Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Tópicos Avanzados en Optimización Combinatoria y Teoría de Grafos

Trabajo Práctico

HEURÍSTICAS Y COTAS INFERIORES PARA EL PROBLEMA DEL VIAJANTE DE COMERCIO

Alumno: Rodríguez Jeremías

23 de junio de 2020

1. Resultados Numéricos

Las siguientes dos tablas resumen los costos de las 20 instancias de TSP consideradas en este trabajo práctico:

- Costo de una solución óptima, calculado con Concorde.
- Costo de las heurísticas vecino más cercano, vecino más cercano mejorado, inserción más cercana y lejana.
- Algoritmo de aproximación (Double spanning tree, ejercicio 11)

Instancia	**Óptimo**	**NN**		**NN mejorado**		**Inserción L**		**Inserción C**		**Aproximado**	
		val	**coc**	**val**	**coc**	**val**	**coc**	**val**	**coc**	**val**	**coc**
a280.tsp	2579.00	3148.11	1.22	3094.28	1.20	2971.37	1.15	3069.03	1.19	3629.72	1.41
att532.tsp	86729.00	112099.45	1.29	102786.05	1.19	94676.74	1.09	106814.53	1.23	116872.18	1.35
berlin52	7542.00	8980.92	1.19	8182.19	1.08	8120.63	1.08	9043.19	1.20	10115.33	1.34
bier127	118282.00	135751.78	1.15	133970.65	1.13	127794.76	1.08	145544.08	1.23	158503.66	1.34
ch130	6110.00	7575.29	1.24	7198.74	1.18	6654.73	1.09	7283.95	1.19	8128.76	1.33
eil101	629.00	825.24	1.31	736.37	1.17	687.40	1.09	746.16	1.19	829.54	1.32
kroA150	26524.00	33609.87	1.27	31482.02	1.19	28789.03	1.09	31588.40	1.19	35122.57	1.32
lin318	42029.00	54033.58	1.29	49215.61	1.17	45874.74	1.09	52299.12	1.24	58145.44	1.38
pr299	48191.00	59899.01	1.24	58288.15	1.21	52545.30	1.09	60236.62	1.25	64646.04	1.34
rat575	6773.00	8449.32	1.25	7972.33	1.18	7542.52	1.11	8253.05	1.22	9438.93	1.39
AVG			1.24		1.17		1.10		1.21		1.35

Instancia	**Óptimo**	**NN**		**NN mejorado**		**Inserción L**		**Inserción C**		**Aproximado**	
		val	**coc**	**val**	**coc**	**val**	**coc**	**val**	**coc**	**val**	**coc**
1	773.00	1011.15	1.31	899.14	1.16	818.65	1.06	949.79	1.23	1012.73	1.31
2	778.00	994.99	1.28	934.92	1.20	847.78	1.09	997.46	1.28	1082.36	1.39
3	784.00	1041.34	1.33	887.43	1.13	818.77	1.04	937.26	1.20	1092.40	1.39
4	727.00	970.30	1.33	821.31	1.13	782.47	1.08	925.26	1.27	1014.99	1.40
5	764.00	880.56	1.15	851.45	1.11	811.62	1.06	883.63	1.16	900.31	1.18
6	785.00	890.32	1.13	886.55	1.13	854.95	1.09	931.19	1.19	1056.12	1.35
7	804.00	972.43	1.21	891.47	1.11	895.35	1.11	920.99	1.15	968.14	1.20
8	789.00	993.17	1.26	936.43	1.19	861.04	1.09	919.18	1.16	1046.33	1.33
9	803.00	980.51	1.22	945.73	1.18	835.76	1.04	995.89	1.24	1052.27	1.31
10	746.00	928.76	1.24	854.66	1.15	804.75	1.08	907.67	1.22	1042.19	1.40
AVG			1.25		1.15		1.07		1.21		1.33

2. Interpretación de los resultados

- De forma consistente, tanto en las instancias aleatorias como las instancias de TSPLIB, el órden de performance respecto al óptimo de los métodos estudiados es:
  1. Inserción más lejana
  2. Vecino más cercano mejorado
  3. Inserción más cercana
  4. Vecino más cercano
  5. Algoritmo aproximado árbol doble
- La mejora a la heurística de vecino más cercano (ejercicio 9) incurre en una mejora del 10 % respecto al vecino más cercano con $v_1 = 1$. Esto muestra que decisiones arbitrarias como 'qué nodo es el primero' pueden tener un considerable impacto en el costo final.
- En cuanto a las heurísticas de inserción, se verifica lo descripto en el libro de Cook, dado que inserción más lejana supera ampliamente a inserción más cercana.

- El algoritmo aproximado estudiado no da buenos resultados en comparación a las heurísticas. Por otro lado, vemos que los costos obtenidos para este algoritmo respetan la cota teórica del peor caso, pues los cocientes son menores a 2.

3. Comparación con otros trabajos

En esta sección compararé los valores promedio obtenidos en ambas tablas de la sección anterior, con los valores promedios descriptos en LCO, capítulo 7. Adicionalmente, agregué otra columna con los números reportados por Junter et al (1995), dado que LCO no reporta números promedio para algunos de los métodos estudiados.

Algoritmo	**Avg TSPLIB**	**Avg sintética**	**LCO**	**Junter et al**
Vecino más cercano	1.25	1.25	1.26	1.24
Farthest Insertion	1.1	1.07	1.16	1.1
Closest Insertion	1.21	1.21	-	1.2
Double Spanning Tree	1.35	1.33	-	1.38
Christofides	-	-	1.14	1.19

- Se puede observar valores obtenidos en la sección 2 se condicen con los valores reportados por LCO y Junter et al. Las pequeñas discrepancias en las filas probablemente se deban a que:
  1. Las columnas 2 y 3 están basadas en datasets muy pequeños (10 elementos). Incrementar la cantidad de instancias posiblemente estabilizará el resultado.
  2. Las instancias del dataset sintético tienen siempre 100 puntos, quizás habría que introducir cierta varianza en la cantidad de puntos de cada instancia para obtener mejores generalizaciones.
- Incluí una fila con los números referidos a la heurística de Christofides, dado que es similar al alogritmo aproximado del ejercicio 11 ('Double spanning tree'). Ambos algoritmos utilizan un árbol recubridor mínimo y el concepto de atajo, pero podemos apreciar que el caso promedio de Christofides es ampliamente superior.

4. Ejercicio 11 - Peor caso

Demostraré que el algoritmo del ejercicio 11 es un algoritmo de aproximación con factor 2. Para ello, demostraré dos sencillos lemas primero.

De aquí en adelante, $G = (V, E)$ es una instancia de TSP con costos positivos $c \in \mathbb{R}^{+E}$ que verifican la desigualdad triangular, H^* un tour óptimo y T^* un árbol de expansión de costo mínimo de G

Lema 1: El árbol de expansión de costo mínimo determina una cota inferior al costo del tour óptimo:

$$c(T^*) \leqslant c(H^*)$$

d) Consideremos el tour óptimo H^*. Si eliminamos cualquier arista, obtenemos un árbol recubridor T de G, cuyo costo total es menor o igual dado que las aristas tienen costo no negativo:

$$c(T) \leqslant c(H^*)$$

Finalmente, dado que T^* es un árbol de expansión de costo mínimo, su costo será menor al de T:

$$c(T^*) \leqslant c(T) \leqslant c(H^*)$$

Lema 2: Se define la opreación $shortcut(H, e_1, e_2)$, donde H es cualquier circuito euleriano de algún (multi)grafo G' con costos positivos que verifican la desigualdad triangular; y donde $e_1 = (v_1, v_2)$ y $e_2 = (v_2, v_3)$ son dos aristas consecutivas de H. *shortcut* consiste en eliminar ambas aristas de H y reemplazarlas por la arista (v_1, v_3). Se cumple que, para todo H y para todo par de aristas consecutivas $e_1 = (v_1, v_2)$ y $e_2 = (v_2, v_3)$ de H:

$$c(H) \geqslant c(shortcut(H, e_1, e_2))$$

Esto se debe a que $c(v_1, v_2) + c(v_2, v_3) \geqslant c(v_1, v_3)$ por la desigualdad triangular.

Finalmente: Sea D un tour generado por el algoritmo del ejercicio 11. Entonces, D es el resultado de la aplicación sucesiva de la operación shortcut al circuito Euleriano E. Las aristas de E son exactamente las aristas de un arbol de expansión mínimo de T^* de G, por duplicado. Entonces:

$$c(D) \leqslant c(E) = 2 * c(T^*) \leqslant 2 * c(H^*)$$

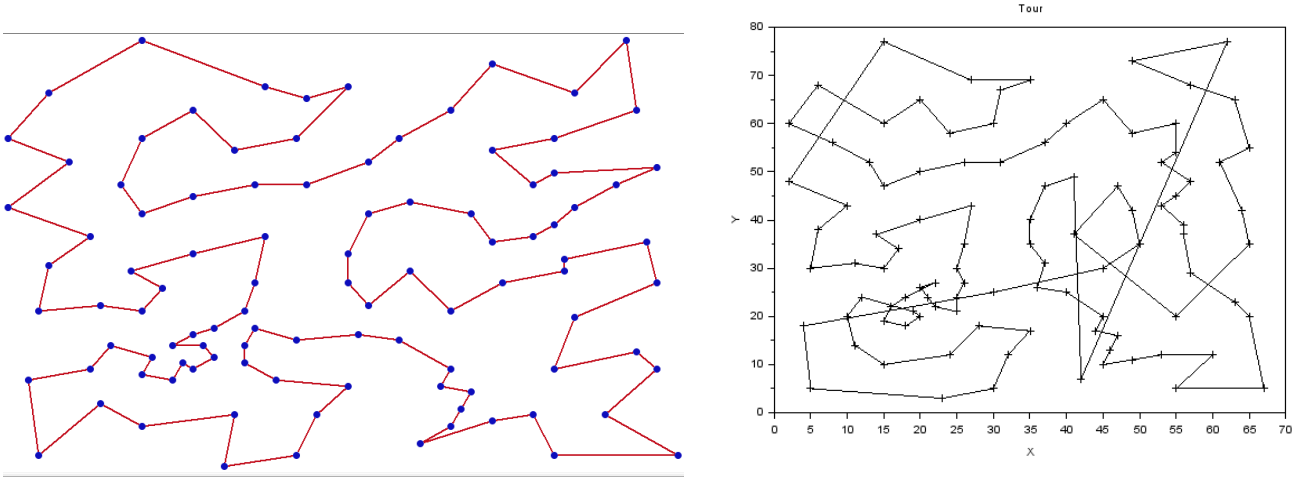
- La primera desigualdad vale por lema 2.
- La segunda igualdad vale por definición, todas las aristas del arbol mínimo son usadas exactamente dos veces.
- La tercera desigualdad vale por lema 1.

Por lo tanto, el algoritmo obtiene soluciones cuyo costo es a lo sumo dos veces el costo de un tour óptimo.

5. Gráficos

5.1. Peor caso en TSPLIB

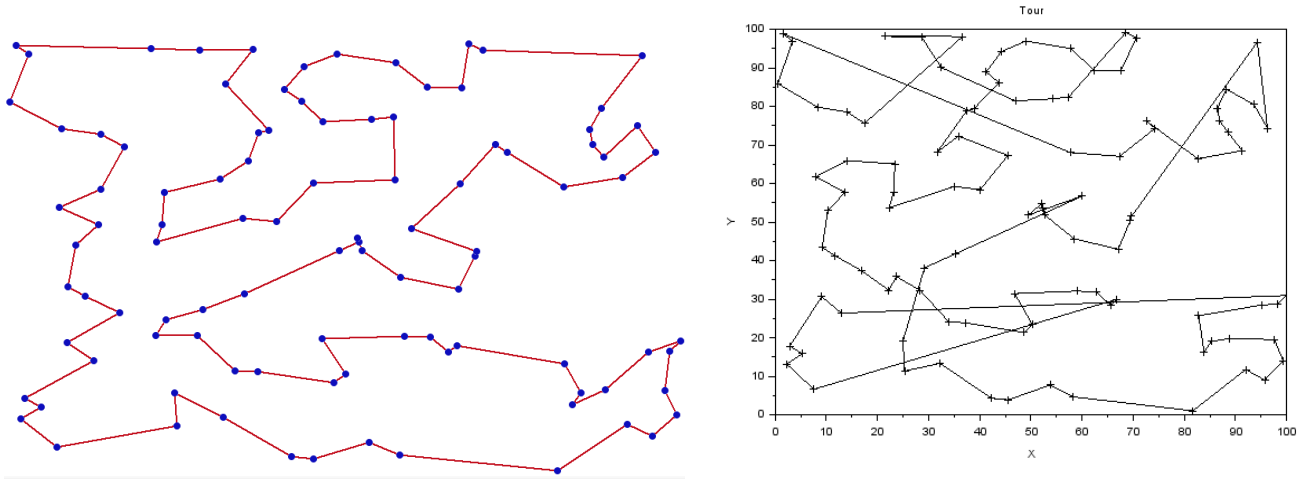
Los plots corresponden a la instancia eil101, a la izquierda el óptimo y a la derecha el vecino más cercano:



Podemos apreciar cómo la heurística agrega aristas extremadamente costosas al quedarse sin vecinos cercanos que estén realmente cerca. Esta es la principal debilidad del método.

5.2. Peor caso en data sintética

Los plots corresponden a la instancia aleatoria 3, a la izquierda el óptimo y a la derecha el vecino más cercano:



La misma apreciación sobre las aristas costosas aplica aquí.

6. Codigo fuente

- Las instancias aleatorias están en el la carpeta random_ds
- Las funciones a implementar se encuentran en sus respectivos .sci files

El código fuente nuevo también se incluye a continuación:

```
function [tour, valor] = vecino_mas_cercano_mejorado(A)
// Ejecuta el algoritmo ''mejorado'' del Vecino más cercano sobre la instancia TSP dada
// Entrada:
//  A = matríz de distancias de la instancia TSP
// Salida:
//  tour = vector con los vértices a ser recoridos por el tour
//  valor = valor del tour generado

[nodos, zzz] = size(A); // en "nodos" se guarda la cantidad de vértices de la instancia

valor = %inf
best_idx  = -1

for n=1:nodos
  [tmp_tour,tmp_valor] = vecino_mas_cercano(A,n);
  //disp("iter ",n, " value ",tmp_valor);
  if (tmp_valor < valor)
      valor = tmp_valor;
      best_idx = n;
      tour = tmp_tour;
  end
end


endfunction

function [tour, valor] = insercion_mas_cercana(A)
// Ejecuta el algoritmo de la Inserción más cercana sobre la instancia TSP dada
// Entrada:
//  A = matríz de distancias de la instancia TSP
// Salida:
//  tour = vector con los vértices a ser recoridos por el tour
//  valor = valor del tour generado


[nodos, zzz] = size(A); // en "nodos" se guarda la cantidad de vértices de la instancia

vertices_permitidos = 1:nodos; // al principio todos los vertices son permitidos

// Buscamos la MINIMA distancia entre 2 vértices, y agregamos ambos al tour
// (basta con buscar sobre los pares (u, v) tales que u < v)
minima_dist = %inf;
v1 = -1;
v2 = -1;
for u=1:(nodos-1)
    for v=(u+1):nodos
        if (minima_dist > A(u,v)) then
            minima_dist = A(u,v);
```

```
            v1 = u;
            v2 = v;
        end
    end
end
tour = [v1 v2];

// Prohibimos el uso de los vértices v1 y v2 en el futuro
vertices_permitidos(find(vertices_permitidos==v1)) = [];
vertices_permitidos(find(vertices_permitidos==v2)) = [];

for longitud=2:(nodos-1)
    // Por cada vertice permitido elegimos el vertice w mas cercano del tour
    wvector = -ones(1,nodos); // wvector va a tener el w correspondiente
                              // a cada v; al comienzo el vector vale -1
    for v=vertices_permitidos
        minima_dist = %inf;
        minimo_w = -1;
        for w=tour
            if (minima_dist > A(v,w)) then
                minima_dist = A(v,w);
                minimo_w = w;
            end
        end
        wvector(v) = minimo_w;
    end

    // Elegimos el próximo vértice "v" cuyo "w" es el MAS CERCANO
    min_dist = %inf;
    min_v = -1;
    for v=vertices_permitidos
        if (min_dist > A(v,wvector(v))) then
            min_dist = A(v,wvector(v));
            min_v = v;
        end
    end

    // Ahora elegimos donde insertarlo (entre dos vértices del tour que más mejore el tour)
    minima_suma = %inf;
    minimo_j = -1;
    for j=1:longitud
        // En "siguiente" colocamos el siguiente a "j", es decir j+1
        // (excepto para el último vértice del tour, en cuyo caso el siguiente es el primero)
        siguiente = j+1;
        if (j == longitud) then siguiente = 1; end
        // Calculamos la suma de distancias entre el v hallado, y los vertices del tour en "j" y
// "siguiente", menos la distancia entre "j" y "siguiente"
        suma = A(tour(j),min_v) + A(min_v,tour(siguiente)) - A(tour(j),tour(siguiente));
        if (suma < minima_suma) then
            minima_suma = suma;
            minimo_j = j;
        end
    end

    // Insertamos el (maxímo) "v" en la posición (mínimo) "j", en el tour; luego prohibimos
//dicho vértice.
    tour = [tour(1:minimo_j) min_v tour(minimo_j+1:$)];
```

```
    vertices_permitidos(find(vertices_permitidos==min_v)) = [];
end

// Calcula el valor del tour
valor = 0;
for j=1:nodos
    // En "siguiente" colocamos el siguiente a "j", es decir j+1
    siguiente = j+1;
    if (j == nodos) then siguiente = 1; end
    // Añadimos el costo de ir de "j" al siguiente
    valor = valor + A(tour(j),tour(siguiente));
end

endfunction

// Input
// circuito: Una lista de vértices formando un circuito
// unused: Una lista de aristas disponibles para expandir el circuito
//
// Output
// circuito: El mismo circuito input expandido
// unused: Las aristas que no se llegaron a usar
function [circuito,unused] = expandir_circuito(circuito,unused)

    // Elegir el vértice inicial desde donde apliaremos el circuito.
    circuit_size = length(circuito);
    [zz,n_aristas_no_usadas] = size(unused);

    if circuit_size==0 then
        initial_v = 1;    // Si el circuito inicial está vacío, elegimos cualquiera
    else
        // Si ya hay un circuito previo, necesitamos un vertice inicial que tenga alguna arista
        // incidiente no usada aún
        for (c_vert_idx=1:circuit_size)
            c1 = circuito(c_vert_idx)

            found = %F

            for (a_idx = 1:n_aristas_no_usadas)

                a = unused(:,a_idx)
                v1 = a(1)
                v2 = a(2)

                if (c1==v1 | c1==v2)
                    found = %T
                    initial_v = c1;
                    break;
                end
            end

            if (found)
               break;
            end

        end

        assert_checktrue(found);
```

```
end

// En este punto, tenemos un vertice en el circuito existente a partir del cuál podemos insertar
//un segundo circuito, hecho a partir de las aristas no usadas aún

last_v = initial_v;
tmp_circuit = [initial_v];

while (%T)
   [zz,n_aristas_no_usadas] = size(unused);
    // Buscar un arista NO USADA que sea incidente en last_v, y agregarla al ciruito
    for a_idx=1:n_aristas_no_usadas
        a = unused(:,a_idx);

        if (a(1)==last_v)
            tmp_circuit = [tmp_circuit, a(2)];
            last_v = a(2);
            unused(:,a_idx)=[]
            break;
        end

        if (a(2)==last_v)
            tmp_circuit = [tmp_circuit, a(1)];
            last_v = a(1);
            unused(:,a_idx)=[]
            break;
        end

    end

    if (last_v == initial_v) // Una vez que cerramos el circuito, paramos.
        break;
    end

end

// En este punto, tenemos un nuevo circuito en tmp_circuit que puede ser insetado
//en el circuito provisto como argumento
if circuit_size==0
    circuito = tmp_circuit;  // No hay nada que combinar si el circuito anterior estaba vacio
else
    // Si el circuito anterior tenía elementos, entonces debemos combinarlos:
    circuito_copia = circuito;
    circuito = [];

   // Primero copiamos el primer tramo del circuito original
    appended = %F;

    for (v = circuito_copia)
        if (v==initial_v & appended~=%T)
           circuito = [circuito,tmp_circuit];
           appended = %T;
        else
           circuito = [circuito,v]
        end
    end
```

```
    end

endfunction

function [circuito] = generar_circuito_euleriano(edges)

    unused = edges;
    circuito = [];

    while(unused ~= [])
        [circuito,unused] = expandir_circuito(circuito,unused);
    end

endfunction


function [tour, valor] = alg2aprox(A)
    // Ejecuta el algoritmo 2-aproximado
    // Entrada:
    //  A = matríz de distancias de la instancia TSP
    // Salida:
    //  tour = vector con los vértices a ser recoridos por el tour
    //  valor = valor del tour generado

    [nodos, zzz] = size(A); // en "nodos" se guarda la cantidad de vértices de la instancia


    // 1- Obtener un arbol de expansion mimimo
    [G, costos] = genera_grafo_costos(A);
    [F, valor]  = kruskal(G, costos);

    // 2 - Generar un circuito euleriano en (G, F++F)
    aristas = [F,F]
    circuito = generar_circuito_euleriano(aristas);

    // 3- Simplificarlo tomando atajos
    circuito_simplificado = []

    for v=circuito

        was_already_visited = %f;

        for (a=circuito_simplificado)
            if a==v
                was_already_visited = %t;
                break;
            end
        end

        if (~was_already_visited)
            circuito_simplificado = [circuito_simplificado,v]
        end

    end
```

```
    tour = circuito_simplificado;

    // 4 - Calculamos el valor
    valor = 0

    for (i = 1:(length(circuito_simplificado)-1))
        vi  = circuito_simplificado(i);
        vin = circuito_simplificado(i+1);
        cost = A(vi,vin);
        valor = valor + cost;
    end

    valor = valor + A(circuito_simplificado(length(circuito_simplificado)),circuito_simplificado(1))


endfunction
```