DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 3

Heurísticas

Viernes 9 de Mayo de 2014

Algoritmos y Estructuras de Datos III
Entrega de TP

Grupo 7

Integrante	LU	Correo electrónico
Barrios, Leandro E.	404/11	ezequiel.barrios@gmail.com
Benegas, Gonzalo	958/12	gsbenegas@gmail.com
Melnik, Jonathan	571/09	jonathanmelnik@gmail.com
Vanecek, Juan	169/10	juann.vanecek@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

Índice

1. Introducción

En este trabajo práctico nos piden analizar el problema del *Camino Acotado de Costo Mínimo* ($CACM$), y desarrollar distintos algoritmos para resolverlo.

Dado un Grafo $G = (V, E)$, dos funciones de peso $\omega_1, \omega_2 : V \mapsto \mathbb{R}_+$, un natural K y dos nodos $u, v \in V$, el problema consiste en encontrar, entre todos los caminos P entre u y v que cumplen $\omega_1(P) \leq K$, el que minimice $\omega_2(P)$.

Donde si w es una función de peso definida sobre aristas, se entiende $w(P)$ como

$$\sum_{e \text{ arista de } P} w(e)$$

$CACM$ es un problema conocido, y tiene muchas aplicaciones en la vida real. Una agencia de vuelos puede estar interesada en ofrecer el viaje más corto entre dos ciudades, dado un cliente con un presupuesto acotado. Los nodos representan ciudades. u y v son las ciudades de origen y destino, respectivamente. Una arista es un vuelo particular entre dos ciudades. El peso por ω_1 es el costo del pasaje y el peso por ω_2 es la duración del vuelo. Un camino es una secuencia de vuelos, es decir, un vuelo que puede o no tener escalas. K viene a ser el presupuesto del cliente. El camino buscado es entonces el que, entre todos los vuelos (con o sin escalas) entre la ciudad de origen y destino que el cliente puede pagar, tiene la menor duración.

Otro ejemplo similar está relacionado con el Mapa Interactivo de la Ciudad de Buenos Aires. Se busca llegar de un punto de la ciudad a otro en el menor tiempo, aunque se puede especificar la máxima cantidad de metros por caminar. Los nodos son alturas de calles, es decir puntos geográficos sobre alguna calle. Una arista entre dos nodos es un segmento de calle - junto con un medio de transporte - que une dos puntos geográficos. El peso por ω_1 es la longitud del segmento si el medio de transporte es 'caminar'y 0 si no. El peso por ω_2 es el tiempo que demora recorrer el segmento usando el correspondiente medio de transporte. El valor K, especificado por el usuario, es la máxima cantidad de metros que está dispuesto a caminar. Un camino entre un punto u de origen a otro punto v de destino es una sucesión de segmentos recorridos con un correspondiente medio de transporte. Se busca, entre todos los caminos cuyos metros caminados totales no exceden K, el de menor duración.

Aunque $CACM$ es un problema conocido, no se conocen algoritmos polinomiales que lo resuelvan y se estima que pertenece al conjunto de problemas NP. En este trabajo se analizaran varios métodos para resolverlo: una solución exacta y 5 aproximaciones a través de heurísticas. En concreto, se implementarán los siguientes algoritmos:

1. Un *Backtracking* como algoritmo exacto.
2. Una heurística *constructiva greedy*.
3. Una heurística de *búsqueda local*.
4. Una heurística *GRASP*.

El enfoque estará puesto en experimentar sobre cada uno de estos algoritmos, analizando su complejidad y la calidad de las solucion en el caso de las heurísticas. Se intentará definir diferentes familias de grafos sobre los cuales poder obtener resultados concluyentes sobre el comportamiento del algoritmo.

2. Informe de modificaciones

- Se revisaron en general todas las secciones de acuerdo a las correcciones.
- Se agregaron dos nuevas secciones: *Pautas generales de medición* y *Pautas para la generación de grafos*.

3. Instrucciones de compilación

Para compilar el proyecto completo: entrar en la carpeta `src` y correr `make all`.

Para compilar cada una de las partes por separado: en la carpeta `src`, estan las carpetas `backtracking`, `grasp`, `greedy_heuristic_{A,B,C}` y `local_search`. Se puede compilar cada parte de forma independiente, entrando en cada carpeta y ejecutando `make`.

Funcionamiento de los archivos `Makefile`:

- Hay un archivo `Makefile.common` que se usa como input para los `Makefile` del resto de las carpetas. La sintaxis para incluirlo es: `include ../Makefile.common`
- Por defecto, compila el archivo que tenga el mismo nombre de la carpeta, y busca el `.cpp` y el `.h`. Por ejemplo, si estamos en la carpeta `backtracking` va a buscar `backtracking.cpp` y `backtracking.h` y lo va a compilar en `backtracking`.
- Siempre va a crear una carpeta `OBJS` en donde guarda los archivos `.o`
- Cuando se desean incluir archivos que se encuentren en la carpeta `common`, se debe anteponer `COMMON_OBJS` al `include`:

  ```
  COMMON\_OBJS := ClassName1 ClassName2
  include ../Makefile.common
  ```

  en donde `ClassName` es el nombre de la clase. Por ejemplo, `COMMON_OBJS := Graph Edge`
- Se pueden agregar targets específicos que luego del `include ../Makefile.common`.

4. Pautas generales de medición

Para medir cierto valor durante la ejecución de un algoritmo frente a una determinada instancia, se tomaron los siguientes recaudos:

- Se ejecutó el algoritmo frente a la misma instancia una cierta cantidad de veces, en este trabajo cincuenta veces fue suficiente.
- De todos los valores obtenidos se descartaron aquellos que sean menores al percentil 25 o mayores al percentil 75.
- Los valores restantes fueron ordenados y se tomó como valor final a la mediana.

5. Pautas para la generación de grafos

Se generaron dos familias principales de instancias de CACM:

- Instancia *'aleatoria'* El objetivo fue generar una instancia de la que no se pueda suponer nada, para evitar cualquier tipo de sesgo.

  Se tomó un valor fijo de cota para todos los parametros, llamado C.

  - n se eligió uniformemente entre $\{1, ..., C\}$.
  - k se eligió uniformemente entre $\{1, ..., C\}$.
  - u se eligió uniformemente entre $\{1, ..., n\}$.
  - v se eligió uniformemente entre $\{1, ..., n\}$.
  - m se eligió uniformemente entre $\left\{0, ..., \frac{n(n-1)}{2}\right\}$.
  - Para cada arista se eligieron como extremos dos enteros uniformente entre $\{1, ..., n\}$ (si los nodos correspondientes ya eran adyacentes, se vuelve a elegir). Los pesos según ω_1 y ω_2 se eligieron como valores de punto flotante uniformemente en el intervalo $(0, C]$.

  Observación: C depende del algoritmo para el que haya sido generada la instancia. Por ejemplo, para el *backtracking* no tiene sentido usar $C > 30$, pues el tiempo de ejecución puede llegar a ser demasiado grande.

- Instancia *'mágica'*

  Estas instancias fueron generadas para la medición de calidad de las heurísticas. Se conoce de antemano el valor de la solución óptima, lo que en instancias generales solo se podría saber corriendo un algoritmo exacto, que puede tardar mucho en grafos grandes. Además se pretende que el peso de las aristas esté *balanceado*, es decir, a pesos bajos por ω_1 correspondan pesos altos por ω_2, y a la inversa.

  - n es elegida como parámetro.
  - k vale $n - 1$.
  - u vale 1.
  - v vale n.
  - m se eligió uniformemente entre $\left\{n - 1, ..., \frac{n(n-1)}{2}\right\}$.
  - primero se agregan las siguientes $n - 1$ aristas: (1, 2), (2, 3), ..., $(n - 1, n)$ formando un camino hamiltoniano entre u y v. Los pesos de estas aristas, según tanto ω_1 como ω_2, van a ser 1. De esta forma generamos un camino I válido, ya que $\omega_1(I)$ vale $n - 1 \leq k$.
  - Para las aristas que quedan se eligen como extremos dos enteros a, b uniformente entre $\{1, ..., n\}$ (si los nodos correspondientes ya eran adyacentes, se vuelve a elegir). El peso va a guardar relación con $|b - a|$, valor que denotaremos *diff*.

El peso según ω_1 va a ser generado de manera uniforme entre $\{0, ..., 2 \times diff\}$. De esta forma la esperanza del peso va a ser *diff*. El peso según ω_2 va a valer $2 \times diff$ menos el peso según ω_2. De esta forma la esperanza de este peso también va a valer *diff*. Así quedan balanceados los dos pesos simétricamente alrededor de *diff*. Por ejemplo, si tenemos la arista (1, 9), *diff* vale 8. Si ω_1 resulta 6, entonces ω_2 lo fijamos en 10.

Sea P un camino, entonces la esperanza de su peso - tanto por ω_1 como por ω_2 - es la sumatoria de las esperanzas de las aristas que lo constituyen. Es la sumatoria del *diff* de cada arista. Los caminos que nos interesan van de u a v. Cualquier camino de 1 a n va a tener una suma de *diff* mayor o igual a $n-1$, ya que por cada unidad que uno avanza entre 1 y n utilizando una arista, la arista suma 1 a su *diff*. Como se deben avanzar $n-1$ unidades, entonces la sumatoria de los *diff* va a ser mayor o igual a $n-1$.

Lo importante es que el centro de simetría de los pesos de P - la sumatoria de los *diff* - va a ser mayor o igual que $n-1$. Luego, si $\omega_2(P) < n-1$, entonces $\omega_1(P) = 2 \times$ sumatoria(*diff*) - $\omega_2(P) > n-1$. Por lo tanto no sería un camino válido pues se excedería del valor de k. Entonces el camino óptimo tiene $\omega_2 \geq n-1$. El camino I (válido) definido previamente es, entonces, uno de los caminos óptimos. En la familia de grafos *magica* sabemos luego que el camino óptimo tiene $\omega_1 = n-1$.

Otra ventaja de generar grafos de esta forma es que permite una gran cantidad de caminos alternativos para llegar de u a v. Se espera que estos nodos sean los más alejados entre si en todo el grafo, ya que para llegar de 1 a n las aristas del camino van a tener una sumatoria de *diff* - que es la esperanza de la longitud del camino - de por lo menos $n-1$. La esperanza de distancia para cualquier camino entre el nodo 1 y el nodo $n/2$, por ejemplo, es mayor a $n/2-1$.

Si uno forma un camino entre 1 y n como secuencia creciente de vértices, éste tiene esperanza de peso $n-1$. Por ejemplo:

- $1 \rightarrow$ n — La esperanza de peso es $n-1$
- $1 \rightarrow$ n/2 $\rightarrow$ n — La esperanza de peso es $(n/2-1)+(n/2) = n-1$
- $1 \rightarrow 7 \rightarrow$ n — La esperanza de peso es $(6+(n-7) = n-1$

Entonces todos las secuencias crecientes de vertices que empiezan en 1 y terminan en n van a ser candidatos a ser el camino óptimo, ya que todos tienen la esperanza de peso - tanto por ω_1 como por ω_2 - alrededor de $n-1$, que es el valor de k. Esto permite un análisis más interesante de la calidad de las distintas heurísticas.

Para cada familia de instancias, se consideraron tres subfamilias:

- *Densidad baja*

  La cantidad de aristas se definió de antemano como $3n$, es decir, $O(n)$.

- *Densidad media*

  La cantidad de aristas se definió de antemano como $n\sqrt{n}$, es decir, $O(n\sqrt{n})$.

- *Densidad alta*

  La cantidad de aristas se definió de antemano como la máxima: $\frac{n(n-1)}{2}$ es decir, $O(n^2)$.

6. Desarrollo del TP

6.1. Backtracking

6.1.1. Descripción

El algoritmo a cada paso tiene un camino $P = [v_1 = u, \ldots, v_{i-1}]$ e intenta agregar al camino un nuevo nodo v_i de la adyacencia de v_{i-1}, hasta que llega al nodo destino V. De esta forma recorre todos los caminos entre U y V. De entre todos los caminos tales que el peso por ω_1 es menor o igual a K, se guarda el de menor ω_2.

Se implementaron las siguientes podas:

- Se exploran solamente caminos simples. Si un camino tiene ciclos, removiéndolo se obtiene uno con menor longitud tanto en ω_1 como en ω_2. Por lo tanto la solución que buscamos no puede tener ciclos, y nos restringimos a caminos simples. La poda se implementó guardando un arreglo que marca para cada nodo si es parte del camino o no. No se agrega nuevamente si ya es parte del camino.
- Si el camino parcial $P = [v_1 = u, \ldots, v_{i-1}]$ cumple:

  $omega_1(P)$, sumado a la distancia según ω_1 entre v_{i-1} y V, es mayor que K

  entonces se abandona la rama, es decir, no se recorren los caminos que empiezan en P, ya que cualquiera de éstos no va a cumplir con la restricción de que $omega_1$ sea menor o igual a K.
- Si el camino parcial $P = [v_1 = u, \ldots, v_{i-1}]$ cumple:

  $omega_2(P)$, sumado a la distancia según ω_2 entre v_{i-1} y V, es mayor o igual que el peso según ω_2 del mejor camino encontrado hasta el momento

  entonces se abandona la rama, es decir, no se recorren los caminos que empiezan en P, ya que cualquiera de éstos no va a proporcionar una solución mejor que la ya encontrada.

Para saber las distancias entre todos los vértices y v, al principio se corre dos veces Dijkstra, utilizando como peso primero ω_1 y luego ω_2.

A continuación, el pseudocódigo de la función `backtrack`.

Algoritmo 1 `backtrack`(Edge e)

```
1: Node n ← e.toNode
2: currentBranch.path.push_back(n)
3: currentBranch.totalOmega1 += e.omega1
4: currentBranch.totalOmega2 += e.omega2
5: visited[n] ← true
6: podar ← currentBranch.totalOmega1 + distanceToVOmega1[n] > K or current-
   Branch.totalOmega2 + distanceToVOmega2[n] ≥ bestSolutionFound.totalOmega2
7: si not podar entonces
8:     si n = V entonces
9:         bestSolutionFound ← currentBranch
10:    sino
11:        para each Node a in adjacencyList[n]  hacer
12:            si not visited[a]  entonces
13:                Edge f = incidencyMatrix[n][a]
14:                backtrack(f)
15:            fin si
16:        fin para
17:    fin si
18: fin si
19: currentBranch.path.pop_back()
20: currentBranch.totalOmega1 -= e.omega1
21: currentBranch.totalOmega2 -= e.omega2
22: visited[n] ← false
```

6.1.2. Complejidad

Utilizamos una implementación de Dijkstra sobre heap, cuya complejidad es $O(m \times log(n))$. Como lo corremos $2 = O(1)$ veces, la complejidad es $O(m \times log(n))$. La complejidad principal del algoritmo está dada por la función `backtrack`, que procedemos a analizar.

Sea x la cantidad de nodos no visitados, es decir la cantidad de nodos que no están en el camino parcial de u a v. Sea $T(x)$ la cantidad de operaciones que realiza la función `backtrack` cuando la cantidad de nodos no visitados es x. Sea N la cantidad de nodos del grafo. Se propone la siguiente relación:

$$T(x) \leq xT(x-1) + O(N)$$

La cantidad de llamadas a `backtrack` que se hacen es a lo sumo x, que son los nodos no marcados como visitados, y en esas llamadas la cantidad de nodos marcados no visitados disminuye en uno ya que se marca como visitado al nodo n. n no estaba marcado previamente pues entonces no se hubiése llamado a la función con la arista incidente a él.

Además de las llamadas recursivas se ejecuta un ciclo con una cantidad de iteraciones igual al tamaño de la adyacencia de n. La adyacencia es a lo sumo N, la cantidad de

nodos del grafo. Todas las operaciones que restan: accesos aleatorios a arreglos, sumas, son $O(1)$.

Veamos que $T(x) \in O((x+1)!)$, es decir

$$T(x) \leq c(x+1)!$$

para una constante c y para todo $x \geq x_0$. Por inducción en x:

- Caso base: $x = 1$ Si se llama a `backtrack` con un solo nodo no visitado, este es n, y no quedan nodos para llamar recursivamente. Luego la cantidad de operaciones es N.

  $$N \leq c \times 2! \Leftrightarrow N \leq N \times 2!$$

  que es verdadero, tomando $c = N$ y $x_0 = 1$.

- Paso inductivo: Supongo $T(x-1) \leq cx!$, quiero ver que $T(x) \leq c(x+1)!$

  $$T(x) \leq xT(x-1) + O(N) \leq_{pasoinductivo} x \times c \times x! + N$$

  $c \times x \times x! + N \leq c \times (x+1)! \Leftrightarrow N \leq c \times (x+1) \times x! - c \times x \times x! \Leftrightarrow N \leq c \times x!$

  que es válido para los valores que elegimos de c y x_0.

El valor que nos interesa es $T(N)$, que es $O((N+1)!)$. La complejidad final es $O(m \times log(N)) + O((N+1)!) = O((N+1)!)$.

6.1.3. Experimentación

Se efectuó una experimentación sobre el tiempo de ejecución del algoritmo. Primero se evaluó sobre instancias *aleatorias*, de distinto tamaño de entrada. Se obtuvieron dos conclusiones importantes.

Primero, la curva que se puede observar representa una función que crece cada vez más rápido, es decir que no es lineal. Al mismo tiempo, no parece crecer tan rápido como la función factorial que expresa la complejidad teórica del algoritmo.

Segundo, hay mucha amplitud de valores para tamaños de entrada similares. Esto puede deberse a la naturaleza aleatoria de la instancia. Los nodos de origen y destino pueden estar muy cerca o muy lejos, puede haber muchos o pocos caminos entre ellos.

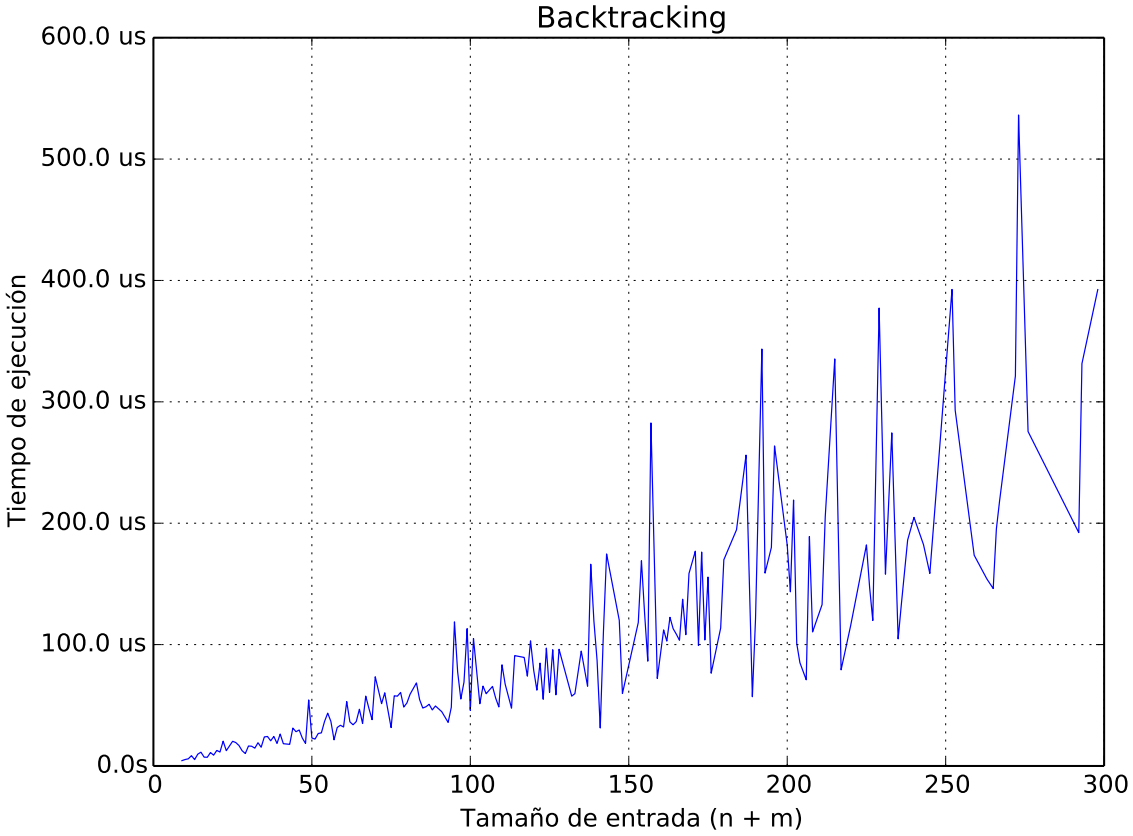


Figura 1: Tiempo de ejecución de `backtracking` en instancias *aleatorias*

A continuación evaluamos el comportamiento del algoritmo frente a instancias *mágicas*. Se notaron grandes diferencias con la experimentación anterior.

Como primera impresión, se puede observar una curva que parece ser la característica de las funciones factoriales. Los tiempos de ejecución fueron mucho mayores que frente a instancias *aleatorias*. Para los mismos tamaños de entrada, en las últimas no se llegó a tiempos de 1 milisegundo, mientras que en las primeras se superó ampliamente el segundo.

La explicación que se encuentra, según lo desarrollado en la sección de *Generación de instancias*, es que las instancias *mágicas* proveen una gran cantidad de caminos entre u y v que son candidatos a ser el óptimo. Mientras que las instancias *aleatorias* no garantizan ni la lejanía de u y v ni una gran cantidad de caminos candidatos a ser el óptimo. Probablemente una vez que se encuentre el camino las otras ramas de solución se puedan descartar rápidamete.

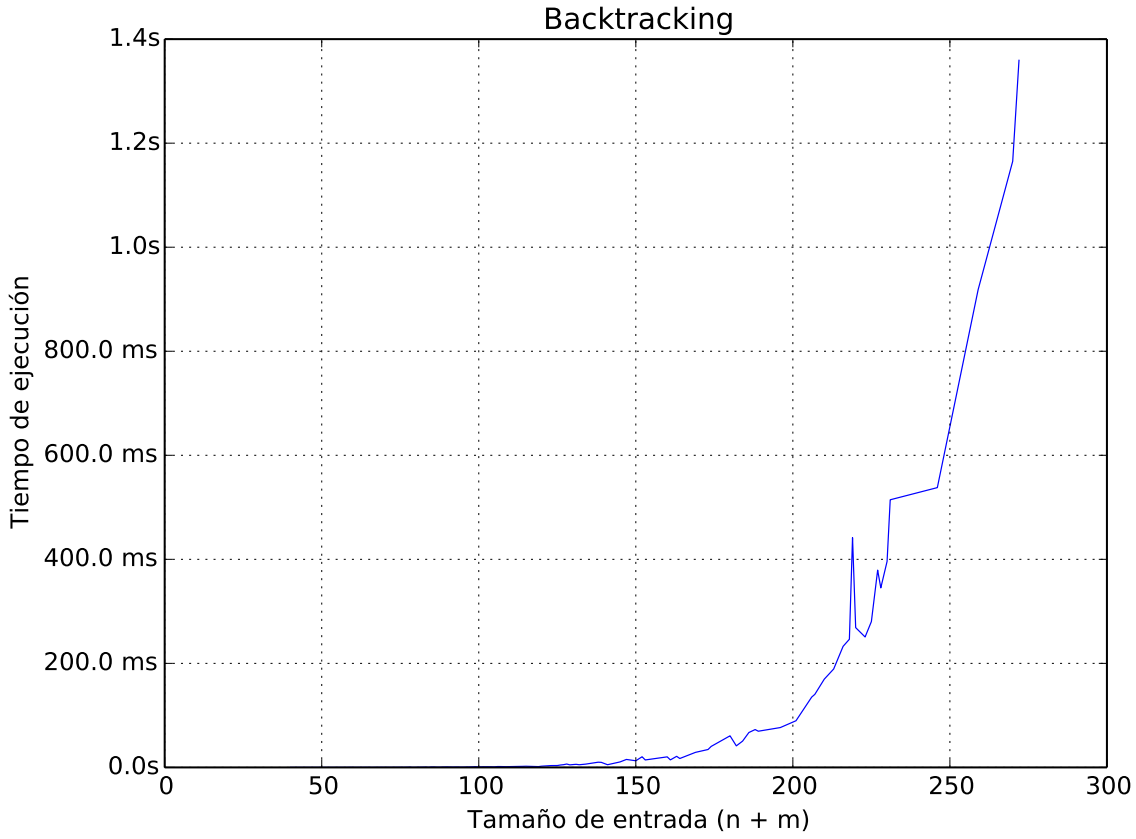


Figura 2: Tiempo de ejecución de `backtracking` en instancias *mágicas*

Se quiso investigar la performance del algoritmo en relación a n, la cantidad de nodos.

Para ello se generaron instancias *mágicas* de distintos valores de n, y con m fijo en la mitad de las aristas posibles, es decir $\frac{n(n-1)}{4}$.

Para ver la magnitud de los valores fue apropiada una escala logarítmica. Se puede percibir una muy ligera concavidad positiva de la curva, lo que vendría a indicar que la función crece más rápido que una exponencial(en escala logarítmica toda recta con pendiente m representa la función m^x). Con estos resultados se añade evidencia al cálculo de complejidad efectuado, que indica una complejidad factorial en función de n.

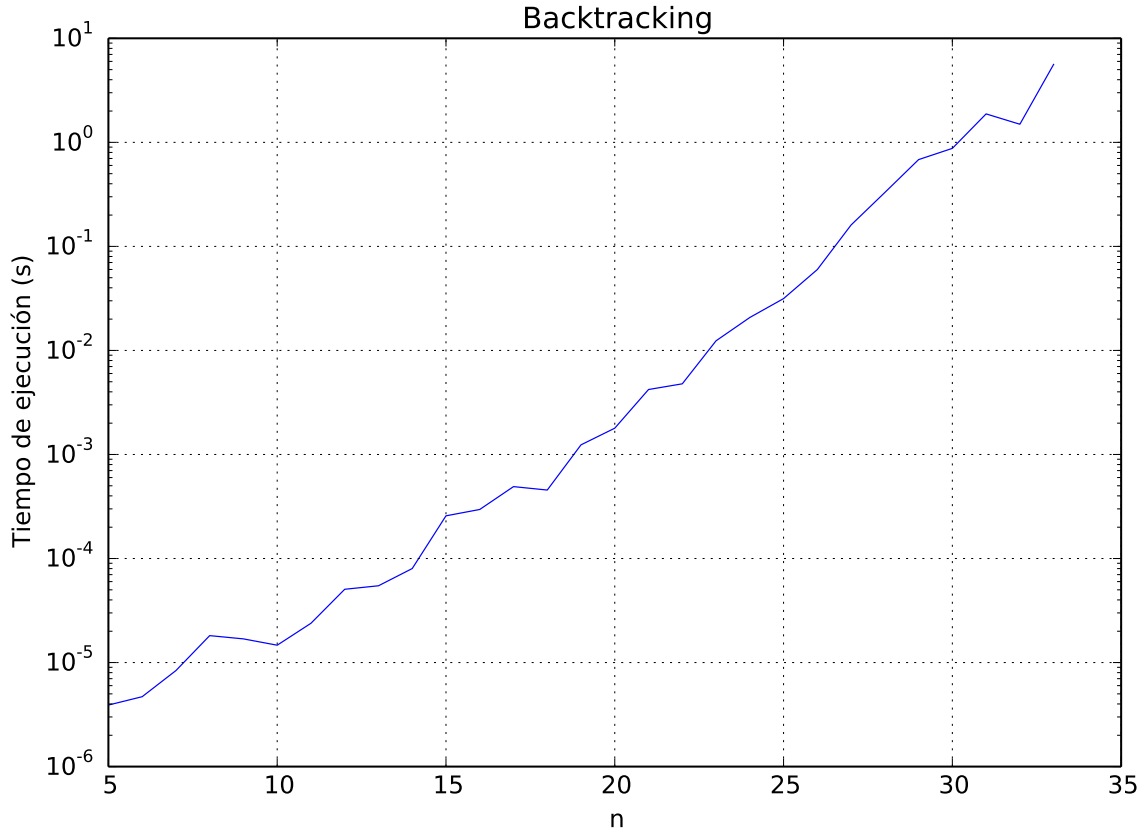


Figura 3: Tiempo de ejecución de `backtracking` en instancias *mágicas*, en función de n

A continuación se exploró el comportamiento del algoritmo en función de m, la cantidad de aristas. El valor de n estuvo fijo en 25 y la cantidad de aristas se varió entre $n-1$ y la máxima cantidad para un grafo de 25 vertices.

Nuevamente con escala logarítmica, se observó un gran crecimiento en el tiempo de ejecución a medida que se aumenta m. Ésto era predecible ya que se aumenta el tamaño de la adyacencia de los vértices y por lo tanto la cantidad de iteraciones del ciclo en el algoritmo. A más alto nivel, la cantidad de caminos entre u y v tiende a aumentar. Sin embargo, la curva observada tiene concavidad negativa, por lo que representa a una función que crece más lento que una exponencial.

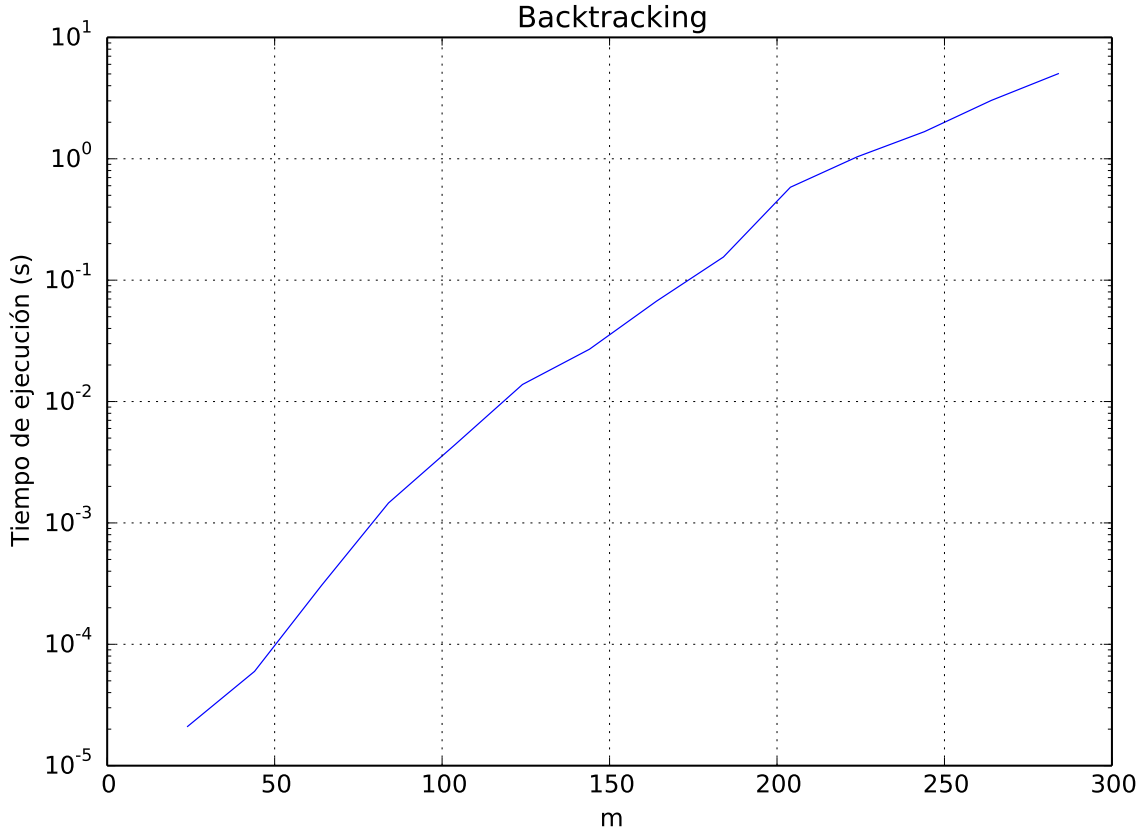


Figura 4: Tiempo de ejecución de `backtracking` en instancias *mágicas*, en función de m

6.2. Greedy

6.2.1. Descripción

Frente a un problema, un algoritmo goloso va armando la solución global usando la heurística de eligir a cada paso la solución óptima local. En general estos algoritmos son eficientes y simples de diseñar e implementar, pero puede ser que nunca lleguen a la solución óptima global del problema.

De acuerdo a la definición de Brassard[1], un algoritmo goloso se compone de los siguientes elementos:

1. Un concepto de candidato, y una estructura de solución a partir de candidatos.
2. Un conjunto de candidatos que ya han sido considerados y seleccionados.
3. Un conjunto de candidatos considerados y rechazados.
4. Una función que comprueba si cierto conjunto de candidatos constituye una solución a nuestro problema, ignorando si es o no óptima por el momento.
5. Una función de factibilidad, que determina si es o no posible completar el conjunto añadiendo otros candidatos para obtener al menos una solución de nuestro problema.
6. Una función de selección que indica en cualquier momento cuál es el más prometedor de los candidatos restantes, que no han sido seleccionados ni rechazados.
7. Una función objetivo, que da el valor de la solución que hemos hallado.

Lo que busca el algoritmo goloso es encontrar la secuencia de candidatos que constituya una solución, y que optimice el valor de la función objetivo. Este algoritmo avanza paso a paso. Inicialmente, el conjunto de elementos seleccionados está vacío. Entonces, en cada paso se considera añadir a este conjunto el mejor candidato sin coniderar los restantes, de acuerdo a nuestra función selección. Si el conjunto ampliado de candidato seleccionados ya no fuera factible, rechazamo el candidato que estamos considerando en ese momento. Sin embargo, si el conjunto aumentado sigue siendo factible, entonces añadimos el candidato actual al conjunto de candidatos seleccionados, en donde pasará a estar desde ahora en adelante. Cada vez que se amplía el conjunto de candidatos seleccionados, comprobamos si éste constituye ahora una solución para nuestro problema. A partir de este esquema, al agregar siempre soluciones óptimas locales a mi conjunto, al finalizar lo que se espera encontrar es la solución óptima global.

El algoritmo de Dijsktra para encontrar caminos mínimos en un grafo pesado es un algoritmo goloso, que funciona y es correcto, como lo fue demostrado por Bassard et al. en el libro mencionado.

[1] Brassard G., Bratley P., *Fundamental of Algorithmics*, Prentice Hall, 1996. (c)

Dado un grafo $G = (V, X)$, Dijkstra guarda un conjunto S de nodos que ya fueron recorridos y un vector π con la distancia mínima encontrada de un nodo u a todo el resto. Como invariante, el valor de π de los nodos en S ya es su distancia a u. En cada fase de Dijkstra, se selecciona un nuevo nodo de $V \backslash S$ cuyo valor en π sea mínimo y se añade a S, actualizando si es necesario π. Al finalizar, π es el vector con la mínima distancia a todos los nodos.

Para resolver el problema se va a implementar Dijkstra con tres funciones objetivo diferentes, que toman una arista y devuelven un peso para ella:

1. $f_A(e) = \omega_1(e)$
2. $f_B(e) = \omega_2(e)$
3. $f_C(e) = \omega_1(e)\omega_2(e)$

A continuación detallamos el pseudocódigo de Dijkstra parametrizado para ejecutarse con distintas funciones de peso. La implementación utililza un heap para mantener la lista de nodos no visitados.

Algoritmo 2 `Dijkstra`

In: Grafo $G = (V, X)$, nodo inicial v_0, ObjectiveFunction f
Out: Arreglo π con camino mínimo en función de f a cada nodo
Arreglo *previo* con nodo anterior en el camino mínimo, para cada nodo

```
1: π(v) = ∞    ∀v ∈ V
2: previo(v) = null    ∀v ∈ V
3: π(v_0) = 0
4: S = ∅
5: para i = 1 ... n − 1 hacer
6:     v ← nodo de V\S de mínimo π.
7:     para each w ∈ V\S adyacente a v hacer
8:         si π(v) + f((v, w))) < π(w) entonces
9:             π(w) = π(v) + f((v, w)))
10:            previo[w] ← v
11:        fin si
12:    fin para
13:    S = S ∪ {v}
14: fin para
15: retornar π, previo
```

Algoritmo 3 `Reconstrucción del camino mínimo`

1: $path \leftarrow []$
2: actual $= v$
3: **mientras** $previo[actual] != null$ **hacer**
4: $\quad path.push\_front(actual)$
5: $\quad actual \leftarrow previo[actual]$
6: **fin mientras**
7: **retornar** path

6.2.2. Complejidad

De acuerdo al libro de Brassard antes mencionado, la complejidad de Dijkstra implementada sobre un heap es $O(m \log n)$. Se utilizó la implementación `priority_queue` de la *Standard Template Library* de $C++$ que garantiza la complejidad de heap[2].

[2] http://www.cplusplus.com/reference/queue/priority_queue/priority_queue/

6.2.3. Familias Malas - Greedy A: utiliza la función de peso f_A

Dado un grafo $G = (V, E)$, obtenemos el camino mínimo entre u y v según ω_1.

A continuación definimos una familia de grafos en los cuales nuestro algoritmo puede devolver resultados muy malos.

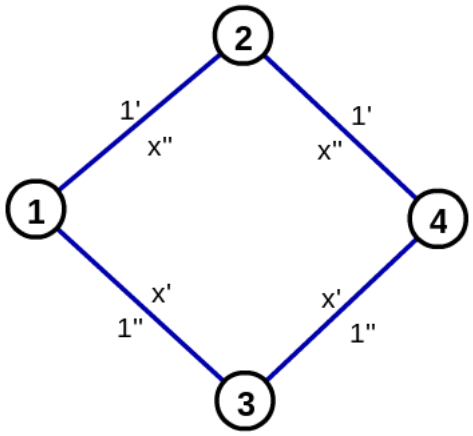


Figura 5: Familia de grafos malos para el Greedy A. En adelante, con un apóstrofe haremos referencia al peso por ω_1 y con dos al peso por ω_2

Para ir de 1 a 4 hay dos caminos posibles: (C_1) $1 \to 2 \to 4$; (C_2) $1 \to 3 \to 4$

$$\omega_1(C_1) = 2 \quad (1)$$
$$\omega_2(C_1) = 2x \quad (2)$$
$$\omega_1(C_2) = 2x \quad (3)$$
$$\omega_2(C_2) = 2 \quad (4)$$

Supongamos que K vale 2x, es decir, los dos caminos son válidos. Nuestro algoritmo elige C_1. $\frac{\omega_2(C_1)}{\omega_2(C_2)} = x$. Como x lo podemos variar, este cociente puede ser tan grande como queramos. Es decir que el algoritmo goloso puede devolver una solución arbitrariamente alejada de la óptima.

El algoritmo puede encontrar una solución no factible. En ese caso, sa sabe que no existe una solución posible, ya que la solución encontrada tiene un valor por ω_1 menor o igual al de cualquier otro camino.

6.2.4. Familias Malas - Greedy B: utiliza la función de peso f_B

Dado un grafo $G = (V, E)$, obtenemos el camino mínimo entre u y v según ω_2.

A continuación definimos una familia de grafos en los cuales nuestro algoritmo puede devolver resultados muy malos.

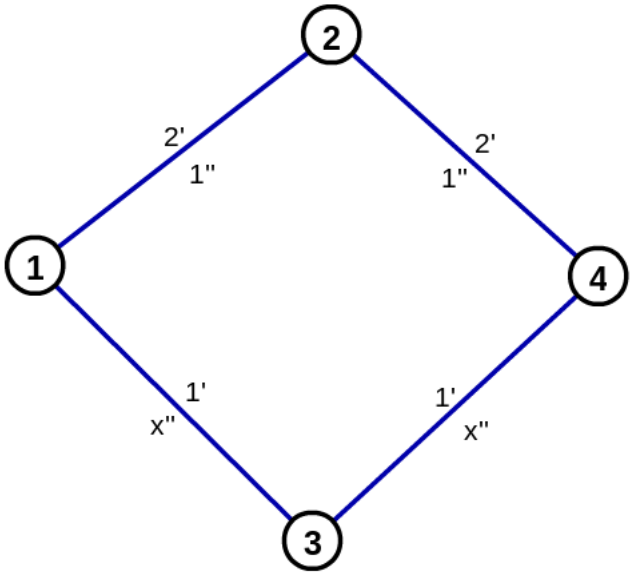


Figura 6

Para ir de 1 a 4 hay dos caminos posibles: (C_1) $1 \to 2 \to 4$; (C_2) $1 \to 3 \to 4$

$$\omega_1(C_1) = 4 \quad (5)$$
$$\omega_2(C_1) = 2 \quad (6)$$
$$\omega_1(C_2) = 2 \quad (7)$$
$$\omega_2(C_2) = 2x \quad (8)$$

Supongamos que K vale 2. Nuestro algoritmo elige C_1, pero al no ser válido, se ve obligado a devolver "no". Pero C_2 era una solución válida. Ésto se cumple para cualquier valor de x.

Esta heurística, si devuelve una solución válida, entonces devuelve la solución óptima. Sin embargo, en cualquier instancia interesante de $CACM$ - que no se reduzca a encontrar el camino mínimo por ω_2 - la heurística devolvería un camino inválido.

6.2.5. Familias Malas - Greedy C: utiliza la función de peso f_C

Dado un grafo $G = (V, E)$, obtenemos el camino mínimo entre u y v según $\omega_1\omega_2$.

Esta heurística se puede comportar de la misma forma que el Greedy B, si consideramos la familia mala de grafos desarrollada para el Greedy B, restringiendonos a valores de x mayores a 2. Se eligiría C_1 para minimizar el producto de las funciones de peso. Como no es válido, se deberá devolver "no", a pesar de que C_2 era válido.

6.2.6. Conclusión: unificación de las tres heurísticas

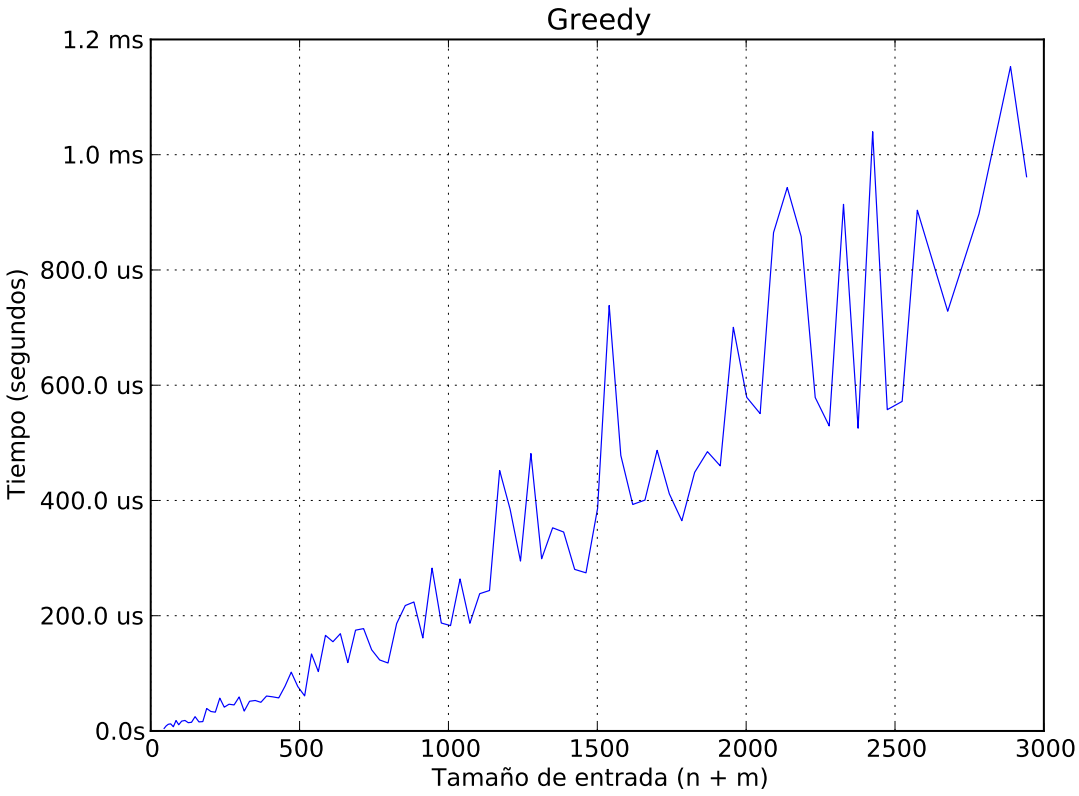
Las tres heurísticas *GreedyA*, *GreedyB* y *GreedyC* no son muy útiles por separado. Se considerara la heurística *Greedy* como:

Algoritmo 4 `Greedy`

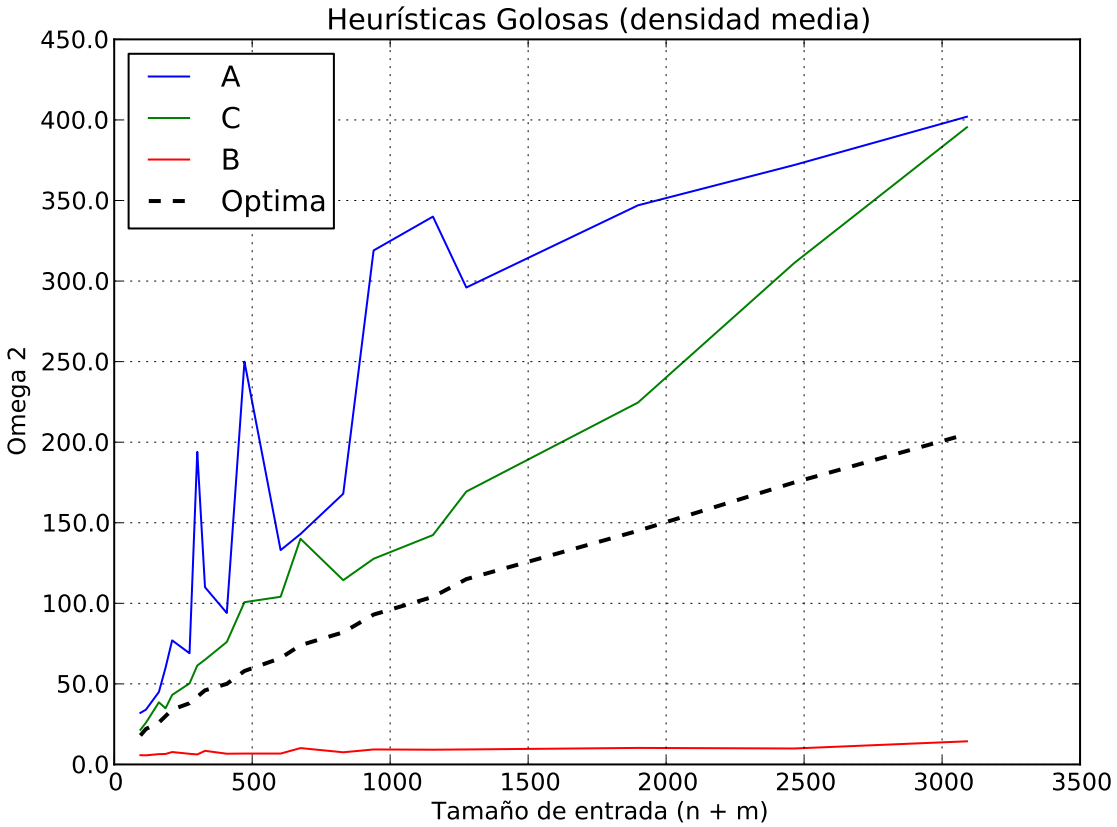
```
1: pathB ← GreedyB()
2: si valido?(pathB) entonces
3:     retornar pathB
4: fin si
5: pathA ← GreedyA()
6: pathC ← GreedyC()
7: si valido?(pathC) entonces
8:     si ω2(pathA) < ω2(pathC entonces
9:         retornar pathA
10:    sino
11:        retornar pathC
12:    fin si
13: fin si
14: si valido?(pathA) entonces
15:    retornar pathA
16: sino
17:    retornar no
18: fin si
```

6.2.7. Experimentación

Utilizar instancias `mágicas` resultó muy importante. Nos cercioramos primero de que exista un camino entre u y v que cumpla con la restricción de k. Luego fuimos agregando caminos, como para dar varias opciones a los algoritmos, con la particularidad de que eran *balanceados*. Ésto significa que si el peso según ω_1 del camino era bajo, entonces le asignábamos un ω_2 alto. Más aún, para todo camino C entre u y v, $\omega_1(C)$ y $\omega_2(C)$ estaban distribuidos simetricamente con respecto a k. Es decir, el camino P más corto según ω_2 que cumpla con la restricción de k era tal que $\omega_1(P) = \omega_2(P) = k$. En particular, nosotros insertábamos este camino P en el grafo, de modo que llevamos cuenta del valor de la solución óptima. Ésto nos va a facilitar evaluar la calidad de una solución cuando la instancia es muy grande como para correr el backtracking.



Se puede interpretar la curva de tiempo de ejecución vs. tamaño de entrada como una función cercana a la lineal, aunque se puede percibir que está ligeramente por encima de ésta, lo que puede se puede interpretar como el peso del factor logarítmico de la expresión de la complejidad teórica.



Lo que salta a simple vista es el bajo peso según ω_2 de la heurística B. ¡Pareciera ser mejor solución que la óptima! Sin embargo, recordando lo expuesto previamente, nos damos cuenta que si el peso según ω_2 es menor que el de la óptima, entonces el peso según ω_1 es mayor que el de la óptima, es decir, que k. Por lo tanto no son soluciones válidas. Ésto era de esperar - la heurística B sólo va construyendo su solución sin reparar siquiera en ω_1.

Por otro lado tenemos la heurística A, que sólo mira ω_1 y por lo tanto termina bastante alejada de la solución óptima. La heurística C resultó ser un acierto, se acerca bastante a la solución óptima, pero no descuida el mantenerse dentro de la cota de k.

6.3. Local Search

La búsqueda local es un método heurístico para mejorar una dada solución factible a un problema. Se basa en, a partir de una solución inicial s, mejorar esa solución iterativamente, tomando la mejor solución de un conjunto de vecinos de s. El conjunto de los vecinos se determina mediante algún criterio y se usa una función objetivo para comparar las soluciones en la vecindad de s y discernir cual es la mejor.

La búsqueda local se puede expresar de la siguiente manera:

Sea $s \in S$ una solución inicial
Mientras exista $s' \in N(s)$ con $f(s') < f(s)$:
    $s \leftarrow s'$

Siendo $N(s)$ la vecindad de s y f la función objetivo que se quiere minimizar. Una versión de la búsqueda local se encarga además de que el vecino que elige minimice la función objetivo entre todas los vecinos.

6.3.1. Solución inicial

Como solución inicial se utilizó la heurística `Greedy` desarrollada en el punto anterior, consistente en elegir la mejor de tres corridas del algoritmo de Dijkstra con distintas funciones de peso.

6.3.2. Definición de la vecindad

En cada iteración definimos la vecindad de s, $N(s)$, de la siguiente manera: dado el grafo G, y una solución formada por un camino $c \in G$, definimos sus soluciones vecinas como aquellas resultantes de tomar un subcamino $d_{n_1,n_2} \subseteq c$ entre un par de nodos n_1, n_2 cualesquiera, y reemplazarlo por otro camino $d^*_{n_1,n_2}$ en G, de tal forma que el camino $c^* = c - d_{n_1,n_2} + d^*_{n_1,n_2}$ resultante cumpla:

- $\omega_1(c^*) <= K$
- $\omega_2(c^*) < \omega_2(c)$
- c^* no contiene ciclos

De la forma descripta, dada una solución formada por un camino c definimos su vecindad como el conjunto S^* de todos los caminos c^* posibles.

Corremos previamente n veces Dijkstra, una vez desde cada nodo. Guardamos los resultados en una matriz tal que en la posición $[i][j]$ guarda el camino mínimo del nodo i al j. Cada camino $d^*_{n_1,n_2}$ es el camino mínimo entre n_1 y n_2 según ω_2, se obtiene de la matriz. Debe además cumplir $\omega_1(c^*) \leq K$. De todos los c^*, se elige el que minimice ω_2. Esto se enmarca en la elección de vecinos utilizando el método *Steepest Descent* consistente en elegir al vecino que minimice la función objetivo.

Puede darse el caso de que, al reemplazar un camino d_{n_1,n_2} por $d^*_{n_1,n_2}$, existan nodos en este último que también existan en $c^* = c - d_{n_1,n_2}$, de tal forma que el camino c^* resultante contenga ciclos. Para resolver este inconveniente, luego de efectuar cada reemplazo, se llama al método removeCycles, el cual toma un camino, ya sea con ciclos o sin ciclos, y se encarga de construir una versión del mismo la cual no contenga ciclos.

Ahora, para calcular $\omega_2(c^*)$, lo calculamos como

$$\omega_2(c) - \omega_2(d_{n_1,n_2}) + \omega_2(d^*_{n_1,n_2})$$

, valores que ya tenemos calculados.

6.3.3. Pseudocódigo

El algoritmo está implementado en la función `main`:

Algoritmo 5 *main*(int tipo_solucionInicial, Graph g, Nodo n1, Nodo n2)

```
1: crearMatrizCaminosMinimos(g)
2: Solution solucion = Greedy(g, n1, n2)
3: si ω1(solucion) ≤ K entonces
4:     mientras True hacer
5:         Solution nuevaSolucion = dameMejorVecino(solucion)
6:         si nuevaSolucion == NULL entonces
7:             break
8:         fin si
9:         solucion = nuevaSolucion
10:    fin mientras
11: fin si
```

Algoritmo 6 *dameMejorVecino*(Solution solucionOriginal)

```
1: Solution mejorSolucion = &solucionOriginal
2: vector< int > nodos = &nodos(solucionOriginal)
3: para i=0; i < size(nodos); i++ hacer
4:     para j=i+1; j < size(nodos); j++ hacer
5:         Solution subSolucion = crearSubSolucionEntre(solucionOriginal, nodos[i], nodos[j])
6:         Solution solucion_ij = dameCaminoResueltoEntre(nodos[i], nodos[j])
7:         removerCiclos(solucion_ij)
8:         Solution nuevaSolucion_ω2 = ω2(solucionOriginal) - ω2(subSolucion) + ω2(solucion_ij)
9:         Solution nuevaSolucion_ω1 = ω1(solucionOriginal) - ω1(subSolucion) + ω1(solucion_ij)
10:        si nuevaSolucion_ω2 < ω2(mejorSolucion) && nuevaSolucion_ω1 ≤ K) entonces
11:            mejorSolucion = crearSolucionReemplazandoCamino(solucionOriginal, solucion_ij)
12:        fin si
13:    fin para
14: fin para
15: return mejorSolucion
```

Algoritmo 7 *crearSolucionReemplazandoCamino*(Solution orig, Solution sub)

```
1: Solution res
2: res = obtenerCaminoHasta(nodos(sub)[0])
3: res += sub
4: int subSize = size(nodos(sub))
5: res += obtenerCaminoDesde(nodos(sub)[subSize-1])
6: return res
```

Algoritmo 8 *removerCiclos*(Solution solution)

```
1: int nextNodes[nodeCount]
2: para i=0; i < nodeCount; i++ hacer
3:     nextNodes[i] = 0
4: fin para
5: para i=0; i < solution.path.size(); i++ hacer
6:     edge = solution.path[i]
7:     nextNodes[edge.fromNode] = edge.toNode
8: fin para
9: vector< int > newPath
10: firstNode = solution.path[0].fromNode
11: lastNode = solution.path[solution.path-1].toNode
12: next = firstNode
13: newPath.push(firstNode)
14: mientras  nextNodes[next] != lastNode hacer
15:     next = nextNodes[next]
16:     newPath.push(next)
17: fin mientras
18: newPath.push(lastNode)
19: Solution newSolution
20: para i=0; i < newPath.size()-1; i++ hacer
21:     edge = solution.getEdgeBetween(newPath[i], newPath[i+1]
22:     newEdge = Edge(edge)
23:     newSolution.addEdge(edge)
24: fin para
25: return newSolution
```

6.3.4. Complejidad

Implementamos el algoritmo en la función *main*. La complejidad del algoritmo resulta la suma de obtener la solución inicial y el ciclo que se usa para mejorarla buscando un mejor vecino en cada iteración. Además, en *main*, inicialmente, se llama a una función crearMatrizCaminosMinimos[3], que abordaremos más adelante.

Creamos la solucion inicial usando Greedy. Esta función como explicamos en el apartado anterior, devuelve un camino mínimo entre 2 nodos usando Greedy A, B y C. Cada una de las 3 funciones Greedy usan la función resolverConDijkstra que ejecuta Dijkstra para encontrar todos los caminos mínimos entre n1 y los demás nodos y después hace un recorrido inverso desde n2 hasta n1 para dar con el camino mínimo entre ellos. Como se comprobó en el apartado anterior, la complejidad de Dijkstra es $O(mlog(n))$ y el recorrido inverso es $O(n)$, por lo que en total la complejidad de obtenerSolucionInicial es $O(mlog(n))$. Cabe notar que la función objetivo usada en Dijkstra no influye en la complejidad, ya que solo compara los valores de ω_1 y ω_2, por

[3]: Estamos haciendo uso de esta función que no se encuentra en el código fuente como tal, pero es para facilitar la legibilidad del pseudocódigo. En el código fuente se crea la matriz de camino mínimos cuando se inicializa NeighbourhoodSelector en la función main en localsearch.cpp.

lo que tiene complejidad $O(1)$.

Para mejorar la solución usamos un ciclo y en cada iteración obtenemos el mejor vecino del camino actual usando dameMejorVecino. Ejecutamos el ciclo mientras en la última iteración hayamos encontrado una mejora al camino actual. Dado que un vecino consiste en reemplazar la porción del camino actual que une 2 nodos n_1 y n_2 por el camino mínimo según ω_2 entre ellos, y que cada vez que se hace el reemplazo se disminuye ω_2 del camino actual, se pueden hacer a lo sumo tantos reemplazos como caminos mínimos entre todo par de nodos del grafo existan. La cantidad de caminos mínimos entre todo par de nodos de un grafo es $n \times (n-1)/2$, ya que cada nodo tiene un camino mínimo hacia todos los demás y no a sí mismo. Esto es, si hay n nodos, el nodo n_1, tiene un camino mínimo hacia los nodos n_2, ..., n_n, el nodo n_2 tiene un camino hacia n_3, ..., n_n ya que no se vuelve a contar el camino entre n_1 y n_2 y así hasta n_{n-1} que tiene un camino hasta n_n. Luego, la cantidad máxima de iteraciones es $n \times (n-1)/2$.

Para analizar la complejidad de cada iteración hay que analizar la complejidad de dameMejorVecino. Lo primero que hace la función es obtener el arreglo de nodos de la solución pasada por parámetro, que llamamos solucionOriginal. Cómo el camino de solucionOriginal está representado como una lista de ejes, lo que hace es iterar por todos los ejes y tomar el nodo1 del eje y al final adicionar el nodo2 del último eje. Por ende, tomando t como la cantidad de nodos en el camino, en cada iteración, por cada eje del camino, se toma un nodo y esto se hace t-1 veces y luego se añade el nodo final. Como t puede ser a lo sumo n, entonces la complejidad de esto resulta $O(n^2)$. Luego de ejecuta un doble for de $t \times (t-1)/2$ iteraciones, que en cada iteración intenta mejorar el mejor camino encontrado hasta el momento, que llamamos mejorSolucion. Inicialmente mejorSolucion es igual a solucionOriginal. Para encontrar una mejor solución en cada iteración hacemos lo siguiente:

- Creamos el subcamino de solucionOriginal entre los nodos n_1 y n_2.
- Obtenemos el camino mínimo entre los nodos n_1 y n_2.
- Obtenemos un nuevo camino reemplazando el el subcamino entre los nodos n_1 y n_2 por el camino mínimo entre ellos.

Para crear el subcamino de solucionOriginal entre n_1 y n_2 usamos crearSubSolucionEntre. Esta función recibe un par de nodos y un camino y devuelve el subcamino que une a los nodos. Para esto tiene que recorrer a lo sumo todos los nodos del camino, o sea t, que puede ser a lo sumo n.

Para obtener el camino mínimo entre los nodos n_1 y n_2 usamos una optimización que es que en la función *main*, al comienzo, ejecutamos crearMatrizCaminosMinimos que crea una matriz de caminos mínimos entre todo par de nodos del grafo. Generamos esta matriz ejecutando Dijkstra para todos los nodos usando como función objetivo ω_2. Usamos esta matriz para obtener en $O(1)$ el camino mínimo entre los nodos n_1 y n_2. Como crearMatrizCaminosMinimos ejecuta un Dijkstra por cada nodo, su complejidad es $O(n \times (mlogn))$.

Para obtener el nuevo camino reemplazando el subcamino entre n_1 y n_2 por el camino mínimo entre ellos, usamos crearSolucionReemplazandoCamino. Lo que hacemos

es generar una nueva solución que es la concatenación de 3 caminos: el camino mínimo entre n_1 y n_2 y los dos pedazos del camino original sin el subcamino que unía n_1 y n_2. Para crear el camino, hay que recorrer el solucionOriginal y añadir todos los nodos hasta n_1 y luego añadir todos los nodos del camino mínimo y al final añadir todos los nodos desde n_2 hasta el final de solucionOriginal. Por ende es como iterar sobre el nuevo camino que a lo sumo puede tener n nodos y por ende la complejidad resulta $O(n)$.

Como es posible que la solución creada a partir de unir los caminos contenga ciclos, usamos la función removerCiclos que se encarga de remover cualquier ciclo en la nueva solución. Esta función itera sobre los nodos de la solución y almacena en un arreglo de enteros el nodo siguiente a cada nodo. De modo que cada elemento i corresponde al nodo i y su valor es el nodo siguiente a ese nodo i. En el caso de que exista un ciclo y un nodo i esta al comienzo de un ciclo, primero el nodo i tendrá como siguiente a otro nodo j, pero cuando se itera sobre la solución y se llega nuevamente al nodo i, al asignarle su siguiente, ese nodo ya no será j, sino otro nodo mas adelente en la solución. Con el arreglo creado de esa manera podemos reconstruir la solución sin ciclos. El camino se obtiene recorriendo el arreglo como una lista enlazada, donde cada elemento tiene el índice del siguiente. Usando este arreglo creamos una nueva solución, que es un vector de ejes con $omega_1$ y $omega_2$ asignados. Los ejes los construimos usando los nodos del arreglo y obtenemos los valores de $omega_1$ y $omega_2$ de la solución recibida por parametro. La complejidad de remover los ciclos resulta igual a la suma de construir el arreglo, que es $O(n)$, ya que a lo sumo puede tener n vertices y luego crear la nueva solución que es $O(n^2)$, ya que hay que crear hasta $n-1$ ejes y por cada eje hay que obtener sus valores de $omega_1$ y $omega_2$ de la solución original. Luego en total es $O(n^2)$.

Entonces, resulta ser que encontrar el mejor vecino, consiste en ejecutar el doble for de algo que tiene complejidad $O(n+1+n+n^2)$, o sea $O(n^2)$, entonces en total es $O(n^2 \times n^2) = O(n^4)$.

Como habíamos explicado que dameMejorVecino se ejecuta en un ciclo de hasta $n \times (n-1)/2$ iteraciones, o sea $O(n^2)$, y entonces resulta que la complejidad de encontrar el mejor vecino es $O(n^2 \times n^4) = O(n^6)$.

Finalmente la complejidad total resulta ser la suma de las complejidades de crearMatrizCaminosMinimos, 2 veces obtenerSolucionInicial y n^2 veces dameMejorVecino. Es decir, $O(n \times (mlogn) + 2 \times (mlogn) + n^6) = O(n^6)$.

6.3.5. Familias malas

Veamos que nuestra heurística de búsqueda local puede quedar arbitrariamente lejos de la solución óptima.

Presentamos la siguiente familia de grafos:

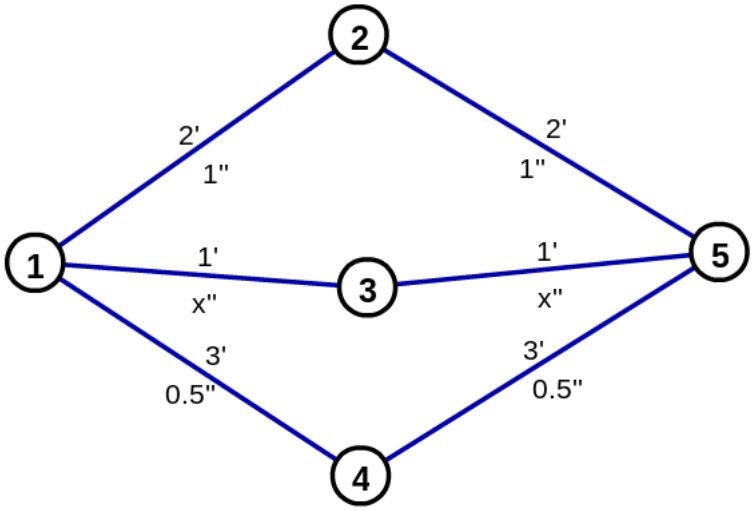


Figura 7

Para ir de 1 a 5 hay tres caminos posibles: (C_1) $1 \to 2 \to 5$; (C_2) $1 \to 3 \to 5$; (C_3) $1 \to 4 \to 5$;

$$\omega_1(C_1) = 4 \tag{9}$$
$$\omega_2(C_1) = 2 \tag{10}$$
$$\omega_1(C_2) = 2 \tag{11}$$
$$\omega_2(C_2) = 2x \tag{12}$$
$$\omega_1(C_3) = 6 \tag{13}$$
$$\omega_2(C_3) = 1 \tag{14}$$

Supongamos que K vale 4. Como decíamos, partimos del camino mínimo entre u y v de acuerdo a ω_1, C_2. El algoritmo va a intentar intercambiar C_2 por C_3, el camino que minimiza ω_2. Sin embargo, como éste se pasa del límite K, el algoritmo no puede seguir y devuelve C_2. Sin embargo C_1 era una solución mejor.

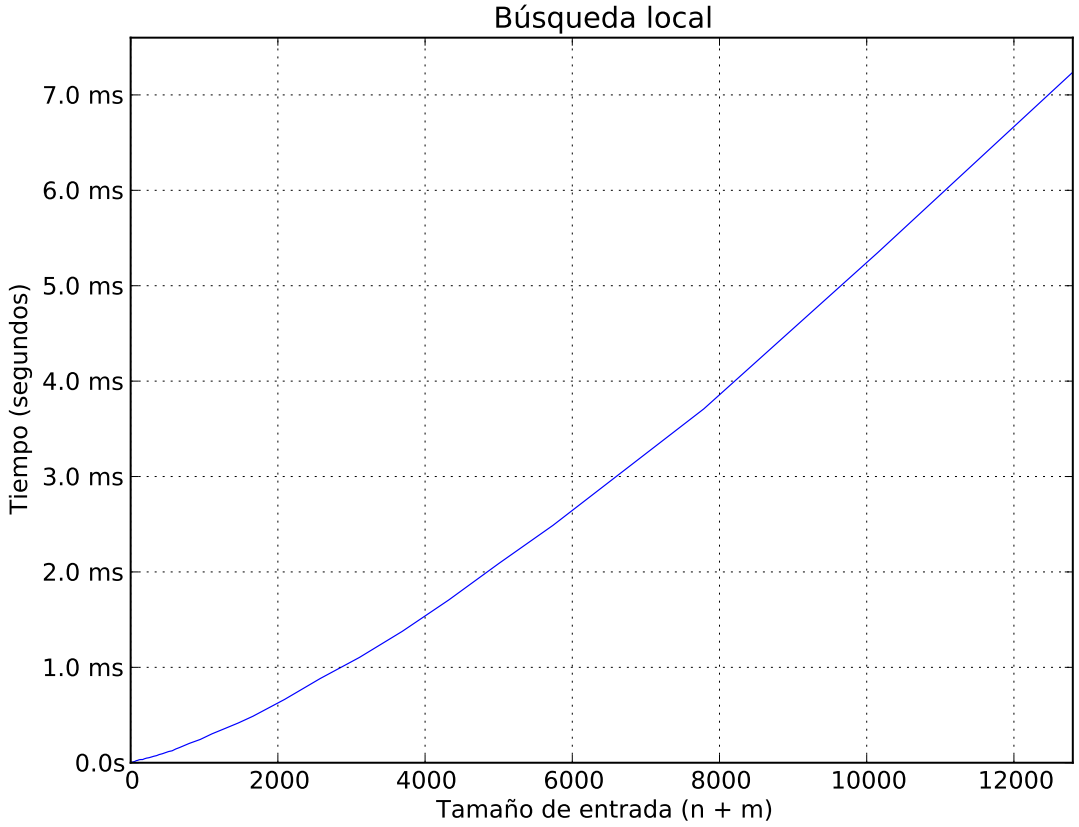
$\frac{\omega_2(C_2)}{\omega_2(C_1)} = x$.

Haciendo crecer el valor de x podemos encontrar grafos en los que nuestro algoritmo devuelve una solución arbitrariamente lejos de la óptima.

En general, la heurística falla cuando los caminos mínimos por ω_2 entre todo par de nodos tienen un valor de ω_1 demasiado alto, que deja a la solución sin vecinos factibles ya que éstos se sobrepasan de la restricción de K.

6.3.6. Experimentación

Pudimos analizar el tiempo de ejecución corriendo el algoritmo en grafos de un tamaño considerable. Utilizamos instancias del tipo *mágico*, variando el valor de n y eligiendo m al azar entre todos los posibles valores dado n. A continuación presentamos un gráfico.

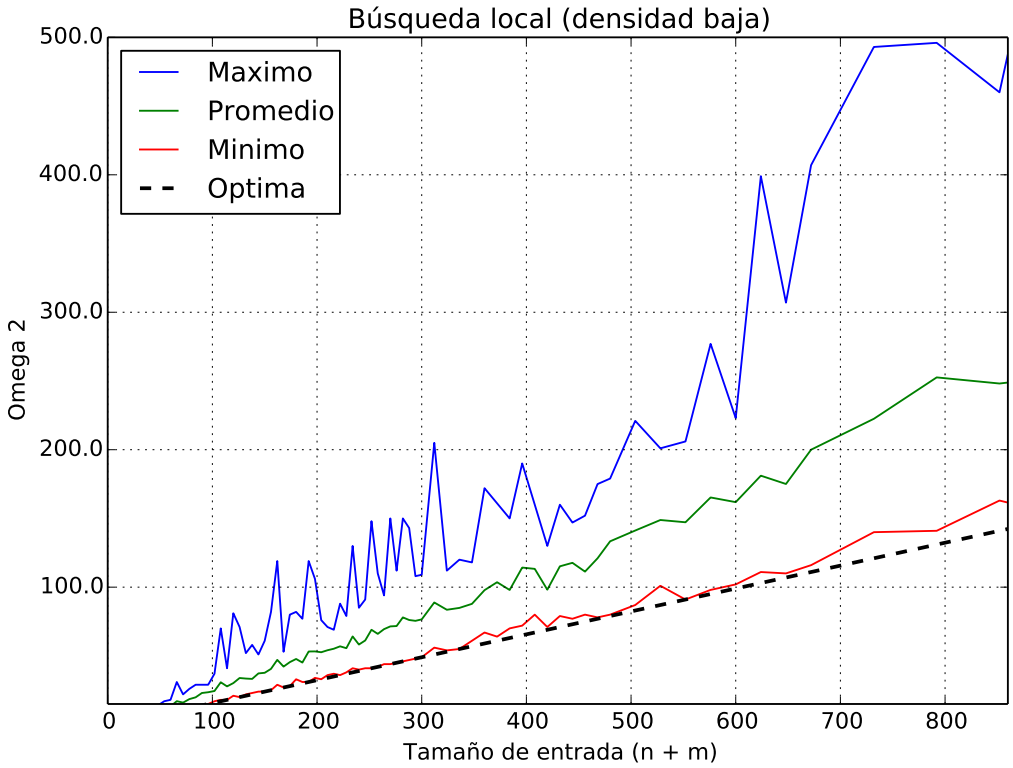


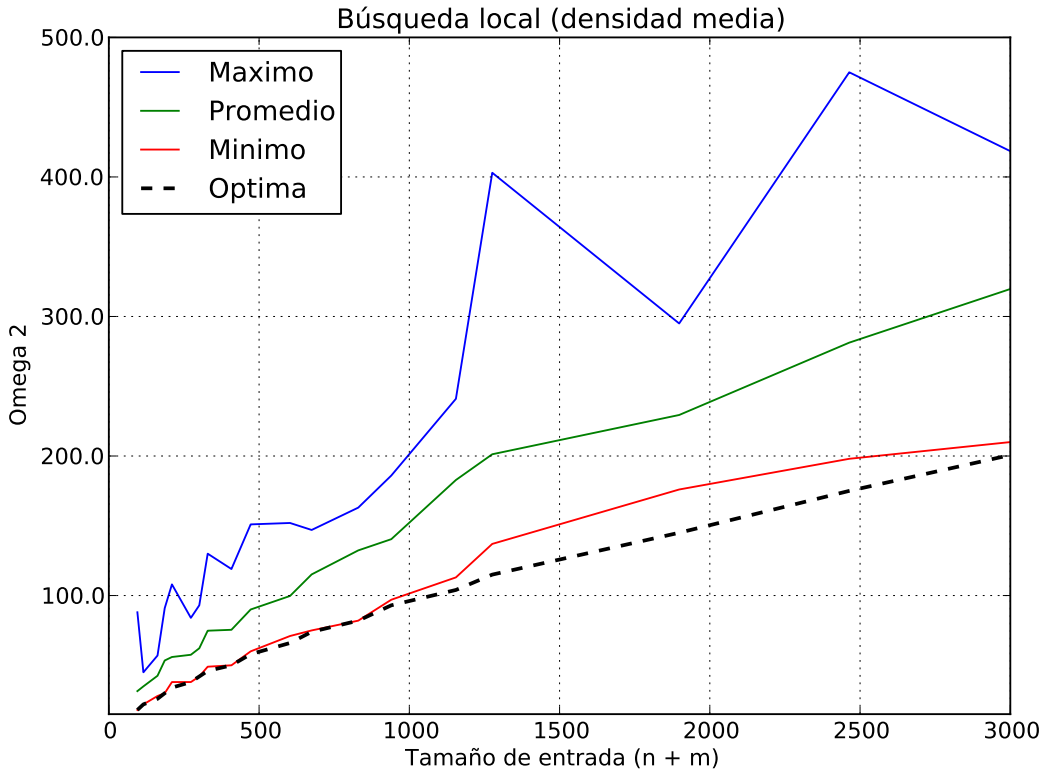
Se puede percibir el carácter no lineal de la complejidad. Sin embargo no se evidencia a luces claras la cota teórica de n^6 desarrollada previamente. Puede ser necesario un tamaño mayor para poder apreciar esa complejidad, o quizás la cota teórica no se alcanza en la práctica. Probablemente haya un factor que se amortice, habiendo escapado de nuestro escrutinio.

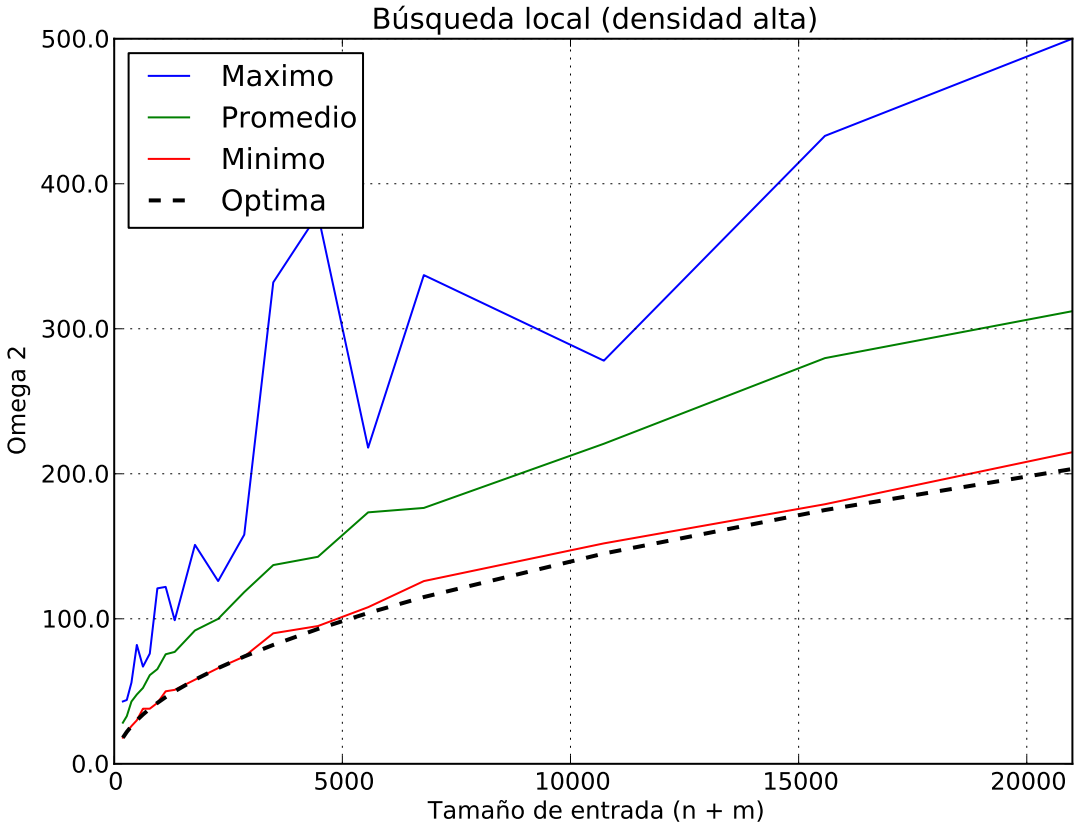
La conclusión que se obtuvo tras haber terminado todas las experimentaciones es que `local_search` hace generalmente pocas iteraciones, por lo que el tiempo de ejecución se asemeja al de `Greedy`, cuya complejidad teórica es $O(m * log(n)$.

Luego se propuso analizar la calidad de la solución encontrada. Experimentamos con distintas densidades de grafos. Primero con grafos densos, es decir con una cantidad de aristas del orden de n^2. Luego con grafos de densidad media, con una cantidad de aristas del orden de $\sqrt{n} \times n$. Finalmente probamos con grafos ralos, con una cantidad de aristas del orden de la cantidad de nodos. Para cada grupo de grafos, analizamos diferentes rangos de tamaños de entrada, con el objetivo de demostrar que el algoritmo mantiene ciertas propiedades, más allá del tamaño.

Se generan cien instancias para cada tamaño de entrada. En los siguientes gráficos se pueden observar comparados el mejor resultado, el peor resultado, y el resultado promedio para cada tamaño de entrada. Al ser instancias del tipo *mágico*, podemos saber el valor de la solución óptima - $n - 1$ - y graficarla.




Búsqueda local (densidad media)
Maximo
Promedio
Minimo
Optima
Omega 2
Tamaño de entrada (n + m)
500.0
400.0
300.0
200.0
100.0
0
500
1000
1500
2000
2500
3000


Búsqueda local (densidad alta)
Maximo
Promedio
Minimo
Optima
Omega 2
Tamaño de entrada (n + m)
500.0
400.0
300.0
200.0
100.0
0.0
0
5000
10000
15000
20000

En los tres gráficos se puede notar, a medida que aumenta el tamaño de la entrada, una mayor amplitud de resultados. Ésto tiene cierta intuición. Se generan muchas instancias para cada tamaño. A medida que aumenta la cantidad de aristas y vértices, aumenta la variedad entre las instancias generadas. Esto permite ver una gran diferencia de resultado entre instancias del mismo tamaño. La solución media, no obstante, parece preservar cierta distancia relativa, cierta escala con respecto a la solución óptima.

La mejor solución encontrada siempre está muy cerca de la solución óptima. Ésto se evidencia particularmente en el tercer gráfico, lo que atribuímos a la mayor densidad que le permite al algoritmo tener una vecindad amplia para recorrer. Sin embargo pareciera que la densidad también aumenta la amplitud y por lo tanto empeora la solución máxima. Puede tener sentido si se considera que al aumentar la cantidad de aristas que se deben generar cada una con su par de nodos y pesos, la diferencia entre dos instancias es combinatoria teniendo en cuenta todas las elecciones posbibles para cada arista.

A continuación se procede a estudiar la cantidad de iteraciones efectuadas por la búsqueda local, en relación a la cantidad de nodos del grafo.

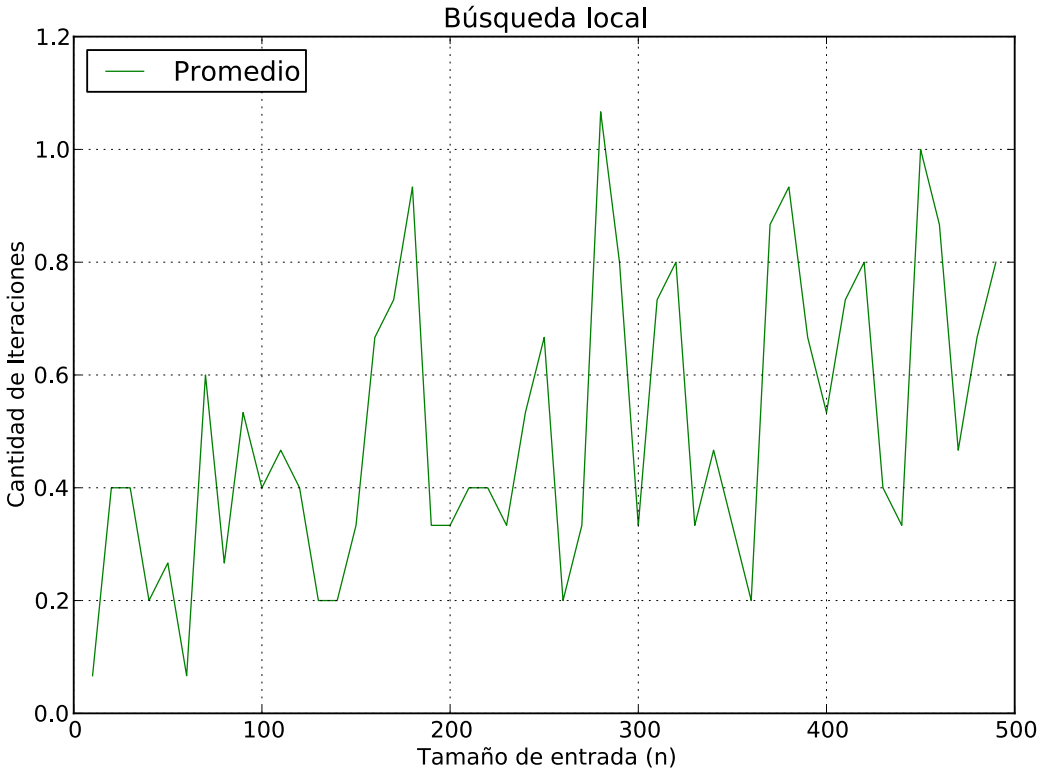


Figura 8: Cantidad de iteraciones en instancias *mágicas*, de densidad media

La cantidad de iteraciones realizadas presenta una gran amplitud, aún para tamaños de grafo similares. Tiene sentido si se considera que diferentes adyacencias de nodos, aún cuando la cantidad de nodos y aristas es la misma, puede permitir o no a la búsqueda local elegir una mejor solución vecina. La cantidad de iteraciones muestra

una tendencia lineal a crecer. Sin embargo se considera que es un resultado más bajo de lo esperado, ya parte importante de una heurística de búsqueda local es que la solución tenga movilidad suficiente y no quedarse estático cerca de la solución inicial. Se notó una gran cantidad de casos en los que la búsqueda local no pudo mejorar la solución inicial.

Nos propusimos mejorar la perspectiva de la búsqueda local implementada, analizando la cantidad de unidades de ω_2 que se disminuyen en cada iteración de la búsqueda local.

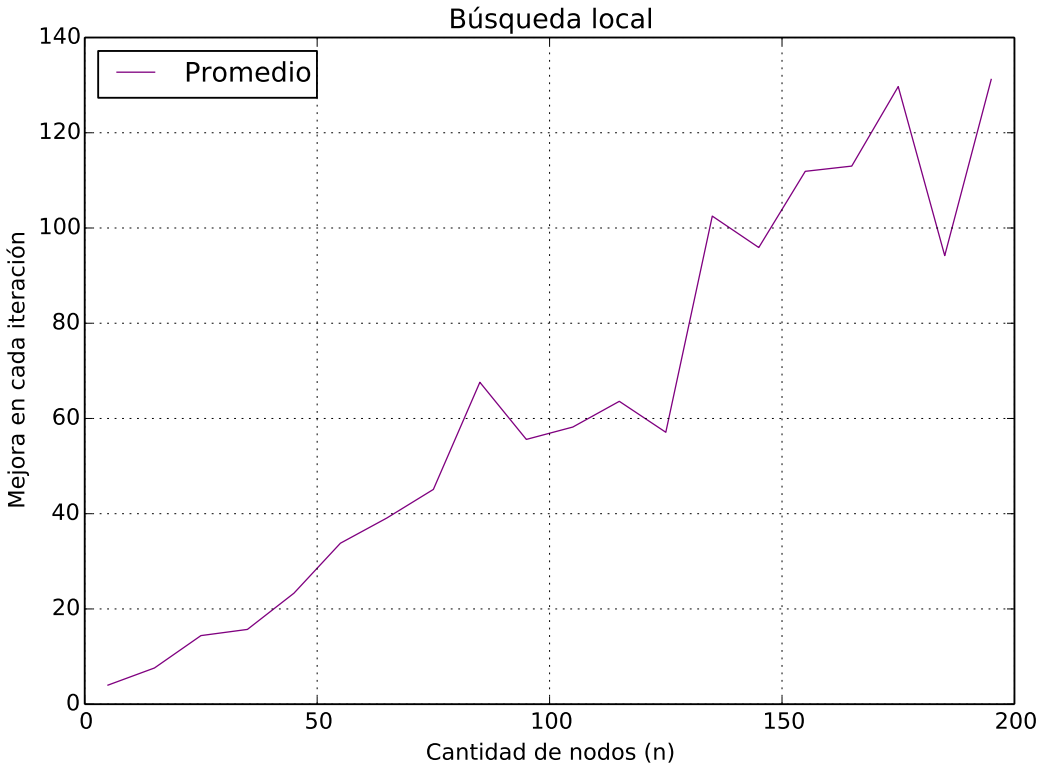


Figura 9: Mejora en cada iteración en instancias *mágicas* de densidad media

Los resultados fueron sorprendentes. Como quedo expuesto anteriormente, una corrida de búsqueda local no suele hacer muchas iteraciones, pero cuando las hace, disminuye notablemente el valor de ω_2 de la solución. Debe tenerse en cuenta que, siendo éstas instancias de tipo *mágico*, el valor de ω_2 de la solución óptima es $n-1$. De modo que lo que se ve en el gráfico es que el promedio de mejora en una iteración es del 60 % del valor de la solución óptima, proporción que se mantiene en las magnitudes de grafos consideradas.

6.4. GRASP

La metaheurística Grasp consiste en generar varias soluciones iniciales a partir de una heurística golosa aleatorizada y correr búsqueda local sobre ellas. La solución final es la mejor encontrada tras todas las búsquedas locales. Sea S el conjunto de soluciones iniciales, el algoritmo que se usa es el siguiente:

Mientras no se alcance el criterio de terminación:
Obtener $s \in S$ mediante una heurística golosa aleatorizada.
Mejorar s mediante búsqueda local.
Recordar la mejor solución obtenida hasta el momento.

6.4.1. Solución inicial

Primeramente utilizamos Dijkstra como nuestra heurística golosa, pero modificado para agregarle aleatoriedad. El factor aleatorio consistía en que en cada iteración de Dijkstra, en vez de tomar el nodo no visitado que minimiza la función objetivo, tomábamos uno de entre los $beta$[4] menores, al azar.

Nuestra intención era utilizar Dijkstra con la función de peso ω_1. De esta forma se intentó dar variedad a las soluciones iniciales, pero al mismo tiempo que éstas sean factibles - es decir, que cumplan con la restricción de $\omega_1 < K$ - en su mayoría.

Luego de diversas experimentaciones, nos dimos cuenta que realizar un Dijkstra con componente aleatorio no nos iba a servir a propósitos de nuestro algoritmo, ya que en cada se agregaba un vértice desde un conjunto reducido de candidatos, pero luego en los pasos subsiguientes los pesos de las aristas que conectaban a este vértice eran relajados por sus vértices aledaños. Tarde o temprano, finalmente, debido a que al elegir las **"buenas aristas"** se relajaban las **"malas aristas"** que habían sido elegido en pasos anteriores, se terminaban formando caminos comunes, es decir, un conjunto muy reducido de caminos que, pese a ser diferentes, se repetían constantemente y terminaban conformando un limitado **"conjunto de soluciones iniciales posibles"**.

Frente a esta problemática, decidimos continuar con nuestra idea, pero evitando relajar los pesos de las aristas en cuanto se encontraran caminos con menos peso. De esta forma, evitamos esta suerte de **"reducción de caminos iniciales posibles a un conjunto limitado elegido aleatoriamente"**. Como curiosidad: este nuevo enfoque era más bien digno de apodarse, por su forma de construir el camino inicial, **"Prim Randomizado"**.

Dada la aleatoriedad de la solución inicial, es posible que esta no sea factible. En caso de obtener una solución de este tipo, no ejecutamos el ciclo de búsqueda local, y pasamos a la siguiente iteración de GRASP.

[4]: Encontramos que $beta$=10 es un valor que presentaba suficiente aleatoriedad y resultados esperables para GRASP según mostramos en las experimentaciones.

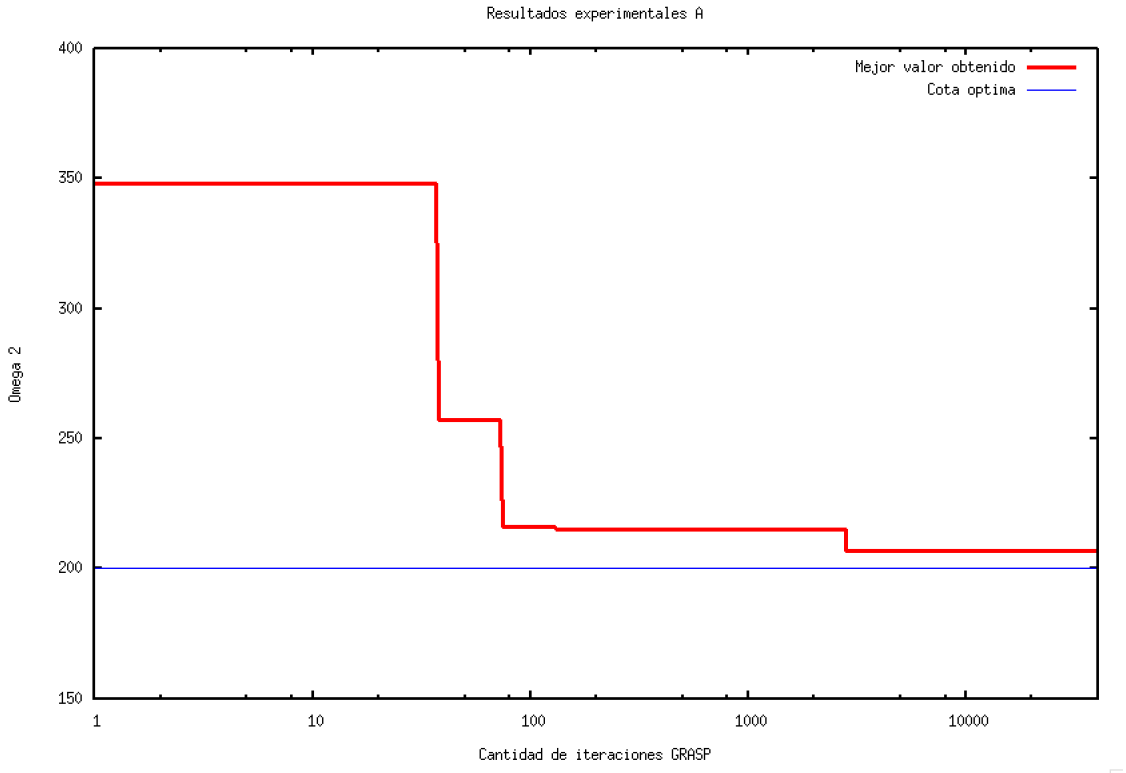
6.4.2. Criterio de terminación

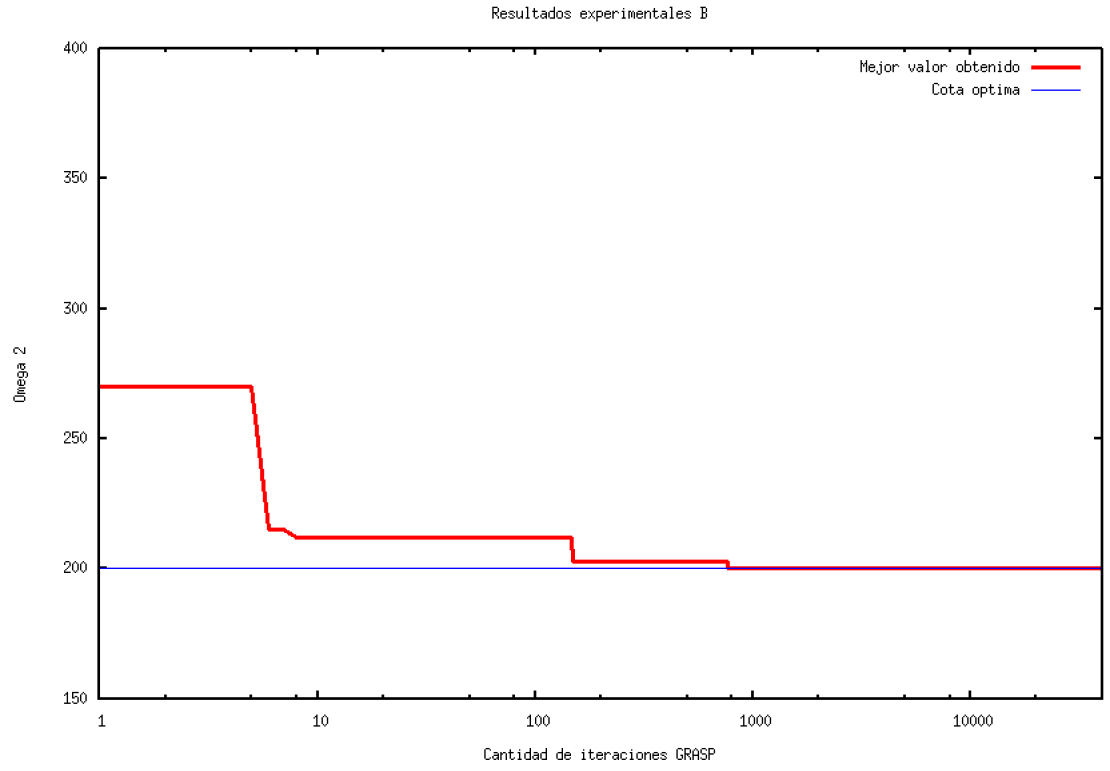
Usamos 3 criterios de terminación distintos al mismo tiempo. De alcanzarse alguno de los criterios, se termina la ejecución del algoritmo.

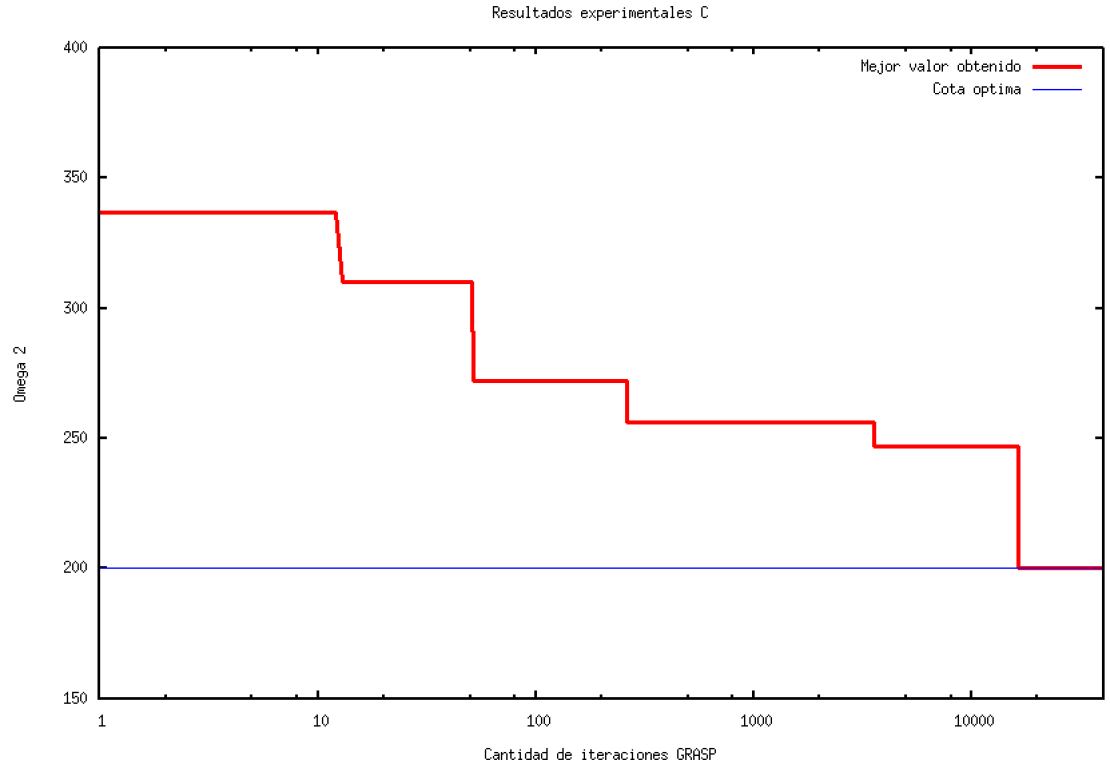
Los criterios que usamos son:

- Cantidad máxima de iteraciones.
- Cantidad máxima de iteraciones sin haber encontrado mejoras.
- Cantidad máxima de iteraciones sin haber encontrado una solución inicial factible.

Parametrizamos estos criterios usando n. Para elegir valores adecuados realizamos distintas experimentaciones sobre grafos de tipo *mágico* con baja, media y alta densidad. Los resultados obtenidos se pueden ver en los siguientes gráficos:







Las experimentaciones A, B y C se realizaron corriendo GRASP usando grafos de entrada de 200 nodos y 400, 4000 y 10000 ejes respectivamente. Cada una de las expe-

rimentaciones ejecuta 40000 iteraciones[5] en total, realizando en cada una un muestreo del valor de la mejor solución obtenida hasta el momento.

Cada figura muestra como evoluciona el valor de ω_2 a medida que aumentan las iteraciones. Además, fue graficado el camino óptimo, cuya deducción es posible gracias a las características de los grafos utilizados, en donde para el mismo el valor de ω_2 es igual a la cantidad de nodos del grafo, menos uno, es decir 199. De esta forma, pudimos comparar cuán buenos fueron los valores obtenidos.

Luego de tomar las muestras, dada la naturaleza de los datos obtenidos, decidimos utilizar escala logarítmica en el eje X para facilitar la visualización de los resultados, ya que como característica general notamos que se presenta una veloz *mejoría* en las soluciones durante las primeras iteraciones. Esta característica se encuentra fuertemente acentuada en los **experimentos A y B**, en los que se puede observar que durante las **primeras 1000 iteraciones**, es decir el primer $2,5\,\%$ del muestreo total, la **distancia a la solución óptima** realiza una disminución crítica, de 150 a 10 en el caso del **experimento A**, y de 70 a 0 en el caso del **experimento B**. El **experimento C**, también demuestra una notable mejoría en la distancia a la solución óptima, de 150 a 50, en el transcurso de las primeras 1000 iteraciones; la misma es, sin embargo, mucho más escalonada. La vertiginosa velocidad de acercamiento al valor óptimo (en comparación con la cantidad de iteraciones realizadas) justificó la utilización de esta escala.

Nuestro objetivo al momento de considerar los valores óptimos para los criterios de parada se basó en encontrar una forma de elegirlos de manera tal que los mismos se adecúen correctamente a grafos de variada densidad.

Creando grafos con una cantidad fija y representativa de nodos, y variando la densidad de aristas de los mismos, encontramos que para el caso de grafos de 200 vértices, el algoritmo puede iterar aproximadamente 10000 iteraciones **sin encontrar mejoras**. Este valor lo obtuvimos del anteúltimo salto de la **experimentación C**, y se corresponde con $n^2/4$.

La **máxima cantidad de iteraciones necesarias** para encontrar un **valor considerablemente cercano al óptimo** fue observada también en la **experimentación C**, siendo esta de alrededor de 20000 iteraciones, o $n^2/2$.

Para elegir un valor para la **cantidad máxima de iteraciones que pueden transcurrir sin haber encontrado una solución inicial factible**, tomamos el mismo valor que para la cantidad máxima de iteraciones sin haber encontrado mejoras, es sea $n^2/4$. Esto se fundamenta en que si no se encontró una solución inicial factible, tampoco se mejoró la última solución encontrada y por ende este criterio nos sirve para acotar el valor.

Además de encontrar valores que consideramos útiles y adecuados para los ya mencionados criterios, encontramos interesante la influencia de la densidad de los grafos en el comportamiento del algoritmo. Dada una entrada de baja densidad de aristas, es más difícil que el mismo encuentre una buena solución inicial, y más aún **mejorar**

[5]Se consideró que n^2 era una cantidad de iteraciones suficientemente grande como para realizar las pruebas. En caso de resultar insuficientes se realizaría una nueva prueba con una cantidad mayor.

una solución relativamente cercana al óptimo (en el muestreo concreto se puede observar que transcurren unas 39000 iteraciones, las últimas, en donde sólo se sucede una pequeña mejora de la solución). Dedujimos que esto sucede porque en la el grafo A hay pocos caminos (dada la baja densidad) y por consiguiente hay menos chances de encontrar los **mejores caminos**. Creemos que por esto sucede que no se logra llegar al óptimo aún luego de correr 40000 iteraciones, y por tal motivo creemos que llegado este punto es inútil seguir intentando encontrar un camino mejor.

El experimento que muestra el mejor comportamiento del algoritmo es el B, usando densidad media. Logra encontrar el camino óptimo mucho antes de llegar a nuestra cota para la cantidad de iteraciones: tarda alrededor de 1000 iteraciones (muy por debajo de las $20000(n^2/2)$ iteraciones que utilizaremos finalmente como cota máxima). Además, logra encontrar un camino inicial bastante mejor que las otras dos opciones($\omega_2 = 250$ frente a ω_2=300+ de A y C, es decir un 25 % mejor dado el óptimo de 200).

Por último, creemos que la experimentación C muestra un caso representativo de la gran filosofía GRASP, ya que se que mejora la solución escalonadamente hasta dar con la óptima, luego de varias mejoras parciales.

6.4.3. Búsqueda local

La búsqueda local que realizamos es la misma que en el apartado anterior.

6.4.4. Pseudocódigo

El algoritmo está implementado en la función `main`:

Algoritmo 9 *main*(int tipo_solucionInicial, Graph g, Nodo n1, Nodo n2)

```
1: crearMatrizCaminosMinimos(g)
2: int n = |nodes(g)|
3: int iteracionesSinMejorarCount = 0
4: int iteracionesSinMejorarMax = n
5: int iteracionesMax = n × log(n)
6: int iteracionesSinInitialPathCount = 0
7: int iteracionesSinInitialPathMax = n
8: mejorSolucion = NULL
9: para i=0; i<iteracionesMax; i++ hacer
10:     Solution solucion = obtenerSolucionInicial(tipo_solucionInicial, g, n1, n2)
11:     si ω1(solucion) > K entonces
12:         iteracionesSinInitialPathCount++
13:         si iteracionesSinInitialPathCount ≥ iteracionesSinInitialPathMax entonces break
14:         fin si
15:     fin si
16:     si ω1(solucion) ≤ K entonces
17:         mientras True hacer
18:             Solution nuevaSolucion = dameMejorVecino(solucion)
19:             si nuevaSolucion == NULL entonces
20:                 break
21:             fin si
22:             solucion = nuevaSolucion
23:         fin mientras
24:         si mejorSolucion == NULL entonces
25:             mejorSolucion = solucion
26:         sino si ω2(solucion) < ω2(mejorSolucion) entonces
27:             mejorSolucion = solucion
28:         sino
29:             iteracionSinMejorarCount++
30:         fin si
31:     fin si
32:     si iteracionesSinMejorarCount > iteracionesSinMejorarMax entonces
33:         break
34:     fin si
35: fin para
```

Algoritmo 10 *obtenerSolucionInicial*(int tipo, Graph g, Nodo n1, Nodo n2)

```
1: si tipo == Greedy_C entonces
2:     return resolverConDijkstraAleatorio(g, n1, n2, ObjectiveFunctionC)
3: fin si
4: return resolverConDijkstraAleatorio(g, n1, n2, ObjectiveFunctionA)
```

Las demás funciones tienen el mismo pseudocódigo que en Búsqueda local.

6.4.5. Complejidad

El algoritmo se ejecuta en un ciclo hasta cumplir con alguno de los criterios de terminación. Dado que el criterio de mayor valor es la cantidad de iteraciones totales($iteracionesMax$), tomamos ese valor como cota para calcular la complejidad. Cada iteración del ciclo es casi idéntica a la ejecución de búsqueda local. La única diferencia es cómo se obtiene la solución inicial. Para encontrar la solución inicial se usa resolverConDijkstraAleatorio. Esta función, en vez de tomar el nodo no visitado con ω_2 mínimo, toma uno entre los *beta* menores. Pero resulta que la complejidad no se altera con este cambio, ya que Dijkstra itera sobre todos los nodos de cualquier manera y lo único que cambia es el orden en que se toma el nodo no visitado.

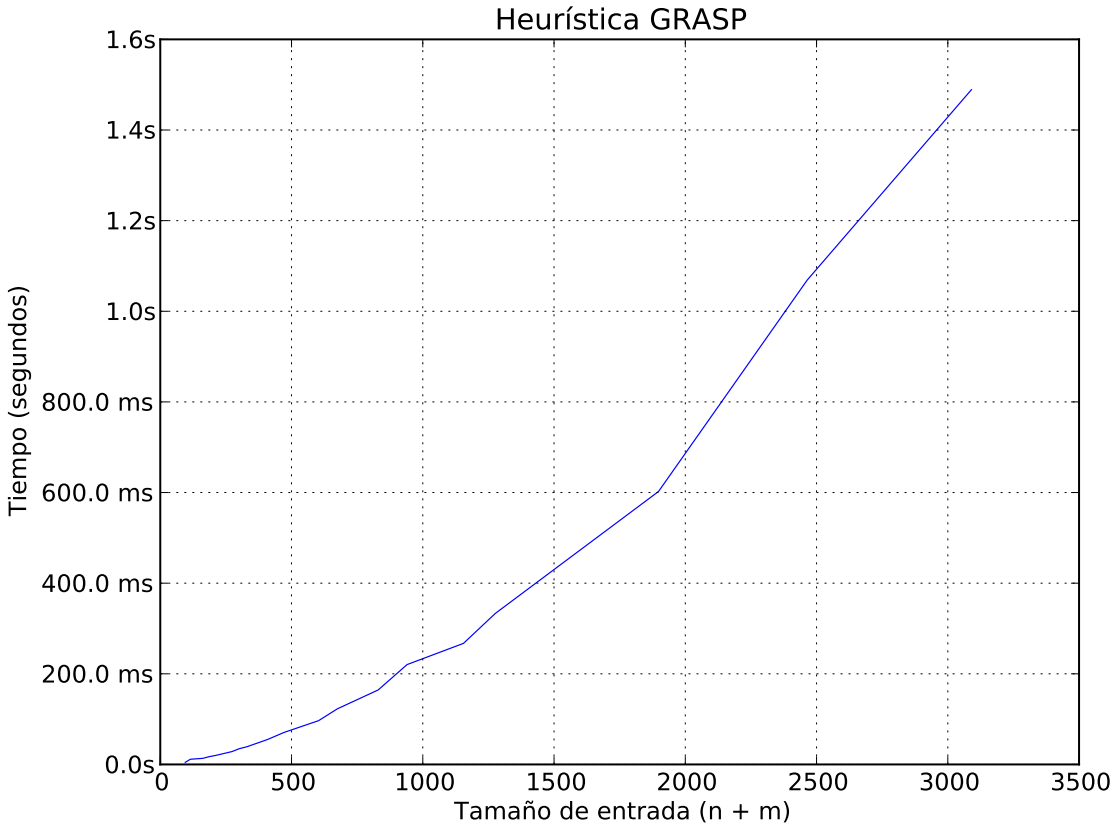
Vale hacer una aclaración que es que como se usa una cola con prioridad para los nodos no visitados en Dijkstra, para sacar uno entre los *beta* menores, se remueven los primeros *beta* nodos de la cola y luego se agregan todos menos uno que es con el que nos quedamos. Como remover y agregar de la cola con prioridad toma $O(log(n))$, entonces para remover y agregar los *beta* nodos se toma $O((beta + beta - 1) \times log(n)) = O(\beta * log(n))$. Nosotros tomamos β constante igual a 10.

Por lo tanto la complejidad de resolverConDijkstraAleatorio no resulta diferente resolverConDijkstra, usada en busqueda local y por ende la complejidad de cada iteración resulta igual a la complejidad de una ejecución de búsqueda local, o sea tiene complejidad $O(n^5)$.

Luego la complejidad total es $O(n^5 \times iteracionesMax) = O(n^5 \times n \times log(n)) = O(n^6 \times log(n))$.

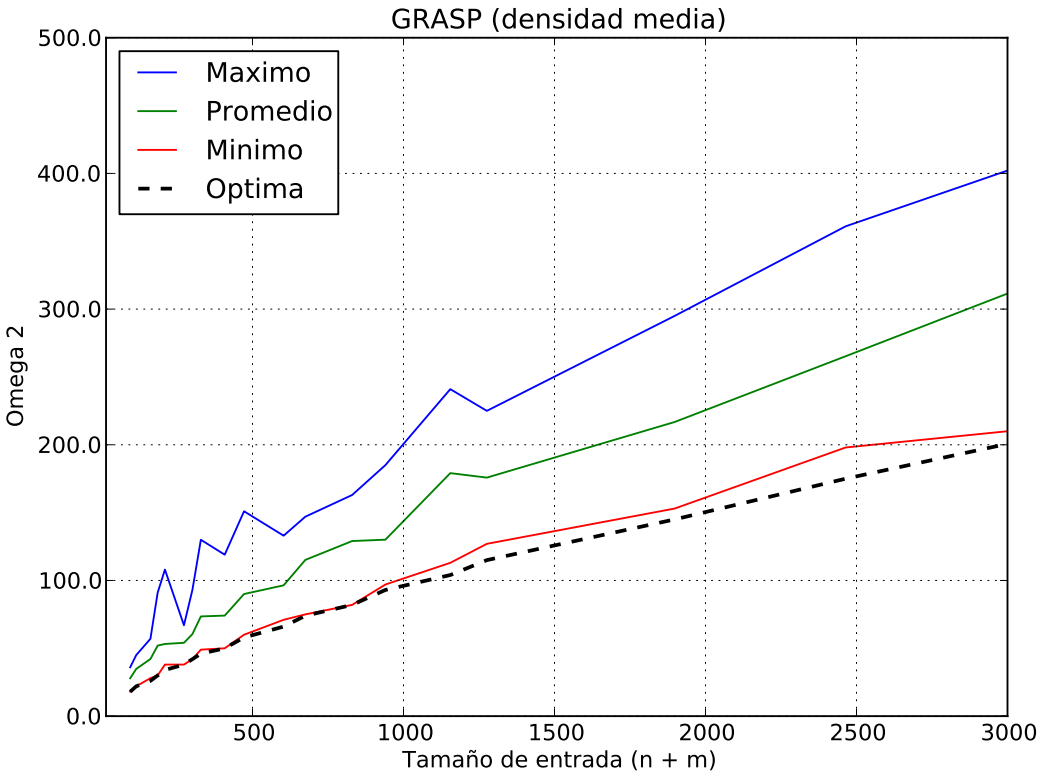
6.4.6. Experimentación

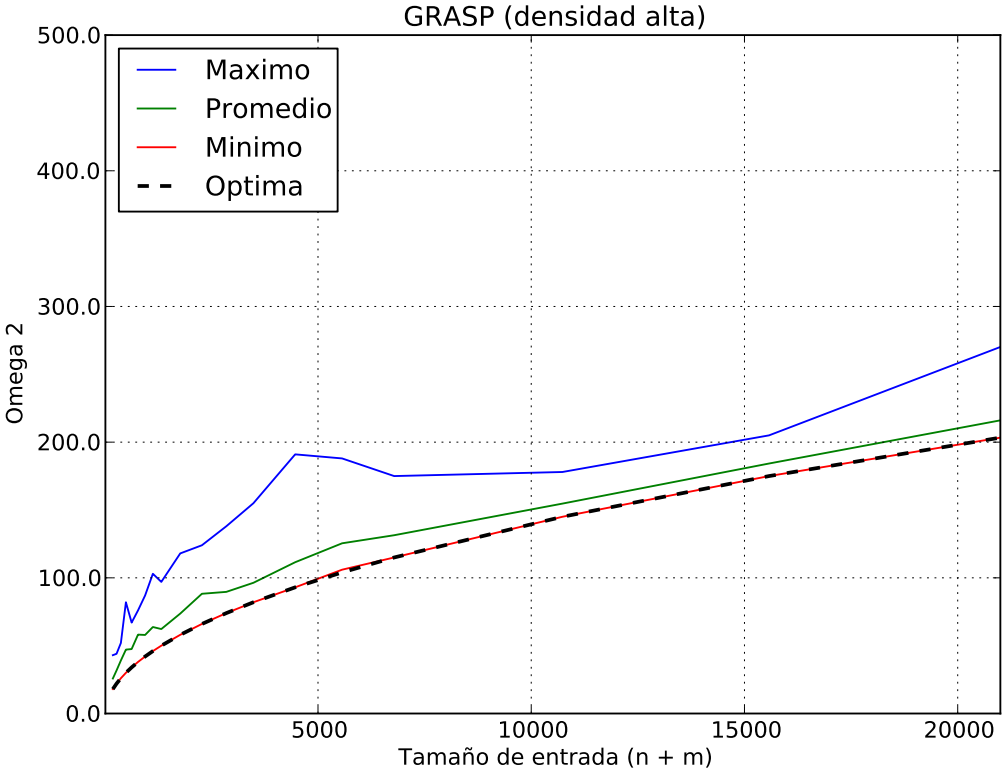
A continuación presentamos los resultados de la experimentación del tiempo de ejecución de la metaheurística GRASP. Se utilizaron instancias del tipo *mágico*, con una densidad media.



La curva que describe el tiempo de ejecución no evidencia un polinomio de grado 6, sin embargo, es evidente que tiene mucha más concavidad que la gráfica de la búsqueda local. Ésto proviene del hecho de repetir una cantidad que puede llegar a ser $n \times log(n)$ veces la búsqueda local.

Se prosigue la experimentación con la calidad de la solución.


GRASP (densidad media)
Maximo
Promedio
Minimo
Optima
Omega 2
Tamaño de entrada (n + m)
0.0
100.0
200.0
300.0
400.0
500.0
500
1000
1500
2000
2500
3000


GRASP (densidad alta)
Maximo
Promedio
Minimo
Optima
Omega 2
Tamaño de entrada (n + m)
0.0
100.0
200.0
300.0
400.0
500.0
5000
10000
15000
20000

La performance de GRASP nos llamó mucho la atención. Al igual que la búsqueda local, el resultado mínimo de nuestro algoritmo está muy cerca del óptimo. Pero esta vez hemos logrado reducir la amplitud entre nuestros resultados y de esta forma acercar todo el cuerpo de nuestras soluciones a la solución óptima. Los buenos resultados obtenidos se deben a la naturaleza de GRASP, que consiste en iterativamente correr una búsqueda local sobre múltiples soluciones iniciales generadas con un componente aleatorio. El repetir el experimento disminuye la varianza entre las diferentes corridas de GRASP y incrementa la posibilidad de una mejor solución final.

6.5. Comparación general

Tras estudiar ardúamente el comportamiento de los distintos algoritmos desarrollados en el trabajo presente, se procedió a evaluarlos comparativamente todos sobre un nuevo conjunto de instancias del tipo `mágico`, con las siguientes decisiones:

- Heurística golosa `Greedy` como unión de las heurísticas `GreedyA`, `GreedyB`, `GreedyC`
- Metaheurística GRASP con los siguientes parámetros:
  - Restricted Candidate List con $\beta = 10$
  - Condición de terminación con
    - Máx. cantidad de iteraciones sin encontras solución inicial factible = n^4
    - Máx. cantidad de iteraciones sin mejoras = n^4
    - Máx. cantidad de iteraciones totales = n^2

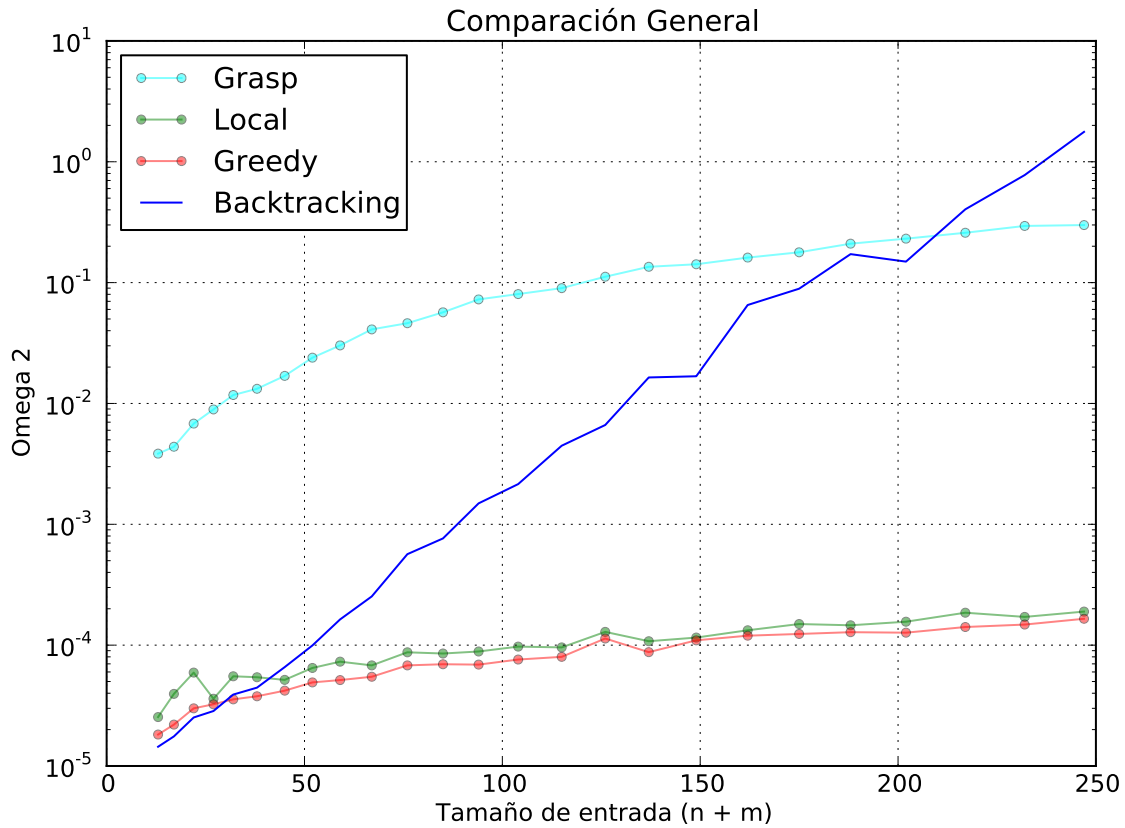


Figura 10: Tiempo de ejecución de los distintos algoritmos en nuevas instancias *mágicas* de densidad media

Primero se evaluó el tiempo de ejecución en entradas chicas en las que `Backtracking` es todavía razonable en cuanto al tiempo de ejecución. En el gráfico se utilizó una escala logarítmica para observar con detalle las cuatro curvas, ya que en su defecto los

tiempos de ejecución de `Greedy` y `Local` resultaban imperceptibles. Para tamaños de grafo menores a 50, `Grasp` presenta un tiempo de ejecución casi tes órdenes de magnitud por encima del resto de los algoritmos. Ésto se debe a la considerable cantidad de iteraciones que se le está permitido efectuar antes de retornar la mejor solución encontrada.

Por otra parte es notable la evolución del tiempo de ejecución de `Backtracking`. Comienza siendo el algoritmo más rápido, con una baja constante comparado con los otros algoritmos más sofisticados. Sin embargo, la cualidad factorial de su complejidad lo hace sobrepasar ampliamente a los otros algoritmos a medida que el tamaño del grafo aumenta.

Los tiempos de ejecución de `Greedy` y `local` presentan un desarrollo muy similar, manteniendo valores muy por debajo a los demás algoritmos. Especialmente en tamaños de grafo de tamaño reducido, `Local`, tras obtener la solución inicial mediante `Greedy`, no logra hacer muchas mejoras, por su tiempo de ejecución está muy ligeramente por encima del correspondiente a `Greedy`.

Luego se procedió a examinar la calidad de las soluciones obtenidas por las distintas heurísticas, siempre teniendo como parámetro la solución óptima encontrada por `backtrack`. Se utilizaron las mismas instancias generadas para estudiar los tiempos de ejecución en la sección anterior.

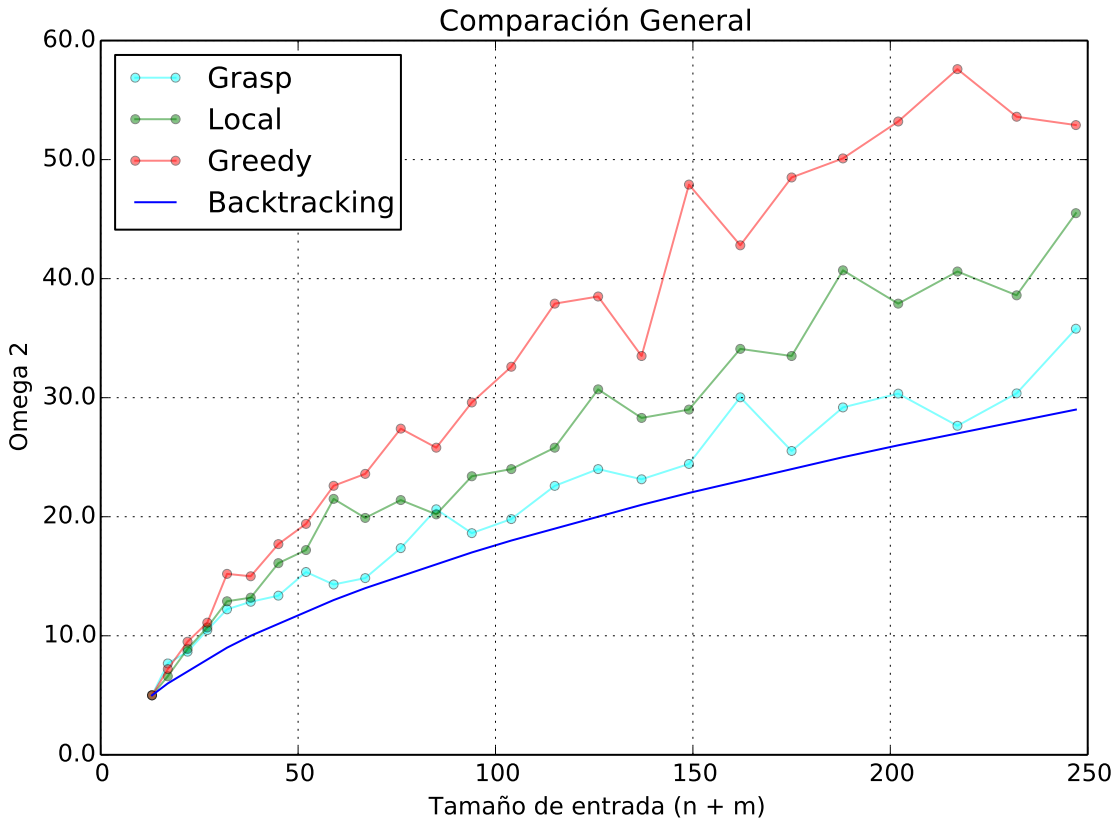


Figura 11: Calidad de solución de los distintos algoritmos en nuevas instancias *mágicas* de densidad media

La solución encontrada por `Greedy`, en éstos tamaños de entrada, parece tener un ω_2 acotado por el doble del ω_2 de la solución óptima. Sin embargo, por lo que se puede llegar a percibir en las instancias de este tamaño, cabe la posibilidad de que su ω_2 esté en realidad divergiendo del óptimo.

La solución encontrada por `Local` tiene un ω_2 claramente por debajo de la solución encontrada por `Greedy`. A pesar de tener un tiempo de ejecución muy poco por encima de `Greedy`, logra un solución mucho más cercana a la óptima. Parece mantener cierta distancia relativa a la solución encontrada por `backtracking`.

La performace de `Grasp` merece consideración. Como invariante frente a los distintos tamaños de grafo, devuelve una solución bastante cerca de la óptima. Aunque para estos tamaños de grafo, el *overhead* de la constante hace a `backtracking` quizás una alternativa más rápida.

[FIXME]lo siguiente iria como en una sección de conclusion

Asumimos trivialmente que el backtracking es el algoritmo que logra una mejor **calidad** al momento de obtener la solución, característica que sólo puede ser contrastada frente al inmenso tiempo que el mismo demora en correr. Resulta también interesante observar el comportamiento del **algoritmo goloso**, el cual presenta ejecuciones relativamente buenas, mientras que otras ejecuciones directamente no encuentran el algoritmo (se pasan del K); para poder representar este fenómeno se las representó por debajo del algoritmo exacto, de forma tal que se entiende que en las instancias en que el local arrojó un valor menor al backtracking, es porque la solución obtenida no era factible. Por otra parte, el tiempo de esta heurística es mínimo frente a todo el resto de los algoritmos, representando una buena opción para realizar un "acercamiento rápido" o "estimación" a la solución, ya sea obteniendo una válida o inválida. Aun así, también es necesario destacar que así como muchas de las instancias obtenidas son "buenas" desde cierto punto de vista, es notable el hecho de que muchas veces los resultados obtenidos son, aunque factibles, extremadamente malos en relación a la solución exacta. Esto último nos permite inferir que este tipo de heurísticas no nos permiten realizar una estimación "segura" sobre la factibilidad o calidad del resultado obtenido, siendo su resultado más bien una especie de "tiro de ruleta". Finalmente podemos observar a las heurísticas locales y GRASP, las cuales presentan soluciones en común, dado que GRASP es básicamente una "aplicación iterativa" de la local, combinada con un factor de aleatoriedad y ciertos criterios de terminación. Así, el resultado no nos sorprende al mostrarnos lo que de cierta forma ya esperabamos, la heurística de GRASP se comporta siempre "mejor o igual" que la local, es decir, existen instancias en que gracias a la aplicación de la aletoriedad en el goloso inicial, y debido a la iteració, se logra una mejoría notable de la calidad de la solución. Por otro lado, aunque GRASP presenta la desventaja de tener un costo tempral mayor a la local, luego de realizar un correcto **tuneo** de los criterios de finalización logramos establecer una buena relación tiempo/calidad.

7. Apéndices

7.1. Código Fuente (resumen)

7.1.1. Backtracking

Listing 1: backtracking.cpp

```
Timer timer( cerr );

int main() {
    int N;
    while (cin >> N && N) {
        BacktrackingHeuristic b;
        b.parseInput(N);
        timer.setInitialTime( "todo_el_codigo" );

        if (b.U != b.V) {
            b.initialize();
            vector<Node> adjacent = b.G->getAdjacent(b.U); // se
                devuelve por referencia
            for (int i = 0; i < adjacent.size(); i++) {
                Node n = adjacent[i];
                Edge *f = b.G->getEdge(b.U, n);
                b.backtrack(f);
            }
        }

        timer.setFinalTime( "todo_el_codigo" );
        timer.saveAllTimes();
        b.printSolution();
    }
    return 0;
}
```

Listing 2: BacktrackingHeuristic::parseInput()

```
void BacktrackingHeuristic::parseInput(int N) {
    this->N = N;
    cin >> this->M >> this->U >> this->V >> this->K;
    this->G = new Graph(N);
    this->visited = vector<bool>(N, false);
    int v1, v2;
    double w1, w2;
    for(int i = 0; i < M; i++) {
        cin >> v1 >> v2 >> w1 >> w2;
        this->G->addEdge(v1, v2, w1, w2);
    }
}
```

Listing 3: BacktrackingHeuristic::initialize()

```
void BacktrackingHeuristic::initialize() {
```

```
2       this->currentBranch.path.push_back(this->U);
3       this->bestSolutionFound.totalOmega2 = INFINITE;
4
5       DijkstraSolution byOmega1(N, V);
6       DijkstraSolution byOmega2(N, V);
7       Dijkstra<ObjectiveFunctionA> dijsktra1;
8       Dijkstra<ObjectiveFunctionB> dijsktra2;
9       dijsktra1.findPath(this->G, &byOmega1);
10      dijsktra2.findPath(this->G, &byOmega2);
11      this->distancesOmega1 = byOmega1.distances;
12      this->distancesOmega2 = byOmega2.distances;
13  }
```

Listing 4: BacktrackingHeuristic::backtrack()

```
1   void BacktrackingHeuristic::backtrack(Edge *e) {
2       Node toNode = e->toNode;
3       this->currentBranch.path.push_back(toNode);
4       this->currentBranch.totalOmega1 += e->omega1;
5       this->currentBranch.totalOmega2 += e->omega2;
6       this->visited[toNode] = true;
7
8       bool podar = ((this->currentBranch.totalOmega1 + this->
            distancesOmega1[toNode]) > this->K) ||
9           ((this->currentBranch.totalOmega2 + this->distancesOmega2[
                toNode]) >= this->bestSolutionFound.totalOmega2);
10
11      if (!podar) {
12          if (toNode == this->V) {
13              this->bestSolutionFound = currentBranch;
14          } else {
15              vector<Node> adjacent = G->getAdjacent(toNode); // se
                    devuelve por referencia
16              for (int i = 0; i < adjacent.size(); i++) {
17                  Node n = adjacent[i];
18                  if (!visited[n]) {
19                      Edge *f = this->G->getEdge(toNode, n);
20                      backtrack(f);
21                  }
22              }
23          }
24      }
25
26      this->currentBranch.path.pop_back();
27      this->currentBranch.totalOmega1 -= e->omega1;
28      this->currentBranch.totalOmega2 -= e->omega2;
29      this->visited[toNode] = false;
30  }
```

Listing 5: BacktrackingHeuristic::printSolution()

```
1   void BacktrackingHeuristic::printSolution() {
2       Solution *s = &(this->bestSolutionFound);
3       if (s->totalOmega2 == INFINITE) {
4           cout << "no" << endl;
5           return;
```

```
6       }
7
8       cout << s->totalOmega1 << " " << s->totalOmega2 << " " << (s->path
            .size()+1);
9       for (int i = 0; i < s->path.size(); i++)
10          cout << " " << s->path[i];
11      cout << endl;
12      return;
13  }
```

7.1.2. Greedy

Listing 6: greedy_heuristic_All.cpp

```
1  #include <iostream>
2  #include "../common/Timer.h"
3  #include "GreedyHeuristicAll.h"
4
5  using namespace std;
6
7  int main(int argc, char* argv[])
8  {
9    Timer timer(cerr);
10   GreedyHeuristicAll greedy(&timer);
11   greedy.run();
12
13     return 0;
14 }
```

Listing 7: GreedyHeuristicAll.cpp

```
1  #include "GreedyHeuristicAll.h"
2  #include "../common/Dijkstra.h"
3  #include "../common/ObjectiveFunctions.h"
4
5  GreedyHeuristicAll::GreedyHeuristicAll()
6  {
7    solution = new Solution();
8  }
9
10 GreedyHeuristicAll::GreedyHeuristicAll(Timer* t)
11 {
12   solution = new Solution();
13   timer = t;
14 }
```

Listing 8: createSolutionA()

```
1  Solution* createSolutionA(ProblemInstance* instance)
2  {
3      // creo el dijkstra
4      Dijkstra<ObjectiveFunctionA> dijsktra;
5      // creo la solucion
6      DijkstraSolution dijkstraSolution( instance->graph->nodeCount,
           instance->u);
```

```
7       // cargo en la solucion, todos los paths del dijkstra desde el
            nodo inicial
8       dijsktra.findPath( instance->graph, &dijkstraSolution );
9       // obtengo el path que me interesa
10      Solution* solution = new Solution();
11      dijkstraSolution.getPath( instance->v, instance->graph, solution->
            path, solution->totalOmega1, solution->totalOmega2 );
12      return solution;
13  }
```

Listing 9: createSolutionB()

```
1   Solution* createSolutionB(ProblemInstance* instance)
2   {
3       // creo el dijkstra
4       Dijkstra<ObjectiveFunctionB> dijsktra;
5       // creo la solucion
6       DijkstraSolution dijkstraSolution( instance->graph->nodeCount,
            instance->u);
7       // cargo en la solucion, todos los paths del dijkstra desde el
            nodo inicial
8       dijsktra.findPath( instance->graph, &dijkstraSolution );
9       // obtengo el path que me interesa
10      Solution* solution = new Solution();
11      dijkstraSolution.getPath( instance->v, instance->graph, solution->
            path, solution->totalOmega1, solution->totalOmega2 );
12      return solution;
13  }
```

Listing 10: createSolutionC()

```
1   Solution* createSolutionC(ProblemInstance* instance)
2   {
3       // creo el dijkstra
4       Dijkstra<ObjectiveFunctionC> dijsktra;
5       // creo la solucion
6       DijkstraSolution dijkstraSolution( instance->graph->nodeCount,
            instance->u);
7       // cargo en la solucion, todos los paths del dijkstra desde el
            nodo inicial
8       dijsktra.findPath( instance->graph, &dijkstraSolution );
9       // obtengo el path que me interesa
10      Solution* solution = new Solution();
11      dijkstraSolution.getPath( instance->v, instance->graph, solution->
            path, solution->totalOmega1, solution->totalOmega2 );
12      return solution;
13  }
```

Listing 11: getBestSolution()

```
1   Solution* getBestSolution(ProblemInstance* instance)
2   {
3       Solution* solutionB = createSolutionB(instance);
4       if(solutionB->totalOmega1 <= instance->K) {
5           return solutionB;
6       }
```

```
7       // si la solucion no es factible probamos con otras funciones
            objetivo
8       Solution* solutionA = createSolutionA(instance);
9       Solution* solutionC = createSolutionC(instance);
10      if(solutionC->totalOmega1 <= instance->K) {
11          if(solutionA->totalOmega2 < solutionC->totalOmega2) {
12              return solutionA;
13          }
14          return solutionC;
15      }
16
17      if(solutionA->totalOmega1 <= instance->K) {
18          return solutionA;
19      }
20      return NULL;
21  }
```

Listing 12: GreedyHeuristicAll::resolveInstance()

```
1   void GreedyHeuristicAll::resolveInstance( ProblemInstance* instance ){
2     solution = getBestSolution(instance);
3   }
```

Listing 13: GreedyHeuristicAll::run()

```
1   void GreedyHeuristicAll::run()
2   {
3     Parser parser;
4     parser.parseInput();
5
6     for ( int i = 0; i < parser.problemInstances.size(); i++ )
7     {
8       ProblemInstance* instance = parser.problemInstances[i];
9
10      timer->setInitialTime("todo_el_codigo");
11      resolveInstance( instance );
12      timer->setFinalTime("todo_el_codigo");
13      timer->saveAllTimes();
14
15      if(!solution) {
16        cout << "no" << endl;
17      } else{
18        solution->print();
19      }
20    }
21  }
```

7.1.3. Local Search

Listing 14: local_search.cpp

```
1   #include "../common/Graph.h"
2   #include "../common/Parser.h"
3   #include "../common/DijkstraSolution.h"
4   #include "../common/Dijkstra.h"
```

```
5  #include "../common/GreedyHeuristic.h"
6  #include "../common/Solution.h"
7  #include "../common/Parser.h"
8  #include "../common/Timer.h"
9  #include "NeighbourhoodSelectorA.h"
10 #include "InitialSolution.h"
11
12 Parser parser;
13 Timer timer( cerr );
14
15
16 int main( int argc, char const* argv[] )
17 {
18   /*****************
19     Initialization
20   ******************/
21   // instantiate the initial solution using the initial solution
        parameter
22   InitialSolution* initialSolution = new InitialSolution();
23   // instantiate the neighborhood selector using the neighborhood
        selector parameter
24   NeighbourhoodSelector* selector = new NeighbourhoodSelectorA();
25   // parse the input
26   parser.parseInput();
27
28   /************************
29     Iterate over instances
30   ******************'*******/
31   for(auto instance:parser.problemInstances)
32   {
33     /*************
34       Resolution
35     **************/
36     selector->initialize(instance);
37
38     // obtain the initial time
39     timer.setInitialTime( "todo_el_codigo" );
40     // obtain the initial solution
41
42     int initialSolutionOmega2;
43
44     Solution* solution = initialSolution->getInitialSolution( instance
           );
45
46     // El dijkstra de omega1 debe cumplir con el K, sino no tiene
          sentido correr la heuristica
47     // si solution no es valida entonces, entonces es NULL
48     if(solution != NULL) {
49       initialSolutionOmega2 = solution->totalOmega2;
50
51       // run the heuristic
52       Solution* newSolution = NULL;
53       bool huboMejora = false;
54       do
55       {
56         newSolution = selector->getBestNeighbour( solution );
57         // Si no logro mejorar la solucion, termino
58         if(newSolution != NULL) {
```

```
59            delete solution;
60            solution = newSolution;
61            huboMejora = true;
62          } else {
63            huboMejora = false;
64          }
65        } while(huboMejora);
66      }
67
68      // obtain the final time
69      timer.setFinalTime( "todo_el_codigo" );
70
71      /***************
72        Output Print
73      ***************/
74      // print the solution
75      if(solution) {
76        solution->print();
77        delete solution;
78      }
79      else
80      {
81        cout << 0 << endl;
82      }
83
84      // save all obtained times to output
85      timer.saveAllTimes();
86    }
87    return 0;
88  }
```

Listing 15: createSolutionA()

```
1  Solution* createSolutionA(ProblemInstance* instance)
2  {
3      // creo el dijkstra
4      Dijkstra<ObjectiveFunctionA> dijsktra;
5      // creo la solucion
6      DijkstraSolution dijkstraSolution( instance->graph->nodeCount,
           instance->u);
7      // cargo en la solucion, todos los paths del dijkstra desde el
           nodo inicial
8      dijsktra.findPath( instance->graph, &dijkstraSolution );
9      // obtengo el path que me interesa
10     Solution* solution = new Solution();
11     dijkstraSolution.getPath( instance->v, instance->graph, solution->
           path, solution->totalOmega1, solution->totalOmega2 );
12     return solution;
13 }
```

Listing 16: createSolutionB()

```
1  Solution* createSolutionB(ProblemInstance* instance)
2  {
3      // creo el dijkstra
4      Dijkstra<ObjectiveFunctionB> dijsktra;
```

```
5       // creo la solucion
6       DijkstraSolution dijkstraSolution( instance->graph->nodeCount,
            instance->u);
7       // cargo en la solucion, todos los paths del dijkstra desde el
            nodo inicial
8       dijsktra.findPath( instance->graph, &dijkstraSolution );
9       // obtengo el path que me interesa
10      Solution* solution = new Solution();
11      dijkstraSolution.getPath( instance->v, instance->graph, solution->
            path, solution->totalOmega1, solution->totalOmega2 );
12      return solution;
13  }
```

Listing 17: createSolutionC()

```
1   Solution* createSolutionC(ProblemInstance* instance)
2   {
3       // creo el dijkstra
4       Dijkstra<ObjectiveFunctionC> dijsktra;
5       // creo la solucion
6       DijkstraSolution dijkstraSolution( instance->graph->nodeCount,
            instance->u);
7       // cargo en la solucion, todos los paths del dijkstra desde el
            nodo inicial
8       dijsktra.findPath( instance->graph, &dijkstraSolution );
9       // obtengo el path que me interesa
10      Solution* solution = new Solution();
11      dijkstraSolution.getPath( instance->v, instance->graph, solution->
            path, solution->totalOmega1, solution->totalOmega2 );
12      return solution;
13  }
```

Listing 18: InitialSolution::getInitialSolution()

```
1   Solution* InitialSolution::getInitialSolution(ProblemInstance*
        instance)
2   {
3       Solution* solutionB = createSolutionB(instance);
4       if(solutionB->totalOmega1 <= instance->K) {
5           return solutionB;
6       }
7       // si la solucion no es factible probamos con otras funciones
            objetivo
8       Solution* solutionA = createSolutionA(instance);
9       Solution* solutionC = createSolutionC(instance);
10      if(solutionC->totalOmega1 <= instance->K) {
11          if(solutionA->totalOmega2 < solutionC->totalOmega2) {
12              return solutionA;
13          }
14          return solutionC;
15      }
16      if(solutionA->totalOmega1 <= instance->K) {
17          return solutionA;
18      }
19      return NULL;
20  }
```

Listing 19: NeighbourhoodSelectorA::removeCycles()

```
Solution* NeighbourhoodSelectorA::removeCycles(Solution* solution)
{
  int* nextNodes = new int[nodeCount];
  for(int i=0; i<nodeCount; i++) {
    nextNodes[i] = 0;
  }
  for(unsigned int i=0; i<solution->path.size(); i++) {
    Edge* edge = solution->path[i];
    nextNodes[edge->fromNode] = edge->toNode;
  }

  vector<int> newPath;
  int firstNode = solution->path[0]->fromNode;
  int lastNode = solution->path[solution->path.size()-1]->toNode;
  int next = firstNode;
  newPath.push_back(firstNode);
  int a = 0;
  while(nextNodes[next] != lastNode)
  {
    next = nextNodes[next];
    newPath.push_back(next);
  }

  newPath.push_back(lastNode);
  //cout << lastNode << endl;

  Solution* newSolution = new Solution();
  for(int i=0; i<newPath.size()-1; i++){
    Edge* edge = solution->getEdgeBetween(newPath[i], newPath[i+1]);
    Edge* newEdge = new Edge(edge->fromNode, edge->toNode, edge->
        omega1, edge->omega2);
    newSolution->path.push_back(newEdge);
    newSolution->totalOmega1 += newEdge->omega1;
    newSolution->totalOmega2 += newEdge->omega2;
  }

  return newSolution;
}
```

Listing 20: createNewSolutionReplacingPath()

```
Solution* createNewSolutionReplacingPath(const Solution* orig, const
    Solution* sub) {
  int node1 = sub->path[0]->fromNode;
  int node2 = sub->path[sub->path.size()-1]->toNode;
  Solution* res = new Solution();
  vector<Edge*> edgesToRemove;
  bool subPathStartsAtNode1 = false;
  int subPathStartsAtEdgeIndex = 0;
  // primero agrego todos los edges del path original hasta encontrar
      alguno de los nodos del subpath
  for(int i=0; i<orig->path.size(); i++) {
    Edge* edge = orig->path[i];
    if(edge->fromNode == node1 || edge->fromNode == node2) {
      subPathStartsAtNode1 = edge->fromNode == node1;
```

```
13          subPathStartsAtEdgeIndex = i;
14          break;
15        } else {
16          res->path.push_back(edge);
17          res->totalOmega1 += edge->omega1;
18          res->totalOmega2 += edge->omega2;
19        }
20      }
21
22      // agrego todos los ejes del sub path
23      for(int i=0; i<sub->path.size(); i++) {
24        Edge* edge = sub->path[i];
25        res->path.push_back(edge);
26        res->totalOmega1 += edge->omega1;
27        res->totalOmega2 += edge->omega2;
28      }
29
30      // busco el nodo desde donde continua el pedazo del path original
31      // agrego todos los ejes del path original a partir de ahi
32      int fromNode = subPathStartsAtNode1 ? node2 : node1;
33      bool addEdges = false;
34      for(int i=subPathStartsAtEdgeIndex+1; i<orig->path.size(); i++) {
35        Edge* edge = orig->path[i];
36        if(edge->fromNode == fromNode) {
37          addEdges = true; // encontre el nodo, asi que comienzo a aniadir
                los nodos desde aca
38        }
39        if(addEdges) {
40          res->path.push_back(edge);
41          res->totalOmega1 += edge->omega1;
42          res->totalOmega2 += edge->omega2;
43        }
44      }
45
46      return res;
47    }
```

Listing 21: NeighbourhoodSelectorA::getBestNeighbour()

```
1   Solution* NeighbourhoodSelectorA::getBestNeighbour(const Solution*
        origSolution)
2   {
3     // Itero sobre la matriz de soluciones
4     // Por cada par de nodos, tengo que buscar si existe un path en la
          matriz,
5     // tal que sustituyendolo pot path entre el par de nodos de la
          solucion original
6     // se obtenga una nueva solucion tal que se su totalOmega2 sea menor
7
8     // Nota: Habiamos pensado en usar DeltaOmega2, pero si K es muy
          grande,
9     // nos sobraria mucho K que podriamos haber usado para sustituir en
          cada paso
10    // por un path con minimo omega2
11
12    // El path de una solution es un vector de ejes
```

```
13    // En cada iteracion que encuentro una solucion mejor, voy a cambiar algunos de los nodos,
14    // por lo que tengo que recalcular los nodos de la mejor solucion
15    int edgesCount = origSolution->path.size();
16    int nodesCount = edgesCount+1;
17    vector<int> nodes(nodesCount);
18    for(int i=0; i<edgesCount; i++) {
19      Edge* edge = origSolution->path[i];
20      nodes[i] = edge->fromNode;
21    }
22    nodes[nodesCount-1] = origSolution->path[edgesCount-1]->toNode; // ultimo nodo del path
23
24    Solution* bestSolution = NULL;
25    for(int i=0; i<nodes.size() - 1; i++) {
26      for(int j=i+1; j<nodes.size(); j++) {
27        Solution* subSolution = origSolution->createSubSolutionBetween(nodes[i], nodes[j]); // solution con sub path entre los nodos
28        Solution* solution_ij = getSolvedPathBetween(nodes[i], nodes[j]); // dijkstra por omega2 entre los nodos
29        if(solution_ij == NULL) {
30          continue;
31        }
32
33        // Si el path creado con dijkstra usando omega2, tiene menos omega2 total, que el path actual entre
34        // los nodos i y j, entonces chequeo si al crear una nueva solucion tendra menos omega2 que la mejor solucion.
35        // En ese caso me guardo esta nueva solucion como la mejor hasta ahora.
36        // Ademas chequeo que se cumpla con el K requerido.
37        double bestSolutionOmega2 = bestSolution == NULL ? origSolution->totalOmega2 : bestSolution->totalOmega2;
38        double newSolutionOmega2 = origSolution->totalOmega2 - subSolution->totalOmega2 + solution_ij->totalOmega2;
39        double newSolutionOmega1 = origSolution->totalOmega1 - subSolution->totalOmega1 + solution_ij->totalOmega1;
40        if(newSolutionOmega2 < bestSolutionOmega2 && newSolutionOmega1 <= K) {
41          if(bestSolution != NULL) {
42            delete bestSolution;
43          }
44          // creo la nueva mejor solucion
45          bestSolution = createNewSolutionReplacingPath(origSolution, solution_ij);
46          Solution* bestSolutionWithoutCycles = removeCycles(bestSolution);
47          delete bestSolution;
48          bestSolution = bestSolutionWithoutCycles;
49        }
50        delete subSolution;
51      }
52    }
53
54    return bestSolution;
55  }
```

Listing 22: NeighbourhoodSelector::initialize()

```
1  void NeighbourhoodSelector::initialize(ProblemInstance* instance)
2  {
3    deleteMatrix();
4    nodeCount = instance->graph->nodeCount;
5    pathMatrix = new Solution*[nodeCount*nodeCount];
6    K = instance->K;
7
8    // Como los paths entre j y k son iguales a los paths entre k y j,
         pero al reves
9    // primero encuentro los paths entre j y k y luego los doy vuelta
10   for(int j=1; j<=nodeCount; j++) {
11     Dijkstra<ObjectiveFunctionOmega2> dijsktra;
12     DijkstraSolution dijkstraSolution( instance->graph->nodeCount, j )
           ;
13     dijsktra.findPath( instance->graph, &dijkstraSolution );
14     for(int k=1; k<=nodeCount; k++) {
15       if(j==k) continue;
16         Solution* solution = new Solution();
17         dijkstraSolution.getPath(k, instance->graph, solution->path,
               solution->totalOmega1, solution->totalOmega2);
18         // si no hay ninun path entre los nodos j y k entonces, cargo
               NULL en la matrix
19         pathMatrix[j-1 + (k-1) * nodeCount] = solution->path.size() >
               0 ? solution : NULL;
20     }
21   }
22 }
```

Listing 23: NeighbourhoodSelector::getSolvedPathBetween()

```
1  Solution* NeighbourhoodSelector::getSolvedPathBetween(int node1, int
       node2) {
2    return pathMatrix[node1-1 + (node2-1) * nodeCount];
3  }
```

7.1.4. Grasp

Listing 24: grasp.cpp

```
1  #include "../common/Graph.h"
2  #include "../common/Parser.h"
3  #include "../common/DijkstraSolution.h"
4  #include "../common/Dijkstra.h"
5  #include "../common/DijkstraRandomized.h"
6  #include "../common/GreedyHeuristic.h"
7  #include "../common/Solution.h"
8  #include "../common/Parser.h"
9  #include "../common/Timer.h"
10 #include "NeighbourhoodSelectorA.h"
11 #include "InitialSolution.h"
12 #include <math.h>
13
14 Parser parser;
```

```
15 Timer timer( cerr );
16
17
18 int main( int argc, char const* argv[] )
19 {
20     /* Semilla random */
21     srand (time(NULL));
22
23     /*****************
24       Initialization
25     *****************/
26
27     // parse the input
28     parser.parseInput();
29
30     // obtain the initial time
31     timer.setInitialTime( "todo_el_codigo" );
32     // obtain the initial solution
33
34     /************************
35       Iterate over instances
36     ************************/
37     for(auto instance:parser.problemInstances)
38     {
39         /*************
40          Resolution
41         *************/
42         NeighbourhoodSelector* selector = new NeighbourhoodSelectorA()
               ;
43         selector->initialize(instance);
44
45         // valores arbitrarios basados en n para criterio de
               terminaciones
46         int n = instance->graph->nodeCount;
47         int iteracionesSinMejorarCount = 0;
48         int iteracionesSinMejorarMax = n*n/4;
49         int iteracionesMax = n*n/2;
50         int iteracionesSinInitialPathCount = 0;
51         int iteracionesSinInitialPathMax = n*n/4;
52
53         Solution* bestSolution = NULL;
54
55         for(int i = 0; i<iteracionesMax; i++) {
56             // instantiate the initial solution using the initial
                   solution parameter
57             InitialSolution* initialSolution = new InitialSolution();
58             Solution* solution = initialSolution->getInitialSolution(
                   instance );
59             if(solution->path.size() == 0) {
60                 // no encontre un path entre u y v
61                 iteracionesSinInitialPathCount++;
62                 delete solution;
63                 if(iteracionesSinInitialPathCount <
                       iteracionesSinInitialPathMax) {
64                     continue; // sigo intentando buscar soluciones
65                 } else {
66                     break; // me rindo, dejo de buscar soluciones
67                 }
```

```
68              }
69
70              // El dijkstra de omega1 debe cumplir con el K, sino no
                    tiene sentido correr la heuristica
71              if(solution->totalOmega1 <= instance->K) {
72                  // run the heuristic
73                  Solution* newSolution = NULL;
74                  bool huboMejora = false;
75                  do
76                  {
77                      newSolution = selector->getBestNeighbour( solution
                            );
78                      // Si no logro mejorar la solucion, termino
79                      if(newSolution != NULL) {
80                          //cout << newSolution->totalOmega2 << endl;
81                          delete solution;
82                          solution = newSolution;
83                          huboMejora = true;
84                      } else {
85                          huboMejora = false;
86                      }
87                  } while(huboMejora);
88
89                  if(bestSolution == NULL) {
90                      bestSolution = solution;
91                  } else if(solution->totalOmega2 < bestSolution->
                        totalOmega2) {
92                      delete bestSolution;
93                      bestSolution = solution;
94                  } else {
95                      delete solution;
96                      iteracionesSinMejorarCount++;
97                  }
98              }
99
100             if(iteracionesSinMejorarCount > iteracionesSinMejorarMax)
                    {
101                 break;
102             }
103         }
104
105         // obtain the final time
106         timer.setFinalTime( "todo_el_codigo" );
107
108         /***************
109           Output Print
110         ***************/
111         // print the solution
112         if(bestSolution != NULL) {
113             bestSolution->print();
114             delete bestSolution;
115         } else {
116             cout << "no" << endl;
117         }
118
119         // save all obtained times to output
120         timer.saveAllTimes();
121     }
```

```
122
123     return 0;
124 }
```

Listing 25: InitialSolution::getInitialSolution()

```
1  Solution* InitialSolution::getInitialSolution(ProblemInstance*
       instance)
2  {
3    // creo el dijkstra
4      DijkstraRandomized<ObjectiveFunctionA> dijsktraRandom;
5      // creo la solucion
6      DijkstraSolution dijkstraSolution( instance->graph->nodeCount,
           instance->u);
7      // cargo en la solucion, todos los paths del dijkstra desde el
           nodo inicial
8      dijsktraRandom.findPath( instance->graph, &dijkstraSolution );
9      // obtengo el path que me interesa
10     Solution* solution = new Solution();
11     dijkstraSolution.getPath( instance->v, instance->graph, solution->
           path, solution->totalOmega1, solution->totalOmega2 );
12     return solution;
13 }
```

Listing 26: DijkstraRandomized::findPath()

```
1  void DijkstraRandomized<ObjectiveFunction>::findPath(Graph* graph,
       DijkstraSolution* solution) {
2      int* prevNodes = solution->prevNodes;
3      double* dist = new double[graph->nodeCount];
4      for (int i=0; i<graph->nodeCount; i++) {
5          dist[i] = INF;
6          prevNodes[i] = -1;
7      }
8      dist[solution->fromNode-1] = 0;
9
10     ObjectiveFunction objFunc;
11
12     unvisited.push(new UnvisitedNode(solution->fromNode, 0, 0));
13
14     while (unvisited.size() > 0) {
15         int rcl_size = min(RCL_SIZE, int(unvisited.size()));
16         int chosen = rand()%rcl_size + 1; // genera un int entre 1 y
               rcl_size
17         list<UnvisitedNode*> stack;
18         for (int i = 0; i < chosen; i++) {
19             stack.push_front(unvisited.top());
20             unvisited.pop();
21         }
22         UnvisitedNode* currNode = stack.front();
23         stack.pop_front();
24         for (list<UnvisitedNode*>::iterator it = stack.begin(); it !=
               stack.end(); it++)
25             unvisited.push(*it);
26
27         vector<Node> adjNodes = graph->getAdjacent(currNode->node);
```

```
28            for (int i=0; i<adjNodes.size(); i++) {
29                Node toNode = adjNodes[i];
30                if(prevNodes[toNode-1] != -1) continue;
31                Edge* edge = graph->getEdge(currNode->node, adjNodes[i]);
32                double weight = objFunc.weight(edge->omega1, edge->omega2)
                      ;
33                if(dist[currNode->node-1] + weight < dist[toNode-1]) {
34                    prevNodes[toNode-1] = currNode->node;
35                    dist[toNode-1] = dist[currNode->node-1] + weight;
36                    unvisited.push(new UnvisitedNode(toNode, edge->omega1,
                          edge->omega2));
37                }
38            }
39        }
40    }
```

Listing 27: DijkstraSolution::getPath()

```
1   void DijkstraSolution::getPath(int toNode, Graph* graph, vector<Edge*>
        &path, double &totalOmega1, double &totalOmega2) {
2       int prevNode = prevNodes[toNode-1];
3       totalOmega1 = 0;
4       totalOmega2 = 0;
5       list<Edge*> pathList;
6       while (prevNode != -1) {
7         Edge* edge = graph->getEdge(prevNode, toNode);
8           pathList.push_front(new Edge(prevNode, toNode, edge->omega1,
                edge->omega2));
9           totalOmega1 += edge->omega1;
10          totalOmega2 += edge->omega2;
11          toNode = prevNode;
12          prevNode = prevNodes[prevNode-1];
13      }
14
15      // toNode va cambiando dentro del while
16      // si llegado a este punto, toNode != fromNode, significa que no
            hay camino entre toNode y fromNode
17      if(toNode != fromNode) {
18          totalOmega1 = INF;
19          totalOmega2 = INF;
20          // el path que se devuelve en la solucion queda vacio
21      } else {
22          // solo devuelvo un camino si existe un camino posible entre
                fromNode y toNode
23          path.resize(pathList.size());
24          int index = 0;
25          for(list<Edge*>::iterator it = pathList.begin(); it !=
                pathList.end(); it++) {
26              path[index] = *it;
27              index++;
28          }
29      }
30  }
```