Universitat Politècnica de Catalunya

Facultat d'Enginyeria Informàtica

Selecció del mínim conjunt objectiu per influenciar xarxes socials

Projecte d'Algorísmia

Quadrimestre de primavera, Abril de 2023

Abstract

Aquest projecte té com a objectiu la implementació de diferents algorismes per a resoldre un problema d'optimització combinatòria i la conseqüent experimentació que permetrà comparar la qualitat de les solucions proposades. El problema a abordar és el de *Target Set Selection* (TSS).

Lluis Llull, Martí Recalde, Miquel Muñoz, Iván Serrano

Contents

1 Introducció **1**
- 1.1 Models de Difusió . . . 1
  - 1.1.1 Independent Cascade (IC) . . . 1
  - 1.1.2 Linear Threshold (LT) . . . 2

2 Primera part: implementació dels models de difusió **3**
- 2.1 Implementació del model d'*Independent Cascade* (IC) . . . 3
- 2.2 Implementació del model d'*Linear Threshold* (LT) . . . 4

3 Segona part: algorismes **6**
- 3.1 Algorisme voraç . . . 6
  - 3.1.1 Algorisme voraç pel model de *Independent Cascade* . . . 6
  - 3.1.2 Algorisme voraç pel model de *Linear Threshold* . . . 9
- 3.2 Algorisme de cerca local . . . 11
- 3.3 Algorisme basat en una metaheurística . . . 14
  - 3.3.1 Procediment bàsic de *Simulated Annealing* . . . 14
  - 3.3.2 Algorisme basat en SA per la maximització de la influència . . . 15

4 Experimentació **17**
- 4.1 Grafs utilitzats . . . 17
- 4.2 Model de difusió . . . 20
  - 4.2.1 Versió 1 . . . 21
  - 4.2.2 Versió 2 . . . 21
  - 4.2.3 Versió 3 . . . 21
- 4.3 Algorisme Greedy . . . 22
  - 4.3.1 IC . . . 22
  - 4.3.2 LT . . . 22
- 4.4 Cerca Local . . . 23
  - 4.4.1 Independent Cascade . . . 23
  - 4.4.2 Linear Treshold . . . 25
- 4.5 Metaheurística . . . 27
  - 4.5.1 LT . . . 27
  - 4.5.2 IC . . . 29

5 Conclusió **31**

A Annex 1: Comparativa de versions pel model IC **32**

1 Introducció

En aquest projecte treballarem en el problema de *Target Set Selection* (TSS), un problema de combinatòria basat en diversos models de difusió de grafs.

El TSS té dues variants principalment, la de **maximització** i la de **minimització**. La variant de maximització busca maximitzar l'abast de la difusió donada una quantitat de recursos limitada, és a dir, un cert nombre k de vèrtex. D'aquesta forma, el problema es troba en identificar quin és el k subconjunt de vèrtex que permet maximitzar la difusió.

D'altra banda tenim la variant de minimització, que serà en la que ens focalitzarem en aquest projecte. L'objectiu d'aquesta variant és trobar quin és el conjunt mínim d'usuaris o nodes que permet extendre la difusió a tota la xarxa.

Un cop vista la definició del problema, veiem que aquest resulta especialment interessant en l'àmbit de les ciències socials, i especialment el *marketing*, doncs aplicat a aquesta àrea ens pot ajudar a fer arribar una campanya publicitària a la major quantitat de potencials consumidors minimitzant els recursos.

1.1 Models de Difusió

Per desenvolupar la variant de minimització del TSS farem servir dos models de difusió diferents:

1. Independent Cascade – IC
2. Linear Threshold – LT

Veiem a continuació una breu definició de cadascun dels models.

1.1.1 Independent Cascade (IC)

Aquest és un model estocàstic, és a dir, que es pot considerar com un sistema dinàmic que varia amb el temps i que està subjecte a certa aleatorietat.

Es fonamenta en la idea de que quan un node i es troba "actiu", aquest intentarà influenciar als seus veïns estocàsticament amb probabilitat $\mathbf{p_{ij}}$, on j és un veí. Per simplicitat, en aquest treball la probabilitat de difusió serà fixa per totes les arestes del graf.

Un cop un node ha realitzar el seu intent de difusió, no ho tornarà a intentar més, però romandrà actiu per sempre. El procés de difusió comença en un cert subconjunt de nodes del graf original i finalitza quan ja no s'activen més nodes.

Formalment, el model de difusió d'*Independent Cascade* és defineix de la següent forma:

Definició 1.1. Donat un graf $G = (V, E, p)$ que representa una xarxa, un conjunt $S \subseteq V$ inicial i una probabilitat $p \in [0, 1]$ (on $\forall (i, j) \in E : p_{ij} = p$), denotarem el nombre total esperat de nodes activats (o influenciats) després de t passos sota un model IC com a $F_t^p(S)$.

1.1.2 Linear Threshold (LT)

En aquest model, cada node té un llindar d'influència que representa la seva resistència a ser influenciat pels altres. A més, cada aresta *(i,j)* té un pes w que representa la influència que el node i té sobre j.

A cada iteració els nodes ja actius influiran en els seus veïns de tal forma que, si algun dels nodes inactius rep una influència total que supera el seu llindar de resistència, aquest s'activa també i s'uneix al conjunt de nodes actius. La propagació es deté quan una iteració no aconsegueix expandir el subconjunt dels nodes actius.

La definició formal d'aquest model de difusió és la següent:

Definició 1.2. Donat un graf $G = (V, E, w, \theta)$ que representa una xarxa (amb pesos a les arestes $w : E \to \mathbb{Z}^+$ i llindars d'influència $\theta : V \to [0, 1]$) als nodes) i un conjunt $S \subseteq V$ inicial de nodes activats, denotarem el nombre total de nodes activats (o influenciats) després de t passos sota un model LT com a $F_t^{w,\theta}(S)$.

En iniciar la difusió ($t = 0$) només hi ha actius els nodes del subconjunt inicial S. A la iteració t+1, un node serà activat si, i només si, la suma de les influències que els nodes actius adjacents tenen sobre ell supera el seu llindar, això és,

$$\sum_{\substack{j \in F_t^{w,\theta}(S) \\ i \in \mathcal{N}(j)}} w_{ji} \geq \theta(i),$$

Tal i com s'indica a l'enunciat del projecte, nosaltres aplicarem algunes simplificacions a aquest model. En primer lloc, considerarem **no dirigits** tot els grafs amb què treballem. A més, suposarem que el llindar de resistència de qualsevol node s'obtindrà a partir de multiplicar el seu grau per una ràtio constant de valor entre 0 i 1 (r). Per últim, assignarem a 1 el pes de totes les arestes. Així, la activació d'un node dependrà estrictament de la fracció dels nodes adjacents a ell que ja estan activats.

2 Primera part: implementació dels models de difusió

La primera part del nostre projecte consisteix a implementar un programa que permeti simular els models de difusió esmentats en l'apartat anterior. Donat que en aquest treball l'eficiència és un factor rellevant, hem decidit fer servir **C++**.

2.1 Implementació del model d'*Independent Cascade* (IC)

Donat un graf no dirigit, una probabilitat de difusió i un subconjunt inicial de nodes activats, volem simular el procés de difusió d'influència fent servir IC. Formalment, l'entrada és la següent.

Entrada: $G = (V, E)$, $p \in [0, 1]$, i $S \subseteq V$
Sortida: $|C|$, on $C = F_t^p(S)$ i $t = \arg\min_{0 \leq \tau \leq n} \{F_\tau^p(S) = F_{\tau+1}^p(S)\}$.

Començant per la modelització del problema, hem decidit implementar el graf com a una llista d'adjacències, concretament com un *array* d'*arrays*. D'aquesta forma, ens allunyem dels costos quadràtics de la matriu d'adjacència, i garantim accessos aleatoris en temps constant a qualsevol dels nodes.

Procedint amb la implementació del programa, el pseudocodi és el següent:

```
1:  procedure IC-DIFUSION
2:      Q ← subset inicial de nodes a propagar
3:      while not Q.emptY() do
4:          u ← Q.front()
5:          for v in u.veins() do                ▷ es visiten tots els veïns del node u
6:              if v no està actiu then
7:                  pp ← ϵ[0, 1]
8:                  if pp > (1 − p) then         ▷ p és la probabilitat de propagació
9:                      Q.push(v)
10:                 end if
11:             end if
12:         end for
13:     end while
14: end procedure
```

Per tal de dur a terme la propagació es van guardant els nodes a propagar en una cua, i no s'intenta continuar la difusió amb un node veí si aquest es troba actiu, per tal de garantir que un node que es trobi actiu no intenta propagar-se a un veí més que un únic cop.

Tal com mostra el pseudocodi, la propagació d'un node a un altre només succeeix quan es venç la probabilitat de difusió de l'aresta entre ambdós nodes.

Per últim, respecte a la complexitat d'aquest algorisme, podem afirmar que és del ordre de O(n + m), ja que en el pitjor dels casos, es revisen els n nodes en cas de que es propagui en cada aresta, i per tant cadascun dels nodes visita tots els seus veïns. Aquest cos és possible gràcies a fer servir una llista d'adjacències en comptes d'una matriu, doncs en aquell cas el cost seria quadràtic.

2.2 Implementació del model d'*Linear Threshold* (LT)

Donat un graf no dirigit G de tamany n, un ràtio r, i un subconjunt de nodes originalment acitus S, volem simular el procés de difusió d'influència segons el model LT tot just explicat (amb les simplificacions indicades). L'entrada i la sortida de l'algorisme a construir es defineixen formalment com segueix:

Entrada: $G = (V, E)$, $p \in [0, 1]$, i $S \subseteq V$
Sortida: $|C|$, on $C = F_t^p(S)$ i $t = \arg\min_{0<\tau<n} \{F_\tau^p(S) = F_{\tau+1}^p(S)\}$.

Per la modelització d'aquest problema seguirem els criteris de disseny emprats a l'apartat anterior i representarem el graf mitjançant una llista d'adjacències.

El pseudocodi de l'algorisme és el següent:

```
1: procedure LT-DIFUSION
2:     l ← Empty list
3:     while (not l.size() == n) and prev != 0 do
4:         prev ← 0
5:         u ← Q.front()
6:         for v in G do                          ▷ es visiten tots els veïns del node u
7:             if not actiu(v) then
8:                 c ← 0
9:                 for u in v.veins() do
10:                    if actiu(u) then
11:                        c++
12:                    end if
13:                end for
14:                if c ≥ r * grau(v) then          ▷ r és el ràtio de propagació
15:                    l.afegeix(v)
16:                    actiu(v) = cert
17:                    prev++
18:                end if
19:            end if
20:        end for
21:    end while
22:    return l
23: end procedure
```

Aquest codi retorna, dins la llista l, el conjunt dels nodes que s'han pogut influenciar partint del subconjunt inicial S.

Com que, forçosament, a cada iteració (a excepció, potser, de la última) afegirem un node nou a la llista, aquest algorisme que ens permet representar la difusió seguint el model LT té cost $O(n+m)$ ja que només recorrem cada vèrtexs un cop, així com les arestes que en surten.

3 Segona part: algorismes

En aquesta segona part del projecte ens focalitzarem en trobar solucions òptimes per al *Target Set Selection*. El nostre objectiu serà trobar el conjunt de nodes **mínim** que aconsegueix influenciar a tot el graf amb tres aproximacions diferents al problema:

1. Un algorisme voraç determinista
2. Un algorisme de cerca local
3. Una metaheurística

3.1 Algorisme voraç

3.1.1 Algorisme voraç pel model de *Independent Cascade*

La idea de l'algorisme voraç és la de prendre la decisió òptima en cada iteració d'aquest, amb l'esperança de trobar una solució òptima per la totalitat del problema.

Després de realitzar un bon procés d'investigació, ens hem trobat amb que un algorisme voraç és certament complex, i res eficient des d'un punt de vista computacional.

Una primera aproximació a aquest algorisme seria la de pensar que en cada iteració hem de seleccionar el node que pot ser més influent en la xarxa. Un simple pseudocodi per aquest algorisme seria el següent:

```
1: procedure GreedyIC
2:     S ← empty
3:     while (not S.size() == n) do
4:         u ← node ins V-S with maximum influence
5:         S.add(u)
6:         if S is a solution then
7:             return S
8:         end if
9:     end while
10:    return S
11: end procedure
```

La part més rellevant d'aquest algorisme es troba en com es computa la influència d'un node. Intuïtivament, la influència d'un node vindrà determinada per la quantitat de nodes que pot arribar a influir, però això no és un càlcul trivial, ja que s'està treballant amb un model estocàstic, i per tant la influència d'un node estarà subjecte a una probabilitat.

Una forma comú d'afrontar aquest escenari és amb el mètode de Montecarlo, molt útil per models no deterministes. Aquest mètode es basa en la simulació reiterada de l'algorisme en qüestió, per tal de poder extreure conclusions sobre el model, que en aquest cas seria identificar el node més influent.

Un pseudocodi per un algorisme *greedy* basat en el mètode de Montecarlo seria el següent.

```
1: procedure GreedyIC-Montecarlo(n)
2:     S ← empty
3:     for i = 1 to n do
4:         Influences[] ← empty set with node influences
5:         for each node u V-S do
6:             Influence[u] ← MC-Simulation(S U u)
7:         end for
8:         v ← node with maximum Influence
9:         S ← S U v
10:        if S is a solution then
11:            return S
12:        end if
13:    end for
14: end procedure
```

És notable a simple vista que aquest no és un bon algorisme voraç, doncs requereix en cada iteració O(nK) avaluacions de la influència, on K és el nombre d'execucions del mètode de Montecarlo, per cada node.

A més, tal com es menciona en l'article *Modeling and maximizing influence diffusion in social networks for viral marketing* [6], un algorisme greedy com l'anterior no és òptim, és tan sols una aproximació amb un factor r = (1 - 1/e - a), per algun a superior *0*.

Descartada la opció anterior, ens hem decantat per una simplificació d'un mètode que és l'estat de l'art en aquest camp, anomenat *Maximum Influence Arborescence Model* (MIA).

Aquest model, introduït per Wei Chen en l'article *Scalable Influence Maximization for Prevalent Viral Marketing in Large-Scale Social Networks* [7], proposa un algorisme polinòmic d'aproximació pel càlcul de la influència dels nodes d'una xarxa, però degut a la seva complexitat nosaltres ens quedarem tan sols amb la següent idea.

> For a path $P = \langle u = p_1, p_2, \ldots, p_m = v \rangle$, we define the *propagation probability* of the path, $pp(P)$, as
>
> $$pp(P) = \Pi_{i=1}^{m-1} pp(p_i, p_{i+1}).$$

Concretament, de tots els camins que poden haver entre dos nodes, només ens interessa el camí que pot resultar en una màxima influència. Al tenir totes les arestes un pes fix, la probabilitat de que un node u pugui influenciar un node v serà igual a p^d, on d és la distància del camí més curt entre ambdós nodes.

Tenint en compte el que s'ha mencionat, la implementació final del nostre algorisme seria com la proposada en el primer algorisme *GreedyIC*, i el càlcul de la influència d'un node es realitza de la següent forma:

```
1: procedure computeInfluence(src)
2:     Q ← empty queue
3:     Q.push(src)
4:     distances(n) ← [inf,inf...inf]
5:     visited(n) ← [false,false...false]
6:     visited[src] ← true
7:     distances[src] ← 0
8:     while not Q.empty() do
9:         u ← Q.front()
10:        Q.pop()
11:        for every v in u.neighbours() do
12:            if not visited[v] and not spreadedNodes[v] then
13:                distances[v] ← distances[u] + 1
14:                visited[v] ← True
15:                Q.push()
16:            end if
17:        end for
18:    end while
19:    globalInfluence ← 0
20:    for i = 1 to n do
21:        globalInfluence ← pow(p, distances[i])
22:    end for
23:    return globalInfluence
24: end procedure
```

Vist tot l'anterior, passem ara a estudiar el cost de l'algorisme voraç proposat. Podem dividir l'estructura d'aquest en 3 blocs.

- Difusió
- Càlcul d'influència
- Propagació

Tot comença amb la difusió. Inicialment no hi ha cap node propagat, per tant en aquesta part el que es fa és seleccionar quin serà el següent node a afegir al subconjunt de vèrtex influenciats. Per fer tal cosa, es calculen les influències de cadascun dels nodes de la part no influenciada i s'afegeix el que sigui màxim. A continuació, es propaga el subconjunt actual i s'avalua si s'ha aconseguit arribar a la totalitat del graf. En cas que no, es repeteix el procés.

El càlcul de la influència es realitza a partir d'un cerca en amplada, que té un cost de *O(n+m)*. D'altra banda, el cost de la propagació és en el pitjor cas també de O(n+m), ja que l'algorisme està basat en una cerca en amplada però subjecte a una probabilitat, i només en el cas de vèncer la probabilitat de propagació en cada aresta del graf s'arribarà al cost esmentat.

Per tant, en cada iteració de la difusió es realitzen n càlculs d'influència en el pitjor cas, és a dir, $\mathbf{O(n^2 + nm)}$, i també es realitza una execució de l'algorisme de propagació, que triga O(n+m). Com en el pitjor cas es fan n iteracions en la difusió (si només es propaga un node cada vegada), el cost de l'algorisme voraç en la seva totalitat serà en el pitjor escenari de $\mathbf{O(n^3 + n^2m + n^2 + nm)}$.

3.1.2 Algorisme voraç pel model de *Linear Threshold*

L'esquema a seguir per aquest algorisme és el mateix que en el cas del model de difusió IC: partim d'un conjunt inicial buit, comprovem si podem difondre la influència de manera que cobrim tots els nodes del graf i, en cas que no, seleccionem el vèrtex de major influència dels que encara no formen part del subconjunt a partir de què expandim. Repetim aquesta operació fins que el subconjunt generat a partir de cadascuna de les adicions de nodes ens permet expandir la influència a la totalitat de la xarxa. Així, el pseudocodi de l'algorisme (com abans) és el següent:

```
1:  procedure GreedyLT
2:      S ← empty
3:      while (not S.size() == n)  do
4:          u ← node ins V-S with maximum influence
5:          S.add(u)
6:          if S is a solution then
7:              return S
8:          end if
9:      end while
10:     return S
11: end procedure
```

Com es veu, aquest algorisme minimitza el tamany de la solució però no garanteix assolir la més petita de totes.

Un cop més, el gruix computacional de l'algorisme es troba en el càlcul de les influències. Aquest l'hem modelat seguint l'article *Maximizing the spread of influence through a social network* [9]. Som conscients que, tal i com es comenta a l'article de Wei Chen citat anteriorment [7], el model usat per Kempe no és òptim. Tot i això, donades les simplificacions comentades en definir el model de difusió LT, la perspectiva adoptada a [9] resulta d'allò més útil i es mostra com una bona opció.

Així, la influència d'un node es prèn com la suma del nodes a què es pot arribar des d'ell, ponderada per les probabilitats d'arribar-hi (aquest plantejament neix d'una reducció de la influència a l'accessibilitat mitjançant camins amb arestes *live* un cop bloquejades unes o altres d'elles en funció dels seus pesos dins el graf [9]). En conseqüència, podem calcular, per cada node, la seva influència, aplicant un algorisme de Djisktra que compti la distància que el separa de cada node del graf prenent com a distància la distància del node anterior més la probabilitat d'accedir a l'actual a través de l'aresta per on es fa (on aquesta probabilitat és una funció del grau del node i el pes de l'aresta).

Ho fem de la manera següent:

```
1: procedure computeInfluence(src)
2:     Q ← empty queue    ▷ Cua de prioritat inicialment buida, distàncies com claus
3:     Q.push(0,src)                                  ▷ Implementada com una minQueue
4:     distances(n) ← [inf,inf...inf]
5:     visited(n) ← [false,false...false]
6:     distances[src] ← 0
7:     while not Q.empty() do
8:         u ← Q.front().second
9:         Q.pop()
10:        if !visited[u] then
11:            for every v in u.neighbours() do
12:                if distances[v] ≤ distances[u] + distances[u]/degree(v)]) then
13:                    distances[v] = distances[u]+distances[u]/degree(v)
14:                    Q.push(distances[v], v)
15:                end if
16:            end for
17:        end ifvisited[u] = true
18:    end while
19:    globalInfluence ← 0
20:    for i = 1 to n do
21:        globalInfluence += distance[i]
22:    end for
23:    return globalInfluence
24: end procedure
```

Així, calculem la influència de cada node en funció de com de fàcil seria arribar des d'ell a qualsevol altre node si això depengués de la porció del grau del segon que el primer representa.

Com en el cas anterior, l'algorisme queda reduït al càlcul de les influències, la posterior ampliació del subconjunt inicial i la comprovació que aquest sigui una solució vàlida. El cost de l'algorisme dependrà doncs del cost de cadascuna d'aquestes operacions així com de com les combinem. A la secció d'experimentació hem provat diverses versions pel que fa a la comprovació de les solucions (és a dir, la modelització de la difusió) i el que refereix al càlcul de les influènices: hem provat a calcular-les només un cop al principi i guiar la cerca en aquest sentit o, per contra, calcular-les cada cop que hem d'afegir un node (tenint en compte els ja pertanyents al subconjunt) i triar el que la maximitzi. En el cas de la difusió hem implementat optimitzacions donat el fet que V(S+u) = V(V(S)+u). Així, hem comparat una versió en què cada cop comprovem la solució expandint des de zero a partir del subconjunt que tenim a la iteració actual, amb una altra on comprovem la solució expandint a partir dels nodes que hem estat capaços d'influenciar en la iteració anterior més el node a afegir.

Pel que fa al cost de l'algorisme, en funció de quina versió fem servir l'haurem de calcular d'una o altra manera. Sigui com sigui, però, el cost dependrà exclusivament d'un combinació de la difusió i el càlcul d'influències. Així doncs, cal saber primer quin cost tenen aquestes operacions. Com hem vist a la secció dedicada als models

de difusió, la nostra implementació de LT té cost computacional O(n+m). Per la seva banda, el cost del càlcul d'influències (tenint en compte que l'hem implementat mitjançant l'algoritme de Djisktra i usant una cua implementada amb un heap binari és O((n+m)*log n). En funció de quantes crides fem a cadascuna d'aquestes operacions el nostre algorisme tindrà un cost o altre.

En el cas de la primera i la segona versió cridem el càlcul d'influències per cadascun dels nodes, com a molt, n vegades, per la. A més cridem a la difusió com a molt n vegades però, com que cada cop examinem menys nodes perquè n'hi ha més que formen part del subconjunt inicial, no podem considerar aquest un cost quadràtic. Així, tenim quelcom amb cost de l'ordre de O(n) * (O((n+m)*log n))*O(n) + O(n+m)). En conclusió, podem considerar que que el cost d'aquestes versions està completament dominat pel cost del càlcul de les influències i determinar el cost del nostre algorisme com $\boldsymbol{O(n^2*\log n*(n+m))}$. Tpt i que, aparentment, ambdues versions tenen el mateix cost temporal, cal notar que, amb la segona versió, l'algorisme itera moltes menys vegades ja que, cada iteració, el nombre de nodes del subconjunt inicial no s'incrementa només en un (excepte en el cas pitjor). Així, s'arriba abans al cas final en què la mida d'S és igual a la de G.

En el cas de la tercera versió, per contra, només calculem la influència de cada node un cop (amb cost total O(n*(n+m)*log n)), de tal manera que el cost de l'algorisme, encara dominat per aquesta operació donat que només es fan, com a molt, n comprovacions de solució (de cost O(n+m)), és ara aquest mateix: $\boldsymbol{O(n*\log n*(n+m))}$.

3.2 Algorisme de cerca local

Un algorisme de cerca local, és aquell que a partir d'un estat inicial, va visitant els veïns amb la intenció de trobar solucions més bones fins a arribar a un mínim local.

Per tant, és important tenir una solució arbitrària com punt de partida per millorar reiteradament aquesta solució fins a arribar a un punt on la solució ja no millora, és a dir, que la solució ha convergit.

En el nostre cas, utilitzarem 3 sistemes de solució inicial diferents, per a després a l'apartat d'*Experimentació* poder comprovar quines són les diferències tant a nivell de cost de computació com de qualitat de la resposta. Són els següents:

- **Nodes aleatoris:** Seleccionarem un conjunt de nodes aleatoris, el cost serà el de recórrer el conjunt de nodes, per tant, *O(n)*
- **L'algorisme Greedy:** També hem pensat d'utilitzar l'algorisme Greedy implementat a l'apartat 2.1, ja que com hem explicat anteriorment la solució és una aproximació de la solució òptima, per tant, pot tenir rang de millora.
- **Vector amb tots els nodes**: Finalment, utilitzarem com a solució inicial el conjunt de tots els nodes del graf.

També varem proposar utilitzar el *Minimum Dominant Set* com a solució inicial, el *minimum Dominant Set* o *MDS* és el *Dominant Set* de mínima cardinalitat, i un *Dominant Set* és el conjunt nodes on el conjunt més tots els seus veïns és el conjunt total de nodes, és a dir $V = MDS + Neighbors(MDS)$. Finalment hem descartat aquest mètode, ja que si que és una solunció inicial per el model de difusió de *Independent*

Cascade, però no sempre ho és per *Linear Treshold*, sobretot en casos on l'*spreading ratio* és alt.

Un cop tenim una solució, com bé determinen els principis de la cerca local, ara ens interessa buscar solucions òptimes o subòptimes en els veïns de les solucions donades. Per això, és necessari determinar un mètode per determinar quins són els veïns, en el nostre cas no es tracta de res més que eliminar el nombre de nodes de la solució per veure si existeix alguna solució amb una cardinalitat menor, és a dir, aplica'm la tècnica *first to improve*, on la solució que prosseguirà serà la primera per la qual trobem una solució.

Però, és més, també hem determinat una heurística per decidir quins dels nodes serà el que eliminarem, fent així que sigui més probable trobar la solució abans i també arribar a una solució més òptima. L'heurística mencionada consistirà en el següent, s'eliminarà el node on la mitjana de la influència de tots els seus veïns sigui la mínima. Ja que tal i com es comenta en el *paper* següent [8], és una manera eficaç d'eliminar nodes que no són tan útils per la difusió del graf.

$\min_{v \in V}(\text{influence}(N(v)))$

On *influence(x)* és el càlcul de la influència de x sobre tot el graf, i *N(x)* és el conjunt de veïns de x.

En el moment que examinem un veï, és a dir, el conjunt de nodes un cop eliminat el node corresponent seguint l'heurística mencionada anteriorment, és necessari comprovara si és una solució vàlida. En tots dos casos comprovar la solució consisteix en realitzar una propagació i comprovar si el nombre de nodes de la solució obtinguda és igual al nombre de nodes total del graf. El problema el trobem en el model *Independent Cascade*, ja que el fet que la propagació depèn d'una probabilitat, fa que alguns cops la propagació ens determini que no és una solució, quan podria arribar a ser-ho. De totes maneres, hem decidit menysprear aquest error, perquè l'objectiu de la *Cerca Local* és disminuir el conjunt de la solució inicial, per tant, si al comprovar la solució ens retorna fals quan realment és cert, no minimitzarà, però tampoc ens retornarà una solució incorrecta.

Com és lògic el cost d'aquest algoritme dependrà en primera instància del mètode escollit per trobar la solució inicial, i, per altra banda, de quin model de difusió utilitzem, ja que tant el càlcul de les influències d'un node i la propagació (per comprovar si la solució és vàlida) tenen costs diferents.

El cost en el cas pitjor és el següent:

getInitialSolution + getNodesInfluences() + costFinsConvergir(n) * (calculMijanaInfluence (n) + costBucleSolucions (n)*costComprovarSolució(n+m)) =>
O(getInitialSolution + 2*$n^2 + n^3 + n^2 * m + n * m$)

On *getInitialSolution* és el cost d'obtenir la solució inicial. I getNodesInfluences, que és el cost d'obtenir les influències de tots els nodes és O($n^2 + n * m$) en el cas del model *IC* i O($n * logn + n * m$) en el cas de LT.

	getInitialSolution
Random	O(n*propagate)
All nodes	O(1)
Greedy	O(nˆ2*lg n *lgn)

On *propagate* és el cost de fer la propagació que és O(n+m) en la *Independent Cascade*, i O(n^2) en *Linear Treshold*.

Per tant, en el nostre escenari principal, on l'estat inicial serà obtingut mitjançant l'algorisme *Greedy*, el cost final en el cas pitjor serà el següent: $O(n^3 + n^2 * m + n^2 + nm + n^2 + n^3 + n^2 * m)$.

3.3 Algorisme basat en una metaheurística

Una metaheurística és un mètode heurístic d'optimització que es fa servir per resoldre problemes on obtenir una solució òptima fent servir algorismes deterministes clàssics condueix a costos massa elevats.

La idea d'aquesta metaheurística és que en cada pas es millori la situació actual de la solució del problema, basant-se en un procés de cerca global entre totes les possibles solucions.

Tant pel model de *Independent cascade* com pel de *Linear threshold* emprarem la metaheurística de *Simulated Annealing*. El model de funcionament d'aquest algorisme procedeix del procés físic del temperat de metalls, on per aconseguir unes propietats moleculars desitjades (solució), és necessari controlar el procés de refredament.

En *Simulated Annealing*, l'objectiu és el de trobar una bona aproximació de la solució òptima per a un espai de cerca molt gran. En aquest algorisme, la solució en un cert punt es considera com a un estat, i és mitjançant transformacions que es pretén arribar a un estat òptim.

3.3.1 Procediment bàsic de *Simulated Annealing*

L'algorisme de *Simulated annealing* es fonamenta en el concepte de reducció de temperatura, per tant, és necessari tenir una funció que permeti calcular la quantitat total d'energia d'un estat. A més, s'ha de definir abans de començar quins són els paràmetres de l'execució; temperatura inicial, nombre d'iteracions i factor de refredament.

Un cop vists aquests conceptes bàsics, procedim amb l'algorisme com a tal. El procés de refredament es realitza iterativament, i cadascuna d'aquestes iteracions es pot descompondre en els següents passos.

1. **Selecció del moviment**
   Es realitza un moviment, ja sigui de forma aleatòria o determinista, per tal de crear un potencial nou estat.

2. **Avaluació de la solució**
   S'avalua el potencial nou estat generat fent servir la funció d'energia. Es farà la transició a aquest nou estat si el resultat obtingut amb dita funció és millor que el de l'estat actual.

3. **Actualització de la temperatura**
   Cada iteració (o cada cert nombre d'iteracions) es va reduint la temperatura del sistema mitjançant un factor de refredament.

La condició de parada d'aquest algorisme serà quan la temperatura quedi per sota d'un cert llindar. Veiem que la clau d'aquest algorisme es trobarà en la seva capacitat per explorar noves solucions, i no quedar estancar-se a extrems relatius locals.

3.3.2 Algorisme basat en SA per la maximització de la influència

```
1: procedure SA(n)(10)
2:     Partim d'una temperatura inicial
3:     while temperatura > 0 do
4:         for i = 1 to n do
5:             //sigui 'n' un nombre fix d'iteracions
6:             Enou ← GeneraSuccessorAleatori(Eactual)
7:             ΔE ← f(Eactual) - f(Enou)
8:             if ΔE > 0 then
9:                 Eactual← Enou
10:            else
11:                amb probabilitat e^(ΔE/T): Eactual ← Enou
12:            end if
13:        end for
14:        Disminuim la temperatura
15:    end while
16:    return Eactual
17: end procedure
```

Algorisme basat en [10]

Tal i com es pot veure a l'algoritme partirem d'una solució inicial per tractar d'optimitzar-la amb el *Simulated Annealing*. El nostre estat inicial partirà de tres tipus de solucions diferents.

- **Nodes aleatoris:** Seleccionarem un conjunt de nodes aleatoris, el cost serà el de recórrer el conjunt de nodes, per tant, *O(n)*
- **L'algorisme Greedy:** També hem pensat d'utilitzar l'algorisme Greedy implementat a l'apartat 2.1, ja que com hem explicat anteriorment la solució és una aproximació del 63 % a la solució òptima, per tant, encara té rang de millora.
- **Vector amb tots els nodes**: Finalment, utilitzarem com a solució inicial el conjunt de tots els nodes del graf.

Aleshores, definim una temperatura inicial, com disminueix la temperatura i un nombre 'n' d'iteracions. També hem de definir la variació de l'energia donat que en aquest cas sempre variarà de la mateixa manera ja que en el cas de que la solució sigui millor haurem tret només un node. Aleshores caldrà definir aquesta constant. Podem entreveure que un dels desavantatges d'aquest algoritme és que hi ha molts paràmetres amb els quals s'ha d'experimentar per justificar com s'inicialitzen.

Un cop tenim els paràmetres inicialitzats l'algoritme genera un estat aleatori dins de l'espai de solucions. Si la solució és millor haurem eliminat un node i en canvi, si la solució és pitjor haurem tret un node. D'aquesta manera sempre que el nou estat aleatori generat sigui millor el substituirem per l'estat actual. D'altra banda, si la solució empitjora substituirem el nostre estat en funció de la probabilitat (línia 11).

El cost de la solució inicial depenent de si la generem aleatòriament o amb l'algorisme Greedy serà $\mathbf{O(n)}$ o $\mathbf{O(n * logn * (n + m))}$ respectivament pel cas de *Linear Threshold*, o bé $\mathbf{O(n^3 + n^2m + n^2 + nm)}$. L'altra part de l'algorisme, com que la temperatura i nombre d'iteracions son constants, la complexitat del LT està determinada per generar l'espai de solucions a cada iteració. En el pitjor dels casos, tindrem tots els nodes del graf al subconjunt i aleshores caldrà veure per cada un d'ells si obtenim una solució més òptima eliminant-los. Per tant, per cada un dels nodes caldrà fer la propagació pel graf sense el propi node eliminat. La propagació té $\mathbf{O(n + m)}$ per tant el cost total de l'algorisme sense tenir en compte la solució inicial és $\mathbf{O(n(n + m))}$. Si sí que tenim en compte la solució inicial pel greedy, obtindrem complexitat per al LT $\mathbf{O(n * logn * (n + m))}$ i per a la solució aleatòria obtindrem $\mathbf{On(n + m)}$. Altrament, el IC té una complexitat de $\mathbf{O(n^3 + n^2m + n^2 + nm)}$ en el cas del greedy com a solució inicial i $\mathbf{On(n + m)}$ per a la solució inicial aleatòria.

4 Experimentació

Per garantitzar comparatives equitatives entre els diferents tipus d'algorismes i models de difusió, realitzarem tots els experiments amb el mateix ordinador, el qual te les següents especificacions.

- Processador AMD Ryzen 5 4600H with Radeon Graphics 3.00 GHz
- RAM instalada: 32,0 GB (31,4 GB usable)
- 6 Cores, 12 Threads, TDP = 45W,
- Caché L2 total 3MB
- Caché L3 total 8MB

A més, per garantitzar una idea general, els experiments pel model de IC els realitzarem uns quants cops (entre 3 i 5 depenent de la magnitud del graf) i ens quedarem amb la mitjana, per poder tenir resultats més globals.

Tindrem en compte tres factors, el nombre d'iteracións, el temps d'execució i la quantitat de nodes a la solució final (nodes finals menys nodes inicials).

4.1 Grafs utilitzats

Per comprovar els diferents algorismes utilitzarem grafs genèrics per poder realitzar comparacions entre ells i amb altres algorísmes del *benchmarck* que se'ns ha proporcionat

Network			MTS		MDG		BRKGA		
	Vertices	Edges	Result	Time	Result	Time	Best	Average	Avg. Time
Karate	62	159	7	< 0.01	8	< 0.01	**6**	6.0	< 0.01
Football	115	613	29	< 0.01	31	< 0.01	**22**	22.3	29.6
Dolphins	34	78	**3**	< 0.01	**3**	< 0.01	**3**	3.0	< 0.01
Jazz	198	2742	33	< 0.01	31	< 0.01	**20**	20.4	10.0
CA-AstroPh	18772	198050	2228	45.54	1638	0.05	**1500**	1508.6	182.7
CA-GrQc	5242	14484	992	3.98	1033	0.01	**942**	947.4	90.0
CA-HepPh	12008	118489	1832	20.5	1529	0.03	**1394**	1402.6	115.2
CA-HepTh	9877	25973	1388	15.1	1388	0.01	**1307**	1312.	76.4
CA-CondMat	23133	93439	3156	73.76	2938	0.05	**2760**	2777.6	216.4
Email-Enron	36692	183831	3143	219.48	2881	0.1	**2745**	2759.9	355.6
ncstrlwg2	6396	15872	1080	5.7	1108	0.01	**1027**	1045.7	64.3
actors-data	10042	145682	1250	11.93	1014	0.02	**937**	943.4	95.6
ego-facebook	4039	88234	580	1.87	528	0.01	**493**	501.1	65.2
socfb-Brandeis99	3898	137567	421	1.94	395	0.02	**338**	358.0	80.8
socfb-nips-ego	2888	2981	**10**	2.18	**10**	< 0.01	**10**	10.0	< 0.01

Figure 1: Benchmark

1. **Graf Karate o Graf de Zachary.**

   Aquest grau descriu l'estructura social d'un club de karate a la dècada de 1970 i s'ha utilitzat com a exemple comú en la teoria de grafs i en l'anàlisi de xarxes socials. El grau consta de 34 nodes que representen als membres del club i les arestes,78 , que connecten als nodes representen les relacions d'amistat entre ells.

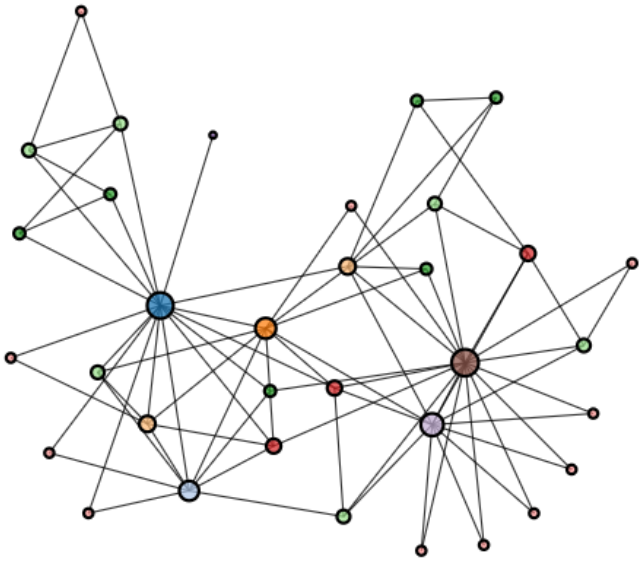

Figure 2: graf karate

2. **Graf Dolphins** Fa referència al graf de la xarxa social dels dofins de nas de botella habiten a Doubtful Sound, Nova Zelanda. El grau consta de 62 nodes que representen als dofins i les arestes,159 , que connecten als nodes representen les relacions de co-ocurrència entre els dofins en grups d'alimentació.

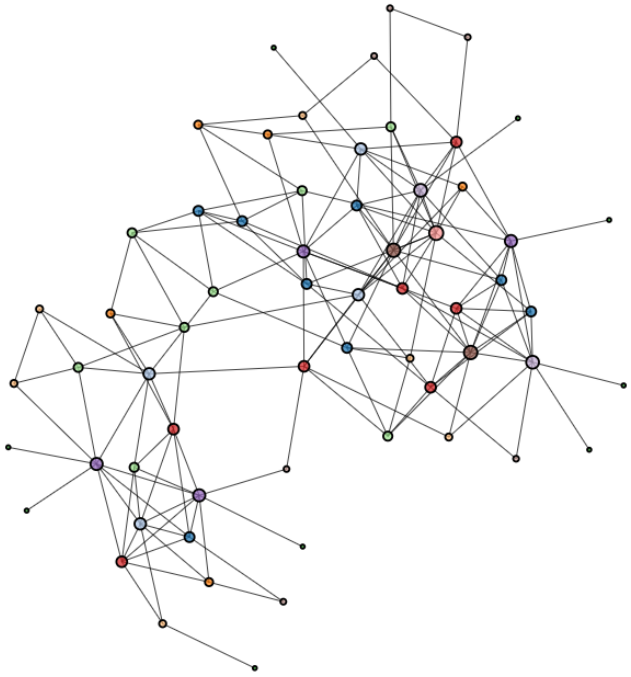

Figure 3: graf dolphins

3. **Jazz** Fa referència a la xarxa de col·laboracions entre músics de jazz. Aquest grau es construeix connectant dos músics si han tocat junts en alguna gravació. El graf Jazz és interessant perquè mostra una estructura de xarxa altament connectada, on molts músics estan connectats entre si a través d'un petit nombre d'intermediaris.El grau Jazz consta de 198 nodes (músics) i 2.742 arestes (col·laboracions).

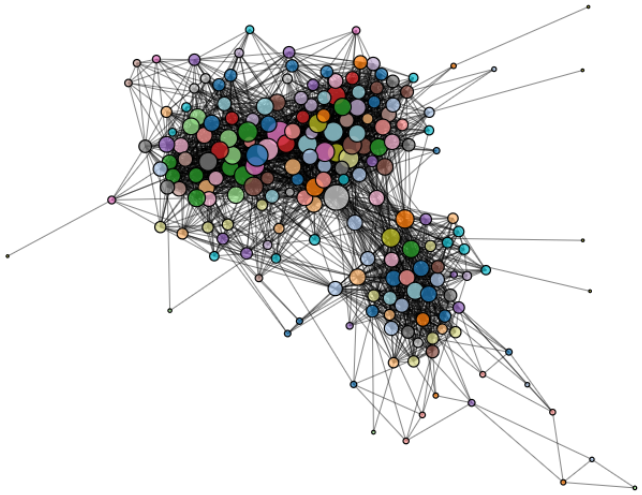

Figure 4: graf jazz

4. **socfb-nips-ego o Graf Ego Network** És una subxarxa de la xarxa social de Facebook, on es van recopilar dades d'estudiants de la Universitat de Stanford durant un període de tres anys. El grau representa les connexions d'un usuari específic a Facebook (l'ego) i els seus amics directes (els "alter"). El grau conté un total de 4039 nodes (amb l'ego inclòs) i 88234 arestes que representen les amistats entre els usuaris de la xarxa social.

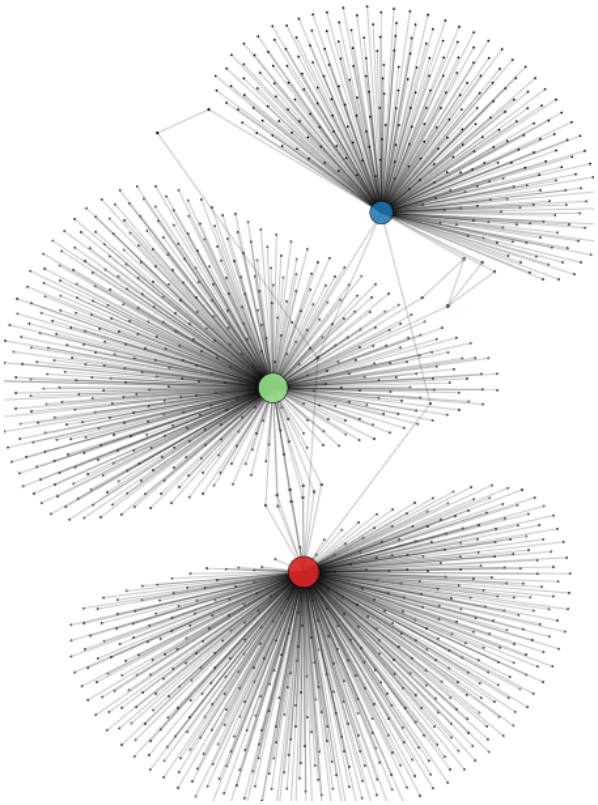

Figure 5: Graf socfb-nips-ego

4.2 Model de difusió

Com hem comentat en seccions anteriors, el model de difusió resulta determinant pel nostre algoritme doncs és el que ens permet comprovar si un subconjunt de nodes és solució pel problema. També es clau (sobretot tenint en compte que usarem l'algorisme greedy per generar solucions inicials pels algorsimes posteriors) examinar quina és la forma òptima en què combinar la difusió amb el càlcul de les influències, com comentàvem al final de la secció 3.1. En conseqüència, hem desenvolupat tres versions diferents: una on es calculen la influència de tots els nodes cada cop que volem afegir-ne un al subconjunt inicial, una altra on es fa això mateix però es comprova l'expansió a partir dels nodes que el subconjunt anterior ha estat capaç d'influenciar (a més del nou afegit), i una última on només calculem la influència dels nodes un cop (al principi) i mantenim la optimització pel que fa a la difusió implementada a la versió dos.

Hem provat l'eficiència de cadascuna per decidir quina utilitzar pels nostres algorismes. Els resultats aquí mostrats són els de l'experimentació amb el model LT. Els del model IC estan recollits a l'apèndix A i en comentarem les conclusions al final d'aquesta secció.

4.2.1 Versió 1

Grafs	p = 0.4			p = 0.6			p = 0.8		
	temps	**iters**	**nodes**	**temps**	**iters**	**nodes**	**temps**	**iters**	**nodes**
Dolphins	0,008837	11	3	0,00802434	12	8	0,011036	30	13
Jazz	0,195902	25	19	0,857147	182	44	0,0555409	188	102
nips-ego	0,71101	9	9	0,815219	10	10	0,80803	10	10

4.2.2 Versió 2

Grafs	p = 0.4			p = 0.6			p = 0.8		
	temps	**iters**	**nodes**	**temps**	**iters**	**nodes**	**temps**	**iters**	**nodes**
Dolphins	0,00617500	3	3	0,007483	8	8	0,0101018	13	13
Jazz	0,177955	19	19	0,436845	44	44	0,47043	102	102
nips-ego	0,702682	9	9	0,797389	10	10	0,77579	10	10

4.2.3 Versió 3

Grafs	p = 0.4			p = 0.6			p = 0.8		
	temps	**iters**	**nodes**	**temps**	**iters**	**nodes**	**temps**	**iters**	**nodes**
Dolphins	0,002142	3	3	0,00178615	7	7	0,0022301	14	14
Jazz	0,0400642	19	19	489.226	44	44	0,0661104	100	100
nips-ego	0,109437	9	9	0,104061	10	10	0,117413	10	10

Com es pot observar, hi ha una certa milloria entre les versions 1 i 2. Si bé potser en el cas dels grafs amb què estem treballant aquesta optimització no es fa tan palesa, com hem vist al final de la secció dedicada a l'algorisme *golafre*, la diferència en el cost computacional entre les versions 1 i 2 no és menyspreable, especialment a mesura que escala el tamany del problema.

Pel que fa a la tercera versió, resulta evident que millora molt considerablement el temps d'execució i, si bé el càlcul de la influència no és tan exhaustiu, no sembla massa perjudicial pel que fa al tamany de la solució obtinguda (quasi sempre és, de fet, la mateixa).

En vista d'aquests resultats, per les diferents fases de l'experimentació hem decidit emprar la tercera versió del algorisme greedy pel model LT.

Pel que fa al model IC, la versió triada ha estat la 2 ja que representava una millora notable pel que fa a temps d'execució respecte de la versió 1 (sobretot a mesura que els grafs amb què ha de treballar creixen), mentre que la 3 no suposava una gran milloria per la qual cosa hem optat per la que ens ha semblat més exhaustiva. Aquestes mesures es poden consultar al primer annex del present article.

4.3 Algorisme Greedy

4.3.1 IC

Per aquesta experimentació hem triat tres probabilitats de propagació diferents (0.1, 0.4 i 0.8) de forma que puguem mesurar el rendiment de l'algorisme segons augmenta la dificultat per propagar nodes (0.1 representa una gran dificultat i 0.8 una dificultat pro baixa). A més, hem fet servir diversos grafs per comprovar si l'algorisme és escalable (malgrat articles previs indiquen que no ho és [7]) i veure com evoluciona el temps d'execució.

Grafs	**p = 0.1**			**p = 0.4**			**p = 0.8**		
	temps	**iters**	**nodes**	**temps**	**iters**	**nodes**	**temps**	**iters**	**nodes**
Dolphins	0.0130433	12	12	0.0034925	3	3	0,00155092	2	2
Karate	0,017612	14	14	0.005982	5	5	0,006572	3	3
Football	0.020331	7	7	0.0060632	1	1	00.65025	1	1
Jazz	0.035108	10	10	0.029042	3	3	0.22855	2	2
nips-ego	5.2418	62	62	1,64995	16	16	1.07152	6	6

Examinant els resultats obtinguts, hi ha diversos aspectes que destaquen. En primer lloc resulta evident que la dificultat per influenciar nodes veïns afecta notòriament el rendiment del programa, la qual cosa és coherent i esperable donat la natura del problema. Així, contra més baixa la probabilitat d'influenciar, més alts són tant el temps d'execució com el nombre de nodes que formen part del subconjunt solució mínim trobat.

Pel que fa a l'escalabilitat de l'algorisme cal dir que, si bé segueix trobant conjunts solució pel problema (subòptims), l'augment del tamany del graf el fa alentir-se molt noablement. Així, quan executem el programa sobre el graf nips-ego, amb 2888 nodes, el temps d'execució multiplica per 500 aproximadament la marca obtiguda amb el graf Dolphins (de tamany 98, tan sols 30 vegades més petit que nips-ego). Podem, a més, afirmar que aquesta pujada depèn del nombre de nodes però no del nombre d'arestes ja que el graf Jazz té un nombre similar al de nips-eco. Així doncs, el problema de l'escalabilitat de l'algorisme *greedy* en el cas del model IC passa pel tamany de l'entrada però no per la seva densitat. Aquest fet també és coherent amb els models de difusió i la representació que n'hem fet, així com amb la tria de versions que hem adoptat.

4.3.2 LT

En aquest cas, com abans, hem triat també tres ràtios de propagació diferents per veure com hi reacciona l'algorisme. Ara, però, no hem triat valors tan extrems sino que hem optat per una aproximació més realista en relació amb les necessitats d'aplicació de l'algorisme a qüestions de *marketing*. Així doncs, hem fet servir com a valors 0.4, 0.6 i 0.8. Els resultats són aquests:

Grafs	r = 0.4			r = 0.6			r = 0.8		
	temps	**iters**	**nodes**	**temps**	**iters**	**nodes**	**temps**	**iters**	**nodes**
Dolphins	0,00208363	3	3	0,00454561	7	7	0,00845092	14	14
Karate	0,00356638	5	5	0,0102997	17	17	0,0181135	31	31
Football	0,0143531	20	20	0,03027	49	49	0,0460462	75	75
Jazz	0,0161539	19	19	0,0330787	44	44	0,0711607	100	100
nips-ego	0,0462312	9	9	0,0470753	10	10	0,0460178	10	10

Com en el cas del model IC, l'augment de la dificultat (en aquest cas representat per l'augment del ràtio mínim d'influència) sembla afectar proporcionalment al temps d'execució i a la mida de la solució trobada. Un cop més, això resulta coherent amb el problema que tractem i sembla un símptoma d'adequació a aquest.

Pel que fa a l'escalabilitat aquest model rendeix evidentment millor que l'anterior. Si bé el temps d'execució augmenta segons ho fa el tamany de l'entrada, ara ho fa molt més lentament i de forma coherent amb l'escalada d'aquest.

4.4 Cerca Local

Per l'experimentació de l'algorisme implementat mitjançant *cerca local*, realitzarem proves amb els tres sistemes d'obtenció de solució inicial.

4.4.1 Independent Cascade

Pel model de difusió *IC* és molt important tenir en compte la probabilitat de difusió, ja que el fet de que depengui d'una probabilitat pot provocar que una difusió dos cops al mateix graf dona resultats completament diferent, per això realitzarem l'experiment 5 cops i obtendrem els resultats en mitjana. A més, en casos on la probabilitat sigui baixa, serà molt complicat minimitzar la solució i per altra banda, on la probabilitat sigui alta les solucions minimitzaran molt fàcilment, fins a tal punt que, si la probabilitat és 1, les solucions seran d'un sol node, per tant, per realitzarem experiments amb una probabilitat relativament baixa (0.4) i una relativament alta alta (0.8).

Probabilitat = 0.4

	Random Starting subset			**All-True Starting subset**		
	temps	iteracions	start vs fin nodes	temps	iteracions	start vs fin nodes
DOLPHIN	0,058538	26	32-28	0,079185	37	34 - 17
JAZZ	0.232938	84	114-112	0,196857	102	197-191
NIPS-EGO	7,787950	1913	2887 - 2883	4,708850	1498	2888-2883

Com hem comentat abans, les difusions seguint un model IC venen condicionades per la probabilitat, al tenir tan sols d'un 40 percent, al model li costa trobar solucions millors, per això, la diferencia del subset inicial i final de nodes és petita.

	n	m	Greedy Output Starting subset		
			temps	iteracions	start vs fin nodes
DOLPHINS	34	78	0,00656	2	4-4
JAZZ	198	2742	0.819789	2	4-4
NIPS-EGO	2888	2981	3,906030	7	15-15

Per altra banda aquí, el fet de tenir una probabilitat gran

En el cas d'utilitzar el *Greedy* com a solució inicial, podem veure que a n'aquests grafs no ens minimitza la solució, això és a causa de que l'implementació ja ens dona solucions prou fitades, sobretot per aquests grafs que són relativament petits. Tot i així, podem veure que pel graf *nips-ego*, el temps d'execució és menor, ja que al no tenir un conjunt tan gran de nodes convergeix abans a una solució òptima.

Probabilitat = 0.8

	Random Starting subset			All-True Starting subset		
	temps	**iters**	**start vs final nodes**	**temps**	**iters**	**start vs final nodes**
Dolphins	0,005642	3	6-5	0,056038	26	34 - 16
Jazz	0,0421453	2	3- 3	0,438643	208	198 - 159
nips-ego	6,035340	1590	2881-2877	5,808810	3051	2888-2881

Al tenir una probabilitat tan alta, podem comprovar com les solucions inicials del *mètode ràndom* ja són subsets bastant petits, per això, la minimització és petita, fins a l'extrem on es produeix cap canvi(Dolphins). Per altra banda, el fet de dependre de probabilitas provoca que en algun cas no es trobi la millor solució possible, per això, utilitzant el mètode d'*All-True*, al tenir un conjunt de nodes tan gran no minimitza fins als millor minims locals.

Al Greedy ens trobem exacatament la mateixa situació que a l'executar *IC* amb una probabilitat del 40 percent.

	Greedy Output Starting subset		
	temps	**iters**	**start - final nodes**
Dolphins	0,003786	2	2 -2
Jazz	0,0782324	2	2 - 2
nips-ego	3,267700	3	6 -6

4.4.2 Linear Treshold

Per altra banda, els experiments seguint aquest mètode no varien si repetim l'experiment, per tant, aquí podrem obtenir resultats que ens donaran més joc a realitzar comparatives i estudiar quines opcions són idònies. Realitzarem l'experiment pels valors de 0.4 i 0.7

Spreading ratio = 0.4

Comparant els mètodes de *Random Starting* i *All-True* podem veure, que a diferència del model *IC*, aquí si que hi trobem diferències. Les solucions són millors utilitzant tots els nodes del conjunt com a solució inicial, tot i això, el temps d'execució també són més alts.

Això té sentit, ja que tenir una solució inicial més petita, no garanteix que la solució final sigui l'òptima, encara més, al tenir tots els nodes a true, ens dona més opcions a eliminar nodes amb una menor influència per així evitar tenir nodes poc útils a la solució, com si passa amb el Random. És a dir, al començar amb una solució *Random*, tendrem el dilema d'acabar quedant estancats a un mínim local, per altra mà, començant amb tots els nodes ens dona l'opció d'arribar a un màxim absolut, o almenys, a una aproximació. Això queda reflectit al nombre d'iteracions. Per tant, podem obtenir una millor solució amb el *All-True* mètode, això si, sacrificant temps d'execució.

	Random Starting subset			**All-True Starting subset**		
	temps	iteracions	start vs fin nodes	temps	iteracions	start vs fin nodes
DOLPHINS	0,030745	7	10 - 8	0,09914	47	34 - 3
JAZZ	0,281657	106	42 - 29	0,75762	326	198-20
NIPS-EGO	24,249	8477	1363-881	78,6442	8477	2888-553

Utilizant el Greedy, ens ocorr molts cops que la solució del Greedy ja no te marge a millorar, però no sempre, existeixen casos on si que és útil aplicar l'algorisme de *Local Search* després del *Greedy*, per exemple en el graf de Jazz, ja que a més, com que les solucions inicials són petites, el temps d'execució no sol suposar un problema.

	Greedy Output Starting subset		
	temps	**iters**	**start - final nodes**
Dolphins	0,0045686	2	3 -3
Jazz	0,0815615	22	19 - 14
nips-ego	0,140524	4	9-9

Spreading ratio = 0.8

Al tenir un speading ratio relativament alt, la 'resistència' d'un node perquè propagui és major, per tant és lògic que les solucions finals contenguin un major nombre de nodes.

	Random Starting subset			All-True Starting subset		
	temps	**iters**	**start vs final nodes**	**temps**	**iters**	**start vs final nodes**
Dolphins	0,103030	48	32-21	0,119433	50	34-25
Jazz	1,060310	464	196 - 125	1,19958	508	198-117
nips-ego	53,7965	22862	2320-997	160,307	73335	2888-1194

Però aquí, ens trobem una situació completamnet diferent a la que ens hem trobat quan utilitzavem el mateix experiment per spreading ratio a 0'4. Aquí tant el mètode Random com All-True ens donen resultats molt semblants, això pot ser causa del fet de que sigui més complicat propagar, ja que no hi ha tants de nodes que no són importants i sigui intrascandent eliminarlos, per tant, tenir una solució i optimitzar-la, serà molt semblant a arribar a una solució a partir del total, tot això condicionat també en part per l'aletorietat de obtenir una solució a l'atzar.

	Greedy Output Starting subset		
	temps	**iters**	**start - final nodes**
Dolphins	0,0170552	7	14 - 11
Jazz	0,382546	142	100-93
nips-ego	0,138271	5	10 - 9

Per tancar el capítol de la *Cerca Local*, veiem que el amb spreadings ratios alts si que existeix un petit marge a millora, ja que la solució amb *Greedy* ens dona una solució bastant bona, però amb la *Cerca Local* podem acabar d'optimitzar la solució.

En resumen, la millor opció és agafar l'output del *Greedy* com a solució inicial, tant a nivell de temps d'execució com a qualitat de la solució final. En grafs petits, agafar tots els nodes és una opció vàlida a nivell de qualitat de la solució final, però no és tan eficient si contemplam el temps d'execució.

4.5 Metaheurística

Per a l'experimentació del SA els valors que s'ha triat presentar son els que s'han cregut més representatius per a les difèrencies dels paràmetres i els seus respectius resultats. D'aquesta manera, les següents taules son un resum de moltes proves que s'han dut a terme.

4.5.1 LT

Per a mostrar l'experimentació dels paràmetres hem triat el graf Jazz ja que és un graf amb un tamany mitjà (198 nodes) però amb bastantes arestes (2.7k arestes). Un cop testejats com varien els paràmetres triarem la combinació més òptima i l'extrapolarem per experimentar amb la resta de grafs. Cal tenir en compte que les execucions de temps son costoses ja que per garantir l'aletorietat en l'espai de solucions hem de generar un conjunt de solucions a cada iteració. Això és perquè hem de trobar els nodes que es poden eliminar de l'estat actual que generen un nou estat solució. D'altra banda, tots els nodes que no estan propagats seran candidats a una nova solució pitjor.

	T = 100							
	It = 20				**It = 50**			
	ΔE = 25		**ΔE = 75**		**ΔE = 25**		**ΔE = 75**	
	Nodes	**Temps**	**Nodes**	**Temps**	**Nodes**	**Temps**	**Nodes**	**Temps**
Aleatori	36	33,2862	31	26,9058	30	112.483	35	62
Greedy Output	32	30,1387	25	20,2358	31	118.755	26	34,2596

	T = 50							
	It = 20				**It = 50**			
	ΔE = 25		**ΔE = 75**		**ΔE = 25**		**ΔE = 75**	
	Nodes	**Temps**	**Nodes**	**Temps**	**Nodes**	**Temps**	**Nodes**	**Temps**
Aleatori	37	64,9151	27	44,4311	34	183,327	29	148,29
Greedy Output	33	65,3615	28	47,6387	34	178,02	31	141,123

Analitzant els resultats, podem veure que tenim millors resultats per a "T = 100" que per a "T = 50" en quan a nodes disminuïts. Si més no, veiem que és molt més costós en temps. En quan a les iteracions i la variació d'energia, observem que pujar-les de 20 a 50 i de 25 a 75 respectivament no té un gran impacte en la solució en ambdós casos, però el temps empitjora molt. Finalment, observem com el Greedy com a solució inicial dona un millor resultat que la generació aleatòria de nodes com podríem esperar. Això és degut a que quan més nodes tinguem inicialment, més cops haurem de propagar per cada un d'ells per comprovar si pot estar dins de l'espai de solucions el graf generat amb l'eliminació del propi node.

Grafs a testejar	Resultat	
	Temps (segons)	Nodes
Karate	0,4395979	8
Football	5,98821	28
Dolphins	0,0890189	3
Jazz	20,2358	25
nips-ego	4,55002	22
CA_GrQc	2454.72	1001

Network			MTS		MDG		BRKGA		
	Vertices	Edges	Result	Time	Result	Time	Best	Average	Avg. Time
Karate	62	159	7	< 0.01	8	< 0.01	**6**	6.0	< 0.01
Football	115	613	29	< 0.01	31	< 0.01	**22**	22.3	29.6
Dolphins	34	78	**3**	< 0.01	**3**	< 0.01	**3**	3.0	< 0.01
Jazz	198	2742	33	< 0.01	31	< 0.01	**20**	20.4	10.0
CA-AstroPh	18772	198050	2228	45.54	1638	0.05	**1500**	1508.6	182.7
CA-GrQc	5242	14484	992	3.98	1033	0.01	**942**	947.4	90.0
CA-HepPh	12008	118489	1832	20.5	1529	0.03	**1394**	1402.6	115.2
CA-HepTh	9877	25973	1388	15.1	1388	0.01	**1307**	1312.	76.4
CA-CondMat	23133	93439	3156	73.76	2938	0.05	**2760**	2777.6	216.4
Email-Enron	36692	183831	3143	219.48	2881	0.1	**2745**	2759.9	355.6
ncstrlwg2	6396	15872	1080	5.7	1108	0.01	**1027**	1045.7	64.3
actors-data	10042	145682	1250	11.93	1014	0.02	**937**	943.4	95.6
ego-facebook	4039	88234	580	1.87	528	0.01	**493**	501.1	65.2
socfb-Brandeis99	3898	137567	421	1.94	395	0.02	**338**	358.0	80.8
socfb-nips-ego	2888	2981	**10**	2.18	**10**	< 0.01	**10**	10.0	< 0.01

Figure 6: Benchmark

Finalment, hem analitzat l'algoritme amb els paràmetres que produïen millors resultats. És a dir, T = 100, It = 20, Δ E = 75 i solució inicial Greedy. Aquesta casella proporciona la millor combinació tant en temps com en nodes minimitzats. Analitzant el seu rendiment, podem concloure que en grafs petits (Karate,Football i Dolphins) s'aproxima bastant a la solució òptima tardant més que MTS i MDG però considerablement menys que BRKGA. Per altra banda, la qualitat de la solució en alguns casos com el de Jazz o Football minimitza més que MTS i MDG amb més cost temporal però menys que BRKGA ja que entre altres factors, té més cost temporal. També cal comentar que en el cas del nips-ego, com que en l'algoritme Greedy (que és la solució inicial) ja troba el conjunt de vèrtexs òptim, el SA empitjora la solució ja que no troba nodes per treure i aleshores afegeix nodes de més. En grafs molt grans com CA_GrQc té un cost temporal molt elevat condicionada per la naturalesa del propi algoritme. Si més no, arriba a trobar una solució de vèrtexs més òptima que MDG.

En conclusió, veiem que temporalment en grafs petits és més costos MTS i MDG però considerablement menys que BRKGA. En canvi en grafs més grans temporalment obtenim resultats semblants als de BRKGA. En quant a la minimització de vèrtexs, en la majoria de casos trobem un subconjunt menor que MTS o MDG tant en grafs grans com grafs petits. Es podria considerar no generar l'espai de solucions cada cop que iterem, sinó generar un node aleatori i aleshores comprovar en el cas que formi part del subconjunt propagat, si treure'l dona un subconjunt solució. Tot i així aquesta opció s'ha descartat perquè tot i que el temps seria menor, estaríem esbiaixant l'aleatorietat de l'espai de solucions i s'empitjoraria el rendiment de l'algoritme. Per tant, els resultats son els esperats ja que no podem garantir la solució òptima amb el nostre algoritme ni tampoc podem esperar una gran optimització temporal donat que, generar l'espai de solucions és costós com s'ha comentat reiteradament.

4.5.2 IC

Procedim ara amb l'experimentació de *Simulated Annealing* pel model de *Independent Cascade*. Com ja s'ha mencionat anteriorment, aquest és un model complicat d'experimentar amb ell, ja que al no ser determinista fa que testejar mètriques com hem fet amb el model de *Linear Threshold* sigui complicat, a la vegada que poc precís. Per tant, ens hem limitat a testejar els mateixos escenaris que hem emprat per l'algorisme voraç i per la cerca local, fent servir les mètriques que ens han proporcionat els millors resultats en el model anterior.

Igual que en les execucions dels altres algorismes d'aquest model, cadascun dels escenaris mostrats a la taula a continuació són resultat de la mitjana de cinc execucions consecutives en contexts idèntics.

Veiem que comparat amb la cerca local i el greedy, els resultats obtinguts en els escenaris descrits anteriorment són lleugerament millors que amb els altres dos mètodes en quant a la mida del subset resultant, però no en quant a temps d'execució ni iteracions, especialment comparat amb l'algorisme *golafre*.

Grafs	**p = 0.1**			**p = 0.4**			**p = 0.8**		
	temps	**iters**	**nodes**	**temps**	**iters**	**nodes**	**temps**	**iters**	**nodes**
Dolphins	0,0772665	7,00	7,00	0,0424169	2,2	2,20	0,0414739	1.666	1.6666
Karate	0,1782580	13,00	13,00	0,1176710	5,00	5,00	0,0643665	2,00	2,00
Football	0,1549440	7,00	7,00	0,1165100	1,25	1,25	0,0407805	1,00	1,00
Jazz	1,0615400	10,00	10,00	0,5569640	2,75	2,75	0,2780570	1,75	1,75
nips-ego	14,16290	56,00	56,00	5,9372700	16,00	16,00	2,4446400	4,60	4,60

Veiem a continuació una taula amb l'execució del *Simulated Annealing* pel model de IC amb els mateixos grafs del *benchmark* que en LT. Els resultats són força bons, però igual que en el cas de *Linear Threshold* l'algorisme no escala gens bé quan augmentem la mida dels grafs, segurament degut a que el conjunt inicial es calcula a partir d'una execució de l'algorisme voraç, a més de les limitacions de *hardware* que de la màquina que hem fet servir.

Grafs a testejar	**Resultat**	
	Temps (segons)	**Nodes**
Dolphins	0,0562343	2
Karate	0,0916393	5
Football	0,0405361	1
Jazz	0,286177	2,666
nips-ego	4,28382	12
CA_GrQc	397,778	362

Com podem veure, si comparam l'algorisme de *Cerca Local* amb el *Metaherusitic*, podem veure com aquest segon ens dona millors resultats, segurament això sigui a causa que estem arribant a minims absoluts o almenys a una aproximació, ja que el fet de poder afegir nodes quan no troba una solució, ens permet no quedar estancats en minims locals o relatius com fa la *Cerca Local*.

5 Conclusió

Hem dissenyat i sotmès a experimentació tres algorismes diferents que ens permeten abordar el problema de *Target Set Selection* (TSS) tant quan modelem l'expansió d'influència dins el graf seguint el model *Independent Cascade* (IC) com quan ho fem amb el model *Linear Threshold* (LT).

Hem vist com, partint d'un resultat subòptim generat per l'algorisme greedy que, malgrat les seves limitacions, rendeix molt bé (malgrat no poder afrontar grafs de tres milions de nodes), els algorismes desenvolupats a continuació ens permetien millorar les solucions trobades. Hem comprovat com la cerca local, a través de la investigació sobre les solucions properes a aquella amb què comencem, permet optimitzar els resultats que proposem pel problema, aproximació molt adequada donat que el projecte ens instava a començar dissenyant un algorisme golafre que, necessàriament, havia de ser subòptim i, per tant, no satisfer completament la solució que la naturalesa del problema demana. Per últim, hem vist com, en millorar la cerca local convertint-la en un algorisme metaheurístic que permet explorar diferents combinacions d'operadors, la depuració de la solució inicial esdevé un procés molt més eficient pel que fa a la relació entre la qualitat de la solució i el temps emprat per depurar-la.

Cap mencionar aquí també, el fet palès que fer ús de LT o d'IC per representar la difusió d'influència marca una gran diferència. Si bé per la construcció dels algorismes no suposa una diferència (i així es pretenia que fos), l'experimentació deixa clar que, mentre LT és un model de representació estable que permet extreure conclusions fonamentades, la natura probabilística del model IC dificulta notòriament la sistematització dels resultats per la possibilitat d'obtenir-ne de tipus molt diversos sota condicions similars. Tot i així, hem estat capaços d'obtenir resultats experimentals que ens permetin comparar el rendiment dels diferents algorismes quan operen amb aquest model i constatar la progressió en la qualitat (a costa, això si, d'un major temps d'execució) que comentàvem anteriorment.

A Annex 1: Comparativa de versions pel model IC

Tal com i s'ha mencionat al llarg d'aquest treball, el model d'*Independent Cascade* és força problemàtic a l'hora d'experimentar degut a la seva naturalesa estocàstica. Per aquest motiu, hem volgut dedicar-li una secció complementària per tal d'estudiar-lo una mica millor.

Per realitzar l'experimentació següent hem dut terme proves amb grafs de diferents mides i densitats, per tal de provar l'algorisme d'IC en diferents escenaris, amb l'objectiu d'entrende-ho millor. Per controlar la densitat dels grafs hem implementat un *script* que construeix grafs de mida n subjectes a una probabilitat de *wiring*. Aquesta probabilitat, uniforme per totes les arestes, determinarà si es construeix o no una aresta entre dos nodes, permetent-nos experimentar amb diverses densitats d'*edges*. És notable que en el cas de $p = 1$, s'obté un graf complet.

Primer provarem una versió "base" d'aquest, per veure com es comporta, i seguidament buscarem aplicar algunes optimitzacions per tal de veure si es possible millorar l'eficiència sense alterar el resultat en excés.

Per avaluar l'execució de l'algorisme sobre qualsevol dels grafs anteriors es fan servir dues mètriques:

- Nombre de Iteracions.
- Temps d'execució del programa.

Els grafs que es faran servir en l'etapa d'experimentació són els següents.

Graph ID	number of nodes (n)	**number of edges (m)**
G1	100	209
G2	100	519
G3	100	2512
G4	100	4950
G5	500	2474
G6	500	12510
G7	500	62344
G8	500	124570

Veiem a continuació els resultats obtinguts diferents execucions de la primera versió del programa. Per cadascun dels grafs G1...G8 es realitzen 2 execucions amb probabilitats de difusions distintes. Això ens permetrà analitzar com afecta al comportament de l'algorisme la mida i la densitat del graf sobre el que s'aplica, a més de la probabilitat de propagació.

Veiem els resultats de la primera versió de l'algorisme.

- **n = 100**

	p=0,4		**p = 0.8**	
	Iteracions	**temps**	**iteracions**	**nodes**
G1	98	0.2174	93	0.198562
G2	2	0.00929	1	0.00444
G3	1	0.01094	1	0.01144
G4	1	0.01999	1	0.019313

- **n = 500**

	p=0,4		p = 0.8	
	Iteracions	**temps**	**iteracions**	**nodes**
G5	499	13.56	3	13.479
G6	3	0.2276	2	0.226
G7	1	1.043	1	0.9999
G8	1	1.999	1	2.01

Analitzant els resultats obtinguts, veiem per una banda que l'algorisme triga més en convergir per grafs més grans, per exemple el graf G8 arriba a una solució fins a quinze vegades més lent que el G1, que té milers d'arestes menys.

D'altra banda, no tot és la mida del graf, la seva densitat també és rellevant de cara a trobar una solució. Veiem que hi han execucions que han trigat menys temps en grafs més grans però millor connectats que no en grafs més petits, com per exemple és el cas de G1 i G2 per $p = 0.8$. Aquest fenomen té sentit, ja que quant més dens sigui un graf, més oportunitats d'influenciar a altres nodes hi haurà, i per tant serà més senzill arribar a propagar la totalitat d'aquests.

Una altra observació a fer dels resultats obtinguts és la ineficiència de l'algorisme en certs escenaris, com per exemple grafs petits (com G1 i G2), on el nombre d'iteracions arriba a ser igual al de nodes pràcticament, o com en G5 i G6 (grafs igualment despoblats com G1 i G2, però de dimensió major). Per tal de solucionar això, implementem una segona versió.

Una primera optimització per aquest algorisme que ja s'ha vist en aquest treball és la de **conservar els resultats de l'expansió en cada iteració**. Fins al moment, el que es feia al afegir un node era realitzar la propagació, i en cas de que no s'arribés a la totalitat del graf, es buscava un nou node i es tornava a començar. Això és degut a que V(S+u) és en realitat el mateix que V(V(S)+u).

El que canviarem per aquesta segona versió serà evitar fer càlculs redundants en cada iteració, guardant després de cada intent de propagació el conjunt de nodes al que s'ha arribat i fent-lo servir com entrada (afegint el pròxim node més influent) a la pròxima iteració.

Veiem els resultats amb aquesta segona versió.

- **n = 100**

	p = 0.4		**p = 0.8**			
	time	**iterations**	**time**	**iterations**	**time**	**iterations**
G1	0.0320803	22	0.00690548	6	0.0053126	3
G2	0.0164735	7	0.00582793	3	0.00414518	1
G3	0.0110817	2	0.0101498	1	0.010163	1
G4	0.0185737	1	0.0187637	1	0.0180087	1

- **n = 500**

	p = 0.1		**p = 0.4**		**p = 0.8**	
	time	**iterations**	**time**	**iterations**	**time**	**iterations**
G5	0.288929	15	0.0660101	2	0.0638968	1
G6	0.235004	2	0.232924	1	0.226245	1
G7	1.01215	1	1.03345	1	1.02768	1
G8	2.01582	1	2.03707	1	2.01529	1

Les conclusions que es poden extreure en aquesta versió de l'algorisme són les mateixes que en l'escenari anterior, però a diferència d'ell es pot veure que s'ha millorat considerablement l'eficiència.

Aquesta millora tan considerable es deu principalment a que al reutilitzar els resultats obtinguts en propagacions anteriors ens estem estalviant un gran nombre d'iteracions alhora que arribem a uns mateixos resultats, doncs els nodes als que arribem en una certa iteració seran els mateixos als que arribarem en iteracions futures en cas de guardar-los per les pròximes propagacions.

Tot i això, veiem que aquesta millora únicament té efecte en grafs on es triga un cert nombre d'iteracions en arribar a una solució. Per grafs molt densos, com per exemple G4 i G8, on en una sola iteració ja es pot arribar a la solució, aquesta optimització no sorgeix cap efecte.

Per concloure amb aquest apartat, hem intentat solucionar el que considerem que és el major inconvenient en quant a eficiència per aquest algorisme; el càlcul de les influències. Actualment, en cada iteració del procés de difusió es calculen les influències de tots els nodes que es troben a *V-S*, i això clarament llastra el cost de l'algorisme.

Veiem a continuació què és el que succeeix si únicament realitzem aquest càlcul un sol cop, de forma que durant el procés de difusió no es tingui en compte els nodes que ja estan a la part propagada de cara a calcular la influència.

	p = 0.1		**p = 0.4**		**p = 0.8**	
	time	**iterations**	**time**	**iterations**	**time**	**iterations**
G1	0.0760312	100	0.0585978	100	0.0752572	100

És suficient amb una sola execució per veure que realitzar un únic càlcul de les influències no és una bona idea, doncs tant les iteracions com el temps d'execució es disparen respecte a la darrera optimització. Això es degut a que al realitzar aquest càlcul un sol cop, durant el procés de difusió no estarem tenint en compte els nodes que ja han estat propagat, i per tant les influències dels nodes no seran representatives.

Per exemple, podríem trobar-nos en una situació en que la majoria de veïns d'un node es troben a la part propagada, i per tant aquest node perdria la majoria de la seva influència, doncs no seria possible influenciar als seus veïns. En canvi, d'aquesta forma obtindríem que aquest node si que és influent, ja que no tindríem en compte que els seus veïns ja es troben al subconjunt S.

Per concloure, veiem que reduir el càlcul de les influències no té cap efecte en grafs densament connectat, on en una sola iteració ja s'arriba a la solució, com és el cas del graf G8.

	p = 0.1		**p = 0.4**		**p = 0.8**	
	time	**iterations**	**time**	**iterations**	**time**	**iterations**
G8	2.00389	1	1.99497	1	1.99807	1

References

[1] Cheng Long, Raymond Chi-Wing Wong, *Viral marketing for dedicated customers*

[2] Xiaojian Wu, Daniel Sheldon, Shlomo Zilberstein *Efficient Algorithms to Optimize Diffusion Processes under the Independent Cascade Model*

[3] Nitesh Trivedi, Anurag Singh *Efficient Influence MAximization in Social-Networks Under The Independet Cascade Model*

[4] Wenjing Yang, Leonardo Brenner, Alessandro Giua *Influence Maximization in Independent Cascade Networks Based on Activation Probability Computation*

[5] Paulo Shakarian, Abhinav Bhatnagar, Ashkan Aleali, Elham Shaabani, Ruocheng Guo *Difussion in Social Networks*

[6] Wenjun Wang, W.Nick Street *Modeling and maximizing influence diffusion in social networks for viral marketing*

[7] Wei Chen, Chi Wang, Yajun Wang *Scalable Influence Maximization for Prevalent Viral Marketing in Large-Scale Social Networks*

[8] Bithika Pal, Suman Banerjee, Mamata Jenamani *Threshold-Based Heuristics for Trust Inference in a Social Network*

[9] D. Kempe, J.M. Kleinberg, and É. Tardos. *Maximizing the spread of influence through a social network*

[10] Javier Béjar *Intel·ligència Artificial (GEI-IA) - Teoria. https://sites.google.com/upc.edu/intelligencia-artificial/inicio/teoria*

[11] *https://networkrepository.com/index.php*