Paradigmas de programación – (CE-1106)

Grupo 01

Tarea I

Wazitico

Profesor:

Marco Rivera Meneses

Estudiantes:

Luis Felipe Jimenez Ulate 20222111166

Mauricio Luna Acuña 2022

Carlos Eduardo Rodríguez Segura 2022437835

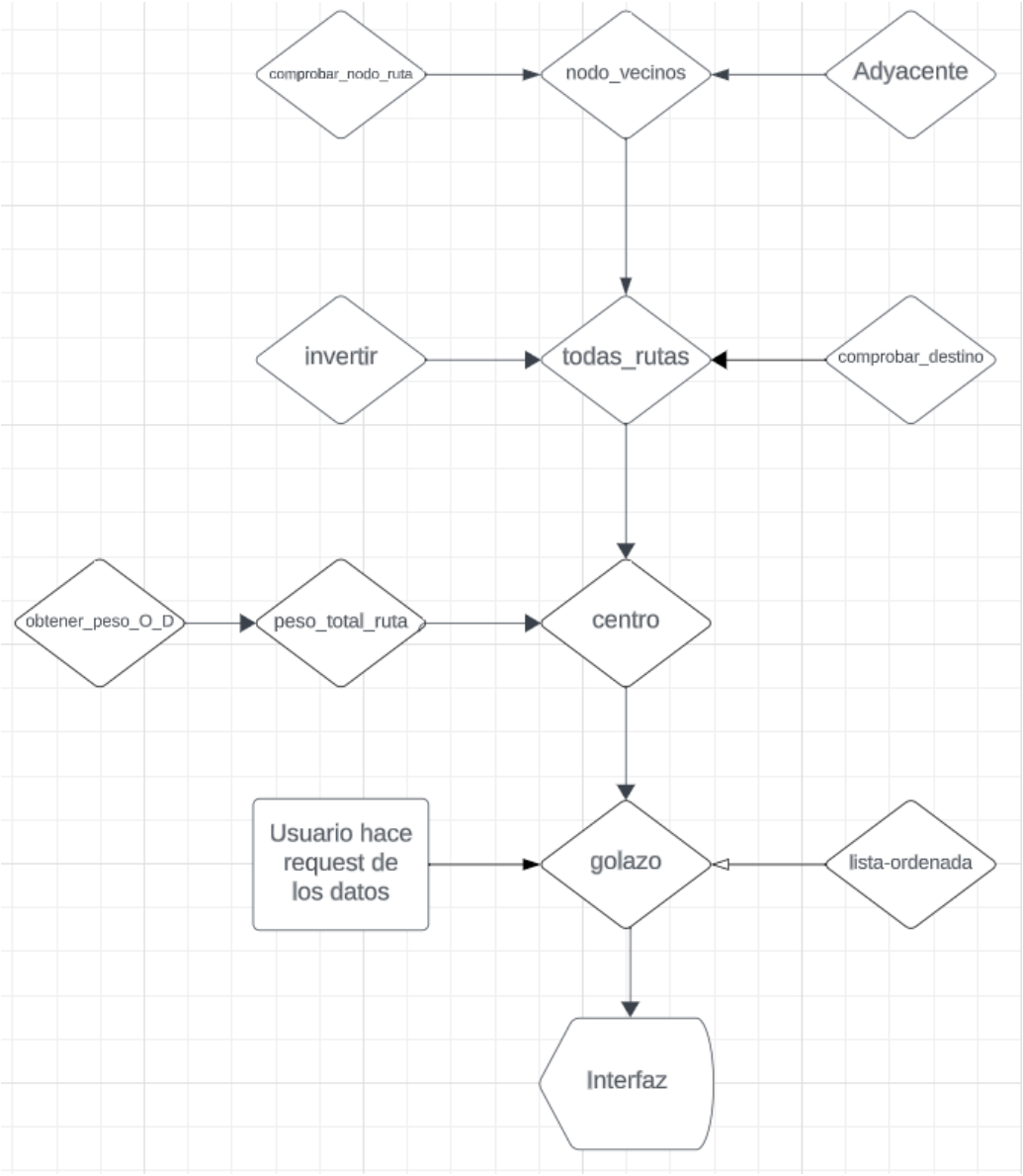
13 / 09 / 2023

II Semestre

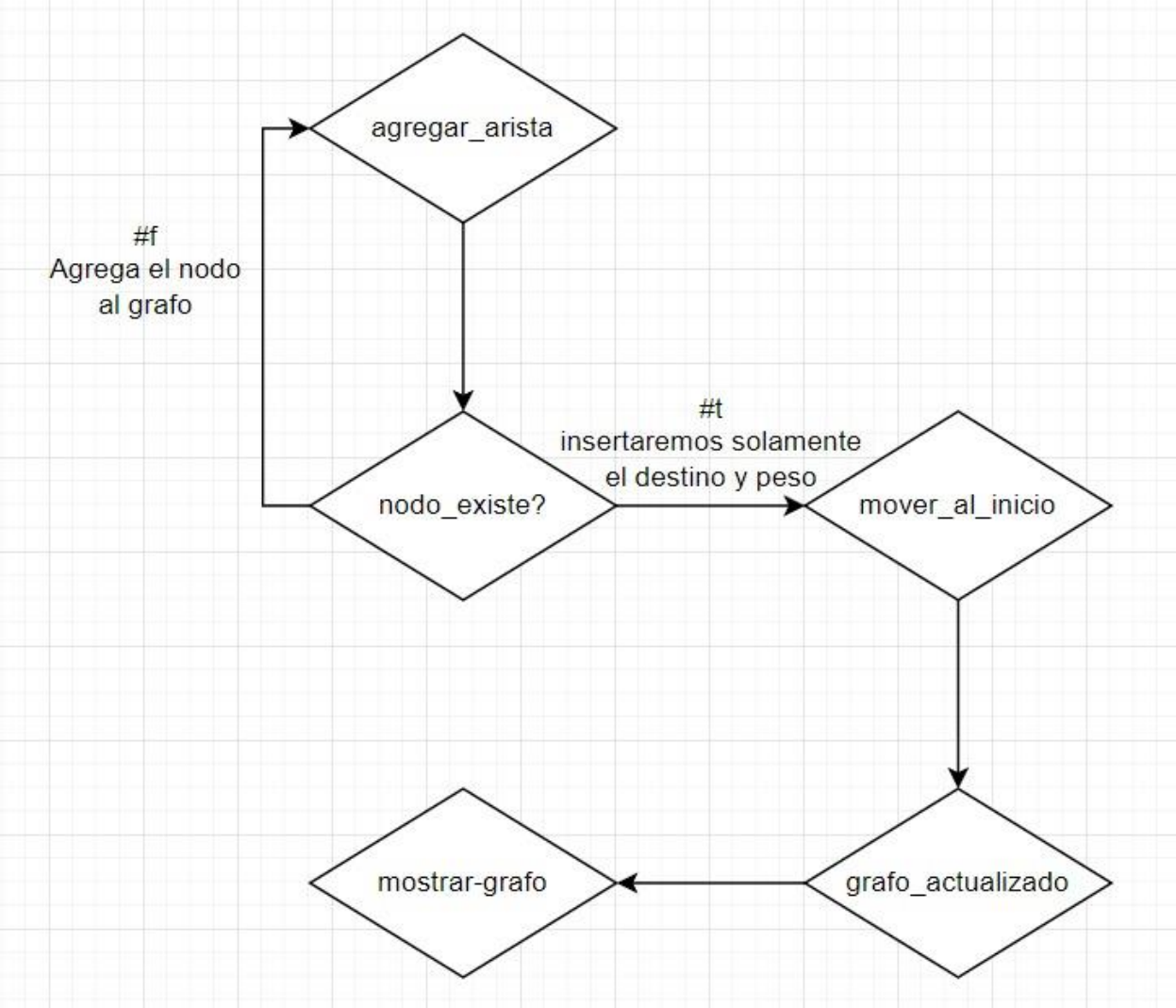
Tabla de Contenidos

Descripción detallada de (los) algoritmo(s) de solución desarrollado(s)

1.1 Diagrama.



1.2 Diagrama.


agregar_arista
#f
Agrega el nodo
al grafo
#t
insertaremos solamente
el destino y peso
nodo_existe?
mover_al_inicio
mostrar-grafo
grafo_actualizado

Descripción de las funciones implementadas.

agregar_arista:

Esta función recibe un nodo de origen, destino, el peso y un grafo previamente definido para insertar estos nodos. La función comprueba primero si el grafo a insertar los nodos es vacío, si es verdadero insertará los nodos y el peso con su respectiva estructura ((A) ((B 1)))), en caso contrario significa que ya hay un nodo con esta estructura en el grafo, por lo que pasará a la siguiente comprobación de la función “nodo_existe?” esto se hace para comprobar si ya hay un nodo con el mismo origen a insertar, así no tener dos nodos origen duplicados y solamente insertar el destino a dicho origen, por ejemplo, en vez de quedar ((A) ((B 1))) (A) ((C 2)))), quede de la forma ((A) ((B 1)(C 2)))), estos lo hacemos usando la función “mover_al_inicio” y después separamos el grafo en dos diferentes, un con el grafo que vamos a insertar el destino y el otro con el resto del grafo que posteriormente los vamos a volver a unir con la función “grafo_actualizado” en caso contrario de que no haya un nodo con ese origen, simplemente se insertará el nodo con la primera estructura mencionada.

Nodo_existe?:

Esta función recibe el grafo que estamos usando y el nodo de origen a insertar, esta es una función recursiva que compara el nodo a insertar con cada nodo origen del grafo.

mover_al_inicio:

Esta función recibe el grafo que estamos usando y un nodo de origen a buscar, esta función mueve el nodo que contiene es origen al inicio del grafo, esto lo hace separando los nodos que no contiene el nodo a buscar y los insertar a una lista aparte para que al final el nodo con el nodo que estamos buscando quede al inicio, la razón de esto es para una mejor manipulación del grafo para insertar un nodo

grafo_actualizado:

Esta función recibe dos grafos, el grafo con el nodo con su nuevo destino y el grafo con el resto de los nodos, esta función une ambos grafos que anteriormente habíamos separado, lo hace recursivamente insertando cada nodo del grafo que no estamos trabajando al grafo en el que le insertamos su nuevo destino.

Mostrar-grafo:

Esta función muestra el grafo con el nodo ya insertado.

Adyacente:

Función encargada de mostrar todos los nodos adyacentes de un nodo. A esta le entra un nodo y el grafo, y retorna una lista con los nodos adyacentes.

Comprobar_nodo_ruta:

Función encargada de comprobar si un nodo está en la ruta de otro nodo. Recibe un nodo y una lista de rutas. Retorna false, si no está en la lista, retorna true, si sí está en la lista de rutas.

nodos_vecinos:

Función encargada de retornar los nodos vecinos de la forma (destino origen), esto para ayudar posteriormente a la función “todas las rutas”. Esta recibe

invertir:

Recibe un nodo y sus elementos, y los invierte. Esto sirve para la construcción de rutas, ya que siempre se general al revés.

comprobar_destino:

Función que comprueba si ya se llegó al destino deseado. Recibe el destino y una lista de recorrido. Si el primer elemento del recorrido es igual al destino (nodo actual en revisión), retorna true. Sino, retorna false.

Todas_rutas:

Función más importante del código, ya que este recibe el nodo origen, el nodo destino y el grafo con el que se está trabajando, y retorna todas las posibles combinaciones de rutas entre estos dos nodos. Esta tiene una función auxiliar, la que nos permite ir reservando en una lista vacía todas las rutas generadas. Su caso base es comprobar si su elemento rutas (una lista), es nulo, mientras que las llamadas recursivas llaman a las funciones comprobar_destino y a nodos vecinos, para recorrer el grafo en su totalidad. Finalmente llama a la función invertir, para dar todos los nodos visitados de la lista en el orden que fueron visitados, hasta llegar al destino. Esto recursivamente hasta que todas las rutas ya hayan sido visitadas.

obtener_peso_O_D:

Función embargada de retornar el peso entre dos nodos. Entra un origen u un destino, sale un número entero.

Peso_total_ruta:

Función encargada de recibir una lista de relaciones entre nodos, y retornar su peso total. Recibe una lista de relaciones o ruta y el grafo con el que se trabaja, y retorna un número entero. Suma recursivamente con una función auxiliar.

comparar-sublistas:

Función encargada de ordenar sublistas dentro de una lista, en base a su primer elemento. Entra una lista de listas, retorna la misma lista ordenada.

Centro:

Función encargada de colocar al inicio de todas las rutas su peso. El peso es proporcionado por la función peso_total_ruta. Recibe una lista de listas y un grafo, y retorna una lista de listas.

Golazo:

Función final, encargada de proporcionar al usuario todas las rutas, acomodadas de menor a mayor, utiliza la función centro y comparar-sublistas para obtener todas las listas de todos los caminos posibles, y los retorna ordenados.

Descripción de la ejemplificación de las estructuras de datos desarrolladas.

Se utilizó la estructura de datos, llamada grafos, en la cual está constituida por nodos o vértices, los cuales tienen 3 partes claves, su nombre, también conocido en este proyecto como origen, un destino, y su relación llamada aristas. En este proyecto se trabajaron los grafos con la siguiente estructura:
((A ((B 5) (C 4)))
(B ((A 5) (A 7)))
(C ((B 3))))
En donde los tres vertices son: A B C, y de ejemplo de aristas se pueden observar A con B, y tiene peso de 5, o C con B, que tiene peso de 3.

Problemas Encontrados:

- Un nodo no podía tener más de 2 destinos pero se arregló modificando el orden del uso de append y list.
- Convertir las listas dadas por la función “todas_rutas” a string.
- Los strings se anteponen, se soluciona dibujando un cuadro de texto encima del anterior.

Problemas sin solución encontrados:

- Dibujar los caminos de forma gráfica. Se intentó colorear las flechas de la ruta utilizada, sin embargo, solo se puede dibujar, once puede borrar, quedando ilegible el menú
- Que el usuario seleccione una ruta específica, como no se podía dibujar la ruta, no se veía muy útil sacar esta funcionalidad. Se decidió enfocar el tiempo en varios bugs.
- Dibujar flechas con dirección resultaba imposible, se recurrió a dibujar un círculo en lugar de una flecha, pero esto se tiene a confundir con un destino o origen.

Plan de Actividades realizadas por estudiantes:

Descripción de la tarea	Tiempo estimado de completitud.	Responsable a cargo	Fecha de entrega [Septiembre, 2023]
Interfaz gráfica: Una interfaz con los siguientes botones y cuadros de texto: > Origen > Destino > Agregar nodo > Eliminar nodo > Ruta elegida (de base se muestra el dijkstra, luego todas las demás, sin ningún orden específico)	3 día(s)	Luis Felipe Jimenez	Domingo 10
Dibujar nodos. El usuario tiene permitido agregar o eliminar nodos, la interfaz debe ser capaz de eliminarlos o agregarlos gráficamente.	3 día(s)	Luis Felipe Jimenez	Domingo 10
Dijkstra. Función que permita encontrar la ruta más rápida, de un nodo a, hacia un nodo b, sin usar variables	3 - 4 día(s)	Carlos Rodríguez	Sabado 9
Agregar nodos. Función que agregue nodos a un grafo. Origen, destino y su peso.	0.5 día(s)	Mauricio Luna	Sabado 9
Encontrar todas las rutas posibles. Una función que envía un grafo, un origen y un destino, y de todas las rutas existentes a eso.	4 día(s)	Carlos Rodríguez Segura	Sabado 9
Conexión entre .rkt Conectar 2 archivos .rkt	1 día(s)	Luis Felipe Jimenez	Lunes 11

Conclusiones.

Se creó una aplicación para encontrar rutas según los lugares ingresados por el usuario y sus conexiones entre sí, utilizando el lenguaje de programación racket.

Utilizando el paradigma funcional, se logró resolver el problema planteado, de búsquedas aritméticas de rutas.

Se logró manipular listas, de tal manera de que se pudo resolver la problemática planteada, aplicando esta estructura de datos.

Recomendaciones.

Para trabajos como este, se recomienda trabajar de una forma ordenada, ya que el racket es un lenguaje en donde el orden y la claridad de lo que hacen las funciones es clave. También trabajar con tiempo, sino saben o no se trabaja con regularidad con este lenguaje, ya que aprender un nuevo lenguaje de programación requiere tiempo y paciencia, y con racket, que no hay variables (al menos en este proyecto), es aún más problemático.

Se recomienda leer librerías sobre racket, ver videos de ejemplos para la parte gráfica y funcional. Aprender a usar el debugger de racket es clave para entender el funcionamiento de las funciones y arreglar los bugs en este.

Bibliografía consultada en todo el proyecto

The racket Guide. (n.d.). https://docs.racket-lang.org/guide/

4.10 pairs and lists. (n.d.). https://docs.racket-lang.org/reference/pairs.html

7 errors. (n.d.). https://docs.racket-lang.org/unstable/error.html

Webalanta. (2018, June 13). *21. Invertir elementos lista - DrRacket* [Video]. YouTube. https://www.youtube.com/watch?v=SXUVY1FGcWk

Iwantcode. (n.d.). *Reddit - Dive into anything*. https://www.reddit.com/r/Racket/comments/4aq339/help_with_an_error_car_violation/

Computer Science. (2016, May 7). *Graph Data Structure 4. Dijkstra's Shortest Path algorithm* [Video]. YouTube. https://www.youtube.com/watch?v=pVfj6mxhdMw

How do I convert a list of strings into a list of lists in Racket? (n.d.). Stack Overflow. https://stackoverflow.com/questions/67065078/how-do-i-convert-a-list-of-strings-into-a-list-of-lists-in-racket

8.3 Reading and writing racket data. (n.d.). https://docs.racket-lang.org/guide/read-write.html

Diego Torre. (2020, April 10). *Búsqueda en Profundidad* [Video]. YouTube. https://www.youtube.com/watch?v=UYtnHmls2ec

Bitácora de trabajo.

Fecha [Septiembre, 2023]	Actividad realizada Ej: a) Actividad 1 b) Actividad 2 c) Actividad 3 (La actividad a, la hizo el estudiante a)	Miembro del grupo Ej: a) Estudiante 1 b) Estudiante 2 c) Estudiante 3
Domingo 3	a) Primera Reunión del grupo. Se plantea una división inicial del trabajo.	a) Felipe Jimenez, Mauricio Luna, Carlos Rodrñiguez
Lunes 4	a) Se inicia la interfaz gráfica del proyecto, en un .rkt y se inicia la investigación de cómo comunicar 2 archivos .rkt b) Se inicia el dijkstra del proyecto, para encontrar la ruta más rápida	a) Luis Felipe Jimenez b) Carlos Rodríguez
Martes 5	-	-
Miercoles 6	a) Se inicia y termina la función de agregar. b) Se inicia la función de buscar todas las rutas posibles c) Se sigue trabajando en la interfaz gráfica	a) Mauricio Luna b) Carlos Eduardo Rodríguez Segura c) Luis Felipe Jimenez
Jueves 7	-	-

Viernes 8	-	-
Sabado 9	a) Se sigue trabajando en la interfaz gráfica	a) Luis Felipe Jimenez
Domingo 10	a) Se termina la función para mostrar todas las rutas posibles de un nodo a, hacia un nodo b b) Se termina la interfaz Gráfica, se trabaja en la comunicación entre los archivos .rkt c) Con base en la interfaz gráfica, se trabaja en el manual de usuario y se inicia la doc externa	a) Carlos Rodríguez b) Luis Felipe Jimenez c) Mauricio Luna
Lunes 11	a) Se termina el manual de usuario y la doc externa b) Se trabaja en bugs del trabajo	a) Carlos Rodríguez b) Luis Felipe Jimenez, Mauricio Luna, Carlos Rodríguez
Martes 12	a) Se trabaja en bugs del trabajo y en la doc externa.	a) Luis Felipe Jimenez, Mauricio Luna, Carlos Rodríguez
Miércoles 13	a) Se documenta el código, se termina la doc externa, y se envía el trabajo	a) Luis Felipe Jimenez, Mauricio Luna, Carlos Rodríguez

Link del github:

https://github.com/MauroCR52/Wazitico.git