DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

Algoritmos y Estructura de Datos III
Primer Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Iván Arcuschin	678/13	iarcuschin@gmail.com
Martín Jedwabny	885/13	martiniedva@gmail.com
José Massigoge	954/12	jmmassigoge@gmail.com
Lucas Puterman	830/13	lucasputerman@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Ejercicio 1 - Demostración

1.1. Definiciones

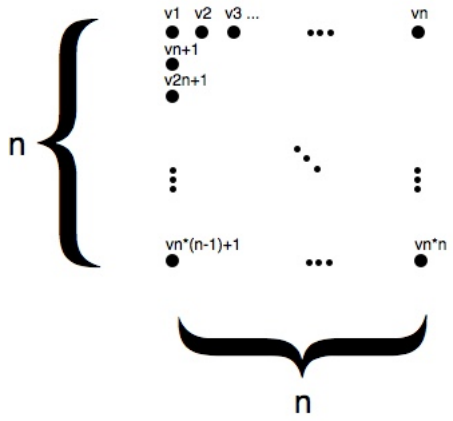
Sea G=(V,E) un grafo simple:

- Definición 1: D $\subseteq$ V es un **conjunto dominante** (CD) $\iff$ $\forall$ v $\in$ V, v $\in$ D ó $\exists$ w $\in$ D tal que (v,w) $\in$ E (tiene un vecino en D).
- Definición 2: D $\subseteq$ V es un **conjunto independiente** (CI) $\iff$ $\forall$ v,w $\in$ D, (v,w) $\notin$ E.
- Definición 3: D $\subseteq$ V es un **conjunto independiente dominante** (CID) $\iff$ D es dominante e independiente.
- Definición 4: D $\subseteq$ V es un **conjunto independiente dominante mínimo** (CIDM) $\iff$ D es el conjunto independiente dominante de V con menos nodos. Es decir que $\forall$ D' $\subseteq$ V tal que D' es independiente dominante, #(D) $\leq$ #(D').
- Definición 5: D $\subseteq$ V es un **conjunto independiente maximal** (CIMax) $\iff$ D es independiente y $\nexists$ D' $\subseteq$ V independiente tal que D $\subset$ D' ($\subset$ estricto).

1.2. Ejercicio A

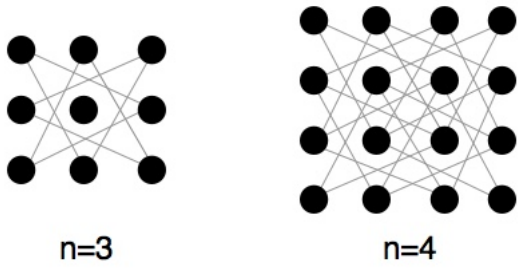
Relacionar el problema de CIDM con el problema 3 del TP 1 (i.e., similitudes y diferencias):

Si definimos a G=(V,E) grafo simple para que represente al tablero de ajedrez que se forma en el ejercicio 3 del tp1, los nodos serían una cuadricula de la pinta (sin considerar los ejes, eso lo hacemos después):



Donde 'n' es el parámetro del ejercicio que indica el tamaño de lado del tablero.

Ahora, como el ejercicio consiste en poner la mínima cantidad de caballos para que todas las casillas tengan un caballo o bien estén amenazadas, podemos representar las 'amenazas' como los ejes del grafo, por ejemplo:



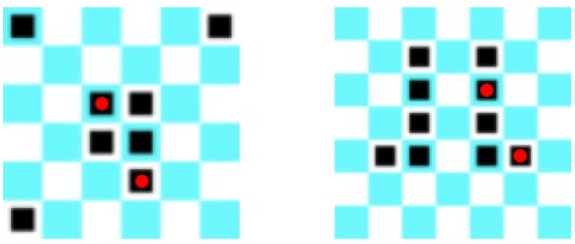
Observación: como cualquier posición x que amenaza a otra posicion 'y' sería amenazada si pongo un caballo en 'y', podemos representar el problema con un grafo simple y no un digrafo.

Entonces los ejercicios son similares ya que lo que estamos buscando en el ejercicio del TP1 es la mínima cantidad de caballos para que todas las posiciones (nodos) tengan un caballo o esten amenazadas (dominadas).

Es decir, si los caballos son un conjunto D de vértices en V, queremos hallar #(D) tal que:

- $\forall$ v $\in$ V, v $\in$ D o $\exists$ w $\in$ D tal que (v,w) $\in$ E $\iff$ D es dominante.
- $\forall$ D' $\subseteq$ V tal que D' es dominante, #(D) $\leq$ #(D') $\iff$ D es mínimo entre los conjuntos dominantes de V.

Por lo tanto el problema consiste en hallar un conjunto dominante mínimo pero, a diferencia del problema en este TP, puede o no ser independiente. Esto se debe a que existen respuestas para el problema del TP1 donde hay caballos en ciertas posiciones que se amenazan entre sí:



Observación: estos graficos fueron extraidos de la página: http://home.earthlink.net/ morgenstern/solution/knsols1.htm. El primero representa una solución óptima del ejercicio del TP1 con un tablero de 6x6 y la segunda imágen es para un tablero de 7x7. En ambas, se marcan con un punto negro los casilleros que tienen un caballo y, como vemos señalado en rojo, hay caballos en posiciones que se amenazan entre sí.

Mas alla de eso, ambos problemas son de optimización combinatoria. Los conjuntos factibles son los vertices de un grafo que cumplen ciertas condiciones (si bien como dijimos antes, no son las mismas para los dos problemas) y la función de optimalidad consiste en minimizar la cantidad de vértices del conjunto.

1.3. Ejercicio B

Demostrar que todo conjunto independiente maximal es un conjunto dominante:

Sea G=(V,E) un grafo simple y D $\subseteq$ V un conjunto independiente maximal, quiero ver que D es un conjunto dominante.
Supongamos (por absurdo) que D no es un conjunto dominante:

$\Rightarrow \exists$ v $\in$ V tal que v $\notin$ D $\wedge \forall$ w $\in$ D, (v,w) $\notin$ E (no es vecino de ningún nodo en D).

$\Rightarrow$ D$\cup${v} es independiente ya que como D es independiente sucede que $\forall$ w $\in$ D$\cup${v} $\nexists$ x $\in$ D tal que (x,w) $\in$ E.

$\Rightarrow$ Absurdo! pues D era maximal y $\exists$ D' = D$\cup${v} tal que D' es independiente y D $\subset$ D', asi contradiciendo el hecho de que D es un conjunto independiente maximal.

Este absurdo vino de suponer que D no era dominante.
Por lo tanto, si D es un conjunto independiente maximal $\Rightarrow$ D es dominante.

1.4. Ejercicio C

Describir situaciones de la vida real que puedan modelarse utilizando CIDM:

- **El turista:** tenemos un conjunto de ciudades que forman un grafo simple conexo donde los nodos son las ciudades y dos nodos estan conectados si y solo si las ciudades son vecinas. Suponiendo que queremos conocer tantas culturas distintas como sea posible, decidimos que no queremos visitar ningun par de ciudades vecinas ya que sus culturas son muy similares. Como el problema que proponemos consiste en hallar un conjunto mínimo de ciudades (nodos en el mapa) tal que toda ciudad sea visitada o bien una ciudad vecina sea visitada pero no ambas (el conjunto de nodos sea independiente y dominante), podemos decir que estamos buscando un CIDM.
- **Spamear una red social:** supongamos que tenemos un virus que se ocupa de spamear Facebook de manera que un mensaje sea propagado por toda la red. Asumimos que los usuarios de Facebook son nodos en un grafo simple conexo y que dos nodos estan conectados si y solo si esos dos usuarios son amigos en la red. Ahora, el virus no quiere ser descubierto, asi que lo que debe hacer es infectar la menor cantidad de cuentas que no sean amigas, ya que si spamea demasiadas cuentas o un par de cuentas que son amigas, sería mucho más fácil de descubrirlo. Luego, estamos buscando el mínimo conjunto de nodos en la red de usuarios tal que todos los usuarios vean el mensaje (dominación) y dos amigos no sean infectados a la vez (independencia), el cual es un CIDM.
- **Cámaras de seguridad en un barrio:** tenemos casas en un barrio que representan los nodos de un grafo simple conexo, donde dos nodos estan conectados si y solo si sus casas respectivas con vecinas. Queremos poner cámaras de seguridad en el barrio de manera que toda casa tenga una cámara de seguridad o la tenga una casa vecina, ya que el rango de cobertura de la cámara es amplio y puede filmar la casa donde está y también las vecinas (dominación). No queremos poner cámaras en dos casas vecinas (independencia) para que no sea tan notorio y arruine el paisaje. A la vez, tenemos un presupuesto limitado, por lo cual queremos poner poner tan pocas cámaras como sea posible (minimalidad). Por lo tanto, el conjunto de casas que estamos buscando para ponerles cámaras es un CIDM.

2. Ejercicio 2 - Algoritmo exacto

Diseñar e implementar un algoritmo exacto para CIDM.

2.1. Ejercicio A

Explicar detalladamente el algoritmo implementado. Elaborar podas y estrategias que permitan mejorar los tiempos de resolucion.

2.1.1. Estrategia

El algoritmo implementado se basa en la Técnica Algorítmica de *Backtracking*.

En primer lugar notese que un conjunto dominante trivial es el conjunto de todos los nodos del grafo. Si bien este conjunto en principio no es independiente (a menos que $X(G) = \emptyset$), es el conjunto dominante más grande posible (no hay ningún nodo que quede afuera, es decir: $V(G) \setminus D = \emptyset$).

Luego, la idea principal del algorítmo es empezar con este primer conjunto dominante e ir sacando nodos recursivamente mientras chequeamos dominancia e independencia. Cada vez que encontramos un conjunto dominante e independiente, nos fijamos su cardinal. Si el cardinal de este nuevo conjunto es menor que el mínimo hasta ese momento nos quedamos con el nuevo y lo guardamos como mínimo. En caso de que el cardinal sea mayor, seguimos quitando nodos para encontrar un conjunto más chico.

2.1.2. Podas

Antes de mostrar cuales son las podas que utilizamos en el algoritmo, veamos algunos Lemas.

Lema 2.1. *Sea C un conjunto dominante e independiente de un grafo G. Entonces, cualquier subconjunto de C (distinto de C) es no-dominante respecto a G.*

Demostración. Veamos que vale por absurdo. Supongamos H un subconjunto de C tal que $H \subset C$ y H es dominante de G. Como H está estrictamente incluido en C, entonces $\exists\ v \in C$ tal que $v \notin H$.

Ahora, como C es independiente, no hay nodos de C que sean adyacentes en G, en particular: $\forall\ w \in C,\ (v, w) \notin X(G)$.

Entonces, $v \in V(G)$ pero $v \notin H$ y $\forall\ w \in H \subset C,\ (v, w) \notin X(G)$. Luego, H no es dominante, ya que el nodo v no está en H ni tiene algún vecino que esté en H. Absurdo. □

Lema 2.2. *Sea C un conjunto no-dominante respecto a G. Entonces, cualquier subconjunto de C es no-dominate respecto a G.*

Demostración. Trivial. Sea H un subconjunto de C ($H \subseteq C$).

Como C es no-dominante, $\exists\ v \in V(G)$ tal que $v \notin C$ y $\forall\ w \in C,\ (v, w) \notin X(G)$.

Pero como $H \subseteq C$, $v \notin H$ y $\forall\ w \in H,\ (v, w) \notin X(G)$.

Luego H es no-dominante respecto a G. □

Usando los resultados de estos dos lemas podemos optimizar el algortimo cortando ramas en el árbol de recursión.

1. Por el primer Lema, cada vez que encontramos un conjunto dominante e independiente podemos ahí mismo devolver el que tenga menor cardinal entre ese conjunto y el mínimo encontrado hasta ese momento. Esto es así ya que sabemos que cualquier subconjunto del nuevo conjunto es no-dominante.

2. Por el segundo Lema, al momento de sacar un nodo de un conjunto podemos chequear si el subconjunto es dominante o no. En caso de que no lo sea, ni siquiera lo procesamos, ya que no es dominante y ninguno de sus subconjuntos lo será.

2.1.3. Pseudocódigo

```
funcion resolver:
    Creamos un vector de n elementos, llenandolo con los nodos desde 0 a n-1.
    llamamos a resolver_aux pasandole como parametro la matriz de adyacencia, el vector
        recién creado y la cantidad de nodos en el grafo.

funcion resolver_aux:
    Llamemos dom al conjunto dominante pasado por parametro.
    Llamemos cidm al conjunto dominante e independiente con menor cardinal encontrado
        hasta ahora.
    Si el cidm tiene tamaño 1 hacer
        Devuelvo cidm
    Sino hacer
        // Chequeamos si dom es independiente:
        Para i desde 0 hasta |dom| hacer:
            Para j desde i+1 hasta |dom| hacer:
                Si matriz_adyacencia[dom[i]][dom[j]] == TRUE hacer
                    dom NO es independiente
        Si es independiente hacer
            Devolvemos el conjunto con menor cardinal entre cidm y dom.
    Para i desde 0 hasta |dom| hacer
        Crear un vector nuevo llamado copia con los mismos nodos que dom.
        Borrar el nodo en la posicion i del vector copia.
        // Chequeamos si el conjunto copia es dominante:
        Para i desde 0 hasta n hacer:
            Si i no pertenece a copia hacer
                Bool nodo_valido = FALSE
                Para j desde 0 hasta |copia| hacer:
                    Si copia[j] == i hacer
                        nodo_valido = TRUE
                    Si matriz_adyacencia[i][copia[j]] == TRUE hacer
                        nodo_valido = TRUE
                Si nodo_valido == FALSE hacer
                    copia NO es dominante
        Si copia es dominante hacer
            Hacer un llamado recursivo a resolver_aux, pasando como parametro la matriz
                de adyacencia, cidm, copia y la cantidad de nodos en el grafo.
            Llamemos nuevo_cidm al conjunto que devuelve el llamado recursivo.
            Devolvemos el conjunto con menor cardinal entre cidm y nuevo_cidm.
```

2.2. Ejercicio B

Calcular el orden de complejidad temporal de peor caso del algoritmo.

Veamos las distintas partes del algoritmo.

En la función *resolver* tenemos un ciclo de $\Theta(n)$ más un llamado al constructor de la estructura vector para construir el primer conjunto dominante que se realiza solo una vez. Este constructor tiene costo $\Theta(n)$[1], y dentro del ciclo solo tenemos operaciones constantes (asignación), por lo que el costo, sin contar el llamado a la función *resolver_aux*, es de $\Theta(n)$. En la función *resolver_aux* tenemos 5 ciclos distintos, 2 anidados entre sí y otros 3 anidados entre sí.

Los primeros 2 corresponden al chequeo de indepedencia. Dentro de ellos solo hay operaciones de costo constante (una asignacion y dos comparaciones) y se repiten cada uno $|dom|$ veces, donde dom es

[1] Referencia: `http://www.cplusplus.com/reference/vector/vector/vector/`

el conjunto dominante actual. Luego, el chequeo de independencia toma: $\mathcal{O}(|dom| * |dom|)$, que en el peor caso es $\mathcal{O}(n^2)$.

De los otros 3 ciclos anidados, el primero corresponde a la iteración sobre el conjunto dom mientras vamos probando quitar distintos nodos, y los otros 2 corresponden al chequeo de dominancia de los nuevos conjuntos generados. El primer ciclo se ejecuta $|dom|$ veces, mientras que el segundo y el tercero n y $|copia|$ veces respectivamente, donde $copia$ es el nuevo conjunto generado (como se muestra en el pseudocódigo). Además dentro del primer ciclo utilizamos la función *erase* de vectores que permite borrar un elemento dado su índice y el constructor de la estructura vector por copia. Ambas operaciones tiene costo en el peor caso lineal en la cantidad de elementos del vector original [2]. El resto de las operaciones en los 3 ciclos es de costo constante (comparaciones y asignaciones).

Luego, la complejidad de los 3 ciclos nos queda: $\mathcal{O}(|dom| * (|dom| + |copia| + |n| * (|copia|)))$, que asintoticamente equivale a $\mathcal{O}(n^3)$

Fuera de los ciclos en la función *resolver_aux* solo tenemos operaciones constantes (comparaciones y asignaciones, más la función *size* de vectores, que tiene costo constante[3]). Entonces, el costo de ejecutar una vez la función *resolver_aux*, sin tener en cuenta el llamado recursivo a si misma, es de $\mathcal{O}(n^3)$.

Ahora, como vimos en el pseudocódigo, la función *resolver_aux* parte del conjunto dom y hace en el peor caso $|dom|$ llamados recursivos a si misma, cada uno quitando un nodo posible de $|dom|$.

Luego, el peor caso posible es que revisemos todas las posibles combinaciones de nodos. Es decir, pasar por *resolver_aux* la cantidad de veces del cardinal del Conjunto de Partes de V(G). O sea, en el peor caso *resolver_aux* toma: $\mathcal{O}(2^n * n^3)$.

Entonces, la complejidad final del algoritmo es:

$$\mathcal{O}(2^n * n^3)$$

2.3. Ejercicio C

Realizar una experimentación que permita observar los tiempos de ejecución del algoritmo en función de los parámetros de la entrada y de las podas y/o estrategias implementadas.

[2]Referencias: `http://www.cplusplus.com/reference/vector/vector/erase/` y `http://www.cplusplus.com/reference/vector/vector/vector/`

[3]Referencia: `http://www.cplusplus.com/reference/vector/vector/size/`

3. Ejercicio 3 - Heurística constructiva golosa

3.1. Ejercicio A

Realizamos una heurística golosa para resolver el problema. Lo que hace el algoritmo es lo siguiente. Primero toma los datos de entrada del problema, arma la matriz de adyacencia del grafo y además crea un vector de nodos de tamaño n donde cada nodo contiene guardado su número de nodo y el grado que tiene en el grafo.
Una vez que se procesaron los datos de entrada, se procede a resolver el problema. Para esto, se ordena el arreglo de nodos según sus grados de mayor a menor. Luego, se crea un arreglo de booleanos de tamaño n, donde cada uno está inicializado en $false$. En este arreglo se guarda si el nodo ya fue visitado o no. También se crea un arreglo llamado $cidm$ donde se guardará la solución.
Por último, se recorre en orden el arreglo de nodos ordenados según su grado. Si un nodo no fue visitado, se agrega el nodo a $cidm$ y luego se recorre su fila en la matriz de adyacencia, marcando como visitados a todos sus adyacentes (ya que queremos que sea dominante y mínimo). Una vez que se recorre todo el arreglo, ya tenemos el $cidm$, solo resta mostrarlo por pantalla.

3.2. Ejercicio B

Calculemos la complejidad del algoritmo. Para esto dividamos el algoritmo en tres etapas, la entrada de datos, la resolución del problema y la salida de datos.

En la entrada de datos, se crea la matriz de adyacencia, esto tiene costo temporal $\mathcal{O}(m)$, a la vez, se crea el vector de Nodos, con costo $\mathcal{O}(n)$. Por lo tanto, la entrada de datos tiene costo $\mathcal{O}(n+m)$

La resolución del problema, comienza ordenando el arreglo de nodos, esto tiene costo $\mathcal{O}(n*log(n)$. Luego, crea los vectores con costo constante. Por último, recorre todos los nodos (n), y por cada nodo, si no fue visitado, lo agrega a la solución con costo constante y se fija en su fila en la matriz de adyacencia quienes son sus vecinos y los marca como visitados, pero notemos que la matriz de adyacencia tiene marcados $2m$ posiciones, por lo tanto, se realizarán a lo sumo $2m$ operaciones en todo el ciclo, que todo el proceso tiene costo $\mathcal{O}(n+2m) \in \mathcal{O}(n+m)$.

Por último, se muestra el $cidm$ aproximado por pantalla, para eso se recorre el vector $cidm$ calculado en la resolución, que tiene a lo sumo tamaño n, por lo que tiene costo $\mathcal{O}(n)$.

Por lo tanto, juntando las tres etapas nos queda un costo total de $\mathcal{O}(n*log(n)+n+m+n) \in \mathcal{O}(n*log(n)+m$.

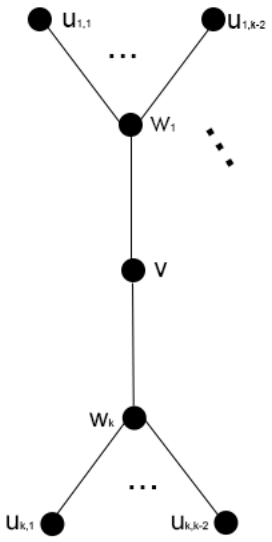
Entonces, la complejidad del algoritmo es de:

$$\mathcal{O}(n*log(n)+m)$$

3.3. Ejercicio C

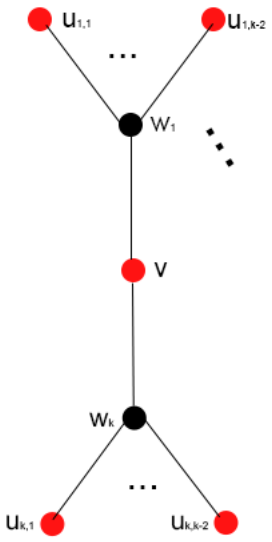
Veamos un ejemplo de instancias donde la heurística golosa no funciona. Supongamos que tenemos un grafo G con un nodo central v y sean w_i con $1 \leq i \leq k$ sus k vecinos, y supongamos que todo w_1 tiene a su vez $k-2$ vecinos de grado 1, llamemoslos $u_{i,j}$ con $1 \leq i \leq k$ y $1 \leq j \leq k-2$.

Ahora, al ordenar los nodos según su grado nos quedaría v con k vecinos, luego w_i con $1 \leq i \leq k$, donde cada w_i tiene grado $k-1$, y por último los $k*(k-2)$ nodos u de grado 1. Al correr el algoritmo goloso, este arrojaría un DCIM con cardinalidad $k*(k-2)+1$ ya que el algoritmo seleccionará a v y a $u_{i,j}$ con $1 \leq i \leq k$ y $1 \leq j \leq k-2$.

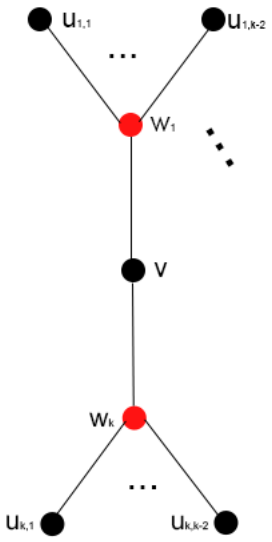


Sin embargo, si seleccionamos a todos los w_i con $1 \leq i \leq k$, tendríamos un DCIM real con cardinalidad k que es mucho menor que el arrojado por el algoritmo.

Veamos que en este grafo $G = (V, E)$ tenemos $|V| = 1 + k * (k - 1)$ nodos y nuestro algoritmo goloso arroja una solución de tamaño $1 + k * (k - 2)$ es decir, solo quedan sin seleccionar k nodos. Sin embargo, la solución optima sólo utiliza k nodos. Notemos que el error es cuadrático.



4. Ejercicio 4 - Heurística de búsqueda local



5. Ejercicio 5 - Metaheurística GRASP

Diseñar e implementar un algoritmo para CIDM que use la metaheurística GRASP.

5.1. Ejercicio A

Explicar detalladamente el algoritmo implementado. Plantear distintos criterios de parada y de selección de la lista de candidatos (RCL) de la heurística golosa aleatorizada.

5.1.1. Idea general

Como pide el enunciado, la idea general del algoritmo es usar la metaheurística GRASP, para lo cual es necesario tener implementaciones de: heurística constructiva golosa y heurística de búsqueda local, que fueron convenientemente implementadas en los puntos anteriores.

La estructura general de un algoritmo GRASP es:

```
1. Poner en mejor_solucion una primera solucion Random.
2. Mientras no se cumpla el criterio de parada hacer:
3.     Poner en nueva_solucion una solucion usando la funcion ConstruirGreedyRandom()
4.     Intentar mejorar la nueva_solucion usando la funcion BusquedaLocal()
5.     Si costo(nueva_solucion) < costo(mejor_solucion) hacer:
6.         Poner en mejor_solucion la nueva_solucion
```

En nuestro algoritmo se implementó de la siguiente forma:

1. Se utilizo para la primera solución Random el mismo método ConstruirGreedyRandom que se utiliza al generar una solución golosa randomizada.
2. Los criterios de parada considerados se detallan más adelante.
3. La función ConstruirGreedyRandom es una variación del algoritmo goloso implementado para el Ejercicio 3 (agregando lista de candidatos), detallado más adelante.
4. La función BusquedaLocal es identica al algoritmo implementado para el Ejercicio 4. En la experimentación se probo con los distintos criterios de vecindad expuestos en ese mismo Ejercicio.
5. Definimos el costo de una solución como la cantidad de nodos de dicha solución, por lo que decimos que una es mejor que otra si la primera tiene menor cantidad de nodos.
6. Si encontramos una solución con menor cantidad de nodos que la mejor hasta ese momento, la guardamos como mejor solución.

5.1.2. Criterios de parada

Los criterios de parada que se utilizaron para la implementación se pensaron en función de la cantidad de nodos del grafo original:

1. Criterio: realizar n iteraciones, con n la cantidad de nodos del grafo.
2. Criterio: realizar $f(n)$ iteraciones, con n la cantidad de nodos del grafo y siendo $f(n) = n^2$.

5.1.3. Selección de lista de candidatos (RCL)

Al algoritmo con heurística constructiva golosa del Ejercicio 3 se lo modificó de la siguiente forma:

- En vez de iterar en los elementos del array de nodos ordenados por grado, iteramos hasta que hayamos visitado n nodos usando un contador, ya que no necesariamente vamos a visitar secuencialmente todos los nodos desde el índice 0 hasta el (n-1)-ésimo.
- Dentro del ciclo principial, lo primero que hacemos es elegir el indice de un nodo para agregar a la solución.

  A diferencia del algoritmo goloso original, que elegiamos siempre el nodo con grado más alto no visitado hasta ese momento, ahora vamos a tener una lista de candidatos (nodos) a agregar a la solución, y de todos ellos vamos a elegir alguno de manera aleatoria.

  La lista de candidatos se construye tomando como referencia el nodo con grado más alto no visitado hasta ese momento. Si d_{max} es dicho grado, agregaremos a la lista de candidatos todos los nodos que tengan grado a lo sumo un $\alpha\,\%$ menos que d_{max}, es decir $d \geq d_{max} - dmax * \alpha$. Donde α es un valor entre 0 y 1.

  Esto nos asegura que, si bien el nodo a agregar a la solución es aleatorio, se encuentra dentro de cierto grupo de nodos mejores que otros.

  Notese además que si α es igual a cero, la solución golosa es la misma que daría el algoritmo del Ejercicio 3, es decir que no hay componente aleatorio. Y si α es igual a 1, la solución es completamente aleatoria.
- Luego de que se eligió un nodo, se lo agrega a la solución, y luego se lo borra de los nodos posibles para futuras iteraciones (se lo marca como *visitado*). Además, se aumenta en uno la cantidad de nodos visitados.
- Por último, se itera sobre todos los nodos adyacentes al elegido, borrandolos de los nodos posibles y aumentando en uno el contador de nodos visitados.

5.1.4. Pseudocódigo

El esquema general del algoritmo GRASP ya se mostró en la sección Idea General, y el algoritmo de Busqueda Local es identico al utilizado en el Ejercicio 4, por lo que mostraremos aquí solo el pseudocódigo de la función ConstruirGreedyRandom:

```
Poner nodos = un vector de structs Nodo, que tiene el indice del nodo y su grado
    en el grafo, de tamaño n.
Ordenar dicho conjunto de mayor a menor grado.
Poner solucion = un vector de enteros inicializados en 0. El valor en cada indice
    representa si el nodo con dicho indice pertenece o no a la solución.
Poner nodos_visitados = 0
Mientras nodos_visitados < n hacer:
    Poner mejor_grado = nodos[0].grado
    Poner limite_indice = 0
    Para i desde 0 hasta |nodos| hacer:
        Si nodos[i].grado >= mejor_grado - mejor_grado * alpha hacer:
            limite_indice = i
        Sino
            Salir ciclo Para
        Fin Si
    Fin Para

    Poner indice_nuevo = random_in_range(0, min(limite_indice, |nodos|-1))
    Poner nodo_nuevo = nodos[indice_nuevo].indice
    Agrego nodo_nuevo al vector solucion y lo borro del vector nodos
    Incrementar nodos_visitados en uno

    Para v en Adyacentes(nodo_nuevo) hacer:
        Si v esta en nodos hacer:
            Borrar v del vector nodos
            Incrementar nodos_visitados en uno
        Fin Si
    Fin Para
Fin Mientras
Devolver solucion
```

5.2. Ejercicio B

Realizar una experimentación que permita observar los tiempos de ejecución y la calidad de las soluciones obtenidas.

6. Ejercicio 6 - Experimentación final