Práctica 3. Parte 2: EL PROBLEMA DEL VIAJANTE DE COMERCIO

Álvaro Fernández García
Alexandra-Diana Dumitru
Ana María Romero Delgado
Germán Castro Montes
Marino Fajardo Torres

Índice

Descripción del problema

El ejercicio propuesto es el siguiente:

> Dado un conjunto de ciudades y una matriz con todas 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 sea mínima. Más formalmente, dado un grafo G, conexo y ponderado, se trata de hallar el ciclo hamiltoniano de mínimo peso de dicho grafo.

Por tanto se deben realizar las siguientes tareas:

- Diseñar los algoritmos greedy mencionados arriba para intentar obtener un ciclo hamiltoniano de peso míno del grafo G.
- Demostrar que los algoritmos son correctos, o poner contraejemplos.

Algoritmo greedy basado en la técnica de los vecinos más cercanos

La idea principal es: dada una ciudad v_0, se elige como ciudad siguiente aquella v_i (no incluida en en ciclo) que se encuentre más cercana a v_0. El procedimiento se repite hasta que se hayan visitado todas las ciudades.

El algorimto cuenta con los siguientes elementos:

- **Conjunto de candidatos (C):** ciudades del mapa, representadas por un identificador y sus coordenadas.
- **Conjunto de seleccionados (S):** todas las ciudades del mapa ordenadas según el recorrido planteado.
- **Función solucion:** el conjunto de candidatos (C) se encuentra vácio, es decir, se han visitado todas las ciudades y se ha vuelto a la ciudad de partida.
- **Función selección (FS):** selecciona la ciudad más cercana a la ciudad actual.
- **Función de factibilidad (FF):** la ciudad elegida no figura en el conjunto de seleccionados (*Prescindible en este caso por propia implementación del algoritmo*).
- **Función objetivo:** elegir el ciclo hamiltoniano de mínimo peso.

En definitiva, el funcionamiento básico del algoritmo consiste en ir seleccionando del conjunto de candidatos aquella ciudade con menor distancia de la ciudad actual y repetir el proceso hasta que no queden ciudades por

visitar.

A continuación se desarrolla, en pseudocódigo, el algorítmo necesario para resolver el problema.

Algoritmo 1 Viajante de comercio: Vecino más cercano

Entrada: Conjunto de candidatos y Matriz de Distancias (MD)
Salida: Conjunto Solución y distancia

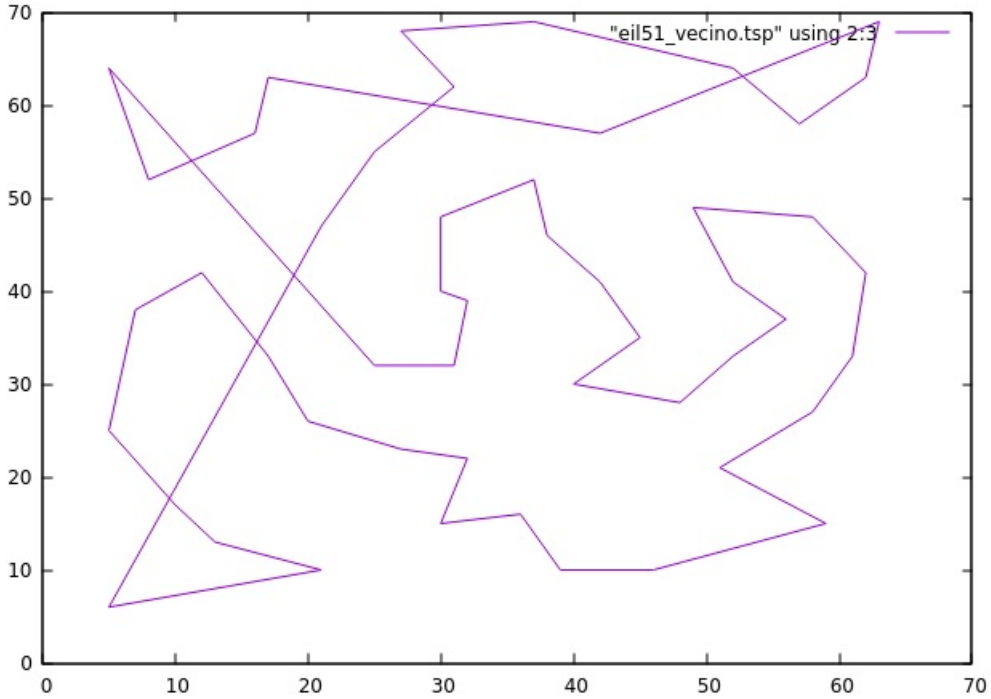
```
1: mejorCoste ← ∞
2: para todo candidato hacer
3:    candidatosTmp ← candidatos
4:    ciuadadActual ← siguienteCiudadInicio()
5:    borrarCandidatoTmp(ciudadActual)
6:    solucionTmp.Añadir(ciudadActual)
7:    mientras candidatosTmp no vacío hacer
8:       ciudadSiguiente ← VecinoMasCercano(ciudadActual, MD)
9:       solucionTmp.Añadir(ciudadSiguiente)
10:      ciudadActual ← ciudadSiguiente
11:      borrarCandidatoTmp(ciudadSiguiente)
12:   fin mientras
13:   si coste(solucionTmp, MD) < mejorCoste entonces
14:      mejorSolucion ← solucionTmp
15:      mejorCoste ← coste(solucionTmp, MD)
16:   fin si
17: fin para
18: devolver mejorSolucion, mejorCoste
```

Además, podemos ver el código del algoritmo que, como todos los implementados, cuenta con un método auxiliar para calcular la distancia euclídea entre dos puntos y otro para calcular la matriz de distancias entre las ciudades.

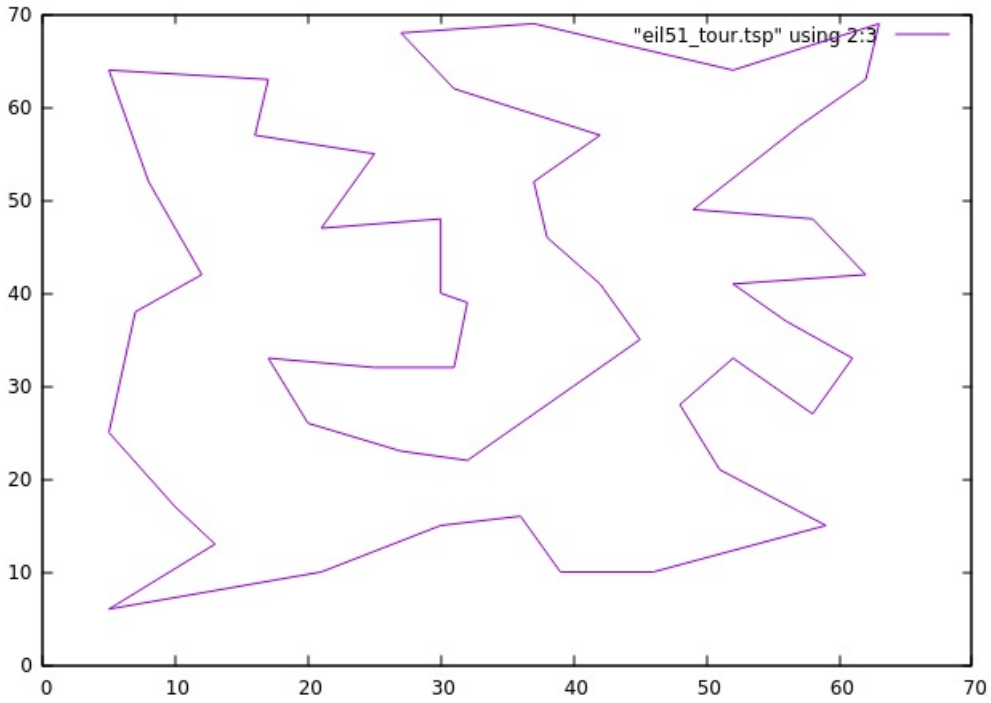
```
//Función para calcular la distancia euclídea entre dos puntos:
double Distancia(pair<double,double> p1, pair<double,double> p2){
    return sqrt(pow(p2.first-p1.first,2) + pow(p2.second-p1.second,2));
}


//Función para calcular la matriz de distancias:
void CalculaMD(double** M, int n, map<int,pair<double,double>> & mapa){
    for(int i = 1; i<n; ++i)
        for(int j = 1; j<n; ++j)
            M[i][j] = Distancia(mapa[i], mapa[j]);
}


//Método para calcular un recorrido para el viajante de comercio basado en el vecino más cercano:
pair<vector<int>,double> ViajanteDeComercio(vector<int> & candidatos, double** MD){

    //Variables para el calculo de la distancia;
    vector<int> S, Stemp, candidatos_tmp;
```

```
    double MejorDist = -1.0, distTemp, min_dist;
    int c_actual, c_siguiente;

    //Calcular la distancia partiendo de distintas ciudades:
    for(int i = 0; i < candidatos.size(); ++i){

        distTemp = 0.0;
        candidatos_tmp = candidatos;
        c_actual = candidatos_tmp[i];
        Stemp.push_back(c_actual); //Añadir a la lista de seleccionados.
        candidatos_tmp.erase(candidatos_tmp.begin()+i); //Borrar de entre los candidatos la ciuda
d.

        while(!candidatos_tmp.empty()){

            //Función Selección. El vecino más cercano:
            min_dist = MD[c_actual][candidatos_tmp[0]];
            c_siguiente = 0;
            for(int j = 1; j < candidatos_tmp.size(); ++j)
                if(MD[c_actual][candidatos_tmp[j]] < min_dist){
                    c_siguiente = j;
                    min_dist = MD[c_actual][candidatos_tmp[j]];
                }
            //Las ciudades elegidas son factibles, ya que las ya elegidas no figuran en candidato
s.
            distTemp += min_dist;
            Stemp.push_back(candidatos_tmp[c_siguiente]);
            c_actual = candidatos_tmp[c_siguiente];
            candidatos_tmp.erase(candidatos_tmp.begin()+c_siguiente);
        }

        //Añadir a distTemp la distancia entre la primera y la última ciudad:
        distTemp += MD[Stemp.front()][Stemp.back()];

        //Comprobar si es mejor que la que ya había:
        if(MejorDist == -1.0){
            MejorDist = distTemp;
            S = Stemp;
        }
        else if(distTemp < MejorDist){
            MejorDist = distTemp;
            S = Stemp;
        }
        Stemp.clear(); // Vaciar la solución temporal.
    }

    return pair<vector<int>,double>(S, MejorDist);
}
```

A continuación se muestran un par de ejemplos ilustrados del algoritmo.



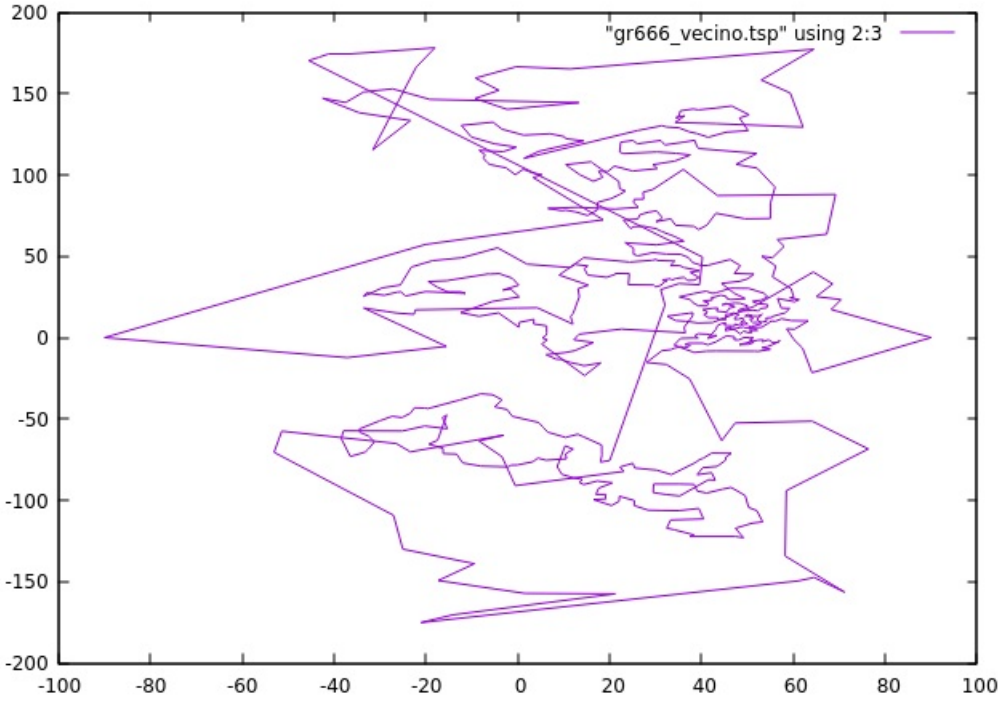
En la gráfica anterior se puede ver la solución encontrada para un mapa con 51 ciudades.



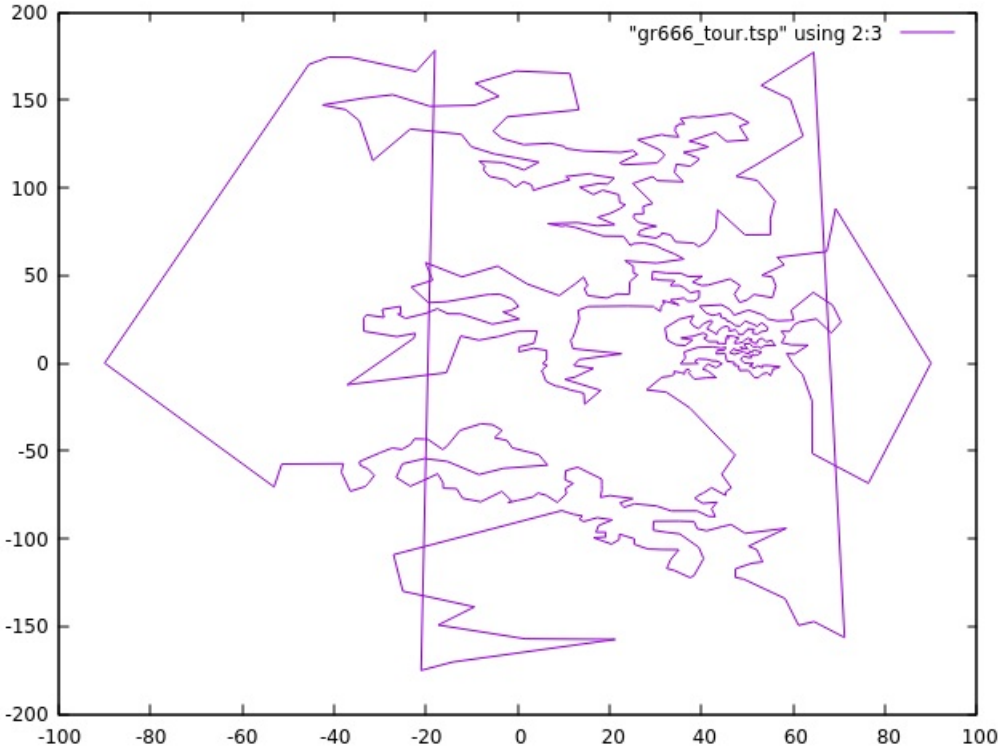
Sin embargo, la segunda ilustración es un claro contraejemplo que demuestra que nuestro algoritmo no devuelve la solución óptima.

De esta manera, cabe ver un ejemplo más que nos ayude ver mejor este hecho.

La siguiente gráfica muestra la solución encontrada por nuestro algoritmo para un mapa con 666 ciudades.



En definitiva, la solución encontrada no es la más óptima ya que sabemos que hay una solución mejor.



Algoritmo greedy basado en la técnica de inserción

La idea principal de este algoritmo es partir de un recorrido inicial que incluya solo algunas de las ciudades, y luego, extender dicho ciclo insertando las ciudades restantes mediante algún criterio de tipo greedy. Se debe tener en cuenta lo siguiente:

> i. Cómo se construye el recorrido parcial inicial.
> ii. Cuál es el nodo siguiente a insertar en el recorrido parcial.
> iii. Dónde se inserta el nodo seleccionado.

El algoritmo cuenta con los siguientes elementos:

- **Conjunto de candidatos (C):** ciudades del mapa, representadas por un identificador y sus coordenadas.
- **Conjunto de seleccionados (S):** todas las ciudades del mapa ordenadas según el recorrido planteado.
- **Función solucion:** Todas las ciudades del conjunto de candidatos (C) aparecen como seleccionadas.
- **Función selección (FS):** selecciona la ciudad, y la posición en la que la vamos a insertar, que supone la inserción más económica, es decir, provoca el menor incremento en la longitud total del circuito.
- **Función de factibilidad (FF):** la ciudad elegida para insertar no figura en el conjunto de seleccionados (*Prescindible en este caso por propia implementación del algoritmo*).
- **Función objetivo:** elegir el ciclo hamiltoniano de mínimo peso.

Resumiendo, se parte de una solución parcial inicial (que en nuestro caso es un cuadrado formado por las ciudades más al norte, sur, este y oeste) y a continuación se procede a ir insertando cada una de las ciudades entre las ciudades ya presentes en la solución de tal forma que en cada paso, insertemos la mejor ciudad, en la mejor posición para que el coste total del recorrido se incremente lo menos posible.

A continuación se desarrolla, en pseudocódigo, el algorítmo necesario para resolver el problema.

Algoritmo 2 Viajante de comercio: Método de inserción

Entrada: Conjunto de candidatos, Matriz de Distancias (MD) y Coordenadas
Salida: Conjunto Solución y distancia

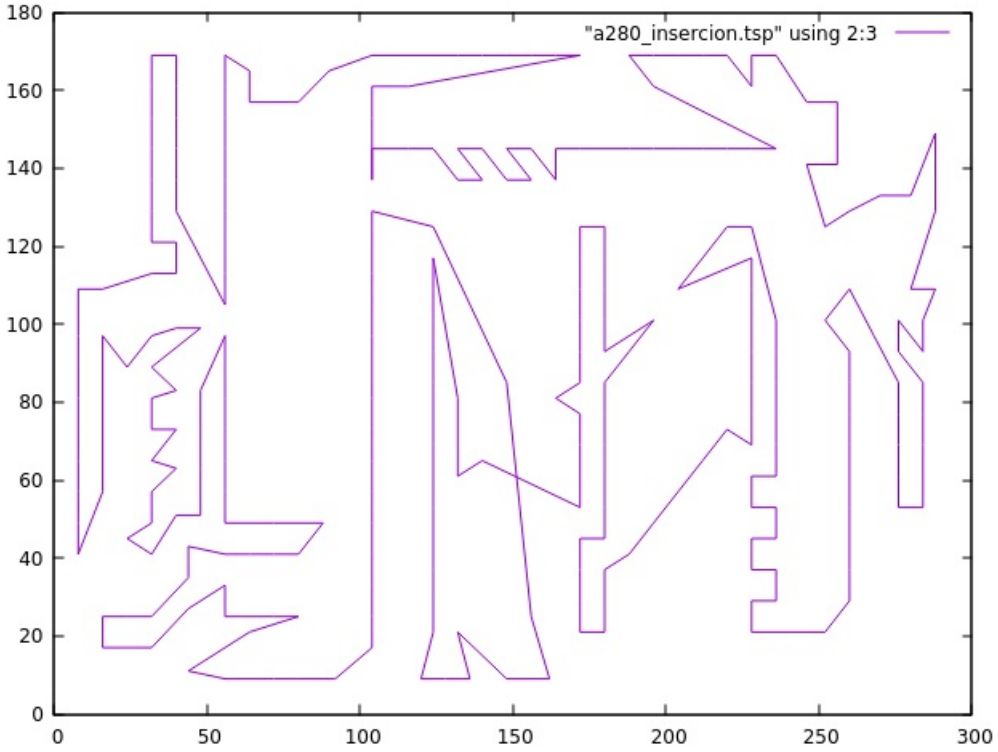
```
1:  solucion ← CuadradoInicial(Coordenadas)
2:  distancia ← coste(solucion, MD)
3:  para i = 0 hasta candidatos.size()−4 hacer
4:     mejorCoste ← ∞
5:     para todo candidato no seleccionado hacer
6:        ciudad ← siguienteCandidato()
7:        para todo posicion posible en solucion hacer
8:           posicion ← {siguiente posicion en la que insertar ciudad}
9:           costeActual ← {coste tras insertar ciudad entre posicion y posi-
             cion+1}
10:          si costeActual < mejorCoste entonces
11:             mejorCoste ← costeActual
12:             mejorCiudad ← ciudad
13:             mejorPosicion ← posicion + 1
14:          fin si
15:       fin para
16:    fin para
17:    solucion.insertar(mejorPosicion, mejorCiudad)
18:    mejorCiudad.seleccionada ← cierto
19:    distancia ← mejorCoste
20: fin para
21: devolver solucion, distancia
```

Además, podemos ver el código del algoritmo que, como todos los implementados, cuenta con un método auxiliar para calcular la distancia euclídea entre dos puntos y otro para calcular la matriz de distancias entre las ciudades.

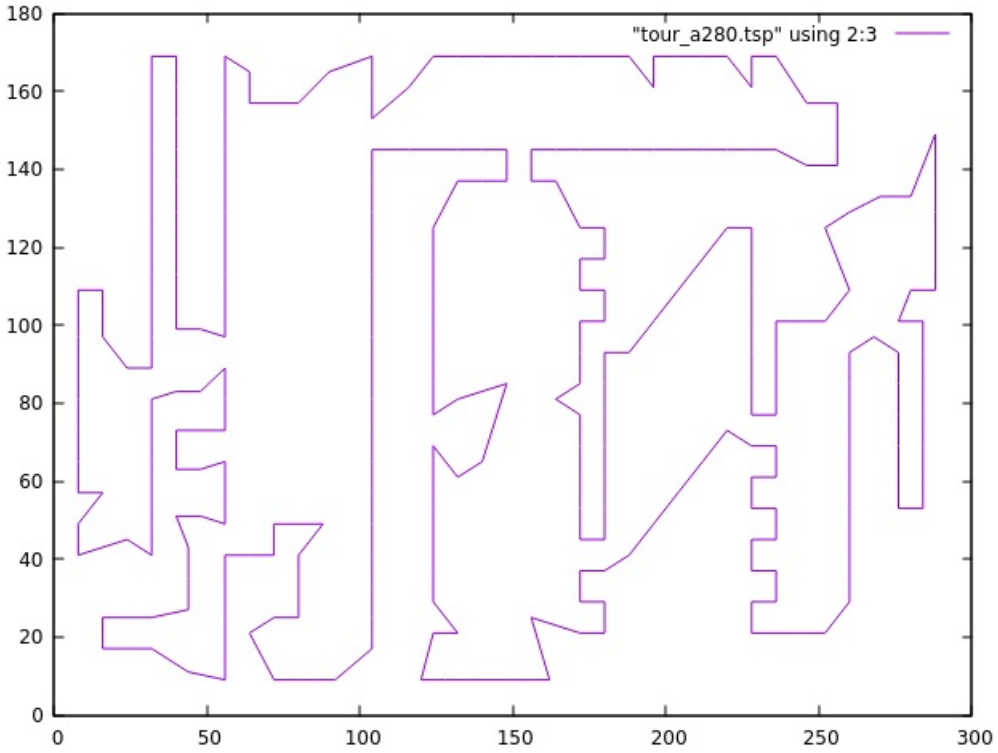
```
//Función para calcular la distancia euclídea entre dos puntos:
double Distancia(pair<double,double> p1, pair<double,double> p2){
    return sqrt(pow(p2.first-p1.first,2) + pow(p2.second-p1.second,2));
}


//Función para calcular la matriz de distancias:
void CalculaMD(double** M, int n, map<int,pair<double,double>> & mapa){
    for(int i = 1; i<n; ++i)
        for(int j = 1; j<n; ++j)
            M[i][j] = Distancia(mapa[i], mapa[j]);
}


//Método para calcular un recorrido para el viajante de comercio basado en la inserción:
pair<vector<int>,double> ViajanteDeComercio(vector<int> & candidatos, double** MD,  map<int,pair<
double,double>> puntos){
  vector<int> Sol;
  double distancia = 0;

  ////////////////////////////////////////////////////////////////////////
  //Partiremos de un cuadrado inicial:
  ////////////////////////////////////////////////////////////////////////
  double N = 0, S = 1.84467e+19, E = 0, O = 1.84467e+19; //Coordenadas
  int cn = 0, cs = 0, ce = 0, co = 0; //Índice ciudad.
  vector<bool> seleccionados(candidatos.size()+1, false);

  for(auto it = puntos.begin(); it != puntos.end(); ++it){
    if((it->second).second > N && !seleccionados[it->first]){
      N = (it->second).second;
      seleccionados[cn] = false;
      cn = it->first;
      seleccionados[cn] = true;
    }
    if((it->second).second < S && !seleccionados[it->first]){
      S = (it->second).second;
      seleccionados[cs] = false;
      cs = it->first;
      seleccionados[cs] = true;
    }
    if((it->second).first > E && !seleccionados[it->first]){
      E = (it->second).first;
      seleccionados[ce] = false;
      ce = it->first;
      seleccionados[ce] = true;
    }
    if((it->second).first < O && !seleccionados[it->first]){
      O = (it->second).first;
      seleccionados[co] = false;
      co = it->first;
      seleccionados[co] = true;
    }
  }
```

```
  //Añadir las cuatro ciudades elegidas:
  Sol.push_back(cn);
  Sol.push_back(ce);
  Sol.push_back(cs);
  Sol.push_back(co);

  //Calcular la distancia que llevamos hasta ahora:
  distancia = MD[cn][ce] + MD[ce][cs] + MD[cs][co] + MD[co][cn];
  ////////////////////////////////////////////////////////////////////////
  double mejor_coste, distancia_aux;
  int mejor_ciudad;
  vector<int>::iterator mejor_pos;

  //Habrá que repetir el proceso las siguientes veces:
  for(unsigned int i = 0; i < candidatos.size()-4; ++i){
    mejor_coste = 1.84467e+19;
    //Para cada una de las ciudades en candidatos:
    for(unsigned int j = 0; j < candidatos.size(); ++j){
      //Si no he insertado la ciudad:
      if(!seleccionados[candidatos[j]]){
        for(auto it = Sol.begin(); it != Sol.end(); ++it){
          if((it+1) == Sol.end())
            distancia_aux = distancia - MD[Sol.front()][Sol.back()] + MD[Sol.front()][candidatos[
j]] + MD[Sol.back()][candidatos[j]];
          else
            distancia_aux = distancia - MD[*it][*(it+1)] + MD[*it][candidatos[j]] + MD[*(it+1)][c
andidatos[j]];
          //Si es mejor que lo que tenía actualizo:
          if(distancia_aux < mejor_coste){
            mejor_coste = distancia_aux;
            mejor_ciudad = candidatos[j];
            mejor_pos = it+1;
          }
        }
      }
    }
    //Aplicar los cambios:
    Sol.insert(mejor_pos, mejor_ciudad);
    seleccionados[mejor_ciudad] = true;
    distancia = mejor_coste;
  }

  return pair<vector<int>,double>(Sol, distancia);
}
```

A continuación se mostraran un par ilustraciones de soluciones encontradas por nuestro algoritmo para dos mapas diferentes así como las soluciones óptimas de dichos mapas.

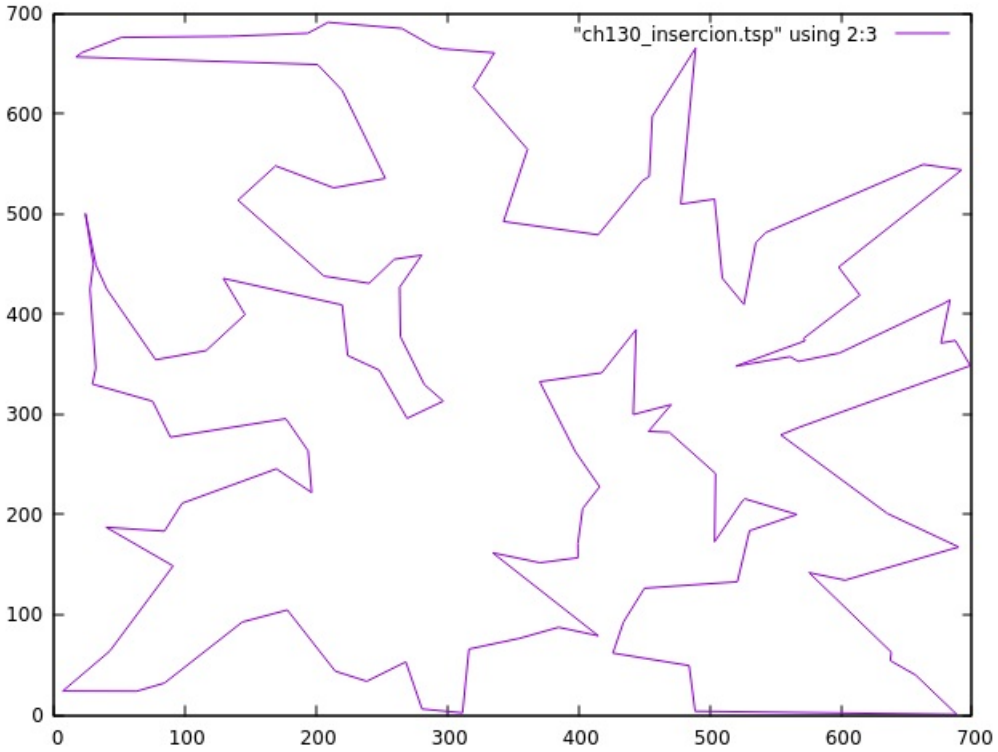


La gráfica anterior muestra la solución encontrada por el algoritmo basado en la técnica de inserción para un mapa con 280 ciudades.

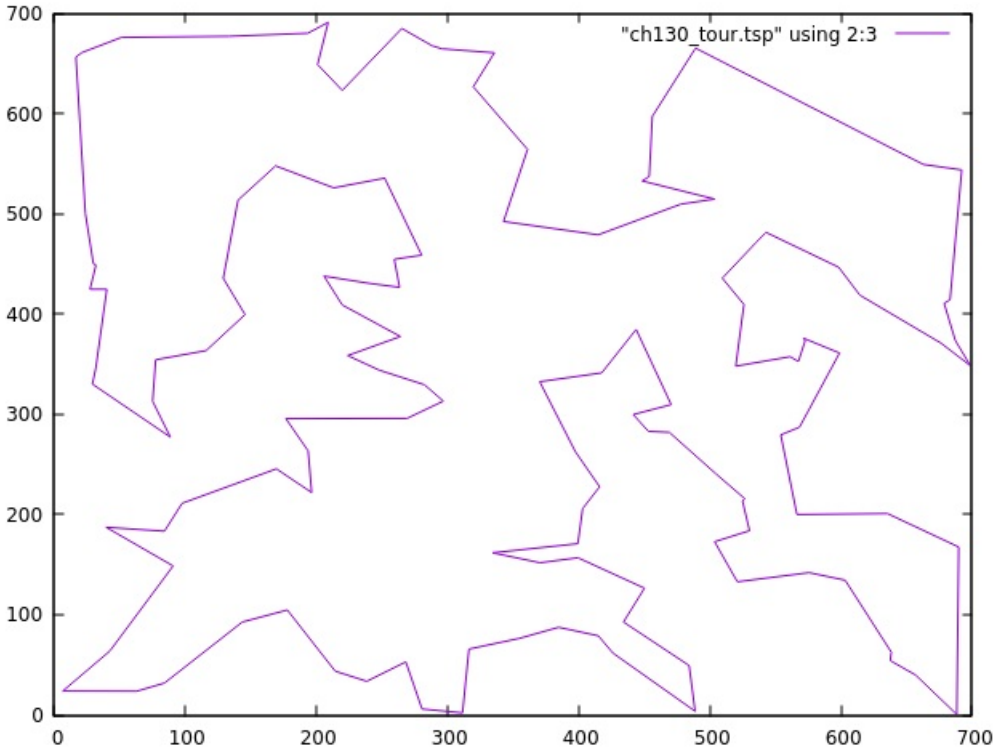


No obstante, el contraejemplo de la segunda ilustración demuestra que la solución encontrada por nuestro algoritmo no es la más óptima.

El segundo ejemplo que vamos a ver es un mapa que consta de 130 ciudades. La siguiente gráfica muestra la solución encontrada por nuestro algoritmo para dicho mapa.



En conclusión, el algoritmo basado en la técnica de inserción tampoco devuelve la solución optima, aunque, tal como se puede ver en la siguiente ilustración, se acerca más a ésta que el algoritmo basado en la técnica de los vecinos cercanos.



Algoritmo greedy basado en la técnica de intercambio 2-opt

Se trata de una heurística de mejora, y por lo tanto, se ha de partir de una solución factible ya conocida (*en nuestro caso, la que nos proporcione el algoritmo del vecino más cercano*). Se basa en la sustitución de k arcos de una ruta por otros k arcos de la misma. Se dice que un tour es k-optimo (k-opt) si es imposible obtener una ruta con menor distancia (o coste) remplazando k de sus arcos por otros k arcos distintos. En este caso se ha considerado un valor de k=2 de ahí que sea 2-opt.

La heurística 2-opt está basada en el hecho de que para un problema Euclídeo, si dos aristas se cruzan, pueden ser sustituidas por otras dos que no se crucen, mejorando así el problema.

El algoritmo cuenta con los siguientes elementos:

- **Conjunto de candidatos (C):** aquellas ciudades del mapa presentes en una solución inicial S.
- **Conjunto de seleccionados (S):** todas las ciudades del mapa ordenadas según el recorrido planteado.
- **Función solucion:** no es posible encontrar otro 2-optimo que mejore el coste actual de la solución.
- **Función selección (FS):** mejor par de ciudades cuyo intercambio reduce el coste actual de la solución.
- **Función de factibilidad (FF):** *no es necesaria para este algoritmo, ya que se parte de una solución factible y solo se intercambia el orden de las ciudades*.
- **Función objetivo:** elegir el ciclo hamiltoniano de mínimo peso.

En definitiva, se trata de ir buscado el el mejor movimiento 2-opt que incluya la arista de una ciudad `i` y su sucesor en el ciclo, de tal forma que se reduzca el coste actual de la solución y repetir este proceso mientras seamos capaces de encontrar intercambios que produzcan una mejora en la longitud total del circuito Hamiltoniano.

A continuación se desarrolla, en pseudocódigo, el algorítmo necesario para resolver el problema.

Algoritmo 3 Viajante de comercio: Intercambio 2-opt

Entrada: Conjunto de candidatos y Matriz de Distancias (MD)
Salida: Conjunto Solución y distancia

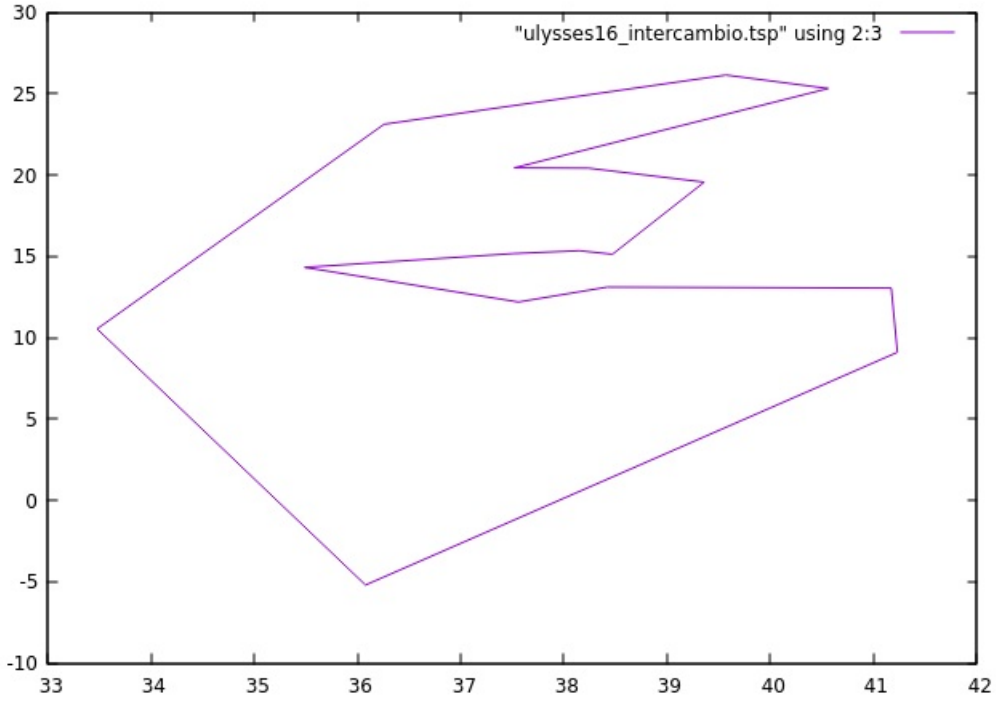
```
1: solucion ← solucionInicial(candidatos, MD)
2: mejorCoste ← coste(solucion, MD)
3: cambioRealizado ← cierto
4: mientras cambioRealizado hacer
5:    cambioRealizado ← falso
6:    para i = 0 hasta solucion.size()−1 hacer
7:       cambioEncontrado ← falso
8:       para j = i + 1 hasta solucion.size() hacer
9:          coste ← {coste tras intercambiar solucion[i] y solucion[j]}
10:         si coste < mejorCoste entonces
11:            mejorCoste ← coste
12:            posicion ← j
13:            cambioEncontrado ← cierto
14:         fin si
15:      fin para
16:      si cambioEncontrado entonces
17:         solucion ← Intercambiar(solucion[i], solucion[posicion])
18:         cambioRealizado ← cierto
19:      fin si
20:   fin para
21: fin mientras
22: devolver solucion, mejorCoste
```

A continuación podemos ver el código del algoritmo que, como todos los implementados, cuenta con un método auxiliar para calcular la distancia euclídea entre dos puntos y otro para calcular la matriz de distancias entre las ciudades. Además este algoritmo también requiere de un método extra que calcula el coste total de una solución y como es lógico, el algoritmo del vecino más cercano para la solución inicial.

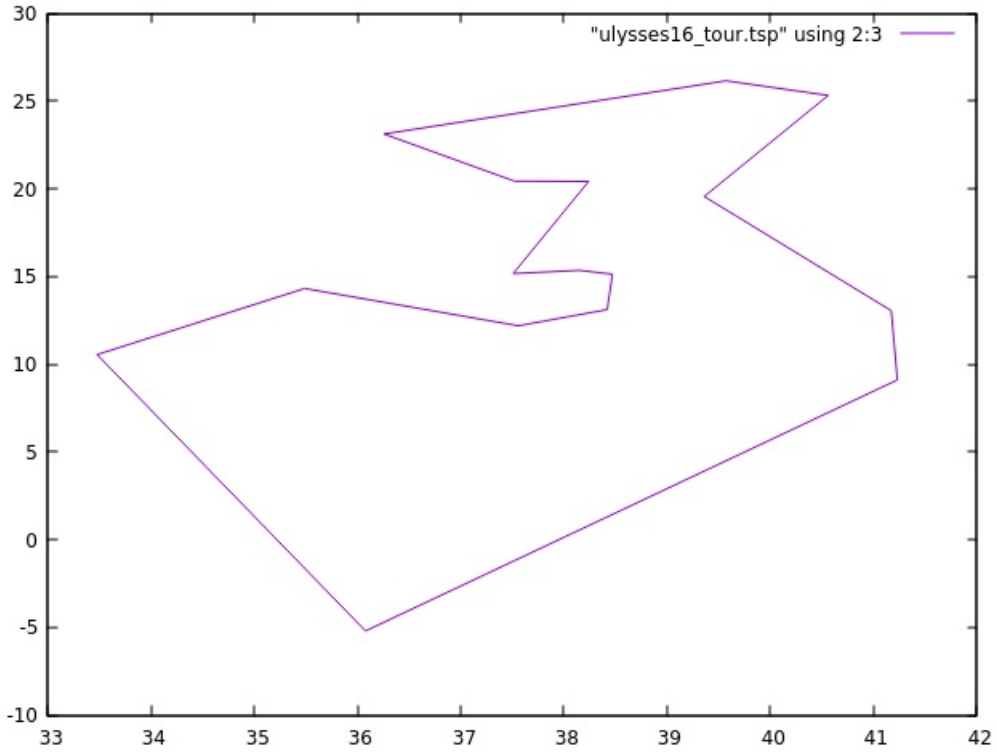
```
//Función para calcular la distancia euclídea entre dos puntos:
double Distancia(pair<double,double> p1, pair<double,double> p2){
    return sqrt(pow(p2.first-p1.first,2) + pow(p2.second-p1.second,2));
}


//Función para calcular la matriz de distancias:
void CalculaMD(double** M, int n, map<int,pair<double,double>> & mapa){
    for(int i = 1; i<n; ++i)
        for(int j = 1; j<n; ++j)
            M[i][j] = Distancia(mapa[i], mapa[j]);
}

//Método para calcular el coste de una solución:
double coste(vector<int> Solucion, double** MD){
  double coste = 0;
  for(unsigned int i = 0; i < Solucion.size()-1; ++i)
    coste += MD[Solucion[i]][Solucion[i+1]];

  coste += MD[Solucion.front()][Solucion.back()];
```

```
  return coste;
}

//Método para calcular un recorrido para el viajante de comercio basado en el vecino más cercano:
pair<vector<int>,double> VecinoMasCercano(vector<int> & candidatos, double** MD){
    //Mismo código que el algoritmo anterior...
}

//Método para calcular un recorrido para el viajante de comercio basado en el intercambio de aris
tas 2-opt:
pair<vector<int>,double> ViajanteDeComercioIntercambio(vector<int> & candidatos, double** MD){

  int aux;
  unsigned int pos;
  vector<int> S_aux, S;
  double coste_aux;
  bool mejor_enc, cambio_realizado = true;

  //Considerar una solución inicial:
  //En este caso utilizaremos el método del vecino más cercano:
  auto SolInicio = VecinoMasCercano(candidatos, MD);
  S = SolInicio.first;
  double mejor_coste = SolInicio.second;

  while(cambio_realizado){
    cambio_realizado = false;
    for(unsigned int i = 0; i < S.size()-1; ++i){
      mejor_enc = false;
      //Examinar los movimientos 2-opt que incluyan la arista de i a su sucesor en el ciclo:
      for(unsigned int j = i + 1; j < S.size(); ++j){
        S_aux = S;
        //Intercambiar
        aux = S_aux[i];
        S_aux[i] = S_aux[j];
        S_aux[j] = aux;
        //Calcular el coste de la nueva solución:
        coste_aux = coste(S_aux, MD);
        //Si es mejor guardo:
        if(coste_aux < mejor_coste){
          mejor_enc = true;
          pos = j;
          mejor_coste = coste_aux;
        }
      }
      //Si se ha encontrado algo mejor, intercamio:
      if(mejor_enc){
        cambio_realizado = true;
        aux = S[i];
        S[i] = S[pos];
        S[pos] = aux;
      }
    }
  }
  return pair<vector<int>, double>(S, mejor_coste);
}
```

A continuación se mostraran un par ilustraciones de soluciones encontradas por nuestro algoritmo para dos mapas diferentes así como las soluciones óptimas de dichos mapas.

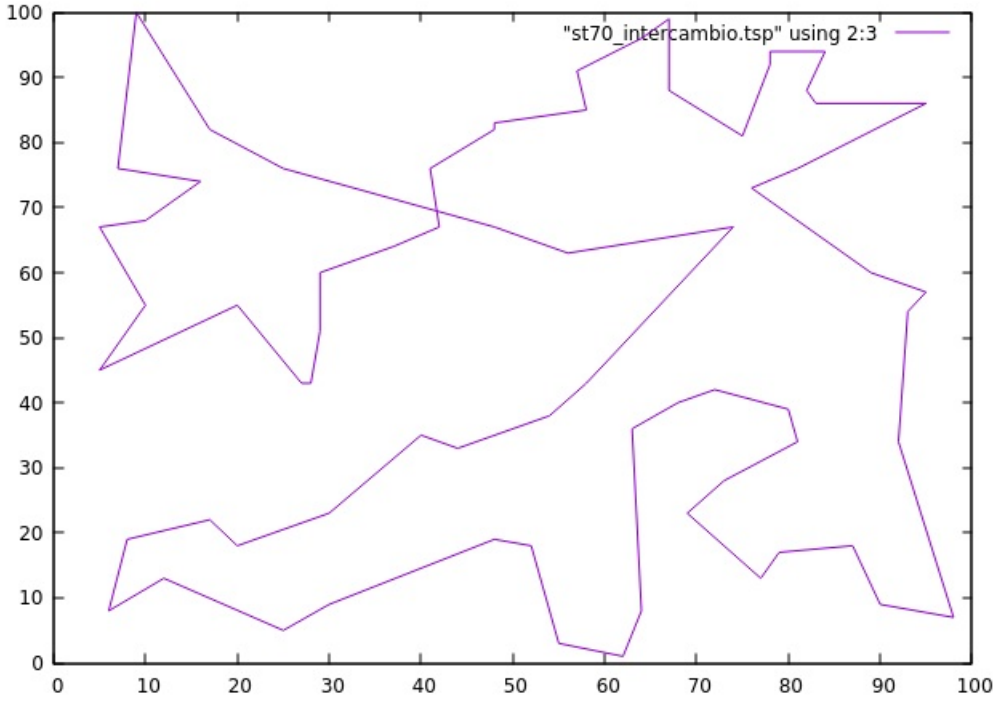


La gráfica anterior muestra la solución encontrada por el algoritmo basado en la técnica de intercambio 2-opt para un mapa con 16 ciudades.

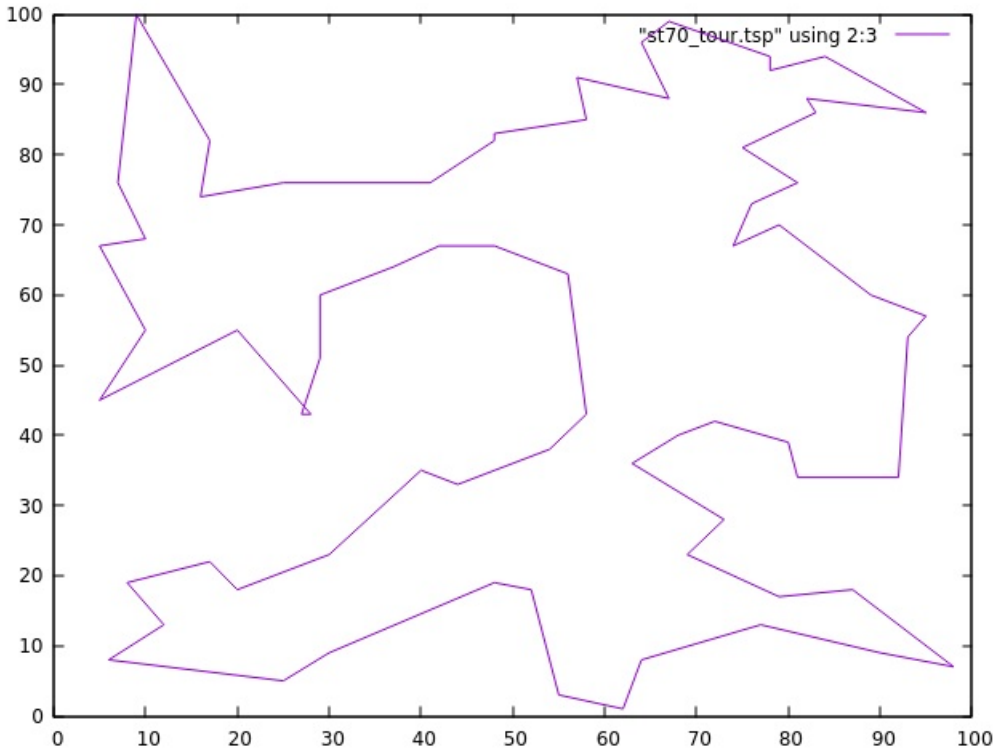


No obstante, el contraejemplo de la segunda ilustración demuestra que la solución encontrada por nuestro algoritmo no es la más óptima.

El segundo ejemplo que vamos a ver es un mapa que consta de 70 ciudades. La siguiente gráfica muestra la solución encontrada por nuestro algoritmo para dicho mapa.



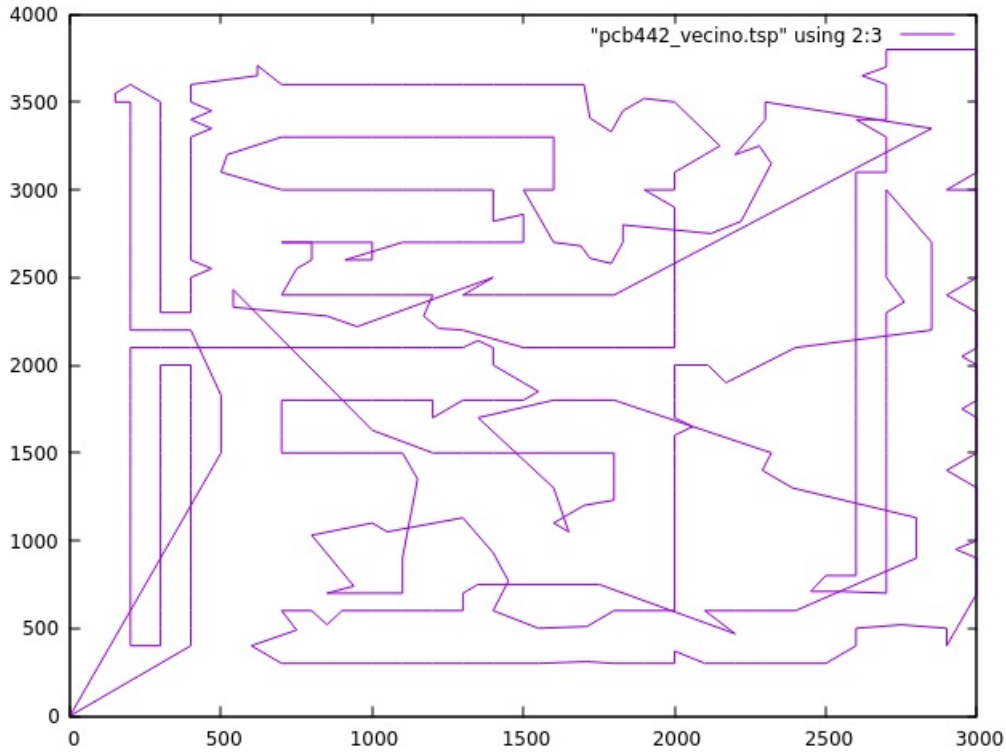
No obstante la solución óptima es la que sigue:



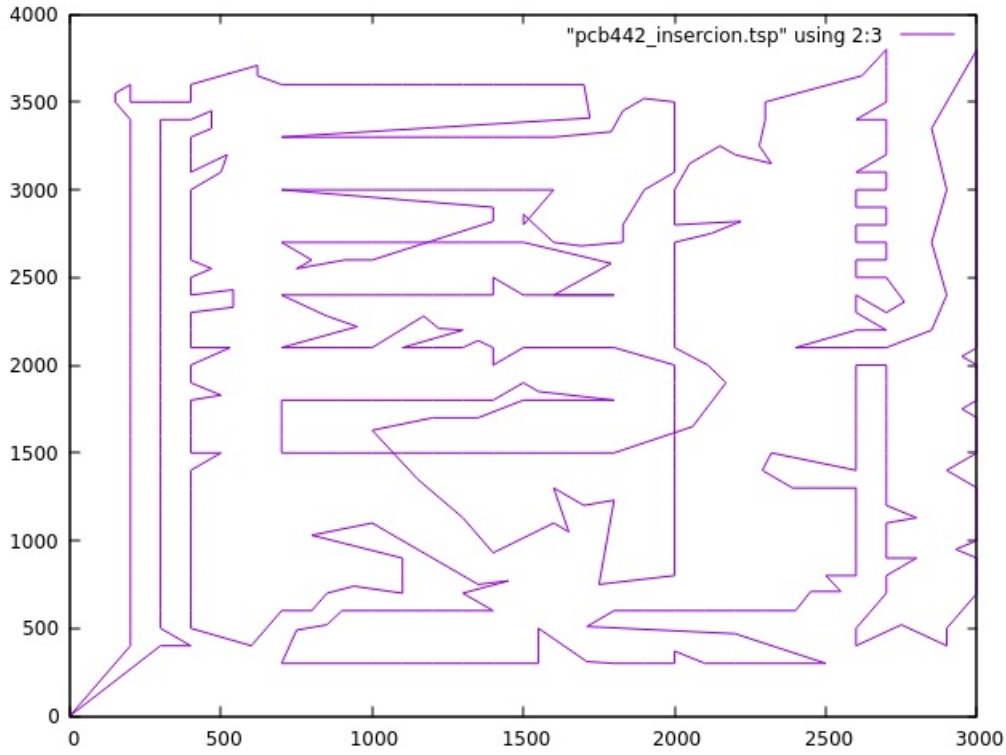
Estudio comparativo entre las tres estrategias

En primer lugar, vamos a ver las soluciones halladas por los tres algoritmos para un mapa que consta de 442 ciudades, así como la solución óptima para dicho mapa.

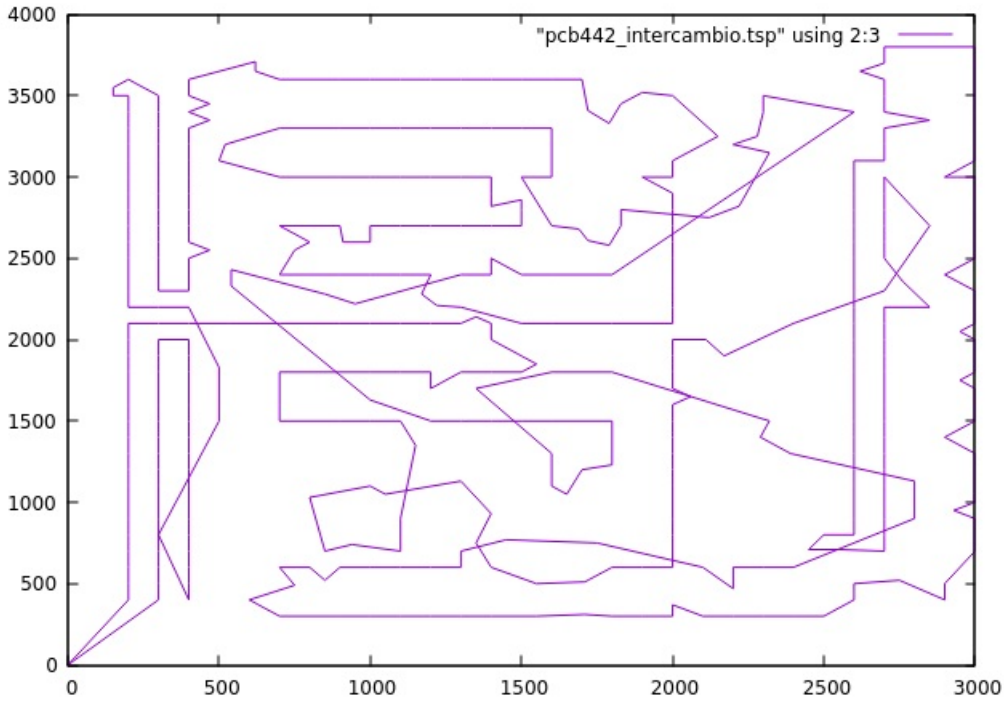
Solución hallada por el algoritmo basado en la técnica de los vecinos cercanos.



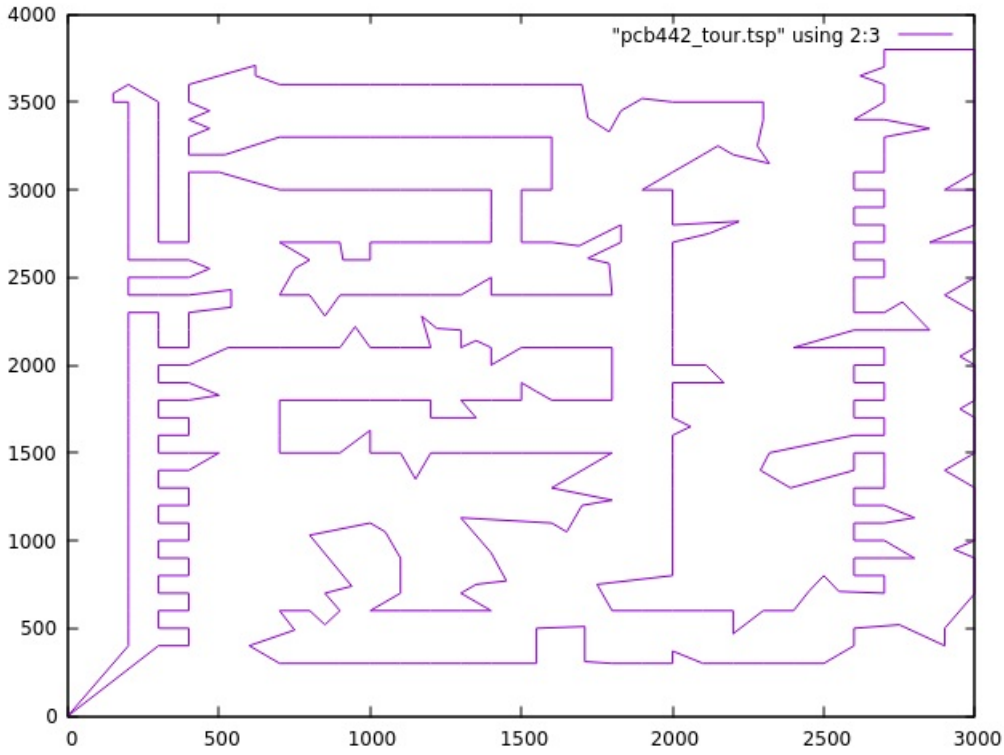
Solución hallada por el algoritmo basado en la técnica de la inserción



Solución hallada por el algoritmo basado en la técnica del intercambio entre dos ciudades



Solución óptima del mapa de 442 ciudades



Como se puede ver en las gráficas anteriores, ninguno de nuestros algoritmos devuelve la solución óptima. No obstante, el que más se acerca a la mejor solución es el algoritmo basado en la técnica de inserción.

Además, para finalizar, se presenta una tabla en la que se compara el porcentaje de error (*es decir, lo diferente*

que es el mapa obtenido con respecto a la solución óptima), para cada uno de los algoritmos:

Mapas	Inserción	Intercambio	Vecino
ulysses16	1,55661	3,10552	4,0726
st70	5,80821	7,05109	12,2446
pcb442	12,0496	13,6876	16,0867
eil51	8,76286	13,0160	17,6263
ch130	8,9924	13,5545	17,8023
a280	15,9464	16,3863	19,6194
Media Error %	**8,85268**	**11,13350**	**14,57532**