Laboratorio N°5
Estructura de datos y algoritmos

Integrantes: Hugo Rojas e Isidora González
Profesor laboratorio: Mauricio Hidalgo
Profesor cátedra: Yerko Ortiz
Sección 04
Mails: hugo.rojas1@mail.udp.cl isidora.gonzalez4@mail.udp.cl

Introducción

El siguiente informe trata sobre un laboratorio que tiene como objetivo crear un código que resuelva el problema de optimización de rutas para los conductores de Uber, en donde hay que buscar la forma más eficiente para atender las solicitudes de los clientes de la aplicación, así buscando la ruta más corta y de menor costo por recorrer.

Para resolver el problema planteado se hizo la implementación de 2 algoritmos, uno de ellos se llama Greedy, el cual toma una decisión local óptima cada vez sin tomar en cuenta las posibles consecuencias a largo plazo y el otro llamado Random (algoritmo aleatorio uniforme) el cual escoge una solución al azar de entre todas las alternativas posibles sin favorecer o tener preferencia con alguna de ellas. Con estos algoritmos será posible poder cumplir con las solicitudes de los clientes de la manera más eficiente y optimizada.

Para poder llevar a cabo un análisis de estas soluciones y poder hacer sus respectivas comparaciones, se hizo una serie de construcción de gráficos, en donde es más sencillo visualizar el cambio que tiene cada algoritmo según el costo. Los gráficos son distintos, ya que estos dependen de cada input que arroja un algoritmo según su horario de mañana, tarde o noche, por lo que ahí demuestran ser diferentes en base a la estructura de cada algoritmo.

Los inputs que se utilizaron consisten en la configuración de cada vehículo de Uber en el horario de mañana, tarde y noche y también las solicitudes de los clientes. Los archivos **“manana.dat”, “tarde.dat” y “noche.dat”** contienen la posición inicial de los vehículos en el horario correspondiente, mientras que los archivos **"requests1.dat", "requests2.dat" y "requests3.dat"** contiene las solicitudes de los clientes, entregando su coordenada de origen, destino y el tiempo en el que se realiza la solicitud. Gracias a estos archivos es posible poder simular cómo sería programar la aplicación de Uber e intentar obtener los resultados de optimización de tiempos, destinos y rutas de manera realista según la cantidad y posición de los vehículos, además de las solicitudes de los clientes.

A continuación será posible entender detenida y detalladamente el paso a paso de cómo se creó este código para poder llevar a cabo el objetivo, dejando a conocer las dificultades, problemas y estrategias que fueron utilizadas para poder lograr el correcto funcionamiento del código con los algoritmos propuestos.

Algoritmo Greedy

El algoritmo Greedy que se utiliza en el código resuelve el problema de asignar vehículos de Uber a las solicitudes de los clientes de manera rápida, por lo que toma decisiones locales óptimas en cada solicitud sin tener en consideración las posibles consecuencias en un plazo.
A continuación, se hará una descripción en detalle de la lógica del algoritmo:

1. Se extraen de los archivos entregados las ubicaciones de los vehículos de Uber y las solicitudes de los clientes para la mañana, tarde y noche.

```java
public static ArrayList<Coordenada> Ubicaciones(String archivo){
    ArrayList<Coordenada> ubicaciones = new ArrayList<>();
    try(BufferedReader br = new BufferedReader(new FileReader(archivo))){
        String linea;
        while((linea = br.readLine()) != null){
            String[] partes = linea.split( regex: " ");
            int x = Integer.parseInt(partes[0]);
            int y = Integer.parseInt(partes[1]);
            ubicaciones.add(new Coordenada(x, y));
        }
    } catch (IOException e){
        e.printStackTrace();
    }
    return ubicaciones;
}
```

ilustración 1: Método Ubicaciones(para los distintos horarios).

```java
public static ArrayList<Solicitud> Solicitudes(String archivo) {
    ArrayList<Solicitud> solicitudes = new ArrayList<>();
    try (BufferedReader br = new BufferedReader(new FileReader(archivo))){
        String Archivo;
        while ((Archivo = br.readLine()) != null){
            String[] partes = Archivo.split( regex: " ");
            int xi = Integer.parseInt(partes[0]);
            int yi = Integer.parseInt(partes[1]);
            int xf = Integer.parseInt(partes[3]);
            int yf = Integer.parseInt(partes[4]);
            int T = Integer.parseInt(partes[6]);
            Coordenada origen = new Coordenada(xi, yi);
            Coordenada destino = new Coordenada(xf, yf);
            solicitudes.add(new Solicitud(origen, destino, T));
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    return solicitudes;
}
```

ilustración 2: Método Solicitudes(para los archivos request).

2. Para cada input de ubicaciones y solicitudes según el horario que se entregue, el algoritmo itera sobre cada solicitud y busca el Uber más cercano disponible para llevar a cabo el viaje.

```
public static ArrayList<String> TransformarSolicitudes(ArrayList<Coordenada> ubicaciones, ArrayList<Solicitud> solicitudes){
    ArrayList<String> Distancias = new ArrayList<>();
    for (Solicitud solicitud : solicitudes){
        Coordenada Uber = UberMasProximo(ubicaciones, solicitud);
        if (Uber != null){
            int distancia = calcularDistancia(Uber, solicitud.destino);
            Distancias.add(solicitud.TLlegada + " " + distancia);
        }
    }
    return Distancias;
}
```

Ilustración 3: Método TransformarSolicitudes.

3. Para encontrar el Uber más cercano se utiliza el método **"UberMasProximo"** el cual calcula la distancia entre cada ubicación y el origen de la solicitud utilizando la distancia de Manhattan (suma de las diferencias absolutas de las coordenadas x e y), luego de eso se selecciona el Uber con la distancia mínima y se determina como el más cercano.

```
public static Coordenada UberMasProximo(ArrayList<Coordenada> ubicaciones, Solicitud solicitud){
    double distanciaMinima = Double.MAX_VALUE;
    Coordenada Uber = null;
    for (Coordenada ubicacion : ubicaciones){
        double distancia = calcularDistancia(ubicacion, solicitud.origen);
        if (distancia < distanciaMinima){
            distanciaMinima = distancia;
            Uber = ubicacion;
        }
    }
    if (Uber != null){
        ubicaciones.remove(Uber);
    }
    return Uber;
}
```

ilustración 4: Método UberMasProximo.

4. Si se encuentra un Uber disponible se calcula la distancia desde la ubicación del Uber hasta el destino de la solicitud utilizando el método **"calcularDistancia"**, en donde esta distancia calculada se agrega a una lista de distancias acumuladas para un determinado horario.

```
public static int calcularDistancia(Coordenada origen, Coordenada destino) {
    int dx = Math.abs(origen.x - destino.x);
    int dy = Math.abs(origen.y - destino.y);
    return (int) Math.round(Math.sqrt(dx * dx + dy * dy));
}
```

ilustración 5: Método CalcularDistancia.

5. Una vez que se han procesado todas las solicitudes para cierto horario determinado, se llama al método **"Guardar"** para almacenar las distancias acumuladas en un archivo de output correspondiente al horario.

```
public static void Guardar(ArrayList<String> Distancias, String horarios){
    String archivo = "C:\\RESULTADOS_" + horarios.toUpperCase() + ".txt";
    try (FileWriter writer = new FileWriter(archivo)){
        for (String resultado : Distancias){
            writer.write( str: resultado + "\n");
        }
    } catch (IOException e){
        e.printStackTrace();
    }
}
```

ilustración 6: Método Guardar.

Los inputs correspondientes a cada horario (mañana, tarde y noche) tienen una estructura de dos columnas, en donde la primera columna representa el instante de tiempo y la segunda columna el costo total acumulado.

```
9 425
14 844
40 659
67 561
67 710
73 299
89 634
99 612
129 482
130 97
137 675
175 351
177 247
186 297
197 295
```

Ilustración 7: Extracto de input de horario mañana.

Para poder llevar a cabo un análisis más visual, estos datos se llevaron a gráficos de función costo/tiempo en donde es posible evidenciar el tiempo como una variable independiente y el costo como dependiente.

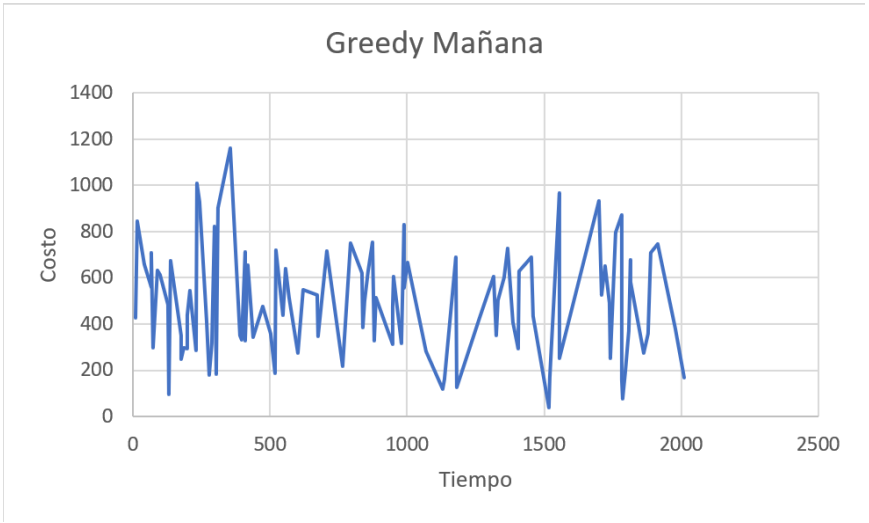

Ilustración 8: Gráfico de la recopilación de datos de horario mañana.

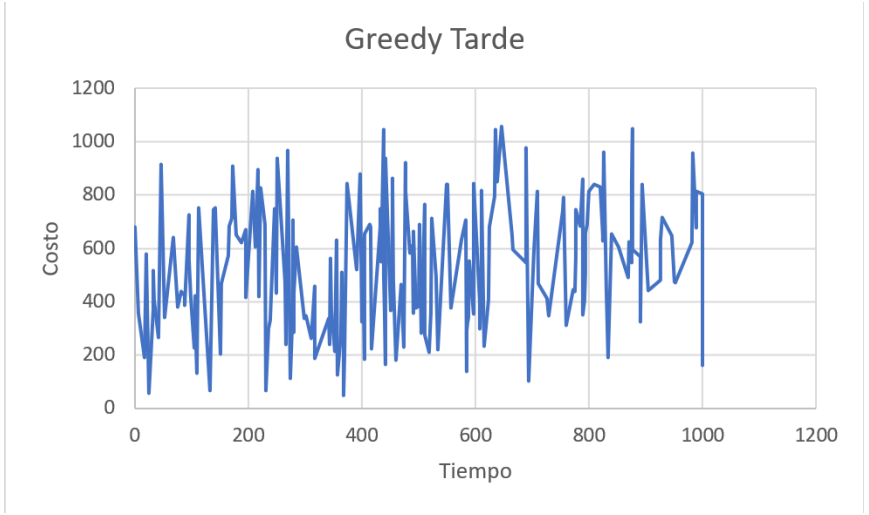

Ilustración 9: Gráfico de la recopilación de datos de horario tarde.

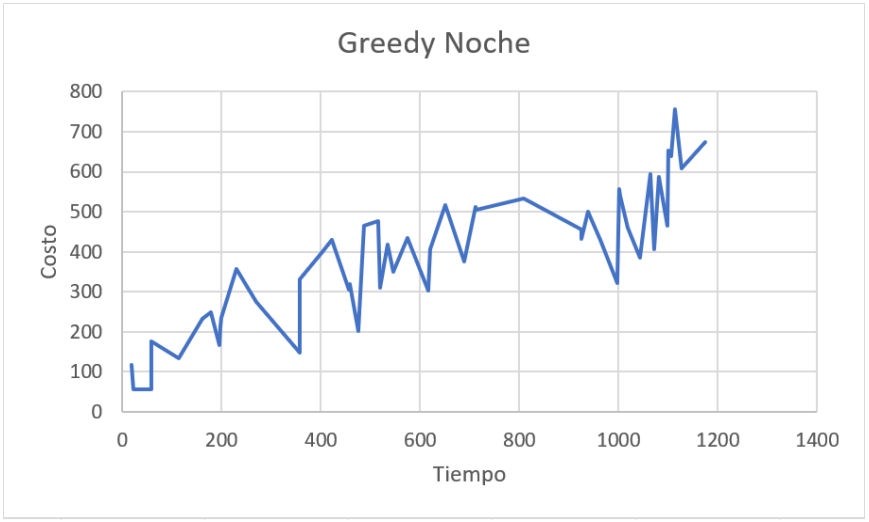

Ilustración 10: Gráfico de la recopilación de datos de horario noche.

Algoritmo aleatorio uniforme

Este algoritmo utilizado en el código resuelve el problema de asignación de vehículos de Uber a las solicitudes de los clientes de manera aleatoria y uniforme. A continuación, se describe en detalle la implementación del algoritmo y cómo se generan las decisiones aleatorias:

1. El algoritmo sigue una estructura similar al algoritmo Greedy, comenzando por leer los archivos con las ubicaciones de los vehículos de Uber y las solicitudes de los clientes para los variados horarios.

```java
public static ArrayList<Coordenada> Ubicaciones(String archivo){
    ArrayList<Coordenada> ubicaciones = new ArrayList<>();
    try (BufferedReader br = new BufferedReader(new FileReader(archivo))){
        String Archivo;
        while ((Archivo = br.readLine()) != null){
            String[] partes = Archivo.split( regex: " ");
            int x = Integer.parseInt(partes[0]);
            int y = Integer.parseInt(partes[1]);
            ubicaciones.add(new Coordenada(x, y));
        }
    } catch (IOException e){
        e.printStackTrace();
    }
    return ubicaciones;
}
```

ilustración 11: Método Ubicaciones.

```java
public static ArrayList<Solicitud> Solicitudes(String archivo){
    ArrayList<Solicitud> solicitudes = new ArrayList<>();
    try (BufferedReader br = new BufferedReader(new FileReader(archivo))){
        String Archivo;
        while ((Archivo = br.readLine()) != null){
            String[] partes = Archivo.split( regex: " ");
            int xi = Integer.parseInt(partes[0]);
            int yi = Integer.parseInt(partes[1]);
            int xf = Integer.parseInt(partes[3]);
            int yf = Integer.parseInt(partes[4]);
            int T = Integer.parseInt(partes[6]);
            Coordenada origen = new Coordenada(xi, yi);
            Coordenada destino = new Coordenada(xf, yf);
            solicitudes.add(new Solicitud(origen, destino, T));
        }
    } catch (IOException e){
        e.printStackTrace();
    }
    return solicitudes;
}
```

ilustración 12: Método Solicitudes.

2. Para cada conjunto de ubicaciones y solicitudes correspondiente a un horario determinado, el algoritmo itera sobre cada solicitud y toma una decisión aleatoria para asignar un Uber.

```
public static ArrayList<String> TransformarSolicitudes(ArrayList<Coordenada> ubicaciones, ArrayList<Solicitud> solicitudes){
    ArrayList<String> Distancias = new ArrayList<>();
    for (Solicitud solicitud : solicitudes){
        Coordenada Uber = UberMasProximo(ubicaciones, solicitud);
        if (Uber != null){
            int distancia = calcularDistancia(Uber, solicitud.destino);
            Distancias.add(solicitud.TLlegada + " " + distancia);
        }
    }
    return Distancias;
}
```

ilustración 13: Método TransformarSolicitudes.

3. En cada paso, se generan decisiones aleatorias mediante el método **"UberAleatorio"**, el cual selecciona un Uber disponible al azar de entre todas las ubicaciones de vehículos, esto es posible generando un índice aleatorio dentro del rango de la lista de ubicaciones y seleccionando el Uber correspondiente a ese índice.

```
public static Coordenada UberAleatorio(ArrayList<Coordenada> ubicaciones){
    if (ubicaciones.isEmpty()){
        return null;
    }
    Random random = new Random();
    int aux = random.nextInt(ubicaciones.size());
    Coordenada Uber = ubicaciones.get(aux);
    ubicaciones.remove(aux);
    return Uber;
}
```

ilustración 14: Método UberAleatorio.

4. Si se encuentra un Uber disponible, se calcula la distancia desde la ubicación del Uber hasta el destino de la solicitud utilizando el método **"calcularDistancia"**, esta distancia se agrega a la lista de distancias acumuladas para el determinado horario.

```
public static int calcularDistancia(Coordenada origen, Coordenada destino){
    int dx = Math.abs(origen.x - destino.x);
    int dy = Math.abs(origen.y - destino.y);
    return (int) Math.round(Math.sqrt(dx * dx + dy * dy));
}
```

ilustración 15: Método CalcularDistancia.

5. Se repite el proceso para todas las solicitudes del horario actual y se almacenan las distancias acumuladas utilizando el método **"Guardar"**.

```java
public static void Guardar(ArrayList<String> Distancias, String horarios){
    String archivo = "C:\\RESULTADOS_" + horarios.toUpperCase() + ".txt";
    try (FileWriter writer = new FileWriter(archivo)){
        for (String resultado : Distancias){
            writer.write( str: resultado + "\n");
        }
    } catch (IOException e){
        e.printStackTrace();
    }
}
```

ilustración 16: Método Guardar.

Cabe destacar que al utilizar el algoritmo aleatorio pueden haber casos en los que al ejecutarse den resultados diferentes, esto se debe a la naturaleza de hacer elecciones aleatorias, pero de todos modos el algoritmo se asegura de asignar un Uber disponible para cada solicitud, sólo que como fue mencionado anteriormente puede variar en cada ejecución.

En los inputs correspondientes al algoritmo Random se puede visualizar a simple vista el cambio del costo debido a este mismo algoritmo.

```
9 583
14 626
40 453
67 274
67 400
73 511
89 515
99 660
129 381
130 464
137 78
175 935
177 799
186 490
197 608
```

Ilustración 17: Extracto de input de horario mañana.

Como se indicó anteriormente, los inputs de esta compilación tienen la misma estructura que en el algoritmo Greedy, por lo cual se procede a llevar dichos datos a gráficos.

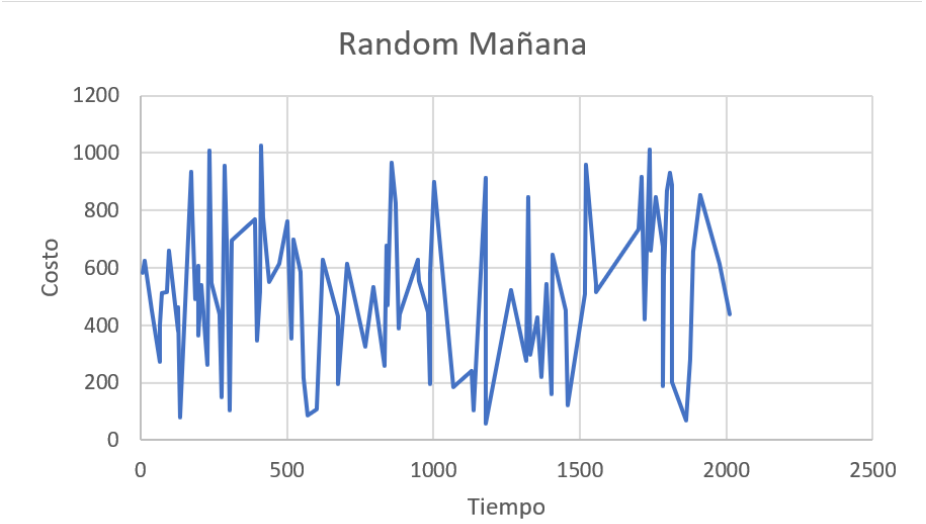

ilustración 18: Gráfico de la recopilación de datos de horario mañana.

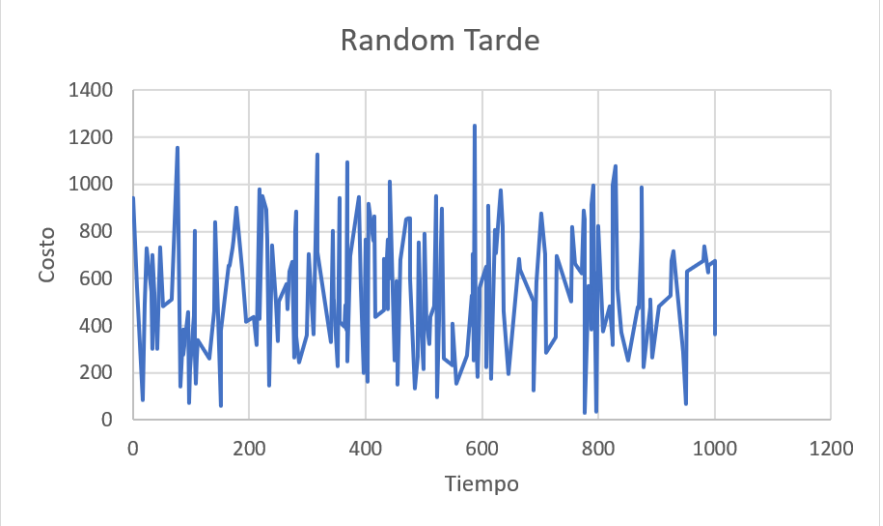

ilustración 19: Gráfico de la recopilación de datos de horario tarde.

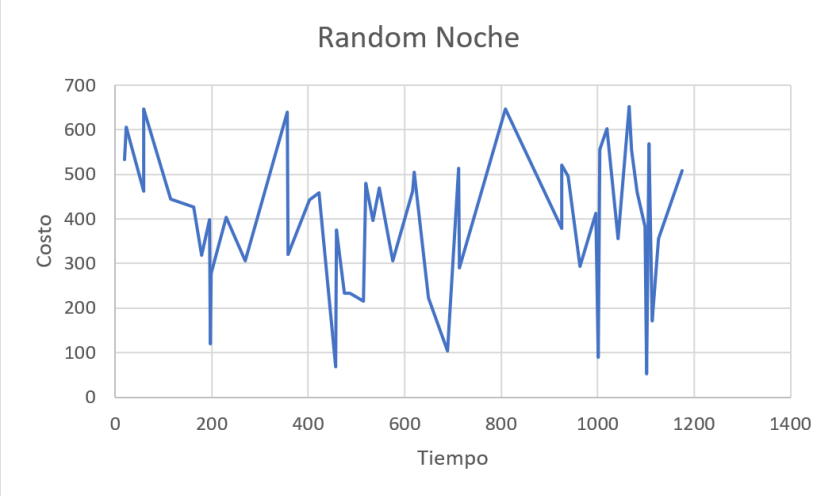

ilustración 20: Gráfico de la recopilación de datos de horario Noche.

Comparación de soluciones

A continuación, se hará una comparación y contraste de las soluciones proporcionadas por los algoritmos Greedy y Random en términos de eficiencia, calidad de la solución y otras métricas relevantes, también se identifican patrones y tendencias observadas en diferentes inputs analizados:

1. Eficiencia:

- Greedy: Tiene una eficiencia relativamente alta, ya que utiliza un enfoque determinista para seleccionar el Uber más cercano a cada solicitud, lo cual implica que las asignaciones se realizan en función de la distancia mínima en cada paso, lo que puede llevar a una menor cantidad de movimientos totales y menor tiempo de ejecución.

- Aleatorio uniforme: Este algoritmo es menos eficiente en términos de tiempo de ejecución, ya que se basa en decisiones aleatorias en cada paso, por lo que pueden haber más movimientos y un mayor tiempo de ejecución en comparación con el algoritmo Greedy.

2. Calidad de la solución:

- Greedy: Tiende a proporcionar soluciones de alta calidad, ya que selecciona el Uber más cercano en cada paso, por lo que las distancias totales recorridas son menores y se logra una asignación más eficiente.

- Aleatorio uniforme: Este algoritmo puede proporcionar soluciones de calidad variable debido a su naturaleza aleatoria, aunque de todas maneras es posible garantizar que todas las solicitudes serán atendidas.

3. Otros métricas relevantes:

- Greedy: Este algoritmo tiende a tener un menor costo total acumulado, lo que implica que se recorre una menor distancia en total, esto puede resultar en un menor consumo de combustible y menor desgaste de los vehículos.

- Aleatorio uniforme: Este algoritmo puede tener un costo total acumulado mayor debido a las decisiones aleatorias tomadas en cada paso, esto puede resultar en una mayor distancia recorrida y por lo tanto, un mayor consumo de combustible y desgaste de los vehículos.

4. Puntos fuertes y debilidades:

- Greedy:

1. Puntos fuertes: Soluciones de alta calidad, menor distancia total recorrida, menor tiempo de ejecución.
2. Debilidades: No considera otras variables más allá de la distancia mínima, puede haber situaciones donde la asignación óptima no sea alcanzada.

- Aleatorio uniforme:

1. Puntos fuertes: Capacidad para generar soluciones diferentes, garantiza que todas las solicitudes sean atendidas.
2. Debilidades: Mayor costo total acumulado, tiempo de ejecución más largo, menor calidad de la solución en comparación con Greedy.

5. Patrones y tendencias observadas en diferentes inputs:

El algoritmo Greedy es más eficiente y proporciona soluciones de mayor calidad en términos de distancia total recorrida, sin embargo, el algoritmo Random tiene la ventaja de generar soluciones diferentes y garantizar que todas las solicitudes sean atendidas. La elección entre ambos algoritmos dependerá de las prioridades del problema y la importancia relativa de la eficiencia y la calidad de la solución.

Conclusiones

En base a todo este laboratorio, es posible concluir que el algoritmo Greedy es mucho más eficiente y proporciona soluciones de mayor calidad en base a la distancia recorrida total, ya que asigna de manera más eficiente los vehículos según las solicitudes, además de que tiene un menor costo total acumulado, lo que implica menos consumo de combustible. Por otro lado, el algoritmo Random tiene la ventaja de poder crear soluciones distintas y que todas las solicitudes sean atendidas y puede mostrar una mayor variabilidad en los resultados.

Según lo mencionado anteriormente, es necesario recomendar el análisis del contexto y las necesidades del problema al elegir entre estos dos algoritmos utilizados, así teniendo en cuenta la importancia de la eficiencia y calidad de la solución. Una de las maneras para poder saber qué algoritmo es correcto es a través de la exploración de variantes y algoritmos utilizables, realizando pruebas entre los códigos con diferentes conjuntos de datos, así adaptando a cada situación el algoritmo escogido. Estas recomendaciones buscan mejorar la eficacia y la calidad de las soluciones, y proporcionar opciones más flexibles y adaptadas a las restricciones y condiciones particulares.