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

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 tres heurísticas diferentes:

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

Todas las heurísticas desarrolladas tienen varias características en común:

- **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.

Por tanto, será la función de selección la que diferencie una de otra.

2. Vecino más cercano

- **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

- **Función de selección**. Seleccionamos la ciudad que incremente mínimamente la distancia total del circuito.

El conjunto de seleccionados comienza inicializado por el circuito formado por las ciudades más al norte, más al este y más al oeste.

4. Derivado de Kruskal

- **Función de selección**. Elegiremos aquella arista cuyo coste sea menor y cuyas ciudades no hayan sido visitadas aún.

Si el número de ciudades es impar, forzosamente tendremos que añadir la ciudad faltante al final, justo antes de cerrar el circuito.

5. Comparación de estrategias

ulysses16.tsp	
Vecino más cercano	77.1269
Inserción más económica	51.9733
Derivado de Kruskal	117.705
Solución más óptima	74.1087

gr96.tsp	
Vecino más cercano	603.302
Inserción más económica	620.367
Derivado de Kruskal	2026.4
Solución más óptima	512.309

a280.tsp	
Vecino más cercano	3094.28
Inserción más económica	3192.42
Derivado de Kruskal	6298.26
Solución más óptima	2586.77

att48.tsp	
Vecino más cercano	39236.9
Inserción más económica	40341.7
Derivado de Kruskal	101148
Solución más óptima	33523.7

pa561.tsp	
Vecino más cercano	18347
Inserción más económica	17688.4
Derivado de Kruskal	-
Solución más óptima	19330.8

lin105.tsp	
Vecino más cercano	16939.4
Inserción más económica	19063.7
Derivado de Kruskal	-
Solución más óptima	14383

pr76.tsp	
Vecino más cercano	130921
Inserción más económica	136636
Derivado de Kruskal	339435
Solución más óptima	108159

tsp225.tsp	
Vecino más cercano	4633.2
Inserción más económica	4734.51
Derivado de Kruskal	-
Solución más óptima	3859

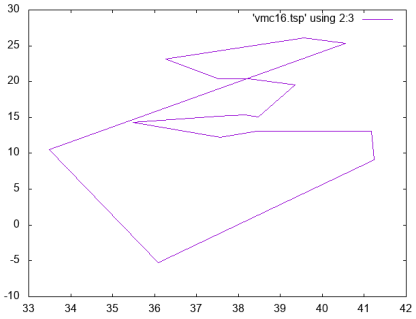
ch130.tsp	
Vecino más cercano	7198.74
Inserción más económica	7455.72
Derivado de Kruskal	27989.2
Solución más óptima	6110.86

berlin52.tsp	
Vecino más cercano	8182.19
Inserción más económica	9064.04
Derivado de Kruskal	18510.4
Solución más óptima	7544.37

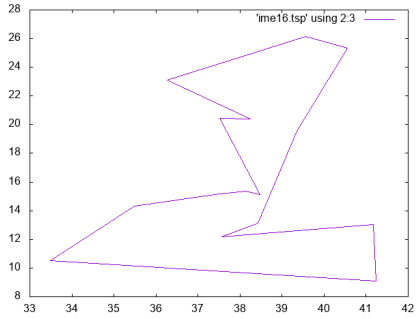
rd100.tsp	
Vecino más cercano	9427.33
Inserción más económica	9655.6
Derivado de Kruskal	32970.2
Solución más óptima	9724.75

st70.tsp	
Vecino más cercano	761.689
Inserción más económica	824.228
Derivado de Kruskal	2096.12
Solución más óptima	678.597

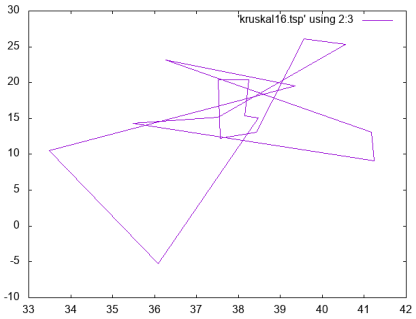
Figura 1: Comparación de resultados.



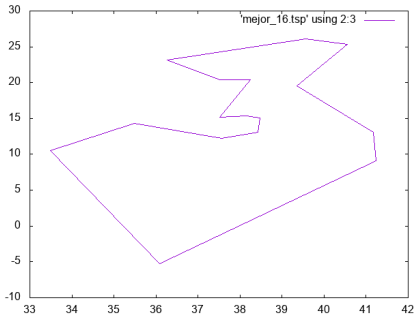
(a) Vecino más cercano



(b) Inserción más económica

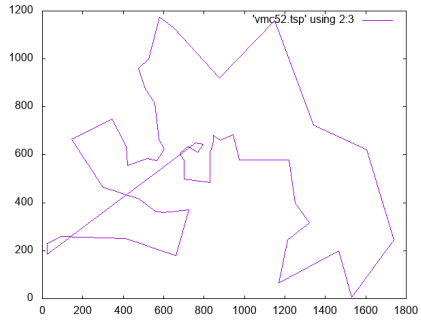


(c) Derivado de Kruskal

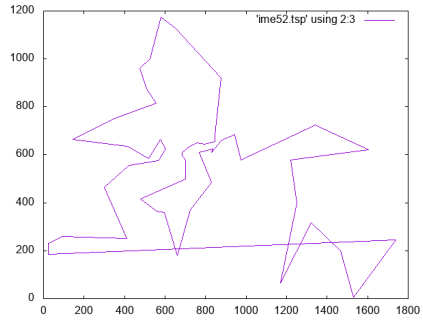


(d) Versión óptima

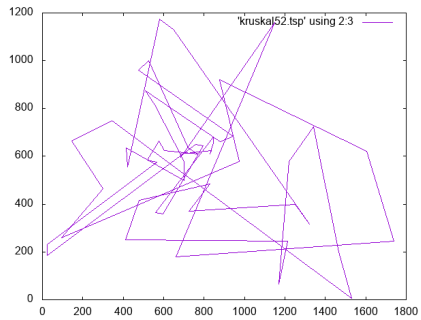
Figura 2: *ulysses16.tsp*



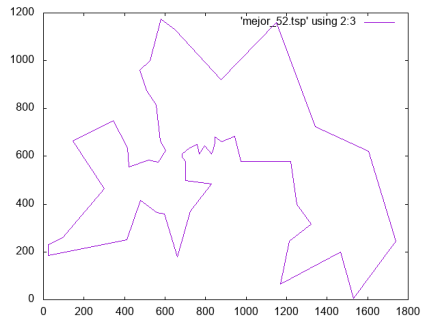
(a) Vecino más cercano



(b) Inserción más económica



(c) Derivado de Kruskal



(d) Versión óptima

Figura 3: *berlin52.tsp*

6. Anexo: código fuente

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

#define TEST 0

struct ciudad {
  int n;
  double x;
  double y;
  //int grado = 0;
};

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;
```

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

```
92      pair<vector<ciudad>, double> par;
93      par.first = resultado;
94      par.second = distancia;
95      return par;
96    }
97
98    void InicializarMatrizDistancias() {
99      for (int i = 0; i < ciudades.size(); i++)
100       for (int j = 0; j < ciudades.size(); j++)
101         matriz_distancias[i][j] = distanciaEuclidea(ciudades[i],
            ↪  ciudades[j]);
102   }
103
104   void Reservar(int n) {
105     visitados.resize(n);
106     matriz_distancias.resize(n);
107     for (int i = 0; i < n; i++)
108       matriz_distancias[i].resize(n);
109   }
110
111   void ResetVisitados() {
112     for (auto it = visitados.begin(); it != visitados.end(); ++it)
113       *it = false;
114   }
115
116   void Triangularizar(ciudad & norte, ciudad & este,
117                       ciudad & oeste) {
118
119     double x_max, x_min, y_max;
120     x_max = numeric_limits<double>::min();
121     y_max = numeric_limits<double>::min();
122     x_min = numeric_limits<double>::max();
123
124     for (int i = 0; i < ciudades.size(); i++) {
125       if (ciudades[i].x > x_max) {  //Más al este
126         x_max = ciudades[i].x;
127         este = ciudades[i];
128       }
129       if (ciudades[i].x < x_min) {  //Más al oeste
130         x_min = ciudades[i].x;
131         oeste = ciudades[i];
132       }
133       if (ciudades[i].y > y_max) {  //Más al norte
134         y_max = ciudades[i].y;
135         norte = ciudades[i];
136       }
137     }
138   }
```

```
139
140   void seleccionarNuevaCiudad() {
141     ciudad actual;
142     ciudad a_insertar;
143     int indice;
144     double dist_minima = numeric_limits<double>::max();
145     double d_aux;
146     for (int i = 0; i < ciudades.size(); i++) {    //Itero por todas las
      ↪  posibles ciudades
147       actual = ciudades[i];
148       if (!visitados[actual.n]) {        //Si no la he visitado
149         for (int j = 1; j < camino.size() - 1; j++) {    //Veo en que
          ↪  posición podría insertarla
150           vector<ciudad> aux = camino;
151           aux.insert(aux.begin() + j, actual);
152           d_aux = calcularDistanciaCamino(aux);
153           if (d_aux < dist_minima) {    //Me quedo con la que menos
            ↪  incrementa la distancia
154             dist_minima = d_aux;
155             a_insertar = actual;
156             indice = j;
157           }
158         }
159       }
160     }
161     camino.insert(camino.begin() + indice, a_insertar);
162     visitados[a_insertar.n] = true;
163   }
164   /*
165     pair< pair<ciudad, ciudad>, double> calularMinimo(pair< pair<ciudad,
    ↪  ciudad>, double> &min,
166             vector<vector <bool> > &matrizVisitados,
    ↪  vector<list<ciudad>> &conex, vector<bool> &aniadida) {
167       cout << "entra calcularminimo" << endl;
168
169       int x = 0;
170       int y = 0;
171       pair< pair<ciudad, ciudad>, double> resultado;
172       resultado.second = INFINITY;
173
174       for (int i = 0; i < matriz_distancias.size(); i++) {
175
176         for (int j = i + 1; j < matriz_distancias.size(); j++) {
177
178           if (ciudades[i].grado < 2 and ciudades[j].grado < 2) {
179
180             if (matriz_distancias[i][j] >= min.second and
181                     matriz_distancias[i][j] < resultado.second and
```

```
182                     matriz_distancias[i][j] != 0 and
183                     not matrizVisitados[i][j] and
184                     !hay_ciclo (conex, ciudades[i], ciudades[j],
    ↪ matrizVisitados)) {
185
186               resultado.second = matriz_distancias[i][j];
187               (resultado.first).first = ciudades[i];
188               (resultado.first).second = ciudades[j];
189               x = i;
190               y = j;
191             }
192           }
193         }
194       }
195
196
197       if (!conex[x].empty() and !aniadida[y] and not (conex[x].size() ==
    ↪ ciudades.size() || conex[y].size() == ciudades.size())) {
198
199         if (conex[y].size() > 1) {
200
201           for (auto it = conex[y].begin(); it != conex[y].end(); ++it)
    ↪ {
202             cout << "pb1" << endl;
203
204             conex[x].push_back(*it);
205           }
206
207         }
208         else {
209           cout << "pb2" << endl;
210
211           conex[x].push_back(ciudades[y]);
212         }
213
214         aniadida[y] = true;
215         conex[y].clear();
216       }
217       else if (!conex[x].empty() and aniadida[y] and not
    ↪ (conex[x].size() == ciudades.size() || conex[y].size() ==
    ↪ ciudades.size())) {
218
219         for (int k = 0; k < conex.size(); k++) {
220           for (auto it = conex[k].begin(); it != conex[k].end(); ++it)
    ↪ {
221
222             if (*it == ciudades[y]) {
223               if (conex[x].size() > 1) {
```

```
224
225                 for (auto it2 = conex[x].begin(); it2 != conex[x].end();
    ↪  ++it2) {
226                   cout << "pb3" << endl;
227
228                   conex[k].push_back(*it2);
229                 }
230               }
231               else {
232                 cout << "pb4" << endl;
233
234                 conex[k].push_back(ciudades[x]);
235               }
236               aniadida[x] = true;
237               conex[x].clear();
238             }
239           }
240         }
241       }
242       else if (conex[x].empty() and !aniadida[y] and not
    ↪  (conex[x].size() == ciudades.size() || conex[y].size() ==
    ↪  ciudades.size())) {
243
244         for (int k = 0; k < conex.size(); k++) {
245           for (auto it = conex[k].begin(); it != conex[k].end(); ++it)
    ↪  {
246
247             if (*it == ciudades[x]) {
248               if (conex[y].size() > 1) {
249
250                 for (auto it2 = conex[y].begin(); it2 != conex[y].end();
    ↪  ++it2) {
251                   cout << "pb5" << endl;
252                   conex[k].push_back(*it2);
253
254                 }
255               }
256               else {
257                 cout << "pb6" << endl;
258
259                 conex[k].push_back(ciudades[y]);
260
261               }
262               aniadida[y] = true;
263               conex[y].clear();
264             }
265           }
266         }
```

```
267         }
268 
269         matrizVisitados[x][y] = true;
270         ciudades[x].grado += 1;
271         ciudades[y].grado += 1;
272 
273         cout << "Fin calculaMin" << endl;
274 
275         return resultado;
276       }
277 
278       bool hay_ciclo(vector<list<ciudad>> &conex, const ciudad & primera,
    ↪  const ciudad & segunda, vector<vector <bool> > &matrizVisitados) {
279 
280         cout << "entra hay ciclo" << endl;
281         bool primera_enc = false;
282         bool hay = false;
283         ciudad restante;
284         bool primera_vez = true;
285 
286         for (int i = 0; i < conex.size(); i++) {
287           for (auto it = conex[i].begin(); it != conex[i].end() && !hay;
    ↪  ++it) {
288 
289             if (*it == primera || *it == segunda && primera_vez) {
290 
291               primera_enc = true;
292               primera_vez = false;
293 
294               if (*it == primera) {
295 
296                 restante = segunda;
297               }
298               else {
299 
300                 restante = primera;
301               }
302             }
303 
304             if (primera_enc && *it == restante) {
305 
306               hay = true;
307               matrizVisitados[primera.n][segunda.n] = true;
308             }
309           }
310 
311           primera_vez = true;
312         }
```

```
313         cout << "Fin hayCiclo" << endl;
314         return (hay);
315       }
316     */
317
318     pair<int, int> ArcoMenor() {
319       pair<int, int> resultado;
320       double d_minima = numeric_limits<double>::max();
321       for (int i = 0; i < ciudades.size(); i++) {
322         for (int j = 0; j < ciudades.size(); j++) {
323           if (i != j && !visitados[i] && !visitados[j]) {
324             if (matriz_distancias[i][j] < d_minima) {
325               d_minima = matriz_distancias[i][j];
326               resultado.first = i;
327               resultado.second = j;
328             }
329           }
330         }
331       }
332
333       visitados[resultado.first] = true;
334       visitados[resultado.second] = true;
335       return resultado;
336     }
337
338   public:
339     TSP() {
340       distancia_total = 0;
341       ResetVisitados();
342     }
343     TSP(char *archivo) {
344       CargarDatos(archivo);
345       distancia_total = 0;
346       ResetVisitados();
347     }
348     ~TSP() {
349
350     }
351
352     void DerivadoKruskal() {
353       camino.clear();
354       while (camino.size() != ciudades.size()) {
355         pair<int, int> arcoMenor = ArcoMenor();
356         camino.push_back(ciudades[arcoMenor.first]);
357         camino.push_back(ciudades[arcoMenor.second]);
358       }
359       for (int i = 0; i < ciudades.size(); i++)
360         if (!visitados[i]){
```

```
361           camino.push_back(ciudades[i]);    //Si queda alguna por visitar.
              ↪ Pasa cuando NCIUDADES es impar
362           visitados[i]=true;
363         }
364       camino.push_back(camino[0]);
365       distancia_total = calcularDistanciaCamino(camino);
366
367     }
368
369
370     void VecinoMasCercano() {
371       pair<vector<ciudad>, double> minimo, temp;
372       minimo = VecinoMasCercanoParcial(0);    //Calculo el vecino más
          ↪ cercano comenzando por el primero
373       for (int i = 0; i < ciudades.size(); i++) {
374         ResetVisitados();
375         temp = VecinoMasCercanoParcial(i);
376   #if TEST
377         cout << temp.first << endl;
378         cout << "Distancia " << temp.second << endl;
379   #endif
380         if (temp.second < minimo.second)
381           minimo = temp;
382       }
383       camino = minimo.first;
384       distancia_total = minimo.second;
385     }
386
387
388     int GetTamanio() {
389       return ciudades.size();
390     }
391
392
393     void CargarDatos(char *archivo) {
394       ifstream datos;
395       string s;
396       int n;
397       ciudad aux;
398       datos.open(archivo);
399       if (datos.is_open()) {
400         datos >> s;  //Leo DIMENSIÓN (cabecera)
401         datos >> n;  //Leo NÚMERO de ciudades y reservo espacio en
            ↪ matrices y vector.
402         Reservar(n);
403
404         for (int i = 0; i < n; i++) {
405           datos >> aux.n;  // Leo número de ciudad
```

```
406         aux.n--;      //Decremento el número: los índices del archivo
            ↪  comienzan en 1. Los del vector en 0.
407         datos >> aux.x >> aux.y; //Leo coordenadas
408         ciudades.push_back(aux);
409       }
410       datos.close();
411     }
412     else
413       cout << "Error al leer " << archivo << endl;
414
415     InicializarMatrizDistancias();
416   }
417
418   void imprimirResultado() {
419     cout << endl << "Mejor solución:" << endl;
420     cout << camino << endl;
421     cout << "Distancia: " << distancia_total << endl;
422   }
423
424   void Exportar(const char *name) {
425     ofstream salida;
426     salida.open(name);
427     if (salida.is_open()) {
428       salida << "DIMENSION: ";
429       salida << ciudades.size() << endl;
430       salida << "DISTANCIA: " << distancia_total << endl;
431       for (auto it = camino.begin(); it != camino.end(); ++it) {
432         salida << it->n + 1 << " " << it->x << " " << it->y << endl;
433       }
434       salida.close();
435     }
436     else
437       cout << "Error al exportar." << endl;
438   }
439
440   void InsercionMasEconomica() {
441     camino.clear();
442     ciudad norte, este, oeste;
443     //Hallo triángulo inicial
444     Triangularizar(norte, este, oeste);
445     //Añado al camino
446     camino.push_back(oeste);
447     camino.push_back(norte);
448     camino.push_back(este);
449     camino.push_back(oeste);
450
451     visitados[norte.n] = true;
452     visitados[este.n] = true;
```

```
453     visitados[oeste.n] = true;
454
455     cout << "Circuito Parcial:" << endl;
456     imprimirResultado();
457
458
459     //Voy eligiendo la siguiente
460     while (camino.size() != ciudades.size())    //Mientras no recorramos
        ↪  todas las ciudades
461       seleccionarNuevaCiudad();
462     //Añado el inicio
463     //camino.push_back(camino[0]);
464     distancia_total = calcularDistanciaCamino(camino);
465
466   }
467   /*
468     void algoritmo_de_laura_y_jose() {
469       vector< pair<ciudad, ciudad> > aristas;
470       pair< pair<ciudad, ciudad>, double> arista_minima;
471       ciudad ciudad_actual;
472       vector<vector <bool> > matrizVisitados (matriz_distancias.size(),
    ↪  vector<bool>(matriz_distancias.size(), false));
473       vector<list<ciudad>> ciudades_conex;
474       vector<bool> aniadida (ciudades.size(), false);
475
476       ciudades_conex.resize (ciudades.size());
477
478       //Cada ciudad está conexa consigo mismo
479       for (int in = 0; in < ciudades_conex.size(); in++)
480         ciudades_conex[in].push_back (ciudades[in]);
481
482       arista_minima.second = 0;
483
484       for (int i = 0; i < ciudades.size(); i++) {
485
486         arista_minima = calularMinimo(arista_minima, matrizVisitados,
    ↪  ciudades_conex, aniadida);
487         cout << "i= " << i << endl;
488         aristas.push_back(arista_minima.first);
489         distancia_total += arista_minima.second;
490       }
491
492       vector<bool> arista_leida (aristas.size(), false);
493
494       camino.push_back(aristas[0].first);
495       camino.push_back(aristas[0].second);
496       ciudad_actual = aristas[0].second;
497       arista_leida[0] = true;
```

```
498         for (auto it = matrizVisitados.begin(); it !=
    ↪  matrizVisitados.end(); ++it)
499           it->clear();
500         matrizVisitados.clear();
501 
502         for (auto it = aristas.begin(); it != aristas.end(); ++it)
503           cout << "(" << it->first.n << "," << it->second.n << ") ";
504         cout << endl;
505 
506         while (camino.size() < ciudades.size()) {
507 
508           for (int i = 1; i < aristas.size(); i++) {
509 
510             if (ciudad_actual == aristas[i].first && !arista_leida[i]) {
511 
512               ciudad_actual = aristas[i].second;
513               camino.push_back(ciudad_actual);
514               arista_leida[i] = true;
515             }
516             else if (ciudad_actual == aristas[i].second &&
    ↪  !arista_leida[i]) {
517 
518               ciudad_actual = aristas[i].first;
519               camino.push_back(ciudad_actual);
520               arista_leida[i] = true;
521             }
522           }
523         }
524       }
525     */
526 };
527 
528 
529 int main(int argc, char **argv) {
530 
531   if (argc != 2) {
532     cerr << "Error de formato: " << argv[0] << " <fichero>." << endl;
533     exit(-1);
534   }
535 
536   string nombre = "";
537 
538   /********** Vecino más cercano*******************/
539   cout << "**********************************************" << endl;
540 
541   TSP vecino_mas_cercano(argv[1]);
542   cout << "Heurística del Vecino más cercano." << endl;
543   vecino_mas_cercano.VecinoMasCercano();
```

```
544     vecino_mas_cercano.imprimirResultado();
545
546     nombre = "vmc";
547     nombre += to_string(vecino_mas_cercano.GetTamanio());
548     nombre += ".tsp";
549     vecino_mas_cercano.Exportar(nombre.c_str());
550     cout << "Exportado archivo " << nombre << endl;
551
552     cout << "*********************************************" << endl;
553
554
555     /*********** Inserción más económica******************/
556
557     cout << "*********************************************" << endl;
558
559
560     TSP insercion_mas_economica(argv[1]);
561
562     cout << "Heurística de la inserción más económica." << endl;
563     insercion_mas_economica.InsercionMasEconomica();
564     insercion_mas_economica.imprimirResultado();
565
566     nombre = "ime";
567     nombre += to_string(insercion_mas_economica.GetTamanio());
568     nombre += ".tsp";
569     insercion_mas_economica.Exportar(nombre.c_str());
570     cout << "Exportado archivo " << nombre << endl;
571
572     cout << "*********************************************" << endl;
573
574
575
576     /*********** Derivado de Kruskal******************/
577
578     cout << "*********************************************" << endl;
579
580
581     TSP derivado_kruskal(argv[1]);
582
583     cout << "Heurística derivada de Kruskal." << endl;
584     derivado_kruskal.DerivadoKruskal();
585     //derivado_kruskal.algoritmo_de_laura_y_jose();
586     derivado_kruskal.imprimirResultado();
587
588     nombre = "kruskal";
589     nombre += to_string(derivado_kruskal.GetTamanio());
590     nombre += ".tsp";
591     derivado_kruskal.Exportar(nombre.c_str());
```

```
592   cout << "Exportado archivo " << nombre << endl;
593
594   cout << "*****************************************" << endl;
595
596
597 }
```

Figura 4: 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 SegunOrden(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.SegunOrden(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 5: Programa que calcula la distancia del circuito a partir de un fichero con coordenadas y una lista ordenada de ciudades con sus respectivas coordenadas.

```
1 #!/bin/bash
2 
```

```
./calcular_distancia ../datosTSP/a280.tsp ../datosTSP/a280.opt.tour
./calcular_distancia ../datosTSP/att48.tsp ../datosTSP/att48.opt.tour
./calcular_distancia ../datosTSP/berlin52.tsp
↪  ../datosTSP/berlin52.opt.tour
./calcular_distancia ../datosTSP/ch130.tsp ../datosTSP/ch130.opt.tour
./calcular_distancia ../datosTSP/gr96.tsp ../datosTSP/gr96.opt.tour
./calcular_distancia ../datosTSP/lin105.tsp ../datosTSP/lin105.opt.tour
./calcular_distancia ../datosTSP/pa561.tsp ../datosTSP/pa561.opt.tour
./calcular_distancia ../datosTSP/st70.tsp ../datosTSP/st70.opt.tour
./calcular_distancia ../datosTSP/pr76.tsp ../datosTSP/pr76.opt.tour
./calcular_distancia ../datosTSP/rd100.tsp ../datosTSP/rd100.opt.tour
./calcular_distancia ../datosTSP/tsp225.tsp ../datosTSP/tsp225.opt.tour
./calcular_distancia ../datosTSP/ulysses16.tsp
↪  ../datosTSP/ulysses16.opt.tour
```

Figura 6: Script que genera archivos de coordenadas a partir de la mejor opción dada como lista.

```
#!/bin/bash

./tsp ../datosTSP/a280.tsp
./tsp ../datosTSP/att48.tsp
./tsp ../datosTSP/berlin52.tsp
./tsp ../datosTSP/ch130.tsp
./tsp ../datosTSP/gr96.tsp
#./tsp ../datosTSP/lin105.tsp
#./tsp ../datosTSP/pa561.tsp
./tsp ../datosTSP/st70.tsp
./tsp ../datosTSP/pr76.tsp
./tsp ../datosTSP/rd100.tsp
#./tsp ../datosTSP/tsp225.tsp
./tsp ../datosTSP/ulysses16.tsp
```

Figura 7: Script que genera archivos de coordenadas a partir de las tres heurísticas desarrolladas.

```
#!/usr/bin/gnuplot


set terminal png size 640,480

set output 'a280_mejor.png'
plot 'mejor_280.tsp' using 2:3 with lines

set output 'a280_vmc.png'
plot 'vmc280.tsp' using 2:3 with lines

set output 'a280_ime.png'
```

```
plot 'ime280.tsp' using 2:3 with lines

set output 'a280_kruskal.png'
plot 'kruskal280.tsp' using 2:3 with lines



set output 'att48_mejor.png'
plot 'mejor_48.tsp' using 2:3 with lines

set output 'att48_vmc.png'
plot 'vmc48.tsp' using 2:3 with lines


set output 'att48_ime.png'
plot 'ime48.tsp' using 2:3 with lines

set output 'att48_kruskal.png'
plot 'kruskal48.tsp' using 2:3 with lines



set output 'berlin52_mejor.png'
plot 'mejor_52.tsp' using 2:3 with lines

set output 'berlin52_vmc.png'
plot 'vmc52.tsp' using 2:3 with lines

set output 'berlin52_ime.png'
plot 'ime52.tsp' using 2:3 with lines

set output 'berlin52_kruskal.png'
plot 'kruskal52.tsp' using 2:3 with lines



set output 'ch130_mejor.png'
plot 'mejor_130.tsp' using 2:3 with lines

set output 'ch130_vmc.png'
plot 'vmc130.tsp' using 2:3 with lines

set output 'ch130_ime.png'
plot 'ime130.tsp' using 2:3 with lines

set output 'ch130_kruskal.png'
plot 'kruskal130.tsp' using 2:3 with lines

```

```
61
62
63  set output 'gr96_mejor.png'
64  plot 'mejor_96.tsp' using 2:3 with lines
65
66  set output 'gr96_vmc.png'
67  plot 'vmc96.tsp' using 2:3 with lines
68
69  set output 'gr96_ime.png'
70  plot 'ime96.tsp' using 2:3 with lines
71
72  set output 'gr96_kruskal.png'
73  plot 'kruskal96.tsp' using 2:3 with lines
74
75
76
77
78  #set output 'lin105_mejor.png'
79  #plot 'mejor_105.tsp' using 2:3 with lines
80
81  #set output 'lin105_vmc.png'
82  #plot 'vmc105.tsp' using 2:3 with lines
83
84  #set output 'lin105_ime.png'
85  #plot 'ime105.tsp' using 2:3 with lines
86
87  #set output 'lin105_kruskal.png'
88  #plot 'kruskal105.tsp' using 2:3 with lines
89
90
91
92
93  #set output 'pa561_mejor.png'
94  #plot 'mejor_561.tsp' using 2:3 with lines
95
96  #set output 'pa561_vmc.png'
97  #plot 'vmc561.tsp' using 2:3 with lines
98
99
100 #set output 'pa561_ime.png'
101 #plot 'ime561.tsp' using 2:3 with lines
102
103 #set output 'pa561_kruskal.png'
104 #plot 'kruskal561.tsp' using 2:3 with lines
105
106
107
108
```

```
109 set output 'st70_mejor.png'
110 plot 'mejor_70.tsp' using 2:3 with lines
111
112 set output 'st70_vmc.png'
113 plot 'vmc70.tsp' using 2:3 with lines
114
115 set output 'st70_ime.png'
116 plot 'ime70.tsp' using 2:3 with lines
117
118 set output 'st70_kruskal.png'
119 plot 'kruskal70.tsp' using 2:3 with lines
120
121
122
123 set output 'pr76_mejor.png'
124 plot 'mejor_76.tsp' using 2:3 with lines
125
126 set output 'pr76_vmc.png'
127 plot 'vmc76.tsp' using 2:3 with lines
128
129 set output 'pr76_ime.png'
130 plot 'ime76.tsp' using 2:3 with lines
131
132 set output 'pr76_kruskal.png'
133 plot 'kruskal76.tsp' using 2:3 with lines
134
135
136
137 set output 'rd100_mejor.png'
138 plot 'mejor_100.tsp' using 2:3 with lines
139
140 set output 'rd100_vmc.png'
141 plot 'vmc100.tsp' using 2:3 with lines
142
143 set output 'rd100_ime.png'
144 plot 'ime100.tsp' using 2:3 with lines
145
146 set output 'rd100_kruskal.png'
147 plot 'kruskal100.tsp' using 2:3 with lines
148
149
150
151 #set output 'tsp225_mejor.png'
152 #plot 'mejor_225.tsp' using 2:3 with lines
153
154 #set output 'tsp225_vmc.png'
155 #plot 'vmc225.tsp' using 2:3 with lines
156
```

```
#set output 'tsp225_ime.png'
#plot 'ime225.tsp' using 2:3 with lines

#set output 'tsp225_kruskal.png'
#plot 'kruskal225.tsp' using 2:3 with lines




set output 'ulysses16_mejor.png'
plot 'mejor_16.tsp' using 2:3 with lines

set output 'ulysses16_vmc.png'
plot 'vmc16.tsp' using 2:3 with lines

set output 'ulysses16_ime.png'
plot 'ime16.tsp' using 2:3 with lines

set output 'ulysses16_kruskal.png'
plot 'kruskal16.tsp' using 2:3 with lines
```

Figura 8: Script que genera las gráficas de los problemas a partir de archivos de coordenadas.

```
#!/bin/bash

echo "Ejecutando tres heurísticas..."
./ejecucion_tres_heuristicas.sh

echo "Ejecutando mejor opción..."
./ejecucion_mejor_opcion.sh

echo "Ejecutando gnuplot..."
./gnuplot.sh
```

Figura 9: Script genera todo lo necesario para la práctica.