



DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 3

25 de julio de 2014

Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Chapresto, Matías	201/12	matiaschapresto@gmail.com
Dato, Nicolás	676/12	nico_dato@hotmail.com
Fattori, Ezequiel	280/11	ezequieltori@hotmail.com
Vileriño, Silvio	106/12	svilerino@gmail.com

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		



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) 4576-3300

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

Índice

1. Situaciones de la vida real que pueden ser modeladas con el problema del Camino acotado de costo mínimo	3
1.1. Notación y definiciones iniciales	3
1.2. Viaje en ruta: Minimizar la distancia de viaje dada una cantidad fija de dinero	3
1.3. Camino mas conveniente segun relieve: Expedicion en la colina	4
2. Algoritmo exacto para la resolución de CACM	5
2.1. Notación	5
2.2. Explicación detallada del algoritmo propuesto	5
2.2.1. Descripción	5
2.2.2. Algoritmo	5
2.3. Complejidad temporal para el peor caso	7
2.4. Experimentacion: Mediciones de Performance	8
2.4.1. Rendimiento para grafos con densidad lineal de aristas	9
2.4.2. Rendimiento para grafos con densidad cuadratica de aristas	12
2.4.3. Análisis factorial	15
3. Heurística Golosa	16
3.1. Explicacion detallada de la heurística propuesta	16
3.1.1. Inicialización	16
3.1.2. Heurística golosa	17
3.1.3. Pseudocódigo	17
3.1.4. Complejidad	18
3.1.5. Solución Factible	19
3.2. Nivel de optimalidad de las soluciones	20
3.3. Experimentacion: Mediciones de Performance	21
3.3.1. Consideraciones acerca de la complejidad teórica	21
3.3.2. Rendimiento para grafos con densidad lineal de aristas	21
3.3.3. Rendimiento para grafos con densidad cuadratica de aristas	24
3.3.4. Conclusión	27
3.4. Experimentacion: Mediciones de Optimalidad de las soluciones obtenidas con esta heurística	28
3.4.1. Grafos aleatorios de baja densidad de aristas	28
3.4.2. Grafos aleatorios de alta densidad de aristas	28
4. Heurística de búsqueda local	30
4.1. Criterios de vecindad	30
4.2. Explicacion detallada del algoritmo propuesto	31
4.2.1. Obtencion de la solucion inicial factible	31
4.2.2. Criterio de terminación	31
4.2.3. Pseudocodigo	31
4.2.4. Reemplazar un nodo intermedio en una 3-upla consecutiva de nodos del camino	32
4.2.5. Insertar un nodo intermedio en un par consecutivo de nodos del camino	32
4.2.6. Contracción de 3-uplas	32
4.3. Nivel de optimalidad de las soluciones	33
4.3.1. Familias de grafos malas para esta heurística	33
4.4. Complejidad	34
4.5. Taboo list	35
4.6. Precómputo de vecinos en comun	35
4.7. Experimentacion: Mediciones de Performance	35
4.7.1. Rendimiento para grafos con densidad lineal de aristas	35
4.7.2. Rendimiento para grafos con densidad cuadratica de aristas	38

4.8.	Experimentacion: Variación evolutiva de mejora en cada iteracion y Variación absoluta de w2 en busqueda local	42
4.8.1.	Rendimiento evolutivo y absoluto en grafos con alta densidad de aristas	42
4.8.2.	Rendimiento evolutivo en grafos con densidad lineal de aristas	44
4.8.3.	Conclusion de experimentos - Relacion entre cantidad de aristas y efectividad de busqueda local	45
4.9.	Experimentacion: Distintas soluciones iniciales factibles	46
4.9.1.	Grafos particulares	46
4.9.2.	Grafos aleatorios de alta densidad de aristas	46
4.9.3.	Grafos aleatorios de baja densidad de aristas	47
4.10.	Variación de efectividad de busqueda local sobre conjunto fijo de grafos cambiando vecindades	47
4.10.1.	Experimentación sobre conjunto de grafos con baja densidad de aristas	47
4.10.2.	Experimentación sobre conjunto de grafos con alta densidad de aristas	49
4.11.	Experimentacion: Mediciones de Optimalidad de las soluciones obtenidas con esta heurística	50
4.11.1.	Grafos aleatorios de baja densidad de aristas	50
4.11.2.	Grafos aleatorios de alta densidad de aristas	51
5.	Metaheuristica GRASP: Solucion propuesta	52
5.1.	Modificación a la heurística golosa	52
5.2.	Critero de terminación	53
5.3.	Consideraciones	53
5.4.	Pseudocódigo	53
5.5.	Análisis de complejidad	55
5.6.	Experimentacion: Mediciones de Performance	55
5.6.1.	Rendimiento para grafos con densidad cuadratica de aristas	55
5.6.2.	Rendimiento para grafos con densidad lineal de aristas	58
5.7.	Experimentación de optimalidad	61
5.7.1.	Optimalidad para grafos lineales con $w1 = 120$	61
5.7.2.	Optimalidad para grafos lineales con $w1 = 200$	62
5.7.3.	Optimalidad para grafos cuadráticos con $w1 = 150$	63
5.7.4.	Optimalidad para grafos cuadráticos con $w1 = 120$	64
5.7.5.	Optimalidad para grafos con RCL por cantidad	65
6.	Experimentacion General	69
6.1.	Generacion de conjuntos de grafos aleatorios	69
6.1.1.	Generador de grafos aleatorios	69
6.1.2.	Script generador de conjuntos de grafos	69
6.2.	Scripts de optimalidad - Calculo de puntajes y estadísticas	70
6.3.	Scripts de optimalidad - Graficos de optimalidad comparativos	70
6.4.	Calidad de las heurísticas respecto a la solución exacta	70
6.4.1.	Comparacion Exacta-Golosa-Busqueda Local-GRASP	70
6.4.2.	Grafos aleatorios de baja densidad de aristas	71
6.4.3.	Grafos aleatorios de intermedia densidad de aristas	72
6.4.4.	Grafos aleatorios de alta densidad de aristas	73
6.4.5.	Gráficos de distribucion de las soluciones	77
6.5.	Experimentacion pura entre heurísticas	77
6.5.1.	Grafos aleatorios de alta densidad de aristas	77
6.5.2.	Grafos aleatorios de densidad intermedia de aristas	78
6.5.3.	Grafos aleatorios de densidad baja de aristas	78
7.	Conclusion	80
8.	Compilacion	81

9. Entregable	81
10.Reentrega: Informe de modificaciones	82
11.Apéndice: Código fuente relevante	84
11.1. Generador aleatorio de grafos Fisher Yates	84
11.2. Script de minimalidad	85
11.3. Script de optimalidad	85
11.4. Script de performance	85
11.5. Algoritmo exacto para la resolución de CACM	85
11.6. Heurísticas	88

Resumen

El objetivo de este documento es describir el problema presentado, y las diversas soluciones que se fueron realizando utilizando distintas técnicas de programación, para la solución exacta, o diversas heurísticas para resolver el problema de forma polinomial, dado que no se conoce algoritmo de solución exacta con complejidad polinomial.

El problema que se nos presenta es, dado un grafo $G = (V, E)$, dos vértices $u, v \in V$, dos funciones w_1 y w_2 de costo asociadas a las aristas del grafo, y un valor $K \in \mathbb{R}$, se pide resolver el problema de Camino Acotado de Costo Mínimo (CACM), el cual consiste en encontrar un camino P entre u, v tal que el costo del camino respecto a la función w_2 sea mínimo entre todos los caminos con $w_1 \leq K$ y valga la condición del costo del camino $w_1 \leq K$.

Se nos requirió modelar situaciones de la vida real con CACM y luego implementar diversas soluciones para este problema, preferimos C y C++ como lenguajes de programación de este trabajo práctico y en él fueron implementados los siguientes algoritmos:

- Solución exacta (C/C++)
- Heurística constructiva golosa (C++)
- Heurística de búsqueda local (C++)
- Metaheurística GRASP (C++)

Todas las soluciones listadas arriba fueron testeadas con diversos casos de testing y serán claramente documentadas en las secciones de este documento. Para cada punto se detalla el algoritmo, se realizan justificaciones acerca de su correcto funcionamiento donde sea necesario, se establece una cota teórica sobre la complejidad temporal, y se brindan ejemplos de funcionamiento. Además se realizaron benchmarks de performance sobre diferentes conjuntos de instancias distintas de entrada que afirman la complejidad teórica y son presentadas con gráficos que acompañan dichos experimentos. Para las heurísticas golosas y de búsqueda local se detallan familias o instancias malas de entradas donde la solución provista no es la óptima, se realizaron diversos análisis de optimalidad y variación de parámetros de las heurísticas detallados en cada sección de cada heurística.

Para la metaheurística GRASP se ajustaron los parámetros para obtener cierto tradeoff, y fueron fijadas en esos parámetros para la sección de experimentación general.

Finalmente se realizó una experimentación general con respecto al rendimiento de las heurísticas respecto a exacta, las heurísticas solas entre ellas para entradas más grandes de las que soporta el algoritmo exacto en la práctica, los experimentos miden y comparan la cercanía de las soluciones respecto a la solución óptima con diversos estimadores estadísticos, así como también el tiempo insumido en obtener las soluciones para realizar un tradeoff entre porcentaje de efectividad de la heurística y tiempo consumido por el algoritmo.

1. Situaciones de la vida real que pueden ser modeladas con el problema del Camino acotado de costo minimo

1.1. Notación y definiciones iniciales

Sea un grafo $G = (V, E)$ un grafo simple. Definimos el costo asociado a una funcion $f : E \rightarrow \mathbb{R}_+$ de un camino $P = \langle v_1, \dots, v_{k-1}, v_k \rangle$ de la siguiente forma:

$$\text{costo}_f(P) = \sum_{i=1}^{k-1} f(v_i v_{i+1})$$

1.2. Viaje en ruta: Minimizar la distancia de viaje dada una cantidad fija de dinero

Sea $G = (V, E)$ un grafo simple, se quiere modelar un mapa de ciudades y rutas entre ellas, para ello definamos V , como el conjunto de nodos donde cada nodo representa una ciudad en el mapa. Asimismo, E será el conjunto de aristas en donde, sean $v_1, v_2 \in V$ entonces $\exists (v_1, v_2) \in E \Leftrightarrow$ existe una ruta entre las ciudades v_1 y v_2 . A continuación definiremos dos funciones sobre el conjunto E de aristas tales que:

- $f : E \rightarrow \mathbb{R}_+$: funcion asociada a la distancia entre v_{src} y v_{dst} .
- $g : E \rightarrow \mathbb{R}_+$: funcion asociada al costo de viajar entre v_{src} y v_{dst} .

Sea ademas $k \in \mathbb{R}_+$ el dinero del que se dispone para realizar el viaje, el objetivo de esta situacion es, dados $v_1, v_2 \in V$ se quiere llegar de la ciudad v_{src} a la ciudad v_{dst} minimizando la distancia del viaje, pero se debe poder cubrir el costo total del viaje con la cantidad k de dinero disponible.

El la siguiente figura se ve un ejemplo de como podria ser un grafo de estas características, además se ve que el camino mas corto en respecto de la funcion distancia f , no necesariamente cumple el requisito de estar dentro de la cota respecto a la funcion de costo g y el dinero disponible k . Los pares ordenados en las aristas indican (f: distancia, g: costo).

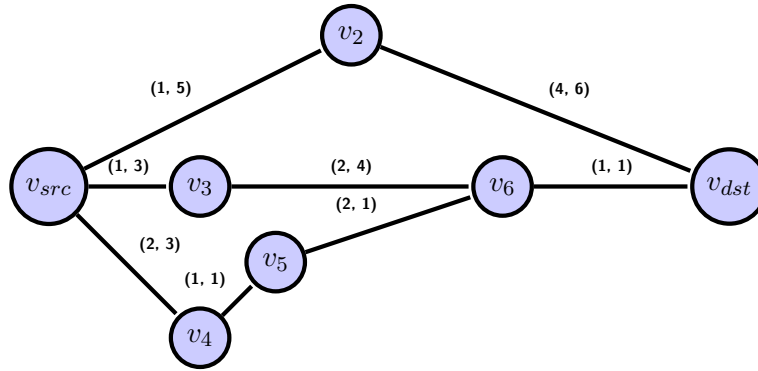


Figura 1: Ejemplo de conexion entre ciudades.

Veamos 3 posibles caminos y analicemos cada uno:

- $c_1 = \langle v_{src}, v_2, v_{dst} \rangle$
 - $\text{dist}(c_1) = 1 + 4 = 5$
 - $\text{costo}(c_1) = 5 + 6 = 11$
- $c_2 = \langle v_{src}, v_3, v_6, v_{dst} \rangle$
 - $\text{dist}(c_2) = 1 + 2 + 1 = 4$
 - $\text{costo}(c_2) = 3 + 4 + 1 = 8$
- $c_3 = \langle v_{src}, v_4, v_5, v_6, v_{dst} \rangle$
 - $\text{dist}(c_3) = 2 + 1 + 2 = 5$

- $\text{costo}(c_3) = 3 + 1 + 1 = 5$

Supongamos que se dispone de $K=9$ para realizar el viaje, entonces veamos que el camino mas corto seria c_1 pero vemos que con esos costos se pasa de $K=9$, en particular el costo asociado a c_1 es 11. Entonces el mejor camino acotando el costo por 9 es c_2 . Para instancias mas grandes, seguramente no será facil ver los caminos de esta forma o poder encontrar el óptimo, ahí es cuando podemos modelar este problema utilizando CACM para lograr el objetivo.

1.3. Camino mas conveniente segun relieve: Expedicion en la colina

Otra posible aplicacion para este problema es la siguiente situacion: Se quiere ir de un punto A a otro B en un lugar geografico accidentado, donde existe una zona con un relieve complicado, llamemos C a esta zona, impidiendo cualquier camino directo entre A y B sin pasar por C. La zona accidentada, tiene varias estaciones de reabastecimiento para los viajeros. Procedamos a modelar con grafos la situacion: Sea $G = (V, E)$ un grafo simple, los puntos A y B son dos nodos no adyacentes de G, ambos conectados a una componente conexas C(no necesariamente por una sola arista) (Figura 2),luego, como C es conexa y A,B estan ambos conectados con C, existe al menos un camino entre A y B, que pasa por C. Dentro de C, los nodos denotan estaciones de reabastecimiento y si existe una arista entre dos nodos dentro de C, significa que se puede viajar entre dichas estaciones, mas formalmente:

Sean $v_1, v_2 \in V$ entonces $\exists (v_1, v_2) \in E$ de peso $(t, l) \in \mathbb{R}_+^2 \Leftrightarrow$ existe un sendero entre los puntos v_1 y v_2 con costo de l litros de nafta que toma t minutos en ser recorrido. Cada una de las aristas (v_i, v_j) , tanto entre A,B hacia C o mismas aristas contenidas en C, tienen asociadas dos funciones f, g definidas sobre el conjunto E de aristas del grafo G:

- $f : E \rightarrow \mathbb{R}_+$: funcion asociada al tiempo de viaje en minutos entre v_i y v_j .
- $g : E \rightarrow \mathbb{R}_+$: funcion asociada al costo de viajar entre v_i y v_j , en cantidad de litros de nafta segun el terreno de dicho sendero.

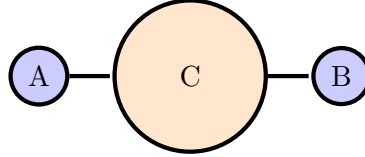


Figura 2: Nodos A,B no adyacentes entre si, conectados a componente conexa C

El objetivo es llegar de A a B en la menor cantidad de tiempo, disponiendo de una cantidad fija $K \in \mathbb{R}_+$ de nafta para todo el viaje. Nuevamente este problema puede ser modelado utilizando CACM minimizando la funcion f tal que la funcion g quede acotada por la cantidad K de nafta.

2. Algoritmo exacto para la resolución de CACM

2.1. Notación

- G = Grafo al que se le quiere encontrar el CACM
- P = El camino a encontrar
- v, w = Los nodos de origen y destino del camino P a encontrar
- K = Cota superior del peso ω_1 del camino P a encontrar

2.2. Explicación detallada del algoritmo propuesto

2.2.1. Descripción

El algoritmo que se propone para encontrar la solución exacta al problema del CACM, consiste en encontrar todos los caminos simples de v a w y encontrar el camino P que tenga el menor peso ω_2 pero que ω_1 no supere a K .

Se va a realizar un backtracking que sólo chequee los caminos que no superen a K el peso de ω_1 , y que descarte los caminos en cuanto detecta que no puede mejorar una solución que ya tenga almacenada como candidato a ser la de menor peso ω_2 .

El algoritmo será una función recursiva, pasándole como argumentos el grafo, los nodos de origen (u el cual en la primer llamada va a ser v , el nodo origen del problema) y destino (w), el límite K , y un camino encontrado hasta el momento, que va desde el nodo de origen del problema (v), al nodo de origen pasado a la función (u).

En cada llamada recursiva, la función recorre todos los adyacentes a u , y por cada uno (nombre z) llama recursivamente quitando el eje (u, z) al grafo, indicándole como nodo de origen a z , y agregando al camino encontrado hasta el momento el nodo z . Luego se queda con el mejor camino que encontró entre cada uno de los adyacentes.

A su vez, antes de comenzar con todas las llamadas recursivas, se ejecuta *Dijkstra* desde el nodo u sobre los pesos ω_1 . Si la peso ω_1 del camino mínimo desde u a w tiene un valor mayor a K , entonces se retorna que no hay solución sin comenzar con la recursión, ya que los caminos que hubiesen, va a tener un peso ω_1 mayor a K (ya que el mínimo es mayor a K).

Se agrega también antes de comenzar con la recursión, se obtienen los caminos mínimos (tanto de ω_1 como de ω_2) desde el nodo de destino w al resto del grafo. Estando dentro de las llamadas recursivas, sólo avanza al siguiente nodo si el camino mínimo desde dicho nodo hacia w cumple con la cota de K en ω_1 y que sea mejor en peso ω_2 que el mejor camino ya calculado (como no son dirigidos los grafos, es igual al camino desde w a ese nodo).

Como lo que se busca son los caminos simples de v a w , estando en un momento de la iteración y agregando un nodo z al posible camino, ese nodo z no debe aparecer más adelante en el camino que se busca, por lo que no hay que pasar a volver a analizar el vértice z una vez que ya está en el camino. Es por esto que en el algoritmo se marcan los nodos que se fueron visitando para no volver a pasar sobre ellos; si se pasasen, el camino tendría 2 veces el mismo nodo y ya no sería un camino simple.

2.2.2. Algoritmo

Algorithm 1 `cacm_exacto`

Require: *grafo*: el grafo al cual se tiene que encontrar el CACM

Require: u : el nodo de origen

Require: w : el nodo de destino

Require: K : cota del peso ω_1

Require: *solucion*: posible solución que se tiene hasta el momento. Por acá se recibe un camino posible y se actualiza en cada llamada recursiva con el mejor camino encontrado. Cuando terminen todas

las llamadas recursivas, se retorna por aca el *CACM* encontrado. En la primera llamada se debe llamar con una solución que sólo contenga al nodo de partida v .

Ensure: Retorna *verdadero* si encontró un camino de u a w , *falso* si no encontró camino.

```

1: solucion_mejor  $\leftarrow$  []

2: procedure CACM_EXACTO(Grafo: grafo, Nodo: u, Nodo: w, Real: K, Camino: solucion)  $\rightarrow$  bool
3:   distancias  $\leftarrow$  DIJSKTRA_EN_W1(grafo, u)
4:   if distancias[w] > K then
5:     return falso
6:   else
7:     distancias_w1  $\leftarrow$  DIJSKTRA_EN_W1(grafo, w)
8:     distancias_w2  $\leftarrow$  DIJSKTRA_EN_W2(grafo, w)
9:     return  $\leftarrow$  CACM_EXACTO_RECURSIVO(grafo, u, w, K, distancias_w1, distancias_w2, solucion)
10:  end if
11: end procedure

12: procedure CACM_EXACTO_RECURSIVO(Grafo: grafo, Nodo: u, Nodo: w, Real: K, Real: distan-
    cias_w1[], Real: distancias_w2, Camino: solucion)  $\rightarrow$  bool
13:   if u = w then
14:     if mejor_solucion = []  $\vee$   $\omega_2(solucion) < \omega_2(solucion\_mejor)$  then
15:       mejor_solucion  $\leftarrow$  solucion_nueva
16:       return verdadero
17:     else
18:       return falso
19:     end if
20:   end if
21:   solucion_nueva  $\leftarrow$  []
22:   solucionado  $\leftarrow$  false
23:   MARCARVISITADO(grafo, u, true)
24:   for z  $\leftarrow$  ADYACENTES(grafo, u) do
25:     if  $\neg$  ESTAVISITADO(grafo, z) then
26:       peso1  $\leftarrow$   $\omega_1(u, z)$ 
27:       peso2  $\leftarrow$   $\omega_2(u, z)$ 
28:       if distancias_w1[z] + peso1  $\leq$  K  $\wedge$  ( $\omega_2(solucion\_mejor) = [] \vee$  distancias_w2[z] + peso2 +
           $\omega_2(solucion) < \omega_2(solucion\_mejor)$ ) then
29:         QUITAR_ARISTA(grafo, u, z)
30:         solucion_nueva gets solucion + [z]
31:         encuentre  $\leftarrow$  CACM_EXACTO_RECURSIVO(grafo, z, w, K - peso1, solucion_nueva)
32:         if encuentre then
33:           solucionado  $\leftarrow$  true
34:         end if
35:         AGREGAR_ARISTA(grafo, u, z, peso1, peso2)
36:       end if
37:     end if
38:   end for
39:   MARCARVISITADO(grafo, u, false)
40:   if solucionado then
41:     solucion  $\leftarrow$  solucion_mejor
42:   end if
43:   return solucionado
44: end procedure

```

2.3. Complejidad temporal para el peor caso

En el Algoritmo (1) se puede ver que depende de la cantidad de aristas de cada nodo (ya que para cada nodo, recorre todos los adyacentes), dados dos grafos con igual cantidad de nodos, el algoritmo realizará más iteraciones con el grafo que contenga más aristas. Por ende, los peores casos son los grafos completos (K_n), que son los grafos con mayor cantidad de vecinos en cada nodo.

En cada llamada recursiva, se itera sobre todos los nodos adyacentes a u , y se llama recursivamente si ese nodo (z) no fue ya marcado, es decir, si no está ya dentro del camino que se está calculando.

Antes de ejecutar la recursión, se ejecutan 3 funciones de camino mínimo *Dijkstra*, pero al ser la función recursiva con una complejidad algorítmica mucho mayor, el análisis se va a centrar en la función recursiva.

Teniendo un grafo K_n , como todos los nodos tienen $n - 1$ aristas que lo conectan con el resto de los nodos del grafo, en cada llamada recursiva se está quitando uno de los nodos que tiene disponible, por lo que reduce en 1 a la cantidad de vecinos a los que se va a volver a llamar recursivamente sea cual sea el nodo al que se avance. Ver la Figura 3.

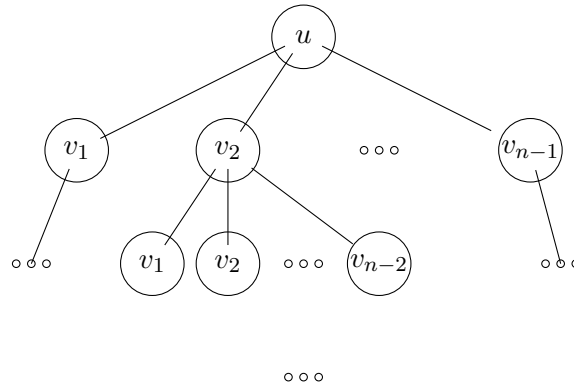


Figura 3: Árbol de llamadas recursivas. u es el nodo de origen, y v_i es el vecino i disponible para avanzar en la recursión del padre del nodo en el árbol

Si se implementa una *matriz de adyacencia*, obtener los adyacentes se realiza en $O(n)$, con n la cantidad de nodos del grafo. Marcar un nodo y saber si está marcado se puede implementar en $O(1)$ con un *array*. La copia y asignación de una solución se puede realizar acotándolo por $O(n)$ teniendo un *array* con todos los nodos que tiene el camino, siendo n (total de nodos en el grafo) la cantidad de nodos máximos que puede tener un camino simple.

Se puede plantear una función de recurrencia donde en cada llamada recursiva se está quitando un nodo posible para visitar, y acotando la cantidad de vecinos posibles del nodo por $n - 1$, ya que $d(v) \leq n - 1$

$$T(1) = 1$$

$$\begin{aligned} T(n) &= 2n' + (n - 1)[T(n - 1) + n'] = \\ T(n) &= 2n' + (n - 1) * T(n - 1) + (n - 1) * n' = \\ T(n) &= (n - 1) * T(n - 1) + (n + 1) * n' \end{aligned}$$

Donde n' es igual a la cantidad de nodos siempre, y n representa la a cantidad de nodos disponibles que hay, inicialmente $n = n'$. En el caso base, cuando solo queda un nodo por visitar, no tiene vecinos a los que recorrer.

El primer $2n'$ que aparece en la ecuación representa obtener todos los adyacentes del nodo y el guardar la solución final cuando se recorrió todos los adyacentes, $T(n - 1)$ es la llamada recursiva que se realiza $(n - 1)$ veces, y el último n' es el caso de estar mejorando la solución y en todas las llamadas tener que copiar siempre la solución, es decir, copiar $(n - 1)$ veces la nueva solución de tamaño acotado

por n' .

$$\begin{aligned}
T(n) &= (n-1) * T(n-1) + (n+1) * n' \\
&= \\
T(n) &= (n-1) * [(n-2) * T(n-2) + (n) * n'] + (n+1) * n' \\
&= \\
T(n) &= (n-1) * (n-2) * T(n-2) + (n-1) * (n) * n' + (n+1) * n' \\
&= \\
T(n) &= (n-1) * (n-2) * [(n-3) * T(n-3) + (n-1) * n'] + \\
&\quad (n-1) * (n) * n' + (n+1) * n' \\
&= \\
T(n) &= (n-1) * (n-2) * (n-3) * T(n-3) + (n-1)^2 (n-2) * n' + \\
&\quad (n-1) * (n) * n' + (n+1) * n' \\
&= \\
T(n) &= (n-1) * (n-2) * (n-3) * [(n-4) * T(n-4) + (n-2) * n'] + \\
&\quad (n-1)^2 * (n-2) * n' + (n-1) * (n) * n' + (n+1) * n' \\
&= \\
T(n) &= (n-1) * (n-2) * (n-3) * (n-4) * T(n-4) + (n-1)(n-2)^2 (n-3) * n' + \\
&\quad (n-1)^2 * (n-2) * n' + (n-1) * (n) * n' + (n+1) * n' \\
&\dots
\end{aligned}$$

Si se sigue desarrollando la ecuación, nos quedaría:

$$T(n) = (n-1)! + n' * \sum_{i=-1}^{n-2} ((n-i) \prod_{j=1}^{i+1} (n-j))$$

Como ya no hay recurrencia, y $n = n'$ en la primer llamada, entonces nos queda

$$\begin{aligned}
T(n) &= (n-1)! + n * \sum_{i=-1}^{n-2} ((n-i) \prod_{j=1}^{i+1} (n-j)) \\
&= (n-1)! + \sum_{i=-1}^{n-2} (n * (n-i) \prod_{j=1}^{i+1} (n-j)) \\
&= (n-1)! + \sum_{i=-1}^{n-2} ((n-i) \prod_{j=0}^{i+1} (n-j)) \\
&= (n-1)! + \sum_{i=-1}^{n-2} ((n-i) \frac{n!}{(n-i-2)!})
\end{aligned}$$

Ahora expandimos la suma

$$\begin{aligned}
T(n) &= (n-1)! + (n+1) \frac{n!}{(n-1)!} + (n) \frac{n!}{(n-2)!} + \dots + 3 \frac{n!}{1!} + 2 \frac{n!}{0!} \\
&\implies T(n) \in O((n+1)!)
\end{aligned}$$

Quedandonos entonces que la función de recurrencia que se planteó pertenece a la clase de complejidad $O(n!)$

2.4. Experimentacion: Mediciones de Performance

En esta sección se mostrarán resultados de complejidad temporal empírica, veremos que los resultados coinciden razonablemente con el análisis de complejidad teórica.. Para tener una mejor idea

del comportamiento del algoritmo, realizamos pruebas sobre grafos aleatorios de distintos tamaños (en cantidad de nodos), y por cada cantidad n de nodos, variamos las densidades de aristas dentro de cierto rango alrededor de una función de n , las densidades elegidas fueron:

- $m = a * n + b$. Es decir una cantidad lineal de aristas en base a los nodos. $a \in \mathbb{N}_{>1}$
- $m = \frac{n*(n-1)}{2}$. Es decir grafos cercanos o iguales a cliques de n nodos.

Nota: Como mencionamos anteriormente, las funciones son variadas en un rango, es decir, por ejemplo, para el caso de cliques, los grafos generados tienen entre $\frac{n*(n-1)}{5}$ y $\frac{n*(n-1)}{2}$ aristas para aleatorizar más la generación de grafos densos.

Nota: Los gráficos que contienen puntos rojos y una curva verde, indican, para cada valor del eje X (cantidad de nodos), los puntos rojos son los tiempos de ejecución para los diferentes valores de aristas en el rango de la familia, asimismo, la curva verde indica el promedio de estos puntos para cada X.

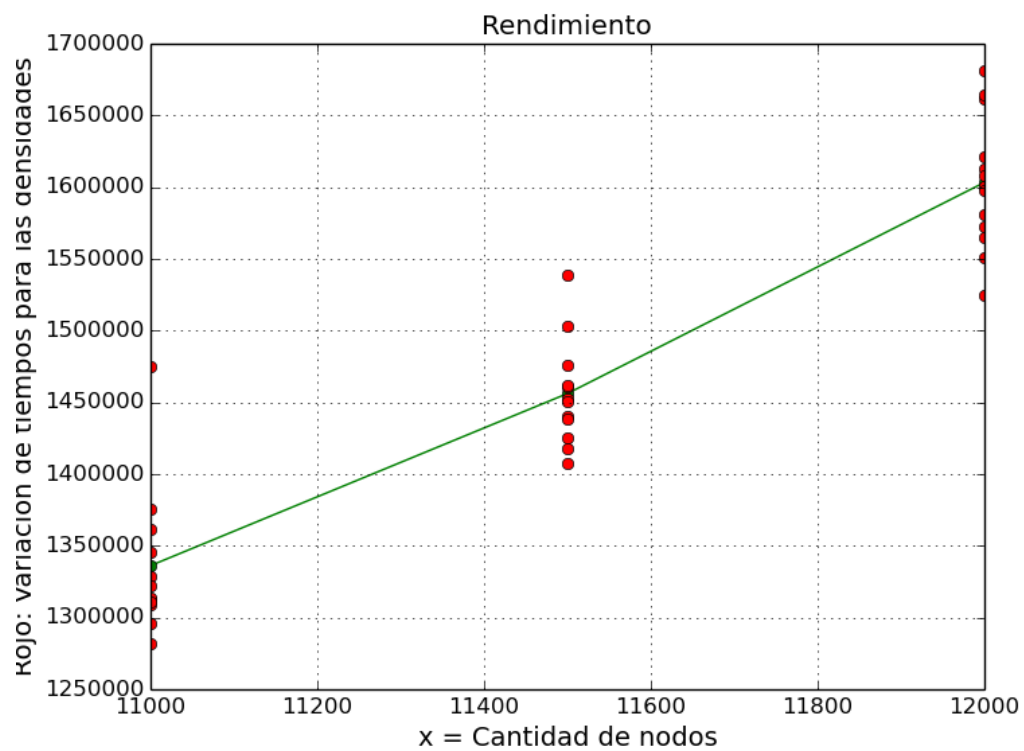
En esta primera sección dividiremos las funciones por polinomios n , n^2 , n^3 y n^4 con la mayor cantidad de nodos posible e intentaremos ver que la función se mantiene creciente. Dadas las limitaciones de nuestra computadora de benchmarking, el análisis de división por factorial se hará aparte con menos cantidad de nodos.

2.4.1. Rendimiento para grafos con densidad lineal de aristas

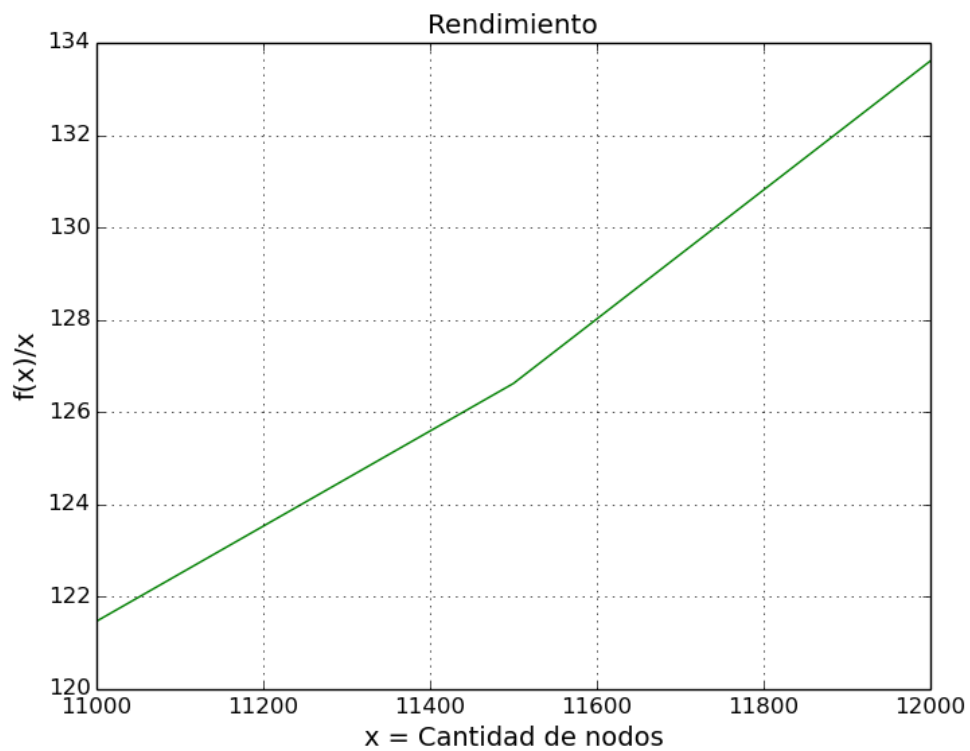
- cant nodos min = 10000
- cant nodos max = 12000
- peso maximo w1 = 250
- peso maximo w2 = 400
- step nodos = 500
- step aristas = 1000
- aristas minimas = $(n - 1)$
- aristas maximas = $(10 * n)$

Tiempo de ejecución en microsegundos para esta familia

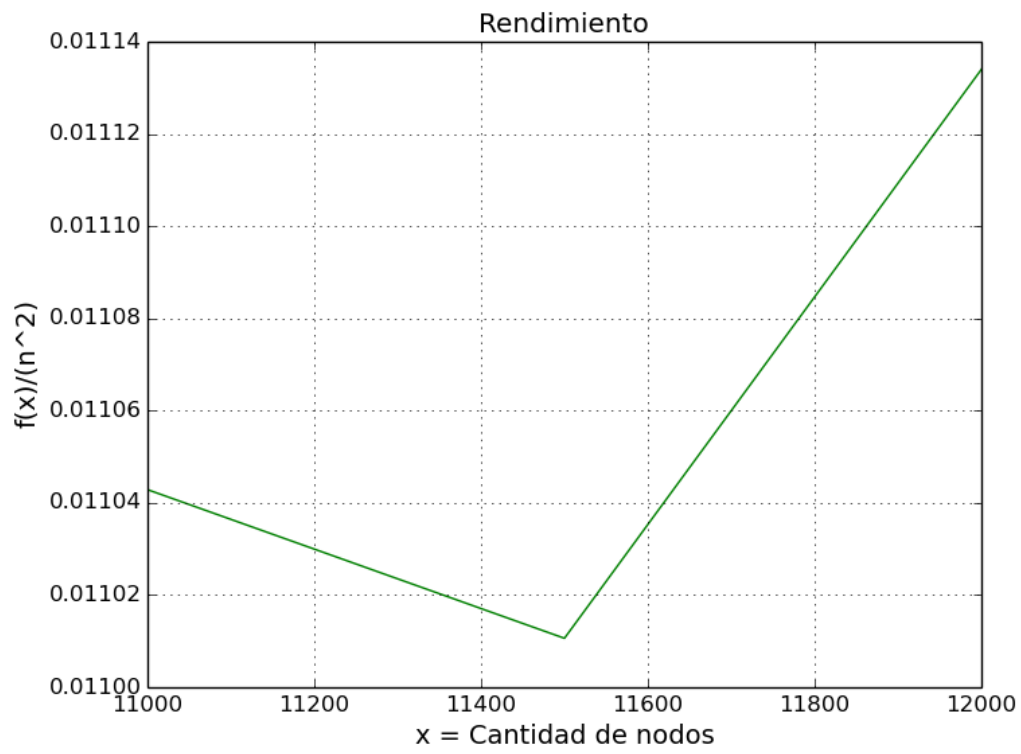
$$y = f(x)$$



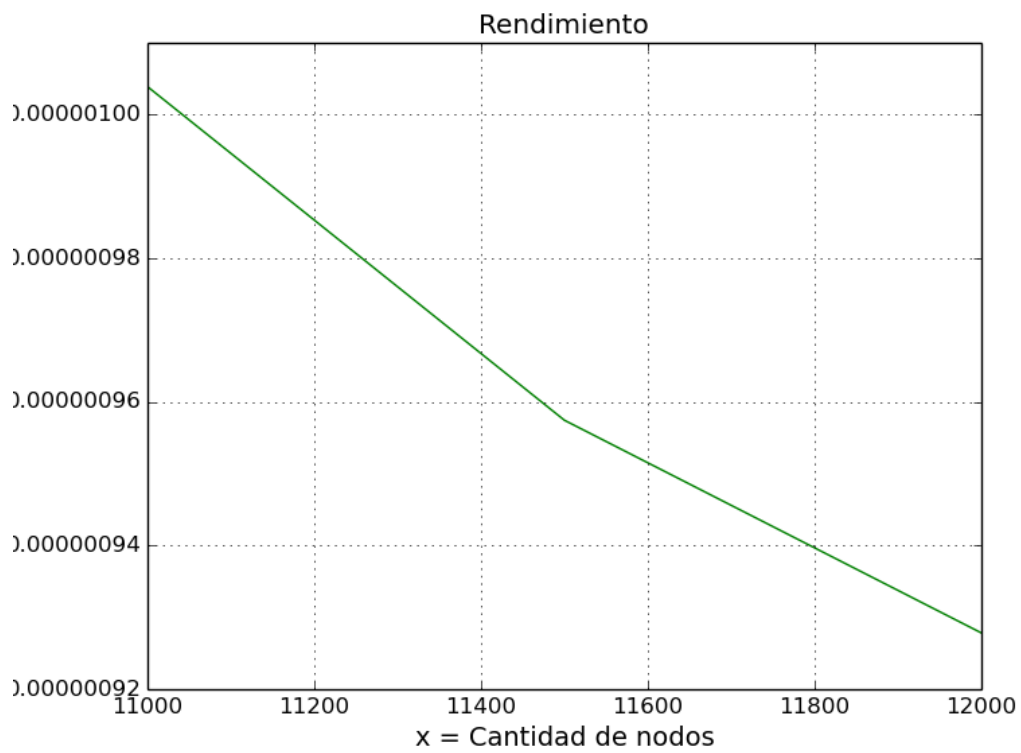
$$y = f(x)/x$$



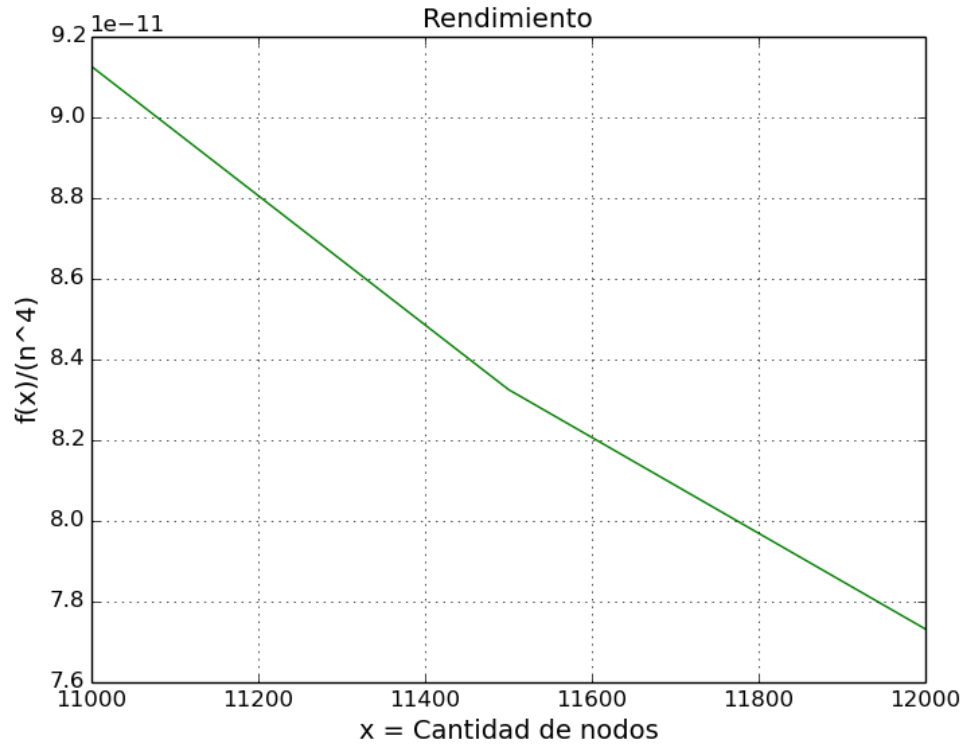
$$y = f(x)/x^2$$



$$y = f(x)/x^3$$



$$y = f(x)/x^4$$

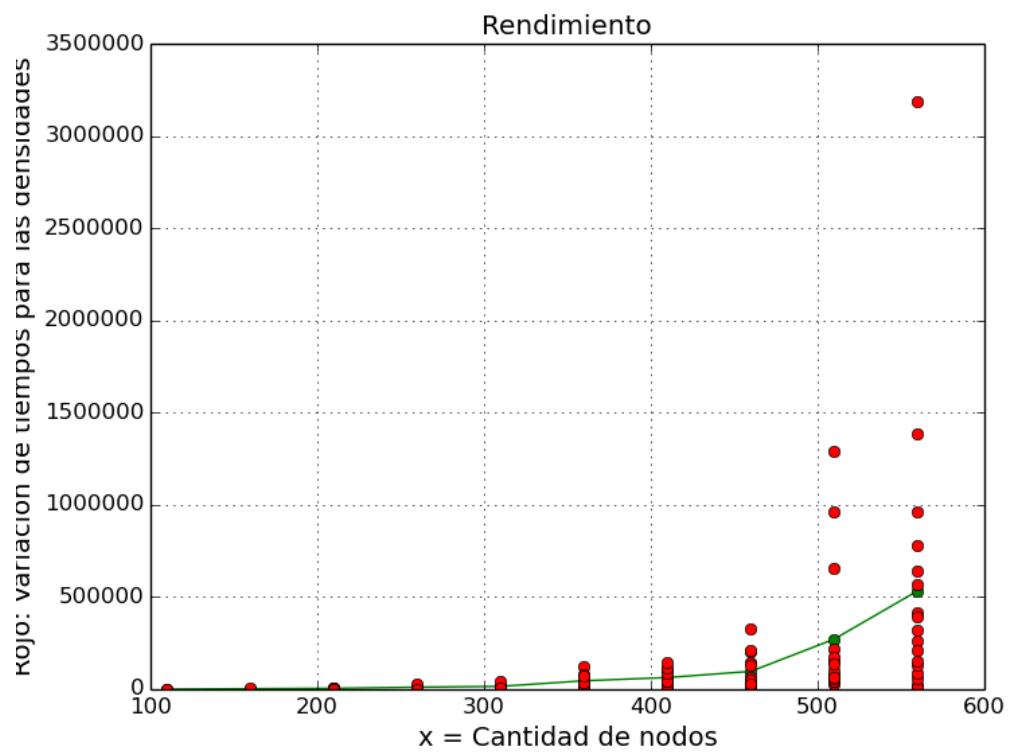


2.4.2. Rendimiento para grafos con densidad cuadratica de aristas

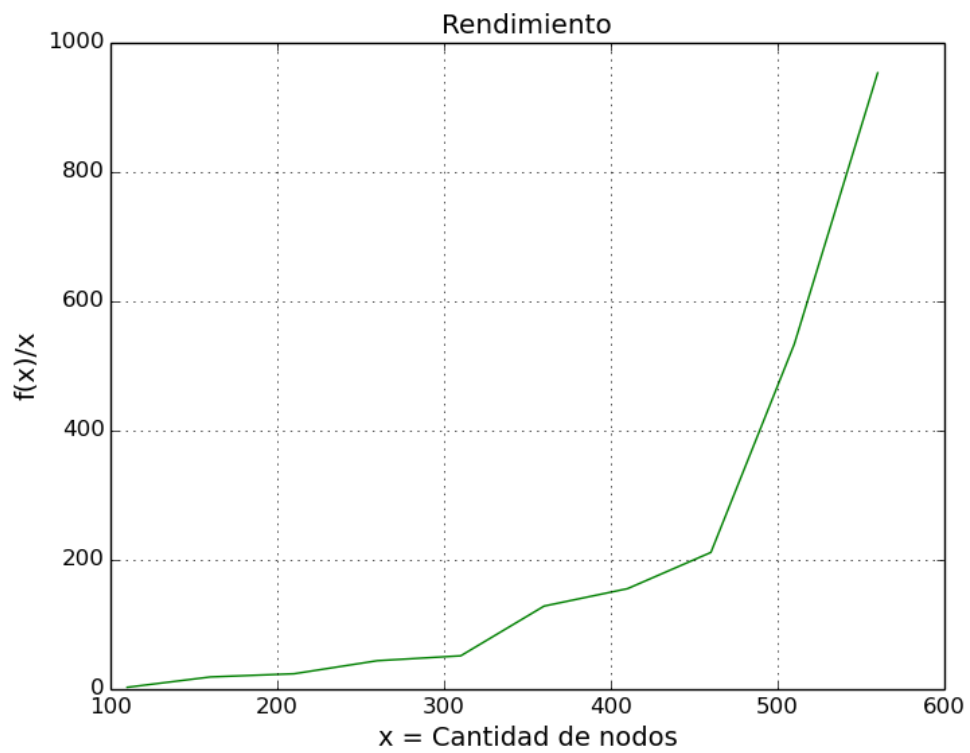
- cant nodos min = 10
- cant nodos max = 600
- peso maximo w1 = 250
- peso maximo w2 = 400
- step nodos = 50
- step aristas = 7000
- aristas minimas = $(n * (n - 1))/10$
- aristas maximas = $(n * (n - 1))/2$

Tiempo de ejecución en microsegundos para esta familia

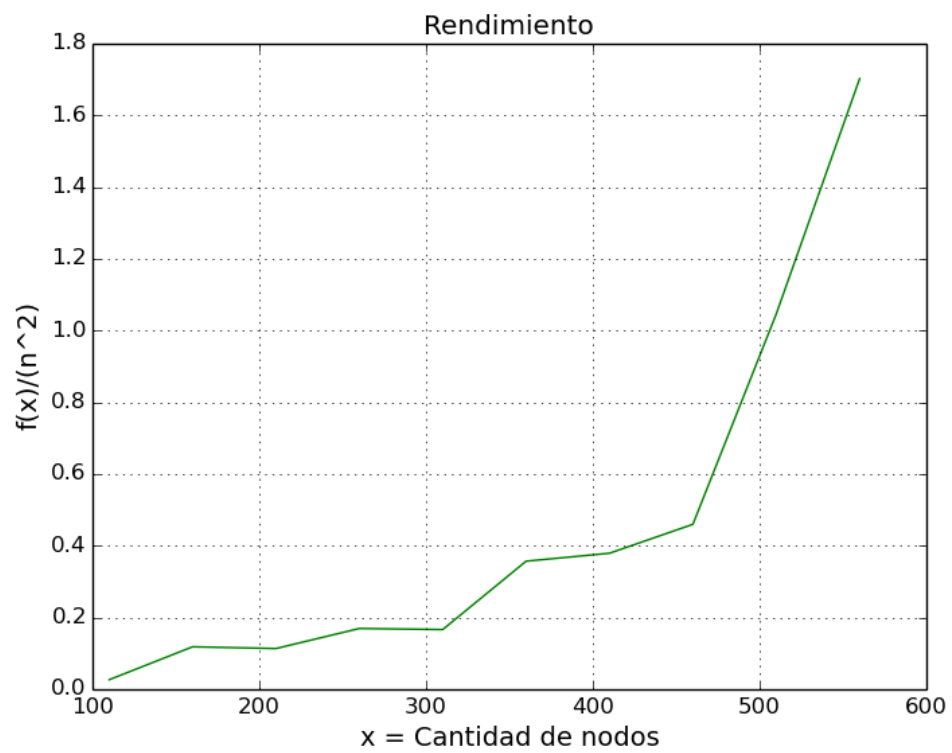
$$y = f(x)$$



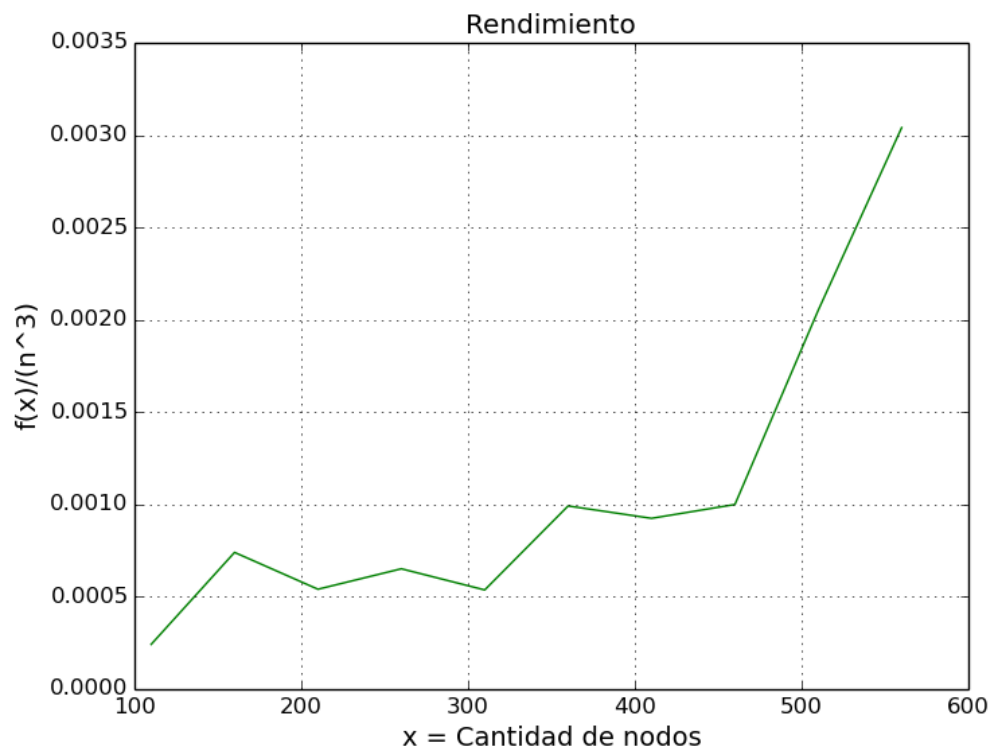
$$y = f(x)/x$$



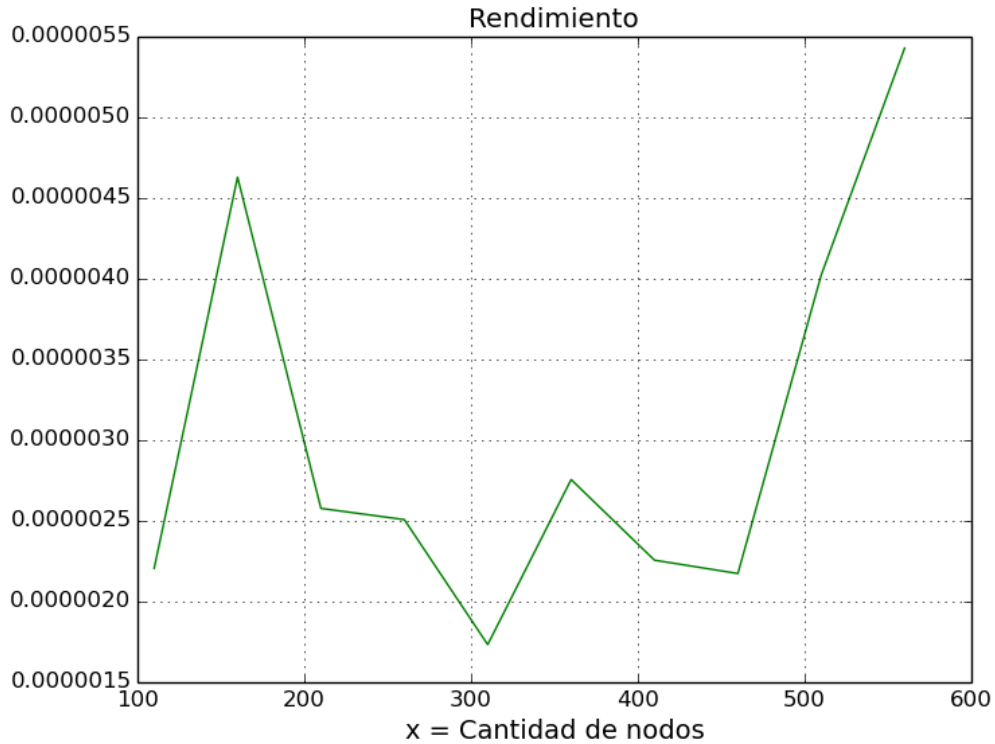
$$y = f(x)/x^2$$



$$y = f(x)/x^3$$



$$y = f(x)/x^4$$



En el caso de los grafos con densidad de aristas cuadrática, puede notarse que la curva se sigue manteniendo creciente siendo dividida por todos los polinomios de la experimentación. Si bien no es concluyente, es un indicio de que la complejidad se sitúa en un orden mayor. En los grafos con densidad de aristas lineal, la curva se convierte en constante en el orden cuadrático y decreciente en los siguientes. Suponemos esto se debe a la gran reducción del árbol de recursión por la pequeña cantidad de aristas, la disminución del grado de los nodos y la calidad de las podas. Notemos que en la complejidad del algoritmo exacto el grado de los nodos indica la cantidad de llamadas recursivas, al disminuir la densidad y estar distribuidas uniformemente las aristas, disminuye $\Delta(G)$ el grado máximo del grafo, dando una complejidad menor que la cota propuesta para el peor caso.

2.4.3. Análisis factorial

Podemos observar que para el caso de alta densidad de aristas en los grafos, al dividir la función que expresa el tiempo en función de la entrada por varios polinomios de grados 1,2,3,4, el gráfico sigue con un alto nivel de crecimiento para los últimos valores del eje X, intentamos dividir la función por factorial y realizar el gráfico $f(n/n!)$ pero dados los números muy grandes, el generador de gráficos finalizaba con overflow sin producir el gráfico. Asumimos - dada la naturaleza del problema (no se conoce algoritmo polinomial) y los resultados de la primera entrega sin podas- que cuando n tiende a infinito, la función se ajusta al análisis de complejidad teórica. No podemos testear con valores más grandes de n pues nuestro hardware no lo permite. Asimismo, la calidad de las podas influyeron muchísimo a este resultado, en la entrega anterior, pudimos probar apenas con valores muy pequeños de nodos ($n < 17$) y observamos que la complejidad era factorial efectivamente.

3. Heurística Golosa

3.1. Explicación detallada de la heurística propuesta

La heurística desarrollada consiste en una modificación del algoritmo de camino mínimo de Dijkstra.

3.1.1. Inicialización

En la etapa de inicialización se declaran cuatro vectores a utilizar:

1. CostosW2
2. Predecesores
3. CostoRecorrido
4. CostosW1

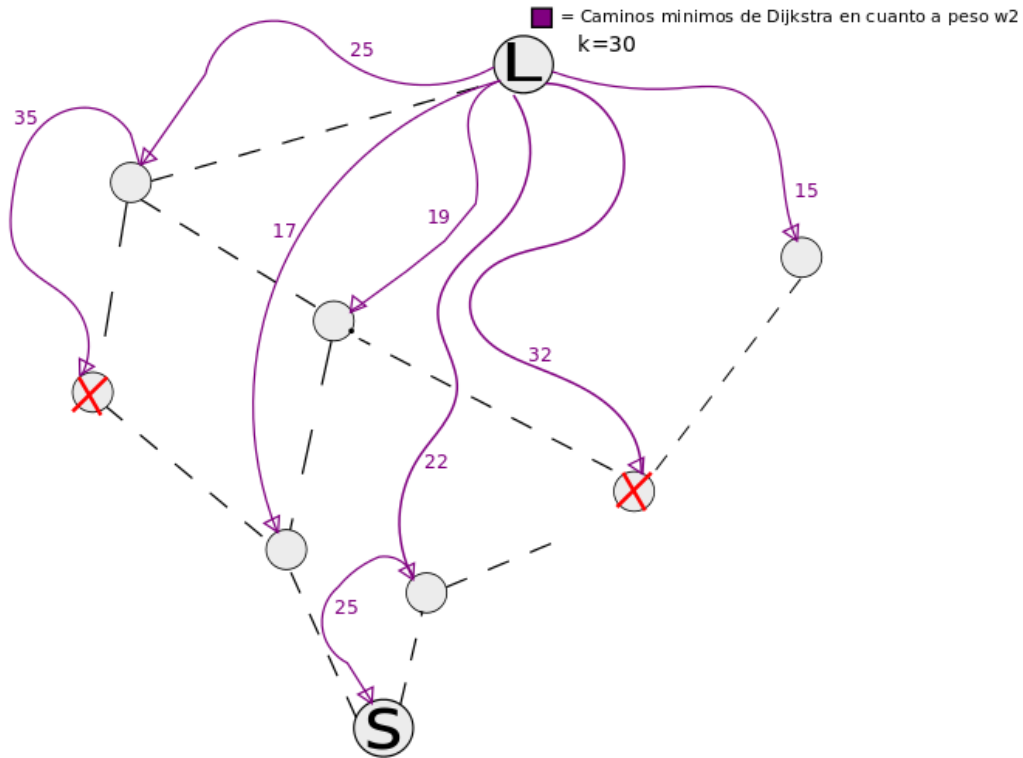
CostosW2, es análogo al vector distancias del algoritmo de Dijkstra, siendo w_2 la distancia a minimizar, guarda el costo w_2 del camino provisorio para cada nodo que va armando el algoritmo. Predecesores es análogo al vector predecesores de dicho algoritmo, indicando para cada nodo cuál es su predecesor en su camino, mientras que CostoRecorrido almacena el costo w_1 del camino recorrido.

CostosW1 almacena el costo w_1 del camino mínimo de cada nodo hasta el nodo de llegada, por razones que explicamos a continuación:

Luego de la declaración de los vectores, se calculan los caminos mínimos en cuanto a costo w_1 de todos los nodos hacia el nodo de llegada. Ya que el grafo es simple, las aristas no tienen orientación definida, por lo cual dados dos nodos v_1 y v_2 , el camino mínimo de v_1 a v_2 es igual al camino mínimo de v_2 a v_1 . Esto nos permite ejecutar una única vez el algoritmo de Dijkstra desde el nodo de llegada hasta todos los demás sobre w_1 y obtener lo buscado.

Una vez obtenidos los caminos mínimos de cada nodo a la llegada en cuanto a costo w_1 , éstos nos permiten conocer lo siguiente:

1. Al iniciar el algoritmo, aquellos nodos para los cuales no existe un camino de longitud $w_1 \leq K$ hasta el nodo llegada, los cuales nuestro algoritmo va a ignorar.
2. En medio de la ejecución del algoritmo, desde un nodo v podemos saber si el nodo x de la próxima arista (v, x) a considerar tiene al menos un camino c hasta el nodo llegada tal que, el costo del camino formado por la unión de c ($costoW1(x)$) y el recorrido hasta ahora ($costoRecorrido(v)$) es $\leq K$. En caso de no tenerlo, desde ese nodo no hay un camino válido hasta la llegada, por lo tanto nuestro algoritmo va a descartar esta arista por otra que tenga al menos un camino posible hasta la llegada. Esto lo podemos conocer, ya que la ejecución de Dijkstra inicial nos brinda el costo w_1 del camino mínimo desde el nodo x hasta la llegada, por lo tanto si $costoRecorrido(v) + costo(v, x) + Costosw_1(x) \leq K$ sabemos que existe al menos este camino válido, caso contrario descartamos la arista.



Este dibujo representa la ejecución de la inicialización de la heurística, donde el algoritmo de Dijkstra calcula los costos w_1 de los caminos mínimos de todos los nodos al nodo llegada (en violeta), y puede verse que aquellos nodos con caminos de costo mayor a K serán descartados.

3.1.2. Heurística golosa

El algoritmo goloso en sí comparte su estructura con el algoritmo de Dijkstra sobre los costos w_2 . El agregado al algoritmo es la estrategia previamente explicada, teniendo la información que nos brindan $CostosW1$ y $CostoRecorrido$, cuando sacamos un nodo v de la cola y procedemos a actualizar sus vecinos x , sólo vamos a considerar actualizar los valores $CostosW2$ y $predecesor$ para aquellos que sean factibles, es decir, aquellos para los cuales el costo w_1 del camino recorrido potencial recorrido hasta v + el costo de la arista entre los nodos (v, x) + el costo w_1 del camino mínimo de x a la llegada es menor o igual a K , dicho en términos del algoritmo: $costoRecorrido[v] + costo(v, x) + CostosW1[x] \leq K$. Esto nos asegura que el camino de un nodo nunca va a ser actualizado por un camino no factible.

La componente golosa del algoritmo, al igual que en el algoritmo de Dijkstra, es la elección en cada iteración del mínimo nodo de la cola de nodos.

3.1.3. Pseudocódigo

- 1: **procedure** HEURÍSTICA GOLOSA(grafo g , nodo u , nodo v , cota K) -> camino
- 2: $vector < int > costosw_1[n]$
- 3: $vector < int > costosw_2[n]$
- 4: $vector < int > predecesores[n]$
- 5: $vector < int > costoRecorrido[n]$
- 6: $camino\ c \leftarrow []$
- 7: $colaPrioridad\ cola \leftarrow []$

```

8:   Dijkstra(g, v, costosw1)

9:   for i from 0 to n − 1 do
10:    costosw2[i] ← ∞
11:    predecesores[i] ← NULL
12:    costoRecorrido[i] ← 0
13:   end for
14:   costosw2[u] ← 0

15:   cola.push(costosw2, u)
16:   while cola! = [] do
17:    par < costow2, nodo > actual ← cola.pop
18:    for w : adyacentes(actual2) do
19:     if costoRecorrido[actual2] + costow1Arista(actual2, w) + costosw1(w) ≤ K then
20:      if costosw2[w] > costosw2[actual2] + costow2Arista(actual, w) then
21:       costosw2[w] ← costosw2[actual2] + costow2Arista(actual, w)
22:       costoRecorrido[w] ← costoRecorrido[actual2] + costow1Arista(actual2, w)
23:       predecesor[w] = actual
24:       cola.push(costow2(w), w)
25:      end if
26:     end if
27:    end for
28:   end while
29:   c ← reconstruirCamino(predecesor, v)
30:   return c
31: end procedure

```

3.1.4. Complejidad

El algoritmo comienza declarando vectores de longitud de tamaño n , un camino de tamaño n^2 y una cola de prioridad, en total tiempo $O(n^2)$. Acto seguido ejecuta el algoritmo de Dijkstra desde nodo llegada a todos los demás, en tiempo $O((m + n)\log n)$. Se ejecuta un ciclo que deja preparados los vectores a utilizar por el ciclo goloso, en $O(n)$ y dos asignaciones de tiempo constante.

El ciclo while va a iterar a lo sumo n veces, ya que hay nodos no factibles que no serán anadidos, y aquellos que si lo son, son anadidos a lo sumo una única vez (por la demostración de Dijkstra, al elegir el nodo con costo mínimo de la cola nos aseguramos que ese va a ser su costo mínimo y no va a ser necesario actualizarlo nuevamente). En cada iteración se obtiene y elimina el mínimo elemento de la cola en $O(\log n)$. Nos queda analizar el ciclo *for* interior, que para cada nodo itera sobre su lista de adyacentes. Como cada nodo es agregado una vez, entonces cada arista en la lista de cada nodo es examinada una vez durante el transcurso del algoritmo, por lo cual este ciclo *for* itera m veces durante el transcurso del algoritmo. Dentro de este ciclo se realizan operaciones de tiempo constante excepto la operación *push*, la cual ejecuta en tiempo $O(\log n)$, con lo cual el ciclo *for* posee complejidad $O(m\log n)$.

Esto nos deja con una complejidad de: $O(n^2) + O((m + n)\log n) + O(n\log n) + O(n\log m) = O(n^2 + m\log n)$.

Usamos el tipo *set* de la librería STL de c++, la cual nos asegura estas complejidades:

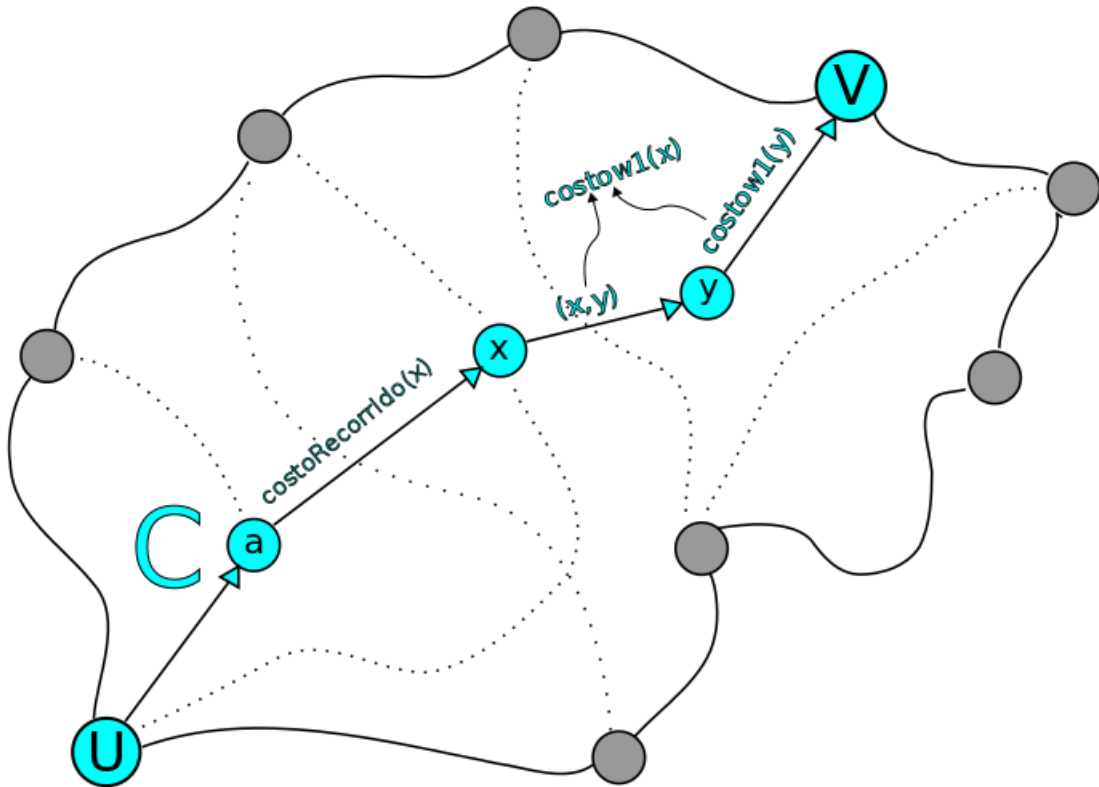
1. <http://www.cplusplus.com/reference/set/set/>
2. <http://www.cplusplus.com/reference/set/set/begin/>
3. <http://www.cplusplus.com/reference/set/set/erase/>

3.1.5. Solución Factible

A continuación, vamos a demostrar que en caso de haber algún camino factible desde la salida a la llegada (de costo w_1 total $\leq K$), la heurística siempre va a encontrar un camino factible y por lo tanto, devolver una solución.

Consideremos el camino C , de costo w_1 mínimo del nodo de salida u al nodo de llegada v . Supongamos que este existe y además su costo total es menor a la cota K . Supongamos que el algoritmo no puede encontrar un camino factible, y lleguemos a un absurdo. Tomemos a x , como el último nodo $\in C$ para el cual el algoritmo puede actualizar sus valores, es decir, el último nodo para el cual tuvo algún vecino a tal que $\text{costoRecorrido}[a] + \text{costow}_1\text{Arista}(a, x) + \text{costosw}_1(x) \leq K$ y el algoritmo actualizó valores.

Sabemos que este nodo existe ya que existe C un camino factible de costo mínimo $w_1 \leq K$ de u a v , por lo tanto este nodo es al menos el segundo nodo de C , ya que sabemos que $\text{costoRecorrido}[u] + \text{costow}_1\text{Arista}(u, \text{segundo}) + \text{costosw}_1(\text{segundo}) \leq K$, al ser $\text{costoRecorrido}[u] = 0$ y $\text{costow}_1\text{Arista}(u, \text{segundo}) + \text{costosw}_1(\text{segundo}) \leq K$ necesariamente ya que es un subcamino de C , y que el problema de camino mínimo cumple el principio de optimalidad de Bellman, y el camino mínimo de u a v puede ser descompuesto como dos caminos $u - \text{segundo}$ y $\text{segundo} - v$ que a su vez son mínimos.

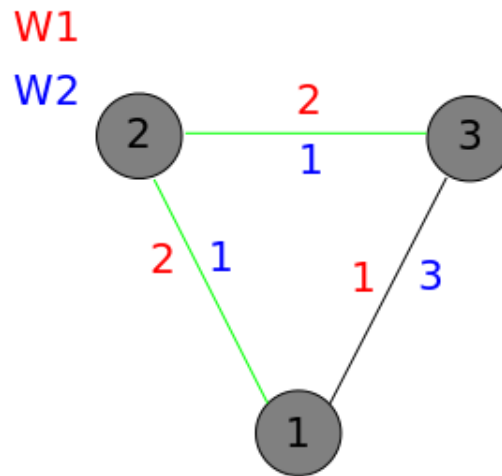


Ahora consideremos el nodo siguiente del camino, llamémosle y . y es el primer nodo del camino C para el cual el algoritmo no puede actualizarlo ni insertarlo en la cola. Pero sabemos que x es adyacente a y , y sabemos que para x , por haber sido actualizado por el algoritmo vale $\text{costoRecorrido}[x] + \text{costosw}_1(x) \leq K$. Y por no haber podido ser actualizado y desde x , vale $\text{costoRecorrido}[x] + \text{costow}_1\text{Arista}(x, y) + \text{costosw}_1(y) > K$. Pero esto nos deja en un absurdo, ya que $\text{costosw}_1(x) = \text{costow}_1\text{Arista}(x, y) + \text{costosw}_1(y)$ exactamente, ya que ambos nodos forman parte de C , y por principio de optimalidad al sere C el camino mínimo, el camino mínimo de x a la llegada va a ser el subcamino de C de x a la llegada, del cual forma parte y y la arista entre x e y .

Llegamos a un absurdo al suponer que existe el camino C , y que el algoritmo en algún punto para un nodo de C no puede actualizarlo formando un camino factible. Llegamos a la conclusión de que, o bien no existe ningún camino factible en el grafo, o existe al menos uno y el algoritmo lo devuelve.

3.2. Nivel de optimalidad de las soluciones

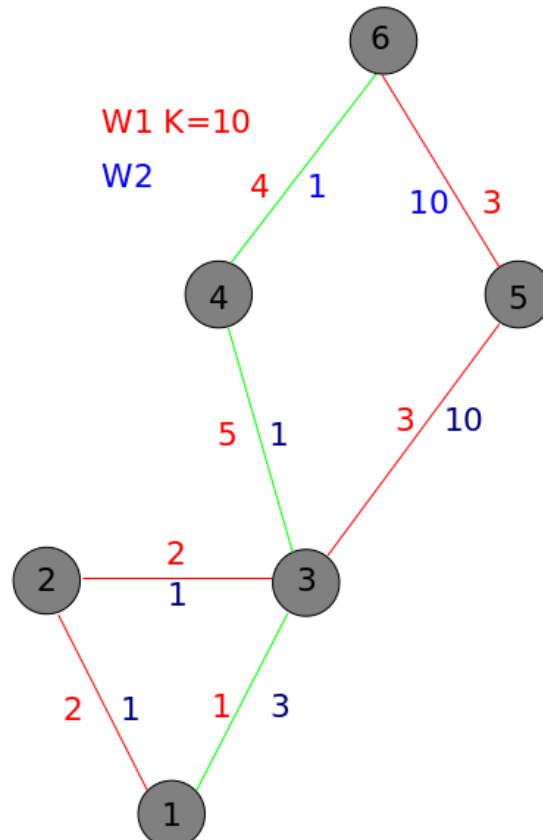
Consideremos el siguiente grafo: $Origen = 1$, $Destino = 3$, $K = 10$.



Salida golosa:

4 2 3 1 2 3

La salida de nuestro algoritmo nos devuelve el camino $1 \rightarrow 2 \rightarrow 3$ (para un K no restrictivo), ya que la decisión golosa toma el mínimo de la cola en cada iteración y actualiza el nodo 3 con el nodo 2 ya que el camino $1 \rightarrow 2$ (en rojo) tiene menos costo w_2 que la arista $(1, 3)$ (en verde). Pero en este caso particular puede notarse como prima la decisión golosa en el algoritmo, sin tener en cuenta que este camino es mucho más costoso en cuanto a w_1 que la arista. Si el grafo tuviese más nodos adyacentes al nodo 3, el haber elegido el camino va a restringir los caminos que va a poder tomar el algoritmo en las siguientes iteraciones. Por ejemplo, si el grafo fuese: $Origen = 1$, $Destino = 6$, $K = 10$.



Salida exacto: 10 5 4 1 3 4 6

Salida golosa: 10 22 5 1 2 3 5 6

Tenemos en color rojo la salida del algoritmo y en color verde el camino óptimo. Vemos como la desición golosa inicial restringió claramente la elección de caminos posterior, forzando a tomar el camino mucho más pesado que le permite llegar a destino sin superar la cota, el cual podemos hacerlo arbitrariamente más costoso en cuanto a w_2 y alejar la solución de nuestro algoritmo de la óptima tanto como queramos, aumentando los pesos w_2 o estirando el camino $3- > 5- > 6$ cuanto queramos (sin pasarnos de la cota K).

3.3. Experimentacion: Mediciones de Performance

En esta sección se mostrarán resultados de complejidad temporal empírica, veremos que los resultados coinciden razonablemente con el análisis de complejidad teórica.. Para tener una mejor idea del comportamiento del algoritmo, realizamos pruebas sobre grafos aleatorios de distintos tamaños(en cantidad de nodos), y por cada cantidad n de nodos, variamos las densidades de aristas dentro de cierto rango alrededor de una funcion de n , las densidades elegidas fueron:

- $m = a * n + b$. Es decir una cantidad lineal de aristas en base a los nodos. $a \in \mathbb{N}_{>1}$
- $m = \frac{n*(n-1)}{2}$. Es decir grafos cercanos o iguales a cliques de n nodos.

Nota: Como mencionamos anteriormente, las funciones son variadas en un rango, es decir, por ejemplo, para el caso de cliques, los grafos generados tienen entre $\frac{n*(n-1)}{5}$ y $\frac{n*(n-1)}{2}$ aristas para aleatorizar mas la generacion de grafos densos.

Nota: Los graficos que contienen puntos rojos y una curva verde, indican, para cada valor del eje X(cantidad de nodos), los puntos rojos son los tiempos de ejecucion para los diferentes valores de aristas en el rango de la familia, asimismo, la curva verde indica el promedio de estos puntos para cada X.

Nota: Para verificar que se trata de una curva cuadratica dividimos las funciones por n^1 , n^2 , n^3 , n^4 y como en los trabajos practicos anteriores concluimos de que curva se trata.

3.3.1. Consideraciones acerca de la complejidad teórica

En la sección de análisis de complejidad del algoritmo concluimos acotando la complejidad en $O(n^2 + m \log n)$. Sin embargo, como informamos en la sección anterior, nos vamos a limitar a dividir las funciones por potencias de n de 1 a 4. Esto se debe a, además de una mayor simplicidad, a que en el caso de prueba con $m = a * n + b$, m es lineal en cuanto a n , por lo tanto la complejidad se aproximaría a $n^2 + n \log n$, con lo cual quedará en el orden cuadrático, y consideramos que va a bastar con considerarlo de este orden para realizar el análisis de las divisiones. Por otro lado en el caso $m = n * (n - 1) / a$, m es cuadrático en cuanto a n , y la complejidad se aproximará a $n^2 + n^2 \log n$, lo cual es del orden $O(n^2 \log n)$. Pero, considerando que como máximo en nuestra experimentación usamos $n = 2000$, $\log(2000) \approx 11$, lo cual es casi despreciable, por lo cual también consideraremos al algoritmo en el orden cuadrático para realizar el análisis.

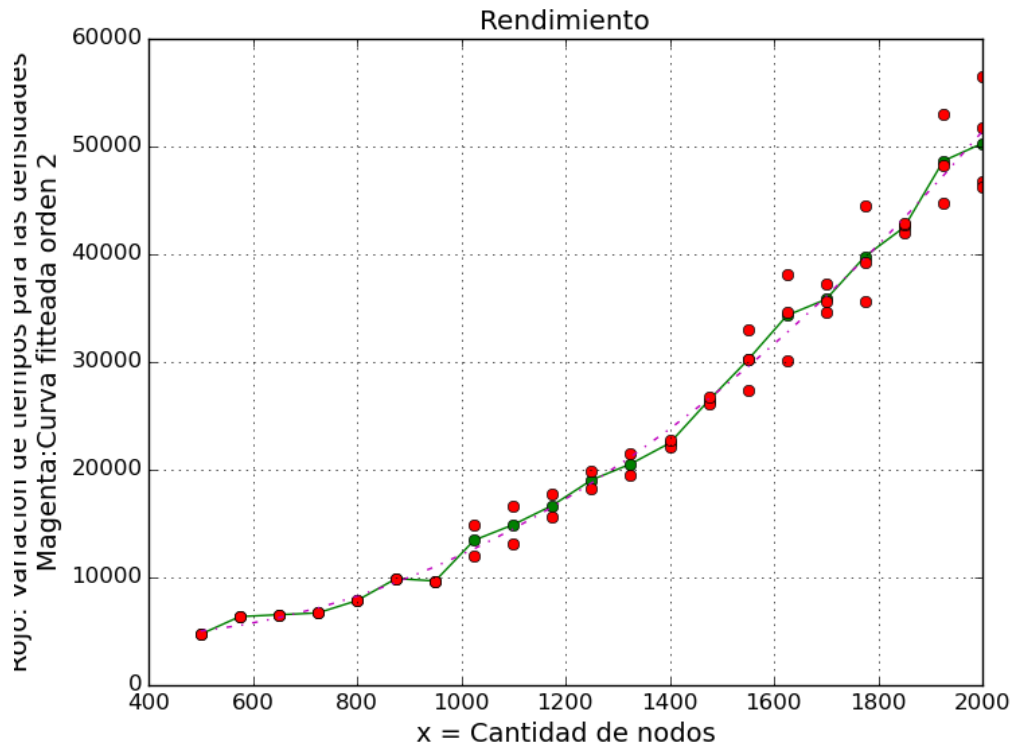
3.3.2. Rendimiento para grafos con densidad lineal de aristas

- cant nodos min = 50
- cant nodos max = 2000
- peso maximo $w_1 = 250$

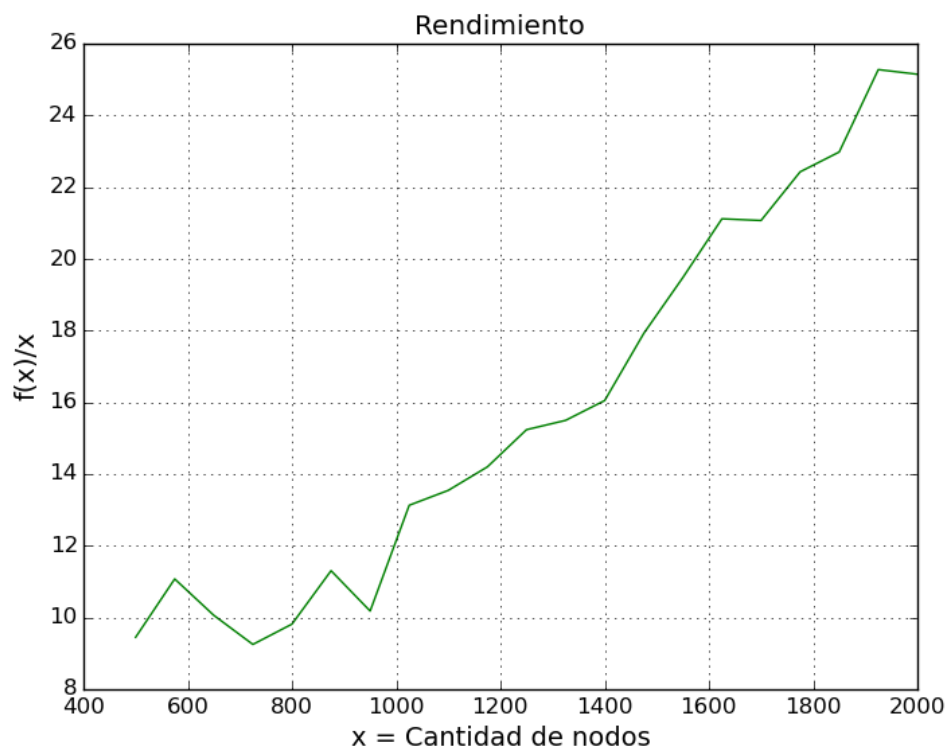
- peso maximo $w_2 = 400$
- step nodos = 75
- step aristas = 4500
- aristas minimas = $n - 1$
- aristas maximas = $10 * n$

Tiempo de ejecución en microsegundos para esta familia

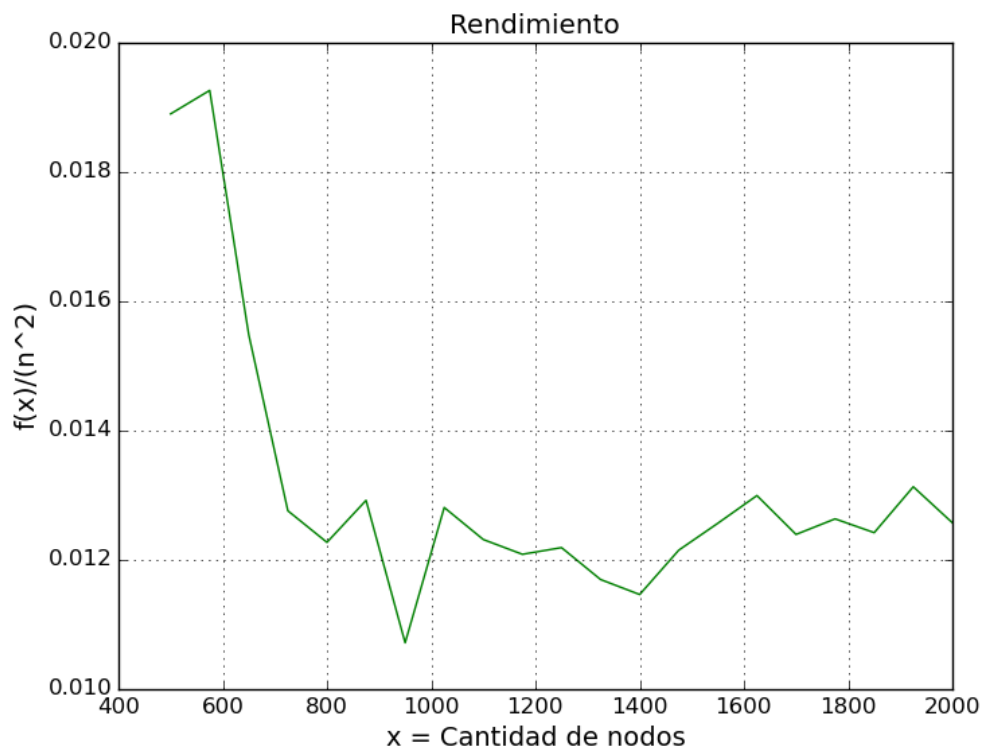
$$y = f(x)$$



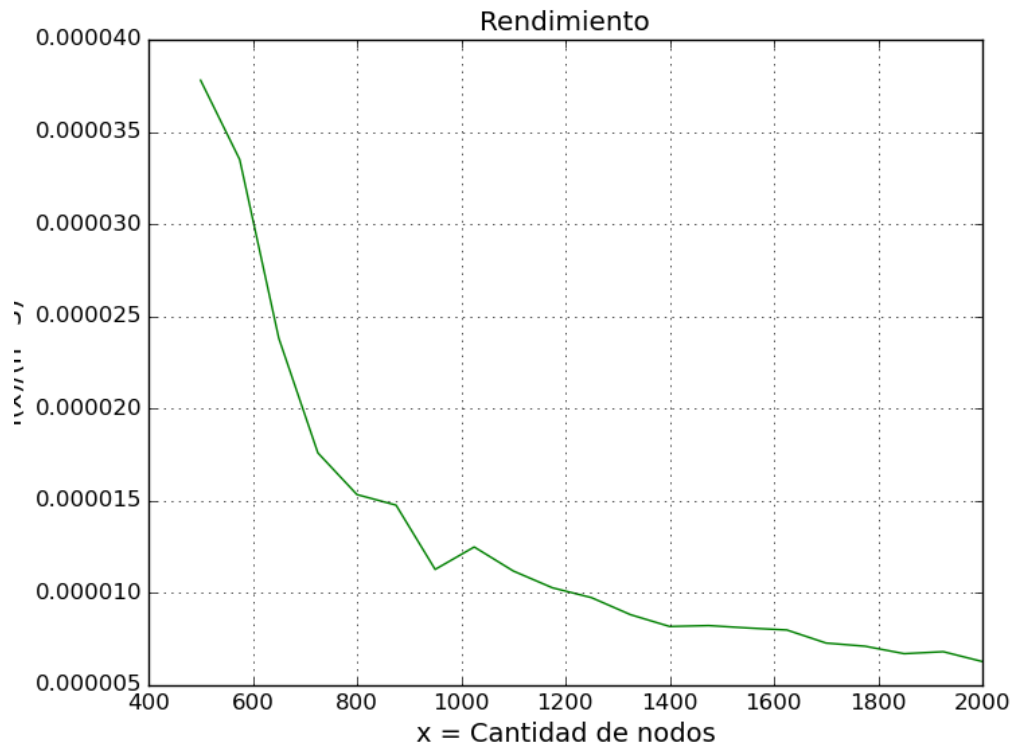
$$y = f(x)/x$$



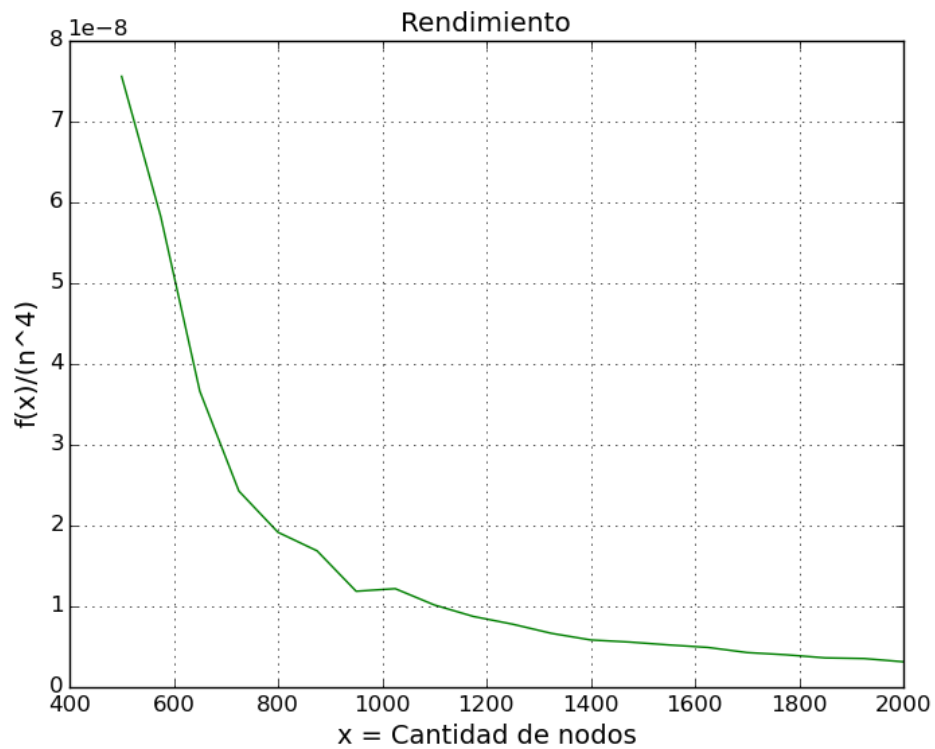
$$y = f(x)/x^2$$



$$y = f(x)/x^3$$



$$y = f(x)/x^4$$



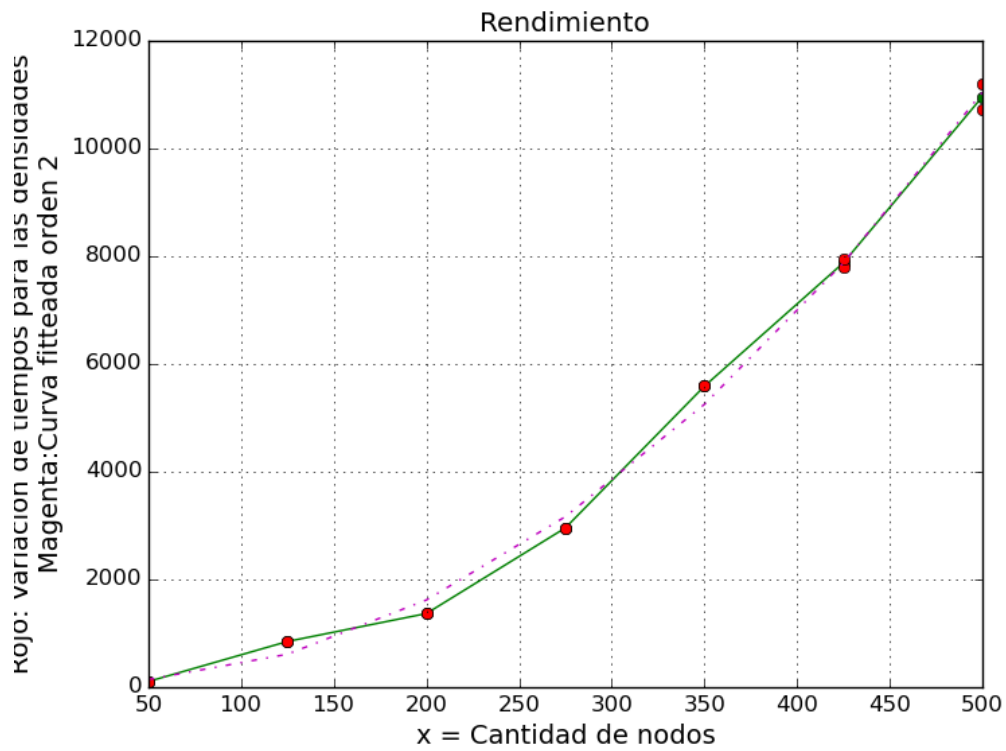
3.3.3. Rendimiento para grafos con densidad cuadratica de aristas

- cant nodos min = 5
- cant nodos max = 500
- peso maximo w1 = 250

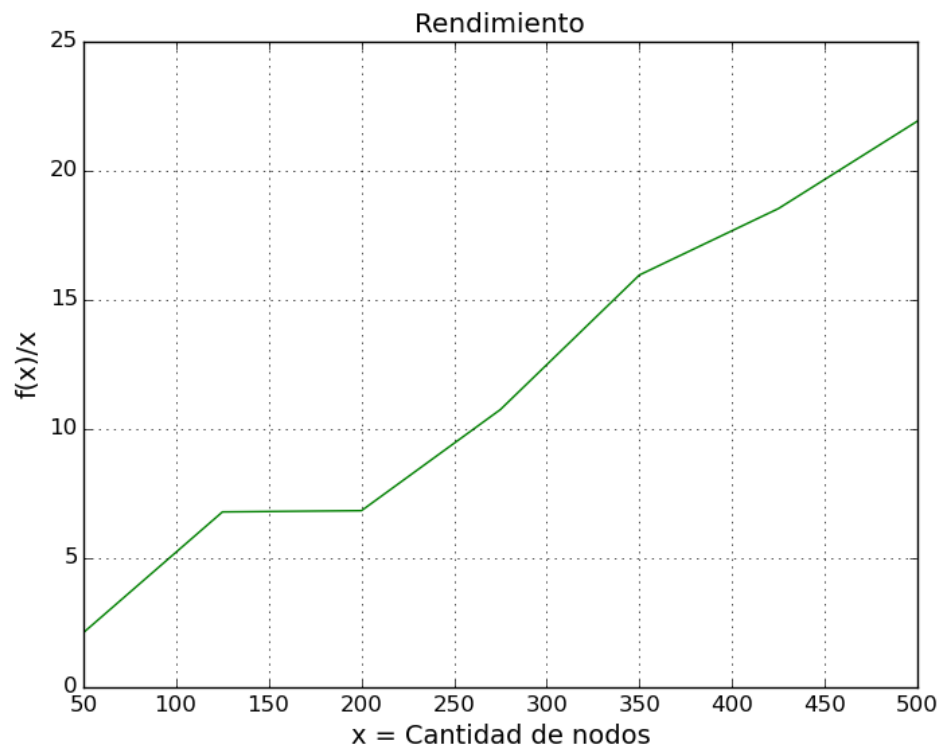
- peso maximo $w_2 = 400$
- step nodos = 75
- step aristas = 4500
- aristas minimas = $\frac{n*(n-1)}{9}$
- aristas maximas = $\frac{n*(n-1)}{7}$

Tiempo de ejecución en microsegundos para esta familia

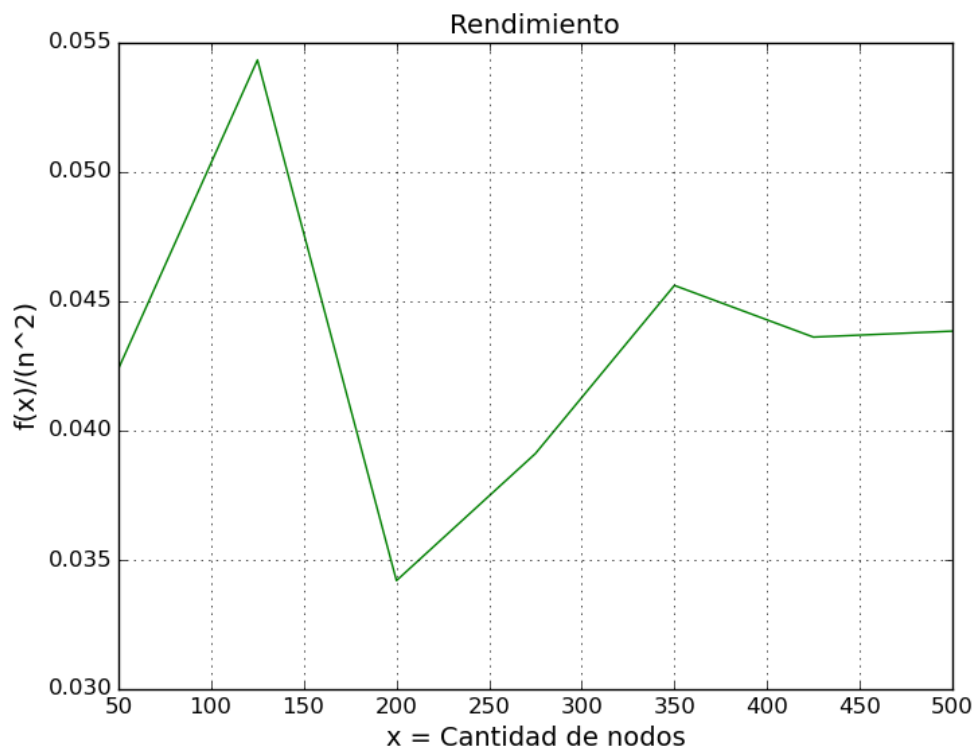
$$y = f(x)$$



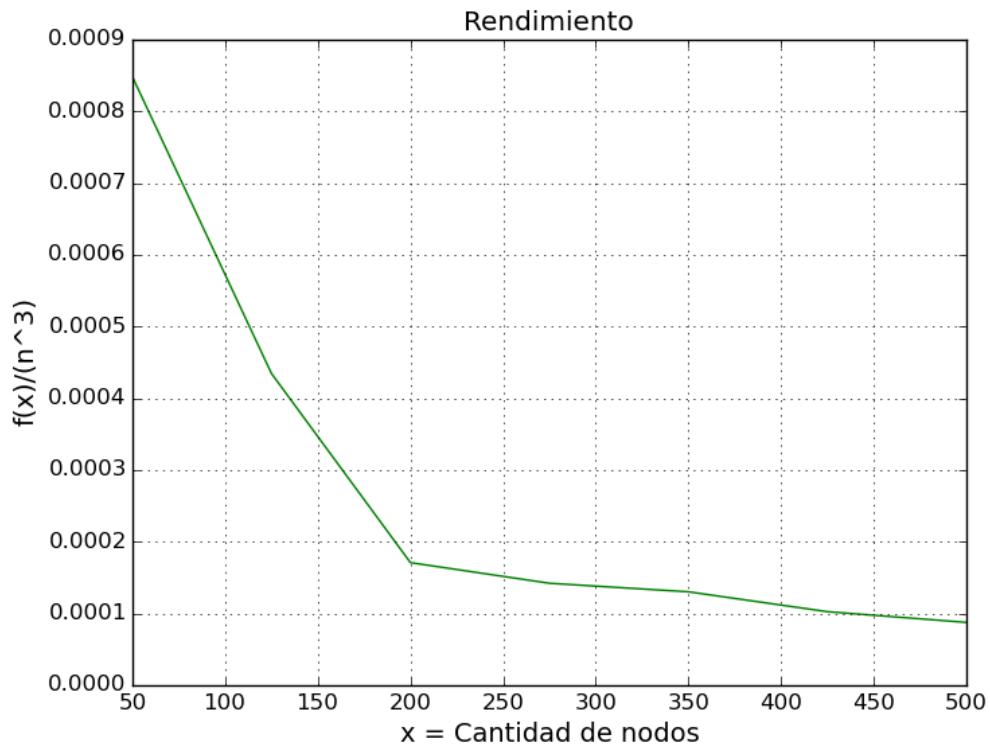
$$y = f(x)/x$$



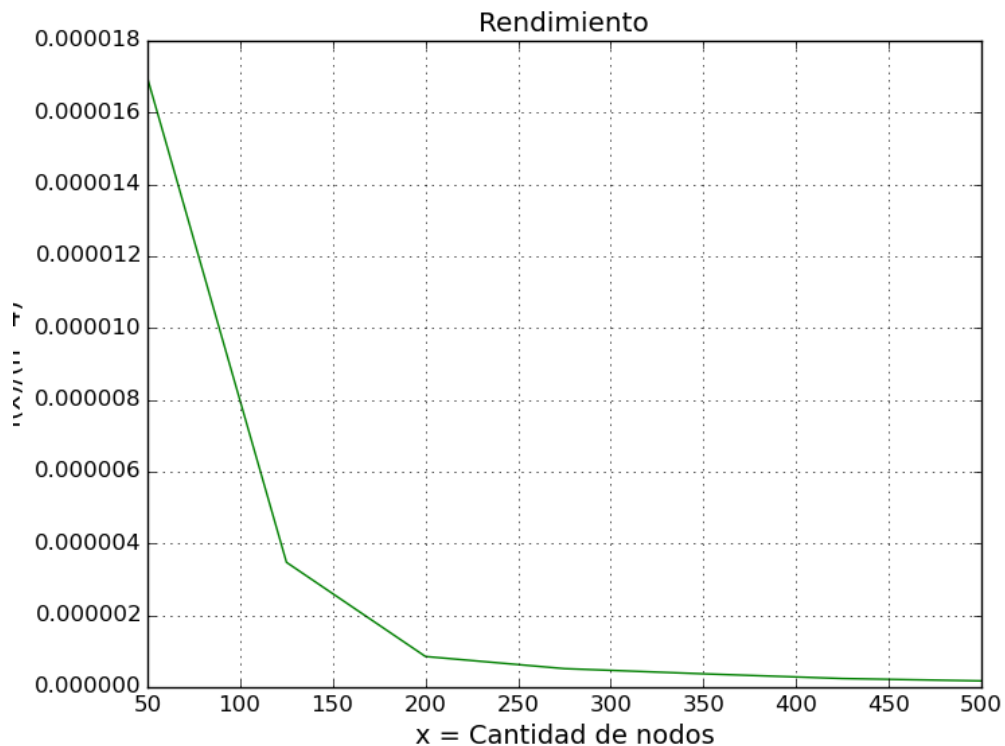
$$y = f(x)/x^2$$



$$y = f(x)/x^3$$



$$y = f(x)/x^4$$



3.3.4. Conclusión

Como podemos ver en ambos casos, $f(x)$ es una función creciente, $f(x)/x$ también es creciente, asemejándose a una curva lineal, y $f(x)/x^2$ deja de ser creciente para asemejarse a una constante en un intervalo entre 0,04 – 0,01 en ambos casos. $f(x)/x^3$ y $f(x)/x^4$ dan como resultado valores muy cercanos al 0, lo cual es el resultado de dividir una constante por la variable de experimentación (tiende a 0).

Los experimentos nos dan indicios de complejidad cuadrática, y esto se condice con la complejidad teórica $O(n^2 + m \log n)$ planteada acotada a grafos con m lineal ($O(n^2)$) y cuadrática ($O(n^2 * \log n)$ con $\log(n) \approx 11$ como máximo para $n = 2000$).

3.4. Experimentacion: Mediciones de Optimalidad de las soluciones obtenidas con esta heurística

En esta sección nos vamos a dedicar al análisis de la calidad de las soluciones, es decir, dados conjuntos de grafos aleatorios, vamos a examinar el porcentaje de soluciones óptimas obtenidas por esta heurística y realizaremos algunos cálculos estadísticos acerca de la lejanía promedio de las soluciones obtenidas con respecto a la solución exacta en los grafos del conjunto de instancias de las muestras. Para esto, se presentarán 2 experimentos, los cuales representan 2 densidades de grafos generados aleatoriamente.

Nota: La lejanía entre 2 soluciones se mide haciendo el siguiente cálculo: $100 * (\frac{\text{solucionHeuristica}}{\text{solucionOptima}} - 1)$
Nota: Los cálculos estadísticos (promedio y desviación estándar) se realizan sobre la lista de resultados obtenida de la ejecución secuencial y el cálculo de la lejanía mencionado aquí arriba para cada uno de los algoritmos (exacto y heurística) sobre cada instancia del conjunto de pruebas.

3.4.1. Grafos aleatorios de baja densidad de aristas

Parametros del experimento:

- Cantidad de grafos analizados: 25
- Cantidad mínima de nodos: 100
- Cantidad máxima de nodos: 2200
- Rango peso w_1 : [0..250]
- Rango peso w_2 : [0..400]
- Cota de w_1 : 200
- Cantidad mínima de aristas: $n - 1$
- Cantidad máxima de aristas: $10 * n$

Resultados del analisis:

Heurística da la solución óptima: 100.0% de los casos
 Lejanía promedio de la heurística a la solución óptima: 0%
 Desv. estándar de la lejanía entre las soluciones: 0
 Mínima lejanía entre bqlocal y exacta: 0
 Máxima lejanía entre bqlocal y exacta: 0

Realmente podemos observar una efectividad impecable para grafos de baja densidad. La heurística golosa resultó ser excelente para casi todos los casos de poca densidad (tengamos en cuenta que esto es un muestreo de grafos, no representan todos los grafos de baja densidad.)

3.4.2. Grafos aleatorios de alta densidad de aristas

Parametros del experimento:

- Cantidad de grafos analizados: 48
- Cantidad mínima de nodos: 10
- Cantidad máxima de nodos: 300
- Rango peso w_1 : [0..250]

- Rango peso w_2 : $[0..400]$
- Cota de w_1 : 200
- Cantidad minima de aristas: $\frac{n*(n-1)}{3}$
- Cantidad maxima de aristas: $\frac{n*(n-1)}{2}$

Resultados del analisis:

Heuristica da la solucion optima: 79.166% de los casos Lejania promedio de la heuristica a la solucion optima: 14.622 % Desv. estandar de la lejania entre las soluciones: 63.0996 Minima lejania entre bqlocal y exacta: 0 Maxima lejania entre bqlocal y exacta: 433.300%

Para grafos de alta densidad, ya la heurística comienza a fallar, pero de forma relativamente leve, solo en un 20.833 % de los casos, consideramos que un 14.622 % de lejania en las soluciones provistas por la heuristica respecto a la solución óptima sería aceptable en algunas aplicaciones que no requieran una solución estrictamente optima dada la mejora en tiempo que se obtiene utilizando la heurística. (Recordemos que la solución exacta implementada tiene complejidad no polinomial). Con respecto a la desviación estandar y la máxima lejania podemos ver que hay casos donde la solución se aleja bastante, un 433 % del optimo, pero dada la desviación de 63.0996 vemos que no es algo que ocurra en forma general.

Como conclusión pensamos que es una heurística bastante bien lograda dados estos valores de optimalidad y su complejidad temporal.

4. Heurística de búsqueda local

Para encontrar soluciones aproximadas al problema de optimizar una función $f : S \rightarrow \mathbb{R}$ definida sobre un conjunto S de instancias, la heurística de búsqueda local consiste en tomar un elemento $e \in S$ y buscar entre los elementos “ceranos” a e uno e' en el cual el valor que toma la función sea mejor ($f(e') < f(e)$ ó $f(e') > f(e)$ dependiendo de si se busca mínimo a o máximo). El concepto de cercanía o vecindad se puede definir según cualquier criterio que se crea conveniente de forma tal que evaluar la vecindad sea barato. Se busca en lo posible que las vecindades de cada punto sean pequeñas para que se pueda encontrar rápidamente un vecino mejor en caso de que exista. La heurística itera el procedimiento de optimización local creando una sucesión de soluciones aproximadas (X_0, X_1, \dots, X_m) en S donde el elemento X_m tiene la propiedad de ser un extremo local de la función f según el criterio definido de vecindad.

4.1. Criterios de vecindad

Para el problema enunciado del TP, el conjunto S es el conjunto de caminos entre el nodo de partida y el de llegada. Debido a que las w_1 y w_2 son funciones no negativas, el camino acotado de costo mínimo estará en $S' \subseteq S$, el subconjunto de todos los caminos simples. Por lo tanto solo se considerará S' como conjunto de instancias.

Establecemos la siguiente notación: $C = (v_1, \dots, v_m)$ será un camino y A_c el conjunto de aristas que lo conforman. Dos caminos $C = (v_1, \dots, v_n)$ y $C' = (v'_1, \dots, v'_m)$ en S' son vecinos $C \sim C'$ en su vecindad N_j correspondiente si se cumple su condicion asociada.

Definiremos a continuacion varias vecindades:

1. $N_1(C) : \exists v \notin C \mid A_{c'} = A_c - \{(v_i, v_{i+1}), (v_{i+1}, v_{i+2})\} + \{(v_i, v), (v, v_{i+2})\}$
Este criterio consiste en reemplazar un nodo intermedio en una tripla, tal que las sumas de los costos w_2 de las nuevas aristas sea menor que los costos de las aristas que existian originalmente, teniendo en cuenta tambien, que la mejora no implique que los nuevos costos w_1 sobrepasen la cota $K \in \mathbb{R}_{>0}$ establecida en el problema.
2. $N_2(C) : \exists v \notin C \mid A_{c'} = A_c - \{(v_i, v_{i+1})\} + \{(v_i, v), (v, v_{i+1})\}$
Este criterio consiste en la inserción de un nodo v adyacente en comun a v_i y v_{i+1} entre ellos, de forma tal que disminuya el costo w_2 en el sendero entre $v_i \rightarrow v_{i+1}$ pero que nuevamente no se sobrepase la cota sobre w_1 establecida para el problema.
3. $N_3(C) : A_{c'} = A_c - \{(v_i, v_{i+1}), (v_{i+1}, v_{i+2})\} + \{(v_i, v_{i+2})\}$
Este criterio consiste en la eliminación de un nodo intermedio entre una tripla consecutiva de nodos, siendo reemplazado por una arista existente en el grafo que conecte directamente los nodos $v_i \rightarrow v_{i+1}$, de forma tal que la contracción del sendero entre $v_i \rightarrow v_{i+1}$ disminuya el costo w_2 pero tampoco sobrepase la cota sobre w_1 estipulada en el problema.
4. $N(C) : N_1 \cup N_2 \cup N_3$
Se trata de la union de todos los criterios anteriores, una exploracion completa de esta vecindad en una iteración consiste en buscar las mejores modificaciones de las vecindades previas y la aplicacion de la mejor de ellas sobre la solucion actual de dicha iteracion.

Nota: $C = C'$ se considera que son caminos vecinos en todas las vecindades. Además de los criterios de cada vecindad, debe valer, que el nuevo camino resultante C' sea factible, es decir, respete la cota de w_1 de K luego de ser modificado.

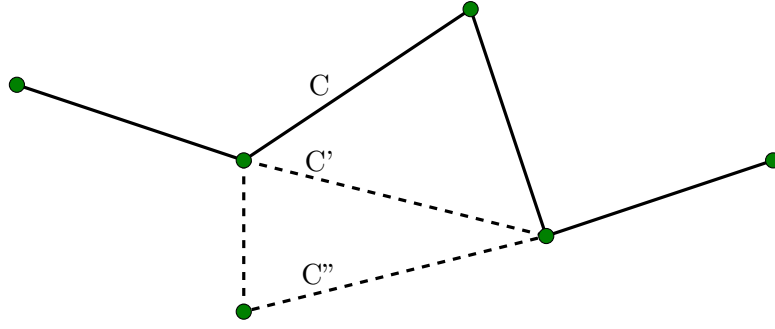


Figura 4: Tres caminos vecinos entre si.

4.2. Explicacion detallada del algoritmo propuesto

Sea s una solución factible, es decir, un camino entre u, v tal que el costo del camino $w_1 \leq K$, para plantear esta heurística definimos $N(s) = \{ \text{conjunto de soluciones vecinas de } s \} = \{ \text{caminos entre } u, v \text{ tal que } w_1 \leq K \text{ y difieren en solo un nodo de } s \}$.

Se plantearon 3 posibles enfoques para aplicar búsqueda local sobre esta vecindad, a continuación se explicarán cada uno.

4.2.1. Obtencion de la solución inicial factible

Para obtener la solución inicial factible de nuestra heurística ejecutamos el algoritmo de camino mínimo de Dijkstra, entre los nodos u y v , minimizando la función w_1 , acto seguido validamos si $\text{dist}(w_1, \text{src}, \text{dst}) > K$ entonces no existe solución factible, caso contrario tenemos una solución factible inicial para comenzar las iteraciones de búsqueda local.

4.2.2. Criterio de terminación

El criterio de terminación fue repetir las iteraciones sobre las nuevas soluciones que se iban obteniendo, hasta que no se obtiene mejora. Pueden aplicarse las iteraciones con exploraciones sobre las vecindades N_1, N_2, N_3 por separado o bien combinarse en N como mencionamos anteriormente. Esto se selecciona mediante un parámetro en el método main del código de la búsqueda local. Para todos los experimentos salvo donde se indique lo contrario, se utilizó la vecindad N .

4.2.3. Pseudocódigo

A continuación el pseudocódigo de la búsqueda local.

```

procedure Bq_Local(Grafo  $g$ , vertice  $v_1$ , vertice  $v_2$ , int  $k$ , tipo_búsqueda)  $\rightarrow$  lista  $< eje >$  camino
  lista  $< eje >$  camino = dijkstra( $g$ ,  $v_1$ ,  $v_2$ , costo $W_1$ )
  if camino.costo $W_1 > K$  then
    No existe solución.
  end if
  bool valor_mejora =  $\infty$ 
  while valor_mejora  $> 0$  do
    switch(tipo_búsqueda)
      case subdividirPares
        valor_mejora = bqLocalEntrePares( $g$ , camino)
      case contraerTriplas
        valor_mejora = bqLocalContraerTriplas( $g$ , camino)
      case mejorarTriplas :
        valor_mejora = bqlocalEntreTriplasReemplazando( $g$ , camino)
      case combinarVecindades :
        valor_mejora.1 = estimar_bqLocalEntrePares( $g$ , camino)
    
```

```

    valor_mejora_2 = estimar_bqLocalContraerTriplas(g, camino)
    valor_mejora_3 = estimar_bqlocalEntreTriplasReemplazando(g, camino)
    valor_mejora = Aplicar la mejor de las 3 vecindades exploradas o setear valor_mejora
    en 0 indicando que no se pudo mejorar mas.
  end while
  return camino
end procedure

```

A continuación se explicarán las vecindades definidas al comienzo de esta sección.

4.2.4. Reemplazar un nodo intermedio en una 3-upla consecutiva de nodos del camino

Si el camino tiene longitud 2, es decir hay una arista directa entre u, v , no hay nada que hacer en este caso, caso contrario, sea $S = \{u, \dots, v_l, v_{l+1}, v_{l+2}, v_{l+3}, \dots, v\}$ la solución actual. Lo que hace en este caso es iterar sobre todas las 3-uplas consecutivas del camino, en este ejemplo sea (v_l, v_{l+1}, v_{l+2}) una 3-upla, $(v_{l+1}, v_{l+2}, v_{l+3})$ la siguiente a iterar, etc...

Para cada 3-upla iterada se revisan los vecinos en común entre los extremos de la 3-upla buscando una mejor conexión entre extremos reemplazando el nodo intermedio, es decir, se busca una nueva conexión entre los extremos tal que, si el nuevo nodo intermedio es v_t , vale que:

- Mejore la conexión de la 3-upla respecto a w_2 , $w_2(v_l, v_t) + w_2(v_t, v_{l+2}) < w_2(v_l, v_{l+1}) + w_2(v_{l+1}, v_{l+2})$
- Este cambio, reflejado en la nueva solución candidata S' , no sobrepase la cota K establecida sobre w_1 , es decir, sea factible.

Una iteración consiste en recorrer todas las 3-uplas del camino obteniendo de los vecinos en común de los extremos de cada 3-upla, si es posible, la mejor forma de mejorar esta conexión, además recordando la mejor 3-upla para aplicar la mejora a lo largo de la iteración, tal que el camino resultante de aplicar esta mejora sea el mínimo sobre w_2 en la vecindad $N_1(s)$. Mas formalmente, se busca una solución vecina del camino S , tal que siendo $w_1(P), w_2(P)$ los costos sobre w_1 y w_2 respectivamente de una solución, y $S' = S \setminus \{(v_k, v_{k+1}); (v_{k+1}, v_{k+2})\} \cup \{(v_k, v_t); (v_t, v_{k+2})\}$ la nueva solución obtenida, entonces valga que:

- Sea factible, pertenezca a $N_1(S)$, $w_1(S') \leq K$
- Sea mínima respecto de w_2 en la vecindad $N(S)$, $w_2(S') \leq w_2(T) \forall T \in N(S)$

4.2.5. Insertar un nodo intermedio en un par consecutivo de nodos del camino

Este enfoque es básicamente igual al anterior, solo que en lugar de iterar sobre 3-uplas reemplazando el nodo intermedio, se itera sobre pares consecutivos de nodos de la solución S (v_k, v_{k+1}) buscando si existe, un vecino en común entre los extremos tal que el sendero (v_k, v_t, v_{k+1}) tenga costo menor (de forma mínima entre los vecinos) sobre w_2 que la arista directa y que reflejado en la solución candidata S' que surge de aplicar este cambio, siga siendo factible ($w_2(S') \leq K$). La iteración de sigue siendo sobre toda la vecindad $N_2(S)$ y quedándose con el mejor $S' \in N_2(S)$ posible.

4.2.6. Contracción de 3-uplas

Similar a los casos anteriores, itera sobre las 3-uplas de nodos del camino, intentando eliminar el nodo intermedio y buscar una arista que conecte directamente los nodos de los extremos. Sean los 3 nodos v_1, v_2 y v_3 , y el subcamino de aristas formado por ellos $[(v_1, v_2), (v_2, v_3)]$, se encontrar la arista (v_1, v_3) , tal que:

- $costoW2((v_1, v_3)) \leq costoW2((v_1, v_2)) + costoW2((v_2, v_3))$
- Que el camino, eliminando el nodo v_2 y las aristas $(v_1, v_2), (v_2, v_3)$ y conectando los nodos v_1 y v_3 mediante la arista directa (v_1, v_3) siga sin sobrepasar la cota K establecida sobre w_1 , es decir, sea factible.

4.3. Nivel de optimalidad de las soluciones

4.3.1. Familias de grafos malas para esta heurística

La heurística va a fallar en todos los casos en los cuales no se pueda mejorar un par o tripla de nodos agregando, quitando o reemplazando de a un nodo, por ejemplo, casos en los que la mejora entre dos nodos sea un subcamino de longitud mayor a 2, o casos en los que el camino óptimo no tiene ningún tipo de conexión con el camino mínimo según w_1 . Podría intentar arreglarse el problema generalizando la idea de contraer triplas a contraer subcaminos consecutivos de mayor longitud con alguna variación de BFS pero podría incrementarse mucho el cardinal de las vecindades, siendo esto un problema para la exploración de las mismas.

En el proximo ejemplo. Mostraremos un grafo en el cual la heurística no devuelve la solución óptima.

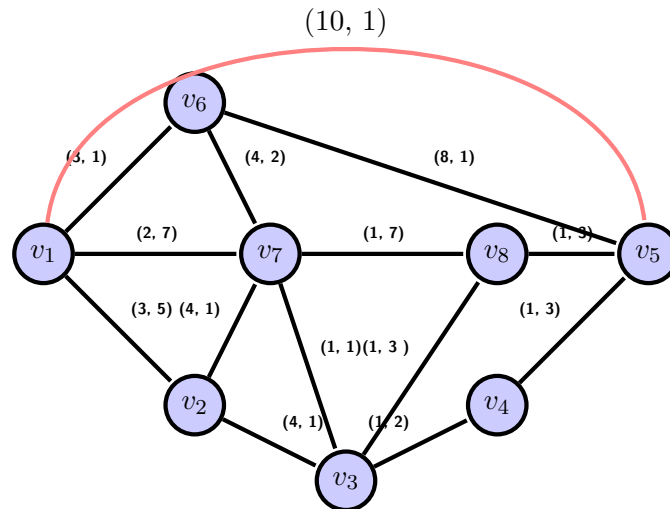


Figura 5: Ejemplo grafo malo para la heurística.

Para esta entrada, la solución inicial factible que nos brinda el algoritmo de dijkstra comienza siendo:

Nota: Los pesos entre corchetes representan a w_1 y w_2 respectivamente.

Camino inicial: $(1) \text{-----} [2, 7] \text{-----} > (7) \text{-----} [1, 7] \text{-----} > (8) \text{-----} [1, 3] \text{-----} > (5)$

La cual no tiene ninguna relación con la solución óptima. La búsqueda local finaliza devolviendo un camino:

$(1) \text{-----} [3, 1] \text{-----} > (6) \text{-----} [4, 2] \text{-----} > (7) \text{-----} [1, 1] \text{-----} > (3) \text{-----} [1, 2] \text{-----} > (4) \text{-----} [1, 3] \text{-----} > (5)$

Veamos que no es el camino óptimo: Salida del algoritmo de solución exacta

10 1 2 1 5, es decir. La arista directa entre (1) y (5) con un peso w_1 : 10 y peso w_2 : 1

Otro ejemplo, en el cual el camino que devuelve el dijkstra inicial, y el camino óptimo son disjuntos en nodos, por lo cual no hay ninguna mejora para hacer, y con lo cual vemos que la solución de la heurística puede alejarse de la óptima arbitrariamente.

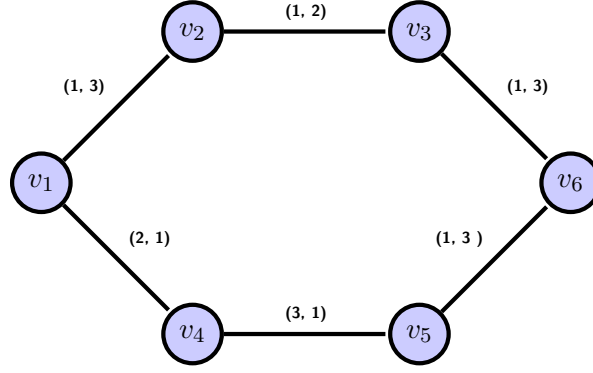


Figura 6: Ejemplo grafo malo para la heurística.

Camino inicial: (1) ---- [1, 3] ---- > (2) ---- [1, 2] ---- > (3) ---- [1, 3] ---- > (6)

Solución final de la heurística:

(1) ---- [1, 3] ---- > (2) ---- [1, 2] ---- > (3) ---- [1, 3] ---- > (6)

Veamos que no es el óptimo. Solución del algoritmo exacto:

6.000000 5.000000 4 1 4 5 6

Donde vemos que el camino óptimo es el 1,4,5,6. Esto se da por la poca densidad del grafo, y en particular, por ser disjuntos los caminos de salida a llegada, la búsqueda local no encuentra mejora para realizar y se queda con la solución que brinda el dijkstra inicial.

Es fácil notar que el hecho de agregarle aristas al grafo, posibilita a la búsqueda a comenzar a realizar mejoras, por lo tanto el algoritmo funciona mejor para grafos con una mayor densidad de aristas. Vamos a realizar pruebas sobre esto en la sección de experimentación.

4.4. Complejidad

Para la exploración de las vecindades N_1 y N_2 se recorren las duplas/triplas consecutivas de nodos, segun corresponda, lo cual toma $O(n)$, y por cada una de las triplas, se obtienen los vecinos en comun de los nodos extremos, esto cuesta $O(n)$, sin embargo, fue mejorada la constante debido a un preprocesamiento mencionado debajo. Luego la complejidad temporal de explorar sucesivamente las vecindades mencionadas es $k * O(n^2)$, donde k es la cantidad de iteraciones necesarias hasta finalizar la búsqueda local.

Para la vecindad N_3 se recorren todas las triplas consecutivas, nuevamente con un costo temporal de orden lineal. Para cada tripla, se verifica en tiempo constante por matriz de adyacencia que exista una arista directa entre los extremos y se validan las restricciones de factibilidad y mejora de w_1 y w_2 respectivamente. Luego, el costo temporal de exploracion sucesiva se la vecindad es $k * O(n)$, donde k es la cantidad de iteraciones que fueron realizadas hasta finalizar la búsqueda local.

Complejidad exploracion combinada: En este caso la exploración completa de la vecindad N consiste en una exploracion consecutiva de las vecindades N_1, N_2, N_3 , con lo cual la complejidad de explorar toda la vecindad, es la suma de las complejidades de las vecindades que la componen, es decir $O(n) + O(n^2) + O(n^2) = O(n^2)$. Luego de la exploracion, en tiempo constante se obtiene la mejora factible que mas impacte en la minimizacion de w_2 y se la aplica en tiempo constante dado que el camino esta implementado con listas enlazadas, y poseemos el iterador al punto del camino a modificar, ya sea insertando o eliminando nodos. El costo temporal total de la búsqueda local combinada es $k * O(n^2)$ donde k es la cantidad de iteraciones que fueron necesarias hasta obtener el extremo local.

Nota: Se acotan por $O(n)$ los recorridos de triplas y duplas consecutivas, porque se mantiene siempre un camino **simple** y a lo sumo contiene a todos los nodos del grafo, es decir que el camino tiene longitud n como máximo.

4.5. Taboo list

Definimos una taboo list como una lista en la cual se encuentran los nodos que pertenecen al camino solución actual, de forma de evitar que la búsqueda local intente mejorar las conexiones utilizando nodos que ya pertenecen al camino. Esto es indispensable para evitar la generación de ciclos en el caso de agregar un nodo al subdividir una arista o reemplazar un nodo intermedio entre otros dos. Si se mantienen disjuntos el conjunto de nodos del camino actual y los nodos restantes del grafo, cualquier elección que hagamos no generará ciclos.

La taboo list está implementada como un vector de bool que forma parte de los atributos de un camino del grafo, de forma de poder consultar la pertenencia de un nodo al camino en tiempo constante.

Otra opción considerada fue no restringir la búsqueda de nuevos nodos, pero luego debería realizarse una poda de ciclos del camino. En muchos de los casos esto realizaría una mejora importante del camino, pero esto aumenta la complejidad de la heurística y deja de ser búsqueda local, dado que la solución que surja de la poda en muchos casos no va a estar en la vecindad del camino.

4.6. Precómputo de vecinos en comun

Para agilizar la búsqueda de vecinos en comun entre 2 nodos para la exploración de las vecindades y aprovechando que el grafo es estático, es decir, no se agregan o quitan aristas luego de ser leído de la entrada, se precomputa, para cada par de nodos v_i, v_j una lista con los nodos adyacentes en comun. Luego para obtener los vecinos en comun en los diversos algoritmos implementados, solo basta con obtener esta información precomputada.

4.7. Experimentación: Mediciones de Performance

En esta sección se mostrarán resultados de complejidad temporal empírica acerca del costo promedio de una iteración de búsqueda local, veremos que tal como analizamos en la sección anterior de complejidad, dicho costo está en el orden cuadrático $O(n^2)$. Para tener una mejor idea del comportamiento del algoritmo, realizamos pruebas sobre grafos aleatorios de distintos tamaños (en cantidad de nodos), y por cada cantidad n de nodos, variamos las densidades de aristas dentro de cierto rango alrededor de una función de n , las densidades elegidas fueron:

- $m = a * n + b$. Es decir una cantidad lineal de aristas en base a los nodos. $a \in \mathbb{N}_{>1}$
- $m = \frac{n*(n-1)}{2}$. Es decir grafos cercanos o iguales a cliques de n nodos.

Nota: Como mencionamos anteriormente, las funciones son variadas en un rango, es decir, por ejemplo, para el caso de cliques, los grafos generados tienen entre $\frac{n*(n-1)}{5}$ y $\frac{n*(n-1)}{2}$ aristas para aleatorizar más la generación de grafos densos.

Nota: Los gráficos que contienen puntos rojos y una curva verde, indican, para cada valor del eje X (cantidad de nodos), los puntos rojos son los tiempos de ejecución para los diferentes valores de aristas en el rango de la familia, asimismo, la curva verde indica el promedio de estos puntos para cada X.

Nota: Para verificar que se trata de una curva cuadrática dividimos las funciones por n^1, n^2, n^3, n^4 y como en los trabajos prácticos anteriores concluimos de qué curva se trata.

A continuación presentamos los resultados de estos experimentos:

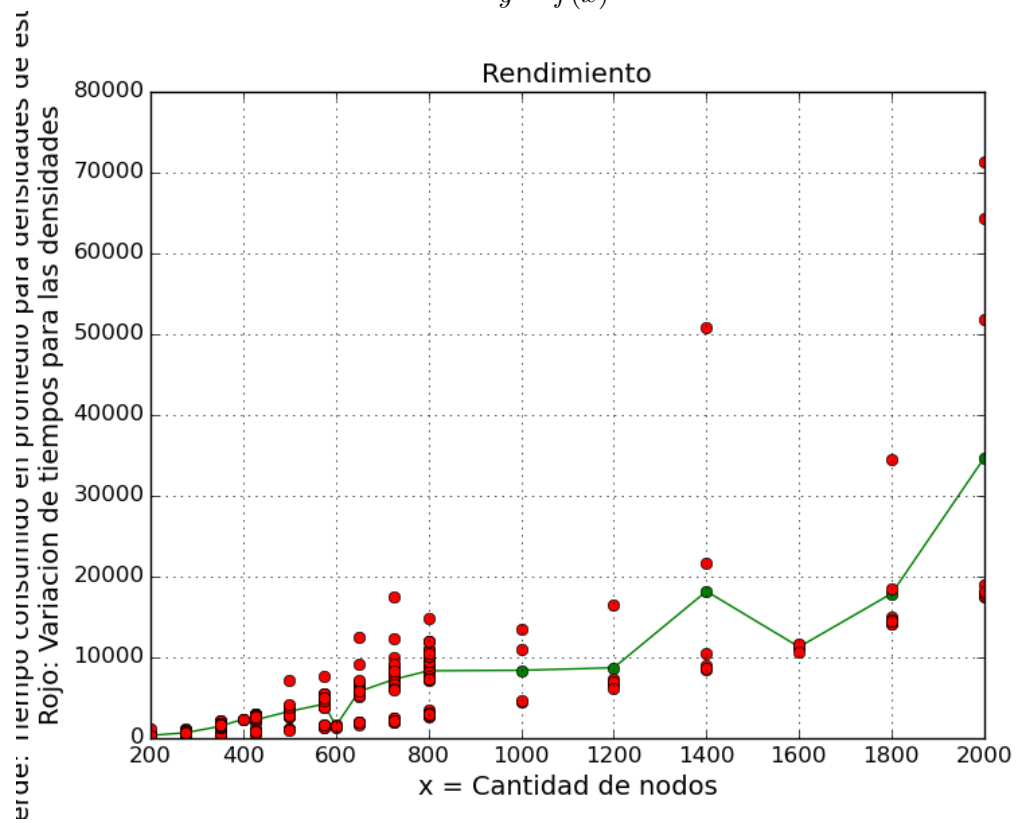
4.7.1. Rendimiento para grafos con densidad lineal de aristas

- cant nodos min = 200
- cant nodos max = 2000
- peso maximo w1 = 200

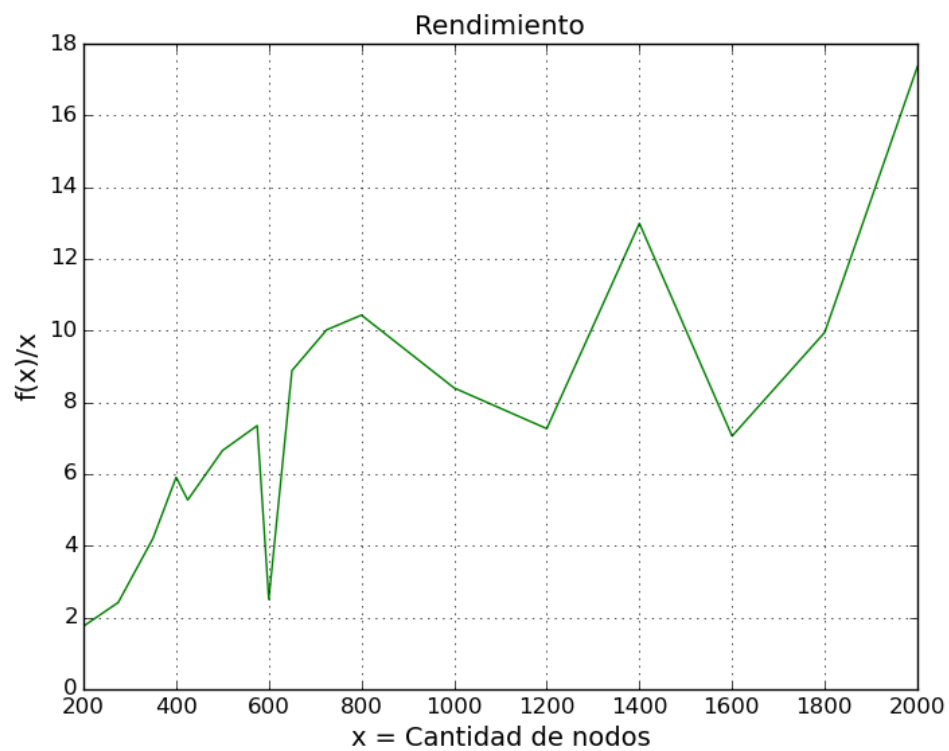
- peso maximo $w_2 = 200$
- step nodos = 200
- step aristas = 2500
- aristas minimas = $n - 1$
- aristas maximas = $10 * n$

Tiempo de ejecución en microsegundos para esta familia

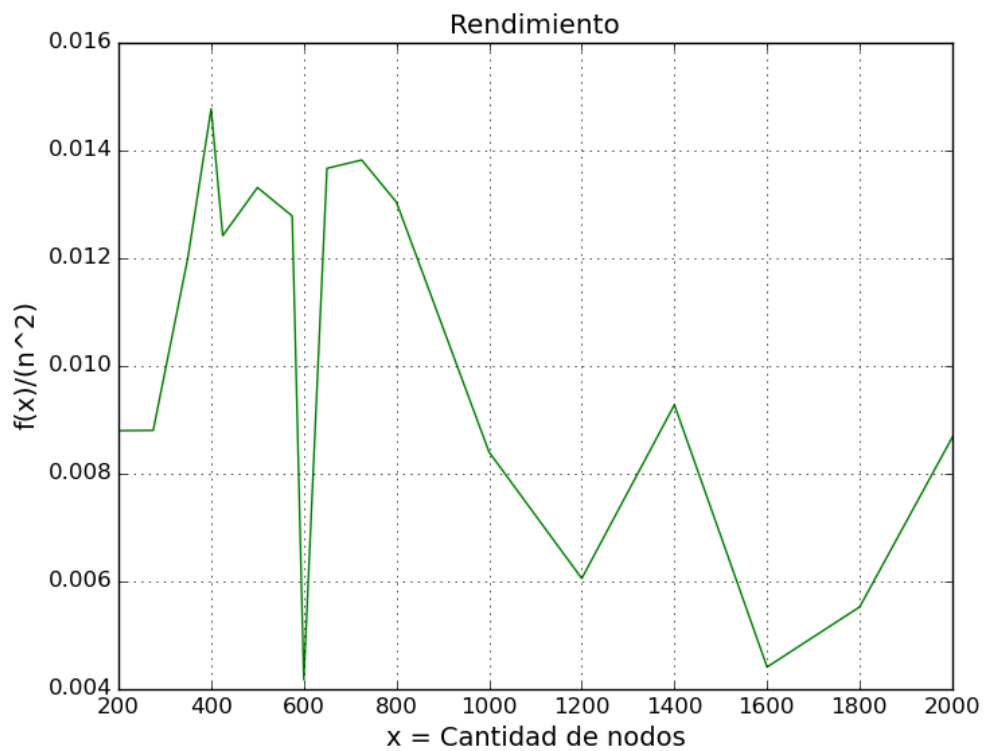
$$y = f(x)$$



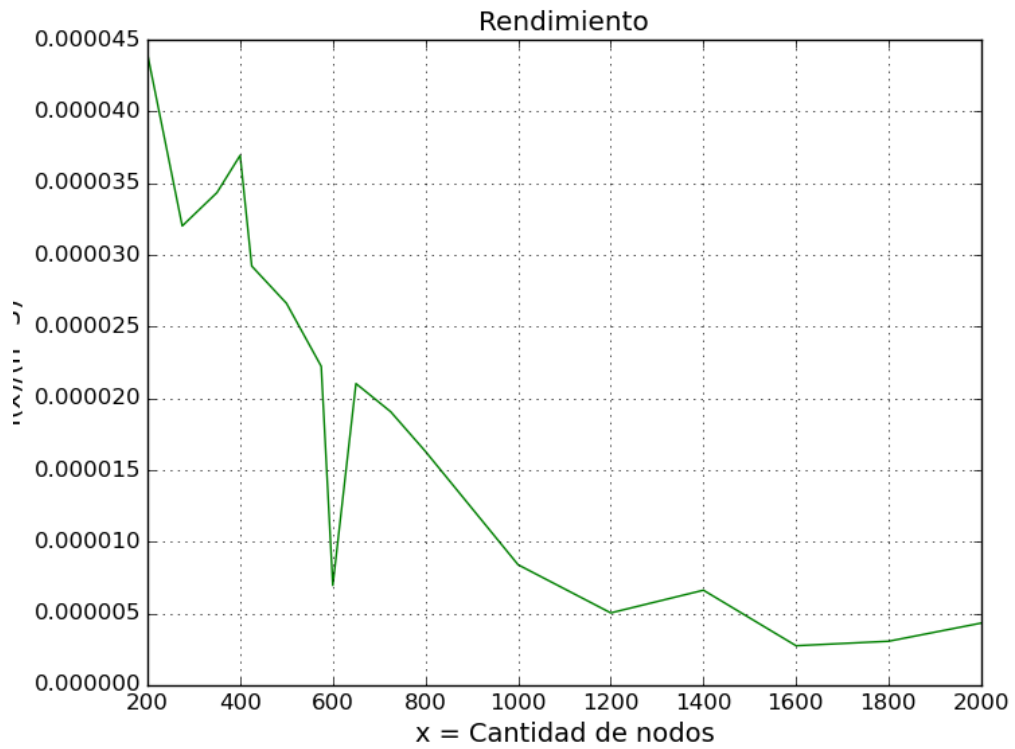
$$y = f(x)/x$$



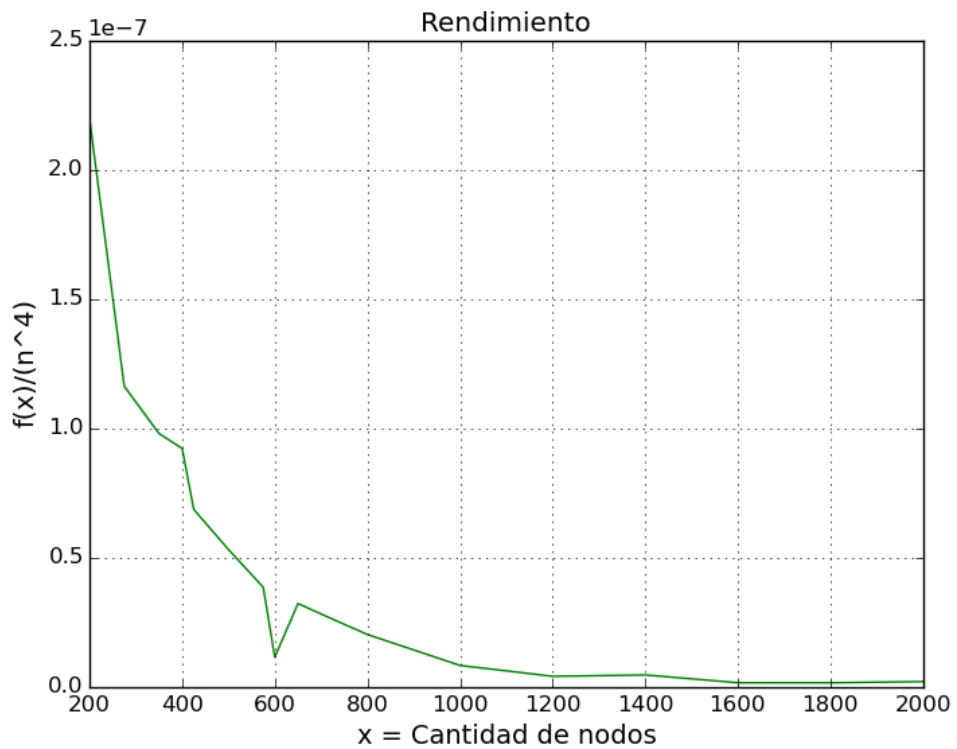
$$y = f(x)/x^2$$



$$y = f(x)/x^3$$



$$y = f(x)/x^4$$



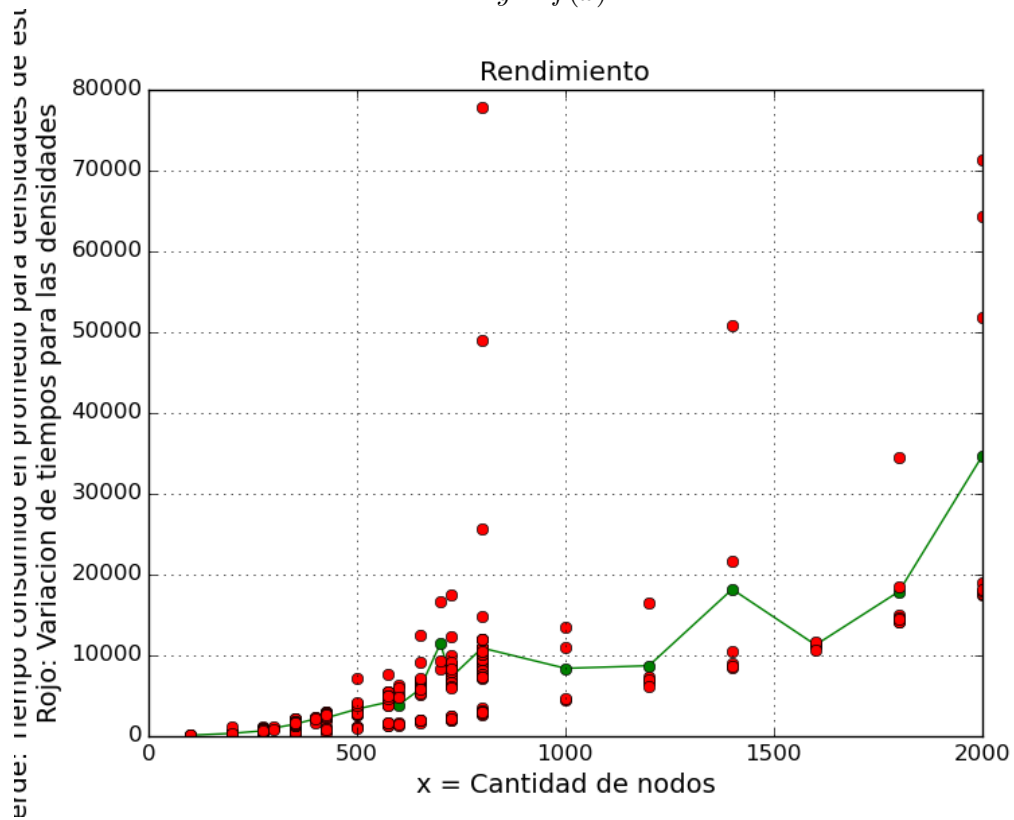
4.7.2. Rendimiento para grafos con densidad cuadratica de aristas

- cant nodos min = 100
- cant nodos max = 2000
- peso maximo w1 = 200

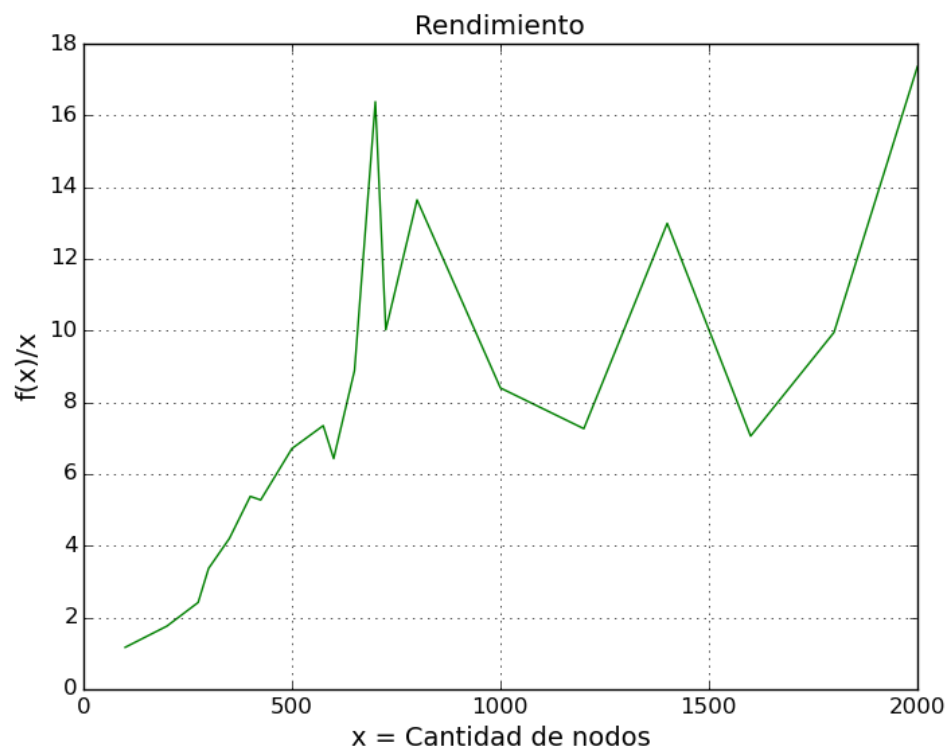
- peso maximo $w_2 = 200$
- step nodos = 200
- step aristas = 2500
- aristas minimas = $\frac{n*(n-1)}{17}$
- aristas maximas = $\frac{n*(n-1)}{14}$

Tiempo de ejecución en microsegundos para esta familia

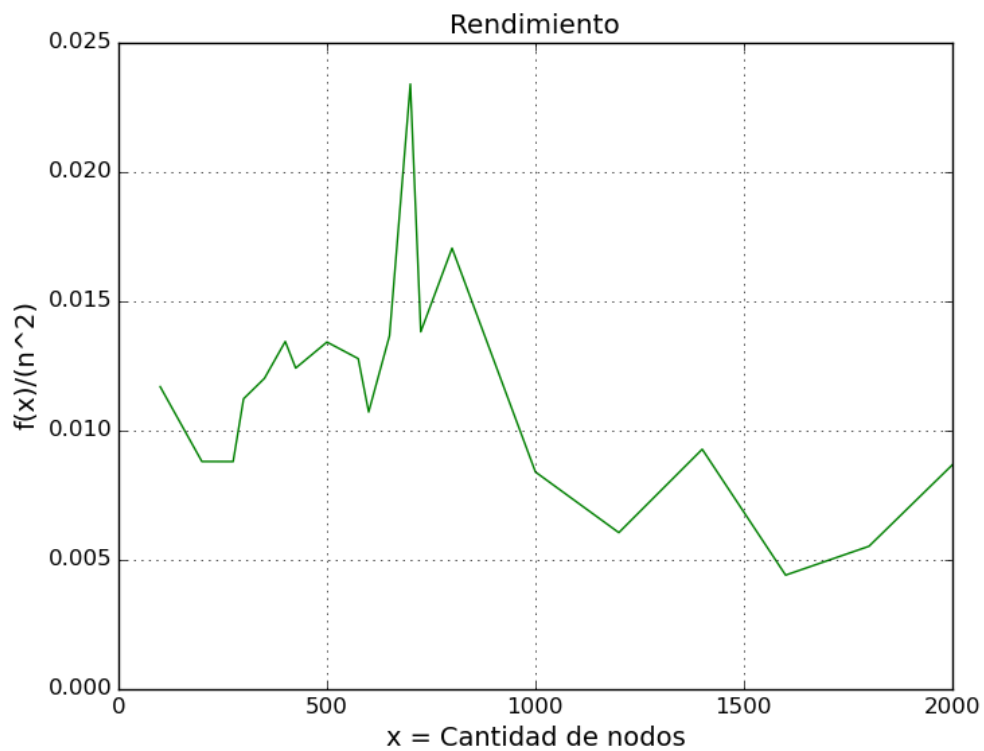
$$y = f(x)$$



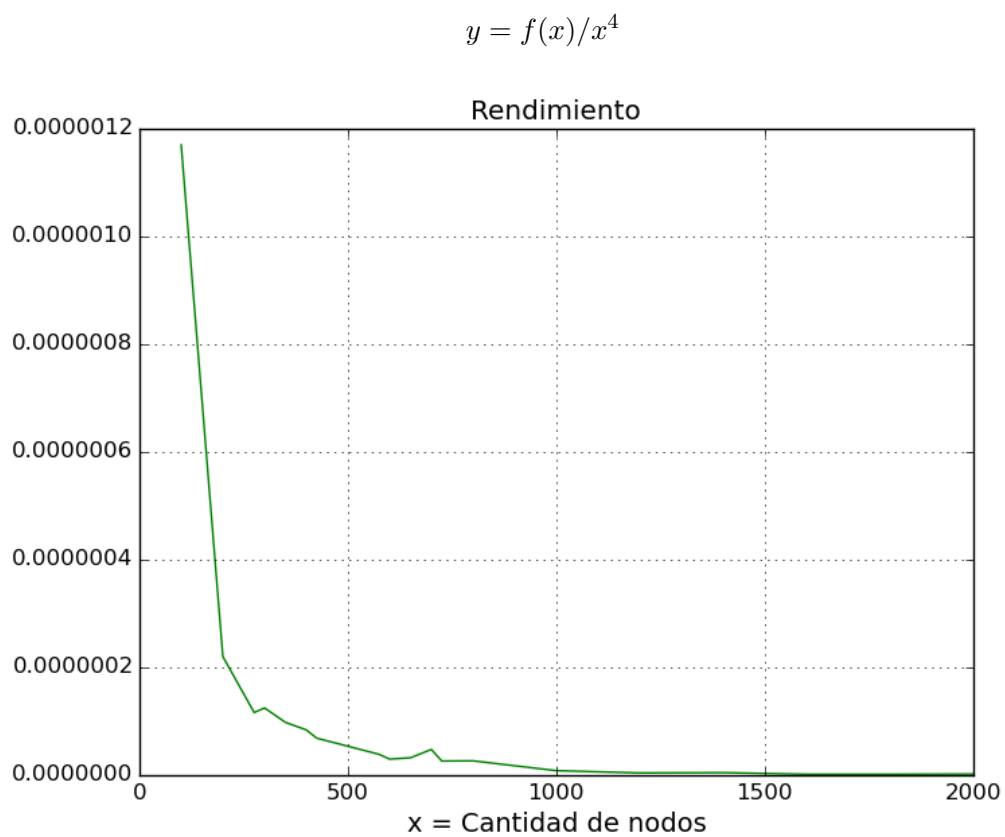
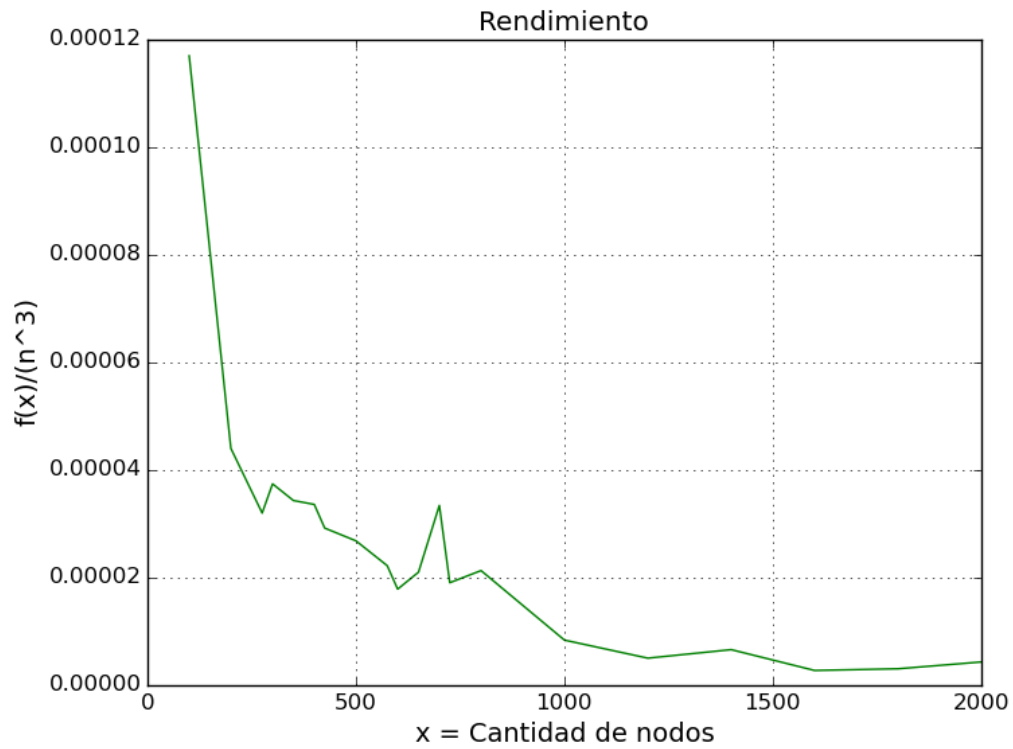
$$y = f(x)/x$$



$$y = f(x)/x^2$$



$$y = f(x)/x^3$$



Como vemos en las 2 secciones de resultados anteriores, para n^2 se mantiene constante dentro de un rango muy reducido en el eje Y, pero para n^3 y n^4 es una función marcadamente decreciente, con lo cual, llegamos a la conclusión de que la curva es o bien, cuadrática o cúbica. Pero analizando con más detalle notamos que la constante para n^2 es positiva y un número razonable (mayor a 1), mientras que la constante de n^3 tiende a cero y se aplasta contra el eje X, lo cual es el resultado de dividir una constante por la variable de experimentación. Con lo cual, al ser $f(x)/x$ creciente, $f(x)/x^2$ una

constante cercana a 10, y $f(x)/x^3$ una curva que tiende a cero, es un buen indicio de que la complejidad teórica de $O(n^2)$ es acertada.

4.8. Experimentacion: Variación evolutiva de mejora en cada iteracion y Variación absoluta de w_2 en busqueda local

En esta seccion se realizaron dos tipos de analisis, el primero corresponde a las variaciones entre cada iteracion del costo a minimizar, el segundo corresponde a la variación absoluta del costo a minimizar.

Las mediciones consisten en que, en cada iteracion de busqueda local se almacenen los costos w_1 y w_2 del camino actual obtenido y los saltos de costo w_2 entre iteraciones.

Luego se realizan dos graficos, el primero, referido a la variacion entre cada iteracion, indica en el eje X el numero de la iteracion y en el eje Y la mejora realizada en dicha iteracion a la solucion previa. El segundo, referido a la variacion absoluta del costo a minimizar, indica en el eje X la cantidad de iteraciones y en el eje Y el valor absoluto del costo de la solucion. Si realizamos rudimentario analisis estadistico, un promedio y una desviacion estandar sobre este ultimo dato, nos podrán dar idea para diferentes sets de grafos como es el rendimiento de la busqueda local.

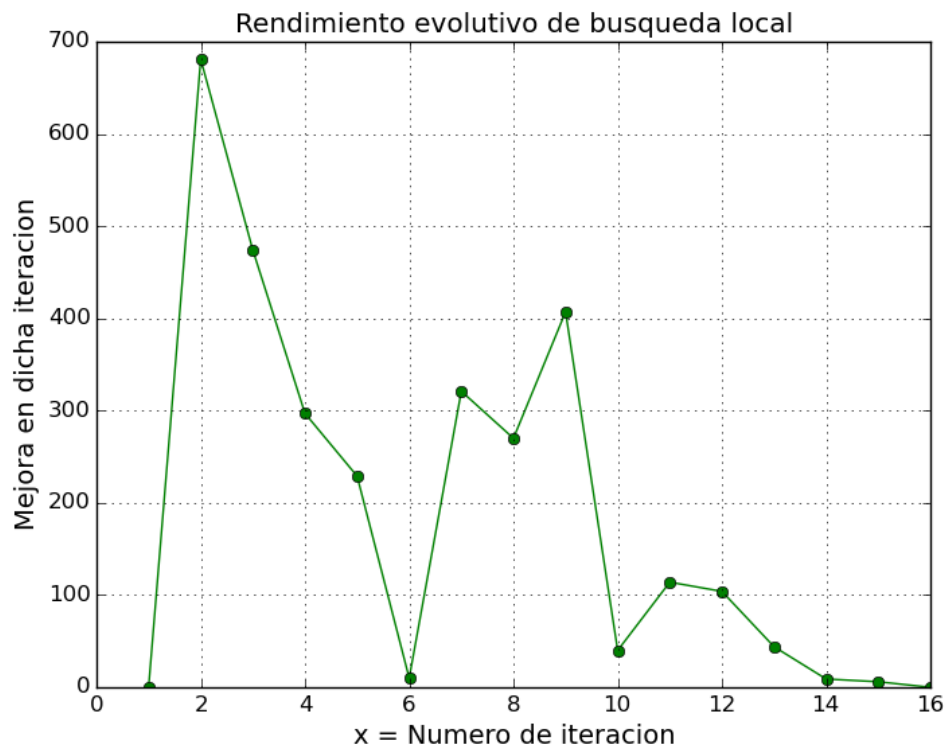
4.8.1. Rendimiento evolutivo y absoluto en grafos con alta densidad de aristas

Parametros del experimento:

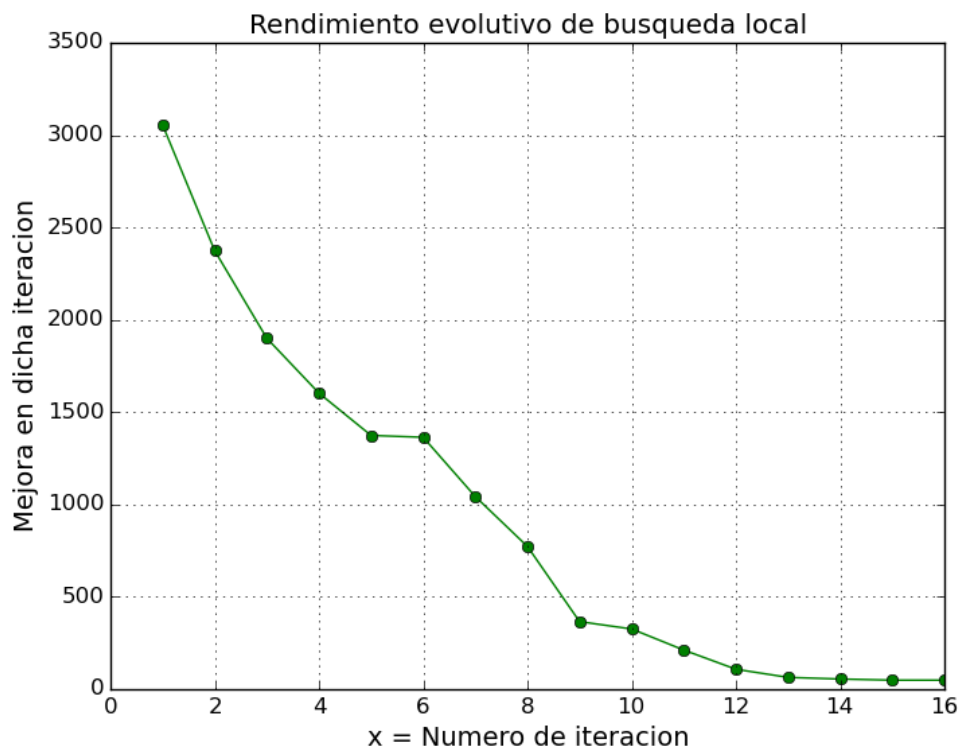
- Cantidad minima de nodos: 50
- Cantidad maxima de nodos: 220
- Rango peso w_1 : [0..250]
- Rango peso w_2 : [0..400]
- Cota de w_1 : 400
- Cantidad minima de aristas: $\frac{n*(n-1)}{3}$
- Cantidad maxima de aristas: $\frac{n*(n-1)}{2}$

Ejemplo de la evolución en la mejora de la solucion durante las iteraciones de la busqueda local

$y = f(x)$, para cada x =numero de iteracion, $f(x)$ expresa la mejora obtenida en w_2 en dicha iteracion



Ejemplo de funcion costo target a minimizar durante las iteraciones de la búsqueda local
 $y = f(x)$, para cada x =numero de iteracion, $f(x)$ expresa el costo w_2 en dicha iteracion



Resultados del analisis:

Cantidad de tests realizados: 34
 Iteraciones promedio: 5

```

Iteraciones stddev: 2.99726
Minima cantidad de iteraciones: 2
Maxima cantidad de iteraciones: 15
Mejora promedio del costo w2: 934
Mejora stddev del costo w2: 523.031
Minima mejora en w2 registrada: 201 en 2 iteraciones
Maxima mejora en w2 registrada: 3006 en 15 iteraciones

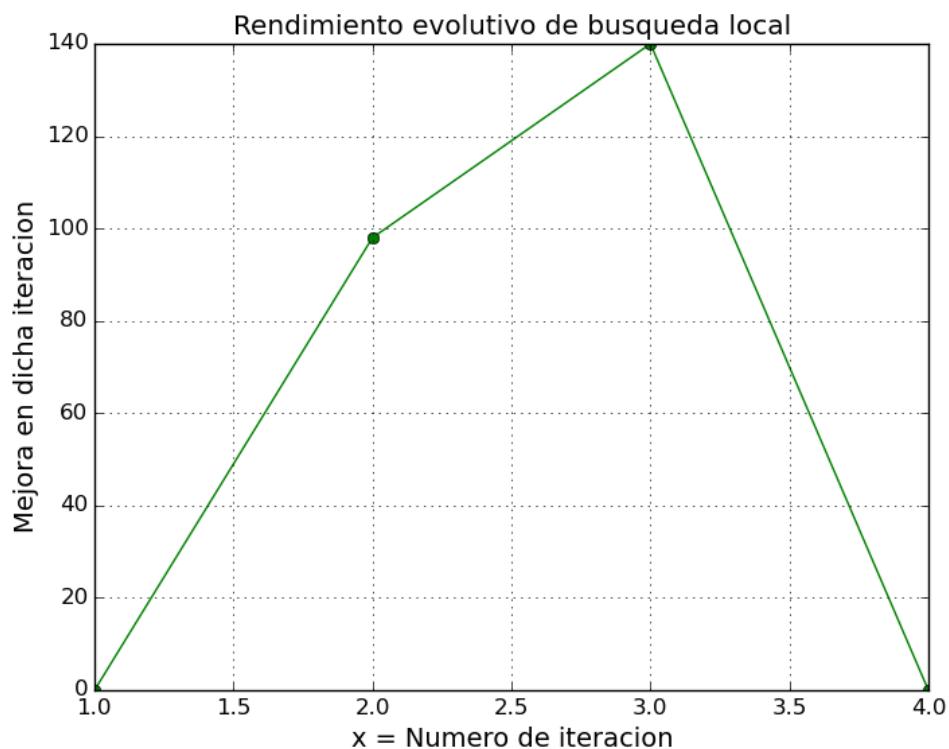
```

4.8.2. Rendimiento evolutivo en grafos con densidad lineal de aristas

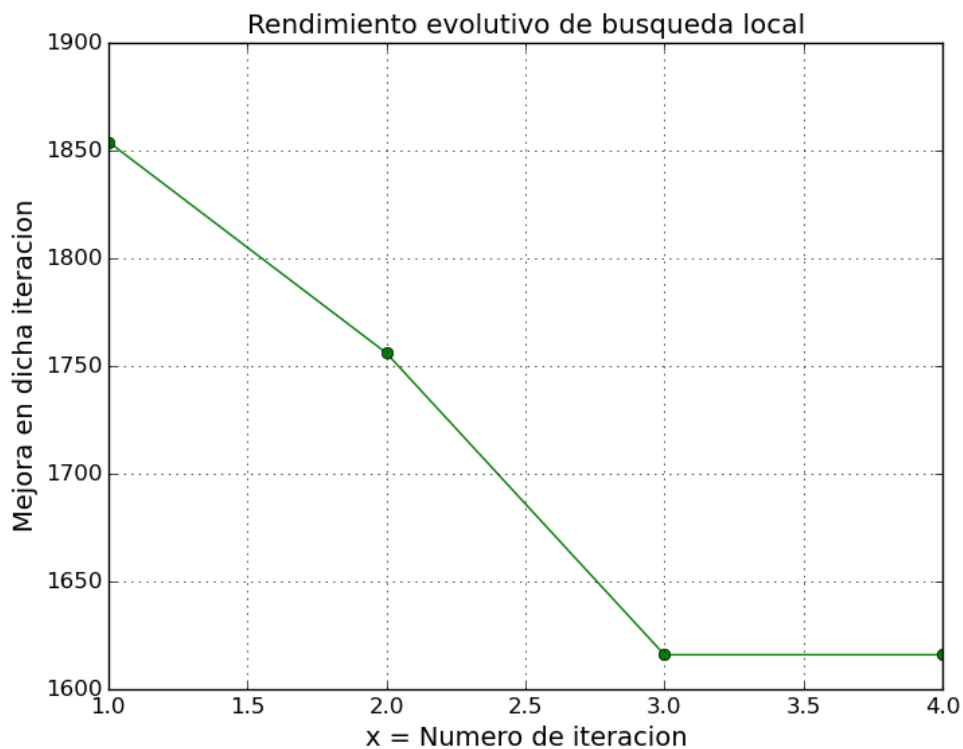
Parametros del experimento:

- Cantidad minima de nodos: 100
- Cantidad maxima de nodos: 2000
- Rango peso w_1 : [0..250]
- Rango peso w_2 : [0..400]
- Cota de w_1 : 400
- Cantidad minima de aristas: $n - 1$
- Cantidad maxima de aristas: $10 * n$

Ejemplo de la evolución en la mejora de la solución durante las iteraciones de la búsqueda local
 $y = f(x)$, para cada x =numero de iteracion, $f(x)$ expresa la mejora obtenida en w_2 en dicha iteracion



Ejemplo de funcion costo target a minimizar durante las iteraciones de la busqueda local $y = f(x)$, para cada x =numero de iteracion, $f(x)$ expresa el costo w_2 en dicha iteracion



Resultados del analisis:

```
Cantidad de tests realizados: 54
Iteraciones promedio: 0
Iteraciones stddev: 0.526955
Minima cantidad de iteraciones: 1
Maxima cantidad de iteraciones: 3
Mejora promedio del costo w2: 45
Mejora stddev del costo w2: 135.415
Minima mejora en w2 registrada: 0 en 1 iteraciones
Maxima mejora en w2 registrada: 481 en 2 iteraciones
```

4.8.3. Conclusion de experimentos - Relacion entre cantidad de aristas y efectividad de busqueda local

Podemos observar que para grafos con menor densidad de aristas disminuye drasticamente la mejora tanto en cantidad de iteraciones como en valor absoluto promedio de mejora, con medidas de dispersion similares en cada caso(respecto a los maximos minimos y promedios correspondientes).

Veamos que por ejemplo, para grafos con poca densidad de aristas, la mejora es muy baja, y la cantidad de iteraciones promedio es 0, lo cual es un numero contundente. Esto indica que hay una relacion entre la densidad de aristas del grafo y la efectividad de la heuristica.

Creemos que se debe a una relacion directamente proporcional entre la densidad del grafo y el cardinal de la vecindad N , lo cual tiene sentido, porque a mayor cantidad de aristas en el grafo, mas modificaciones locales pueden hacerse al camino inicial a nivel local.

Teniendo en cuenta la cantidad promedio de iteraciones y mejora promedio para grafos de baja densidad y los mismos resultados para grafos de alta densidad, notemos que aunque la cantidad de nodos aumente mucho mas en los grafos poco densos, los indicadores estadisticos siguen siendo bajisimos. Vamos a

basarnos en estos resultados para deducir que una familia mala de grafos para la búsqueda local serán aquellos con poca densidad de aristas, lo cual produce un cardinal bajo en la vecindad, dando lugar a muy pocas o nulas mejoras locales de la solución inicial.

4.9. Experimentación: Distintas soluciones iniciales factibles

4.9.1. Grafos particulares

Como mencionamos antes, se utiliza Dijkstra sobre w_1 para obtener la solución inicial factible, para comenzar las iteraciones de búsqueda local. En esta sección decidimos variar esto y establecer la solución inicial de búsqueda local con la heurística golosa. Para ciertos grafos preestablecidos que pueden encontrarse en la carpeta **grafos especiales** las soluciones finales de búsqueda local con los 2 modos de solución inicial son iguales, lo que varió de forma notable fue la cantidad de iteraciones de búsqueda local necesarias para llegar a dicha solución final, debajo se muestra una tabla indicando esto:

Grafo especial numero	Cant. iters Dijkstra	Cant. iters Greedy
Grafo_1.txt	3	0
Grafo_2.txt	2	0
Grafo_3.txt	1	0
Grafo_4.txt	0	0

Asumimos que esto se debe a la calidad de la solución provista por el algoritmo goloso.

Nota: Estos grafos se encuentran en la carpeta `codigo/heuristicas/grafos_especiales`.

4.9.2. Grafos aleatorios de alta densidad de aristas

Parametros del experimento:

- Cantidad minima de nodos: 50
- Cantidad maxima de nodos: 220
- Rango peso w_1 : [0..250]
- Rango peso w_2 : [0..400]
- Cota de w_1 : 400
- Cantidad minima de aristas: $\frac{n*(n-1)}{3}$
- Cantidad maxima de aristas: $\frac{n*(n-1)}{2}$

Resultados del analisis:

```
Cantidad de tests realizados: 28
Iteraciones promedio: 1
Iteraciones stddev: 0.185577
Minima cantidad de iteraciones: 1
Maxima cantidad de iteraciones: 2
Mejora promedio del costo w2: 0
Mejora stddev del costo w2: 4.08269
Minima mejora en w2 registrada: 0 en 1 iteraciones
Maxima mejora en w2 registrada: 22 en 2 iteraciones
```

Vemos que al contrastar estos resultados con la sección anterior, que evalúa el rendimiento con dijkstra sobre w_1 como solución inicial, para la misma familia de grafos aleatorios el uso de la heurística golosa como solución inicial perjudica muchísimo el rendimiento de la búsqueda local, reduciendo la cantidad de iteraciones promedio de 5 a 1, con una gran reducción en la desviación estándar y una reducción en la amplitud entre cantidad máxima y mínima de iteraciones. Asimismo, los indicadores

acerca de la mejora en w_2 tambien se ven **muy reducidos**. Llegamos a la conclusión que el gran rendimiento en optimalidad de la heurística golosa opaca las posibilidades de búsqueda local de mejorar la solución a nivel local. Esta experimentación sirve como prelude para la sección de la metaheurística GRASP, dado que la heurística golosa determinística es un caso particular para ciertos valores muy chicos del parametro beta de la metaheurística, para los cuales la componente de la búsqueda local no será efectiva y las soluciones de GRASP serán cercanas a las de la heurística golosa.

4.9.3. Grafos aleatorios de baja densidad de aristas

Parametros del experimento:

- Cantidad minima de nodos: 100
- Cantidad maxima de nodos: 2000
- Rango peso w_1 : [0..250]
- Rango peso w_2 : [0..400]
- Cota de w_1 : 400
- Cantidad minima de aristas: $n - 1$
- Cantidad maxima de aristas: $10 * n$

Resultados del analisis:

```

Cantidad de tests realizados: 125
Iteraciones promedio: 0
Iteraciones stddev: 0
Minima cantidad de iteraciones: 1
Maxima cantidad de iteraciones: 1
Mejora promedio del costo w2: 0
Mejora stddev del costo w2: 0
Minima mejora en w2 registrada: 0 en 1 iteraciones
Maxima mejora en w2 registrada: 0 en 1 iteraciones

```

En el caso de baja densidad de aristas es peor, directamente no se obtuvo ninguna mejora en ningun grafo del conjunto del experimento.

4.10. Variación de efectividad de busqueda local sobre conjunto fijo de grafos cambiando vecindades

Para evaluar la elección de la mejor vecindad de busqueda local, se realizaron experimentos sobre 2 densidades de aristas de grafos. Se generaron dos conjuntos de grafos, uno con cada densidad de aristas. Luego se corrieron los algoritmos de busqueda local con diferentes vecindades para estos dos conjuntos fijos. A continuación se muestran los resultados obtenidos:

4.10.1. Experimentación sobre conjunto de grafos con baja densidad de aristas

Parametros del experimento:

- Cantidad minima de nodos: 100
- Cantidad maxima de nodos: 2000
- Rango peso w_1 : [0..250]
- Rango peso w_2 : [0..400]
- Cota de w_1 : 400

- Cantidad minima de aristas: $n - 1$
- Cantidad maxima de aristas: $10 * n$

Vecindad N_1 :

Resultados del analisis:

Cantidad de tests realizados: 54
Iteraciones promedio: 0
Iteraciones stddev: 0.580091
Minima cantidad de iteraciones: 1
Maxima cantidad de iteraciones: 3
Mejora promedio del costo w2: 42
Mejora stddev del costo w2: 121.582
Minima mejora en w2 registrada: 0 en 1 iteraciones
Maxima mejora en w2 registrada: 411 en 3 iteraciones

Vecindad N_2 :

Resultados del analisis:

Cantidad de tests realizados: 54
Iteraciones promedio: 0
Iteraciones stddev: 0.373193
Minima cantidad de iteraciones: 1
Maxima cantidad de iteraciones: 3
Mejora promedio del costo w2: 12
Mejora stddev del costo w2: 83.0652
Minima mejora en w2 registrada: 0 en 1 iteraciones
Maxima mejora en w2 registrada: 460 en 3 iteraciones

Vecindad N_3 :

Resultados del analisis:

Cantidad de tests realizados: 54
Iteraciones promedio: 0
Iteraciones stddev: 0.373193
Minima cantidad de iteraciones: 1
Maxima cantidad de iteraciones: 3
Mejora promedio del costo w2: 23
Mejora stddev del costo w2: 155.133
Minima mejora en w2 registrada: 0 en 1 iteraciones
Maxima mejora en w2 registrada: 772 en 3 iteraciones

Vecindad $N(C) : N_1 \cup N_2 \cup N_3$:

Resultados del analisis:

Cantidad de tests realizados: 54
Iteraciones promedio: 0
Iteraciones stddev: 0.915552
Minima cantidad de iteraciones: 1
Maxima cantidad de iteraciones: 5
Mejora promedio del costo w2: 78
Mejora stddev del costo w2: 243.037
Minima mejora en w2 registrada: 0 en 1 iteraciones
Maxima mejora en w2 registrada: 1037 en 5 iteraciones

4.10.2. Experimentación sobre conjunto de grafos con alta densidad de aristas

Parametros del experimento:

- Cantidad minima de nodos: 50
- Cantidad maxima de nodos: 220
- Rango peso w_1 : [0..250]
- Rango peso w_2 : [0..400]
- Cota de w_1 : 400
- Cantidad minima de aristas: $\frac{n*(n-1)}{3}$
- Cantidad maxima de aristas: $\frac{n*(n-1)}{2}$

Vecindad N_1 :

Resultados del analisis:

```
Cantidad de tests realizados: 34
Iteraciones promedio: 3
Iteraciones stddev: 2.25399
Minima cantidad de iteraciones: 1
Maxima cantidad de iteraciones: 10
Mejora promedio del costo w2: 536
Mejora stddev del costo w2: 350.713
Minima mejora en w2 registrada: 0 en 1 iteraciones
Maxima mejora en w2 registrada: 1385 en 8 iteraciones
```

Vecindad N_2 :

Resultados del analisis:

```
Cantidad de tests realizados: 34
Iteraciones promedio: 2
Iteraciones stddev: 1.18854
Minima cantidad de iteraciones: 1
Maxima cantidad de iteraciones: 4
Mejora promedio del costo w2: 150
Mejora stddev del costo w2: 162.863
Minima mejora en w2 registrada: 0 en 1 iteraciones
Maxima mejora en w2 registrada: 658 en 4 iteraciones
```

Vecindad N_3 :

Resultados del analisis:

```
Cantidad de tests realizados: 34
Iteraciones promedio: 3
Iteraciones stddev: 2.04025
Minima cantidad de iteraciones: 1
Maxima cantidad de iteraciones: 10
Mejora promedio del costo w2: 751
Mejora stddev del costo w2: 545.173
Minima mejora en w2 registrada: 0 en 1 iteraciones
Maxima mejora en w2 registrada: 2595 en 10 iteraciones
```

Vecindad $N(C) : N_1 \cup N_2 \cup N_3$:

Resultados del analisis:

Cantidad de tests realizados: 34
Iteraciones promedio: 4
Iteraciones stddev: 1.8492
Minima cantidad de iteraciones: 1
Maxima cantidad de iteraciones: 8
Mejora promedio del costo w2: 877
Mejora stddev del costo w2: 428.166
Minima mejora en w2 registrada: 0 en 1 iteraciones
Maxima mejora en w2 registrada: 1631 en 8 iteraciones

Conclusion:

Podemos observar de los resultados que en general, para grafos de baja densidad es mejor la heurística con vecindad combinada N , pero para grafos densos, no lo es. Lo cual nos lleva a la conclusion de que expandir la vecindad no necesariamente proporciona una mejor solución final, pues la decisión golosa de obtener siempre el camino vecino que mas optimice localmente la función target puede pasar por alto el caso donde haber obtenido una solución peor en este paso, nos dará una mas alta mejora en una iteración siguiente, produciendo una mejor solución final.

4.11. Experimentacion: Mediciones de Optimalidad de las soluciones obtenidas con esta heurística

En esta sección nos vamos a dedicar al analisis de la calidad de las soluciones, es decir, dados conjuntos de grafos aleatorios, vamos a examinar el porcentaje de soluciones optimas obtenidas por esta heurística y realizaremos algunos calculos estadísticos acerca de la lejanía promedio de las soluciones obtenidas con respecto a la solución exacta en los grafos del conjunto de instancias de las muestras. Para esto, se presentaran 2 experimentos, los cuales representan 2 densidades de grafos generados aleatoriamente.

Nota: La lejanía entre 2 soluciones se mide haciendo el siguiente calculo: $100 * (\frac{solucionHeuristica}{solucionOptima} - 1)$

Nota: Los calculos estadisticos (promedio y desviacion estandar) se realizan sobre la lista de resultados obtenida de la ejecución secuencial y el calculo de la lejanía mencionado aqui arriba para cada uno de los algoritmos(exacto y heurística) sobre cada instancia del conjunto de pruebas.

4.11.1. Grafos aleatorios de baja densidad de aristas

Parametros del experimento:

- Cantidad de grafos analizados: 25
- Cantidad minima de nodos: 100
- Cantidad maxima de nodos: 2200
- Rango peso w_1 : [0..250]
- Rango peso w_2 : [0..400]
- Cota de w_1 : 200
- Cantidad minima de aristas: $n - 1$
- Cantidad maxima de aristas: $10 * n$

Resultados del analisis:

Heuristica da la solucion optima: 20.0% de los casos Lejania promedio de la heuristica a la solucion optima: 144.248 % Desv. estandar de la lejania entre las soluciones: 295.092 Minima lejania entre bqlocal y exacta: 0 Maxima lejania entre bqlocal y exacta: 1502.60%
--

Vemos que la desviacion estandar, nos indica que la maxima lejania entre la heuristica y la solucion optima no es algo que ocurra comunmente, sin embargo, este máximo sigue siendo un numero muy grande como cota extrema de las mediciones.

4.11.2. Grafos aleatorios de alta densidad de aristas

Parametros del experimento:

- Cantidad de grafos analizados: 48
- Cantidad minima de nodos: 10
- Cantidad maxima de nodos: 300
- Rango peso w_1 : [0..250]
- Rango peso w_2 : [0..400]
- Cota de w_1 : 200
- Cantidad minima de aristas: $\frac{n*(n-1)}{3}$
- Cantidad maxima de aristas: $\frac{n*(n-1)}{2}$

Resultados del analisis:

Heuristica da la solucion optima: 12.5% de los casos Lejania promedio de la heuristica a la solucion optima: 944.600 % Desv. estandar de la lejania entre las soluciones: 1578.35 Minima lejania entre bqlocal y exacta: 0 Maxima lejania entre bqlocal y exacta: 9411.100%

Como podemos observar, búsqueda local no provee una buena calidad de soluciones respecto al algoritmo exacto(al menos no con dijkstra sobre w_1 como sol. inicial, analizaremos greedy como sol. inicial entre otras cosas al analizar la metaheuristica GRASP). Podemos observar que con una baja densidad de aristas, el panorama es aun peor, ya que no solo disminuye la cantidad % de los casos donde se obtiene la solución óptima ya que tambien aumenta de forma considerable la lejania promedio y la desviacion estandar, dandonos un maximo enorme de distancia entre la solucion optima y la provista por la heuristica.

Consideramos que la búsqueda local sería una buena forma de mejorar soluciones factibles, pero afecta demasiado la eleccion de la solucion inicial, ya que esto nos modifica considerablemente(como explicamos en la seccion anterior cuando comparamos dijkstra sobre w_1 y greedy como solucion inicial) la solucion final obtenida. Observando el alto porcentaje de casos donde la heuristica golosa proporciona la solucion optima y su lejania al optimo cuando no la obtiene podemos observar que alimentar la busqueda local con la heuristica golosa mejoraria de forma considerable la solucion final y por ende, cambiarian estos resultados presentados previamente.

5. Metaheurística GRASP: Solucion propuesta

Dado que la metaheurística GRASP es una combinación entre una heurística golosa aleatorizada y una búsqueda local, decidimos utilizar nuestras heurísticas previamente mencionadas, con algunas modificaciones.

5.1. Modificación a la heurística golosa

Durante el ciclo del algoritmo goloso, recordemos que la primera instrucción del ciclo principal, al igual que el algoritmo de Dijkstra, es obtener el mínimo nodo de la cola según w_2 . En lugar de esto, ahora armamos una lista restringida de candidatos, o RCL. El algoritmo ahora tiene dos parámetros, *tipo_ejecucion* que indica tres tipos de ejecución: determinístico, por valor y por cantidad, y un parámetro β . El tipo determinístico es análogo a la heurística golosa antes descrita, simplemente tomando el mínimo y el parámetro β es ignorado.

El tipo de ejecución por valor arma una lista de candidatos filtrados por su valor, según un porcentaje de alejamiento del valor del mínimo, indicado por el parámetro β . Es decir, se toma la cola y se filtran los candidatos factibles cuyo valor sobrepase $(\beta + 1) * valor_del_minimo$. Luego se elige al azar uno de los candidatos.

El tipo de ejecución por cantidad se basa en tomar la cola, y tomar los β nodos de valor mínimo. Luego se elige uno al azar. Como la cola está ordenada, cumple la condición de RCL por cantidad, con lo que basta tomar un número aleatorio i entre 0 y $\min\{cola.size(), parametro_beta\} - 1$ y devolver el i -ésimo elemento de la cola.

Nota: Los numeros aleatorios generados para elegir de la RCL en un principio se realizaban con los generadores uniformes de c++11, pero dado que en las mediciones se disparaban los tiempos de ejecucion al usar la libreria random, utilizamos el `rand()` legacy de C con una semilla inicial `time(NULL)`.

Por otro lado, el invariante de Dijkstra nos asegura que tomando el mínimo en cada iteración, podemos sacarlo de la cola y estar seguros de que no volverá a ser actualizado (por principio de optimalidad de Bellman aplicado a caminos mínimos). El hecho tomar uno aleatorio no nos asegura esto, por lo tanto cada nodo puede ser visitado y encolado más de una vez. En particular cada nodo es encolado tantas veces como pueda ser mejorado por todos sus vecinos, y sin tener alguna tabla de nodos visitados, el algoritmo itera hasta no poder mejorar más ningún nodo, momento en el que se vacía la cola y de esa forma llega a la solución golosa determinística. Notamos esto prematuramente en la experimentación al obtener resultados idénticos entre la heurística golosa normal y aleatorizada en absolutamente todos los casos y decidimos solucionarlo marcando cada vez que un nodo ingresa en la cola de prioridad y restringirlo a una sola vez, mediante un *vector* $< bool >$. De esta forma nos aseguramos de esto, y una vez implementada esta mejora, comenzamos a obtener soluciones distintas entre ambos algoritmos, y pudimos ver distintos resultados al variar el parámetro β (mientras más pequeño, más se acerca al resultado determinístico).

```
while !cola =  $\emptyset$  do
  if tipo_ejecucion == deterministico then
    nodo_minimo = minimo(cola)
  else if tipo_ejecucion == por_cantidad then
    int random = random(0, min{cola.size(), parametro_beta} - 1)
    minimo = cola[random] ▷  $O(random)$ , no es iterador de acceso aleatorio
  else if tipo_ejecucion == por_valor then
    lista_nodo_candidatos =  $\emptyset$ 
    int i = 0
    while i < tamano(cola) do
      if cola[i] ≤ valor_limite then
        agregar(cola[i], candidatos)
      end if
    end while
```

```

    int random = random(0, min{cola.size(), parametro_beta} - 1)
    minimo = candidatos[random]
    visitados[minimo] = true
end if
end while

```

No fue necesario realizar modificaciones a la heurística de búsqueda local.

5.2. Criterio de terminación

Fijada ya la heurística golosa aleatorizada, y la heurística de búsqueda local, queda definir por el criterio de terminación.

Hay dos criterios implementados:

1. Cantidad de iteraciones límite (fijo o variable)
2. Cantidad de iteraciones sin mejora consecutivas

Cantidad de iteraciones límite, como su nombre lo indica itera hasta un límite dado, ya sea una constante, o una variable del problema, por ejemplo la cantidad de nodos del grafo. Cantidad iteraciones sin mejora corta la iteración cuando se haya alcanzado una cantidad de iteraciones mínimas sin que haya habido alguna mejora en el camino.

5.3. Consideraciones

Dada la naturaleza aleatoria de la heurística greedy aleatorizada, en una cantidad de casos despreciable en cuanto al total de experimentos (pero aun así, ocurrieron), la heurística, aún habiendo un camino factible en el grafo, no pudo proporcionar una solución. Por esto, decidimos validar la solución obtenida del algoritmo goloso, y en caso de que no sea válida, ejecutarlo nuevamente hasta llegar a un tope de iteraciones fijo. Superado este tope de iteraciones, ajustamos el parámetro β de GRASP, en el caso de que la búsqueda golosa sea por cantidad, lo decrementamos en 1, en el caso por valor, simplemente lo fijamos en 0 lo cual equivale a la búsqueda golosa determinística.

5.4. Pseudocódigo

Nota: la entrada *criterios* se refiere a las variables:

1. *tipo_golosa* : el tipo de ejecución para la parte golosa.
2. *tipo_bq* : el tipo de ejecución para la búsqueda local
3. β : el parámetro para armar la RCL de la parte golosa.
4. *criterio_terminacion* : el criterio de terminación de GRASP.
5. *max_its* : la cantidad de iteraciones límite del primer criterio de terminación.
6. *max_its_sin_mejora* : la cantidad de iteraciones en la cual el segundo criterio debe cortar el algoritmo sin obtener mejora.
7. *bad_rgreedy_its* : la cantidad de iteraciones consecutivas para las cuales corremos la heurística golosa aleatorizada hasta que de una solución factible.

```

1: procedure Sol_GRASP(Grafo  $g$ , vertice  $v_1$ , vertice  $v_2$ , int  $k$ , criterios)  $\rightarrow$  lista  $\langle$  eje  $\rangle$  camino
2:   bool condicion_terminacion = false
3:   lista  $\langle$  eje  $\rangle$  mejor_solucion =  $\emptyset$ 
4:   int costo_mejor_solucion =  $\infty$ 

```



```

5:  lista < eje > camino =  $\emptyset$ 
6:  int cant_iters = 0
7:  int cant_iters_sin_mejora = 0
8:  int cant_iters_sin_sol_rgreedy_factible = 0
9:  bool sol_valida_rgreedy = false
10: vector < pair < int, costo > > mejora_iters_grasp

11:  while !condicion_terminacion do
12:      camino = solucion_golosa( $g, v_1, v_2, tipo\_golosa, \beta$ )
13:      sol_valida_rgreedy = validar_solucion(camino) ▷ Obtenemos solucion greedy

14:      if haysolucion then
15:          if sol_valida_rgreedy then ▷ Validamos si es factible
16:              cant_iters_sin_sol_rgreedy_factible = 0
17:              int mejora_iteracion_actual = 0
18:              int cant_iters_bqlocal = 0

19:              while mejora_iteracion_actual > 0 do ▷ Aplicamos Busqueda Local
20:                  mejora_iteracion_actual = busqueda_local( $g, tipo\_bq, camino$ )
21:                  cant_iters_bqlocal ++
22:              end while
23:              int costo_sol_actual = costo_w2(camino)

24:              if costo_sol_actual < costo_mejor_solucion then ▷ Reemplazamos si es mejor
solucion
25:                  if cant_iters > 0 then
26:                      agregar(mejora_iters_grasp, par < cant_iters, costo_mejor_solucion - costo_solucion_act
)
27:                  end if
28:                  costo_mejor_solucion = costo_sol_actual
29:                  mejor_solucion = camino
30:                  cant_iters_sin_mejora = 0
31:              else
32:                  if cant_iters > 0 then
33:                      agregar(mejora_iters_grasp, par < cant_iters, 0 >)
34:                  end if
35:                  cant_iters_sin_mejora ++
36:              end if
37:              cant_iters ++
38:          else ▷ sol_valida_rgreedy = false
39:              cant_iters_sin_sol_greedy_rand_factible ++
40:              if cant_iters_sin_sol_greedy_rand_factible  $\geq$  bad_rgreedy_its then
41:                  if tipo_golosa == por_cantidad then ▷ Maximo de its de greedy sin Solucion
42:                      if parametro_beta  $\geq$  2 then ▷ Ajustamos parametros de GRASP
43:                          parametro_beta --
44:                      end if
45:                  else if tipo_golosa == por_valor then
46:                      parametro_beta = 0
47:                  end if
48:                  cant_iters_sin_sol_greedy_rand_factible = 0
49:              end if
50:          end if
51:      else ▷ No hay Solucion
52:          break

```

```

53:     end if
54:     if criterio_terminacion == 1 then                                ▷ Si se cumple el criterio de terminacion
55:         condicion_terminacion = (cant_iters < max_its)
56:     else if criterio_terminacion == 2 then
57:         condicion_terminacion = (cant_iters_sin_mejora < max_its_sin_mejora)
58:     end if
59: end while
60:     return mejor_solucion
61: end procedure

```

5.5. Análisis de complejidad

A continuación realizaremos el análisis de complejidad teórica de una iteración de la metaheurística GRASP.

Dentro del ciclo principal, la primera instrucción es generar una solución golosa aleatorizada inicial. La complejidad de la heurística golosa determinística es de $O(n^2 + m \log n)$. La heurística golosa aleatorizada difiere de la determinística en la elección del nodo a desencolar, en caso de ser RCL por cantidad, extraer un elemento random de la cola cuesta (*avanzar el iterador de la cola* $\min\{\beta, n\}$ veces) + $O(1)$ eliminar, en caso de ser por valor, se filtra toda la cola, por lo que cuesta $O(n)$. El hecho de marcar los nodos y encolarlos una sola vez nos indica que el ciclo *while* externo itera n veces.

El ciclo *for* interno no recibió modificaciones excepto por un condicional que se ejecuta en tiempo constante. El peor de los casos se da cuando la cota K es de mayor peso a cualquier camino simple del grafo y este condicional es siempre verdadero, por lo que el ciclo se vuelve análogo al de Dijkstra y ejecuta en $O(m \log n)$ (sabemos que cada arista se analiza una vez porque marcamos los nodos).

Por lo tanto en caso de ser RCL por valor la complejidad es de: $O(n^2 + m * \log n + \beta * n)$ y en caso de ser por cantidad es de $O(n^2 + m \log n)$. El parámetro β puede ser acotado por n , dado que nunca va a poder armarse una RCL con más de n candidatos, ya que cada nodo está en la cola a lo sumo una vez, con lo cual la complejidad de la heurística aleatorizada no difiere de la determinística.

Acto seguido se valida si esta solución es factible, en tiempo constante, se declaran enteros y se procede a ejecutar la búsqueda local, cuya complejidad es de $O(k * n^2)$, siendo k la cantidad de iteraciones hasta cumplida la condición de terminación. Lo que resta son simplemente condicionales y asignaciones de tiempo constante, lo que nos da un resultado de $O(n^2 + m \log n + k * (n^2))$.

5.6. Experimentacion: Mediciones de Performance

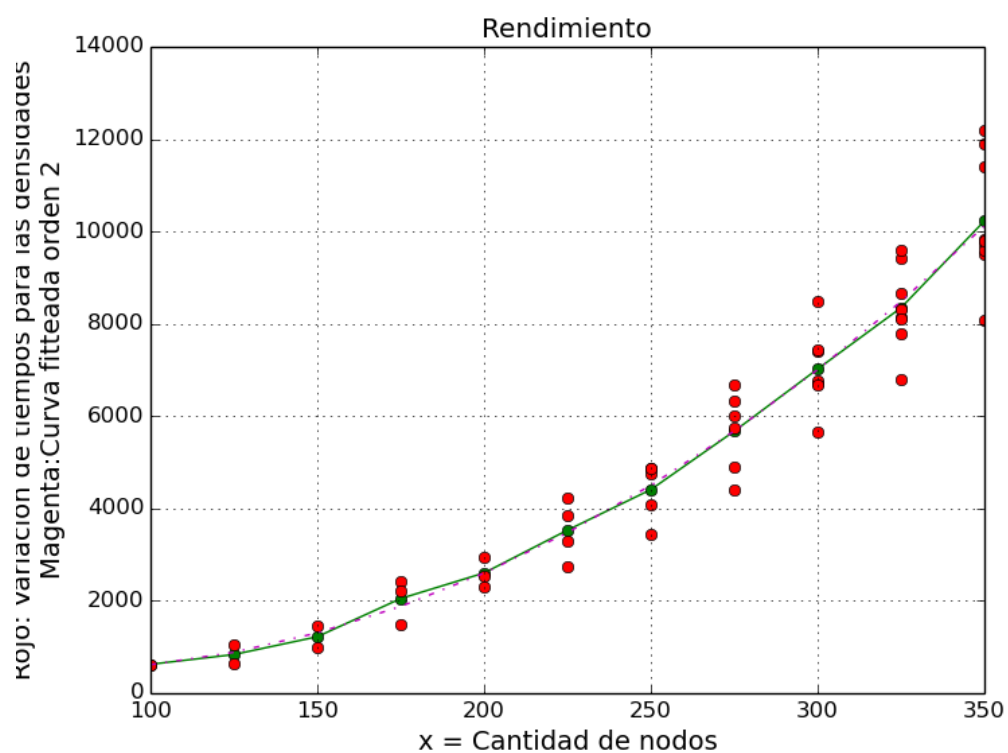
A continuacion presentamos los resultados de los experimentos, análogos a los de los algoritmos anteriores, con la salvedad de que para las mismas instancias de grafos, analizaremos primero con RCL por cantidad, con $\beta = n$ (peor caso), y luego con RCL por valor, con $\beta =$

5.6.1. Rendimiento para grafos con densidad cuadratica de aristas

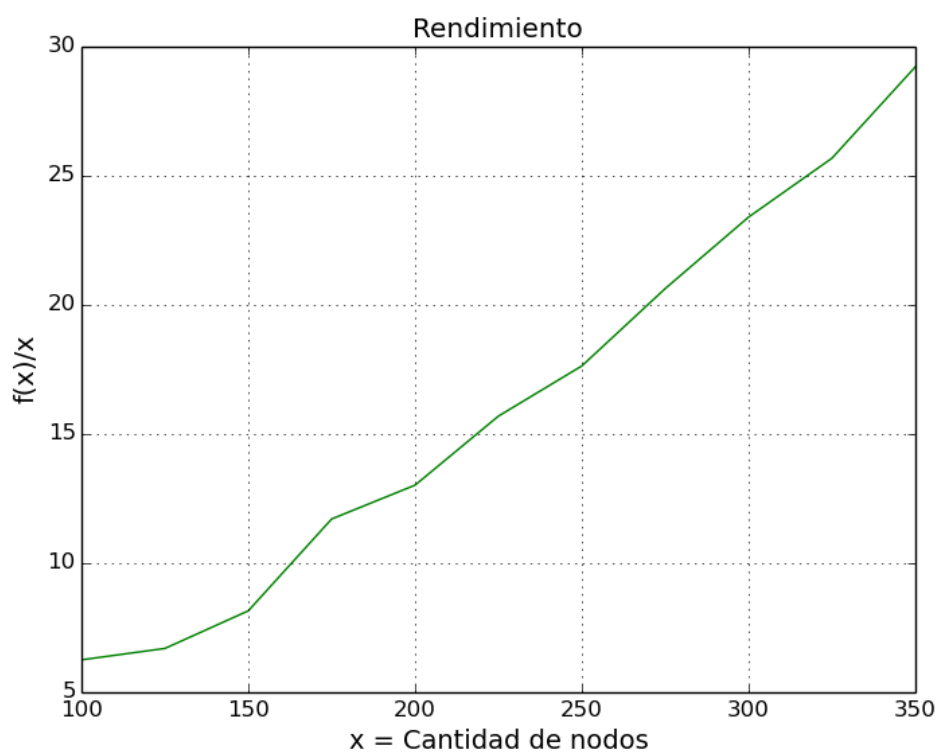
- cant nodos min = 200
- cant nodos max = 350
- peso maximo w1 = 250
- peso maximo w2 = 400
- step nodos = 25
- step aristas = 2500
- aristas minimas = $\frac{n*(n-1)}{8}$
- aristas maximas = $\frac{n*(n-1)}{3}$

Tiempo de ejecución en microsegundos para esta familia

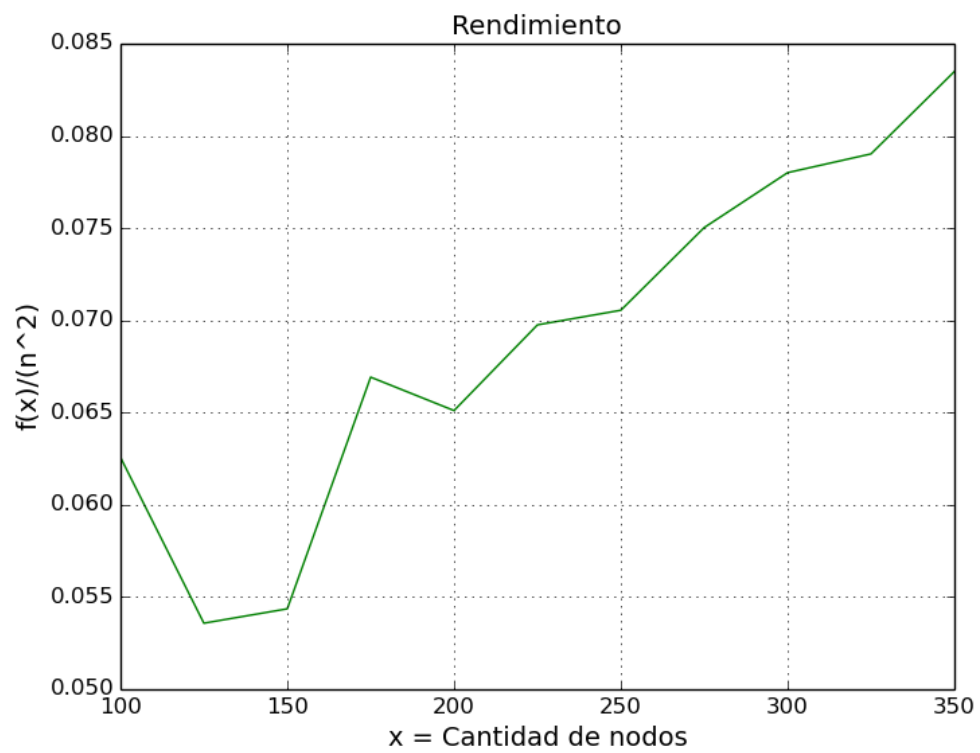
$$y = f(x)$$



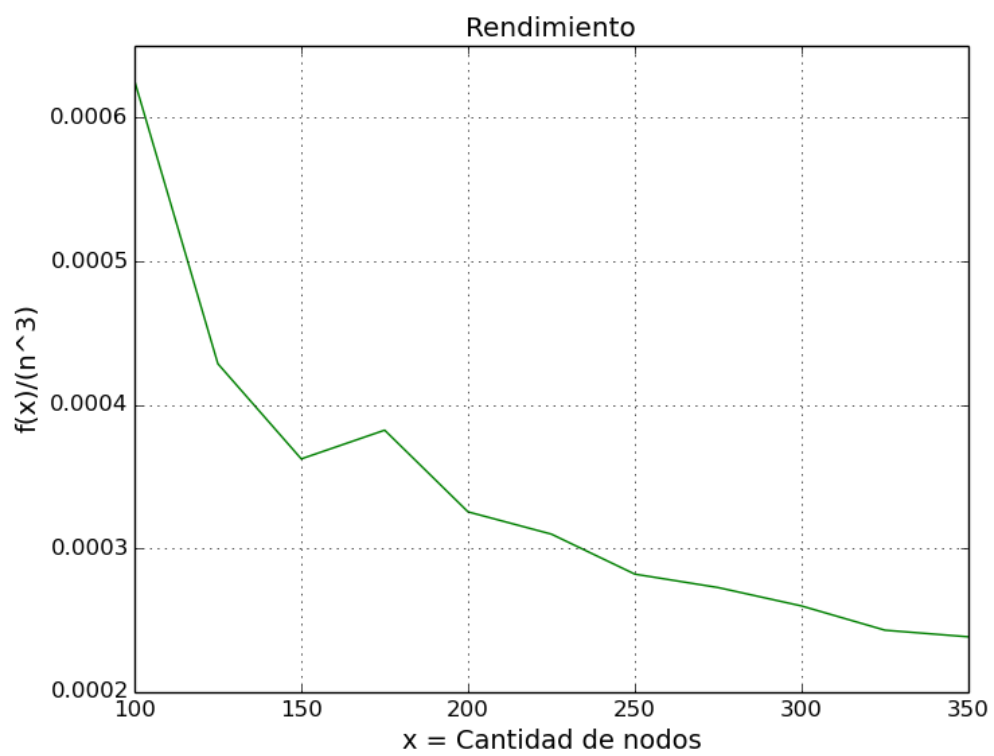
$$y = f(x)/x$$



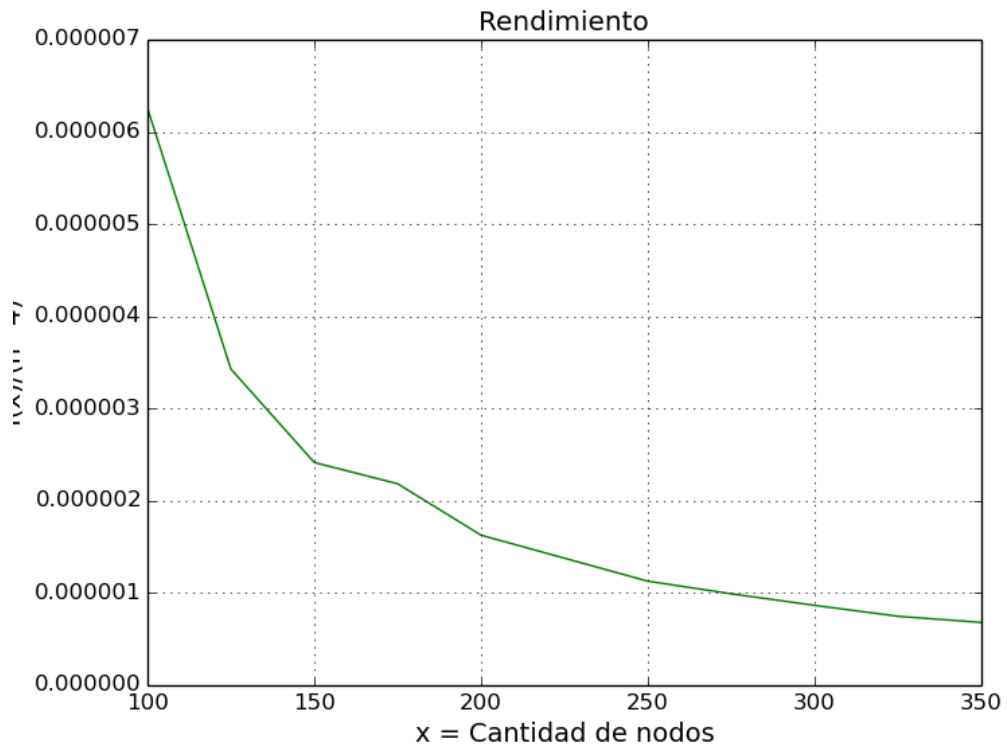
$$y = f(x)/x^2$$



$$y = f(x)/x^3$$



$$y = f(x)/x^4$$

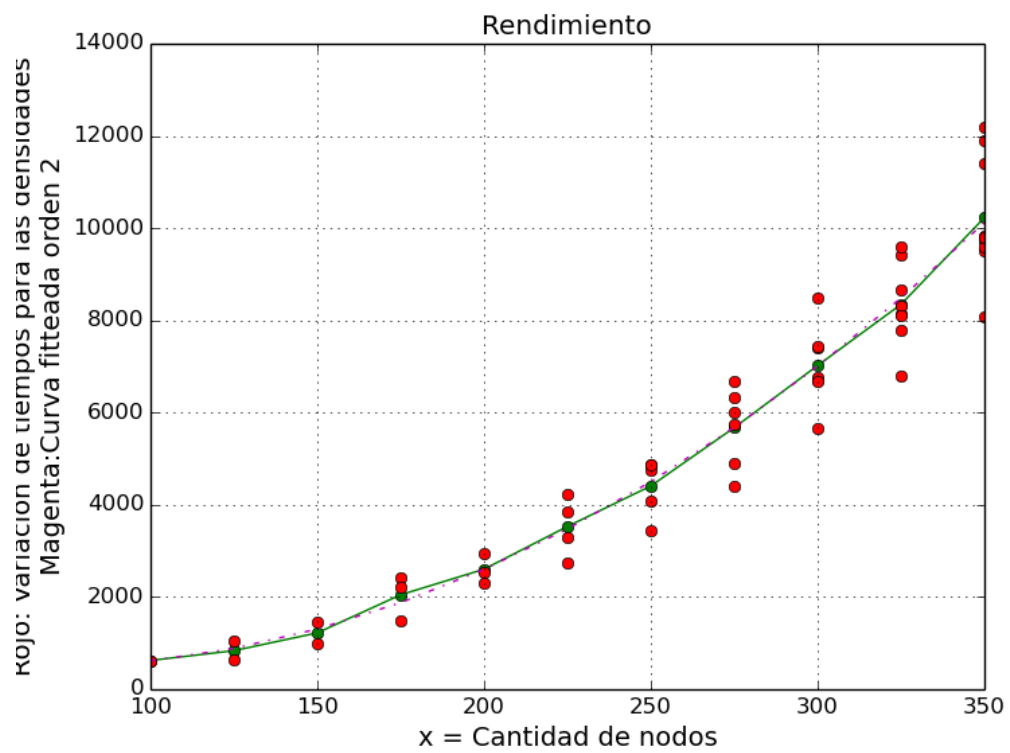


5.6.2. Rendimiento para grafos con densidad lineal de aristas

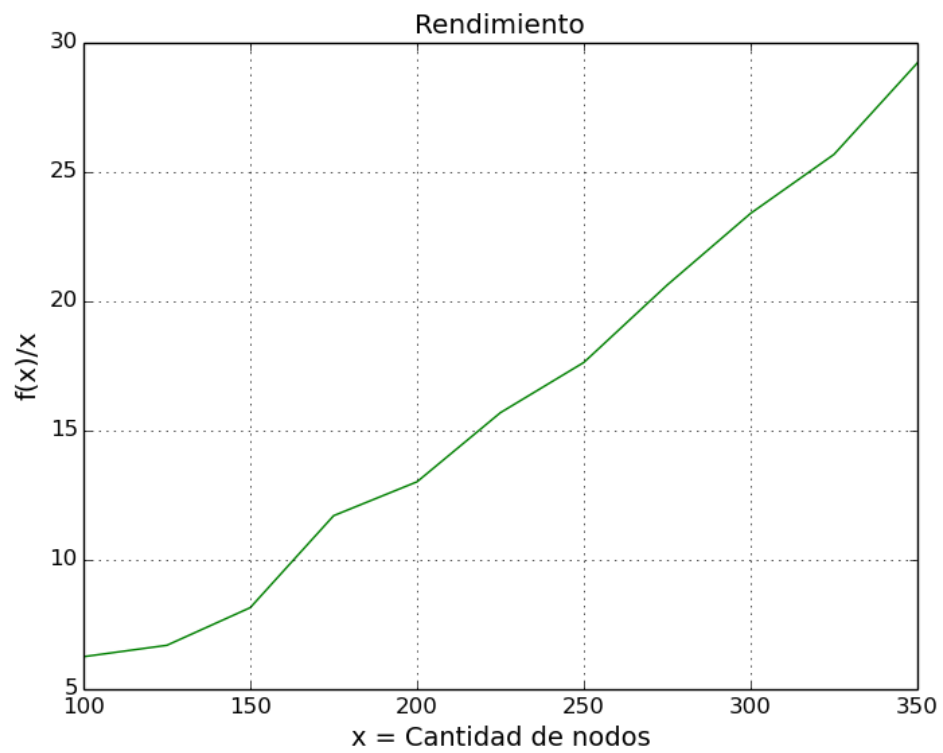
- cant nodos min = 200
- cant nodos max = 2000
- peso maximo w1 = 200
- peso maximo w2 = 200
- step nodos = 200
- step aristas = 2500
- aristas minimas = $n - 1$
- aristas maximas = $10 * n$

Tiempo de ejecución en microsegundos para esta familia

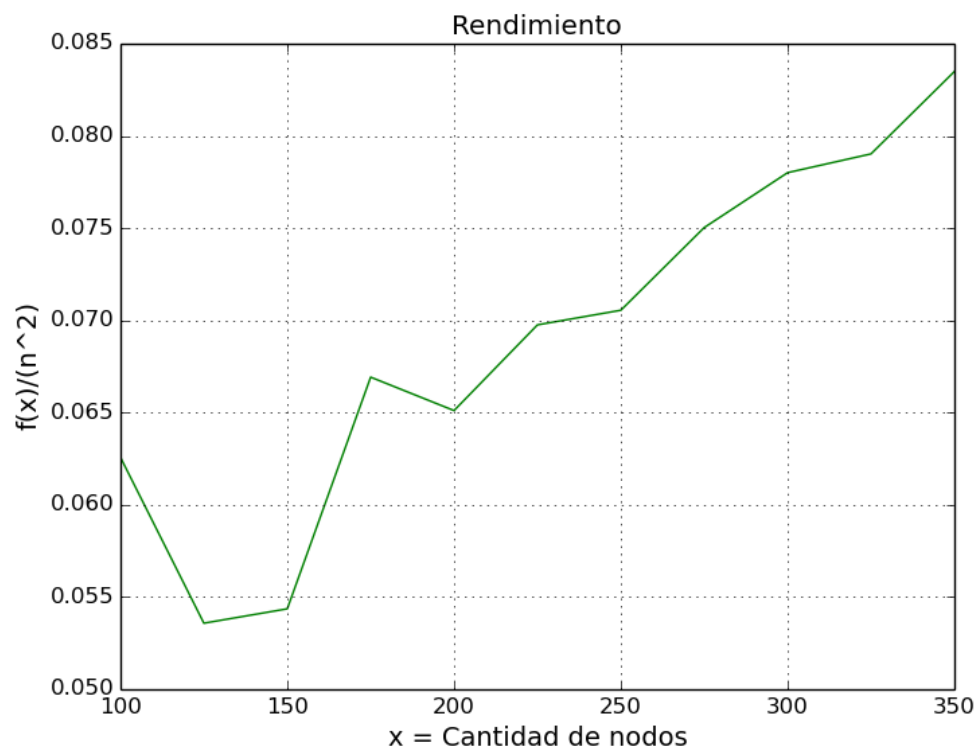
$$y = f(x)$$



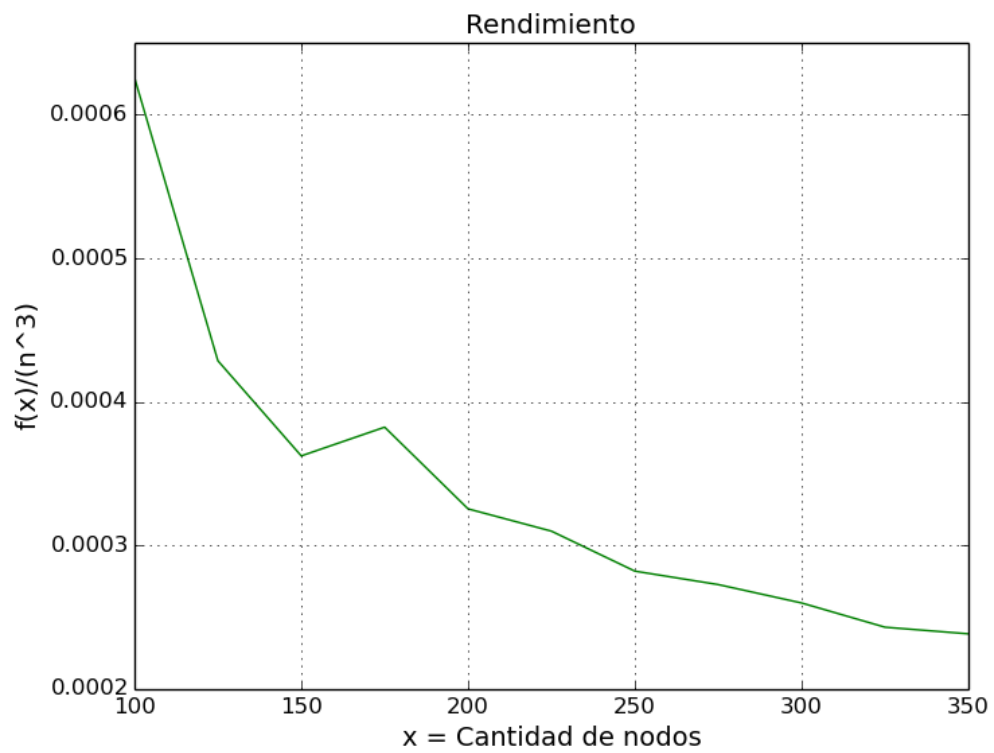
$$y = f(x)/x$$



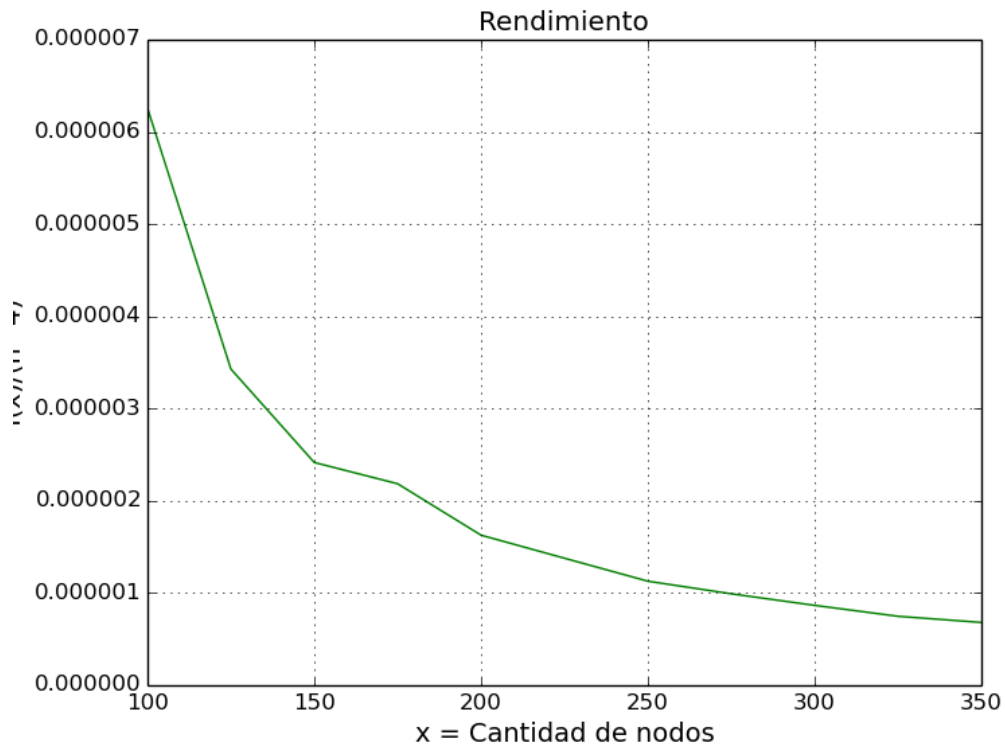
$$y = f(x)/x^2$$



$$y = f(x)/x^3$$



$$y = f(x)/x^4$$

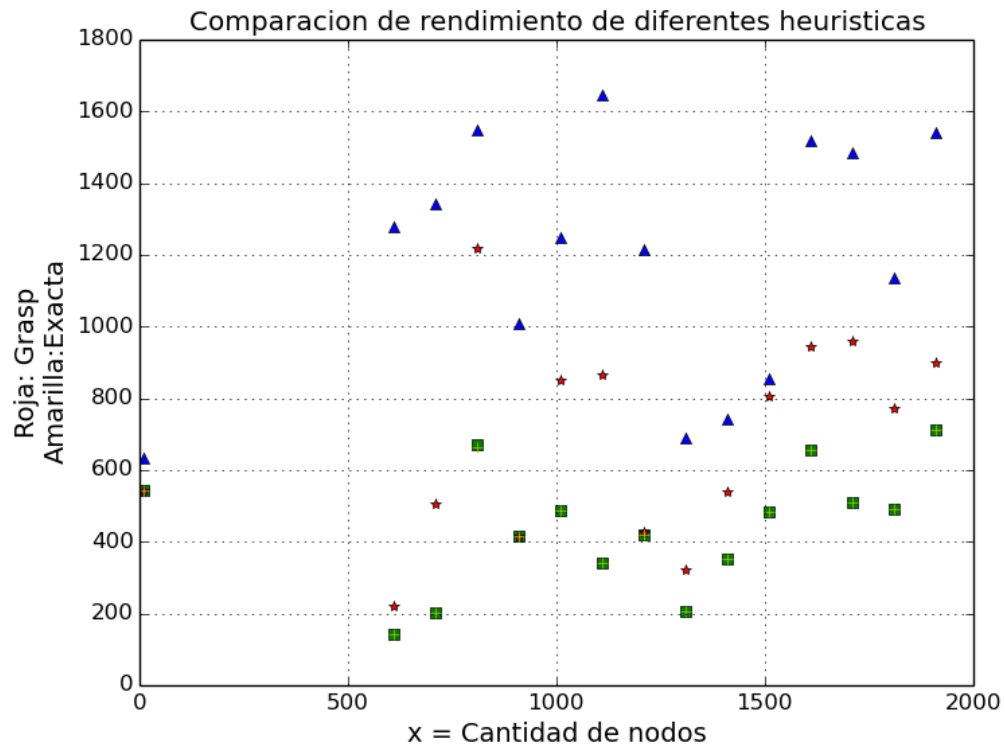


5.7. Experimentación de optimalidad

A continuación los experimentos de análisis de solución de la metaheurística, comparada con el algoritmo exacto y las demás. Para estos experimentos además, fijando un costo w_1 máximo a las aristas, variamos la cota K de los grafos para verificar optimalidad-

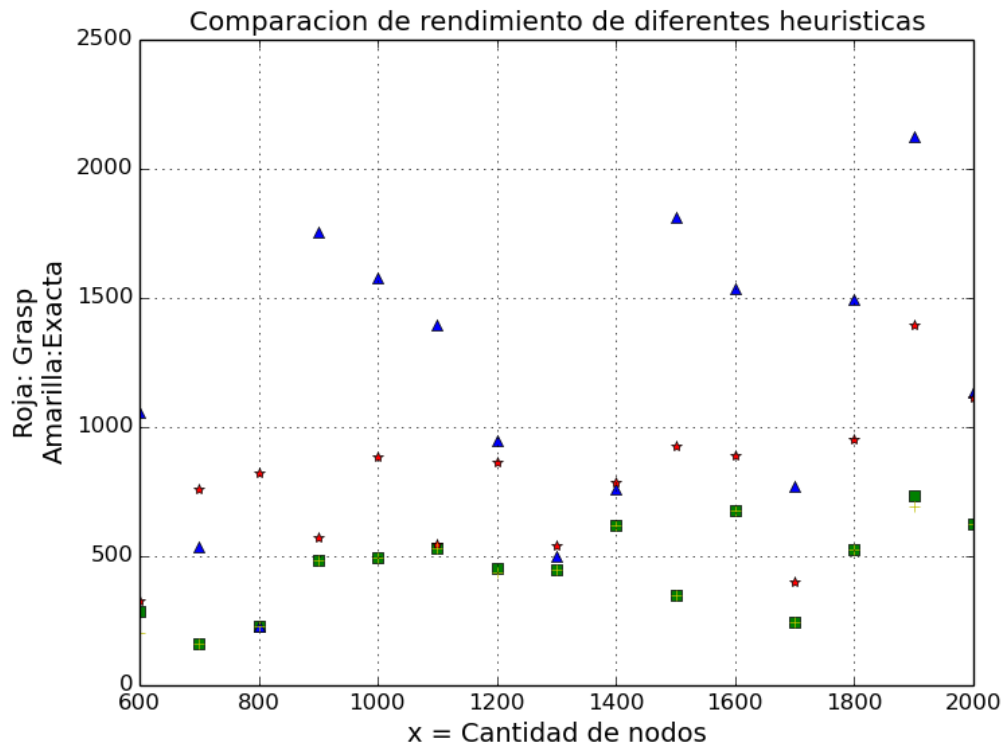
5.7.1. Optimalidad para grafos lineales con $w_1 = 120$

- cant nodos min = 200
- cant nodos max = 2000
- peso maximo $w_1 = 250$
- peso maximo $w_2 = 400$
- limit $w_1 = 120$
- step nodos = 100
- step aristas = 2500
- aristas minimas = $2 * n$
- aristas maximas = $20 * n$



5.7.2. Optimalidad para grafos lineales con $w1 = 200$

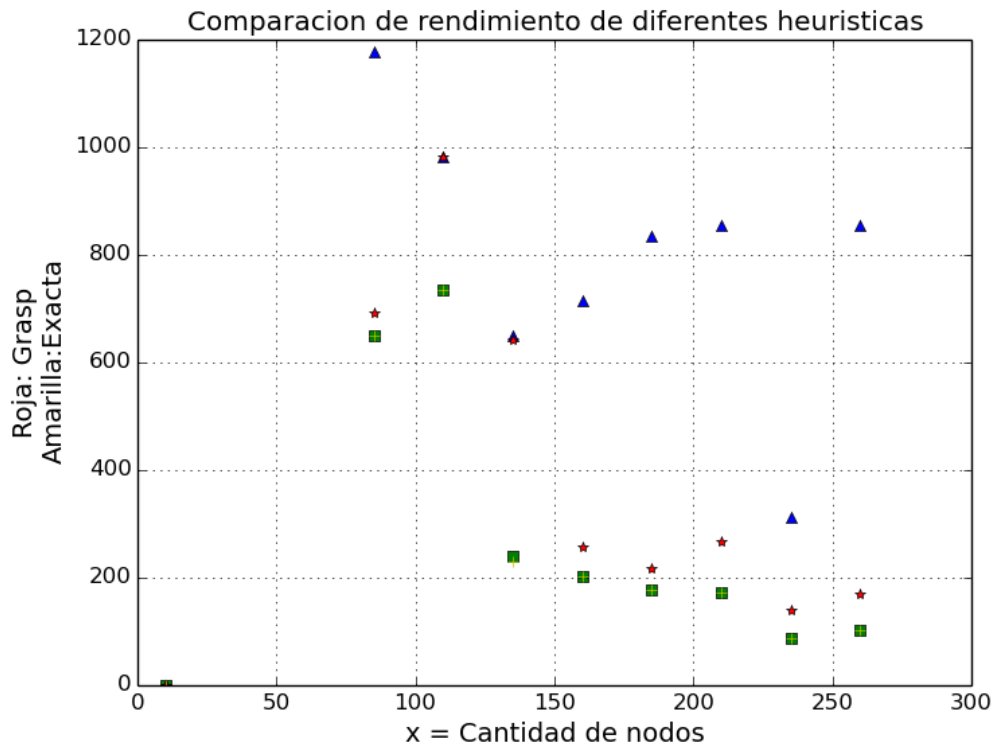
- cant nodos min = 200
- cant nodos max = 2000
- peso maximo $w1 = 250$
- peso maximo $w2 = 400$
- limit $w1 = 200$
- step nodos = 100
- step aristas = 2500
- aristas minimas = $2 * n$
- aristas maximas = $20 * n$



Cantidad de tests realizados: 56
 Tiempo promedio microsegundos heuristica: 63565.671
 Tiempo promedio microsegundos exacto: 31439.678
 Porcentaje de aciertos(cantidad de veces que GRASP da la sol exacta/
 cantidad de tests hechos): 14.285
 Porcentaje de alejamiento de la heuristica a la solucion exacta promedio
 entre grasp y exacta: 109.460
 Desviacion estandar del alejamiento de la heuristica a la solucion exacta
 promedio entre grasp y exacta: 121.493
 Minimo alejamiento porcentual entre grasp y exacta: 0
 Maximo alejamiento porcentual entre grasp y exacta: 522.400

5.7.3. Optimalidad para grafos cuadráticos con $w_1 = 150$

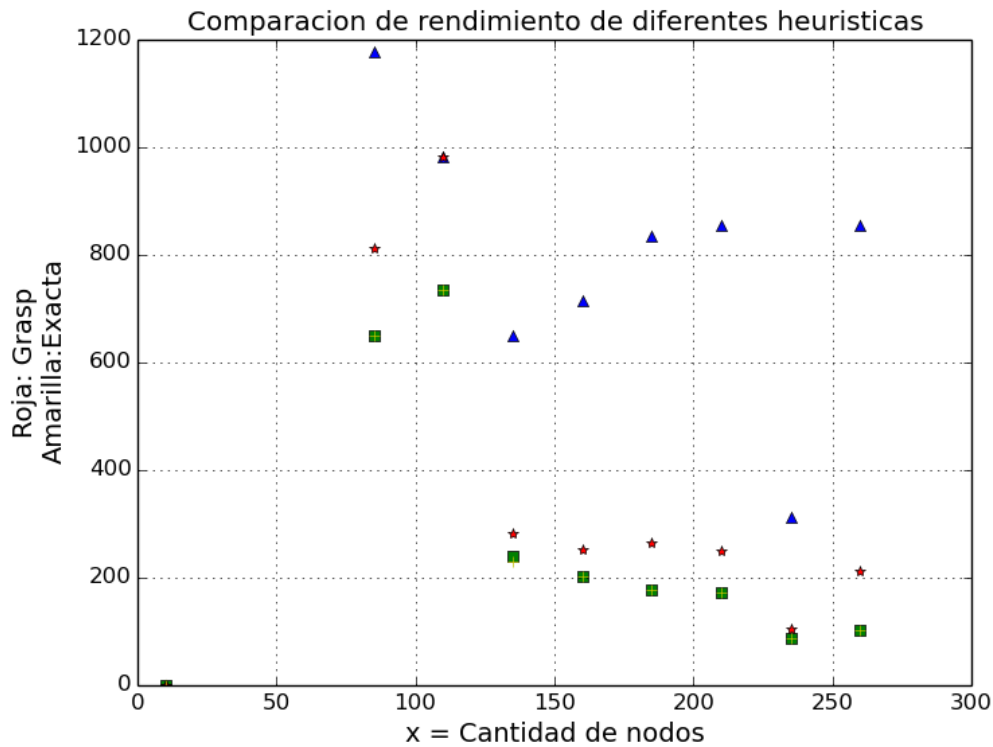
- cant nodos min = 10
- cant nodos max = 260
- peso maximo $w_1 = 250$
- peso maximo $w_2 = 400$
- limit $w_1 = 150$
- step nodos = 25
- step aristas = 5000
- aristas minimas = $(n * (n - 1)) = /12$
- aristas maximas = $n * (n - 1)) = /2$



Cantidad de tests realizados: 26
 Tiempo promedio microsegundos heuristica: 2877.366
 Tiempo promedio microsegundos exacto: 5129.884
 Porcentaje de aciertos(cantidad de veces que GRASP da la sol exacta/
 cantidad de tests hechos): 42.307
 Porcentaje de alejamiento de la heuristica a la solucion exacta promedio
 entre grasp y exacta: 50.108
 Desviacion estandar del alejamiento de la heuristica a la solucion exacta
 promedio entre grasp y exacta: 69.2576
 Minimo alejamiento porcentual entre grasp y exacta: 0
 Maximo alejamiento porcentual entre grasp y exacta: 259.600

5.7.4. Optimalidad para grafos cuadráticos con $w_1 = 120$

- cant nodos min = 10
- cant nodos max = 260
- peso maximo $w_1 = 250$
- peso maximo $w_2 = 400$
- limit $w_1 = 120$
- step nodos = 25
- step aristas = 5000
- aristas minimas = $(n * (n - 1)) = /12$
- aristas maximas = $n * (n - 1)) = /2$



Cantidad de tests realizados: 26
 Tiempo promedio microsegundos heurística: 2751.938
 Tiempo promedio microsegundos exacto: 4685.692
 Porcentaje de aciertos(cantidad de veces que GRASP da la sol exacta/
 cantidad de tests hechos): 53.846
 Porcentaje de alejamiento de la heurística a la solución exacta promedio
 entre grasp y exacta: 81.088
 Desviación estándar del alejamiento de la heurística a la solución exacta
 promedio entre grasp y exacta: 189.693
 Mínimo alejamiento porcentual entre grasp y exacta: 0
 Máximo alejamiento porcentual entre grasp y exacta: 789.400

Estas pruebas fueron resueltas con el objetivo de analizar como varián las soluciones de la heurística según los caminos factibles que delimita la cota K . A continuación, fijaremos la cota K en 120 (que fue la que mostró soluciones más interesantes en la práctica) y variaremos el parámetro β , por cantidad y luego por valor. Condensaremos las familias de análisis en un mismo gráfico, y las corremos sobre un mismo conjunto de instancias.

5.7.5. Optimalidad para grafos con RCL por cantidad

- cant nodos min = 10
- cant nodos max = 260
- peso maximo w1 = 250
- peso maximo w2 = 400
- limit w1 = 150
- step nodos = 25
- step aristas = 20000

- aristas minimas = n
- aristas maximas = $n * (n - 1) = /2$

$$\beta = n/4$$

Cantidad de tests realizados: 13
Tiempo promedio microsegundos heuristica: 4781.004
Tiempo promedio microsegundos exacto: 15248.461
Porcentaje de aciertos(cantidad de veces que GRASP da la sol exacta/ cantidad de tests hechos): 38.461
Porcentaje de alejamiento de la heuristica a la solucion exacta promedio entre grasp y exacta: 68.915
Desviacion estandar del alejamiento de la heuristica a la solucion exacta promedio entre grasp y exacta: 116.131
Minimo alejamiento porcentual entre grasp y exacta: 0
Maximo alejamiento porcentual entre grasp y exacta: 396.900

$$\beta = n/32$$

Cantidad de tests realizados: 13
Tiempo promedio microsegundos heuristica: 4558.685
Tiempo promedio microsegundos exacto: 14349.692
Porcentaje de aciertos(cantidad de veces que GRASP da la sol exacta/ cantidad de tests hechos): 38.461
Porcentaje de alejamiento de la heuristica a la solucion exacta promedio entre grasp y exacta: 21.646
Desviacion estandar del alejamiento de la heuristica a la solucion exacta promedio entre grasp y exacta: 28.0819
Minimo alejamiento porcentual entre grasp y exacta: 0
Maximo alejamiento porcentual entre grasp y exacta: 78.100

$$\beta = n/64$$

Cantidad de tests realizados: 13
Tiempo promedio microsegundos heuristica: 4516.103
Tiempo promedio microsegundos exacto: 14427.461
Porcentaje de aciertos(cantidad de veces que GRASP da la sol exacta/ cantidad de tests hechos): 46.153
Porcentaje de alejamiento de la heuristica a la solucion exacta promedio entre grasp y exacta: 16.130
Desviacion estandar del alejamiento de la heuristica a la solucion exacta promedio entre grasp y exacta: 23.74
Minimo alejamiento porcentual entre grasp y exacta: 0
Maximo alejamiento porcentual entre grasp y exacta: 72.000

Vemos que variar el parámetro β y dividiéndolo sucesivamente por potencias de dos va a justando el alejamiento de los no aciertos y aumenta la cantidad de aciertos.

Para un experimento en particular $\beta = n/128$, vemos que iguala en cantidad de aciertos a la heurística golosa y reduce el alejamiento de los no aciertos.

- cant nodos min = 10
- cant nodos max = 280
- peso maximo w1 = 250
- peso maximo w2 = 400

- limit w1 = 150
- step nodos = 25
- step aristas = 20000
- aristas minimas = n
- aristas maximas = $n * (n - 1) = /2$

Golosa:

Porcentaje de aciertos(cantidad de veces que GOLOSA da la sol exacta/
cantidad de tests hechos): 83.018

Porcentaje de alejamiento de la heuristica a la solucion exacta promedio
entre golosa y exacta: 4.990

Desviacion estandar del alejamiento de la heuristica a la solucion exacta
promedio entre golosa y exacta: 18.1754

GRASP:

Porcentaje de aciertos(cantidad de veces que GRASP da la sol exacta/
cantidad de tests hechos): 83.018

Porcentaje de alejamiento de la heuristica a la solucion exacta promedio
entre grasp y exacta: 3.611

Desviacion estandar del alejamiento de la heuristica a la solucion exacta
promedio entre grasp y exacta: 12.1007

[frame=single]

Ahora variamos β en RCL por valor

$\beta = 0,9$

Cantidad de tests realizados: 14

Tiempo promedio microsegundos heuristica: 4620.666

Tiempo promedio microsegundos exacto: 13796.214

Porcentaje de aciertos(cantidad de veces que GRASP da la sol exacta/
cantidad de tests hechos): 64.285

Porcentaje de alejamiento de la heuristica a la solucion exacta promedio
entre grasp y exacta: 18.246

Desviacion estandar del alejamiento de la heuristica a la solucion exacta
promedio entre grasp y exacta: 44.6027

Minimo alejamiento porcentual entre grasp y exacta: 0

Maximo alejamiento porcentual entre grasp y exacta: 175.000

$\beta = 0,8$

Cantidad de tests realizados: 14

Tiempo promedio microsegundos heuristica: 4600.035

Tiempo promedio microsegundos exacto: 14057.071

Porcentaje de aciertos(cantidad de veces que GRASP da la sol exacta/
cantidad de tests hechos): 71.428

Porcentaje de alejamiento de la heuristica a la solucion exacta promedio
entre grasp y exacta: 3.376

Desviacion estandar del alejamiento de la heuristica a la solucion exacta
promedio entre grasp y exacta: 5.34277

Minimo alejamiento porcentual entre grasp y exacta: 0

Maximo alejamiento porcentual entre grasp y exacta: 17.000

$\beta = 0,6$

```
\begin{}
Cantidad de tests realizados: 14
Tiempo promedio microsegundos heuristica: 4568.095
Tiempo promedio microsegundos exacto: 13775.857
Porcentaje de aciertos(cantidad de veces que GRASP da la sol exacta/
    cantidad de tests hechos): 78.571
Porcentaje de alejamiento de la heuristica a la solucion exacta promedio
    entre grasp y exacta: 2.615
Desviacion estandar del alejamiento de la heuristica a la solucion exacta
    promedio entre grasp y exacta: 5.04769
Minimo alejamiento porcentual entre grasp y exacta: 0
Maximo alejamiento porcentual entre grasp y exacta: 17.000
```

Vemos como también reducir el valor límite causa que no sólo produzca más aciertos sino que también sus no aciertos se ajustan más.

6. Experimentacion General

En esta sección analizaremos la calidad de las heurísticas mediante la comparación y análisis estadístico de las soluciones, así también el tiempo insumido en obtener dichas soluciones, para diferentes grupos de instancias de grafos generados al azar.

6.1. Generacion de conjuntos de grafos aleatorios

6.1.1. Generador de grafos aleatorios

El generador aleatorio de grafos es un binario aparte de los algoritmos, que recibe como parámetros

- cantidad de nodos
- cantidad de aristas
- peso mínimo w_1
- peso máximo w_1
- peso mínimo w_2
- peso máximo w_2
- límite w_1

y devuelve por salida estándar una instancia del problema tal como está especificado el formato de entrada en el enunciado de este TP.

Nota: Los generadores de números aleatorios de este generador de grafos tienen distribución uniforme(Usan random de C++11).

La generación del grafo se realiza de la siguiente manera: Sean $n = \{\text{cantidad de nodos del grafo}\}$ y $m = \{\text{cantidad de aristas}\}$, se inicializa un vector *aristas* $= \langle (0, 1), \dots, (0, n - 1), (1, 2), \dots, (1, n - 1), \dots, (n - 2, n - 1) \rangle$ conteniendo todas las aristas posibles en el grafo(notar que como no es digrafo, no se repiten aristas simétricas, ni tampoco se asignan self-loops).

Luego se mezcla aleatoriamente este vector usando el **Algoritmo de Shuffle de Knuth o Fisher Yates Shuffle** y se imprime la cabecera del grafo a la salida estándar conteniendo los nodos origen, destino y el parámetro k generados como números aleatorios uniformes. Ahora basta tomar la cantidad m de aristas requeridas por parámetro y serializar la salida línea por línea, nuevamente generando números aleatorios con distribución uniforme sobre los rangos de pesos w_1 y w_2 pasados por parámetro. Finalmente se imprime un 0 indicando el final de la entrada.

6.1.2. Script generador de conjuntos de grafos

Se realizó un script el cual genera conjuntos de grafos usando el generador de la sección anterior, básicamente, se setean dos rangos, de cantidad de nodos y cantidad de aristas, y los parámetros fijos como límite w_1 , límites de los pesos, etc. El script genera iterativamente el conjunto de grafos llamando repetidamente al generador, notemos que podemos(y es lo que hicimos en los experimentos), poner la cantidad de aristas en función de la cantidad de nodos y de esta forma poder determinar la densidad de los conjuntos de grafos generados.

6.2. Scripts de optimalidad - Calculo de puntajes y estadísticas

Se realizaron diversos scripts para automatizar el análisis de optimalidad y performance, vamos a analizar el script de optimalidad que se ejecuto para obtener los resultados de esta sección.

Se trata de un script en bash que para cada instancia del conjunto de grafos generados aleatoriamente, ejecuta los 4 algoritmos (exacta, golosa, busqueda local, grasp) y va realizando calculos estadísticos de los resultados obtenidos, tanto de la solución como del tiempo consumido para obtenerla. Los análisis estadísticos que se realizan son:

- Tiempo promedio microsegundos consumido por el algoritmo
- Porcentaje de veces que la heurística da la solución óptima
- Desviación estándar de veces que la heurística da la sol. óptima
- Lejanía promedio de la solución obtenida a la solución óptima
- Desviación estándar de la lejanía de las soluciones entre la obtenida y la óptima
- Mínima y máxima lejanía obtenida en este conjunto de instancias

Nota: La lejanía entre 2 soluciones se mide haciendo el siguiente calculo: $100 * (\frac{\text{solucionHeuristica}}{\text{solucionOptima}} - 1)$ que indica en porcentaje cual es el ratio de distancia entre los dos valores del cociente.

Nota: Los calculos estadísticos (promedio y desviación estándar) se realizan sobre la lista de resultados obtenida de la ejecución secuencial y el calculo de la lejanía mencionado aquí arriba para cada uno de los algoritmos(exacto y heurística) sobre cada instancia del conjunto de pruebas.

Podemos considerar una especie de puntuación asignada a cada heurística viendo el porcentaje de optimalidad y también podemos analizar, que cuando no da la solución óptima, la lejanía a la óptima, con cierta dispersión sea adecuada, según el problema y el las necesidades del contexto donde se aplica.

6.3. Scripts de optimalidad - Graficos de optimalidad comparativos

El script de optimalidad también realiza un gráfico, en el cual puede verse en el eje X la cantidad de nodos de la instancia y en el eje Y, **un promedio de los pesos w_2 de soluciones para la variación de aristas para esa cantidad de nodos** de cada algoritmo corrido, distintas referencias y colores indican para cada cantidad de nodos, el valor w_2 de las soluciones obtenidas, en este gráfico podemos apreciar la distribución de las soluciones sobre el eje Y a medida que varía la cantidad de nodos.

Referencias del gráfico: Los triángulos azules representan las soluciones de búsqueda local, los signos + amarillos, representan las soluciones del algoritmo exacto, los cuadrados verdes simbolizan las soluciones del algoritmo goloso, y los asteriscos rojos representan las soluciones de GRASP.

6.4. Calidad de las heurísticas respecto a la solución exacta

Se corrieron los scripts de optimalidad con diferentes conjuntos de instancias de grafos aleatorios, y se realizaron los gráficos y el análisis estadístico correspondiente, a continuación se presentan los resultados.

6.4.1. Comparación Exacta-Golosa-Búsqueda Local-GRASP

Se eligieron, 3 diferentes densidades de grafos, y se corrieron los algoritmos con los scripts mencionados anteriormente, a continuación presentamos los resultados estadísticos, así también como los gráficos.

6.4.2. Grafos aleatorios de baja densidad de aristas

Parametros del experimento:

- Parametro Beta GRASP: $\frac{cantNodos}{128}$
- Cantidad de grafos analizados: 71
- Cantidad minima de nodos: 100
- Cantidad maxima de nodos: 1800
- Rango peso w_1 : [0..250]
- Rango peso w_2 : [0..400]
- Cota de w_1 : 200
- Cantidad minima de aristas: $n - 1$
- Cantidad maxima de aristas: $10 * n$

Resultados del analisis (Golosa):

Tiempo promedio microsegundos heuristica: 19636.985
Tiempo promedio microsegundos exacto: 16899.281
Heuristica da la solucion optima: 98.591% de los casos
Lejania promedio de la heuristica a la solucion optima: 0.261%
Desv. estandar de la lejania entre las soluciones: 2.19181
Minima lejania entre bqlocal y exacta: 0
Maxima lejania entre bqlocal y exacta: 18.600%

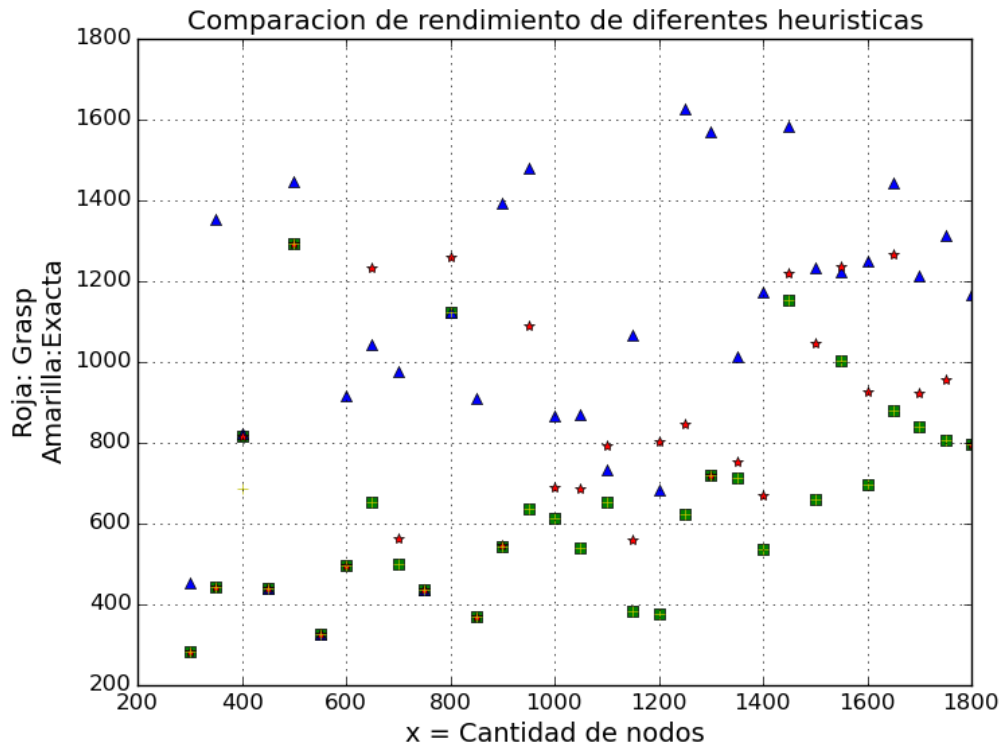
Resultados del analisis (Busqueda local):

Tiempo promedio microsegundos heuristica: 8091.119
Tiempo promedio microsegundos exacto: 16899.281
Heuristica da la solucion optima: 32.394% de los casos
Lejania promedio de la heuristica a la solucion optima: 87.626%
Desv. estandar de la lejania entre las soluciones: 118.585
Minima lejania entre bqlocal y exacta: 0
Maxima lejania entre bqlocal y exacta: 662.700%

Resultados del analisis (Metaheuristica GRASP):

Tiempo promedio microsegundos heuristica: 26709.723
Tiempo promedio microsegundos exacto: 16899.281
Heuristica da la solucion optima: 57.746% de los casos
Lejania promedio de la heuristica a la solucion optima: 33.373%
Desv. estandar de la lejania entre las soluciones: 70.8898
Minima lejania entre bqlocal y exacta: 0
Maxima lejania entre bqlocal y exacta: 453.300%

Distribucion de los resultados



6.4.3. Grafos aleatorios de intermedia densidad de aristas

Parametros del experimento:

- Parametro Beta GRASP: $\frac{cantNodos}{128}$
- Cantidad de tests realizados: 255
- Cantidad minima de nodos: 100
- Cantidad maxima de nodos: 600
- Rango peso w_1 : [0..250]
- Rango peso w_2 : [0..400]
- Cota de w_1 : 200
- Cantidad minima de aristas: $n * \sqrt{n}$
- Cantidad maxima de aristas: $5 * n * \sqrt{n}$

Resultados del analisis (Golosa):

Tiempo promedio microsegundos heuristica: 2700.713
 Tiempo promedio microsegundos exacto: 6803.960
 Heuristica da la solucion optima: 86.666% de los casos
 Lejania promedio de la heuristica a la solucion optima: 2.590%
 Desv. estandar de la lejania entre las soluciones: 9.77461
 Minima lejania entre bqlocal y exacta: 0
 Maxima lejania entre bqlocal y exacta: 79.000%

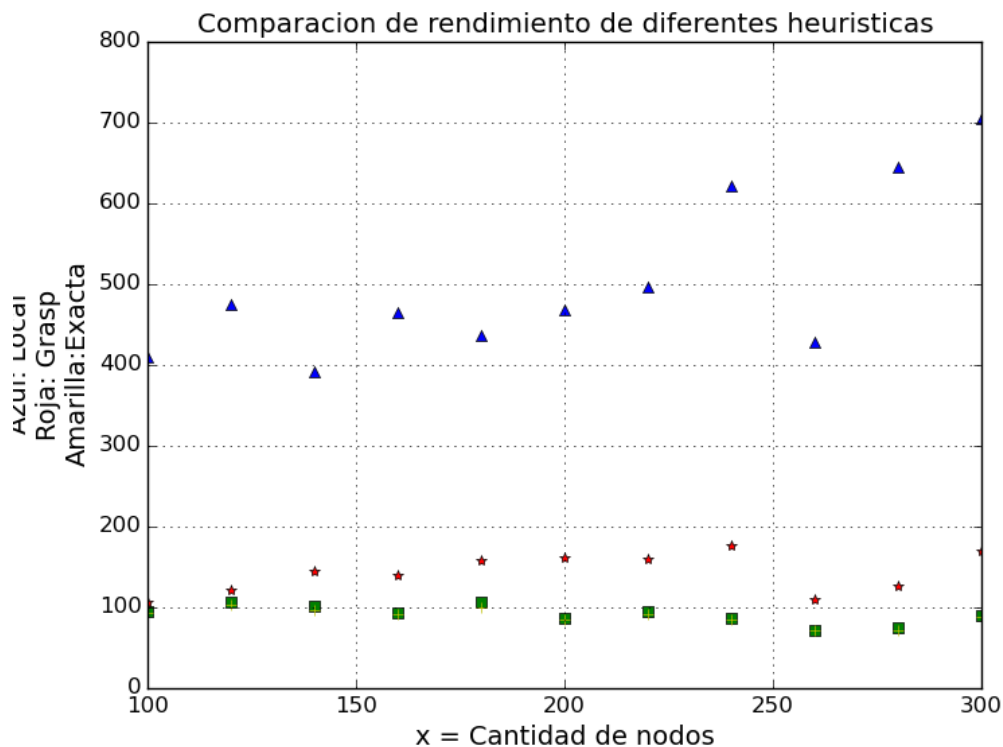
Resultados del analisis (Busqueda local):

Tiempo promedio microsegundos heuristica: 529.705
Tiempo promedio microsegundos exacto: 6803.960
Heuristica da la solucion optima: 10.588% de los casos
Lejania promedio de la heuristica a la solucion optima: 726.246%
Desv. estandar de la lejania entre las soluciones: 904.16
Minima lejania entre bqlocal y exacta: 0
Maxima lejania entre bqlocal y exacta: 5900.000%

Resultados del analisis (Metaheuristica GRASP):

Tiempo promedio microsegundos heuristica: 3645.660
Tiempo promedio microsegundos exacto: 6803.960
Heuristica da la solucion optima: 34.901% de los casos
Lejania promedio de la heuristica a la solucion optima: 94.901%
Desv. estandar de la lejania entre las soluciones: 172.781
Minima lejania entre bqlocal y exacta: 0
Maxima lejania entre bqlocal y exacta: 1541.800%

Distribucion de los resultados



6.4.4. Grafos aleatorios de alta densidad de aristas

Parametros del experimento:

- Parametro Beta GRASP: $\frac{cantNodos}{128}$
- Cantidad de grafos analizados: 53

- Cantidad minima de nodos: 10
- Cantidad maxima de nodos: 280
- Rango peso w_1 : [0..250]
- Rango peso w_2 : [0..400]
- Cota de w_1 : 200
- Cantidad minima de aristas: $\frac{n*(n-1)}{3}$
- Cantidad maxima de aristas: $\frac{n*(n-1)}{2}$

Resultados del analisis (Golosa):

Tiempo promedio microsegundos heuristica: 4569.584
 Tiempo promedio microsegundos exacto: 19571.283
 Heuristica da la solucion optima: 83.018% de los casos
 Lejania promedio de la heuristica a la solucion optima: 4.990%
 Desv. estandar de la lejania entre las soluciones: 18.1754
 Minima lejania entre bqlocal y exacta: 0
 Maxima lejania entre bqlocal y exacta: 110.300%

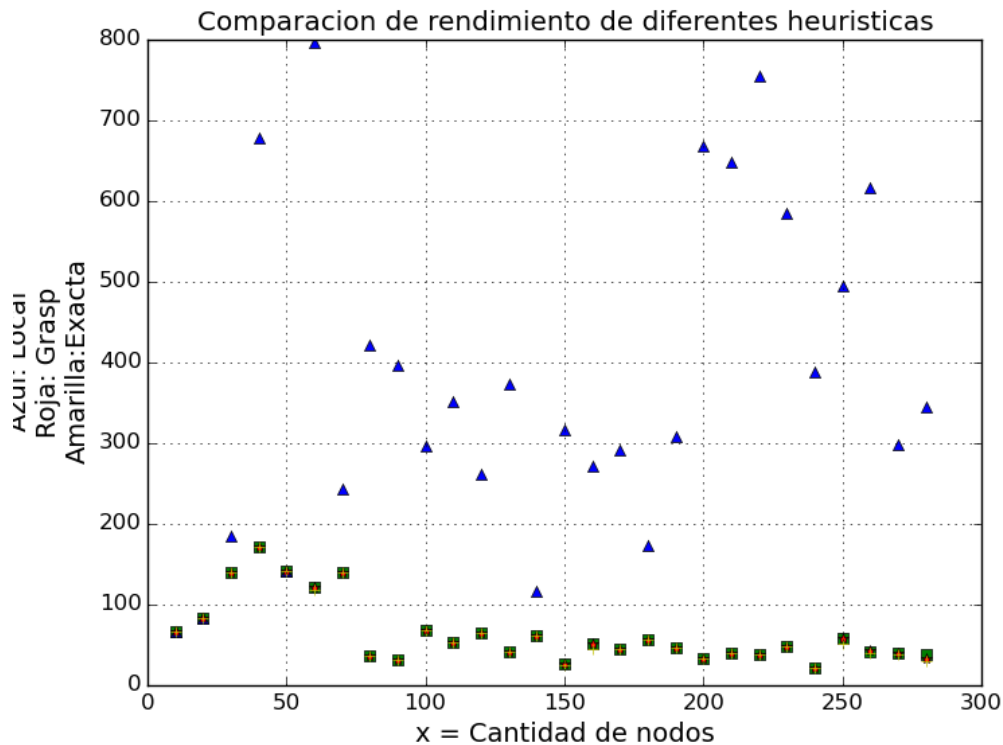
Resultados del analisis (Busqueda local):

Tiempo promedio microsegundos heuristica: 550.485
 Tiempo promedio microsegundos exacto: 19571.283
 Heuristica da la solucion optima: 13.207% de los casos
 Lejania promedio de la heuristica a la solucion optima: 1042.275%
 Desv. estandar de la lejania entre las soluciones: 1153.29
 Minima lejania entre bqlocal y exacta: 0
 Maxima lejania entre bqlocal y exacta: 4473.000%

Resultados del analisis (Metaheuristica GRASP):

Tiempo promedio microsegundos heuristica: 3637.888
 Tiempo promedio microsegundos exacto: 19571.283
 Heuristica da la solucion optima: 83.018% de los casos
 Lejania promedio de la heuristica a la solucion optima: 3.611%
 Desv. estandar de la lejania entre las soluciones: 12.1007
 Minima lejania entre bqlocal y exacta: 0
 Maxima lejania entre bqlocal y exacta: 73.300%

Distribucion de los resultados



Conclusion: Ahora analizaremos como se comportan los algoritmos heuristicos(goloso y busqueda local) respecto a los distintos conjuntos de tests y sus resultados.

Algoritmo Goloso para grafos de baja densidad:

Vemos que el algoritmo goloso tiene optimalidad cercana al 100 %, aqui dio 98.6 % y en la sección propia del algoritmo goloso dio 100 % cuando se trata de grafos con densidad lineal de aristas, en caso de no dar el optimo, tiene un 0.261 % con una desviacion estandar del 2 % de lejanía respecto a la solución óptima. Como contraparte, el tiempo de ejecución es sutilmente superior al del algoritmo exacto(con 4 podas), seguramente(como se puede apreciar en la sección de rendimiento del algoritmo exacto) para grafos con mayor cantidad de nodos, el tiempo de ejecución del algoritmo exacto se dispare. Consideramos que el algoritmo goloso es una muy buena opcion si se trata de aproximar soluciones optimas para grafos con densidad lineal con una cantidad de nodos considerablemente alta que produzca que el algoritmo exacto no sea viable.

Algoritmo Goloso para grafos de densidad intermedia:

Para instancias de grafos con densidades intermedias de aristas, podemos ver como baja levemente el nivel de optimalidad a 86 % de aciertos respecto a la solución exacta. Tambien se incrementaron a 2.59 % la lejanía promedio y a 9.7 % la desviacion estandar de la lejanía. Sin embargo, sigue siendo una muy buena heurística, porque en este caso si, puede apreciarse que el algoritmo exacto tarda un 250 % mas de tiempo en promedio para llegar a la solución. Teniendo en cuenta que la cantidad de nodos de los grafos generados va disminuyendo a medida que aumentamos la densidad(por cuestiones prácticas de realizar los tests en el hardware disponible), observamos que para densidades mas altas de aristas en los grafos testeados empieza a ser una muy buena alternativa la solución golosa, siempre y cuando el contexto de aplicación acepte los estimadores de lejanía al optimo.

Algoritmo Goloso para grafos de alta densidad:

Finalmente, para el caso mas complicado para el algoritmo exacto(recordemos que en la complejidad, el grado de los nodos es la cantidad de llamadas recursivas que realiza y si la densidad es alta el grado

promedio va a ser alto dada la distribución uniforme de las aristas sobre el grafo), la solución golosa disminuye muy levemente la efectividad con respecto al caso anterior, tiene un 83 % de aciertos a la solución óptima, asimismo aumentan los estimadores de lejanía, en promedio a un 4.99 % con una desviación de 18.17 %. Respecto al tiempo de ejecución, el algoritmo exacto tarda en promedio 4.28 veces mas de tiempo para obtener la solución exacta. Para ser el caso mas difícil de los casos de prueba que consideramos, pensamos que los resultados son muy buenos para las aplicaciones prácticas.

Algoritmo de Búsqueda local para grafos de baja densidad:

Respecto a la heurística de búsqueda local, observamos un 30 % de efectividad, con una lejanía promedio de 87.62 % y una desv. estandar en la lejanía al óptimo de 118.5 %, consideramos que 30 % es un número muy bajo y junto a los indicadores de lejanía, si nos interesa realmente una solución cercana a la óptima no elegiríamos esta heurística (al menos no con dijkstra sobre w_1 como solución inicial. Ver Experimentación de la solución inicial en la sección búsqueda local). Como contraparte, toma aproximadamente la mitad de tiempo que el algoritmo exacto (recordemos que la medición es por iteración y para grafos lineales hay 1 iteración promedio.), no consideramos que sea un buen tradeoff, la mitad del tiempo de ejecución para esta calidad de soluciones.

Algoritmo de Búsqueda local para grafos de densidad intermedia:

A medida que vamos aumentando la densidad de los grafos, la búsqueda local debería poder mejorar mas la solución, de hecho cuando analizamos la variación de la solución a medida que avanzan las iteraciones, el mejor caso era donde la densidad era máxima, lo que atribuimos a una vecindad de mayor cardinal, pero cuando comparamos optimalidad promedio de la heurística, nos da un número bajísimo, 10 % de aciertos a la solución óptima, con un promedio y dispersión de lejanía enormes. Atribuimos este bajo rendimiento a la solución inicial provista realizando dijkstra sobre w_1 , mas adelante cuando analicemos GRASP, para ciertos valores del parámetro beta (bajos, para que se acerque la solución randomizada a la heurística golosa pura), los experimentos representarán una búsqueda local alimentada inicialmente por un algoritmo de mayor optimalidad. Dado que la cantidad promedio de iteraciones para densidades intermedias nos dio cercano a 4 iteraciones, el tiempo de ejecución (por iteración) multiplicado por 4, nos da como resultado que el algoritmo de búsqueda local tarda un tercio del tiempo que el algoritmo exacto, nuevamente, no consideramos que sea un tradeoff aceptable dada la calidad de las soluciones.

Algoritmo de Búsqueda local para grafos de alta densidad:

Este caso es casi idéntico al anterior con respecto a la optimalidad de las soluciones, tal vez un poco peor, pero despreciable, lo único que varió considerablemente fue el tiempo de ejecución del algoritmo exacto a 19571 microsegundos y la cantidad de iteraciones promedio de la búsqueda local a 5 iteraciones promedio con un tiempo por iteración de 550 microsegundos. Veamos entonces que el algoritmo de búsqueda local se ejecuta aproximadamente 7 veces mas rápido que el algoritmo exacto. Las conclusiones son las mismas que en el caso de densidad intermedia, no consideramos que sea un buen tradeoff dado el bajísimo número de aciertos a la solución óptima y la enorme lejanía al óptimo.

Como conclusión de búsqueda local consideramos que depende mucho de la solución inicial la calidad final de las soluciones obtenidas (nuevamente, ver sección de variación de soluciones iniciales en sección búsqueda local). Utilizaríamos un esquema de búsqueda local para refinar soluciones relativamente buenas obtenidas con algún otro método inicial.

Algoritmo de Metaheurística GRASP para grafos de baja densidad:

Notamos como primera observación que la metaheurística toma mas tiempo que el algoritmo exacto, y al tener un 57 % de porcentaje de aciertos con respecto a la solución óptima, descartamos que sea una buena heurística para este tipo de grafos, sin importar la lejanía ni la dispersión de la lejanía, en menos tiempo podemos obtener la solución exacta.

Algoritmo de Metaheurística GRASP para grafos de densidad intermedia:

Observamos que la heurística no es muy efectiva, con una alta lejanía y dispersión de lejanía respecto

a la solución óptima, a pesar de que el tiempo que toma sea la mitad que el algoritmo exacto, no lo utilizaríamos como heurística para este tipo de grafos.

Algoritmo de Metaheurística GRASP para grafos de alta densidad:

Notemos que toma menos tiempo que el algoritmo goloso (y mucho menos tiempo que el algoritmo exacto), nos provee el mismo alto porcentaje de efectividad respecto a la solución óptima que el algoritmo goloso, y aun mejor, la lejanía promedio y dispersión de esta, se disminuyeron respecto a la solución. Para este tipo de grafos, con alta densidad, encontramos que con este parámetro beta, ocurrían este tipo de mejoras, si bien esto es el resultado de una muestra de toda la familia de grafos de alta densidad, podríamos considerar aplicar esta heurística aun mas que la heurística golosa para resolver el problema de manera aproximada.

6.4.5. Gráficos de distribución de las soluciones

Distribución de soluciones para grafos de alta densidad:

Podemos observar que las soluciones de búsqueda local siempre están lejos del óptimo indicado por un + amarillo en el eje Y. Con respecto a GRASP, algoritmo goloso y algoritmo exacto se encuentran casi superpuestos, salvo casos particulares.

Distribución de soluciones para grafos de densidad intermedia:

Se observa que los valores de las soluciones provistos por búsqueda local están sutilmente mas cerca de los demás algoritmos, igual siguen estando relativamente lejos, respecto a las otras heurísticas (Golosa y GRASP) comienzan a verse su "despegue" de la solución óptima, mas que nada en la metaheurística GRASP.

Distribución de soluciones para grafos de baja densidad:

Finalmente aquí observamos que la solución de búsqueda local sigue "lejos" del óptimo. Asimismo, GRASP tampoco provee la solución óptima ni una muy cercana, como si lo hace la heurística golosa con su casi perfecto índice de aciertos respecto al algoritmo exacto.

6.5. Experimentación pura entre heurísticas

En esta sección experimentaremos para varios conjuntos aleatorios de grafos de diferentes densidades con las heurísticas implementadas, sin tener en cuenta la solución exacta, para poder realizar experimentos con una mayor cantidad de nodos y aristas en las instancias. Dados los resultados, analizaremos de las 3 heurísticas implementadas cual obtiene las soluciones con peso w_2 menor, y de esta forma, mas cercanas a la solución óptima (que siempre es menor o igual al mínimo de las 3 heurísticas, por definición de solución óptima).

Esto se realizará, para cada ejecución de las 3 heurísticas, incrementando un contador en la que corresponda si su peso w_2 es mínimo, al final tendremos, para cada heurística, cuantas veces dio la mínima solución, y la cantidad total de instancias testeadas, con estos datos podremos realizar los análisis estadísticos.

6.5.1. Grafos aleatorios de alta densidad de aristas

Parámetros del experimento:

- Parámetro Beta GRASP: $\frac{cantNodos}{128}$
- Cantidad de grafos analizados: 17
- Cantidad mínima de nodos: 10

- Cantidad maxima de nodos: 340
- Rango peso w_1 : [0..250]
- Rango peso w_2 : [0..400]
- Cota de w_1 : 200
- Cantidad minima de aristas: $\frac{n*(n-1)}{3}$
- Cantidad maxima de aristas: $\frac{n*(n-1)}{2}$

Resultados del analisis:

Porcentaje de instancias donde da el minimo w2:
 Golosa: 94.1 %
 Busqueda local: 17.6 %
 Grasp: 41.1 %

6.5.2. Grafos aleatorios de densidad intermedia de aristas

Parametros del experimento:

- Parametro Beta GRASP: $\frac{cantNodos}{128}$
- Cantidad de grafos analizados: 226
- Cantidad minima de nodos: 100
- Cantidad maxima de nodos: 600
- Rango peso w_1 : [0..250]
- Rango peso w_2 : [0..400]
- Cota de w_1 : 200
- Cantidad minima de aristas: $n * \sqrt{n}$
- Cantidad maxima de aristas: $5 * n * \sqrt{n}$

Resultados del analisis:

Porcentaje de instancias donde da el minimo w2:
 Golosa: 96.9 %
 Busqueda local: 6.6 %
 Grasp: 33.1 %

6.5.3. Grafos aleatorios de densidad baja de aristas

Parametros del experimento:

- Parametro Beta GRASP: $\frac{cantNodos}{128}$
- Cantidad de grafos analizados: 7
- Cantidad minima de nodos: 1500
- Cantidad maxima de nodos: 3000
- Rango peso w_1 : [0..250]

- Rango peso w_2 : $[0..400]$
- Cota de w_1 : 200
- Cantidad minima de aristas: $15 * n$
- Cantidad maxima de aristas: $20 * n$

Resultados del analisis:

<p>Porcentaje de instancias donde da el minimo w2:</p> <p>Golosa: 100 %</p> <p>Busqueda local: 0 %</p> <p>Grasp: 0 %</p>
--

Vemos que la heurística golosa da mas veces la solución mínima a medida que va a disminuyendo la densidad de los grafos, lo que coincide con el decrecimiento del porcentaje de aciertos respecto al optimo a medida que aumenta la densidad en la sección anterior. Con respecto a la busqueda local, como vimos en la seccion de busqueda local, la heurística funciona mejor con alta densidad de grafos dado que esta relacionada la densidad del grafo con el cardinal de la vecindad. La metaheurística GRASP disminuye su rendimiento a medida que decrece la densidad de los grafos, suponemos que se debe a que la heurística golosa randomizada, a lo sumo da tan bien como la golosa pura, y la busqueda local sobre golosa no mejora la solucion, finalmente la heurística golosa obtiene una mejor solucion.

7. Conclusion

En este trabajo, al desarrollar varios algoritmos que solucionan, de forma exacta o aproximada un problema dado, pudimos observar la performance, tanto temporal como de optimalidad de varias tecnicas para el desarrollo de heurísticas, analizar varios aspectos de las mismas, como por ejemplo que familias de grafos son buenas o malas para cada una, deducir, variando la densidad de las instancias de prueba, para que familia convenia aplicar cual u otra heurística, analizar las variaciones absolutas y relativas de la función target a medida que avanzan las iteraciones de ciertas heurísticas, en busqueda local por ejemplo: probar con distintas combinaciones de vecindad y solucion inicial.

Finalmente, en la sección de experimentacion general se explican la generacion aleatoria de grafos, los

scripts utilizados para realizar los experimentos, el modelo de puntajes porcentuales y estimadores que utilizamos para realizar las comparaciones de optimalidad, los graficos realizados y sus referencias, y para 3 tipos distintos de densidades de grafos generados aleatoriamente, se realizaron 2 experimentos y se escribieron conclusiones acerca de sus resultados. Estos dos experimentos a grandes rasgos evalúan, el primero, el porcentaje de veces que las heurísticas dan la solucion optima y que tan lejos de ella estan junto con la comparacion del tiempo de ejecucion entre el algoritmo de la heurística y el algoritmo de solucion exacta, y el segundo, entre las 3 heurísticas, para los valores máximos de nodos y densidad que pudimos evaluar con el hardware con el que contamos, con que porcentaje cada heurística da el mínimo valor de la solucion(por ende el mas cercano al óptimo), la que tenga el porcentaje mas alto, será la que mas se acerque a una solucion exacta. Si tuvieramos que elegir una sola heurística para

solucionar el problema de forma aproximada, elegiríamos la heurística golosa, que demostró tener la mayor performance y tiempos de ejecución aceptables(salvo algunos casos, ver seccion de exp. general de golosa, grafos de baja densidad.). La heurística de búsqueda local alimentada con dijkstra sobre w_1 provee soluciones lejanas al optimo, la utilizaríamos solo para refinar soluciones factibles de buena calidad ya obtenidas, no como heurística para solucionar el problema. La metaheurística GRASP, dado lo que dijimos antes, deberia ser la mejor, dado que la solución golosa es muy buena y la busqueda local podría refinarla, pero no es así, vimos que variando el parametro beta de la metaheurística tanto en RCL por valor o cantidad, al disminuir este parametro la golosa randomizada se acerca a la golosa pura y como vimos en la seccion de busqueda local(greedy como solucion inicial), en promedio nunca hay ninguna mejora a nivel local para las soluciones golosas.Al aumentar el parametro, la solucion inicial baja el porcentaje de aciertos con el optimo, asi que mantuvimos el parametro a niveles bajos, y no obtuvimos mejora significativa, en el mejor de los casos, GRASP da el mismo porcentaje de aciertos respecto al optimo que la heurística golosa.

8. Compilacion

Para compilar el trabajo practico, basta entrar a la carpeta `/codigo/heuristicas/` y ejecutar `make clean all` para que se recompilen todos los algoritmos del tp(generator, exacto, golosa, bqlocal y grasp). Los binarios generados aceptan la entrada en el formato especificado en el enunciado del tp por entrada estandar del sistema e imprimen la salida tambien en el formato especificado por la salida estandar del sistema.

9. Entregable

En el archivo comprimido adjunto se entregan los codigos fuente de las `heuristicas` en `/codigo/heuristicas/`, del `generador de grafos aleatorios fisher yates` en `/codigo/heuristicas/`, del algoritmo `exacto` en `/codigo/sol_exacta/` y el `informe en pdf` en la carpeta `/informe/`.

10. Reentrega: Informe de modificaciones

En esta sección se detallaran brevemente, esperamos no olvidarnos de ninguna, las modificaciones realizadas con respecto a la primera entrega, decidimos utilizar un sistema de tickets que nos permitio trackear cada modificacion que realizamos. A continuacion se listaran las modificaciones:

- **Exacto:** Complejidad: Explicar porque las marcas hacen que el algoritmo funcione
- **Exacto:** Cambiar Kn por acotar debidamente para grafos G genericos
- **Exacto:** Falta caso base $T(1)$ en analisis complejidad
- **Exacto:** Fix calculo de complejidad, ver pagina 9 de la devolucion.
- **Exacto:** Poda 1: Validacion de existencia de camino factible
- **Exacto:** Poda 2: Finalizar la busqueda si se va a realizar una llamada sobre una rama no factible
- **Exacto:** Poda 3: Verificacion de optimalidad de la solucion en construccion
- **Golosa:** Descripcion, Correctitud(siempre hay solucion si hay factible), PseudoCodigo y Complejidad Teorica
- **Golosa:** Familias de Grafos Malas para esta heuristica
- **Golosa:** Experimentaciones y graficos de performance
- **Golosa:** Experimentacion de optimalidad
- **Bqlocal:** Comparacion de mejora evolutiva de iteraciones segun diferentes soluciones iniciales
- **Bqlocal:** Experimentacion rendimiento
- **Bqlocal:** Medicion evolutiva de mejora en iteraciones de una instancia
- **Bqlocal:** Preprocesar listas de vecinos en comun
- **Bqlocal:** Combinar los 3 criterios de bqlocal en una vecindad unificada
- **Bqlocal:** Reescribir complejidad teorica
- **Bqlocal:** Pseudocodigo mas acorde a la combinacion de vecindades
- **Bqlocal:** Definicion bien las vecindades por separado
- **Bqlocal:** Explicar con palabras los 3 modos de vecindad especificados con simbolos
- **Bqlocal:** Nivel de optimalidad de las soluciones
- **Bqlocal:** Ejemplos de familias de grafos malas para esta heuristica
- **Bqlocal:** Comparacion de rendimiento y optimalidad sobre distintas vecindades
- **Bqlocal:** Analisis de cantidad de iteraciones sobre greedy como sol inicial
- **Grasp:** Descripcion, planteo de la RCL, analisis del parametro
- **GRASP:** Complejidad y graficos de complejidad
- **GRASP:** Experimentacion de optimalidad
- **GRASP:** Experimentacion de variacion de parametros

- **Generador de grafos:** Explicacion de la generador aleatoria de grafos
- **Experimentacion:** Explicar scripts de performance y optimalidad
- **Experimentacion:** Realizar la experimentacion de comparacion entre todos los algoritmos
- **Experimentacion:** Realizar experimentacion entre las heurísticas unicamente para instancias mas grandes

A rasgos mas generales, las secciones de heurística golosa, heurística de busqueda local, metaheurística GRASP y experimentacion general fueron modificadas casi por completo o reescritas en su totalidad. La seccion de la vida real no tuvo cambios, la seccion del algoritmo exacto tuvo cambios menores que incluyen informacion acerca de las podas realizadas y correcciones pedidas en la devolucion.

11. Apéndice: Código fuente relevante

11.1. Generador aleatorio de grafos Fisher Yates

```
1 int main(int argc, char** argv){
2     int cant_nodos = 0;
3     int cant_aristas = 0;
4     int pesomin_w1 = 0;
5     int pesomax_w1 = 0;
6     int pesomin_w2 = 0;
7     int pesomax_w2 = 0;
8     int limit_w1 = 0;
9
10    if(argc != 8){
11        cout << "Use: _bin_<cant_nodos>_<cant_aristas>_<pesomin_w1>_<pesomax_w1>_<pesomin_w2>_<
12            pesomax_w2>_<limit_w1>_" << endl;
13        return 1;
14    }else{
15        cant_nodos = atoi(argv[1]);
16        cant_aristas = atoi(argv[2]);
17        pesomin_w1 = atoi(argv[3]);
18        pesomax_w1 = atoi(argv[4]);
19        pesomin_w2 = atoi(argv[5]);
20        pesomax_w2 = atoi(argv[6]);
21        limit_w1 = atoi(argv[7]);
22    }
23    int aristas_maximas = cant_nodos*(cant_nodos-1)/2;
24
25    //validacion maximas aristas
26    if(cant_aristas > aristas_maximas){
27        cant_aristas = aristas_maximas;
28    }
29
30    //inicializo en <(0, 1), ..., (0, n-1), (1, 2), ..., (1, n-1), ..., (n-2, n-1)>
31    vector<pair<int, int>> aristas;
32    for(int i=0;i<cant_nodos;i++){
33        for(int j=i+1;j<cant_nodos;j++){
34            aristas.push_back(make_pair(i, j));
35        }
36    }
37
38    //ahora shufleo uniformemente con fisher yates
39    //http://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle
40
41    random_device rd;
42    mt19937 gen(rd());
43
44    uniform_int_distribution<int> dis_srcdst(1, cant_nodos);
45    int nodo_src = dis_srcdst(gen);
46    int nodo_dst = dis_srcdst(gen);
47
48    //header line: nodos aristas src dst limitw1
49    cout << cant_nodos << "_" << cant_aristas << "_" << nodo_src << "_" << nodo_dst << "_" <<
50        limit_w1 << endl;
51
52    int permutations_number = aristas_maximas; //C(cant_nodos, 2)
53    vector<pair<int, int>>::iterator begin_iter = aristas.begin();
54    for(int i=0;i<permutations_number;i++){
55        uniform_int_distribution<int> dis(i, permutations_number);
56        int x = dis(gen);
57        iter_swap(begin_iter + i, begin_iter + x);
58    }
59
60    //ahora agarro del vector randomizado de aristas, las primeras cant_aristas y las imprimo
61    //con pesos aleatorios entre
62    aristas.resize(cant_aristas);
63
64    uniform_int_distribution<int> dis_w1(pesomin_w1, pesomax_w1);
65    uniform_int_distribution<int> dis_w2(pesomin_w2, pesomax_w2);
66    for(auto arista : aristas){
67        //notemos que la salida es entre 1..n!!
68        int partida = arista.first + 1;
69        int destino = arista.second + 1;
70        int peso_w1 = dis_w1(gen);
71        int peso_w2 = dis_w2(gen);
72        cout << partida << "_" << destino << "_" << peso_w1 << "_" << peso_w2 << endl;
```

```

70     }
71     cout << "0";
72 }

```

11.2. Script de minimalidad

No se pudo incluir el script por problemas de formato con latex. leerlo del archivo de codigo (./codigo/scripting_min.sh)

11.3. Script de optimalidad

No se pudo incluir el script por problemas de formato con latex. leerlo del archivo de codigo (./codigo/scripting_opt.sh)

11.4. Script de performance

No se pudo incluir el script por problemas de formato con latex. leerlo del archivo de codigo (./codigo/scripting_per.sh)

11.5. Algoritmo exacto para la resolucion de CACM

```

1  #include <stdio.h>
2  #include <iostream>
3  #include <string.h>
4  #include <math.h>
5  #include "grafo.h"
6  #include "parser.h"
7  #include "timing.h"
8
9  #define CANT.ITERES_MEDICION 1
10
11 using namespace std;
12
13 typedef struct peso{
14     int w1;
15     int w2;
16     int otra_cosa;
17 } peso;
18
19 typedef struct Solucion{
20     float W1;
21     float W2;
22     int k;
23     int *v;
24 } Solucion;
25
26 Solucion solucion_mejor;
27
28 void setW1(void *p, float w)
29 {
30     ((peso *)p)->w1 = w;
31 }
32 void setW2(void *p, float w)
33 {
34     ((peso *)p)->w2 = w;
35 }
36
37 bool resolver(Grafo<peso> &g, int u, int w, int K, float *distancia_w1, float *distancia_w2,
38               Solucion *solucion)
39 {
40     Solucion solucion_nueva;
41     int cantidad_aristas, *adyacentes, i;
42     peso *p;
43     float w2, w1;
44     bool solucionado = false;
45
46     solucion_nueva.W1 = 0;
47     solucion_nueva.W2 = 0;
48     solucion_nueva.k = 0;
49
50     if(u == w){
51         if(solucion_mejor.W2 < 0 || solucion_mejor.W2 > solucion->W2){

```



```

51         solucion_mejor.W1 = solucion->W1;
52         solucion_mejor.W2 = solucion->W2;
53         solucion_mejor.k = solucion->k;
54         memcpy(solucion_mejor.v, solucion->v, 2 * g.CantidadAristas() * sizeof(int));
55         return true;
56     }
57     return false;
58 }
59 g.MarcarNodo(u, true);
60 adyacentes = g.Adyacentes(u, &cantidad_aristas);
61 for(i = 0; i < cantidad_aristas; i++){
62     if(g.VerMarcaDeNodo(adyacentes[i]) == false){
63         p = g.Peso(u, adyacentes[i]);
64         w1 = p->w1;
65         w2 = p->w2;
66         if(distancia_w1[adyacentes[i]] + w1 <= K && (solucion_mejor.k == 0 || solucion->W2
67             + w2 + distancia_w2[adyacentes[i]] < solucion_mejor.W2)){
68             g.QuitarArista(u, adyacentes[i]);
69             solucion_nueva.W1 = solucion->W1 + w1;
70             solucion_nueva.W2 = solucion->W2 + w2;
71             solucion_nueva.v = solucion->v;
72             solucion_nueva.v[solucion->k] = adyacentes[i];
73             solucion_nueva.k = solucion->k + 1;
74             if(resolver(g, adyacentes[i], w, K - w1, distancia_w1, distancia_w2, &
75                 solucion_nueva)){
76                 solucionado = true;
77             }
78         }
79     }
80     g.MarcarNodo(u, false);
81     if(solucionado){
82         solucion->W1 = solucion_mejor.W1;
83         solucion->W2 = solucion_mejor.W2;
84         memcpy(solucion->v, solucion_mejor.v, solucion_mejor.k * sizeof(int));
85         solucion->k = solucion_mejor.k;
86     }
87     free(adyacentes);
88     return solucionado;
89 }
90
91 void dijkstra(Grafo<peso> *g, int nodo, float **_distancia_w1, float **_distancia_w2)
92 {
93     int cantidad_nodos, nodo_minimo, cantidad_adyacentes, *adyacentes, cantidad_vistos;
94     int i;
95     float *distancia_w1, *distancia_w2;
96     float peso_minimo;
97     bool *nodos_vistos;
98
99     if(g == NULL || _distancia_w1 == NULL || _distancia_w2 == NULL)
100         return;
101
102     cantidad_nodos = g->CantidadNodos();
103     if(nodo < 1 || nodo > cantidad_nodos)
104         return;
105
106     nodos_vistos = new bool[cantidad_nodos + 1];
107     *_distancia_w1 = new float[cantidad_nodos + 1];
108     *_distancia_w2 = new float[cantidad_nodos + 1];
109     distancia_w1 = *_distancia_w1;
110     distancia_w2 = *_distancia_w2;
111
112     for(i = 1; i <= cantidad_nodos; i++){
113         distancia_w1[i] = INFINITY;
114         distancia_w2[i] = INFINITY;
115         nodos_vistos[i] = false;
116     }
117     distancia_w1[nodo] = 0;
118     distancia_w2[nodo] = 0;
119     cantidad_vistos = 0;
120
121     while(cantidad_vistos < cantidad_nodos){
122         peso_minimo = INFINITY;
123         nodo_minimo = 1;
124         for(i = 1; i <= cantidad_nodos; i++){
125             if(!nodos_vistos[i] && distancia_w1[i] < peso_minimo){

```

```

126         nodo_minimo = i;
127         peso_minimo = distancia_w1[i];
128     }
129 }
130 nodos_vistos[nodo_minimo] = true;
131 cantidad_vistos++;
132 adyacentes = g->Adyacentes(nodo_minimo, &cantidad_adyacentes);
133 for(i = 0; i < cantidad_adyacentes; i++){
134     if(!nodos_vistos[adyacentes[i]] && distancia_w1[adyacentes[i]] > distancia_w1[
        nodo_minimo] + g->Peso(nodo_minimo, adyacentes[i])->w1){
135         distancia_w1[adyacentes[i]] = distancia_w1[nodo_minimo] + g->Peso(nodo_minimo,
            adyacentes[i])->w1;
136     }
137 }
138 free(adyacentes);
139 }
140
141 for(i = 1; i <= cantidad_nodos; i++){
142     nodos_vistos[i] = false;
143 }
144 cantidad_vistos = 0;
145
146 while(cantidad_vistos < cantidad_nodos){
147     peso_minimo = INFINITY;
148     nodo_minimo = 1;
149     for(i = 1; i <= cantidad_nodos; i++){
150         if(!nodos_vistos[i] && distancia_w2[i] < peso_minimo){
151             nodo_minimo = i;
152             peso_minimo = distancia_w2[i];
153         }
154     }
155     nodos_vistos[nodo_minimo] = true;
156     cantidad_vistos++;
157     adyacentes = g->Adyacentes(nodo_minimo, &cantidad_adyacentes);
158     for(i = 0; i < cantidad_adyacentes; i++){
159         if(!nodos_vistos[adyacentes[i]] && distancia_w2[adyacentes[i]] > distancia_w2[
            nodo_minimo] + g->Peso(nodo_minimo, adyacentes[i])->w2){
160             distancia_w2[adyacentes[i]] = distancia_w2[nodo_minimo] + g->Peso(nodo_minimo,
                adyacentes[i])->w2;
161         }
162     }
163     free(adyacentes);
164 }
165
166 delete [] nodos_vistos;
167
168 }
169
170 void comenzar(Grafo<peso> *grafo, int u, int v, int K, int nodos, int aristas)
171 {
172     int i;
173     Solucion solucion;
174     float *distancia_w1_v;
175     float *distancia_w2_v;
176
177     solucion_mejor.W1 = -1;
178     solucion_mejor.W2 = -1;
179     solucion_mejor.k = 0;
180     solucion_mejor.v = (int *)calloc(2 * grafo->CantidadAristas(), sizeof(int));
181     solucion.W1 = 0;
182     solucion.W2 = 0;
183     solucion.k = 1;
184     solucion.v = (int *)calloc(2 * aristas, sizeof(int));
185     solucion.v[0] = u;
186     dijkstra(grafo, v, &distancia_w1_v, &distancia_w2_v);
187     if(distancia_w1_v[u] > K){
188         printf("no");
189     }
190     else if(resolver(*grafo, u, v, K, distancia_w1_v, distancia_w2_v, &solucion)){
191         printf("%0f_%.0f_%.0f", solucion.W1, solucion.W2, solucion.k);
192         for(i = 0; i < solucion.k; i++){
193             printf("_%.0f", solucion.v[i]);
194         }
195         printf("\n");
196     }
197     else{
198         printf("no");

```

```

199     }
200     free(solucion.v);
201     free(solucion_mejor.v);
202     delete [] distancia_w1_v;
203     delete [] distancia_w2_v;
204
205 }
206
207 int main(int argc, char **argv)
208 {
209     int medir_tiempo = 0;
210     GrafoAdyacencia<peso> *grafo;
211     peso **pesos;
212     int u, v, K, nodos, aristas;
213
214     //if(argc != 1)
215     medir_tiempo = 1; //dejalo asi, lo necesito para los scripts, idem la cantidad de
        iteraciones(necesito que imprima la salida una sola vez). Firma: Silvio.
216     grafo = new GrafoAdyacencia<peso>(0);
217     while(Parsear<peso>(*grafo, stdin, &pesos, setW1, setW2, &u, &v, &K, &nodos, &aristas)){
218         if(medir_tiempo){
219             double promedio_medicion = 0.0;
220             MEDIR_TIEMPO_PROMEDIO(
221                 comenzar(grafo, u, v, K, nodos, aristas);
222                 , CANT_ITERS_MEDICION, &promedio_medicion);
223             //cerr << promedio << " " << nodos << " " << aristas << endl;
224             cerr << nodos << " " << aristas << " " << CANT_ITERS_MEDICION << " " <<
                promedio_medicion;
225         }
226         else{
227             comenzar(grafo, u, v, K, nodos, aristas);
228         }
229         int i;
230         for(i = 0; i < aristas; i++){
231             delete pesos[i];
232         }
233         free(pesos);
234         delete grafo;
235         grafo = new GrafoAdyacencia<peso>(0);
236     }
237     delete grafo;
238     return 0;
239 }

```

11.6. Heurísticas

```

1  // #define DEBUG_MESSAGES_ON
2  // #define CYCLE_PREVENT_MESSAGE_ON
3  // #define VECINOS_COMUNES_LAZY
4
5  using namespace std;
6
7  typedef int nodo_t;
8  typedef int longitud_t;
9  typedef double distancia_t;
10 typedef double costo_t;
11 typedef enum tipo_costo_t {COSTO_W1, COSTO_W2} tipo_costo_t;
12 typedef enum tipo_ejecucion_golosa_t {RCL_DETERMINISTICO, RCL_POR_VALOR, RCL_POR_CANTIDAD}
    tipo_ejecucion_golosa_t;
13 typedef enum tipo_ejecucion_bqlocal_t {BQL_SUBDIVIDIR_PARES, BQL_CONTRAER_TRIPLAS_A_PARES,
    BQL_MEJORAR_CONEXION_TRIPLAS, BQL_COMBINAR} tipo_ejecucion_bqlocal_t;
14 typedef enum criterio_terminacion_grasp_t {CRT_K_ITERS_SIN_MEJORA, CRT_K_ITERS_LIMIT_REACHED}
    criterio_terminacion_grasp_t;
15 typedef enum formato_entrada_t {FORMATO_0_N_OPEN, FORMATO_1_N_CLOSED} formato_entrada_t;
16 typedef formato_entrada_t formato_salida_t;
17
18 const costo_t costo_infinito = numeric_limits<double>::infinity();
19 const distancia_t distancia_infinita = numeric_limits<double>::infinity();
20 const costo_t costo_nulo = 0;
21 const nodo_t predecesor_nulo = -1;
22
23 class Arista{
24 private:
25     bool presente;
26     costo_t costo_w1;
27     costo_t costo_w2;
28 public:

```

```

29     Arista();
30     Arista(bool pres, costo_t w1, costo_t w2);
31     ~Arista();
32
33     bool esta_presente();
34     void desmarcar_presente();
35     void marcar_presente(costo_t w1, costo_t w2);
36     costo_t obtener_costo_w1() const;
37     costo_t obtener_costo_w2() const;
38
39     bool operator==(const Arista &other) const
40     {
41         return (other.presente == this->presente &&
42                 other.costo_w1 == this->costo_w1 &&
43                 other.costo_w2 == this->costo_w2);
44     }
45 };
46
47 typedef vector<vector<Arista>> matriz_adyacencia_t;
48 typedef list<pair<nodo_t, Arista>> lista_adyacentes;
49 typedef vector<lista_adyacentes> lista_adyacencia_t;
50
51 class Vecino{
52 private:
53     nodo_t i;
54     nodo_t j;
55     nodo_t en_comun;
56     Arista desde_i_a_comun;
57     Arista desde_j_a_comun;
58 public:
59     Vecino(nodo_t i, nodo_t j, nodo_t comun, Arista desde_i, Arista desde_j);
60     Vecino();
61     ~Vecino();
62     nodo_t obtener_nodo_i();
63     nodo_t obtener_nodo_j();
64     nodo_t obtener_nodo_comun();
65     Arista obtener_arista_i_comun();
66     Arista obtener_arista_j_comun();
67 };
68
69 typedef vector<vector<vector<Vecino>>> vecinos_comunes_t;
70
71 class Camino{
72 private:
73     list<nodo_t> camino;
74     vector<bool> esta_en_camino;
75     costo_t costo_camino_w1;
76     costo_t costo_camino_w2;
77     matriz_adyacencia_t mat_adyacencia;
78 public:
79     //Camino();
80     Camino(matriz_adyacencia_t& mat_adyacencia);
81     ~Camino();
82
83     void agregar_nodo(nodo_t target);
84     void agregar_nodo_adelante(nodo_t target);
85     costo_t obtener_costo_total_w1_camino();
86     costo_t obtener_costo_total_w2_camino();
87     //pre: 0 <= i <= j < cantidad_nodos y que i,j sean adyacentes
88     costo_t obtener_costo_w1_entre_nodos(nodo_t i, nodo_t j);
89     //pre: 0 <= i <= j < cantidad_nodos y que i,j sean adyacentes
90     costo_t obtener_costo_w2_entre_nodos(nodo_t i, nodo_t j);
91
92     void imprimir_camino(ostream& out);
93
94     list<nodo_t>::iterator obtener_iterador_begin();
95     list<nodo_t>::iterator obtener_iterador_end();
96     list<nodo_t>::const_iterator obtener_iterador_const_begin();
97     list<nodo_t>::const_iterator obtener_iterador_const_end();
98     longuitud_t obtener_longuitud_camino();
99
100     //pre: at.obtener_nodo_i() y at.obtener_nodo_j() deben pertenecer al camino
101     //Se reemplazara la conexion directa entre i y j por i -> en_comun -> j indicado por el
102     //Vecino pasado
103     //por parametro. Devuelve true si se inserto, false sino.
104     bool insertar_nodo(Vecino& at);

```

```

105 //pre: at.obtener_nodo_i() y at.obtener_nodo_j() deben pertenecer al camino
106 //Se reemplazara la conexion i..loquesea..j por i -> en comun -> j indicado por el Vecino
    pasado
107 //por parametro. Devuelve true si se inserto, false sino.
108 bool mejorar_tripla(Vecino& at);
109
110 bool realizar_salto_entre_3_nodos(nodo_t punto_de_salto);
111 bool pertenece_a_camino(nodo_t target); //O(1)
112 };
113
114 class Grafo{
115 private:
116     //atributos
117     matriz_adyacencia_t mat_adyacencia;
118     vecinos_comunes_t vecinos_comunes;
119     lista_adyacencia_t lista_adyacencia;
120     int cantidad_nodos;
121     int cantidad_aristas;
122
123     //atributos propios del problema
124     nodo_t nodo_src;
125     nodo_t nodo_dst;
126     costo_t cota_w1;
127     Camino camino_obtenido;
128     bool sol_valida;
129     //dado el camino, podemos obtener los pesos de cada "salto" indexando en la matriz de
        adyacencia el costo de cada salto
130     //tanto para w1 como w2
131
132     //metodos auxiliares
133     bool mejorar_conexion_entre_pares(nodo_t nodo_i, nodo_t nodo_j, costo_t costo_ij_w1,
        costo_t costo_ij_w2, costo_t total_w1, costo_t total_w2,
134         Vecino& mejor_vecino);
135
136     bool mejorar_conexion_salteando(nodo_t nodo_i, nodo_t nodo_j, costo_t costo_ij_w1, costo_t
        costo_ij_w2, costo_t total_w1, costo_t total_w2,
137         Arista& mejor_vecino);
138
139     void precalcular_adyacentes_en_comun(nodo_t i, nodo_t j);
140
141     //Grasp
142     set<pair<costo_t, nodo_t> >::iterator obtener_candidato_randomizado(tipo_ejecucion_golosa_t
        tipo_ejecucion, const set<pair<costo_t, nodo_t> > & cola, double parametro_beta);
143
144     int busqueda_local_entre_pares_insertando(Camino& solucion_actual, Vecino&
        conexion_ij_minima_w2);
145     int busqueda_local_entre_triplas_reemplazando_intermedio(Camino& solucion_actual, Vecino&
        conexion_ij_minima_w2);
146     int busqueda_local_entre_triplas_salteando(Camino& solucion_actual, list<nodo_t>::
        const_iterator& punto_de_salto_it);
147 public:
148     //constructor y destructor
149     Grafo(int cant_inicial_nodos);
150     ~Grafo();
151
152     //Modificadores
153     void agregar_nodos(int cantidad_nodos);
154     void agregar_arista(nodo_t i, nodo_t j, costo_t w1, costo_t w2);
155
156     //Consultas
157     vector<Arista> obtener_vector_fila_vecinos(nodo_t target);
158     lista_adyacentes obtener_lista_vecinos(nodo_t target);
159     int obtener_cantidad_nodos();
160     int obtener_cantidad_aristas();
161     Arista obtener_arista(nodo_t i, nodo_t j);
162     //pre: 0 <= i <= j < cantidad_nodos y que i,j sean adyacentes
163     vector<Vecino> obtener_adyacentes_en_comun(nodo_t i, nodo_t j);
164     nodo_t obtener_nodo_origen();
165     nodo_t obtener_nodo_destino();
166     costo_t obtener_limite_w1();
167     costo_t obtener_costo_actual_w1_solucion();
168     costo_t obtener_costo_actual_w2_solucion();
169     Camino obtener_camino_solucion();
170     void establecer_camino_solucion(Camino c);
171
172     //Entrada - Salida
173     void imprimir_matriz_adyacencia(ostream& out);

```

```

174 void imprimir_lista_adyacencia(ostream& out);
175 void serialize(ostream& out, formato_salida_t formato);
176 bool unserialize(istream& in, formato_entrada_t formato);
177
178 //Algoritmos
179 //Realiza la busqueda local sobre una solucion inicial factible creada por dijkstra sobre
180 //COSTO_W1 entre src y dst
181 int busqueda_local(tipo_ejecucion_bqlocal_t tipo_ejecucion);
182 //Devuelve el camino minimo entre origen y destino(calcula el arbol, pero reconstruye solo
183 //el camino de origen a destino)
184 Camino dijkstra(nodo_t origen, nodo_t destino, tipo_costo_t target_a_minimizar);
185 //Aplica dijkstra desde nodo origen y calcula el arbol de caminos minimos por referencia a
186 //los vectores por parametro
187 void dijkstra(nodo_t origen, tipo_costo_t target_a_minimizar, vector<costo_t>& costo_minimo,
188 vector<nodo_t>& predecesor);
189 //Dado un nodo_t origen se calcula para cada nodo, la distancia minima en cantidad de
190 //aristas de peso constante 1 de cualquier nodo a origen
191 void breadth_first_search(nodo_t origen, vector<distancia_t>& distancias_en_aristas_a_origen);
192
193 //Golosa
194 Camino obtener_solucion_golosa();
195 Camino obtener_solucion_golosa_randomizada(tipo_ejecucion_golosa_t tipo_ejecucion, double
196 parametro_beta);
197
198 //Metodos utilitarios
199 static list<Grafo> parsear_varias_instancias(formato_entrada_t formato);
200 void establecer_se_encontro_solucion(bool se_encontro);
201 bool hay_solucion();
202 Camino obtener_camino_vacio();
203 };
204 //
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236

```

```

200
201
202 bool Camino::realizar_salto_entre_3_nodos(nodo_t nodo_target){
203     list<nodo_t>::iterator runner_it = this->camino.begin();
204     list<nodo_t>::iterator final_it = this->camino.end();
205     while(runner_it != final_it){
206         if(*runner_it == nodo_target){
207             break;
208         }
209         ++runner_it;
210     }
211
212     if(runner_it == final_it){
213         cerr << "Camino::realizar_salto_entre_3_nodos._No_se_encontro_el_nodo_target_pasado_por
214             _parametro_en_el_camino" << endl;
215         cerr << "Nodo_target_(" << nodo_target << ") " << endl;
216         cerr << "Camino: ";
217         this->imprimir_camino(cerr);
218         return false;
219     }
220
221     //aca vale *runner_it == nodo_target
222
223     nodo_t nodo_i = *runner_it;
224
225     runner_it++;
226     list<nodo_t>::iterator deletion_target = runner_it;
227     nodo_t nodo_intermedio_viejo = *runner_it;
228
229     runner_it++;
230     nodo_t nodo_j = *runner_it;
231
232     costo_t costo_i_intermedio_w1 = obtener_costo_w1_entre_nodos(nodo_i, nodo_intermedio_viejo);
233     ;
234     costo_t costo_i_intermedio_w2 = obtener_costo_w2_entre_nodos(nodo_i, nodo_intermedio_viejo);
235     ;
236
237     costo_t costo_intermedio_j_w1 = obtener_costo_w1_entre_nodos(nodo_intermedio_viejo, nodo_j);
238     ;
239     costo_t costo_intermedio_j_w2 = obtener_costo_w2_entre_nodos(nodo_intermedio_viejo, nodo_j);
240     ;
241

```

```

237     costo_t costo_i_j_w1 = obtener_costo_w1_entre_nodos(nodo_i, nodo_j);
238     costo_t costo_i_j_w2 = obtener_costo_w2_entre_nodos(nodo_i, nodo_j);
239
240     //elimino el nodo intermedio entre i y j
241     this->camino.erase(deletion_target);
242     this->esta_en_camino[nodo_intermedio_viejo] = false;
243
244     //actualizo los costos del camino
245     this->costo_camino_w1 = (this->costo_camino_w1 - (costo_i_intermedio_w1 +
        costo_intermedio_j_w1) + costo_i_j_w1);
246     this->costo_camino_w2 = (this->costo_camino_w2 - (costo_i_intermedio_w2 +
        costo_intermedio_j_w2) + costo_i_j_w2);
247
248     return true;
249 }
250
251 bool Camino::pertenece_a_camino(nodo_t target){
252     return this->esta_en_camino[target];
253 }
254
255 //pre: at.obtener_nodo_i() y at.obtener_nodo_j() deben pertenecer al camino
256 //Se reemplazara la conexion i..loquesea..j por i -> en comun -> j indicado por el Vecino pasado
257 //por parametro. Devuelve true si se inserto, false sino.
258 bool Camino::mejorar_tripla(Vecino& at){
259     nodo_t nodo_target = at.obtener_nodo_i();
260     nodo_t nodo_sig_target = at.obtener_nodo_j();
261     nodo_t nodo_intermedio = at.obtener_nodo_comun();
262     costo_t i_comun_w1 = at.obtener_arista_i_comun().obtener_costo_w1();
263     costo_t i_comun_w2 = at.obtener_arista_i_comun().obtener_costo_w2();
264     costo_t j_comun_w1 = at.obtener_arista_j_comun().obtener_costo_w1();
265     costo_t j_comun_w2 = at.obtener_arista_j_comun().obtener_costo_w2();
266     nodo_t intermedio_viejo = -1;
267
268     list<nodo_t>::iterator insertion_target = this->camino.begin();
269     list<nodo_t>::iterator final_it = this->camino.end();
270     while(insertion_target != final_it){
271         if(*insertion_target == nodo_target){
272             insertion_target++;
273             intermedio_viejo = *insertion_target;
274             break;
275         }
276         ++insertion_target;
277     }
278
279     if(insertion_target == final_it){
280         cerr << "Camino::mejorar_tripla..No se encontro el nodo_target pasado por parametro en el camino" << endl;
281         return false;
282     }
283
284     //aca vale que el iterator it apunta a target + 1
285
286     //elimino el intermedio viejo
287     nodo_t nodo_intermedio_viejo = *insertion_target;
288     insertion_target = this->camino.erase(insertion_target);
289     this->esta_en_camino[nodo_intermedio_viejo] = false;
290
291     //inserto el nodo en el medio
292     this->camino.insert(insertion_target, nodo_intermedio);
293     this->esta_en_camino[nodo_intermedio] = true;
294
295     //actualizo los costos
296     //resto el costo entre i y j y sumo las 2 aristas nuevas
297
298     costo_t costo_ij_w1 = obtener_costo_w1_entre_nodos(nodo_target, intermedio_viejo);
299     costo_ij_w1 += obtener_costo_w1_entre_nodos(intermedio_viejo, nodo_sig_target);
300
301     costo_t costo_ij_w2 = obtener_costo_w2_entre_nodos(nodo_target, intermedio_viejo);
302     costo_ij_w2 += obtener_costo_w2_entre_nodos(intermedio_viejo, nodo_sig_target);
303
304     costo_t costo_i_comun_j_w1 = i_comun_w1 + j_comun_w1;
305     costo_t costo_i_comun_j_w2 = i_comun_w2 + j_comun_w2;
306
307     this->costo_camino_w1 = (this->obtener_costo_total_w1_camino() - costo_ij_w1 +
        costo_i_comun_j_w1);
308     this->costo_camino_w2 = (this->obtener_costo_total_w2_camino() - costo_ij_w2 +
        costo_i_comun_j_w2);

```

```

309     return true;
310 }
311 }
312
313 //pre: at.obtener_nodo_i() y at.obtener_nodo_j() deben pertenecer al camino
314 //Se reemplazara la conexion directa entre i y j por i -> comun -> j indicado por el Vecino
315 //por parametro
316 bool Camino::insertar_nodo(Vecino& at){
317     nodo_t nodo_target = at.obtener_nodo_i();
318     nodo_t nodo_sig_target = at.obtener_nodo_j();
319     nodo_t nodo_intermedio = at.obtener_nodo_comun();
320     costo_t i_comun_w1 = at.obtener_arista_i_comun().obtener_costo_w1();
321     costo_t i_comun_w2 = at.obtener_arista_i_comun().obtener_costo_w2();
322     costo_t j_comun_w1 = at.obtener_arista_j_comun().obtener_costo_w1();
323     costo_t j_comun_w2 = at.obtener_arista_j_comun().obtener_costo_w2();
324
325     list<nodo_t>::iterator insertion_target = this->camino.begin();
326     list<nodo_t>::iterator final_it = this->camino.end();
327     while(insertion_target != final_it){
328         if(*insertion_target == nodo_target){
329             insertion_target++;
330             break;
331         }
332         ++insertion_target;
333     }
334
335     if(insertion_target == final_it){
336         cerr << "Camino::insertar_nodo._No se encontro el nodo target pasado por parametro en el camino" << endl;
337         return false;
338     }
339
340     //aca vale que el iterator it apunta a target + 1
341
342     //inserto el nodo en el medio
343     this->camino.insert(insertion_target, nodo_intermedio);
344     this->esta_en_camino[nodo_intermedio] = true;
345
346     //actualizo los costos
347     //resto la arista entre i y j y sumo las 2 aristas nuevas
348     costo_t costo_ij_w1 = obtener_costo_w1_entre_nodos(nodo_target, nodo_sig_target);
349     costo_t costo_ij_w2 = obtener_costo_w2_entre_nodos(nodo_target, nodo_sig_target);
350
351     costo_t costo_i_comun_j_w1 = i_comun_w1 + j_comun_w1;
352     costo_t costo_i_comun_j_w2 = i_comun_w2 + j_comun_w2;
353
354     this->costo_camino_w1 = (this->obtener_costo_total_w1_camino() - costo_ij_w1 +
355                             costo_i_comun_j_w1);
356     this->costo_camino_w2 = (this->obtener_costo_total_w2_camino() - costo_ij_w2 +
357                             costo_i_comun_j_w2);
358     return true;
359 }
360
361 // ----- Vecino -----
362
363 Vecino::Vecino(nodo_t i, nodo_t j, nodo_t comun, Arista desde_i, Arista desde_j){
364     this->i = i;
365     this->j = j;
366     this->en_comun = comun;
367     this->desde_i_a_comun = desde_i;
368     this->desde_j_a_comun = desde_j;
369 }
370
371 Vecino::Vecino(){
372     this->i = 0;
373     this->j = 0;
374     this->en_comun = 0;
375     //aristas constructor por defecto
376 }
377
378 Vecino::~Vecino(){
379 }
380
381 nodo_t Vecino::obtener_nodo_i(){
382     return this->i;
383 }

```



```

382 nodo_t Vecino::obtener_nodo_j(){
383     return this->j;
384 }
385
386 nodo_t Vecino::obtener_nodo_comun(){
387     return this->en_comun;
388 }
389
390 Arista Vecino::obtener_arista_i_comun(){
391     return this->desde_i_a_comun;
392 }
393
394 Arista Vecino::obtener_arista_j_comun(){
395     return this->desde_j_a_comun;
396 }
397
398 // ----- Grafo -----
399
400 Camino Grafo::obtener_camino_vacio(){
401     Camino c(this->mat_adyacencia);
402     return c;
403 }
404
405 Grafo::Grafo(int cant_inicial_nodos) : camino_obtenido(mat_adyacencia) {
406     this->cantidad_nodos=cant_inicial_nodos;
407     this->cantidad_aristas = 0;
408     nodo_src = 0;
409     nodo_dst = 0;
410     cota_wl = 0;
411     sol_valida = false;
412
413     //inicializo matriz de adyacencia
414     vector<Arista> vec_fila(cantidad_nodos, Arista(false, 0, 0));
415     this->mat_adyacencia.resize(cantidad_nodos, vec_fila);
416
417     //inicializo lista de adyacencia
418     this->lista_adyacencia.resize(cantidad_nodos);
419
420     //inicializo matriz de vecinos comunes
421     vector<vector<Vecino> > vec_fila_vecinos(cantidad_nodos);
422     this->vecinos_comunes.resize(cantidad_nodos, vec_fila_vecinos);
423 }
424
425 Grafo::~Grafo() {
426
427 }
428
429 //modificadores
430 void Grafo::agregar_nodos(int cantidad_nodos_nuevos){
431     //agrego cantidad_nodos_nuevos columnas en todas las filas existentes
432     for(int i=0; i < this->cantidad_nodos; i++){
433         this->mat_adyacencia[i].resize(this->cantidad_nodos + cantidad_nodos_nuevos, Arista(
434             false, 0, 0));
435     }
436
437     //agrego cantidad_nodos_nuevos filas al final de la matriz de adyacencia
438     vector<Arista> vec_fila(this->cantidad_nodos + cantidad_nodos_nuevos, Arista(false, 0, 0));
439     this->mat_adyacencia.resize(this->cantidad_nodos + cantidad_nodos_nuevos, vec_fila);
440
441     //redimensiono lista adyacencias
442     this->lista_adyacencia.resize(cantidad_nodos + cantidad_nodos_nuevos);
443
444     //redimensiono matriz de vecinos comunes
445     vector<vector<Vecino> > vec_fila_vecinos(this->cantidad_nodos + cantidad_nodos_nuevos);
446     this->vecinos_comunes.resize(this->cantidad_nodos + cantidad_nodos_nuevos, vec_fila_vecinos);
447
448     //actualizo cantidad_nodos total del grafo
449     this->cantidad_nodos+=cantidad_nodos_nuevos;
450 }
451
452 void Grafo::agregar_arista(nodo_t i, nodo_t j, costo_t w1, costo_t w2){
453     Arista arista = obtener_arista(i, j);
454     //marco doble, no es un digrafo
455     this->mat_adyacencia[i][j].marcar_presente(w1, w2);
456     this->mat_adyacencia[j][i].marcar_presente(w1, w2);

```

```

457     this->lista_adyacencia[i].push_back(make_pair(j, obtener_arista(i, j)));
458     this->lista_adyacencia[j].push_back(make_pair(i, obtener_arista(i, j)));
459
460     if(!arista.esta_presente()){
461         this->cantidad_aristas++;
462     }
463 }
464
465 //consultas
466 Arista Grafo::obtener_arista(nodo_t i, nodo_t j){
467     return this->mat_adyacencia[i][j];
468 }
469
470 void Grafo::imprimir_matriz_adyacencia(ostream& out){
471     out << "Cantidad_nodos:" << this->cantidad_nodos << endl;
472     out << "Cantidad_aristas:" << this->cantidad_aristas << endl;
473     out << "Matriz_adyacencia:" << endl;
474     for (int i = 0; i < this->cantidad_nodos; i++){
475         for (int j = 0; j < this->cantidad_nodos; j++){
476             out << "|_" << this->mat_adyacencia[i][j].esta_presente() << ")-|";
477         }
478         out << endl;
479     }
480     out << endl;
481     out << endl;
482 }
483
484 void Grafo::imprimir_lista_adyacencia(ostream& out){
485     out << "Cantidad_nodos:" << this->cantidad_nodos << endl;
486     out << "Cantidad_aristas:" << this->cantidad_aristas << endl;
487     out << "Lista_adyacencia:" << endl;
488     for (int i = 0; i < this->cantidad_nodos; i++){
489         out << "Vecinos_de[" << i << "]:_";
490         lista_adyacentes::iterator adyacentes_i_it = this->lista_adyacencia[i].begin();
491         lista_adyacentes::iterator final_it = this->lista_adyacencia[i].end();
492         while(adyacentes_i_it != final_it){
493             out << "(" << adyacentes_i_it->first << ")_--(" << this->mat_adyacencia[i][
                adyacentes_i_it->first].obtener_costo_w1() << ",_" << this->mat_adyacencia[i][
                adyacentes_i_it->first].obtener_costo_w2() << ")->_";
            adyacentes_i_it++;
494         }
495         out << "Nil" << endl;
496     }
497     out << endl;
498     out << endl;
499 }
500 }
501
502 costo_t Grafo::obtener_limite_w1(){
503     return this->cota_w1;
504 }
505
506 void Grafo::establecer_camino_solucion(Camino c){
507     this->camino_obtenido = c;
508 }
509
510 Camino Grafo::obtener_camino_solucion(){
511     return this->camino_obtenido;
512 }
513
514 int Grafo::obtener_cantidad_nodos(){
515     return this->cantidad_nodos;
516 }
517
518 int Grafo::obtener_cantidad_aristas(){
519     return this->cantidad_aristas;
520 }
521
522 void Grafo::serialize(ostream& out, formato_salida_t formato){
523     if(this->sol_valida){
524         out << this->camino_obtenido.obtener_costo_total_w1_camino() << " ";
525         out << this->camino_obtenido.obtener_costo_total_w2_camino() << " ";
526         out << this->camino_obtenido.obtener_longitud_camino() << " ";
527         list<nodo_t>::const_iterator it = this->camino_obtenido.obtener_iterador_const_begin();
528         while(it != camino_obtenido.obtener_iterador_const_end()){
529             out << ( (formato == FORMATO_1_N_CLOSED) ? ( (*it) + 1 ) : *it ) << " ";
530             ++it;
531         }

```

```

532     }else{
533         out << "no";
534     }
535 }
536
537 bool Grafo::unserialize(istream& in, formato_entrada_t formato){
538     //primera linea:
539     //n m u v K
540     //cant nodos, cant aristas, src, dst, K(cota w1)
541     int cant_nodos_nuevos = 0, cantidad_aristas_nuevas = 0;
542     in >> cant_nodos_nuevos;
543
544     if(cant_nodos_nuevos == 0){
545         return false; //es la ultima linea de la entrada!!
546     }
547
548     in >> cantidad_aristas_nuevas;
549     in >> this->nodo_src;
550     in >> this->nodo_dst;
551     in >> this->cota_w1;
552
553     if(formato == FORMATO_1_N_CLOSED){
554         this->nodo_src--;
555         this->nodo_dst--;
556     }
557
558     this->agregar_nodos(cant_nodos_nuevos);
559
560     int count = 0;
561     while(count < cantidad_aristas_nuevas){
562         //las lineas de las aristas vienen asi:
563         // v1 v2 w1 w2
564         nodo_t nodo_a = 0, nodo_b = 0; costo_t costo_w1 = 0, costo_w2 = 0;
565         in >> nodo_a;
566         in >> nodo_b;
567         in >> costo_w1;
568         in >> costo_w2;
569
570         if(formato == FORMATO_1_N_CLOSED){
571             nodo_a--;
572             nodo_b--;
573         }
574
575         this->agregar_arista(nodo_a, nodo_b, costo_w1, costo_w2);
576         count++;
577     }
578
579     //precalcular vecinos en comun
580     for(int i=0; i<cant_nodos_nuevos; i++){
581         for(int j=0; j<cant_nodos_nuevos; j++){
582             precalcular_adyacentes_en_comun(i, j);
583         }
584     }
585     return true;
586 }
587
588 //Devuelve el camino minimo entre origen y destino(calcula el arbol, pero reconstruye solo el
589 //camino de origen a destino)
589 Camino Grafo::dijkstra(nodo_t origen, nodo_t destino, tipo_costo_t target_a_minimizar){
590     vector<costo_t> costo_minimo;
591     vector<nodo_t> predecesor;
592     this->dijkstra(origen, target_a_minimizar, costo_minimo, predecesor);
593
594     //armar camino entre origen y destino
595     Camino c(this->mat_adyacencia);
596     nodo_t nodo = destino;
597     do{
598         //cout << nodo << " " ;
599         c.agregar_nodo_adelante(nodo);
600         nodo = predecesor[nodo];
601     }while(nodo != predecesor_nulo);
602     return c;
603 }
604
605 //Aplica dijkstra desde nodo origen y calcula el arbol de caminos minimos por referencia a los
606 //vectores por parametro

```

```

606 void Grafo::dijkstra(nodo_t origen, tipo_costo_t target_a_minimizar, vector<costo_t>&
costo_minimo, vector<nodo_t>& predecesor){
607     //inicializacion
608     int n = this->cantidad_nodos;
609     costo_minimo.clear();
610     costo_minimo.resize(n, costo_infinito);
611     costo_minimo[origen] = costo_nulo;
612
613     predecesor.clear();
614     predecesor.resize(n, predecesor_nulo);
615
616     set<pair<costo_t, nodo_t>> cola;
617     cola.insert(make_pair(costo_minimo[origen], origen));
618
619     while(!cola.empty()){
620         costo_t cost_to_u = cola.begin()->first;
621         nodo_t nodo_u = cola.begin()->second;
622         //marcar como visto en este caso es eliminarlo de la cola
623         cola.erase(cola.begin());
624
625         //Para cada vecino de u, obtengo el vector fila de la matriz de adyacencia
626         //de dicho nodo e itero sobre todos, quedandome con las aristas presentes
627         lista_adyacentes::iterator adyacentes_i_it = this->lista_adyacencia[nodo_u].begin();
628         lista_adyacentes::iterator final_it = this->lista_adyacencia[nodo_u].end();
629         while(adyacentes_i_it != final_it){
630             nodo_t nodo_v = adyacentes_i_it->first;
631
632             costo_t cost_v = (target_a_minimizar == COSTO.W1) ? (adyacentes_i_it->second).
obtener_costo_w1() : (adyacentes_i_it->second).obtener_costo_w2();
633             int costo_a_v_pasando_por_u = cost_to_u + cost_v;
634             //si mejora, sobrescribo el camino y su costo
635             if(costo_a_v_pasando_por_u < costo_minimo[nodo_v]){
636                 //elimino de la cola de nodos la distancia vieja
637                 cola.erase(make_pair(costo_minimo[nodo_v], nodo_v));
638
639                 //actualizo estructuras
640                 costo_minimo[nodo_v] = costo_a_v_pasando_por_u;
641                 predecesor[nodo_v] = nodo_u;
642
643                 //sobrescribo la distancia en la cola de nodos la distancia nueva
644                 cola.insert(make_pair(costo_minimo[nodo_v], nodo_v));
645             }
646             adyacentes_i_it++;
647         }
648     }
649 }
650
651 void Grafo::breadth_first_search(nodo_t origen, vector<distancia_t>& distancias){
652     distancias.clear();
653     distancias.resize(this->cantidad_nodos, distancia_infinita);
654     queue<nodo_t> cola;
655     vector<bool> visitado(this->cantidad_nodos, false); //inicializo todos los nodos sin visitar
656     cola.push(origen);
657     visitado[origen] = true;
658     distancias[origen] = 0; //dist(src, src) = 0
659
660     while(!cola.empty()){
661         nodo_t target = cola.front(); //obtengo el primero
662         cola.pop(); //desencolo el primero
663
664         //para todos los vecinos de target
665         lista_adyacentes::iterator adyacentes_i_it = this->lista_adyacencia[target].begin();
666         lista_adyacentes::iterator final_it = this->lista_adyacencia[target].end();
667         while(adyacentes_i_it != final_it){
668             nodo_t vecino_candidato = adyacentes_i_it->first;
669             if(!visitado[vecino_candidato]){
670                 cola.push(vecino_candidato);
671                 visitado[vecino_candidato] = true;
672                 distancias[vecino_candidato] = distancias[target] + 1;
673             }
674             adyacentes_i_it++;
675         }
676     }
677 }
678
679 //para impl. con listas de adyacencia, es O(n**2) si no estan ordenados haciendo 2 fors

```

```

680 //se puede mejorar, "ordenando" las listas en espacios auxiliares en O(nlogn) y intersecando en
    O(n)
681 //dejando complejidad de O(nlogn), de cualquier manera, como se nos pidio que fuera polinomial
    unicamente
682 //uso la matriz de adyacencia, y en O(n) recorro los vecinos de ambos, y donde se cumpla
    vecindad en ambos
683 //lo selecciono como vecino en comun.
684 void Grafo::precalcular_adyacentes_en_comun(nodo_t i, nodo_t j){
685     vector<Vecino> res;
686     vector<Arista> adyacentesFila_i = this->mat_adyacencia[i];
687     vector<Arista> adyacentesFila_j = this->mat_adyacencia[j];
688     for(int idx=0;idx<this->cantidad_nodos;idx++){
689         if(adyacentesFila_i[idx].esta_presente() && adyacentesFila_j[idx].esta_presente()){
690             //el nodo idx es adyacente de i y j.
691             res.push_back(Vecino(i, j, idx, adyacentesFila_i[idx], adyacentesFila_j[idx]));
692         }
693     }
694     this->vecinos_comunes[i][j] = res;
695 }
696
697 vector<Vecino> Grafo::obtener_adyacentes_en_comun(nodo_t i, nodo_t j){
698     //cout << "Vecinos comunes de (" << i << ") y (" << j << ") : ";
699     #ifdef VECINOS_COMUNESLAZY
700         vector<Vecino> res;
701         vector<Arista> adyacentesFila_i = this->mat_adyacencia[i];
702         vector<Arista> adyacentesFila_j = this->mat_adyacencia[j];
703         for(int idx=0;idx<this->cantidad_nodos;idx++){
704             if(adyacentesFila_i[idx].esta_presente() && adyacentesFila_j[idx].esta_presente()){
705                 //el nodo idx es adyacente de i y j.
706                 res.push_back(Vecino(i, j, idx, adyacentesFila_i[idx], adyacentesFila_j[idx]));
707                 //cout << "(" << idx << ") --->";
708             }
709         }
710         //cout << "Nil" << endl;
711         return res;
712     #else
713         //list<Vecino> tmp = this->vecinos_comunes[i][j];
714         //for(Vecino v : tmp){
715             //cout << "(" << v.obtener_nodo_comun() << ") ---> ";
716         //}
717         //cout << "Nil" << endl;
718         return this->vecinos_comunes[i][j];
719     #endif
720 }
721
722 vector<Arista> Grafo::obtener_vector_fila_vecinos(nodo_t target){
723     return this->mat_adyacencia[target];
724 }
725
726 lista_adyacentes Grafo::obtener_lista_vecinos(nodo_t target){
727     return this->lista_adyacencia[target];
728 }
729
730 nodo_t Grafo::obtener_nodo_origen(){
731     return this->nodo_src;
732 }
733
734 nodo_t Grafo::obtener_nodo_destino(){
735     return this->nodo_dst;
736 }
737
738 bool Grafo::mejorar_conexion_salteando(nodo_t nodo_i, nodo_t nodo_j, costo_t costo_ij_w1,
    costo_t costo_ij_w2, costo_t total_w1, costo_t total_w2,
739 Arista& mejor_vecino){
740     //es un grafo no dirigido, es simetrico buscar si a i es adyacente a j o viceversa
741     //me fijo si i y j son adyacentes
742     Arista candidato_a_mejor_camino = this->obtener_arista(nodo_i, nodo_j);
743     if(candidato_a_mejor_camino.esta_presente()){
744         //FIX. ADEMÁS DE EXISTIR LA ARISTA, EL CAMINO DIRECTO TIENE QUE SER MEJOR Estricto QUE
            LA SUMA DE LAS DOS ARISTAS EXISTENTES
745
746         costo_t hipotetico_w1_total_camino = (total_w1 - (costo_ij_w1) +
            candidato_a_mejor_camino.obtener_costo_w1());
747         bool es_mejora_factible = (candidato_a_mejor_camino.obtener_costo_w2() < costo_ij_w2)
            /*mejora la arista directa*/&&
748             (hipotetico_w1_total_camino <= this->obtener_limite_w1()); /*no se pasa de la cota
            */

```

```

749         if(es_mejora_factible){
750             //la arista directa mejora el camino en w2 pero no se pasa de w1 el camino
751
752             //OJO en esta linea!: si asigno candidato_a_mejor_camino estoy devolviendo una
              referencia a una variable de stack y catapunchis!
753             mejor_vecino = this->obtener_arista(nodo_i, nodo_j);
754             return true;
755         }else{
756             //o no mejora o se pasa w1 => false
757             return false;
758         }
759     }else{
760         //no hay arista directa => false
761         return false;
762     }
763 }
764
765 bool Grafo::mejorar_conexion_entre_pares(nodo_t nodo_i, nodo_t nodo_j, costo_t costo_ij_w1,
      costo_t costo_ij_w2, costo_t total_w1, costo_t total_w2,
766 Vecino& mejor_vecino){
767     //SEA LA VECINDAD Vc = {Caminos C' tal que difieran de C en tan solo un nodo}
768
769     //DEFINO UNA TABOO LIST COMO UNA LISTA EN LA CUAL VOY A DESCARTAR LAS SOLUCIONES QUE USEN
      NODOS DE DICHA LISTA
770     //ESTO ES NECESARIO PARA EVITAR LA GENERACION DE CICLOS EN EL CASO DE AGREGAR UN NODO AL
      SUBDIVIDIR UNA ARISTA O REEMPLAZAR UN NODO
771     //INTERMEDIO ENTRE OTROS DOS. SI EL NODO ELEGIDO YA PERTENECIA AL CAMINO, SE GENERAN CICLOS
      Y NO QUEREMOS ESTO YA QUE BUSCAMOS UN CAMINO MINIMO.
772     //SI MANTENEMOS DISJUNTOS EL CONJUNTO DE NODOS DEL CAMINO ACTUAL Y LOS NODOS RESTANTES DEL
      GRAFO, CUALQUIER ELECCION QUE HAGAMOS NO GENERARA CICLOS.
773
774     //OTRA OPCION SERIA NO RESTRINGIR LA ELECCION DE LAS SOLUCIONES DE LA VECINDAD, PERO
      DEBERIAMOS LUEGO REALIZAR UNA PODA DE CICLOS DEL CAMINO
775     //CREEMOS QUE ESTA OPCION SERIA MEJOR, PORQUE AGREGANDO EL NODO SE MEJORA LA SOLUCION, Y
      ELIMINANDO EL CICLO, SE MEJORA AUN MAS, PERO A NUESTRO ENTENDER
776     //DEJA DE SER BUSQUEDA LOCAL, DADO QUE LA SOLUCION QUE SURJA DE ESTO PUEDE NO ESTAR EN LA
      VECINDAD Vc
777
778     //busco la conexion entre i y j pasando por un nodo intermedio tal que minimice la
      distancia de w2 sin pasarme de la cota total de w1 para el camino
779     vector<Vecino> vecinosEnComun = this->obtener_adyacentes_en_comun(nodo_i, nodo_j);
780     vector<Vecino>::iterator vecinos_it = vecinosEnComun.begin();
781     vector<Vecino>::iterator final_vecinos = vecinosEnComun.end();
782
783     //me fijo todos los caminos alternativos agregando un nodo entre los nodos ij,
784     vector<Vecino>::iterator mejor_vecino_it = vecinosEnComun.end();//inicializamos en algo que
      indique que no hay mejora
785     costo_t mejor_camino_ij_w2 = costo_ij_w2;
786     while(vecinos_it != final_vecinos){
787         costo_t i_comun_w1 = vecinos_it->obtener_arista_i_comun().obtener_costo_w1();
788         costo_t i_comun_w2 = vecinos_it->obtener_arista_i_comun().obtener_costo_w2();
789         costo_t j_comun_w1 = vecinos_it->obtener_arista_j_comun().obtener_costo_w1();
790         costo_t j_comun_w2 = vecinos_it->obtener_arista_j_comun().obtener_costo_w2();
791         costo_t costo_i_comun_j_w1 = i_comun_w1 + j_comun_w1;
792         costo_t costo_i_comun_j_w2 = i_comun_w2 + j_comun_w2;
793         costo_t hipotetico_w1_total_camino = (total_w1 - costo_ij_w1 + costo_i_comun_j_w1);
794         costo_t hipotetico_w2_total_camino = (total_w2 - costo_ij_w2 + costo_i_comun_j_w2);
795
796         //veamos si el camino es una solucion factible
797         if(hipotetico_w1_total_camino <= this->obtener_limite_w1()){
798             //es factible, veamos si mejora al ultimo mejor revisado
799
800             //VEAMOS ADEMAS, QUE NO ESTE EN NUESTRA TABOO LIST(QUE NO PERTENEZCA AL CAMINO DE
              LA SOL. ACTUAL)
801             nodo_t nodo_comun = vecinos_it->obtener_nodo_comun();
802             if(!this->camino_obtenido.pertenece_a_camino(nodo_comun)){//puedo verlo en O(1)
              //el nodo no esta en la taboo list
803                 if(costo_i_comun_j_w2 < mejor_camino_ij_w2){
804                     //encontre mejora, actualizo variables
805                     mejor_camino_ij_w2 = hipotetico_w2_total_camino;
806                     mejor_vecino_it = vecinos_it;
807                 }
808             }else{
809                 //taboo list skipped!
810                 #ifdef DEBUG_MESSAGES_ON
811                 #ifdef CYCLE_PREVENT_MESSAGE_ON

```

```

813         cout << "Salteamos el nodo (" << nodo_comun << ") entre (" << nodo_i << ") y (" << nodo_j << ") como candidato a mejorar la solución actual porque al pertenecer al camino generaría un ciclo" << endl;
814     #endif
815 #endif
816     }
817 }
818 ++vecinos_it;
819 }
820 if (mejor_vecino_it != final_vecinos) {
821     mejor_vecino = *mejor_vecino_it;
822     return true;
823 }
824 return false;
825 }
826
827 //Defino la vecindad del camino como Vc = {Todos los caminos c' que difieren en un nodo de c}
828 //Intento mejorar un camino vk→vk+1 con otro vk→vj→vk+1 tal que mejora w2 y w1 no se pasa en el costo total del camino
829 //Primero reviso todos los pares de nodos del camino buscando posibles subdivisiones que mejoren la solución.
830 //Me voy a quedar únicamente (si existen varias) con la mejor solución de la vecindad, notemos que revisar toda la vecindad es cuadrático
831 //La longitud de un camino simple puede acotarse por la cantidad de nodos n, luego hay n-1 pares de nodos en el camino
832 //Para cada par, es lineal la obtención de los vecinos en común, y al ser el camino una lista enlazada, la modificación tiene costo
833 //constante O(1), en total esto nos da un costo cuadrático O(n**2)
834 int Grafo::busqueda_local_entre_pares_insertando(Camino& solucion_actual, Vecino& conexion_ij_minima_w2) {
835     //cout << "-----Comienza iteración de búsqueda local insertando entre pares-----" << endl;
836     list<nodo_t>::const_iterator it = solucion_actual.obtener_iterador_const.begin();
837     list<nodo_t>::const_iterator runner_it = solucion_actual.obtener_iterador_const.begin();
838     runner_it++;
839     list<nodo_t>::const_iterator final_camino = solucion_actual.obtener_iterador_const.end();
840     costo_t total_w1 = solucion_actual.obtener_costo_total_w1_camino();
841     costo_t total_w2 = solucion_actual.obtener_costo_total_w2_camino();
842
843     costo_t total_inicial_w2 = total_w2;
844
845     //variables para guardar la mejor solución de la vecindad
846     costo_t mejor_costo_w2 = costo_infinito;
847     list<nodo_t>::const_iterator punto_de_insercion_mejora_it = final_camino;
848     bool hay_mejoras_para_el_camino = false;
849
850     //busco la mejor solución en la vecindad
851 #ifdef DEBUG_MESSAGES_ON
852     cout << "Solución actual: ";
853     solucion_actual.imprimir_camino(cout);
854     cout << "Costo total del camino actual: W1: " << total_w1 << " W2: " << total_w2 << endl;
855 #endif
856     while (runner_it != final_camino) {
857         nodo_t nodo_i = *it;
858         nodo_t nodo_j = *runner_it;
859         costo_t costo_ij_w1 = solucion_actual.obtener_costo_w1_entre_nodos(nodo_i, nodo_j);
860         costo_t costo_ij_w2 = solucion_actual.obtener_costo_w2_entre_nodos(nodo_i, nodo_j);
861
862         Vecino mejor_conexion_ij;
863         //le paso una ref a una var tipo vecino, si devuelve true, se escribe por referencia el mejor camino, sino, no cambia lo que le pasamos.
864         //cout << "Buscando mejorar la conexión (" << nodo_i << ")---[" << costo_ij_w1 << ", " << costo_ij_w2 << "]->(" << nodo_j << ") agregando un nodo intermedio..." << endl;
865         bool encuentre_mejora = mejorar_conexion_entre_pares(nodo_i, nodo_j, costo_ij_w1, costo_ij_w2, total_w1, total_w2, mejor_conexion_ij);
866
867         //hay que ver si encontramos una mejora
868         if (encontre_mejora) { //la función asegura que si dio true, me da el vecino por puntero en mejor_conexion_ij
869             //nodo_t nodo_comun = mejor_conexion_ij.obtener_nodo_comun();
870             costo_t i_comun_w1 = mejor_conexion_ij.obtener_arista_i_comun().obtener_costo_w1();
871             costo_t i_comun_w2 = mejor_conexion_ij.obtener_arista_i_comun().obtener_costo_w2();
872             costo_t j_comun_w1 = mejor_conexion_ij.obtener_arista_j_comun().obtener_costo_w1();

```

```

876         costo_t j_comun_w2 = mejor_conexion_ij.obtener_arista_j_comun().obtener_costo_w2();
877         costo_t costo_i_comun_j_w1 = i_comun_w1 + j_comun_w1;
878         costo_t costo_i_comun_j_w2 = i_comun_w2 + j_comun_w2;
879         //cout << "\tSe encontro una posible mejora. El mejor sendero entre los nodos que
            se encontro en todos los vecinos entre (" << nodo_i << ") y (" << nodo_j << ")
            es " << endl;
880         //cout << "\tCamino (" << nodo_i << ")----[" << i_comun_w1 << ", " << i_comun_w2 <<
            "]"---->(" << nodo_comun << ")----[" << j_comun_w1 << ", " << j_comun_w2 <<
            "]"---->(" << nodo_j;
881         //cout << ") Nuevo costo del sendero entre (" << nodo_i << ") y (" << nodo_j <<
            ") aplicando a esta modificacion: W1: " << costo_i_comun_j_w1 << " W2: "
            << costo_i_comun_j_w2 << endl;
882
883         costo_t hipotetico_w1_total_camino = (total_w1 - costo_ij_w1 + costo_i_comun_j_w1);
884         costo_t hipotetico_w2_total_camino = (total_w2 - costo_ij_w2 + costo_i_comun_j_w2);
885
886         if( (hipotetico_w2_total_camino < mejor_costo_w2) && (hipotetico_w1_total_camino <=
            this->obtener_limite_w1())){
887             mejor_costo_w2 = hipotetico_w2_total_camino;
888             conexion_ij_minima_w2 = mejor_conexion_ij;
889             punto_de_insercion_mejora_it = it;
890             hay_mejoras_para_el_camino = true;
891         }
892
893         //cout << "\tSi aplicamos este cambio los costos del camino total quedarian: W1:
            " << hipotetico_w1_total_camino << " W2: " << hipotetico_w2_total_camino <<
            endl;
894     }else{
895         //cout << "\tNo se encontro mejora." << endl;
896     }
897     //cout << endl;
898     ++it;
899     ++runner_it;
900 }
901
902 //Si hubo mejoras, tengo guardada la mejor
903 if(hay_mejoras_para_el_camino){
904     #ifdef DEBUG_MESSAGES_ON
905         cout << "Se encontro mejora. La mejor mejora encontrada entre todos los pares de
            nodos fue:" << endl;
906     #endif
907     //insertar conexion_ij_minima_w2 en punto_de_insercion_mejora_it
908     //y actualizar todos los atributos necesarios.
909
910     nodo_t nodo_i = conexion_ij_minima_w2.obtener_nodo_i();
911     nodo_t nodo_j = conexion_ij_minima_w2.obtener_nodo_j();
912     nodo_t nodo_comun = conexion_ij_minima_w2.obtener_nodo_comun();
913     costo_t i_comun_w1 = conexion_ij_minima_w2.obtener_arista_i_comun().obtener_costo_w1();
914     costo_t i_comun_w2 = conexion_ij_minima_w2.obtener_arista_i_comun().obtener_costo_w2();
915     costo_t j_comun_w1 = conexion_ij_minima_w2.obtener_arista_j_comun().obtener_costo_w1();
916     costo_t j_comun_w2 = conexion_ij_minima_w2.obtener_arista_j_comun().obtener_costo_w2();
917     costo_t costo_i_comun_j_w1 = i_comun_w1 + j_comun_w1;
918     costo_t costo_i_comun_j_w2 = i_comun_w2 + j_comun_w2;
919     #ifdef DEBUG_MESSAGES_ON
920         cout << "Se agregara un nodo intermedio en: (" << nodo_i << ")----[" << i_comun_w1
            << ", " << i_comun_w2 << "]"---->(" << nodo_comun << ")----[" << j_comun_w1 <<
            ", " << j_comun_w2 << "]"---->(" << nodo_j << ") " << endl;
921         cout << "Nuevo costo del sendero entre (" << nodo_i << ") y (" << nodo_j << ")
            aplicando a esta modificacion: W1: " << costo_i_comun_j_w1 << " W2: " <<
            costo_i_comun_j_w2 << endl;
922     #endif
923     return (total_inicial_w2 - mejor_costo_w2); //mejora respecto a w2 en el camino del
            inicial al actual
924 }else{
925     #ifdef DEBUG_MESSAGES_ON
926         cout << "No se pudo mejorar la solucion." << endl;
927     #endif
928     return 0; //mejoro en 0 el camino respecto a w2
929 }
930 }
931
932 int Grafo::busqueda_local_entre_triplas_reemplazando_intermedio(Camino& solucion_actual, Vecino
    & conexion_ij_minima_w2){
933     //Caso en que reemplazo vk+1 por otro vecino comun vj, convirtiendo vk---->vk+1---->vk+2 en
    vk---->vj---->vk+2 tal que mejora w2 y w1 no se pasa en el costo total del camino
934     //cout << "Comienza iteracion de busqueda local reemplazando
    intermedio" << endl;

```



```

935 if(solucion_actual.obtener_longitud_camino()<3){
936     #ifdef DEBUG_MESSAGES_ON
937         cerr << "Camino de menos de 3 nodos. No se puede mejorar nada." << endl;
938     #endif
939     return false;
940 }
941
942 //aca vale size(camino)>=3
943
944 list<nodo_t>::const_iterator it = solucion_actual.obtener_iterador_const_begin();
945 list<nodo_t>::const_iterator it_sig = solucion_actual.obtener_iterador_const_begin();
946 list<nodo_t>::const_iterator runner_it = solucion_actual.obtener_iterador_const_begin();
947 it_sig++;
948 runner_it++;runner_it++;
949
950 list<nodo_t>::const_iterator final_camino = solucion_actual.obtener_iterador_const_end();
951 costo_t total_w1 = solucion_actual.obtener_costo_total_w1_camino();
952 costo_t total_w2 = solucion_actual.obtener_costo_total_w2_camino();
953
954 costo_t total_inicial_w2 = total_w2;
955
956 //variables para guardar la mejor solucion de la vecindad
957 costo_t mejor_costo_w2 = costo_infinito;
958 list<nodo_t>::const_iterator punto_de_insercion_mejora_it = final_camino;
959 bool hay_mejoras_para_el_camino = false;
960
961 //busco la mejor solucion en la vecindad
962 #ifdef DEBUG_MESSAGES_ON
963     cout << "Solucion actual: ";
964     solucion_actual.imprimir_camino(cout);
965     cout << "Costo total del camino actual: W1: " << total_w1 << " W2: " << total_w2
966         << endl;
967 #endif
968 while(runner_it != final_camino){
969     nodo_t nodo_i = *it;
970     nodo_t nodo_medio = *it_sig;
971     nodo_t nodo_j = *runner_it;
972
973     costo_t costo_i_medio_w1 = solucion_actual.obtener_costo_w1_entre_nodos(nodo_i,
974         nodo_medio);
975     costo_t costo_i_medio_w2 = solucion_actual.obtener_costo_w2_entre_nodos(nodo_i,
976         nodo_medio);
977     costo_t costo_medio_j_w1 = solucion_actual.obtener_costo_w1_entre_nodos(nodo_medio,
978         nodo_j);
979     costo_t costo_medio_j_w2 = solucion_actual.obtener_costo_w2_entre_nodos(nodo_medio,
980         nodo_j);
981
982     //cout << "Intentando mejorar el nodo intermedio entre (" << nodo_i << ")----[" <<
983         costo_i_medio_w1 << ", " << costo_i_medio_w2;
984     //cout << "]----->(" << nodo_medio << ")----[" << costo_medio_j_w1 << ", " <<
985         costo_medio_j_w2 << "]----->(" << nodo_j << ")" << endl;
986
987     Vecino mejor_conexion_ij;
988     //le paso una ref a una var tipo Vecino, si devuelve true, se escribe por referencia el
989         mejor camino, sino, no cambia lo que le pasamos.
990     bool encuentre_mejora = mejorar_conexion_entre_pares(nodo_i, nodo_j,
991         (costo_i_medio_w1 + costo_medio_j_w1), (
992             costo_i_medio_w2 + costo_medio_j_w2),
993         total_w1, total_w2,
994         mejor_conexion_ij);
995
996     //hay que ver si encontramos una mejora
997     if(encuentre_mejora){//la funcion asegura que si dio true, me da el vecino por puntero
998         en mejor_conexion_ij
999         //nodo_t nodo_comun = mejor_conexion_ij.obtener_nodo_comun();
1000         costo_t i_comun_w1 = mejor_conexion_ij.obtener_arista_i_comun().obtener_costo_w1();
1001         costo_t i_comun_w2 = mejor_conexion_ij.obtener_arista_i_comun().obtener_costo_w2();
1002         costo_t j_comun_w1 = mejor_conexion_ij.obtener_arista_j_comun().obtener_costo_w1();
1003         costo_t j_comun_w2 = mejor_conexion_ij.obtener_arista_j_comun().obtener_costo_w2();
1004         costo_t costo_i_comun_j_w1 = i_comun_w1 + j_comun_w1;
1005         costo_t costo_i_comun_j_w2 = i_comun_w2 + j_comun_w2;
1006         //cout << "\tSe encontro una posible mejora. El mejor sendero entre los nodos que
1007             se encontro en todos los vecinos entre (" << nodo_i << ") y (" << nodo_j << ")
1008             es " << endl;
1009         //cout << "\tCamino (" << nodo_i << ")----[" << i_comun_w1 << ", " << i_comun_w2 <<
1010             "]----->(" << nodo_comun << ")----[" << j_comun_w1 << ", " << j_comun_w2 <<

```



```

1059     #endif
1060     return false;
1061 }
1062
1063 //aca vale size(camino)>=3
1064
1065 list<nodo_t>::const_iterator it = solucion_actual.obtener_iterador_const_begin();
1066 list<nodo_t>::const_iterator it_sig = solucion_actual.obtener_iterador_const_begin();
1067 list<nodo_t>::const_iterator runner_it = solucion_actual.obtener_iterador_const_begin();
1068 it_sig++;
1069 runner_it++;runner_it++;
1070
1071 list<nodo_t>::const_iterator final_camino = solucion_actual.obtener_iterador_const_end();
1072 costo_t total_w1 = solucion_actual.obtener_costo_total_w1_camino();
1073 costo_t total_w2 = solucion_actual.obtener_costo_total_w2_camino();
1074
1075 costo_t total_inicial_w2 = total_w2;
1076
1077 //variables para guardar la mejor solucion de la vecindad
1078 costo_t mejor_costo_w2 = costo_infinito;
1079 Arista conexion_ij_minima_w2;
1080 punto_de_salto_it = final_camino;
1081 bool hay_mejoras_para_el_camino = false;
1082
1083 //busco la mejor solucion en la vecindad
1084 #ifdef DEBUG_MESSAGES_ON
1085     cout << "Solucion Actual: ";
1086     solucion_actual.imprimir_camino(cout);
1087     cout << "Costo total del camino actual: W1: " << total_w1 << " W2: " << total_w2
1088         << endl;
1089 #endif
1090 while(runner_it != final_camino){
1091     nodo_t nodo_i = *it;
1092     nodo_t nodo_medio = *it_sig;
1093     nodo_t nodo_j = *runner_it;
1094
1095     costo_t costo_i_medio_w1 = solucion_actual.obtener_costo_w1_entre_nodos(nodo_i,
1096     nodo_medio);
1097     costo_t costo_i_medio_w2 = solucion_actual.obtener_costo_w2_entre_nodos(nodo_i,
1098     nodo_medio);
1099     costo_t costo_medio_j_w1 = solucion_actual.obtener_costo_w1_entre_nodos(nodo_medio,
1100     nodo_j);
1101     costo_t costo_medio_j_w2 = solucion_actual.obtener_costo_w2_entre_nodos(nodo_medio,
1102     nodo_j);
1103
1104     //cout << "Intentando remover el nodo intermedio entre (" << nodo_i << ")----[" <<
1105     costo_i_medio_w1 << ", " << costo_i_medio_w2;
1106     //cout << "]"---->(" << nodo_medio << ")----[" << costo_medio_j_w1 << ", " <<
1107     costo_medio_j_w2 << "]"---->(" << nodo_j << ") << endl;
1108
1109     Arista mejor_conexion_ij;
1110     //le paso una ref a una var tipo arista, si devuelve true, se escribe por referencia el
1111     mejor camino, sino, no cambia lo que le pasamos.
1112     bool encuentre_mejora = mejorar_conexion_salteando(nodo_i, nodo_j,
1113     (costo_i_medio_w1 + costo_medio_j_w1), (
1114     costo_i_medio_w2 + costo_medio_j_w2),
1115     total_w1, total_w2,
1116     mejor_conexion_ij);
1117
1118     //hay que ver si encontramos una mejora
1119     if(encuentre_mejora){//la funcion asegura que si dio true, me da el vecino por puntero
1120     en mejor_conexion_ij
1121     costo_t costo_ij_directo_w1 = mejor_conexion_ij.obtener_costo_w1();
1122     costo_t costo_ij_directo_w2 = mejor_conexion_ij.obtener_costo_w2();
1123     //cout << "\tSe encontro una posible mejora. Se encontro una arista directa entre
1124     (" << nodo_i << ") y (" << nodo_j << ") y es " << endl;
1125     //cout << "\tCamino (" << nodo_i << ")----[" << costo_ij_directo_w1 << ", " <<
1126     costo_ij_directo_w2 << "]"---->(" << nodo_j << ")";
1127     //cout << " Nuevo costo del sendero entre (" << nodo_i << ") y (" << nodo_j << ")
1128     aplicando a esta modificacion: W1: " << costo_ij_directo_w1 << " W2: "
1129     << costo_ij_directo_w2 << endl;
1130
1131     costo_t hipotetico_w1_total_camino = (total_w1 - (costo_i_medio_w1 +
1132     costo_medio_j_w1) + costo_ij_directo_w1);
1133     costo_t hipotetico_w2_total_camino = (total_w2 - (costo_i_medio_w2 +
1134     costo_medio_j_w2) + costo_ij_directo_w2);

```

```

1120
1121         if( (hipotetico.w2.total_camino < mejor_costo_w2) && (hipotetico.w1.total_camino <=
1122             this->obtener_limite_w1())){
1123             mejor_costo_w2 = hipotetico.w2.total_camino;
1124             hay_mejoras_para_el_camino = true;
1125             conexion.ij_minima_w2 = mejor_conexion.ij;
1126             punto_de_salto_it = it;
1127         }
1128         //cout << "\tSi aplicamos este cambio los costos del camino total quedarían:  W1:
1129         " << hipotetico.w1.total_camino << "      W2: " << hipotetico.w2.total_camino <<
1130         endl;
1131     }
1132     //cout << endl;
1133     ++it;
1134     ++it_sig;
1135     ++runner_it;
1136 }
1137 //Si hubo mejoras, tengo guardada la mejor
1138 if(hay_mejoras_para_el_camino){
1139     #ifdef DEBUG_MESSAGES_ON
1140         cout << "Se encontró mejora. La mejor mejora encontrada entre todos los pares de
1141         nodos fue:" << endl;
1142     #endif
1143     //Realizar el salto directo entre 2 nodos dada la arista obtenida
1144     list<nodo_t>::const_iterator nodo_j_it = punto_de_salto_it;
1145     nodo_j_it++;nodo_j_it++;
1146     nodo_t nodo_i = *punto_de_salto_it;
1147     nodo_t nodo_j = *nodo_j_it;
1148
1149     costo_t costo_ij_directo_w1 = conexion.ij_minima_w2.obtener_costo_w1();
1150     costo_t costo_ij_directo_w2 = conexion.ij_minima_w2.obtener_costo_w2();
1151     #ifdef DEBUG_MESSAGES_ON
1152         cout << "\tSe encontró una arista directa entre (" << nodo_i << ")_y_(" << nodo_j
1153         << ")_yes_" << endl;
1154         cout << "\tCamino_(" << nodo_i << ")----[" << costo_ij_directo_w1 << ",_" <<
1155         costo_ij_directo_w2 << "]---->(" << nodo_j << ")";
1156         cout << "\tNuevo costo del sendero entre (" << nodo_i << ")_y_(" << nodo_j << ")_
1157         aplicando a esta modificación: W1:_" << costo_ij_directo_w1 << " W2:_"
1158         << costo_ij_directo_w2 << endl;
1159     #endif
1160     return (total_inicial_w2 - mejor_costo_w2);
1161 }else{
1162     #ifdef DEBUG_MESSAGES_ON
1163         cout << "No se pudo mejorar la solución." << endl;
1164     #endif
1165     return 0;
1166 }
1167 }
1168
1169 int Grafo::busqueda_local(tipo_ejecucion_bqlocal_t tipo_ejecucion){
1170     //decido que busquedas se van a correr
1171     bool bql_subdiv = false;
1172     bool bql_mejorar = false;
1173     bool bql_contraer = false;
1174     switch(tipo_ejecucion){
1175         case BQL_SUBDIVIDIR_PARES:
1176             bql_subdiv = true;
1177             break;
1178         case BQL_CONTRAER_TRIPLAS_A_PARES:
1179             bql_contraer = true;
1180             break;
1181         case BQL_MEJORAR_CONEXION_TRIPLAS:
1182             bql_mejorar = true;
1183             break;
1184         case BQL_COMBINAR:
1185             bql_subdiv = true;
1186             bql_contraer = true;
1187             bql_mejorar = true;
1188             break;
1189         default:
1190             cerr << "busqueda_local: tipo_ejecucion_invalido!" << endl;
1191             return 0;
1192             break;
1193     }
1194 }

```

```

1189 }
1190
1191 //establezco solucion actual y corro los algoritmos seleccionados previamente
1192 Camino solucion_actual = this->camino_obtenido;
1193 costo_t mejora_en_w2_bql_entre_pares = 0;
1194 Vecino conexion_ij_minima_w2_entre_pares;
1195
1196 costo_t mejora_en_w2_entre_triplas_reemplazando = 0;
1197 Vecino conexion_ij_minima_w2_entre_triplas;
1198
1199 costo_t mejora_en_w2_entre_triplas_salteando = 0;
1200 list<nodo_t>::const_iterator punto_de_salto_it = solucion_actual.obtener_iterador_const_end
    ();
1201
1202 if(bql_subdiv){
1203     #ifdef DEBUG_MESSAGES_ON
1204         cout << endl << "\t——_Ejecutando_búsqueda_local_entre_pares_insertando_——" <<
            endl << endl;
1205     #endif
1206     mejora_en_w2_bql_entre_pares = búsqueda_local_entre_pares_insertando(solucion_actual ,
        conexion_ij_minima_w2_entre_pares);
1207     #ifdef DEBUG_MESSAGES_ON
1208         if(mejora_en_w2_bql_entre_pares > 0){
1209             cout << "\t——_Mejora_tentativa_sobre_w2_en_el_camino_actual_aplicando_esta_
                mejora:_ " << mejora_en_w2_bql_entre_pares << endl;
1210         }
1211     #endif
1212 }
1213
1214 if(bql_mejorar){
1215     #ifdef DEBUG_MESSAGES_ON
1216         cout << endl << "\t——_Ejecutando_búsqueda_local_entre_triplas_reemplazando_
            intermedio_——" << endl << endl;
1217     #endif
1218     mejora_en_w2_entre_triplas_reemplazando =
        búsqueda_local_entre_triplas_reemplazando_intermedio(solucion_actual ,
            conexion_ij_minima_w2_entre_triplas);
1219     #ifdef DEBUG_MESSAGES_ON
1220         if(mejora_en_w2_entre_triplas_reemplazando > 0){
1221             cout << "\t——_Mejora_tentativa_sobre_w2_en_el_camino_actual_aplicando_esta_
                mejora:_ " << mejora_en_w2_entre_triplas_reemplazando << endl;
1222         }
1223     #endif
1224 }
1225
1226 if(bql_contraer){
1227     #ifdef DEBUG_MESSAGES_ON
1228         cout << endl << "\t——_Ejecutando_búsqueda_local_entre_triplas_salteando_——" <<
            endl << endl;
1229     #endif
1230     mejora_en_w2_entre_triplas_salteando = búsqueda_local_entre_triplas_salteando(
        solucion_actual , punto_de_salto_it);
1231     #ifdef DEBUG_MESSAGES_ON
1232         if(mejora_en_w2_entre_triplas_salteando > 0){
1233             cout << "\t——_Mejora_tentativa_sobre_w2_en_el_camino_actual_aplicando_esta_
                mejora:_ " << mejora_en_w2_entre_triplas_salteando << endl;
1234         }
1235     #endif
1236 }
1237
1238 //si son todas las mejoras 0, no mejoro ninguna nada
1239 if( (mejora_en_w2_bql_entre_pares == 0) && (mejora_en_w2_entre_triplas_reemplazando == 0)
    && (mejora_en_w2_entre_triplas_salteando == 0) ){
1240     return 0;
1241 }else{
1242     //existe al menos uno de los 3 que mejoro, es decir que es > 0
1243     if(mejora_en_w2_bql_entre_pares > mejora_en_w2_entre_triplas_reemplazando){
1244         //vale mejora_en_w2_bql_entre_pares > mejora_en_w2_entre_triplas_reemplazando
1245         if(mejora_en_w2_bql_entre_pares > mejora_en_w2_entre_triplas_salteando){
1246             //max = mejora_en_w2_bql_entre_pares;
1247             if(mejora_en_w2_bql_entre_pares > 0){
1248                 if(solucion_actual.insertar_nodo(conexion_ij_minima_w2_entre_pares)){
1249                     #ifdef DEBUG_MESSAGES_ON
1250                         cout << endl << "Nueva_solucion_obtenida:_ ";
1251                         solucion_actual.imprimir_camino(cout);
1252                         cout << "Nuevos_costos_totales_del_camino:___W1:_ " <<
                            solucion_actual.obtener_costo_total_w1_camino() << "___W2:_ "

```

```

1253         << solucion_actual.obtener_costo_total_w2_camino() << endl;
1254     #endif
1255     this->establecer_camino_solucion(solucion_actual);
1256     return mejora_en_w2_bql_entre_pares;
1257 }else{
1258     return 0;
1259 }
1260 }else{
1261     return 0;
1262 }
1263 }else{
1264     //max = mejora_en_w2_entre_triplas_salteando;
1265     if(mejora_en_w2_entre_triplas_salteando > 0){
1266         nodo_t nodo_i = *punto_de_salto_it;
1267         if(solucion_actual.realizar_salto_entre_3_nodos(nodo_i)){
1268             //No puede haber ciclos, porque el camino quedo igual o con menos nodos
1269             #ifdef DEBUG_MESSAGES_ON
1270                 cout << endl << "Nueva_solucion_obtenida: ";
1271                 solucion_actual.imprimir_camino(cout);
1272                 cout << "Nuevos_costos_totales_del_camino: W1: " <<
1273                     solucion_actual.obtener_costo_total_w1_camino() << " W2: "
1274                     << solucion_actual.obtener_costo_total_w2_camino() << endl;
1275             #endif
1276             this->establecer_camino_solucion(solucion_actual);
1277             return mejora_en_w2_entre_triplas_salteando;
1278         }else{
1279             return 0;
1280         }
1281     }
1282 }else{
1283     //vale mejora_en_w2_bql_entre_pares <= mejora_en_w2_entre_triplas_reemplazando
1284     if(mejora_en_w2_entre_triplas_reemplazando > mejora_en_w2_entre_triplas_salteando){
1285         //max = mejora_en_w2_entre_triplas_reemplazando;
1286         if(mejora_en_w2_entre_triplas_reemplazando > 0){
1287             if(solucion_actual.mejorar_tripla(conexion_ij_minima_w2_entre_triplas)){
1288                 #ifdef DEBUG_MESSAGES_ON
1289                     cout << endl << "Nueva_solucion_obtenida: ";
1290                     solucion_actual.imprimir_camino(cout);
1291                     cout << "Nuevos_costos_totales_del_camino: W1: " <<
1292                         solucion_actual.obtener_costo_total_w1_camino() << " W2: "
1293                         << solucion_actual.obtener_costo_total_w2_camino() << endl;
1294                 #endif
1295                 this->establecer_camino_solucion(solucion_actual);
1296                 return mejora_en_w2_entre_triplas_reemplazando;
1297             }else{
1298                 return 0;
1299             }
1300         }else{
1301             return 0;
1302         }
1303     }else{
1304         //max = mejora_en_w2_entre_triplas_salteando;
1305         if(mejora_en_w2_entre_triplas_salteando > 0){
1306             nodo_t nodo_i = *punto_de_salto_it;
1307             if(solucion_actual.realizar_salto_entre_3_nodos(nodo_i)){
1308                 //No puede haber ciclos, porque el camino quedo igual o con menos nodos
1309                 #ifdef DEBUG_MESSAGES_ON
1310                     cout << endl << "Nueva_solucion_obtenida: ";
1311                     solucion_actual.imprimir_camino(cout);
1312                     cout << "Nuevos_costos_totales_del_camino: W1: " <<
1313                         solucion_actual.obtener_costo_total_w1_camino() << " W2: "
1314                         << solucion_actual.obtener_costo_total_w2_camino() << endl;
1315                 #endif
1316                 this->establecer_camino_solucion(solucion_actual);
1317                 return mejora_en_w2_entre_triplas_salteando;
1318             }else{
1319                 return 0;
1320             }
1321         }else{
1322             return 0;
1323         }
1324     }
1325 }
1326 }
1327 }
1328 }
1329 }
1330 }
1331 }
1332 }

```

```

1323 }
1324
1325 list<Grafo> Grafo::parsear_varias_instancias(formato_entrada_t formato){
1326     list<Grafo> instancias;
1327     //parseo todas las instancias
1328     bool instancia_valida = true;
1329     do{
1330         Grafo i(0);
1331         instancia_valida = i.unserialize(cin, formato);
1332         if(instancia_valida)
1333             instancias.push_back(i);
1334     }while(instancia_valida);
1335     //cout << "[Parse input]Se leyeron " << instancias.size() << " instancias de stdin" << endl
1336     << endl;
1337     return instancias;
1338 }
1339
1340 void Grafo::establecer_se_encontro_solucion(bool se_encontro){
1341     this->sol_valida = se_encontro;
1342 }
1343
1344 bool Grafo::hay_solucion(){
1345     return this->sol_valida;
1346 }
1347
1348 Camino Grafo::obtener_solucion_golosa(){
1349     int n = this->cantidad_nodos;
1350     int k = obtener_limite_w1();
1351     nodo_t origen = obtener_nodo_origen();
1352     nodo_t destino = obtener_nodo_destino();
1353
1354     //camino a devolver
1355     Camino c(this->mat_adyacencia);
1356
1357     //dijkstra de inicializacion
1358     vector<costo_t> costosw1;
1359     vector<nodo_t> predecesores;
1360     this->dijkstra(destino, COSTO_W1, costosw1, predecesores);
1361
1362     //ACA YA PUEDO SABER SI NO VA A HABER SOLUCION FACTIBLE CON EL DIJKSTRA PREPROCESADO
1363     if(costosw1[origen] == costo_infinito){//dist_w1(origen, destino) == infinito?
1364         cerr << "No existe solucion factible. No existe camino entre origen(" << origen << ") y
1365             << destino(" << destino << ")_" << endl;
1366         //ESTO HACE SI LE PONES FALSE, QUE GRAFO IMPRIMA "no" EN EL METODO SERIALIZE DE LA
1367         SALIDA.
1368         //ADEMAS DE SER UTIL PARA PREGUNTAR DESDE EL MAIN SI HUBO SOL.
1369         this->establecer_se_encontro_solucion(false);
1370     }else if(costosw1[origen] > k){//dist_w1(origen, destino) > limit_w1 = k?
1371         cerr << "No existe solucion factible. El camino minimo respecto a_w1 de origen(" <<
1372             << origen << ") a destino(" << destino << ") es de costo_" << costosw1[origen] << endl
1373         ;
1374         //ESTO HACE SI LE PONES FALSE, QUE GRAFO IMPRIMA "no" EN EL METODO SERIALIZE DE LA
1375         SALIDA.
1376         //ADEMAS DE SER UTIL PARA PREGUNTAR DESDE EL MAIN SI HUBO SOL.
1377         this->establecer_se_encontro_solucion(false);
1378     }else{
1379         //_____ Comienza Greedy
1380
1381         //Estructuras del greedy
1382         vector<costo_t> costosw2(n, costo_infinito);
1383         // contiene el costo w1 del camino recorrido hasta cada nodo
1384         vector<costo_t> costoCamino(n, costo_nulo);
1385
1386         //Comienza el algoritmo
1387         //reseteamos los predecesores
1388         predecesores.clear();
1389         predecesores.resize(n, predecesor_nulo);
1390
1391         costosw2[origen] = costo_nulo;
1392
1393         set<pair<costo_t, nodo_t>> cola;
1394         cola.insert(make_pair(costosw2[origen], origen));
1395
1396         while(!cola.empty()){
1397             pair<costo_t, nodo_t> actual = *cola.begin();
1398             cola.erase(cola.begin());

```

```

1393         lista_adyacentes vecinos = this->obtener_lista_vecinos(actual.second);
1394
1395     for(auto w : vecinos)
1396     {
1397         nodo_t nodoW = w.first;
1398         Arista aristaActualW = w.second;
1399
1400         if (costoCamino[actual.second] + costosw1[nodoW] + aristaActualW.
1401             obtener_costo_w1() <= k){
1402             costo_t costo_tentativo = costosw2[actual.second] + aristaActualW.
1403                 obtener_costo_w2();
1404             if (costosw2[nodoW] > costo_tentativo)
1405             {
1406                 cola.erase(make_pair(costosw2[nodoW], nodoW));
1407
1408                 costosw2[nodoW] = costo_tentativo;
1409                 costoCamino[nodoW] = costoCamino[actual.second] + aristaActualW.
1410                     obtener_costo_w1();
1411                 predecesores[nodoW] = actual.second;
1412                 cola.insert(make_pair(costosw2[nodoW], nodoW));
1413             }
1414         }
1415     }
1416
1417     //
1418     //{
1419     //
1420     //armar camino encontrado por la greedy
1421     nodo_t nodo = destino;
1422     //cout <<"Nodos" << endl;
1423     do{
1424         //cout << nodo << " ";
1425         c.agregar_nodo_adelante(nodo);
1426         nodo = predecesores[nodo];
1427     }while(nodo != predecesor_nulo);
1428
1429     //cout << endl << "Fin Nodos" << endl;
1430     this->establecer_se_encontro_solucion(true);
1431     //}
1432     //----- Fin Greedy
1433 }
1434 return c;
1435 }
1436
1437 //parametro beta en RCLPORVALOR indica el porcentaje a alejarse del mejor candidato
1438 //parametro beta en RCLPORCANTIDAD indica la cantidad de "mejores" candidatos a elegir
1439 set<pair<costo_t, nodo_t> >::iterator Grafo::obtener_candidato_randomizado(
1440     tipo_ejecucion_golosa_t tipo_ejecucion, const set<pair<costo_t, nodo_t> > & cola, double
1441     parametro_beta){
1442     set<pair<costo_t, nodo_t> >::iterator retorno;
1443     if(tipo_ejecucion == RCLDETERMINISTICO){
1444         retorno = cola.begin();
1445     }else if(tipo_ejecucion == RCLPORCANTIDAD){
1446         //tengo que elegir un numero i, aleatorio entre 0 y min{cola.size(), parametro_beta} -1
1447         //elemento de la cola
1448         //y devolver el i-esimo elemento en orden de la cola
1449         //dado que la cola esta ordenada, cumple una RCLPORCANTIDAD
1450         uint rcl_target_top = (uint) std::min((uint)cola.size(), (uint) parametro_beta) - 1;
1451
1452         //generacion numero random c++11 con distribucion uniforme
1453         //random_device rd;
1454         //mt19937 gen(rd());
1455         //uniform_int_distribution<> dis(0, rcl_target_top);
1456         //uint rcl_target_random = dis(gen); // generates number in the range 0..rcl_target_top
1457
1458         //lo cambio por esto de C-legacy porque lo de c++11 me aumenta violentamente el tiempo
1459         //de ejecucion
1460         srand(time(NULL));
1461         uint rcl_target_random = 0;
1462         if(rcl_target_top != 0){
1463             rcl_target_random = (unsigned int) rand() % rcl_target_top;
1464         }
1465     }

```



```

1462 //cout << "cola size: " << cola.size() << endl;
1463 //cout << "parametro-beta: " << parametro_beta << endl;
1464 //cout << "rango random [0.." << rcl_target_top << "]" << endl;
1465 //cout << "tomando random: " << rcl_target_random << endl;
1466 //for(int i=0; i<22;i++){
1467 //    cout << dis(gen) << endl;
1468 //}
1469
1470 //sabemos que esta en rango del iterador pues rcl_target_top <= cola.size() -1 y
1471 //    tomamos begin() que es el primero, es decir 0
1472 retorno = cola.begin();
1473 uint avance = 0;
1474 assert(avance >= 0);
1475 assert(avance < cola.size());
1476 while(avance < rcl_target_random){
1477     retorno++;
1478     avance++;
1479 }
1480 if(avance == 0){
1481     assert(retorno == cola.begin());
1482 }
1483 }else if (tipo_ejecucion == RCLPOR_VALOR){
1484     //itero sobre toda la cola, filtrando los elementos que esten dentro del porcentaje del
1485     //    parametro
1486     //luego selecciono uno al azar del vector de candidatos filtrados
1487     pair<costo_t, nodo_t> minimo = *cola.begin();
1488     double valor_limite = (parametro_beta + 1) * minimo.first;
1489     vector<set<pair<costo_t, nodo_t> >::iterator > candidatos;
1490
1491     set<pair<costo_t, nodo_t> >::iterator it_cola = cola.begin();
1492     set<pair<costo_t, nodo_t> >::iterator it_cola_fin = cola.end();
1493
1494     while(it_cola != it_cola_fin){
1495         if(it_cola->first <= valor_limite){
1496             candidatos.push_back(it_cola);
1497         }
1498         it_cola++;
1499     }
1500
1501     //random_device rd;
1502     //mt19937 gen(rd());
1503     //uniform_int_distribution<> dis(0, candidatos.size() -1);
1504     //uint rcl_target_random = dis(gen); // generates number in the range 0..candidatos.
1505     //    size() -1
1506
1507     //lo cambio por esto de C-legacy porque lo de c++11 me aumenta violentamente el tiempo
1508     //    de ejecucion
1509     uint rcl_target_top = candidatos.size() -1;
1510     srand(time(NULL));
1511     uint rcl_target_random = 0;
1512     if(rcl_target_top > 0){
1513         rcl_target_random = (unsigned int) rand() % rcl_target_top;
1514     }
1515
1516     retorno = candidatos[rcl_target_random];
1517 }else{
1518     cerr << "[Error]_Parametro_no_soportado_de_randomizacion_de_RCL._Asumiendo_
1519     RCL_DETERMINISTICO_" << tipo_ejecucion << endl;
1520     retorno = cola.begin();
1521 }
1522 return retorno;
1523 }
1524
1525 costo_t Grafo::obtener_costo_actual_w1_solucion(){
1526     return this->camino_obtenido.obtener_costo_total_w1_camino();
1527 }
1528
1529 costo_t Grafo::obtener_costo_actual_w2_solucion(){
1530     return this->camino_obtenido.obtener_costo_total_w2_camino();
1531 }
1532
1533 //typedef enum tipo_ejecucion_golosa_t {RCL_DETERMINISTICO, RCLPOR_VALOR, RCLPOR_CANTIDAD}
1534 //    tipo_ejecucion_golosa_t;
1535 Camino Grafo::obtener_solucion_golosa_randomizada(tipo_ejecucion_golosa_t tipo_ejecucion,
1536     double parametro_beta){
1537     int n = this->cantidad_nodos;
1538     int k = obtener_limite_w1();

```

```

1532 nodo_t origen = obtener_nodo_origen();
1533 nodo_t destino = obtener_nodo_destino();
1534
1535 //camino a devolver
1536 Camino c(this->mat_adyacencia);
1537
1538 vector<bool> visitados(n, false);
1539
1540 //dijkstra de inicializacion
1541 vector<costo_t> costosw1;
1542 vector<nodo_t> predecesores;
1543 this->dijkstra(destino, COSTO_W1, costosw1, predecesores);
1544
1545 //ACA YA PUEDO SABER SI NO VA A HABER SOLUCION FACTIBLE CON EL DIJKSTRA PREPROCESADO
1546 if(costosw1[origen] == costo_infinito){//dist_w1(origen, destino) == infinito?
1547     cerr << "No existe solucion factible. No existe camino entre origen(" << origen << ")_y
1548         _destino(" << destino << ")_" << endl;
1549     //ESTO HACE SI LE PONES FALSE, QUE GRAFO IMPRIMA "no" EN EL METODO SERIALIZE DE LA
1550     SALIDA.
1551     //ADEMAS DE SER UTIL PARA PREGUNTAR DESDE EL MAIN SI HUBO SOL.
1552     this->establecer_se_encontro_solucion(false);
1553 }else if(costosw1[origen] > k){//dist_w1(origen, destino) > limit_w1 = k?
1554     cerr << "No existe solucion factible. El camino minimo respecto a_w1 de origen(" <<
1555         origen << ")_a_destino(" << destino << ")_es de costo_" << costosw1[origen] << endl
1556         ;
1557     //ESTO HACE SI LE PONES FALSE, QUE GRAFO IMPRIMA "no" EN EL METODO SERIALIZE DE LA
1558     SALIDA.
1559     //ADEMAS DE SER UTIL PARA PREGUNTAR DESDE EL MAIN SI HUBO SOL.
1560     this->establecer_se_encontro_solucion(false);
1561 }else{
1562     //----- Comienza Greedy randomized
1563
1564     //Estructuras del greedy
1565     vector<costo_t> costosw2(n, costo_infinito);
1566     // contiene el costo w1 del camino recorrido hasta cada nodo
1567     vector<costo_t> costoCamino(n, costo_nulo);
1568
1569     //Comienza el algoritmo
1570     //reseteamos los predecesores
1571     predecesores.clear();
1572     predecesores.resize(n, predecesor_nulo);
1573
1574     costosw2[origen] = costo_nulo;
1575
1576     visitados[origen] = true;
1577
1578     set<pair<costo_t, nodo_t> > cola;
1579     cola.insert(make_pair(costosw2[origen], origen));
1580
1581     while(!cola.empty()){
1582         //Seleccion de decision greedy segun tipo de ejecucion requerido por parametro
1583         set<pair<costo_t, nodo_t> >::iterator it_candidato = obtener_candidato_randomizado(
1584             tipo_ejecucion, cola, parametro_beta);
1585
1586         pair<costo_t, nodo_t> actual = *it_candidato;
1587         cola.erase(it_candidato);
1588         visitados[actual.second] = true;
1589
1590         lista_adyacentes vecinos = this->obtener_lista_vecinos(actual.second);
1591
1592         for(auto w : vecinos)
1593         {
1594             nodo_t nodoW = w.first;
1595             Arista aristaActualW = w.second;
1596
1597             if ((costoCamino[actual.second] + costosw1[nodoW] + aristaActualW.
1598                 obtener_costo_w1() <= k) && !(visitados[nodoW])){
1599                 costo_t costo_tentativo = costosw2[actual.second] + aristaActualW.
1600                     obtener_costo_w2();
1601                 if (costosw2[nodoW] > costo_tentativo)
1602                 {
1603                     cola.erase(make_pair(costosw2[nodoW], nodoW));
1604
1605                     costosw2[nodoW] = costo_tentativo;
1606                     costoCamino[nodoW] = costoCamino[actual.second] + aristaActualW.
1607                         obtener_costo_w1();
1608                     predecesores[nodoW] = actual.second;
1609                 }
1610             }
1611         }
1612     }
1613 }

```

```

1599         cola.insert(make_pair(costosw2[nodoW], nodoW));
1600     }
1601 }
1602 }
1603 }
1604 }
1605
1606 //if(se_encontro_sol?) suponiendo que si hay sol factible el algoritmo encuentra una
1607 //solucion, esto no hace falta
1608 //{
1609 //
1610 //armar camino encontrado por la greedy
1611 nodo_t nodo = destino;
1612 //cout <<"Nodos" << endl;
1613 do{
1614     //cout << nodo << " " ;
1615     c.agregar_nodo_adelante(nodo);
1616     nodo = predecesores[nodo];
1617 }while(nodo != predecesor_nulo);
1618
1619 //assert replaced
1620 //assert(c.obtener_costo_total_w1_camino() == costoCamino[destino]);
1621 if(c.obtener_costo_total_w1_camino() != costoCamino[destino]){
1622     //c.imprimir_camino(cout);
1623     //cout << "Camino invalido" << endl;
1624     this->establecer_se_encontro_solucion(false);//no es sol factible
1625 }
1626
1627 //cout << endl << "Fin Nodos" << endl;
1628 this->establecer_se_encontro_solucion(true);
1629 //}
1630 //----- Fin Greedy randomized
1631 }
1632 return c;
1633 }
1634 //

```

```

1635
1636 #include "grafo.h"
1637 #include "timing.h"
1638 #include <fstream>
1639 #include <cmath>
1640
1641 #define FILE.ITERES.MEJORA "evolucion_iteraciones_grasp.txt"
1642 #define FILE.ITERES.COSTOS.ABSOLUTOS "costos_absolutos_iteraciones_grasp.txt"
1643 #define FILE.ITERES.COSTOS.ABSOLUTOS.STATISTICS "costos_absolutos_iteraciones_grasp_analisis.txt"
1644
1645 void ejecutar_grasp(Grafo &g);
1646
1647 //----- Main -----
1648 int main(int argc, char **argv){
1649     list<Grafo> instancias = Grafo::parsear_varias_instancias(FORMATO_1_N_CLOSED);
1650     uint64_t instance_number = 1;
1651     for(Grafo &g : instancias){
1652         #ifdef DEBUG_MESSAGES_ON
1653             cout << endl << endl << "Aplicando metaheuristica GRASP a la " << instance_number
1654                 << "-esima instancia de input..." << endl;
1655         #endif
1656         ejecutar_grasp(g);
1657         instance_number++;
1658     }
1659     return 0;
1660 }
1661 void ejecutar_grasp(Grafo &g){
1662     #ifdef DEBUG_MESSAGES_ON
1663         //g.imprimir_lista_adyacencia(cout);
1664         //g.imprimir_matriz_adyacencia(cout);
1665     #endif
1666
1667 //----- Configuracion del criterio de terminacion de GRASP -----
1668

```

```

1669 //typedef enum criterio_terminacion_grasp_t {CRT_K_ITEERS_SIN_MEJORA,
1670         CRT_K_ITEERS_LIMIT_REACHED} criterio_terminacion_grasp_t;
1671 criterio_terminacion_grasp_t criterio_terminacion = CRT_K_ITEERS_SIN_MEJORA;
1672 //este parametro denota la cantidad de iteraciones maxima, dependiendo del tipo de criterio
1673         , de cantidad fija de iteraciones o cantidad de iters
1674 //consecutivas sin mejora
1675 uint64_t ITEERS_LIMIT = 5;
1676 //consecutivas sin que greedy rand me de una solucion factible
1677 uint64_t RAND_GREEDY_BAD_ITEERS_LIMIT = 3;
1678 //este parametro denota el valor aceptable de la funcion objetivo w2 a partir del cual,
1679         dejamos de mejorar la solucion y consideramos que es lo suficientemente buena
1680
1681 //—— Configuracion de los modos de la busqueda local y golosa ——
1682
1683 //typedef enum tipo_ejecucion_bqlocal_t {BQL_SUBDIVIDIR_PARES, BQL_CONTRAER_TRIPLAS_A_PARES
1684         , BQL_MEJORAR_CONEXION_TRIPLAS, BQL_COMBINAR} tipo_ejecucion_bqlocal_t;
1685 tipo_ejecucion_bqlocal_t modo_busqueda_local = BQL_COMBINAR;
1686 //typedef enum tipo_ejecucion_golosa_t {RCL_DETERMINISTICO, RCL_POR_VALOR, RCL_POR_CANTIDAD
1687         } tipo_ejecucion_golosa_t;
1688 tipo_ejecucion_golosa_t modo_golosa = RCL_POR_CANTIDAD;
1689 //si el tipo de golosa es RCL_POR_VALOR, este parametro indica el porcentaje de alejamiento
1690         del minimo de los candidatos de la lista
1691 //mas formalmente filtra todos los candidatos factibles locales que no cumplan candidato->
1692         costo_w2 <= valor_limite
1693 //donde valor_limite es (parametro_beta + 1) * minimo.second.obtener_costo_w2();
1694 //si el tipo de golosa es RCL_POR_CANTIDAD, este parametro indica la cantidad min{
1695         cant_candidatos, parametro_beta} de soluciones a considerar en la lista
1696 //si el tipo es RCL_DETERMINISTICO, este parametro es ignorado por el metodo.
1697 double parametro_beta = ceil(g.obtener_cantidad_nodos()/(float)1);
1698 if(modo_golosa == RCL_POR_CANTIDAD){
1699     assert(parametro_beta >= 1);
1700 } else if(modo_golosa == RCL_POR_VALOR){
1701     assert(parametro_beta > 0);
1702 }
1703
1704 //—————
1705
1706 bool condicion_terminacion = false;
1707 Camino mejor_solucion = g.obtener_camino_vacio();
1708 costo_t costo_mejor_solucion = costo_infinito;
1709 uint64_t cant_iters = 0;
1710 uint64_t cant_iters_sin_sol_greedy_rand_factible = 0;
1711 uint64_t cant_iters_sin_mejora = 0;
1712 double tiempo_golosa_randomized = 0;
1713 double promedio = 0;
1714 Camino camino = g.obtener_camino_vacio();
1715 bool sol_valida_greedy = false;
1716 vector<pair<uint, costo_t>> mejora_iters_grasp;
1717 vector<pair<costo_t, costo_t>> costo_camino_en_iteraciones; //costos w1, w2
1718
1719 do{
1720     tiempo_golosa_randomized=0;
1721     MEDIR_TIEMPO_PROMEDIO(
1722         camino = g.obtener_solucion_golosa_randomizada(modo_golosa, parametro_beta);
1723         //cout << "Solucion inicial de la greedy:" << endl;
1724         //camino.imprimir_camino(cout);
1725         , 1, &tiempo_golosa_randomized);
1726
1727     //la sol greedy rand a veces da cosas no factibles, asi que verifico:
1728     //g.hay_solucion() nos indica si existe una sol factible(greedy rand lo setea en false
1729         si el minimo dijktra sobre w1 > limit_w1)
1730     //(camino.obtener_costo_total_w1_camino() <= g.obtener_limite_w1()); para chequear la
1731         validez del camino final de greedy rand
1732     sol_valida_greedy = (camino.obtener_costo_total_w1_camino() <= g.obtener_limite_w1());
1733     if(g.hay_solucion()){
1734         if(sol_valida_greedy){//puede que la greedy randomized no encuentre solucion!
1735             //hago iteraciones de busqueda local hasta que no haya mejora(la funcion
1736                 devuelve true si hubo mejora, false sino)
1737             g.establecer_camino_solucion(camino);
1738
1739             //reseteo el contador de sol malas greedy rand
1740             cant_iters_sin_sol_greedy_rand_factible=0;
1741
1742             int mejora_current_iteration = 0;
1743             uint64_t cant_iters_bqlocal = 0;
1744             double promedio-parcial_bqlocal = 0;
1745             double promedio_bqlocal = 0;

```

```

1735     do{
1736         promedio-parcial-bqlocal = 0;
1737         MEDIR.TIEMPO.PROMEDIO(
1738             mejora-current-iteration = g.busqueda_local(modos_busqueda_local);
1739             , 1, &promedio-parcial-bqlocal);
1740         cant-iters-bqlocal++;
1741         promedio-bqlocal += promedio-parcial-bqlocal;
1742     }while(mejora-current-iteration > 0);
1743     promedio-bqlocal = promedio-bqlocal /((double) cant-iters-bqlocal);
1744
1745     //el tiempo de esta iteracion es la greedy randomized + bqlocal sobre esa sol
1746     //inicial
1747     promedio += tiempo_golosa-randomized;
1748     promedio += promedio-bqlocal;
1749
1750     //en este punto la bqlocal mejoro todo lo que pudo la sol. inicial obtenida con
1751     //la randomized greedy
1752     //me fijo si esta solucion es mejor que la que tenia guardada, de ser asi
1753     //actualizo el maximo y guardo la sol actual como la nueva mejor.
1754     camino = g.obtener_camino_solucion();
1755     costo_t costo_solucion_actual = camino.obtener_costo_total_w2_camino();
1756
1757     //almaceno el costo total en esta iteracion
1758     if(costo_solucion_actual < costo_mejor_solucion){
1759         //guardamos que en esta iteracion se encontro una mejora
1760         if(cant-iters>0){//sino la primera vez pone infinito(costo_mejor_solucion
1761             //es infinito al inicializar)
1762             mejora-iters-grasp.push_back(make_pair(cant-iters , costo_mejor_solucion
1763                 - costo_solucion_actual));
1764         }
1765         costo_mejor_solucion = costo_solucion_actual;
1766         mejor_solucion = g.obtener_camino_solucion();
1767         //reseteo el contador
1768         cant-iters-sin-mejora = 0;
1769     }else{
1770         //una iteracion consecutiva mas sin mejora
1771         //guardamos que en esta iteracion se encontro una mejora
1772         if(cant-iters>0){//sino la primera vez pone infinito(costo_mejor_solucion
1773             //es infinito al inicializar)
1774             mejora-iters-grasp.push_back(make_pair(cant-iters , 0));
1775         }
1776         cant-iters-sin-mejora++;
1777     }
1778     //costo de la mejor sol guardada
1779     costo_t costo_w1_mejor_solucion = mejor_solucion.obtener_costo_total_w1_camino
1780     ();
1781     costo_t costo_w2_mejor_solucion = mejor_solucion.obtener_costo_total_w2_camino
1782     ();
1783     costo_camino-en-iteraciones.push_back(make_pair(costo_w1_mejor_solucion ,
1784         costo_w2_mejor_solucion));
1785     cant-iters++;
1786 }else{
1787     cant-iters-sin-sol-greedy-rand-factible++;
1788     //si ya no tengo mas chances, modifco los parametros de la metaheuristica
1789     if(cant-iters-sin-sol-greedy-rand-factible >= RAND.GREEDY_BAD_ITSERS_LIMIT){
1790         //si es rcl por cantidad bajo el parametro beta
1791         if(modos_golosa == RCLPOR.CANTIDAD){
1792             if(parametro_beta>2){
1793                 parametro_beta--;
1794             }else{
1795                 //cout << "[GRASP] Golosa deterministica seteada." << endl;
1796                 parametro_beta=1;
1797                 modos_golosa = RCL.DETERMINISTICO;
1798             }
1799         }else if(modos_golosa == RCLPOR.VALOR){
1800             //si es rcl por valor, seteo greedy deterministica
1801             //es muy delicado ajustar el porcentaje adaptativamente!
1802             parametro_beta = 0; //es lo mismo que poner golosa deterministico!
1803             modos_golosa = RCL.DETERMINISTICO;
1804         }
1805         //cerr << "[GRASP] Greedy randomized dio sol mala. Cambiando parametro_beta
1806         : " << parametro_beta << endl;
1807         cant-iters-sin-sol-greedy-rand-factible = 0;
1808     }
1809 }
1810 }else{

```

```

1802         break;//no hay solucion factible
1803     }
1804     if(criterio_terminacion == CRT_K_ITEERS_LIMIT_REACHED){
1805         condicion_terminacion = (cant_iters < ITEERS_LIMIT);
1806     }else if(criterio_terminacion == CRT_K_ITEERS_SIN_MEJORA){
1807         condicion_terminacion = (cant_iters_sin_mejora < ITEERS_LIMIT);
1808     }
1809 }while(condicion_terminacion);
1810 promedio = promedio / (double) cant_iters;
1811
1812 //imprimo mediciones en stderr
1813 if(g.hay_solucion()){
1814     cerr << g.obtener_cantidad_nodos() << " " << g.obtener_cantidad_aristas() << " " <<
1815         cant_iters << " " << promedio;
1816     //mejora en iteraciones
1817     ofstream evolucion_iteraciones;
1818     evolucion_iteraciones.open(FILE_ITEERS_MEJORA);
1819     for(auto element : mejora_iters_grasp){
1820         evolucion_iteraciones << element.first << " " << element.second << endl;
1821     }
1822     evolucion_iteraciones.close();
1823
1824     //evolucion absoluta
1825     evolucion_iteraciones.open(FILE_ITEERS_COSTOS_ABSOLUTOS);
1826     for(uint i=1;i<=costo_camino_en_iteraciones.size();i++){
1827         //imprimr <iteracion> <costo_w2> <costo_w1>
1828         evolucion_iteraciones << i << " " << costo_camino_en_iteraciones[i-1].second << " "
1829             << costo_camino_en_iteraciones[i-1].first << endl;
1830     }
1831     evolucion_iteraciones.close();
1832
1833     costo_t costo_w2_inicial = costo_camino_en_iteraciones.front().second;
1834     costo_t costo_w2_final = costo_camino_en_iteraciones.back().second;
1835     costo_t mejora_total_costo_w2 = costo_w2_inicial - costo_w2_final;
1836
1837     evolucion_iteraciones.open(FILE_ITEERS_COSTOS_ABSOLUTOS_STATISTICS);
1838     evolucion_iteraciones << costo_w2_inicial << " " << costo_w2_final << " " <<
1839         mejora_total_costo_w2;
1840     evolucion_iteraciones.close();
1841 }
1842 g.serialize(cout, FORMATO_1_N_CLOSED);
1843 }
1844 //

```

```

1845 void ejecutar_greedy(Grafo &g){
1846     int limit_w1 = g.obtener_limite_w1();
1847     nodo_t nodo_src = g.obtener_nodo_origen();
1848     nodo_t nodo_dst = g.obtener_nodo_destino();
1849 #ifdef DEBUG_MESSAGES_ON
1850     //g.imprimir_lista_adyacencia(cout);
1851     //g.imprimir_matriz_adyacencia(cout);
1852 #endif
1853 #ifdef DEBUG_MESSAGES_ON
1854     cout << "Se requiere un camino entre (" << nodo_src << ")_y_(" << nodo_dst << ")_que no_
1855         exceda el costo " << limit_w1 << endl;
1856 #endif
1857     double promedio_medicion = 0;
1858     Camino camino = g.obtener_camino_vacio();
1859     MEDIR_TIEMPO_PROMEDIO(
1860         camino = g.obtener_solucion_golosa();
1861         , CANT_ITEERS_MEDICION, &promedio_medicion);
1862
1863     g.establecer_camino_solucion(camino);
1864     if(g.hay_solucion()){
1865         cerr << g.obtener_cantidad_nodos() << " " << g.obtener_cantidad_aristas() << " " <<
1866             CANT_ITEERS_MEDICION << " " << promedio_medicion;
1867         #ifdef DEBUG_MESSAGES_ON
1868             cout << endl << "Solucion obtenida con golosa" << endl;
1869             camino.imprimir_camino(cout);
1870         #endif
1871     }

```

```

1872     g.serialize(cout, FORMATO_1N_CLOSED);
1873 }
1874
1875 //

```

```

1876
1877
1878 #include "grafo.h"
1879 #include "timing.h"
1880 #include <fstream>
1881 #define FILE_ITSERS_MEJORA "evolucion_iteraciones.txt"
1882 #define FILE_ITSERS_COSTOS_ABSOLUTOS "costos_absolutos_iteraciones_bqlocal.txt"
1883 #define FILE_ITSERS_COSTOS_ABSOLUTOS_STATISTICS "costos_absolutos_iteraciones_bqlocal_analisis.txt"
1884
1885 void ejecutar_búsqueda_local(Grafo &g);
1886
1887 // ----- Main -----
1888 int main(int argc, char **argv){
1889     list<Grafo> instancias = Grafo::parsear_varias_instancias(FORMATO_1N_CLOSED);
1890     uint64_t instance_number = 1;
1891     for(Grafo &g : instancias){
1892         #ifdef DEBUG_MESSAGES_ON
1893             cout << endl << endl << "Aplicando_búsqueda_local_a_la_" << instance_number << "ésima_instancia_de_input..." << endl;
1894         #endif
1895         ejecutar_búsqueda_local(g);
1896         instance_number++;
1897     }
1898     return 0;
1899 }
1900
1901 void ejecutar_búsqueda_local(Grafo &g){
1902     // #ifdef DEBUG_MESSAGES_ON
1903     //     g.imprimir_lista_adyacencia(cout);
1904     //     //g.imprimir_matriz_adyacencia(cout);
1905     // #endif
1906
1907     costo_t limit_w1 = g.obtener_limite_w1();
1908     nodo_t nodo_src = g.obtener_nodo_origen();
1909     nodo_t nodo_dst = g.obtener_nodo_destino();
1910
1911     //----- Busco solución inicial -----
1912
1913     vector<costo_t> costo_minimo;
1914     vector<nodo_t> predecesor;
1915     g.dijkstra(nodo_src, COSTO_W1, costo_minimo, predecesor);
1916
1917     costo_t costo_src_dst = costo_minimo[nodo_dst]; // costo(src, dst)
1918
1919     //----- Valido la factibilidad de la solución -----
1920     #ifdef DEBUG_MESSAGES_ON
1921         cout << "Se_requiere_un_camino_entre_(" << nodo_src << ")_y_(" << nodo_dst << ")_que_no_exceda_el_costo_" << limit_w1;
1922     #endif
1923     if(costo_src_dst == costo_infinito){
1924         cerr << "No_existe_solución_factible._No_existe_camino_entre_origen(" << nodo_src << ")_y_destino(" << nodo_dst << ")_" << endl;
1925         g.establecer_se_encontro_solucion(false);
1926     } else if(costo_src_dst > limit_w1){
1927         cerr << "No_existe_solución_factible._El_camino_minimo_respecto_a_w1_de_origen(" << nodo_src << ")_a_destino(" << nodo_dst << ")_es_de_costo_" << costo_src_dst << endl;
1928         g.establecer_se_encontro_solucion(false);
1929     } else{
1930         // armar camino entre origen y destino y lo establezco como sol inicial
1931         Camino c = g.obtener_camino_vacio();
1932         nodo_t nodo = nodo_dst;
1933         do{
1934             //cout << nodo << " ";
1935             c.agregar_nodo_adelante(nodo);
1936             nodo = predecesor[nodo];
1937         } while(nodo != predecesor_nulo);
1938         g.establecer_camino_solucion(c);
1939
1940         // Camino c = g.obtener_solucion_golosa();

```



```

1941 //g.establecer_camino_solucion(c);
1942
1943 //imprimo sol inicial.
1944 #ifdef DEBUG_MESSAGES_ON
1945     cout << " _Costo_minimo_obtenido:_ " << c.obtener_costo_total_w1_camino();
1946     cout << "...Ok!!" << endl;
1947     cout << "Camino_inicial:_";
1948     c.imprimir_camino(cout);
1949     cout << endl;
1950 #endif
1951
1952 //----- Comienzo la busqueda local -----
1953 //typedef enum tipo_ejecucion_bqlocal_t {BQLSUBDIVIDIR_PARES,
1954 //    BQLCONTRAER_TRIPLAS_A_PARES, BQLMEJORAR_CONEXION_TRIPLAS, BQLCOMBINAR}
1955 //    tipo_ejecucion_bqlocal_t;
1956 tipo_ejecucion_bqlocal_t tipo_ejecucion = BQLCOMBINAR;
1957
1958 //hago iteraciones de busqueda local hasta que no haya mejora(la funcion devuelve true
1959 //    si hubo mejora, false sino)
1960 uint64_t cant_iters = 0;
1961 double promedio_parcial = 0;
1962 double promedio = 0;
1963 int mejora_current_iteration = 0;
1964 vector<costo_t> mejora_en_iteraciones;
1965 vector<pair<costo_t, costo_t>> costo_camino_en_iteraciones; //costos w1, w2
1966 costo_t costo_w1_current_iteration = c.obtener_costo_total_w1_camino();
1967 costo_t costo_w2_current_iteration = c.obtener_costo_total_w2_camino();
1968
1969 //ponemos el costo inicial de la iteracion 0 del camino
1970 mejora_en_iteraciones.push_back(0); //mejora 0 en iteracion 0
1971 costo_camino_en_iteraciones.push_back(make_pair(costo_w1_current_iteration,
1972     costo_w2_current_iteration));
1973 do{
1974     promedio_parcial = 0;
1975     MEDIR_TIEMPO_PROMEDIO(
1976         mejora_current_iteration = g.busqueda_local(tipo_ejecucion);
1977         , 1, &promedio_parcial);
1978     cant_iters++;
1979     promedio += promedio_parcial;
1980     costo_w1_current_iteration = g.obtener_costo_actual_w1_solucion();
1981     costo_w2_current_iteration = g.obtener_costo_actual_w2_solucion();
1982     costo_camino_en_iteraciones.push_back(make_pair(costo_w1_current_iteration,
1983         costo_w2_current_iteration));
1984
1985     mejora_en_iteraciones.push_back(mejora_current_iteration);
1986 }while(mejora_current_iteration > 0);
1987 promedio = promedio /((double) cant_iters);
1988
1989 #ifdef DEBUG_MESSAGES_ON
1990     switch(tipo_ejecucion){
1991     case BQLSUBDIVIDIR_PARES:
1992         cout << "Finalizo_la_busqueda_local_insertando_entre_pares_porque_no_se_
1993             obtuvieron_nuevas_mejoras." << endl;
1994         break;
1995     case BQLCONTRAER_TRIPLAS_A_PARES:
1996         cout << "Finalizo_la_busqueda_local_salteando_entre_triplos_porque_no_se_
1997             obtuvieron_nuevas_mejoras." << endl;
1998         break;
1999     case BQLMEJORAR_CONEXION_TRIPLAS:
2000         cout << "Finalizo_la_busqueda_local_reemplazando_entre_triplos_porque_no_se_
2001             obtuvieron_nuevas_mejoras." << endl;
2002         break;
2003     case BQLCOMBINAR:
2004         cout << "Finalizo_la_busqueda_local_combinada_porque_no_se_obtuvieron_
2005             nuevas_mejoras." << endl;
2006         break;
2007     }
2008 #endif
2009 g.establecer_se_encontro_solucion(true);
2010 cerr << g.obtener_cantidad_nodos() << " _" << g.obtener_cantidad_aristas() << " _" <<
2011     cant_iters << " _" << promedio;
2012
2013 //mejora en iteraciones
2014 ofstream evolucion_iteraciones;
2015 evolucion_iteraciones.open(FILE_ITERS_MEJORA);
2016 for(uint i=1; i<=mejora_en_iteraciones.size(); i++){
2017     evolucion_iteraciones << i << " _" << mejora_en_iteraciones[i-1] << endl;

```



```

2008     }
2009     evolucion_iteraciones.close();
2010
2011     evolucion_iteraciones.open(FILE_ITEERS_COSTOS_ABSOLUTOS);
2012     for(uint i=1;i<=costo_camino_en_iteraciones.size();i++){
2013         //imprimr <iteracion> <costo-w2> <costo-w1>
2014         evolucion_iteraciones << i << " " << costo_camino_en_iteraciones[i-1].second << " "
                << costo_camino_en_iteraciones[i-1].first << endl;
2015     }
2016     evolucion_iteraciones.close();
2017
2018     costo_t costo_w2_inicial = costo_camino_en_iteraciones.front().second;
2019     costo_t costo_w2_final = costo_camino_en_iteraciones.back().second;
2020     costo_t mejora_total_costo_w2 = costo_w2_inicial - costo_w2_final;
2021
2022     evolucion_iteraciones.open(FILE_ITEERS_COSTOS_ABSOLUTOS_STATISTICS);
2023     evolucion_iteraciones << costo_w2_inicial << " " << costo_w2_final << " " <<
            mejora_total_costo_w2;
2024     evolucion_iteraciones.close();
2025 }
2026
2027 g.serialize(cout, FORMATO_1_N_CLOSED);
2028 }
2029
2030 //

```
