ECE241 PROJECT 3: Pattern Matching

Due: Dec 20, 2018, 11PM on Moodle

Introduction:

In our world, we can often represent data as all kinds of strings. The words in the English language consist of 26 letters. The computer uses a two-symbol system which contains often the binary number system's 0 and 1. We usually represent DNA sequences as a string of nucleotides A, C, G and T. As a computer engineer, one of our primary task is to write programs analyzing and understanding the patterns of those strings.

Here we will analyze these sequences using a Finite Automata (FA) based pattern searching algorithm. In an FA based algorithm, we preprocess the pattern and build a 2D array that represents a Finite Automata. The construction of the FA is the tricky part of this algorithm. Once the FA is built, searching is simple. In search, we simply need to start from the first state of the automata and the first character of the text. At every step, we consider the next character of text, look for the next state in the FA and move to a new state. If we reach the final state, then the pattern is found in the text.

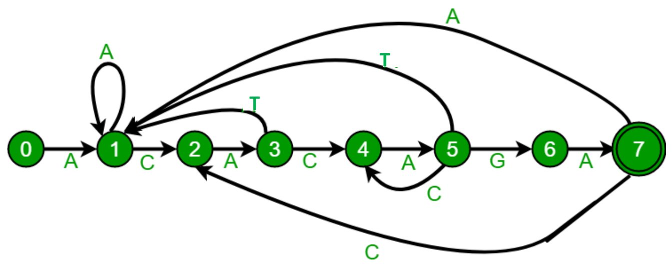
In this assignment, you will still read in a subset of the Million Song Dataset from an Excel file. Beside the fields such as title, artist, song duration, track ID and collaborative artists in the previous assignment, there is a new attribute as the feature of the song. The feature is encoded like a DNA sequence, which consists of A, C, G, and T. You need to analyze those DNA-like features to find the connections among those songs.

Task Overview

In this project, you will perform the following specific tasks.

1) Manage your data in Python classes and objects.
    a. You need to load the Songs library for Song object which contains the title, artist, song duration, track ID and DNA feature
    b. You need to load the coArtist graph similar to Project 2.

2) Build the following regular expression as a state machine (or finite automaton). This regular expression is used to test whether the DNA feature of a song will be accepted.



The regular expression is represented as a directed graph. Each node is a Vertex object. The edge between two nodes is an Edge object. Please use the DFA code in the lecture 16 slides (page 11-12) as the reference. In your graph (DFA class), the self.start is the starting point of the regular expression (Vertex0 in this example). The Vertex with self.isAcceptingState as true will be the final state (Vertex7 in this example).

3) Write a function *testMatch(self, seq)* to test whether a given string will be accepted by this FA graph. For example, a sequence ACACAGA will be accepted, the function will return True. A sequence ACACATT will not be accepted. The function will return False.

4) Write a function *testAccept(self, seq)* to return whether there is a suffix substring that will be accepted by the FA graph. For example, there is a song that contains a DNA feature as TTACACAGA. As we know the suffix substring ACACAGA will be accepted. So we consider this DNA feature can be accepted by the FA graph as well. By using this function, we will return 2 for TTACACAGA, which is the index of the first character of the accepted sequence. We will return 0 for ACACAGA and return -1 for ACACATT (not accepted for any suffix substrings)

5) Analyze the acceptance of the whole song library. In function *testSongLibrary()*, you need to test the acceptance of each song and return an array of the index (in *testAccept* function).

6) (**Bonus question 20 points**) As we know, programming is to translate our language (English) to another language that the computer can understand (Python). In this task, we look back a the single source shortest path problem. We want to implement the *BellmanFord*(self, s) algorithm in the ArtistConnections class based on the following pseudo-code.
In this task, the edges are the connections between coArtists, and the weight is the number of songs they collaborate.

```
    Algorithm BellmanFord(G, s)
1.  for all v ∈ G.vertList()    ### initialize the distance of all the vertex from source s
2.    if v = s
3.        setDistance(v, 0)
4.    else
5.        setDistance(v, ∞)
6.    for i in range(len(G.vertList()))
7.        for each edge e=(u, v) ∈ G.edges()
                        ### G.edges() is a collection of all the edges in the graph
8.            r ← getDistance(u) + weight(e)     ### relax edge e
9.            if r < getDistance(z)
10.               setDistance(z, r)      ### update the distance from u to z
```

Hints and suggestions

Successfully completing the project and achieving a good grade requires completing the project as described above and clearly commenting the code. As always, it makes sense to start the project early. Unless you are an amazing programmer, you probably won't be able to finish in one day. Build your project code step by step. For example, verify that you have successfully read the Song library and artist graph before attempting pattern matching and BellmanFord.

You can use the solution of your project 1 and 2 to load the SongLibrary and Artist Graph. Or use the solution from my code. Note that, there is no issue with the '\n' character in the coArtist list. You can use the normal way to parse the string as

```
artistList.split(';')
```

For ***BellmanFord*** function it runs much slower than the Dijkstra algorithm (Think about why). It may take several minutes to run for the whole graph. So here we make a small graph for test, we only read 500 song records (already in the skeleton code). Even so, it may take 5-10 second to finish the run. You can also think about some optimization strategies to accelerate the running. Like, you only need to relax the distance of your neighbor only when your distance is relaxed.

What to submit:

For Task 2-6, you should submit your code to Gradescope for auto-grading. Remember to comment your code properly.

Reminder: The course honesty policy requires you to write all code yourself, except for the code that we give to you. Your submitted code will be compared with all other submitted code for the course to identify similarities. Note that our checking program is not confused by changed variable or method names.

Grading:

- Code works on Gradescope (100%, including 80% for tasks 2-5 and 20% for task 6 bonus)
- Program structure and comments (20%)