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

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 'D' 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 propio 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, y se arma la matriz o las listas de adyacencia del grafo (según la implementación elegida). Además, se 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 marcan 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. Pseudocódigo

3.2.1. Implementación sobre listas de adyacencia

```
Crear listas de adyacencia con el input
Crear arreglo nodos que guarda numero de nodo y grado para cada nodo

Ordenar arreglo nodos segun su grado
Crear vector de booleanos visitado de tamano n para guardar nodos visitados
Crear vector cidm para guardar la solucion

Para cada nodo u en el arreglo ordenado:
        Si el nodo no fue visitado:
                agregar el nodo a cidm
                Para cada nodo w en su lista de adyacencia:
                        marcar w como visitado
                fin Para
        fin Si
fin Para

mostrar cidm
```

3.2.2. Implementación sobre matriz de adyacencia

```
Crear matriz de adyacencia con el input
Crear arreglo nodos que guarda numero de nodo y grado para cada nodo

Ordenar arreglo nodos segun su grado
Crear vector de booleanos visitado de tamano n para guardar nodos visitados
Crear vector cidm para guardar la solucion

Para cada nodo u en el arreglo ordenado:
        Si el nodo no fue visitado:
                agregar el nodo a cidm
                Para cada nodo w en su fila de la matriz de adyacencia:
                        Si el nodo w es adyacente:
                                marcar w como visitado
                        fin Si
                fin Para
        fin Si
fin Para

mostrar cidm
```

3.3. Ejercicio B

Calculemos la complejidad del algoritmo. Para esto vamos a calcular la complejidad para una implementación sobre matriz de adyacencia y una sobre listas de adyacencia. A su vez dividiremos el algoritmo en tres etapas, la entrada de datos, la resolución del problema y la salida de datos.

- Matriz de adyacencia

  n la entrada de datos, se crea la matriz de adyacencia, esto tiene costo temporal $\mathcal{O}(n^2)$, a la vez, se crea el vector de Nodos, con costo $\mathcal{O}(n)$. Por lo tanto, la entrada de datos tiene costo $\mathcal{O}(n + n^2) \in \mathcal{O}(n^2)$.

  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. Esto tiene un costo de $\mathcal{O}(n^2)$. Por lo tanto todo el proceso tiene costo $\mathcal{O}(n * log(n) + n^2) \in \mathcal{O}(n^2)$.

  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 $\mathcal{O}(n^2 + n^2 + n) \in \mathcal{O}(n^2)$.

- Listas de adyacencia

  Primero se deben crear las listas de adyacencia, como tengo m aristas y por cada una agrego en tiempo constante un elemento a 2 listas, esto tiene costo $\mathcal{O}(2m) \in \mathcal{O}(m)$.

  La resolución del problema, comienza ordenando el arreglo de nodos, esto tiene costo $\mathcal{O}(n * log(n)$. Luego, tenemos n listas pero la suma del tamaño de todas es de a lo sumo $2m$, por lo tanto, se realizarán a lo sumo $2m$ operaciones en todo el ciclo, por lo 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 $\mathcal{O}(n)$.

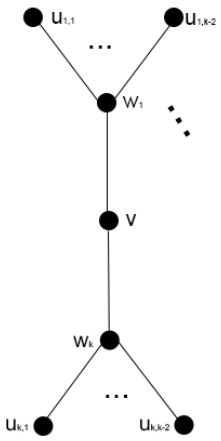
Por lo tanto, juntando las tres etapas nos queda un costo total de $\mathcal{O}(n * log(n) + n + m + n) \in \mathcal{O}(n * log(n) + m$

Entonces, la complejidad del algoritmo sobre listas de adyacencia es menor y es de:

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

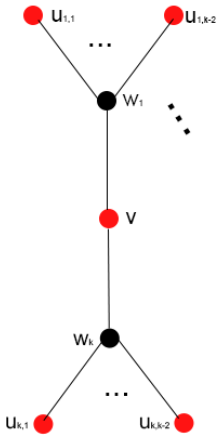
3.4. Ejercicio C

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 $1 \leq j \leq k-2$.



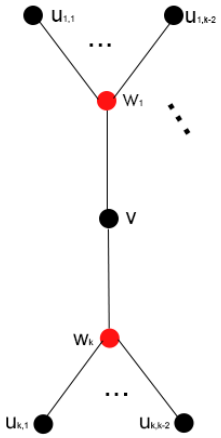
(a)

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 $1 \leq j \leq k-2$.



(a)

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.



(a)

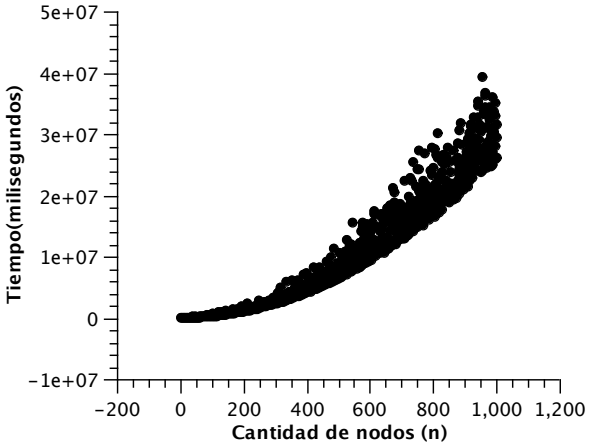
Veamos que en este grafo $G = (V, E)$ tenemos $|V| = 1 + k * (k - 1)$ nodos y nuestro algoritmo goloso arroja una solución de tamaño $1 + k * (k - 2)$ es decir, solo quedan sin seleccionar k nodos. Sin embargo, la solución optima sólo utiliza k nodos. Notemos que el error es cuadrático.

3.5. Ejercicio D

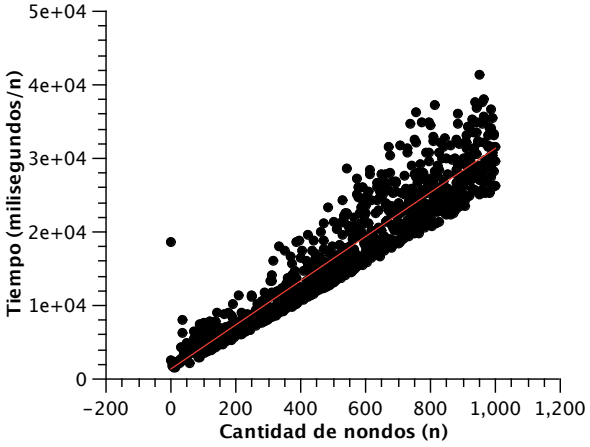
Para la experimentación decidimos comparar distintas familias de instancias tanto en la implementación sobre matriz como el a de listas de adyacencia y comparar la eficiencia de ambas.

3.5.1. Experimentación sobre grafo completo

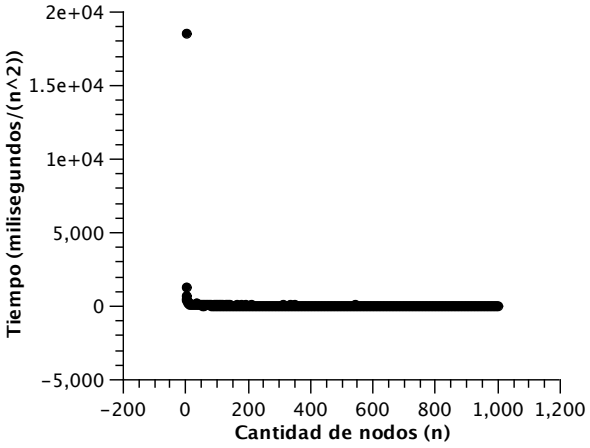
Se crearon instancias de grafos completos con $1 \leq n \leq 1000$.
En la implementación sobre matrices arrojo los siguientes resultados:



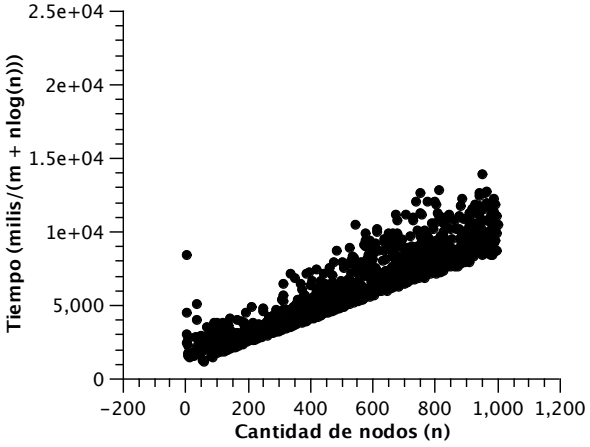
(a) Tiempos sin procesar, en milisegundos



(b) Dividiendo a los tiempos por n



(a) Dividiendo a los tiempos por n^2



(b) Dividiendo a los tiempos por $n + n * log(n)$

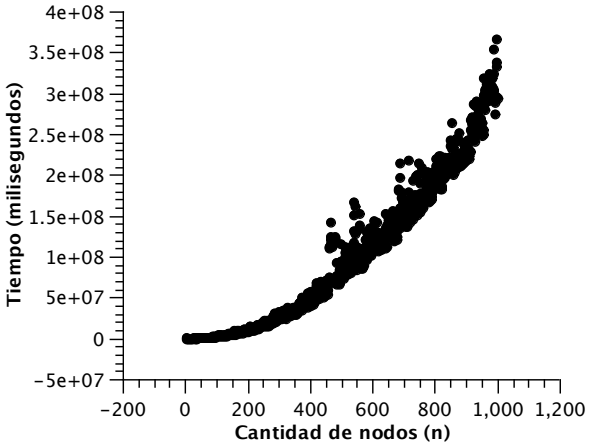
A continuación, adjuntamos una tabla con los últimos 20 valores obtenidos en las instancias, teniendo

en cuenta que los casos fueron previamente ordenados según el tamaño (n):

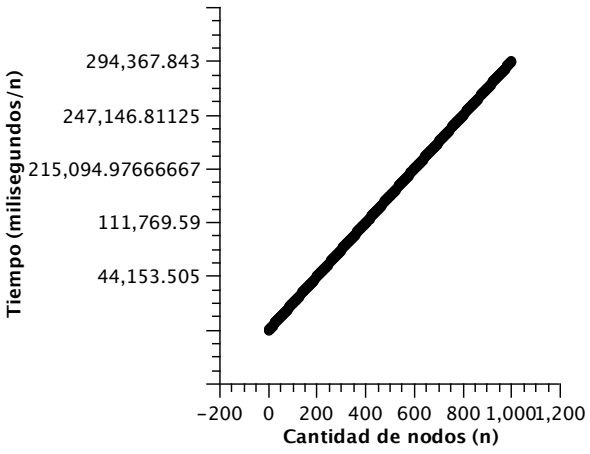
n	Tiempo(milis)	m	Tiempo(mili/(n))	Tiempo(mili/(n^2)))	Tiempo(mili/($n * log(n) + m$)))
980	24,845,670	479710	25,352.72448979592	25.870127030404	8,475.696536553674
981	33,412,818	480690	34,059.95718654434	34.71963015957629	11,384.934998666
982	24,857,309	481671	25,312.94195519348	25.77692663461658	8,459.892640536484
983	29,750,650	482653	30,265.15768056969	30.78856325592033	10,113.4891991821
984	25,165,002	483636	25,574.18902439025	25.99002949633155	8,544.681225102551
985	28,427,751	484620	28,860.66091370558	29.3001633641681	9,641.314760497822
986	27,111,085	485605	27,496.02941176471	27.88643956568429	9,184.088121506702
987	36,119,756	486591	36,595.49746707194	37.07750503249436	12,221.65041349897
988	33,731,556	487578	34,141.25101214575	34.5559220770706	11,400.34121094489
989	30,542,586	488566	30,882.29120323559	31.22577472521294	10,310.60678389385
990	31,176,002	489555	31,490.91111111111	31.80900112233446	10,512.26503410799
991	30,168,178	490545	30,442.15741675076	30.71862504212993	10,160.68392416765
992	25,911,896	491536	26,120.86290322581	26.33151502341311	8,717.090324045907
993	35,168,815	492528	35,416.73212487412	35.6663969031965	11,817.59489008868
994	28,113,874	493521	28,283.5754527163	28.45430126027797	9,436.079245553556
995	33,158,039	494515	33,324.66231155779	33.49212292618873	11,116.28719039287
996	26,079,861	495510	26,184.59939759036	26.28975843131563	8,733.26701997587
997	33,013,519	496506	33,112.85757271815	33.21249505789183	11,042.42206178822
998	29,605,093	497503	29,664.42184368737	29.72386958285308	9,891.007222031083
999	26,197,019	498501	26,223.24224224224	26.24949173397622	8,742.346964977023
1,000	31,613,146	499500	31,613.146	31.613146	10,537.71533333333

Como podemos ver la experimentación se condice con el cálculo teórico de la complejidad y arroja que es de $\mathcal{O}(n^2)$.

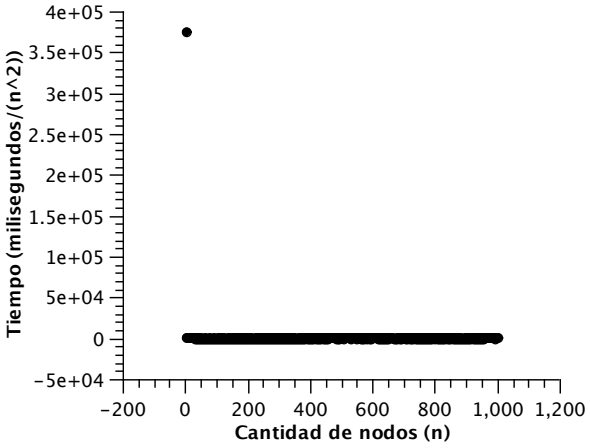
Veamos ahora los resultados en la implementación sobre listas de adyacencia:



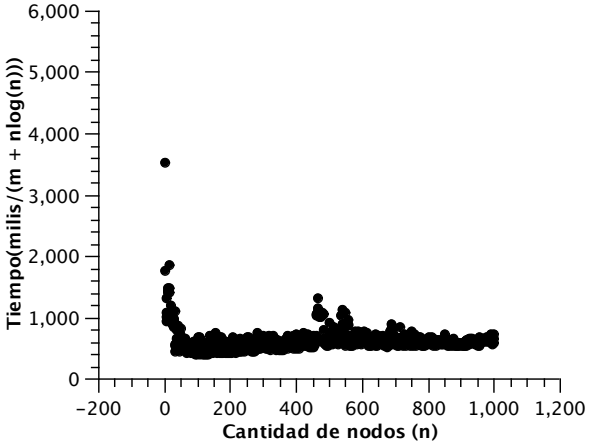
(a) Tiempos sin procesar, en milisegundos



(b) Dividiendo a los tiempos por n



(a) Dividiendo a los tiempos por n^2



(b) Dividiendo a los tiempos por $n + n * log(n)$

A continuación, adjuntamos una tabla con los últimos 20 valores obtenidos en las instancias, teniendo

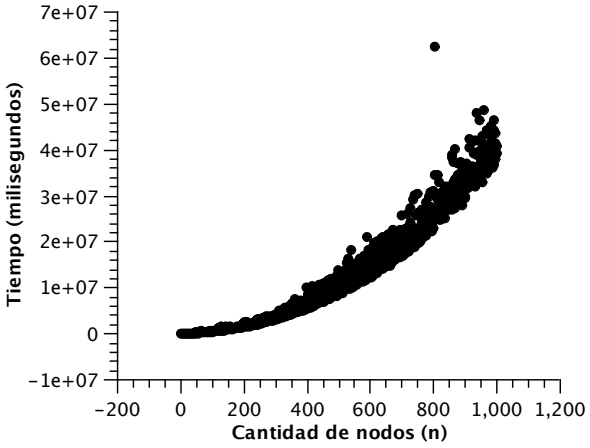
en cuenta que los casos fueron previamente ordenados según el tamaño (n):

n	Tiempo(milis)	Tiempo(mili/(n))	Tiempo(mili/(n^2)))	Tiempo(mili/($n * log(n) + m$)))	m
980	296,879,716	302,938.48571429	309.1209037900875	615.1144826034448	479,710
981	323,033,524	329,290.03465851	335.667721364436	667.9423920520295	480,690
982	292,431,614	297,791.86761711	303.2503743555071	603.4379490268261	481,671
983	297,273,430	302,414.47609359	307.6444314278647	612.1842807953875	482,653
984	318,286,205	323,461.59044715	328.7211285032058	654.1277503491588	483,636
985	308,937,975	313,642.6142132	318.4188976783736	633.6298448756557	484,620
986	323,384,902	327,976.57403651	332.6334422276989	661.9185207263685	485,605
987	304,973,689	308,990.56636272	313.0603509247369	622.9719891480256	486,591
988	302,281,783	305,953.22165992	309.6692526922257	616.2264905731232	487,578
989	353,599,216	357,532.06875632	361.5086640609904	719.3873732062154	488,566
990	292,805,769	295,763.4030303	298.7509121518212	594.5045184459034	489,555
991	289,577,857	292,207.72653885	294.8614798575678	586.7671292958553	490,545
992	289,158,638	291,490.5625	293.841292842742	584.7394227940042	491,536
993	293,229,428	295,296.50352467	297.3781505787238	591.7801781541916	492,528
994	275,111,868	276,772.50301811	278.443161990049	554.102004459966	493,521
995	333,606,954	335,283.37085427	336.9682119138405	670.5696612605566	494,515
996	338,319,005	339,677.71586345	341.041883397042	678.679115312912	495,510
997	332,699,588	333,700.69007021	334.7048044836616	666.0709764219323	496,506
998	366,440,239	367,174.58817635	367.9104089943414	732.1539875453961	497,503
999	296,662,430	296,959.38938939	297.2566460354248	591.5530805302149	498,501
1,000	294,367,843	294,367.843	294.367843	585.8066527363184	499,500

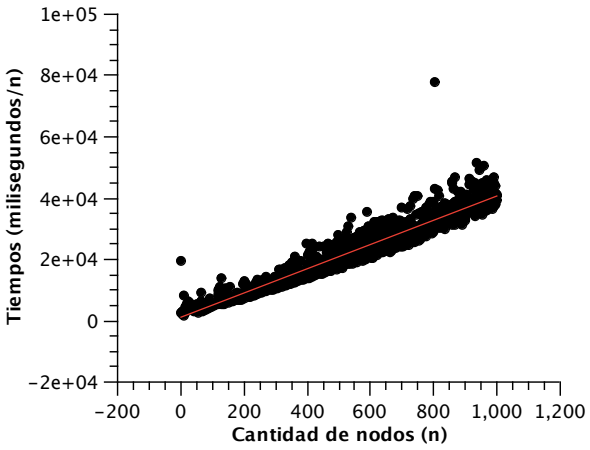
Como podemos ver, la experimentación se condice con el cálculo teórico de la complejidad y arroja que es de $\mathcal{O}(n * log(n) + m)$, aunque en un grafo completo es casi $\mathcal{O}(n^2)$.

3.5.2. Experimentación sobre el complemento del grafo completo

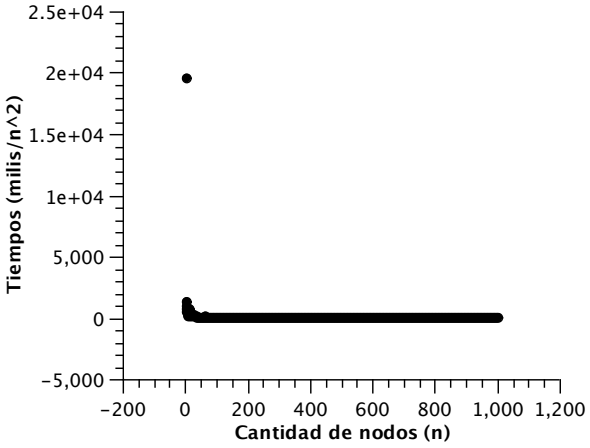
Se crearon instancias de complementos grafos completos con $1 \leq n \leq 1000$ y $m = 0$.
En la implementación sobre matrices arrojo los siguientes resultados:



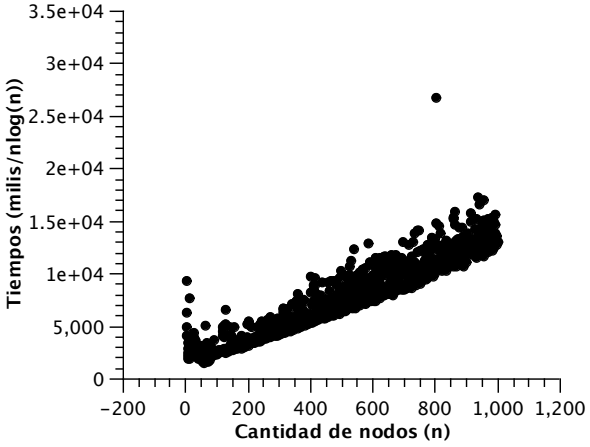
(a) Tiempos sin procesar, en milisegundos



(b) Dividiendo a los tiempos por n



(a) Dividiendo a los tiempos por n^2



(b) Dividiendo a los tiempos por $n + n * log(n)$

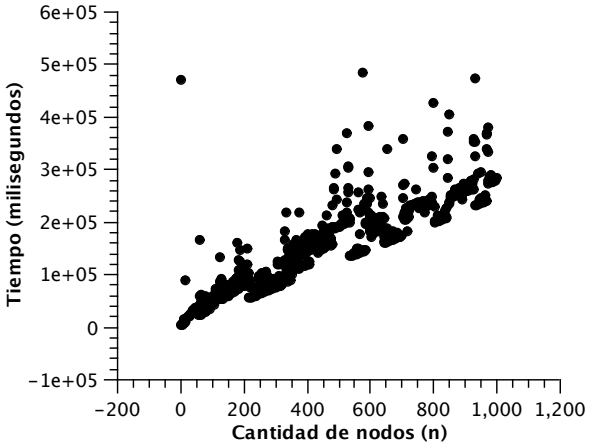
A continuación, adjuntamos una tabla con los últimos 20 valores obtenidos en las instancias, teniendo

en cuenta que los casos fueron previamente ordenados según el tamaño (n):

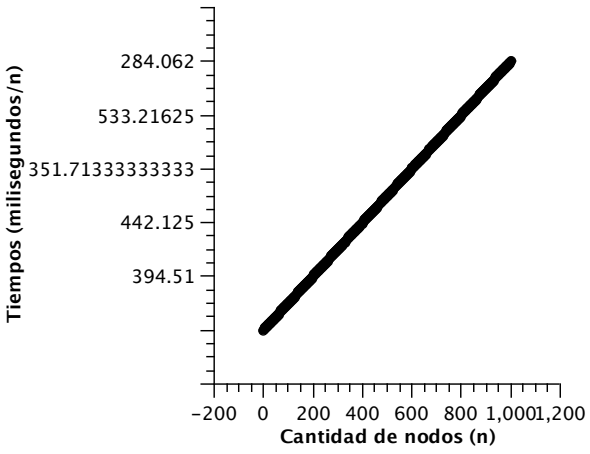
n	Tiempo(milis)	Tiempo(mili/(n))	Tiempo(mili/(n^2)))	Tiempo(mili/($n * log(n) + m$)))
980	41,753,937	42,606.0581632653	43.47556955435235	14,243.67703581269
981	37,549,011	38,276.25993883792	39.01759422919258	12,794.28300537819
982	39,391,112	40,113.14867617108	40.84842024050008	13,406.30148305065
983	45,239,903	46,022.28179043744	46.81819103808488	15,378.93358170479
984	36,000,279	36,585.6493902439	37.18053799821535	12,223.75853853513
985	43,702,767	44,368.29137055838	45.0439506300085	14,821.85954673998
986	40,825,343	41,405.01318458418	41.99291398030849	13,829.89827602756
987	38,862,643	39,374.5116514691	39.89312224059685	13,149.74655118415
988	36,831,350	37,278.6943319838	37.73147199593502	12,447.98660517573
989	44,397,040	44,890.83923154702	45.39013066890497	14,987.61178273532
990	39,102,722	39,497.69898989899	39.89666564636262	13,185.08310395429
991	40,118,814	40,483.16246215944	40.85081984072598	13,512.07184160979
992	46,471,583	46,846.35383064516	47.22414700669875	15,633.62968546942
993	41,389,046	41,680.81168177241	41.97463412061673	13,907.7469205387
994	43,743,891	44,007.93863179074	44.27358011246553	14,682.10400263076
995	38,823,308	39,018.4	39.21447236180904	13,015.57795408459
996	37,960,325	38,112.77610441767	38.26583946226674	12,711.63425257771
997	39,036,157	39,153.61785356068	39.27143215001071	13,056.88500714597
998	40,876,628	40,958.54509018036	41.0406263428661	13,656.80637315606
999	40,732,820	40,773.59359359359	40.81440800159519	13,593.16666151807

Como podemos ver, la experimentación se condice con el cálculo teórico de la complejidad y, aunque el no haya aristas en el grafo, la complejidad que arroja es de $\mathcal{O}(n^2)$.

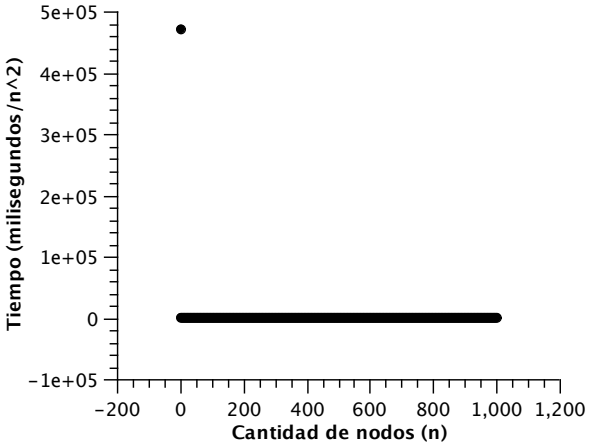
Veamos ahora los resultados en la implementación sobre listas de adyacencia:



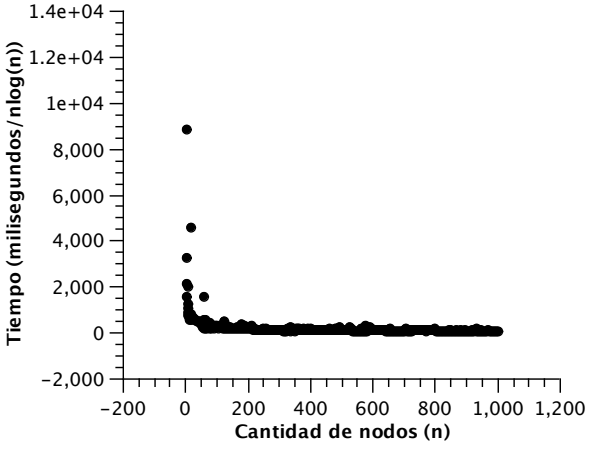
(a) Tiempos sin procesar, en milisegundos



(b) Dividiendo a los tiempos por n



(a) Dividiendo a los tiempos por n^2



(b) Dividiendo a los tiempos por $n + n * log(n)$

A continuación, adjuntamos una tabla con los últimos 20 valores obtenidos en las instancias, teniendo

en cuenta que los casos fueron previamente ordenados según el tamaño (n):

n	Tiempo(milis)	Tiempo(mili/(n))	Tiempo(mili/(n^2)))	Tiempo(mili/($n * log(n) + m$)))
980	283,093	288.87040816327	0.2947657226155768	96.57257621237783
981	279,943	285.36493374108	0.2908918794506428	95.38653274714977
982	280,415	285.5549898167	0.290789195332689	95.43594581360509
983	288,893	293.88911495422	0.2989716327102968	98.20680338813817
984	276,797	281.29776422764	0.2858717116134576	93.98537417420873
985	277,190	281.41116751269	0.2856966167641526	94.00940786565884
986	276,617	280.54462474645	0.2845280169842295	93.70613178730468
987	277,663	281.3201621074	0.2850254935231977	93.95135777670718
988	281,872	285.2955465587	0.2887606746545592	95.26500875949687
989	282,107	285.24469160768	0.2884172817064555	95.23405608103857
990	277,617	280.42121212121	0.283253749617386	93.60993375526336
991	278,533	281.06256306761	0.2836150989582326	93.81029823710891
992	278,372	280.61693548387	0.2828799752861603	93.64786998548107
993	278,950	280.91641490433	0.2828966917465562	93.73412480887504
994	279,345	281.03118712274	0.2827275524373606	93.75874548093792
995	279,336	280.73969849246	0.2821504507461933	93.64785410306024
996	278,955	280.07530120482	0.2812001016112644	93.41263366232546
997	283,107	283.95887662989	0.2848133165796286	94.69414583300491
998	283,434	284.00200400802	0.2845711463006173	94.69477906957285
999	283,640	283.92392392392	0.2842081320559799	94.65501754783944
1,000	284,062	284.062	0.284062	94.68733333333333

Como podemos ver, la experimentación se condice con el cálculo teórico de la complejidad y arroja que es de $\mathcal{O}(n * log(n) + m)$, sin embargo, como el grafo no tiene aristas, aqui la complejidad es de $\mathcal{O}(n * log(n))$ siendo así más eficiente que en la implementación sobre matriz de adyacencia.

4. Ejercicio 4 - Heurística de búsqueda local

4.1. Algoritmo implementado

Sea G=(V,E) un grafo simple, la heuristica de busqueda local propuesta genera una solucion inicial valida, es decir un V' $\subseteq$ V que es dominante e independiente (CID), de dos formas:

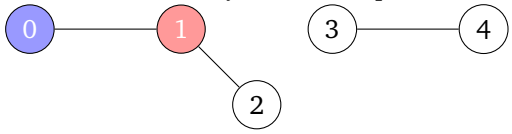
1. **Heuristica constructiva golosa**: procedimiento descripto en el ejercicio anterior.
2. **Procedimiento BFS modificado**: detallado a continuación.

4.1.1. Procedimiento BFS modificado

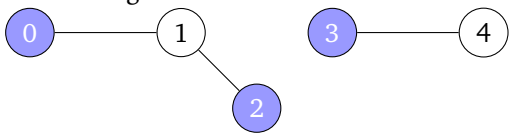
Partimos de incluir un vertice inicial v a V' y luego vamos a ir agregando vertices a V' determinando si el vertice analizado debe incluirse en V'. El BFS modificado funciona de la siguiente manera:

```
Los vertices estan numerados de 0 a n-1.

Creamos un vector, llamado solucionInicial, de tamaño n para guardar el estado de los
vertices (si fue VISITADO o no)

Creamos un vector de tamaño n en donde para cada posicion guardamos si
pertence al CID (INCLUIDO o no)

Al vertice inicial v lo ponemos como VISITADO y INCLUIDO y lo incluimos en la cola.

Luego mientras no este vacia la cola:
    Sacamos el primer elemento de la cola (w) y lo ponemos INCLUIDO.
    Revisamos cada adyacente a w:
        Si algun adyacente esta INCLUIDO entonces hacemos w = NO INCLUIDO.
        Si el adyacente no fue VISITADO entonces lo ponemos como VISITADO y
        lo agregamos a la cola.

Repetimos el procedimiento para el resto de las componentes conexas, empezando por
el vertice de menor numeracion de la componente analizada.
```

A continuacion mostramos un ejemplo del recorrido BFS, en donde el vertice 0 ya fue visitado y se esta analizando sus adyacentes, en particular el vertice 1, el cual es provisoriamiente INCLUIDO:



Para luego ser desmarcado debido a la presencia de un adyacente INCLUIDO, siendo la solucion generada la siguiente:

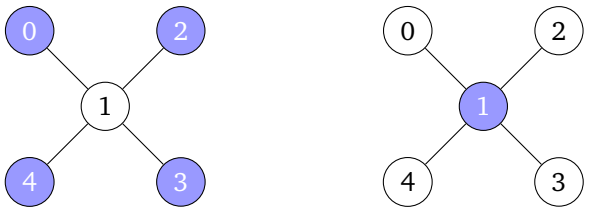


4.1.2. Primer Criterio de Vecindad

El primer criterio de vecindad implementado consiste en generar soluciones vecinas a partir de quitar k vertices que pertenecen al subconjunto CID de la solucion inicial y agregar 1 vertice al subconjunto, donde k $\in \mathbb{N}$ y k ≥ 2. Logrando de esta manera una reduccion en el cardinal del subcojunto CID de, al menos, un vertice.

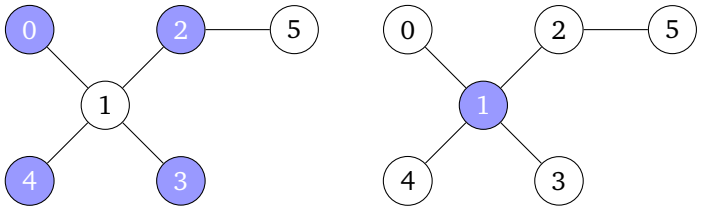
Para llevar adelante exitosamente este intercambio, debemos buscar aquellos vertices no incluidos en CID en la solucion inicial, que tengan, al menos, dos vertices adyacentes incluidos en CID, para poder incluir ese vertice en la solucion vecina y quitar sus adyacentes.

- Ejemplo de un cambio 4 por 1:



Sin embargo, para lograr una solucion valida, los vertices quitados no pueden tener otros vertices adyacentes no incluidos en el subconjunto que, a su vez, no sean adyacentes al vertice agregado.

- Ejemplo de solucion invalida:



El procedimiento de busqueda de los posibles soluciones vecinas funciona de la siguiente manera:

```
Para todo vertice, u, en el Grafo:
  Creamos un vector de tamaño n, llamado solucionAuxiliar, al cual le copiamos
  el contenido de la solucionInicial.
  Si solucionInicial[u] = NO INCLUIDO y |adyacentes a u| > 1 entonces:
     solucionAuxiliar[u] = INCLUIDO
     cantAdyacentesIncluidos = 0
     Para todo adyacente, v, de u:
         Si solucionInicial[v] = INCLUIDO entonces:
             cantAdyacentesIncluidos ++
             solucionAuxiliar[v] = NO INCLUIDO

  Si cantAdyacentesIncluidos > 1 entonces:
     Si esSolucion?(solucionAuxiliar) entonces:
       Buscar Nuevos Vecinos a partir de la solucionAuxiliar
       Interrumpir el ciclo
```

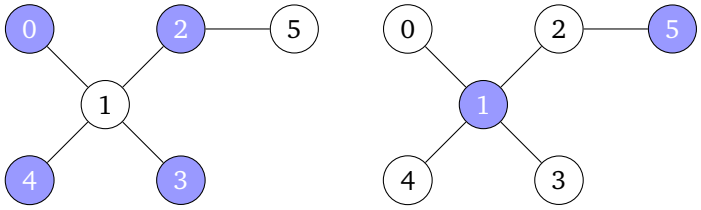
En el procedimiento descripto anteriormente falta detallar el comportamiento de la funciona auxiliar *esSolucion?*, la cual sera descripta en el apartado siguiente, ya que es utilizado por ambos criterios.

4.1.3. Segundo Criterio de Vecindad

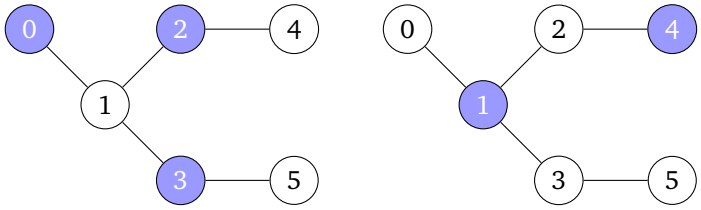
El segundo criterio de vecindad implementado consiste en generar soluciones vecinas a partir de quitar k vertices que pertenecen al subconjunto CID de la solucion inicial y agregar, hasta, k-1 vertices al subconjunto, donde $k \in \mathbb{N}$ y $k \geq 2$. Logrando de esta manera, una reduccion en el cardinal del subcojunto CID de, al menos, un vertice. El caso donde $k = 2$ no difere del criterio aplicado en la primer vecindad, ya que $k - 1 = 1$. Sin embargo a partir de $k \geq 3$ se observa un comportamiento distinto, ya que podemos agregar k-1 vertices para ”salvar”la solucion.

En este caso, para los casos no contemplados en el criterio anterior, debemos buscar vertices no incluidos en CID en la solucion inicial, que tengan, al menos k vertices adyacentes incluidos en CID, donde $k \geq 3$, y que a su vez tengan hasta k-1 vertices que son adyacentes a los adyacentes del vertice buscado que no estan incluidos.

- Ejemplo de un caso 4-2, el cual fallaba en el criterio anterior:



- Caso donde falla el segundo criterio:



El procedimiento de busqueda de los posibles soluciones vecinas funciona de la siguiente manera:

```
Para todo vertice, u, en el Grafo:
  Creamos un vector de tamaño n, llamado solucionAuxiliar, al cual le copiamos
  el contenido de la solucionInicial.
  Si solucionInicial[u] = NO INCLUIDO y |adyacentes a u| > 1 entonces:
     solucionAuxiliar[u] = INCLUIDO
     cantAdyacentesIncluidos = 0
     Para todo adyacente, v, de u:
         Si solucionInicial[v] = INCLUIDO entonces:
             cantAdyacentesIncluidos ++
             solucionAuxiliar[v] = NO INCLUIDO

  Si cantAdyacentesIncluidos > 1 entonces:
     cantCambiosPosibles = cantAdyacentesIncluidos - 2
     arreglarSolucion(solucionAuxiliar, cantCambiosPosibles)
     Si esSolucion?(solucionAuxiliar) entonces:
       Buscar Nuevos Vecinos a partir de la solucionAuxiliar
       Interrumpir el ciclo
```

Falta detallar los procedimientos *arreglarSolucion* y *esSolucion?*, los cuales se pueden realizar en una sola funcion que llameremos *esSolucion?* cuyo comportamiento es el siguiente:

- La funcion recibe como parametros un vector con la solucion a analizar y un entero con la cantidad de cambios posibles a realizar
- Miramos cada vertice del grafo, los cuales o estan INCLUIDOS o NO INCLUIDOS en el subconjunto CID.
- Si el vertice esta INCLUIDO, sus adyacentes NO pueden estar INCLUIDOS. En caso de encontrar algun adyacente INCLUIDO, sabemos que el subconjunto analizado no es solucion valida.
- Si el vertice NO esta INCLUIDO, entonces, al menos, 1 vertice adyacente tiene que estar INCLUIDO. En caso de no encontrar algun vertice adyacente INCLUIDO, tenemos dos casos:
  1. La variable entera que representa la cantidad de cambios posibles es 0. En este caso sabemos que el subconjunto analizado no es solucion valida.
  2. La variable entera que representa la cantidad de cambios posibles es mayor a 0. En este caso el vertice pasa a estar INCLUIDO en el subconjunto, manteniendose la validez de la solucion, ya que el vertice NO tiene adyacentes INCLUIDOS. Tambien reducimos la cantidad de cambios posibles en una unidad.

- En pseudocodigo:

```
Como entrada tenemos el vector solucionAuxiliar con la solucion a analizar y el entero
cantCambiosPosibles, que tiene la cantidad de vertices que podemos incluir.

Creamos una variable booleana, esSolucion inicializadas en true.

Luego, para todo vertice, u, en el Grafo:
  Si solucionAuxiliar[u] = INCLUIDO y |adyacentes a u| > 0 entonces:
     Para todo adyacente, v, de u:
         Si solucionInicial[v] = INCLUIDO entonces:
             esSolucion = false
             Interrumpir el ciclo

  Sino Si |adyacentes a u| > 0, entonces:
       adyacenteIncluido = false
       Para todo adyacente, v, a u:
          Si solucionInicial[v] = INCLUIDO entonces:
             adyacenteIncluido = true
       Si not(adyacenteIncluido) y cantCambiosPosibles = 0 entonces:
         esSolucion = false
         Interrumpir el ciclo
       Sino Si not(adyacenteIncluido) entonces:
         solucionAuxiliar[u] = INCLUIDO
         cantCambiosPosibles --

  Sino entonces:
       esSolucion = (solucionAuxiliar[u] = INCLUIDO)
```

Es necesario aclarar que para el primer criterio de vecindad, la cantidad de cambios posibles es 0.

4.2. Complejidad propuesta

La estructura de datos que utilizamos para representar los grafos son vectores con listas, donde cada posicion del vector representa un vertice y las listas contienen los adyacentes a ese vertice.

A partir de los procedimientos expuestos en el punto anterior, pasamos a analizar la complejidad de la heuristica propuesta, para solo una iteracion:

1. **Solucion Inicial**
   - Heuristica Golosa: $\Theta(n * log(n) + m)$. Justificada anteriormente.
   - BFS modificado: Las cambios implementos en el BFS no alteran su complejidad original, siendo la misma $\mathcal{O}(n + m)$. [4]
2. **Primer Criterio de Vecindad**

   Tenemos un ciclo que se repite n veces, el cual tiene varias operaciones que se realizan internamente:
   - Creacion de un vector tamaño n y copia de contenido: $\Theta(n)$
   - Comparaciones y Asignaciones: $\mathcal{O}(1)$
   - Ciclo de los adyacentes, cuya complejidad, sumada a la del ciclo principal, es $\mathcal{O}(n + m)$.
   - Complejidad de la funcion esSolucion?: $\mathcal{O}(n + m)$. Detallada en el punto 4.

[4] Referencia `https://en.wikipedia.org/wiki/Breadth-first_search`

Por lo tanto la complejidad total de este procedimiento es: $\mathcal{O}(n * (n + n + m) + n + m)$, lo cual es: $\mathcal{O}(n * (n + m))$

3. **Segundo Criterio de Vecindad**

   Misma situacion que el punto anterior, tenemos un ciclo que se repite n veces, el cual tiene varias operaciones que se realizan internamente:

   - Creacion de un vector tamaño n y copia de contenido: $\Theta(n)$
   - Comparaciones y Asignaciones: $\mathcal{O}(1)$
   - Ciclo de los adyacentes, cuya complejidad, sumada a la del ciclo principal, es $\mathcal{O}(n + m)$.
   - Complejidad de la funcion esSolucion?: $\mathcal{O}(n + m)$. Detallada en el punto 4.

   Por lo tanto la complejidad total de este procedimiento es: $\mathcal{O}(n * (n + n + m) + n + m)$, lo cual es: $\mathcal{O}(n * (n + m))$

4. **Procedimiento esSolucion?**

   Tenemos un ciclo que se repite n veces, el cual tiene varias operaciones que se realizan internamente:

   - Comparaciones y Asignaciones: $\mathcal{O}(1)$
   - Ciclo de los adyacentes, cuya complejidad, sumada a la del ciclo principal, es $\mathcal{O}(n + m)$.

   Por lo tanto la complejidad total de este procedimiento es: $\mathcal{O}(n + m)$.

Podemos concluir que la complejidad temporal de la heuristica es independiente del procedimiento utilizado para armar la solucion inicial, y que utilizando el primer o segundo criterio de vecindad la complejidad es la misma: $\mathcal{O}(n * (n + m))$.

Cota superior para la cantidad de iteraciones: Sabemos que cada iteracion de las vecindades reduce, como minimo, en 1 el cardinal del subconjunto CID. Por lo tanto, si partimos de una solucion inicial en donde el cardinal del subconjunto es asistonticamente igual a la cantidad de vertices del Grafo, es posible que iteraremos hasta n-1 veces, hasta alcanzar una solucion de 1 vertice. Es evidente que este es un caso extremo, de dificil realizacion, sin embargo brinda una cota supeior a la cantidad de iteraciones.

4.3. Experimentación computacional

La función que utilizamos para llevar a cabo las mediciones fue `std::clock`[5]. La unidad temporal que utilizamos para este ejercicio fue nanosegundos. La complejidad teórica calculada es de $\mathcal{O}(n * (n + m))$ para cualquier combinacion de solucion inicial y criterio de vecindad.

Para generar las instancias aleatorias utilizamos la función `std::rand`[6] con determinados intervalos de valores para la variables, para obtener instancias coherentes. El detalle de intervalos es el siguiente:

1. Cantidad de nodos n: $2 \leq n \leq 50$.
2. Cantidad de aristas m: $0 \leq m \leq \frac{n*(n-1)}{2}$.
3. Se generan m ejes, asegurandose la validez de los mismos, es decir que no haya ejes repetidos ni loops.

Se generaron 500 instancias construidas de esta forma, en donde se midio no solo el tiempo de ejecucion, sino tambien el tamaño del subconjunto generado, para los 4 tipos contemplados en la heuristica:

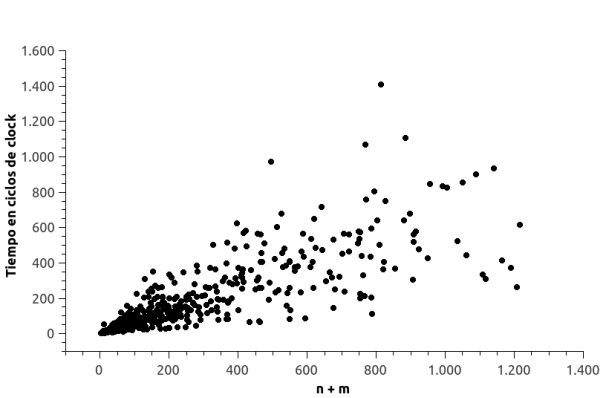
- Tipo B1: Utilizamos el BFS modificado para la solucion original, y el primer criterio de vecindad.
- Tipo G1: Utilizamos la heuristica golosa para la solucion original, y el primer criterio de vecindad.

[5]Referencia `http://en.cppreference.com/w/cpp/chrono/c/clock`
[6]Referencia `http://en.cppreference.com/w/cpp/numeric/random/rand`

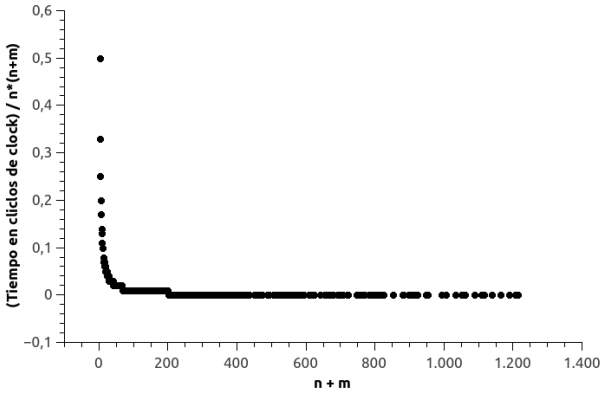
- Tipo B2: Utilizamos el BFS modificado para la solucion original, y el segundo criterio de vecindad.
- Tipo G2: Utilizamos la heuristica golosa para la solucion original, y el segundo criterio de vecindad.

La medicion de tiempos de los distintos tipos, arroja los siguientes graficos para cada tipo:

1. **Tipo B1:**

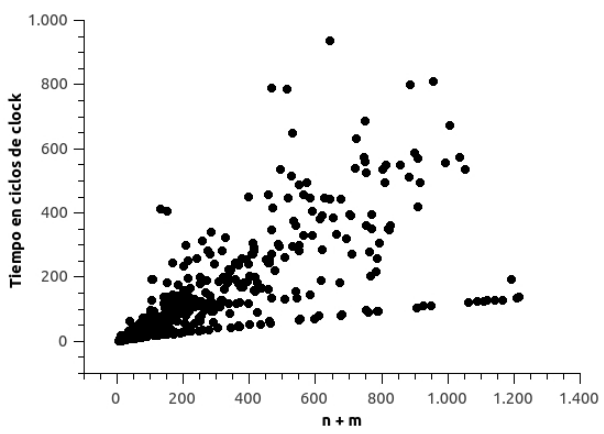


(a) Tiempos sin procesar

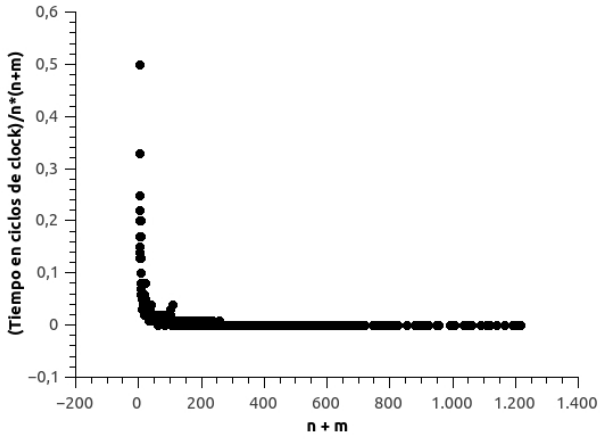


(b) figura (a) / n*(n+m)

2. **Tipo G1:**

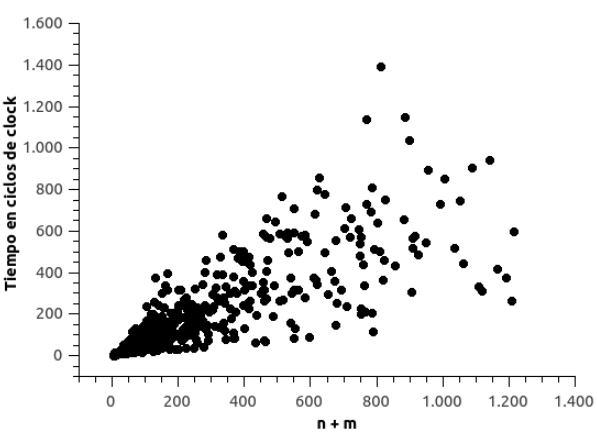


(a) Tiempos sin procesar

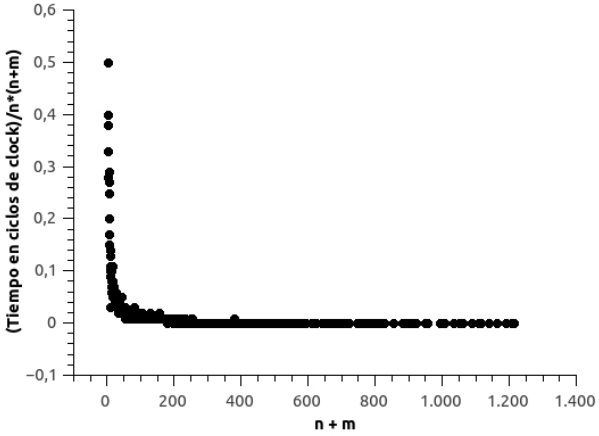


(b) figura (a) / n*(n+m)

3. **Tipo B2:**

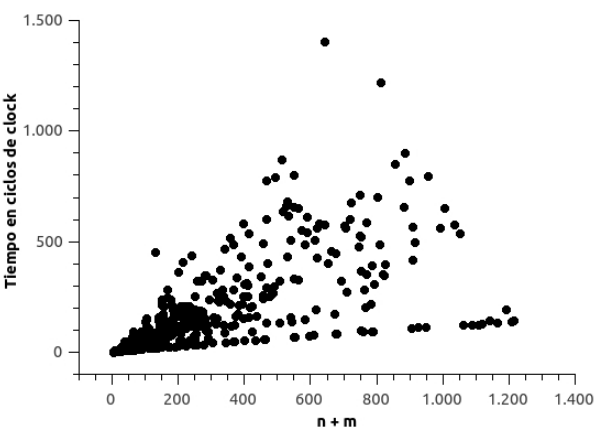


(a) Tiempos sin procesar

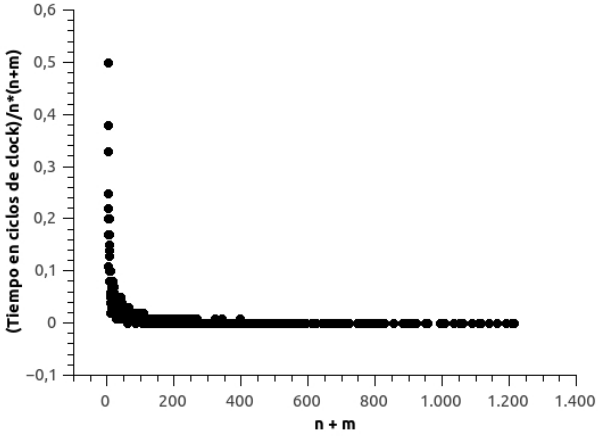


(b) figura (a) / n*(n+m)

4. **Tipo G2:**



(a) Tiempos sin procesar

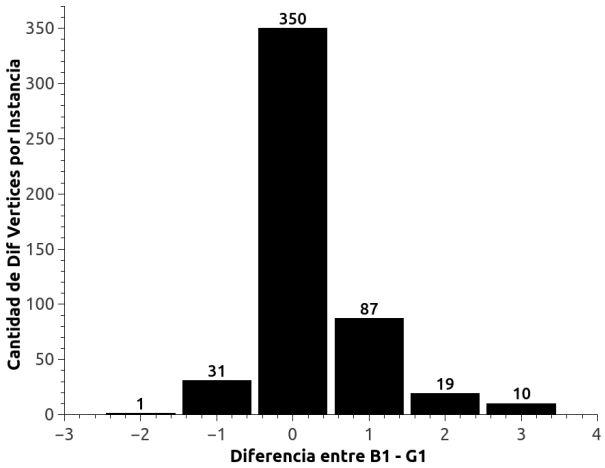


(b) figura (a) / n*(n+m)

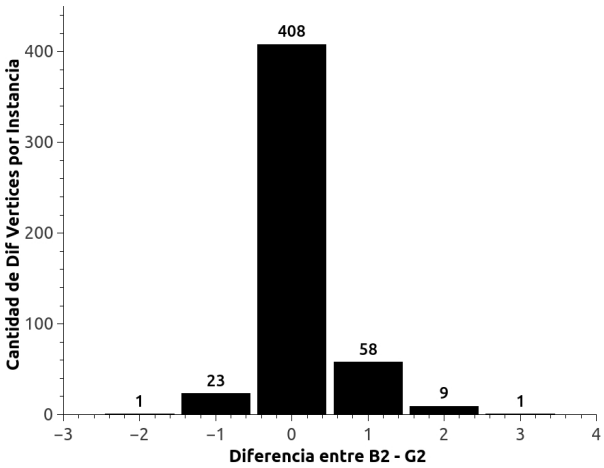
Como podemos ver de los 4 gráficos, al dividir los tiempos por $n * (n + m)$, tienden a un número constante mayor a cero. Entonces nuestro algoritmo tendría complejidad $\mathcal{O}(c * n * (n + m))$, donde c es la constante a la cual converge el gráfico. Por lo tanto concluimos que la complejidad temporal experimental coincide con nuestra predicción de complejidad.

Por el lado de la calidad de las soluciones obtenidas con cada combinacion, debemos comparar por un lado el uso del BFS modificado o de la heuristica golosa como solucion inicial y por el otro el uso del primer criterio de vecindad o el del segundo criterio de vecindad como metodo de mejora de la solucion inicial:

- **BFS modificado vs Heuristica golosa como solucion inicial**. Para realizar la comparacion, tomamos el tamano de la solucion final para cada instancia, usando primero el BFS modificado como solucion incial y luego la heuristica golosa, para despues restar al tamano de la solucion final del tipo B1/B2, el tamano de la solucion final del tipo G1/G2. :



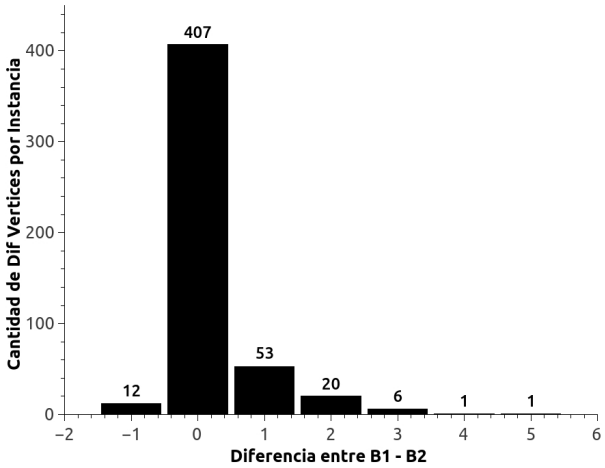
(a) Usando el Primer Criterio de Vecindad



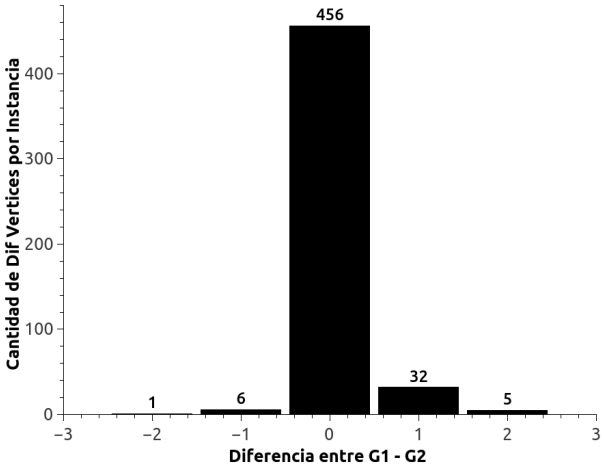
(b) Usando el Segundo Criterio de Vecindad

En ambos casos, se aprecia una paridad entre ambos soluciones, sin embargo hay una leve tendencia hacia la heuristica golosa como mejor procedimiento para construir la solucion inicial.

- **Criterio de Vencindad**. El analisis comparado es el siguiente (utilizando la misma metodologia que en el caso de BFS vs Golosa):



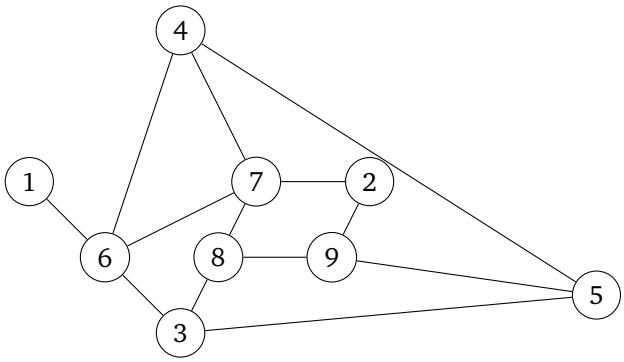
(a) Usando BFS como solucion incial



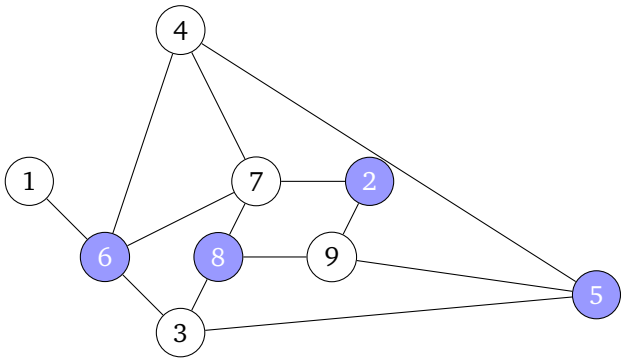
(b) Usando Goloso como solucion incial

Estos resultados nos inclinan a pensar que el segundo criterio de vecindad es el mejor criterio, lo cual es coherente con el hecho que el segundo criterio de vecindad es capaz de .ªrreglar"soluciones que el primer criterio de vecindad da como invalidas. Es decir, podriamos argumentar que como el primer criterio de vecindad es un caso particular del segundo criterio de vecindad (cuando no incluimos ningun vertice al subconjunto, mas alla del candidato) deberia ser siempre mejor al primero. Sin embargo, la experimentacion muestra casos donde el primer criterio es mejor, y eso se da en instancias donde arreglar una solucion invalida nos inhibe de seguir explorando la solucion original en busqueda de un mejor caso. Por ejemplo:

- Grafo de n = 9 y m = 13

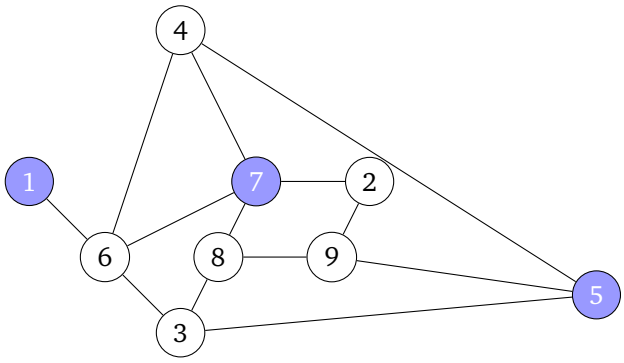


- **Solucion Golosa**: como el mayor grado, 3, es compartido por varios vertices, empezamos por el menor numero de etiqueta, que es el 5, y asi sucesivamente.

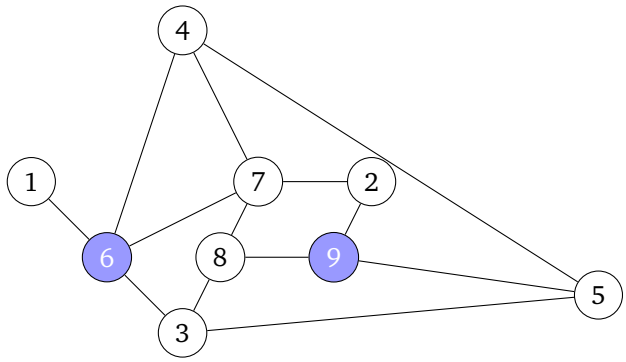


- **Solucion aplicando el segundo criterio de vecindad a la solucion golosa**: tenemos 4 candidatos que cumplen con el hecho de tener 2 o mas adyacentes incluidos: 3, 4, 7, 9.

  1. Vertice 3: incluimos el 3, quitamos el 5, 6 y 8, por lo tanto podemos incluir un vertice mas, que sera el 1. Sin embargo el 4 no es dominado por nadie, por lo cual no es una solucion valida.
  2. Vertice 4: incluimos el 4, y quitamos el 5 y el 6. No podemos incluir ningun vertice mas, por lo tanto el 1 no es dominado por nadie, no es una solucion valida.
  3. Vertice 7: incluimos el 7 y quitamos el 2, 6 y 8. Incluimos el 1, y nos queda una solucion valida.
  4. Vertice 9: no es contemplado, ya que obtuvimos una solucion mejor con el vertice 7. Solucion que ya no es posible mejorar.



- **Solucion aplicando el primer criterio de vecindad a la solucion golosa**: los candidatos son los mismos que en el segundo criterio, sin embargo al no poder incluir vertice mas alla del candidato, la solucion probando con el vertice 7 es invalida, ya que el 1 no es dominado por nadie. Por lo tanto probamos con incluir el 9, y quitar el 2, 5 y 8. Al hacer esto nos queda una solucion valida de menor cardinal que en la solucion golosa original y que si hubiesemos utilizado el segundo criterio



Sin embargo estos tipos de instancias son muy particulares, por lo cual podemos afirmar que la **mejor combinacion** para la **heuristica de busqueda local** planteada es aquella que toma como **solucion inicial** la generada por la **heuristica golosa** y luego es mejorada por el **segundo criterio de vecindad**.

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 conveniemente 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