ALGORÍTMICA

PRÁCTICA 3: ALGORITMOS GREEDY

Memoria final de la práctica

Ignacio Aguilera Martos
Luis Balderas Ruiz
Diego Asterio de Zaballa Rodríguez
Miguel Ángel Torres López

ugr | Universidad de Granada

ÍNDICE

En lo que sigue, los miembros del grupo combinamos sistemas operativos y maquinas diferentes para experimentar de la forma más completa y variada la eficiencia de los algoritmos. Estas son las prestaciones de las máquinas:

- Luis: Fujitsu. Intel Core i5. Ubuntu 14.04
- Ignacio: Toshiba. Intel Core i7. Ubuntu 14.04
- Diego: Mac. Intel Core i7. OS X El Capitán
- Miguel Ángel: Toshiba. Intel Core i7. Windows 10

1. Cintas con pesos de acceso

El problema se basa en el problema original de ordenación de cintas, pero ahora, cada programa lleva asociado un peso que se corresponde con la frecuencia de uso de éste.

1.1 Explicación del problema y primera aproximación Greedy

Nuestro fin es almacenar un determinado número de programas en una cinta magnética de forma óptima. Cada programa tiene una longitud en kilobytes y un peso asociado a la cantidad de veces que vamos a utilizar ese programa. Nuestro reto es minimizar la sumatoria de los tiempos de acceso multiplicados por su correspondiente peso.
Para ello vamos a aplicar un algoritmo greedy que se basa en ordenar los programas en orden no creciente de sus pesos multiplicados por sus longitudes. A pesar de que el enunciado no lo pedía, mostraremos el diseño de dicho algoritmo y los tiempos de ejecución para distintos datos.
Se nos pide además comprobar que al tomar una función de selección a la hora de ordenar los programas resulta una ordenación no óptima.
Ésta última disertación se tratará en el siguiente apartado.

1.2 Errores de optimización según criterio de selección Greedy

Se desea minimizar el tiempo de acceso a determinados programas empleando un algoritmo voraz para la ordenación de los mismos en una cinta magnética. Se propone demostrar que el criterio de selección tomado no nos proporciona una solución óptima al problema, para ello daremos el correspondiente contraejemplo:

	P1	P2	P3
Пi	0,5	0,25	0,25
li	2	2	4

Tomando los criterios de selección que se piden en la práctica, es decir, en orden no creciente de los li, en no decreciente de los Пi y en orden no creciente de Пi/li para el caso propuesto, nos sale un tiempo mejor que si tomásemos el criterio de la multiplicación de ambos parámetros. Si bien, en el caso medio este criterio aporta la mejor aproximación a la solución.

1.3 Diseño del Algoritmo Greedy

El algoritmo Greedy diseñado ha sido basado en los principios básicos de los algoritmos greedy estándar. Su especificación viene detallada como sigue:

- Conjunto de candidatos. Conjunto de todos los programas que aún no fueron colocados en la cinta de memoria.
- Candidatos usados. Programas ya colocados en la cinta.
- Función solución. La solución se alcanza cuando todos los programas se han colocado en la cinta.
- Función de selección. En este caso, se selecciona el programa con menor producto de peso y tamaño del programa de entre los candidatos no seleccionados.
- Función objetivo. Minimizar el tiempo de acceso a todos los programas en función de los pesos que indican la frecuencia de acceso del mismo.

```
void solucionGreedy(vector<int>& tam, vector<double>& pesos)
{
  std::vector<int> aux1;
  std::vector<double> aux2;

  while (! tam.empty()) {
    size_t index = 0;
    for (size_t i = 0; i < tam.size(); i++) {
      if(tam[i]*pesos[i]<tam[index]*pesos[index])
        index = i;
```

```
    }

    aux1.push_back(tam[index]);
    aux2.push_back(pesos[index]);

    tam.erase(tam.begin()+index);
    pesos.erase(pesos.begin()+index);
  }
  tam = aux1;
  pesos = aux2;
}
```

Datos:

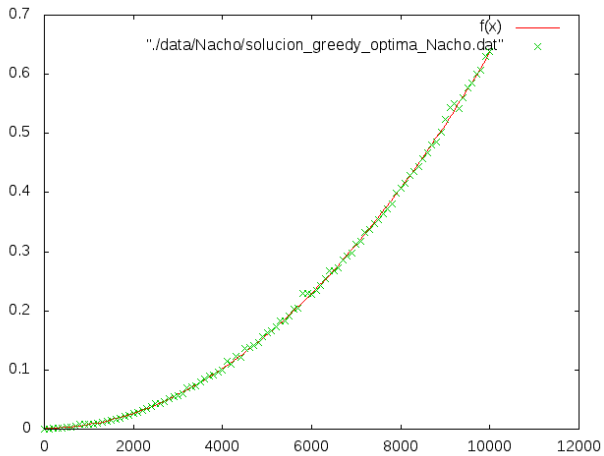
Presentamos aquí los datos del tiempo de ejecución en función de la cantidad de programas ordenados. En la primera columna se encuentra la cantidad de programas ordenados, a la derecha, el tiempo transcurrido durante la ejecución del algoritmo de ordenación. Como podemos observar en la gráfica, la eficiencia es cuadrático, bastante mejor que la solución obvia, la cual sería probar todas las combinaciones posibles y por lo tanto tendría eficiencia n!.

-Tabla y gŕafica de Nacho(Toshiba,Linux):

1	1.448e-06
101	2.3251e-05
201	4.4163e-05
301	6.4638e-05
401	8.8091e-05
501	0.000150763
601	0.000136857
701	0.000161163
801	0.000182353
901	0.000219793
1001	0.000222145
1101	0.000255275
1201	0.000289554
1301	0.000303106
1401	0.000326158
1501	0.00037003
1601	0.000394445
1701	0.000405358
1901	0.000694084
2001	0.000507769
2101	0.000573266
2201	0.000555689

2301	0.000570536
2401	0.000591993
2501	0.000623456
2601	0.0006408
2701	0.000651429
2801	0.000691036
2901	0.000729247
3001	0.000803079
3101	0.000784451
3201	0.000864756
3301	0.000904023
3401	0.00087565
3501	0.000922638
3601	0.000923679
3701	0.00097688
3801	0.00102227
3901	0.00104023
4001	0.00106472
4101	0.00109963
4201	0.00116829
4301	0.00118066
4401	0.00122682
4501	0.00125017
4601	0.00122889
4701	0.00172478
4801	0.00137799
4901	0.0013691
5001	0.00146179
5101	0.00147286
5201	0.0014886
5301	0.00164179
5401	0.00175905
5501	0.00161656
5601	0.00159279
5701	0.00166364
5801	0.00174661
5901	0.0017199
6001	0.00183617
6101	0.00188634
6201	0.00194099
6301	0.00197737
6401	0.00197436
6501	0.00215843
6601	0.00208401
6701	0.00218019

6801	0.00223222
6901	0.00238866
7001	0.002408
7101	0.00232052
7201	0.00240139
7301	0.00255687
7401	0.0024607
7501	0.00250561
7601	0.00260509
7701	0.00268728
7801	0.00265893
7901	0.00278936
8001	0.00293977
8101	0.0030318
8201	0.00291427
8301	0.00311451
8401	0.0031767
8501	0.00319791
8601	0.00341352
8701	0.00315575
8801	0.00331111
8901	0.00378304
9001	0.00363813
9101	0.00364826
9201	0.00366124
9301	0.0038169
9401	0.0037703
9501	0.0038607
9601	0.00400593
9701	0.00405131
9801	0.00411764
9901	0.00398986
10001	0.00432513



-Tabla y gŕafica de Luis(Fujitsu,Linux):

1	5.4174e-05
101	0.00031804
201	0.0010222
301	0.00241192
401	0.00357345
501	0.00522351
601	0.00528273
701	0.00518096
801	0.00691202
901	0.00649326
1001	0.00890535
1101	0.00949216
1201	0.012749
1301	0.0132475
1401	0.0161573
1501	0.0170031
1601	0.0201808
1701	0.0302286
1801	0.0262586
1901	0.0285862
2001	0.0305975
2101	0.0335396
2201	0.0365801
2301	0.0399691
2401	0.0727413
2501	0.0615204
2601	0.0554231
2701	0.0549157
2801	0.0589804
2901	0.0632054
3001	0.0673277
3101	0.0721431
3201	0.118458
3301	0.0880739
3401	0.0859388
3501	0.0909603
3601	0.0959603
3701	0.102113

3801	0.161642
3901	0.160992
4001	0.120095
4101	0.124093
4201	0.12959
4301	0.136294
4401	0.142768
4501	0.148939
4601	0.156321
4701	0.162084
4801	0.169128
4901	0.176787
5001	0.233622
5101	0.191428
5201	0.198413
5301	0.205733
5401	0.213153
5501	0.221226
5601	0.230014
5701	0.237235
5801	0.246074
5901	0.254184
6001	0.264092
6101	0.271848
6201	0.280758
6301	0.289581
6401	0.355343
6501	0.308136
6601	0.318563
6701	0.326839
6801	0.337519
6901	0.346908
7001	0.357356
7101	0.367044
7201	0.378277
7301	0.388313
7401	0.398274
7501	0.410117
7601	0.421207
7701	0.431831
7801	0.443264
7901	0.454709

8001	0.467162
8101	0.477457
8201	0.489565
8301	0.501337
8401	0.512922
8501	0.585817
8601	0.537686
8701	0.550587
8801	0.638904
8901	0.576116
9001	0.588328
9101	0.602034
9201	0.615866
9301	0.628733
9401	0.64255
9501	0.65606
9601	0.669717
9701	0.756675
9801	0.697905
9901	0.711525
10001	0.725498

-Tabla y gŕafica de Miguel(Toshiba,Windows):

1	1.1871e-05
101	0.00015468
201	0.00037219
301	0.00058718
401	0.00074967
501	0.00155908
601	0.00132202
701	0.00317985
801	0.00413914
901	0.00424955
1001	0.00439384
1101	0.00794445
1201	0.0118671
1301	0.00851352
1401	0.0100132
1501	0.0114089
1601	0.0143764
1701	0.0190291

1801	0.0166353
1901	0.0211087
2001	0.0210445
2101	0.0280167
2201	0.0253617
2301	0.0311113
2401	0.0299264
2501	0.0320306
2601	0.0378631
2701	0.0401021
2801	0.0457488
2901	0.0489652
3001	0.0503568
3101	0.0514639
3201	0.0535491
3301	0.058013
3401	0.0650592
3501	0.0644995
3601	0.0689316
3701	0.0762895
3801	0.078095
3901	0.0824891
4001	0.0881145
4101	0.0872685
4201	0.0908993
4301	0.0958122
4401	0.104255
4501	0.105139
4601	0.110702
4701	0.11517
4801	0.115458
4901	0.120887
5001	0.128015
5101	0.132494
5201	0.13601
5301	0.142146
5401	0.147973
5501	0.155095
5601	0.162948
5701	0.160409
5801	0.172513
5901	0.172728
6001	0.18454
6101	0.181984
6201	0.193322

6301	0.197174
6401	0.20385
6501	0.215209
6601	0.215486
6701	0.226188
6801	0.222511
6901	0.240591
7001	0.26732
7101	0.285024
7201	0.280008
7301	0.296458
7401	0.281514
7501	0.291183
7601	0.31024
7701	0.307642
7801	0.296954
7901	0.299291
8001	0.314339
8101	0.320409
8201	0.372867
8301	0.359138
8401	0.364071
8501	0.366577
8601	0.389309
8701	0.38869
8801	0.401134
8901	0.447012
9001	0.415066
9101	0.41497
9201	0.425124
9301	0.434716
9401	0.464636
9501	0.515042
9601	0.514154
9701	0.512568
9801	0.478681
9901	0.47833
10001	0.495032

-Tabla y gŕafica de Diego(MacBook Pro,MacOS El Capitán):

1	1.1611e-05
101	0.000112658
201	0.000358196

301	0.000585189
401	0.0009796
501	0.00154908
601	0.00192202
701	0.00518485
801	0.00419114
901	0.00422355
1001	0.00496984
1101	0.00780545
1201	0.0108521
1301	0.00851352
1401	0.0100132
1501	0.0114089
1601	0.0143764
1701	0.0190291
1801	0.0166353
1901	0.0211087
2001	0.0210445
2101	0.0280167
2201	0.0253617
2301	0.0311113
2401	0.0299264
2501	0.0320306
2601	0.0378631
2701	0.0401021
2801	0.0457488
2901	0.0489652
3001	0.0503568
3101	0.0514639
3201	0.0535491
3301	0.058013
3401	0.0650592
3501	0.0644995
3601	0.0689316
3701	0.0762895
3801	0.078095
3901	0.0824891
4001	0.0881145
4101	0.0872685
4201	0.0908993
4301	0.0958122
4401	0.104255
4501	0.105139
4601	0.110702
4701	0.11517

4801	0.115458
4901	0.120887
5001	0.128015
5101	0.132494
5201	0.13601
5301	0.142146
5401	0.147973
5501	0.155095
5601	0.162948
5701	0.160409
5801	0.172513
5901	0.172728
6001	0.18454
6101	0.181984
6201	0.193322
6301	0.197174
6401	0.20385
6501	0.215209
6601	0.215486
6701	0.226188
6801	0.222511
6901	0.240591
7001	0.26732
7101	0.285024
7201	0.280008
7301	0.296458
7401	0.281514
7501	0.291183
7601	0.31024
7701	0.307642
7801	0.296954
7901	0.299291
8001	0.314339
8101	0.320409
8201	0.372867
8301	0.359138
8401	0.364071
8501	0.366577
8601	0.389309
8701	0.3886
8801	0.401134
8901	0.447012
9001	0.415066
9101	0.411004
9201	0.425124

9301	0.434716
9401	0.464636
9501	0.515042
9601	0.514154
9701	0.512568
9801	0.478681
9901	0.47833
10001	0.495032

2. VIAJANTE DE COMERCIO

2.1 Explicación del Problema

En su forma más sencilla, el problema del viajante de comercio (TSP, por Traveling Salesman Problem) se define como sigue: dado un conjunto de ciudades y una matriz con las distancias entre todas ellas, un viajante debe recorrer todas las ciudades exactamente una vez, regresando al punto de partida, de forma tal que la distancia recorrida sea mínima. Mas formalmente, dado un grafo G, conexo y ponderado, se trata de hallar el ciclo hamiltoniano de mínimo peso de ese grafo G.

2.2 Algoritmo Greedy del vecino más cercano

La primera estrategia propuesta es la del vecino más cercano. Para darle consistencia, identificamos las características propias del algoritmo greedy que lo definen:

- Conjunto de candidatos. Conjunto de todas las ciudades disponibles para visitar.
- Candidatos usados. Ciudades ya visitadas.
- Función solución. La solución se alcanza cuando se han recorrido todas las ciudades, es decir, si el número de ciudades visitadas coincide con el número de ciudades candidatas.
- Función de selección. En este caso, para cada ciudad C, se escoge aquella ciudad D del conjunto de candidatos tal que distancia(C,D) = min { distancia(P,C); siendo P una ciudad de los candidatos}.
- Función objetivo. Distancia total del recorrido de forma que sea óptima (mínima posible).

Para mayor especificación daremos su prediseño en pseudocódigo:

ALGORITMO TSP_vecino_mas_cercano:
Se toma una ciudad de los candidatos para empezar y se borra de los mismos.
Mientras que el número de usados < conjunto de candidatos
    ciudad_nueva=menorDistancia de una ciudad a otra.
    Inclusión de ciudad_nueva en el conjunto de usados.
    Borrado de ciudad_nueva en el conjunto de candidatos.

Presentamos ahora el código del mismo una vez diseñado:

```
void TSP::TSP_vecino_mas_cercano(vector<City>& solucion)
{
  solucion.push_back(ciudades[0]);
  vector<City> candidatos(ciudades);
  candidatos.erase(candidatos.begin());
```

```
  while((int)solucion.size() < nCiudades)
  {
    vector<City>::iterator it = menorDistancia(solucion.at(solucion.size()-1), candidatos);
    solucion.push_back(*it);
    candidatos.erase(it);
  }
}
```

Datos:

En esta ocasión, estamos más interesados en la solución que en el tiempo de ejecución. Los algoritmos greedy suelen ser bastante rápidos, el problema está en la solución no óptima que encuentra, así que mostraremos los "caminos" creados por el algoritmo entre las ciudades.

-Mapa de Nacho(Toshiba,Linux):

Solo pondremos el mapa de un ordenador, ya que al ser el mismo algoritmo, el camino hallado como solución es el mismo en todos los ordenadores.

2.3 Algoritmo Greedy de inserción

La segunda estrategia que vamos a utilizar para hallar la solución del problema también es Greedy. Para darle consistencia, identificamos las características propias del algoritmo greedy que lo definen:

- Conjunto de candidatos. Conjunto de todas las ciudades disponibles para visitar.
- Candidatos usados. Ciudades ya visitadas.
- Función solución. La solución se alcanza cuando se han recorrido todas las ciudades, es decir, si el número de ciudades visitadas coincide con el número de ciudades candidatas.
- Función de selección. En este caso, para dos ciudades ya elegidas, se busca la ciudad que cumple que la suma de las distancias a las dos distancias es el menor con respecto del resto de ciudades no elegidas.
- Función objetivo. Distancia total del recorrido de forma que sea óptima (mínima posible).

Propondremos ahora el pseudocódigo asociado a este algoritmo:

Tomamos los tres puntos más alejados del centro del cúmulo de ciudades.
Mientras candidatos no vacío
    Buscar los dos candidatos_usados con mayor distancia de conexión.
    Buscar la ciudad que minimiza la suma de las distancias con los dos candidatos.
    Insertar ciudad_nueva entre dichas ciudades.

Por último presentamos el código final del algoritmo realizado.

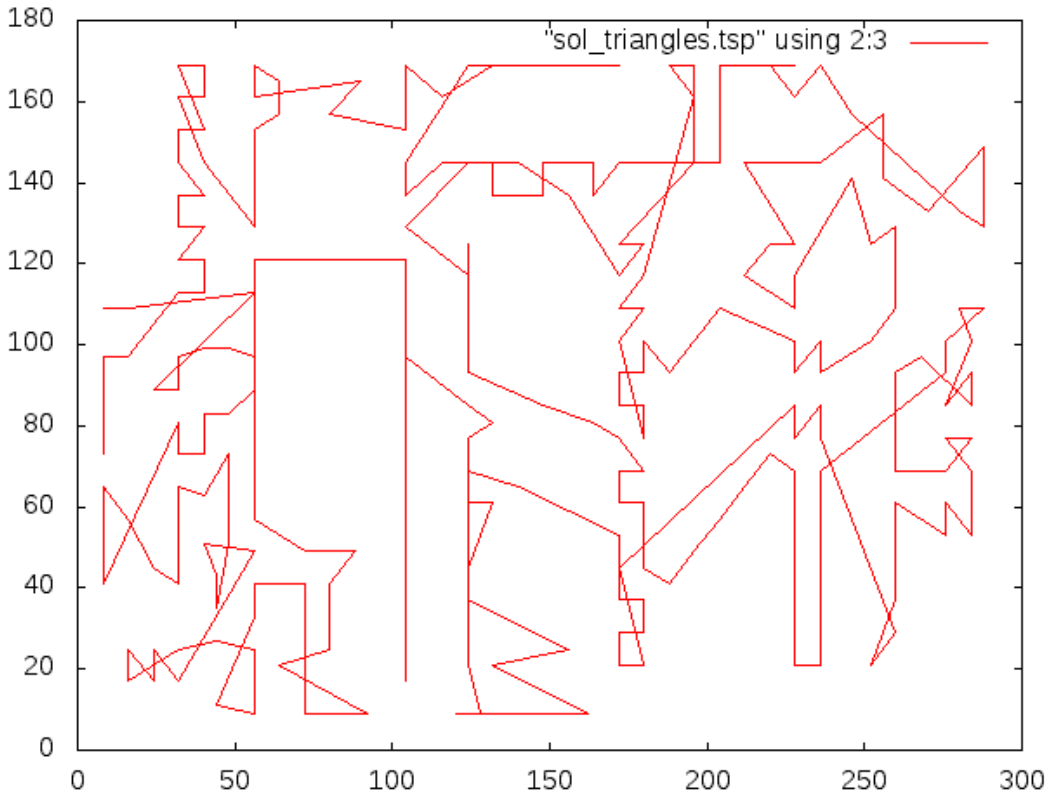
```
void TSP::TSP_triangles(vector<City>& solucion){

  vector<City> candidatos(ciudades);
  vector<City>::iterator minb = candidatos.begin();
  for (vector<City>::iterator it = candidatos.begin(); it != candidatos.end(); it++) {
    if ((*minb).coord_x>(*it).coord_x)
      minb = it;
  }
  solucion.push_back(*minb);
  candidatos.erase(minb);
  vector<City>::iterator maxb = candidatos.begin();
  for (vector<City>::iterator it = candidatos.begin(); it != candidatos.end(); it++) {
    if ((*maxb).coord_x<(*it).coord_x)
      maxb = it;
  }
  solucion.push_back(*maxb);
  candidatos.erase(maxb);


  vector<City>::iterator maxh = candidatos.begin();
  for (vector<City>::iterator it = candidatos.begin(); it != candidatos.end(); it++) {
```

```
    if ((*maxh).coord_y<(*it).coord_y)
      maxh = it;
  }
  solucion.push_back(*maxh);
  candidatos.erase(maxh);
City aux;
  while (!candidatos.empty()){
    vector<City>::iterator mayor_lado, nearest;
    find_max_edge(solucion, mayor_lado);
    find_nearest_point(solucion,mayor_lado,candidatos, nearest);
    aux = *nearest;
    vector<City>::iterator insertar;
    MejorInsercion(aux, solucion, insertar);
    solucion.insert(insertar,aux);
    candidatos.erase(nearest);
  }
}
```

Datos:

En esta ocasión, estamos más interesados en la solución que en el tiempo de ejecución. Los algoritmos greedy suelen ser bastante rápidos, el problema está en la solución no óptima que encuentra, así que mostraremos los "caminos" creados por el algoritmo entre las ciudades.

-Mapa de Nacho(Toshiba,Linux):



Solo pondremos el mapa de un ordenador, ya que al ser el mismo algoritmo, el camino hallado como solución es el mismo en todos los ordenadores

2.4 Algoritmo de intercambios

Esta tercera implementación propuesta es la menos seguro de todas debido a su aleatoriedad. Se basa en tomar al azar un recorrido de todas las ciudades que forme un ciclo y que, casi con seguridad, no será el óptimo. A partir de ahí se genera intercambios aleatorios de bloques de tres ciudades en tres ciudades. Se deshacen los intercambios si el recorrido resultante no es mejor que el anterior.

Presentamos aquí el código:

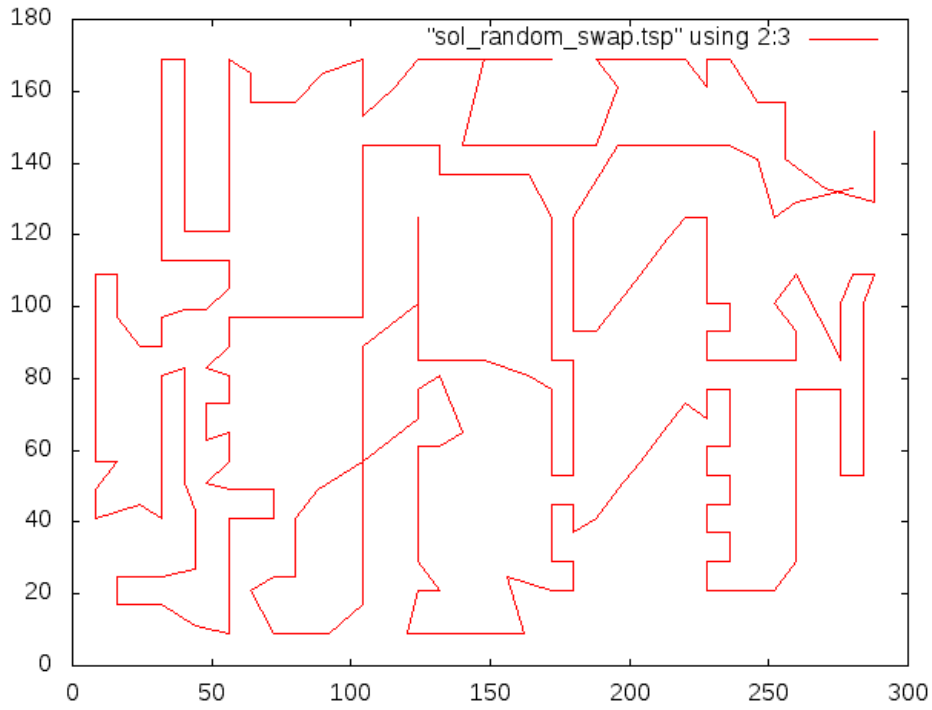
```
void TSP::TSP_RandomSwap(int n, vector<City>& solucion){
	City aux1,aux2,aux3;
	srand(time(0));
	int j,k;
	double distant,distlueg;
	for(vector<City>::iterator it = ciudades.begin(); it!=ciudades.end();it++){
		solucion.push_back(*it);
	}
	for(int i = 0; i < n; i++){
		j = nCiudades*rand()/(RAND_MAX + 1.0);
		do{
			k = (nCiudades)*rand()/(RAND_MAX + 1.0);
		}while((j > nCiudades - 3 && k < 3)||(k > nCiudades - 3 && j < 3));
		j = j%nCiudades;
		k = k%nCiudades;
		distlueg = distant = 0;
		for(vector<City>::iterator it = ciudades.begin(); it!=ciudades.end()-1;it++){
			vector<City>::iterator it2 = it;++it2;
			distant += distancia((*it).coord_x,(*it2).coord_x,(*it).coord_y,(*it2).coord_y);
		}

	distant+=distancia(ciudades[0].coord_x,ciudades[nCiudades1].coord_x,ciudades[0].coord_y
,ciudades[nCiudades-1].coord_y);
			aux1 = ciudades[j];
			aux2 = ciudades[j+1];
			aux3 = ciudades[j+2];
			ciudades[j] = ciudades[k];
			ciudades[j+1] = ciudades[k+1];
			ciudades[j+2] = ciudades[k+2];
			ciudades[k] = aux1;
			ciudades[k+1] = aux2;
			ciudades[k+2] = aux3;
		for(vector<City>::iterator it = ciudades.begin(); it!=ciudades.end()-1;it++){
			vector<City>::iterator it2 = it;
			++it2;
			distlueg += distancia((*it).coord_x,(*it2).coord_x,(*it).coord_y,(*it2).coord_y);
		}

	distlueg+=distancia(ciudades[0].coord_x,ciudades[nCiudades1].coord_x,ciudades[0].coord_
y,ciudades[nCiudades-1].coord_y);
		if(distant < distlueg){
			aux1 = ciudades[j];
			aux2 = ciudades[j+1];
			aux3 = ciudades[j+2];
			ciudades[j] = ciudades[k];
```

```
                        ciudades[j+1] = ciudades[k+1];
                        ciudades[j+2] = ciudades[k+2];
                        ciudades[k] = aux1;
                        ciudades[k+1] = aux2;
                        ciudades[k+2] = aux3;
                    }
                }

}
```

Datos:

En esta ocasión, estamos más interesados en la solución que en el tiempo de ejecución. Los algoritmos greedy suelen ser bastante rápidos, el problema está en la solución no óptima que encuentra, así que mostraremos los "caminos" creados por el algoritmo entre las ciudades.

- Mapa de Nacho(Toshiba,Linux):



La solución correcta de este algoritmo depende del azar, luego en cada ordenador el camino hallado será distinto. En un ordenador podría haberse realizado un intercambio terminal, es decir, un intercambio que conduce a una rama que no podría mejorar nada con intercambios, aunque no fuera la solución óptima total. Sin embargo no vemos necesario repetir las gráficas.

2.5 Algoritmo de Dijsktra

Esta tercera implementación propuesta es la menos seguro de todas debido a su aleatoriedad. Se basa en tomar al azar un recorrido de todas las ciudades que forme un ciclo y que, casi con seguridad, no será el óptimo. A partir de ahí se genera intercambios aleatorios de bloques de tres ciudades en tres ciudades. Se deshacen los intercambios si el recorrido resultante no es mejor que el anterior.

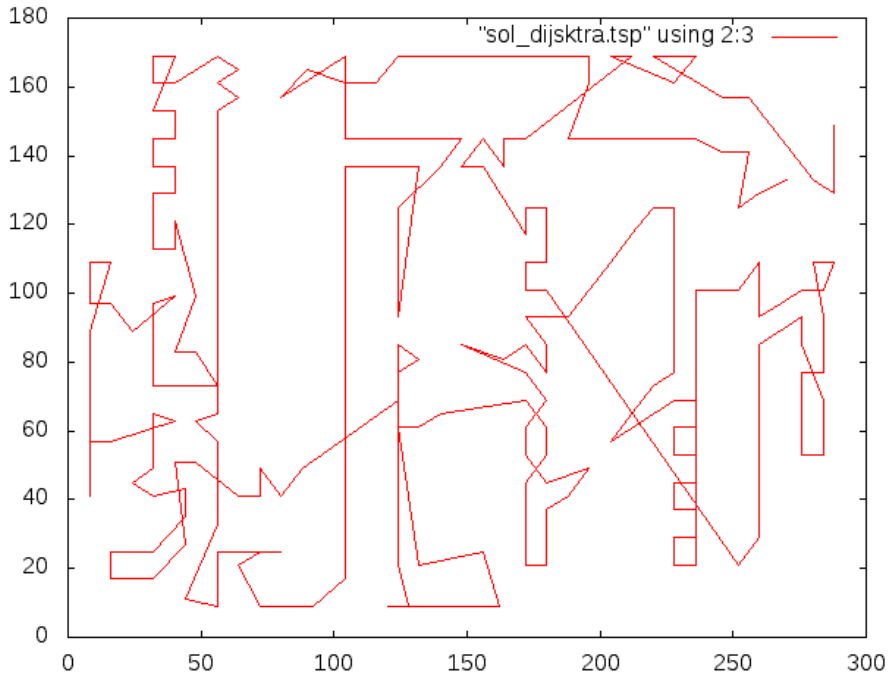
Presentamos aquí el código:

```
void TSP::Dijsktra(vector<City>& res)
{
  vector<City> candidatos(ciudades);
  res.push_back(candidatos[0]);
  candidatos.erase(candidatos.begin());

  while(candidatos.size()!=0)
  {
    double dist = INT_MAX;
    vector<City>::iterator min_dist;
    for(vector<City>::iterator it = res.begin();it!=res.end();++it)
    {
      pair<double, vector<City>::iterator> f =DevuelveMenorDistancia(*it, candidatos);
      if(dist>f.first)
      {
        min_dist = f.second;
        dist = f.first;
      }
    }
    vector<City>::iterator mejor;
    MejorInsercion(*min_dist,res,mejor);
    if(mejor==res.end())
      res.push_back(*min_dist);
    else
      res.insert(mejor, *min_dist);
    candidatos.erase(min_dist);
  }
}
```

Datos:

En esta ocasión, estamos más interesados en la solución que en el tiempo de ejecución. Los algoritmos greedy suelen ser bastante rápidos, el problema está en la solución no óptima que encuentra, así que mostraremos los “caminos” creados por el algoritmo entre las ciudades.

- Mapa de Nacho(Toshiba,Linux):



Solo pondremos el mapa de un ordenador, ya que al ser el mismo algoritmo, el camino hallado como solución es el mismo en todos los ordenadores.

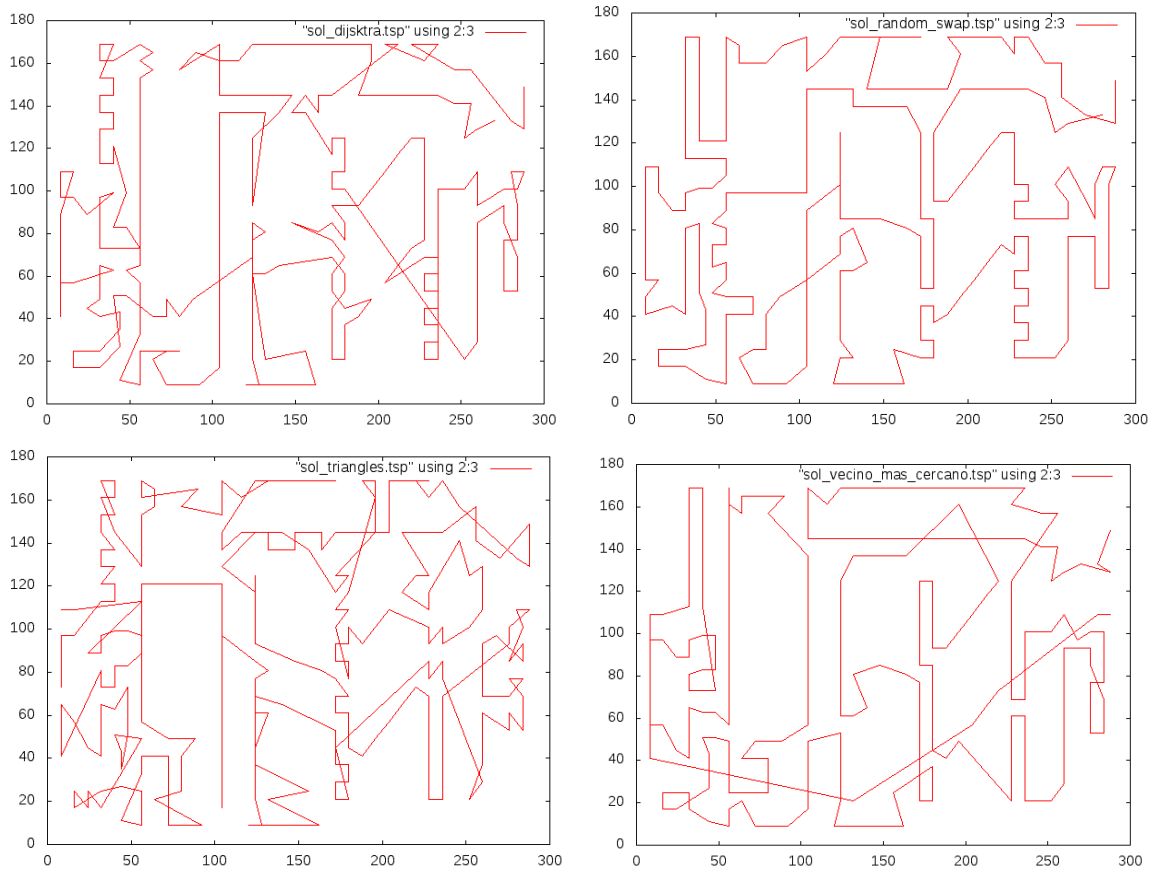
2.6 Comparación de algoritmos

Las soluciones halladas para un mismo mapa de ciudades por los distintos algoritmos nos arrojan los siguientes datos:

Distancia random swap: 2800,73
Distancia vecino más cercano: 3108,11
Distancia triangles: 3843,78
Distancia dijsktra: 3233,25

Hemos aplicado los cuatro algoritmos sobre un mismo mapa, para poder comparar bien cuál de ellos rinde mejor a gran escala, hemos cogido el mapa más grande, de

280 ciudades. El peor algoritmo es el triangular. Le sigue el algoritmo de Dijsktra y, a continuación, seguido por el del vecino más cercano. Además, cuando los intercambios realizados en el random swap tienden a infinito, el último algoritmo da las mejores prestaciones, aunque esta mejora es costosa en tiempo. El número de iteraciones es bastante alto si se quiere que la mejora sea significativa.



3. Bibliografía

-Algoritmos de la práctica anterior.
-Jose Luis Verdegay, “Curso de Teoría de Algoritmos”
-Brassard, Bradley, “Fundamentos de Algoritmia”
-www.gnuplot.com
-Knuth, “The art of computer programming”