# PRÁCTICA 3:

## Algoritmos Greedy:

-Minimizando el número de visitas al proveedor

-El problema del viajante de comercio

Componentes Grupo : Antonio Javier Rodríguez Pérez, Alba Moreno Ontiveros, Jesus Mesa González , Sergio Díaz Rueda

# 1.- Minimizando el número de visitas al proveedor

Un granjero necesita disponer siempre de un determinado fertilizante. La cantidad máxima la consume en *r* dias, y antes de eso necesita acudir a una tienda para abastecerse. El problema es que dicha tienda tiene un horario de apertura irregular.
El granjero sabe que días abre, y desea minimizar el número de desplazamientos al pueblo para abastecerse.

Lo primero que hemos hecho ha sido ver si reúne las seis características, que tiene cualquier problema Greedy:

1- Conjunto de candidatos : Los días que abre la tienda
2- Candidatos ya usados : Los días que planificamos ir a la tienda
3- Función solución : Todos los días tenemos fertilizante
4- Criterio de factibilidad : Vamos el mínimo número de días a la tienda
5- Función de selección : El candidato más prometedor será, el día $r = 0$, y la tienda abra, en caso de no abrir la tienda, nos tendremos que remontar en el tiempo hasta encontrar un día que abra.
6- Función objetivo : Minimizar el número de visitas a la tienda

Como podemos comprobar, el problema cumple las seis características necesarias para poder resolverlo con un método Greedy.

Nuestro planteamiento del problema ha sido el siguiente.

Hemos creado una matriz dinámica mes, que se inicializa de forma aleatoria con valores 0 y 1. Esta matriz será el mes sobre el que planificaremos las visitas a la tienda.

Como única condición hemos puesto que el día 1 del mes abra.

```
void CreaMes(Mes &mes){
    MyRandom generador(0,1);
    //Objeto que voy a usar para generar una matriz mes, que me dice los dias que habre o cierra la tienda
    mes.datos = new int *[mes.semanas];
    //las filas son las semanas

    for(int i = 0; i < mes.semanas;i++)
        mes.datos[i] = new int [mes.dias];

    //Relleno el mes con:  0 y 1
    //0 = cerrado   1 = abierto
    for(int semana = 0; semana < mes.semanas; semana++)
        for(int dia = 0; dia < mes.dias; dia++)
            mes.datos[semana][dia] = generador.Next();
    mes.datos[0][0] = 1;
}
```

Una vez tenemos el horario de la tiende ese mes, lo que hicimos fue calcular *r* ,  la cantidad máxima de fertilizante. La vamos a obtener del máximo número de días consecutivos en los que la tienda no abre. Esto lo hacemos con una función:

Componentes Grupo : Antonio Javier Rodríguez Pérez, Alba Moreno Ontiveros, Jesus Mesa González , Sergio Díaz Rueda

```
int CalculoCapacidadMaxima(const Mes &mes){
    int capacidad;
    int maximo = 0;
    int aux = 0;
    int contador = 0;

    for(int semana = 0; semana < mes.semanas; semana++){
        for(int dia= 0 ; dia < mes.dias; dia++){

            if(mes.datos[semana][dia] == 0){
            //compruebo si hay dias consecutivos que no abre
                contador++;
                aux = contador;
            }
            else{
                contador = 0;
                if(maximo < aux){
                    maximo = aux;
                }
            }

        }
    }
    return(maximo);
}
```

Una ves, sabemos la capacidad máxima de fertilizante y los días que abre la tienda, hemos diseñado un algoritmo Greedy que hace lo siguiente;

Recibe com argumentos el mes sobre el que queremos ver los días que abre la tienda, el mes planning que tendrá a 1 los diás que debo ir a la tienda a rellenar el fertilizante y la capacidad máxima.

Hace lo siguiente:

Con un doble bucle for recorro el entero, y todos los días disminuyo la cantidad de fertilizante que tengo, *r*--. *Si* me quedo sin fertilizante y ese día abre la tienda, el mejor caso, relleno *r* y ese día lo marco a 1 en el planning.
Si me quedo sin fertilizante y la tienda no abre, entonces tengo que considerar dos casos:

1) El lunes de esa semana la tienda abre; si ocurre esto entonces lo que hacemos es retroceder en la semana hasta encontrar un día que abra. Dicho día se añade al planning y se rellena *r*.

2)El lunes de esa semana no abre; procedemos como en el caso anterior, pero una vez llegamos al lunes, puesto que no abre, tendremos que irnos a la semana anterior, *semana*--, y empezar a comprobar los días desde el domingo, *dia = 6*.

si no hubiésemos tenido en cuenta, el programa daría un error en tiempo de ejecución.

Componentes Grupo : Antonio Javier Rodríguez Pérez, Alba Moreno Ontiveros, Jesus Mesa González , Sergio Díaz Rueda

```
void CalculaPlanning(const Mes &mes, Mes &planning, int r){

    for(int semana = 0; semana < mes.semanas; semana++){
        for(int dia = 0; dia < mes.dias; dia++){
            r--;

            if(r == 0 && mes.datos[semana][dia] == 1){
                r = CalculoCapacidadMaxima(mes)+1;
                planning.datos[semana][dia] = 1;
            }

            else{
                if(r == 0 && mes.datos[semana][dia] != 1){  //
                    // si el lunes abre me planteo recorrer todos los dias de la semana hacia atras
                    if(mes.datos[semana][0] == 1){

                        while(mes.datos[semana][dia] != 1){
                            dia--;  //retrocedo hasta encontrar un dia que abra

                        }

                        if(r == 0 && mes.datos[semana][dia] == 1){
                            r = CalculoCapacidadMaxima(mes)+1;
                            planning.datos[semana][dia] = 1;
                        }
                    }
                    else{
                        if(mes.datos[semana-1][0] ==1){
                            while(mes.datos[semana][dia] != 1){
                                dia--;  //retrocedo hasta encontrar un dia que abra
                                if(dia == 0 && mes.datos[semana][dia] != 1){
                                    semana--;
                                    dia = 6;
                                }
                            }


                        }
                    }
                }
            }
        }
    }
}
```

Ejemplo de soluón usando el algoirtmo Greedy de arriba:

```
Calendario de apertura de la tienda

 1 1 1 1 0 1 0
 1 0 0 0 1 0 0
 0 0 1 1 1 1 0
 1 0 1 1 0 0 1
*******************************
Capacidad máxima-> 5
El plan de visitas al proveedor es->

 0 0 0 1 0 0 0
 1 0 0 0 1 0 0
 0 0 1 0 0 0 0
 1 0 0 1 0 0 0
*******************************
```

Componentes Grupo : Antonio Javier Rodríguez Pérez, Alba Moreno Ontiveros, Jesus Mesa González , Sergio Díaz Rueda

*r = 5*; el sábado y el domingo de la segunda semana la tienda no abre, junto con el lunes y el martes de la semana siguiente. Dado que no podemos quedarnos sin fertilizante y gastamos el mismo día que compramos, la capacidad máxima será de 5.

La primera semana me quedo sin fertilizante el viernes, sin embargo la tienda no abre, y por tanto deberé comprarlo el jueves. La última semana compro el lunes porque ese día me quedo sin fertilizante y el jueves, ya que viernes la tienda está cerrada.

Para éste problema el algoritmo Greedy siempre encuentra la solución óptima, ya que no hay otro conjunto de días que minimicen las visitas. El mejor caso es que el mismo día que gastemos *r*, la tienda abra en cuyo caso el número de visitas a la tienda es de 28/*r*.

## ***2.- El problema del viajante de comercio***

Un viajante debe recorrer todas las ciudades exactamente una vez regresando al punto de partida, de forma tal que la distancia recorrida sea mínima.

Hacemos como en el ejercicio anterior y reconocemos las 6 características de los problema Greedy:

1- Conjunto de candidatos : Las ciudades que tenemos que recorrer
2- Candidatos ya usados : Las ciudades por las que hemos pasado
3- Función solución : Pasar por todas las ciudades una sóla vez y volver al principio
4- Critero de factibilidad : Seleccionamos las ciudades de forma que la recorremos la menor distancia posible
5- Función de selección : Escoger la ciudad que menos incremente la distancia, según la ciudad en que estemos
6- Función objetivo : Minimizar la distancia recorrida si pasamos por todas las ciudades y volvemos a la nuestra

Como podemos observar cumple las caracteristicas de un problema Greedy y por tanto podemos aplicar un algoritmo para resolverlo, aunque en este caso la solución óptima no la obtenemos siempre, como explicaremos más adelante.

Hemos usado tres algoritmos Greedy a la hora de resolver el ejercicio.

## *Método 1*

Usando la heurística vecino más cercano; dadda una ciudad $v_0$ se añade aquella $v_i$, , que se encuentra más cerca de $v_0$ . El procedimiento lo repetimos hasta que tengamos todas las ciudades visitadas.

Para ello lo primero que hacemos es calcular la matriz de distancias, que contiene las distancias de todas las ciudades entre sí. Creamos también el vector solución, que tendrá las ciudades que sean válidadas según nuestro criterio de selección.

```
double distancias[n_ciudades][n_ciudades];

for(int i = 0; i < n_ciudades; i++)
    for(int j = 0;j < n_ciudades;j++)
        distancias[i][j] = sqrt(pow(ciudad[i][0]-ciudad[j][0],2)+pow(ciudad[i][1]-ciudad[j][1],2));

int solucion[n_ciudades];
bool candidatos[n_ciudades];

//Inicializamos el vector solucion a -1
//Ponemos todos los candidatos a true (disponibles)

for(int i = 0; i < n_ciudades; i++){
    solucion[i] = -1;
    candidatos[i] = true;
}
```

El algoritmo Greedy que hemos diseñado consiste en un ciclo for con el que damos tantas vueltas como ciudades haya.

Empezamos en la primera ciudad, la marcamos como usada y la metemos en el vector solución, *cont* lo usamos para insertar las ciudades en el vector solución.  Comprobamos si tenemos más ciudades que evaluar, en caso afirmativo, entramos en el while.

Cogemos la distancia minima a la primera ciudad, mientras la ciudad no sea candidata y no sea ella misma , buscamos la ciudad más cercana en la matriz de distancias y la añadimos. Con esta ciudad lo que hacemos es calcular la distancia desde ella a todas las ciudades y vemos si es la mínima del conjunto.

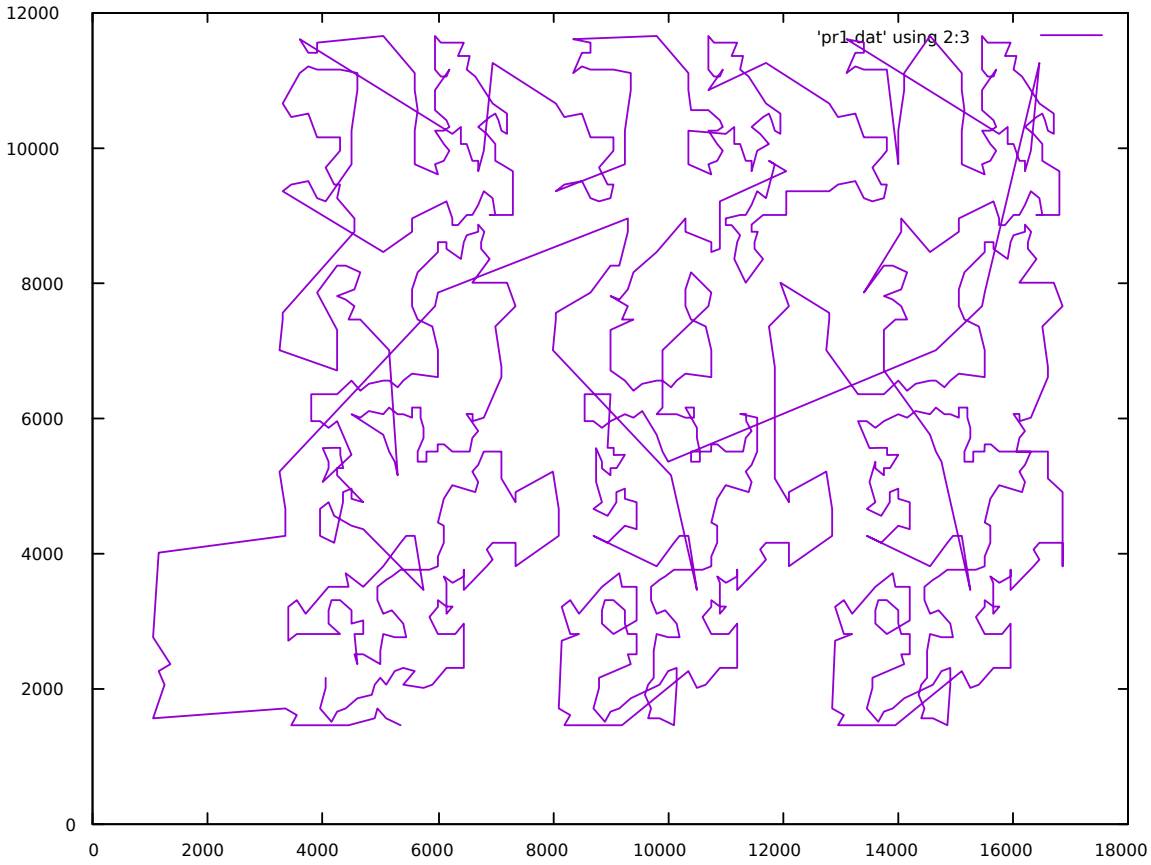
Metemos la ciudad que obtenemos de este último ciclo en el vector solución y volvemos a repetir el ciclo while hasta que nos quedemos sin candidatos.

Componentes Grupo : Antonio Javier Rodríguez Pérez, Alba Moreno Ontiveros, Jesus Mesa González , Sergio Díaz Rueda

```
for(int k = 0;k < n_ciudades; k++){

//Realizamos el algoritmo empezando por todas las ciudades
//Metemos la primera ciudad en solucion y la quitamos de candidatos
    int cont = 0;
    solucion[cont] = k;
    candidatos[k] = false;
    cont++;

    for(int i = 0; i < n_ciudades;i ++)
        if(candidatos[i] == true) vacio = false;


    //Mientras queden ciudades en candidatos
    while(!vacio){

        bool encontrado = false;
        //cogemos la distancia minima la de la primera ciudad
        double dist_min = distancias[solucion[cont-1]][0];
        int min = 0;
        //mientras que la ciudad no este disponible en candidatos y no sea ella misma seguimos buscando
        while(!encontrado){
            if(candidatos[min] && dist_min != 0)
                encontrado = true;

            else{
                min++;
                dist_min=distancias[solucion[cont-1]][min];
            }
        }

        //Ahora que tenemos una distancia disponible buscamos la minima
        for(int i = min + 1; i < n_ciudades; i++)
            if(distancias[solucion[cont-1]][i] < dist_min && candidatos[i] && distancias[solucion[cont-1]][i]!=0){
                min = i;
                dist_min = distancias[solucion[cont-1]][i];
            }

        //metemos la ciudad minima en solución y la quitamos de candidatos
        candidatos[min] = false;
        solucion[cont] = min;
        cont++;


        vacio = true;

        for(int i = 0; i < n_ciudades; i++)
            if(candidatos[i] == true)
            vacio = false;
    }

    //calculamos distancia de la solución obtenida
    int suma = 0;



    for(int i = 0; i < n_ciudades; i++){
        int n1 = solucion[i];
        int n2 = solucion[(i+1) % n_ciudades];
        suma += distancias[n1][n2];
    }

    //si es la primera iteración la guardamos
    if(k == 0){
        distancia_min = suma;
        for(int i = 0; i < n_ciudades; i++)
            solucion_final[i] = solucion[i];
    }
    else{
    //si no es la primera iteración comparamos
        if(suma < distancia_min){
            distancia_min = suma;

            for(int i = 0;i < n_ciudades; i++)
                solucion_final[i] = solucion[i];
        }

    }
    //reiniciamos solución y candidatos
    for(int i = 0;i < n_ciudades;i++){
        solucion[i] = -1;
        candidatos[i] = true;
    }

    cont = 0;
```

Componentes Grupo : Antonio Javier Rodríguez Pérez, Alba Moreno Ontiveros, Jesus Mesa González , Sergio Díaz Rueda

Usando este algoritmo hemos dibujado el siguiente camino hamiltoniano como entrada hemos usado el fichero pr1002.tsp, tiene 1002 ciudades a recorrer.




Nos da una distancia mínima de 311987 unidades.

Componentes Grupo : Antonio Javier Rodríguez Pérez, Alba Moreno Ontiveros, Jesus Mesa González , Sergio Díaz Rueda

## *Método 2*

Aplicamos un algoritmo de inserción, comenzando con un recorrido parcial, que incluye tres ciudades, la más norte, la más al este y la más al oeste. Para este método la función de selección es diferente,  extendemos el recorrido de forma que el impacto sobre el recorrido inicial sea mínimo. Se plantean con este criterio dos dificultades:

1) Qué nodo insertamos después
2)En qué posición

El procedimiento que hemos seguido es el siguiente:

Primero de todo vemos que ciudades van a conformar nuestro recorrido parcial, las marcamos como usadas  y las añadimos a v, vector de usados que tendrá la solución.

Seguidamente creo la matriz de distancias.

```
for(int i = 0; i < n_ciudades; i++){
//lo que primero voy a hacer es la ciudad que hay más al oeste, el menor valor de X que hay

    if(ciudad[i][0] < ciudad[oeste][0])
        oeste = i;


    if(ciudad[i][0] > ciudad[este][0])
            este = i;


    if(ciudad[i][1] > ciudad[norte][1])
            norte = i;
}


usados[oeste] = 1;          // Marco que están siendo usados puesto que forman parte del triangulo inicial.
usados[este] = 1;
usados[norte] = 1;

double **distancia;                         //Matriz de distancias.
distancia = new double*[n_ciudades];

for(int i=0; i<n_ciudades;i++){
    distancia[i]=new double[n_ciudades];
}


//Calculo de distancias
for(int i=0; i < n_ciudades; i++){
    for(int j=0; j<n_ciudades; j++){
        distancia[i][j]= sqrt(pow((ciudad[j][0]-ciudad[i][0]),2)+pow((ciudad[j][1]-ciudad[i][1]),2));
    }
}

v[0]=oeste;
v[1]=norte;
v[2]=este;
```

Hemos usado dos vectores auxiliares, v_aux lo vamos a usar para calcular la longitud de cada ciudad según la posición que ocupe en el vector, v_inicial lo que contiene es el primer circuito que creamos.

Como ya hemos usado tres ciudades tendremos que realizar menos iteraciones, anidamos dos bucles for, el primero hará el número de ciudades iteraciones,  el segundo lo que hace es ver para la posición *i*  que longitud conlleva la inserción de la ciudad *j*, mientras no esté marcada como usada,

Componentes Grupo : Antonio Javier Rodríguez Pérez, Alba Moreno Ontiveros, Jesus Mesa González , Sergio Díaz Rueda

en *v_aux*, el circuito incial, así podremos insertar la ciudad *j* en la posicion *i* minimizando la distancia.

```
int v_aux[n_ciudades];
int v_inicial[n_ciudades];

CopiarVector(v, v_inicial, 3);

for(int h = 3; h < n_ciudades; h++){

    for(int j = 0; j < n_ciudades; j++){

        for(int i = 1; i < (h+1); i++){

            if(!usados[j]){                 //En primer lugar comprobamos que la ciudad a probar no esté ya en el circuito.
                CopiarVector(v_inicial, v_aux, h);
                Insertar(v_aux, j, i, h);

                if(dist_min > DistanciaVector(v_aux, h+1, distancia)){
                    dist_min = DistanciaVector(v_aux, h+1, distancia);
                    CopiarVector(v_aux, v, h+1);
                    aniadir = j;
                }

            }

        }

    }

    CopiarVector(v, v_inicial, h+1);
    usados[aniadir]=1;
    dist_min = 999999;
}
```

Comprovamos si la longitud mínima es mayor que el total de la distancia del circuito. Marcamos j como usada y la añadimos al vector v, solución.

Las funciones auxiliares usadas en el algoritmo de arriba son:

```
double DistanciaVector(int V[],int tamanio, double **distancia){
// Devuelve la distancias del circuito entre las ciudades pasadas en el vector V (si V={1,2,3} devuelve la distancia total del triángulo 1,2,3,1)
    double total = 0;
    total = distancia[ V[0] ][ V[tamanio-1] ];
    for(int i=0; i < tamanio-1 ; i++){
        total += distancia[V[i]][V[i+1]];
    }
    return total;
}

/*****************************************************************************/
/*****************************************************************************/

void Insertar(int v[], int elemento, int pos, int tamanio){               // El vector tendra que tener un espacio al final para que no se pierda el ultimo valor.
    for(int i = tamanio; i >=pos; i--){
        v[i]=v[i-1];
        if(i==pos)
            v[i]=elemento;

    }
}

/*****************************************************************************/
/*****************************************************************************/

void CopiarVector(const int origen[], int destino[], int tamanio){         // Copia la el vector origen en el vector destino.

    for(int i = 0; i< tamanio; i++){
        destino[i] = origen[i];
    }
}
```

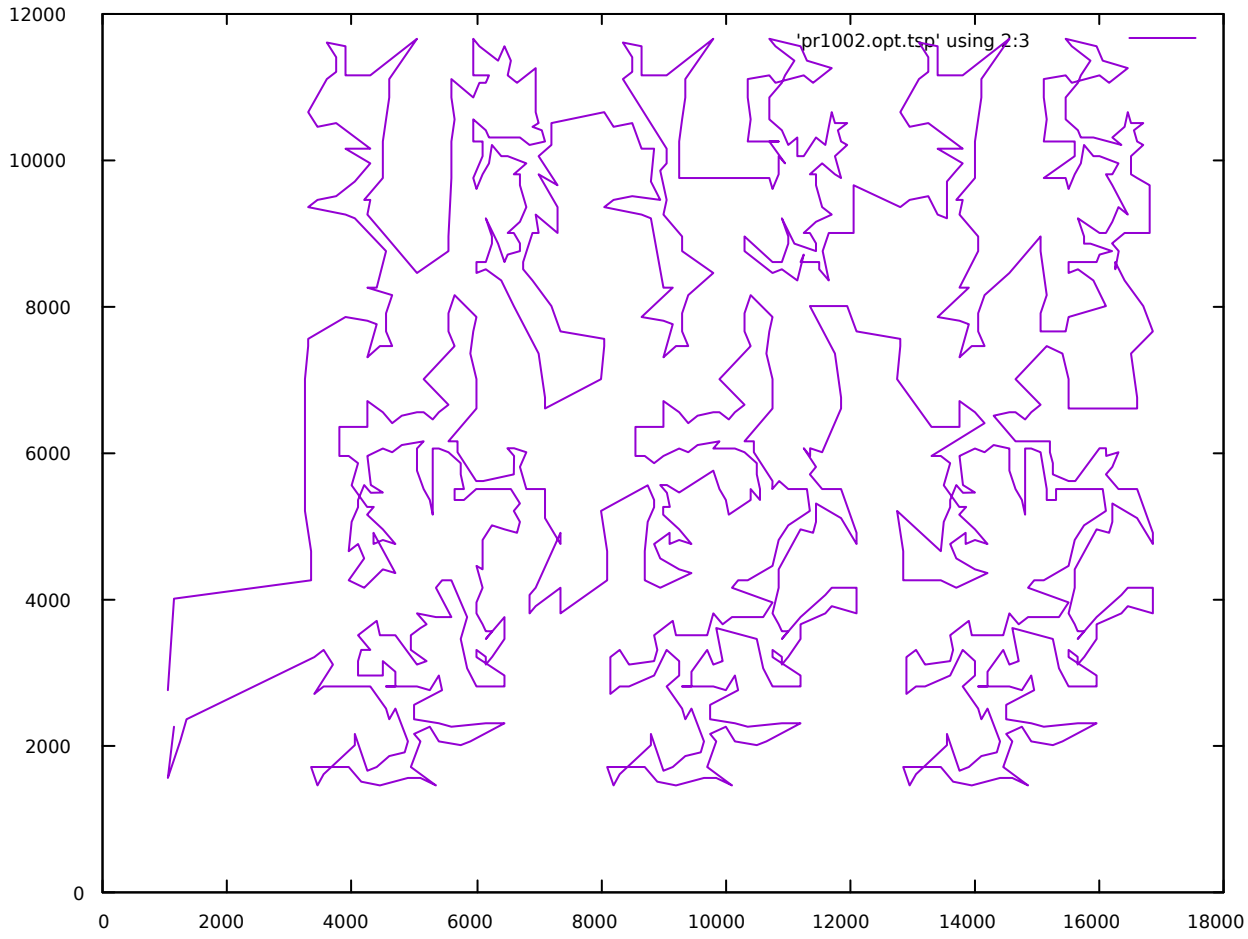
*DistanciaVector* calcula la longitud total del triángulo que forman las ciudad pasadas en *v* , argumento, usando la matriz de distancias.

*Insertar* lo único que hace es añadir un elemento en el vector según la posición que se le ha pasado.

*CopiaVector* copia dos vectores.

Componentes Grupo : Antonio Javier Rodríguez Pérez, Alba Moreno Ontiveros, Jesus Mesa González , Sergio Díaz Rueda

Usando este algoritmo hemos dibujado el siguiente camino hamiltoniano como entrada hemos usado el fichero pr1002.tsp, tiene 1002 ciudades a recorrer.




A simple vista no podemos apreciar una mejoría sin embargo si calculamos la distancia que hemos recorrido obtenemos 295504, son 16483 unidades menos que con el método anterior.

Como podemos observar este método es mejor que el anterior, ya que para el mismo conjunto de ciudades, nos da una longitud mínima menor.

Componentes Grupo : Antonio Javier Rodríguez Pérez, Alba Moreno Ontiveros, Jesus Mesa González , Sergio Díaz Rueda

## *Método 3*

Para este método hemos aplicado un algoritmo Greedy parecido al primero, sólo que en vez de tener un $v_0$ e ir buscando el vecino más cercano, lo que hacemos es con dos extremos coger una ciudad y ver de qué extremo está más cerca.

El procedimiento que hemos seguido es:

Creación de la matriz de distancias, del vector solución y del vector candidatos, inicializado a true. Porque tenemos todas las ciudades disponibles

```
double distancias[n_ciudades][n_ciudades];

for(int i = 0; i < n_ciudades; i++)
    for(int j = 0; j < n_ciudades; j++)
        distancias[i][j] = sqrt(pow(ciudad[i][0]-ciudad[j][0],2)+pow(ciudad[i][1]-ciudad[j][1],2));

int solucion[n_ciudades];
bool candidatos[n_ciudades];

//Inicializamos el vector solucion a -1
//Ponemos todos los candidatos a true (disponibles)
for(int i = 0;i < n_ciudades;i++){
    solucion[i] = -1;
    candidatos[i] = true;
}
```

Insertamos en el vector solución la primera ciudad y la borramos de los candidatos,  aumentamos cont, que es el índice del vector solución.

Mientras que queden ciudades como candidatos,  declaramos una distancia minima que es la de la primera ciudad.  Buscamos la distancia mínima y añadimos la ciudad correspondiente al vector solución, la quitamos de los candidatos. Este procedimiento lo aplicamos si sólo tenemos una ciudad en el vector solución.

Cuando tenemos más de una ciudad en el vector solución, consideramos dos extremos, el extremo 1 que se encuentra en la posición 0 del vector y el extremo 2 que se encuentra en la posicion cont-1 del vector. Calculamos la distancia mínima de cada extremo y nos quedamos con quedamos con la más pequeña de ambas.

Si se trata de una distancia mínima por el extremo 2 la añadimos a la posición cont del vector solución. Pero si se trata de una distancia mínima por el extremo 1, desplazamos las ciudades del vector hacia la derecha y añadimos la ciudad a la posición 0 del vector.

Volvemos a repetir el ciclo while hasta que nos quedemos sin candidatos.

Componentes Grupo : Antonio Javier Rodríguez Pérez, Alba Moreno Ontiveros, Jesus Mesa González , Sergio Díaz Rueda

```
int solucion_final[n_ciudades];
int distancia_min;

for(int k = 0; k < n_ciudades; k++){

    //Realizamos el algoritmo empezando por todas las ciudades
    //Metemos la primera ciudad en solucion y la quitamos de candidatos
    solucion[cont] = k;
    candidatos[k] = false;
    cont++;

    for(int i = 0 ; i < n_ciudades;i++)
        if(candidatos[i] == true)
            vacio = false;


    //Mientras queden ciudades en candidatos
    while(!vacio){

        //si solo tenemos 1 ciudad
        if(solucion[1] == -1){
            bool encontrado = false;

            //cogemos la distancia minima la de la primera ciudad
            double dist_min=distancias[k][0];
            int min = 0;

            //mientras que la ciudad no este disponible en candidatos y no sea ella misma seguimos buscando
            while(!encontrado){
                if(candidatos[min] && dist_min != 0)
                    encontrado = true;

                else{
                    min++;
                    dist_min=distancias[k][min];
                }
            }

            //Ahora que tenemos una distancia disponible buscamos la minima
            for(int i = min + 1; i < n_ciudades; i++)
                if(distancias[k][i] < dist_min && candidatos[i] && distancias[k][i] != 0){
                    min = i;
                    dist_min = distancias[k][i];
                }
            //Metemos la ciudad minima en solución y la quitamos de candidatos
            candidatos[min] = false;
            solucion[cont] = min;
            cont++;

        }
```

Componentes Grupo : Antonio Javier Rodríguez Pérez, Alba Moreno Ontiveros, Jesus Mesa González , Sergio Díaz Rueda

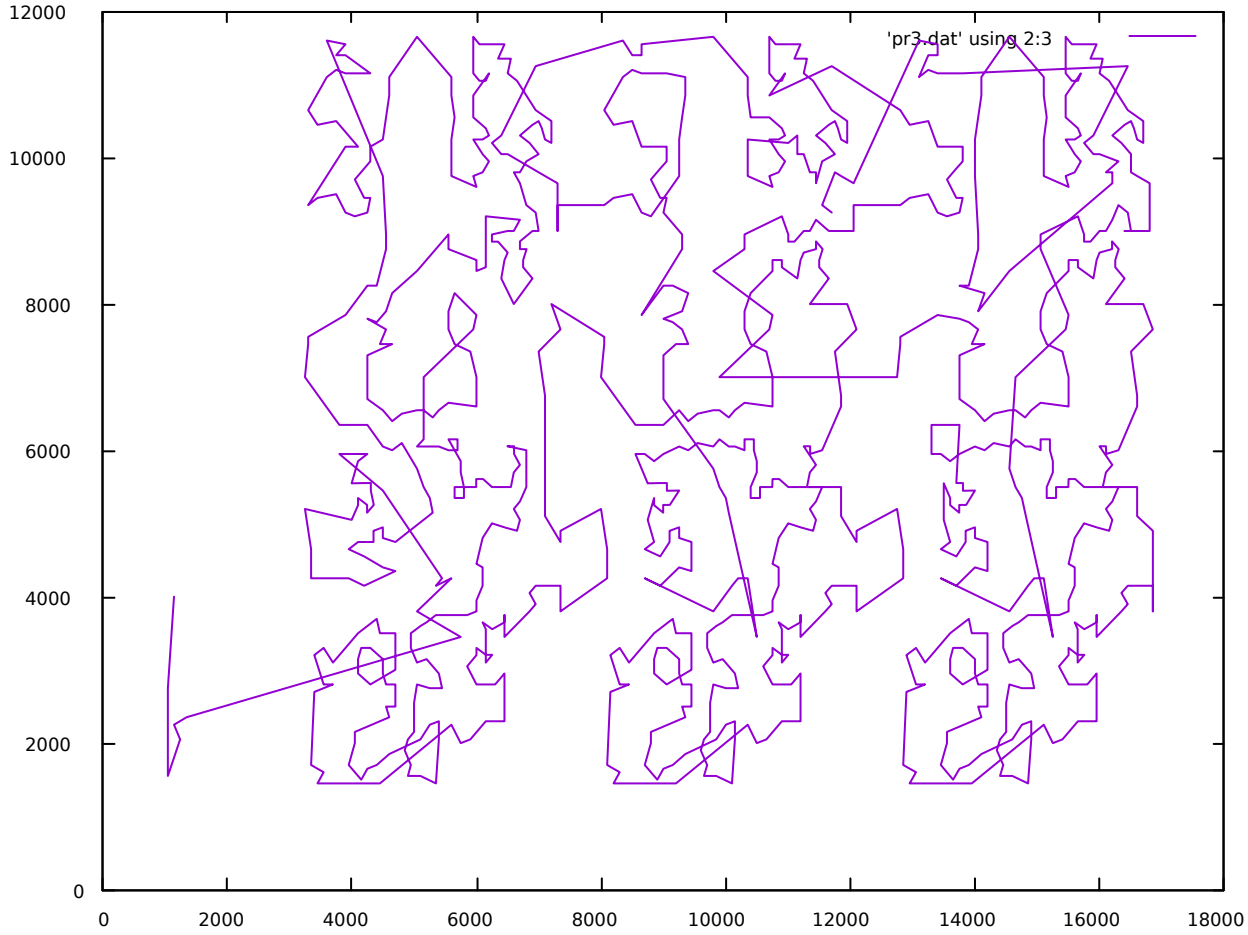
```
else{
    bool encontrado = false;
    int extremo1 = solucion[0],extremo2 = solucion[cont-1];
    double dist_min1 = distancias[extremo1][0];
    int min1 = 0;
    //calculamos la distancia disponible del extremo1
    while(!encontrado){
        if(candidatos[min1] && dist_min1 != 0)
            encontrado = true;

        else{
            min1++;
            dist_min1=distancias[extremo1][min1];
        }
    }

    //calculamos la distancia mínima del extremo1
    for(int i = min1 + 1; i < n_ciudades; i++)
        if(distancias[extremo1][i] < dist_min1 && candidatos[i] && distancias[extremo1][i] != 0){
            min1 = i;
            dist_min1 = distancias[extremo1][i];
        }

    encontrado = false;

    double dist_min2 = distancias[extremo2][0];
    int min2 = 0;

    //calculamos la distancia disponible del extremo2
    while(!encontrado){
        if(candidatos[min2] && dist_min2 != 0)
            encontrado=true;

        else{
            min2++;
            dist_min2=distancias[extremo2][min2];
        }
    }

    //calculamos la distancia mínima del extremo2
    for(int i = min2 + 1; i < n_ciudades; i++)
        if(distancias[extremo2][i] < dist_min2 && candidatos[i] && distancias[extremo2][i] != 0){
            min2 = i;
            dist_min2=distancias[extremo2][i];
        }

    //comparamos y nos quedamos con la menor
    if(dist_min1 < dist_min2){
        for(int i = cont; i > 0; i--)
            solucion[i] = solucion[i-1];

        candidatos[min1] = false;
        solucion[0] = min1;
        cont++;
    }
```

Componentes Grupo : Antonio Javier Rodríguez Pérez, Alba Moreno Ontiveros, Jesus Mesa González , Sergio Díaz Rueda

```
            else{
                candidatos[min2] = false;
                solucion[cont] = min2;
                cont++;
            }

        }

        vacio = true;
        for(int i = 0; i < n_ciudades; i++)
            if(candidatos[i] == true)
                vacio = false;
    }

    //calculamos distancia de la solución obtenida
    int suma = 0;
    for(int i = 0; i < n_ciudades; i++){
        int n1 = solucion[i];
        int n2 = solucion[(i+1) % n_ciudades];
        suma += distancias[n1][n2];
    }

    //si es la primera iteración la guardamos
    if(k == 0){
        distancia_min = suma;
        for(int i = 0;i < n_ciudades; i++)
            solucion_final[i] = solucion[i];

    }

    else{
        //si no es la primera iteración comparamos
        if(suma < distancia_min){
            distancia_min = suma;
            for(int i = 0;i < n_ciudades; i++)
                solucion_final[i] = solucion[i];
        }
    }

    //reiniciamos solución y los candidatos
    for(int i = 0;i < n_ciudades; i++){
        solucion[i] = -1;
        candidatos[i] = true;
    }

    cont = 0;
}
```

Componentes Grupo : Antonio Javier Rodríguez Pérez, Alba Moreno Ontiveros, Jesus Mesa González , Sergio Díaz Rueda

Usando este algoritmo hemos dibujado el siguiente camino hamiltoniano como entrada hemos usado el fichero pr1002.tsp, tiene 1002 ciudades a recorrer.




La distancia recorrida usando este algoritmo es 312038 unidades, este método es peor que los visto anteriormente porque supone un incremento de la distancia en 51 unidades.

El que proporciona una solución más óptima es por tanto el método 2, usando un algoritmo de inserción.

Componentes Grupo : Antonio Javier Rodríguez Pérez, Alba Moreno Ontiveros, Jesus Mesa González , Sergio Díaz Rueda