Práctica 4:

Algoritmos de Vuelta Atrás (Backtracking)

Ramificación y Poda (Branch and Bound)

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

Problema 1: Laberinto

Descripción:

El problema consiste en encontrar la salida de un laberinto. Más concretamente, supondremos que el laberinto se representa mediante una matriz bidimensional de tamaño n × n. Cada posición almacena un valor 0 si la casilla es transitable y cualquier otro valor si la casilla no es transitable. Los movimientos permitidos son a casillas adyacentes de la misma fila o la misma columna. Suponemos que las casillas de entrada y salida del laberinto son la (0,0) y la (n − 1, n − 1) respectivamente.

Objetivo:

Dada una matriz que representa el laberinto, encontrar si existe un camino para ir desde la entrada hasta la salida.

Apartados:

a) Diseñar e implementar un algoritmo vuelta atrás para resolver el problema.
b) Modificar el algoritmo para que encuentre el camino más corto.

Procedimiento:

El backtracking es un algoritmo que utiliza una estrategia de búsqueda en profundidad y busca la mejor combinación de las variables para solucionar el problema. Durante la búsqueda, si se encuentra una alternativa incorrecta, que la casilla sea distinta 0, la búsqueda retrocede hasta el paso anterior y toma la siguiente alternativa. Cuando se han terminado las posibilidades, se vuelve a la elección anterior y se toma la siguiente opción. Si no hay más alternativas la búsqueda falla. De esta manera, se crea un árbol implícito, en el que cada nodo es un estado de la solución.

Restricciones asociadas al problema:

Restricciones explícitas: posibles valores que pueden tomar las “casillas” del laberinto.

$$Casilla_i = \{0,1,2\}$$

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

Donde el 0 significa que es transitable, 1 que no es transitable y el 2 que está marcada.

Restricciones implícitas: la condición que asegura que la n-tuppla que obtenemos de “casillas” se considera solución. Esto es cuando las coordenadas de la casilla solución coinciden con las coordenadas de la casilla donde nos encontramos.

Apartado a)

Hemos implementado una solución recursiva del algoritmo Backtracking.

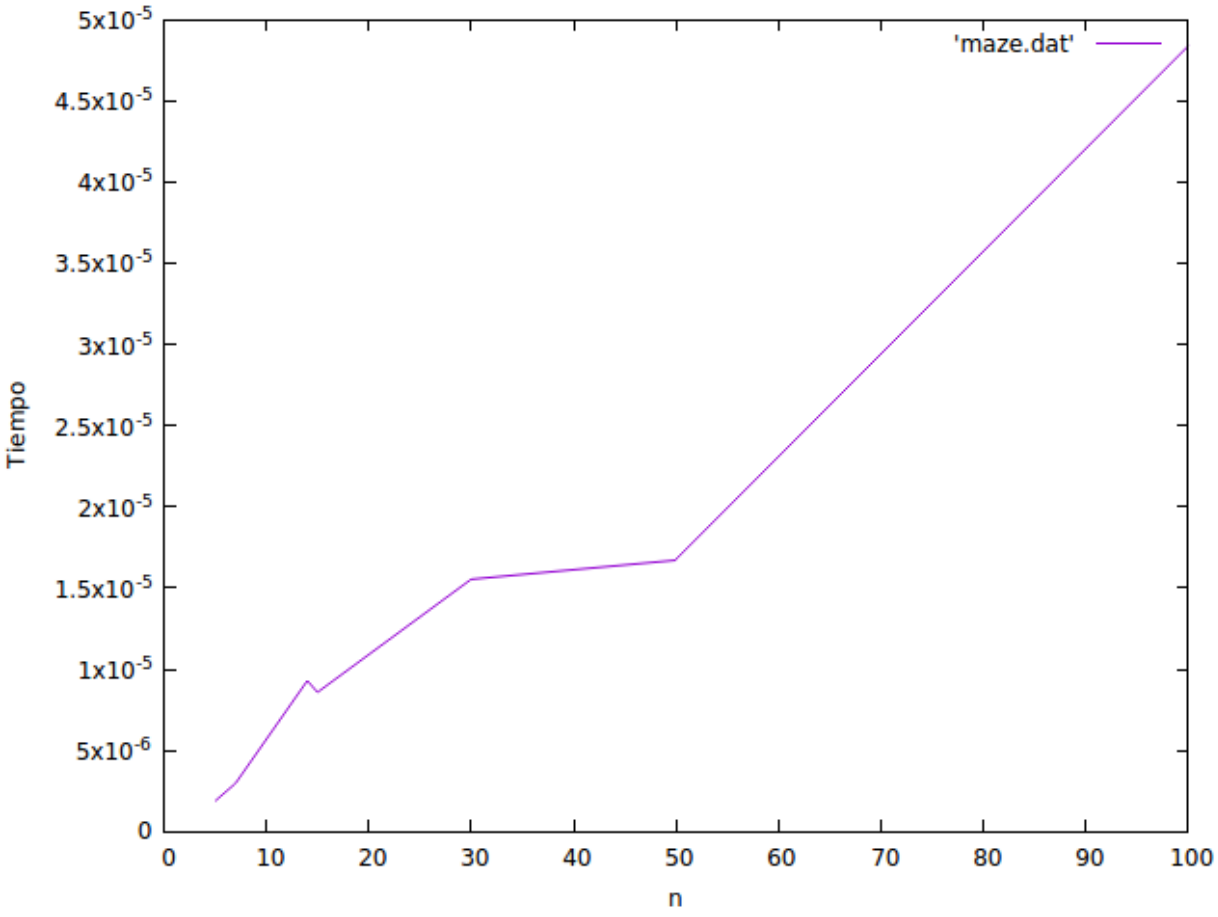
```
37
38  bool HaySalida(int **&lab, int filFin, int colFin, int i , int j, bool **&camino){
39
40      if(i == filFin && j == colFin){
41          solucion = true;
42          camino[i][j]=1;
43      }
44      else{
45          lab[i][j] = 2;
46
47          if(en_limites(i-1,j,filFin, colFin))
48          if (lab[i-1][j] == 0 && !solucion){     //arriba
49              if(HaySalida(lab, filFin, colFin, i-1,j,camino))
50                  camino[i][j]=1;
51          }
52
53          if(en_limites(i,j+1,filFin, colFin))
54          if(lab[i][j+1] == 0 && !solucion){      //derecha
55              if(HaySalida(lab, filFin, colFin, i,j+1,camino))
56                  camino[i][j]=1;
57
58          }
59
60          if(en_limites(i+1,j,filFin, colFin))
61          if (lab[i+1][j] == 0 && !solucion){     //abajo
62              if(HaySalida(lab, filFin, colFin, i+1,j,camino))
63                  camino[i][j]=1;
64          }
65
66
67          if(en_limites(i,j-1,filFin, colFin))
68          if (lab[i][j-1] == 0 && !solucion){     //izquierda
69              if(HaySalida(lab, filFin, colFin, i,j-1,camino))
70                  camino[i][j]=1;
71          }
72
73
74
75
76      }
77      return solucion;
78  }
79
```

Recibe:
El laberinto, las coordenadas de la casilla de salida, la posición inicial y la matriz booleana camino, que será donde se almacene el camino solución del laberinto si es que existe.

Primero comprobamos el caso base, si se ha llegado al final del laberinto. Si no se da entonces marcamos la casilla en la que nos encontramos, para no avanzar por ella, y nos metemos en las condiciones.
Al tratarse de una función booleana, si el camino que seguimos lleva a un rincón sin salida, donde no hay movimientos posibles, devuelve false y "retrocedemos" hasta una casilla que tenga alguna cercana sin marcar y sin muro.

En el peor de los casos la eficiencia teórica del algoritmo, dependiendo de las "casillas" transitable, sería $O(p(n)2^n)$, siendo p un polinomio en n.

Vamos a hacer un cálculo empírico, obtenemos un tabla con los tiempos de ejecución para distintos n y realizamos la gráfica con GNUPLOT.



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

n	Tiempo
5	1.881e-06
7	2.968e-06
14	9.282e-06
15	8.578e-06
30	1.5532e-05
50	1.6704e-05
100	4.8317e-05

No hemos podido realizar un ajuste muy fiable para el algoritmo que hemos usado, ya que T(n), depende de 4 factores principales:

1.- el tiempo que tarda en buscar un nodo válido

2.- el número de valores que tiene que comprobar, que validen las restricciones explícitas

3.- el tiempo de checkeo de las funciones de acotación

4.- el número de nodos que satisfacen las restricciones implicitas

Por ejemplo: Si tenemos un laberinto generado aleatoriamente, con un tamaño de 5x5 la salida que obtenemos es:

```
0 1 0 1 1
0 0 0 1 1
0 0 0 0 0
0 1 0 1 0
1 1 0 0 0

EL LABERINTO TIENE SOLUCION
2 1 0 1 1
2 0 0 1 1
2 2 2 0 0
2 1 2 1 0
1 1 2 2 0

Camino a seguir para salir del laberinto

1 0 0 0 0
1 0 0 0 0
1 1 1 0 0
0 0 1 0 0
0 0 1 1 1

Tamaño del camino -->   9
```

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

La distancia que obtenemos en este caso es óptima, sea cual es orden de los condicionales.

Con un 2 podemos ver las casillas por las que hemos pasado y como hemos “retrocedido” en la coordenada (3,1).

Como para resolver el laberinto se empieza en la casilla (0, 0) y se termina en la de la esquina inferior derecha esto significa que siempre buscaremos una salida que vaya hacia abajo a la derecha, es decir hacia el sureste.
Esto se nos ha ocurrido aplicarlo al orden de las condiciones en la función para que compruebe las casillas hacia abajo y hacia la derecha en primer lugar, lo que nos dará por norma general por lo tanto la solución optima (el camino más corto).

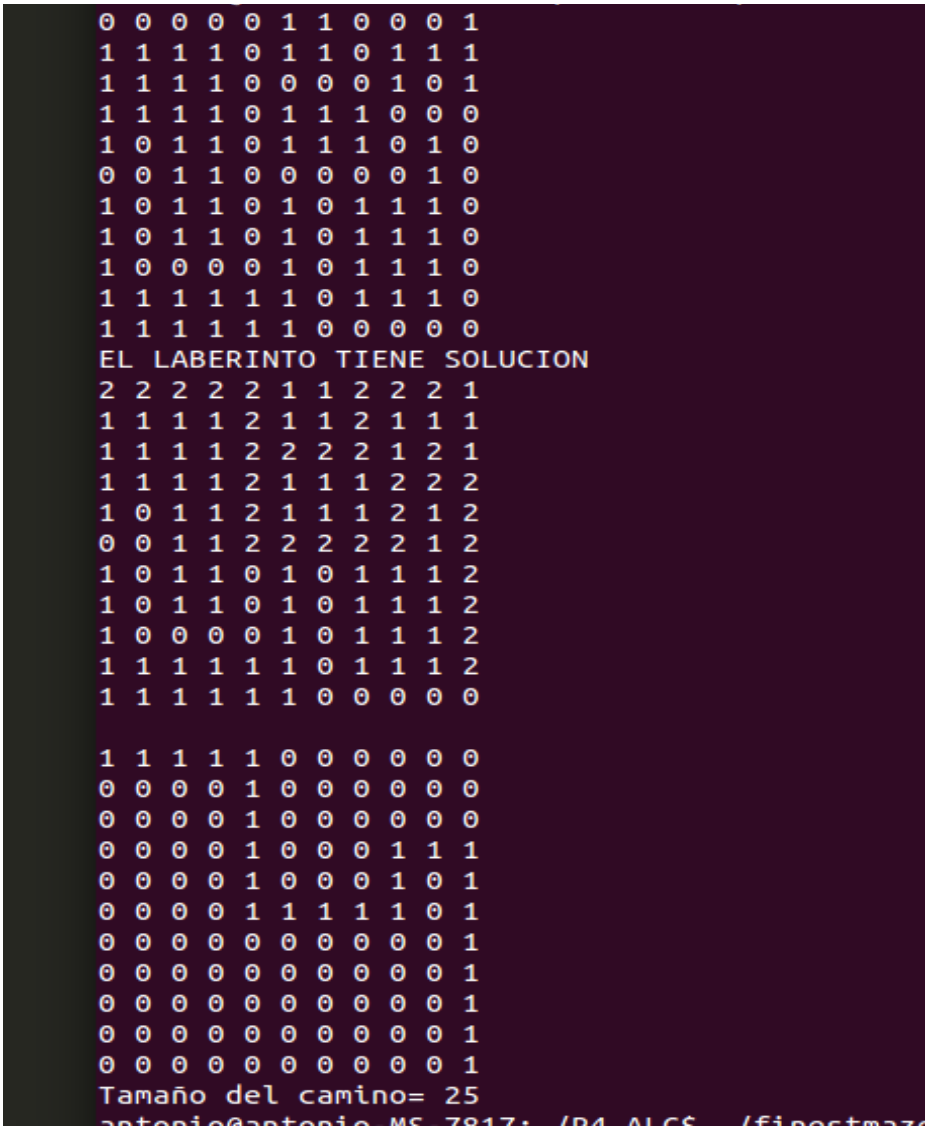
Esta diferencia se puede apreciar en las imagenes a continuacion con un laberinto de prueba de 11x11 con dos caminos posibles por los que se puede llegar a la solución.

En primer lugar este sería el cambio en el orden de las condiciones:

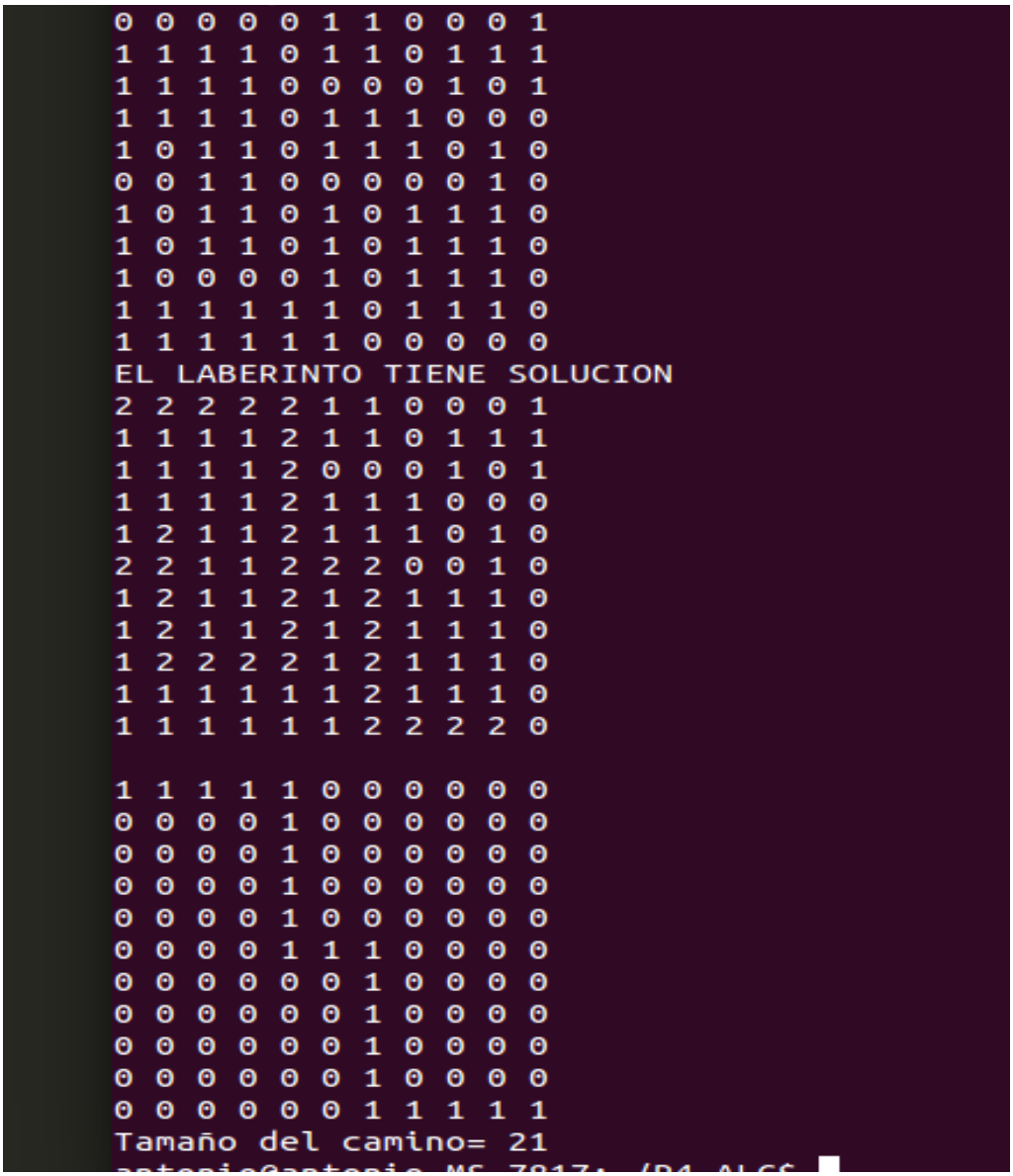
```
bool HaySalida(int **&lab, int filFin, int colFin, int i , int j, bool **&camino){

    if(i == filFin && j == colFin){
        solucion = true;
        camino[i][j]=1;
    }
    else{
        lab[i][j] = 2;
        if(en_limites(i+1,j,filFin, colFin))
        if (lab[i+1][j] == 0 && !solucion){     //abajo
            if(HaySalida(lab, filFin, colFin, i+1,j,camino))
                camino[i][j]=1;
        }


        if(en_limites(i,j+1,filFin, colFin))
        if(lab[i][j+1] == 0 && !solucion){      //derecha
            if(HaySalida(lab, filFin, colFin, i,j+1,camino))
                camino[i][j]=1;

        }


        if(en_limites(i,j-1,filFin, colFin))
        if (lab[i][j-1] == 0 && !solucion){     //izquierda
            if(HaySalida(lab, filFin, colFin, i,j-1,camino))
                camino[i][j]=1;
        }


        if(en_limites(i-1,j,filFin, colFin))
        if (lab[i-1][j] == 0 && !solucion){     //arriba
            if(HaySalida(lab, filFin, colFin, i-1,j,camino))
                camino[i][j]=1;
        }

    }
    return solucion;
}
```

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

El laberinto resuelto con el primer código(nos da una longitud del camino = 25):



Y el laberinto resuelto con el cambio en el código(nos da una longitud del camino = 21 el cuál es el camino mas corto):



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

Problema 2: Viajante de comercio

Descripción:

El problema del viajante de comercio ya se ha comentado y utilizado en la práctica anterior, donde se estudiaron métodos para encontrar soluciones razonables (no óptimas necesariamente) a este problema. Si se desea encontrar una solución óptima es necesario utilizar métodos más potentes (y costosos).

Para emplear un algoritmo de ramificación y poda es necesario utilizar una cota inferior: un valor menor o igual que el verdadero coste de la mejor solución (la de menor coste) que se puede obtener a partir de la solución parcial en la que nos encontremos.

Procedimiento:

La solución es una n-tuppla, un vector, de n-ciudades ordenadas en el orden a recorrer.
Vamos a explicar el procedimiento usando como esquema la función main:

```
int main(int argc, char * argv[])
{
    map<int, pair<double, double> >  m;//un mapa con las posiciones y las dimensiones.
    vector< vector<int> >  matriz;//creamos una matriz con la distnacia entre ciudades
    vector<int> ciu;//vector
    vector<int> menor_arista;
    int tam = 0,distancia=0;
    //--------------------------------------------------------------------
    //calcualr matriz, menor_arista y m.
    string fp;
    if (argc<2) {
        cout << "Error Formato:  tsp puntos" << endl;
        exit(1);
    }
    CLParser cmd_line(argc,argv,false);
    fp = cmd_line.get_arg(1);
    leer_puntos(fp,m);
    tam = m.size();
    matriz.resize(tama, vector<int>(tama,-1));
    calcular_matriz(m,matriz);
    menor_arista = calcularMenorArista(matriz);
    //--------------------------------------------------------------------------
    Solucion s = Branch_and_Bound(matriz,menor_arista,tama);
    //s.mostrar();
    ciu=s.devolverSolucion();
    distancia=s.getDistancia();
    mostrar_resultado(ciu,m);//añade la ciduad 1
    cerr<<"distancia: "<<distancia<<endl;
    cerr<<endl;
    return 0;
}
```

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

Una vez leemos los datos y hacemos las comprobaciones pertinentes, procedemos a realizar el cálculo de la matriz de distancias, la menor arista y el mapa de los puntos.

Una vez hecho esto llamamos al método Branch and Bound pasandole la matriz de distancias, la menor arista y el tamaño de la matriz.

```
//////////////////////////////////////////////////////////////
Solucion Branch_and_Bound(const vector< vector<int> > & matriz,const vector<int> & arista_menor,int tama)
{
    vector<int> aux;
    priority_queue<Solucion> Q;
    Solucion n_e(tama), mejor_solucion(tama) ; //nodo en expansion
    mejor_solucion.greedy(matriz);//calculamos la cota por greedy
    int CG = mejor_solucion.getDistancia(); // Cota Global
    int distancia_actual=0;
    Q.push(n_e);
    while ( !Q.empty() && (Q.top().CotaLocal() < CG) ) {
        n_e = Q.top();
        Q.pop();
        aux=n_e.resto_ciudades();
        for(int i=0; i<aux.size(); i++) {
            n_e.anadirciudad(aux[i],matriz);//añadimos ciudad y le sumamos distancia
            n_e.quitarcidudadRestante(aux[i]);//quietamos la ciudad de ciudadrestante
            if ( n_e.EsSolucion(tama) ) {
                distancia_actual = n_e.Evalua(matriz);
                if (distancia_actual < CG) {
                    CG = distancia_actual;
                    mejor_solucion = n_e;
                }
            }
            else {
                n_e.calcularCotaLocal(arista_menor);
                if (n_e.CotaLocal()<CG ) {
                    Q.push( n_e );
                }
            }
            n_e.quitarciudad(matriz);//la ultima que se ha añadido
            n_e.anadircidudadRestante(aux[i]);//añadimos la ultima que se quito
        }
    }
    return mejor_solucion;
}
```

El método hace lo siguiente:

Mientras la lista no este llena, con todas las ciudades, y el valor de la cota local sea menor que el de la cota que habiamos calculado con Greedy hace:

1- Inserta el nodo en expansión Q, la lista, avanza al siguiente nodo y quita del vector de ciudades la que hemos insertado en Q.

2- Para las ciudades restantes en el vector, añadimos la ciudad a la lista que teniamos y esta es nuestro nuevo nodo en expansión. Lo quitamos del vector de candidatos.

Vemos si la lista que tenemos tiene el mismo numero de ciudades que el mapa que queremos recorrer.
Si no se da entonces seguimos cogiendo ciudades, comprobando para todas que la cota local sea la mínima posible.
Evaluamos si la distancia de las ciudades que tenemos en la lista es menor que la cota global. Si es el caso entonces la mejor solución pasa a ser la lista que teniamos.

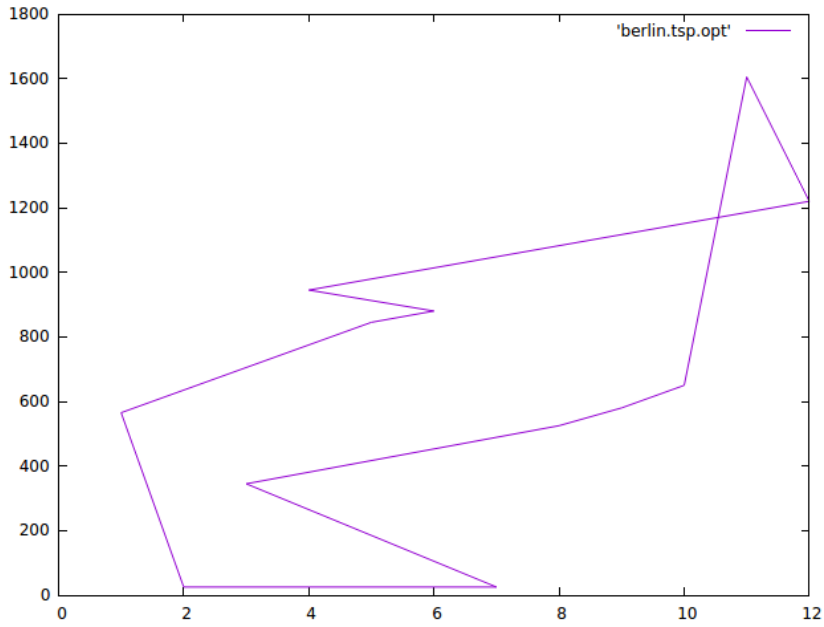
Nosotros decidimos que la cota inferior que vamos a usar es la distancia que obtenemos al aplicar un Algoritmo Greedy.

Devolvemos la mejor solución.

El resto de funciones y métodos del main son para realizar cálculo de distancias o tratar los datos.

Por ejemplo: Para el archivo berlin12.tsp obtenemos los siguientes resultados

```
 1  565 575
 5  845 655
 6  880 660
 4  945 685
12  1220 580
11  1605 620
10  650 1130
 9  580 1175
 8  525 1000
 3  345 750
 7  25 230
 2  25 185
 1  565 575
```

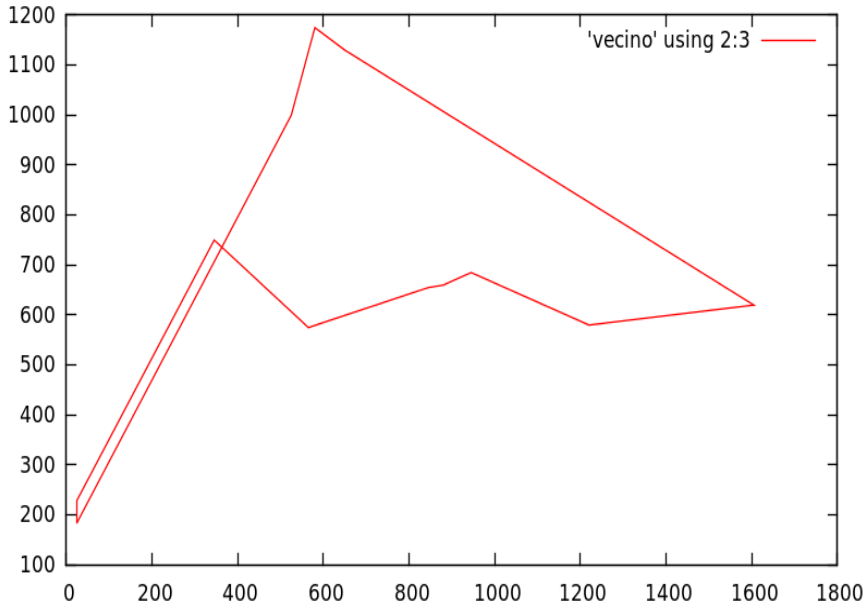


Siendo la distancia total recorrida de 4053 unidades.

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

En contraste el resultado que obtenemos con el algoritmo del vecino más cercano de la práctica anterior con estos mismos datos es:

```
DIMENSION: 12
2 25 185
7 25 230
3 345 750
1 565 575
5 845 655
6 880 660
4 945 685
12 1220 580
11 1605 620
10 650 1130
9 580 1175
8 525 1000
```



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