Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez

En esta práctica vamos a analizar el uso de los algoritmos "voraces" o "greedy", algoritmos que seleccionan en cada momento lo mejor de entre un conjunto de candidatos, sin tener en cuenta lo ya hecho, para obtener una solución "rápida" al problema.

Vamos a tener dos problemas a los cuales vamos a aplicar esta manera de resolverlos y mediremos su eficiencia teórica.

Una vez diseñado el algoritmo, veremos los resultados de la ejecución y los compararemos con los resultados "óptimos", generados tras resolver el problema de la mejor manera posible.

Recordemos que los algoritmos greedy no aseguran generar soluciones optimales siempre; esta desventaja es una ventaja en problemas en los que es muy difícil alcanzar la solución óptima, apliquemos el algoritmo que apliquemos, como el problema que se propone a continuación. No obstante, veremos que los resultados, a pesar de no ser los óptimos, son bastante eficientes, así como sobretodo el tiempo de ejecución del algoritmo.

1. Problema común (Viajante de comercio)

Como hemos comentado anteriormente, aplicar un algoritmo que nos dé el resultado más óptimo para este problema es bastante complicado y su tiempo de ejecución se incrementaría bastante.

Es por eso por lo que el enfoque greedy es una manera eficiente de solucionar este problema, generando un resultado que no es el óptimo pero se acerca a ello.

El problema se resume en encontrar un circuito hamiltoniano para una serie de puntos, en este caso ciudades, de manera que se recorran todas ellas sin volver a pasar por ninguna, de manera que la distancia total entre estas ciudades, es decir, del circuito, sea la mínima (y así minimizamos el recorrido).

1.1. Algoritmo basado en cercanía

En primer lugar, hemos desarrollado una estrategia basada en encontrar el "vecino más cercano": tomamos una ciudad inicial de manera arbitraria, y buscamos en el vector de ciudades que no se han visitado la ciudad más cercana a esta. Una vez encontrada, se procede a hacer un borrado lógico de la ciudad en el vector, y se procede a encontrar la ciudad más cercana a esta última visitada.

El procedimiento se repite hasta que todas las ciudades se hayan visitado, obteniendo el camino.

Hemos creado también una clase matriz que hemos usado de forma auxiliar para simplificar la parte del código del algoritmo que se detalla a continuación.

Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez

1.1.1. Código del programa

Aquí se muestra la parte del código del programa desarrollado en C++ que contiene el algoritmo principal utilizado.

```
1  //Declaramos los vectores que albergaran los conjuntos
   ↪Candidato y Solucion
2  vector<int> solucion;
3  vector<int> candidatos;
4
5  // Inicializamos el conjunto de candidatos, el rango sera
   ↪[0,15].
6  for(int i = 0; i<dimension;++i){
7      candidatos.push_back(i);
8  }
9
10 // Abergamos la primera ciudad en el conjunto solucion.
11 int i = 0;
12 solucion.push_back(0);
13 candidatos[0] = -1;
14
15 // variable donde guardaremos el indice, es decir, la
   ↪ciudad a donde nos dirigimos.
16 int menor;
17
18 /*CUERPO DEL ALGORITMO:
19 * La idea es encontrar la ciudad mas cercana haciendo uso
   ↪de la matriz de
20 * distancias. Una vez encontrada la ciudad (indice) al que
   ↪ nos dirigimos,
21 * la posicion candidatos[indice] lo hacemos -1 para
   ↪mostrar que esa ciudad
22 * ya la hemos visitado y introducimos el indice en el
   ↪vector de soluciones.
23 *
24 * Para concluir, asignamos el valor del indice a la
   ↪variable i para empezar
25 * de nuevo todo el proceso
26 */
27 while(solucion.size()< dimension   ){
28     vector <double> c;
29     m.get_Fila(i,c);
30     menor =BuscaMenor(c, candidatos);
31     solucion.push_back(menor);
32     candidatos[menor] = -1;
33     i = menor;
34 }
35
36 // Imprimimos el vector solucion teniendo en cuenta que
   ↪para la implementacion
```

Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez


```
// La ciudad numero 1 ha sido el indice numero 0, por lo tanto tenemos que sumar
// 1 a los valores del vector solucion.

for(int i = 0; i< solucion.size();i++){
    cout << solucion [i]  + 1<< " --> ";
}
// Aniadimos la ciudad inicial para indicar que completamos un ciclo.
cout << " 1 " << " FIN.";
```

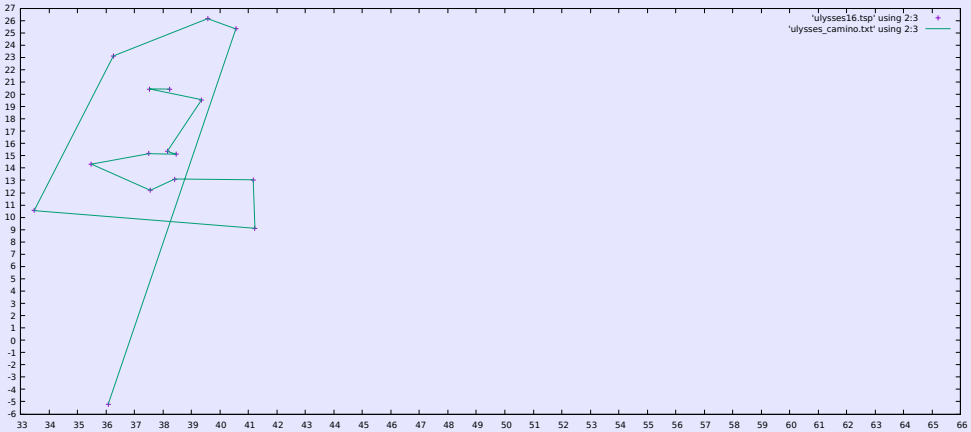
1.1.2. Pseudocódigo

El algoritmo por cercanía en pseudocódigo es el siguiente:

```
N = |V|
S = {primera ciudad de V}
Repetir
    U = Buscar ciudad del conjunto V mas cercana a la ultima ciudad insertada en S
    Eliminar U de V
    Insertar U en S
Hasta que |S| = N
Insertar de nuevo en S la primera ciudad que habiamos insertado al principio
Devolver S
```

1.1.3. Visualización

Aquí podemos observar un resultado de aplicar el algoritmo.



1.1.4. Eficiencia teórica

La eficiencia teórica O(n) depende del número de ciudades que hay. Tomamos, por tanto, $TAM = dimension = n$.

Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez

La eficiencia del algoritmo, en el peor de los casos, es

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{n}$$

$$T(n) = n * n$$

$$T(n) \in O(n^2)$$

Esto es debido a que la función *BuscaMenor* es de tiempo n, y se ejecuta también n veces en el bucle *while* de la línea 30.

Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez

1.2. Algoritmo basado en inserción

Este algoritmo greedy para resolver el problema del viajante de comercio consiste en partir de un circuito inicial. En nuestro caso las ciudades elegidas para el circuito inicial son la más al Este, la más al Oeste y la más al Norte. Una vez escogido el recorrido inicial comienza el algoritmo. Nuestro algoritmo de inserción se basa en insertar en cada iteración la ciudad que menos aumenta el tamaño de este.

1.2.1. Código del programa

Aquí se muestra la parte del código del programa desarrollado en C++ que contiene el algoritmo principal utilizado.

```
//Elegimos el recorrido inicial
int E = 0, O = 0, N = 0;
double mas_al_E = v_coordenadas[0].first;
double mas_al_O = v_coordenadas[0].first;
double mas_al_N = v_coordenadas[0].second;

for(int i=1; i<num_ciudades; ++i){
    if(v_coordenadas[i].second > mas_al_N){
        mas_al_N = v_coordenadas[i].second;
        N = i;
    }
    if(v_coordenadas[i].first > mas_al_E){
        mas_al_E = v_coordenadas[i].first;
        E = i;
    }
    if(v_coordenadas[i].first < mas_al_O){
        mas_al_O = v_coordenadas[i].first;
        O = i;
    }
}

solucion.push_back(O); candidatos[O] = -1;
solucion.push_back(N); candidatos[N] = -1;
solucion.push_back(E); candidatos[E] = -1;

int tam_solucion = solucion.size(); //Tamanio del conjunto
   ↪ solucion

//Comienzo del algoritmo
vector<int>::iterator sol_it, cand_it;  //Iteradores de
   ↪los vectores de candidatos y solucion

//Buscamos la ciudad que menos aumenta el tamanio del
   ↪recorrio
while(tam_solucion < num_ciudades){
    int ciudad_insertada;
    vector<int>::iterator pos_insercion;
```

Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez


```
35      double aumento_minimo = INF;
36
37      for(cand_it = candidatos.begin(); cand_it !=
           ↪candidatos.end(); ++cand_it){
38          if(*cand_it != -1){
39              for(sol_it=solucion.begin(); sol_it!=solucion.
                   ↪end(); ++sol_it){
40                  //Tenemos en cuenta que es un ciclo por lo
                       ↪ que la ciudad siguiente al
41                  //ultimo elemento del conjunto solucion es
                       ↪ el primer elemento
42                  vector<int>::iterator ciudad_siguiente =
                       ↪sol_it;
43                  if(sol_it == --solucion.end())
44                      ciudad_siguiente = solucion.begin();
45                  else
46                      ++ciudad_siguiente;
47
48                  //Calculamos el aumento del recorrido al
                       ↪insertar un elemento de los
                       ↪candidatos
49                  double aumento_distancia = (distancias[*
                       ↪sol_it][*cand_it]+distancias[*cand_it
                       ↪][*ciudad_siguiente]) - distancias[*
                       ↪sol_it][*ciudad_siguiente];
50                  //Nos quedamos con la ciudad que menos
                       ↪aumente el recorrido
51                  if(aumento_distancia < aumento_minimo){
52                      ciudad_insertada = *cand_it;
53                      aumento_minimo = aumento_distancia;
54                      pos_insercion = ciudad_siguiente;
55                  }
56              }
57          }
58      }
59
60      //Insertamos la ciudad
61      solucion.insert(pos_insercion, ciudad_insertada);
62      candidatos[ciudad_insertada] = -1;  //La "eliminamos"
           ↪del vector de candidatos
63      ++tam_solucion;                     //Aumentamos el
           ↪tamanio del vector solucion
64  }
65
66  //Insertamos de nuevo el primer elemento del conjunto
       ↪solucion
67  // ya que es un camino cerrado
68  solucion.push_back(*solucion.begin());
69  ++tam_solucion;
```

Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez


```
70
71 //Mostramos la solucion
72 cout << "Solucion: " << endl;
73
74 for(int i=0; i<tam_solucion; ++i){
75     cout << solucion[i]+1 << " ";
76 }
77 cout << endl;
```

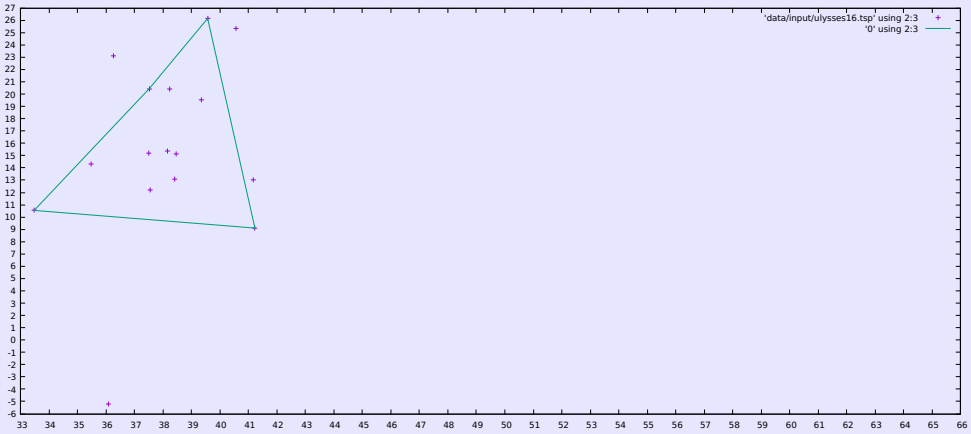
1.2.2. Pseudocódigo

El algoritmo por inserción en pseudocódigo es el siguiente:

```
1  N = |V|
2  S = {ciudad mas al Norte,
3      ciudad mas al Este,
4      ciudad mas al Oeste}
5  Repetir
6      U = Buscar ciudad del conjunto V que menos aumenta la
          ↪distancia de S
7      Eliminar U de V
8      Insertar U en S
9  Hasta que |S| = N
10 Insertar de nuevo en S la primera ciudad que habiamos
      ↪insertado al principio
11 Devolver S
```

1.2.3. Visualización

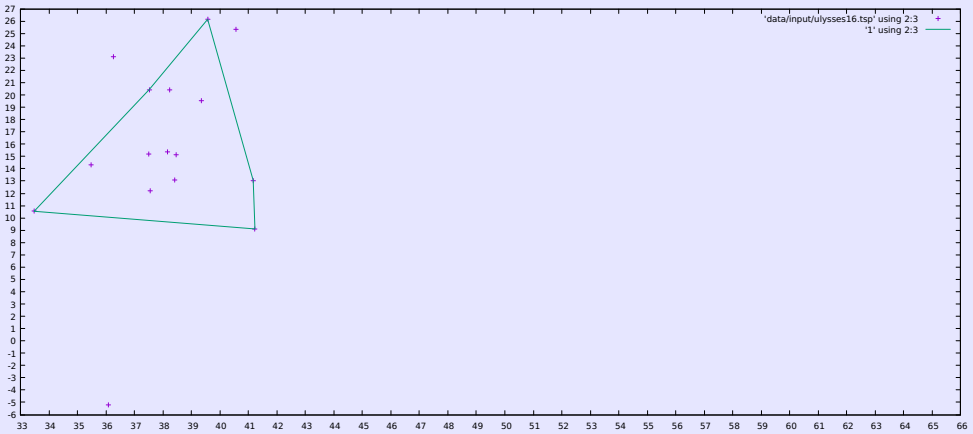
Empezamos seleccionando 3 ciudades distanciadas.



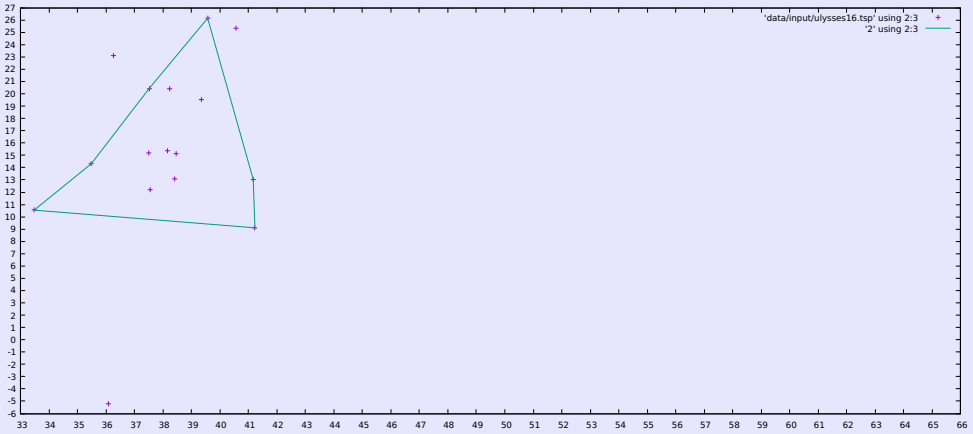
Continuamos añadiendo aquella ciudad que aumente en menor medida el recorrido total.

Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez

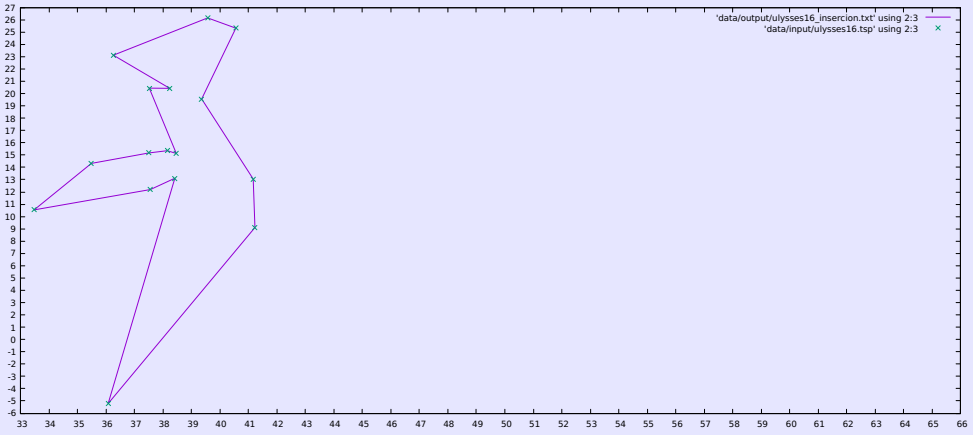
UNIVERSIDAD DE GRANADA



Como podemos comprobar, se añaden las ciudades en el lugar que hagan que el recorrido total sea menor.



Tras terminar obtenemos el recorrido final.



Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez

1.2.4. Eficiencia teórica

Fijándonos en el pseudocódigo, vemos que la búsqueda de la ciudad del conjunto V que menos aumenta la distancia de S es de orden $O(n^2)$ por lo que estaríamos hablando de un algoritmo de orden $O(n^3)$.

Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez

1.3. Algoritmo con otra estrategia

A continuación, se ha implantado una estrategia nueva basada en las aristas para resolver este algoritmo.

1.3.1. Código del programa

Aquí se muestra la parte del código del programa desarrollado en C++ que contiene el algoritmo principal utilizado.

```
//Generamos vector de candidatos
for(int i=0; i<num_ciudades; ++i)
    candidatos.push_back(i);

while(tam_solucion < num_ciudades){
    double mayor = INF * -1; // Valor de la coordenada y de la ciudad que esta mas al Norte
    int ciudad_mas_al_Norte;
    for(int i=0; i<num_ciudades; ++i){
        if ((v_coordenadas[i].second > mayor) && (candidatos[i] != -1)){
            ciudad_mas_al_Norte = i;
            mayor = v_coordenadas[i].second;
        }
    }
    solucion.push_back(ciudad_mas_al_Norte);
    candidatos[ciudad_mas_al_Norte] = -1;
    ++tam_solucion;
}

//Mostramos la solucion
cout << "Solucion: " << endl;

for(int i=0; i<tam_solucion; ++i){
    cout << solucion[i]+1 << " ";
}
cout << endl;
```

1.3.2. Pseudocódigo

1.3.3. Visualización

1.3.4. Eficiencia teórica

Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez

1.4. Comparación de algoritmos

En este apartado, se adjuntan, tras ejecutar los algoritmos y obtener sus resultados en un archivo de texto, la representación gráfica de los recorridos solución de los tres algoritmos.

Como podemos observar, y era de esperar, la solución obtenida en estos tres no es la más óptima (no es el óptimo global), sino el resultado de aplicar una estrategia que minimice la distancia de ciudad en ciudad, por lo que se obtiene un resultado que no es el más óptimo pero es bastante eficiente en relación al resultado que da (que en algunos casos se aproxima al óptimo).

Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez

2. Problema específico

2.1. Ahorro de gasolina

El problema trata de partir de una ciudad y llegar a otra con un vehículo con cierta autonomía pasando por el menor número de gasolineras posibles.

Para entender el algoritmo lo podemos imaginar gráficamente. La autonomía del coche va a ser el radio de la circunferencia de centro la primera ciudad o gasolinera en donde nos encontremos en cada momento.

Dentro de esa circunferencia se encontrarán las gasolineras a las que podemos llegar con la autonomía del vehículo. Solo nos queda elegir a cual de ellas. Muy facil, nos vamos a la gasolinera que este más cerca de la ciudad objetivo.

Así nos vamos moviendo de gasolinera en gasolinera hasta que dentro de nuestra circunferencia se encuentre a la ciudad objetivo.

En el desarrollo de este algoritmo nos encontramos un error en tiempo de ejecución de violación de segmento. Esto se debía a que no hacíamos un clear del vector que contenía las distancias desde la posición actual hasta el resto de gasolineras.

En el código, más concretamente en la funcion "BuscarGasolinera",recopilamos las ciudades que están dentro de la circunferencia en el vector de "indices_posibles_gasolineras" en donde guardamos los índices de las gasolineras.

Después de recopilarlas, buscamos en él, el índice de la gasolinera que minimiza la distancia a la ciudad objetivo y guardamos el índice de aquella que cumple el criterio de optimalidad.

La función devuelve el índice que será añadido al vector solución y borrado de manera lógica del vector de candidatos.

En el caso de que el índice devuelto de la función sea -1 significa que hemos llegado a un punto en el que desde la gasolinera que nos encontramos no podemos ir a ninguna otra con la autonomía dada. Para indicarlo se muestra un mensaje de error y finaliza el algoritmo.

Si el algoritmo finaliza sin mensaje de error significa que se ha conseguido llegar a la ciudad objetivo.

2.1.1. Código del programa

Aquí se muestra la parte del código del programa desarrollado en C++ que contiene el algoritmo principal utilizado.

```
1 int BuscarGasolinera(const int autonomia,const int
    ↪pos_actual, const int ciudad_destino, matriz<double>
    ↪& grafo,  vector<int> &candidatos){
2
3     int parada = -1;
```

NOTA: Se ha añadido aquí una función a la cual hace referencia el algoritmo principal.

Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez


```
    vector<int> indices_posibles_gasolineras;
    double minimo = INF;

    for (int i = 0 ;i <candidatos.size();++i){
        double distancia_actual_candidato = grafo[
            ↪pos_actual][i];
        double distancia_candidato_destino = grafo[i][
            ↪ciudad_destino];
        if((distancia_actual_candidato <= autonomia) && (
            ↪candidatos[i] != -1) && (
            ↪distancia_candidato_destino < minimo)){
            parada = i;
            minimo = distancia_candidato_destino;
        }
    }

    return(parada);
}
```

```
vector<int> candidatos;
vector<int> solucion;
solucion.push_back(ciudad_origen);

for(int i = 0; i< num_ciudades; i++)
    candidatos.push_back(i);

candidatos[ciudad_origen] = -1;
bool fin = false;
pos_actual = ciudad_origen;


while(fin == false){
    if(autonomia >= distancias[pos_actual][ciudad_destino
        ↪]){
        cout << "FIN = DESTINO" <<  endl;
        solucion.push_back(ciudad_destino);
        fin = true;
    }

    else{
        pos_actual = BuscarGasolinera(autonomia,pos_actual
            ↪, ciudad_destino, distancias, candidatos);

        if(pos_actual != -1){
            candidatos[pos_actual] = -1;
            solucion.push_back(pos_actual);
        }
        else{
```

NOTA: Este es el algoritmo principal.

Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez


```
28              cout << "No podemos llegar a ninguna otra
                   ↪gasolinera" << endl;
29              fin = true;
30          }
31      }
32  }
33
34  for (int i = 0; i<solucion.size(); ++i){
35      cout << solucion[i]+1 << " --> ";
36  }
37  cout << "FIN" << endl;
```

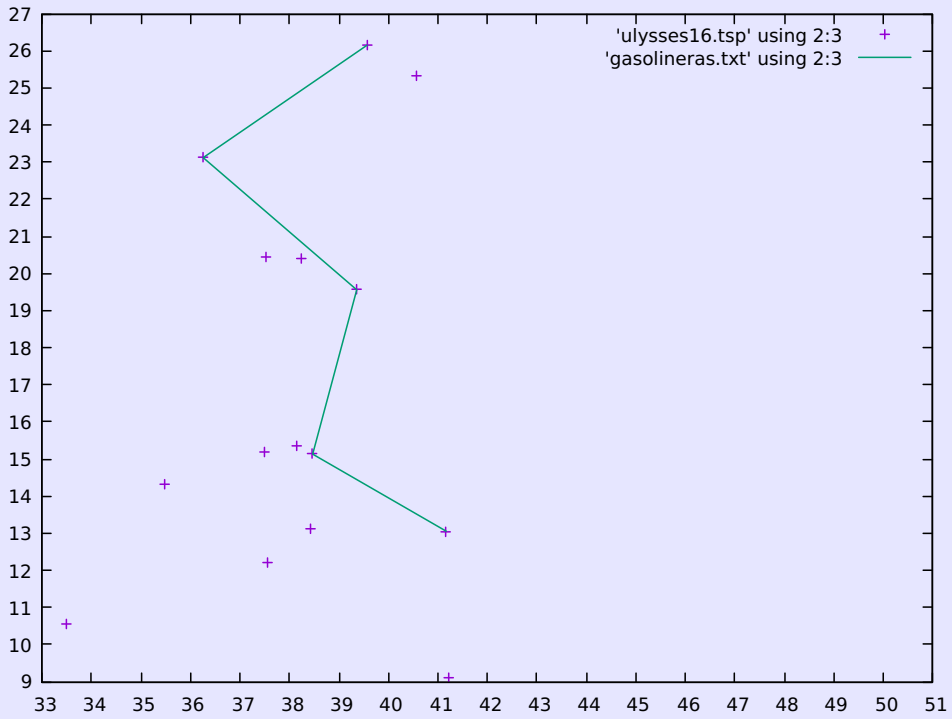
2.1.2. Pseudocódigo

```
1   Mientras no se llegue al destino o a un punto sin salida:
2       Encontrar ciudades posibles con la autonomia;
3       Si podemos ir a gasolineras o a la ciudad objetivo:
4           Si podemos ir a la ciudad objetivo:
5               FIN;
6           Si podemos ir a una o varias gasolineras:
7               Elegir la mas cercana a la ciudad objetivo;
8               Anadir al vector solucion;
9               Posicionarnos en la nueva gasolinera;
10      Si no podemir a ningun lado:
11          FIN;
```

NOTA: Con *ciudades* también se puede entender a las gasolineras.

Adrián Carmona Lupiáñez
Ignacio Sánchez Herrera
Jacobo Casado de Gracia
Jesús José Mª Maldonado Arroyo
Juan Miguel Hernández Gómez

2.1.3. Visualización



2.1.4. Eficiencia teórica

La eficiencia teórica O(n) depende del número de ciudades que hay. Tomamos, por tanto, $TAM = num\_ciudades = n$.

La eficiencia del algoritmo, en el peor de los casos, es

$$T(n) = \sum_{i=1}^{n}\sum_{j=i}^{n}$$

$$T(n) = n * (n - i)$$

$$T(n) \in O(n^2)$$

Para hallar esto nos debemos de fijar en el bucle que comienza en la línea 112 y analizarlo. Nos damos cuenta que, en el peor de los casos, el algoritmo revisa todas las ciudades y escoge la última, y luego vuelve a revisarlas todas y escoger la última, y así consecutivamente.