Algoritmos de grafos temporales funcionales

Trabajo Práctico Final

Programación Funcional

Instituto Tecnológico de Buenos Aires

2020

Eliseo Parodi Almaraz 56399

Índice

Introducción

El objetivo de este trabajo práctico es implementar algoritmos de **grafos temporales** explicados en el paper **Path Problems in Temporal Graphs**[1] del departamento de ciencias de la computación e ingeniería de The Chinese University of Hong Kong y del Institute of High Performance Computing en Singapore. Estos grafos temporales son una extensión de la estructura de grafo con nodos y aristas, en la que las últimas en vez de tener pesos como números enteros, son intervalos temporales. Esto hace que algunos algoritmos que dependen del peso de las aristas, como Dijkstra o el algoritmo que dado dos nodos, devuelve si hay un camino entre ellos, varíen debido a que la existencia de los caminos dependen de la interpretación que le queramos dar. Por ejemplo, si tomamos un grafo temporal en el que los nodos representa aeropuertos y las aristas vuelos de un aeropuerto a otro cuyo intervalo es el tiempo de partida y el tiempo de llegada, existe un camino entre dos aeropuertos si existe un vuelo directo, o si los intervalos de los vuelos me permiten hacer el transbordo.

Lo interesante también es que en clase vimos estructuras de datos como árboles que no tienen ningún ciclo dentro de su estructura. Las funciones **map** y **foldr** que aprendimos terminan por la naturaleza de estas estructuras. Si los aplicamos a grafos, debido a la naturaleza de funcional que no permite efectos secundarios en las funciones, estas funciones no terminarían de computar porque la existencia de ciclos haría que se le aplicará las funciones infinitas veces a los nodos dentro de los ciclos, debido a que guardar los nodos visitados con un flag como se hace en los lenguajes es parte de los efectos secundarios que la programación funcional no acepta.

Para poder utilizar los algoritmos de manera más cómoda se realizó una interfaz gráfica en **Happstack**[2] y **Blaze**[3]. Happstack es un framework open source implementado en Haskell bastante completo que maneja HTTP. Blaze es una librería de template implementada en Haskell que en vez de escribir el código en HTML, se escribe el código en Haskell y devuelve el código HTML.

[1] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, Yanyan Xu. (2014) Path Problems in Temporal Graphs. https://www.vldb.org/pvldb/vol7/p721-wu.pdf
[2] Happstack. https://happstack.com/page/view-page-slug/1/happstack
[3] Blaze HTML. https://jaspervdj.be/blaze/

Planteo del problema

Para entender el problema, primero tenemos que definir cómo va a ser el grafo temporal con el que vamos a trabajar. Este grafo está definido por dos conjuntos, el **conjunto de nodos**, y el **conjunto de aristas**. Definimos una arista como un cuatriplete que contiene el **nodo origen**, el **nodo destino**, el **tiempo de partida** y la **duración**, en ese sentido, el tiempo de partido sumado con la duración, es el **tiempo de llegada**.

Para este trabajo se van a implementar ocho algoritmos de grafos temporales. Todos estos se basan en un tipo de **camino temporal** de un nodo origen a un nodo destino, en el cual si ordenamos desde el inicio hasta el final, el tiempo de llegada de una arista es menor al tiempo de partida de la siguiente. Por ejemplo, esto pasa con el grafo mencionado anteriormente de vuelos, para poder hacer una escala, el tiempo del vuelo en el que llegó al aeropuerto de escala, tiene que ser menor al tiempo en el que sale el vuelo que me deja en destino. Para un camino el **tiempo de partida** es el tiempo de partida de la arista que sale de un nodo origen, y el tiempo de llegada, es el tiempo de llegada de la arista que llega al nodo destino.

Para este tipo de caminos, el paper describe 4 tipos de caminos.

- **Earliest Arrival Path:** El camino entre dos nodos en el que el tiempo de llegada sea el mínimo entre todos los caminos existentes dentro de un intervalo.
- **Latest Departure Path:** El camino entre dos nodos en el que el tiempo de partida sea el máximo entre todos los caminos existentes dentro de un intervalo.
- **Fastest Path**: El camino más rápido entre dos nodos. En el que la diferencia entre el tiempo de llegada y el tiempo de partida sea la mínima.
- **Shortest Path**: El camino más corto entre dos nodos, es decir el que contenga menos nodos.

Para cada tipo de camino hay dos tipos de algoritmos:

- Dado un nodo, me devuelve para todos los nodos dentro del grafo, dependiendo el tipo de camino, uno de los siguientes valores:
    - Los tiempos de llegada mínimo partiendo desde el nodo elegido para el **Earliest Arrival Path**.
    - Los tiempos de partida máximo partiendo hacia el nodo elegido para el **Latest Departure Path**.
    - Las duraciones mínimas del camino más rápido desde el nodo elegido para el **Fastest Path**.

    - La longitud del camino más corto desde el nodo elegido para el **Shortest Path**.
- Dado dos nodos, devuelve los nodos del camino y el intervalo correspondiente al mismo.

Para poder implementar el algoritmo, se tiene que implementar la estructura correcta para poder ejecutarlo, para eso se utilizó la estructura de grafo definida por John Launchbury en su paper **Graph Algorithms with a Functional Flavour**[4] como una base del proyecto.

Dos problemas que tienen las estructuras de grafos en programación funcional:

- Muchos algoritmos usan efectos secundarios para recorrer el grafo, como es el caso de Dijkstra en el que se guarda el nodo anterior al buscar los caminos.
- La estructura de grafo no es **foldable** como la estructura de árbol u otras vistas en clases, debido a la aparición de ciclos.

Para poder ejecutar y visualizar los resultados, decidimos implementar una interfaz web con Happstack, en donde se puedan subir archivos CSV con la información de los gráficos, se pueda elegir qué algoritmos aplicar y sobre qué nodos para que el servidor los ejecute.

[4] John Launchbury. (1995) Graph Algorithms with a Functional Flavour. http://www.cse.chalmers.se/edu/year/2016/course/DIT960/slides/graph-algorithms-with-a-functional-flavour.pdf

Sección técnica

Implementación del grafo temporal

A partir de la implementación de Launchbury de grafo se realizó una ampliación para que aceptará ejes con peso como primera prueba. La implementación original es la de un grafo dirigido sin pesos y la que se quiere utilizar es dirigida con pesos. Esta implementación usa el tipo **Array** de Haskell que permite el acceso a un índice del mismo, con orden O(1), pero a diferencia de otros lenguajes de programación donde la edición de elementos de un array es común, Haskell no permite esto sin tener que crear otro **Array** debido a que los dos son entidades diferentes matemáticamente. Se puede editar el valor de un array con el uso de mónadas y **Mutable Array**, pero para la definición de nuestro grafo, el tipo **Array** es suficiente. El **Array** recibe qué tipo de índice se va a usar para accederlo. Para este caso se decidió usar **Int** como índice por una cuestión de facilidad al programar y comparar resultados.

```
type Index = Int
type Table a = Array Index a
type Graph = Table [Index]
```

Definición de grafo usada por Launchbury

```
type EdgeNode a = (Index, a)
type WeightedGraph a = Table [EdgeNode a]
```

Definición de grafo pesado basada en el grafo de Launchbury

La versión original usa un **Array** de listas de índices de nodos destinos para un índice mientras que la versión modificada usa una lista de **EdgeNode** en la que se describe el nodo destino y el peso de la arista. Esta estructura funciona como una lista de adyacencia, una forma de representación típica de un grafo.

Lo único que faltaría es expandir esta estructura para que sea un grafo temporal. Para esto lo único que bastaría es definir el tipo dentro de **EdgeNode**, por lo que necesitaríamos un tipo para representar un intervalo e insertarlo dentro de **EdgeNode**. Con eso ya tendríamos la implementación lista para implementar los algoritmos.

```
type Interval = (Time, Length)
type TemporalGraph = WeightedGraph Interval
```

Definición de grafo temporal basada en el grafo pesado

¿Cómo crear un grafo?

Para esto se utiliza una lista de aristas pero todavía no definimos qué es una arista. Como un grafo temporal está basado en un grafo pesado, vamos a definir el tipo de la arista del último para expandir la definición al tipo de grafo deseado.

```
type WeightedEdge a = (Index, Index, a)
type TemporalEdge = WeightedEdge Interval
```

Definición de eje pesado y eje temporal

Ahora hace falta definir una función para poder crear el grafo. Esta función va a recibir los límites del array (debido a que es inmutable, necesita saber cuál es el tamaño del array y cuales son los valores que se van a usar como índices) y una lista de ejes y va a devolver el grafo pesado. Esta función también funciona para grafos temporales si a es de tipo Interval.

```
type Bounds = (Index, Index)
buildWG :: Bounds -> [WeightedEdge a] -> WeightedGraph a
buildWG bnds es =
    accumArray (flip (:)) [] bnds [(v1, (v2, w)) | (v1, v2, w) <-
es]
```

Definición de la función para crear grafos a partir de una lista de ejes

Primero se transforman la lista de ejes a una lista de **[Index, EdgeNode]**, luego estas listas se agrupan dentro del **Array** usando su índice para crear una lista de **EdgeNode**, que le va a corresponder a dicho índice en el **Array** resultante, creando el grafo.

Implementación de algoritmos que devuelven valores

Para estos algoritmos se usan los grafos como una representación de ejes en una lista ordenados por el **tiempo de partida** en orden ascendente o descendente dependiendo el algoritmo. A esta lista la llamaremos **EdgeStream** para el primer caso y **ReversedEdgeStream** para el segundo.

Estos algoritmos empiezan desde el nodo inicial y van iterando el **Stream** de aristas, si encuentran una arista que salga de este guardan el resultado, en el nodo destino. Ahora el nodo destino está disponible para avanzar y si se encuentra una arista que parta de este, se repite el mismo proceso para el siguiente nodo destino, y así sigue sucesivamente hasta encontrar el resultado para todos los nodos. Esto va armando el resultado a medida que se recorre el grafo guardando los mismos porque las aristas están ordenadas en el **Stream** y se recorre sólo por los nodos que están habilitados para avanzar.

Los casos más sencillos son los casos de **Earliest Arrival Path** y **Latest Departure Path**. Para estos casos, el primer eje que llega a un nodo es la que tiene el valor que se va a guardar para ese nodo, debido a que para el primer caso, se usa el **EdgeStream** y como me interesa el tiempo de llegada mínimo, el primer eje que llega va a ser el que tenga este valor. Para el segundo caso se utiliza el **ReversedEdgeStream** y el tiempo de partida máximo es el primero que llega.

Esta función va a retornar los estados como tipo **Maybe a**, donde devuelve **Nothing** si no encuentra el resultado o **Just a,** sí encuentra el resultado.

Para realizar esto tenemos dos problemas, cómo guardar el resultado si no puedo editar un **Array** sin tener un costo computacional alto o rompiendo el paradigma funcional y cómo itero el **Stream**.

Para resolver el primer problema, decidimos usar una mónada llamada **State Monad**. La idea de esta mónada es ocultar el comportamiento de guardar estados y pasarlos entre funciones que van a compartir el mismo estado para ejecutarse. Para este caso vamos a guardar el estado en un **STArray** que es un array que puede guardar el estado y devolver el resultado como un Array al terminar la ejecución del algoritmo. Esto nos permite también editar los valores dentro del mismo dentro de la mónada y leerlos. Para esto se implementaron las funciones **getInitialState**, que devuelve un estado vacío para todos los índice menos para el índice del nodo inicial donde se guarda un estado. **getValue** para conseguir un valor dado un índice, y **setValue** para guardar el estado en un índice del **Array**.

```
type TemporalSet s = STArray s Index (Maybe Time)

getEmptyState :: Bounds -> ST s (TemporalSet s)
getEmptyState bnds = newArray bnds Nothing

getValue :: TemporalSet s -> Index -> ST s (Maybe Time)
getValue m v = readArray m v
```

```
setValue :: TemporalSet s -> Index -> Time -> ST s ()
setValue m v t = writeArray m v (Just t)

getInitialState :: Bounds -> Index -> Time -> ST s (TemporalSet s)
getInitialState bnds v t = getEmptyState bnds >>= \m -> setValue m
v t >>= \_ -> return m
```

*Funciones que ayudan para utilizar el **StateMonad** en los algoritmos*

Para iterar los ejes del stream se usa una función **foldr** sobre el mismo en el que se manejan **State Monads**. Esta función se llama **edgeStreamFoldr**, recibe los ejes, una función para aplicar el **foldr** y el estado inicial del algoritmo, y devuelve una mónada con el estado final después de correr el algoritmo.

```
type FoldrStateFunc s = TemporalEdge -> ST s (TemporalSet s) -> ST
s (TemporalSet s)

edgeStreamFoldr :: [TemporalEdge] -> FoldrStateFunc s -> ST s
(TemporalSet s) -> ST s (TemporalSet s)
edgeStreamFoldr edgeStream f initialState = foldr f initialState
edgeStream
```

Función para iterar por los ejes

Para el caso de **Earliest Arrival Path** y **Latest Departure Path**, se le pasa a esta función una función para conseguir estos valores, y se recupera el valor dentro de la mónada usando **RunSTArray** que devuelve el estado dentro de la mónada si es un **Array**. El uso de esta mónada hace el código parecer imperativo, como se puede ver en el ejemplo de **earliestArrivalTimeAux**.

```
earliestArrivalAux :: FoldrStateFunc s
earliestArrivalAux (v1, v2, (t, l)) set =
    m <- set
    t1 <- getValue m v1
    if t `maybeGTE` t1 then
        t2 <- getValue m v2
            if (t + l) `maybeLT` t2 then
                _ <- setValue m v2 (t + l)
                return m
            else
                return m
```

```
    else
        return m
```

La función para ejecutar earliestArrivalTime,
como se ve el estilo de la función es bastante imperativo

Para el caso de **Shortest Path** y **Fastest Path** no es tan trivial, porque el recorrido del grafo no es tan trivial. No necesariamente el primer eje que llega del **Stream** es el que completa ese camino, si no que se necesita mantener una lista de posibles resultados por nodo y ejecutar una función de dominancia para decidir cuál de los mismos tomar como resultado, y con qué estados quedarse.

Para esto pensamos una estructura similar a la anterior usando la **State Monad**, pero en vez de guardar un **Maybe Time** se guarda una dupla **(Maybe Time, [(Int, Int)])**. El primer elemento de la dupla es el resultado que vamos a mostrar y una lista de dupla **Integer** e **Integer** que representan los estados que tengo que guardar. En el primer valor se guarda el tamaño del camino para el **Shortest Path** y el tiempo de partida del nodo inicial para el **Fastest Path**, y en el segundo valor, se guarda el **tiempo de partida**. Estos valores se usan para calcular si los valores son válidos y para saber qué valor elegir como resultado. La idea es ir guardando un registro de todos los caminos, que serían los estados, que posiblemente puedan llegar a contener el valor que nosotros queremos.

En este caso se va a agregar una función para obtener el estado inicial (**getInitialStateVL**), leer un valor (**getValueVL**), escribir un valor en el array. Para este caso, se puede poner la dupla con el resultado (**setValueVL**), el resultado (**insertF**) o cambiar la lista de estados (**insertVL**) en un índice.

La iteración por el stream se hace también con un **foldr** parecido al anterior que devuelve también una **State Monad**. La función que recibe el foldr es la que hace la magia de alterar el estado y al igual que en el caso de **EarliestPath** también tiene un estilo muy imperativo.

Implementación de algoritmos de caminos

Para la implementación que devuelven caminos, el paper propone transformar el grafo temporal en un grafo en el que en los nodos estén el identificador del nodo del grafo temporal y uno de los tiempos de las aristas, los que se encuentran en el **tiempo de partida**, se denominan **Vout** y los que contienen el **tiempo de llegada**, se denominan **Vin**.

Después se crean aristas usando tres reglas:

- Se arman aristas del nodo **Vout** al **Vin** que representan los intervalos temporales.

- Se ordenan los nodos dentro de **Vin** por el tiempo que contienen en una lista, en orden ascendente, y después se crea una aristas desde el primer nodo, al segundo, del segundo al tercero y así sucesivamente hasta llegar al último.
- Luego se conectan los nodos de **Vin** y **Vout** para nodos que partieron del mismo nodo del grafo temporal original. Para esto se conecta para cada nodo de **Vin** con un nodo de **Vout** tal que el tiempo del nodo del último sea el mínimo que sea mayor al tiempo de nodo **Vin**.

Con las dos últimas condiciones, se hacen las conexiones entre los nodos que le corresponden a un sólo nodo del grafo temporal original, que representan a los casos en los que se puede salir de un nodo, mientras haya llegado antes del tiempo de salida al mismo.

Luego de hacer la transformación, y obtener el grafo transformado, se pasó a implementar la función **generate** que define Launchbury en su paper. Esta función genera un árbol a través de expandir los nodos por los que va pasando. Esto en un grafo con ciclos dirigidos genera un árbol infinito debido a que siempre va a expandir el mismo nodo varias veces, por ejemplo si tenemos un grafo **a -> b -> a**, al expandir **a**, se llega a **b**, al expandir **b**, se llega a **a,** y así infinitamente.

Al hacer esto, encontramos una propiedad interesante sobre el grafo implementado, y es que todos los nodos tienen aristas que van a nodos cuyo tiempo que contienen es mayor al suyo, porque las duraciones son siempre números positivos. Por eso, no pueden existir ciclos dirigidos dentro del grafo, y la función **generate** siempre termina para estos casos.

Luego, para buscar dentro del árbol, se hizo una función **pathsToNode** que devuelve todos los caminos desde el nodo inicial al destino, y se pasan los caminos por una función **foldr** que devuelve el camino deseado. El tipo de camino está definido por la función que recibe **foldr** para comparar los caminos.

Implementación de la UI

Para hacer la UI se decidió usar Happstack que es un framework web con Blaze para escribir los templates de HTML. La ventaja de esto es que la interfaz se hace con HTML que estamos acostumbrados a utilizar, y entendemos cómo funciona el protocolo HTTP. Otra ventaja de Happstack es que tiene una documentación amplia.

Esta UI se hizo para poder ver los datos devueltos de una manera más limpia que en la consola y para poder manejar los algoritmos más fácilmente que por consola.

Templating

Blaze usa mónadas con los nombres de los tags HTML para dar una sensación de estar trabajando con código HTML de una manera funcional con sabor a Haskell. Esto sirve para formar un árbol como el XML, donde los tags más cerca de la raíz en el árbol son los primeros que se llaman en la mónada y dentro de la ejecución de la mónada están los tags que contiene ese tag en la jerarquía.

```
baseTemplate :: String -> Html -> Html
baseTemplate activeLink html_ =
    html $ do
        getHead "Upload Form"
        body ! class_ "body-text-center" $ do
            B.div ! class_ "cover-container d-flex w-100 h-100 p-3
mx-auto flex-column" $ do
                header ! class_ "masthead mb-auto" $ do
                    B.div ! class_ "inner" $ do
                        h3 ! class_ "masthead-brand" $ do
                            B.a ! href "/" $ do "Graphkell"
                    nav ! class_ "nav nav-masthead
justify-content-center" $ do
                        B.a ! href "/" ! class_ (toValue
("nav-link " ++ (getActiveLink "algorithms" activeLink))) $ do
"Algorithms"
                        B.a ! href "/paths" ! class_ (toValue
("nav-link " ++ (getActiveLink "paths" activeLink))) $ do "Paths"
                main ! role "main" ! class_ "inner cover" $ do
                    html_
                footer ! class_ "mastfoot mt-auto" $ do
                    B.div ! class_ "inner" $ do
                        p "ITBA - Eliseo Parodi Almaraz 2020"
```

Ejemplo de cómo funciona el templating en Blaze

Como se ve, los **do** indican cuando empieza un tag y con el indentado se ve cuando termina. Esta función **baseTemplate**, muestra otro caso interesante de templating, en el que se puede pasar un tipo **HTML** dentro del mismo. Esta función se utiliza para usar siempre el mismo header y footer de la página, ya que no varían dentro de la misma y pasarle el contenido HTML que se quiere tener dentro de esa estructura. Así para distintas páginas dentro de mi aplicación, se usa el mismo header y footer, pero se muestra la página para correr los algoritmos o los resultados de los mismos, sin tener que escribir el código dos veces.

Formularios

Los formularios se realizaron usando el estilo para poder crearlos desde el **HTML** sin necesidad de usar Javascript para no tener que utilizar otros lenguajes. Esto se hizo a través de los templates siguiendo el estilo mencionado en la sección anterior. Para recibir los datos de los formularios en **Happstack** se utiliza la función **look** que dado el nombre del campo dentro del formulario, devuelve una mónada con el resultado. Para utilizar los algoritmos se carga un archivo **CSV** que se va a guardar en una carpeta de nuestra computadora y se puede leer desde **Haskell**. Este archivo para poder leerlo necesita un tratamiento especial usando la función **lookFile**, que devuelve información del archivo, también en una mónada. Luego se chequea si la información existe y tenga sentido.

Integración con los algoritmos

Para poder correr los algoritmos, se lee el archivo y se lo pasa por un analizador de CSV. La lectura del archivo devuelve una mónada IO con el texto del archivo. El analizador de CSV devuelve una lista de **Record**, con la data que lee del CSV. Este CSV contiene las aristas y se transforman los **Record** en las aristas para el grafo temporal, y luego se arma el mismo. En el formulario se recibe la información sobre el algoritmo que se va a correr, se busca en una lista que indica la correspondencia de un **String** con una función algoritmo. Con el algoritmo y el grafo ya se puede correr el algoritmo. Se corre el mismo y el resultado se lo agrega al html con una función que transforma el resultado a un html.

Conclusiones

Teniendo en cuenta el desarrollo del trabajo práctico, terminamos utilizando estructuras de grafos que son foldables, como es la lista de aristas y el grafo dirigido sin ciclos. A pesar del poco uso, la lista de adyacencia propuesta por Launchbury resultó ser eficiente a la hora de obtener valores indexados dentro de la misma, debido a que funciona como un array. La desventaja que tiene es que a diferencia de los lenguajes imperativos, para poder modificar un array eficientemente, se tiene que usar mónadas y arrays mutables, lo que hacen más oscuro el código. Al final, el resultado es un código que parece imperativo por la naturaleza del manejo de mónadas dentro del bloque **do** que ofrece el lenguaje.

El uso de la **State Monad** es interesante porque aísla el comportamiento de un ente que va pasando por diferentes funciones y así evitar exponer los efectos secundarios que se generan dentro del ente dentro de la mónada. Lo interesante también es que este tipo de mónadas, a diferencia de las IO, permiten conseguir el valor dentro de la misma, ya que no lidian con efectos secundarios como que se cierre un socket donde se lee un archivo, o errores típicos similares de manejarse con recursos de la computadora.

Con respecto a la UI, nos pareció interesante cómo funciona un framework web, que es algo muy común en los trabajos disponibles actualmente en programación. A pesar de que no se usó de una manera tan extensa, con base de datos y permisos para diferentes usuarios, si no para servir HTML dinámico y manejar archivos CSV del lado del servidor. Es interesante también cómo se arma la estructura con mónadas con **Blaze**, que se arma un árbol que representa el HTML con funciones que representan los tags.