**Domingo 28 Noviembre 2021**

# Conjunto Minimal Dominador de Influencia Positiva

Carla Campàs, Beatriz Gomes, Pau Vallespí, Andres Bercowsky

Algorismia
Universitat Politècnica de Catalunya
2021-2022 Q1

**Abstract**

El objetivo de esta práctica es identificar conjuntos dominadores de influencia positiva e intentar conseguir un conjunto minimal de estos. Veremos diferentes técnicas para conseguir estos. Usando algoritmos voraces, búsqueda local con heurística y metaheurística, e Integer Linear Programming. Posteriormente compararemos las soluciones de estas con soluciones ya existentes del mismo problema. También sacaremos conclusiones de la mejor metodología para obtener conjuntos minimales dominadores de influencia positiva.

# 1 Introducción

El concepto de los conjuntos dominadores está al auge, descubriéndose cada vez más aplicaciones que este tiene en la vida real. Aparte del aspecto más tecnológico de este concepto como serían las redes de conexiones entre ordenadores. También se han visto aplicaciones en la vida cotidiana, explicados a continuación.

Muchos usos de los grafos dominadores están centrados en planificación. Un ejemplo de esto sería la planificación de eventos y charlas en una conferencia. Si les preguntamos a todos los asistentes de la conferencia que charlas y/o eventos les gustaría más asistir, con esta lista podríamos determinar un conjunto de horarios y conferencias para maximizar los asistentes en todas estas. Un problema similar es el de planificación de rutas de buses escolares donde hay ciertas restricciones como, la distancia máxima que un niño deberá caminar hasta llegar a la recogida del bus o el número de minutos que un bus puede estar en ruta. Podemos equiparar este a un mapa con los diferentes puntos de recogida de los niños y deberíamos determinar qué conjunto dominador podemos crear para recoger a todos los niños con las restricciones dadas.

Hay otros problemas que vemos que se pueden equiparar también a conjuntos dominadores. Como por ejemplo, encontrar el número de colores necesarios para pintar un mapamundi de colores distintos sin que dos países vecinos tengan el mismo color. La versión de esto donde intentamos minimizar el número de colores en el mapa, es un problema de conjuntos dominadores minimales, que también veremos a continuación. El Sudoku, también suele ser un problema ejemplar de los conjuntos dominadores. Entendemos que dos casillas estarán conectadas si están en la misma fila, columna o cuadrado de nueve. En el caso que tengamos un sudoku inicial no vacío, estos ya tendrán un “color” (del 1 al 9) y se tiene que encontrar el conjunto dominador donde ninguna de las conexiones tenga el mismo “color”.

Uno de los problemas más interesantes que se han desarrollado en términos de los grafos dominadores es el problema de las redes sociales. Específicamente de la influencia que estas tienen sobre las personas. El problema se presenta como un conjunto dominante de influencia positiva. De esta manera se intenta encontrar un conjunto de personas con el cual la red de usuarios de las redes sociales se verá positivamente influenciada. Vamos a realizar un estudio de los conjuntos dominadores en las redes sociales. En este vamos a explorar diferentes metodologías para la obtención de los conjuntos dominadores de influencia positiva. También intentaremos abordar el problema de encontrar un conjunto minimal. A continuación describiremos el problema que tenemos entre manos en más detalle y expondremos las técnicas que vamos a utilizar para realizar este estudio.

# 2 Descripción del Problema

## 2.1 Grafos

Un grafo es una tupla (V(G), E(G)) donde V(G) es un conjunto finito del cual los elementos se llaman vértices. E(G) es un conjunto de pares no ordenados, en nuestro caso este par serán dos elementos distintos y el par será único. Los elementos contenidos en cada elemento del par deberá estar en V(G). Los elementos que están en E(G) se llaman aristas. El grafo también se puede expresar de la siguiente forma G = (V, E).

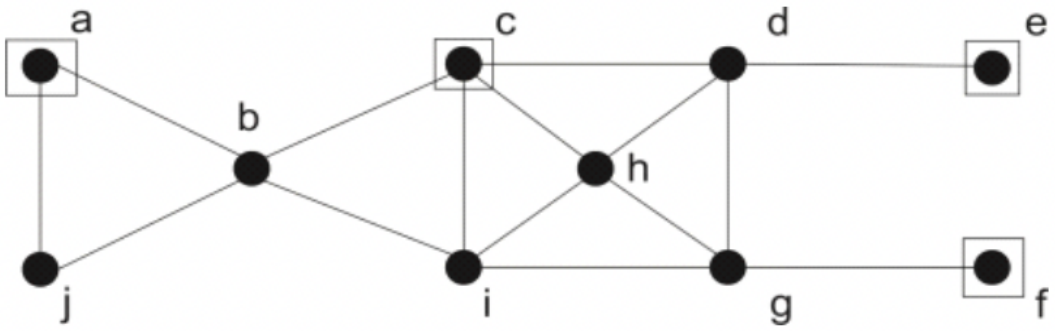
En nuestro caso, las aristas serán no-direccionadas. Esto quiere decir que si una arista va de un nodo (e1) a otro (e2) existe un camino directo desde e1 a e2 y de e2 a e1.

El grafo con el que trataremos en esta práctica también es conexo, este concepto quiere decir que hay un camino entre cualquier dos elementos de V(G).

## 2.2 Conjuntos Dominadores

Dado un Grafo En los grafos un conjunto dominador es un subconjunto de nodos ($D \subseteq V$) si para todo vertex de V ($v \in V$) es un elemento de S o adyacente a un elemento de S.

Damos un ejemplo de este tipo de grafos a continuación:




Tenemos el siguiente grafo:
$G = (V, E)$ donde $V = \{a, b, c, d, e, f, g, h, i, j\}$ (indicado en la figura con un punto negro) y $E = \{(a, b), (a, j), (b, c), (b, i), (b, j), (c, d), (c, h), (c, i), (d, e), (d, g), (d, h), (f, g), (g, h), (g, i), (h,$ representado como una línea entre dos vértices. El subconjunto $S = \{a, c, e, f\}$ Es un set dominador en G (indicado por un cuadrado en el grafo).

Observamos que todos los vértices de la figura están conectados a un nodo del subconjunto. Esos que no están conectados, son los nodos del subconjunto.

### 2.2.1 Conjuntos Dominadores de Influencia Positiva

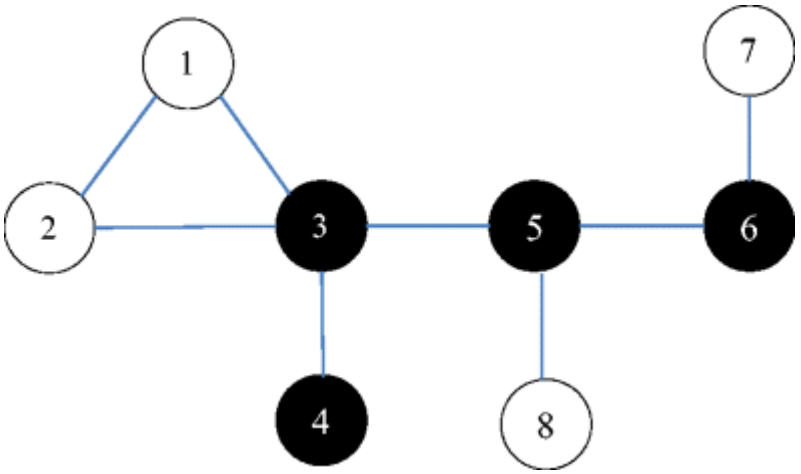
Una vez vista la definición anterior podemos observar que un conjunto dominador de influencia positiva, como está descrito en la documentación de la práctica. Es un grafo dominador con todos los nodos que no sean del subconjunto conectados a como mínimo 50% de nodos que están en el subconjunto de nodos.

### 2.2.2 Conjuntos Dominadores de Influencia Positiva Minimal

Un conjunto denominador de influencia positiva minimal es un conjunto denominador de influencia positiva donde si quitamos cualquier nodo del subconjunto dado, ya no tendremos un conjunto denominador de influencia positiva.

Dicho de otro modo, dado cualquier elemento del conjunto V(G) ($v \in V(G)$) como mínimo 50% de sus vecinos (nodos adyacentes a $v$) pertenecerán al subconjunto $s \in S$ donde $S \subset V$. Además si quitamos cualquier elemento del subconjunto ($s \in S$) tendremos como mínimo un vértice donde menos del 50% de sus nodos vecinos pertenecen al subconjunto.

A continuación vemos un ejemplo:




En este ejemplo nuestro conjunto de vértices será: $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$. El conjunto de aristas: $E = \{(1,2), (1,3), (2,3), (3,4), (3,5), (5, 6), (5, 8), (6, 7)\}$ y finalmente nuestro subconjunto será $S = \{3, 4, 5, 6\}$. Observamos que si quitamos cualquier nodo del subconjunto ya no sería dominador positivo. Si quitamos el vértice 3 del subconjunto, 1, 2, 4 ya no estarían conectados a ningún nodo del subconjunto por lo tanto este no sería ni dominador ni dominado positivo. Si quitamos el 4, el nodo 3 estaría rodeado por una mayoría de nodos fuera del subconjunto, por lo tanto el grafo seguiría siendo dominador pero no dominador positivo. En el caso del nodo 5, tendríamos que el nodo 6 ya no estaría conectado a ningún nodo del subconjunto volviendo a dejar el grafo como no-dominador. Y finalmente, si quitamos el 6, el 7 nos quedaría conectado únicamente a nodos fuera del subconjunto, otra vez dejando este subconjunto como no-dominador.

Entendiendo este ejemplo vemos el objetivo de este estudio, vamos a comparar diferentes metodologías para llegar a un conjunto dominador de influencia positiva minimal.

## 2.3 Conceptualización Temporal del Problema

Inicialmente vamos a mirar la conceptualización temporal de encontrar un set dominante. Este problema es NP-Completo. A continuación vemos la explicación de esto.

Inicialmente vemos que si un problema es NP-Completo, la comprobación de este problema se deberá verificar en tiempo polinómico. Para verificar que un conjunto de vértices forma un set dominante, tenemos que mirar que todos los vértices están conectados a un nodo del subconjunto o pertenece al subconjunto. Podemos hacer esta comprobación en un tiempo $O(|V| + |E|)$ usando la siguiente estrategia.

***for v en V:***
    ***si v no esta en S***
        ***tiene alguna arista conectada con un nodo en el conjunto S***
        ***si esto no es verdad***
            ***return false***
***return true***

Además podemos comprobar que este problema es NP-Difícil, reduciendo este a otro problema NP-Difícil. Este problema es similar al problema NP-Difícil ***Vertex Cover***, por lo tanto vamos a reducir el problema del conjunto dominador al problema del vertex cover.

Para el problema del vertex cover tenemos un grafo G = (V, E) y un número entero *k* que es el subconjunto de vértices. Construiremos un nuevo grafo G' = (V', E') para el problema del set dominador. Esto lo haremos de la siguiente manera:

- ***E':*** Para cada par de vértices del conjunto del conjunto de aristas del grafo original $(u, v) \in E$, crearemos un nuevo vértice {uv} y uniremos este a los nuevos vértices u y v.
- ***V':*** Va a ser igual que el conjunto V del grafo original G.

El nuevo grafo se puede conseguir en tiempo polinómico $O(|V| + |E|)$. Y la reducción se puede probar con las dos siguientes afirmaciones:

- Asumimos que el grafo G tiene el vertex cover VC con tamaño k. Cada arista en G tiene como mínimo uno de sus vértices $\{u, v\}$ en el VC. Si u está en VC, entonces el vértice adyacente también está cubierto por algún elemento del VC. Para todos los nuevos vértices UV para cada arista, el vértice es adyacente al a tanto u como v, uno de los cuales es parte de VC, como hemos demostrado previamente. Por lo tanto, todos los vértices añadidos al grafo G' correspondientes a las aristas también estarán cubiertos por VC. El conjunto de vértices que forma el vertex cover con tamaño k también forma el conjunto dominador del grafo G'. Por lo tanto si G tiene un vertex cover G' tiene un conjunto dominador del mismo tamaño.
- Asumimos que el grafo G' tiene un conjunto dominador (CD) del tamaño k y surgen dos posibilidades. El vértice está en CD és un vértice original o es parte del conjunto de nuevos vértices añadidos UV para cada arista {u, v}. En el segundo caso, como cada vértice está conectado a dos vértices de la arista (u y v), podemos sustituirlo tanto por u como por v. Esto eliminará todos los nuevos vértices mientras miramos todas las aristas de G'. Los nuevos vértices son dominados por el CD modificado y cubre todas las aristas de G con u o v para cada arista UV. Por lo tanto, si G' tiene un conjunto dominador de tamaño k, G tendrá un vertex cover de tamaño máximo k.

Con esta reducción podemos decir que G' tendrá un set dominador si y sólo si G tiene un vertex cover. Por lo tanto, cualquier instancia del problema del conjunto dominador puede ser reducida a cualquier instancia del vertex cover. Y el problema del conjunto dominador también es NP-DIfícil y NP-Completo.

Por lo tanto vemos que todos los problemas que vamos a abordar en este estudio serán de tiempo NP-Difícil y NP-Completos.

## 2.4 Descripción de Conceptos

Para la realización de esta práctica vamos a utilizar diferentes métodos y algoritmos para encontrar la solución más eficiente o más ajustada posible. A continuación damos una breve explicación de estos conceptos.

### 2.4.1 Algoritmos Voraces

Por definición, un algoritmo voraz (también conocido como greedy) es una estrategia de búsqueda por la cual se sigue una heurística consistente en elegir la opción óptima en cada paso local con la esperanza de llegar a una solución general óptima. Normalmente se aplica a los problemas de optimización. En los problemas que resolvemos con algoritmos voraces, existe una entrada de tamaño n que son los candidatos a formar parte de la solución. Además, existe un subconjunto de estos n candidatos que satisface ciertas restricciones: se llama solución factible. Por último, hay que obtener la solución factible que maximice o minimice una cierta función objetivo: se llama solución óptima.

En nuestro caso utilizaremos este tipo de algoritmos para poder buscar una solución inicial para posteriormente utilizarla en los algoritmos de búsqueda local. Lo que haremos será, en cada paso del algoritmo voraz, coger el nodo con más vecinos y así buscaremos la mejor solución inicial que podamos.

### 2.4.2 Búsqueda Local

La búsqueda local es la base de muchos de los métodos usados en problemas de optimización. Se puede ver como un proceso iterativo que empieza en una solución y la mejora realizando modificaciones locales. Básicamente empieza con una solución inicial y busca una mejor solución siguiendo una heurística que hay que definir. Si la encuentra, el algoritmo reemplazará su solución actual por la nueva y continua con el proceso, hasta que no se pueda mejorar la solución actual. Una vez terminado el algoritmo de búsqueda local, habremos encontrado la solución.

En nuestro caso utilizaremos la búsqueda local para, una vez conseguida una solución inicial buena, poder explorar el espacio de soluciones para mejorar esta solución, y así intentar conseguir una solución óptima para el problema.

### 2.4.3 Metaheuristica

Una metaheurística es un método heurístico para resolver un tipo de problema computacional general, usando los parámetros dados por el usuario sobre unos procedimientos genéricos y abstractos de una manera que se espera eficiente. Normalmente, estos procedimientos son heurísticos. Así pues, una heurística consiste en la capacidad de realizar innovaciones positivas para conseguir los fines que se pretenden.

Para nuestro problema, implementaremos una metaheurística para explorar el espacio de soluciones y poder encontrar una solución óptima al problema.

### 2.4.4 Integer Linear Programming

ILP, o *Integer Linear Programming* engloba todos aquellos problemas de optimización en los que las variables que forman la solución son siempre enteros, mientras que tanto las restricciones como la función son lineales. Por lo que cualquier valor no entero queda completamente fuera del espacio de soluciones.

No es difícil ver que nuestro problema pertenece a esta clase. Las variables son los nodos, que pueden pertenecer o no al conjunto dominante de influencia positiva. Esa binariedad las convierte inmediatamente en variables con soluciones únicamente dentro del espacio de enteros. Por otro lado, la restricción que impide que ningún nodo no dominante tenga más de la mitad de sus vecinos también fuera del conjunto dominante, es lineal. Buscamos minimizar la cantidad de nodos, por lo que nuestra función objetivo es también lineal.

### 2.4.5 CPLEX

CPLEX es una herramienta de resolución de problemas de optimización proporcionada por la empresa IBM. Al proporcionarle a su software package la representación de un problema, con sus respectivas restricciones, éste nos devuelve una solución óptima. Es por tanto una herramienta muy potente, puesto que sustituye el uso de heurísticas o simples reglas con la resolución matemática (en problemas complejos, difícilmente calculable) de los modelos. CPLEX permite resolver una amplia variedad de clases de problemas, pero nos centraremos en los de tipo ILP dado que esa es la clase a la que pertenece el problema del *Minimum Positive Influence Dominant Set* que pretendemos resolver.

En esta práctica, programaremos en C++, para el cual CPLEX incorpora Concert, una librería que nos permite crear objetos para el modelaje y otros para la resolución del problema. Nosotros haremos uso de los siguientes: IloEnv, IloMdel, IloNumVarArray, IloObjective y finalmente IloExpr.

***IloEnv*** nos permite crear un entorno (en inglés, environment), en el que añadimos una serie de modelos con sus respectivas variables y restricciones.

***IloModel*** crea un modelo asignado a un entorno, y nos permite rellenarlo con variables, restricciones, funciones objetivo, etc.

***IloNumVarArray*** es el equivalente a una array normal, donde cada elemento es una variable. Para su inicialización es necesario especificar a qué entorno pertenece, cuál es el mínimo y el máximo valor que pueden tomar las variables y finalmente, de qué tipo son. Cabe decir que el rango es inclusivo (tanto para el mínimo como para el máximo). Y respecto a las variables, pueden ser o bien enteros (ILOINT), continuas o reales (ILOFLOAT) o también booleanas (ILOBOOL).

***IloObjective*** nos permite crear un objeto que represente una función objetivo que hay que minimizar (IloMinimize, la que usaremos) o bien maximizar (IloMaximize). En ambos casos hay que pasarle el entorno al que pertenecen y una expresión. Ésta puede ser directamente una ecuación, o bien una expresión de tipo IloExpr.

***IloExpr*** crea dinámicamente una expresión, a la cual añadimos variables**.**

# 3 Comprobación de Conjuntos Dominadores de Influencia Positiva

## 3.1 Comprobación de Conjuntos Dominadores de Influencia Positiva

Para comprobar que un conjunto de nodos sea dominador de influencia positiva, tal que al menos la mitad de los vecinos del grafo original formen parte del conjunto. Para hacer tal comprobación necesitamos iterar sobre todos los nodos del grafo original y mirar los vecinos que tiene cada nodo. En caso que el nodo vecino esté en el subconjunto, incrementaremos un contador, y cuando acabemos de explorar los nodos de la solución, si esté contador es menor que el número de vecinos que tiene el nodo del grafo original entre dos, entonces estará incumpliendo la condición de ser un conjunto dominador de influencia positiva y consecuentemente devolverá falso.

Una vez vistos todos los nodos del grafo original, si ninguno ha devuelto falso, podremos asegurar que el conjunto es dominador de influencia positiva.

Su complejidad temporal es $O(|V| * d(v))$, donde |V| es el número de nodos que hay en el grafo original, y d(v) es el grado del nodo con grado máximo del grafo original. En el caso peor (caso en el que el grafo original sea completo) tenemos que $d(v) = |V| - 1$por lo que la complejidad temporal será de $O(|V|^2)$. El coste de la función find en un unordered set en caso medio es constante, por lo que no influye en gran medida en el coste total. Así pues, cuando acabe de ejecutarse el algoritmo habremos iterado una vez por cada nodo y dos veces sobre todas las aristas.

Adjuntamos el pseudocódigo para facilitar la comprensión:

```
check_PIDS(solution) {
    for all neighbors_of_n in neighbors do:
        count = 0
        for all node in neighbors_of_n do:
            if node in solution:
                count ++
        if count < neighbors_of_n.size/2:
            return false
    return true
}
```

## 3.2 Comprobación de Conjuntos Minimales Dominadores de Influencia Positiva

Para que el conjunto sea minimal dominador de influencia positiva, primero habrá que comprobar que sea dominador de influencia positiva, y posteriormente comprobar que sea minimal.

Para el primer paso podemos utilizar la comprobación del apartado anterior en la que hemos explicado cómo comprobar que un conjunto sea dominador de influencia positiva. La dificultad reside pues, en demostrar que sea minimal. Por definición un grafo es minimal si no puede reducirse más, es decir, no se puede eliminar más nodos del grafo sin que deje de ser dominante. Como el hecho de ser un conjunto dominante de influencia positiva ya implica ser un conjunto dominante, podemos usar la función del apartado anterior también para comprobar que sea minimal.

En este apartado hemos hecho dos soluciones. La primera es más ineficiente en tiempo de ejecución que la segunda. En cuanto a la primera, hemos traducido exactamente las palabras de la definición de ser un conjunto dominante minimal de influencia positiva, de forma que eliminamos un nodo,

comprobamos que sea PIDS y si lo sigue siendo aún eliminando ese nodo, entonces sabemos que no es minimal. Posteriormente volveríamos a añadirlo en el conjunto. Este algoritmo (check_MPIDS) depende del coste de la función check_PIDS que hemos dicho anteriormente que tiene coste $O(|V| * d(v))$, y cómo además iteramos sobre cada nodo del subconjunto, llegaríamos a tener un coste temporal de $O(|v| * |V| * d(v))$, lo cuál en el peor caso, llegará a ser $O(|V|^3)$, ya que d(v) puede llegar a ser |V|-1 y |v| puede llegar a ser |V| en el peor de los casos.

Para mejorar esta solución, creamos una nueva función aprovechando una estructura que habíamos creado llamada neighbors_popularity, un vector de enteros, la cuál se explica posteriormente en el apartado de algoritmos voraces. Antes de llamar a check_MPIDS_v2, se llena el vector con el número de vecinos que tiene cada nodo de la solución en la solución. Para analizar su complejidad temporal necesitamos saber la complejidad temporal de la función del apartado anterior. Como ya hemos dicho anteriormente, la función tenía coste $O(|V| * d(v))$. En esta función o haciendo una pre computación, calculamos para cada nodo del grafo original, el número de nodos vecinos que pertenecen al subconjunto. Esto tiene el mismo coste que comprobar si el subconjunto es PIDS: $O(|V| * d(v))$ ya que para cada nodo, hay que contar cuántos de sus nodos adyacentes pertenecen al subconjunto y guardarlo en un vector. Finalmente, el último paso es iterar sobre todos los nodos del subconjunto y comprobar que eliminando este, ninguno de sus vecinos baje del 50% de vecinos en el subconjunto. Como previamente hemos guardado para cada nodo, el número de vecinos que tiene que pertenecen al subconjunto, solo hay que comprobar que este número menos 1, sea mayor o igual al número de vecinos de este nodo entre 2. Guardamos en un contador el número de veces que esto ocurre y, finalmente, si este contador es igual al número de vecinos del nodo del subconjunto, podemos asegurar que al quitar este nodo del subconjunto, seguiría siendo PIDS, por lo que no sería MPIDS. Esto tiene un coste temporal inicial de $O(|V| * d(v))$ y después $O(|v| * d(v))$donde |v| es el número de nodos que hay en el subset. como la primera parte tiene un coste mayor, obtenemos que el coste total del algoritmo es $O(|V| * d(v))$.

El pseudocódigo equivalente para la demostración es el que adjuntamos a continuación para facilitar la comprensión de la demostración. En el if, habría la llamada a la función check_PIDS:

```
check_MPIDS(solution) {
    for all node in solution do:
        if solution - {node] is a dominant set:
            return false
    return true
}
```

Este es el pseudocódigo de la segunda solución que hemos nombrado anteriormente, que disminuye el coste temporal del algoritmo:

```
check_MPIDS_v2(solution) {
    for all node in solution do:
        if solution - {node] is a dominant set:
            return false
    return true
}
```

# 4 Algoritmos Voraces

## 4.1 Algoritmo Propuesto

Por definición sabemos que un algoritmo voraz es aquel algoritmo que consiste en elegir la opción óptima en cada paso local, con la esperanza de poder llegar a una solución óptima general. Para el desarrollo de esta solución .

Con esa definición llegamos a la conclusión que la mejor opción para hacer un algoritmo voraz era empezar ordenando los nodos de forma decreciente según su grado. Así pues tendremos en la posición 0 el nodo con más vecinos, y consecuentemente hasta el último nodo del grafo. Para tal ordenación, usamos la función *sort* que nos proporciona C++, que implementa un algoritmo merge sort con coste $O(n \cdot \log n)$. Hemos estudiado alternativas como el counting sort aunque al no ser el cuello de botella, la diferencia entre ambas alternativas era insignificante. Por ello, hemos optado por el merge sort, ya que mejora la legibilidad del código. En nuestro caso, hemos elegido una estructura de datos para almacenar el grafo es un vector de unordered_set, donde el índice del vector representa el nodo y el unordered_set representa el conjunto de los vecinos. Por ello, no podemos alterar el orden de esta estructura de datos y para solucionar esto, en vez de ordenar este vector, creamos otro vector en el cual guardamos los índices de los nodos ordenados, y para ordenarlo usaremos una función "compare", que comparará el número de vecinos que tienen los nodos entre ellos, para así tener una lista ordenada de forma decreciente según su grado.

Una vez ordenados los nodos según su grado, empezaremos con el algoritmo voraz. Para ello necesitaremos una estructura de datos adicional (llamada neighbors_popularity) en la cuál cada posición representa un nodo, guardaremos, para cada nodo, el número de vecinos de ese nodo que están en la solución. Inicialmente estará inicializado a 0 para todas sus posiciones, ya que la solución empezará siendo vacía.

En el algoritmo voraz iteramos sobre todos los nodos ordenados. Antes de empezar a iterar, para optimizar la solución, iteramos sobre todos los nodos con ningún vecino (ya que no nos interesan) y sobre todos los nodos con un sólo vecino, como consecuencia de tener un sólo vecino, éste vecino deberá estar obligatoriamente en la solución para poder ser un conjunto dominador de influencia positiva. Ésto se debe a que, por definición de conjunto dominador de influencia positiva, un nodo debe tener en la solución al menos un 50% de sus vecinos, y como el nodo solo tiene 1 vecino, éste deberá estar en la solución. Posteriormente, para cada nodo del vector ordenado, llamado nodo N, miraremos, mediante una función auxiliar llamada ***check_adjacent_nodes***, si el número de vecinos de N aún no tiene al menos la mitad de sus vecinos en la solución, entonces añadiremos el nodo N a la solución e incrementaremos en 1 el número de vecinos en la solución (en la estructura de datos mencionada anteriormente), para cada uno de los vecinos de N. Una vez recorrido todo el vector ordenado, el algoritmo nos devolverá una solución inicial que podremos utilizar para la búsqueda local posteriormente.

Cuando sacamos la primera solución vimos que podíamos exprimir aún más la solución, ya que podíamos disminuir el número de nodos. Para llevar a cabo tal optimización, añadimos una función auxiliar nueva llamada ***remove_nodes*** que para cada nodo del grafo, mira si esta en la solución, y en caso que esté mirará si se puede quitar con una función llamada ***can_remove*** cuya función es mirar que si quitamos un nodo de la solución, que sus vecinos no dejen de cumplir el requisito de PIDS. Para usar esta función, ordenamos los nodos de la misma manera que antes pero ahora en orden

creciente. Una vez eliminados los nodos que podamos eliminar, encontraremos una solución cercana más próxima a la solución óptima global.

## 4.2 Pseudocódigo

Adjuntamos pseudo-código para facilitar la comprensión del algoritmo y de su función auxiliar:

```
check_adjacent_nodes(neighbors[top]) {
    for all node in neighbors[top]
        if neighbors_popularity[node] < neighbors[top].size/2:
            for all node in neighbors[top] do:
                increment neighbors_popularity[node] by 1
            return true
    return false
}

can_remove(neighbors[n]) {
    for all node in neighbors[n] do:
        if neighbors_popularity[node] - 1 < neighbors[node].size / 2
            return false
    for all node in neighbors[n] do:
       decrement neighbors_popularity[node] by 1
   return true
}



remove_nodes(solution) {
    sort neighbors by degree in ascending order
    for all n in neighbors do:
        if n is in solution:
            if can_remove(neighbors[n]):
                solution = solution − {n}
    return solution
}

Greedy {
    solution = {}
    for all nodes, set neighbors_popularity  to 0
    sort neighbors by degree in descending order
    position = neighbors.size - 1
    for all n in neighbors with no neighbors do:
        position--
    for all n in neighbors with one neighbor do:
        node = neighbor of n
        solution U {node}
        for all neighbor_node in neighbors[node] do:
            increment neighbors_popularity[node] by 1
        position--
    top = 0
```

```
    for all n in neighbors do:
        if check_adjacent_nodes(neighbors of top, neighbors_popularity):
            solution = solution U {n}
    remove_nodes(solution)
    return solution
}
```

## 4.3 Alternativas Vistas

### 4.3.1 Wang's Greedy

Wang introdujo este problema en su tesis doctoral. Esta trataba diferentes algoritmos que se podían usar para abordar el problema de la influencia positiva en redes sociales. Usualmente, propuso Wang los algoritmos trataban de sets dominadores de influencia positiva (PIDS), pero estos usaban algoritmos voraces ineficientes para el ratio de crecimiento de las redes sociales. Por lo tanto, propone una nueva alternativa que permitiese diferentes pesos de influencia, el conjunto dominador de influencia positiva con pesos (WPIDS). Propone un nuevo algoritmo auto estabilizador de tiempo $O(n^3)$.

### 4.3.2 Raei's Greedy

Propone una optimización del algoritmo que funciona en tiempo cuadrático $O(n^2)$. En su estudio hace también un análisis teórico y una simulación de los resultados para verificar la eficiencia, que demuestra que el uso de este algoritmo reduce significativamente el tamaño del PIDS relativo al algoritmo de Wang. This algorithm optimizes those previously used by using a greedy heuristic method.

### 4.3.3 Fei's Greedy

Un algoritmo basado en Wang's Greedy con una alteración usada también en Raei's greedy. Fei propone usar la estrategia de romper empates que usa Raei en su algoritmo. Este como los últimos dos vistos resulta muy ineficiente con grafos más grandes.

### 4.3.4 Pan's Greedy

Pan propone un algoritmo con heurística rápida que baja la complejidad temporal del algoritmo a $O(n * logn + m)$. Este algoritmo mejora los previamente vistos tanto en tiempo computacional como en calidad de la solución. Hace esto ordenando los vértices de forma ascendente en grado de incidencia previamente a empezar el algoritmo.

### 4.3.5 Boumama's Improved Greedy

Propone un cambio a Pan's Greedy para mejorar la complejidad temporal. Este usa un algoritmo de pruning para podar el grafo de nodos que no son interesantes inicialmente en la solución. También proponen una nueva heurística que asimila dos heurísticas (*cover degree* y *need degree*), donde en el de Pan solo se usa *cover degree*. Posteriormente se quitan los vértices repetidos.

# 5 Búsqueda Local

## 5.1 Algoritmo Propuesto

Búsqueda Local es un tipo de búsqueda heurística donde no encontraremos el máximo global, si no el máximo local desde la posición en la que empezemos. Hay varios métodos de búsqueda local, en este estudio hemos decidido centrarnos en la búsqueda local determinista y la búsqueda local estocástica. Para la determinista hemos decidido usar el algoritmo ***Hill Climbing*** y para la estocástica hemos

decidido usar el algoritmo ***Simulated Annealing***. El algoritmo de hill climbing está implementado con ***best-improvement***, ya que hemos planteado la solución inicial como greedy mejorar esta puede llegar a ser muy complicada y nos resultó mejor hacer el best-improvement. Hemos decidido que hacer estos dos sería la mejor opción para visitar un amplio rango de soluciones. La solución inicial, heurística y operadores para cada uno se mantendrá igual.

### 5.1.1 Solución Inicial

En la solución inicial nos proponemos varias soluciones y comparamos estas entre sí. Inicialmente nos proponemos empezar con un set de todos los vértices como solución inicial. Vemos que esta solución sería siempre una solución inicial de un conjunto dominador positivo ya que cada nodo está conectado a un 100% de nodos del conjunto. Esto nos propone un reto muy grande para la heurística. Dado que nuestra intención es minimizar el número de nodos del conjunto de soluciones, la heurística no solo debería considerar el número de nodos sino la optimalidad posterior que tendría quitar uno de estos nodos. Esto nos pareció una solución inicial demasiado compleja para la ejecución de una búsqueda local así que decidimos no progresar con ella.

Decidimos comparar dos soluciones iniciales, esas mencionadas en la descripción de la práctica. Por lo tanto, nos proponemos dos soluciones iniciales nuevas.

Empezar con una solución randomizada nos propone ciertas ventajas sobre cualquier otra solución inicial, esto es dado a que cómo con muy baja probabilidad nos saldrán dos conjuntos iguales siempre estaremos buscando un punto local mínimo en diferentes sitios del espacio de soluciones. De esta manera podremos ver resultados diferentes, y en ciertos casos mejores. Pero también nos supone un reto, y un impedimento. En la búsqueda local tenemos que empezar con una solución y la random nos dará una no-solución con una probabilidad muy alta. En el caso que no empezáramos por una solución los operadores nos deberían devolver a esta con cierta rapidez, que tampoco es el caso. Por lo tanto asumimos que esta tampoco es una solución viable.

Empezar con la solución greedy nos propone otro reto interesante que supone minimizar una solución que en muchos casos ya supone una solución mínima (global o local). Por lo tanto nuestro algoritmo para este caso debería no solo minimizar nuestra solución inicial sino explorar otros espacios (como haría el simulated annealing) para ver la optimalidad de esta solución en un espectro global. En el hill climbing seguramente veremos que se nos queda en la misma solución (o una muy parecida a la que encontramos en nuestra solución greedy).

### 5.1.2 Heurística

La heurística más simple en la que pensamos fue por el tamaño del conjunto resultante. Esto no acabará resultando como una buena heurística ya que el algoritmo cogería la primera solución que minimizará el tamaño del vector. En muchos casos esta no sería la solución además nos forzará a mirar si un conjunto es pids en cada uso de operadores. Esta heurística también es incorrecta ya que no permite el uso del operador ***switch*** cambiar un nodo de la solución por uno que aún no está en la solución. Esto, específicamente en el caso de la inicialización en greedy nos puede ser muy beneficioso.

Por lo tanto investigamos otras opciones de heurística. Optamos por una heurística que intente minimizar el número de nodos del subconjunto incidentes para cada nodo, penalizando esos nodos que tienen menos del 50% de incidencia de nodos del subconjunto. De esta forma, habilitamos el

operador ***switch*** y nuestra heurística no depende únicamente del tamaño de nuestra solución si no que también depende de cómo minimizar este subconjunto para obtener un grafo minimal.

### 5.1.3 Operadores

Inicialmente propusimos usar dos operadores ***addNode*** y ***removeNode.*** Estos parecían suficientes para analizar todo el espacio de soluciones ya que un switch acaba siendo quitar un nodo y añadir un nodo. Vimos que esto no era precisamente así. Cuando empezamos con una solución que ya está bastante minimizada, este

El factor de ramificación del operador ***addNode*** es de $|V|$, ya que este puede coger cualquier nodo del grafo con el que trabajamos y decidir añadir este en el subconjunto si no estaba previamente. Donde V es el conjunto de nodos del grafo total.

El factor de ramificación del operador ***removeNode*** es de $|S|$ esto es porque solo podemos quitar esos nodos que estén en la solución. Donde S es el subconjunto de nodos que tenemos como conjunto dominador de influencia positiva.

El factor de ramificación del operador ***switchNodes*** ed de $|V| * |S|$. Esto es porque tenemos que cambiar un nodo que aún no está en la solución (como era el caso de ***addNode***). Por un nodo que ya está en el subconjunto de la solución. Donde V es el conjunto de nodos del grafo total y S es el subconjunto de nodos que tenemos como conjunto dominador de influencia positiva.

## 5.2 Pseudocodigo

Para facilitar la legibilidad del pseudocódigo este está roto en bloques. Además como ciertas características se repiten entre algoritmos es más legible que repetir el código varias veces. La solución inicial no está incluida puesto que para la random usamos la clase dada en el código base y para la greedy usamos el código visto en el 4.2.

### 5.2.1 Heurística

```
heurística
    if set hasn't been modified then
        return the same heuristic value
    end if

    if node has been added to solution then
        add size of outgoing edges

        for each neighbor of added edge
            if previously had less than half of nodes in subset but now doesn't then
                remove size of nodes previously added to heuristics
            end if
        end for
    end if
    else if node has been removed from solution then
        remove size of outgoing edges
            for each neighbor of removed edge
            if node now has less than half of the incoming edges in the subset
```

```
                    add size of nodes
                end if
            end for
        end if
        else if nodes have been switched from solution then
            remove size of outgoing edges

            for each neighbor of removed edge
                if node now has less than half of the incoming edges in the subset
                    add size of nodes
                end if
            end for
            add size of outgoing edges of node2
            for each neighbor of added edge of node2
                if previously had less than half of nodes in subset but now doesn't then
                    remove size of nodes previously added to heuristics
                end if
            end for
        end if
```

***main***

```
        ...
        coger solución inicial
        // inicializar valores heurísticos
        for all nodes (n), get connected nodes
            initialize count to 0
            for all connected nodes (n1)
                if n1 is in initial set then
                    add one to count
                end if
            end for
            if count is less than half the size of the connected nodes then
                add the amount of nodes to heuristic value
                // it's very unlikely to obtain such a large solution but still feasible
            end if
        end for
        ...
```

### 5.2.2 Operadores

***addNodeToSolution***

```
        if node is not in dominating set then
            set node1 to node
            set modified to true
            set op to "add"
            insert node into solution
            return true
        end if
        return false
```

***removeNodeFromSolution***

```
if node is in dominating set
        set node1 to node
        set modified to true
        set op to "remove"
        remove node from solution
        return true
end if
```

***switchNodesInSolution***

```
if node is in dominating set
        set node1 to n1
        set node2 to n2
        set modified to true
        set op to "switch"
        add node1 to solution
        remove node2 from solution
        return true
end if
```

### 5.2.3 Hill Climbing

***findNeighborsHillClimbing***

```
for each node in graf
        if addSolution returns true then
                add heuristic to heuristics vector
                add solution to neighbors vector
        end if

        for each node in dominating set
                if switchNodes returns true
                        add heuristic to heuristics vector
                        add solution to neighbors vector
                end if
        end for
end for

for each node in dominating set
        if removeNode returns true
                add heuristic to heuristics vector
                add solution to neighbors vector
        end if
end for
```

***hillClimbing***

```
while a better alternative is found do
        find all neighbors
        for every neighbor found
                if heuristics is better then
                        set dominating set to new set
                        set heuristics to newHeuristics
                end if
        end for
end while
```

### 5.2.4 Simulated Annealing

***nextNeighborSimulated***

```
randomize x from 0 to 2
initialize a boolean correcto to true
if x is equal to 0 then
        randomize n from 0 to number of nodes in graf
        if addNode returns false then
                change correct value to false
        end if
end if
else if x is equal to 1 then
        randomize n from 0 to number of nodes in the solution
        if removeNode returns false then
                change correct value to false
        end if
end if

otherwise then
        randomize n1 from 0 to number of nodes in graf
        randomize n from 0 to number of nodes in the solution
        if switchNodes returns false then
                change correct value to false
        end if
end if

if not correct then
        return initial set value
end if
return aux
```

```
simulatedAnnealing
    initialize min to max int value
    initialize current solution to greedy

    while T > Tmin do
        for 0 to numIterations
            if current less than min then
                set min solution to current
            end if

            find new solution
            calculate probability ap that the current solution is set

            if ap is greater than a random number
                set current to new solution
            end if
        end for

        multiply T by alpha
    end while
```

# 6 Metaheuristica

## 6.1 Algoritmo Propuesto

En nuestro caso, la metaheurística que utilizaremos estará basada en el ***tabú local search***. Este método es parecido al Hill Climbing visto en el apartado de Búsqueda Local. Se diferencia de este algoritmo en que el tabú usa diferentes estructuras de datos para poder salir de óptimos locales y conseguir óptimos globales o óptimos mejores que el local en el que estamos. Esta estructura de datos guardará movimientos *tabú*, una vez aplicado un operador sobre un nodo, se guardará en la estructura para no repetir el operador sobre este nodo en un número de iteraciones determinado.

El algoritmo consiste en un bucle que acabará cuando el tiempo propuesto pase. Dentro del bucle, para cada nodo aplicaremos los operadores cuando sea posible (ya que si un nodo ya está en la solución, no podemos volver a meterlo y lo mismo con Delete) y nos guardaremos la mejor solución encontrada en esa iteración, que será el siguiente estado, además de guardar la acción realizada en la lista tabú. Como vemos, este estado no tiene por que ser mejor que el mejor estado encontrado hasta el momento, y justamente esa es la gracia de la búsqueda tabú, ya que nos permite explorar soluciones que a priori son peores pero que pueden llevar al óptimo global.

### 6.1.1 Solución Inicial

La solución inicial de la metaheurística admite la misma discusión que esa de la búsqueda local. Además de esta podemos visitar la idea de usar un algoritmo de búsqueda local para precomputar un máximo local del espacio de soluciones. Como veremos más adelante la búsqueda local no nos da resultados eficientes que mejoran significativamente la solución voraz que obtenemos del cuarto apartado. Por lo tanto, también podemos descartar esta como solución inicial. Por lo tanto, del mismo modo que lo hemos hecho para Búsqueda Local, cogeremos la solución inicial de este como la greedy.

### 6.1.2 Heurística

La heurística utilizada en este apartado usará el tamaño de nuestra solución y un porcentaje extraído de la cantidad de nodos que están conectados a nuestra solución.

### 6.1.3 Operadores

En nuestro caso, tenemos 2 operadores: Add (añade un nodo al subconjunto solución) y Delete (elimina un nodo del conjunto solución). Para guardar los movimientos *tabú* utilizamos unordered_set como estructura de datos ya que el acceso en media es constante.

## 6.2 Pseudocodigo

### 6.2.1 Heuristics

***compute_percentage_neighbors***

```
initialize percentage to zero
for every node in the graph
        for every adjacent node do
                count++
        end for
        if size of adjacent is greater than 0 then
                add percentage divided by size
        end if
end for
```

***computeHeuristics***

```
solution size * factor + score
```

### 6.2.2 Operadores

***canDelete***

```
for each node in graf
        if neighbors_popularity is less than half of the neighbors size
                return false
        end if
end for
return true
```

***addNode***

```
insert node to solution set
for every adjacent neighbor
        add one to neighbor popularity
        increase percentage by 1/adjacent nodes
end for
```

***deleteNode***

```
delete node from solution set
for every adjacent neighbor
        remove one from neighbor popularity
        decrease percentage by 1/adjacent nodes
end for
```

### 6.2.3 Metaheuristica

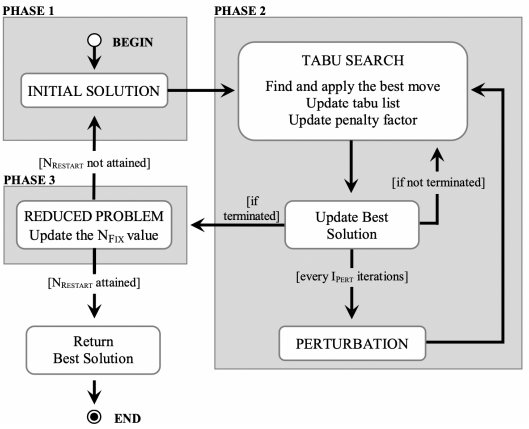
***tabu search***

```
initialize values
while not time_limit_exceeded do
        for all nodes in graf
                add node to solution
                if valid node to add
                        check values
                        if add node is better than local maximum
                                set new solution as local maximum
                        end if
                end if
        end for

        for all nodes in graf
                add node to solution
                if valid node to remove
                        check values
                        if remove node is better than local maximum
                                set new solution as local maximum
                        end if
                end if
        end for
end while
```

## 6.3 Alternativas Vistas

Para expandir este apartado hemos visto que podríamos usar Integer Linear Programming [11]. El algoritmo propuesto por este estudio propone usar una solución randomizada. Esta se optimiza mediante el algoritmo de búsqueda local ***tabú*** y mecanismos de perturbación que sirven para reducir el problema que posteriormente se solucionará con ILP. Este proceso se repite varias veces (Nrestart). Cuando INI iteraciones se han hecho sin mejorar los resultados obtenidos, entonces se para la ejecución.




# 7 ILP con CPLEX

Finalmente, para la resolución de éste problema de optimización también hemos utilizado la herramienta CPLEX. Como hemos explicado anteriormente, nos permite (entre otras cosas) encontrar la solución óptima en problemas de minimización o de maximización. Simplemente hay que representar el problema como una serie de variables con sus respectivas restricciones y CPLEX devuelve una serie de modelos que satisfagan el problema. En más o menos tiempo, dependiendo de la complejidad del problema propuesto, la herramienta es capaz de encontrar soluciones óptimas.

## 7.1 Representación del problema

Podemos dividir la representación problema en tres partes. Tenemos por un lado la representación del grafo, y por otro, la representación de las variables y sus restricciones.

La representación del grafo es trivial; con conocer el número de nodos, el número de adyacencias y cuáles son dichas adyacencias tenemos más que suficiente. Además, al tratarse de un grafo conexo, no dirigido y sin pesos, no nos hace falta información adicional sobre las aristas, y las estructuras de datos necesarias para su representación se simplifican.

Para el número de nodos y de aristas usamos enteros, y almacenaremos las adyacencias en un vector de sets de enteros. Cada posición representa un nodo, y el set contiene sus nodos vecinos, identificados por un entero. La estructura es entonces, la siguiente:

***vector< set<int> > neighbours***

En segundo lugar, tenemos las variables. Como hemos mencionado anteriormente, éstas solamente pueden adquirir un valor entero, indicando a qué conjunto pertenecen. E incluso podemos afinar más puesto que solo hay dos opciones posibles como valores que puede tomar cada nodo en un modelo del problema: o bien pertenece al conjunto dominante de influencia positiva, o no lo hace. En otras palabras los únicos valores que puede tomar cada variable (o nodo) en nuestra solución no solamente son enteros, si no que además también son binarios. Un cero representa que el nodo no pertenece al conjunto dominante, y un 1 que sí que lo hace.

Por lo que a las restricciones respecta, simplemente se debe cumplir para cada nodo que al menos la mitad de sus vecinos forme parte del conjunto dominante de influencia positiva. Puesto que hemos representado nuestro grafo como un vector de adyacencias, esta tarea es de lo más sencilla. Si, en un nodo, el tamaño del set de sus respectivos vecinos que pertenece al conjunto dominante es o supera la mitad, se cumple la restricción. Si ocurre en todos los nodos no pertenecientes al conjunto dominante, se considera válido el modelo.

Es importante tener en cuenta que para un número impar de nodos vecinos, por ejemplo 3, no podemos poner como restricción que al menos “un nodo y medio” pertenecen al conjunto dominante. Además en tipo entero, que es como representamos el número de vecinos, la restricción para el caso anterior pasaría a ser al menos un nodo. Esto, aunque lógico, tampoco cumpliría correctamente la restricción, ya que un tercio no supera la mitad. Para un número de vecinos impar pues, la restricción será de la mitad (entera) más uno.

## 7.2 Llamada a CPLEX

Ahora que quedan aclaradas las estructuras de datos y la representación del problema, pasamos a describir cómo proporcionarle esta información a CPLEX para que éste pueda encontrarnos un modelo óptimo.

El primer paso es crear un *environment* o entorno, que pertenece a una de las clases de CPLEX; la llamada *IloEnv*. Es este el objeto donde añadiremos nuestros modelos de optimización. Para este problema, solamente tenemos un modelo que creamos mediante la clase *IloModel*, y al cual le asignamos el entorno creado.

Ahora que hemos creado un modelo para el entorno, empleamos la estructura *IloNumVarArray* para almacenar una variable por cada nodo en el grafo. Para hacerlo, especificamos el entorno al cual pertenece, el número de variables que tendrá la estructura, qué valores puede tomar cada una y finalmente de qué tipo son. En nuestro caso, este objeto al que llamaremos "varArray" queda de la siguiente manera:

**IloNumVarArray varArray(entorno, número de nodos, 0, 1, ILOINT)**

Nótese que para especificar qué valores puede tomar cada variable, especificamos el valor mínimo y el máximo. Por otro lado, *ILOINT* especifica que son de tipo entero. Como hemos explicado antes, las variables de IloNumVarArray también pueden ser de tipo booleano, por lo que en este caso donde las variables pueden tomar solo valores binarios, los tipos son intercambiables. Escogemos ILOINT por defecto, y porque de escoger ILOBOOL habría que especificar el rango igualmente, de modo que tampoco ganaríamos nada.

Para terminar de proporcionarle los datos necesarios a CPLEX, debemos crear un objeto que contenga todas las variables de *varArray*. Se trata de una expresión, que en CPLEX se declara como un objeto de tipo *IloExpr*. Hay que asignarlo al entorno adecuado y pedirle que minimice el modelo creado. Para hacerlo, crearemos un objeto *IloObjective* de tipo *IloMinimize*, que se encarga de minimizar la función objetivo, es decir el número de nodos en el conjunto dominante, o variables a 1 en el modelo.

Por último, debemos crear las restricciones que debe cumplir cada nodo de nuestro grafo, y lo haremos creando una expresión para cada uno. En cada expresión nos aseguramos de que esté asignada al entorno correcto, y después añadimos como variables aquellos nodos vecinos al que estamos visitando, y finalmente establecemos la restricción que deben cumplir y lo añadimos al modelo.

Con esto, el modelo queda terminado, y solo falta crear un objeto de tipo *IloCplex* al que asignársele, y llamar a la herramienta de resolución mediante el método solve() de la clase.

### 7.3 Resultados

Una vez resuelto, CPLEX nos devolverá los modelos encontrados, así como información sobre si ha encontrado o no una solución óptima (método getStatus()), dentro de un marco de tiempo.

Entre los resultados obtenidos, los que nos interesan son por una parte el número de nodos en la solución, y por otra el tiempo que ha tardado CPLEX en dar con ella. Obtenemos esta información mediante los métodos elapsed_time() de la clase Timer y getObjValue() de la clase IloCplex, respectivamente. Éstos son los datos que recopilaremos más adelante, durante la fase de experimentación.

## 8 Experimentación

Antes de explicar los experimentos, cabe mencionar que éstos se han llevado a término mediante un ordenador Macbook pro de 2020, con un chip M1 con 16GB de RAM.

### 8.1 Experimentación con algoritmo voraz

Vemos que con nuestro algoritmo voraz, obtenemos unas soluciones muy razonables y bastante parecidas a las que hay publicadas en la página web de la asignatura.

En la primera instancia vemos que obtenemos un número de 70 nodos, en los experimentos de la página web (en el mejor de los casos de los algoritmos voraces) obtienen 68 nodos. En cuanto al tiempo depende de la máquina en la que lo ejecutemos, pero vemos que el tiempo es prácticamente de cero segundos.

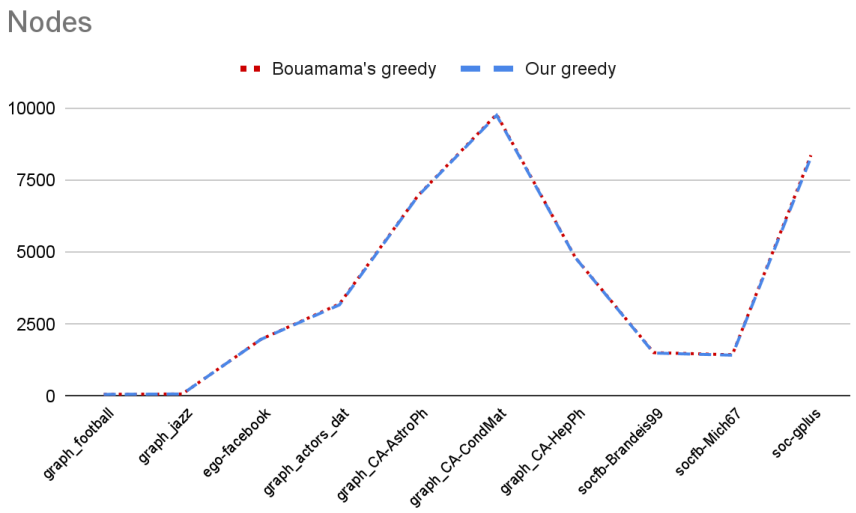
En la segunda instancia graph_jazz, obtenemos 82 nodos y un tiempo de 0.0048s, mientras que los resultados publicados obtienen 81 nodos. Vemos que la diferencia entre sus resultados y nuestros resultados es prácticamente nula.

Si seguimos comparando los resultados veremos que en el caso en el que hay más diferencia de nodos es en el caso de la instancia graph_CA-CondMat, en la cual obtenemos 9782 nodos y el resultado publicado es de 9748. La diferencia es de 34 nodos. Es una diferencia menor que el 1% del total de nodos, por lo tanto nuestro algoritmo voraz, en cuanto a nodos, obtiene resultados bastante buenos hasta el punto de ser prácticamente iguales que los que hay publicados.

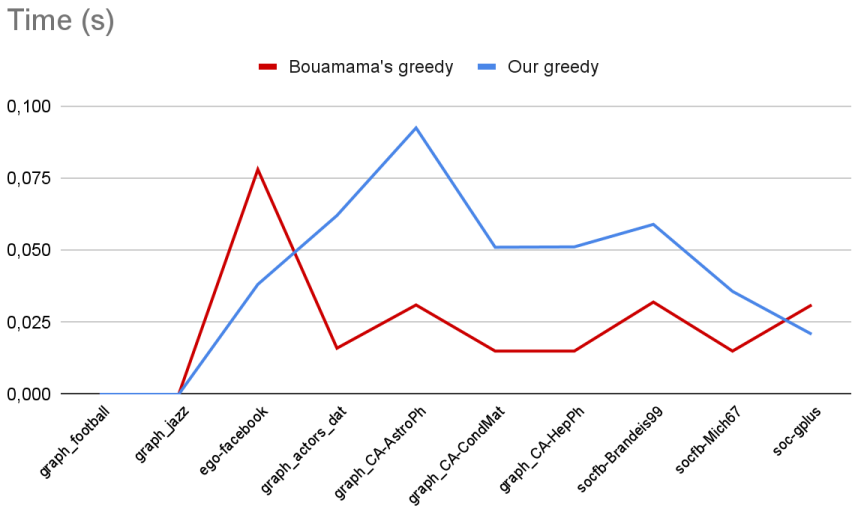
En cuanto al tiempo es difícil de comparar, ya que cómo hemos dicho anteriormente, depende de la máquina en la cual se ejecute el programa, y así pues no podemos compararlos directamente.
Aún así, los tiempos que obtenemos, aún siendo mayores que los que hay publicados, podemos considerar que nuestro algoritmo es considerablemente rápido, considerando que el número de nodos que hay en las instancias es un número elevado.

Por lo tanto, podemos concluir que nuestro algoritmo voraz es un algoritmo que obtiene resultados parecidos a los publicados, con un tiempo ligeramente superior a los publicados, pero aún así válidos.

En el gráfico adjunto vemos que la diferencia que hay entre el número de nodos que se obtiene mediante el algoritmo voraz de Bouamama's (el que obtiene el mejor resultado en todas las instancias menos la última) y nuestro algoritmo es prácticamente irrelevante.




En el siguiente gráfico podemos ver la diferencia en cuanto al tiempo usando las mismas instancias que en el gráfico anterior. Podemos ver que nuestro algoritmo es considerablemente más lento en la mayoría de casos, pero en alguna instancia llega a ser más rápido.




Aquí vemos los resultados de nuestro algoritmo aplicado a las diferentes instancias que se nos han proporcionado tanto para los nodos (Value) como para el tiempo (Time) en segundos.

| Network | Value | Time |
|---|---|---|
| **graph_football** | 70 | 0,000837 |
| **graph_jazz** | 82 | 0,002477 |
| **ego-facebook** | 1977 | 0,038146 |
| **graph_actors_dat** | 3205 | 0,062035 |
| **graph_CA-AstroPh** | 6966 | 0,092425 |
| **graph_CA-CondMat** | 9782 | 0,051019 |
| **graph_CA-HepPh** | 4823 | 0,051147 |
| **socfb-Brandeis99** | 1516 | 0,058954 |
| **socfb-Mich67** | 1443 | 0,035715 |
| **soc-gplus** | 8362 | 0,020872 |

## 8.2 Experimentación con búsqueda local

A continuación realizaremos experimentos de búsqueda local tanto de Hill Climbing como de Simulated Annealing para ver la optimización que aportan estos dos al resultado del greedy. Para el Hill Climbing cómo es determinista solo vamos a ejecutar una vez este algoritmo. En el caso de el Simulated Annealing observamos que nos puede dar resultados distintos por cada ejecución del algoritmo, por lo tanto lo ejecutaremos 10 veces para obtener un resultado que sea más representativo del trabajo del Simulated Annealing.

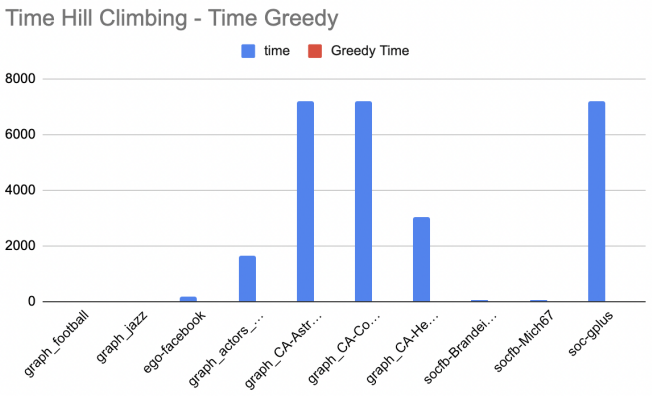
### 8.2.1 Hill Climbing

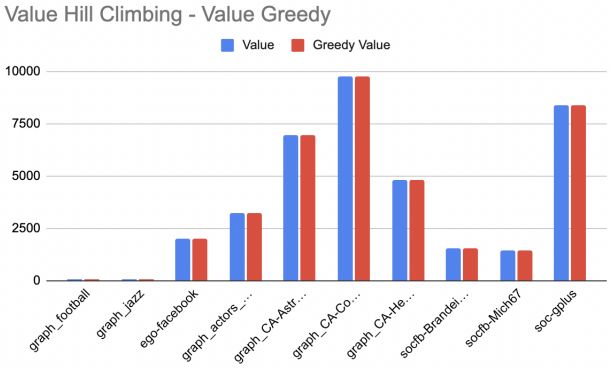
A continuación vemos los resultados obtenidos al ejecutar el Hill Climbing. Como hemos mencionado anteriormente puesto que este algoritmo es determinista solo lo tendremos que ejecutar una vez.

| Network | Value | Time |
|---|---|---|

| | | |
|---|---|---|
| **graph_football** | 70 | 0,01 |
| **graph_jazz** | 82 | 0,02 |
| **ego-facebook** | 1977 | 185,06 |
| **graph_actors_dat** | 3205 | 1656,63 |
| **graph_CA-AstroPh** | 6966 | 7200 |
| **graph_CA-CondMat** | 9782 | 7200 |
| **graph_CA-HepPh** | 4823 | 3048,23 |
| **socfb-Brandeis99** | 1516 | 61,78 |
| **socfb-Mich67** | 1443 | 53,58 |
| **soc-gplus** | 8363 | 7200 |

Como era predecible, los resultados de este experimento dejan mucho a desear. Vemos que no hay ningún cambio significativo comparado con la greedy pero supone un incremento temporal muy grande. Vemos a continuación la comparativa en gráficos.




Vemos que hay un incremento temporal muy importante entre el greedy y el hill climbing pero no hay ninguna diferencia en el valor de los nodos de la solución final. Esto quiere decir que el Hill Climbing puede que encuentre una solución mejor para la heurística que tiene. Pero esto no supone un cambio en el número de nodos de la solución final.

Es interesante mencionar que esto es un resultado esperado ya que hay una probabilidad muy alta de que el greedy de un mínimo local y por lo tanto no nos podamos mover de esta solución inicial.

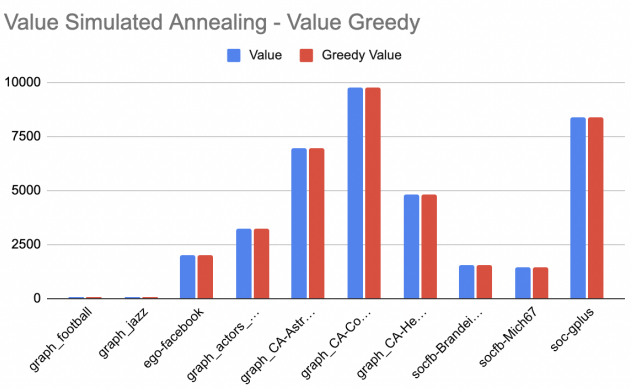
### 8.2.2 Simulated Annealing

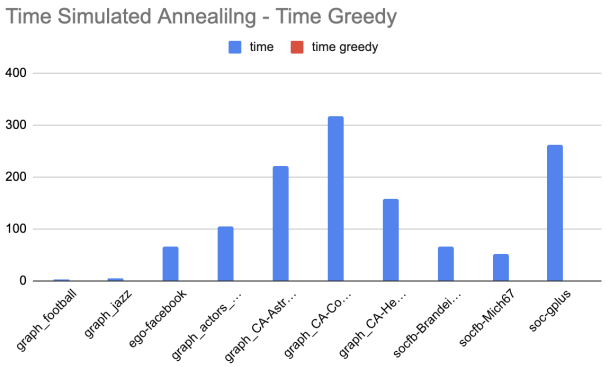
El simulated annealing tiene varias variables a tener en consideración. La temperatura inicial la hemos puesto a 1, la temperatura mínima a 0.0001, el alpha por el que vamos a multiplicar nuestra temperatura a 0.95 y el número de iteraciones a 1000. Hemos considerado que estos valores son suficientemente grandes para llegar a una solución correcta.

A continuación exponemos los resultados obtenidos de la ejecución de este algoritmo.

| Network | Best | Mean | standard dev | time | time standard |
|---|---|---|---|---|---|
| graph_football | 70 | 70 | 0 | 2,79 | 0,06 |
| graph_jazz | 82 | 82 | 0 | 3,7 | 0,03 |
| ego-facebook | 1977 | 1977 | 0 | 65,31 | 0,37 |
| graph_actors_dat | 3205 | 3205 | 0 | 104,68 | 0,59 |
| graph_CA-AstroPh | 6966 | 6966 | 0 | 221,35 | 1,35 |
| graph_CA-CondMat | 9782 | 9782 | 0 | 317,08 | 1,5 |
| graph_CA-HepPh | 4823 | 4823 | 0 | 157,58 | 1,32 |
| socfb-Brandeis99 | 1516 | 1516 | 0 | 65,58 | 2,53 |
| socfb-Mich67 | 1443 | 1443 | 0 | 50,6 | 1,34 |
| soc-gplus | 8363 | 8363 | 0 | 261,92 | 1,12 |

Como vemos, nos vuelve a pasar algo similar al caso de hill climbing donde este algoritmo vuelve a darnos una solución que no optimiza el número de nodos del greedy pero nos retrasa en cuanto a tiempo.




De este algoritmo es menos esperado ya que el simulated annealing tiene la propiedad que con cierta probabilidad va a coger una solución peor que quiere decir que con cierta probabilidad nos cogerá una peor solución para acabar en una mejor solución. Vemos que esto puede ser por el hecho de que no exploramos todo el espacio de soluciones y con cierta probabilidad nunca encontraremos mejor solución.

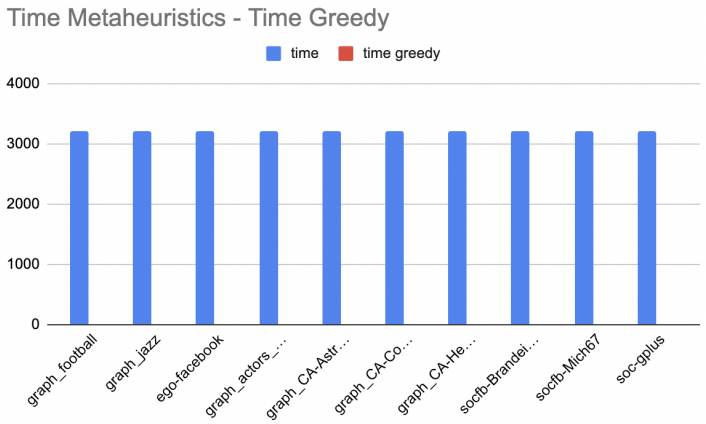
## 8.3 Experimentación con metaheurística

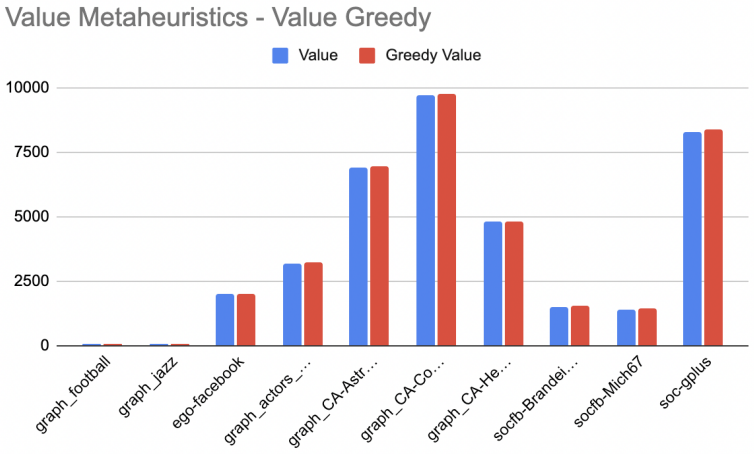
De forma similar al Simulated Annealing la metaheurística no se compone de un algoritmo determinista, por lo tanto para extraer conclusiones deberemos ejecutar el algoritmo diez veces. Y tomar la media de este.

Vemos en los datos obtenidos de la ejecución de nuestra metaheurística de que hay un cambio significativo al ejecutar esta metaheurística. El algoritmo usa suficientes cambios de cambio que encuentra una solución mejor a la que obtenemos previamente usando únicamente la ejecución del greedy (o búsqueda local que hemos visto que era lo mismo).

| Network | Value | Greedy Value |
|---|---|---|
| **graph_football** | 67 | 70 |
| **graph_jazz** | 81 | 82 |
| **ego-facebook** | 1975 | 1977 |
| **graph_actors_dat** | 3171 | 3205 |
| **graph_CA-AstroPh** | 6921 | 6966 |
| **graph_CA-CondMat** | 9712 | 9782 |
| **graph_CA-HepPh** | 4790 | 4823 |
| **socfb-Brandeis99** | 1476 | 1516 |
| **socfb-Mich67** | 1404 | 1443 |
| **soc-gplus** | 8281 | 8362 |

Representamos estas ejecuciones en gráficos a continuación.



Observamos que nuestra metaheurística funciona muy bien para nodos con soluciones más grandes ya que tiene un espacio de soluciones más amplio en el que buscar. También paramos atención al hecho de que todas nuestras desviaciones estándar para la metaheurística sean cero. Esto es porque le damos suficiente tiempo al algoritmo para que siempre encuentre una solución óptima y porque empezamos siempre con la misma solución inicial.

De este modo vemos que nuestra metaheurística es eficiente en cuanto a optimizar la solución inicial. Pero por el tiempo que tarda este algoritmo en encontrar esta solución vemos que no es tan óptima en cuanto a tiempo como la ejecución de simplemente la greedy. Aunque nos acerca más a nuestra solución deseada para encontrar el MPIDS.

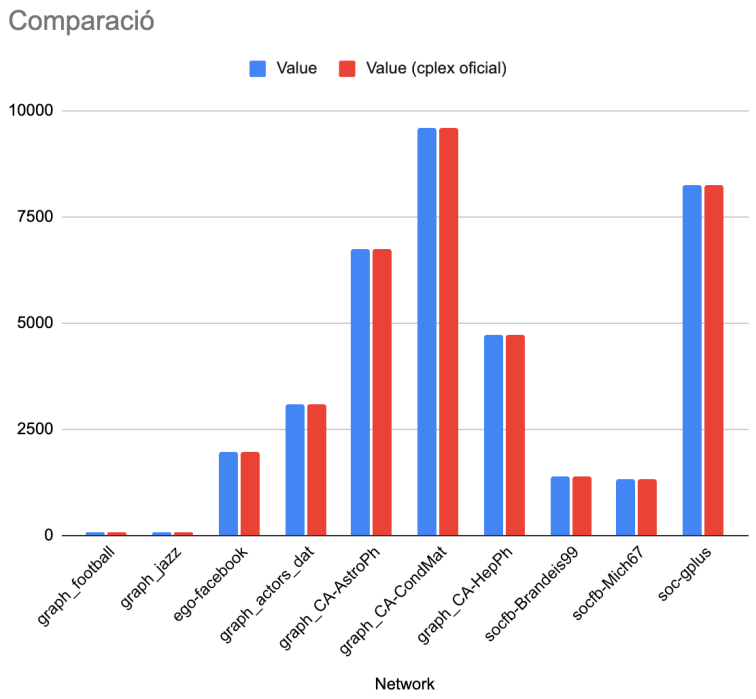
## 8.4 Experimentación con CPLEX

Añadimos a continuación los resultados obtenidos mediante CPLEX. De nuevo, tenemos para cada instancia el número de nodos y el tiempo que el optimizador ha necesitado para encontrar esta solución.

| Network | Time | Value | Comparative Value |
|---|---|---|---|
| **graph_football** | 6,38 | 63 | 63 |
| **graph_jazz** | 0,17 | 79 | 79 |
| **ego-facebook** | 1,19 | 1973 | 1973 |
| **graph_actors_dat** | 2075,61 | 3091 | 3092 |
| **graph_CA-AstroPh** | 734,38 | 6741 | 6740 |
| **graph_CA-CondMat** | 2282,08 | 9584 | 9584 |
| **graph_CA-HepPh** | 286,81 | 4718 | 4718 |
| **socfb-Brandeis99** | 3196,43 | 1400 | 1400 |
| **socfb-Mich67** | 3193,98 | 1331 | 1329 |
| **soc-gplus** | 0,33 | 8244 | 8244 |

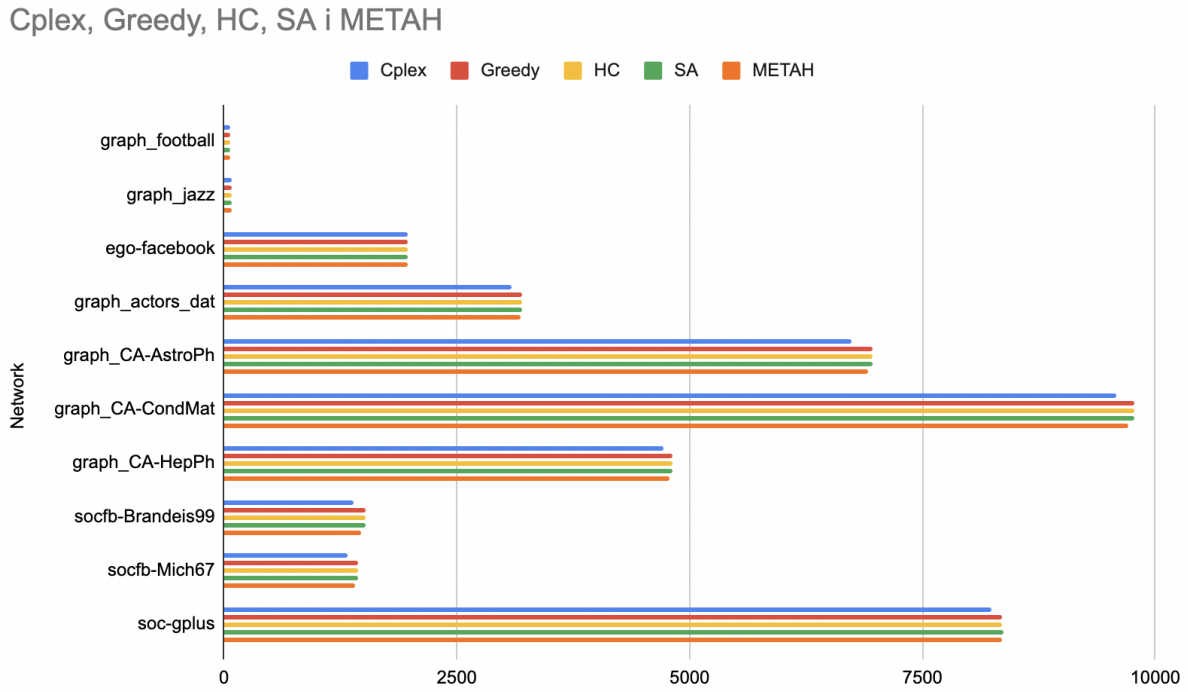
Sólo hace falta comparar los resultados de CPLEX con los obtenidos mediante otros métodos para ver que el número de nodos encontrados es siempre menor, lo que se explica gracias al hecho de que CPLEX no usa reglas ni heurísticas para dar con la solución, si no algoritmos matemáticos que prueban la optimalidad de las soluciones encontradas. Nótese que CPLEX va encontrando varios modelos que satisfacen las características del problema, y sólo las consideramos cuando aparece un nuevo modelo que contenga menos nodos (en el caso de estar minimizando) que la última solución. Ésta tabla es, entonces, la recopilación de los resultados más mínimos encontrados, o bien hasta haber terminado de trabajar el grafo, o bien hasta haber agotado el tiempo límite proporcionado.

Por lo que a la última columna respecta, presentamos los mejores resultados conocidos para estas instancias, usando CPLEX con un máximo de tiempo de 7200. Podemos ver que, en todos los casos excepto para la instancia de *graph_actors_dat*, los mejores resultados conocidos superan a aquellos obtenidos en nuestros experimentos, aunque es siempre con una diferencia de uno o dos nodos. A continuación, añadimos un gráfico donde podemos visualizar mejor los resultados obtenidos.



### 8.5 Comparación global

En el siguiente gráfico, mostramos todos los algoritmos simultáneamente, con la intención de ofrecer una visión más global del proceder de cada uno de los algoritmos que hemos usado para resolver éste problema de optimización, y de la optimalidad de sus resultados.



# 9 Conclusiones

Tras la realización de los experimentos con los diversos algoritmos, podemos extraer una serie de conclusiones y también confirmar ciertas hipótesis.

En primer lugar, hemos podido confirmar que para todos los casos, nuestros resultados obtenidos por nuestros algoritmos se asemejan bastante a los que se encuentran en la web de la asignatura. Esto nos reafirma el correcto funcionamiento de los algoritmos que hemos programado así como el buen uso de la herramienta CPLEX. Y por lo que al tiempo respecta, si bien es cierto que nuestros algoritmos son algo más lentos en la mayoría de casos, consiguen llegar a los mismos resultados en una diferencia de tiempo no muy pronunciada.

Por otra parte, nos queda comparar la calidad de cada algoritmo con el resto de soluciones implementadas. Lo haremos fijándonos tanto en el tanto el tiempo empleado como en el número de nodos obtenidos en la solución.

Lo primero que comprobamos es que nuestra solución greedy, encuentra las mismas soluciones que los algoritmos de Hill Climbing y Simulated Annealing, en un tiempo notablemente menor. Como ya hemos mencionado, esto se debe muy al hecho de que la greedy probablemente da con un máximo local, lo que limita mucho el margen de mejora que pueda obtenerse con los otros dos algoritmos, que parten de la greedy como solución inicial.

Podemos concluir pues, que la búsqueda local no es la más adecuada para abordar este problema, pues solamente nos devuelven resultados que pueden ser obtenidos con algoritmos menos informados, en un tiempo considerablemente menor. Esto ya lo sospechábamos, puesto que el modo en que se mueven estos dos algoritmos por el espacio de soluciones no es del todo compatible con este algoritmo. Tanto Hill Climbing como Simulated Annealing (aunque en menor medida, sobre todo al principio), son algoritmos que solamente consideran un cambio de estado si observan una mejora inmediata, o terminan. Mientras que, por otro lado, pueden existir numerosas instancias de nuestro problema en las que sea necesario pasar por empeoramientos en la calidad de la solución por tal de llegar a otras soluciones mejores. Son precisamente estas situaciones mejores las que quedan automáticamente fuera del alcance de los algoritmos de búsqueda local, especialmente Hill Climbing.

También observamos que los resultados de todos los enfoques comentados hasta ahora, nos encuentran mínimos con algunos nodos más que los que encontramos con la última herramienta que empleamos para resolver este problema de optimización. Podemos comprobar que herramientas de optimización como CPLEX son las más acertadas para la resolución de problemas de optimización, pues devuelven resultados óptimos o muy buenos, dentro de un marco de tiempo establecido.

Comparado con los resultados de la web, comprobamos que con CPLEX alcanzamos también todas las soluciones óptimas en las instancias en que se ha encontrado una. En cuanto al resto de grafos, obtenemos en la mayoría de los casos el mismo resultado; de hecho hay uno en que incluso lo mejoramos. Hay solamente dos instancias en las que el resultado obtenido no sea igual o mejor a los de la web, y en éstos, la diferencia es de tan sólo uno y tres nodos.

Por otro lado, si comparamos los resultados obtenidos con CPLEX y los que obtenemos con el algoritmo voraz, podemos ver que obtenemos menos nodos en todos los casos, y que la diferencia crece para los grafos más grandes. Sin embargo, hay un trade-off importante que es muy necesario

mencionar, pues el tiempo que tardamos en encontrar dichas soluciones con CPLEX es mucho mayor al que se consigue con la greedy, que nunca llega al segundo, ni siquiera para los grafos más grandes. Mientras que CPLEX puede en algunos casos tardar tiempos considerablemente mayores. Respecto a los tiempos entre CPLEX y los algoritmos de búsqueda local, parece ser que Hill Climbing acaba en menos tiempo para la mayoría de instancias, mientras que entre CPLEX y Simulated Annealing está bastante igualado; en la mitad de instancias es CPLEX la más lenta, y en la otra es Simulated Annealing.

En resumen, tenemos que por un lado todos nuestros resultados van muy acorde con los que presenta la página web de la asignatura. Por otra parte, nuestra solución greedy obtiene resultados tan buenos como los de búsqueda local a mucha más velocidad, mientras que CPLEX obtiene mejores resultados pese a necesitar, en esencia, el mismo tiempo que los algoritmos de búsqueda local. Habría que decidir el uso de uno o de otro en función de cómo valoremos la relación entre la calidad de la solución obtenida y el tiempo necesario para calcularla. De ser más importante la optimización, escogeremos CPLEX, mientras que si obtener resultados casi instantáneos nos compensa el hecho de tener una solución en cierto modo peor, nos decantamos por el algoritmo voraz. Finalmente, parece que los algoritmos de búsqueda local quedan fuera de lugar para abordar este problema de optimización, y nos ofrecen lo peor de los dos planteamientos anteriores.

# 10 Bibliografía


[1] Venkateswari , R. “Applications of Domination Graphs in Real Life.” *A journal of composition theory*, http://jicrjournal.com/.

[2] “Algoritmo Voraz.” *Wikipedia*, Wikimedia Foundation, 23 Feb. 2021, https://es.wikipedia.org/wiki/Algoritmo_voraz.

[3] “Counting Sort Algorithm.” *Programiz*, https://www.programiz.com/dsa/counting-sort.

[4] “Example of a minimum positive influence dominating set in a typical input network” *ResearchGate*, Mar 2018, https://www.researchgate.net/figure/Example-of-a-minimum-positive-influence-dominating-set-in-a-typical-input-network_fig1_318635766.

[5] “Proof That Dominant Set of a Graph Is NP-Complete.” *GeeksforGeeks*, 4 June 2020, https://www.geeksforgeeks.org/proof-that-dominant-set-of-a-graph-is-np-complete/.

[6] *Algoritmos Voraces - Webdiis.unizar.es*. https://webdiis.unizar.es/asignaturas/EDA/ea/slides/2-Algoritmos%20Voraces.pdf.

[7] *4.2 Búsqueda Local (Local Search)*, https://ccc.inaoep.mx/~emorales/Cursos/Busqueda/node58.html.

[8] “Metaheurística.” *Wikipedia*, Wikimedia Foundation, 13 Aug. 2020, https://es.wikipedia.org/wiki/Metaheur%C3%ADstica.

[9] *Domination Problems in Social Networks - USQ EPrints*. https://eprints.usq.edu.au/27867/1/Wang_whole_2014.pdf.

[10] Bouamama, Salim, and Christian Blum. “An Improved Greedy Heuristic for the Minimum Positive Influence Dominating Set Problem in Social Networks.” *MDPI*, Multidisciplinary Digital Publishing Institute, 28 Feb. 2021, https://www.mdpi.com/1999-4893/14/3/79/htm.

[11] *Tutorial On CPLEX Linear Programming*. https://www.cs.upc.edu/~erodri/webpage/cps/lab/lp/tutorial-cplex-slides/slides.pdf.

[12] *IBM ILOG CPLEX Optimization Studio*. https://www.ibm.com/downloads/cas/V1WDO6OR.

[13] “CPLEX.” *Wikipedia*, Wikimedia Foundation, 15 Nov. 2021, https://en.wikipedia.org/wiki/CPLEX.

[14] *Namespaces*, https://www.ibm.com/docs/en/icos/12.7.1.0?topic=cplex-net-reference-manual.

[15] “ILOG CPLEX 9.0.” *ILOG CPLEX 9.0 README*, http://www.mat.uc.pt/~jsoares/teaching/software/cplex/readme.html.