# BLG 322E - Computer Architecture

Assignment 2

Feyza Nur Keleş
820220332

June 7, 2025

## Contents

**1 Problem Definition** **2**
1.1 Dataset Overview . . . 2
1.2 Expected Output . . . 2

**2 Implementation Motivation** **2**

**3 Implementation Explanation** **3**
3.1 Graph Representation . . . 3
3.2 Reading Input Files . . . 3
3.3 Breadth-First Search (BFS) . . . 4
3.4 Path Formatting and Output . . . 5
3.5 Main Function Logic . . . 5

**4 Results** **6**
4.1 Short Path Example . . . 6
4.2 Long Path Example . . . 6
4.3 No Path Example . . . 6
4.4 Multiple MIDs with Same Name . . . 6
4.5 Execution with runner.py . . . 7

**5 Comment on the Obtained Results** **7**

**6 Overall Conclusion** **7**

## 1 Problem Definition

We were expected to write a Haskell program that finds the shortest path between two entities in a directed graph derived from Freebase. Each entity is identified by a Machine Identifier, and the path should be returned as distance and a sequence of names.

### 1.1 Dataset Overview

Two files are provided:

- `mid2name.tsv`: Maps MIDs to names.
- `freebase.tsv`: Contains 45k row data for directed edges in the form of source MID, relation, target MID in tsv.

Only the MIDs are used to from the edges. Edges are directional.

### 1.2 Expected Output

For each input pair of names:

- Print the shortest distance (in edges).
- Print the full path using names.
- If no path exists, print a message like: `No connection between ...`

Example:

```
Shortest distance: 3
Full path: Hebei -> China -> United States -> Boise State University
```

## 2 Implementation Motivation

The graph doesn't change while running the code, which made Haskell's immutability be really helpful. Pattern matching and strong types made it easier to work with nodes and paths. I chose BFS instead of DFS because we needed the shortest path, and BFS is better for that in unweighted graphs.

At the start, I thought about how to turn the input files into a graph. I decided to use a Map from MID to list of neighbors. One small challenge was that the graph is directed, so I had to make sure the direction of edges were handled correctly. Also, I needed to keep track of the actual path, not just the distance, so I used a queue where each item keeps the path so far. After trying few ways, BFS with name lookups at the end worked best.

## 3 Implementation Explanation

### 3.1 Graph Representation

In my implementation, I represent the graph using a Haskell `Map MID [MID]`, where each key is a node (machine ID), and the value is a list of its neighbors. This is a common adjacency list representation that allows fast lookup. I define the following aliases to make the code more readable:

```haskell
type MID = T.Text
type Name = T.Text
type Graph = Map.Map MID [MID]
type MIDToName = Map.Map MID Name
```

I chose `Text` over `String` for performance and compatibility with the TSV input format.

### 3.2 Reading Input Files

To read the `mid2name.tsv` file, I wrote a function `loadMid2Name` which parses each line into a (MID, Name) pair and stores only the first occurrence using a reverse fold:

```haskell
loadMid2Name :: FilePath -> IO MIDToName
loadMid2Name filepath = do
    content <- TIO.readFile filepath
    let linesText = T.lines content
        validLines = filter (not . T.null) linesText
        parseLine line = case T.splitOn "\t" line of
            [mid, name] -> Just (T.strip mid, T.strip name)
            _ -> Nothing
        pairs = map parseLine validLines
        validPairs = [p | Just p <- pairs]
    return $ foldr (\(mid, name) acc ->
        if Map.member mid acc then acc else Map.insert
        mid name acc)
        Map.empty (reverse validPairs)
```

The `freebase.tsv` file is loaded in a similar way. I only use the first and third columns to build directed edges:

```
loadGraph :: FilePath -> IO Graph
loadGraph filepath = do
    content <- TIO.readFile filepath
    let linesText = T.lines content
        validLines = filter (not . T.null) linesText
        parseRelation line = case T.splitOn "\t" line of
            [source, _, target] -> Just (T.strip source,
            T.strip target)
            _ -> Nothing
        relations = [r | Just r <- map parseRelation validLines]
    return $ foldr addEdge Map.empty relations
  where
    addEdge (from, to) = Map.insertWith (++) from [to]
```

## 3.3 Breadth-First Search (BFS)

The most important part is the BFS algorithm. I use a queue that stores tuples of the current node and the path taken to reach it. I also maintain a `Set` to track visited nodes. I used this approach to avoid cycles and unnecessary repetition.

```
bfs :: Graph -> MID -> MID -> Maybe (Int, [MID])
bfs graph start end
    | start == end = Just (0, [start])
    | otherwise = bfsHelper (BFSState [(start, [start])]
    (Set.singleton start))
  where
    bfsHelper (BFSState [] _) = Nothing
    bfsHelper (BFSState ((current, path):rest) visitedSet)
        | current == end = Just (length path - 1, reverse path)
        | otherwise =
            let neighbors = Map.findWithDefault [] current graph
                unvisitedNeighbors = filter (\n -> not
                (Set.member n visitedSet)) neighbors
                newPaths = [(neighbor, neighbor : path)
                | neighbor <- unvisitedNeighbors]
                newVisited = foldr Set.insert visitedSet
                unvisitedNeighbors
                newQueue = rest ++ newPaths
            in bfsHelper (BFSState newQueue newVisited)
```

This BFS returns both the distance (number of edges) and the full path. I reverse the path at the end because I construct it in reverse order during the search.

## 3.4 Path Formatting and Output

To print the names instead of just MIDs, I use a helper function:

```
pathToNames :: MIDToName -> [MID] -> [Name]
pathToNames mid2name path = map (\mid -> Map.findWithDefault mid
mid mid2name) path
```

This function uses a fallback in case a MID has no name (rare but possible).

## 3.5 Main Function Logic

My `main` function takes two command-line arguments (MIDs), loads the data, and runs the BFS:

```
main :: IO ()
main = do
    args <- getArgs
    case args of
        [sourceMID, targetMID] -> do
            mid2name <- loadMid2Name "mid2name.tsv"
            graph <- loadGraph "freebase.tsv"
            let source = T.pack sourceMID
                target = T.pack targetMID
            case (Map.lookup source mid2name, Map.lookup target
            mid2name) of
                (Just _, Just _) -> do
                    case bfs graph source target of
                        Nothing -> putStrLn "No connection"
                        Just (distance, path) -> do
                            let pathNames = pathToNames mid2name
                            path
                            putStrLn $ "Shortest distance: " ++
                            show distance
                            putStrLn $ "Full path: " ++
                            intercalate " -> " (map T.unpack
                            pathNames)
                _ -> putStrLn "No connection"
        _ -> do
            putStrLn "Usage: pathfinder <source_MID> <target_MID>"
            exitFailure
```

The program prints the path in a human-readable format. If any MID is not found or there's no connection, it prints an appropriate message.

# 4 Results

## 4.1 Short Path Example

For example, from **Hebei** (`/m/03cyx`) to **Boise State University** (`/m/01w3k0`), the program found this path:

- Distance: 3
- Path: Hebei → China → United States → Boise State University

This was a pretty direct path and shows that the data has good country-level connections.

## 4.2 Long Path Example

From **North Sea** (`/m/075pwf`) to **Ted (film)** (`/m/08ns5s`), the output was:

- Distance: 4
- Path: North Sea → Spanish Civil War → Pan's Labyrinth → Visual effects supervisor → Ted (film)

Kinda unexpected chain but shows how people/movies can connect things that don't seem related at first.

## 4.3 No Path Example

One test gave no connection at all:

- From **Decanoic acid** to **Long snapper**
- Output: `No connection between /m/075pwf and /m/08ns5s`

This shows the program handles disconnected pairs correctly and doesn't crash or hang.

## 4.4 Multiple MIDs with Same Name

In `mid2name.tsv`, some MIDs had the same name (e.g., multiple entries for "John Smith" or such). But only the first name was used, just like it was supposed to. So even if later lines had better names, they were skipped. This made the name lookup consistent and fast.

### 4.5 Execution with runner.py

I used the Python file `runner.py` given to test my code.

- Compiles the Haskell file with `ghc`
- Loads the names from `mid2name.tsv`
- Randomly samples MID pairs
- Calls `./pathfinder` for each one

Only change I had to make was using `./pathfinder` instead of just `pathfinder` since I'm on Unix-based terminal. Otherwise it worked great.

## 5 Comment on the Obtained Results

Most of the paths were surprisingly short (like 3–4 steps) even if they're far in meaning. A lot of the graph is connected well. But there are also some isolated parts, like science vs sports. Some things just have no way to reach each other and makes sense for some topics. The BFS is fast even for long paths. Memory also seemed fine. Haskell's lazy eval and immutability probably helped a bit here too.

## 6 Overall Conclusion

This project showed how we can model real-world data as graphs and use simple algorithms like BFS to find useful paths and relations. Haskell helped by keeping the logic clean and avoiding side effects. Also, pattern matching and type system made bugs less likely. And Freebase-style graphs are used in real solutions like Google Search or recommendation systems. So it was interesting to see how those knowledge graphs are traversed internally.