



DEPARTAMENTO  
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

## Trabajo Práctico III

---

Algoritmos y Estructuras de Datos III  
Primer Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Aldasoro Agustina	86/13	agusaldasoro@gmail.com
Noriega Francisco	660/12	frannoriega.92@gmail.com
Zimenspitz Ezequiel	155/13	ezeqzim@gmail.com
Zuker Brian	441/13	brianzuker@gmail.com



Facultad de Ciencias Exactas y Naturales  
Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

<http://www.fcen.uba.ar>

Resumen va aquí

## Índice

<b>1. Introducción al problema</b>	<b>3</b>
1.1. Conjunto Independiente Dominante Mínimo (CIDM)	3
1.2. Paralelismo con “El señor de los caballos”	3
1.3. Todo conjunto independiente maximal es un conjunto dominante	5
1.4. Situaciones de la vida real	5
<b>2. Algoritmo Exacto</b>	<b>6</b>
2.1. Explicación y mejoras	6
2.2. Complejidad Temporal	7
2.3. Experimentación	7
2.3.1. Mejor caso	7
2.3.2. Nodos fijos	8
<b>3. Heurística Constructiva Golosa</b>	<b>10</b>
3.1. Explicación	10
3.2. Complejidad Temporal	10
3.3. Comparación de resultados con solución óptima	10
3.4. Experimentación	10
<b>4. Heurística de Búsqueda Local</b>	<b>11</b>
4.1. Explicación	11
4.1.1. Elección de Solución Inicial	11
4.1.2. Elección de Solución Inicial	11
4.2. Complejidad Temporal	11
4.3. Experimentación	11
<b>5. Metaheurística GRASP</b>	<b>12</b>
5.1. Explicación	12
5.2. Experimentación	13
<b>6. Comparación entre todos los métodos</b>	<b>16</b>

## 1. Introducción al problema

### 1.1. Conjunto Independiente Dominante Mínimo (CIDM)

Sea  $G = (V, E)$  un grafo simple. Un conjunto  $D \subseteq V$  es un *conjunto dominante* de  $G$  si todo vértice de  $G$  está en  $D$  o bien tiene al menos un vecino que está en  $D$ .

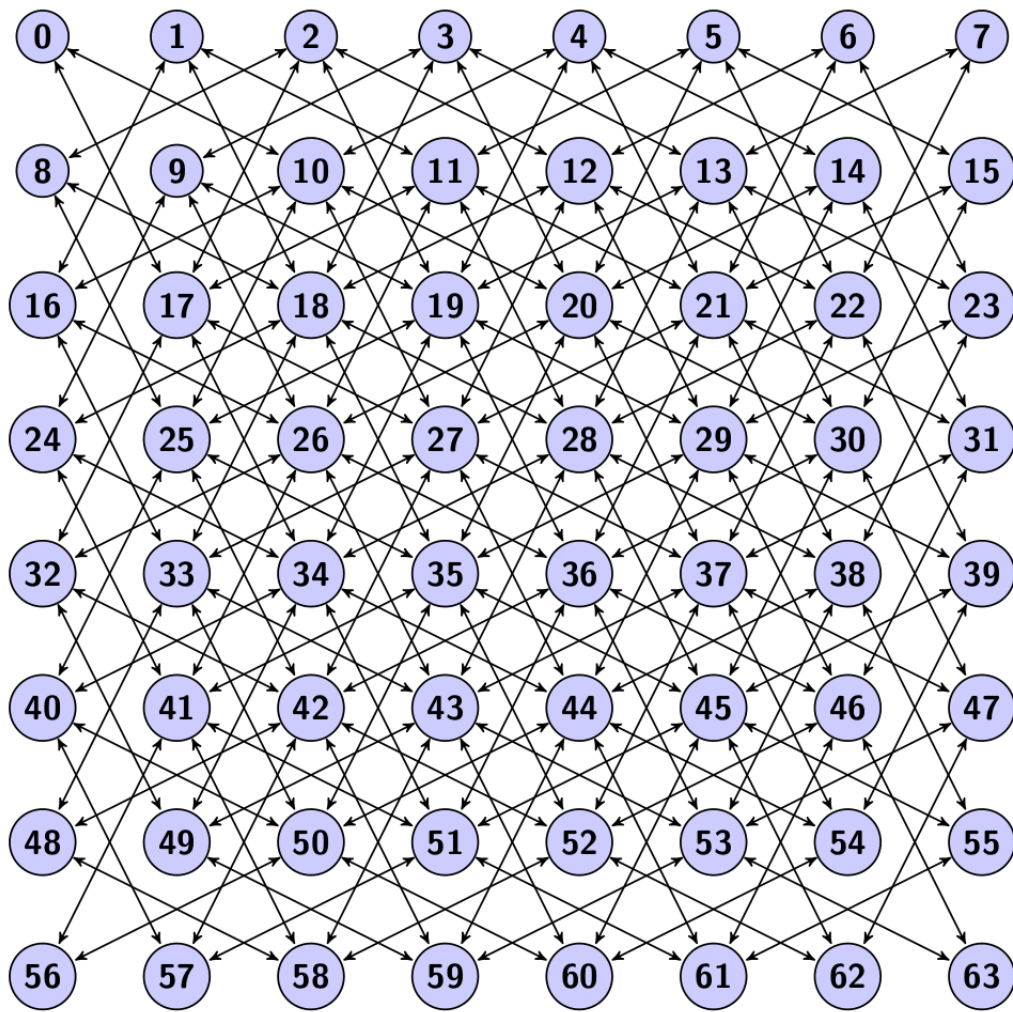
Por otro lado, un conjunto  $I \subseteq V$  es un *conjunto independiente* de  $G$  si no existe ningún eje de  $E$  entre dos vértices de  $I$ .

Definimos entonces un *conjunto independiente dominante* de  $G$  como un conjunto independiente que a su vez es un conjunto dominante del grafo  $G$ .

El problema de Conjunto Independiente Dominante Mínimo (CIDM) consiste en hallar un conjunto independiente dominante de  $G$  con mínima cardinalidad.

### 1.2. Paralelismo con “El señor de los caballos”

El problema “El señor de los caballos” es similar al problema de encontrar un *Conjunto Independiente Dominante Mínimo* considerando solamente **al tablero inicial como vacío (?)** y una familia específica de grafos. Esta es la familia de grafos donde cada nodo modela un casillero de un tablero de ajedrez y sólo existe un eje entre dos nodos  $v_1$  y  $v_2$  si el movimiento desde  $v_1$  a  $v_2$  o *viceversa* es un movimiento permitido para la pieza de un caballo.



Insertar imagen del grafo tablero

Insertada, espero que les guste

La relación con nuestro problema es la siguiente:

- “El señor de los caballos” busca un conjunto dominante, dado que intenta ocupar el tablero, y para ello requiere que o bien cada casillero tenga un caballo, o bien que cada casillero sea amenazado por un caballo.
- “El señor de los caballos” busca un conjunto mínimo, es decir, que utilice la menor cantidad de caballos posibles.
- “El señor de los caballos” **NO** busca un conjunto independiente, dado que si fuese necesario, un caballo puede ubicarse en un casillero que estuviese siendo amenazado por otro caballo.

### 1.3. Todo conjunto independiente maximal es un conjunto dominante

Sea  $G = (V, E)$  un grafo simple, un *conjunto independiente* de  $I \subseteq V$  se dice *maximal* si no existe otro conjunto independiente  $J \subseteq V$  tal que  $I \subset J$ , es decir que  $I$  está incluido estrictamente en  $J$ .

Todo conjunto independiente maximal es un *conjunto dominante*.

#### Demostración

Sean  $G = (V, E)$  grafo simple,  $I \subseteq V$  conjunto independiente maximal.

Quiero ver que  $I$  es un *conjunto dominante*:

Lo que es equivalente a probar que  $(\forall \text{ nodo } v \in V) ((v \in I) \vee (\exists \text{ nodo } w \in \text{adyacentes}(v), w \in I))$

Supongo por el absurdo que:  $(\exists \text{ nodo } v \in V) \text{ tq } ((v \notin I) \wedge (\forall \text{ nodo } w \in \text{adyacentes}(v), w \notin I))$

Considero a  $\text{adyacentes}(I)$  como el conjunto que se obtiene de concatenar todos los vecinos de cada elemento de  $I$ . Por lo tanto, es equivalente decir  $(\forall \text{ nodo } w \in \text{adyacentes}(I), w \notin I)$  y decir  $(v \notin \text{adyacentes}(I))$ .

$\Rightarrow v \notin I \wedge v \notin \text{adyacentes}(I)$

$\Rightarrow \exists$  conjunto independiente  $J: J \subseteq V$  tq  $J = I + \{v\}$

$J$  es independiente porque  $I$  lo era y al agregarle el nodo  $v$  se mantiene esta propiedad ya que  $v$  no pertenecía a  $I$  y además no estaba conectado a ningún nodo del conjunto  $I$ .

$\Rightarrow \exists J$  conjunto independiente tq  $I \subset J$ . **Absurdo!** ( $I$  era un conjunto Independiente Maximal)

El absurdo provino de suponer que  $I$  era un conjunto independiente maximal, pero no dominante. Por lo tanto,  $I$  debe ser un conjunto dominante.

se fijan que onda?

### 1.4. Situaciones de la vida real

Situaciones de la vida real que puedan modelarse utilizando CIDM:

- **Ubicación de estudiantes al momento de rendir un examen:** Encontrar la mejor manera de ubicar a todos los estudiantes en el aula, tal que ninguno esté suficientemente cerca de otro como para copiarse, pero tal que entre la mayor cantidad de estudiantes posibles. Esta situación es lo mismo que modelar el aula como un grafo donde cada asiento es un nodo y cada asiento es adyacente a los asientos que están a sus costados (en todas las direcciones); luego buscar el CIDM de dicho grafo.

- **Ubicación de servicios en ciudades:** Para minimizar costos, es probable que si se quiere situar centros de servicios (cualesquiera sean estos: hospitales, estaciones de servicio, distribuidoras, etc), se los situe de manera tal que tengan amplia cobertura, pero sin situar demasiados centros, es decir, situando la mínima cantidad. Tampoco se querría que un centro cubra la misma zona que otro centro. Si se modela a la ciudad, tomando cada zona (arbitraria, como barrios, o manzanas, o conjunto de manzana) como un nodo, en los que cada nodo es adyacente al nodo que representa la zona vecina, entonces el problema de situar estos centros minimizando costos, es igual a encontrar un *CIDM* en el grafo mencionado.

## 2. Algoritmo Exacto

De acuerdo a lo ya explicado en el inciso 1.2, podemos establecer una analogía con este problema y “El señor de los caballos”. Por lo tanto, la metodología empleada para la implementación del algoritmo exacto también fue la de *Backtracking*.

De este modo, nos vemos obligados a recorrer inteligentemente todos los conjuntos dentro del conjunto de partes del total de nodos  $V$ . Mediante el backtracking podemos realizar podas y estrategias para saltar algunas ramas de decisión donde se predice que no se va a poder encontrar la solución óptima allí.

### 2.1. Explicación y mejoras

Nuestro algoritmo recorre ordenadamente el conjunto de partes de  $V$  y por cada uno de ellos verifica que cumpla la función `esIndependienteMaximal()`. La misma devuelve 0 si es falso, la cantidad de nodos en el conjunto en caso contrario.

Es decir, se itera sobre los nodos y, considerando el nodo actual como presente o como ausente, termina encontrando el mínimo conjunto independiente maximal.

En una variable se acumula la solución óptima hasta el momento, la cual se actualiza cuando se encuentra un nuevo conjunto independiente maximal que tiene un cardinal menor al óptimo actual. En ella va a quedar la solución buscada luego de correr el algoritmo.

La poda que implementamos consistió en verificar que una futura solución posible no cuente con una cantidad igual o superior de nodos que la solución óptima obtenida hasta el momento. Esto quiere decir que si la óptima actual consiste de  $k$  nodos y nos encontramos ante la pregunta de agregar un nuevo nodo a una posible solución de  $k - 1$  nodos, ésta será a lo sumo tan buena como la que ya teníamos, por lo que no nos resulta útil contemplarla. De esta forma, se descartan rápidamente todas las soluciones peores o iguales que la actual. ... **completar**

Y las estrategias fueron ... **completar**

Poner el pseudocódigo...

```

unsigned int calcularCIDM(Matriz& adyacencia, unsigned int i, vector<unsigned int>& conjNodos,
vector<unsigned int>& optimo){

    if (conjNodos.esIndependienteMaximal()) then
        | optimo ← conjNodos;
        | return optimo.size();
    end
    if (i.estaEnRango()) then
        | if (conjNodos es más grande que el optimo actual) then
        | | return 0;
        | end
        | siNoAgrego ← calcularCIDM(adyacencia, i+1, conjNodos, optimo);
        | agrego el nodo i a conjNodos;
        | siAgrego ← calcularCIDM(adyacencia, i+1, conjNodos, optimo);
        | elimino el nodo i de conjNodos;
        | return el mejor entre siNoAgrego y siAgrego;
    end
    return 0;
}

```

**Algorithm 1:** algoritmo exacto

## 2.2. Complejidad Temporal

Al ser un algoritmo de *Backtracking* o fuerza bruta, tiene una complejidad exponencial. En este caso, la misma es de  $O(2^n)$ , siendo  $n$  la cantidad de nodos del grafo.

Se llega a dicha complejidad dado que para cada nodo, tenemos que preguntarnos qué ocurre, tanto si lo agregamos al conjunto como si no.

Dicho de otra forma, vamos a recorrer todos los conjuntos dentro del conjunto de partes del total de nodos de  $V$ . El cardinal de dicho conjunto de partes es  $2^n$ . **demostrar o no hace falta?**

Para cada uno de los subconjuntos, el algoritmo verifica si cumple la función `esIndependienteMaximal()`, la cual tiene una complejidad en peor caso de  $O(n^2)$ . De esta forma, nuestro algoritmo tiene complejidad  $O(2^n * n^2)$ , lo que es igual a  $O(2^n)$ .

Al tener en cuenta la poda utilizada, se puede ver que la misma no disminuye la complejidad teórica planteada dado que en el peor caso, podría haber que recorrer completamente el conjunto de partes. De todas formas, dicha poda mejora notablemente los tiempos de ejecución del algoritmo, como veremos más adelante, ya que descarta las soluciones peores que la óptima hasta el momento.

## 2.3. Experimentación

Para realizar la experimentación, se generaron grafos específicos, que nos permitieron ver el funcionamiento de nuestro algoritmo. Las conexiones entre los nodos de cada grafo son aleatorias, pero respetando la cantidad de ejes que nosotros definamos previamente.

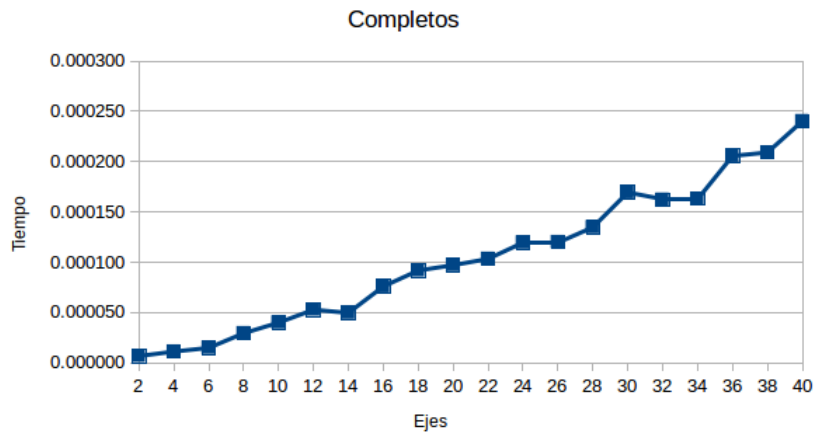
Las instancias con tiempos de ejecución bajos fueron corridas 100 veces, obteniendo luego un promedio de todas, con el fin de eliminar outliers. Aquellas instancias que tardaban mucho más tiempo, determinamos que los outliers no modificaban en forma considerable, por lo que no nos pareció necesario realizarlas reiteradas veces.

### 2.3.1. Mejor caso

De acuerdo a cómo fue planteado nuestro algoritmo, se puede ver que el mejor caso posible será cuando los grafos pasados por parámetro sean completos. Esto es así debido a la poda implementada, ya que cuando considera el primer nodo, ya encuentra una posible solución y luego descarta rápidamente todas las demás, ya que una solución con un solo nodo ya es una de las óptimas.

#### **Tabla**

Se puede ver que el gráfico tiende a una recta, lo que era esperable ya que encontrará la solución en el primer nodo, pero luego deberá descartar los siguientes  $n$  nodos, teniendo una complejidad de  $O(n)$ .

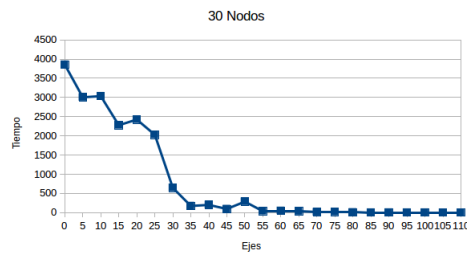
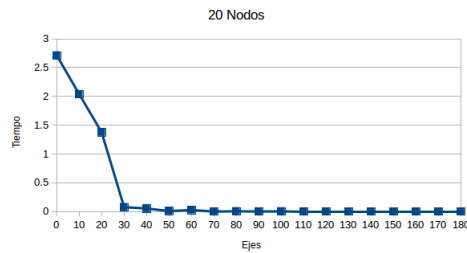
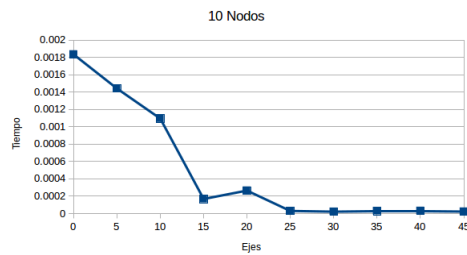


**2.3.2. Nodos fijos**

Para continuar viendo el comportamiento del algoritmo, decidimos realizar experimentos fijando la cantidad de nodos y variando la cantidad de ejes.

Los tiempos de ejecución para nodos fijos en 10, 20 y 30 fueron los siguientes:

**TABLA**



Como podemos ver, en todos los casos ocurre algo similar: cuando no tienen ejes la ejecución es más lenta, ya que debe recorrer todas las posibles opciones del conjunto de partes, sin lograr descartar



ninguna. A medida que se van agregando ejes, los tiempos van decreciendo y tienden a ser más lineales, ya que gracias a la poda, no es necesario recorrer todas las posibles soluciones.

### **3. Heurística Constructiva Golosa**

#### **3.1. Explicación**

#### **3.2. Complejidad Temporal**

#### **3.3. Comparación de resultados con solución óptima**

#### **3.4. Experimentación**

## 4. Heurística de Búsqueda Local

Un algoritmo de Búsqueda Local consiste en dos simples pasos: elegir una solución inicial y luego, **Aca explicarlo bien, esta horrible** iterar

modificarla (“mejorandola”), reemplazándola paso a paso con distintas soluciones que pertenezcan a la vecindad de la misma.

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.

Sea  $s \in S$  una solución inicial

Mientras exista  $s \in N(s)$  con  $f(s) > f(s)$

$s \leftarrow s$

### 4.1. Explicación

Considerando el problema a tratar, establecimos nuestros criterios para encontrar las soluciones iniciales y las vecindades.

#### 4.1.1. Elección de Solución Inicial

Al momento de seleccionar la solución Inicial, determinamos dos criterios.

##### Criterio I Solución Inicial: Golosa

Se realiza una ejecución del algoritmo Goloso de la Sección 3.

Esto quiere decir, se ordenan los nodos por grado de manera decreciente. Se eligen los nodos de a uno (de mayor a menor), de modo que al elegir un nodo se descartan sus vecinos para sus futuras elecciones.

##### Criterio II Solución Inicial: Secuencial

Los nodos al ser ingresados como parámetro del algoritmo tienen como identificador un número entre 0 y  $n - 1$ . El orden que vamos a utilizar para recorrerlos es el que haya sido dado cuando fueron ingresados como parámetro.

Lo primero que realizamos es tomar al nodo 0 y considerarlo parte de la solución. Se descartan todos los nodos vecinos a él y se continúa el proceso con el nodo que tenga menor número de *id*.

De este modo se forma un conjunto solución tal que en cada paso añade al nodo disponible que tenga su identificador número menor.

#### 4.1.2. Elección de Solución Inicial

Al momento de seleccionar la solución Inicial, determinamos dos criterios.

##### Criterio I Solución Inicial

##### Criterio I Solución Inicial

**poner cómo ejecutar cada uno de los criterios**

### 4.2. Complejidad Temporal

### 4.3. Experimentación

## 5. Metaheurística GRASP

### 5.1. Explicación

La metaheurística *Greedy Randomized Adaptive Search Procedure* (**GRASP**), es una mezcla de las dos heurísticas previas (vistas en 3 y 4). Dicho de manera simple: genera un punto de partida de forma golosa para el algoritmo de búsqueda local.

La distinción de este algoritmo radica en cómo se construye “golosamente” la solución inicial.

Como la sigla lo indica, consiste en un algoritmo *Goloso Randomizado*. Es decir que se escogen candidatos a solución inicial de una manera golosa ligeramente distinta a la utilizada en la sección anterior. El método *Greedy* de la sección 3 escoge a los nodos que van a pertenecer al conjunto solución, de a uno siempre eligiendo al que tiene mayor grado. En cambio, en este caso por cada paso no se elige al nodo de mayor grado sino que se elige uno al azar entre los que “mejor grado” tienen.

Hablar de “mejor grado” nos obliga a dar un criterio para ello, lo que da pie a la definición de la *Restricted Candidate List* (**RCL**), que es el conjunto de candidatos elegibles para la solución base.

La **solución inicial** se puede formar de diversas maneras. En todos los casos, se añade de a un nodo al conjunto solución hasta que se forme una solución válida. Tres formas para determinar la elección por nodo son:

- Elegir un nodo entre los  $\alpha\%$  nodos que tengan mayor grado hasta completar una solución válida.
- Elegir un nodo entre los  $\alpha$  nodos que tengan mayor grado hasta completar una solución válida.
- Elegir un nodo entre los nodos que cumplan determinada propiedad en un  $\alpha\%$  hasta completar una solución válida (como por ej: “Los nodos que tengan grado, a lo sumo,  $\alpha\%$  menor que el nodo de mayor grado”).

Optamos por implementar las primeras dos opciones para generar una solución inicial. Una vez obtenida la solución inicial bajo el método deseado, se aplica el algoritmo de *búsqueda local* explicado en el inciso 4 sin modificaciones.

Un aspecto también diferencial de esta heurística, es que no generamos una única instancia inicial, sino que se toma una determinada cantidad de ellas (acorde al criterio de parada). Se ejecuta el algoritmo para la primer instancia y se guarda la solución como óptima, luego si en alguna ejecución futura se mejora (se obtiene otra solución con menor cantidad de nodos) se actualiza óptima.

Las **vecindades** utilizadas son las mismas que se utilizaron en la heurística de búsqueda local (4).

El **criterio de parada** que adoptamos fue contabilizar las ejecuciones que no produjeron mejora, de modo que sólo se ejecute una determinada cantidad de repeticiones “malas”. Es decir, siempre que la ejecución otorgue una solución óptima con menos cantidad de nodos que la existente, se seguirá ejecutando. Pero si las ejecuciones no otorgan mejoras se suman al contador, de modo que al llegar a la cantidad indicada se terminará la ejecución.

Otra opción podría haber sido correr un número fijo de veces y quedarnos con la mejor solución encontrada; o también si conociéramos alguna cota, acercarnos a esta en un determinado porcentaje; o bien una combinación de todas.

## 5.2. Experimentación

Para analizar que combinación de vecindades de búsqueda local y elección de solución inicial, se acercan a encontrar el óptimo en distintos escenarios, se experimentó con una serie de grafos según criterios:

- Mantener los ejes fijos, variando la cantidad de nodos
- Mantener los ejes nodos, variando la cantidad de ejes
- Grafos Tablero (análogos a los del "Señor de los Caballos")

Lo que realizamos fue obtener un promedio de la cantidad de nodos que requería la solución óptima en cada ejecución del algoritmo. Luego, sabiendo cuántos hacen falta por el algoritmo exacto, divimos y obtuvimos en qué porcentaje la heurística falla en encontrar el óptimo verdadero.

La siguiente tabla muestra los valores obtenidos, las columnas 2, 3 y 4 indican el criterio aplicado al grafo.

Vecindad 2x1 indica que la vecindad usada fue la de quitar dos nodos y agregar uno, vecindad 3x1 quitar tres y agregar uno.

N representa el alfa elegido.

N mejores indica que como criterio de búsqueda de solución inicial, se tomó uno entre los N mejores según el criterio greedy establecido, N % mejores indica seleccionar uno entre los que pertenezcan al N % de los mejores

La diferencia en las dos tablas radica en el criterio de parada, para la primera, se tomó la decisión de no seguir buscando si no se modificó el óptimo luego de 5 iteraciones, para la segunda, si no se lo modificó luego de 10.

	Ejes Fijos	Nodos Fijos	Tableros
Vecindad 2x1 3 mejores	0.0706349206	0.1209055759	0.3333333333
Vecindad 2x1 5 mejores	0.0895743146	0.1270116294	0.125
Vecindad 2x1 7 mejores	0.1195959596	0.137305459	0.5333333333
Vecindad 2x1 10 mejores	0.1071500722	0.1220588458	0.6583333333
Vecindad 2x1 12 mejores	0.1856132756	0.0979182496	0.6583333333
Vecindad 3x1 3 mejores	0.218488456	0.1334987987	0.55
Vecindad 3x1 5 mejores	0.2886399711	0.1589123677	0.2916666667
Vecindad 3x1 7 mejores	0.2561724387	0.2224135112	0.7333333333
Vecindad 3x1 10 mejores	0.2550937951	0.2245729711	0.9416666667
Vecindad 3x1 12 mejores	0.2745815296	0.2259357556	0.9416666667
Vecindad 2x1 10 % mejores	0.0706349206	0.0957461757	0.125
Vecindad 2x1 25 % mejores	0.0672438672	0.1098904806	0.9916666667
Vecindad 2x1 50 % mejores	0.1260894661	0.1237291042	0.325
Vecindad 2x1 75 % mejores	0.3068614719	0.0978193541	0.35
Vecindad 2x1 100 % mejores	0.2456349206	0.1207595377	0.8333333333
Vecindad 3x1 10 % mejores	0.1113455988	0.1737028679	0.1666666667
Vecindad 3x1 25 % mejores	0.1935064935	0.1672620534	0.4833333333
Vecindad 3x1 50 % mejores	0.2420526696	0.1807902717	0.35
Vecindad 3x1 75 % mejores	0.3499422799	0.2287379706	0.4833333333
Vecindad 3x1 100 % mejores	0.262049062	0.240385845	0.8583333333

Cuadro 1: Promedio de porcentajes de error de GRASP con respecto al algoritmo exacto para cada vecindad y cada selección de solución inicial. Con criterio de parada fijado en 5 repeticiones sin mejorar la mejor solución hallada

	Ejes Fijos	Nodos Fijos	Tableros
Vecindad 2x1 3 mejores	0.1444444444	0.0970751665	0
Vecindad 2x1 5 mejores	0.0623015873	0.0818842512	0.4166666667
Vecindad 2x1 7 mejores	0.1297510823	0.0979054066	0
Vecindad 2x1 10 mejores	0.2647799423	0.1184019209	0.4166666667
Vecindad 2x1 12 mejores	0.1652128427	0.1197314734	0.7
Vecindad 3x1 3 mejores	0.2100324675	0.1881385561	0
Vecindad 3x1 5 mejores	0.2343542569	0.2078457808	0.6333333333
Vecindad 3x1 7 mejores	0.3034018759	0.2769039011	0
Vecindad 3x1 10 mejores	0.3261291486	0.2172843582	0.55
Vecindad 3x1 12 mejores	0.2863239538	0.2270750463	0.4833333333
Vecindad 2x1 10 % mejores	0.0944444444	0.0672958833	0
Vecindad 2x1 25 % mejores	0.0765873016	0.1072437596	0.1666666667
Vecindad 2x1 50 % mejores	0.1878246753	0.1158854895	0
Vecindad 2x1 75 % mejores	0.1126262626	0.0926548434	0.5416666667
Vecindad 2x1 100 % mejores	0.1781024531	0.1162941871	0.45
Vecindad 3x1 10 % mejores	0.1523268398	0.2101337874	0
Vecindad 3x1 25 % mejores	0.1327561328	0.2044939959	0.1666666667
Vecindad 3x1 50 % mejores	0.25	0.2308454751	0.25
Vecindad 3x1 75 % mejores	0.2540656566	0.209238541	0.6666666667
Vecindad 3x1 100 % mejores	0.2406024531	0.2213013582	0.4916666667

Cuadro 2: Promedio de porcentajes de error de GRASP con respecto al algoritmo exacto para cada vecindad y cada selección de solución inicial. Con criterio de parada fijado en 10 repeticiones sin mejorar la mejor solución hallada

Para seleccionar la mejor combinación de parámetros de la heurística, respecto a la solución encontrada contra la óptima del algoritmo exacto, tomamos como criterio que se minimice la suma de cada fila, es decir, que para casos donde los grafos incrementan su cantidad de ejes, o bien solo su cantidad de nodos, y tableros de caballos, la diferencia contra la óptima sea mínima.

Esto nos lleva a que la combinación óptima es: Vecindad 2x1 10% mejores con 10 iteraciones consecutivas sin ver modificaciones en el óptimo hallado hasta ese momento. Con un error promedio del 5,3%.

## 6. Comparación entre todos los métodos

Una vez elegidos los mejores valores de configuración para cada heurística implementada (si fue posible), realizar una experimentación sobre un conjunto nuevo de instancias para observar la performance de los métodos comparando nuevamente la calidad de las soluciones obtenidas y los tiempos de ejecución en función de los parámetros de la entrada. Para los casos que sea posible, comparar también los resultados del algoritmo exacto implementado. Presentar todos los resultados obtenidos mediante gráficos adecuados y discutir al respecto de los mismos.