DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Heurísticas y Metaheurísticas

CIDM

Algoritmos y Estructuras de Datos III
Primer Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Barañao, Facundo	480/11	facundo_732@hotmail.com
Confalonieri, Gisela Belén	511/11	gise_5291@yahoo.com.ar
Mignanelli, Alejandro Rubén	609/11	minga_titere@hotmail.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. Introducción

En el presente trabajo, se pretende analizar y comparar diferentes maneras de encarar el problema de *Conjunto Independiente Dominante Mínimo (CIDM)*, el cual consiste en hallar un conjunto independiente dominante de un grafo, con mínima cardinalidad. En particular se utilizará un algoritmo exacto, una heurística golosa, una heurística de busqueda local, y la metahuristica de GRASP.

A continuación se presentan algunas definiciones que nos serán útiles para comprender y abordar el problema:

- Sea $G = (V, E)$ un grafo simple. Un conjunto $D \subseteq V$ es un conjunto dominante de G si todo vértice de G está en D o bien tiene al menos un vecino que está en D. Por otro lado, un conjunto $I \subseteq V$ es un conjunto independiente de G si no existe ningún eje de E entre dos vértices de I. Definimos entonces un conjunto independiente dominante de G como un conjunto independiente que a su vez es un conjunto dominante del grafo G.
- Un conjunto independiente de $I \subseteq V$ se dice maximal si no existe otro conjunto independiente $J \subseteq V$ tal que $I \subset J$, es decir tal que I está incluido estrictamente en J.

2. Relación con trabajos previos y aplicaciones

En el TP1 de la materia, hemos encontrado y analizado un algoritmo que resolvía un problema que fue llamado **El Señor de los Caballos**. El problema era el siguiente:

Se tiene un juego de mesa cuyo tablero, dividido en casillas, posee igual cantidad de filas y columnas y hace uso de una conocida pieza del popular ajedrez: el caballo. El juego es solamente para un jugador y consiste en, teniendo caballos ubicados en distintos casilleros, insertar en casilleros vacíos la mínima cantidad de caballos extras, de manera tal que, siguiendo las reglas del movimiento de los caballos en el ajedrez, todas las casillas se encuentren ocupadas o amenazadas por un caballo. Aspectos a tener en cuenta:

- *Se conoce la cantidad de filas y columnas del tablero.*
- *Se conoce la cantidad de caballos que ocupan el tablero inicialmente.*
- *Para cada uno de estos caballos, se sabe su ubicación en el tablero.*
- *Una casilla se considera amenazada si existe un caballo tal que en una movida pueda ocupar dicha casilla.*

Observemos que si consideramos a cada casilla como un nodo, y que dos nodos son adyacentes cuando un caballo puede llegar de uno a otro con un movimiento, entonces **El Señor de los Caballos** puede ser visto como la búsqueda de un conjunto dominante mínimo que contenga a los nodos/casillas que tienen un caballo preubicado. Sin embargo, dado un grafo, no es correcto decir que la cardinalidad del CIDM es menor o igual a la cardinalidad de un conjunto. Para probar esto, podemos observar la figura X. Corriendo el backtracking de CIDM que hemos creado (y que detallaremos en próximas secciones) la solución es la figura X2, mientras que la Figura X3 es el conjunto dominante mínimo. Entonces, no podemos afirmar que El Señor de los Caballos pueda adaptarse a un problema de CIDM.

De todos modos, cabe preguntarse en qué situaciones de la vida real podría aplicarse este problema. Veamos algunos ejemplos realistas y alguno no tanto.

Gaseosas: Una empresa de bebidas gaseosas tiene sus actividades divididas en áreas, las cuales se interrelacionan de cierta manera (no necesariamente todas con todas). Esta empresa está por sacar una nueva bebida sabor cola y quiere hacerlo antes que su rival, quien está más adelantado en la producción. Para lograr sacar el producto antes que su contrincante, nuestra empresa debe lograr una mejora en el trabajo

de sus diferentes áreas. Por cómo está estructurada la empresa, si un área tiene una mejora considerable en tiempo, todas las áreas que se relacionan con ella se verán beneficiadas, pero este beneficio ya no impacta en las áreas relacionadas con estas últimas. Queremos entonces lograr impulsar mejoras considerables en las áreas que sean necesarias para que toda la empresa funcione más rápido y logre sacar su producto a tiempo. Para ello se contratarán especialistas con sueldos sumamente importantes, por lo cual se desea que la cantidad de áreas a mejorar considerablemente sea mínima (para minimizar la inversión en especialistas). Además, sabemos que estos especialistas son excelentes, pero muy soberbios/tercos, y ponerlos a cargo de áreas relacionadas puede devenir en discusiones que retrasarían a la empresa, por lo cual procuraremos colocarlos en áreas ”no adyacentes”.

Detectives: Se tiene un caso de asesinato, y debemos resolverlo. Nuestras investigaciones previas nos han dado información de todas las personas que tuvieron algo que ver con el incidente, y sabemos cuáles se conocen entre sí y cuáles no, además de que sabemos que toda persona que conoce a otra, sabe lo que hizo dicha persona el día del incidente. Debido a que somos detectives principiantes, el alquiler por día de nuestra oficina nos sale caro, y entrevistar a un testigo, independientemente del volumen de información obtenida, nos toma un día. Si para resolver el caso debiéramos escuchar información sobre todas las personas involucradas, ¿a quienes deberíamos llamar de testigos, de manera tal que reduzcamos el alquiler de oficina al mínimo? (Estos testigos no pueden conocerse entre sí, para evitar complot y falsos datos).

Caballeros del Zodíaco: Milo de Escorpio[1], guardián de la casa de Escorpio, ha recibido muchas quejas de parte del gran patriarca, debido a que por su casa pasa todo el mundo. Cansado de ser tantas veces derrotado, nos pide ayuda, y para esto nos explica la verdad sobre su gran técnica, la aguja escarlata. Lejos de lo que se cuenta en el animé *Los caballeros del zodíaco*[2], el verdadero poder de la aguja escarlata es el siguiente:

Milo nos explica que el cuerpo de cada ser humano puede ser dividido en sectores energéticos. Estos sectores energéticos pueden tener una correspondencia entre sí, las cuales no son necesariamente a nivel local (por ejemplo, una porcion del dedo indice de un humano, puede estar conectada a una porción de la cabeza). Cuando la aguja escarlata toca un sector energético de un cuerpo, éste y todos aquéllos sectores que tengan una correspondencia, se inmovilizan. Pero esto sucede sólo si la aguja escarlata impacta contra un sector energético “sano”. Si el sector en cuestión ya estuviese inmovilizado, entonces la aplicación de la aguja escarlata no hace ningun efecto.

Dado que el uso de una aguja escarlata, resulta en un fuerte uso del cosmos de Milo, se nos pide que, dado un oponente y la información sobre todos sus sectores energéticos y correspondencias, nosotros le fabriquemos un algoritmo tal que le diga en qué sectores del cuerpo debe usar la aguja escarlata, de manera tal que deba utilizar la mínima cantidad de agujas escarlata posibles. Milo nos asegura que el tiempo de dicho algoritmo no tiene importancia, dado que le pidió prestada la habitación del tiempo[3] a Kamisama[4], que está equipada con una notebook y un enchufe para dejar la bateria cargada (eso sí, no tiene wifi dado a una muy mala señal, por lo que debemos darle un pendrive con el código).

3. Propiedades

En esta sección demostraremos ciertas propiedades que serán usadas para elaborar una conclusión que nos ayudará a abordar el problema propuesto.

[1] http://es.wikipedia.org/wiki/Milo_de_Escorpio
[2] http://es.wikipedia.org/wiki/Saint_Seiya
[3] http://es.dragonball.wikia.com/wiki/Habitacion_del_Tiempo
[4] http://es.wikipedia.org/wiki/Kamisama_(personaje)

3.1. Todo conjunto independiente maximal es dominante.

Lo probaremos por absurdo. Supongamos que existe algún grafo $G = (V, E)$ tal que tiene un conjunto independiente maximal C que no es un conjunto dominante. Como C no es dominante, entonces existe un nodo $v \in V$ tal que $v \notin C$, y que además dentro del grafo original, v no es vecino de ningún elemento de C. Pero entonces, si añadimos a v a C se obtiene un conjunto $C' = C + v$ que es independiente, o sea que $C \subset C'$, lo cual es absurdo, puesto que habíamos dicho que C era maximal. El absurdo proviene de suponer que el conjunto no es dominante, por lo tanto, el conjunto debe ser dominante.

3.2. Todo conjunto independiente dominante es independiente maximal.

Lo probaremos por absurdo. Supongamos que existe un grafo $G = (V, E)$ tal que tiene un conjunto independiente dominante I que no es independiente maximal. Como I no es independiente maximal, podemos suponer que existe algún conjunto $J \subseteq V$ independiente, tal que $I \subset J$. Entonces, existe v un nodo que pertenece a J pero no a I, tal que v no es vecino de ningún elemento de I, lo cual es absurdo, ya que suponer eso es decir que I no era dominante.

3.3. Conclusión

Por las propiedades anteriormente demostradas, se puede concluir que el problema de buscar un CIDM es idéntico al problema de buscar un conjunto independiente maximal mínimo(CIMM), o sea, de todos los conjuntos independientes maximales que son posibles formar dado un grafo cualquiera, aquel cuya cardinalidad es la menor. Por esta razón, nuestros algoritmos buscarán encontrar o aproximar un CIMM para un grafo dado.

4. Plataforma de pruebas

Para toda la experimentación se utilizará un procesador Intel Core i3, de 4 nucleos a 2.20 GHZ.
El software utilizado será Ubuntu 14.04, y G++ 4.8.2.

5. Acerca de los Experimentos

En esta sección explicaremos de qué manera se tomó el tiempo de ejecución de los programas en los experimentos, cómo fueron procesadas estas mediciones, y de qué manera se generaron las instancias que luego llamaremos $aleatorias$, $estrellas$ y $galaxias$.

5.1. Mediciones de tiempo

Para medir el tiempo de ejecución de los programas implementados se utilizaron las funciones `clock`[5] y `difftime`[6] provistas por la librería `ctime`[7] de C++ como se muestra a continuación:

```
clock_t start = clock();
programa_a_evaluar;
clock_t end = clock();
double t = difftime(end,start);
```

[5] http://www.cplusplus.com/reference/ctime/clock/
[6] http://www.cplusplus.com/reference/ctime/difftime/
[7] http://www.cplusplus.com/reference/ctime/

De esta manera, la variable `t` contendrá el tiempo incurrido por `programa_a_evaluar` en ciclos de clock.

5.2. Procesamiento de las mediciones

En los experimentos relativos al tiempo de ejecución los programas, cada instancia fue evaluada 20 veces y se ha recopilado el tiempo incurrido cada vez. Luego, fueron considerados outliers los 2 valores más grandes y los 2 valores más chicos, y se calculó el promedio de los tiempos obtenidos ignorando estos 4 valores (excepto en los casos llamados $aleatorios$).

5.3. Generación de instancias particulares

Dentro de la experimentación, la aleatoriedad provee una herramienta muy útil ya que permite dar una idea de cómo funcionan los programas al ser aplicados a instancias generales. Sin embargo, la generación de aleatoriedad es un tema controversial. Por este motivo, buscamos utilizar instancias pseudo-aleatorias para analizar el comportamiento de nuestros programas. Para ello, utilizaremos las funciones `srand`[8] y `random`[9] provistas por la librería `cstdlib`[10] de C++.

Pasaremos ahora a mostrar cómo fueron generadas dichas instancias.

5.3.1. Aleatorios

Para generar un grafo que llamaremos $aleatorio$, dada una cantidad determinada de vértices y de aristas, se van tomando de forma pseudo-aleatoria pares de nodos no conectados para colocarlos como extremos de una arista, hasta completar la cantidad de aristas fijada.

5.3.2. Estrellas

Para generar un grafo que llamaremos $estrella$, dado el grado δ del vértice que llamaremos $central$, se genera 1 nodo que será el $central$ y δ nodos que se conectan a él. Luego, por cada uno de estos δ vértices se agregan k nodos conectados a él, donde k es un número generado pseudo-aleatoriamente entre 1 y $\delta - 1$.

Más adelante definiremos mejor a este tipo de instancias y haremos un análisis sobre ellas.

5.3.3. Galaxias

Para generar un grafo que llamaremos $galaxia$, dado el grado δ del vértice que llamaremos $central$, se genera 1 nodo que será el $central$ y δ nodos que se conectan a él. Luego, por cada uno de estos δ vértices se agregan k nodos conectados a él, donde k es un número generado pseudo-aleatoriamente entre δ y $2 \times \delta$.

Más adelante definiremos mejor a este tipo de instancias y haremos un análisis sobre ellas.

6. Algoritmo Exacto

6.1. Desarrollo de la idea y correctitud.

Para resolver el problema de CIDM de manera exacta hemos decidido utilizar la técnica de backtracking. Por todo lo dicho en la seccion de propiedades, nuestro backtracking se encargará de, dado un grafo, ver todos los posibles conjuntos independientes maximales, y tomará aquel que sea menor en cardinalidad.

[8] `http://www.cplusplus.com/reference/cstdlib/srand/`
[9] `http://www.cplusplus.com/reference/cstdlib/rand/`
[10] `http://www.cplusplus.com/reference/cstdlib/`

Para esto, le daremos a los nodos un orden en particular, y para cada nodo consideraremos las siguientes dos opciones o "ramas":

- **Tomar el nodo como parte del conjunto solución.** Esta rama sólo será considerada cuando el nodo no sea adyacente a otro nodo que ya fue colocado anteriormente en el conjunto. Por eso, en caso de tomar el nodo actual, marcaremos a este nodo y a todos sus vecinos, de manera de no tomarlos nuevamente en el futuro de esa rama, puesto que si tomásemos a alguno de ellos en el conjunto, este no sería independiente.
- **No tomar el nodo como parte del conjunto solución.** En este caso no se hará nada, y se avanzará hacia el próximo nodo, de existir éste.

Luego de la elección tomada para un determinado nodo, consultaremos las siguientes posibilidades:

- Si el nodo tratado en el último paso es el último nodo y no están todos los nodos marcados, el conjunto obtenido no es un independiente maximal, por lo que no lo tomamos en cuenta.
- Si todos los nodos quedaron marcados, el conjunto obtenido es independiente maximal. Se verá entonces la cardinalidad de este conjunto, y de ser mejor que el de la mejor solución obtenida hasta el momento, se lo guardará como nueva mejor solución.
- De no haber visto el último nodo, y de existir nodos no marcados aún, avanzaremos al siguiente nodo y repetiremos el procedimiento.

Para poder afirmar que nuestro algoritmo es correcto, basta con poder probar que todo conjunto que forma es independiente maximal, y que realmente observa todo conjunto independiente maximal de un grafo:

- Podemos afirmar que este procedimiento encuentra **conjuntos independientes**, puesto que sólo se toman aquellos nodos que no están marcados, o sea, que no tienen ninguna arista en común con los elementos del conjunto.
- Podemos afirmar que este procedimiento encuentra conjuntos independientes que son **maximales** puesto que el programa deja de agregar nodos cuando todos estos están marcados, lo que significa que todos los nodos del grafo, o bien son adyacentes a algún elemento del conjunto, o bien están dentro del conjunto. Por eso, no podemos tomar ningún nuevo elemento de modo tal que el nuevo conjunto sea independiente.
- Podemos afirmar que se observan **todos** los posibles conjuntos independientes maximales por lo siguiente: sea C un conjunto independiente maximal del grafo G, conformado por los vertices $v_1, v_2, ..., v_h$, entonces, por cómo esta diseñado nuestro algoritmo, se llegaría a observar este conjunto independiente maximal cuando estemos en la rama que solo toma a $v_1, v_2, ..., v_h$ y no toma a los demás. Notemos que esta rama existe, pues $v_1, v_2, ..., v_h$ son nodos independientes, y por lo tanto, al tomar uno de ellos, los otros no se marcan y quedan disponibles para ser tomados como parte de la solución.

Para mejorar la velocidad de ejecución del algoritmo, se han aplicado las siguientes podas:

- **Poda Clásica:** Si en la rama actual que está revisando nuestro algoritmo, la cantidad de elementos del conjunto maximal de esta rama es mayor a la cantidad de elementos de la mejor solución encontrada hasta el momento, esta rama deja de ser considerada, puesto que, de conseguir una solución, seguro no es la mejor.
- **Nodos solitarios:** Si el grafo tiene algún nodo con grado 0, no tiene sentido considerar la opción de no tomarlo como parte de la solución, por lo cual dicha rama no será revisada cuando el nodo cumple esta característica.

```
CIDM_EXACTO(lista_nodos cidm, lista_nodos cidm_sol, nodo, int n, int res_sol, int res)
 1  if se encontró una solución mejor a la obtenida hasta el momento
 2      cidm_sol ← cidm
 3      res_sol ← res
 4      return
 5  if se llegó al final y no se encontró una solución
 6      return
 7  if nodo no está "tomado"
 8      cidm ← agregar nodo
 9      incrementar res
10      marcar a nodo y a sus vecinos como "tomados"
11      CIDM_EXACTO(cidm, cidm_sol, nodo_siguiente, n, res_sol, res)
12  if nodo se tomó en la rama anterior
13      cidm ← sacar nodo
14      decrementar res
15      marcar a nodo y a sus vecinos como "no tomados"
16  CIDM_EXACTO(cidm, cidm_sol, nodo_siguiente, n, res_sol, res)
```

Figura 1: Algoritmo exacto para CIDM

- **El nodo decisivo:** Si durante el procesamiento de un nodo, al tomarlo, cubre a todos los que estaban libres hasta el momento, entonces no se considerará la rama resultante de no tomarlo, ya que en el mejor de los casos uno de los nodos siguientes también cubre a todos y esto no mejora la cardinalidad del conjunto hallado, y en casos peores, será necesario tomar más de uno de los nodos siguientes para lograr un conjunto independiente maximal.

6.2. Análisis de complejidad.

Trataremos ahora de justificar que la complejidad de nuestro algoritmo es $\mathcal{O}(2^n \cdot n)$.

Observemos que, como también dijimos anteriormente, nuestro backtracking irá generando distintas soluciones mediante la generación de distintas ramas en base a si un nodo es tomado como parte de la solución o no. Así que partiendo de esto tenemos 2^n casos posibles.

Como se muestra en el pseudocódigo (Figura 1), sea cual sea la ruta que tomemos, nos vemos obligados a marcar a este nodo y a sus vecinos como "no tomados" o "tomados". Esto no es más que recorrer una lista de vecinos realizando operaciones $\mathcal{O}(1)$, llegando a tener un costo de, en el peor caso, $\mathcal{O}(n)$. Luego por cada uno de los 2^n casos tendriamos un costo de $\mathcal{O}(n)$.

Por otro lado hay que mencionar que si se llegara a una solución, esta tendría que ser comparada con la solución óptima obtenida hasta ese momento y, en caso de ser mejor, reemplazarla mediante la copia de todos los elementos que componen a esta solución: $\mathcal{O}(n)$. A este costo tendríamos que incurrir una cantidad k de veces, donde k es la cantidad de soluciones que resultan ser mejores que las que se poseían hasta el momento, la cual se ve acotada, tal vez de manera bruta pero efectiva, por 2^n (esto nos permite absorberla dentro del coste total $\mathcal{O}(2^n \cdot n)$). Esta cota resulta casi trivial dado que implicaria que cada nodo puede ser una solución y la a vez no puede (hay casos que se contradicen).

Teniendo todo esto en cuenta, y siendo $T(n)$ la complejidad de nuestro algoritmo tenemos:

$$T(n) = \mathcal{O}(n) * 2^n + k\mathcal{O}(n)$$
$$T(n) = \mathcal{O}(2^n * n)$$

6.3. Experimentación y gráficos.

En esta sección, trataremos de mostrar de manera empírica la complejidad de nuestro algoritmo y la utilidad de las podas.

6.3.1. Complejidad

Si fuera cierto que nuestro algoritmo es exponencial, entonces correr instancias muy grandes puede tardar horas, días, o siglos, dependiendo el caso, por lo tanto, se ha decidido correr instancias de a lo sumo 20 nodos. Por otro lado, debido a la gran variedad de posibles grafos que aún con pocos nodos se pueden generar, se ha decidido probar la complejidad acudiendo a la ley de los grandes números, y por lo tanto generando muchos grafos de manera aleatoria. Para esto, se han generado 100 grafos $aleatorios$ de entre 5 y 15 nodos. Luego, se ha corrido el algoritmo exacto sin podas, tomando los tiempos de ejecución, y se ha calculado un promedio sobre aquellos grafos que tenían la misma cantidad de nodos. Esto dio como resultado el gráfico de la Figura 2.

Luego, se ha tomado cada uno de estos valores, se lo ha dividido por la cantidad de n nodos, y se le ha aplicado la raíz n-ésima, obteniendose el gráfico de la Figura 3. En esta Figura se puede observar cómo el número obtenido se mantiene dentro de una constante cercana al 2, lo cual tiende a sugerir que realmente nuestro algoritmo tiene una complejidad del estilo $\mathcal{O}(2^n \cdot n)$, o sea, que es exponencial.

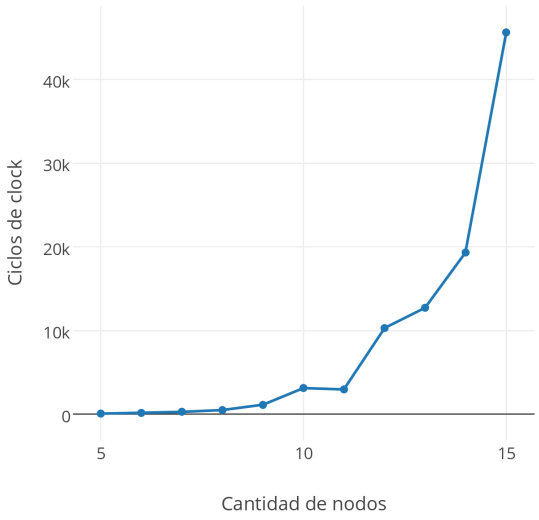

Figura 2: Exacto - Tiempo de ejecución

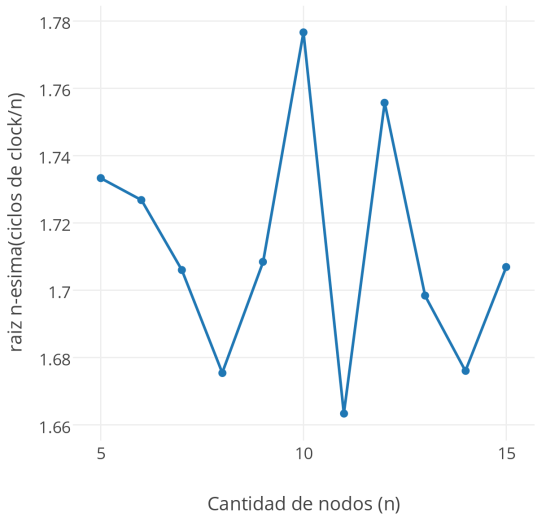

Figura 3: Exacto - Tiempo de ejecución

6.3.2. Podas

Además de la poda clásica, se han generado dos podas más, llamadas “nodos solitarios” y “nodo decisivo”, explicadas anteriormente. Trataremos de ver en esta sección la utilidad de las mismas. Para ello veremos cómo repercuten estas podas en algunas familias de grafos, y tomaremos las instancias $aleatorias$ que usamos para mostrar la complejidad, para ver cómo mejoran con cada poda, y con la combinación de varias.

Para ello, cada instancia se ha corrido de 5 maneras:

1. Sin ninguna poda

2. Con todas las podas
3. Con la poda Clásica
4. Con la poda Clásica y la poda de los nodos solitarios
5. Con la poda Clásica y la poda del nodo decisivo

Los experimentos propuestos son los siguientes:

Grafo compuesto por nodos de grado 0 Se han creado 18 instancias, con grafos que tienen de 3 a 20 nodos, todos de grado 0.

Hipótesis Dadas las características de estos grafos, esperamos ver en los resultados que la poda de los *nodos solitarios* tiene mayor impacto en la eficiencia que las demás.

Luego de ejecutar el programa con las instancias mencionadas, el gráfico obtenido con los resultados es el que se muestra en la Figura 4.

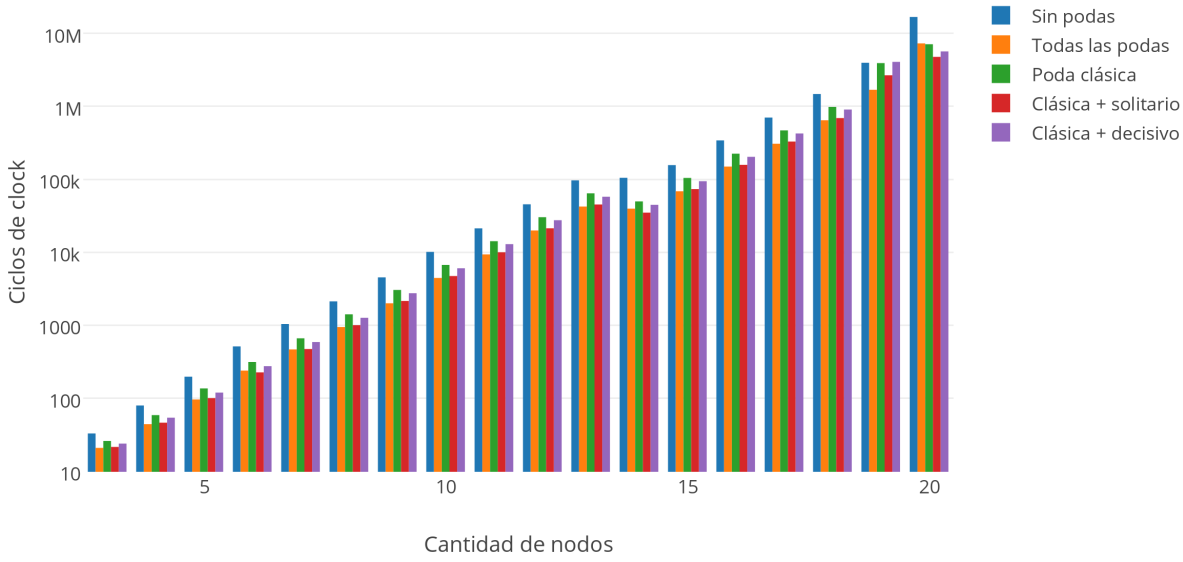

Figura 4: Exacto - Nodos solitarios

Por lo visto en este gráfico, la poda de los *nodos solitarios* es la que más optimiza el tiempo de ejecución para cuando el grafo está formado por nodos de grado 0.

Grafos completos Se han creado 18 instancias, con grafos completos, de entre 3 y 20 nodos.

Hipótesis Dadas las características de estos grafos, esperamos ver en los resultados que la poda del *nodo decisivo* tiene mayor impacto en la eficiencia que las demás.

El gráfico obtenido con los resultados es el que se muestra en la Figura 5.
Por lo visto en este gráfico, la poda del *nodo decisivo* es la que mayor impacto parece tener en el tiempo de ejecución cuando el grafo tiene nodos de grado máximo.

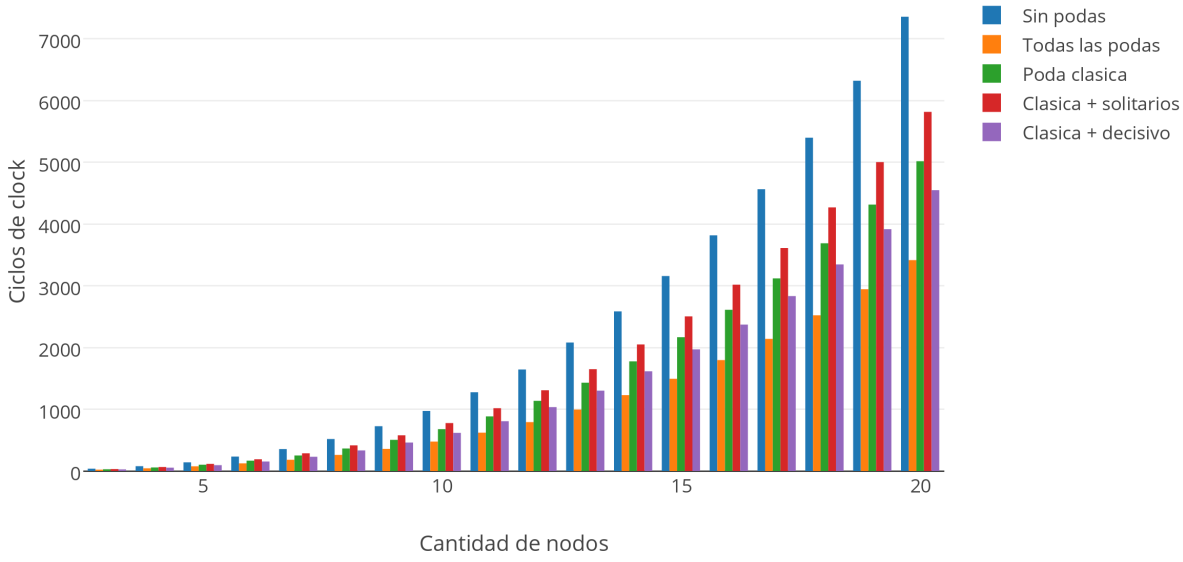

Figura 5: Exacto - Grafos completos

Grafo compuesto por varios K_2 Se han creado 9 instancias, con grafos compuestos por n K_2, con n desde 1 a 9. El orden de los nodos fue aleatorizado para no depender de un rotulado en particular.

Hipótesis Esperamos ver en los resultados que ninguna combinación de las podas produce un impacto relevante en la eficiencia, ya que la poda de los *nodos solitarios* funciona sólo sobre nodos de grado 0 (que estos grafos no poseen) y la poda del *nodo decisivo* funcionará de manera diferente dependiendo el orden en el que se consulten los nodos dentro del programa.

El gráfico obtenido con los resultados es el que se muestra en la Figura 6.

Por lo visto en este gráfico, se trata de una familia en la cual nuestras podas no surten efectos significativos.

Grafos aleatorios Se han utilizado las mismas instancias creadas para contrastar la complejidad del algoritmo.

Hipótesis Esperamos ver en los resultados que la combinación de todas las podas mejora la performance de manera más significativa que cada poda por separado, ya que considerarlas a todas descarta más ramas de decisión.

El gráfico obtenido con los resultados es el que se muestra en la Figura 7.

Por lo visto en este gráfico, en general, parecería que usar todas las podas mejora el tiempo de ejecución de nuestro algoritmo, más que utilizar las podas por separado.

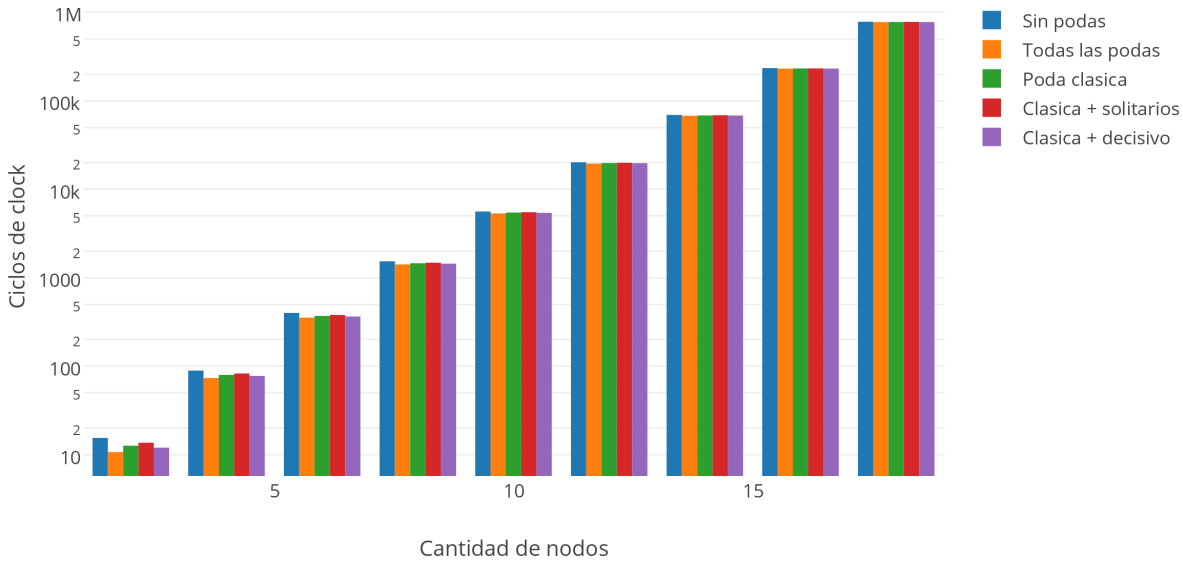

Figura 6: Exacto - Grafos formados por K_2

7. Heurística Golosa Constructiva

7.1. Desarrollo de la idea.

Sea G un grafo cualquiera, I un conjunto independiente de ese nodo, y n_1,n_2 nodos de G que no pertenecen a I y no tienen aristas en común con ningún elemento de I, diremos que n_1 es óptimo si no existe ningún n_2 tal que $(\#(Vecinos(n_2)) - \#(Vecinos(n_2) \cap Vecinos(I))) > (\#(Vecinos(n_1)) - \#(Vecinos(n_1) \cap Vecinos(I)))$. Definimos $Vecinos(\alpha)$ como el conjunto de nodos a los cuales α lleva una arista si α es un nodo, y si α es un conjunto, entonces es el conjunto de nodos a los cuales les llega un arista desde por lo menos un elemento de α. De manera más informal, podemos decir que un nodo óptimo es aquél que más vecinos "libres" tiene, siendo un nodo "libre" uno que no está en el conjunto solución ni es adyacente a un nodo de la solución.

Nuestra heurística se basa en formar un conjunto independiente maximal de la siguiente manera: Primero, considerando al conjunto independiente vacío, tomamos a un nodo óptimo, y lo agregamos como nuevo elemento de nuestro conjunto independiente. Luego con nuestro nuevo conjunto independiente, tomamos un nodo óptimo, y lo agregamos a nuestro conjunto independiente. Repetimos esto hasta que no exista un nodo óptimo, puesto que nuestro conjunto independiente se transformó en maximal, y por lo tanto en un conjunto dominante.

7.2. Análisis de complejidad.

Pasemos a analizar la complejidad del algoritmo en cuestión, tal vez abstrayendonos del peor caso, pero si tratando de maximizar los costos de los distintos pasos a realizar. Notemos primero que buscar el nodo "óptimo" nos toma $\mathcal{O}(n)$ ya que se trata de recorrer los nodos del grafo haciendo comparaciones sobre la cantidad de vecinos que possen $\mathcal{O}(1)$. (Creo que no sería errado decir que si entro n veces a

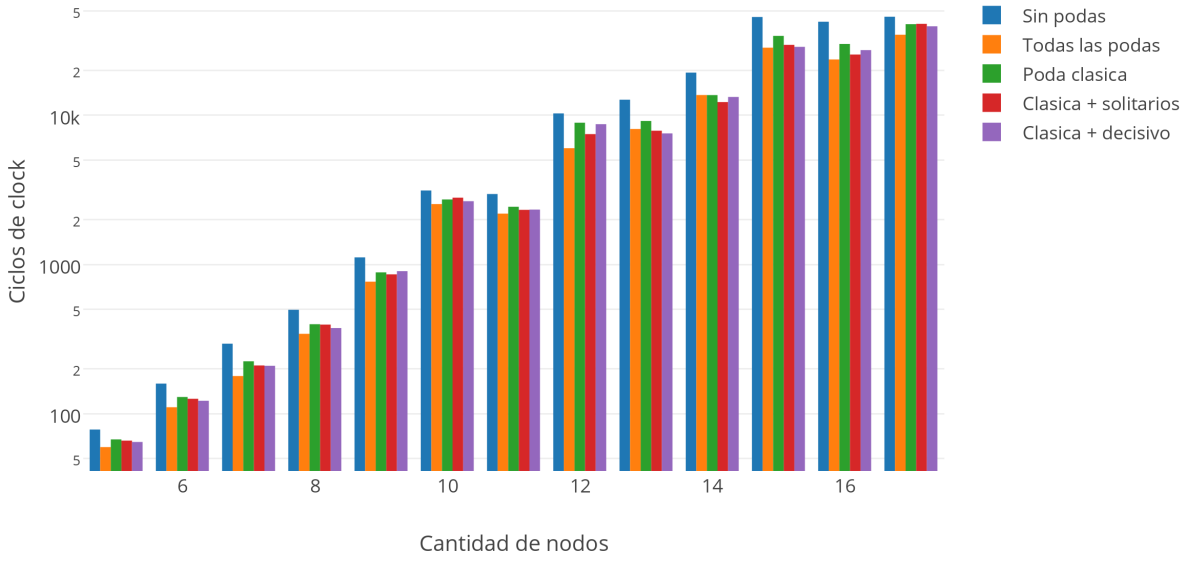

Figura 7: Exacto - Grafos aleatorios

buscar al optimo, es porque todos tenian grado 0, sino se irian eliminando posibilidades, por ende la complejidad seria $\mathcal{O}(n^2)$).

Supongamos un caso en el que debo entrar $n-1$ veces a buscar el "óptimo" $\mathcal{O}(n)$. Luego tendría un costo de $\mathcal{O}(n^2)$.

A este se le suma el costo de marcar a los nodos elegidos y a sus vecinos como "tomados". Para esto debemos tener en cuenta que marcarlos toma $\mathcal{O}(1)$, y que cada nodo puede tener como máximo $n-1$ vecinos, pero la suma total de nodos vecinos a marcar como tomados es siempre $n-1$ ($\mathcal{O}(n)$). Luego, y ya metiéndonos un poco con lo que respecta a la implementación, debemos actualizar los grados de los nodos vecinos de los vecinos del nodo tomado como "óptimo". Como ya dijimos, la suma de los vecinos del nodo tomado puede llegar a $n-1$, y estos a su vez podrían llegar a tener $n-1$ vecinos. Como debo recorrerlos para actualizarlos estaríamos hablando de una complejidad de $\mathcal{O}(n^2)$

Por último la solución del algoritmo consiste integrar cada nodo "óptimo" a la solución final, lo que puede llegar a tomar $\mathcal{O}(n)$.

Siendo $T(n)$ la complejidad de nuestro algoritmo tenemos:

$$T(n) = \mathcal{O}(n^2) + \mathcal{O}(n) + \mathcal{O}(n^2)$$
$$T(n) = 2\mathcal{O}(n^2)T(n) = \mathcal{O}(n^2)$$

7.3. Instancias no óptimas.

A continuación presentaremos dos familias de grafos para las cuales nuestra heurística golosa no siempre encuentra una solución óptima.

Familia 1 - Estrellas

La forma de los grafos pertenecientes a esta familia responde a las siguientes características:

```
CIDM_GOLOSO(lista_nodos cidm_sol)
1  res ← 0
2  while no se hayan "tomado" todos los nodos
3       elegido ← nodo "óptimo"
4       cidm_sol ← agregar elegido
5       incrementar res
6       marcar a elegido y a sus vecinos como "tomados"
7  return res
```

Figura 8: Heurística golosa constructiva para CIDM

- Existe un único vértice v con grado máximo. Sea δ_{max} el grado de este nodo.
- Para cada vértice w adyacente a v, $1 < \delta(w) < \delta_{max}$, y sus nodos adyacentes (salvo v) tienen grado 1.

Dado que el algoritmo propuesto va tomando en cada paso el nodo "óptimo", o sea, aquél que más nodos "libres" cubre, en un grafo como el descrito primero tomará al vértice v con grado máximo δ_{max}. Luego, para cumplir con la independencia y la dominancia, deberá tomar a los vértices adyacentes a los vecinos de v. Como el grado de cada vecino de v es menor a δ_{max}, el peor de los casos sería que cada uno de ellos tuviera grado $\delta_{max} - 1$. En este caso, cada vértice adyacente a v tendría $\delta_{max} - 2$ vecinos además del mismo v, y entonces la solución hallada por nuestra heurística estaría conformada por $\delta_{max} \times (\delta_{max} - 2)$ nodos.

Sin embargo, para un grafo como el detallado, existe una solución mejor conformada por δ_{max} nodos, y corresponde al conjunto compuesto por los vecinos del nodo v. Veamos que esta solución es óptima.

- Como ya dijimos, si tomamos al nodo v nos vemos obligados a tomar como parte del conjunto a los vértices adyacentes a cada vecino de v, y ya notamos que esta solución no es buena.
- Analicemos qué sucede si no tomamos a v como parte de la solución. Supongamos por un momento que se elimina dicho nodo. En ese caso, nos quedan δ_{max} componentes conexas que son $K_{1,n}$. Trivialmente, para un grafo no conexo, el CIDM es la unión de los CIDMs de cada una de sus componentes conexas, y en el caso de los grafos $K_{1,n}$ el CIDM consta del elemento perteneciente a la partición con un único nodo. Luego, el CIDM del grafo total contiene exactamente al vértice correspondiente a cada componente. Si volvemos a considerar a v como parte del grafo, vemos que el mismo queda "cubierto" por cada uno de los vértices que ubicamos en el conjunto, así que dicho CIDM es también un CIDM para el grafo original.

Comparemos entonces la solución hallada con la solución óptima analizada. Como vimos, en el peor de los casos la solución hallada por nuestra heurística estaría conformada por $\delta_{max} \times (\delta_{max} - 2)$ nodos. Pero observemos que para valores de δ_{max} muy grandes, podemos decir que dicha solución contiene aproximadamente δ^2_{max} elementos, y esa es la diferencia con una solución óptima.

La Figura 9 muestra un ejemplo de grafo perteneciente a esta familia, y la Figura 10 muestra en color verde la solución hallada por nuestro algoritmo, la cual claramente es peor que la solución formada por los vértices que quedaron en rojo en la misma Figura.

Familia 2 - Circuitos

Como miembros de esta familia consideraremos a los circuitos simples. Para este tipo de grafos, la calidad de la solución hallada por nuestro algoritmo dependerá de la cantidad de vértices que tenga y de la forma en que se hayan rotulado los mismos.

Consideremos un circuito C_n y supongamos por un momento que los vértices están rotulados como $v_0, v_1, ..., v_{n-1}$ tal que existe una arista entre v_i y v_j cuando $j = (i + 1)mod(n)$.

1. Una forma de armar un conjunto independiente maximal C para este grafo podría ser la siguiente:

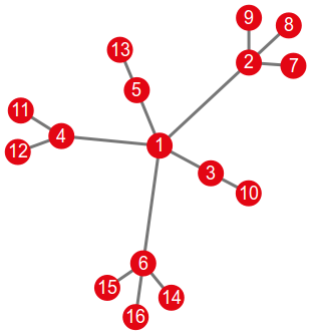

Figura 9: Grafo perteneciente a la Familia 1

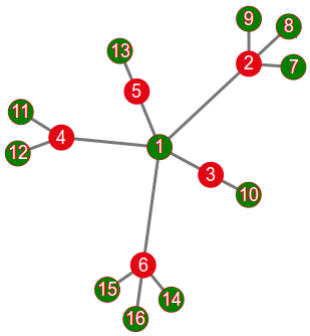

Figura 10: Solución hallada por nuestra heurística

- Si $n \equiv 0(3)$, tomar $C = \{v_i | i \equiv 1(3)\}$.
- Si $n \equiv 1(3)$, tomar $C = \{v_i | i \equiv 1(3)\} \cup \{v_{n-1}\}$.
- Si $n \equiv 2(3)$, tomar $C = \{v_i | i \equiv 1(3)\}$.

2. Veamos que C efectivamente es independiente maximal, y por lo tanto, dominante.

Independencia: Cuando $n \equiv 0(3)$, el conjunto C está formado por los vértices v_i tal que $i \equiv 1(3)$, por lo cual v_i se comunica con v_{i-1} y con v_{i+1}. Como $i - 1 \equiv 0(3)$ y $i + 1 \equiv 2(3)$, entonces los vértices incluidos en C no se comunican entre sí. Cuando $n \equiv 1(3)$ ó $n \equiv 2(3)$, todo lo antes escrito también vale, considerando que v_{n-1} (que en ambos casos pertenece a C) tiene por vecinos a v_{n-2} y v_0, y sucede que $0 \equiv 0(3)$ y $n - 2 \equiv 2(3)$ ó $n - 2 \equiv 0(3)$, por lo cual sigue cumpliéndose la independencia.

Maximalidad: Supongamos que C no es maximal, es decir, que podemos incluir al menos un vértice más y seguir teniendo un conjunto independiente. Analicemos los siguientes casos:

- Agregar un vértice v_i tal que $i \equiv 1(3)$. Pero todos ya forman parte de C. ABSURDO.
- Agregar un vértice v_i tal que $i \equiv 2(3)$. Pero todos se encuentran conectados al vértice v_{i-1} y como en este caso $i - 1 \equiv 1(3)$, entonces por definición $v_{i-1} \in C$. ABSURDO.
- Agregar un vértice v_i tal que $i \equiv 0(3)$. Pero entonces v_i o bien se conecta al vértice v_{i+1} con $i + 1 \equiv 1(3)$ y por lo tanto $v_{i+1} \in C$, o bien $i = n - 1$ cuando $n \equiv 1(3)$ así que también pertenece a C. ABSURDO.

Luego, C es un conjunto independiente maximal, y por consiguiente, dominante.

3. Veamos ahora que C también es mínimo (o sea, un CIDM de C_n).

Primero, diremos que, dentro de un grafo, un vértice v "cubre" a un vértice w o bien cuando $v = w$, o bien cuando v y w son adyacentes. Por lo tanto, un vértice v cubre exactamente a $\delta(v) + 1$ vértices.

Particularmente, dado un CIDM para un determinado grafo $G = (V, E)$, entre todos los vértices pertenecientes a dicho conjunto "cubren" a todos los vértices de G (porque es dominante). Notemos además, que dos vértices diferentes del CIDM pueden cubrir a un mismo vértice dentro de G (es decir, pueden tener vecinos en común). Entonces, si tomamos un CIDM S para G, y sabiendo que cada vértice $v \in S$ cubre a $\delta(v) + 1$ vértices dentro de G, podemos decir que

$$\sum_{v \in S} \text{nodos cubiertos por } v = |S| + \sum_{v \in S} \delta(v) \geq |V|$$

Volviendo ahora a nuestro conjunto independiente maximal C del circuito C_n definido anteriormente. En principio, fácilmente vemos su cardinalidad:

- Cuando $n \equiv 0(3)$, $|C| = \frac{n}{3}$.

- Cuando $n \equiv 1(3), |C| = \frac{n-1}{3} + 1$.
- Cuando $n \equiv 2(3), |C| = \frac{n-2}{3} + 1$.

Para probar que C es un CIDM, supongamos que no lo es. Es decir, que existe un conjunto independiente y dominante C' con menor cardinalidad que C. Podemos decir que $|C'| = |C| - k$ para algún $k \in [1, |C|)$. Como C_n es un circuito simple, cada vértice tiene grado exactamente 2. Entonces,

$$\sum_{v \in C'} \text{nodos cubiertos por } v = |C'| + \sum_{v \in C'} \delta(v) = |C'| + 2|C'| = 3|C'| = 3(|C| - k) = 3|C| - 3k$$

- Cuando $n \equiv 0(3), 3|C| - 3k = 3\frac{n}{3} - 3k = n - 3k < n$ pues k es al menos 1. ABSURDO.
- Cuando $n \equiv 1(3), 3|C| - 3k = 3(\frac{n-1}{3} + 1) - 3k = n + 2 - 3k < n$ pues k es al menos 1. ABSURDO.
- Cuando $n \equiv 2(3), 3|C| - 3k = 3(\frac{n-2}{3} + 1) - 3k = n + 1 - 3k < n$ pues k es al menos 1. ABSURDO.

Luego, concluimos que C es un CIDM de C_n.

4. Supongamos ahora que vamos armando al conjunto C tomando los nodos v_i de la manera indicada anteriormente, pero de forma secuencial, de menor a mayor valor de i. Notamos entonces que:

- Cuando $n \equiv 0(3)$, cada vértice v_i que se incluye en C cubre exactamente 3 vértices "libres" (sus dos vecinos y él mismo).
- Cuando $n \equiv 1(3)$, cada vértice v_i que se incluye en C tal que $i \equiv 1(3)$ también cubre exactamente 3 vértices "libres", y por último v_{n-1} sólo se cubre a sí mismo, ya que sus vecinos fueron cubiertos anteriormente por v_1 y v_{n-3}.
- Cuando $n \equiv 2(3)$, cada vértice v_i que se incluye en C tal que $i \equiv 1(3)$ también cubre exactamente 3 vértices "libres", y por último v_{n-1} se cubre a sí mismo y a v_{n-2}, ya que v_0 fue cubierto anteriormente por v_1.

Por lo tanto, podemos decir que al momento de tomar cada vértice de C, el mismo califica como nodo "óptimo".

5. Definimos ahora la siguiente función $f : V \to \mathbb{N} \in (0, n]$ para rotular los vértices de C_n:

$$f(v_i) = \begin{cases} \dfrac{i-1}{3} + 1 & \text{si } i \equiv 1(3) \\ \\ k \in (|C|, n] \text{ no utilizado aún} & \text{si no} \end{cases}$$

6. Por una decisión implementativa, nuestra heurística golosa va tomando secuencialmente los nodos "óptimos" en el orden en el que fueron rotulados (o sea que siempre toma el nodo "óptimo" con rotulado menor). Entonces, si tomamos un circuito C_n donde los vértices son $v_0, v_1, ..., v_{n-1}$ tal que existe una arista entre v_i y v_j cuando $j = (i + 1) mod(n)$, podemos formar un conjunto C como se indica en el punto 1, y sabemos por los puntos 2 y 3 que se trata de un CIDM. Si además, rotulamos los vértices como se indica en el punto 5, podemos ver que, por lo dicho en el punto 4, nuestro algoritmo encuentra una solución exacta, idéntica al conjunto C. La Figura 11 muestra un ejemplo de circuito rotulado de la manera indicada, y la Figura 12 muestra la respuesta hallada por nuestro algoritmo.

A. Ahora, considerando el circuito C_n inicial, tomemos el siguiente conjunto de vértices $C = \{v_i | (i \equiv 1(4)) \vee (i \equiv 3(4))\}$. Básicamente, tomamos los nodos v_i tal que i es impar.

B. Veamos que C es independiente maximal, y por lo tanto, dominante.

Independencia: El conjunto C está formado por los vértices v_i tal que $i \equiv 1(4)$ ó $i \equiv 3(4)$. Si n es impar, sabemos que v_i se comunica con $v_{(i-1)}$ y con $v_{(i+1)}$. Si $i \equiv 1(4)$ entonces $i - 1 \equiv 0(4)$ y $i + 1 \equiv 2(4)$, y si $i \equiv 3(4)$ entonces $i - 1 \equiv 2(4)$ y $i + 1 \equiv 0(4)$, así que los vértices incluidos en C no se comunican entre sí. Si n es impar, todo lo antes escrito también vale, considerando que v_{n-1} (que pertenece a C) tiene por vecinos a v_{n-2} y v_0, y sucede que $0 \equiv 0(4)$ y $n - 2 \equiv 2(4)$ ó $n - 2 \equiv 0(4)$, por lo cual sigue cumpliéndose la independencia.

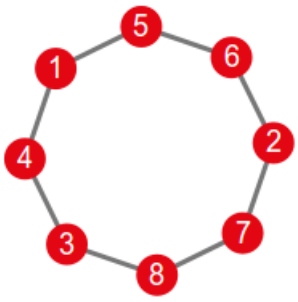

Figura 11: Circuito simple rotulado “en orden”

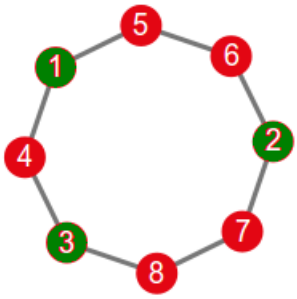

Figura 12: Solución hallada por nuestra heurística

Maximalidad: Supongamos que C no es maximal, es decir, que podemos incluir al menos un vértice más y seguir teniendo un conjunto independiente. Analicemos los siguientes casos:

- Agregar un vértice v_i tal que $i \equiv 1(4)$. Pero todos ya forman parte de C. ABSURDO.
- Agregar un vértice v_i tal que $i \equiv 3(4)$. Pero todos ya forman parte de C. ABSURDO.
- Agregar un vértice v_i tal que $i \equiv 0(4)$. Pero todos se encuentran conectados al vértice v_{i+1} y como en este caso $i + 1 \equiv 1(4)$, entonces por definición $v_{i+1} \in C$. ABSURDO.
- Agregar un vértice v_i tal que $i \equiv 2(4)$. Pero todos se encuentran conectados al vértice v_{i-1} y como en este caso $i - 1 \equiv 1(4)$, entonces por definición $v_{i+1} \in C$. ABSURDO.

Luego, C es un conjunto independiente maximal, y por consiguiente, dominante.

C. Veamos ahora que el conjunto C es un conjunto independiente máximo. Para ello observemos la cardinalidad de C (la cual es trivial dado que sólo tomamos a los vértices v_i con i impar):

- Cuando n es par,$|C| = \frac{n}{2}$.
- Cuando n es impar, $|C| = \frac{n-1}{2}$.

Para probar que C es un conjunto independiente máximo, supongamos que no lo es. Es decir, que existe un conjunto independiente maximal C' con mayor cardinalidad que C. Podemos decir que $|C'| = |C| + k$ para algún $k \in [1, (n - |C|))$. Pero dada la cardinalidad de C, tener un conjunto con más elementos implicaría que dicho conjunto contiene a más de la mitad de los vértices del grafo, y como se trata de un circuito simple y cada vértice tiene grado 2, obliga a que al menos dos vértices de C' sean adyacentes. Esto es absurdo puesto que supusimos C' independiente. Entonces, C es un conjunto independiente máximo.

D. Supongamos ahora que vamos armando al conjunto C tomando los nodos v_i de la manera indicada anteriormente, pero de forma secuencial, de menor a mayor valor de i con $i \equiv 1(4)$, y luego de menor a mayor valor de i con $i \equiv 3(4)$. Notamos entonces que:

- Cuando tomamos los v_i tal que $i \equiv 1(4)$, cada uno cubre exactamente 3 vértices “libres” (sus dos vecinos y él mismo).
- Cuando pasamos a tomar los v_i tal que $i \equiv 3(4)$ $n \equiv 1(3)$, cada uno sólo cubre 1 vértice “libre” que es él mismo. Esto sucede porque sus dos vecinos fueron cubiertos anteriormente por los v_i con $i \equiv 1(4)$.

Por lo tanto, podemos decir que al momento de tomar cada vértice de C, el mismo califica como nodo “óptimo”.

E. Definimos ahora la siguiente función $f : V \to \mathbb{N} \in (0, n]$ para rotular los vértices de C_n:

$$f(v_i) = \begin{cases} \frac{i-1}{4} + 1 & \text{si } i \equiv 1(4) \\ \frac{i-3}{4} + \left\lfloor \frac{n}{4} \right\rfloor + 1 & \text{si } i \equiv 3(4) \wedge (n \equiv 0(4) \vee n \equiv 1(4)) \\ \frac{i-3}{4} + \left\lceil \frac{n}{4} \right\rceil + 1 & \text{si } i \equiv 3(4) \wedge (n \equiv 2(4) \vee n \equiv 3(4)) \\ k \in \left(\frac{n}{2}, n\right] \text{ no utilizado aún} & \text{si } (i \equiv 0(4) \vee i \equiv 2(4)) \wedge (n \equiv 0(4) \vee n \equiv 2(4)) \\ k \in \left(\frac{n-1}{2}, n\right] \text{ no utilizado aún} & \text{si } (i \equiv 0(4) \vee i \equiv 2(4)) \wedge (n \equiv 1(4) \vee n \equiv 3(4)) \end{cases}$$

F. Como hemos decidido nuestra heurística golosa vaya tomando secuencialmente los nodos "óptimos" en el orden en el que fueron rotulados, entonces si tomamos un circuito C_n donde los vértices son $v_0, v_1, ..., v_{n-1}$ tal que existe una arista entre v_i y v_j cuando $j = (i+1) mod(n)$, podemos formar un conjunto C como se indica en el punto A, y sabemos por los puntos B y C que se trata de un conjunto independiente máximo, y por consiguiente, dominante. Si además, rotulamos los vértices como se indica en el punto E, podemos ver que, por lo dicho en el punto D, nuestro algoritmo encuentra una solución idéntica al conjunto C.

Hay una excepción para el caso $n \equiv 2(4)$. En estos casos, el vértice $n-1 \equiv 1(4)$, pero el vértice v_{n-1} no será tomado por nuestro algoritmo dado que se conecta al vértice v_0 que ya fue cubierto por v_1 y sólo podría cubrir a su otro vecino y a él mismo como nodos "libres", así que en su lugar se tomará al vértice v_{i-2} que puede cubrir a 3 vértices "libres" (él mismo, v_{n-1} y v_{n-3}, ya que $n-3 \equiv 3(4)$ y aún no fue tomado). Por consiguiente, al comenzar a tomar a los v_i tales que $i \equiv 3(4)$, v_{n-3} no será tomado en cuenta ya que no está "libre". Luego, cuando $n \equiv 2(4)$, la solución hallada por nuestra heurística toma a todos los v_i tal que $i \equiv 1(4)$ salvo v_{n-1}, luego toma a v_{i-2}, y finalmente toma a todos los v_i tal que $i \equiv 3(4)$ salvo v_{n-3}. El tamaño del conjunto hallado, entonces, difiere sólo en una unidad respecto al conjunto planteado en el punto A.

La Figura 13 muestra un ejemplo de circuito rotulado de la manera indicada, y la Figura 14 muestra la respuesta hallada por nuestro algoritmo.

Para poder comparar las soluciones halladas por nuestra heurística para los rotulados propuestos, aproximaremos la cardinalidad del conjunto encontrado a $\frac{n}{3}$ para el primer caso y $\frac{n}{2}$ para el último, ya que a valores de n muy grandes, la diferencia respecto a $\frac{n-1}{3} + 1$ y a $\frac{n-2}{3} + 1$ para el primero, y a $\frac{n-1}{2}$ en el segundo, es despreciable. Entonces podemos decir que la mejor solución para un circuito simple C_n contiene $\frac{n}{3}$ elementos mientras que la peor solución hallada para el mismo circuito contiene $\frac{n}{2}$ elementos. En otras palabras, para un rotulado como el propuesto en E, el conjunto encontrado por nuestro algoritmo es un $50\,\%$ peor que el exacto.

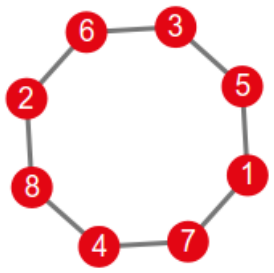

Figura 13: Circuito simple rotulado "no en orden"

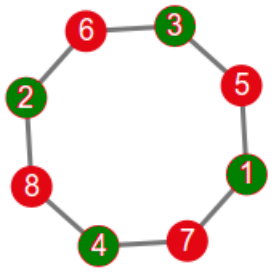

Figura 14: Solución hallada por nuestra heurística

7.4. Instancias óptimas.

A continuación presentaremos una familia de grafos para la cual nuestra heurística golosa siempre encuentra una solución óptima.

La forma de los grafos pertenecientes a esta familia responde a las siguientes características:

- Existe un único vértice v que llamaremos "central", con grado $\delta(v)$.
- Para cada vértice w adyacente a v, $\delta(w) > \delta(v)$, y sus nodos adyacentes (salvo v) tienen grado 1.

Dado que el algoritmo propuesto va tomando en cada paso el nodo "óptimo", o sea, aquél que más nodos "libres" cubre, en un grafo como el descrito se irá formando el conjunto solución con los vértices adyacentes al nodo "central" v, y por lo tanto tendrá $\delta(v)$ elementos. Veamos que esta solución es la mejor. Para esto, veamos que la única diferencia entre esta familia de grafos, y la familia de los grafos estrella, es que el nodo central tiene menor grado que los nodos periféricos. También notemos, que en este caso, nuestra heurística ha tomado la solución óptima descripta para los grafos estrellas. Dado que la explicación sobre por qué esa solución era realmente óptima no tomaba en cuenta el grado del nodo central y de sus periféricos (salvo que implícitamente pedía que estos últimos tengan al menos grado dos, cuestión que en este caso también sucede), podemos usarla para probar que nuestro goloso realmente halla un óptimo.

La Figura 15 muestra un ejemplo de grafo perteneciente a esta familia, y la Figura 16 muestra en color verde la solución hallada por nuestro algoritmo.

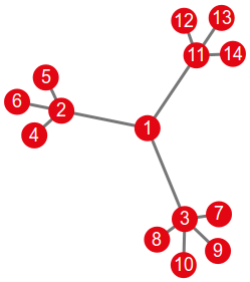

Figura 15: Grafo perteneciente a la Familia óptima

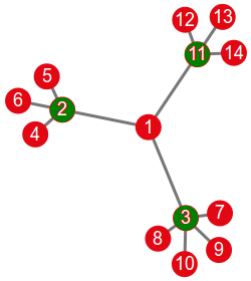

Figura 16: Solución hallada por nuestra heurística

7.5. Experimentación y gráficos.

En esta sección, trataremos de mostrar de manera empírica que nuestro algoritmo posee complejidad cuadrática, y exponer una estimación de la calidad de las soluciones que presenta el mismo.

7.5.1. Complejidad:

Para empezar tratamos de analizar distintas instancias de grafos particulares, como son los grafos "solitarios" y los grafos "completos". Tal vez breve explicacion de que caracteristicas tiene cada uno.

Se puede observar que ambas funciones de complejidad, al ser divididas por n nos perimten ver lo que serían rectas. En cambio, al ser divididas por n^2 asemejan a una curva cuadrática. Inicialmente esto parece justificar nuestra afirmación sobre que la complejidad de nuestro algoritmo es $\mathcal{O}(n^2)$.

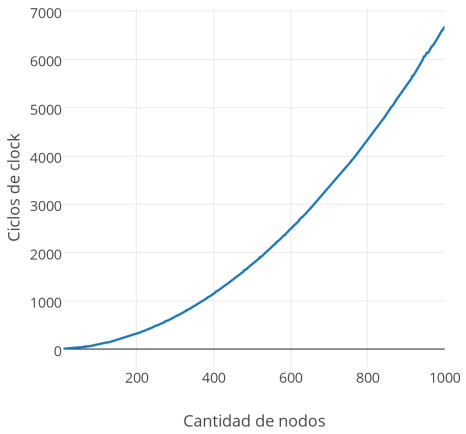

Figura 17: Goloso - Solitarios

Para un análisis más general del algortimo planteamos también distintas instancias generadas de manera aleatoria. Parecería ser, de nuevo, que la curva descripta por la complejidad en estos casos, es una función cuadrática.

Todo lo previamente indicado nos permitiría llegar a la conclusión de que, resultararía dificil describir un contexto para el algoritmo que se pudiera calificar como el peor caso. Todas las corridas realizadas estarían diciendo que, cualquiera sea el caso, la complejidad de nuestro algoritmo es $\mathcal{O}(n^2)$, que era lo que queríamos observar en definitiva.

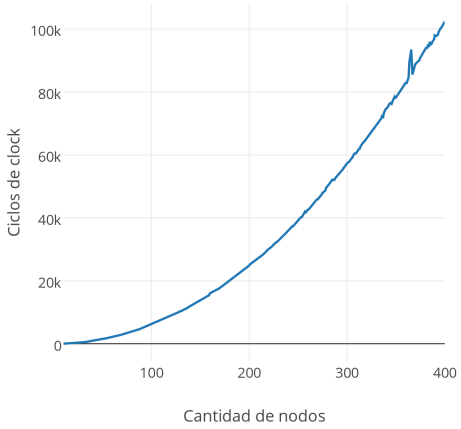

Figura 18: Goloso - completos

8. Heurística de Busqueda Local

8.1. Desarrollo de la idea.

La heurística de búsqueda local que hemos diseñado, parte de una solución inicial, y a partir de ahí irá encontrando nuevas soluciones "vecinas". Si una solución "vecina" resulta ser mejor, es decir, es un conjunto independiente maximal con menos elementos que la solución anterior, entonces la reemplazará. Estos pasos se repitirán hasta que se encuentre una solución que no pueda mejorarse, es decir, una solución óptima local.

Se analizarán dos posibles soluciones iniciales:

- La solución hallada por el algoritmo goloso presentado anteriormente.
- Una solución hallada de la siguiente manera: tomando los nodos del grafo en cierto orden, si el nodo actual no forma parte del conjunto solución ni es adyacente a un nodo del conjunto solución, entonces agregarlo al mismo; en caso contrario, avanzar al siguiente nodo.

Para cada solución factible S, se define $N(S)$ como el conjunto de "soluciones vecinas" de S. Plantearemos dos "vecindades" posibles para las soluciones.

- **Vecindad 1:** Una solución $S' \in N(S)$ si y sólo si puede obtenerse intercambiando tres nodos de S por dos nodos que no pertenecía a S. Es decir, para $u, v, w \in S$, y x, y nodos del grafo original tal que $x \notin S$ y $y \notin S$, definimos $S' = S - \{u, v, w\} + \{x, y\}$ y decimos que S' es "vecina" de S si y sólo si S' es un conjunto independiente maximal del grafo original. La Figura 21 es un ejemplo de solución "vecina" conisderando a la Figura 20 como solución inicial para el grafo de la Figura 19. En este caso, se han quitado los nodos 1, 8 y 11 para agregar los nodos 3 y 5.
- **Vecindad 2:** Una solución $S' \in N(S)$ si y sólo si puede obtenerse agregando a S un nodo del grafo original que no pertenezca a S, y quitando todos los nodos de S adyacentes a este nuevo nodo. Es decir, para v un nodo del grafo original tal que $v \notin S$, y $A \subseteq S$ tal que para todo $w \in S$, si w es adyacente a v entonces $w \in A$, definimos $S' = S - A + \{v\}$ y decimos que S' es "vecina" de S si y sólo si S' es un conjunto independiente maximal del grafo original. La Figura 22 es un ejemplo de solución "vecina" conisderando a la Figura 20 como solución inicial para el grafo de la Figura 19. En este caso, se ha agregado el nodo 2 y se han quitado los nodos 1, 6 y 7.

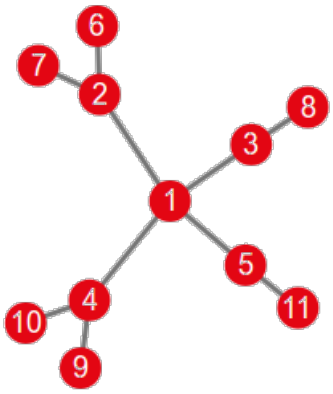

Figura 19: Ejemplo de grafo

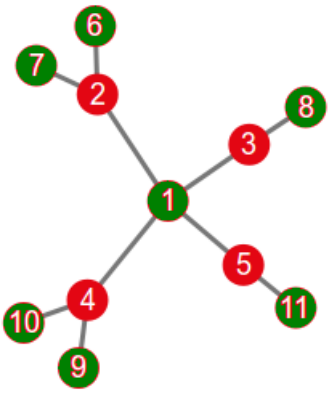

Figura 20: Posible solución inicial

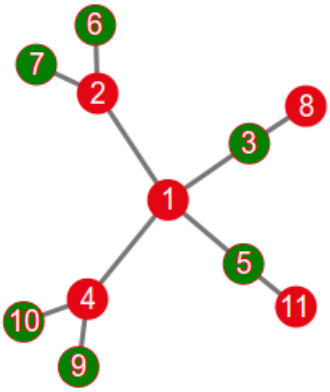

Figura 21: Solución vecina según Vecindad 1

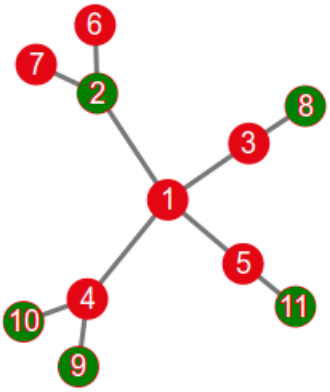

Figura 22: Solución vecina según Vecindad 2

8.2. Análisis de complejidad de una iteración.

Para analizar la complejidad del algoritmo en este caso, tendremos que hacer una división en dos casos, con mejorador1 y con mejorador2. Debemos señalar que conseguir la solución inicial no formará parte del análisis de complejidad dado que es un factor externo, por no estar incluido dentro de una iteración del algoritmo.

En el caso de hacer uso de mejorador1, el procedimiento a realizar en una iteración consiste en obtener un grupo de 3 nodos (esto se hace con 3 iteraciones distintas sobre los elementos de la solución, siendo $\mathcal{O}(n)$ cada iteración y llegando a un costo posible de $\mathcal{O}(n^3)$). Por cada posible combinación de estos 3 nodos, se modifican ciertas variables internas para simular la eliminación de estos elementos (toma $3\mathcal{O}(n)$ ya que implica recorrer los vecinos de estos 3 nodos) y se buscan 2 candidatos entre sus vecinos (uniendo los conjuntos de vecinos de los 3 nodos y eliminando los repetidos), mediante 2 iteraciones anidadas sobre el conjunto de vecinos y verificando si resultan viables ($\mathcal{O}(1)$ la verificación y $\mathcal{O}(n^2)$ encontrar los 2 candidatos, en caso de que los haya). En caso de que se cumplan las condiciones para modificar la solución actual, termina la iteración, de lo contrario, se deshacen las modificaciones realizadas ($3\mathcal{O}(n)$) y termina la iteración.

Con esta información, y siendo $T(n)$ la complejidad de nuestro algoritmo, tenemos:

$$T(n) = \mathcal{O}(n^3) * \mathcal{O}(n^2) + 3\mathcal{O}(n) + 3\mathcal{O}(n)$$
$$T(n) = \mathcal{O}(n^5) + 6\mathcal{O}(n)$$
$$T(n) = \mathcal{O}(n^5)$$

En el caso de utilizar mejorador2, el procedimiento es distinto. Partimos buscando nodos que se

```
CIDM_BUSQUEDA(int mej)
1  cidm_sol ← lista de nodos de una solución inicial
2  res ← |cidm_sol|
3  while haya mejoras y la última solución tenga más de un nodo
4      if mej == 1
5          MEJORADOR1(cidm_sol, res)
6      else MEJORADOR2(cidm_sol, res)
```

Figura 23: Heurística de búsqueda local para CIDM

```
MEJORADOR1(lista_nodos cidm_sol, int res)
 1  for cada grupo de 3 de nodos en cidm_sol
 2      "sacar" los 3 nodos
 3      for cada par de vecinos n1, n2 de estos nodos
 4          if n1 y n2 quedaron "libres" y no son adyacentes
 5              "agregar" n1 y n2
 6              if se forma una solución válida
 7                  salir del ciclo
 8      if se encontró una solución mejor
 9          actualizar cidm_sol
10          actualizar res
11          return
12  return
```

Figura 24: Pseudocódigo de la mejora 1

relacionen con al menos dos nodos de la solución inicial (preguntar por cada nodo nos toma $\mathcal{O}(n)$). Por cada nodo que cumpla, se actualizan variables relacionadas con ese nodo para simular su incorporación a la solución (para esto se requiere recorrer los vecinos de ese nodo con costo $\mathcal{O}(n)$). Luego revisamos si incorporando el nodo previamente seleccionado llegamos a una solución válida ($\mathcal{O}(n)$), de ser así, quitamos los nodos que se relacionan con éste, modificamos variables para seguir teniendo en cuenta el cambio realizado ($\mathcal{O}(n)$) y terminamos la iteración. En caso contrario, terminamos la iteración (al no modificar las variables del otro caso, estas se verán reseteadas en la siguiente iteración).

En este caso, siendo $T(n)$ la complejidad de nuestro algoritmo, tenemos:

$$T(n) = \mathcal{O}(n) * \mathcal{O}(n) + \mathcal{O}(n) + \mathcal{O}(n)$$
$$T(n) = \mathcal{O}(n^2) + 2\mathcal{O}(n)$$
$$T(n) = \mathcal{O}(n^2)$$

```
MEJORADOR2(lista_nodos cidm_sol, int res)
 1  for cada nodo n
 2      if n se conecta con al menos dos nodos de cidm_sol
 3          for cada nodo de cidm_sol que se conecta a n
 4              "sacar" el nodo
 5              if se forma una solución válida
 6                  salir del ciclo
 7          if se encontró una solución mejor
 8              actualizar cidm_sol
 9              actualizar res
10              return
11  return
```

Figura 25: Pseudocódigo de la mejora 2

9. Metaheurística de GRASP

9.1. Desarrollo de la idea.

GRASP (Greedy Randomized Adaptative Search Procedure) es una combinación entre una heurística golosa "aleatorizada" y un procedimiento de búsqueda local. La idea es la siguiente:

```
1  while no se alcance el criterio de terminación
2      Obtener una solución inicial mediante una heurística golosa aleatorizada.
3      Mejorar la solución mediante búsqueda local.
4      Recordar la mejor solución obtenida hasta el momento.
```

Como criterios de terminación, analizaremos dos casos particulares:

- Se realizaron k iteraciones.
- Se realizaron k iteraciones sin encontrar una solución mejor.

En cuanto a la heurística golosa aleatorizada, hemos continuado con la idea del algoritmo goloso descrito anteriormente, con la variación de que, en cada paso, se genera una Lista Restricta de Candidatos (RCL) y se elige aleatoriamente un candidato de esa lista. Analizaremos dos posibles maneras de construir dicha RCL:

- Hallar al mejor candidato, es decir, el nodo que hemos definido como óptimo, y colocar en la RCL los nodos candidatos cuya cantidad de vecinos "libres" sea no menor a un cierto porcentaje α de la cantidad de vecinos "libres" del mejor candidato.
- Hallar al mejor candidato y colocar en la RCL los β mejores candidatos, es decir, los β nodos que más nodos "libres" cubran.