Algoritmos y Estructuras de Datos III

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

Trabajo práctico III

Integrante	LU	Correo electrónico
Delger, Agustín	274/14	agusdel_94@hotmail.com
Delmagro, Nicolás Agustín	596/14	agustin.delmagro@gmail.com
Lopez Valiente, Patricio	457/15	patricio454@gmail.com
Zar Abad, Ciro Román	129/15	ciromanzar@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Informe de Modificaciones

1.1. Informe

1.1.1. Carátula

- Se corrigió el error de tilde en la palabra **cátedra**.

1.1.2. Marco experimental

- Se reemplazó el término eficiencia por eficacia en todo el informe dado que se corresponde con el significado que pretendemos darle en los experimentos.
- Se agregó definición de eficacia.

1.1.3. Backtracking

- Se reescribió análisis de complejidad para brindar mayor claridad en la cantidad de llamados recursivos y los costos de cada uno.
- Se corrigió la descripción del experimento de tiempos en grafos completos, por uno más acorde al análisis empírico de los datos.
- En los experimentos de tiempos se detalló y justificó la elección de la cantidad de mediciones promediadas.

1.1.4. Greedy

- Se agregó experimento de comparación de iteraciones entre GLF y RI para poder explicar la diferencia de tiempos de ejecución entre ambas.

1.1.5. Random

- Se le cambió el nombre al algoritmo Greedy Random por Random Insertion, ya que no era correcto considerarlo un algoritmo goloso, y se lo ubicó en una nueva categoría de heurísticas aleatorias.

1.1.6. CNLS

- Se separó el cálculo de complejidad de algunas funciones auxiliares para facilitar su comprensión.
- Se corrigió el cálculo de complejidad del algoritmo CNLS en la parte en que se analiza una cota para la cantidad de iteraciones que realiza la búsqueda local.
- Se compararon tiempos de cómputo del algoritmo con un polinomio.
- Se demostró que CNLS no mejora con respecto a GLF bajo ninguna instancia.

1.1.7. GRASP

- Se agregó descripción de configuración óptima de GRASP

1.1.8. Grafos particulares

- Se generó una familia de grafos en donde GRASP no encuentra una solución exacta y se experimentó sobre los mismos.

1.1.9. Conclusión

- Se extendió la conclusión.

1.2. Código

- Codigo redistribuido en headers específicos para cada método.
- Se agregan instrucciones de uso en README.
- Se agregó el script correspondiente al nuevo experimento çontarCiclos".

2. Introducción

En este trabajo se analiza el problema de *clique de máxima frontera* (CMF). Para definir en qué consiste este problema, primero es necesario definir el término *clique* y luego su *frontera*.

Dado un grafo simple $G = (V, E)$, un subconjunto de vértices de G es una *clique* si y sólo si éste induce un subgrafo completo de G. Es decir, $K \subseteq V$, tal que $K \neq \emptyset$, es una clique de G si y sólo si para todo par de vértices $u, v \in K, u \neq v$, existe la arista (u, v) en E.

Definimos la *frontera* de una clique K como el conjunto de aristas de G que tienen un extremo en K y otro en $V \setminus K$. Formalmente, la frontera de una clique K queda definida por

$\delta(K) = \{(v, w) \in E / v \in K \wedge w \in V \setminus K\}$.

Una vez definido esto, el problema de clique de máxima frontera (CMF) en un grafo G consiste en hallar una clique K de G cuya frontera δ(K) tenga cardinalidad máxima. Algunas situaciones de la vida real que pueden ser modelados a través de este problema se pueden ver en la sección siguiente.

Si bien CMF es un problema conocido, no se conocen algoritmos polinomiales que lo resuelvan. En este trabajo, primero se resolverá el problema de forma exacta, utilizando la técnica *Backtracking*. Luego, con el fin de mejorar la complejidad, se presentarán varios algoritmos heurísticos que también intenten resolverlo.

En concreto, se implementarán los siguientes algoritmos:

1. Un algoritmo exacto utilizando *Backtracking*.
2. Una heurística constructiva *golosa*.
3. Una heurística constructiva aleatoria.
4. Una heurística de *búsqueda local*.
5. Una metaheurística *GRASP*.

El objetivo de este trabajo es experimentar sobre cada uno de los algoritmos implementados, estudiando su complejidad y tiempo de ejecución en función del tamaño de la entrada, y, en los algoritmos heurísticos, también analizar la calidad de los datos. Se intentarán definir familias de grafos que optimicen los resultados, o que hagan los contrario, es decir, que no proporcionen una solución óptima.

3. Situación real modelada con CMF

3.1. Pinche Trump Cat Problem

Los Estados Unidos de América acaban de invadir México y el emperador Ronald Trump decidió que su nuevo territorio necesita con urgencia una reestructuración política completa.

Una de las primeras medidas a tomar es cambiar la distribución de las provincias del país, y en particular mudar la capital administrativa de México D.F. a un territorio denominado Centro de Reinado de México (CRM). Este territorio puede estar formado por una o más ciudades y debe cumplir con los siguientes requerimientos estrictos:

- Todas las ciudades dentro del territorio deben estar conectadas entre sí por rutas directas (es decir sin pasar por otras ciudades intermedias), para asegurar una fluida comunicación.
- Dichas ciudades tienen que tener la mayor cantidad posible de rutas que comuniquen el CRM con ciudades fuera de este, para favorecer el comercio interno pero principalmente para proveer vías de escape en caso de atentados terroristas realizados contra su Deidad Gobernante.
- No es problema si dentro del territorio delimitado queda alguna ciudad que no cumpla el primer punto, ya que estas pueden formar parte de otras provincias dentro del CRM.

Ronald Trump decidió contratar (en un intento por corregir la imagen pública de xenófobo que le fue injustamente atribuida) a un equipo de latinos del Departamento de Computación de la UBA, en Argentina, para determinar la mejor ubicación del CRM.

Para resolver este problema, el territorio mexicano puede ser representado como un grafo, donde los nodos son las ciudades y existe una arista entre ellos si hay alguna ruta directa que conecte estas ciudades.

Por lo tanto, la nueva capital administrativa será una clique en nuestro grafo ya que es condición necesaria que todas las ciudades dentro del CRM se conecten entre sí, es decir, que exista una arista entre ellas. Además, de todas las cliques posibles del grafo debe de ser una de las cliques (ya que puede existir más de una) que tenga mayor cantidad de aristas que conecten a los nodos de la clique con los que no pertenecen a ésta. Esto se debe a que se busca maximizar las rutas que conectan a la nueva capital con ciudades fuera de ésta para garantizar rutas de escape.

Esto quiere decir que para resolver el problema, lo que buscamos es justamente la clique de máxima frontera o CMF dentro del grafo.

4. Marco Experimental

En este trabajo se desarrollaron heurísticas eficaces para buscar la *clique de máxima frontera* (CMF) de un grafo, la cual se presenta como un problema difícil de resolver, y por lo tanto interesante para nosotros en grafos no regulares, como pueden ser los ciclos simples, grafos completos, etc. Por eso es que para medir la calidad de nuestras soluciones y la eficiencia de nuestros algoritmos optamos por analizar un escenario lo mas diverso posible, por lo tanto usando grafos generados de forma pseudo-aleatoria.

Sin embargo, a la hora de realizar mediciones en términos de complejidad temporal, reducir la varianza de nuestros grafos y por lo tanto aumentar la consistencia de las mediciones era una propiedad deseada, es por esto que para este tipo de mediciones usamos grafos bien definidos.

A continuación se describen los marcos experimentales usados tanto para medir la eficacia, como la complejidad de nuestros algoritmos.

4.1. Complejidad Temporal

Para la complejidad temporal, se midieron los tiempos de cómputo de nuestros algoritmos sobre grafos completos de tamaño n, $\forall n \in [10, 500] \cap \mathbb{N}$. En algunos casos, las mediciones fueron realizadas sobre un conjunto mas acotado, dado el enorme tiempo de cómputo que requerían, como es el caso de *Backtracking* y *GRASP con búsqueda Local*.

Utilizamos grafos completos ya que están bien definidos y en términos de complejidad temporal representan el peor caso de nuestros algoritmos.

Para medición de estos tiempos, se utilizó la librería *chronos* de *C++*. Además, éstas fueron realizadas 5 veces y luego promediadas para reducir errores de medición, como pueden ser las políticas de *scheduling* del SO y demás funcionalidades del sistema que impacten negativamente la veracidad de las mediciones. Dado que la varianza entre las mediciones tomadas resultó despreciable, consideramos que 5 mediciones eran suficientes y se siguió utilizando este parámetro en todos los experimentos.

Todas las mediciones fueron realizadas con optimizaciones *O3* al compilar, lo cual nos permitía obtener el máximo rédito de nuestros algoritmos, y ya que lo aplicamos a todas, no influía en nuestros experimentos.

4.2. Eficacia

Para una instancia y una heurística particulares definimos la eficacia como el cociente entre la solución provista por la heurística y la solución exacta, multiplicado por 100 para representarlo como un porcentaje.

Para comprobar la eficacia de nuestros algoritmos, se utilizaron grafos pseudo-aleatorios[1] generados con una herramienta que toma dos valores como parámetro: *n (> 10)* y *d (entre 0 y 10)*. Ésta crea un grafo de n nodos y toma $n * d/10$ de esos nodos y los conecta entre sí formando cliques disjuntas de tamaño aleatorio. Luego, conecta nodos al azar de forma que cada posible arista tiene una probabilidad de $d/10$ de estar en el grafo. De tal forma que con $d = 0$ se genera un grafo con n nodos aislados, y con $d = 10$ un grafo completo.

De esta forma, se generaron grafos pseudo-aleatorios de tamaño n, $\forall n \in [10, 150] \cap \mathbb{N}$. Sobre ellos se obtuvo la solución exacta utilizando *backtracking*, y luego se comparó con la solución obtenida por nuestras heurísticas, comprobando la eficacia de nuestros algoritmos.

[1] Para todas las decisiones aleatorias se utilizó la función *rand*() de C++.

5. Solución exacta

5.1. Backtracking

Para resolver el problema de forma exacta, se utiliza la técnica algorítmica *Backtracking*, la cual consiste en realizar una búsqueda exhaustiva entre todas las posibles soluciones al problema.

En nuestro caso, esta técnica genera todas las cliques posibles y para cada una de ellas se calcula su frontera. Por cada nueva frontera calculada, se compara con el mejor resultado ya obtenido, y de hallar uno mejor, lo reemplaza.

Esto se realiza de forma recursiva, a través de la función MaximaFrontera, la cual toma una lista de nodos que es potencialmente una clique, y un nodo. Esta función se llama inicialmente con una lista de nodos vacía y con el primer nodo. Para cada llamado, la función analiza la lista recibida. En caso de ser clique, calcula su frontera y la compara con los valores máximos ya obtenidos, actualizando estos últimos en caso de encontrar una mejor solución. Luego, se llama recursivamente avanzando al siguiente nodo tomando las dos opciones posibles: agregarlo o no agregarlo. De esta forma, se van generando todos los conjuntos de nodos posibles y se lo estudia a cada uno.

De todas formas, si se determina que una lista de nodos no es clique, se descarta la rama (a menos que sea una lista vacía), ya que agregar nuevos nodos no puede recuperar la propiedad de ser clique. Esto hace que no se recorran todos los conjuntos posibles de nodos, sino los que son potencialmente cliques.

Esto finaliza cuando ya se recorrieron todos los nodos y por lo tantos se formaron todas las cliques posibles.

El algoritmo mencionado se puede ver en el siguiente pseudocódigo:

```
function MaximaFrontera(potencialClique, nodo)

    seAgregoUnNodo ← ¬vacia(potencialClique) ∧ ultimo(potencialClique) = nodo − 1 ;
    si seAgregoUnNodo ∧ esClique(potencialClique) entonces
        tamFrontera ← TamañoFrontera(potencialClique);
        si tamFrontera >maxFrontera entonces
            ActualizarMaximos(potencialClique, tamFrontera)
        fin
    en otro caso
        si seAgregoUnNodo entonces
            retornar
        fin
    fin
    si recorrí todo los nodos entonces
        retornar
    fin
    maximaFrontera(potencialClique, nodo+1) ;
    clique.Agregar(nodo) ;
    maximaFrontera(potencialClique, nodo+1) ;
```

Algoritmo 1: Algoritmo exacto.

Algo importante de destacar sobre este algoritmo es que en toda llamada la lista de nodos que se recibe es una potencial clique. Esto se debe a que, por como fue construida, todos los nodos de la lista, excepto el último, forman una clique. De no serlo, en el llamado anterior, al determinar que no era clique, no se hubiese continuado y agregado un nodo a esa lista.

Por este motivo, determinar si una lista de nodos (o potencial clique) es una clique consiste únicamente en ver si el último nodo es adyacente a todos los anteriores, lo cual reduce considerablemente la complejidad de la función.

Para evitar controlar innecesariamente si una lista de nodos es clique luego de no haber

agregado nada en la iteración anterior (y por lo tanto es necesariamente clique debido a que en la iteración anterior lo era), el algoritmo verifica en un comienzo si algo fue agregado, y solo en caso afirmativo chequea si se trata de una clique. En caso contrario, saltea el paso asumiendo que ya lo realizó en la iteración anterior. Esto es lo que en el pseudocódigo se muestra como *seAgregoUnNodo*.

Este paso mencionado consiste sencillamente en verificar que la lista recibida no es vacía y que el último nodo de dicha lista es el anterior al nodo actual (nodo-1), ya que la lista va a haber cambiado de la iteración anterior si es que se agregó ese nodo.

5.2. Complejidad

Al tratarse de una función recursiva, la complejidad se divide en dos partes: la cantidad de llamados recursivos y el costo de cada uno de ellos.

En cada llamado recursivo hay 3 casos posibles:

1. En el paso anterior se agregó un nodo y la potencial clique no resulta ser clique y se descarta la rama.
2. En el paso anterior se agregó un nodo y la potencial clique resulta ser clique, y se avanza al siguiente nodo sin agregar el nodo actual en una rama, y agregándolo en otra.
3. En el paso anterior no se agregó ningún nodo y directamente se avanza al siguiente nodo sin agregar el nodo actual en una rama, y agregándolo en otra.

En cuanto a cantidad de llamados, el peor caso se presenta cuando se cae en los casos 2 o 3, ya que éstas llevan a dos nuevos llamados recursivos. Dado que de cada llamado se generan dos más (siempre y cuando no se hayan recorrido todos los nodos), se genera un árbol de llamados recursivos binario, donde, en peor caso, el nivel j del árbol (donde se recorrieron j nodos) contiene 2^j llamados recursivos. Luego, la cantidad de llamados totales es la sumatoria de la cantidad de llamados en cada nivel, o sea, $\sum_{j=0}^{n} 2^j$, que equivale a $2^{n+1} - 1$ llamados recursivos en el peor caso, o, en términos de complejidad, $O(2^{n+1})$.

Notar que este peor caso puede suceder, ya que en un grafo completo (y por lo tanto clique), todo subconjunto de nodos es también clique y por lo tanto siempre se cae en los casos 2 y 3.

En cuanto al costo de cada llamado recursivo, el peor caso claramente se da cuando en el paso anterior se agregó un nodo y la lista de nodos resulta ser clique (caso 2), ya que, al cumplirse esta condición, luego se debe calcular el tamaño de la frontera, y posteriormente realizar dos llamados recursivos. De todas formas, esto puede ocurrir a lo sumo la mitad de las veces, ya que, si se realizan los dos llamados recursivos, uno de ellos se hará pasando la misma lista que fue recibida y por lo tanto ese llamado caerá en el caso 3, es decir, la mitad de los llamados caerán en los casos 1 y 2 y la otra mitad en el caso 3. Luego, para estudiar el peor caso es necesario determinar el costo de un llamado que cae en el caso 2 y el de uno que cae en el caso 3.

De forma más detallada, el peor caso en un llamado recursivo (caso 2) se presenta cuando se cumplen dos cosas:

- La potencial clique recibida es efectivamente clique (y se ha agregado un nodo en la iteración anterior, ya que en caso contrario, no hace falta estudiar ese caso). Esto se debe a que no solo hay que determinar si es clique, sino que, además, luego hay que calcular el tamaño de la frontera para poder compararlo con el máximo ya obtenido.
- Aún quedan nodos por recorrer. Al quedar nodos por recorrer, se realizan dos llamados recursivos, pasando la clique obtenida en el primer llamado, y pasando en el segundo una nueva potencial clique al agregarle a la clique anterior el nodo actual. Este último paso agrega el costo de copiar ambas listas.

Por lo tanto, el costo en peor caso en un llamado es la suma de las siguientes cosas:

1. Determinar si una lista es clique.
2. Calcular su frontera.
3. Copiar las listas para los llamados recursivos.

Como se mencionó previamente, como la lista de nodos recibida es una potencial clique, para determinar si es efectivamente una clique solo hay que verificar que el último nodo sea adyacente a los anteriores. Esto tiene un costo $O(k)$, siendo k el tamaño de la lista, o simplemente $O(n)$, ya que con matriz de adyacencias determinar si dos nodos son adyacentes es $O(1)$.

En cuanto a calcular el tamaño de la frontera, hay que sumar la cantidad de adyacencias de cada nodo de la clique, es decir, para cada nodo de la clique ($O(k)$) se compara contra todo nodo ($O(n)$) para ver si es adyacente ($O(1)$) y de ser así se agrega al contador. Finalmente, se resta la cantidad de adyacencias que son internas a la clique ($k * (k-1)$, cada nodo de la clique se conecta con todos menos sí mismo). Por lo tanto, el costo de este paso es $O(k * n * 1)$, lo cual se puede acotar por $O(n^2)$.

Una vez calculado el tamaño de la frontera, compararlo con el tamaño máximo ya obtenido y actualizarlo en caso de ser mayor no aporta un costo extra ya que se trata de una comparación y una asignación de números enteros y dicha información se guarda en una variable global por lo que puede ser accedida directamente. Actualizar la clique de frontera máxima, en cambio, requiere copiar la lista y eso tiene un costo lineal en la cantidad de elementos, por lo tanto $O(k) = O(n)$. De todas formas, la complejidad asintótica de esta parte se mantiene en $O(n^2)$.

Por último, se deben copiar dos listas para los llamados recursivos. Dado que una lista tiene k y la segunda $k+1$ elementos (ya que se agrega un nodo), el costo de copiar ambas es $O(k+(k+1)) = O(k)$ o simplemente $O(n)$.

Finalmente, al sumar las tres partes mencionadas, el costo de cada llamado recursivo que cae en el caso 2 es $O(n) + O(n^2) + O(n) = O(n^2)$.

Un llamado recursivo que cae en el caso 3, en cambio, solo tiene el costo de copiar las listas, ya que los primeros dos pasos se evitan. Luego, el costo de un llamado de caso 3 es $O(n)$.

Habiendo calculado tanto la cantidad de llamados recursivos como el costo de cada uno de ellos, solo resta multiplicar ambos costos. Recordando que de los $O(2^{n+1})$ llamados que podrían realizarse en el algoritmo, en peor caso mitad caen en el caso 2 y la otra mitad en el caso 3, esto se separa en dos. Los $O(2^n)$ llamados (mitad de los llamados) que caen en el caso 2 tienen complejidad $O(n^2)$ y los $O(2^n)$ llamados restantes que caen en el caso 3 tienen complejidad $O(n)$.

Así, el costo total en peor caso del algoritmo es $O(2^n)*O(n^2) + O(2^n)*O(n) = O(2^n)*O(n^2) + O(2^n)*O(n^2) = 2*O(2^n)*O(n^2) = O(2^{n+1})*O(n^2) = O(2^{n+1}.n^2)$.

5.3. Experimentación

La experimentación sobre el algoritmo exacto consiste en medir los tiempos[2] de cómputo para grafos de distintos tamaños. Para cada grafo, en principio se midieron los tiempos 5 veces y luego se promediaron los resultados para minimizar la aleatoriedad que se da cuando se miden tiempos en computadoras que pueden estar corriendo otros procesos. Dado que la varianza entre las distintas mediciones resultó despreciable, consideramos que 5 mediciones eran suficientes y se siguió utilizando este parámetro para los siguientes experimentos.

El objetivo de este experimento es comprobar de forma empírica cómo se comporta el algoritmo en función del tamaño de entrada. Además, se espera que otorgue información útil para los experimentos subsiguientes cuando queramos comparar los resultados obtenidos con otros métodos con el resultado exacto, y así saber con grafos de que tamaño experimentar en función del tiempo que estemos dispuestos a dedicar.

En un principio calculamos los tiempos de cómputo con grafos generados aleatoriamente con n entre 10 y 150, con la herramienta cuyo funcionamiento esta detallado en la sección Marco

[2]Para las mediciones de tiempo se utilizó la librería chronos de C++.

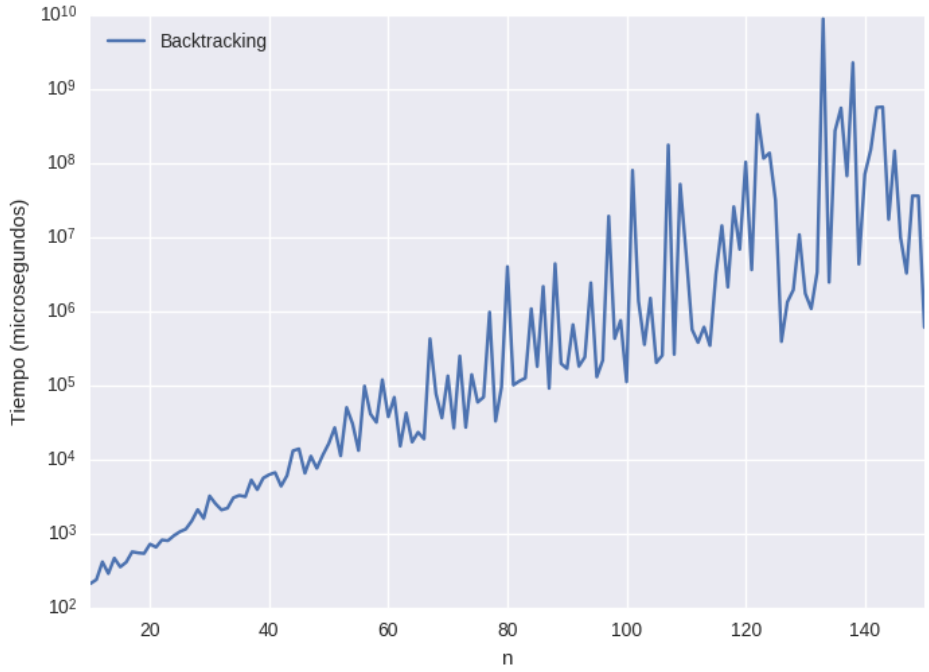


Figura 1: Tiempos de cómputo en microsegundos del algoritmo exacto sobre grafos aleatorios según cantidad de nodos. El eje y está en escala logarítmica.

Experimental, y en la figura 1 se puede observar el resultado de las mediciones. Tal como era esperable, los tiempos aumentan de forma exponencial según el tamaño de entrada. Sin embargo, presenta variaciones significativas dadas por alguna otra variable.

Observando algunos de los grafos que llevaron mayor tiempo resolver, notamos que tenían mayor cantidad de aristas (m) que sus vecinos en n, por lo que analizamos la posibilidad de que la cantidad de aristas afecte de forma más directa a la complejidad del algoritmo que la cantidad de nodos. Esto es posible ya que por como fueron generados los grafos, la relación entre cantidad de nodos y aristas es solo directa en probabilidad, y presenta variaciones aleatorias. En la figura 2 se grafican los mismos datos, pero ordenados según cantidad de aristas, y se puede observar que presentan aún más variación que la figura anterior.

Por lo tanto las variables que afectan a la complejidad del algoritmo, además de n y m, deben deberse a alguna otra propiedad, como puede ser por ejemplo la cantidad total de cliques, que en el caso de los grafos aleatorios es más difícil de caracterizar.

Sin embargo si realizamos las mediciones con grafos completos, la cantidad de cliques es igual al cardinal del conjunto de partes de n y dejaría de haber ambigüedad. Además este debería ser el peor caso de backtracking, por lo que podemos comparar la función tiempo de ejecución ($T(n)$) con la función $g(n) = 2^n * n^2 * c$, que tiene la misma complejidad que la calculada de forma teórica para el algoritmo. En la figura 3 se observa que efectivamente la medición de tiempos se encuentra acotada por la función $g(n)$.

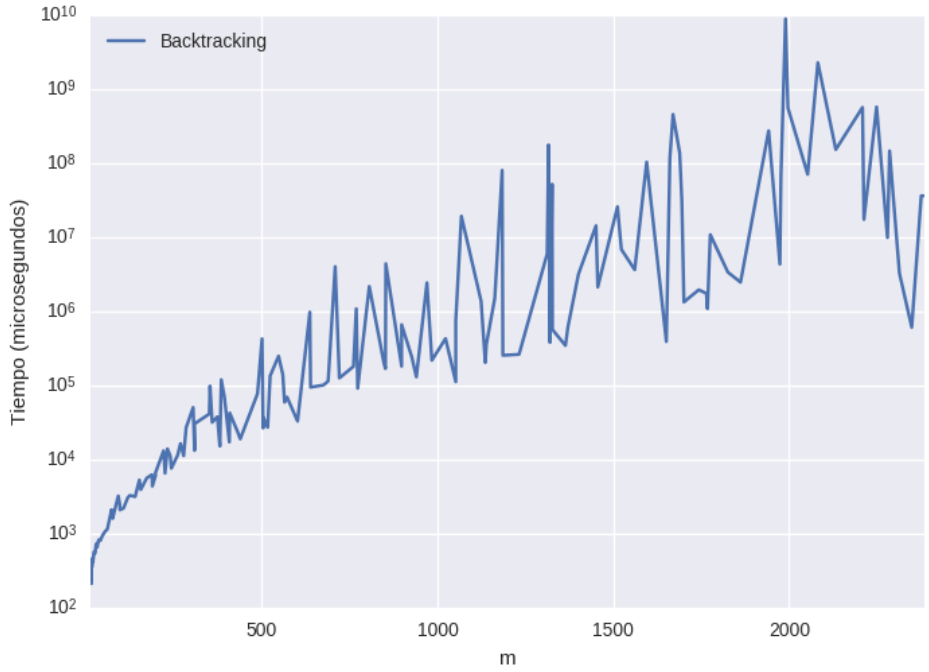


Figura 2: Tiempos de cómputo en microsegundos del algoritmo exacto sobre grafos aleatorios según cantidad de aristas. El eje y está en escala logarítmica.

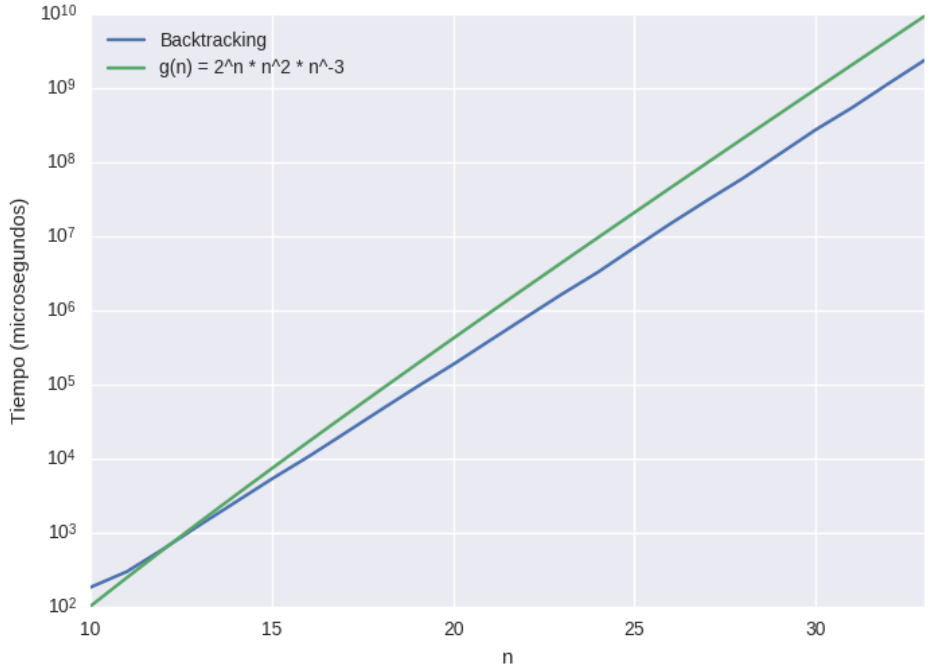


Figura 3: Tiempos de cómputo en microsegundos de Backtracking sobre grafos completos. El eje y está en escala logarítmica.

6. Heurísticas constructivas

6.1. GLF *(Greedy Largest First)*

Inspirada en la heurística LFS (*Largest First Sequence*) esta heurística golosa, o *greedy* funciona de forma iterativa y se basa en la intuición de que un nodo de mayor grado tiene más probabilidades de pertenecer a una clique de frontera máxima. Esto se debe a que como tiene muchas adyacencias, muchas de estas contribuirán al tamaño de la frontera, o en caso de que la mayoría de sus vecinos sean parte de la clique también, será mayor la cantidad de vecinos que aumentan la frontera.

6.1.1. Algoritmo

Para implementar esta idea, el primer paso es armar una lista de todos los nodos del grafo, ordenados de mayor a menor según su grado. La solución al problema es una clique representada como un conjunto de nodos.

Comenzando con una clique vacía y por lo tanto de frontera nula como solución parcial, en cada paso iterativo se decide si el próximo nodo es agregado a la solución. Para ser agregado debe cumplir:

- La solución anterior más el nuevo nodo forman una clique.
- La clique nueva tiene frontera mayor que la solución anterior.

Esta función recorre todos los nodos que pueden ser agregados y devuelve los nodos que forman la clique encontrada y el tamaño de su frontera. El funcionamiento del algoritmo se puede entender mejor viendo el siguiente pseudocódigo:

```
function maxFronCliqueGLF
    nodos ← Vector con los nodos y sus grados ;
    Ordeno nodos de mayor a menor según su grado;
    clique ← Vector vacío;
    maxFront ← 0;
    tamClique ← 0;
    para cada i desde 1 hasta n, y mientras nodos_i pueda mejorar la solución hacer
        clique.Agregar(nodos_i) ;
        si clique es clique y tiene mayor frontera que antes entonces
            Actualizar(maxFront) ;
            tamClique ← tamClique + 1
        en otro caso
            clique.Remover(nodos_i)
        fin
    fin
    devolver clique y maxFront
```

Algoritmo 2: Algoritmo goloso GLF.

En este pseudocódigo hay tres operaciones que no son triviales de resolver. Una es verificar que un conjunto de nodos forman efectivamente una clique. Sabiendo que antes de agregar el último nodo ya era una clique, esto se realiza verificando que el último nodo esta conectado con todos los anteriores.

Para entender las otras dos operaciones, que son verificar que $nodos_i$ pueda mejorar la solución y calcular la frontera de $clique \cup nodos_i$, hace falta demostrar un pequeño lema primero.

Lema 1: Sea C_k una clique de G de k nodos, $f(C_k)$ el tamaño de su frontera, y v' un nodo de G tal que $v' \notin C_k$. Si $C_{k+1} = C_k + v'$ es una clique, $f(C_{k+1}) = f(C_k) + d(v') - 2 * k$.

Demostración: De forma constructiva podemos ver que al agregar un nodo v' a la clique C_k, todas las aristas que unían a C_k con v' dejan de estar en la frontera. Por otro lado, ahora pertenecen a la frontera todas las aristas incidentes a v' que no se conectan con nodos de C_k. Como sabemos que C_{k+1} es una clique, tiene que haber exactamente una arista entre cada nodo de C_k y v', por lo que $f(C_{k+1}) = (f(C_k) - k) + (d(v') - k) = f(C_k) + d(v') - 2 * k$.

De esta forma es trivial calcular el tamaño de la frontera y, además, se deduce inmediatamente que para que el nodo a añadir agrande la frontera de la clique, debe tener grado mayor a $2 * k$. Esto nos ahorra verificar con todo nodo que no cumpla esta condición.

6.1.2. Complejidad

En este algoritmo, la complejidad se divide en tres partes: La complejidad del *set up*, la complejidad del ciclo y retornar los resultados.

En la primera parte, inicialmente se genera un vector de pares, donde se coloca cada nodo con su grado. Dado que construir un par tiene costo $O(1)$, el costo de agregar un par nuevo es el costo de calcular el grado del nodo. Esto se realiza recorriendo la columna de la matriz de adyacencias correspondiente al nodo, contando la cantidad de adyacencias. Al tratarse de una matriz cuadrada de $n \times n$, el costo de determinar el grado es $O(n)$. Así, como se agrega un par por cada nodo, el costo final de esta operación es $O(n^2)$.

Una vez obtenido el vector de pares, se lo ordena descendentemente. Debido a que la comparación entre dos pares se realiza en $O(1)$ (ya que consta en comparar solo la primera componente del par), ordenar los n pares tiene costo $O(n * log(n))$.

En cuanto a la primera parte, solo resta hacer pequeñas asignaciones las cuales no implican un costo adicional. Por lo tanto, el costo total de la primera parte es $O(n^2) + O(n * log(n)) = O(n^2)$.

La segunda parte es el cuerpo del algoritmo, el cual está compuesto por un ciclo que, en peor caso, recorre todos los nodos. Determinar si un nodo es candidato para mejorar la solución consiste simplemente en verificar que el grado del nodo es mayor a dos veces tamaño de la clique. Como el grado del nodo ya fue calculado en la primera parte, esta validación tiene costo constante. Así, el ciclo realiza $O(n)$ iteraciones donde el paso de una a otra no implica costo significativo.

En cada iteración primero se agrega un nuevo nodo, lo cual tiene costo $O(1)$, y luego se verifican dos cosas:

- Que la nueva tira de nodos es clique luego de agregar al último nodo. Como solo es necesario verificar que el último nodo sea adyacente a los anterior (debido a que ya era clique antes de agregar al último nodo) el costo de esta operación es $O(n)$[3].
- Si se cumple que es clique, se verifica si el tamaño de la frontera es mayor. Esto tiene costo constante utilizando la propiedad mencionada previamente.

Una vez realizadas estas verificaciones, el costo restante es constante, independientemente de si se cumple lo anterior o no. De cumplir, solo se guardan los valores nuevos, y si no se cumple, se remueve el nodo agregado, el cual se encuentra siempre en la última posición.

Finalmente, el costo de la segunda parte del algoritmo está dominado por determinar si una tira de nodos es clique en cada iteración, y por lo tanto el costo es $O(n)$ * $O(n) = O(n^2)$.

La tercera y última parte es retornar los resultados. Esto no tiene costo significativo ya que copiar el tamaño máximo de la frontera es constante, y la clique se devuelve por referencia.

Habiendo calculado las tres partes, el costo total en peor caso el la suma de las complejidades de las tres partes. Esto es $O(n^2) + O(n^2) + O(1) = O(n^2)$.

[3]Ver costo de operación *esClique* en algoritmo de Backtracking.

6.2. RI *(Random Insertion)*

Esta heurística imita la heurística anterior, pero sin considerar ningún orden particular para los nodos. Es claro que, si bien el nodo de mayor grado parece una buena forma de comenzar, éste no siempre formará parte de la solución. Por este motivo, un orden aleatorio resulta una alternativa tentadora a la hora de experimentar.

6.2.1. Algoritmo

El algoritmo de esta heurística presenta muy pocas modificaciones con respecto al anterior. La principal es, sin duda, que ya no se ordenan los nodos en base a su grado, sino que se genera una lista con todos los nodos y luego se los mezcla aleatoriamente[4]. Luego, en el cuerpo del algoritmo se busca agregar nodos que mejoren la frontera en el orden provisto por la tira mezclada.

```
function maxFronCliqueRI
    nodos ← Vector con los nodos;
    Mezclar(nodos);
    clique ← Vector vacío;
    maxFront ← 0;
    tamClique ← 0;
    para cada i desde 1 hasta n hacer
        clique.Agregar(nodos_i);
        si clique es clique y tiene mayor frontera que antes entonces
            Actualizar(maxFront) ;
            tamClique ← tamClique + 1;
        en otro caso
            clique.Remover(nodos_i) ;
        fin
    fin
    devolver clique y maxFront
```

Algoritmo 3: Algoritmo RI.

6.2.2. Complejidad

Al igual que el algoritmo de GLF, la complejidad se divide en tres partes: La complejidad del *set up*, la complejidad del ciclo y retornar los resultados.

La tercera y última parte se mantiene igual al otro algoritmo, por lo tanto la complejidad es $O(1)$.

En cuanto a las dos primeras partes, ambas sufren algún cambio con respecto a GLF. En la primera parte ya no se calculan todos los grados y luego se ordenan, sino que simplemente se ponen todos los nodos en un vector ($O(n)$) y luego se los mezcla aleatoriamente. Esta última operación tiene orden lineal en la cantidad de elementos del vector, por lo tanto, la complejidad final de la primera parte es $O(n) + O(n) = O(n)$.

En cuanto a la segunda parte, los cambios que se realizan son muy sutiles. El primero es que, al no tener los grados de los nodos calculados, ya no se puede evitar considerar unos de los nodos por tener grado muy bajo. Esto hace que se recorran todos los nodos hasta el final de la lista. De todas formas, en peor caso, esto no altera la complejidad. Se siguen realizando $O(n)$ iteraciones. El segundo cambio que se realiza es a la hora de calcular la nueva frontera de la clique. Como antes se tenían calculados los grados de los nodos, este cálculo resultaba tener costo constante. En este algoritmo, en cambio, hay que calcularlo con costo $O(n)$. De todas formas, como previamente se debe calcular si la tira de nodos es efectivamente una clique en $O(n)$, el costo final de esta parte no se ve afectado. Al igual que en GLF, lo restante es de orden constante.

[4]Se usa funcion random_shuffle de C++ con costo $O(n)$

Así, la complejidad del cuerpo del algoritmos $O(n)$ * $O(n)$ = $O(n^2)$, y sumando a esto el costo de la primera parte o *set up*, la complejidad final del algoritmo es $O(n^2) + O(n) = O(n^2)$, al igual que el algoritmo de GLF.

6.3. Experimentación

Si bien GLF y RI son esencialmente el mismo algoritmo, con diferente orden para recorrer los nodos como única diferencia a simple vista, vamos a compararlos tanto en tiempo de ejecución como en eficacia de soluciones para desenmascarar las diferencias más sutiles.

6.3.1. Complejidad temporal

Para comparar tiempos, se utilizaron grafos completos de tamaño 10 hasta 500, de modo que se pueda apreciar el comportamiento de ambos algoritmos a medida que la cantidad de nodos aumenta. Es difícil realizar una predicción sobre los resultados ya que ambos algoritmos tienen complejidad $O(n^2)$, obteniéndose muy similarmente en ambos.

Los resultados se muestran en el siguiente gráfico:

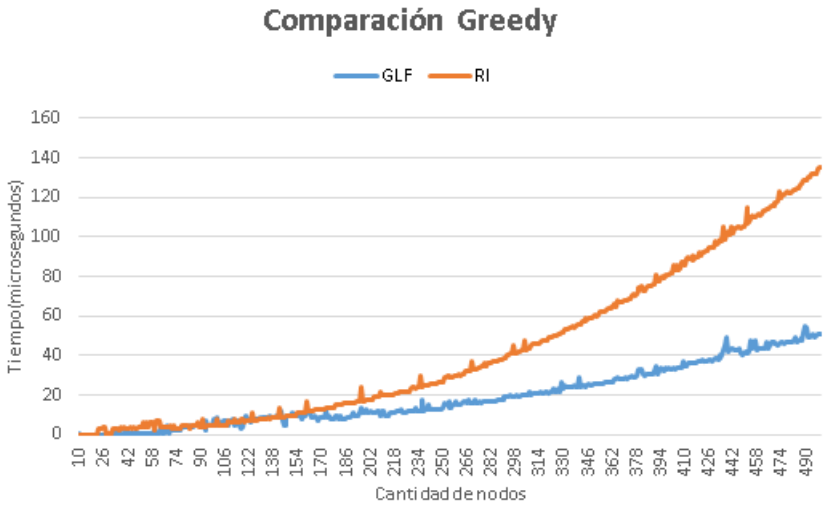


Figura 4: Comparación de tiempos de cómputo en microsegundos entre GLF y RI sobre grafos completos.

Se puede ver a simple vista que GLF es más veloz que RI desde el primer instante y, a medida que aumenta el tamaño del grafo, la brecha entre ambos se amplía. Si bien no se aprecia en el análisis de complejidad de peor caso, GLF tiene una condición de corte que lo hace evitar agregar nodos que de antemano se sabe que no agrandarán la frontera, y, como los nodos están ordenados, no hace falta mirar ninguno después de eso. Si bien en este caso se midieron los tiempos únicamente con grafos completos, esta condición de corte es efectiva en todo grafo, ya que, recordando que el nodo a considerar debe tener grado mayor a dos veces el tamaño de la clique, es claro que a medida que el tamaño de la clique aumenta utilizando los nodos de mayor grado, rápidamente los nodos restantes (los de menor grado) no cumplirán esta restricción. Esta propiedad hace que GLF sea considerablemente más veloz que RI, ya que este último, al tener un orden aleatorio de los nodos, debe considerar hasta el último nodo.

Para poner a prueba esta hipótesis, contamos la cantidad de iteraciones que realizan GLF y RI para los mismos grafos completos en los cuales se hizo la medición de tiempos obteniendo los siguientes resultados:

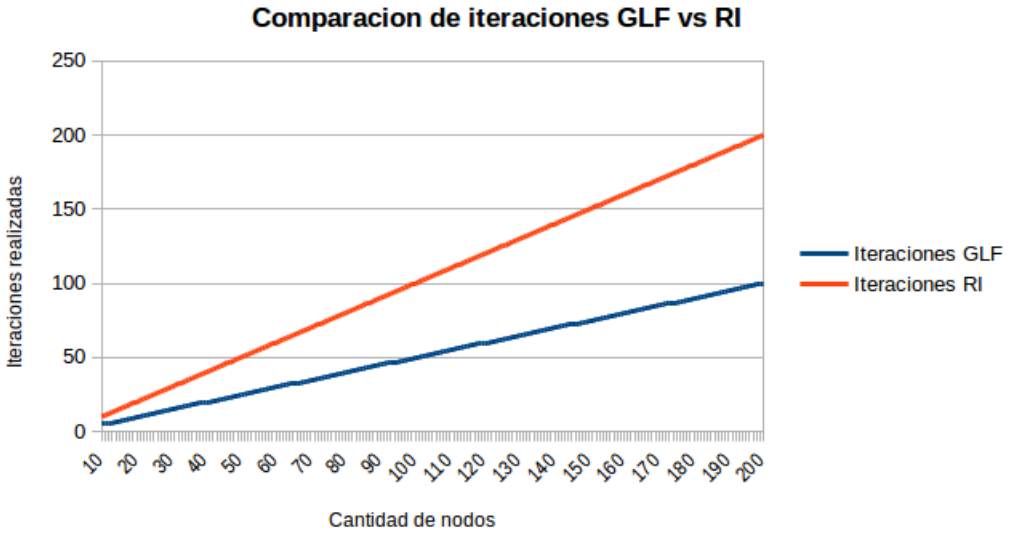


Figura 5: Comparación de iteraciones entre GLF y RI sobre grafos completos.

Se puede apreciar que para cada K_n RI siempre realiza n iteraciones mientras que GLF está haciendo la mitad de las iteraciones. Que GLF realice exactamente la mitad se debe a que estamos trabajando con grafos completos donde como todos los nodos forman cliques entre sí al haber agregado la mitad de los nodos ya no se puede mejorar la frontera porque quedan todos nodos de grado $n - 1$. Sin embargo para grafos en general puede hacer más o menos iteraciones pero siempre a lo sumo iterará n veces, por lo que la cantidad de iteraciones de GLF está acotada por las iteraciones que realiza RI. Dado que dentro del ciclo ambos algoritmos tienen la misma complejidad y como GLF realizará a lo sumo las iteraciones que hace RI, GLF presentará siempre tiempos de cómputo menores o iguales a los de RI.

6.3.2. Eficacia

Si bien un algoritmo es más veloz que el otro, esto no determina que el algoritmo sea mejor. Es importante comparar la calidad de las soluciones obtenidas por cada uno para así poder llegar a una comparación completa entre ambos.

Para comparar la eficacia, se generaron grafos de entre 10 y 150 nodos de forma aleatoria que fueron corridos inicialmente con el algoritmo exacto. De esta forma, a la hora de calcular la efectividad de una heurística sobre un grafo, lo realizamos mediante el cociente entre la solución obtenida y la solución exacta. Si bien *a priori* uno puede pensar intuitivamente que las soluciones de GLF serán mejores por la forma en que se eligieron los nodos, es difícil determinarlo previo a la experimentación, ya que los algoritmos con factores aleatorios han demostrado tener gran poder en las heurísticas.

La eficacia obtenida para los algoritmos GLF y RI se muestra a continuación:

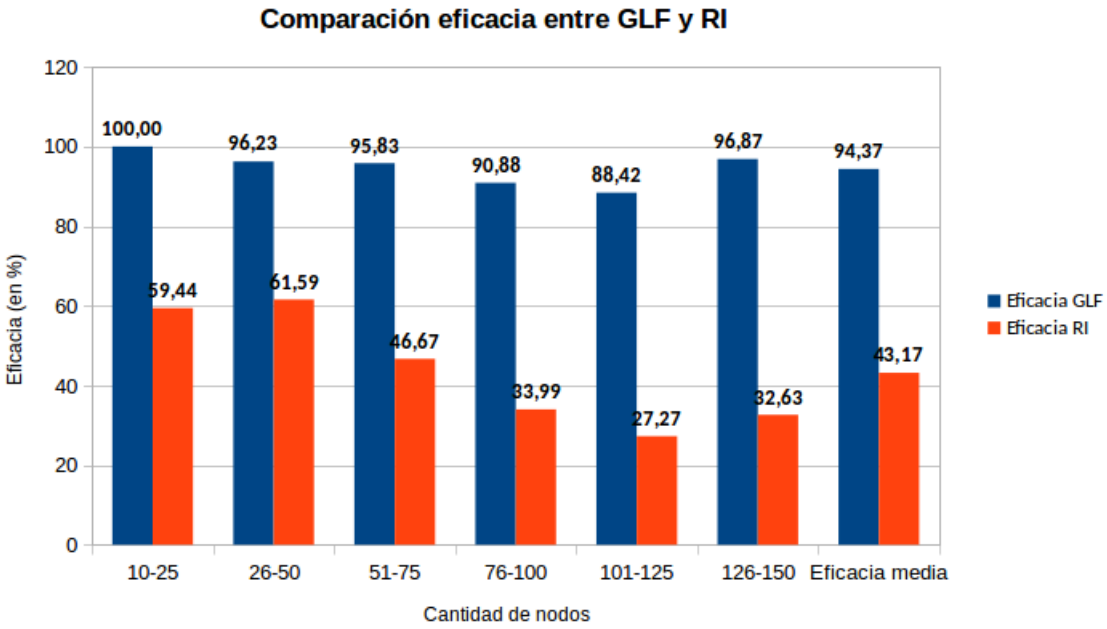


Figura 6: Comparación de eficacia entre GLF y RI sobre grafos completos.

Como se puede apreciar, la diferencia entre ambos algoritmos es muy grande. GLF presenta una eficacia promedio de casi 95 %, mientras que RI no llega ni a la mitad de eso. Esto se debe justamente a cómo beneficia incluir a los nodos de mayor grado dentro de la clique, pues o bien aumentan la frontera debido a su gran cantidad de vecinos fuera de la clique, o poseen muchos vecinos dentro de la clique que entre todos generan una mayor frontera. Como este es un problema que depende justamente de los grados de los nodos, resulta improbable que el caso aleatorio devuelva una buena solución.

Más allá de la comparación entre ambos, algo que se ve para ambos algoritmos es que a medida que el tamaño de los grafos aumenta, la calidad de las soluciones disminuye levemente. Esto puede deberse a que a medida que crece el grafo, es mayor la cantidad de cliques que este puede contener. Como ambos algoritmos no calculan la frontera de todas las cliques del grafo (ya que si lo hicieran tendrían complejidad exponencial), es posible que la proporción de cliques analizadas sea menor con grafos de mayor tamaño.

En resumen, el algoritmo GLF supera al algoritmo RI tanto en eficacia como en complejidad. Esto lo hace un algoritmo mucho más confiable, y debido a la buena calidad de las soluciones sería un buen algoritmo para implementar en un caso real. Por el otro lado, RI no presenta ninguna ventaja sobre GLF, por lo cual, fuera del ámbito experimental, no es recomendable como herramienta de aproximación.

7. Heurísticas de búsqueda local

7.1. CNLS *(Close Neighbourhood Local Search)*

En la búsqueda local intentaremos mejorar una solución inicial moviéndonos en una vecindad chica de esta solución, es decir, analizaremos soluciones que distan a lo sumo un nodo de la solución original buscando encontrar una clique de máxima frontera local.

Para ello, se le aplicarán 3 operaciones a una solución, las cuales consisten en:

1. Agregar un nodo vecino a la clique a la solución
2. Eliminar un nodo de la clique solución
3. Intercambiar un nodo de la solución por un nodo vecino a la clique

Para cada una de estas operaciones, siempre se elegirá modificar la solución de forma que se maximice la CMF obtenida de entre las posibles modificaciones a realizar.

A continuación se detallarán las operaciones mencionadas anteriormente para una mejor comprensión.

Mejorar la solución agregando un nodo consiste en ver para cada nodo vecino de la clique si al agregarlo a la solución, ésta sigue siendo una clique. En caso de ser una clique se calculará su frontera y se verá si mejora la frontera con respecto a la solución inicial y con los nodos vecinos que se verificó anteriormente. Luego se quitará el nodo agregado para seguir probando con el resto de los vecinos. Finalmente se agregará el nodo vecino que más mejore la frontera de la clique, de no existir ninguno que la mejore no se agregará nada.

```
function MEJORARAGREGANDO(cliqueMF, maxFront)
    vecinos ← dameVecinosClique(cliqueMF);
    para i entre 0 y tamaño de vecinos hacer
        Agregar vecinos_i a la cliqueMF;
        si Sigue siendo una clique y tiene mayor frontera que maxFront entonces
            Actualizar(maxFront);
            mejorNodo ← vecinos_i
        fin
        Eliminar vecinos_i de la cliqueMF
    fin
    si agregar algun nodo mejora la frontera entonces
        Agregar mejorNodo a la clique
    fin
    devolver maxFront
```

Algoritmo 4: Mejora la solución agregando un nodo vecino

Mejorar la solución eliminando un nodo toma un nodo de la clique y lo elimina de la solución, calcula la frontera de esa nueva clique y ve si mejora con respecto a la mejor solución obtenida hasta el momento. Luego vuelve a agregar el nodo eliminado para verificar el resto. Finalmente se elimina el nodo que más mejora la frontera al eliminarlo, si no existe ninguno que mejore no se eliminará ningún nodo.

```
function MEJORARELIMINANDO(cliqueMF, maxFront)
    para i entre 0 y tamaño de cliqueMF hacer
        nodoEliminado ← cliqueMF[0];
        Eliminar primer elemento de cliqueMF;
        si tiene mayor frontera que maxFront entonces
            Actualizar(maxFront);
            mejorNodo ← nodoEliminado
        fin
        Agregar atras nodoEliminado a la cliqueMF
    fin
    si Si eliminar algun nodo mejora la frontera entonces
        Eliminar nodoEliminado de la clique
    fin
    devolver maxFront
```

Algoritmo 5: Mejora la solución eliminando un nodo de la clique

Mejorar la solución intercambiando un nodo por otro puede considerarse una combinación de las operaciones anteriores dado que en primer lugar se elimina un nodo de la clique y luego se busca cuál es el nodo vecino a la clique que más mejora la frontera al agregarlo. Luego se vuelve a agregar el nodo eliminado y se repite el mismo procedimiento eliminando a cada nodo de la clique. Finalmente se aplicará la operación de eliminar un nodo de la clique y agregar un nodo vecino que más mejore la frontera, si no hay operación que mejore no se hará nada.

```
function MEJORARINTERCAMBIANDO(cliqueMF, maxFront)
    para i entre 0 y tamaño de cliqueMF hacer
        nodoEliminado ← cliqueMF[0];
        Eliminar primer elemento de cliqueMF;
        vecinos ← dameVecinosClique(cliqueMF);
        para j entre 0 y tamaño de vecinos hacer
            agregar vecinos_i a la clique;
            si es clique y tiene mayor frontera que maxFront entonces
                Actualizar(maxFront);
                mejorNodo ← vecinos_i
            fin
            Eliminar vecinos_i de la clique
        fin
        Agregar atrás nodoEliminado a la cliqueMF
    fin
    si intercambiar algun nodo mejora la frontera entonces
        Eliminar nodoEliminado de la clique;
        Agregar mejorNodo a la clique
    fin
    devolver maxFront
```

Algoritmo 6: Mejora la solución intercambiando un nodo de la clique por un vecino

Ahora nos queda describir cómo usa la búsqueda local estas operaciones, el comportamiento es bastante sencillo, primero se obtiene una solución inicial mediante alguno de los algoritmos descriptos en la sección de heurísticas constructivas. Luego, se ingresará a un ciclo que continuará mientras se siga mejorando la solución. En este ciclo se aplicarán las 3 operaciones de agregar, eliminar e intercambiar nodos a la solución obtenida hasta el momento pero solo se mantendrá la operación que mejoró más la solución, si ninguna operación pudo mejorar la solución se considera que se llegó a una solución óptima local por lo que se saldrá del ciclo finalizando el algoritmo.

Se puede ver de forma más clara el funcionamiento de búsqueda local en el siguiente pseudocódigo:

```
function BusqLocal(cliqueMF)
    res ← GREEDY(cliqueMF);
    CMFAgregar ← Vacio;
    CMFEliminar ← Vacio;
    CMFIntercambiar ← Vacio;
    mientras Sigue mejorando la solución hacer
        CMFAgregar ← cliqueMF;
        CMFEliminar ← cliqueMF;
        CMFIntercambiar ← cliqueMF;
        resAgregar ← mejorarAgregando(CMFAgregar, res);
        resEliminar ← mejorarEliminando(CMFEliminar, res);
        resIntercambiar ← mejorarIntercambiando(CMFIntercambiar, res);
        Actualizar res y cliqueMF con la operación que más mejore la solución;
        Si ninguna operación mejora, se sale del ciclo
    fin
```

Algoritmo 7: Búsqueda local

7.2. Complejidad

Con el objetivo de facilitar el cálculo y la comprensión de la complejidad del algoritmo, se calculará en primer lugar la complejidad de las funciones auxiliares que se encargan de intentar mejorar una solución en cada iteración. En todos se parte de un grafo que tiene n nodos.

7.2.1. mejorarAgregando

Inicia con la función $dameVecinosClique$ con costo $O(n^3)$. Luego se inicia un ciclo que realizará n iteraciones en el peor caso donde en cada iteración se ve si agregar un nodo a la clique hace que ésta siga siendo una clique con costo $O(n)$ y se calcula su frontera con costo $O(n^2)$. Por lo tanto la complejidad de este algoritmo es $O(n^3) + O(n) * (O(n) + O(n^2)) = O(n^3)$.

7.2.2. mejorarEliminando

Comienza con un ciclo que realizará n iteraciones en el peor caso, dentro del ciclo se eliminará el primer elemento de un vector con costo $O(n)$ y se calculará su frontera con costo $O(n^2)$. Una vez fuera del ciclo se buscará el nodo a eliminar en el vector con costo $O(n)$ y luego se lo eliminará también con costo $O(n)$. Entonces, la complejidad del algoritmo será $O(n) * (O(n^2) + O(n)) + O(n) * O(n) = O(n^3)$

7.2.3. mejorarIntercambiando

Inicia con 2 ciclos anidados con una particularidad: el peor ciclo de uno de los ciclos garantiza que el otro ciclo está en su mejor caso.

En el caso de que la clique sea de tamaño n, es decir, que contiene a todos los nodos del grafo, el ciclo externo realiza n iteraciones. Dentro de este ciclo se eliminará el primer elemento de la clique con costo $O(n)$ y se calcularán sus vecinos con costo $O(n^3)$. Ahora, como la clique tiene tamaño n no puede tener vecinos que no pertenezcan a la clique por lo que el vector de vecinos estará vació. Debido a esto el ciclo interno no realizará ninguna iteración y luego se agregará el nodo eliminado nuevamente con costo $O(1)$. La complejidad de esto será entonces $O(n) * (O(n) + O(n^3)) = O(n^4)$.

En el caso contrario de que la clique sea lo más chica posible, es decir, de tamaño 1, el ciclo externo realizará 1 iteración y al eliminar un nodo con costo $O(1)$ (dado que hay un solo elemento), quedará una clique vacía por lo que no tendrá vecinos. Esto hará que el ciclo interno

no realice ninguna iteración. Como todas las operaciones tienen costo $O(1)$, la complejidad para este caso será $O(1)$.

Si la clique es de tamaño 2, el ciclo externo realizará 2 iteraciones. Eliminará el primer nodo con costo $O(1)$ y calculará sus vecinos con costo $O(n)$ ya que tiene que ver los vecinos de 2 nodos, la cantidad de vecinos puede ser a lo sumo $n-1$ por lo que el ciclo interno puede llegar a realizar $n-1$ iteraciones. Dentro del ciclo interno se agrega un nodo con costo $O(1)$, se verifica que siga siendo clique al agregarlo con costo $O(n)$ y se calcula su frontera con costo $O(n^2)$. Luego se elimina el último elemento agregado con costo $O(1)$. La complejidad de esto será entonces $O(2)$ * ($O(n^3)$ + $O(n)$ * ($O(n)$ + $O(n^2)$)) = $O(n^3)$.

Sin embargo, queda un caso a analizar y es el que se presenta cuando los 2 ciclos realizan la *mitad* de las iteraciones. Esto sucede cuando la clique tiene tamaño $\frac{n}{2}$ por lo que el ciclo externo realizará $\frac{n}{2}$ iteraciones. Dentro del ciclo, eliminará el primer nodo con costo $O(n)$ y calculará sus vecinos con costo $O(n^3)$. La clique puede tener a lo sumo $\frac{n}{2}$ vecinos por lo que el ciclo interno puede llegar a realizar $\frac{n}{2}$ iteraciones. En el ciclo interno se agrega un nodo con costo $O(1)$, se verifica si sigue siendo clique al agregarlo con costo $O(n)$ y se calcula la frontera con costo $O(n^2)$. Luego se elimina el último elemento agregado con costo $O(1)$. La complejidad en este caso será de $O(n)$ * ($O(n)$ + $O(n^3)$ + $O(n)$ * ($O(1)$ + $O(n)$ + $O(n^2)$)) = $O(n^4)$.

Finalmente luego de estos ciclos anidados en el caso de que exista algún intercambio que mejore la solución se busca el nodo que hay que eliminar y se lo elimina con costo $O(n^2)$ y luego se agrega otro nodo con costo $O(1)$.

La complejidad de *mejorarIntercambiando* será entonces máx($O(n^4)$, $O(1)$, $O(n^3)$, $O(n^4)$) + $O(n^2)$ = $O(n^4)$.

7.2.4. CNLS

Ahora vamos a calcular la complejidad del algoritmo de búsqueda local, podemos dividirlo en 3 partes: la complejidad del *set up*, la complejidad del ciclo y retornar el resultado.

En el *set up* se crean 3 vectores vacíos con costo $O(1)$ y se ejecuta el algoritmo constructivo que tanto en el caso de GLF como RI, como ya se demostró en la sección de algoritmos golosos, ambos tienen complejidad $O(n^2)$. Por lo tanto, ésta parte tiene complejidad $O(n^2)$.

En la segunda parte, el ciclo del algoritmo, podemos analizar primero la complejidad de las operaciones dentro del mismo.

En primer lugar se copia la CMF a 3 vectores auxiliares con costo $O(n)$ para cada uno.

Se ejecuta *mejorarAgregando*, *mejorarEliminando* y *mejorarIntercambiando* con costo $O(n^3)$, $O(n^3)$ y $O(n^4)$ respectivamente.

Luego de ejecutadas las 3 operaciones, se elegirá cual es la que conviene con costo $O(1)$ y se actualizará la clique con costo $O(n)$. Por lo tanto la complejidad interna del ciclo es $O(n)$ + $O(n^3)$ + $O(n^3)$ + $O(n^4)$ + $O(n)$ = $O(n^4)$

Ahora nos queda ver cuantas iteraciones realiza el ciclo. Dado que depende de que la solución continúe mejorando para seguir ejecutando el ciclo analizaremos hasta cuando puede mejorar la solución. Un grafo siempre tiene una frontera mayor o igual a 0 y menor o igual a la cantidad de aristas m. Como $m \leq n^2$ una cota para la cantidad de mejoras que se pueden hacer a una solución es n^2. Por lo tanto el ciclo puede realizar a lo sumo n^2 iteraciones en la que mejore en 1 la solución en cada iteración, luego de esta cantidad de iteraciones ya no es posible mejorar la solución porque sino tendría que haber más de n^2 aristas y esto no es posible. Por lo tanto el ciclo hace a lo sumo $O(n^2)$.

Entonces la complejidad del ciclo será $O(n^2)$ * $O(n^4)$ = $O(n^6)$

La tercera y última parte es retornar los resultados. Esto no tiene costo significativo ya que copiar el tamaño máximo de la frontera es constante, y la clique se devuelve por referencia.

Podemos concluir entonces que la complejidad de la búsqueda local será entonces $O(n^2)$ + $O(n^6)$ + $O(1)$ = $O(n^6)$

7.3. Experimentos

Dadas las dos heurísticas de búsqueda local planteadas (usando RI y GLF), se realizaron experimentos sobre la complejidad temporal y efectividad de las mismas. Ya que el algoritmo de Búsqueda Local sobre la heurística RI y GLF presentan la misma complejidad temporal, se analizo solo el experimento sobre GLF en este apartado.

7.3.1. Complejidad Temporal

En este experimento se utilizo el marco experimental de Complejidad Temporal descripto en la sección *Marco experimental*.

Se realizo la medición de tiempos utilizando como base el algoritmo GLF

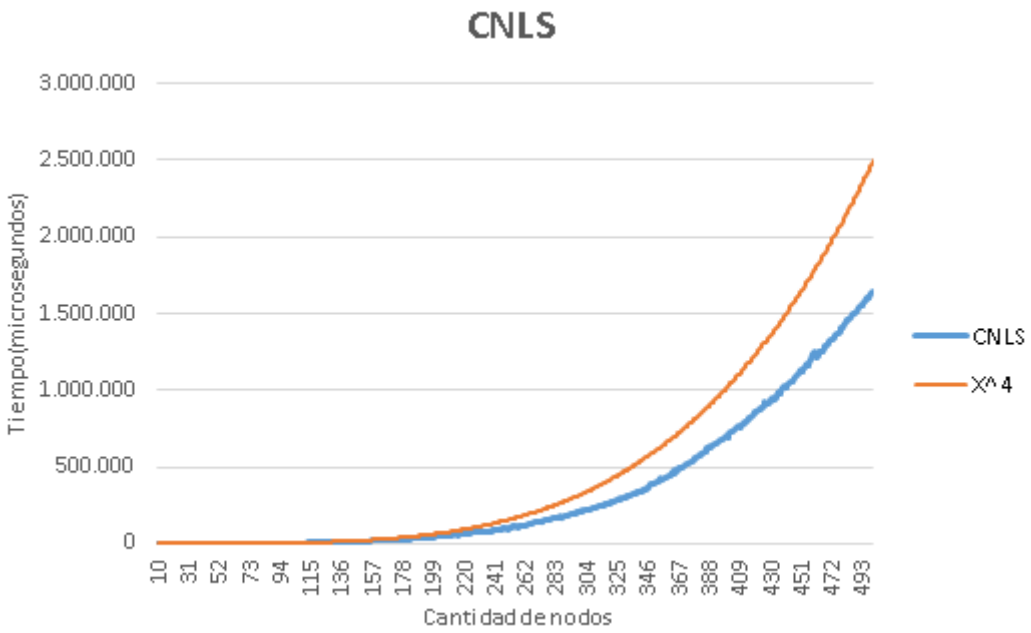


Figura 7: Complejidad temporal en microsegundos de CNLS sobre grafos completos.

En la Figura 7 se aprecia fácilmente como el tiempo de cómputo crece polinomialmente con respecto a la entrada, esto era una característica esperada. En este caso lo comparamos con el polinomio X^4 dado que sobre completos GLF encuentra la solución exacta siempre por lo que CNLS realizará una sola iteración teniendo una complejidad de $O(n^4)$.

Otro punto importante que se puede apreciar en este experimento es que el tiempo de cómputo es órdenes de magnitud mayor que el registrado por nuestros algoritmos GLF y RI. Lo cual, si bien es esperable ya que la complejidad de nuestro algoritmo de búsqueda local es $O(n^6)$, no es un elemento a obviar ya que como se puede apreciar en la Figura 8, los tiempos de computo son significativamente mayores.

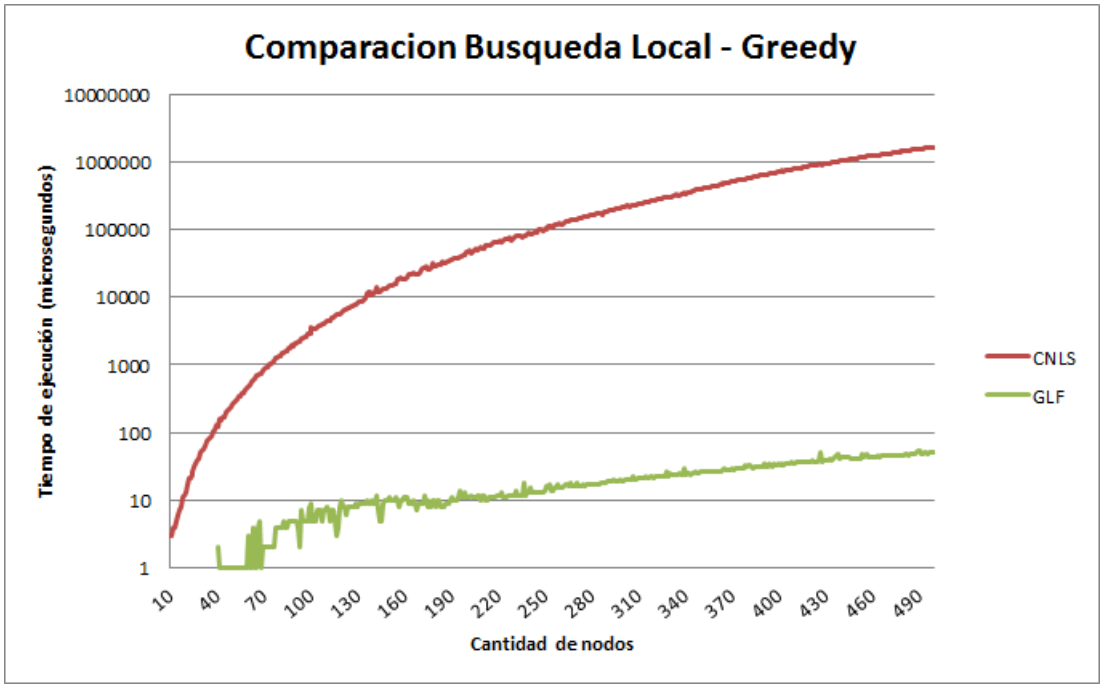


Figura 8: Comparación de tiempos de cómputo en microsegundos de CNLS y GLF

Ahora, un dato a aclarar es que como la medición de tiempos se está realizando sobre grafos completos GLF encuentra siempre la solución exacta por lo que CNLS no puede mejorar la primera solución. Esto se traduce en que CNLS solo realizará una iteración dado que en ésta no podrá mejorar la solución por lo que tendrá una complejidad de $O(n^4)$. Además, como se vio anteriormente, debido a la condición de corte que tiene GLF sobre grafos completos realiza $\frac{n}{2}$ iteraciones por lo que su complejidad siempre resulta menor a $O(n^2)$. Por lo tanto se tiene un algoritmo de complejidad $O(n^4)$ contra uno que es siempre menor a $O(n^2)$, de aquí se explica la diferencia de tiempos obtenida en el gráfico.

7.3.2. Eficacia

En este experimento se utilizo el marco experimental de Eficacia descripto en la sección *Marco experimental*.

Como la búsqueda local actúa como una mejora de nuestros algoritmos constructivos, en este experimento no solamente buscamos ver la eficacia con respecto a la solución exacta, sino también la mejora, si es que la presenta, con respecto al algoritmo base.

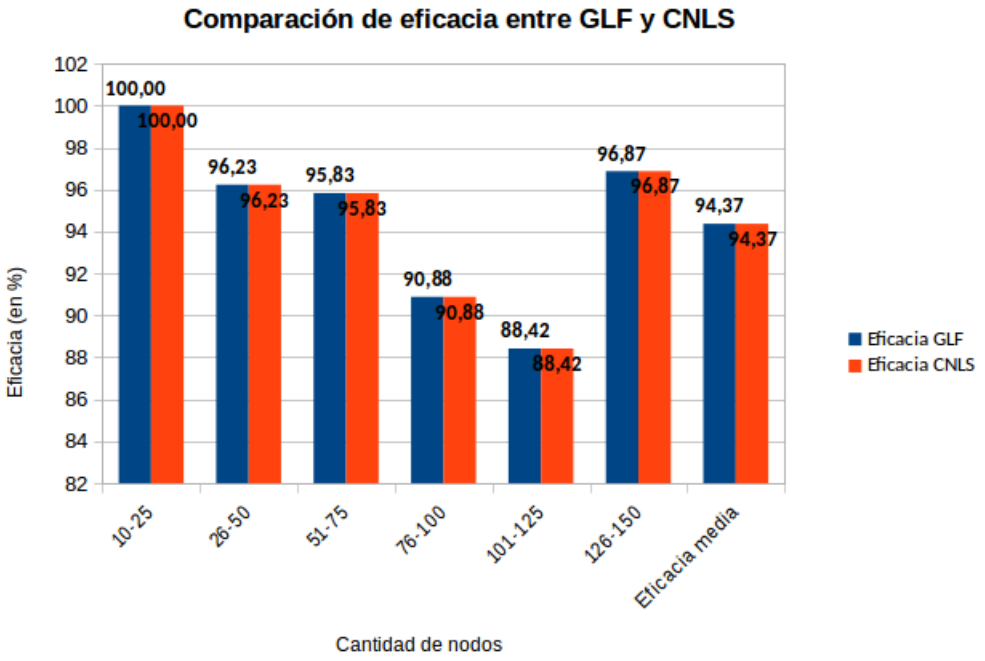


Figura 9: Eficiencia de búsqueda local partiendo de una solución brindada por GLF

En la Figura 9 vemos como el Algoritmo de Búsqueda Local no presenta ninguna mejora con respecto al algoritmo GLF, esto se debe a que el algoritmo GLF ya utiliza la vecindad inmediata, coincidiendo con la considerada por nuestra búsqueda local, la cual a pesar de considerar las posibilidades de agregar, quitar o intercambiar nodos, la cual se puede considerar como una combinación de las dos anteriores. Estas no mejoran la solución ya que de hacerlo el algoritmo GLF las habría utilizado.

Para entender porqué CNLS no arroja ninguna mejora sobre el resultado obtenido con GLF, hay que analizar las operaciones que realiza cada algoritmo. Como ya vimos la búsqueda local analiza 3 tipos de vecindades: agregar nodo, quitar nodo e intercambiar nodos. Para cada una de ellas se puede demostrar que en la primera iteración el algoritmo no tiene chances de mejorar la frontera.

- *Agregar:* Esto es exactamente lo que hace GLF, y dado que el algoritmo llega a una solución maximal, no hay forma de que agregando un nodo se encuentre una clique con mayor frontera.
- *Eliminar:* Sean V los nodos de la clique encontrada por GLF, $k = |V|$ y f el tamaño de su frontera. Por el orden en el que agrega los nodos (primero los de mayor grado), y por la condición de corte, sabemos que $\forall v \in V, d(v) > 2 * (k-1)$. Luego como el tamaño de la frontera al quitar v es: $Front(V - v) = f - (d(v) - (k-1)) + k - 1 = f - d(v) + 2 * (k-1)$. Deducimos que $Front(V - v) < f - 2 * (k-1) + 2 * (k-1) = f$
- *Intercambiar:* Si se intercambia un nodo $v_1 \in V$ por uno $v_2 \notin V$, este nodo v_2 puede que no haya sido agregado por GLF por dos motivos:
  - $d(v_2) < 2 * (k-1)$, por lo tanto $d(v_2) < d(v_1)$ y el intercambio no aumentaría la frontera.
  - Si $d(v_2) \geq 2 * (k-1)$, v_2 no fue agregado a la solución porque no formaba clique con los nodos ya existentes. Para que se pueda agregar a la solución, hay que sacar

un nodo v_1 que al no estar conectado a v_2 no le permitía agregarse a la clique. Sin embargo como v_1 ya estaba en la clique cuando el algoritmo consideró agregar a v_2, $d(v_1) \geq d(v_2)$ por lo que $f \geq front(V - v_1 + v_2)$.

Esto podría hacernos pensar que la búsqueda local no es de utilidad, sin embargo esto se ve que no es cierto en la Figura 10 donde se ve la eficacia de la búsqueda local aplicada a nuestro algoritmo RI.

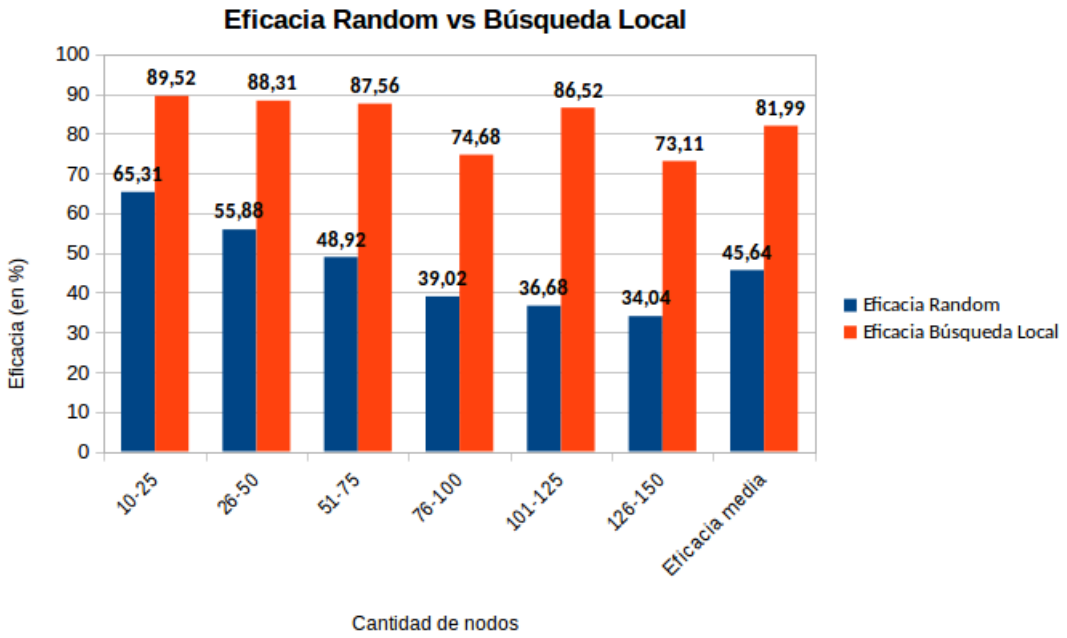


Figura 10: Eficacia de búsqueda local partiendo de una solución brindada por RI

Aquí se ve como mejora notablemente logrando una eficacia media de aproximadamente el doble con respecto al algoritmo RI. Esto se debe a la naturaleza aleatoria del algoritmo RI. En este caso nuestra vecindad resulta de utilidad, ya que puede considerar la mejor de las posibilidades de los nodos frontera.

Esto nos permite concluir que la efectividad de la Búsqueda Local depende casi exclusivamente de la correcta elección de la vecindad, la cual debe ser pensada especialmente en función del algoritmo base utilizado según las flaquezas que éste presente.

8. Metaheurísticas

8.1. GRASP

Dado que ninguna heurística garantiza soluciones de calidad para cualquier tipo de grafo, en este algoritmo nos concentraremos en utilizar las heurísticas de forma inteligente intentando que la solución brindada sea siempre de buena calidad sin importar el grafo que se establece en la entrada.

Para ello nos basaremos en el algoritmo de GRASP (*Greedy Randomized Adaptive Search Procedure*) el cual genera soluciones greedys partiendo desde distintos nodos del grafo buscando cubrir la mayor cantidad de soluciones posibles devolviendo finalmente la mejor solución.

Una parte fundamental de GRASP es el algoritmo que genera las soluciones greedy aleatorias, dado que debe de poder generar soluciones lo suficientemente diferentes para que se garantice explorar un amplio espectro del espacio de soluciones.

En nuestro algoritmo, nuestra solución aleatoria se construye eligiendo un nodo al azar de una lista de nodos[5] que no fueron utilizados en las últimas n soluciones y luego agregando sus vecinos que más mejoren la solución. Se puede ver cómo funciona el algoritmo en el siguiente pseudocódigo:

```
function CMFRandomGreedy(cliqueMF)
    si NoUsados es Vacio entonces
        para Desde i entre 1 y n hacer
            noUsados ← AgregarAtras(noUsados, i)
        fin
        noUsados ← Mezclar(noUsados)
    fin
    nodo ← Ultimo(noUsados);
    noUsados ← EliminarUltimo(noUsados);
    clique ← Vacio;
    clique ← AgregarAtras(clique, nodo);
    maxFront ← TamFrontera(clique);
    tamClique ← 1;
    vecinos ← dameVecinosCliqueyGrado(clique);
    vecinos ← OrdenarporGrado(Vecinos);
    para i desde 1 hasta Tamaño(vecinos) y mientras vecinos_i pueda mejorar la solución
hacer
        clique ← AgregarAtras(vecinos_i, clique) ;
        si clique es clique y tiene mayor frontera que antes entonces
            Actualizo maxFront ;
            tamClique ← tamClique + 1;
        fin
        en otro caso
            clique ← EliminarUltimo(clique)
        fin
    fin
    cliqueMF ← clique;
    devolver maxFront
```

Algoritmo 8: Algoritmo greedy aleatorio

Otra parte importante de GRASP es cuándo termina el algoritmo, es decir, cuándo consideramos que ya buscó una cantidad suficientemente buena de soluciones diferentes por lo que ya deberíamos obtener una solución de buena calidad con alta probabilidad. Ńuestro algoritmo terminará luego de buscar una cantidad de soluciones sin que se mejore la mejor solución obte-

[5]Se mezclan los nodos usando la operación random_shuffle de C++ con costo $O(n)$. Más información en http://www.cplusplus.com/reference/algorithm/random_shuffle/

nida hasta el momento. Esta cantidad de iteraciones sin mejorar la solución se determinará en la entrada y se verá luego las variaciones en el rendimiento y en las soluciones que se obtienen al modificar este valor.

Finalmente, el algoritmo de GRASP que realizamos se puede resumir de la siguiente forma:

```
function CMFGRASP(cliqueMF, limite)
    maxFront ← maxFronCliqueLFS(cliqueMF);
    maxFrontAux ← 0;
    CMFAux ← Vacio;
    iterSinMejorar ← 0;
    mientras iterSinMejorar <limite hacer
        maxFrontAux ← CMFRandomGreedy(CMFAux);
        maxFrontAux ← MejorarBusqLocalVecChica(CMFAux, maxFrontAux);
        si maxFrontAux >maxFront entonces
            maxFront ← maxFrontAux;
            Actualizo cliqueMF
        fin
        en otro caso
            iterSinMejorar ← iterSinMejorar + 1
        fin
        Vaciar(CMFAux)
    fin
    devolver maxFront
```

Algoritmo 9: Algoritmo GRASP

8.2. Complejidad

Vamos a analizar la complejidad del algoritmo dividiendo el algoritmo en partes. En primer lugar comenzaremos con la inicialización de variables donde se crean vectores vacíos y enteros con costo $O(1)$ salvo por el llamado a $maxFronCliqueLFS$ el cual tiene un costo de $O(n^2)$. La complejidad de esta parte será entonces $O(n^2)$

Luego estaremos en el ciclo, comenzaremos por ver el costo de las operaciones dentro del mismo.

Se llama a la función $CMFRandomGreedy$ la cual inicia llenando un vector vacío con n nodos con costo $O(n)$ y luego mezcla los elementos del mismo también con costo $O(n)$. Se realizan asignaciones y se elimina el último elemento de un vector con costo $O(1)$. Se calcula la frontera de la clique de tamaño 1 generada hasta el momento con costo $O(n)$, se buscan los vecinos de la clique y su grado con costo $O(n^2)$ dado que la clique tiene un solo nodo. Se ordenan los vecinos de forma decreciente según su grado en $O(nlogn)$. Luego se ingresa a un ciclo que realizará a lo sumo n iteraciones donde la única operación que no es elemental es ver si se forma una clique al agregar un nodo que tiene costo $O(n)$. Al salir del ciclo se copia la mejor solución en $O(n)$. Por lo tanto la complejidad de $CMFRandomGreedy$ sera $O(n) + O(n) + O(1) + O(n^2) + O(nlogn) + O(n) * O(n) + O(n) = O(n^2)$.

Se llamará a la búsqueda local para la que ya se mostró que tiene complejidad $O(n^6)$

Luego se copiará la solución obtenida en esta iteración si mejora la mejor solución con costo $O(n)$.

Nos queda ver cuantas iteraciones hará a lo sumo este ciclo, por un lado depende de un parámetro l que se establece en la entrada que indica cuantas iteraciones realizará el ciclo sin que se mejore la solución, por el otro lado debemos saber cuantas iteraciones se pueden hacer mejorando la solución, por la forma en que funciona $CMFRandomGreedy$ puede generar n soluciones distintas por lo que luego de n iteraciones donde se mejore siempre la solución es imposible que se siga mejorando. Por lo tanto el ciclo hace a lo sumo $n + l$ iteraciones.

Entonces el costo total del ciclo será de $O(n+l) * (O(n^2) + O(n^6) + O(n)) = O((n+l) * n^6)$

Finalmente se devuelve la solución con costo $O(1)$

Concluimos que la complejidad total del algoritmo será de $O(n^2) + O((n+l) * n^6) + O(1) = O((n+l) * n^6)$.

8.3. Experimentación

En una primera medida quisimos ver qué tan buenas eran las soluciones provistas por esta metaheurística con respecto a la solución exacta.

Para poner a prueba esto, generamos grafos de 10 a 150 nodos con cantidad y distribución aleatoria de las aristas. Para cada grafo calculamos la eficacia de GRASP obteniendo lo siguiente:

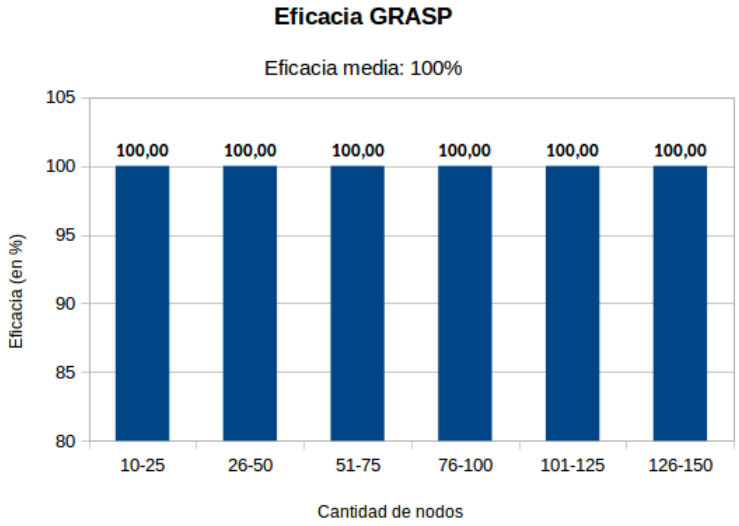


Figura 11: Eficacia de GRASP con respecto a la solución exacta.

Podemos ver que para todas las instancias de prueba nuestro algoritmo dió con la solución exacta. Esto no significa que GRASP sea una solución exacta al problema de CMF sino que para el conjunto de grafos utilizado siempre se encontró la mejor solución, puede que para algún grafo particular GRASP no llegue siempre a la solución exacta.

Dado que el algoritmo GRASP encuentra una solución greedy en cada iteración y luego intenta mejorarla con búsqueda local, y como se vio en experimentos anteriores que la búsqueda local no mejora a las soluciones calculadas de forma golosa, queremos ver como varía en tiempos de ejecución y eficacia la utilización o no de la búsqueda local en GRASP.

En primer lugar medimos los tiempos de ejecución para grafos completos de entre 10 y 280 nodos utilizando búsqueda local en GRASP y no utilizándola. Se pueden ver los resultados en el siguiente gráfico:

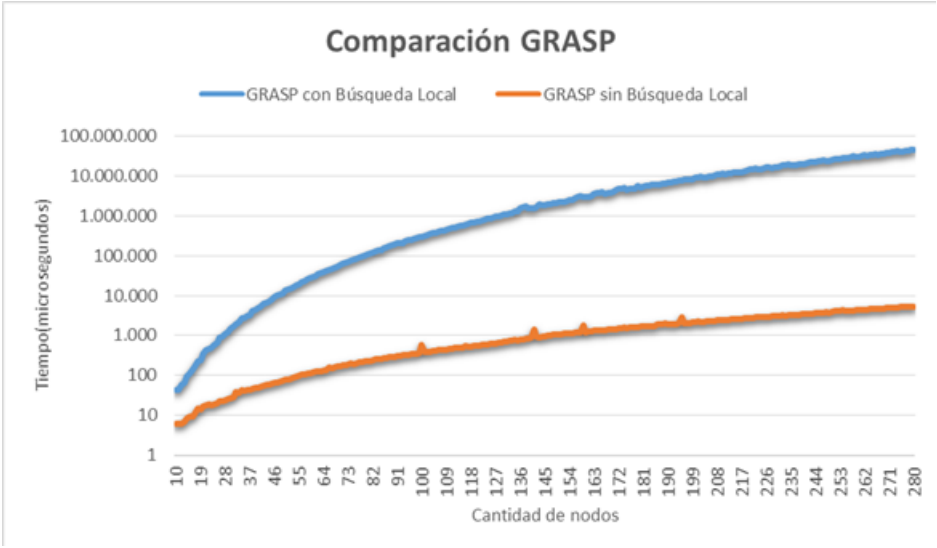


Figura 12: Comparación de tiempos de cómputo en milisegundos entre metaheurísticas GRASP utilizando y sin utilizar búsqueda local sobre grafos completos.

Podemos ver que disminuyen mucho los tiempos de ejecución al no utilizar la búsqueda local, esto se debe a que con la búsqueda local el algoritmo GRASP tiene una complejidad $O((n+l)*n^6)$ mientras que sin la búsqueda local la complejidad del mismo es de $O((n+l)*n^2)$ donde l es el límite de iteraciones que puede realizar el algoritmo sin mejorar una solución. Dado que se está utilizando n como límite, la complejidad de GRASP será de $O(n^7)$ con búsqueda local y $O(n^3)$ sin la misma.

También queremos ver si al quitar la búsqueda local la eficacia obtenida con GRASP varía, por lo que tomaremos como instancias de prueba los mismos grafos de 10 a 150 nodos con una cantidad aleatoria de aristas generados anteriormente y veremos la eficacia utilizando GRASP sin usar búsqueda local.

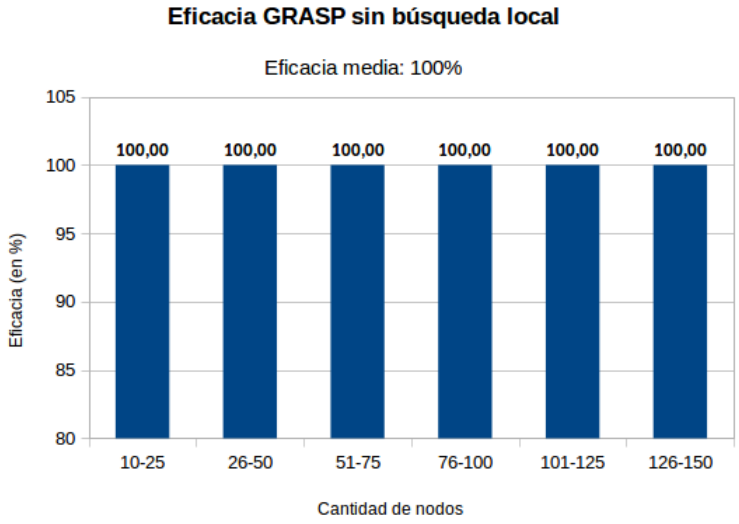


Figura 13: Eficacia de GRASP con respecto a la solución exacta.

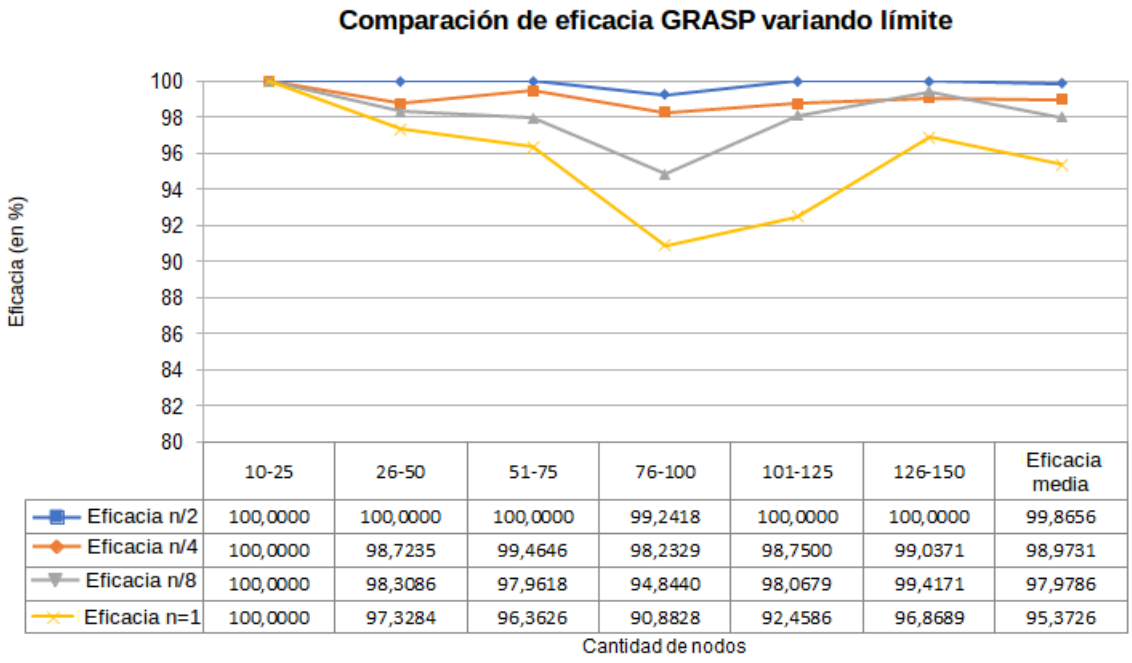
Ya que al quitar la búsqueda local disminuyen los tiempos de ejecución y se mantiene la

misma eficacia encontrando una solución óptima para todos los grafos de prueba, consideramos una optimización para el algoritmo de GRASP el quitar la búsqueda local del mismo.

Ahora queremos ver la variación que obtenemos en la eficacia si variamos la cantidad de iteraciones que realiza GRASP, esto lo logramos al modificar el parámetro *límite* el cual indica luego de cuántas iteraciones sin mejorar una solución va a continuarse ejecutando GRASP. Por defecto establecimos n como límite para las experimentaciones previas dado que esto garantiza que el algoritmo *greedy* aleatorio explore todo el espacio de soluciones que puede generar ya que en cada iteración inicia desde 1 nodo diferente.

Si disminuimos el valor del *límite*, el algoritmo realizará menos iteraciones por lo que los tiempos de ejecución serán menores. Sin embargo, puede que se encuentre o no la mejor solución dado que no se está explorando todo el espacio de soluciones que puede encontrar el algoritmo *greedy* aleatorio, dependerá de que el nodo que elija como inicial la heurística constructiva *random* en alguna de las iteraciones sea parte de alguna solución óptima.

Para poner a prueba esto, fuimos disminuyendo el valor de *límite* y viendo la eficacia que obteníamos para los mismos grafos para los que habíamos obtenido una solución exacta en todos los casos con un limite igual a n.



	10-25	26-50	51-75	76-100	101-125	126-150	Eficacia media
Eficacia n/2	100,0000	100,0000	100,0000	99,2418	100,0000	100,0000	99,8656
Eficacia n/4	100,0000	98,7235	99,4646	98,2329	98,7500	99,0371	98,9731
Eficacia n/8	100,0000	98,3086	97,9618	94,8440	98,0679	99,4171	97,9786
Eficacia n=1	100,0000	97,3284	96,3626	90,8828	92,4586	96,8689	95,3726

Figura 14: Variación en la eficacia de GRASP al variar el límite

Se puede ver que a medida que se disminuye el *límite* por lo general baja la eficacia de GRASP. Igualmente, puede que al ejecutar GRASP con un *límite* bajo para algún grafo siga dando una solución exacta, pero la probabilidad de esto será menor.

Ahora veremos la variación de tiempos al disminuir el *límite*, para ello corrimos nuevamente el algoritmo GRASP variando su *límite* sobre grafos completos obteniendo los siguientes resultados:

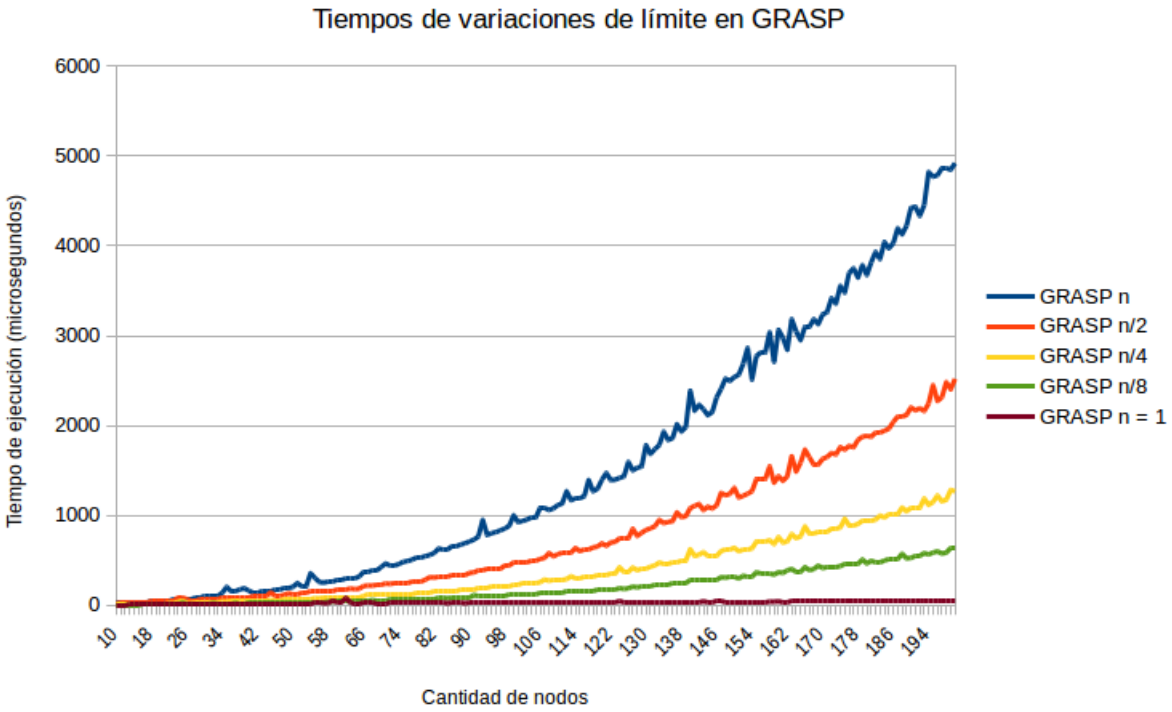


Figura 15: Variación de tiempos al modificar límite de GRASP

Es bastante notable cómo disminuyen los tiempos al bajar el *límite*, esto se debe a que si bien la complejidad de GRASP sin búsqueda local es de $O((n+l)*n^2)$ donde l es el límite, el n que se encuentra dentro de $(n+l)$ solo ocurre en el peor caso de que se mejore la solución n veces sin llegar al límite de iteraciones sin mejorar, en el promedio de los casos es más probable que finalice el ciclo dado que se alcanzó el límite que puede haber de iteraciones sin mejorar la solución.

Por lo tanto, a la hora de utilizar GRASP debemos de tener cuidado sobre los resultados que queremos obtener. Si necesitamos soluciones de la mejor calidad posible será mejor utilizar un *limite* cercano a la cantidad de nodos, si lo que necesitamos es un buen rendimiento temporal deberíamos disminuir el *límite* pudiendo llegar a perder un poco de precisión.

8.4. Configuración óptima

Luego de analizar los experimentos realizados podemos concluir que la mejor configuración que podemos tomar para GRASP será la siguiente:

- Se quitará la búsqueda local del ciclo de GRASP ya que se mostró que no mejora la soluciones obtenidas a través de GLF y tampoco desde el algoritmo greedy aleatorio de GRASP dado que realiza lo mismo que GRASP salvo por el primer nodo que agrega a la clique.
- Se dejará como límite a la cantidad de nodos n del grafo, esto se debe a que de todas las configuraciones probadas fue la única que nos dió una eficacia del 100 % para todos los grafos generados aleatoriamente. Si bien para otras configuraciones se obtuvieron tiempos de ejecución menores vamos a anteponer la eficacia frente a los tiempos de cómputo dado que ya tenemos al algoritmo GLF que nos da soluciones con una eficacia superior al 90 % para los mismos grafos. La única configuración probada para GRASP que nos da una

complejidad cercana a $O(n^2)$ en algunos casos (pero incluso su peor caso sigue siendo $O(n^3)$) es con el límite = 1 que en los casos en que no se mejora la solución en la primer iteración es igual a ejecutar GLF 2 veces salvo que la segunda ejecución se inicia desde un nodo aleatorio por lo que podríamos perder un poco la eficacia esperada para GRASP.

Para poner a prueba esta configuración generamos un nuevo conjunto de grafos random y comparamos la eficacia de GRASP con la de GLF dado que fue el otro algoritmo con el que obtuvimos buenos resultados en tiempos menores. No realizamos una comparación de tiempos dado que ya se vió anteriormente que los tiempos de cómputo de GLF siempre serán menores o iguales a los de GRASP.

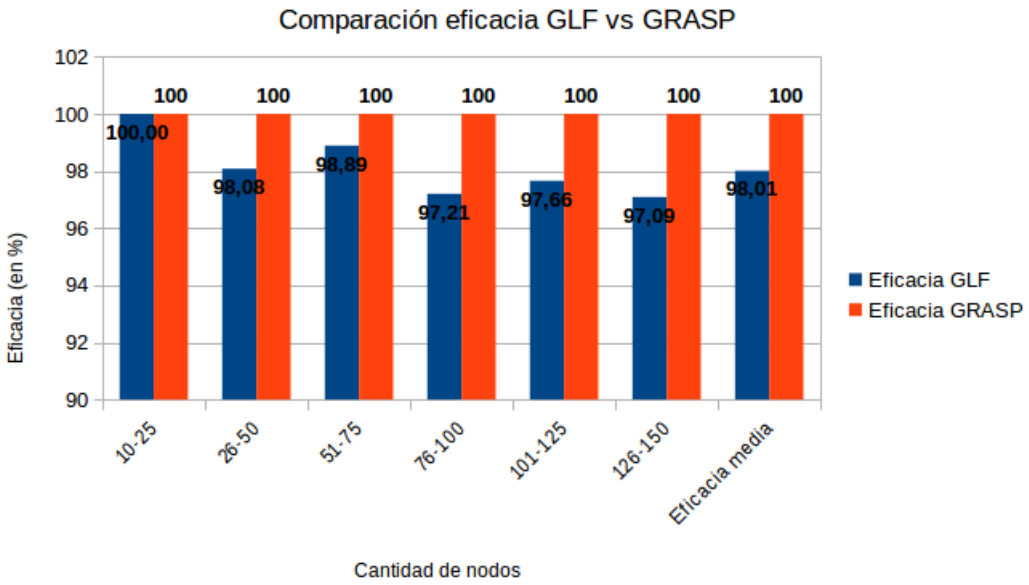


Figura 16: Comparación de eficacia entre GRASP y GLF para nuevos grafos random

Se puede ver que obtuvimos una eficacia del 100 % para todos los grafos en GRASP frente a una eficacia superior al 95 % en GLF. Con esto podemos concluir que con esta configuración GRASP sigue siendo el mejor algoritmo que tenemos si lo que necesitamos es eficacia, y además al quitar la búsqueda local bajamos su complejidad a $O(n^3)$ por lo que los tiempos de ejecución también son aceptables.

Cabe aclarar que si bien se obtuvo una eficacia del 100 % en todos los experimentos realizados, esto no significa que GRASP resuelva el problema de CMF para todo grafo en tiempo polinomial. Existen grafos para los que GRASP no da una solución exacta, más adelante se intentará construir alguno de ellos en la sección de grafos particulares.

9. Grafos particulares

A continuación se describirán como se comportan nuestros algoritmos frente a familias de grafos conocidas o para tipos de grafos para los que hayamos encontrado alguna particularidad.

9.1. Grafos completos

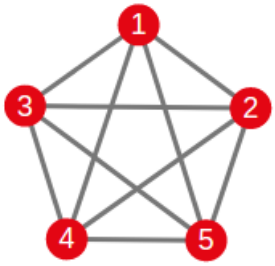


Figura 17: Grafo completo conocido como K_5

Para esta familia de grafos se tiene el caso de que todos los nodos se conectan con el resto por lo que tienen grado $n-1$, esto hace que cualquier subconjunto de nodos del grafo sea una clique por lo que en principio serían muchas cliques para las cuales se debiera chequear su frontera. Sin embargo se puede ver de forma analítica que para grafos completos de tamaño mayor a 1 la máxima frontera será siempre $\lfloor n^2/4 \rfloor$ donde n es la cantidad de nodos. Esto se logra al formar una clique con la mitad de los nodos del grafo si n es par o $\lfloor n/2 \rfloor$ si es impar, cada uno tiene $n/2$ o $\lceil n/2 \rceil$ vecinos respectivamente, de aquí es que se obtiene la frontera $\lfloor n^2/4 \rfloor$. Esta no puede aumentar más ya que si se agrega un nodo más se le quitan $n/2$ aristas a la máxima frontera anterior ya que el nodo que se agrega era vecino de todos para agregarle $n/2-1$ aristas correspondientes a los vecinos del nodo a agregar que no pertenecen a la clique, así queda demostrado que la frontera no puede mejorar más.

Como todos los nodos tienen el mismo grado, tanto el algoritmo GLF como el RI comenzarán por alguno de estos nodos y seguirá agregando nodos a la clique hasta que deje de mejorar la frontera, cuando deje de mejorar se habrá llegado a la solución óptima dado que en los completos la frontera siempre mejora al agregar un nodo hasta que se llegó a la mejor solución.

Como GLF encuentra una solución óptima, también lo harán la búsqueda local y GRASP ya que parten de una solución generada por GLF.

9.2. Grafos estrella

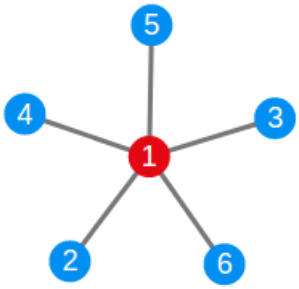


Figura 18: Grafo estrella donde el nodo rojo es la CMF

En este tipo de grafos en que el nodo de grado máximo es siempre la CMF, el algoritmo GLF siempre encontrará la solución óptima ya que comenzará a construir una solución a partir de este nodo y al no obtener mejoras en la frontera se quedará con esta clique de tamaño 1.

Dado que tanto la búsqueda local como GRASP parten de una solución generada con GLF, al ser esta solución óptima ya no podrán mejorarla pero igualmente devolverán la solución óptima.

En cuanto al algoritmo RI, solo llegará a la solución óptima si elige como nodo inicial para agregar a la clique al nodo de mayor grado, caso contrario la solución brindada sera $CMF - 1$ donde CMF es la máxima frontera real, dado que al iniciar de otro nodo se agregará el nodo de grado máximo a la solución ya que es el único vecino del nodo inicial y mejorará la frontera, pero se estará ocupando una arista para unir a estos 2 nodos que debía ser parte de la máxima frontera.

9.3. Circuitos simples

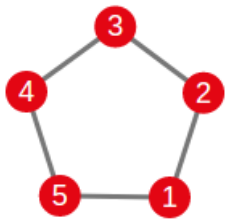


Figura 19: Circuito simple conocido como C_5

En este caso cada nodo del grafo tiene grado 2 por lo que la máxima frontera será siempre 2 y la clique de máxima frontera puede ser cualquier nodo del grafo o cualquier clique de tamaño 2.

Por lo tanto, el algoritmo GLF encontrará siempre la solución óptima, dado que todos los nodos tienen el mismo grado iniciará de algún nodo y esa ya será una solución óptima. Esto hace que tanto la búsqueda local como GRASP también devuelvan una solución óptima.

El algoritmo RI también encontrará siempre la mejor solución ya que cualquier nodo desde el que comience a construir la clique ya será una solución óptima.

9.4. Unión de estrellas

Luego de haber analizado los resultados que obtuvimos al experimentar sobre grafos generados de forma aleatoria notamos que el algoritmo GLF en muchos casos llega a una solución exacta, pero en los casos en que la solución encontrada no fue la mejor se debía a que el nodo de grado máximo por el que se comienza a construir la clique no era parte de ninguna solución exacta.

En base a estos resultados, ideamos un tipo de grafo para que el algoritmo GLF siempre de una solución que no es óptima. Consiste en un grafo estrella cuyo nodo central tiene el grado máximo en el grafo que se une mediante uno de los nodos de las puntas a una clique de 2 nodos donde cada nodo tiene grado $grado\ máximo\ -1$. Por lo tanto GLF devolverá como solución a $grado\ máximo$ mientras que la solución exacta será la provista por la clique de tamaño 2, que es $(2 * grado\ máximo) - 2$.

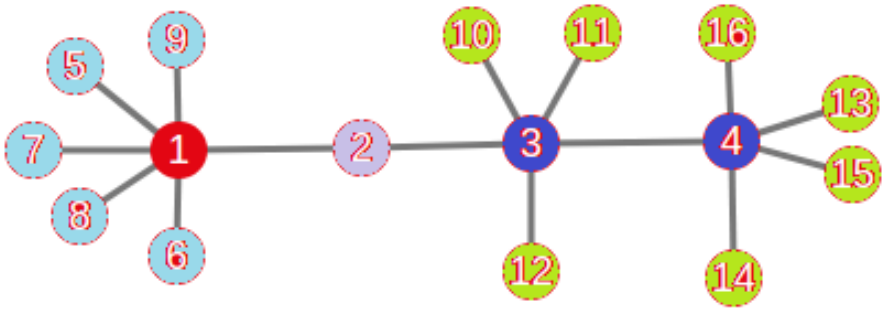


Figura 20: Grafo que denominamos unión de estrellas

En el grafo de ejemplo se puede ver que por cada nodo celeste que se agrega se puede agregar un nodo verde a cada nodo azul. Por lo tanto, a medida que generemos grafos de este estilo de mayor cantidad de nodos, GLF se alejará más de la solución exacta.

Es claro que la búsqueda local no mejorará la solución provista por GLF ya que no hay ningún nodo en la vecindad cercana que mejore la solución.

El algoritmo GRASP debería llegar a la solución exacta si en alguna iteración la heurística random toma como nodo inicial alguna de los 2 nodos que pertenecen a la clique de tamaño 2 formada por los nodos azules.

Por otro lado, RI solo llegará a la solución exacta si inicia desde uno de los 2 nodos azules, por lo tanto la probabilidad de que RI brinde la solución exacta es de $2/n$ donde n es la cantidad de nodos del grafo.

Para poner a prueba esto, generamos grafos de la familia *unión de estrellas* de entre 16 y 199 nodos y verificamos la eficacia de nuestros algoritmos para resolverlos.

En primer lugar comparamos la eficacia de GLF contra GRASP dado que son los algoritmos para los cuales obtuvimos una mejor eficacia hasta el momento.

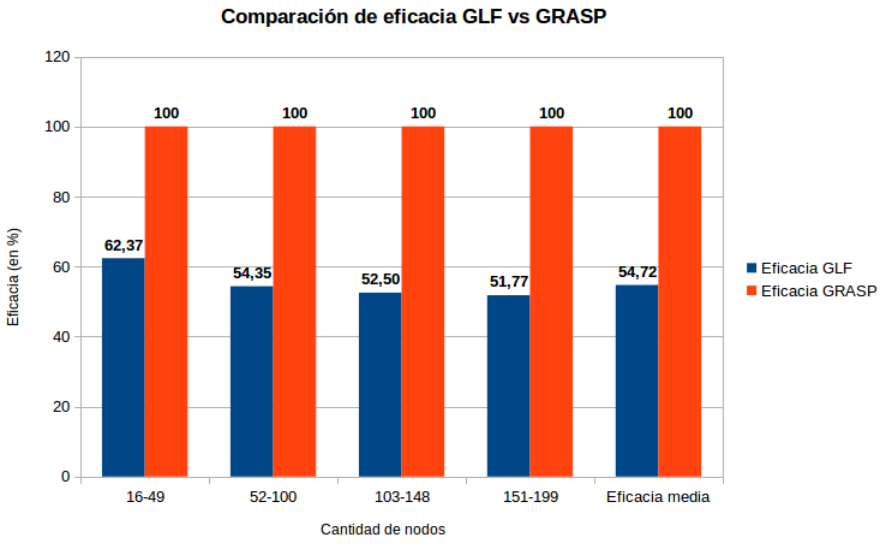


Figura 21: Comparación de eficacia entre GLF y GRASP para unión de estrellas

Se puede ver que realmente este tipo de grafos es un peor caso en el sentido de la eficacia

para GLF dado que anteriormente había dado una eficacia media cercana al 90 % y ahora con estos grafos tiene una eficacia cercana al 55 %. Por otro lado se puede ver que GRASP sigue llegando a la solución exacta para este tipo de grafos por lo que representa una mejora de eficacia considerable con respecto a GLF.

Es posible corregir el mal rendimiento de GLF con respecto a estos grafos, por ejemplo además de formar una clique desde el nodo de máximo grado se puede armar una segunda clique iniciando desde el nodo de máximo grado que no pertenece a la primer clique. La complejidad seguiría siendo $O(n^2)$ dado que se harían operaciones similares a 2 ejecuciones del GLF normal.

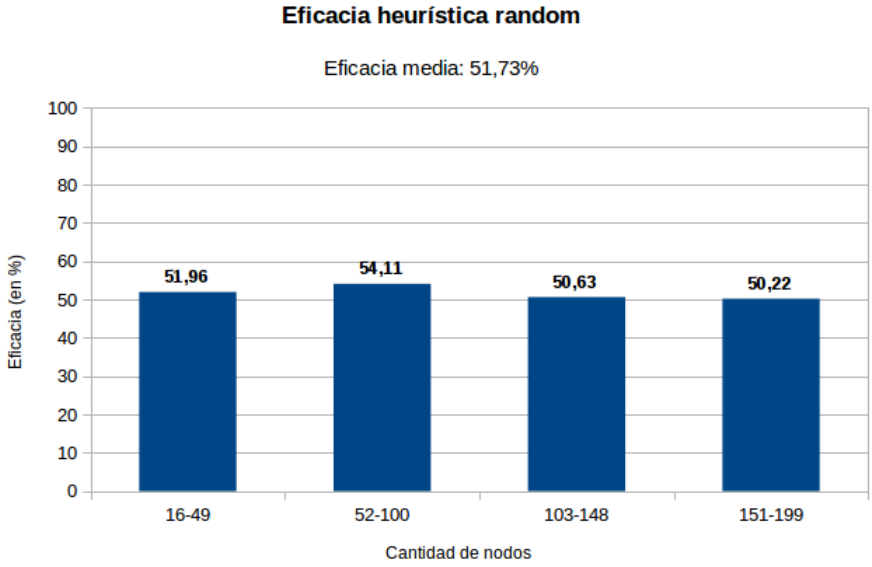


Figura 22: Eficacia en RI para unión de estrellas

Por otro lado quisimos ver la eficacia de RI con este tipo de grafos y podemos ver que la eficacia también es cercana al 50 % por lo que en este caso RI tiene una eficacia similar que GLF.

9.5. Tipo MM

Con el objetivo de encontrar grafos donde GRASP no obtenga la solución adecuada, construimos este nuevo tipo de grafos, al cual llamamos MM. Sabiendo que nuestra mejor configuración de GRASP básicamente ejecuta el algoritmo Greedy tomando como nodo inicial un nodo distinto del grafo en cada instancia, estos grafos se construyeron de forma tal de que aunque cualquier nodo de la Clique de Máxima Frontera sea tomado como punto de inicio, el siguiente nodo elegido por el algoritmo Greedy aleatorio sea uno no perteneciente a la CMF.

Para esto a la CMF se la llamo Clique Interna (CI), la cual tendrá tamaño n. Para cada nodo perteneciente a la CI, se le conectan $n-1$ nuevos nodos, los cuales tienen grado $d=1$, exceptuando el nodo V el cual tiene grado $d=K$.

De esta forma obtenemos si queremos que cualquier nodo de la CI elija un nodo V, como nodo a agregar necesitamos que el nodo V tenga mayor grado que cualquiera perteneciente a CI, por lo tanto:

$$K > (n-1) + (n-1)$$
$$K > (2 * n) - 2$$
$$K \geq (2 * n) - 1$$

Tomamos como K mínimo $K = (2 * n) - 1$. En la Figura 23 se muestra cómo se conforma un Grafo MM, mostrando la configuración de los nodos externos para un nodo particular de la Clique Interna.

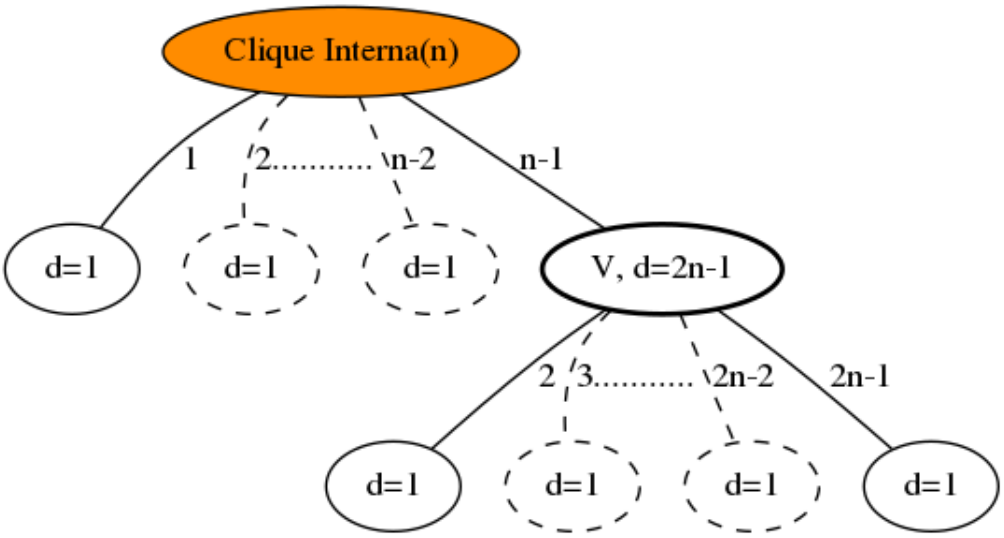


Figura 23: Forma Genérica de un Grafo MM

Ahora solo falta ver que el tamaño de la Frontera de CI n es mayor que la frontera de la clique entre un nodo de CI y su correspondiente V, llamémosla Clique Externa(CE).

Frontera(CE): $F(CE) = (n-1) + ((n-1)-1) + (((2*n)-1)-1) = 4*n-5$

Frontera(CI): $F(CI) = (n-1)*n$

$$(n-1)*n > (4*n) - 5$$
$$n^2 - n > (4*n) - 5$$
$$n^2 - (5*n) + 5 > 0$$
$$n \geq 4$$

De esta forma obtenemos un tipo de grafo, en el cual a partir de $n \geq 4$ se obtiene una frontera mayor en CI, pero en el que GRASP dará como resultado el tamaño de la frontera de CE.
Una observación que se puede realizar sobre los Grafos MM es que cualquier subgrafo completo de CI de $(n-1)$ nodos, también posee Frontera igual a $(n*(n-1)$ en el grafo original.

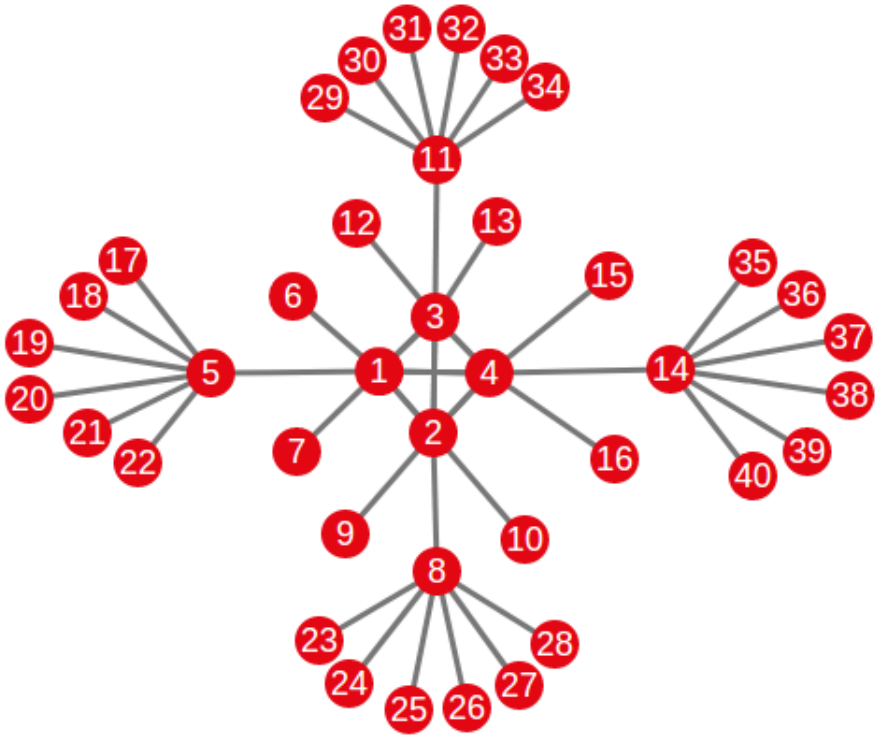


Figura 24: Grafo MM, con Clique Interna de 4 nodos

En la Figura 24 se muestra a modo de ejemplo un grafo MM con una Clique Interna de 4 nodos.

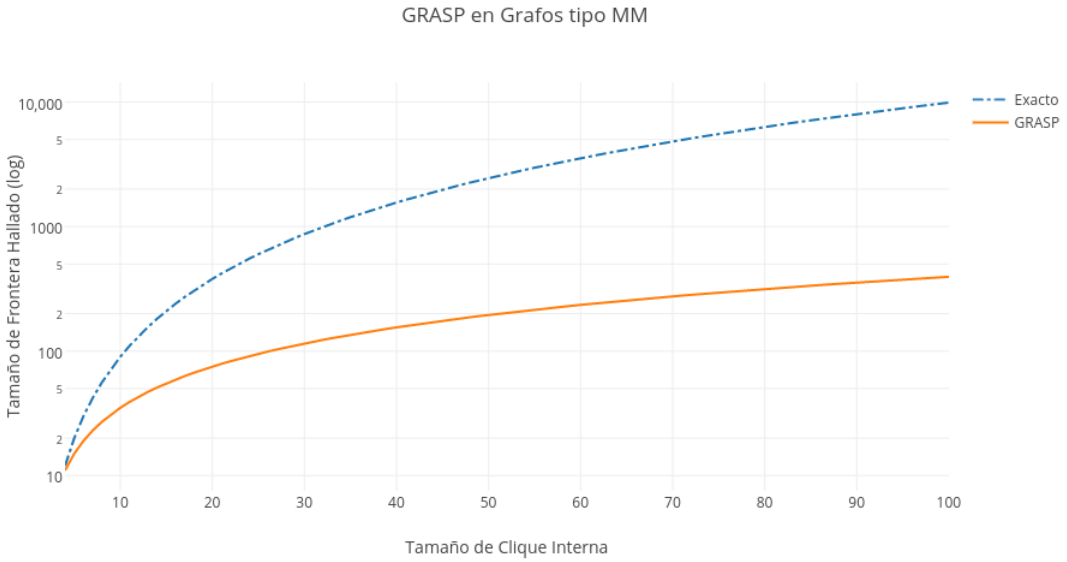


Figura 25: Comparacion GRASP vs Exacto, en función del tamaño de CI

Como la solución hallada por GRASP va a ser de orden lineal con respecto al tamaño de

CI, mientras que la solución exacta es de orden cuadrático(ver Figura 25), a medida que CI crezca GRASP perderá progresivamente eficacia con respecto a la solución hallada. Esto mismo se puede observar en la Figura 26. Esto si bien muestra una baja eficacia para GRASP, no es determinante, ya que estos son casos forzados para que el mismo falle, de forma tal que en la mayoria de los casos este algoritmo sigue siendo la mejor opción a la hora de calcular la CMF.

Ahora, vamos a analizar como se comportarían el resto de los algoritmos realizados.

GLF va tomar a unos de los nodos de mayor grado para iniciar por lo que la clique que devolverá será una CE la cual no es la CMF.

CNLS tampoco encontrará la solución exacta dado que como ya se vio anteriormente iniciará con una solución provista por GLF la cual no podrá mejorar.

RI puede encontrar la solución exacta si el orden en que se agregan los nodos a la posible clique comienza con un nodo que pertenece a la CI y el siguiente nodo que se encuentra en la lista mezclada de nodos que forma una clique con el nodo de CI que ya se agregó a la clique también pertenece a CI. Aquí se terminará formando la clique CI dado que no hay otro nodo del grafo que forme una clique con 2 nodos de CI. Sin embargo, dado que depende del orden aleatorio que se le da a la lista de nodos esto es poco probable. Más precisamente la probabilidad de que suceda esto es si el grafo tiene t nodos es $\frac{|CI|}{t} \times \frac{|CI|-1}{2*|CI|-1} = \frac{|CI|}{2t}$ la cual es una probabilidad muy baja dado que $t = |CI| + |CI| * (|CI| - 1) + |CI| * (2|CI| - 2)$. Por ejemplo si |CI| = 10, la probabilidad de que esto suceda es aproximadamente de 0,018 y a medida que aumente el tamaño de la clique interna la probabilidad será aun menor.

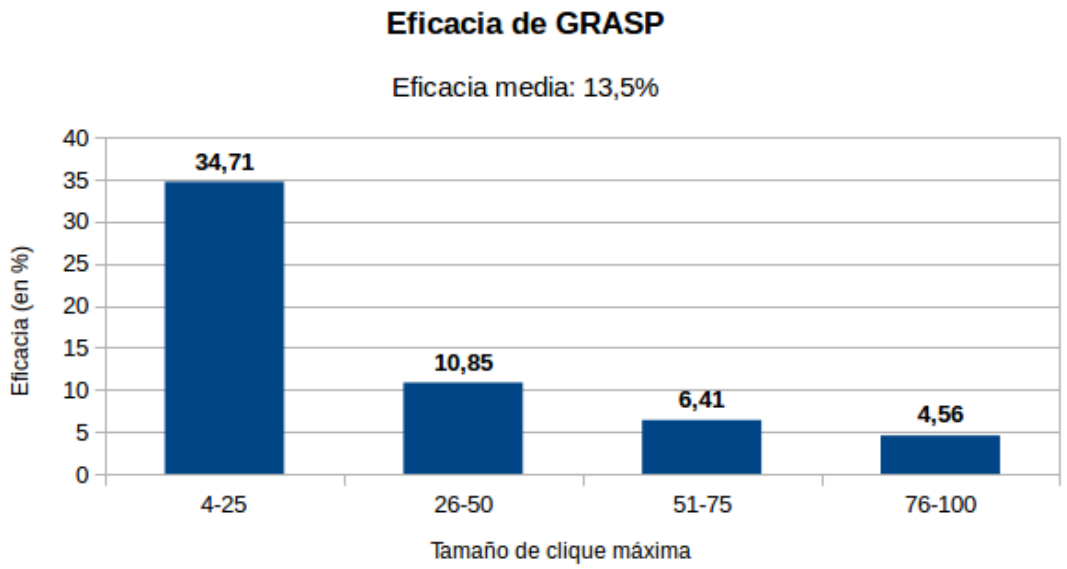


Figura 26: Eficacia GRASP en Grafos MM

10. Conclusión

En este trabajo ha quedado en evidencia lo útiles que pueden ser las heurísticas como herramienta para resolver problemas, especialmente problemas difíciles que son muy costosos de resolver de forma exacta.

Si bien en muchos escenarios uno necesariamente requiere una solución exacta, en otros a veces es necesario priorizar la velocidad de cómputo por sobre la calidad de la solución. Es en ese momento cuando las heurísticas surgen como una alternativa muy viable. Lo aprendido en este trabajo es que puede haber muchas heurísticas distintas que resuelvan el mismo problema, y siempre es posible hallar una que se ajuste a las necesidades de uno.

En nuestro caso, por ejemplo, si se requiriera resolver el problema de *clique de máxima frontera* priorizando fuertemente el tiempo que se tarda en resolverlo, en un grafo del que no tenemos información adicional, se podría aplicar el algoritmo goloso GLF, el cual provee resultados de alta calidad (más de 90 % de efectividad en el caso promedio) a un muy bajo costo. Si, en cambio, se deseara priorizar la calidad de las soluciones permitiéndose pagar un costo temporal un poco más alto, entonces se podría utilizar el algoritmo GRASP con parámetro de corte límite igual a n el cual devuelve soluciones muy precisas (100 % de efectividad en la mayoría de los casos) y, si bien tiene un costo superior a GLF, también tiene complejidad polinomial.

Otra conclusión que se puede llegar sobre las heurísticas es que, como no proveen soluciones exactas, a veces presentan serias flaquezas ante diversos escenarios específicos, pero esto no es motivo para no utilizarlas, ya que, como se vio en este trabajo, se pueden combinar varias heurísticas o aplicar unas a otras para obtener una mejor solución ante escenarios adversos.

Esto resulta en una amplia forma de tratar distintos problemas ya que, en algunos casos, técnicas algorítmicas tan simples y naturales como iterar la misma heurística desde distintas instancias iniciales puede mejorar notablemente la solución y al mismo tiempo resguardarnos de los casos problemáticos que presentaba la heurística original.