Juan Carlos Gallegos, V00816131, December 2018

CSC 490 Final Report, Fall 2018

Abstract

The Knowledge Representation component of our system will play a central role in our application's functionality. It will be responsible for storing, organizing, and exposing information regarding a variety of music entities for promptly fulfilling user requests. This report provides: detailed requirements for our Knowledge Representation system in the context of our project goals; a summary of the research conducted regarding Knowledge Representation approaches and technology; explanation and justification for several key design choices; implementation details that illustrate how the chosen technologies and architecture design solve our endeavour's core problems; a brief account of how I filled the database with data gathered from Spotify's Web API; and, finally, documentation for the resulting knowledge representation API. (The source code for the project can be found in this public GitHub repo.)

Introduction

Overview

The aim of our project is to create a system capable of answering questions about musical entities (songs, artists, genres, etc.) and relationships between them (e.g. similarity of tempo, mood, style, etc.). In particular, we strive to create a natural language interface that allows users to play and explore music; and so, the system must be able to execute database queries that find answers to a user's questions. Overall, we aim to support two core types of questions:

- *A*. Retrieving music metadata
    - e.g. Who is the song "Despacito" by?
- *B*. Finding music based on **arbitary relationships** to other music
    - e.g. Play a song similar to "Despacito".

Finding an answer to questions of type *A* would be natural with a relational database architecture: in the simplest case, we would query a `songs` table for the value in some column (e.g. artist's name). Indeed, a single title ("Despacito") may correspond to multiple songs, but the information would be easily retrievable after having the user provide disambiguation.

However, we would need to change our schema design in order to answer questions of type *B*. One possibility is to create binary relationships (e.g. `similar to`) that connect two rows in a single table (e.g. two rows in `songs`). In the discussion below, we'll see how we settled on a schema design that incorporates this idea.

Semantic Networks

Before designing a database capable of answering the types of questions that we aim to support, I researched Knowledge Representation (KR) systems to gather a high-level picture of the existing solutions. Semantic networks appeared repeatedly in the literature [5] as a way to represent meaning of entities with respect to each other, making them a natural candidate for our app's KR system.

As a data structure, a semantic network is a directed, labeled graph where edges represent relations between entities: for example, a semantic network may represent the meaning of a word by connecting it with another word with an edge labeled "synonym of". In the same way, we could represent similarity between songs as well as other entities using a semantic network.

Once we decided on semantic networks, I had to find an implementation approach that would allow us to easily query for entities based on their relations to other entities. In particular, we required an easy way to find songs similar to some specific song. I found two distinct implementation options: Resource Descriptor Format (RDF) storage, which we would query with SPARQL, an SQL-like query language; traditional relational storage, which we would query with SQL.

Implementation

RDF storage exists primarily for use by the Semantic Web [4]: an approach to structuring data available on the internet for ease of consumption by computers, rather than humans. At its heart, the Semantic Web is a semantic network stored in triples of *subject*, *verb*, and *object*, where the verb describes the relationship between the two entities, subject and object. By contrast, ConceptNet, originally built and launched at the MIT Media Lab [3], exemplifies the use of a relational schema to represent a semantic network. Instead of using triple storage, the ConceptNet schema uses a `nodes` table to store information about entities and an `edges` table to store information about how the entites are related. Thus, finding nodes connected to some arbitrary source node with a certain relation is a matter of joining the two tables and selecting for the source node and the desired edge type. Notably, finding any kind of relation is possible with essentially the same query, requiring only minor modifications to the `JOIN` condition and `WHERE` clause(s); the details are discussed in the **Tech** section below.

In general, the relational representation would suffice to store and find arbitrary relations between our entities (e.g. songs, artists, etc.). However, our app's purpose differs from that of ConceptNet in that our entities are *rich*; ConceptNet needs to store only words, but we store several different entities, each with a number of attribute fields. For songs alone, we might store name, artist, length, beats-per-minute, genre, etc. It should be noted that we should not aim to represent relative tempo as a relation between songs because that would lead to a heavily connected graph. Instead, we will store song tempos individually and use conventional SQL queries to answer user requests like "Find songs faster than 'Despacito'".

Requirements

As part of the design process, we determined the type of questions we aimed to answer. With these in hand, choosing a KR system architecture became a matter of supporting these requests. Ideally, our KR system would answer the same question differently as it accrues more data about entities and the relations between them.

Questions for the Relational Database

A significant portion of the types of questions that we aim to answer may be easily answered with SQL queries.

- Who is the **artist** of the song "Despacito"?
    - Requires basic knowledge about songs, artists, releases, etc.
    - Requires determining *which* song entitled "Despacito" is being referenced.
- What are some songs **faster** than "Despacito"?
    - Requires knowledge about relations between songs based on **single attributes**.
    - As mentioned earlier, since this question requires knowlede about the relation between all nodes, we do not want to represent it as a relation in our semantic network; instead, we can query the `songs` table and order the results on the `tempo` column.
- What are some songs that are **more obscure** than "Despacito"?
    - Again, this type of information should not be represented by the semantic network since it would exhaustively connect all nodes to all other nodes (with "less obscure than" or "more obscure than" relations). Instead, we can maintain a `play_count` column in the `songs` table from which we could get info about relative obscurity by ordering the results.

Questions for the Semantic Network

- What are some songs **similar** to "Despacito"?
    - Requires knowledge about similarity between songs based on **various attributes** (though the representation is ultimately an edge between song nodes in the semantic network).

Intersection of Relational Database and Semantic Network

Finally, there are questions that require combination of answers that we would acquire through distinct queries: one to semantic network tables and the other to the traditional relational tables.

- What are some songs **similar** to and **faster** than "Despacito"?

- What are some songs **by Justin Timberlake** that are **similar** to songs **by Justin Bieber**?

Stretch Goals

Indeed, many desirable features exist, but most are currently out of scope, given our short time window. For one, we would like to incorporate *human-in-the-loop* correction: allow our KR system to "learn" from user interaction. Not only would this allow our system to correct misinformation, but also to potentially personalize the app; we could attempt to reflect the nuances of a user's personal taste in the KR system. This would require a sophisticated ability to discern when relations in the database ought to be modified based on interactions with the user.

Implementation

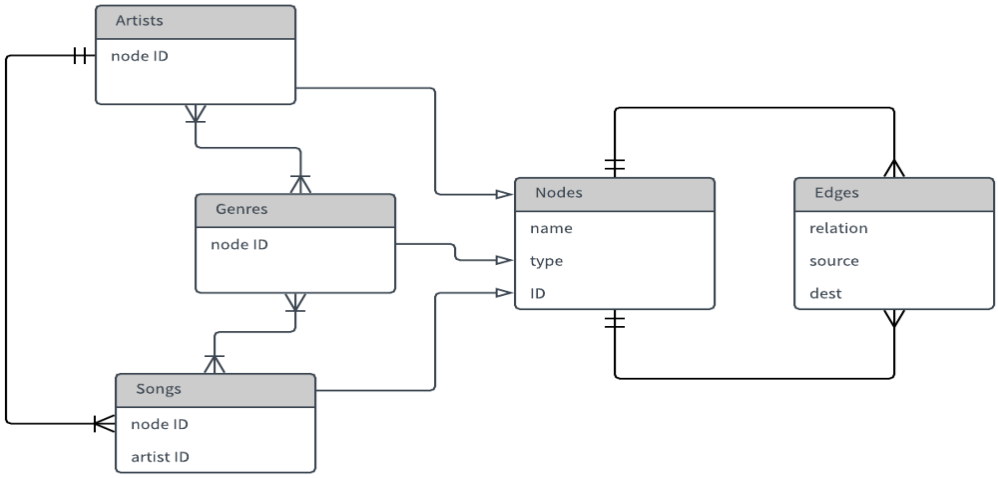
Schema

As discussed earlier, I opted for the simplicity of an RDBMS instead of an RDF database. As demonstrated by ConceptNet's schema, we can use a relational database model to represent the semantic network itself.

ConceptNet concerns the meaning of words and phrases, a homogenous domain. However, our domain-specific semantic network must deal with a heterogenous mix of entities: songs, artists, genres, and more. Moreover, they each contain various attribute fields. And so, we created a hybrid schema capable of both storing the data fields for each different entity and treating them all uniformly as an abstract node in the semantic network. To accomplish this, we created conventional tables for each entity and made each a **subclass of the `nodes` table**, representing an "is a" relationship between each entity and the parent `nodes` table.

Thus, each entity is represented by a `node` in the semantic network, regardless of type; we achieve this with *Class Table Inheritance*, a design pattern that appears in *Patterns of Enterprise Application Architecture* by Martin Fowler [1], [2].

The simplified ER diagram below illustrates the overall structure of our database without information about the relationships between entities.



The database schema is included below. The key things to notice are:

- the `nodes` and `edges` table, which abstractly represent entities and the relations between them; and
- the foreign key references from each of `artists`, `songs`. and `genres` to the `nodes` table, representing the "is a" relationship between them.

```
CREATE TABLE nodes(
    -- e.g. "Despacito", "Justin Bieber", "Pop", etc.
    name varchar(50) NOT NULL,

    -- e.g. "song", "artist", "genre", etc.
    type varchar(50) NOT NULL,

    -- NOTE: this field is an alias for SQLite's "row_id" column
    -- Setting this column to NULL at insert, will automatically populate it
    id INTEGER PRIMARY KEY
);

-- TODO: add constraints about node type (probably in a trigger function)
CREATE TABLE edges(
    source  int NOT NULL REFERENCES nodes(id),
    dest    int NOT NULL REFERENCES nodes(id),
    rel     varchar(40) NOT NULL,
    score   real NOT NULL CHECK (score >= 0 AND score <= 100),
    PRIMARY KEY (source, dest, rel)
);

CREATE TABLE artists(
    node_id                 int PRIMARY KEY REFERENCES nodes(id) NOT NULL,
    num_spotify_followers   int
);
```

```
CREATE TABLE songs(
    main_artist_id  int REFERENCES artists(node_id) NOT NULL,
    popularity      int CHECK ((popularity >= 0 AND popularity <= 100) OR
popularity = NULL),
    duration_ms     int,
    node_id         int REFERENCES nodes(id) NOT NULL
);

CREATE TABLE genres(
    node_id int REFERENCES nodes(id) NOT NULL
);
```

(See `scripts/schema.sql` for source code.)

Knowledge Representation (KR) API

SQL Queries

In this section, we discuss the benefits and consequences of my database design choices with respect to the KR API; we also see a detailed list of the implemented API functions.

In the development of the KR API, we want to implement as much functionality with as few distinct SQL queries as possible. This goal is most clearly manifested in the API function `get_related_entities`, which allows the caller to retrieve related entities for a given entity (e.g. getting similar songs to a given song, getting genres to which a given artist pertains). A single SQL query template suffices:

```
SELECT name
FROM nodes
WHERE id in (
    SELECT dest
    FROM edges join nodes on source == id
    WHERE name = (?) AND rel == (?)
);
```

By inserting "Ariana Grande" and "is similar" into the inner `WHERE` clause, we can query for all artists similar to Ariana Grande. Similarly, by inserting "Ariana Grande" and "of genre", we can query for all genres with which Ariana Grande is associated. It is precisely the design choice of representing all entities as nodes that allows us to use this same template to support all requests for entities that have some relationship to some arbitary entity.

Naturally, there are some drawbacks to our schema design. For example, a feature of our design is that all common entity fields are found in the `nodes` table, which includes any given entity's name. So, if we want to find information about some artist by their name, we must `JOIN` the `nodes` and `artists` table on matching IDs (`nodes.id` and `artists.node_id`). In a traditional

schema design, all information directly pertaining to an artist would be entirely confined to the `artists` table, so this extra `JOIN` would be unnecessary.

The KR API

In general, the KR API exposes a collection of functions that encapsulate all SQL queries and logic relating to managing the database; through the KR API, callers may retrieve from and add information to the database. The following functions allow *addition* of information to the database through an instance `kr_api` of the `KnowledgeBaseAPI` class:

- `kr_api.add_artist()`: Inserts given values into two tables: `artists` and `nodes`.
  - Given artist is only added if they are not already in the database.
  - Given artist is either added to both tables or neither.
- `kr_api.add_song()`: Inserts given values into two tables: songs and nodes.
  - Given song is either added to both tables or neither.
  - Given artist is not ambiguous (only matches one 'artist' entry in nodes table).
  - Given tuple (song, artist) is only added if it does not already exist in database.
- `kr_api.add_genre()`: Adds given value into two tables: genres and nodes.
  - Given genre is eiter added to both tables or neither.
  - Given genre is only added if not already in the database.
- `kr_api._add_node()`: Adds given entity to the nodes table.
  - Intended for "private" use only, as indicated by the leading underscore in the function's name [8]
  - Used by all other `add_` functions.
- `kr_api.connect_entities(source_node_name, dest_node_name, rel_str, score)`: Insert into `edges` table.
  - Creates an edge in semantic network with given label `rev_str` (e.g. "similar to", "of genre") and score

And the following functions allow *retrieval* of information to the database

- `kr_api.get_all_music_entities()`: returns a list of names of all entities (songs, artists, and genres).
  - Used by UI modules to match user input with known entities.
- `kr_api.get_songs(artist)`: Retrieves list of song names for given artist name.
- `kr_api.get_artist_data(artist_name)`: Retrieves associated genres, node ID, and number of Spotify followers for given artist.
- `kr_api.get_song_data(song_name)`: Gets all songs that match given name, along with their artists.
  - List of all hits, each containing info about their node ID, artist (by name), duration in milliseconds, and popularity according to Spotify.

- `kr_api.get_related_entities(entity_name, rel_str)`: Finds all entities connected to the given entity by an edge with the given label `rel_str`.
    - This implements a key part of our KR system's functionality: **querying the semantic network!**
    - The given entity may be any of song, an artist, etc. The returned entity may or may not be the same type of entity.

(See `knowledge_base/api.py` for source code.)

Populating the Database

Spotify's Web API provided a highly practical way of gathering music data. Following the Authorization Guide [7] and consulting their API object documentation [6], I developed a minimal Spotify Web API client to gather initial music information. I implemented the client as an instantiable class with the following key functions:

- `client.get_artist_data(artist)`
- `client.get_top_songs(artist_ID, country_iso_code)`
- `client.get_related_artists(artist_ID)`

Thus, populating our database was simply a matter of using the Spotify client together with the KR API functions described earlier. (See the `scripts/spotify_client.py` and `scripts/test_db_utils.py` for source code.)

Future Work

- Within the KR API module, investigate the possibility of creating a standard set of reusable, composable SQL queries.
    - I wrote each KR API function and its corresponding SQL query in an ad-hoc manner. It very well may be possible to avoid repetition.
    - The advantage of this approach would be to avoid having distinct SQL queries to maintain for every special case.

Research Materials

1. Patterns of Enterprise Application Architecture
2. Representing Inheritance in a DB
3. ConceptNet
4. Python Programming & the Semantic Web
5. Knowledge Representation & Reasoning (Wikipedia)
6. Spotify Web API Object Model

7. Spotify Authorization Guide

8. Private Variables and Class-Local References (Python Docs)