## horizontal line



Práctica 3: Problema del Viajante de Comercio

2019

Realizado por:

Miguel Ángel Campos Cubillas

Alejandro Pinel Martínez

Guillermo Palomino Sánchez

Pablo Lombardero Ros

Nikita Stetskiy

# Resúmen

El problema del viajante de comercio se define como: dado un conjunto de ciudades y una matriz con las distancias entre todas ellas, un viajante debe recorrer todas las ciudades exáctamente una vez, regresando al punto de partida, de forma tal que la distancia recorrida sea mínima. Más formalmente, dado un grafo G, conexo y ponderado, se trata de hallar el ciclo hamiltoniano de mínimo peso de ese grafo.

# Algoritmo - Vecino Más Cercano

Aplicamos una resolución de tipo Greedy básica, buscaremos en cada instante la ciudad más cercana a la actual y la añadiremos al conjunto de soluciones al mismo tiempo que la eliminamos del conjunto de candidatos.

**Elementos del Algoritmo**

1. Conjunto de candidatos: Conjunto de ciudades. C
2. Conjunto de seleccionados: Conjunto de ciudades ordenado. S
3. Función solución: Conjunto de candidatos vacío.
4. Función de factibilidad: Cuando la ciudad no ha sido visitada
5. Función selección: Se selecciona la ciudad más cercana de las no escogidas.
6. Función objetivo: Lista con las ciudades ordenadas según el recorrido

**Pseudocódigo**

**Necesario**: Conjunto de ciudades C,S

x = 0

S = C[0]

**For** i = 1 **to** size(C) **do**

**If** C[i] MenorDist **then**

x = C[i]

S.add[x)

C[i].erase

**end if**

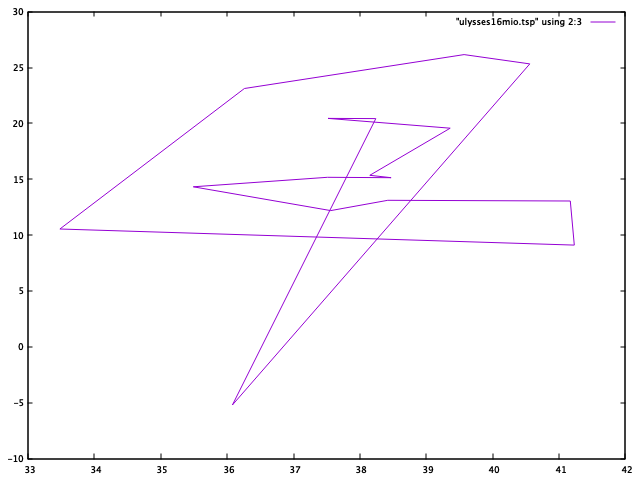
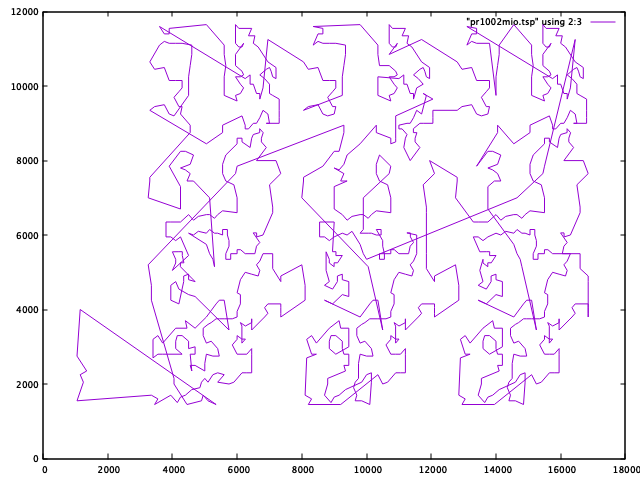
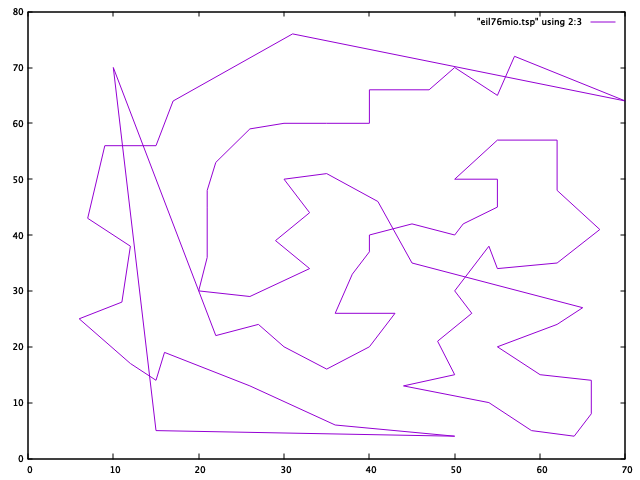
**end for**

**return** S

**Código**

|  |
| --- |
| #include <iostream>  #include <cstdlib>  #include <vector>  #include <fstream>  #include <cmath>  #include <string>  using namespace std;  // Struct Ciudad  // Representa una ciudad mediante:  // 1) Índice  // 2) Coordenadas x e y (posición en el plano)  struct Ciudad{  int indice;  double x; //Coordenadas  double y;  };  // Función leerCiudades  // Lee las ciudades desde un fichero  void leerCiudades(vector<Ciudad> &ciudades, const char fichero[]){  ifstream f(fichero);  double valor\_leido = 0.0;  int num\_ciudades = 0;  if(f){  f.ignore(11); // Ignoramos "DIMENSION: "  f >> num\_ciudades; // Dimension    for(int i = 0; i < num\_ciudades; i++){ // Cargamos el vector de ciudades con el índice  f >> valor\_leido;  ciudad.indice = valor\_leido;  f >> valor\_leido;  ciudad.x = valor\_leido;  f >> valor\_leido;  ciudad.y = valor\_leido;  ciudades.push\_back(ciudad);  }  }  f.close();  }  // Función calcularDistanciaTotal  // Calcula la distancia total de un conjunto de ciudades  double calcularDistanciaTotal(vector<Ciudad> ciudades){  double distancia = 0.0, x1 = 0.0, x2 = 0.0, y1 = 0.0, y2 = 0.0;  for(int i = 1; i < ciudades.size(); i++){  x1 = ciudades[i-1].x;  y1 = ciudades[i-1].y;  x2 = ciudades[i].x;  y2 = ciudades[i].y;  distancia += sqrt(pow(x1-x2, 2)+pow(y1-y2, 2));  }  return distancia;  }  // Función seleccionarCiudad  // Selecciona la siguiente ciudad solución  void seleccionarCiudad(vector<Ciudad> &ciudades, vector<Ciudad> &ciudades\_solucion){  Ciudad ciudad\_actual;  vector<Ciudad> v\_aux;  double min\_distancia = 99999999999;  int indice\_ciudad = 0, indice\_posicion\_insertar = 0;  //vector<Ciudad >::iterator it;    //Vamos cogiendo cada vez la ciudad más cercana dentro del conjunto de ciudades  for(int i = 0; i < ciudades.size(); i++){  ciudad\_actual = ciudades[i];  v\_aux = ciudades\_solucion;  v\_aux.push\_back(ciudad\_actual);  double d = calcularDistanciaTotal(v\_aux);  if(d < min\_distancia){  min\_distancia = d;  indice\_ciudad = i;    }  v\_aux.clear();      }  ciudades\_solucion.push\_back(ciudades[indice\_ciudad]);  ciudades.erase(ciudades.begin() + indice\_ciudad);  }  //Algoritmo Greedy para calcular una ruta  void calcularRuta(vector<Ciudad> &ciudades, vector<Ciudad> &ciudades\_solucion){  ciudades\_solucion.push\_back(ciudades.at(0));  ciudades.erase(ciudades.begin());  while(ciudades.size()){  //Vamos seleccionando las ciudades, dentro de la función también se va eliminando  seleccionarCiudad(ciudades,ciudades\_solucion);  }  }  //MAIN  int main(int argc, char\* argv[]){  vector<Ciudad> ciudades, ciudades\_solucion;  leerCiudades(ciudades, argv[1]);  calcularRuta(ciudades, ciudades\_solucion);  for(int i = 0; i < ciudades\_solucion.size(); i++)  cout << ciudades\_solucion[i].indice << " " << ciudades\_solucion[i].x << " " << ciudades\_solucion[i].y << endl;  //ciudades\_solucion.erase(ciudades\_solucion.end() - 1);  cout << endl << calcularDistanciaTotal(ciudades\_solucion) << endl;  return 0;  } |

A continuación se muestran los resultados obtenidos al usar la herramienta gnuplot para visualizar la solución al problema mediante nuestro algoritmo del vecino más cercano. Como podemos comprobar se trata de una estrategia greedy eficaz.



# Algoritmo - Inserción

En este algoritmo greedy, se sigue el procedimiento que se expone en la guía de la práctica, por lo que nos centraremos ahora en indicar cuáles son los elementos típicos de un algoritmo greedy presentes en nuestro algoritmo:

**Elementos del Algoritmo**

**- Conjunto de candidatos (C):** ciudades no seleccionadas.

**- Conjunto de candidatos seleccionados (S):** ciudades que han cumplido con el criterio y han sido insertadas en la solución.

**- Función solución (FS):** se trata del bucle externo while del método calcularSolucion(). Controla que se ha llegado a la solución, lo que ocurre cuando hay tantas ciudades en el conjunto solución como en el conjunto de candidatos.

**- Función de factibilidad (FF):** no es necesaria, pues el conjunto de candidatos siempre forma solución.

**- Función selección:** son los dos bucles for internos al while del método calcularSolucion(), que prueban todas las posibles combinaciones de ciudades y posiciones en el conjunto solución actual para determinar cual da una menor longitud de circuito resultante.

**- Función objetivo:** es el método mostrarSolucion() que, como su propio nombre indica, se encarga de mostrar el conjunto solución por pantalla.

Pasamos ahora a mostrar el código:

**main.cpp**

|  |
| --- |
| #include <cstdlib>  #include "Mapa.h"  **using** **namespace** std;  **int** main(**int** argc, **char**\*\* argv) {  Mapa mapa;    **if** (argc < 2){  cout << "Se necesita el nombre del fichero de ciudades como argumento\n";  **return** 1;  }    **if**(mapa.cargarDeFichero(argv[1]))  mapa.calcularSolucion();    mapa.mostrarSolucion();    **return** 0;  } |

**Mapa.h**

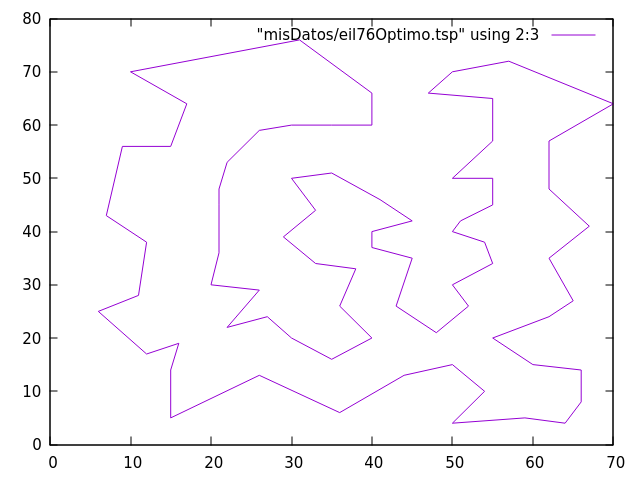
|  |
| --- |
| #include <cstdlib>  #include "Mapa.h"  **using** **namespace** std;  **int** main(**int** argc, **char**\*\* argv) {  Mapa mapa;    **if** (argc < 2){  cout << "Se necesita el nombre del fichero de ciudades como argumento\n";  **return** 1;  }    **if**(mapa.cargarDeFichero(argv[1]))  mapa.calcularSolucion();    mapa.mostrarSolucion();    **return** 0;  } |

**Mapa.cpp**

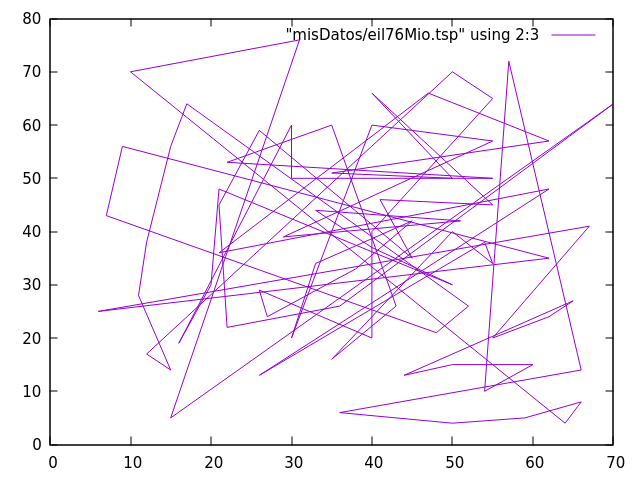
|  |
| --- |
| #include <fstream>  #include <cmath>  #include "Mapa.h"  **using** **namespace** std;  Mapa::Mapa() {  }  Mapa::Mapa(**const** Mapa& orig) {  }  Mapa::~Mapa() {  }  **void** Mapa::calcularSolucion(){  vector<**int**> tmp, vacio;  **int** max\_norte = ciudades[0].second, max\_este = ciudades[0].first, max\_oeste = ciudades[0].first, pos\_este, pos\_oeste, pos\_norte;  **int** dist\_min, dist\_act, min\_pos, lugar\_min, lugar\_aux, c; //min\_pos es la posicion de la solucion en el vector de ciudades y lugar\_min la posicion en el vector de soluciones que ocuapara el nuevo minimo.  **bool** stop;  //Busco e inserto la ciudad mas al este, mas al oeste y mas al norte  **for** (**int** j=0; j<ciudades.size(); j++){  **if**(ciudades[j].first > max\_este){  pos\_este = j;  max\_este = ciudades[j].first;  }**if**(ciudades[j].first < max\_oeste){  pos\_oeste = j;  max\_oeste = ciudades[j].second;  }**if**(ciudades[j].second > max\_norte){  pos\_norte = j;  max\_norte = ciudades[j].second;  }  }    solucion.push\_back(pos\_este);  solucion.push\_back(pos\_oeste);  solucion.push\_back(pos\_norte);  //Recorro el resto de ciudades  **while**(solucion.size() < ciudades.size()){  stop = **false**;  **for**(**int** a=0; a<ciudades.size() && !stop; a++){ //Inicializo dist\_min y min\_pos con la primera ciudad que no este incluida ya en solucion  **if**(!presenteEnSolucion(a)){  dist\_min = calcularLongitud(a, lugar\_min);  min\_pos = a;  stop = **true**;  }  }    **for**(**int** i=0; i<ciudades.size(); i++){ //Recorro el resto de ciudades  **if**(!presenteEnSolucion(i)){  dist\_act = calcularLongitud(i, lugar\_aux); //Para cada una de ellas calculo la longitud de camino que resultaria  **if**(dist\_act < dist\_min){  dist\_min = dist\_act;  min\_pos = i;  lugar\_min = lugar\_aux;  }  }  }  **for**(c=0; c<lugar\_min; c++) //Inserto la ciudad mas conveniente en la posicion mas conveniente  tmp.push\_back(solucion[c]);  tmp.push\_back(min\_pos);  **for**(**int** d=c; d<solucion.size(); d++)  tmp.push\_back(solucion[d]);  solucion = tmp;  tmp = vacio;  }  }  **int** Mapa::calcularLongitud(**int** pos, **int** lugar){  **int** aux = 0, min = 0;  //Inicializo min insertando el elemento en la primera posicion posible, 0:  min += calcularDistancia(pos, solucion[0]);  **for** (**int** t=0; t<solucion.size() - 1; t++){  min += calcularDistancia(solucion[t], solucion[t+1]);  }  min += calcularDistancia(solucion[solucion.size()-1], pos);  //Busco la mejor posicion en la sucesion de ciudades solucion en la que puedo insertar la ciudad pos  **for**(**int** i=0; i<solucion.size() - 1; i++){  aux = 0;  **for**(**int** j=0; j<i; j++)  aux += calcularDistancia(solucion[j], solucion[j+1]);  aux += calcularDistancia(solucion[i], pos) + calcularDistancia(pos, solucion[i + 1]);  **for**(**int** k=i + 1; k<solucion.size() - 1; k++)  aux += calcularDistancia(solucion[k], solucion[k+1]);  aux += calcularDistancia(solucion[solucion.size()-1], solucion[0]);  **if**(aux < min){  min = aux;  lugar = i+1;  }  }    **return** min;  }  **int** Mapa::calcularDistancia(**int** a, **int** b){  pair<**double**,**double**> ciudada = ciudades[a], ciudadb = ciudades[b];  **int** dist = sqrt((ciudadb.first-ciudada.first)\*(ciudadb.first-ciudada.first) + (ciudadb.second-ciudada.second)\*(ciudadb.second-ciudada.second));  **return** dist;  }  **bool** Mapa::presenteEnSolucion(**int** x){  **bool** presente = **false**;    **for** (**int** i=0; i<solucion.size() && !presente; i++)  **if**(solucion[i] == x)  presente = **true**;    **return** presente;  }  **void** Mapa::mostrarSolucion(){  cout << endl;  **for** (**int** i=0; i<solucion.size(); i++)  cout << solucion[i]+1 << "\n";  cout << endl;  }  **bool** Mapa::cargarDeFichero(**const** **char** \*fichero){  **bool** b = **true**;  **int** num, basura;  pair<**double**, **double**> aux;    ifstream fentrada;    fentrada.open(fichero);    **if** (fentrada){  fentrada.ignore(100, ':');  fentrada >> num;    **for**(**int** i=0; i<num; i++){  fentrada >> basura;  fentrada >> aux.first;  fentrada >> aux.second;  ciudades.push\_back(aux);  }    **if** (!fentrada){  cerr << "\nError en la lectura del fichero"<< endl;  b = **false**;  }  fentrada.close();  }**else**{  cerr << "\nError de apertura de fichero" <<endl;  b = **false**;  }    **return** b;  }  mapa.mostrarSolucion();    **return** 0;  **}** |

A continuación, se va a establecer una comparación entre el itinerario solución hallado por el algoritmo y el itinerario solución óptimo:

- Óptimo:



- Greedy:



Como se puede comprobar, este algoritmo greedy queda muy lejos de alcanzar la solución óptima. No obstante, es capaz de resolver estos problemas en relativamente poco tiempo, razón por la cual puede llegar a ser deseable en casos en los que algoritmos óptimos puedan tardar un tiempo inasumible en localizar la solución.

# Algoritmo - Optimización de la Colonia de Hormigas

El último algoritmo propuesto para resolver el problema del viajante de comercio es el de la Colonia de Hormigas, que si bien no es un algoritmo voraz, funciona de forma muy similar. Este algoritmo, como su propio nombre indica, se basa en el comportamiento de las hormigas.

Las hormigas cuando salen en busca de alimento forman un camino, apreciable a simple vista, que lleva hasta dicho alimento, y que con el paso del tiempo, se va transformando en un camino mínimo entre el hormiguero y su destino. Esto se consigue gracias al uso de feromonas. Cuando una hormiga sale de viaje, va dejando un rastro de feromonas que usará más adelante para volver al origen. Pero aunque es un sistema muy eficaz, hay que tener en cuenta que el rastro no es eterno, ya que las feromonas se evaporan, ni inmutable, ya que otra hormiga podría crear un camino completamente nuevo.

Ahora, supongamos que tenemos dos hormigas llamadas A y B y que el camino que ha encontrado una de ellas, A por ejemplo, es más corto que el que ha encontrado B. En este caso, como el camino de A es más corto, tardará menos tiempo en recorrer el camino que B, haciendo que la cantidad de feromonas que se encuentra en el camino de A en cierto momento concreto sea mayor que en el camino de B. Esto lo que provoca es que el camino de A se convierta en el camino predilecto ya que lo transitarán muchas más hormigas que, a su vez, aumentarán aún más la cantidad de feromonas, haciendo que el camino de B desaparezca. De este fenómeno surge el algoritmo de Optimización de la Colonia de Hormigas.

La implementación que hemos propuesto consiste de una función “*ColoniaHormigas*” que se encarga de computar la solución al problema que se nos propone. En dicha función tenemos el mapa que se nos ha propuesto resolver y una serie de hormigas que se situarán en ciudades aleatorias. Una vez inicializadas las hormigas, se crea un monto de posibilidades y con ello se genera un número aleatorio, este será nuestro componente aleatorio que es el que convierte a este algoritmo en uno voraz aleatorizado. Lo que conseguimos con este componente aleatorio es que se prueben varios caminos a la vez, ya que se usa para mandar a cada hormiga a una ciudad aleatoria del mapa. Estos recorridos quedan grabados en una matriz de feromonas que representa el grafo de ciudades. Una vez tenemos la matriz de feromonas solo queda comprobar cuál es el camino más óptimo desde la ciudad escogida.

Este es el fragmento de código obtenido:

|  |
| --- |
| void Mapa::ColoniaHormigas(int n\_iteraciones, int n\_hormigas) {  solucion.clear();  int nciudades = ciudades.size();  double \* \* feromonas = new double \* [nciudades];  for (int i = 0; i < nciudades; i++) {  feromonas[i] = new double[nciudades];  for (int j = 0; j < nciudades; j++)  feromonas[i][j] = 1;  }  bool \* ciudadesVisitadas = new bool[nciudades];  list<int> mejorsolucion;  int mejorcoste = 1e6;  for (int l = 0; l < n\_iteraciones; l++) {  list<int> \* camino = new list<int>[n\_hormigas];  for (int i = 0; i < n\_hormigas; i++) {  int costecamino = 0;  for (int j = 0; j < nciudades; j++)  ciudadesVisitadas[j] = false;  int ciudadanterior = rand()%nciudades;  camino[i].push\_back(ciudadanterior);  ciudadesVisitadas[ciudadanterior] = true;  /\*  //Empiezan todas en la ciudad 0  camino[i].push\_back(0);  ciudadesVisitadas[0] = true;  int ciudadanterior = 0;  \*/  for (int j = 1; j < nciudades; j++) {  int proximaciudad;  //////////////////////////////////////////  //Escoger la ciudad en base a las feromonas.  //Primero vamos a crear un monto de posibilidades  double probabilidades = 0;  for (int k = 0; k < nciudades; k++) {  if (!ciudadesVisitadas[k])  probabilidades += feromonas[ciudadanterior] [k];  }  //Generamos un double al azar  double f = (double)rand() / RAND\_MAX;  f = f \* probabilidades;  //Comprobamos de que ciudad es  for (int k = 0; k < nciudades && f > 0; k++) {  if (!ciudadesVisitadas[k])  f -= feromonas[ciudadanterior][k];  if (f <= 0)  proximaciudad = k;  }  //////////////////////////////////////////  ciudadesVisitadas[proximaciudad] = true;  int distanciarecorrida = calcularDistancia(ciudadanterior, proximaciudad);  camino[i].push\_back(proximaciudad);  costecamino += distanciarecorrida;  ciudadanterior = proximaciudad;  }  costecamino += calcularDistancia(camino[i].front(), camino[i].back());  if (costecamino < mejorcoste) {  mejorcoste = costecamino;  mejorsolucion = camino[i];  }  }  //Colocar feromonas en la matriz  for (int i = 0; i < n\_hormigas; i++) {  list<int>::iterator j = camino[i].begin();  int ciudadanterior = \*j;  int proximaciudad, distanciarecorrida;  j++;  while (j != camino[i].end()) {  proximaciudad = \*j;  distanciarecorrida = calcularDistancia(ciudadanterior, proximaciudad);  if (distanciarecorrida > 0)  feromonas[ciudadanterior][proximaciudad] += 1.0 / distanciarecorrida;  else  feromonas[ciudadanterior][proximaciudad] += 1.0;  ciudadanterior = proximaciudad;  j++;  }  }  delete [] camino;  }  //Al final copiamos el mejor camino encontrado  for (list<int>::iterator i = mejorsolucion.begin(); i != mejorsolucion.end(); i++) {  solucion.push\_back(\*i);  }  for (int i = 0; i < nciudades; i++)  delete [] feromonas[i];  delete [] feromonas;  delete [] ciudadesVisitadas;  } |

Este algoritmo consigue encontrar soluciones muy buenas al problema, pero es más exigente con el PC que los otros, ya que requiere realizar muchas iteraciones y dentro de cada iteración realiza bastantes recorridos, y eso se traduce en tiempos de computación bastante altos.

Los resultados obtenidos con este algoritmo son:

