimitzadors, com ara la lectura i generació de les dades, estructura de les variables on es guardaran i l'acoblament a RouteNet. Finalment, explicarem el funcionament de cada optimitzador per separat.

4.1. Lectura i generació de les dades

Tal com s'havia comentat en l'estat de l'art, els optimitzadors necessitaran llegir una topologia i una matriu de tràfic per tal de poder generar una matriu d'encaminament i poder fer servir RouteNet. Aquestes dades les obtindrem a través de dos canals: un *dataset* d'entrenament amb topologia NSFNet [14] i els generats per nosaltres mateixos.

En el *dataset* només disposa d'una única topologia, la qual conté un graf de 14 nodes connectats a través de 21 enllaços. El fitxer que la conté és té per nom "NetworkNsfnet.ned", i té l'estructura següent:

```
package networksimulator;
network NetworkNsfnet
{
   parameters:
          nombre de nodes
  ...
          datarate
...
connections:
      nombre indefinit de connexions entre 2 nodes
      ...
```

Aquest fitxer és analitzat dins del fitxer “readData.py”, concretament en la funció “readTopo”, del qual se’n extreuen els paràmetres més importants, com són el nombre de nodes de la topologia, l’ample de banda dels enllaços (*datarate*) i, finalment, els enllaços de la topologia. Aquesta informació és emmagatzemada llavors en les següents estructures de dades:

- Una llista d’adjacències emmagatzemada en forma de llista de llistes.
- Un diccionari que té per clau una parella d’identificadors dels nodes A i B (una parella de naturals), i per valor una parella de naturals, els quals es corresponen a l’identificador del port d’A que connecta a B i l’identificador de l’enllaç A-B respectivament.

En el mateix *dataset* també hi tenim fitxers amb matrius de tràfic, ubicats dins dels directori de “delaysNsfnet”. Aquest però es troben en un format especial: compreses en línies. Cada document dins del directori està format per 500 línies, on en cada línia es troba seqüencialment: una matriu d’14x14 demandes, que es corresponen a la matriu de tràfic; 8 matrius més de 14x14 generades a través dels resultats de simulacions, les quals són totalment irrellevants pel nostre

projecte; i finalment un 0 que es correspon a la pèrdua de paquets durant la simulació, el qual tampoc ens aporta res pel treball.

Així doncs, en la funció “readDemands” del fitxer “readData.py” llegim la primera matriu de tràfic del fitxer desitjat, i l’emmagatzemem com un diccionari on la clau és la parella d’identificadors dels nodes A i B, i el valor la demanda que hi ha d’arribar a B sortint d’A. El motiu pel qual la matriu de tràfic està emmagatzemada en un diccionari en comptes d’una llista de llistes és simplement tècnic, ja que en Python3 existeix una funció que ens permetrà ordenar les claus del diccionari en funció del seu valor molt fàcilment. Aquesta funció ens serà molt útil més endavant.

Finalment, en el *dataset* també hi podem trobar les matrius d’encaminament fetes servir durant les simulacions. Es troben ubicades dins del directori “routingNsfnet”, però en tant que l’objectiu dels optimitzadors és generar-ne, no se’ls ha hagut de donar cap mena d’ús.

La matriu de restriccions de latència, en tant que és un element que pertany a la naturalesa d’optimització del projecte, l’hem hagut de generar nosaltres mateixos. Aquesta la generem en el fitxer “dataGenerator.py”, en la funció “genCustDelayRest”. La finalitat de la funció és generar un fitxer anomenat “CustomDemand_*N*_*M*.txt”, on *N* és el nombre de nodes de la topologia, i *M* l’identificador del fitxer. En aquests fitxers de text s’hi troba una matriu NxN, emplenades aleatòriament seguint una distribució normal.

Amb aquestes funcions ja seriem capaços de satisfer les entrades dels dos optimitzadors, però necessitarem més topologies per a poder avaluar amb millor precisió els seus resultats. Per aquest motiu tenim dins del fitxer

“dataGenerator.py” la funció “genCustTopo”, la qual genera topologies noves amb el nombre de nodes que se li passi com a paràmetre. Les topologies són generades amb en fitxers que contenen el nombre de nodes, l’ample de banda dels enllaços i la matriu d’adjacència dels seus nodes. També es disposa de la funció “readCustTopo” per a llegir fitxers amb l’estructura anterior, i per acabar, de la funció “genDemand” per generar demandes seguint el format del *dataset*.

4.2. Estructura de les variables

Les variables tenen com a finalitat emmagatzemar la informació sobre l’estat d’un camí entre dos nodes. Aquesta estructura serà crucial pels dos optimitzadors, tal i com podreu apreciar d’aquí dos apartats.

Les variables d’aquest projecte són les mateixes descrites en el *paper* de DEFO [8], basade en la implementació d’un *sparse-set*. Un *sparse-set* està format per dues llistes, una anomenada *nodes* i una altre anomenada *map*, i dos punters, anomenats *V* i *R*:

- *Nodes*: En la llista *nodes* s’hi desen els identificadors de tots els nodes de la topologia i la seva posició en la llista indica en quina seqüència s’ha accedit a ells.
- *Map*: La llista *map* les posicions representen els identificadors dels nodes, i els valors de cada posició es corresponen a la posició del node en la llista *node*.
- *V*: Els valor de *nodes* que es troben en l’interval [0,*V*] es corresponen als nodes visitats en ordre.
- *R*: Els valors de *nodes* que es troben en l’interval [*V+1*,*R*] es corresponen als nodes candidats a ser visitats en el següent pas.

A més, per facilitar *backtracks* en els optimitzadors, s'ha afegit la llista de llistes *candidateList*, en la qual s'emmagatzemen el conjunt de candidats de cada iteració. En la figura següent hi trobem un exemple del seu funcionament:

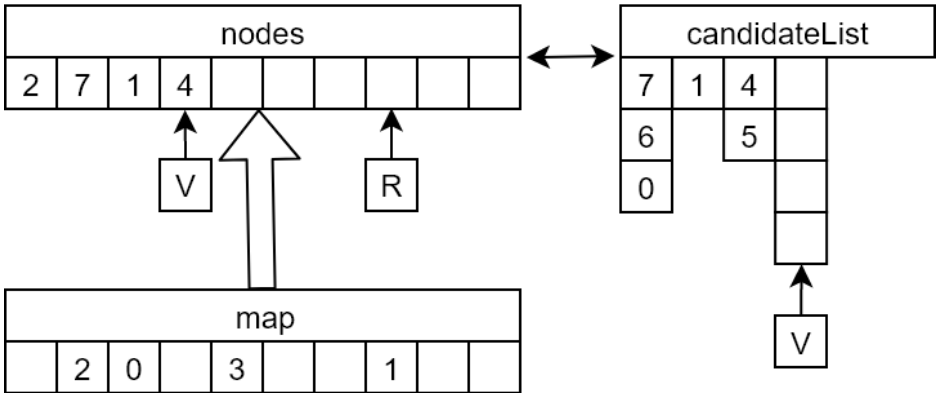

Fig. 4.1 Exemple d'sparse-set.

En tant que en la primera posició de *nodes* hi tenim el 2 sabem que el node 2 és l'origen d'aquest *sparse-set*. Com podem apreciar, en la posició 2 de *map* hi tenim el 0, que és la posició en la que es troba 2 en *nodes*. D'allò que no podem obtenir cap mena d'informació d'en quin node donarem la seqüència per acabada. Si mirem en la primera fila de *candidateList*, els següents candidats a continuar la seqüència eren 7, 6 i 0. Això vol dir que per $V = 0$, les posicions de *map* 7, 6 i 0 es trobaven dins de l'interval $[V+1, R]$, que en tant que $V = 0$ i només tenim 3 candidats, R era igual a 3. Com que l'escollit ha estat 7, a la *map*[7] hi trobarem un 1, i aquesta serà la posició que ocuparà a *nodes*. Aquesta lògica s'aplicaria per la resta de valors.

Per acabar aquesta secció, és necessari parlar de com s'afegeixen i eliminen elements en la seqüència de nodes visitats. Aquestes funcions es troben implementades en el fitxer "sparseSet.py". Per inicialitzar un *sparse-set* és necessari passar-li la llista d'adjacències de la topologia, per tal de poder saber quins són els candidats de cada node. La funció "visited" necessita per paràmetre l'identificador d'un node A, i el seu funcionament és el següent:

1. Incrementa la posició de *V*.
2. Intercanvia les posicions del node que es troba a *nodes*[V] amb el node A.
3. Actualitza la llista de candidats per a A.

Els candidats del node A seran aquells que formin part de la llista d'adjacències de A i no formin part encara del conjunt de nodes visitats, és a dir, que la seu valor en *map* sigui més elevat que *V*.

Finalment tenim la funció "removeVisited", la qual:

1. Decrementa el valor de *V* en 1 per tal d'eliminar l'últim element visitat.
2. Elimina la llista de candidats que tenia assignat aquell últim element.
3. L'elimina de la llista de candidats del que provenia.
4. Restaura el llistat de candidats tant en *map* com en *nodes*.

4.3. Acoblament a RouteNet

Per tal de poder fer servir un model entrenat de RouteNet calen tres requeriments: tenir els imports adequats, inicialitzar un model i passar-li els fitxers correctes.

Els imports necessaris són els següents:

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot
import UPC as upc
```

Un cop satisfets els prerequisits és necessari inicialitzar un únic model de RouteNet en el codi, per tal de demanar-li les prediccions de latència. Per a fer-

ho caldrà haver entrenat o descarregat un model i ubicar-lo, en el nostre cas, en el directori “nsf_new_v1”. El codi per inicialitzar-lo és el següent:

```
tfe = tf.contrib.eager
tf.enable_eager_execution()
hparams =
   upc.hparams.parse("l2=0.1,dropout_rate=0.5,link_state_dim=16,pat
   h_state_dim=32,readout_units=256,learning_rate=0.001,T=8")
model = upc.ComnetModel(hparams)
model.build()
saver =  tfe.Saver(model.variables)
modelPath = os.path.join("nsf_new_v1", "model.ckpt-35000")
saver.restore(modelPath)
```

I per acabar amb aquesta breu secció, ja només ens cal passar-li els fitxers necessaris per a obtenir la seva predicció. Aquests són la topologia, la matriu de tràfic i matriu d’encaminament. La topologia és possible passar-li directament com a una llista d’adjacències, com ja teníem de llegir les dades. Malauradament, la matriu de tràfic i la matriu d’encaminament són necessàries passar-les-hi com a fitxers de text. Això és degut a que per poder-les passar com a variables de Python3 es requeria modificar part del codi intern de RouteNet i no disposàvem del temps suficient.

El codi per a l’execució de RouteNet es troba dins de cada optimitzador (“optimizerA.py” i ”optimizerB.py”), en la funció “executeRouteNet”, el qual rep per paràmetres els directoris als dos fitxers mencionats anteriorment. Cal remarcar, però, que perquè el l’RouteNet generi una predicció, la matriu

d'encaminaments no pot generar un cicle entre si mateixos, és a dir: no pot donar-se el cas en que el conjunt nodes pels que ha de passa el node A per arribar al node B contingui un node C que per l'encaminament cap a B torni a apuntar a A. La matriu d'encaminament té una aparença com la següent:

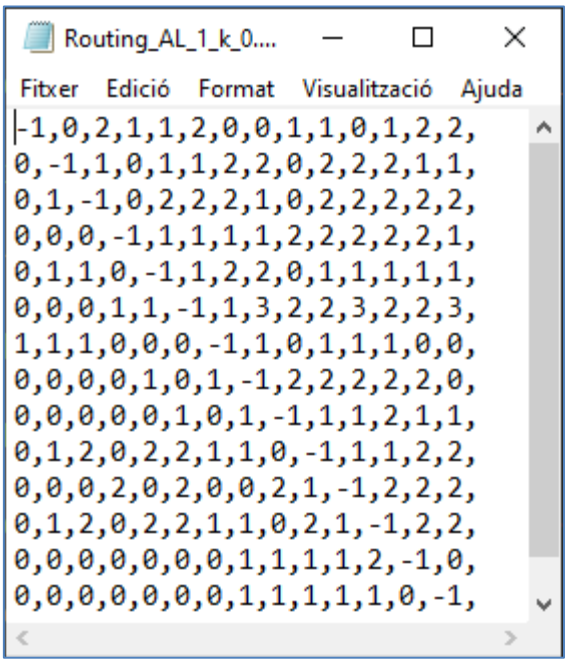

Fig. 4.2 Matriu d'encaminament per una xarxa de 14 nodes.

RouteNet retorna com a predicció una matriu de latències en segons de NxN-1 en una sola fila, ja que no té sentit obtenir les prediccions de latència d'un node a si mateix.

4.4. Optimitzador basat en DEFO

Finalment, a continuació procedirem a exposar el funcionament del primer optimitzador: l'optimitzador basat en DEFO. Tal i com s'havia explicat durant l'estat de l'art, DEFO està fonamentat en les tècniques de *Constraint Programming* i LNS.

A través del Constraint Programming DEFO obté un encaminament que procura minimitzar la saturació global de la xarxa. Per a fer-ho assigna a cada parella de nodes de la xarxa una variable que representarà el camí que recorrerà la

demanda per anar de l'origen al destí. Les variables del problema les defineix com a *sparse-sets*. A continuació defineix un seguit de restriccions que modelaran quins camins seran vàlids i quins no. Aquestes són:

- *La canalització*: consisteix en assegurar que cada enllaç no pugui transportar més informació de la que el seu ample de banda li permet.
- *La llargada*: aquesta restricció bé donada pel problema de *Segment Routing*, en la qual el nombre de nodes pels que pot arribar a passar un paquet està limitat per la mida màxima que pot tenir la llista que els conté. Consisteix, com us podreu imaginar, en limitar la longitud màxima que pot arribar a tenir un camí.
- *La concatenació de serveis*: consisteix en assegurar en les parelles origen-destí per les que es defineixi que un conjunt de nodes s'accediran sempre en un ordre establert.
- *El DAG (Directed Acyclic Graph)*: pretén assegurar que els camins siguin acíclics, tot i que aquesta restricció ja estaria coberta per la forma en que s'han implementat les variables.
- *El Cost Màxim*: aquesta restricció és l'aproximació que fa servir per simular les latències. DEFO entén les latències com a constants dels enllaços, que venen fixades per la topologia. Així que aquesta restricció pretén assegurar que un camí no supera una latència màxima de forma semblant a com assegurava que no excedia l'ample de banda.

Un cop definides les restriccions, totes les proposicions lògiques generades s'envien a un solucionador de SAT, per tal d'obtenir una configuració de la xarxa satisfactòria.

Un cop obtinguda aquesta configuració entra en joc el LNS. En tant que l'objectiu de DEFO és minimitzar la saturació de la xarxa, procedeix a seleccionar el conjunt dels enllaços més saturats d'aquesta. Seguidament d'aquest fa una selecció aleatòria i d'aquells que hagin estat seleccionats n'obté els camins que passen a través d'ells. Aquest camins els torna a recalcular amb *Constraint Programming* deixant tal i com es trobava la resta de la solució. La nova solució accepta sempre i quan compleixi les següents restriccions:

- El conjunt dels enllaços més saturat no pot créixer.
- La saturació dels enllaços més saturats no pot créixer.
- La resta del enllaços no-saturats poden saturar-se més sempre que es mantinguin sota del llindar pel qual l'enllaç es considera saturat.

Si una solució no compleix una de les restriccions anteriors, queda descartada i es torna a fer una selecció aleatòria i es repeteix el procés fins que: A) es millori la solució; o B) que es doni per finalitzada l'execució sense haver assolit una solució on es satisfacin tots els llindars de saturació.

Doncs bé, el nostre primer optimitzador pretén simular el procés que realitza DEFO però amb una canvi considerable: en comptes de fer servir latències estàtiques en cada enllaç a través de la restricció de Cost Màxim es farà servir RouteNet a final de cada iteració del LNS. D'aquesta manera l'algorisme hauria de tenir més precisió a l'hora de preveure la latència. No obstant, tal i com s'havia comentat l'apartat de Metodologia, s'ha acabat optant per fer servir un DFS.

El pseudocodi d'aquest optimitzador es troba en el fitxer "optimizerA" i vindria a ser el següent:

Inicialitza el model de RouteNet

Inicialitza les dades del problema

Obté parelles de nodes en funció del valor de la seva demanda

Per tantes vegades com es vulgui iterar el LNS:

Busca una configuració de camins vàlida

Genera la matriu d'encaminament associada a aquesta

Executa RouteNet

Comprova la qualitat de la solució

Si la solució és satisfactòria:

Surt del cicle de LNS

Si la solució és millor que l'anterior:

Passa a fer servir la nova solució

Prepara la nova iteració del LNS

Altrament:

Prepara la nova iteració del LNS

Retorna la predicció de les latències final

En tant que la inicialització de RouteNet ja ha estat comentada en l'apartat anterior, no tornarà a ser explicada.

Les variables inicialitzar en aquest problema són la matriu d'adjacències (la variable "link" del codi), el diccionari que associa una parella de nodes amb el port de sortida i l'identificador del seu enllaç ("link2port" en el codi), la llista de la capacitat en ús de cada enllaç inicialitzada a 0 ("linkCapacity"), la longitud màxima que podran tenir el camins ("maxLength"), la matriu de tràfic ("demand"),

un diccionari de *sparse-sets* per representar cada camí i una llista de llistes per emmagatzemar la matriu d'encaminament a mesura que va avançant la cerca ("routePort").

Un cop inicialitzades les dades inicials del problema, s'obté un llistat de parelles de nodes origen-destí ordenats decreixentment en funció del valor de la seva demanda de tràfic. Aquest serà l'ordre en que s'assignaran els camins dins del DFS.

A continuació, es procedeix a obtenir una configuració de camins vàlida. Per vàlida entenem que compleix les restriccions totes les restriccions del *Constraint Programming* de DEFO (a excepció de la concatenació de serveis), les quals es troben definides en les funcions de "channeling" i "lenght", juntament amb la funció "portLoop", que controla que no es puguin donar cicles en la matriu d'encaminament . La generació dels camins es troba dins de la funció booleana "getRoute". Aquesta funció és una funció recursiva que rep com a paràmetres la llista de parelles de nodes pels quals ha de computar un camí nou, i un iterador per navegar per tal llista. Té com a cas base que l'iterador arribi al final de la llista, i s'hi pot arribar en dues situacions: si els dos nodes de la parella, l'origen i destí del camí, apuntats per l'iterador són el mateix, i per tant, no hi ha cap mena de camí a computar; i si el node de destí forma part de la llista de candidats del *sparse-set* del camí i és una opció vàlida. També té el cas base en que no hi ha cap candidat vàlid per continuar avançant amb el DFS, i la funció retorna *False*. Les altres possibles situacions són 3:

- *Que el node d'origen i destí siguin el mateix en un iterador intermedi*: en tal cas, es retorna el resultat de "getRoute" pel següent iterador.

- *Que el node de destí sigui un candidat vàlid del sparse-set en un iterador intermedi*: en tal cas s'afegeix com a visitat en el *sparse-set*, s'incrementa la capacitat de l'enllaç entre l'últim node visitat i el de destí pel valor de la seva demanda. El resultat de "getRoute" s'obté pel següent iterador.
- *Que un node candidat qualsevol del sparse-set sigui vàlid en un iterador intermedi*: s'afegeix com a visitat en el *sparse-set*, s'incrementa la capacitat de l'enllaç entre l'últim node visitat i el candidat pel valor de la demanda. Finalment s'obté el resultat de "getRoute" pel mateix iterador.

Cal destacar que en aquesta última opció, per obtenir els candidats es fa servir un resultat precomputat del problema *All Pairs Shortest Path* (APSP), per tal d'obtenir inicialment els candidats més pròxims al destí. En cas de que dos o més candidats siguin vàlids i tinguin el mateix valor per APSP, s'escolliran en un ordre aleatori.

En qualsevol dels casos, si el valor que retorna la crida a "getRoute" és *False*, es desfà tant l'increment de la capacitat de l'enllaç com la visita en el *sparse-set*, i es continua amb l'execució fins a arribar al segon cas base.

Un cop finalitzat el DFS, es genera el fitxer de la matriu d'encaminament a través de la funció "generateRouting", la qual es limita a llegir de la variable "routePort", ara ja emplenada.

Seguidament, es genera la predicció de RouteNet i s'avalua. L'avaluació en el LNS discerneix de l'avaluació que fa DEFO degut a que a través de RouteNet no es cal comprovar quin enllaç es troba més saturat per a poder trobar quines demandes de tràfic han de ser reubicades, sinó que podem anar directament a comprovar quines d'aquestes superen els límits de latència establers. Dit això,

en la primera iteració del LNS sempre s'accepta la predicció com a la millor opció, lògicament, però en la resta d'iteracions la cosa canvia. La predicció obtinguda passa per dues funcions més, "delayRestrictionState" i "checkLNS". La primera s'encarregua d'obtenir el conjunt de demandes que superen els límits de latència i el marge que hi ha entre la restricció i el resultat obtingut. L'altre comprova si aquest conjunt ha crescut o ha incrementat la diferència entre la restricció i la predicció.

Si el conjunt està buit, vol dir que totes les demandes compleixen les restriccions de latència, i per tant, que es troba davant d'una solució satisfactòria. En tal cas, es cessa l'execució del LNS i es retornen els resultats obtinguts de la predicció.

En el cas que s'obtingui un conjunt de demandes que superen els límits de latència amb algun element, es comprovarà que s'hagi millorat la solució. Si és el cas, es substitueix la millor solució anterior per la nova. Seguidament, tant si es millora com si no la solució, es crea una nova llista de parelles de nodes a través del conjunt anterior, es desfan els camins pels quals passaven aquestes demandes (a través de restar la seva demanda en totes les capacitats dels enllaços pels que passaven) i es torna a l'inici del cicle.

4.5. Optimitzador basat en RouteNet

En tant a que aquest optimitzador recicla parts del l'optimitzador basat en DEFO, ens disposarem a marcar amb especial èmfasi en els segments distintius.

El pseudocodi d'aquest segon optimitzador és el següent:

Inicialitza el model de RouteNet

Inicialitza les dades del problema

Obté parelles de nodes en funció del valor de la seva demanda

Busca una configuració de camins satisfactòria

Si "getRoute" retorna *False*:

    Retorna que no s'ha trobat cap solució

Altrament:

    Retorna els resultats obtinguts

El major canvi que es pot apreciar és la desaparició del cicle de LNS. Això és degut a que en aquest optimitzador s'ha optat per integrar a RouteNet dins del DFS. Així el DFS es capaç de saber si la solució que està generant és una solució viable o no abans d'acabar-la, i conseqüentment, només retornarà un resultat si troba una solució. Altrament, retornarà *False*. La resta de les funcions del pseudocodi es mantindran tal i com estaven en l'optimitzador anterior.

Entrant en més detall sobre el nou funcionament del DFS, es pot apreciar com la funció "getRoute" no ha estat alterada. Els canvis s'han produït dins de la funció "orderCandidates", encarregada de generar nous candidats vàlids per continuar amb la cerca. En aquesta funció ja no només es mira que els candidats siguin vàlids segons les restriccions de DEFO, tal com ho assegura la funció "checkCandidate", sinó que a més genera un fitxer que conté una matriu de tràfic intermèdia. Aquesta matriu només conté les demandes de tràfic dels camins ja visitats, i el camí que està recorrent actualment com si el candidat actual en fos el destí. Seguidament es genera la matriu d'encaminament intermèdia a partir d'un procés més complex. Això és degut a que, tal i com havíem comentat

anteriorment, RouteNet només funciona si la matriu d'encaminament no conté cap cicle intern. Per aquest motiu, quan s'està generant aquest fitxer intermedi, no es pot fer de qualsevol manera, sinó que s'ha d'assegurar que s'ompli com s'ompli no es generin cicles entre els ports dels camins ja assignats i els ports dels camins encara per recórrer. Per a poder fer tal assignació s'ha utilitzat una ordenació topològica [15] fixant el destí, on el criteri d'ordenació ha estat una altre vegada el APSP. Un cop ordenats, s'ha assignat el port més pròxim al destí, el qual no produís cap cicle.

Finalment, un cop generats tots els fitxers intermedis, ja només queda executar RouteNet. La predicció que ens retornarà serà doncs la predicció de l'estat actual de la xarxa. No cal oblidar però, que ens trobem en la funció "orderCandidates",i hem de generar una llista de candidats per a l'execució de DFS. Per aquest motiu es comprova a través de la funció "delayRestrictionCheck" que totes les latències dels camins recorreguts, inclòs l'actual, satisfacin les restriccions de latència impostes pel problema. Sinó és el cas, el candidat queda descartat. Altrament, el candidat passa a un diccionari, l'"heur", on s'emmagatzema l'identificador del node com a clau, i la latència del camí actual com a valor. Un cop s'ha repetit tot aquest procés per a tots els candidats, s'ordena la llista en ordre creixent i es retorna a "getRoute".

5. Anàlisis dels resultats

Per donar per finalitzat el projecte, només ens queda analitzar els resultats obtinguts després de múltiples execucions amb les mateixes dades en els dos optimitzadors. Les dades en les que ens centrarem seran el temps d'execució i les latències que ens proporciona el nostre codi. Començarem pels temps.

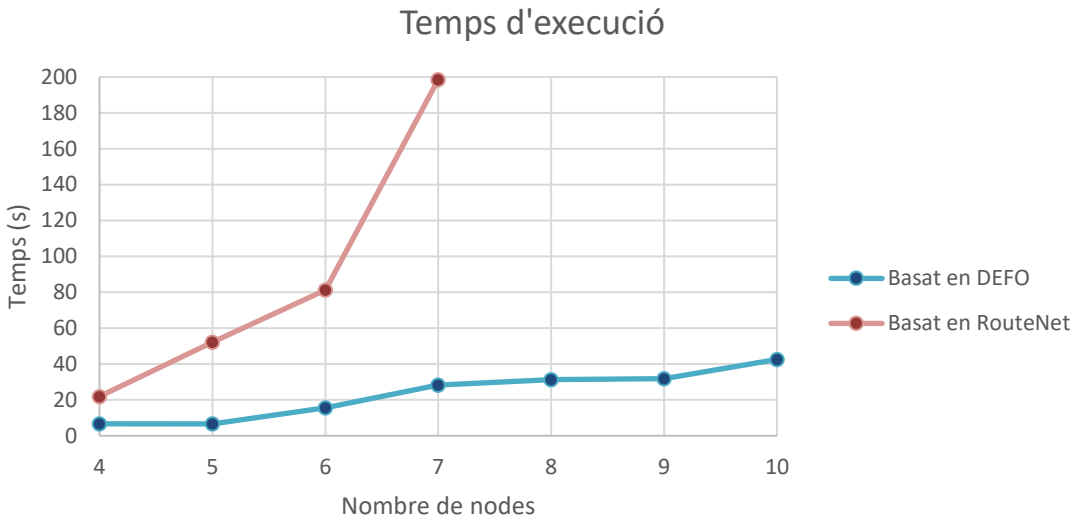

Fig. 5.1 Gràfica del temps d'execució dels dos optimitzadors.

En la figura anterior podem apreciar l'increment del temps d'execució a mesura que va augmentant el nombre de nodes que conformen la xarxa, i amb ells, els enllaços que els uneixen. Fixant-nos primerament en els resultats obtinguts per l'optimitzador basat en RouteNet, podem apreciar clarament com aquest codi es comporta de forma exponencial. La forma en si no és una sorpresa, ja que navegar per totes les possibles combinacions de branques d'un graf ja és exponencial de per si, sinó el seu creixement abrupte en comparació amb el primer optimitzador. Aquest fet és degut principalment a dues raons.

La primera d'elles és el funcionament de RouteNet. En tant que RouteNet ha estat entrenat amb resultats de simulacions, el model fet servir no s'ha trobat mai amb matrius de tràfic parcialment emplenades. Això resulta en resultats menys precisos del que s'obté quan se li passa la matriu plena, la qual cosa afecte al nombre de camins satisfactoris reals que acaba donant per vàlids.

L'altra d'elles n'és la generació de matrius d'encaminament parcials. Aquestes es generen a dins d'una funció que es crida en la majoria de crides del DFS, ja que els camins tendeixen a tenir més d'un enllaç. Malauradament, l'*overhead*

que comporta el control dels cicles en tal generació exagera encara més el comportament exponencial de l'optimtizador.

Per altre banda tenim l'optimitzador basat en DEFO. Aquest optimitzador, en tant que també està implementat amb un DFS intern, també mostrarà un comportament exponencial, però és més subtil, degut a que aquest optimitzador pot tornar tres classes diferents de resultats: solucions trivials, satisfactòries i insatisfactòries. Les solucions trivials són el cas en el que ens trobem per les xarxes de 4 i 5 nodes, on les restriccions de latència eren lo suficientment elevades com per trobar una solució satisfactòria amb la primera passada del DFS, saltant-se així la necessitat continuar iterant en el LNS. En la xarxa de 6 nodes ens trobem davant d'una solució satisfactòria, en qual es va anar millorant el resultat a través del LNS fins a arribar a la vuitena iteració, en la qual va arribar a una solució satisfactòria. I finalment tenim les execucions de la 7 a la 10, en les quals després de les 10 execucions que té LNS per a trobar una solució satisfactòria no és capaç de trobar-la, i es veu forçat a retornar la millor solució a la que ha arribat. En aquest últim cas es pot apreciar com el temps d'execució creix més lleument amb l'augment dels nodes a la xarxa.

Tot i així, aquest optimitzador tampoc és prou bo com per ser capaç d'executar la topologia del *dataset* d'entrenament NSFNet de 14 nodes.

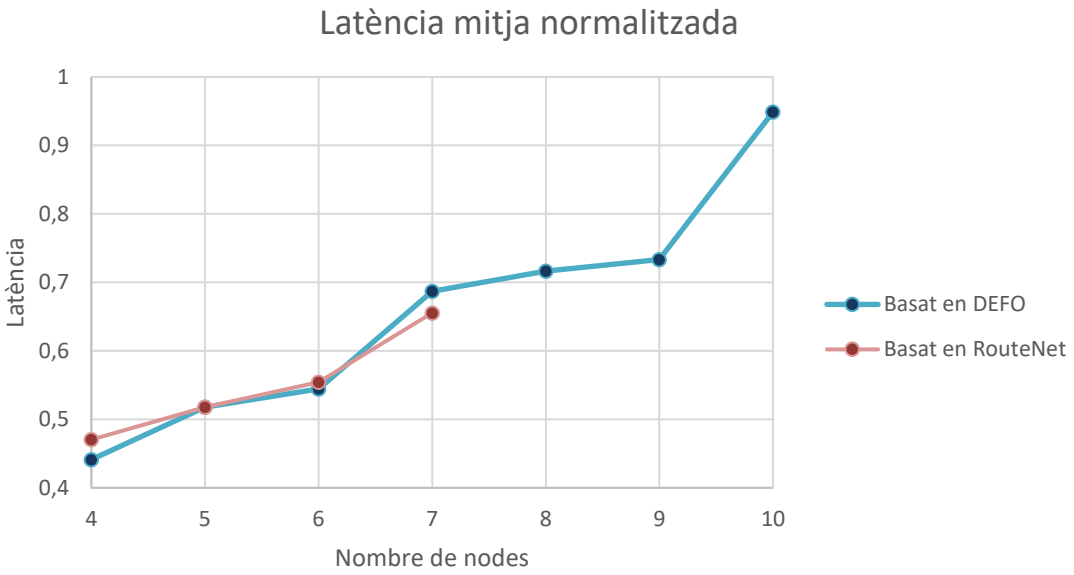

Fig. 5.2 Gràfica de la latència mitja normalitzada dels dos optimitzadors.

Després tenim la gràfica de les latències mitges normalitzades. Els resultats d'aquesta gràfica estan normalitzats a través de dividir entre 0,45. En aquesta gràfica podem apreciar com l'eficàcia dels dos optimitzadors es aparentment molt similar. Tot i així, cal remarcar que la impossibilitat d'obtenir més valors a partir dels 7 nodes per a les dues gràfiques impossibilita l'obtenció de resultats concloents.

6. Conclusions

Concloem aquest projecte de final de grau assegurant que, tot i que els codis implementats són funcionals, no s'han assolit completament els objectius establerts a l'inici del treball.

A través d'aquest projecte s'han assolit els conceptes teòrics esperats per a poder realitzar un optimitzador d'una qualitat pròxima al DEFO. No obstant, el fet que no s'hagi aconseguit fer servir una llibreria de *Constraint Programming* ha perjudicat molt negativament a l'eficiència del primer optimitzador.

Respecte el segon optimitzador, s'ha demostrat la necessitat de millorar part del codi intern de RouteNet per tal de fer-lo més eficient a l'hora de rebre les dades d'entrada, juntament amb la necessitat d'entrenar models amb matrius de tràfic més diverses.

També es necessari esmentar que en el segon optimitzador no és implementable fent servir *Constraint Programming* com a conseqüència de que no es possible generar variables si aquestes varien de valors a mesura que es va resolvent el problema, com seria el cas dels heurístics basats en RouteNet. Per aquest motiu, es podrien implementar en un futur altres tècniques per desplaçar-se en un espai de solucions de manera més eficient de com ho DFS pel nostre cas, com ho seria un algorisme basat en *Branch & Bound*.

Tanquem aquest projecte doncs afirmant que, tot i que els no han estat suficientment bons com per a donar per assolits els objectius inicials, el desenvolupament d'aquest prototip ha estat necessari per a, en un futur, implementar-los de nou millor a partir d'aprendre dels errors aquí comesos.

7. Bibliografia

[1] P. W. Battaglia *et al.*, “Relational inductive biases, deep learning, and graph networks,” pp. 1–40, 2018.

[2] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, “Neural Message Passing for Quantum Chemistry,” 2017.

[3] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, “Software-defined networking: A comprehensive survey,” *Proc. IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[4] Dinesh Thakur, “Routing Algorithms.” [Online]. Available: http://ecomputernotes.com/computernetworkingnotes/routing/routing-algorithms. [Accessed: 19-Jun-2019].

[5] “NSA uses OpenFlow for tracking... its network | Network World.” [Online]. Available: https://www.networkworld.com/article/2937787/nsa-uses-openflow-for-tracking-its-network.html. [Accessed: 22-Jun-2019].

[6] “Software-Defined Networking (SDN) Definition - Open Networking Foundation.” [Online]. Available: https://www.opennetworking.org/sdn-definition/. [Accessed: 20-Jun-2019].

[7] R. Hartert *et al.*, “A Declarative and Expressive Approach to Control Forwarding Paths in Carrier-Grade Networks,” *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 5, pp. 15–28, 2015.

[8] R. Hartert, P. Schaus, S. Vissicchio, and O. Bonaventure, “Solving segment routing problems with hybrid constraint programming techniques,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2015, vol. 9255, pp. 592–608.

[9] “What Is Constraint Programming? | constraint.org.” [Online]. Available: http://www.constraint.org/en/intro.html. [Accessed: 22-Jun-2019].

[10] “3.4.7 Large Neighbourhood Search - Local Search | Coursera.” [Online]. Available: https://www.coursera.org/lecture/solving-algorithms-discrete-optimization/3-4-7-large-neighbourhood-search-brB2N. [Accessed: 29-

May-2019].

[11] K. Rusek, J. Suárez-Varela, A. Mestres, P. Barlet-Ros, and A. Cabellos-Aparicio, “Unveiling the potential of Graph Neural Networks for network modeling and optimization in SDN,” no. i, 2019.

[12] “python-constraint · PyPI.” [Online]. Available: https://pypi.org/project/python-constraint/. [Accessed: 12-Mar-2019].

[13] “CP-SAT Solver | OR-Tools | Google Developers.” [Online]. Available: https://developers.google.com/optimization/cp/cp_solver. [Accessed: 12-Mar-2019].

[14] “Knowledge-Defined Networking Training Datasets.” [Online]. Available: http://knowledgedefinednetworking.org/. [Accessed: 24-Jun-2019].

[15] T. H. Cormen and T. H. Cormen, *Introduction to algorithms*. MIT Press, 2001.

[16] “Network congestion.” [Online]. Available: https://en.wikipedia.org/wiki/Network_congestion. [Accessed: 24-Jun-2019].

[17] M. Gendreau and J.-Y. Potvin, *Handbook of metaheuristics*. Springer, 2010.