Taller 5: Grafos

Carlos Andrés Carrero Sandoval
Ingeniería de Sistemas
Pontificia Universidad Javeriana
Bogotá, Colombia
carlos.carrero@javeriana.edu.co

Édgar Julián González Sierra
Ingeniería de Sistemas
Pontificia Universidad Javeriana
Bogotá, Colombia
ejgonzalez@javeriana.edu.co

Daniel Santiago Silva Gómez
Ingeniería de Sistemas
Pontificia Universidad Javeriana
Bogotá, Colombia
silvag-daniels@javeriana.edu.co

***Abstract*— This programming assignment involves implementing a graph and its fundamental algorithms to optimize the operation of a drilling machine in a computer circuit manufacturing facility. The machine operates by positioning printed circuits under a drill sequentially through a conveyor belt, drilling holes in predetermined locations. The program reads input data specifying hole coordinates for each circuit and organizes them to minimize the drilling distance. The output is a file with the reorganized hole sequence for each circuit. The algorithm ensures the drill starts at the (0,0) position, drills each circuit, and returns to the initial position. Statistics such as the number of circuits, holes per circuit, and total drilling distance are reported.**

Keywords— Graphs, Optimization, Circuit Manufacturing, Distance Minimization

I. INTRODUCCIÓN

Este informe aborda la optimización del proceso de perforación en una fábrica de circuitos para computadoras mediante el uso de estructuras de grafos y algoritmos asociados. El desafío consiste en desarrollar un programa que organice los agujeros en los circuitos impresos, minimizando la distancia recorrida por el taladro. La máquina, operando secuencialmente sobre una correa transportadora, perfora los agujeros en ubicaciones predeterminadas. El programa, ejecutado a través de la línea de comandos, procesa la información de entrada que describe las coordenadas de los agujeros en cada circuito, generando un archivo de salida con la secuencia óptima de perforación para cada circuito.

Este informe detallará la implementación del programa, destacando los resultados obtenidos en términos de lectura correcta del archivo de entrada y la distancia recorrida por el taladro en cada circuito después de haber hallado el recorrido más corto posible.

II. DIAGRAMA DE TADS

A continuación, se presenta el diagrama de los TADs utilizados para realizar el Taller 5 – Grafos, el cual consiste básicamente en el main del programa y el TAD Grafo, el cual se explicará más adelante.

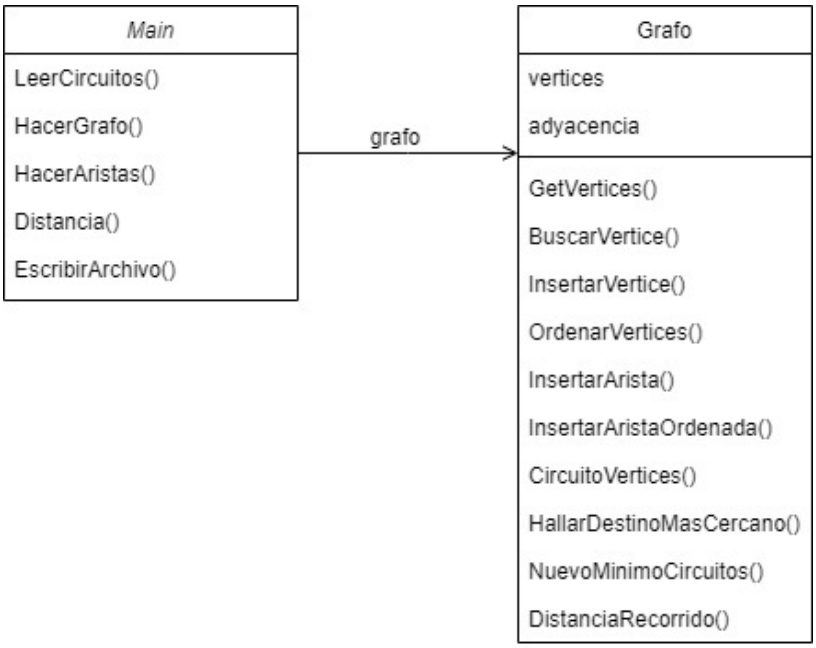

Fig. 1. Diagrama de TADs para el Grafo. Fuente: Elaboración propia.

III. DISEÑO DE TADS

1. *TAD Grafo*

Conjunto mínimo de datos:

- *vertices: vector del tipo de datos que almacena el grafo, almacena los vértices del grafo.*
- *adyacencia: vector de listas de pares, almacena la lista de aristas para cada vértice en pares, siendo el primer valor del par el vértice destino al que está conectado y el peso o distancia que tiene llegar a ese vértice destino.*

Comportamiento del objeto:

- *GetVertices():Devuelve los vértices del grafo.*
- *InsertarVertice(vertice): Inserta un vértice en el grafo si no existe previamente.*
- *BuscarVertice(vertice): Busca un vértice en el grafo y retorna su posición.*
- *OrdenarVertices(): Ordena el vector de vértices del grafo en forma ascendente.*
- *InsertarArista(origen, destino, peso): Inserta una arista con el destino y peso dados en la lista de adyacencia en la posición correspondiente al vértice de origen.*
- *InsertarAristaOrdenada(aristaNueva, posicionOrigen): Se asegura de insertar la arista de forma que la lista de aristas del vértice de origen quede ordenada ascendentemente.*
- *CircuitoVertices(origen): Retorna un vector con el orden óptimo de vértices para recorrer un circuito iniciando y terminando en el vértice de origen dado.*
- *NuevoMinimoCircuitos(aristaActual, pesoMinimo, verticeMinimo): Actualiza el peso mínimo y el vértice mínimo comparando una arista dada con el peso mínimo y en segunda instancia el vértice mínimo.*
- *HallarDestinoMasCercano(posicionActual, verticesIda, verticesVuelta, pesoMinimo, verticeMinimo, verticeBaneado): Encuentra el destino más cercano para un vértice en un circuito, teniendo en cuenta los vértices ya visitados y un vértice que no se tiene en cuenta en la búsqueda del más cercano.*
- *DistanciaRecorrido(verticesRecorrido): Calcula la distancia total recorrida según el vector de vértices recorridos.*

IV. DESARROLLO

1. *Descripción de la función principal (main)*

Lee la información de los circuitos del archivo de entrada, itera sobre cada circuito, construye un grafo y obtiene la secuencia óptima de perforación utilizando las funciones hacerGrafo y circuitoVertices del TAD Grafo. Luego calcula la distancia total recorrida por el taladro en el circuito utilizando la función distanciaRecorrido. Finalmente imprime por pantalla la información requerida y genera el archivo de salida con los agujeros ordenados.

2. *TODO #1: recibir correctamente los archivos TXT.*

```
// TODO #1: recibir correctamente los archivos TXT
if (argc < 3) {
    std::cerr << "Uso: ./" << argv[0] << " archivo_entrada.txt archivo_salida.txt" << std::endl;
    return(-1);
}
std::string archivoEntrada(argv[1]);
std::string archivoSalida(argv[2]);
if (archivoEntrada.substr(archivoEntrada.length() - 4, archivoEntrada.length()) != ".txt") {
    perror("El archivo de entrada debe contener extension '.txt'");
    exit(-1);
}
if (archivoSalida.substr(archivoSalida.length() - 4, archivoSalida.length()) != ".txt") {
    perror("El archivo de salida debe contener extension '.txt'");
    exit(-1);
}
```

Fig. 2. TODO #1. Fuente: Elaboración propia.

Se valida que se ingresen el archivo de entrada y de salida (ambos en formato TXT), como argumentos de programa.

3. *TODO #2: leer correctamente el archivo TXT de entrada.*

```cpp
// TODO #2: leer correctamente el archivo TXT de entrada
std::list<std::list<std::pair<float, float>>> leerCircuitos(const std::string& filename) {
    std::ifstream input(filename.c_str());
    if (!input) {
        perror("No se pudo abrir el archivo especificado");
        exit(-1);
    }

    int numeroDeCircuitos = 0, numeroAgujeros = 0;
    float coordenadaX, coordenadaY;
    std::list<std::list<std::pair<float, float>>> circuitos;
    input >> numeroDeCircuitos;
    for (int i = 0; i < numeroDeCircuitos; i++) {
        input >> numeroAgujeros;
        std::list<std::pair<float, float>> circuito;
        for (int j = 0; j < numeroAgujeros; j++) {
            input >> coordenadaX >> coordenadaY;
            //Crea y añade la coordenada del agujero al circuito
            std::pair<float, float> coordenada(coordenadaX, coordenadaY);
            circuito.push_back(coordenada);
        }
        // Añade el circuito a la lista de circuitos
        circuitos.push_back(circuito);
    }
    //Cierra el archivo
    input.close();
    return circuitos;
}
```

Fig. 3. TODO #2. Fuente: Elaboración propia.

Se guarda en una lista de lista de coordenadas, es decir, en una lista de circuitos, la información contenida en el archivo TXT de entrada.

4. *TODO #3: construir correctamente los vértices de los grafos.*

```cpp
void hacerGrafo(Grafo<std::pair<float, float>>& grafo, std::list<std::pair<float, float>> circuito) {
    // TODO #3: construir correctamente los vértices de los grafos
    grafo.insertarVertice(std::make_pair(0, 0));
    std::list<std::pair<float, float>>::iterator it = circuito.begin();
    for (; it != circuito.end(); it++) {
        grafo.insertarVertice(*it);
    }
    // TODO #4: construir correctamente las aristas de los grafos
    hacerAristas(grafo);
}
```

Fig. 4. TODO #3. Fuente: Elaboración propia.

Se crean en el grafo todos los vértices, en este caso, las coordenadas de los agujeros del circuito, incluyendo al origen (0,0).

5. *TODO #4: construir correctamente las aristas de los grafos.*

```cpp
// TODO #4: construir correctamente las aristas de los grafos
void hacerAristas(Grafo<std::pair<float, float>>& grafo) {
    std::vector< std::pair<float, float>> vertices = grafo.getVertices();
    for (int i = 0; i < vertices.size(); i++) {
        for (int j = 0; j < vertices.size(); j++) {
            if (j != i) {
                float distanciaAgujeros = distancia(vertices[i], vertices[j]);
                grafo.insertarArista(vertices[i], vertices[j], distanciaAgujeros);
            }
        }
    }
}
```

Fig. 5. TODO #4. Fuente: Elaboración propia.

Se crean en el grafo todas las aristas, que corresponden a todos los agujeros conectados con todos los demás agujeros (sin incluirse a sí mismo), siendo el peso de la arista la distancia euclidiana entre los dos agujeros.

6. *TODO #5: encontrar el recorrido más corto del circuito de agujeros.*

```cpp
std::vector<T> Grafo<T>::circuitoVertices(T origen) {
    // ...
    std::vector<T> verticesIda;
    verticesIda.push_back(origen);
    // Se almacenan los vértices de vuelta, finalizando por el origen
    std::deque<T> verticesVuelta;
    verticesVuelta.push_front(origen);

    while (verticesIda.size() + verticesVuelta.size() != this->vertices.size() + 1) {
        // Se utilizan pesoMinimo y verticeMinimo de ida y vuelta para encontrar el vértice más cercano para ida y vuelta
        float pesoMinimoIda = INFINITY, pesoMinimoVuelta = INFINITY;
        T verticeMinimoIda, verticeMinimoVuelta;
        // La posición actual de ida es la posición del último en verticesIda en el vector de vértices del grafo
        int posicionActualIda = buscarVertice(verticesIda[verticesIda.size() - 1]);
        // La posición actual de vuelta es el primero en verticesVuelta en el vector de vértices del grafo
        int posicionActualVuelta = buscarVertice(verticesVuelta[0]);

        // Se envía la información para hallar el destino más cercano de ida
        hallarDestinoMasCercano(posicionActualIda, verticesIda, verticesVuelta, pesoMinimoIda, verticeMinimoIda, origen);
        // Se envía la información para hallar el destino más cercano de vuelta
        hallarDestinoMasCercano(posicionActualVuelta, verticesIda, verticesVuelta, pesoMinimoVuelta, verticeMinimoVuelta, origen);

        // Si el más cercano en la ida y la vuelta es el mismo vuelve a buscar el más cercano del que tenga mayor distancia
        if (verticeMinimoIda == verticeMinimoVuelta) { ... }

        // En el caso de que solo quedé 1 vértice sin visitar solo lo insertamos en verticesIda
        if (verticeMinimoIda == verticeMinimoVuelta) { ... }
        else { ... }
    }

    // Insertar al final del vector el contenido del deque
    verticesIda.insert(verticesIda.end(), verticesVuelta.begin(), verticesVuelta.end());
```

Fig. 6. TODO #5. Fuente: Elaboración propia.

Se toma como parámetro el vértice de inicio y se inician dos conjuntos de vértices: verticesIda y verticesVuelta. El primero representa los vértices visitados en el recorrido de ida, y el segundo los vértices visitados en el recorrido de vuelta. Los vértices de ida inician y los vértices de vuelta finalizan con el vértice de inicio.

En cada iteración, se busca el vértice más cercano al último vértice visitado en verticesIda y verticesVuelta, evitando repetir vértices ya visitados. Se agrega el vértice más cercano a la secuencia correspondiente (verticesIda o verticesVuelta). Se repite el proceso hasta que todos los vértices han sido visitados y la secuencia de perforación está completa.

Finalmente se combinan las secuencias de verticesIda y verticesVuelta para formar la secuencia final del recorrido de perforación. Esta secuencia de vértices, que representa la secuencia óptima de perforación en el circuito, es retornada.

7. *TODO #6: encontrar la distancia total del recorrido más corto del circuito de agujeros.*

```cpp
template <class T>
float Grafo<T>::distanciaRecorrido(std::vector<T> verticesRecorrido) {
    float distanciaPesoTotal = 0.0;
    // Recorrer todos los vértices del recorrido
    for (int i = 0; i < verticesRecorrido.size()-1; i++) {
        int posicionActual = buscarVertice(verticesRecorrido[i]);
        typename std::list< std::pair<T, float >>::iterator it = this->adyacencia[posicionActual].begin();
        // Recorrer todas las aristas del vértice actual
        for (; it != this->adyacencia[posicionActual].end(); it++) {
            // Suma el peso de la arista del vértice actual hacia el vértice siguiente
            if ((*it).first == verticesRecorrido[i+1]) {
                distanciaPesoTotal += it->second;
            }
        }
    }
    return distanciaPesoTotal;
}
```

Fig. 7. TODO #6. Fuente: Elaboración propia.

Se itera sobre la secuencia de vértices recibida, tomando cada par de vértices consecutivos. Para cada par de vértices consecutivos, busca la arista correspondiente en la lista de adyacencia y suma su peso a distanciaPesoTotal.

Repite este proceso hasta recorrer todos los vértices en la secuencia, calculando así la distancia total recorrida. Retorna la distancia total, en este caso la distancia recorrida por el taladro en milímetros.

8. *TODO #7: informar en pantalla la cantidad de circuitos en el archivo, la cantidad de agujeros para cada circuito, y la distancia total recorrida por el taladro al momento de perforar los agujeros de cada circuito.*

```
bufferSalida += std::to_string(agujerosOrdenados.size()) + "\n";
// TODO #7: informar en pantalla la cantidad de agujeros para cada circuito,
//          y la distancia total recorrida por el taladro al momento de perforar los agujeros de cada circuito
std::cout << "\nCantidad de agujeros circuito " << contCircuito << ": " << agujerosOrdenados.size()
    << "\nDistancia total del circuito " << contCircuito << ": " << distanciaTotal << "\n";
for (int i = 0; i < agujerosOrdenados.size(); i++) {
    bufferSalida += std::to_string(agujerosOrdenados[i].first) + " " + std::to_string(agujerosOrdenados[i].second) + "\n";
}
contCircuito++;
```

Fig. 8. TODO #7. Fuente: Elaboración propia.

Se imprime por pantalla la información correspondiente de cada circuito.

9. *TODO #8: almacenar en un buffer la información que contendrá el archivo de salida, que coincida con el formato del archivo de entrada, pero con los agujeros ordenados de tal forma que el taladro de la máquina recorra la menor distancia al perforar los agujeros en dicha secuencia para cada uno de los circuitos impresos.*

```
agujerosOrdenados.erase(agujerosOrdenados.end() - 1);
agujerosOrdenados.erase(agujerosOrdenados.begin());
// TODO #8: almacenar en un buffer la información que contendrá el archivo de salida
bufferSalida += std::to_string(agujerosOrdenados.size()) + "\n";
// TODO #7: informar en pantalla la cantidad de agujeros para cada circuito,
//          y la distancia total recorrida por el taladro al momento de perforar los agujeros de cada circuito
std::cout << "\nCantidad de agujeros circuito " << contCircuito << ": " << agujerosOrdenados.size()
    << "\nDistancia total del circuito " << contCircuito << ": " << distanciaTotal << "\n";
// TODO #8: almacenar en un buffer la información que contendrá el archivo de salida
for (int i = 0; i < agujerosOrdenados.size(); i++) {
    bufferSalida += std::to_string(agujerosOrdenados[i].first) + " " + std::to_string(agujerosOrdenados[i].second) + "\n";
}
```

Fig. 9. TODO #8. Fuente: Elaboración propia.

Se elimina la primera y última posición del vector de agujeros ordenados (que corresponden al punto (0,0)) para poder almacenar la información de los agujeros ordenados de forma correcta en el buffer que posteriormente se escribirá en el archivo de salida.

10. *TODO #9: escribir el archivo de salida con la información almacenada en el buffer.*

```
// TODO #9: escribir el archivo de salida con la información almacenada en el buffer
void escribirArchivo(const std::string& filename, const std::string& bufferSalida) {
    std::ofstream output(filename.c_str());
    if (!output) {
        perror("No se pudo crear o abrir el archivo especificado");
        exit(-1);
    }

    output << bufferSalida;

    //Cierra el archivo
    output.close();
}
```

Fig. 10. TODO #9. Fuente: Elaboración propia.

Se escribe el archivo de salida con el mismo formato que el archivo de entrada, toda esta información ya fue cargada con anterioridad al buffer de salida.

V. PRUEBAS DE FUNCIONAMIENTO

1. *Archivo in_0.txt*

Para el archivo in_0.txt se comprueba el funcionamiento del programa tanto para la impresión de la información por pantalla solicitada, como el archivo de solución solucion_0.txt generado con los puntos de cada circuito ordenados en el recorrido de menor distancia.

```
Cantidad de circuitos en el archivo: 3

Cantidad de agujeros circuito 1: 6
Distancia total del circuito 1: 34.0783

Cantidad de agujeros circuito 2: 3
Distancia total del circuito 2: 25.6328

Cantidad de agujeros circuito 3: 8
Distancia total del circuito 3: 36.4506
```

Fig. 11. Información en pantalla correspondiente a la ejecución con in_0.txt como archivo de entrada. Fuente: Elaboración propia.

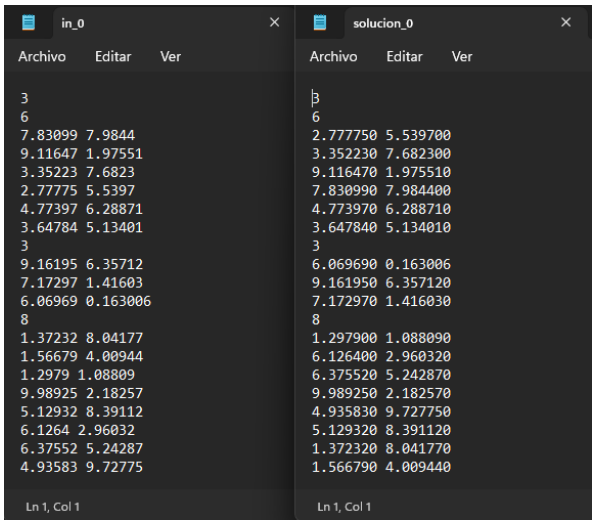

Fig. 12. Comparación del archivo de entrada in_0.txt con el archivo de salida solución_0.txt. Fuente: Elaboración propia.

2. *Archivo in_1.txt*

Para el archivo in_1.txt se comprueba el funcionamiento del programa tanto para la impresión de la información por pantalla solicitada, como el archivo de solución solucion_1.txt generado con los puntos de cada circuito ordenados en el recorrido de menor distancia.

```
Cantidad de circuitos en el archivo: 11

Cantidad de agujeros circuito 1: 2
Distancia total del circuito 1: 20.8559

Cantidad de agujeros circuito 2: 4
Distancia total del circuito 2: 30.5106

Cantidad de agujeros circuito 3: 0
Distancia total del circuito 3: 0

Cantidad de agujeros circuito 4: 16
Distancia total del circuito 4: 38.5682

Cantidad de agujeros circuito 5: 11
Distancia total del circuito 5: 45.1465

Cantidad de agujeros circuito 6: 12
Distancia total del circuito 6: 42.0964

Cantidad de agujeros circuito 7: 5
Distancia total del circuito 7: 20.1599

Cantidad de agujeros circuito 8: 10
Distancia total del circuito 8: 36.4154

Cantidad de agujeros circuito 9: 0
Distancia total del circuito 9: 0

Cantidad de agujeros circuito 10: 14
Distancia total del circuito 10: 47.5912

Cantidad de agujeros circuito 11: 11
Distancia total del circuito 11: 44.1946
```

Fig. 13. Información en pantalla correspondiente a la ejecución con in_1.txt como archivo de entrada. Fuente: Elaboración propia.

in_1

Archivo Editar Ver

```
11
2
5.26745 7.69914
4.00229 8.91529
4
3.52458 8.07725
9.19026 0.697553
9.49327 5.25995
0.860558 1.92214
0
16
3.48893 0.641713
0.20023 4.57702
0.630958 2.3828
9.70634 9.02208
8.5092 2.66666
5.3976 3.75207
7.60249 5.12535
6.67724 5.31606
0.392803 4.37638
9.31835 9.3081
7.20952 2.84293
7.38534 6.39979
3.54049 6.87861
1.65974 4.40105
8.80075 8.29201
3.30337 2.28968
11
3.5036 6.8667
9.56468 5.8864
6.57304 8.58676
4.3956 9.2397
3.98437 8.14767
```

solucion_1

Archivo Editar Ver

```
11
2
5.267450 7.699140
4.002290 8.915290
4
0.860558 1.922140
9.190260 0.697553
9.493270 5.259950
3.524580 8.077250
0
16
0.630958 2.382800
0.392803 4.376380
0.200230 4.577020
1.659740 4.401050
3.540490 6.878610
7.209520 2.842930
8.509200 2.666660
9.706340 9.022080
9.318350 9.308100
8.800750 8.292010
7.385340 6.399790
7.602490 5.125350
6.677240 5.316060
5.397600 3.752070
3.303370 2.289680
3.488930 0.641713
11
4.824910 2.158250
3.984370 8.147670
4.395600 9.239700
6.573040 8.586760
6.842190 9.109720
```

Fig. 14. Comparación del archivo de entrada in_1.txt con el archivo de salida solución_1.txt. Fuente: Elaboración propia.

3. *Archivo in_2.txt*

   Para el archivo in_2.txt se comprueba el funcionamiento del programa tanto para la impresión de la información por pantalla solicitada, como el archivo de solución solucion_2.txt generado con los puntos de cada circuito ordenados en el recorrido de menor distancia.

```
Cantidad de circuitos en el archivo: 10

Cantidad de agujeros circuito 1: 19
Distancia total del circuito 1: 43.4641

Cantidad de agujeros circuito 2: 2
Distancia total del circuito 2: 19.8183

Cantidad de agujeros circuito 3: 18
Distancia total del circuito 3: 44.3247

Cantidad de agujeros circuito 4: 16
Distancia total del circuito 4: 37.2505

Cantidad de agujeros circuito 5: 8
Distancia total del circuito 5: 29.3432

Cantidad de agujeros circuito 6: 5
Distancia total del circuito 6: 26.2958

Cantidad de agujeros circuito 7: 10
Distancia total del circuito 7: 32.7932

Cantidad de agujeros circuito 8: 1
Distancia total del circuito 8: 19.7293

Cantidad de agujeros circuito 9: 13
Distancia total del circuito 9: 35.9146

Cantidad de agujeros circuito 10: 10
Distancia total del circuito 10: 38.3791
```

Fig. 15. Información en pantalla correspondiente a la ejecución con in_2.txt como archivo de entrada. Fuente: Elaboración propia.

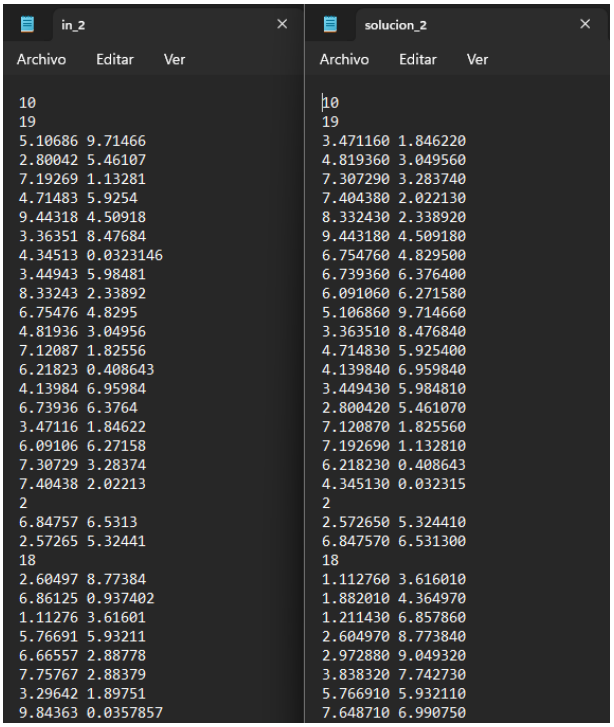

Fig. 16. Comparación del archivo de entrada in_2.txt con el archivo de salida solución_2.txt. Fuente: Elaboración propia.

VI. Conclusión

El desarrollo del taller ha permitido aplicar conceptos de grafos para abordar eficientemente el problema de optimización relacionado con la perforación de circuitos impresos. La implementación de un grafo y la utilización de sus algoritmos han posibilitado la generación de secuencias de perforación que minimizan la distancia total recorrida por el taladro. Este enfoque no solo optimiza el tiempo de vida de la máquina perforadora, sino que también proporciona una solución práctica y efectiva para la disposición ordenada de agujeros en circuitos impresos de diferentes tamaños.