UNIVERSIDAD DE GRANADA

E.T.S. DE INGENIERÍAS INFORMATICA y DE TELECOMUNICACIÓN

Departamento de Ciencias de la Computación e Inteligencia Artificial

Práctica 3: Problema de viajante de comercio

grupo b4

Alejandro Casado Quijada

Andrés Ortiz Corrales

Antonio Jiménez Martínez

Jesús Prieto López

Salvador Rueda Molina

Curso 2013-2014

Grado en Informática

1. Objetivo

El objetivo de esta práctica es que el estudiante aprecie la utilidad de los métodos voraces (greedy) para resolver problemas de forma muy eficiente, en algunos casos obteniendo soluciones óptimas y en otros aproximaciones.

2. Viajante del comercio.

Dado un conjunto de ciudades y una matriz con las distancias entre todas ellas, un viajante debe recorrer todas las ciudades exactamente una vez, regresando al punto de partida, de forma tal que la distancia recorrida sea mínima.
Mas formalmente, dado un grafo G, conexo y ponderado, se trata de hallar el ciclo hamiltoniano de mínimo peso de ese grafo.

Solucionaremos este problema con tres distintos tipos de algoritmos greedy.

a) Estrategias basadas en el vecino más cercano:

Primero creamos una matriz con la distancias entre las ciudades, utilizando el algoritmo de Euclides. Comenzamos seleccionando la ciudad inicial como la 1. Recorremos la fila 1y seleccionamos la ciudad a menor distancia (columnas). Cuando hacemos esto saltamos a la fila de la ciudad seleccionada y volvemos a realizar este procedimiento. Tenemos en cuenta que no podemos introducir ciudades repetidas.

Cuando haya introducido de la lista todas las ciudades sin repeticiones, añadimos a la distancia final el camino entre la última ciudad y la primera, e insertamos de nuevo la ciudad 1 (única que se repite).

Mostramos el enfoque greedy:

-Conjunto de candidatos: Ciudades a visitar
-Conjunto de seleccionados: Ciudades visitadas
-Función solución: Si hemos recorrido todas las ciudades
-Función de factibilidad: Que la ciudad no haya sido visitada
-Función de selección: Seleccionamos la ciudad más cercana
-Función objetivo: Recorrer todas las ciudades y volver a la primera

b) Estrategias de inserción:

En las estrategias de inserción, la idea es comenzar con un recorrido parcial, que incluya algunas de las ciudades, y luego extender este recorrido insertando las ciudades restantes.

El recorrido inicial se puede construir a partir de las tres ciudades que formen un triángulo lo más grande posible: por ejemplo, eligiendo la ciudad que está más a Este, la que está más al Oeste, y la que está más al norte.

Vamos recorriendo el vector de ciudades no seleccionadas e insertando cada una de ellas en cada posible posición. Utilizando el criterio de inserción más económica, elegimos aquella ciudad y su posición en el camino final, que provoque un menor incremente en la longitud total del circuito.

Mostramos el enfoque greedy:

-Conjunto de candidatos: Ciudades a visitar
-Conjunto de seleccionados: Ciudades visitadas

-Función solución: Si hemos recorrido todas las ciudades
-Función de factibilidad: Que la ciudad no haya sido visitada
-Función de selección: Seleccionamos de entre todas las ciudades cual tendría mejor distancia, comprobándolo en todas las ubicaciones posibles.
-Función objetivo: Recorrer todas las ciudades y volver a la primera

c) Algoritmo basado en aristas.

Este algoritmo está implementado por nosotros, en el cuál creamos un map (map<int , pair<int, int> > aristas) formado por la distancia entre dos ciudades cada entrada.

Vamos seleccionando las distancias menores entre dos ciudades y añadiendo estas al conjunto de seleccionados. Para poder insertar una nueva ciudad, esta no puede tener más de 2 aristas ni crear un ciclo. Cómo el camino resultante de este algoritmo no es cerrado, tenemos que recorrer el camino y añadir la distancia entre las 2 ciudades que únicamente tienen una arista.

Mostramos el enfoque greedy:
-Conjunto de candidatos: Ciudades a visitar
-Conjunto de seleccionados: Ciudades visitadas
-Función solución: Si hemos recorrido todas las ciudades
-Función de factibilidad: Que la ciudad no tenga más de 2 arista y no se cree un ciclo al añadirla
-Función de selección: Seleccionamos las ciudades con menor distancia entre ellas.
-Función objetivo: Recorrer todas las ciudades y volver a la primera.

2.1. Código de los algoritmos.

En los tres algoritmos utilizamos:
- la función para crear la matriz de distancias

```
//calcular distancia
int euclides (double xi,double yi,double xj,double yj) {
   double aux1=(xj-xi)*(xj-xi);
   double aux2=(yj-yi)*(yj-yi);
   aux2=aux1+aux2;
   int aux=sqrt(aux2);
   return aux;
}
```

```
//creamos la matriz de distancias
//Asignamos a la distnacia entre la misma ciudad como -1.
void calcular_matriz(map<int, pair<double, double> >  m,vector< vector<int> > & matriz) {
   for(int i=0; i<matriz.size(); i++) {
      for(int j=0; j<matriz[i].size(); j++)
         if(i!=j) {
            matriz[i][j]=euclides(m[i+1].first,m[i+1].second,m[j+1].first,m[j+1].second);
         }
   }
}
```

-Muestra las coordenadas de las cuidades cogidas como ciclo hamiltoniano.

```
//devuelve las ciudades ordenadas con sus coordenadas.
void mostrar_resultado(const vector<int> &result,const map<int,pair<double,double> > &m) {
   for(int i=0; i<result.size(); i++) {
      pair<double,double> p;
      p=(m.find(result[i]))->second;
      cout<<result[i]<<" "<<p.first<<" "<<p.second<<endl;
   }
}
```

-Código Algoritmo 1.

```
//rellenamos la columna de -1 para que esta no se vuelva a utilizar.
void modificar_columna(vector< vector<int> > & m,int colum) {
   for(int i=0; i<m.size(); i++) {
      for(int j=0; j<m[i].size(); j++)
         if(colum==j) {
            m[i][j]=-1;
         }
   }
}
```

```
//tomamos como nodo inicial la ciudad 1.
//entra la matriz
//devolvemos el vector de ciudades cogidas y la distancia.
vector<int> recorrido(vector< vector<int> >  matriz,int & d) {
   vector<int>c;//ciudades selecionadas
   int nodo=0,min=0,pos=0,i=1,j=0;
   //primero introducimos el primero nodo en el vector
   c.push_back(nodo+1);
   for(; i<matriz.size(); i++) { //contador
      j=1;
      while(matriz[nodo][j]==-1) {
         j++;
      }
      min=matriz[nodo][j];
      pos=j;
      for(; j<matriz[nodo].size(); j++) {
         if(matriz[nodo][j]!=-1 and min>matriz[nodo][j]) {
            min=matriz[nodo][j];
            pos=j;
         }
      }
      nodo=pos;
      c.push_back(nodo+1);
      modificar_columna(matriz,nodo);
      d+=min;
   }
   //añadirle la distnacia del ultimo nodo al inicio
   d+=matriz[nodo][0];
   //le añadimos de nuevo el nodo 1..
   c.push_back(1);
   return c;
}
```

-Código Algoritmo 2.

```
//distancia entre  2 iudades
int distancia(int ciudad1,int ciudad2,const vector<vector<int> > &m) {
   ciudad1--;
   ciudad2--;
   return m[ciudad1][ciudad2];
}
int distancia_total(const list<int> &result,const vector<vector<int> > &m) {
   list<int>::const_iterator it1,it2;
   int dist=0;
   it1=result.begin();
   it2=result.begin();
   it1++;
//dist+=distancia(*it1,*it2,m);
//it2=it1;
   for(; it1!=result.end(); it1++) {
      dist+=distancia(*it1,*it2,m);
      it2=it1;
   }
   return dist;
}
```

```
//distancia que supone insertar una ciudad
pair<int,list<int>::iterator> aumento_de_distancia(list<int> &result,const vector<vector<int> > &matriz,int ciudad) {
   list<int>::iterator it1=result.begin(); //apunta a la primera ciudad de la lista
   list<int>::iterator it2=result.end();//apunta a la ultima ciudad de la lista
   list<int>::iterator pos=result.begin();
   it2--;
   int dist;
   dist=distancia(*it1,ciudad,matriz)+distancia(*it2,ciudad,matriz);
   dist-=distancia(*it1,*it2,matriz); //dist tiene el aumento de la distancia
   it2=it1;//it2 apunta a la primera ciudad
   it1++;//it apunta a la segunda ciudad
   for(; it1!=result.end(); it1++) {
      int a=*it1;
      int b=*it2;
      int dist2=distancia(a,ciudad,matriz)+distancia(b,ciudad,matriz);
      dist2-=distancia(a,b,matriz);
      if(dist2<dist) {
         dist=dist2;
         pos=it1;
      }
      it2=it1;
      it1++;
   }
   return make_pair(dist,pos);
}
```

```
//elige la ciudad con menor distancia añadida y devuelve pair<pair<ciudad,distancia_añadida,posicion_en_lista>
pair<int,list<int>::iterator> elegir_ciudad(list<int> &result,const vector<vector<int> > &matriz,list<int> &left) {
   list<int>::iterator it,itselec;
   pair<int,list<int>::iterator> selec;
   bool b=false;
   int cit_selec;
   for(it=left.begin(); it!=left.end(); it++) {
      pair<int,list<int>::iterator> selec2;
      int cit=*it;
      selec2=aumento_de_distancia(result,matriz,cit);
      if(b==false) {
         b=true;
         selec=selec2;
         cit_selec=cit;
         itselec=it;
      }
      else if(selec2.first<selec.first) {
         selec=selec2;
         cit_selec=cit;
         itselec=it;
      }
   }
   left.erase(itselec);
   return make_pair(cit_selec,selec.second);
}
```

```
//tomamos como nodo inicial la ciudad 1. (0 en la matriz)
//entra la matriz
//devolvemos el vector de ciudades cogidas y la distancia.
list<int> recorrido(const vector<vector<int> > &matriz,const map<int, pair<double, double> >  &m,int &d) {
   int n=1,s=1,w=1;
   list<int> result;
   map<int,pair<double,double> >::const_iterator it=m.begin();
   list<int> left;
   left.push_back(1);
   pair<double,double> pn=(*it).second,ps=(*it).second,pw=(*it).second; //todas las posiciones a la primera ciudad de la lista
   it++;
   for(; it!=m.end(); it++) { //escojemos las ciudades mas al norte,sur y oeste
      left.push_back((*it).first);
      pair<double,double> p=(*it).second;
      int act_cit=(*it).first;
      if(p.first<pw.first) {
         pw=p;
         w=act_cit;
      }
      else if(p.second>pn.second || (n==w && n==1)) {
         pn=p;
         n=act_cit;
      }
      else if(p.second<ps.second || (s==w && s==1)) {
         ps=p;
         s=act_cit;
      }
   }
   //n,s,w forman el triangulo mas grande
   //suponemos que empezamos en n, una de las ciudades del triangulo
   result.push_back(n);
   result.push_back(s);
   result.push_back(w); //añadimos el triangulo al recorrido
   left.remove(n);
   left.remove(s);
   left.remove(w);
   //////////////
   while(left.size()>0) {
      pair<int,list<int>::iterator> pres;
      pres=elegir_ciudad(result,matriz,left);
      insertar_ciudad(result,pres.first,pres.second);
   }
   result.push_back(n); //volvemos a la primera ciudad
   d=distancia_total(result,matriz);
   return result;
}
```

-Código Algoritmo 3.

```
multimap<int , pair<int,int> > calcular_aristas(const vector< vector<int> > & m) {
    multimap<int , pair<int,int> > a;
    for(int i=0; i<m.size(); i++) {
        for(int j=0; j<m[i].size(); j++) {
            if(j>i) {
                a.insert(pair<int, pair<int,int> >(m[i][j],pair<int,int>(i+1,j+1)));
            }
        }
    }
    return a;
}
```

```
//si tiene mas de dos aritas alguna ciudad
bool factible( list<pair<int,int> > & aux, const pair<int,int> & s) {
    int A=s.first,B=s.second,contA=0,contB=0;
    list<pair<int,int> >::iterator it=aux.begin();
    for(; it!=aux.end(); it++) {
        if(A==(*it).first or A==(*it).second) contA++;
        if(B==(*it).first or B==(*it).second) contB++;
    }
    return (contA>1 or contB>1);
}
//si al añadirle el nodo s, todos tiene dos aristas, se crea un ciclo
bool hayciclos(list<pair<int,int> > aux, const pair<int,int> & s) {
    int pos,res;
    list<pair<int,int> >::iterator it;
    res=s.second;
    pos=s.first;
    it=buscar2(aux,pos);
    while(it!=aux.end()) {
        if(pos==(*it).first) pos=(*it).second;
        else pos=(*it).first;
        aux.erase(it);
        it=buscar2(aux,pos);
    }
    return pos==res;
}
```

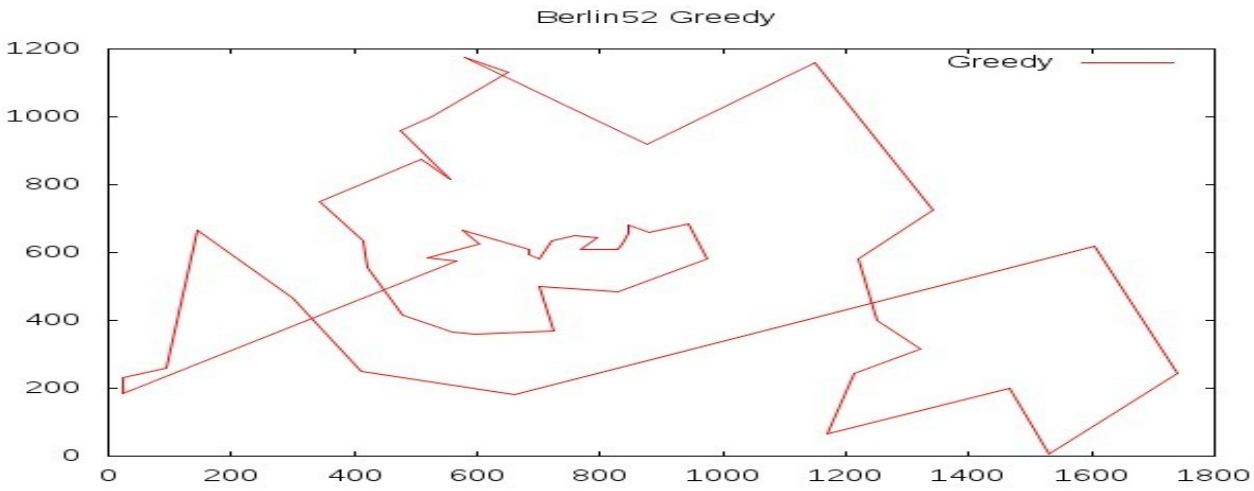
```
list<pair<int,int> >::iterator buscar(list<pair<int,int> >& aux,int & obj) {
   list<pair<int,int> >::iterator it;
   for(it=aux.begin(); it!=aux.end(); it++) {
      if((*it).first==obj) {
         obj=(*it).second;
         return it;
      }
      else if((*it).second==obj) {
         obj=(*it).first;
         return it;
      }
   }
```

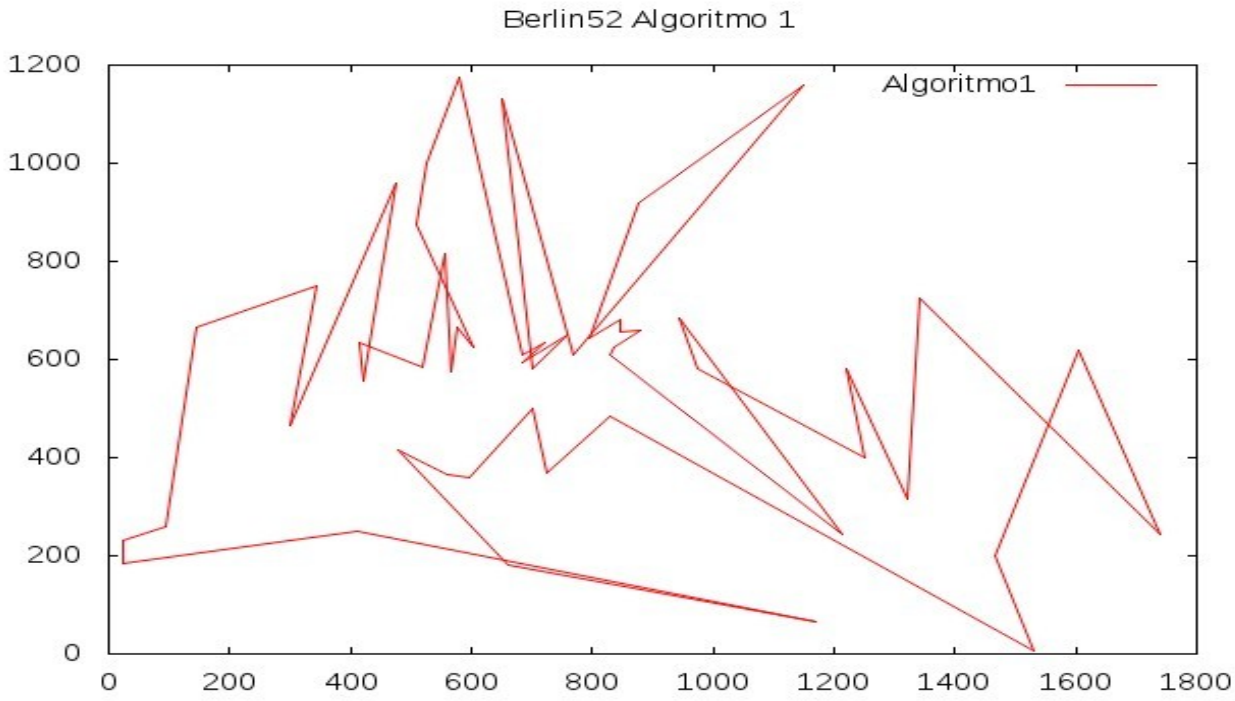
```
//creamos el camino
vector <int> camino(list <pair<int,int> >aux) {
   vector<int>c;
   list <pair<int,int> >::iterator it=aux.begin();
   c.push_back((*it).first);
   //c.push_back((*it).second);
   int obj=(*it).second;
   aux.erase(it);
   while(!aux.empty()) {
      c.push_back(obj);
      it=buscar(aux,obj);
      //c.push_back(obj);
      aux.erase(it);
   }
   c.push_back(c.front());
   return c;
}
```

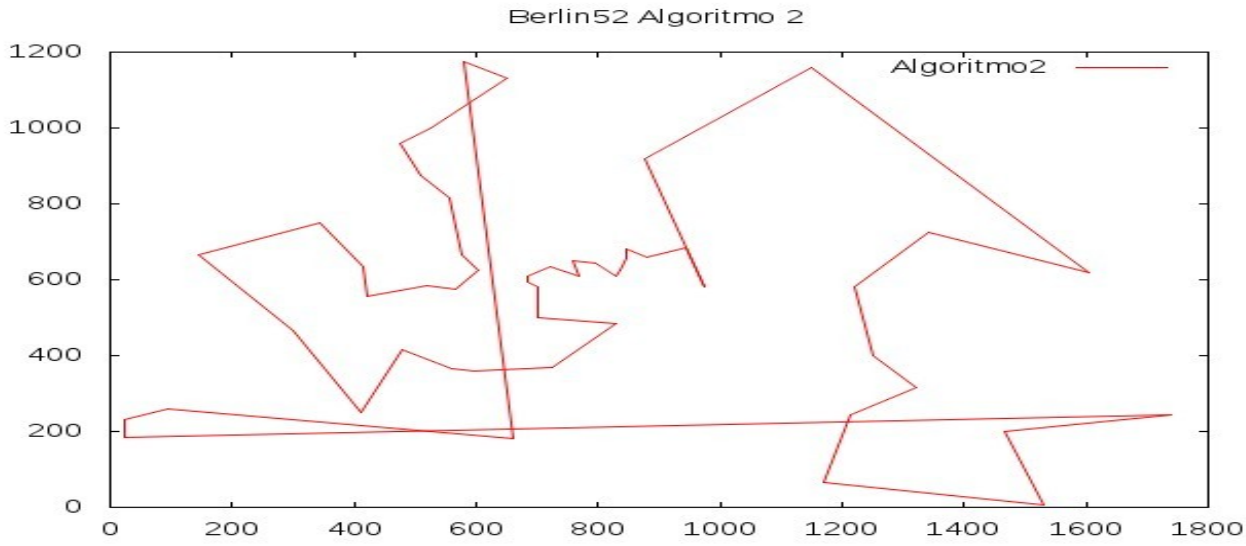
```
vector<int> recorrido(multimap<int, pair<int,int> > & a,int & d) {
   vector<int>c;
   list<pair<int,int> > aux;
   multimap<int , pair<int,int> >::iterator it=a.begin();
   aux.push_back((*it).second);
   d+=(*it).first;
   it++;
   while(it!=a.end()) {
      if(!factible(aux,(*it).second) and !hayciclos(aux,(*it).second)) {
         aux.push_back((*it).second);
         d+=(*it).first;
      }
      it++;
   }
   cerrar_ciclo(aux,a,d);
   //mostrar(aux);
   c=camino(aux);
   return c;
}
```

```
//cerramos el camino, uniendo las dos cuidades con una
void cerrar_ciclo(list <pair<int,int> > & aux,multimap<int, pair<int,int> > & a,int & d) {
   int falta1=0,falta2=0,contadorA=0,contadorB=0;
   list<pair<int,int> >::iterator it1,it2;
   for(it1=aux.begin(); it1!=aux.end(); it1++) {
      for(it2=aux.begin(); it2!=aux.end(); it2++) {
         if((*it1).first==(*it2).first or (*it1).first==(*it2).second) contadorA++;
         if((*it1).second==(*it2).first or (*it1).second==(*it2).second) contadorB++;
         if(contadorA==2 and contadorB==2) break;
      }
      if(contadorA<2 and falta1==0) falta1=(*it1).first;
      else  if(contadorA<2 and falta2==0) falta2=(*it1).first;
      if(contadorB<2 and falta1==0) falta1=(*it1).second;
      else if(contadorB<2 and falta2==0) falta2=(*it1).second;
      contadorA=contadorB=0;
   }
   multimap<int , pair<int,int> >::iterator it;
   for(it=a.begin(); it!=a.end(); it++) {
      if((((*it).second).first==falta1 and ((*it).second).second==falta2)or  (((*it).second).first==falta2 and ((*it).second).second==falta1)) {
         aux.push_back((*it).second);
         d+=(*it).first;
      }
   }
}
```

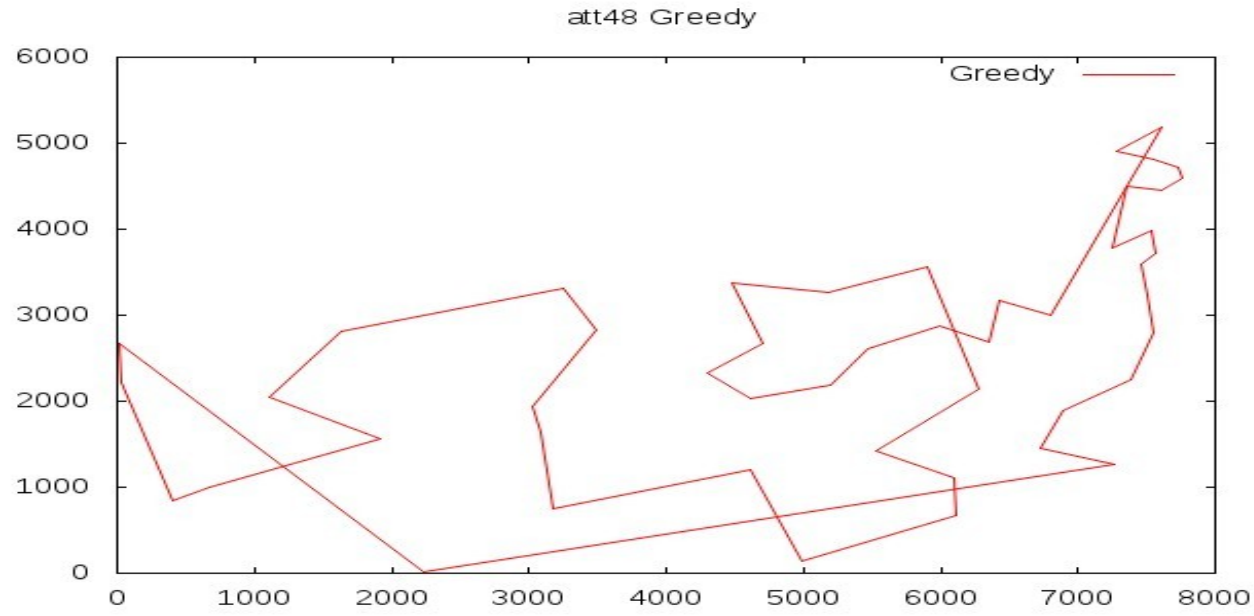
3.Comprobación de algoritmos en ejemplos de ciudad Berlin52

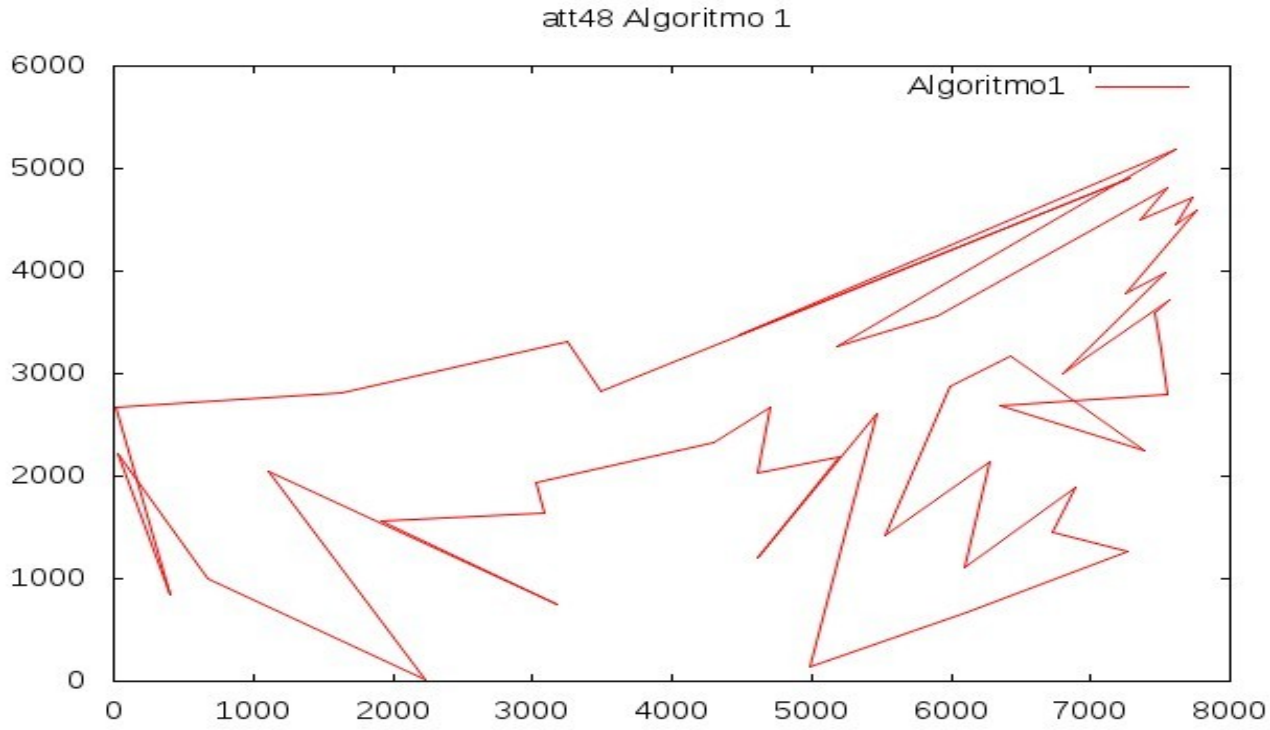


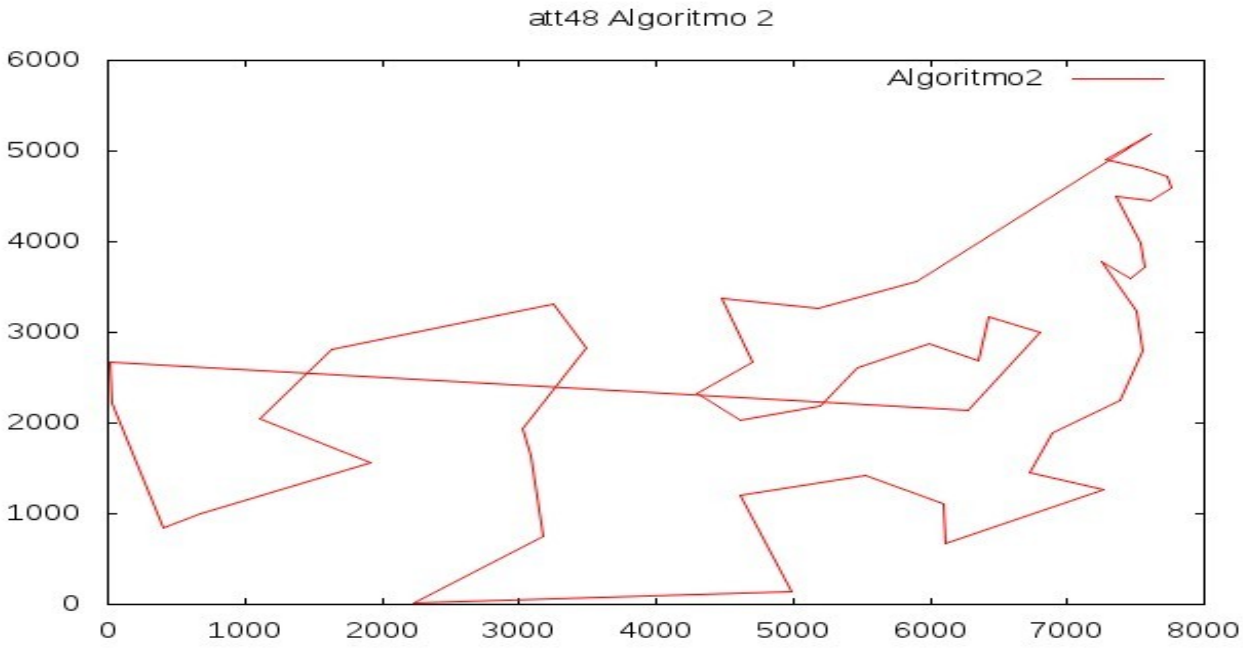




4.Comprobación de algoritmos en ejemplos de ciudad att48




att48 Algoritmo 1
Algoritmo1
6000
5000
4000
3000
2000
1000
0
0
1000
2000
3000
4000
5000
6000
7000
8000


att48 Algoritmo 2
Algoritmo2
6000
5000
4000
3000
2000
1000
0
0
1000
2000
3000
4000
5000
6000
7000
8000