Algoritmos y Estructuras de Datos III
TP3

19 de julio de 2015

Integrante	LU	Correo electrónico
Martin Baigorria	575/14	martinbaigorria@gmail.com
Federico Beuter	827/13	federicobeuter@gmail.com
Juan Rinaudo	864/13	jangamesdev@gmail.com
Mauro Cherubini	835/13	cheru.mf@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Introducción

1.1. Definiciones

Antes de enunciar el problema a resolver en este trabajo práctico, es necesario definir algunos conceptos.
Sea $G = (V, E)$ un grafo simple:

Definición Un conjunto $I \subseteq V$ es un *conjunto independiente* de G si no existe ningún eje de E entre los vértices de I. Es decir, los ejes de I no están conectados por las aristas de G.

Definición Un conjunto $D \subseteq V$ es un *conjunto dominante* de G si todo vértice de G esta en D o bien tiene al menos un vecino que está en D.

Definición Un conjunto *conjunto independiente dominante* de G es un conjunto independiente que a su vez es dominante del grafo G. Desde un conjunto independiente dominante se puede acceder a cualquier vértice del grafo G. Esto se debe a que el vértice pertenece al conjunto o se puede acceder con sólo recorrer una arista desde uno de sus vértices.

Definición Un *Conjunto Independiente Dominante Mínimo* (CIDM) es el conjunto independiente dominante de G de mínima cardinalidad.

1.2. Introducción

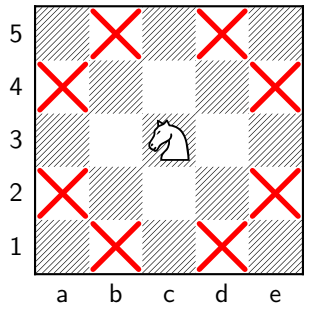
En 1979, Garey y Johnson probaron que el problema de encontrar el CIDM de un grafo es un problema NP-Hard[1]. El objetivo del trabajo es utilizar diferentes técnicas algorítmicas para resolver este problema. En un principio diseñaremos e implementaremos un algoritmo exacto para el mismo. Dada la complejidad del problema, luego propondremos diferentes algoritmos heurísticos para llegar a una solución que sea lo suficientemente buena a fines prácticos en un tiempo razonable.

1.2.1. El señor de los caballos

Si recordamos el problema 3 del TP1, podemos ver claramente que el mismo es un caso particular del problema del conjunto dominante mínimo. El problema consistía en cubrir un tablero de ajedrez con la menor cantidad de caballos posible dados ciertos casilleros que ya estaban cubiertos por caballos de forma tal que todo casillero este ocupado por un caballo o pueda ser accedido por un caballo en otro casillero.

Modelado

El problema fue modelado con un grafo, donde cada casilla era representada por un vertice y el movimiento de los caballos se modelaba con aristas entre nodos. Esto se puede ver en la siguiente figura:



En este caso, el caballo ubicado en el centro de la figura solo tiene 8 movimientos validos que aparecen marcados en la figura con una cruz roja. Al modelar este problema con grafos, cada casilla es adyacente a las casillas que podrían ser accedidas con un movimiento de caballo si hubiese un caballo en esa posición.

[1] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman and Company, San Francisco (1979).

Dominancia

Consideraremos que una casilla esta en el conjunto solución si contiene un caballo. El problema pide que el conjunto final de caballos (o casillas, vértices) sea dominante. Esto se debe a que toda casilla en el tablero debe estar en el conjunto o debe poder ser accedida desde una casilla 'adyacente'.

Independencia

Este no es un caso del CIDM dado que la solución óptima al problema (la menor cantidad de caballos para cubrir el tablero) no necesariamente era independiente. Por ejemplo, los caballos dados al principio del problema no necesariamente son independientes. Por lo tanto, al buscar la solución estaríamos buscando el CDM del grafo.

1.3. Maximalidad y dominancia

Las siguientes proposiciones serán útiles a lo largo del trabajo:

Proposición 1.1 *Sea M un conjunto independiente maximal de G. $\forall v \in G.V$, si $v \notin M \implies \exists u \in M$ tal que u es adyacente a v.*

Demostración Por absurdo. Sea M un conjunto independiente maximal y v un vértice tal que $v \in G.V \wedge v \notin M$. $\nexists u \in M$ tal que u es adyacente a v. Por lo tanto, puedo agregar v a M y el conjunto va a seguir siendo independiente. Esto es absurdo, dado que el conjunto era maximal.

Proposición 1.2 *Dado $G(V, E)$, todo conjunto independiente maximal es un conjunto independiente dominante.*

Demostración Sea M un conjunto independiente maximal. Dado $v \in G.V$, por la propiedad anterior, si $v \notin M \implies \exists u \in M$ tal que u es adyacente a v. Por lo tanto, si $v \notin M$ entonces v tiene algún vecino que está en M. Esto significa que M es dominante.

1.4. Modelado

Muchos problemas se pueden modelar con grafos y se pueden resolver mediante la búsqueda del conjunto independiente dominante mínimo.

1.4.1. Planificador Urbano

Supongamos que un planificador urbano esta diseñando una ciudad con muchos barrios. Con el objetivo de proveer un buen sistema de salud para los habitantes, el planificador determina que cada habitante debe tener que cruzar a lo sumo un barrio para acceder a un hospital público, que cada barrio puede tener a lo sumo un hospital público y que no es eficiente en terminos de costos que existan dos hospitales en dos barrios adyacentes. Aquí podemos modelar a cada barrio con un vértice, y representar la adyacencia entre barrios con una arista. Al obtener el CIDM, obtenemos la ubicación y la mínima cantidad de hospitales públicos necesarios para cumplir con los objetivos del planificador.

1.4.2. Policia

La Policía Federal y la Policía Metropolitana finalmente deciden trabajar en conjunto para mejorar la seguridad en la Ciudad de Buenos Aires. Con el objetivo de mejorar el tiempo de respuesta ante hechos de inseguridad graves, se decide reasignar un conjunto de efectivos policiales para resguardar las zonas con altos indices de inseguridad. Estos efectivos se deben distribuir de forma tal que ningún efectivo tenga que cruzar mas de una zona para atender una situación delictiva.

Debido a que los efectivos se ponen a charlar y se distraen cuando están en dos zonas adyacentes, los jefes policiales deciden que dos efectivos no pueden estar ubicados en zonas adyacentes. A su vez, los jefes policiales buscan utilizar la menor cantidad de recursos posibles.

Por lo tanto, este problema se puede resolver modelándolo con grafos y buscando el CIDM. Cada zona puede ser representada por un vértice, y la adyacencia entre zonas por aristas. Estamos buscando un conjunto dominante dado que se deben resguardar todas estas zonas (suponemos que tenemos la cantidad de efectivos suficiente). Ademas, este conjunto sera independiente dado que si no los efectivos se ponen a charlar y no trabajan.

2. Experimentación

Como ya hemos mencionado, a lo largo de este trabajo practico analizaremos diferentes tipos de estrategias para resolver el problema del CIDM. Para poder comparar entre estrategias, experimentaremos con cada algoritmo utilizando 6 familias de grafos diferentes, las cuales serán descriptas a continuación.

2.1. Grafos Aleatorios

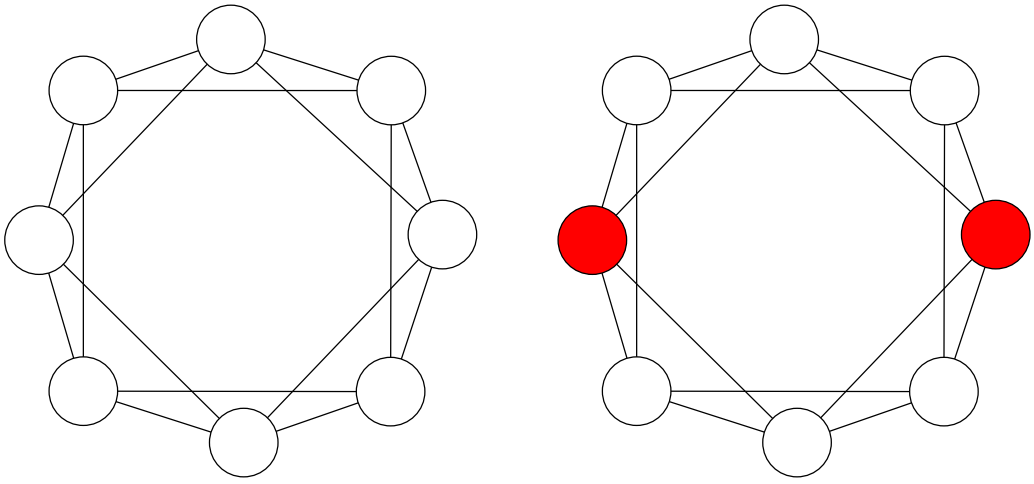
Se tomaron grafos aleatorios para poder experimentar con situaciones mas generales, en donde el resultado es usualmente impredecible a simple vista. Los grafos aleatorios son generados en base a dos variables, estas son:

- La cantidad de nodos (de aquí en adelante n)
- La cantidad de conexiones entre nodos (de aquí en adelante m)

Para la experimentación se decidió variar el n principalmente, sin embargo, también es pertinente investigar el impacto que pueden llegar a tener la cantidad de conexiones en el grafo. Para cada valor de n se generaron 3 grafos con diferentes valores de m, estos fueron $\frac{n}{2}$, n y $2n$.

2.2. Grafos d-regulares conexos

Esta familia fue elegida ya que podemos dar la cantidad de nodos que conforman el CIDM, esta es $\lceil \frac{n}{d} \rceil$ **FALTA JUSTIFICAR**. Podemos ver esto en el siguiente ejemplo:



(a) $n = 8, d = 4$ (b) $\lceil \frac{8}{4} \rceil = 2$

Esto nos permite hacer un análisis sobre el tamaño de las soluciones, permitiéndonos tener una mejor perspectiva a la hora de elegir la mejor configuración. Al igual que con los aleatorios, estos grafos poseen dos variable, las cuales son:

- La cantidad de nodos
- El grado de los nodos (la variable d)

Se siguió una metodología similar que en el caso aleatorio, es decir, para cada n se tomaron 3 valores de d, que fueron $\frac{n}{4}$, $\frac{n}{2}$ y $\frac{3n}{4}$.

2.3. Grafos bipartitos completos

Al igual que en el caso anterior, la principal razón por la cual decidimos probar con esta familia es que podemos determinar el tamaño de la solución de antemano. Como el grafo es bipartito completo, alcanza con tomar todos los nodos de alguno de los dos conjuntos para poder obtener un CIDM, es decir que para todo grafo bipartito completo la solución va a ser el tamaño del conjunto mas pequeño. Las variables involucradas en este caso son:

- La cantidad de nodos en la primer componente
- La cantidad de nodos en la segunda componente

Para la experimentación se vario la cantidad de nodos de la primer componente, y para cada una de ellas se generaron dos grafos bipartitos completos con $\frac{n}{4}$ y $\frac{3n}{4}$.

2.4. Arboles binarios

A diferencia de las dos familias anteriores, donde el tamaño de la solución es único, aquí nos encontramos con un caso donde hay mas de una. Si en un árbol tomamos todos los nodos de un nivel, en el próximo no seria necesario tomar ninguno en el próximo, este patrón se repita hasta llegar al ultimo nivel. Dependiendo de la cantidad de niveles, esto nos permite dar una cota inferior para el CIDM, estas son:

- Si la cantidad de niveles es par ($log_2(n) \mod 2 = 0$), la cota inferior es $\sum_{i=0}^{\frac{log_2(n)}{2}-1} 2^{2i}$
- Si la cantidad de niveles es impar ($log_2(n) \mod 2 \neq 0$), la cota inferior es $\sum_{i=0}^{\lfloor \frac{log_2(n)}{2} \rfloor - 1} 2^{2i+1}$

Podemos ver esto en el siguiente ejemplo:

Para todo árbol sabemos que la cantidad de conexiones es igual a la cantidad de nodos menos uno, es por esto que para la experimentación se vario únicamente la cantidad de nodos.

2.5. Cliques

Al igual que los casos anteriores, se probo también con cliques ya que para cualquier grafo de esta familia sabemos el tamaño de la solución. Para cualquier clique alcanza con tomar un nodo para poder obtener un CIDM, ya que desde esto nodo puedo alcanzar el resto de los nodos del grafo. La idea detrás de experimentar con esta familia es probar la eficiencia de los algoritmos, ya que los mismos deberían poder resolver estos grafos de manera veloz y eficiente.

Al igual que con los arboles binarios, para generar estos grafos solo entra en juego una única variable, y es la cantidad de nodos en el mismo. Al ser una clique, la cantidad de conexiones es siempre $\frac{n(n-1)}{2}$.

2.6. Grafos unión de componentes conexas

Por ultimo se decidió probar con un grafo formado por varias componente conexas, unidas por puentes. Cada una de las componentes conexas es un C_i, para generar estos grafos se crean sucesivos C_i, tomando inicialmente $i = 1$ y aumentando la cantidad de nodos siempre y cuando la cantidad total de nodos los permita. Una vez que las tenemos generadas, se la comienza a unir de manera sucesiva, es decir, se une C_1 con C_2, C_2 con C_3, así hasta llegar al ultimo camino generado. La motivación para probar esta familia es poder analizar el impacto que puede llegar a tener la resolución de cada una de las componentes, teniendo en cuenta la presencia de ejes puentes, los cuales pueden afectar el tiempo que toma resolver el grafo.

Esta familia se reservo para poder hacer `cross-validation` de las mejores configuraciones posibles.

2.7. Metodología

Para hacer el análisis, se hizo variar el valor de n entre 10 y 220. Si el generador para alguna de las familias recibe un segundo parámetro, los utilizados son los mencionados en la sección de cada familia. Para cada valor de n se corrió el algoritmo 100 veces y se tomo el promedio del tiempo.

3. Algoritmo Exacto

3.1. Algoritmo

Utilizando backtracking, recorremos todas los conjuntos dominantes independientes y luego seleccionamos el de menor cardinalidad. Representamos al grafo con un arreglo $graph[n]$ de nodos. Cada nodo tiene los siguientes atributos:

1. adj: Lista de nodos adyacentes al nodo actual.
2. degree: Grado del nodo actual.
3. added: Bool que indica si el nodo ha sido agregado al conjunto que representa el cubrimiento.
4. reachable: Bool que indica si el nodo actual puede ser alcanzado desde un nodo perteneciente al cubrimiento.

Comenzamos definiendo la función *backtracking*, que lo que hace es tomar un nodo del grafo, y luego considera los casos en los que el nodo pertenece o no a un posible cubrimiento. En caso de agregar el nodo al cubrimiento, todos los nodos adyacentes al mismo son ignorados en futuras llamadas recursivas. Si consideráramos los nodos adyacentes, romperíamos la independencia de los cubrimientos y además no solo incrementaría la complejidad del código sino que también el tiempo de ejecución del mismo.

3.2. Podas y estrategias

Para poder resolver el problema lo mas rápido posible, en primer lugar buscamos una forma rápida de verificar si un conjunto solución encontrado es independiente. En vez de tener que verificarlo, decidimos forzar la independencia por construcción. Esto se logró evitando los nodos adyacentes a los que ya agregó el algoritmo al potencial conjunto solución. De esta forma mantenemos la independencia del conjunto y evitamos tener que agregar innecesariamente muchos nodos.

Otro problema importante es verificar si los nodos seleccionados forman un cubrimiento. Esto lo resolvimos simplemente haciendo que la función backtracking lleve un contador con el total de nodos alcanzables por el cubrimiento. Este contador lo incrementamos cada vez que agregamos un nodo, considerando todos sus adyacentes que aún no hemos clasificado como alcanzables. Si ese número es igual al número total de nodos, significa que llegamos a un cubrimiento. De esta manera evitamos funciones auxiliares que tengan que verificar si los nodos seleccionados hasta ahora forman un cubrimiento, y a su vez sabemos que por construcción el mismo es independiente.

Además, antes de comenzar la búsqueda agregamos todos los vértices de $d(v) = 0$ al conjunto solución final. Esto se debe a que estos vértices necesariamente estarán en la solución. Es muy simple probar esto, dado que si no lo estuvieran, algún vértice adyacente debería estar en el conjunto para que lo cubra. Sin embargo, tal vértice no existe. El costo de hacer esto es $\mathcal{O}(n)$, dado que solo tenemos que recorrer un arreglo de nodos una vez, verificando su atributo de grado.

Una poda muy común que también hemos implementado es la de la solución local actual. Dada una solución posible (que aún no sabemos si es la mínima), si en el estado actual del algoritmo se está considerando un número de vértices que no le puede ganar a esta solución, ignoramos esa rama del árbol de estados posibles. Esto se puede verificar en $\mathcal{O}(1)$, dado que solo hay que comparar el numero actual de nodos agregados con el numero de nodos en la mejor solución encontrada hasta el momento.

3.3. Complejidad

3.3.1. Pseudocódigo

```
procedure BACKTRACKING(G, nodoActual, nodosCubiertos, nodosUsados, solucionLocal, nodosEnSolucion)
    if nodoActual == G.size then
        return
    if G[nodoActual].alcanzable == true then
        return backtracking(G, nodoActual + 1, nodosCubiertos, nodosUsados, solucionLocal, nodosEnSolucion)
    if nodosUsados + 1 == nodosUsadosEnSolucion) then
        return
    G[current].added ← true
    agregados ← 0
    adjNodes ← emptyList()
    for all adj ∈ G[current].adj do
        if G[adj].alcanzable == false then
            G[adj].alcanzable ← true
            added.push_front(adj)
            agregados++
    if nodosCubiertos + agregados + 1 == n then
        nodosEnSolucion++
        solucionLocal ← G
    else
        backtracking(G, nodoActual + 1, nodosCubiertos, nodosUsados + 1, solucionLocal, nodosEnSolucion)
    for all e ∈ adjNodes do
        G[e] ← false
    backtracking(G, nodoActual + 1, nodosCubiertos, nodosUsados, solucionLocal, nodosEnSolucion)
```

3.3.2. Complejidad Espacial

Para la representación del grafo, utilizamos un arreglo de nodos. Cada nodo tiene una lista de adyacencia. Por lo tanto, la complejidad espacial de nuestro algoritmo es de $\mathcal{O}(n + 2m)$, donde n es la cantidad total de vértices y m la cantidad total de aristas.

3.3.3. Complejidad Temporal

Al utilizar backtracking, si no consideramos ninguna poda recorremos todos los conjuntos independientes y dominantes una vez. Esto lo hacemos iterando un arreglo de nodos. Al agregar un nodo, marcamos a todos sus adyacentes como alcanzables y seguimos con el siguiente nodo.

Cada llamada recursiva (agrego o no el proximo nodo) tiene como mínimo un costo de $\mathcal{O}(\Delta(G))$ [2]. Esto se debe a que en primer lugar modificamos el grafo agregando un nodo y modificando los atributos de a lo sumo $\Delta(G)$ nodos adyacentes. Al finalizar la llamada recursiva, debemos restaurar el atributo `reachable` de a lo sumo $\Delta(G)$ nodos.

Las podas que se aplican dentro de las llamadas recursivas no empeoran la complejidad del algoritmo, dado que estan en $\mathcal{O}(1)$. La efectividad de las mismas la mostraremos luego en la experimentación.

Cuando encuentra un conjunto solución, lo guarda en una estructura auxiliar en $\mathcal{O}(n)$. En el peor de los casos, el algoritmo recorre todos los conjuntos independientes y dominantes, comenzando con el de mayor cardinalidad. Cada vez que lo encuentra, actualiza la estructura donde guardamos la solución. Para que esto suceda, en realidad todos los conjuntos dominantes deben tener diferente cardinalidad, cosa que en general no sucede. Como todo conjunto de n elementos tiene 2^n subconjuntos, utilizaremos esto para acotar la cantidad de veces que actualiza la solución local. Seguramente hay una cota teórica mucho mejor.

Sin considerar que hay ramas que ignoramos en cada llamada recursiva al forzar la independencia del conjunto por construcción, tenemos a lo sumo 2^n llamadas. Cada llamada debe restaurar el grafo a su estado original al finalizar y/o guardar la solución actual. Por lo tanto, el algoritmo pertenece a $\mathcal{O}((n + \Delta(G)) \times 2^n)$, que es lo mismo que $\mathcal{O}(n \times 2^n)$. Esto se podría reducir primero buscando el tamaño de la solución optima, y luego buscándola a $\mathcal{O}(2^n)$. Sin embargo no seria efectivo en términos de tiempo dado que deberíamos rearmar gran parte del árbol nuevamente.

[2] $\Delta(G)$ denota el máximo grado de un vértice perteneciente al grafo.

4. Heurística Constructiva Golosa

4.1. Algoritmos

Dado que el problema de buscar el CIDM es NP-Hard, para este algoritmo resolveremos el problema por medio de una heurística golosa. La idea básicamente fue ir seleccionando nodos bajo algún criterio, agregándolos al conjunto solución y descartando todos los otros nodos que romperían la independencia de la potencial solución condicional al nodo agregado. Al quedarnos sin nodos para elegir, finalmente tendríamos un conjunto independiente y dominante válido. Dado que la heurística no siempre es optima, muchas veces sucederá que este conjunto que encontraremos no sera el mínimo. Este procedimiento se puede ver en el siguiente pseudocódigo: Para elegir que nodo seleccionar dado los nodos disponibles, utilizaremos la selección por `grado` y por `scoring`, que explicaremos a continuación.

4.1.1. Por grado

Criterio

Al principio decidimos implementar este criterio de selección de nodos utilizando un heap, ordenando los nodos por su grado. Esto se puede hacer facilmente con el algoritmo de Floyd en Luego, desencolamos del heap y vamos actualizando los flags de cada nodo a medida que son alcanzables. El algoritmo tiene $\mathcal{O}(n \times log(n) + m)$.

Pseudocodigo

```
procedure greedyHeapConstructive(G)
    nodeHeap ← buildHeap(G.V)
    while !nodeHeap.isEmpty() do
        node ← nodeHeap.pop()
        if node.reachable == true then
            continue
        node.added = true
        for all adj ∈ node.adj do
            adj.reachable ← true
```

4.1.2. Scoring

Criterio

Aunque este método con el heap es rápido, en realidad podemos mejorar la forma en la que seleccionamos los vértices. Este método consiste en tomar el número de nodos adyacentes efectivos (score) a los que cada nodo puede acceder. Definimos a un nodo adyacente efectivo como un nodo que es adyacente y a su vez no puede ser accedido por otros nodos que ya pertenecen a la solución parcial en construcción. De esta forma, este criterio también nos garantiza la independencia del conjunto, dado que si tomamos dos nodos de la solución, por construcción no pueden ser adyacentes.

Cada nodo va a tener como atributos su score, un flag que indica si ha sido agregado y otro que indica si es alcanzable por el cubrimiento parcial actual.

El algoritmo va a iterar un arreglo de nodos n^2 veces. Cada vez que busquemos un nodo para agregar al conjunto, los iteraremos todos para buscar el de máximo score. Al identificarlo, actualizaremos los scores de los nodos adyacentes a los adyacentes del mismo. A priori parece que la complejidad de este nuevo algoritmo se podría mejorar de forma significativa utilizando algún otro tipo de estructura de datos.

Pseudocodigo

```
procedure greedyConstructive(G)
    for i = 0 to i < G.size() do
        greatest ← 0
        score ← 0
        flag ← false
        for j = 0 to j < G.size() do
            if graph[j] == true then
                continue
            if graph[j].score ≥ score then
                greatest ← j
                score ← graph[j].score
```

```
        flag ← true
    if !flag then
        break
    graph[greatest].added ← true
    graph[greatest].reachable ← true
    for all adjNode ∈ graph[greatest].adj do
        adjNode.reachable ← true
        for all adjToAdj ∈ adjNode.adj do
            adjToAdj.score–
```

4.2. Complejidad

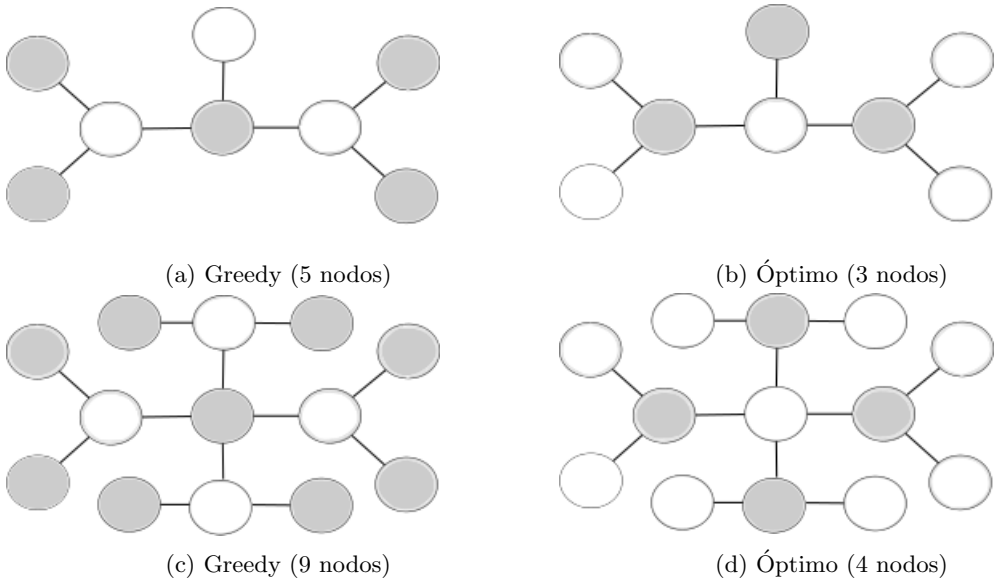
El primer algoritmo resuelve el problema en $\mathcal{O}(n \times log(n) + m)$ simplemente ignorando la actualización de los scores, desencolando de un heap n veces. Sin embargo, este criterio es a simple vista inferior que el de actualización de scores. Aquí hay un tradeoff entre hacer la mejor elección y la complejidad temporal del algoritmo.

El algoritmo basado en el score recorre arreglo n veces. A su vez, buscar los adyacentes de los adyacentes se hace m veces en total. Luego actualizamos en total el score de m nodos. Por lo tanto, el algoritmo tiene orden $\mathcal{O}(n^2 + 2 \times m)$, es decir $\mathcal{O}(n^2)$.

Notar que la forma en que buscamos el máximo es sumamente ineficiente. Esto se debe a que si utilizamos sort, luego es bastante difícil encontrar el nodo al que le debemos actualizar su respectivo score. A su vez, dado que en cada iteración actualizamos el score, mantener el orden es sumamente costoso. Es muy posible que exista una estructura de datos mucho más eficiente para resolver este problema (una especie de heap dinámico), aunque para este trabajo practico nos conformaremos con el algoritmo en $\mathcal{O}(n^2)$.

4.3. Efectividad de la heurística

Nuestra heurística no siempre devuelve la solución optima. Considerar los siguientes ejemplos:



(a) Greedy (5 nodos)

(b) Óptimo (3 nodos)

(c) Greedy (9 nodos)

(d) Óptimo (4 nodos)

Figura 2: Ejemplos de nuestra heurística comparado con el óptimo.

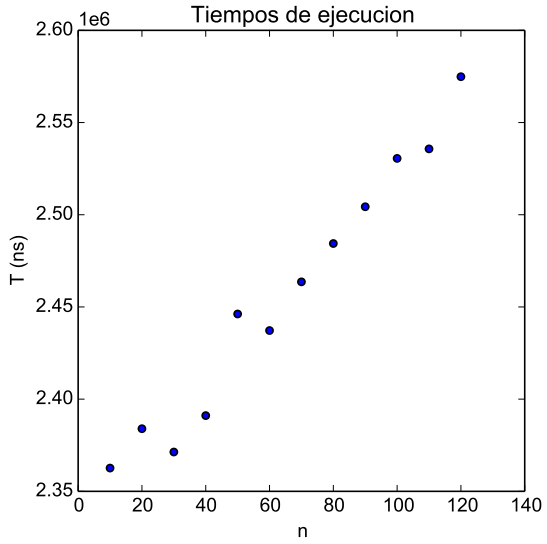
El peor caso es claramente el de la figura (c) y (d). Tenemos un nodo v (el nodo central) de grado $d(v) = 4$, con sus nodos adyacentes de grado $d(u) = 3$. Si tenemos c componentes conexas de ese tipo, utilizaremos $c \times (2 \times 4 + 1)$ nodos, cuando en realidad el optimo tiene $c \times 4$ nodos.

4.4. Experimentación

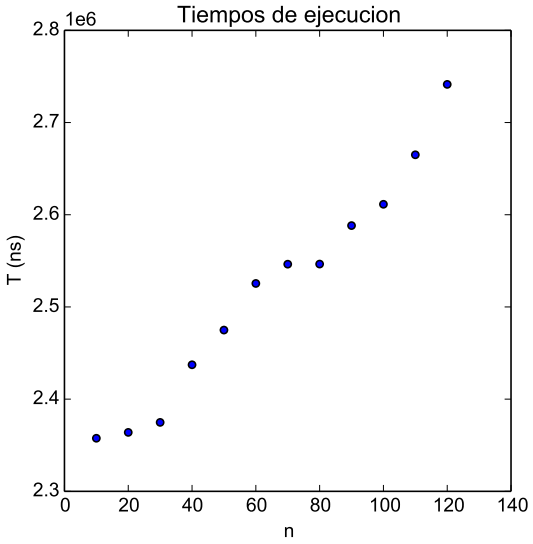
Para la experimentación se siguió con la metodología indicada anteriormente. Los resultados fueron los siguientes.

4.4.1. Heuristica Constructiva Golosa por Grado

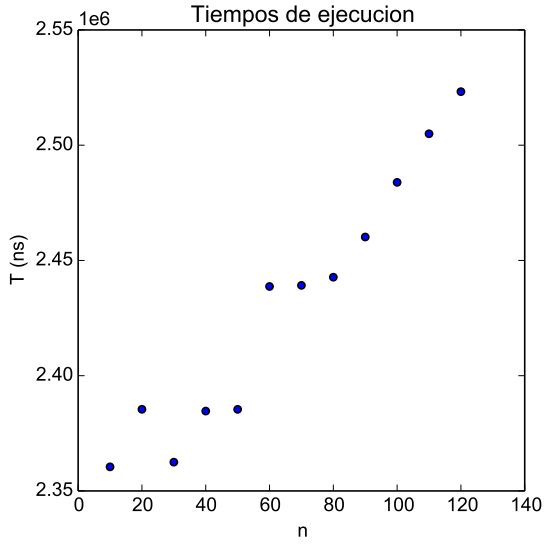
Los resultados temporales obtenidos fueron los siguientes:



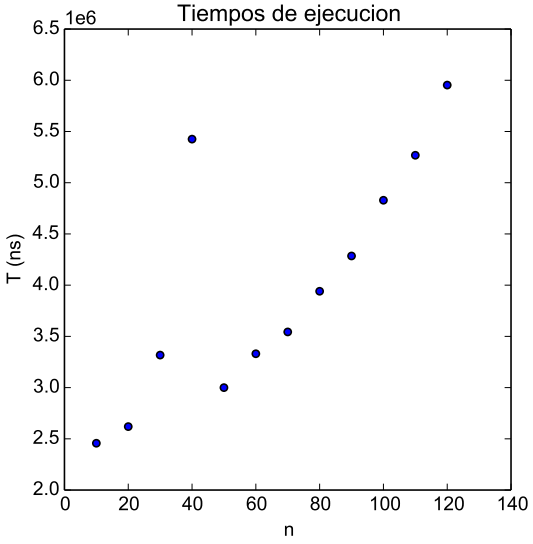
Grafos Aleatorios ($m = n$)



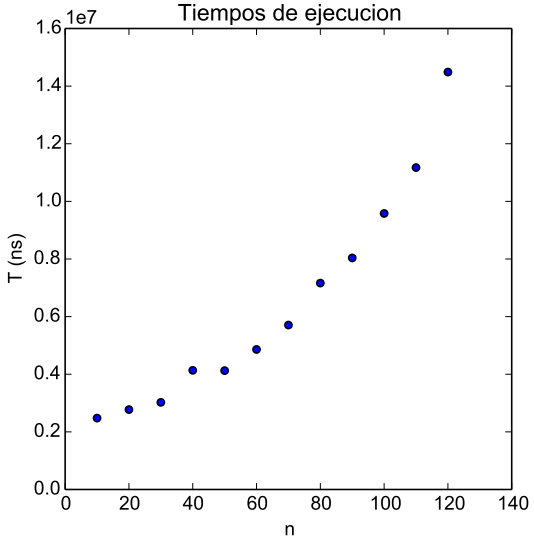
Grafos Aleatorios ($m = 2n$)



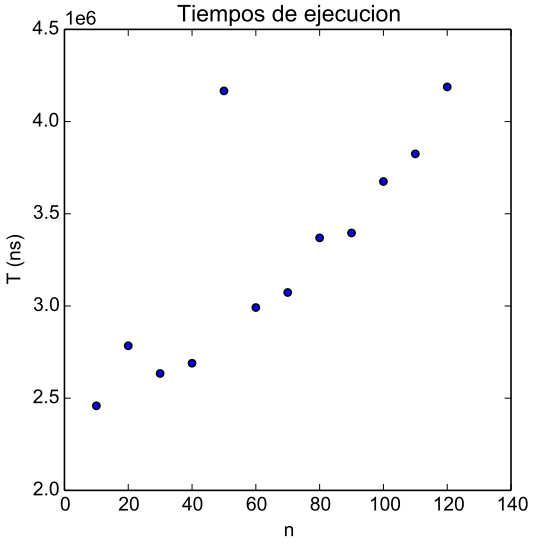
Grafos Aleatorios ($m = \frac{n}{2}$)



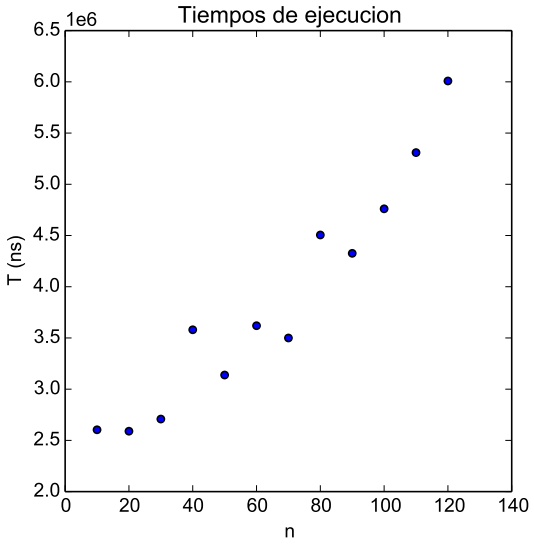
Grafos Bipartitos ($\frac{n}{4}$ nodos en la segunda componente)



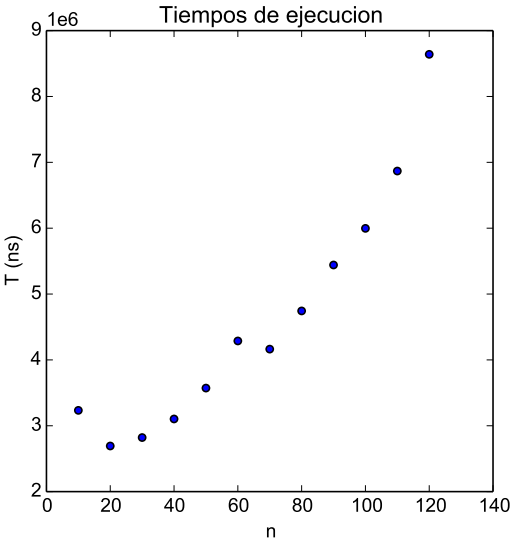
Grafos Bipartitos ($\frac{3n}{4}$ nodos en la segunda componente)



Grafos d-regulares ($d = \frac{n}{4}$)



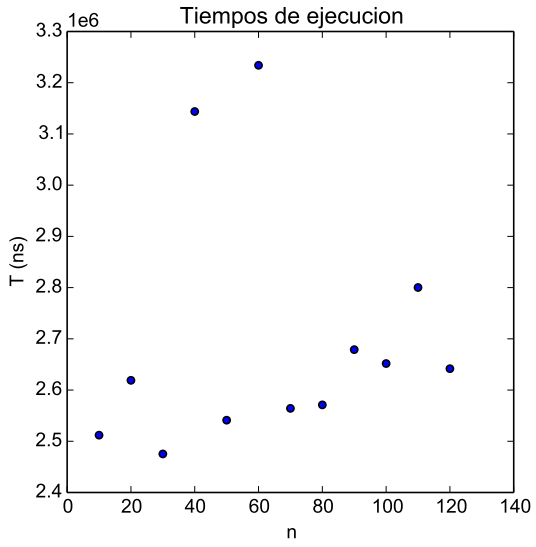
Grafos d-regulares ($m = \frac{n}{2}$)



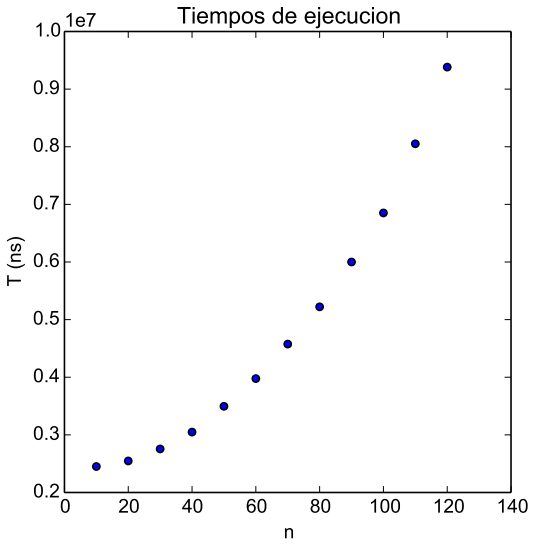
Grafos d-regulares ($m = \frac{3n}{4}$)

Primero vamos a ver los resultados por cada familia.

- Grafos Aleatorios: En este caso podemos ver que la cantidad de conexiones entre nodos afecto al tiempo, igualmente el impacto no fue tan grande como esperábamos, en el caso $n = 120$ la diferencia entre $m = \frac{n}{4}$ y $m = 2n$ fue, en promedio, de 218 segundos.
- Grafos Bipartitos: En este caso nos sorprendió el tiempo que tardo el algoritmo en poder encontrar solución, consideramos que esto se debe a que en un grafo bipartito completo existen solo dos posibles cubrimientos. Otro detalle a destacar, fue el aumento en tiempo que hubo mientras mas equilibradas se encontraban las dos componentes del grafo, con $\frac{3n}{4}$ nodos en la segunda componente se convergió a un resultado en un tiempo mucho mayor.
- Grafos d-regulares: Aquí a diferencia de los grafos aleatorios, al haber una diferencia mas marcada entre la cantidad de conexiones se puede ver en el gráfico que la diferencia entre $d = \frac{n}{4}$ y $d = \frac{3n}{4}$ es muy marcada, la misma siendo de varios minutos.
- Arboles binarios: En este caso lamentablemente no es posible hacer un análisis detallado, ya que los resultados no fueron regulares. Sin embargo, podemos destacar, que los resultados se obtuvieron en un tiempo razonable.



Arboles Binarios



Clique

Greedy Heap

Cuadro 1: Grafos aleatorios

	m = n/2	m = n	m = 2n
n = 40	26	21	12
n = 60	38	27	16
n = 80	49	33	21
n = 100	59	42	24
n = 120	74	55	28

- Cliques: Las cliques se comportaron de manera esperada, al ser un caso facil de resolver el algoritmo no tuvo mayores dificultades.

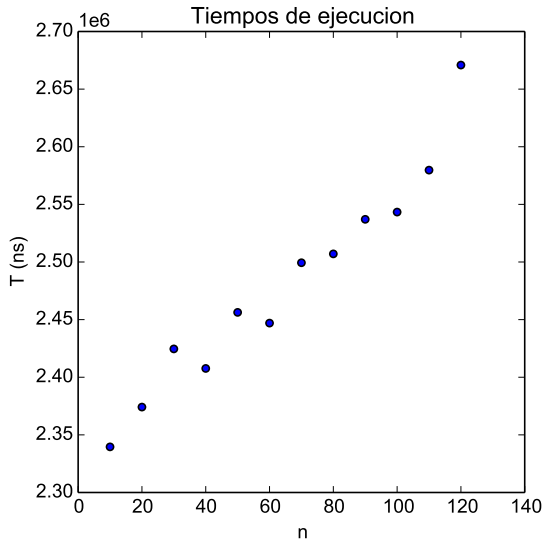
Para el análisis del tamaño de la solución, vamos a ver los resultados por cada familia. En el caso de los aleatorios, los resultados para estas configuraciones fueron los siguiente:

Los tamaños de resultados se comportaron de manera esperada, es decir, a medida que avanzo la cantidad de conexión se redujo el tamaño de solución.

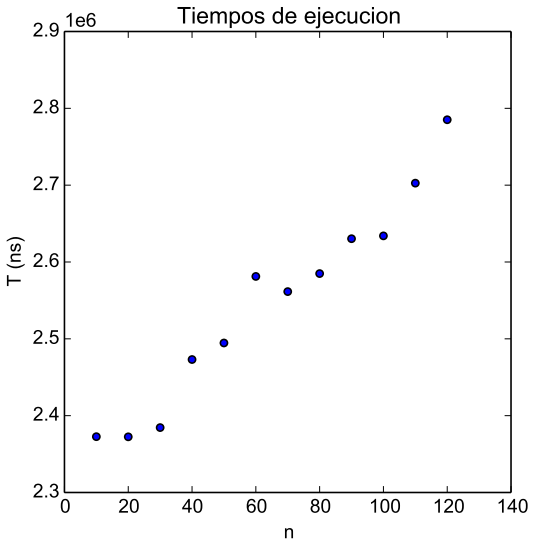
Para los Grafos Bipartitos, los d-regulares y las cliques, el algoritmo encontró la solución optima en todos los casos. Respecto a los arboles, la solución del algoritmo siempre respeto la cota y el resultado fue el menor posible.

4.4.2. Heurística Constructiva Golosa por Scoring

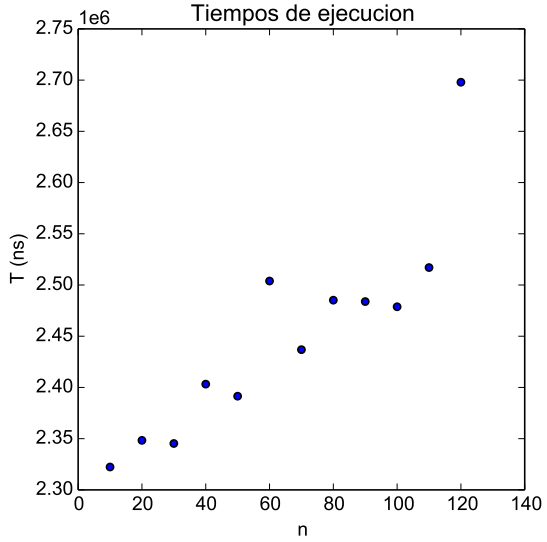
Los resultados temporales obtenidos fueron los siguientes:



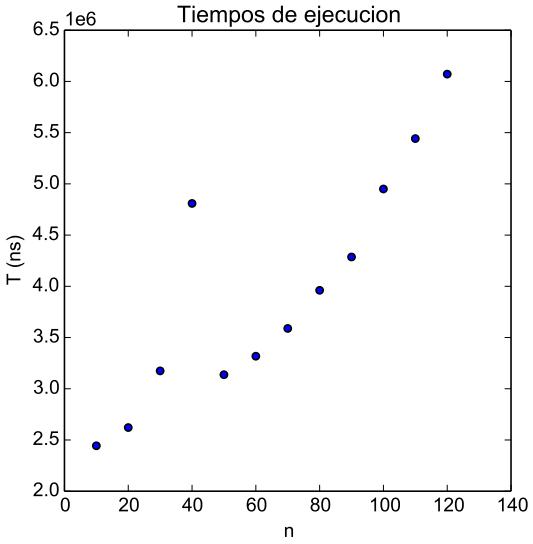
Grafos Aleatorios ($m = n$)



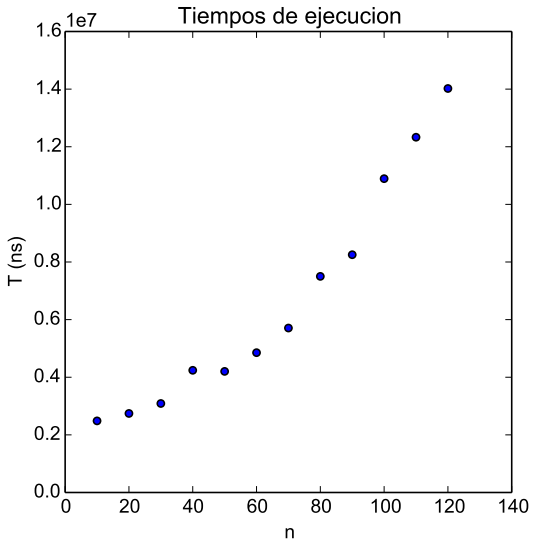
Grafos Aleatorios ($m = 2n$)



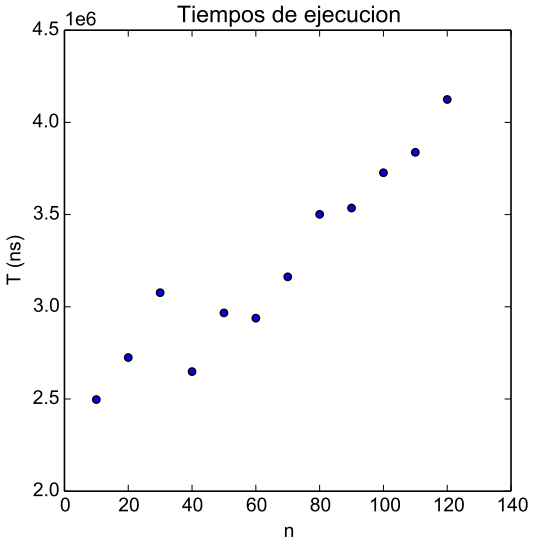
Grafos Aleatorios ($m = \frac{n}{2}$)



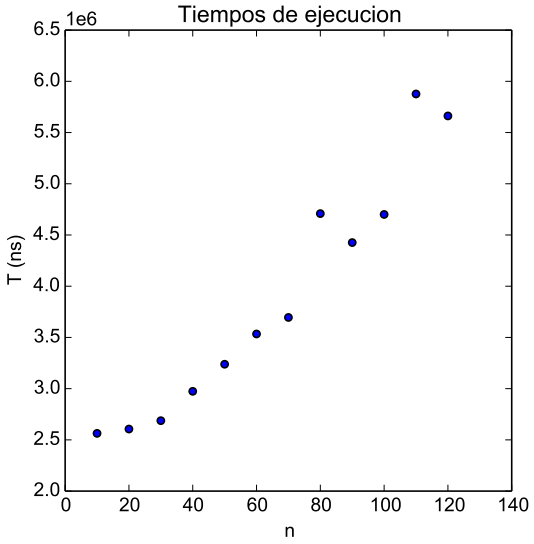
Grafos Bipartitos ($\frac{n}{4}$ nodos en la segunda componente)



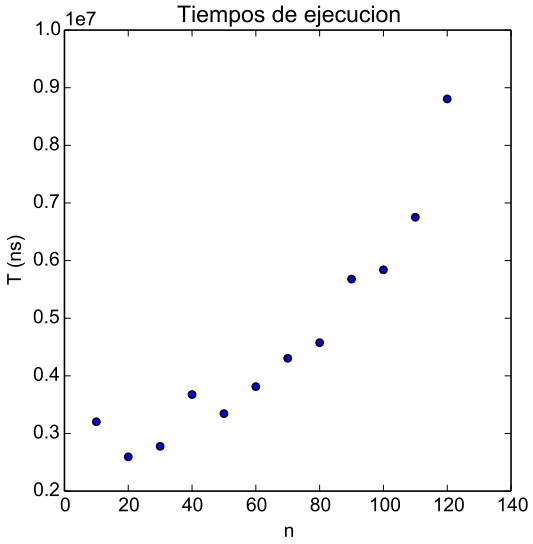
Grafos Bipartitos ($\frac{3n}{4}$ nodos en la segunda componente)



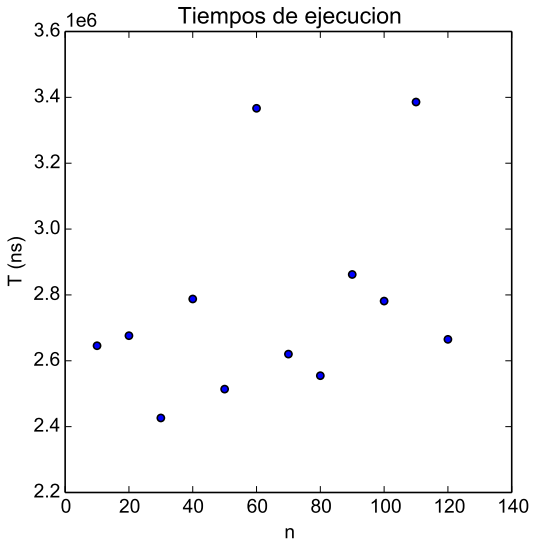
Grafos d-regulares ($d = \frac{n}{4}$)



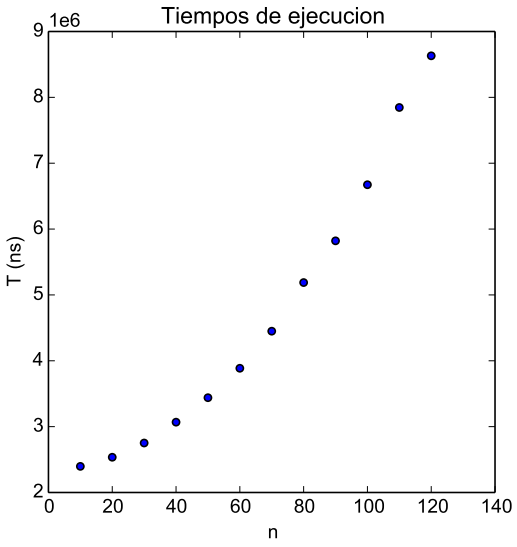
Grafos d-regulares ($m = \frac{n}{2}$)



Grafos d-regulares ($m = \frac{3n}{4}$)



Arboles Binarios



Clique

Greedy Heap

Cuadro 2: My caption

	m = n/2	m = n	m = 2n
n = 40	32	26	16
n = 60	43	33	16
n = 80	56	44	30
n = 100	67	56	40
n = 120	74	66	46

Primero vamos a ver los resultados por cada familia.

Los resultados obtenidos por familia no difirieron en gran medida respecto a lo obtenido con la Heurística Constructiva Golosa por Grado, con lo cual respecto al tiempo se derivan las misma conclusiones de antes.

Para el análisis del tamaño de la solución, vamos a ver los resultados por cada familia. En el caso de los aleatorios, los resultados para estas configuraciones fueron los siguiente:

Aquí es donde la diferencia es mas marcada, para los mismos casos, la Heurística por Scoring dio resultados significativamente peores en el caso aleatorio. Esto también se vio reflejado en las otras familias también, particularmente en el caso de los bipartitos donde siempre se priorizo la solución mas grande.

4.4.3. Conclusión

En lo que respecta al tiempo de ejecución, las heurísticas no se comportaron de manera muy diferente, el tiempo fue similar. Sin embargo, el lugar donde se noto la diferencia fue en el tamaño de las soluciones obtenidas, en prácticamente todos los casos, la Heurística Constructiva por Grado dio mejor resultado, con lo cual consideramos que de las golosas, es mejor la selección por grado que por scoring.

5. Heurística de Búsqueda Local

5.1. Algoritmo

Antes de explicar nuestro algoritmo, comenzemos definiendo que es una heurística de búsqueda local. Para cada solución factible $s \in S$, se define $N(s)$ como el conjunto de soluciones vecinas de s. Un procedimiento de búsqueda local toma una solución inicial s e iterativamente la mejora reemplazándola por otra solución mejor del conjunto $N(s)$, hasta llegar a un óptimo local. El algoritmo se puede ver con el siguiente pseudocódigo:

```
procedure LOCALSEARCH(G)
    s ← getInitialSolution(G)
    localSolution ← true
    while localSolution do
        localSolution ← false
        for all ŝ ∈ N(s) do
            if |ŝ| < |s| then
                s ← ŝ
                localSolution ← true
                break
```

En primer lugar hay que pensar que algoritmo utilizar en la función $getInitialSolution(G)$. Para esto, utilizamos cualquiera de las heurísticas constructivas golosas del paso anterior.

Luego, debemos identificar como construiremos las diferentes $s \in N(s)$, es decir, como construiremos la función que nos devuelve los vecinos de una solución parcial $N(S)$.

5.2. Vecindades

Al aplicar este algoritmo a un grafo $G(V, E)$, utilizaremos los siguientes dos criterios para definir la vecindad de una solución s:

1. Primera vecindad: La primera vecindad $N(s)$ esta dada por el conjunto de nodos tal que para algúna solución $n \in N(s)$, $v \in G.V \land v \notin s$, $v \in n$, $adj(v) \not\subset n$, el resto de los nodos en s están en n y n es un conjunto independiente dominante.

2. Segunda vecindad: La segunda vecindad $N(s)$ esta dada por el conjunto de nodos tal que para algúna solución $n \in N(s)$ y un par de vértices $u, v \in G.V \land u, v \notin s$, $u, v \in n$, $adj(u) \cup adj(v) \not\subset n$, el resto de los nodos en s están en n y n es un conjunto independiente dominante.

5.3. Complejidad

5.3.1. Primera vecindad

Pseudocódigo

```
procedure N(G,s)
    removedNodes ← ∅
    for all v ∈ G.V do
        if v.added == true ∨ v.degree == 1 then
            continue
        v.added ← true
        for all adjNode ∈ v.adj do
            if adjNode.added == false then
                continue
            removed.push(adjNode)
            adjNode.added ← false
            for all adjToAdj ∈ adjNode.adj do
                if !isReachable(G, adjToAdj) then
                    while !removedNodes.isEmpty() do
                        n ← removedNodes.pop()
                        n.added ← true
                        v.added ← false
                        tryNextNode()
```

return

Lo que hace este procedimiento es buscar un vecino válido. De no encontrarlo, restaura el grafo y prueba el próximo nodo para buscar otro vecino con la función $tryNextNode()$, que es una especie de jump. La función $isReachable(G, node)$ simplemente se fija si dado un nodo existe algún vecino que este en el cubrimiento. Caso contrario, no estamos ante un conjunto válido. Esto lo hace en $\mathcal{O}(\Delta(G))$.

Complejidad

En una iteración, el primer algoritmo de vecindad agrega un nodo y luego quita sus adyacentes en $\mathcal{O}(\Delta(G))$. Luego verifica que los adyacentes de estos vértices que hemos quitado son alcanzables. Por lo tanto, en el peor caso una iteración tiene orden $\mathcal{O}(n \times \Delta(G)^3)$. Esto se debe a que se debe verificar que todos los nodos adyacentes a los que saque son adyacentes a algún otro nodo del conjunto en $\mathcal{O}(\Delta(G))$ para cada nodo adyacente ($\Delta(G)$) a los adyacentes que pude quitar ($\Delta(G)$). Si el nuevo conjunto de nodos no es un CIDM, simplemente restauramos el grafo en $\mathcal{O}(\Delta(G))$.

5.3.2. Segunda vecindad

Pseudocódigo

```
procedure N(G,s)
    removedNodes ← ∅
    for all (u,v) ∈ G.V do
        if u.added == true ∨ u.degree == 1 ∨ v.added == true ∨ v.degree == 1 then
            continue
        u.added ← true
        v.added ← true
        for all adjNode ∈ v.adj do
            if adjNode.added == false then
                continue
            removed.push(adjNode)
            adjNode.added ← false
            for all adjToAdj ∈ adjNode.adj do
                if !isReachable(G, adjToAdj) then
                    while !removedNodes.isEmpty() do
                        n ← removedNodes.pop()
                        n.added ← true
                        u.added ← false
                        v.added ← false
                        tryNextPair()
        for all adjNode ∈ u.adj do
            if adjNode.added == false then
                continue
            removed.push(adjNode)
            adjNode.added ← false
            for all adjToAdj ∈ adjNode.adj do
                if !isReachable(G, adjToAdj) then
                    while !removedNodes.isEmpty() do
                        n ← removedNodes.pop()
                        u.added ← false
                        v.added ← false
                        tryNextPair()
        return
```

Complejidad

En el segundo caso, probamos agregando todos los pares de nodos a la solución actual, quitando sus nodos adyacentes y verificando si luego es una solución. Para ello, simplemente repetimos el procedimiento de la primera vecindad.

Este procedimiento lo repetimos para todo par de $v \notin S$. Podemos acotar esto por $\binom{n}{2}$. Por lo tanto la complejidad total de una iteración es de $\mathcal{O}(\binom{n}{2} \times \Delta(G)^3)$.

6. Metaheurística GRASP

6.1. Algoritmo

GRASP (Greedy Randomized Adaptative Search Procedure) es una combinación entre una heurística golosa aleatorizada y un procedimiento de búsqueda local. La metaheurística se puede ver con el siguiente pseudocódigo:

```
procedure GRASP(G, k)
    G bestSolution
    while !terminationCondition(G, k, bestSolution) do
        s ← randomGreedyHeuristic(G, k)
        s ← localSearch(G,s)
        if |s| < |bestSolution| then
            bestSolution ← s
```

De este procedimiento surgen dos preguntas, que en realidad son cosas que debemos definir. De donde proviene la aleatoriedad de la heurística greedy? Cual es criterio de terminación que utilizaremos?

6.2. Random Greedy Heuristic

6.2.1. Por cantidad

Para agregarle una componente aleatoria a `GRASP`, se propone fabricar en cada paso de la heurística constructiva golosa una *Lista Restricta de Candidatos* (RCL) y elegir aleatoriamente un candidato de esta lista. Para ello, decidimos crear la función `greedyHeapConstructiveRandomized(Node graph[], int n, int k)` que lo que hace es ir eligiendo aleatoriamente de los k vértices con mayor grado utilizando un heap como estructura auxiliar. Esto se puede ver en el siguiente pseudocódigo:

```
procedure greedyHeapConstructiveRandomized(G,k)
    nodeHeap ← buildHeap(G.V)
    while !nodeHeap.isEmpty() do
        node ← nodeHeap.topRandomPop(k)
        if node.reachable == true then
            continue
        node.added = true
        for all adj ∈ node.adj do
            adj.reachable ← true
```

6.2.2. Por valor

Al igual que en el criterio anterior, elegimos un candidato aleatorio de una lista desde un heap. Sin embargo, ahora un vértice está en la lista de candidatos si y sólo si el grado de cualquier nodo en la lista esta a una distancia menor o igual a k grados del vértice de mayor grado disponible en el heap. Esto se puede ver en el siguiente pseudocódigo:

```
procedure greedyHeapConstructiveRandomized(G,k)
    nodeHeap ← buildHeap(G.V)
    while !nodeHeap.isEmpty() do
        node ← nodeHeap.topRandomPopByValue(k)
        if node.reachable == true then
            continue
        node.added = true
        for all adj ∈ node.adj do
            adj.reachable ← true
```

6.3. Criterios de terminación

1. No se encontró ninguna mejora en las ultimas j iteraciones.
2. Se alcanzo un límite prefijado de j iteraciones.

7. Codigo

7.1. containers.h

```
#ifndef DATA_STRUCTURES_H_
#define DATA_STRUCTURES_H_

#include <forward_list>

using namespace std;

struct Node {
    int degree;
    int score;
    bool added;
    bool reachable;
    forward_list<int> adj;

    Node() {
        degree = 0;
        score = 0;
        added = false;
        reachable = false;
    }
};

struct _Pair {
    int degree;
    int id;

    _Pair(int _degree, int _id) {
        degree = _degree;
        id = _id;
    }

    bool operator <(const _Pair& x) {
        return this->degree < x.degree;
    }
};

#endif
```

7.2. backtracking.cpp

```
1  #include <iostream>
2  #include <forward_list>
3  #include "../containers.h"
4
5  using namespace std;
6
7  void backtracking(int current, int& n, int coveredNodes, int usedNodes, Node graph[],
       bool localSolution[], int& nodesUsedInSolution);
8
9  int main() {
10
11     int n, m; // n: vertices, m: edges
12     cin >> n >> m;
13
14     Node graph[n]; // graph container
15     bool localSolution[n];
16
17     int u, v;
18     for (int i = 1; i <= m; ++i) { // (u,v) edges
19         cin >> u >> v;
20         u--; // nodes are counted from 0 in array.
21         v--;
22         graph[u].adj.push_front(v);
23         graph[v].adj.push_front(u);
24
25         graph[u].degree++;
26         graph[v].degree++;
27     }
28
29     int initialNodes = 0;
30     for (int i = 0; i < n; ++i) { // add d(v)=0 nodes to cover.
31         if (graph[i].degree == 0) {
32             graph[i].added = true;
33             graph[i].reachable = true;
34             localSolution[i] = true;
35             initialNodes++;
36         }
37     }
38
39     int nodesUsedInSolution = n; // worst case scenario is n, that way I avoid
           setting all the array as true.
40
41     backtracking(0, n, initialNodes, initialNodes, graph, localSolution,
           nodesUsedInSolution);
42
43     // display solution
44     cout << nodesUsedInSolution;
45     for (int i = 0; i < n; ++i) {
46         if (localSolution[i] == true) cout << " " << i + 1;
47     }
48     cout << endl;
49
50     return 0;
51 }
52
53 void backtracking(int current, int& n, int coveredNodes, int usedNodes, Node graph[],
       bool localSolution[], int& nodesUsedInSolution) {
```

```
54
55      if (current == n) return; // no nodes left to add.
56      if (graph[current].reachable == true) return backtracking(current + 1, n,
            coveredNodes, usedNodes, graph, localSolution, nodesUsedInSolution);
57      if (usedNodes + 1 == nodesUsedInSolution) return; // cant beat current solution
58
59      int pushed = 0;
60      forward_list<int> added; // save changes to graph to then restore
61      graph[current].added = true;
62
63      for (auto it = graph[current].adj.begin(); it != graph[current].adj.end(); ++it)
            {
64          int adjNode = *it;
65          if (graph[adjNode].reachable == false) { // node reaches these new vertices
66              graph[adjNode].reachable = true;
67              added.push_front(adjNode);
68              ++pushed;
69          }
70      }
71
72      int tempCoveredNodes = coveredNodes + pushed + 1;
73      if (tempCoveredNodes == n) { // coverage found
74          for (int i = 0; i < n; ++i) {
75              localSolution[i] = graph[i].added;
76          }
77          nodesUsedInSolution = ++usedNodes;
78      } else {
79          backtracking(current + 1, n, tempCoveredNodes, usedNodes + 1, graph,
                localSolution, nodesUsedInSolution); // adding current element to coverage
80      }
81
82      // restore graph state
83      graph[current].added = false;
84      for (auto it = added.begin(); it != added.end(); ++it) {
85          graph[*it].reachable = false;
86      }
87
88      backtracking(current + 1, n, coveredNodes, usedNodes, graph, localSolution,
            nodesUsedInSolution); // skip current node
89
90  }
```

7.3. greedy.cpp

```
1  #include <iostream>
2  #include <algorithm>
3  #include <stdlib.h>
4  #include "../containers.h"
5
6  using namespace std;
7
8  #define E_INVALID_PARAMETER 0
9
10 /* Gready Constructive Randomized Heuristic for MIDS
11  * Using a heap, this function builds a MIDS by picking vertices
12  * randomly from the top k vertices with the highest degree.
13  *
14  * @param graph[] Array of nodes.
15  * @param n Size of graph.
16  * @param k Parameter that indicates from how many nodes to
17  *          pick randomly
18  * @return Nodes used in solution set.
19  */
20 int greedyHeapConstructiveRandomized(Node graph[], int n, int k) {
21
22     if (k == 0) return E_INVALID_PARAMETER;
23
24     vector<_Pair> currentPicks;
25     vector<_Pair> heap;
26     int nodesUsed = 0;
27
28     for (int i = 0; i < n; i++) {
29         if (graph[i].degree == 0) {
30             graph[i].added = true;
31             graph[i].reachable = true;
32             nodesUsed++;
33         } else {
34             graph[i].added = false;
35             graph[i].reachable = false;
36             heap.push_back(_Pair(graph[i].degree, i));
37         }
38     }
39     make_heap(heap.begin(), heap.end());
40
41     int i = 0;
42     while (i < k && i < (int) heap.size()) {
43         _Pair p = heap.front();
44         currentPicks.push_back(p);
45         pop_heap(heap.begin(), heap.end());
46         heap.pop_back();
47         i++;
48     }
49
50     while (currentPicks.size() > 0) {
51         int id = rand() % currentPicks.size();
52         _Pair p = currentPicks.at(id);
53         currentPicks.erase(currentPicks.begin() + id);
54
55         if (heap.size() > 0) {
56             _Pair p2 = heap.front();
57             currentPicks.push_back(p2);
```

```
58              pop_heap(heap.begin(), heap.end());
59              heap.pop_back();
60          }
61
62          if (graph[p.id].reachable == true) continue;
63
64          graph[p.id].added = true;
65          nodesUsed++;
66
67          for (auto it = graph[p.id].adj.begin(); it != graph[p.id].adj.end(); ++it) {
68              int adjNode = *it;
69              graph[adjNode].reachable = true;
70          }
71
72      }
73
74      return nodesUsed;
75  }
76
77  /* Gready Constructive Randomized Heuristic for MIDS
78   * Using a heap, this function builds a MIDS by picking vertices
79   * randomly from the vertices that are k degrees away from the
80   * available vertex with the highest degree.
81   *
82   * @param graph[] Array of nodes.
83   * @param n Size of graph.
84   * @param k Parameter that indicates from how many nodes to
85   *          pick randomly
86   * @return Nodes used in solution set.
87   */
88  int greedyHeapConstructiveRandomized2(Node graph[], int n, int k) {
89
90      if (k < 0) return E_INVALID_PARAMETER;
91
92      vector<_Pair> currentPicks;
93      vector<_Pair> heap;
94      int nodesUsed = 0;
95
96      for (int i = 0; i < n; i++) {
97          if (graph[i].degree == 0) {
98              graph[i].added = true;
99              graph[i].reachable = true;
100             nodesUsed++;
101         } else {
102             graph[i].added = false;
103             graph[i].reachable = false;
104             heap.push_back(_Pair(graph[i].degree, i));
105         }
106     }
107     make_heap(heap.begin(), heap.end());
108
109     int i = 0;
110     int degree = heap.front().degree;
111     while(i < (int) heap.size() && heap.front().degree >= degree - k) {
112         _Pair p = heap.front();
113         currentPicks.push_back(p);
114         pop_heap(heap.begin(), heap.end());
115         heap.pop_back();
116         i++;
```

```
117     }
118
119     while (currentPicks.size() > 0) {
120         int id = rand() %currentPicks.size();
121
122         _Pair p = currentPicks.at(id);
123
124         currentPicks.erase(currentPicks.begin() + id);
125
126         if (currentPicks.size() > 0) {
127             degree = currentPicks.at(0).degree;
128         } else {
129             degree = 0;
130         }
131
132         if (heap.size() > 0 && heap.front().degree >= degree - k) {
133             _Pair p2 = heap.front();
134             currentPicks.push_back(p2);
135             pop_heap(heap.begin(), heap.end());
136             heap.pop_back();
137         }
138
139         if (graph[p.id].reachable == true) continue;
140
141         graph[p.id].added = true;
142         nodesUsed++;
143
144         for (auto it = graph[p.id].adj.begin(); it != graph[p.id].adj.end(); ++it) {
145             int adjNode = *it;
146             graph[adjNode].reachable = true;
147         }
148
149     }
150     return nodesUsed;
151 }
152
153 /* Greedy Constructive Heuristic for MIDS
154  * Using a heap, this function builds a MIDS by picking
155  * the highest degree vertex repeatedly.
156  *
157  * @param graph[] Array of nodes.
158  * @param n Size of graph.
159  * @return Nodes used in solution set.
160  */
161 int greedyHeapConstructive(Node graph[], int n) {
162
163     vector<_Pair> heap;
164     int nodesUsed = 0;
165
166     for (int i = 0; i < n; i++) {
167         if (graph[i].degree == 0) {
168             graph[i].added = true;
169             graph[i].reachable = true;
170             nodesUsed++;
171         } else {
172             heap.push_back(_Pair(graph[i].score, i));
173             graph[i].added = false;
174             graph[i].reachable = false;
175         }
```

```
176     }
177     make_heap(heap.begin(), heap.end());
178 
179     for (int i = 0; i < n; i++) {
180         _Pair p = heap.front();
181         pop_heap(heap.begin(), heap.end());
182         heap.pop_back();
183 
184         if (graph[p.id].reachable == true) continue;
185 
186         graph[p.id].added = true;
187         nodesUsed++;
188 
189         for (auto it = graph[p.id].adj.begin(); it != graph[p.id].adj.end(); ++it) {
190             int adjNode = *it;
191             graph[adjNode].reachable = true;
192         }
193     }
194 
195     return nodesUsed;
196 }
197 
198 /* Greedy Constructive Heuristic for MIDS
199  * This function builds a MIDS by picking vertices by score.
200  * The score is defined as the number of effective reachable
201  * vertices given the vertices that have already been picked.
202  *
203  * @param graph[] Array of nodes.
204  * @param n Size of graph.
205  * @return Nodes used in solution set.
206  */
207 int greedyConstructive(Node graph[], int n) {
208 
209     int nodesUsed = 0;
210 
211     for (int i = 0; i < n; i++) {
212         if (graph[i].degree == 0) {
213             graph[i].added = true;
214             graph[i].reachable = true;
215             nodesUsed++;
216         } else {
217             graph[i].added = false;
218             graph[i].reachable = false;
219             graph[i].score = graph[i].degree;
220         }
221     }
222 
223     for (int i = 0; i < n; ++i) {
224 
225         int greatest = 0;
226         int score = 0;
227         bool flag = false;
228 
229         // search for max score.
230         for (int j = 0; j < n; ++j) {
231             if (graph[j].reachable == true) continue;
232             if (graph[j].score >= score) { // can be improved here!
233                 greatest = j;
234                 score    = graph[j].score;
```

```
235                     flag = true;
236                 }
237             }
238 
239             if (!flag) break; // no more nodes to search.
240 
241             graph[greatest].added = true;
242             graph[greatest].reachable = true;
243 
244             // update adyacent nodes of reachable nodes' scores.
245             for (auto it = graph[greatest].adj.begin(); it != graph[greatest].adj.end();
                    ++it) {
246                 int adjNode = *it;
247                 graph[adjNode].reachable = true;
248                 for (auto it2 = graph[adjNode].adj.begin(); it2 != graph[adjNode].adj.end
                        (); ++it2) {
249                     graph[*it2].score--;
250                 }
251             }
252 
253             nodesUsed++;
254         }
255 
256         return nodesUsed;
257     }
```

7.4. local.cpp

```
1  #include <iostream>
2  #include <algorithm>
3  #include <stdlib.h>
4  #include <list>
5  #include "local.h"
6
7  using namespace std;
8
9  bool isReachable(Node graph[], int u);
10
11 /* Local search by adding 1 node
12  * @param graph[] Array of nodes.
13  * @param n Size of graph.
14  * @param nodesUsedInSolution Size of current solution
15  */
16 int localSearch(Node graph[], int n, int nodesUsedInSolution) {
17
18     int currentNodes = nodesUsedInSolution;
19     list<int> removed;
20
21     for (int i = 0; i < n; ++i) {
22         if (graph[i].added == true || graph[i].degree == 1) continue; // search for a
               node not in S.
23
24         graph[i].added = true;
25         currentNodes++;
26
27         bool reachable = true;
28
29         for (auto it = graph[i].adj.begin(); it != graph[i].adj.end(); ++it) { //
              iterate adj
30
31             if (graph[*it].added == false || *it == i) continue; // doesnt affect adj
                   nodes.
32
33             removed.push_front(*it);
34             graph[*it].added = false;
35             currentNodes--;
36
37             for (auto it2 = graph[*it].adj.begin(); it2 != graph[*it].adj.end(); ++
                  it2) { // iterate adj to adj
38                 if (!isReachable(graph, *it2)) {
39                     reachable = false;
40                     goto stop;
41                 }
42             }
43
44         }
45
46         stop:
47
48         if (reachable == true && currentNodes < nodesUsedInSolution) { // build graph
               once we know we can improve it.
49             removed.clear();
50             nodesUsedInSolution = currentNodes;
51             i = 0; // s <- s'
52         } else {
```

```
53              graph[i].added = false;
54              while (removed.size() > 0) { // restore graph
55                  graph[removed.front()].added = true;
56                  removed.pop_front();
57              }
58              currentNodes = nodesUsedInSolution;
59          }
60      }
61
62      return nodesUsedInSolution;
63  }
64
65  /* Local search by adding 2 nodes
66   * @param graph[] Array of nodes.
67   * @param n Size of graph.
68   * @param nodesUsedInSolution Size of current solution
69   */
70  int localSearch2(Node graph[], int n, int nodesUsedInSolution) {
71
72      int currentNodes = nodesUsedInSolution;
73      list<int> removed;
74
75      for (int i = 0; i < n; ++i) {
76
77          // find index of two nodes not in S.
78          if (graph[i].added == true || graph[i].degree == 1) continue;
79
80          int j;
81          for (j = i + 1; j < n; j++) { // search for a second node
82              if (graph[j].added == true || graph[j].degree == 1) continue;
83
84              if (j == n) break; // no pair found
85
86              graph[i].added = true;
87              graph[j].added = true;
88              currentNodes = currentNodes + 2;
89
90              bool reachable = true;
91
92              // analyze node i
93              for (auto it = graph[i].adj.begin(); it != graph[i].adj.end(); ++it) { //
                    iterate adj
94
95                  if (graph[*it].added == false || *it == i || *it == j) continue; //
                        doesnt affect adj nodes.
96
97                  removed.push_front(*it);
98                  graph[*it].added = false;
99                  currentNodes--;
100
101                 for (auto it2 = graph[*it].adj.begin(); it2 != graph[*it].adj.end();
                        ++it2) { // iterate adj to adj
102                     if (!isReachable(graph, *it2)) {
103                         reachable = false;
104                         goto stop;
105                     }
106                 }
107
108             }
```

```
109
110             // analyze node j
111             for (auto it = graph[j].adj.begin(); it != graph[j].adj.end(); ++it) { //
                    iterate adj
112
113                 if (graph[*it].added == false || *it == i || *it == j) continue; //
                        doesnt affect adj nodes.
114
115                 removed.push_front(*it);
116                 graph[*it].added = false;
117                 currentNodes--;
118
119                 for (auto it2 = graph[*it].adj.begin(); it2 != graph[*it].adj.end();
                        ++it2) { // iterate adj to adj
120                     if (!isReachable(graph, *it2)) {
121                         reachable = false;
122                         goto stop;
123                     }
124                 }
125             }
126
127             stop:
128
129             if (reachable == true && currentNodes < nodesUsedInSolution) { // build
                    graph once we know we can improve it.
130                 removed.clear();
131                 nodesUsedInSolution = currentNodes;
132                 i = 0; // s <- s'
133             } else {
134                 graph[i].added = false;
135                 graph[j].added = false;
136                 while (removed.size() > 0) { // restore graph
137                     graph[removed.front()].added = true;
138                     removed.pop_front();
139                 }
140                 currentNodes = nodesUsedInSolution;
141             }
142         }
143     }
144
145     return nodesUsedInSolution;
146 }
147
148 /* Checks if a node is reachable by other nodes in the set.
149  * @param graph[] Array of nodes.
150  * @param u Node id.
151  * @return Returns if u is reachable by the nodes in the set.
152  */
153 bool isReachable(Node graph[], int u) {
154     for (auto it = graph[u].adj.begin(); it != graph[u].adj.end(); ++it) {
155         if (graph[*it].added) {
156             return true;
157         }
158     }
159     return false;
160 }
```

7.5. grasp.cpp

```
#include <iostream>
#include <forward_list>
#include <algorithm>
#include <stdlib.h>
#include "../greedy/greedy.h"
#include "../local/local.h"

using namespace std;

void displaySolution(Node graph[], int n, int nodesUsedInSolution);
int graspMIDSByIterations(Node graph[], int n, int j, int k, bool localSolution[]);
int graspMIDSByValue(Node graph[], int n, int j, int k, bool localSolution[]);

int main() {

    int n, m; // n: vertices, m: edges
    cin >> n >> m;

    Node graph[n]; // graph container
    bool localSolution[n];

    int u, v;
    for (int i = 1; i <= m; ++i) { // (u,v) edges
        cin >> u >> v;
        u--; // nodes are counted from 0 in array.
        v--;
        graph[u].adj.push_front(v);
        graph[v].adj.push_front(u);

        graph[u].degree++;
        graph[v].degree++;
    }

    int nodesUsedInSolution = graspMIDSByIterations(graph, n, 7, 7, localSolution);
    // int nodesUsedInSolution = graspMIDSByValue(graph, n, 3, 3, localSolution);

    // display solution
    cout << nodesUsedInSolution;
    for (int i = 0; i < n; ++i) {
        if (localSolution[i] == true) cout << " " << i + 1;
    }
    cout << endl;

    return 0;
}


void displaySolution(Node graph[], int n, int nodesUsedInSolution) {
    cout << nodesUsedInSolution;
    for (int i = 0; i < n; ++i) {
        if (graph[i].added == true) cout << " " << i + 1;
    }
    cout << endl;
}

/* GRASP Heuristic
 * Minimum Independent Dominating Set
```

```
58   * Stop criteria: Iterations
59   * @param j Amount of attempts to improve solution.
60   * @param k Parameter used for greedy heuristic.
61   * @return Nodes used in solution set.
62   */
63  int graspMIDSByIterations(Node graph[], int n, int j, int k, bool localSolution[]) {
64      int currentBest = n + 1;
65      while (j > 0) {
66          int nodesUsed = greedyHeapConstructiveRandomized(graph, n, k);
67          //int nodesUsed = greedyHeapConstructiveRandomized2(graph, n, k);
68
69          // nodesUsed = localSearch(graph, n, nodesUsed);
70
71          if (nodesUsed < currentBest) { // save local solution
72              for (int i = 0; i < n; ++i) {
73                  localSolution[i] = graph[i].added;
74              }
75              currentBest = nodesUsed;
76          }
77
78          j--;
79      }
80      return currentBest;
81  }
82
83  /* GRASP Heuristic
84   * Minimum Independent Dominating Set
85   * Stop criteria: Cycles without improvements
86   * @param j Limit to cycles without improvements.
87   * @param k Parameter used for greedy heuristic.
88   * @return Nodes used in solution set.
89   */
90  int graspMIDSByValue(Node graph[], int n, int j, int k, bool localSolution[]) {
91      int currentBest = n + 1;
92      int cycles = 0;
93      while (cycles < j) {
94          int nodesUsed = greedyHeapConstructiveRandomized(graph, n, k);
95          //int nodesUsed = greedyHeapConstructiveRandomized2(graph, n, k);
96
97          nodesUsed = localSearch(graph, n, nodesUsed);
98          // nodesUsed = localSearch2(graph, n, nodesUsed);
99
100         if (nodesUsed < currentBest) { // save local solution
101             for (int i = 0; i < n; ++i) {
102                 localSolution[i] = graph[i].added;
103             }
104             currentBest = nodesUsed;
105             cycles = 0;
106         } else {
107             cycles++;
108         }
109     }
110     return currentBest;
111 }
```