Mínim Conjunt Dominador d'Influència Positiva

Projecte d'Algorismia

Bachelor of Technology
in
COMPUTER SCIENCE AND ENGINEERING

Pau Adell Raventós
Ignasi Fibla Figuerola
Jaya García Fernández
Albert Tarrés Ribalta

DEPARTMENT OF COMPUTER SCIENCE
Universitat Politècnica de Catalunya
Setembre, 2021

Índex

1 Introducció **1**

2 Minimal Positive Influence Dominating Set **2**
2.1 Algorisme per la propietat PIDS 3
2.2 Algorisme per la propietat PIDS minimal 4

3 Aproximació "Greedy" **6**
3.1 Algorisme 6
3.2 Experimentació 8

4 Cerca Local **9**
4.1 Cerca Local Determinista 10
4.2 Algorisme de Hill-Climbing 11
4.3 Proves i Dissenys anteriors 13
4.4 Experimentació 13

5 Metaheurístiques **15**
5.1 Algorisme de GRASP 16
5.2 Experimentació 18

6 Programació Lineal **19**
6.1 Programació Lineal Entera 19
6.2 MPIDS utilitzant ILP 20
6.3 Introducció a CPLEX 20
6.4 MPIDS utilitzant CPLEX 21

7 Conclusions **24**

1 Introducció

Avui en dia, la millor manera de compartir informació és a través de les xarxes socials, tan per donar a conèixer fets com per expressar idees o opinions sobre qualssevol tema. En general, l'ús d'aquestes xarxes és molt útil doncs ens permet estar més atents a fets que puguin passar arreu del món. El problema apareix en el fet que la gent pot escriure el que vulgui, podent manipular la informació o presentant les seves creences sobre certs fets, i això pot influenciar molt el nostre criteri sense que ens adonem.

Per poder entendre la influència en aquest context, va aparèixer el problema del "*conjunt que domina la influència positiva*" del qual parlarem en aquesta pràctica, que intenta abordar com els comentaris de certs usuaris poden influir en d'altres. En concret, parlarem del problema de trobar el conjunt mínim que domina la influència positiva.

Per estudiar aquest problema i les diferents solucions que té, hem implementat diverses tècniques per trobar el Mínim Conjunt Dominador d'Influència Positiva dins d'un graf, que pot representar uns usuaris (nodes) i unes relacions entre aquests usuaris (arestes). Les tècniques empleades han sigut una tècnica voraç, una cerca local basada en Hill-Climbing, una metaheurística i una tècnica basada en programació lineal entera.

En els següents capítols veurem en detall les implementacions de cada tècnica, les diferents experimentacions que hem fet i les comparatives corresponents. La idea d'aquesta pràctica és la de buscar millorar tècniques ja existents per obtenir valors més petits, donat el mateix *benchmark*, de la mida del conjunt Dominador per un cert graf.

Cal destacar que aquest problema és NP-Hard, implicant que és com a mínim tan difícil de resoldre com un problema NP. Per aquesta raó, cada vegada que ens referim a obtenir la solució mínima, realment al que ens referirem és a obtenir la millor aproximació a la solució mínima que hem pogut trobar.

Per abreviar, en el document usarem els acrònims PIDS (Positive Influence Dominating Set) i MPIDS (Minimum Positive Influence Dominating Set) en anglès, per referir-nos al Conjunt Dominador d'Influència Positiva i al Mínim Conjunt Dominador d'Influència Positiva, respectivament.

2 Minimal Positive Influence Dominating Set

Abans d'entrar en detall sobre les diferents tècniques, parlarem de què són els PIDS i els MPIDS, en el context de la informàtica, i de com es pot trobar per un subconjunt donat de vèrtexs d'un graf, si aquest subconjunt és PIDS o no.

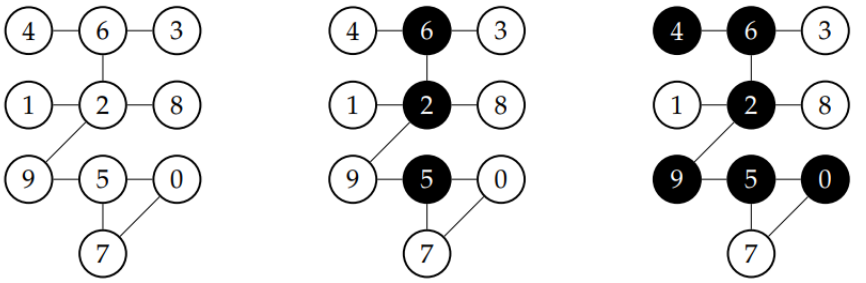
Donat un Graf $G = (V, E)$ i un subconjunt $D \subseteq V$, un Conjunt Dominador d'Influència Positiu, és aquell que compleix l'equació (1)

$$\forall v \in V : \left\lceil \frac{grau(v)}{2} \right\rceil - |N_D(v)| \leq 0 \tag{1}$$

on $N(v)$ és el conjunt de veïns de v que pertanyen al conjunt D.

De manera més informal, un subconjunt D és PIDS quan tots els vèrtexs tenen com a mínim la meitat de les seves adjacències a vèrtexs que pertanyen al conjunt D. Quan un vèrtex compleix aquesta propietat es diu que està *cobert*. Un MPIDS doncs, serà aquell conjunt D' que té la mínima cardinalitat sense perdre la propietat de ser PIDS.

Podem veure en la Figura 1 un exemple d'un graf amb un conjunt d'influència positiu, on es veu la definició més gràficament.



(a) A simple, connected undirected graph

(b) A dominating set

(c) A positive influence dominating set

Figura 1: Exemples de (a) Graf G connex, simple i no dirigit; (b) Conjunt $D = \{2, 5, 6\}$ que és dominador a G, però no dominador d'influència positiva; (c) Conjunt $D = \{0, 2, 4, 5, 6, 9\}$ que és dominador d'influència positiva a G i a més és mínim (Exemple extret de [1])

Ara que ja hem explicat que són els Conjunts Dominadors d'Influència Positius, passarem a parlar de l'algorisme desenvolupat per trobar si, per un graf $G = (V, E)$ i un subconjunt $D \subseteq V$ donats, el subconjunt D és PIDS i en cas que ho sigui si és minimal.

Abans d'entrar a parlar dels algorismes, en la taula 1 hi tenim la mida dels grafs $G = (V, E)$ donats en el *benchmark* en nombre de vèrtexs, $n = |V|$, i en nombre d'areste,s $m = |E|$.

Taula 1: Taula dels valors de n i m per cada graf d'entrada

Benchmark		
Entrada	$\boldsymbol{n}$	$\boldsymbol{m}$
ego-facebook	4039	88234
graph_actors_dat	10042	145682
graph_CA-AstroPh	18772	198050
graph_CA-CondMat	23133	93439
graph_CA-HepPh	12008	118489
graph_football	115	613
graph_jazz	198	2742
soc-gplus	23628	39194
soc-fb-Brandeis99	3898	137567
socfb-Mich67	3748	81903

2.1 Algorisme per la propietat PIDS

L'algorisme rep per entrada un Graf $G = (V, E)$, representat per llistes d'adjacència, i un subconjunt $D \subseteq V$, i retorna si el conjunt és PIDS o no i, en cas que ho sigui, si també és minimal. A continuació tenim el pseudo-codi.

Algorithm 1 Algorisme de comprovació de PIDS

Input: Graf $G = (V, E)$ i un subconjunt $D \subseteq V$
Output: cert o fals

```
1:  S ← ∅
2:  Apilar 0 a S
3:  while S ≠ ∅ do
4:      v ← S.top()
5:      S.pop()
6:      nAdjacencies := 0
7:      for all (v, u) ∈ E do
8:          if u ∈ D then
9:              nAdjacencies := nAdjacencies +1
10:         end if
11:     end for
12:     if nAdjacencies / |N(v)| > 0.5 then
13:         return false
14:     end if
15: end while
16: return true
```

A mode de resum, el codi fa un recorregut en profunditat (DFS) per recórrer totes les adjacències de cada vèrtex i mirar per cada un d'ells si està cobert o no. Si no ho està, l'execució acaba i es retorna fals, i en cas contrari, es segueixen comprovant tots els vèrtexs fins ja els haguem vist tots i retornem cert.

Correctesa La correctesa de l'algorisme prové de comprovar per cada vèrtex si aquest està cobert o no. Si algun no ho compleix, l'execució acaba i es retorna fals. Si l'execució retorna cert, sabem segur que tots els vèrtexs estan coberts i que l'execució s'ha fet fins al final.

Cost Com dins de cada bucle el cost de cada operació és de $O(1)$, el cost d'aquest algorisme serà el mateix que el d'una cerca en profunditat: $O(n+m)$, on $n = |V|$ i $m = |E|$.

2.2 Algorisme per la propietat PIDS minimal

Aquest algorisme, rep per entrada un graf $G = (V, E)$ i un subconjunt $D \subseteq V$ que sabem a priori que és PIDS de G. Abans de veure el pseudo-codi, cal aclarir que quan usem la variable isMPIDS, ens referim a si és Minimal no Mínim:

Algorithm 2 Algorisme de comprovació de PIDS minimal

Input: Graf $G = (V, E)$ i un subconjunt Dominador d'Influència Positiu $D \subseteq V$
Output: cert o fals

```
1:  for all v ∈ D do
2:      for all (v, u) ∈ E do
3:          nAdjacencies := 0
4:          for all (u, w) ∈ E do
5:              if w ∈ D ∧ w ≠ v then
6:                  nAdjacencies := nAdjacencies +1
7:              end if
8:          end for
9:          if nAdjacencies / |N(v)| < 0.5 then
10:             return true
11:         end if
12:     end for
13: end for
14: return false
```

Encara que el psuedo-codi sembli complicat, l'algorisme en sí segueix una idea molt simple. La propietat de minimalitat ens exigeix que si traiem un element del conjunt, la propietat de PIDS es perdrà. Així doncs, sabem que si traiem qualsevol vèrtex i es perd la propietat, el conjunt D segur que és minimal. Per guanyar en eficiència temporal, en comptes d'eliminar el vèrtex del conjunt D el que fem és simular que no hi és (línia 5 del codi).

Correctesa La correctesa es basa en la propietat explicada anteriorment. Com cada vegada que simulem que traiem un dels vèrtexs tornem a comprovar si el conjunt segueix sent PIDS, si el conjunt és minimal, per força en algun moment de l'execució el conjunt deixarà de ser PIDS, per algun cert vèrtex "eliminat", i s'acabarà l'execució retornant cert. En cas contrari, l'execució continuarà fins el final i es retornarà fals, doncs cap vèrtex era vital per mantenir la propietat.

Cost El cost d'aquest algorisme és similar al de l'anterior algorisme, però en aquest cas, hem de comprovar per cada vèrtex dins del conjunt dominant si tots els vèrtexs estan coberts o no. Com es pot veure això pot ser molt ineficient doncs en cas pitjor, tindrem que la cardinalitat del conjunt dominant és la mateixa que la del conjunt de vèrtexs del graf i com hem d'aplicar l'anterior algorisme per cada vèrtex del conjunt dominant, obtindrem un cost exponencial respecte el nombre de vèrtexs. $O(n(n+m)) = O(n^2m)$. En general però, podem dir que el cost de l'algorisme és de $O(k(n+m))$ on $k = |D|$, $n = |V|$ i $m = |E|$

3 Aproximació "Greedy"

3.1 Algorisme

La primera implementació que hem fet per trobar una aproximació del Mínim Conjunt Dominador d'Influència Positiva ha sigut amb una algorisme golafre. Després d'investigar una mica, hem vist que al llarg dels anys s'han desenvolupat diversos algorismes golafres que resolen aquest problema. Nosaltres hem optat per implementar un dels més recents, l'algorisme de *Pan*. A continuació hi tenim el pseudo-codi:

Algorithm 3 Algorisme Voraç de Pan [1]

Input: Graf simple, connex i no dirigit $G = (V, E)$
Output: Un Conjunt Dominador d'Influència Positiva $S \subseteq V$

```
Ordenar G per ordre ascendent del grau dels vèrtexs
S ← ∅
S̄ ← V \ S
for i = 1 to n do
    if h_S(v_i) > 0 : v_i és un vèrtex no cobert then
        ρ ← h_S(v_i)
        for j = 1 to ρ do
            u* ← argmax {cover-degree(u) | u ∈ N_S̄(v_i)}
            S ← S ∪ {u*}
            S̄ ← V \ S
        end for
    end if
end for
return S
```

El pseudo-codi mostrat és el mateix que el que apareix en l'article de Salim i de Blum [1], doncs és en el que ens hem basat per desenvolupar l'algorisme voraç. Cal introduir dos conceptes que apareixen en el codi.

$$h_S = \left\lceil \frac{grau(v)}{2} \right\rceil - |N_S(v)| \qquad on \;\; N_S(v) = N(v) \cap S \tag{2}$$

$$cover\text{-}degree(v) = |\{u \in N(v) : h_S(u) > 0\}| \tag{3}$$

L'equació (2) representa la proporció dels veïns de v que tenen com a mínim la meitat de les adjacències amb vèrtexs que pertanyen al conjunt Dominador S. És un indicador de si el vèrtex està cobert o no.

L'equació (3) en canvi, ens diu quants vèrtexs que són veïns d'v ténen com a mínim la meitat de les adjacències al conjunt Dominador. En altres paraules, els vèrtexs que tenen el valor de $h_S(u)$ més gran que 0 per tot veí de v.

Correctesa La correctesa de l'algorisme prové de només afegir un vèrtex al conjunt solució quan cap dels seus vèrtexs adjacents està cobert. D'aquesta manera, si ho fem per cada vèrtex, cada cop anirem afegint més vèrtexs al conjunt fins que tinguem un PIDS.

Optimalitat L'optimalitat d'aquest algorisme es deu a les dues propietats mostrades anteriorment que compleixen els vèrtexs del graf i a més a l'ordenació previa que fem dels vèrtexs (línia 1). Com ordenem els vèrtexs per ordre ascendent del seu grau, ens assegurem que primer afegirem els vèrtexs que estiguin més aïllats, que segur que hauran de tenir el seu vèrtex adjacent dins del conjunt Dominador. Després d'això, només afegirem vèrtexs quan sigui estrictament necessari, és a dir, quan els vèrtexs anteriors no siguin suficients per cobrir la resta de vèrtexs.

Cost El cost d'aquest algorisme pot ser diferent depenent de la implementació, doncs l'ordenació es pot fer de diverses maneres. En el nostre cas, com vam usar un Merge Sort per ordenar, tenim un cost $O(nlgn)$ d'inici. Com només ordenem els vèrtexs i no les adjacències, el cost és en base de n. Per la resta del codi, haurem de recórrer tots els vèrtexs i per cadascun fer tantes iteracions com calgui per cobrir-los. Aquest cost és una mica més complicat de sospesar, doncs no podem saber a priori quants vèrtexs estaran coberts per cada adjacència de cada vèrtex. En cas pitjor però, haurem de recórrer totes les adjacències de cada vèrtex obtenint un cost de $O(n + m)$. Així doncs el cost total de l'algorisme serà de $O(nlgn) + O(n + m)$ que és $O(nlgn + m)$.

3.2 Experimentació

Tot i haver fer una anàlisis de la correctesa de l'algorisme, és important comprovar que el resultat que doni sigui acceptable i experimentar per veure el temps d'execució amb diverses entrades. En la següent taula tenim els valors obtinguts per cada entrada diferent, tan en mida de la solució com en temps d'execució, a més dels valors més bons coneguts per aquestes entrades. Quan posem un temps d'execució de 0.00 segons volem dir que l'execució ha trigat menys de 0.01 segons.

Taula 2: Taula comparativa d'execucicons entre algorismes greedy de Pan

Execució				
	Implementació Nostra		**Implementació Òptima**	
Entrada	**Mida**	**Temps (s)**	**Mida**	**Temps (s)**
ego-facebook	2141	1.53	1978	0.062
graph_actors_dat	4851	1.41	3215	0.015
graph_CA-AstroPh	9295	0.92	7030	0.031
graph_CA-CondMat	11720	0.33	9816	0.00
graph_CA-HepPh	5752	0.36	4857	0.015
graph_football	81	0.00	69	0.00
graph_jazz	100	0.02	83	0.00
soc-gplus	9202	0.72	8351	0.031
soc-fb-Brandeis99	2026	4.78	1522	0.031
socfb-Mich67	1954	1.20	1458	0.016

Com es pot apreciar en la Taula 2, els valors obtinguts amb la nostra implementació difereixen bastant dels obtinguts amb la millor implementació que es coneix fins ara. En general, les mides de les nostres solucions són més grans i també triguen més en executar-se. En concret, el graf *socfb-Brandeis99* triga 4 segons més del que triga la millor versió. La diferència és realment gran i es pot deure a les diferents decisions que haguem pres en la implementació que fan molt ineficient la cerca.

Per comprovar que el nostre *greedy* funciona bé, també hem utilitzat l'algorisme mencionat en la secció 2, per saber si realment ens donava un Conjunt Dominador d'Influència Positiva i per saber també si era minimal. Per totes les entrades hem vist que els conjunts calculats són PIDS i que també són minimals, cosa que concorda amb la justificació de l'optimalitat donada anteriorment.

4 Cerca Local

La segona tècnica emprada ha sigut la Cerca Local. Aquesta, és una tècnica Heurística molt emprada per resoldre problemes d'optimització [7], problemes d'intel·ligència artificial o problemes NP o NP-Hard i que forma part del conjunt d'algorismes de cerca informada.

L'objectiu dels algorismes de cerca informada és reduir l'arbre d'exploració d'un problema usant una funció que indiqui a quanta distància ens trobem de la solució òptima del problema en concret. A aquesta funció se l'anomena *heurística*[8]. Així doncs, la cerca informada és aquella que pretén trobar una solució òptima usant un heurístic. Encara que la cerca local és una cerca informada, la seva definició difereix una mica de la general.

La cerca local només es pot implementar quan un problema pot ser representat com una solució [5]. Per exemple, en el cas d'aquesta pràctica, una solució pot ser tot el conjunt de vèrtexs del graf, doncs aquest conjunt segur que és PIDS tot i que, probablement, no sigui mínim.

Així doncs, la cerca parteix d'una solució inicial i va fent petits canvis per intentar millorar la solució actual. La definició de si una solució és millor o no la dona l'heurístic i si al fer el canvi es veu que la nova solució dona un pitjor heurístic, el canvi no s'aplica i es busquen altres opcions. Com sempre hem de tenir representada una solució, els canvis no poden generar-nos no solucions. Tornant al nostre problema, una no solució seria treure un vèrtex d'un conjunt dominador que ja és minimal. Per aquestes raons, tot això s'aplica fins que no es poden fer més canvis o fins que l'heurístic deixa de ser millor per qualssevol solució nova possible. [7]

Cal recalcar que la cerca informada és molt costosa i només s'ha d'aplicar quan un problema sigui NP o NP-Hard.

De cerca local n'hi ha de diversos tipus, però nosaltres ens centrarem només en la que hem implementat, que és la cerca local determinista amb *best-improvement*.

4.1 Cerca Local Determinista

La `Cerca Determinista` es basa en generar a cada pas totes les solucions possibles que es poden obtenir i agafar-ne la millor. Per facilitar la notació, direm que els canvis que es poden fer a una solució es diuen operadors i que les possibles solucions generables amb els operadors es diuen successors.

La cerca determinista, o també coneguda com *Hill-Climbing*, comença des d'una solució inicial i busca el millor successor a base d'aplicar tots els operadors possibles.

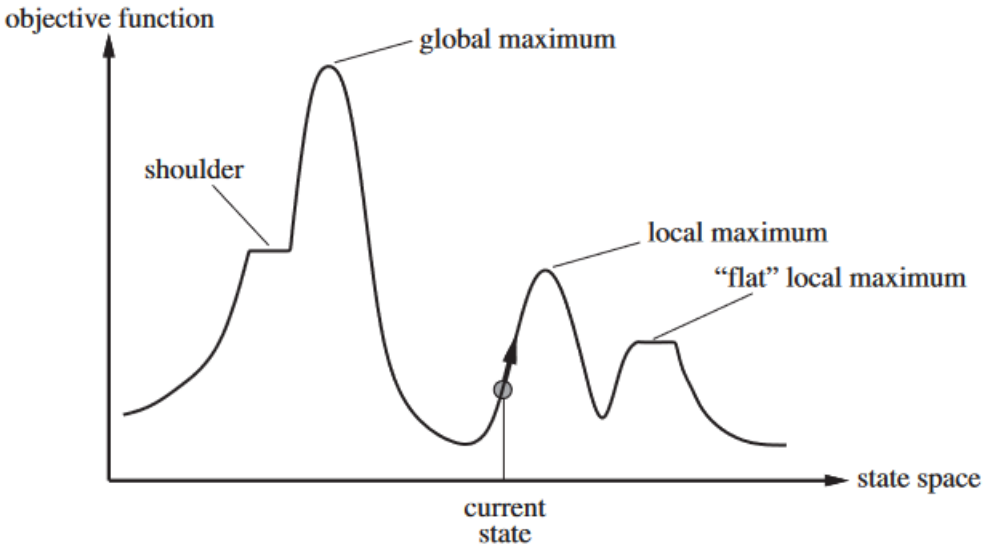


Figura 2: Caracterització de Hill-Climbing al llarg de l'espai de solucions amb una funció objectiu (Extret de [7])

En la Figura 2 podem veure com actua l'algorisme. Sense entrar molt en detall, l'algorisme busca sempre la millor solució des del punt en el que es troba. El problema apareix quan l'estat en el que es troba és un punt molt baix i per molt que apliquem l'heurístic sempre acabarem en un màxim local, com en l'exemple de la figura.

Aquesta és una de les desavantatges de la cerca determinista doncs pot acabar en un màxim local i no en el global que esperaríem. A més a més, aquests algorismes també són molt costosos doncs han de buscar tots els successors possibles aplicant tots els operadors. Tot i això, és molt útil per aproximar-nos molt al màxim local, doncs al mirar tots els successors, ens assegurem que el que agafem és el millor d'entre tots[5].

4.2 Algorisme de Hill-Climbing

Com ja hem mencionat, per la nostra implementació hem optat per seguir una estratègia de *Hill Climbing*. Per al disseny del nostre algorisme, hem seguit l'estructura bàsica d'una cerca local determinista. A continuació hi tenim el psuedo-codi:

Algorithm 4 Algorisme de Cerca Local per Hill Climbing

Input: Graf simple, connex i no dirigit $G = (V, E)$
Output: Un Conjunt d'Influència Positiva $S \subseteq V$

```
1: initialState ← generateInitialState(G)
2: actualState ← initialState
3: end ← false
4: while not end do
5:     Sons ← generateSuccessors(actualState)
6:     if |Sons| > 0 then
7:         actualState ← getBetterSon(Sons)
8:     else
9:         end ← true
10:    end if
11: end while
```

En el nostre algorisme, generem la solució inicial considerant tots els nodes del graf com a solució. Això ens permet explorar tot l'espai de solucions i considerar inicialment qualsevol camí possible.

Per generar els estats successors, apliquem un operador a l'estat actual. En el nostre cas, comptem amb un únic operador d'esborrat intel·ligent, i l'apliquem a cada node del graf. Aquest operador, comprova si el graf que obtenim eliminant un node n és solució, i l'afegeix com a successor en cas afirmatiu.

Un cop obtenim tots els estats successors, comprovem quin d'ells ens proporciona un millor heurístic, i actualitzem l'estat actual.

Per últim, quan trobem que ja no hem generat nous estats successors, o no obtenim un millor heurístic per ningun successor, acabem la cerca i retornem l'estat actual.

Heuristic

L'heurístic que hem escollit és el següent:

$$heuristic = solutionSize - \frac{1}{degree(deletedVertex)}$$

solutionSize: nombre de nodes que té la solució
deletedVertex: vèrtex que estem eliminant de la solució per generar una nova
degree(): retorna el grau del node que eliminem

Podem justificar que el nostre heurístic generarà millors solucions a cada iteració, ja que amb *solutionSize* ens assegurem de que dues solucions amb un nombre diferent de nodes no ens donin el mateix resultat.

Al mateix temps, també tenim en compte el nombre de veïns del node que estem eliminant. De manera que, quan més petit és el grau del node que eliminem, més gran serà l'heurístic que obtinguem.

Per tant, amb el nostre heurístic minimitzem el tamany de la solució i maximitzem els graus dels seus nodes, prioritzant l'eliminació dels vertexs amb menys quantitat de veïns.

Cost

El cost del nostre algorisme prové de diferents parts:
Primerament, al generar la solució inicial, inicialitzem la solució amb tots els vèrtexs amb un cost de $\theta(n)$. També inicialitzem una estructura de dades per guardar el grau dels nodes dins la solució en un temps de $\theta(n)$.

A continuació, entrem al bucle i generem els estats successors en un temps de $O(n)$, i obtenim el heurístic mínim entre els successors en un temps de $O(n)$.

El nostre bucle, l'executarem com a molt $n/2$ cops.

Per tant, el nostre cost total serà de:

$$O(2n) + O((n/2) * (2n)) = O(2n + n^2 = O(n^2).$$

4.3 Proves i Dissenys anteriors

Abans d'arribar al disseny presentat, vam realitzar diferents aproximacions per l'algorisme de *Hill Climbing*.

Com a primer disseny vam plantejar i dissenyar un *Hill Climbing* que començava amb una solució inicial buida i, comptant amb un operador per afegir vèrtexs de manera intel·ligent, construïr una solució minimal. Amb aquest disseny, vam trobar dificultats per a dissenyar un bon heurístic, així com vam observar un cost temporal molt gran.

Després d'aquest primer disseny, vam decidir implementar una solució inicial plena, que comptava amb dos operadors: un per eliminar vèrtexs i un per fer un *swap*, que intercanviava un vèrtex de la solució per a un vèrtex fora de la solució. Aquesta versió del nostre *Hill Climbing*, mirava d'eliminar tots els vèrtexs que es podia, i quan ja no se'n podien eliminar més, s'escollia el millor *swap* segons l'heuristic i es tornava a comprovar si es podien eliminar més vèrtexs. Al executar aquesta versió vam observar un cost en temps i espai molt gran, degut en part al factor de ramificació del operador *swap*.

En la nostra última versió, vam decidir eliminar l'operador *swap*, canviar el nostre heurístic per un millor, i optimitzar en temps i espai el nostre programa, canviant les estructures de dades i part de les nostres funcions. Aquests canvis ens van permetre obtenir un millor temps i resultats.

4.4 Experimentació

Per provar la nostra versió de Cerca Local per *Hill Climbing* en un entorn realista, i mesurar la qualitat de les seves solucions, així com el temps d'execució, hem experimentat amb diverses entrades de diferents mides. Els resultats els podem observar en la taula 3.

Taula 3: Execució de l'algorisme de Cerca Local amb solució inicial tots els vèrtexs

Execució		
	Implementació Nostra	
Entrada	**Val**	**Temps (s)**
ego-facebook	1977	6.75
graph_actors_dat	3208	42.57
graph_CA-AstroPh	6984	126.92
graph_CA-CondMat	9779	61.53
graph_CA-HepPh	4819	41.4
graph_football	70	0.00
graph_jazz	83	0.01
soc-gplus	8310	19.83
soc-fb-Brandeis99	1517	19.48
socfb-Mich67	1441	9.81

Com podem observar a la taula, els resultats obtinguts són millors que els observats amb el *greedy*. Aquesta diferència de resultats és major segons la mida de l'entrada. Al mateix temps, observem una diferència considerable en el temps d'execució respecte el nostre algorisme *greedy*. El nostre algorisme golafre té un temps molt millor al obtingut amb cerca local. També observem que el nostre temps d'execució augmenta segons la mida de l'entrada.

A més a més, hem decidit comprovar quin efecte tindria en les solucions i el temps d'execució provar les mateixes entrades per la mateixa implementació de Cerca Local variant la solució inicial. Ara, en comptes de començar amb la solució plena, fem un càlcul d'una solució inicial amb el nostre *greedy*. Els resultats obtinguts per aquesta variació del nostre algorisme de *Hill Climbing* es mostren en la taula 4.

Taula 4: Execució de l'algorisme de Cerca Local amb solució inicial del càlcul de l'algorisme *greedy*

Execució		
	Implementació Nostra	
Entrada	**Val**	**Temps (s)**
ego-facebook	2004	2.37
graph_actors_dat	3802	6.46
graph_CA-AstroPh	7899	11.22
graph_CA-CondMat	10446	5.08
graph_CA-HepPh	5116	2.9
graph_football	72	0.00
graph_jazz	94	0.02
soc-gplus	8463	1.71
soc-fb-Brandeis99	1746	8.78
socfb-Mich67	1671	3.13

Per aquesta nova versió, seguim observant millors resultats respecte el nostre algorisme de *greedy*, però podem veure que la qualitat de les nostres solucions han disminuït en relació a la versió anterior. Amb el temps d'execució, en canvi, observem una gran millora, sobretot amb entrades molt grans.

Després d'haver analitzat el nostre algorisme de Cerca Local per diferents entrades i amb diferents solucions inicials, hem pogut concluïr que la solució inicial de la que partim sempre influirà en els resultats que obtinguem.
També hem observat que utilitzant una solució inicial plena explorem tot l'espai de solucions i, en conseqüència, obtenim un resultat molt proper al òptim. Amb el *greedy* com a solució inicial, en canvi, obtenim una gran millora en el temps d'execució.

Per tant, si la nostra prioritat és la qualitat de les solucions, la versió amb la que aconseguiríem millors resultats seria amb la primera. En canvi, si no ens importa obtenir una solució amb un marge de diferència més gran respecte la solució òptima, per les entrades grans seria més recomanable utilitzar la segona versió del nostre algorisme.

5 Metaheurístiques

La metaheurística és un procediment dissenyat per trobar, generar o seleccionar un heurístic que pugui proporcionar una solució suficientment bona a un problema d'optimització, especialment quan la informació és incompleta o la capacitat de càlcul és limitada [3]. La metaheurística pot fer poques suposicions sobre el problema d'optimització que s'està solucionant, i per tant, pot utilitzar-se en una gran varietat de problemes.

En comparació a algoritmes d'optimització i mètodes iteratius, les metaheuristiques no garanteixen que es pugui trobar una solució global òptima en alguns problemes. Moltes metaheurisitques s'implementen de forma estocàstica, de manera que la solució obtinguda depèn d'un conjunt de variables aleatòriament generades. En canvi, al buscar sobre un gran conjunt de solucions, la metaheuristica pot trobar bones solucions amb menys esforç computacional que els algoritmes d'optimització, mètodes iteratius o heurístics senzills.

Moltes metaheuristiques es basen a partir de components algorítmics alternatius, com algoritmes voraç i algoritmes de cerca local. Exemples d'aquests són metaheuristiques que inclouen recuita simulada, cerca tabú, cerca local iterada (com l'exemple de la figura 5) i cerca adaptada a algoritmes voraç alteatoris, que és la metaheursítica que hem implementat per aquest problema.

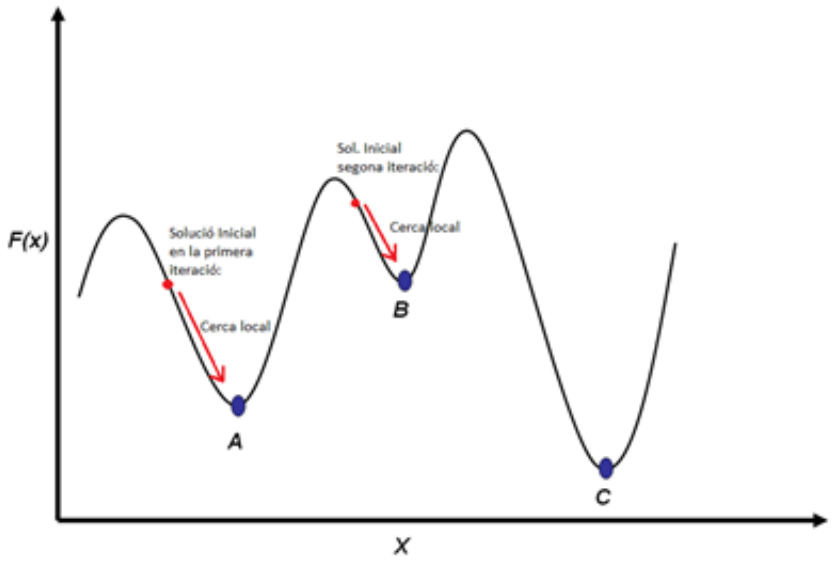


Figura 3: Caracterització de la Metaheurística per la Cerca Local Iterada (Extret de [3])

5.1 Algorisme de GRASP

L'algorisme metaheuristic que hem decidit utilitzar, anomenat Greedy Randomized Adaptive Search Procedure (GRASP) en anglès, acostuma a utilitzar-se per a problemes d'optimització combinatòria. Consisteix en iterar de forma repetitiva la construcció d'una solució inicial amb un algoritme voraç de forma aleatòria i buscar una millor solució d'aquest amb cerca local conservant després de totes les iteracions quin ha estat el millor resultat [6].

Així doncs, si en una iteració partim d'una solució inicial de l'algorisme voraç i aconseguim que la cerca local aconsegueixi el seu mínim local, podríem pensar que iterant de forma repetitiva suficients cops, trobaríem millors solucions i podríem aconseguir la solució global més optima.

Tot i això, no podem garantir, degut a la generació de solucions inicials de forma aleatòria, que puguem trobar mai la solució optima global i menys si l'heurístic de la cerca local tampoc ens garanteix aconseguir el mínim local (que podria ser el global). Tot i això, és una molt bona alternativa per aconseguir bones solucions amb poc esforç i de manera eficient sense necessitar molta capacitat de càlcul.

A continuació tenim el psuedo-codi de l'algorisme (tenir en compte que el nom de les variables no correspon al mateix que al nostre codi i que simplement és una interpretació):

Algorithm 5 Algorisme de GRASP [6]

Input: Graf simple, connex i no dirigit $G = (V, E)$
Output: Un Conjunt d'Influència Positiva $S \subseteq V$

```
1: nApps = 10
2: Resultats ← ∅
3: for i = 1 to nApps do
4:     SolucioInicial ← RandomizedGreedy
5:     SolucioFinal ← LocalSearch
6:     Resultats[i] = SolucioFinal
7: end for
8: MillorSolucio ← max{Resultats}
9: return MillorSolucio
```

L'hem implementat de la següent forma : primer hem decidit que volíem generar 10 iteracions possibles i escollir la millor solució entre aquestes (nApps = 10) i després hem creat un vector per a conservar les solucions obtingudes (Resultats). Per cada iteració, per generar solucions inicials aleatòries, en comptes d'ordenar els vèrtex del graf de forma creixent per executar l'algoritme voraç amb aquesta, hem decidit posicionar els vèrtex de forma aleatòria per a que l'algoritme ens retornes una solució inicial aleatòria. A partir d'aquesta solució inicial apliquem la nostra cerca local i obtenim el nostre resultat, conservant-lo per a comparar-lo posteriorment.

Cost

El cost de l'algoritme varia depenent del nombre d'iteracions que es vol aplicar per solucionar el problema de formes diferents. En el nostre cas com hem escollit simplement iterar 10 cops, considerarem que el cost total serà el cost de cada iteració per 10, ja que també per aconseguir la millor solució, podem generar una variable que es vagi actualitzant en cas de trobar una solució millor, en comptes de buscar la millor un cop acabades les iteracions.

Cada iteració contemplarà una generació per cada posició dels vèrtex aleatòria $O(n)$, l'algoritme voraç que hem implementat, Pan's algorithm, que té una complexitat de $O(nlgn + m)$ però com que no ordenem els vèrtex correspon a $O(n+m)$, i el cost de la nostra cerca local que correspon a $O(2n+n^2)$, tot junt donaria un cost aproximat de $O(n^2 + m)$.

5.2 Experimentació

Degut a que la metaheuristica genera varies solucions per cada experiment, ens ha semblat important recollir les dades com a mitjanes i la millor solució que ha sigut capaç de generar l'algoritme. Els resultat es poden veure en la taula 5.

Taula 5: Taula d'execució de l'algorisme de GRASP per cada entrada

Execució			
Entrada	**Mitjana Val**	**Temps (s)**	Millor Val
ego-facebook	19998	7.51	1994
graph_actors_dat	3686	15.65	3639
graph_CA-AstroPh	7271	28.26	7210
graph_CA-CondMat	10213	27.53	10213
graph_CA-HepPh	4957	8.73	4939
graph_football	73	0.00	73
graph_jazz	98	0.03	96
soc-gplus	8424	3.65	8406
soc-fb-Brandeis99	1707	27.9	1671
socfb-Mich67	1623	6.55	1589

Com podem veure, en general no ens ha proporcionat els millors resultats de les nostres experimentacions, però tenint en compte el temps d'execució, si que hi ha una gran diferencia. És cert que comparant els problemes petits no es nota tant la diferencia, en canvi amb els problemes grans si que es pot apreciar una diferencia considerable. Contemplant els problemes grans, respecte el nostre algoritme voraç aconsegueix uns resultats més bons, tot i que trigui més temps en trobar la solució, en canvi a comparació amb la nostra cerca local, no aconsegueix uns resultats tant bons però si que millora considerablement el temps d'execució.

Com que ja hem comprovat que el nostre algoritme voraç no pot generar resultats que no siguin solucions i que la nostra cerca local tampoc pot generar no-solucions, no hem utilitzat l'algorisme de comprovació de PIDS per comprovar que els resultats són solucions.

6 Programació Lineal

Definirem com a `Programació Lineal` [2] el mètode matemàtic utilitzat per determinar una manera d'aconseguir el resultat òptim a partir d'unes relacions lineals. També es pot definir com una tècnica d'optimització de funcions d'objectiu lineal. Per norma general, tot problema de programació lineal és pot expressar seguint les següents restriccions:

$$\begin{array}{cc} \textbf{maximitzar / minimitzar} & c^T \cdot x \\ \textbf{restriccions} & A \cdot X \leq b \\ & x \geq 0 \end{array} \tag{4}$$

D'aquesta descripció canònica, podem extreure que el valor de la variable x és l'objectiu a calcular, els vectors c i b són coeficients coneguts i, per últim, el valor de A es una matriu amb coeficients coneguts.

La programació lineal està molt estesa en problemes d'optimització. Un dels problemes més coneguts, que es pot resoldre mitjançant programació lineal, és podria definir com:

Suposem que un polític vol guanyar les eleccions del seu territori. Un territori que disposa de tres zones ben diferenciades, on cada zona té unes necessitats i inquietuds diferents. A partir d'aquí, és vol maximitzar el número de votants de cada zona, intentant minimitzar el cost de la publicitat que es farà.

Per tal de minimitzar els costos en publicitat, haurem d'adaptar la campanya electoral per englobar les diferents zones del territori, per exemple, en una zona rural i en una àrea metropolitana, les necessitats i inquietuds són diferents, per això, l'objectiu és saber quin tipus de campanya electoral haurem de fer per tal de maximitzar el nombre de votants. A partir d'unes estimacions del cost que ens suposa obtenir un nombre de vots, en cada zona rural, pels diferents tipus de necessitats que tenen els nostres ciutadans. A partir d'aquestes restriccions, hem de ser capaços de formular una descripció canònica com la que s'ha mostrat al principi d'aquesta secció.

6.1 Programació Lineal Entera

Definirem la `Programació Lineal Entera` [2], o també conegut com `ILP`, com una variació de la `Programació Lineal` tal que les variables que utilitzarem són enteres. Un problema `ILP` és pot definir de la mateixa manera que un problema de `Programació Lineal`, però només acceptarem valors enters com a possible solució del sistema. Aquesta restricció afegida de vegades ens permet solucionar més fàcilment els exercicis o, a vegades, fa que siguin impossibles de resoldre.

6.2 MPIDS utilitzant ILP

Per tal de plantejar aquest problema com a `Programació Lineal Entera` considerem l'existència d'una variable binària que anomenarem x_i, la qual emmagatzemarà si el vèrtex $v_i \in V$ esta dins del subconjunt dominador. Tenint l'existència de la variable x_i en ment, formularem la següent descripció canònica del problema:

$$\begin{aligned} &\textbf{Minimitzar} && \textstyle\sum_{v_i \in V} x_i \\ &\textbf{Subjecte a} && \textstyle\sum_{v_j \in N(v_i)} x_j \geq \left\lceil \frac{\deg(v_i)}{2} \right\rceil \quad \forall v_i \in V \\ & && x_i \in 0, 1 \end{aligned} \tag{5}$$

A més a més, a part de la descripció de x_i, definirem $N(v_i)$ com el veïnatge de v_i al graf G d'entrada i $\deg(v_i) = |N(v_i)|$ és el seu grau.

6.3 Introducció a CPLEX

CPLEX [4] ofereix llibreries per diversos llenguatges de programació, com `C`, `C++`, `Java` i molts altres, per resoldre problemes de programació lineals o relacionats. Més concretament, podríem dir que serveix per resoldre problemes d'optimització amb restriccions lineals.

En els problemes d'optimització més bàsics, les variables de la funció objectiu són continues en el sentit matemàtic, sense espais entre els valors reals. Per resoldre aquests problemes de programació lineal, CPLEX implementa optimitzadors basats en algorismes simples, així com algorismes de barrera logarítmica.

Tal i com s'ha afirmat anteriorment, CPLEX té suport per diferents llenguatges de programació. Nosaltres varem triar `C++`, el llenguatge utilitzat en les assignatures prèvies de la carrera.

Per tal de programar amb CPLEX haurem d'incloure els objectes necessaris procedents de la llibreria. Aquests objectes els podem dividir en dues categories ben diferenciades. Primerament definirem els objectes de modelatge, els quals s'utilitzen per definir el problema d'optimització. Aquests objectes es troben agrupats en la classe `IloModel` que representa el problema d'optimització complert. D'altra banda, també existeixen els objectes `IloCplex`, el qual llegeix un model (`IloModel`) i n'extreu les dades necessaries per la representació.

6.4 MPIDS utilitzant CPLEX

A l'hora d'implementar l'algorisme, vam fer ús de la descripció canònica descrita en la secció 6.2. Primerament, inicialitzarem l'entorn de programació de CPLEX, el qual s'invoca mitjançant la classe `IloEnv`. Normalment, la variable que inicialitza la classe `IloEnv` s'acostuma a anomenar `env`.

Un cop s'hagi definit l'entorn de programació, definirem un model mitjançant la classe `IloModel`, la qual necessita un entorn per la seva constructora. Els objectes de la classe `IloModel` s'utilitza per definir el model d'optimització que després extraurem mitjançant la classe `IloCplex`. En el cas de que creem el model sense un entorn, ens crearà un model buit, el qual no pot ser utilitzar per res, que per fer copies de models construïts anteriorment. Aquest detall és molt important, ja que si no es te en compte, pot portar a múltiples errors.

Un cop s'ha creat el model, haurem de afegir els diferents objectes que ens definiran el model a optimitzar. Les classes més utilitzades per definir aquest model són:

- `IloNumVar:` Representa les variables del model.
- `IloRange:` Defineix la restricció, de forma que $l \leq \text{expr} \leq u$, on expr és la expressió lineal, i "l i u són variables que defineixen l'interval.
- `IloObjective:` Representa una funció objectiu.

Un cop s'han creat els objectes, els podrem afegir al model mitjançant `model.add(object)`, on `model` representa la variable que referencia el model i, `object` representa la variable que referencia l'objecte creat.

En el nostre cas, per tal d'implementar l'algorisme MPIDS amb CPLEX, definirem una variable x, com un valor enter, que esta dins del interval $[0, 1]$. Aquest valor, el definirem mitjançant la construcció `IloNumVar x(env, a, b, type)`, on el nombre x estarà dins del interval $[a, b]$, i serà del tipus TYPE, on el definirem entre ILOFLOAT, ILOINT o ILOBOOL.

Una instancia de la classe `IloExpr` representa una expressió dins del model. Podríem dir que aquesta classe crea handles. Cada variable d'una expressió ha de pertànyer únicament a un entorn. Per exemple, en el nostre cas, definirem la variable *obj*, la qual ens servirà posteriorment per fer els càlculs de minimització del problema. Per afirmar que `obj` és una variable a `minimitzar`, afegirem en el nostre codi el següent `model.add(IloMinimize(env, obj))`, on `model` referencia la variable de `IloModel`, `env` referencia la variable de `IloEnv` i, per últim, `obj` referencia la variable de `IloExpr`.

Per tant, el fragment més important del nostre pseudo-codi, per la versió de CPLEX, és:

Algorithm 6 Algorisme usant Programació Lineal Entera

```
1: IloModel model ( env )
2: IloNumVarArray x ( env, n_of_nodes, 0, 1, ILOINT )
3: IloExpr obj ( env )
4: for i = 1 to n do
5:     obj += x_i
6: end for
7: model.add ( IloMinimize ( env, obj ) )
8: for all i ∈ [0, n] do
9:     IloExpr expr ( env )
10:    for all j ∈ neighbors[i] do
11:        expr += j
12:    end for
13:    value = ⌈ neighbors[i].size / 2 ⌉
14:    model.add ( expr ≥ value )
15: end for
```

Per acabar, definirem que una instancia de la classe `IloCplex` s'encarrega de resoldre una variable de model que referencia una instancia de la classe `IloModel`. En aquesta classe se li pot modificar parametres d'execució, com per exemple, el nombre de threads que utilitzarà, el temps d'execució màxim del programa, entre altres factors.

Cal mencionar que totes les variables utilitzades s'han de finalitzar abans de que el script acabi, ja que sinó es poden quedar ocupant memòria innecessàriament. Per exemple, per finalitzar una instancia de la classe `IloEnv`, caldrà ficar `env.end()` a la última línia de la funció que utilitza l'entorn de programació de CPLEX.

Per tal de comprovar el correcte funcionament del algorisme implementat en CPLEX, s'han realitzat diversos proves amb els diferents datasets donats. Totes les proves s'han realitzat en un equip de sobre taula amb les següents característiques:

Processador:	Intel Core i7-3770 CPU @ 3.40GHz
Memòria RAM:	7.7 GiB a 1600MHz
Sistema Operatiu:	Debian GNU / Linux 11 (bullseye)

A més a més, per intentar reduir els temps d'execució dels diferents datasets, vam modificar el nombre de threads dels que disposava el script de CPLEX, arribant a ficar-li el màxim que ens permetia la CPU, aquest màxim és 8. Es important recordar, que podem modificar el nombre de threads que pot utilitzar CPLEX mitjançant la següent instrucció `cpl.setParam(IloCplex::Threads, 8)` Fent ús de la computadora que acabem de mencionar i modificant el nombre de threads que pot utilitzar CPLEX, vam decidir ficar com a temps límit d'execució 7200 segons, o en altres paraules, 2 hores.

Amb la computadora que hem mencionat anteriorment i un temps límit de 7200 segons, vam començar executant el dataset de `graph_CA-AstroPH.txt`. Aquest data set, que consta de 18771 nodes i 198050 arestes, ens va donar el següent resultat:

value	**time**	**gap**
18771.00	2.02	91.15
7054.00	446.02	6.17
6817.00	447.95	2.91
6816.00	446.41	1.18
...	...	...
6740.00	49326.43	0.32

La solució que ens ha retornat el programa no és òptima, és a dir, hi haurà una solució millor, però aquesta solució ha trigat un total de dues hores en calcular-la i, ens ha retornat timeout. Aquest resultat, el de no trobar un resultat òptim, ens ha sorgit en més d'un dataset.

Malgrat això, un resultat interessant d'analitzar és el de `ego-facebook.txt`, al ser un dataset força petit, el nostre programa ha sigut capaç de trobar un resultat òptim, abans de llençar timeout. El resultat que ens ha retornat és:

value	**time**	**gap**
4039.00	0.33	99.63
2049.00	0.85	3.73
1974.00	0.91	0.08
1973.00	1.34	0.00
1973.00	1.35	0.00

7 Conclusions

Com hem vist al llarg del treball, resoldre un problema NP-Hard no és fàcil, per molt que usem tècniques avançades o heurístiques molt bones. Al final el problema no deixa de ser difícil i no es pot comprovar si la solució obtinguda és correcta o no. Tot i això, hem entès que es poden obtenir aproximacions molt bones a les solucions òptimes depenent de la tècnica que emprem.

De les tècniques que hem usat, la voraç ja la coneixíem doncs la hem treballat en la assignatura. De les altres no en sabíem res abans de fer el treball. Hem après que la cerca local és molt útil quan tenim un problema que pot ser representat com una solució, sobre la qual hi anem fent millores per obtenir un resultat molt bo. De la part de metaheurístiques hem vist que les cerques informades poden ser millorades a través d'aplicar noves heurístiques fins convergir en una solució molt bona. De l'últim apartat, el de Programació Lineal Entera, hem pogut entendre la potència de poder representar un problema mitjançant restriccions per agilitzar el càlcul que es faci sobre un problema.

Amb tot això, hem vist que hi han moltes maneres d'abordar aquest tipus de problemes, i que totes són igual de vàlides, encara que algunes donaran millors resultats que d'altres.

Per concloure, en la taula 6 hi tenim una comparativa de l'execució per cada algorisme implementat en aquesta pràctica del *benchmark* donat. La resposta a quin algorisme és millor no és binària, depèn de múltiples factors. Com podem veure l'algorisme que s'ha implementat utilitzant una estratègia `Greedy` té uns temps d'execució força millors que l'algorisme de `Cerca Local`. En canvi, com podem veure l'algorisme de `Cerca Local` és capaç d'aproximar-se més al resultat òptim, i això ho podem veure fent la comparativa amb els resultats de la `Programació Lineal`.

Per tant, en el món professional, hauries de valorar que és més necessari en la teva solució, si tenir uns temps d'execució inferiors o, per contra, tenir uns resultats més propers a l'òptim.

Taula 6: Taula comparativa del temps d'execució i la mida de la solució entre els diferents algorismes

Execució								
	Greedy		**Cerca Local**		**Metaheurístiques**		**Programació Lineal Entera**	
Entrada	**Mida**	**Temps (s)**	**Mida**	**Temps (s)**	**Mida**	**Temps (s)**	**Mida**	**Temps (s)**
ego-facebook	2141	1.53	1977	6.75	1994	7.51	1973	1.35
graph_actors_dat	4851	1.41	3208	42.57	3639	15.65	3092	2919.07
graph_CA-AstroPh	9295	0.92	6984	126.92	7210	28.26	6740	49326.43
graph_CA-CondMat	11720	0.33	9779	61.53	10213	27.53	9584	12289.71
graph_CA-HepPh	5752	0.36	4819	41.4	4939	8.73	4718	1040.95
graph_football	81	0.00	70	0.00	73	0.00	63	10.69
graph_jazz	100	0.02	83	0.01	96	0.03	79	0.28
soc-gplus	9202	0.72	8310	19.83	8406	3.65	8244	0.10
soc-fb-Brandeis99	2026	4.78	1517	19.48	1671	27.9	1403	49773.05
socfb-Mich67	1954	1.20	1441	9.81	1589	6.55	1332	50099.30

Referències

[1] Bouamama i Blum. "An Improved Greedy Heuristic for the Minimum Positive Influence Dominating Set Problem in Social Networks". Cast. A: *Algorithm* 14.79 (2021). URL: https://www.mdpi.com/journal/algorithms.

[2] Thomas H.Cormen et al. "Linear Programming". Ang. A: *Introduction to Algorithms*. 3a ed. The MIT Press, 2009.

[3] Manar Hosny. *Vehicle Routing with Pickup and Delivery: Heuristic and Meta-heuristic Solution Algorithms*. Gen. de 2012. ISBN: 978-3-659-20258-2.

[4] IBM. *Introduction to CPLEX*. Last accessed 28 November 2021. 2021. URL: https://www.ibm.com/docs/en/icos/12.7.1.0?topic=tutorials-tutorial.

[5] Kleinberg i Tardos. "Chp 12: Local Search". Cast. A: *Algorithm Design*. Pearson/Addison-Wesley, 2006, pàg. 661 - 706.

[6] Mauricio G. C. Resende i Celso C. Ribeiro. "Greedy Randomized Adaptive Search Procedures: Advances and Extensions". A: *Handbook of Metaheuristics*. Ed. de Michel Gendreau i Jean-Yves Potvin. Cham: Springer International Publishing, 2019, pàg. 169 - 220. ISBN: 978-3-319-91086-4. DOI: 10.1007/978-3-319-91086-4_6. URL: https://doi.org/10.1007/978-3-319-91086-4_6.

[7] Russel i Norvig. "Beyond Classical Search". Cast. A: *Artificial Intelligence A Modern Approach*. 3a ed. Pearson, 2010, pàg. 120 - 125.

[8] Russel i Norvig. "Solving Problems by Searching". Cast. A: *Artificial Intelligence A Modern Approach*. 3a ed. Pearson, 2010, pàg. 92 - 109.