Bianca Trombetta   btrombetta@college
Calvin Tonini   cni@college
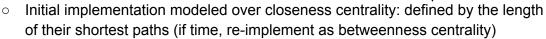Chris Ramirez   cramirez01@college

CS51 Final Specification

Signatures/Interfaces

- Crawler (ids : int list) : dict
  - Input: list of user identifiers
  - Output: dictionary of playlists
- GraphMaker (playlist : dict) (graph : dict) : dict
  - Input: playlist dictionary and graph dictionary
  - Output: graph dictionary with updated values
- PathFinder (s1: node) (s2: node) (graph: dict) : node list
  - An implementation of Floyd-Warshall's between two nodes
  - Input: graph dictionary and two nodes
  - Output: list of songs (nodes), ordered on the shortest path from s1 to s2
- Multiple_PathFinder (s1: node) (graph: dict) : node list list -or a shortest path tree-
  - An implementation of Floyd-Warshall's between a node and the graph
  - Uses PathFinder to to find all shortest paths between a node and all other nodes
  - Input: graph dictionary and a node
  - Output: a list of lists that maps out all the shortest paths from a source node
- All_PathFinder (graph: dict) : node list list list -or a tree list-
  - Looks at shortest distance from all nodes to all other nodes
  - Uses multiple path finder as helper
- Centrality Calculator (graph: dict) (n: int): node list
  - Uses All_Pathfinder to find out what nodes are most central
  - Initial implementation modeled over closeness centrality: defined by the length of their shortest paths (if time, re-implement as betweenness centrality)
  - Input: graph and the number of nodes you want returned (e.g. top 5, 10, etc.).
  - Output: list of nodes in order of centrality
  - Maybe has an option to calculate using different conceptions of centrality
- Neighbor Finder (graph: dict) (song: node) : node list
  - Input: A graph dictionary, and a reference to a specific node
  - Output: list of nodes nearby
- Export/Importer
  - Possible functions that read in and export data structures into disk files.

Modules/Actual Coding

Crawler
PseudoCode : This will run the spotify API over a list of user IDs, returning a dictionary with a bunch of playlists.
Starts with : a list of user IDs
Structure:
    For each user ID, connect to Spotify API
        If user has public public playlists:
            Get from API: songs from each public playlist
            Add playlist as Key with Value as a list of songs
    Iterate over the list (using a for loop)

GraphMaker
**PseudoCode : This will be implemented under the Graph Module
        Takes in a playlist and creates a graph dictionary that translates
        the playlist into nodes (songs) and edges (how many times two
        songs appear together in a playlist)**

Starts with: dictionary of playlists and empty graph
Structure:
    For each playlist in the dictionary:
        For each song in each playlist
            If song already exists in the graph (as a key):
                Iterate over all other songs in the playlist
                If those other songs are already one of the keys inside
                the song, take inverse of value, add 1.0, invert again and
                make it the new value of that song's key
                else add that song as key inside the song with value 1
            Else if song does not exist:
                Adds Song to graph dictionary as key
                Adds all other songs from playlist as key value pairs for
                that song where values are 1.0

    e.g. A graph structure composed of:

            P1: [s1, s2, s4]
            P2: [s2, s5]
            P3: [s1, s3]

would produce:

```
graph = {
        song1 : [{song2:1.0}, {song3:1.0}, {song4:1.0}],
        song2 : [{song1:1.0}, {song4:1.0}, {song5:1.0}]
        song3 : [{song1:1.0}]
        song4 : [{song1:1.0}, {song2:1.0}]
        song5 : [{song2:1.0}]
        }
```

Adding {Playlist4: [song1, song2, song6]} would produce:

```
graph = {
song1 : [{song2: 0.5}, {song3: 1.0}, {song4: 1.0}, {song6: 1.0}],
song2 : [{song1: 0.5}, {song4: 1.0}, {song5: 1.0}, {song6: 1.0}]
song3 : [{song1: 1.0}]
song4 : [{song1: 1.0}, {song2: 1.0}]
song5 : [{song2: 1.0}]
song6 : [{song1: 1.0}, {song2: 1.0}]
}
```

PathFinder
PseudoCode :
Starts with: a weighted graph and two nodes to find the path between them
Return a list of nodes
Structure:
A triple for loop
For each node a
        Looks at every node b
                And node c
                        if the distance from a to b to c is less than the distance from a to
                        c directly, update the distance from a to c with the distance from
                        a to b to c. Also update the Shortest Path Tree that will be
                        returned

CentralityCalculator
PseudoCode: Closeness defined as the reciprocal of the farness.
Starts with: graph output of All_PathFinder.
Structure:
        For every node in the graph:
                Adds together the distance of that node to every other node (farness)
                Stores the reciprocal of that node

Returns a list of the nth most central songs in order, where most central is least far
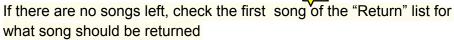
NeighborFinder
PseudoCode:
Starts with: Weighted Graph, and a Target number of songs for playlist
Structure:
Find song in graph
> Look at every song in its value, find the strongest weight (smallest number), store it in our "Return" list, remove it from the graph
>> If there are no songs left, check the first song of the "Return" list for what song should be returned
>>> If it also has no songs left, continue checking songs in the return list until we get another song

if we have returned the target number of songs, stop, otherwise continue searching

Timeline

Week of April 19th
- Write a function that can return a list of songs from a playlist
- Write a function that can return a list of playlists from a user
- Implement saving the results of the above function into a dictionary data structure
- Actually crawl the playlists and save the results
  - Have a single source of playlists as a prototype
  - Goal: 10,000 playlists
- Write modules for saving data to disk in standard format
- Implement PathFinder
- April 24: Functionality Checkpoint

Week of April 26th
- Implement Multiple PathFinder
- Implement PathFinder
- Implement Centrality
- Write a function that generates a graph data structure from playlists (see above)
- April 30 : Implement a graph visualizer
- May 1: Finished Project

Progress Report
Our environment is now set up. We are using Nitrous.io as our cloud based IDE, linked to a GitHub repository, detailed in the next section.

Version Control

https://github.com/CalvinTonini/PlaylistGenome

This repository is public, because it's free! If we decide to pay and make it private, we'll be sure to add read access to course staff.