CS1006 P01 – Autocomplete

170005153

Tutor: John Thomson
19th February 2018

Overview

For this practical the task was to write an autocomplete program which predicts a complete query as a user is typing a search. It should take the name of a data file to be searched and an integer k (denoting the maximum number of matching terms that can be displayed at once). Of the given set of terms, the matches should be ordered in descending order of weight.

The extensions that we made include displaying statistics about the number of matches and search execution time, dealing with special letters, adding a "Did you mean...?" message if no matches are found and showing the query and weight if the user double clicks an item in the list.

Build instructions: terminal command "java Project1 <input file> <integer number of results displayed>" in /src folder

Link to the repository: https://ag307.hg.cs.st-andrews.ac.uk/Project1

Design & Implementation

Various design and implementation choices were made in different classes as follows:

We kept the main class simple as it is only used to get command line arguments and create an array of terms. The only noticeable choice was to firstly put all the created Term objects to an arraylist as we do not exactly know the size of the array and only then could they be transferred to an array.

In Term class, the Term objects and comparators used to sort queries and weights are created. As indicated in the specification, some exceptions (*IllegalArgumentException* and *NullPointerException*) are thrown in the constructor. Usage of lambdas in the methods that return comparators was determined by the fact that it was a shorter and more elegant way than using the "new" command. Finally, we had to shorten words if they were longer than the inputted prefix and use the *toLowerCase* method to make sure that the candidate word from the array and prefix were compared correctly.

We used binary search in order to find matches for the given prefix. It is implemented in BinarySearchDeluxe. We chose to name variables *low* and *high* because they help to keep track of binary search boundaries.

In the Autocomplete class, we used the Java 8 method reference *Term::compareTo* as it was shorter syntax than a full lambda expression. The inputted string is then converted to a Term object with weight 0 as it is easier to compare two Term objects.

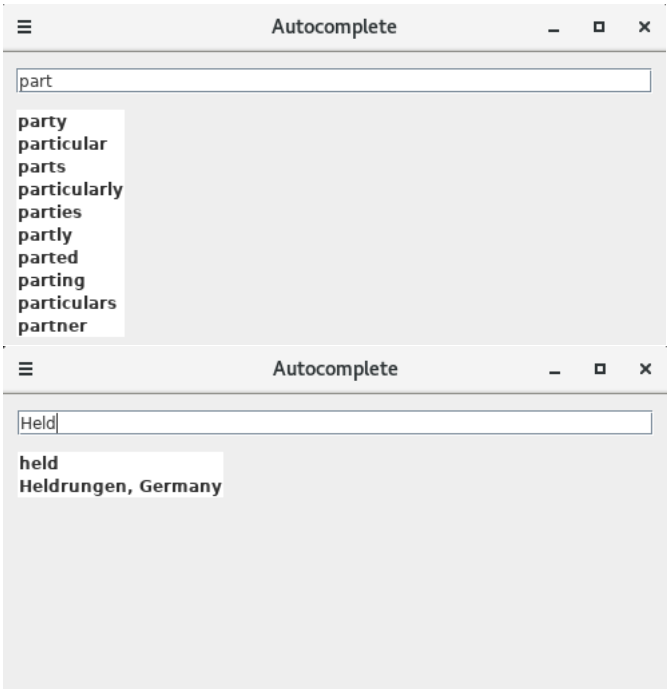
As specified by the project brief, we used Java Swing to produce a GUI for our autocomplete search engine. In the GUI class this consisted of an editable text field object to represent the search box, a list object to represent the autocomplete results and several labels to display additional information for our extensions. The size of window is proportional to the number of results displayed.

To handle GUI updates, we created a new DocumentListener object which implements DocumentListener and added an updateLabel method which takes a DocumentEvent parameter. We then retrieved text from the Document object which triggered the event (the text field) and used this as

the prefix parameter in the autocomplete search, before updating the list model with the matches this returns.

Testing

Screenshots showcase basic test cases:



Performance requirements

Initially, we went about analysing our program's performance by measuring the time required to update the list label using the System.nanotime method. We began this counter at the start of the Document event (when the user edits the search box) and ended it when the list model has been fully updated. From this, we concluded that the program takes far less than the 50ms mentioned in the brief to perform an autocomplete and update the list model.

Also, by timing this we observed that the string comparison function execution times are proportional to the number of characters needed to resolve the comparison. This is visible by appending to the search box entry (i.e. by holding down a letter key) and observing the execution time statistic grow proportional to the query.

The worst case scenarios for the binary search and binary search for first/last occurrences are different: in the former case it is when the match is not in the array ($\log n$ actions) and in the latter case it is when the first key is found at the end of the array ($1 + \lceil \log_2 n \rceil$ actions).

To sort the term array lexicographically, we used the Arrays.sort method therefore the complexity requirements are met. The sum of methods used inside the allMatches method is proportional to $\log n + M \log M$ compares. This consists of the first and last binary searches in addition to the Arrays.sort method.

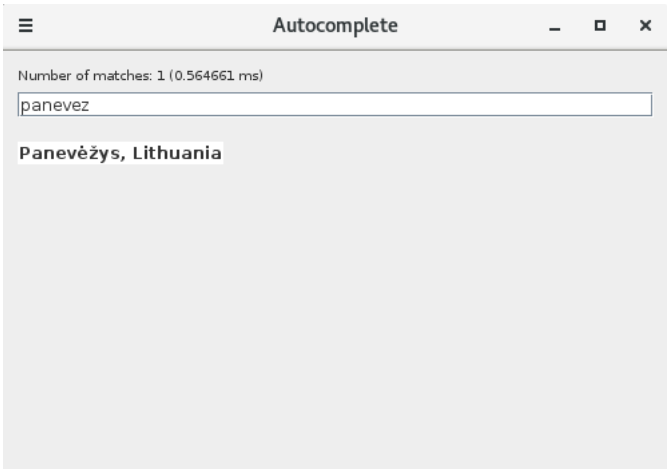
Extensions

Non-ASCII symbols:

While testing cities.txt file, we noticed that a lot of city names include non-ASCII symbols. Since the majority of users do not have all of these special characters on their keyboards, it is impossible

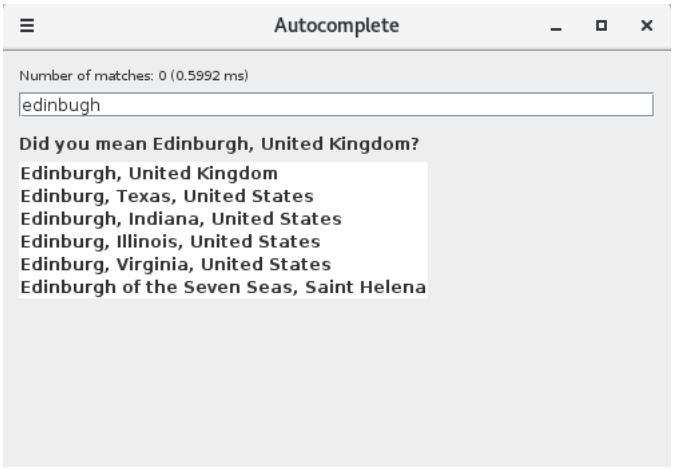
to find such cities using the main part of the program.

To fix this, we called a method from Normalizer class. It used Normalization Form D (NFD) to change diacritical marks to appropriate ASCII symbols. Normalized queries *queriesNorm* are stored in Term objects and are used in *compareTo* methods, but user still sees the original version of the query which is stored as *query*.



Invalid Prefixes:

To handle cases where the user has inputted an invalid prefix which yields no autocomplete results we decided to suggest results for a similar prefix (with a helpful "Did you mean... ?" message). To find this suggestion we iterate through substrings of the prefix (by decreasing the end index) until a valid, shorter prefix is found. The autocomplete function is then performed on this new prefix and a suggestion message is appended to the GUI above the result list.



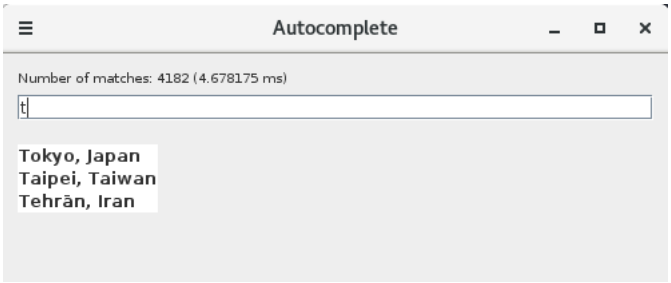
Interactive Results:

To enable our program to behave more like an actual search function we extended it to include interactive results. To achieve this we added the ability to double-click on a result in the list to display its query and weight fields alongside one another. We also included labels informing of the contextual meaning of each term to differentiate between word and city results.

This was implemented by adding a MouseListener object to receive a mouse click event from the JList component of our GUI. We overrode the mouse click method to add the query and weight values for the match to a new label. We also added a condition requiring the click count to be 2 (a double-click) to avoid the user accidentally selecting a result. The list model is then cleared and this new label, containing information about the selection, is displayed until the user edits the search or closes the program.

Search statistics:

To display statistics about how many matches have been found and how long the program took to return these results we added a label above the search box displaying this information. To calculate the time variable we used the System.nanotime method as outlined in the Performance Requirements section.



Personal input

For this practical, we wrote a working solution to the basic requirements together in person. We chose to work in this way for our first practical in order to learn about each other's programming strengths and weaknesses to allow for better delegation of tasks for future practicals. However, we did work separately on the extension tasks, and my personal contribution is as follows:

For my personal contribution to this practical, I implemented the 'Invalid Prefixes' and 'Interactive Results' extensions, as well as the execution time element of the 'Search statistics' extension. Due to this, I handled most of the GUI section of the practical and re-factored this code from the Project1 class into a separate GUI class to improve its legibility. I also commented out the GUI code as a result.

Conclusions

Overall, the program satisfies basic specifications and has several working extensions. Understanding comparators and Java Swing were the most challenging aspects of programming a solution to this practical. However, analysing performance requirements for this report proved even more difficult due to the breadth of the topic and our lack of knowledge of the maths involved when we started. Given more time, we would like to implement a search that can detect if a given search query is present at any position in a term query and return matches according to this.