Algorítmica - Algoritmos voraces (Greedy)
Problema del viajante de comercio

Realizado por:
Sergio Cervilla Ortega
Daniel Díaz Pareja
Marina Hurtado Rosales
Adrián Morente Gabaldón
Joaquín Rodríguez Claverías

Índice

1. Descripción del problema.

En su formulación más sencilla, el problema del viajante de comercio (TSP, por Traveling Salesman Problem) se define como sigue: 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. Más formalmente, dado un grafo G, conexo y ponderado, se trata de hallar el ciclo hamiltoniano de mínimo peso de ese grafo.

Nos centraremos en una serie de algoritmos aproximados de tipo greedy y evaluaremos su rendimiento en un conjunto de instancias del TSP. Para el diseño de estos algoritmos, utilizaremos tres enfoques diferentes:

- Estrategias basadas en alguna noción de cercanía.
- Estrategias de inserción según algún criterio referente a los nodos.
- Estrategia propuesta por nuestro equipo, referente a un criterio de inserción de aristas.

2. Vecino más cercano.

2.1 Descripción del algoritmo.

En el primer caso emplearemos la heurística de la ciudad vecina más cercana, cuyo funcionamiento es extremadamente simple: dada una ciudad inicial v0, se agrega como ciudad siguiente aquella vi (no incluída en el circuito) que se encuentre más cercana a v0. El procedimiento se repite hasta que todas las ciudades se hayan visitado.

2.2 Elementos.

- Conjunto de candidatos (C): Ciudades no visitadas.
- Conjunto de seleccionados (S) : Ciudades ya visitadas.
- Función solución: Que se hayan visitado todas las ciudades solo una vez.
- Función de Factibilidad: siempre hay una carretera de una ciudad al resto, podemos visitar cualquier ciudad en cualquier momento; por lo que la función de factibilidad en este caso se cumple siempre.
- Función de Selección: siempre escogeremos en cada momento la ciudad no visitada aún más cercana.
- Función objetivo: Devuelve el recorrido obtenido y su longitud.

2.3 Implementación.

Código:

```
vector<int> recorrido_greedy(vector< vector<int> >  matriz,
                             int & distancia, const int & nodo_inicial)
{
    vector<int> ciudades;
    int ciudad=nodo_inicial,min=0,pos=0;
    int j;
    discard_column(matriz,ciudad);

    for(int i=1; i<matriz.size(); ++i) {
        j=0;
        while(matriz[ciudad][j]==-1)
            j++;

        min=matriz[ciudad][j];
        pos=j;

        while(j<matriz[ciudad].size()){
            if(matriz[ciudad][j]!=-1 && min>matriz[ciudad][j]) {
                min=matriz[ciudad][j];
                pos=j;
            }
            j++;
        }

        ciudad=pos;
        ciudades.push_back(ciudad+1);
        distancia+=min;

        if (i==matriz.size()-1)
                distancia+=matriz[nodo_inicial][ciudad];

        discard_column(matriz,ciudad);
    }

    return ciudades;
}
```

Parámetros:

- **matriz**: Matriz de distancias entre las ciudades.
- **distancia**: Variable donde se irá almacenando la distancia del recorrido solución.
- **nodo_inicial**: Nodo (de la matriz) desde el que parte el algoritmo.

Variables:

- vector<int> ciudades; // Recorrido solución, esto es: conjunto de ciudades ya ordenado según orden de visita.
- int ciudad; // Se refiere a la posición que ocupa la ciudad en la matriz de distancias. Aquí se almacenará la siguiente ciudad a incluir en el conjunto de seleccionados.
- int min; // Se refiere al siguiente camino con menor longitud desde la posición actual.
- int pos; // Aquí se irá almacenando la ciudad más cercana encontrada en cada iteración.
- int j; // Se utiliza para acceder al recorrido desde la ciudad actual al resto.

Descripción:

Lo primero que realiza el algoritmo es incluir la ciudad inicial en el vector resultado, a partir del nodo inicial que ha sido pasado como argumento (se le suma 1, ya que "nodo_inicial" se refiere a la posición que ocupa la ciudad en la matriz; es decir, la ciudad 1 estará en el índice 0, la ciudad 2 en el índice 1, y así sucesivamente).

Como la ciudad ya ha sido incluida, se pone su columna de distancias a -1, para que no vuelva a ser considerada en el proceso de selección.

Una vez hecho esto, se empieza a iterar desde la ciudad actual al resto. Primero comprobamos cual es la primera ciudad elegible del conjunto de candidatos (es decir, que su distancia desde el nodo actual sea distinta de -1). A partir de aquí, comprobamos cuál es la menor distancia del nodo actual al resto de ciudades, empezando a comprobar por la ya mencionada.

Una vez encontrada esa distancia, guardamos en una variable cuál es el nuevo nodo a colocar en el conjunto de seleccionados, y lo incluimos en dicho conjunto. Además, se añade la distancia recorrida a la distancia total del camino (inicialmente 0), además de modificar la columna de sus distancias a -1. En la siguiente iteración se realizará el mismo proceso desde esta nueva ciudad hasta que las recorramos todas.

Finalmente, en la última iteración, se verifica la condición del último if y se añade a la distancia total la distancia entre el nodo final y el inicial, cerrando el circuito.

2.4 Pruebas.

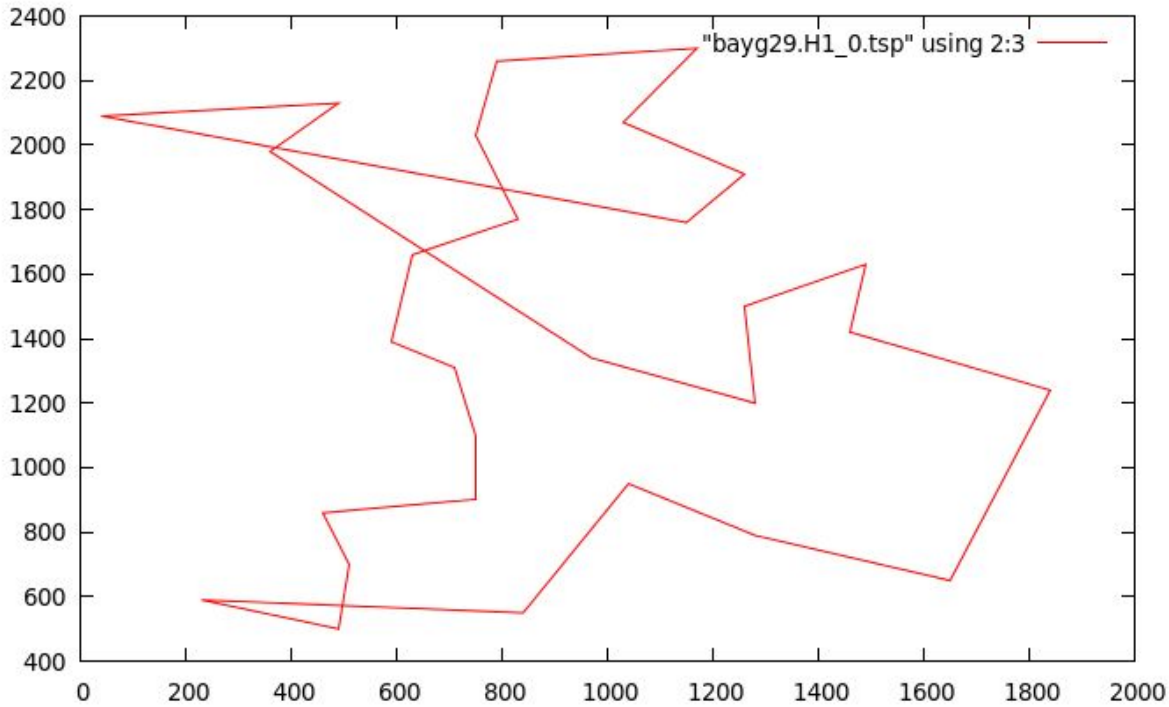
- Diferencias existentes entre elegir distintos nodos iniciales en el mapa bayg29:

Nodo inicial	Distancia total
0	10200
5	10695
13	11761
20	9953
28	10798

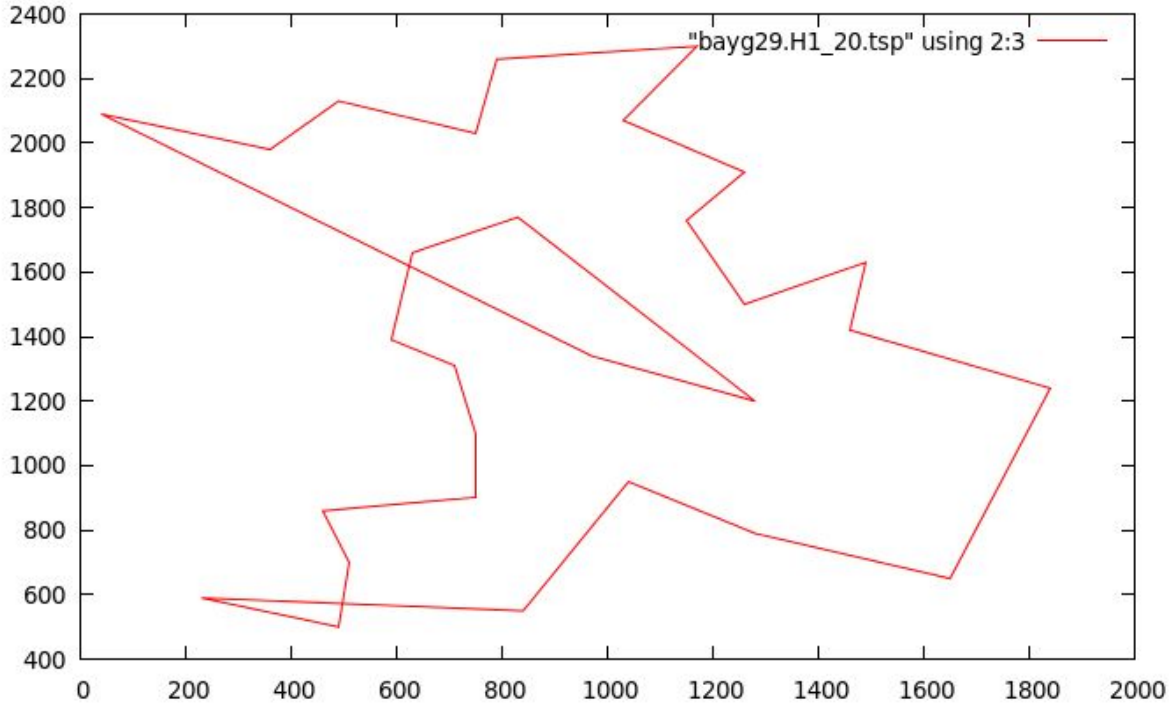
Como vemos, para esta heurística SÍ ES RELEVANTE el nodo inicial a escoger.

- Gráficos obtenidos con GNUPlot:

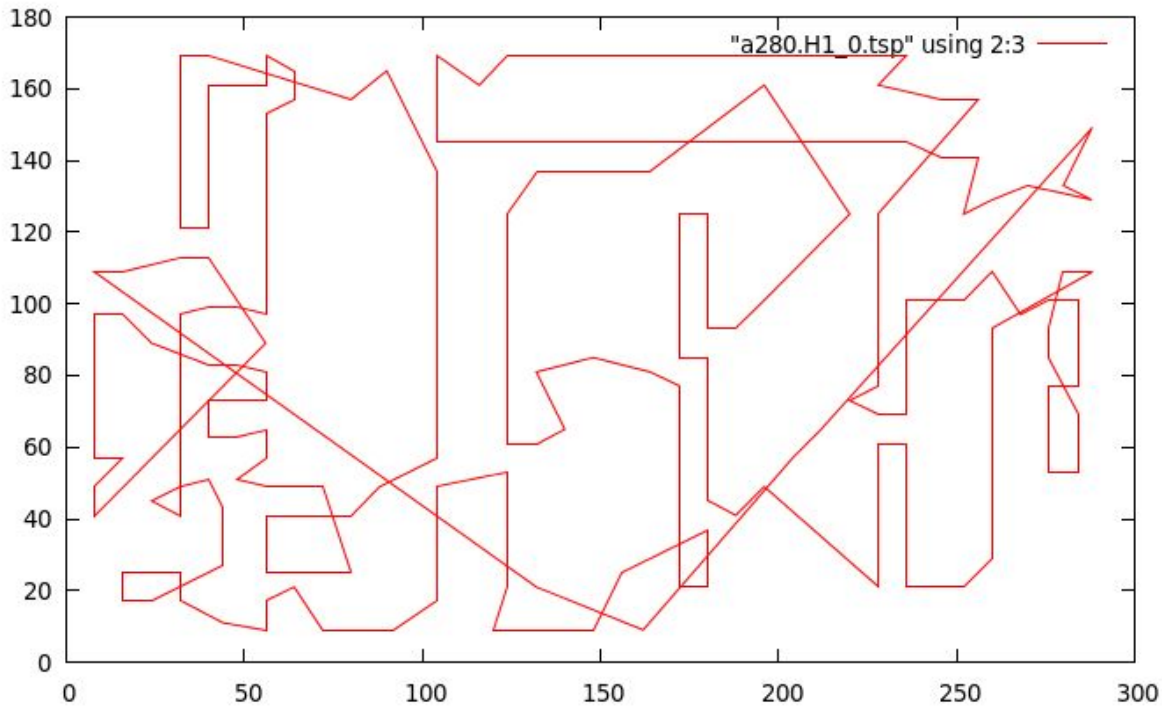
Recorrido para bayg29 empezando desde el nodo 0:



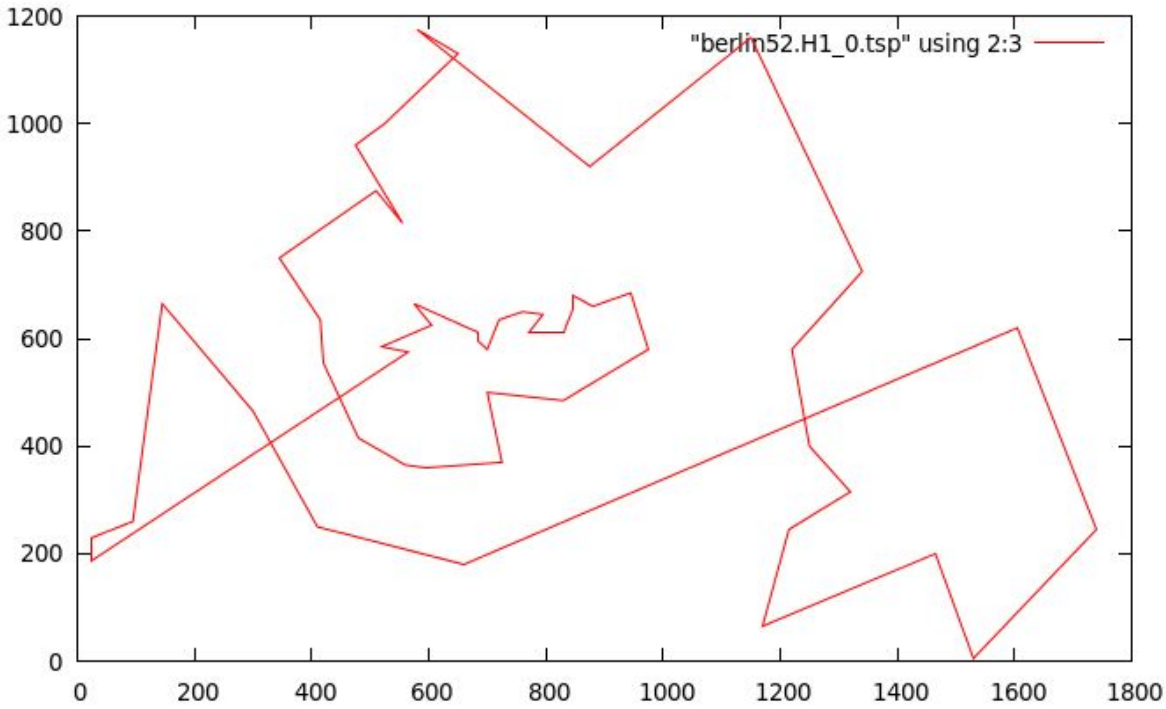
Recorrido para bayg29 empezando desde el nodo 20:



Recorrido para a208 empezando desde el nodo 0:



Recorrido para berlin52 empezando desde el nodo 0:



3. Inserción de ciudades.

3.1 Descripción del algoritmo.

En este segundo caso la idea principal es comenzar con un recorrido parcial, que tan solo incluya unas pocas ciudades, y posteriormente ir ampliando este recorrido hasta finalmente llegar al recorrido solución. En nuestro caso, vamos a optar por definir el recorrido inicial como un triángulo cuyos vértices son:

- La ciudad más al **oeste.**
- La ciudad más al **norte.**
- La ciudad más al **sur.**

Para ir completando este recorrido iremos insertando las ciudades según un determinado criterio de inserción, que en nuestro caso es insertar la ciudad que provoque el menor incremento de la longitud total del recorrido.

3.2 Elementos.

- Conjunto de candidatos (C): Ciudades no visitadas.
- Conjunto de seleccionados (S) : Ciudades ya visitadas.
- Función solución: Que se hayan visitado todas las ciudades solo una vez.
- Función de Factibilidad: siempre hay una carretera de una ciudad al resto, podemos visitar cualquier ciudad en cualquier momento; por lo que la función de factibilidad en este caso se cumple siempre.
- Función de Selección: La ciudad que menos aumento de distancia suponga en el recorrido total actual.
- Función objetivo: Devuelve el recorrido obtenido y su longitud.

3.3 Implementación.

Código:

```
list<int> greedy_insercion(const vector<vector<int> > & matriz,
                           const map<int, pair<double, double> > & m, int & d)
{
    int norte, sur, oeste;
    norte=sur=oeste=1;
    list<int> result;
    map<int, pair<double, double> >::const_iterator it=m.begin();
    list<int> restantes;
    restantes.push_back(1);
    pair<double, double> pnorte=(*it).second, psur=(*it).second, poeste=(*it).second;
    it++;

    for( ; it!=m.end(); it++){
        restantes.push_back((*it).first);
        pair<double,double> p=(*it).second;
        int actual=(*it).first;
        if(p.first < poeste.first){
            poeste = p;
            oeste = actual;
        }
        else if(p.second > pnorte.second || (norte==oeste && norte==1)){
            pnorte = p;
            norte = actual;
        }
        else if(p.second < psur.second || (sur==oeste && sur==1)){
            psur = p;
            sur = actual;
        }
    }

    result.push_back(norte);
    result.push_back(sur);
    result.push_back(oeste);
    restantes.remove(norte);
    restantes.remove(sur);
    restantes.remove(oeste);
```

```
    while(restantes.size()>0){
        pair<int, list<int>::iterator> pres;
        pres=elegir_ciudad(result, matriz, restantes);
        insertar_ciudad(result, pres.first, pres.second);
    }
    result.push_back(*(result.begin()));
    d=distancia_total(result, matriz);
    return result;
}
```

Parámetros:

- **matriz:** Matriz de distancias entre las ciudades.
- **m:** Map donde se almacenan las ciudades y sus coordenadas (clave: número de la ciudad, definición: par de coordenadas).
- **d:** Distancia total del recorrido.

Variables:

- int norte, sur, oeste. // Ciudades más al norte, sur y oeste del recorrido.
- list<int> result. // Lista con las ciudades ya ordenadas por nuestro algoritmo greedy.
- list<int> restantes. // Lista con las ciudades restantes por incluir.

Descripción:

Lo primero que realiza este algoritmo es buscar en toda la lista de ciudades las tres situadas más al norte, sur y oeste.

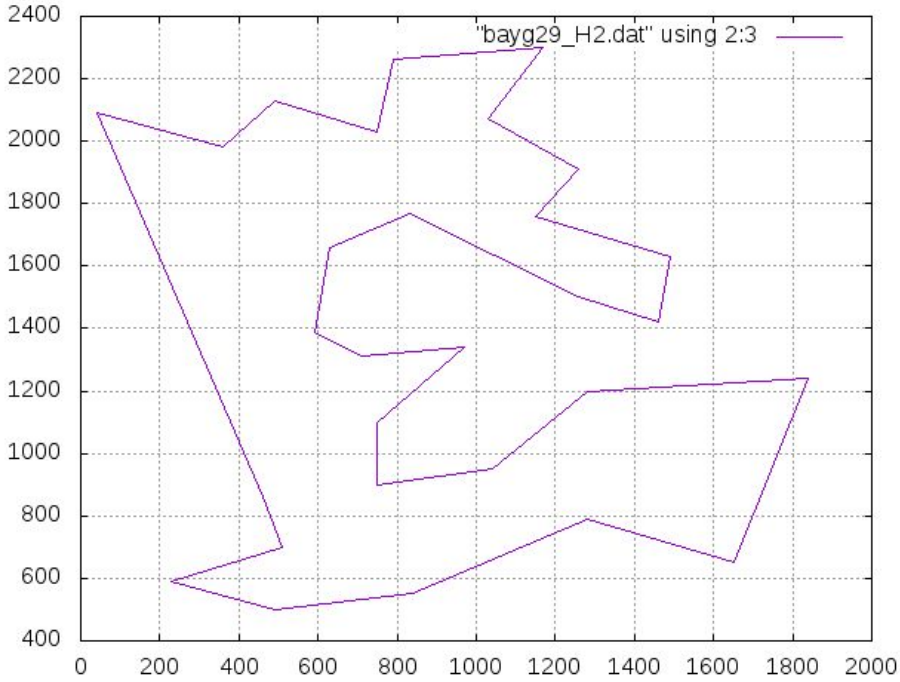
Una vez hecho esto, las incluye en nuestro recorrido resultado pues éste es el recorrido inicial que irá siendo modificado hasta obtener el recorrido solución. Por tanto, las incluimos en la lista de *result* y las eliminamos de la lista de *restantes*. Esto se hará así para todas las ciudades.

A continuación, vamos insertando las ciudades en el recorrido resultante mientras que la lista *restantes* tenga al menos 1 ciudad por insertar. Para ello vamos llamando a la función *elegir_ciudad* que lo único que hace es buscar en la lista de *restantes* la ciudad cuya inserción suponga menor costo para nuestro recorrido total, y una vez elegida ésta, la insertamos.

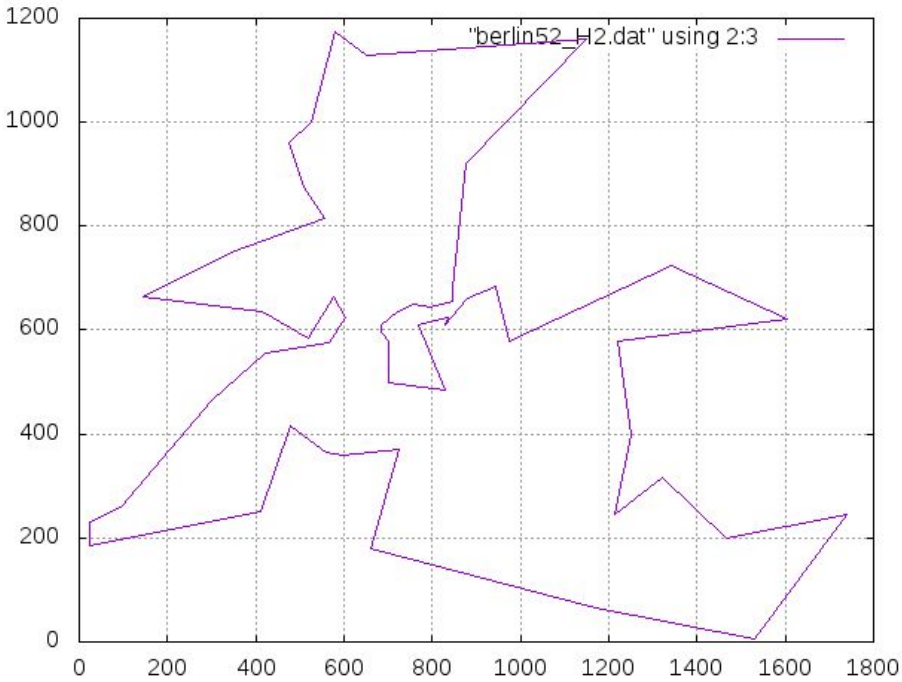
3.4 Pruebas.

Podemos representar gráficamente el recorrido final que ofrece nuestro algoritmo para 4 ciudades de prueba.

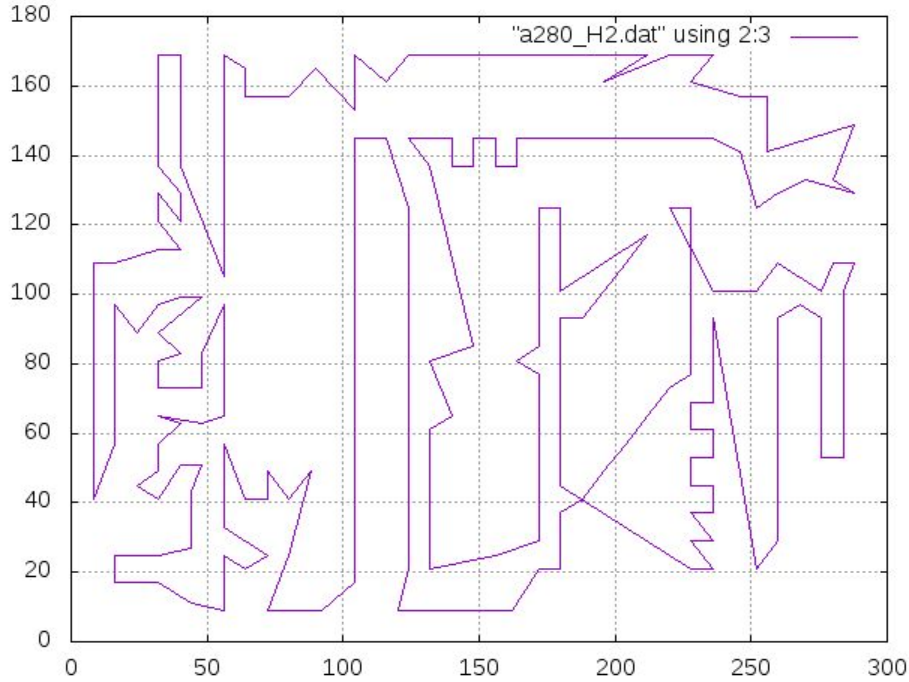
Recorrido para bayg29.



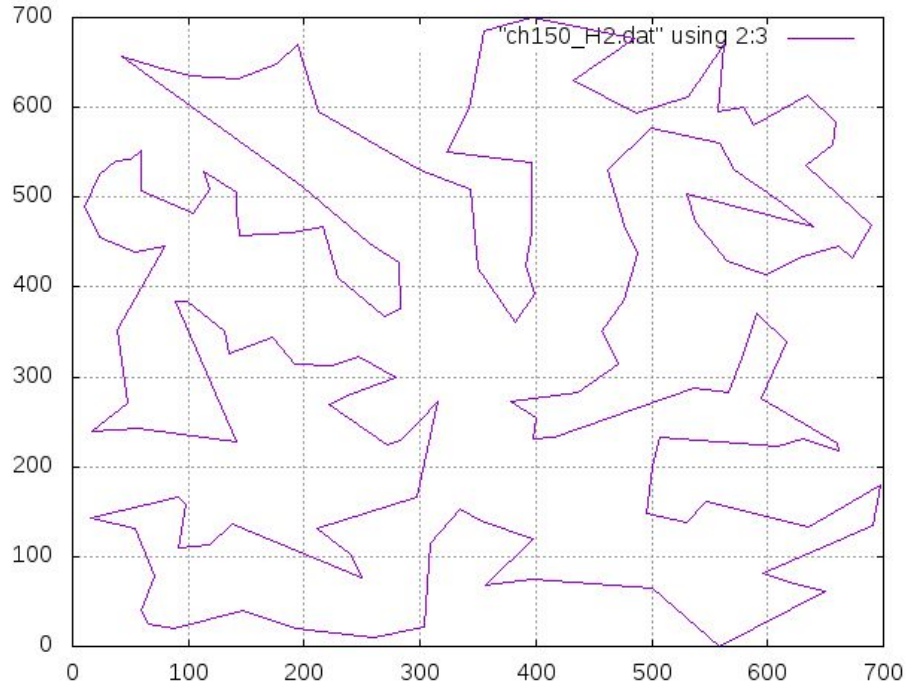
Recorrido para berlin52.



Recorrido para a280.



Recorrido para ch150.



4. Inserción de aristas

4.1. Descripción del algoritmo

En este tercer caso, dejamos un poco de lado la ubicación de cada ciudad para preocuparnos por la longitud de sus carreteras, en la que la heurística que hemos empleado comienza con una ciudad inicial v0, y se tiene una estructura de datos en la que cada entrada está formada por la distancia entre dos ciudades. Se añaden al conjunto de seleccionados las distancias menores entre dos ciudades, teniendo en cuenta que para poder insertar una nueva ciudad, ésta no puede tener más de 2 aristas, ni puede crear un ciclo (o lo que es lo mismo, cerrar el circuito antes de visitar todas las ciudades).
Este algoritmo no produce un camino cerrado, por lo que tenemos que añadir una última arista que vuelva a la ciudad inicial, y sumar la distancia que hemos recorrido para ello.

4.2. Elementos

- Conjunto de candidatos (C): Carreteras sin recorrer.
- Conjunto de seleccionados (S) : Carreteras recorridas.
- Función solución: el conjunto de carreteras seleccionado conecta todas las ciudades, y solo hemos pasado por cada ciudad una vez.
- Función de Factibilidad: el conjunto de carreteras no tiene ningún ciclo.
- Función de Selección: La carretera de menor coste, teniendo en cuenta que sólo puedo tener, como mucho, dos carreteras que pasen por cada ciudad.
- Función objetivo: Devuelve el recorrido obtenido y su longitud.

4.3. Implementación

Código:

```
vector<int> calcular_recorrido(multimap<int, pair<int,int> > &a, int &d){
    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(!nodo_factible(aux,(*it).second) && !hayciclos(aux,(*it).second)){
            aux.push_back((*it).second);
            d += (*it).first;
        }
        it++;
    }
    cerrar_ciclo(aux,a,d);
    vector<int> c = camino(aux);
    return c;
}
```

Parámetros:

- **a:** Map donde se almacenan las carreteras y las ciudades que unen.
- **d:** Distancia total del recorrido.

Variables:

- list<pair<int,int> > aux // Lista con el orden de las carreteras de nuestro recorrido greedy.
- vector <int> c // Vector con el orden de las ciudades de nuestro recorrido greedy.

Descripción:

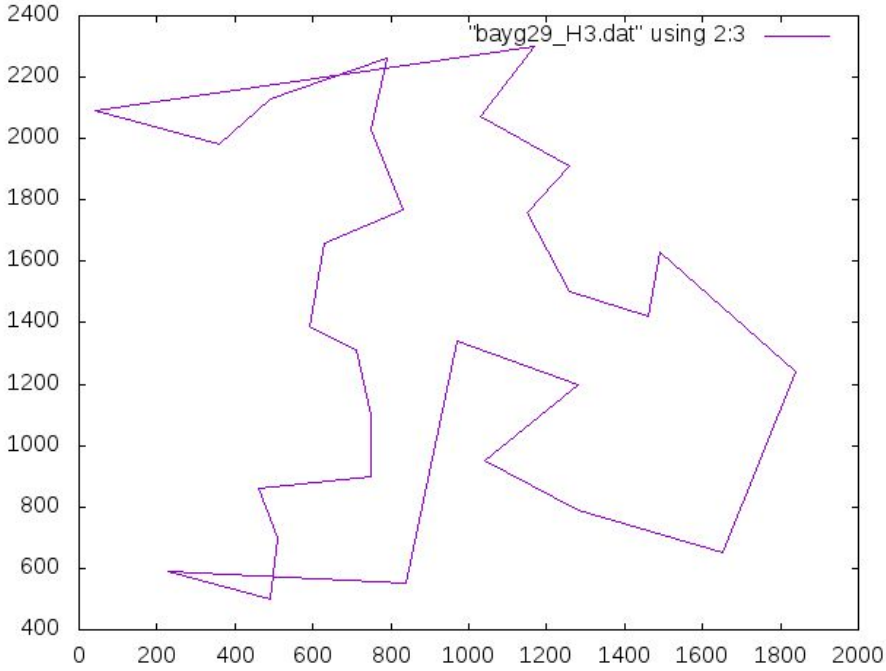
Lo primero que realiza este algoritmo es situarse en la primera ciudad. Una vez hecho esto, selecciona la carretera que suponga un menor coste adicional, y la añade al conjunto de seleccionados. A continuación, vamos comprobando todas las carreteras posibles, buscando la de menor coste (la más corta) que nos lleve a una ciudad en la que no hayamos estado. Para poder añadir una ciudad al recorrido, ésta no puede tener más de dos carreteras que pasen por ella. Una vez encontrada la mejor candidata, como siempre, la agregamos al conjunto solución y la eliminamos del conjunto de candidatos.

Una vez terminado el conjunto de carreteras posibles, nos queda un recorrido que no es cerrado, por lo que tenemos que añadir una última carretera que nos lleve de vuelta a la ciudad inicial, y sumar la distancia de ese recorrido a la distancia total.

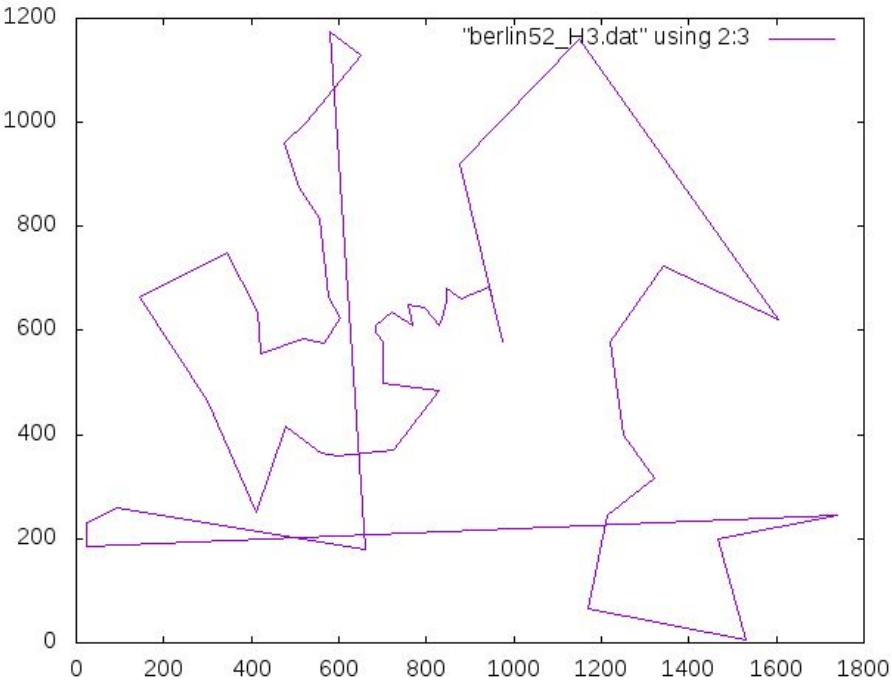
4.4 Pruebas.

Podemos representar gráficamente el recorrido final que ofrece nuestro algoritmo para 4 ciudades de prueba.

Recorrido para bayg29.

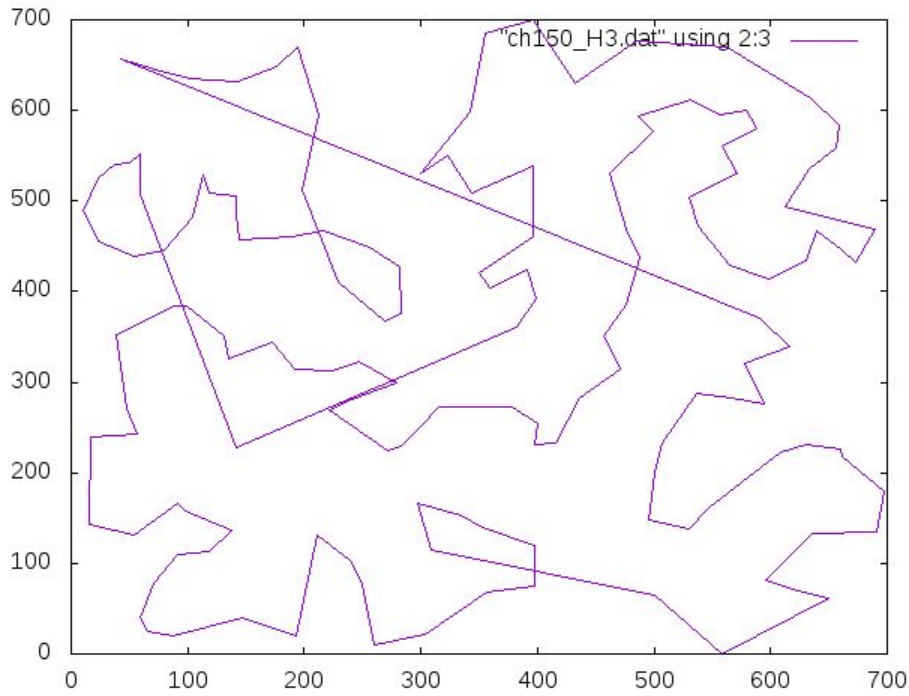


Recorrido para berlin52.



Recorrido para a280.

Recorrido para ch150.



5. Distancias obtenidas y comparación.

Suponiendo que todos los algoritmos empiezan desde el nodo 0, estos son los datos obtenidos para 10 mapas escogidos.

	Vecino más cercano	**Inserción de ciudades**	**Inserción de carreteras**
a280	3023	2955	3096
bayg29	10200	10002	9874
berlin52	8962	8475	9937
ch150	8259	7386	7493
eil76	662	587	583
lin105	20341	17511	16748
kroA100	27772	23793	24158
pa561	18654	16944	17978
pr2392	460671	450936	452803
rd100	9889	9114	9200