CMPUT 291 Mini Project II

November 26, 2018

Mohammad Kebbi; 1496572

Andreea Muresan; 1498868

Westley Peterson; 1413570

Overview

This program allows the user to search for Kijiji ads in the command line. The user can input any number of expressions including a location, minimum/maximum/equal date, category, partial or full term, minimum/maximum/equal price, and whether the output should be brief (showing only ad ID and title) or full (showing ID, date, location, category, title, description, and price). The results will match all of the expressions entered. If there are no results, "No results" will be printed.

User Guide For Running Program:

1. For phase 1, entering '*python3 phase_1.py*' will prompt a user to enter a .txt file to read, and will write files for terms, pdates, prices, and ads.
2. For phase 2, entering '*./phase_2.sh*' or *'perl phase_2.sh'* will build the idx files needed for phase 3.
3. For phase 3, entering '*python3 phase_3.py*' will prompt the user to enter a query or queries and will print out data from the databases based on what they enter. Typing '*output=full'* will print out all information about the ad, and typing '*output=brief*' will only print out the ad ID and title of the ad.

Valid entries for phase 3:

match location:	"location = *location*"*
match (exact or range) price:	"price (=, >, <, >=, <=) *price*"*
match category:	"cat = *category*"*
match (exact or range) date:	"date (=, >, <, <=, >=) *YYYY/MM/DD*"*
match exact term:	"*exact term to match*"*
match partial term:	"*term*%"*
display brief output:	"output = brief"
display full output:	"output = full"
exit program:	"QUIT"

*italics represent varied user input

Algorithm Description

Our algorithm for evaluating queries starts by breaking down multiple entered queries within a list and evaluating them based on the order they are given. We compare each expression against our grammar checks to see what kind of query they input, to call the appropriate function. Each function returns the set of IDs that meet the conditions of the query. For the first expression, the *result_set* becomes the set returned from that function. After each function call, we intersect the *new_set* with the *result_set* so results that contradict all previous expressions are weeded out. If no given expressions apply to any ads, *result_set* is an empty set. Once all expressions have been evaluated, the program calls *print_out()* to print the results. If the result set is empty, "No results" is printed. The user can enter a new query or enter "QUIT" to return from the main function and exit the program.

For evaluating wild cards in *search_date_term()*, we check if a term a user enters contains the "%" symbol to search accordingly. We used Python's *str.startswith()* function to see if the entered term matches the beginning of one or more terms from the database. We used multiple strategies for evaluating and returning range searches. For evaluating less than entries in *less_than_date()* and *less_than_price()*, we start our iteration at the top of the database and add IDs to our set until we hit our value. If we are given a value that is not specifically within our database, we increment or decrement our search value until we hit a value that is within our database to evaluate the range. For evaluating greater than entries in *greater_than_date()* and *greater_than_price()*, we create a set of all values including and past our value and only return the entries that are not equivalent to our value. When evaluating operators that also contain the equals sign, we return a set with IDs that are equal to the user input and a set of IDs with values greater than or less than our range and then union them.

The efficiency of our algorithm is not ideal, as this strategy requires reading through entire databases multiple times when there are methods to evaluate parts of the database that fall under our conditions. As a result, this program runs slower the larger the database becomes.

Testing Strategy

For testing, we used both the 10 and 1k data sets for all three phases and used their values and data to make sure our program works properly.

For Phase 1, we ran our program and compared our resulting .txt with the ones from eclass. They resulted in perfect matches.

For Phase 2, we ran our program and ensured our data was properly sorted and that we were getting the right data put into our .idx files to work with phase 3.

For Phase 3, we used "*output = full*" to determine whether or not each query was returning the proper information. We used separate functions for each grammar check/query which made testing individual queries, and then group queries, fairly straightforward.

Group Work Break Down

The group collaboration was split into separate roles for each member as follows:

- Westley Peterson: Put together phase 3 and parts of phase 2 (15 hours)
- Andreea Muresan: Put together part 3 and formatted code (14 hrs)
- Mohammad Kebbi: Put together phase 1 and 2 and parts of phase 3 (14 hrs)

The method of coordination used to keep progress consistent and on track was to use Git and Github. We created a repository that held all of our project files where each member pushed their work consistently by creating working on individual files containing the functionality for specific parts.