Projecte Algorisimia

Pol Ros, Miquel Puig, Manel Murillo, Martí Duran

April 10, 2023

1 Introduccio:

2 Preliminars

Comencem exposant com hem fet la part de preliminars. A part dels fitxers requerits, hem fet el fitxer `lib.h` que conté mètodes útils per tots els altres codis. Hi podem trobar:

- `read_graph (filename)`: que llegeix i retorna el graf especificat dels exemples facilitats a l'Slack.
- `numberof_connected_components (graph)`: que retorna el nombre de components connexes que té el graf.

2.1 Linear Threshold

En el fitxer `difusioLT.h` conté el mètode `LT (graph, S, r)` que simula la difusió per LT. Descriurem breument com ho hem implementat.

1. **Inicialitzar:** Inicialitzem un vector `active` a fals dels vèrtexs que determina si el vèrtex està actiu o no. I posem com activat tots els vèrtexs que se'ns proporciona al conjunt inicial `S`.
   Llavors emprenem un vector `influenciable` que anirà guardant els vèrtexs que són susceptibles de ser activats en la següent iteració. (ho inicialitzem amb el conjunt de veïns dels vèrtexs de `S`).
2. **Bucle:** Anirem iterant sempre que hi hagi vèrtexs a activar (influenciables) i mentre s'hagi activat algun vèrtex a la iteració anterior (notar que si no s'ha activat cap node, ja no se'n poden activar més).
   - A la primera part del bucle és **calcula quins nodes que passen a actiu.**
   - Després s'actualitzen les dades utilitzades per seguir.

2.2 Independent Cascade

En el fitxer `difusioIC.h` conté el mètode `IC (graph, S, p)` que simula la difusió per IC. Seguidament descriurem com ho hem implementat.

1. **Inicialització:** Omplim una cua amb aquells nodes que són al vector solució, cua que usarem per al bucle.
2. **Bucle:** Mentre la cua no sigui buida, agafarem el node en front, l'esborrarem de la cua, mirarem els nodes al seu voltant que encara no estiguin activats i provarà d'activar-los amb la probabilitat hi ha establerta. Si l'activa, aquest node anirà a la cua de nodes actius. El bucle acaba quan ja no queden nodes a la cua i es retorna el nombre de nodes actius.

3 Algorismes

3.1 Greedy

3.1.1 LT

Hem pensat una solució simple per aquest algoritme. La nostra proposta consisteix a anar afegint els vèrtexs en ordre de grau fins que la solució sigui vàlida. Això està a `greedyLT.h`, a on també hi ha un altre parell de mètodes que es fan servir en altres algoritmes que detallarem a continuació:

- `greedy_until (graph, r, completion_ratio)` : implementa la idea que hem exposat abans, però en lloc de buscar que el conjunt de vèrtexs que aconsegueixi activar tot el graf, volem que només activi un percentatge (que és el tercer paràmetre). Ho hem implementat així.

  1. **Ordenem** els vèrtexs del graf segons el grau (està a la llista `vertexs`). I a la solució `S` hi posem el de grau màxim (`vertexs[0]`).
  2. **Bucle:** Mentre la nostra solució `S` no assoleixi activar el percentatge desitjat de vèrtexs del graf:
     (a) afegim a `S` el vèrtex de següent de `vertexs[i]`, i si arriba a més vèrtexs que el conjunt `S` anterior el conservem, sinó el trèiem i passem al següent.

  D'aquesta manera només afegim a la solució els vèrtexs que aporten alguna cosa. Hem fet aquesta distinció perquè en altres algoritmes calcularem una solució jugant amb el paràmetre ratio, ja que aquest mètode dona solucions aproximades de forma ràpida.

- `greedy (graph,r)`: que és una crida al mètode anterior, però que busca arribar a activar tot el graf. (`completion_ratio=1`). Aquest és el nostre algoritme greedy.

- `random_greedy_inverse (graph,r)`: genera una solució aleatòria de manera ràpida. Primer afegim tots els vertexs de grau mes gran que 1, i després treiem una cinquena part dels nodes de manera aleatoria. Si el conjunt resultat no es solució es trona a fer.

3.1.2 IC

En aquest model hem plantejat un algoritme que agafa primer aquells nodes que més grau tenen. Donat que així, augmenten les probabilitats que es propagui. En el cas que no sigui així, anirem afegint al conjunt solució els següents nodes en grau fins a trobar una solució vàlida. En el pitjor dels casos, el conjunt solució serà el total de tots els nodes del graf. L'algoritme ha sigut implementat de la següent forma:

- `greedyFunction (graph, diffusionProbability)`

1. **Ordenem** els nodes del graf segons el seu grau amb un sort i inicialitzem el vector solució amb el node de grau més alt.
2. **Iterem** mentre no trobem la solució, anem afegint al vector solució nous nodes.

3.2 Local Search

3.2.1 LT

L'algoritme de cerca local que hem triat per implementar és el de Simulated Annealing. Anem a descriure la nostra implementació `localSearch (graph, r)`:

- per calcular la nostra **aproximació inicial** fem servir l'algortime greedy anterior.
- el **bucle** te dos parts
  - **generar una solucio proxima** que ho fem de manera aleatòria eliminant i afegint elements al vector. Sempre que sigui solució.
  - despres **decidim** si la solució trobada ens la quedem o la n'escollim un altre, fent els càlculs de temperatura i probabilitat del simulated anneling.

Aquest algoritme té bastants paràmetres, el valor dels quals no teníem gaire clar. Hem fet la seva tria a base de prova i error. Els que hi han ara posats són els que ens han donat millors resultats.

NOTA: *aquí hem tingut molts problemes a l'hora de determinar els paràmetres. No hem pogut accedir als documents facilitats de la pàgina web* `https://dl.acm.org/` *ja que no tenim llicència amb la UPC, l'algoritme convergeix però molt lentament. No sabia com fer solucions veïnes aleatòries ni quins eren els paràmetres òptims. Hem provat moltes però no anava.*

3.2.2 IC

L'algoritme de cerca local que hem triat per implementar en aquest apartat es el de HillClimbing, més en concret, l'*Stochastic Hill Climbing*. La nostra implementació de `localSearch (graph, diffusionProbability)` ha sigut la següent:

- Com a **solució inicial**, hem agafat la que ens dona l'algoritme greedy que hem descrit anteriorment. Ens retorna un conjunt solució i el nostre objectiu és minimitzar-lo.
- En el **bucle** agafem un node random del conjunt solució i l'eliminem, mirem si aquest nou conjunt dona solució, en cas afirmatiu seguim eliminant un altre node random. En cas que no doni solució, tornem a afegir el node al conjunt i n'agafem un altre node random.

Com a observació d'aquest algoritme usat en aquest model, hem de dir que si s'executa el temps suficient, sempre retornarà un conjunt solució amb tamany 1, ja que al treballar amb probabilitats, en temps infinit sempre esta la possibilitat que només amb un node es pugui influenciar tot el graf.

3.3 Metaheuristic

3.3.1 Biased Random Key Genetic Algorithm

Hem seguit la recomanació de l'Slack i hem implementat un algorisme de Biased Random Key Genetic Algorithm. Comencem amb un conjunt solucions amb més nodes del compte (molt més gran que la solució) i les anem barrejant de manera que la mida vagi disminuint. Anomenem *població* al conjunt de solucions que anem fent reproduir. El fitxer `metaheuristicLT.h` i `metaheuristicIC2.h` conté els mètodes utilitzats:

- `reproduce (graph, r, A, B)` que dones dues aproximacions, retorna un *fill*. Tant A com B són solucions del problema, per tant, comencem generant el *fill* amb els nodes que estant als dos *pares*, i anem afegint elements aleatòriament de A o B fins que sigui solució. De manera que el fill normalment serà més petit que els dos *pares*.
- `choose_parents ( population)` donada una població, triem de manera pseudoaleatòria un conjunt de pares (amb els paràmetres actuals en triem 4 de 10).
- `population_indicator ( population )` retorna un enter que fa d'indicador de la població passada. Consisteix a sumar la mida de cada solució.
- `metaheuristic( graph, r)` Comenca buscant una població inicial aleatòria fent servir `random_greedy_inverse`. A continuació comenca el bucle.

  1. tria un subconjunt de pare cridant `choose_parents`
  2. reprodueix els pares entre si fent servir `reproduce`
  3. posa a la nova població els fills i els millor pares.

NOTA: *Com en la versió de local Search, no teníem accés a la pàgina web d'articles acadèmics. Jo crec que fallàvem sobretot en la reproducció dels fills. Ja que donats dos pares el fill tenia la majoria de nodes que tenien els pares (inicialment que el fill fos el mínim conjunt que activava tot el graf que estigués entre la unió i la interacció dels pares). I això fa que la població convergís cap a la intersecció de tots, i que deixes d'evolucionar. Hem mirat per internet, però no he sabut implementar maneres de reproduir fills que funcionessin. He provat moltes maneres de fer fills, i de triar quina és la següent població a partir dels pares i els fills i no convergia cap.*
Potser això estava als llibres, però no ho he sabut trobar. Ho hauria posat per l'Slack, però per Setmana Santa no crec que ningú m'hagués respost.

3.3.2 Tabu Search

Per a aquest model, en un primer moment vam decidir provar amb la metaheuristica anomenada Tabu Search, que és una metaheuristica de cerca local, aprofitant que a l'apartat anterior hem utilitzat el Hill Climbing. Es troba en el fitxer `metaheuristicIC.h`. El funcionament de l'algoritme és el següent:

- **Inicialització:** Creem una solució inicial amb l'algoritme greedy i guardem el conjunt solució en una llista de *tabus*, que registrarà les solucions positives que hem anat obtenint.
- **Bucle:** La condició del bucle és arbitrària i pot ser en funció de l'estat de la millor solució, o en un límit d'iteracions. Agafarem un "veí" del nostre conjunt solució, que serà un conjunt amb un element menys, esborrat de forma aleatòria del conjunt solució original. Si aquest nou candidat no està ja a la llista de tabú, comprovem si ens proporciona una millor solució que la nostra millor solució, en cas afirmatiu, posem aquesta com a millor solució a comparar a partir d'ara. Una vegada completa la condició del bucle, es retornarà la millor solució obtenida.

NOTA: *Aquest algoritme no hem sigut capacos de fer-lo funcionar de forma eficient, i fent proves amb entrades molt grans com les donades a l'Slack pel professorat, el programa no era capac de crear una sortida de bona qualitat.*