DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico3

Técnicas Algorítmicas Avanzadas

Viernes 9 de Mayo de 2014

Algoritmos y Estructuras de Datos III

Entrega de TP

Grupo ??

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 resolver el mismo.

El problema consiste en que dado un Grafo $G = (V, E)$, dos funciones de peso $\omega_1, \omega_2 : V \mapsto \mathbb{R}_+$, y un natural K, encontrar un camino P entre dos nodos $u, v \in V$ con costo $\omega_1(P) \leq K$ de manera tal que $\omega_2(P)$ sea mínimo.

Donde el costo del camino $\omega_x(P)$, con $1 \leq x \leq 2$, se define como

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

$CACM$ es un problema conocido, y tiene muchas aplicaciones en la vida real. Por ejemplo, supongamos que somos una empresa de turismo que ofrece paquetes de viajes. Una situación que se nos puede presentar es que un cliente nos pide organizarle un viaje de una ciudad X a otra ciudad Y para poder llegar en el mínimo tiempo, pero nos dice que el presupuesto que cuenta para gastar en transporte es de $\$K$. Este problema se puede modelar con $CACM$, donde las ciudades son los nodos del grafo, las aristas del mismo existen si entre las ciudades en cuestión hay algún medio de transporte, ω_1 representa el costo del viaje, y ω_2 es el tiempo que toma el viaje.

Otro ejemplo similar tiene que ver 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. Es un grafo muy evidente, donde las aristas son cuadras a recorrer. Las aristas recorridas caminando suman 100 metros a la cantidad caminada, mientras que las recorridas en colectivo no suman metros caminados.

Aunque $CACM$ es un problema conocido, no se conocen algoritmos polinomiales que lo resuelvan y por lo tanto pertenece al conjunto de los problemas NP. Nosotros en este TP analizaremos 6 métodos para resolverlo, una solución exacta y 5 aproximaciones a través de heurísticas. En concreto, los algoritmos que implementaremos son:

1. Un *Backtracking* como algoritmo exacto.
2. Tres heurísticas *constructivas greedy*, cada una con un criterio goloso diferente.
3. Una heurística de *búsqueda local*.
4. Una heurística *GRASP*.

Nos centraremos en experimentar sobre estos algoritmos, analizando su complejidad y la calidad de las soluciones de las heurísticas. También trataremos de definir las familias de grafos para las cuáles las heurísticas implementadas funcionan muy bien, y aquellas para las cuáles las mismas hallan una solución muy alejada de la óptima, o lo que es peor, que podrían no hallar ninguna.

2. Instrucciones de uso

3. Desarrollo del TP

3.1. Backtracking

3.1.1. Descripción

Debido a la dificultad computacional del problema, no existe aún una solución exacta de tiempo polinomial, y a pesar de que nos entretuvimos discutiendo, nosotros no pudimos encontrarla tampoco. En su defecto implementamos un algoritmo de backtracking que recorre todos los caminos posibles de u a v y se queda con el de menor ω_2 tal que $\omega_1 \leq K$.

El algoritmo funciona de la siguiente manera: en un momento dado va a tener construido un camino $P = [v_1 = u, \ldots, v_{i-1}]$, y toma un nodo v_i aun no visitado de la adyacencia de v_{i-1}, lo marca como visitado y lo agrega a P. Si $\omega_1(P)$ se pasa de K, este camino ya no nos sirve y no sigo avanzando en la recursión. Caso contrario, se llama a la recursión sobre el camino aumentado.

En el caso en que $v_i = v$, sabemos que tenemos una solución candidata. Comparamos $\omega_2(P)$ con el menor ω_2 entre las soluciones candidatas, y si es mejor solución, se guarda. Cuando hemos llegado a v no hace falta llamar a la recursión.

Siempre antes de retornar tras explorar un nodo de un camino, se lo marca como no visitado, para que pueda ser recorrido en otros caminos que se recorran.

A continuación, escribimos el pseudocódigo de la función `main`.

Algoritmo 1 *main*()

```
1:  Camino mejorSolucion
2:  ω2(mejorSolucion) ← ∞
3:  Camino ramaActual ← []
4:  para i en 1 a n hacer
5:      visitados[i] = False
6:  fin para
7:  backtrack( u, null )
8:  si ω2(mejorSolucion) < ∞ entonces
9:      imprimir(mejorSolucion)
10: sino
11:     imprimir("no")
12: fin si
```

Algoritmo 2 *backtrack*(Nodo actual, Nodo padre)

```
1: ramaActual.push( actual )
2: visitados[actual] ← true
3: si ω1(ramaActual) < K entonces
4:     si actual = v and ω2(ramaActual) < ω2(mejorSolucion) entonces
5:         mejorSolucion ← ramaActual
6:     sino si actual ≠ v and ω1(ramaActual) ≤ K entonces
7:         para cada Nodo n en adyacentes( actual ) hacer
8:             si no visitado[n] entonces
9:                 backtrack( n, actual )
10:            fin si
11:        fin para
12:    fin si
13: fin si
14: ramaActual.pop( actual )
15: visitados[actual] ← false
```

3.1.2. Complejidad

Analicemos cuantas llamadas se hacen a *backtrack*. Se exploran caminos de a los sumo n nodos. El primer nodo está fijo en u. El segundo nodo pertenece a la adyacencia del primero, que en peor caso tiene tamaño $n-1$. El tercero pertenece a la adyacencia del segundo, que no hayan sido visitados, cuyo tamaño es a lo sumo $n-2$. Y asi sucesivamente. En peor caso se llama a *backtrack* $n!$ veces.

Analicemos el cómputo que se realiza en cada llamada a *backtrack*. Preguntar si un nodo está visitado es acceder a un arreglo en forma constante. El costo de una función de peso asociada a un camino se va acumulando a medida que se agregan nodos. El costo se guarda y se puede acceder en forma constante. Lo queda por estudiar es la complejidad del ciclo for que recorre la adyacencia del nodo visitado. Guardamos una lista de adyacencia por lo que podemos recorrer la adyacencia de cualquier nodo con complejidad lineal en relación a su tamaño. El tamaño de la adyacencia es a lo sumo m. Para cada nodo adyacente se efectúan operaciones constantes, a excepción de las llamadas a *backtrack*. El costo de estas llamadas lo estamos calculando por separado.

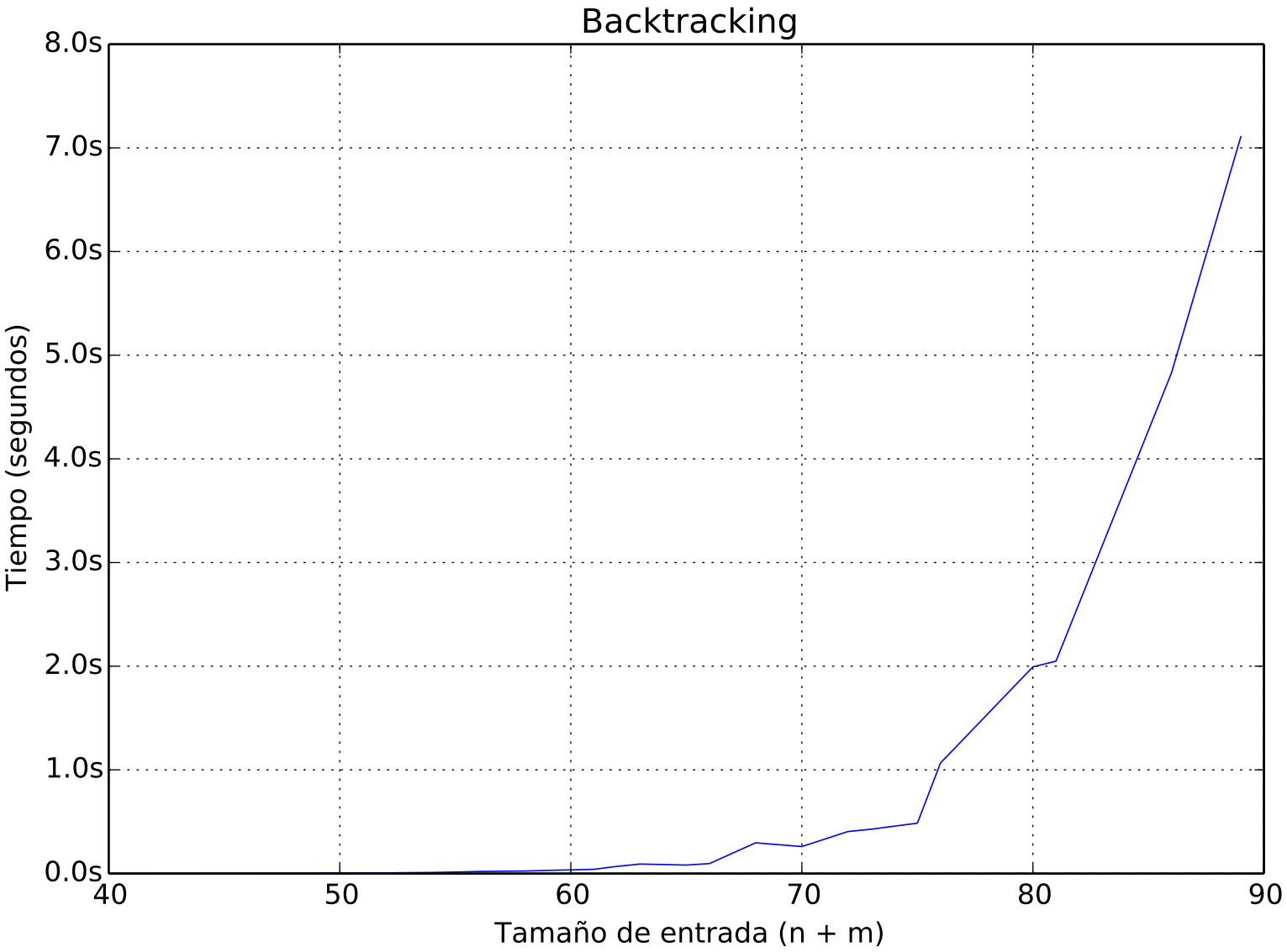
En definitiva hacemos $O(n!)$ llamadas a una función que de por si toma $O(m)$ operaciones. La complejidad del algoritmo es $O(m * n!)$.

3.1.3. Experimentación

Generamos en primera instancia grafos con una cantidad de nodos pequeña - entre 3 y 15. La cantidad de aristas, es decir, la densidad del grafo, se eligió aleatoriamente entre 0 y (n * (n - 1)) / 2, la cantidad máxima posible. La distribución de las aristas, sus pesos, y el valor de k también se eligieron de forma aleatoria.

A continuación presentamos los tiempos de ejecución de nuestro algoritmo frente

a estas instancias.


Backtracking
8.0s
7.0s
6.0s
5.0s
4.0s
3.0s
2.0s
1.0s
0.0s
Tiempo (segundos)
40
50
60
70
80
90
Tamaño de entrada (n + m)

Para tamaños de grafo menores que 40, los tiempos de ejecución fueron despreciables. Se puede apreciar el carácter exponencial del tiempo de ejecución de nuestro algoritmo en función del tamaño de entrada.

3.2. Greedy

3.2.1. Descripción

Un algoritmo goloso usa una heurística que consiste en elegir, en cada paso, una solución óptima local entre un conjunto de opciones, esperando encontrar al final la solución óptima global. En general estos algoritmos son eficientes y simples de diseñar e implementar, pero puede ser que nunca lleguen a la solución óptima del problema.

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

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

Lo que busca el algoritmo goloso es encontrar el conjunto 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 seleccciones, 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 subsoluciones óptimas a mi conjunto, al finalizar lo que se espera encontrar es la solución óptima.

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 en el libro mencionado.

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 de un nodo u a todos los de S. En cada fase de Dijkstra, se selecciona un nuevo nodo de $V \backslash S$ cuyo valor en π sea mínima

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

y lo añadimo a S, actualizando si es necesario π. Al finalizar, π es el vector con la mínima distancia a todos los nodos.

Entonces, nosotros para resolver el problema vamos 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. $f_C(e) = \omega_1(e)\omega_2(e)$

Luego el pseudocódigo de Dijkstra modificado con la nueva definición de distancia queda dado por:

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

1: $\pi(v) = \infty \quad \forall v \in V$
2: $\pi(v_0) = 0$
3: $S = \emptyset$
4: **para** $i = 1 \dots n - 1$ **hacer**
5: $\quad v \leftarrow$ nodo de $V \backslash S$ de mínimo π.
6: $\quad$ **para each** $w \in V \backslash S$ adyacente a v **hacer**
7: $\quad\quad \pi(w) = \text{mín}(\pi(w), \pi(v) + f((v, w)))$
8: $\quad$ **fin para**
9: $\quad S = S \cup \{v\}$
10: **fin para**
11: **retornar** π

La modificación está la línea 7, que en vez de sumar a $\pi(v)$ el peso de la arista, como es en el algoritmo original, le suma el valor de una función que define el peso de la arista. Esto nos permite mucha flexibilidad a la hora de cambiar la "decisión golosa".

3.2.2. Complejidad

Veamos que nuestro algoritmo es el mismo que el de Dijkstra salvo por que ejecuta la función $f((v, w))$, que al hacer operaciones que son constantes en el tiempo no altera la complejidad de final del algoritmo.

De acuerdo al libro de Brassard antes mencionado, demuestran que la complejidad de Dijkstra implementada sobre un heap es $O(m \log n)$, y como esta es la mejor complejidad que se puede conseguir, nosotros lo implementamos sobre un `priority_queue` de $C++$ que nos garantiza las mismas complejidades de un heap[2].

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

3.2.3. Familias Malas: Greedy 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.

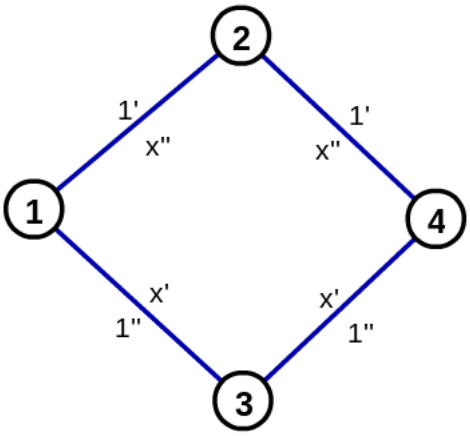


Figura 1

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.

3.2.4. Familias Malas: Greedy 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.

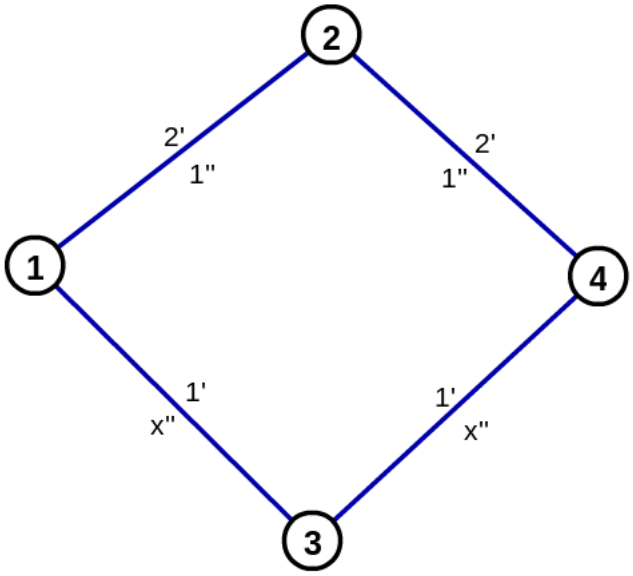


Figura 2

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

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

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.

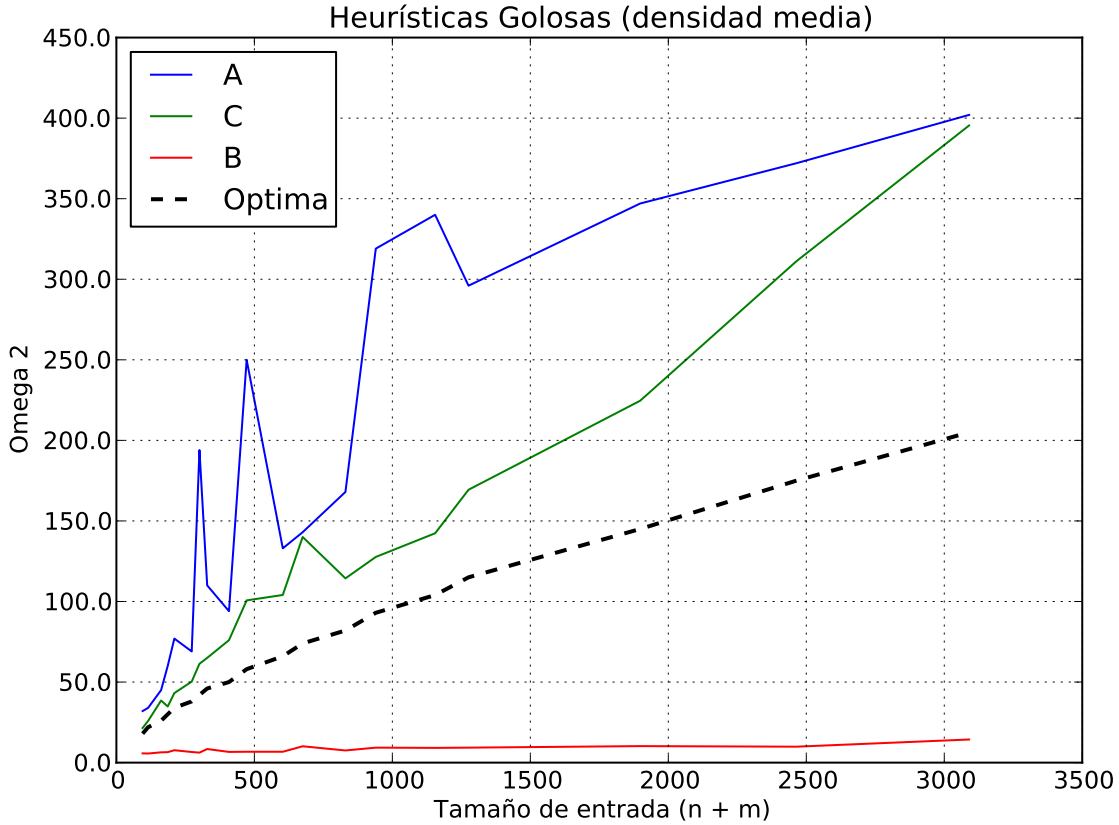
3.2.5. Familias Malas: Greedy C

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.

3.2.6. Experimentación

Resulta importante destacar las decisiones que tomamos para la generación de grafos. 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.



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 válida, pero no descuida el mantenerse dentro de la cota de k.

3.3. Local Search

La búsqueda local es un método heurístico para encontrar una solución factible a un problema. Se basa en obtener una solución inicial, s, y luego 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.

3.3.1. Solución inicial

Para obtener una solución inicial utilizamos Dijkstra. Experimentamos con la funciones objetivo Greedy A y Greedy C descritas en el apartado anterior(ver sección de heurística golosa). No experimentamos con Greedy B, ya que de encontrar una solución usando la sumatoria de los pesos ω_2 como función objetivo, podía pasar que no encontremos una solución o que encontremos la solución y en ese caso, no tiene sentido usar búsqueda local, ya que no hay mejor solución posible.

Resultó que el comportamiento experimentado de Greedy A y Greedy C no mostró diferencias notables. Por esa razón decimos usar únicamente Greedy A como solución inicial.

3.3.2. Definición de la vecindad

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

- $\omega_1(c^*) < K$
- $\omega_2(c^*) < \omega_2(c)$

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

Para obtener el camino $d^*_{n_1,n_2}$ utilizamos Dijkstra con la sumatoria de los pesos ω_2 como función objetivo(Greedy B). Lo que buscamos es mejorar el subcamino $d_{n_1,n_2} \in c$

obteniendo otro camino que tenga menor ω_2, y dado que usamos Dijkstra para encontrar el camino con ω_2 mínimo entre los nodos, entonces estamos obteniendo un camino que va a tener igual o menor ω_2. En caso de que sea posible hacer el reemplazo, como disminuimos el ω_2 de una parte del camino y dejamos el resto del camino igual, estamos logrando disminuir el ω_2 del camino completo.

3.3.3. Selección de vecino

Dada la vecindad S^*, se elige al vecino usando Steepest descent, con ω_2 como función objetivo.

3.3.4. Pseudocódigo

El algoritmo está implementado en la función `main`:

Algoritmo 3 *main*(int tipo_solucionInicial, Graph g, Nodo n1, Nodo n2)

```
1: crearMatrizCaminosMinimos(g)
2: Solution solucion = obtenerSolucionInicial(tipo_solucionInicial, g, n1, n2)
3: si tipo_solutionInicial ≠ Greedy_A && ω1(solucion) > K entonces solucion =
   obtenerSolucionInicial(Greedy_A, g, n1, n2)
4: fin si
5: si ω1(solucion) ≤ K entonces
6:     mientras True hacer
7:         Solution nuevaSolucion = dameMejorVecino(solucion)
8:         si nuevaSolucion == NULL entonces
9:             break
10:        fin si
11:        solucion = nuevaSolucion
12:    fin mientras
13: fin si
```

Algoritmo 4 *obtenerSolucionInicial*(int tipo, Graph g, Nodo n1, Nodo n2)

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

Algoritmo 5 *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:         si subSolucion == NULL entonces
7:             continue
8:         fin si
9:         Solution solucion_ij = dameCaminoResueltoEntre(nodos[i], nodos[j])
10:        Solution nuevaSolucion_ω2 = ω2(solucionOriginal) - ω2(subSolucion) + ω2(solucion_ij)
11:        Solution nuevaSolucion_ω1 = ω1(solucionOriginal) - ω1(subSolucion) + ω1(solucion_ij)
12:        si nuevaSolucion_ω2 < ω2(mejorSolucion) && nuevaSolucion_ω1 < K) entonces
13:            mejorSolucion = crearSolucionReemplazandoCamino(solucionOriginal, solucion_ij)
14:        fin si
15:    fin para
16: fin para
17: return mejorSolucion
```

Algoritmo 6 *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
```

3.3.5. 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 que abordaremos más adelante.

Para obtener la solución inicial, usamos obtenerSolucionInicial. Esta función devuelve un camino mínimo entre 2 nodos llamando a resolverConDijkstra con alguna función objetivo(Greedy_A o Greedy_C). La función resolverConDijkstra primero ejecuta Dijkstra para encontrar todos los caminos mínimos entre n1 y los demás nodos y después hace un traceback 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 traceback 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 comparara los valores de ω_1 y ω_2 y por 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 que hayamos encontrado una mejora al camino actual en la última iteración. 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 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 $n * (n-1)/2$, 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 $n * (n-1)/2$.

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 $t*(t-1)/2$ 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 $O(n * (mlogn))$.

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 $O(n)$.

Entonces, resulta ser que encontrar el mejor vecino, consiste en ejecutar el doble for de algo que tiene complejidad $O(n + 1 + n)$, o sea $O(n)$, entonces en total es $O(n^2 * n) = O(n^3)$.

Como habíamos explicado que dameMejorVecino se ejecuta en un ciclo de hasta $n*(n-1)/2$ iteraciones, o sea $O(n^2)$, y entonces resulta que la complejidad de encontrar el mejor vecino es $O(n^2 * n^3) = O(n^5)$.

Finalmente la complejidad total resulta ser la suma de las complejidades de crearMatrizCaminosMinimos, 2 veces obtenerSolucionInicial y n^2 veces dameMejorVecino. Es decir, $O(n * (mlogn) + 2 * (mlogn) + n^5) = O(n^5)$.

3.3.6. Familias malas

Si elegimos como solución inicial a la heurística Greedy B en la sección previa, se ha expuesto que puede fallar en el intento de dar una solución factible, aún existiendo una. El Greedy A siempre encuentra una solución factible de existir ésta.

Veamos que tomando nuestra heurística de búsqueda local como solución inicial, puede quedar arbitrariamente lejos de la solución óptima.

Presentamos la siguiente familia de grafos:

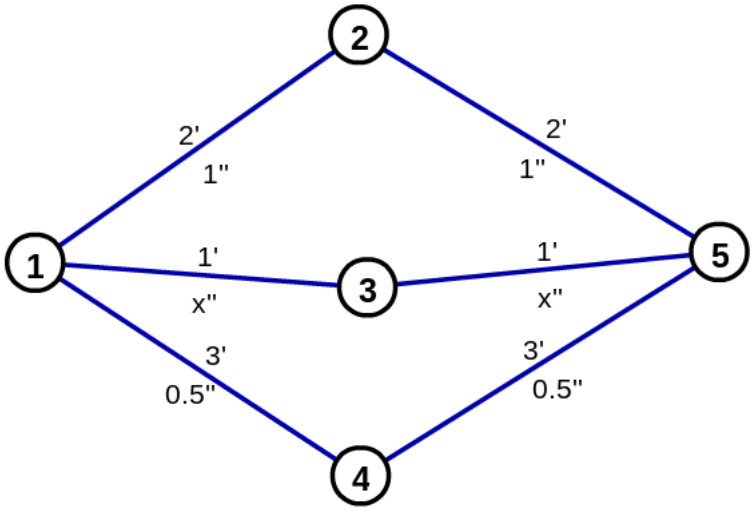


Figura 3

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) $1 \to 4 \to 5$;

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

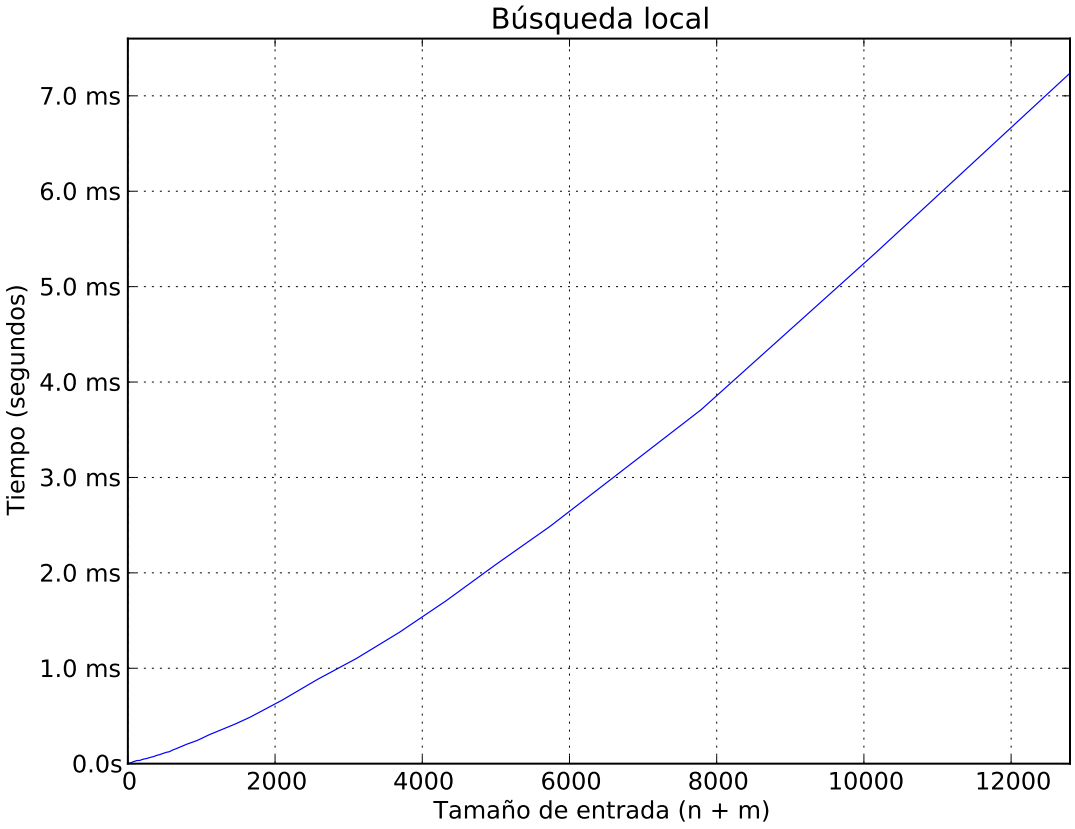
Supongamos que K vale 4. Como decíamos, partimos del camino mínimo entre u y v de acuerdo a ω_1, C_2. El algoritmo va a intentar intercambiar C_2 por C_3, el camino que minimiza ω_2. Sin embargo, como éste se pasa del límite K, el algoritmo no puede seguir y devuelve C_2. Sin embargo C_1 era una solución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

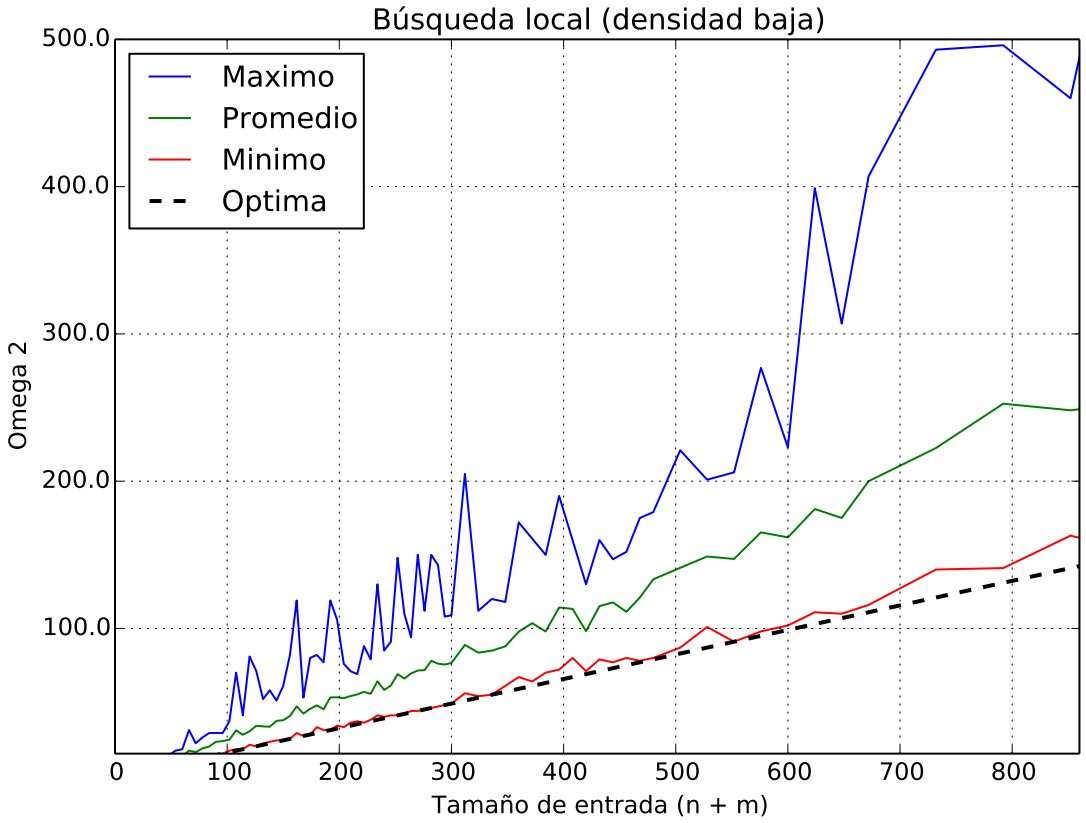
3.3.7. Experimentación

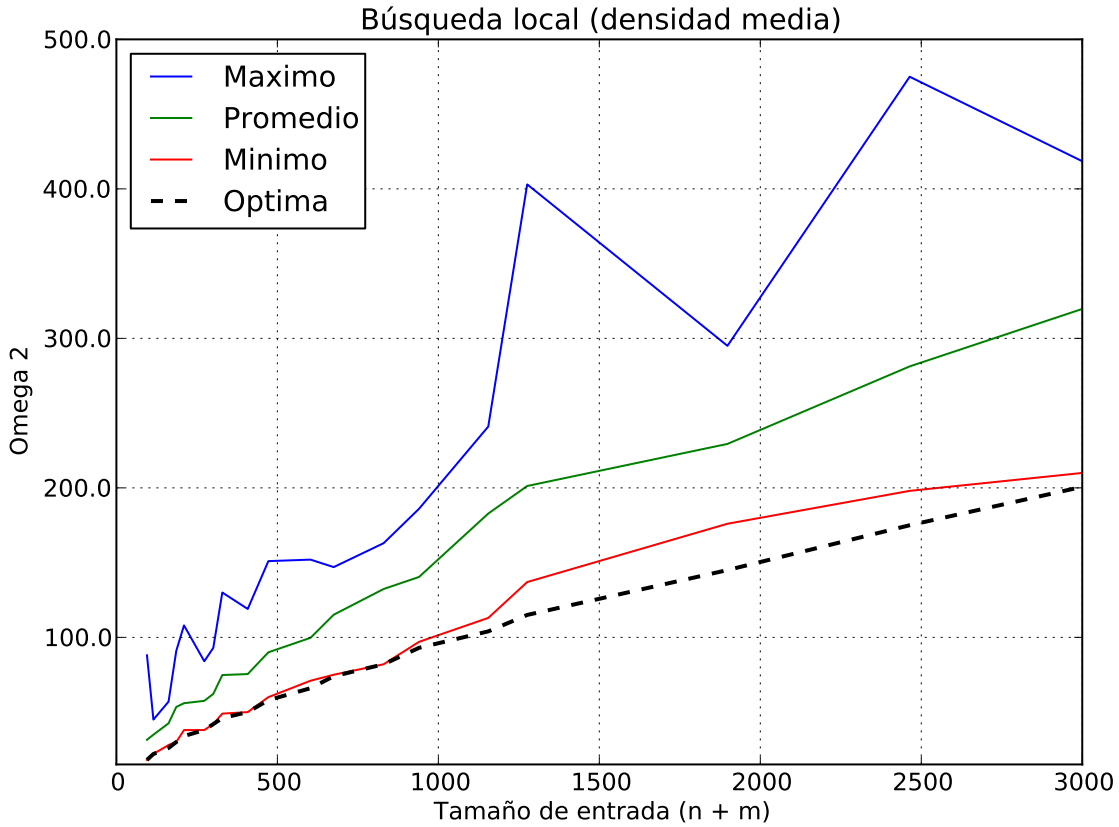
Pudimos analizar el tiempo de ejecución frente a grafos de un tamaño considerable. A continuación presentamos un gráfico.

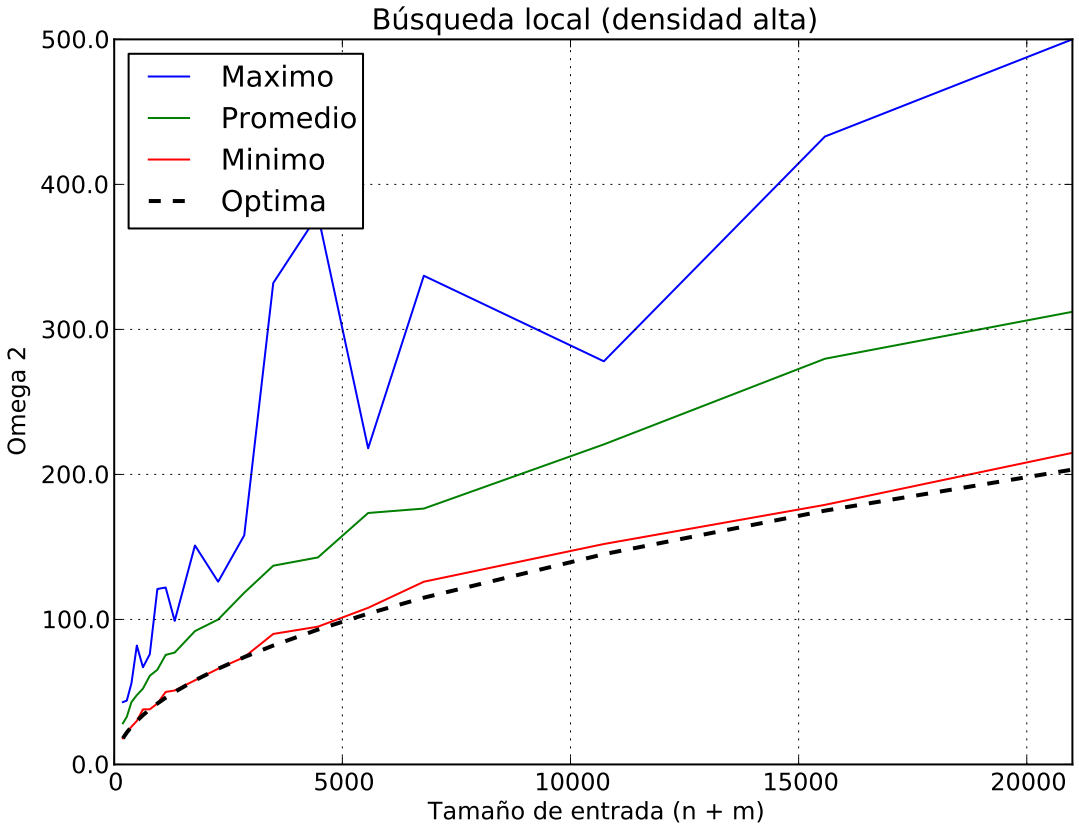


Se puede percibir el carácter no lineal de la complejidad. Sin embargo no se evidencia a luces claras la cota teórica de n^5 desarrollada previamente. Puede ser necesario un tamaño mayor para poder apreciar esa complejidad, o quizás la cota teórica no se alcanza en la práctica. Probablemente haya un factor que se amortice, habiendo escapado de nuestro escrutinio.

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 $\sqrt{n} * n$. Finalmente probamos con grafos ralos, con una cantidad de aristas del orden de la cantidad de nodos.


Búsqueda local (densidad baja)
Maximo
Promedio
Minimo
Optima
Omega 2
Tamaño de entrada (n + m)
500.0
400.0
300.0
200.0
100.0
0
100
200
300
400
500
600
700
800


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



En todos los gráficos se puede notar, a medida que aumenta el tamaño del grafo, un mayor rango de resultados. Ésto tiene cierta intuición. A medida que el grafo es más grande, son más los caminos posibles y hay más variedad de soluciones. La solución media, no obstante, parece preservar cierta distancia relativa a la solución óptima. La mejor solución encontrada guarda muy estrecha distancia con la óptima.

3.4. GRASP

La metaheurística Grasp es una combinación entre una heurística golosa aleatorizada y un procedimiento de búsqueda local. 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.

3.4.1. Solución inicial

Usamos Dijkstra como nuestra heurística golosa, pero modificado para agregarle aleatoriedad. El factor aleatorio consiste en que en cada iteración de Dijkstra, en vez de tomar el nodo no visitado que minimiza la función objetivo, tomamos uno de entre los $beta$[3] menores.

Dado la aleatoriedad de la solución inicial, es posible que esta no sea factible. En caso de obtener una solución de ese tipo, no ejecutamos búsqueda local.

Para Dijkstra usamos la tres funcion objetivo Greedy, A definida en el apartado Greedy.

3.4.2. 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 valores usando n. Elegimos usar n para la cantidad máxima de iteraciones sin haber encontrado mejores y sin haber encontrado una solución factible y $n * log(n)$ para la cantidad máxima de iteraciones. Esta elección para las constantes fue a partir de experimentar con distintos valores basados en n, que nos pareció un buen parámetro para definirlos, ya que generalmente mientras más nodos tenga el grafo,

[3]: Luego de experimentar con distintos valores, encontramos que $beta$=10 era un valor que presentaba suficiente aleatoriedad.

más chance tiene de fallar en encontrar una solución inicial factible o de no encontrar una solución que mejore a la actual y al mismo tiempo nos interesa ejecutar por más iteraciones, para aumentar la chance de dar con una buena solución.

3.4.3. Búsqueda local

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

3.4.4. Pseudocódigo

El algoritmo está implementado en la función `main`:

Algoritmo 7 *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 8 *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.

3.4.5. 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, hay que remover los primeros *beta* nodos de la cola y luego volver a agregar 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 $O((beta + beta - 1) * log(n))$, pero como *beta* es constante entonces la complejidad resulta $O(log(n))$.

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 $O(n^5 * iteracionesMax) = O(n^5 * n * log(n)) = O(n^6 * log(n))$.

4. Apéndices

4.1. Código Fuente (resumen)

4.1.1. Backtracking

```
int N, M, U, V, K;
Graph *G;
vector<bool> visitados;
Solucion mejorSolucion;
Solucion ramaActual;

void backtrack(Node actual, Node padre) {
  // primero agrego el nodo
  ramaActual.camino.push_back(actual);

  Edge *e = G->getEdge(padre, actual);

  if (actual != padre) {  // me cubro del nodo inicial
    ramaActual.sumaOmega1 += e->omega1;
    ramaActual.sumaOmega2 += e->omega2;
  }
  ramaActual.cantAristas++;
  visitados[actual] = true;

  if (ramaActual.sumaOmega1 > K) {
    // no haces nada
  } else if (actual == V) {
    if (ramaActual.sumaOmega2 < mejorSolucion.sumaOmega2)
      mejorSolucion = ramaActual;
  } else { // llamas a la recursion
    vector<Node> vecinos = G->getAdjacent(actual);
    for (int i = 0; i < vecinos.size(); i++) {
      Node vecino = vecinos[i];
      if (! visitados[vecino]) {
        backtrack(vecino, actual);
      }
    }
  }
  // dejas todo como estaba, antes de retornar
  ramaActual.camino.pop_back();
  if (actual != padre) {
    ramaActual.sumaOmega1 -= e->omega1;
    ramaActual.sumaOmega2 -= e->omega2;
  }
  ramaActual.cantAristas--;
  visitados[actual] = false;
}
```

4.1.2. Greedy

```
void DijkstraSolution::getPath(int toNode, Graph* graph, vector<Edge*> &path,
                                            double &totalOmega1, double &totalOmega2) {
  int prevNode = prevNodes[toNode-1];
  totalOmega1 = 0;
  totalOmega2 = 0;
  list<Edge*> pathList;
  while (prevNode != -1) {
            Edge* edge = graph->getEdge(prevNode, toNode);
    pathList.push_front(new Edge(prevNode, toNode, edge->omega1, edge->omega2));
    totalOmega1 += edge->omega1;
    totalOmega2 += edge->omega2;
    toNode = prevNode;
    prevNode = prevNodes[prevNode-1];
  }

  // toNode va cambiando dentro del while
  // si llegado a este punto, toNode != fromNode, significa que no hay camino
  // entre toNode y fromNode
  if(toNode != fromNode) {
    totalOmega1 = INF;
    totalOmega2 = INF;
    // el path que se devuelve en la solucion queda vacio
  } else {
    // solo devuelvo un camino si existe un camino posible entre fromNode y toNode
    path.resize(pathList.size());
    int index = 0;
    for(list<Edge*>::iterator it = pathList.begin(); it != pathList.end(); it++) {
      path[index] = *it;
      index++;
    }
  }
}

template<class ObjectiveFunction>
void GreedyHeuristic<ObjectiveFunction>::resolveInstance( ProblemInstance* instance ){
  // creo el dijkstra
  Dijkstra<ObjectiveFunction> dijsktra;
  // creo la solucion
  DijkstraSolution sol( instance->graph->nodeCount, instance->u );
  // cargo en la solucion, todos los paths del dijkstra desde el nodo inicial
  dijsktra.findPath( instance->graph, &sol );
  // obtengo el path que me interesa
  sol.getPath( instance->v, instance->graph, solution->path,
                      solution->totalOmega1, solution->totalOmega2 );
}
```

4.1.3. Local Search

```
int main( int argc, char const* argv[] )
{
  ....
  InitialSolution* initialSolution =
    heuristicFactory.createInitialSolution( initialSolutionParameter );
  InitialSolution* initialSolutionBestOmega1 =
    heuristicFactory.createInitialSolution( INITIAL_SOLUTION_A );
  NeighbourhoodSelector* selector =
    heuristicFactory.createNeighborhoodSelector( neighborhoodSelectorParameter );

  // parse the input
  parser.parseInput();

  for(auto instance:parser.problemInstances)
  {
  Solution* solution = initialSolution->getInitialSolution( instance );
  // si no encuentro el path que cumpla con K, pruebo usando dijkstra con omega1.
  // Tambien puede pasar que no encuentre ningun path,
  // por lo que totalOmega1 = INF, y en ese
  // caso tambien pruebo buscar otro path
  if( initialSolution->type != initialSolutionBestOmega1->type &&
    solution->totalOmega1 > instance->K) {
    delete solution;
    solution = initialSolutionBestOmega1->getInitialSolution(instance);
  }

  // El dijkstra de omega1 debe cumplir con el K, sino no
  // tiene sentido correr la heuristica
  if(solution->totalOmega1 <= instance->K) {
    // run the heuristic
    Solution* newSolution = NULL;
    bool huboMejora = false;
    do
    {
      newSolution = selector->getBestNeighbour( solution );
      // Si no logro mejorar la solucion, termino
      if(newSolution != NULL) {
           delete solution;
           solution = newSolution;
           huboMejora = true;
      } else {
           huboMejora = false;
      }
    } while(huboMejora);
  }
  solution->print();
  delete solution;
  }
```

```
  return 0;
}

Solution* NeighbourhoodSelectorA::getBestNeighbour(const Solution* origSolution)
{
  // Itero sobre la matriz de soluciones
  // Por cada par de nodos, tengo que buscar si existe un path en la matriz,
  // tal que sustituyendolo pot path entre el par de nodos de la solucion original
  // se obtenga una nueva solucion tal que se su totalOmega2 sea menor

  // El path de una solution es un vector de ejes
  // En cada iteracion que encuentro una solucion mejor, voy a cambiar algunos
  // de los nodos, por lo que tengo que recalcular los nodos de la mejor solucion
  int edgesCount = origSolution->path.size();
  int nodesCount = edgesCount+1;
  vector<int> nodes(nodesCount);
  for(int i=0; i<edgesCount; i++) {
    Edge* edge = origSolution->path[i];
    nodes[i] = edge->fromNode;
  }
  // ultimo nodo del path
  nodes[nodesCount-1] = origSolution->path[edgesCount-1]->toNode;

  Solution* bestSolution = NULL;
  for(int i=0; i<nodes.size() - 1; i++) {
  for(int j=i+1; j<nodes.size(); j++) {
    // solution con sub path entre los nodos
    Solution* subSolution = origSolution->createSubSolutionBetween(nodes[i], nodes[j]);
    if(subSolution == NULL) continue;
    // dijkstra por omega2 entre los nodos
    Solution* solution_ij = getSolvedPathBetween(nodes[i], nodes[j]);
    if(solution_ij == NULL) continue;


    // Si el path creado con dijkstra usando omega2, tiene menos omega2 total,
    // que el path actual entre los nodos i y j, entonces chequeo si al
    // crear una nueva solucion tendra menos omega2 que la mejor solucion.
    // En ese caso me guardo esta nueva solucion como la mejor hasta ahora.
    // Ademas chequeo que se cumpla con el K requerido.
    double bestSolutionOmega2 = bestSolution == NULL ?
                origSolution->totalOmega2 :
                bestSolution->totalOmega2;
    double newSolutionOmega2 = origSolution->totalOmega2 - subSolution->totalOmega2
                 + solution_ij->totalOmega2;
    double newSolutionOmega1 = origSolution->totalOmega1 - subSolution->totalOmega1
                 + solution_ij->totalOmega1;
    if(newSolutionOmega2 < bestSolutionOmega2 && newSolutionOmega1 <= K) {
    if(bestSolution != NULL) {
      delete bestSolution;
    }
    // creo la nueva mejor solucion
```

```
    bestSolution = createNewSolutionReplacingPath(origSolution, solution_ij);
    }
    delete subSolution;
  }
  }
  return bestSolution;
}

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;
      subPathStartsAtEdgeIndex = i;
      break;
    } else {
      res->path.push_back(edge);
      res->totalOmega1 += edge->omega1;
      res->totalOmega2 += edge->omega2;
    }
  }

  // agrego todos los ejes del sub path
  for(int i=0; i<sub->path.size(); i++) {
    Edge* edge = sub->path[i];
    res->path.push_back(edge);
    res->totalOmega1 += edge->omega1;
    res->totalOmega2 += edge->omega2;
  }

  // busco el nodo desde donde continua el pedazo del path original
  // agrego todos los ejes del path original a partir de ahi
  int fromNode = subPathStartsAtNode1 ? node2 : node1;
  bool addEdges = false;
  for(int i=subPathStartsAtEdgeIndex+1; i<orig->path.size(); i++) {
    Edge* edge = orig->path[i];
    if(edge->fromNode == fromNode) {
      addEdges = true; // encontre el nodo, asi que comienzo a añadir los nodos desde aca
    }
    if(addEdges) {
      res->path.push_back(edge);
      res->totalOmega1 += edge->omega1;
```

```
      res->totalOmega2 += edge->omega2;
    }
  }

  return res;
}
```

4.1.4. Grasp

```
int main( int argc, char const* argv[] )
{
  // instantiate the neighborhood selector using the neighborhood selector parameter
  NeighbourhoodSelector* selector =
          heuristicFactory.createNeighborhoodSelector( neighborhoodSelectorParameter );
  // parse the input
  parser.parseInput();

  for(auto instance:parser.problemInstances)
  {
    selector->initialize(instance);

    // valores arbitrarios basados en n para criterio de terminaciones
    int n = instance->graph->nodeCount;
    int iteracionesSinMejorarCount = 0;
    int iteracionesSinMejorarMax = n;
    int iteracionesMax = n * log(n);
    int iteracionesSinInitialPathCount = 0;
    int iteracionesSinInitialPathMax = n;

    Solution* bestSolution = NULL;
    // aca faltaria hacer un dijkstra por omega1, para ver que al
    // menos existe una solucion factible

    for(int i = 0; i<iteracionesMax; i++) {
      // instantiate the initial solution using the initial solution parameter
      InitialSolution* initialSolution =
              heuristicFactory.createInitialSolution( initialSolutionParameter );
      Solution* solution = initialSolution->getInitialSolution( instance );
      if(solution->path.size() == 0) {
        // no encontre un path entre u y v
        iteracionesSinInitialPathCount++;
        delete solution;
        if(iteracionesSinInitialPathCount < iteracionesSinInitialPathMax) {
          continue; // sigo intentando buscar soluciones
        } else {
          break; // me rindo, dejo de buscar soluciones
        }
      }
```

```
      // El dijkstra de omega1 debe cumplir con el K, sino no tiene
      // sentido correr la heuristica
      if(solution->totalOmega1 <= instance->K) {
        // run the heuristic
        Solution* newSolution = NULL;
        bool huboMejora = false;
        do
        {
          newSolution = selector->getBestNeighbour( solution );
          // Si no logro mejorar la solucion, termino
          if(newSolution != NULL) {
            delete solution;
            solution = newSolution;
            huboMejora = true;
          } else {
            huboMejora = false;
          }
        } while(huboMejora);

        if(bestSolution == NULL) {
          bestSolution = solution;
        } else if(solution->totalOmega2 < bestSolution->totalOmega2) {
          delete bestSolution;
          bestSolution = solution;
        } else {
          delete solution;
          iteracionesSinMejorarCount++;
        }
      }

      if(iteracionesSinMejorarCount > iteracionesSinMejorarMax) {
        break;
      }
    }

    if(bestSolution != NULL) {
      bestSolution->print();
      delete bestSolution;
    } else {
      cout << "no" << endl;
    }
  }

  return 0;
}
```