Costa Rican Institute of Technology

Computer Engineering

Wazitico

Manual de Usuario

Profesor Ing. Marco Rivera Meneses

Por: Henry Nuñez Perez - 2022089224

Mariana Rojas Rojas - 2020076936

Acuña Durán Oscar Arturo - 2022049304

Computer Engineering

13 de septiembre del 2023

Índice

1. Definición general del proyecto

El presente proyecto tenía como principal objetivo que el estudiante comprendiera el funcionamiento del paradigma funcional. En este paradigma el programador lo que hace es hacer una abstracción del mundo mediante funciones y con la construcción de estas funciones va poco a poco construyendo esta conceptualización del mundo.

Para poder aplicar este paradigma lo que se realizó fue una aplicación similar a *Waze*, llamada **Wazitico**. Esta es una aplicación de navegación que le permite al usuario ver cual es la ruta óptima desde un punto A hasta un punto B.

Dentro de las funcionalidades que tiene la aplicación implementada, en esta se puede construir un mapa, el sistema permite definir lugares y poner carreteras entre estos lugares. Para estas carreteras se puede definir si la carretera es de una vía o si es una vía que se puede transitar en ambas direcciones. Además las carreteras poseen un peso que se relaciona con la distancia entre un punto y otro.

Se puede definir un lugar como el origen para el usuario, donde él va a comenzar su viaje, y luego se puede definir otro lugar como el destino. Una vez realizado esto se puede realizar una búsqueda dentro de la aplicación que mostrará la ruta más corta.

Cabe destacar que la aplicación fue realizada en el lenguaje Racket, implementando únicamente funciones que provienen del paradigma funcional.

2. Descripción de la estructura de datos desarrollada

Se implementó la utilización de un hash para la posterior creación de un grafo que organiza las ciudades y las conecta por sus rutas. Cuando el usuario ingresa una nueva ciudad se llama la función .ªddCitytoGraph". CitiesGraph es un hash que representa el grafo de las ciudades. Cada clave en el hash es el nombre de una ciudad y el valor asociado es una lista de rutas desde esa ciudad hasta otras ciudades. Cuando se agrega una ciudad al grafo, se le asigna una posición en el lienzo y se asignan estos datos a un número único en la función .ªddPostoHash". Ahora bien, cuando un usuario ingresa una nueva ruta entre dos ciudades se llama la función .ªdd-route"la cual asigna una nueva lista de rutas relacionada a las ciudades elegidas.

```
(define (add-route city1 city2 weight)
  (define routes-city1 (hash-ref citiesGraph city1 '()))
  (define routes-city2 (hash-ref citiesGraph city2 '()))
  (set! routes-city1 (cons (cons city2 weight) routes-city1))
  (set! routes-city2 (cons (cons city1 weight) routes-city2))
  (hash-set! citiesGraph city1 routes-city1)
  (hash-set! citiesGraph city2 routes-city2)
)
```

Una vez se construye el grafo deseado y se hace la búsqueda de rutas entre dos ciudades, se transforma el hash a un grafo adecuado para la función de pathfinding, la cual se describe de la siguiente manera:

```
(define (transformGraph city-graph)
  (let ([cities (hash-keys city-graph)])
```

```
    (map
     (lambda (city)
       (list city
             (map
              (lambda (route)
                (list (car route) (cdr route)))
              (hash-ref city-graph city '())))))
     cities)))


)
```

La función "transformGraph"genera una lista de listas para representar el grafo en forma de lista de adyacencia en lugar de un hash con relaciones. La función obtiene todas las ciudades en el grafo y utiliza map para recorrer cada ciudad y crear una lista de ciudades adyacentes con sus pesos.

2.1. Ejemplificación:

```
  '((San-Jose (Alajuela 20) (Heredia 15) (Cartago 30))
    (Alajuela (San-Jose 20) (Heredia 25) (Puntarenas 90))
    (Heredia (San-Jose 15) (Alajuela 25) (Limon 100))
    (Cartago (San-Jose 30) (Puntarenas 80))
    (Puntarenas (Alajuela 90) (Cartago 80) (Limon 75))
    (Limon (Heredia 100) (Puntarenas 75))))
```

Este es un ejemplo de un grafo de listas de adyacencias, el primer elemento o nodo de cada una de las sublistas representa cada ciudad en el grafo y la listas siguientes a este nodo son las otras ciudades a las que se conecta

y el costo de llegar ahí.

3. Descripción de las funciones implementadas

3.1. Pathfinding

Se hizo una implementación de un pathfinding que se encarga de devolver un camino entre dos puntos en el caso en el que exista. A continuación se exponen todas las funciones que permiten que el pathfinding funcione:

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

(define (find_path_aux paths end graph)
  (define (helper current-path)
    (cond
      ((null? current-path) '())
      ((equal? end (car current-path)) (reverse current-path))
      (else
       (helper
        (append-map
         (lambda (neighbor)
           (unless (member neighbor current-path)
             (list neighbor (car current-path))))
         (neighbors (car current-path) graph))
        current-path))))

  (helper (list end))
)
```

Como se puede observar, esta función recibe un punto de origen, un punto de llegada y el grafo. La misión de esta función es encontrar un solo camino entre estos dos nodos con eficiencia. Es pertinente mencionar que esta función es sumamente eficiente pero la función de esta eficiencia será detallada posteriormente.

```
52 (define (find_all_paths start end graph)
53   (cond((not(null? graph))
54         (find_all_paths_aux (list (list start)) end graph '()))
55        (else
56         '())
57        )
58 )
59
60 (define (find_all_paths_aux paths end graph result)
61   (cond ((null? paths) (reverse_all result))
62         ((equal? end (caar paths)) (find_all_paths_aux (cdr paths)
63                                                        end
64                                                        graph
65                                                        (cons (car paths)
                                                              result)) )
66         (else (find_all_paths_aux (append
67                                    (extend (car paths) graph)
68                                    (cdr paths))
69                                   end
70                                   graph
71                                   result))
72    )
73 )
```

Como se puede observar en el anterior algoritmo, lo que se hace es

implementar una búsqueda por anchura para poder encontrar todos los caminos posibles desde un punto de origen hasta un punto meta especificado.

Similar a como la función anterior arroja los resultados, la presente función es una lista con n cantidad de elementos donde cada elemento es una sublista con el mismo formato de la función recién descrita anteriormente.

```
75 (define (extend path graph)
76   (extend_aux (neighbors (car path) graph) '() path)
77 )
78
79 (define (extend_aux neighbors result path)
80   (define (helper neighbors result path acc)
81     (cond
82       ((null? neighbors) (append result acc))
83       ((member (car neighbors) path)
84        (helper (cdr neighbors) result path acc))
85       (else
86        (helper (cdr neighbors) result path
87                (append acc (list (cons (car neighbors) path)))))))
88
89   (helper neighbors result path '())
90 )
```

Esta función recibe como entrada un camino y un grafo y a partir de ello lo que esta función busca es extender las rutas. Esta función es sumamente pertinente y es utilizada en la función $find\_all\_paths$, es debido a ello que es sumamente importante de comprender.

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

Esta función busca si un nodo existe dentro del grafo. Si el nodo existe retorna el nodo encontrado y en el caso en el que no exista ella va a retornar una lista vacía.

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

(define (neighbors_aux pairs result)
  (define (helper pairs acc)
    (cond
      ((null? pairs) (reverse acc))
      (else (helper (cdr pairs) (cons (caar pairs) acc)))))

  (helper pairs '())
)
```

En esta función se recibe el nodo y lo que se va a obtener es una lista que contiene todos los nodos que están conectados al nodo que se le dio a la función como parámetro. Esta función es sumamente pertinente de considerar ya que es utilizada en la implementación de extend.

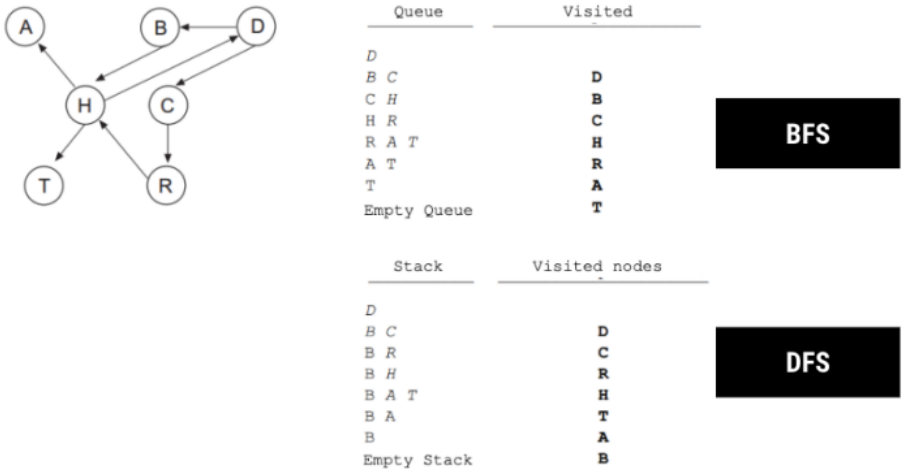
4. Descripción detallada de los algoritmos desarrollados

4.1. Búsqueda por profundidad

Se hizo la implementación de una búsqueda por profundidad. En esta búsqueda el algoritmo va recorriendo nodo a nodo y cada nodo que ya ha sido recorrido tiene necesariamente que ser marcado. Es entonces que se va moviendo hacia las *Profundidades* del grafo al moverse entre nodos. El algoritmo tiene una condición en donde cuando se llega a un punto en donde se alcanza un nodo en donde los vecinos ya han sido marcados, entonces se procede a retroceder.

4.2. Búsqueda por Anchura

En la implementación de este algoritmo lo que se realiza es una visita de todos los nodos del grafo. Se marca cada nodo que se ha visitado y se marcan los vecinos del nodo actual que aún no han sido marcados. Se continúa este proceso donde nunca se visita un mismo nodo dos veces, esto hasta que todos los nodos hayan sido visitados.



Noguera Mena, L. D. (2023). Curso: Algoritmos y estructuras de datos II [Clase magistral]. Instituto Tecnológico de Costa Rica.

5. Justificación

- Dijkstra y el algoritmo A-Star dan únicamente una ruta, siendo esta la mejor encontrada. El algoritmo Backtracking usa muchísima recursividad y además muestra la primera solución encontrada que no es un criterio para saber si es la solución o no, por eso escogimos como grupo desarrollar el algoritmo de búsqueda por profundidad y búsqueda por anchura. Después también analizamos los pesos de cada ruta para mostrarle al usuario la más corta en primer lugar y sucesivamente hasta llegar a la más larga.

6. Problemas Conocidos

El programa no tiene validaciones para cuando no se ha seleccionado una ruta y se le da show, para cuando no se tienen nodos construidos y se

le da add city o cuando no se tiene una ruta seleccionada y se le da add route. Este tipo de validaciones al no estar realizadas si se ejecuta a estas implementaciones el código va a lanzar un error en la consola.

7. Problemas Encontrados

7.1. Errores en analisis y planificacion

En un inicio por una incorrecta interpretación de la especificación del proyecto el grupo comenzó a implementar la lógica de la aplicación en Racket y la interfaz en Python con una API que conectaba ambas plataformas. Después de conversar con el profesor el grupo se dio cuenta que esta interpretación no era correcta y por lo tanto se tenía que cambiar el acercamiento hacia el proyecto. Se rechazó la idea y se procedió a tener un acercamiento totalmente en racket.

8. Plan de Actividades

Un método que es sumamente útil para poder hacer un correcto planeamiento de actividades y que se acopla con lo que el nuevo plan de estudios enseña en el curso *Principios de Modelado en Ingeniería* es lo que se conoce como historias de usuario. Para este proyecto se van a proponer una serie de historias y estas se van a repartir entre los miembros, de esta manera se va a dar un mejor seguimiento al cumplimiento de las tareas.

- (1) Yo como Usuario quiero poder construir mapas para poder visua-

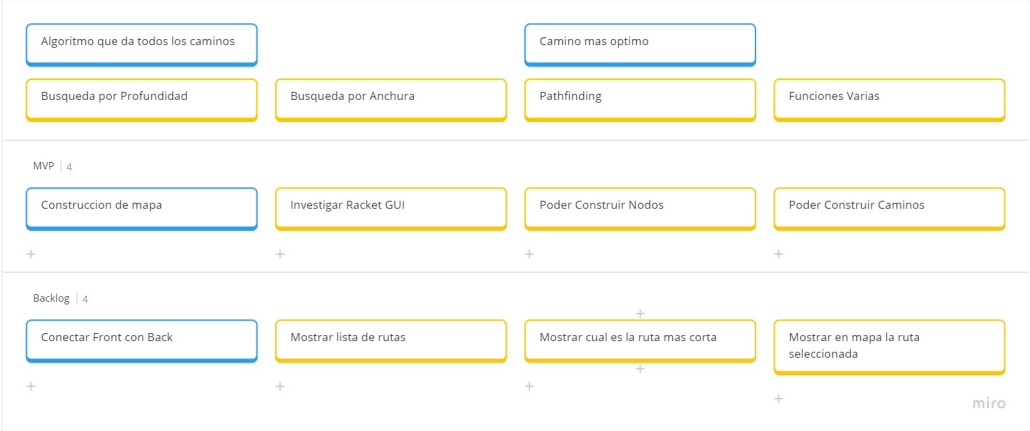
lizar la geografía de las rutas que quiero recorrer.

- (2) Yo como usuario quiero poder escoger un origen y un destino para poder visualizar de esta manera la ruta más corta y las otras rutas.
- (3) Yo como usuario quiero que la ruta seleccionada se visualiza en el mapa que está en pantalla para poder recorrer el camino seleccionado.
- (4) Yo como desarrollador quiero un algoritmo que me permita obtener todos los caminos entre un nodo y otro para poder dárselos al usuario.
- (5) Yo como desarrollador quiero un algoritmo que me permita obtener el camino más corto entre un nodo y otro para poder dárselo al usuario.

Lo que se puede hacer a partir de estas historias de usuario dadas es repartirlas entre los desarrolladores. Observe la siguiente propuesta para la repartición de horas en el proyecto.

- (1) - Henry: Un aproximado de 12 horas
- (4) - Arturo: Un aproximado de 10 horas.
- (5) - Mariana : Un aproximado de 11 horas.

A partir de esto se tiene que establecer un orden en el cual se tendría que desarrollar el proyecto, es entonces que se proponen las siguientes fechas.



9. Conclusiones y recomendaciones

La principal meta del proyecto era aprender cómo funcionaba el paradigma funcional y cómo este proceso de abstracción era diferente a lo que anteriormente había sido utilizado por los miembros del equipo, a partir de esto se puede concluir lo siguiente:

- El proceso de abstracción entre el paradigma funcional y otros paradigmas es sumamente distinto y a la hora de programar hace que varíe mucho la forma en la que tiene que pensar el programador.
- Vale la pena aprender paradigmas viejos como el funcional para volverse programadores más versátiles, de esta manera si algún día aparece un paradigma nuevo va a ser más sencillo poder adaptarse.
- La implementación de ciertas funciones en el paradigma funcional puede ser, en cantidad de líneas de código, mucho más corta que en otros paradigmas como el orientado a objetos.
- Programar en un lenguaje que no tiene tanta documentación como

Racket llega a volverse más complicado, eso permite darse cuenta de lo pertinente que es volverse programadores que no sean dependientes de esa documentación que hay en internet de cada lenguaje moderno y poder crear cosas por cuenta propia.

Luego las recomendaciones que se dan para el proyecto son las siguientes:

- Tener una correcta planificación de qué es lo que se va a realizar para evitar atrasos, de esta manera no se produce código que después se tiene que desechar.
- Hacer una lectura a profundidad de las herramientas del lenguaje que se está utilizando, en el caso concreto de este proyecto se invirtió mucho tiempo en la interfaz gráfica por la poca comprensión que se tenía en como funcionaba RacketGUI.
- Comprender bien el paradigma en el que se está trabajando para no hacer código sucio, la abstracción del mundo tiene que ser la correcta cuando se implementa.
- Tener una orgánica y correcta comunicación entre los miembros del equipo, de esta manera el proceso de desarrollo se vuelve mas rápido y mas amigable para todos los miembros del equipo.

10. Bitacora

Se adjunta en la siguiente tabla la bitácora de trabajo para el desarrollo del proyecto.

Fecha	Actividad Realizada	Comentarios	Participantes
30 de agosto	Desarrollo de funcionalidades basicas en Racket usando el paradigma funcional. Creacion de un grafo para representar la estructura de los mapas.	Se delimito que funcionalidades no se pueden usar y se hace un grafo.	Mariana, Henry, Arturo
31 de agosto	Diseno inicial de la interaz grafica.	Interfaz grafica desarrollada con RacketGUI	Mariana, Henry, Arturo
1 septiembre	Revision y correccion de errores en la representacion del grafo.	Se corrige el grafo para poder representarlo correctamente	Henry, Arturo
2 septiembre	Implementacion de la interfaz grafica del proyecto. Inicio de la implmentacion del algoritmo de busqueda de rutas usando profundidad.	Conexion del grafo con la interfaz e inicio de busqueda (mostrar los nodos)	Arturo, Mariana
3 septiembre	Continuacion y finalizacion del algoritmo de busqueda de rutas usando profundidad.	Relacion entre vecinos y caminos entre los nodos.	Mariana, Henry
4 septiembre	Inicio de implementacion dle algoritmo de busqueda de rutas usando anchura.	Busqueda de rutas usando anchura, recorrer el grafo.	Arturo, Mariana, Henry
5 septiembre	Pruebas y depuracion del algoritmo de busqueda de rutas en anchura.	Pruebas de caminos en el grafo pero sin la interfaz	Arturo, Mariana, Henry
6 septiembre	Inicio de implementacion del algoritmo de evaluacion de rutas segun pesos.	Modificacion para que encuentra la ruta que menos peso tenga.	Henry, Arturo, Mariana
7 septiembre	Continuacion y finalizacion de la implementacion del algoritmo de evaluacion de rutas segun pesos.	Modificacion para que encuentre todas las rutas de la de menor peso a la de mayor peso.	Henry, Mariana
8 septiembre	Integracion de los algoritmos de busqueda con el proyecto principal y la interfaz grafica.	Se logran unir las funcionalidades del grafo con el grafo generado con la interfaz.	Arturo, Henry
9 septiembre	Pruebas de integracion y correccion de errores. Implementacion de la posibilidad de rutas bidireccionales entre los nodos.	Se hacen diferentes pruebas del proyecto para gantizar su correcta funcionalidad. Creacion del boton one way y del boton both ways.	Arturo, Mariana
10 septiembre	Pruebas y depuracion de las rutas unidireccionales. Ajustes finales en la interfaz grafica.	Ajustes a como se presentan las rutas en la interfaz y pruebas en caso de rutas en uno y/o dos sentidos.	Mariana, Henry
11 septiembre	Revision final del proyecto y correccion de detalles. Documentacion del codigo.	Pruebas finales y se comentanas funcionalidades implementadas.	Arturo, Mariana, Henry
12 septiembre	Documentacion externa del proyecto	Se realiza el manual de usuario y la documentacion tecnica	Arturo, Mariana, Henry

Puedes encontrar la primera tarea en el siguiente repositorio de GitHub:

`https://github.com/arty4325/CE1106-Primera-Tarea.git`