DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Número 3

Algoritmos y Estructuras de Datos III

Grupo XV

Integrante	LU	Correo electrónico
Ferrante, Alejandro	371/09	matapalabras@gmail.com
González, Tomás Abel	237/14	tagonzalez95@gmail.com
Muchinik, Francisco	238/14	franmuchinik@hotmail.com
Gallardo, Agustin	502/10	agallardo@dc.uba.ar

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

Índice

1. Ejercicio 1

1.1. Problema

En el paper se muestran esfuerzos por intentar resolver algunos casos particulares de "maximum common subgraph", se explican diferencias entre MCS exacto y aproximado, luego se distingue también entre conexos e inconexos y se divide a los mismos entre MCIS (a partir de vértices) y MCES (a partir de aristas). En el trabajo práctico el problema a resolver es el de "maximum common subgraph isomorphism", que aunque muy similar, es diferente por poder tratar a los isomorfismos de un subgrafo como instancias válidas de la respuesta.

1.2. Diferencias con MCS

Se le da mayor importancia al MCIS (el máximo común subgrafo a partir del conjunto de vértices). Dentro del campo de la química, existen además particularidades como la valencia de los diferentes componentes químicos, que modifican una arista más allá de la información de sus vértices, ya que importa también en qué nivel se ubica la misma, o los ángulos que mantienen entre vértices (ya que dan la forma 3D de una molécula). En los problemas estudiados en el paper es además de vital importancia la etiqueta de un nodo, ya que se debe distinguir entre átomos o moléculas distintas, siendo esto algo sin importancia para el problema que se busca resolver en el trabajo práctico.

1.3. Aplicaciones

Las aplicaciones de las que se hace mención en el paper son búsquedas en bases de datos de moléculas, identificación de componentes activos de las mismas, desarrollo de fármacos mediante simulaciones y aproximaciones de los efectos de ciertos componentes. Usando los algoritmos nombrados en el paper los investigadores pueden encontrar similitudes entre componentes químicos y obtener mayor información topológica que usando un algoritmo simple de MCS, la información extra es de vital importancia a la hora de buscar ligandos con actividad biológica similar.

2. Ejercicio 2

En esta sección detallaremos la implementación de un algoritmo exacto para resolver el problema de MCS, y así convencer a Marty de que este es el problema a resolver.

2.1. Algoritmo

El algoritmo utiliza backtracking para conseguir el MCS dados dos grafos G_1 y G_2. Consideramos a G_1 como el grafo con mayor cantidad de nodos entre los dos. Gracias a esto, sabemos que podemos incluir todos los nodos de G_2 en el subgrafo, y entonces eligiremos el mapeo de G_2 como la función identidad, y nos concentraremos únicamente en el de G_1. Luego de procesar la entrada y cargar las estructuras correspondientes, el algoritmo llama a la función *llamarBacktracking* que se encarga de asegurarse que G_1 sea el grafo con mayor cantidad de nodos, inicializar estructuras, llamar al procedimiento de backtracking y devolver la respuesta como un arreglo de aristas.

Luego, el programa entra en la función *backtracking* la cual toma los siguientes parámetros:

- un arreglo de enteros *mapActual*: el mapeo de G_1 que tenemos realizado hasta el momento (tiene valores indefinidos al llamarse desde *llamarBacktracking*).
- un entero *comunActual*: la cantidad de aristas en común que tiene nuestro mapeo, la cantidad de aristas es lo que buscamos maximizar con el MCS (vale 0 al llamarse desde *llamarBacktracking*).
- un entero *mapProx*: el nodo próximo a ser mapeado (vale 0 al llamarse desde *llamarBacktracking*).
- un arreglo de bool *noMapeadas*: define en la posición i del arreglo si el nodo i está mapeado o no (tiene *true* en todas las posiciones al llamarse desde *llamarBacktracking*).

Llamamos a N, N' a las cantidades de nodos de cada grafo de entrada tales que $N \leq N'$ y M,M' a la cantidad de aristas asociada a G_1 y G_2 respectivamente. Las complejidades no comentadas son $O(1)$.

```
1: function BACKTRACKING(arreglo mapeoActual, int comunActual, int mapProx, arreglo de
   bool noMapeadas)
2:     if mapProx < N then
3:         for cada nodo i de G1 / noMapeadas[i] do                    ▷ O(N' − mapProx)
4:             comunNuevo ← comunActual
5:             noMapeadas[i] ← false
6:             mapeoActual[mapProx] ← i
7:             for cada nodo v adyacente a mapProx en G2 do                        ▷ O(M)
8:                 if mapeoActual[v] es adyacente a i en G1 then                   ▷ O(M')[1]
9:                     comunNuevo++
10:                end if
11:            end for
12:            if not poda(mapProx + 1, comunNuevo, maxComun) then                 ▷ O(N)
13:                backtracking(mapeoActual, comunNuevo, mapProx + 1, noMapeadas)
14:            end if
15:            if mapProx = N − 1 ∧ maxComun < comunNuevo then
16:                maxComun ← comunNuevo
17:                mapeoOptimo ← mapeoActual
18:            end if
19:            noMapeadas[i] ← true
20:            mapeoActual[mapProx] ← −1
21:        end for
22:    end if
23:    return mapeoOptimo
24: end function
```

```
function PODA(int mapProx, int comunNuevo, int comunMax)
    diferencia ← comunMax − comunNuevo
    for cada nodo i en G2 do
        diferencia ← diferencia − d_G2(i)
        if diferencia ≤ 0 then
            return false
        end if
    end for
    return true
end function
```

La idea general del algoritmo es, sabiendo ya cómo es el mapeo de G_2, elegir el de G_1 que maximice el tamaño del subgrafo común. Para esto, se utiliza el invariante de que si $mapProx = i$, todas las posiciones menores que i ya tendrán algún nodo de G_1 mapeado. Cuando todas las posiciones tengan un mapeo, se volverá hacia atrás de a una posición, asegurándose de llenar el mapeo con todas las posibles posibles. Luego de realizar cada map, se verifica si alguno de los vecinos del nodo de G_2 correspondiente a *mapProx* se mapea a la misma posición que alguno de los vecinos del nodo mapeado de G_1, y aumenta la cantidad de aristas en común que tiene el mapeo actual adecuadamente. Se vé en el código que al final de la función antes de retornar el mapeo óptimo para nuestra solución se reinician los valores de *noMapeadas* y *mapeoActual* a los de previo a entrar a la iteración. De esta manera, se logra pasar por todos los posibles mapeos, guardando el mejor hasta el momento en nuestra variable global *mapeoOptimo*. Una vez que se hayan recorrido todas las combinaciones posibles para los mapeos, *mapeoOptimo* tendrá un arreglo de tamaño N donde para cada posición i tendrá como valor el mapeo del nodo i.

Luego de actualizar el mapeo, se llama a la poda, de forma de evitar recorrer casos que necesariamente son peores que los que tenemos. Esta busca, de una manera no muy costosa dado que se llamará en cada iteración, eliminar ramas de iteraciones que no pueden mejorar el máximo encontrado. Por ejemplo, supongamos que tenemos $N = 5$, $mapProx = 4$, $d(4) = 3$, $d(5) = 2$, $comunNuevo = 4$, $comunMax = 10$. Como a lo sumo se podrán agregar 5 aristas más (inclusive con $(4, 5)$ potencialmente siendo contada dos veces), y $comunNuevo + 5 < comunMax$, se devuelve $false$.

Al retornar de esta función continuamos con la ejecución de la función *llamarBacktracking*.

```
function LLAMARBACKTRACKING
    ...
    arreglo de aristas res
    for cada nodo i en G2 do
        for cada vecino j de i en G2 do
            unMapi ← mapeoOptimo[i]          ▷ El mapeo del nodo i
            unMapv ← mapeoOptimo[j]          ▷ El mapeo del nodo j
            if unMapi ≤ unMapv then
                if si la arista (unMapi, unMapv) está en G1 then
                    agrego (i, j) a res
                end if
            end if
        end for
    end for
    return res
end function
```

Esta parte de la función encuentra las aristas del subgrafo común conformado por los mapeos elegidos. Para ello, recorre las aristas adyacentes a cada nodo de G_2, y se busca si estas aristas pertenecen, después de aplicar el mapeo, a G_1. Por cada arista, sólo se permite agregarla a la respuesta una vez, gracias a la comparación de índices (si no, aparecería una vez por cada extremo de la arista).

[1] Verificar esto requiere recorrer la lista de adyacentes del nodo i para encontrar el mapeo del vecino actual si es que existe.

2.2. Complejidad

En la sección anterior, llamamos N, N' a la cantidad de nodos de cada grafo, de manera que $N \leq N'$, y M,M' a la cantidad de aristas asociadas.

La función *llamarBacktracking* se encarga de inicializar las estructuras para llamar a la función *backtracking* y la llama. Hasta este punto la función cumple con una complejidad lineal pues la inicialización de estructuras es lineal y llamar a una función es constante. Luego una vez que se retorna de la función *backtracking* se realiza un recorrido de todo el grafo de menor tamaño, verificando si las aristas de G_2 pertenecen al subgrafo común y agregándolas al arreglo resultado en caso de que lo sean. Este recorrido por todo el grafo más chico tiene un costo de $O(N + (MM'))$, dado que se recorren todos los nodos de G_2 y sus aristas, y para cada arista se recorren potencialmente todas las de G_1. Esta cota no es muy fina, pero no importará si consideramos que el backtracking tendrá costo exponencial.

Luego calculamos el costo de la funcion *backtracking*. Dado que ésta recorre todos los posibles mapeos que se puedan realizar para resolver el problema de MCS, la función tendrá una complejidad temporal no polinomial. Se ve en el pseudocódigo que cuando $mapProx$ está en rango, el algoritmo entra en un ciclo de $N' - mapProx$ iteraciones, puesto que habrá una posición de G_1 ocupada por cada llamado anterior al algoritmo. Adentro del ciclo, recorre $d_{G_2}(mapProx)$ aristas del grafo de menor tamaño y para cada una de ellas, recorre los nodos adyacentes al nodo al cual se mapea $mapProx$, $mapProx'$. Esto resulta en una complejidad de $O(d_{G_2}(mapProx)d_{G_1}(mapProx'))$. También se debe considerar el costo de la poda, que es $O(N)$, puesto que se recorren los nodos de G_2, y se pregunta por su grado. Luego, se llama a la función recursivamente, incrementando el valor de $mapProx$ por una unidad. Es decir, consideramos una complejidad de $O((N' - mapProx)(d_{G_2}(mapProx)d_{G_1}(mapProx') + N))$ para la función sin contar la llamada recursiva. Por simplicidad, acotaremos $d_{G_2}(mapProx)d_{G_1}$ por MM' y $N' - mapProx$ por N'. Esta última cota, particularmente si $N' = O(N)$, resultará en que la complejidad calculada sea muy grande respecto a la real, dado que para valores más grandes de $mapProx$, el número de llamados recursivos se vuelve $O(1)$.

Consideramos la siguiente función $f : [0, \ldots, N] \to \mathbb{N}$,

$$f(mapProx) = \begin{cases} N'(MM' + N + f(mapProx + 1)) & \text{if } mapProx < N \\ 1 & \text{if } mapProx = N \end{cases}$$

La función definida devuelve el costo del algoritmo para el valor de $mapProx$ que se pasa por parámetro. La variable pasa por todos los valores enteros de 0 a N hasta terminar. Entonces podemos hacer los siguientes cálculos para determinar la complejidad de nuestro algoritmo, dado que inicialmente se lo llama con $mapProx = 0$:

$$\begin{aligned} f(0) &= N'(MM' + N + f(1)) \\ &= N'(MM' + N) + N'f(1) \end{aligned} \tag{1}$$

Aplicamos la definición para $f(1)$:

$$\begin{aligned} f(0) &= N'(MM' + N) + N'f(1) \\ &= N'(MM' + N) + N'(N'(MM' + N + f(2))) \\ &= N'(MM' + N) + N'^2(MM' + N) + N'^2 f(2) \end{aligned} \tag{2}$$

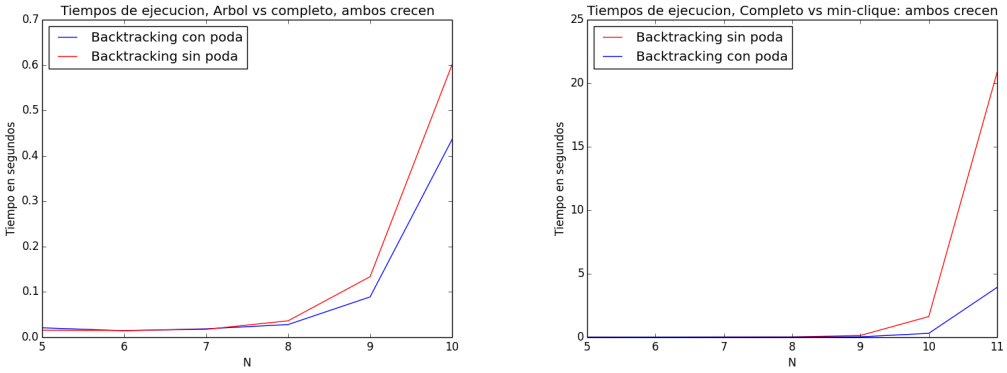
Aplicamos la definición para $f(2)$:

$$
\begin{aligned}
f(0) &= N'(MM' + N) + N'^2(MM' + N) + N'^2 f(2) \\
f(0) &= N'(MM' + N) + N'^2(MM' + N) + N'^2(N'(MM' + N) + N' f(3)) \\
&= N'(MM' + N) + N'^2(MM' + N) + N'^3(MM' + N) + N'^3 f(3) \\
&\vdots \\
&= \cdots + N'^N(MM' + N) + N'^N f(N) \\
&= \cdots + N'^N(MM' + N) + N'^N \text{pues } f(N) = 1 \\
&= \sum_{i=1}^{N} N'^i(MM' + N) + N'^N \\
&= (MM' + N) \sum_{i=1}^{N} N'^i + N'^N \\
&= O((MM' + N)\frac{N'(N'^N - 1)}{N' - 1} + N'^N) \\
&= O((MM' + N)N'^N)
\end{aligned}
\tag{3}
$$

Con estos resultados, obtenemos que nuestro algoritmo cumple con una complejidad de $O((MM' + N)N'^N)$.

2.3. Experimentación

En esta sección vamos a detallar la experimentación que se llevó a cabo para el algoritmo de backtracking realizado para resolver el problema de *MCS*. Variaremos entre correr el algoritmo con la poda y sin ella. Tomamos un grupo de instancias conformado por un árbol binario completo, un grafo completo, y un grafo con una clique mínima de un tamaño especificado (llamaremos a este tipo de instancia *min-clique*), por lo general será $N/4$ donde N es el número de nodos.



(a) Arbol vs completo

(b) Completo vs min-clique

Figura 1: Experimento 2.1

En el experimento 2.1 se llama al algoritmo con un árbol y un grafo completo (subfigura izquierda), donde estos crecen en tamaño simultaneamente a lo largo de las instancias, realizamos esto mismo con un grafo completo y un grafo *min-clique* (subfigura derecha). Vemos que los tiempos de ejecución del algoritmo sin el uso de la poda son mayores a los tiempos con la poda. Esto se debe a que no se descarta ninguna posibilidad y se prueba todas las posibles permutaciones de las soluciones sin importar que tan malas sean desde un principio. Dado que los tiempos de ejecución de una instancia a la otra incrementan considerablemente como se vé en la figura no pudimos conseguir resultados para valores de N mayores en un tiempo razonable.

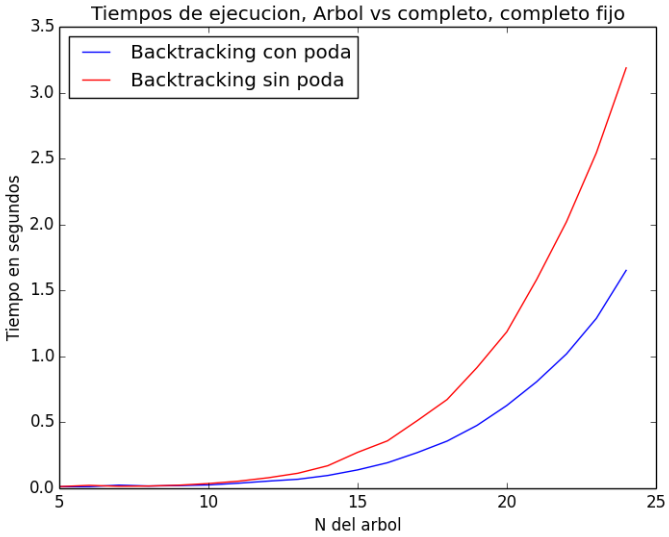
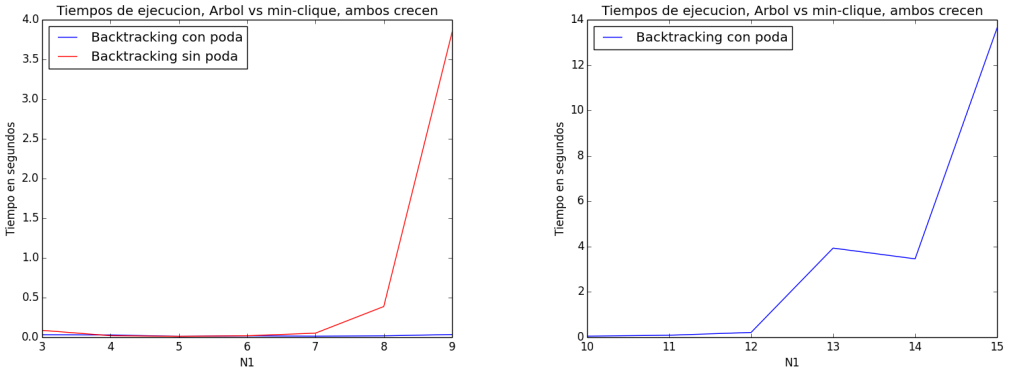


Figura 2: Experimento 2.2

Para este experimento tomamos las mismas instancias que en el anterior, pero decidimos mantener fijo el tamaño del grafo completo y observar los tiempos de ejecución respecto del aumento al tamaño del árbol. En la figura se pueden observar curvas más pronunciadas e instancias con mayor tamaño respecto al experimento anterior. Esto ocurre porque al fijar el tamaño de uno de los grafos, la complejidad se vuelve polinomial. Tenemos una complejidad $O((MM'+N)N'^N) = O((kM'+c)N'^c) = O(kM'N'^c)$ con k, c constantes.

Se ve nuevamente que el uso de la poda reduce el tiempo de ejecución del algoritmo.



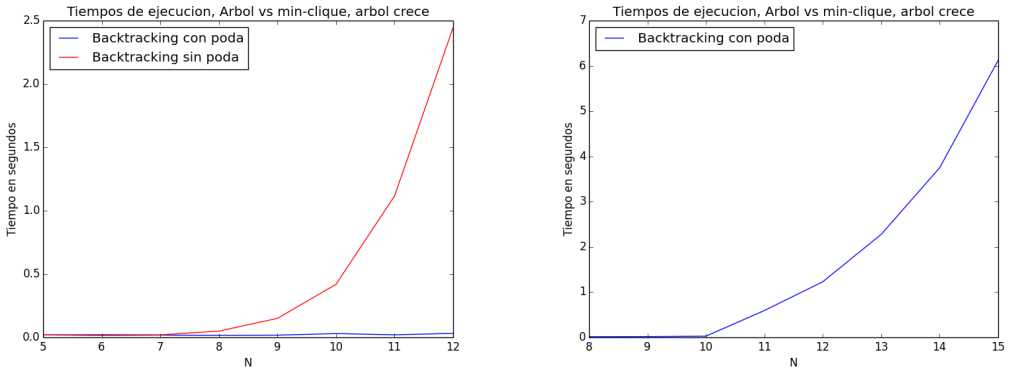
(a) Con y sin poda

(b) Sin poda

Figura 3: Experimento 2.3

En el experimento 2.3 comparamos los tiempos de ejecución para los tipos de instancias árbol y *min-clique* donde ambos crecen en tamaño de una instancia a la otra. Aquí encontramos una diferencia importante entre los tiempos de ejecución para el algoritmo con la poda y sin la poda. Para poder apreciar mejor el comportamiento del algoritmo con la poda incrementamos las instancias y experimentamos solamente para este algoritmo. Los resultados se ven en la subfigura derecha. Ambos algoritmos muestran un patrón muy similar: a partir de un tamaño N_0 el tiempo se dispara y comienza a crecer exponencialmente, haciendo complicado conseguir buenos resultados en un tiempo razonable.

Para el experimento 2.4 ejecutamos los algoritmos tomando las mismas instancias del anterior, pero mantuvimos fijo el *min-clique* mientras aumentamos el tamaño del árbol. Vemos que tenemos resultados similares al experimento anterior ya que hay una diferencia notable entre los tiempos con y sin la poda. En particular, para este tipo de experimento, el algoritmo tendrá una complejidad



(a) Con y sin poda

(b) Sin poda

Figura 4: Experimento 2.4

polinomial de la misma forma que en el experimento 2.2. Sin embargo, dado el tamaño reducido de las instancias, cualquier tamaño razonable para el *min-clique*, aunque se mantenga fijo, seguirá pareciendosé a N. Esto resulta en curvas que crecerán exponencialmente.

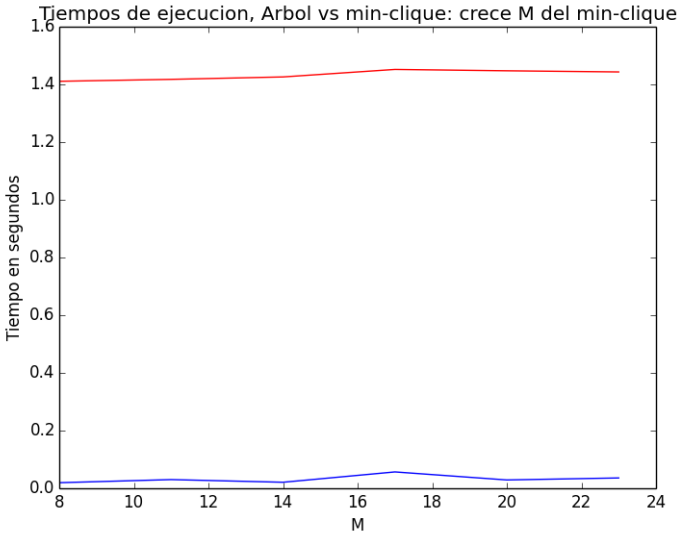


Figura 5: Experimento 2.5

En el experimento 2.5 tomamos las mismas instancias que en los dos últimos experimentos. En este experimento decidimos mantener fijos los números de nodos de ambas instancias y variar el número de aristas del grafo *min-clique*. Los resultados obtenidos muestran que variar la cantidad de aristas no influye de una manera importante en el tiempo de ejecución. Dado que, para un N fijo, existe una cantidad acotada de aristas posibles, se puede acotar M por una constante dado que, a lo sumo, habrá $O(N^2)$ aristas donde N es constante.

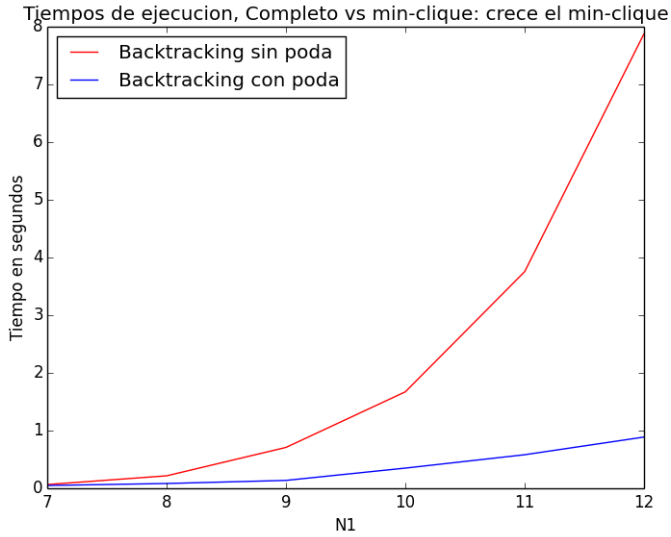


Figura 6: Experimento 2.6

Este experimento es muy similar a los experimentos anteriores (2.2, 2.4) donde se mantuvo a uno de los grafos con el tamaño fijo y se varió el tamaño del otro. Vemos nuevamente que los tiempos con la poda son menores a los tiempos sin la poda y que al fijar uno de los grafos de entrada conseguimos una complejidad polinomial para el algoritmo.

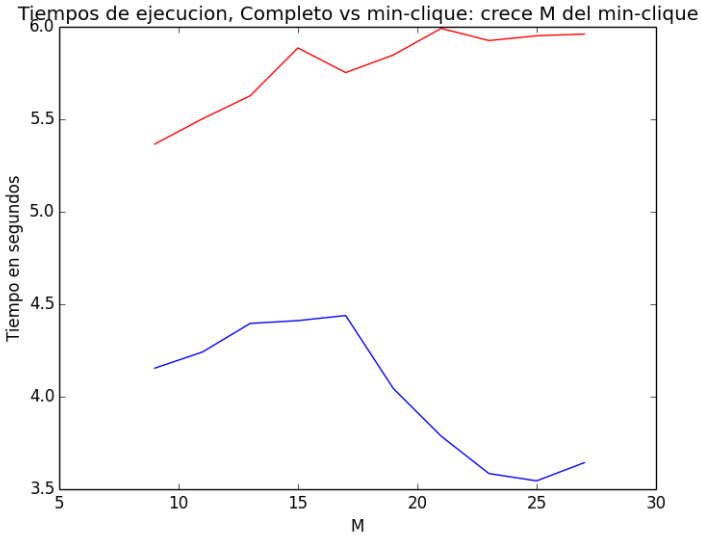


Figura 7: Experimento 2.7

Para este experimento, se tomó un grafo completo y uno *min-clique* y se incrementó la cnatidad de aristas de este último, manteniendo las cantidades de nodos de ambos fijas. Por un lado, podemos observar que el tiempo de ejecución para el algoritmo con la poda, se reduce a medida que aumentamos la cantidad de aristas. Dado que la manera en que se generan el *min-clique* es primero generando la clique y luego rellenando el grafo con la cantidad de aristas restantes de manera aleatorea para llegar a las deseadas, puede ocurrir que para ciertas instancias el algorimo alcance una solución actual muy buena con lo que la poda se activa para una cantidad mayor de soluciones, reduciendo el tiempo de ejecución.

Por otro lado, el algoritmo sin la poda incrementa sus tiempos respecto de la cantidad de aristas. Sin embargo, estos incrementos no son muy considerables dado que, para las instancias finales, este se estanca por las razones que vimos en el experimento 2.5.

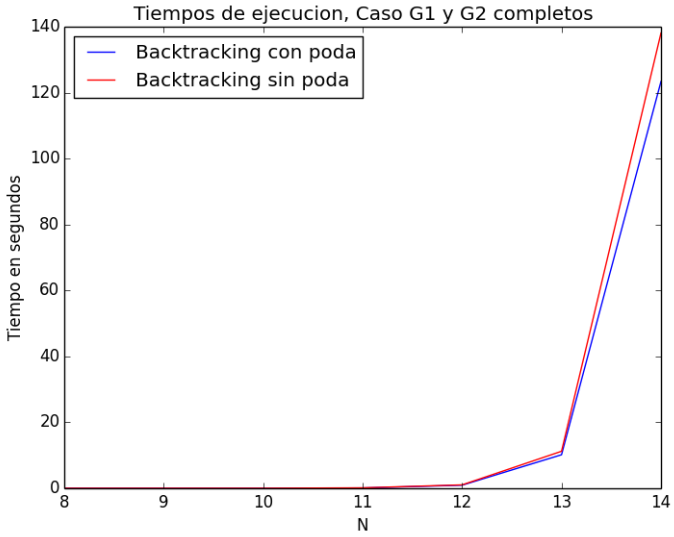


Figura 8: Experimento 2.8

En general, se vio que la poda es una buena técnica para reducir los tiempos de ejecución del algoritmo para los distintos tamaños de entrada. Sin embargo, su efectividad no es garantizada puesto que depende de la entrada. Por ejemplo, en el experimento 2.8, graficamos los tiempos para dos grafos completos del mismo tamaño y vemos que las diferencias en los tiempos de ejecución son despreciables. Esto ocurre porque en este caso la poda solamente se activaría para cuando el algoritmo solo debe llenar un último mapeo dado que todas las soluciones parciales son igual de buenas.

De todas formas, el uso de la poda para la mayoría de las entradas posibles, es, según nuestra experimentación, es al menos tan malo como no usarla.

3. Ejercicio 3

En esta sección se presenta el algoritmo para resolver el problema de MCS, dada la particularidad que G_1 es un cografo y G_2 un grafo completo.

Con esta entrada, buscamos cumplir con que el algoritmo presentado tenga una complejidad temporal de orden polinomial.

3.1. Algoritmo

3.1.1. Estructuras de datos

Para este problema, se lidia con un cografo G_1. Un cografo es un grafo que puede ser definido de la siguiente manera:

- todo grafo K_1 es cografo
- si G es cografo, su complemento es cografo
- si G y H son cografos, la unión disjunta es cografo

Un cografo puede ser representado mediante una estructura de datos conocida como un *cotree* (co-árbol). Este es un árbol que cumple con las siguientes propiedades:

- un subárbol que consista de una única hoja y nada más corresponde a un subgrafo inducido de un K_1
- un subárbol con raíz de valor 0 corresponde a la unión de los subgrafos representados por los hijos de la raíz
- un subárbol con raíz de valor 1 corresponde al *join* de los subgrafos representados por los hijos de la raíz

Se diseño una estructura *nodoCoTree*, que representa un nodo de un cotree y cuenta con la siguiente estructura interna, dado un nodo D:

- un entero $hojas$ con la cantidad de hojas del subárbol con D como raiz.
- un entero *tipo* con el tipo del nodo. Si el nodo es hoja, el tipo será el número de nodo en G_1. Si no, será 0 ó 1 dependiendo de si la unión representada es disjunta ó un *join* respectivamente.
- vector de vector de int $nodoMax$, donde $nodoMax[i]$ = una lista de i nodos de G_1 pertenecientes al subgrafo actual tales que maximizan la cantidad de aristas.
- vector de int $scores$, donde $scores[i]$ = la cantidad máxima de aristas posibles provenientes de elegir cualquier i nodos contenidos en el subárbol. La relación con $nodoMax$ es tal que la cantidad de aristas obtenida al elegir los nodos en $nodoMax[i]$ es igual a $scores[i]$.

3.1.2. Ejemplo

El algoritmo implementado busca aprovechar las características de la entrada. En primer lugar, se puede intuir que si $N_1 \leq N_2$, el MCS para el caso es el cografo en sí.

Si no, obtener la solución resulta en un procedimiento más complejo. El objetivo en este caso es elegir un subgrafo de G_1 con N_2 nodos tales que la cantidad de aristas sea máxima. Para esto, se aprovecha la representación cotree del cografo. Veamos el siguiente ejemplo:

Tenemos un cografo G_1 de 8 nodos y un grafo completo G_2 de 6 nodos.

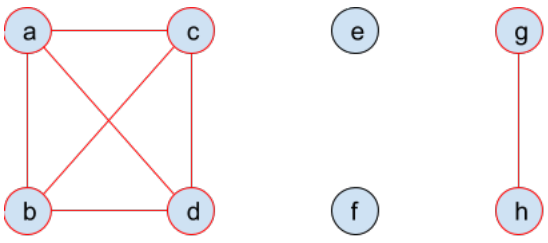


Figura 9: Cografo G_1

En la imagen vemos que los nodos y aristas que forman la solución a nuestro problema están marcados en rojo. Como ya dijimos, el algoritmo tomará la entrada y convertira a cotree el grafo G_1.

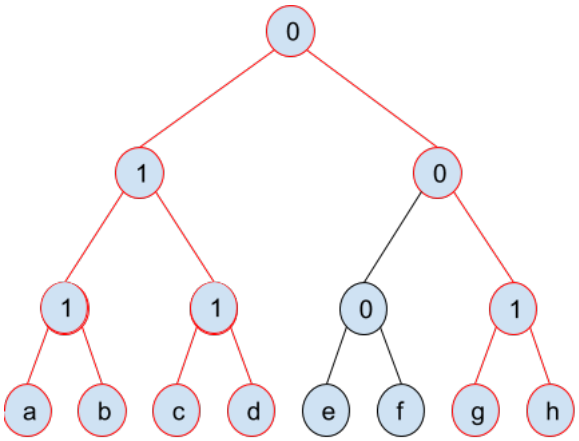


Figura 10: Cotree G_1

Vemos nuevamente en la imagen la solución en nuestro problema marcado en rojo. Partiendo de la raíz, la solución implica tomar 2 nodos del subárbol derecho y 4 nodos del subárbol izquierdo. La decisión sobre cuántos nodos tomar de cada subárbol es fundamental a nuestro algoritmo. En el algoritmo, se consideraran todas las combinaciones posibles para esta decisión.

Llamamos P_i al conjunto de pares (a, b) de números naturales (el cero incluído) tales que $a+b = i$. El primer elemento del par será los nodos a tomar del izquierdo y el segundo los nodos a tomar del subárbol derecho. Para este caso, queremos encontrar $N_2 = 6$ nodos pues el MCS tendrá N_2 nodos. Entonces para el nodo raíz, probaremos todas las combinaciones encontradas en P_6. Para los subárboles, tanto el derecho como el izquierdo tienen solamente 4 hojas, por lo cual solamente podemos elegir 1,2,3, o 4 nodos. Consideramos entonces todas las combinaciones de la unión de los conjuntos $P_1, \ldots, P_4$. Para cada subárbol de los subárboles, cada uno tiene 2 hojas, entonces consideramos las combinaciones de P_1, C_2.

Nodo interno	i	P_i
$raiz$	6	$\{(0,6),(1,5),(2,4),(3,3),(4,2),(5,1),(6,0)\}$
$raiz \to der$	1	$\{(0,1),(1,0)\}$,
	2	$\{(2,0),(1,1),(0,2)\}$,
	3	$\{(0,3),(1,2),(2,1),(3,0)\}$,
	4	$\{(0,4),(1,3),(2,2),(3,1),(4,0)\}$
$raiz \to izq$	1	$\{(0,1),(1,0)\}$,
	2	$\{(2,0),(1,1),(0,2)\}$,
	3	$\{(0,3),(1,2),(2,1),(3,0)\}$,
	4	$\{(0,4),(1,3),(2,2),(3,1),(4,0)\}$
$raiz \to der \to izq$	1	$\{(0,1),(1,0)\}$,
	2	$\{(0,2),(1,1),(2,0)\}$
$raiz \to der \to der$	1	$\{(0,1),(1,0)\}$,
	2	$\{(0,2),(1,1),(2,0)\}$
$raiz \to izq \to izq$	1	$\{(0,1),(1,0)\}$,
	2	$\{(0,2),(1,1),(2,0)\}$
$raiz \to izq \to der$	1	$\{(0,1),(1,0)\}$,
	2	$\{(0,2),(1,1),(2,0)\}$

La tabla nos muestra, para cada nodo interno, todas las distintas combinaciones que el algoritmo prueba. Para cada conjunto P_i, la combinación que resulte tener el *score* más alto, lo guardará en $scores[i]$ y los nodos elegidos para llegar a ese *score* serán guardados en un vector dentro de $nodosMax[i]$. De esta manera, tendremos en $nodosMax[N_2]$ los nodos pertenecientes a la respuesta al problema de MCS. En nuestro ejemplo, será $nodosMax[6]$.

3.1.3. Pseudocódigo

```
1: function MCS
2:     if N_1 ≤ N_2 then
3:         for i ← 0 ... N_1 do                                   ▷ O(N_1)
4:             nodosRes[i] ← i
5:         end for
6:     else
7:         raiz ← conCoT(G_1)
8:         nodosRes ← (raiz → nodosMax[N_2])
9:     end if
10:    for cada nodo en nodosRes do                               ▷ O(max(N_1, N_2))
11:        for cada vecino v de nodo en G_1 do
12:            if v está en nodosRes then
13:                agrego (nodo, posicion(nodosRes, v)) a res
14:            end if
15:        end for
16:    end for
17:    return res
18: end function
```

Este pseudocódigo corresponde a la función principal de nuestro programa. Su funcionamiento consiste en primero guardar en *nodosRes* los nodos que serán parte del MCS y luego recorrerlo para determinar las aristas entre los nodos y agregarlas en la salida. A continuación veremos el pseudocódigo de la función *conCoT* que es la encargada de convertir al cografo G_1 en un cotree.

En la función hacemos uso de las siguientes variables:

- *compConexas*: vector de vector de enteros, cada elemento del vector es un vector con los nodos perteneciente a una componente conexa del subgrafo
- *subgrafos*: vector de subgrafos, cada componente conexa será convertida en un subgrafo, y se almacenará en esta variable
- *subtrees*: vector de punteros a nodoCotree, cada subgrafo será convertido en cotree y se almacenará el puntero a la raíz como elemento de esta variable

Llamamos C a la cantidad de componenentes conexas para el grafo de entrada en conCoT G.

```
1: function conCoT(subgrafo G)
2:     if G es K_1 then
3:         yo ← nodoCoTree(true, indiceOriginal(G), NULL, NULL)                ▷ O(N_2^2)
4:         return yo
5:     end if
6:     compConexas ← encontrarComponentes(G)                                   ▷ O(N(N + M))
7:     if el grafo no es conexo then
8:         subgrafos ← crearSubgrafosDisjuntos(compConexas, G)                 ▷ O(M_1(N_1^2))
9:         arreglo de cotrees subtrees
10:        for cada subgrafo i de subgrafos do
11:            agrego conCoT(i) a subtrees
12:        end for
13:        for i ← 1 ... |compConexas| − 1 do
14:            conexionDisjunta ← nodoCoTree(false, 0, subtrees[i − 1], subtrees[i])   ▷ O(N_2^2)
15:            subtrees[i] ← conexionDisjunta
16:        end for
17:        return subtrees[size(subtrees) − 1]
18:    else
19:        H ← complemento(G)                                                  ▷ O(N^3)
20:        compConexas ←encontrarComponentes(H)
21:        subgrafos ← crearSubgrafosDisjuntos(compConexas, G)                 ▷ O(M_1(N_1^2))
22:        arreglo de cotrees subtrees
23:        for cada subgrafo i de subgrafos do
24:            agrego conCoT(i) a subtrees
25:        end for
26:        for i ← 1 ... |compConexas| − 1 do
27:            conexionJoinnodoCoTree(false, 1, subtrees[i − 1], subtrees[i])   ▷ O(N_2^2)
28:            subtrees[i] ← conexionJoin
29:        end for
30:        return subtrees[size(subtrees) − 1]
31:    end if
32: end function
```

Finalmente tenemos la función mediante la cual se crea un nodo del *coTree*. La función calcula los *scores* para cada nodo y los guarda en las estructura de datos.

```
1: function nodoCoTree(bool esHoja, int type, nodoCoTree* I,nodoCoTree* D)
2:     tipo ← type
3:     izq ← I
4:     der ← D
5:     if esHoja then
6:         hojas ← 1
7:     else
8:         hojas ← izq → hojas + der → hojas
9:     end if
10:    intnodosAObservar ← min(hojas + 1, N2 + 1)
11:    scores[0] ← 0
12:    scores[1] ← 0
13:    if esHoja then
14:        nodosMax[1].push_back(tipo)
15:    else
16:        nodosMax[1].push_back(izq → nodosMax[1][0])
17:    end if
18:    for i ← 2 ... nodosAObservar do                                    ▷ O(N_2)
19:        maxScore ← −1
20:        for j ← 0 ... i + 1 do
21:            if izq → scores.size() > j and der → scores.size() > i − j then
22:                scoreActual ← izq → scores[j] + der → scores[i − j]
23:                if type = 1 then
24:                    scoreActual ← scoreActual + izq → nodosMax[j].size() ∗ der → nodosMax[i −
   j].size()
25:                end if
26:                if maxScore < scoreActual then
27:                    maxScore ← scoreActual
28:                    ladoI ← &(izq → nodosMax[j])
29:                    ladoD ← &(der → nodosMax[i − j])
30:                end if
31:            end if
32:        end for
33:        scores[i] ← maxScore
34:        for h ← 0 ... i do                                             ▷ O(N_2)
35:            if h < ladoI → size() then
36:                nodosMax[i][h] = (∗ladoI)[h]
37:            else
38:                nodosMax[i][h] = (∗ladoD)[h − ladoI → size()]
39:            end if
40:        end for
41:    end for
42: end function
43: Complejidad: O(N_2^2)
```

3.2. Complejidad

Para calcular la complejidad total de nuestro algoritmo *MCS*, tenemos que considerar el costo de muchas funciones auxiliares que fueron usadas:

Llamamos $C = \{c_1, \ldots, c_{|C|-1}\}$ al vector de componentes conexas generado por la función *encontrarComponentes*.

- el constructor de nodo de cotree *nodoCoTree*: $O(nodosAObservar^2) = O((\min(hojas + 1, N_2 + 1))^2) = O(N_2^2)$, esto se vé claramente en el pseudocódigo encontrado en la sección anterior
- la función *complemento*: esta función toma un grafo de N nodos y devuelvo su complemento. Esto implica crear un grafo nuevo H del mismo tamaño ($O(N)$) generar todas las aristas posibles

y agregarlas a H si y solo si no están en el grafo G. La llamamos en el programa con grafo de entrada G_1, entonces la complejidad es $O(N_1^2)$

- la función *encontrarComponentes*: toma un grafo G con N vértices y M aristas y devuelve un vector con las componentes conexas. Esto se realiza llamando a el algoritmo *BFS* ($O(N+M)$). El costo total es de $O(N_1 + M_1)$ pues recorre todos los nodos y todas las aristas del grafo G_1.
- el constructor de subgrafos *subgrafo*: Tenemos dos variantes, una que toma solamente un grafo y otra que toma un grafo G más un vector con los nodos de G pertenecientes al subgrafo. En nuestro caso, estos son los nodos de la componente conexa c_i. Por cada uno de estos nodos ($O(c_i)$) recorremos los vecinos del nodo en G_1 ($O(M_1)$), verificamos que el vecino pertenezca a la componente conexa ($O(c_i)$) y si pertenece la agregamos al subgrafo. Costo total: $O(c_i^2 M_1)$
- la función *crearSubgrafosDisjuntos*: toma el vector de componentes conexas generado por *encontrarComponentes* y genera subgrafos disjuntos. Para esto recorre el vector de componentes conexas y crea un subgrafo usando el constructor del item anterior.

Costo total de *crearSubgrafosDisjuntos*:

$$\begin{aligned} \sum_{i=0}^{|C|-1} |c_i|^2 M_1 &= M_1 \sum_{i=0}^{|C|-1} |c_i|^2 \\ \text{Sabemos que } \sum_{i=0}^{|C|-1} |c_i| &= N_1 \\ \Rightarrow \sum_{i=0}^{|C|-1} |c_i|^2 &\leq (\sum_{i=0}^{|C|-1} |c_i|)^2 = N_1^2 \\ \Rightarrow M_1 \sum_{i=0}^{|C|-1} |c_i|^2 &\leq M_1 N_1^2 \\ \Rightarrow \text{Costo: } O(M_1 N_1^2) & \end{aligned} \tag{4}$$

Como ya vimos, la función *MCS* tiene una llamada a la función *conCoT*. Esta función realiza varias llamadas a las funciones auxiliares que vemos en los ítems anteriores. Queremos calcular el costo total de *conCoT* para tomarlo en cuenta al calcular la complejidad de *MCS*.

Consideramos 3 ramas posibles de ejecución: el caso base, el paso recursivo con G_1 conexo, y el paso recursivo con G_1 no conexo.

Si estamos en la rama del caso base, el costo de la función es simplemente crear un nodo. El costo de esto es $O(N_2^2)$.

Si estamos en la rama del paso recursivo con G_1 conexo. Se toman en cuenta los siguientes costos:

- Costo de encontrar componentes: $O(N_1 + M_1)$
- Costo de crear el grafo complemento: $O(N_1^2)$
- Costo de crear subgrafos disjuntos a partir de las componentes conexas: $O(M_1 N_1^2$
- Para cada subgrafo, realizar una llamada recursiva de la función y así conseguir lo que llamamos un subárbol. Para cada una de estas llamadas, se entra en el caso base. Esto es porque al tomar complemento de un grafo conexo, se obtiene un grafo con nodos los cuales ninguno es adyacente a otro. Entonces se tienen N_1 subgrafos, cada uno isomorfo a un K_1. Por lo tanto, tenemos el costo de N_1 llamadas recursivas a la función las cuales entran al caso base. Costo: $O(N_1(N_2^2))$
- Para cada subárbol, crear un nodo. Costo: $O(N_1(N_2^2))$

Costo total de la rama G conexo:

$$O(N_1 + M_1) + O(N_1^2) + O(M_1 N_1^2) + O(N_1(N_2^2)) + O(N_1(N_2^2)) = O(M_1 N_1^2 + N_1(N_2^2))$$

Si estamos en la rama del paso recursivo con G_1 no conexo tenemos en cuenta los siguientes costos:

- Costo de encontrar componentes: $O(N_1 + M_1)$

- Costo de crear subgrafos disjuntos a partir de las componentes conexas: $O(M_1 N_1^2$
- Para cada subárbol creado en el siguiente item, crear un nodo. Costo: $O(|C|(N_2^2))$
- Tenemos $|C|$ subgrafos, para cada subgrafo c_i realizamos la llamada recursiva para crear el subárbol. Cada subgrafo también es conexo, con lo cual al realizar la llamada recursiva, entraremos en la rama con grafo de entrada conexo, pero en este caso el tamaño de ese grafo será $|c_i|$. Entonces, cada llamada recursiva tendra un costo de $O(O(M_1|c_i|^2 + |c_i|N_2^2)$. El costo total será la suma de estos costos de 0 a $|C| - 1$. Costo total:

$$\text{Sabemos que } \sum_{i=0}^{|C|-1} |c_i| = N_1$$

$$\text{También sabemos que } \sum_{i=0}^{|C|_1} |c_i|^2 \leq N_1^2$$

$$\text{Entonces, } \sum_{i=0}^{|C|-1} (M_1|c_i|^2 + |c_i|N_2^2) = \sum_{i=0}^{|C|_1} M_1|c_i|^2 + \sum_{i=0}^{|C|_1} |c_i|N_2^2 \tag{5}$$

$$\text{Entonces, } \sum_{i=0}^{|C|_1} (M_1|c_i|^2 + |c_i|N_2^2) = M_1 \sum_{i=0}^{|C|-1} |c_i|^2 + N_2^2 \sum_{i=0}^{|C|_1} |c_i|$$

$$\text{Entonces, } \sum_{i=0}^{|C|-1} (M_1|c_i|^2 + |c_i|N_2^2) \leq M_1 N_1^2 + N_2^2 N_1$$

Costo total de la rama G no conexo:

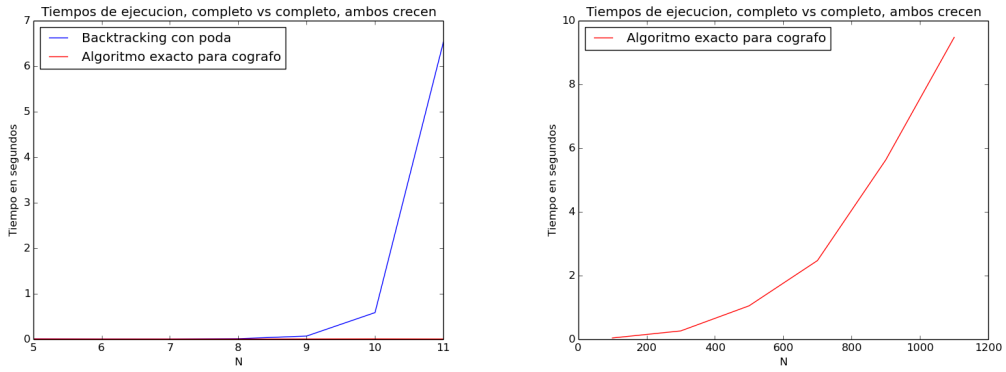
$$O(N_1 + M_1) + +O(M_1 N_1^2) + O(N_1(N_2^2)) + O(|C|(N_2^2)) = O(M_1 N_1^2 + N_1(N_2^2))$$

En conclusión, para el peor caso, la función *conCoT* tiene una complejidad de $O(M_1 N_1^2 + N_1(N_2^2))$. Dado que $N_1 > N_2$ en este caso, tenemos que la complejidad es $O(M_1 N_1^2 + N_1^3)$.

Entonces, la complejidad para el algoritmo de *MCS* en el peor caso es $O(M_1 N_1^2 + N_1^3)$ pues la llamada a *conCoT* es la operación más costosa.

3.3. Experimentación

En esta sección, se muestra como el algoritmo exacto para cografos es estrictamente mejor que el backtracking para instancias que cumplen la precondición del problema.

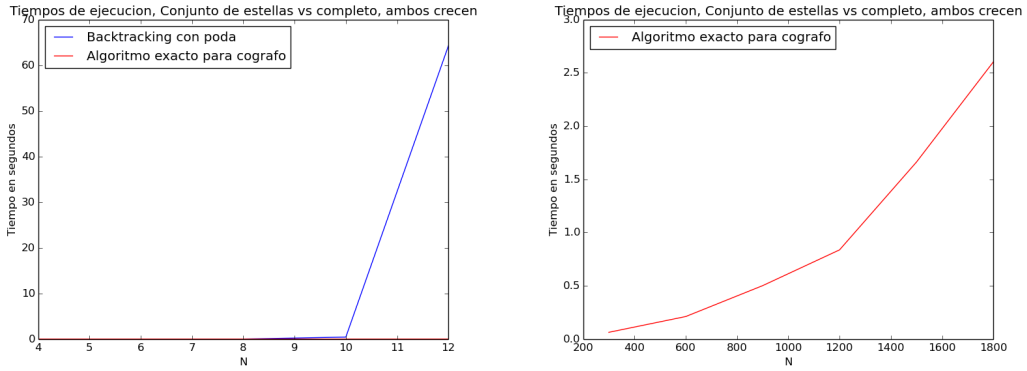


(a) (b)

Figura 11: Experimento 3.1

En el experimento 3.1, podemos ver en la subfigura izquierda, una comparación de tiempos para el backtracking y el algorimto para cografos, donde el cografo es un completo, y se aumenta la cantidad de nodos de ambos grafos (manteniendo más chico al grafo completo, dado que si no el algoritmo se realiza en tiempo constante). En este caso, el tiempo de ejecución del exacto es negligible comparado al

del backtracking, cuya complejidad es exponencial. Para observar mejor el funcionamiento del exacto para cografos, se repite el experimento, en la subfigura derecha, para éste algoritmo únicamente, con instancias de mayor cantidad de nodos. Se puede ver que el tiempo de ejecución para una cantidad de nodos del orden de 100 toma segundos. Esto indica que el algorimto es efectivamente polinomial, dado que tales cantidades son prohibitivas para un algoritmo exponencial.



(a) (b)

Figura 12: Experimento 3.2

En el experimento 3.2, se repite el experimento anterior, pero para un nuevo tipo de cografo: un camino, donde cada nodos en éste es el centro de una estrella. Se repite, también, lo que podemos observar. Por un lado, el algoritmo exacto para cografos se ejecuta en un tiempo menor a la décima de segundo para aquellas instancias para las cuales el backtracking tarda más de un minuto. Esto implica que se podrán realizar experimentos de mayor tamaño, como se ve en la subfigura derecha. En este caso, es posible procesar grafos de tamaños del orden de 100 en segundos.

4. Ejercicio 4

A partir de esta sección, nos importarán únicamente las heurísticas, dejando de lado los algoritmos exactos. En particular, todos los algoritmos deberán usar un método para, dados los mapeos de los grafos, encontrar las aristas en común. Éste es como sigue:

```
procedure aristasEnComun(grafo G_1, grafo G_2, arreglo MapG_1, arreglo MapG_2)
    arreglo UnMapG_1, UnMapG_2, res
    for G_k ∈ {G_1, G_2} do
        for i ∈ [0, |V_k|) do
            UnMapG_k[MapG_k[i]] ← i])
        end for
    end for
    for cada nodo i en G_2 do
        unMapi ← unMapG_2[i]
        for cada vecino j de i en G_2 do
            unMapj ← unMapG_2[j]
            if unMapi ≤ unMapj ∧ (MapG_1[unMapi], MapG_1[unMapj]) ∈ E_1 then
                agrego (unMapi, unMapj) a res
            end if
        end for
    end for
    return res
end procedure
```

La función toma los mapeos de G_1 y G_2, y devuelve las aristas del subgrafo común generado por tal mapeo. Primero, para ahorrar tiempo de cómputo luego, dado un mapeo, genera el mapeo inverso; es decir, si antes valía, dado un mapeo A y un mapeo inverso A', $A[i] = j$, tendremos $A'[j] = i$. Luego, se fija para cada arista posible de G_2, si ésta pertenece, calculando los índices correctamente, a G_1. En tal caso, la agrega a la respuesta (sólo lo hace una vez por arista, por eso se verifica que un índice sea menor que el otro).

La complejidad de la función es como sigue: recorrer los mapeos para llenar el $unMap$ cuesta $O(|V_1| + |V_2|)$. Recorrer cada nodo y las aristas de G_2 conlleva $O(|V_2| + |E_2|)$ iteraciones. Dentro de éstos ciclos todas las operaciones se realizan en $O(1)$, dado que son accesos a vectores, inserciones en un vector, y para saber si dos nodos son vecinos, se mantiene el grafo no sólo como lista de adyacencia, sino como matriz, de forma que se puede verificar en tiempo constante. Esto resulta en que la función cuesta $O(|V_1| + |V_2| + |E_2|)$.

4.1. Algoritmo

En esta sección, se describen implementaciones de heurísticas constructivas golosas para el problema de MCS. Este tipo de heurísticas busca construir una solución donde, en cada paso elige, basándose únicamente en la información que posee en ese momento, el mejor candidato a nivel local.

Nuestra primera intuición fue concentrarnos en los nodos de mayor grado. Decidimos mapear entonces el nodo de mayor grado del primer grafo al de mayor grado del segundo grafo, el de segundo mayor grado del primer grafo al segundo de mayor grado del segundo grafo, etc. Se continúa así hasta que se alcanza el tamaño máximo de algún grafo, de forma que no le queden nodos restantes que asignar.

Para el algoritmo, asumimos la precondición $|V_1| \geq |V_2|$, donde V_1 y V_2 son los conjuntos de nodos del primer y segundo grafo respectivamente. Para permitir una implementación simple, elegimos representar un grafo como lista de adyacencia, donde además cada nodo en la lista tiene guardado su número original, dado que se los reordenará. El algoritmo, además de los grafos, toma como parámetros arreglos donde guardará los mapeos hechos.

```
procedure greedy_1(grafo G_1, grafo G_2, arreglo MapG_1, arreglo MapG_2)
    ordenar los nodos de G_1 decrecientemente respecto a sus grados
    ordenar los nodos de G_2 decrecientemente respecto a sus grados
    for i ∈ [0, |V_2|) do
        MapG_1[i] ← posOriginal(G_1, G_1[i])
        MapG_2[i] ← posOriginal(G_2, G_2[i])
    end for
end procedure
```

La idea detrás de $greedy_1$ es intentar maximizar la cantidad de chances que tienen las aristas de los grafos de pertenecer al subgrafo determinado por los mapeos elegidos. Por ejemplo, se tienen los siguientes nodos de dos grafos y sus grados:

Nodo	Vecinos en G_1	Vecinos en G_2
0	3	2
1	3	3
2	4	0
3	2	3
4	1	1
5	0	4

Si corremos el algoritmo, el mapeo quedaría de la siguiente forma (eligiendo el nodo de mayor número en caso de comparar nodos de igual grado):

Nodo	Mapeo de G_1	Mapeo de G_2
0	2	5
1	0	1
2	1	3
3	3	0
4	4	4
5	5	2

Tomemos una arista $(2, i)$ cualquiera incidente al nodo 2 de G_1 y otra $(5, j)$ incidente al 5 de G_2. Esta arista pertenece al MCS si i se mapea a la misma posición que j. Con la información que se tiene, no se puede determinar la probabilidad de que esto ocurra. Notemos entonces que dada la arista $(2, i)$ en G_1, se la puede comparar con 4 aristas diferentes para ver si coinciden, dado que éste es el grado de 5 en G_2. Diremos que cada una de estas posibilidades es una "chance".

El algoritmo busca maximizar la cantidad de chances de que una arista pertenezca al MCS. Sin embargo, maximizar este valor no implica que el resultado final sea bueno. Algo que no tiene en cuenta, es que, por ejemplo, la arista $(2, 0)$ de G_1 cuenta dos veces sus chances, una vez desde cada una de sus nodos. Sería correcto que sumara una sola vez, dado que buscar nuevamente si la arista pertenece al subgrafo desde el nodo 0 no puede generar un resultado distinto a buscar desde el 2.

Pensamos entonces en una heurística que no busque maximizar la suma de chances de los nodos, sino que lo haga para la suma de chances de las aristas; es decir, que cada arista sea contabilizada una única vez.

Entonces, implementamos una nueva heurística que considere esto:

```
procedure greedy_2(grafo G_1, grafo G_2, arregloMapG_1, arregloMapG_2)
    for i ∈ [0, |V_2|) do
        MapG_1[i] ← buscarMaxYRemover(G_1)
        MapG_2[i] ← buscarMaxYRemover(G_2)
    end for
end procedure
```

$buscarMaxYRemover$ toma un grafo, al cual le busca el nodo de mayor grado y lo remueve:

```
procedure buscarMaxYRemover(grafo G)
    indiceMax ← −1
    maxVecinos ← 0
    for cada nodo i de G do
        if d(i) ≥ maxVecinos then
            indiceMax ← i
            maxVecinos ← d(i)
        end if
    end for
    indiceReal ← posOriginal(G, G[indiceMax])
    for cada nodo i de G do
        for cada vecino j de i en G do
            if j = indiceReal then
                borro j de los vecinos de i en G
            end if
        end for
    end for
    borro indiceMax de los nodos de G
    return indiceReal
end procedure
```

La idea del algoritmo es, similarmente al primer greedy, tomar los nodos de mayor grado y asignarlos entre sí, hasta quedarnos sin nodos. La diferencia reside en que una vez que se encuentra el nodo de mayor grado en el grafo actual, se lo remueve bajo la idea de que las aristas incidentes a ese nodo ya contabilizarán sus chances de pertenecer al MCS desde tal nodo, y no es necesario considerarlas nuevamente.

4.2. Complejidad

La heurística $greedy_1$ consiste en dos partes principales: el ordenamiento y la asignación. Por un lado, la asignación consiste en un ciclo que itera $|V_2|$ veces y realiza asignaciones en $O(1)$, resultando en $O(|V_2|)$. Por otro lado, se ordenan ambos grafos con *std::sort*, que tiene complejidad $O(NlogN)$, si N es el tamaño del arreglo[2]. La complejidad final es, por lo tanto, $O(|V_1|log|V_1| + |V_2|log|V_2| + |V_2|) = O(|V_1|log|V_1|)$, dado que $|V_2| \leq |V_1|$ por la precondición.

La heurística $greedy_2$, llama $2|V_2|$ veces a $buscarMaxYRemover$. Esta última función consiste principalmente en dos ciclos. En el primero, se recorre un grafo de N vértices y M aristas, y se busca el máximo y su índice. Esto se realiza en $O(N)$, puesto que dentro del ciclo se realizan únicamente asignaciones y comparaciones en $O(1)$. El segundo ciclo contiene otro adentro suyo, y recorre todos los nodos, y los vecinos de estos. Dentro de los ciclos, la operación más cara es el borrado de un vecino, que cuesta[3] $O(d(i))$. Dado que puede haber un único borrado por cada nodo, y que la sumatoria de los grados es equivalente a $2M$, todos los borrados costarán $O(M)$. La cantidad de iteraciones realizadas es $O(N + M)$, dado que se recorre cada nodo y, en el peor caso, todos sus vecinos. Esto resulta en que el segundo ciclo tenga complejidad $O(N + M)$. Además, se remueve de la lista de adyacencia al nodo de grado máximo, cuyo costo es $O(N + d(i)) = O(N + M)$, dado que en el peor caso se borra al primer elemento. Como esta función es llamada $|V_2|$ veces para cada grafo, la complejidad del algoritmo será $O(|V_2|(|V_1| + |E_1| + |V_2| + |E_2|)) = O(|V_2|(|V_1| + |E_1| + |E_2|))$.

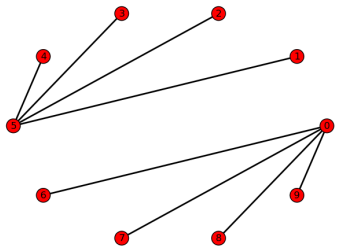
Además, al final del algoritmo se llama a $aristasEnComun$, que cuesta $O(|V_1| + |V_2| + |E_2|) = O(|V_1| + |E_2|)$, que no modifica la complejidad final.

4.3. Optimalidad

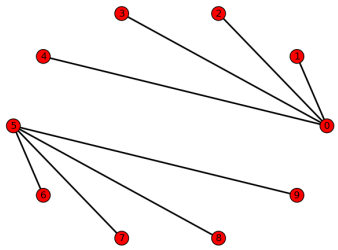
En general, una heurística golosa constructiva no resultará en la solución óptima, pero puede generar un resultado útil en la práctica. Sin embargo, la calidad de sus resultados no está garantizada, a menos que sean heurístcas aproximadas.

[2] http://www.cplusplus.com/reference/algorithm/sort/
[3] http://www.cplusplus.com/reference/vector/vector/erase/

Los algoritmos descriptos en esta sección no son aproximados, con lo cual los resultados podrían ser, potencialmente, tan malos como querramos si elegimos una instancia adecuada. Además, no garantizan una cota inferior, por lo que buscaremos ejemplos que resulten en un MCS sin aristas. Sean los siguientes grafos:



Grafo 1



Grafo 2

Los grafos tienen igual cantidad de nodos y aristas, y los nodos 0 y 5 tienen grado 4 en ambos grafos, y pertenecen a componentes conexas distintas. Además, los restantes nodos están conectados a 0 ó a 5.

El siguiente mapeo fue generado por el algoritmo exacto, y resulta en un MCS de tamaño máximo, es decir isomorfo a G_1 y G_2:

Nodo	Mapeo de G_1	Mapeo de G_2
0	0	0
1	6	1
2	7	2
3	8	3
4	9	4
5	5	5
6	1	6
7	2	7
8	3	8
9	4	9

Si ejecutamos los algoritmos *greedy* para la misma instancia, obtendremos un mapeo uno a uno (el nodo i de G_1 se mapea al nodo i de G_2) en ambos casos. Este mapeo genera un subgrafo común sin aristas. Lo que ocurre para $greedy_1$ es lo siguiente:

Se busca el nodo de mayor grado en cada grafo. Se elige para ambos grafos el nodo 5 puesto que en caso de empate, el algoritmo se queda con el nodo de mayor número, y se los mapea al nodo 0 del subgrafo común. Luego, se eligen los nodos 0 y se los mapea al nodo 1 del subgrafo común. Dado que todos los nodos restantes tienen grado 1, entonces se seleccionan los nodos en orden descendiente desde el 9 y se los mapea incrementalmente. Aquí, dado que en G_1 el 0 está conectado a $[6, 9]$, mientras que en G_2 no está conectados a ellos, sino a $[1, 4]$, ninguna arista con un extremo en 0 pertenecerá al subgrafo común. Ocurre lo mismo en el nodo 5 con los rangos invertidos, resultado entonces en que ninguna arista pertenecerá al subgrafo resultante.

Para el caso de $greedy_2$, ocurrirá lo mismo, dado que la remoción de nodos de los grafos sólo llevará a que todos los nodos distintos de 0 y 5 tengán grado 0 en vez de 1.

El ejemplo puede ser extendido al tamaño que se quiera, simplemente agregando de a pares de nodos en cada grafo, por ejemplo, agregando los nodos a y b, y las aristas $(0, a)$ y $(5, b)$ en G_1 y $(0, b)$ y $(5, a)$ en G_2. Para cualquier número de nodos agregados, el subgrafo común seguirá sin contener ninguna arista.

Modificar el algoritmo para que esto deje de ocurrir no es simple, sin cambiarlo significativamente. Se puede cambiar la manera en que se desempata entre dos nodos de igual grado. Una forma sería quedarnos con el de menor número en vez del mayor, pero esto no cambiaría los resultados en el ejemplo anterior. También se podría hacer que para un grafo en caso de desempate se elija al mayor y en el otro al menor; en tal caso el ejemplo anterior dejaría de ser malo, pero modificándolo un poco

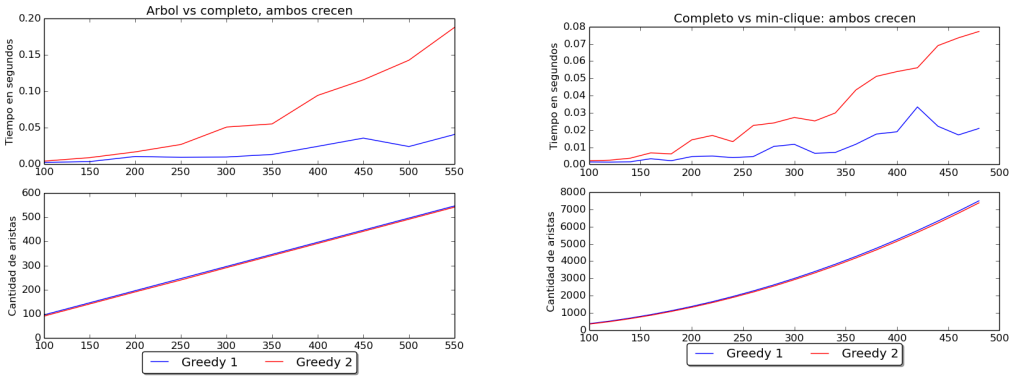
se puede llegar a un resultado tan malo como se quiera. También se puede aleatorizar la selección de nodos en caso de empate, pero esto implicaría sacrificar el comportamiento determinístico de la heurística, que puede ser especialmente malo si se la quiere integrar a otro algoritmo.

4.4. Experimentación

En esta sección vamos a detallar la experimentación que se llevó a cabo para el algoritmo de backtracking realizado para resolver el problema de *MCS*. Variaremos entre correr el algoritmo con la poda y sin ella. Tomamos un grupo de instancias conformado por un árbol binario completo, un grafo completo, y un grafo con una clique mínima de un tamaño especificado (llamaremos a este tipo de instancia *min-clique*), por lo general será $N/4$ donde N es el número de nodos.

También observaremos las diferencias en la solución obtenida para el problema de MCS devuelta por las distintas heurísticas golosas.

En las siguientes figuras, el cartel con las leyendas de las curvas oculta el rótulo del eje x, a menos que se especifique lo contrario, este sería N.



(a) Arbol vs completo

(b) Completo vs min-clique

Figura 14: Experimento 4.1

En el experimento 4.1 se llama al algoritmo con un árbol y un grafo completo (subfigura izquierda), donde estos crecen en tamaño simultaneamente a lo largo de las instancias, realizamos esto mismo con un grafo completo y un grafo *min-clique* (subfigura derecha). Cuando tomamos un árbol y un grafo completo, decidimos generar el árbol para que tenga una cantidad de nodos mayor a a la del grafo completo (apróximadamente el doble), de esta manera nos aseguramos que el árbol no entre en el grafo completo.

Tenemos por un lado que en ambos casos los resultados para el problema de MCS son idénticos. Esto es así puesto que ambos algoritmos se enfocan en encontrar el nodo de mayor grado para el grafo actual, donde la definición de grafo actual varía dependiendo del algoritmo. Sin embargo, en este caso, dado que el árbol es completo y el otro grafo es un grafo completo, el proceso resultará en un mapeo muy similar.

Respecto a los tiempos de ejecución, vemos que el algoritmo usando el segundo greedy tarda más que usando el primero. Esto se debe a que, como se explica en la sección anterior, la complejidad del segundo es mayor que la del primero. En este caso, entonces optaríamos por usar el primer greedy ya que los tiempos de ejecución son menores.

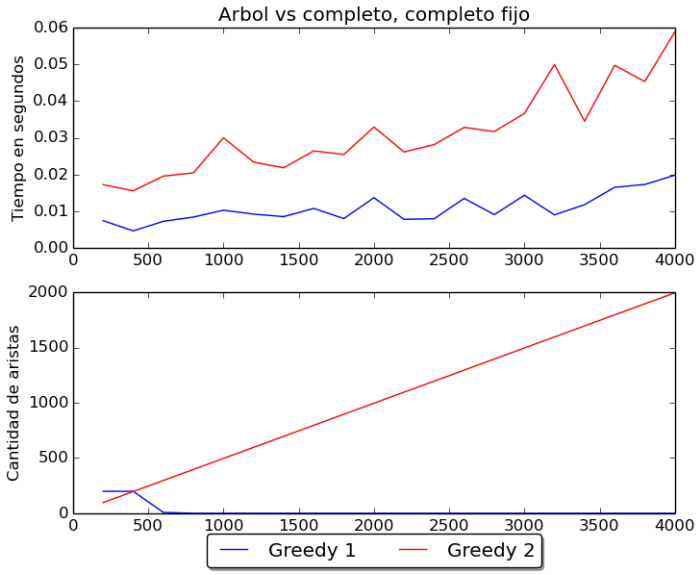


Figura 15: Experimento 4.2

Para este experimento tomamos las mismas instancias que en el anterior, pero decidimos mantener fijo el tamaño del grafo completo y observar los tiempos de ejecución respecto del aumento al tamaño del árbol.

Respecto a los tiempos de ejecución, otra vez ocurre que el greedy 1 finaliza antes que el dos. Sin embargo, si uno mira los resultados del MCS, ya hay un cambio interesante. Por un lado el greedy 1 para la mayoría de las instancias, no consigue incorporar ninguna arista al subgrafo común. Ya vimos en la sección de optimalidad que esto era algo posible y que no significa un error de programación en el algoritmo. Por otro lado, el greedy 2 logra obtener resultados que, comparado a su contraparte, son mucho mejores.

Para este caso, optaríamos por usar el greedy 2, ya que consigue resultados aunque tarde más en ejecutar.

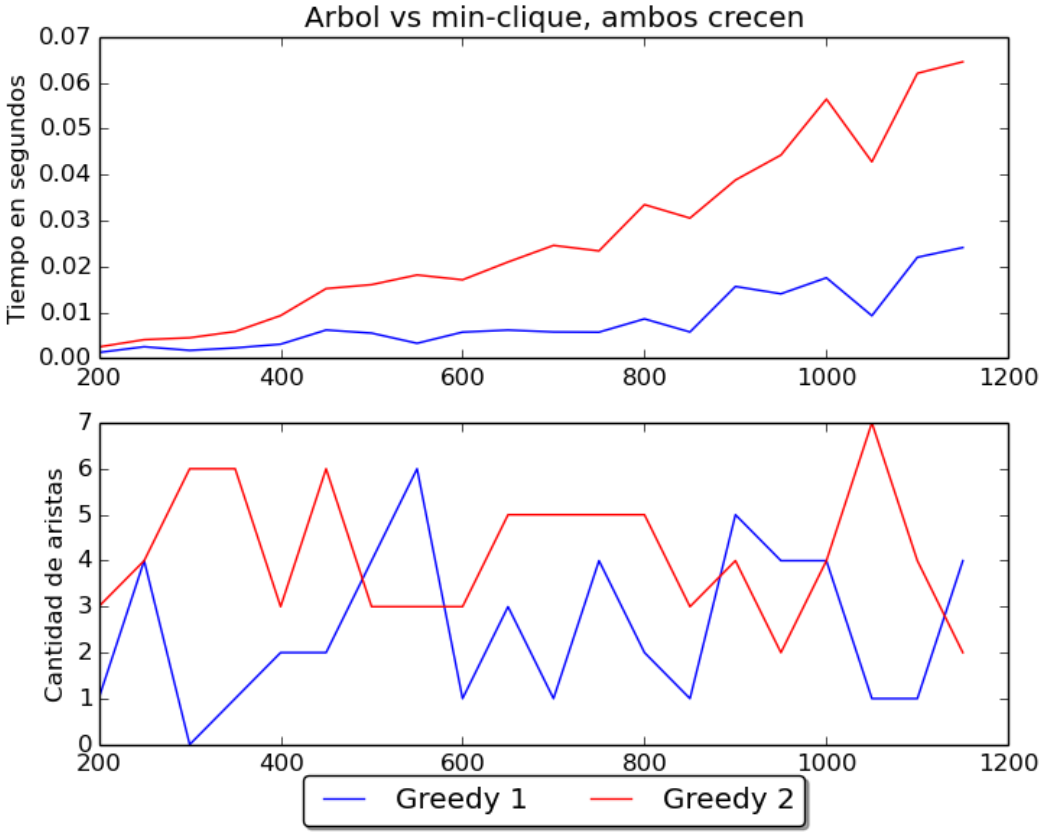


Figura 16: Experimento 4.3

En el experimento 4.3 comparamos los tiempos de ejecución para los tipos de instancias árbol y *min-clique* donde ambos crecen en tamaño de una instancia a la otra.

En cuanto a los resultados, vemos que no podemos discernir un patrón para ninguno de los casos. Las cantidades de aristas en común encontradas varían en un rango de uno a cinco a pesar de que el tamaño de los grafos aumenta. Esto sugiere que ambos greedys no son buenas heurísticas para este caso en particular y no hay manera de optar por uno o por el otro pues ambos son igual de erráticos. Por eso, de tener que elegir uno, nos quedaríamos con el greedy 1 simplemente por los menores tiempos de ejecución

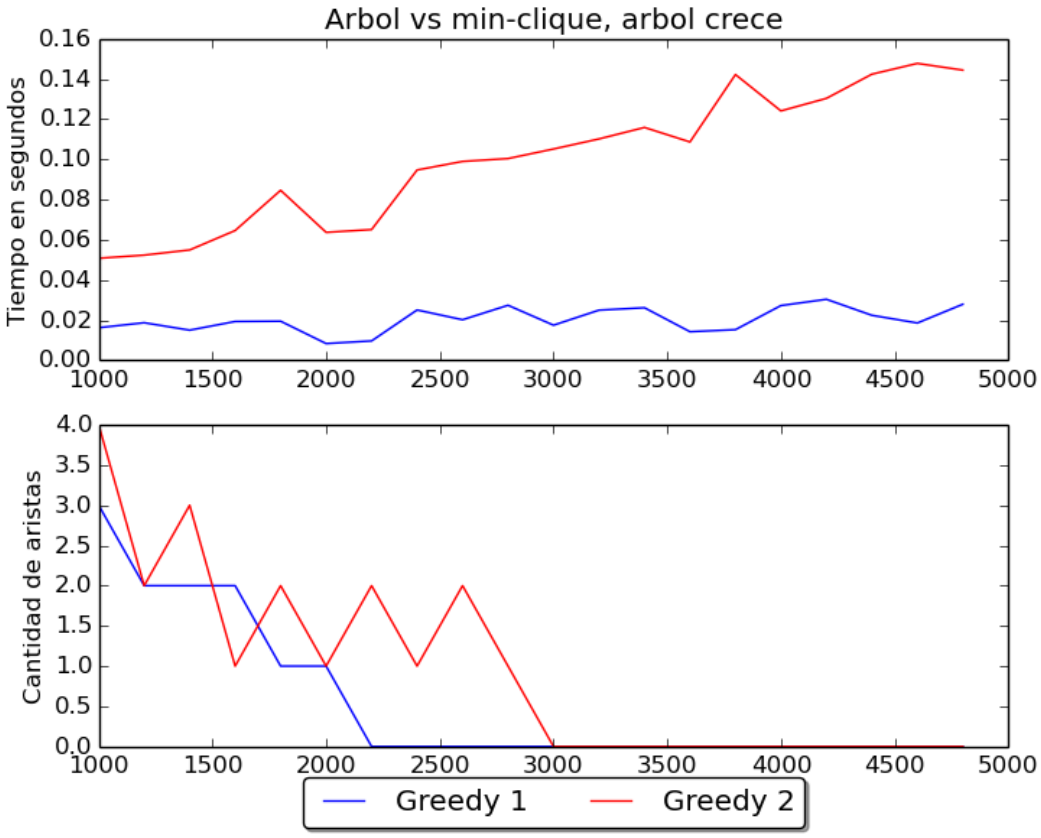


Figura 17: Experimento 4.4

Para el experimento 4.4 ejecutamos los algoritmos tomando las mismas instancias del anterior, pero mantuvimos fijo el *min-clique* mientras aumentamos el tamaño del árbol.

Para los tiempos de ejecución, los resultados son los esperados: el greedy 1 tarda menos que el dos puesto que su complejidad es menor y la cantidad de iteraciones no depende de un mejor o peor caso.

En cuanto a los resultados del MCS, se puede ver que después de un N_0 estos se vuelven siempre 0. De manera similar al experimento 4.2, al incrementar el tamaño del árbol mientras el tamaño del otro grafo queda fijo, los resultados se vuelven peores. Sin embargo, en este caso, ambos greedys demuestran este comportamiento. Nuevamente, optaríamos por el greedy 1 simplemente por su menor complejidad temporal.

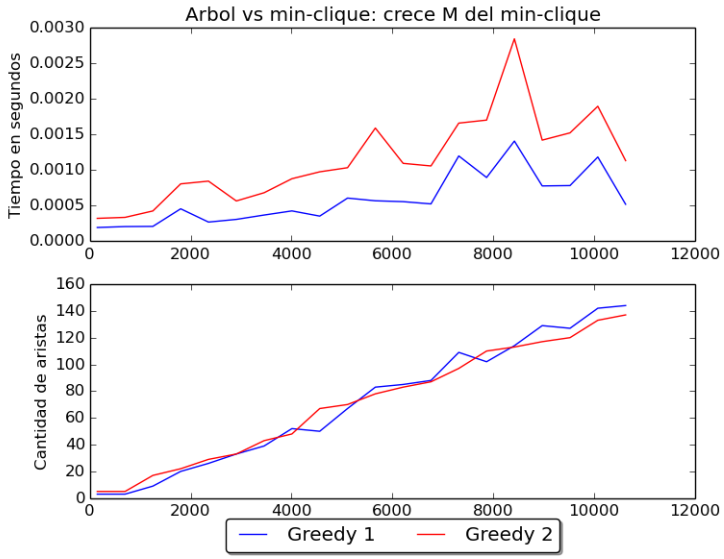


Figura 18: Experimento 4.5

En el experimento 4.5 tomamos las mismas instancias que en los dos últimos experimentos. En este experimento decidimos mantener fijos los números de nodos de ambas instancias y variar el número de aristas del grafo *min-clique*.

En este experimento, nos topamos con la siguiente dificultad: al establecer una cantidad de nodos base, esta no podía ser muy grande porque si lo fuese el tamaño de las instancias crecería inmanejablemente dado que la cantidad de aristas M es hasta cuadrática respecto de la cantidad de nodos. Entonces, optamos por una cantidad de nodos que permitiera generar las instancias en un tiempo razonable, aunque los tiempos de ejecución hayan resultado ser muy chicos.

En cuanto a los resultados del MCS, no se ve mucha diferencia, mas allá de que hacia el final el primer greedy es mejor al segundo. En cuanto a los tiempos de ejecución, vemos por el gráfico que nuevamente el primer greedy tarda menos en finalizar la ejecución. Sin embargo, como ya dijimos los tiempos son muy chicos y por lo tanto las diferencias los son aún más.

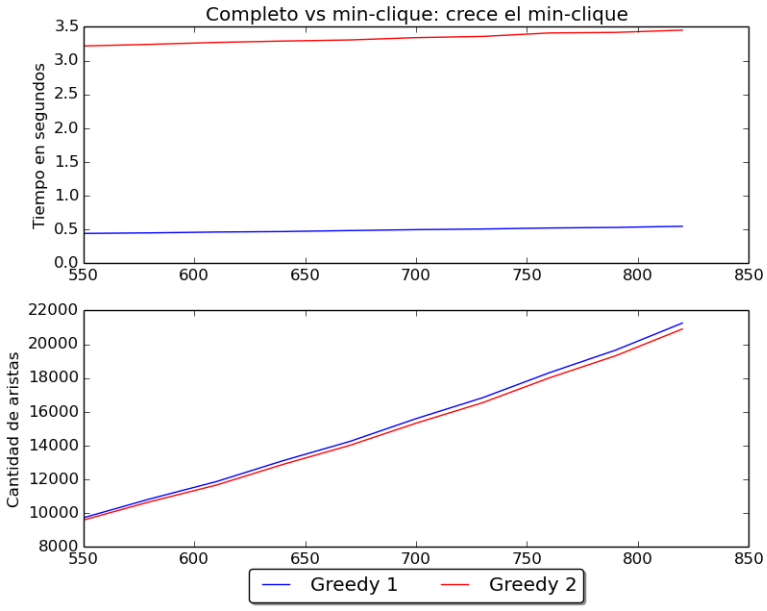


Figura 19: Experimento 4.6

En este experimento se tomaron instancias de *min-clique* y de grafo completo. El grafo completo se mantiene fijo mientras que el *min-clique* aumenta en tamaño.

En cuanto a los tiempos de ejecución, observamos que parecen ser constantes para el conjunto de instancias dado. Sin embargo, hay un incremento en estos de una instancia a la siguiente, que indica que no es constante. En particular, el greedy 1 se reduce a $O(NlogN)$ y el greedy 2 a $O(N + M)$ donde N y M son la cantidad de nodos y aristas del grafo que variamos.

En la figura, vemos que ambas heurísticas producen resultados similares, con el greedy 1 teniendo una leve ventaja. Por esta razón, optamos por el greedy 1 al este tener un mejores resultados en menores tiempos.

Por lo general, el primer greedy resultó en mejores resultados para el problema de MCS con la excepción de lo visto en el experimento 4.2. Además, los tiempos de ejecución fueron menores a los del greedy 2 en todos los casos, dado que tiene una complejidad temporal menor. Es por esto que clasificando al experimento 4.2 como un caso atípico es que elegimos al greedy 1 como una mejor heurística.

5. Ejercicio 5

5.1. Algoritmo

En esta sección se describe la implementación de una heurística de búsqueda local para el problema de MCS. Lo principal para esta heurística es la vecindad definida. Ésta establece qué soluciones podrán ser alcanzadas por el algoritmo dada una solución inicial. Nuestras soluciones consisten en dos arreglos, donde cada uno es el mapeo de un grafo. Los algoritmos asumirán la misma precondición que en la sección anterior, $|V_1| \geq |V_2|$.

La primera vecindad que elegimos consiste en utilizar swaps para llegar de una solución a otra. Un swap consiste en intercambiar los valores de dos posiciones en un arreglo. En particular, definiremos que una solución $A = (A_1, A_2)$ es vecina de otra $B = (B_1, B_2)$ si y sólo si se puede llegar de A_1 a B_1 con un único swap. Esto quiere decir que, luego de la llamada al greedy, el mapeo de G_2 permanecerá intacto. Esta vecindad se ayuda fuertemente de la precondición, puesto que sabemos que en el mapeo incial de G_2 estarán ya todos los nodos posibles de este grafo, y para acceder a todas las soluciones (o permutaciones de ellas), alcanza con modificar el mapeo de G_1.

La segunda vecindad elegida se parece a la primera, con la excepción de que además de de hacer un swap en A_1, se hace un shift hacia la derecha en el mapeo de G_1.

El algoritmo para la primer vecindad es como sigue:

```
procedure busqLocal(grafo G1, grafo G2)
    arreglo resMax, maxMap1, maxMap2
    greedy(G1, G2, maxMap1, maxMap2)
    resMax ← aristasEnComun(G1, G2, maxMap1, maxMap2)
    valorMax ← size(resMax)
    valorMaxVecindadAnterior ← −1
    localMap1 ← maxMap1
    localMap2 ← maxMap2
    while valorMaxVecindadAnterior < valorMax do
        valorMaxVecindadAnterior ← valorMax
        for cada swap S1 de localMap1 do
            resLocal ← aristasEnComun(G1, G2, S1, S2)
            valorLocal ← size(resLocal)
            if valorLocal > valorMax then
                valorMax ← valorLocal
                resMax ← resLocal
                maxMap1 ← S1
                maxMap2 ← S2
            end if
        end for
        localMap1 ← maxMap1
        localMap2 ← maxMap2
    end while
    return resMax, maxMap1, maxMap2
end procedure
```

El algoritmo busca primero una solución inicial llamando a alguno de los *greedy*. Luego busca, para esta solución, el vecino que maximiza la cantidad de aristas del subgrafo común, y lo elige como la nueva solución inicial. El algoritmo iterará hasta que para una solución seleccionada como local, se cumpla que todos sus vecinos resulten resultado igual o peor, pues significa que se alcanzó un máximo local.

El algoritmo para la segunda vecindad es idéntico, con la excepción de que en vez de solamente el swap, se llama al shift de *localMap*1. Este llamado se hace previo al *for*.

5.2. Complejidad

Para la primera vecindad, la heurística se divide en dos partes. Primero, el llamado a algún *greedy* y la configuración de las estructuras de datos necesarias. La complejidad depende de qué *greedy* se

utilice, y el resto de las operaciones es a lo sumo lineal en el tamaño de los grafos (nodos y aristas).

Segundo, se tiene el ciclo principal. Veamos cuál es el costo de éste. La cantidad de swaps posibles para el mapeo es $\sum_{i=1}^{N}(N-1)-i$, dado que, para no repetir swaps, un par de índices sólo se intercambiarán una vez. Esto resultá en un número de orden cuadrático respecto a N. En particular, sólo se utilizan permutaciones del mapeo de G_1, entonces se realizan $O(|V_1|^2)$ iteraciones. Para cada solución vecina, se llama a $aristasEnComun$, que tiene complejidad $O(|V_1|+|V_2|+|E_2|)$, y se copian, de cumplirse la guarda, los mapeos, de costo lineal en la cantidad de nodos totales. Entonces, el ciclo en total, costará $O(|V_2|^2(|V_1|+|V_2|+|E_2|)) = O(|V_2|^2(|V_1|+|E_2|))$.

Para la segunda vecindad, la complejidad del ciclo exterior es idéntica, con la excepción de que ésta incluye el llamado al shift en cada iteración. Éste se realiza sobre $localMap_1$ en tiempo lineal. Entonces, en vez de costar $O(|V_2|^2(|V_1|+|E_2|))$, costará $O(|V_1|^2(|V_1|+|E_2|)+|V_1|)$, que resulta en una complejidad final idéntica a la de la primera vecindad.

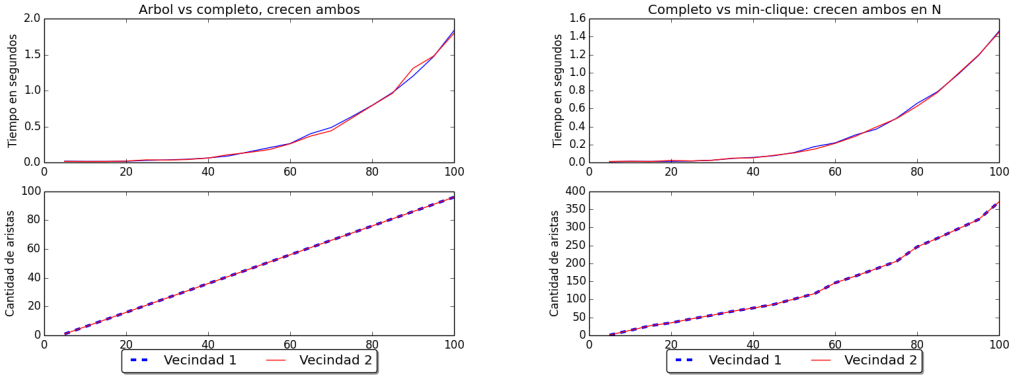
5.3. Experimentación

En esta sección vamos a detallar la experimentación que se llevó a cabo para el algoritmo de backtracking realizado para resolver el problema de *MCS*. Variaremos entre correr el algoritmo con la poda y sin ella. Tomamos un grupo de instancias conformado por un árbol binario completo, un grafo completo, y un grafo con una clique mínima de un tamaño especificado (llamaremos a este tipo de instancia *min-clique*), por lo general será $N/4$ donde N es el número de nodos.

Vamos a observar el comportamiento de tanto los tiempos de ejecución como los resultados para el problema de MCS para las variaciones del algoritmo. Estas dependen de las vecindades elegidas. Nos referiremos a las variaciones como vecindad 1 y vecindad 2.

Para estos experimentos, la solución inicial fue encontrada usando el greedy 1 dado que, como concluímos en la experimentación del ejericio 4, este era mejor.

En las siguientes figuras, el cartel con las leyendas de las curvas oculta el rótulo del eje x, a menos que se especifique lo contrario, este sería N.



(a) Arbol vs completo (b) Completo vs min-clique

Figura 20: Experimento 5.1

En el experimento 5.1 se llama al algoritmo con un árbol y un grafo completo (subfigura izquierda), donde estos crecen en tamaño simultaneamente a lo largo de las instancias, realizamos esto mismo con un grafo completo y un grafo *min-clique* (subfigura derecha). Cuando tomamos un árbol y un grafo completo, decidimos generar el árbol para que tenga una cantidad de nodos mayor a a la del grafo completo (apróximadamente el doble), de esta manera nos aseguramos que el árbol no entre en el grafo completo.

Los resultados obtenidos en ambos casos resultan idénticos. Esto se debe a que, dado que en ambos grafos los nodos se parecen mucho entre sí, es decir tienen grados parecidos, hacer un *shift* en los mapeos no resulta en cambios significativos de subgrafo común.

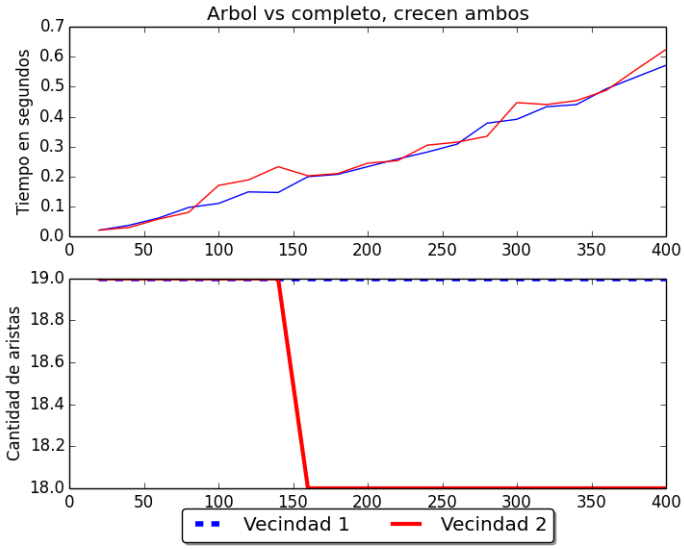


Figura 21: Experimento 5.2

Para este experimento tomamos las mismas instancias que en el anterior, pero decidimos mantener fijo el tamaño del grafo completo y observar los tiempos de ejecución respecto del aumento al tamaño del árbol.

En este caso, vemos que los tiempos nuevamente son muy similares. En cuanto a los resultados, vemos que difieren por solamente una unidad a partir de un N_0. Otra vez esto se debe a lo comentado en el experimento anterior. En este caso, podemos optar por la vecindad 1 ya que para todo N tiene el subgrafo común tiene una cantidad de aristas mayor o igual a la del subgrafo común encontrado por la segunda vecindad.

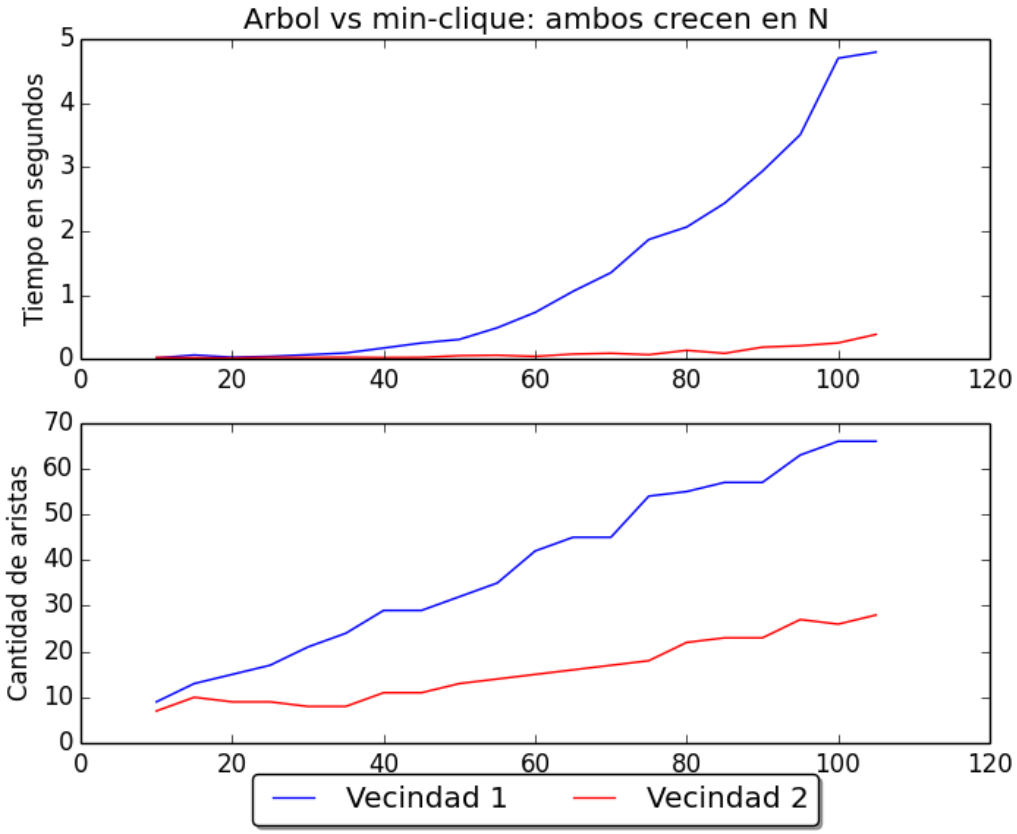


Figura 22: Experimento 5.3

En el experimento 5.3 comparamos los tiempos de ejecución para los tipos de instancias árbol y *min-clique* donde ambos crecen en tamaño de una instancia a la otra.

Podemos observar la primer discrepancia significativa en la calidad de las respuestas obtenidas para el MCS. Vemos que la vecindad 1 nos provee con resultados cada vez mejores a lo largo de las instancias respecto a los de la segunda vecindad.

En cuanto a los tiempos de ejecución, vemos que la vecindad 1 tarda más a medida que sus soluciones para el problema difieren de la vecindad 2. Esto se debe a, dada la naturaleza de la busqueda local, mientras la solución pueda mejorar, el algoritmo seguirá iterando, resultando en un mayor tiempo de ejecución.

Dada que la diferencia en la calidad de la solución es notable y parece incrementar al incrementar el tamaño de las intancias, optamos por a vecindad 1 como nuestra variación del algoritmo preferida para este caso.

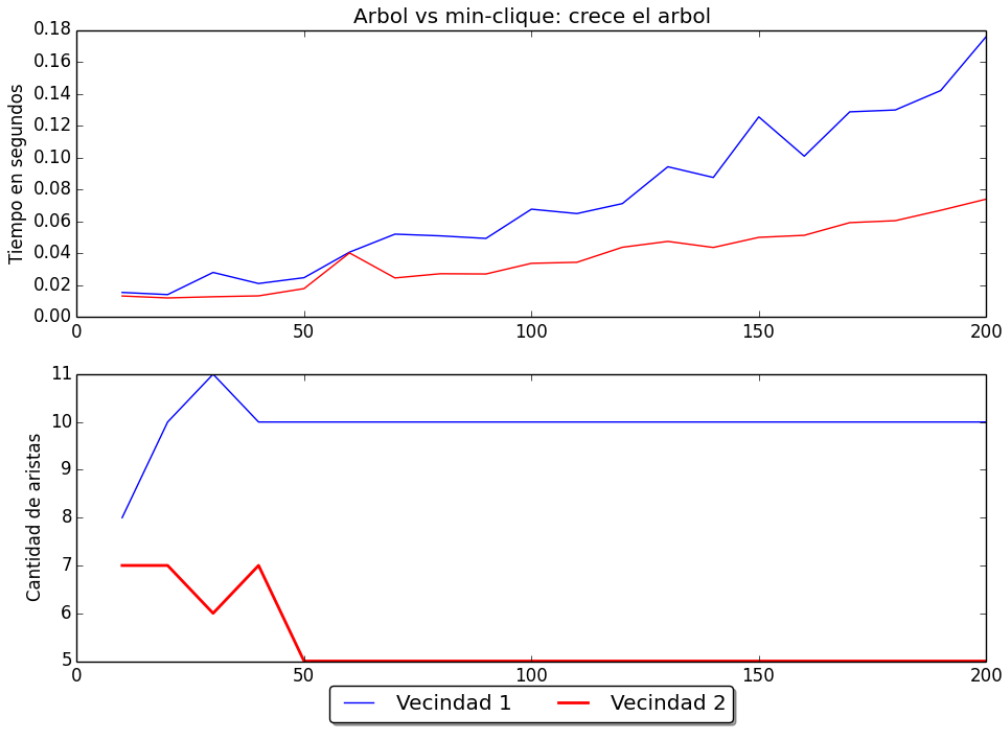


Figura 23: Experimento 5.4

Para el experimento 5.4 ejecutamos los algoritmos tomando las mismas instancias del anterior, pero mantuvimos fijo el *min-clique* mientras aumentamos el tamaño del árbol.

En este experimento, vemos que los resultados para MCS dados por ambas vecindades se tornan constantes a partir de un N_0. Esto sugiere, comparando la cantidad de aristas existentes, que la heurística de busqueda local en sí está fallando en algún punto. Posiblemente el greedy usado resulta en una solución inicial cerca de un máximo local mucho menor al máximo global al cual queremos llegar idealmente.

Respecto a los tiempos de ejecución vemos nuevamente que la vecindad 1 tarda más en ejecutar y esto se debe a que realiza más iteraciones mediante las cuales consigue mejores resultados en comparación a la vecindad 2. Por estas razones, optamos nuevamente por la vecindad 1.

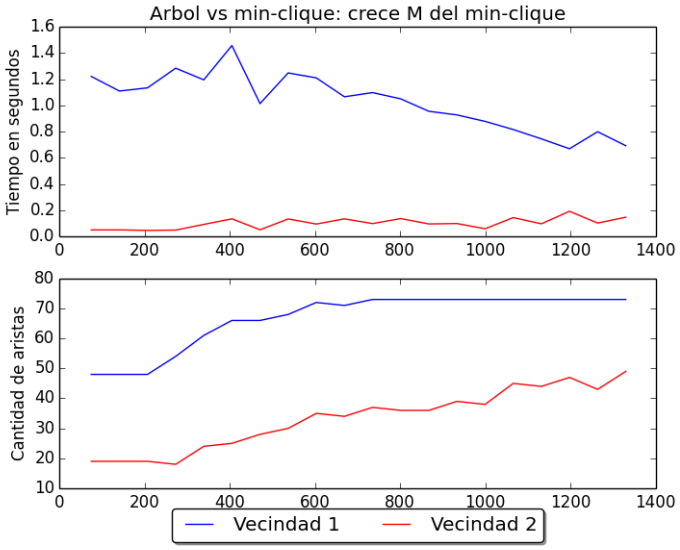


Figura 24: Experimento 5.5

En el experimento 5.5 tomamos las mismas instancias que en los dos últimos experimentos. En este experimento decidimos mantener fijos los números de nodos de ambas instancias y variar el número de aristas del grafo *min-clique*.

Sobre los resultados, vemos que la vecindad 1 resulta en mejor calidad de solución problema que la vecindad 2. Es interesante notar, que los tiempos de ejecución de la vecindad 1, comienzan a caer cuando la cantidad de aristas de la solución se estanca en valores cercanos a 70. Esto significa que el algoritmo realiza una menor cantidad de iteraciones para alcanzar una respuesta de similar calidad, lo cual ocurre porque el greedy nos provee con una solución inicial más cercana a un máximo local que genera resultados en el rango mencionado.

Dados estos resultados, optamos nuevamente por la vecindad 1.

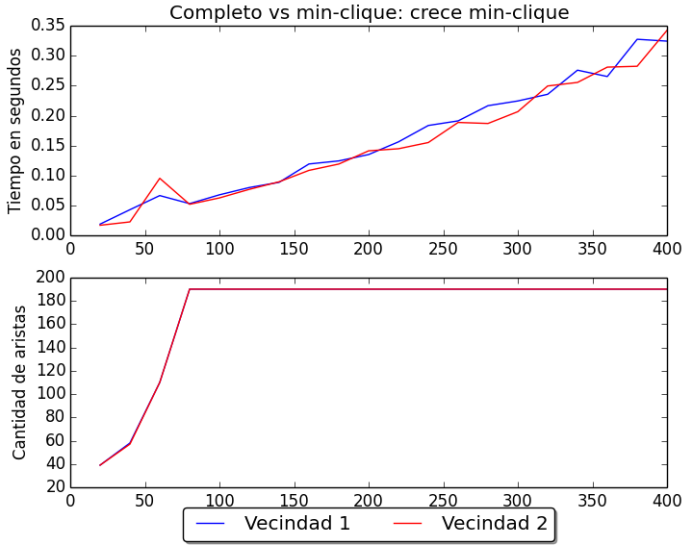


Figura 25: Experimento 5.6

En este experimento se tomaron instancias de *min-clique* y de grafo completo. El grafo completo se mantiene fijo mientras que el *min-clique* aumenta en tamaño.

Tanto en tiempo como en calidad de soluciones vemos que las vecindades nos devuelven resultados muy parecidos lo cual nos imposibilita optar por uno de ellos.

Para entender mejor la curva que describe la cantidad de aristas de la solución es necesario considerar el tipo de instancias usado. Se está manteniendo un grafo completo de tamaño fijo e incrementando un grafo que tiene una clique mínima de tamaño $N/4$ donde N es la cantidad de nodos de este grafo. Entonces, eventualmente el tamaño de la clique mínima sobrepasará a la del grafo completo y la solución óptima será el subgrafo común entre el grafo completo y un subgrafo de la clique mínima.

En el gráfico, se puede ver que mientras esto no pasa, los algoritmos irán generando mejores respuestas progresivamente hasta alcanzar este techo. Esto significa que la heurística de busqueda local funciona correctamente para este tipo de problemas.

En todos los casos optamos por la vecindad 1, puesto que conseguía al menos una solución de una calidad igual a la de la vecindad 2. Que los tiempos de ejecución sean mayores en algunos casos solo significa que el algoritmo realizó más iteraciones, lo cual en general significa que se está obteniendo un mejor máximo local que el que la vecindad 2 podría alcanzar con a lo sumo ese número de iteraciones.

6. Ejercicio 6

6.1. Algoritmo

En esta sección se describe la implementación de una heurística de Tabu Search para el problema de MCS. A diferencia de la búsqueda local, el algoritmo no debe terminar una vez que se alcanza un máximo local, sino que, mediante el mantenimiento de una lista tabú, se permita escapar del máximo y posiblemente alcanzar nuevos, idealmente mejores que los anteriores. Lo principal para esta metaheurística es defiinir qué búsqueda local se usará, criterios para definir que soluciones son tabú, y criterios de parada.

Los algoritmos asumirán la misma precondición que en la sección anterior, $|V_1| \geq |V_2|$. El algoritmo para la primer vecindad es como sigue:

---

procedure $tabuSearch_1$(**grafo** G_1, **grafo** G_2, **int** $maxTabuSize$)
  arreglo $resMax$, $maxMap_1$, $maxMap_2$, $tabu$
  $greedy(G_1, G_2, maxMap_1, maxMap_2)$
  $resMax \leftarrow aristasEnComun(G_1, G_2, maxMap_1, maxMap_2)$
  $valorMax \leftarrow size(resMax)$
  $valorMaxVecindadAnterior \leftarrow -1$
  $localMap_1 \leftarrow maxMap_1$
  $localMap_2 \leftarrow maxMap_2$
  while no se cumpla el criterio de parada **do**
    arreglo $maxMapLocal_1$, $maxMapLocal_2$, $maxResLocal$
    $maxValorLocal \leftarrow -1$
    for cada swap S_1 de $localMap_1$ **do**
      $resLocal \leftarrow aristasEnComun(G_1, G_2, S_1, S_2)$
      $valorLocal \leftarrow size(resLocal)$
      if $valorLocal > maxValorLocal \wedge (S_1, S_2) \notin tabu$ **then**
        $maxValorLocal \leftarrow valorLocal$
        $maxResLocal \leftarrow resLocal$
        $maxMapLocal_1 \leftarrow S_1$
        $maxMapLocal_2 \leftarrow S_2$
      end if
    end for
    if $maxValorLocal > valorMax$ **then**
      $maxMap_1 \leftarrow maxMapLocal_1$
      $maxMap_2 \leftarrow maxMapLocal_2$
      $valorMax \leftarrow maxValorLocal$
      $resMax \leftarrow maxResLocal$
    end if
    if $size(tabu) = maxTabuSize$ **then**
      remuevo el elemento más viejo de $tabu$
    end if
    agrego $(localMap_1, localMap_2)$ a $tabu$
    $localMap_1 \leftarrow maxMapLocal_1$
    $localMap_2 \leftarrow maxMapLocal_2$
  end while
  return $resMax$, $maxMap_1$, $maxMap_2$
end procedure

El algoritmo utiliza la búsqueda local con la primera vecindad definida en la sección anterior. En general, se parece bastante a una búsqueda local, con ciertas excepciones. Primero, es necesario mantener una solución que sea la mejor alcanzada hasta el momento, y otra que sea la mejor pero restringida a la iteración de búsqueda local actual. Esto es porque en algún momento, el algoritmo buscará escapar de un máximo local, y necesitará saltar a una solución que no sea la mejor encontrada hasta el momento, con lo que hay que mantener una versión de cada una.

El algoritmo con la segunda vecindad es equivalente, con la expcepción de que previo al swap, se hace un shift hacia la derecha del arreglo $localMap_1$. Además, a diferencia de la búsqueda local,

debemos luego restaurar este arreglo a su estado original, dado que será necesario ingresarlo a la lista tabú, por lo que se hará un shift hacia la izquierda.

Además, se agrega la lógica de la lista tabú. La idea es que al alejarnos de un máximo local, logremos hacerlo, evitando volver a él. La lista tabú se mantiene copiada en dos estructuras diferentes. Una es un *set*. Éste es útil pues permite consultar si una solución es tabú en tiempo logarítmico respecto al tamaño de la lista (sin incluir el costo de las comparaciones entre soluciones, que tiene costo lineal en la cantidad de nodos de los grafos). Además, se mantiene un *queue*, de manera que encontrar la solución a remover se haga en tiempo constante.

Es necesario definir un criterio de parada, puesto que ahora el algoritmo no debe terminar en el primer máximo local que encuentre. Se eligieron inicialmente distintos criterios, enumerados a continuación:

1. La cantidad de iteraciones del ciclo externo realizadas supera un número previamente definido.
2. La cantidad de iteraciones del ciclo externo realizadas desde la última vez que se actualizo la máxima respuesta encontrada supera un número previamente definido.
3. La vecindad de la respuesta local es vacía (todos los vecinos son tabú).

El primer y segundo criterio son simples de implementar. Se guarda una variable que cuenta el número de iteraciones hechas, y se termina el ciclado cuando ésta supera un máximo definido. El primer criterio tiene la ventaja de que permite tener control sobre la cantidad exacta de veces que se iterará. Esto es beneficioso, por un lado, porque facilita el análisis de la metaheurística, y por otro porque nos asegura que el algoritmo terminará en un tiempo conocido. Sin embargo, pueden ocurrir dos cosas. Primero, podría pasar que el algoritmo esté demasiado lejos del máximo global, o que ya se lo haya alcanzado, y se estaría usando tiempo de cómputo de más. Por otro, si el algoritmo está bien encaminado hacia un máximo, bajo este criterio se podría terminar antes de alcanzarlo. El segundo criterio maneja esto mejor, dado que permite al algoritmo seguir si parece estar bien encaminado.

El tercer criterio apunta a terminar el algoritmo únicamente cuando ya no tengamos una solución vecina a la que movernos que no hayamos visitado antes. La implementación consiste en preguntar si se encontró algún vecino no tabú en la iteración previa, que se puede hacer en tiempo constante. Tal criterio crea un número de problemas. Primero, es necesario tener un tamaño de lista tabú suficientemente grande: ocurrió, por ejemplo, que dados dos grafos de 3 nodos cada uno, y un tamaño máximo de la lista de 16, el algoritmo ciclaba indefinidamente. Dado que en total hay 3!3!=36 soluciones posibles (hay algunas repetidas por ser permutaciones, pero esto es invisible para el algoritmo), el tamaño de la lista es un porcentaje importante del espacio total de soluciones. Si consideramos que éste espacio crece factorialmente, garantizar que el algoritmo terminará podrá ser muy costoso o complicado. Experimentando brevemente con el último criterio, vimos que puede tardar demasiado en cumplirse, especialmente cuando los grafos crecen en tamaño. Por eso decidimos no utilizarlo solo, si no que lo combinaremos con alguno de los criterios anteriores.

Ya notamos como una elección equivocada puede llevar a que el algoritmo no termine si se está utilizando el último criterio de parada especificado. Este problema no tiene tanto que ver con el criterio elegido, sino con el tamaño de la lista. Que el algoritmo no termine implica que se formó un ciclo en la sucesión de soluciones elegidas, del que no se puede escapar. Esto sólo puede ocurrir porque estamos permitiendo revisitar ciertas soluciones, que a la vez sólo ocurre porque éstas son desplazadas de la lista tabú en algún momento. Mantener una lista de tamaño igual al total de soluciones posibles no es práctico, porque se vuelve demasiado grande para grafos más grandes. Se puede optar, por un lado por elegir un tamaño fijo en función de la entrada, como $N_1 * N_2$, $(N_1 * N_2)^2$, etc. Por otro lado, se puede probar modificar el tamaño máximo dinámicamente. Por ejemplo, se puede mantener un conjunto con algunos de las soluciones alcanzadas, de forma que si se alcanza una de vuelta sabemos que se formó un ciclo, el cual hay que romper de alguna forma. La manera más simple de hacer es incrementar el tamaño máximo de la lista tabú, puede ser doblándolo. Si se revisita alguna de estas respuestas, se procede a doblar nuevamente el tamaño. Miremos mejor esta idea:

La técnica es simple de implementar, pero es necesario considerar todos los casos posibles. En primer lugar, identificamos todas las soluciones que son mejores que la solución de la que provienen. Esto es fácil de hacer, pero se pueden crear, por ejemplo, ciclos donde todas las soluciones den un subgrafo de igual número de aristas, donde ninguno de estos sea mejor que las demás (que se siguen revisitando porque el tamaño de la lista tabú no es suficiente). Para cubrir este caso, nos ayudamos de la estrategia utilizada en el segundo criterio de parada, de forma que si después de un número

de iteraciones no se encontró una mejoría, agregamos a la solución actual al nuevo conjunto, tal que cuando se la visite de nuevo, el tamaño máximo de la lista tabú se doble.

6.2. Experimentación

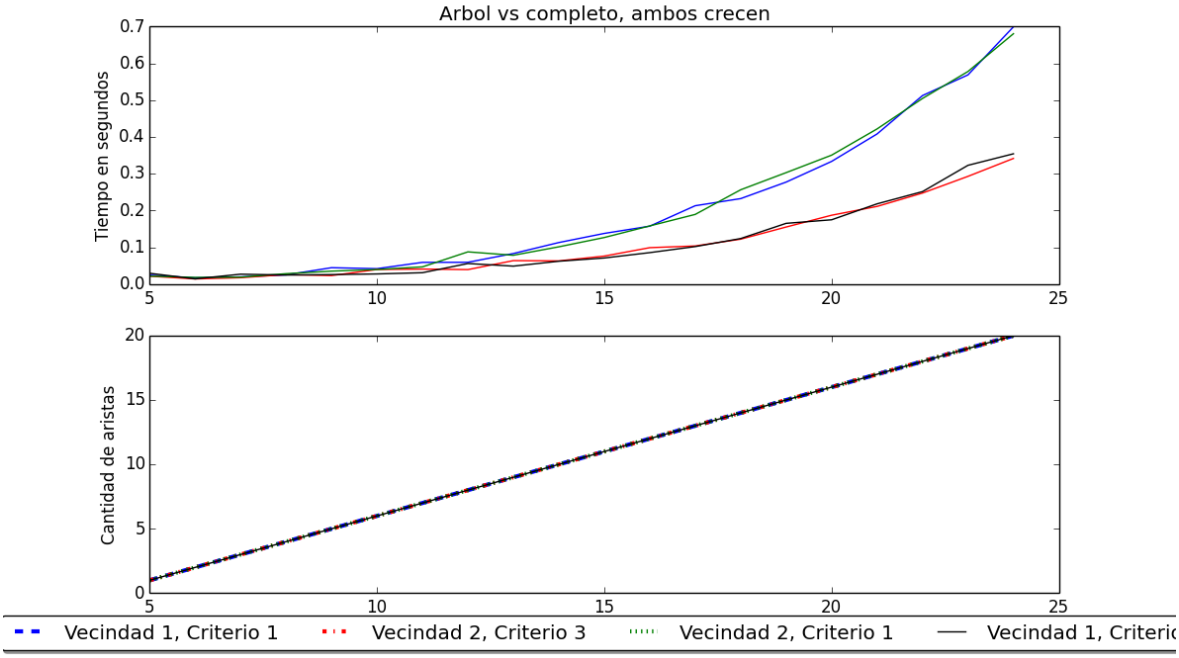
En esta sección vamos a detallar la experimentación que se llevó a cabo para el algoritmo de backtracking realizado para resolver el problema de *MCS*. Variaremos entre correr el algoritmo con la poda y sin ella. Tomamos un grupo de instancias conformado por un árbol binario completo, un grafo completo, y un grafo con una clique mínima de un tamaño especificado (llamaremos a este tipo de instancia *min-clique*), por lo general será $N/4$ donde N es el número de nodos.

Vamos a observar el comportamiento de tanto los tiempos de ejecución como los resultados para el problema de MCS para las variaciones del algoritmo. Estas dependen de las vecindades elegidas y de los criterios. Tendremos para vecindad 1 y vecindad 2, criterio 1 y criterio 3.

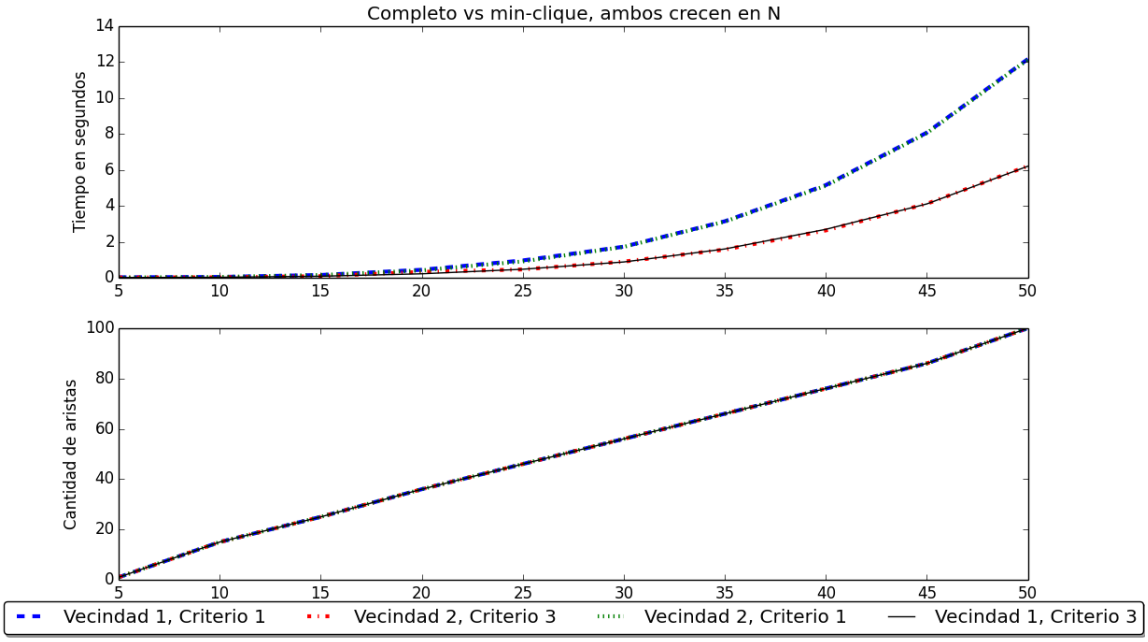
Para el tamaño de la lista tabu decicidimos utilizar únicamente la lista de tamaño dinámico con un tamaño inicial de $(N_1 N_2)^3$. Esto es porque por un lado encontramos que el tamaño inicial es lo suficientemente grande para la mayor;ia de los casos y que por otro el tamaño dinámico permite a la lista crecer si se encuentra repitiendo soluciones.

En todas las imagenes, la línea negra representa los resultados para la corrida del algoritmo con el criterio de parada 3, y vecindad 1.

En las siguientes figuras, el cartel con las leyendas de las curvas oculta el rótulo del eje x, a menos que se especifique lo contrario, este sería N.



(a) Arbol vs completo



(b) Completo vs min-clique

Figura 26: Experimento 6.1

En el experimento 6.1 se llama al algoritmo con un árbol y un grafo completo (subfigura izquierda), donde estos crecen en tamaño simultaneamente a lo largo de las instancias, realizamos esto mismo con un grafo completo y un grafo *min-clique* (subfigura derecha). Cuando tomamos un árbol y un grafo completo, decidimos generar el árbol para que tenga una cantidad de nodos mayor a a la del grafo completo (apróximadamente el doble), de esta manera nos aseguramos que el árbol no entre en el grafo completo.

En cuanto a la cantidad de aristas de la solución, el hecho de que sean idénticas para los cuatro casos indica que se alcanza un máximo local temprano en la ejecución y luego la busqueda tabú no

logra llegar a un nuevo máximo. Esta hipótesis está soportada por el hecho de que los algoritmos que utilizan el criterio 3 terminan antes, indicando que se hacen a lo sumo cinco soluciones mejores que las anteriores.

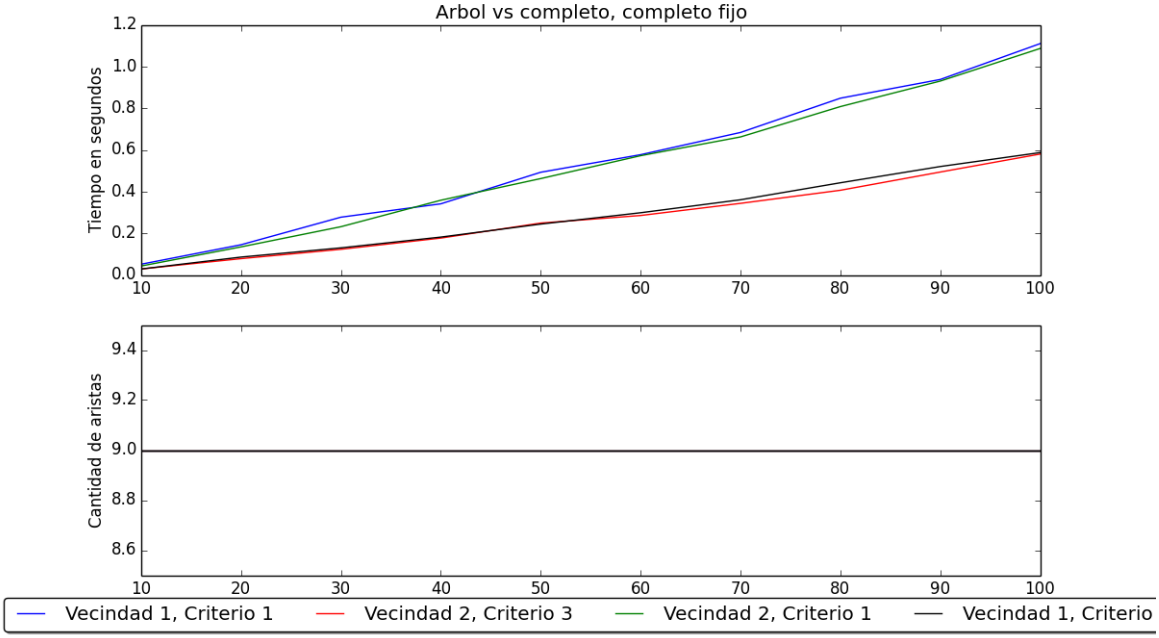


Figura 27: Experimento 6.2

Para este experimento tomamos las mismas instancias que en el anterior, pero decidimos mantener fijo el tamaño del grafo completo y observar los tiempos de ejecución respecto del aumento al tamaño del árbol.

En este experimento, nuevamente parece que se alcanza un máximo local temprano en la ejecución y luego no se encuentra una mejora para esta solución. Esta idea puede ser explicada por tanto los resultados constantes que alcanzan todos los algoritmos como los tiempos que tardan en ejecutarse (siguiendo la misma idea del experimento 6.1).

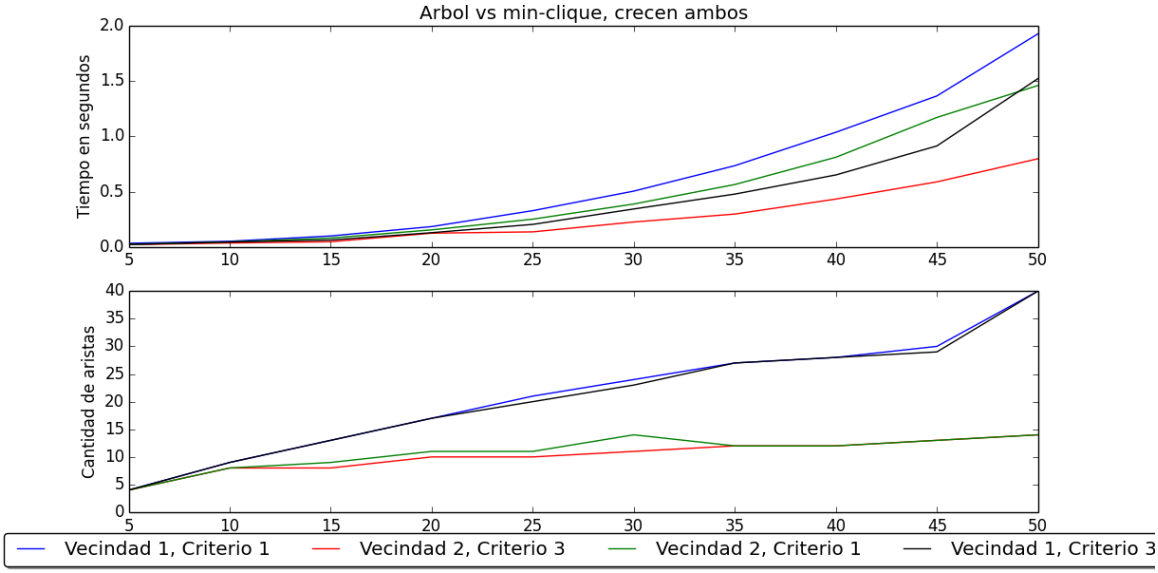


Figura 28: Experimento 6.3

En el experimento 6.3 comparamos los tiempos de ejecución para los tipos de instancias árbol y *min-clique* donde ambos crecen en tamaño de una instancia a la otra.

En cuanto a las soluciones para MCS vemos que las variaciones del algoritmo en las cuales se toma la vecindad 1 producen unas de mejor calidad. Esto es parecido a lo que ocurrió en el experimento 5.3 donde pudimos observar una discrepancia similar de acuerdo a las vecindades que usaba el algoritmo.

La diferencia en tiempos de ejecución entre las variaciones con la misma vecindad surgen debido a que, otra vez, el criterio 3 resulta en una cantidad de iteraciones totales menor al 1.

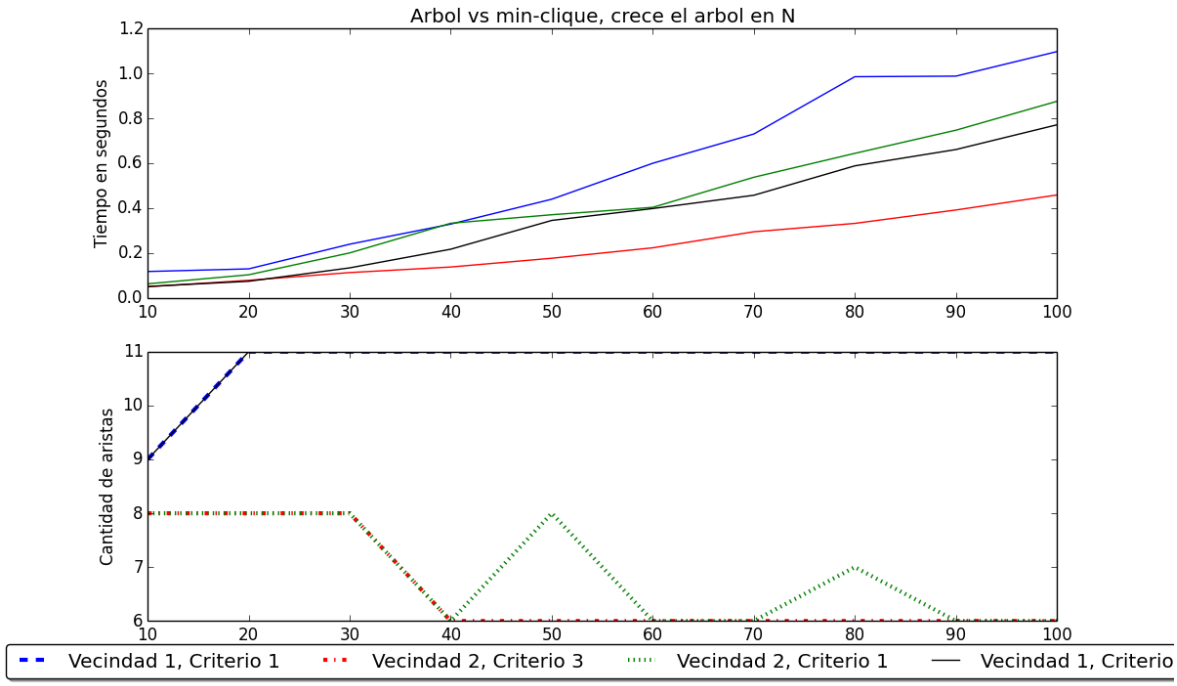


Figura 29: Experimento 6.4

Para el experimento 6.4 ejecutamos los algoritmos tomando las mismas instancias del anterior, pero mantuvimos fijo el *min-clique* mientras aumentamos el tamaño del árbol.

En este experimento, la vecindad 1 genera respuestas mejores que las de la vecindad 2 nuevamente. Dentro de la vecindad 1, ambos criterios generan las mismas respuestas mientras que dentro de la vecindad 2 el criterio de parada 1 permite algunas mejoras en ciertas instancias.

Para los tiempos de ejecución, que para vecindad 1 el criterio 3 termine antes que el criterio 1 sugiere que son menos de 5 la cantidad de veces que se mejora la solución. En particular es interesante notar que para la vecindad 2, hay casos donde ocurre una mejora luego de que el criterio 3 haya terminado la ejecución, con lo que en este caso este criterio podría beneficiarse de una mayor cantidad de iteraciones.

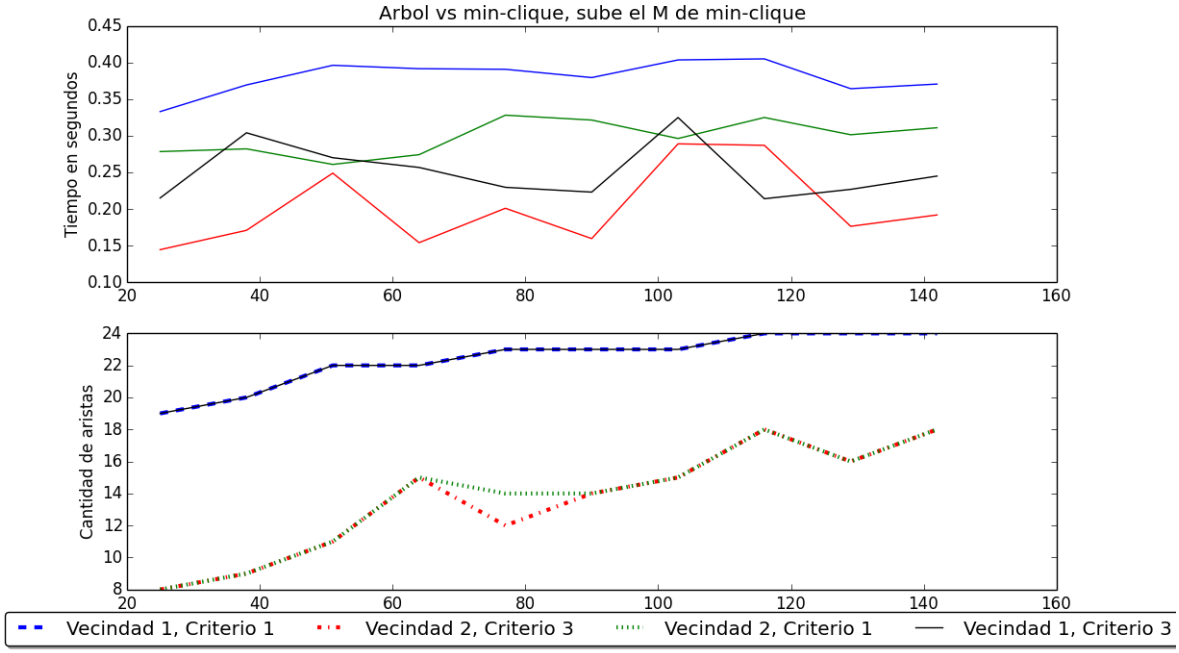


Figura 30: Experimento 6.5

En el experimento 6.5 tomamos las mismas instancias que en los dos últimos experimentos. En este experimento decidimos mantener fijos los números de nodos de ambas instancias y variar el número de aristas del grafo *min-clique*.

Sobre los resultados de este experimento, presentan muchas similitudes al anterior, por lo cual el análisis sobre el anterior vale también para este.

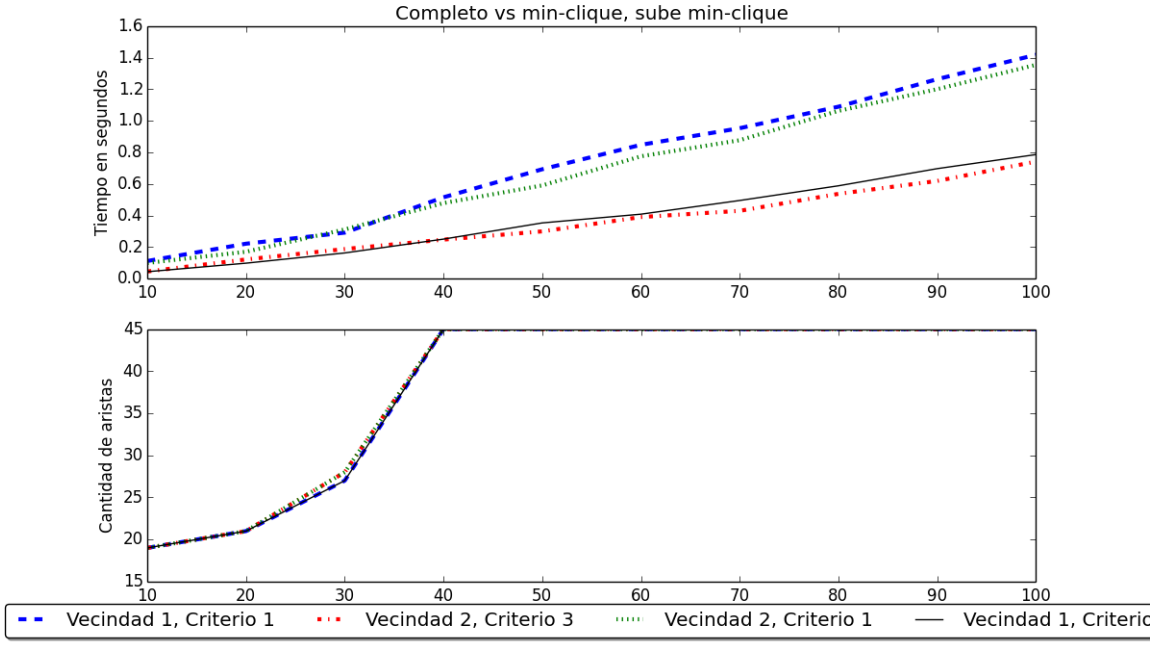


Figura 31: Experimento 6.6

En este experimento se tomaron instancias de *min-clique* y de grafo completo. El grafo completo se mantiene fijo mientras que el *min-clique* aumenta en tamaño.

En este caso, el patrón que sigue las respuestas es similar a la del experimento 5.6. Esto es lógico pues si una busqueda local puede alcanzar un máximo local que es en particular la solución óptima, un tabú search debería poder hacerlo también. En particular, es más interesante considerar los tiempos de ejecución. Nuevamente, vemos como se alcanza el máximo local temprano en la ejecución y no se lo mejora, que lleva a que el criterio de parada 3 se active antes que el 1. Esta es otra situación en la que esto ocurre, llevando a que el criterio 1 itere más veces que el criterio 3 sin lograr nada.

En todos los ejemplos, vimos como la vecindad 1 lleva a mejores resultados para el problema MCS que la vecindad 2. Además vimos que el criterio de parada 1, a pesar de tener más iteraciones en general, no logra mejorar la respuesta final respecto a la que se consigue con el criterio 3. Es decir, toma m;as tiempo para conseguir la misma respuesta. Por esta razón elegimos como nuestra variación óptima del algoritmo a la que implementa la vecindad 1 junto con el criterio de parada 3.

7. Ejercicio 7

Para esta sección, se tomó un nuevo conjunto de instancias para realizar una serie de experimentos sobre los algoritmos que implementan heurísticas. Esto es particularmente útil pues aunque pudimos definir claramente para las tres heurísticas una mejor variación del algoritmo, queda inconcluso la ventaja de utilizar tabú search por sobre busqueda local. Se busca en esta sección definir si existe tal ventaja, comparándolos a la vez con la heurística más rudimentaria (constructiva golosa).

El conjunto de instancias para esta experimentación consiste de tres tipos de grafos: grafos *path* que son caminos simple, grafos bipartitos y grafos que son caminos de estrellas, es decir, que son estrellas donde los nodos centrales forman un path, a estos grafos los llamamos *centipede*.

Para cada algoritmo, se utiliza la variación óptima concluida de todas las secciones anteriores de experimentación. Para la heurística golosa, el greedy 1, para la busqueda local, vecindad 1 y para el tabú search, tamaño de lista dinámico, vecindad 1, criterio de parada 3 con 10 iteraciones máximas sin mejoría (la aumentamos de 5 a 10 para poder ver si el tabú search efectivamente puede encontrar una mejora al máximo local).

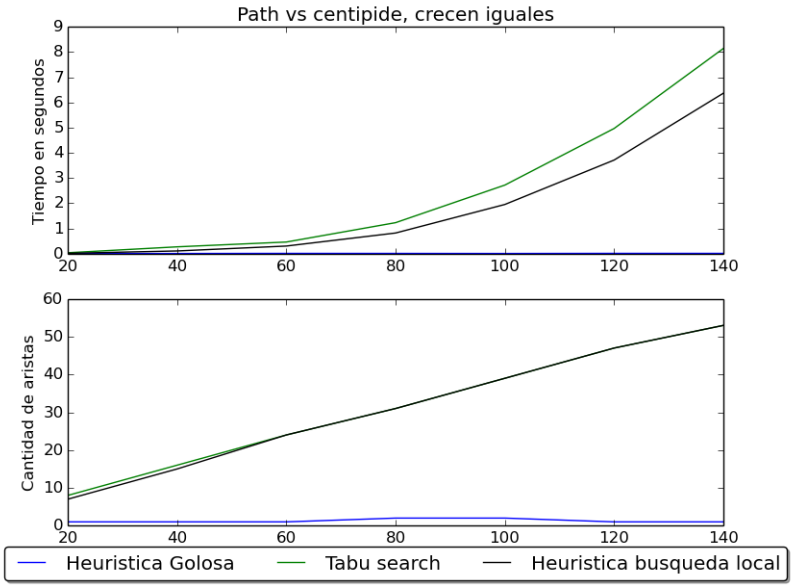


Figura 32: Experimento 7.1

Para el experimento 7.1 se tomó un grafo *path* y un grafo *centipede* con estrellas de 5 nodos y se les incrementó la cantidad de nodos a ambos a lo largo de las instancias.

En este primer experimento podemos observar como la busqueda local es tan buen como el tabú search. Esto quiere decir que el primer máximo local que se alcanza en el tabú search es el que termina como solución. Además como el tabú search realiza al menos diez iteraciones más que la heurística local, su tiempo de ejecución será mayor.

Se puede también confirmar que hacer busqueda local (y por ende, tabú search) es significativamente mejor que usar la heurística constructiva golosa dada la calidad de las respuestas obtenidas por esta.

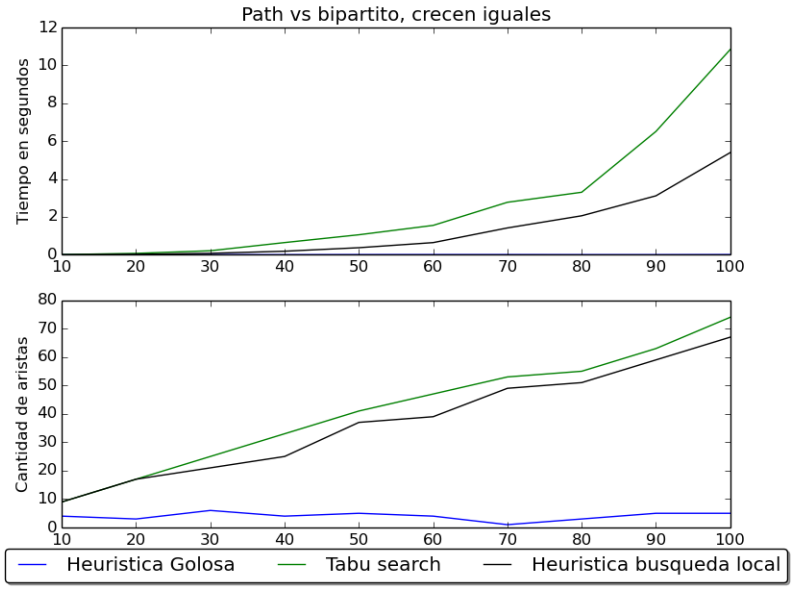


Figura 33: Experimento 7.2

Las instancias para este experimento consisten en un grafo *path* y uno bipartito los cuales aumentan su número de nodos a lo largo de las instancias. En este gráfico podemos observar como el tabú search produce mejores soluciones para el problema que la busqueda local dado que no se quedó con el primer máximo local encontrado. De esta manera vemos la utilidad de usar tabú search por encima de una heurística de busqueda local. Una vez más, vemos como la heurística golosa no es de utilidad como solución final puesto que aunque para las primeras instancias su solución se parezca a la de las demás heurísticas, estos se mantienen aproximademente constantes a lo largo del experimento mientras que las otras soluciones mejoran.

En cuanto a los tiempos, dado que tabú search encuentra una mejoría, evidentemente termina más tarde. Vemos también que la diferencia entre los tiempos de tabú search y la busqueda local son mayores a la vista en el experimento anterior. Es interesante notar como las diferencias en las respuestas para el problema obtenidas por las heurísticas causan una discrepancia mayor en el tiempo de ejecución.

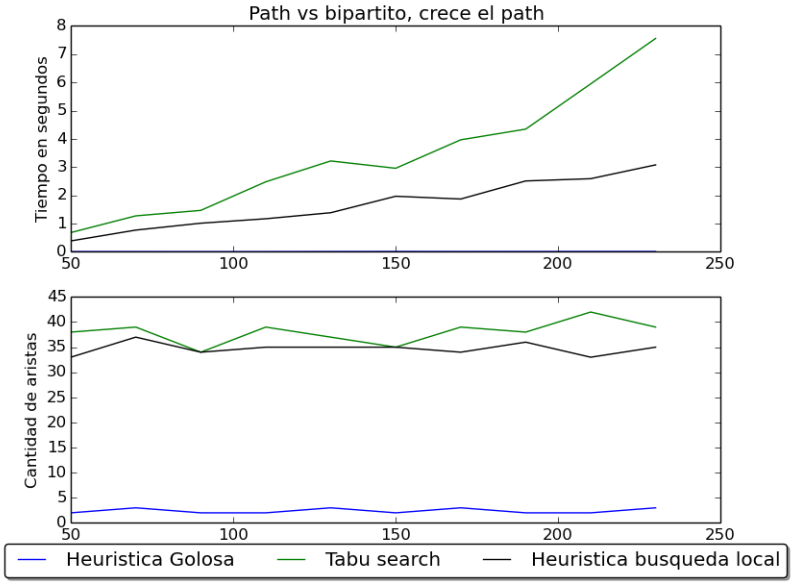


Figura 34: Experimento 7.3

En este experimento tomamos un grafo bipartito, un grafo *path* e incrementamos el número de nodos del grafo *path*. Este problema es equivalente, si el tamaño del *path* es mayor a la cantidad de aristas del bipartito, a encontrar el camino simple de máxima longitud en el grafo bipartito.

Vemos nuevamente la mejora que trae el tabú search y como eso refleja en los tiempos de ejecución como ya vimos en el experimento anterior.

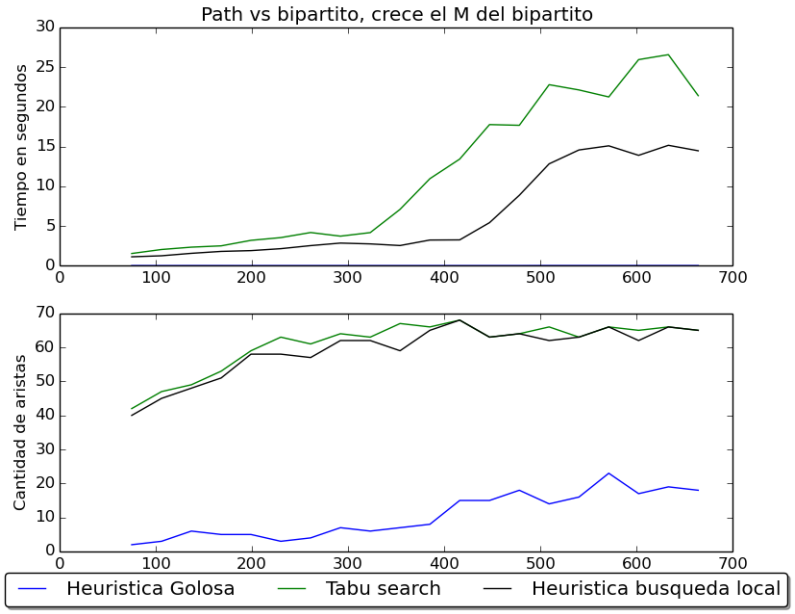


Figura 35: Experimento 7.4

En este experimento tomamos un grafo bipartito, un grafo *path* e incrementamos el número de aristas del grafo bipartito.

Vemos en el gráfico que, por primera vez, un resultado para la heurística golosa que en comparación con los resultados en experimentos anteriores es buena. Además, la diferencia entre la calidad de las soluciones ofrecidas por las otras dos heurísticas es muy poca mientras que la diferencia entre los tiempos es, en comparación más notable, explicado por la diferencia mínima entre cantidad de iteraciones.

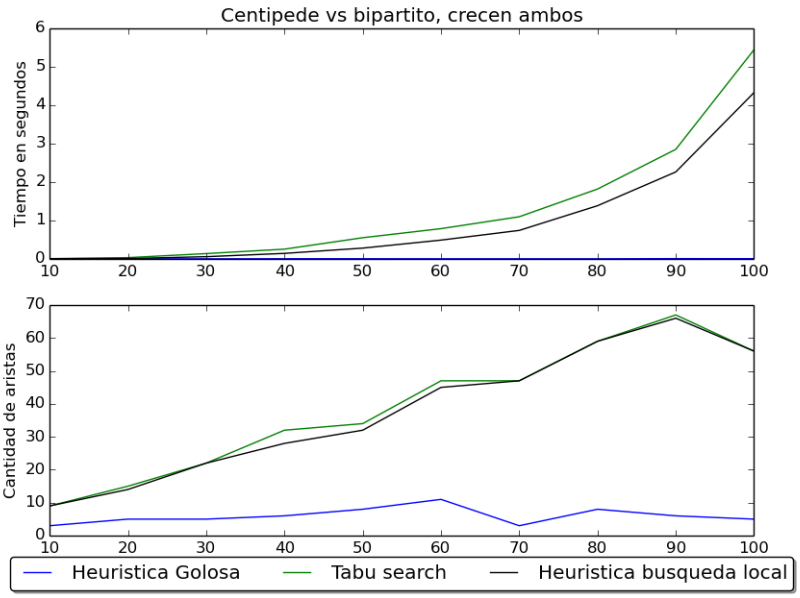


Figura 36: Experimento 7.5

En este experimento tomamos un grafo bipartito, un grafo *centipede* e incrementamos el número de nodos de ambos grafos.

Vemos resultados similares a los del experimento 7.1 donde tanto la calidad de la solución como el tiempo de ejecución de la busqueda local y el tabú search no difieren a grandes rasgos. Esto se debe a que el tabú search se queda con el primer máximo local que encontró y el resto del tiempo de ejecución itera a ver si mejora.

Vemos de nuevo una pobre performance por parte de la heurística golosa dados los resultados muy bajos para las cantidades de aristas en comparación con las otras dos heurísticas.

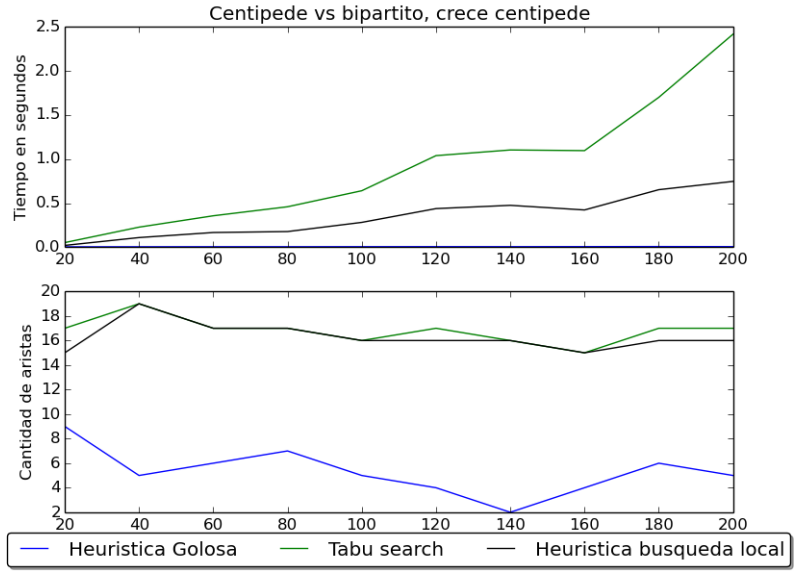


Figura 37: Experimento 7.6

En este experimento tomamos un grafo bipartito, un grafo *centipede* e incrementamos el número de nodos del grafo *centipede*.

En este caso las respuestas obtenidas por la busqueda tabú fueron mejores o iguales a las de la busqueda local y, en el caso que hayan sido mejores, lo fueron por a lo sumo 2 aristas.

Respecto de los tiempos, vemos que el tabú search tarda más que la busqueda local debido a las mejoras que encuentra y a la cantidad mínima de iteraciones que debe realizar. Vemos el mismo comportamiento en el siguiente gráfico para el experimento 7.7.

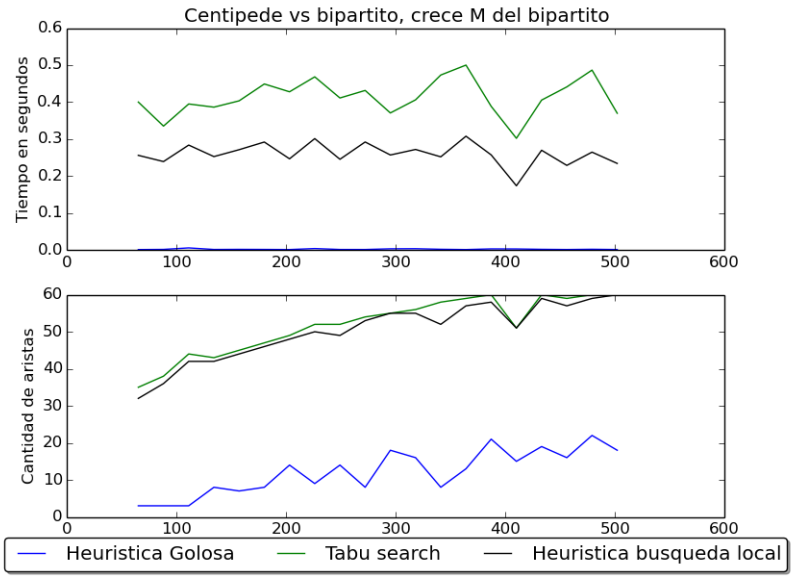


Figura 38: Experimento 7.7

A lo largo de esta experimentación, vimos que la heurística golosa no es de mucha utilidad para proveernos con una solución final para el problema de MCS. La única ventaja a su favor es el corto tiempo de ejecución que toma. En cuanto a las heurísticas restantes, vimos que tabú search suele mejorar la calidad de la solución en comparación con la heurística de busqueda local. Sin embargo, las diferencias entre las cantidades de aristas de las soluciones de MCS de tabú search y busqueda local fueron, en la mayoría de los casos, insignificantes. La ventaja que provee tabú search sobre busqueda local es que garantiza una respuesta tan buena como la de busqueda local, la desventaja es que garantiza un tiempo de ejecución igual o mayor a la de busqueda local.

Sin embargo, dado que el tabú search permite controlar la cantidad máxima de iteraciones a realizar hasta conseguir una mejora en la solución, se puede, si se desea, ponerla en uno de manera que emule una busqueda local. No obstante, esto vencería el propósito de implementar una heurística de tabú search.

En conclusión, optamos por el tabú search dado que da lugar a ser optimizado vía sus parametros y criterios, especialmente si se conocen las particularidades de la entrada para el problema.