Algorítmica

Grado en Ingeniería Informática

Práctica 3

El viajante de comercio

Autores

María Jesús López Salmerón

Nazaret Román Guerrero

Laura Hernández Muñoz

José Baena Cobos

Carlos Sánchez Páez

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Curso 2017-2018

Índice

1. Descripción de la práctica **1**

2. Vecino más cercano **1**

3. Inserción más económica **1**

4. Derivado de Kruskal **1**

5. Comparación de estrategias **2**

6. Anexo: código fuente **9**

Índice de figuras

1. Comparación de resultados. 2
2. *ulysses16.tsp* 3
3. *gr96.tsp* 3
4. *a280.tsp* 4
5. *att48.tsp* 4
6. *pa561.tsp* 5
7. *lin105.tsp* 5
8. *pr76.tsp* 6
9. *tsp225.tsp* 6
10. *ch130.tsp* 7
11. *berlin52.tsp* 7
12. *rd100.tsp* 8
13. *pr2392.tsp* 8
14. Programa que calcula el orden según las distintas heurísticas 14
15. Programa que calcula la distancia del circuito a partir de un fichero con coordenadas y una lista ordenada de ciudades. 18

1. Descripción de la práctica

El objetivo de esta práctica es abarcar el problema del viajante de comercio (TSP, *Travel Salesman Problem*) mediante estrategias voraces. En concreto, seguiremos tres heurísticas diferentes:

1. **Vecino más cercano**.
2. **Inserción más económica**.
3. **Derivado de Kruskal**.

2. Vecino más cercano

- **Conjunto de candidatos**. Ciudades a visitar.
- **Conjunto de seleccionados**. Aquellas ciudades que vayamos incorporando al circuito.
- **Función solución**. Todas las ciudades han sido visitadas y hemos vuelto a la primera.
- **Función de factibilidad**. La ciudad no ha sido visitada aún.
- **Función de selección**. Seleccionaremos aquella ciudad cuya distancia euclídea sea menor con respecto a la última ciudad añadida al conjunto de seleccionados.

Para abarcar más posibilidades, ejecutaremos el algoritmo voraz teniendo en cuenta todas las posibles ciudades de inicio, quedándonos con la distancia total más pequeña.

3. Inserción más económica

- **Conjunto de candidatos**. Ciudades por visitar.
- **Conjunto de seleccionados**. Inicialmente está formador por el triángulo de las ciudades más al norte, este y oeste.
- **Función solución**. Todas las ciudades han sido visitadas y hemos vuelto a la primera.
- **Función de factibilidad**. La ciudad no ha sido visitada aún.
- **Función de selección**. Seleccionamos la ciudad que incrementa minimamente la distancia total del circuito.

4. Derivado de Kruskal

- **Conjunto de candidatos**.
- **Conjunto de seleccionados**.
- **Función solución**.
- **Función de factibilidad**.
- **Función de selección**.

5. Comparación de estrategias

ulysses16.tsp	
Vecino más cercano	77.1269
Inserción más económica	
Derivado de Kruskal	
Solución más óptima	74

gr96.tsp	
Vecino más cercano	603.302
Inserción más económica	
Derivado de Kruskal	
Solución más óptima	512

a280.tsp	
Vecino más cercano	3094.28
Inserción más económica	
Derivado de Kruskal	
Solución más óptima	2586

att48.tsp	
Vecino más cercano	39236.9
Inserción más económica	
Derivado de Kruskal	
Solución más óptima	33523

pa561.tsp	
Vecino más cercano	18347
Inserción más económica	
Derivado de Kruskal	
Solución más óptima	19330

lin105.tsp	
Vecino más cercano	16939.4
Inserción más económica	
Derivado de Kruskal	
Solución más óptima	14382

pr76.tsp	
Vecino más cercano	130921
Inserción más económica	
Derivado de Kruskal	
Solución más óptima	108159

tsp225.tsp	
Vecino más cercano	4633.2
Inserción más económica	
Derivado de Kruskal	
Solución más óptima	14806

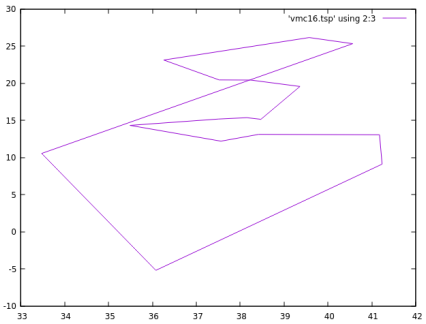
ch130.tsp	
Vecino más cercano	7198.74
Inserción más económica	
Derivado de Kruskal	
Solución más óptima	6110

berlin52.tsp	
Vecino más cercano	8182.19
Inserción más económica	
Derivado de Kruskal	
Solución más óptima	7544

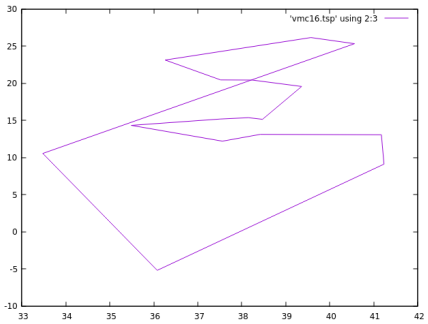
rd100.tsp	
Vecino más cercano	9427.33
Inserción más económica	
Derivado de Kruskal	
Solución más óptima	9724

pr2392.tsp	
Vecino más cercano	458790
Inserción más económica	
Derivado de Kruskal	
Solución más óptima	50560

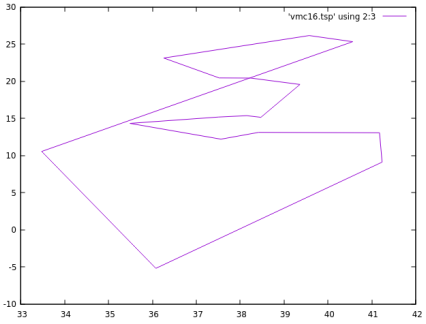
Figura 1: Comparación de resultados.



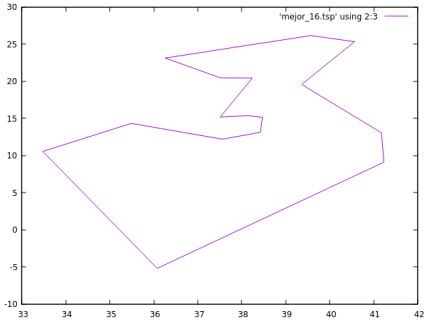
(a) Vecino más cercano



(b) Inserción más económica

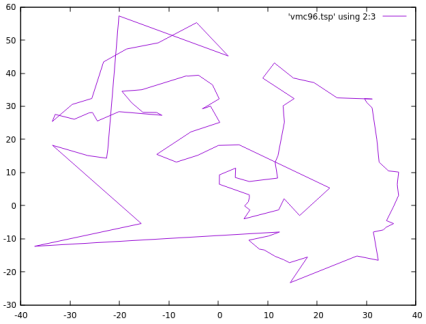


(c) Derivado de Kruskal

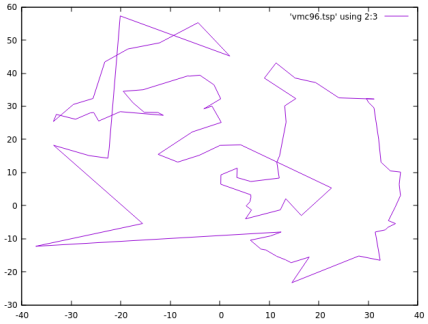


(d) Versión óptima

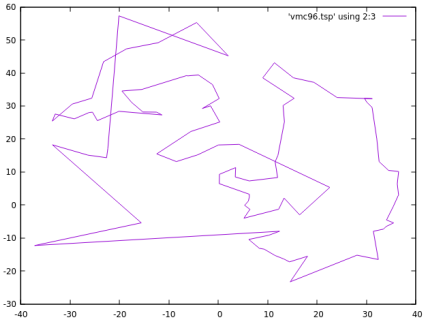
Figura 2: *ulysses16.tsp*



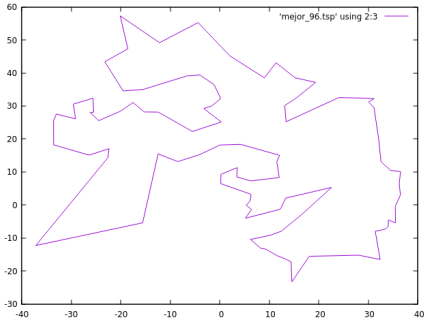
(a) Vecino más cercano



(b) Inserción más económica

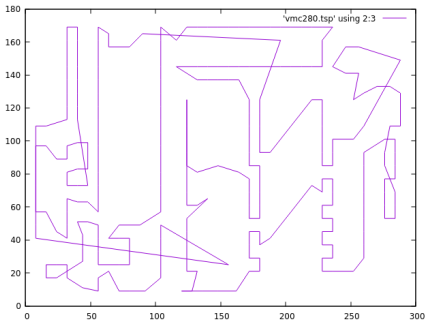


(c) Derivado de Kruskal

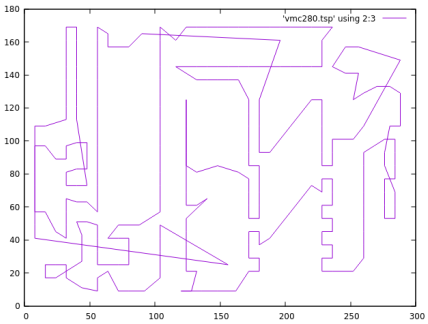


(d) Versión óptima

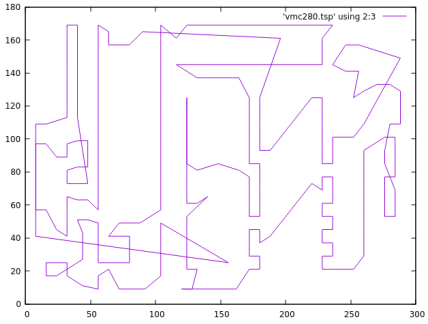
Figura 3: *gr96.tsp*



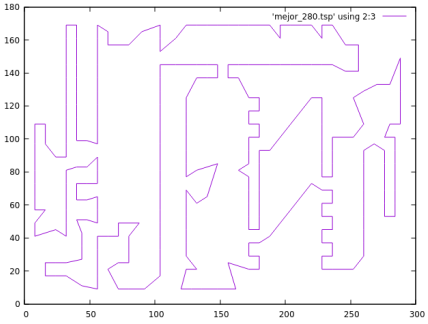
(a) Vecino más cercano



(b) Inserción más económica

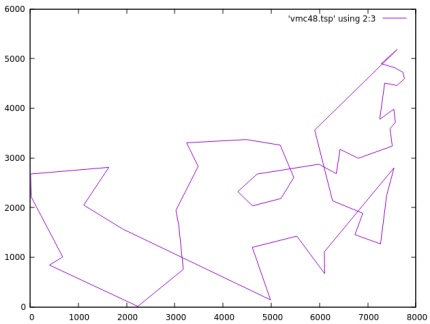


(c) Derivado de Kruskal

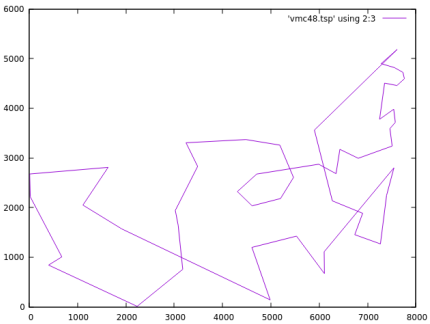


(d) Versión óptima

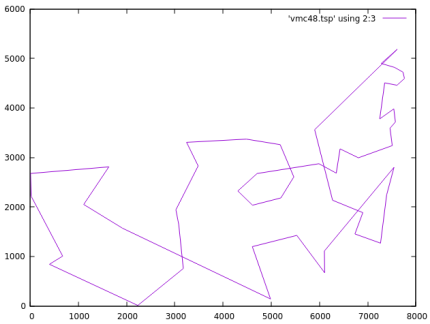
Figura 4: *a280.tsp*



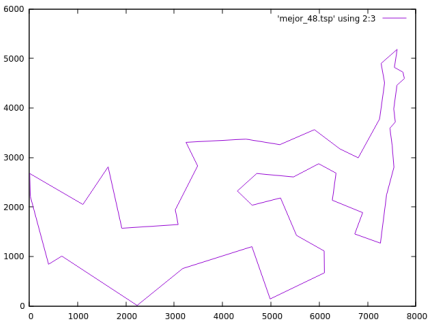
(a) Vecino más cercano



(b) Inserción más económica

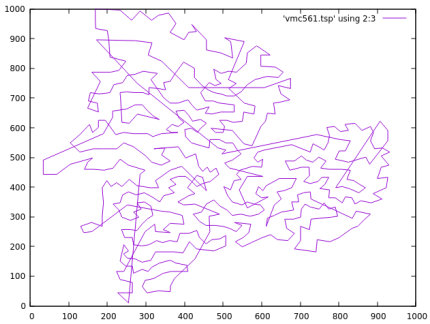


(c) Derivado de Kruskal

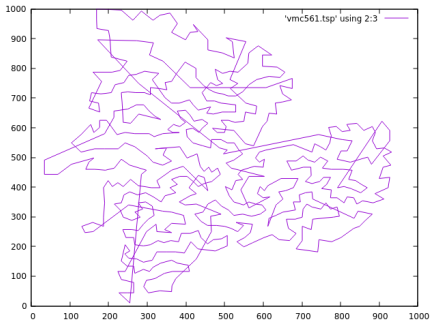


(d) Versión óptima

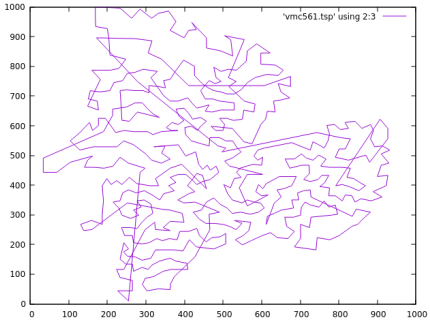
Figura 5: *att48.tsp*



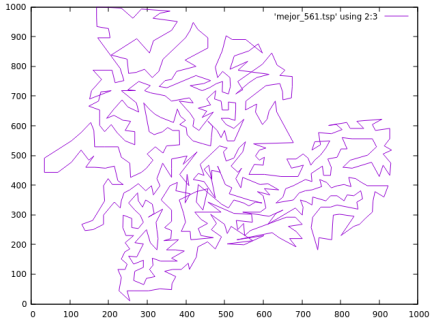
(a) Vecino más cercano



(b) Inserción más económica

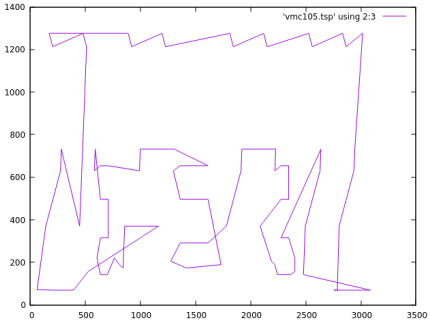


(c) Derivado de Kruskal

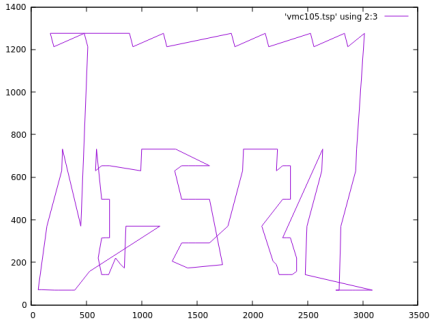


(d) Versión óptima

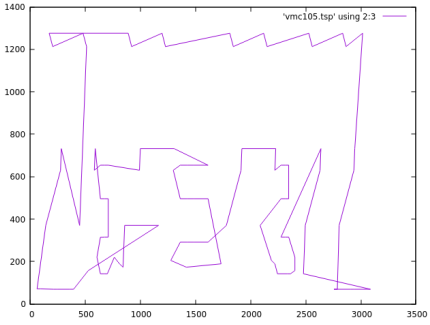
Figura 6: *pa561.tsp*



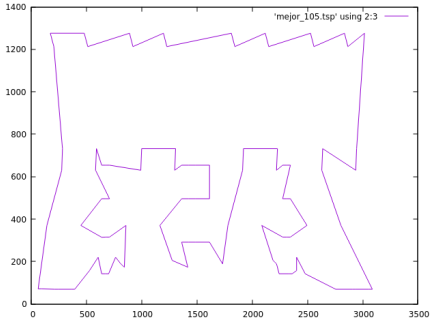
(a) Vecino más cercano



(b) Inserción más económica

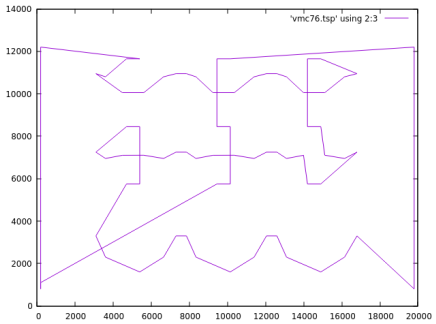


(c) Derivado de Kruskal

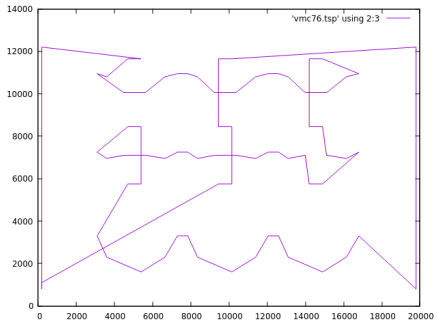


(d) Versión óptima

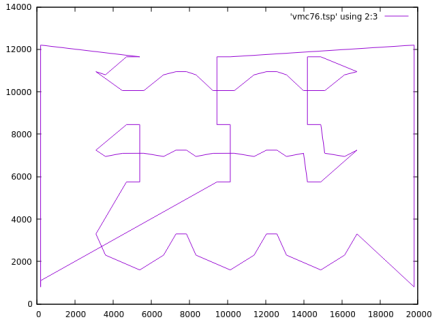
Figura 7: *lin105.tsp*



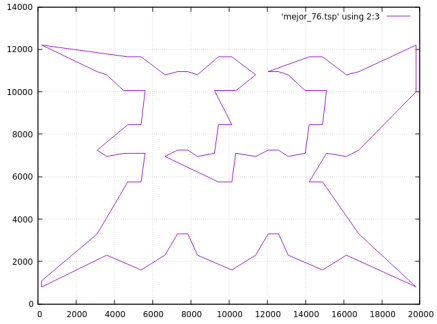
(a) Vecino más cercano



(b) Inserción más económica

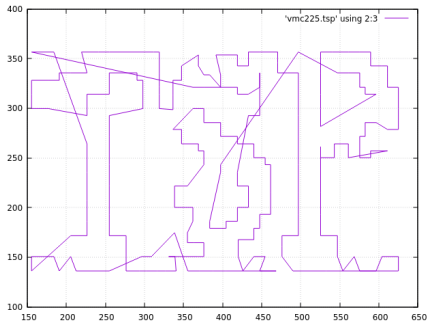


(c) Derivado de Kruskal



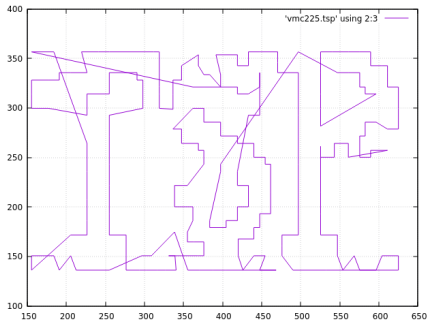
(d) Versión óptima

Figura 8: *pr76.tsp*

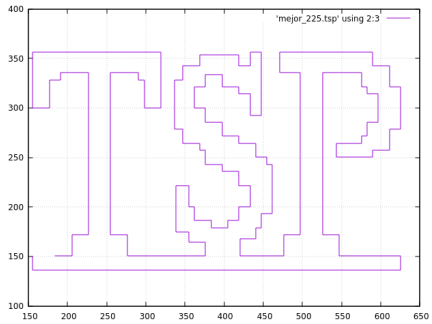


(a) Vecino más cercano

(b) Inserción más económica

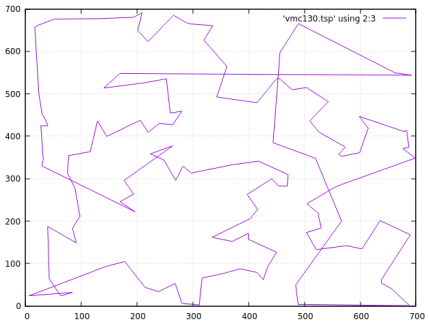


(c) Derivado de Kruskal

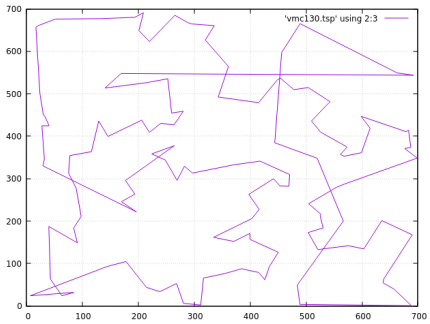


(d) Versión óptima

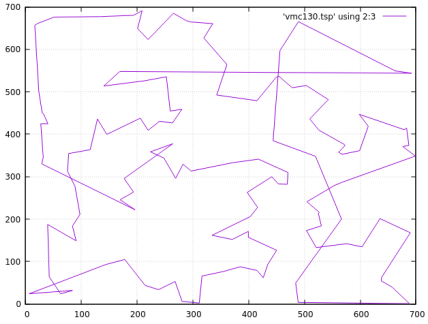
Figura 9: *tsp225.tsp*



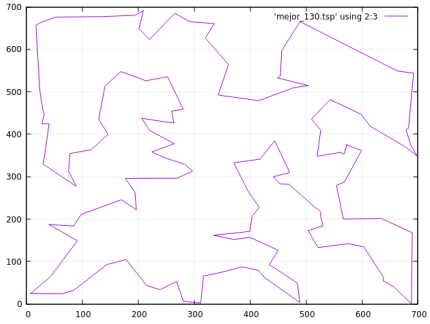
(a) Vecino más cercano



(b) Inserción más económica

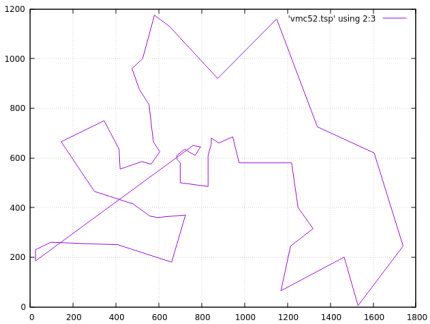


(c) Derivado de Kruskal

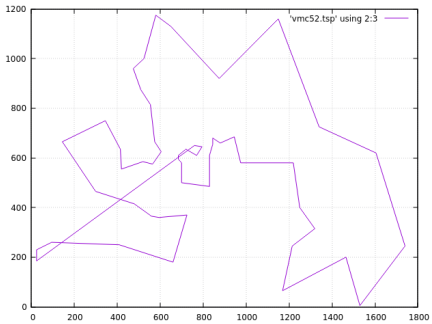


(d) Versión óptima

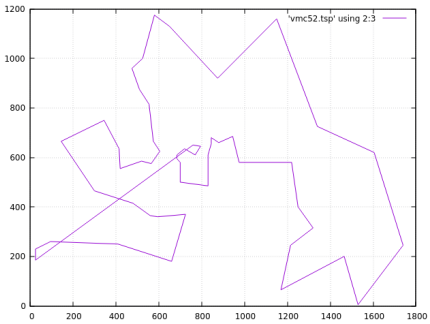
Figura 10: *ch130.tsp*



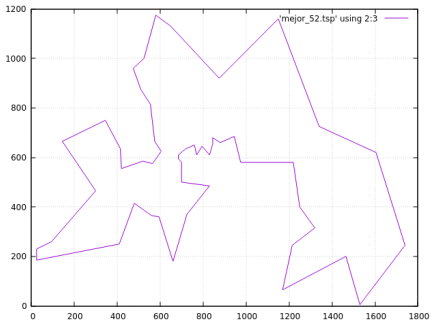
(a) Vecino más cercano



(b) Inserción más económica

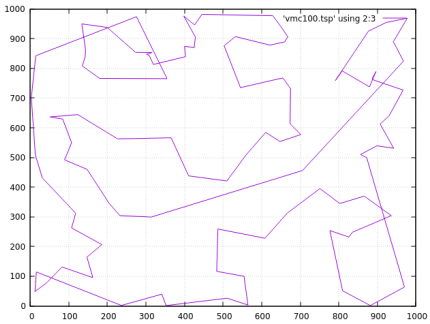


(c) Derivado de Kruskal

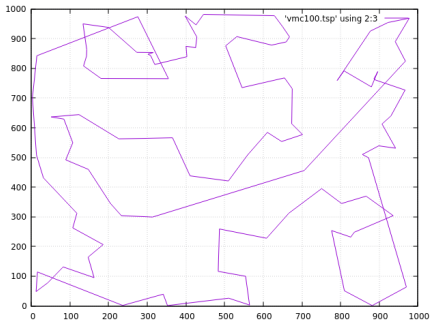


(d) Versión óptima

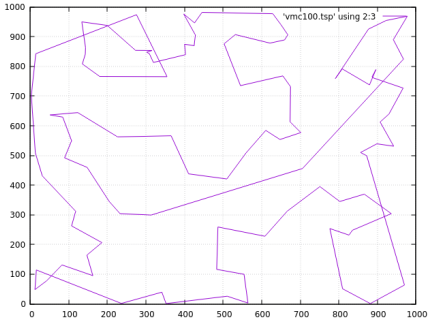
Figura 11: *berlin52.tsp*



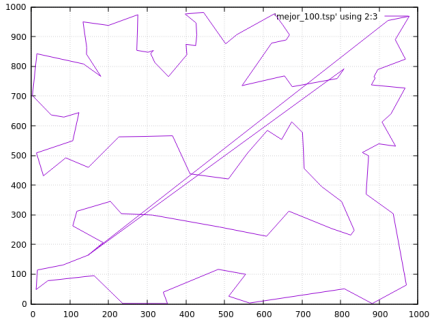
(a) Vecino más cercano



(b) Inserción más económica

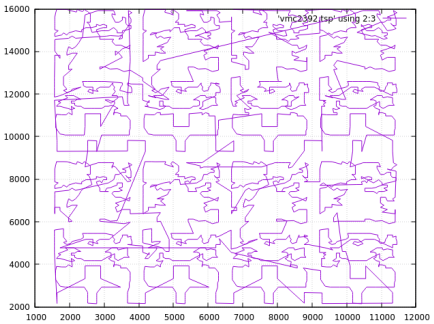


(c) Derivado de Kruskal

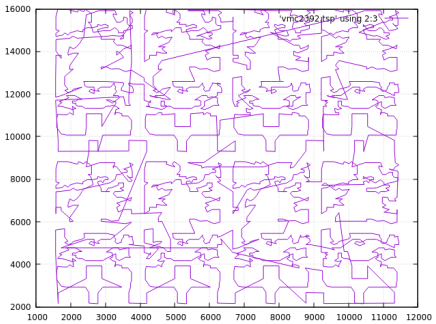


(d) Versión óptima

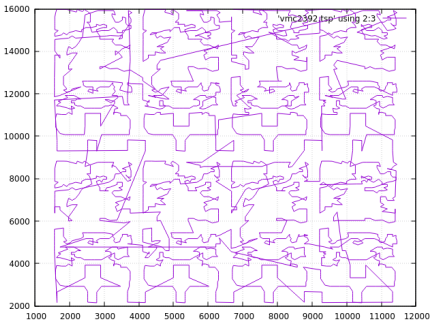
Figura 12: *rd100.tsp*



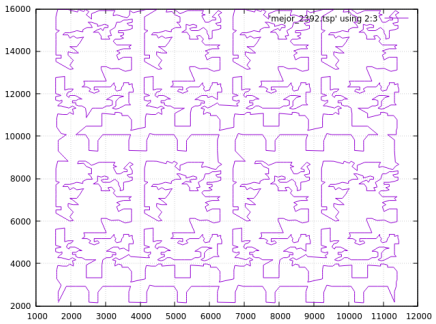
(a) Vecino más cercano



(b) Inserción más económica



(c) Derivado de Kruskal



(d) Versión óptima

Figura 13: *pr2392.tsp*

6. Anexo: código fuente

```
#include <iostream>
#include <fstream>
#include <vector>
#include <climits>
#include <cmath>
#include <string>

using namespace std;

#define TEST 0

struct ciudad {
  int n;
  double x;
  double y;
};

bool operator==(const ciudad &una, const ciudad &otra) {
  return una.x == otra.x && una.y == otra.y;
}
bool operator!=(const ciudad &una, const ciudad &otra) {
  return !(una == otra);
}

ostream& operator<<(ostream &flujo, const vector<ciudad> &v) {
  for (auto it = v.begin(); it != v.end(); ++it) {
    if (it != v.begin())
      flujo << "->";
    flujo << it->n ;
  }
  return flujo;
}


class TSP {

private:
  vector<ciudad> ciudades;
  double distancia_total;
  vector<ciudad> camino;
  vector<vector<double>> matriz_distancias;
  vector<bool> visitados;

  double calcularDistanciaCamino(const vector<ciudad> &path) {
    double distancia = 0;
    for (int i = 0, j = 1; j < path.size(); i++ , j++)
```

```
47        distancia += distanciaEuclidea(path[i], path[j]);
48      return distancia;
49    }
50
51    double distanciaEuclidea(const ciudad &una, const ciudad &otra) {
52      double resultado;
53      if (una == otra)
54        resultado = 0;
55      else
56        resultado = sqrt(pow(una.x - otra.x, 2) + pow(una.y - otra.y, 2));
57      return resultado;
58    }
59
60    int CiudadMasCercana(ciudad actual) {
61      double distancia_minima = INFINITY;
62      int ciudad = -1;
63      for (int i = 0; i < ciudades.size(); i++) {
64        if (matriz_distancias[actual.n][i] < distancia_minima &&
          ↪  !visitados[i]) {
65          distancia_minima = matriz_distancias[actual.n][i];
66          ciudad = i;
67        }
68      }
69      return ciudad;
70    }
71
72    pair<vector<ciudad>, double> VecinoMasCercanoParcial(int inicial) {
73      vector<ciudad> resultado;
74      ciudad actual = ciudades[inicial];
75      ciudad siguiente;
76      bool fin = false;
77      resultado.push_back(actual);
78      while (!fin) {
79        visitados[actual.n] = true;            //Pongo como visitados
80        int indice_siguiente = CiudadMasCercana(actual);  //Busco el indice
          ↪  de la siguiente ciudad
81        if (indice_siguiente != -1)              // Si he recorrido todas
          ↪  las ciudades, añado la primera.
82          siguiente = ciudades[indice_siguiente];
83        else {
84          fin = true;
85          siguiente = resultado[0];  //Volvemos al inicio
86        }
87        resultado.push_back(siguiente);
88        actual = siguiente;
89      }
90      double distancia = calcularDistanciaCamino(resultado);
91      pair<vector<ciudad>, double> par;
```

```
92      par.first = resultado;
93      par.second = distancia;
94      return par;
95    }
96
97    void InicializarMatrizDistancias() {
98      for (int i = 0; i < ciudades.size(); i++)
99        for (int j = 0; j < ciudades.size(); j++)
100         matriz_distancias[i][j] = distanciaEuclidea(ciudades[i],
            ↪  ciudades[j]);
101   }
102
103   void Reservar(int n) {
104     visitados.resize(n);
105     matriz_distancias.resize(n);
106     for (int i = 0; i < n; i++)
107       matriz_distancias[i].resize(n);
108   }
109
110   void ResetVisitados() {
111     for (auto it = visitados.begin(); it != visitados.end(); ++it)
112       *it = false;
113   }
114
115 public:
116   TSP() {
117     distancia_total = 0;
118     ResetVisitados();
119   }
120   TSP(char *archivo) {
121     CargarDatos(archivo);
122     distancia_total = 0;
123     ResetVisitados();
124   }
125   ~TSP() {
126
127   }
128
129   void VecinoMasCercano() {
130     pair<vector<ciudad>, double> minimo, temp;
131     minimo = VecinoMasCercanoParcial(0);    //Calculo el vecino más
        ↪  cercano comenzando por el primero
132     for (int i = 0; i < ciudades.size(); i++) {
133       ResetVisitados();
134       temp = VecinoMasCercanoParcial(i);
135 #if TEST
136       cout << temp.first << endl;
137       cout << "Distancia " << temp.second << endl;
```

```
138 #endif
139       if (temp.second < minimo.second)
140         minimo = temp;
141     }
142     camino = minimo.first;
143     distancia_total = minimo.second;
144   }
145
146   void InsercionMasEconomica() {
147
148   }
149
150   int GetTamanio() {
151     return ciudades.size();
152   }
153
154   void DerivadoKruskal() {
155
156   }
157
158   void CargarDatos(char *archivo) {
159     ifstream datos;
160     string s;
161     int n;
162     ciudad aux;
163     datos.open(archivo);
164     if (datos.is_open()) {
165       datos >> s;  //Leo DIMENSIÓN (cabecera)
166       datos >> n;  //Leo NÚMERO de ciudades y reservo espacio en
        ↪  matrices y vector.
167       Reservar(n);
168
169       for (int i = 0; i < n; i++) {
170         datos >> aux.n;  // Leo número de ciudad
171         aux.n--;      //Decremento el número: los índices del archivo
          ↪  comienzan en 1. Los del vector en 0.
172         datos >> aux.x >> aux.y; //Leo coordenadas
173         ciudades.push_back(aux);
174       }
175       datos.close();
176     }
177     else
178       cout << "Error al leer " << archivo << endl;
179
180     InicializarMatrizDistancias();
181   }
182
183   void imprimirResultado() {
```

```
184       cout << endl << "Mejor solución:" << endl;
185       cout << camino << endl;
186       cout << "Distancia: " << distancia_total << endl;
187     }
188
189     void Exportar(const char *name) {
190       ofstream salida;
191       salida.open(name);
192       if (salida.is_open()) {
193         salida << "DIMENSION: ";
194         salida << ciudades.size() << endl;
195         salida << "DISTANCIA: "<<distancia_total<<endl;
196         for (auto it = camino.begin(); it != camino.end(); ++it) {
197           salida << it->n + 1 << " " << it->x << " " << it->y << endl;
198         }
199         salida.close();
200       }
201       else
202         cout << "Error al exportar." << endl;
203     }
204
205   };
206
207
208   int main(int argc, char **argv) {
209
210     if (argc != 2) {
211       cerr << "Error de formato: " << argv[0] << " <fichero>." << endl;
212       exit(-1);
213     }
214
215     string nombre="";
216
217     /*********** Vecino más cercano*******************/
218     TSP vecino_mas_cercano(argv[1]);
219     cout << "Heurística del Vecino más cercano." << endl;
220     vecino_mas_cercano.VecinoMasCercano();
221     vecino_mas_cercano.imprimirResultado();
222
223     nombre = "vmc";
224     nombre += to_string(vecino_mas_cercano.GetTamanio());
225     nombre += ".tsp";
226     vecino_mas_cercano.Exportar(nombre.c_str());
227     cout << "Exportado archivo " << nombre << endl;
228
229     /*********** Inserción más económica*******************/
230
231     TSP insercion_mas_economica(argv[1]);
```

```
232
233   cout << "Heurística de la inserción más económica." << endl;
234   insercion_mas_economica.InsercionMasEconomica();
235   insercion_mas_economica.imprimirResultado();
236
237   /*nombre = "ime_";
238   nombre += argv[1];
239   insercion_mas_economica.Exportar(nombre.c_str());
240   cout << "Exportado archivo " << nombre << endl;*/
241
242
243   /*********** Derivado de Kruskal******************/
244
245   TSP derivado_kruskal(argv[1]);
246
247   cout << "Heurística derivada de Kruskal." << endl;
248   derivado_kruskal.DerivadoKruskal();
249   derivado_kruskal.imprimirResultado();
250
251   /*nombre = "kruskal_";
252   nombre += argv[1];
253
254   derivado_kruskal.Exportar(nombre.c_str());
255   cout << "Exportado archivo " << nombre << endl;*/
256
257 }
```

Figura 14: Programa que calcula el orden según las distintas heurísticas

```
1  #include <iostream>
2  #include <fstream>
3  #include <vector>
4  #include <climits>
5  #include <cmath>
6  #include <string>
7
8  using namespace std;
9
10
11 struct ciudad {
12   int n;
13   double x;
14   double y;
15 };
16
17 bool operator==(const ciudad &una, const ciudad &otra) {
18   return una.x == otra.x && una.y == otra.y;
19 }
```

```
20  bool operator!=(const ciudad &una, const ciudad &otra) {
21    return !(una == otra);
22  }
23
24  ostream& operator<<(ostream &flujo, const vector<ciudad> &v) {
25    for (auto it = v.begin(); it != v.end(); ++it) {
26      if (it != v.begin())
27        flujo << "->";
28      flujo << it->n ;
29    }
30    return flujo;
31  }
32
33
34  class TSP {
35
36  private:
37    vector<ciudad> ciudades;
38    double distancia_total;
39    vector<ciudad> camino;
40
41    double calcularDistanciaCamino(const vector<ciudad> &path) {
42      double distancia = 0;
43      for (int i = 0, j = 1; j < path.size(); i++ , j++)
44        distancia += distanciaEuclidea(path[i], path[j]);
45      return distancia;
46    }
47
48    double distanciaEuclidea(const ciudad &una, const ciudad &otra) {
49      double resultado;
50      if (una == otra)
51        resultado = 0;
52      else
53        resultado = sqrt(pow(una.x - otra.x, 2) + pow(una.y - otra.y, 2));
54      return resultado;
55    }
56
57
58  public:
59    TSP() {
60      distancia_total = 0;
61    }
62    TSP(char *archivo) {
63      CargarDatos(archivo);
64      distancia_total = 0;
65    }
66    ~TSP() {
67
```

```
68    }
69
70    int GetTamanio() {
71      return ciudades.size();
72    }
73
74
75    void MismoOrden(char *archivo) {
76      camino.clear();
77      ifstream datos;
78      string s;
79      int n;
80      int auxiliar;
81      datos.open(archivo);
82      if (datos.is_open()) {
83        datos >> s;  //Leo cabecera
84        datos >> n;  //Leo número ciudades
85        for (int i = 0; i < ciudades.size(); i++) {
86          datos >> auxiliar;    //Leo número de ciudad
87          auxiliar--;
88          bool fin = false;
89          for (auto it = ciudades.begin(); it != ciudades.end() && !fin;
            ↪  ++it) {  //Busco la ciudad y la inserto en el camino
90            if (it->n == auxiliar) {
91              camino.push_back(*it);
92              fin = true;
93            }
94          }
95        }
96        camino.push_back(camino[0]);
97        distancia_total = calcularDistanciaCamino(camino);
98      }
99      else
100       cout << "Error al leer" << archivo << endl;
101   }
102
103   int getDistancia(){
104     return distancia_total;
105   }
106
107   void CargarDatos(char *archivo) {
108     ifstream datos;
109     string s;
110     int n;
111     ciudad aux;
112     datos.open(archivo);
113     if (datos.is_open()) {
114       datos >> s;  //Leo DIMENSIÓN (cabecera)
```

```
115         datos >> n;  //Leo NÚMERO de ciudades .
116
117         for (int i = 0; i < n; i++) {
118           datos >> aux.n;  // Leo número de ciudad
119           aux.n--;      //Decremento el número: los índices del archivo
              ↪  comienzan en 1. Los del vector en 0.
120           datos >> aux.x >> aux.y; //Leo coordenadas
121           ciudades.push_back(aux);
122         }
123         datos.close();
124       }
125       else
126         cout << "Error al leer " << archivo << endl;
127     }
128
129     void Exportar(const char *name) {
130       ofstream salida;
131       salida.open(name);
132       if (salida.is_open()) {
133         salida << "DIMENSION: ";
134         salida << ciudades.size() << endl;
135         salida << "DISTANCIA: " << distancia_total << endl;
136         for (auto it = camino.begin(); it != camino.end(); ++it) {
137           salida << it->n + 1 << " " << it->x << " " << it->y << endl;
138         }
139         salida.close();
140       }
141       else
142         cout << "Error al exportar." << endl;
143     }
144
145 };
146
147 // Modo de uso: ./programa coordenadas orden
148 int main(int argc, char **argv) {
149
150   if (argc != 3) {
151     cerr << "Error de formato: " << argv[0] << " <coordenadas> <orden>."
      ↪  << endl;
152     exit(-1);
153   }
154
155   string nombre = "";
156
157   /********** Vecino más cercano*******************/
158   TSP instancia(argv[1]);
159
160
```

```
161
162
163     instancia.MismoOrden(argv[2]);
164
165     nombre = "mejor_";
166     nombre += to_string(instancia.GetTamanio());
167     nombre += ".tsp";
168     instancia.Exportar(nombre.c_str());
169     cout << "Exportado archivo " << nombre << endl;
170
171     cout<<"DISTANCIA "<<instancia.getDistancia()<<endl;
172
173
174 }
```

Figura 15: Programa que calcula la distancia del circuito a partir de un fichero con coordenadas y una lista ordenada de ciudades.