INSTITUTO TECNOLÓGICO SUPERIOR DEL SUR DE GUANAJUATO

SISTEMA DE RUTA ÓPTIMA Y PLAN DE VIAJE PARA EL SISTEMA DE TRANSPORTE PÚBLICO

Opción 2: Titulación Integral – Tesis profesional

Elaborada por:

Oswaldo Castro Tinoco

Que presenta para obtener el título de:

INGENIERO EN SISTEMAS COMPUTACIONALES

Asesor:

Dr. Rogelio Hasimoto Beltrán

Co-Asesor:

Dr. Luis Germán Gutiérrez Torres

Uriangato, Gto. Mayo, 2023

“Sistema de ruta óptima y plan de viaje para el sistema de transporte público”

Elaborada por:

Oswaldo Castro Tinoco

Aprobado por. . . . . Dr. Luis Germán Gutiérrez Torres
Docente de la carrera de Ingeniería en Sistemas Computacionales
Asesor de opción 2: Titulación Integral – Tesis profesional

Revisado por. . . . . MTW. Efrén Vega Chávez
Docente de la carrera de Ingeniería en Sistemas Computacionales
Revisor de opción 2: Titulación Integral – Tesis profesional

Revisado por. . . . . Ing. Diego Jesús Morales Mejía
Docente de la carrera de Ingeniería en Sistemas Computacionales
Revisor de opción 2: Titulación Integral – Tesis profesional

3.- LIBERACIÓN DE PROYECTO PARA LA TITULACIÓN INTEGRAL

Instituto Tecnológico Superior del Sur de Guanajuato

TECNOLOGÍA Y CALIDAD PARA LA VIDA

LIBERACIÓN DE PROYECTO PARA LA TITULACIÓN INTEGRAL

Uriangato,Gto., 21/abril/2023

Asunto: Liberación de proyecto para la titulación integral

Ing. José Gabriel Aguilera González
Director Académico
ITSUR
PRESENTE

Por este medio informo que ha sido liberado el siguiente proyecto para la titulación integral:

Nombre de estudiante y/o egresado(a): Oswaldo Castro Tinoco	
Carrera: Ingeniería en Sistemas Computacionales	Núm. de control: S16120248
Nombre del proyecto: Sistema de ruta óptima y plan de viaje para el sistema de transporte público	
Producto: Opción 2: Titulación Integral – Tesis profesional	

Agradezco de antemano su valioso apoyo en esta importante actividad para la formación profesional de nuestras y nuestros egresados.

ATENTAMENTE

Mca. Miguel Cruz Pineda
Jefe de División de Ingeniería en Sistemas Computacionales
ITSUR

Instituto Tecnológico Superior del Sur de Guanajuato

COORDINACIÓN INGENIERÍA SISTEMAS COMPUTACIONALES

La comisión revisora ha tenido a bien aprobar la reproducción de este trabajo.

DR. LUIS GERMAN GUTIERREZ TORRES	MTW. EFREN VEGA CHAVEZ	ING. DIEGO JESUS MORALES MEJIA

c.c.p.- Expediente

Resumen

El sistema de transporte público es importante en prácticamente todas las ciudades de México debido a su alta demanda. Y al ser tan usado se debe de encontrar una manera eficaz para hacer uso del sistema de transporte público.

Por eso en el presente trabajo se propuso la implementación de este sistema de ruta óptima y plan de viaje. Este sistema ayudará a los usuarios del transporte público a encontrar el camino más corto haciendo uso de autobuses para ir de un punto a otro.

Esto es de gran ayuda, especialmente para las personas que usen el sistema y que no conozcan la ciudad donde se encuentran y necesiten recorrer grandes distancias. Para encontrar la ruta óptima se usó teoría de grafos, usando específicamente el algoritmo de Dijkstra y Web Sockets para establecer comunicación de un cliente en una aplicación móvil para dispositivos Android, a un servidor donde se calcula el plan de viaje.

Palabras clave

Dijkstra, Transporte público, Teoría de grafos, Sockets, Ruta óptima

Abstract

The public transport system is important in most of the cities of Mexico due to its high demand. And being so used, an efficient way must be found to make use of the public transport system.

Thats why in the present work the implementation of this optimal route and trip plan system was proposed. This system will help to the users of the public transport to find the shortest path using buses to go from one point to another.

This is very helpful, especially for the people using the system and don’t know the city where they are and need to travel long distances. The graph theory was used to

find the optimal route, using specifically the Dijkstra algorithm and Web Sockets to set the communication between a client in the app for Android, to a server where the trip plan is calculated.

Keywords

Dijkstra, Public transport, Graph theory, Sockets, Optimal route

Contenido

Capítulo 1 10
Introducción. 10
1.1 Identificación 11
1.2 Justificación 12
1.3 Alcance 13
Capítulo 2 14
Fundamento teórico 14
2.1 Teoría de Grafos 14
2.2 Algoritmo de Dijkstra 15
2.3 Fórmula de Haversine 21
2.4 Python 24
2.5 NetworkX 25
2.6 Java 26
2.7 Android Studio 28
2.8 Protocolos de comunicación (Sockets) 29
2.9 Google Maps 33
Capítulo 3 34
Procedimiento 34
3.1 Metodología incremental 35
3.2 Curso Python 37
3.3 Creación de los grafos para las rutas de transporte 39
3.4 Queries para la obtención de rutas 44
3.5 Comunicación Cliente - Servidor 47
3.6 Funcionamiento e implementación de la activity en Android 54
Capítulo 4 66
Evaluación 66
4.1 Resultados de la implementación del plan de viaje 66
4.2 Impacto 78
Capítulo 5 79

Conclusiones y recomendaciones 79
5.1 Conclusión 79
5.2 Recomendaciones y trabajo a futuro 81
Bibliografía 82

Tabla de Figuras

Figura 1. Ejemplo de Digrafo (izquierda) y Grafo no orientado (derecha) 14
Figura 2. Grafo a resolver con Dijkstra 17
Figura 3. Dijkstra aplicando la primera iteración 17
Figura 4. Dijkstra aplicando segunda iteración 18
Figura 5. Dijkstra aplicando tercera iteración 19
Figura 6. Grafo resuelto por Dijkstra 20
Figura 7. Ejemplo de un esferoide oblato 22
Figura 8. Implementación de la fórmula de Haversine en Python 23
Figura 9. Diagrama TCP Cliente – Servidor (Stevens, 1998) 29
Figura 10. Three-way Handshake TCP (Stevens, 1998) 30
Figura 11. Curso Python en Sololearn 37
Figura 12. Curso Python por parte del CIMAT 38
Figura 13. Código Python para unir dos nodos de un grafo 40
Figura 14. Ejemplo sencillo de archivo de ruta 41
Figura 15. Rutas agregadas actualmente 41
Figura 16. Representación gráfica del grafo con las rutas anteriormente agregadas 43
Figura 17. Código Java para obtener la ruta óptima 46
Figura 18. Fragmento de código del servidor TCP en java 47
Figura 19. Fragmento de código cliente TCP en java (Android) 51
Figura 20. Método onMapClick 55
Figura 21. Fragmento del archivo JSON que genera un TreeMap 58
Figura 22. Ejemplo de búsqueda en el TreeMap 60
Figura 23. Ejemplo de llamada al método "dibujador" 60
Figura 24. Ejemplo de llenado de una lista de puntos para después dibujarla en el mapa 62
Figura 25. Ejemplo de método addAll de la clase Polyline 63
Figura 26. Ejemplo de uso de patrones con polylines 63

Figura 27. Arreglos que son utilizados para llenar el RecyclerView con instrucciones 64
Figura 28. Ejemplo de inicializar el RecyclerView 64
Figura 29. Llamada al clickListener del RecyclerView y resaltar rutas especificas 65
Figura 30. Ejemplo de enfoque a parada seleccionada y cambio de color de ruta 65
Figura 31. Explicación de la comunicación durante la primera fase de pruebas ... 66
Figura 32. Pantalla de inicial de la aplicación SIIBUS 67
Figura 33. Pantalla de la aplicación con la lista de ciudades disponibles 68
Figura 34. Pantalla de la aplicación con las rutas de la ciudad de Guanajuato 68
Figura 35. Primeros resultados de la aplicación 69
Figura 36. Funcionamiento de la comunicación para la obtención del JSON 70
Figura 37. Ejemplo de ruta óptima con el TreeMap ya implementado 71
Figura 38. Funcionamiento de la automatización de actualización del grafo 72
Figura 39. Pantalla de la aplicación con instrucciones en pantalla 73
Figura 40. Ejemplo de la búsqueda de ruta óptima 74
Figura 41. Ejemplo del funcionamiento del RecyclerView 75
Figura 42. Ejemplo de ruta óptima compleja 76

Capítulo 1

Introducción.

La ciudad de Guanajuato se caracteriza por sus calles angostas y tener muchos túneles prácticamente por todo el centro histórico, además de ser uno de los atractivos turísticos más grandes del bajío por los museos y áreas comerciales. Todo esto provoca un congestionamiento vehicular, sobre todo en las horas pico. Y por consecuencia de todo lo anterior, el transporte público muchas veces se retrasa y satura en su recorrido, eso hace que existan retrasos muy largos para los siguientes vehículos y además de tiempos largos en los traslados para llegar a los destinos.

Conscientes del avance de la tecnología, el CIMAT ha desarrollado un proyecto para el ***Monitoreo de Transporte Urbano (MTU)*** que es capaz, entre otras funciones, de ubicar las unidades de transporte público de una ciudad en tiempo real y enviar notificaciones a solicitud de los usuarios (en tiempo real) sobre la llegada de la unidad a una parada específica. Para incrementar las capacidades de la aplicación móvil y dar un mejor servicio a los usuarios, en este trabajo se presenta el desarrollo de una herramienta que encuentra la ruta óptima desde una ubicación específica del usuario (parada de autobús o de la ubicación actual) hacia un destino final. Esto permitirá a los usuarios trasladarse de una manera fácil y sencilla (con instrucciones claras de bajadas en paradas correspondientes), ahorrándole tiempo de viaje dentro de la ciudad de Guanajuato.

Este proyecto ayudará tanto a los habitantes como a los turistas de la ciudad a llegar a sus destinos, teniendo o no conocimiento de la ciudad con tan solo tener un dispositivo móvil conectado a una red de datos o una red de WIFI.

Existen aplicaciones que ayudan a encontrar la ruta óptima pero no existen aún para la ciudad de Guanajuato entonces, ¿Será posible igualar a estas aplicaciones que buscan la ruta óptima en otras ciudades de México?

En el documento se presenta una visión general sobre los temas relevantes que se necesitan para completar el proyecto, por ejemplo, temas como el algoritmo de Dijkstra, fórmula de Haversine, Python, Java, entre otros temas de vital importancia.

1.1 Identificación

El ahorro de tiempo siempre ha sido importante en la sociedad, y más en un país como México donde la gran mayoría de las personas viven su día a día con una gran intensidad. Día con día, los ciudadanos buscan realizar las tareas cotidianas de una manera rápida y eficaz, sobre todo en lo referente al tiempo de traslado (casa, oficina, etc.) mediante el transporte público. La ciudad de Guanajuato se caracteriza principalmente por tener muchos callejones de difícil acceso y tránsito, por lo que sería deseable que los habitantes hicieran uso de los servicios de transporte público para moverse a través de la ciudad en lugar de usar sus propios vehículos y así, bajar los niveles de congestionamiento y contaminación de la ciudad.

Además, la ciudad de Guanajuato, al ser un lugar turístico, llega a complicar en gran medida el ahorro del tiempo, debido a que muchas personas se conglomeran en el centro de la ciudad, o puntos de interés, como museos o lugares históricos que hay por toda la ciudad y entorpece el tráfico vehicular, por ende, el tráfico de las unidades de transporte público. Lo anterior provoca que algunas unidades de transporte lleguen en intervalos de tiempo muy largos y después la siguiente unidad de transporte se tarde menos en llegar a la parada y así perdiendo la planificación que tenía el sistema de transporte y por lo tanto llega a perjudicar a los usuarios del sistema de transporte.

El tráfico y retraso de las unidades de transporte público como tal no se puede evitar al cien por ciento, pero se busca reducirlos mediante el uso del transporte público para ayudar a los habitantes, estudiantes y turistas de la ciudad de Guanajuato a que midan mejor el tiempo de moverse de sus trabajos, casas, escuelas, etc. (la

aplicación ya cuenta con un sistema para determinar el tiempo de llegada del usuario desde una parada inicial a su destino).

El problema de elegir la ruta más corta que se debe tomar de punto A hacia el punto B, afecta principalmente a los “nuevos” usuarios, es decir, a los usuarios que no tienen conocimiento de las rutas de la ciudad de Guanajuato, en especial a los turistas que se puedan llegar a aventurar a visitar varios puntos de la ciudad y para eso tengan que hacer uso del transporte público.

De esta manera, se busca tener una aplicación que indique al usuario que ruta tomar, para que sea la ***más corta y económica posible***, ahorrando así tiempo y dinero.

1.2 Justificación

Guanajuato es una ciudad grande con una gran población de 194.500 (INEGI, 2020), donde la mayor parte de las personas hacen uso del transporte público (el 43.4% de la población en el estado de Guanajuato vive en situación de pobreza según él (Consejo Nacional de Evaluación de la Política de Desarrollo Social, 2020)). Cabe mencionar, que adicionalmente, la Ciudad de Guanajuato cuenta con una gran comunidad estudiantil foránea, saturando aún más el servicio de transporte público.

Como se mencionó en el apartado de identificación, muchas personas de Guanajuato capital tienen sus viviendas en callejones, por lo que no es posible el paso de vehículos, por lo tanto, muchas de estas personas no son propietarios de algún vehículo, y hacen uso del transporte público, entonces este proyecto ayudará a muchas personas, independientemente que formen parte de este grupo o no.

Para solucionar esta problemática, se ofrecerá dentro de la aplicación móvil la opción de ***plan de viaje***, cuya finalidad es ofrecer a los usuarios el cálculo de la ruta óptima de su trayecto completo. El usuario podrá ingresar su origen y destino a la aplicación móvil, y ésta con el uso de una red Wifi o red de datos, enviará la petición

a un servidor cuya respuesta es la ruta más corta calculada. Se espera que esto ayude a miles de personas, ya sean tanto habitantes de la ciudad, así como a turistas que deseen transportarse en la ciudad.

1.3 Alcance

En este proyecto se desarrollará un subsistema añadido a la aplicación SIIBUS denominado "***Plan de Viaje***", que calcula la ruta óptima a partir de un origen y un destino proporcionado por el usuario. Es decir, el usuario ingresará el lugar donde comenzará su recorrido y a dónde quiere llegar, teniendo estos datos la aplicación le debe regresar una respuesta, la cual será desplegada en un mapa para la comodidad del usuario y además una serie de instrucciones sobre cómo llegar al destino a través de las rutas de transporte urbano, aunque tambien existe la posibilidad de que en algunos tramos del trayecto el usuario tenga que trasladarse caminando, en dado caso la aplicación le deberá especificar a donde debe caminar. Aunque se debe de tener en cuenta que aquí existe una gran limitación, porque en caso de que se presente un evento social (festivales, conciertos, marchas, desfiles, etc.), el plan de viaje calculado podría no coincidir con el cambio temporal de ruta.

Capítulo 2

Fundamento teórico

2.1 Teoría de Grafos

El plan de viaje del usuario considera las paradas de las rutas de una ciudad como un problema computacional representado como un conjunto de nodos conectados, por tal motivo se puede resolver mediante el uso de la teoría de grafos.

Siguiendo a (Meza H. & Ortega F., 2004), un grafo G es una terna (V, E, φ) donde V es un conjunto finito de elementos llamados vértices o nodos del grafo, E es un conjunto finito de elementos llamados lados y φ es una función que asigna a cada elemento E un par de elementos V. Si todos estos elementos no están ordenados, se puede decir que el grafo es no orientado o no dirigido y los sus lados se llaman aristas. Cuando los pares son ordenados, se denotan (a, b) y se les llama arcos. En este caso se puede decir que el grafo es orientado o dirigido y también se le puede llamar ***Dígrafo***. Si G tiene n vértices decimos que G es de orden n.

A cada grafo se le puede dar una representación gráfica o también asociar una figura, la cual ayuda a estudiar las propiedades que tiene. Esta figura es un dibujo, en donde los nodos aparecen como círculos o puntos en el plano y cada lado o arista se representa mediante una línea que une a dos puntos (nodos) que corresponden a sus extremos. Cuando es un Dígrafo, los arcos se presentan con una flecha que va del nodo inicial al nodo final.

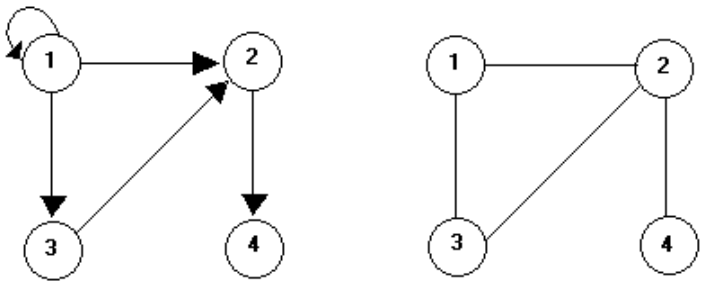

Figura 1. Ejemplo de Dígrafo (izquierda) y Grafo no orientado (derecha)

2.2 Algoritmo de Dijkstra

El algoritmo de Dijkstra es de tipo voraz (Greedy) que determina la ruta más corta desde un nodo origen hacia los demás nodos; para esto, se requiere como entrada un grafo cuyas aristas representan los pesos para viajar entre los nodos que conectan.

Según (Nolasco & Atoche, 2014) el algoritmo consiste en ir encontrando los caminos más cortos mediante la activación de todos los puntos que conforman el grafo empezando en un nodo inicial “i” hasta un nodo final “j”. Para la realización de este trabajo se considera que cualquier paradero puede convertirse en nodo de origen o nodo final según la necesidad del transeúnte.

Algunas consideraciones:

- Si los pesos de las aristas son de valor 1, entonces bastará con usar el algoritmo de BFS (BFS es un algoritmo de búsqueda el cual recorre los nodos de un grafo, comenzando en la raíz, para luego explorar los vecinos de ese nodo. Después para cada uno de los vecinos se exploran sus respectivos vecinos y así sucesivamente, hasta que se recorra todo el grafo)

- Si los pesos de las aristas son negativos no se puede usar el algoritmo de Dijkstra, para pesos negativos tenemos otro algoritmo llamado Algoritmo de Bellmand-Ford.

Algoritmos Greedy o algoritmos voraces, se caracterizan porque se utilizan generalmente para resolver problemas de optimización (obtener el máximo o el mínimo), tomando decisiones en función de la información que está disponible en cada momento. Una vez tomada la decisión, ésta no vuelve a replantearse en el futuro. Suelen ser rápidos y fáciles de implementar, pero no siempre garantizan alcanzar la solución óptima.

Los pasos que se deben de seguir para aplicar el algoritmo Dijkstra son los siguientes:

1. Se elige un nodo de inicio al cual se le marcara un peso de la siguiente forma: [X, Y](N)

   Donde ‘X’ equivale a el valor del recorrido actual de las aristas, ‘Y’ equivale a el nodo anterior o de origen y ‘N’ al número de iteración u operación actual (cuando es la primera iteración, es decir, del origen, la iteración es igual a cero).

2. A los nodos adyacentes del nodo seleccionado como nodo de inicio, se deben asignar un peso de igual forma al punto anterior, ([X,Y](N)).

3. De los nodos con los pesos calculados se toma el nodo con menor valor en X y este será el siguiente a visitar.

4. Los pasos 2 y 3 deben repetirse teniendo en cuenta que, si al intentar calcular los pesos para los nodos adyacentes a un nodo que está siendo visitado, uno de estos ya tiene un peso asignado, deben calcularse los demás pesos cuantas veces sean necesario, y siempre se tomará el peso mínimo calculado.

5. Los nodos pueden ser visitados una sola vez.

Ejemplo del algoritmo:

Problema: se quiere llegar del nodo inicial ‘A’ al nodo final ‘E’ utilizando el algoritmo Dijkstra para encontrar la ruta óptima. El grafo se muestra a continuación:

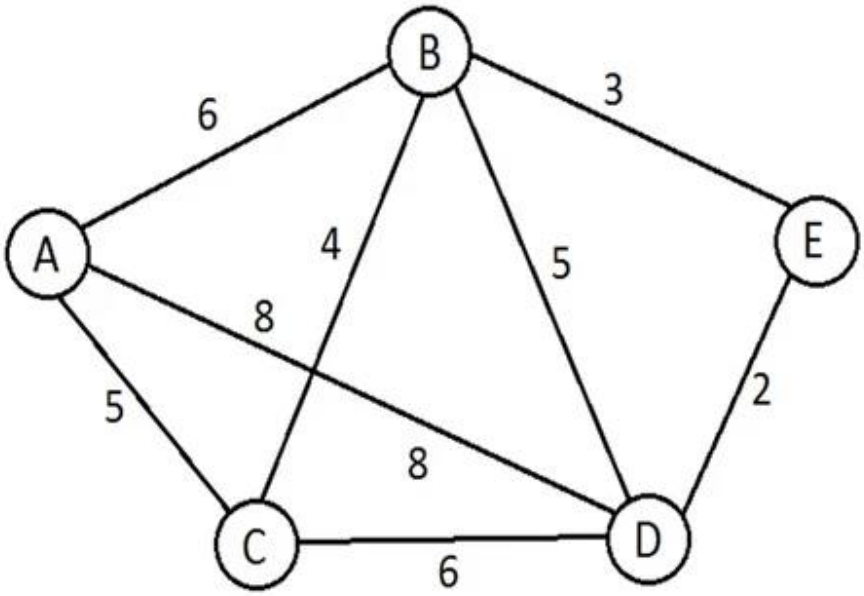

Figura 2. Grafo a resolver con Dijkstra

Se quiere llegar de ‘A’ a ‘E’, por lo tanto, nuestro nodo inicial es ‘A’: [0, -] (0).

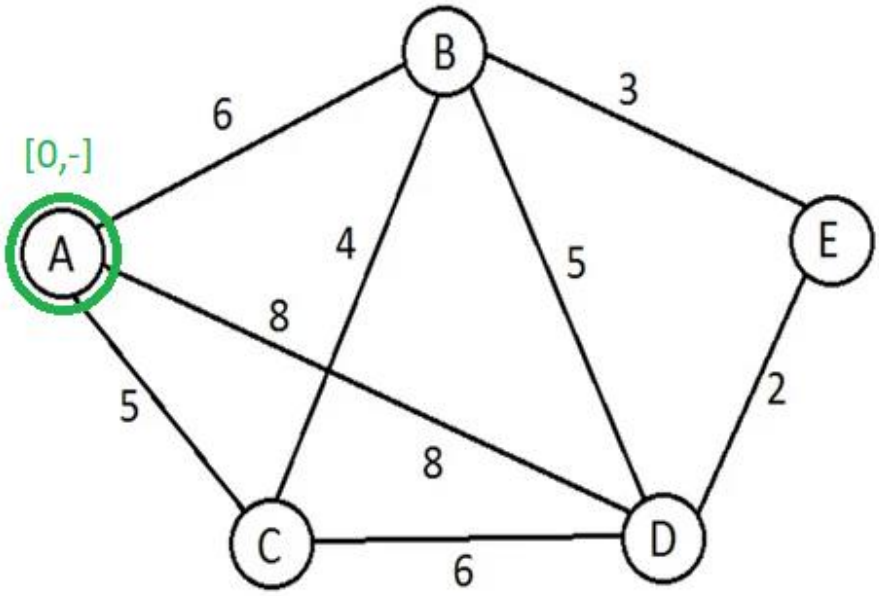

Figura 3. Dijkstra aplicando la primera iteración

El nodo ‘A’ tiene tres adyacencias, ‘B’, ‘C’ y ‘D’.

Para el nodo B se calcula: [6, A] (1), para el nodo C se calcula: [5, A] (1) y para el nodo D se calcula: [8, A] (1):

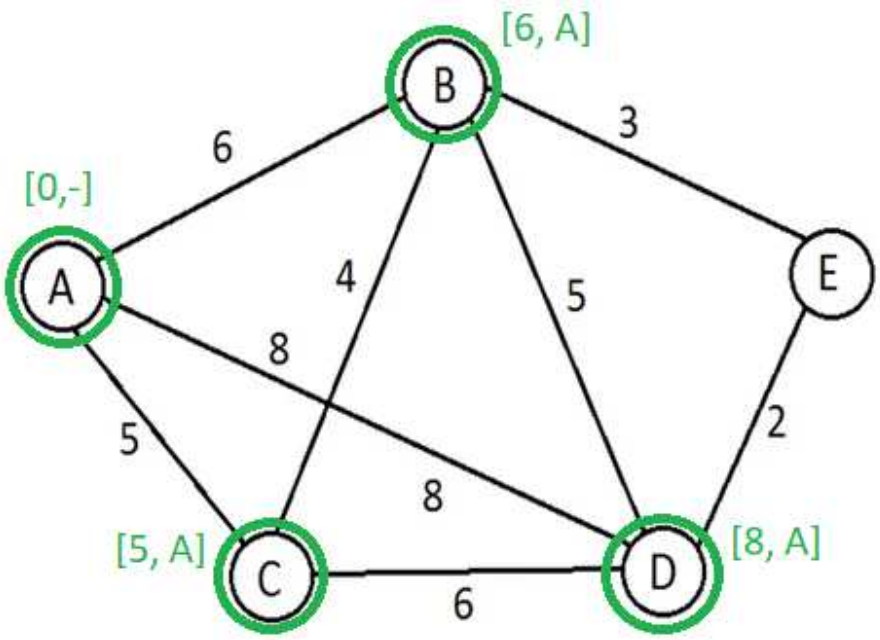

Figura 4. Dijkstra aplicando segunda iteración

El siguiente paso es tomar el nodo con la distancia más corta la cual es la del nodo C con [5, A] (1), como podemos ver, este nodo C tiene adyacencia con A, B y D, como el nodo A es el nodo de origen este nodo se convierte en un nodo permanente y ya no se vuelve a calcular, porque como lo dice la regla número 5, solo se puede pasar por un nodo solo una vez. Entonces, calculamos los nodos B y D: para el nodo B se calcula: [9, C] (2) y para el nodo D se calcula: [11, C] (2), pero en este nodo D anteriormente teníamos [8, A] (1), el cual resulta ser menor que estos dos anteriores, así que esta es la ruta más corta, entonces, se convierte en un nodo permanente:

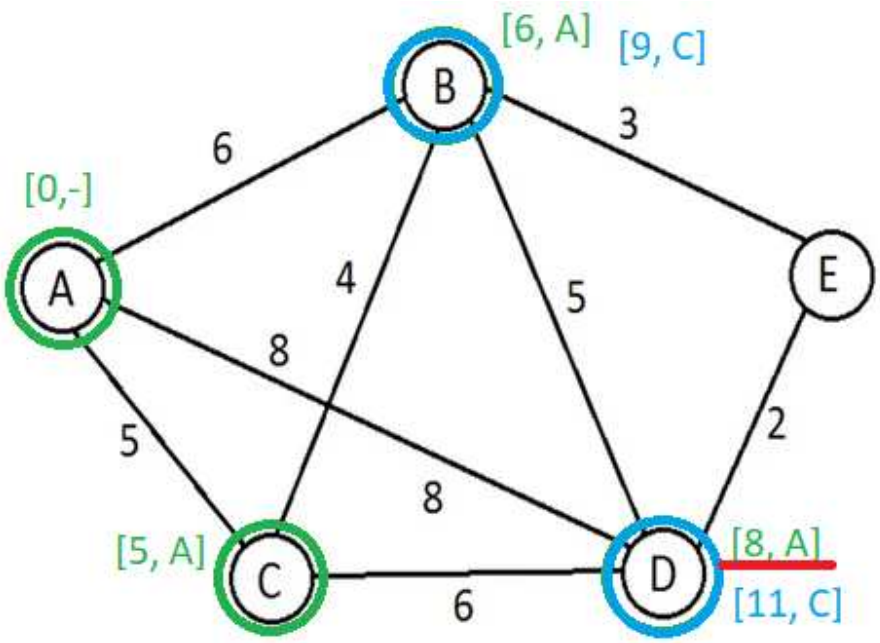

Figura 5. Dijkstra aplicando tercera iteración

Ahora calculamos la ruta más corta con base al nodo D, el nodo D tiene adyacencia con B y E, entonces se calcula para el nodo B: [13, D] (3) y para el nodo E se calcula: [10, D] (3), así tenemos que la distancia más corta es la del nodo E, y además tenemos que considerar que este es el nodo al que queríamos llegar, por ende, este es el resultado final.

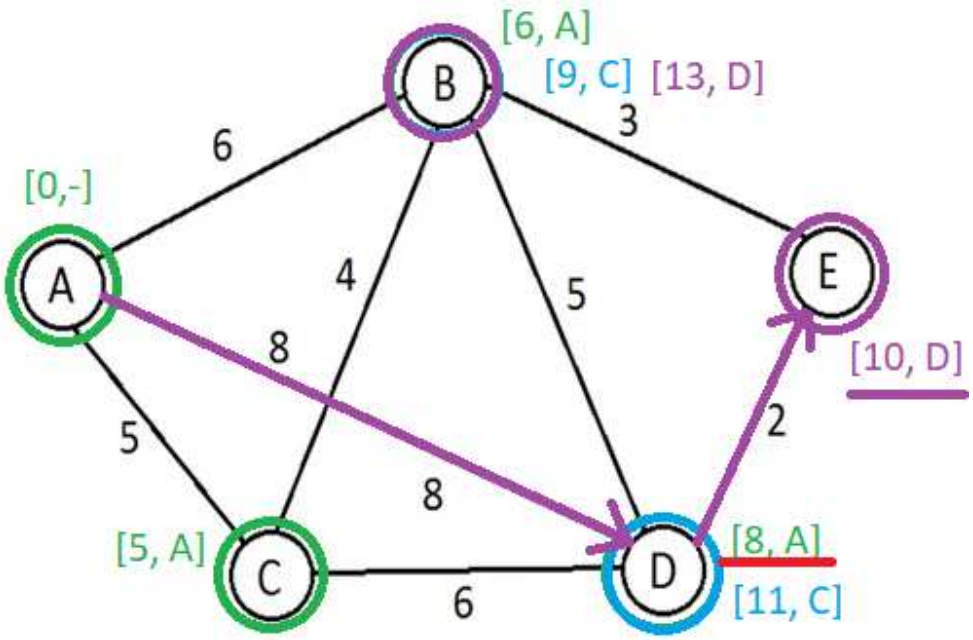

Figura 6. Grafo resuelto por Dijkstra

La ruta más corta utilizando este algoritmo es de A -> D -> E con un valor final de 10.

2.3 Fórmula de Haversine

La fórmula de Haversine (o semiverseno en español) siguiendo a (Diaz, 2012) es una formula general de la trigonometría esférica muy utilizada en la navegación. Calcula la distancia entre dos puntos en una esfera usando sus coordenadas (latitud y longitud) a través de la superficie. El semiverseno puede ser expresado en una función trigonométrica como:

$$haversine(\theta) = sin^2\left(\frac{\theta}{2}\right)$$

El semiverseno del ángulo central (el cual se representa con d/r) se calcula con la siguiente fórmula:

$$\left(\frac{d}{r}\right) = haversine(\Phi_2 - \Phi_1) + cos(\Phi_1)cos(\Phi_2)haversine(\lambda_2 - \lambda_1)$$

Donde r es el radio de la tierra (6371 km), d es la distancia entre los dos puntos Φ_1, Φ_2 es la latitud de los dos puntos y λ_1, λ_2 son la longitud de los puntos respectivamente.

Para despejar la distancia (d) se aplica la función de semiverseno inverso o mediante el uso de la función de arcoseno:

$$d = rhaversine^{-1}(h) = 2rarcsin\left(\sqrt{h}\right)$$

Donde h es semiverseno (d/r), es decir:

$$d = 2rarcsin\left(\sqrt{sin^2\left(\frac{\Phi_2 - \Phi_1}{2}\right) + cos(\Phi_1)cos(\Phi_2)sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$$

Ejemplo de uso de la fórmula de Haversine:

La distancia entre El Big Ben en Londres (51.5007° N, 0.1246° W) y La Estatua de la Libertad en Nueva York es de 5574.8 Km. Cabe mencionar que esta no es la medida exacta porque la fórmula asume que la Tierra es una esfera perfecta,

cuando no lo es, es un esferoide oblato (es un elipsoide rotacionalmente simétrico en el cual el eje polar es más pequeño que el diámetro de su círculo ecuatorial. Dícese, también, aplanado o achatado por los polos)

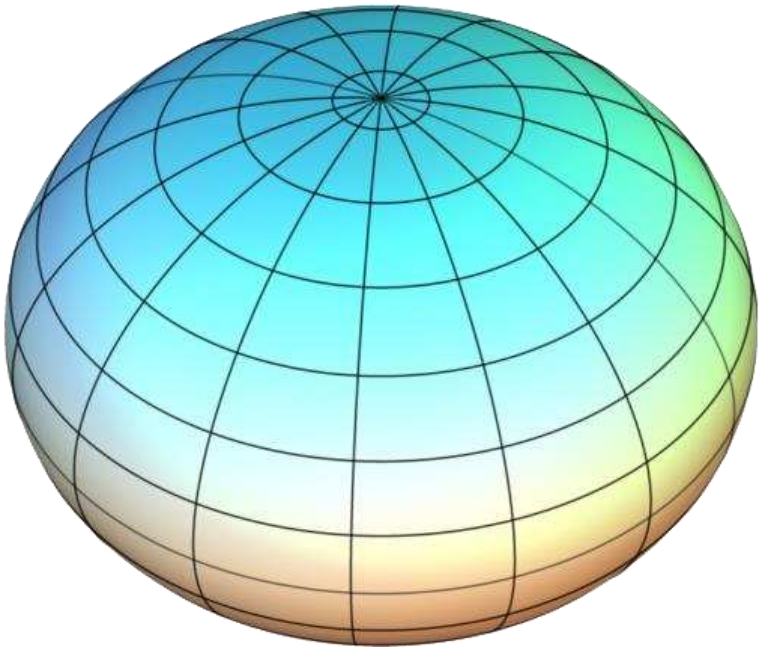

Figura 7. Ejemplo de un esferoide oblato

A continuación, se presenta la implementación en código Python con el ejemplo anterior calculando la distancia entre el Big Ben y la Estatua de la Liberta con la fórmula de Haversine:

```python
# Programa en Python para la formula de Haversine
def haversine(lat1, lon1, lat2, lon2):

    # distancia entre latitudes
    # y longitudes
    dLat = (lat2 - lat1) * math.pi / 180.0
    dLon = (lon2 - lon1) * math.pi / 180.0

    # conversion a radianes
    lat1 = (lat1) * math.pi / 180.0
    lat2 = (lat2) * math.pi / 180.0

    # aplicacion de formula
    a = (pow(math.sin(dLat / 2), 2) +
         pow(math.sin(dLon / 2), 2) *
             math.cos(lat1) * math.cos(lat2));
    rad = 6371
    c = 2 * math.asin(math.sqrt(a))
    return rad * c

# Driver code
if __name__ == "__main__":
# latitud y longitud del Big Ben
    lat1 = 51.5007
    lon1 = 0.1246
# latitud y longitud de la Estatua de la Libertad
    lat2 = 40.6892
    lon2 = 74.0445

    print(haversine(lat1, lon1,lat2, lon2), "K.M.")
```

Figura 8. Implementación de la fórmula de Haversine en Python

Output:

5574.840456848555 K.M.

2.4 Python

Python es un lenguaje de programación de alto nivel que se utiliza para desarrollar aplicaciones de todo tipo. A diferencia de otros lenguajes como Java o .NET, se trata de un lenguaje interpretado, es decir, que no es necesario compilarlo para ejecutar las aplicaciones escritas en Python, sino que se ejecutan directamente por el ordenador utilizando un programa denominado interpretador, por lo que no es necesario “traducirlo” a lenguaje máquina.

Python es relativamente simple, por lo que es fácil de aprender, ya que requiere una sintaxis única que se centra en la legibilidad. Los desarrolladores pueden leer y traducir el código Python mucho más fácilmente que otros lenguajes.

Python fue creado por el informático Guido van Rossum, el cual había trabajado anteriormente con un lenguaje llamado ABC el cual no tuvo éxito, sin embargo, a principios de los años 90, decidió hacer uno nuevo basándose en este lenguaje, así es como nació Python.

Su filosofía fue la misma desde el primer momento: crear un lenguaje de programación que fuera muy fácil de aprender, escribir y entender, sin que esto detuviera su potencial para crear cualquier tipo de aplicación.

Python es un lenguaje de programación de propósito general, que es otra forma de decir que puede ser usado para casi todo. Lo más importante es que se trata de un lenguaje interpretado, lo que significa que el código escrito no se traduce realmente a un formato legible por el ordenador en tiempo de ejecución.

Este tipo de lenguaje también se conoce como «lenguaje de scripting» porque inicialmente fue pensado para ser usado en proyectos sencillos.

El concepto de «lenguaje de scripting» ha cambiado considerablemente desde su creación, porque ahora se utiliza Python para programar grandes aplicaciones de estilo comercial, en lugar de sólo las simples aplicaciones comunes.

2.5 NetworkX

Según (L. Platt, 2019) NetworkX es una biblioteca de Python para el estudio de grafos y análisis de redes. NetworkX es un software libre publicado bajo la licencia BSD-new.

Con NetworkX, puede almacenar redes en formatos de datos estandarizados y no estandarizados, generar una variedad de redes aleatorias y redes clásicas, analizar estructuras de red, establecer modelos de red, diseñar nuevos algoritmos de red y realizar dibujos de redes, etc.

Los siguientes son los tipos de grafo básicos que existen y están disponibles como clases de Python:

- **Graph:** esta clase implementa un grafo no dirigido. Ignora múltiples aristas entre dos nodos. Permite enlazar una arista en un mismo nodo.
- **DiGraph:** esta clase implementa un grafo dirigido, es decir, grafos con aristas dirigidas. Permite hacer operaciones comunes a grafos dirigidos.
- **MultiGraph:** se refiere a varios grafos no dirigidos, es decir, el número de aristas entre dos nodos es más de uno y el vértice puede relacionarse consigo mismo a través de la misma arista.
- **MultiDiGraph:** es una versión dirigida del MultiGraph.

2.6 Java

Según (Robledano, 2019), Java es un lenguaje de programación de propósito general orientado a objetos, que fue diseñado específicamente para tener tan pocas dependencias de implementación como fuera posible permitiendo a desarrolladores escribir un programa y ejecutarlo en cualquier tipo de dispositivo sin tener que compilarlo una y otra vez.

Java fue creado por Sun MicroSystems en el año 1991 como una herramienta de programación para ser usada en un proyecto de set-top-box en una pequeña operación denominada *The Green Project*. Su equipo de desarrollo trabajó por más de 18 meses hasta lograr liberar su primera versión estable.

Como curiosidad, decir que no está claro el origen del nombre, aunque se cree que podría tratarse de las iniciales de sus diseñadores: James Gosling, Arthur Van Hoff, y Andy Bechtolsheim. La hipótesis que más fuerza tiene es la de que Java debe su nombre a un tipo de café disponible en la cafetería cercana; de ahí que el icono de Java sea una taza de café caliente.

Como un lenguaje de programación orientado a objetos (POO) el programador puede generar fragmentos de código autónomo, que puedan interactuar con otros objetos para resolver un problema ofreciendo soporte para diferentes tecnologías. De hecho, también es común referirse a Java como un conjunto de tecnologías en referencia a los diferentes productos y versiones que componen su familia.

Sus principios básicos son:

- **Simple**. Una de las ventajas de Java reside en su sencillez con una moderada curva de aprendizaje. Esto hace que sea el lenguaje más usado en escuelas y universidades para mostrar los fundamentos de la programación.
- **Multihilo**. Considerando el entorno multithread (multihilo), cada thread (hilo, flujo de control del programa) representa un proceso individual ejecutándose en un sistema. Cada hilo controla un único aspecto dentro de un programa,

como puede ser supervisar la entrada en un determinado periférico o controlar toda la entrada/salida del disco. Todos los hilos comparten los mismos recursos, al contrario que los procesos, en donde cada uno tiene su propia copia de código y datos (separados unos de otros).

- **Seguro**. Java es un lenguaje de programación seguro y estable. Pensado para poder operar en multitud de entornos. Desde el sector más lúdico a aplicaciones empresariales.

- **Multiplataforma**. Podemos desarrollar nuestro código una única vez y ejecutarlo en cualquier plataforma. Lo que facilita el poder portar nuestro proyecto a diferentes sistemas operativos.

2.7 Android Studio

Según (Google Developers, 2022), Android Studio es un software que ofrece las herramientas y servicios con los que los creadores pueden confeccionar aplicaciones para el sistema operativo Android, sin tener que recurrir a nada más.

Como ocurre con la mayoría de entornos de desarrollo modernos, nos ofrecen las herramientas necesarias en la generación del código, lo que se denomina la lógica de la aplicación. Pero también los mecanismos con los que diseñar la interfaz de usuario que lucirá el desarrollo final.

El entorno permite y guía al creador para que pueda desde realizar sus primeras pruebas de código hasta la publicación del desarrollo final en Play Store, la tienda de aplicaciones de Google. Para realizar el proceso no faltan elementos necesarios como el compilador, el analizador de archivos APK o un emulador.

Al poder desarrollar aplicaciones para el sistema operativo móvil de Google, por extensión también las estamos confeccionando para ChromeOS, por lo que se tiene en cuenta esta opción. También los diferentes destinos que puede tener, más allá del teléfono: relojes, tabletas, autos, etc.

2.8 Protocolos de comunicación (Sockets)

Siguiendo a (Stevens, 1998) los sockets son una forma de comunicación entre procesos que son parte integral de los programas de red, para lograr comunicación en una estructura cliente – servidor se debe de proporcionar un punto de comunicación para poder enviar y recibir información. Los sockets tienen un ciclo de vida, el cual depende si son sockets de un servidor esperando a los sockets clientes o si son sockets de un cliente esperando a un servidor para establecer comunicación.

Los sockets TCP (Transmission Control Protocol) tienen funciones para poder completar una estructura cliente – servidor TCP. En el siguiente diagrama se explica cómo está conformada la estructura cliente-servidor TCP:

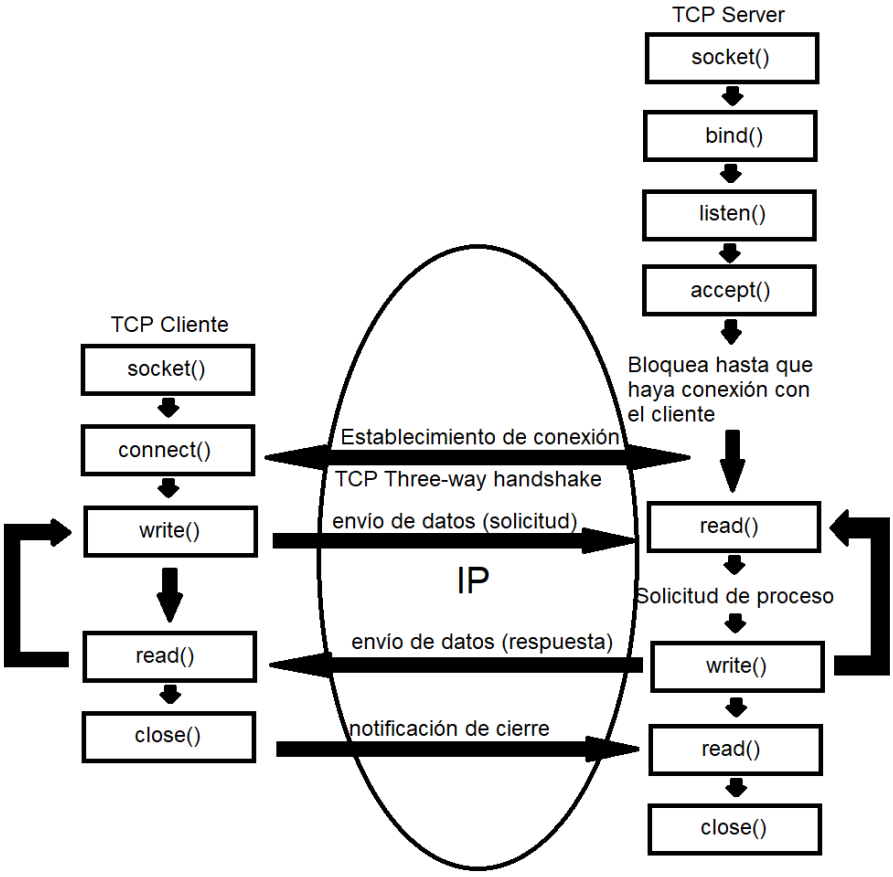

Figura 9. Diagrama TCP Cliente – Servidor (Stevens, 1998)

En la parte del servidor hay una función llamada “bind”, la cual se encarga de asociar el socket a la dirección local especificada para que el socket se asigne al puerto especificado de la misma.

Después de la función bind, se encuentra la función “listen”, que solo es llamada por el servidor y hace que un elemento orientado a socket a la conexión escuche los intentos de conexión entrantes

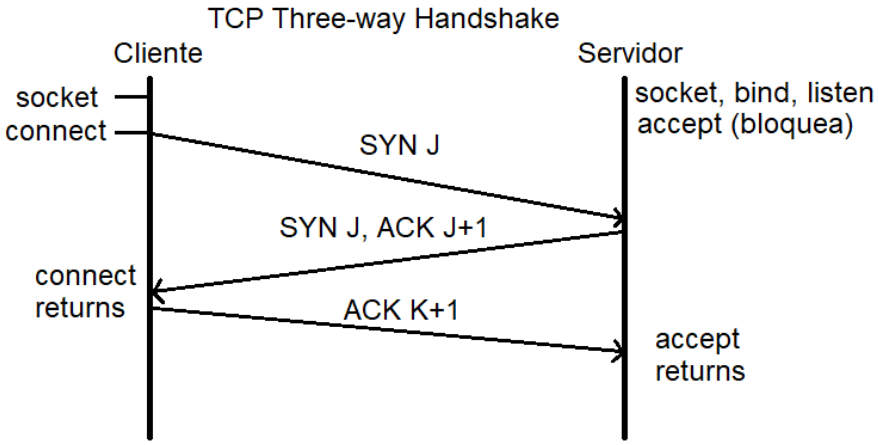

Figura 10. Three-way Handshake TCP (Stevens, 1998)

Como se puede observar en la imagen anterior, el cliente TCP tiene una función llamada “connect”, la cual establece la conexión con un servidor TCP. Cuando esta función es llamada, ocurre un Three-way handshake, el cual consta de 3 pasos.

Paso 1: la conexión entre el cliente y el servidor es establecida, para esto el servidor debe tener los puertos abiertos para poder aceptar e iniciar nuevas conexiones con clientes. El cliente envía un paquete de datos SYN (Synchronize Sequence Number) sobre una dirección IP al servidor, que puede estar en la misma red o en una diferente.

Este paquete SYN es una secuencia aleatoria de números que el cliente usa para establecer una conexión (por ejemplo, X). El objetivo de este paquete en preguntarle al servidor si está abierto para nuevas conexiones.

Paso 2: el servidor recibe el paquete SYN del cliente, cuando se recibe el servidor regresa un paquete ACK (Acknowledgement Sequence Number) o paquete ACK/SYN, este paquete incluye dos secuencias numéricas.

La primera secuencia es el ACK, el cual es establecido por el servidor tomando la secuencia que recibió del cliente, pero agregando uno más (ejemplo, X+1).

La segunda es la secuencia SYN enviada por el servidor, la cual es otra secuencia aleatoria de números (por ejemplo, Y).

Esta secuencia indica que el servidor ha reconocido el paquete del cliente y le está enviando su propio paquete para ser reconocido.

Paso 3: el cliente recibe el paquete ACK/SYN del servidor y responde con su propio paquete ACK, y como lo hizo el servidor, ahora es turno del cliente de responder con su ACK, pero incrementado en uno (ejemplo, Y+1) reenviándolo al servidor.

Al completarse este proceso, la conexión del cliente y servidor estará creada y ahora se pueden comunicar.

Para la comunicación, tanto el cliente como el servidor pueden usar las funciones “write” y “read” las cuales sirven para enviar y recibir información correspondientemente. Y cualquiera de las dos partes puede iniciar la conexión, por ejemplo, el cliente puede usar la función write y por lo tanto el servidor debe de estar en listo con la función read para poder percatar la información del cliente, y justo despues de esa función el servidor debe de enviar una respuesta, para lo que necesita enviar la función write con lo que sea que se requiera enviar. Del mismo modo, el cliente ya debe estar esperando con la función read para recibir dicha respuesta del servidor.

Las funciones read y write se pueden repetir las veces que sean necesarias, tanto en el lado del cliente y del servidor.

Para finalizar la conexión, existe la función close, esta función la usa el cliente para avisar al servidor de que ya obtuvo la información que necesitaba y por lo tanto ya no es necesario mantener viva la conexión entre ambas partes, esta función la tiene que escuchar el servidor, por ende, hace uso de la función read para escuchar la función close, y cuando lo hace, el servidor también llama a la función close, para así cerrar la conexión del socket.

2.9 Google Maps

Siguiendo a (Insaurralde, 2022) Google Maps es un servicio de aplicaciones de ubicación con el que se pueden obtener imágenes de casi cualquier parte del mundo, así como visualizar fotografías y obtener indicaciones para llegar. El mismo muestra una perspectiva satelital y aérea de los diferentes países o ciudades a nivel mundial, permitiendo incluso mirar casas, edificios, etc., de diferentes regiones.

Esta aplicación sirve para obtener la ubicación exacta de cualquier punto específico en el mapa, permitiendo, además, la posibilidad de conocer las coordenadas y posibles rutas de acceso al mismo. Puesto que, ofrece diferentes funciones de geolocalización con las que es muy simple saber dónde queda un local comercial, sitio turístico, establecimiento público, etc.

Google Maps funciona por medio de acceso a Internet, usando la tecnología de ubicación satelital y aérea del ordenador o el celular. La misma permite mostrar imágenes reales de diversos puntos geográficos en casi cualquier parte del mundo. Además, esta aplicación cuenta con diferentes funciones básicas y avanzadas, tales como: coordenadas, información del sitio, navegación, etc.

Google Maps cuenta con un SDK que se puede integrar a los proyectos de Android que necesiten hacer uso de esta herramienta. Según (Google Developers, 2022), para hacer uso de Google Maps Plataform, se debe tener un proyecto creado en dicha plataforma para administrar credencias, facturación, servicios, etc. La facturación es obligatoria, pero existe un periodo gratuito y solo se cobra si este periodo se excede. Para así tener credenciales de la API y poder implementar al proyecto.

Capítulo 3

Procedimiento

Este proyecto se realizó en el Centro de Investigación en Matemáticas A. C. (CIMAT) de la ciudad de Guanajuato, ciudad en la cual se pretende implementar inicialmente el proyecto, y en un futuro llevarlo a más ciudades de México o del mundo.

Este proyecto se llevó a cabo por el equipo de investigación del Dr. Rogelio Hasimoto Beltrán, quien funge como asesor externo e investigador titular en el CIMAT.

El CIMAT es un centro de investigación el cual se ubica en la colonia Valenciana en la ciudad de Guanajuato, donde se lleva a cabo el desarrollo de varios proyectos en las áreas de matemáticas, computación, estadística, etc., además de ofrecer postgrados a estudiantes.

El presente proyecto, cuenta con algunos años en desarrollo, pero en esta ocasión se le buscó añadir una nueva funcionalidad, la cual es la obtención de la ruta más corta o ruta óptima para la ciudad de Guanajuato.

Durante los primeros días se explicaron todos los proyectos desarrollados con anterioridad que han tenido que ver con este proyecto, y las nuevas adicciones que se querían implementar al mismo. También se habló de una manera muy general de la problemática que quería resolver, y como se pretendía resolver. Para así entender en que se iba a trabajar a lo largo de esta tesis y de esta manera dando inicio a las actividades del presente proyecto.

3.1 Metodología incremental

Para el desarrollo de este proyecto se eligió la metodología incremental, ya que este proyecto tiene un alcance muy grande que el periodo de residencias no es un periodo suficiente para poder cubrirlo en su totalidad, por ende, posteriormente se realizará una tesis para poder completar exitosamente el proyecto, continuando desde donde se dejó este trabajo. Además de tener ciertas funcionalidades en periodos cortos de tiempo y tener resultados para el reporte de residencias.

La metodología incremental se describe a continuación.

Según Pérez, A. (2016), el modelo de gestión incremental no es un modelo necesariamente rígido, es decir, que puede adaptarse a las características de cualquier tipo de proyecto, existen al menos 7 fases que debemos tener en cuenta a la hora de implementarlo:

1. **Requerimientos:** son los objetivos generales y específicos que persigue el proyecto.

2. **Definición de las tareas y las iteraciones:** teniendo en cuenta lo que se busca, el siguiente paso es hacer una lista de tareas y agruparlas en las iteraciones que tendrá el proyecto. Esta agrupación no puede ser aleatoria. Cada una debe perseguir objetivos específicos que la definan como tal.

3. **Diseño de los incrementos:** establecidas las iteraciones, es preciso definir cuál será la evolución del producto en cada una de ellas. Cada iteración debe superar a la que le ha precedido. Esto es lo que se denomina incremento.

4. **Desarrollo del incremento:** posteriormente se realizan las tareas previstas y se desarrollan los incrementos establecidos en la etapa anterior.

5. **Validación de incrementos:** al término de cada iteración, los responsables de la gestión del proyecto deben dar por buenos los incrementos que cada

una de ellas ha arrojado. Si no son los esperados o si ha habido algún retroceso, es necesario volver la vista atrás y buscar las causas de ello.

6. **Integración de incrementos:** una vez son validados, los incrementos dan forma a lo que se denomina línea incremental o evolución del proyecto en su conjunto. Cada incremento ha contribuido al resultado final.

7. **Entrega del producto:** cuando el producto en su conjunto ha sido validado y se confirma su correspondencia con los objetivos iniciales, se procede a su entrega final.

Como se explicó anteriormente, se eligió esta metodología porque es muy conveniente para este proyecto y se adapta muy bien a la forma en que se está desarrollando el proyecto, pues se van añadiendo mejoras al proyecto de manera gradual y en pequeñas partes las cuales las podemos tomar como las iteraciones con las que trabaja esta metodología.

Cabe mencionar que aproximadamente cada semana se realizaron reuniones con el encargado del proyecto para ver el avance del proyecto y decidir qué es lo que se iba a trabajar a continuación.

3.2 Curso Python

Para poder desarrollar correctamente el proyecto bajo las instrucciones del encargado del proyecto, se optó por tomar dos cursos de Python ya que se debía comprender a la perfección este lenguaje para desarrollar el sistema de ruta óptima.

El primer curso se tomó en la plataforma de Sololearn, porque es una plataforma muy interactiva y además muy intuitiva para el usuario. El curso en cuestión es Python for Beginners, porque de esta manera se puede introducir al investigador fácilmente al lenguaje Python.

Figura 11. Curso Python en Sololearn

Posteriormente, se proporcionó otro curso de Python para los investigadores a cargo del Dr. Hasimoto, el cual era ya un poco más avanzado a comparación del anterior curso. El cual nos sirvió para aprender funcionalidades más complejas del lenguaje.

Figura 12. Curso Python por parte del CIMAT

Estos cursos fueron muy importantes, porque fueron de utilidad para tener una mejor idea y percepción del lenguaje Python y poder desarrollar el grafo de las rutas fuertemente conexas que se agregaron al proyecto para posteriormente en la aplicación Java calcular la ruta óptima.

3.3 Creación de los grafos para las rutas de transporte

Para la creación de la red del transporte público se debe de contar con los siguientes requisitos:

- Python 2.7
- Biblioteca de Python NetworkX (2.8)
- Biblioteca de Python Matplotlib (3.5.2)

En caso de no contar con estas bibliotecas de Python, desde la consola de Linux (Ubuntu) basta con lanzar los comandos:

$ pip install networkx

Este comando instalará la última versión de NetworkX, pero para obtener los mismos resultados que se obtuvieron en este proyecto se recomienda que al final se especifique la versión, en este caso la versión 2.8

$ pip install networkx==2.8

Y para la biblioteca de matplotlib, basta con lanzar este comando desde la consola:

$ pip install matplotlib

En el caso de matplotlib se está usando la versión más reciente, la cual es 3.5.2.

Y el algoritmo funciona de la siguiente manera, dado un array de grafos (este array se crea leyendo unos archivos de texto donde se tienen las rutas y sus atributos, como nombre de la ruta, nombre de la parada, latitud y longitud de la parada. Donde cada ruta se convierte en un grafo y este se almacena en el array) se hace una unión de todos los grafos que se encuentren en el arreglo. A continuación, se presenta el código en lenguaje Python.

```python
def union_graphs(v_g):

    G = nx.DiGraph()
    nodos = []
    aristas = []
    for g in v_g:
        nodos = nodos + list(g.nodes(data = True))
        aristas = aristas + list(g.edges(data = True))
    print ("Tenemos -> " , len(aristas) , "aristas")

    G.add_nodes_from(nodos)

    for n1 , n2 , data in aristas:
        try :
            G.adj[n1][n2]['ruta'].append(data['ruta'])
            print ("agregando ruta")
        except KeyError:
            G.add_edge(n1 , n2 , distancia = data['distancia'] , ruta = [ data['ruta'] ])

    G = nx.convert_node_labels_to_integers(G , first_label =0)

    G1  = G.copy()

    edges = G.edges()
    R = 0
    while not nx.is_strongly_connected(G1): ### checando si existen caminos todos contra todos
        G1 = G.copy()
        R +=50
        print ("Probando si es fuertemente conexa con  R=" , R)
        nuevas= 0
        for n1  , data1 in G.nodes(data = True):
            for n2 ,data2 in G.nodes(data = True):
                d = haverseno(data1['position']  , data2['position'])
                if n1 != n2 and  d < R and not (n1,n2) in edges:
                    G1.add_edge(n1,n2 , distancia = d , color = 'red' , ruta = ["a_pie"])
                    nuevas +=1

    print ("se agregaron" , nuevas , "aristas ")

    return G1
```

Figura 13. Código Python para unir dos nodos de un grafo

La manera en que los une es, dados dos nodos n1 y n2, si n2 está a una distancia menor o igual a R, donde n1 es el inicio de R, se agrega una arista para unir dichos nodos. Y en caso de que no se encuentren dichos nodos, R va aumentar de manera gradual (50 metros en cada iteración), así hasta encontrar caminos de todos contra todos creando un grafo fuertemente conexo.

Ahora bien, los archivos de rutas están compuestos de la siguiente forma. Son archivos de texto (.txt) donde llevan por nombre el nombre de su ruta correspondiente, por ejemplo, tomemos la ruta alhondiga-valenciana.txt, este archivo es un ejemplo sencillo y está compuesto de la siguiente manera:

```
1  Ruta Alhondiga-Valenciana
2  Alhondiga       21.019516       -101.258756
3
4  Dos Rios        21.022931       -101.258722
5
6  San Javier      21.028004       -101.259179
7
8  Macro1  21.030062       -101.255947
9
10 Valenciana      21.042177       -101.258009
```

Figura 14. Ejemplo sencillo de archivo de ruta

Donde podemos ver que, en el primer renglón tiene el nombre de la ruta, seguido del nombre de la parada junto con sus coordenadas, están separadas por una tabulación. El formato correcto es: nombre de la parada \t latitud \t longitud.

Actualmente, solo se cuenta con cinco rutas, esto se debe a que el proyecto aún está en fase de pruebas, pero al final del proyecto se planea que se encuentren todas las rutas de la ciudad de Guanajuato, o en su defecto, la gran mayoría de estas.

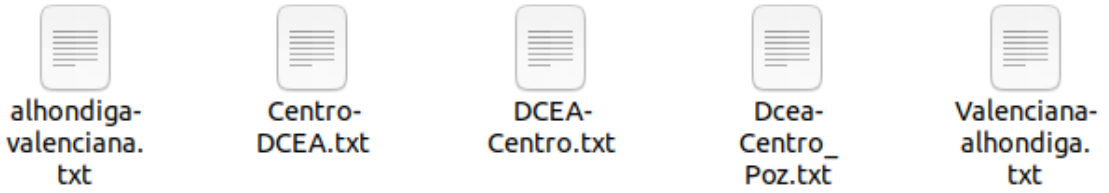

Figura 15. Rutas agregadas actualmente

Después de haber explicado el formato en que deben de estar los archivos de las rutas, se podrá entender correctamente cómo es que se crean los grafos para cada ruta, es decir, a partir de cada archivo de texto como se crea un grafo dirigido para las rutas con las que se cuenta.

Para esto se tiene una función llamada read_grafo_txt, la cual tiene como parámetro un archivo de texto. Y en pocas palabras, este método lee el archivo de texto, separa

a partir de la tabulación el nombre de la parada, latitud y longitud, y las guarda en tres variables separadas, para que de esta manera sea más fácil manipular los datos.

Habiendo creado ya un grafo dirigido, se va agregando un nodo por cada iteración que hace al leer un renglón del archivo de texto. Y a este nodo nuevo que se agregó se le añaden los atributos que se mencionaron anteriormente: nombre de la parada, latitud y longitud. Después de esto va construyendo aristas en el orden que aparecen las rutas de los camiones, la primera parada se conecta con la segunda, la segunda con la tercera, y así sucesivamente hasta llegar al final de la ruta con la última parada. Además de eso, les da un valor de 1 a las aristas para que exista una facilidad de conexión porque se encuentran en la misma ruta.

Estos grafos se guardan en un arreglo y ese mismo arreglo es el que se usa en el primer método, llamado union_graphs.

Para poder visualizar el grafo, hacemos uso de las bibliotecas NetworkX y matplotlib, con sus métodos draw y show respectivamente.

Para ejecutar este algoritmo, basta con lanzar desde la consola de Ubuntu:

python mi_algoritmo.py **--path** mis_rutas

Donde mi_algoritmo es el nombre del archivo python donde está escrito el código y mis_rutas es la carpeta donde se encuentran todos los archivos de las rutas con sus respectivas paradas.

Y también con la biblioteca matplotlib genera una imagen de cómo se vería el grafo.

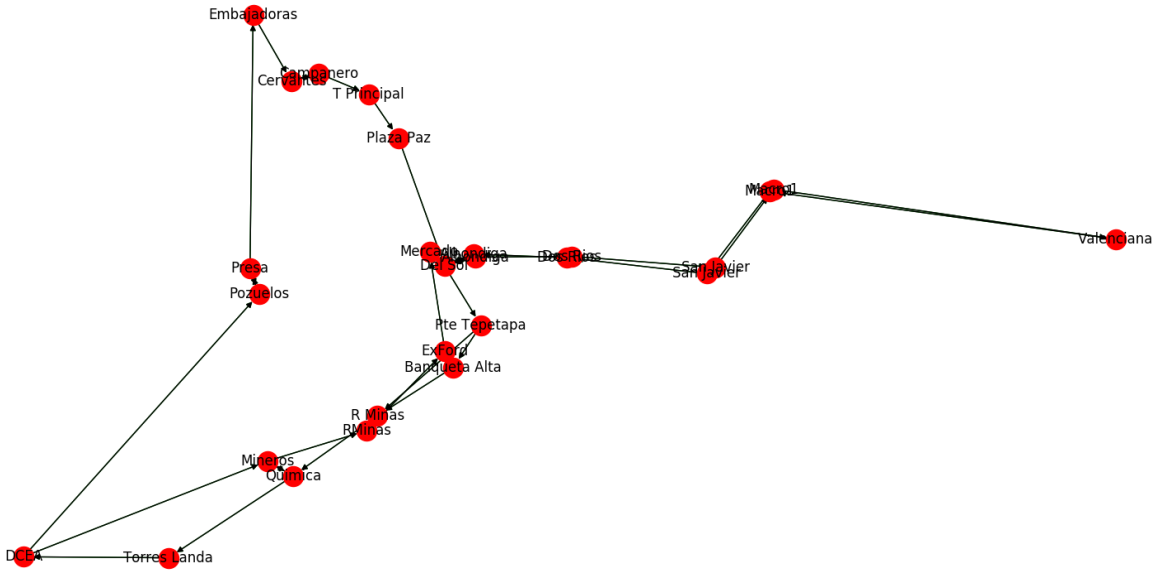

Figura 16. Representación gráfica del grafo con las rutas anteriormente agregadas

El formato en el que se guarda el grafo es un archivo de texto donde guarda una lista de todos los nodos con sus respectivos atributos, y seguido de otra lista donde se indican como están conectadas las aristas.

Este archivo que se genera, es el que nos permitirá hacer las consultas o queries en el programa de Java para poder obtener la ruta óptima.

3.4 Queries para la obtención de rutas

En la parte de la aplicación en Java la cual calcula la ruta óptima a partir del archivo de salida que genera Python, tiene una clase llamada grafo, esta clase contiene todo lo necesario para trabajar con el archivo de Python anteriormente mencionado.

Para la creación de queries en este punto se utilizó lo siguiente:

- Archivo generado en Python con el grafo fuertemente conexo con las rutas conectadas.
- IntelliJ IDEA

A continuación, se describen los elementos que tiene la clase con los cuales se trabaja:

- **Integer num_nodes:** número de nodos que tiene el grafo.
- **Integer num_aristas:** número de aristas que tiene el grafo.
- **String[] paradas:** arreglo que guarda el nombre de las paradas de las rutas del transporte público, que son los nodos del grafo.
- **double [][] positions:** arreglo de dos dimensiones que contiene las coordenadas (latitud y longitud) de las paradas del camión.
- **Map <Vector <Integer>, Double> distancia:** Vector de dos entradas como argumento, el cual representa una arista, y este devuelve la distancia que conecta a los dichos puntos (nodos).
- **Map <Vector <Integer>, Vector <String>>rutas_paradas:** Tiene como argumento un vector de dos entradas (arista). Regresa un vector de Strings, los cuales son los nombres de las rutas que conectan con dichas paradas del camión.

Ahora bien, describiremos las funciones que tiene esta clase y que utilizan las queries para obtener la ruta óptima:

- **grafo(String ruta):** es el constructor de la clase, al cual se le pasa un archivo como parámetro y lee el archivo creado con Python donde contiene todos los atributos del grafo de las rutas disponibles con caminos de todos contra todos (fuertemente conexo).

- **closest_bus_stop(double[] pos):** tiene como parámetro una posición geográfica (coordenada) y a partir de esta posición regresa la parada más cercana con respecto a la coordenada que se proporcionó.

- **dijkstras(int a, int b):** tiene como parámetros dos puntos geográficos los cuales se interpretan como punto a y punto b, es decir, el lugar de inicio del recorrido y el destino. Regresa un vector de enteros, los cuales son los identificadores de las paradas del camión por la que se tiene que pasar.

- **print_route(Vector<Integer> v):** tiene como parámetro un vector de enteros, básicamente se envía el resultado del método anterior (dijkstras) para que este método lo interprete e imprima la información detallada del recorrido que tiene que hacer el transporte.

A continuación, se presenta parte del código en Java, específicamente del método Main:

```
public static void main(String[] args) throws FileNotFoundException {

    grafo f = new grafo("lista_aristas_transporte_gto.txt");

    System.out.printf("\n\nTermino de leer grafo \n\n");

    double[] start = new double[2];
    double[] end = new double[2];

    start[0] = 21.003527;
    start[1] = -101.271159;
    System.out.printf("Comenzamos en viaje en la coordenada (%f,%f) \n", start[0], start[1]);

    end[0] = 21.042177;
    end[1] = -101.258009;
    System.out.printf("Queremos llegar a la coordenada (%f,%f )\n \n", end[0], end[1]);

    int parada_start = f.closest_bus_stop(start);
    int parada_end = f.closest_bus_stop(end);

    Vector<Integer> path1;
    path1 = f.dijkstras(parada_start, parada_end);

    System.out.printf("\n\n Ruta del camino \n\n");
    f.print_route(path1);

}
```

Figura 17. Código Java para obtener la ruta óptima

En grafo *f* se guarda el archivo generado con Python donde está toda nuestra red de rutas.

En los arreglos start y end, se guarda la geolocalización de inicio y la del destino. Y con closest_bus_stop busca la parada más cercana a partir del parámetro el array que contiene la coordenada de inicio o la del destino.

Cuando se llama al método dijkstras comienza a calcular la ruta óptima dadas las paradas de autobús que se obtuvieron con closest_bus_stop. Y finalmente, print_route imprime a detalle los nodos (paradas) por donde se pasa para llegar al destino, y también especifica si se debe tomar un camión o se debe ir a pie.

Para que sea posible que la aplicación móvil pueda hacer peticiones a estos métodos vistos, es necesario crear una comunicación y así poder hacer peticiones, entonces se deberá crear una comunicación cliente – servidor.

3.5 Comunicación Cliente - Servidor

En la aplicación móvil para encontrar la ruta óptima fue necesario crear un protocolo de comunicación cliente – servidor, donde el servidor es una aplicación en Java, y la parte del cliente dentro de la aplicación Android.

A continuación, se muestran las partes tanto del servidor como la del cliente.

3.5.1 Servidor

Para una mejor comprensión de lo que se habla, se muestra una parte del servidor en Java:

```
public class Servidor {
    public static void main(String args[]) throws IOException {
        System.out.println("Servidor iniciado");
        //Inicializando Grafo con el archivo del grafo creado con Python
        Grafo f = new Grafo( ruta: "asd.txt");
        System.out.printf( s: "\n\nEl grafo ha sido leido\n\n");
        System.out.println("Esperando Clientes");
        //Especifica el puerto al que se conecta
        ServerSocket ss = new ServerSocket( i: 8007);
        while (true) {
            Socket cliente = ss.accept();
            System.out.println("Nuevo cliente conectado");
            new Thread(new Runnable() {
                @Override
                public void run() {
                    BufferedReader in = null;
                    try {
                        in = new BufferedReader(new InputStreamReader(cliente.getInputStream()));
                        BufferedWriter out = new BufferedWriter(new OutputStreamWriter(cliente.getOutputStream()));
                        // Comunicación con el client
                        String line = in.readLine();
```

Figura 18. Fragmento de código del servidor TCP en java

Como se puede observar en la imagen del servidor, se llama a la clase Grafo (vista en la sección 3.4), la cual contiene todos los métodos que se utilizan para la obtención de la ruta óptima. Para llamar a la clase se hace uso de su constructor, y este constructor de la clase Grafo requiere un archivo de texto. Este archivo de texto es el grafo que se creó en el algoritmo de Python.

Ahora para, poder establecer conexión con clientes, se hace uso de la clase ServerSocket de Java, para la que, al momento de inicializar, necesita que se le

asigne un puerto, donde escuchará a los clientes (ej. 8007). Justo después de creada la instancia, el Socket se queda esperando la conexión con nuevos clientes que intenten entrar por ese puerto que se asignó.

Cuando un cliente establezca la conexión, el código entrará en un ciclo while, se creará una variable nueva para el cliente de tipo Socket y se le asigna el método accept(). Porque como se vio en la *Figura 8.*, esta parte se bloquea hasta que haya una conexión.

Para que el servidor pueda seguir escuchando a nuevos clientes que se conecten si es que alguno se conectó primero, se creó un hilo para los clientes y de esta manera evitar que tengan que esperar a que los atienda el servidor.

Se pueden leer las peticiones de los clientes usando la clase BufferedReader, esta clase es muy importante para la comunicación porque es la que se encarga de leer las peticiones, ya sean del servidor o del cliente, en este caso son las peticiones del cliente las que va a leer. Con el método readLine() se logra leer lo que es cliente mandó, en formato String, para que esto fuera posible se usó la siguiente línea de código.

```
in = new BufferedReader(new InputStreamReader(cliente.getInputStream()));

String line = in.readLine();
```

Donde InputStreamReader usa a la variable cliente y esta variable se creó con el método accept() y el método getInputStream() básicamente toma la información que manda el cliente y por donde la envía para así guardarla para poder leerla con el readLine().

Cuando lee la petición del cliente, obtiene una cadena de caracteres y esa cadena contiene coordenadas de inicio y final. Dicha cadena ya tiene un formato definido, el cual se explica con más detalle en la siguiente sección, pero teniendo en cuenta esto se podrá manipular la cadena y obtener los datos correspondientes. Sabiendo que la petición del cliente contiene coordenadas entonces se puede hacer uso del

método visto en la sección **3.4 closest_bus_stop(double[] pos)** que recibe un arreglo que tiene que contener latitud y longitud.

Entonces para cumplir con eso, se crearon dos arreglos (start[] y end[]) de tipo double y usando StringTokenizer en la cadena que se recibe del cliente, se fue dividiendo para así llegar al final y solo tener de manera independiente la latitud y longitud, tanto del inicio y final, y como se recibió una cadena String, se debe aplicar un parse para poder convertir a double y así poder mandar los parámetros correctos.

Ya teniendo correctos los parámetros, ahora si se puede encontrar la parada más cercana a la coordenada. Este método retorna un entero, el cual corresponde al nodo donde está la parada más cercana en el grafo. Para poder usar estos nodos, tanto el inicial como el de destino, se crean dos variables para tenerlos ahí.

```
int parada_start = f.closest_bus_stop(start);
int parada_end = f.closest_bus_stop(end);
```

El siguiente paso es llamar al método **dijkstras(int a, int b)** y los parámetros que recibe son justamente los resultados de la parada más cercana, es decir, el nodo inicial y el nodo final. Este método de Dijkstra va a retornar un vector y el cual interesa guardarlo para al final llamar a otro método llamado **print_route(Vector<Integer> v)** y que ayuda a visualizar la ruta óptima y más importante aún, a obtener la respuesta que se va a enviar al cliente. Entonces para poder realizar todo, se debe crear un vector y guardar todo lo que regresa el método Dijkstra.

```
Vector<Integer> path1;
path1 = f.dijkstras(parada_start, parada_end);
String respuesta = f.print_route(path1);
```

Cuando se ejecutan estas tres líneas de código, se obtiene el plan de viaje y entonces ya se tiene la respuesta que se enviará al cliente. Para eso ocupamos

llamar a la variable que se creó al principio de esta clase de tipo Bufferedwriter, llamada out.

Así como existe la clase BufferedReader, existe también la clase BufferedWriter que se encarga de lo contrario, escribir respuestas, se puede usar en el servidor y en el cliente.

```
BufferedWriter out = new BufferedWriter(new OutputStreamWriter(cliente.getOutputStream()));
```

Con esta instrucción, se le asigna a la variable out una salida para escribir la respuesta al cliente que solicitó la petición. La respuesta se obtiene con los métodos vistos en la sección **3.4** que al final, devuelve la ruta óptima a partir de las coordenadas solicitadas. El formato que se envía al cliente es una cadena de caracteres, y contiene desde paradas, rutas, coordenadas de cada una de las paradas, etc.

Ya al final cuando se tiene dicha respuesta, lo último es mandarla al servidor con el método write(), lo que se debe enviar está dentro de la variable out, entonces se debe usar la instrucción out.write() y seguido de esta instrucción, out.flush(). Esto para liberar los bytes que estan alojados en el Buffer.

Y por último se cierra la conexión con el cliente para que no se quede activa y no sobrecargar el servidor con muchas conexiones al mismo tiempo: in.close(); y cliente.close();

3.5.2 Cliente

```
public class ClientePlanDeViaje {
    @SuppressLint("NewApi")
    public static CompletableFuture<String> enviarPeticion(final double[] start, final double[] end){
        return CompletableFuture.supplyAsync(() -> {
            Socket cliente = null;
            try{
                String msjInicio = start[0]+","+start[1];
                String msjFin = end[0]+","+end[1];
                //Aqui va la IP donde se conecta, junto con el puerto para establecer la comunicación
                cliente  = new Socket( host: "10.10.100.234",  port: 8087);
                BufferedReader in = new BufferedReader(new InputStreamReader(cliente.getInputStream()));
                BufferedWriter out = new BufferedWriter(new OutputStreamWriter(cliente.getOutputStream()));
                System.out.println("Enviando petición");
                out.write( str: msjInicio+"|"+msjFin+"\n");
                out.flush();
                System.out.println("Leyendo ...");
                //Aqui viene la respuesta
                String line = null;
                StringBuilder sb = new StringBuilder();
                while((line= in.readLine())!=null){
                    sb.append(line);
                    sb.append("\n");
                    System.out.println(line);
                }
            System.out.println("Se ha leído:\n"+sb.toString());

                return sb.toString();
            }catch(IOException e){
                e.printStackTrace();
                return "No se ha podido conectar con el servidor: Error";
            }finally {
                try {
                    cliente.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        });
}
```

Figura 19. Fragmento de código cliente TCP en java (Android)

El funcionamiento de esta clase es simple, se puede decir que lo que manda el cliente son las coordenadas que eligió el usuario (en la interfaz de la aplicación), tanto de inicio como de destino, para después mandar una petición al servidor y esperar la respuesta con todas las instrucciones de la ruta óptima que se deben

tomar y con esas instrucciones (coordenadas) poder dibujar en el mapa la ruta a seguir.

Como se puede observar en la imagen del código, se usó la clase CompletableFuture, se hizo uso de esta clase porque se necesitaba que no se saturara el hilo principal y el CompletableFuture sirve para eso. Nos ayuda a no bloquear las tareas trabajando con tareas asíncronas. La programación asíncrona es simplemente escribir código en un hilo separado al principal, y el hilo principal será notificado del progreso, éxito o fracaso del hilo secundario.

Ejecutando estas tareas asíncronas el hilo principal no se bloquea y eso es lo que se busca en la conexión cliente – servidor, el cliente debe intentar la conexión, pero si a la vez varios clientes se conectan, no hacer esperar a los demás bloqueando los procesos de cada uno de ellos, si no usando el CompletableFuture y que se obtenga la respuesta cuando el servidor responda a los clientes que solicitan una respuesta.

Esa es la razón por la que al crear el método “enviarPeticion” es de la clase de CompatibleFuture<String>. El método tiene dos parámetros, los cuales son de tipo double[], estos dos parámetros corresponden a las coordenadas de inicio y de destino (start y end) del plan de viaje, donde la variable start contiene la latitud y longitud del punto inicial y la variable end contiene la latitud y longitud del punto del destino.

CompletableFuture trabaja con funciones lambdas, es por eso que regresa como resultado una lambda, y en esta lambda está el código de la tarea asíncrona. Ahí dentro se crea el cliente con la clase Socket. Después, para enviar una petición, se hace uso de los parámetros que pide el método creado para concatenarlos en una sola variable donde irán las coordenadas de origen y destino. Algo que es muy importante es inicializar la variable cliente, para inicializarla, la Clase Socket, pide un host y un puerto, los cuales se definen desde el servidor, para que así correspondan ambas partes, entonces se toma en cuenta el puerto y a que dirección

IP está conectado el servidor, en este caso se usó el host interno en la red del CIMAT y el host que se especificó en el servidor (8007).

En el cliente se usan las clases BufferedWriter y BufferedReader como se usaron en el servidor, estas dos clases son necesarias tanto en el cliente y en el servidor porque si no las tuvieran, la comunicación sería imposible.

Con la variable creada de BufferedWriter, se envían los parámetros del método “enviarPeticion”, esos parámetros son las coordenadas de inicio y de final, estas se envían en formato String, es decir, el método recibe dos arreglos de tipo double y se concatenan para que el formato sea el siguiente:

“LatitudInicial+,+LongitudInicial+|+LatitudDestino+,+LongitudDestino”.

Donde el símbolo “|” sirve para diferenciar donde acaban las coordenadas de inicio, y donde comienzan las coordenadas de destino. Y las comas entre las coordenadas, sirven para diferenciar la latitud de la longitud. El mensaje, como se vio en la parte del servidor, se envía usando el método write, de la clase BufferedWriter, también se debe liberar el espacio, así que igualmente se usa el método flush.

Cuando se envía al servidor, este regresará la respuesta, para lo cual se necesitará el BufferReader y también se hace uso de la clase StringBuilder, esta clase nos sirve para crear objetos que almacenan cadenas de caracteres que pueden ser modificadas sin necesidad de crear nuevos objetos. Esta clase usa el método append que junto con otros los otros métodos que posee la clase de StringBuilder que son replace e insert, sirven para manipular las cadenas de caracteres.

Se hace uso del método append dentro de un ciclo para la respuesta del servidor y así obtener la respuesta que mandó el servidor al cliente. Y esa variable de StringBuilder, es lo que retorna el método del cliente llamado “enviarPeticion”, dicha variable contiene las instrucciones para obtener el plan de viaje. En este punto ya se podrá manipular según se necesite.

3.6 Funcionamiento e implementación de la activity en Android

Para el desarrollo de la activity en Android, se tomaron en cuenta las necesidades que se propusieron para el funcionamiento y diseño de la app. Cabe mencionar que se trabajó en una aplicación ya creada anteriormente, donde ya se tenía el monitoreo del sistema de transporte de la ciudad de Guanajuato, ahora agregando un nuevo apartado de plan de viaje para de esta forma tener una aplicación cada vez más completa.

La idea principal del funcionamiento de la aplicación es que el usuario pueda elegir dos puntos en un mapa y que despliegue la ruta que se debe tomar y además de eso debe de desplegar unas instrucciones en texto para especificar el camino y cada una de las paradas y las rutas que corresponden con cada parada.

3.6.1 Implementación de la API de Google Maps

Para lograr lo antes mencionado, se tuvo que implementar la API de Google Maps en Android Studio para poder desplegar el mapa en la activity del plan de viaje. Según Google Developers se debe de implementar OnMapReadyCallback, y así una vez que se establece una instancia de esta interfaz en un objeto MapFragment (el cual se implementará en seguida) o MapView el método onMapReady(GoogleMap) se activa cuando el mapa esté listo para usarse y proporciona una instancia no nula de GoogleMap.

Dentro del método de onMapReady ya se puede trabajar con los mapas, esto es porque teniendo este método implementado, se puede programar todo lo necesario como “Listeners”, “Polylines”, “Markers”, etc. Estos elementos mencionados, son necesarios para la funcionalidad de la aplicación, por ejemplo, los Listeners ayudan a saber cuándo el usuario toca la pantalla del dispositivo y de esta manera, colocar Markers (marcadores) en el mapa y estos a la vez ayudan a calcular las rutas y las rutas se dibujan con las Polylines (polilineas). Por esta razón cada uno de los elementos mencionados anteriormente, se llevan de la mano y son importantes para el desarrollo de la aplicación para calcular la ruta óptima.

Dentro del método onMapReady, la página oficial de Google Developers recomienda inicializar una variable de tipo GoogleMap, la cual es una clase que contiene métodos que son útiles cuando se mapas se trata.

Cuando se inicializa un mapa con Google Maps hay un método que sirve para enfocar un lugar en específico, y este es llamado, “moveCamera”.

googleMap.moveCamera(CameraUpdateFactory.*newLatLngZoom*(new LatLng(21.015861, -101.252862), 15));

El método recibe como parámetros la latitud y longitud, además de un número entero que se encarga de dar zoom a la coordenada señalada. Las coordenadas presentadas ahí corresponden a la ciudad de Guanajuato, por lo tanto, cada vez que se inicie la aplicación en esta activity, se mostrará la ciudad de Guanajuato con un nivel de zoom predeterminado de 15, el zoom se puede manipular después con los gestos correspondientes.

3.6.2 Marcadores en el mapa

Ahora para agregar marcadores a la activity de una manera dinámica hay que habilitar el listener del mapa, este es un método llamado onMapClick, este método es necesario, ya que, como se planteó la aplicación el usuario debe colocar dos puntos, el primer punto que se coloque es el punto de inicio, y el segundo punto el de destino para más adelante trazar la ruta óptima. Se muestra el fragmento de código para habilitar el método mencionado.

```
//Para agregar marcadores con un clic en la pantalla
mMap.setOnMapClickListener(new GoogleMap.OnMapClickListener() {
    @RequiresApi(api = Build.VERSION_CODES.N)
    @Override
    public void onMapClick(LatLng punto) {
```

Figura 20. Método onMapClick

Cuando se habilita este método cada vez que el usuario toque la pantalla se ejecutará el código que está dentro. Entonces cuando se toque la pantalla, lo que se espera que haga la aplicación es agregar un marcador, si se toca de nuevo, debe agregar otro y ahí mostrar la ruta, para restringir a solo dos marcadores, se usó un contador, este contador inicia en 0, cuando el usuario da un toque a la pantalla, aumenta en uno, al mismo tiempo que agrega el primer marcador, también al colocar un marcador este tiene la propiedad poder cambiar el icono que se le asigna de manera predeterminada y de obtener las coordenadas de donde fue colocado.

Así que esas coordenadas importan al final para calcular la ruta óptima, dichas coordenadas se guardan en un ArrayList de tipo LatLng para facilitar la manipulación de las coordenadas, y ahora se queda esperando otro clic de parte del usuario para agregar el segundo marcador. Cuando ocurre, mismo caso que el anterior, el contador aumenta en 1 (en este punto el contador tiene un valor de 2) y guarda las coordenadas en el ArrayList, pero aquí entra en una condición, solo entra cuando el contador vale 2.

En esta parte, lo que hace es extraer del ArrayList las coordenadas guardadas, en dos arreglos de tipo double. Las coordenadas que se encuentran en la posición 0 del ArrayList siempre van a corresponder al inicio, y las coordenadas de la posición 1 corresponderán al destino. Así que, aclarado esto, en los arreglos de tipo double se le asignará el de la posición 0 del ArrayList las coordenadas de inicio, y para el otro arreglo las coordenadas que corresponden al destino, es decir las de la posición 1 del ArrayList.

Teniendo las coordenadas ya en arreglos, el siguiente paso es llamar al método “enviarPeticion” (visto en la sección **3.5.2**), y con esto obtener la respuesta de la ruta más corta consultando al servidor.

3.6.3 Creación del TreeMap

La razón principal para usar un TreeMap, es porque se desea dibujar en el mapa los caminos que toman los camiones en la ciudad de Guanajuato. Existe una

herramienta para ello, esta herramienta se llama API Directions y es parte de Google Platform, pero el gran problema que tiene es que es de paga para poder hacer uso de la herramienta. Es por ello que se optó por crear un TreeMap y guardar varias coordenadas dentro y despues consultarlas para dibujar en el mapa.

Para el TreeMap se creó un servicio para poder tener un control para sus versiones y así en caso de que se actualice alguna ruta avisar al servicio y actualizar el TreeMap.

Dentro del TreeMap se guardan varios puntos de una ruta, y estos puntos que se obtienen desde un archivo KML (Keyhole Markup Language) corresponden a coordenadas del camino que toma el camión, de una parada a la parada siguiente.

El TreeMap se genera desde un archivo JSON, como se ha mencionado, se obtiene desde un servicio que se consulta cada vez que se entra a la activity. Y lo que el servicio necesita para poder proceder es la versión de las rutas, la cual se recupera desde una activity anterior con el método “Intent.putExtra”. Pero antes de enviar la versión para obtener el TreeMap, se debía verificar si ya estaba creado el TreeMap, para que de esta manera no se cree cada vez que se entre al activity donde se usa y así ahorrar tiempo de carga y ahorrar datos del usuario.

Para eso primero se verifica con la clase File que contiene el método exists(), y en base a ese método se sabe si el archivo existe o no, si no existe, la versión que se recuperó, no se envía, y en este caso se manda otra versión, en este caso 0, para que así si o si cree un nuevo TreeMap. Cuando el archivo ya existe, se deja la versión recuperada, y puede que se llegue a actualizar el TreeMap, pero solo si ya cambio toda la versión la cual sucedería muy rara vez, si no, seguirá siendo la misma versión.

Cuando ya se validó la versión, se llama al servidor y como en sus parámetros pide la versión de las rutas que tiene el cliente, se le manda la versión que se recuperó. Si el servidor tiene la misma versión que se le manda, responderá con un “OK” y eso quiere decir que no hay necesidad de crear otro TreeMap.

De otra manera, el servidor responde con el JSON. El código entra en una condición donde guarda ese JSON de manera local en el teléfono, y lo que se hace después es crear el TreeMap en base a ese archivo que se guarda. A continuación, se muestra una parte del archivo JSON.


```
{
    "Abarrotes_Mineral_de_Valenciana - Plaza_Cúpulas_Ruta_Valenciana-Cupulas": [...
    ],
    "Abarrotes_don_Beto - Glorieta_Benito_Juárez_Ruta_Valenciana-Cupulas": [
        {
            "latitude": 20.996534549,
            "longitude": -101.288216476
        },
        {
            "latitude": 20.996468122,
            "longitude": -101.288281643
        },
        {
            "latitude": 20.996401695,
            "longitude": -101.288346811
        },
        {
            "latitude": 20.996336037,
            "longitude": -101.288412849
        },
        {
            "latitude": 20.996268773,
            "longitude": -101.288476737
        },
        {
            "latitude": 20.996195047,
            "longitude": -101.288532298
        },
        {
            "latitude": 20.996115355,
            "longitude": -101.288571486
        },
        {
            "latitude": 20.996025643,
            "longitude": -101.288571393
        }
    ],
```


Figura 21. Fragmento del archivo JSON que genera un TreeMap

Como se puede apreciar en la figura, vienen los nombres de las paradas y su ruta correspondiente y varios puntos que corresponden al camino que toma el transporte

por las calles de la ciudad de Guanajuato. Entonces al momento que se crea el TreeMap, se guarda como clave – valor, donde la clave al ser irrepetible, se usan los nombres de las paradas y la ruta que corresponde a esas paradas, y el valor, o valores son todas las coordenadas que contiene. Por ejemplo, el formato de la clave es, "Parada_1 – Parada_2_Ruta_Nombre_de_la_ruta", donde Parada_1 corresponde a la primer parada, Parada_2 corresponde a la siguiente parada que sigue despues de la Parada_1, y estas dos tienen que estar forzosamente separadas por un " - " ya que así es el formato en el que su busca más adelante dentro del TreeMap. En la parte de "Nombre_de_la_ruta", corresponde al nombre que tiene la ruta, antes del nombre de la ruta, se debe de contar con los caracteres de "_Ruta_" y cabe aclarar que "Ruta_" debe de venir con "Nombre_de_la_ruta" ya que no se agrega por defecto, mismo caso que el anterior, se busca en el TreeMap con un formato que ya debe estar fijo, por eso es obligatorio. Y este deberá estar entre las paradas y el nombre de la ruta. A continuación, se muestra un ejemplo real de cómo está conformado este formato para la clave del TreeMap, para así comprenderlo mejor: Valenciana – Macro1_Ruta_Valenciana.

Cuando el usuario hace una petición de un plan de viaje, el servidor (como ya se comentó anteriormente) envía la cadena de caracteres con las instrucciones y se separan en diferentes listas, teniendo en cuenta que ya cada una de las listas corresponden con las demás para que así los datos sean correspondientes y tengan sentido. Al momento de obtener esas instrucciones, se llama a la lista donde se tienen los nombres de los segmentos (un segmento se considera como Parada 1 – Parada 2) y el nombre de la ruta, de esta manera creando una clave, se hace la búsqueda en el TreeMap y con cada coincidencia con la clave se crea una lista dinámica para guardar el camino que se recorre en el transporte público a partir de dichas coincidencias.

Para ejemplificar mejor esto se muestra el código con el ciclo en donde se hace todo este proceso antes mencionado y así entenderlo mejor.

```
//Aqui se itera la lista creada antes para buscar las laves del TreeMap
//y así obtener los segmentos de la ruta que se necesitan
List<LatLng> listaSegmento = null;
for (int i = 0; i < segFinal.length; i++){
    String key = segFinal[i]+"_"+rutaFinal[i];
    if (mapaSegmentos.containsKey(key)){
        listaSegmento = mapaSegmentos.get(key);

        System.out.print("\n");
    }else{
        System.out.println("NO SE HA ENCONTRADO LA CLAVE: "+ key);
    }
}
```

Figura 22. Ejemplo de búsqueda en el TreeMap

La explicación para la figura anterior es simple, concatena el segmento con la ruta y hace la búsqueda dentro del TreeMap (mapaSegmentos) si existe una coincidencia con alguna clave, se agregan las coordenadas a esa lista. En caso de que no exista, arrojará el mensaje que se puede ver, generalmente, cuando no encuentra una clave dentro del mapa, es porque esa ruta es caminando, las rutas a pie, no estan dentro del TreeMap. Esa lista se usará más enseguida para dibujar la ruta.

3.6.4 Dibujar Polylines

```
runOnUiThread(new Runnable() {
    @Override
    public void run() {
        //Aqui pinta la linea de los segmentos de la ruta
        dibujador(segFinal,rutaFinal,nombreParada,coordenadaParada);
        //Aqui agrega las paradas como marcadores en el mapa
        for (int i = 0; i<nombreCoor.length-1;i++){
            Bitmap icon = customIcon(nombreParada.get(i), R.drawable.parada_bus_activa);
            mMap.addMarker(new MarkerOptions().position(coordenadaParada.get(i))
                    .icon(BitmapDescriptorFactory.fromBitmap(icon)));
        }
```

Figura 23. Ejemplo de llamada al método "dibujador"

Al usar el método de “enviarPeticion” se está usando la clase CompletableFuture así que, esta clase para obtener los resultados hace uso de funciones lambda para soportar tareas asíncronas (visto en la sección **3.5.2**).

Cuando se obtiene la respuesta del servidor al cual se le hizo la petición, se debe seleccionar las partes que se necesitaran para las coordenadas de cada parada, los nombres de las paradas y las rutas a las que corresponden. Hay que recordar que la respuesta del servidor viene en una cadena de caracteres y esta cadena viene en secciones, pero separadas con ciertos caracteres especiales para diferenciar las secciones en las que viene, además estan separadas de tal forma para que correspondan todas las coordenadas, rutas, paradas, etc.

Entonces en el código se separa la respuesta que envía el servidor hasta tener separados cada uno de los datos de cada parada del plan de viaje, es decir, las coordenadas, los nombres de las paradas, la ruta a la que corresponde esa parada y un dato llamado segmento, este último, corresponde a un par de paradas y sirve para ubicar el camino que toma el transporte entre cada parada, este camino se guarda en el TreeMap.

Para dibujar las rutas del transporte público de una manera dinámica, se creó un método llamado “dibujador” que recibe cuatro parámetros, estos cuatro parámetros son: un arreglo String de los segmentos (par de paradas), un arreglo String de las rutas, un ArrayList de tipo String que contenga las paradas de manera individual y otro ArrayList de tipo LatLng que contenga las coordenadas de las paradas del ArrayList anterior. Todos estos parámetros se consiguen desde la respuesta del servidor antes mencionado (“enviarPeticion”).

```
List<LatLng> listaSegmento = new ArrayList<>();
Polyline segmentoNormal;

for (int i = 0; i < nombreSegmentos.length; i++){
    String key = nombreSegmentos[i]+"_"+nombreRutas[i];
    if (mapaSegmentos.containsKey(key)){
        List<LatLng> seg = mapaSegmentos.get(key);
        if (seg != null)
            listaSegmento.addAll(seg);
        System.out.print("\n");
    }else{
        String parada = nombreSegmentos[i].substring(0, nombreSegmentos[i].indexOf(" - "));
        String parada2 = nombreSegmentos[i].substring(nombreSegmentos[i].indexOf(" - "));
        for (int y = 0; y < listaParadas.size();y++){
            if (parada.equals(listaParadas.get(y))){
                listaSegmento.add(listaCoorParadas.get(y));
            }
            if (parada2.equals(listaParadas.get(y))){
                listaSegmento.add(listaCoorParadas.get(y));
            }
        }
```

Figura 24. Ejemplo de llenado de una lista de puntos para después dibujarla en el mapa

El fragmento de código que se presenta, es lo que tiene dentro del método “dibujador” y es la manera en que funciona. Crea una lista para guardar las coordenadas (“listaSegmento”) y un objeto Polyline (“segmentoNormal”).

A partir de los parámetros que se piden en el método hace la búsqueda en el TreeMap creado, si hay coincidencia en la búsqueda, se agrega a la “listaSegmento”, y se itera hasta llegar al último elemento.

En el caso de no encontrar la clave en el TreeMap, lo que se hace es conseguir las coordenadas de la parada individualmente, lo que quiere decir esto es, como se busca por segmento (Parada 1 – Parada 2) con la ayuda del método substring, se recorta Parada 1, y Parada 2, y en lugar de buscar en el TreeMap, se busca en la lista de los parámetros del método, así se encuentra la coordenada correspondiente a cada parada y se agrega a la “listaSegmento”.

Solamente al final, el método regresa la Polyline “segmentoNormal” que es lo que se dibuja. Y con el método “addAll” permite agregar una lista de tipo LatLng, justamente la lista creada “listaSegmento”.

```
return segmentoNormal = mMap.addPolyline(new PolylineOptions().addAll(listaSegmento).color(ContextCompat.getColor(con
```

Figura 25. Ejemplo de método addAll de la clase Polyline

Otra cosa importante y de la cual se hace uso, son los patrones que tienen las Polyline. Hay diferentes tipos de patrones que se pueden usar para dar estilo a las líneas, en este caso, se usó el patrón línea punteada.

```
//Creación de un patrón para la polilinea (linea punteada)
List<PatternItem> patronesLinea = Arrays.asList(new Dash( v: 30), new Gap( v: 20));
//Pintamos la linea punteada del punto inicial a la parada mas cercana
Polyline ruta = googleMap.addPolyline(new PolylineOptions().clickable(false)
        .add(listaMarcadores.get(0), listaMarcadores.get(1)).pattern(patronesLinea).zIndex(200));
```

Figura 26. Ejemplo de uso de patrones con polylines

Se usa el objeto PatternItem para crear una lista y ahí guardar este patrón, para cuando se utiliza el método addPolyline dentro de las opciones que permite, agregar dicho patrón con la lista creada anteriormente, justo como se puede ver en el fragmento de código.

3.6.5 Instrucciones en RecyclerView

Al mismo tiempo en que se dibuja la Polyline, se inicializa un RecyclerView para que se visualicen las instrucciones a manera de texto, y agregar funcionalidades al mismo RecyclerView. Por ejemplo, al buscar cierta parada y tocar en el espacio correspondiente de las instrucciones, se enfoque en el mapa la parada inicial y resaltar el camino hacia la parada siguiente.

Eso se logra gracias a la respuesta que se envía desde el servidor y despues es separada en arreglos para así tener independientes los datos de las rutas (hay que recordar que la respuesta es una sola cadena de caracteres y despues se separa a través de ciclos).

```
//#######Resultados para el RecyclerView#######
String[] resultadoRV = resultadoPeticion.split( regex: "!");
String resultadoParadas = resultadoRV[0];
String resultadoRutas = resultadoRV[1];
String[] paradasSeparadas = resultadoParadas.split( regex: ",");
String[] rutasSeparadas = resultadoRutas.split( regex: ",");
//Se llena una lista para usarla en el RV
for (int i = 0; i < paradasSeparadas.length; i++) {
    listaElementosRV.add(new ModeloRutas(paradasSeparadas[i], rutasSeparadas[i]));
    System.out.println(paradasSeparadas[i]+", "+rutasSeparadas[i]);
}
```

Figura 27. Arreglos que son utilizados para llenar el RecyclerView con instrucciones

Como se observa en la imagen, las variables “paradasSeparadas” y “rutasSeparadas” son las cadenas que se mostraran en el RecyclerView, cada uno de esos elementos se guardan en una lista y a continuación, se muestra cuando se carga esa lista en el RecyclerView y carga los adaptadores.

```
runOnUiThread(new Runnable() {
    @Override
    public void run() {
        rvRutaOptima = findViewById(R.id.rvRutaOptima);
        rvRutaOptima.setHasFixedSize(true);
        rAdaptador = new RutasAdaptador(listaElementosRV);
        rvRutaOptima.setLayoutManager(new LinearLayoutManager( context: RutaOptimaMapa.this,
                RecyclerView.HORIZONTAL, reverseLayout: false));
        rvRutaOptima.setAdapter(rAdaptador);
    }
});
```

Figura 28. Ejemplo de inicializar el RecyclerView

Al inicializar el RecyclerView, se manda a llamar a su layout (se debe recordar que es necesario que el RecyclerView se le asigne un layout para manejar el estilo que tendrá) para que cuando se llame al adaptador del recycler, asigne los datos que son de interés a cada uno de los elementos en el layout, que simplemente son dos etiquetas, en la primera aparecen las paradas y en la segunda etiqueta aparece el nombre de la ruta.

También, es necesario programar el evento de “clickListener” ya que el elemento RecyclerView no cuenta con ese evento y es necesario para la aplicación, porque se quiere que cada vez que se le dé clic al elemento del recycler, resalte la ruta, así como se explicó al principio de esta sección.

```
//Aqui entra si se le da clic al recycler
public void onItemClick(View view, int position) {
    //Busca por la posicion del elemento del recycler para generar la clave y buscar en el arbol la clave que se genero
    String clave = listaElementosRV.get(position).getParadas()+"_"+listaElementosRV.get(position).getRuta();
    System.out.println("LA CLAVE SELECCIONADA ES: " + clave);
    //Busca en el mapa la clave
    if (mapaSegmentos.containsKey(clave)){
        List<LatLng> seg = mapaSegmentos.get(clave);
        //Si la polyline lleva algo, la borra y limpia la lista de segmentos
        if (rutaResaltada != null){
            //listaTempSeg.clear();
            rutaResaltada.remove();
        }
```

Figura 29. Llamada al clickListener del RecyclerView y resaltar rutas especificas

Así que cuando se le da clic a un elemento del recycler, se buscará en la lista de donde se obtienen los datos y así obtiene los elementos dependiendo de la posición que se obtiene con el evento “clickListener”. Cuando ya se guardan los datos del elemento que se seleccionó, genera una clave para buscarla en el TreeMap y así en caso de existir dicha clave, guardar los puntos del camino entre las paradas.

```
//Enfocamos la pantalla a la parada inicial del segmento
googleMap.moveCamera(CameraUpdateFactory.newLatLngZoom(
        new LatLng(seg.get(0).latitude,seg.get(0).longitude) , v: 17));
//Creamos la polyline que resaltara el segmento de color dorado
rutaResaltada = googleMap.addPolyline(new PolylineOptions().addAll(seg)
        .color(ContextCompat.getColor( context: RutaOptimaMapa.this, R.color.Dorado)).zIndex(200));
```

Figura 30. Ejemplo de enfoque a parada seleccionada y cambio de color de ruta

Y después de haberlos obtenido, resaltar la ruta y enfocar la misma para que los usuarios que no estén familiarizados con la ciudad tengan una idea de donde se encuentra la parada y el camino que toma. La ruta se enfoca con el método “moveCamera” del cual ya se habló anteriormente. Y así mismo, se llama al método de la clase Polyline (visto en la sección **3.6.4**) “addPolyline” que invoca la clase de “PolylineOptions” y ayuda a personalizar la polyline que se va a crear.

Capítulo 4

Evaluación

4.1 Resultados de la implementación del plan de viaje

Se obtuvieron algunos resultados que fueron satisfactorios en su mayoría porque la aplicación funcionó tal como se esperaba, aunque hubo algunos errores en ciertos casos en los que el plan de viaje actúa de una manera inesperada, estos problemas son de optimización de parte del grafo, la ruta óptima si funciona correctamente, pero al aplicarse el algoritmo de Dijkstra, este trabaja sin tomar en cuenta ciertos casos específicos donde la ruta carece de sentido al ser aplicado en la vida real. Estas limitaciones se explican más adelante.

4.1.1 Primera fase de la implementación del plan de viaje

En una fase temprana de la implementación del plan de viaje para iniciar se implementó una comunicación cliente – servidor simple, la cual ya se explicó en el la **sección 3.5**.

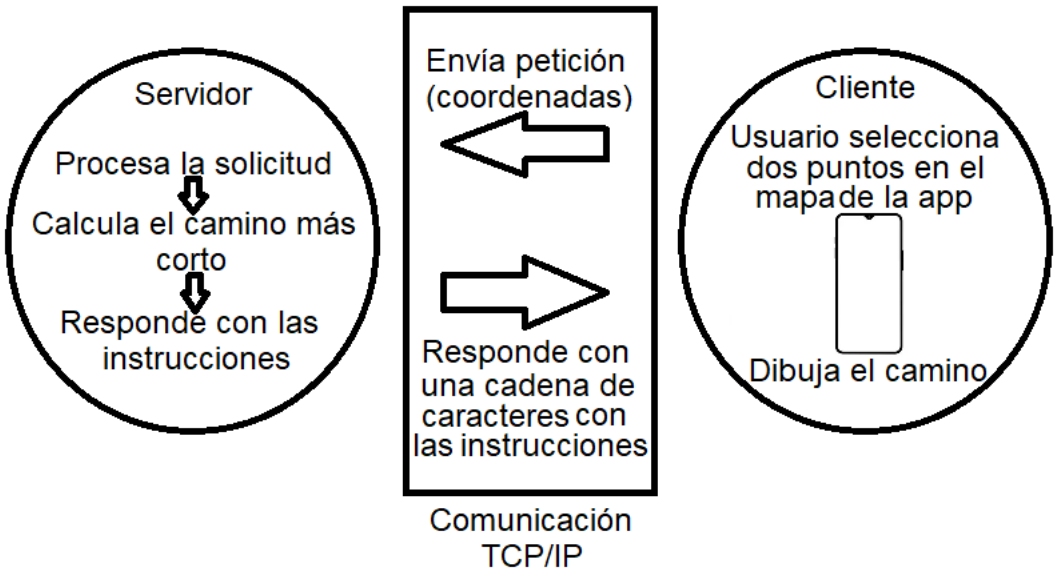

Figura 31. Explicación de la comunicación durante la primera fase de pruebas

Al ser una implementación en una aplicación ya desarrollada, se agregó una nueva activity a la aplicación móvil. Para acceder a esta activity se debe de navegar de la siguiente manera.

Figura 32. Pantalla de inicial de la aplicación SIIBUS

Esta es la pantalla de inicio, se muestra cuando se abre la aplicación. Contiene el apartado de ciudades evaluaciones y notificaciones, cuando se elige la sección de ciudades abre otra activity que despliega una lista de las ciudades donde hay soporte para la aplicación.

Figura 33. Pantalla de la aplicación con la lista de ciudades disponibles

Aqui aparece la lista de ciudades disponibles para monitorear el sistema de transporte. Para este trabajo y para hacer las pruebas correspondientes, se seleccionó la ciudad de Guanajuato, ya que es la única ciudad que contará con un sistema de plan de viaje por el momento.

Figura 34. Pantalla de la aplicación con las rutas de la ciudad de Guanajuato

Dentro de esta pantalla, se muestra la lista de rutas disponibles de la ciudad que se eligió, en este caso Guanajuato, y en la esquina superior derecha aparece un botón con las siglas RO (Ruta Óptima) es ahí donde se debe entrar para poder hacer las búsquedas del plan de viaje. Un punto importante de esta activity es que para mostrar esas rutas consulta un servicio, y cuando lo hace se genera un dato importante, el cual es la versión de los KML (rutas), este dato será importante más adelante.

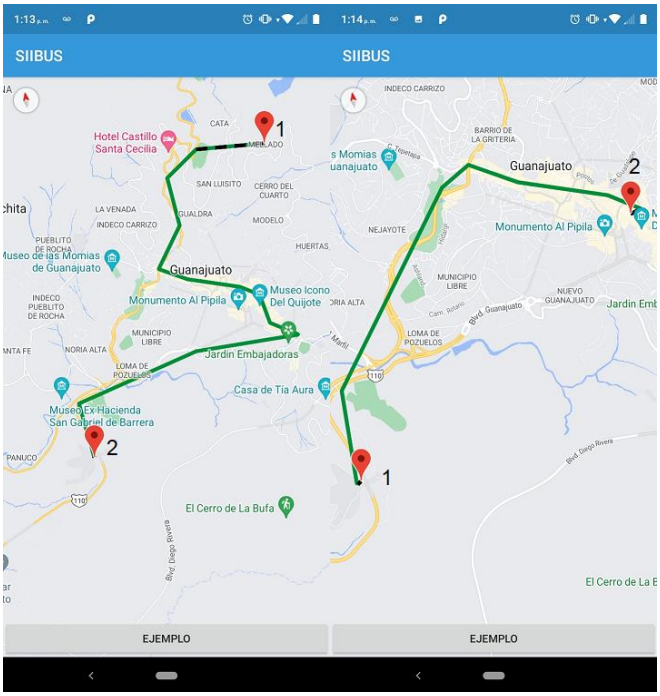

Figura 35. Primeros resultados de la aplicación

Desde la aplicación (cliente) se manda la petición, esta petición envía las coordenadas del punto de origen (1) al punto de destino (2) y el servidor procesa la petición. Después de procesar la petición se genera una respuesta y esta respuesta contiene la ruta óptima que se dibuja de una manera muy simple porque esto se hizo en una fase muy temprana del desarrollo, es por eso se muestran varias líneas rectas, estas líneas van desde una parada a la siguiente según las instrucciones que se mandan desde el servidor. Vale la pena mencionar que cuando el usuario elige el punto de inicio (1) y el de final (2) se dibuja en el mapa una línea punteada

para especificar que se debe caminar desde ese punto seleccionado hasta la parada, está parada es la más cercana que se encuentra al punto elegido por el usuario eso lo calcula el algoritmo del servidor.

Esta fue la primera fase del desarrollo de la aplicación del plan de viaje. Lo siguiente que se tuvo que implementar fue mejorar el dibujado de las rutas.

4.1.2 Segunda fase de la implementación del plan de viaje

En esta fase se implementó un dibujado para las rutas más exacto, antes solo se conectaban las paradas con solo líneas rectas, pero no tomaba en cuenta el camino real que toma el transporte (es decir, las calles o avenidas por donde pasa el camión). Para que esto fuese posible, se implementó un TreeMap, y a través de este se hace una búsqueda con una clave para obtener los valores que contiene, dichos valores son un conjunto de coordenadas del camino por donde pasa el camión. El TreeMap se manda al cliente a través de un archivo JSON con un servicio implementado desde el servidor.

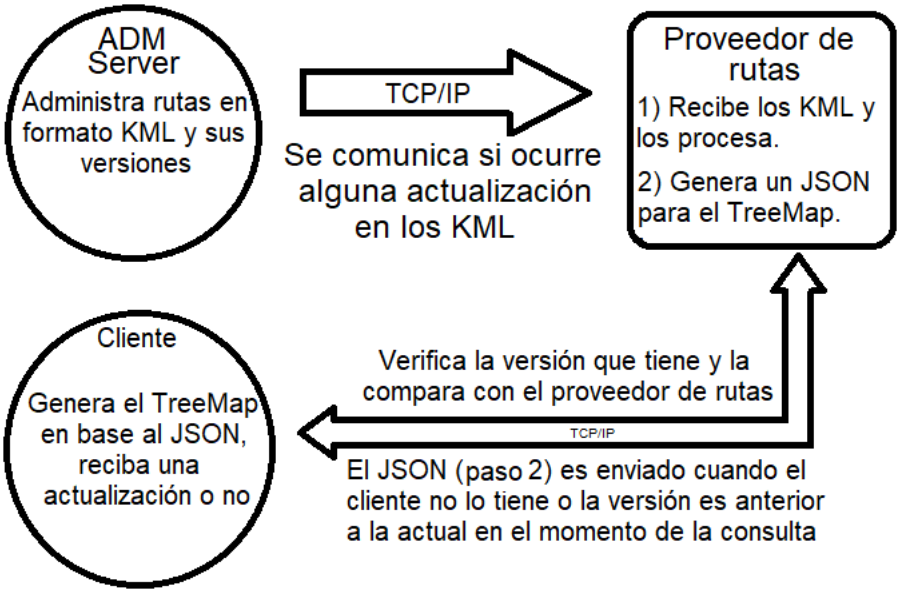

Figura 36. Funcionamiento de la comunicación para la obtención del JSON

Como se aprecia en la figura, el servicio se implementa en un servidor diferente al que se hacen las consultas de la ruta óptima, esto se planteó así para no sobrecargar el servidor que esperará a los clientes, y de esta manera evitar que los tiempos de espera sean muy largos, ya que existe la posibilidad de que, si hay miles de clientes haciendo peticiones al mismo servidor, pero a diferentes servicios, procesar todas las peticiones sería tardado.

Así que cuando el cliente recibe el JSON lo siguiente que hace es crear el TreeMap en base a este archivo (véase la sección 3.6.3) y de esta manera estaría listo el TreeMap para hacer las búsquedas cuando haya respuestas de las peticiones de una ruta. Como, por ejemplo.

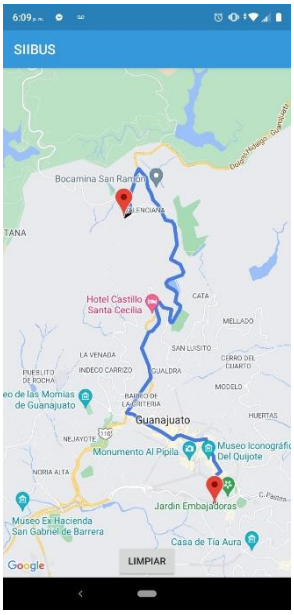

Figura 37. Ejemplo de ruta óptima con el TreeMap ya implementado

De esta manera cuando se hace una petición al servidor y obteniendo las instrucciones con las que responde, al hacer las búsquedas correspondientes al TreeMap, la ruta en el mapa aparecerá graficada correctamente en la aplicación como en la figura anterior.

Además, se agregó una funcionalidad extra, el botón de limpiar, sirve para reiniciar los marcadores y líneas dibujadas para realizar una nueva consulta. Resulta ser

muy útil si el usuario llegase a cometer algún error al momento de colocar los marcadores.

4.1.3 Tercera fase de la implementación del plan de viaje

En este punto del desarrollo aún faltaban algunas funcionalidades para la aplicación, una de estas nuevas funcionalidades es la automatización en la creación del grafo, esto solo se ejecutará cada vez que exista una actualización en los archivos KML, aunque estas actualizaciones no son tan comunes y virtualmente no se hacen con mucha frecuencia, es necesario automatizarlo.

Para que esto funcionase, se agregó un nuevo servicio para el funcionamiento. El cual se muestra a continuación.

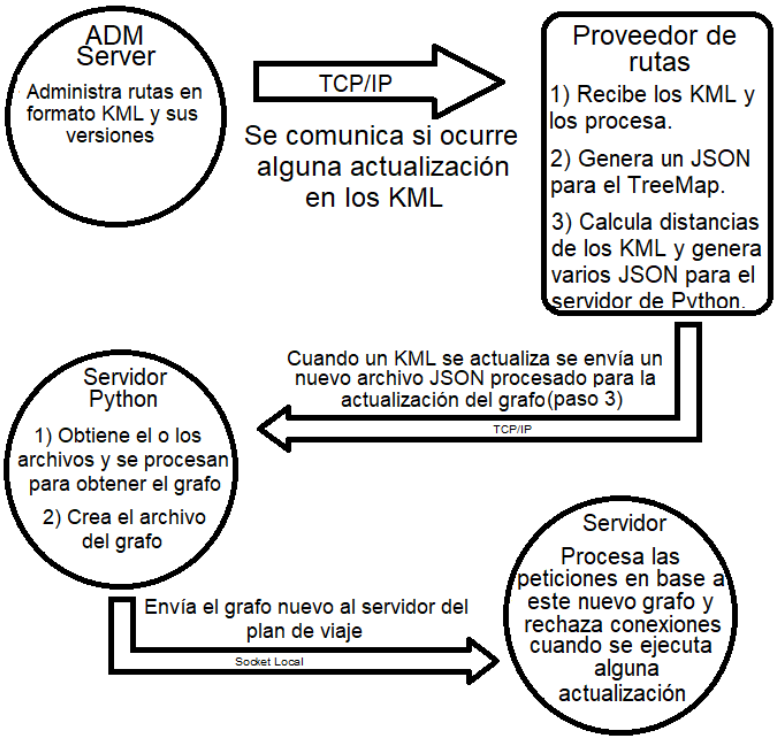

Figura 38. Funcionamiento de la automatización de actualización del grafo

Cuando había alguna actualización en los archivos KML, todo se tenía que hacer manualmente por un administrador, y esta nueva funcionalidad que se agregó en esta parte se ahorra esa parte haciéndolo así automatizado.

Así también se evitan los errores que podían existir cuando a algún administrador olvidaba cambiar estos archivos y se seguían mostrando rutas desactualizadas, ahora, sin importar cuando se actualicen dichas rutas, siempre se contará con la última versión.

Otra nueva función agregada en este punto fue poder ver las instrucciones del camino a seguir y en base a esas instrucciones mostrar en el mapa esa instrucción especifica, esto sirve de ayuda para los usuarios que tengan la aplicación, pero no conozcan la ciudad o la zona, así al verlo al mapa puede brindar una mejor ubicación sobre el lugar por donde pasa el transporte público.

Figura 39. Pantalla de la aplicación con instrucciones en pantalla

Esas instrucciones del camino a seguir son mostradas en la activity de la aplicación a través de un ReciclerView (véase la sección 3.6.5) donde en cada elemento del recycler se va a mostrar un segmento de cada ruta, es decir, Parada 1 – Parada 2

y la ruta de estas paradas, y en caso de ser una ruta que se debe tomar caminando se especificará en la sección de ruta con un enunciado que diga “a pie”.

Para dejar en claro lo antes mencionado, se procederá a mostrar varias capturas de pantalla y a explicar cómo es que funciona esta nueva funcionalidad de la aplicación.

Figura 40. Ejemplo de la búsqueda de ruta óptima

Para una mejor comprensión de cómo funciona el RecyclerView se eligió una ruta simple con pocas paradas y así poder ver con claridad los caminos y las paradas. Y el mensaje del RecyclerView cambia con los datos obtenidos del mapa, es decir, las paradas y las rutas que se toman en cuenta para la ruta.

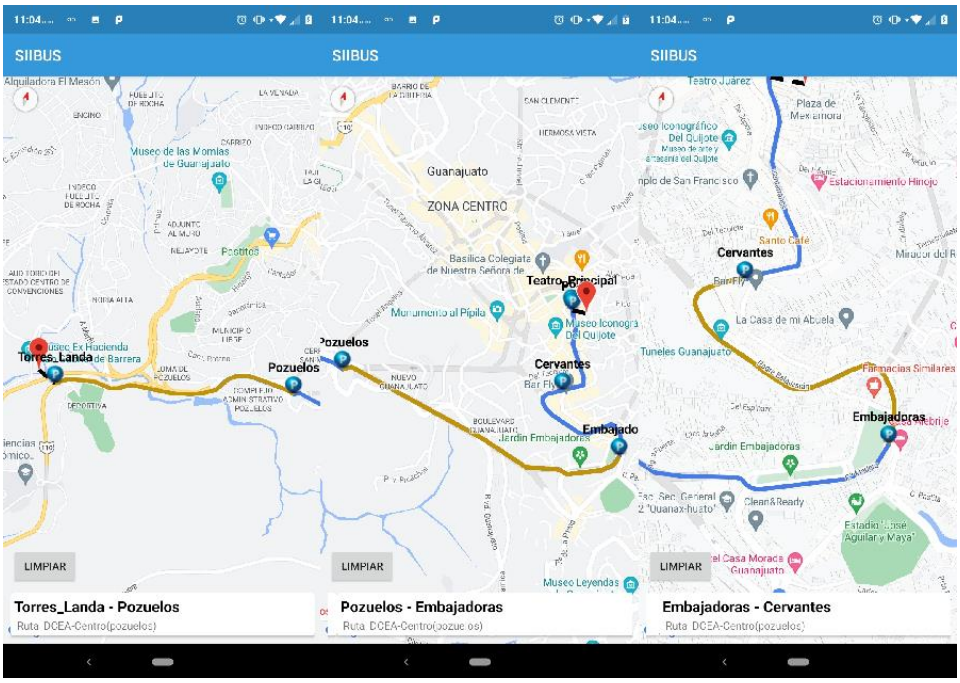

Figura 41. Ejemplo del funcionamiento del RecyclerView

Cuando se elige una ruta y hace todo el proceso de la petición al servidor, esa ruta seleccionada se dibujará en el mapa, y en la parte inferior, donde se encuentra el RecyclerView se encuentra descrito el camino con instrucción por instrucción de cada par de paradas o segmento.

El RecyclerView es de tipo horizontal, es decir, se visualiza hacia los laterales para ver las paradas que contiene la ruta, y también tiene la opción de cuando se le dé un toque al elemento, el mapa indicara el camino de ese par de paradas, cambiando el color de la línea a otro para indicar que es esa la ruta y haciendo un zoom a la parada inicial a la parada final, justamente como se puede observar en los tres ejemplos mostrados en la figura anterior.

Este ejemplo solo fue teniendo en cuenta una ruta muy sencilla y sin muchos cambios de ruta, algo que ya en vida real no es tan común que suceda, pero se tomo como ejemplo para explicar cómo funciona el RecyclerView y como resalta las paradas, más que nada para personas que no conozcan o no ubiquen ciertas

paradas o rutas. Es por eso que a continuación, se muestra una ruta más compleja y más cercana a un ejemplo real de ruta óptima.

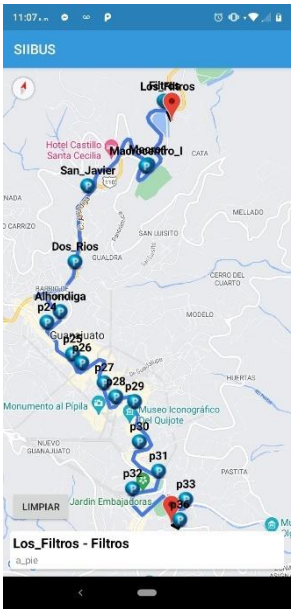

Figura 42. Ejemplo de ruta óptima compleja

Esta es un ejemplo de una ruta óptima considerada más parecida a lo que un usuario promedio escogería para ir de un punto a otro, un trayecto largo en pocas palabras. Y como se explicó en la figura 40, si se elige un elemento del RecyclerView el par de paradas la aplicación muestra en el mapa y se hará el zoom correspondiente a dichas paradas.

4.1.4 Análisis de resultados

Al revisar los resultados obtenidos en las pruebas de la aplicación del plan de viaje, se consideró que el sistema trabajaba de manera aceptable porque cumplía con la función de encontrar la ruta óptima aplicando el algoritmo de Dijkstra.

Sin embargo, se encontraron algunas discrepancias en partes de algunas rutas por donde el algoritmo recomienda que el usuario tome dicha ruta para llegar a su destino.

Estas discordancias que se mencionan van desde indicar al usuario que baje del camión que se encuentra y tome una ruta a pie hacia una parada de la misma ruta en la que iba para volver a tomar el transporte del que ya estaba haciendo uso, volviendo así a pagar (lo cual es ilógico) cuando se podía llegar a la misma parada usando el transporte y sin hacer el esfuerzo de caminar. Esto pasa porque el algoritmo Dijkstra no considera algunas limitantes de los caminos en las ciudades, no porque no este no funcione de manera correcta.

4.2 Impacto

Al termino de las pruebas para corroborar el correcto funcionamiento del plan de viaje que en su totalidad fue satisfactorio, se comenzó a analizar su impacto que este proyecto puede llegar a tener a futuro.

El impacto que puede llegar a tener en la sociedad aun ahora así tal como funciona el proyecto, llegaría a beneficiar más que nada a las personas que no tengan conocimiento de la ciudad Guanajuato y deseen hacer uso del sistema de transporte urbano de la ciudad, como turistas o personas que no lleven mucho tiempo viviendo en la ciudad.

Se sabe de antemano que el algoritmo que se ha probado hasta este momento contiene seis rutas, pero llega a cubrir los puntos más importantes de la ciudad, y para los turistas les puede resultar muy útil porque las rutas usadas pasan por varios puntos de interés dentro de Guanajuato.

Guanajuato siendo una ciudad muy turística, si los turistas usan el plan de viaje implicaría un aumento en el uso de los autobuses y por consecuencia más ganancias para el sistema de transporte en general.

Ya que la mayoría de los turistas optan por contratar servicios de traslado en furgonetas a puntos turísticos y estos resultan demasiado costosos, es por eso que se propone esta alternativa para el turista que visite la ciudad de Guanajuato, le ayudará a ahorrar tiempo y dinero.

Capítulo 5

Conclusiones y recomendaciones

5.1 Conclusión

Se ha trabajado en este proyecto por más de seis meses, y con seguridad se puede decir que se ha cumplido con el propósito del proyecto, crear un plan de viaje funcional que permita ver a los usuarios seguir y la ruta óptima entre dos puntos aplicando el algoritmo de Dijkstra y la programación móvil.

Como se planteó en la introducción de este documento, las personas de la ciudad de Guanajuato hacen un gran uso del sistema de transporte para trasladarse por toda la ciudad. Entonces esta herramienta de plan de viaje es bastante útil para toda clase de personas que usan el transporte, tanto los habitantes, así como los turistas que visitan a diario la ciudad de Guanajuato.

Claro está que se le debe de dar una buena publicidad a esta herramienta para que pueda llegar a ser usada por todos los usuarios del transporte público.

Tanto la aplicación del monitoreo de unidades y el plan de viaje pueden llegar a dar competencia a las aplicaciones existentes que tienen funcionalidades parecidas, pero que no estan implementadas dentro de Guanajuato. Es por ello que se le debe dar la publicidad adecuada para que se haga conocida dentro de esta y más ciudades, implementando después en otras ciudades la herramienta del plan de viaje así tener una gran oferta no solo a nivel Guanajuato capital, si no a nivel nacional.

Los datos obtenidos a lo largo del proyecto poco a poco fueron ayudando para obtener este resultado final, un plan de viaje funcional, ya que se fue haciendo un desarrollo incremental, prestando atención a los tópicos que dejaban cosas a desear y mejorándolos, integrando nuevas funciones al proyecto que terminaron ayudando

a obtener un mejor resultado y no solo las que ya se habían considerado desde el principio.

Un ejemplo muy claro de esto es la automatización del grafo y de los archivos de las rutas cuando exista alguna actualización dentro de cualquiera de esos archivos. Al principio ni siquiera se tuvo en consideración que podían existir actualizaciones en los archivos de rutas y se solvento con esta automatización en las comunicaciones cliente – servidor, tanto locales como no locales.

Ahora bien, analizando el plan de viaje como un usuario más, es una herramienta que ayuda a ahorrar tiempo y dinero, dos cosas muy importantes. Es por eso que este proyecto aún tiene mucho margen de mejora, y siempre lo tendrá porque cada vez que se analizan las partes de este, pueden surgir detalles que antes no fueron vistos o alguna nueva funcionalidad que mejoraría por mucho al plan de viaje haciendo uso de herramientas que siempre han estado ahí tal como lo es la tecnología y fundamentos teóricos aplicados.

5.2 Recomendaciones y trabajo a futuro

Para futuras adiciones a este proyecto se recomienda primero que nada que se agreguen nuevas rutas que ayuden al usuario para tener más opciones y llegar a lugares que actualmente no se puede llegar, mejorando considerablemente el plan de viaje.

También un problema muy evidente al ingresar las rutas fue que se repetían paradas, pero en diferentes rutas. Esto se pudo corregir poco a poco haciendo coincidir las coordenadas de las paradas en un mismo punto, por eso se recomienda evitar asignarle diferentes coordenadas a una misma parada.

Así mismo se recomienda tener un estándar al momento de crear las rutas, este punto va muy de la mano con el anterior, ya que, si bien se pueden tener rutas casi iguales pero que cambian su trayecto en ciertos puntos para llegar a diferentes lugares, se debe tener mucho cuidado con las coordenadas de las paradas y hacerlas coincidir en un mismo punto, aunque sean de diferentes rutas.

Si esto no se logra en un futuro, al momento de que el algoritmo de Python genere el grafo, este grafo al tener varios nodos (paradas) que no coincidan en un mismo punto, creará muchas aristas y esto pone en riesgo al plan de viaje porque actuara de manera poco eficaz y es lo contrario de lo que se busca.

Por último, actualmente se sigue trabajando en la optimización de un caso en específico que arroja el plan de viaje, justamente se menciona en la sección 4.1.4, este caso se trata de que el plan de viaje al no considerar si hay algún camino disponible a pie, recomienda al usuario bajar el camión y tomar una parada “cercana” pero a la cual no hay un camino para llegar, además de que dicha parada ya pertenece a la ruta que se recomienda tomar, para estos casos, se deberán agregar unas condiciones en el post procesamiento de la ruta óptima, y así que el plan de viaje tenga sentido en cualquier caso posible.

Bibliografía

Consejo Nacional de Evaluación de la Política de Desarrollo Social. (2020). *Informe de pobreza y evaluación 2020. Guanajuato.* Obtenido de https://www.coneval.org.mx/coordinacion/entidades/Documents/Informes_de_pobreza_y_evaluacion_2020_Documentos/Informe_Guanajuato_2020.pdf

Diaz, A. I. (2012). *Calculadora Geoespacial*. Obtenido de http://sedici.unlp.edu.ar/handle/10915/125034

Google Developers. (2022). *Introducción a Android Studio*. Obtenido de https://developer.android.com/studio/intro?hl=es-419

Google Developers. (2022). *Maps SDK for Android*. Obtenido de https://developers.google.com/maps/documentation/android-sdk/cloud-setup?hl=es-419

INEGI. (2020). *División Municipal de Guanajuato*. Recuperado el 27 de Abril de 2022, de INEGI: https://cuentame.inegi.org.mx/monografias/informacion/gto/territorio/div_municipal.aspx?t

Insaurralde, N. (19 de Septiembre de 2022). *Google Maps: qué es, para qué sirve y cómo funciona*. Obtenido de https://www.mundocuentas.com/google/maps/

L. Platt, E. (2019). *Network Science with Python and NetworkX Quick Start Guide.* Reino Unido: Packt Publishing.

Meza H., O., & Ortega F., M. (2004). *Grafos y Algoritmos*. Recuperado el 12 de Agosto de 2022, de https://gecousb.com.ve/guias/GECO/Algoritmos%20y%20Estructuras%203

%20(CI-2613)/Material%20Te%C3%B3rico%20(CI-2613)/CI-2613%20Grafos%20y%20Algoritmos.pdf

Nolasco, J., & Atoche, W. (22 - 24 de Julio| de 2014). *Obtención del tiempo más corto para el problema de transporte en el sistema local de transporte público peruano 'El Metropolitano'*. Obtenido de http://www.laccei.org/LACCEI2014-Guayaquil/RefereedPapers/RP146.pdf

Robledano, A. (12 de Agosto de 2019). *Qué es Java: Principios básicos y evolución*. Obtenido de https://openwebinars.net/blog/que-es-java/

Stevens, W. (1998). *UNIX Network Programming* (Segunda ed., Vol. I). New Jersey: Prentice Hall.