TEC | Tecnológico de Costa Rica

LENGUAJES, COMPILADORES E INTÉRPRETES

Profesor: Marco Rivera Meneses

Erick Josué Barrantes Cerdas - 2017125821

María José Zamora Vargas - 2018

José Daniel Acuña Solano - 2018

Tarea Corta #1
Grupo 02

Contenidos

Definición general del proyecto

El presente proyecto de software tiene como fin el desarrollo de un programa que simule la famosa aplicación Waze. Este programa brinda la opción al usuario de crear una nueva ciudad donde puede agregar nuevos lugares y rutas. Esta ciudad está desarrollada como un grafo mixto con peso, donde las aristas representan las vías entre lugares y su sentido, y el peso representa la distancia entre los diferentes puntos del mapa.

El sistema proveerá al usuario la opción de escoger un lugar de origen y un lugar de destino, para que pueda calcular y mostrar todas las rutas posibles entre ambos puntos, mostrando la distancia total para cada ruta.

Este sistema se desea implementar para cumplir los siguientes objetivos:

- Crear una aplicación que resuelva el problema descrito haciendo uso de *DrRacket*.
- Aplicar conceptos de la programación funcional.
- Crear y manipular listas como estructuras de datos.

El presente informe está dirigido para personas con experiencia en el área de programación, con conocimientos sobre el paradigma funcional y su aplicación con *DrRacket*, recursividad de pila y cola, además del uso de grafos como estructuras de datos.

Descripción de la estructura de datos desarrollada

Para este proyecto programado se requiere simular diferentes ciudades con varios lugares que se conectan entre sí por medio de calles. Para esto, se implementó un grafo como estructura de datos para realizar los cálculos que solicite el usuario sobre la ciudad.

En este grafo, cada nodo representará un lugar dentro de la ciudad y la conexión entre estos nodos (aristas) son las calles. Estas aristas les dan el sentido a las calles y además, se hace el uso de un peso en estas conexiones para representar la distancia entre dos puntos.

A continuación, se presenta un grafo general creado con listas de Racket con n cantidad de nodos, donde no existe ninguna arista entre los nodos:

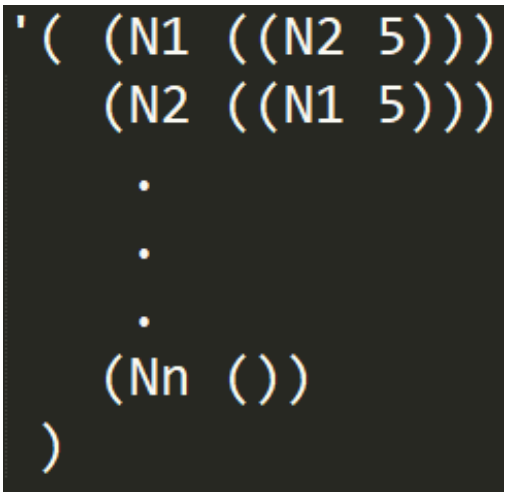

Imagen 6. Implementación de grafo con N cantidad de nodos y sin aristas.

Como se puede observar, el grafo es una lista de Racket, donde cada elemento es a su vez, una lista que representa a cada nodo. El primer elemento de los nodos hace referencia al nombre del nodo mientras que el segundo, por el momento, es una lista vacía.

En esa lista vacía van a ser ingresadas las conexiones entre los nodos. Por ejemplo, este sería el resultado si se desea conectar el nodo N1 con el nodo N2, para luego hacer la misma conexión, pero con sentido contrario, ambas con un peso de 5:

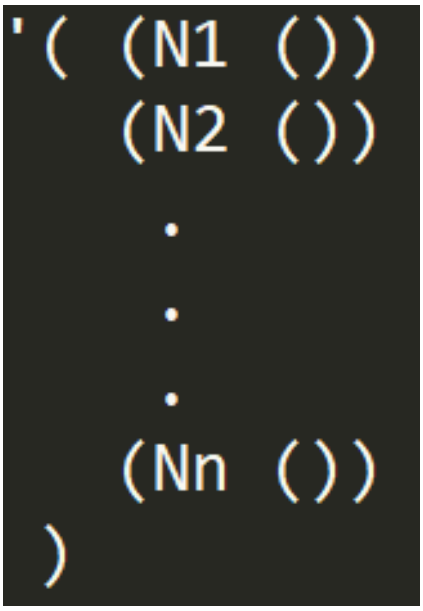

Imagen 7. Implementación de grafo de n nodos y con dos conexiones.

En las listas que anteriormente se encontraban vacías, se encuentra una sublista con un par de elementos, donde el primer elemento es el nodo con el que está conectado y el segundo elemento es el peso de esa conexión. De igual manera, las demás conexiones que se crean serán representadas de la misma manera.

En este proyecto programado se tiene un grafo que representa la ciudad que el usuario va a crear. Esta estructura se encuentra vacía inicialmente, pero con las acciones que se le brindan al usuario, se irán insertando nuevos nodos y aristas con las funciones que generan el grafo, descritas anteriormente. Este mismo grafo se utiliza para la parte gráfica y para las funciones de búsqueda de caminos.

Descripción de las funciones implementadas

En esta sección del documento se va a dar una detallada descripción de las funciones implementadas. Las funciones serán separadas en dos grupos: funciones para crear un grafo y funciones para lograr encontrar todos los caminos posibles entre dos puntos del grafo.

Funciones Pathfinding

Estas son todas las funciones necesarias y que fueron utilizadas para el módulo de pathfinding, el cual es el encargado de manejar un grafo con el formato establecido, para devolver una lista con el camino entre dos puntos, si este existe.

Función *find_path*

```
(define (find_path start end graph)
  (find_path_aux (list (list start)) end graph)
)

(define (find_path_aux paths end graph)
  (cond ((null? paths) '())
        ((equal? end (caar paths)) (reverse (car paths)))
        (else (find_path_aux (append
                              (extend (car paths) graph)
                              (cdr paths))
                             end
                             graph))
   )
)
```

Imagen 1. Código implementado para find_path

La función *find_path* utiliza el algoritmo de Búsqueda por Profundidad, el cuál será explicado más adelante. Esta función recibe un punto de origen y un punto de destino, además

del grafo. Su objetivo es encontrar un solo camino entre ambos nodos. Esta función se desempeña con mayor eficiencia en grafos sin muchas interconexiones, aun cuando son nodos muy lejanos.

El camino de resultado es una lista con n cantidad de elementos, donde el primer elemento es el nodo de origen y el último elemento es el nodo de destino, entre estos se encuentran en orden todos los nodos que completan el camino. Por ejemplo, la forma general del resultado de esta función es: (O N1 N2 … Nn D)

Donde O es el origen, N1 hasta Nn son los nodos del camino encontrado y D es el destino.

Función *find_all_paths*

```
1  (define (find_all_paths start end graph)
2    (find_all_paths_aux (list (list start)) end graph '())
3  )
4
5  (define (find_all_paths_aux paths end graph result)
6    (cond ((null? paths) (reverse_all result))
7          ((equal? end (caar paths)) (find_all_paths_aux (cdr paths)
8                                                         end
9                                                         graph
10                                                        (cons (car paths)
                                                          result)) )
11         (else (find_all_paths_aux (append
12                                    (extend (car paths) graph)
13                                    (cdr paths))
14                                   end
15                                   graph
16                                   result))
```

Imagen 2. Código implementado para find_all_paths

La función *find_all_paths* implementa el algoritmo de búsqueda por anchura, el cuál de igual manera será explicado más adelante. Esta es la función encargada de encontrar todos los caminos posibles entre un punto de origen y un punto de destino especificados como parámetros dentro del grafo. Esta función es eficiente inclusive en grafos con muchas conexiones.

El resultado de esta función será una lista con n cantidad de elementos, donde cada elemento es a su vez una sublista que cumple con el formato del resultado de *find_path*. La forma general del resultado de esta función es:

((O N1 N2 … Nn D) (O M1 M2 … Mn D) …).

Donde O es el origen, D es el destino y tanto N como M son nodos que representan diferentes caminos.

Función *extend*

```
(define (extend path graph)
  (extend_aux (neighbors (car path) graph) '() path)
)

(define (extend_aux neighbors result path)
  (cond ((null? neighbors) result)
        ((member (car neighbors) path) (extend_aux (cdr neighbors)
        result path ))
        (else (extend_aux (cdr neighbors)
                          (append result (list(list* (car neighbors)
                          path)))
                          path ))
   )
)
```

Imagen 3. Código implementado para función extend

La función *extend* recibe como entrada un camino y un grafo, donde el camino debe de existir dentro de ese grafo. Su objetivo es extender las rutas a otras posibles a partir del camino dado. Por esto, es utilizada en la función *find_all_paths*.

Función find_node

```
(define (find_node node graph)
  (cond ((null? graph) '())
        ((equal? node (caar graph)) (car graph))
        (else (find_node node (cdr graph)))
   )
)
```

Imagen 4. Código implementado para la función find_node

Esta es una función muy simple que busca si el nodo ingresado como parámetro existe dentro del grafo. Si esto es verdadero, retorna el nodo encontrado, en caso contrario retorna una lista vacía.

Función *neighbors*

```
(define (neighbors node graph)
  (neighbors_aux (last (find_node node graph)) '())
)

(define (neighbors_aux pairs result)
  (cond ((null? pairs) result)
        (else  (neighbors_aux (cdr pairs) (append result (list(caar
        pairs)) )))
  )
)
```

Imagen 5. Código implementado para la función neighbors

Esta función recibe un nodo y el grafo al cual pertenece. Va a dar como resultado una lista con los nodos que están conectados al nodo de entrada. Esta función es utilizada en la función *extend*.

Funciones GraphCreator

En este módulo se encuentran las funciones para poder crear un grafo con el formato que se fue establecido. Se explicarán detalladamente todas las funciones para lograr este objetivo.

Función hasNode?

```
1 (define (hasNode? node graph)
2   (cond ((null? graph)
3          #f)
4         ((equal? node (caar graph))
5          #t)
6         (else
7          (hasNode? node (cdr graph)))))
```

Imagen 6. Código implementado para la función hasNode?

Esta es una función de utilidad para otras funciones de este módulo. Recibe el nombre de un grafo y se encarga de verificar si ese nodo pertenece al grafo que también se recibe como parámetro. Si esto se cumple, retorna un #t que es un valor booleano indicando que este caso verdadero, de lo contrario, devuelve #f indicando que es falso.

Función addNode

```
1 (define (addNode newNode graph)
2   (cond ((not(hasNode? newNode graph))
3          (append graph (list (list newNode '()))))
4         (else
5          graph)))
```

Imagen 7. Código implementado para función addNode.

Esta función recibe el nombre de un nuevo nodo que desea agregar al final del grafo ingresado. Primero, debe verificar con la función *hasNode?* si el nodo no existe en el grafo. Si esto se cumple, devuelve la unión del grafo con el nuevo nodo con el siguiente formato: (…(newNode ())).

Si el nodo ya existe dentro del grafo, la función devuelve el grafo que fue recibido sin modificación alguna.

Función addAllNodes

```
1 (define (addAllNodes nodeList graph)
2   (cond ((null? nodeList)
3          graph)
4         (else
5          (addAllNodes (cdr nodeList) (addNode (car nodeList) graph)))))
```

Imagen 8. Código implementado para función addAllNodes.

Esta función hace uso de *addNode*, solo que aplica recursividad para agregar al grafo todos los nodos que se encuentran dentro de una lista ingresada por parámetro. El formato de esa lista es: (N1 N2 N3 … Nn)

Función hasEdge?

```
1 (define (hasEdge? newNode edgesList)
2   (cond ((null? edgesList)
3          #f)
4         ((equal? newNode (caar edgesList))
5          #t)
6         (else
7          (hasEdge? newNode (cdr edgesList)))))
```

Imagen 9. Código implementado para la función hasEdge.

Esta es una función de utilidad para la inserción de aristas en el grafo. Recibe el nombre de un nodo y una lista de aristas de un nodo del grafo. Esta función se encarga de indicar si el nodo ingresado ya se encuentra en la lista de aristas. La lista de aristas cumple con el formato: ((Nx px) (Ny py) …), donde Nx y Ny son dos nodos cualesquiera del grafo, junto con sus respectivos pesos px y py. Esta lista puede tener cualquier cantidad de elementos.

Si el nodo existe dentro de alguna de las aristas, retorna un verdadero. De lo contario, retorna un falso.

Función addEdge

```
(define (addEdge originNode endNode weight isDirected? graph)
  (cond ((and (hasNode? originNode graph) (hasNode? endNode graph))
         (cond (isDirected?
                (addEdgeAux originNode endNode weight graph))
               (else
                (addEdgeAux endNode originNode weight (addEdgeAux originNode endNode weight graph)))))
        (else
         '())))


(define (addEdgeAux originNode endNode weight tempGraph)
  (cond ((null? tempGraph)
         '())
        ((equal? originNode (caar tempGraph))
         (cond ((not(hasEdge? endNode (cadar tempGraph)))
                (cond ((equal? (cadar tempGraph) '())
                       (append (list(list originNode (list (list endNode weight))))
                               (cdr tempGraph)))
                      (else
                       (append (list(list originNode (append (list (list endNode weight)) (cadar tempGraph))))
                       (cdr tempGraph)))))
               (else
                (append tempGraph (addEdgeAux originNode endNode weight '()))))
        (else
         (append (list(car tempGraph)) (addEdgeAux originNode endNode weight (cdr tempGraph))))))
```

Imagen 10. Código implementado para la función addEdge.

Esta función está encargada de la inserción de aristas de un nodo a otro dentro del grafo. Recibe los siguientes argumentos:

- Nodo de origen.
- Nodo de destino.
- Peso de la arista.
- Booleano indicando si es dirigido o no.
- El grafo donde se desea crear la arista.

Su tamaño es notablemente más grande a las demás funciones de este módulo porque tiene que cumplir con numerosas validaciones, las cuales son:

- Ambos nodos deben de existir dentro del grafo

- Debe de verificar si la arista será dirigida. Si es así, solo hace la conexión con el sentido indicado por los argumentos. En caso de que no sea dirigido, además deberá crear una conexión con el sentido contrario.
- La nueva arista será creada si y solo si una arista del nodo de origen con el mismo nodo de destino no existe.

Esta función dará como resultado el grafo con la nueva arista creada, si se cumplen todas las condiciones descritas. Sigue manteniendo siempre el formato establecido para esta estructura.

Descripción detallada de los algoritmos desarrollados

Se utilizan dos diferentes algoritmos sobre los grafos.

Búsqueda por profundidad

Con este algoritmo se avanzada en el grafo nodo por nodo, marcando cada nodo visitado. La búsqueda siempre avanza hacia un nodo no marcado, internándose "profundamente" en el grafo sin repetir ningún nodo. Cuando se alcanza un nodo cuyos vecinos han sido marcados, se retrocede al anterior nodo visitado y se avanza desde éste. Es decir, hace uso de Backtracking.

Este algoritmo sigue los siguientes pasos:

1. Se comienza en un nodo, el cual sería el nodo actual temporalmente y es marcado como visitado.
2. Se escoge un nodo que sea vecino sin visitar del nodo actual. Si ese nodo no existe (no tiene más vecinos sin visitar), se pasa al paso 4.
3. Se marca el nuevo nodo como visitado y este será nuestro nuevo nodo actual. Luego de esto se va al paso 2.
4. Si todos los nodos del grafo ya han sido visitados, se termina el algoritmo. De lo contrario, se toma el nodo padre del nodo actual, este se marca como el nuevo nodo actual y se vuelve al paso 2.

Por ejemplo, si se tiene el grafo de la Imagen 8 y se desea hacer un recorrido por profundidad desde el nodo D, su resultado sería (D C R H T A B). Esto porque se viaja a los vecinos desde D hasta llegar a T. Cuando se determina que este no tiene vecinos, se viaja al padre de él (H) y se repite el proceso, obteniendo por último a los nodos A y B.

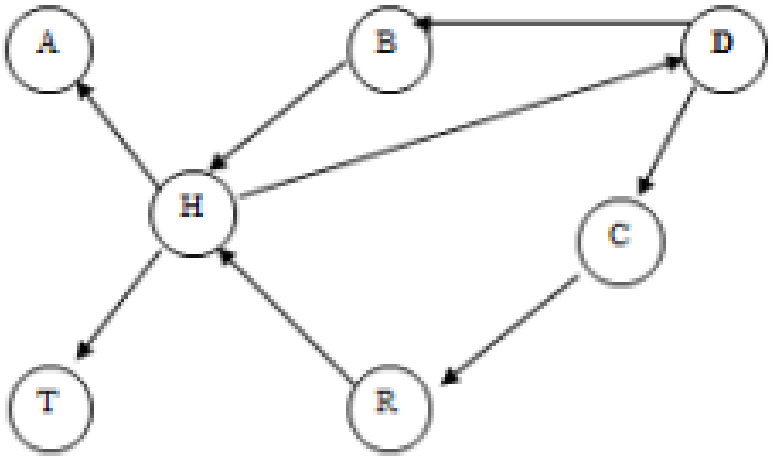

Imagen 11. Grafo de ejemplo para algoritmos implementados.

Búsqueda por anchura

Este es otro algoritmo para visitar todos los nodos de un grafo. Al igual que el algoritmo anterior, se comienza en un nodo que se marca como visitado y como el primer nodo actual. Donde se diferencian ambos algoritmos, es que la búsqueda por anchura marca como visitados todos los vecinos del nodo actual que aún no hayan sido marcados. Se continua este proceso donde nunca se visita un mismo nodo dos veces, hasta que todos los nodos hayan sido visitados.

Por ejemplo, haciendo uso del mismo grafo de la Imagen 8, empezando por D, el resultado de la búsqueda de anchura sería (D B C H R A T).

Problemas conocidos

Hasta el presente día de desarrollo, el equipo de trabajo no ha encontrado un problema conocido sin solución luego de múltiples pruebas a toda la funcionalidad del programa. Sin embargo, esto no significa que no existan posibles errores no contemplados. En caso de que esto suceda, los errores serán reportados en esta sección.

Para errores encontrados durante el desarrollo, vea la siguiente sección titulada **Problemas encontrados** para ver la solución de estos.

Problemas Encontrados

En esta sección se van a describir los problemas que pudieron ser resueltos durante el desarrollo del proyecto.

Errores en implementación de lógica

- A la hora de obtener el peso total de una ruta, no se contempló si esa ruta era vacía. Su solución fue validar este caso y retornar un -1 para indicar un error.
- En función *find_all_paths* tenía que contemplarse el caso de que el grafo insertado fuera vaío. Se solucionó este problema agregando la validación y retornando una lista vacía para indicar que no se encontró un camino.
- Cuando se ingresaba una nueva arista a un nodo, las aristas anteriores a estas se eliminaban. Se solucionó obteniendo las aristas del nodo que existían, insertar la nueva arista como el primer elemento de la lista y luego unir las conexiones antiguas.

Errores en implementación de GUI

- Al mostrar el resultado de una ruta, se tuvo dificultades para eliminar esta ruta y mostrar otra. Esto se solucionó limpiando todos los widgets que se encontraban en el mapa y volviendo a graficarlos.
- Las líneas que se dibujaban en un *bitmap* para representar todos los caminos no se estaban mostrando. Su solución fue dibujar estas líneas en el mismo *canvas* donde se encontraba el mapa.

Errores en análisis y planificación del proyecto

A la hora de analizar los requerimientos para el proyecto, se decidió hacer la ciudad con un grafo estático precargado, donde no era posible agregar nuevos nodos o aristas. Luego de realizar la consulta al profesor, fue explicado que era un requerimiento un grafo dinámico con cualquier cantidad de nodos y conexiones. Esto cambió la distribución de tareas para cada integrante, al igual que las horas invertidas en el proyecto.

Este problema pudo ser solucionado, cambiando la interfaz estática que fue desarrollada por una interfaz más flexible para poder agregar nodos y aristas de manera dinámica por el usuario. Además, se crearon funciones para manejar un grafo y poder insertar tanto nodos, como aristas.

Plan de Actividades

Para el planeamiento y análisis previo al desarrollo del proyecto, se hizo uso del método de historias de usuario para obtener las necesidades del programa. Al ser un programa pequeño, se analizaron las siguientes seis historias de usuario:

No.	Yo como	quiero	para
1	Desarrollador	crear un grafo mixto y con peso	la representación de ciudades
2	Desarrollador	investigar sobre GUI en Racket	escoger la mejor herramienta para el proyecto
3	Usuario	disponer con diferentes mapas	visualizar lugares y calles dentro de una ciudad
4	Usuario	escoger un lugar de origen y destino	visualizar la ruta más corta y rutas alternas
5	Usuario	que la aplicación se asemeje a Waze	tener una mejor simulación con GUI amigable
6	Desarrollador	realizar pruebas a la aplicación terminada	encontrar y resolver posibles problemas

Tabla 1. Historias de usuario obtenidas de la especificación del proyecto

Para estas historias de usuario, se dividieron en dos tareas cada una, a las cuales se les asignó un aproximado de horas para cada tarea y un integrante de grupo para que desarrolle esas tareas.

No.	Tarea 1	Horas	Tarea 2	Horas	Asignado a
1	Crear formato para grafo en lista de Racket	1	Definir tres grafos para cada ciudad	1	Erick
2	Investigar sobre herramientas para interfaz gráfica en Racket	2	Ejemplos de aprendizaje con la herramienta escogida	1	María
3	Graficar tres diferentes cuidades	6	Brindar opción al usuario para escoger un punto de origen y de destino	2	María
4	Crear algoritmo que obtenga todas las rutas posibles de un nodo a otro (Acuña)	13	Mostrar todas las rutas disponibles en GUI (Erick)	8	Acuña/Erick
5	Cargar imágenes a GUI (Logo, botones) y diseño con paleta de colores	1	Representación del mapa simulando a la aplicación Waze	2	María
6	Pruebas a elementos de interfaz gráfica	1	Pruebas al comportamiento del algoritmo de búsqueda de rutas en diferentes casos	1	Erick

Tabla 2. División de historias de usuario en tareas, junto con su integrante asignado.

Luego de obtener cada tarea, se discutió el orden en el que el proyecto tenía que desarrollarse, haciendo un mapa de historias de usuario donde estas se ordenan por secuencia y criticidad, junto con las fechas límites para cada iteración del proyecto. Esto se puede observar en la Imagen 9.

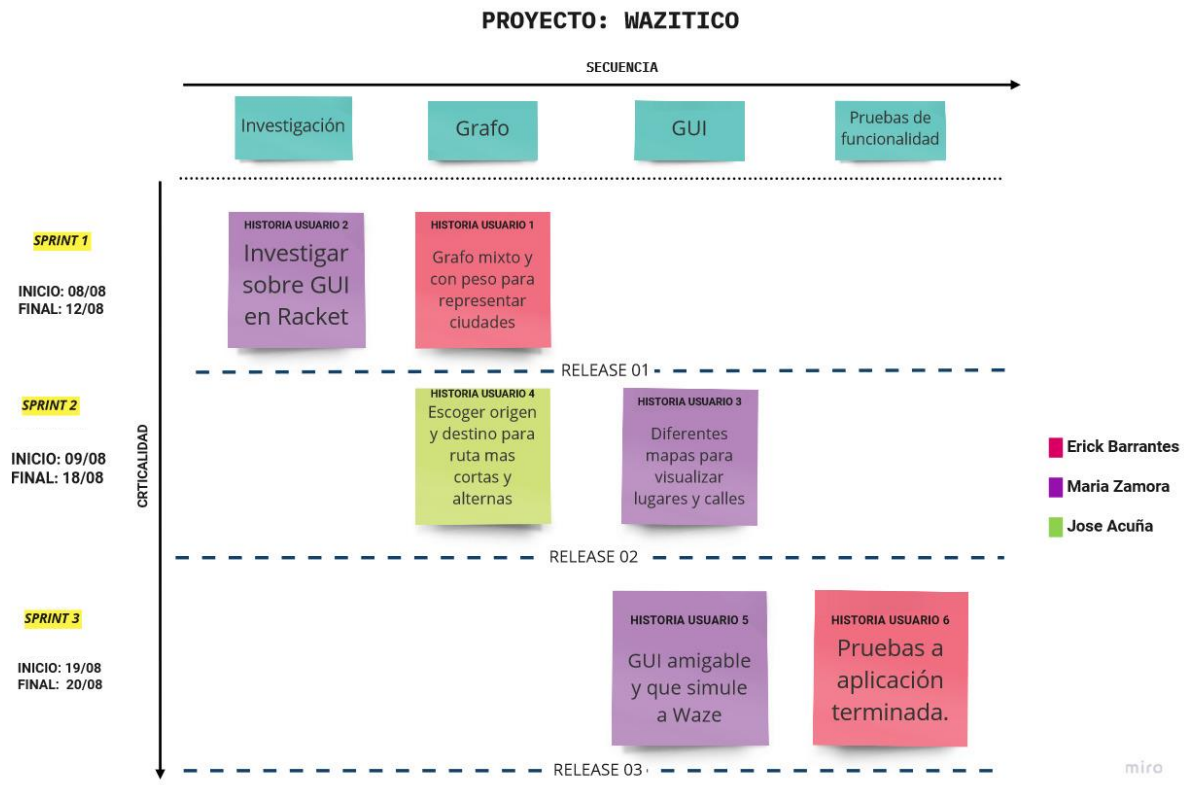

Imagen 12. Mapa de historias de usuario.

Se hace la observación que, debido al problema encontrado en la planificación del proyecto, se realizaron varios cambios a la asignación de tareas y se trabajó más horas de las planteadas. Todos estos cambios se pueden ver representados en la sección de la Bitácora.

Conclusiones y recomendaciones

En todo el proyecto programado se hizo uso del lenguaje de programación *Racket* con el paradigma funcional. Este fue un nuevo lenguaje para todos los integrantes del equipo, con el cual se pueden concluir los siguientes puntos:

- Se logró desarrollar una aplicación y resolver el problema descrito haciendo uso del lenguaje de programación *Racket*
- Se aplicaron conceptos de la programación funcional, como el uso de listas en *Racket* y funciones de recursividad en pila y cola para el manejo y control de estas listas.
- Se desarrolló un grafo como estructura de datos para la representación de las ciudades. Estos grafos se implementaron con listas que siguen un formato, simulando una lista de adyacencia. Se crearon funciones recursivas para encontrar caminos en este grafo
- Se realizó una investigación e implementación de interfaz gráfica con *DrRacket*, para ventanas de inicio, representación de las ciudades, obtener input del usuario y poder graficar rutas en la ciudad.

Por último, queda como aprendizaje y recomendación evitar algunos errores de planificación, donde las horas planeadas de trabajo no fueron acertadas, provocando atrasos en las fechas de entrega esperadas para cada iteración del proyecto.

Bibliografía

Guzman, J. E. (2006). *Introducción a la programació con Scheme*. Cartago: Editorial Tecnológica de Costa Rica.

Racket. (2017, 08 15). *The Racket Graphical Interface Toolkit*. Retrieved from Racket: http://download.racket-lang.org/releases/6.9/doc/gui/

Búsqueda en Profundidad y Búsqueda en anchura. Vicerrectoría de Docencia: Universidad de Antioquia. (n.d.). Retrieved from http://docencia.udea.edu.co/

Bitácora

Sábado 10 de agosto:

En esta fecha el equipo de trabajo se reúne por primera vez. Se analiza los requerimientos del proyecto y se habla de cómo abordar el proyecto.

Erick Barrantes realiza las siguientes actividades:

- Tras analizar los requerimientos, obtuvo las historias de usuario, las tareas en las que están divididas y el mapa de historias de usuario.
- Diseñó los grafos estáticos en papel.

José Acuña realiza las siguientes actividades:

- Inicio la investigación de algoritmos de búsqueda
- Creación del módulo de pruebas

María Zamora realiza las siguientes actividades:

- Investigación sobre bibliotecas de interfaz gráfica para DrRacket
- Inicio del desarrollo de la interfaz gráfica específicamente pantalla principal

Domingo 11 de agosto:

José Acuña realiza las siguientes actividades:

- Implementa la función que busca nodos vecinos
- Implementa función que extiende las rutas adyacentes
- Crea función de búsqueda por profundidad primero
- Crea función de búsqueda por anchura para buscar todos los caminos posibles
- Implementa función para calcular distancia entre dos nodos
- Implementa función para calcular distancia de un camino completo

Erick Barrantes realiza la siguiente actividad:

- Pasa a código los grafos que fueron diseñados en papel.

Lunes 12 de agosto:

José Acuña realiza las siguientes actividades:

- Implementa función para aplicar reverse a varios caminos para reemplazar la función map.

María Zamora realiza las siguientes actividades:

- Finalización de la pantalla principal, inicio de pantalla ciudad.

Miércoles 14 de agosto:

- María Zamora realiza la investigación sobre la incorporación de imágenes a la interfaz gráfica, manejo de paneles.

Viernes 16 de agosto:

- María Zamora define botones en pantalla de ciudad e implementación de imágenes en la pantalla principal.

Sábado 17 de agosto:

José Acuña realiza las siguientes actividades:

- Crea módulo de dibujo
- Implementa funcion para dibujar lineas de colores como caminos
- Los caminos dibujados se renderizan como un BitMap

Domingo 18 de agosto:

María Zamora realiza las siguientes tareas:

- Finalización de la pantalla seleccionarCiudad, inicio y finalización de la interfaz para la pantalla de cada provincia.
- Inicio de la pantalla para agregar nodos dinámicamente.

Erick Barrantes empieza la redacción de la documentación técnica.

Martes 20 de agosto:

Erick Barrantes crea todo el módulo de graphCreator, con todas las funciones necesarias para la creación de grafos de manera dinámica.

Miércoles 21 de agosto:

El equipo se reúne nuevamente para lidiar con los cambios debido al problema de análisis de requerimientos. En esta reunión
María Zamora realiza las siguientes tareas:

- Finalización de la pantalla para agregar nodos dinámicamente, una pantalla para agregar nodo y otra para agregar caminos.
- Incorporación de interfaz con lógica.
- Arreglos a interfaz.

Jueves 22 de agosto:

El equipo se reúne por última vez, para realizar pruebas y arreglo de bugs. Este día se finaliza el proyecto y se entrega al profesor Marco Rivera Meneses.