Universitat Politècnica de Catalunya

Intel·ligència Artificial

Projecte Laboratori

Cerca Local - Bicing

Autors:
Aleix Marro
Guillem Pla
Arnau Ruana

5 de novembre de 2019

Índex

1 Introducció

L'objectiu principal d'aquesta primera pràctica de laboratori és familiaritzar-se amb diferents tècniques de resolució de problemes basades en cerca local. Concretament veurem i utilitzarem les tècniques *Hill Climbing* i *Simulated Annealing*, dues tècniques d'optimització matemàtica molt populars i utilitzades en el món de la intel·ligència artificial.

Per fer-ho ens ajudarem de la llibreria de Java AIMA en la qual trobem les dues tècniques esmentades anteriorment ja programades. Com veurem més endavant, simplement li haurem de passar, a cada una d'elles, el nostre estat actual del problema, els seus successors, un comprovador d'estat final i l'heurístic a utilitzar per tal que ens retorni el nou estat que millor s'ajusta a la solució desitjada.

Després de raonar sobre els elements necessaris per a plantejar el problema, determinar quines formes existeixen per a la seva resolució i comparar-les, aquests han estat els nostres resultats.

2 Problema

Per començar vam analitzar l'enunciat del problema i identificar tots els seus components i restriccions. Havent fet això ens calia escollir una representació adequada i implementable en Java que representés la totalitat del problema i satisfés totes les restriccions trobades. A continuació exposem detalladament tots els passos seguits.

2.1 Descripció del problema

El problema consisteix en una ciutat virtual de $10 * 10$ kilòmetres amb un cert nombre d'estacions de Bicing disseminades aleatòriament per tota la ciutat i se'ns demana ajudar a Bicing a optimitzar la distribució de bicicletes en les seves estacions de manera que els usuaris puguin trobar bicicletes sempre que les necessitin.

Per a poder resoldre el problema, Bicing ens garanteix una certa informació a cada una de les estacions segons les estadístiques de la demanda i moviment de bicis en una hora determinada del dia. En concret disposa de la següent informació:

- La previsió del nombre de bicicletes que no seran usades en una estació durant una hora especificada i que, per tant, podrien ser mogudes a una altra estació.
- La previsió del nombre de bicicletes que hi haurà en una estació la pròxima hora. Només té en compte les bicis mogudes entre estacions per culpa dels usuaris, no té en compte les mogudes per altres mitjans.
- La previsió del nombre de bicicletes que hauria d'haver en una estació la pròxima hora per tal de cobrir la demanda prevista.

Per a trobar la distància entre dos punts qualsevol de la ciutat utilitzem la següent funció:

$$d(i,j) = |i_x - j_x| + |i_y - j_y| \tag{1}$$

Com hem dit anteriorment, Bicing té disseminades E estacions de bicicletes per tota la ciutat i, a la vegada, té B bicicletes en total que estan distribuïdes entre totes les estacions. Suposarem, per a aquest problema, que el nombre de bicicletes que hi caben dins d'una estació és il·limitat. A més, s'han enregistrat dos escenaris diferents de demanda:

- Demanda EQUILIBRADA: la demanda de bicicletes en cada estació és força similar durant certes hores del dia.
- Demanda HORA_PUNTA: la demanda de bicicletes en algunes estacions és molt superior a la demanda de les altres durant certes hores del dia.

Bicing planteja que resolguem el problema traslladant bicicletes d'una estació a una altra per tal d'aproximar-se a la demanda prevista d'una estació en una hora concreta. Disposem

d'una flota d'F furgonetes que ens permetran dur a terme el trasllat de bicicletes entre estacions. Cada una d'elles és capaç de transportar fins a un màxim de 30 bicicletes. A més, cada furgoneta en una mateixa hora només pot fer un únic viatge, transportant bicicletes des d'una estació a, com a màxim, dues estacions més i no hi pot haver dues furgonetes que agafin bicicletes de la mateixa estació. En el problema no caldrà usar les F furgonetes en una solució si no són necessàries.

L'acord amb Bicing consisteix en que ens pagui 1 € per cada bicicleta que transportem que faci que el nombre de bicis d'una estació s'apropi a la demanda. Per contra ens cobrarà 1 € per cada bicicleta transportada que allunyi una estació de la seva previsió de demanda. A més a més, el simple fet de transportar bicicletes dins les furgonetes també ens suposa un cost. Suposant que nb representa el nombre de bicicletes transportades, el cost en euros per kilòmetre recorregut és:

$$Cost = \frac{nb + 9}{10} \tag{2}$$

Tenint en compte tot l'anterior, volem obtenir la solució del problema per a una hora del dia concreta. La solució ha d'indicar per a cada furgoneta...

- l'estació d'origen,
- el destí o destins (dos com a màxim) que tindrà i l'ordre en el qual els recorrerà,
- el nombre de bicicletes transportades a cada estació destí.

2.2 Representació del problema

Quan es tracta de representar un problema per a resoldre'l amb algorismes de cerca local hem d'aconseguir que la representació sigui eficient, és a dir, que ocupi el menor espai en memòria possible. Per a fer això, només guardem informació estrictament necessària i, en cas que no es canviïn, guardem les variables de forma estàtica de manera que es comparteixin entre totes les instàncies d'una mateixa classe. Per a fer la representació del problema hem creat les classes *State* (secció 2.2.1) i *Van* (secció 2.2.2) que a continuació les expliquem detalladament.

2.2.1 Classe *State*

La classe *State* permet representar el problema en la seva totalitat. És com si fos un fotografia de l'estat en el qual ens trobem en cada moment, és a dir, un instant de temps en el qual les furgonetes estan distribuïdes d'una certa manera pel la ciutat (mapa).

Els atributs que formen la classe *State* són els següents:

- ***stations***: és un objecte de la classe *Estaciones* i és una representació de les estacions de Bicing.

- ***nest***, ***nbic***, ***seed***, ***nvan***: són paràmetres estàtics que s'utilitzen per a inicialitzar un nou objecte *Estaciones* en cas que sigui necessari fer una còpia.
- ***isVisited***: és un *ArrayList* de booleans que ens indica quines estacions han estat visitades, d'aquesta manera podem complir la restricció de no agafar bicicletes de més d'una estació.
- ***fleet***: és un *ArrayList* de *Van* (secció 2.2.2) que ens serveix per a representar la flota de furgonetes de la qual disposem per a moure les bicicletes entre les diferents estacions.
- ***cost***: enter que representa el cost de transportar les bicicletes.
- ***demandSupplied***: enter que representa les bicicletes que hem transportat i en han reportat un benefici.
- ***benefits***: enter que representa la diferència entre el benefici obtingut en proporcionar bicicletes i el cost.

Els enters *cost*, *demandSupplied* i *benefits* s'actualitzen cada cop que apliquem un operador (secció 4). Aquests ens serveixen per a que els heurístics (secció 6) retornin el resultat correcte.

2.2.2 Classe *Van*

La classe *Van* representa una furgoneta de transport de bicicletes i consta dels següents atributs:

- ***stationID***: enter que representa l'identificador de l'estació d'origen on ha estat generada la furgoneta i d'on ha agafat ja, si ha pogut, les bicicletes necessàries.
- ***CAPACITY***: enter estàtic que indica la capacitat màxima d'una furgoneta.

3 Generació d'una solució inicial

L'origen de les furgonetes no està determinat inicialment, per tant, la generació d'un estat inicial és necessària per a començar la cerca local. Per a generar aquest estat hem d'assignar a cada furgoneta disponible en la nostra flota una estació.

En aquest treball hem assumit que una furgoneta no pot inicialitzar-se en un punt qualsevol de la ciutat, per tant, les inicialitzem sempre al costat d'una estació denominada l'estació d'origen d'una furgoneta. A continuació mostrem les diferents estratègies que hem implementat.

3.1 Solució inicial aleatòria

Aquesta solució inicial es basa en inicialitzar cada furgoneta en una estació escollida de forma aleatòria. D'aquesta forma les furgonetes poden estar en estacions on no hi hagi excés de bicicletes. També pot ocórrer que, per mala sort, s'assigni més d'una furgoneta a una estació, el que implicaria que, un cop una furgoneta hagi carregat bicicletes, les demés passessin a estar inutilitzades.

Per culpa de tot això, creiem que aquesta solució no ens donarà els millors resultats degut a que algunes furgonetes no es mouran. De totes formes hem cregut interessant veure com es comporten els nostres experiments (secció 7) amb aquesta solució.

3.2 Solució inicial fixa

A diferència de l'estat anterior (secció 3.1), en aquest no es genera res aleatòriament. En aquest cas assignem les furgonetes a les estacions que tenen més excés de bicicletes. Això es fa per ordre d'excés, és a dir, cada furgoneta generada s'assignarà a l'estació amb el màxim excés d'entre les estacions que no han estat prèviament assignades.

Ens ha semblat una solució molt adient ja que les furgonetes es podran carregar amb el nombre màxim de bicicletes i per tant maximitzar el seu trasllat.

3.3 Solució inicial combinada

En aquest últim cas utilitzem una combinació de les dues tècniques esmentades amb anterioritat (seccions 3.1 i 3.2 respectivament). Aquesta combinació consisteix en generar les furgonetes en una estació escollida aleatòriament, però aquest cop, entre les estacions que tenen un excés de bicicletes. És una solució molt semblant a l'anterior, ja que maximitzem el trasllat de bicicletes, però aquest cop les furgonetes no cal que es generin en estacions que tenen un excés màxim. D'aquesta manera podem aconseguir una reducció del cost de transport per què les estacions amb excés màxim poden estar molt lluny de les que tenen un dèficit de bicicletes.

4 Operadors

En aquesta secció ens disposem a detallar els operadors plantejats durant la realització de la pràctica i els experiments (secció 7) duts a terme per decidir quins i com els hem utilitzat.

4.1 Descripció d'operadors

En un principi vam pensar la possibilitat d'utilitzar diferents operadors simples per representar els conceptes de carregar i descarregar una bicicleta i moure una furgoneta d'una estació a una altra. No obstant, ens vam adonar que el fet de carregar una bici, moure una furgoneta i, finalment, descarregar una bici és un procediment seqüencial que sempre s'efectua en la seva totalitat sense admetre variacions (almenys en qualsevol desplaçament de bicis productiu). Així doncs vam decidir que un operador realitzés tot aquest procediment de forma atòmica.

Aquesta decisió comportava l'existència d'un segon operador per tal de permetre la segona descàrrega en un destí alternatiu i per això vam decidir crear un operador que es comportés de la mateixa forma que el primer, però descarregant bicis a dos destins diferents.

Amb tot això esmentat, ens vam decantar pels operadors següents:

- ***singleMove***: permet carregar bicicletes en una estació d'origen i descarregar-les en la seva totalitat en una única estació destí.
- ***doubleMove***: permet carregar bicicletes en una estació d'origen i descarregar-les en la seva totalitat en dos estacions de destí diferents. L'aplicació d'aquest operador implica la satisfacció de les restriccions següents:
  1. Es descarregarà com a mínim una bicicleta en cada estació.
  2. La demanda a suplir de la primera estació quedarà en qualsevol cas satisfeta després de descarregar-hi les bicis.

4.2 Conjunts d'operadors

Utilitzant els dos operadors escollits podíem accedir a tot l'espai d'estats accessibles amb cerca local. Malgrat això, ens oferien dos possibles conjunts d'operadors alternatius fent ús, en cada cas, d'un únic operador. El conjunt que, aparentment, semblava ser més adequat era el primer (secció 4.2.1) ja que, utilitzant un sol operador, no generàvem tot l'espai d'estats accessibles. No obstant, ens calia verificar la nostra hipòtesi i comprovar també els resultats obtinguts amb el segon operador (secció 4.2.2).

4.2.1 Conjunt *singleMove*

En aquest cas l'operador aplicat és sempre el mateix: *singleMove*, el qual ens permet carregar bicis en una estació i descarregar-les directament en una altra. A priori, es pot preveure que aquest conjunt aporta un bon rendiment amb demandes a suplir força grans i que anirà disminuint segons les demandes esdevinguin més petites i nombroses. Fem aquesta hipòtesi

ja que, per aprofitar al màxim la nostra capacitat de càrrega, independentment del conjunt d'operadors escollit, només podrem descarregar en un destí per furgoneta. Caldrà però, confirmar o desmentir aquesta hipòtesis en la fase d'experimentació (secció 7).

4.2.2 Conjunt *doubleMove*

En aquest cas l'operador aplicat és també sempre el mateix: *doubleMove*, el qual ens permet carregar bicis en una estació i descarregar-les de forma seqüencial en dos destins diferents. Cal matisar que en ambdues estacions de destí es deixarà com a mínim una bicicleta i la demanda a suplir del primer destí quedarà sempre coberta completament. D'aquestes restriccions podem sintetitzar que en cap cas la demanda a suplir de la primera estació de destí serà superior a la capacitat d'una furgoneta o al nombre de bicis excedents i no usades de l'estació d'origen. Aquesta restricció ens resultarà indispensable a l'hora d'implementar els operadors per evitar tenir estats incorrectes.

A priori es pot preveure que aquest conjunt aporta un bon rendiment amb grans nombres de demandes reduïdes i que aquest anirà disminuint segons les demandes esdevinguin més grans i menys nombroses. Fem aquesta hipòtesi ja que, per a poder repartir a més d'un destí, la demanda del primer haurà de ser sempre inferior al nombre de bicis carregades a l'origen i estarem obligats a descarregar dos cops. Malgrat això, caldrà confirmar o desmentir la hipòtesis en la fase d'experimentació (secció 7) anàlogament a cóm ho farem per l'operador *singleMove* (secció 4.2.1).

4.2.3 Conjunt *singleMove* + *doubleMove*

Per últim tenim la possibilitat d'aplicar en cada cas tots dos operadors i veure quin ens ofereix millors resultats. Això ens permet cobrir totes les possibilitats ja que, indistintament, podem descarregar en una o dues estacions segons les necessitats de l'estat actual. Degut a això, podem preveure que aquesta opció ens aporta un bon rendiment en qualsevol cas, però sobretot, en demandes equilibrades. Fem aquesta hipòtesi ja que, en aquesta opció, podem cobrir les mateixes possibilitats que amb els dos conjunts anteriors (seccions 4.2.1 i 4.2.2 respectivament) i, a més a més, aquelles possibilitats on una combinació sigui més adequada.

Havent fet aquestes apreciacions, cal esmentar que no ens hem decantat directament per aquest conjunt ja que, tot i que cobreix més possibilitats, això té una repercussió notable en els temps d'execució i recursos utilitzats. Això es deu a que el nombre de possibilitats a explorar en cada cas serà molt superior. Per tant caldrà veure fins a quin punt la hipotètica millora és real i si justifica l'increment espacial i temporal en recursos.

5 Generació d'estats successors

Després d'haver escollit els operadors pertinents (secció 4) ja podíem implementar les funcions encarregades de generar els estats successors, és a dir, els possibles moviments que es poden realitzar partint d'un estat qualsevol.

A continuació veurem cóm vam aplicar els operadors de dues formes diferents segons es demanava en l'enunciat: exhaustivament per l'algorisme de cerca *Hill Climbing* i aleatòriament pel *Simulated Annealing*.

5.1 Funció generadora pel *Hill Climbing*

Per utilitzar l'algorisme de cerca local *Hill Climbing* (explicat breument en la secció 1) es demanava que s'apliquessin en cada cas tots els operadors del conjunt d'operadors escollit amb tots els possibles valors per a aquests. Com teníem tres conjunts d'operadors que volíem avaluar, vam haver d'implementar tres classes diferents per generar successors amb aquest mètode:

- ***SuccessorSingle***: aplicant el conjunt amb *singleMove* (secció 4.2.1) com a únic element.
- ***SuccessorDouble***: aplicant el conjunt amb *doubleMove* (secció 4.2.2) com a únic element.
- ***SuccessorBoth***: aplicant el conjunt amb *singleMove* i *doubleMove* com a operadors.

En qualsevol cas el conjunt només determina els operadors aplicats, els quals sempre aplicarem amb tots els valors possibles. Per exemple: per cada furgoneta, si apliquem el conjunt amb l'operador *singleMove* tenim un estat successor per a cada possible destí vàlid de la furgoneta. Podríem, en realitat, generar per cada destí vàlid, tants estats com bicis fos possible transportar-hi en un sol viatge o tantes com necessites (el mínim d'aquests dos valors). No obstant, com estem utilitzant cerca local, sempre transportarem el màxim de bicicletes que ens sigui possible, per això limitem la generació a un únic estat per destí vàlid.

El mateix succeeix per a la resta dels conjunts d'operadors: *SucessorDouble* i *SuccessorBoth*.

5.2 Funció generadora pel *Simulated Annealing*

Per utilitzar l'algorisme de cerca local *Simulated Annealing* (explicat breument en la secció 1) es demanava que s'apliques un sol operador escollit aleatòriament. Per obtenir un estat accessible i vàlid vam haver d'acotar aquests valors aleatoris aplicant les restriccions següents:

- En cap cas cap destí pot ser el mateix que l'origen. Si en generar els valors aleatòriament es produeix aquest fet es tornen a generar valors fins que no es produeixi.

- El nombre de bicicletes transportades satisfarà les restriccions d'operador. És a dir mai serà superior al nombre de bicis excedents i disponibles, a la capacitat d'una furgoneta o a la demanda conjunta dels destins. També serà superior a la demanda del primer destí en el cas que s'apliqui un operador *doubleMove*.

El nombre de bicicletes a desplaçar amb *Hill Climbing* ha estat sempre calculat amb la funció *calculateNumBikes* quan apliquem l'operador *singleMove* (secció 4.2.1) i amb la funció *calculateNumBikesDouble* quan apliquem l'operador *doubleMove* (secció 4.2.2).

En cas que estem utilitzant *Simulated Annealing* els resultats d'aquestes funcions seran les fites superiors dels nombres generats aleatòriament.

6 Heurística

En la resolució d'un problema de cerca local és imprescindible tenir una funció heurística per tal que representi les relacions de qualitat entre les diferents solucions. Aquesta ha d'aproximar la qualitat de la solució i, per a fer-ho, ha d'avaluar els diferents elements d'un estat i maximitzar-los o minimitzar-los segons es desitgi.

Per a trobar la distància entre estacions i el cost de transport de bicicletes, necessaris pels heurístics, utilitzem pertinentment les dues equacions definides anteriorment (equacions 1 i 2 respectivament).

En aquest cas hem creat dues funcions heurístiques que utilitzen criteris diferents per a avaluar les solucions. D'aquesta manera podrem comparar i veure quin dels dos ens interessa més per a implementar la solució al nostre problema.

6.1 Heurística *MaxObtained*

En aquesta primera funció heurística només tenim en compte la quantitat de diners que Bicing ens paga per a cada bicicleta transportada correctament, suposant que el cost de transportar-les és negligible. Per a implementar aquesta avaluació incrementem el nombre total de bicicletes mogudes correctament a cada estat fins a arribar al total.

6.2 Heurística *MinCost*

En aquest segon heurístic, a part del que ja teníem en compte en l'anterior heurístic (secció 6.1), també tenim en compte el cost de transportar les bicicletes. En aquest cas el resultat de l'avaluació serà la diferència entre els diners obtinguts per moure les bicicletes correctament i el cost de transportar-les. D'aquesta manera es penalitza negativament el fet de transportar bicicletes entre dues estacions molt llunyanes entre sí.

7 Experimentació

En aquesta secció explicarem detalladament cóm hem realitzat cada un dels experiments per tal de poder arribar a una conclusió sobre quin operador, estat inicial i algorisme de cerca local són més adients per a la resolució del problema (secció 2).

7.1 Conjunt d'operadors

El primer pas per escollir una solució final és escollir el conjunt d'operadors més adequat d'entre les opcions plantejades (secció 4). Per veure quin d'aquests oferia millors resultats vam realitzar una execució seguint les indicacions de l'enunciat amb 25 estacions, 1250 bicis i 5 furgonetes i una demanda equilibrada.

Com a estratègia per a generar l'estat inicial vam utilitzar la que establia les furgonetes en estacions amb les demandes més altes (secció 3.3), ja que en les execucions inicials ja semblava oferir millors resultats. Pel que fa a l'heurística vam utilitzar la que optimitzava el primer criteri (secció 6.1) tal com es demanava.

Amb aquestes dades vam executar el programa 10 cops canviant la llavor en cada execució. Els valors dels heurístics i temps d'execució obtinguts per a cada un dels operadors van ser els següents:

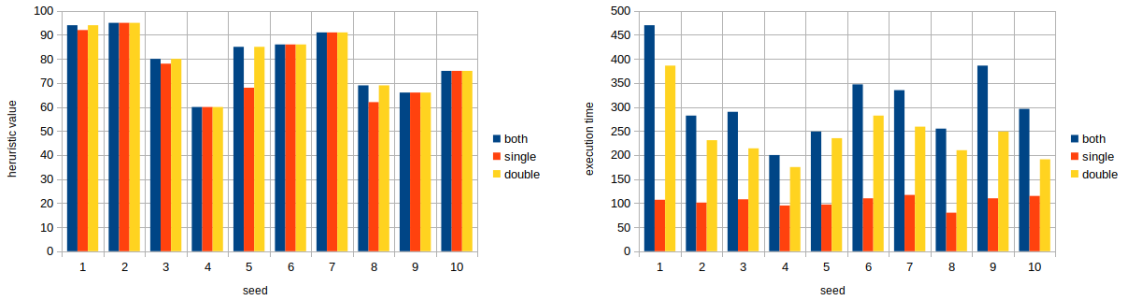


Figura 1: Valor heurístic i temps d'execució dels operadors amb demanda equilibrada.

Amb aquests primers resultats semblava que el conjunt amb només l'operador *doubleMove* (secció 4.2.2) era la millor opció ja que donava els mateixos resultats que el conjunt amb ambdós operadors però amb un temps menor. No obstant aquests resultats també confirmaven parcialment la nostra hipòtesi inicial de que *doubleMove* oferiria un millor rendiment en demandes múltiples i similars. És per això que abans de decidir-nos per aquest operador vam voler veure com es comportaven els operadors en un experiment amb les mateixes dades per a una demanda d'hora punta.

D'igual forma que en l'experiment anterior vam executar el programa 10 cops canviant la llavor en cada execució. Els valors dels heurístics i temps d'execució obtinguts en aquest cas van ser els següents:

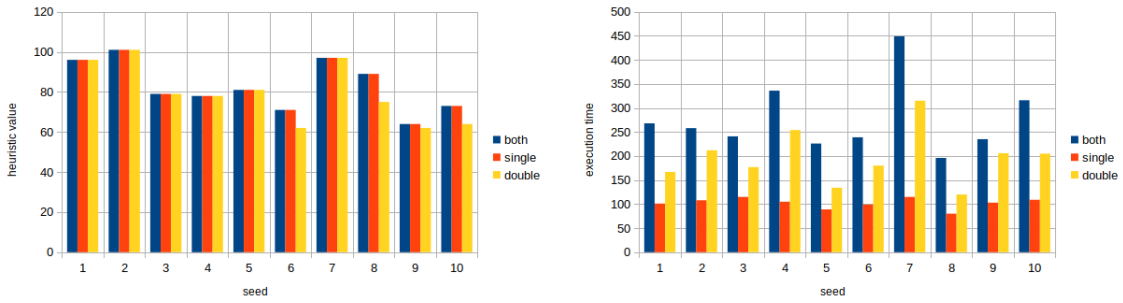


Figura 2: Valor heurístic i temps d'execució dels operadors amb demanda no equilibrada.

Després d'aquest segon experiment ja no podíem decantar-nos pel conjunt amb només l'operador *doubleMove*. En aquest cas el millor rendiment l'aportava el conjunt *singleMove* (secció 4.2.1) que en el primer experiment havia tingut el pitjor rendiment. Aquest fet confirmava la segona part de la nostra hipòtesi i ens obligava a escollir el conjunt amb ambdós operadors ja que oferia un rendiment màxim independentment del tipus de demanda.

Sí que és cert però, que aquesta versatilitat suposava un increment notable del temps d'execució, no obstant no ens semblava una raó de pes ja que estàvem treballant amb temps d'execució molt reduïts.

7.2 Estat inicial

Un cop tenim el conjunt d'operadors que millor s'ajusta al nostre problema (secció 7.1) hem realitzat el segon experiment. Aquest consisteix en decidir quin dels estats inicials plantejats anteriorment (secció 3) és més adient i aconsegueix una millor solució pel problema.

Per poder decidir quin estat inicial és millor executem els següents experiments i recopilem els valors de l'heurístic i el temps d'execució de cada un. A continuació mostrem les gràfiques obtingudes:

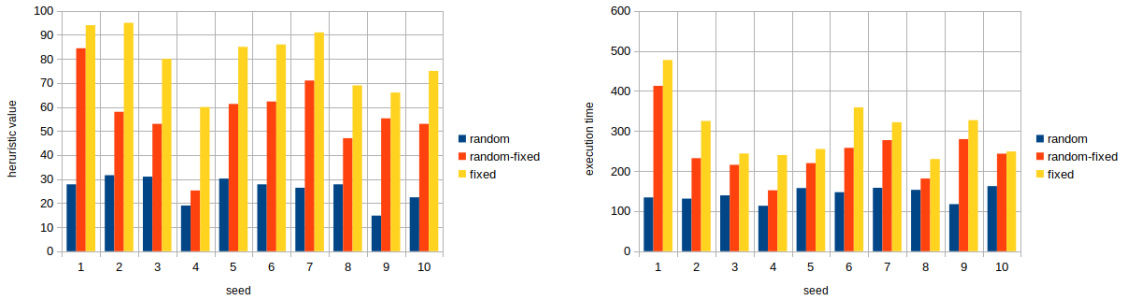


Figura 3: Valor heurístic i temps d'execució dels estats inicials.

Com era d'esperar, l'estat inicial *random* (secció 3.1) és el que obté els pitjors resultats. Això és degut a que les furgonetes poden estar inicialitzades en estacions on no hi hagi

excedents de bicicletes i, per tant, no es moguin i no generin beneficis.

Per altra banda tenim l'estat inicial *random-fixed* (secció 3.3) que se situa en segona posició. En aquest cas les furgonetes només es poden assignar a les estacions que tenen excedents malgrat aquesta assignació es realitza aleatòriament. Té sentit que aquest estat inicial produeixi valors d'heurístic força superiors als que produeix el *random* en la majoria dels casos.

Finalment tenim l'estat inicial *fixed* (secció 3.2) que presenta els millors resultats tal i com ja havíem predit. És evident que el fet d'inicialitzar les furgonetes per ordre d'excedents en les estacions és una molt bona aproximació que permet obtenir un marge de beneficis més elevat que en els altres dos casos.

7.3 *Simulated Annealing*

Per a aconseguir que l'algorisme *Simulated Annealing* ens reporti uns bons resultats necessitem assignar un conjunt de paràmetres, aquests són:

- Nombre total d'iteracions.
- Iteracions per cada canvi de temperatura.
- Paràmetre k de la funció d'acceptació d'estats.
- Paràmetre λ de la funció d'acceptació d'estats.

Inicialitzar aquests paràmetres no és trivial, és per això que necessitem experimentar amb un conjunt de paràmetres per a veure en quins obtenim els millors resultats. En el nostre cas ens hem adonat que moltes de les combinacions possibles donaven uns resultats molt pobres, i per aquest motiu no els mostrem a la següent gràfica.

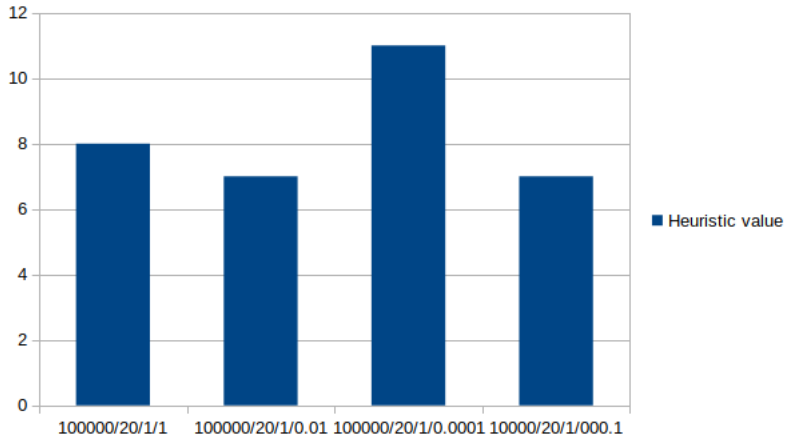


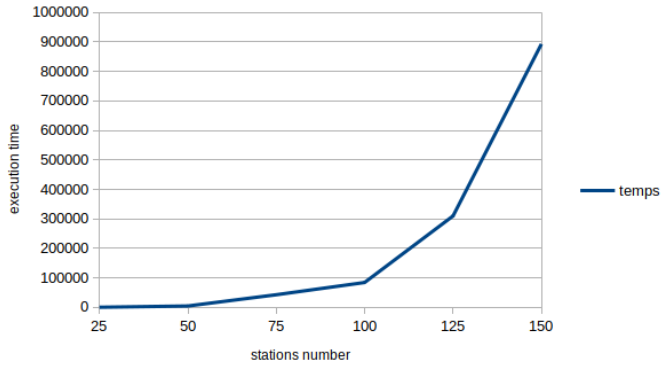
Figura 4: Valor de l'heurístic per a diferents paràmetres de l'algorisme.

Dels resultats obtinguts hem decidit utilitzar 100000/20/1/0.0001 ja que, com podem veure a l'anterior gràfica són els que millor resultat ens ha donat.

7.4 Temps d'execució amb *Hill Climbing*

En aquest quart experiment havíem de computar empíricament el temps d'execució del nostre programa augmentant en cada iteració el nombre de furgonetes, estacions i bicicletes per tal d'obtenir una gràfica en la qual quedi representat en quin moment el programa deixa de ser computable degut als elevats costos de processament que implica.

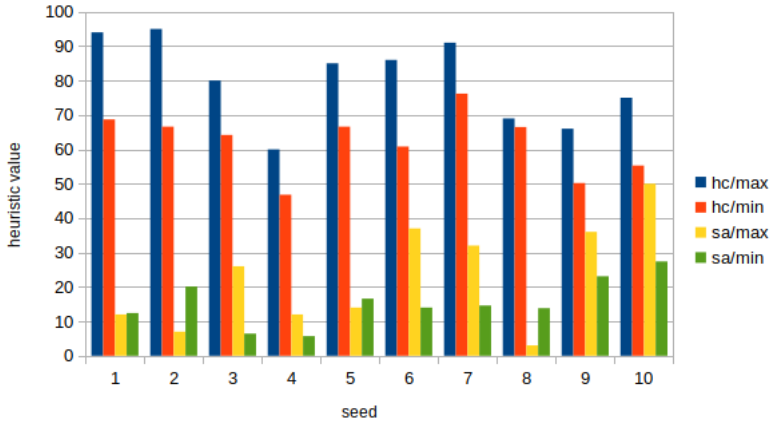
És evident que hem executat el nostre programa amb l'operador i estat inicials amb els quals hem obtingut millor resultats anteriorment (seccions 7.1 i 7.2 respectivament).



En la gràfica podem observar com el temps d'execució es dispara a partir de les 115 estacions i esdevé poc rentable d'executar a partir de les 125.

7.5 Diferències entre algorismes de cerca local

En aquest experiment l'objectiu principal era observar les diferències en el comportament i resultats del programa en funció de l'algorisme de cerca utilitzat i també de l'heurístic utilitzat. Com ja disposàvem dels resultats per *Hill Climbing* amb l'heurística d'optimització pel primer criteri ens restava experimentar amb l'heurística pel segon criteri i l'algorisme *Simulated Annealing*. Hem sintetitzat els resultats obtinguts en la següent gràfica:



Com es pot observar en el gràfic el rendiment dels experiments efectuats amb l'algorisme de cerca *Hill Climbing* han estat molt superiors que els dels realitzats amb *Simulated Annealing*. No obstant a causa dels problemes que hem tingut amb la generació de successors aleatòria no hem pogut experimentar exhaustivament amb el *Simulated Annealing* per determinar quins valors resultaven amb un rendiment màxim.

A les conclusions (secció 8) explicarem més detalladament els inconvenients que això ens ha comportat i les mesures adoptades.

7.6 Tipus de demanda

El sisè experiment demanat consistia en repetir el primer experiment realitzat (secció 7.1) però aquest cop amb una demanda d'hora punta. Recordem, que fent un anàlisis més exhaustiu per decidir el conjunt d'operadors ja havíem realitzat aquest experiment i en conseqüència disposàvem de les dades. Així doncs observem a continuació els resultats amb demanda equilibrada a l'esquerra, i demanda d'hora punta a la dreta:

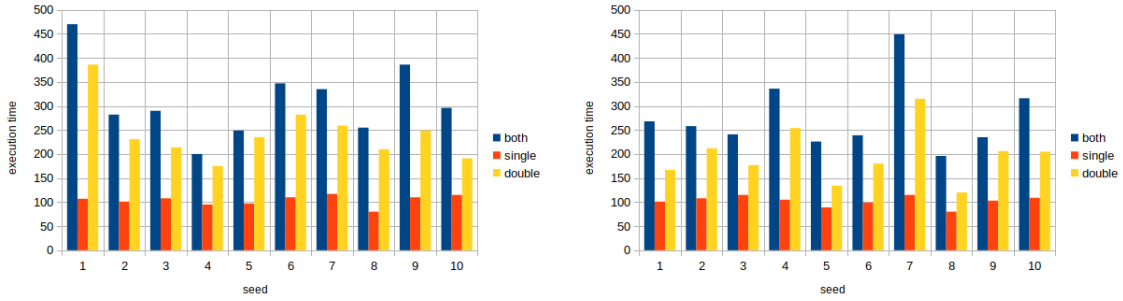


Figura 5: Temps d'execució dels operadors per les dues opcions de demanda

Com podem observar el fet de tenir una demanda d'hora punta fa que el temps d'execució amb el conjunt *doubleMove* augmenti lleugerament i en conseqüència també ho faci el conjunt d'ambdós operadors ja que inclou aquest primer. Aquest augment és significatiu per *doubleMove* ja que en augmentar el nombre de demandes l'augment del nombre d'estats possibles es quadràtic, tot i no generar-los tots, cada possible primer destí té un estat addicional per cada estació restant amb demanda, exceptuant la d'origen. En canvi pel *simpleMove* aquest increment es lineal.

7.7 Furgonetes òptimes

El darrer experiment que es demanava consistia en repetir l'execució del primer apartat pels dos tipus de demanda augmentant progressivament el nombre de furgonetes fins a trobar el benefici màxim.

Podem observar els resultats a continuació, amb demanda equilibrada a l'esquerra i amb demanda d'hora punta a la dreta:

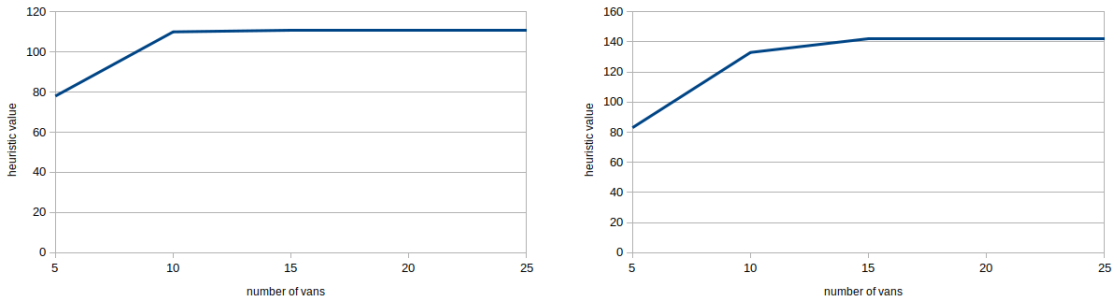


Figura 6: Valor de l'heurístic segons els nombre de furgonetes per a les dues demandes.

Com era d'esperar el nombre de furgonetes necessàries per aconseguir el millor rendiment és més gran per demandes d'hora punta. Un altre fet ressenyable que s'observa en el grafs és que amb el mateix nombre de furgonetes aquestes s'utilitzen més eficientment en demandes d'hora punta.

8 Conclusions

Com a conclusió podem dir que s'han complert bastant les nostres prediccions sobre quin estat inicial, operador o heurístic oferiria millors resultats abans, fins i tot, de realitzar els experiments. No obstant això, sabem que és molt important en un problema de cerca local provar-ho empíricament per descartar possibles errors en les nostres prediccions.

Amb l'estudi i l'experimentació finalitzats hem cregut oportú remarcar els següents punts:

- Ha quedat pendent d'implementar l'optimització per visitar aquelles estacions amb una demanda positiva, tot i que ja no hi generem un successor encara hi accedim.
- Optimització d'estat inicial tenint en compte la distancia a les estacions amb més demanda.
- La generació d'estats successors amb operadors aleatoris i valors aleatoris ens ha resultat molt problemàtica ja que els operadors funcionaven inicialment només per generar estats que satisfessin totes les restriccions i aportessin alguna millora. És per això que inicialment vam fer una generació aleatòria molt acotada que satisfés les necessitats dels operadors. Finalment però, vam concloure que aquesta no aportaria mai millors resultats que el *Hill Climbing* i en tot cas ens donaria una solució pitjor. Tot i haver implementat novament aquesta funció i adaptat la resta de classes perquè aquesta funcioni correctament, hem adjuntat al treball la primera versió a mode d'annex per a possibles simulacions. Els fitxers en qüestió són *SuccessorRandomAcotat.java* i *StateRandom.java* (amb els operadors originals que funcionen correctament per *Hill Climbing*) i es poden trobar a la següent ruta: *src/domain*.
- Com també s'ha esmentat en l'apartat pertinent a causa dels problemes que hem tingut amb la generació aleatòria no hem disposat del temps suficient per experimentar exhaustivament amb els valors del *Simulated Annealing*.