Instasalle

Programació avançada i estructures de dades

By

Alex Almansa, Marc Llort

Departament d'enginyeria

La Salle URL

Gener 2019

Contents

1 Resum del enunciat

En aquesta segona part de la pràctica de InstaSalle, haurem de, amb els diferents algorismes mostrats a classe, dur a terme dues funcionalitats relacionades amb els servidors de la plataforma.

1.1 Funció 1: Disponibilitat

Aquesta primera funcionalitat, tracta de, donats dos fitxer json de servidors i nodes, trobar els camins entre el servidor origen i el servidor destí. Aquest camins que haurem de trobar seran dos, el primer s'encarregarà de maximitzar la fiabilitat, i el segon de minimitzar la distància.
Tant els valors de fiabilitat, com la distancia entre nodes, el trobarem a la variable reliability i dins del connectsTo, respectivament. Per trobar aquests camins haurem de fer servir: backtracking, branch bound, greedy, backtracking + greedy i branch bound + greedy.

1.2 Funció 2: Distribució de càrrega

En aquesta segona funcionalitat haurem de repartir els N usuaris entre els M servidors disponibles. Hi haurà dos factors que afectaran en la decisió d'afegir un usuari en un servidor o un altre.

Equitivitat de carrega La més important serà que, dins de certs marges decidits per nosaltres, els servidors estiguin balancejats en quant a la suma del nombre de hores que fa cadascun dels seus usuaris. Caldrà minimitzar la diferència d'activitat dels usuaris entre els diferents servidors.

Proximitat servidor - usuari Caldrà ordenar els usuaris en els servidors més propers depenent de la localització.

Per testejar de forma còmode les anteriors funcionalitats caldrà realitzar un menú amb les diferents opcions.

2 Espai de cerca

En aquest apartat ens centrarem en explicar quin és l'espai de cerca d'ambdós problemes, i com funciona el nostre programa per solucionar-los.

2.1 Funció 1: Disponibilitat

En un principi tindrem un Servidor, el qual serà el origen, el inici de la nostra cerca.
A partir d'aquest moment, és quan ja tenim amplada, els diferents nodes que es connecten amb el nostre servidor origen. En aquest moment, segons el algorisme que estem fent servir procedirem de diferents maneres.

Greedy En el cas de greedy mirarem quin dels nodes connectats té un valor de fiabilitat/cost (segons la opció seleccionada) que ens interessa més. A continuació farà un salt de profunditat en aquella opció que semblava més prometedora, ignorant completament la altra. Aquest procés es va repetint fins arribar a la solució. Per trobar la solució, es farà mirant les diferents opcions(amplada) de cada nivell de profunditat i mirant si algun dels nodes del nivell on es troba, coincideix que es tracti d'un dels que connecta amb el servidor origen.

Backtracking En el cas de backtracking, quan mirem les opcions que tenim (amplada), primer anirem solucionant la primera que analitzem, baixant un nivell en profunditat, fins arribar a la solució (la trobem de la mateixa forma que greedy). Un cop tenim la primera solució, anirem pujant en profunditat nivell a nivell, i a cada nivell que pugem realitzarem el recorregut de profunditat avall un alter cop, fins haver recorregut totes les opcions. En el següent apartat explicarem com hem fet la poda per estalviar-nos molts d'aquests recorreguts.

Branch Bound A cada nivell de profunditat, analitzarem les diferents opcions, mirant quines ens queden disponibles. Les diferents solucions que anem creant a cada salt de profunditat les guardarem ordenades en un array en ordre de més prometedora a menys, per així anar solucionant sempre la que sembla que serà la bona. En aquest cas el salt de profunditat és realitzarà sempre després d'haver analitzat totes les opcions (amplada)

2.2 Funció 2: Distribució de càrrega

A la funció 2, on hem de repartir els diferents usuaris entre els diferents servers ens trobem amb una situació molt similar a la del knapsack problema, simplement que en comptes de tenir una motxilla en tondríem X numero de servers.

És per això, que la amplitud serà la decisió de posar un usuari a un servidor o no, i la profunditat serà la acumulació d'aquests usuaris afegits a cada servidor. Sabrem quan hem acabat una solució, quant no quedin més usuaris per afegir a cap servidor, perquè tots han sigut ja repartits.

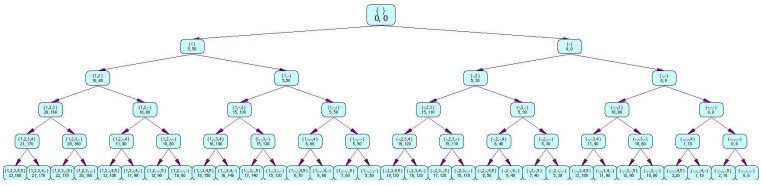

Figure 1: Knapsack de "un server"

3 Codificació

3.1 Funció 1: Disponibilitat

Per codificar els diferents algorismes, hem fet us dels apunts donats a classe i els hem adaptat a les nostres necessitats.
Un cop adaptat correctament el lector de json de la fase anterior per llegir correctament els nous fitxers de usuari vàrem crear les classes per les de Node i Server. La informació que recopila el lector la guardem en arrays dels tipus que llegeix.
En un primer moment, al realitzar la funcionalitat de disponibilitat, vam començar per escriure els tres algorismes sense cap tipus de poda, per aconseguir el funcionament bàsic. En el cas que cal minimitzar el cost de longitud, ens va ser molt fàcil realitzar la poda, ja que un cop teníem una solució, ja sigui generada per el propi algorisme, o per greedy en cas de que fem servir una combinació d'algorismes, només ens era necessària fer una comparació a la funció promising per així descartar la solució en cas de que sobrepasses el cost de la millor solució actual.
Per realitzar la poda quan es el cas de la fiabilitat, la manera de la que ho hem fet és una mica menys eficient, ja que hem de iterar per els nodes que li queden per recórrer i anar sumant el seu cost, per posteriorment poder comparar amb la millor solució que tenim fins el moment, el qual fa que el cost del nostre algorisme en el cas de calcular la fiabilitat creixi en N.
Per veure quins nodes ens falten, tenim una copia del array de nodes, la qual un cop iterem per la solució que estem tractant actualment per veure si ja ha passat per algun node, també anem borrant els nodes pels que ja ha passat del altre array, per tant ens quedem amb un array el qual té només aquells nodes pels que encara ha de passar.

Per detectar si hem arribat a la solució o no, al poder haver-hi més de un node que connecti amb el Servidor destí, hem hagut de modificar una mica el algorisme bàsic per fer-ho funcionar. Dins els diferents bucles dels diferents algorismes, envoltant el if que ens mira si es tracta de la solució del problema, tenim un for, el qual ens mira si el node on ens trobem és algun dels nodes que connecten amb el servidor destí. En cas de ser-ho, activem un booleà per a que un cop surti del for, no entri mai en el if de isPromising, i continuï trobant una solució la qual ja hauria passat per un node solució.
En quant a les solucions que son combinació de backtracking o branch bound amb greedy, primer cridem greedy, la qual ens retorna una solució que passarem per paràmetres a bactracking o branch, i dins la funció, la posarem com a solució best, per així poder realitzar una poda més ràpida i efectiva.

3.2 Funció 2: Distribució de càrrega

Aquest segon problema, de forma general ha sigut més ràpid de codificar donat que teníem la base del primer problema, i només hem tingut que adaptar-lo al nou problema per fer-lo funcionar correctament.
Cal recalcar que en el codi, al main.c, funció amb nom "funcio2", hi ha dos integers els quals ens permeten modificar el marge que es farà servir per calcular la equitivitat. Un serveix per la funció de backtracking i l'altra per la de branch and bound. Tot i això, hem tingut diversos problemes alhora d'adaptar backtracking i branch bound, sobretot aquest segon.

Greedy En el cas de greedy simplement vam tenir que modificar les diferents condicions alhora d'afegir un usuari a un servidor o no. Primer miràvem si la equitivitat estava dins de un marge regulable, en un inici l'hem posat a 2. En cas de retornar que hi ha equitivitat, agafarà la opció de usuari més aprop de server. En cas contrari, posarà el usuari al servidor que està més "buit".

Backtracking Alhora de fer backtracking, li passarem una solució, buida o no, que servirà per determinar el best, per si volem fer us de la combinació amb greedy.
Per fer la poda, apart de descartar aquelles opcions que tenien usuaris en dos servidors, hem fet us de anar mirant si la diferencia de equitivitat era més gran, que la equitivitat de la millor solució més un marge (que posem al executar el backtracking, ara hi ha 2), doncs descartàvem la solució. Alhora de determinar quina és la millor solució, hem decidit agafar la que té més equitivitat, però en cas de que una solució amb menys equitivitat sigui dins dels marges explicats anteriorment, agafarem aquella que tingui una millor "distancia". Aquesta distancia la hem determinat sumant les distancies de la solució que tenen els seus usuaris amb els seus respectius servidors, per tant simplement ens caldrà escollir aquella opció que minimitzi aquest valor.

Branch Bound Hem fet servir gairebé totes les mateixes funcions que a backtracking, les mateixes condicions... L'únic que canvia es la codificació, ja que es tracta del algorisme de branch and bound.
Hem tingut bastants problemes alhora de determinar com cal fer el problema de knapsack amb diferents motxilles, en aquest cas servidors. Ens ha sigut necessari fer us d'una poda més agressiva que en el cas de Backtracking, degut a que sinó el arbre de solucions anava creixent sense parar i no aconseguíem trobar mai una solució.
Per fer la poda més agressiva, el que hem fet a sigut descartar a la funció promising, apart d'aquelles solucions que tenien una equitivitat superior + un marge a la nostra millor solució, també hem descartat aquelles que tenint una equitivitat més gran o igual que la nostra solució. Finalment, tot hi haver tingut bastants problemes hem aconseguit arreglar-los i el funcionament ara és molt ràpid.

4 Comparativa d'algorismes

4.1 Funció 1

En aquesta secció tractarem de mostrar acompanyat de gràfiques la diferència entre l'eficiència d'uns i altres algorismes.

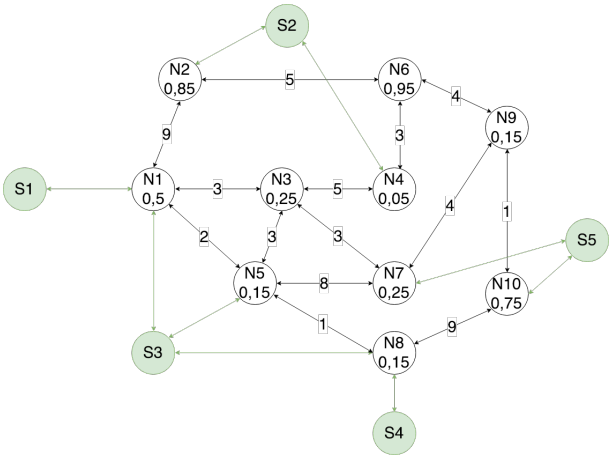

Figure 2: Esquema nodes++

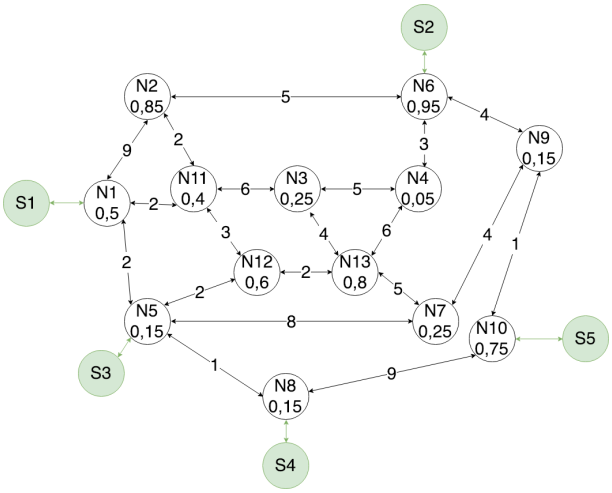

Figure 3: Nodes personalitzats

Les anteriors figures son el esquema dels nodes que hem utilitzat per a la realització de la pràctica. En primer lloc, la figura 2, és l'estructura que s'ens va proporcionar per fer la pràctica, anomenada servers i nodes plus, però com que amb aquesta no en teníem prou, ja que per realitzar les gràfiques necessitàvem el temps i amb aquesta els temps eren sempre 0, vam fer-ne una de més gran.

Així doncs, la figura 3, és l'estructura que hem creat nosaltres, a partir de la que ens donaven, per fer les proves i a la que hem anomenat nodes i servers plusplus.

Tot i que creiem que això ens seria molt útil a l'hora de fer els gràfics, hem acabat veient que no ho és tant ja que els temps segueixen sent molt baixos, tot i afegir nodes i connexions. Donat aquest problema, vam intentar solucionar-lo canviant la unitat de temps en que mesuràvem els temps de mili segons a nano segons, però vam adonar-nos que mesurar el temps amb nano segons és poc útil ja que els temps varien moltíssim en funció del que esta executant el ordinador en aquest moment i altres factors que no controlàvem. Llavors el que ens passava es que una mateixa tasca mesurada en nano segons podia tardar una vegada 90.000 ns i la següent 500.000 ns.

És per això que finalment hem decidit fer-ho amb aquest dataset, creat per nosaltres i comparar els algorismes no només segons el temps, en el que les variacions son relativament petites, sinó que també amb el nombre d'iteracions i si s'ha trobat o no la millor solució.

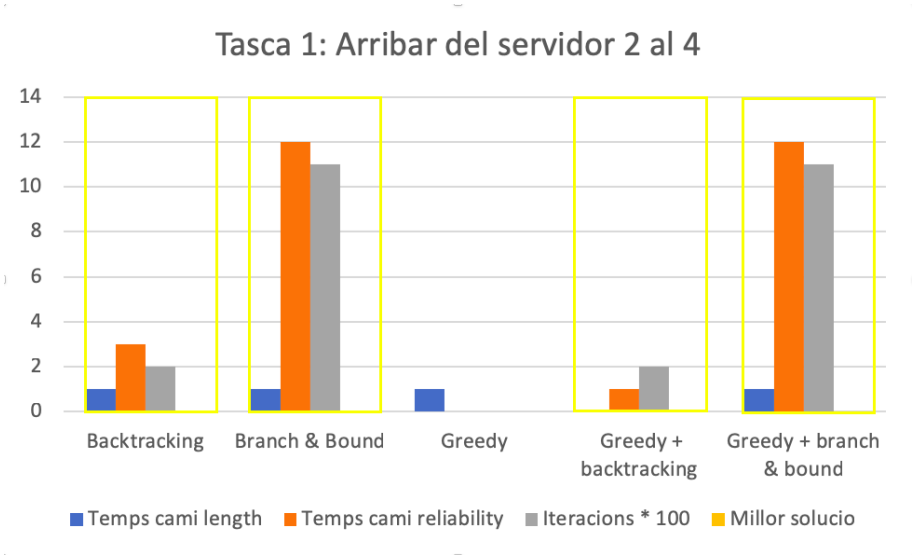

Figure 4: Gràfic 1

A la gràfica anterior es pot observar que els temps d'execució de la cerca del camí més curt, son bastant semblants tant pel bactracking com al branch and bound com al greedy, amb la diferència que el greedy no ha trobat la millor solució. Si ens fixem en la segona tasca, i en el nombre d'iteracions, observem com en aquest cas, el backtracking obté uns millors resultats passant de unes 200 crides a unes 1100 del branch and bound. En quant al greedy, hem de dir que tot i no trobar la millor solució, s'hi ha acostat molt i ho ha fet molt ràpid.

Si anem a les combinacions de greedy i backtracking i greedy i branch and bound, observem com la tendència és la mateixa, en el backtracking i greedy hem observat una petita millora, passant el temps de la tasca 1 de 1 a 0 i reduint el temps de la tasca 2. Al branch and bound i greedy, en canvi, no hem observat cap milloria, però tampoc cap empitjorament, és a dir, s'ha comportat igual.

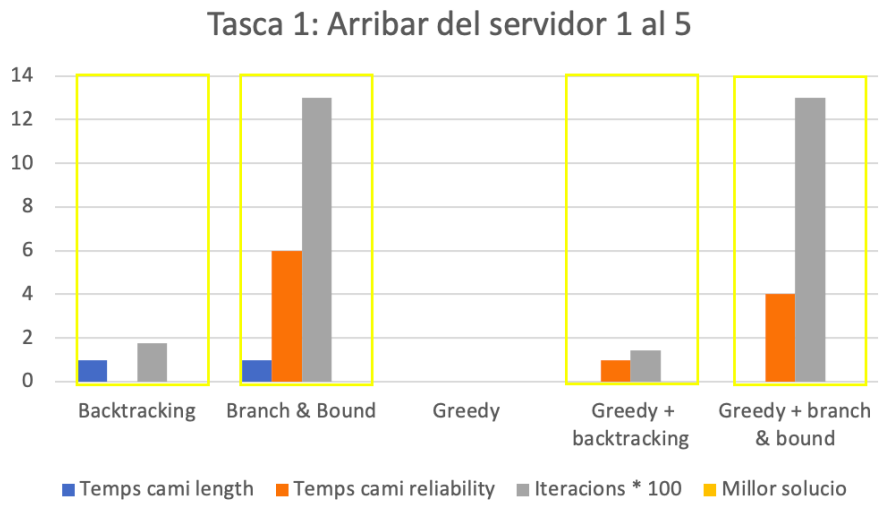

Figure 5: Gràfic 2

Aquesta gràfica representa exactament el mateix que la anterior però en aquest cas estem buscant els millors camins entre els servidors 1 i 5. D'aquest gràfic podem extreure algunes conclusions, en primer lloc, el greedy en aquest cas ha anat molt bé i en un dels dos camins si que ha trobat la millor solució i en l'altre no però s'ha quedat molt aprop. En quant als temps del backtracking i branch and bound, el backtracking s'ha comportat de manera semblant al anterior amb la diferència que al afegir-li el greedy ha millorat el temps en un dels dos camins però ha empitjorat en l'altre. Tot i això el nombre de crides al afegir greedy ha disminuït, per això podríem dir que ha millorat el global. En quant al branch and bound, que en la Gràfica 1 no havia tingut cap millora al afegir-li el greedy, en aquest cas si que ha tingut una millora, i ha sigut en els dos camins.

4.2 Funció 2

Al moment de realitzar aquesta segona funcionalitat de repartir els usuaris entre els diferents servidors disponibles, per intentar maximitzar la equitivitat, ens hem trobat amb que es tracta d'una tasca bastant més complexa en quant al temps d'execució. Això, és degut a que hi ha moltes més combinacions possibles en comparació amb el primer problema, i apart de tenir moltes més possibilitats també està el fet de intentar fer que els usuaris també estiguin aprop del seu servidor més proper.

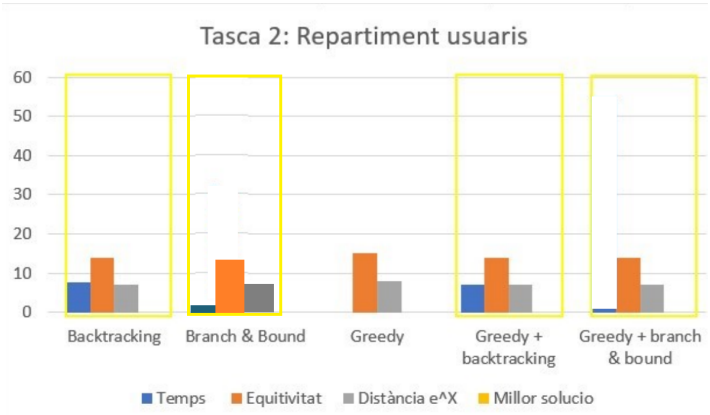

Figure 6: Gràfic 3

Primer, el backtracking ens trobem que és el més lent de tots, ja tarda un temps relativament baix (7.8s), i troba sempre la millor solució, ja que és la equitivitat més baixa possible i la distancia total respecte usuaris i els seus servidors també ha sigut minimitzada.
En el cas de backtracking ens trobem que al combinar-lo amb greedy no suposa cap tipus de millora en aquest problema, ja que a la primera iteració total de backtracking troba una solució que ja guarda al best, per tant la diferencia no es gaire notable. Depenent de la execució, a vegades va més ràpid el backtracking sense greedy.
Branch and bound és el algorisme que millor funciona en aquest problema. Un cop arreglats els problemes que teníem, trobem que el fet de que sempre treballi en la millor solució fa que acceleri molt el procés (167ms).
En cas de combinar-lo amb greedy encara millorem més el ja molt bon temps anterior, ja que tarda 35ms i ens troba sempre la millor solució.

Greedy és evidentment el més ràpid, i troba solucions bastant bones en quant a equitivitat, però no de distància. El temps és molt ràpid, ja que tarda 0 ms.

5 Mètode de proves

Per poder provar de forma més eficient el nostre programa ja des de un principi, vam crear el menú que hauríem de tenir a la versió final, per així poder introduir les dades de origen destí en el cas del primer problema i anar fent provatures del nostre codi més ràpid.
Una de les altres pràctiques que va ajudar-nos molt alhora de fer la poda als diferents algorismes, va ser posar un missatge cada cop que es trobes un nou best. D'aquesta forma vàrem poder veure de forma més entenedora com quan fèiem una combinació de greedy amb qualsevol dels altres dos algorismes, aconseguíem reduir considerablement el nombre de canvis que havia de fer el best.

Respecte al us dels datasets, en un principi vam fer us de els dataset++. Un cop vàrem veure el correctament amb aquest dataset, vam provar amb el dataset bàsic i també vam comprovar que funcionava correctament.
Quan no estàvem fent cap tipus de poda ambdós ens eren útils, ja que els ms que tardaven en solucionar els problemes eren suficients com per fer les gràfiques de la comparació d'algorismes, però quan vam tenir la poda funcionant correctament, tots els resultats eren entre 0 i 1ms, per tant va ser necessari, a partir del dataset++, modificar-lo afegint-li alguns nodes i servidors per fer que el temps de resolució fos més lent i així poder, apart de generar els gràfics, comprovar en un cas més complex el correcte funcionament del nostre programa. El mapa d'aquest nou dataset està inclòs a la memòria per si es vol fer tests amb ell.

6 Problemes Observats

Apart dels típics problemes de programació, com podrien ser el modificar fitxers referenciats, pensant que havíem creat un nou (sobretot amb arraylists) i altres problemes més petits, ens hem trobat amb uns quants que tenen més a veure amb el objectiu de la pràctica, el d'aprendre a fer un us correcte dels algorismes de cerca.
El primer amb el que vàrem trobar-nos va ser, que les solucions de el mode de fiabilitat del primer problema, sempre ens sortien maximitzats al màxim, es a dir que passava per tots els nodes.
El camí que l'algoritme realitzava era totalment correcte, però el problema era que passava per nodes que eren solució i no parava allà, és llavors quan vàrem adonar-nos que teníem que iterar per totes les solucions cada cop que miràvem si un node era solució, ja que sinó sempre tindríem el mateix problema alhora de maximitzar.

Vam tenir bastants problemes alhora d'adaptar el programa a que funciones de forma que un servidor pogués estar connectat a més d'un node, ja que el problema explicat anterior és un derivat d'haver-ho programat pensant que el dataset inicial, on només podia haver un servidor - node estava correcte. També vam tenir problemes amb el dataset++, però vam adonar-nos que els errors eren del json i vam arreglar-los fàcilment un cop detectat el problema.

De forma ja més teòrica vam tenir diversos dubtes respecte el plantejament de la segona funcionalitat del programa, ja que dubtàvem de quanta importància donar al aspecte de la localització. També ens va suposar un problema el determinar quin seria un marge correcte per la equitivitat dels servidors.
També ens ha sigut molt complicat saber com fer us de branch and bound per el 2n problema, ja que no sabíem com teníem que iterar amb un doble bucle, i dur a terme el expand, deque... correctament en aquesta situació. Vam tenir que realitzar una poda molt agressiva, ja que en les primeres iteracions de la implementació ens tardava uns 6 minuts al realitzar la prova amb branch. Amb diferents millores a la poda varem aconseguir rebaixar-ho fins als 55s. Posteriorment a la entrega i al examen, vam adonar-nos de que el principal problema era que Branch estava mal implementat ja de base (el expand sobretot), i que dins del while feiem un doble bucle de for, que ens relentitzava molt tot el programa. Un cop reescrit tot desde 0, el branch va funcionar a la perfecció, amb resultats de entre 30 i 180ms. La resta d'algorismes van ser relativament ràpids d'adaptar al 2n problema, sobretot greedy ja que és molt senzill.

7 Conclusions

Aquesta segona part de la pràctica, tot hi que ens ha semblat que era una carrega de treball bastant major a la primera, ha sigut una mica més entretinguda, ja que es tracta d'un concepte més interessant que els mètodes d'ordenació.
Apart de servir-nos molt per ajudar a consolidar els mètodes apresos a classe, ens ha semblat bastant entretingut el anar jugant amb diferents tipus de poda per aconseguir minimitzar els temps de solució del problema.
Ajuda també el fet de veure que amb un mateix algorisme pots solucionar problemes molt diferents, ja que t'obre els ulls en quant a les moltes funcions que podries arribar a donar-li en un futur, que no tinguin perquè ser relacionades amb trobar un camí o omplir una motxilla. En relació amb aquest fet, es interessant veure com per un problema o un altre els diferents algorismes es comporten millor o pitjor.

Un altre punt interessant, ha sigut veure com en alguns casos algun algorisme pot ser molt difícil, o més difícil que d'altres d'aplicar en determinats problemes o situacions, el qual ens acostuma a portar a fer una codificació no tant bona i acaba repercutint directament com hem pogut veure als gràfics.

8 Bibliografia

"0/1 Knapsack Using Branch and Bound." GeeksforGeeks, 20 Nov. 2018, www.geeksforgeeks.org/0-1-knapsack-using-branch-and-bound/.

"Backtracking Algorithms." GeeksforGeeks, www.geeksforgeeks.org/backtracking-algorithms/.

"Basics of Greedy Algorithms Tutorials Notes | Algorithms." HackerEarth Blog, www.hackerearth.com/practice/algorithms/greedy/basics-of-greedy-algorithms/tutorial/.
"Branch Bound." Mutuah, www.mutah.edu.jo/userhomepages/CS252/branchandbound.html.

"Greedy Algorithms." GeeksforGeeks, www.geeksforgeeks.org/greedy-algorithms/. Kalczynski, Pawel Jan.

"A Java Implementation of the Branch and Bound Algorithm: The Asymmetric Traveling Salesman Problem - Semantic Scholar." Undefined, 1 Jan. 1970, www.semanticscholar.org/paper/A-Java-Implementation-of-the-Branch-and-Bound-The-Kalczynski/545faa97bdf76932724e2dbff390f6266b3c0747.

"Recursion and Backtracking Tutorials Notes | Basic Programming." HackerEarth Blog, www.hackerearth.com/practice/basic-programming/recursion/recursion-and-backtracking/tutorial/.

"The Knapsack Problem - Yogendra Gadilkar." HackerEarth Blog, www.hackerearth.com/practice/notes/the-knapsack-problem/.

"Travelling Salesman Problem | Set 1 (Naive and Dynamic Programming)." GeeksforGeeks, 6 Sept. 2018, www.geeksforgeeks.org/travelling-salesman-problem-set-1/.

"Design and Analysis of Algorithms 0-1 Knapsack." Www.tutorialspoint.com, Tutorials Point, www.tutorialspoint.com/$design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_01_knapsack.htm$.