ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 458E
Functional Programming

Homework 3

Ali Emre Kaya
150210097
kayaemr21@itu.edu.tr

SPRING 2025

Contents

1 Introduction

In this homework, a large collaborative knowledge base consisting of data is given to us in `freebase.tsv`. This file has codes (MIDs) like `/m/0kfv9` and `/m/011lsq`. This codes connected with edges like `/tv/tv program/regular cast ./tv/regular tv appearance/actor`. `mid2name.tsv` includes the names of this codes (as descripted in the homework paper, I only take the first occurance).

The purpose is to find minimum path between two codes.

2 How to run?

To run program:

runhaskell *main.hs freebase.tsv mid2name.tsv* $< MID1 > < MID2 >$

An example output for this script:

runhaskell *main.hs freebase.tsv mid2name.tsv /m/0glt670 /m/0g5llry*

Search: /m/0glt670 to /m/0g5llry
Distance: 3

MID - Name

/m/0glt670 - Hip hop music
/m/011lb90 - Tyrese Gibson
/m/01kgxf - Paul Walker
/m/0g5llry - The Church of Jesus Christ of Latter-day Saints

3 Code

3.1 File Parsing

```haskell
-- Parse a line of freebase.tsv into (source, destination)
parseEdgeLine :: T.Text -> Maybe (MID, MID)
parseEdgeLine line =
    case T.splitOn "\t" line of
        (src : _rel : dst : _) -> Just (T.strip src, T.strip dst)
        _ -> Nothing

-- Parse a line of mid2name.tsv into (MID, Name)
parseNameLine :: T.Text -> Maybe (MID, T.Text)
parseNameLine line =
    case T.splitOn "\t" line of
        (mid : name : _) -> Just (T.strip mid, T.strip name)
        _ -> Nothing
```

Since the files are TSV (tab-separated values), I split each line using the `\t` delimiter into three parts: source, relation, and destination. If there are not three distinct parts to split using `\t`, I return `Nothing`. I use `Maybe` to do this; it allows me to provide type safety without stopping the program.

The only difference is that I take 2 values instead of 3 to parse the file with MID and name.

I also check that a MID value reflects only one value, when assigning the parsed values to a map with `foldl`, I do not parse this value if it already exists in the map.

3.2 Graph

I needed to create a graph structure to convert the system to a directed graph.

```
type Graph = Map.Map MID [MID]
-- [MID0] : [MID1, MID2, MID3, ...]
```

So, I defined the Graph type. This was a structure that points a MID value to a MID array in a map.

```
-- Build the graph as an adjacency list
buildGraph :: [(MID, MID)] -> Graph
buildGraph = foldr insertEdge Map.empty
  where
    insertEdge (src, dst) = Map.insertWith insertOnce src [dst]
    insertOnce new old = if head new `elem` old then old else new ++ old
```

To create the graph, I took 2 MID values and made `src -> dst`. To do this, I first read the entire `freebase.tsv` file and parsed each line to 2 MID values. Then I gave this array to the `buildGraph` function. This function creates a src MID (if not exists) and adds the new values to the dst array it points to. The `insertOnce` function prevents exactly the same edge pairs from forming (if I had designed weighted edges with relations, it would have made sense to include them).

3.3 BFS

I used BFS to calculate the distance, because I considered the weight of each edge as 1, so there is no need for more complicated algorithms like Dijkstra.

The function I prepared, Graph, takes MID(src) and MID(dst) and converts the passed paths, and the length. I get the result as Maybe, because if it cannot find a path, the result will be Nothing. Using Set prevents revisits, and using Seq I can use FIFO operations as I want and run BFS properly.

```haskell
-- BFS to find the shortest path
bfs :: Graph -> MID -> MID -> Maybe ([MID], Int)
bfs graph start goal = go Set.empty (Seq.singleton (start, [start]))
  where
    go _ Seq.Empty = Nothing
    go visited ((current, path) Seq.:<| queue)
      | current == goal = Just (path, length path - 1)
      | current `Set.member` visited = go visited queue
      | otherwise =
          let neighbors = Map.findWithDefault [] current graph
              newPaths = [(n, path ++ [n]) | n <- neighbors]
          in go (Set.insert current visited) (queue Seq.>< Seq.fromList newPaths)
```

The actual bfs function uses a helper function called go. The visited set is initialized with an empty one. The queue is only initialized with start. queue: [start, [start]] As new values arrive, the change will be as follows:

Neighbors of `start`:	`mid0`, `mid1`
Neighbors of `mid0`:	`mid00`, `mid01`

Queue:

1. (`start`, [`start`])
2. (`mid0`, [`start`, `mid0`])
   (`mid1`, [`start`, `mid1`])
3. (`mid00`, [`start`, `mid0`, `mid00`])
   (`mid01`, [`start`, `mid0`, `mid01`])
   (`mid1`, [`start`, `mid1`])

If the goal is reached, the function returns the path and length it found, if mid has been visited before, mid is skipped, otherwise the go function continues itself recursively with the newly updated visited and queue.

3.4 Others

```haskell
{-# LANGUAGE OverloadedStrings #-}
```

Since I am using T.Text and Haskell does not work properly when assigning strings to T.Text, this extension is important for ease of converting strings to the type I want.

```haskell
import qualified Data.Map.Strict as Map
import qualified Data.Set as Set
import qualified Data.Sequence as Seq
import qualified Data.Text as T
import qualified Data.Text.IO as TIO
import System.Environment (getArgs)
import Data.Maybe (mapMaybe)
```

I used Map to do MID = Name mappings in general. I used Set to keep visited MIDs in BFS. I used Seq to design the necessary queue logic while writing the BFS algorithm. I used Data.Text to make string operations easier. I used Data.Text.IO to read tsc files as text. I used System.Environment(getArgs) to run Haskell with arguments from the terminal. I used mapMaybe to write only valid data to a map.

REFERENCES