Algoritmos y Estructuras de Datos III

Primer Cuatrimestre de 2009

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Practico N°3
Reentrega

Grupo 10

Integrante	LU	Correo electrónico
Mancuso Emiliano	597/07	emiliano.mancuso@gmail.com
Villaverde Bacskay Romina	740/07	romina.villaverde@gmail.com
Mataloni Alejandro	706/07	amataloni@gmail.com
Blandini Juan	509/07	juanboca@gmail.com

Reservado para la catedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

Conjunto Independiente de Peso Máximo

Sea $G = (V, E)$ un grafo. Cada vértice $v \in V(G)$ tiene asociado un peso positivo $p(v)$. Un subconjunto de vértices $V' \subseteq V(G)$ es un conjunto independiente si los vértices no son adyacentes 2 a 2. El peso de un conjunto independiente, $p(V')$, se define como la suma de los pesos de cada vértice de V', es decir, $p(V') = \sum_{v \in V'} p(v)$.

El problema del CONJUNTO INDEPENDIENTE DE PESO MÁXIMO (CIPM) consiste en encontrar el subconjunto de vértices independiente, cuyo peso sea máximo.

1. Ejemplos de la Vida Real

- Describir situaciones de la vida real que puedan modelarse utilizando CIMP.

En un área alejada de Capital Federal, se encuentran ubicados muchos restaurantes de paso que pertenecen al señor González.

Pero como en este último tiempo se le está complicando administrar los restaurantes, decide cerrar algunos de ellos. Decide mantener abiertos los locales que mejor ganancia dejen y se encuentren a más de 500 metros de cualquiera de sus otros restaurantes, para no perder clientes.

Esta situación puede modelarse utilizando CIMP, ya que podemos considerar que cada nodo represanta un restaurant, cuya ganancia sería el peso del nodo; y dos nodos son adyacentes si los restaurantes que representan se encuentran a menos de 500 metros de distancia.

Otra situación que podemos modelar utilizando CIMP es la siguiente. En un colegio secundario se realizan distintas actividades: arte, teatro, canto, música, diversos deportes y más, donde cada alumno puede realizar más de una actividad. Las autoridades del colegio quieren elegir un equipo de delegados entre todos los alumnos de manera tal que dentro de una actividad no haya más de un delegado, lo que es análogo a que no hayan dos alumnos que sean delegados y que además realicen la misma actividad.

También se quiere que los alumnos que sean elegidos como delegados tengan el mayor promedio posible.

Para modelar esta situación utilizando CIMP, podemos representar a los alumnos con los nodos, donde su peso representa el promedio del alumno; y dos nodos son adyacentes si los alumnos que éstos representan realizan una actividad juntos.

2. Estructuras Utilizadas

Para representar el grafo utilizamos una lista de adyacencia. Además, contamos con otras estructuras que describiremos a continuación, para facilitar la implementación del algoritmo y las heurísticas.

2.1. Clase `Vertice`

Esta clase la utilizamos para representar un nodo. Sus atributos son:

- `numero`: de tipo `int` que representa el número del nodo.
- `peso`: de tipo `int` que representa el peso del nodo.
- `vecinos`: `ArrayList<Vertice>`. Esta lista contiene todos los vecinos del nodo ordenados por su número.
- `componenteConexa`: de tipo `int` que representa el número de componente conexa a la que pertenece.

2.2. Clase `Solución`

Con esta clase representamos tanto las soluciones parciales que se van generando durante la ejecución del algoritmo y las heurísticas, y también la solución final, de cada componente conexa y del grafo. Sus atributos son:

- `conjunto`: `ArrayList<Vertice>`. Representa al conjunto de vértices que forman parte de la solución, por lo tanto, no son adyacentes dos a dos.
- `posibles`: `ArrayList<Vertice>`. Representa a los posibles vértices para agregar al conjunto, éstos son aquéllos que no tienen ningún vecino que forme parte de ésta. Esta lista la actualizamos cada vez que agregamos y sacamos un nodo de la solución parcial.

- `adyacencias`: `int[]`. Contador de veces que cada nodo es adyacente a alguno de la solución, es decir, la posición i del array indica cuántos vecinos tiene el vértice i en la solución parcial. Este array los actualizamos cada vez que agregamos un nodo a la solución y cada vez que eliminamos uno para encontrar otra solución mejor.
- `peso`: de tipo `int` que representa el peso total del conjunto de nodos que forman parte de la solución.
- `sacarDeSolucion`: este método toma como parámetro un vértice para sacar de la solución. Lo elimina del conjunto de vértices, actualiza el peso de la solución y actualiza el arreglo de adyacencias indicando que los vecinos de este vértice tienen un adyacente menos en la solución, y si para cada uno de estos vértices, no quedó ningún vecino en la solución, lo agrego a la lista `posibles`, para que sea considerado en las próximas iteraciones.
- `agregarASolucion`: este método toma como parámetro un vértice para agregar a la solución. Agrega el vértice al conjunto, actualiza el peso de la solución y actualiza el arreglo `adyacencias` incrementando en 1 la cantidad de adyacentes que tiene cada uno de sus vecinos en la solución.

2.3. Inicialización de la Estructura de Datos

- Como la primera línea del archivo representa la cantidad de nodos del grafo, creamos los vértices que necesitamos.
- En cada línea que leemos tenemos la información del nodo. Por lo tanto, cada vez que leemos una línea, asignamos los atributos del vértice: su número, su peso y sus vecinos.
- Llenamos la lista *vertices* que contiene los vértices del grafo. Sólo para la heurística constructiva la ordenamos de forma descendente según su peso utilizando *QuickSort*, ya que en cada iteración agregamos el vértice de mayor peso, siempre que sea posible.

2.4. Aclaración sobre el cálculo de las complejidades

Para calcular la complejidad de los algoritmos, usamos el modelo uniforme, puesto que podemos considerar los pesos de los nodos acotados por un cierto número (el peso máximo), y por lo tanto las operaciones básicas, como agregar un nodo a una solución parcial, sumar y restar un entero que representa al peso, etc, las consideramos como constantes; es decir, la complejidad del algoritmo no depende de los pesos que tengan los nodos, sino del tamaño del grafo (la cantidad de nodos y aristas).

3. Algoritmo Exacto

Desarrollar e implementar un algortimo exacto para CIPM.

3.1. Introducción

Como teníamos que desarrollar un algoritmo exacto para resolver el problema, sí o sí tenemos que evaluar todos los casos que potencialmente pueden ser una solución y quedarnos con el conjunto de nodos, tales que no sean adyacentes dos a dos y además, que el peso sea máximo. Como estamos buscando obtener un conjunto independiente de peso máximo, separamos el grafo en componentes conexas utilizando BFS, aplicamos el algoritmo exacto para cada una de ellas, y luego unimos las soluciones de cada una. Esto lo podemos hacer ya que entre cada componente conexa los nodos no se comunican entre sí; por lo tanto el conjunto que nos queda al unir la solución de cada una de las componentes conexas es independiente.

Para cada componente conexa debemos calcular el subconjunto independiente de peso máximo. Para ello debemos evaluar todos los posibles subconjuntos que puedan llegar a formar parte de la solución del grafo de manera ordenada, y quedarnos con el de mayor peso. Por esto es que decidimos implementar un algoritmo que utilice la técnica de *backtracking* para resolver el problema.

3.2. Descripción y Correctitud de los Algoritmos

Para resolver este problema utilizaremos las siguientes estructuras:

- La lista de los vértices *vertices*, en la cual están los vértices de la componente conexa;
- Un array de enteros *adyacencia*, en el cual en la posición i me va a indicar cuántos vértices de la solución son adyacentes al vértice v_i;
- Una pila *indices*, que en cada iteración, se apila la posición del nodo que se acaba de agregar a la solución;
- Un entero *peso*, que indica el peso total del subconjunto de nodos de la solución;
- Una lista *solucionComponenteConexa* donde se guarda la mejor solución de esa componente conexa;
- Una lista *solucionParcial* que iremos generando en cada iteración; y si, una vez que no quedan más vértices que se puedan agregar a la solucion parcial, ésta supera a la solución de la componente conexa, la reemplazamos por la que acabamos de generar.

Una vez que tenemos todas las estructuras cargadas, separamos el grafo en componentes conexas, y ejecutamos nuestro algoritmo. Para cada componente conexa, tomamos el primer vértice de la lista y lo agregamos a la solución, a su vez, por cada uno de sus vecinos, incrementamos en 1 su valor correspondiente en el arreglo `adyacencias`, para indicar que no se pueden agregar, ya que si agrego alguno de ellos, la solución parcial deja de ser un conjunto independiente. Luego, seguimos recorriendo la lista hasta la última posición, para agregar los nodos que no tengan ningún vecino en la solución, de la misma manera que describimos anteriormente. Una vez que no tenemos más nodos en la lista `vertices` que puedan ser agregados, la guardamos como `solucionComponenteConexa` para compararla con las demás soluciones que generemos, esto lo podemos hacer porque es la primer solución que obtuvimos.

Ahora tenemos que seguir evaluando distintos subconjuntos independientes que puedan superar el peso de la mejor solución que tenemos hasta el momento. Aquí es donde utilizamos la pila `indices`: cada vez que agregamos un nodo a la solución parcial, guardamos en la pila la posición que éste ocupa en el arreglo `vertices`. Entonces, si la posición del último vértice que agregamos era i, para generar nuevos subconjuntos eliminamos este nodo, actualizamos el arreglo `adyacencias`, y volvemos a recorrer `vertices` desde la posición $i+1$ hasta el final, agregando los nodos que no tengan vecinos en la solución. Cuando nos volvemos a quedar sin nodos disponibles para agregar, comparamos la solución parcial que obtuvimos recién, con la de la componente conexa (que es la mejor de todas las anteriores), y si es mejor, la reemplazamos y nos quedamos con la nueva.

Cada vez que, al eliminar repetitivamente los nodos que agregamos, sólo nos queda un nodo v en el conjunto de la solución parcial, significa que ya evaluamos todos los subconjuntos que contienen a v, entonces lo eliminamos de la solución y armamos nuevos subconjuntos que contengan al vértice que le sigue a v pero no lo contienen. El algoritmo sigue iterando hasta evaluar todos los posibles subconjuntos que potencialmente podrían mejorar la mejor solución obtenida hasta el momento. Y una vez que conseguimos la mejor solución, la unimos con la solución global, que contiene las soluciones de las otras componentes conexas evaluadas anteriormente.

Para reducir la cantidad de casos a considerar, sin perder una solución, implementamos una poda en nuestro algoritmo de *backtracking*. Esta poda consiste en calcular en cada iteración el peso total de todos los nodos disponibles que quedan por evaluar, y si éste sumado al peso de la solución parcial que obtuvimos hasta el momento, no supera al de la componente conexa, no tiene sentido seguir evaluando, ya que por esa rama nunca conseguiremos una solución mejor. En este caso, procedemos de la misma forma que cuando no podíamos agregar más vértices al subconjunto que tenemos armado, eliminamos el último que agregamos y evaluamos los vértices que le siguen a éste.

Veamos ahora por qué, en cada componente conexa, generamos siempre un conjunto independiente y luego veamos que es de peso máximo. Cada vez que intentamos agregar un nodo a la solución, vemos si éste tiene algún vecino que ya se encuentre en ella. De no se así, lo agregamos y actualizamos a todos sus vecinos, es por esto que nunca vamos a tener un nodo que no se pueda agregar como disponible. Por lo tanto, nunca tenemos dos nodos adyacentes en el conjunto de la solución, por ende, este conjunto es siempre un conjunto independiente. En cuanto al peso del conjunto, nosotros obtenemos todos los subconjuntos independientes distintos que potencialmente puedan formar parte de la solución, no eliminamos ni agregamos soluciones, y cada vez que conseguimos una nueva solución, comparamos su peso con el de la mejor obtenida hasta el momento, y si es mejor, tomamos a esta nueva solución como la mejor obtenida. Por este motivo, no puede existir un conjunto independiente que tenga mayor peso que el de la solución de la componente conexa.

Tenemos que asegurar también que la solución que devolvemos en cada componente conexa es un conjunto maximal. Sabemos que para obtener la solución de la componente conexa evaluamos todos los subconjuntos que potencialmente pueden llegar a mejorar a la solución que obtuvimos hasta el momeneto, sin obviar ni calcular más de una vez ninguno de ellos. Obtenemos una nueva solución cuando todos los vértices con los que disponemos no están disponibles para agregar, esto significa que, para cada nodo, o pertenece a la solución o tiene al menos un vecino en ella; por lo tanto, al agregar cualquiera de los que no fueron incluidos, la solución parcial de la componente conexa deja de ser un conjunto independiente. Esto ocurre para cada solución que obtenemos, por lo tanto, la solución final de la componente conexa también es un conjunto maximal. Para devolver la solución del grafo, unimos la solución de cada componente conexa, que cada una de ellas es un conjunto independiente maximal, por lo tanto, la solución final también lo es, ya que todos los vértices del grafo que no fueron incluidos, tienen al menos un vecino en ella y si agrego al menos uno de ellos, nuestra solución deja de ser un conjunto indepentiente.

¿Por qué dividimos el grafo en componentes conexas y no alteramos la solución del problema? Como estamos buscando un conjunto independiente, podemos buscar soluciones parciales en cada componente conexa, quedarnos con la mejor de cada una de ellas, y luego generar una solución global con la mejor solución de cada componente conexa, ya que, como sabemos que la solución de cada componente conexa es un conjunto independiente, la unión de la solución de cada componente conexa también va a ser un conjunto independiente, ya que no existen dos vértices en el grafo que pertenezcan a componentes conexas distintas y sean adyacentes. En cuanto al peso de la solución, como en cada componente conexa no existe otro conjunto independiente de peso mayor que el de la solución, tampoco puede haber una solución global mejor, ya que, si existiera, habría una solución mejor para al menos una componente conexa, lo cual no puede ocurrir.

3.3. Pseudocódigo

algoritmoExacto($Grafo\ G$) → Solucion. Dado un grafo, genera un conjunto independiente de peso máximo. Como éste es el algoritmo exacto, no existe otra solución que sea un conjunto independiente y cuyo peso sea mayor que el que genera este algoritmo.

```
1:  algoritmoExacto(Grafo G) → Solucion {
2:      paraTodo (ComponenteConexa C de G) hacer
3:          mientras (vertices no es vacia) hacer
4:              agrego el primer elemento de vertices, v a la solucion parcial y lo borro de la lista
5:              genero todos los subconjuntos independientes que contienen a v, aplicando la poda antes descrita
                  y actualizo la solucionFinal si alguno tiene peso mayor a la solucion generada hasta el momento
6:          fin mientras
7:          solucionGlobal ← solucionGlobal ∪ solucionFinal
8:      fin paraTodo
9:      devolver solucionGlobal
10: }
```

3.4. Cálculo de Complejidad

Sea n_c la cantidad de nodos en un grafo conexo, y m_c la cantidad de aristas del mismo.

Primero calculamos la complejidad para un grafo conexo.

- Probar todos los subconjuntos independientes $O(2^{n_c})$
- Para cada conjunto:
  - Aplicamos la poda. Recorremos los vertices posibles a agregar, desde cada posicion hasta el final. $O(n_c^2)$
  - Recorremos la vecindad de cada vertice para agregar las limitaciones de sus adyacentes. $O(d(v))$

$$2^{n_c} \times (n_c^2 + \sum_{i=1}^{n} d(v_i)) \leq 2^{n_c} \times (n_c^2 + 2m_c)$$

$$\Rightarrow \quad O(2^{n_c} \times (n_c^2 + 2m_c)) \quad \Rightarrow \quad O(2^{n_c} \times (n_c^2 + m_c))$$

Ahora, si un grafo contiene ccc componentes conexas, n nodos y m aristas, el calculo completo es:

- Separar en componentes conexas por BFS. $O(n + m)$
- Por cada componente conexa, analizarla como un grafo conexo antes descrito. $O(2^{n_c} \times (n_c^2 + m_c))$

$$n + m + \sum_{i=1}^{ccc} (2^{n_{ci}} \times (n_{ci}^2 + m_{ci}))$$

Para simplificar, decimos que el algoritmo parte de grafos conexos, entonces $ccc = 1$, es decir, $n_c = n$ y $m_c = m$.

$$n + m + \sum_{i=1}^{ccc} (2^{n_{ci}} \times (n_{ci}^2 + m_{ci})) \quad \Rightarrow \quad n + m + \sum_{i=1}^{1} (2^{n_i} \times (n_{ci}^2 + m_{ci})) \quad \Rightarrow \quad n + m + 2^n \times (n^2 + m) \quad \Rightarrow$$

$$O(2^n \times (n^2 + m) + n + m) \quad \in \quad O(2^n \times (n^2 + m))$$

Cálculo de Complejidad en función de la entrada

$$T = log\ n + \sum_{i=1}^{n} log\ p + \sum_{j=1}^{d(v_i)} log\ v_j \ \geq \ log\ n + \sum_{i=1}^{n} (log\ p + d(v_i)) \ \geq$$

$$log\ n + \sum_{i=1}^{n} 1 + \sum_{i=1}^{n} d(v_i) \ \geq \ log\ n + n + 2m$$

$$\Rightarrow \ T \ \geq \ n \ \wedge \ T \ \geq \ 2m$$

Como el algoritmo es $O(2^n \times (n^2 + 2m))$ y $T \ \geq \ n \ \wedge \ T \ \geq \ 2m \ \Rightarrow \ O(2^T \times (T^2 + T)) \ \in \ O(2^T \times T^2)$

$$\Rightarrow \text{ el algoritmo es } O(2^T \times T^2)$$

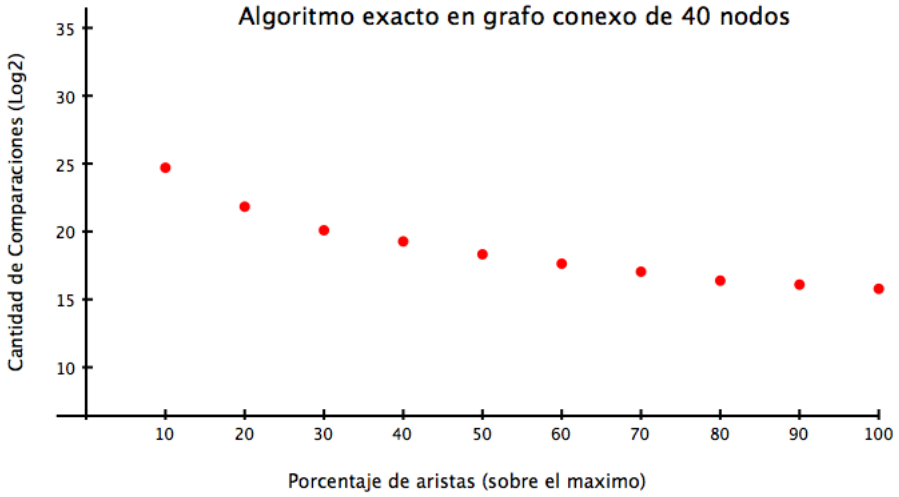
3.5. Casos de Prueba y Gráficos

Para realizar los gráficos introducimos una variable entera que cuenta la cantidad de comparaciones que realizan los algoritmos.

En el siguiente gráfico queremos mostrar que pasa con un mismo grafo, si le cambiamos la cantidad de aristas. Lo que hicimos fue correr el algoritmo con un grafo poco denso ($m = n - 1$, y conexo) y luego ir agregando aristas hasta completar el grafo.

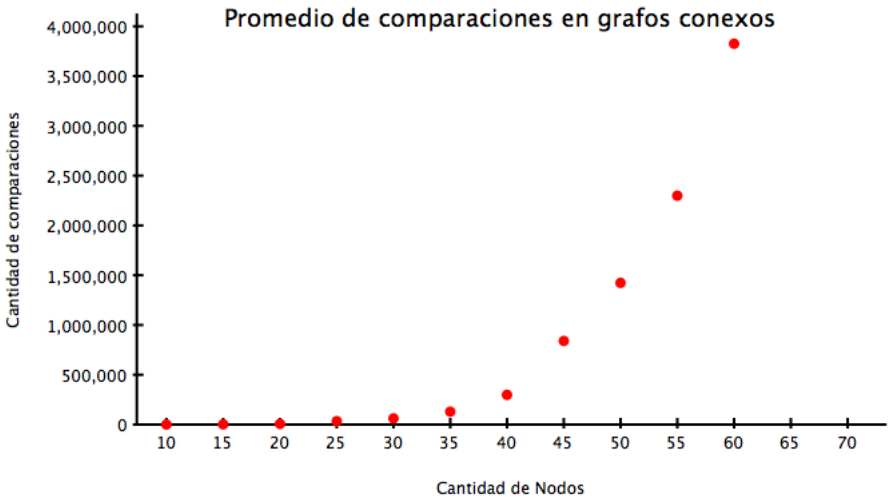
Para realizar este gráfico, corrimos el algoritmo con grafos de 40 nodos, donde el primero de ellos tiene sólo un 10 % de las aristas posibles, el siguiente un 10 % más que el anterior y así hasta completarlo, siempre manteniendo las aristas anteriores, la cantidad de nodos y los pesos de los mismos. El porcentaje siempre es sobre la cantidad de aristas totales que puede tener el grafo, o sea, si es completo.

Realizamos la prueba para varias instancias de grafos con las caracteristicas especificadas y sacamos un promedio de la cantidad de comparaciones. Decidimos hacerlo con grafos de 40 nodos ya que podemos obtener las soluciones en poco tiempo con esa cantidad de nodos, y que con estos resultados podríamos mostrar la hipótesis.



Lo que podemos observar, como era de esperar, que cuantas menos aristas tenga el grafo más comparaciones y peor se va a comportar el algoritmo. Esto se debe a que en un grafo muy denso los posibles subconjuntos independientes son menos que en el caso en el que tenga pocas aristas. Por esta razon en un grafo muy denso no se recorren muchas vecindades, lo que disminuye la cantidad de comparaciones.

Por otro lado, mostramos a continuación el comportamiento del algoritmo, pero modificando la cantidad de nodos y su relación con las comparaciones que realiza el algoritmo exacto.
A medida que incrementamos la cantidad de nodos en los grafos, esperamos que la cantidad de comparaciones que realiza se incremente también.



Efectivamente, podemos ver que la cantidad de comparaciones que realiza el algoritmo según la cantidad de nodos, crece exponencialmente, como la complejidad del algoritmo.

Lo que realmente vemos en el gráfico es un promedio entre los valores obtenidos para los grafos evaluados.
Esta prueba se llevo a cabo sobre 50 grafos, de densidad variable entre el 30 % y el 60 % de las aristas y con pesos bien distribuidos. Los grafos parten con 10 nodos, y se le incrementan de a 5 llegando hasta grafos de 120 nodos. Estos ultimos no se muestran en el gráfico pues la cantidad de comparaciones era tan grande que hacía despreciable la diferencia entre grafos con menos nodos.

4. Heurística Constructiva

- Desarrollar e implementar una heurística constructiva para CIMP.

4.1. Descripción de la Heurística

Una vez que leemos el archivo de entrada y tenemos todas las estructuras cargadas -la lista de adyacencia, los vértices con sus atributos y la lista con los vértices para recorrer, ordenados por peso de mayor a menor- comenzamos la ejecución de nuestra heurística.

Para desarrollar la heurística constructiva implementamos un algoritmo goloso, que partiendo de una lista de vértices ordenada descendentemente por su peso, arma la solución eligiendo, en cada iteración, el primer vértice que tenga disponible.

Para generar la solución, creamos un iterador para la lista de los vértices del grafo, tomamos el primero y lo agregamos a la solución, esto lo podemos hacer ya que al ser el primer vértice de la lista, no tiene ningún vecino que ya esté agregado. Al agregarlo, marcamos este vértice como utilizado, e incrementamos en el contador de adyacencias a todos sus vecinos para que no se puedan agregar en las próximas iteraciones.

En la próxima iteración agregamos el siguiente vértice que no esté marcado, es decir, que no tenga ningún vecino en la solución e incrementamos a los nodos adyacentes indicando que tienen un vecino más que forma parte de la solución.

Continuamos iterando hasta que hayamos recorrido todos los vértices que forman parte del grafo, agregando a la solución aquéllos que no tengan vecinos ya agregados.

Veamos que el conjunto de vértices que devuelve la heurística constructiva es un conjunto independiente. Nosotros tenemos una lista de vértices ordenada descendentemente por peso, e inicialmente agregamos a la solución el primer vértice de la lista, y marcamos a sus vecinos para que no puedan ser agregados luego. En las próximas iteraciones, la heurística va a recorrer los siguientes vértices hasta encontrar alguno que no esté marcado, es decir, que no tenga ningún vecino que ya forme parte de la solución; y al agregarlo, incrementa en 1 el valor correspondiente de cada uno de sus adyacentes en el array `adyacencias`, para indicar que tienen un vecino más en la solución. Seguimos ejecutando la heurística de esta manera, hasta que hayamos recorrido todos los vértices del grafo. Como la condición para agregar un vértice a la solución es que no tenga ningún vecino en ella, y al agregarlo marcamos a todos sus vecinos, sin obviar a ninguno, nunca vamos a tener dos nodos adyacentes en la solución, por lo tanto, el conjunto de vértices que devuelve la heurística constructiva es independiente.

4.2. Pseudocódigo

heuristicaConstructiva($Grafo\ G$) → Solucion. Genera una solución agregando nodos de mayor a menor (por peso) sólo si el nodo por agregar no es adyacente a ninguno que ya forme parte de la solución.

```
1: heuristicaConstructiva(Grafo G) {
2:     paraTodo (Vertice vertice ∈ vertices) hacer
3:         if (vertice no tiene vecinos en solucion)
4:             agregarASolucion(vertice)
5:         end if
6:     fin
7:     devolver solucion
8: }
```

agregarASolucion ($Verticev$). Agrega el vértice v a la solución que estamos generando, además, actualiza el peso de la misma, se marca ese vértice como utilizado, y se incrementa la cantidad de vecinos en la solución de todos sus adyacentes.

```
1: agregarASolucion(Vertice v) {
2:     agregar v al conjunto de vertices de la solucion
3:     sumar peso de v a la solucion
4:     marcar a v como usado
5:     deshabilitar vecinos de v para la solucion
6: }
```

4.3. Cálculo de Complejidad

- Ordenar los nodos por peso y de forma descendente con un algoritmo Quicksort, es del orden de $O(n^2)$
- Verificar si se puede agregar un nodo a la solución toma $O(1)$ pues es verificar que el contador de adyacencias de ese nodo este en 0.
- Agregar un nodo a la solución pertenece al orden $O(d(v_i))$ ya que tiene que incrementar el contador de adyacencias de sus vecinos.
- Para cada nodo que se puede agregar a la solución, lo agregamos. $O(d(v_i))$

$$n^2 + \sum_{i=1}^{n} d(v_i) \leq n^2 + 2m \quad \Rightarrow \quad O(n^2 + 2m) \quad \in \quad O(n^2 + m)$$

Cálculo de Complejidad en función de la entrada

$$T = log\ n + \sum_{i=1}^{n} log\ p + \sum_{j=1}^{d(v_i)} log\ v_j \ \geq \ log\ n + \sum_{i=1}^{n} (log\ p + d(v_i)) \ \geq$$

$$log\ n + \sum_{i=1}^{n} 1 + \sum_{i=1}^{n} d(v_i) \ \geq \ log\ n + n + 2m$$

$$\Rightarrow \quad T \ \geq \ n \ \wedge \ T \ \geq \ 2m \quad \Rightarrow \quad T \ \geq \ m$$

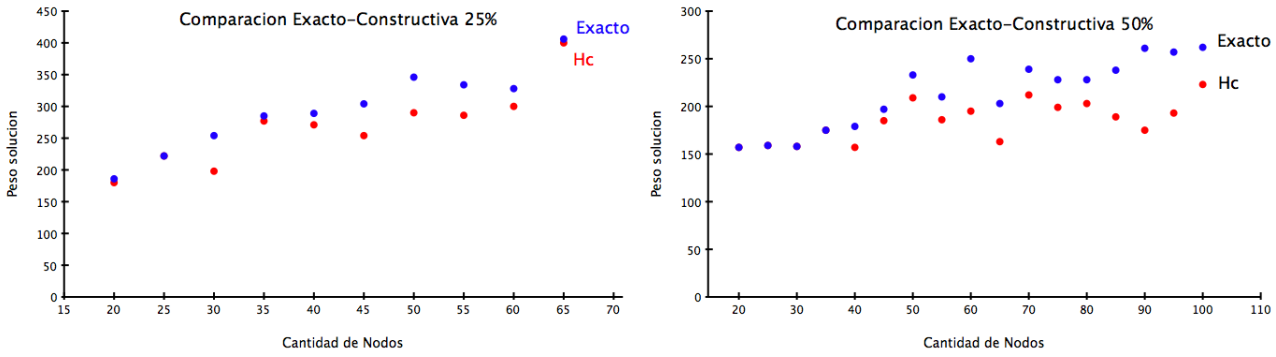
Como el algoritmo es $O(n^2 + m)$ y $T \ \geq \ n \ \wedge \ T \ \geq \ m \quad \Rightarrow$ el algoritmo es $O(T^2 + T) \quad \in \quad O(T^2)$

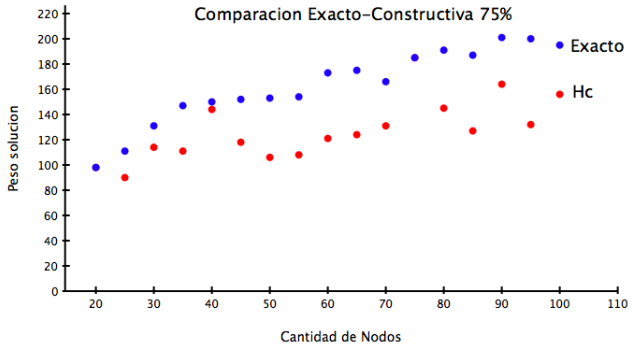
4.4. Casos de Prueba y Gráficos

Debido que la complejidad de la heurística es polinomial y no tiene sentido comparar la velocidad de esta heurística con el algoritmo exacto, decidimos comparar la calidad de los resultados que arroja la heurística frente a los obtenidos por el algoritmo exacto, para ver que tan "bueno" es nuestro algoritmo goloso.

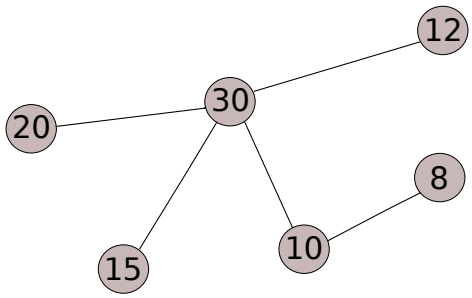
Para realizar estas pruebas generamos grafos al azar para evaluar los comportamientos en grafos generados de manera aleatoria, y ver que sin necesidad que cumplan algún patrón los resultados de la heurística son generalmente buenos. Armamos distintos casos de prueba que describiremos a continuación y compararemos los resultados obtenidos por la heurística con los obtenidos por el algoritmo exacto, además, de esta forma podremos ver qué tan buena es la solución con la que partiremos al ejecutar las heurísticas HBL y GRASP. Para hacer las pruebas, desarrollamos un algoritmo que genera grafos conexos, partiendo de un árbol y agregando aristas hasta la alcanzar la densidad buscada. Los pesos de estos grafos se generan aleatoriamente, acotados superior e inferiormente por un parámetro.

Dividimos los gráficos de los casos de prueba según la densidad del grafo y la amplitud de los pesos. Observamos que cuando el grafo es menos denso, la calidad de la solución es mejor comparada a los otros casos. Esto se debe a que, como los vértices no tienen tantos vecinos, cada vez que agregamos el nodo de mayor peso entre los disponibles, son menos los que anulamos para agregar a la solución en las próximas iteraciones.





Haciendo los casos de prueba, pensamos en si la amplitud de los pesos de los nodos podría afectar la solución generada por la heurística constructiva, ya que si tenemos un grafo en donde los pesos de los nodos no varían mucho, nos podría pasar que agregamos a la solución el nodo disponible de mayor peso cuyos vecinos disponibles tienen peso menor, pero parecido al de éste. Si podemos armar un conjunto independiente con estos nodos, como su peso no es mucho menor al del nodo elegido, el peso del conjunto podría superar notablemente al peso del nodo. Y en un grafo en donde la diferencia entre el peso mínimo y el peso máximo es grande, la posibilidad de que ocurra esto es menor.



La figura anterior ilustra lo que queremos evaluar. La heurística agregaría a la solución el nodo de peso 30, deshabilitando a sus vecinos para que puedan ser agregados luego. Sin embargo, podemos formar un conjunto independiente con estos nodos y su peso sería 57, que es bastante mayor.

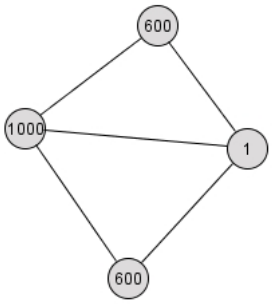
Probamos con varios grafos conexos, y para obtener un resultado que sea válido con respecto a lo que queríamos testear, para cada uno de ellos mantuvimos la cantidad de nodos y la vecindad de cada uno de ellos, modificando sólo el rango del peso de cada nodo. Ejecutamos la heurística constructiva y el algoritmo exacto para cada uno de estos grafos y obtuvimos los resultados que se ven reflejados en la siguiente tabla.

Grafos con Pesos de 10 a 300			Grafos con Pesos de 10 a 30		
Algoritmo Exacto	Heuristica Constructiva	Porcentaje	Algoritmo Exacto	Heuristica Constructiva	Porcentaje
2024	1641	81.08 %	238	165	69.33 %
2040	1711	83.87 %	237	207	87.34 %
2151	1970	91.59 %	241	196	81.33 %
2199	295	13.42 %	208	190	91.35 %
2318	1727	74.5 %	222	217	97.75 %
1964	1933	98.42 %	242	204	84.3 %
2283	1786	78.23 %	243	203	83.54 %
2397	1888	78.77 %	236	167	70.76 %
2243	1888	84.17 %	253	197	77.87 %
2185	1679	76.84 %	228	218	95.61 %
2506	1799	71.79 %	236	188	79.66 %
2153	1673	77.71 %	270	222	82.22 %
1929	1380	71.54 %	225	166	73.78 %
2108	1986	94.21 %	226	162	71.68 %
2441	1955	80.09 %	260	174	66.92 %
2494	1931	77.43 %	247	165	66.8 %
2099	1683	80.18 %	226	209	92.48 %
2326	2016	86.67 %	242	160	66.12 %
2103	1629	77.46 %	242	199	82.23 %
2265	1522	67.2 %	228	157	68.86 %
Promedio		77.26 %	Promedio		79.5 %

Estos grafos están formados por 80 nodos con una densidad del 50 %, cuyos pesos varían como figura en la tabla. En cada línea tenemos dos grafos isomorfos, cuya única diferencia entre sí son los pesos. Podemos ver que en algunos casos el porcentaje del peso obtenido por la heurística constructiva con respecto al del peso de la obtenida por el algoritmo exacto es mayor en los grafos con pesos más grandes que en los grafos con pesos más chicos, pero en otros casos es al revés (los que escribimos en rojo). No sólo que no pudimos encontrar ningún patrón, sino que además al sacar el promedio de los promedios de cada tipo de grafos, la diferencia entre estos dos es despreciable.

Realizamos la misma prueba con distintos grafos, variando las densidades y todos con hasta 100 nodos (para poder ejecutar el algoritmo exacto) y en todos los casos la diferencia era despreciable, como en el caso que presentamos en esta tabla. Por lo tanto, la solución de la heurística no es mejor ni peor dependiendo del intervalo que determina el peso de los nodos.

4.5. Casos en los que el método no proporciona una solución óptima



Como podemos observar en este grafo, la heurística constructiva nos devolvería una solución formada por el nodo con peso 1000; y sin embargo, la solución óptima está formada por los dos nodos de peso 600. Esto ocurre debido a que la heurística agrega a la solución los nodos de mayor peso siempre que no sea adyacente a ninguno de la solución. Esto queda salvado en $GRASP$ y la $busquedaLocal$.

Pero en la práctica pudimos ver que no ocurre frecuentemente, y en Búsqueda Local y GRASP queda salvado. En HBL, en alguna de las combinaciones, el nodo de 1000 sería quitado de la solución, pudiéndose agregar los dos de 600 al buscar el mejor conjunto, y superando así el peso de la solución anterior. Y en GRASP, con el factor aleatorio, puede ser que en alguna iteración se elija al de 1000, pero en otras, si los nodos adyacentes son bastante cercanos en cuanto al peso, puede tomarlos como inicio de la solución. Si no los toma por no entrar dentro del 20 % (como sería este caso), con Búsqueda Local se solucionaría este problema. Esto es un problema que puede presentarse en la Heurística Constructiva, pero que no afecta a la hora de tomarlo como parámetro para Hbl o GRASP, ya que cada uno a su manera lo puede solucionar.

5. Heurística Busqueda Local

- Desarrollar e implementar una heurística de busqueda local para CIMP.

5.1. Descripción de la Heurística

5.1.1. Criterio de Vecindad

Para hacer la búsqueda local primero tuvimos que decidir qué criterio de vecindad tomábamos. En un primer momento pensamos en sacarle a la solución un porcentaje de nodos y luego agregar (como la heurística constructiva) los posibles nodos. Pero luego en la práctica, nos dimos cuenta que esto no aportaba mucho y que la vecindad era demasiado grande.

Impulsados por esto y luego de varias consultas, nos enfocamos en cambiar el criterio de vecindad. Primero, lo que decidimos fue no sacar un porcentaje de nodos, sino sacar un número fijo de la solución. Despues de hacer varias pruebas, concluímos en sacar de la solución un subconjunto formado por 2 nodos, actualizando el conjunto de posibles vértices para agregar a la solución.

Además modificamos la forma en que agregamos los nuevos posibles nodos que se habilitaron al sacar los 2 nodos de la solución. En vez de continuar con la heurística constructiva decidimos encontrar el mejor conjunto (de 1, 2 o 3 elementos) independiente perteneciente a los posibles. La razón por la cual fijamos de esa manera el cardinal del conjunto a agregar, es que luego de varios testeos, fue con el que obteníamos la mejor relación costo/beneficio. Luego sí, con el mismo criterio que la heurística constructiva, le agregamos (si es que hay) los nodos que falten para hacer la solución maximal, cuidando que siga siendo independiente.

Aunque generar todos los conjuntos de 1, 2, o 3 elementos, y quedarnos con el que mejor incrementaba el peso de la solución, nos aumentaba la complejidad teórica, en la práctica nos mejoraba muchísimo la solución obtenida con un aumento despreciable del tiempo de ejecución.

5.1.2. Explicación

Sea S la solución de un grafo conexo que recibe ésta heurística, calculamos las soluciones vecinas con el criterio de vecindad.

En esta etapa teníamos dos opciones:

- $HBL1$: Movernos a la mejor solución vecina de todas las que mejoran, y partir de esta nueva solución.
- $HBL2$: Movernos a la primer solución vecina que mejore y partir de esta nueva solución.

HBL1	
Peso	Comparaciones
1547	2649214
1602	1597778
1646	2994873
1723	3021388
1337	3525897
1690	1482242
1944	1986955
1911	1911242
1771	1844278
1814	1503834

HBL2	
Peso	Comparaciones
1516	2268464
1602	1598084
1640	2817863
1744	2708938
1312	2458041
1690	1482548
1944	1698988
1911	1776960
1771	1543561
1814	1504254

Observando esta tabla, decidimos quedarnos con la opción HBL1, que se queda con la mejor solución vecina de toda la vecindad, a pesar de que ésta realice más comparaciones, el aumento del tiempo de ejecución era despreciable como para descartar esta opción.

Una vez elegido el método, lo que hacemos es, dada una solución, nos movemos a la mejor solución vecina, y volvemos a aplicar búsqueda local con esta nueva solución. Este procedimiento finaliza cuando, de una determinada solución, ninguna vecindad mejora o bien cuando mejoró cierta cantidad de veces.

Este párametro BQL_VECES lo fijamos luego en los casos de prueba.

Veamos que la heurística es correcta, es decir, que la solución que devuelve es un conjunto independiente. Nosotros partimos de una solución generada por la heurística constructiva, que como ya vimos, es un conjunto independiente. En cada iteración, hacemos lo siguiente para cada componente conexa: para cada subconjunto de dos nodos, lo sacamos de la solución que ya teníamos, y marcamos como disponibles los vértices correspondientes (aquéllos que, al haber sacado el subconjunto, no tienen vecinos en la solución). Hasta este punto, seguimos teniendo un conjunto independiente, ya que sólo eliminamos nodos de la solución sin agregar ninguno, lo cual no provoca que el conjunto pueda dejar de ser independiente.

En el siguiente paso, para cada vértice que tengamos disponible, calculamos los subconjuntos independientes de 1, 2 ó 3 nodos que se encuentran disponibles y lo contengan. Nos quedamos con el que sume el mayor peso y lo agregamos a la solución, actualizando para cada nodo fuera de la solución la cantidad de adyacentes que tiene en ella. Como los nodos de este conjunto eran nodos que estaban disponibles, y no son adyacentes entre sí, el conjunto de nodos no deja de ser un conjunto independiente después de agregar los vértices, y por lo tanto, la solución que tenemos hasta ahora es válida.

Puede ocurrir que al haber sacado 2 nodos de la solución y agregar 1, 2 ó 3 vértices nuevos, queden otros vértices que no se encuentren en ella y que estén disponibles para agregar; en este caso, procedemos como en la heurística constructiva, agregando los de mayor peso, siempre y cuando la solución no deje de ser un conjunto independiente. Y como éstos estaban disponibles, no tenían ningún vecino en la solución, por lo tanto, el conjunto de nodos de la solución sigue siendo un conjunto independiente y ahora es maximal.

Repetimos estos pasos para cada componente conexa, y una cantidad fija de veces (BQL_VECES), en donde en cada iteración se genera una nueva solución parcial (si es que mejora la que teníamos antes) y como vimos recién, esta nueva solución también es un conjunto independiente. Luego que la heurística iteró la cantidad necesaria de veces y ha intentado mejorar cada componente conexa, nos devuelve la nueva solución. Cada componente conexa puede haber sido modificada para mejorar la solución que teníamos antes, y el conjunto de vértices de cada una de ellas sigue siendo un conjunto independiente, por lo tanto al unir todas estas componentes conexas para formar la nueva solución, los nodos de ésta también formarán un conjunto indepeniente.

5.2. Pseudocódigo

```
1:  busquedaLocal(Solucion solucionInicial) {
2:      paraCada (ComponenteConexa C de G) hacer BQL_VECES
3:          solucionParcial ← solucionInicial
4:          paraCada (Subconjunto s de 2 elementos)
5:              sacar s de solucionParcial
6:              conj ← mejorConjuntoDePosibles
7:              agregar conj a la solucionParcial
8:              completar la solucionParcial para que sea maximal
9:              if (mejora la solucion)
10:                 mejorSolucion ← solucionParcial
11:             end if
12:         fin paraCada
13:         solucionInicial ← mejorSolucion
14:     fin paraCada
15:     devolver mejor
16: }
```

```
1:  mejorConjuntoDePosibles(solucion) {
2:      paraTodo (nodo disponible para agregar, que no tiene ningun adyacente en la solucion) hacer
3:          conj1 ← mejor conjunto de 1 nodo
4:          conj2 ← mejor conjunto de 2 nodos que no sean adyacentes entre si
5:          conj3 ← mejor conjunto de 3 nodos que no sean adyacentes entre si
6:      fin paraTodo
7:      devolver el mejor conjunto de los 3 (el mas pesado)
8:  }
```

5.3. Cálculo de Complejidad

Sea n_c la cantidad de nodos de un grafo conexo y m_c la cantidad de aristas del mismo. Calculemos la complejidad para un grafo conexo:

- Tomar todos los posibles conjuntos de 2 nodos para quitar de la solución. $O(n_c^2)$
- A cada conjunto formado de 2 nodos para quitar, lo quitamos de la solución, habilitando nodos posibles a agregar. $O(n_c)$
- Tomar el conjunto de mayor peso de los conjuntos posibles de 1, 2 y 3 elementos dentro de los nodos posibles a agregar pertenece al orden de $O(n_c + n_c^2 + n_c^3) \quad \Rightarrow \quad O(n_c^3)$
- Agregar el mejor conjunto seleccionado anteriormente a la solución. $O(d(v)) \leq O(n_c)$
- Completar la solución con los nodos posibles, hasta que sea maximal $O(lon(posibles)) \leq O(n_c)$

$$n_c^2 \times (n_c + n_c^3 + n_c + n_c) \quad \Rightarrow \quad n_c^2 \times (n_c^3 + 3n_c)$$

$$\Rightarrow \quad O(n_c^2) \times O(n_c^3) \quad \Rightarrow \quad O(n_c^5)$$

Ahora, si consideramos que un grafo puede tener *ccc* componentes conexas, el calculo final del algoritmo seria:

- Separar en componentes conexas, utilizando BFS. $O(n + m)$
- A cada componente conexa, aplicar el algoritmo para grafos conexos antes descrito. $O(n_c^5)$

Pero, el peor caso lo encontramos cuando $ccc = 1$, o sea la cantidad de nodos de la única componente conexa es n, pues la cantidad de subconjuntos (2^n) que se pueden formar es mayor, por lo tanto, habría más soluciones que comprobar.

La complejidad es:

$$n + m + \sum_{i=1}^{ccc} n_c^5 \quad \Rightarrow \quad n + m + \sum_{i=1}^{1} n^5 \quad \Rightarrow \quad n + m + n^5 \quad \in \quad O(n + m + n^5) \quad \in \quad O(n^5)$$

Cálculo de Complejidad en función de la entrada

$$T = log\ n + \sum_{i=1}^{n} log\ p + \sum_{j=1}^{d(v_i)} log\ v_j \ \geq \ log\ n + \sum_{i=1}^{n} (log\ p + d(v_i)) \ \geq$$

$$log\ n + \sum_{i=1}^{n} 1 + \sum_{i=1}^{n} d(v_i) \ \geq \ log\ n + n + 2m$$

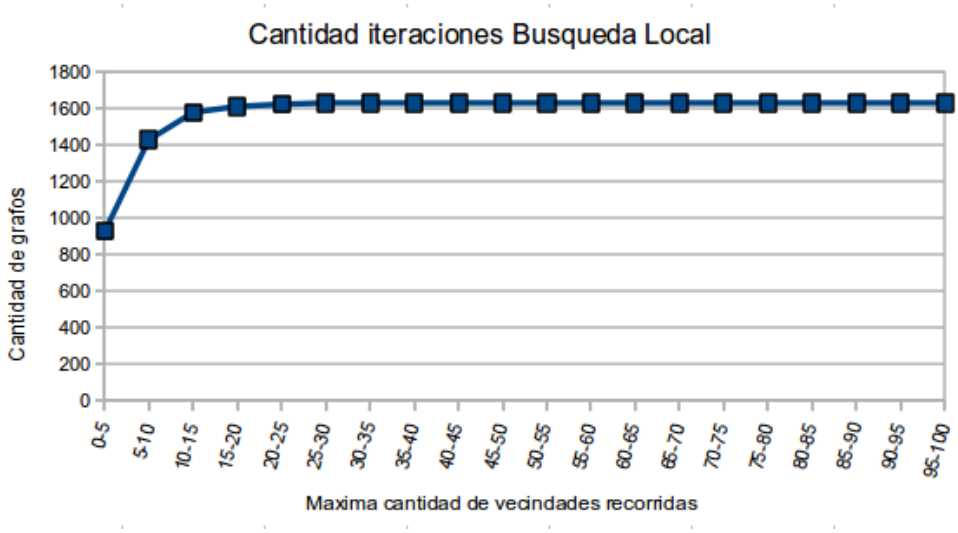
$$\Rightarrow \quad T \ \geq \ n$$

Como el algoritmo es $O(n^5)$ y $T \ \geq \ n \quad \Rightarrow$ el algoritmo es $O(T^5)$

5.4. Casos de Prueba y Gráficos

Para poder fijar la cantidad de vecindades a recorrer, debemos evaluar donde nuestra heurística obtiene mejores resultados. Para esta evaluación, creamos 1630 grafos conexos, entre 100 y 1000 nodos al azar, y con densidad variable de aristas. Una vez que creamos los grafos, para cada uno utilizamos la Heurística de Busqueda Local y guardamos la cantidad de vecindades que recorrió. Forzosamente recorremos 400 vecindades, para asegurarnos que mejora lo más posible.

Llevamos una suma de la cantidad de vecindades que recorrió en cada caso, y las agrupamos en pequeños intervalos, asi podemos observar cuantas vecindades deberíamos recorrer para asegurarnos un buen resultado de parte de la heurística.



Podemos ver que la pendiente de crecimiento de la curva, a medida que se incrementa la cantidad de vecindades recorridas, va disminuyendo hasta llegar a 0.
Esto significa que la busqueda local encontró su vecino máximo antes que llegue a 0 la pendiente. En el gráfico vemos que esto sucede alrededor de las 30 vecindades recorridas, pero como la diferencia en tiempo de ejecución de recorrer 30 vecindades y 50 es despreciable, elegimos como parámetro, recorrer 50 vecindades para tener un espectro más grande y mejorar algun otro grafo que todavía puede mejorar. ($BQL\_VECES = 50$)

6. Metaheurística GRASP

- Desarrollar e implementar una metaheurística GRASP.

6.1. Descripción de la metaheurística

Una metaheurística es una estrategia de alto nivel que utiliza otras heurísticas para alcanzar o mejorar una solución. En particular, una metaheurística GRASP se compone en una heurística constructiva para obtener una solución inicial, - aplicándole un factor de aleatoriedad para no generar siempre la misma solución inicial - y una heurística de búsqueda local para mejorar la solución inicial.

En nuestro caso, aplicamos una heurística constructiva como la antes descrita; sólo que, para ir formando la solución, tomamos aleatoriamente un nodo de un porcentaje de los nodos de mayor peso. Este $PORCENTAJE$ es un parámetro que luego fijaremos al final de los casos de prueba.

Veamos que esta solución es un conjunto independiente. Esta heurística funciona igual que la heurística constructiva, con la diferencia que en lugar de tomar el nodo de peso máximo en cada iteración, tomamos uno al azar de los de mayor peso, o sea los que se encuentran dentro del parámetro *pocentaje*. Una vez que tenemos este nodo, queremos ver si podemos agregarlo a la solución, esto lo podemos hacer sólo si este nodo no tiene ningún vecino que ya se encuentre en ella, de ser así, lo agregamos y actualizamos el estado de sus vecinos, indicando que tienen un adyacente más en la solución. Procedemos de la misma forma, hasta no tener más nodos disponibles para agregar. Cada vez que que agregamos un nodo a la solución verificamos que éste no sea adyacente a ninguno que ya se encuentre en ella, y luego de agregarlo (si es posible) marcamos a todos sus vecinos indicando que tienen un vecino más en la solución. Por lo tanto los nodos que figuran como disponibles son sólo aquéllos que no tienen ningún adyacente en la solución parcial que estamos generando, y son los únicos que podrán ser agregados a la solución en futuras iteraciones. Luego, la solución que generamos es un conjunto independiente de nodos.

Luego de obtener la solución inicial, a ésta le aplicamos la misma búsqueda local que definimos anteriormente y nos quedamos con la nueva solución si es de mejor peso. Como vimos anteriormente, la Heurística de Búsqueda Local puede modificar la solución, pero ésta sigue siendo un conjunto independiente. Este procedimiento lo realizamos $CANTIDAD\_VECES$ veces, ya que como el resultado obtenido tiene un factor de aleatoriedad, no siempre va a ser la mejor solución. Es por esto que la ejecutamos varias veces y nos quedamos con la solución de mayor peso de todas las que obtuvimos.

Nuestra heurística obtiene $CANTIDAD\_VECES$ soluciones, que son conjuntos independientes, y devuelve sólo aquélla de mayor peso. Por lo tanto, la solución obtenida por la Heurística GRASP también es conjunto independiente de vértices.

6.2. Pseudocódigo

metaheuristicaGRASP($Grafo\ G$) → Solucion. Devuelve la mejor solución encontrada, luego de $CANTIDAD\_VECES$ iteraciones.

```
1:  metaheuristicaGRASP(Grafo G) {
2:      repetir CANTIDAD_VECES
3:          solucionParcial ← heuristicaContructivaAleatoria(vertices, PORCENTAJE)
4:          solucionParcial ← busquedaLocal(solucionParcial)
5:          if (es mejor que la solucion global)
6:              solucionGlobal ← solucionParcial
7:          end if
8:      fin
9:      devolver solucionGlobal
10: }
```

heuristicaContructivaAleatoria() → Solucion. Devuelve la mejor solución encontrada, luego de $CANTIDAD\_VECES$ iteraciones.

```
1:  heuristicaContructivaAleatoria(vertices, PORCENTAJE) {
2:      mientras (vertices no este vacío) hacer
3:          v ← tomar uno al azar del PORCENTAJE de los mas grandes
4:          if (v no tiene vecinos en solucion)
5:              agregarASolucion(v)
6:          end if
7:          sacarDeVertices(v)
8:      fin
9:      devolver solucion
10: }
```

6.3. Cálculo de Complejidad

- Generar una solución inicial aleatoria, partiendo de una heurística constructiva. $O(n^2 + m)$
- Aplicar busqueda local a esta solución. $O(n^6)$
- Actualizar la solucion. $O(1)$
- Repetir este proceso $CANTIDAD\_VECES$. $O(k)$

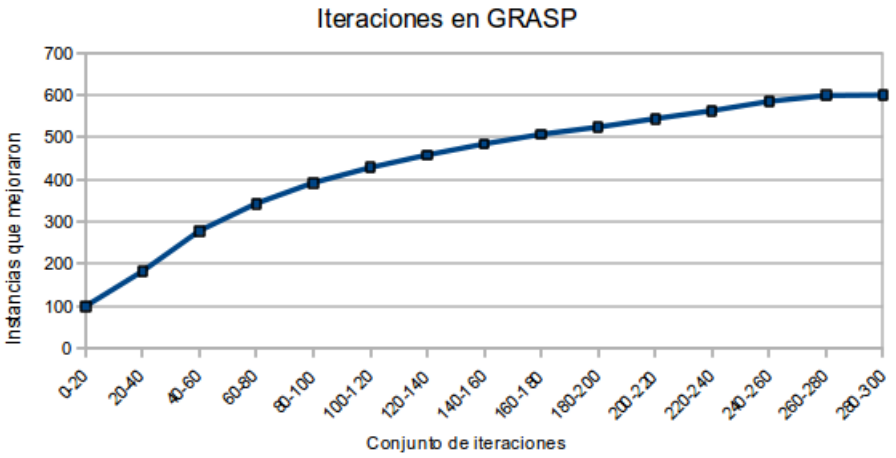
$$k \times (n^2 + m + n^6) \quad \Rightarrow \quad O(k \times (n^2 + m + n^6)) \quad \in \quad k \times O(n^6) \quad \in \quad O(n^6)$$

6.4. Casos de Prueba y Gráficos

Tenemos que decidir dos parámetros para nuestra Heurística GRASP.

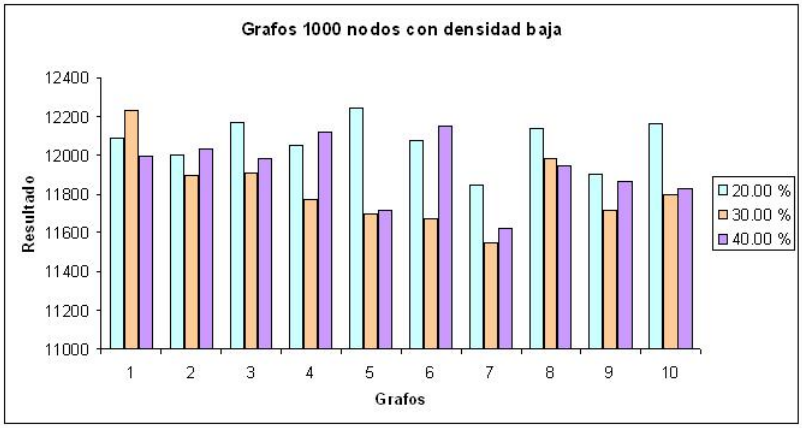
Para determinar la $CANTIDAD\_VECES$ que ejecutamos GRASP, utilizamos el mismo proceso que en Busqueda Local. Acumulamos el número de iteraciones donde el algoritmo alcanzo el máximo y lo volcamos en un gráfico.

A diferencia de la Busqueda Local, GRASP demora mucho más tiempo, entonces redujimos la cantidad de grafos donde testear la heurística. Creamos 600 grafos, con alrededor de 100 y 1000 nodos, pero con una densidad media de aristas.

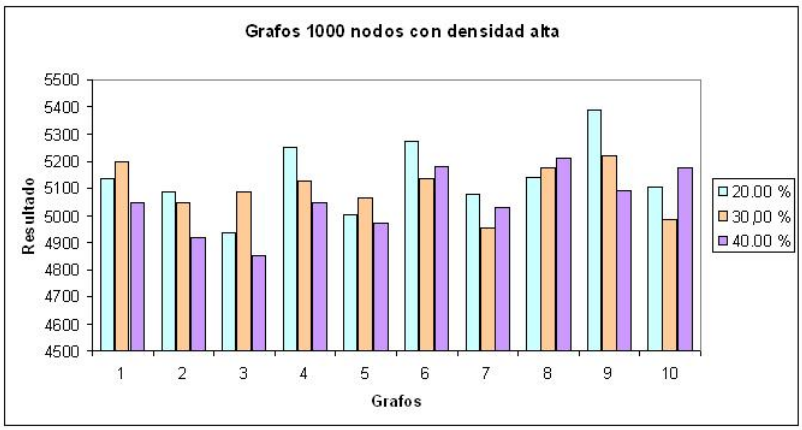


Nuevamente, la pendiente de la curva nos informa donde el algoritmo llega a sus máximos. En este caso, la pendiente no se vuelve 0 tan rapidamente como en la Busqueda Local, pero si podemos ver una gran diferencia alrededor de las 120 iteraciones, que es donde cada vez menos grafos llegan a alcanzar el máximo.
Pese a que la heurística sigue mejorando con más iteraciones, se vuelve un proceso muy costoso con respecto al tiempo de ejecución. Por eso decidimos fijar el parámetro en 150, que es donde la pendiente va decreciendo, y en la mayoría de los casos queda cubierta la mejor solución. ($CANTIDAD\_VECES = 150$)

Otro de los parámetros a tomar en cuenta para este algoritmo es el porcentaje de la lista restringida de candidatos (RCL) que se utiliza en el algoritmo constructivo. Para realizar estas pruebas, en cada gráfico tomamos 10 instancias de grafos conexos de 1000 nodos. Primero generamos los 10 grafos tal que tengan una baja densidad y corrimos la heurística de GRASP para cada uno de ellos.



Para este segundo gráfico generamos nuevamente 10 instancias, pero esta vez cada instancia tenía una densidad mayor al 75 %.



En ambos gráficos podemos ver que, si bien no ocurre en la totalidad de los casos, el resultado mayor se obtiene cuando el porcentaje de la RCL es del 20 %.

Este parámetro está muy ligado al de la cantidad de iteraciones de GRASP, y a la cantidad de nodos del grafo. Dependiendo del tiempo disponible que tengamos para lograr una mejor relación tiempo-benficio, vemos que cuanto mayor sea el porcentaje de la RCL, mayor va a ser el número de diferentes resultados generados por el algoritmo constructivo, pero por tal motivo necesitaríamos más iteraciones de GRASP, lo cual inevitablemente sería mucho más costoso en tiempo.

Por esta razón es la que decidimos fijar el parámetro del porcentaje de la RCL en 20 %, ya que es el que mejor resultados nos da, asumiendo que tenemos un tiempo acotado para conseguir la mejor relación tiempo-beneficio. Pero sin duda si tuviesemos algo más de tiempo aumentaríamos este parámetro conjuntamente con el de iteraciones de GRASP. ($PORCENTAJE = 20$)

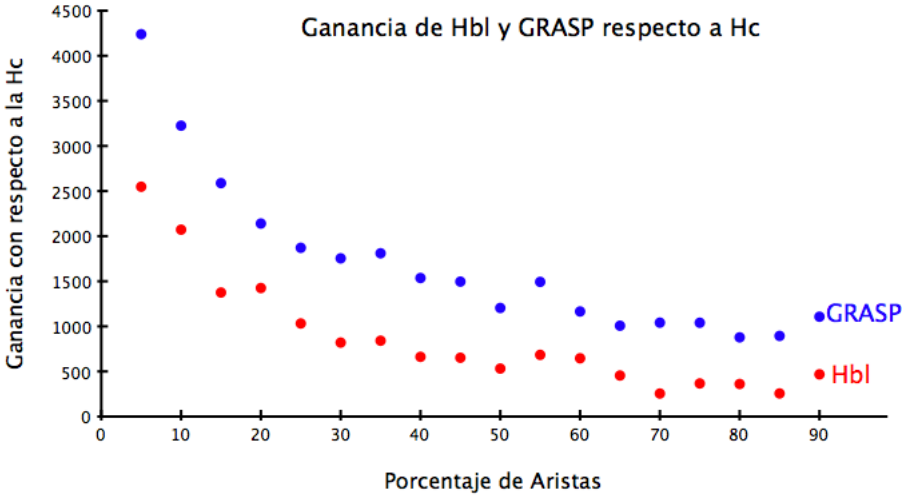
7. Conclusiones

Luego de fijar los parametros de la heurística GRASP, quisimos hacer un grafico final para comparar los resultados de las diferentes heurísticas.

Para comparar el comportamineto de las diferentes heurísticas, conjuntamente con GRASP lo que hicimos fue correr los tres algoritmos (HC, HBL, GRASP) sobre las mismas instancias.
Utilizamos diez instancias de pruebas con las siguientes caracteristicas.

- Grafo conexo
- 500 nodos
- Incrementamos las aristas de a un 5 %

Utilizamos grafos de 500 nodos, ya que con la cantidad de instancias que iba a correr, demoraba mucho tiempo, pero el comportamiento que queriamos destacar, se puede observar de todas maneras.



En este gráfico queremos mostrar la diferencia de los resultados generados por $GRASP$ y HBL con respecto al de la *heurística Constructiva*. Como podemos observar en el gráfico, en todos los casos $GRASP$ nos da un mejor resultado, como era de esperar. Sobre todo en los casos en los que HBL cae en un máximo local en la vecindad de la solución dada por la *heurística Constructiva*.

8. Conclusion Final

Como conclusión de este trabajo destacamos la importancia de las heurísticas como recursos para resolver problemas en los cuales el algoritmo exacto es exponencial, con lo cual, no se podrían resolver instancias con un tamaño de entrada grande, ya que demandaría incluso años de tiempo de ejecución.

Como vimos a lo largo de todo el trabajo, estas heurísticas nos permiten acercarnos a una buena solución en un tiempo razonable.
Por otra parte, si quisieramos mejores resultados podríamos invertir tiempo de ejecución, lo que nos proporcionaria soluciones más cercanas a la óptima, pero eso depende de las necesidades de cada problema.