Práctica 3.2: Problema del Viajante de Comercio (TSP)

Adrián Portillo Sánchez
Alejandro Durán Castro
Javier Labrat Rodríguez
Jose Antonio Martínez López
Jose Juán Pérez González

4 de mayo de 2015

Índice

1. Introducción

En esta práctica resolveremos el problema del viajante de comercio, en el cual se tiene un grafo de ciudades representadas como nodos en un mapa, y con el que se pretende encontrar un camino que recorra todas las ciudades del grafo sin repetir ninguna ciudad en el camino.
Las heurísticas que se plantearan no tratarán de encontrar la solución óptima, sino encontrar una solución que se aproxime lo suficiente, para ello se emplearan tres heurísticas, dos ya planteadas por el profesor, y una tercera que inventaremos nosotros, comenzamos por la heurística de la ciudad más cercana.

2. Marco de implementación de los algoritmos

Todos los algoritmos contarán con dos funciones comunes a los tres que serán leer_puntos (Que lee los archivos .tsp del problema y crea un vector con las coordenadas) y calcular_matriz_dimensiones (Que calcula las distancias entre todos los pares de puntos del problema):

2.1. leer_puntos()

```
void leer_puntos(const char * tsp, vector< pair<double, double> > & P){
    ifstream f(tsp);
    string s;
    pair<double,double> paraux;
    int n,act;

    if (f) {
        f >> s; // Leer "Dimension:"
        f >> n; // leer numero
        int i=0;
        while (i<n) {
            f >> act >> paraux.first >> paraux.second;
                P.push_back(paraux);
                i++;
        }
    } else {
      cout << "Error de Lectura" << endl;
    }
    f.close();
 }
```

2.2. calcular_matriz_dimension()

```
void calcular_matriz_dimension(const vector< pair<double, double> > & P,
                               vector< vector<int> > & D){
    for(int i=0; i<D.size(); ++i)
        for(int j=0; j<D.size(); ++j){
            D[i][j] = round( sqrt( pow( (P[j].first) - (P[i].first), 2)
                                 + pow( (P[j].second) - (P[i].second), 2) ) );
        }}
```

3. Algoritmo de la Ciudad Más Cercana

Este algoritmo crea un camino partiendo de un nodo, y a partir de él eligiendo el nodo no recorrido más cercano hasta llegar al nodo inicial de nuevo.
Luego continúa realizando lo mismo comenzando por otro nodo, y así con todos los nodos del problema. Una vez realizado un camino a partir de cada nodo, se comparan las soluciones obtenidas y se elige la mejor.

```
#include <iostream>
#include <vector>
#include <map>
#include <string>
#include <algorithm>
#include <cmath> // sqrt, round
#include <iostream>
#include <fstream>
#include <sstream>
using namespace std;

void leer_puntos()
void calcular_matriz_dimension()

int calcular_solucion(const vector< vector<int> > & D, vector<int> & solucion){
    //distancia final
    int resultado = 0;

    for(int inicial = 0; inicial < D.size(); inicial ++){
        vector<int> seleccionados;
        seleccionados.push_back(inicial);
        int siguiente = inicial;
        int total_coste = 0;

        //calculamos la solucion desde un nodo
        while(seleccionados.size()!=D.size()){
            int coste = 0;
            int nodo_seleccionado;

            //buscar siguiente nodo
            for(int i = 0; i<D.size(); ++i){
                //buscar i en el vector seleccionados si no esta
                vector<int>::iterator it;
                it = find (seleccionados.begin(), seleccionados.end(), i);
                if (it == seleccionados.end()){ //elemento no seleccionado
                    if(coste==0){ //primero en seleccionar
                        coste = D[siguiente][i];
                        nodo_seleccionado = i;
                    } else if(coste>D[siguiente][i]) { //comparar con otro seleccionado
                        coste = D[siguiente][i];
                        nodo_seleccionado = i;
                    }
                }
```

```
            }

            total_coste += coste;
            seleccionados.push_back(nodo_seleccionado);
            siguiente=nodo_seleccionado;
        } //while

        //ahora comparamos con la solucion final escogida anteriormente
        if(resultado==0){ //primera solucion
            resultado=total_coste;
            solucion = seleccionados;
        } else if(total_coste<resultado){ //si la nueva solucion es mejor se cambia
            resultado=total_coste;
            solucion = seleccionados;
        }

    } //for
    return resultado;
}

int main(int argc, char * argv[])
{
    vector< pair<double, double> >  P; //vector con puntos
    vector< vector<int> > D;//matriz distancias
    vector<int> solucion;
    int distancia;

    if (argc != 2) {
        cout << "Error Formato" << endl;
        exit(1);
    }

    leer_puntos(argv[1],P);

    D.resize(P.size(), vector<int>(P.size()) );

    calcular_matriz_dimension(P,D);
    distancia=calcular_solucion(D,solucion);

    cout << "La distancia obtenida es: " << distancia << endl;
    cout << "Dimension: " << solucion.size() << endl;
    for (int i=0;i<solucion.size(); i++){
        cout << solucion[i]+1 << endl;

    }
    return 0;
}
```

4. Algoritmo de la Inserción Más Barata

Esta algoritmo compara todas las distancias entre nodos y de ellas elige la menor de las distancias no tomadas y que no formen un bucle, dando como resultado una aproximación bastante buena que tomaré la mayoría de caminos menores del problema.

```
#include <map>
#include <fstream>
#include <iostream>
#include <vector>
#include <math.h>
#include <algorithm>

using namespace std;

const int MENOS_INF = -2147483648;
const int MAS_INF = 2147483647;

void leer_puntos()
void calcular_matriz_dimensiones()

int calcular_solucion(const vector< pair<double, double> > & C,
                      const vector< vector<int> > & MD, vector<int> & S) {

        int nodo_este = -1, nodo_oeste = -1, nodo_norte = -1;
        int distancia_act;
        vector<int> seleccionados;

        double este = MENOS_INF, oeste = MAS_INF, norte = MENOS_INF;
        for (int i = 0; i < C.size(); i++)
        {
                if (C[i].first > este)
                {
                        nodo_este = i;
                        este = C[i].first;
                }

                else if (C[i].first < oeste)
                {
                        nodo_oeste = i;
                        oeste = C[i].first;
                }

                if (C[i].second > norte)
                {
                        nodo_norte = i;
                        norte = C[i].second;
                }
        }
```

```
        distancia_act = MD[nodo_este][nodo_oeste]
        + MD[nodo_oeste][nodo_norte] + MD[nodo_norte][nodo_este];
        seleccionados.push_back(nodo_este);
        seleccionados.push_back(nodo_oeste);
        seleccionados.push_back(nodo_norte);

        for (int i = 0; i < C.size(); i++)
        {
                vector<int>::iterator it=find(seleccionados.begin(),seleccionados.end(),i);
                if (it == seleccionados.end())
                {
                        int dif_distancias = MAS_INF;
                        vector<int>::iterator posicion = seleccionados.begin();

                        it = seleccionados.begin();
                        for (int s = 0; s < seleccionados.size()-1; s++, ++it)
                        {
                                int dd_aux = (MD[i][*it] + MD[i][seleccionados[s+1]])
                                                        - MD[*it][seleccionados[s+1]];
                                if (dd_aux < dif_distancias)
                                {
                                        dif_distancias = dd_aux;
                                        posicion = it; ++posicion;
                                }
                        }
                                int dd_aux= (MD[i][seleccionados[seleccionados.size()-1]]
                                           + MD[i][0])
                                           - MD[seleccionados[seleccionados.size()-1]][0];
                                if (dd_aux < dif_distancias)
                                {
                                        dif_distancias = dd_aux;
                                        posicion = seleccionados.begin();
                                }

                        distancia_act += dif_distancias;
                        seleccionados.insert(posicion, i);
                }
        }

        S = seleccionados;
        return distancia_act;
}
```

```
int main(int argc, char ** argv)
{
        vector< pair<double, double> > Ciudades;
        vector< vector<int> > Matriz_Dimensiones;
        vector<int> Solucion;
        int distancia;

        if (argc != 2) {
                cout << "Faltan argumentos." << endl;
                exit(1);
        }

        leer_puntos(argv[1], Ciudades);Ciudades.size(), vector<int>(Ciudades.size()));

        calcular_matriz_dimensiones(Ciudades, Matriz_Dimensiones);

        distancia = calcular_solucion(Ciudades, Matriz_Dimensiones, Solucion);

        cout << "La distancia resultante es: " << distancia << endl;
        cout << "Dimension: " << Solucion.size() << endl;
        for (int i = 0; i < Solucion.size(); i++)
                cout << Solucion[i]+1 << endl;
}
```

5. Algoritmo de la Heurística Propia

En esta ultima heurística se toma la solución de la primera heurística y se realizan inversiones entre los caminos, viendo si la distancia resultante de intercambiar el camino entre 2 nodos por el de otros dos nodos da mejores resultados de la siguiente forma, y así encontrar una solución mejor que la de la primera heurística.
Este algoritmo realizará la prueba para todas las combinaciones de nodos posibles, por lo que tal vez tarde un poco en encontrar los resultados, pero las soluciones obtenidas son notablemente mejores, por lo que es una heurística que puede merecer la pena en algunas situaciones.

```
– A    B –              – A – B –
    X          ==>
– C    D –              – C – D –
```

```
#include <iostream>
#include <vector>
#include <map>
#include <string>
#include <algorithm>
#include <cmath> // sqrt, round
#include <iostream>
#include <fstream>
#include <sstream>
#include<list>

#define SIGUIENTE(a,c) {a++; if(a == c.end()) a = c.begin();}

using namespace std;

typedef list<int>::iterator Lit;

void leer_puntos()
void calcular_matriz_dimension()
int calcular_solucion_parcial() //Heuristica de la ciudad mas cercana

void resolverLista (const vector< vector<int> > & D, list<int>& camino){
    // El algoritmo continuara hasta que no haya producido ningun cambio
    bool continuar = true;
    int coste=0;
    while (continuar){
        continuar = false;
        Lit actual = camino.begin();
        Lit siguiente = actual; ++siguiente;
        Lit possiguiente = siguiente; ++possiguiente;


        while(possiguiente != camino.end() && siguiente != camino.end()){
            // Fija una pareja de puntos con 'actual' y 'siguiente'
            // Fija una pareja siguiente con 'otro' y 'sig_otro'
            // Probara a intercambiar cada una de estas parejas
            Lit otro = possiguiente;
```

```
            Lit sig_otro = otro; ++sig_otro;
            bool hay_cambios = false;

            // Recorre el camino probando intercambios
            while(otro != camino.end() && !hay_cambios){
                if (sig_otro == camino.end()) sig_otro = camino.begin();

                // Realiza el intercambio si la distancia mejora con el cambio.
                double distancia_actual = D[*actual][*siguiente]
                    + D[*otro][*sig_otro];
                double distancia_nueva = D[*actual][*otro]
                    + D[*siguiente][*sig_otro];

                if (distancia_nueva < distancia_actual){
                    // Rota la zona intermedia entre los puntos
                    list<int> corte;
                    corte.splice(corte.begin(), camino, siguiente, otro);
                    corte.reverse();
                    camino.splice(sig_otro, corte, corte.begin(), corte.end());

                    // Recoloca los iteradores
                    siguiente = actual; SIGUIENTE(siguiente, camino);
                    otro = sig_otro; --otro;
                    hay_cambios = true;
                    continuar = true;
                }
                ++otro;
                ++sig_otro;
            }

            ++actual;
            ++siguiente;
            possiguiente = siguiente; ++possiguiente;
        }
    }
}

int resolver(const vector< vector<int> > & D, int tama,
             const vector<int>& solucion_parcial, vector<int> & solucion){
    // Trabajaremos el camino en una lista enlazada
    // Asi hacemos que sea menos costoso computacionalmente rotar partes del camino
    list<int> camino;
    for (int i=0; i<tama; i++)
        camino.push_back(solucion_parcial[i]);

    // Resolucion
    int resultado = resolverLista(D, camino);

    // Devuelve el camino como un array normal
    vector<int> seleccionados;
    seleccionados.reserve(tama);
```

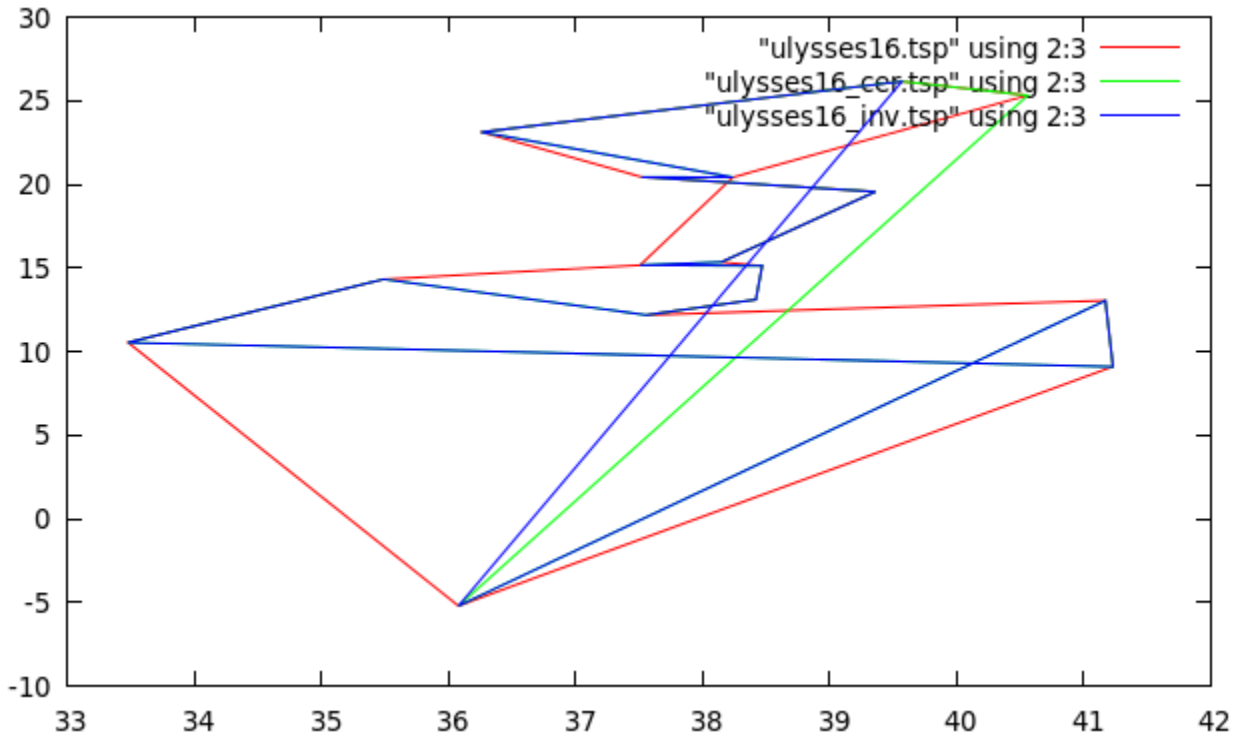
```
    int coste = 0;
    Lit i = camino.begin(); i++;
    for (Lit j = camino.begin(); i != camino.end(); i++){
        seleccionados.push_back(*i);
        coste += D[*j][*i];
        j = i;
    }
        solucion = seleccionados;
        return coste;
}

int main(int argc, char * argv[])
{
    vector< pair<double, double> >  P; //vector con puntos
    vector< vector<int> > D;//matriz distancias
    vector<int> solucion_parcial;
    vector<int> solucion;
    int distancia;

    if (argc != 2) {
        cout << "Error Formato" << endl;
        exit(1);
    }

    leer_puntos(argv[1],P);

    D.resize(P.size(), vector<int>(P.size()) );

    calcular_matriz_dimension(P,D);

    calcular_solucion_parcial(D,solucion_parcial);
    distancia = resolver(D,P.size(),solucion_parcial,solucion);

    cout << "La distancia obtenida es: " << distancia << endl;
    cout << "Dimension: " << solucion.size() << endl;
    for (int i=0;i<solucion.size(); i++){
        cout << solucion[i]+1 << endl;

    }
    return 0;
}
```

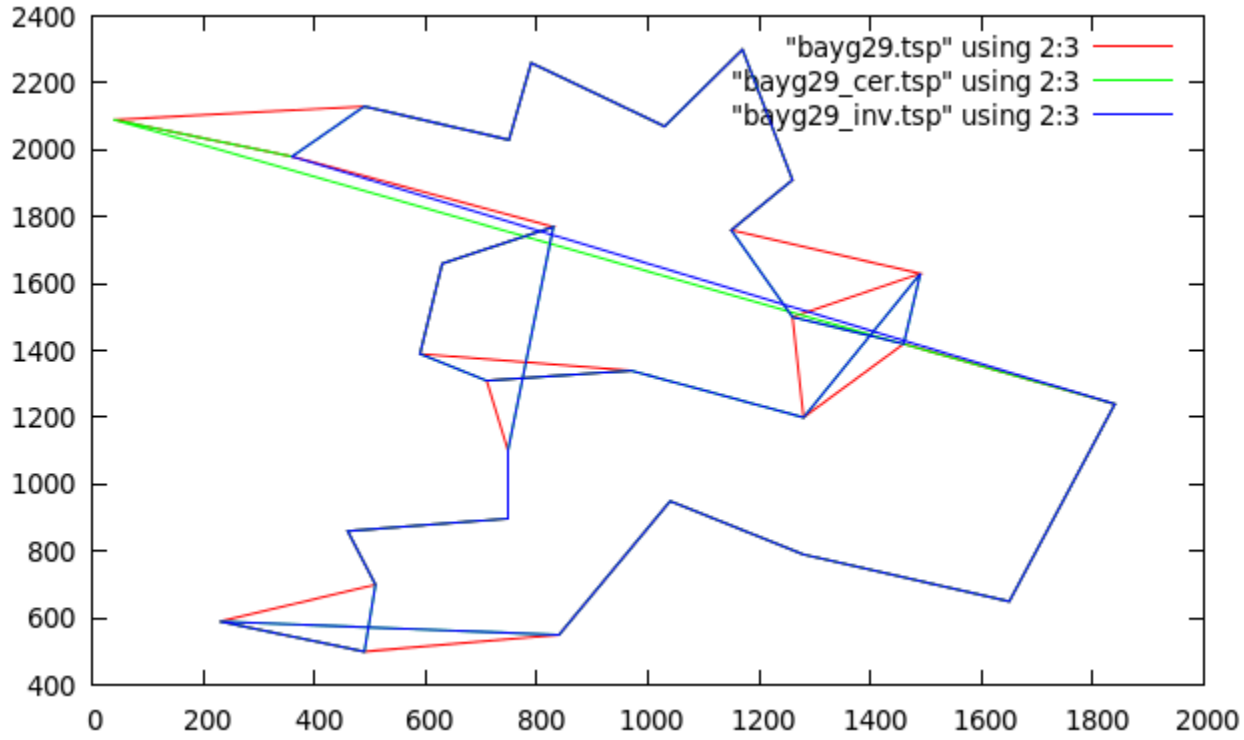
6. Comparativa de los Resultados obtenidos.

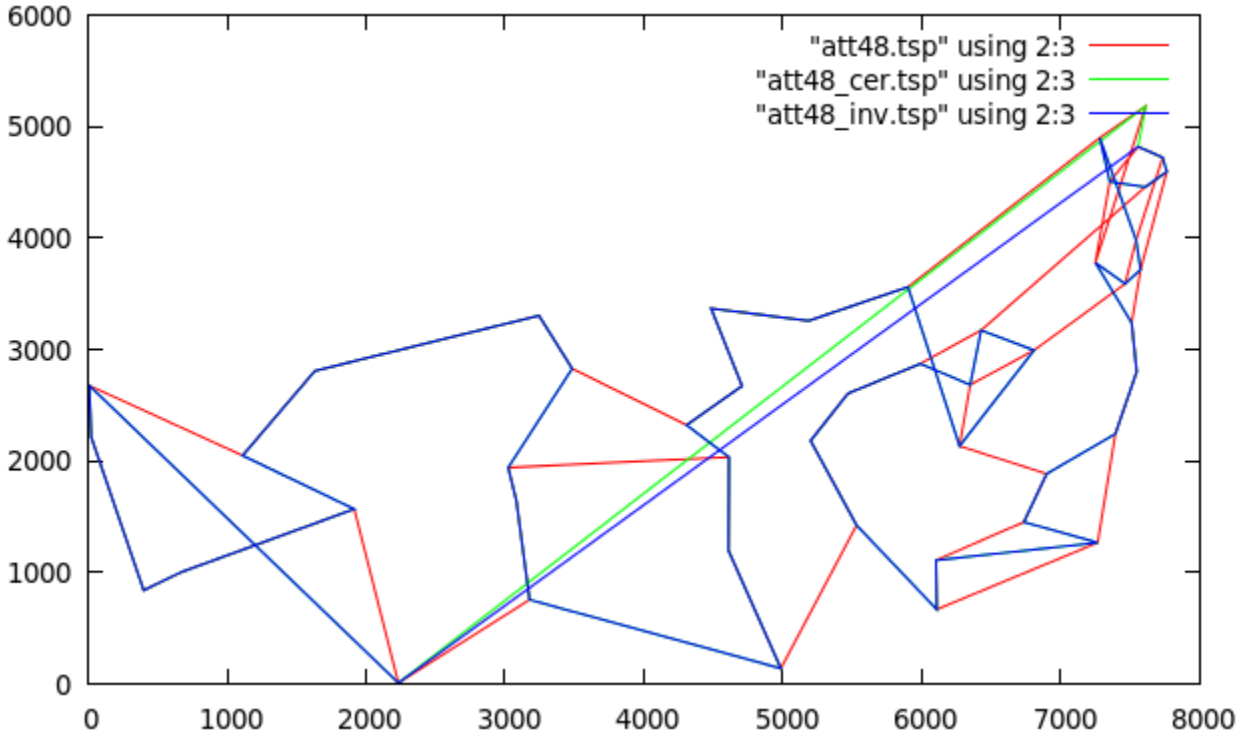
6.1. Resultados obtenidos a partir de los algoritmos

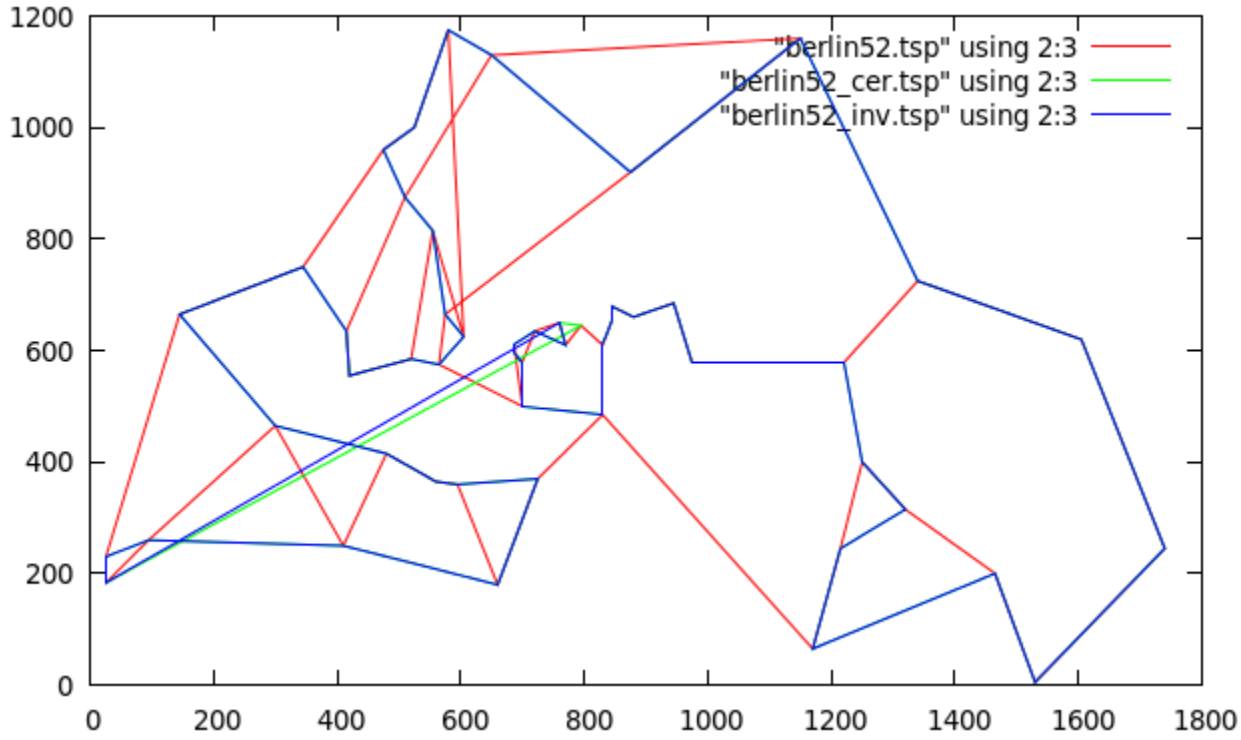
En primer lugar, mostraremos los resultados obtenidos para algunos de los problemas de TSP aportados por el profesor, de forma gráfica, para que se puedan apreciar las diferencias entre los algoritmos.

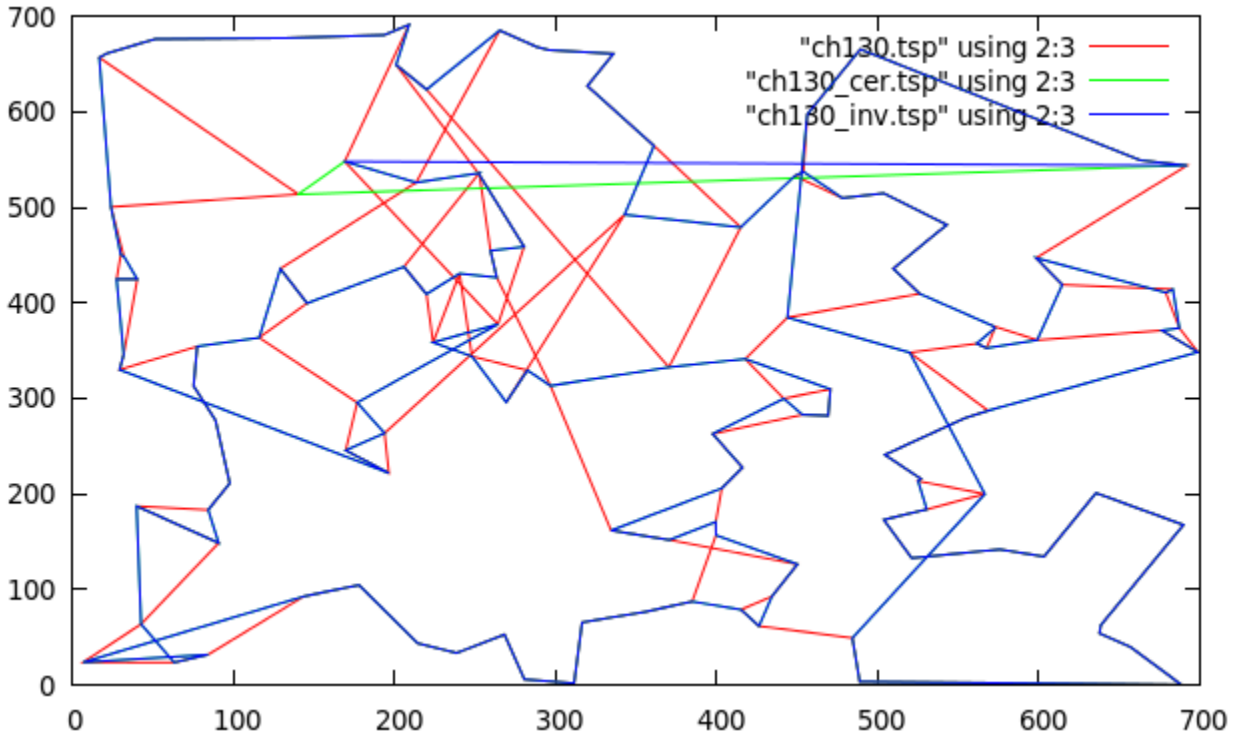
Índice de gráficos

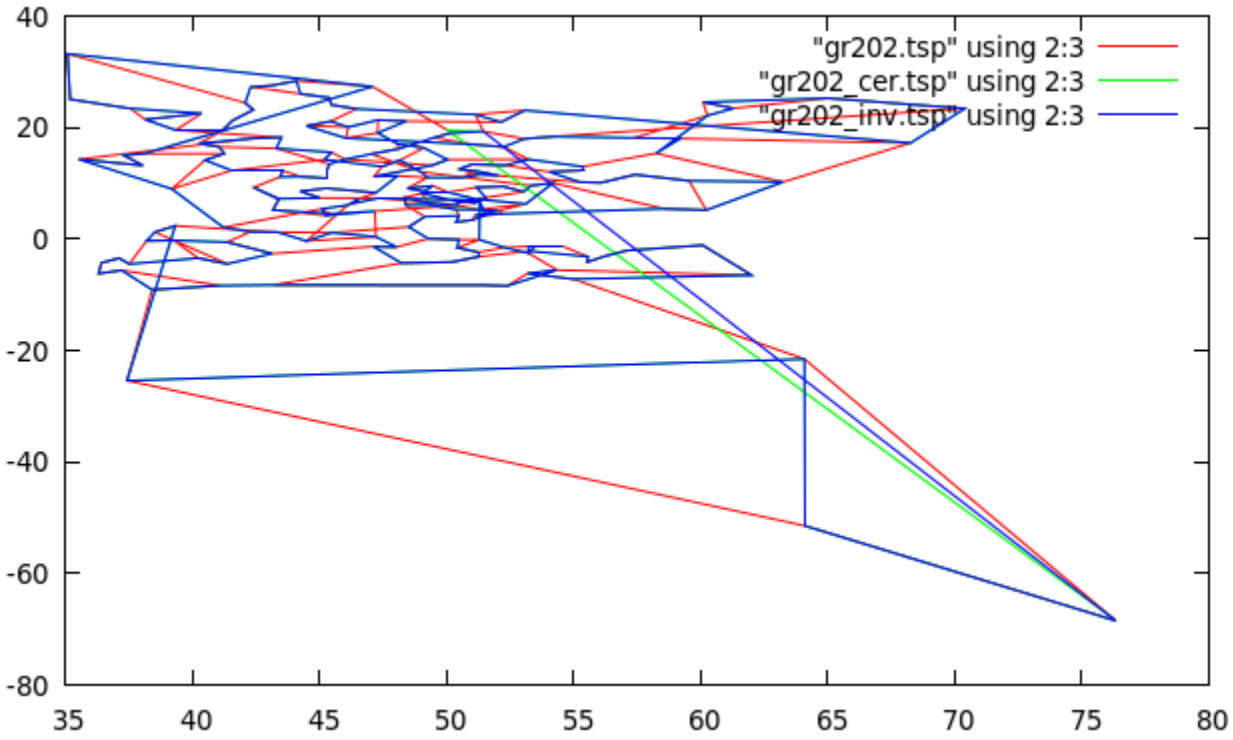


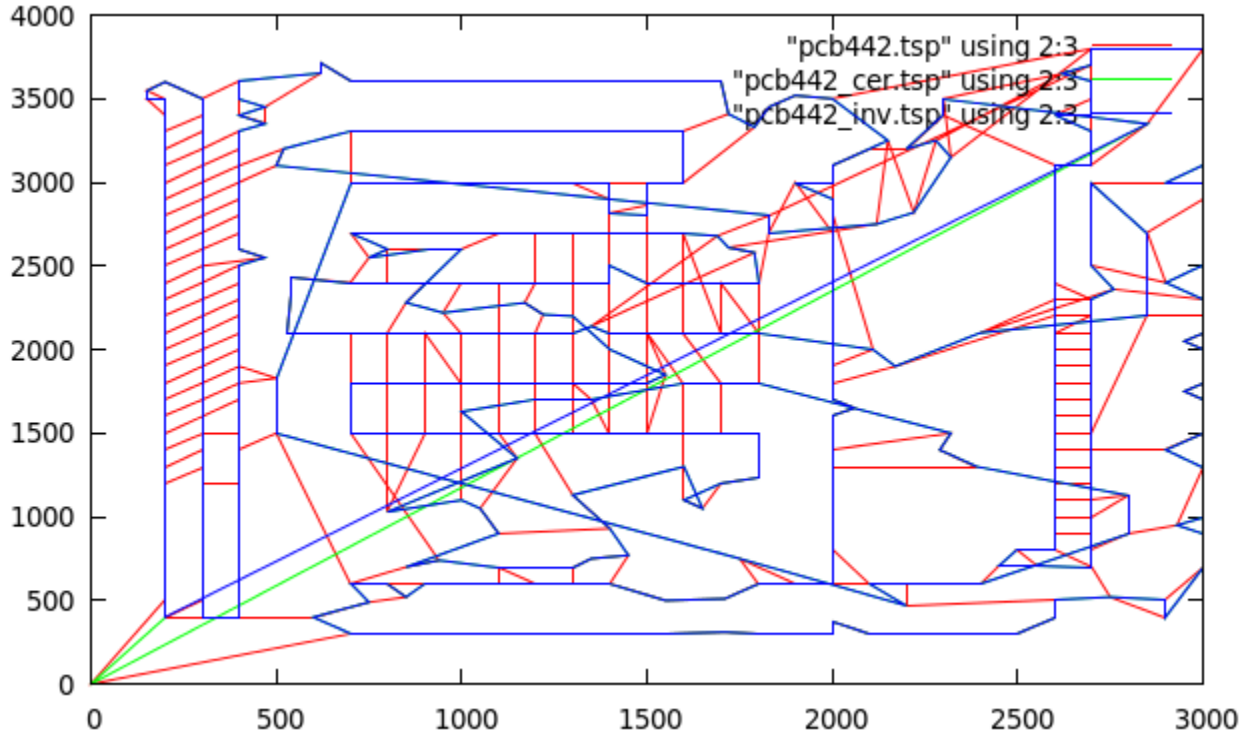

"bayg29.tsp" using 2:3
"bayg29_cer.tsp" using 2:3
"bayg29_inv.tsp" using 2:3
2400
2200
2000
1800
1600
1400
1200
1000
800
600
400
0
200
400
600
800
1000
1200
1400
1600
1800
2000

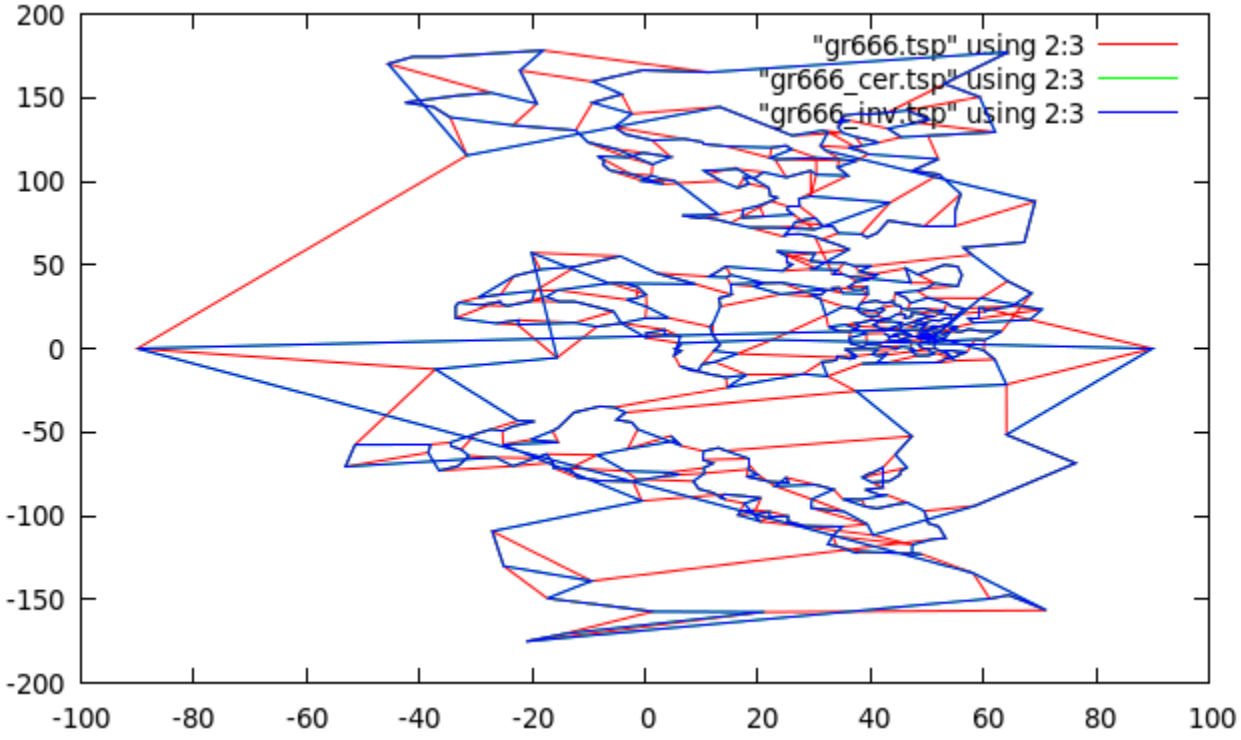

"att48.tsp" using 2:3
"att48_cer.tsp" using 2:3
"att48_inv.tsp" using 2:3
6000
5000
4000
3000
2000
1000
0
0
1000
2000
3000
4000
5000
6000
7000
8000


"berlin52.tsp" using 2:3
"berlin52_cer.tsp" using 2:3
"berlin52_inv.tsp" using 2:3
0
200
400
600
800
1000
1200
1400
1600
1800


"ch130.tsp" using 2:3
"ch130_cer.tsp" using 2:3
"ch130_inv.tsp" using 2:3
0
100
200
300
400
500
600
700


"gr202.tsp" using 2:3
"gr202_cer.tsp" using 2:3
"gr202_inv.tsp" using 2:3
40
20
0
-20
-40
-60
-80
35
40
45
50
55
60
65
70
75
80


"pcb442.tsp" using 2:3
"pcb442_cer.tsp" using 2:3
"pcb442_inv.tsp" using 2:3
4000
3500
3000
2500
2000
1500
1000
500
0
0
500
1000
1500
2000
2500
3000


"gr666.tsp" using 2:3
"gr666_cer.tsp" using 2:3
"gr666_inv.tsp" using 2:3
200
150
100
50
0
-50
-100
-150
-200
-100
-80
-60
-40
-20
0
20
40
60
80
100

6.2. Comparativa de resultados entre Algoritmos.

Tabla comparativa entre resultados.

Archivo	**cheapest_insertion**	**ciudad_mas_cercana**	algoritmo_**inventado**
ulysses16.tsp	71	58	58
bayg29.tsp	9163	8818	8818
att48.tsp	31158	35134	35134
berlin52.tsp	7700	7309	7309
ch130.tsp	5338	6644	6644
gr202.tsp	512	527	527
pcb442.tsp	43926	57581	57581
gr666.tsp	3289	3829	3829
pr1002.tsp	295595	304268	304268

Como se puede observar en los resultados, los 3 algoritmos son muy similares en cuanto a los resultados que ofrecen, aunque el algoritmo de la inserción más barata es el mejor de los 3, también se puede observar que todos ellos dan resultados bastante aproximados a la solución óptima, y funcionan en un tiempo relativamente rápido teniendo en cuenta el tamaño de algunas soluciones, son bastante eficientes.

A continuación muestro una gráfica que enseña los resultados de una forma más directa para que se puedan observar de forma gráfica los resultados del problema.

Gráfica comparativa entre resultados.