



ugr

Universidad  
de Granada

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



**DECSAI**  
**Departamento de Ciencias de la Computación e I.A.**  
Universidad de Granada

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE TELECOMUNICACIÓN

CURSO 2017-2018

# Índice

<b>1. Descripción de la práctica</b>	<b>1</b>
<b>2. Descripción del algoritmo</b>	<b>1</b>
2.1. Datos y estructuras utilizadas . . . . .	1
2.2. Procedimiento . . . . .	1
<b>3. Resultados obtenidos</b>	<b>2</b>
<b>4. Conclusiones</b>	<b>7</b>
<b>5. Anexo: código fuente</b>	<b>8</b>

# Índice de figuras

1. Tiempos obtenidos . . . . .	2
2. ulysses6.tsp . . . . .	3
3. ulysses7.tsp . . . . .	3
4. ulysses8.tsp . . . . .	4
5. ulysses9.tsp . . . . .	4
6. ulysses10.tsp . . . . .	5
7. ulysses11.tsp . . . . .	5
8. ulysses12.tsp . . . . .	6
9. ulysses13.tsp . . . . .	6
10. ulysses14.tsp . . . . .	7
11. ulysses15.tsp . . . . .	7
12. TSP mediante Branch&Bound . . . . .	13

# 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:

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

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 anteriormente 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	$1,27 \cdot 10^{-5}$
7	$4,39 \cdot 10^{-5}$
8	0,0002036
9	0,0054381
10	0,0325048
11	0,381596
12	2,23487
13	8,90865
14	107,772 (2 minutos y 20 segundos)
15	1192,761 (19 minutos y 53 segundos)
16	+4h

Figura 1: Tiempos obtenidos

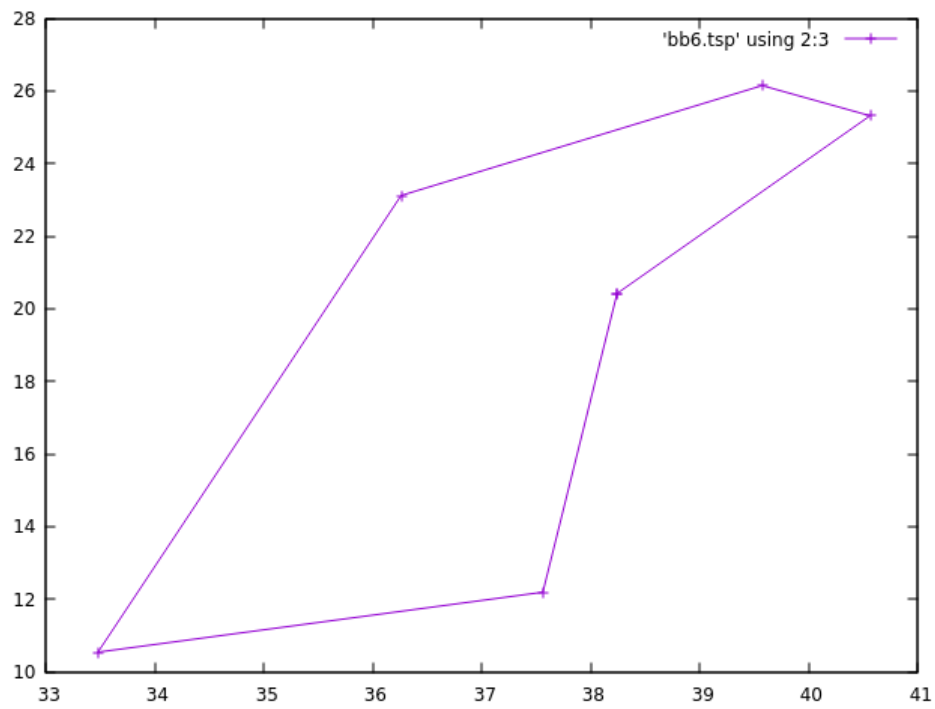


Figura 2: ulysses6.tsp

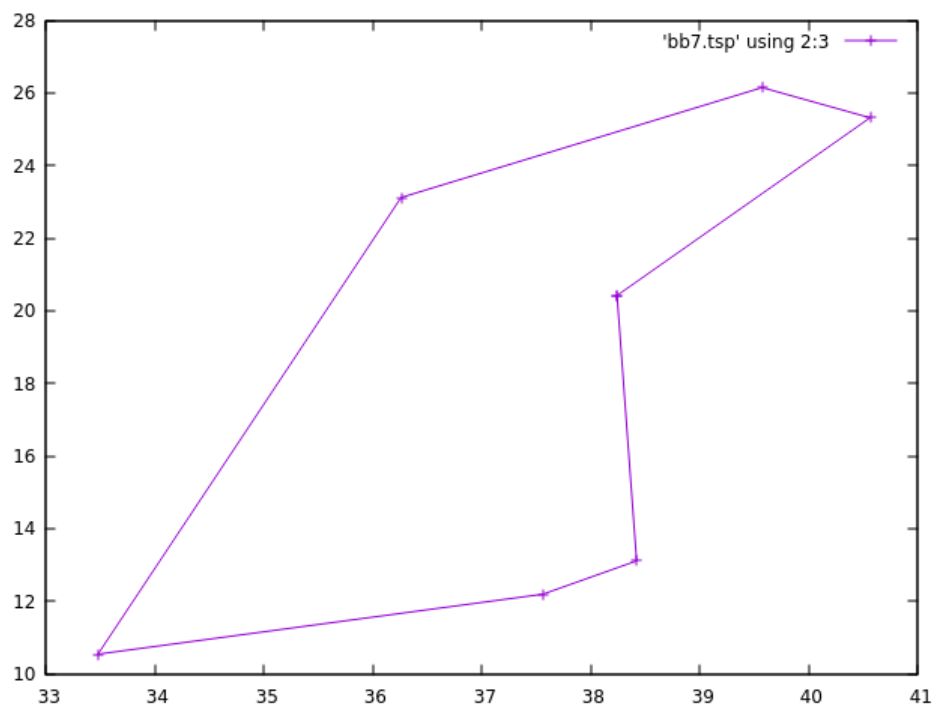


Figura 3: ulysses7.tsp

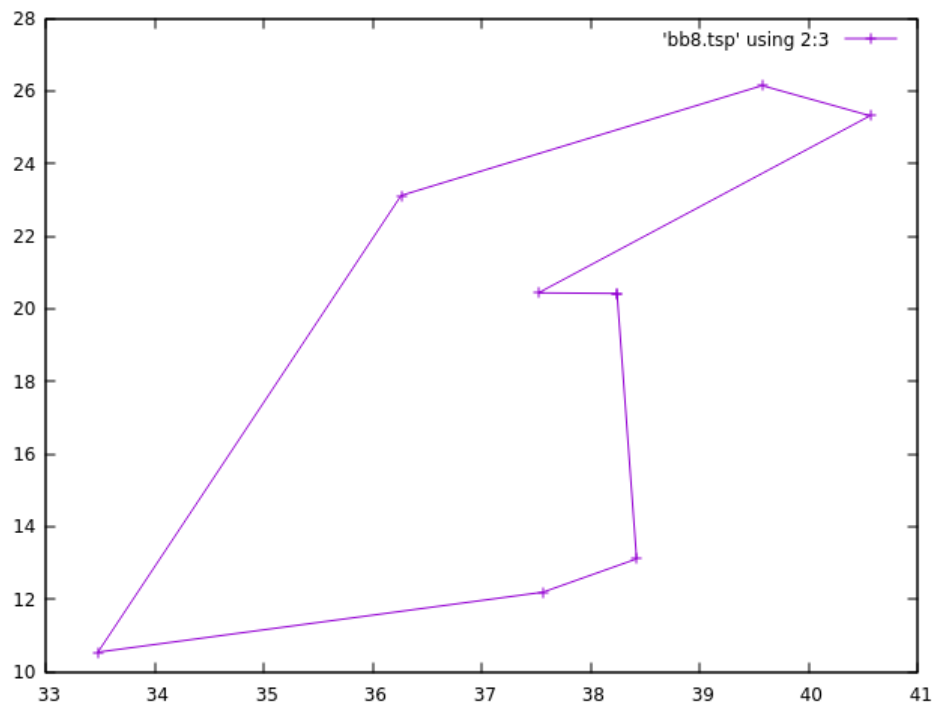


Figura 4: ulysses8.tsp

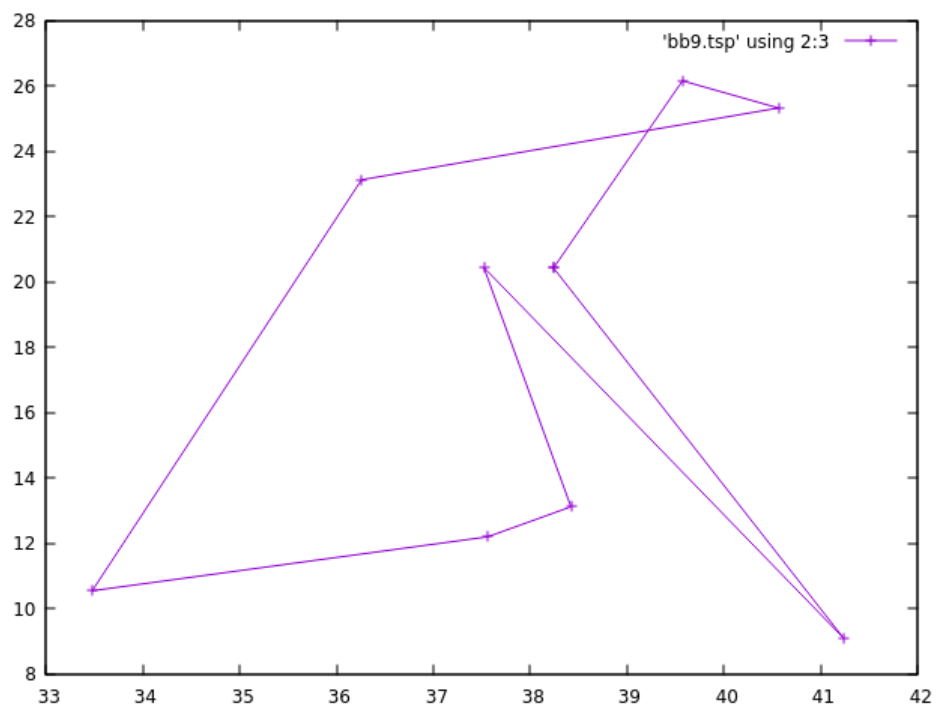


Figura 5: ulysses9.tsp

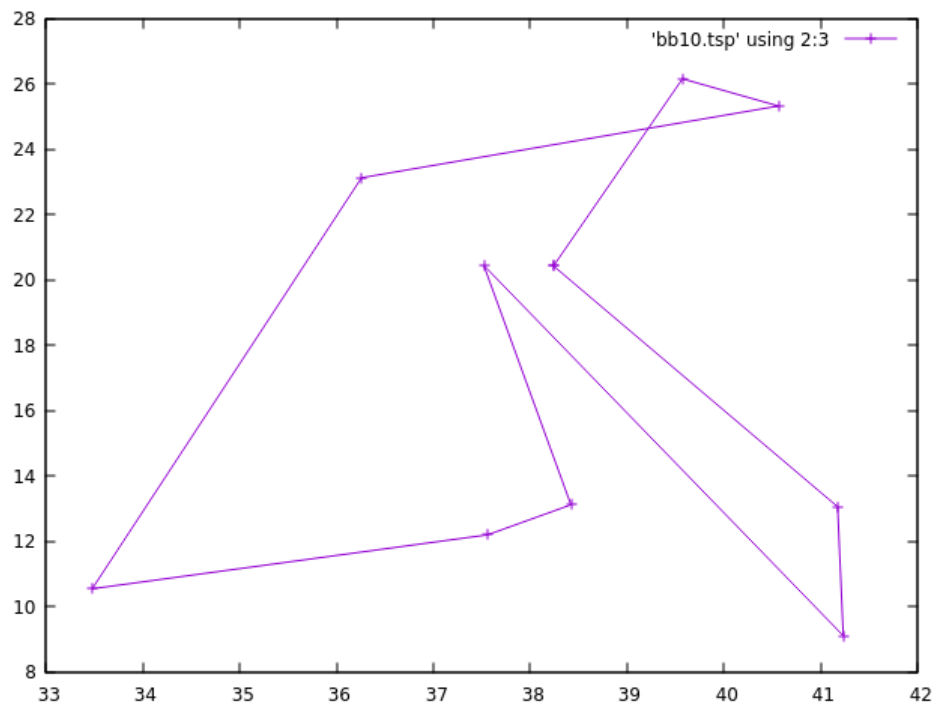


Figura 6: ulysses10.tsp

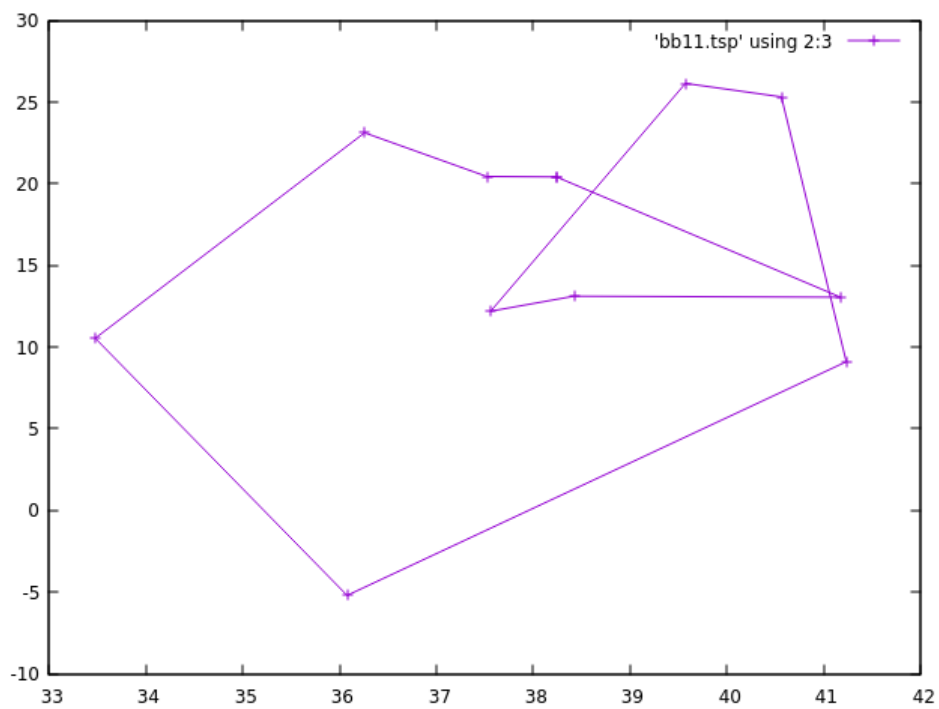


Figura 7: ulysses11.tsp

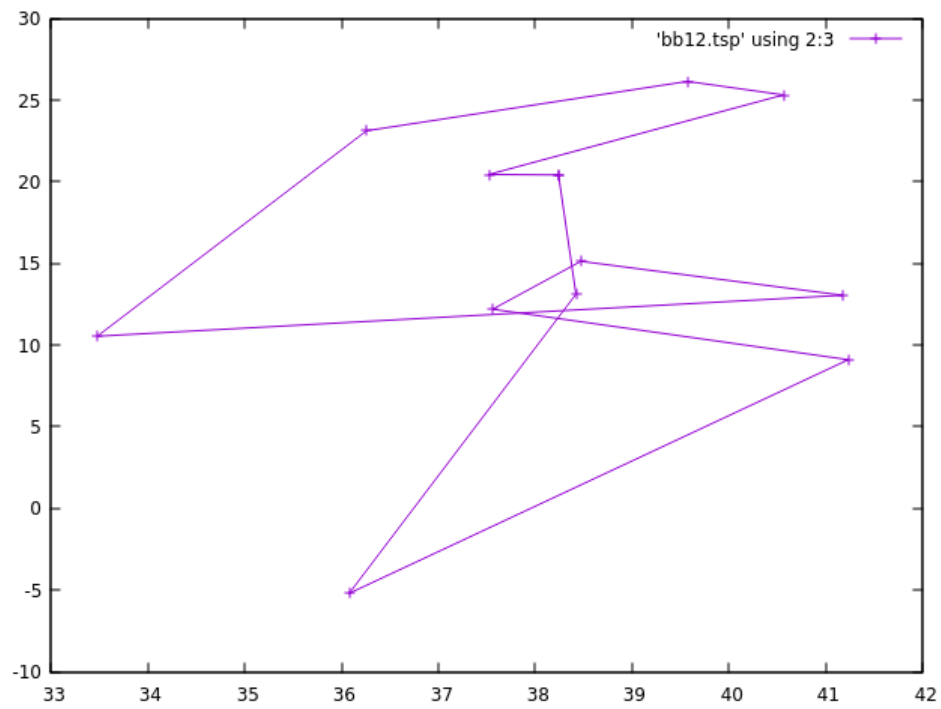


Figura 8: ulysses12.tsp

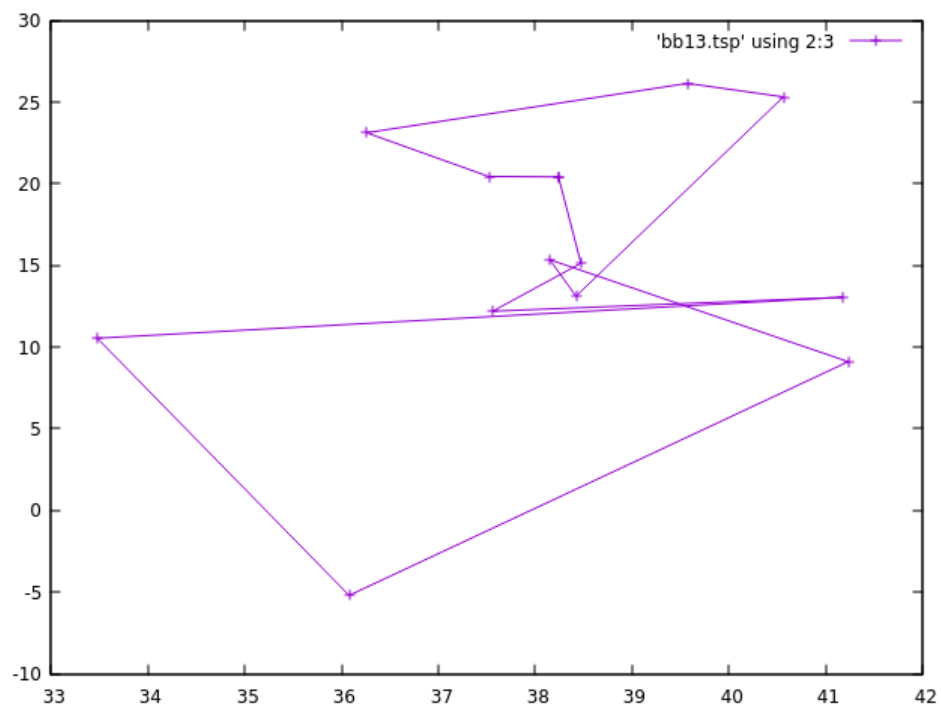


Figura 9: ulysses13.tsp



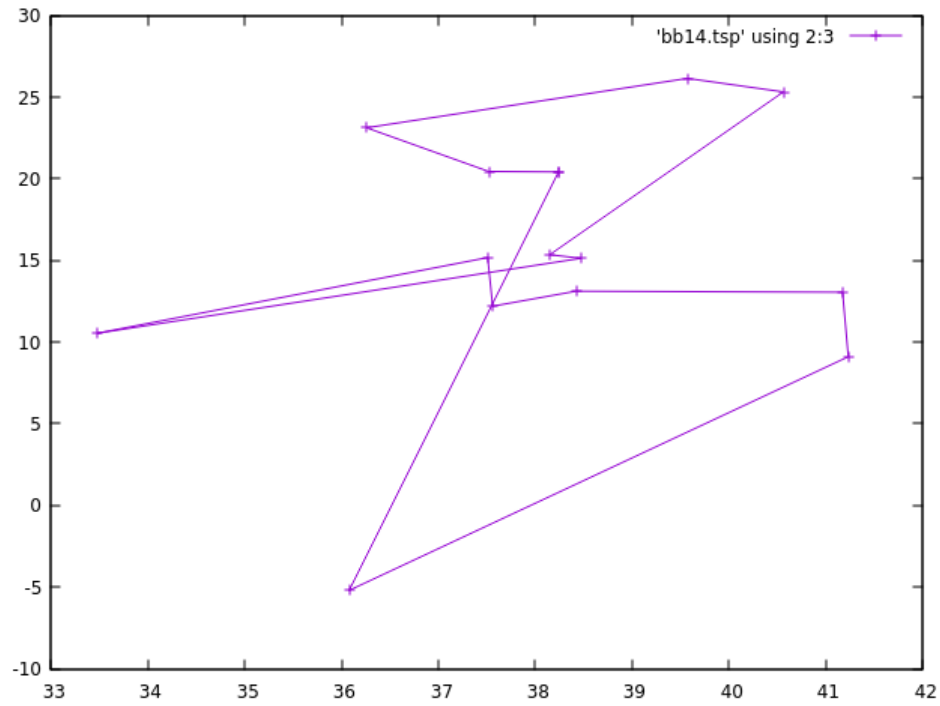


Figura 10: ulysses14.tsp

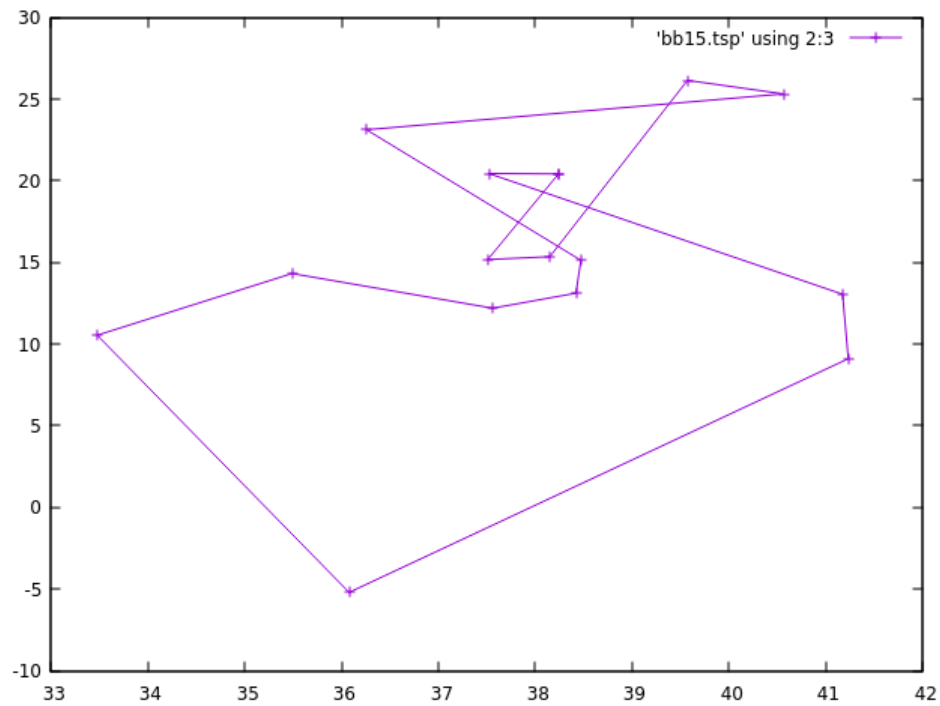


Figura 11: ulysses15.tsp

## 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

```
1  #include <chrono>
2  #include <climits>
3  #include <cmath>
4  #include <ctime>
5  #include <fstream>
6  #include <iostream>
7  #include <limits>
8  #include <list>
9  #include <string>
10 #include <vector>
11 using namespace std;
12 using namespace std::chrono;
13
14 #define DEBUG 0
15
16 struct ciudad {
17     int n;
18     double x;
19     double y;
20 };
21
22 bool operator==(const ciudad &una, const ciudad &otra) {
23     return una.x == otra.x && una.y == otra.y;
24 }
25 bool operator!=(const ciudad &una, const ciudad &otra) {
26     return !(una == otra);
27 }
28
29 ostream &operator<<(ostream &flujo, const vector<ciudad> &v) {
30     for (auto it = v.begin(); it != v.end() ; ++it) {
31         if (it != v.begin())
32             flujo << "->";
33         flujo << it->n + 1;
34     }
35     return flujo;
36 }
37 class TravelSalesman {
38
39 private:
40     const double MAX = numeric_limits<double>::max();
41     vector<ciudad> ciudades; // Almaceno problema
42     double distancia_total; // Distancia del circuito
43     vector<ciudad> camino; // Solución final.
44     vector<vector<double>> matriz_distancias;
45     vector<bool> visitados;
46     double calcularDistanciaCamino(const vector<ciudad> &path) const;
```

```

47     double distanciaEuclidea(const ciudad &una, const ciudad &otra) const;
48     void InicializarMatrizDistancias();
49     void Reservar(int n);
50     void ResetVisitados();
51     void RecBranchBound(int cota_actual, double peso_actual, int nivel,
52                         vector<ciudad> solucion_parcial);
53     double CalcularCotaInicial() const;
54
55     double MenorEntrante(const ciudad &ciudad) const;
56
57     double MenorSaliente(const ciudad &ciudad) const;
58
59 public:
60     TravelSalesman();
61     TravelSalesman(char *archivo);
62     int GetTamano() const;
63     void CargarDatos(char *archivo);
64     void imprimirResultado() const;
65     void Exportar(const char *name) const;
66
67     void BranchBound();
68 };
69
70 TravelSalesman::TravelSalesman() {
71     distancia_total = MAX;
72     ResetVisitados();
73 }
74
75 TravelSalesman::TravelSalesman(char *archivo) {
76     CargarDatos(archivo);
77     distancia_total = MAX;
78     ResetVisitados();
79 }
80
81 void TravelSalesman::imprimirResultado() const {
82     cout << endl << "Mejor solución:" << endl;
83     cout << camino << endl;
84     cout << "Distancia: " << distancia_total << endl;
85 }
86
87 void TravelSalesman::Exportar(const char *name) const {
88     ofstream salida;
89     salida.open(name);
90     if (salida.is_open()) {
91         salida << "DIMENSION: ";
92         salida << ciudades.size() << endl;
93         salida << "DISTANCIA: " << distancia_total << endl;
94         for (auto it = camino.begin(); it != camino.end(); ++it) {

```

```

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

```

```

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

```

```

191     cota /= 2;
192     return cota;
193 }
194
195 void TravelSalesman::BranchBound() {
196     double cota_inferior = CalcularCotaInicial();
197     vector<ciudad> solucion_parcial;
198     camino.resize(ciudades.size() + 1);
199     solucion_parcial.resize(ciudades.size() + 1);
200     ciudad primera = ciudades[0];
201     solucion_parcial.push_back(primeras); // Meto primera ciudad.
202     visitados[primera.n] = true;
203     RecBranchBound(cota_inferior, 0, 1, solucion_parcial);
204     camino.erase(camino.end()-1);
205 }
206
207 void TravelSalesman::RecBranchBound(int cota_actual, double peso_actual,
208                                     int nivel,
209                                     vector<ciudad> solucion_parcial) {
210     int n_primera = solucion_parcial[0].n;
211     int n_ultima = solucion_parcial[nivel - 1].n;
212     if (nivel == ciudades.size()) { // Caso base
213         double resultado_actual =
214             peso_actual + matriz_distancias[n_primera][n_ultima];
215         if (resultado_actual < distancia_total) {
216             distancia_total = resultado_actual;
217             camino = solucion_parcial;
218         }
219     } else { // Sigo expandiendo
220         for (auto it = ciudades.begin(); it != ciudades.end(); ++it) {
221             if (matriz_distancias[n_ultima][it->n] != 0 && !visitados[it->n]) {
222                 double temp = cota_actual; // Guardo cota actual
223                 peso_actual += matriz_distancias[n_ultima][it->n];
224                 if (nivel == 1)
225                     cota_actual -= (MenorEntrante(solucion_parcial[nivel - 1]) +
226                                     MenorEntrante(*it)) /
227                                     2;
228                 else
229                     cota_actual -= (MenorSaliente(solucion_parcial[nivel - 1]) +
230                                     MenorEntrante(*it)) /
231                                     2;
232                 double actual = cota_actual + peso_actual;
233                 if (actual < distancia_total) { // La solución puede mejorar
234                     solucion_parcial[nivel] = *it;
235                     visitados[it->n] = true;
236                     RecBranchBound(cota_actual, peso_actual, nivel + 1,
237                                     ↪ solucion_parcial);
237                 }

```

```

238         // Podemos
239         peso_actual -= matriz_distancias[it->n][n_ultima];
240         cota_actual = temp; // Restauro la cota.
241         ResetVisitados();
242         for (auto it = solucion_parcial.begin(); it != solucion_parcial.end();
243             ++it)
244             visitados[it->n] = true;
245     }
246 }
247 }
248 }
249 int main(int argc, char **argv) {
250
251     if (argc != 2) {
252         cerr << "Error de formato: " << argv[0] << " <fichero>." << endl;
253         exit(-1);
254     }
255     TravelSalesman instancia(argv[1]);
256
257     auto tantes = high_resolution_clock::now();
258
259     instancia.BranchBound();
260
261     auto tdespues = high_resolution_clock::now();
262
263     double tiempo = duration_cast<duration<double>>(tdespues - tantes).count();
264
265     cout << "Tamaño=" << instancia.GetTamano() << " Tiempo (s)=" << tiempo
266         << endl;
267
268     string nombre;
269     nombre = "bb";
270     nombre += to_string(instancia.GetTamano());
271     nombre += ".tsp";
272     instancia.Exportar(nombre.c_str());
273
274     #if DEBUG
275     instancia.imprimirResultado();
276     #endif
277
278     return 0;
279 }

```

Figura 12: TSP mediante Branch&Bound