

Algorítmica

Curso 2023-2024

Grupo Viterbi



PRÁCTICA 3-ALGORITMOS GREEDY

Integrantes:

Miguel Ángel De la Vega Rodríguez	miguevrod@correo.ugr.es
Alberto De la Vera Sánchez	joaquinrojo724@correo.ugr.es
Joaquín Avilés De la Fuente	adelaveras01@correo.ugr.es
Manuel Gomez Rubio	e.manuelgmez@go.ugr.es
Pablo Linari Perez	e.pablolinari@go.ugr.es

Facultad de Ciencias UGR
Escuela Técnica Ingeniería Informática UGR
Granada
2023-2024

Índice general

Autores

- **Miguel Ángel De la Vega Rodríguez: 20%**
 - Algoritmos Greedy del Viajante
 - Redacción memoria sección Viajante
- **Joaquín Avilés De la Fuente: 20%**
 - Programacion Dijkstra
 - Programación creación de grafos (casos de prueba)
 - Redacción memoria sección Dijkstra
- **Alberto De la Vera Sánchez: 20%**
 - Redacción L^AT_EX
 - Estudio de umbrales teóricos y empíricos de DyV
 - Graficas y ajustes
- **Manuel Gomez Rubio 20%**
 - Redacción y demostración del algorimo greedy
 - Redacción memoria del algoritmo Dijkstra
 - Programacion Dijkstra
- **Pablo Linari Pérez: 20%**
 - Programacion Aulas
 - Redacción Aulas
 - Redacción objetivos

Apartado 2

Equipo de trabajo

- **Miguel Ángel De la Vega Rodríguez:** (Ordenador donde se ha realizado el compute)
 - AMD Ryzen 7 2700X 8-Core
 - 16 GB RAM DDR4 3200 MHz
 - NVIDIA GeForce GTX 1660 Ti
 - 1 TB SSD NVMe
 - Debian 12 Bookworm
 - Compilador GCC 12.2.0

Apartado 3

Objetivos

El objetivo de esta práctica es implementar la solución para cuatro problemas usando algoritmos Greedy. Los problemas a resolver son los siguientes:

- **Hijo predilecto** Un individuo desea repartir sus N bienes entre sus dos hijos. Para cada bien se conoce el valor (positivo) del mismo. Previo al fallecimiento del individuo, el juzgado, de una forma ciega, determina un valor k , que nos indica el número de bienes que se le deben asignar a un hijo al otro se le asignan $N - k$. Conocido estos datos, el individuo desea distribuir los bienes entre sus hijos de forma que uno de ellos salga lo mas beneficiado posible. El beneficio obtenido por cada hijo se define como la suma de los beneficios de los bienes que se les lega.
- **Asignación de aulas** Se deben programar N exámenes en un día determinado, y para cada examen, el profesorado ha especificado el tiempo requerido para su realización y la hora de inicio ideal. El centro cuenta con m aulas disponibles, donde m es mayor que N , lo que garantiza la viabilidad de programar los exámenes. Sin embargo, cada vez que se utiliza un aula, el centro debe asegurar su vigilancia, lo que implica contratar a un vigilante distinto para cada aula utilizada.
- **Problema del camino mínimo** Tenemos un conjunto de n ciudades (puntos en un plano), cada una definida por las coordenadas en el mapa (x_i, y_i) , con $i = 1, \dots, n$. Cada ciudad se puede conectar por carretera con un subconjunto de ciudades, asegurando que la red de vías que se genere crea un grafo conexo. Dada una ciudad de origen, x , y una ciudad de destino, y , se pide encontrar el camino de costo mínimo que permita viajar de x a y .
- **problema del viajante de comercio** Tenemos un conjunto de n ciudades (puntos en un plano), cada una definida por las coordenadas en el mapa (x_i, y_i) , con $i = 1, \dots, n$. La distancia entre dos ciudades viene dada por la distancia euclídea entre sus coordenadas. El problema de viajante de comercio consiste en encontrar el orden en el que un viajante, partiendo de la ciudad de origen (por ejemplo (x_1, y_1)) pase por todas y cada una de las ciudades una única vez, para volver a la ciudad de partida, formando un ciclo.

Para cada uno de los problemas se ha de implementar un algoritmo que de una solución óptima o aproximada dependiendo del problema. En todos los problemas se identificarán los componentes de un algoritmo Greedy y se demostrará que el algoritmo greedy da la solución óptima en el caso de que el problema sea de optimización.

Con esta práctica se pretende trabajar y comprender los algoritmos greedy y reconocer las situaciones para las cuales puede ser beneficioso y usar esta técnica.

Apartado 4

Asignación de aulas

En esta sección se estudia lo referente al algoritmo usado para resolver el segundo problema el cual nos pide lo siguiente :

Se quieren realizar N exámenes en un día en la ETSIIT , el centro cuenta con m aulas donde m es mayor que N . Sin embargo cada vez que se utiliza un aula el centro debe contratar un vigilante por tanto se debe diseñar un algoritmo que calcule en base al horario de los exámenes el menor costo posible para la escuela , es decir hay que encontrar la manera de distribuir los exámenes para usar el menor número de aulas posibles .

4.1 Estructura de los datos

Una vez planteado el problema procedemos a describir las herramientas que hemos usado para resolverlo Para representar los horarios de los exámenes hemos usado la siguiente estructura :

```
1 struct tiempo{
2     int hora;
3     int min;
4     tiempo(int h , int m){
5         hora = h;
6         min = m ;
7     }
8 };
9 struct examen{
10     tiempo inicio; // Hora de inicio
11     tiempo final; // Hora de finalizacion
12     examen(tiempo t , tiempo t1) : inicio(t), final(t1){}
13 };
14
15 struct compare{
16     bool operator()(const examen&a,const examen&b){
17         if(a.inicio.hora==b.inicio.hora){
18             return a.inicio.min < b.inicio.min;
19         }
20         return a.inicio.hora < b.inicio.hora;
21     }
22 };
```

El struct compare se utiliza para ordenar el vector de entrada de exámenes de manera que se ordenen de menor a mayor en función de la hora de inicio de los exámenes , de esta manera se facilita la ejecución del algoritmo que se encarga de asignar los exámenes a las aulas de la manera más eficiente posible.

4.2 Algoritmo

A continuación vemos el algoritmo usado para la distribución de las aulas.

```

1      bool sesolapan(examen e1,examen e2){
2          if(e1.final.hora == e2.inicio.hora){
3              return e1.final.min > e2.inicio.min;
4          }
5          return e1.final.hora > e2.inicio.hora;
6      }
7
8      int minAulas(const vector<examen> & v){
9
10         // Conjunto de candidatos v
11
12         int naulas = 1;
13         vector<queue<examen>> horario;
14         //Conjunto de seleccionados horario (vector de colas , cada
           cola es un aula)
15
16         horario.emplace_back(queue<examen>());
17         horario[0].push(v[0]);
18         int i;
19         //
20         for(int nex = 1;nex < v.size();nex++){
21             i=0;
22             //Funcion de seleccion
23             while(i<horario.size() && (sesolapan(horario[i].back(),
           v[nex]))){i++;}
24             //Funcion de factibilidad
25             if(i==horario.size()){
26                 horario.emplace_back(queue<examen>());
27                 horario[i].push(v[nex]);
28                 naulas++;
29             }
30             else{
31                 horario[i].push(v[nex]);
32             }
33         }
34         return naulas; //Funcion objetivo
35     }

```

Uno de los factores importantes que hay que destacar de este algoritmo es que el vector **v** se supone ordenado de mayor a menor , es decir que en el vector los exámenes aparecen en orden creciente de horario de inicio de esta manera la asignación de aulas es más rápida y eficiente.

El algoritmo comienza añadiendo el primer examen al primer aula , cada aula se representa con una posición del vector y el horario del aula se representa con una cola ya que solo nos interesa el último examen hecho en el aula.

Posteriormente se entra al bucle for el cual recorre todo el vector de candidatos ya que todos los exámenes deben ser realizados .

Dentro del bucle for se entra a un while el cual comprueba mediante la función **sesolapan(examen e1 ,examen e2)** si el examen que está buscando un aula se solapa con el que se está realizando en el aula , si no se solapan se añade a ese aula , si se solapan se busca otra aula en la que no se solapen y se añade a esa aula. En el caso de que se hallan agotado las aulas actuales se crea una nueva aula y se añade el examen a esa aula.

4.3 Justificación Greedy

En este apartado vamos a analizar el algoritmo identificando los componentes que lo hacen un algoritmo greedy y demostrando que siempre llega a la solución óptima.

4.4 Componentes

- **Candidatos a seleccionar:**
En este problema los candidatos a seleccionar son los exámenes los cuales se tendrán que distribuir entre aulas.
- **Candidatos seleccionados:** Los candidatos seleccionados en este caso son el conjunto aula-exámenes del cual nos interesa para la solución el número de aulas en el que se distribuyen los exámenes.
- **Función Solución:** La función solución en este caso es el for de la línea 20 ya que la solución se alcanza cuando el vector de exámenes se ha conseguido recorrer entero, lo que significará que todos los exámenes han sido asignados a un aula. Esto siempre se conseguirá dada la condición del problema.
- **Función de Selección:**
La función de selección en este algoritmo se encuentra en las líneas 21 a 24 donde el bucle while recorre las posibles aulas para seleccionar la primera en la que haya un hueco para el posible candidato.
- **Función de factibilidad:**

En este caso no hay función de factibilidad ya que siempre se puede seleccionar un examen por la condición inicial del problema.

- **Función objetivo:**
La función objetivo es la línea 34 en la cual se devuelve la cantidad de aulas usadas.

4.4.1 optimalidad

Una vez definidas las funciones, conjuntos de candidatos, etc, pasaremos a realizar la demostración del algoritmo greedy. Para la demostración se usará el método de inducción en el cual definiremos dos casos base uno para 1 examen y otro para 2 exámenes para que sea más fácil la demostración.

- **Caso base $n=1$:** En el caso base se tiene un solo examen y por tanto solo se necesita un aula para realizarlo. Por lo que se asigna el aula y el examen sin problema.
- **Caso base $n=2$:** En el caso base $n=2$ se tienen dos exámenes y caben 2 posibilidades, que los exámenes se solapen por tanto se necesitarán 2 aulas o que no se solapen y por tanto se necesitará solo un aula.
- **Hipótesis de inducción:** Supongamos que el algoritmo es óptimo para n exámenes y veamos que también lo es para $n+1$ exámenes.
- **Paso de inducción:** Supongamos que para n exámenes tenemos una solución óptima de k aulas. Veamos que ocurre para $n+1$ donde caben dos opciones:
 - Que el examen $n+1$ se solape con alguno de los n exámenes anteriores que se están realizando en una de las k aulas por tanto se necesitará una nueva aula para el examen $n+1$ por lo que el número de aulas aumentará a $k+1$ y como la solución para n exámenes era óptima la solución para $n+1$ exámenes también lo será ya que el único cambio es el de añadir un aula.

- Que el examen $n+1$ no se solape con ninguno de los n exámenes anteriores, en este caso se podrá añadir a un aula ya existente en la que no se esté haciendo ningún examen por lo que el número de aulas necesarias será k y como para n exámenes k era la solución óptima y este último examen no ha modificado el valor de k la solución para $n+1$ exámenes también será óptima.
- Conclusión: Dado que se ha demostrado que existe solución óptima para $n+1$ exámenes concluimos la demostración por inducción.

4.5 Ejecución del Algoritmo

Para ejecutar el programa se usan los siguientes parámetros: ./Aulas <número de exámenes> <duración máxima del examen>

0:37 - 2:16 0:51 - 1:56 2:44 - 5:11 8:32 - 10:2 10:47 - 11:17 13:12 - 16:12 13:45 - 13:56 15:51 - 17:10 17:51 - 20:46 18:30 - 18:30 19:24 - 19:43 19:51 - 22:34 21:51 - 23:57 22:18 - 23:36

0 0:37 - 2:16 2:44 - 5:11 8:32 - 10:2 10:47 - 11:17 13:12 - 16:12 17:51 - 20:46 21:51 - 23:57

1 0:51 - 1:56 13:45 - 13:56 15:51 - 17:10 18:30 - 18:30 19:24 - 19:43 19:51 - 22:34

2 22:18 - 23:36

numero de aulas : 3

4.6 Graficas

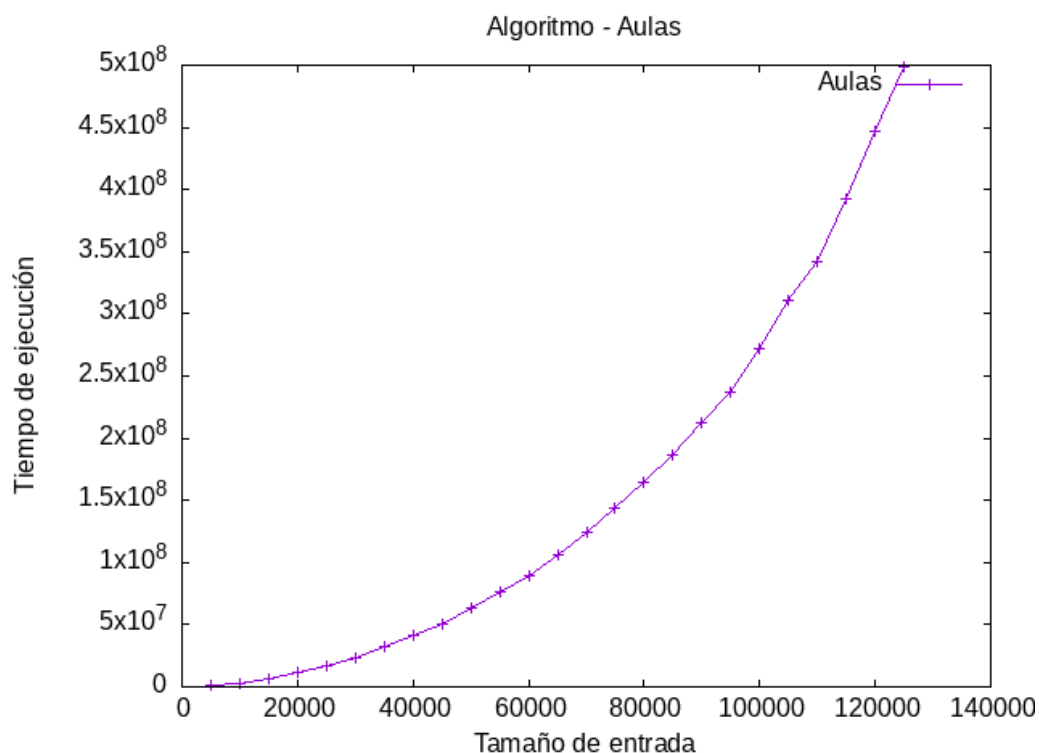


Figura 4.1: Ejecución de 5000 a 125000 exámenes de maximo 3 horas

Camino mínimo

En esta sección se analiza todo lo relacionado con el tercer problema propuesto, el problema del camino mínimo entre dos ciudades. Comentaremos el algoritmo usado, es decir, el **algoritmo de Dijkstra**, y su implementación en C++, así como la justificación de su uso mediante la demostración de que dicho algoritmo Greedy calcula la solución óptima.

Destacar también que mostraremos la gestión que hemos hecho para la creación de los posibles casos (**grafos**), así como la ejecución de los mismos y la obtención de los resultados.

5.1 Algoritmo de Dijkstra

A continuación, veremos las distintas partes usadas para la demostración como bien pueden ser la función de selección o la forma en que se seleccionan los nodos, y finalmente la demostración mediante inducción.

Definiremos pues cada parte del algoritmo:

- **Candidatos a seleccionar:**
En este problema los candidatos a seleccionar son los diferentes vértices del grafo que no hayan sido seleccionados previamente.
- **Candidatos seleccionados:** Llamaremos S al conjunto de los vértices seleccionados.
En nuestro problema siempre partiremos de un vértice, por lo que, como mínimo siempre habrá un vértice en S que será el origen desde donde partiremos a considerar los nuevos vértices.
Una vez definido S hay que definir también $V \setminus S$ que será el conjunto de vértices candidatos a ser seleccionados, siendo por tanto, V el total de vértices del grafo.
- **Función Solución:**
La solución se obtendrá cuando el nodo origen y el destino estén en S , siendo el nodo destino el último seleccionado, teniendo así el camino mínimo $M(o, v)$
- **Función de Selección:**
La función de selección en este algoritmo es siempre tomar el camino de menor distancia posible entre los nodos que estén conectados con el origen, actualizando los nuevos posibles candidatos tras una elección
- **Función de factibilidad:**
Un vértice es seleccionable en el caso en el que su distancia respecto al origen no sea infinito, esto quiere decir que debe haber vértices dentro de los seleccionados que nos permitan, mediante caminos directos, alcanzar el vértice sobre el que nos preguntamos si es factible

- Funcion objetivo:

El problema tendrá siempre tamaño \mathcal{V} , es decir, el número de vértice del grafo. (Se puede considerar el tamaño real como $\mathcal{V} - 1$ ya que el nodo de origen siempre es el primero en seleccionarse sin tener en cuenta ningún criterio)

Una vez definidas las funciones, conjuntos de candidatos, etc, pasaremos a realizar la demostración del algoritmo greedy.

5.1.1 Demostracion de la optimalidad del algoritmo de Dijkstra

Para la demostración, usaremos inducción y reducción al absurdo, veámoslo paso a paso:

- El caso base será cuando $\#S = 1$ o bien es igual a 2.
En ambos casos el algoritmo nos dará una solución trivial, ya que, en el caso $\#S = 1$ la distancia mínima entre un vértice consigo mismo es 0. En el caso de que $\#S = 2$, tenemos que la distancia mínima entre los dos nodos será la que los conecte con menor distancia, puesto que el grafo es conexo y no pueden no estar conectados.
- Paso de inducción: Sea v el nuevo vértice que ha sido seleccionado por el algoritmo, y por tanto $v \in S$. Ahora usaremos reducción al absurdo y tomaremos que la distancia $d(v)$ (que representa algún camino de o a v , con o como el nodo de origen) no es la longitud mínima.
Tomaremos entonces, \mathcal{L} como la distancia mínima de o a v . Y tenemos en cuenta que en \mathcal{L} hay un camino que parte de un nodo x , $(c(x, y)$ camino de x a y), siendo x el último nodo en S
Con todo esto tenemos:

$$d(v) > \mathcal{M}(o, v) = \mathcal{M}(o, x) + c(x, y) + \mathcal{M}(y, v) > \mathcal{M}(o, x) + c(x, y) = d(x) + c(x, y) \geq d(y)$$

Donde hemos usado que \mathcal{L} es el camino mínimo y está formado por la primera igualdad.

En la primera desigualdad, usamos la hipótesis, puesto que suponíamos que $d(v) > \mathcal{L}$, mientras que en la última igualdad se usa la hipótesis de inducción, pues $d(x) = \mathcal{M}(o, x)$ es la distancia mínima de o a x , ya que es un punto escogido con anterioridad, y por tanto, $x \in S$

Por último, nos falta explicar la contradicción que hemos obtenido, que está marcada en negrita. Esta contradicción nos indica que al ser la distancia de y menor que la de v , nuestro algoritmo debería haber elegido al vértice y en vez de a v

Luego, hemos probado la optimalidad de nuestro algoritmo.

5.2 Estructura de datos y grafos

Para la implementación del algoritmo de Dijkstra, hemos usado una estructura de datos que nos permita representar un grafo, en este caso, hemos usado una matriz de adyacencia. Por tanto, a continuación se explicara de forma detallada la creación de dichos grafos así como la forma de representarlos en archivos de texto, ya que al ser la primera vez que se trabaja con grafos es útil tener claro como se ha hecho.

Para la creación de los grafos se ha hecho uso de un programa en C++ que nos permite generar grafos aleatorios, a partir de un vector que tenga los distintos puntos. Destacar en primer lugar, que para la creación de los puntos de forma aleatoria hemos usado de este trozo de código:

```

1 // Inicializar el generador de numeros aleatorios
2 srand(time(NULL));
3
4 // Generar puntos aleatorios y escribirlos en el archivo
5 for (int i = 0; i < numPuntos; i++) {

```

```

6         int coordX = rand() % (rangoMax + 1); // Generar coordenada x
7         int coordY = rand() % (rangoMax + 1); // Generar coordenada y
8         file << coordX << " " << coordY << endl; // Escribir coordenadas
           al archivo
9     }

```

donde file es el archivo donde se escriben los puntos, numPuntos es el número de puntos que se quieren generar y rangoMax es el rango máximo de las coordenadas.

Tenemos ahora archivos con distintos puntos aleatorios, y para la creación de los grafos, hemos usado el siguiente código:

```

1 // Funcion para la creacion de grafos a partir de un vector de puntos
  en 2D
2 void creacionGrafos(const vector<Point>& vec, vector<vector<double>>&
  matriz) {
3     int size = vec.size();
4
5     // Inicializacion de srand con resolucion mas alta
6     unsigned seed = chrono::high_resolution_clock::now().
       time_since_epoch().count();
7     srand(seed);
8
9     for (int i = 0; i < size; i++) {
10         // Num nodos que conectaremos con el nodo i
11         int conexiones = rand() % (size/3 - 1) + 1; // Asegura al
           menos una conexion por nodo
12         for (int j = 0; j < conexiones; j++) {
13             int nodo_conectar;
14             do {
15                 nodo_conectar = rand() % size;
16             } while (nodo_conectar == i); // Asegura no
           conectarse a si mismo
17
18             // Aniadimos su distancia a la matriz de forma
              simetrica
19             double distancia = vec[i].distanceTo(vec[
              nodo_conectar]);
20             matriz[i][nodo_conectar] = distancia;
21             matriz[nodo_conectar][i] = distancia; // Simetria
22         }
23     }
24
25     // Garantizar que al menos un nodo (p_inicio) este conectado a
      todos los demas
26     int p_inicio = rand() % size;
27     for (int i = 0; i < size; i++) {
28         if (i != p_inicio) {
29             double distancia = vec[p_inicio].distanceTo(vec[i]);
30             matriz[p_inicio][i] = distancia;
31             matriz[i][p_inicio] = distancia; // Simetria
32         }
33         matriz[i][i] = 0; // Distancia a si mismo es cero
34     }
35 }

```

donde la matriz pasada por referencia será la llamada **matriz de adyacencia** que representa el grafo, y **vec** es el

vector de puntos leído del archivo. Importante recalcar las líneas donde se implementan las distancias de forma simétrica en la matriz.

Las matrices de adyacencia que hemos obtenido, las hemos guardado en archivos de texto, para poder leerlas posteriormente y ejecutar el algoritmo de Dijkstra sobre ellas, por lo que para ello hemos creado funciones que permitan leer y guardar dichas matrices, pues hay que recordar que los puntos que no tengan conexión directa, tendrán un valor de infinito en la matriz. Tenemos así estas dos funciones, una para escribir la matriz en un archivo de texto y otra para leerla:

```

1 // Funcion para mostrar una matriz en un archivo de salida
2 void mostrarMatriz(std::ofstream &salida, const std::vector<std::vector
3 <double>>& matriz) {
4     int anchoMaximo = 0;
5     const std::string infStr = "INF";
6
7     // Primero, vamos a calcular el ancho maximo para los numeros y "
8     INF"
9     for (const auto& fila : matriz) {
10         for (double elemento : fila) {
11             std::ostringstream ss;
12             if (elemento != std::numeric_limits<double>::max()) {
13                 ss << std::fixed << std::setprecision(3) << elemento;
14                 // Convertir el numero a string con 3 decimales
15             } else {
16                 ss << infStr; // Si es infinito, usamos "INF"
17             }
18             anchoMaximo = std::max(anchoMaximo, static_cast<int>(ss.str
19             ().length())); // Actualizar el ancho maximo
20         }
21     }
22
23     // Configuracion de la salida con el ancho encontrado
24     salida << std::fixed << std::setprecision(3);
25
26     // Mostrar la matriz con el ancho maximo
27     for (const auto& fila : matriz) {
28         for (double elemento : fila) {
29             if (elemento == std::numeric_limits<double>::max()) {
30                 salida << std::right << std::setw(anchoMaximo) <<
31                 infStr << "\t";
32             } else {
33                 salida << std::right << std::setw(anchoMaximo) <<
34                 elemento << "\t";
35             }
36         }
37         salida << std::endl;
38     }
39 }
40
41 // Funcion para leer la matriz desde un archivo
42 void lecturaMatriz(istream &leer, vector<vector<double>>& matrix) {
43     string line;
44     while (getline(leer, line)) {
45         vector<double> row;

```

```

41         stringstream ss(line);
42         string val_str;
43         while (ss >> val_str) {
44             if (val_str == "INF") {
45                 row.push_back(numeric_limits<double>::max());
46             } else {
47                 row.push_back(stod(val_str));
48             }
49         }
50         matrix.push_back(row);
51     }
52 }
```

donde `numeric_limits<double>::max()` es el valor que hemos usado para representar la ausencia de conexión entre dos nodos, que en la matriz escrita y leída se mostrará como **INF** de infinito.

5.3 Estudio eficiencia teórica

En esta sección estudiaremos la eficiencia teórica de dicho algoritmo, para ello veamos primero el código usado:

```

1 // Funcion auxiliar para encontrar el vertice con la distancia minima
2 int minDistance(const vector<double>& dist, const vector<bool>& visited
3 ) {
4     double min = INF;
5     int min_index = -1; // Inicializado a -1 para detectar errores si
6                         // no se encuentra ninguno
7
8     for (int v = 0; v < dist.size(); v++) {
9         if (!visited[v] && dist[v] <= min) {
10             min = dist[v];
11             min_index = v;
12         }
13     }
14     return min_index;
15 }
16 // Funcion para implementar el algoritmo de Dijkstra desde inicio hasta
17 // un nodo destino w
18 void dijkstra(const vector<vector<double>>& graph, int inicio, int
19 final) {
20     int V = graph.size();
21     vector<double> dist(V, INF); // Distancias minimas
22     vector<bool> visited(V, false); // Nodos visitados
23     vector<int> parent(V, -1); // Para rastrear el camino
24
25     // Inicializar la distancia del nodo inicial a si mismo como 0
26     dist[inicio] = 0;
27
28     for (int count = 0; count < V - 1; count++) {
29         int u = minDistance(dist, visited);
30         visited[u] = true;
31
32         // Si el nodo seleccionado es el nodo destino, terminar el
33         // algoritmo
34     }
```

```

29         if (u == final)
30             break;
31
32         for (int v = 0; v < V; v++) {
33             if (!visited[v] && graph[u][v] != 0 && dist[u] != INF
34                 && dist[u] + graph[u][v] < dist[v]) {
35                 dist[v] = dist[u] + graph[u][v];
36                 parent[v] = u; // Registrar el nodo padre para
37                               rastrear el camino
38             }
39         }
40
41         // Imprimir la distancia minima
42         cout << "Distancia minima desde el nodo " << inicio << " al nodo
43             " << final << ": " << dist[final] << endl;
44
45         // Imprimir el camino recorrido
46         printPath(parent, inicio, final);
47     }

```

donde hemos definido **const double INF = numeric_limits<double>::max()** para representar la ausencia de conexión entre dos nodos. La función **minDistance** nos permite encontrar el nodo con la distancia mínima, y la función **dijkstra** nos permite ejecutar el algoritmo de Dijkstra desde un nodo inicial hasta un nodo final, mostrando la distancia mínima y el camino recorrido, haciendo uso de la función **printPath** que se muestra a continuación:

```

1 // Funcion para imprimir el camino recorrido
2 void printPath(const vector<int>& parent, int src, int dst) {
3     stack<int> path;
4     int current = dst;
5     while (current != src) {
6         path.push(current);
7         current = parent[current];
8     }
9     path.push(src);
10
11     cout << "Camino desde el nodo " << src << " al nodo " << dst << "
12         : ";
13     while (!path.empty()) {
14         cout << path.top() << " ";
15         path.pop();
16     }
17     cout << endl;

```

Visto el código, pasamos a analizar la eficiencia teórica del algoritmo de Dijkstra.

Para ello, primero veamos que la función **minDistance** tiene una eficiencia de $O(V)$, donde V es el número de vértices del grafo, es decir, tomando $V = n$ como el tamaño del problema, tenemos por tanto que la eficiencia de dicha función es $O(n)$.

Por otro lado, en la función **dijkstra** desde la línea 16 hasta la línea 19 se inicializan varios vectores con valores predeterminados y con tamaño $V = n$ (número de filas, es decir, la cantidad de puntos), por lo que la eficiencia de esta parte es $O(n)$. Es importante recalcar que aunque se tome $V = graph.size()$, en realidad, V es el número

de vértices del grafo, ya que para cada fila de la matriz de adyacencia tenemos las distintas distancias con el resto de nodos.

A continuación, encontramos un bucle que se ejecuta $V - 1$ veces, y en cada iteración se llama a la función **minDistance**, por lo que la eficiencia de este bucle es $O(n^2)$. De hecho en cada iteración del bucle de la línea 24 se hace otro bucle que se ejecuta V veces (línea 32), para actualizar las distancias a los nuevos nodos, por lo que tomando la función suma, tenemos que la eficiencia de este bucle es $O(n^2)$.

Por último, la llamada a la función **printPath** tiene una eficiencia de $O(n)$, ya que se recorre (imprimiendo) el camino recorrido, que como mucho tendrá n nodos.

Por tanto, la eficiencia teórica del algoritmo Dijkstra implementado es $O(n^2)$, donde n es el número de nodos del grafo.

5.4 Representaciones gráficas

Finalmente, mostraremos varios ejemplos de la ejecución del algoritmo de Dijkstra sobre distintos grafos, y la representación gráfica de los mismos donde se pueden observar los posibles caminos y el mínimo escogido por dicho algoritmo.

Veamos a continuación un ejemplo en el que se han graficado 7 puntos con distintas conexiones y se ha elegido un camino:

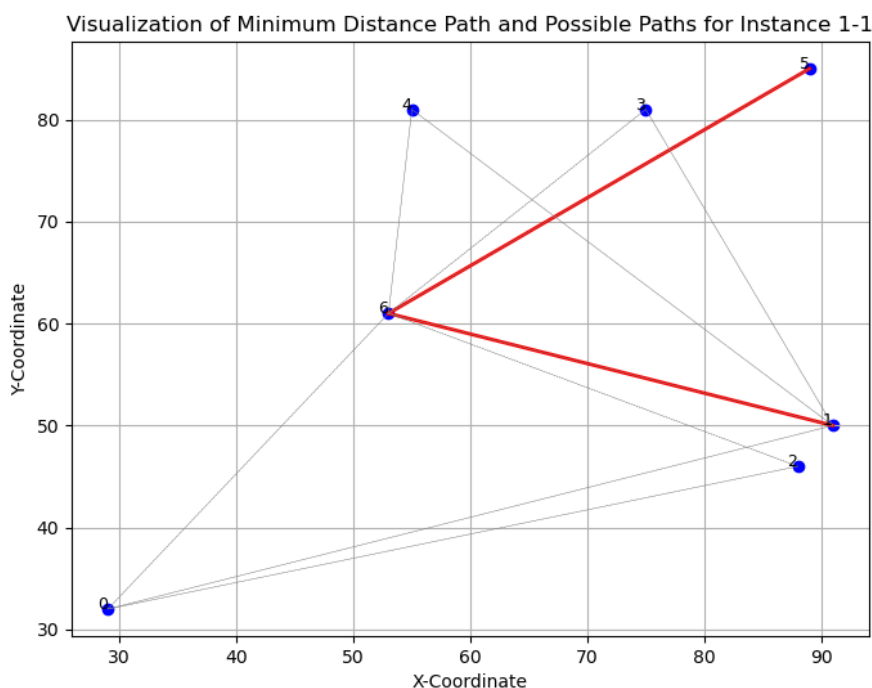


Figura 5.1: Grafo con 7 puntos y conexiones

Mostremos a continuación otro gráfico con los mismos puntos y distintas conexiones y veamos como cambia el camino mínimo escogido:

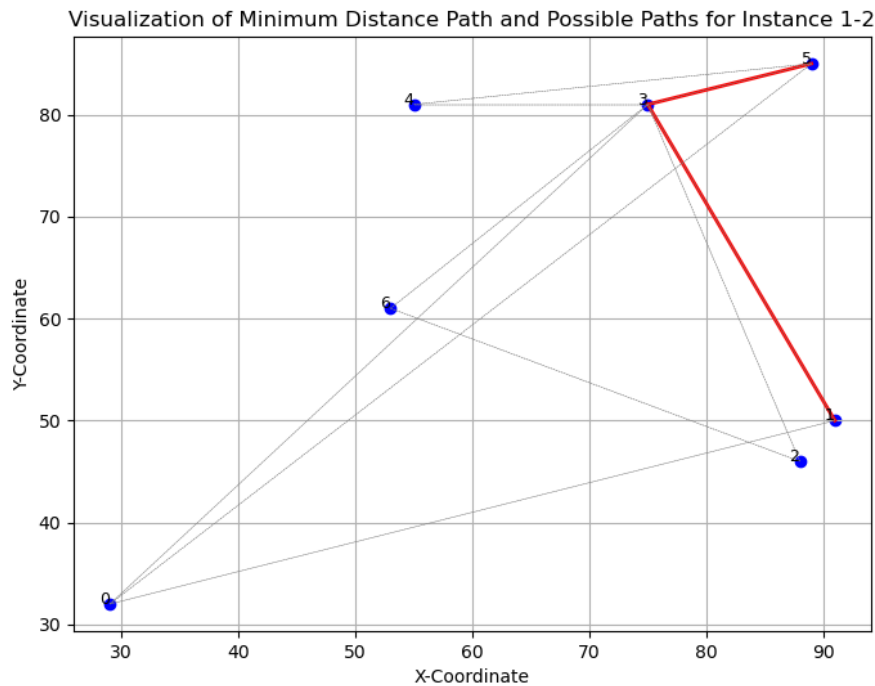


Figura 5.2: Grafo con 7 puntos y conexiones

Destacar que en ambos casos se busca el camino mínimo entre el nodo 1 y 5.

También podemos observar casos donde si existe un camino directo entre los nodos inicial y final, el algoritmo escogerá dicho camino mínimo como es de esperar, veámoslo

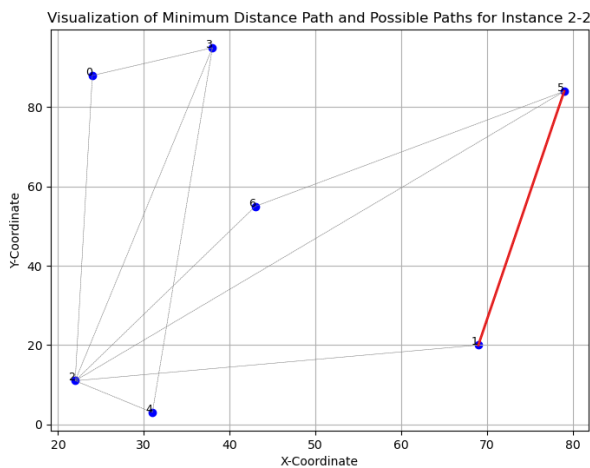


Figura 5.3: Grafo con 7 puntos y conexiones

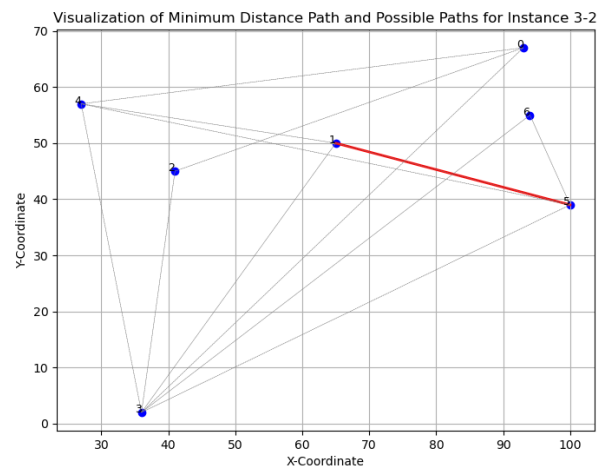


Figura 5.4: Grafo con 7 puntos y conexiones

En ambos casos se busca el camino mínimo entre el nodo 1 y 5.

Por ultimo pondremos un ejemplo con 25 puntos en el que se busca el camino mínimo entre el nodo 4 y 11:

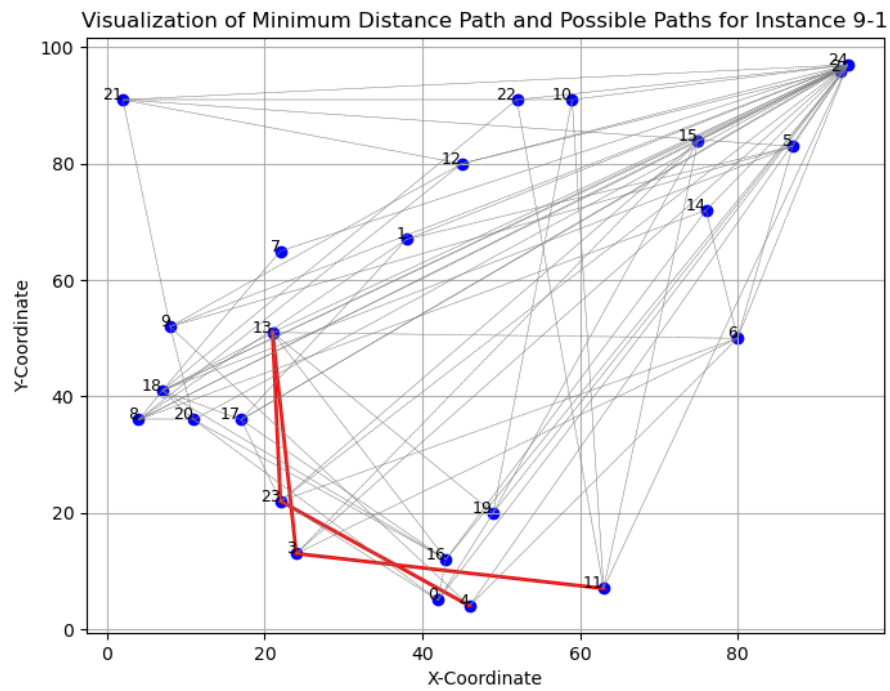


Figura 5.5: Grafo con 25 puntos y conexiones

5.5 Estudio eficiencia empírica

Apartado 6

Viajante

En esta sección se analiza todo aquello referente al cuarto problema propuesto, el problema del viajante. Siguiendo las directrices indicadas, se han estudiado y diseñado diferentes algoritmos "Greedy" que aproximan el problema, con el objetivo final de comparar cual nos proporciona mejores resultados en cuanto a términos de eficiencia y precisión. Para determinar esto último, también cabe destacar detalles de menor importancia como la complejidad de los algoritmos, la estabilidad frente a distintas entradas o la posibilidad de mejora de los mismos mediante la elección de distintos puntos de partida o con mejoras posteriores como pueden ser aquellas que proporcionan algoritmos como λ -opt o genéticos.

De aquí en adelante, describiremos por secciones los algoritmos implementados y al final proporcionaremos una sección comparativa sobre la cual nos basaremos para determinar conclusiones, en particular, elegiremos el algoritmo cuya solución consideremos más conveniente. En lo que sigue se muestra únicamente el resultado obtenido por el algoritmo sin mejoras posteriores, este tipo de mejoras suponen una pequeña desvirtuación del objetivo de encontrar el mejor algoritmo y es por ello, que su uso se limita a la conclusión final.

6.1 Algoritmo *Nearest Neighbour*

Tal y como el nombre indica, el primer algoritmo implementado no es nada más, ni nada menos que el primer algoritmo que probablemente se le puede ocurrir a cualquier persona que se enfrente a este problema. La idea es sencilla e intuitiva, nos dan un conjunto de puntos y queremos encontrar el camino más corto que los recorra, para ello, **elegimos** un punto inicial y a partir de ese punto, en un proceso iterativo, elegimos en cada paso el siguiente punto más cercano al actual, o lo que es lo mismo, el punto para el cual la distancia al punto actual es la menor (No confundir con el punto para el cual la distancia total es la menor, ya que, aunque parecido, no es lo mismo). Este proceso se repite hasta que todos los puntos han sido visitados.

A continuación se proporciona la implementación propuesta del algoritmo en cuestión:

```
1 vector<Point> nearestNeighborTSP(const vector<Point>& points) {
2     vector<Point> path;
3     vector<bool> visited(points.size(), false);
4     path.reserve(points.size());
5     path.emplace_back(points[0]);
6     visited[0] = true;
7
8     for (int i = 0; i < points.size() - 1; ++i) {
9         double minDistance = numeric_limits<double>::max();
10        int nearestNeighbor = -1;
```

```

11     for (int j = 0; j < points.size(); ++j) {
12         if (!visited[j]) {
13             double distance = path[i].distanceTo(points[j]);
14             if (distance < minDistance) {
15                 minDistance = distance;
16                 nearestNeighbor = j;
17             }
18         }
19     }
20     path.emplace_back(points[nearestNeighbor]);
21     visited[nearestNeighbor] = true;
22 }
23
24 return path;
25 }

```

Como se puede apreciar, se ha hecho uso de un vector de puntos visitados (Programación dinámica) para evitar visitar un punto más de una vez, esto no supone una complejidad notable, sin embargo, si que supone una gran mejora en términos de eficiencia.

6.2 Algoritmo Ordenación

Como el nombre indica, este algoritmo consiste en ordenar los puntos de partida de acuerdo a un criterio específico (en nuestro caso los hemos ordenado por la coordenada x de menor a mayor, teniendo en cuenta también para puntos cercanos en x, la coordenada y). Una vez ordenados los puntos, simplemente recorreremos el vector de puntos en el orden en el que se encuentren, de esta forma, el camino total trata de reducir cruces y distancias con la idea intuitiva de que si dos puntos están cerca en el plano, probablemente, la distancia que se recorra al pasar por ellos, sea mínima. Como veremos en el siguiente algoritmo y en la conclusión, esto en la práctica no es del todo cierto. Primero veamos la implementación del algoritmo:

```

1 vector<Point> orderedTSP(const vector<Point>& points) {
2     vector<Point> tour;
3     tour.reserve(points.size());
4     copy(points.begin(), points.end(), back_inserter(tour));
5     sort(tour.begin(), tour.end());
6
7     return tour;
8 }

```

Como se puede apreciar, la implementación es muy sencilla, simplemente se copian los puntos de partida a un vector auxiliar, se ordenan y se devuelve el vector ordenado, sin embargo cuando vemos el resultado de la ejecución, nos damos cuenta de que aunque los puntos esten muy cerca respecto a la coordenada x, los saltos que se dan en la coordenada y pueden ser muy grandes, lo que hace que el camino total sea mucho mayor de lo esperado, además para cerrar el camino, la distancia es la máxima posible en cuanto a x. Para mejorar esto, podemos observar que si nuestro problema son los saltos en la coordenada y y la distancia de cierre, podemos intentar minimizarlos, de donde surge el siguiente algoritmo.

6.3 Algoritmo Circular

Para solucionar el problema anterior, se propone una alternativa que se basa en la misma idea de ordenar por la coordenada x, sin embargo, en este caso la coordenada y solo se tiene en cuenta tras ordenar, lo que se hace es guardar el la coordenada y de los puntos máximo y mínimo y se calcula el punto medio entre ambos, esto se

hace para recorrer los puntos de forma circular como el nombre indica, de modo que los puntos que se quedan en la parte de abajo del plano, se recorren en sentido horario y los puntos que se quedan en la parte de arriba del plano, se recorren en sentido antihorario. De esta forma, se consigue minimizar la cantidad de saltos en el eje y y la distancia de cierre, resulta sugerente pensar que este proceso de división en dos partes, se puede hacer de forma recursiva, sin embargo, en la práctica, esto supone una complejidad superior a la hora de implementar mas bucles, y no supone una mejora significativa en términos de precisión, por lo que se ha optado por dejarlo de esta forma.

El problema que presenta esta implementación es que dependiendo de la forma que tenga el conjunto de puntos que se le pase, se obtendrán resultados muy distintos, sin embargo esto ocurre en todos los algoritmos, por lo que no se considera un problema en si mismo, ya que es inherente al problema. A continuación se muestra la implementación del algoritmo:

```

1  vector<Point> CircTSP(const vector<Point>& points) {
2      vector<Point> tour;
3      vector<Point> circularTour;
4      tour.reserve(points.size());
5      circularTour.reserve(points.size());
6      tour = points;
7      sort(tour.begin(), tour.end());
8
9      int min = numeric_limits<int>::max();
10     int max = numeric_limits<int>::min();
11
12     for (int i = 0; i < points.size(); ++i) {
13         min = min(min, points[i].getY());
14         max = max(max, points[i].getY());
15     }
16
17     double mid = (max + min )/2;
18
19     for (int i = 0; i < points.size(); ++i) {
20         if (tour[i].getY() <= mid) {
21             circularTour.emplace_back(tour[i]);
22         }
23     }
24
25     for (int i = points.size(); i >= 0; --i) {
26         if (tour[i].getY() > mid) {
27             circularTour.emplace_back(tour[i]);
28         }
29     }
30
31     return circularTour;
32 }

```

Una vez mas, podemos apreciar que se puede aumentar la eficiencia del algoritmo optimizando bucles y la ordenación de manera que el máximo y el mínimo se calculen mientras se ordenan los puntos, sin embargo, esto no supone una mejora en cuanto al orden de eficiencia y si supone una mayor complejidad del codigo, por lo tanto, como el compilador se encarga de optimizar esos detalles que no afectan al orden de eficiencia, se ha optado por dejarlo de esta forma para facilitar la lectura, en una implementación real, se recomendaría la optimización manual de estos detalles, ya que, aunque bueno, el compilador no es perfecto y no siempre optimiza de la mejor forma.

6.4 Conclusiones

Una vez presentados los tres algoritmos, nos disponemos a obtener conclusiones acerca de ellos así como a compararlos para determinar cual es el mejor, o cómo podríamos mejorar la precisión de los mismos. Comenzamos mostrando visualmente los resultados de ejecución de cada uno de ellos, con el objetivo de que el lector pueda apreciar de forma sencilla la idea que se ha querido transmitir con cada uno de los algoritmos.

6.4.1 Graficas

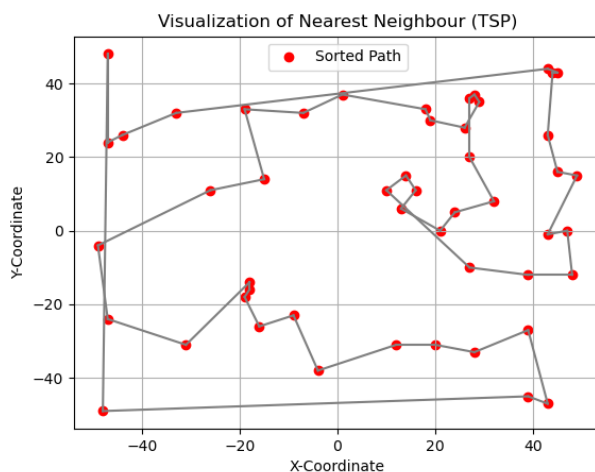


Figura 6.1: Nearest Neighbour

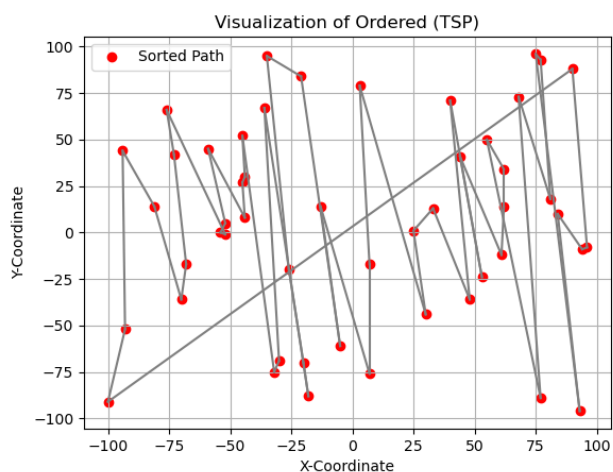


Figura 6.2: Ordenación

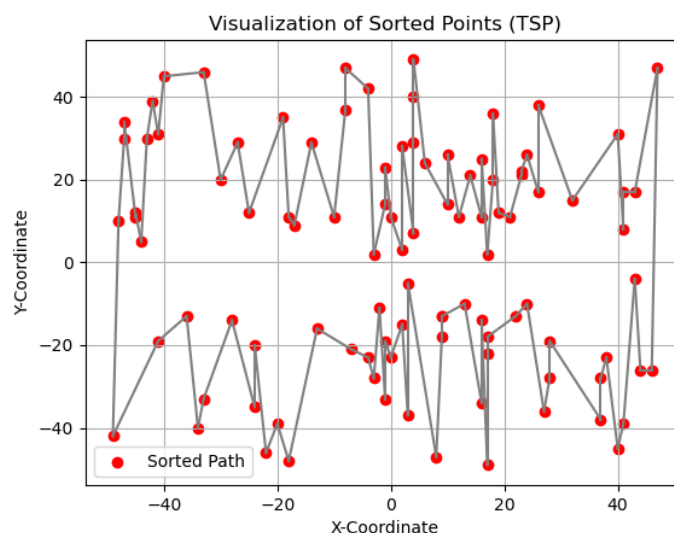


Figura 6.3: Circular

Una vez graficadas, podemos ver los problemas principales de cada uno de los algoritmos, en el caso de *Nearest Neighbour*, se puede apreciar que la distancia de cierre es muy grande, sin embargo, aparenta un camino pare-

cido al óptimo, en el caso de *Ordenación*, se puede apreciar que, mientras los puntos x están cerca, los saltos en y son muy grandes, y además la distancia de cierre es máxima, en el caso de *Circular*, se puede apreciar que los saltos en y son más pequeños y la distancia de cierre es menor.

De este análisis visual, podemos deducir que el algoritmo *Nearest Neighbour* es el que mejor resultado proporciona, le sigue *Circular* y por último *Ordenación*. Podríamos dejar el estudio aquí, sin embargo, se proponen maneras de mejorar los algoritmos.

6.4.2 Mejoras

Para mejorar y comparar los algoritmos, comenzamos notando que en el caso de *Nearest Neighbour*, la elección del punto de partida nos proporciona un camino distinto, por lo que, para mejorar la precisión del algoritmo, se puede ejecutar el algoritmo con distintos puntos de partida y quedarnos con el mejor resultado. En los casos de *Ordenación* y *Circular*, esto no tendría mucho sentido, sin embargo podríamos jugar con la ordenación respecto de un eje u otro, o con la división en dos partes, respectivamente, para mejorar la precisión de los mismos.

Este tipo de mejoras, son sencillas y no suponen un gran aumento en cuanto al tiempo de ejecución, sin embargo, tampoco suponen una mejora significativa en cuanto a la precisión. Su mejor virtud es que nos proporcionan una velocidad de ejecución muy rápida y una precisión aceptable, por lo que, en la práctica, se recomienda su uso, sin embargo, habrá aplicaciones en las que se necesite una precisión mayor, en cuyo caso se recomienda el uso de algoritmos más complejos como λ -opt o genéticos.

6.4.3 Precisión y Tiempos

En esta sección se proporcionan los tiempos de ejecución de los algoritmos y se comparan con la precisión de los mismos, para ello, se han ejecutado los algoritmos sobre un entorno cuya solución óptima se conoce, de esta forma, se puede comparar la solución obtenida con la solución óptima, el entorno en cuestión es un conjunto de Países del mundo que se han dividido en pueblos y ciudades de interés. A continuación mostramos los resultados que hemos obtenido:

Comenzamos exponiendo los resultados que arrojan los algoritmos sin mejoras:

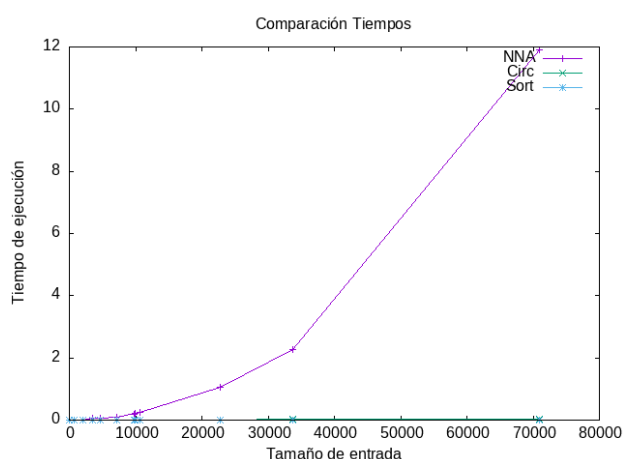


Figura 6.4: Tiempos completos

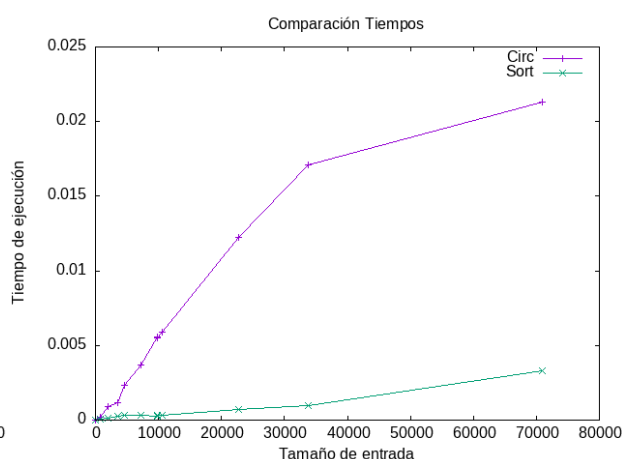


Figura 6.5: Tiempos Circ y Sort

Como se puede ver, el algoritmo *Nearest Neighbour* es el más lento, esto era de esperar debido a que, como

veremos ahora, es el que mejor precisión ofrece. El mismo razonamiento se puede aplicar a los algoritmos *Ordenación* y *Circular*. Si nos fijamos ahora en los tiempos de ejecución de los algoritmos tras las mejoras,

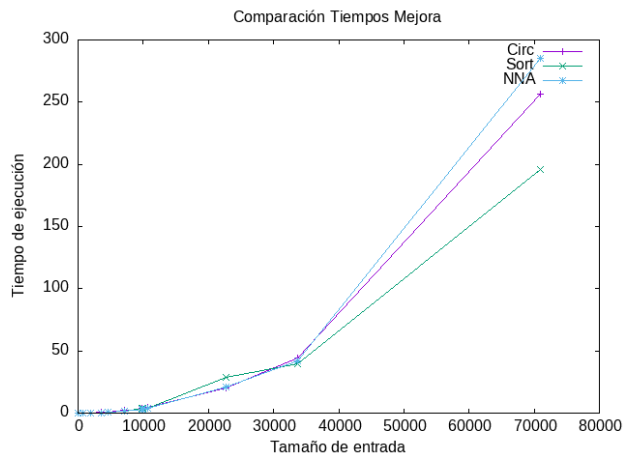


Figura 6.6: Tiempos completos

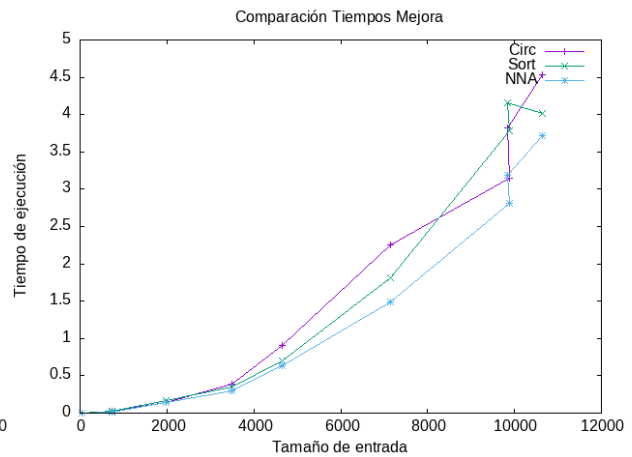


Figura 6.7: Tiempos menor Tamaño

Podemos ver como hay un gran aumento en el tiempo, sin embargo, son tiempos similares entre ellos, por lo que nos interesa quedarnos con el que mejor precisión ofrece:

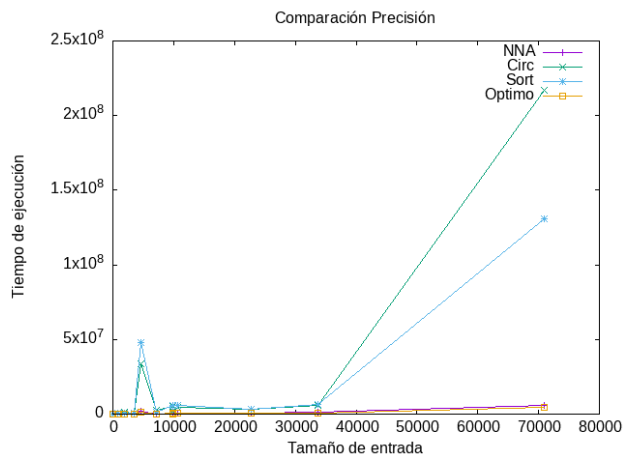


Figura 6.8: Precisión sin mejoras

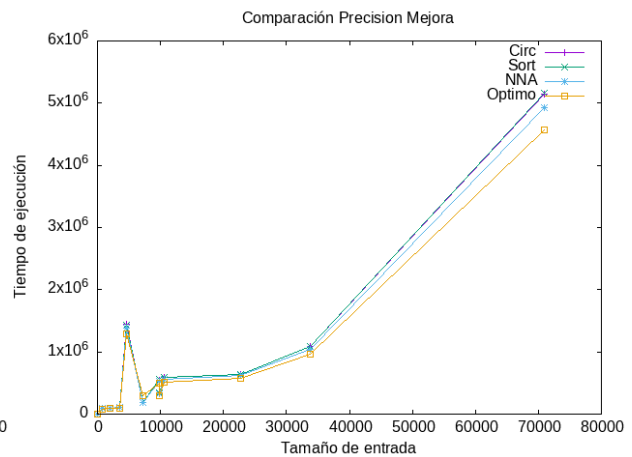


Figura 6.9: Precisión tras mejora

Como se puede ver, todos los algoritmos con mejora, se quedan muy cerca de la solución óptima, sin embargo, el algoritmo *Nearest Neighbour* es el que mejor precisión ofrece, tanto en el caso de mejora como en el caso sin mejora, por lo que, en la práctica, se recomienda su uso. Para aplicaciones donde la precisión sea un factor crítico, se recomienda el uso combinado de distintas mejoras como la elección de distintos puntos de partida o el uso de algoritmos más complejos como λ -opt o genéticos.