Algorítmica

grado en ingeniería informática

Práctica 4

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

Índice de figuras

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 la heurística de *Branch and Bound*.

Tiene las siguientes características:

- **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 selección**. De entre todos los candidatos, elegimos aquella ciudad que incrementa menos el coste del circuito que llevamos hasta el momento.

2. Descripción del algoritmo

2.1. Datos y estructuras utilizadas

- **Distancia**. Comienza inicializada a $+\infty$.
- **Cota inferior**. Utilizamos una cota inferior optimista que inicializamos mediante un algoritmo greedy, aproximado al algoritmo *vecino más cercano*. La heurística que sigue el algoritmo es la siguiente:

$$\frac{1}{2}\sum_{i=0}^{n} coste_{entrada}(i) + coste_{salida}(i)$$

  En la sumatoria se acumulan progresivamente los costes de entrada y de salida de cada nodo que sean menores entre las aristas posibles de cada uno. Tras completar la sumatoria se divide a la mitad debido a que la salida de un nodo es la entrada del siguiente.
- **Solución parcial**. Es un vector que contiene la solución, pudiendo o no estar completa. En el caso de que esté completa pasa a ser la nueva cota.
- **Visitados**. Es un vector de booleanos que contiene *true* si la ciudad ya ha sido visitada y *false* en otro caso.

2.2. Procedimiento

El ajuste inicial que se lleva a cabo es el siguiente:

1. Se calcula la cota inferior inicial mediante el algoritmo greedy anteoriormente explicado.
2. Se toma la primera ciudad y se introduce en la solución parcial. La ciudad ya ha sido visitada, por tanto se pone a *true* en el vector de visitados.

3. Se llama entonces al método recursivo que lleva a cabo el algoritmo *Branch and Bound* propiamente dicho.

El seguimiento del algoritmo es el que sigue:

1. En el caso base se comprueba si hemos llegado a un nodo hoja del árbol. Si efectivamente estamos en un nodo hoja, cerramos el circuito y comprobamos si la solución parcial actual es mejor que la global (la distancia es menor). En caso afirmativo, ésta se actualiza.

2. Si no estamos en el caso base, se siguen los siguientes pasos

   a) Recorremos todas las ciudades restantes.

   b) Comprobamos que no ha sido visitada y que no es la misma ciudad en la que estamos actualmente.

   c) Calculamos el coste que supone añadir la nueva ciudad al recorrido.

   d) Calculamos la cota de la rama actual. Se puede calcular de dos formas: si el nivel es el 1 la calculamos como la media entre el menor arco entrante de la última ciudad de la solución parcial y la nueva que queremos añadir. Si el nivel no es el 1, se calcula como la media entre el menor arco saliente de la última ciudad de la solución parcial y el menor arco entrante de la ciudad a añadir.

   e) Comprobamos si la suma entre la cota actual y el peso actual es menor que la distancia total, hemos encontrado una cota menor y por tanto exploramos los hijos mediante una llamada recursiva.

   f) Deshacemos los cambios que hemos realizado para que el siguiente hijo que evaluemos encuentre las variables en las mismas condiciones que el que se acaba de comprobar.

3. Resultados obtenidos

Número de ciudades	**Tiempo(s)**
6	$3{,}3886 \cdot 10^{-5}$
7	0.000140369
8	0.000602488
9	0.00357393
10	0.0154363
11	0.128727
12	0.658938
13	4.02953
14	31.2847
15	245.842 (4 minutos y 5.842 segundos)
16	1541.08 (25 minutos y 41 segundos)

Figura 1: Tiempos obtenidos

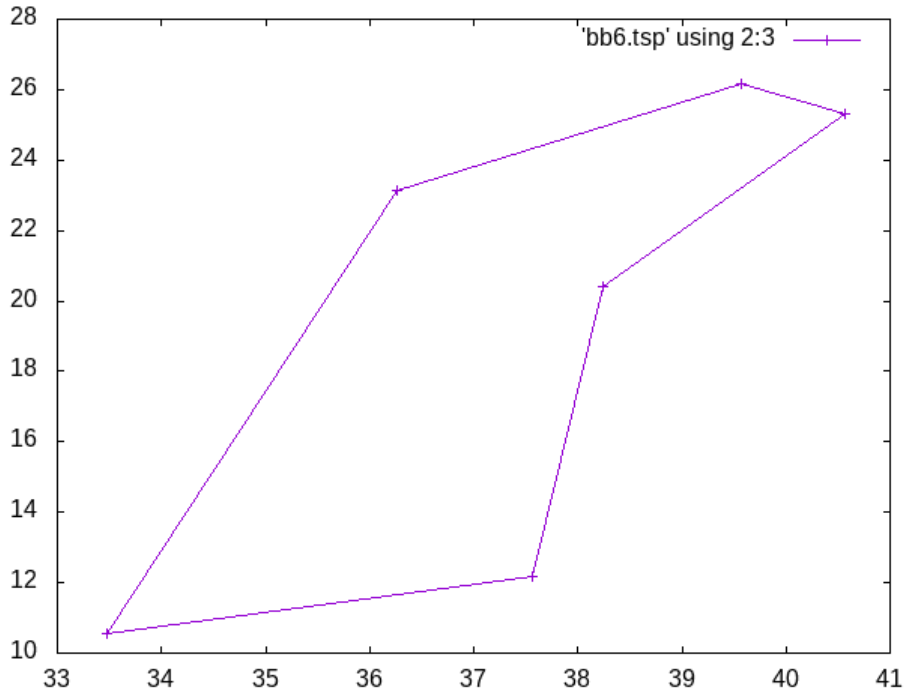

Figura 2: ulysses6.tsp

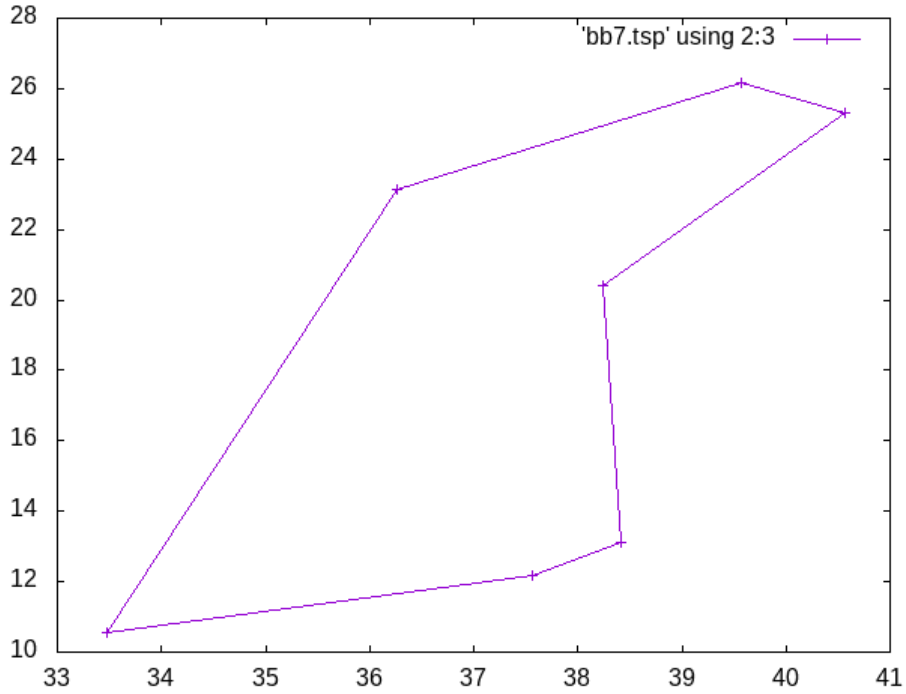

Figura 3: ulysses7.tsp

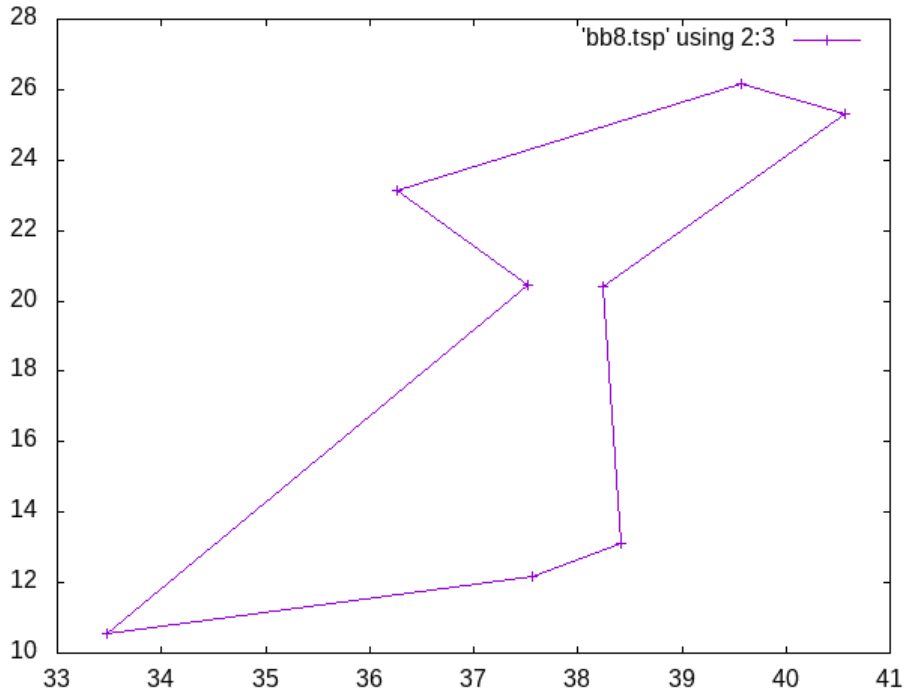

Figura 4: ulysses8.tsp

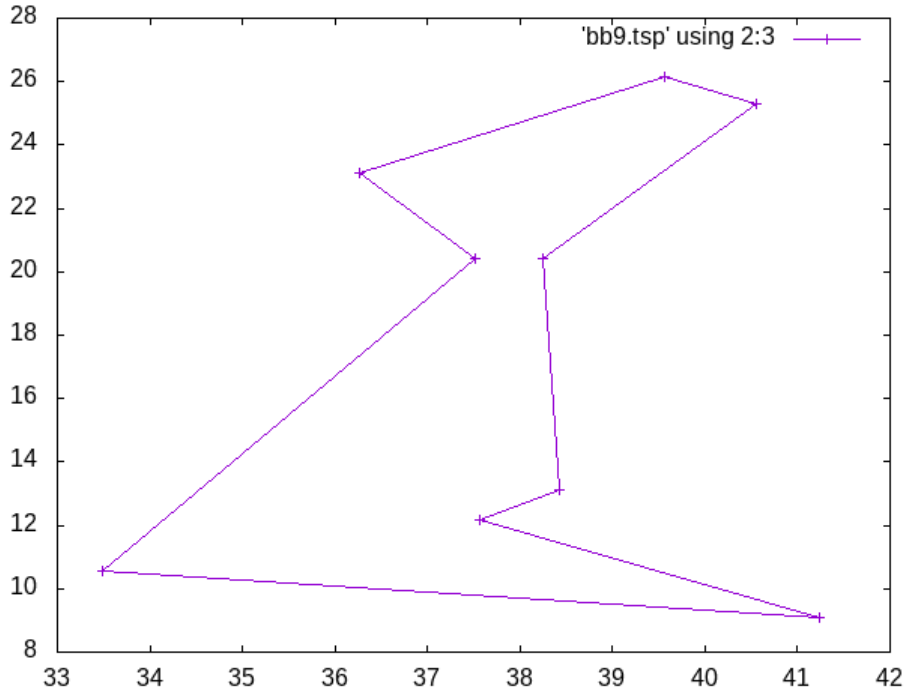

Figura 5: ulysses9.tsp

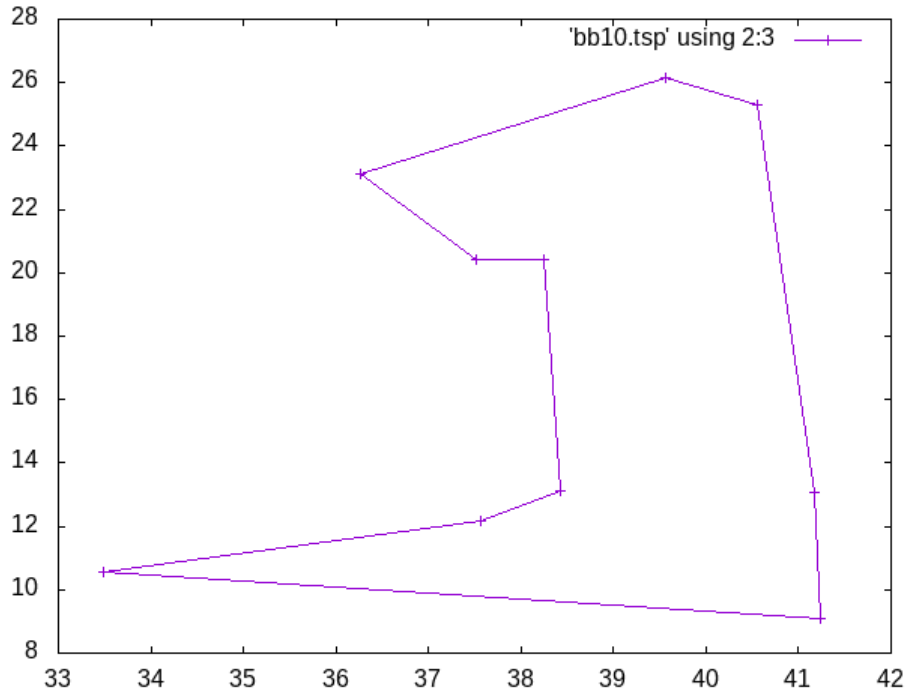

Figura 6: ulysses10.tsp

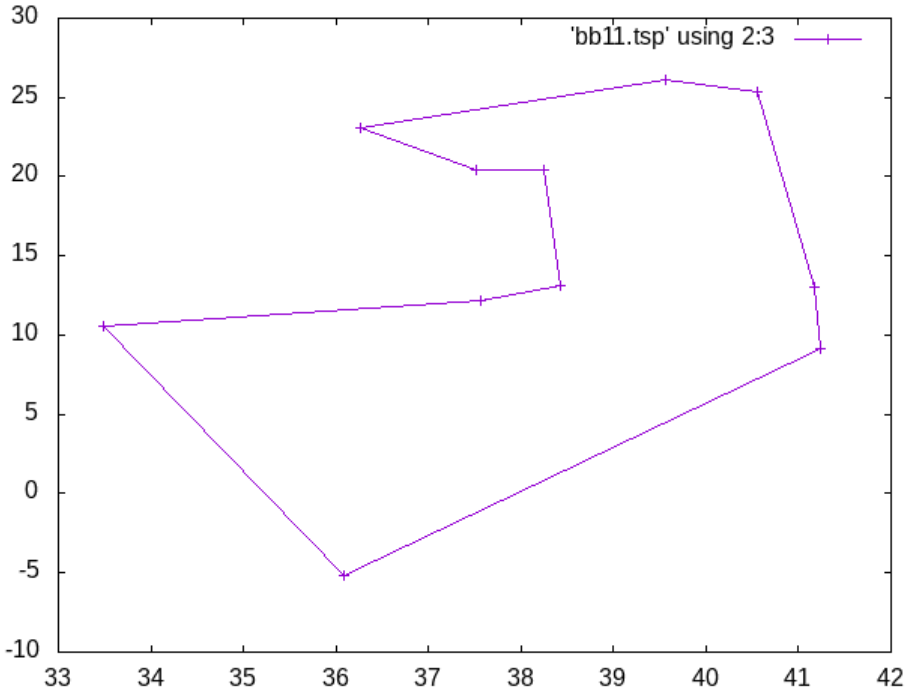

Figura 7: ulysses11.tsp

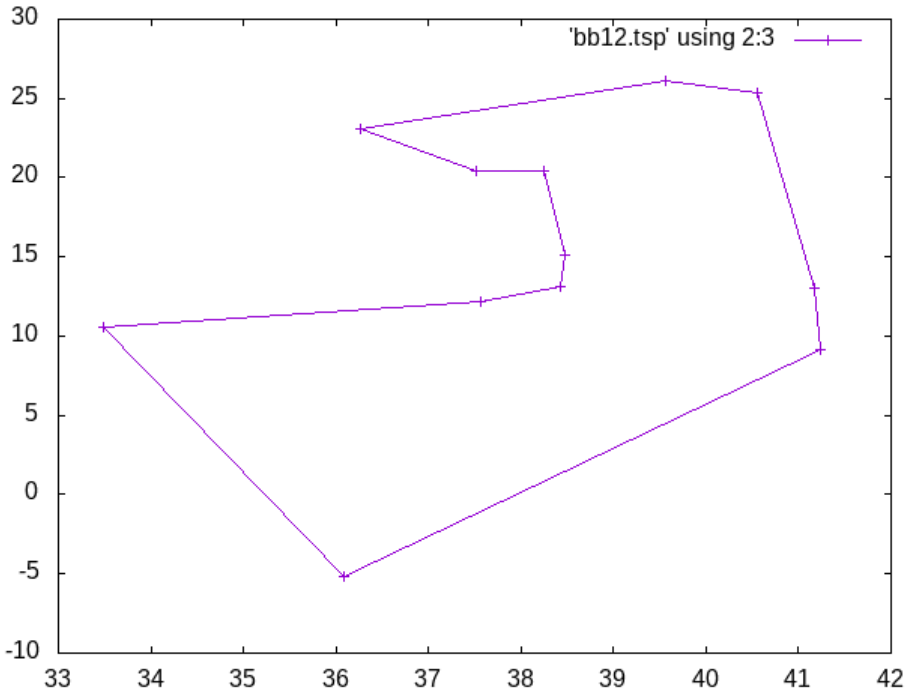

Figura 8: ulysses12.tsp

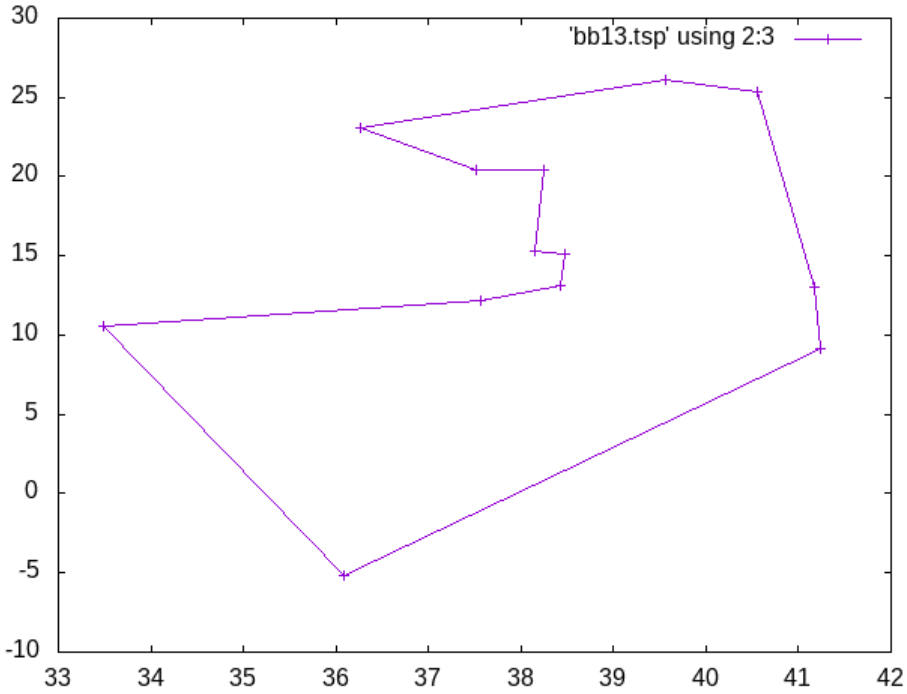

Figura 9: ulysses13.tsp

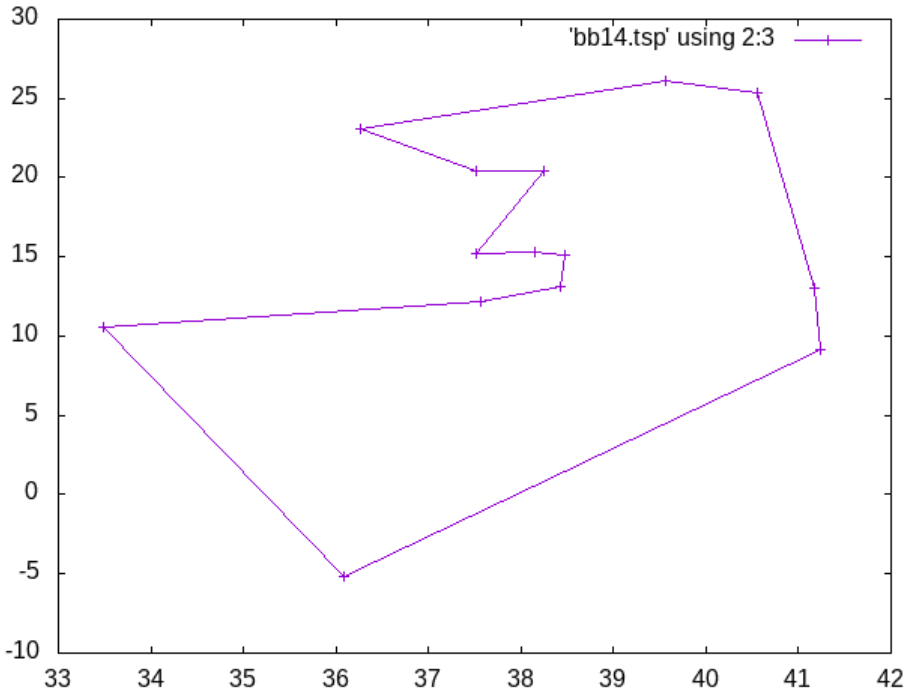

Figura 10: ulysses14.tsp

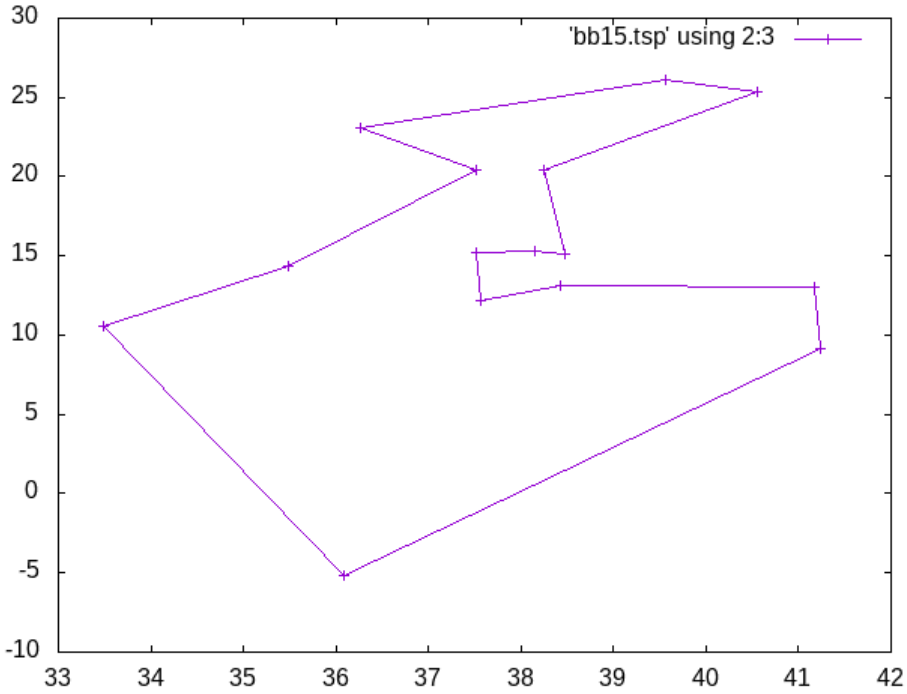

Figura 11: ulysses15.tsp

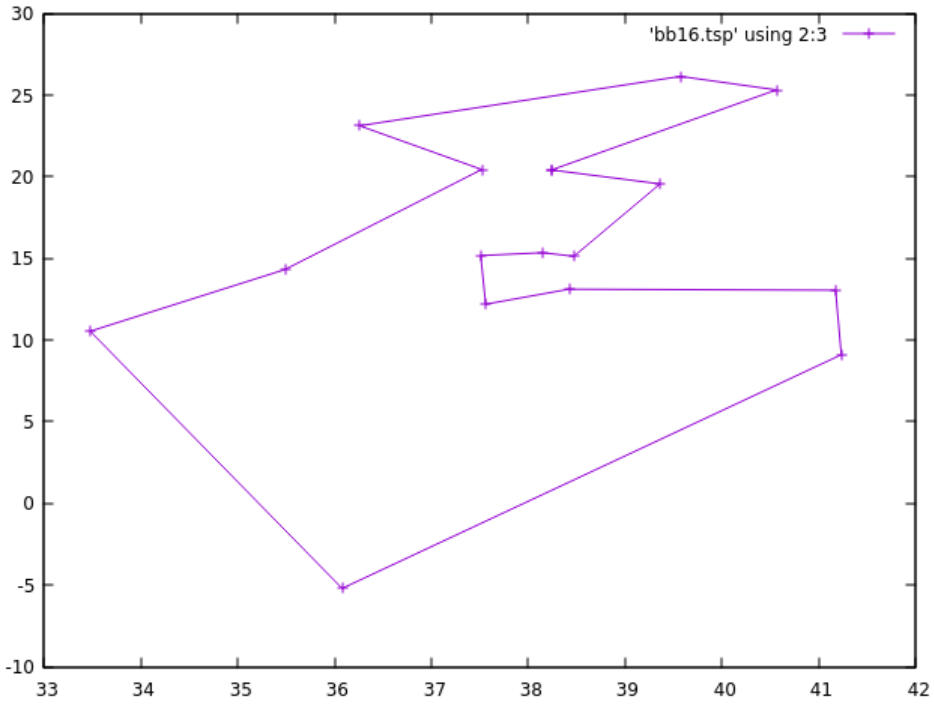

Figura 12: ulysses16.tsp

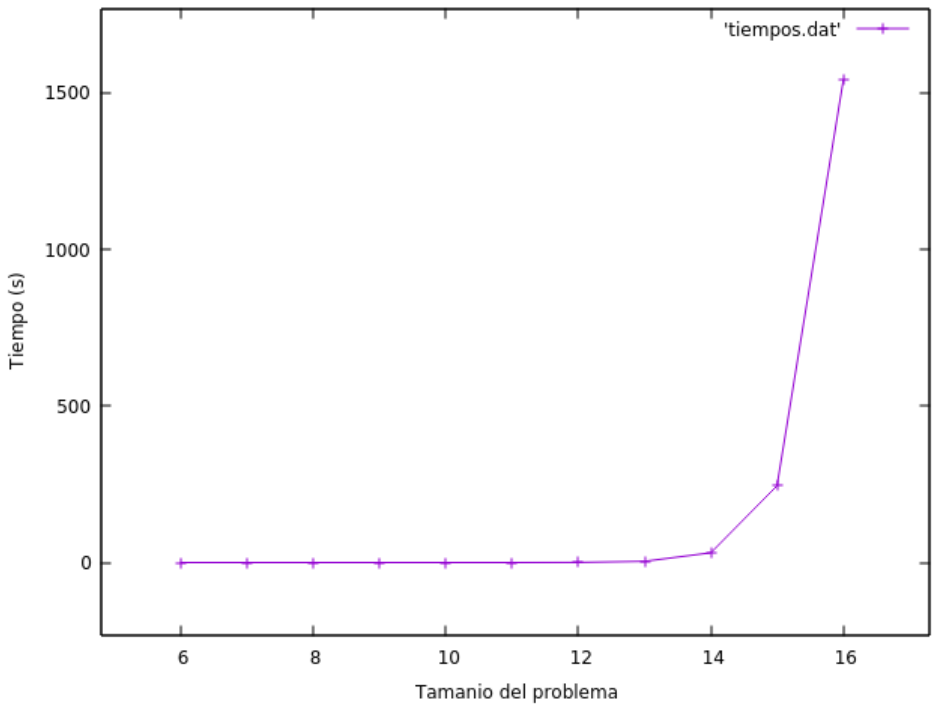

Figura 13: Evolución del tiempo de ejecución

4. Conclusiones

Lo más destacable de este algoritmo es que en términos de orden de eficiencia es pésimo pero, sin embargo, proporciona una solución muy óptima.

5. Anexo: código fuente

```
#include <chrono>
#include <climits>
#include <cmath>
#include <ctime>
#include <fstream>
#include <iostream>
#include <limits>
#include <list>
#include <string>
#include <vector>
using namespace std;
using namespace std::chrono;

#define DEBUG 0

struct ciudad {
  int n;
  double x;
  double y;
  ciudad &operator=(const ciudad &otra) {
    if (this != &otra) {
      x = otra.x;
      y = otra.y;
      n = otra.n;
    }
    return *this;
  }
};

bool operator==(const ciudad &una, const ciudad &otra) {
  return una.n == otra.n;
}
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 + 1;
  }
  return flujo;
}
class TravelSalesman {

```

```
47  private:
48    const double MAX = numeric_limits<double>::max();
49    vector<ciudad> ciudades; // Almaceno problema
50    double distancia_total;  // Distancia del circuito
51    vector<ciudad> camino;   // Solución final.
52    vector<vector<double>> matriz_distancias;
53    vector<bool> visitados;
54    double calcularDistanciaCamino(const vector<ciudad> &path) const;
55    double distanciaEuclidea(const ciudad &una, const ciudad &otra) const;
56    void InicializarMatrizDistancias();
57    void Reservar(int n);
58    void ResetVisitados();
59    void RecBranchBound(double cota_actual, double peso_actual, int nivel,
60                        vector<ciudad> solucion_parcial);
61    double CalcularCotaInicial() const;
62
63    double MenorEntrante(const ciudad &ciudad) const;
64
65    double MenorSaliente(const ciudad &ciudad) const;
66    double Distancia(const ciudad &a, const ciudad &b) const;
67
68  public:
69    TravelSalesman();
70    TravelSalesman(char *archivo);
71    int GetTamanio() const;
72    void CargarDatos(char *archivo);
73    void imprimirResultado() const;
74    void Exportar(const char *name) const;
75
76    void BranchBound();
77  };
78
79  TravelSalesman::TravelSalesman() {
80    distancia_total = MAX;
81    ResetVisitados();
82  }
83
84  TravelSalesman::TravelSalesman(char *archivo) {
85    CargarDatos(archivo);
86    distancia_total = MAX;
87    ResetVisitados();
88  }
89
90  double TravelSalesman::Distancia(const ciudad &a, const ciudad &b) const {
91    return matriz_distancias[a.n][b.n];
92  }
93  void TravelSalesman::imprimirResultado() const {
94    cout << endl << "Mejor solución:" << endl;
```

```
95    cout << camino << endl;
96    cout << "Distancia: " << distancia_total << endl;
97  }
98
99  void TravelSalesman::Exportar(const char *name) const {
100   ofstream salida;
101   salida.open(name);
102   if (salida.is_open()) {
103     salida << "DIMENSION: ";
104     salida << ciudades.size() << endl;
105     salida << "DISTANCIA: " << distancia_total << endl;
106     for (auto it = camino.begin(); it != camino.end(); ++it) {
107       salida << it->n + 1 << " " << it->x << " " << it->y << endl;
108     }
109     salida.close();
110   } else
111     cout << "Error al exportar." << endl;
112 }
113
114 void TravelSalesman::CargarDatos(char *archivo) {
115   ifstream datos;
116   string s;
117   int n;
118   ciudad aux;
119   datos.open(archivo);
120   if (datos.is_open()) {
121     datos >> s; // Leo DIMENSIÓN (cabecera)
122     datos >> n; // Leo NÚMERO de ciudades .
123     Reservar(n);
124     for (int i = 0; i < n; i++) {
125       datos >> aux.n; // Leo número de ciudad
126       aux.n--; // Decremento el número: los índices del archivo comienzan
127       // en 1. Los del vector en 0.
128       datos >> aux.x >> aux.y; // Leo coordenadas
129       ciudades.push_back(aux);
130     }
131     datos.close();
132   } else
133     cout << "Error al leer " << archivo << endl;
134   InicializarMatrizDistancias();
135 }
136
137 int TravelSalesman::GetTamanio() const { return ciudades.size(); }
138
139 void TravelSalesman::ResetVisitados() {
140   for (auto it = visitados.begin(); it != visitados.end(); ++it)
141     *it = false;
142 }
```

```
143
144 double
145 TravelSalesman::calcularDistanciaCamino(const vector<ciudad> &path) const {
146   double distancia = 0;
147   for (int i = 0, j = 1; j < path.size(); i++, j++)
148     distancia += distanciaEuclidea(path[i], path[j]);
149   return distancia;
150 }
151
152 double TravelSalesman::distanciaEuclidea(const ciudad &una,
153                                          const ciudad &otra) const {
154   double resultado;
155   if (una == otra)
156     resultado = 0;
157   else
158     resultado = sqrt(pow(una.x - otra.x, 2) + pow(una.y - otra.y, 2));
159   return resultado;
160 }
161
162 void TravelSalesman::InicializarMatrizDistancias() {
163   for (int i = 0; i < ciudades.size(); i++)
164     for (int j = 0; j < ciudades.size(); j++)
165       matriz_distancias[i][j] = distanciaEuclidea(ciudades[i], ciudades[j]);
166 }
167
168 void TravelSalesman::Reservar(int n) {
169   visitados.resize(n);
170   matriz_distancias.resize(n);
171   for (int i = 0; i < n; i++)
172     matriz_distancias[i].resize(n);
173 }
174
175 double TravelSalesman::MenorEntrante(const ciudad &ciudad) const {
176   double menor = MAX;
177   for (int i = 0; i < ciudades.size(); i++)
178     if (i != ciudad.n && matriz_distancias[i][ciudad.n] < menor)
179       menor = matriz_distancias[i][ciudad.n];
180   return menor;
181 }
182
183 double TravelSalesman::MenorSaliente(const ciudad &ciudad) const {
184   double menor_entrante = MAX, menor_saliente = MAX;
185   for (int i = 0; i < ciudades.size(); i++) {
186     if (ciudad.n != i) {
187       if (matriz_distancias[i][ciudad.n] <= menor_entrante) {
188         menor_saliente = menor_entrante;
189         menor_entrante = matriz_distancias[i][ciudad.n];
190       } else if (matriz_distancias[i][ciudad.n] <= menor_saliente &&
```

```
191                   matriz_distancias[i][ciudad.n] != menor_entrante)
192           menor_saliente = matriz_distancias[i][ciudad.n];
193       }
194     }
195     return menor_saliente;
196   }
197
198   double TravelSalesman::CalcularCotaInicial() const {
199     double cota = 0;
200     for (auto it = ciudades.begin(); it != ciudades.end(); ++it)
201       cota += MenorEntrante(*it) + MenorSaliente(*it);
202     cota /= 2;
203     return cota;
204   }
205
206   void TravelSalesman::BranchBound() {
207     double cota_inicial = CalcularCotaInicial();
208     vector<ciudad> solucion_parcial;
209     solucion_parcial.push_back(ciudades[0]); // Meto primera ciudad.
210     visitados[0] = true;
211     RecBranchBound(cota_inicial, 0, 1, solucion_parcial);
212   }
213
214   void TravelSalesman::RecBranchBound(double cota_actual, double peso_actual,
215                                       int nivel,
216                                       vector<ciudad> solucion_parcial) {
217     if (nivel == ciudades.size()) { // Caso base
218       double resultado_actual = peso_actual + Distancia(solucion_parcial.back(),
219
                                                          ↪  solucion_parcial.front());
220       if (resultado_actual < distancia_total) {
221         distancia_total = resultado_actual;
222         camino = solucion_parcial;
223         camino.push_back(camino.front());
224       }
225     } else { // Sigo expandiendo
226       for (auto it = ciudades.begin(); it != ciudades.end(); ++it) {
227         if (Distancia(*it, solucion_parcial.back()) != 0 && !visitados[it->n]) {
228           double cota_local = cota_actual;
229           double p_nuevo = peso_actual + Distancia(*it,
            ↪  solucion_parcial.back());
230
231           if (nivel == 1)
232             cota_local -= (MenorEntrante(solucion_parcial[nivel - 1]) +
233                            MenorEntrante(*it)) /
234                           2;
235           else
236             cota_local -= (MenorSaliente(solucion_parcial[nivel - 1]) +
```

```
237                             MenorEntrante(*it)) /
238                            2;
239           double actual = cota_local + peso_actual;
240           if (actual < distancia_total) { // La solución puede mejorar
241             solucion_parcial.push_back(*it);
242             visitados[it->n] = true;
243             RecBranchBound(cota_local, p_nuevo, nivel + 1, solucion_parcial);
244             solucion_parcial.erase(solucion_parcial.end()-1); //Deshago el
                ↪  cambio.
245           }
246           // Deshacemos cambios
247           ResetVisitados();
248           for (auto it = solucion_parcial.begin(); it != solucion_parcial.end();
249                ++it)
250             visitados[it->n] = true;
251         }
252       }
253     }
254   }
255   int main(int argc, char **argv) {
256
257     if (argc != 2) {
258       cerr << "Error de formato: " << argv[0] << " <fichero>." << endl;
259       exit(-1);
260     }
261     TravelSalesman instancia(argv[1]);
262
263     auto tantes = high_resolution_clock::now();
264
265     instancia.BranchBound();
266
267     auto tdespues = high_resolution_clock::now();
268
269     double tiempo = duration_cast<duration<double>>(tdespues - tantes).count();
270
271     cout << "Tamaño=" << instancia.GetTamanio() << " Tiempo (s)=" << tiempo
272          << endl;
273
274     string nombre;
275     nombre = "bb";
276     nombre += to_string(instancia.GetTamanio());
277     nombre += ".tsp";
278     instancia.Exportar(nombre.c_str());
279
280   #if DEBUG
281     instancia.imprimirResultado();
282   #endif
283
```

```
284   return 0;
285 }
```

Figura 14: TSP mediante Branch&Bound

```
1  #!/bin/bash
2  g++ -o tsp tsp.cpp -Ofast;
3  ./tsp ../datosTSP/ulysses6.tsp;
4  ./tsp ../datosTSP/ulysses7.tsp;
5  ./tsp ../datosTSP/ulysses8.tsp;
6  ./tsp ../datosTSP/ulysses9.tsp;
7  ./tsp ../datosTSP/ulysses10.tsp;
8  ./tsp ../datosTSP/ulysses11.tsp;
9  ./tsp ../datosTSP/ulysses12.tsp;
10 ./tsp ../datosTSP/ulysses13.tsp;
11 ./tsp ../datosTSP/ulysses14.tsp;
12 ./tsp ../datosTSP/ulysses15.tsp;
13 ./tsp ../datosTSP/ulysses16.tsp;
14 ./gnuplot.sh
```

Figura 15: Ejecución de la práctica

```
1  #!/usr/bin/gnuplot
2
3  set terminal png size 640,480
4
5  set output 'bb6.png'
6  plot 'bb6.tsp' using 2:3 with linespoints
7
8  set output 'bb7.png'
9  plot 'bb7.tsp' using 2:3 with linespoints
10
11 set output 'bb8.png'
12 plot 'bb8.tsp' using 2:3 with linespoints
13
14 set output 'bb9.png'
15 plot 'bb9.tsp' using 2:3 with linespoints
16
17 set output 'bb10.png'
18 plot 'bb10.tsp' using 2:3 with linespoints
19
20 set output 'bb11.png'
21 plot 'bb11.tsp' using 2:3 with linespoints
22
23 set output 'bb12.png'
24 plot 'bb12.tsp' using 2:3 with linespoints
25
26 set output 'bb13.png'
```

```
27 plot 'bb13.tsp' using 2:3 with linespoints
28
29 set output 'bb14.png'
30 plot 'bb14.tsp' using 2:3 with linespoints
31
32 set output 'bb15.png'
33 plot 'bb15.tsp' using 2:3 with linespoints
34
35 set output 'bb16.png'
36 plot 'bb16.tsp' using 2:3 with linespoints
```

Figura 16: Generador de gráficas