Algoritmos y Estructuras de Datos III

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

Trabajo Práctico 3

Heurística GRASP aplicada al problema CIPM

Grupo 3

Integrante	LU	Correo electrónico
Andrés Osinski	405/07	andres.osinski@gmail.com
Patricio Reboratti	713/06	darthpolly@gmail.com
Damián Silvani	658/06	dsilvani@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Introducción

1.1. Explicación del problema

Dado un grafo con pesos asignados a sus vértices, el problema del Conjunto Independiente de Peso Máximo (CIPM) consiste en encontrar el subconjunto de peso máximo tal que sus vértices no sean adyacentes de a pares. Si lo analizamos por partes, podemos ver que la dificultad es muy variable. Para encontrar un conjunto independiente basta con tomar un vértice cualquiera. Teniendo en cuenta que nuestro fin es encontrar el CIPM, podemos pensar en encontrar un conjunto de vértices independiente y *maximal*. Basta entonces con recorrer todos los vértices y tomar aquellos que no sean adyacentes con ninguno que ya hayamos tomado, lo cual se puede hacer con un algoritmo de complejidad polinomial. Pero si quisieramos encontrar el conjunto de vértices independientes con mayor cardinalidad, se nos presenta un problema más complicado, porque entra en juego el orden en el que agregamos los vértices. Por ejemplo, en un grafo de tipo rueda (*wheel*), si empezamos por el centro luego no podremos agregar ningún otro vértice, pero seguro que hay un conjunto maximal con mayor cardinalidad si empezamos por un vértice del borde. Está demostrado que no existe un algoritmo que encuentre en un grafo el conjunto independiente de mayor cardinalidad en tiempo polinomial. Finalmente, si volvemos al problema que nos interesa resolver, observamos que encontrar nuestro máximo ya no depende de la cardinalidad del conjunto sino de la suma de los pesos de los vértices, con lo cual podemos tener un conjunto de un solo vértice pesando mucho más que otro con cientos de ellos (cabe aclarar que ambos problemas son de clase NP-Completo).

1.2. Aplicaciones

Es posible aplicar este problema a una situación que puede ser modelada como un grafo pesado cuyos ejes representan conflictos, es decir, dos vértices son adyacentes sí y sólo sí no pueden ser *seleccionados* al mismo tiempo. El peso de los vértices representa el *beneficio* de seleccionarlos. El objectivo en estas situaciones es buscar el mayor beneficio mientras no se generen conflictos, y esto equivale, en nuestro modelo, a buscar el CIPM.

1.2.1. Aplicación 1

Se desea hacer una encuesta a usuarios populares de una red social para promocionar en una nueva versión de un producto. Se seleccionó una muestra de usuarios candidatos que cumplen ciertos requisitos, y se quiere seleccionar a usuarios de la muestra para que promuevan el producto a la mayor cantidad de amigos posibles, sin que ocurra que dos personas seleccionadas se conozcan entre sí en la red para que no interactúen entre sí para la promoción.

Este problema se modelaría con nodos representando a las personas seleccionadas para la muestra, con los pesos indicando la cantidad de amigos que tiene cada usuario en la red social, y las adyacencias indicando una relación de conocimiento entre dos personas. Como se quiere maximizar a la cantidad de personas a las que llega la campaña sin que los promotores se conozcan, se busca el conjunto independiente de peso máximo.

1.2.2. Aplicación 2

Una empresa que provee servicios de Internet desea instalar centrales en una zona. La zona está dividida en barrios, y en cada barrio se puede instalar una central, aunque por la infraestructura de cada uno, una central instalada en un barrio puede tener una capacidad máxima de ancho de banda. Una central instalada en un barrio puede cubrir a los barrios adyacentes. Se quiere seleccionar el conjunto de barrios donde se construirán las centrales tales que la capacidad de ancho de banda en la zona sea máxima, sin tener más centrales de lo necesario, o sea, sin tener dos centrales instaladas en barrios adyacentes.

Para este problema, cada nodo representa un barrio, cada nodo tiene un peso correspondiente al ancho de banda que puede proveer, y una arista representa una adyacencia entre barrios donde la central en un barrio puede proveer a otro.

1.2.3. Generalidad de aplicación

El problema de **CIPM** sirve para modelar problemas en la vida real en que hayan objetos donde, por motivos externos, solo se puede seleccionar uno de los dos que comparten una relación, y el objetivo es seleccionar los objetos de que provean mayor valor. En ambos ejemplos citamos casos donde se tiene una red de distribución donde se puede seleccionar ciertos candidatos bajo un criterio restrictivo de relación, y se quiere maximizar el valor obtenido.

2. Algoritmo exacto

2.1. Explicación

El algoritmo exacto que proponemos es un algoritmo de backtracking que encuentra el conjunto independiente de peso máximo en un grafo pesado en los vértices. Primero genera el grafo complemento a partir del grafo de entrada, luego busca recursivamente la clique de peso máximo, evaluando todos los conjuntos de vértices posibles (todas las combinaciones posibles de vértices que son adyacentes dos a dos). Como una clique en un grafo equivale a un conjunto independiente en el complemento de ese grafo, el algoritmo termina encontrando el conjunto independiente de peso máximo, resolviendo nuestro problema.

La función recursiva **backtrack** tiene una única poda implementada: si el conjunto $T \cup \{v\}$ es un conjunto que ya fue armado previamente (visitado), no procesa ese conjunto de nuevo. De esta manera, se evita procesar todas las permutaciones posibles de cada conjunto independiente del grafo de entrada, y sólo considera cada conjunto independiente una sóla vez. Como no encontramos una manera de identificar un conjunto de vértices que no sea con el conjunto mismo, la función almacena cada conjunto independiente en memoria, y esto conlleva una complejidad espacial exponencial. Es importante aclarar que **no conocemos una manera de acotar la cantidad de conjuntos independientes posibles en un grafo**, con lo cual, el tamaño del conjunto *visitados* será, en el peor caso, la cantidad máxima de subconjuntos posibles (a lo sumo 2^n).

2.2. Pseudocódigo

```
ENCONTRARCIPM( G: GrafoPesado) → res: conjunto
 1  G^c ← complemento(G)
 2  S ← ∅
 3  visitados ← ∅
 4  for each v vértice in G^c
 5        S' ← backtrack(G^c, visitados, v, {v})
 6        if peso(S') > peso(S) then
 7           S ← S'
 8        end if
 9  end for
10  return S
```

```
BACKTRACK( G^c: GrafoPesado, visitados: conjunto, v: vertice, T: conjunto) → res: conjunto
 1  S ← T
 2  for each w in N(v)
 3        if w ∉ T ∧ ((∀ v ∈ T) v ∈ N(w)) ∧ T ∪ {w} ∉ visitados then
 4           S' ← backtrack(G^c, visitados, w, T ∪ {w})
 5           if peso(S') > peso(S) then
 6              S ← S'
 7           end if
 8           visitados ← visitados ∪ {T ∪ {w}}
 9        end if
10  end for
11  return S
```

2.3. Detalles de implementación

Para el cálculo de complejidad es necesario aclarar ciertos aspectos de la implementación del algoritmo. Los grafos pesados están representados con listas de adyacencias y un arreglo de enteros positivos que define el peso de cada vértice.

Los conjuntos de vértices están implementados sobre `std::set` de la librería estándar de C++, y de esta manera el costo de insertar y buscar vértices en los conjuntos es logarítmico en el peor caso[1]. Existía la posibilidad de implementar los conjuntos sobre arreglos, pero la complejidad de inicializarlos es $O(n)$. Como el algoritmo es recursivo, se estarían definiendo al menos un conjunto por cada llamada, implicando una complejidad mucho mayor.

[1] En la implementación de la librería STL de GNU, `std::set` está programado sobre Red-Black Trees, árboles binarios de búsqueda autobalanceables.

2.4. Cálculo de complejidad en el peor caso

Siendo un algoritmo recursivo, primero se analizarán ciertas líneas de cada función, y luego se analizará cuántas veces se llamará la función recursiva en el peor caso.

encontrarCIPM

2 Se genera el complemento del grafo de entrada, armando las nuevas listas de adyacencias de los vértices. Para esto agrega en las nuevas listas los vértices que no están en las listas originales. Luego, la complejidad temporal es $O(n^2)$.

5-7 Para obtener el peso de un conjunto se suman el peso de los vértices que lo conforman, y como en el peor caso tiene n elementos, el costo es $O(n)$. El peso de la mejor solución se almacena en una variable, con lo cual cuesta $O(1)$.

backtrack

1 Se hace una copia del conjunto pasado por parámetro, el costo es $O(n)$.

3 Se comprueba si el vértice w no está en el conjunto T ($O(\log n)$), y si w es adyacente a todos los vértices del conjunto. Para eso basta recorrer la lista de adyacencias de w y buscar cada elemento de T (se lo itera de forma ordenada), y esto tiene complejidad $O(d(w)) = O(m)$. Finalmente, se verifica que $T \cup \{w\} \notin visitados$, y para esto se agrega el vértice w a T ($O(\log 2^n) = O(n)$) y se busca ese conjunto en $visitados$ ($O(n \log 2^n) = O(n^2)$). Por lo tanto, esta línea cuesta $O(\log n + m + n^2) = O(n^2)$.

8 Se agrega el conjunto armado a $visitados$, y esto tiene de costo $O(n \log 2^n) = O(n^2)$ (hace una copia del conjunto, y lo inserta de forma ordenada).

Como vimos, la poda impone una cota a la cantidad de veces que será llamada recursivamente **backtrack**, y esto es, en el peor caso, 2^n veces. De esta manera, sólo se la llamará a lo sumo por la máxima cantidad de subconjuntos posibles, evitando todas sus permutaciones. Sin embargo, utilizar la poda tiene un costo de $O(n^2)$, y ésto ocurrirá por cada permutación de cada subconjunto de vértices armado. Por lo tanto, una posible cota para la complejidad temporal del algoritmo exacto es la siguiente:

$$T(n, m) \in \mathbf{O(2^n n! n^2 m)}$$

donde $2^n n!$ es la cantidad de subconjuntos de vértices y sus posibles permutaciones, y $n^2 m$ corresponde al costo de ejecutar la línea 3 por cada iteración del ciclo de la función. Claramente, la complejidad es de **orden exponencial**, donde el factor que más impacto tiene es 2^n, la cantidad de subconjuntos de vértices.

2.5. Pruebas y análisis de tiempos

Para las siguientes pruebas, construimos grafos pesados de determinada cantidad de vértices y ejes (asignados de forma aleatoria sobre el grafo), y con pesos que varíaban entre 1 y 100.

2.5.1. Comparación de tiempo de resolución para grafos de distinta densidad

La primera prueba consistió en comparar el tiempo de ejecución del algoritmo frente a grafos de distintos tamaños, variando su *densidad*. Éste es un valor porcentual sobre la máxima cantidad de aristas que puede tener un grafo simple ($\frac{n(n-1)}{2}$). En la figura 1 podemos observar la tendencia exponencial del algoritmo: a medida que el valor de n crece y la densidad disminuye, el tiempo aumenta exponencialmente. Esto se debe a que, cuantas menos aristas haya, más cantidad de conjuntos independientes habrá. El tiempo de resolución para los grafos con densidad del 2 % tiene un crecimiento exponencial muy similar a 2^n, y esto se debe a su muy baja densidad y cantidad de vértices, con lo cual prácticamente no tienen aristas.

En la figura 2 podemos ver como aumenta el tiempo de la misma manera para un grafo de 500 nodos. A medida que su densidad disminuye, el tiempo crece exponencialmente.

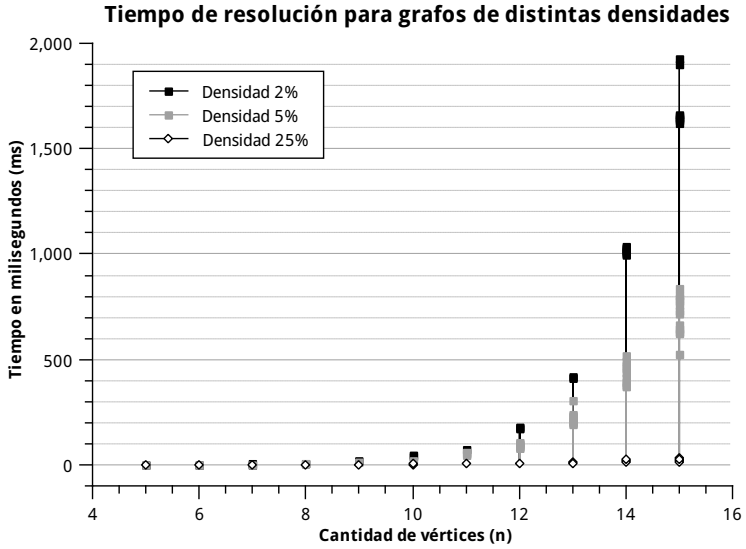


Figura 1: *El tiempo aumenta exponencialmente a medida que la densidad del grafo disminuye.*

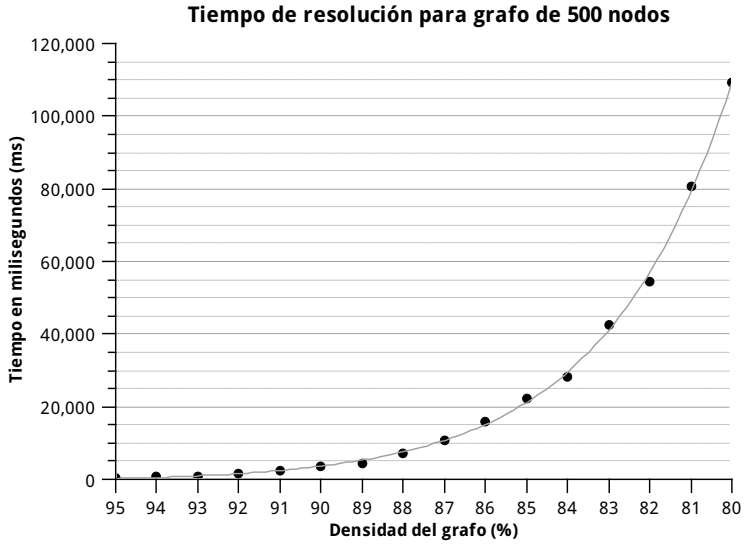


Figura 2: *Mismo comportamiento para un grafo de 500 vértices.*

2.5.2. Comparación de tiempo de resolución para grafos con densidad constante

Como segunda prueba, preparamos un conjunto de grafos de distinto tamaño pero de igual densidad, para poder observar de qué manera crece el tiempo conforme la cantidad de nodos aumenta. Podemos observar en el gráfico de la figura 3 que el tiempo sigue teniendo un comportamiento exponencial. Sin embargo, podemos destacar que el crecimiento es menos acentuado que en la prueba anterior, donde variaba la densidad y el cantidad de vértices se mantenía fija.

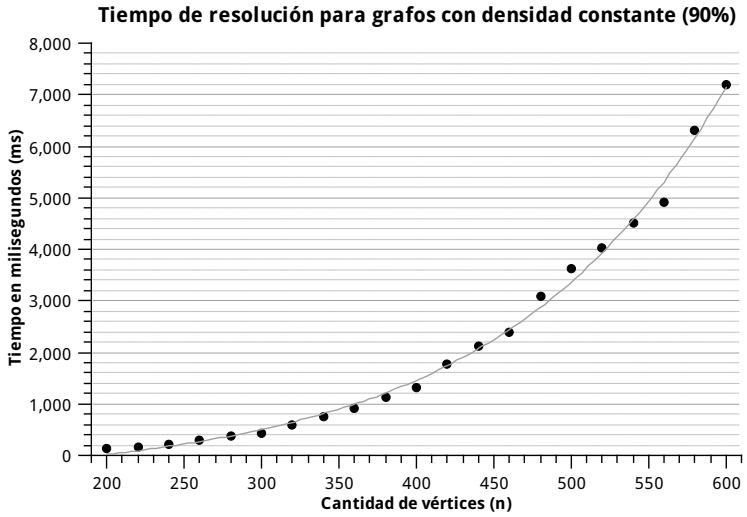


Figura 3: *Sigue habiendo un crecimiento exponencial, pero menos acelerado.*

3. Heurística constructiva

3.1. Ideas principales

Las heurísticas constructivas para este problema consisten en *construir* un conjunto independiente de vértices que, en lo posible, sea parecido al conjunto de peso máximo. Como mencionamos anteriormente, no hay relación alguna entre las adyacencias de los vértices y su peso, con lo cual puede darse la situación que, para un caso particular de grafo, el conjunto independiente de peso máximo esté compuestos por pocos, ó incluso un sólo nodo. De cualquier manera, como los pesos son enteros positivos, creemos que la noción de "cuantos más vértices tenga el conjunto, mayor peso va a tener" sigue siendo válida para muchos casos.

Basándonos en esta idea, la primer heurística constructiva que pensamos es un algoritmo goloso que construye un único conjunto independiente secuencialmente, a partir de una lista de vértices dada:

Heurística 1: Armado de conjunto

1. Ordena los vértices según un criterio de orden.
2. Ciclo goloso: intenta agregar cada vértice de la lista a un conjunto independiente.
3. Devuelve el conjunto armado.

Nos pareció razonable ordenar los vértices del grafo de forma creciente por el grado, y además de forma decreciente por el peso. De esta manera, se intentaría armar un conjunto con los vértices de mayor peso y menor grado. Por intuición, resultaría *más facil* agregar los vértices de menor grado, ya que *probablemente* tengan menos adyacentes a vértices dentro del conjunto armado. No obstante, es fácil generar ejemplos de grafos donde seguir alguno de estos criterios produzca un resultado claramente inferior a las alternativas:

1. Greedy por peso: Para cualquier grafo rueda W_n, si el vértice universal tiene un peso levemente mayor que el resto de los nodos, será el primero seleccionado, si bien impide la selección del resto de los nodos.
2. Greedy por adyacencias: Dado un grafo G con un vértice universal de peso mayor a la suma del resto de los nodos, este no será seleccionado.

Si bien estos casos son un tanto extremos, podemos ver que para grafos con nodos con mucha varianza de pesos una selección por pesos probablemente resultaría superior, mientras que en uno con gran varianza entre los grados de cada nodo pero pesos más uniformes el criterio de adyacencias da mejores resultados. No obstante, como no poseemos de ninguna propiedad teórica que nos respalde, éste no fue el único criterio para ordenar los vértices que consideramos al hacer pruebas.

Por otro lado, en el coloreo de un grafo, las clases de colores por definición representan conjuntos independientes de vértices. A partir de esta propiedad, descubrimos otra idea de heurística constructiva similar a la anterior, pero que en general provee mejores resultados. Como la complejidad temporal era la misma, y porque siempre generaba mismos o mejores resultados, decidimos desarrollar ésta como heurística constructiva principal.

Heurística 2: Coloreo

1. Ordena los vértices según un criterio de orden.
2. Ciclo goloso: colorea el grafo tomando cada vértice y asignándole el primer color disponible.
3. Devuelve la clase de color de mayor peso.

El algoritmo está basado en la conocida y sencilla heurística para coloreo de grafos: el **algoritmo de selección**. Éste a su vez, al igual que nuestra primera idea de heurística constructiva, ordena los vértices del grafo de acuerdo a un criterio de orden establecido.

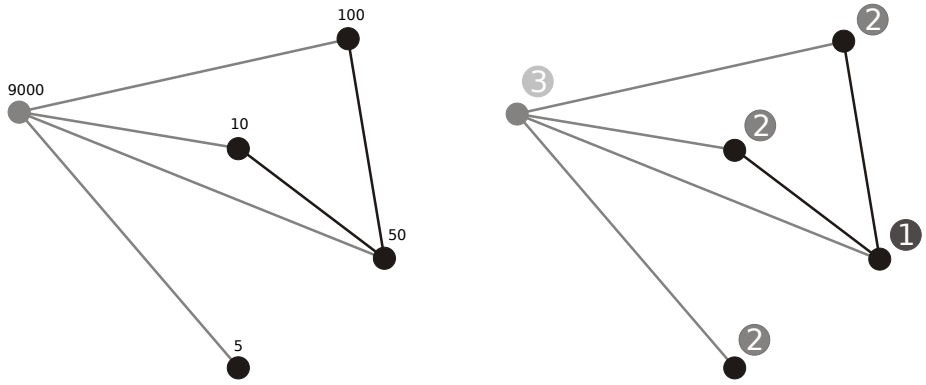
Es importante mencionar que el criterio de orden es esencial en el comportamiento de estos dos algoritmos. Si bien, se puede demostrar que existe una permutación de los vértices que provoca que los algoritmos encuentren la solución óptima, existe también la posibilidad de que caiga en una solución muy mala. Afortunadamente, si uno ordena de determinada manera, se puede demostrar que existe una cota mínima que asegura que el algoritmo no genere una solución de menor calidad que la cota.

3.2. Explicación

La heurística constructiva que proponemos consiste en construir un conjunto independiente de vértices coloreando el grafo, dado que los vértices que tienen el mismo color no son adyacentes dos a dos.

3.2.1. Comparación con la primera heurística

Como vimos en clase, los conjuntos de colores en un grafo coloreado corresponden a conjuntos independientes. Como la heurística de coloreo usa el mismo criterio de órden de vértices que la primera heurística, la diferencia principal radica en la forma que **greedyColoring** (la heurística de coloreo) selecciona uno de todos los conjuntos de colores, mientras que la otra devuelve siempre el primero conjunto independiente, es decir, la primera clase de color, aunque no sea el de mayor peso.



(a) Grafo con nodo universal de gran peso (b) Mismo grafo coloreado

Figura 4: *En este ejemplo, la primer heurística obtiene una peor solución que la heurística de coloreo.*

En el ejemplo de la figura 4 tenemos un grafo sin ninguna particularidad más que un nodo universal que es adyacente a todos los demás, y con un peso mayor a cualquier otro. Suponemos que el criterio de órden para éste ejemplo es `ORDEN_MENOR_GRADO_MAYOR_PESO`, y por lo tanto, el ordenamiento de los vértices para las dos heurísticas quedaría de la siguiente manera (dado que los pesos son todos distintos, se nombrará cada vértice con su peso para facitilitar la lectura):

$$Orden = [5, 100, 10, 50, 9000]$$

El de peso 9000 (el nodo universal) aparece al final de la lista porque es el que tiene mayor grado. Se puede observar que la solución al CIPM en este grafo es un conjunto con ése único nodo. Sin embargo **greedyGrados** (la primera heurística) hubiera encontrado como solución al conjunto {5, 10, 100}.

Por esta razón, podemos concluir que la heurística de coloreo **greedyColoring** siempre encontrará una solución de mayor o igual peso que la heurística **greedyGrados**.

3.3. Pseudocódigo

```
GREEDYCOLORING(G: grafo) → solucion: CIPM
 1  var coloreo[ ]
 2  ordenarNodos(G)
 3    { Colorea el primer vértice con el color 1. }
 4  coloreo[1] ← 1
 5    { Luego colorea el resto de los vértices usando el mínimo color disponible. }
 6  for each v in nodosOrd
 7        var U ← {1, ..., d(v) + 1}
 8        for w in N(v)
 9              var c ← coloreo[w]
10              if c ≠ 0 and c < d(v) + 1 then
11                 U ← U − {c}
12              end if
13        end for
```

14 $coloreo[v] \leftarrow \text{mín}\{U\}$
15 **end for**
16 **return** máx $\big(claseColor(coloreo)\big)$

3.4. Detalles de implementación

Para este ejercicio tomamos a un grafo como un arreglo de listas de adyacencia y un arreglo de pesos de los nodos. A partir de eso, el factor más importante para determinar la funcionalidad del algoritmo es la definición de la función de comparación, que dependiendo del criterio que se usa, determina la manera en que funcionará $ordenarNodos()$. Cabe notar que a través del uso de `#DEFINE`s en el código implementamos cinco criterios de órden distintos:

- `ORDEN_MAYOR_PESO`: considera solo el peso de los nodos.
- `ORDEN_MENOR_GRADO_MAYOR_PESO`: considera el menor grado, y entre los de mismo grado, el de mayor peso.
- `ORDEN_MAYOR_PESO_MENOR_GRADO`: ordena primero mayor peso, luego por menor cantidad de grados.
- `ORDEN_MAYOR_GRADO_MAYOR_PESO`: mayor grado y mayor peso.
- `ORDEN_MAYOR_PESO_MAYOR_GRADO`: igual que el anterior, pero priorizando el peso por sobre los grados.

Se utilizó la función `std::sort` de la librería estándar de C++, la cual está implementada con el algoritmo *introsort*, un híbrido entre quicksort y heap sort que promete una complejidad de $O(n \log n)$ en el peor caso.

3.5. Cálculo de complejidad en el peor caso

1 Se define e inicializa un arreglo de tamaño n que determina de qué color es cada vértice del grafo. Su costo es $O(n)$.

2 Se ordenan los vértices según el criterio de orden definido. La complejidad del algoritmo de sorting utilizada es $O(n \log n)$.

7 Se define un arreglo de tamaño $d(v) + 1$ que determina qué colores están disponibles para colorear el vértice v, luego el costo es $O(d(v) + 1)$.

8-13 Este ciclo itera por cada vecino del vértice v, es decir $d(v)$ veces. En cada iteración se *tacha* el color utilizado por el vecino, y como se utilizaron arreglos, estas operaciones tienen una complejidad constante. Luego, la complejidad del ciclo es $O(d(v))$.

14 Obtener el mínimo color disponible implica recorrer el arreglo hasta el último elemento en el peor caso, y como su tamaño es $d(v) + 1$, el costo de esta línea es $O(d(v) + 1)$.

6-15 Es el ciclo principal de coloreo del grafo. Itera por cada vértice para colorearlo, es decir, n veces. Como la complejidad de cada iteración es $O(d(v_i) + 1)$ donde v_i es el vértice de esa iteración, y además vale que $\sum_{i=1}^{n} d(v_i) + 1 = \sum_{i=1}^{n} d(v_i) + n = n + m$, entonces la complejidad temporal del ciclo es $O(n + m)$.

16 Es posible buscar la clase de color de mayor peso recorriendo el arreglo de coloreo, luego el costo es $O(n)$.

Podemos concluir que la complejidad temporal del algoritmo es $O(n + n \log n + (n + m) + n)$, es decir, $\mathbf{O(n \log n + m)}$.

3.5.1. Complejidad en función del tamaño de la entrada

Como la entrada de la función es un grafo representado con listas de adyacencia y una lista de pesos para cada vértice, podemos ver que la entrada es la variable n que determina la cantidad de nodos del grafo, seguido de las listas de adyacencias de cada nodo y su peso. Luego el tamaño de entrada es:

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

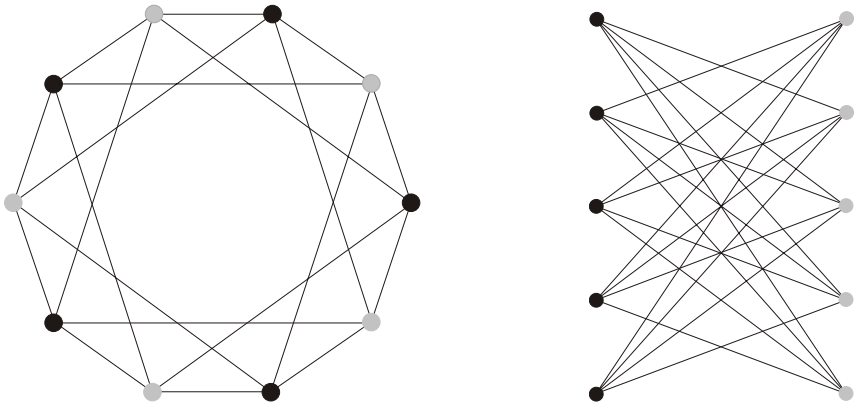
Luego se tiene que $E(n,m) \geq n+m$, pero como n y m son enteros positivos, también se tiene que $E(n,m) \geq n$ y $E(n,m) \geq m$. Por lo tanto, la complejidad en función del tamaño de entrada es:

$$T(n,m) \in O(n \log n + m) = O(E(n,m) \log(E(n,m)) + E(n,m)) = \mathbf{O(E(n,m) \log(E(n,m))}$$

3.6. Peores casos

Como explicamos al principio, el criterio de orden de los vértices para el coloreo goloso es el factor determinante de la calidad de la solución generada. Mostramos a continuación una familia de grafos donde el algoritmo no sólo produce un resultado subóptimo, sino que es particularmente malo en comparación con el óptimo.

Un grafo corona (*crown graph*) es un grafo bipartito donde cada partición U y V tiene n vértices, y existe un eje entre los vértices $u_i \in U$ y $v_j \in V$ si y sólo si $i \neq j$ (ver figura 5). Luego, el grado de todos los vértices es siempre $n-1$, y la cantidad de ejes que tiene el grafo es $n(n-1)$.

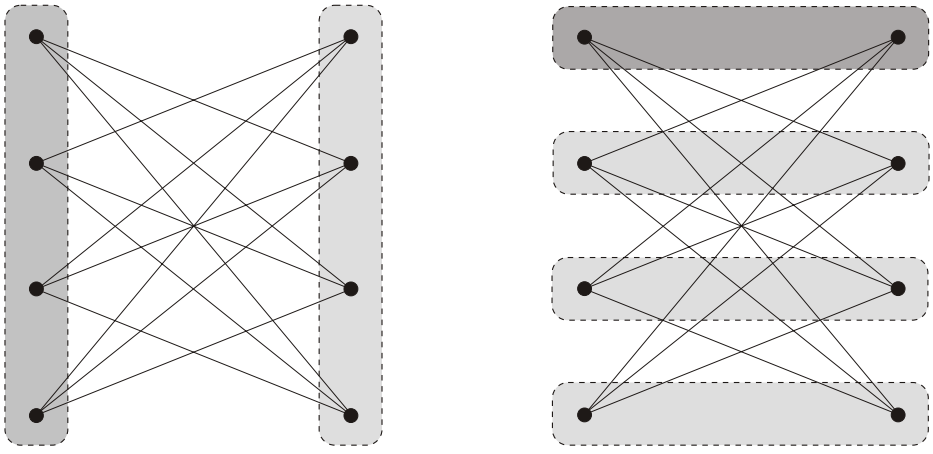


(a) Representación característica (b) Como grafo bipartito

Figura 5: *Dos maneras de representar un grafo corona de $n = 5$.*

En el problema de coloreo de grafos, es conocido que el grafo corona es un caso malo para el algoritmo selectivo (coloreo goloso). Si el orden de vértices es $u_1, v_1, u_2, v_2, ...$, podemos ver que el algoritmo utilizaría n colores, cuando claramente el grafo es 2-coloreable siendo un grafo bipartito.

Volviendo a nuestro problema, observamos que el orden de los vértices depende del criterio utilizado. Como en este grafo el grado de los vértices es siempre el mismo, los criterios se resuelven en uno solo: **mayor peso**. En un grafo corona pesado donde vale que $p(u_1) > p(u_2) > ... > p(u_n) > p(v_1) > p(v_2) > ... > p(v_n)$, el algoritmo va a devolver como solución la partición U, y como coincide con la solución óptima, es un mejor caso sobre este grafo (ver figura 6(a)).



(a) Mejor caso (b) Peor caso

Figura 6: *Grafo corona con $n = 4$, mejor caso y peor caso del algoritmo.*

Por otro lado, si tenemos un grafo donde vale que $p(u_1) > p(v_1) > p(u_2) > p(v_2) > ... > p(u_n) > p(v_n)$, el algoritmo formaría conjuntos independientes de 2 vértices (ver figura 6(b)). Aún así, puede que uno de esos conjuntos sea casualmente la solución óptima. Para evitar esto, agregamos otra condición sobre los pesos:

$p(u_1) < \sum_{i=2}^{n} p(v_i)$ y $p(v_1) < \sum_{i=2}^{n} p(u_i)$. El caso en el cual la diferencia de pesos entre la solución devuelta y la óptima es mayor es cuando los pesos de los vértices van disminuyendo en 1, es decir, $p(v_i) = p(u_i) - 1$ y $p(u_{i+1}) = p(v_i) - 1$, con $1 \leq i \leq n$.

Observamos entonces que la elección de formar un conjunto independiente entre u_1 y v_1 es mala en comparación con el conjunto formado por la partición U ó V. Creemos que, en general, la heurística constructiva falla en generar una buena solución cuando ocurre una situación similar a la mostrada. Es interesante acotar que las probabilidades de encontrarse con un grafo corona de estas propiedades aumentan a medidad que el peso máximo aumenta, dado que habría más combinaciones de pesos que cumplen con las condiciones presentadas.

3.7. Análisis de tiempo y calidad

Para el siguiente análisis, realizamos pruebas con grafos construidos manualmente para casos particulares. Consideramos que la relevancia de estas pruebas va más allá del marco puramente teórico, por lo cual, para las pruebas automatizadas de tamaños variantes, tomamos tres modelos de generación de grafos aleatorios:

- Grafos generados por el modelo Erdos-Renyi.
- Grafos rueda W_n.
- Grafos generados por el modelo Barabasi-Albert.

Contemplamos en un momento modelar también árboles para estas pruebas, pero como el algoritmo no se aprovecha de ninguna propiedad particular de los árboles, decidimos hacer el análisis con grafos más complejos.

3.7.1. Modelo Erdos-Renyi

El modelo Erdos-Renyi fue desarrollado en la década de los 30ś para hacer pruebas sobre grafos aleatorios. Por las características de este, tiene la ventaja teórica de que para ciertos valores de sus parámetros, se manifiestan varias propiedades con una muy alta probabilidad (si el grafo es conexo, la longitud media de los caminos, etc.). El modelo toma dos parámetros: n, que indica la cantidad de nodos en el grafo, y p, la probabilidad de que exista una arista entre dos nodos. Para las pruebas en cuestión probamos generar grafos con n de 5 a 600 nodos, y con un p que varia de 1 % a 90 %, de esa manera produciendo una gama de grafos aleatorios muy chicos y muy grandes, y densos y esparsos. Además, se generaron 10 instancias de cada tipo de grafo.

3.7.2. Modelo rueda

El grafo rueda W resulta útil para probar la utilidad de los distintos criterios de comparación que se utilizan en las heurísticas. Cuenta con un único nodo universal, y nodos periféricos, cada uno con tres adyacencias, formando una *rueda* alrededos del nodo universal. Lo que nos permite esto es probar a *grosso modo* la efectividad de heurísticas donde la selección de un nodo en particular para el conjunto independiente resultaría en casos muy malos. En efecto, la selección del nodo universal para el conjunto independiente casi siempre resulta la peor opción factible, a menos que su peso sea superior al del resto de los nodos que se pueden conseguir de la perfieria. Esto muestra a cierto nivel la vulnerabilidad de la heurística al consumo de nodos con una gran cantidad de adyacencias.

3.7.3. Modelo Barabasi-Albert

El modelo Barabasi-Albert fue formulado en 1999 para generar grafos que modelan el comportamiento de varias estructuras reales que exhiben crecimiento sin escala (scale-free growth) característico en redes sociales y de comunicación y distribución. La cantidad de grados de los nodos en este grafo sigue una distribución exponencial negativa para cualquier tamaño, y en la práctica resulta muy útil para analizar estos casos. Dado que las aplicaciones del problema de conjunto independiente de peso máximo incluyen estas clases de redes, consideramos pertinent incluirlo en nuestra bateria de pruebas.

Este modelo fue propuesto en el paper *Emergence of Scaling in Random Networks, 1999*, publicado por Albert László-Bárabási y Reká Albert.

En todos los caso, a menos que sea explícitamente mencionado, los pesos de los nodos fueron asignados un valor entre 1 y 100 en forma aleatoria. Más adelante hacemos hincapié en la relevancia de esto.

3.7.4. Comparación de resultados contra la solución exacta

En principio, analizamos los valores de la heurística para los casos donde es factible correr el algoritmo de solución exacta. Para eso comparamos los algoritmos candidatos contra la solución presentada para el algoritmo exacto con grafos con n entre 5 y 10, dando un tiempo de resolución aceptable para el exacto.

Con la tabla 1 vemos que para estos grafos chicos las heúristicas dan, en muchos casos, la solución exacta, y se observa que el factor determinante del éxito de la heurística radica en el criterio de selección. No obstante, el coloreo parece ganar en un par de casos adicionales, donde tiene la ventaja de poder elegir el conjunto de mayor valor. Esto es meramente indicativo de que los criterios de selección son sensatos para un conjunto importante de grafos chicos que no apuntan a tener nodos de excesivo peso o grado, ni estructuras donde los buenos resultados sean altamente dependientes de unos pocos nodos seleccionados o no seleccionados.

n	m	Exacto	GC>G	GC>P	GC<G	GG>G	GG>P	GG<G
5	6	28	28	21	21	13	21	21
6	9	23	20	23	20	16	23	23
7	9	33	27	30	30	19	27	27
8	13	43	25	43	32	25	43	43
9	11	45	43	45	45	26	45	45
10	15	48	40	48	48	35	48	48

n	Cantidad de vértices
m	Cantidad de aristas
P	Peso
G	Grado
GC	Greedy Coloring
GG	Greedy Grados

Cuadro 1: *Observamos que el para grafos chicos ambas heurísticas muestran buenos resultados si su criterio golozo es el peso.*

3.7.5. Comparación con grafos aleatorios Erdos-Renyi

Pasamos a ver cómo funciona el algoritmo para grafos aleatorios más grandes donde una comparación exacta no es factible, dada la naturaleza NP-hard de encontrar la solución exacta. En la tabla 2 presentamos los resultados de los algoritmos ejecutados sobre grafos Erdos-Renyi con n variable y $p = 20\,\%$.

Vemos entonces que surgen claros patrones de la efectividad de cada estrategia. El Greedy Coloring que ordena por mayor peso y luego por menor grado tiene la mayor cantidad de resultados máximos, seguido por el coloreo que orden en segundo lugar por mayor grado. Le siguen los golosos convencionales con los mismos órdenes, y en forma muy distante el resto de los criterios.

3.7.6. Comparación de resultados con grafos Barabasi-Albert

Ahora pasamos a ver cómo compiten ambas heurísticas bajo el modelo de Barabasi, con las distintas comparaciones. Dado que estos grafos son más relevantes para aplicaciones reales, los resultados fueron tomados con más peso a la hora de elegir la técnica más útil. Los elementos en la tabla 3 muestran los pesos obtenidos para cada una.

Vemos que bajo este modelo los resultados son mucho más parejos entre el coloreo y la golosa convencional. Esto se debe a que, dada la distribución de nodos bajo este modelo, es muy dificil que se formen conjuntos independientes con valores cercanos al del conjunto más grande, debido al efecto de *clustering* que se produce y la distribución de grados.

3.7.7. Comparación de resultados con grafos rueda

Utilizamos el grafo rueda por ser un indicador efectivo, si bien un tanto extremo, de la capacidad de una heurística para lidiar con grafos donde la selección de un nodo puede tener efectos importantísimos sobre el peso del conjunto seleccionado. En este caso, la selección del nodo universal produce como conjunto independiente a este.

El gráfico 7 muestra el tiempo de resolución para el algoritmo Greedy Grados por mayor grado, el único algoritmos “perdedor”, comparado con el mínimo entre el resto de los otros algoritmos. Dado que ninguno de

n	GC >G>P	GC >P>G	GC >P<G	GC <G>P	GG >G>P	GG >P>G	GG >P<G	GG >P	GG <G>P
10	319	319	319	291	319	319	319	319	291
20	487	520	520	553	487	520	520	520	553
30	531	681	681	547	331	681		681	547
50	655	745	745	710	431	745	745	745	710
75	738	1105	1105	902	506	1105		1105	902
100	857	1056	1056	770	687	950	950	950	756
125	865	1395	1395	1027	615	1395	1395		1027
150	965	1275	1275	1021	768	1275	1275	1275	1021
175	946	1304	1304	957	717	1304	1304	1304	975
200	944	1303	1322	988	802	1184	1184	1184	912
250	1089	1561	1561	1103	929	1561	1561	1561	997
300	1070	1592	1552	1279	1070	1592	1552	1589	1072
350	1241	1662	1643	1373	966	1547	1544	1544	1290
400	1160	1732	1768	1094	1139	1732	1768	1768	1031
450	1312	1789	1671	1476	1169	1789	1671	1639	1476
500	1469	1797	1792	1306	964	1792	1792	1792	1285
600	1311	1899	1861	1418	1086	1899	1754	1754	1418

n	Cantidad de vértices
P	Peso
G	Grado
GC	Greedy Coloring
GG	Greedy Grados

Cuadro 2: *Vemos que el criterio de selección por mayor peso casi siempre produce los mejores resultados para todos los tamaños, y Greedy Coloring gana en más casos que Greedy Grados.*

n	GC >G>P	GC >P>G	GC >P<G	GC <G>P	GG >G>P	GG >P>G	GG >P<G	GG >P	GG <G>P
10	368	368	368	384	368	368	368	368	384
20	571	616	616	680	571	616	616	616	680
30	890	1145	1145	1110	890	1145	1145	1145	1110
50	1634	1936	1967	1912	1634	1936	1967	1936	1912
75	2037	3147	3147	3147	2037	3147	3147	3147	1912
100	3040	4314	4314	4438	3040	4314	4314	4314	1912
125	3689	4549	4549	4344	3689	4549	4549	4549	4344
150	4300	5464	5616	5545	4300	5464	5616	5616	5545
175	5027	5867	5867	5981	5027	5867	5867	5867	5981
200	5032	6686	6923	6639	5032	6686	6923	6686	6639
250	6812	8325	8311	8229	6812	8325	8311	8311	8229
300	7494	9573	9573	9344	7494	9573	9573	9573	9344
350	7356	10163	10163	9961	7356	10163	10163	10174	9961
400	8921	11306	11306	10665	8921	11306	11306	11306	10665
450	9234	12747	12989	11796	9234	12747	12989	12919	11796
500	11076	13879	13879	13846	11076	13879	13879	13879	13846
600	11899	15111	15126	14126	11899	15111	15126	15168	14126

n	Cantidad de vértices
P	Peso
G	Grado
GC	Greedy Coloring
GG	Greedy Grados

Cuadro 3: *El criterio de comparación para hacer la asignación golosa sigue siendo el factor más importante para los puntajes de cada algoritmo. Aquí la opción de coloreo y la puramente golosa se comportan en forma casi idéntica.*

los algoritmos "buenos" difirió en su resultado por más de 100 unidades, y esto ocurrió en solo un par de valores de n, consideramos que hacer un análisis más detallado de los valores obtenidos no agrega a este punto.

Casi todos los criterios golozos producen resultados buenos, salvo el GreedyGrados por mayor grado y mayor peso. Se observa aca la ventaja de la estrategia de coloreo: aún con este peor caso, se puede elegir un conjunto independiente adecuado que el sistema golozo convencional hubiera obviado.

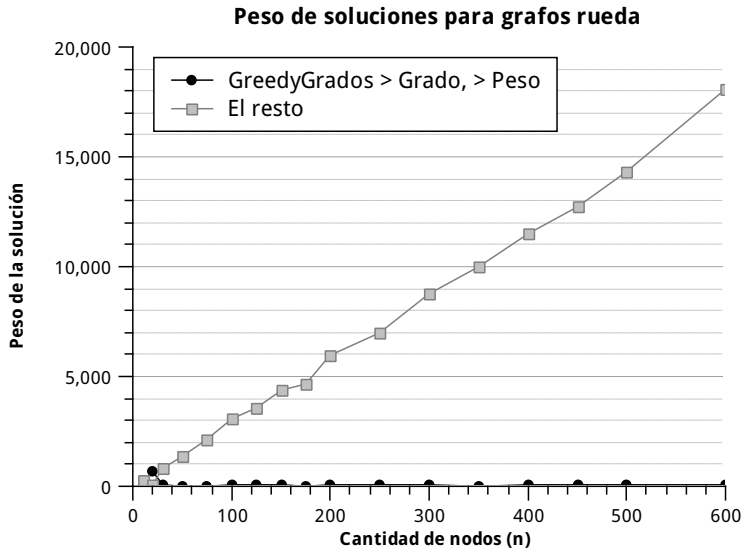


Figura 7: *La mayoria de los algoritmos obtienen buenos resultados en los W_n.*

3.7.8. Resultados bajo grafos con rango bajo de grados

Una hipótesis que surge bajo los analisis realizados es si por los pesos que admiten los nodos, (entre 1 y 100) no resulta más importante para un buen resultado la priorización del peso por encima de la de los grados en forma indeseadas. Para comprobar esto hicimos un análisis sobre grafos Barabasi-Albert, donde el rango admisible de los nodos varia solo entre 1 y 5. De esa forma, pensamos, la importancia de no seleccionar un nodo central en los clusters que se arman es más importante para seleccionar un conjunto independiente de mayor peso que la selección individual de nodos que podrían inhabilitar conjuntos de nodos adyacentens. En la tabla 4 vemos los resultados con grafos de 10 a 500 nodos. Es aparente que aun teniendo la clase de clusters vistos en fenómenos naturales, el peso es de mayo importancia. Nótese, sin embargo, que la diferencia entre la heurística que prioriza los pesos no es mucho mejor que la que prioriza los grados más bajos, y se podrian considerar sus resultados como satisfactorios.

N	GC *G* >P	GC >P >G	GC >P <G	GC <G >P	GG >G >P	GG >P >G	GG >P <G	GG <G >P
10	22	26	26	26	22	26	26	26
20	44	52	52	52	33	52	52	52
30	45	62	64	59	45	63	64	59
50	95	117	114	114	95	117	114	114
100	158	218	225	221	258	218	225	221
150	213	297	313	316	213	297	313	316
200	336	378	401	388	336	378	401	388
250	455	505	541	529	455	505	541	529
300	425	546	576	523	425	546	576	523
350	457	588	620	594	457	588	620	594
400	455	603	647	620	455	603	647	620
450	569	735	760	742	569	735	760	742
500	607	726	804	758	607	726	804	758

Cuadro 4: *Aún en un grafo donde la selección de nodos es más importante que en otros ejemplos, las heurísticas que ordenan por peso tienen mejores resultados.*

3.7.9. Análisis de tiempos

Ahora pasamos a ver cómo escala el algoritmo de coloreo propuesto aqui. En primer lugar analizamos qué pasa con grafos dada una cantidad fija de nodos. Observamos en la figura 8 que cuando la densidad es baja, los tiempos de resolución fluctuan mucho, aun cuando se toman bajo la suma de 10 casos distintos. Esto es de esperar, dado que el grafo no tiene ninguna estructura fija y es muy sujeto a las particularidades de cada grafo generado, por lo que casos como grupos aislados de nodos, cliques muy grandes, y cadenas largas de vértices afectan sustancialemente el tiempo de resolución.

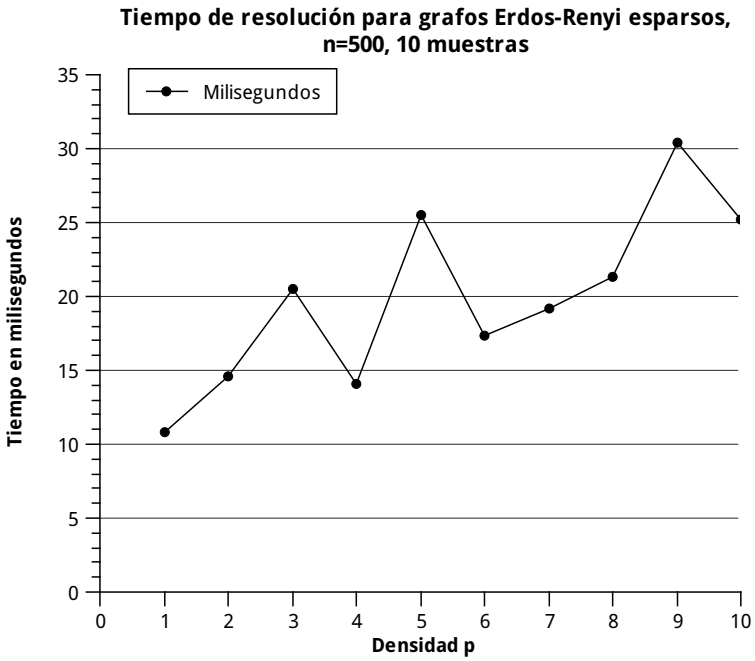


Figura 8: *Los grafos muy esparsos tienen tiempos muy variables, pero conforme aumenta la densidad se sigue un trend temporal claro.*

Al analizar grafos más densos, vemos que los resultados son mucho más regulares relativamente (fluctuan similarmente a los grafos más chicos, pero proporcionalmente el efecto es menor). Como se puede observar en la figura 9, efectivamente las "anomalias" que pueden ocurrir en grados esparsos tiene efectos mucho menores (además de surgir menos debido a la cantidad de aristas). Vemos que la heurísitica incurre un tiempo correspondiente con la cota temporal de $O(n \log n + m)$, donde la componente más significativa radica en la cantidad de aristas, que crece linealmente debido al incremeneto del porcentaje de adyacencias.

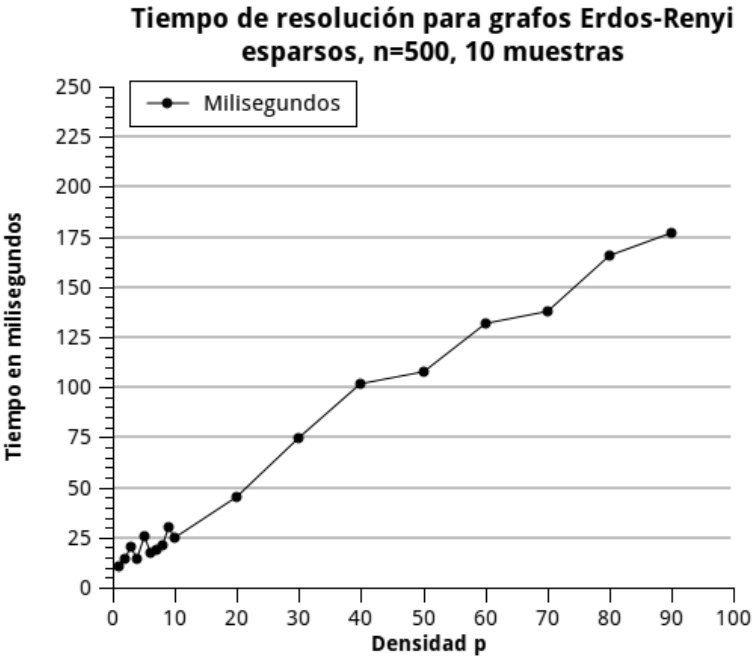


Figura 9: *Se exhibe una tendencia de incremento de tiempo que corresponde con la cota temporal.*

Por último, hacemos un análisis del tiempo para grafos rueda, que tienen siempre una cantidad proporcionalmente idéntica de aristas por nodo, probando para distintas cantidades de nodos. La figura 10 muestra un crecimiento que, salvo un level pico en la marca de $n = 300$ debida a factores externos, es completamente lineal.

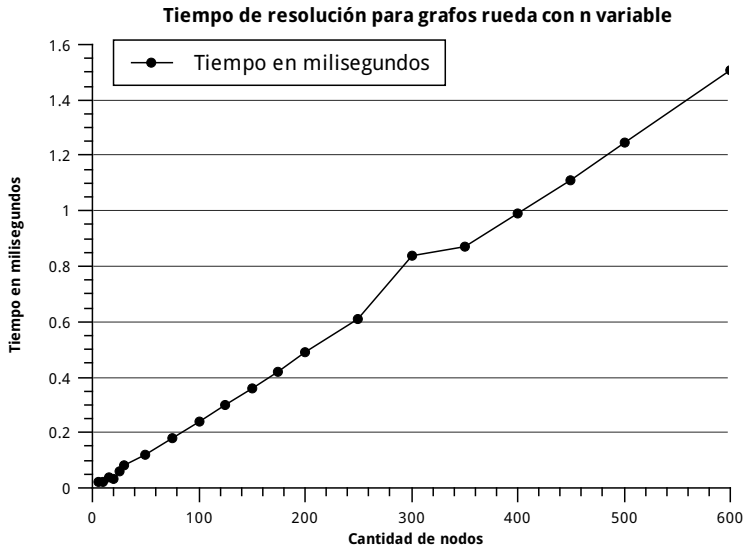


Figura 10: *El crecimiento es esencialmente lineal respecto a la cantidad de nodos.*

Vemos que la complejidad crece linealmente respecto a la cantidad de nodos que tiene el grafo. Esto es de esperar, dado que cada nodo solo agrega, en términos implementativos, una lista de adyacencia adicional con una cantidad fija y chica de nodos (3). Este, al igual que los otros gráficos de tiempos, muestran que la heurística escala muy bien, por lo que es en términos temporales un muy buen candidato para aplicar en la vida real a grafos que superan lo realizable con algoritmos exactos.

3.8. Parámetros finales

Tras haber realizado las pruebas anteriores, elegimos como criterio de órden en el coloreo goloso el de **mayor peso y menor grado**, es decir, dado dos vértices u, v, vale que

$$u < v \quad \Leftrightarrow \quad p(u) > p(v) \quad \vee \quad \big(p(u) = p(v) \quad \wedge \quad d(u) < d(v)\big)$$

4. Heurística de búsqueda local

4.1. Explicación

La heurística de búsqueda local tiene como objetivo tomar una solución existente a un problema y mejorarla a través de un proceso de selección iterativa que intenta maximizar un resultado dentro de una *vecindad* local.

El algoritmo consiste en realizar una cierta cantidad de iteraciones de búsqueda local. La cantidad de veces que se realiza esta búsqueda está acotada por el parámetro `CANT_ITER_MAX`. No obstante, cuando se alcanza el límite definido por el parámetro `CANT_MAX_ITER_SIN_MEJORAS`, la búsqueda alcanzó un máximo local y el algoritmo se detiene, aún si no se alcanzó la cota de iteraciones máxima.

El componente crítico de la búsqueda local es definir una vecindad sobre la cual realizar la búsqueda, de tal forma que la generación de la vecindad y la búsqueda de la mejor solución dentro de esa vecindad sea eficiente y logre mejorar la solución inicial. Nuestro concepto de vecindad para este problema consiste en eliminar un vértice del conjunto, e insertar los adyacentes a éste mientras el conjunto siga siendo independiente. Cómo se asume que la heurística constructiva utilizada genera un conjunto independiente maximal, sabemos que los únicos vértices que podrían agregarse al conjunto son los vértices adyacentes del eliminado.

Si bien el tamaño máximo de la vecindad de un conjunto es igual a su cardinal, el parámetro `TAM_VECINDAD` (valor porcentual sobre el tamaño máximo) determina el tamaño de la vecindad que genera el algoritmo. Para la selección de vecinos que forman parte de la vecindad, se ordena la lista de vértices bajo un criterio arbitrario (como en el caso de la heurística constructiva), que en este caso es *mayor grado del nodo eliminado*. La elección de este criterio se debe a la noción de que uno desea agregar la mayor cantidad de vértices posible.

Una vez que se genera la vecindad, el algoritmo puede *visitar* ciertos vecinos y volver a realizar una búsqueda en profundidad sobre ellos. De esta manera, el espectro de posibilidades aumenta considerablemente, a costa de una mayor complejidad temporal. La cantidad de vecinos que se visitará está definido por `CANT_VECINOS_A_VISITAR`, un valor porcentual sobre el tamaño de la vecindad generado. La cantidad de niveles que realiza la búsqueda en profundidad está definida por el parámetro `NIVEL_PROFUNDIDAD`. Una vez realizada esta búsqueda, el algoritmo elige el *camino* que lleva a la solución de mejor peso, finalizando la iteración de búsqueda local.

4.2. Pseudocódigo

```
BÚSQUEDALOCAL(G: Grafo) → res: conjunto
 1  var actual ← construirSolucionInicial(G)
 2  var mejor ← actual
 3  var iterSinMejorar ← 0
 4  while i < CANT_MAX_ITER and iterSinMejorar < CANT_MAX_ITER_SIN_MEJORAS
 5        var nueva ← buscar(G, actual, NIVEL_PROFUNDIDAD)
 6        if peso(nueva) > peso(mejor) then
 7           mejor ← nueva
 8           iterSinMejorar ← 0
 9        else
10           iterSinMejorar ← iterSinMejorar + 1
11        end if
12        actual ← nueva
13        i ← i + 1
14  end while
15  return mejor

BUSCAR(G: Grafo, sol: conjunto, nivelProfundidad: entero) → res: conjunto
 1  var vecindad ← armarVecindad(G, sol)
 2  if nivelProfundidad > 0 then
 3     for i in [1..CANT_VECINOS_A_VISITAR]
 4           vecindad[i] ← buscar(G, vecindad[i], nivelProfundidad − 1)
 5     end for
 6  end if
 7  return pesoMax(vecindad)

ARMARVECINDAD(G: Grafo, S: conjunto) → res: arreglo de conjuntos
 1  var vecinos ← ordenarLuegoSeleccionar(S, TAMAÑO_VECINDAD)
```

```
2  var vecindad ← arreglo vacío
3  var i ← 0
4  for each v in vecinos
5      var C ← S \ {v}
6      for each w in N(v)
7          if ¬hayConflicto(C, w) then
8              C ← C ∪ {w}
9          end if
10     end for
11     vecindad[i] ← C
12     i ← i + 1
13 end for
14 return vecindad
```

4.3. Detalles de implementación

- El grafo de entrada está representado con listas de adyacencias y un arreglo de pesos, al igual que el algoritmo exacto y la heurística constructiva.
- Durante el proceso de búsqueda local, el conjunto solución se almacena en un arreglo de booleanos de tamaño n, dado que permite aplicar *cambios* como eliminar e insertar vértices en tiempo constante. También se almacena el peso de la solución, para hacer comparaciones en tiempo constante.
- Los conjuntos vecinos son representados como *cambios*, y se almacenan en memoria como una tupla $(entero, lista < entero >)$ que representan el vértice eliminado y la lista de vértices insertados. Al igual que el conjunto solución, se almacena el peso de la solución si se aplicara el cambio. Un *camino* o lista de cambios se almacena como una lista de estas tuplas. Cuando se aplica una serie de cambios, se itera la lista y se insertan/eliminan los vértices en el orden correspondiente.
- El algoritmo de sorting utilizado es Introsort, que tiene complejidad $O(n \log n)$ como ya mencionamos en la sección anterior.

4.4. Cálculo de complejidad en el peor caso

Para realizar el análisis de complejidad temporal del algoritmo, se comenzará estudiando las funciones auxiliares, culminando en la función principal de búsqueda local. Como el parámetro `NIVEL_PROFUNDIDAD` afecta considerablemente la complejidad del algoritmo, será considerado en el análisis como la variable h. Si bien este parámetro es fijo, su valor no está definido aún. A partir de las pruebas, el análisis de calidad y las conclusiones obtenidas, se definirá un valor para este parámetro (y para los demás).

armarVecindad

1 La función *ordenarLuegoSeleccionar* en principio ordena los vértices del conjunto bajo un criterio arbitrario, y luego selecciona los vértices candidatos a ser vecinos, según el parámetro `TAM_VECINDAD`. Como se mencionó en **Detalles de implementación**, el ordenamiento se puede realizar una sóla vez al principio de la búsqueda (porque siempre se ordenan los mismos vértices y el criterio no cambia), pero la selección sí se realiza (pues el tamaño de la vecindad depende del tamaño del conjunto), con lo cual, el costo es $O(n)$.

5 Eliminar un vértice del conjunto es $O(1)$ porque es un arreglo.

7 $hayConflicto$ recorre la lista de adyacencias del vértice w y busca todos los elementos del conjunto, por lo tanto el costo es $O(d(w))$.

6-10 El ciclo recorre los adyacentes del vértice v eliminado para intentar agregarlos al conjunto. Como el costo de $hayConflicto$ es $O(d(w))$ y w es distinto en cada iteración, la complejidad de este ciclo en el peor caso es $O(m)$.

4-13 Es el ciclo principal que arma la vecindad, itera por cada vértice seleccionado a formar parte de la vecindad. La complejidad es $O(n \times m)$.

Entonces, la complejidad del armado de vecindad es $\mathbf{O(n \times m)}$ en el peor caso.

buscar

Siendo una función recursiva, se estudiará primero el caso base, con lo cual, el parámetro de la función $nivelProfundidad$ es 0.

1 Como vimos recién, el costo del armado de una vecindad es $O(n \times m)$.

7 Esta función recorre la vecindad en busca del conjunto de peso máximo, su costo es $O(n)$.

Podemos observar que el costo de llamar la función en el caso base es $O(n \times m)$, el costo del armado de la vecindad. Si $h = 1$, se arma una vecindad, y luego se arma la vecindad de cada vecino, y la complejidad pasa a ser $O(n \times m + n(n \times m)) = O(n^2 m)$. Si $h = 2$, se arman las vecindades en otro nivel de profundidad y luego es $O(n^3 m)$. Por lo tanto, la complejidad de **buscar** es $\mathbf{O(n^{h+1}m)}$, **de orden polinomial** si tenemos en cuenta que h en realidad va a ser una constante.

búsquedaLocal

1 Se construye una solución usando la heurística constructiva que presentamos en la sección anterior. Como vimos, su complejidad es $O(n \log n + m)$.

***** Si bien no está presente en el pseudocódigo de esta función, como mencionamos en la función **armarVecindad**, antes de entrar en el ciclo principal de búsqueda local, se realiza el sorting de los vértices para la selección de candidatos a formar parte de la vecindad. Su costo es $O(n \log n)$.

Luego de la inicialización de estructuras de datos, comienza el ciclo principal de búsqueda local. Este ciclo realiza cierta cantidad de llamadas a la función **buscar**, acotada por la constante `CANT_ITER_MAX`. Luego, se compara la solución generada por la búsqueda, y la última mejor solución, y se queda con la mejor.

Por lo tanto, la complejidad de esta función es $O(n \log n + m + n^{h+1}m) = \mathbf{O(n \log n + n^{h+1}m)}$.

4.4.1. Complejidad en función del tamaño de la entrada

Al igual que en la heurística constructiva, la entrada es el grafo con sus listas de adyacencia y su lista de pesos para cada vértice, luego la entrada es la variable n que determina la cantidad de nodos del grafo, seguido de las listas de adyacencias de cada nodo y su peso. Entonces el tamaño de entrada es:

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

Entonces tenemos que $E(n,m) \geq n+m$, pero como n y m son enteros positivos, también vale que $E(n,m) \geq n$ y $E(n,m) \geq m$. Por lo tanto, la complejidad en función del tamaño de entrada es:

$$T(n,m) \in O(n \log n + n^{h+1}m) = O(E(n,m)\log(E(n,m)) + E(n,m)^{h+2})$$

Como $h > 0$, concluimos que $T(n,m) \in \mathbf{O(E(n,m)^{h+2})}$, de **orden polinomial** (si tenemos en cuenta que h es constante).

4.5. Peores casos

Para encontrar un caso en el cual la búsqueda local no pueda armar un vecino que mejore la suma de pesos del conjunto independiente, aprovechamos el factor goloso en la generación de vecindad.

Dado que la vecindad es generada respecto del grado de los vértices del conjunto (genera vecinos para un porcentaje de los nodos empezando por aquellos que tengan el mayor grado), buscamos un grafo que esté conformado por una vasta cantidad de nodos de grado alto y unos pocos nodos aislados con grados estrictamente menores. La idea es que los pesos más grandes estén almacenados en los nodos de menor grado y que el método constructivo tome una mala decisión al colorear estos nodos. Normalmente la búsqueda local es la encargada de mejorar las "malas elecciones" de la constructiva, pero en este caso, por encontrarse entre los nodos de menor grado, no entran en su rango de acción y son ignorados.

Siguiendo esa idea, elegimos empezar por un árbol binario porque podemos clasificar claramente la distribución de grados: 1 para las hojas, 2 para la raíz y 3 para los nodos internos. Luego buscamos hacer más grande esa diferencia y para eso agregamos aristas. Probamos armar a partir de los nodos de altura 3 en adelante, subgrafos del tipo H (*hole*) conformados por aquellos nodos que compartan la altura. En el grafo resultante

(ver figura 11), la distribución de grados pasó a ser: 3 para los nodos que antes eran hojas y para los nodos de altura 2, 2 para la raíz, y 5 para todos los demás nodos internos. Entonces si queremos ordenar todos los nodos por mayor grado (como lo hace la búsqueda local), los nodos internos van a aparecer en los primeros lugares, luego los nodos de la periferia y los de altura 2 y finalmente la raíz. Tenemos entonces parte de nuestra idea ya armada y queda pendiente asignar los pesos de forma tal que los nodos de grado menor sean los más importantes.

Una consideración que no pudimos dejar de lado es que para la mayor parte de estos casos en los que la busqueda local podría fallar, la constructiva devuelve una buena solución y el resultado deja de ser malo. Entonces distribuimos los pesos y "dirigimos" a la constructiva para que arme una solución lejana a la óptima. Finalmente tenemos un caso armado tanto para que la búsqueda local no funcione como para que la constructiva devuelva un valor lejano a la solución óptima.

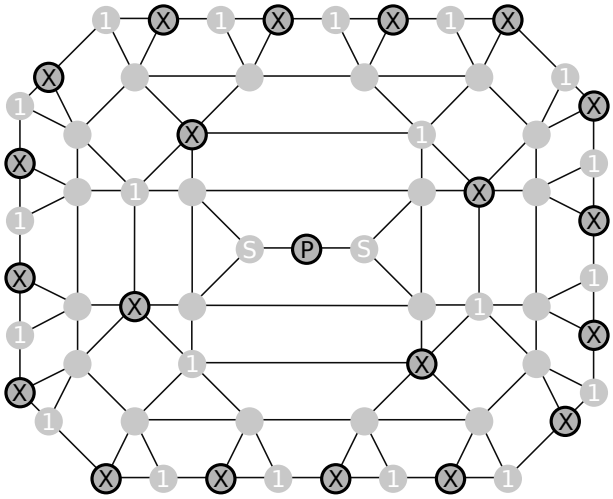


Figura 11: *Este es un grafo que coincide con la descripción anterior. En particular es de altura 6.*

¿Por qué es un mal caso?

Siguiendo con las indicaciones descriptas en los párrafos anteriores podríamos plantear la siguiente distribución de pesos: el nodo P pesa 1981; cada nodo S pesa 1980; cada nodo X pesa 100 y los demás nodos pesan todos 1. Con esa distribución la solución constructiva por colores va a devolver como CIPM al conjunto formado por el nodo P y los nodos X como muestra la figura 11. Entre las demás combinaciones posibles de nodos, la que más peso suma es la que se puede ver en color blanco que incluye los dos nodos S y algunos nodos de peso 1. Si comparamos ambos conjuntos podemos ver lo siguiente:

$$X = 100 \; ; \; P = 1981 \; ; \; S = 1980$$
$$CIPM \text{ } segun \text{ } Constructiva \; = \; 20X + P = 2000 + 1981 = 3981$$
$$Mayor \text{ } conjunto \text{ } restante \text{ } posible \; = \; 20 \cdot 1 + 2S = 20 + 3960 = 3980$$

El algoritmo constructivo siempre va a devolver al conjunto de los 20 nodos X y el nodo P como el de peso máximo. Pero si uno hiciera un pequeño cambio de elección y sustituyera al nodo P por los dos nodos S quedaría un nuevo conjunto formado por 20 nodos X y 2 nodos S sumando:

$$20X + 2S = 2000 + 3960 = 5960$$

Mediante este intercambio obtuvimos el verdadero CIMP para el grafo dado. Es importante tener en cuenta que este es el comportamiento que uno espera que tenga el algoritmo de búsqueda local que programamos ya que intercambia los nodos del primer resultado por sus vecinos. Como sólo lo hace para un porcentaje de los nodos entonces nos fue posible generar este caso. A continuación vemos por qué es que falla:

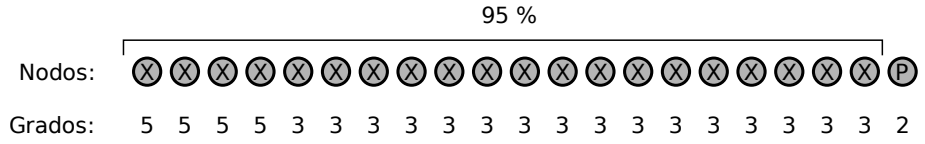


Figura 12: *Nodos de la solución constructiva ordenados según mayor grado.*

Para que el algoritmo de búsqueda local obtenga la solución óptima debería cambiar al nodo P por sus vecinos, y esto sólo lo va a poder hacer si tiene como parámetro de nodos a recorrer un porcentaje mayor al 95 % lo cual es muy poco probable. Sin contar que este análisis es para un grafo de altura 6 y que conforme crece la altura del grafo, la cantidad de nodos X crece exponencialmente mientras que el nodo P siempre es único. Ya que el porcentaje de nodos X respecto del nodo P puede ser tan grande como queramos, el algoritmo de búsqueda local sólo encontraría la solución óptima con el parámetro de porcentaje al 100 %.

4.6. Análisis de tiempo y calidad

Realizar pruebas sobre la heurística de búsqueda local resulta más dificultoso por contar con 5 parámetros configurables que debieron ser probados en combinación para obtener un resultado óptimo:

1. El tamaño de la vecindad, en porcentaje de nodos en el grafo.
2. La cantidad de vecinos analizados, en porcentaje de nodos.
3. La profunidad de búsqueda. Al visitar ciertos nodos de la vecindad, se puede tomar la modificación que se aplica por demarcar un nodo y recorrer la nueva vecindad varias veces, buscando una mejora del peso de la solución. La cantidad de niveles en los que se puede descender es parametrizable, aunque la cantidad debe ser baja, dado que aumenta exponencialmente la complejidad del problema (dado que para cada nodo seleccionado de la vecindad se debe volver a seleccionar y probar una nueva vecindad).
4. La cantidad máxima de iteraciones.
5. La cantidad máxima de iteraciones sin mejora.

4.6.1. Comparación de búsqueda local con parámetros default en grafos Barabasi-Albert

Teniendo esto en cuenta, mostramos los resultados tabulados para una prueba inicial con grafos Barabasi, tomando parámetros intermedios para los valores parametrizables sobre la búsqueda, y permitiendo unas 500 iteraciones, obviando la falta de mejoras. La tabla 5 muestra estos resultados iniciales.

Si bien existe una mejora, vemos que no es demasiado notable en relación con los resultados producidos por el armado constructivo de la solución. Esto es evidencia de que la búsqueda parece quedarse en un máximo local bajo, sin posibilidad de tomar caminos que puedan llegar a rendir mejores resultados. En la siguiente figura vemos cómo, para la búsqueda con $n = 500$, van mejorando los resultados. Dado que todas las búsquedas donde hubieron mejoras llegaron a un máximo en menos iteraciones que esta (por lo general a la quinta se alcanzó), se puede considerar como un ejemplo representativo.

Como se puede apreciar en la figura 13, tras una pocas iteraciones del ciclo de búsqueda se llega a un máximo local y dejan de haber mejoras. Esto puede ocurrir, entre varios motivos, porque la vecindad es demasiado chica o porque no se prueban suficientes alternativas que no rindan buenos resultados en la primera iteración pero puede que mejoren con mayor profundidad y lleven a un mejor máximo local.

4.6.2. Comparación de resultados en grafos con diferentes tamaño de vecindad

Entre los parámetros que podemos probar para mejorar los resultados anteriores se encuentra el tamaño de las vecindades. Aquí analizaremos cómo incide el tamaño de la vecindad para dicho resultado. Tomamos la prueba sobre un grafo Barabasi-Albert de 500 nodos, con todos los parámetros default con la excepción del porcentaje de la vecindad, cuya variación se observa en la tabla 4.6.2.

Los resultados muestran que el aumento de tamaño de vecindades provee mejoras, una vez más observables, pero el costo temporal en proporción a la solución constructiva es muy alto (nótese que en la figura el tiempo de la solución constructiva más grande toma menos de 200 milisegundos, mientras que la máxima mejora de búsqueda local vista aqui se produce con casi 16 segundos de runtime). Luego de hacer el mismo análisis para otros grafos, considerando que la variación de mejoras entre cada grafo individual es alta, decidimos considerar

Resultado de búsqueda local para grafos BA varios			
n	Resultado Const.	Resultado BL	% Mejora
5	284	284	0
10	336	336	0
15	538	563	4.65
20	858	858	0
25	927	927	0
30	1330	1330	0
50	1863	1968	5.64
75	2924	3092	5.75
100	3311	3441	3.93
125	4686	4765	1.69
150	5701	5828	2.23
175	6132	6234	1.66
200	7120	7438	4.47
250	7941	8070	1.62
300	8988	9305	3.53
350	10764	11380	5.72
400	12181	12839	5.4
450	12205	12547	2.8
500	14293	14698	2.83
600	14950	15570	4.15

Cuadro 5: *Con los parametros iniciales, existe una mejora marcada, pero no supera sustancialmente a buenos resultado constructivos.*

Resultados de BL para grafos BA con vecindad variable			
%Vecindad	Resultado	Tiempo	% Mejora
10	14535	1261.44	1.69
20	14654	3167.64	2.53
30	14679	5800.36	2.7
40	14698	8580.07	2.83
50	14698	12120.9	2.83
60	14792	15828.3	3.49
70	14792	19375.7	3.49
80	14792	22815.3	3.49
90	14792	26592.4	3.49
100	14792	29860.8	3.49

Cuadro 6: *Observamos que para mayor cantidad de vecinos hay mejoras escalonadas en los resultados, no obstante sigue constituyendo una proporción muy baja sobre el resultado inicial, e incrementa el tiempo de búsqueda sustancialmente.*

al tamaño óptime de vecindad en un 75 % de los nodos del grafo. Si bien para este valor el costo de búsqueda es relativamente alto, tiene probabilidades satisfactorias de maximizar la mejora que se puede dar con este parámetro.

4.6.3. Comparación de resultados con diferentes cantidades de vecinos analizados

Más allá de la cantidad de nodos que se consideran para evaluar en una vecindad, también se debe tomar en cuenta la cantidad de nodos que se seleccionan para analizar en profundidad. En otras palabras, los nodos que una vez procesados por la busqueda local, persistirán en su estado de selección/deselección y se volverá a buscar otro nodo para deseleccionar, tantas veces como el parámetro de profundidad indique. Para estudiar la efectividad de alterar este parámetro realizamos 4 tablas (7, 8, 9, y 10) para distintos tamaños de vecindades, en un grafo aleatorio Barabasi-Albert de 500 nodos, aumentando progresivamente la cantidad de nodos que se pueden considerar candidatos para procesamiento.

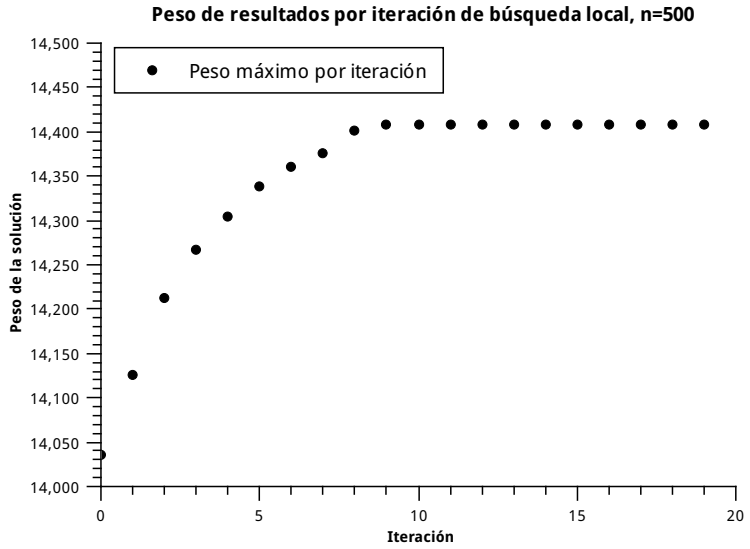


Figura 13: *Observamos que tras una serie de mejoras sucesiva, el algoritmo llegó a un máximo local.*

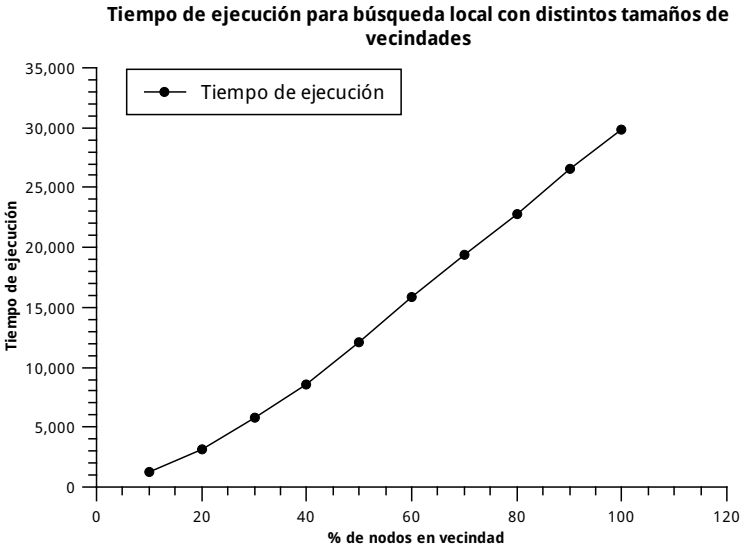


Figura 14: *Observamos que el incremento de la vecindad incrementa el tiempo de ejecución levemente por encima de una curva lineal.*

En la tabla 11 producimos los promedios de mejora para los mismos casos, pero utilizando 10 casos de grafos aleatorios BA con $n = 100$, a modo de estudiar el comportamiento generalizado del algoritmo para dichos parametros sin quedarse en un grafo específico que no necesariamente es representativo.

4.6.4. Comparación de resultados en grafos con diferentes niveles de profundidad de búsqueda

La posibilidad de buscar en profundidad nos permite probar más alternativas que no aparentan ser óptimas a primera vista, no obstante al realizar otras sustituciones pueden obtener mejores resultados. Aquí hacemos una comparación entre los resultados obtenidos en las pruebas anteriores y los mismos con dos niveles de profundidad de búsqueda. Cabe aclarar que graficar la diferencia temporal entre ambas resulta dificultoso o imposible, dado que difieren por varios órdenes de magnitud.

En la tabla 12 vemos los resultados para los mismos parámetros planteados en la tabla 11 en nivel de profundidad 2. Ahi vemos que existe una mejora, pero debemos también considerar el tiempo de ejecución de

Resultados de BL con cant. variable de vecinos, vecindad 25 %				
%Vecinos	Resultado const.	Resultado BL	Tiempo	% Mejora
10	13887	14533	79.05	4.65
20	13887	14533	131.13	4.65
30	13887	14533	183.5	4.65
40	13887	14578	227.24	4.98
50	13887	14578	287.29	4.98
60	13887	14578	324.71	4.98
70	13887	14578	383.48	4.98
80	13887	14578	415.41	4.98
90	13887	14578	514.91	4.98
100	13887	14578	531.75	4.98

Cuadro 7: *La vecindad pequeña limita la máxima mejora que se puede obtener.*

Resultados de BL con cant. variable de vecinos, vecindad 50 %				
%Vecinos	Resultado const.	Resultado BL	Tiempo	% Mejora
10	13887	14869	213.75	7.07
20	13887	14869	375.87	7.07
30	13887	14886	493.56	7.19
40	13887	14886	651.72	7.19
50	13887	14886	838.78	7.19
60	13887	14886	1021.25	7.19
70	13887	14886	1140.88	7.19
80	13887	14886	1317.54	7.19
90	13887	14886	1447.03	7.19
100	13887	14886	1698.6	7.19

Cuadro 8: *Los nodos buscados hacen muy poca mejora en relación al incremento de vecindad.*

Resultados de BL con cant. variable de vecinos, vecindad 75 %				
%Vecinos	Resultado const.	Resultado BL	Tiempo	% Mejora
10	13887	14913	378.82	7.39
20	13887	14978	632.35	7.86
30	13887	14978	940.01	7.86
40	13887	14978	1293.3	7.86
50	13887	15007	1617.36	8.07
60	13887	15007	1969.93	8.07
70	13887	15007	2177.19	8.07
80	13887	15007	2438.35	8.07
90	13887	15007	2956.39	8.07
100	13887	15007	3279.44	8.07

Cuadro 9: *Una vez más vemos que el efecto de la cantidad de vecinos seleccionados para analizar en profundidad es limitado.*

esta solución para ver si el tradeoff entre la mejora *vs.* el tiempo adicional necesario para obtenerlo.

En la figura 15 observamos cómo varia el promedio de los tiempos cuando agregamos un nivel más de profundidad para la búsqueda local. Efectivamente, vemos un incremento en complejidad que no escala de la misma forma que con profundidad 1. Tomando esto en consideración, llegamos a la conclusión que utilizar un nivel de profunidad es óptimo, dado que con el mayor tiempo necesario para nivel 2 se podrían correr más iteraciones constructivas y de búsqueda cuando se utilicen en el GRASP.

Resultados de BL con cant. variable de vecinos, vecindad 100 %				
%Vecinos	Resultado const.	Resultado BL	Tiempo	% Mejora
10	13887	14913	493.61	7.39
20	13887	14978	900.63	7.86
30	13887	14978	1343.78	7.86
40	13887	15007	1870.39	8.07
50	13887	15007	2383.81	8.07
60	13887	15007	2735.83	8.07
70	13887	15007	3195.82	8.07
80	13887	15007	3776.18	8.07
90	13887	15007	4088.49	8.07
100	13887	15007	4570.1	8.07

Cuadro 10: *Pasando un cierto tamaño de vecindad, se llega a un máximo local.*

Promedio de BL con vecinos y vecindades variables				
% Procesados	Tamaño vecindad			
	25.00 %	50.00 %	75.00 %	100.00 %
10	3.37	4.57	4.94	5.08
20	3.39	4.79	5.31	5.39
30	3.46	4.92	5.39	5.42
40	3.51	5	5.42	5.46
50	3.52	5	5.44	5.53
60	3.62	5.13	5.53	5.53
70	3.62	5.13	5.53	5.53
80	3.62	5.13	5.53	5.53
90	3.62	5.13	5.53	5.53
100	3.62	5.13	5.53	5.53

Cuadro 11: *Para el promedio de casos también se llega a un tope de crecimiento con ambos parámetros lo suficimiente grandes.*

Promedio de BL con 2 niveles de búsqueda				
% Procesados	Tamaño vecindad			
	25.00 %	50.00 %	75.00 %	100.00 %
10	3.37	4.39	5.13	5.35
20	3.46	5	5.51	5.66
30	3.57	5.13	5.77	5.74
40	3.65	5.26	5.74	5.92
50	3.66	5.29	5.9	6.14
60	3.64	5.42	6.13	6.14
70	3.72	5.45	6.14	6.14
80	3.83	5.45	6.14	6.14
90	3.83	5.48	6.14	6.14
100	3.85	5.51	6.14	6.14

Cuadro 12: *Se observa una mejora adicional por sobre la búsqueda de primer nivel, sin embargo no deja de ser chica.*

4.6.5. Comparación de resultados con distintas cantidades de iteraciones sin mejoras

La cantidad de iteraciones que podemos tolerar sin mejoras depende fuertemente del comportamiento que toman los gráficos analizados. Para ello queremos ver cuántas iteraciones suelen tomar para alcanzar un máximo cuando se los prueba con una cantidad grande (500) de iteraciones. En la figura 16 tomamos los resultados de todos los grafos Erdos-Renyi y Barabasi-Albert que fueron analizados para las pruebas de búsqueda local, y para cada uno se obtuvo el primer número de iteración en el cual se llegó al máximo. Cada barra es la cantidad de grafos analizados cuyo máximo se alcanza en esa iteración. Es evidente que se sigue una distribución normal. Habiendo analizado 790 grafos diferentes, consideramos satisfactorio como criterios se detención unas

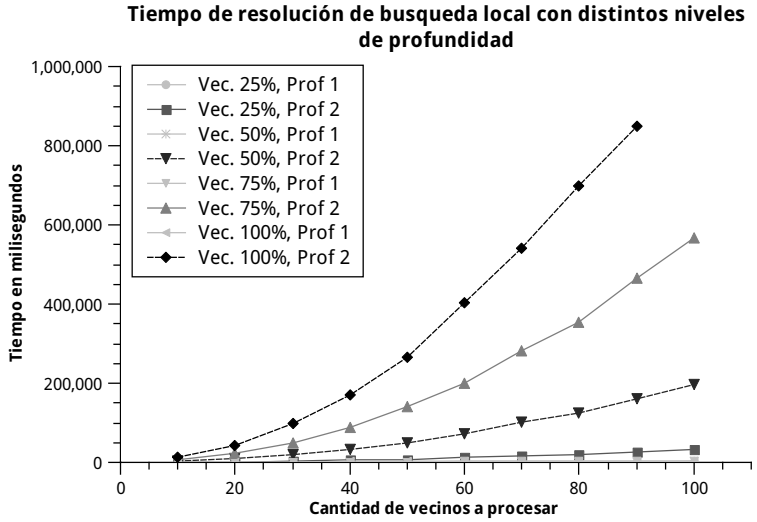


Figura 15: *Vemos un incremento polinómico en el tiempo de resolución al incrementar la profundidad en 1.*

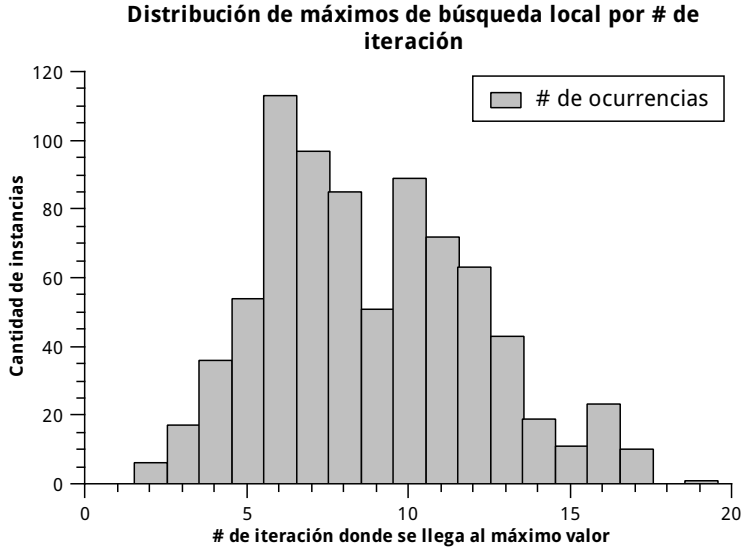


Figura 16: *Se sigue una distribución normal para las iteraciones en las que se llega a un máximo.*

100 iteraciones en total, con hasta 20 iteraciones sin mejoras. Se podría decir que estas cifras son un tanto conservadoras, pero el tiempo de ejecución para cada caso es muy reducido, y a la vez contempla todos los casos que realísticamente podrían ser analizados en la práctica.

4.6.6. Comparación de resultados para distintos tamaños de grafos

Habiendo encontrado ya los parámetros más óptimos en las pruebas anteriores, procedemos a analizar una vez más grafos de distintos tamaños, de tipo Erdos-Renyi, diferenciando entre grafos esparsos ($p = 2\,\%$) y densos ($p = 20\,\%$).

Los resultados nos muestran la relevancia que tiene la densidad de los grafos a la hora de encontrar buenas mejoras por búsqueda local. En general, los grafos esparsos se vieron poco beneficiados por la búsqueda en relación a los grafos densos, donde cada nodo adicional seleccionado cuenta importantemente hacia los resultados. Nótese, sin embargo, que la mejora sigue siendo dependiente de la instancia de grafo que se está probando, por lo que no se puede garantizar que una corrida de búsqueda local pueda mejorar los resultados sustancialmente a la que se obtendría solo corriendo una heurística constructiva.

Resultados de búsqueda local para grafos ER esparsos				
N	Resultado Const.	Resultado BL.	Tiempo	% Mejora
10	305	305	45.64	0
20	1103	1103	136.33	0
25	1165	1165	185.01	0
30	1502	1502	246.27	0
50	2216	2218	520.27	0.09
75	2778	2836	1045.65	2.09
100	3731	3795	1719.88	1.72
125	4740	4850	2626.34	2.32
150	5125	5423	3414.09	5.81
175	6004	6005	4137.85	0.02
200	6621	6764	5848.49	2.16
250	7186	7922	8757.26	10.24
300	8685	9344	11034.2	7.59
350	9568	10450	13696.4	9.22
400	10346	11077	17623.9	7.07
450	10823	11523	21432.8	6.47
500	11162	11973	25189	7.27
600	12789	14073	36194	10.04

Cuadro 13: *Vemos que los resultados obtenidos siguen dentro del márgen de lo esperado, pero en general son mejores que con los parámetros que fueron probados por defecto antes de conocer los valores óptimos.*

Resultados de búsqueda local para grafos ER densos				
N	Resultado Const.	Resultado BL.	Tiempo	% Mejora
5	205	205	21.21	0
10	323	323	28.73	0
15	266	266	55.27	0
20	511	511	70.96	0
25	640	640	106.97	0
30	532	598	131.63	12.41
50	785	785	298.65	0
75	985	1145	677.91	16.24
100	1052	1141	818.26	8.46
125	1211	1257	1454.93	3.8
150	1321	1384	2442.47	4.77
175	1377	1505	2584.31	9.3
200	1404	1546	3856.93	10.11
250	1515	1604	6577.8	5.87
300	1618	1661	7763.61	2.66
350	1702	1950	15396.6	14.57
400	1834	1924	10389.3	4.91
450	1733	1902	24524.1	9.75
500	1795	2126	19681.6	18.44
600	2012	2151	51595.5	6.91

Cuadro 14: *Para grafos más densos la solución obtenida es más chica, por lo que seleccionar nodos adicionales con la búsqueda local tiene mayor importancia.*

4.7. Parámetros finales

Después de los análisis realizados, decidimos adoptar los siguientes parámetros para utilizar en GRASP:

1. Tamaño de vecindad: 75 %
2. Cantidad de vecinos procesados: 75 %
3. Nivel de profundidad de análisis: 1
4. Cantidad de iteraciones: 50
5. Cantidad de iteraciones sin mejora: 25

Si bien algunos de estos parámetros se pueden probar con valores distintos dentro de GRASP, los análisis que hicimos nos dan criterios judiciosos sobre cuales valen la pena ajustar, y cuales serán fijados por tener efectos que no mejoran los resultados.

4.8. Investigación adicional

Dado el tamaño del trabajo práctico y la profunidad con la que se podría investigar cada uno de los temas vistos, resultó imposible seguir todas las lineas de investigación y búsqueda que hubieramos deseado analizar. No obstante, de haber tenido tiempo para probar otras cosas, hubieramos visto los siguientes temas:

- Una comparación más detallada de la efectividad de la búsqueda local dependiendo de la densidad de un grafo, viendo valores de densidad más granulares que los que fueron analizados.
- Análisis de búsqueda local en grafos chicos donde se puede obtener una solución exacta. Más allá de que no sea parte del caso de uso de la heurística, creemos que es un tema de interés.
- Probar la efectividad de la búsqueda con distribuciones distintas de grados para los nodos (por ejemplo, una distribución normal o exponencial negativa), y con mayor y menor rango de pesos posibles.
- Analizar grafos donde exista una correlación entre los grados de un nodo y su peso. Esto modelaría con mayor precisión el comportamiento de redes de distribución reales, donde un nodo distribuidor tiene un valor de capacidad de distribución generalmente relacionado a cuántos nodos periféricos puede proveer. Por la dificultad de crear un generador de grafos de esta índole, y por el hecho de que introduce muchas restricciones arbitrarias para otros casos de grafos, no la consideramos por su especificidad.
- Aplicar otro concepto de vecindad, como intentar marcar un nodo dentro del conjunto independiente y restar los que estén en conflicto. Por motivos de tiempo se tuvo que decidir a priori entre un criterio u otro, y por los casos desarrollados a mano, llegamos a la conclusión de que la eliminación de nodos del conjunto daría mejores resultados en general que la suma de ellos.

5. Metaheurística GRASP

5.1. Explicación

GRASP (Greedy Randomized Adaptive Search Procedure) es una metaheurística que consiste en sucesivas construcciones de soluciones que luego son mejoradas a través de una búsqueda local. Se utiliza como heurística constructiva una adaptación del algoritmo ya presentado, que contempla la selección de vértices de forma aleatoria dentro de una **lista restringida de candidatos** (RCL). El tamaño de la RCL depende del parámetro `TAM_RCL`, valor porcentual sobre n. También se utiliza la misma heurística de búsqueda local presentada en la sección anterior, con la única diferencia que ésta recibe una solución ya construida. La cantidad de iteraciones del ciclo está acotada por el parámetro `CANT_ITER_GRASP`.

La diferencia entre la constructiva original y la implementada en este algoritmo se encuentra en la manera de ordenar los vértices para luego colorearlos. Primero se los ordena según el mismo criterio de orden original (mayor peso y menor grado). Luego comienza un ciclo en el que, en cada iteración, se genera la RCL y se selecciona un candidato de manera aleatoria. Las sucesivas selecciones de vértices formarán el arreglo final que determinará el orden utilizado en el coloreo goloso.

5.2. Pseudocódigo

```
GRASP(G: Grafo)→ res: conjunto
 1  var mejor ← ∅
 2  while i < CANT_ITER_GRASP
 3        var nueva ← constructiva(G)
 4        nueva ← busquedaLocal(G, nueva)
 5        if peso(nueva) > peso(mejor) then
 6           mejor ← nueva
 7        end if
 8        i ← i + 1
 9  end while
10  return mejor

CONSTRUCTIVA(G: Grafo)→ res: conjunto
 1    { Ordena los vértice por mayor peso y menor grado. }
 2  nodos ← ordenarNodos(G)
 3    { Reordena los vértices de acuerdo a una selección aleatoria. }
 4  for i in [0..n − 1]
 5        var numCandidatos ← ⌈((n − i) × TAM_RCL) / 100,0⌉
 6        var elegido ← rand(0, numCandidatos − 1) + i
 7          { Intercambia el vértice elegido con el original. }
 8        var temp ← nodos[i]
 9        nodos[i] ← nodos[elegido]
10        nodos[elegido] ← temp
11  end for
12    { Finalmente, colorea el grafo en el orden propuesto. }
13  return greedyColoring(G, nodos)
```

5.3. Cálculo de complejidad

constructiva

2 El ordenamiento de los vértices según mayor peso y menor grado es $O(n \log n)$.

6 Se asume que la generación de un número pseudoaleatorio es $O(1)$.

4-6 El ciclo que reordena los vértices del arreglo en la línea 2 itera n veces, con lo cual el costo es $O(n)$.

13 Según el análisis de complejidad de la constructiva original, el costo del coloreo es $O(n + m)$ (no se tiene en cuenta el ordenamiento inicial de vértices).

Por lo tanto, la complejidad de **constructiva** en el peor caso es $\mathbf{O(n \log n + m)}$.

GRASP

Como el algoritmo de búsqueda local no fue modificado, su complejidad sigue siendo $O(n \log n + n^{h+1}m)$, donde h es el nivel de profundidad. El algoritmo construye una solución y la mejora a través de la búsqueda local, una cantidad de veces fija. Luego, podemos concluir que la complejidad de **GRASP** es $O\big((n \log n + m) + (n \log n + n^{h+1}m)\big) = \mathbf{O(n \log n + n^{h+1}m)}$.

5.3.1. Complejidad en función del tamaño de entrada

Puesto que la complejidad temporal del algoritmo es la misma que la de búsqueda local, y la entrada sigue siendo el grafo pesado, el análisis es exactamente el mismo que fue presentado en la sección anterior.

$$T(n, m) \in \mathbf{O(E(n, m)^{h+2})}$$

5.4. Análisis de calidad

5.4.1. Comparación con distintos tamaños de lista de candidatos

Iniciamos el análisis de GRASP tomando los parámetros por defecto de búsqueda local, usando una lista de candidatos del 20 % con un máximo de 50 ciclos de GRASP. Realizamos las pruebas sobre todas las instancias generadas de grafos Barabasi-Albert, con n entre 10 y 500, y calculamos el promedio de los pesos totales de 10 muestras, comparando con los resultados obtenidos por la búsqueda local y por la solución constructiva.

Comparación de promedios de resultados						
N	Constructiva	BL	GRASP	Tiempo prom	% Mejora Const.	% Mejora BL
10	347.8	356.3	356.3	75.35	2.44	0
20	744	770.6	770.6	254.4	3.58	0
30	1146.3	1186.7	1186.7	519.22	3.52	0
50	1931.9	1986	1987.7	1374.28	2.89	0.09
75	2925	3000.7	3008.3	2734.79	2.85	0.25
100	3768.6	3868.6	3885	4831.4	3.09	0.42
125	4723.5	4871	4875.7	7673.26	3.22	0.1
150	5228.9	5396.6	5402.7	10798.6	3.32	0.11
175	6118.9	6353	6372.7	14772.8	4.15	0.31
200	7100.5	7253.1	7261.3	18845.5	2.26	0.11
250	8354.9	8573	8591.6	33291.6	2.83	0.22
300	9271.1	9650.2	9686	43414.6	4.48	0.37
350	10645	11035.9	11122.1	58303.5	4.48	0.78
400	11884.3	12435.9	12543.2	80572.6	5.54	0.86
450	12714.8	13232.8	13325.4	413358	4.8	0.7
500	13727.4	14450	14511.6	132990	5.71	0.43
600	15074.9	15872	16029.7	192814	6.33	0.99

Cuadro 15: *GRASP muestra resultados mejores en promedio pero no por mucho*

Observamos que con los parámetros como están puestos para los valores iniciales, el promedio de mejoras es bajo, aunque se observa que la mejora incrementa para grafos más grandes. Para estudiar cómo se pueden alterar los parámetros de búsqueda para rendir mejores resultados, estudiamos la distribución de ciclos de mejora de GRASP, análogamente a cómo se estudiaron los ciclos de búsqueda local en la sección anterior.

Como vemos en la figura 17, hay mucha varianza entre el número de ciclo de iteración GRASP donde un grafo deja de mejorar sus resultados. En principio esto indicaría que hay ciertos casos que se podrían beneficiar, aunque sea por poco, extendiendo el número de iteraciones. No obstante, el gráfico no discrimina entre el tamaño de los grafos que proveyeron. Las figuras 18 y 19 producen el mismo análisis para grafos con 50 nodos o menos y para grafos con 250 nodos o más.

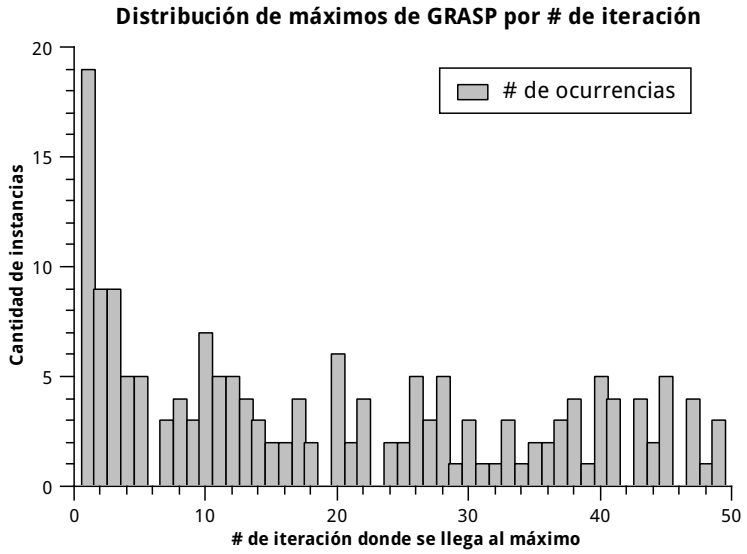


Figura 17: *La distribución de número de ciclo en el que se alcanza un máximo tiene un máximo en valores bajos pero es uniforme aun en valores más grandes.*

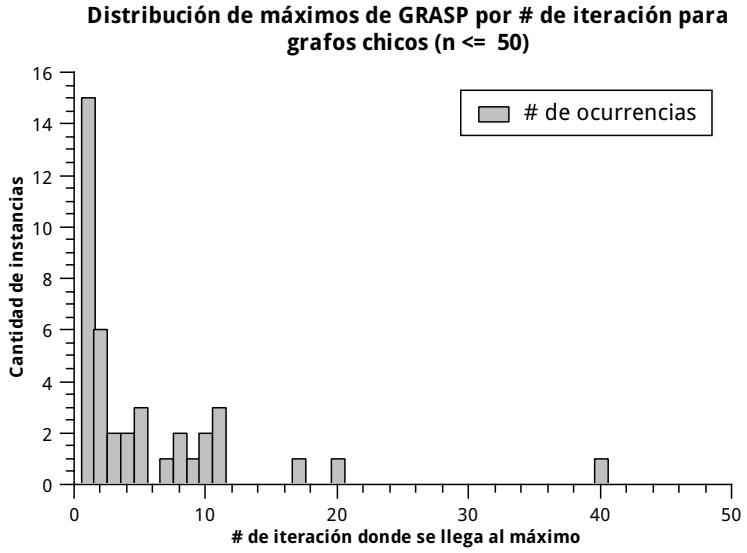


Figura 18: *Los grafos más chicos encuentran rápidamente un máximo local.*

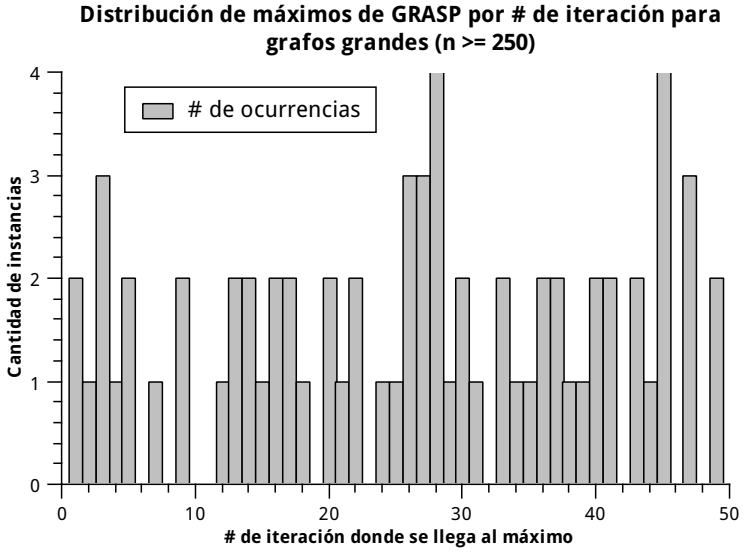


Figura 19: *Grafos más grandes exhiben comportamientos más variados.*

Las gráficas indican que los grafos chicos tienen una clara tendencia a exhibir su mejor caso rápidamente. Sin embargo, para grafos grandes, la distribución de iteraciones máximas resulta uniforme, y es posible extrapolar que si se corrieran más iteraciones que las que se permitieron, se podrían mejorar los resultados. Más allá de que sea posible, se debe considerar si el costo temporal de aumentar la cantidad de iteraciones vale la pena en relación a las mejoras obtenidas. Como vimos en la tabla 15, las mejoras que obtuvo GRASP en relación a una búsqueda local determinística con muy bajas.

Dado que los tiempos de prueba para una gran cantidad de iteraciones tienen un costo temporal prohibitivamente alto como para reproducir toda la batería de realizada, mostramos los casos de comparación para 10 grafos Barabasi-Albert con $n = 500$, y 10 grafos Erdos-Renyi con densidad $p = 2\,\%$ y 10 con densidad $p = 20\,\%$, corriendo con 100 iteraciones GRASP. No se contemplan los casos con grafos más chicos, no obstante consideramos que resulta de mayor interés hacer el análisis con grafos grandes. La tabla 16 muestra los resultados para el análisis de la instancias BA, la 17 muestra los mismos para las instancias Erdos-Renyi densas, y la 18 los muestra para los ER esparsos.

Resultados de GRASP para 10 instancias de grafos BA					
Instancia	Peso const,	Peso BL	Peso iter 50	Peso iter 100	% Mejora 50 a 100 ciclos
1	12480	13145	13154	13154	0
2	11688	12540	12682	12690	0.06
3	11551	11771	11828	11893	0.55
4	11930	12602	12689	12689	0
5	12682	13188	13306	13310	0.03
6	11193	11979	12303	12303	0
7	11465	12507	12811	12811	0
8	12848	13302	13404	13404	0
9	11162	11973	12190	12278	0.73
10	11831	12325	12389	12401	0.1

Cuadro 16: *Los grafos Barabasi-Albert no se benefician sustancialmente con más ciclos.*

Resultados de GRASP para 10 instancias de grafos ER densos					
Instancia	Peso const.	Peso BL	Peso iter 50	Peso iter 100	% Mejora 50 a 100 ciclos
1	1792	1924	2170	2170	0
2	1897	1934	2323	2323	0
3	1862	2011	2144	2296	7.56
4	1786	1928	1984	2093	5.65
5	1760	1953	1869	2097	11.67
6	1804	2031	1926	2161	11.57
7	1924	1986	1923	2239	15.91
8	1823	1984	1866	2189	16.28
9	1795	2126	2000	2263	12.37
10	1875	1917	2176	2327	7.88

Cuadro 17: *Los grafos más densos tienen importantes mejoras.*

Resultados de GRASP para 10 instancias de grafos ER esparsos					
Instancia	Peso const.	Peso BL	Peso iter 50	Peso iter 100	% Mejora 50 a 100 ciclos
1	13879	15326	15382	15382	0
2	13911	14511	14525	14525	0
3	13535	13966	13903	14043	1
4	12947	13699	13569	13705	0.99
5	14453	15282	15314	15330	0.1
6	13701	13958	14041	14041	0
7	13053	13650	13708	13726	0.13
8	13463	14370	14472	14481	0.06
9	14039	14946	14996	14996	0
10	14293	14792	14943	14943	0

Cuadro 18: *Una vez más, grafos más esparsos no ven grandes beneficios.*

El análisis de esta serie de casos nos permite llegar a dos observaciones de interés:

- Los grafos más grandes se benefician en mayor medida por las iteraciones GRASP. Esto se debe a que estos tienen casos más variados de soluciones constructivas aleatorias, dado que tienen más nodos en su lista restringida de candidatos. Efectivamente, 20 % de los nodos en un grafo de 20 nodos nos da 4 posibles candidatos, mientras que en 500 nodos, la cantidad se expande a 100 candidatos. Entonces, en un grafo chico, ocurre que existe menos variación entre las posibles soluciones constructivas posibles, y además la búsqueda local cubrirá más rápidamente los casos donde puede realizar sustituciones de nodos favorables, por lo que los ciclos de búsqueda local de las primeras iteraciones de GRASP ya habrán pasado por la mayoría de los casos buenos. En cambio, un grafo más grande admite mayor variación de soluciones constructivas, y las posibles permutaciones para explorar que pueden generar mejores máximos son más, pero a la vez se encuentran en vecindades con muchos más nodos. Esto hace que el efecto de probar más combinaciones a través de más iteraciones resulte de mayor beneficio.

- Los grafos más densos se benefician en mayor medida por las iteraciones GRASP. Dado que, en general, los grafos más densos admiten una proporción menor de nodos dentro de su conjunto independiente que un grafo con un mismo n pero menor densidad, el efecto de incluir nodos de mayor peso al conjunto independiente resulta en un aumento relativamente más alto.

Podemos concluir con las tablas anteriores que los grafos Barabasi-Albert son un caso malo para la heurística, dado que no resulta capaz de mejorar los resultados provistos por una búsqueda local determinística. Por motivos prácticos se desconoce en cuánto es mejorable la solución provista en relación a la solución exacta, no obstante el hecho de que GRASP no encuentre mejoras sustanciales, e incluso a veces no llegue a encontrar un resultado tan bueno como la búsqueda local, indica que no tiene sentido aplicar la metaheurística a este caso cuando la simple búsqueda local provee resultados que se pueden considerar como buenos.

5.4.2. Comparación con distintos tamaños de listas restringidas de candidatos

El segundo parámetro a tomar en cuenta es del tamaño de la lista restringida de candidatos que se utiliza en el algoritmo constructivo. Tomamos aquí como ejemplo 10 instancias de grafos Barabasi con $n = 500$, y 10 grafos Erdos-Renyi densos, con $n = 500$ y $p = 20\,\%$‘. Cada instancia corrió 100 ciclos de GRASP. Dado el tiempo necesario para realizar estas pruebas, y el hecho de que los grafos ER esparsos tienen densidad similar a los Barabasi y exhiben un comportamiento parecido para las otras pruebas GRASP, estos fueron excluidos del análisis.

Las tablas 5.4.2 y 20 muestran los resultados obtenidos para las instancias probadas con RCLs que componen el 25 %, 50 %, y 75 %. Para la segunda tabla, realizamos también pruebas con una vecindad de 100 %, lo que equivale a hacer un coloreo aleatorio, para estudiar más a fondo el impacto del determinismo del algoritmo constructivo.

Resultados de GRASP para distintos tamaños de RCL con grafos BA								
# Inst.	HC	HBL	W 25 %	W 50 %	W 75 %	%I 20 %	%I 50 %	%I 75 %
1	13879	15326	15374	15382	15382	0.31	0.37	0.37
2	13911	14511	14515	14490	14518	0.03	-0.14	0.05
3	13535	13966	14043	14061	14012	0.55	0.68	0.33
4	12947	13699	13708	13693	13671	0.07	-0.04	-0.2
5	14453	15282	15312	15314	15327	0.2	0.21	0.29
6	13701	13958	14016	14052	14009	0.42	0.67	0.37
7	13053	13650	13728	13728	13701	0.57	0.57	0.37
8	13463	14370	14481	14432	14426	0.77	0.43	0.39
9	14039	14946	14996	14996	14961	0.33	0.33	0.1
10	14293	14792	14943	14943	14911	1.02	1.02	0.8

Inst	# de instancia
HC	Peso de heurística constructiva
HBL	Peso de heurística de búsqueda local
W n %	Peso de GRASP con RCL de n %
I n %	Porcentaje de mejora de GRASP respecto a búsqueda local

Cuadro 19: *El tamaño de RCL parece no afectar los resultados.*

Nuevamente, por los motivos anteriormente explicados, los grafos de nodos con menor promedio de adyacencias muestran poca o ninguna mejora ante la búsqueda local determinística, mientras que los de mayor densidad muestran un aumento dramático. La densidad parece ser el factor más relevante para el valor de GRASP. Se nota también que en la tabla 20, hay una predominancia de valores máximos cuando la RCL es del 25 % de los nodos, sin embargo 4 de los 10 resultados máximos se encuentran en otros tamaños. Esto quiere decir que la calidad de la solución es también dependiente de las particularidades de cada grafo, a tal punto que lo mejor para un análisis exhaustivo de GRASP probablemente debería incluir distintos tamaños de RCLs.

Resultados de GRASP para distintos tamaños de RCL con grafos ER densos										
# Inst	HC	HBL	W 25 %	W 50 %	W 75 %	W 100 %	I 25 %	I 50 %	I 75 %	I 100 %
1	1792	1924	2257	2138	2122	2187	17.31	11.12	10.29	13.67
2	1897	1934	2251	2161	2270	2198	16.39	11.74	17.37	13.65
3	1862	2011	2296	2102	2170	2089	14.17	4.53	7.91	3.88
4	1786	1928	2114	2081	2129	2079	9.65	7.94	10.43	7.83
5	1760	1953	2107	2070	2073	2122	7.89	5.99	6.14	8.65
6	1804	2031	2197	2173	2173	2087	8.17	6.99	6.99	2.76
7	1924	1986	2133	2105	2080	2042	7.4	5.99	4.73	2.82
8	1823	1984	2198	2139	2104	2156	10.79	7.81	6.05	8.67
9	1795	2126	2150	2182	2296	2249	1.13	2.63	8	5.79
10	1875	1917	2276	2193	2107	2054	18.73	14.4	9.91	7.15

Inst	# de instancia
HC	Peso de heurística constructiva
HBL	Peso de heurística de búsqueda local
W n %	Peso de GRASP con RCL de n %
I n %	Porc. mejora de GRASP respecto a BL

Cuadro 20: *Para Erdos-Renyi densos, la diferencia que hace el tamaño de la RCL es muy sustancial.*

5.4.3. Análisis de tiempos

Analizaremos los parámtros restantes que nos indican cómo crecen los tiempos de resolución: el tamaño del grafo, y el tamaño de la RCL. La figura 20 nos muestra el crecimiento del promedio de tiempo de solución para grafos Barabasi-Albert con n de 10 a 600. Como se puede observar, se exhibe un crecimiento cuadrático que corresponde a la componente n^2m de la cota de complejidad, por lo que se puede considerar que los tiempos de resolución son buenos.

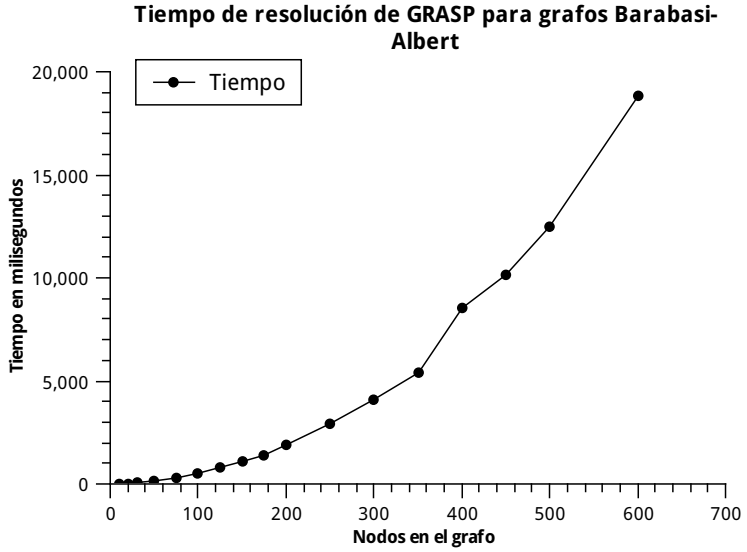


Figura 20: *El incremento de tiempo corresponde a la cota teórica indicada por los cálculos de complejidad.*

El tiempo de resolución para los problemas Erdos-Renyi densos como se ven en la figura 21, obtenidos de las pruebas cuyos resultados aparecen en la tabla 20, van aumentando en forma lineal conforme aumenta el tamaño de la lista restringida de candidatos, debido a que el costo de seleccion de cada nodo escala en un órden de $O(n\log(n))$, pero el grueso del tiempo sigue siendo tomado por la componente de búsqueda local. Esto se debe a que, conforme aumenta el tamaño de la RCL, el resultado de la solución constructiva se asemeja más a un coloreo aleatorio, por lo cual la búsqueda local toma una mayor cantidad de iteraciones encontrando mejoras al resultado original, extendiendo el tiempo antes de que se cumpla el criterio de detención (la cantidad de iteraciones sin mejoras).

La última variable a contemplar dentro de GRASP, la cantidad de iteraciones, tiene una relación claramente lineal entre el número de iteraciones y el tiempo necesario, por lo que no consideramos necesario incluir gráfica

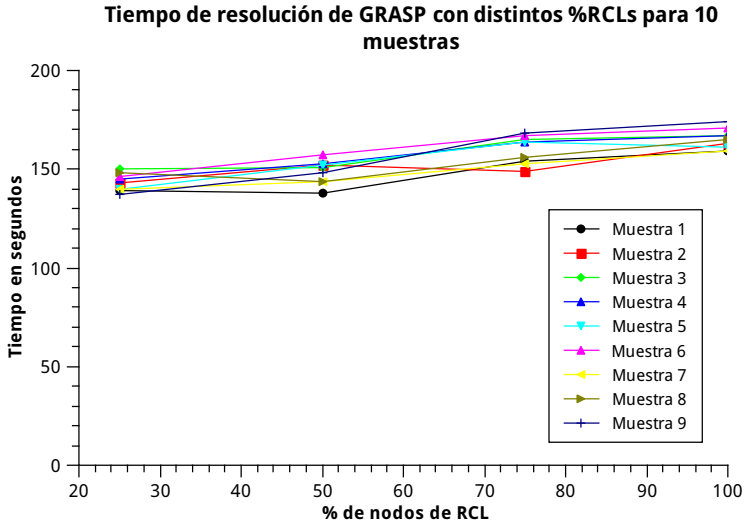


Figura 21: *Al aumentar el tamaño de la RCL se ve una tendencia de incremento lineal en tiempo.*

que apoye esto.

5.5. Parámetros finales

Tras haber hecho los análisis descritos anteriormente, decidimos entregar el resultado con los siguientes parámetros:

1. Parámetros de búsqueda local: Igual que los mencionados en la sección correspondiente de este trabajo práctico.
2. Porcentaje de nodos que componen la RCL: 25 %.
3. Cantidad de iteraciones: 100.

Estos valores no rendirán resultados óptimos para todos los problemas, y serán excesivos para otros, pero para la clase de grafos sobre los que fue probado producen un buen compromiso entre calidad de resultados y runtime, asumiendo que se desea dedicar no más que algunos minutos para resolver una solución grande. Si se admitiera más tiempo, aumentar la cantidad de iteraciones es una buena forma de aprovecharlo.

5.6. Conclusiones

- La heurística constructiva, para casi todos los casos, obtuvo resultados muy buenos, y su tiempo de ejecución fue relativamente muy bajo dentro del contexto del GRASP. Son muy pocos los casos donde la búsqueda local mejoró los resultados constructivos por encima de un 10 %, si bien el tiempo de ejecución de la componente constructiva es relativamente bajo en relación a la búsqueda local.
- No necesariamente el mejor resultado constructivo llevó al mejor resultado de búsqueda local. Este es el motivo por el cual se utiliza una lista de candidatos sobre la cual seleccionar nodos en forma aleatoria.
- Las caracteristicas de la clase de grafo siendo probado suele imponer límites prácticos a cuánto una solución puede ser mejorada por GRASP en relación a una búsqueda local determinística. Dentro de estas clases, el grafo individual sigue siendo el factor más importante a la hora de determinar la mejora que se puede obtener por el GRASP.
- Los grafos densos obtiene mejoras mucho más grandes que los esparsos, a tal punto que para los grafos Barabasi-Albert no hubo una mejora muy apreciable por correr GRASP. Sin embargo, para grafos Erdos-Renyi densos, las mejoras llegaron hasta un 18 % por encima de lo provisto por la búsqueda local.
- El tiempo invertido en GRASP suele seguir la ley de rendimientos decrecientes: una vez que se encuentran las mejoras "fáciles", toma cada vez más tiempo encontrar otras mejoras. Dependiendo del tradeoff tiempo-a-beneficio que se toma, esto puede significar que correr GRASP más allá de unos pocos ciclos no valga la

pena, o que si se desea mejorar el resultado hasta lo máximo factible, se puede dejar corriendo por horas, tomando en cuenta que las mejoras obtenidas sobre casos más simples no superarán un par de puntos de porcentaje en general.

5.7. Investigación adicional

Una vez más los temas desarrollados aquí se pueden expandir enormemente, por lo que listamos algunos de los temas que, con más tiempo, hubieramos investigado:

- Comparación de la solución exacta con grafos grandes y extremadamente densos ($p > 75\,\%$), dado que las pruebas que hicimos en la primera sección mostraron que es factible usar el algoritmo exacto para estos casos.
- Un análisis detallado para balancear los parámetros de GRASP y búsqueda local dado una cantidad limitada de tiempo para resolver un problema. Esto sería aplicar lo conocido a casos específicos. No obstante, por la cantidad de variables a tomar en cuenta, y la variabilidad de factores desconocidos, esto no resulta factible dentro del *scope* de este trabajo.
- Una comparación más granular de tamaños de RCL. Por el tiempo de ejecución de GRASP para las pruebas grandes no resultó factible hacer esto.
- Aplicar el problema para grafos reales, sin depender de la generación aleatoria de los mismos, y compararlos con otras heurísticas. Esto daría un buen criterio para ver la utilidad de GRASP *vs* otras técnicas.