DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III - Recuperatorio

27 / 6 / 2014 — Algoritmos y Estructura de Datos III

Integrante	LU	Correo electrónico
Abdala, Leila	950/12	abdalaleila@gmail.com
Cingolani, Luis Ignacio	490/12	luiscingo@gmail.com
Nale, Sebastian Claudio	655/11	sebinale@gmail.com
Straminsky, Axel	769/11	axelstraminsky@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Informe de modificaciones

Solo se modifico el item 5.

2. Heurística GRASP

2.1. Descripción del algoritmo

El algoritmo corre Dijkstra entre d y todos los demás nodos para ambos pesos por separado, obteniendo así el peso de cada camino mínimo. Estos pesos mínimos se guardan en dos tablas de distancias $tDW1$ y $tDW2$.

Luego de hacer esto generamos un camino aleatorio como base para poder utilizar la búsqueda local sobre él, esto se logra de la siguiente manera: En el nodo inicial v, chequea todos los nodos desde los cuales es posible obtener una solución, es decir, todos los nodos para los cuales aún hay un camino para llegar al nodo destino sin superar la cota para w_1, llamamos a este conjunto de nodos *nodosPosibles*, luego se ordenan estos nodos de acuerdo a su peso entre este y el nodo destino, y se queda con un tercio de estos en *nodosPosibles*, eligiendo entre todos los que tienen camino mínimo de menor peso al nodo destino, en el caso en que no se encuentre ningún nodo posible se extenderá el camino usando el camino mínimo entre el nodo actual y el destino, y luego se le quitarán los ciclos al camino.

Luego el algoritmo elige aleatoriamente un nodo v' entre *nodosPosibles* que marca como el nodo inicial y repite el proceso. Cuando encuentra como adyacente al nodo destino, termina el proceso. Sin embargo, podría ocurrir que *nodosPosibles* sea vacío. Esto ocurre cuando se usaron todos los nodos de la componente conexa donde se encuentran el inicio y el destino que tienen un camino acotado, pero el nodo actual tiene una arista con el destino de peso w_1 que, sumado con el resto de las aristas, supera la cota pedida. En este caso, obtenemos el camino mínimo entre el nodo actual y el destino según w_1 y lo unimos con el construido hasta el momento, quitando los ciclos que se formen, y lo devolvemos.

Teniendo este camino como base podemos correr nuestra búsqueda local y guardar este camino como el *mejorCamino*. Repetimos el proceso de crear una base y de correr nuestra búsqueda local hasta que *iteracionesSinCambios* supere *cotaSinCambios* que es un número pasado por parámetro, cada vez que lo hacemos comparamos este nuevo *caminoActual* con el *mejorCamino* y si es mejor que este reemplazamos *mejorCamino* por *caminoActual* y volvemos *iteracionesSinCambios* a cero, caso contrario aumentamos *iteracionesSinCambios* en uno. Lo que esto logra es que siga intentando mejorar el *mejorCamino* hasta que pase una cantidad determinada de iteraciones sin conseguir una mejora.

2.2. Pseudocódigo y complejidad

A fin de hacer más amena la lectura del pseudocódigo utilizaremos el renombre n para la cantidad de vértices del grafo recibido como entrada, y m para la cantidad de aristas. El orden de complejidad de cada línea está escrito al final de la misma, las aclaraciones/justificaciones de los órdenes no triviales se encuentran debajo del pseudocódigo.

2.2.1. Pseudocódigo

GRASP(**in/out** *grafo*: `Grafo`, **in** *u*: `Vértice`, **in** *v*: `Vértice`, **in** *K*: `double`, **in** *semilla*: `Nat`, **in** *cotaIteraciones*: `Nat`) → *res* : `Camino`
Complejidad: $O(m * cotaIteraciones * (n^3 + n * m * (log(n) + log(m))))_1$

vector<double>tablaDistanciasW1 ← crear con n casillas inicializadas con el valor -1 /* $O(n)$ */
vector<double>tablaDistanciasW2 ← crear con n casillas inicializadas con el valor -1 /* $O(n)$ */
inicializar tablaDistanciasW1 y tablaDistanciasW2 con los pesos W1 y W2, respectivamente, de los caminos minimos desde destino a todos los nodos del grafo /* $O(m * log(n) + n^2)_2$ */
if *no hay un camino desde u con peso* $W_1 \leq K$ **then** /* $O(1)$ */
  return camino vacio /* $O(1)$ */
end if
gen ← inicializar semilla /* $O(1)$ */
Camino mejorCamino ← CaminoRandomGoloso(grafo, u, v, K, tablaDistanciasW1, tablaDistanciasW2, gen)
/* $O(m * (log(n) + n * log(m)) + n^2)$ */
mejorCamino ← BusquedaLocal(grafo, u, v, K, mejorCamino) /* $O(n * m * log(n))_4$ */
Nat iteracionesSinCambios ← 0 /* $O(1)$ */
while *iteracionesSinCambios* <*cotaIteraciones* **do**
/* $O(m * cotaIteraciones * (n^3 + n * m * (log(n) + log(m))))_3$ */
  Camino caminoActual ← CaminoRandomGoloso(grafo, u, v, K, tablaDistanciasW1, tablaDistanciasW2, gen) /* $O(m * (log(n) + n * log(m)) + n^2)$ */
  caminoActual ← BusquedaLocal(grafo, u, v, K, caminoActual) /* $O(n^3 + n * m * log(n))_4$ */
  if *mejorCamino.PesoTotalEnW2()* >*caminoActual.PesoTotalEnW2()* **then** /* $O(1)$ */
    mejorCamino ← caminoActual /* $O(n)$ */
    iteracionesSinCambios ← 0 /* $O(1)$ */
  else
    iteracionesSinCambios++ /* $O(1)$ */
  end if
end while
return mejorCamino /* $O(n)$ */

CaminoRandomGoloso(**in/out** *grafo*: `Grafo`, **in** *inicio*: `Vértice`, **in** *destino*: `Vértice`, **in** K: `double`, **in** *tablaDistanciasW1*: `vector<double>`, **in** *tablaDistanciasW2*: `vector<double>`, **in** *gen*:) → *res* : `Camino`
Complejidad: $O(m * (log(n) + n * log(m)) + n^2)_5$

```
Camino caminoRandom(grafo)                                                    /* O(n) */
Vertice verticeActual ← inicio                                                /* O(1) */
definir fraccionDeOpciones 3                                                  /* O(1) */
double kRestante ← K                                                          /* O(1) */
while verticeActual ≠ destino do                                              /* O(n * m * log(m))_6 */
    list<Arista>aristas ← agregar las aristas incidentes a verticeActual, que no estén ya en el camino y que
    además pertenezcan a un camino entre verticeActual y destino cuyo peso en W_1 sea menor a kRestante
    /* O(m)_7 */
    Nat cantOpciones ← aristas.size()                                         /* O(1) */
    if cantOpciones = 0 then                                                  /* O(m * log(n) + n) */
        Camino extension ← grafo.HallarCaminoMinimoEntre(verticeActual, destino, PesoW1)
        /* O(m * log(n)) */
        caminoRandom.UnirSinCiclos(extension, destino)                        /* O(n) */
        return caminoRandom                                                   /* O(n) */
    end if
    vector<AristaYPotencial>mejoresAristas ← Ordenar(aristas, tablaDistanciasW2, verticeActual)
    /* O(m * log(m)) */
    Nat maxOpciones ← grafo.CantidadDeVertices() / fraccionDeOpciones         /* O(1) */
    if cantOpciones >maxOpciones then                                         /* O(m) */
        cantOpciones ← maxOpciones                                            /* O(1) */
    end if
    std::uniformintdistribution<>dis(0, cantOpciones-1); Nat numeroDeArista ← dis(gen)   /* O(1) */
    /* Elegimos una arista al azar y actualizamos la informacion.                */
    Arista& aristaElegida ← mejoresAristas[numeroDeArista].first              /* O(1) */
    caminoRandom.AgregarArista(aristaElegida)                                 /* O(1) */
    verticeActual ← VerticeDeDestino(aristaElegida, verticeActual)            /* O(1) */
    kRestante ← kRestante - aristaElegida.PesoW1()                            /* O(1) */
end while
return caminoRandom                                                           /* O(n) */
```

2.2.2. Aclaraciones

1) Este algoritmo es iterativo por lo que la complejidad total es la suma de las complejidades de cada linea.

2) En en código, en esta parte llamamos a la función InicializarCaminosMinimos. A continuación agregamos un breve análisis de la complejidad de esta función como explicación del orden de complejidad mencionado.

InicializarCaminosMinimos(**in/out** *grafo*: `Grafo`, **in** *destino*: `Vertice`, **in/out** *tablaDistanciasW1*: `vector<double>`, **in/out** *tablaDistanciasW2*: `vector<double>`)
Complejidad: $O(mlog(n) + n^2)$

```
/* tablaDistancias tiene que tener en la posición i el valor de peso total del camino mínimo
   para el vértice i al destino.                                                          */
vector<Vertice>caminosW1 ← grafo.HallarCaminosMinimosDesde(destino, PesoW1)   /* O(m * log(n)) */
vector<Vertice>caminosW2 ← grafo.HallarCaminosMinimosDesde(destino, PesoW2)   /* O(m * log(n)) */
for Nat i ← 1; i ← grafo.CantidadDeVertices(); i++ do                         /* O(n^2) */
    Camino miCaminoW1 ← grafo.ArmarCaminoEntre(destino, i, caminosW1)         /* O(n) */
    Camino miCaminoW2 ← grafo.ArmarCaminoEntre(destino, i, caminosW2)         /* O(n) */
    if miCaminoW1.Longitud() ≠ 0 then
        tablaDistanciasW1[i] = miCaminoW1.PesoTotalEnW1()                     /* O(1) */
        tablaDistanciasW2[i] = miCaminoW2.PesoTotalEnW2()                     /* O(1) */
    end if
end for
tablaDistanciasW1[destino] ← 0                                                /* O(1) */
tablaDistanciasW2[destino] ← 0                                                /* O(1) */
```

3) La complejidad del ciclo depende de la complejidad del código que se ejecuta en cada iteración, que es $O(n^3 + n * m * (log(n) + log(m)))$, y de la cantidad de veces que se ejecuta este código, es decir, cuantas veces itera. En principio, la cantidad de veces que itera es *cotaIteraciones*, pero dentro del ciclo podría pasar que la función variante del ciclo vuelva al punto inicial. Esto sucede cuando la búsqueda local me devuelve un mejor camino que el almacenado en camino actual, léase mejor como de menor peso, lo que puede suceder m veces.

Esto es porque búsqueda local no me puede devolver infinita cantidad de caminos mejores, sino que está acotada por la cantidad de caminos fundamentalmente distintos que les pasemos (léase como fundamentalmente distintos que no compartan aristas que generen una solución óptima local), debido a que caminos no distintos fundamentalmente generan la misma solución óptima local. Luego, la cantidad de caminos fundamentalmente distintos puede ser acotada por m ya que en el peor de los casos los caminos fundamentalmente distintos no tienen ninguna arista en común.

Luego, la búsqueda local solo puede devolverme m soluciones óptimas distintas, así que el índice del ciclo solo puede volver a inicializarse m veces. Luego, la complejidad del ciclo es la complejidad del código interior por $m *$ *cotaIteraciones* que es la cantidad máxima de veces que puede iterar.

4) Por el análisis de complejidad presentado en el item 5.2 del TP3 original.

5) Este algoritmo es iterativo por lo que la complejidad total es la suma de las complejidades de cada linea.

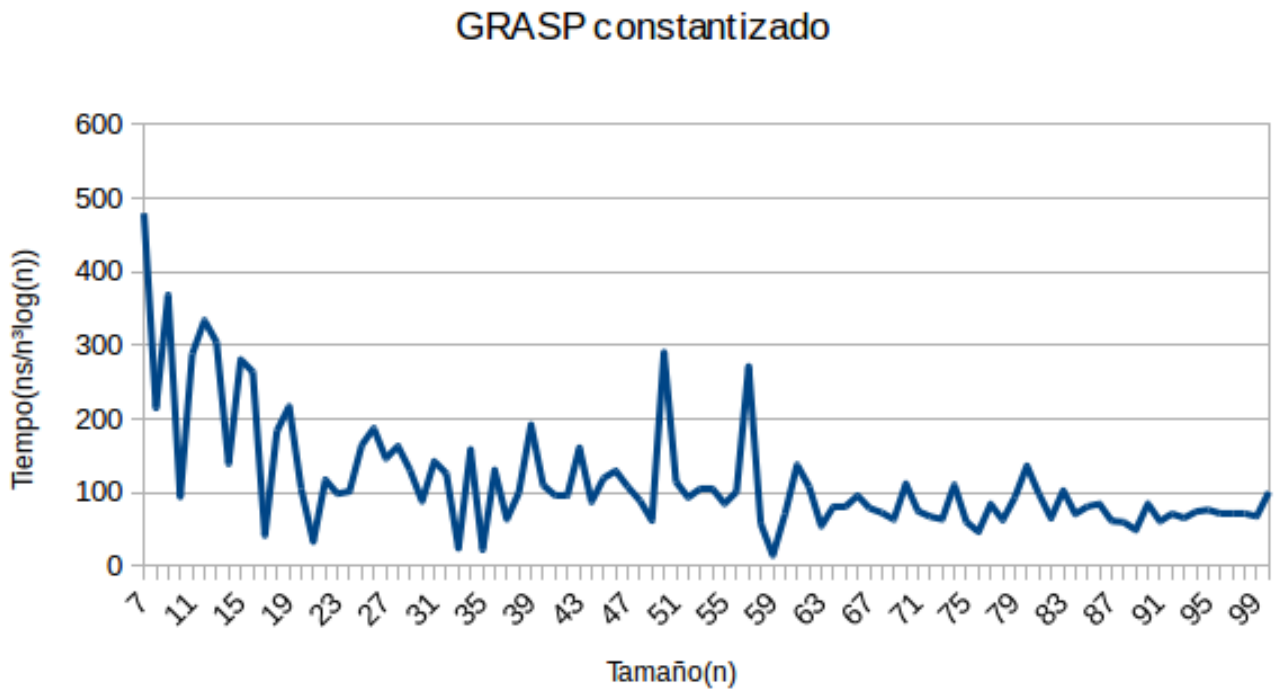
6) La complejidad del ciclo es $O(n * mlog(m))$ si itera sobre todos los nodos. Si no lo hace hay un costo de $O(mlog(n) + n)$ al cortarlo. Entonces su complejidad de peor caso es $O(m(log(n) + nlog(m)) + n)$.

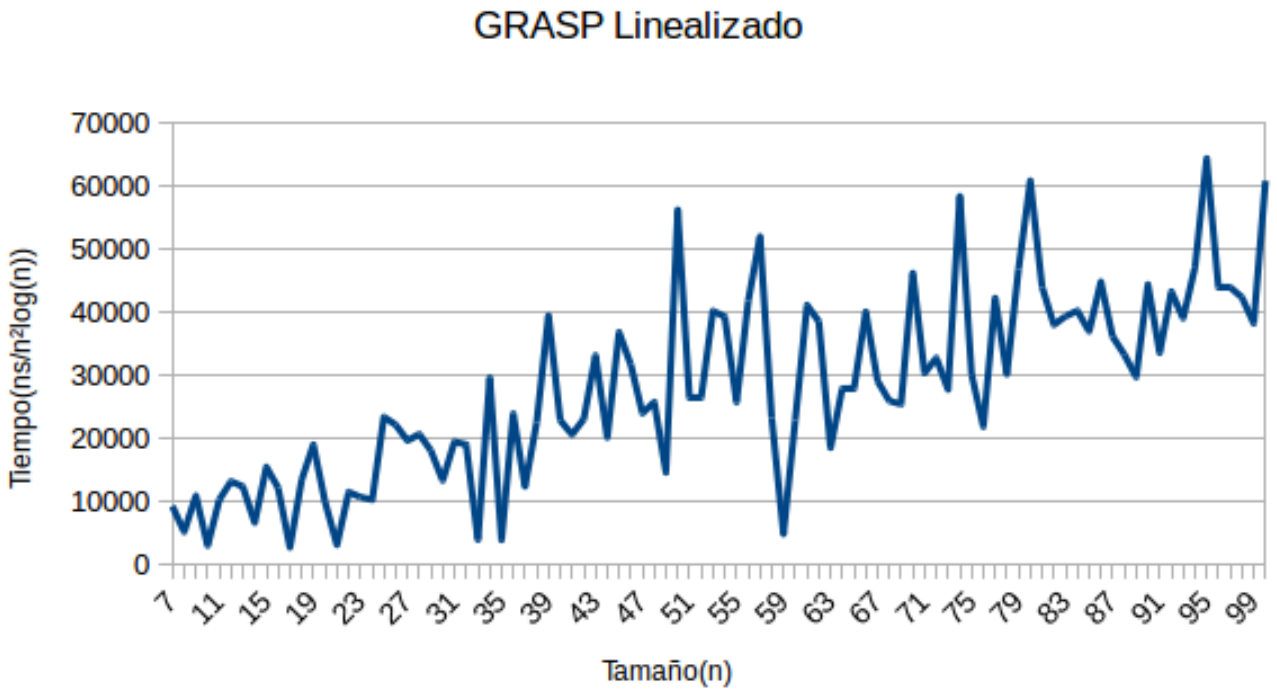
7) En el código, en esta parte llamamos a la función FiltrarInfactibles. A continuación agregamos un breve análisis de la complejidad de esta función como explicación del orden de complejidad mencionado.

FiltrarInfactibles(**in** *aristas* : `list<Arista>`, **in** *verticeActual* : `Vertice`, **in** *kRestante* : `double`, **in** *miCamino* : `Camino`, **in** *tablaDistanciasW1* : `vector<double>`) → *res* : `list<Arista>`
Complejidad: $O(m)$

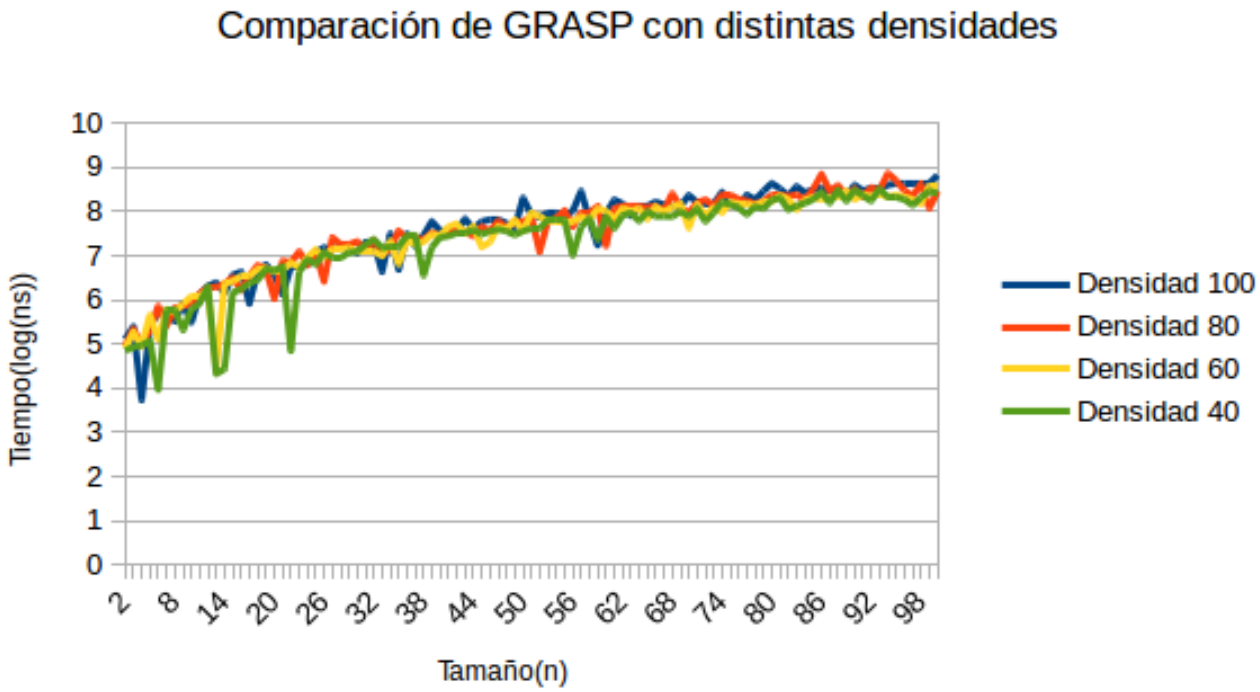
```
for Iterador(list) it ← aristas.begin(); it ≠ aristas.end(); it++ do        /* O(m) */
    double pesoOpcion ← it→PesoW1()                                         /* O(1) */
    Vertice verticeDeOpcion ← VerticeDeDestino(*it, verticeActual)          /* O(1) */
    bool opcionNoValida ← miCamino.enCamino(verticeDeOpcion) ∨ NoHayCamino(verticeDeOpcion,
    tablaDistanciasW1, kRestante-pesoOpcion)                                /* O(1) */
    if ¬opcionNoValida then                                                 /* O(1) */
        res.AgregarAtras(*it)                                               /* O(1) */
    end if
end for
return res                                                                  /* O(m) */
```

2.3. Experimentación





Ambos gráficos fueron realizados con grafos completos, y por lo tanto $m = \frac{n*(n-1)}{2}$



2.3.1. Conclusiones

La experimentación se realizó con el parámetro $cantIteraciones = 100$, y se dividieron los tiempos de ejecución por la complejidad $O(cotaIteraciones * (n^3 + n*m*(log(n) + log(m))))$. Una observación interesante es que la complejidad de peor caso de nuestro algoritmo es $O(m * cotaIteraciones * (n^3 + n*m*(log(n) + log(m))))$ ya que no es fácil acotar cuantas veces se correrá el algoritmo de búsqueda local hasta que no suceda un cambio en $cotaIteraciones$ iteraciones.

Sin embargo en los primeros dos gráficos podemos notar que si utilizamos $O(cotaIteraciones * (n^3 + n*m*(log(n) + log(m))))$ para constantizar y linealizar nos quedan gráficos muy parecidos a los que uno esperaría si esa fuera su complejidad, por lo que podemos suponer que $O(cotaIteraciones * (n^3 + n*m*(log(n) + log(m))))$ es su complejidad promedio. También podríamos decir que dadas las mediciones la complejidad del peor caso rara vez se alcanza.

En el último gráfico, se ve que GRASP corre ligeramente más rápido a menor densidad, aunque la diferencia de tiempo de ejecución entre las distintas densidades es menor que la que uno intuitivamente esperaría. Pero tiene sentido al considerar que en el caso promedio el único lugar en el que m aparece en la complejidad del algoritmo es en la búsqueda local, que tiene complejidad $O(n^3 + n*m*log(n))$, dado que m esta acotada por n^2 entonces la complejidad se parece mucho independientemente de el tamaño de la m.

Esto muestra nuevamente que la complejidad de peor caso es una cota muy superior a la complejidad promedio, ya que si la cantidad de ejes afectara tanto a la complejidad debería verse reflejada en el ultimo gráfico.

Observación: Para tomar las mediciones, se corre GRASP 20 veces para cada n y se toma el mínimo de todos esos tiempos, en lugar de hacer un promedio.