Practica 3 : Algoritmos Voraces

El Problema del Viajante de comercio

Algoritmica - 2015/2016

Alejandro Campoy Nieves
David Criado Ramón
Nour Eddine El Alaoui
Luis Gallego Quero

Índice :

1 - Descripción del problema :

Básicamente el problema trata de un conjunto de ciudades y una matriz con las distancias entre todas ellas, tenemos un viajante que debe recorrer todas las ciudades solo una vez, y regresar al punto de partida, tal que la distancia recorrida sea la mínima.

Más formalmente, dado un grafo G, conexo y ponderado, se trata de hallar el ciclo hamiltoniano de mínimo peso de ese grafo.

Para resolver este problema tenemos que usar tres algoritmos distintos :

a) - Algoritmo del vecino más cercano.
b) - Algoritmo de inserción.
c) - Algoritmo basado en aristas.

2 - Heurística 1 : El vecino más cercano

❖ Enfoque Greedy

- Conjunto de candidatos: Ciudades a visitar
- Conjunto de seleccionados: Ciudades visitadas
- Función solución: Si hemos recorrido todas las ciudades
- Función de factibilidad: Que la ciudad no haya sido visitada
- Función de selección: Seleccionamos la ciudad más cercana
- Función objetivo: Recorrer todas las ciudades y volver a la ciudad inicial

Primero creamos una matriz con la distancias entre las ciudades, utilizando el algoritmo de Euclides.

Empezamos seleccionando la cuidad inicial, y vamos mirando por orden creciente de distancia desde la ciudad actual, se inserta la ciudad actual y la longitud del trayecto,y se vuelve a hacer el mismo procedimiento, haciendo uso de un filtro para asegurar que las cuidades introducidas no están repetidas.

Una vez hecho esto, insertamos de nuevo la primera ciudad, que es la única que se puede repetir.

2.1 El main y el código usado por los tres algoritmos

❖ *Para las tres estrategias, hemos hecho uso de :*

- Una función *LeerArchivo*, para leer los datos del archivo de entrada
- otra función *DistanciaCuidad*, para calcular la distancia Euclídea

```
vector<coordenadas> leerArchivo(char* nombre) {
    // Esta función sólo lee los datos del archivo de entrada.
    ifstream archivo(nombre);
    archivo.ignore(10);

    int N = 0;
    archivo >> N;


    if (!archivo.is_open() || N == 0){
        cerr << "Error de lectura de archivo";
        exit (-1);
    }

    vector<coordenadas> salida;
    salida.reserve(N);
    double x, y;
    for (int i = 0; i < N; ++i) {
        archivo >> x;
        archivo >> x;
        archivo >> y;
        salida.emplace_back((int)x,(int)y);
    }

    return salida;
}

int distanciaCiudad(ciudad c1, ciudad c2, const vector<coordenadas>& v) { // Distancia Euclídea
    return round(sqrt(pow(get<X>(v[c1])-get<X>(v[c2]),2) +
                      pow(get<Y>(v[c1])-get<Y>(v[c2]),2)));
}
```

- y por último la función *MostrarRecorrido* para imprimir el recorrido de cada estrategia :

```
void mostrarRecorrido(const vector<ciudad>& trayectoria, const vector<coordenadas>& ciudades,
                      unsigned long longitud) {
#if SHOWRECORRIDO
    for (auto c : trayectoria)
        cout << c+1 << " " << get<X>(ciudades[c]) << " " << get<Y>(ciudades[c]) << endl;
#endif
#if SHOWLONGITUD
    cout << "Longitud del recorrido: " << longitud << endl;
#endif
}

int main(int argc, char* argv[]){
    if (argc != 2) {
        cerr << "Poner como argumento nombre archivo";
        return -1;
    }
    auto   ciudades = leerArchivo(argv[1]);

    vector<ciudad> trayectoria;
#if HEURISTICA == 1
    auto distancias = calcularDistancias(ciudades);
    auto longitud = seleccionarMasCercanas(ciudades.size(), trayectoria,
                                           distancias);
#elif HEURISTICA == 2
    auto longitud = heuristicaInsercion(ciudades,calcularDistancias(ciudades),trayectoria);
#elif HEURISTICA == 3
    auto longitud = heuristica3(crearAristas(ciudades),trayectoria);
#endif

    mostrarRecorrido(trayectoria, ciudades, longitud);

}
```

2.2 Código del Algoritmo :

```
// Heurística Propuesta 1
distancia seleccionarMasCercanas (unsigned N, vector<ciudad>& trayectoria,
                                  map<ciudad, multimap<distancia, ciudad>>& distancias) {
    // Para trabajar trabajamos de 0 a N-1, los resultados se sacan de 1 a N con
    // las funciones creadas para ello
    unsigned long longitud = 0;

    // Empezamos en la ciudad 0
    int ciudad_actual = 0;
    trayectoria.reserve(N+1);
    trayectoria.push_back(ciudad_actual);

    while (trayectoria.size() < N) {
        // Miramos por orden creciente de distancia desde la ciudad actual
        auto it = distancias[ciudad_actual].begin();

        // Avanzamos en el caso de que ya hubiese sido introducida en trayectoria
        while (it != distancias[ciudad_actual].end() &&
               find(trayectoria.begin(), trayectoria.end(),
                    it->second) != trayectoria.end()
               ) ++it;
        // Introducimos la ciudad y añadimos la longitud del trayecto
        trayectoria.push_back(it->second);
        longitud += it->first;
        ciudad_actual = it->second;
    }

    // Añadimos la ciudad inicial para completar el ciclo
    trayectoria.push_back(0);
    return longitud;
}
```

2.3 - Ejemplos de demostración :

➔ *Ejemplo : Att48*

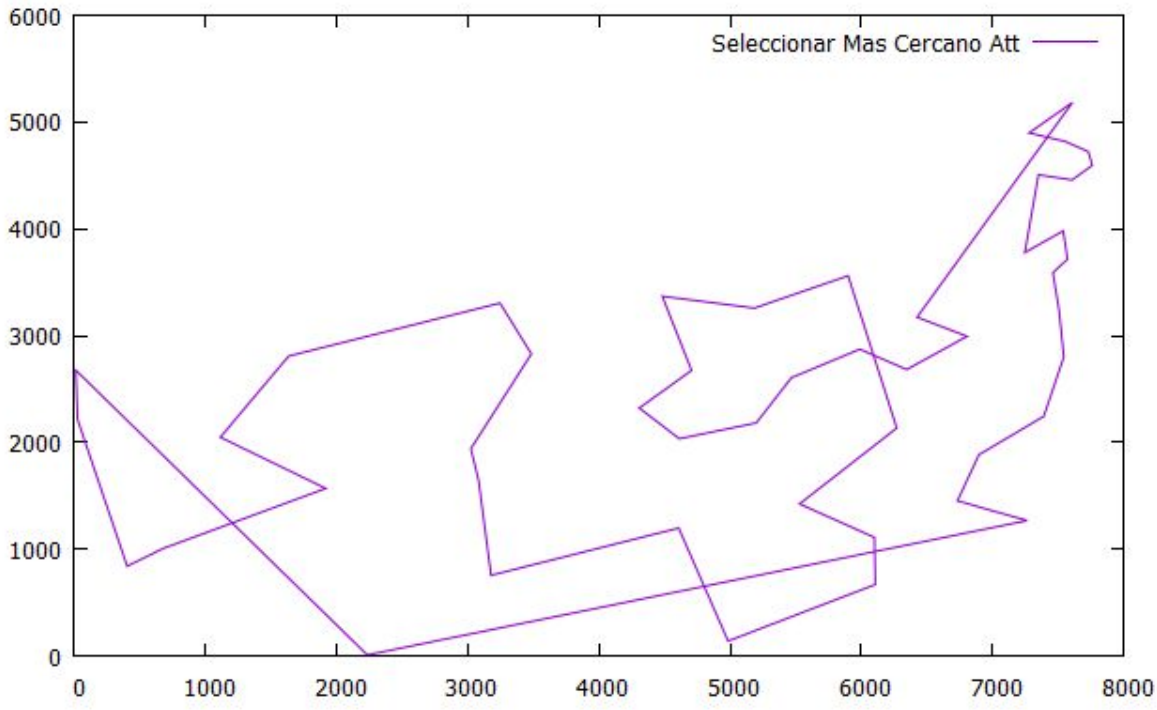

Figura 1 : demostración para la ciudad Att48

```
7 7573 3716
28 7541 3981
36 7248 3779
30 7352 4506
6 7608 4458
37 7762 4595
19 7732 4723
27 7555 4819
43 7280 4899
17 7611 5184
33 6426 3173
46 6807 2993
15 6347 2683
12 5989 2873
11 5468 2606
23 5199 2182
14 4612 2035
25 4307 2322
13 4706 2674
21 4483 3369
47 5185 3258
20 5900 3561
40 6271 2135
3 5530 1424
22 6101 1110
16 6107 669
41 4985 140
34 4608 1198
29 3177 756
5 3082 1644
48 3023 1942
39 3484 2829
32 3245 3305
24 1633 2809
10 1112 2049
42 1916 1569
26 675 1006
4 401 841
35 23 2216
45 10 2676
2 2233 10
8 7265 1268
1 6734 1453
Longitud del recorrido: 40021
```

➔ *Ejemplo : Bayg29*

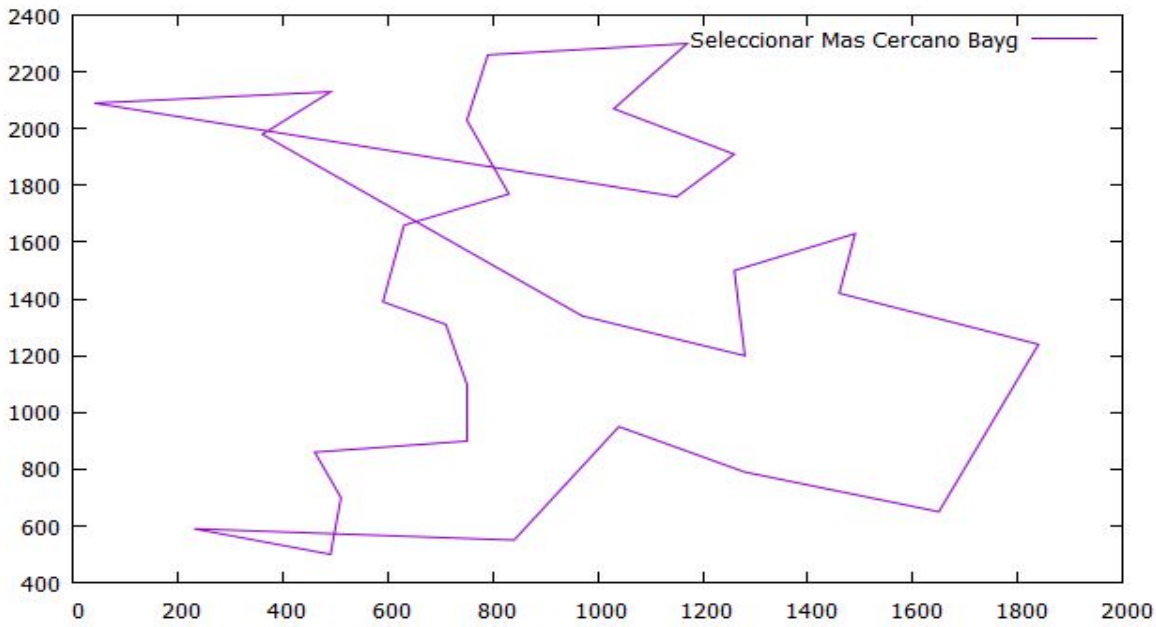

Figura 2 : demostración para la ciudad Bayg29

```
1 1150 1760
28 1260 1910
6 1030 2070
12 1170 2300
9 790 2260
5 750 2030
21 830 1770
2 630 1660
20 590 1390
10 710 1310
4 750 1100
15 750 900
18 460 860
14 510 700
22 490 500
17 230 590
11 840 550
19 1040 950
25 1280 790
7 1650 650
23 1840 1240
27 1460 1420
8 1490 1630
24 1260 1500
16 1280 1200
13 970 1340
29 360 1980
26 490 2130
3 40 2090
1 1150 1760
Longitud del recorrido: 9051
```

➔ *Ejemplo : Berlin52 :*

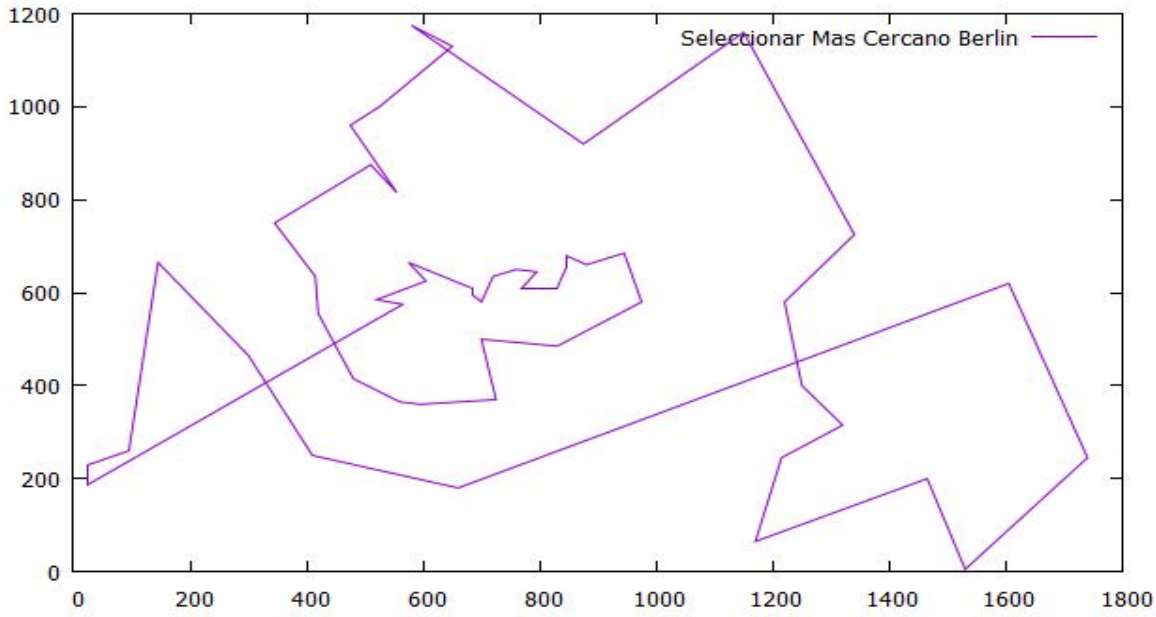

Figura 3 : demostración para la ciudad Berlin52

```
48 830 610
24 835 625
5 845 655
15 845 680
6 880 660
4 945 685
25 975 580
46 830 485
44 700 500
16 725 370
50 595 360
20 560 365
23 480 415
31 420 555
18 415 635
3 345 750
19 510 875
45 555 815
41 475 960
8 525 1000
10 650 1130
9 580 1175
43 875 920
33 1150 1160
51 1340 725
12 1220 580
28 1250 400
27 1320 315
26 1215 245
47 1170 65
13 1465 200
14 1530 5
52 1740 245
11 1605 620
29 660 180
30 410 250
21 300 465
17 145 665
42 95 260
7 25 230
2 25 185
1 565 575
Longitud del recorrido: 8314
```

3. Heurística 2 : Algoritmo de inserción

❖ Enfoque Greedy :

- Conjunto de candidatos: Ciudades a visitar
- Conjunto de seleccionados: Ciudades visitadas
- Función solución: Si hemos recorrido todas las ciudades
- Función de factibilidad: Que la ciudad no haya sido visitada

- Función de selección: Seleccionamos de entre todas las ciudades cual tendría mejor distancia,comprobandolo en todas las ubicaciones posibles.
- Función objetivo: Recorrer todas las ciudades y volver a la primera

La idea de esa heurística es crear un recorrido inicial, por ejemplo nos fijamos en la tres ciudades que forman el triángulo más grande (la ciudad que está más al norte, la más al este y la más al oeste) y a partir de este recorrido, vamos insertando las ciudades que faltan.

Empezamos recorriendo el vector de ciudades no seleccionadas e insertando cada una de ellas en cada posible posición, luego cogemos la de menor distancia y la añadimos.

3.1 Código del Algoritmo :

```
// Heurística Propuesta 2
vector<ciudad> seleccionarTriangulo(const vector<coordenadas>& loc) {
    // Buscamos la más al norte, la más al este y la mas al oste y devuelve un vector con las mismas;
    coordenadas norte, este, oeste;
    ciudad cnorte, ceste, coeste;
    norte = este = oeste = loc.front();
    for (unsigned city = 0; city < loc.size(); ++city) {
        if (get<Y>(loc[city]) > get<Y>(norte)) { norte = loc[city]; cnorte = city ;}
        if (get<X>(loc[city]) > get<X>( este)) {  este = loc[city]; ceste  = city ;}
        if (get<X>(loc[city]) < get<Y>(oeste)) { oeste = loc[city]; coeste = city ;}
    }
    vector<ciudad> salida;
    salida.push_back(cnorte);
    salida.push_back(ceste);
    salida.push_back(coeste);
    salida.push_back(cnorte);
    return salida;
}

distancia agregarCiudad(vector<ciudad> trayectoria, ciudad city, unsigned posicion,
                        const vector<vector<distancia>>& distancias) {
    // Trayectoria se pasa por copia puesto que no se quiere modificar la trayectoria inicial,
    // sólo calcular la supuesta distancia del camino.
    distancia dist = 0;
    trayectoria.insert(trayectoria.begin()+posicion,city);
    for (unsigned i = 1; i < trayectoria.size(); ++i)
        dist += distancias[trayectoria[i-1]][trayectoria[i]];
    return dist;
}
```

```
distancia heuristicaInsercion(const vector<coordenadas>& ciudades,
                              const vector<vector<distancia>>& distancias,
                              vector<ciudad>& trayectoria) {
    // Primero seleccionamos las ciudades más al norte, este y oeste
    trayectoria = seleccionarTriangulo(ciudades);

    // Recorremos las ciudades en orden numérico
    for (unsigned city = 0; city < ciudades.size(); ++city) {
        // Si no se encuentran en el camino todavía
        if (find(trayectoria.begin(),trayectoria.end(),city) == trayectoria.end()) {
            unsigned pos = 1;
            distancia dist_min = agregarCiudad(trayectoria, city, pos, distancias);
            // Comprobamos que ocurriría si la metiésemos entre cada una de las aristas válidas
            for (unsigned j = 2; j < trayectoria.size(); ++j) {
                distancia dist = agregarCiudad(trayectoria, city, j, distancias);
                if (dist < dist_min) { // Cogemos la de menor distancia
                    dist_min = dist;
                    pos = j;
                }
            }
            trayectoria.insert(trayectoria.begin()+pos,city); // y la añadimos
        }
    }

    distancia dist = 0;
    for (unsigned i = 1; i < trayectoria.size(); ++i)
        dist += distancias[trayectoria[i-1]][trayectoria[i]];
    return dist;
}
```

3.2 - Ejemplos de demostración :

➔ *Ejemplo : Att48*

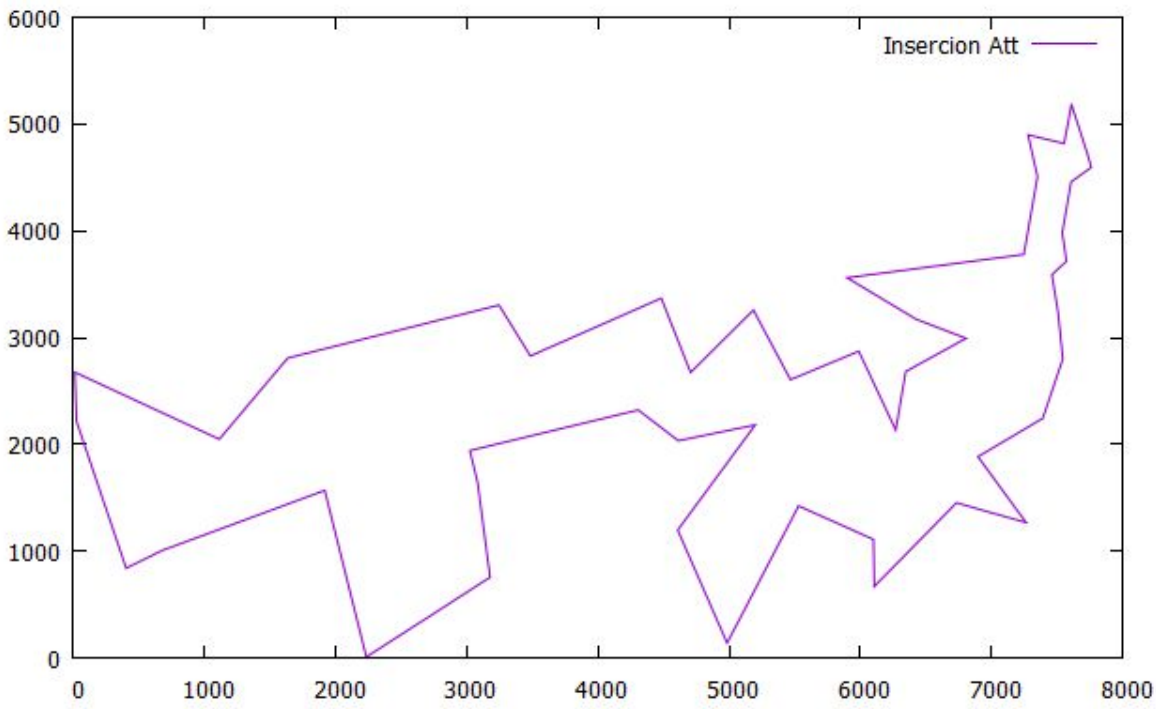

Figura 4 : demostración para la ciudad Att48

```
18 7462 3590
44 7509 3239
31 7545 2801
38 7392 2244
9 6898 1885
8 7265 1268
1 6734 1453
16 6107 669
22 6101 1110
3 5530 1424
41 4985 140
34 4608 1198
23 5199 2182
14 4612 2035
25 4307 2322
48 3023 1942
5 3082 1644
29 3177 756
2 2233 10
42 1916 1569
26 675 1006
4 401 841
35 23 2216
45 10 2676
10 1112 2049
24 1633 2809
32 3245 3305
39 3484 2829
21 4483 3369
13 4706 2674
47 5185 3258
11 5468 2606
12 5989 2873
40 6271 2135
15 6347 2683
46 6807 2993
33 6426 3173
20 5900 3561
36 7248 3779
30 7352 4506
43 7280 4899
27 7555 4819
17 7611 5184
Longitud del recorrido: 35133
```

➔ *Ejemplo : Bayg29*

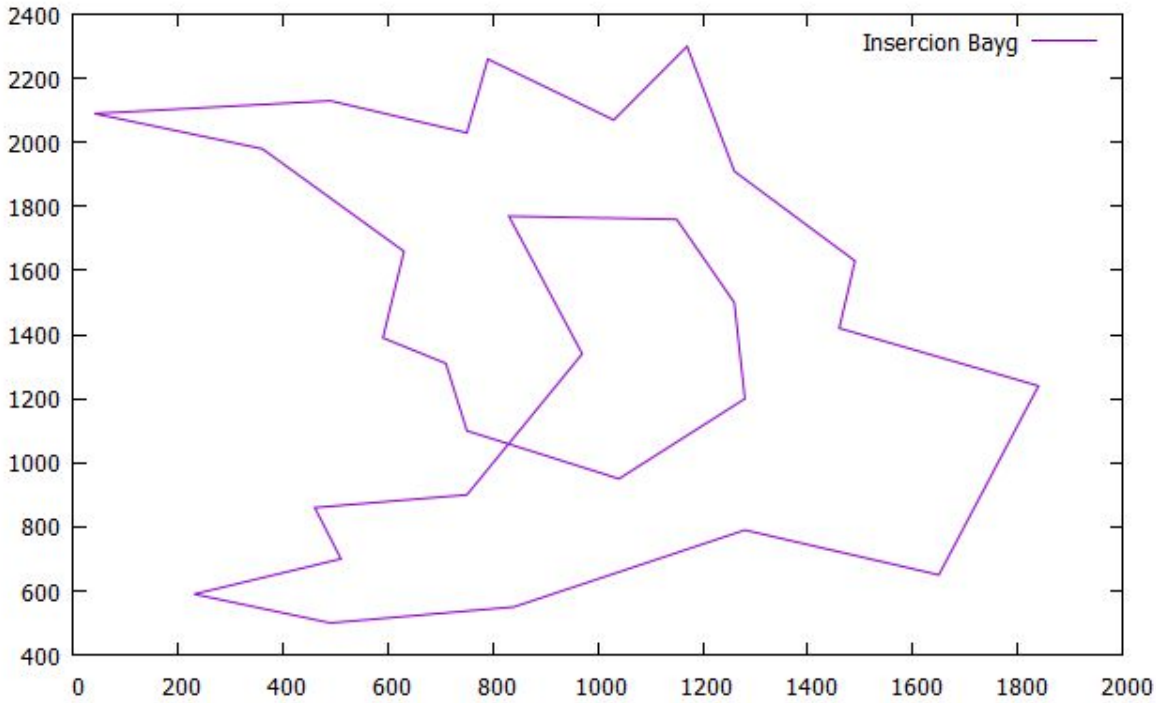

Figura 5 : demostración para la ciudad Bayg29

```
12 1170 2300
28 1260 1910
8 1490 1630
27 1460 1420
23 1840 1240
7 1650 650
25 1280 790
11 840 550
22 490 500
17 230 590
14 510 700
18 460 860
15 750 900
13 970 1340
21 830 1770
1 1150 1760
24 1260 1500
16 1280 1200
19 1040 950
4 750 1100
10 710 1310
20 590 1390
2 630 1660
29 360 1980
3 40 2090
26 490 2130
5 750 2030
9 790 2260
6 1030 2070
12 1170 2300
Longitud del recorrido: 9749
```

➔ *Ejemplo : Berlin52*

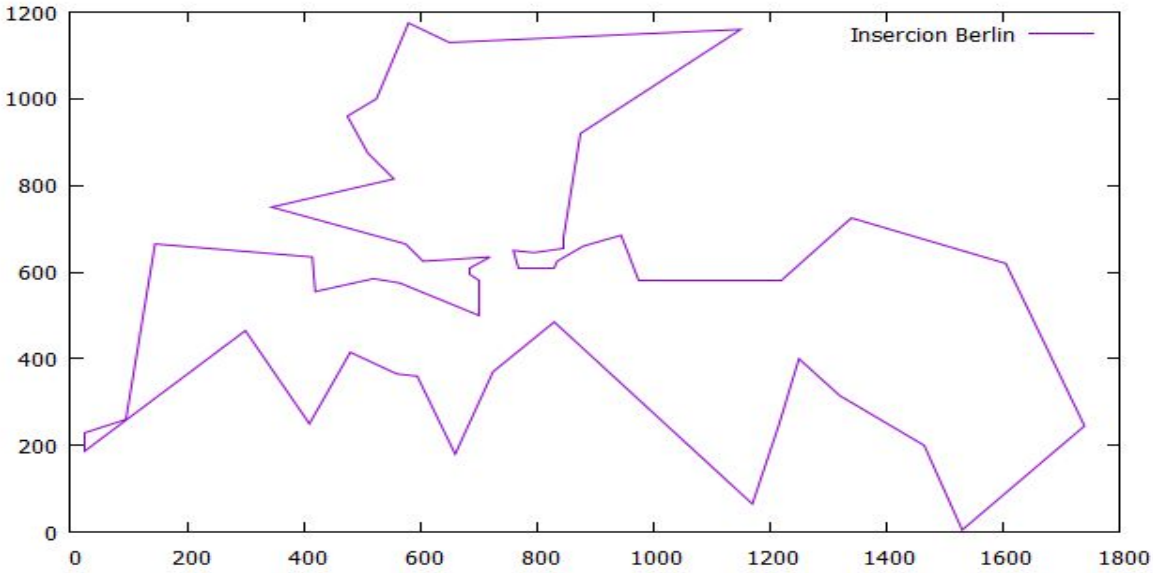

Figura 6 : demostración para la ciudad Berlin52

```
6 880 660
4 945 685
25 975 580
12 1220 580
51 1340 725
11 1605 620
52 1740 245
14 1530 5
13 1465 200
27 1320 315
28 1250 400
26 1215 245
47 1170 65
46 830 485
16 725 370
29 660 180
50 595 360
20 560 365
23 480 415
30 410 250
21 300 465
2 25 185
7 25 230
42 95 260
17 145 665
18 415 635
31 420 555
22 520 585
1 565 575
44 700 500
34 700 580
35 685 595
36 685 610
39 720 635
49 605 625
32 575 665
3 345 750
45 555 815
19 510 875
41 475 960
8 525 1000
9 580 1175
Longitud del recorrido: 8355
```

4. Heurística 3 : Algoritmo basado en aristas

❖ Enfoque Greedy :

● Conjunto de candidatos: Ciudades a visitar

- Conjunto de seleccionados: Ciudades visitadas
- Función solución: Si hemos recorrido todas las ciudades
- Función de factibilidad: Que la ciudad no tenga más de 2 arista y no se cree un ciclo al añadirla
- Función de selección: Seleccionamos las ciudades con menor distancia entre ellas.
- Función objetivo: Recorrer todas las ciudades y volver a la primera.

Para este algoritmo hemos creado un multimap (multimap<distancia, pair<ciudad, ciudad>> crearAristas).formado por la distancia entre dos ciudades cada entrada.

Vamos seleccionando las distancias menores entre dos ciudades y añadiendo estas al conjunto de seleccionados.

Para poder insertar una nueva ciudad, esta no puede tener más de 2 aristas ni crear un ciclo.

Cómo el camino resultante de este algoritmo no es cerrado, tenemos que recorrer el camino y añadir la distancia entre las 2 ciudades que únicamente tienen una arista.

4.1 Código del Algoritmo :

```
// Heurística 3
multimap<distancia, pair<ciudad, ciudad>> crearAristas(const vector<coordenadas>& ciudades) {
    // Crea un multimap con clave la longitud de cada una de las posibles aristas
    multimap<distancia, pair<ciudad, ciudad>> salida;
    for (unsigned i = 0; i < ciudades.size(); ++i)
        for (unsigned j = 0; j < ciudades.size(); ++j)
        {
            if (j > i)
                salida.insert({distanciaCiudad(i,j,ciudades), {i,j}});
        }

//    for (auto c: salida) {
//        cout << c.first << '\t' << c.second.first+1 << '\t' << c.second.second+1 << endl;
//    }
    return salida;
}

bool esFactible(const list<pair<ciudad, ciudad>>& aristas,
                const pair<ciudad,ciudad>& arista) {
    // Devuelve que es factible si no hay dos aristas que llegan a involucran a un vértice
    int contador1 = 0, contador2 = 0;
    for (auto a : aristas) {
        if (a.first == arista.first || a.second == arista.first)
            ++contador1;
        if (a.first == arista.second || a.second == arista.second)
            ++contador2;
    }
    return contador1 <= 1 && contador2 <= 1;
}
```

```
// Devuelve iterador a un par del vector de aristas que contega como primero o segundo el dato d
// Colocando como de el dato de la otra punta de la arista neuva
list<pair<ciudad,ciudad>>::const_iterator buscar_par(const list<pair<ciudad, ciudad>>& a, ciudad& d) {

    return find_if(a.begin(),a.end(),[&](const pair<ciudad, ciudad>& p) {
        if (p.first == d) {
            d = p.second;
            return true;
        }
        else if (p.second == d){
            d = p.first;
            return true;
        }
        else
            return false;
    });

}
```

```
bool hayCiclos(list<pair<ciudad, ciudad>> aristas,
               const pair<ciudad,ciudad>& arista) {
    // Devuelve si se produciría algún ciclo al añadir la arista arista
    auto buscar = [&] (const ciudad& c) {
        return find_if(aristas.begin(),aristas.end(),[&](const pair<ciudad, ciudad>& p) {
            return p.first == c || p.second == c;
        });
    };

    ciudad res=arista.second;
    ciudad pos=arista.first;
    for (auto it=buscar(pos); it !=aristas.end(); it=buscar(pos)) {
        pos = (pos == it->first) ? it->second : it->first;
        aristas.erase(it);
    }
    return pos==res;
}
```

```cpp
vector<ciudad> aristasATrayectoria(list<pair<ciudad, ciudad>>& aristas) {
    // Aunque aristas se borre se pasa por referencia por que no se va a usar más
    vector<ciudad> salida;

    auto c_it = aristas.cbegin();
    salida.push_back(aristas.begin()->first); // Introducimos la primera ciudad
    ciudad city = c_it->second; // Recordamos la segunda
    aristas.erase(c_it); // Eliminamos
    // Vaciamos el contenido de aristas repitiendo el proceso anterior
    while (!aristas.empty()) {
        salida.push_back(city);
        c_it = buscar_par(aristas, city);
        aristas.erase(c_it);
    }

    // Añadimos el ciudad inicial para cerrar el camino
    salida.push_back(salida.front());
    return salida;
}
```

```cpp
void cerrarCiclo(list<pair<ciudad, ciudad>>& aristas,
                 const multimap<distancia, pair<ciudad, ciudad>>& dist_aristas,
                  distancia& longitud) {
    int queda_cerrar1 = 0, queda_cerrar2 = 0;
    // Buscamos aquellos vértices que tengan grado menor que 2
    for (auto p : aristas) {
        int contador1 = count_if(aristas.begin(), aristas.end(),
                                 [&](const pair<ciudad, ciudad>&c) {
            return p.first == c.first || p.first == c.second;
        });
        int contador2 = count_if(aristas.begin(), aristas.end(),
                                 [&](const pair<ciudad, ciudad>&c) {
            return p.second == c.first || p.second == c.second;
        });

        // Determinamos las puntas de arista sin cerrar
        if (contador1 < 2 && queda_cerrar1 == 0) queda_cerrar1 = p.first;
        else if (contador1 < 2 && queda_cerrar2 == 0) queda_cerrar2 = p.first;
        if (contador2 < 2 && queda_cerrar1 == 0) queda_cerrar1 = p.second;
        else if (contador2 < 2 && queda_cerrar2 == 0) queda_cerrar2 = p.second;
    }
    // Añadimos la nueva arista y la distancia correspondiente
    for (auto p: dist_aristas) {
        auto f = p.second.first;
        auto s = p.second.second;
        if ((queda_cerrar1 == f && queda_cerrar2 ==s) || (queda_cerrar1 == s && queda_cerrar2 == f)) {
            aristas.push_back(p.second);
            longitud += p.first;
            break;
        }
    }
}
```

```cpp
distancia heuristica3(const multimap<distancia, pair<ciudad, ciudad>>& dist_aristas, vector<int>& trayectoria) {
    list<pair<ciudad, ciudad>> aristas;
    distancia longitud = 0;
    auto it = dist_aristas.begin();
    // Metemos la arista más corta
    aristas.push_back(it->second);
    longitud += it->first;
    it++;
    while (it!=dist_aristas.end()) { // Llendo de más corta a más larga comprobamos si podemos introducirla
        if (esFactible(aristas, it->second) && !hayCiclos(aristas, it->second)) {
            aristas.push_back(it->second);
            longitud += it->first;
        }
        it++;
    }
    cerrarCiclo(aristas,dist_aristas, longitud); // Cerramos el ciclo
    trayectoria = aristasATrayectoria(aristas); // Pasamos las aristas a un camino
    return longitud;
}
```

4.2 - Ejemplos de demostración :

➔ *Ejemplo : Att48*

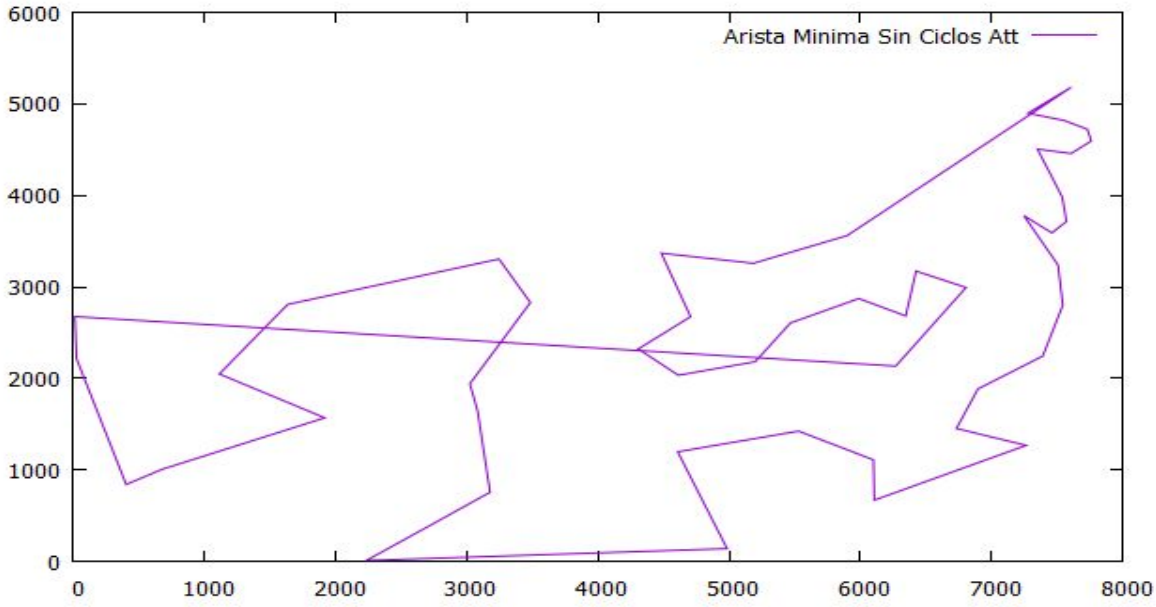

Figura 7 : demostración para la ciudad Att48

```
18 7462 3590
36 7248 3779
44 7509 3239
31 7545 2801
38 7392 2244
9 6898 1885
1 6734 1453
8 7265 1268
16 6107 669
22 6101 1110
3 5530 1424
34 4608 1198
41 4985 140
2 2233 10
29 3177 756
5 3082 1644
48 3023 1942
39 3484 2829
32 3245 3305
24 1633 2809
10 1112 2049
42 1916 1569
26 675 1006
4 401 841
35 23 2216
45 10 2676
40 6271 2135
46 6807 2993
33 6426 3173
15 6347 2683
12 5989 2873
11 5468 2606
23 5199 2182
14 4612 2035
25 4307 2322
13 4706 2674
21 4483 3369
47 5185 3258
20 5900 3561
17 7611 5184
43 7280 4899
27 7555 4819
19 7732 4723
Longitud del recorrido: 40159
```

➔ *Ejemplo : Bayg29*

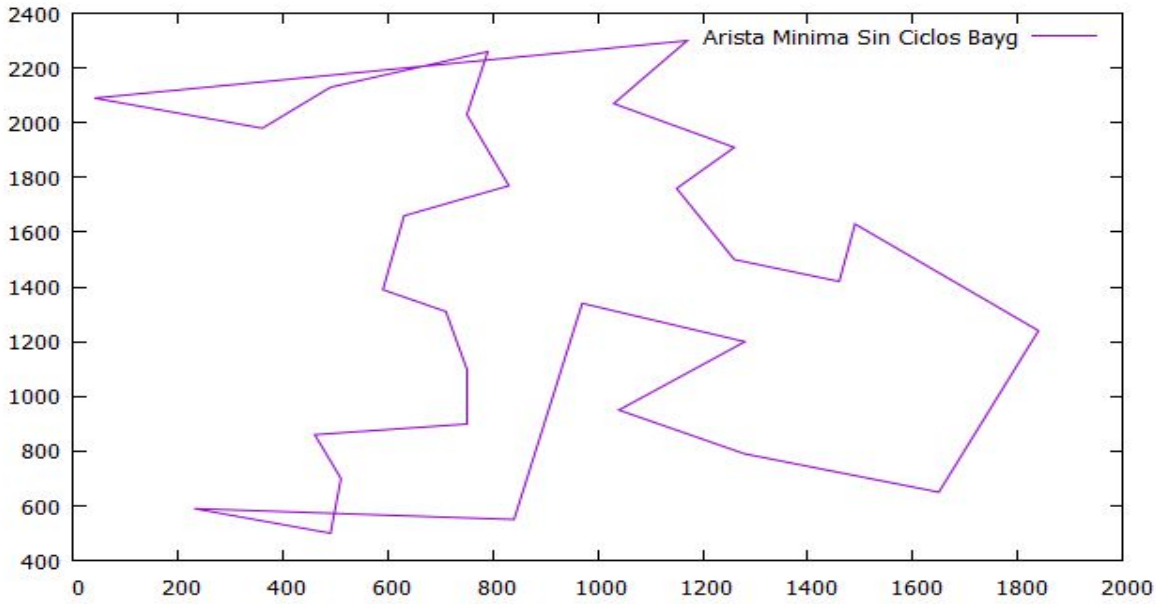

Figura 8 : demostración para la ciudad Bayg29

```
10 710 1310
20 590 1390
2 630 1660
21 830 1770
5 750 2030
9 790 2260
26 490 2130
29 360 1980
3 40 2090
12 1170 2300
6 1030 2070
28 1260 1910
1 1150 1760
24 1260 1500
27 1460 1420
8 1490 1630
23 1840 1240
7 1650 650
25 1280 790
19 1040 950
16 1280 1200
13 970 1340
11 840 550
17 230 590
22 490 500
14 510 700
18 460 860
15 750 900
4 750 1100
10 710 1310
Longitud del recorrido: 9884
```

➔ *Ejemplo : Berlin52*

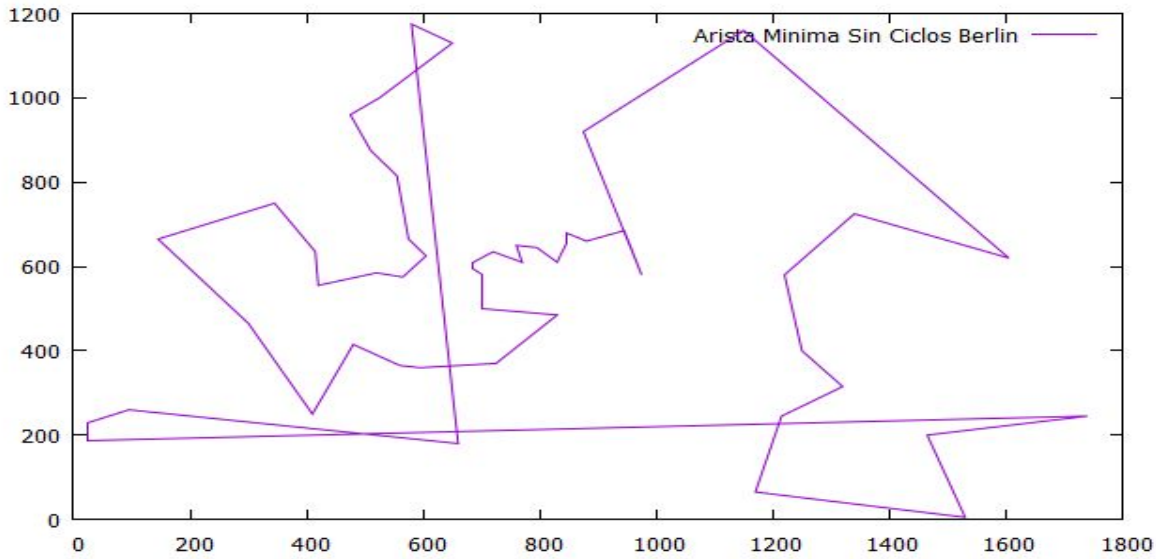

Figura 9 : demostración para la ciudad Berlin52

```
4 945 685
25 975 580
43 875 920
33 1150 1160
11 1605 620
51 1340 725
12 1220 580
28 1250 400
27 1320 315
26 1215 245
47 1170 65
14 1530 5
13 1465 200
52 1740 245
2 25 185
7 25 230
42 95 260
29 660 180
9 580 1175
10 650 1130
8 525 1000
41 475 960
19 510 875
45 555 815
32 575 665
49 605 625
1 565 575
22 520 585
31 420 555
18 415 635
3 345 750
17 145 665
21 300 465
30 410 250
23 480 415
20 560 365
50 595 360
16 725 370
46 830 485
44 700 500
34 700 580
35 685 595
Longitud del recorrido: 9951
```