Wazitico

Grevin Carrillo Rodríguez

Fabián Castillo Cerdas

Jorge Guillén Campos

Instituto Tecnológico de Costa Rica

Profesor Marco Rivera Meneses

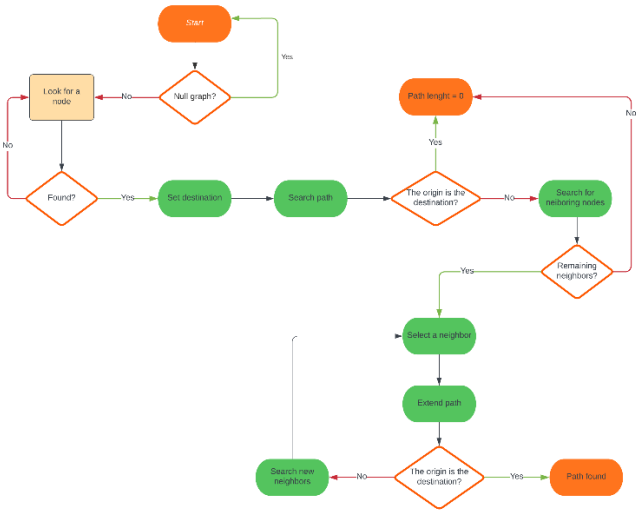
Paradigmas de programación

13 de septiembre de 2023

Índice

Descripción Detallada de los Algoritmos de Solución Desarrollados

El algoritmo de búsqueda implementado para la identificación de las rutas está basado en Djikstra, con la diferencia clara de que además de sugerir la ruta más corta también ofrece las demás rutas que conlleven a la misma ciudad de destino.



El algoritmo consiste en primeramente buscar en un grafo que no sea nulo, el nodo que haya sido seleccionado como el origen, de pertenecer al grafo entonces se corroborará que el nodo destino también se ubique en el grafo. Una vez que se corrobora la presencia de ambos se procede a buscar la ruta más corta trasladándose por medio de los nodos vecinos.

En primera instancia se verifica que el nodo origen y destino no sean el mismo porque automáticamente no habría camino y la distancia es 0, de otro modo se buscan todos los nodos a los que esté conectado el primero seleccionado. Si no posee vecinos automáticamente se cumpliría la anterior condición y sería distancia 0, sino seleccionaría uno de los vecinos para extender la ruta y seguir verificando hasta que se cumpla que el nodo actual es igual al destino y por tanto haber encontrado una ruta.

En consideración del tipo de programa y que la cantidad de nodos, es decir, de ciudades que van a ser ubicadas no es en exceso, el algoritmo de Djikstra o en este caso la variación realizada, consigue resultados certeros eficientemente. El tiempo que le toma en calcular las rutas es bastante rápido y no requiere de espera al tener una respuesta inmediata. Aunque se empleó recursión para determinar las rutas no afecta demasiado, nuevamente en consideración de que el entorno no pretende manejar un flujo de datos excesivamente grande.

Descripción de las Funciones Implementadas

(find_path start end graph): función principal encargada de recibir el nodo de origen y de destino, así como la totalidad del grafo. Esta función no realiza una operación directa sobre el grafo, sino que se extiende a una función auxiliar que realiza las operaciones respectivas para encontrar las rutas, en especial más corta.

(find_path_aux paths end graph): esta función auxiliar es llamada por la principal para reconocer los posibles caminos que conduzcan al destino descrito, por lo tanto, paths corresponde a los caminos, end es el nodo de destino y graph es el grafo donde se realiza la búsqueda. Si paths es nulo entonces no hay caminos, de no ser así entonces se revisará si el primer elemento del primer camino coincide con el destino y de serlo se retornará el reverso de ese primer camino para que los nodos estén ordenados desde la salida hasta el destino. En caso contrario, si no coinciden, entonces recursivamente se llamará a sí misma, pero añadiendo a paths los caminos extendidos a los que este nodo se encuentra conectado.

(find_all_paths start end graph): encuentra todos los caminos siempre que graph no esté vació, de estarlo entonces no hay caminos disponibles. Realiza un llamado recursivo a su función auxiliar con una lista de listas que incluye el nodo de inicio, además, se envía el nodo de destino, el grafo y una lista vacía para almacenar los caminos que vayan a ser encontrados.

(find_all_paths_aux paths end graph result): cuando paths esté vació se llama a reverse all para darle la vuelta a los caminos y que coincidan con la dirección del recorrido. Se revisa que si end ya es igual a uno de los nodos de paths entonces se llamara recursivamente omitiendo ese nodo y añadiéndolo a result porque este ya sería un camino encontrado, de otra forma se llamará recursivamente a sí misma pero extendiendo los caminos hasta lograr hallar otros caminos posibles.

(extend path graph): llama a su función auxiliar con los nodos vecinos de path, una lista vacía para almacenar el camino extendido y path, por lo tanto, su finalidad es extender un camino.

(extend_aux neighbors result path): opera hasta que neighbors esté vació, retornando los caminos extendidos. Se encarga de revisar si el primer elemento de neighbors está en path, de ser así entonces se omite y se llama recursivamente para revisar el resto de los vecinos.

Cuando un elemento de neighbors no está en path entonces es ingresado en result como parte de la extensión que se desea realizar.

(find_node node graph): revisa graph hasta encontrar el nodo, su condición de finalización es que graph esté vacío, en cuyo caso retornará una lista vacía, es decir, que el nodo no fue encontrado, o comparará node con el primer elemento de la primera lista de graph, en donde de ser iguales retornará la lista en la que se ubica el nodo buscado. Por otro lado, de no cumplirse ninguna de las dos condiciones anteriores habrá un llamado recursivo sobre la misma función, pero sin el primer elemento de graph para poder revisar si node se encuentra en alguna de las otras posiciones.

(neighbors node graph): se encarga de buscar los vecinos de un nodo en específico, llama a su función auxiliar con los elementos que acompañan al nodo y una lista vacía.

(neighbors_aux pairs result): cuando pairs esté vacío se retorna result con los nodos vecinos, en caso contrario se aplica recursión a sí misma, pero con el resto de los vecinos y guardando en result el primer vecino.

(path_distance path graph): calcula la distancia total de un camino que pertenece al grafo. Primero se revisa si el camino existe, de existir entonces se comprueba si la longitud de path es menor o igual a uno ya que eso representaría que no hay distancia entre nodos. Finalmente, y como última condición, se va sumando las distancias entre los nodos que se van recorriendo mientras que recursivamente se va cambiando de nodo hasta llegar al de destino para así obtener la distancia total.

(nodes_distance start end graph): se encarga de calcular la distancia entre dos nodos, primero se revisa si el nodo de destino es vecino del nodo de inicio, de serlo se llama a su función auxiliar con el nodo de destino y las conexiones del nodo de inicio. Si lo anterior no se cumple no hay conexión entre los nodos.

(nodes_distance_aux end neighbors): verifica el punto en el que el nodo de destino es igual a uno de los vecinos, en el momento que ocurra se retorna la distancia entre esos nodos, si no hay coincidencia directa entonces se va comparando end con los otros vecinos.

(reverse_all lst): recibe una lista y llama una función auxiliar con dos elementos, la lista ingresada y una lista vacía con la finalidad de emplearla para invertir el orden de los elementos de la lista.

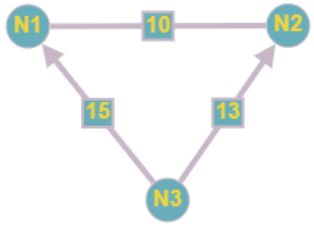
(reverse_all_aux lst result): revisa cuando lst ya fue completamente trasladada a la otra lista y devuelve el inverso de la lista, en caso contrario se hace un llamado recursivo ingresando el primer elemento de lst en result hasta que lst esté vacía.

Descripción de la Ejemplificación de las Estructuras de Datos Desarrolladas

La estructura de datos construida para la elaboración del mapa es un grafo. La forma en la que es estructuró es:

‘((N1 (N2 10))
(N2 (N1 10))
(N3 (N1 15))
(N3 (N2 13))
)

Una ejemplificación visual del anterior grafo sería:



Por lo tanto, la estructuración del grafo es a través de una lista de listas, cada sublista contiene el nodo, así como una lista que posee el nodo al que estará conectado y el peso. De forma contextualiza cada nodo corresponde a una ciudad, mientras que la sublista que lo acompañada contiene el nombre de la ciudad al que está conectado y la distancia que las separa.

Problemas Encontrados

Durante el desarrollo de este proyecto, se encontraron varios problemas o bugs que afectaron el funcionamiento del programa. A continuación, se detallarán algunos de estos problemas concretos que se presentaron durante el desarrollo, así como las soluciones específicas que se implementaron para resolverlos. Además, se proporcionarán recomendaciones y conclusiones basadas en la experiencia adquirida al abordar estos problemas.

Problema: Error al Agregar una Nueva Ciudad

> **Descripción**: Al intentar agregar una nueva ciudad mediante la interfaz gráfica, se encontró un error que permitía agregar una ciudad sin proporcionar un nombre, una latitud o una longitud. Esto llevaba a la creación de nodos en el grafo con información faltante o incorrecta.
>
> **Solución**: Se implementó una validación en la interfaz de usuario para verificar que todos los campos de entrada estuvieran completos antes de agregar una nueva ciudad. Además, se agregaron mensajes de error informativos para guiar al usuario a completar todos los campos requeridos.

Problema: Error en la creación de nodos duplicados.

> **Descripción**: En la aplicación, era posible crear nodos con el mismo nombre, lo que llevaba a la duplicación de ciudades en el grafo, lo cual no debería permitirse.
>
> **Solución**: Se implementó una función llamada '**hasNode?,** que verifica si un nodo ya existe en el grafo antes de agregarlo. Cuando se intenta agregar un nuevo nodo con un nombre existente, la función '**addNode'** ahora verifica primero si el nodo ya está presente en el grafo y evita su duplicación

Problema: Error al borrar todos los nodos y rutas

> **Descripción**: Al presionar el botón "Clear Map", se pretendía borrar todos los nodos y rutas en el lienzo, pero había un problema en la función que manejaba esta acción.
>
> **Solución**: Se implementó la función '**clearAllNodes'** para borrar todos los nodos y rutas en el lienzo. La función limpia la lista de coordenadas (**coords-list**), borra el lienzo (**dc clear**), y finalmente, llama a '**dc flush'** para actualizar la visualización.

Problema: Error en el cálculo de la distancia mínima

> **Descripción**: El algoritmo para encontrar la distancia mínima entre dos ciudades no estaba funcionando correctamente y devolvía resultados incorrectos.

Solución: Se revisó la función '**min'** utilizada para encontrar la distancia mínima entre dos rutas. Se corrigieron los errores de comparación y cálculo, asegurando que la distancia mínima se calculara correctamente.

Problema: Dibujar Rutas Incorrectas

Descripción: En algunas situaciones, la aplicación dibujaba rutas incorrectas entre las ciudades, lo que generaba confusión en los usuarios al visualizar las conexiones entre las ciudades.

Solución: Se identificó que el problema se debía a errores en el cálculo de rutas en el grafo subyacente. Se revisaron las funciones encargadas de encontrar rutas y se corrigieron los errores en los algoritmos. También se mejoró la representación gráfica de las rutas en el lienzo.

Problema: Búsqueda de Rutas Incorrectas

Descripción: En ciertos escenarios, la aplicación no encontraba la ruta correcta entre dos ciudades, incluso cuando debería existir una conexión válida en el grafo.

Solución: Después de un análisis exhaustivo, se descubrió que este problema se debía a la presencia de aristas dirigidas que no se estaban gestionando correctamente al buscar rutas. Se ajustó el algoritmo de búsqueda de rutas para tener en cuenta las aristas dirigidas y garantizar que se encontraran rutas válidas en todas las situaciones.

Problema: Rutas No Actualizadas Después de Modificaciones en el Grafo

Descripción: Después de agregar o eliminar ciudades o rutas en el grafo, las rutas previamente calculadas no se actualizaban automáticamente, lo que resultaba en rutas obsoletas en la interfaz.

Solución: Se implementó una función de "actualización de rutas" que detectaba cambios en el grafo y recalculaba las rutas automáticamente cuando se realizaban modificaciones en el grafo. Esto garantizaba que las rutas reflejaran siempre la estructura actual del grafo.

Problema: Dificultades en la Depuración de Algoritmos de Búsqueda

Descripción: Durante el desarrollo de la búsqueda de rutas, se encontraron dificultades para depurar los algoritmos de búsqueda, especialmente en grafo con una gran cantidad de ciudades y conexiones.

Solución: Se implementó un sistema de registro detallado para los algoritmos de búsqueda, lo que permitió rastrear y registrar los pasos de cada búsqueda. Además, se agregaron puntos de control en el código para inspeccionar el estado del grafo y los datos de búsqueda en tiempo real. Estos registros y puntos de control facilitaron la depuración de los algoritmos y la identificación precisa de problemas en la lógica de búsqueda.

Problemas sin Solución Encontrados

En esta sección, se describen los problemas encontrados en el proyecto que aún no han sido resueltos satisfactoriamente. Cada problema se presenta con una descripción detallada, los intentos de solución realizados hasta el momento, las soluciones que se han encontrado y se detallan las recomendaciones, conclusiones y bibliografía consultada para abordar estos problemas específicos. A continuación, se presentan algunos de los problemas sin solución encontrados en el proyecto:

Problema: Creación de Rutas con Nodos Inexistentes

Descripción: Actualmente, la aplicación permite la creación de rutas entre ciudades incluso si uno o ambos nodos no existen en el grafo. Esto puede llevar a resultados incorrectos y rutas que no tienen sentido.

Soluciones Intentadas: Se han realizado intentos de implementar una validación más estricta al agregar rutas, incluyendo la verificación de la existencia de los nodos involucrados. Sin embargo, esta validación ha demostrado ser compleja debido a la interacción con otros componentes de la aplicación.

Recomendaciones: Se recomienda revisar y refactorizar la lógica de creación de rutas para garantizar que solo se puedan crear rutas entre ciudades que realmente existen en el grafo. Esto podría requerir una revisión más profunda de la arquitectura de la aplicación.

Conclusión: La validación de la creación de rutas entre ciudades es crucial para mantener la integridad del grafo y evitar resultados incorrectos. Se necesita una revisión profunda de la lógica de creación de rutas para garantizar que solo se puedan crear rutas entre ciudades existentes en el grafo.

Referencias Bibliográficas:

- Sitio web oficial de Racket. (https://racket-lang.org/)
- Stelly, J. W. (Año de publicación). *Racket Programming the Fun Way*. Editorial.
- Butterick, M. (Año de publicación). *Beautiful Racket*. Editorial.

Problema: Elementos que se Sobreponen en la Interfaz Gráfica

Descripción: En casos de ciudades con nombres largos o rutas cercanas entre sí, los elementos en la interfaz gráfica, como nombres de ciudades o distancias, pueden superponerse, lo que dificulta la legibilidad y la interacción del usuario.

Soluciones Intentadas: Se han intentado diferentes enfoques de diseño y disposición de elementos en la interfaz gráfica para evitar la superposición. Sin embargo, estos enfoques no han resuelto completamente el problema en todos los casos.

Recomendaciones: Se recomienda explorar opciones de diseño y presentación más avanzadas para manejar de manera efectiva la superposición de elementos en la interfaz gráfica. Esto podría incluir la implementación de herramientas de zoom o de ajuste automático de elementos.

Conclusión: La superposición de elementos en la interfaz gráfica puede afectar negativamente la usabilidad de la aplicación. Se necesita una investigación más profunda sobre técnicas avanzadas de diseño de interfaces para abordar este problema de manera efectiva.

Referencias Bibliográficas:

- Sitio web oficial de Racket. (https://racket-lang.org/)
- Stelly, J. W. (Año de publicación). *Racket Programming the Fun Way*. Editorial.
- Butterick, M. (Año de publicación). *Beautiful Racket*. Editorial.

Problema: Manejo de Búsquedas de Rutas Inválidas

Descripción: Cuando un usuario intenta buscar una ruta entre dos ciudades que no están conectadas en el grafo, o que simplemente no existen en el grafo, la aplicación actualmente genera un error que bloquea el programa en lugar de proporcionar una respuesta amigable al usuario.

Soluciones Intentadas: Se han realizado esfuerzos para implementar una validación más robusta al buscar rutas, incluyendo la verificación de la existencia de los nodos y su conexión en el grafo. Sin embargo, estas soluciones aún no han resuelto completamente el problema de manera satisfactoria.

Recomendaciones: Se recomienda revisar la lógica de búsqueda de rutas y la gestión de errores para proporcionar un mensaje claro y amigable al usuario cuando se encuentre una ruta inválida o inexistente. Además, se debe garantizar que el programa no se bloquee por completo en caso de errores.

Conclusión: La gestión de errores y la respuesta amigable al usuario son esenciales al realizar búsquedas de rutas en un grafo. Se debe mejorar la validación y proporcionar mensajes claros para evitar bloqueos del programa.

Referencias Bibliográficas:

- Sitio web oficial de Racket. (https://racket-lang.org/)
- Stelly, J. W. (Año de publicación). *Racket Programming the Fun Way*. Editorial.
- Butterick, M. (Año de publicación). *Beautiful Racket*. Editorial.

Problema: Implementación de la Selección de Idioma

Descripción: La aplicación no ha implementado una funcionalidad de selección de idioma, lo que limita su accesibilidad y utilidad para usuarios que prefieren idiomas diferentes al inglés.

Soluciones Intentadas: Se ha considerado la posibilidad de implementar un sistema de selección de idioma, pero aún no se ha encontrado una solución adecuada que sea compatible con todos los componentes de la aplicación.

Recomendaciones: Se recomienda investigar y utilizar bibliotecas o frameworks que simplifiquen la implementación de la selección de idioma y aseguren su coherencia en toda la aplicación. También se puede considerar la colaboración con expertos en internacionalización de software.

Conclusión: La falta de una funcionalidad de selección de idioma limita la accesibilidad de la aplicación a usuarios de diferentes regiones y preferencias lingüísticas. Se necesita investigar soluciones específicas de internacionalización y localización.

Referencias Bibliográficas:

- Sitio web oficial de Racket. (https://racket-lang.org/)
- Stelly, J. W. (Año de publicación). *Racket Programming the Fun Way*. Editorial.
- Butterick, M. (Año de publicación). *Beautiful Racket*. Editorial.

Plan de Actividades

Actividad	Encargado			Fecha
	Greivin	Fabian	Jorge	
Dividir el proyecto	x	x	x	25-ago
Definir las estructuras de datos	x	x	x	
Investigar sobre el algoritmos de busqueda	x			28-ago
Investigar sobre la biblioteca grafica			x	
Crear la base del algoritmo	x			30-ago
Creacion de la base de la interfaz			x	31-ago
Buscar problemas de compatibilidad	x	x		1-sep
Implementar la solucion final		x		4-sep
Detallar la interfaz grafica		x	x	6-sep
Hacer el manual de usuario		x		8-sep
Hacer la documentacion	x			
Revision final	x	x	x	12-sep

Conclusiones

Durante el desarrollo de este proyecto, se pudo consolidar y profundizar en la comprensión del paradigma de programación funcional. La aplicación en Racket sirvió como un entorno de aprendizaje efectivo para aplicar conceptos clave, como funciones de orden superior, recursión, y manipulación de listas. Esta experiencia reforzó la importancia de la programación funcional en la resolución de problemas complejos.

La creación de la aplicación en Racket significó un desafío emocionante. A lo largo del proyecto, se adquirió un dominio más sólido de este lenguaje, incluyendo su sintaxis única y la utilización efectiva de sus características funcionales. Esta experiencia enriquecedora demostró la versatilidad de Racket en la implementación de algoritmos y la manipulación de estructuras de datos.

Uno de los aspectos más destacados del proyecto fue la aplicación exitosa de la programación funcional para resolver un problema real. La capacidad de diseñar un sistema de rutas y encontrar la ruta más corta entre dos puntos dentro de una ciudad simulada resalta la eficacia de este enfoque de programación en la solución de desafíos prácticos.

La creación y manipulación de listas como estructuras de datos fundamentales fue un componente esencial de la aplicación. Este proyecto subrayó la importancia de las listas en la programación funcional y cómo pueden ser utilizadas de manera eficiente para representar información compleja.

La implementación de una interfaz gráfica para representar el mapa de la ciudad y las rutas resaltó la versatilidad de Racket en la creación de aplicaciones interactivas. La capacidad de visualizar de manera efectiva las rutas y mostrar información relevante, como la distancia entre puntos, es fundamental en aplicaciones de navegación como esta.

Recomendaciones

1. Mejoras en la Interfaz de Usuario: Se sugiere la incorporación de características adicionales para mejorar la experiencia del usuario. Esto incluye la posibilidad de seleccionar entre diferentes mapas o ciudades, la capacidad de agregar íconos personalizados para representar lugares de interés y la inclusión de una leyenda para facilitar la interpretación del mapa.
2. Optimización de Algoritmos: A medida que el proyecto continúe creciendo y manejando conjuntos de datos más grandes, se recomienda la optimización de los algoritmos utilizados para encontrar rutas. Esto podría incluir la implementación de algoritmos más eficientes para encontrar la ruta más corta, lo que mejoraría el rendimiento general de la aplicación.
3. Implementación de Funcionalidades Adicionales: Para hacer que la aplicación sea aún más completa, se podría considerar la implementación de características adicionales, como la capacidad de calcular estimaciones de tiempo de llegada, ofrecer múltiples rutas alternativas y proporcionar información sobre el tráfico en tiempo real.
4. Localización y Traducción: Para llegar a un público más amplio, se podría considerar la localización de la aplicación a múltiples idiomas. Esto requeriría la traducción de la interfaz de usuario y la adaptación de los datos geográficos a diferentes regiones.
5. Pruebas Extensivas: Antes de cualquier lanzamiento importante, es esencial realizar pruebas exhaustivas para identificar y corregir posibles problemas. Esto incluye pruebas de usabilidad, pruebas de carga y pruebas de seguridad.
6. Colaboración y Contribuciones: Fomentar la colaboración con la comunidad de desarrolladores de Racket podría llevar a mejoras adicionales en el proyecto. La recepción de contribuciones externas y la participación en comunidades en línea pueden enriquecer la aplicación.

Bibliografía

- Dybvig, R. K., Hieb, R., & Bruggeman, C. (1993). Syntactic abstraction in Scheme. Lisp and Symbolic Computation, 5(4), 295-326.

- Findler, R. B., Clements, J., Flatt, M., Krishnamurthi, S., Steckler, P., & Felleisen, M. (2002). DrScheme: A programming environment for Scheme. Journal of Functional Programming, 12(2), 159-182.

- Felleisen, M., Findler, R. B., Flatt, M., Krishnamurthi, S., & Steckler, P. (2009). How to design programs: An introduction to programming and computing. MIT Press.

- Kent Dybvig. (2020). Racket: a programmable programming language. URL: https://racket-lang.org/

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3rd ed.). The MIT Press.

- SICP (Structure and Interpretation of Computer Programs). (n.d.). MIT OpenCourseWare. URL: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-001-structure-and-interpretation-of-computer-programs-spring-2005/index.htm

- Racket Documentation. (n.d.). Racket Documentation. URL: https://docs.racket-lang.org/

- Racket Package Index. (n.d.). Racket Package Index. URL: https://pkgs.racket-lang.org/

- McConnell, S. (2004). Code complete: A practical handbook of software construction (2nd ed.). Microsoft Press.

- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design patterns: Elements of reusable object-oriented software. Addison-Wesley.

Bitácora Digital

Greivin:

- Se hizo una reunión para dividir las partes del trabajo y para declarar como ir trabajando. (25 agosto)
- Investigue sobre algoritmos de búsqueda. (28 agosto)
- Cree un plan de actividades a realizar para el proyecto. (28 agosto)
- Cree un código base del algoritmo de búsqueda por anchura. (30 agosto)
- Cree una función para que utiliza listas para la creación de nodos y a su vez grafos. (30 agosto)
- Cree una función auxiliar para obtener una lista inversa para solucionar un bug en la creación de los nodos del grafo. (2 septiembre)
- Trabaje en la documentación. (8 septiembre)
- Se hizo una reunión para finalizar detalles y revisar el código final. (12 septiembre)

Fabian:

- Se realizo una reunión inicial para definir la metodología del trabajo. (25 agosto)
- Modifique la interfaz para que comprenda los datos enviados y dibuje el grafo. (1 septiembre)
- Implemente una nueva ventana para la obtención de los nodos desde la interfaz. (1 septiembre)
- Implemente en la ventana de dibujo una sección que muestre los pesos de la ruta más corta. (1 septiembre)
- Arregle los bugs de la creación de nodos y sus conexiones con nodos inexistentes. (6 septiembre)
- Realice la sección de recomendaciones y conclusiones para la documentación. (10 septiembre)
- Realice el manual de usuario. (8 septiembre)
- Se realizo una reunión para finalizar el código y concluir con la solución del proyecto. (12 septiembre)

Jorge:

- Nos reunimos para dividir el trabajo y como trabajarlo. (25 agosto)
- Investigue sobre la biblioteca grafica de racquet. (28-29 agosto)
- Hice una interfaz simple usando la biblioteca de racquet que se usara como menú. (31 agosto)
- Hice que la interfaz dibuje dependiendo de los datos que se le den al programa, ya sean círculos, líneas o etiquetas para definir los nodos más adelante. (31 agosto)
- Definí la creación de ventanas mediante botones, junto con la actualización de datos y cambio en los dibujos. (31 agosto)

- Trabaje en la modificación del grafo al aumentarlo y su visualización en la interfaz. (1 septiembre)
- Hice la parte de los algoritmos y estructuras de bases de la documentación. (8 septiembre)
- Nos reunimos para revisar el código. (12 septiembre)