MINIMUM POSITIVE INFLUENCE DOMINATING SET

Algorithmics 2021/22

Roberto Amat Alins
Marc Camarillas Parés
Marc Nebot Moyano
Alejandro Salvat Navarro

Índex

1. GRAPH

1.1 Descripción

Hemos considerado conveniente que la representación del grafo venga dada por su lista de adyacencia.

1.2 Funciones de la clase

Tenemos dos funciones modificadoras:

- **addEdge**: Añade una nueva arista al grafo
- **addNode**: Añade un nuevo nodo al grafo

A parte de estas funciones modificadoras, tenemos diversas funciones consultoras, siendo estas:

- **size**: Nos devuelve el tamaño actual del grafo
- **getDegree**: Nos devuelve el grado del nodo pasado por parámetro
- **getNodes**: Pasado un vector por parámetro, le introduce los nodos del grafo
- **getSetNodes**: Pasado un set por parámetro, le introduce los nodos del grafo
- **getAdjacent**: Dado un nodo y un vector vacío, llena el vector con los nodos adyacentes al nodo pasado por parámetro

1.3 Experimentación

Inicialmente hemos probado con el grafo predeterminado que se nos proporciona en el enunciado, para ver que nos funcionara.
Además, hemos generado 3 grafos a mano con las siguientes características:

- **Grafo23.txt** → 23 nodos, 34 aristas.
- **Grafo50.txt**→ 50 nodos, 109 aristas.
- **Grafo50_1_.txt** → 50 nodos, 62 aristas.

Una vez creados los grafos hemos probado que nos genere bien la lista de adyacencia del grafo.
Para comprobar que funciona con grafos más grandes, hemos cogido los grafos que se nos han proporcionado para

2.BASICS

2.1 Descripción

Dado un grafo $G(V, E)$, y un conjunto de nodos $S \subseteq V$, determina en tiempo polinómico si un S es conjunto dominante de influencia positiva de G, y en caso de serlo determina si es minimal o no.

2.2 Funciones de la clase

Para esta clase hemos creído conveniente tener las dos siguientes consultoras:

- **isPIDS**: nos dice si el conjunto que le pasamos es PIDS o no.
    - Para saber si es PIDS, recorremos todos los nodos del grafo y para cada uno de ellos miramos si $|Adj_s(n)| \geq |Adj(n)|$. Si encontramos alguno que no cumpla dicha condición, *S* no será PIDS.
      Siendo:
      $|Adj_s(n)|$ → aristas adyacentes al nodo n, que pertenecen a *S.*
      $|Adj(n)|$ → grado del nodo n
- **isMinimalPIDS**: si es PIDS nos dice si es minimal o no.
    - Dado un conjunto *S* que sea dominante de influencia positiva en el grafo *G,* sabemos que es minimal, si $\forall s \in S$, $S - \{s\}$ no es un conjunto dominante en el grafo *G.*

2.3 Experimentación

Hemos probado con el grafo predeterminado que nos daban en el enunciado y hemos comprobado que si eliminamos algún nodo del conjunto y se dejaba de cumplir la condición para que fuera PIDS, o si añadimos nodos, el programa nos daba los resultados esperados.

3. GREEDY

3.1 Descripción

Dado un grafo $G(V, E)$, y un conjunto de nodos $S \subseteq V$, encontrar un conjunto de nodos S que sea dominante de influencia positiva mediante un algoritmo voraz. El algoritmo escogido es Pan's, cuyo pseudocódigo es el siguiente:

Algorithm 2 Pan's greedy algorithm [17]

Input: a simple, connected undirected graph $G = (V, E)$
Output: a positive influence dominating set S

1: Rename the vertices from V such that $\{v_1, v_2, \cdots, v_n\}$ are the vertices in ascending order of the degree
2: $S \leftarrow \emptyset$
3: $\overline{S} \leftarrow V \setminus S$
4: **for** $i = 1$ **to** n **do**
5: **if** $h_S(v_i) > 0 : v_i$ is an uncovered vertex **then**
6: $\rho \leftarrow h_S(v_i)$
7: **for** $j = 1$ **to** ρ **do**
8: $u^* \leftarrow$ **argmax**$\{cover\text{-}degree(u) \mid u \in N_{\overline{S}}(v_i)\}$
9: $S \leftarrow S \cup \{u^*\}$
10: $\overline{S} \leftarrow V \setminus S$
11: **end for**
12: **end if**
13: **end for**
14: **return** S

Fig.1 Pseudocódigo del algoritmo Pan's greedy

Este algoritmo tiene coste $\theta(nlogn)$, de manera que permite tratar eficientemente ficheros de gran tamaño.

El algoritmo se basa en el uso de la siguiente función voraz:

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

Donde $h_S(v) = \lceil \frac{deg(v)}{2} \rceil - |N_S(v)|$

- **Cover degree** determina el número de vecinos de un vértice dado V que tengan un valor de Hs > 0, es decir, que la resta de su grado entre dos menos el número de sus vecinos que pertenezcan al conjunto solución S sea estrictamente positiva.

  Esta propiedad es la que nos ayuda a seleccionar eficientemente los vértices que necesitaremos seleccionar para encontrar una solución.
  Cada iteración de todos los covers degree calculados nos quedamos con el argmax y lo añadimos al conjunto solución.

Puntualización: Aunque hayamos acabando usado Pan's, intentamos resolver el problema mediante el algoritmo IGA PIDS, el cual es a priori un poco más eficiente que Pan's, pero tuvimos algún problema con su implementación y por eso decidimos escoger Pan's, que con una complejidad temporal practicamente identica tiene una facilidad de implementación mucho mayor.

3.2 Funciones de la clase

Funciones privadas:

- **getN**: Nos devuelve el número de vecinos de un nodo que pertenecen al conjunto solución, usada por Pans y IGA PIDS.
- **getH**: Devuelve el valor de Hs, explicado previamente, usada por Pans y IGA PIDS.
- **coverDegree**: devuelve el valor del Cover Degree de un vértice, explicado previamente.
- **needDegree**: Función utilizada para el algoritmo IGA PIDS.

Funciones públicas:

- **greedy**: Creadora de la clase.
- **Pans**: Realiza todos los cálculos del algoritmo Pan's.
- **graphPrunning**: Función auxiliar de algoritmo IGA_PIDS.
- **IGA_PIDS**: Realiza todos los cálculos del algoritmo voraz IGA_PIDS.

3.3 Experimentación

Hemos ejecutado el programa con todas las instancias proporcionadas y hemos obtenido los siguientes resultados: (tener en cuenta que este resultado ha sido ejecutado en un mismo pc, pero puede variar si se ejecuta en un pc con especificaciones diferentes, es simplemente una medida informativa):

FICHERO	TIEMPO (S)	# NODOS DEL CONJUNTO SOLUCIÓN.
graph_football	0,001	75
graph_jazz	0,005	86
ego-facebook	0,11	1981
graph_actors_dat	0,2	3278
graph_CA-AstroPh	0,36	7128
graph_CA-CondMat	0,16	9944
graph_CA-HepPh	0,17	4911
socfb-Brandeis99	0,19	1578
socfb-Mich67	0,1	1506
soc-gplus	0,26	8532

Fig.2 Tabla de resultados de la experimentación con el algoritmo greedy

4. STATE

4.1 Descripción

Para poder aplicar *local search* primero debemos definir qué es un estado, la representación de estado y cómo podemos identificar un estado solución.
Nuestra representación de estado para este problema consiste en 3 elementos:

- El grafo sobre el que estamos trabajando, representado por sus nodos y aristas
- El subconjunto solución de nodos del grafo, siendo estos un subconjunto dominante de influencia positiva, pero que no será minimal en estados intermedios
- Un subconjunto de nodos del grafo que no están presentes en la solución, puede expresarse como el subconjunto de nodos del grafo complementario al anterior

El criterio para identificar si un estado es mejor que otro se va a basar en nuestra función heurística (véase la figura 3)

Aquellos estados que vamos a considerar estados solución dentro del problema, son aquellos que, el subconjunto solución de nodos, marcan las características de los MPIDS, se trate de un subconjunto de nodos dominante del grafo, pero que a su vez, al menos la mitad de los nodos adyacentes a los de la solución también formen parte de la misma y el subconjunto sea minimal, es decir, que sea imposible representar un subconjunto solución con un número menor de nodos y que este sea una solución válida al problema.

5.LOCAL SEARCH

5.1 Descripción

El algoritmo de búsqueda local escogido, ha sido el algoritmo determinista de Hill Climbing, el cual dada una solución inicial aplica un seguido de operadores para modificar la solución inicial y explorar posibles mejores soluciones. Una vez los estados generados por los operadores no mejoran la solución se considera que se ha llegado a un óptimo local y se termina el algoritmo.

5.2 Solución Inicial

Para este algoritmo hemos diseñado dos posibles soluciones iniciales:

- La primera se trata de un subconjunto de vértices igual al conjunto de vértices del grafo, es decir, seleccionando todos los vértices del mismo e introduciendolos a nuestra solución.
- La segunda posible solución inicial es la generada por el algoritmo greedy

5.3 Operadores

Una vez generada nuestra solución inicial, pasamos a ejecutar el algoritmo con 2 operadores que nos permiten explorar todo el espacio de soluciones:

- Un operador de swap, que nos permite intercambiar un vértice que forma parte del conjunto solución con otro que no está
- Un operador erase, que nos elimina el vértice escogido de la solución

Con solo estos dos operadores, podemos explorar todo el espacio de soluciones ya que queremos minimizar el tamaño del subconjunto de la solución, por lo que un operador para añadir vértices a la solución nos explorará soluciones peores. Gracias al operador de swap en una solución, podemos escoger un grupo de vértices tal que uno o más sean innecesarios, de manera que aplicamos el erase para obtener una solución mejor.

5.4 Heurística

La heurística que hemos utilizado para Hill Climbing se basa en la fórmula:

$$(\sum_{u=0}^{v} \frac{ASu}{gu}) + SS * NV$$

Fig.3 Función Heurística usada en nuestro algoritmo de búsqueda local

Siendo:

- AS_u: Número de vértices adyacentes al vértice u que pertenecen al PIDS
- g_u: Grado del vertice
- SS: Tamaño de la solución
- NV: Número de nodos que tiene el grafo

(Añadimos NV para usarlo a modo de ponderador y así darle más peso al tamaño de la solución en la función heurística)

La cual se busca minimizar, hasta que encontremos un mínimo local.

5.5 Experimentación

Hemos probado con grafos generados a mano y hemos comprobado que la solución inicial obtenida con el greedy no expande muchos nodos, debido a que está muy cerca de un óptimo local y no permite la exploración. En cambio la solución inicial que coge todos los nodos, nos permite explorar muchos nodos, pero no llega a mejorar la solución obtenida en el caso anterior.

FICHERO	TIEMPO (S)	# NODOS DEL CONJUNTO SOLUCIÓN.
graph_football	6	67
graph_jazz	7	81
ego-facebook	na	na
graph_actors_dat	na	na
graph_CA-AstroPh	na	na
graph_CA-CondMat	na	na
graph_CA-HepPh	na	na
socfb-Brandeis99	na	na
socfb-Mich67	na	na
soc-gplus	na	na

Fig.4 Tabla de experimentación

Conclusión: Los resultados con NA son porque el ordenador no puede ejecutarlos por falta de potencia, tardan demasiado tiempo o peta la ejecución.

6.METAHEURISTICA

6.1 Descripción

La metaheurística escogida ha sido la *Iterative Local Search la* cual dada una solución inicial generada por el algoritmo greedy o el *local search*. Una vez generado el estado inicial, procedemos a ejecutar un bucle que irá explorando el espacio de soluciones durante n segundos para encontrar una mejor solución

6.2 Funciones de la clase

Para la metaheurística que hemos escogido hemos decidido seguir el siguiente pseudocódigo:

$s_0 \leftarrow$ GenerateInitialSolution()
$\hat{s} \leftarrow$ LocalSearch(s_0)
while termination conditions not met **do**
  $s' \leftarrow$ Perturbation($\hat{s}, history$)
  $\hat{s'} \leftarrow$ LocalSearch(s')
  $\hat{s} \leftarrow$ ApplyAcceptanceCriterion($\hat{s}, \hat{s'}, history$)
endwhile

Fig.5 Pseudocódigo de la metaheurística

- **disturbance**: Hemos decidido aplicar la función *disturbance* en nuestra metaheurística que lo que hará será una transformación muy drástica al estado solución actual para poder explorar máximos locales nuevos que nos permitan encontrar mínimos locales mejores que el que tenemos actualmente.

Por cada perturbación lo que haremos será lo siguiente:

1. Quitaremos ¼ de nodos aleatorios de nuestro estado solución actual para conseguir una permutación que nos permite explorar distintos estados y encontrar nuevos mínimos locales.

2. Si al quitar ¼ de nodos aleatorios encontramos un estado no-solución, regresaremos a nuestra solución anterior y le añadiremos el número de nodos de los que no han sido añadidos todavía a nuestro estado dados por la siguiente ecuación:

$$\frac{\frac{ns*4}{3} + ndg}{2}$$

Fig. 6 Fórmula de la perturbación

ns = número de nodos en el estado solución

ndg = número de nodos totales

3. Si al añadir esa cantidad de nodos resulta que llegamos a un estado no-solución, no realizaremos ninguna de las perturbaciones anteriores y nos acabaremos quedando con el estado solución que estábamos intentando permutar.

- **acceptanceRequirements:** Esta función nos permitirá escoger el nuevo punto a partir del cuál seguiremos explorando, deberemos decidir si cogemos nuestro estado inicial actual o seguir con el nuevo que hemos perturbado, para esto aplicaremos una función de probabilidad.

- **probabilityOfAcceptation:** Dada una solución actual y una nueva, si la heurística de la nueva solución es menor, es decir, mejora (recordemos que minimizamos la heurística), cogeremos la nueva solución como siguiente punto a partir del cuál exploramos con una probabilidad del 100%.

  Por otro lado, si no mejora, la probabilidad de coger el nuevo estado será dada por la siguiente ecuación:

$$e^{-\frac{Sh'-Sh}{T*ns}} \; x \; 100\%$$

Fig.7 Probabilidad de aceptación de un nuevo estado

ns = número de nodos en el estado solución
Sh' = valor de la heurística del nuevo estado solución
Sh = valor de la heurística del estado solución
T = temperatura

Aclaración: La temperatura vendrá dada gracias a la fase de experimentación, contra más grande sea esta mayor será el índice de aceptación. Podemos observar que la funcionalidad de temperatura es similar a la del *Simulated Annealing*

6.3 Experimentación

FICHERO	# NODOS DEL CONJUNTO SOLUCIÓN.	DIFERENCIAS EN CUANTO A # DE NODOS
graph_football	67	-8
graph_jazz	80	-5
ego-facebook	na	na
graph_actors_dat	na	na
graph_CA-AstroPh	na	na
graph_CA-CondMat	na	na
graph_CA-HepPh	na	na
socfb-Brandeis99	na	na
socfb-Mich67	na	na
soc-gplus	na	na

Fig.8 Tabla de experimentación

Conclusión: Los resultados con NA son porque el ordenador no puede ejecutarlos por falta de potencia, tardan demasiado tiempo o peta la ejecución.

7.CPLEX

7.1 Descripción

En el apartado de CPLEX teníamos una plantilla que encontraba un conjunto dominante. El trabajo necesario para encontrar el MPIDS que buscamos es adaptar la plantilla, de manera que el subconjunto solución cumpla con los requisitos de un MPIDS.

El cambio se ha basado en cambiar la condición con la que se introducía un vértice al subconjunto solución, cambiando la restricción a que por lo menos la mitad de sus vecinos deben permanecer también en el subconjunto solución. De esta manera, el programa devuelve el valor deseado.

Una vez hecho el cambio, el programa crea un modelo de cplex para nuestro problema, al cual se irán añadiendo todos aquellos nodos de manera que dicho modelo forme el MPIDS.

7.2 Experimentación

FICHERO	TIEMPO (S)	# NODOS DEL CONJUNTO SOLUCIÓN.
graph_football	9,62	58
graph_jazz	0,17	75
ego-facebook	0,57	1964
graph_actors_dat	76,87	2916
graph_CA-AstroPh	29,55	5866
graph_CA-CondMat	9,47	7721
graph_CA-HepPh	2,06	3911
socfb-Brandeis99	63,15	1382
socfb-Mich67	193,35	1289
soc-gplus	1,63	8179

Fig.9 Tabla de experimentación

8.MANUAL DE LA APLICACIÓN

8.1 Compilación

Para facilitar la compilación de nuestro código a la par de su ejecución, hemos creado un archivo Makefile, el cual tiene 2 funcionalidades principales:

- make: Se trata de la funcionalidad por defecto, esta crea los ficheros objeto de todas las clases implementadas en el código y un ejecutable para cada uno de los distintos algoritmos implementados
- make clean: Esta funcionalidad se basa en borrar todos los ficheros objeto creados junto a los ejecutables de la aplicación

8.2 Ejecución

Nuestro programa se divide en 3 ejecutables, uno para cada algoritmo implementado, empezando los 3 por escoger si el grafo con el que quieres trabajar es el grafo por defecto (se puede ver en el enunciado de la práctica), o quieres escribir un nuevo grafo desde 0. Una vez escogida la opción, el comportamiento de cada ejecutable es ligeramente diferente.

8.2.1 Ejecución Basics

Una vez ejecutado basics y escogido el grafo con el que vamos a trabajar, basics nos pide que introducimos el subcon junto de vértices del mismo, del cual queremos comprobar si es MPIDS o no. Una vez introducido el subconjunto, marcando su final con un -1, el programa nos imprimirá por terminal el grafo escogido previamente y un mensaje que nos dirá si ese subconjunto se trata o no de un MPIDS

8.2.2 Ejecución Greedy

Una vez introducido el grafo con el que queremos trabajar, Greedy nos encontrará automáticamente un subconjunto de vértices MPIDS del mismo. A parte, tambien nos indicará si el subconjunto también es minimal

8.2.3 Ejecución Local Search

Una vez escogido el grafo sobre el que vamos a trabajar, este ejecutable nos va a pedir cual queremos que sea la solución inicial de la que partirá el algoritmo, estas pueden ser:

- Greedy: La solución inicial se tratará de la solución encontrada por el algoritmo Greedy.
- Initially all nodes are part of the S set: La solución inicial tratará de un subconjunto en el que se encuentran todos los nodos del vértice, de manera que aplicando una serie de operadores podamos recorrer todo el espacio de soluciones.

Una vez escogida la solución inicial, el algoritmo de búsqueda local nos imprimirá el grafo con el que hemos trabajado por terminal, nos indicará cuánto tiempo ha tardado y cuantos nodos ha expandido, y, por último, nos imprimirá la solución encontrada junto a un mensaje que nos dirá si este subconjunto es minimal o no.

9.CONCLUSIONES

Este trabajo nos ha servido para tener más conocimientos acerca de los métodos de optimización de problemas computacionalmente complejos como es el caso de *MPIDS*. Además de teóricamente también nos ha permitido ponerlos en práctica y ver el funcionamiento de algoritmos como el *Hill-Climbing* (usado en el *Local Search*),el *Iterative Local Search* (usado para la metaheurística) y el uso de CPLEX. Por otra parte hemos podido profundizar más en los conocimientos impartidos en clase sobre los algoritmos greedy.

De acuerdo con los resultados de la experimentación de nuestro trabajo, hemos observado lo siguiente:

- **En el greedy:** El hecho de calcular las cosas inicialmente y luego solo tenerlas que calcular cuando era preciso ha hecho que nos disminuya mucho la complejidad temporal, lo cual ha hecho que nos pueda dar la solución en el tiempo deseado.

- **En el local search:** La importancia de una buena solución inicial, partir de una mejor solución inicial acaba encontrando soluciones mejores.

- **En la metaheurística:** La importancia de la generación de múltiples soluciones iniciales para conseguir un espacio de estados mucho mayor y la obtención de una exploración más en profundidad, consiguiendo así la optimización de soluciones

- **En el CPLEX:** La importancia de la implementación de las debidas restricciones para obtener la solución deseada.

10.WEBGRAFIA

- *Development of hybrid metaheuristics based on instance reduction for combinatorial optimization problems* (Abril 2017). https://www.iiia.csic.es/media/publications/TESIS_PINACHO_DAVIDSON_PEDRO_PABLO.pdf

- *4.4 Búsqueda Local Iterativa (Iterated local search)*. (2004). 4.4 Búsqueda Local Iterativa (Iterated local search). https://ccc.inaoep.mx/%7Eemorales/Cursos/Busqueda/node65.html

- *Stanford Large Network Dataset Collection*. (2014). Stanford. https://snap.stanford.edu/data/

- *Búsqueda local iterada - programador clic*. (2016). Programmer Click SL. https://programmerclick.com/article/29821842823/

- *ILOG CPLEX Optimization Studio*. (2020). IBM. https://www.ibm.com/docs/en/icos/20.1.0?topic=optimization-example-c-api