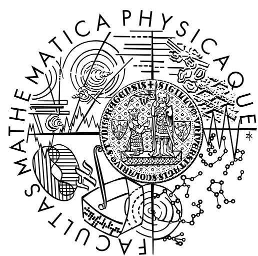


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Pavel Gajdušek

Visualisation of User's Preferences in the Music Domain

Department of Software Engineering

Supervisor of the master thesis: Mgr. Ladislav Peška, Ph.D.
Study programme: Informatika
Study branch: Obecná informatika

Prague 2020

This is not a part of the electronic version of the thesis, do not scan!

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Praha date 28.5.2020 . . . .

Author's signature

Děkuji svému vedoucímu Ladislavu Peškovi za cenné rady a vědecký nadhled při psaní jak práce, tak kódu.

Děkuji Romanu Sobkuliakovi za neocenitelnou radu používat TypeScript a za nadšení pro věc.

Thanks to all of my friends that provided their precious data and thus helped me to debug many edge cases.

Title: Visualisation of User's Preferences in the Music Domain

Author: Pavel Gajdušek

Department: Department of Software Engineering

Supervisor: Mgr. Ladislav Peška, Ph.D., Department of Software Engineering

Abstract: Most of the music portals offer users lists of songs that are the result of black-box algorithms. The recommendation is often nontransparent for users, therefore the irrelevant recommendation might have negative consequences. The recommendation is mainly based on the computation of similarities between users or objects. The computation relies on collaborative techniques or similarity of the contents of the objects. The purpose of this bachelor thesis is to design and implement suitable visualization of these relations in the form of an interactive graph for a certain Spotify user. The visualization should help users realize that their data have inner structures and the recommendations are based on them. The final application should also provide a music playback using the songs contained in the graph.

Keywords: music, graph, cluster, visualization, SpotifyAPI, ReactJS, Django

Název práce: Vizualizace uživatelových preferencí v hudební doméně

Autor: Pavel Gajdušek

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: Mgr. Ladislav Peška, Ph.D., Katedra softwarového inženýrství

Abstrakt: Většina on-line hudebních portálů nabízí uživatelům seznamy doporučených skladeb, které jsou výstupem "black-box" doporučovacích algoritmů. Doporučení často bývá pro uživatele netransparentní a případné doporučení nebo přehrávání irelevantního obsahu tak má výraznější negativní dopady. Doporučování probíhá především na základě vypočtených podobností mezi uživateli nebo objekty založené buď na kolaborativním principu, nebo podobnosti vlastního obsahu. Tato bakalářská práce si klade za cíl navrhnout a implementovat vhodnou vizualizaci těchto vztahů ve formě interaktivního grafu pro konkrétní uživatele platformy Spotify. Vizualizace by měla uživatelům pomoci uvědomit si, že jejich data obsahují vnitřní struktury, ze kterých vychazí doporučování skladeb a umělců. Výstupní program by měl také umožnit přehrávání skladeb obsažených v grafu.

Klíčová slova: hudba, graf, cluster, visualizace, SpotifyAPI, ReactJS, Django

Contents

Introduction **3**

1 Problem analysis **5**

2 Graph-based algorithms **8**
2.1 Graph clustering . . . 8
2.1.1 Louvain algorithm . . . 8
2.1.2 Spectral clustering . . . 9
2.2 Graph visualisation . . . 11
2.2.1 Force-directed layout . . . 11
2.2.2 Growing tree overlapping nodes removal . . . 12

3 Spotify API **15**
3.1 Authorization . . . 15
3.2 Spotify Data . . . 15
3.3 Spotify API limitations . . . 18

4 Application Design and GUI **19**
4.1 Architecture overview . . . 19
4.2 Inicialization . . . 19
4.3 Graph construction . . . 21
4.3.1 Artists . . . 21
4.3.2 Edges among artists . . . 21
4.3.3 Clustering . . . 22
4.3.4 Edges among clusters . . . 24
4.4 Layout . . . 24
4.4.1 Main graph layout . . . 24
4.4.2 Inside cluster graph layout . . . 26
4.5 GUI elements . . . 26
4.5.1 Main graph GUI . . . 26
4.5.2 Main graph nodes (clusters) . . . 26
4.5.3 Inside cluster graph . . . 29
4.5.4 Artist side-bar . . . 29
4.5.5 Similar clusters bar . . . 31
4.5.6 Left side-bar . . . 31
4.5.7 Player . . . 32
4.6 Graph caching . . . 32

5 Code Overview **33**
5.1 Back-end . . . 33
5.1.1 Technologies . . . 33
5.1.2 Back-end structure . . . 33
5.1.3 Further code extensions . . . 36
5.2 Front-end . . . 37
5.2.1 Technologies . . . 37
5.2.2 Graph visualization . . . 38

5.2.3 Front-end structure 40
5.2.4 Further code extensions 42
5.3 Deployment 42
5.3.1 Custom deployment 43

6 Case studies 45

Conclusion 50

Bibliography 52

List of Figures 54

List of Abbreviations 55

A Attachments 56
A.1 Electronic attachments 56

Introduction

Motivation

Recommender systems are present everywhere on the internet and users come in touch with them every day. One of the most important branches that use recommendations is the music industry. This is caused by the number of artist and songs that are easily accessible everywhere on Earth. However, the final recommendation given to the users is the result of black-box algorithms and the users might not understand, why the songs or artist were recommended to them.

Sometimes, this attitude works well, users are satisfied with it and listen to the music that is given to them. The situation is quite different for more picky users who listen to numerous genres and music styles or get annoyed quickly by monotonous results of recommender systems. In this case, it would be desirable to bring in some more or less abstract structure that would show the inner relations between artists and recommended music. This structure could be visualized and given to users that would recognize their habits in a new manner.

In this bachelor thesis, we would like to suggest and implement some visualisation methods that would make clearer the user's preferences and music taste. We will take the user's data and build a graph structure on them. With the help of graph visualizing and clustering algorithms, we will show the graph in the web application and allow users to explore the data in a web browser.

To obtain the data about the artists, songs and users, we chose the Spotify music platform, because of its easily accessible API and a large amount of data and metadata provided to developers. However, the biggest motivation to use Spotify is the number of monthly users (286 million users in March 2020) and the popularity of this music service among its customers.

The goal of this thesis is to make user realize, how connected their world of music is and make them enjoy the moment of exploring inner structures of it.

Related work

Exploring and visualizing hidden structures in the music domain is tempting for many scientists and developers. We would like to introduce a few projects that inspired us and explain which pieces are missing and what we would like to add.

The Music-Map[1] project enables users to search for an arbitrary artist and similar artists are shown around the chosen artists. It shows nicely the closeness of artists. On the other hand, the functionality is very limited, because the only available action is to click artists and see their similar artists. We would like to personalize the data and add more features, such as a music player or draggable nodes.

The Everynoise[2] project visualizes all possible genres of Spotify and allows users to play a chosen song of a chosen genre. It shows similar genres close to each other. However, the data are not personalized and the orientation on the

[1]https://www.music-map.com/

[2]http://everynoise.com/

page is slightly confusing. All genres found on Spotify are displayed as a one long list on the page. We will group artists with their genres into clusters and put more attention to the data as a whole rather than showing every small detail.

The Artist Explorer[3] uses the idea of related artists available on Spotify. The application displays a tree where the root is a chosen artist. We can grow the tree by clicking on artists and explore other artists that are similar to the clicked ones. We like this approach on music data, nevertheless, we want to create a general graph (not a tree) from data obtained from a logged-in user.

Generally speaking, our approach differs from most of the available web applications by the personalization, graph dynamic visualization, systematic clustering and detailed artists data.

[3]https://github.com/fsahin/artist-explorer

1. Problem analysis

The main purpose of this application is to show the user's music preferences. To do so, we must set some rules and approaches towards this goal:

- The shown entities should be very intuitive and easy to understand.
- The data should be structured. In other words, if the whole page is covered with text or images without any order, it will not help the user to understand anything.
- There should be neither too many nor too few entities on the screen. Too many entities are disturbing and too few are boring and not really describing the structure of the given data.
- If there are some relations between the chosen entities, they should be nicely visualized.
- User should know what the shown data mean and why did it appear in his/her visualisation.
- In the music domain, it is not enough to just show names or images of artists and songs. It is necessary to allow users to play some pieces of music.

Let us analyze these requirements and suggest a solution.

Entities

First, we must choose the data that we want to work with. As we are exploring the music domain, we decided to work with the most intuitive entities - artists and songs. Both of those are easy to manipulate with and are relatively "atomic" (there are some exceptions, e.g. artists belonging to music bands or longer pieces divided into parts). Next considerable musical structure that comes into mind is an album. We decided to ignore this concept for the following reasons:

- It is not always strictly defined:
    - some albums contain songs that were firstly published as singles,
    - it is not clear what albums should mean in classical music.
- It is hard to define relations between albums.
- Users do not always know which album their favourite pieces belongs to.

Thus, we will work with artists and songs (in this thesis, we will use both *song* and *track*). Let's suppose, we know how to get the user's favourite songs and artists. Now, we must find a way how to connect them.

Principal data structure - graph

The main point of visualizing some data is to show more or less hidden relations between entities. A graph is the most straightforward data structure that captures the idea of relations very well. A graph consists of nodes and edges that represent entities and relations between them.

We took into account a few possibilities of what entities in the graph should represent in the music domain and how exactly the graph should look like:

- The user is the node in the middle and his/her favourite artists are nodes around.
- Nodes are representations of songs and edges of the similarities between them.
- Nodes are representations of artists and edges of the similarities between them.
- Songs and artists are in one graph with edges representing the similarity or relation of authorship and other combinations.

We chose the most intuitive choice, to have nodes as artists and edges as some similarity between them. This similarity is composed of two factors: genres that two artists have in common and particularly the concept of related artists introduced by Spotify. This concept will be explained further in the Spotify API chapter.

Graph clustering

The number of favourite artists can be quite large. For this reason, we would like to divide them into some groups and show only a reduced part of the big amount of data. There are some desired properties of groups, such as that there should be only similar artists in the group (for graph's theory sake, the groups will be called *clusters*). We will describe this clustering problem further in the Graph algorithms chapter.

Graph visualizing

After we obtained artist clusters, we would like to visualize them. There will be two main views on the data.

1. The graph, where nodes represent the computed clusters and edges the similarities between clusters - let us call it the **main graph** (see figure 1.1).
2. We want to allow the user to click the clusters and see what is inside. We will do it by showing a graph where nodes are artists inside the cluster and edges relations between them - from now on, let us denote it as the **inside cluster graph** (see figure 1.2).

To guarantee that the user knows why the data is shown to him/her, we must include the functionality of clicking artists in the graph. After clicking a certain artist, some details will appear, e.g. songs by the chosen artist that the user liked.

Playing the music

When the data about artists and their songs are given to the user, he/she might want to listen to the songs that are enlisted in the provided information. We will allow the user to chose a certain song, listen to a part of it and also offer him/her a link of the full song leading to the Spotify web player. The reasons for this functionality will be given in the Spotify API chapter 3 and possible improvements in the Conclusion chapter 6 in Further work discussion.

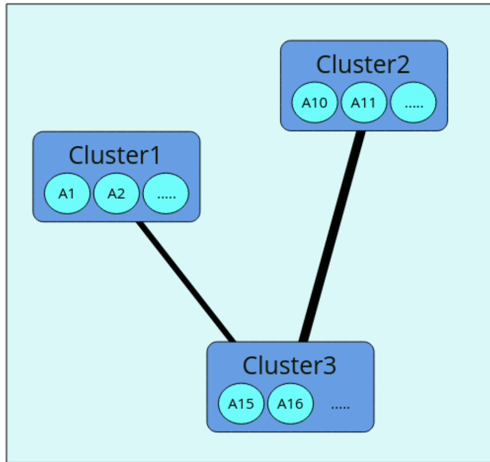


Figure 1.1: Main graph - clusters are nodes

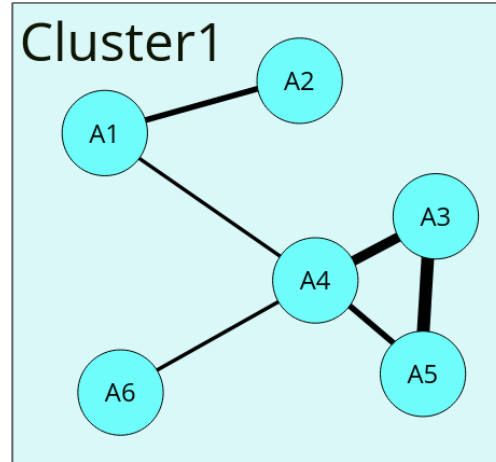


Figure 1.2: Inside cluster graph - Cluster1 was chosen and its artists are shown nodes

Data source

The last but crucial task we need to solve is where will we get the data. As the music domain is truly large, we need some reliable source of data to obtain everything we need. For this purpose, we would like to use some API. The world's most popular and in our opinion the best music service is Spotify. Fortunately, they also provide a REST-based API that can be used for free in non-profit apps. During the development of the app, we found out, this was a really good choice. However, there was a big hidden bottleneck in the form of the request limit.

We will describe this API further in the Spotify API chapter.

2. Graph-based algorithms

2.1 Graph clustering

Data clustering is one of the unsupervised machine learning techniques. The principal idea is that we want to divide the given data among several partitions (the number of clusters may or may not be known). The goal is to separate data in such a way that entities in a certain cluster are similar to each other and, at the same time, different from data in other clusters.

Let us formally define the clustering in the graph domain. Have a graph $G = (V, E)$, where V is set of nodes and E set of edges with weights. Graph clustering divides the nodes into disjoint sets $V_1, V_2, ...$, where $V_1 \cup V_2 \cup ... = V$. The sets will be denoted as *clusters* or *communities*. We can measure the goodness of clustering methods with different metrics. The metrics usually combine weights of edges among nodes inside individual clusters and edges between nodes from two different clusters. Within the scope of our thesis, we were considering two variants of graph clustering: the Louvain algorithm and the spectral clustering algorithm.

2.1.1 Louvain algorithm

For Louvain community detection, we need to introduce modularity that was presented by M. E. J. Newman [1]. Modularity measures how well the network is partitioned into communities. The formula of modularity is:

$$Q = \frac{1}{2m} \sum_{i,j} \left(A_{i,j} - \frac{k_i k_j}{2m} \right) \delta(c_i, c_j),$$

where A is adjacency matrix of the weighted graph, k_i is sum of weights of edges attached to the node i, c_i is the community to which the node i belongs to, $\delta(c_i, c_j)$ is 1 if $c_i = c_j$ and 0 otherwise, and m is sum of weights of all edges in the graph.

The purpose of modularity is to calculate how random the division into communities is. Consider edges to be randomly distributed but the node degrees from the original graph would be kept. Then the expected number of edges between nodes i and j would be $\frac{k_i k_j}{2m}$. To see the significance of the partition, we calculate the difference $A_{i,j} - \frac{k_i k_j}{2m}$ and sum over all pairs of nodes, that ends up in the formula of modularity given above.

The algorithm 1 developed in the university of Louvain tries to maximize the modularity by adding and removing nodes from communities and checking the new modularity value.

Algorithm 1: Louvain algorithm [2]

1 • add every node its own community
2 **while** *improvement exists* **do**
3 **for** *node i* **do**
4 • remove i from its community
5 • add i into the community C of each of its neighbors and compute the change of modularity:

$$\Delta Q = \left[\frac{\sum_{in} + k_{i,in}}{2m} - \left(\frac{\sum_{tot} + k_i}{2m}\right)^2\right] - \left[\frac{\sum_{in}}{2m} - \left(\frac{\sum_{tot}}{2m}\right)^2 - \left(\frac{k_i}{2m}\right)^2\right]$$

6 where $\sum_{in}$ is the sum of weights of the edges inside community C, k_i is the sum of weights of the edges incident to i, $\sum_{tot}$ is the sum of weights of the edges incident to all nodes in C, $k_{i,in}$ is the sum of the edges from i to all nodes in C and m is the sum of all weights of all edges in the graph.
7 • If there is no improvement (ΔQ is smaller than 0), keep i in the original community, otherwise add i to the community, that had the biggest ΔQ.

Although the algorithm iterates through multiple vertices multiple times, it is fast and efficient thanks to the computation of ΔQ. The Louvain algorithm is not deterministic, because it depends on the order of nodes.

2.1.2 Spectral clustering

Spectral clustering is a graph clustering technique using the ideas of graph cut and Laplacian matrix.

Given a graph $G = (V, E)$, adjacency matrix A, set $S \in V$, set $\overline{S}$ is complement of S, define cut as following:

$$cut(S, \overline{S}) = \sum_{i \in S, j \in \overline{S}} A_{i,j}$$

In other words, the cut is the sum of edges going out of the set S to the rest of the graph. The previous equation is considering the partition only into two parts. Let's generalize the equation for k clusters.

$$cut(S_1, ..., S_k) = \sum_{i=1}^{k} cut(S_i, \overline{S_i})$$

For a given k, we would like to minimize the cut to obtain the best clustering. The problem is that even if the cut would be minimal, the results might be weak because cut prefers smaller isolated clusters. We would like to add some condition that would handicap clusters with smaller sizes. Therefore, the $Ncut$ was introduced by Shi and Malik in image segmentation related paper. [3]

$$Ncut(S_1, ..., S_k) = \sum_{i=1}^{k} \frac{cut(S_i, \overline{S_i})}{assoc(S_i)}$$

$$assoc(S_i) = \sum_{v \in S_i} deg(v)$$

Instead of computing only cuts, we add the sum of weights of edges incident to the nodes of the certain cluster. Smaller clusters will have a smaller chance to be included in the final clustering because a small divisor will increase the Ncut value.

Again, we are interested in the minimal value of Ncut. However, Papadimitriou proved (appendix of [3]) that computing the optimal value of Ncut is NP-complete. For finding the suboptimum, we must introduce the following terms:

- matrix D: a diagonal matrix where numbers on the diagonal are the degrees of the nodes in G. Formally: $D_{i,i} = deg(v_i)$.
- adjacency matrix A
- Laplacian matrix: $L = D - A$
- Normalized Laplacian matrix: $L_{sym} = D^{-1/2} L D^{-1/2} = I - D^{-1/2} A D^{-1/2}$

Shi and Malik showed [3], that the second smallest eigenvector is a relaxed solution to the normalized cut problem. In the ideal case, the solution is a vector with two discrete values for two clusters. The relaxed solution gives a vector with real values. As suggested in the paper, we can obtain the clustering by setting a splitting point and dividing the vector values into two parts. The splitting point can either be directly set to 0 or the mean of the vector values or be found as a value where Ncut is the smallest possible.

The algorithm 2 developed by Ng, Jordan and Weiss [4] takes the idea of normalized Laplacian even further. It allows us to use multiple eigenvectors simultaneously.

Algorithm 2: Spectral clustering

1. A - adjacency matrix of the given graph G
2. L_{sym} - normalized Laplacian
3. k - number of clusters
4. • compute first k eigenvectors of L_{sym}: $x_1, ..., x_k$
5. • matrix $X = [x_1, x_2, ..., x_k]$ (the eigenvectors are the columns of X)
6. • matrix $Y_{ij} = X_{ij} / (\sum_j X_{ij}^2)^{1/2}$ (rows of Y are normalized rows of X)
7. • take the rows of Y as points in $\mathbb{R}^k$ and cluster them into k clusters with the use of some clustering algorithm, e.g. k-means
8. • assign every node i to the cluster, to which the row Y_i was assigned

Furthermore, there are some heuristics on how to choose the number of clusters to get better results. One of the heuristics uses the properties of eigenvalues; in particular, it checks the eigengaps between the eigenvalues (the difference between two consecutive items in the array of sorted eigenvalues). If the eigengap between λ_i and λ_{i+1} is large, we take $k = i$. [4]

2.2 Graph visualisation

Graph visualisation is a large part of the graph theory. The goal is to display the graph data with an emphasis on readability and underlining the inner structures that might be hidden on first sight. This is extremely useful in exploring social and other networks properties.

There are a lot of approaches to graph visualization. In this thesis, we set a few rules and we tried to obey them to make the visualization "look good":

- The whole graph should be visible after the loading is finished and zoom should be enabled.
- The nodes shouldn't be overlapping (small overlaps would be tolerated).
- The inner structure (clusters) should be pointed out clearly.
- The less crossing edges we have, the better.

To achieve these goals, we must combine a few algorithms. First, we will compute the basic layout of the given graph. We will use a force-directed layout. Unfortunately, the calculated positions might not reflect the size of the nodes and there might be overlaps. The node positions adjustment will be done with the help of the growing tree algorithm.

2.2.1 Force-directed layout

The goal of force-directed layout algorithms is to provide a well-balanced, visually satisfying layout with a few crossing edges. The algorithm tries to avoid too long edges; it means the nodes belonging to one edge won't be spread too far.

The basic idea of the force-directed layout is to think about the graph as of a system where edges between nodes are replaced with springs. We model the forces that act on the nodes and let the system get into more or less equilibrium state in several iterations. One of the best-known force-directed layout algorithms is the Fruchterman-Reingold algorithm. [5]

In the algorithm, we will use the following notions and principles:

- Repulsive force function and attractive force function calculate a repulsive and an attractive forces between two nodes based on their distance, area of the graph and the number of nodes in the graph.
- Every node v consists of two vectors - $v.pos$ is the current position of v and $v.disp$ is a displacement vector by which the $v.pos$ vector will be updated.
- During the algorithm, we run several iterations. In every iteration, the displacement vector is calculated for every node. The displacement vector is influenced by repulsive forces (the node tends to get far from every other node in the graph) and attractive forces (the node tends to get close to its neighbours).
- We also add the idea of temperature. At the beginning of the algorithm, the system is "hot" and nodes tend to change positions more rapidly. In the end, the temperature cools down and position change is reduced.

Algorithm 3: Fruchterman-Reingold algorithm [5]

```
1  k = √(area/|V|)
2  f_r(x) = k²/x                                  /* repulsive force function */
3  f_a(x) = x²/k                                  /* attractive force function */
4  for i=1 to iterations do
5      /* calculate repulsive forces: */
6      for v ∈ V do
7          v.disp = 0                             /* displacement vector */
8          for u ∈ V where u ≠ v do
9              Δ = v.pos − u.pos
10             v.disp = v.disp + (Δ/|Δ|) ∗ f_r(|Δ|)
11     /* calculate attractive forces: */
12     for (v,u) ∈ E do
13         Δ = v.pos − u.pos
14         v.disp = v.disp − (Δ/|Δ|) ∗ f_a(|Δ|)
15         u.disp = u.disp + (Δ/|Δ|) ∗ f_a(|Δ|)
16     /* update the nodes positions: */
17     for v ∈ V do
18         v.pos = v.pos + (v.disp/|v.disp|) ∗ min(|v.disp|, t)
19         if v.pos outside of frame then
20             move v.pos inside the frame
21     t = cool(t)                                /* reduce temperature */
```

2.2.2 Growing tree overlapping nodes removal

Growing tree (also denoted as GTree) is an algorithm for node overlaps removal. [6] After we obtained layout from a certain graph layout algorithm, we must deal with the fact, that during our visualization, the nodes have non-trivial width and height. It happens very often that one node covers more than half of another node and makes the resulting graph very chaotic. GTree algorithm removes the overlaps with the help of Delaunay triangulation and minimum spanning tree (*MST*) algorithm. The name comes from the behaviour of the algorithm; it iteratively increases the lengths of the edges and the graph "grows". Let us go through the terms and ideas the GTree uses.

The Delaunay triangulation is a triangulation where the following property holds: the circumcircle of each triangle does not contain any point in its interior.

Now, let us define the cost function c that we will use further. The cost function takes two nodes; their centre points and also 2D shapes (widths and heights). In other words, we are considering nodes as rectangles around their centre points.

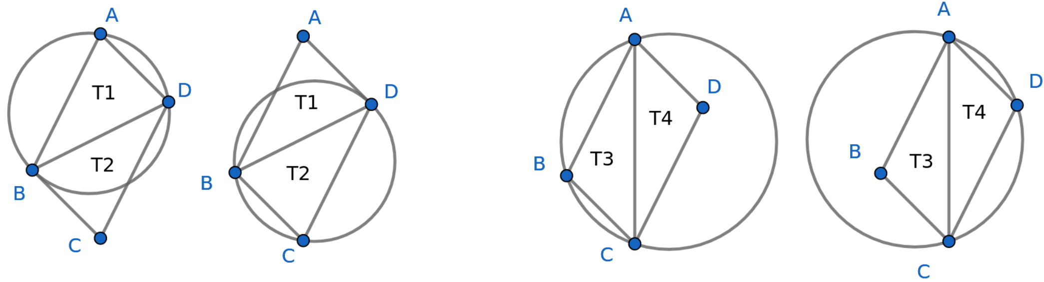


Figure 2.1: {T1,T2} is Delaunay triangulation of {A,B,C,D}, {T3,T4} is not (points B and D lie in circumcircles of T3 and T4)

The value of the cost function for nodes i and j depends on if they are overlapping or not. If there are no overlaps, the $c(i, j)$ is simply the distance between the node rectangles, i.e. the distance of their closest points. If the nodes overlap, we compute the cost function from the following values:

- scalar $s = ||p_i - p_j||$, where p_i and p_j are positions of nodes i and j
- scalar d is a distance that nodes i and j would have if they would be shifted in the direction of vector $p_i - p_j$, such that their rectangles would touch with one of their sides
- scalar $t_{i,j}$ such that $d = t_{i,j}s$
- cost function value: $c(i, j) = d - s$

These parameters are illustrated on the figure 2.2.

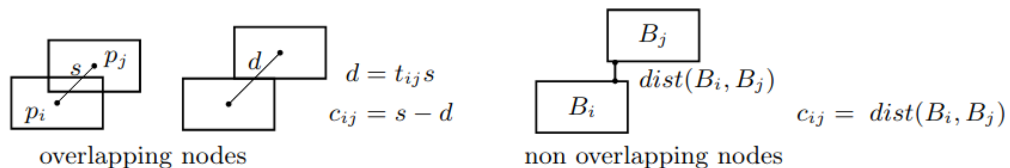


Figure 2.2: GTree cost function, source: [6]

The algorithm's input is the initial positions given by some graph layout algorithm. The positions are the centres of the given nodes. We do several iterations until there are no overlaps. In every iteration, we obtain a Delaunay triangulation of the points. To every edge of the triangulation, we assign weight calculated by the cost function described above. Then we run an MST algorithm (for example the Kruskal's algorithm [7]) on the triangulation with weights. This ensures us, that the edges of overlapping nodes are contained in the MST. As a last step of the iteration, we recursively grow the tree. Growing tree means starting at one of the nodes and prolonging too short edges, thus moving too close nodes further apart and therefore removing the overlaps. At the same time when a certain edge is enlarged, the whole sub-tree is moved further away and the original layout shape is practically preserved.

Algorithm 4: GTree algorithm [6]

```
1  ∀i : p_i - centre of node i ∈ V (from the precomputed layout)
2  c - cost function on edges as defined above
3  while Ending condition do
4      D - Delaunay triangulation of the set p_i
5      E - Edges of D, with edge weights computed with c cost function
6      • T - minimum spanning tree on E (Prim's algorithm is suggested)
7      • r - arbitrary node of T
8      • prepare new positions: p'_r = p_r
9      • call recursive GrowAtNode(r, p, p')
10     • ∀i : p_i = p'_i (change the positions to the computed ones)
11 • return positions p
12 Function GrowAtNode(i, p, p'):
13     foreach j ∈ Children(i) do
14         p'_j = p'_i + t_ij(p_j − p_i)
15         GrowAtNode(j, p, p')
```

The *Ending condition* in the `while` cycle on the line 3 is true when there are overlapping nodes in the graph. In our modification, if the algorithm didn't finish in a certain number of steps, we will stop the iterations even if there are some overlaps.

The `GrowAtNode` function has complexity $O(|V|)$. In our modification, we used Kruskal's algorithm for MST, that was implemented in Python library networkx. The complexity of Kruskal's algorithm is $O(|E|log(|V|))$.

3. Spotify API

Spotify API is accessible for all developers for free and it provides a large amount of music data and metadata. The data can be obtained by sending REST requests to an API endpoint. In this chapter, we will describe some methods and principles we used. All the information was taken from the official Spotify API documentation page [8] or GitHub Issues page [9].

The first action that needs to be done is creating an application on the page for Spotify developers. We created an application called SpotifyGraph. For every app, there exist two keys generated by Spotify API - Client ID and Client Secret. The second one shouldn't be accessible to anyone but the application. Both keys are needed during user authorization.

3.1 Authorization

As mentioned in the authorization guide [10], there exist three possible authorization flows:

1. **Authorization Code Flow** (see fig. 3.1) is the most complex and universal way to authorize a user. It requires both client ID and secret key, that are exchanged for access and refresh tokens in several steps.
2. **Implicit Grant Flow** (see fig. 3.2) is meant to be used in the front-end JavaScript apps with no need for server-side code. Application sends Client ID to the Spotify endpoint, and an access token is returned.
3. **Client Credentials Flow** (see fig. 3.3) is used, when no user's information such as favourite songs or account details is needed. On the other hand, according to the documentation, the rate limit should be higher. The type of authentication is server-to-server.

In our app, we chose to use the Authorization Code flow, because of the following reasons:

- We need to obtain the user's data (his/her favourite songs and followed artists). Therefore, the Client Credentials is not a suitable solution.
- Even though we send most of the requests directly from the front-end, we want to keep the possibility to control authorization also from the back-end. Thus, we exclude Implicit Grant Flow. Authorization code has also possibility to use refresh token which is not the case of Implicit Grant.
- We also thought about using both Authorization Code Flow and Client Credentials Flow, because the higher limit could be useful for obtaining bigger amounts of data, as we will mention below. However, for simplicity, we used only the Authorization Code flow.

3.2 Spotify Data

Developers can access all kinds of Spotify data. From now on, we will use words track and song as synonyms (track is used in the official API documentation). In our application, we send the following requests to obtain the desired data. The exact API URLs and parameters are omitted for clarity:

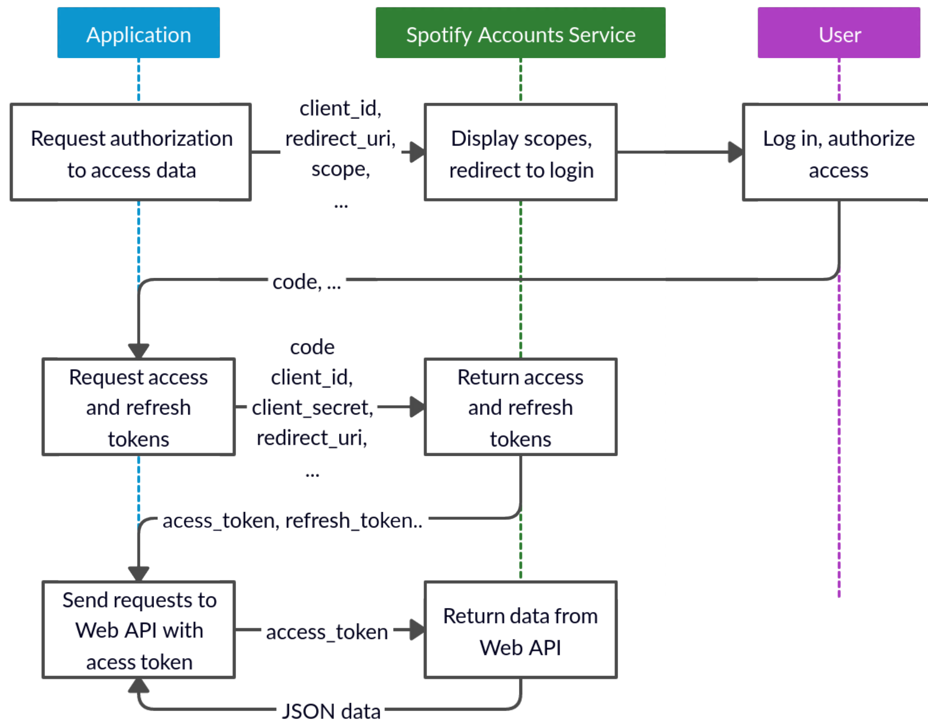


Figure 3.1: Authorization Code Flow

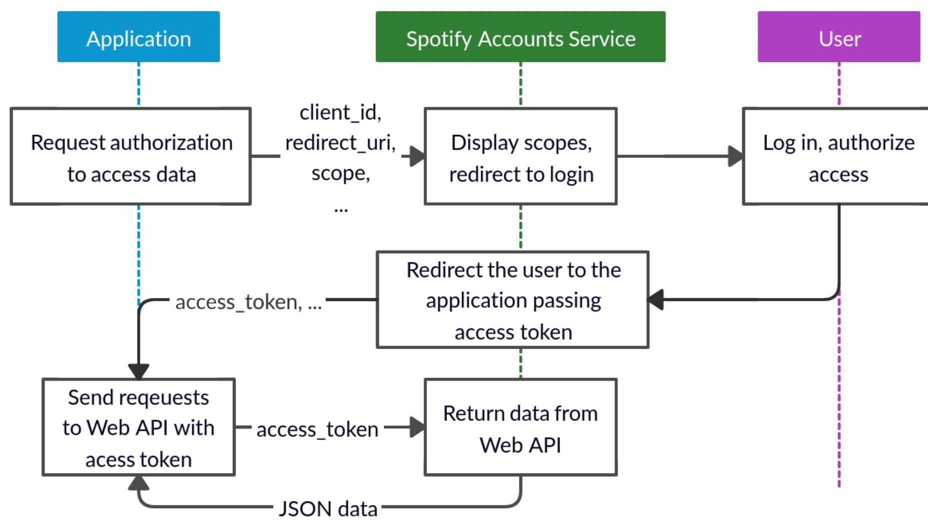


Figure 3.2: Implicit Grant Flow

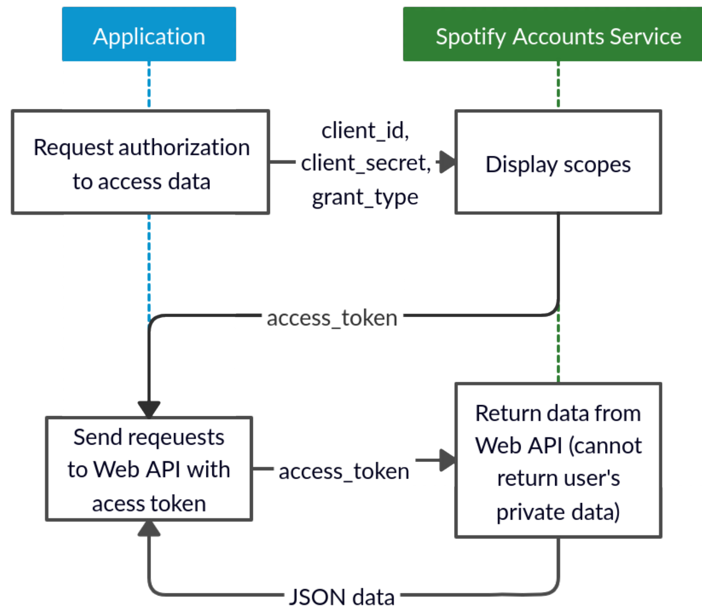


Figure 3.3: Client Credentials Flow

- **Get a User's Saved Tracks** - returns songs, that were liked by the user usually by clicking on the heart symbol.
- **Get User's Followed Artists** - returns artists followed by the user.
- **Get Artist's Related Artists** - returns IDs of artist's similar artists. These artists are computed by Spotify algorithms based on the community's listening history [11]. This information is also shown in the official Spotify applications on artists' pages under the "Fans also like" section.
- **Get a Track** - returns general information about the requested song. This includes album information, featuring artists, popularity, links to song cover image, 30 seconds preview URL, etc. There is also **Get Audio Features for a Track** endpoint that returns music features like loudness, danceability or key. We didn't use this option in this thesis.
- **Get an Artist** - returns general information about the requested artist, it means image links, popularity, number of followers and genres in particular.
- **Get Several Artists** and **Get Several Tracks** - methods that return information about multiple artists, respectively tracks by given IDs. These methods are extremely useful and can significantly increase the speed of data fetching. E.g., if we wanted to get 100 songs, we would have to send 100 request on the Get a Track, but only 2 on the Get Several Tracks endpoint.
- **Get an Artist's Top Tracks** - returns a list of up to 10 tracks that have the highest popularity according to Spotify.

3.3 Spotify API limitations

In our thesis, one of the key information we are obtaining from the Spotify API endpoints is artists' related artists. As mentioned in the list above, we can reach a certain artist's related artists with one HTTP request. But most of the users have a big number of favourite artists and for each of them, we need to call one request, thus the number of requests might be in hundreds.

There are three solutions to this problem. The first one is to cache the related artists to our database. We didn't proceed with this option, however, we will provide a description of this option as a potential extension in the "Further Work" chapter. The second solution was based on the hope that the Spotify Developers would add an endpoint that would be similar to getting multiple tracks; something like "Get multiple artists related artists". Unfortunately, till now after a few months, our request on the official Spotify GitHub issues page was not heard and it does not seem probable that it will ever be. Therefore we had to use the third solution and that means asking one by one. During our testing we observed that fetching 100 artists' related artists takes about 15 seconds.

Another unpleasant behaviour appears when Spotify API returns empty preview URLs for relatively big amount of songs. This is caused by Spotify's market policy. It can happen that a certain song has different ID for different markets or possibly not be available on a certain market at all. Sometimes it also happens that non-premium users receive much less mp3 previews for copyright issues. Another reason is that some songs just do not have a preview URL. To deal with this painful aspect of Spotify's API, we decided to let user know that the song is not available. We could have used a track relinking option (it matches the different markets song IDs) but this would require another requests and we decided not to proceed in this direction. First, it might slow down the loading of songs, second, the results might not be satisfying, due to the premium account policy.

4. Application Design and GUI

Let us now describe how the final design of the application looks like. We will provide screenshots of various parts of the page and high-level description of how the algorithms were used and adapted for our situation.

4.1 Architecture overview

Our application runs on two servers: the front-end server and the back-end server. Front-end server serves web pages with the visualization, back-end server is responsible for graph calculations, such as graph clustering and graph layout computation. The back-end is working as a REST API, that receives HTTP requests with parameters, runs implemented methods with necessary graph algorithms and returns results that can be used in the front-end. Without back-end server running, the application won't be functional. We will get back to the technical point of view and code details in the Code Overview (Chapter 5).

4.2 Inicialization

First, we will describe the standard sequence of steps that the user does, when he/she enters the web page and uses the application (see figure 4.1). We will look closer at individual parts of the page in the sections that follow.

1. The user visits the home page. He/she can see a short description of the application and by clicking at the "continue" button is redirected to the login page.
2. The login page informs the user what will happen next and gives the user the only possibility to proceed by clicking the "login" button.
3. After clicking the "login" button, the user is redirected to Spotify login page. If the user is already logged in the browser, the Spotify login page will be skipped. The user logs in and if he/she is using the application for the first time then the scope will be shown and he/she must agree that our application will get access to his/her data. If the login didn't succeed for an arbitrary reason, the user is informed about it by alert and is redirected back to the login page.
4. After successful login, the main page appears. At this moment, we send requests to Spotify Web API and get necessary information about artists, songs and related artists. The progress is visualized in the form of a progress bar.
5. If the Spotify data was loaded without any problems, the front-end sends the artists data to the back-end and receives a graph of the given artists. The graph contains information about edges, clusters and node positions.
6. Now, everything is set and the graph can be visualized and the user can explore the clusters, artists and use other functionalities that will be described in more details further on.

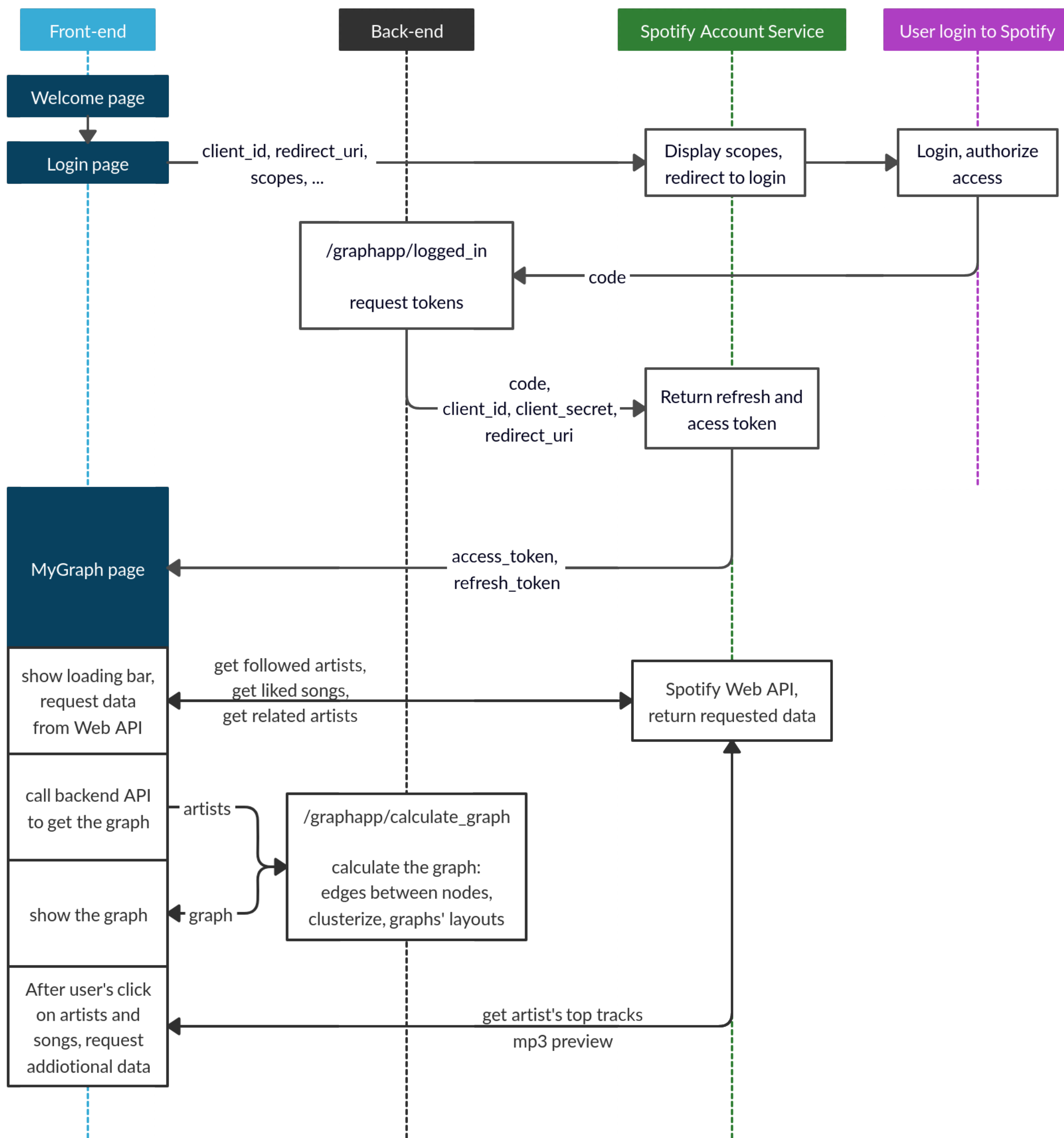


Figure 4.1: Application flow

4.3 Graph construction

4.3.1 Artists

In the beginning, we must obtain data from Spotify API. With the requests, we are able to put together the following data for every artist:

- ID - Spotify artist ID
- name - artist's name
- genres - list of artist's genres
- related artists - list of related artists
- image - link to the artist's image
- liked songs - list of songs, that is liked by the current user
- is followed - a binary value indicating if an artist is followed by the current user
- popularity - popularity of an artist given by Spotify

It would be convenient to have a certain score for every artist. We could sort artists according to this score and show more important artists first. We want the score to be personalized, that's why we take into account the number of liked songs, the fact that the artist is followed or not and popularity among other Spotify users. We set such a score in the algorithm 5.

Algorithm 5: Artist score

```
1 Function ComputeArtistScore(artist, lsConst, popConst, follConst):
2     lsScore = artist.likedSongsByUser.length() * lsConst
3     follScore = artist.isFollowedByUser ? follConst : 0
4     popScore = artist.spotifyPopularity * popConst
5     return lsScore + popScore + follScore
```

We had to make a choice about the parameters. We wanted to give the highest importance to user's preferences but also add a little importance to artist popularity. The Spotify popularity is the number from 1 to 100.

We set `follConst` = **5**, `lsConst` = **1** and `popConst` = **0.1**.

4.3.2 Edges among artists

After we have the list of artists, we want to create edges among them from the information we gained from Spotify. We decided to use two properties: related artists and genres.

Let's say we want to compute the weight of an edge between two artists. We will use a formula that takes into account if the first artist belongs to the related artists of the second one and vice versa. The formula also places importance on the number of common genres of the two chosen artists. The exact calculation is captured in the algorithm 6.

Algorithm 6: Weight of an edge between two artists

```
1 Function ComputeEdgeWeight(a1, a2, relArtistConst, genreConst):
2     r1 = (a1 ∈ a2.relatedArtists) ? relArtistConst : 0
3     r2 = (a2 ∈ a1.relatedArtists) ? relArtistConst : 0
4     g = intersection(a1.genres, a2.genres).length() * genreConst
5     return r1 + r2 + g
```

We had to choose the value of two hyperparameters:

- `relArtistConst` says how important is Spotify information about related artists. This value is very important to us because the concept of related artists is based on the history of thousands of users. We set this value to **1**.
- `genreConst` is less important, because it only compares the genres of two artists. We set this value to **0.2**.

4.3.3 Clustering

Artists and edges between them form a basic graph. However, we do not want to directly present this graph to users. There might be too many nodes and edges and we wouldn't provide any additional information about the data. Therefore, we will split artists into groups and inform users about deeper structures that are present in their graphs. Let's have a look at the exact algorithm we will use to divide artists into clusters.

First, we divide the graph into connected components.

We take all artists that are alone in their components, i.e. they have no neighbours. We put all these artists into a cluster named "TOTAL MIX". It might happen that this cluster will contain relatively lot of artists, but let us remind that we do not have any chance to assign these artists into any cluster because they have no neighbours.

Thanks to the previous step, only components with more than 1 artists are remaining. Now, we prepare three values:

- `starting threshold`
- `minimum artists threshold`
- `maximum artists threshold`

For every connected component, we will run the following steps that use the prepared thresholds:

1. Check, if the component has enough artists to be worth splitting. If the number of artists in the cluster is smaller than the `starting threshold`, we will return the whole component as a cluster. Otherwise, we proceed to the next step. This condition is important because we do not need to split components with a few artists, e.g. component with 5 artists is too small to be divided into clusters.
2. If the component is large enough to be split, we run a certain clustering algorithm and obtain a number of clusters. We decided to use the spectral clustering algorithm. This choice will be justified in the Code Overview (Chapter 5). However, the algorithm can be simply replaced and we can

suppose for now that the algorithm returns a few newly created clusters. For each newly obtained cluster, we check if its size is bigger than the `maximum artists threshold`. If the size is bigger, we split the cluster again and continue until there is no cluster bigger than the `maximum artists threshold`.

3. In the previous step, we obtained a list of clusters. We want to check that none of them is too small, i.e. smaller than the `minimum artists threshold`. For every cluster smaller than the threshold, we put all its artists into the closest cluster. Cluster B is the closest to S if

$$\sum_{u \in S, v \in B} weight((u, v))$$

is the biggest of such sums of all neighboring clusters of S.

Note, that after the clustering is finished, it might happen that some clusters' sizes will be greater than `maximum artists threshold`. When we are getting rid of too small clusters, we may add the artists into a larger cluster and increase its size so it's greater than the threshold. This is not a serious problem because the priority of having a few small clusters if bigger than of having a few clusters that are too huge.

Algorithm 7: The graph clustering algorithm

```
1  Function clusterize(component):
2      if component.artists.length() ≤ starting threshold then
3          return cluster(component.artists)
4      clusters_queue ⟵ get_clusters(component)
5      clusters ⟵ []
6      while clusters_queue is not empty do
7          current_cluster = clusters_queue.pop()
8          if current_cluster.length() ≥ maximum artists threshold then
9              new_clusters ⟵ get_clusters(current_cluster)
10             clusters_queue.push(new_clusters)
11         else
12             clusters.push()
13         end
14     end
15     foreach cluster in clusters do
16         if cluster.artists.length() ≤ minimum artists threshold then
17             put all the artists from cluster to the closest cluster
18         end
19     end
20     return cluster

21 clusters_to_return = []
22 for component in G do
23     clusters_to_return.extend(clusterize(component))
24 end
```

The `get_clusters` function in the algorithm 7 is one of the clustering algo-

rithms mentioned in the Algorithms chapter 2.

We have to decide, what will be the values of our three thresholds. First, we were thinking about setting all three values fixed. This approach turned out to be not really flexible for diverse graph sizes. For example, for smaller graphs, the `maximum artists threshold` value can be around 15. But this value wouldn't be very suitable for graphs with 1000 favourite artists because it could create a big amount of clusters that wouldn't fit on the screen. Therefore, we set the following values of thresholds:

- `starting threshold` $= 20$
- `minimum artists threshold` $= 5$
- `maximum artists threshold` $= max(15, \lfloor artists.length()/10 \rfloor)$

4.3.4 Edges among clusters

As mentioned in the Analysis (Chapter 1), Main graph's nodes are clusters and edges are the similarities among them. The exact weight of edge between two clusters C_1 and C_2 is calculated as sum of edges that have one node in C_1 and second in C_2. Particularly: $weight(C_1, C_2) = \sum_{u \in C_1, v \in C_2} weight((u, v))$.

4.4 Layout

There are two types of graphs we will need to visualize: Main graph and Inside cluster graph. For both of them, we will use slightly different methods, because both of them have different requirements. Let us describe differences between two layout algorithms that will be used to calculate positions of nodes in both types of graphs.

4.4.1 Main graph layout

We described the Fruchterman-Reingold algorithm and node overlap removal algorithm in the Graph-based Algorithms (Chapter 2). However, these algorithms do not completely fit our goals. The problem is with components, that have one or only a few clusters. Let us denote the clusters from small components as **lonely clusters**. If we include lonely clusters to an input of the graph layout algorithms, they could occupy too much space that should be given to bigger components. In other words, bigger components need more space to be fully shown with as least overlaps as possible and the browser size is limited. In the force-directed algorithm, it could happen that lonely clusters would make bigger components shrink and the structure wouldn't be shown very well.

Moreover, small components' graphs are easy to visualize, as there are not many possibilities on how to arrange them. For example, the component with two clusters will be depicted as two clusters next to each other connected with one edge and no complex algorithm has to be run to find a nice layout.

Let us define what a small component means. We decided to label a component with at most 3 clusters as small. This threshold was set mainly for the simplicity of visualization and small space that these components require.

Finally, the complete algorithm computing the positions of the clusters in the Main graph will go through the following steps:

1. Separate all components, that include at most 3 clusters and put them into a list of small components.
2. Run a layout algorithm (e.g. Fruchterman-Reingold) on clusters of remaining components and remove overlaps with the Growing tree algorithm.
3. Fit the calculated positions from step 2 to the width and height of the div in the page where the graph is supposed to be.
4. Create a rectangular mesh. Size of each rectangle is a preset size of a cluster plus a small margin. Let's say you can fit w rectangles on the width and h rectangles on the height. Make a binary matrix $M = \{0, 1\}^{(h,w)}$ where $M_{i,j} = 1$ where there exist an overlap of rectangle and some cluster that is already placed and $M_{i,j} = 0$ where there is no overlap with rectangle on the position i, j.
5. For every small component in the list we created in step 1, we apply the following steps:
   (a) Try to find a space in the mesh for the small component.
   (b) If space was found, place the component there and update the matrix on the positions where the component was added.
   (c) If no space was found in the mesh, place the component under the graph and if necessary, enlarge the height.

In figure 4.2 we can see step 5 of the previous algorithm. Note that components 6,7 and 8 cannot be fit into the rectangle, therefore the height is enlarged while the width remains the same.

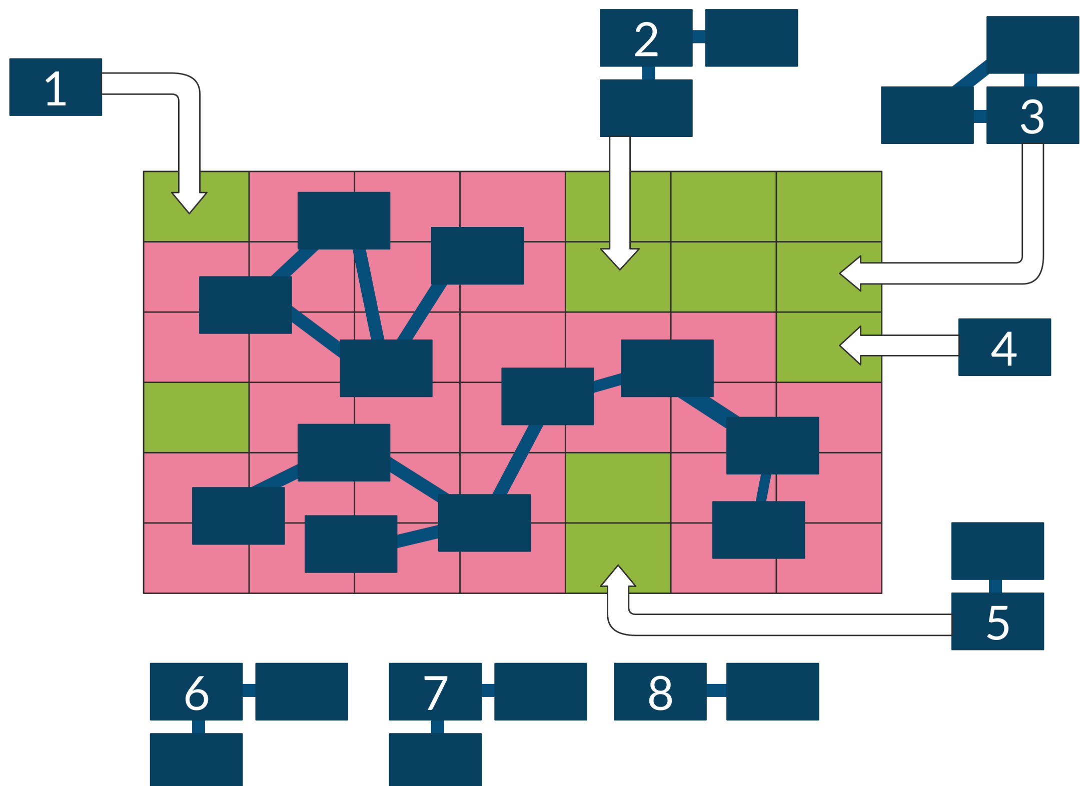


Figure 4.2: Fitting small components into the layout

4.4.2 Inside cluster graph layout

In the case of an inside cluster graph, the situation is easier. First, artists inside clusters are typically connected with edges and there are not many lonely nodes. Second, nodes in an inside cluster graph will be much smaller and there won't be too much struggle to fit them all in the window. For these reasons, we decided to run a force-directed layout algorithm (again Fruchterman-Reingold) and the growing tree to remove overlaps. Positions calculated by these algorithms will be taken as a final layout and no additional steps will be added.

4.5 GUI elements

Suppose that all computations were successful and we want to show the results to the user. Let us remind the main idea of our graph visualization. There are two types of graphs we want to visualize: Main graph contains clusters and edges among them and Inside cluster graph shows artists and edges among them after a certain cluster is chosen. Both graphs are depicted in figures 4.3a) and 4.3b). In both figures, there are numbers pointing at individual parts of the page. We will go through all of them and describe their functionality.

1. Main graph GUI
2. Main graph nodes (clusters)
3. Inside cluster graph
4. Artist bar
5. Similar clusters
6. Left side-bar
7. Player

4.5.1 Main graph GUI

The main graph's nodes are representing the clusters. Edges represent the similarities among clusters. The clusters' positions were calculated with the algorithm described in the Layout (Subsection 4.4.1). Example of users' Main graphs will be shown in the Case Studies (Chapter 6).

Users are allowed to do several actions with the graph:

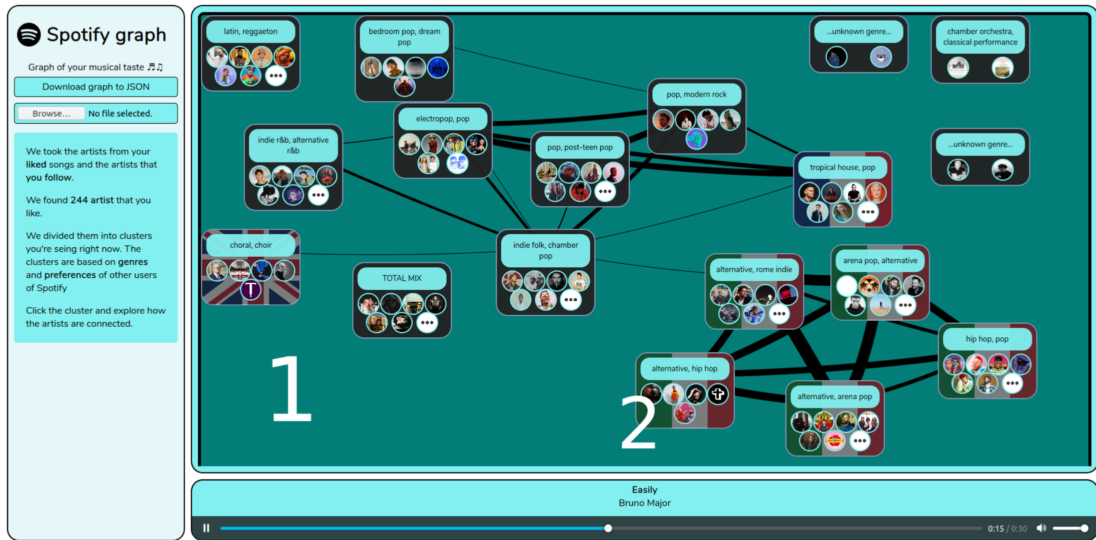
- zoom the graph with mouse/touch-pad scrolling
- move clusters around
- click at the clusters - after the cluster is clicked, the main graph is hidden and an Inside cluster graph appears.

4.5.2 Main graph nodes (clusters)

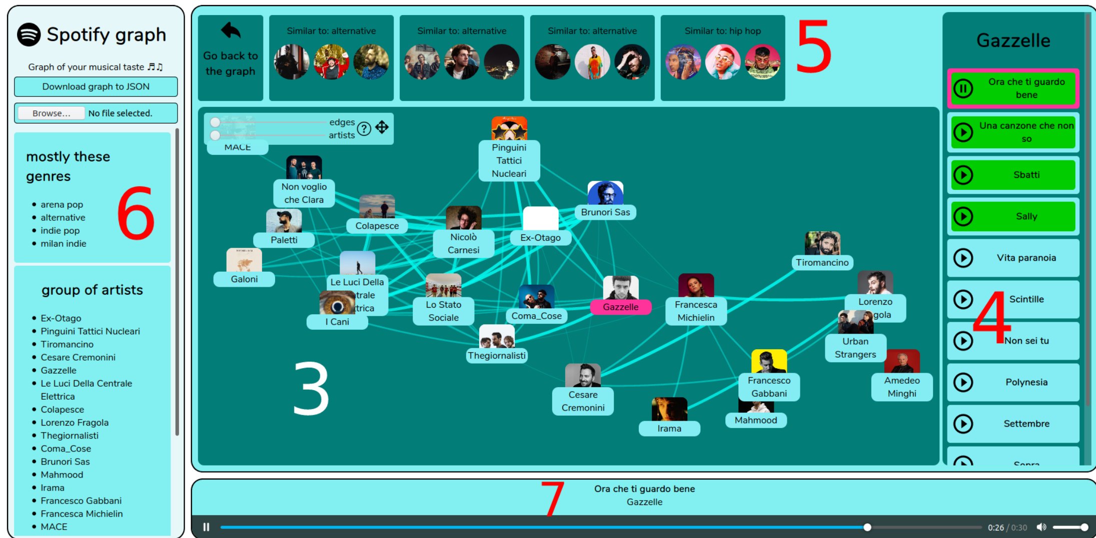
Every cluster contains information about three cluster properties: cluster description, cluster artists and country.

Cluster description

The cluster description is basically a list of the most common genres of artists in the cluster that were sorted and organized according to the algorithm 8. In



(a)



(b)

Figure 4.3: Two types of graphs: (a) Main graph (b) Inside cluster graph

the future, the cluster description could be extended with additional information, but for now, we will rely only on genres.

Algorithm 8: The cluster description calculation

```
1  Function ComputeGenres(artists):
2      genres ⟵ concatenate lists of genres of all artists in the cluster
3      for i ← 0 to genres.length() do
4          genres[i] ⟵ removeCountryName(genres[i])
5      end
6      genres ⟵ sort genres by frequency of genres and remove duplicates
7      genres_to_return ⟵ []
8      dictionary_of_words ⟵ {}
9      foreach genre in genres do
10         words ⟵ genre.split(' ')
11         if every word in words is in dictionary_of_words or ∃ word in
            words: dictionary_of_words[word] > 2 then
12             don't add the genre to the final list
13         else
14             genres_to_return.add(genre)
15             dictionary_of_words.update(words)
16         end
17     end
18     return genres_to_return
```

Line 2 ensures that every genre of the artists will be taken into account. The lines 3-5 remove the cases where a genre is in fact duplicated, once with a country name and once without; e.g. *italian pop* and *pop*. Line 6 ensures that the most common genre of the cluster will certainly be present in the description.

The condition on the line 11 assures us that the words won't repeat too often; e.g. if the `genres_to_return` already contain *indie pop* and *pop rock* the first part of the condition will exclude *indie rock* and the second will exclude *teen pop*.

In practice, the algorithm 8 can return a lot of genres but we do not need many of them. This is due to the fact that the space for cluster description is limited and generally speaking, users will pay more attention to the cluster description if it is short enough. For these reasons, we decided to show only 2 genres in the initial view. If a user zooms the main graph, more genres will appear in the cluster description. We set the maximal number of enlisted genres to 5. This behaviour is shown in the figures 4.4a and 4.4b.

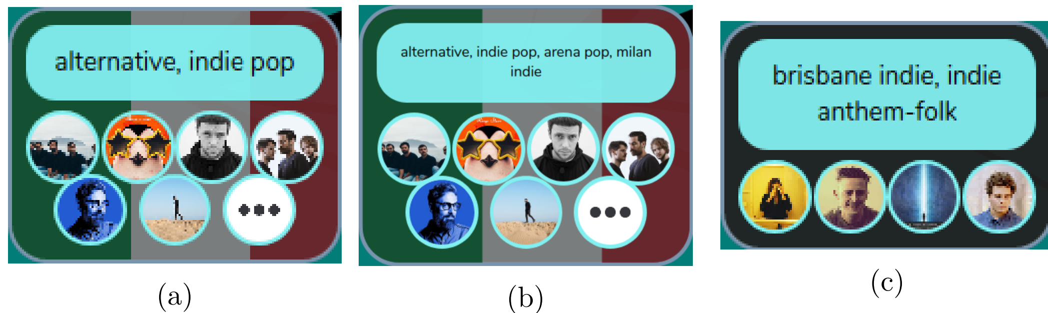
Cluster artists

The artists in the cluster are visualized as one or two lines of rounded images. Artists are sorted by a score, that was computed by algorithm 5. The maximal number of shown previews is set to 7 and if the number of artists in the cluster is bigger than 7, it is indicated by three dots (e.g. figure 4.4a).

Country of origin

If at least 40% of the artist comes from a certain country and it is explicitly mentioned in their genres, the flag of the country is set as a background of the

cluster. Generally speaking, the country is not always mentioned; especially US singers do not inform about their country of origin in their genres. However, the country of origin sometimes adds interesting information about the user's music taste.



(a) (b) (c)

Figure 4.4: (a) A cluster with more than 7 artists, mostly from Italy (b) Cluster after zoom with more genres (c) A cluster with less than 7 artists, without explicitly mentioned country of origin

4.5.3 Inside cluster graph

After a certain cluster is clicked, the inside cluster graph with other elements appear. This time, every node represents one artist and edge between them has weight computed by algorithm 6.

There is also an element for controlling the number of artists and edges that are shown in the graph. This element is by default in the top left corner and can be moved by grabbing the symbol in the right part.

Sometimes, there can be a lot of artists and edges in the cluster. This happens when users have a big amount of favourite artists. With too many edges, the rendering slows down. For this reason, we set a maximal number of edges that will appear by default. Users can change the slider so every edge will appear. However, when the Inside cluster graph appears, this threshold will prevent too slow rendering.

4.5.4 Artist side-bar

Sometimes it happens that users do not recognize some of the artists in their graphs. This confusion might occur when a user saved a song to his/her liked songs a long time ago. Sometimes, several artists cooperate on a certain song and a user knows only one of them. To make the appearance of artists in the graph clearer, after clicking an artist, the artist side-bar appear.

On top of the panel, there is the artist's name and information if a current user follows this artist. Below this information, songs by the chosen artists are shown. First, the current user's liked songs are enlisted and after them, the top songs by the artist follow. Let us describe how the song elements will be visualized.

Song element

The right panel offers certain tracks interpreted by the artist. There are two kinds of songs, distinguished by colour. Songs highlighted by bright green colour

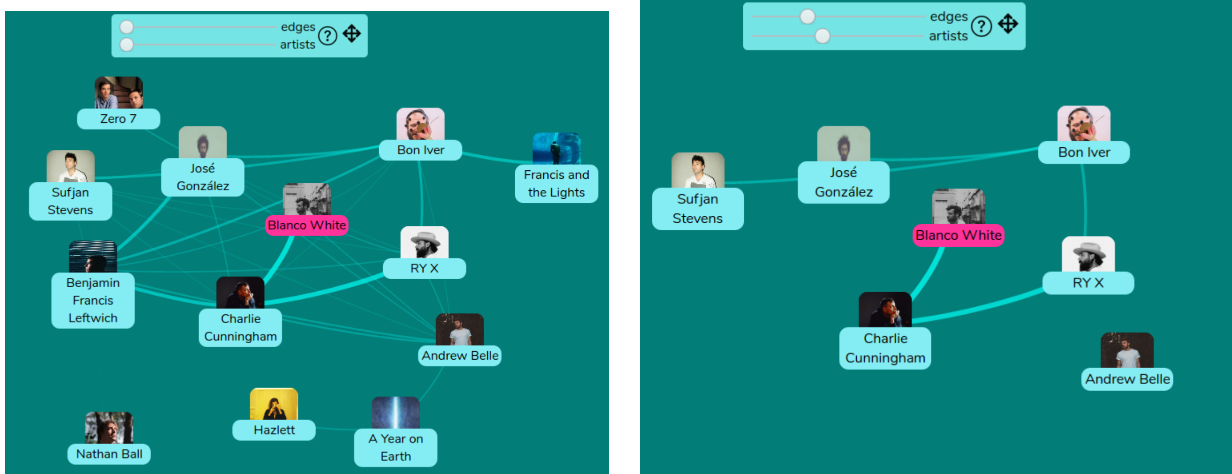


Figure 4.5: The inside cluster graph with indie artists. Right image shows the usage of slider that hides less important edges and artists.

are already liked by the user. The rest of the songs have azure colour and they were taken as top tracks of the artist. The top tracks were computed by black-box Spotify algorithms and they are usually the best-known pieces by the author.

Every song element has the following functionalities:

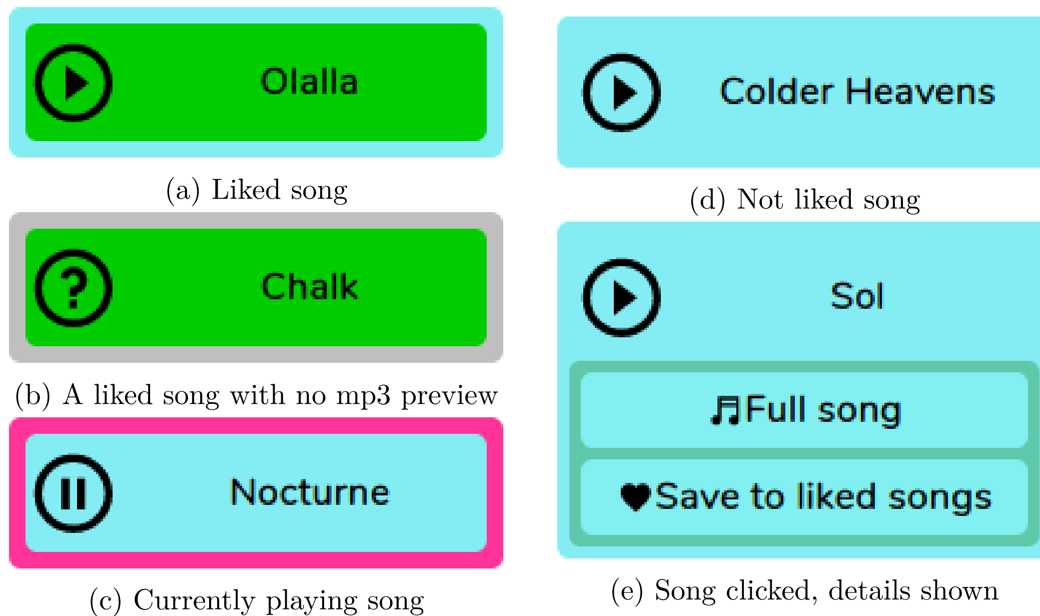
- After clicking the icon with "play" symbol, 30-second mp3 preview is played in the bottom of the page. If the user leaves the cluster details or clicks another artist, the preview will keep playing until it ends or until another preview is played. The played song is highlighted with pink colour.
- After clicking the song element, the div with two options pops up:
  - Button that opens the Spotify web player in a new browser tab and allows the user to play the full song.
  - Button that saves the current song to the liked songs.

All song states are shown in the figure 4.6.

Let us discuss, why we didn't implement the possibility to play the full song in the browser. This function would be provided by Spotify's Web Playback SDK. [12] As mentioned in the documentation, only premium users can play the full song using SDK. According to the Spotify info page, almost half of the Spotify users pay for the service. [13] This divides potential users of our application into two big groups that need to be managed. We were thinking about two solutions to this problem:

1. Premium users would play the whole song and non-premium would play just the 30-second preview.
2. For all users, only the preview will be available.

We decided for the simple variant, where all users will play only previews. A fully functional web player is out of the scope of this bachelor thesis. On the other hand, there might be done some further progress in this direction.



(a) Liked song

(b) A liked song with no mp3 preview

(c) Currently playing song

(d) Not liked song

(e) Song clicked, details shown

Figure 4.6: Song state examples of Blanco White's songs

4.5.5 Similar clusters bar

On the top of the page with Inside cluster graph, the similar clusters are shown. The clusters are sorted by the similarity, the most related clusters first. After clicking some of the similar clusters, the details of the chosen cluster will appear instead of the current one.

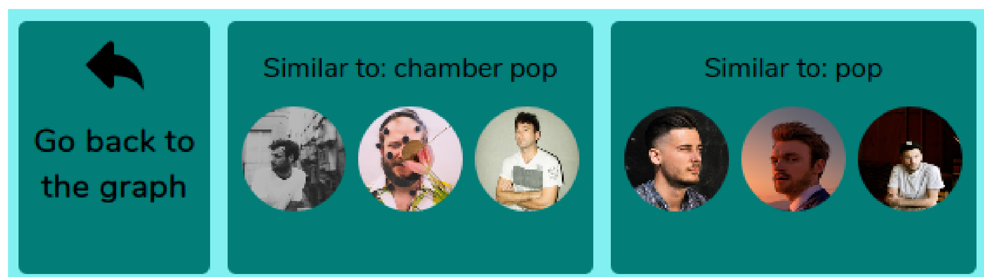


Figure 4.7: Panel with similar clusters

4.5.6 Left side-bar

Let's say, a user is excited by our application and wants that his/her friends use it too. He also wants to compare graphs and let his/her friends know how his/her graph looks like. For this purpose, we added the functionality of downloading and uploading the graph data as a JSON file.

For this purpose, there are two buttons on top of the left side-bar. The "Download graph to JSON" button servers for downloading the graph data as a JSON file. The second button allows uploading a JSON file with graph data. If the file is corrupted, an alert with error information pops up and the current graph remains on the page. If the file was correctly loaded, the new graph from the file is shown.

Under these two buttons, there is information either about the main graph or about the chosen cluster.

4.5.7 Player

In the bottom part of the page, there is a simple HTML5 Audio player element. It has basic functionality as play/pause and navigation in the music played. The name of the played song and artists are shown too.

4.6 Graph caching

The graph loading is the most annoying part for users. They have to wait until all of their favourite artists are loaded. The more artists a user has, the longer it takes to send requests to Spotify API (related artists requests take most of the loading time). As this process takes so much time, we want to avoid reloading already obtained data to make the initialization less painful for users.

There are two possibilities on how to speed up the initialization procedure. The first one would save the related artists information in a database. It will be described further in the Future Work (Section 6).

We decided to implement the second option - caching the related artists information in a browser. We used local storage that enables us to save and load data on the page. If users refresh the page, it is necessary to login again, but this time, the related artist loading will be much faster and users more satisfied.

5. Code Overview

5.1 Back-end

The back-end server is written in Python3. In the current version, no database is used because we are not caching users' data. Our back-end works as a RESTful API. It is used to get the login token and calculate the graph. The API is built on the Python Django library.

In this section, we will provide a technical overview of the back-end server. First, we will look at technologies that we used, the code structure and at last, we will point out the code parts that can be extended or improved in the future.

5.1.1 Technologies

Django

Django is a high-level open-source Python Web framework [14]. File structure might differ project by project, but generally speaking, there are some specific files that most of the Django projects contain.

The `manage.py` is a python script that allows developers to run a server with a simple command. The script sets environment variables from `settings.py` file and serves the endpoints on the given location.

The `urls.py` file defines URL endpoints that will be accessible from outside. For every endpoint, there must exist a method that receives a request and returns an HTTP response. These methods are located in the `views.py` file.

For the database purposes, the `admin.py`, `models.py` files and `migrations` folder are prepared. In our case, these files are practically empty.

NetworkX

NetworkX is a Python package with tools for graph processing, analyzing and visualizing. It allows us to manipulate and modify the graph effortlessly. It works well with other common libraries, e.g. `numpy`. We used it for most of the algorithms described in Chapter 2.

Namely, the following functions are included in our code:

- `nx.spring_layout` - Fruchterman-Reingold force-directed algorithm
- `nx.minimum_spanning_tree` - Kruskal's algorithm
- graph manipulation functions such as: `nx.connected_components` (returns connected components of the graph), `nx.subgraph` or `nx.isolates` (enlists nodes with no neighbors).

5.1.2 Back-end structure

The back-end folder structure is depicted on the figure 5.1. We will briefly summarize what is the purpose of individual files and folders.

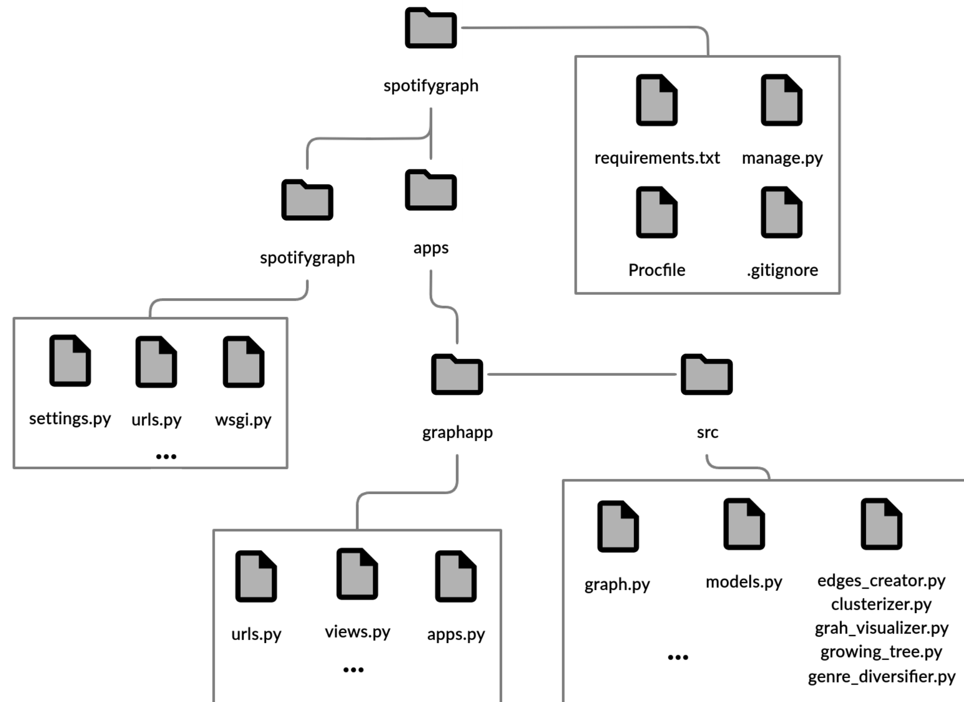


Figure 5.1: Back-end folder structure

Spotifygraph base folder

- `requirements.txt` - list of packages (with versions) on which our project depends
- `manage.py` - Python script responsible for running the server
- `Procfile` - the file necessary for Heroku deployment (see in the Deployment section 5.3). It contains only one line, that informs Heroku, where to find the `wsgi.py` file and that Gunicorn WSGI will be used (see more details in the wsgi.py file below).
- `.gitignore` - list of files and folders that are not included in the git repository. Includes folders as `__pycache__` etc.

Spotifygraph/spotifygraph folder

- `settings.py` - contains server settings, such as middleware list, CORS origin whitelist and Heroku settings.
- `urls.py` - list of URL patterns. In our case, this file just says that all useful URL endpoints are enlisted in the `urls.py` file in the `graphapp` folder.
- `wsgi.py` - WSGI (Web Server Gateway Interface) is a convention on how server and application written in Python programming language should communicate. During the application development, it is enough to run the server on the localhost. In the production, we must use some WSGI to serve the application. In our case, we used Gunicorn, that is convenient for the Heroku environment.

Graphapp folder

- `urls.py`
  The server serves at two enpoints: `graphapp/logged_in` and `graphapp/calculate_graph`. Both enpoints have their implementation in the `views.py` file.

- `views.py`
  For `logged_in` endpoint, the equally named method was implemented. It is called when the user is logged into Spotify. The method receives a code as a parameter, encodes a client ID and client secret and sends a request to Spotify API. After that, Spotify API redirects the user back to the frontend with access and refresh tokens.
  The graph edges, clusters and positions are all wrapped in the `calculate_graph` method. As a parameter of the HTTP request, the list of artists is sent. With data contained in this list, edges are calculated and then the `process` method of the `Graph` class is called. After everything is calculated, it is returned in a JSON format as an HTTP response.

- `apps.py` - gives information about apps in the current Django project. In our Django project, there is only one app - `graphapp`.

Src folder

This folder carries files with methods and classes that provide the classes responsible for graph calculations.

- `app_client.py` wrapper about static method that return Spotify application client ID and client secret. These values are read from the environment variables.

- `user_connection` - the `UserConnection` class is used for sending requests to Spotify API. For now, only a method for obtaining the access token is implemented.

- `models.py` is the file with all data classes, e.g. the classes `Artist`, `Song`, `Cluster`, `Position`, etc.

- `edges_creator.py` - `EdgesCreator` class is a wrapper around `get_edges` method, that creates edges from the list of artists. It implements the algorithm 6.

- `graph.py` contains the `Graph` class that is used as a wrapper for all necessary methods for working with graphs. It uses all the following classes to achieve all the goals given in the beginning. The parts are designed to be easily removed or changed so different algorithms could be used.

- `clusterizer.py` contains `Clusterizer` class used for graph clustering. This class contains contains parameters for minimal and maximal number

of artists in clusters. We were considering two clustering methods:

  - `SpectralClustering` method from the `sklearn.cluster` package
  - `community.best_partition` method from the `python-louvain` package

We had to decide, which method to use. The `best_parttion` method does not allow us to specify the number of clusters. Moreover, this method returned small clusters more often than desired. We had more control over the `SpectralClustering` method, therefore we decided for the spectral clustering. We were also thinking about mixing these two methods, but this approach was more complicated and better results were not guaranteed.

- `graph_visualizer.py` is the file of the `GraphVisualizer` class, which is responsible for graph layout computation. The `visualize` method is used for computing both main graph layout and inside cluster graphs.
  We were experimenting with two methods:
  - `nx.spring_layout`
  - `nx.drawing.nx_agraph.graphviz_layout`

  The second method gave slightly better results, especially for denser graphs. On the other hand, it was significantly slower and Graphviz visualization software was needed to be installed. This was also a problem during the deployment. For these reasons, we decided to use the `nx.spring_layout` method.

- `growing_tree.py` - the `GrowingTree` class implements a growing tree algorithm described in the Graph-based algorithm chapter. For Delaunay triangulation, the `Delaunay` method of `scipy.spatial` package is used. Minimum spanning tree is calculated with `networkx` method `minimum_spanning_tree`. There are a few algorithms available: Kruskal's, Prim's and Boruvka's. We kept the default Kruskal's version.

- `genre_diversifier.py` - the `GenreDiversifier` class is used for choosing the genres of individual clusters that will be shown in the cluster description (see algorithm 8).

5.1.3 Further code extensions

The code was written to be easily extensible on multiple levels. There are basically three places, where the code could be extended.

1. **endpoints** - the developer could add a new endpoint. In this case, the `views.py` and `urls.py` files in the `graphapp` folder should be edited.
2. **graph algorithms change** - if one of the graph algorithms should be changed, it is enough to change the code belonging to the file, that deals with a certain part of graph calculations. For example, if we wanted to change the clustering part, we will edit the `clusterizer.py` file.
3. **database** - if we want to add database in the future, we will have to create models in the `graphapp/model.py` file and add code to many other files. The exact procedure is available on the Django tutorial pages.

5.2 Front-end

We will look at the front-end code overview in this section. First, we will introduce the technologies we used. Second, we will analyze the graph visualization in web applications. In the end, we will go through the folder structure and possible code extensions.

5.2.1 Technologies

Web applications are usually written in HTML, JavaScript (JS) and CSS programming languages. However, it is very convenient to use helping tools to make the code more readable and extendable. We used the following tools to develop our application and manage the code:

Node.js, npm

Node.js is an asynchronous event-driven JavaScript runtime [15]. We used Node.js together with npm.

Node package manager (npm) is a JavaScript package manager [16]. Every package we used was installed via npm install command. The packages are downloaded into node_modules folder. The size of this folder is quite large and is never pushed to the version control repositories. Instead, all necessary information is written into the package.json file, that keeps the current version of all used packages.

Starting and deploying the server with npm is really simple. To start a server, we run the following command: npm start. To deploy the server on a preset homepage location, we run: npm run deploy.

TypeScript

TypeScript is a typed superset of JavaScript that compiles to plain JavaScript [17]. Among other functionalities, TypeScript makes it possible to create our own types (interfaces), removes the burden of type check of parameters and makes a developer's work easier.

React.js

React.js is a JavaScript library used for building UI components [18]. It uses JS XML (JSX) format that is being translated into JavaScript.

Example of JXS code: const element = <h1>Hello World!</h1>

The compiler responsible for JSX translation is called Babel.

React applications consist of building blocks called components. It is a good habit to have every component in its own file.

Every component can accept props (stands for properties). When these properties are changed from outside of the component, the component is notified and re-renders itself. There are some exceptions of this behaviour, but a standard flow is to update the component on props change.

A second important principle is the state of a component. The state object keeps the inner state of the component. If changed, the component is rerendered. The state should be changed via `setState` method.

The power of React lies in its work with virtual DOM. Virtual DOM is a programming concept, where an ideal representation of UI (virtual DOM) is separated from the real DOM. These two states are synchronized by some library, in case of React, it is ReactDOM [19].

Redux

Redux is a state container for JavaScript applications. It is extremely useful in React application because it provides a global state called store. Instead of passing the variables and callbacks deep down the DOM tree via props, we can simply connect the store to the individual components. The store is changed via dispatch actions that can be also connected to the components.

Sometimes it is necessary to have asynchronous actions. For this purpose, we will need to use Redux Thunk middleware. This middleware lies between an action being dispatched and the action reaching the reducers [20].

For debugging purposes, it is good to use a logger. There exist loggers that print nicely organized and coloured logs messages to the console. We used `redux-logger`.

5.2.2 Graph visualization

A graph is a very common visualization technique and a lot of JavaScript libraries were developed for this purpose. We needed to check the methods and approaches to graph visualization of available graph libraries to choose the best one for our goal. We considered libraries as `D3 graph`, `Cytoscape`, `vis.js`, `Sigma.js` and others. However, during the development, we run into a lot of problems that can be described in the following points:

Graph board

All the libraries have its way how to deal with space where the graph is located. This space has different names in different libraries, such as board, pan, box, etc. This space can be modified with dragging, scrolling or zooming. It is necessary to have a programmatic way to control this space, therefore most of the libraries have their API implemented to do the important moves. During the development, we found out that some functionalities are not really intuitive and difficult to control. For example, the work with graph coordinates was uncomfortable because the coordinates of nodes didn't correspond with the coordinates of the board. Sometimes, the coordinates got messy after zoom or board move.

Another requirement we demanded was the ability to set some limitations where the nodes can be moved. For this purpose, some libraries offer some form of border-box. Regrettably, they were not working as expected for its poor cooperation with HTML elements. For example, we tried to work with `Cytoscape.js` library. After every drag of nodes, we had to check, if the node got out of the

border-box with an additional condition. We expect this functionality to be included in the library we would use.

Node style

The most crucial requirement we had for graph visualizing was the ability to control how the nodes will look like. There are two ways of modelling the nodes: with SVG or HTML element. SVG might be flexible; however, for more complex nodes and layouts, it becomes unbearable to deal with. All the work that can be easily done with HTML `Flexbox` CSS styling (i.e. centring or using multiple rows and columns) is done manually using coordinates. More complicated structures are almost unmanageable in SVG and working with coordinates becomes very uncomfortable.

Thus, we wanted to work with HTML elements in the Main graph visualization. In the Inside cluster graph, we had smaller requirements for the node style. Nevertheless, it turned out in the end that the custom HTML element was still the best option.

In conclusion, we needed to have the possibility to use an HTML as a node and we didn't find this option in any library we tried.

Graph layout

Even though we couldn't have used the graph visualization libraries at its best, we wanted to use at least some graph layout algorithm, that would only calculate the positions of nodes and wouldn't show anything.

We considered the following libraries and found the following problems on each of them:

- `dagre.js` [21] - Library used for directed acyclic graphs, therefore does not fit for our purpose.
- `graph dracula` [22] - The layout seems nice enough, but it does not deal well with overlapping nodes.
- `react-force-graph` [23] - Very nice library with well done force algorithms; there's even the possibility to show the layout in 3D visualization. Unfortunately, the nodes can't be custom HTML elements.
- `D3 graph` [24] - one of the most used data visualization libraries. D3 has also implemented the graph visualization, e.g. force layout. Unfortunately, the layout computation and visualization are strongly connected. The graph layout changes dynamically, while it is being computed. We could try and separate the positions from D3 representations, but this approach would not be really clean.

Solution

All these problems and limits of the libraries led us into one final solution: we didn't use any JavaScript library directly meant for graph visualization. Instead, we used already computed positions of nodes (in the Main graph, as well in the Inside cluster graphs). For this purpose, we chose two React packages:

- `react-draggable` - `Draggable` div can be dragged around. Moreover, there are some useful functionalities as limiting where the `Draggable` can be dragged or choosing only a part of the div by which the element can be grabbed.
- `react-zoom-pan-pinch` - adds a zooming functionality. On mouse scroll (or touch-pad scroll) zooms the chosen div. Parameters as maximum zoom, speed of zooming and others can be set. When the div is zoomed, the user can move around with "dragging the background", the dragging functionality of nodes is preserved.

Advantages and limitations of the solution

As mentioned in the previous points, the biggest advantages of our approach are the following points:

- freedom of creating the node and edge design
- graph board controls - we are in control of zooming, board dragging and nodes movement
- layout - we can choose a layout that is suitable for us

There are also some limitations and disadvantages.

- The layouts of all inside clusters are computed even if they might not be needed. In other words, only the main graph is shown automatically and the remaining graphs are shown only if the user clicks on some cluster.
- The results must be sent from the back-end, this includes the time of the request. For users with a big amount of favourite artists or liked songs, this process can take a few seconds. (However, the time spent on the computation of graphs is proportionally much less than the time spent on obtaining the data from Spotify.)

5.2.3 Front-end structure

The folder structure of the front-end directory is shown in Figure 5.2. Three dots mean that there are other files in a folder. We will go through the important files and folders and describe what is their purpose.

Frontend folder

There are three important files in the base folder of frontend directory:

- `package.json` - information about all packages with versions on which our source code depends
- `tsconfig.json` - TypeScript compiler options
- `.env` - list of environment variables. Contains development and production variables accessible from JavaScript with: `process.env.VARIABLE_NAME`

Public folder

- `index.html` - a template with basic information like page title, logo and GitHub pages Single Page Apps workaround (see the Deployment section 5.3)

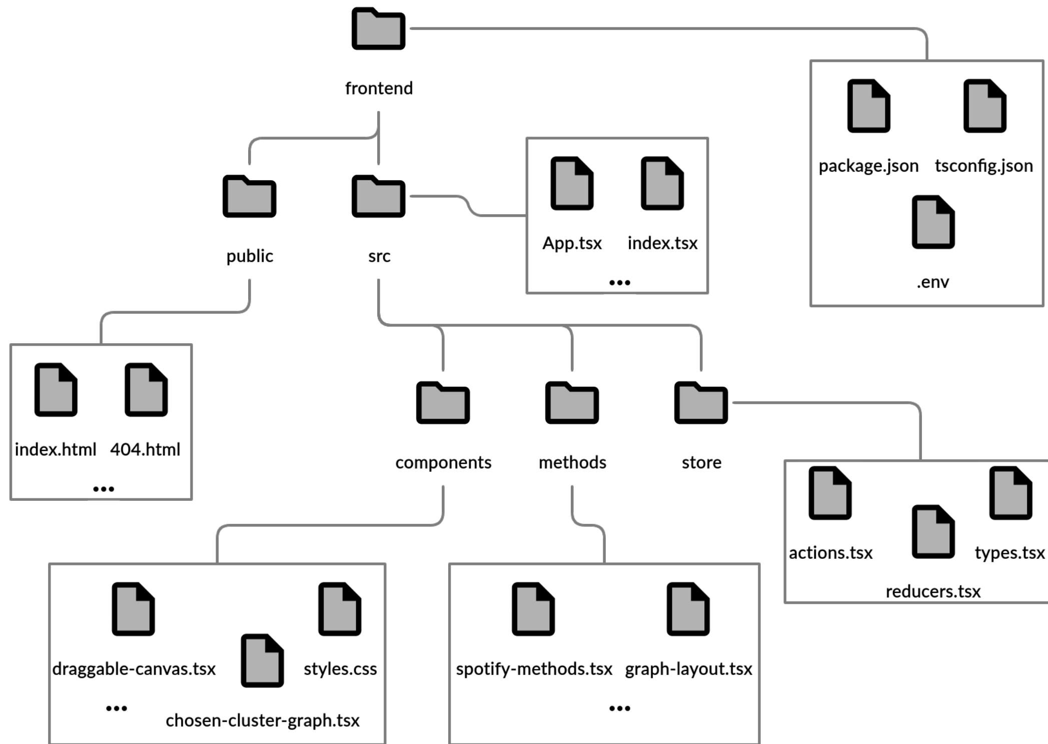


Figure 5.2: Front-end folder structure

- `404.html` - file that is necessary for GitHub Pages routing.

Src folder

- `App.tsx` - the main component that is responsible for connecting the Redux store to the whole application and route switching the correct URLs.
- `index.tsx` - has only one important line, that tells the React renderer, that it should render the App component.

Components folder

This folder involves code of the components from which the page is built.

- the files `welcome-page.tsx`, `login-page.tsx` and `my-graph-page.tsx` are wrappers of the pages, already mentioned in the previous chapter.
- `draggable-canvas.tsx` is the component where the cluster graph is visualized. It takes care of creating the cluster nodes and cluster edges.
- `cluster-node.tsx` and `cluster-edge.tsx` are components for visualizing the Main graph nodes and edges.
- `artist-node.tsx` and `artist-edge.tsx` are components for visualizing the Inside cluster graph nodes and edges.
- `chosen-cluster.tsx` is the component that is shown when certain cluster is clicked.
- `side-bar.tsx` is the component of the left side-bar and `chosen-artist-bar.tsx` is the right side-bar with artist's songs that appears when a certain artist is clicked.

- `player.tsx` is the component on the bottom that allows user to control the playback.
- `styles.css` and other style files contain CSS classes

Methods folder

Methods folder contains files with methods that are used by components, but whose code is not directly depending on HTML rendering.

- `graph-layout.tsx` - contains functions that receive the precomputed layouts obtained from back-end and return the final positions of clusters and artists as it will be shown on the page.
- `graph-parsing.tsx` - is responsible for parsing the JSON objects that were received from the back-end.
- `spotify-methods.tsx` - all methods that are responsible for requesting the Spotify API.

Store folder

Store folder contains files with methods and interfaces responsible for correct functionality of Redux store.

- `types.tsx` - a file with Typescript interfaces that are used in components.
- `actions.tsx` - contains actions that modify store.
- `reducers.tsx` - initializes store instance and applies the middleware, e.g. logger and Thunk.

5.2.4 Further code extensions

There are several aspects of front-end that could be changed in the future:

- Spotify API requests - if we want to add some methods that would retrieve data from Spotify, we should put it in the `spotify-methods.tsx` file and export it.
- store - for adding new variables to the global store, we must edit the `types.tsx` file and change the State interface. If we want to create a state manipulating method, we should add it into the `actions.tsx` file.
- further functionality of components - components can be easily extendable. We can add more variables from the global state and provide more information to the user.
- methods - if we want to add methods that are not directly depending on the DOM, we should add them to the `methods` folder.

5.3 Deployment

We needed to deploy the python server and front-end part. We decided to deploy them separately, as the back-end server works only as a REST service.

Frontend

During the app development, we used the GitLab platform as an online tool for version control. GitLab has a possibility to deploy HTML pages available for all developers using GitLab. It would have been the simplest possibility because everything would be on the same place and committing changes would immediately launch deploy. Unfortunately, the work with GitLab pages turned out problematic and not as fluent as needed, so we decided to use GitHub pages.

As GitLab, GitHub allowed us to deploy our application on the address `gajdusep.github.io/spotifygraph`.

During the process, we had to deal with URL routing, that is not very intuitive on GitHub pages, because the URL paths are not recognized by GitHub pages. We used a workaround written by Rafael Pedicini that deals with this problem and makes routing work. [25]

Python server

We had to decide which server provider to chose. As a first option, we were thinking about Amazon's AWS with the help of a Docker container system. This turned out to be more complicated than needed, therefore we chose Heroku.

Heroku is a container-based cloud Platform as a Service.[26] It offers developers a simple way to deploy their apps. For a Python project, it is enough to add a `Procfile`, `requirements.txt` file and a few lines into several files and push the repository to the Heroku servers with the following command:

```
git push Heroku master
```

After that, the server opens endpoints on the given location.

5.3.1 Custom deployment

If you want to run the application on your machine, you need to run both servers separately.

Front-end

To start the front-end server, you need to have npm installed[1] on your computer. Run the following commands to start the server:

`npm install` - install dependencies from the `package.json` file.

`npm start` - runs the frontend server. The server will be served at localhost:3000 by default.

Back-end

For running the back-end, you need to have Python3 installed on your PC. We used the pip package manager. Install all requirements from the requirements.txt file. Furthermore, you have to set environment variables. On Linux systems the best option is to edit the `.bashrc` file. The first two variables should be kept secret:

[1]follow the tutorial on https://www.npmjs.com/get-npm

- `SPOTIFY_CLIENT_ID` - client ID from Spotify APP
- `SPOTIFY_CLIENT_SECRET` - client secret from the Spotify APP
- `SPOTIFY_GRAPH_AUTHORIZE_URI` - redirect URI - on the local development, change to http://localhost:3000/spotifygraph/mygraph

To run the server, navigate to the base spotifygraph folder and type:

```
pip3 install -r requirements.txt
python3 manage.py runserver
```

The server should be now running at localhost:8000.

6. Case studies

In this chapter, we would like to show some examples of graphs that were sent by users. We will see if the results are satisfying and well shown and discuss the parts that went well and the things that could be improved in the future.

Let us have several users with their behaviour on Spotify (the number of artists is just approximate). All of the following data were sent by real users, for their privacy, we changed their names.

- Alice - up to 50 artists - she does not use Spotify very often. She has a few liked songs and listens mostly to the same set of genres. (See figure 6.1.)
  We can see that when a user does not have many artists, the TOTAL MIX cluster will be relatively large because there are fewer edges and less inner structures.
- Bob, Bart - up to 150 artists - they use Spotify more often than Alice and have more favourite genres, but still prefer to listen to the artists and songs that they already know. (See figures 6.2, 6.3.)
  In Bob's graph, we can see that some images of artists are blank (light blue). This happens either when no image is available for an artist or when the browser loads some images more slowly than others.
  In the top right corner of Bob's graph, there lies a cluster with the Swedish flag that contains only two artists. It might happen that only one of them is from Sweden because of the 40% limit to show the flag (Subsection 4.5.2). Therefore the information about the country is not fully precise. However, we decided to keep it this way to show rather more information than less.
- Cecilia, Caroline - up to 400 artists - they use Spotify quite often and like to explore new artists. They have a couple of favourite genres that they change according to their mood. (See figures 6.4, 6.5.)
  Cecilia's and also Caroline's graph are densely connected because their clusters are relatively similar to each other. The edges between clusters in Caroline's graph are thicker than in Cecilia's graph. The thickness and number of edges can be slightly disturbing. However, it shows clearly that Caroline's favourite artists are more similar to each other than Cecilia's artists.
- David and Eve - up to 1000 artists - both are heavy Spotify users. They like a lot of songs and they're not afraid of anything new. However, while David likes a lot of electronic dance songs and keeps exploring them, Eve's musical taste is spread more. She likes everything, from J. S. Bach to One Direction. (See figures 6.6, 6.7.)
  The clustering and also visualizing techniques turned out very well in Eve's graph because it visibly separated modern pop and dance music from classical music and classical performance. Generally speaking, the more diversified musical taste a user has, the more visible the separation will be and the nicer the structures will be.

On the screenshots, we can see that the visualization is really nice and clear up to 400 artists. Above this number, it becomes quite difficult to keep the number of clusters low and point out the hidden structure in the data.

Figure 6.1: Alice (50 artists)

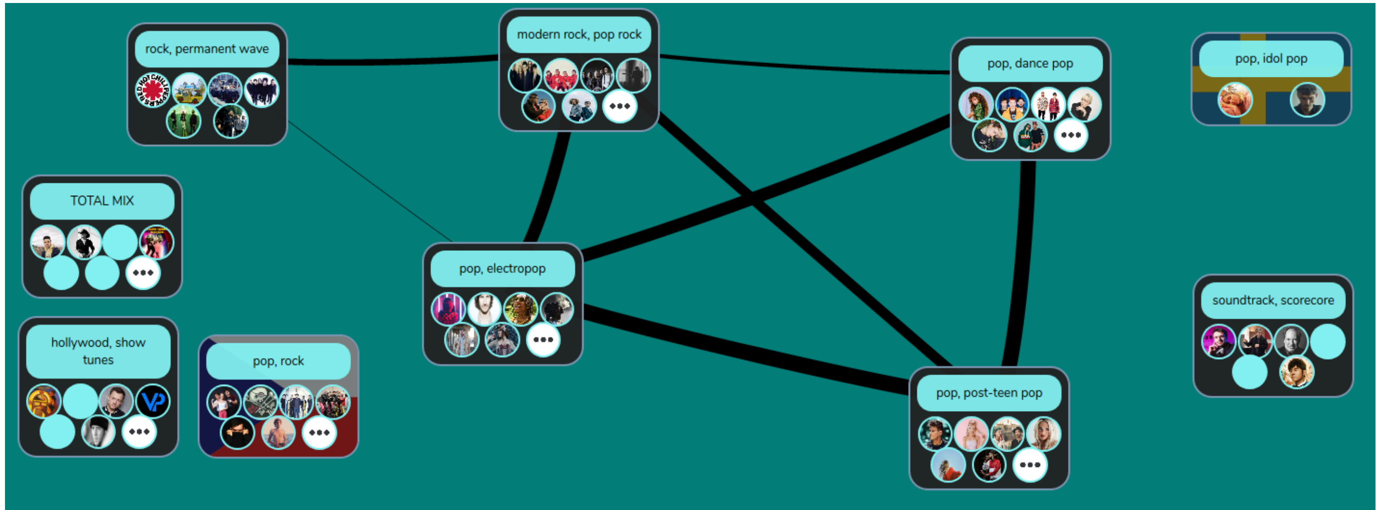


Figure 6.2: Bob (125 artists)

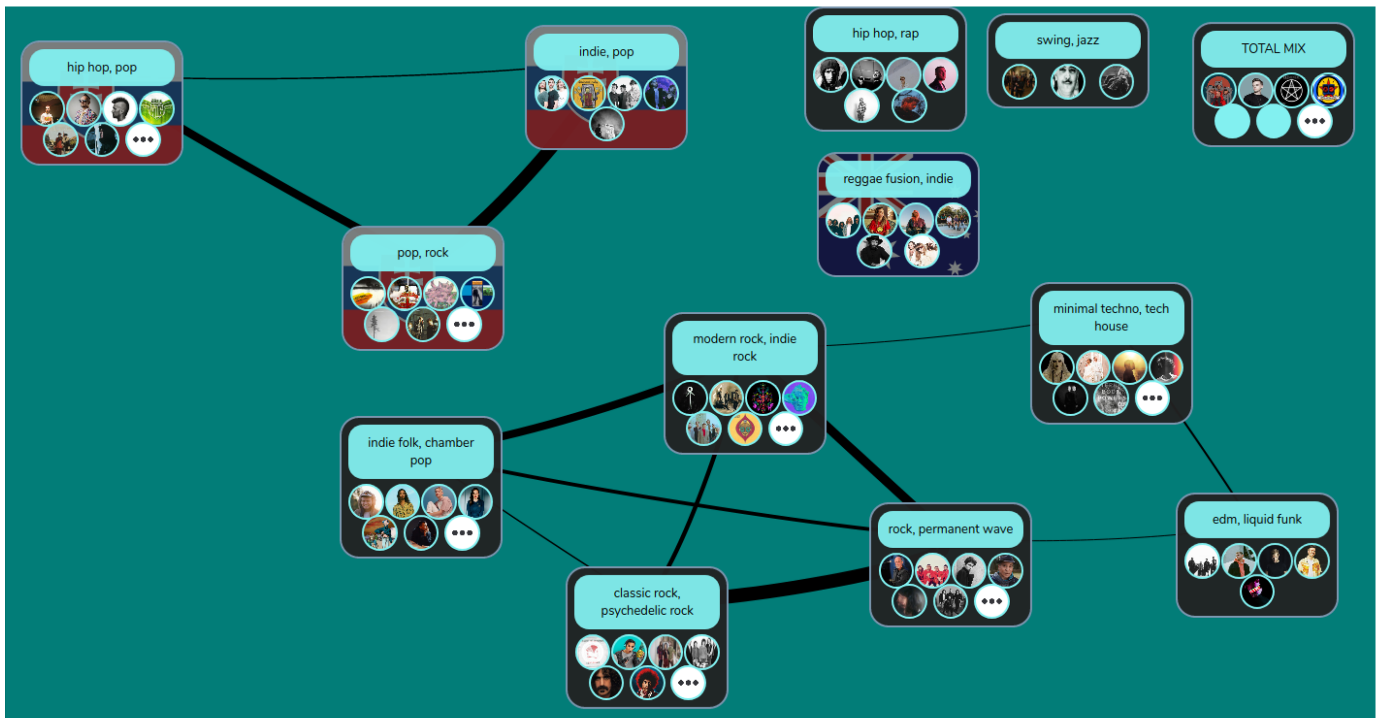


Figure 6.3: Bart (153 artists)

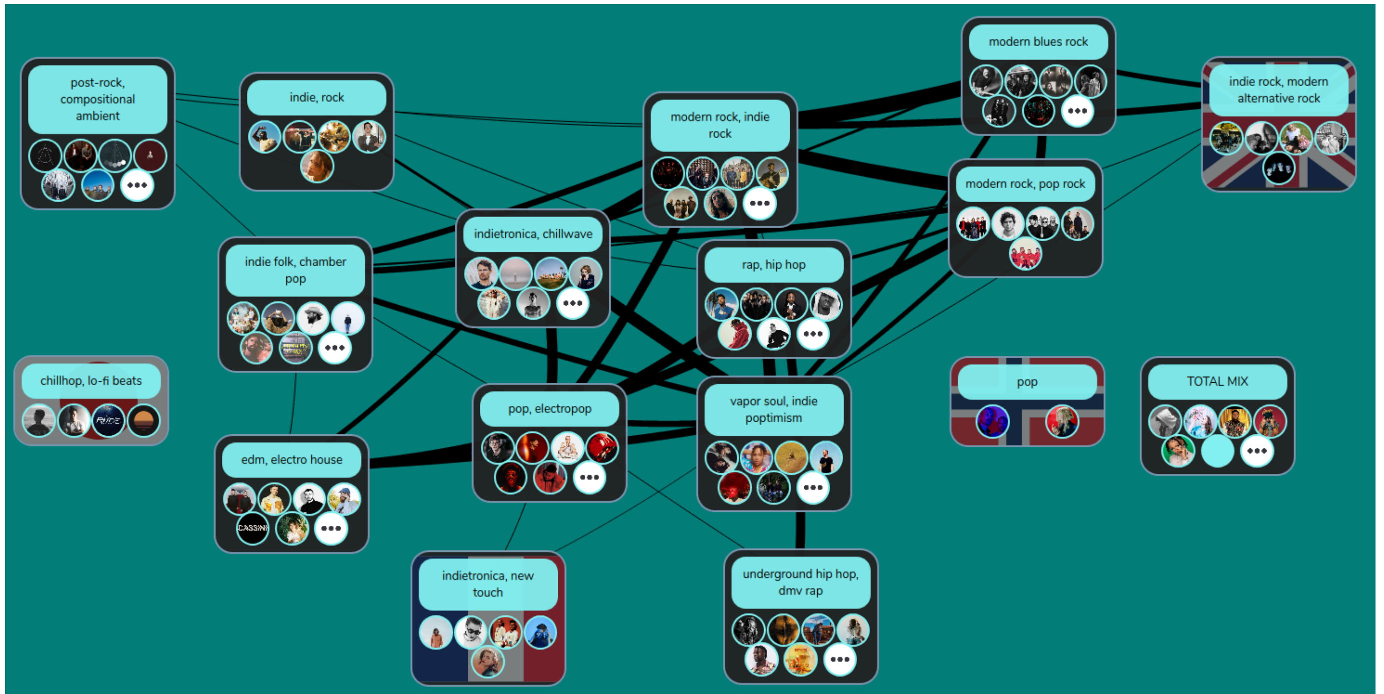


Figure 6.4: Cecilia (208 artists)

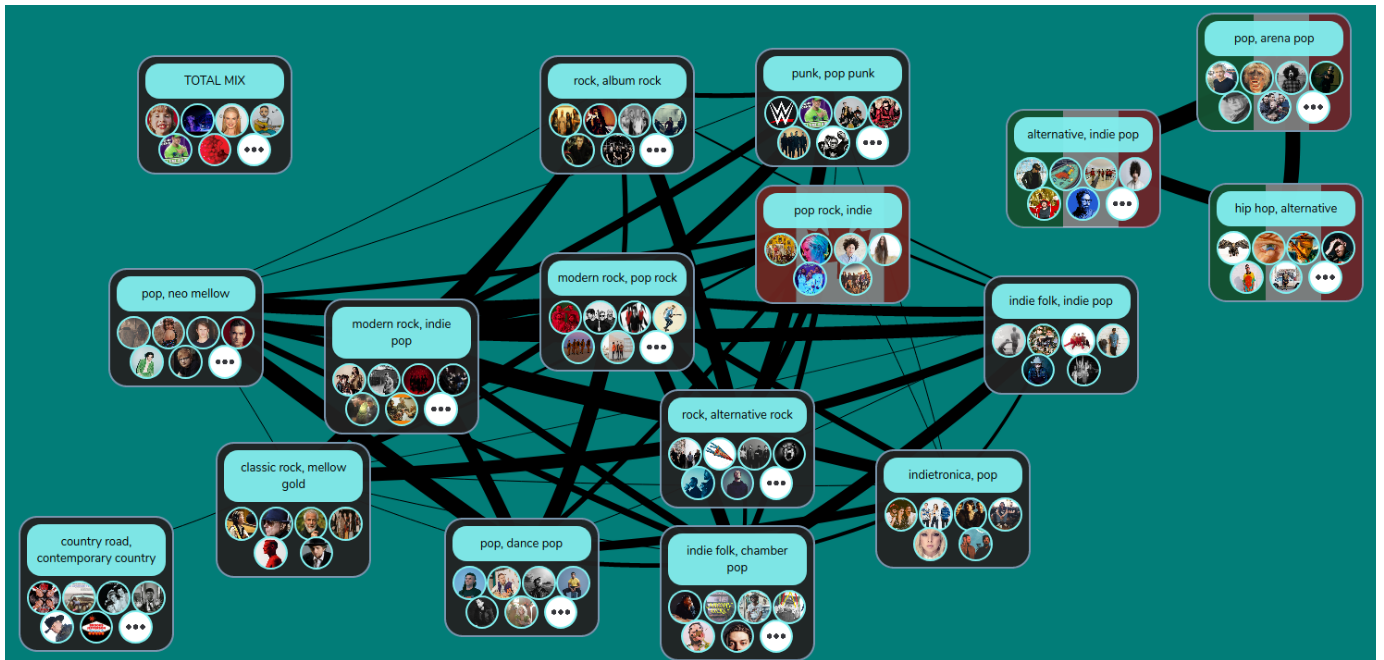


Figure 6.5: Caroline (301 artists)

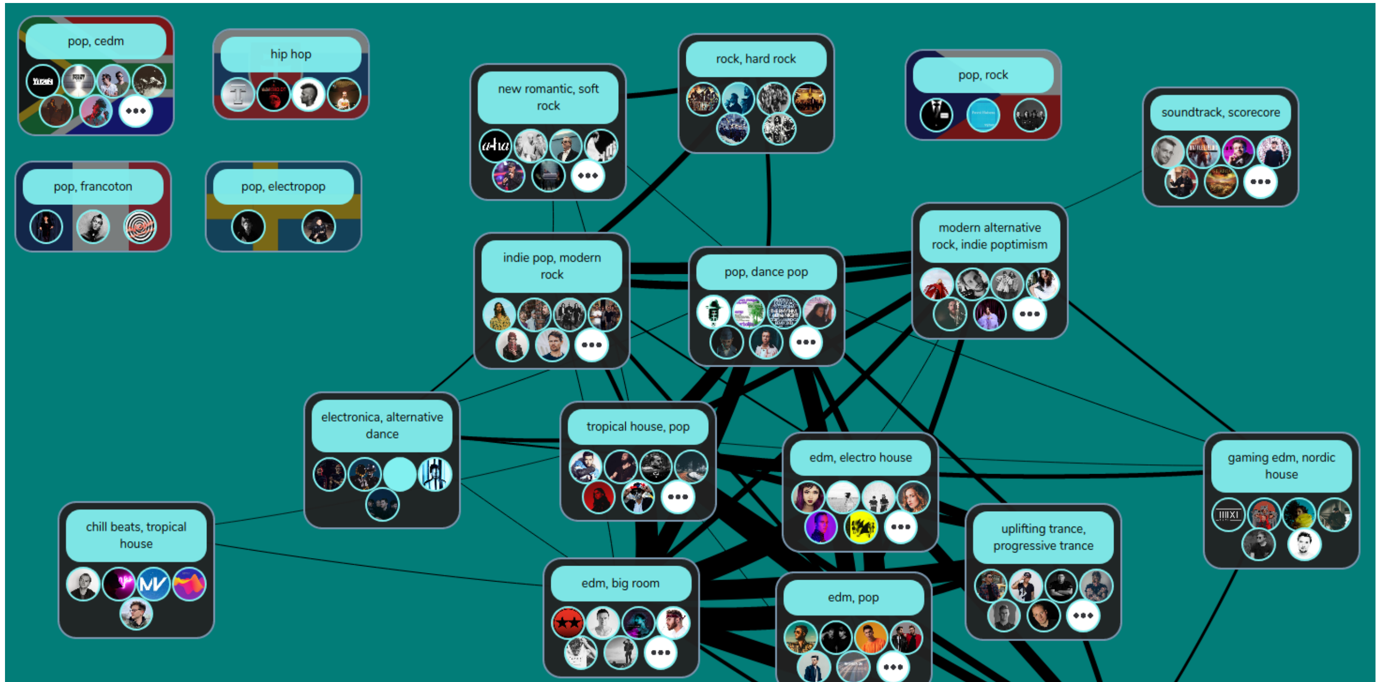


Figure 6.6: David (446 artists)

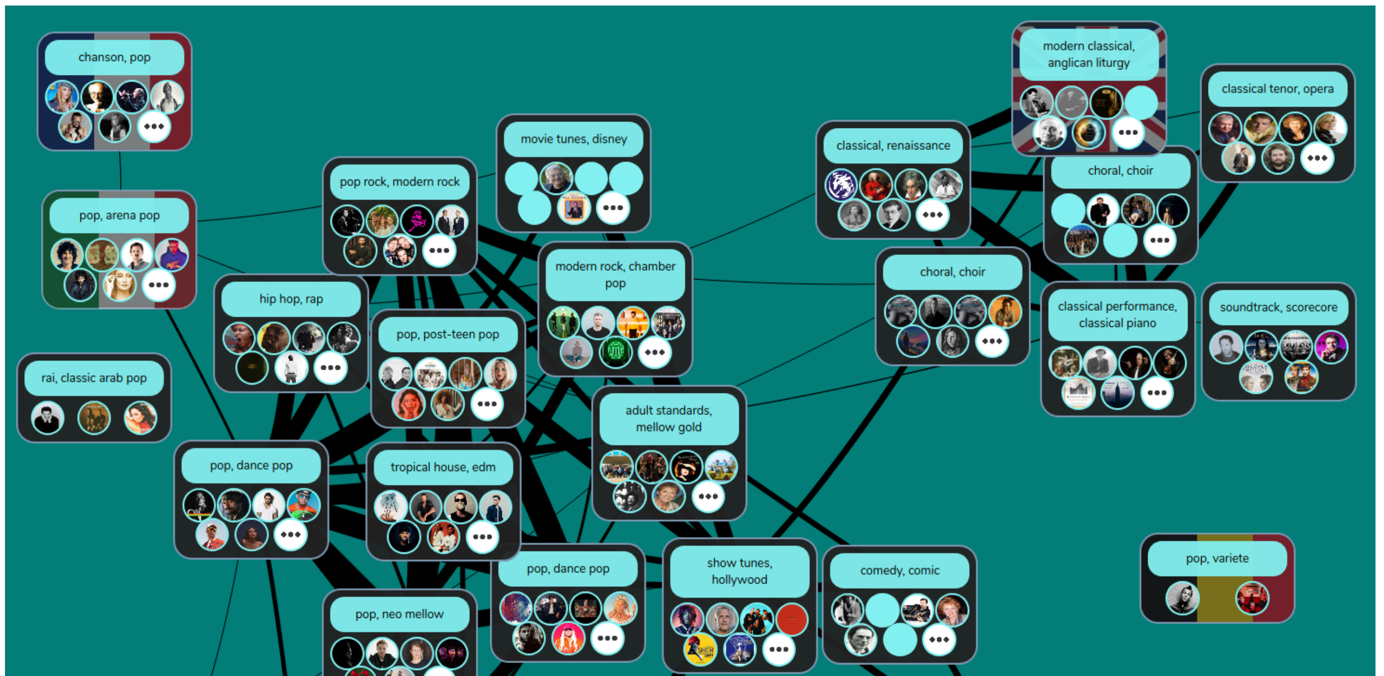
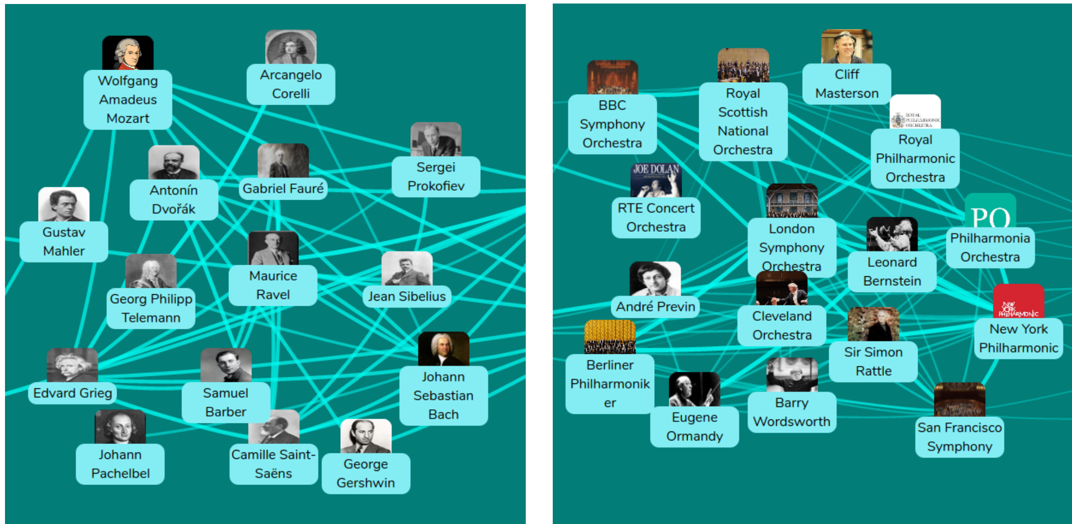
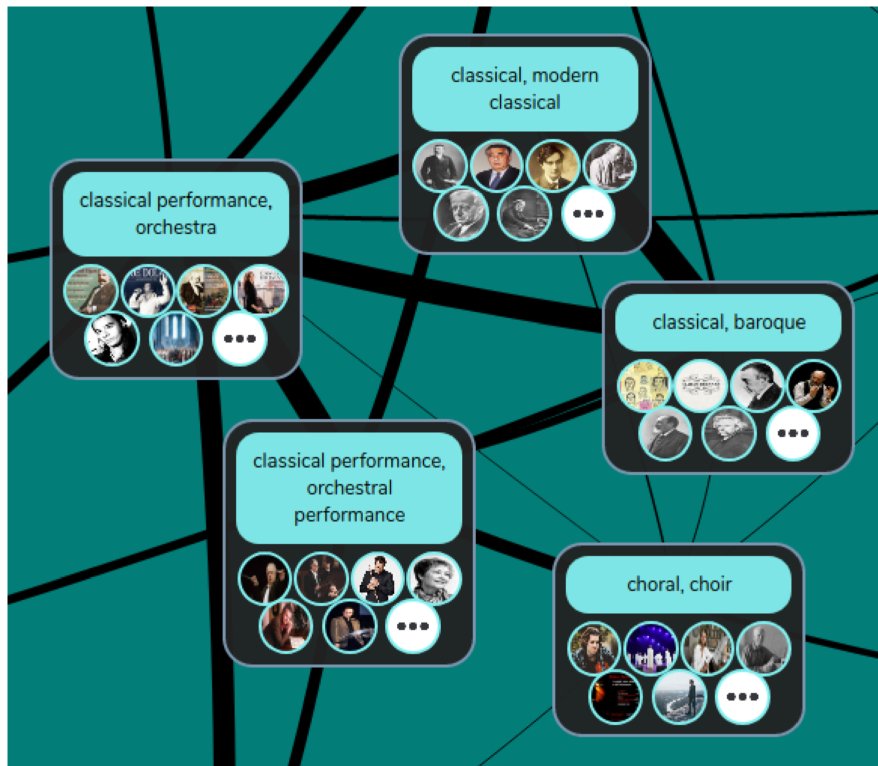


Figure 6.7: Eve (937 artists)

Let us mention one interesting detail that we discovered while going through the users' graphs. When clustering classical music, clusters clearly separate composers from performers. The example is shown in Figure 6.8.



(a) (b)



(c)

Figure 6.8: (a) Inside cluster graph with composers. (b) Inside cluster graph with performers. (c) Main graph cutout.

Conclusion

The goal of this thesis was to find hidden structures of Spotify users' data and visualize them in a new way. We achieved these goals with the help of graph clustering and graph visualization techniques.

We approached the visualization as a two-level graph visualization. The top-level layer showed the division of artists into groups and similarities between those groups. In the lower level, the artists and relations between them were visualized.

The application revealed users' data and showed them their preferences on Spotify. Users were able to interact with Spotify via our application, i.e. they played songs and saved them to their liked songs playlist. If users liked their graphs, they were able to download them and send them to their friends.

We demonstrated that we were able to make a functional system that can work with external sources as Spotify API. The code is extendable and ready to be improved in the future with numerous features.

Future work

During the development of our application, we thought about a lot of features that were not included or programmed in our final version. We would like to work on the following points which might significantly improve user experience.

Better related artists caching

Even though the current implementation includes related artists caching, it is still the slowest part of our application and some users might find is very annoying. That's why in the future development, implementing the database with the cached related artist might be convenient.

Improved player

Sometimes, it is inconvenient when the user starts to listen to the chosen song, but after 30 seconds it suddenly stops playing. This behaviour is quite annoying and it would be worthy to allow full song player to the premium users. This would include some form of song queue; from a programming point of view, we would have to use Web Playback SDK that allows playing the whole songs.

New artists recommendation

In the current version of our application, users can explore only their already liked artists. They can play songs that they do not know yet, but in the future, we could also add a possibility of exploring new artists.

New artists would be shown in the Inside cluster graph. The newly shown artists would be the most similar to as many artists in the cluster as possible. The similarity would be computed using the related artists information and similar genres.

Further work with data browsing

There are still a lot of features remaining that would help the users orient in their graphs. The following list contains the possibilities we had in mind:

- **Search artist in the graph** - after filling the search box and clicking some of the provided search results, the cluster with the artist would open and the artist data would be shown.
- **Filter artists** based on genres or popularity or other criteria we would choose.

Personalization

In the future, we would like to allow more personalization for the users. We could for example explicitly ask users several questions via HTML form regarding their preferred settings::

- How should the importance of artists be computed? What are the user's habits - is it more common to like songs, follow artists or is the artist's popularity also important?
- Should the songs be also taken from other saved playlist? If so, which ones?

Publicly accessible graphs

Sometimes it may happen that some users would like to share their graphs with their friends. For that purpose, it would be useful to have graphs accessible via URL, e.g.: `https://.../graphs/graphID`.

Other design improvements

Except previously mentioned points, there are some small improvements that could be made:

- Blank images - if there is some artist that doesn't have image, the application wouldn't show it as blank image in the cluster view. Instead, the artist wouldn't be shown until the cluster would be clicked.
- Graph file saving - in the current version, the graph is saved as it was obtained from the back-end. The saving procedure doesn't take into account the changes that a user did to the layout by grabbing and moving nodes. In the future, we could save the graph with the layout that was edited by a user.

Bibliography

[1] M. E. J. Newman. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences*, 103(23):8577–8582, 2006. ISSN 0027-8424. doi: 10.1073/pnas.0601602103. URL https://www.pnas.org/content/103/23/8577.

[2] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, Oct 2008. ISSN 1742-5468. doi: 10.1088/1742-5468/2008/10/p10008. URL http://dx.doi.org/10.1088/1742-5468/2008/10/P10008.

[3] Jitendra Malik Jianbo Shi. Normalized cuts and image segmentation. *Copyright 2000 IEEE. Reprinted from IEEE Transactions on Pattern Analysis and Machine Intelligence, Volume 22, Issue 8, August 2000, pages 888-905. Publisher URL: http://dx.doi.org/10.1109/34.868688*, 2000.

[4] Andrew Y. Ng, Michael I. Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*, NIPS'01, page 849–856, Cambridge, MA, USA, 2001. MIT Press.

[5] Edward M. Reingold Thomas M. J. Fruchterman. Graph drawing by force-directed placement. *Department of Computer Science, University of Illinois at Urbana-Champaign, 1304 W.Springfield Avenue, Urbana, IL 61801-2987, U.S.A*, 1991.

[6] Sergey Bereg Leishi Zhang andAlexander Holroyd Lev Nachmanson, Arlind Nocaj. Node overlap removal by growing a tree. *Hu Y., Nöllenburg M. (eds) Graph Drawing and Network Visualization. GD 2016. Lecture Notes in Computer Science, vol 9801. Springer, Cham*, 2016.

[7] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956. ISSN 00029939, 10886826. URL http://www.jstor.org/stable/2033241.

[8] Spotify. Spotify api documentation, . URL https://developer.spotify.com/documentation/web-api/.

[9] Spotify. Spotify api github issues, . URL https://github.com/spotify/web-api/issues.

[10] Spotify. Spotify authorization guide, . URL https://developer.spotify.com/documentation/general/guides/authorization-guide/.

[11] Spotify. Spotify api documentation, . URL https://developer.spotify.com/documentation/web-api/reference/artists/get-related-artists/.

[12] Spotify. Web playback sdk, . URL https://developer.spotify.com/documentation/web-playback-sdk/reference/.

[13] Spotify. Spotify info page, . URL https://newsroom.spotify.com/company-info/.

[14] Django. Documentation, tutorial. URL https://docs.djangoproject.com/en/3.0/intro/tutorial01/.

[15] Node.js. About node.js. URL https://nodejs.org/en/about/.

[16] Inc npm. npm. URL https://www.npmjs.com/.

[17] Microsoft. Typescript. URL https://www.typescriptlang.org/.

[18] w3schools. What is react? URL https://www.w3schools.com/whatis/whatis_react.asp.

[19] Inc. Facebook. React virtual dom. URL https://reactjs.org/docs/faq-internals.html.

[20] Alligator.io. Redux thunk. URL https://www.digitalocean.com/community/tutorials/redux-redux-thunk.

[21] dagrejs. Dagrejs. URL https://github.com/dagrejs/dagre/wiki.

[22] Johann Philipp Strathausen. Graph dracula. URL https://www.graphdracula.net/documentation/.

[23] Vasco Asturiano. react-force-graph. URL https://www.npmjs.com/package/react-force-graph.

[24] D3. D3 graph. URL https://d3js.org/.

[25] Rafael Pedicini. Single page apps for github pages. URL https://github.com/rafrex/spa-github-pages.

[26] Heroku. About heroku. URL https://www.heroku.com/about.

List of Figures

1.1 Main graph - clusters are nodes 7
1.2 Inside cluster graph - Cluster1 was chosen and its artists are shown nodes 7

2.1 {T1,T2} is Delaunay triangulation of {A,B,C,D}, {T3,T4} is not (points B and D lie in circumcircles of T3 and T4) 13
2.2 GTree cost function, source: [6] 13

3.1 Authorization Code Flow 16
3.2 Implicit Grant Flow 16
3.3 Client Credentials Flow 17

4.1 Application flow 20
4.2 Fitting small components into the layout 25
4.3 Two types of graphs: (a) Main graph (b) Inside cluster graph . . . 27
4.4 (a) A cluster with more than 7 artists, mostly from Italy (b) Cluster after zoom with more genres (c) A cluster with less than 7 artists, without explicitly mentioned country of origin 29
4.5 The inside cluster graph with indie artists. Right image shows the usage of slider that hides less important edges and artists. 30
4.6 Song state examples of Blanco White's songs 31
4.7 Panel with similar clusters 31

5.1 Back-end folder structure 34
5.2 Front-end folder structure 41

6.1 Alice (50 artists) 46
6.2 Bob (125 artists) 46
6.3 Bart (153 artists) 46
6.4 Cecilia (208 artists) 47
6.5 Caroline (301 artists) 47
6.6 David (446 artists) 48
6.7 Eve (937 artists) 48
6.8 (a) Inside cluster graph with composers. (b) Inside cluster graph with performers. (c) Main graph cutout. 49

List of Abbreviations

- MST: minimum spanning tree
- API: application programming interface

A. Attachments

A.1 Electronic attachments

To see the application, visit the following link:

- https://gajdusep.github.io/spotifygraph/

Sometimes it happens that Heroku server slows down rapidly after a certain period of time of not being used. Then, it might happen that the application shows you an error message. In this case, please contact us and we will restart the server.

The code is available in the public repository:

- https://gitlab.com/gajdusep/spotifygraph

The `zip` file with the code is also uploaded as the bachelor thesis attachment. After unpacking the file, you will find the `README.md` file that will contain further information, `frontend` and `spotifygraph` folders with the code and the `testfiles` folder with files ready to be uploaded as a test files on the page.