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

Resumen

El objetivo de este documento es describir el problema presentado, y las diversas soluciones que se fueron realizando utilizando distintas tecnicas de programación, para la solucion exacta, o diversas heurísticas para resolver el problema de forma polinomial, dado que no se conoce algoritmo de solucion 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 Minimo (CACM), el cual consiste en encontrar un camino P entre u, v tal que el costo del camino respecto a la funcion w_2 sea minimo entre todos los caminos con $w_1 \leq K$ y valga la condicion 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 practico y en el fueron implementados los siguientes algoritmos:

- Solución exacta (C/C++)
- Heurística constructiva golosa (C++)
- Heurística de busqueda 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 teorica sobre la complejidad temporal, y se brindan ejemplos de funcionamiento. Ademas se realizaron benchmarks de performance sobre diferentes conjuntos de instancias distintas de entrada que afirman la complejidad teorica y son presentadas con graficos que acompanãn dichos experimentos. Para las heurísticas golosas y de busqueda local se detallan familias o instancias malas de entradas donde la solucion provista no es la optima, se realizaron diversos analisis de optimalidad y variacion de parametros de las heuristicas detallados en cada seccion de cada heuristica.

Para la metaheuristica `GRASP` se ajustaron los parametros para obtener cierto tradeoff, y fueron fijadas en esos parametros para la seccion de experimentacion general.

Finalmente se realizó una experimentacion general con respecto al rendimiento de las heuristicas respecto a exacta, las heuristicas solas entre ellas para entradas mas grandes de las que soporta el algoritmo exacto en la practica, los experimentos miden y comparan la cercania de las soluciones respecto a la solucion optima con diversos estimadores estadisticos, asi como tambien el tiempo insumido en obtener las soluciones para realizar un tradeoff entre porcentaje de efectividad de la heuristica 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 $f : E \to \mathbb{R}_+$ de un camino P = $< v_1, \dots, v_{k-1}, v_k >$ de la siguiente forma:

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

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:

- $f : E \to \mathbb{R}_+$: funcion asociada a la distancia entre v_{src} y v_{dst}.
- $g : E \to \mathbb{R}_+$: 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 caracteristicas, 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).

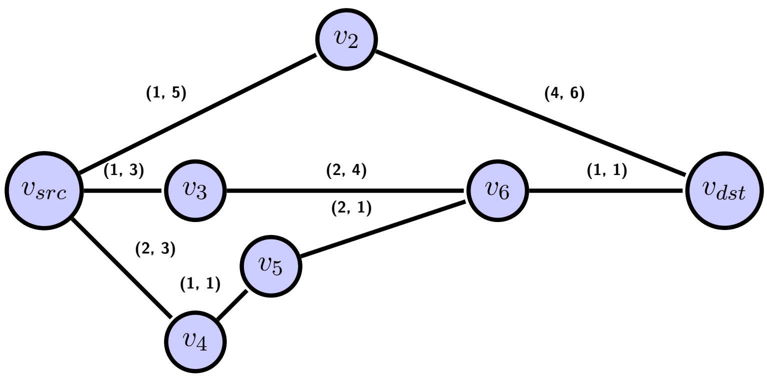


Figura 1: Ejemplo de conexion entre ciudades.

Veamos 3 posibles caminos y analicemos cada uno:

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

- 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, ahi 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 viajantes. 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 conexa 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:

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

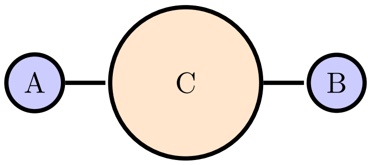


Figura 2: Nodos A,B no adyacentes entre si, conectados a componente conexa C

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 poblema 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 limite K, y un camino encontrado hasta el momento, que va desde el nodo de origen del problema (v), al nodo de orgen pasado a la función (u).

En cada llamada recursiva, la función recorre todos los adyacentes a u, y por cada uno (nombremoslo 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 hubiesenn, 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 ésto 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 aca 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 *verdarero* si encontró un camino de u a w, $falso$ si no encontró camino.

```
solucion_mejor ← []

procedure CACM_EXACTO(Grafo: grafo, Nodo: u, Nodo: w, Real: K, Camino: solucion)→ bool
    distancias ← DIJSKTRA_EN_W1(grafo, u)
    if distancias[w] > K then
        return falso
    else
        distancias_w1 ← DIJSKTRA_EN_W1(grafo, w)
        distancias_w2 ← DIJSKTRA_EN_W2(grafo, w)
        return ← CACM_EXACTO_RECURSIVO(grafo, u, w, K, distancias_w1, distancias_w2, solucion)
    end if
end procedure

procedure CACM_EXACTO_RECURSIVO(Grafo: grafo, Nodo: u, Nodo: w, Real: K, Real: distan-
cias_w1[], Real: distancias_w2, Camino: solucion)→ bool
    if u = w then
        if mejor_solucion = [] ∨ ω2(solucion) < ω2(solucion_mejor) then
            mejor_solucion ← solucion_nueva
            return verdadero
        else
            return falso
        end if
    end if
    solucion_nueva ← []
    solucionado ← false
    MARCARVISITADO(grafo, u, true)
    for z ← ADYACENTES(grafo, u) do
        if ¬ ESTAVISITADO(grafo, z) then
            peso1 ← ω1(u, z)
            peso2 ← ω2(u, z)
            if distancias_w1[z] + peso1 <= K ∧ (solucion_mejor = [] ∨ distancias_w2[z] + peso2 +
ω2(solucion) < ω2(solucion_mejor) then
                QUITAR_ARISTA(grafo, u, z)
                solucion_nueva gets solucion +[z]
                encontre ← CACM_EXACTO_RECURSIVO(grafo, z, w, K − peso1, solucion_nueva)
                if encontre then
                    solucionado ← true
                end if
                AGREGAR_ARISTA(grafo, u, z, peso1, peso2)
            end if
        end if
    end for
    MARCARVISITADO(grafo, u, false)
    if solucionado then
        solucion ← solucion_mejor
    end if
    return solucionado
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 esta 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.

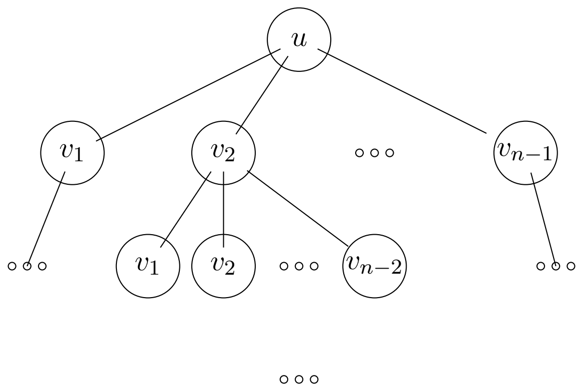


Figura 3: Árbol de llamadas recursivas. u es el nodo de origen, y v_i es el vecino i disponible para avanzar en la recursión del padre del nodo en el árbol

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 esta 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 $d(v) <= n-1$

$$T(1) = 1$$

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

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 equación 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'.

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

Si se sigue desarrollando la equación, 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

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

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 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.

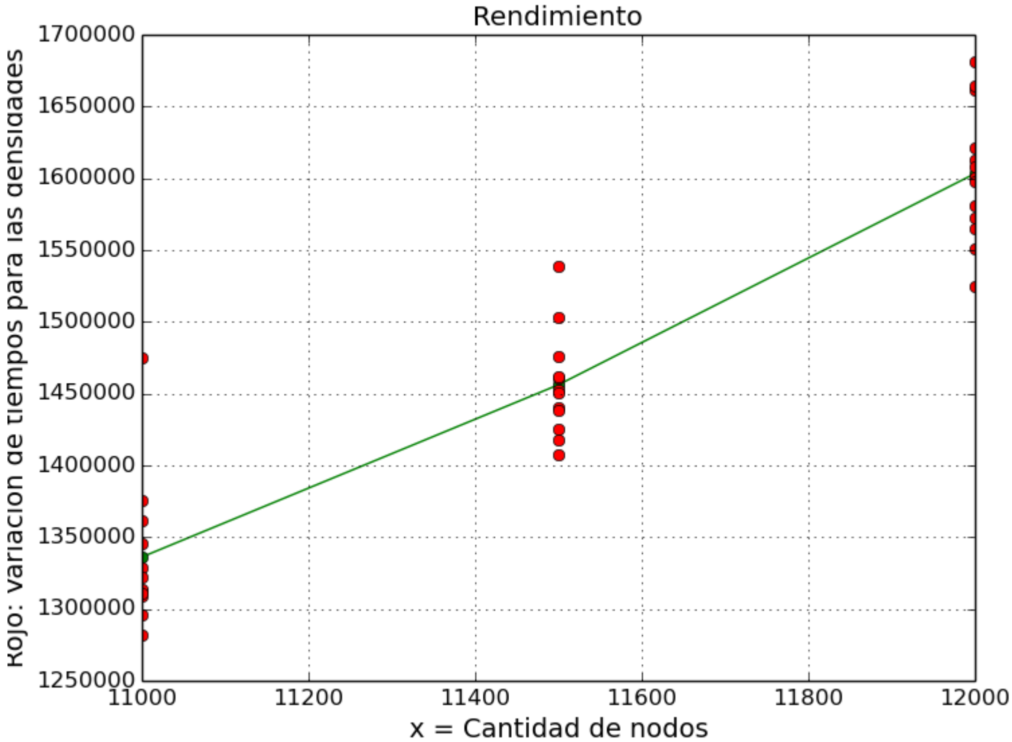
En esta primera sección dividiremos las funciones por polinomios n, n^2, n^3 y n^4 con la mayor cantidad de nodos posible e intentaremos ver que la función se mantiene creciente. Dadas las limitaciones de nuestra computadora de benchmarking, el análisis de división por factorial se hará aparte con menos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

2.4.1. Rendimiento para grafos con densidad lineal de aristas

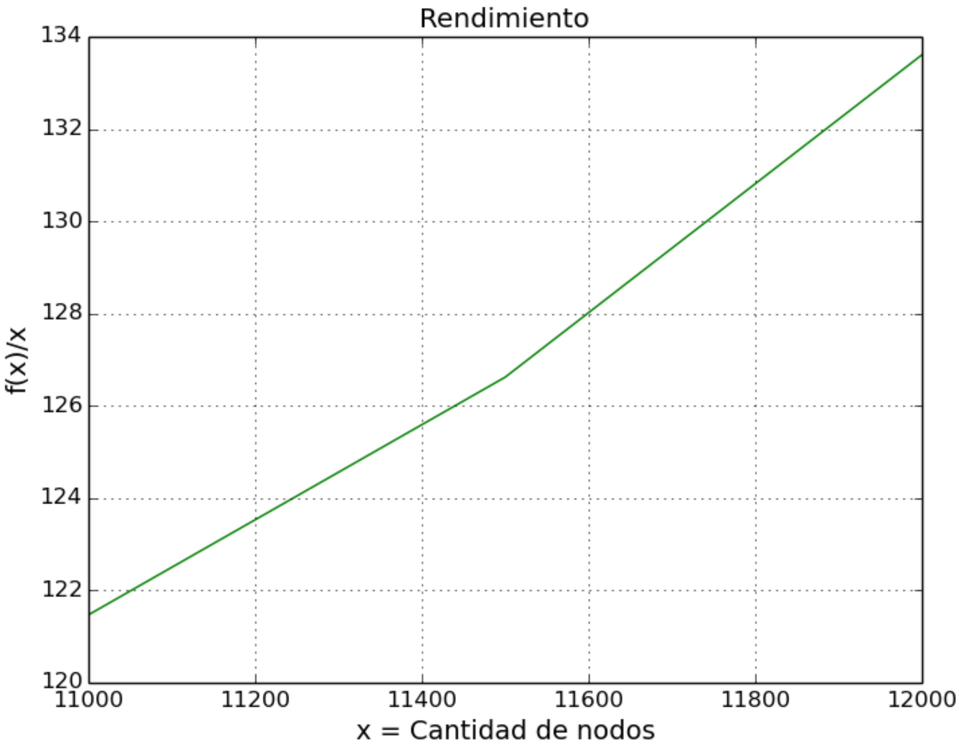
- cant nodos min = 10000
- cant nodos max = 12000
- peso maximo w1 = 250
- peso maximo w2 = 400
- step nodos = 500
- step aristas = 1000
- aristas minimas = $(n-1)$
- aristas maximas = $(10 * n)$

Tiempo de ejecución en microsegundos para esta familia

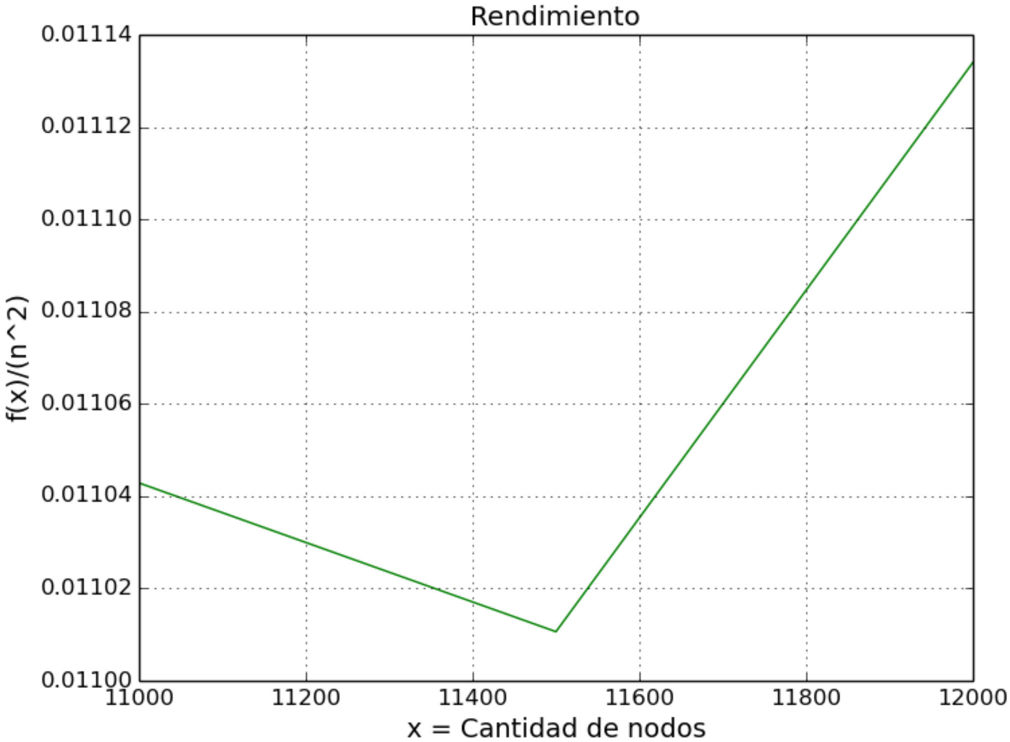
$y = f(x)$


Rendimiento
Rojo: variacion de tiempos para las densidades
x = Cantidad de nodos
1700000
1650000
1600000
1550000
1500000
1450000
1400000
1350000
1300000
1250000
11000
11200
11400
11600
11800
12000

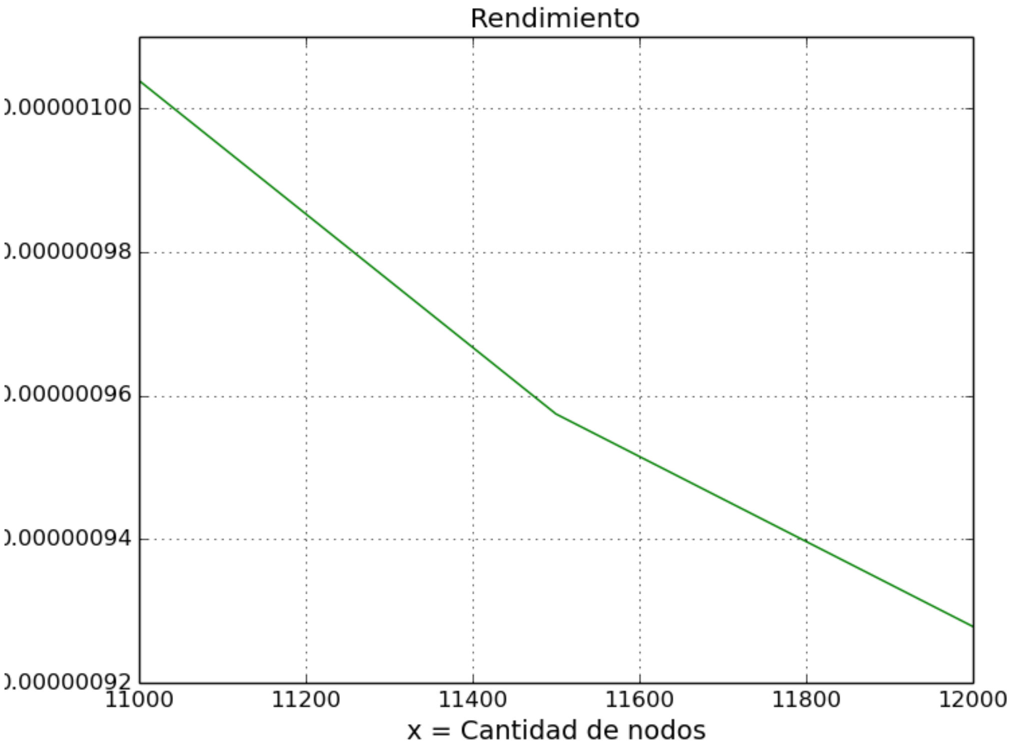
$y = f(x)/x$


Rendimiento
f(x)/x
x = Cantidad de nodos
134
132
130
128
126
124
122
120
11000
11200
11400
11600
11800
12000

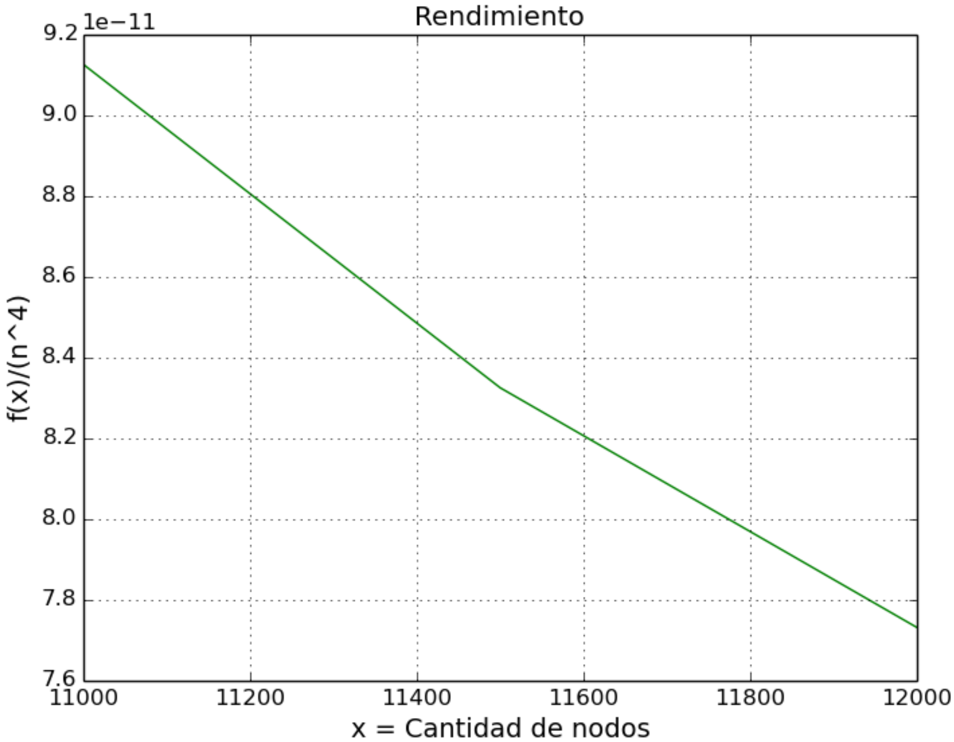
$y = f(x)/x^2$


Rendimiento
f(x)/(n^2)
0.01114
0.01112
0.01110
0.01108
0.01106
0.01104
0.01102
0.01100
11000
11200
11400
11600
11800
12000
x = Cantidad de nodos

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


Rendimiento
).00000100
).00000098
).00000096
).00000094
).00000092
11000
11200
11400
11600
11800
12000
x = Cantidad de nodos

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

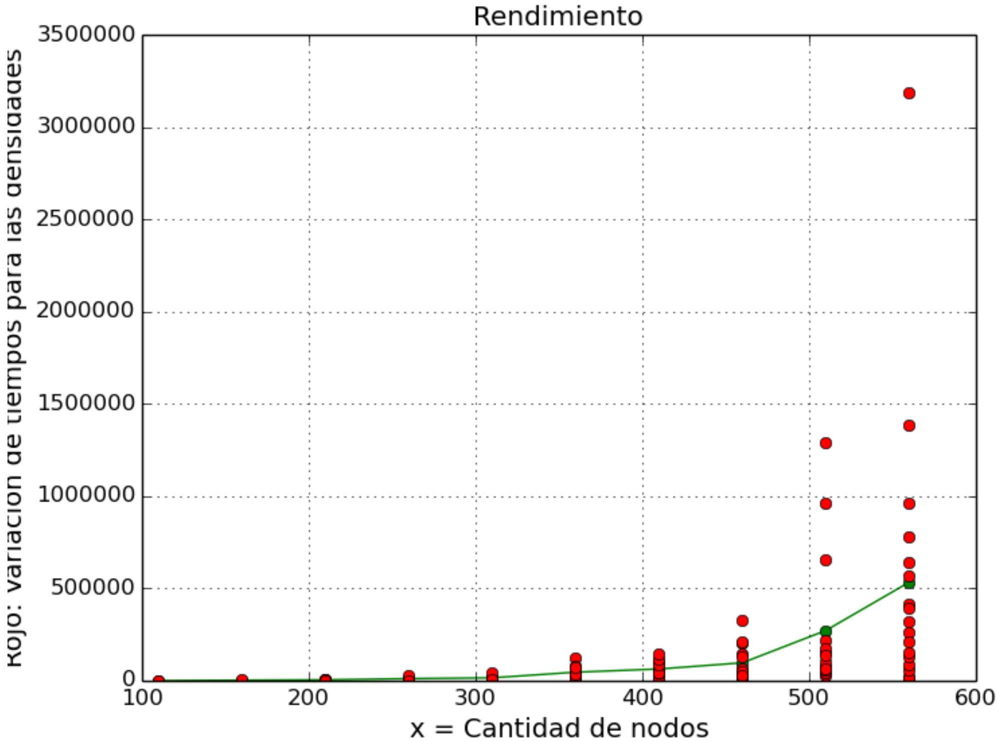


2.4.2. Rendimiento para grafos con densidad cuadratica de aristas

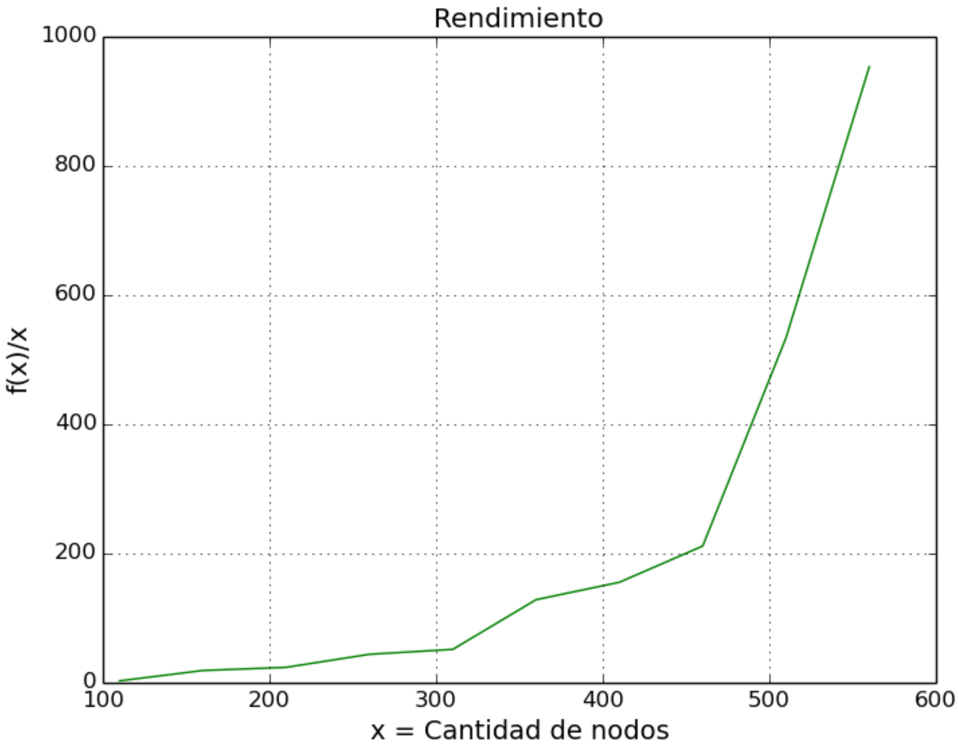
- cant nodos min = 10
- cant nodos max = 600
- peso maximo w1 = 250
- peso maximo w2 = 400
- step nodos = 50
- step aristas = 7000
- aristas minimas = $(n * (n - 1))/10$
- aristas maximas = $(n * (n - 1))/2$

Tiempo de ejecución en microsegundos para esta familia

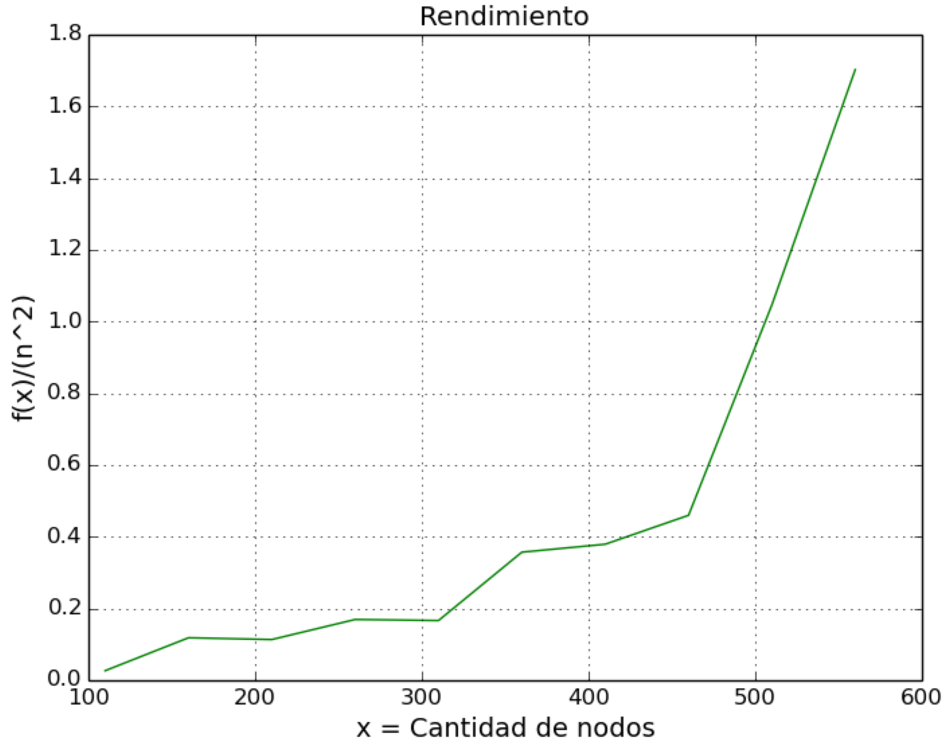
$y = f(x)$


Rendimiento
Rojo: variación de tiempos para las densidades
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
100
200
300
400
500
600
x = Cantidad de nodos

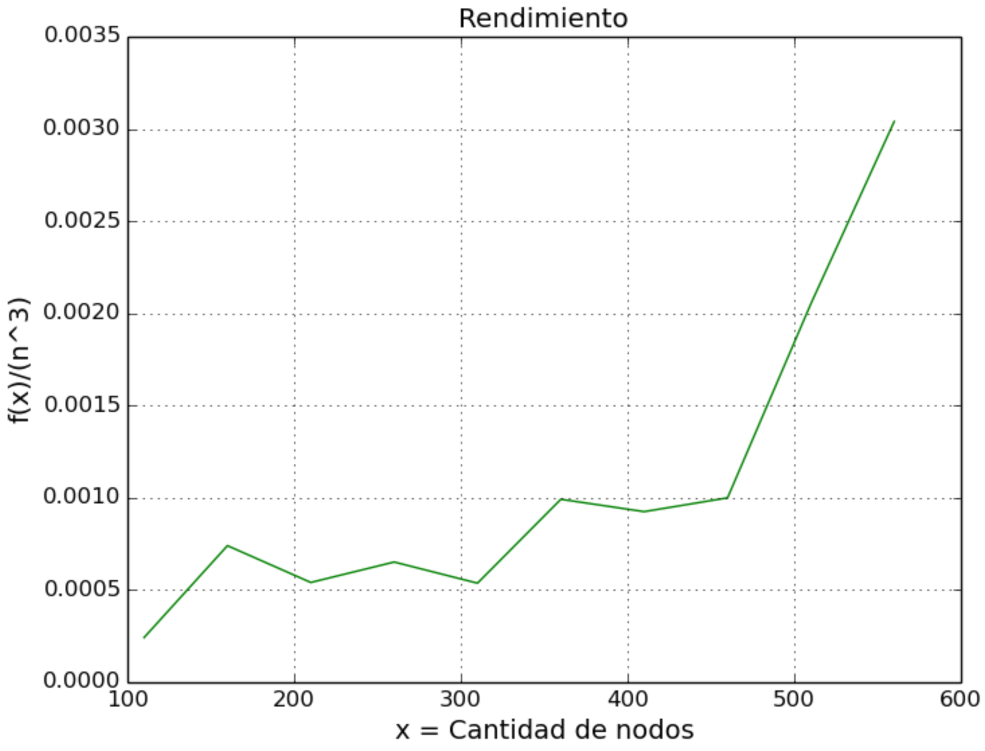
$y = f(x)/x$


Rendimiento
f(x)/x
1000
800
600
400
200
0
100
200
300
400
500
600
x = Cantidad de nodos

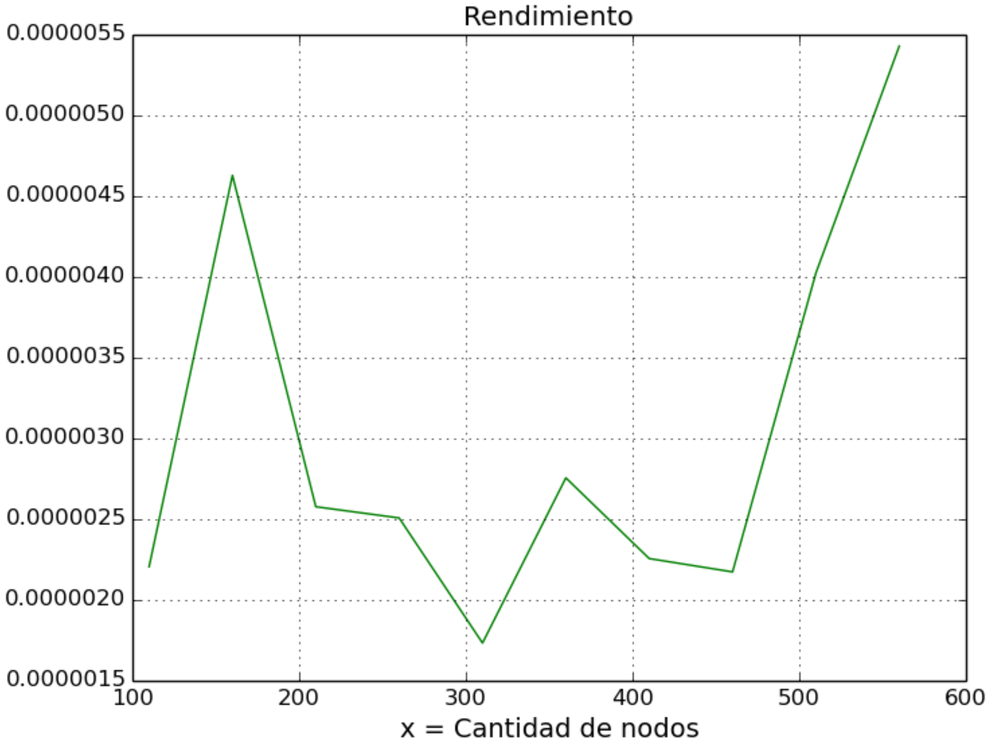
$y = f(x)/x^2$


Rendimiento
f(x)/(n^2)
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
100
200
300
400
500
600
x = Cantidad de nodos

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


Rendimiento
f(x)/(n^3)
0.0035
0.0030
0.0025
0.0020
0.0015
0.0010
0.0005
0.0000
100
200
300
400
500
600
x = Cantidad de nodos

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



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 reduccón 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 maximo 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 funcion que expresa el tiempo en funcion de la entrada por varios polinomios de grados 1,2,3,4, el grafico sigue con un alto nivel de crecimiento para los ultimos valores del eje X, intentamos dividir la funcion por factorial y realizar el grafico $f(n/n!)$ pero dados los numeros muy grandes, el generador de graficos finalizaba con overflow sin producir el grafico. Asumimos - dada la naturaleza del problema (no se conoce algoritmo polinomial) y los resultados de la primer entrega sin podas- que cuando n tiende a infinito, la funcion se ajusta al analisis de complejidad teorica. No podemos testear con valores mas grandes de n pues nuestro hardware no lo permite. Asimismo, la calidad de las podas influyeron muchisimo a este resultado, en la entrega anterior, pudimos probar apenas con valores muy pequenõs de nodos($n < 17$) y observamos que la complejidad era factorial efectivamente.

3. Heurística Golosa

3.1. Explicacion 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

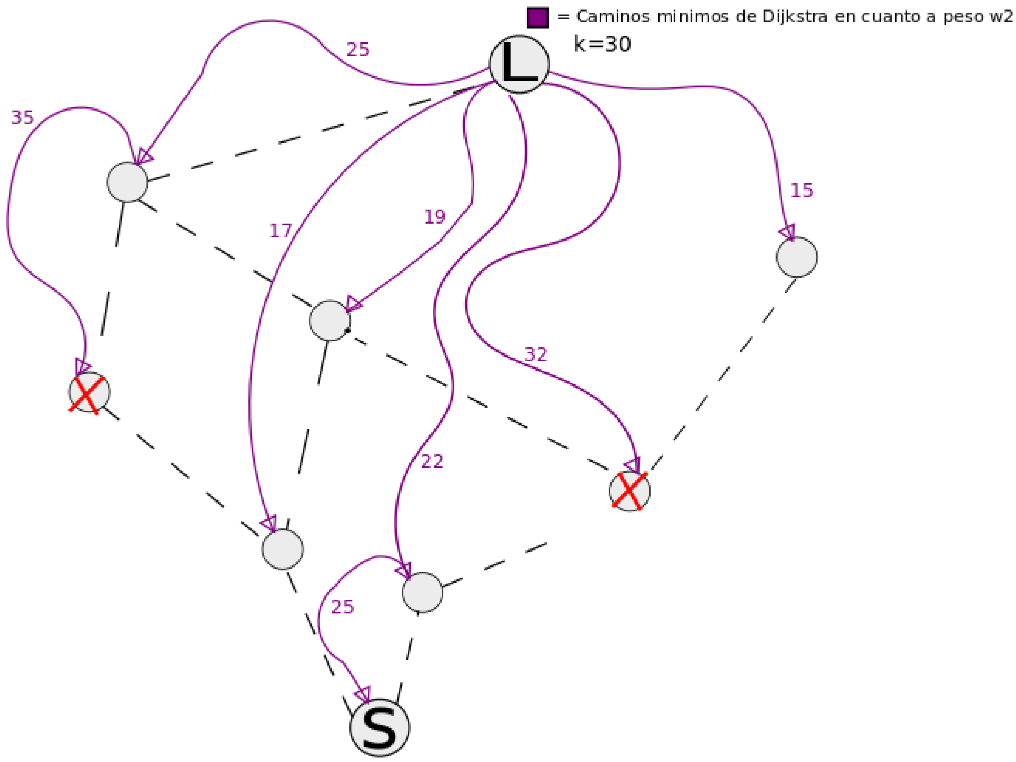
CostosW2, es análogo al vector distancias del algoritmo de Dijkstra, siendo w_2 la distancia a minimizar, guarda el costo w_2 del camino provisorio para cada nodo que va armando el algoritmo. Predecesores es análogo al vector predecesores de dicho algoritmo, indicando para cada nodo cuál es su predecesor 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 Dikjstra 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 las 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 proóxima 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 heurstica, 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 *precedesor* 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

procedure HEURÍSTICA GOLOSA(grafo g, nodo u, nodo v, cota K) -> camino
  $vector < int > \ costosw_1[n]$
  $vector < int > costosw_2[n]$
  $vector < int > predecesores[n]$
  $vector < int > \ costoRecorrido[n]$
  $camino\ c \leftarrow []$
  $colaPrioridad\ cola \leftarrow []$

$Dijkstra(g, v, costosw_1)$

for $i\ from\ 0\ to\ n-1$ **do**
  $costosw_2[i] \leftarrow \infty$
  $predecesores[i] \leftarrow NULL$
  $costoRecorrido[i] \leftarrow 0$
end for
$costosw_2[u] \leftarrow 0$

$cola.push(costosw_2, u)$
while $cola! = []$ **do**
  $par < costow_2, nodo > actual \leftarrow cola.pop$
  for $w\ :\ adyacentes(actual_2)$ **do**
    if $costoRecorrido[actual_2] + costow_1Arista(actual_2, w) + costosw_1(w) \leq K$ **then**
      if $costosw_2[w] > costosw_2[actual_2] + costow_2Arista(actual, w)$ **then**
        $costosw_2[w] \leftarrow costosw_2[actual_2] + costow_2Arista(actual, w)$
        $costoRecorrido[w] \leftarrow costoRecorrido[actual_2] + costow_1Arista(actual_2, w)$
        $predecesor[w] = actual$
        $cola.push(costow_2(w), w)$
      end if
    end if
  end for
end while
$c \leftarrow reconstruirCamino(predecesor, v)$
$return\ c$
end procedure

3.1.4. Complejidad

El algoritmo comienza declarando vectores de longitud de tamano n, un camino de tamano 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 $O((m+n)logn)$. 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 sos, 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 $O(logn)$. 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 $O(logn)$, con lo cual el ciclo for posee complejidad $O(mlogn)$.

Esto nos deja con una complejidad de: $O(n^2) + O((m+n)logn) + O(nlogn) + O(nlogm) = O(n^2 + mlogn)$.

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

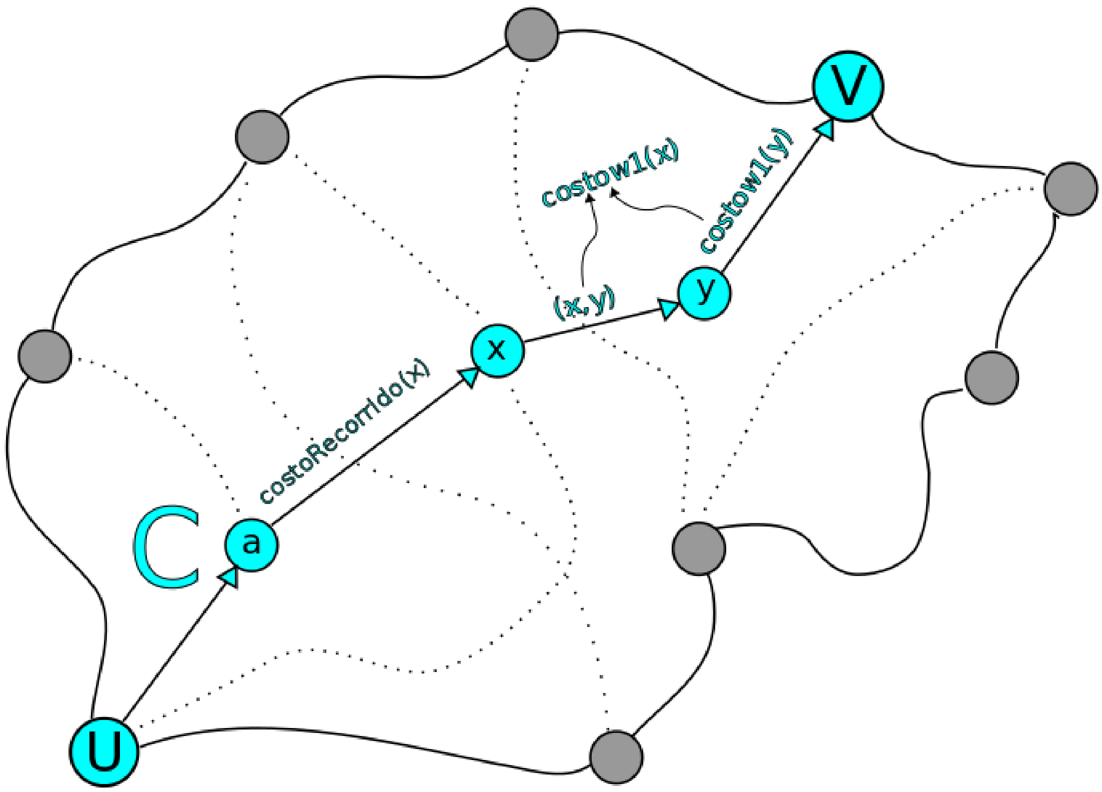
1. http://www.cplusplus.com/reference/set/set/
2. http://www.cplusplus.com/reference/set/set/begin/
3. http://www.cplusplus.com/reference/set/set/erase/

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 alguń vecino a tal que $costoRecorrido[a] + costow_1Arista(a, x) + costosw_1(x) \leq K$ 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 $costoRecorrido[u] + costow_1Arista(u, segundo)+costosw_1(segundo) \leq K$, al ser $costoRecorrido[u] = 0$ y $costow_1Arista(u, segundo)+costosw_1(segundo) \leq K$ necesariamente ya que es un subcamino de C, y que el problema de camino míno cumple el principio de optimalidad de Bellman, y el camino mínimo de u a v puede ser descompuesto como dos caminos $u - segundo$ y $segundo - v$ que a su vez son mínimos.

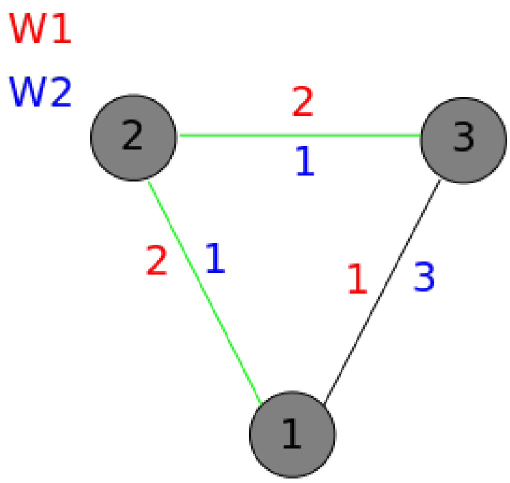


Ahora consideremos el nodo siguiente del camino, llamémosle y. y es el primer nodo del camino C para el cual el algoritmo no puede actualizarlo ni insertarlo en la cola. Pero sabemos que x es adyacente a y, y sabemos que para x, por haber sido actualizado por el algoritmo vale $costoRecorrido[x]+costosw_1(x) \leq K$. Y por no haber podido ser actualizado y desde x, vale $costoRecorrido[x]+costow_1Arista(x, y)+costosw_1(y) > K$. Pero esto nos deja en un absurdo, ya que $costosw_1(x) = costow_1Arista(x, y) + costosw_1(y)$ exactamente, ya que ambos nodos forman parte de C, y por principio de optimalidad al sere C el camino mínimo, el camino mńimo 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 ninguún camino factible en el grafo, o existe al menos uno y el algoritmo lo devuelve.

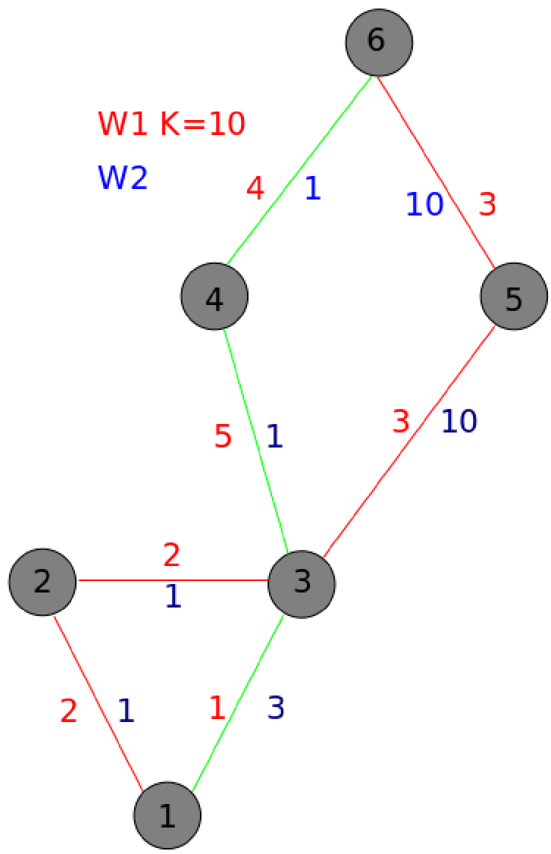
3.2. Nivel de optimalidad de las soluciones

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 $1- > 2- > 3$ (para un K no restrictivo), ya que la desición golosa toma el mínimo de la cola en cada iteración y actualiza el nodo 3 con el nodo 2 ya que el camino $1- > 2$ (en rojo) tiene menos costo w_2 que la arista $(1, 3)$ (en verde). Pero en este caso particular puede notarse como prima la desición 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 analísis de complejidad del algoritmo concluimos acotando la complejidad en $O(n^2 + mlogn)$. 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 $n^2 + nlogn$, 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 $n^2 + n^2logn$, lo cual es del orden $O(n^2logn)$. 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.

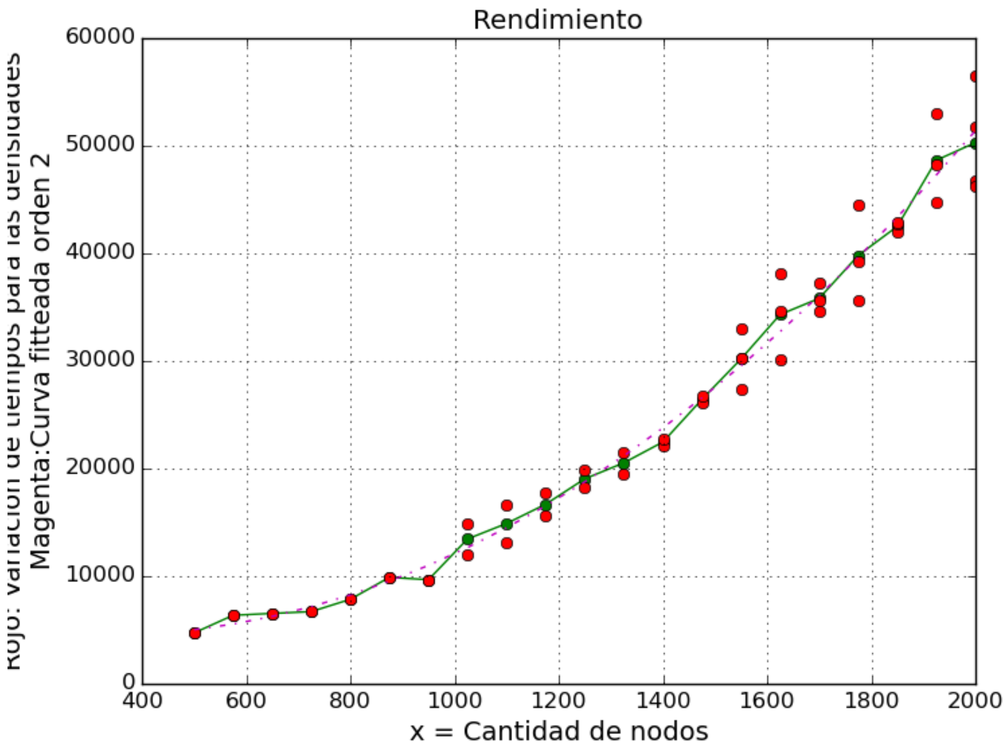
3.3.2. Rendimiento para grafos con densidad lineal de aristas

- cant nodos min = 50
- cant nodos max = 2000
- peso maximo w1 = 250

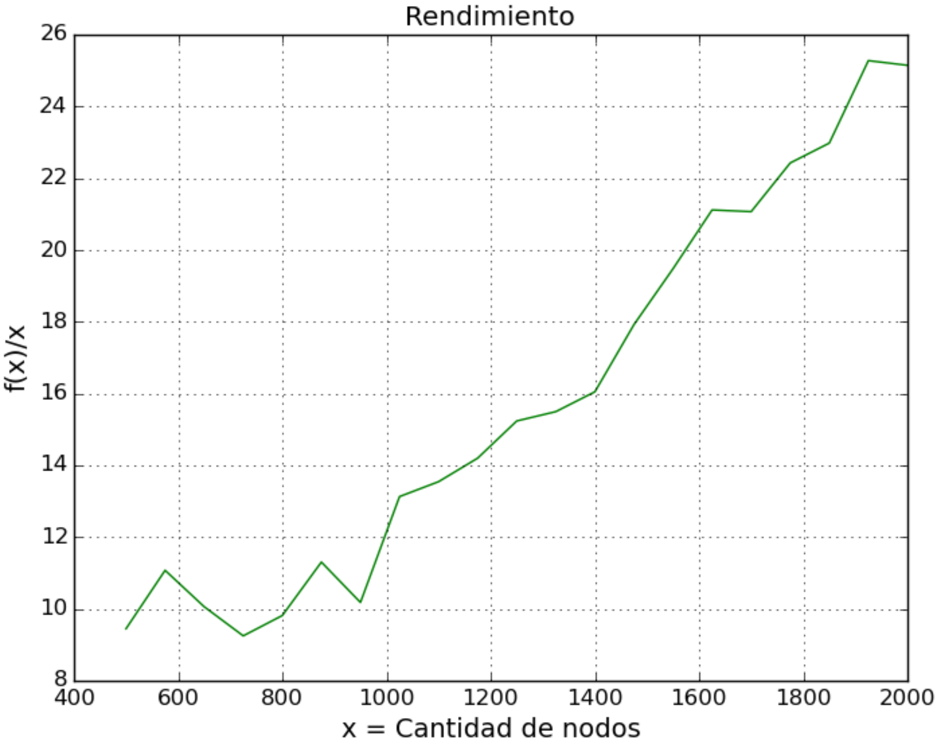
- peso maximo w2 = 400
- step nodos = 75
- step aristas = 4500
- aristas minimas = $n - 1$
- aristas maximas = $10 * n$

Tiempo de ejecución en microsegundos para esta familia

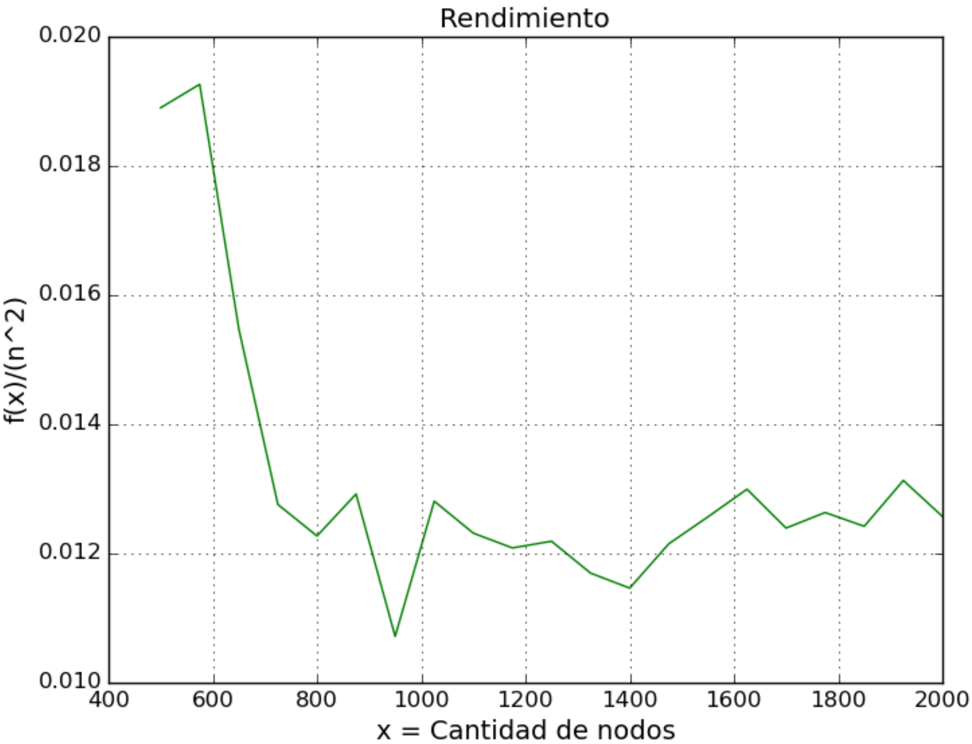
$y = f(x)$



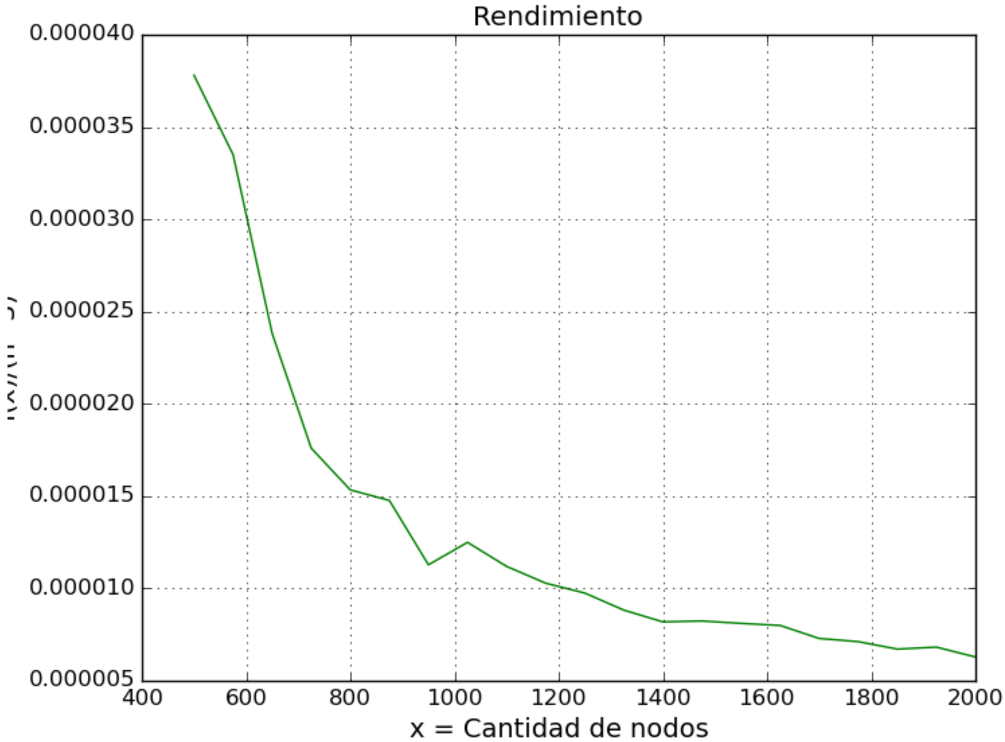
$y = f(x)/x$


Rendimiento
f(x)/x
x = Cantidad de nodos
26
24
22
20
18
16
14
12
10
8
400
600
800
1000
1200
1400
1600
1800
2000

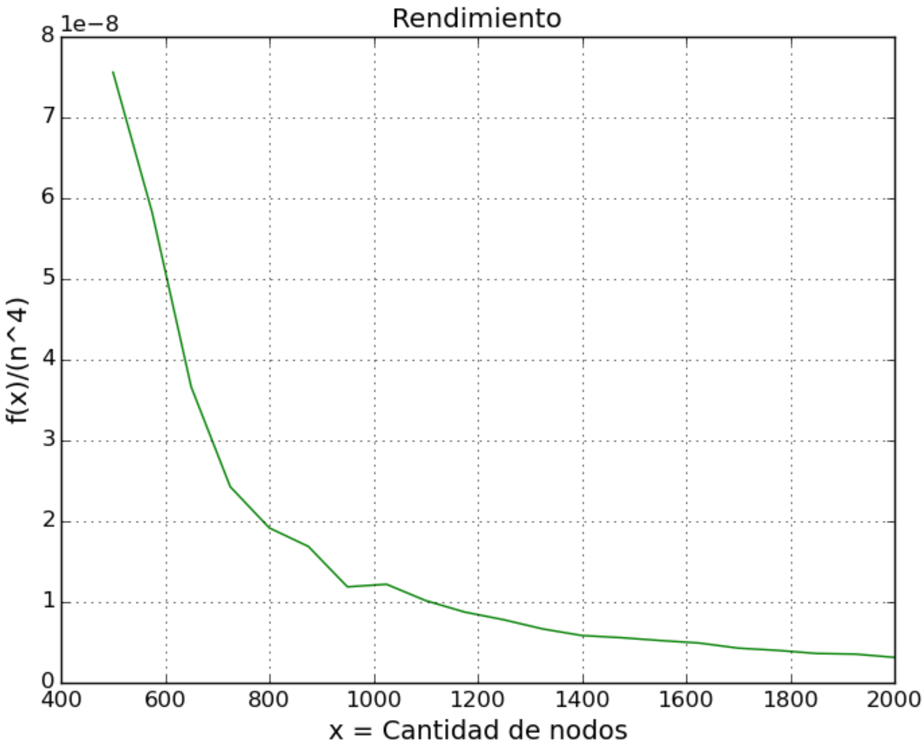
$$y = f(x)/x^2$$


Rendimiento
f(x)/(n^2)
x = Cantidad de nodos
0.020
0.018
0.016
0.014
0.012
0.010
400
600
800
1000
1200
1400
1600
1800
2000

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



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



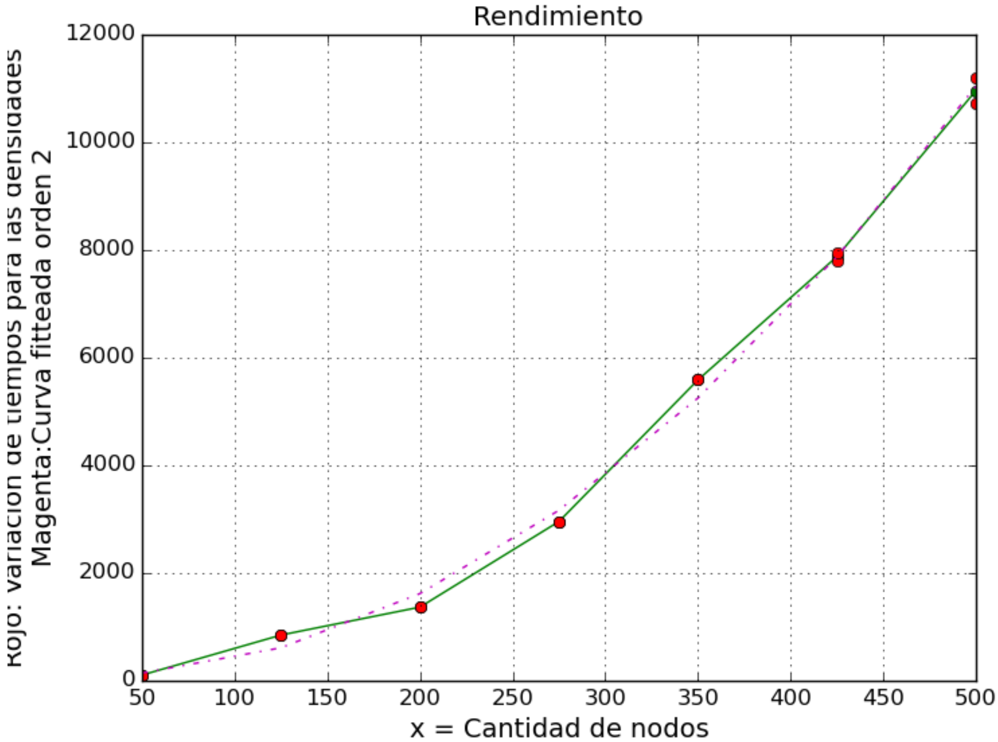
3.3.3. Rendimiento para grafos con densidad cuadratica de aristas

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

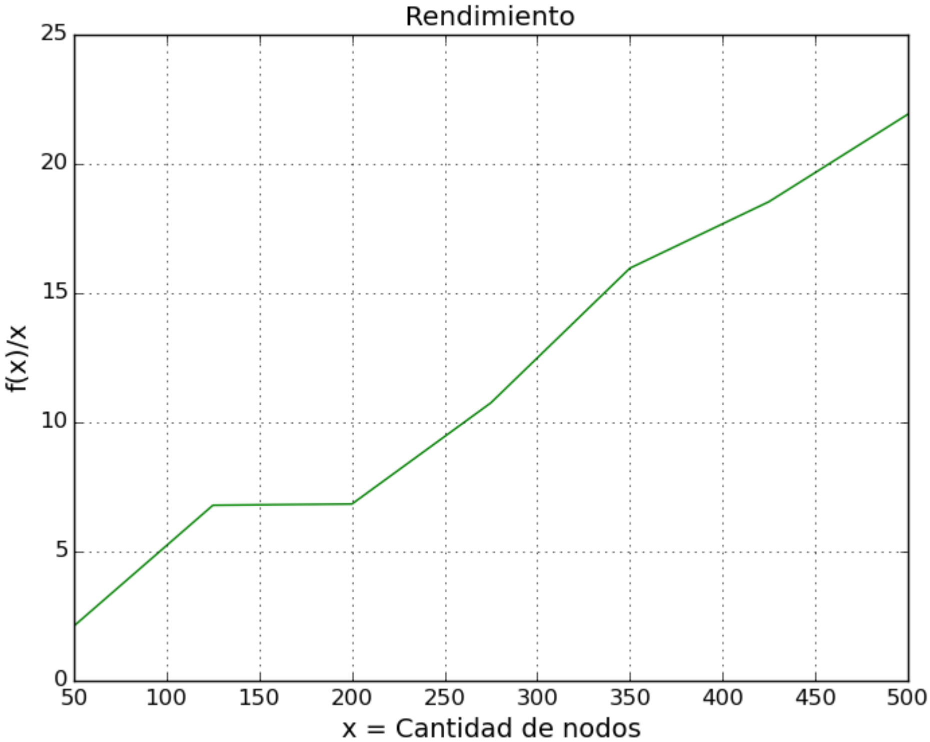
- peso maximo w2 = 400
- step nodos = 75
- step aristas = 4500
- aristas minimas = $\frac{n*(n-1)}{9}$
- aristas maximas = $\frac{n*(n-1)}{7}$

Tiempo de ejecución en microsegundos para esta familia

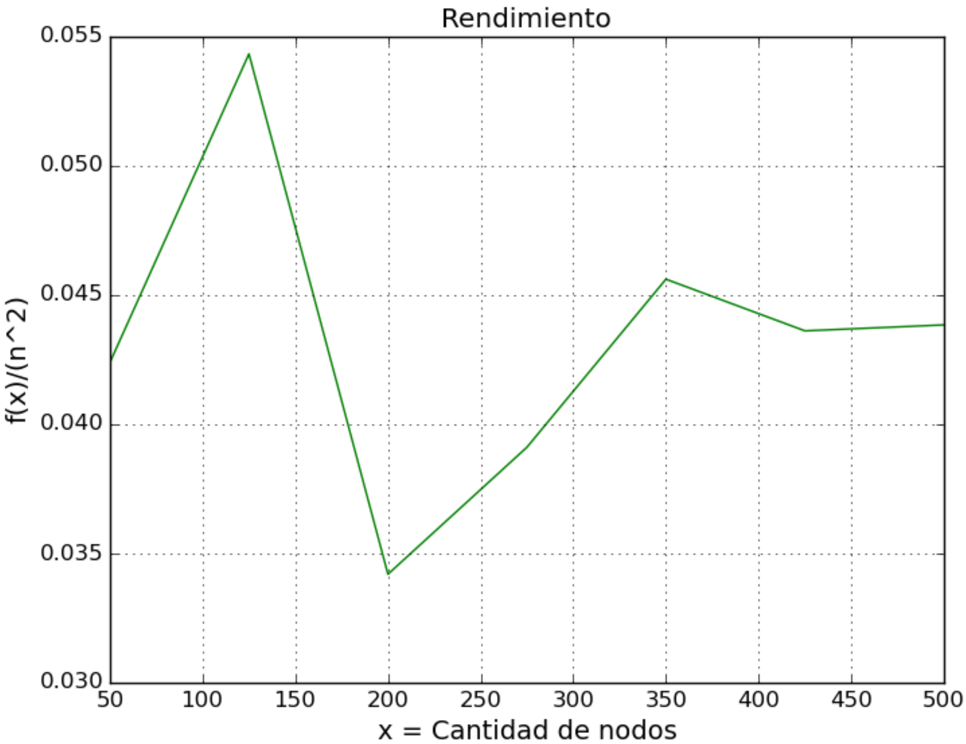
$y = f(x)$



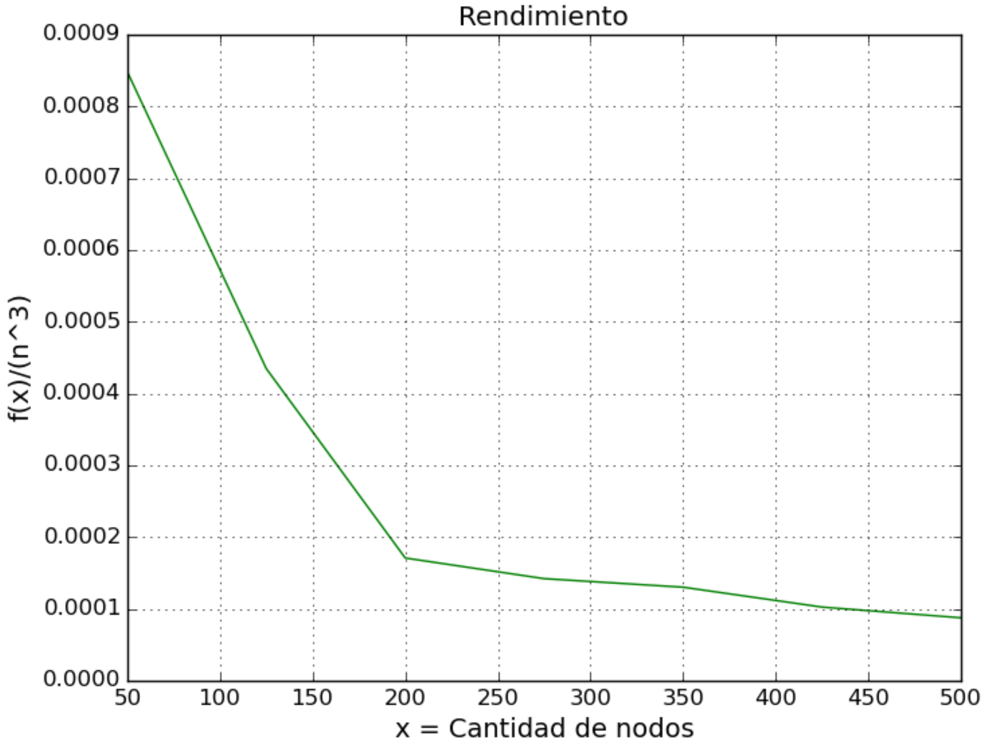
$y = f(x)/x$


Rendimiento
f(x)/x
x = Cantidad de nodos
25
20
15
10
5
0
50
100
150
200
250
300
350
400
450
500

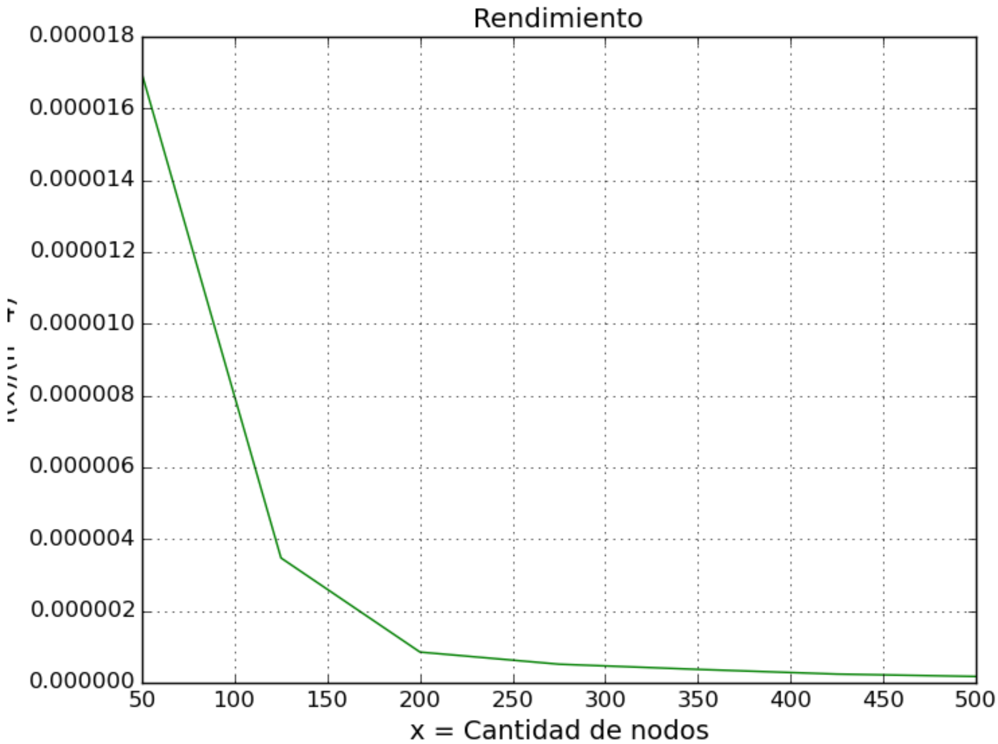
$$y = f(x)/x^2$$


Rendimiento
f(x)/(n^2)
x = Cantidad de nodos
0.055
0.050
0.045
0.040
0.035
0.030
50
100
150
200
250
300
350
400
450
500

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



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



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 $O(n^2 + mlogn)$ planteada acotada a grafos con m lineal ($O(n^2)$) y cuadrática ($O(n^2 * logn)$ con $log(n) \approx 11$ como máximo para $n = 2000$).

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

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 heuristica y realizaremos algunos calculos estadísticos acerca de la lejanía promedio de las soluciones obtenidas con respecto a la solucion 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 ejecucion secuencial y el calculo de la lejania mencionado aqui arriba para cada uno de los algoritmos(exacto y heuristica) 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 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: 100.0% de los casos
Lejania promedio de la heuristica a la solucion optima: 0%
Desv. estandar de la lejania entre las soluciones: 0
Minima lejania entre bqlocal y exacta: 0
Maxima lejania entre bqlocal y exacta: 0
```

Realmente podemos observar una efectividad impecable para grafos de baja densidad. La heurística golosa resulto 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 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: 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 lejanía podemos ver que hay casos donde la solucíon 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. Heuristica de busqueda local

Para encontrar soluciones aproximadas al problema de optimizar una función $f : S \to \mathbb{R}$ 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 “cercanos” 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 vecidad 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 $(X_0, X_1, ..., X_m)$ 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: $C = (v_1, ..., v_m)$ será un camino y A_c el conjunto de aristas que lo conforman. Dos caminos $C = (v_1, ..., v_n)$ y $C' = (v'_1, ..., v'_m)$ 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 $v_i \to v_{i+1}$ 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 $v_i \to v_{i+1}$, de forma tal que la contracción del sendero entre $v_i \to v_{i+1}$ 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.

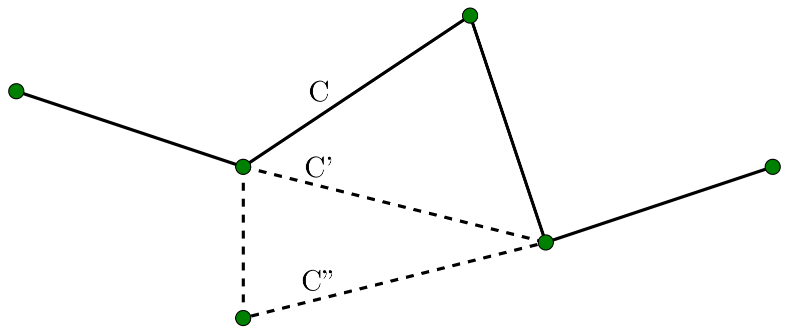


Figura 4: Tres caminos vecinos entre si.

4.2. Explicacion detallada del algoritmo propuesto

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

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

4.2.1. Obtencion de la solucion inicial factible

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

4.2.2. Criterio de terminación

El criterio de terminacion 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 parametro en el metodo main del codigo de la busqueda local. Para todos los experimentos salvo donde se indique lo contrario, se utilizo la vecindad N.

4.2.3. Pseudocodigo

A continuación el pseudocódigo de la búsqueda local.

procedure $Bq\_Local(Grafo\, g,\ vertice\, v_1,\ vertice\, v_2,\ int\, k, tipo\_busqueda) \rightarrow lista < eje >\ camino$
  $lista < eje >\ camino\ =\ dijkstra(g,\ v_1,\ v_2,\ costoW1)$
  if camino.costoW1 > K **then**
    No existe solucion.
  end if
  $bool\ valor\_mejora\ =\ \infty$
  while $valor\_mejora > 0$ **do**
    $switch(tipo\_busqueda)$
     $case\ subdividirPares$
      $valor\_mejora\ =\ bqLocalEntrePares(g, camino)$
     $case\ contraerTriplas$
      $valor\_mejora\ =\ bqLocalContraerTriplas(g, camino)$
     $case\ mejorarTriplas$:
      $valor\_mejora\ =\ bqlocalEntreTriplasReemplazando(g, camino)$
     $case\ combinarVecindades$:
      $valor\_mejora\_1\ =\ estimar\_bqLocalEntrePares(g, camino)$

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 continuacíon 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 longuitud 2, es decir hay una arista directa entre u, v , no hay nada que hacer en este caso, caso contrario, sea $S = \{u, ..., v_l, v_{l+1}, v_{l+2}, v_{l+3}..., v\}$ la solucion actual. Lo que hace en este caso es iterar sobre todas las 3-uplas consecutivas del camino, en este ejemplo sea (vl, 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 comun entre los extremos de la 3-upla buscando una mejor conexión entre extremos reemplazando el nodo intermedio, es decir, se busca una nueva conexion entre los extremos tal que, si el nuevo nodo intermedio es v_t, vale que:

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

Una iteracion 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 conexion, ademas 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 minimo sobre w_2 en la vecindad $N_1(s)$. Mas formalmente, se busca una solucion vecina del camino S, tal que siendo $w_1(P), w_2(P)$ los costos sobre w_1 y w_2 respectivamente de una solucion, 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 solucion obtenida, entonces valga que:

- Sea factible, pertenezca a $N_1(S)$, $w_1(S') \leq K$
- Sea minima 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 basicamente 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 solucion S (v_k, v_{k+1}) buscando si existe, un vecino en comun entre los extremos tal que el sendero (v_k, v_t, v_{k+1}) tenga costo menor(de forma minima entre los vecinos) sobre w_2 que la arista directa y que reflejado en la solucion candidata S' que surge de aplicar este cambio, siga siengo factible($w_2(S') \leq K$). La iteracion de sigue siendo sobre toda la vecindad $N_2(S)$ y quedandose 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 aristass $(v_1, v_2), (v_2, v_3)$ 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 ningín tipo de conexión con el camino mínimo según w_1. Podria intentar arreglarse el problema generalizando la idea de contraer triplas a contraer subcaminos consecutivos de mayor longuitud 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.

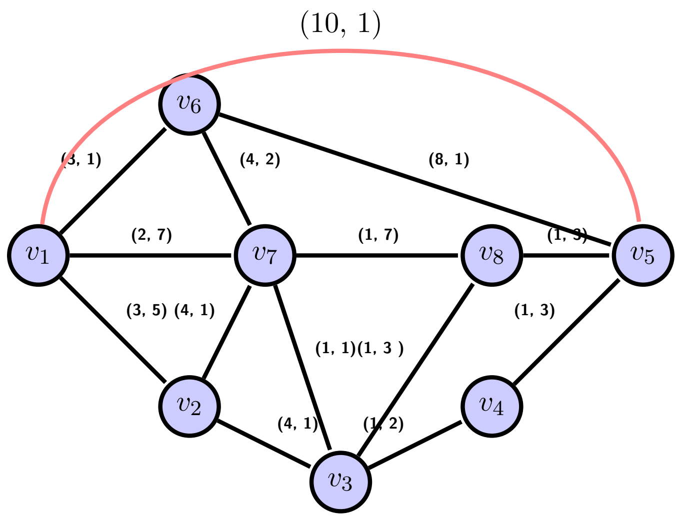


Figura 5: Ejemplo grafo malo para la heurística.

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)----[2, 7]---->(7)----[1, 7]---->(8)----[1,
    3]---->(5)
```

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

```
(1)----[3, 1]---->(6)----[4, 2]---->(7)----[1, 1]---->(3)----[1,
    2]---->(4)----[1, 3]---->(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 w1: 10 y peso w2: 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.

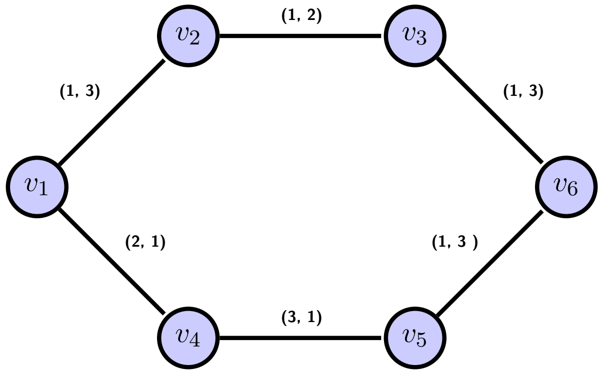


Figura 6: Ejemplo grafo malo para la heurística.

```
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 exploracíon 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 órden 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 longuitud 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 utilizano nodos que ya pertenecen al camino.Esto es indispesable para evitar la generacion 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 disjuntos el conjunto de nodos del camino actual y los nodos restantes del grafo, cualquier eleccion que hagamos no generar´a 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 opcion considerada fue no restringir la búsqueda de nuevos nodos, pero luego debería realizarse una poda de ciclos del camino. En la muchos de los casos esto realizariía 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 exploracion de las vecindades y aprovechando que el grafo es estatico, es decir, no se agregan o quitan aristas luego de ser leido 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. Experimentacion: Mediciones de Performance

En esta sección se mostrarán resultados de complejidad temporal empírica acerca del costo promedio de una iteracion de busqueda local, veremos que tal como analizamos en la sección anterior de complejidad, dicho costo esta situacion en el orden cuadratico $O(n^2)$. 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 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.

A continuacion presentamos los resultados de estos experimentos:

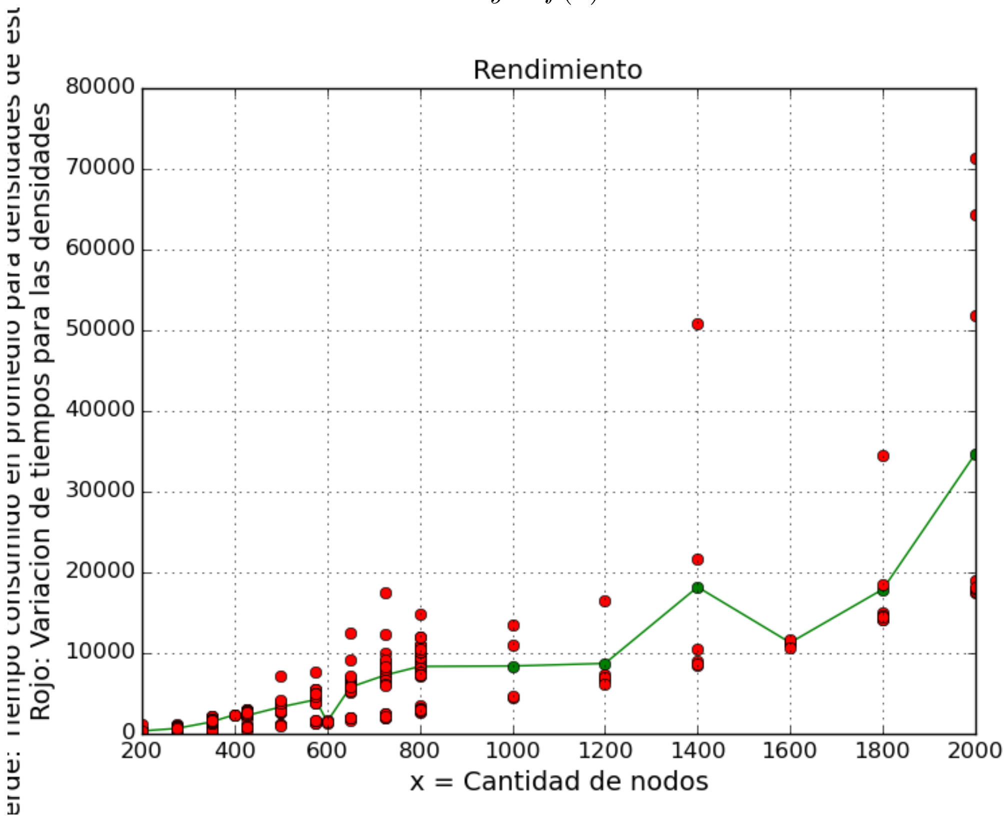
4.7.1. Rendimiento para grafos con densidad lineal de aristas

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

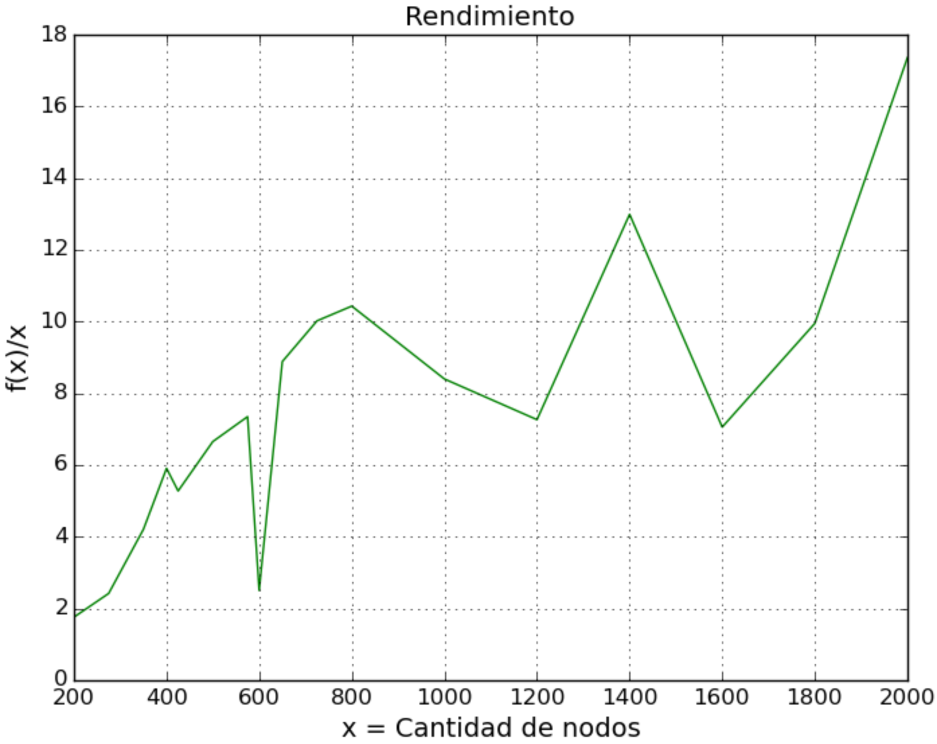
- peso maximo w2 = 200
- step nodos = 200
- step aristas = 2500
- aristas minimas = $n - 1$
- aristas maximas = $10 * n$

Tiempo de ejecución en microsegundos para esta familia

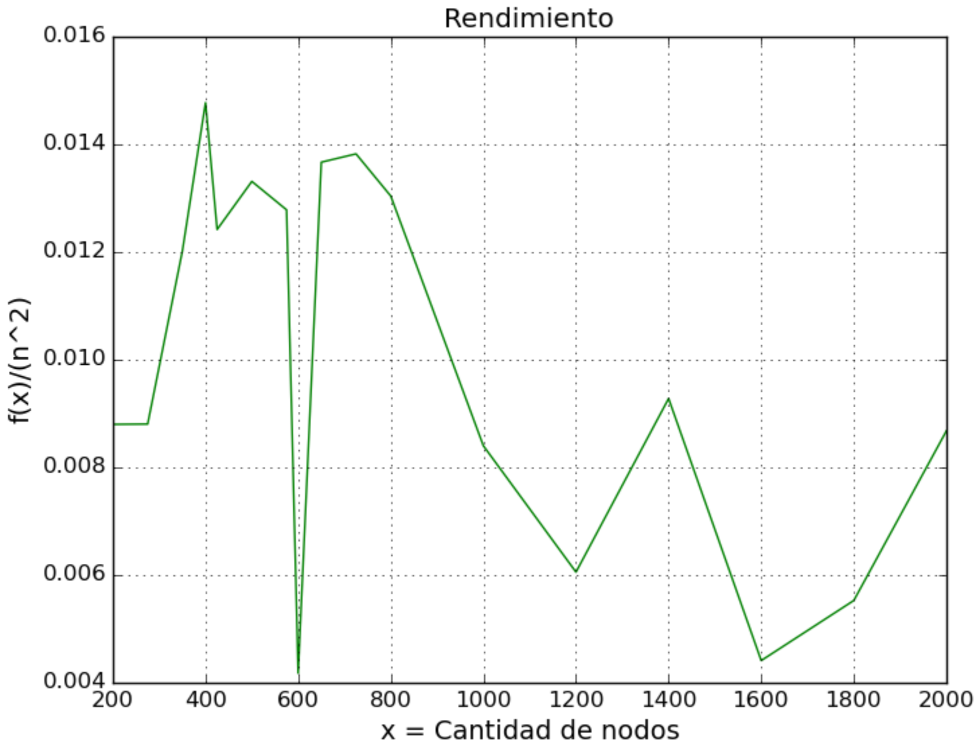
$y = f(x)$



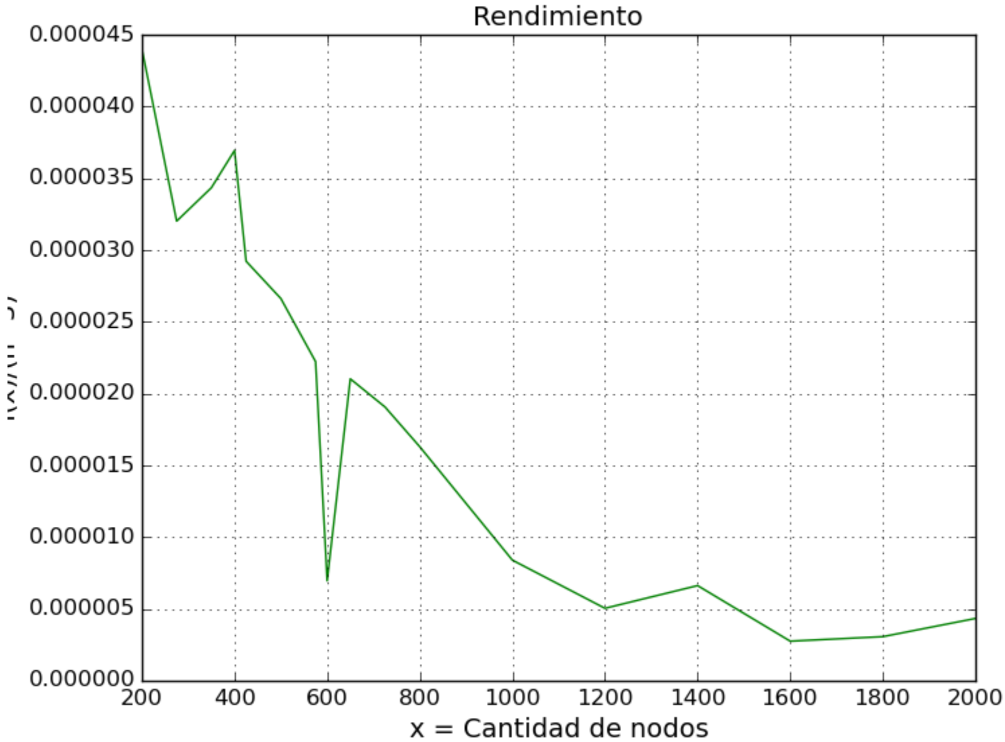
$y = f(x)/x$


Rendimiento
f(x)/x
x = Cantidad de nodos

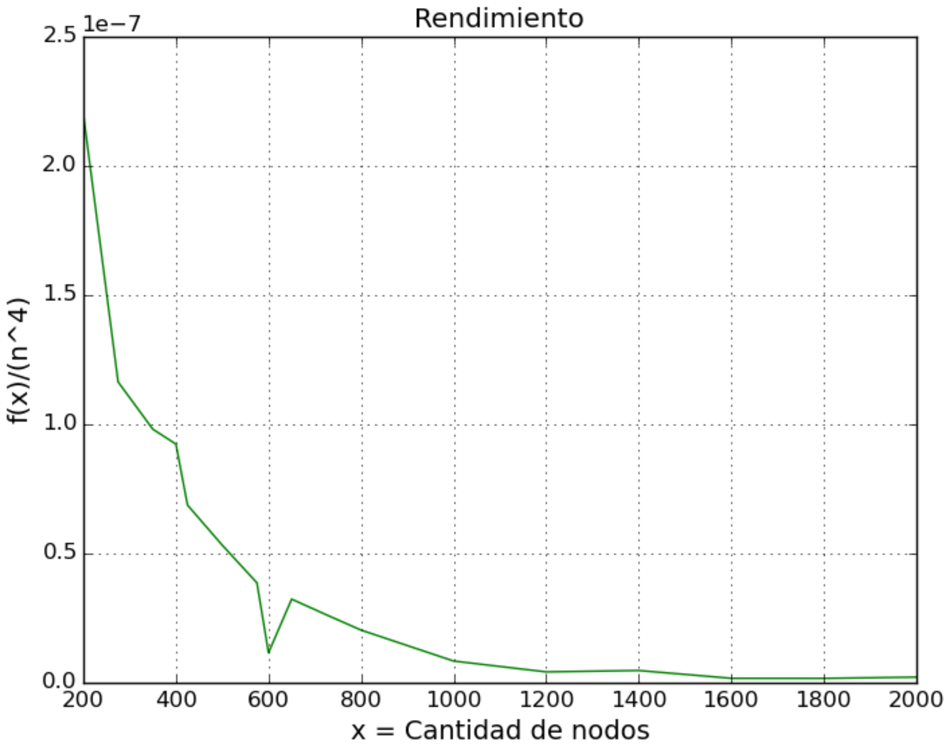
$$y = f(x)/x^2$$


Rendimiento
f(x)/(n^2)
x = Cantidad de nodos

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



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



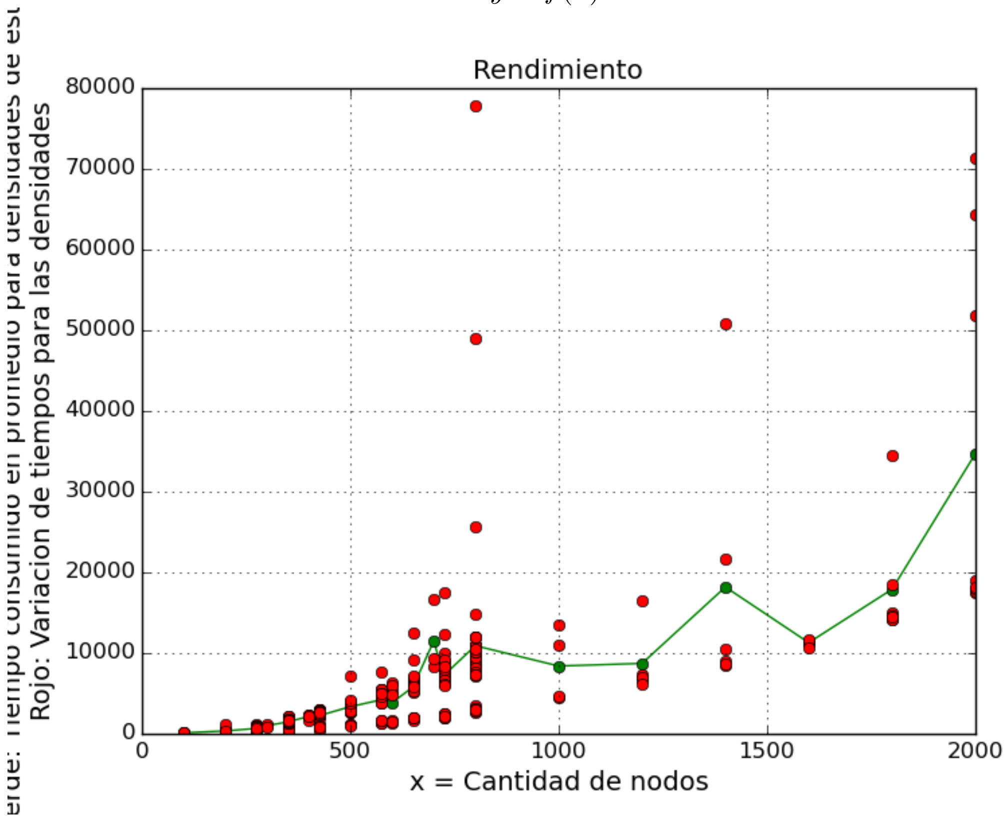
4.7.2. Rendimiento para grafos con densidad cuadratica de aristas

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

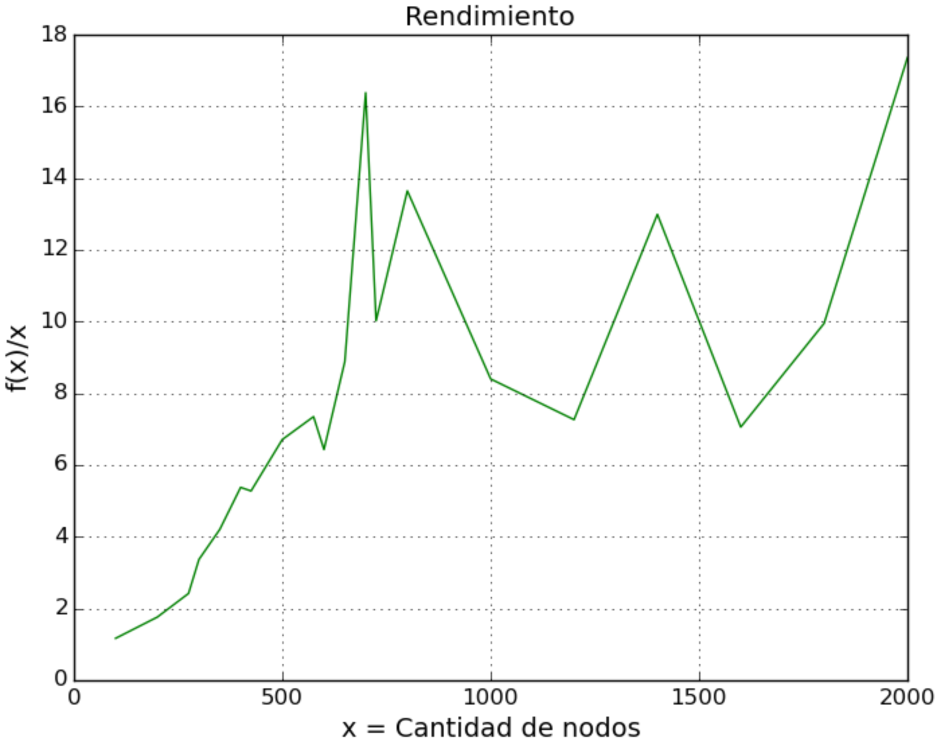
- peso maximo w2 = 200
- step nodos = 200
- step aristas = 2500
- aristas minimas = $\frac{n*(n-1)}{17}$
- aristas maximas = $\frac{n*(n-1)}{14}$

Tiempo de ejecución en microsegundos para esta familia

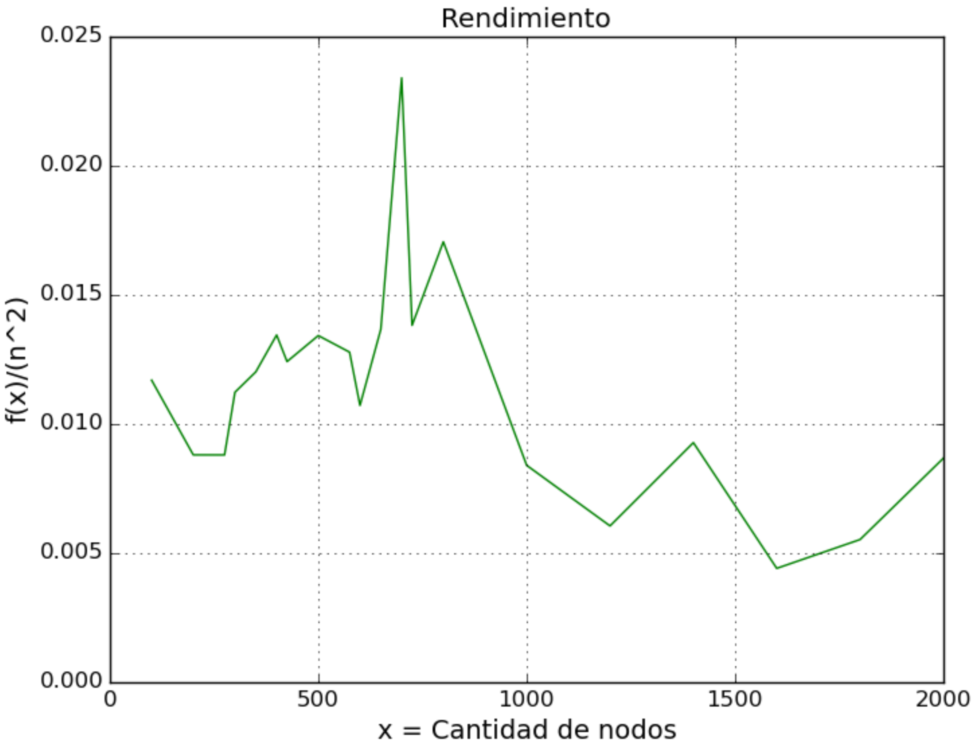
$y = f(x)$



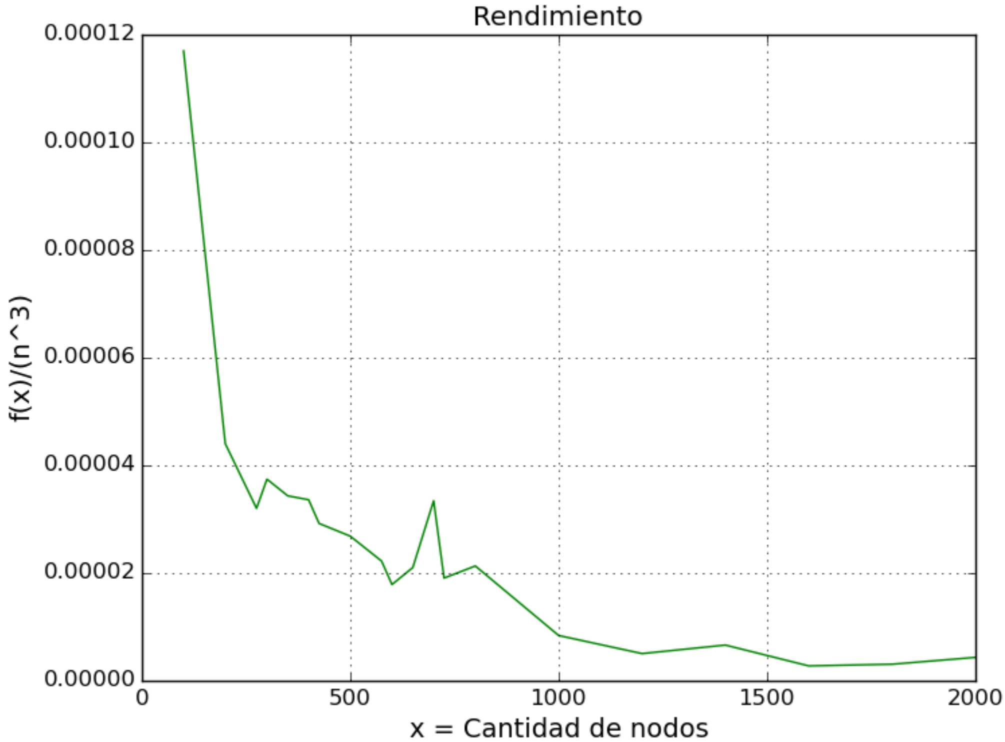
$y = f(x)/x$


Rendimiento
18
16
14
12
10
8
6
4
2
0
f(x)/x
0
500
1000
1500
2000
x = Cantidad de nodos

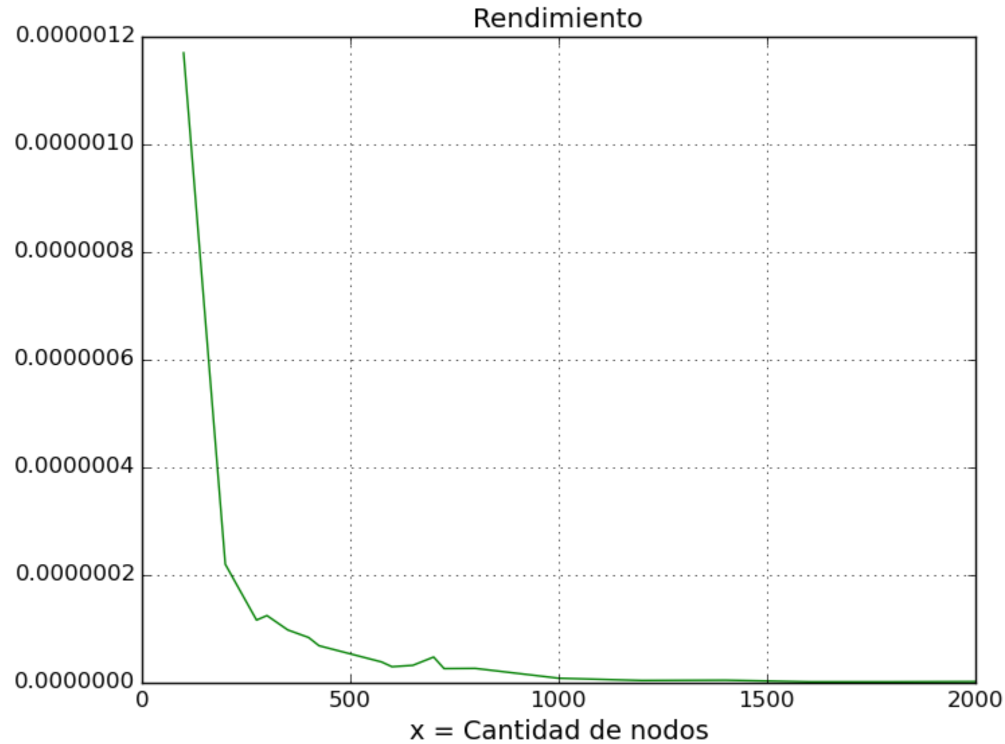
$$y = f(x)/x^2$$


Rendimiento
0.025
0.020
0.015
0.010
0.005
0.000
f(x)/(n^2)
0
500
1000
1500
2000
x = Cantidad de nodos

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



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



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 funcion marcadamente decreciente, con lo cual, llegamos a la conclusion 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 w2 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.

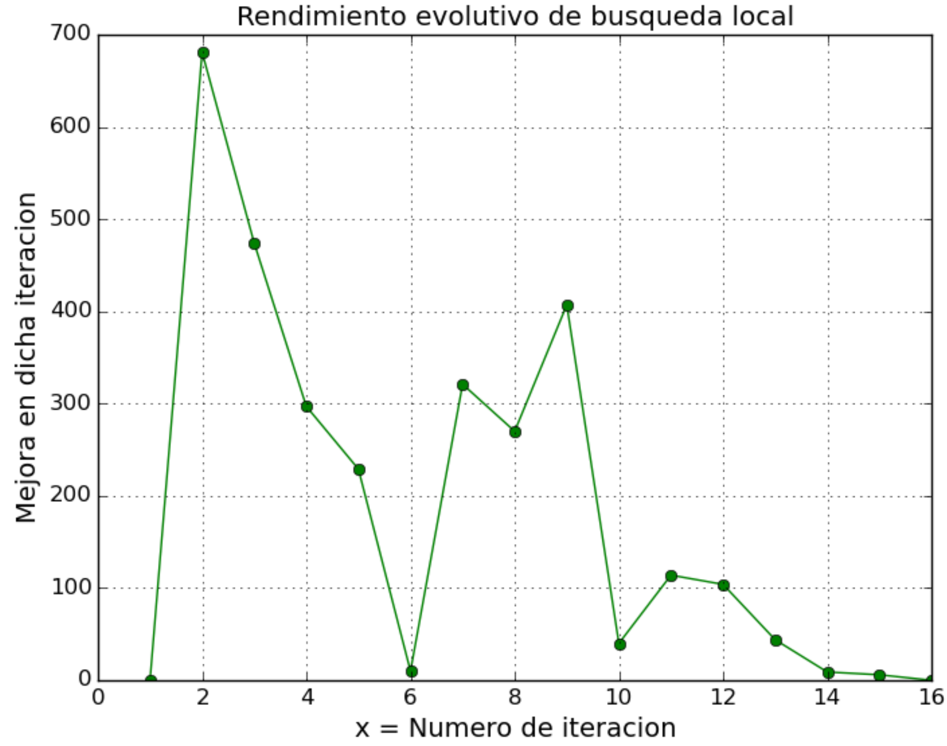
Luego se realizan dos graficos, el primero, referido a la variacion entre cada iteracion, indica en el eje X el numero de la iteracion y en el eje Y la mejora realizada en dicha iteracion a la solucion previa. El segundo, referido a la variacion absoluta del costo a minimizar, indica en el eje X la cantidad 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

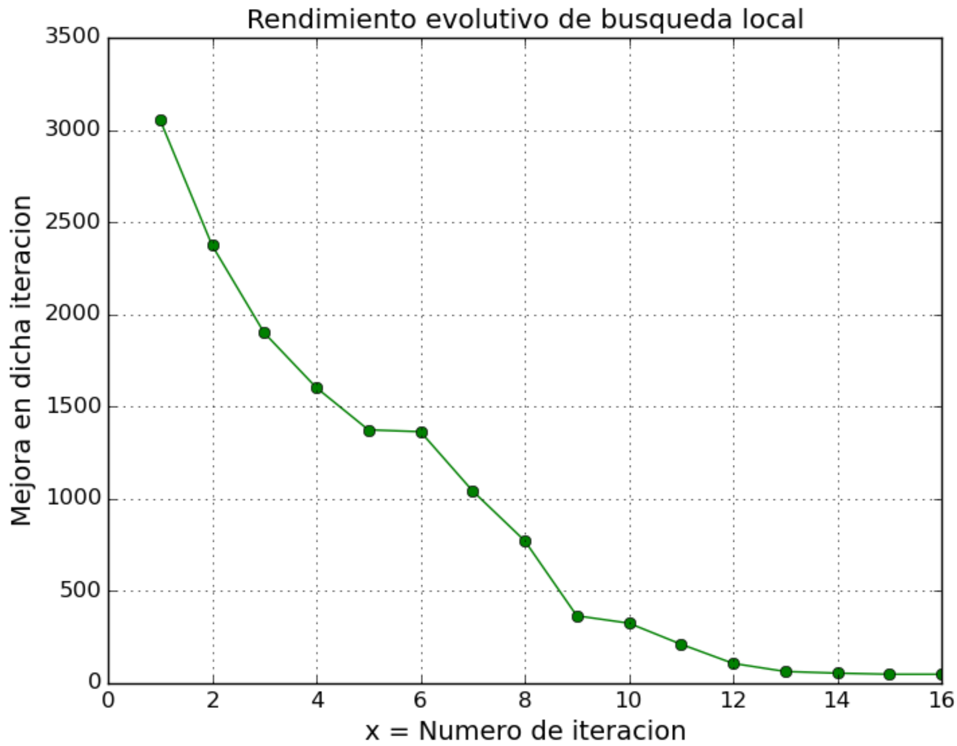
Parametros del experimento:

- 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 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: 34
Iteraciones promedio: 5
```

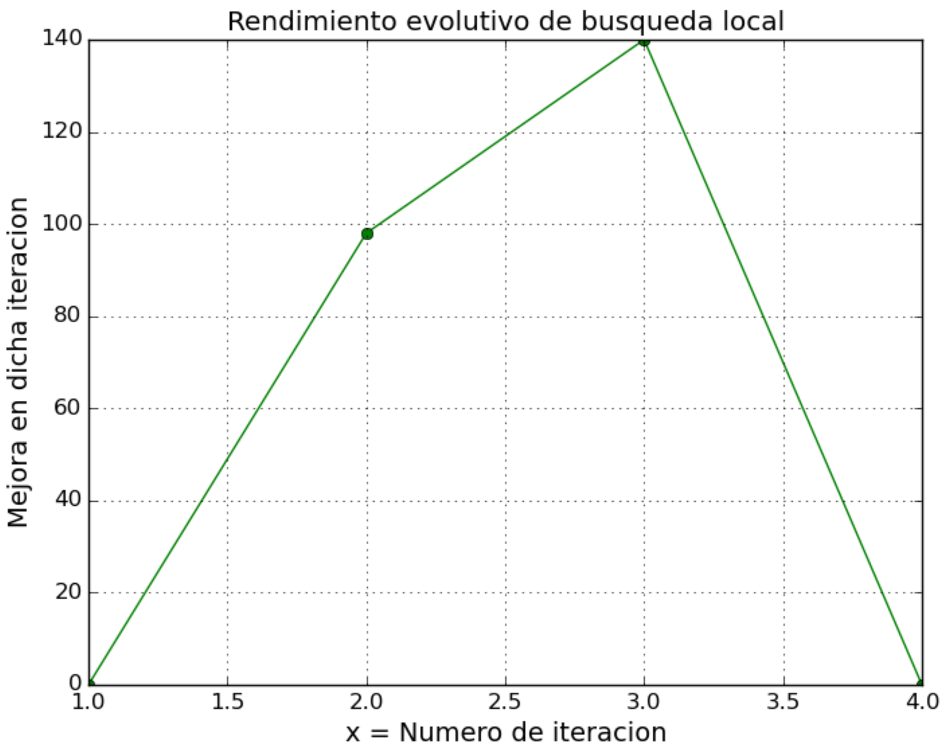
```
Iteraciones stddev: 2.99726
Minima cantidad de iteraciones: 2
Maxima cantidad de iteraciones: 15
Mejora promedio del costo w2: 934
Mejora stddev del costo w2: 523.031
Minima mejora en w2 registrada: 201 en 2 iteraciones
Maxima mejora en w2 registrada: 3006 en 15 iteraciones
```

4.8.2. Rendimiento evolutivo en grafos con densidad lineal de aristas

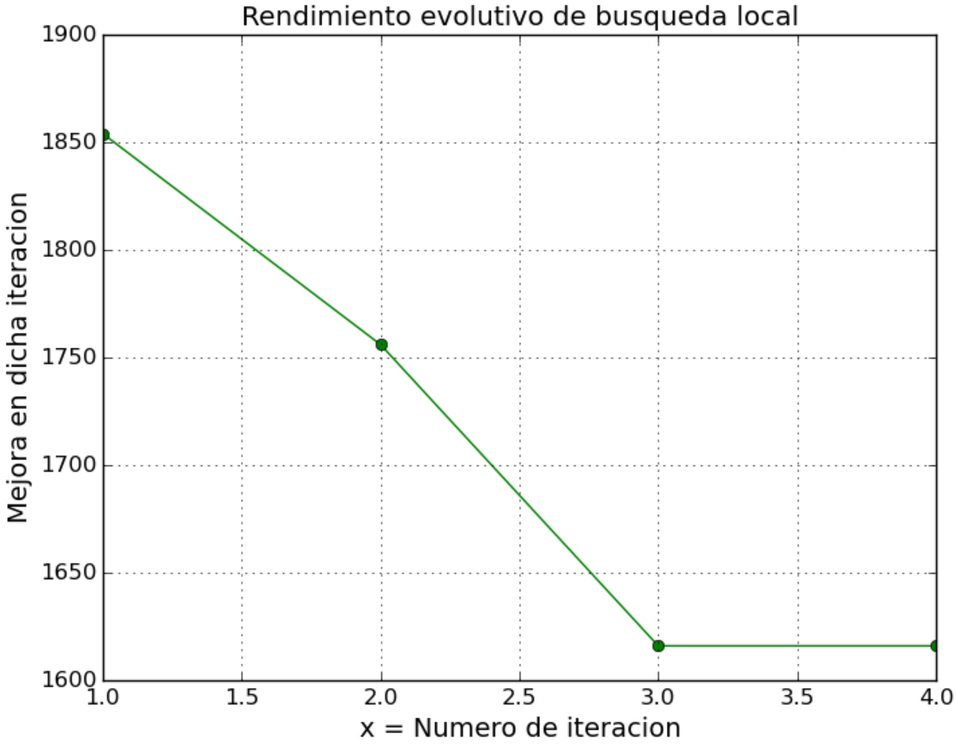
Parametros del experimento:

- 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 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 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 modificacióñes 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 bajísimos. Vamos a

basarnos en estos resultados para deducir que una familia mala de grafos para la busqueda 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 solucion inicial.

4.9. Experimentacion: Distintas soluciones iniciales factibles

4.9.1. Grafos particulares

Como mencionamos antes, se utiliza Dijkstra sobre w_1 para obtener la solucion inicial factible, para comenzar las iteraciones de búsqueda local. En esta sección decidimos variar esto y establecer la solucion inicial de busqueda local con la heuristica golosa. Para ciertos grafos preestablecidos que pueden encontrarse en la carpeta `grafos especiales` las soluciones finales de busqueda local con los 2 modos de solucion inicial son iguales, lo que varió de forma notable fue la cantidad de iteraciones de busqueda local necesarias para llegar a dicha solucion 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 seccion anterior, que evalua el rendimiento con dijkstra sobre w_1 como solucion inicial, para la misma familia de grafos aleatorios el uso de la heuristica golosa como solucion inicial perjudica muchísimo el rendimiento de la busqueda local, reduciendo la cantidad de iteraciones promedio de 5 a 1, con una gran reduccion en la desviacion estandar y una reducción en la amplitud entre cantidad máxima y minima 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 solucion a nivel local. Esta experimentación sirve como preludio 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 metaheuristica, 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 heuristica 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 solucion final, pues la decision golosa de obtener siempre el camino vecino que mas optimice localmente la funcion target puede pasar por alto el caso donde haber obtenido una solucion peor en este paso, nos dará una mas alta mejora en una iteracion siguiente, produciendo una mejor solucion final.

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

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 heuristica y realizaremos algunos calculos estadísticos acerca de la lejanía promedio de las soluciones obtenidas con respecto a la solucion 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 ejecucion secuencial y el calculo de la lejania mencionado aqui arriba para cada uno de los algoritmos(exacto y heuristica) 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. Metaheuristica 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 seguún w_2. En lugar de esto, ahora armamos una lista restringida de canditatos, 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 descripta, 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 mÃnimo 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 vaciá la cola y de esa forma llega a la solución golosa determinística. Notamos esto prematuramente en la experimentación al obtener resultados idénticos entre la heurśtica 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 pequeno, 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 $tipo_e jecucion == por\_valor$ **then**
    $lista\,nodo\,candidatos = \emptyset$
    $int\,i = 0$
    while $i < tamano(cola)$ **do**
      if $cola[i] \leq valor\_limite$ **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. Critero 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 búqueda 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. $citerio\_terminacion$: el criterio de terminación de GRASP.
5. max_its : la cantidad de iteraciones lḿite 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.

procedure $Sol\_GRASP(Grafo\, g,\ vertice\, v_1,\ vertice\, v_2,\ int\, k, criterios) \rightarrow lista < eje >\ camino$

$bool\, condicion\_terminacion\ =\ false$
$lista < eje >\ mejor\_solucion\ = \emptyset$
$int\, costo\_mejor\_solucion\ =\ \text{ínf}$

```
    lista < eje > camino = ∅
    int cant_iters = 0
    int cant_iters_sin_mejora = 0
    int cant_iters_sin_sol_rgreedy_factible = 0
    bool sol_valida_rgreedy = false
    vector < pair < int, costo >> mejora_iters_grasp

    while !condicion_terminacion do
        camino = solucion_golosa(g, v1, v2, tipo_golosa, β)
        sol_valida_rgreedy = validar_solucion(camino)        ▷ Obtenemos solucion greedy

        if haysolucion then
            if sol_valida_rgreedy then                       ▷ Validamos si es factible
                cant_iters_sin_sol_rgreedy_factible = 0
                int mejora_iteracion_actual = 0
                int cant_iters_bqlocal = 0

                while mejora_iteracion_actual > 0 do          ▷ Aplicamos Busqueda Local
                    mejora_iteracion_actual = busqueda_local(g, tipo_bq, camino)
                    cant_iters_bqlocal + +
                end while
                int costo_sol_actual = costo_w2(camino)

                if costo_sol_actual < costo_mejor_solucion then     ▷ Reemplazamos si es mejor
solucion
                    if cant_iters > 0 then
                        agregar(mejora_iters_grasp, par < cant_iters, costo_mejor_solucion−costo_solucion_act
)
                    end if
                    costo_mejor_solucion = costo_sol_actual
                    mejor_solucion = camino
                    cant_iters_sin_mejora = 0
                else
                    if cant_iters > 0 then
                        agregar(mejora_iters_grasp, par < cant_iters, 0) >)
                    end if
                    cant_iters_sin_mejora + +
                end if
                cant_iters + +
            else                                              ▷ sol_valida_rgreedy = false
                cant_iters_sin_sol_greedy_rand_factible + +
                if cant_iters_sin_sol_greedy_rand_factible ≥ bad_rgreedy_its then
                    if tipo_golosa == por_cantidad then       ▷ Maximo de its de greedy sin Solucion
                        if parametro_beta ≥ 2 then            ▷ Ajustamos parametros de GRASP
                            parametro_beta − −
                        end if
                    else if tipo_golosa == por_valor then
                        parametro_beta = 0
                    end if
                    cant_iters_sin_sol_greedy_rand_factible = 0
                end if
            end if
        else                                                  ▷ No hay Solucion
            break
```

```
        end if
        if criterio_terminacion == 1 then          ▷ Si se cumple el criterio de terminacion
            condicion_terminacion = (cant_iters < max_its)
        else if criterio_terminacion == 2 then
            condicion_terminacion = (cant_iters_sin_mejora < max_its_sin _mejora)
        end if
    end while
    return mejor_solucion
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 $O(n^2 + mlogn)$. 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 $O(mlogn)$ (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: $O(n^2 + m * logn + \beta * n)$ y en caso de ser por valor es de $O(n^2 + mlogn)$. 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 difere 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 $O(n^2 + mlogn + k * (n^2)$.

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, analizamos primero con RCL por cantidad, con $\beta = n$(peor caso), y luego con RCL por valor, con $\beta = 0{,}99$. Las dos experimentaciones tienen como premisa agrandar lo máximo posible la RCL para poder ver la complejidad en peor caso, y los resultados de ambos experimentos fueron muy similares, condensados en estos gráficos.

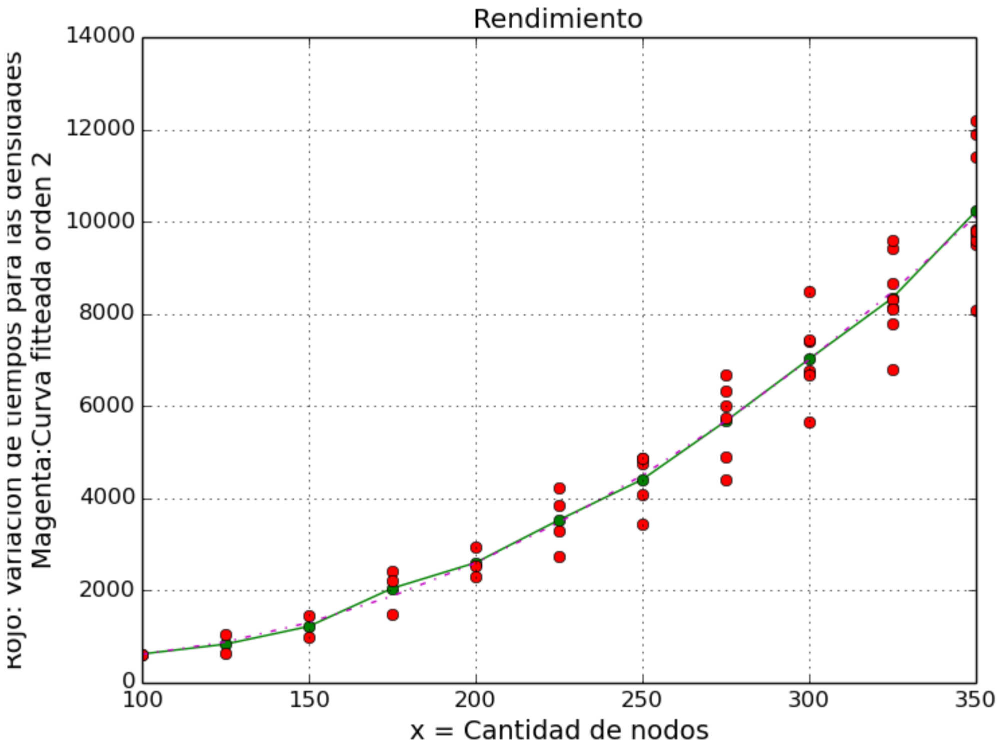
5.6.1. Rendimiento para grafos con densidad cuadratica de aristas

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

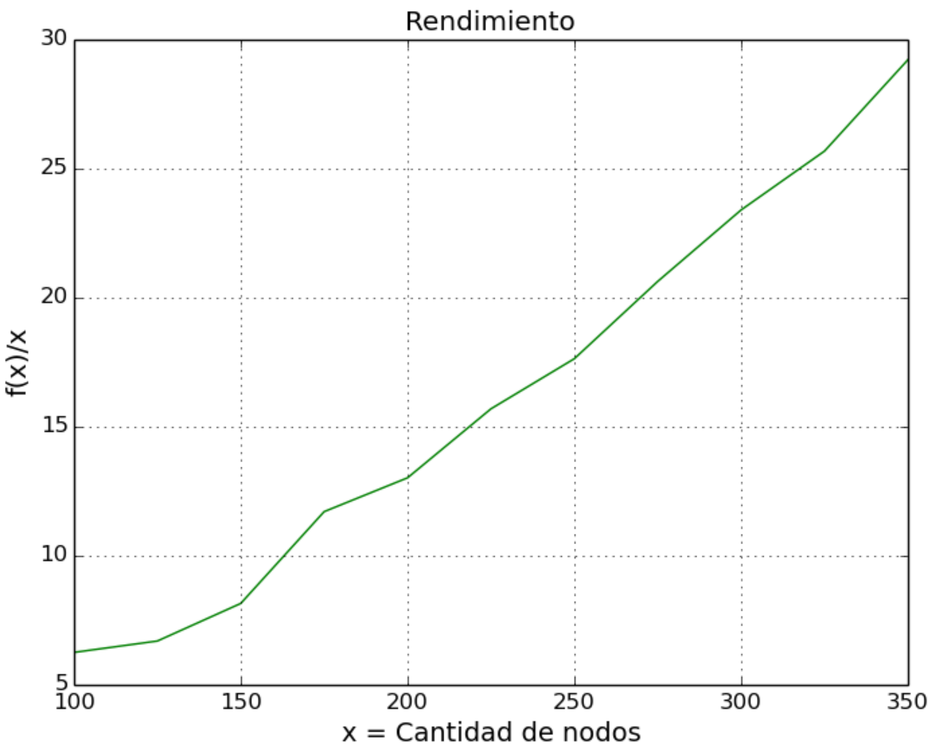
- aristas maximas $= \frac{n*(n-1)}{3}$

Tiempo de ejecución en microsegundos para esta familia

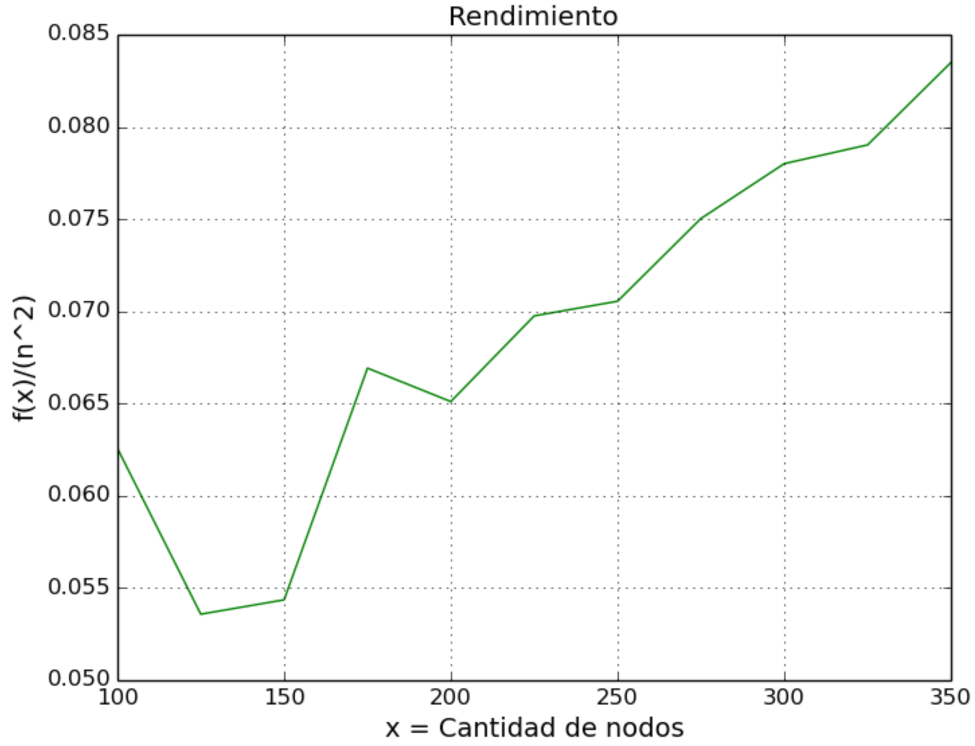
$y = f(x)$



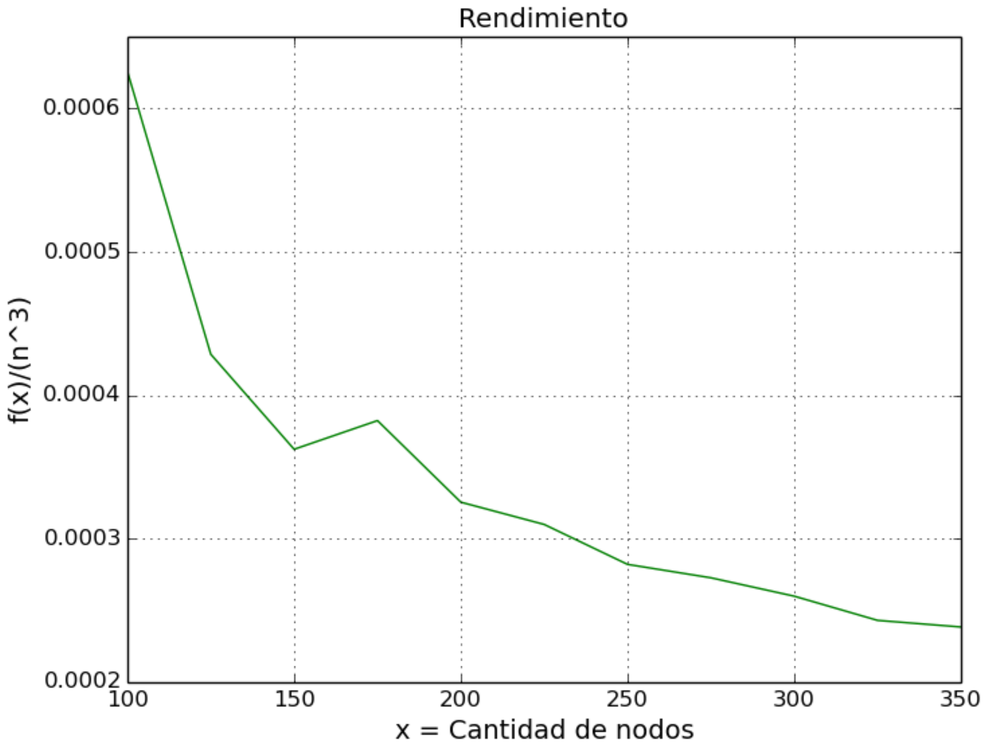
$y = f(x)/x$



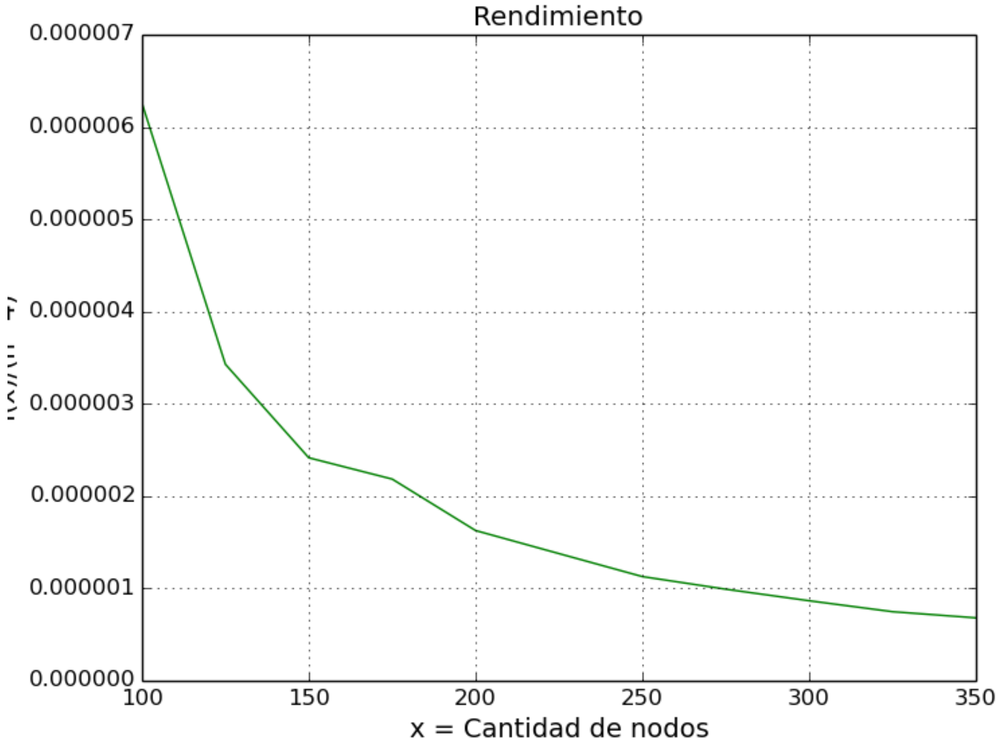
$y = f(x)/x^2$


Rendimiento
0.085
0.080
0.075
0.070
0.065
0.060
0.055
0.050
f(x)/(n^2)
100
150
200
250
300
350
x = Cantidad de nodos

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


Rendimiento
0.0006
0.0005
0.0004
0.0003
0.0002
f(x)/(n^3)
100
150
200
250
300
350
x = Cantidad de nodos

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

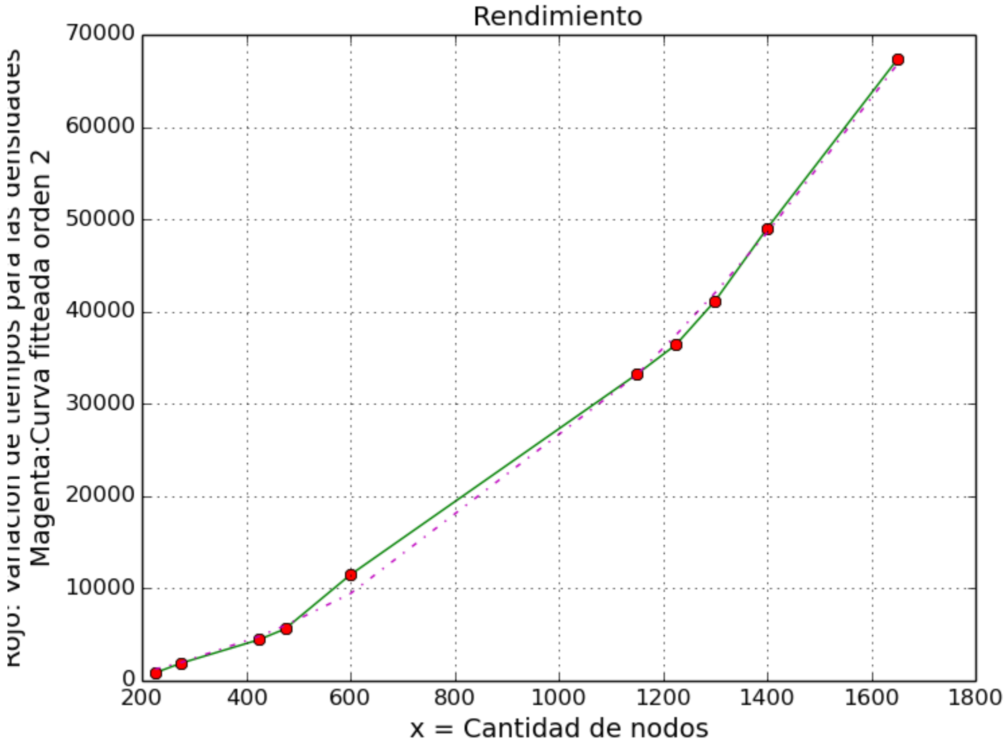


5.6.2. Rendimiento para grafos con densidad lineal de aristas

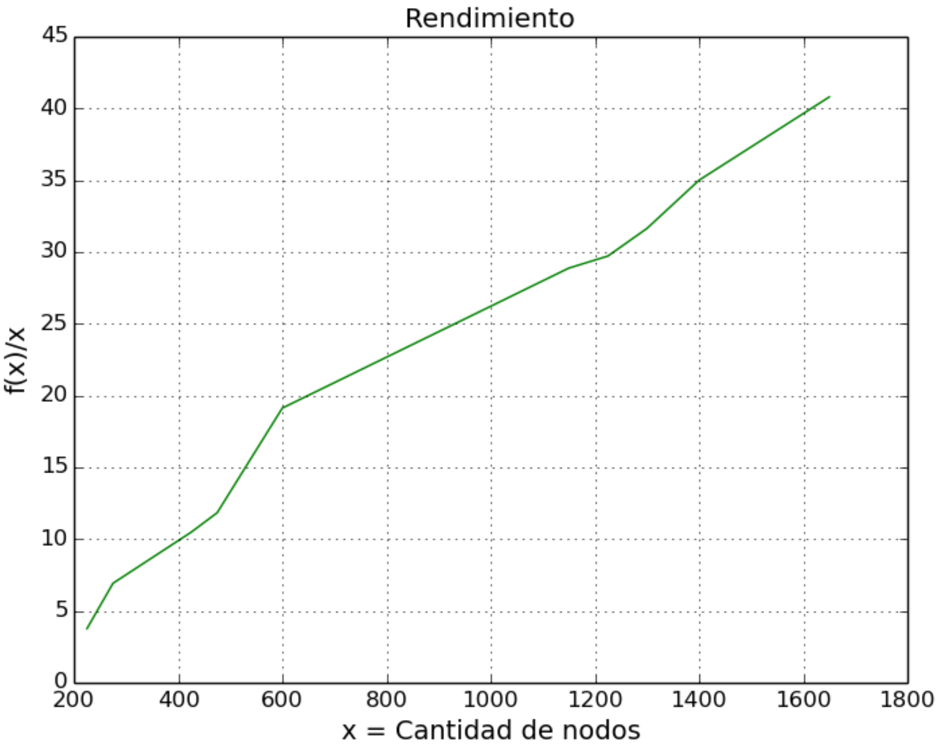
- cant nodos min = 200
- cant nodos max = 2000
- peso maximo w1 = 200
- peso maximo w2 = 200
- step nodos = 200
- step aristas = 2500
- aristas minimas = $n - 1$
- aristas maximas = $10 * n$

Tiempo de ejecución en microsegundos para esta familia

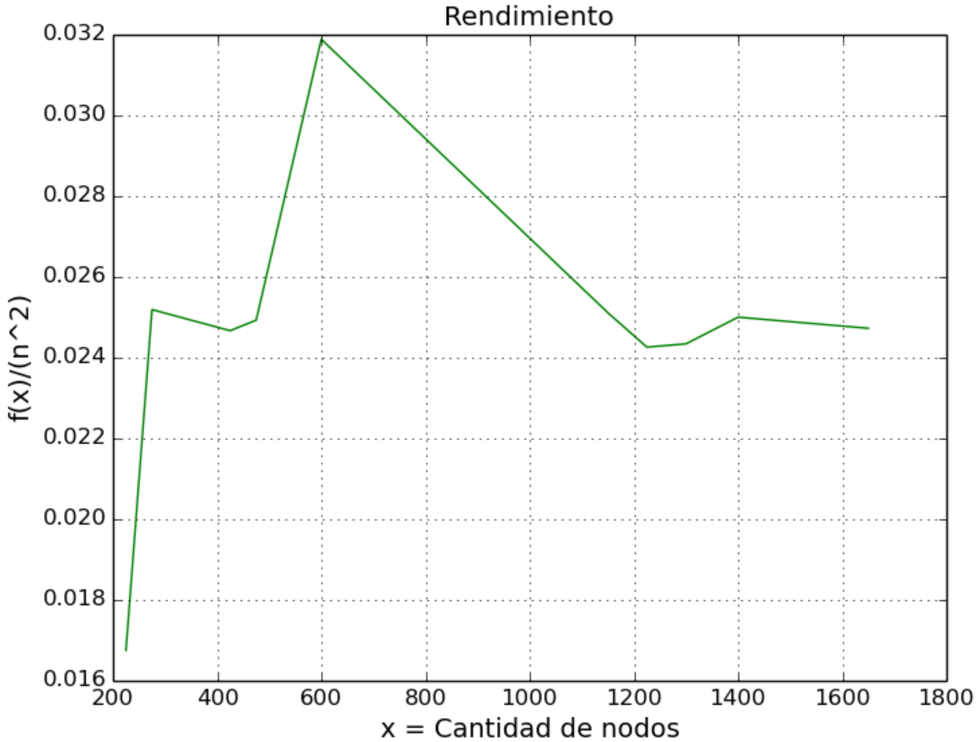
$y = f(x)$


Rendimiento
Rojo: variación de tiempos para las densidades
Magenta:Curva fitteada orden 2
70000
60000
50000
40000
30000
20000
10000
0
200
400
600
800
1000
1200
1400
1600
1800
x = Cantidad de nodos

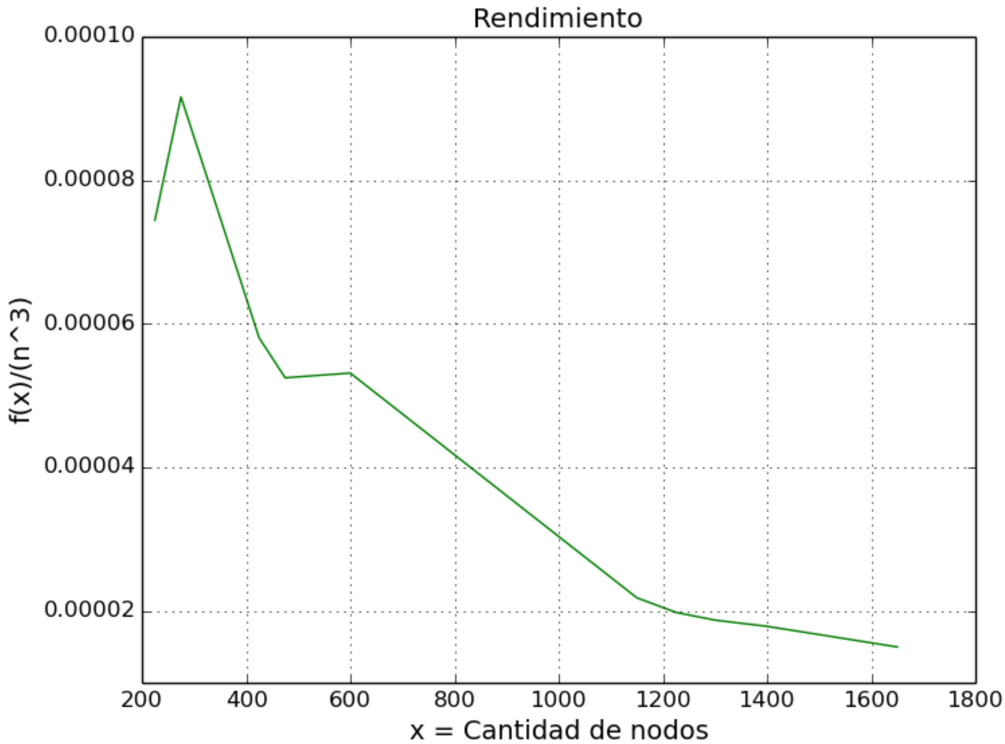
$$y = f(x)/x$$


Rendimiento
f(x)/x
45
40
35
30
25
20
15
10
5
0
200
400
600
800
1000
1200
1400
1600
1800
x = Cantidad de nodos

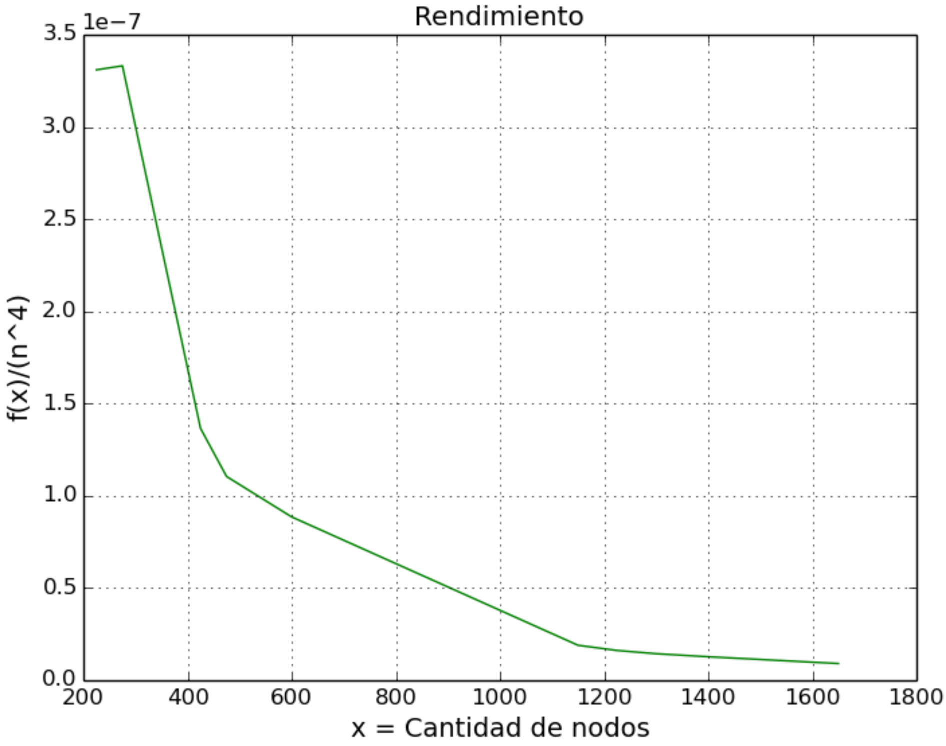
$$y = f(x)/x^2$$


Rendimiento
f(x)/(n^2)
0.032
0.030
0.028
0.026
0.024
0.022
0.020
0.018
0.016
200
400
600
800
1000
1200
1400
1600
1800
x = Cantidad de nodos

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


Rendimiento
f(x)/(n^3)
0.00010
0.00008
0.00006
0.00004
0.00002
200
400
600
800
1000
1200
1400
1600
1800
x = Cantidad de nodos

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



En el caso lineal puede verse claramente que la curva divida por n^3 es decreciente y la curva dividida por n^2 parece mantenerse constante en valores más grandes lo cual coincide con nuestra complejidad teórica.

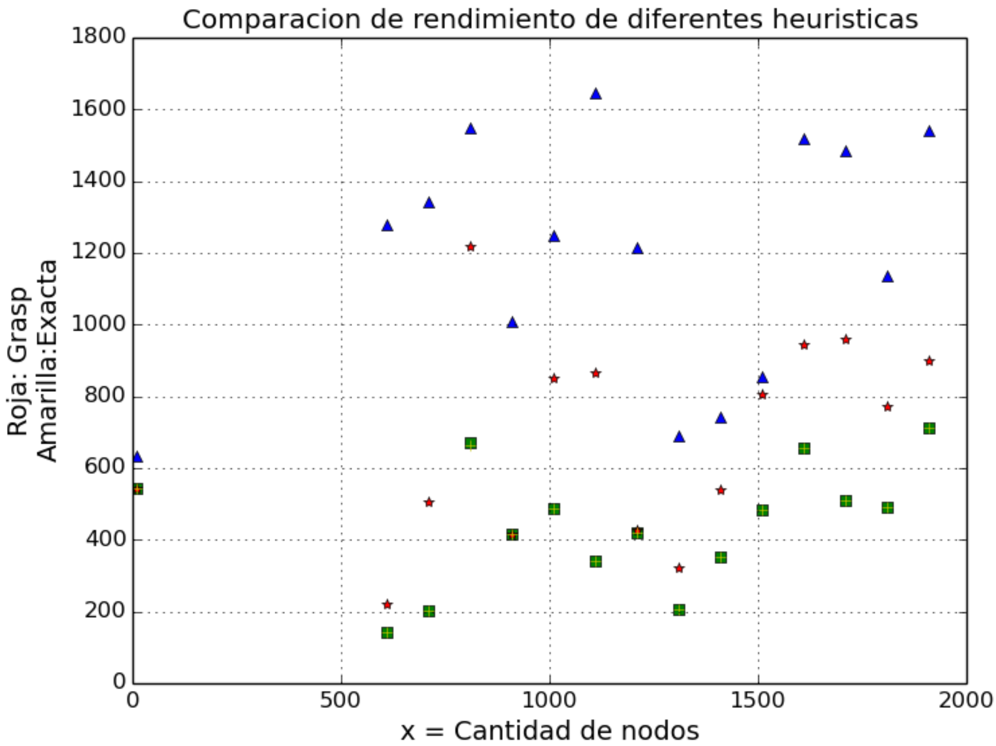
En el caso de densidad cuadrática, la curva divida por n^3 es decreciente, y la curva dividida por n^2 es creciente, con lo cual la complejidad se sitúa en un orden entre éstos. Nuestra complejidad teórica de $O(n^2 + mlogn + k * (n^2)$ puede ser ajustada considerando que $m \in O(n^2)$, nos quedará dominada por $O(n^2logn + k * (n^2))$, lo cual coincide con los resultados empíricos ya que $n^2logn \in O(n^3)$ pero n^2logn no pertenece a $O(n^2)$, y además al crecer n y la densidad de aristas del grafo crece k, la cantidad de iteraciones que puede realizar la búsqueda local, por lo cual la curva debe situarse sobre n^2 y debajo de n^3.

A continuación se muestran experimentos de análisis de solución de la metaheurística es decir, dados conjuntos de grafos aleatorios, vamos a examinar el porcentaje de soluciones optimas obtenidas por esta heuristica y realizaremos algunos calculos estadísticos acerca de la lejanía promedio de las soluciones obtenidas con respecto a la solucion exacta en los grafos del conjunto de instancias de las muestras. Para esto, se presentaran varios experimentos, primero variando la cota K de los grafos y analizando soluciones, luego variando el parámetro β. **Nota:** La lejanía entre 2 soluciones se mide haciendo el siguiente calculo: $100 * (\frac{solucionHeuristica}{solucionOptima} - 1)$

5.6.3. Optimalidad para grafos lineales con $w1 = 120$

- cant nodos min = 200
- cant nodos max = 2000
- peso maximo w1 = 250
- peso maximo w2 = 400
- limit w1 = 120
- step nodos = 100

- step aristas = 2500
- aristas minimas = $2 * n$
- aristas maximas = $20 * n$

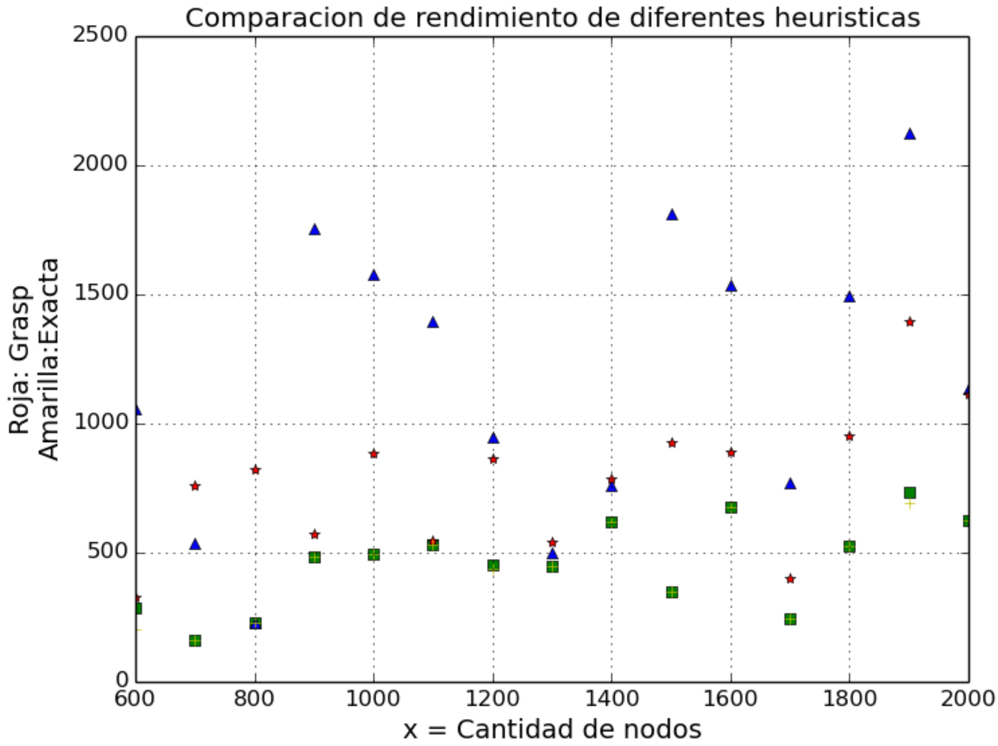



```
Cantidad de tests realizados: 28
Tiempo promedio microsegundos heuristica: 53290.671
Tiempo promedio microsegundos exacto: 29453.678
Porcentaje de aciertos(cantidad de veces que GRASP da la sol exacta/
    cantidad de tests hechos): 19.455
Porcentaje de alejamiento de la heuristica a la solucion exacta promedio
    entre grasp y exacta: 41.852
Desviacion estandar del alejamiento de la heuristica a la solucion exacta
    promedio entre grasp y exacta: 93.201
Minimo alejamiento porcentual entre grasp y exacta: 0
Maximo alejamiento porcentual entre grasp y exacta: 173.798
```

5.6.4. 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 = $20 * n$

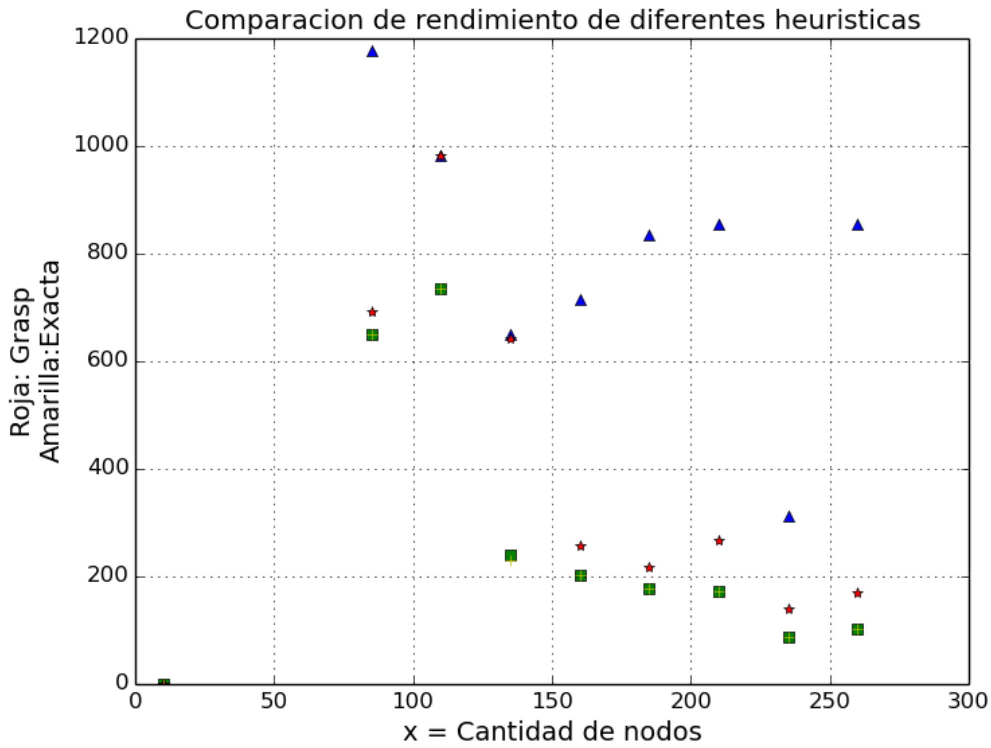



```
Cantidad de tests realizados: 28
Tiempo promedio microsegundos heuristica: 63565.671
Tiempo promedio microsegundos exacto: 31439.678
Porcentaje de aciertos(cantidad de veces que GRASP da la sol exacta/
    cantidad de tests hechos): 14.285
Porcentaje de alejamiento de la heuristica a la solucion 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
```

5.6.5. Optimalidad para grafos cuadráticos con $w1 = 200$

- cant nodos min = 10
- cant nodos max = 260
- peso maximo w1 = 250
- peso maximo w2 = 400
- limit w1 = 200
- step nodos = 25
- step aristas = 5000
- aristas minimas = $(n * (n - 1)) = /12$

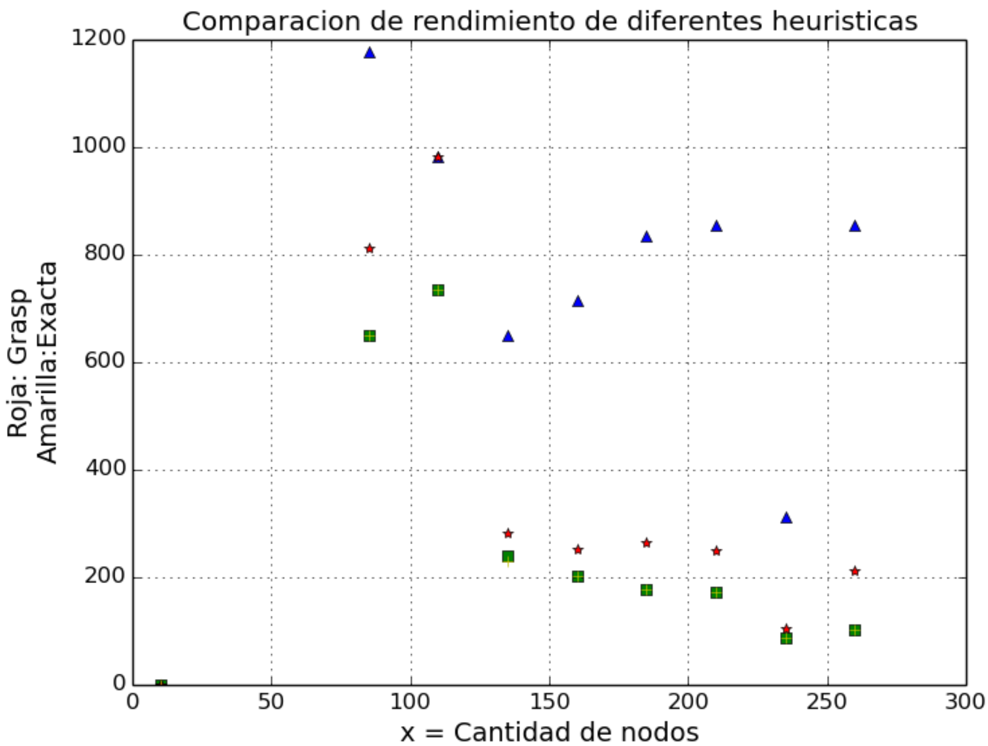
- aristas maximas $= n * (n-1)) = /2$




```
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
```

5.6.6. Optimalidad para grafos cuadráticos con $w1 = 150$

- cant nodos min = 10
- cant nodos max = 260
- peso maximo w1 = 250
- peso maximo w2 = 400
- limit w1 = 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 heuristica: 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 heuristica a la solucion exacta promedio
    entre grasp y exacta: 81.088
Desviacion estandar del alejamiento de la heuristica a la solucion exacta
    promedio entre grasp y exacta: 189.693
Minimo alejamiento porcentual entre grasp y exacta: 0
Maximo alejamiento porcentual entre grasp y exacta: 789.400
```

Estas pruebas fueron resueltas con el objetivo de analizar como varián las soluciones de la heurística seguń los caminos factibles que delimita la cota K. A continuación, fijaremos la cota K en 120 (que fue la que mostró soluciones más interesates 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.

5.6.7. 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. Nuestra suposición es que a medida que se reduce el parámetro β, la heurística golosa aleatoria se acerca a la heurística golosa determin 'istica, la cual mostró un gran nivel de optimalidad de solucíon. En los casos que no provee la solución exacta, mientras más pequeño el β más se acerca a la óptima, ya que es más cercana a la determinística y la búsqueda local puede refinar luego la solución parcial obtenida.

Para un experimento en particular de $\beta = n/128$, vemos que iguala en cantidad de aciertos a la heurística golosa y reduce el alejamiento de los no aciertos, lo cual fue interesante de notar, y suponemos se da gracias al refinamiento posterior que produce la búsqueda local.

- 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
```

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
```

$\beta = 0{,}7$

```
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
```

$\beta = 0{,}4$

```
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
```

Análoga a la anterior experimentación, vemos como también reducir el valor lmite de la RCL y por lo tanto acercándose a la solución golosa, causa que no sólo produzca más aciertos sino que también sus no aciertos se ajustan más al valor óptimo.

5.6.8. Conclusión

El análisis realizado nos da la pauta de que GRASP tiene un comportamiento muy variable dominado por sus parámetros, en muchos casos, en el mejor caso la mejor solución que puede proveer es igual a la de la heurística golosa. Pero en otros casos particulares, como RCL por cantidad y $\beta = n/128$ notamos que se acerca al comportamiento goloso pero hace uso de sus beneficios como el refinamiento de sus soluciones mediante búsqueda local y las sucesivas iteraciones para obtener una mejor solución, para poder acertar la misma cantidad de veces que la heurística golosa y en los casos en que no acierta puede ajustar aun más la solución y acercarse a la óptima. En conclusión vemos que es una ventaja reducir el tamaño de la RCL porque de esa forma se acerca al comportamiento goloso que nos dio excelentes resultados, pero también tiene sus beneficios agregados. Consideramos de que es una metaheurística con mucho potencial pero que el ajuste de sus parámetros en relación a las instancias de ejecución no debe ser tomado a la ligera y se deben realizar sucesivos experimentos y variaciones para llegar a un resultado notable y en casos superior a la golosa.

6. Experimentacion General

En esta sección analizaremos la calidad de las heuristicas mediante la comparación y analisis estadistico de las soluciones, asi tambien 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 parametros

- cantidad de nodos
- cantidad de aristas
- peso minimo w_1
- peso maximo w_1
- peso minimo w_2
- peso maximo w_2
- limite w_1

y devuelve por salida estandar una instancia del problema tal como esta especificado el formato de entrada en el enunciado de este TP.
Nota: Los generadores de numeros aleatorios de este generador de grafos tienen distribucion uniforme(Usan random de C++11).

La generación del grafo se realiza de la siguiente manera: Sean n={cantidad de nodos del grafo} y m={cantidad de aristas}, se inicializa un vector $aristas =< (0,1), ..., (0, n-1), (1,2), ..., (1, n-1), ..., (n-2, n-1) >$ conteniendo todas las aristas posibles en el grafo(notar que como no es digrafo, no se repiten aristas simetricas, ni tampoco de 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 estandar conteniendo los nodos origen, destino y el parametro k generados como numeros aleatorios uniformes. Ahora basta tomar la cantidad m de aristas requeridas por parámetro y serializar la salida linea por linea, nuevamente generando numeros aleatorios con distribucion 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 realizo un script el cual genera conjuntos de grafos usando el generador de la seccion anterior, basicamente, se setean dos rangos, de cantidad de nodos y cantidad de aristas, y los parametros fijos como limite w_1, limites 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 funcion 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 estadisticas

Se realizaron diversos scripts para automatizar el analisis 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 estadisticos de los resultados obtenidos, tanto de la solucion como del tiempo consumido para obtenerla. Los analisis estadisticos que se realizan son:

- Tiempo promedio microsegundos consumido por el algoritmo
- Porcentaje de veces que la heuristica da la solucion optima
- Desviacion estandar de veces que la heuristica da la sol. optima
- Lejanía promedio de la solucion obtenida a la solucion optima
- Desviacion estandar de la lejanía de las soluciones entre la obtenida y la optima
- Minima y maxima lejanía obtenida en este conjunto de instancias

Nota: La lejanía entre 2 soluciones se mide haciendo el siguiente calculo: $100 * (\frac{solucionHeuristica}{solucionOptima} - 1)$ que indica en porcentaje cual es el ratio de distancia entre los dos valores del cociente.
Nota: Los calculos estadisticos (promedio y desviacion estandar) se realizan sobre la lista de resultados obtenida de la ejecucion secuencial y el calculo de la lejania mencionado aqui arriba para cada uno de los algoritmos(exacto y heuristica) sobre cada instancia del conjunto de pruebas.

Podemos considerar una especie de puntuacion asignada a cada heuristica viendo el porcentaje de optimalidad y tambien podemos analizar, que cuando no da la solucion optima, la lejania a la optima, con cierta dispersion sea adecuada, segun el problema y el las necesidades del contexto donde se aplica.

6.3. Scripts de optimalidad - Graficos de optimalidad comparativos

El script de optimalidad tambien realiza un grafico, 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 variacion 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 grafico podemos apreciar la distribucion de las soluciones sobre el eje Y a medida que varía la cantidad de nodos.
Referencias del gráfico: Los triangulos azules representan las soluciones de busqueda 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 heuristicas 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 analisis estadístico correspondiente, a continuacion se presentan los resultados.

6.4.1. Comparacion Exacta-Golosa-Busqueda Local-GRASP

Se eligieron, 3 diferentes densidades de grafos, y se corrieron los algoritmos con los scripts mencionados anteriormente, a continuacion presentamos los resultados estadisticos, asi tambien 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%
```

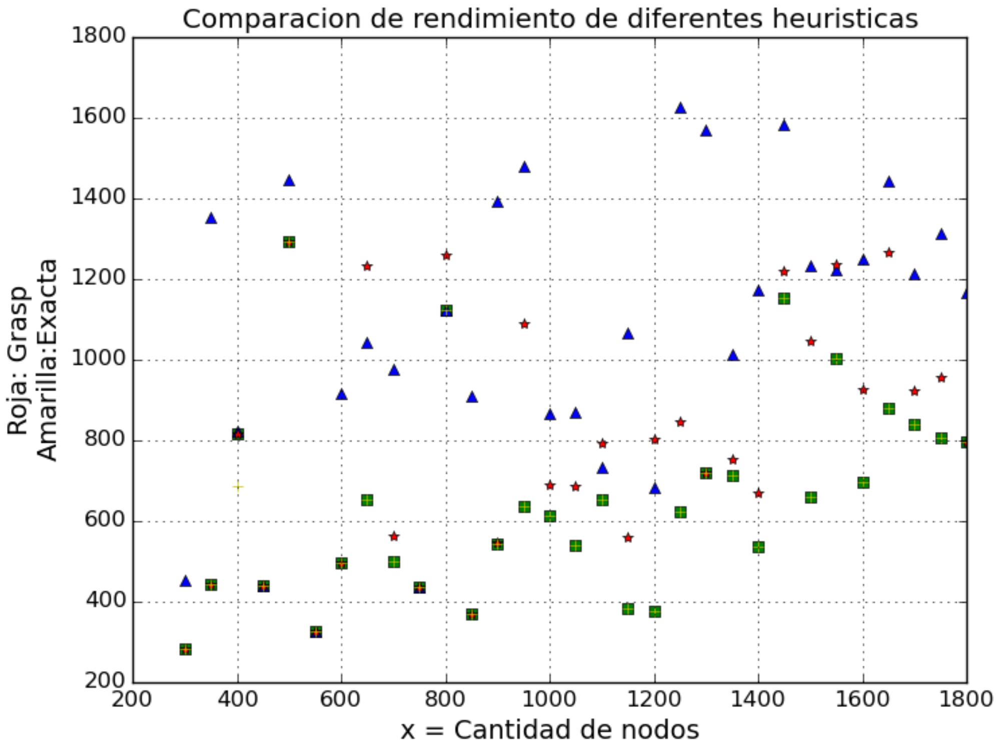
Resultados del analisis (Busqueda local):

```
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%
```

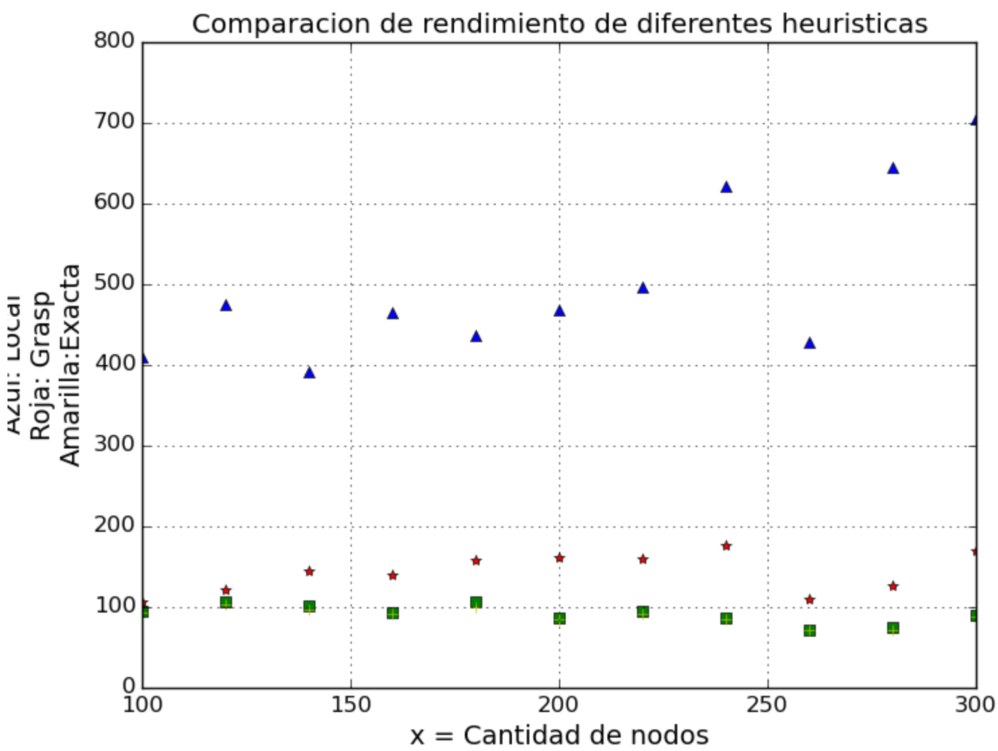
Resultados del analisis (Busqueda local):

```
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%
```

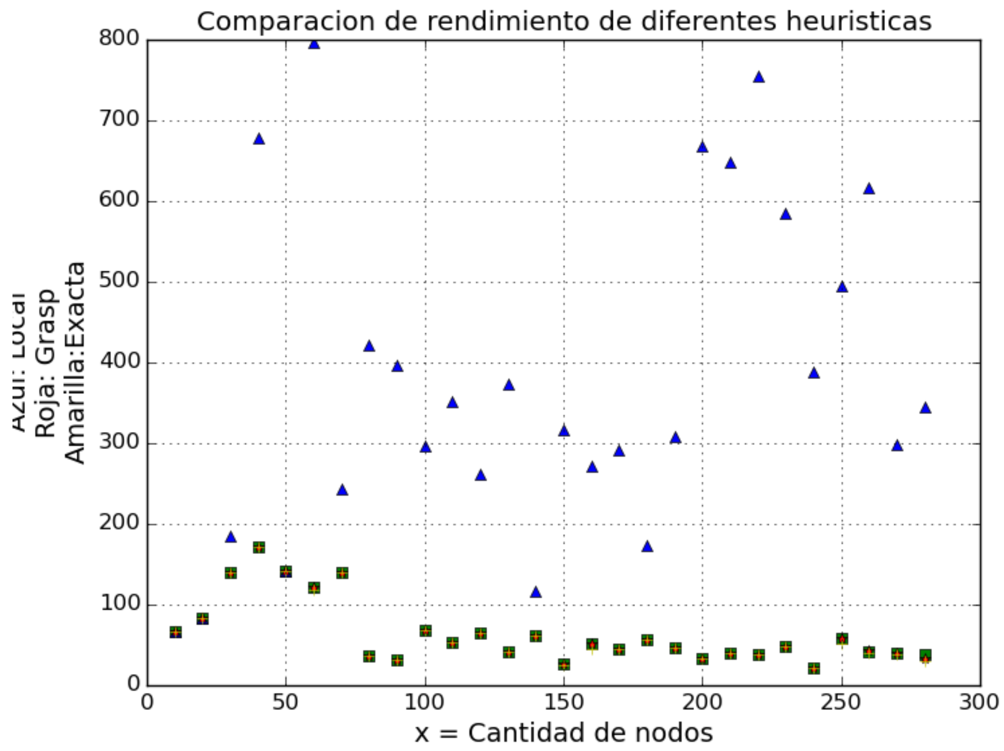
Resultados del analisis (Busqueda local):

```
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 distribucion 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 dificil de los casos de prueba que consideramos, pensamos que los resultados son muy buenos para las aplicaciones prácticas.

Algoritmo de Busqueda local para grafos de baja densidad:
Respecto a la heuristica de busqueda local, observamos un 30 % de efectividad, con una lejanía promedio de 87.62 % y una desv. estandar en la lejanía al optimo de 118.5 %, consideramos que 30 % es un numero muy bajo y junto a los indicadores de lejanía, si nos interesa realmente una solucion cercana a la óptima no elegiriamos esta heurística(al menos no con dijkstra sobre w_1 como solución inicial. Ver Experimentacion de la solucion inicial en la sección busqueda local). Como contraparte, toma aproximadamente la mitad de tiempo que el algoritmo exacto(recordemos que la medicion es por iteracion y para grafos lineales hay 1 iteracion promedio.), no consideramos que sea un buen tradeoff, la mitad del tiempo de ejecución para esta calidad de soluciones.

Algoritmo de Busqueda 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 atribuiamos a una vecindad de mayor cardinal, pero cuando comparamos optimalidad promedio de la heurística, nos da un numero bajísimo, 10 % de aciertos a la solución optima, 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 iteracion) 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 Busqueda 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 ejecucion del algoritmo exacto a 19571 microsegundos y la cantidad de iteraciones promedio de la búsqueda local a 5 iteraciones promedio con un tiempo por iteracion de 550 microsegundos. Veamos entonces que el algoritmo de búsqueda local se ejecuta aproximadamente 7 veces mas rapido 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` numero de aciertos a la solución optima y la enorme lejanía al optimo.

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

Algoritmo de Metaheuristica GRASP para grafos de baja densidad:
Notamos como primer observacion que la metaheuristica 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 lejania ni la dispersion de la lejanía, en menos tiempo podemos obtener la solucion exacta.

Algoritmo de Metaheuristica GRASP para grafos de densidad intermedia:
Observamos que la heuristica no es muy efectiva, con una alta lejania y dispersion de lejania respecto

a la solucion optima, a pesar de que el tiempo que toma sea la mitad que el algoritmo exacto, no lo utilizariamos como heuristica para este tipo de grafos.

Algoritmo de Metaheuristica 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 solucion optima que el algoritmo goloso, y aun mejor, la lejanía promedio y dispersion de esta, se disminuyeron respecto a la solución. Para este tipo de grafos, con alta densidad, encontramos que con este parametro beta, ocurrian este tipo de mejoras, si bien esto es el resultado de una muestra de toda la familia de grafos de alta densidad, podriamos considerar aplicar esta heuristica aun mas que la heuristica golosa para resolver el problema de manera aproximada.

6.4.5. Gráficos de distribucion de las soluciones

Distribución de soluciones para grafos de alta densidad:
Podemos observar que las soluciónes de búsqueda local siempre estan 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 busqueda local estan sutilmente mas cerca de los demas algoritmos, igual siguen estando relativamente lejos, respecto a las otras heuristicas(Golosa y GRASP) comienzan a verse su "despegue"de la solución óptima, mas que nada en la metaheuristica GRASP.

Distribución de soluciones para grafos de baja densidad:
Finalmente aqui observamos que la solución de busqueda 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 indice de aciertos respecto al algoritmo exacto.

6.5. Experimentacion pura entre heuristicas

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 solucion optima(que siempre es menor o igual al minimo de las 3 heuristicas, por definicion de solucion optima).

Esto se realizara, para cada ejecucion de las 3 heuristicas, incrementando un contador en la que corresponda si su peso w_2 es minimo, al final tendremos, para cada heuristica, cuantas veces dio la minima solucion, y la cantidad total de instancias testeadas, con estos datos podremos realizar los analisis estadisticos.

6.5.1. Grafos aleatorios de alta densidad de aristas

Parametros del experimento:

- Parametro Beta GRASP: $\frac{cantNodos}{128}$
- Cantidad de grafos analizados: 17
- Cantidad minima 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:

```
Porcentaje de instancias donde da el minimo w2:
        Golosa: 100%
        Busqueda local: 0%
        Grasp: 0%
```

Vemos que la heuristica 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 heuristica funciona mejor con alta densidad de grafos dado que esta relacionada la densidad del grafo con el cardinal de la vecindad. La metaheuristica GRASP disminuye su rendimiento a medida que decrece la densidad de los grafos, suponemos que se debe a que la heuristica golosa randomizada, a lo sumo da tan bien como la golosa pura, y la busqueda local sobre golosa no mejora la solucion, finalmente la heuristica 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 heuristicas, 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 heuristica, analizar las variaciones absolutas y relativas de la función target a medida que avanzan las iteraciones de ciertas heuristicas, 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 heuristicas dan la solucion optima y que tan lejos de ella estan junto con la comparacion del tiempo de ejecucion entre el algoritmo de la heuristica y el algoritmo de solucion exacta, y el segundo, entre las 3 heuristicas, para los valores máximos de nodos y densidad que pudimos evaluar con el hardware con el que contamos, con que porcentaje cada heuristica 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 heuristica para solucionar el problema de forma aproximada, elegiriamos la heuristica 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 heuristica de búsqueda local alimentada con dijkstra sobre w_1 provee soluciones lejanas al optimo, la utilizariamos solo para refinar soluciones factibles de buena calidad ya obtenidas, no como heuristica para solucionar el problema. La metaheuristica 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 metaheuristica 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 heuristica 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(generador, 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 heuristicas unicamente para instancias mas grandes`

A rasgos mas generales, las secciones de heuristica golosa, heuristica de busqueda local, metaheuristica 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

```
int main(int argc, char** argv){
    int cant_nodos = 0;
    int cant_aristas = 0;
    int pesomin_w1 = 0;
    int pesomax_w1 = 0;
    int pesomin_w2 = 0;
    int pesomax_w2 = 0;
    int limit_w1 = 0;

    if(argc != 8){
        cout << "Use:␣bin␣<cant_nodos>␣<cant_aristas>␣<pesomin_w1>␣<pesomax_w1>␣<pesomin_w2>␣<pesomax_w2>␣<limit_w1>␣" << endl;
        return 1;
    }else{
        cant_nodos = atoi(argv[1]);
        cant_aristas = atoi(argv[2]);
        pesomin_w1 = atoi(argv[3]);
        pesomax_w1 = atoi(argv[4]);
        pesomin_w2 = atoi(argv[5]);
        pesomax_w2 = atoi(argv[6]);
        limit_w1 = atoi(argv[7]);
    }
    int aristas_maximas = cant_nodos*(cant_nodos-1)/2;

    //validacion maximas aristas
    if(cant_aristas > aristas_maximas){
        cant_aristas = aristas_maximas;
    }

    //inicializo en <(0, 1), ..., (0, n-1), (1, 2), ..., (1, n-1), ..., (n - 2, n-1)>
    vector<pair<int, int> > aristas;
    for(int i=0;i<cant_nodos;i++){
        for(int j=i+1;j<cant_nodos;j++){
            aristas.push_back(make_pair(i, j));
        }
    }

    //ahora shufleo uniformemente con fisher yates
    //http://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle

    random_device rd;
    mt19937 gen(rd());

    uniform_int_distribution<int> dis_srcdst(1, cant_nodos);
    int nodo_src = dis_srcdst(gen);
    int nodo_dst = dis_srcdst(gen);

    //header line: nodos aristas src dst limitw1
    cout << cant_nodos << "␣" << cant_aristas << "␣" << nodo_src << "␣" << nodo_dst << "␣" << limit_w1 << endl;

    int permutations_number = aristas_maximas;//C(cant_nodos, 2)
    vector<pair<int, int> >::iterator begin_iter = aristas.begin();
    for(int i=0;i<permutations_number;i++){
        uniform_int_distribution<int> dis(i, permutations_number);
        int x = dis(gen);
        iter_swap(begin_iter + i, begin_iter + x);
    }

    //ahora agarro del vector randomizado de aristas, las primeras cant_aristas y las imprimo con pesos aleatorios entre
    aristas.resize(cant_aristas);

    uniform_int_distribution<int> dis_w1(pesomin_w1, pesomax_w1);
    uniform_int_distribution<int> dis_w2(pesomin_w2, pesomax_w2);
    for(auto arista : aristas){
        //notemos que la salida es entre 1..n!!
        int partida = arista.first + 1;
        int destino = arista.second + 1;
        int peso_w1 = dis_w1(gen);
        int peso_w2 = dis_w2(gen);
        cout << partida << "␣" << destino << "␣" << peso_w1 << "␣" << peso_w2 << endl;
```

```
    }
    cout << "0";
}
```

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

```
#include <stdio.h>
#include <iostream>
#include <string.h>
#include <math.h>
#include "grafo.h"
#include "parser.h"
#include "timing.h"

#define CANT_ITERS_MEDICION 1

using namespace std;

typedef struct peso{
    int w1;
    int w2;
    int otra_cosa;
} peso;

typedef struct Solucion{
    float W1;
    float W2;
    int k;
    int *v;
} Solucion;

Solucion solucion_mejor;

void setW1(void *p, float w)
{
    ((peso *)p)->w1 = w;
}
void setW2(void *p, float w)
{
    ((peso *)p)->w2 = w;
}

bool resolver(Grafo<peso> &g, int u, int w, int K, float *distancia_w1, float *distancia_w2,
    Solucion *solucion)
{
    Solucion solucion_nueva;
    int cantidad_aristas, *adyacentes, i;
    peso *p;
    float w2, w1;
    bool solucionado = false;

    solucion_nueva.W1 = 0;
    solucion_nueva.W2 = 0;
    solucion_nueva.k = 0;

    if(u == w){
        if(solucion_mejor.W2 < 0 || solucion_mejor.W2 > solucion->W2){
```

```
            solucion_mejor.W1 = solucion->W1;
            solucion_mejor.W2 = solucion->W2;
            solucion_mejor.k = solucion->k;
            memcpy(solucion_mejor.v, solucion->v, 2 * g.CantidadAristas() * sizeof(int));
            return true;
        }
        return false;
    }
    g.MarcarNodo(u, true);
    adyacentes = g.Adyacentes(u, &cantidad_aristas);
    for(i = 0; i < cantidad_aristas; i++){
        if(g.VerMarcaDeNodo(adyacentes[i]) == false){
            p = g.Peso(u, adyacentes[i]);
            w1 = p->w1;
            w2 = p->w2;
            if(distancia_w1[adyacentes[i]] + w1 <= K && (solucion_mejor.k == 0 || solucion->W2
                + w2 + distancia_w2[adyacentes[i]] < solucion_mejor.W2)){
                g.QuitarArista(u, adyacentes[i]);
                solucion_nueva.W1 = solucion->W1 + w1;
                solucion_nueva.W2 = solucion->W2 + w2;
                solucion_nueva.v = solucion->v;
                solucion_nueva.v[solucion->k] = adyacentes[i];
                solucion_nueva.k = solucion->k + 1;
                if(resolver(g, adyacentes[i], w, K - w1, distancia_w1, distancia_w2, &
                    solucion_nueva)){
                    solucionado = true;
                }
                g.AgregarArista(u, adyacentes[i], p);
            }
        }
    }
    g.MarcarNodo(u, false);
    if(solucionado){
        solucion->W1 = solucion_mejor.W1;
        solucion->W2 = solucion_mejor.W2;
        memcpy(solucion->v, solucion_mejor.v, solucion_mejor.k * sizeof(int));
        solucion->k = solucion_mejor.k;
    }
    free(adyacentes);
    return solucionado;
}

void dijkstra(Grafo<peso> *g, int nodo, float **_distancia_w1, float **_distancia_w2)
{
    int cantidad_nodos, nodo_minimo, cantidad_adyacentes, *adyacentes, cantidad_vistos;
    int i;
    float *distancia_w1, *distancia_w2;
    float peso_minimo;
    bool *nodos_vistos;

    if(g == NULL || _distancia_w1 == NULL || _distancia_w2 == NULL)
        return;

    cantidad_nodos = g->CantidadNodos();
    if(nodo < 1 || nodo > cantidad_nodos)
        return;

    nodos_vistos = new bool[cantidad_nodos + 1];
    *_distancia_w1 = new float[cantidad_nodos + 1];
    *_distancia_w2 = new float[cantidad_nodos + 1];
    distancia_w1 = *_distancia_w1;
    distancia_w2 = *_distancia_w2;

    for(i = 1; i <= cantidad_nodos; i++){
        distancia_w1[i] = INFINITY;
        distancia_w2[i] = INFINITY;
        nodos_vistos[i] = false;
    }
    distancia_w1[nodo] = 0;
    distancia_w2[nodo] = 0;
    cantidad_vistos = 0;

    while(cantidad_vistos < cantidad_nodos){
        peso_minimo = INFINITY;
        nodo_minimo = 1;
        for(i = 1; i <= cantidad_nodos; i++){
            if(!nodos_vistos[i] && distancia_w1[i] < peso_minimo){
```

```
                nodo_minimo = i;
                peso_minimo = distancia_w1[i];
            }
        }
        nodos_vistos[nodo_minimo] = true;
        cantidad_vistos++;
        adyacentes = g->Adyacentes(nodo_minimo, &cantidad_adyacentes);
        for(i = 0; i < cantidad_adyacentes; i++){
            if(!nodos_vistos[adyacentes[i]] && distancia_w1[adyacentes[i]] > distancia_w1[nodo_minimo] + g->Peso(nodo_minimo, adyacentes[i])->w1){
                distancia_w1[adyacentes[i]] = distancia_w1[nodo_minimo] + g->Peso(nodo_minimo, adyacentes[i])->w1;
            }
        }
        free(adyacentes);
    }

    for(i = 1; i <= cantidad_nodos; i++){
        nodos_vistos[i] = false;
    }
    cantidad_vistos = 0;

    while(cantidad_vistos < cantidad_nodos){
        peso_minimo = INFINITY;
        nodo_minimo = 1;
        for(i = 1; i <= cantidad_nodos; i++){
            if(!nodos_vistos[i] && distancia_w2[i] < peso_minimo){
                nodo_minimo = i;
                peso_minimo = distancia_w2[i];
            }
        }
        nodos_vistos[nodo_minimo] = true;
        cantidad_vistos++;
        adyacentes = g->Adyacentes(nodo_minimo, &cantidad_adyacentes);
        for(i = 0; i < cantidad_adyacentes; i++){
            if(!nodos_vistos[adyacentes[i]] && distancia_w2[adyacentes[i]] > distancia_w2[nodo_minimo] + g->Peso(nodo_minimo, adyacentes[i])->w2){
                distancia_w2[adyacentes[i]] = distancia_w2[nodo_minimo] + g->Peso(nodo_minimo, adyacentes[i])->w2;
            }
        }
        free(adyacentes);
    }

    delete [] nodos_vistos;

}

void comenzar(Grafo<peso> *grafo, int u, int v, int K, int nodos, int aristas)
{
    int i;
    Solucion solucion;
    float *distancia_w1_v;
    float *distancia_w2_v;

    solucion_mejor.W1 = -1;
    solucion_mejor.W2 = -1;
    solucion_mejor.k = 0;
    solucion_mejor.v = (int *)calloc(2 * grafo->CantidadAristas(), sizeof(int));
    solucion.W1 = 0;
    solucion.W2 = 0;
    solucion.k = 1;
    solucion.v = (int *)calloc(2 * aristas, sizeof(int));
    solucion.v[0] = u;
    dijkstra(grafo, v, &distancia_w1_v, &distancia_w2_v);
    if(distancia_w1_v[u] > K){
        printf("no");
    }
    else if(resolver(*grafo, u, v, K, distancia_w1_v, distancia_w2_v, &solucion)){
        printf("%.0f %.0f %d", solucion.W1, solucion.W2, solucion.k);
        for(i = 0; i < solucion.k; i++){
            printf(" %d", solucion.v[i]);
        }
        printf("\n");
    }
    else{
        printf("no");
```

```
    }
    free(solucion.v);
    free(solucion_mejor.v);
    delete [] distancia_w1_v;
    delete [] distancia_w2_v;

}

int main(int argc, char **argv)
{
    int medir_tiempo = 0;
    GrafoAdyacencia<peso> *grafo;
    peso **pesos;
    int u, v, K, nodos, aristas;

    //if(argc != 1)
    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.
    grafo = new GrafoAdyacencia<peso>(0);
    while(Parsear<peso>(*grafo, stdin, &pesos, setW1, setW2, &u, &v, &K, &nodos, &aristas)){
        if(medir_tiempo){
            double promedio_medicion = 0.0;
            MEDIR_TIEMPO_PROMEDIO(
                comenzar(grafo, u, v, K, nodos, aristas);
            , CANT_ITERS_MEDICION, &promedio_medicion);
            //cerr << promedio << " " << nodos << " " << aristas << endl;
            cerr << nodos << " " << aristas << " " << CANT_ITERS_MEDICION << " " <<
                promedio_medicion;
        }
        else{
            comenzar(grafo, u, v, K, nodos, aristas);
        }
        int i;
        for(i = 0; i < aristas; i++){
            delete pesos[i];
        }
        free(pesos);
        delete grafo;
        grafo = new GrafoAdyacencia<peso>(0);
    }
    delete grafo;
    return 0;
}
```

11.6. Heuristicas

```
//#define DEBUG_MESSAGES_ON
//#define CYCLE_PREVENT_MESSAGE_ON
//#define VECINOS_COMUNES_LAZY

using namespace std;

typedef int nodo_t;
typedef int longuitud_t;
typedef double distancia_t;
typedef double costo_t;
typedef enum tipo_costo_t {COSTO_W1, COSTO_W2} tipo_costo_t;
typedef enum tipo_ejecucion_golosa_t {RCL_DETERMINISTICO, RCL_POR_VALOR, RCL_POR_CANTIDAD}
    tipo_ejecucion_golosa_t;
typedef enum tipo_ejecucion_bqlocal_t {BQL_SUBDIVIDIR_PARES, BQL_CONTRAER_TRIPLAS_A_PARES,
    BQL_MEJORAR_CONEXION_TRIPLAS, BQL_COMBINAR} tipo_ejecucion_bqlocal_t;
typedef enum criterio_terminacion_grasp_t {CRT_K_ITERS_SIN_MEJORA, CRT_K_ITERS_LIMIT_REACHED}
    criterio_terminacion_grasp_t;
typedef enum formato_entrada_t {FORMATO_0_N_OPEN, FORMATO_1_N_CLOSED} formato_entrada_t;
typedef formato_entrada_t formato_salida_t;

const costo_t costo_infinito = numeric_limits<double>::infinity();
const distancia_t distancia_infinita = numeric_limits<double>::infinity();
const costo_t costo_nulo = 0;
const nodo_t predecesor_nulo = -1;

class Arista{
private:
    bool presente;
    costo_t costo_w1;
    costo_t costo_w2;
public:
```

```
    Arista();
    Arista(bool pres, costo_t w1, costo_t w2);
    ~Arista();

    bool esta_presente();
    void desmarcar_presente();
    void marcar_presente(costo_t w1, costo_t w2);
    costo_t obtener_costo_w1() const;
    costo_t obtener_costo_w2() const;

    bool operator== (const Arista &other) const
    {
        return (other.presente == this->presente &&
                other.costo_w1 == this->costo_w1 &&
                other.costo_w2 == this->costo_w2);
    }
};

typedef vector<vector<Arista> > matriz_adyacencia_t;
typedef list<pair<nodo_t, Arista> > lista_adyacentes;
typedef vector<lista_adyacentes> lista_adyacencia_t;

class Vecino{
private:
    nodo_t i;
    nodo_t j;
    nodo_t en_comun;
    Arista desde_i_a_comun;
    Arista desde_j_a_comun;
public:
    Vecino(nodo_t i, nodo_t j, nodo_t comun, Arista desde_i, Arista desde_j);
    Vecino();
    ~Vecino();
    nodo_t obtener_nodo_i();
    nodo_t obtener_nodo_j();
    nodo_t obtener_nodo_comun();
    Arista obtener_arista_i_comun();
    Arista obtener_arista_j_comun();
};

typedef vector<vector<vector<Vecino> > > vecinos_comunes_t;

class Camino{
private:
    list<nodo_t> camino;
    vector<bool> esta_en_camino;
    costo_t costo_camino_w1;
    costo_t costo_camino_w2;
    matriz_adyacencia_t mat_adyacencia;
public:
    //Camino();
    Camino(matriz_adyacencia_t& mat_adyacencia);
    ~Camino();

    void agregar_nodo(nodo_t target);
    void agregar_nodo_adelante(nodo_t target);
    costo_t obtener_costo_total_w1_camino();
    costo_t obtener_costo_total_w2_camino();
    //pre: 0 <= i <= j < cantidad_nodos y que i,j sean adyacentes
    costo_t obtener_costo_w1_entre_nodos(nodo_t i, nodo_t j);
    //pre: 0 <= i <= j < cantidad_nodos y que i,j sean adyacentes
    costo_t obtener_costo_w2_entre_nodos(nodo_t i, nodo_t j);

    void imprimir_camino(ostream& out);

    list<nodo_t>::iterator obtener_iterador_begin();
    list<nodo_t>::iterator obtener_iterador_end();
    list<nodo_t>::const_iterator obtener_iterador_const_begin();
    list<nodo_t>::const_iterator obtener_iterador_const_end();
    longuitud_t obtener_longuitud_camino();

    //pre: at.obtener_nodo_i() y at.obtener_nodo_i() deben pertenecer al camino
    //Se reemplazara la conexion directa entre i y j por i -> encomun -> j indicado por el Vecino pasado
    //por parametro. Devuelve true si se inserto, false sino.
    bool insertar_nodo(Vecino& at);

```

```
    //pre: at.obtener_nodo_i() y at.obtener_nodo_i() deben pertenecer al camino
    //Se reemplazara la conexion i..loquesea..j por i -> encomun -> j indicado por el Vecino
        pasado
    //por parametro. Devuelve true si se inserto, false sino.
    bool mejorar_tripla(Vecino& at);

    bool realizar_salto_entre_3_nodos(nodo_t punto_de_salto);
    bool pertenece_a_camino(nodo_t target);//O(1)
};

class Grafo{
private:
    //atributos
    matriz_adyacencia_t mat_adyacencia;
    vecinos_comunes_t vecinos_comunes;
    lista_adyacencia_t lista_adyacencia;
    int cantidad_nodos;
    int cantidad_aristas;

    //atributos propios del problema
    nodo_t nodo_src;
    nodo_t nodo_dst;
    costo_t cota_w1;
    Camino camino_obtenido;
    bool sol_valida;
    //dado el camino, podemos obtener los pesos de cada "salto" indexando en la matriz de
        adyacencia el costo de cada salto
    //tanto para w1 como w2

    //metodos auxiliares
    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,
     Vecino& mejor_vecino);

    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,
     Arista& mejor_vecino);

    void precalcular_adyacentes_en_comun(nodo_t i, nodo_t j);

    //Grasp
    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);

    int busqueda_local_entre_pares_insertando(Camino& solucion_actual, Vecino&
        conexion_ij_minima_w2);
    int busqueda_local_entre_triplas_reemplazando_intermedio(Camino& solucion_actual, Vecino&
        conexion_ij_minima_w2);
    int busqueda_local_entre_triplas_salteando(Camino& solucion_actual, list<nodo_t>::
        const_iterator& punto_de_salto_it);
public:
    //constructor y destructor
    Grafo(int cant_inicial_nodos);
    ~Grafo();

    //Modificadores
    void agregar_nodos(int cantidad_nodos);
    void agregar_arista(nodo_t i, nodo_t j, costo_t w1, costo_t w2);

    //Consultas
    vector<Arista> obtener_vector_fila_vecinos(nodo_t target);
    lista_adyacentes obtener_lista_vecinos(nodo_t target);
    int obtener_cantidad_nodos();
    int obtener_cantidad_aristas();
    Arista obtener_arista(nodo_t i, nodo_t j);
    //pre: 0 <= i <= j < cantidad_nodos y que i,j sean adyacentes
    vector<Vecino> obtener_adyacentes_en_comun(nodo_t i, nodo_t j);
    nodo_t obtener_nodo_origen();
    nodo_t obtener_nodo_destino();
    costo_t obtener_limite_w1();
    costo_t obtener_costo_actual_w1_solucion();
    costo_t obtener_costo_actual_w2_solucion();
    Camino obtener_camino_solucion();
    void establecer_camino_solucion(Camino c);

    //Entrada - Salida
    void imprimir_matriz_adyacencia(ostream& out);
```

```
    void imprimir_lista_adyacencia(ostream& out);
    void serialize(ostream& out, formato_salida_t formato);
    bool unserialize(istream& in, formato_entrada_t formato);

    //Algoritmos
    //Realiza la busqueda local sobre una solucion inicial factible creada por dijkstra sobre COSTO_W1 entre src y dst
    int busqueda_local(tipo_ejecucion_bqlocal_t tipo_ejecucion);
    //Devuelve el camino minimo entre origen y destino(calcula el arbol, pero reconstruye solo el camino de origen a destino)
    Camino dijkstra(nodo_t origen, nodo_t destino, tipo_costo_t target_a_minimizar);
    //Aplica dijkstra desde nodo origen y calcula el arbol de caminos minimos por referencia a los vectores por parametro
    void dijkstra(nodo_t origen, tipo_costo_t target_a_minimizar, vector<costo_t>& costo_minimo, vector<nodo_t>& predecesor);
    //Dado un nodo_t origen se calcula para cada nodo, la distancia minima en cantidad de aristas de peso constante 1 de cualquier nodo a origen
    void breadth_first_search(nodo_t origen, vector<distancia_t>& distancias_en_aristas_a_origen);

    //Golosa
    Camino obtener_solucion_golosa();
    Camino obtener_solucion_golosa_randomizada(tipo_ejecucion_golosa_t tipo_ejecucion, double parametro_beta);

    //Metodos utilitarios
    static list<Grafo> parsear_varias_instancias(formato_entrada_t formato);
    void establecer_se_encontro_solucion(bool se_encontro);
    bool hay_solucion();
    Camino obtener_camino_vacio();
};

// ------------------------------------------------------------------------------------------------


bool Camino::realizar_salto_entre_3_nodos(nodo_t nodo_target){
    list<nodo_t>::iterator runner_it = this->camino.begin();
    list<nodo_t>::iterator final_it = this->camino.end();
    while(runner_it != final_it){
        if(*runner_it == nodo_target){
            break;
        }
        ++runner_it;
    }

    if(runner_it == final_it){
        cerr << "Camino::realizar_salto_entre_3_nodos. No se encontro el nodo target pasado por parametro en el camino" << endl;
        cerr << "Nodo target (" << nodo_target << ")" << endl;
        cerr << "Camino: ";
        this->imprimir_camino(cerr);
        return false;
    }

    //aca vale *runner_it == nodo_target

    nodo_t nodo_i = *runner_it;

    runner_it++;
    list<nodo_t>::iterator deletion_target = runner_it;
    nodo_t nodo_intermedio_viejo = *runner_it;

    runner_it++;
    nodo_t nodo_j = *runner_it;

    costo_t costo_i_intermedio_w1 = obtener_costo_w1_entre_nodos(nodo_i, nodo_intermedio_viejo);
    costo_t costo_i_intermedio_w2 = obtener_costo_w2_entre_nodos(nodo_i, nodo_intermedio_viejo);

    costo_t costo_intermedio_j_w1 = obtener_costo_w1_entre_nodos(nodo_intermedio_viejo, nodo_j);
    costo_t costo_intermedio_j_w2 = obtener_costo_w2_entre_nodos(nodo_intermedio_viejo, nodo_j);

```

```cpp
    costo_t costo_i_j_w1 = obtener_costo_w1_entre_nodos(nodo_i, nodo_j);
    costo_t costo_i_j_w2 = obtener_costo_w2_entre_nodos(nodo_i, nodo_j);

    //elimino el nodo intermedio entre i y j
    this->camino.erase(deletion_target);
    this->esta_en_camino[nodo_intermedio_viejo] = false;

    //actualizo los costos del camino
    this->costo_camino_w1 = (this->costo_camino_w1 - (costo_i_intermedio_w1 +
        costo_intermedio_j_w1) + costo_i_j_w1);
    this->costo_camino_w2 = (this->costo_camino_w2 - (costo_i_intermedio_w2 +
        costo_intermedio_j_w2) + costo_i_j_w2);

    return true;
}

bool Camino::pertenece_a_camino(nodo_t target){
    return this->esta_en_camino[target];
}

//pre: at.obtener_nodo_i() y at.obtener_nodo_i() deben pertenecer al camino
//Se reemplazara la conexion i..loquesea..j por i -> encomun -> j indicado por el Vecino pasado
//por parametro. Devuelve true si se inserto, false sino.
bool Camino::mejorar_tripla(Vecino& at){
    nodo_t nodo_target = at.obtener_nodo_i();
    nodo_t nodo_sig_sig_target = at.obtener_nodo_j();
    nodo_t nodo_intermedio = at.obtener_nodo_comun();
    costo_t i_comun_w1 = at.obtener_arista_i_comun().obtener_costo_w1();
    costo_t i_comun_w2 = at.obtener_arista_i_comun().obtener_costo_w2();
    costo_t j_comun_w1 = at.obtener_arista_j_comun().obtener_costo_w1();
    costo_t j_comun_w2 = at.obtener_arista_j_comun().obtener_costo_w2();
    nodo_t intermedio_viejo = -1;

    list<nodo_t>::iterator insertion_target = this->camino.begin();
    list<nodo_t>::iterator final_it = this->camino.end();
    while(insertion_target != final_it){
        if(*insertion_target == nodo_target){
            insertion_target++;
            intermedio_viejo = *insertion_target;
            break;
        }
        ++insertion_target;
    }

    if(insertion_target == final_it){
        cerr << "Camino::mejorar_tripla. No se encontro el nodo target pasado por parametro en
            el camino" << endl;
        return false;
    }

    //aca vale que el iterator it apunta a target + 1

    //elimino el intermedio viejo
    nodo_t nodo_intermedio_viejo = *insertion_target;
    insertion_target = this->camino.erase(insertion_target);
    this->esta_en_camino[nodo_intermedio_viejo] = false;

    //inserto el nodo en el medio
    this->camino.insert(insertion_target, nodo_intermedio);
    this->esta_en_camino[nodo_intermedio] = true;

    //actualizo los costos
    //resto el costo entre i y j y sumo las 2 aristas nuevas

    costo_t costo_ij_w1 = obtener_costo_w1_entre_nodos(nodo_target, intermedio_viejo);
    costo_ij_w1 += obtener_costo_w1_entre_nodos(intermedio_viejo, nodo_sig_sig_target);

    costo_t costo_ij_w2 = obtener_costo_w2_entre_nodos(nodo_target, intermedio_viejo);
    costo_ij_w2 += obtener_costo_w2_entre_nodos(intermedio_viejo, nodo_sig_sig_target);

    costo_t costo_i_comun_j_w1 = i_comun_w1 + j_comun_w1;
    costo_t costo_i_comun_j_w2 = i_comun_w2 + j_comun_w2;

    this->costo_camino_w1 = (this->obtener_costo_total_w1_camino() - costo_ij_w1 +
        costo_i_comun_j_w1);
    this->costo_camino_w2 = (this->obtener_costo_total_w2_camino() - costo_ij_w2 +
        costo_i_comun_j_w2);
```

```cpp

    return true;
}

//pre: at.obtener_nodo_i() y at.obtener_nodo_i() deben pertenecer al camino
//Se reemplazara la conexion directa entre i y j por i -> encomun -> j indicado por el Vecino pasado
//por parametro
bool Camino::insertar_nodo(Vecino& at){
    nodo_t nodo_target = at.obtener_nodo_i();
    nodo_t nodo_sig_target = at.obtener_nodo_j();
    nodo_t nodo_intermedio = at.obtener_nodo_comun();
    costo_t i_comun_w1 = at.obtener_arista_i_comun().obtener_costo_w1();
    costo_t i_comun_w2 = at.obtener_arista_i_comun().obtener_costo_w2();
    costo_t j_comun_w1 = at.obtener_arista_j_comun().obtener_costo_w1();
    costo_t j_comun_w2 = at.obtener_arista_j_comun().obtener_costo_w2();

    list<nodo_t>::iterator insertion_target = this->camino.begin();
    list<nodo_t>::iterator final_it = this->camino.end();
    while(insertion_target != final_it){
        if(*insertion_target == nodo_target){
            insertion_target++;
            break;
        }
        ++insertion_target;
    }

    if(insertion_target == final_it){
        cerr << "Camino::insertar_nodo. No se encontro el nodo target pasado por parametro en el camino" << endl;
        return false;
    }

    //aca vale que el iterator it apunta a target + 1

    //inserto el nodo en el medio
    this->camino.insert(insertion_target, nodo_intermedio);
    this->esta_en_camino[nodo_intermedio] = true;

    //actualizo los costos
    //resto la arista entre i y j y sumo las 2 aristas nuevas
    costo_t costo_ij_w1 = obtener_costo_w1_entre_nodos(nodo_target, nodo_sig_target);
    costo_t costo_ij_w2 = obtener_costo_w2_entre_nodos(nodo_target, nodo_sig_target);

    costo_t costo_i_comun_j_w1 = i_comun_w1 + j_comun_w1;
    costo_t costo_i_comun_j_w2 = i_comun_w2 + j_comun_w2;

    this->costo_camino_w1 = (this->obtener_costo_total_w1_camino() - costo_ij_w1 + costo_i_comun_j_w1);
    this->costo_camino_w2 = (this->obtener_costo_total_w2_camino() - costo_ij_w2 + costo_i_comun_j_w2);
    return true;
}
// --------------- Vecino -----------------------------------

Vecino::Vecino(nodo_t i, nodo_t j, nodo_t comun, Arista desde_i, Arista desde_j){
    this->i = i;
    this->j = j;
    this->en_comun = comun;
    this->desde_i_a_comun = desde_i;
    this->desde_j_a_comun = desde_j;
}

Vecino::Vecino(){
    this->i = 0;
    this->j = 0;
    this->en_comun = 0;
    //aristas constructor por defecto
}

Vecino::~Vecino(){
}

nodo_t Vecino::obtener_nodo_i(){
    return this->i;
}

```

```
nodo_t Vecino::obtener_nodo_j(){
    return this->j;
}

nodo_t Vecino::obtener_nodo_comun(){
    return this->en_comun;
}

Arista Vecino::obtener_arista_i_comun(){
    return this->desde_i_a_comun;
}

Arista Vecino::obtener_arista_j_comun(){
    return this->desde_j_a_comun;
}

// ---------------- Grafo ------------------------------------

Camino Grafo::obtener_camino_vacio(){
    Camino c(this->mat_adyacencia);
    return c;
}

Grafo::Grafo(int cant_inicial_nodos) : camino_obtenido(mat_adyacencia) {
    this->cantidad_nodos=cant_inicial_nodos;
    this->cantidad_aristas = 0;
    nodo_src = 0;
    nodo_dst = 0;
    cota_w1 = 0;
    sol_valida = false;

    //inicializo matriz de adyacencia
    vector<Arista> vec_fila(cantidad_nodos, Arista(false, 0, 0));
    this->mat_adyacencia.resize(cantidad_nodos, vec_fila);

    //inicializo lista de adyacencia
    this->lista_adyacencia.resize(cantidad_nodos);

    //inicializo matriz de vecinos comunes
    vector<vector<Vecino> > vec_fila_vecinos(cantidad_nodos);
    this->vecinos_comunes.resize(cantidad_nodos, vec_fila_vecinos);
}

Grafo::~Grafo(){

}

//modificadores
void Grafo::agregar_nodos(int cantidad_nodos_nuevos){
    //agrego cantidad_nodos_nuevos columnas en todas las filas existentes
    for(int i=0;i < this->cantidad_nodos;i++){
        this->mat_adyacencia[i].resize(this->cantidad_nodos + cantidad_nodos_nuevos, Arista(
            false, 0, 0));
    }

    //agrego cantidad_nodos_nuevos filas al final de la matriz de adyacencia
    vector<Arista> vec_fila(this->cantidad_nodos + cantidad_nodos_nuevos, Arista(false, 0, 0));
    this->mat_adyacencia.resize(this->cantidad_nodos + cantidad_nodos_nuevos, vec_fila);

    //redimensiono lista adyacencias
    this->lista_adyacencia.resize(cantidad_nodos + cantidad_nodos_nuevos);

    //redimensiono matriz de vecinos comunes
    vector<vector<Vecino> > vec_fila_vecinos(this->cantidad_nodos + cantidad_nodos_nuevos);
    this->vecinos_comunes.resize(this->cantidad_nodos + cantidad_nodos_nuevos, vec_fila_vecinos
        );

    //actualizo cantidad_nodos total del grafo
    this->cantidad_nodos+=cantidad_nodos_nuevos;
}

void Grafo::agregar_arista(nodo_t i, nodo_t j, costo_t w1, costo_t w2){
    Arista arista = obtener_arista(i, j);
    //marco doble, no es un digrafo
    this->mat_adyacencia[i][j].marcar_presente(w1, w2);
    this->mat_adyacencia[j][i].marcar_presente(w1, w2);

```

```cpp
    this->lista_adyacencia[i].push_back(make_pair(j, obtener_arista(i, j)));
    this->lista_adyacencia[j].push_back(make_pair(i, obtener_arista(i, j)));

    if(!arista.esta_presente()){
        this->cantidad_aristas++;
    }
}

//consultas
Arista Grafo::obtener_arista(nodo_t i, nodo_t j){
    return this->mat_adyacencia[i][j];
}

void Grafo::imprimir_matriz_adyacencia(ostream& out){
    out << "Cantidad nodos: " << this->cantidad_nodos << endl;
    out << "Cantidad aristas: " << this->cantidad_aristas << endl;
    out << "Matriz adyacencia:" << endl;
    for (int i = 0;i<this->cantidad_nodos;i++){
        for (int j = 0;j<this->cantidad_nodos;j++){
            out << "| (" << this->mat_adyacencia[i][j].esta_presente() << ") |";
        }
        out << endl;
    }
    out << endl;
    out << endl;
}

void Grafo::imprimir_lista_adyacencia(ostream& out){
    out << "Cantidad nodos: " << this->cantidad_nodos << endl;
    out << "Cantidad aristas: " << this->cantidad_aristas << endl;
    out << "Lista adyacencia:" << endl;
    for (int i = 0;i<this->cantidad_nodos;i++){
        out << "Vecinos de [" << i << "]: ";
        lista_adyacentes::iterator adyacentes_i_it = this->lista_adyacencia[i].begin();
        lista_adyacentes::iterator final_it = this->lista_adyacencia[i].end();
        while(adyacentes_i_it != final_it){
            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++;
        }
        out << "Nil" << endl;
    }
    out << endl;
    out << endl;
}

costo_t Grafo::obtener_limite_w1(){
    return this->cota_w1;
}

void Grafo::establecer_camino_solucion(Camino c){
    this->camino_obtenido = c;
}

Camino Grafo::obtener_camino_solucion(){
    return this->camino_obtenido;
}

int Grafo::obtener_cantidad_nodos(){
    return this->cantidad_nodos;
}

int Grafo::obtener_cantidad_aristas(){
    return this->cantidad_aristas;
}

void Grafo::serialize(ostream& out, formato_salida_t formato){
    if(this->sol_valida){
        out << this->camino_obtenido.obtener_costo_total_w1_camino() << " ";
        out << this->camino_obtenido.obtener_costo_total_w2_camino() << " ";
        out << this->camino_obtenido.obtener_longuitud_camino() << " ";
        list<nodo_t>::const_iterator it = this->camino_obtenido.obtener_iterador_const_begin();
        while(it != camino_obtenido.obtener_iterador_const_end()){
            out << ( (formato == FORMATO_1_N_CLOSED) ? ( (*it) + 1 ) : *it ) << " ";
            ++it;
        }
```

```
    }else{
        out << "no";
    }
}

bool Grafo::unserialize(istream& in, formato_entrada_t formato){
    //primera linea:
    //n m u v K
    //cant nodos, cant aristas, src, dst, K(cota w1)
    int cant_nodos_nuevos = 0, cantidad_aristas_nuevas = 0;
    in >> cant_nodos_nuevos;

    if(cant_nodos_nuevos == 0){
        return false;//es la ultima linea de la entrada!!
    }

    in >> cantidad_aristas_nuevas;
    in >> this->nodo_src;
    in >> this->nodo_dst;
    in >> this->cota_w1;

    if(formato == FORMATO_1_N_CLOSED){
        this->nodo_src--;
        this->nodo_dst--;
    }

    this->agregar_nodos(cant_nodos_nuevos);

    int count = 0;
    while(count < cantidad_aristas_nuevas){
        //las lineas de las aristas vienen asi:
        // v1 v2 w1 w2
        nodo_t nodo_a = 0, nodo_b = 0; costo_t costo_w1 = 0, costo_w2 = 0;
        in >> nodo_a;
        in >> nodo_b;
        in >> costo_w1;
        in >> costo_w2;

        if(formato == FORMATO_1_N_CLOSED){
            nodo_a--;
            nodo_b--;
        }

        this->agregar_arista(nodo_a, nodo_b, costo_w1, costo_w2);
        count++;
    }

    //precalcular vecinos en comun
    for(int i=0;i<cant_nodos_nuevos;i++){
        for(int j=0;j<cant_nodos_nuevos;j++){
            precalcular_adyacentes_en_comun(i, j);
        }
    }
    return true;
}

//Devuelve el camino minimo entre origen y destino(calcula el arbol, pero reconstruye solo el
    camino de origen a destino)
Camino Grafo::dijkstra(nodo_t origen, nodo_t destino, tipo_costo_t target_a_minimizar){
    vector<costo_t> costo_minimo;
    vector<nodo_t> predecesor;
    this->dijkstra(origen, target_a_minimizar, costo_minimo, predecesor);

    //armar camino entre origen y destino
    Camino c(this->mat_adyacencia);
    nodo_t nodo = destino;
    do{
        //cout << nodo << " " ;
        c.agregar_nodo_adelante(nodo);
        nodo = predecesor[nodo];
    }while(nodo != predecesor_nulo);
    return c;
}

//Aplica dijkstra desde nodo origen y calcula el arbol de caminos minimos por referencia a los
    vectores por parametro
```

```cpp
void Grafo::dijkstra(nodo_t origen, tipo_costo_t target_a_minimizar, vector<costo_t>& costo_minimo, vector<nodo_t>& predecesor){
    //inicializacion
    int n = this->cantidad_nodos;
    costo_minimo.clear();
    costo_minimo.resize(n, costo_infinito);
    costo_minimo[origen] = costo_nulo;

    predecesor.clear();
    predecesor.resize(n, predecesor_nulo);

    set<pair<costo_t, nodo_t> > cola;
    cola.insert(make_pair(costo_minimo[origen], origen));

    while(!cola.empty()){
        costo_t cost_to_u = cola.begin()->first;
        nodo_t nodo_u = cola.begin()->second;
        //marcar como visto en este caso es eliminarlo de la cola
        cola.erase(cola.begin());

        //Para cada vecino de u, obtengo el vector fila de la matriz de adyacencia
        //de dicho nodo e itero sobre todos, quedandome con las aristas presentes
        lista_adyacentes::iterator adyacentes_i_it = this->lista_adyacencia[nodo_u].begin();
        lista_adyacentes::iterator final_it = this->lista_adyacencia[nodo_u].end();
        while(adyacentes_i_it != final_it){
            nodo_t nodo_v = adyacentes_i_it->first;

            costo_t cost_v = (target_a_minimizar == COSTO_W1) ? (adyacentes_i_it->second).obtener_costo_w1() : (adyacentes_i_it->second).obtener_costo_w2();
            int costo_a_v_pasando_por_u = cost_to_u + cost_v;
            //si mejora, sobreescribo el camino y su costo
            if(costo_a_v_pasando_por_u < costo_minimo[nodo_v]){
                //elimino de la cola de nodos la distancia vieja
                cola.erase(make_pair(costo_minimo[nodo_v], nodo_v));

                //actualizo estructuras
                costo_minimo[nodo_v] = costo_a_v_pasando_por_u;
                predecesor[nodo_v] = nodo_u;

                //sobreescribo la distancia en la cola de nodos la distancia nueva
                cola.insert(make_pair(costo_minimo[nodo_v], nodo_v));
            }
            adyacentes_i_it++;
        }
    }
}

void Grafo::breadth_first_search(nodo_t origen, vector<distancia_t>& distancias){
    distancias.clear();
    distancias.resize(this->cantidad_nodos, distancia_infinita);
    queue<nodo_t> cola;
    vector<bool> visitado(this->cantidad_nodos, false);//inicializo todos los nodos sin visitar
    cola.push(origen);
    visitado[origen] = true;
    distancias[origen] = 0;//dist(src, src) = 0

    while(!cola.empty()){
        nodo_t target = cola.front();//obtengo el primero
        cola.pop();//desencolo el primero

        //para todos los vecinos de target
        lista_adyacentes::iterator adyacentes_i_it = this->lista_adyacencia[target].begin();
        lista_adyacentes::iterator final_it = this->lista_adyacencia[target].end();
        while(adyacentes_i_it != final_it){
            nodo_t vecino_candidato = adyacentes_i_it->first;
            if(!visitado[vecino_candidato]){
                cola.push(vecino_candidato);
                visitado[vecino_candidato] = true;
                distancias[vecino_candidato] = distancias[target] + 1;
            }
            adyacentes_i_it++;
        }
    }
}

//para impl. con listas de adyacencia, es O(n**2) si no estan ordenados haciendo 2 fors
```

```
//se puede mejorar, "ordenando" las listas en espacios auxiliares en O(nlogn) y intersecando en O(n)
//dejando complejidad de O(nlogn), de cualquier manera, como se nos pidio que fuera polinomial unicamente
//uso la matriz de adyacencia, y en O(n) recorro los vecinos de ambos, y donde se cumpla vecindad en ambos
//lo selecciono como vecino en comun.
void Grafo::precalcular_adyacentes_en_comun(nodo_t i, nodo_t j){
    vector<Vecino> res;
    vector<Arista> adyacentesFila_i = this->mat_adyacencia[i];
    vector<Arista> adyacentesFila_j = this->mat_adyacencia[j];
    for(int idx=0;idx<this->cantidad_nodos;idx++){
        if(adyacentesFila_i[idx].esta_presente() && adyacentesFila_j[idx].esta_presente()){
            //el nodo idx es adyacente de i y j.
            res.push_back(Vecino(i, j, idx, adyacentesFila_i[idx], adyacentesFila_j[idx]));
        }
    }
    this->vecinos_comunes[i][j] = res;
}

vector<Vecino> Grafo::obtener_adyacentes_en_comun(nodo_t i, nodo_t j){
    //cout << "Vecinos comunes de (" << i << ") y (" << j << ") : ";
    #ifdef VECINOS_COMUNES_LAZY
        vector<Vecino> res;
        vector<Arista> adyacentesFila_i = this->mat_adyacencia[i];
        vector<Arista> adyacentesFila_j = this->mat_adyacencia[j];
        for(int idx=0;idx<this->cantidad_nodos;idx++){
            if(adyacentesFila_i[idx].esta_presente() && adyacentesFila_j[idx].esta_presente()){
                //el nodo idx es adyacente de i y j.
                res.push_back(Vecino(i, j, idx, adyacentesFila_i[idx], adyacentesFila_j[idx]));
                //cout << "(" << idx << ") --->";
            }
        }
        //cout << "Nil" << endl;
        return res;
    #else
        //list<Vecino> tmp = this->vecinos_comunes[i][j];
        //for(Vecino v : tmp){
            //cout << "(" << v.obtener_nodo_comun() << ") ---> ";
        //}
        //cout << "Nil" << endl;
        return this->vecinos_comunes[i][j];
    #endif
}

vector<Arista> Grafo::obtener_vector_fila_vecinos(nodo_t target){
    return this->mat_adyacencia[target];
}

lista_adyacentes Grafo::obtener_lista_vecinos(nodo_t target){
    return this->lista_adyacencia[target];
}

nodo_t Grafo::obtener_nodo_origen(){
    return this->nodo_src;
}

nodo_t Grafo::obtener_nodo_destino(){
    return this->nodo_dst;
}

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,
 Arista& mejor_vecino){
    //es un grafo no dirigido, es simetrico buscar si a i es adyacente a j o viceversa
    //me fijo si i y j son adyacentes
    Arista candidato_a_mejor_camino = this->obtener_arista(nodo_i, nodo_j);
    if(candidato_a_mejor_camino.esta_presente()){
        //FIX. ADEMAS DE EXISTIR LA ARISTA, EL CAMINO DIRECTO TIENE QUE SER MEJOR ESTRICTO QUE LA SUMA DE LAS DOS ARISTAS EXISTENTES

        costo_t hipotetico_w1_total_camino = (total_w1 - (costo_ij_w1) + candidato_a_mejor_camino.obtener_costo_w1());
        bool es_mejora_factible = (candidato_a_mejor_camino.obtener_costo_w2() < costo_ij_w2) /*mejora la arista directa*/&&
            (hipotetico_w1_total_camino <= this->obtener_limite_w1()); /*no se pasa de la cota */
```

```
        if(es_mejora_factible){
            //la arista directa mejora el camino en w2 pero no se pasa de w1 el camino

            //OJO en esta linea!: si asigno candidato_a_mejor_camino estoy devolviendo una referencia a una variable de stack y catapunchis!
            mejor_vecino = this->obtener_arista(nodo_i, nodo_j);
            return true;
        }else{
            //o no mejora o se pasa w1 => false
            return false;
        }
    }else{
        //no hay arista directa => false
        return false;
    }
}

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,
 Vecino& mejor_vecino){
    //SEA LA VECINDAD Vc = {Caminos C' tal que difieran de C en tan solo un nodo}

    //DEFINO UNA TABOO LIST COMO UNA LISTA EN LA CUAL VOY A DESCARTAR LAS SOLUCIONES QUE USEN NODOS DE DICHA LISTA
    //ESTO ES NECESARIO PARA EVITAR LA GENERACION DE CICLOS EN EL CASO DE AGREGAR UN NODO AL SUBDIVIDIR UNA ARISTA O REEMPLAZAR UN NODO
    //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.
    //SI MANTENEMOS DISJUNTOS EL CONJUNTO DE NODOS DEL CAMINO ACTUAL Y LOS NODOS RESTANTES DEL GRAFO, CUALQUIER ELECCION QUE HAGAMOS NO GENERARA CICLOS.

    //OTRA OPCION SERIA NO RESTRINGIR LA ELECCION DE LAS SOLUCIONES DE LA VECINDAD, PERO DEBERIAMOS LUEGO REALIZAR UNA PODA DE CICLOS DEL CAMINO
    //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
    //DEJA DE SER BUSQUEDA LOCAL, DADO QUE LA SOLUCION QUE SURJA DE ESTO PUEDE NO ESTAR EN LA VECINDAD Vc

    //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
    vector<Vecino> vecinosEnComun = this->obtener_adyacentes_en_comun(nodo_i, nodo_j);
    vector<Vecino>::iterator vecinos_it = vecinosEnComun.begin();
    vector<Vecino>::iterator final_vecinos = vecinosEnComun.end();

    //me fijo todos los caminos alternativos agregando un nodo entre los nodos ij,
    vector<Vecino>::iterator mejor_vecino_it = vecinosEnComun.end();//inicializamos en algo que indique que no hay mejora
    costo_t mejor_camino_ij_w2 = costo_ij_w2;
    while(vecinos_it != final_vecinos){
        costo_t i_comun_w1 = vecinos_it->obtener_arista_i_comun().obtener_costo_w1();
        costo_t i_comun_w2 = vecinos_it->obtener_arista_i_comun().obtener_costo_w2();
        costo_t j_comun_w1 = vecinos_it->obtener_arista_j_comun().obtener_costo_w1();
        costo_t j_comun_w2 = vecinos_it->obtener_arista_j_comun().obtener_costo_w2();
        costo_t costo_i_comun_j_w1 = i_comun_w1 + j_comun_w1;
        costo_t costo_i_comun_j_w2 = i_comun_w2 + j_comun_w2;
        costo_t hipotetico_w1_total_camino = (total_w1 - costo_ij_w1 + costo_i_comun_j_w1);
        costo_t hipotetico_w2_total_camino = (total_w2 - costo_ij_w2 + costo_i_comun_j_w2);

        //veamos si el camino es una solucion factible
        if(hipotetico_w1_total_camino <= this->obtener_limite_w1()){
            //es factible, veamos si mejora al ultimo mejor revisado

            //VEAMOS ADEMAS, QUE NO ESTE EN NUESTRA TABOO LIST(QUE NO PERTENEZCA AL CAMINO DE LA SOL. ACTUAL)
            nodo_t nodo_comun = vecinos_it->obtener_nodo_comun();
            if(!this->camino_obtenido.pertenece_a_camino(nodo_comun)){//puedo verlo en O(1)
                //el nodo no esta en la taboo list
                if(costo_i_comun_j_w2 < mejor_camino_ij_w2){
                    //encontre mejora, actualizo variables
                    mejor_camino_ij_w2 = hipotetico_w2_total_camino;
                    mejor_vecino_it = vecinos_it;
                }
            }else{
                //taboo list skipped!
                #ifdef DEBUG_MESSAGES_ON
                    #ifdef CYCLE_PREVENT_MESSAGE_ON
```

```
                    cout << "Salteamos el nodo (" << nodo_comun << ") entre (" << nodo_i << ") y (" << nodo_j << ") como candidato a mejorar la solucion actual porque al pertenecer al camino generaria un ciclo" << endl;
                    #endif
                #endif
            }
        }
        ++vecinos_it;
    }
    if(mejor_vecino_it != final_vecinos){
        mejor_vecino = *mejor_vecino_it;
        return true;
    }
    return false;
}

//Defino la vecindad del camino como Vc = {Todos los caminos c' que difieren en un nodo de c}
//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
//Primero reviso todos los pares de nodos del camino buscando posibles subdivisiones que mejoren la solucion.
//Me voy a quedar unicamente (si existen varias) con la mejor solucion de la vecindad, notemos que revisar toda la vecindad es cuadratico
//La longuitud de un camino simple puede acotarse por la cantidad de nodos n, luego hay n-1 pares de nodos en el camino
//Para cada par, es lineal la obtencion de los vecinos en comun, y al ser el camino una lista enlazada, la modificacion tiene costo
//constante O(1), en total esto nos da un costo cuadratico O(n**2)
int Grafo::busqueda_local_entre_pares_insertando(Camino& solucion_actual, Vecino& conexion_ij_minima_w2){
    //cout << "-------------------------------------Comienza iteracion de busqueda local insertando entre pares-------------------------------------" << endl;
    list<nodo_t>::const_iterator it = solucion_actual.obtener_iterador_const_begin();
    list<nodo_t>::const_iterator runner_it = solucion_actual.obtener_iterador_const_begin();
    runner_it++;
    list<nodo_t>::const_iterator final_camino = solucion_actual.obtener_iterador_const_end();
    costo_t total_w1 = solucion_actual.obtener_costo_total_w1_camino();
    costo_t total_w2 = solucion_actual.obtener_costo_total_w2_camino();

    costo_t total_inicial_w2 = total_w2;

    //variables para guardar la mejor solucion de la vecindad
    costo_t mejor_costo_w2 = costo_infinito;
    list<nodo_t>::const_iterator punto_de_insercion_mejora_it = final_camino;
    bool hay_mejoras_para_el_camino = false;

    //busco la mejor solucion en la vecindad
    #ifdef DEBUG_MESSAGES_ON
        cout << "Solucion actual: ";
        solucion_actual.imprimir_camino(cout);
        cout << "Costo total del camino actual:   W1: " << total_w1 << "    W2: " << total_w2 << endl;
    #endif
    while(runner_it != final_camino){
        nodo_t nodo_i = *it;
        nodo_t nodo_j = *runner_it;
        costo_t costo_ij_w1 = solucion_actual.obtener_costo_w1_entre_nodos(nodo_i, nodo_j);
        costo_t costo_ij_w2 = solucion_actual.obtener_costo_w2_entre_nodos(nodo_i, nodo_j);

        Vecino mejor_conexion_ij;
        //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.
        //cout << "Buscando mejorar la conexion (" << nodo_i << ")----[" << costo_ij_w1 << ", " << costo_ij_w2 << "]---->(" << nodo_j << ") agregando un nodo intermedio..." << endl;
        bool encontre_mejora = mejorar_conexion_entre_pares(nodo_i, nodo_j,
                                                            costo_ij_w1, costo_ij_w2,
                                                            total_w1, total_w2,
                                                            mejor_conexion_ij);

        //hay que ver si encontramos una mejora
        if(encontre_mejora){//la funcion asegura que si dio true, me da el vecino por puntero en mejor_conexion_ij
            //nodo_t nodo_comun = mejor_conexion_ij.obtener_nodo_comun();
            costo_t i_comun_w1 = mejor_conexion_ij.obtener_arista_i_comun().obtener_costo_w1();
            costo_t i_comun_w2 = mejor_conexion_ij.obtener_arista_i_comun().obtener_costo_w2();
            costo_t j_comun_w1 = mejor_conexion_ij.obtener_arista_j_comun().obtener_costo_w1();
```

```
            costo_t j_comun_w2 = mejor_conexion_ij.obtener_arista_j_comun().obtener_costo_w2();
            costo_t costo_i_comun_j_w1 = i_comun_w1 + j_comun_w1;
            costo_t costo_i_comun_j_w2 = i_comun_w2 + j_comun_w2;
            //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;
            //cout << "\tCamino (" << nodo_i << ")----[" << i_comun_w1 << ", " << i_comun_w2 << "]---->(" << nodo_comun << ")----[" << j_comun_w1 << ", " << j_comun_w2 << "]---->(" << nodo_j;
            //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;

            costo_t hipotetico_w1_total_camino = (total_w1 - costo_ij_w1 + costo_i_comun_j_w1);
            costo_t hipotetico_w2_total_camino = (total_w2 - costo_ij_w2 + costo_i_comun_j_w2);

            if( (hipotetico_w2_total_camino < mejor_costo_w2) && (hipotetico_w1_total_camino <= this->obtener_limite_w1())){
                mejor_costo_w2 = hipotetico_w2_total_camino;
                conexion_ij_minima_w2 = mejor_conexion_ij;
                punto_de_insercion_mejora_it = it;
                hay_mejoras_para_el_camino = true;
            }

            //cout << "\tSi aplicamos este cambio los costos del camino total quedarian:   W1: " << hipotetico_w1_total_camino << "    W2: "  << hipotetico_w2_total_camino << endl;
        }//else{
            //cout << "\tNo se encontro mejora." << endl;
        //}
        //cout << endl;
        ++it;
        ++runner_it;
    }

    //Si hubo mejoras, tengo guardada la mejor
    if(hay_mejoras_para_el_camino){
        #ifdef DEBUG_MESSAGES_ON
            cout << "Se encontro mejora. La mejor mejora encontrada entre todos los pares de nodos fue:" << endl;
        #endif
        //insertar conexion_ij_minima_w2 en punto_de_insercion_mejora_it
        //y actualizar todos los atributos necesarios.

        nodo_t nodo_i = conexion_ij_minima_w2.obtener_nodo_i();
        nodo_t nodo_j = conexion_ij_minima_w2.obtener_nodo_j();
        nodo_t nodo_comun = conexion_ij_minima_w2.obtener_nodo_comun();
        costo_t i_comun_w1 = conexion_ij_minima_w2.obtener_arista_i_comun().obtener_costo_w1();
        costo_t i_comun_w2 = conexion_ij_minima_w2.obtener_arista_i_comun().obtener_costo_w2();
        costo_t j_comun_w1 = conexion_ij_minima_w2.obtener_arista_j_comun().obtener_costo_w1();
        costo_t j_comun_w2 = conexion_ij_minima_w2.obtener_arista_j_comun().obtener_costo_w2();
        costo_t costo_i_comun_j_w1 = i_comun_w1 + j_comun_w1;
        costo_t costo_i_comun_j_w2 = i_comun_w2 + j_comun_w2;
        #ifdef DEBUG_MESSAGES_ON
            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;
            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;
        #endif
        return (total_inicial_w2 - mejor_costo_w2);//mejora respecto a w2 en el camino del inicial al actual
    }else{
        #ifdef DEBUG_MESSAGES_ON
            cout << "No se pudo mejorar la solucion." << endl;
        #endif
        return 0;//mejoro en 0 el camino respecto a w2
    }
}

int Grafo::busqueda_local_entre_triplas_reemplazando_intermedio(Camino& solucion_actual, Vecino& conexion_ij_minima_w2){
    //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
    //cout << "-----------------------------------Comienza iteracion de busqueda local reemplazando intermedio-----------------------------------" << endl;
```

```
    if(solucion_actual.obtener_longuitud_camino()<3){
        #ifdef DEBUG_MESSAGES_ON
            cerr << "Camino de menos de 3 nodos. No se puede mejorar nada." << endl;
        #endif
        return false;
    }

    //aca vale size(camino)>=3

    list<nodo_t>::const_iterator it = solucion_actual.obtener_iterador_const_begin();
    list<nodo_t>::const_iterator it_sig = solucion_actual.obtener_iterador_const_begin();
    list<nodo_t>::const_iterator runner_it = solucion_actual.obtener_iterador_const_begin();
    it_sig++;
    runner_it++;runner_it++;

    list<nodo_t>::const_iterator final_camino = solucion_actual.obtener_iterador_const_end();
    costo_t total_w1 = solucion_actual.obtener_costo_total_w1_camino();
    costo_t total_w2 = solucion_actual.obtener_costo_total_w2_camino();

    costo_t total_inicial_w2 = total_w2;

    //variables para guardar la mejor solucion de la vecindad
    costo_t mejor_costo_w2 = costo_infinito;
    list<nodo_t>::const_iterator punto_de_insercion_mejora_it = final_camino;
    bool hay_mejoras_para_el_camino = false;

    //busco la mejor solucion en la vecindad
    #ifdef DEBUG_MESSAGES_ON
        cout << "Solucion actual: ";
        solucion_actual.imprimir_camino(cout);
        cout << "Costo total del camino actual:   W1: " << total_w1 << "    W2: " << total_w2 << endl;
    #endif
    while(runner_it != final_camino){
        nodo_t nodo_i = *it;
        nodo_t nodo_medio = *it_sig;
        nodo_t nodo_j = *runner_it;

        costo_t costo_i_medio_w1 = solucion_actual.obtener_costo_w1_entre_nodos(nodo_i, nodo_medio);
        costo_t costo_i_medio_w2 = solucion_actual.obtener_costo_w2_entre_nodos(nodo_i, nodo_medio);
        costo_t costo_medio_j_w1 = solucion_actual.obtener_costo_w1_entre_nodos(nodo_medio, nodo_j);
        costo_t costo_medio_j_w2 = solucion_actual.obtener_costo_w2_entre_nodos(nodo_medio, nodo_j);


        //cout << "Intentando mejorar el nodo intermedio entre (" << nodo_i << ")----[" << costo_i_medio_w1 << ", " << costo_i_medio_w2;
        //cout << "]---->(" << nodo_medio << ")----[" << costo_medio_j_w1 << ", " << costo_medio_j_w2 << "]---->(" << nodo_j << ")" << endl;

        Vecino mejor_conexion_ij;
        //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.
        bool encontre_mejora = mejorar_conexion_entre_pares(nodo_i, nodo_j,
                                             (costo_i_medio_w1 + costo_medio_j_w1), (costo_i_medio_w2 + costo_medio_j_w2),
                                             total_w1, total_w2,
                                             mejor_conexion_ij);

        //hay que ver si encontramos una mejora
        if(encontre_mejora){//la funcion asegura que si dio true, me da el vecino por puntero en mejor_conexion_ij
            //nodo_t nodo_comun = mejor_conexion_ij.obtener_nodo_comun();
            costo_t i_comun_w1 = mejor_conexion_ij.obtener_arista_i_comun().obtener_costo_w1();
            costo_t i_comun_w2 = mejor_conexion_ij.obtener_arista_i_comun().obtener_costo_w2();
            costo_t j_comun_w1 = mejor_conexion_ij.obtener_arista_j_comun().obtener_costo_w1();
            costo_t j_comun_w2 = mejor_conexion_ij.obtener_arista_j_comun().obtener_costo_w2();
            costo_t costo_i_comun_j_w1 = i_comun_w1 + j_comun_w1;
            costo_t costo_i_comun_j_w2 = i_comun_w2 + j_comun_w2;
            //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;
            //cout << "\tCamino (" << nodo_i << ")----[" << i_comun_w1 << ", " << i_comun_w2 << "]---->(" << nodo_comun << ")----[" << j_comun_w1 << ", " << j_comun_w2 <<
```

```
                "]---->(" << nodo_j;
            //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;

            costo_t hipotetico_w1_total_camino = (total_w1 - (costo_i_medio_w1 + costo_medio_j_w1) + costo_i_comun_j_w1);
            costo_t hipotetico_w2_total_camino = (total_w2 - (costo_i_medio_w2 + costo_medio_j_w2) + costo_i_comun_j_w2);

            if( (hipotetico_w2_total_camino < mejor_costo_w2) && (hipotetico_w1_total_camino <= this->obtener_limite_w1())){
                mejor_costo_w2 = hipotetico_w2_total_camino;
                conexion_ij_minima_w2 = mejor_conexion_ij;
                punto_de_insercion_mejora_it = it;
                hay_mejoras_para_el_camino = true;
            }

            //cout << "\tSi aplicamos este cambio los costos del camino total quedarian:   W1: " << hipotetico_w1_total_camino << "    W2: "  << hipotetico_w2_total_camino << endl;
        }//else{
            //cout << "\tNo se encontro mejora." << endl;
        //}

        //cout << endl;
        ++it;
        ++it_sig;
        ++runner_it;
    }

    //Si hubo mejoras, tengo guardada la mejor
    if(hay_mejoras_para_el_camino){
        #ifdef DEBUG_MESSAGES_ON
            cout << "Se encontro mejora. La mejor mejora encontrada entre todos los pares de nodos fue:" << endl;
        #endif

        //reemplazar conexion_ij_minima_w2 en punto_de_insercion_mejora_it y punto_de_insercion_mejora_it + 2
        //y actualizar todos los atributos necesarios.

        nodo_t nodo_i = conexion_ij_minima_w2.obtener_nodo_i();
        nodo_t nodo_j = conexion_ij_minima_w2.obtener_nodo_j();
        nodo_t nodo_comun = conexion_ij_minima_w2.obtener_nodo_comun();
        costo_t i_comun_w1 = conexion_ij_minima_w2.obtener_arista_i_comun().obtener_costo_w1();
        costo_t i_comun_w2 = conexion_ij_minima_w2.obtener_arista_i_comun().obtener_costo_w2();
        costo_t j_comun_w1 = conexion_ij_minima_w2.obtener_arista_j_comun().obtener_costo_w1();
        costo_t j_comun_w2 = conexion_ij_minima_w2.obtener_arista_j_comun().obtener_costo_w2();
        costo_t costo_i_comun_j_w1 = i_comun_w1 + j_comun_w1;
        costo_t costo_i_comun_j_w2 = i_comun_w2 + j_comun_w2;
        #ifdef DEBUG_MESSAGES_ON
            cout << "Se reemplazo el nodo intermedio en: (" << nodo_i << ")----[" << i_comun_w1 << ", " << i_comun_w2 <<  "]---->(" << nodo_comun << ")----[" << j_comun_w1 << ", " << j_comun_w2 << "]---->(" << nodo_j << ")" << endl;
            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;
        #endif
        return (total_inicial_w2 - mejor_costo_w2);
    }else{
        #ifdef DEBUG_MESSAGES_ON
            cout << "No se pudo mejorar la solucion." << endl;
        #endif
        return 0;
    }
}

int Grafo::busqueda_local_entre_triplas_salteando(Camino& solucion_actual, list<nodo_t>::const_iterator& punto_de_salto_it){
    //Caso en los que salteo un nodo vk---->vk+1---->vk+2 convirtiendolo en vk---->vk+2 tal que mejora w2 y w1 no se pasa en el costo total del camino
    //cout << "----------------------------------Comienza iteracion de busqueda local salteando----------------------------------" << endl;
    if(solucion_actual.obtener_longuitud_camino()<3){
        #ifdef DEBUG_MESSAGES_ON
            cerr << "Camino de menos de 3 nodos. No se puede mejorar nada." << endl;
```

```
        #endif
        return false;
    }

    //aca vale size(camino)>=3

    list<nodo_t>::const_iterator it = solucion_actual.obtener_iterador_const_begin();
    list<nodo_t>::const_iterator it_sig = solucion_actual.obtener_iterador_const_begin();
    list<nodo_t>::const_iterator runner_it = solucion_actual.obtener_iterador_const_begin();
    it_sig++;
    runner_it++;runner_it++;

    list<nodo_t>::const_iterator final_camino = solucion_actual.obtener_iterador_const_end();
    costo_t total_w1 = solucion_actual.obtener_costo_total_w1_camino();
    costo_t total_w2 = solucion_actual.obtener_costo_total_w2_camino();

    costo_t total_inicial_w2 = total_w2;

    //variables para guardar la mejor solucion de la vecindad
    costo_t mejor_costo_w2 = costo_infinito;
    Arista conexion_ij_minima_w2;
    punto_de_salto_it = final_camino;
    bool hay_mejoras_para_el_camino = false;

    //busco la mejor solucion en la vecindad
    #ifdef DEBUG_MESSAGES_ON
        cout << "Solucion actual: ";
        solucion_actual.imprimir_camino(cout);
        cout << "Costo total del camino actual:   W1: " << total_w1 << "    W2: " << total_w2 << endl;
    #endif
    while(runner_it != final_camino){
        nodo_t nodo_i = *it;
        nodo_t nodo_medio = *it_sig;
        nodo_t nodo_j = *runner_it;

        costo_t costo_i_medio_w1 = solucion_actual.obtener_costo_w1_entre_nodos(nodo_i, nodo_medio);
        costo_t costo_i_medio_w2 = solucion_actual.obtener_costo_w2_entre_nodos(nodo_i, nodo_medio);
        costo_t costo_medio_j_w1 = solucion_actual.obtener_costo_w1_entre_nodos(nodo_medio, nodo_j);
        costo_t costo_medio_j_w2 = solucion_actual.obtener_costo_w2_entre_nodos(nodo_medio, nodo_j);


        //cout << "Intentando remover el nodo intermedio entre (" << nodo_i << ")----[" << costo_i_medio_w1 << ", " << costo_i_medio_w2;
        //cout << "]---->(" << nodo_medio << ")----[" << costo_medio_j_w1 << ", " << costo_medio_j_w2 << "]---->(" << nodo_j << ")" << endl;

        Arista mejor_conexion_ij;
        //le paso una ref a una var tipo arista, si devuelve true, se escribe por referencia el mejor camino, sino, no cambia lo que le pasamos.
        bool encontre_mejora = mejorar_conexion_salteando(nodo_i, nodo_j,
                                                  (costo_i_medio_w1 + costo_medio_j_w1), (costo_i_medio_w2 + costo_medio_j_w2),
                                                  total_w1, total_w2,
                                                  mejor_conexion_ij);

        //hay que ver si encontramos una mejora
        if(encontre_mejora){//la funcion asegura que si dio true, me da el vecino por puntero en mejor_conexion_ij
            costo_t costo_ij_directo_w1 = mejor_conexion_ij.obtener_costo_w1();
            costo_t costo_ij_directo_w2 = mejor_conexion_ij.obtener_costo_w2();
            //cout << "\tSe encontro una posible mejora. Se encontro una arista directa entre (" << nodo_i << ") y (" << nodo_j << ") y es " << endl;
            //cout << "\tCamino (" << nodo_i << ")----[" << costo_ij_directo_w1 << ", " << costo_ij_directo_w2 << "]---->(" << nodo_j << ")";
            //cout <<   " Nuevo costo del sendero entre (" << nodo_i << ") y (" << nodo_j << ") aplicando a esta modificacion:    W1: " << costo_ij_directo_w1 << "     W2: " << costo_ij_directo_w2 << endl;

            costo_t hipotetico_w1_total_camino = (total_w1 - (costo_i_medio_w1 + costo_medio_j_w1) + costo_ij_directo_w1);
            costo_t hipotetico_w2_total_camino = (total_w2 - (costo_i_medio_w2 + costo_medio_j_w2) + costo_ij_directo_w2);
```

```
            if( (hipotetico_w2_total_camino < mejor_costo_w2) && (hipotetico_w1_total_camino <= this->obtener_limite_w1())){
                mejor_costo_w2 = hipotetico_w2_total_camino;
                hay_mejoras_para_el_camino = true;
                conexion_ij_minima_w2 = mejor_conexion_ij;
                punto_de_salto_it = it;
            }

            //cout << "\tSi aplicamos este cambio los costos del camino total quedarian:   W1: " << hipotetico_w1_total_camino << "    W2: "  << hipotetico_w2_total_camino << endl;
        }

        //cout << endl;
        ++it;
        ++it_sig;
        ++runner_it;
    }

    //Si hubo mejoras, tengo guardada la mejor
    if(hay_mejoras_para_el_camino){
        #ifdef DEBUG_MESSAGES_ON
            cout << "Se encontro mejora. La mejor mejora encontrada entre todos los pares de nodos fue:" << endl;
        #endif
        //Realizar el salto directo entre 2 nodos dada la arista obtenida
        list<nodo_t>::const_iterator nodo_j_it = punto_de_salto_it;
        nodo_j_it++;nodo_j_it++;

        nodo_t nodo_i = *punto_de_salto_it;
        nodo_t nodo_j = *nodo_j_it;

        costo_t costo_ij_directo_w1 = conexion_ij_minima_w2.obtener_costo_w1();
        costo_t costo_ij_directo_w2 = conexion_ij_minima_w2.obtener_costo_w2();
        #ifdef DEBUG_MESSAGES_ON
            cout << "\tSe encontro una arista directa entre (" << nodo_i << ") y (" << nodo_j << ") y es " << endl;
            cout << "\tCamino (" << nodo_i << ")----[" << costo_ij_directo_w1 << ", " << costo_ij_directo_w2 <<  "]---->(" << nodo_j << ")";
            cout << " Nuevo costo del sendero entre (" << nodo_i << ") y (" << nodo_j << ") aplicando a esta modificacion:    W1: " << costo_ij_directo_w1 << "     W2: " << costo_ij_directo_w2 << endl;
        #endif
        return (total_inicial_w2 - mejor_costo_w2);
    }else{
        #ifdef DEBUG_MESSAGES_ON
            cout << "No se pudo mejorar la solucion." << endl;
        #endif
        return 0;
    }
}

int Grafo::busqueda_local(tipo_ejecucion_bqlocal_t tipo_ejecucion){
    //decido que busquedas se van a correr
    bool bql_subdiv = false;
    bool bql_mejorar = false;
    bool bql_contraer = false;
    switch(tipo_ejecucion){
        case BQL_SUBDIVIDIR_PARES:
            bql_subdiv = true;
            break;
        case BQL_CONTRAER_TRIPLAS_A_PARES:
            bql_contraer = true;
            break;
        case BQL_MEJORAR_CONEXION_TRIPLAS:
            bql_mejorar = true;
            break;
        case BQL_COMBINAR:
            bql_subdiv = true;
            bql_contraer = true;
            bql_mejorar = true;
            break;
        default:
            cerr << "busqueda_local: tipo_ejecucion invalido!" << endl;
            return 0;
            break;
```

```
    }

    //establezco solucion actual y corro los algoritmos seleccionados previamente
    Camino solucion_actual = this->camino_obtenido;
    costo_t mejora_en_w2_bql_entre_pares = 0;
    Vecino conexion_ij_minima_w2_entre_pares;

    costo_t mejora_en_w2_entre_triplas_reemplazando = 0;
    Vecino conexion_ij_minima_w2_entre_triplas;

    costo_t mejora_en_w2_entre_triplas_salteando = 0;
    list<nodo_t>::const_iterator punto_de_salto_it = solucion_actual.obtener_iterador_const_end();

    if(bql_subdiv){
        #ifdef DEBUG_MESSAGES_ON
            cout << endl << "\t---- Ejecutando busqueda local entre pares insertando ----" << endl << endl;
        #endif
        mejora_en_w2_bql_entre_pares = busqueda_local_entre_pares_insertando(solucion_actual, conexion_ij_minima_w2_entre_pares);
        #ifdef DEBUG_MESSAGES_ON
            if(mejora_en_w2_bql_entre_pares > 0){
                cout << "\t---- Mejora tentativa sobre w2 en el camino actual aplicando esta mejora: " << mejora_en_w2_bql_entre_pares << endl;
            }
        #endif
    }

    if(bql_mejorar){
        #ifdef DEBUG_MESSAGES_ON
            cout << endl << "\t---- Ejecutando busqueda local entre triplas reemplazando intermedio ----" << endl << endl;
        #endif
        mejora_en_w2_entre_triplas_reemplazando = busqueda_local_entre_triplas_reemplazando_intermedio(solucion_actual, conexion_ij_minima_w2_entre_triplas);
        #ifdef DEBUG_MESSAGES_ON
            if(mejora_en_w2_entre_triplas_reemplazando > 0){
                cout << "\t---- Mejora tentativa sobre w2 en el camino actual aplicando esta mejora: " << mejora_en_w2_entre_triplas_reemplazando << endl;
            }
        #endif
    }

    if(bql_contraer){
        #ifdef DEBUG_MESSAGES_ON
            cout << endl << "\t---- Ejecutando busqueda local entre triplas salteando ----" << endl << endl;
        #endif
        mejora_en_w2_entre_triplas_salteando = busqueda_local_entre_triplas_salteando(solucion_actual, punto_de_salto_it);
        #ifdef DEBUG_MESSAGES_ON
            if(mejora_en_w2_entre_triplas_salteando > 0){
                cout << "\t---- Mejora tentativa sobre w2 en el camino actual aplicando esta mejora: " << mejora_en_w2_entre_triplas_salteando << endl;
            }
        #endif
    }

    //si son todas las mejoras 0, no mejoro ninguna nada
    if( (mejora_en_w2_bql_entre_pares == 0) && (mejora_en_w2_entre_triplas_reemplazando == 0) && (mejora_en_w2_entre_triplas_salteando == 0) ){
        return 0;
    }else{
        //existe al menos uno de los 3 que mejoro, es decir que es > 0
        if(mejora_en_w2_bql_entre_pares > mejora_en_w2_entre_triplas_reemplazando){
            //vale mejora_en_w2_bql_entre_pares > mejora_en_w2_entre_triplas_reemplazando
            if(mejora_en_w2_bql_entre_pares > mejora_en_w2_entre_triplas_salteando){
                //max = mejora_en_w2_bql_entre_pares;
                if(mejora_en_w2_bql_entre_pares > 0){
                    if(solucion_actual.insertar_nodo(conexion_ij_minima_w2_entre_pares)){
                        #ifdef DEBUG_MESSAGES_ON
                            cout << endl << "Nueva solucion obtenida: ";
                            solucion_actual.imprimir_camino(cout);
                            cout << "Nuevos costos totales del camino:   W1: " << solucion_actual.obtener_costo_total_w1_camino() << "    W2: "
```

```
                                << solucion_actual.obtener_costo_total_w2_camino() << endl;
                        #endif
                        this->establecer_camino_solucion(solucion_actual);
                        return mejora_en_w2_bql_entre_pares;
                    }else{
                        return 0;
                    }
                }else{
                    return 0;
                }
            }else{
                //max = mejora_en_w2_entre_triplas_salteando;
                if(mejora_en_w2_entre_triplas_salteando > 0){
                    nodo_t nodo_i = *punto_de_salto_it;
                    if(solucion_actual.realizar_salto_entre_3_nodos(nodo_i)){
                        //No puede haber ciclos, porque el camino quedo igual o con menos nodos
                        #ifdef DEBUG_MESSAGES_ON
                            cout << endl << "Nueva solucion obtenida: ";
                            solucion_actual.imprimir_camino(cout);
                            cout << "Nuevos costos totales del camino:   W1: " <<
                                solucion_actual.obtener_costo_total_w1_camino() << "    W2: "
                                << solucion_actual.obtener_costo_total_w2_camino() << endl;
                        #endif
                        this->establecer_camino_solucion(solucion_actual);
                        return mejora_en_w2_entre_triplas_salteando;
                    }else{
                        return 0;
                    }
                }else{
                    return 0;
                }
            }
        }else{
            //vale mejora_en_w2_bql_entre_pares <= mejora_en_w2_entre_triplas_reemplazando
            if(mejora_en_w2_entre_triplas_reemplazando > mejora_en_w2_entre_triplas_salteando){
                //max = mejora_en_w2_entre_triplas_reemplazando;
                if(mejora_en_w2_entre_triplas_reemplazando > 0){
                    if(solucion_actual.mejorar_tripla(conexion_ij_minima_w2_entre_triplas)){
                        #ifdef DEBUG_MESSAGES_ON
                            cout << endl << "Nueva solucion obtenida: ";
                            solucion_actual.imprimir_camino(cout);
                            cout << "Nuevos costos totales del camino:   W1: " <<
                                solucion_actual.obtener_costo_total_w1_camino() << "    W2: "
                                << solucion_actual.obtener_costo_total_w2_camino() << endl;
                        #endif
                        this->establecer_camino_solucion(solucion_actual);
                        return mejora_en_w2_entre_triplas_reemplazando;
                    }else{
                        return 0;
                    }
                }else{
                    return 0;
                }
            }else{
                //max = mejora_en_w2_entre_triplas_salteando;
                if(mejora_en_w2_entre_triplas_salteando > 0){
                    nodo_t nodo_i = *punto_de_salto_it;
                    if(solucion_actual.realizar_salto_entre_3_nodos(nodo_i)){
                        //No puede haber ciclos, porque el camino quedo igual o con menos nodos
                        #ifdef DEBUG_MESSAGES_ON
                            cout << endl << "Nueva solucion obtenida: ";
                            solucion_actual.imprimir_camino(cout);
                            cout << "Nuevos costos totales del camino:   W1: " <<
                                solucion_actual.obtener_costo_total_w1_camino() << "    W2: "
                                << solucion_actual.obtener_costo_total_w2_camino() << endl;
                        #endif
                        this->establecer_camino_solucion(solucion_actual);
                        return mejora_en_w2_entre_triplas_salteando;
                    }else{
                        return 0;
                    }
                }else{
                    return 0;
                }
            }
        }
    }
```

```
}

list<Grafo> Grafo::parsear_varias_instancias(formato_entrada_t formato){
    list<Grafo> instancias;
    //parseo todas las instancias
    bool instancia_valida = true;
    do{
        Grafo i(0);
        instancia_valida = i.unserialize(cin, formato);
        if(instancia_valida)
            instancias.push_back(i);
    }while(instancia_valida);
    //cout << "[Parse input]Se leyeron " << instancias.size() << " instancias de stdin" << endl << endl;
    return instancias;
}

void Grafo::establecer_se_encontro_solucion(bool se_encontro){
    this->sol_valida = se_encontro;
}

bool Grafo::hay_solucion(){
    return this->sol_valida;
}

Camino Grafo::obtener_solucion_golosa(){
    int n = this->cantidad_nodos;
    int k = obtener_limite_w1();
    nodo_t origen = obtener_nodo_origen();
    nodo_t destino = obtener_nodo_destino();

    //camino a devolver
    Camino c(this->mat_adyacencia);

    //dijkstra de inicializacion
    vector<costo_t> costosw1;
    vector<nodo_t> predecesores;
    this->dijkstra(destino, COSTO_W1, costosw1, predecesores);

    //ACA YA PUEDO SABER SI NO VA A HABER SOLUCION FACTIBLE CON EL DIJKSTRA PREPROCESADO
    if(costosw1[origen] == costo_infinito){//dist_w1(origen, destino) == infinito?
        cerr << "No existe solucion factible. No existe camino entre origen(" << origen << ") y destino(" << destino << ") " << endl;
        //ESTO HACE SI LE PONES FALSE, QUE GRAFO IMPRIMA "no" EN EL METODO SERIALIZE DE LA SALIDA.
        //ADEMAS DE SER UTIL PARA PREGUNTAR DESDE EL MAIN SI HUBO SOL.
        this->establecer_se_encontro_solucion(false);
    }else if(costosw1[origen] > k){//dist_w1(origen, destino) > limit_w1 = k?
        cerr << "No existe solucion factible. El camino minimo respecto a w1 de origen(" << origen << ") a destino(" << destino << ") es de costo " << costosw1[origen] << endl;
        //ESTO HACE SI LE PONES FALSE, QUE GRAFO IMPRIMA "no" EN EL METODO SERIALIZE DE LA SALIDA.
        //ADEMAS DE SER UTIL PARA PREGUNTAR DESDE EL MAIN SI HUBO SOL.
        this->establecer_se_encontro_solucion(false);
    }else{
        //---------------------------------------- Comienza Greedy ------------------------------------------------------

        //Estructuras del greedy
        vector<costo_t> costosw2(n, costo_infinito);
        // contiene el costo w1 del camino recorrido hasta cada nodo
        vector<costo_t> costoCamino(n, costo_nulo);

        //Comienza el algoritmo
        //reseteamos los predecesores
        predecesores.clear();
        predecesores.resize(n, predecesor_nulo);

        costosw2[origen] = costo_nulo;

        set<pair<costo_t, nodo_t> > cola;
        cola.insert(make_pair(costosw2[origen], origen));

        while(!cola.empty()){
            pair<costo_t, nodo_t> actual = *cola.begin();
            cola.erase(cola.begin());
```

```
            lista_adyacentes vecinos = this->obtener_lista_vecinos(actual.second);

            for(auto w : vecinos)
            {
                nodo_t nodoW = w.first;
                Arista aristaActualW = w.second;

                if (costoCamino[actual.second] + costosw1[nodoW] + aristaActualW.obtener_costo_w1() <= k){
                    costo_t costo_tentativo = costosw2[actual.second] + aristaActualW.obtener_costo_w2();
                    if (costosw2[nodoW] > costo_tentativo)
                    {
                        cola.erase(make_pair(costosw2[nodoW], nodoW));

                        costosw2[nodoW] = costo_tentativo;
                        costoCamino[nodoW] = costoCamino[actual.second] + aristaActualW.obtener_costo_w1();
                        predecesores[nodoW] = actual.second;

                        cola.insert(make_pair(costosw2[nodoW], nodoW));
                    }
                }
            }
        }

        //
        //{
            //
            //armar camino encontrado por la greedy
            nodo_t nodo = destino;
            //cout <<"Nodos" << endl;
            do{
                //cout << nodo << " " ;
                c.agregar_nodo_adelante(nodo);
                nodo = predecesores[nodo];
            }while(nodo != predecesor_nulo);

            //cout << endl << "Fin Nodos" << endl;
            this->establecer_se_encontro_solucion(true);
        //}
        //---------------------------------------- Fin Greedy ----------------------------------------
    }
    return c;
}

//parametro beta en RCL_POR_VALOR indica el porcentaje a alejarse del mejor candidato
//parametro beta en RCL_POR_CANTIDAD indica la candidad de "mejores" candidatos a elegir
set<pair<costo_t, nodo_t> >::iterator Grafo::obtener_candidato_randomizado(tipo_ejecucion_golosa_t tipo_ejecucion, const set<pair<costo_t, nodo_t> > & cola, double parametro_beta){
    set<pair<costo_t, nodo_t> >::iterator retorno;
    if(tipo_ejecucion == RCL_DETERMINISTICO){
        retorno = cola.begin();
    }else if(tipo_ejecucion == RCL_POR_CANTIDAD){
        //tengo que elegir un numero i, aleatorio entre 0 y min{cola.size(), parametro_beta} -1 elemento de la cola
        //y devolver el i-esimo elemento en orden de la cola
        //dado que la cola esta ordenada, cumple una RCL_POR_CANTIDAD
        uint rcl_target_top = (uint) std::min((uint)cola.size(), (uint) parametro_beta) - 1;

        //generacion numero random c++11 con distribucion uniforme
        //random_device rd;
        //mt19937 gen(rd());
        //uniform_int_distribution<> dis(0, rcl_target_top);
        //uint rcl_target_random = dis(gen); // generates number in the range 0..rcl_target_top

        //lo cambio por esto de C-legacy porque lo de c++11 me aumenta violentamente el tiempo de ejecucion
        srand(time(NULL));
        uint rcl_target_random = 0;
        if(rcl_target_top != 0){
            rcl_target_random = (unsigned int) rand() %rcl_target_top;
        }

```

```
        //cout << "cola size: " << cola.size() << endl;
        //cout << "parametro_beta: " << parametro_beta << endl;
        //cout << "rango random [0.." << rcl_target_top << "]" << endl;
        //cout << "tomando random: " << rcl_target_random << endl;
        //for(int i=0; i<22;i++){
        //  cout << dis(gen) << endl;
        //}

        //sabemos que  esta en rango del iterador pues rcl_target_top <= cola.size() -1 y tomamos begin() que es el primero, es decir 0
        retorno = cola.begin();
        uint avance = 0;
        assert(avance >= 0);
        assert(avance < cola.size());
        while(avance < rcl_target_random){
            retorno ++;
            avance++;
        }
        if(avance == 0){
            assert(retorno == cola.begin());
        }
    }else if (tipo_ejecucion == RCL_POR_VALOR){
        //itero sobre toda la cola, filtrando los elementos que esten dentro del porcentaje del parametro
        //luego selecciono uno al azar del vector de candidatos filtrados
        pair<costo_t, nodo_t> minimo = *cola.begin();
        double valor_limite = (parametro_beta + 1) * minimo.first;
        vector<set<pair<costo_t, nodo_t> >::iterator > candidatos;

        set<pair<costo_t, nodo_t> >::iterator it_cola = cola.begin();
        set<pair<costo_t, nodo_t> >::iterator it_cola_fin = cola.end();

        while(it_cola != it_cola_fin){
            if(it_cola->first <= valor_limite){
                candidatos.push_back(it_cola);
            }
            it_cola++;
        }

        //random_device rd;
        //mt19937 gen(rd());
        //uniform_int_distribution<> dis(0, candidatos.size() -1);
        //uint rcl_target_random = dis(gen); // generates number in the range 0..candidatos.size() -1

        //lo cambio por esto de C-legacy porque lo de c++11 me aumenta violentamente el tiempo de ejecucion
        uint rcl_target_top = candidatos.size() -1;
        srand(time(NULL));
        uint rcl_target_random = 0;
        if(rcl_target_top > 0){
            rcl_target_random = (unsigned int) rand() % rcl_target_top;
        }

        retorno = candidatos[rcl_target_random];
    }else{
        cerr << "[Error] Parametro no soportado de randomizacion de RCL. Asumiendo RCL_DETERMINISTICO " << tipo_ejecucion << endl;
        retorno = cola.begin();
    }
    return retorno;
}

costo_t Grafo::obtener_costo_actual_w1_solucion(){
    return this->camino_obtenido.obtener_costo_total_w1_camino();
}

costo_t Grafo::obtener_costo_actual_w2_solucion(){
    return this->camino_obtenido.obtener_costo_total_w2_camino();
}

//typedef enum tipo_ejecucion_golosa_t {RCL_DETERMINISTICO, RCL_POR_VALOR, RCL_POR_CANTIDAD} tipo_ejecucion_golosa_t;
Camino Grafo::obtener_solucion_golosa_randomizada(tipo_ejecucion_golosa_t tipo_ejecucion, double parametro_beta){
    int n = this->cantidad_nodos;
    int k = obtener_limite_w1();
```

```
    nodo_t origen = obtener_nodo_origen();
    nodo_t destino = obtener_nodo_destino();

    //camino a devolver
    Camino c(this->mat_adyacencia);

    vector<bool> visitados(n, false);

    //dijkstra de inicializacion
    vector<costo_t> costosw1;
    vector<nodo_t> predecesores;
    this->dijkstra(destino, COSTO_W1, costosw1, predecesores);

    //ACA YA PUEDO SABER SI NO VA A HABER SOLUCION FACTIBLE CON EL DIJKSTRA PREPROCESADO
    if(costosw1[origen] == costo_infinito){//dist_w1(origen, destino) == infinito?
        cerr << "No existe solucion factible. No existe camino entre origen(" << origen << ") y destino(" << destino << ") " << endl;
        //ESTO HACE SI LE PONES FALSE, QUE GRAFO IMPRIMA "no" EN EL METODO SERIALIZE DE LA SALIDA.
        //ADEMAS DE SER UTIL PARA PREGUNTAR DESDE EL MAIN SI HUBO SOL.
        this->establecer_se_encontro_solucion(false);
    }else if(costosw1[origen] > k){//dist_w1(origen, destino) > limit_w1 = k?
        cerr << "No existe solucion factible. El camino minimo respecto a w1 de origen(" << origen << ") a destino(" << destino << ") es de costo " << costosw1[origen] << endl;
        //ESTO HACE SI LE PONES FALSE, QUE GRAFO IMPRIMA "no" EN EL METODO SERIALIZE DE LA SALIDA.
        //ADEMAS DE SER UTIL PARA PREGUNTAR DESDE EL MAIN SI HUBO SOL.
        this->establecer_se_encontro_solucion(false);
    }else{
        //-------------------------------------------------------- Comienza Greedy randomized --------------------------------------------------------
        //Estructuras del greedy
        vector<costo_t> costosw2(n, costo_infinito);
        // contiene el costo w1 del camino recorrido hasta cada nodo
        vector<costo_t> costoCamino(n, costo_nulo);

        //Comienza el algoritmo
        //reseteamos los predecesores
        predecesores.clear();
        predecesores.resize(n, predecesor_nulo);

        costosw2[origen] = costo_nulo;

        visitados[origen] = true;

        set<pair<costo_t, nodo_t> > cola;
        cola.insert(make_pair(costosw2[origen], origen));

        while(!cola.empty()){
            //Seleccion de decision greedy segun tipo de ejecucion requerido por parametro
            set<pair<costo_t, nodo_t> >::iterator it_candidato = obtener_candidato_randomizado(tipo_ejecucion, cola, parametro_beta);

            pair<costo_t, nodo_t> actual = *it_candidato;
            cola.erase(it_candidato);
            visitados[actual.second] = true;

            lista_adyacentes vecinos = this->obtener_lista_vecinos(actual.second);

            for(auto w : vecinos)
            {
                nodo_t nodoW = w.first;
                Arista aristaActualW = w.second;

                if ((costoCamino[actual.second] + costosw1[nodoW] + aristaActualW.obtener_costo_w1() <= k) && !(visitados[nodoW])){
                    costo_t costo_tentativo = costosw2[actual.second] + aristaActualW.obtener_costo_w2();
                    if (costosw2[nodoW] > costo_tentativo)
                    {
                        cola.erase(make_pair(costosw2[nodoW], nodoW));

                        costosw2[nodoW] = costo_tentativo;
                        costoCamino[nodoW] = costoCamino[actual.second] + aristaActualW.obtener_costo_w1();
                        predecesores[nodoW] = actual.second;
```

```
                        cola.insert(make_pair(costosw2[nodoW], nodoW));
                    }
                }
            }
        }

        //if(se_encontro_sol?) suponiendo que si hay sol factible el algoritmo encuentra una
            solucion, esto no hace falta
        //{
            //
            //armar camino encontrado por la greedy
            nodo_t nodo = destino;
            //cout <<"Nodos" << endl;
            do{
                //cout << nodo << " " ;
                c.agregar_nodo_adelante(nodo);
                nodo = predecesores[nodo];
            }while(nodo != predecesor_nulo);

            //assert replaced
            //assert(c.obtener_costo_total_w1_camino() == costoCamino[destino]);
            if(c.obtener_costo_total_w1_camino() != costoCamino[destino]){
                //c.imprimir_camino(cout);
                //cout << "Camino invalido" << endl;
                this->establecer_se_encontro_solucion(false);//no es sol factible
            }

            //cout << endl << "Fin Nodos" << endl;
            this->establecer_se_encontro_solucion(true);
        //}
        //----------------------------------------- Fin Greedy randomized
            ------------------------------------------------------------
    }
    return c;
}

//
    ---------------------------------------------------------------------------------------------------------------------


#include "grafo.h"
#include "timing.h"
#include <fstream>
#include <cmath>

#define FILE_ITERS_MEJORA "evolucion_iteraciones_grasp.txt"
#define FILE_ITERS_COSTOS_ABSOLUTOS "costos_absolutos_iteraciones_grasp.txt"
#define FILE_ITERS_COSTOS_ABSOLUTOS_STATISTICS "costos_absolutos_iteraciones_grasp_analisis.txt
    "

void ejecutar_grasp(Grafo &g);

// --------------- Main -----------------------------------
int main(int argc, char **argv){
    list<Grafo> instancias = Grafo::parsear_varias_instancias(FORMATO_1_N_CLOSED);
    uint64_t instance_number = 1;
    for(Grafo &g : instancias){
        #ifdef DEBUG_MESSAGES_ON
            cout << endl << endl << "Aplicando metaheuristica GRASP a la " << instance_number
                << "-esima instancia de input..." << endl;
        #endif
        ejecutar_grasp(g);
        instance_number++;
    }
    return 0;
}

void ejecutar_grasp(Grafo &g){
    #ifdef DEBUG_MESSAGES_ON
        //g.imprimir_lista_adyacencia(cout);
        //g.imprimir_matriz_adyacencia(cout);
    #endif

//----- Configuracion del criterio de terminacion de GRASP -----

```

```
    //typedef enum criterio_terminacion_grasp_t {CRT_K_ITERS_SIN_MEJORA, CRT_K_ITERS_LIMIT_REACHED} criterio_terminacion_grasp_t;
    criterio_terminacion_grasp_t criterio_terminacion = CRT_K_ITERS_SIN_MEJORA;
    //este parametro denota la cantidad de iteraciones maxima, dependiendo del tipo de criterio, de cantidad fija de iteraciones o cantidad de iters
    //consecutivas sin mejora
    uint64_t ITERS_LIMIT = 5;
    //consecutivas sin que greedy rand me de una solucion factible
    uint64_t RAND_GREEDY_BAD_ITERS_LIMIT = 3;
    //este parametro denota el valor aceptable de la funcion objetivo w2 a partir del cual, dejamos de mejorar la solucion y consideramos que es lo suficientemente buena

//------ Configuracion de los modos de la busqueda local y golosa ------

    //typedef enum tipo_ejecucion_bqlocal_t {BQL_SUBDIVIDIR_PARES, BQL_CONTRAER_TRIPLAS_A_PARES, BQL_MEJORAR_CONEXION_TRIPLAS, BQL_COMBINAR} tipo_ejecucion_bqlocal_t;
    tipo_ejecucion_bqlocal_t modo_busqueda_local = BQL_COMBINAR;
    //typedef enum tipo_ejecucion_golosa_t {RCL_DETERMINISTICO, RCL_POR_VALOR, RCL_POR_CANTIDAD} tipo_ejecucion_golosa_t;
    tipo_ejecucion_golosa_t modo_golosa = RCL_POR_CANTIDAD;
    //si el tipo de golosa es RCL_POR_VALOR, este parametro indica el porcentaje de alejamiento del minimo de los candidatos de la lista
    //mas formalmente filtra todos los candidatos factibles locales que no cumplan candidato->costo_w2 <= valor_limite
    //donde valor limite es  (parametro_beta + 1) * minimo.second.obtener_costo_w2();
    //si el tipo de golosa es RCL_POR_CANTIDAD, este parametro indica la cantidad min{cant_candidatos, parametro_beta} de soluciones a considerar en la lista
    //si el tipo es RCL_DETERMINISTICO, este parametro es ignorado por el metodo.
    double parametro_beta = ceil(g.obtener_cantidad_nodos()/(float)1);
    if(modo_golosa == RCL_POR_CANTIDAD){
        assert(parametro_beta >= 1);
    }else if(modo_golosa == RCL_POR_VALOR){
        assert(parametro_beta > 0);
    }

    //----------------------------------------------------------------------------

    bool condicion_terminacion = false;
    Camino mejor_solucion = g.obtener_camino_vacio();
    costo_t costo_mejor_solucion = costo_infinito;
    uint64_t cant_iters = 0;
    uint64_t cant_iters_sin_sol_greedy_rand_factible = 0;
    uint64_t cant_iters_sin_mejora = 0;
    double tiempo_golosa_randomized = 0;
    double promedio = 0;
    Camino camino = g.obtener_camino_vacio();
    bool sol_valida_greedy = false;
    vector<pair<uint, costo_t> > mejora_iters_grasp;
    vector<pair<costo_t,costo_t> > costo_camino_en_iteraciones;//costos w1, w2

    do{
        tiempo_golosa_randomized=0;
         MEDIR_TIEMPO_PROMEDIO(
            camino = g.obtener_solucion_golosa_randomizada(modo_golosa, parametro_beta);
            //cout << "Solucion inicial de la greedy:" << endl;
            //camino.imprimir_camino(cout);
        , 1, &tiempo_golosa_randomized);

        //la sol greedy rand a veces da cosas no factibles, asi que verifico:
        //g.hay_solucion() nos indica si existe una sol factible(greedy rand lo setea en false si el minimo dijktra sobre w1 > limit_w1)
        //(camino.obtener_costo_total_w1_camino() <= g.obtener_limite_w1()); para chequear la validez del camino final de greedy rand
        sol_valida_greedy = (camino.obtener_costo_total_w1_camino() <= g.obtener_limite_w1());
        if(g.hay_solucion()){
            if(sol_valida_greedy){//puede que la greedy randomized no encuentre solucion!
                //hago iteraciones de busqueda local hasta que no haya mejora(la funcion devuelve true si hubo mejora, false sino)
                g.establecer_camino_solucion(camino);

                //reseteo el contador de sol malas greedy rand
                cant_iters_sin_sol_greedy_rand_factible=0;

                int mejora_current_iteration = 0;
                uint64_t cant_iters_bqlocal = 0;
                double promedio_parcial_bqlocal = 0;
                double promedio_bqlocal = 0;
```

```
                do{
                    promedio_parcial_bqlocal = 0;
                    MEDIR_TIEMPO_PROMEDIO(
                        mejora_current_iteration = g.busqueda_local(modo_busqueda_local);
                        , 1, &promedio_parcial_bqlocal);
                    cant_iters_bqlocal++;
                    promedio_bqlocal += promedio_parcial_bqlocal;
                }while(mejora_current_iteration > 0);
                promedio_bqlocal = promedio_bqlocal /(double) cant_iters_bqlocal;

                //el tiempo de esta iteracion es la greedy randomized + bqlocal sobre esa sol inicial
                promedio += tiempo_golosa_randomized;
                promedio += promedio_bqlocal;

                //en este punto la bqlocal mejoro todo lo que pudo la sol. inicial obtenida con la randomized greedy
                //me fijo si esta solucion es mejor que la que tenia guardada, de ser asi actualizo el maximo y guardo la sol actual como la nueva mejor.
                camino = g.obtener_camino_solucion();
                costo_t costo_solucion_actual = camino.obtener_costo_total_w2_camino();

                //almaceno el costo total en esta iteracion
                if(costo_solucion_actual < costo_mejor_solucion){
                    //guardamos que en esta iteracion se encontro una mejora
                    if(cant_iters>0){//sino la primera vez pone infinito(costo_mejor_solucion es infinito al inicializar)
                        mejora_iters_grasp.push_back(make_pair(cant_iters, costo_mejor_solucion - costo_solucion_actual));
                    }
                    costo_mejor_solucion = costo_solucion_actual;
                    mejor_solucion = g.obtener_camino_solucion();
                    //reseteo el contador
                    cant_iters_sin_mejora = 0;
                }else{
                    //una iteracion consecutiva mas sin mejora
                    //guardamos que en esta iteracion se encontro una mejora
                    if(cant_iters>0){//sino la primera vez pone infinito(costo_mejor_solucion es infinito al inicializar)
                        mejora_iters_grasp.push_back(make_pair(cant_iters, 0));

                    }
                    cant_iters_sin_mejora++;
                }
                //costo de la mejor sol guardada
                costo_t costo_w1_mejor_solucion = mejor_solucion.obtener_costo_total_w1_camino();
                costo_t costo_w2_mejor_solucion = mejor_solucion.obtener_costo_total_w2_camino();
                costo_camino_en_iteraciones.push_back(make_pair(costo_w1_mejor_solucion, costo_w2_mejor_solucion));
                cant_iters++;
            }else{
                cant_iters_sin_sol_greedy_rand_factible++;
                //si ya no tengo mas chanches, modifico los parametros de la metaheuristica
                if(cant_iters_sin_sol_greedy_rand_factible >= RAND_GREEDY_BAD_ITERS_LIMIT){
                    //si es rcl por cantidad bajo el parametro beta
                    if(modo_golosa == RCL_POR_CANTIDAD){
                        if(parametro_beta>2){
                            parametro_beta--;
                        }else{
                            //cout << "[GRASP] Golosa deterministica seteada." << endl;
                            parametro_beta=1;
                            modo_golosa = RCL_DETERMINISTICO;
                        }
                    }else if(modo_golosa == RCL_POR_VALOR){
                        //si es rcl por valor, seteo greedy deterministica
                        //es muy delicado ajustar el porcentaje adaptativamente!
                        parametro_beta = 0; //es lo mismo que poner golosa deterministico!
                        modo_golosa = RCL_DETERMINISTICO;
                    }
                    //cerr << "[GRASP] Greedy randomized dio sol mala. Cambiando parametro_beta : " << parametro_beta << endl;
                    cant_iters_sin_sol_greedy_rand_factible = 0;
                }
            }
        }else{
```

```
            break;//no hay solucion factible
        }
        if(criterio_terminacion == CRT_K_ITERS_LIMIT_REACHED){
            condicion_terminacion = (cant_iters < ITERS_LIMIT);
        }else if(criterio_terminacion == CRT_K_ITERS_SIN_MEJORA){
            condicion_terminacion = (cant_iters_sin_mejora < ITERS_LIMIT);
        }
    }while(condicion_terminacion);
    promedio = promedio / (double) cant_iters;

    //imprimo mediciones en stderr
    if(g.hay_solucion()){
        cerr << g.obtener_cantidad_nodos() << " " << g.obtener_cantidad_aristas() << " " <<
            cant_iters << " " << promedio;
        //mejora en iteraciones
        ofstream evolucion_iteraciones;
        evolucion_iteraciones.open(FILE_ITERS_MEJORA);
        for(auto element : mejora_iters_grasp){
            evolucion_iteraciones << element.first << " " << element.second << endl;
        }
        evolucion_iteraciones.close();

        //evolucion absoluta
        evolucion_iteraciones.open(FILE_ITERS_COSTOS_ABSOLUTOS);
        for(uint i=1;i<=costo_camino_en_iteraciones.size();i++){
            //imprimr <iteracion> <costo_w2> <costo_w1>
            evolucion_iteraciones << i << " " << costo_camino_en_iteraciones[i-1].second << " "
                 << costo_camino_en_iteraciones[i-1].first << endl;
        }
        evolucion_iteraciones.close();

        costo_t costo_w2_inicial = costo_camino_en_iteraciones.front().second;
        costo_t costo_w2_final = costo_camino_en_iteraciones.back().second;
        costo_t mejora_total_costo_w2 = costo_w2_inicial - costo_w2_final;

        evolucion_iteraciones.open(FILE_ITERS_COSTOS_ABSOLUTOS_STATISTICS);
            evolucion_iteraciones << costo_w2_inicial << " " << costo_w2_final << " " <<
                mejora_total_costo_w2;
        evolucion_iteraciones.close();
    }

    g.serialize(cout, FORMATO_1_N_CLOSED);
}


//
    ------------------------------------------------------------------------------------------------------------


void ejecutar_greedy(Grafo &g){
    int limit_w1 = g.obtener_limite_w1();
    nodo_t nodo_src = g.obtener_nodo_origen();
    nodo_t nodo_dst = g.obtener_nodo_destino();
    #ifdef DEBUG_MESSAGES_ON
        //g.imprimir_lista_adyacencia(cout);
        //g.imprimir_matriz_adyacencia(cout);
    #endif
    #ifdef DEBUG_MESSAGES_ON
        cout << "Se requiere un camino entre (" << nodo_src << ") y (" << nodo_dst<< ") que no
            exceda el costo " << limit_w1 << endl;
    #endif
    double promedio_medicion = 0;
    Camino camino = g.obtener_camino_vacio();
    MEDIR_TIEMPO_PROMEDIO(
                camino = g.obtener_solucion_golosa();
    , CANT_ITERS_MEDICION, &promedio_medicion);

    g.establecer_camino_solucion(camino);
    if(g.hay_solucion()){
        cerr << g.obtener_cantidad_nodos() << " " << g.obtener_cantidad_aristas() << " " <<
            CANT_ITERS_MEDICION << " " << promedio_medicion;
        #ifdef DEBUG_MESSAGES_ON
            cout << endl << "Solucion obtenida con golosa" << endl;
            camino.imprimir_camino(cout);
        #endif
    }

```

```
    g.serialize(cout, FORMATO_1_N_CLOSED);
}

// ----------------------------------------------------------------------------------------------------------------



#include "grafo.h"
#include "timing.h"
#include <fstream>
#define FILE_ITERS_MEJORA "evolucion_iteraciones.txt"
#define FILE_ITERS_COSTOS_ABSOLUTOS "costos_absolutos_iteraciones_bqlocal.txt"
#define FILE_ITERS_COSTOS_ABSOLUTOS_STATISTICS "costos_absolutos_iteraciones_bqlocal_analisis.txt"

void ejecutar_busqueda_local(Grafo &g);

// ---------------- Main ------------------------------------
int main(int argc, char **argv){
    list<Grafo> instancias = Grafo::parsear_varias_instancias(FORMATO_1_N_CLOSED);
    uint64_t instance_number = 1;
    for(Grafo &g : instancias){
        #ifdef DEBUG_MESSAGES_ON
            cout << endl << endl << "Aplicando busqueda local a la " << instance_number << "-esima instancia de input..." << endl;
        #endif
        ejecutar_busqueda_local(g);
        instance_number++;
    }
    return 0;
}

void ejecutar_busqueda_local(Grafo &g){
//    #ifdef DEBUG_MESSAGES_ON
//        g.imprimir_lista_adyacencia(cout);
//        //g.imprimir_matriz_adyacencia(cout);
//    #endif

    costo_t limit_w1 = g.obtener_limite_w1();
    nodo_t nodo_src = g.obtener_nodo_origen();
    nodo_t nodo_dst = g.obtener_nodo_destino();

    //----------------------------------- Busco solucion inicial --------------------------

    vector<costo_t> costo_minimo;
    vector<nodo_t> predecesor;
    g.dijkstra(nodo_src, COSTO_W1, costo_minimo, predecesor);

    costo_t costo_src_dst = costo_minimo[nodo_dst];//costo(src, dst)

    //----------------------------------- Valido la factibilidad de la solucion----------------
    #ifdef DEBUG_MESSAGES_ON
        cout << "Se requiere un camino entre (" << nodo_src << ") y (" << nodo_dst<< ") que no exceda el costo " << limit_w1;
    #endif
    if(costo_src_dst == costo_infinito){
        cerr << "No existe solucion factible. No existe camino entre origen(" << nodo_src << ") y destino(" << nodo_dst << ") " << endl;
        g.establecer_se_encontro_solucion(false);
    }else if(costo_src_dst > limit_w1){
        cerr << "No existe solucion factible. El camino minimo respecto a w1 de origen(" << nodo_src << ") a destino(" << nodo_dst << ") es de costo " << costo_src_dst << endl;
        g.establecer_se_encontro_solucion(false);
    }else{
        //armar camino entre origen y destino y lo establezco como sol inicial
        Camino c = g.obtener_camino_vacio();
        nodo_t nodo = nodo_dst;
        do{
            //cout << nodo << " " ;
            c.agregar_nodo_adelante(nodo);
            nodo = predecesor[nodo];
        }while(nodo != predecesor_nulo);
        g.establecer_camino_solucion(c);

        //Camino c = g.obtener_solucion_golosa();
```

```
    //g.establecer_camino_solucion(c);

    //imprimo sol inicial.
    #ifdef DEBUG_MESSAGES_ON
        cout << ". Costo minimo obtenido: " << c.obtener_costo_total_w1_camino();
        cout << "...Ok!!" << endl;
        cout << "Camino inicial: ";
        c.imprimir_camino(cout);
        cout << endl;
    #endif

    //-------------------------------------- Comienzo la busqueda local --------------------
    //typedef enum tipo_ejecucion_bqlocal_t {BQL_SUBDIVIDIR_PARES, BQL_CONTRAER_TRIPLAS_A_PARES, BQL_MEJORAR_CONEXION_TRIPLAS, BQL_COMBINAR} tipo_ejecucion_bqlocal_t;
    tipo_ejecucion_bqlocal_t tipo_ejecucion = BQL_COMBINAR;

    //hago iteraciones de busqueda local hasta que no haya mejora(la funcion devuelve true si hubo mejora, false sino)
    uint64_t cant_iters = 0;
    double promedio_parcial = 0;
    double promedio = 0;
    int mejora_current_iteration = 0;
    vector<costo_t> mejora_en_iteraciones;
    vector<pair<costo_t,costo_t> > costo_camino_en_iteraciones;//costos w1, w2
    costo_t costo_w1_current_iteration = c.obtener_costo_total_w1_camino();
    costo_t costo_w2_current_iteration = c.obtener_costo_total_w2_camino();

    //ponemos el costo inicial de la iteracino 0 del camino
    mejora_en_iteraciones.push_back(0);//mejora 0 en iteracion 0
    costo_camino_en_iteraciones.push_back(make_pair(costo_w1_current_iteration, costo_w2_current_iteration));
    do{
        promedio_parcial = 0;
        MEDIR_TIEMPO_PROMEDIO(
            mejora_current_iteration = g.busqueda_local(tipo_ejecucion);
            , 1, &promedio_parcial);
        cant_iters++;
        promedio += promedio_parcial;
        costo_w1_current_iteration = g.obtener_costo_actual_w1_solucion();
        costo_w2_current_iteration = g.obtener_costo_actual_w2_solucion();
        costo_camino_en_iteraciones.push_back(make_pair(costo_w1_current_iteration, costo_w2_current_iteration));

        mejora_en_iteraciones.push_back(mejora_current_iteration);
    }while(mejora_current_iteration > 0);
    promedio = promedio /(double) cant_iters;

    #ifdef DEBUG_MESSAGES_ON
        switch(tipo_ejecucion){
            case BQL_SUBDIVIDIR_PARES:
                cout << "Finalizo la busqueda local insertando entre pares porque no se obtuvieron nuevas mejoras." << endl;
                break;
            case BQL_CONTRAER_TRIPLAS_A_PARES:
                cout << "Finalizo la busqueda local salteando entre triplas porque no se obtuvieron nuevas mejoras." << endl;
                break;
            case BQL_MEJORAR_CONEXION_TRIPLAS:
                cout << "Finalizo la busqueda local reemplazando entre triplas porque no se obtuvieron nuevas mejoras." << endl;
                break;
            case BQL_COMBINAR:
                cout << "Finalizo la busqueda local combinada porque no se obtuvieron nuevas mejoras." << endl;
                break;
        }
    #endif
    g.establecer_se_encontro_solucion(true);
    cerr << g.obtener_cantidad_nodos() << " " << g.obtener_cantidad_aristas() << " " << cant_iters << " " << promedio;

    //mejora en iteraciones
    ofstream evolucion_iteraciones;
    evolucion_iteraciones.open(FILE_ITERS_MEJORA);
    for(uint i=1;i<=mejora_en_iteraciones.size();i++){
        evolucion_iteraciones << i << " " << mejora_en_iteraciones[i-1] << endl;
```

```
        }
        evolucion_iteraciones.close();

        evolucion_iteraciones.open(FILE_ITERS_COSTOS_ABSOLUTOS);
        for(uint i=1;i<=costo_camino_en_iteraciones.size();i++){
            //imprimr <iteracion> <costo_w2> <costo_w1>
            evolucion_iteraciones << i << " " << costo_camino_en_iteraciones[i-1].second << " "
                << costo_camino_en_iteraciones[i-1].first << endl;
        }
        evolucion_iteraciones.close();

        costo_t costo_w2_inicial = costo_camino_en_iteraciones.front().second;
        costo_t costo_w2_final = costo_camino_en_iteraciones.back().second;
        costo_t mejora_total_costo_w2 = costo_w2_inicial - costo_w2_final;

        evolucion_iteraciones.open(FILE_ITERS_COSTOS_ABSOLUTOS_STATISTICS);
            evolucion_iteraciones << costo_w2_inicial << " " << costo_w2_final << " " <<
                mejora_total_costo_w2;
        evolucion_iteraciones.close();
    }

    g.serialize(cout, FORMATO_1_N_CLOSED);
}

//
    ______________________________________________________________________________________________
```