Function Programming Project Report

- Jiayuan Li

Introduction

In this project, I implemented a serial version of Ford-Fulkerson algorithm to solve the maximum flow problem given a graph. The Ford-Fulkerson algorithm is a famous algorithm for solving the max-flow/min-cut problems. The algorithm can be described as following:

Inputs Given a Network $G = (V, E)$ with flow capacity c, a source node s, and a sink node t

Output Compute a flow f from s to t of maximum value

1. $f(u, v) \leftarrow 0$ for all edges (u, v)
2. While there is a path p from s to t in G_f, such that $c_f(u, v) > 0$ for all edges $(u, v) \in p$:
    1. Find $c_f(p) = \min\{c_f(u, v) : (u, v) \in p\}$
    2. For each edge $(u, v) \in p$
        1. $f(u, v) \leftarrow f(u, v) + c_f(p)$ (*Send flow along the path*)
        2. $f(v, u) \leftarrow f(v, u) - c_f(p)$ (*The flow might be "returned" later*)

(source: from wikipedia)

Implementation

- Graph representation

  ```
  type Graph = ([Int], Map Int (Set Int), Map (Int, Int) Int)
  ```

  The graph in this project is represented by the data structure specified above, which is a tuple consist of 3 elements.

  - `[Int]` is a list of integer representing all vertices in the graph
  - `Map Int (Set Int)` represents the adjacency list, in which each pair `(Int, Set Int)` represents a node in the graph and all its neighbors' node ID.
  - `Map (Int, Int) Int` contains the capacity information. The key `(Int, Int)` represent an edge in the graph, and the value `Int` represents the residual capacity of the edge.
- Find augment path from source to sink

  ```
  findAugmentingPath :: Graph -> Int -> Int -> Maybe [(Int, Int, Int)]
  ```

The function defined above will find an augmented path between the source node and the sink node. The function receives 3 parameters, a graph defined as above, a source node ID, and a sink node ID.

The `findAugmentingPath` will use breath-first-search to find whether there is a path from source to sink in which all the edges has positive residual capacity. If such path doesn't exist, the function will return `Nothing`. If a path is identified, the function will return a list of 3-tuples where each tuple represents an edge in the path and the 3 elements in the tuple are destination of the edge, source of the edge, and residual capacity of the edge respectively.

- Find the bottleneck capacity in a given path

```
bottleneckCapacity :: [(Int, Int, Int)] -> Int
```

As specified above, the 3 elements in the tuple are destination of the edge, source of the edge, and residual capacity of the edge respectively. The function will filter the entire path and return the minimum residual capacity in the path.

- Push flow

```
addFlow :: Int -> [(Int, Int, Int)] -> Map (Int, Int) Int -> Map (Int, Int) Int
```

The function receives 3 parameters. The first parameter is the bottleneck capacity calculated by the last function, which is the flow that can be pushed alone the path. The second parameter is the augmented path, and the third parameter is the capacity information of the graph. The function will return updated capacity information after the flow is pushed among the specified augmented path.

The push process involves both subtract the flow from each edges' residual capacity among the augmented path and add the flow to each reverse edges' residual capacity among the augmented path.

- Final Function Call

```
fordFulkerson :: Graph -> Int -> Int -> Int
```

With the implementation of the previous helper functions, we can now implement the Ford-Fulkerson algorithm. Inside the function there is a helper function `go` that recursively calls itself. The `go` function is defined by

```
go :: Int -> Map (Int, Int) Int -> Int
```

where the first parameter is the current maximum flow and the second parameter is the current residual capacity information of the residual graph. The function will first call `findAugmentingPath`. In the case that `Nothing` is returned, the `go` function will just return the current maximum flow. Otherwise, it will calculate the bottleneck capacity, push the flow, and recursively call itself by `go (currentFlow + bottleneckCapacity) updatedCapacityInformation`.

Dataset and Experiment Result

The dataset is obtained from this github repo. The test dataset used for the project are the graph with 4, 50, 100, 500 vertices. In each file, each line represents an edge in the graph and is given by

```
<src> <dst> <capacity>
```

With the data given, the program we just implemented has the following runtime result

# of Node	# of Edges	Run Time	Max Flow
4	5	0.05s	2000
50	612	0.13s	828
100	2475	2.41s	1251
500	62375	434.39s	7192

We can see as the number of edges and the max flow grows, the run time grows significantly.

Parallel Implementation (Future Work)

Max-flow/min-cut algorithm are very hard to parallelize since the augmented path finding step must be independent and must search the entire graph. It may involve some graph partition technique and advanced all-gather type parallel programming pattern. Due to limited time in the final season, I don't have much time implement the parallel version of the Ford-Fulkerson algorithm. I might implement them in the future if time permitting.

Guide on how to run the project

All the code are in the `src/Libs.hs` file. To run the project, go to the root directory and enter

```
stack ghci
```

Then, we can load the module by

```
ghci> :l Lib
```

In the interactive terminal, we can read the input from dataset by typing

```
ghci> rawInput <- readLines "data/4.txt"
```

and construct the graph by

```
ghci> g4 = buildGraph rawInput
```

and calculate the maximum flow by

```
ghci> fordFulkerson g4 0 3
```

Code Listing

```
module Lib
    (
        parseLine,
        readLines,
        buildGraph,
        findAugmentingPath,
        bottleneckCapacity,
        addFlow,
        fordFulkerson,
    ) where

import Data.List.Split (splitOn)
import Data.Maybe (fromJust)
import Data.Set (Set)
import Data.Map (Map)
import qualified Data.Set as Set
import qualified Data.Map as Map

import qualified Data.Sequence as Seq
import Data.Sequence ((><))

parseLine :: String -> (Int, Int, Int)
parseLine line = (read src, read dest, read weight)
    where
        [src, dest, weight] = case splitOn " " line of
            [x, y, z] -> [x, y, z]
            _ -> error "Invalid input"

readLines :: FilePath -> IO [(Int, Int, Int)]
readLines filePath = do
    fileContent <- readFile filePath
    let fileLines = lines fileContent
    return (map parseLine fileLines)
```

```haskell
type Graph = ([Int], Map Int (Set Int), Map (Int, Int) Int)

buildGraph :: [(Int, Int, Int)] -> Graph
buildGraph edges = (vs, es, cs)
    where
        vs = Set.toList $ Set.fromList $ concat [[s, t] | (s, t, _) <- edges]
        es = Map.fromList
                [
                    (v, Set.union
                            (Set.fromList [ s | (s, t, _) <- edges, t == v])
                            (Set.fromList [ t | (s, t, _) <- edges, s == v]))
                    | v <- vs
                ]
        cs' = foldr (\(src, dst, c) m -> Map.insert (src, dst) c m) Map.empty edges
        cs = foldr (\(src, dst, _) m -> Map.insert (dst, src) 0 m) cs' edges

findAugmentingPath :: Graph -> Int -> Int -> Maybe [(Int, Int, Int)]
findAugmentingPath (_, es, cs) s t = go (Seq.singleton s) Set.empty Map.empty
    where
        getPath v prevLookup
            | v == s = []
            | otherwise = (dst, src, wei) : getPath src prevLookup
              where (dst, src, wei) = fromJust $ Map.lookup v prevLookup
        go pathSeq visited prevLookup
            | Seq.length pathSeq == 0 = Nothing
            | v == t = Just $ getPath t prevLookup
            | Set.member v visited = go path visited prevLookup
            | otherwise =
                let
                    neighbors = Seq.fromList
                        $ filter (\v2 -> (Map.findWithDefault 0 (v, v2) cs) > 0 &&
                                    (not $ Set.member v2 visited))
                            $ Set.toList $ Map.findWithDefault Set.empty v es
                    prevLookup' = foldr
                        (\v2 m -> Map.insert v2 (v2, v, (Map.findWithDefault 0 (v, v2)
cs)) m)
                                prevLookup neighbors
                in go (path >< neighbors) (Set.insert v visited) prevLookup'
                where
                    v = Seq.index pathSeq 0
                    path = Seq.deleteAt 0 pathSeq

bottleneckCapacity :: [(Int, Int, Int)] -> Int
bottleneckCapacity path = minimum (map (\(_, _, c) -> c) path)

addFlow :: Int -> [(Int, Int, Int)] -> Map (Int, Int) Int -> Map (Int, Int) Int
```

```haskell
addFlow f path cs = go path cs
    where
        go [] cs' = cs'
        go ((dst, src, _):path') csTemp =
            let capacity = Map.findWithDefault 0 (src, dst) csTemp
                capacity' = Map.findWithDefault 0 (dst, src) csTemp
                cs' = Map.insert (src, dst) (capacity - f) cs
                cs'' = Map.insert (dst, src) (capacity' + f) cs'
            in go path' cs''

fordFulkerson :: Graph -> Int -> Int -> Int
fordFulkerson (vs, es, cs) s t = go 0 cs
    where
        go f cs' =
            let p = findAugmentingPath (vs, es, cs') s t
            in case p of
                Nothing -> f
                Just path ->
                    let c = bottleneckCapacity path
                        f' = f + c
                        cs'' = addFlow c path cs'
                    in go f' cs''
```