Spell Checker: BKTrees, Levenshtein Distance

Catherine Tuntiserirat, Mick Kajornrattana, Billie Wei, Erika Puente

Version Control : https://github.com/CatherineT/CS51--SpellChecker
Video :https://youtu.be/tZJDVvmyvEs

Initial Draft Specification:
https://www.dropbox.com/s/juoewmlv758rpln/CS51Initial_DraftSpecification.pdf?dl=0
Final Draft Specification:
https://www.dropbox.com/s/lgsfdoq69s7pw8j/CS51Final_ProjectTechnicalSpecification.pdf?dl=0

Overview

We built a spell-checking program that takes in a typed word from user's keyboard and returns a list of top-suggested words for each of the misspelled words, line by line.

We built this program using the OCaml language and implementing a ***Burkhard-Keller Tree*** to store the words in our dictionary. The Burkhard-Keller Tree is a tree-based data structure engineered for quickly finding near-matches to a string, for example, returning "seek" and "peek" for "aeek" using edit distance. In order to index and search our dictionary (BK-Tree) we will be using some other algorithms to help calculate edit distance between two strings.

We implemented three versions of ***Levenshtein's Distance*** calculator, one of which unfortunately cannot be used with BK Tree for reasons to be discussed later. The distance calculating function takes in two strings and returns a number representing the minimum number of insertions, deletions and replacements (and transposition in Damerau-Levenshtein) required to translate one string into the other. Each node of the tree has an arbitrary number of children (an *n*-ary tree), and each edge has a weight determined by the Levenshtein Distance.

For extension feature, we optimized ways to calculate the closest string. A method we are using will use both Levenshtein's distance and probability that derived from the frequency of the word being used. For example, if input is "thew", then the suggested word should be ranked starting at "the" before "thaw" because "the" is more commonly used. We obtained these frequency values through Project Guttenberg, and used those values to narrow the list of suggested spellings that are returned to our user.

Planning

We ended up completing each milestone as planned in time. Here are our draft specification and final specification. Here are the summary list of the tasks that we aimed to accomplish in various drafts specifications and how they played out

Initial Draft Tasks:

1. Input reading feature (read from a text file)
2. Output writing feature (write a .txt file)
3. A structure to store dictionaries (BK Tree)
4. An algorithm to calculate the distance between two words - Levenshtein's.

Result:

1. Finished implementation of the BKTree.
2. Wrote comprehensive tests for the tree, including helper functions.
    - We have tested all functions, including helper functions (Except `multiple_search` which is closely derived from `search`)
3. Implemented Levenshtein distance using dynamic programming concept
4. Wrote a function to measure the performance of naive Levenshtein distance against the one using dynamic programming
5. Implemented Damerau-Levenshtein Distance
    - Checked. However, Damerau-Levenshtein Distance cannot be used with BKTree, because, unlike Regular Levenshtein Distance, it does not preserve "triangular inequality" property (d(x,y) + d(y,z) >= d(x,z)).
    - Click for more information or http://blog.notdot.net/2007/4/Damn-Cool-Algorithms-Part-1-BK-Trees

Final Specification Tasks:

1. Find a way to optimize distance calculation (if any)
2. Write distance function that integrates Levenshtein distance with probability that words appear in real text
3. Adjusting tolerance level to depend upon the length of the input (misspelled word)
4. Write documentation
5. Make presentation video

Result:

1. We optimized distance calculation by using probability. This entails...
    a. Writing a Python program (data\cleandict.py) to turn frequency in the dictionary (data\raw\fruquency_dict.txt) into probability

[The freuquency dictionary is obtained from: http://norvig.com/ngrams/count_1w.txt]

    i. Because this dictionary record most frequent words, some misspelled words like “coool” and “gooogle” are also in the dictionary.
    ii. actually has a lot of garbage words, we found another dictionary and “intersect” them (by cleandict.py).

  b. Using Peter Norvig’s article as a guideline to calculate probability from both frequency and edit distance. Look at **Appendix I** for the formula that we use to calculate probability.

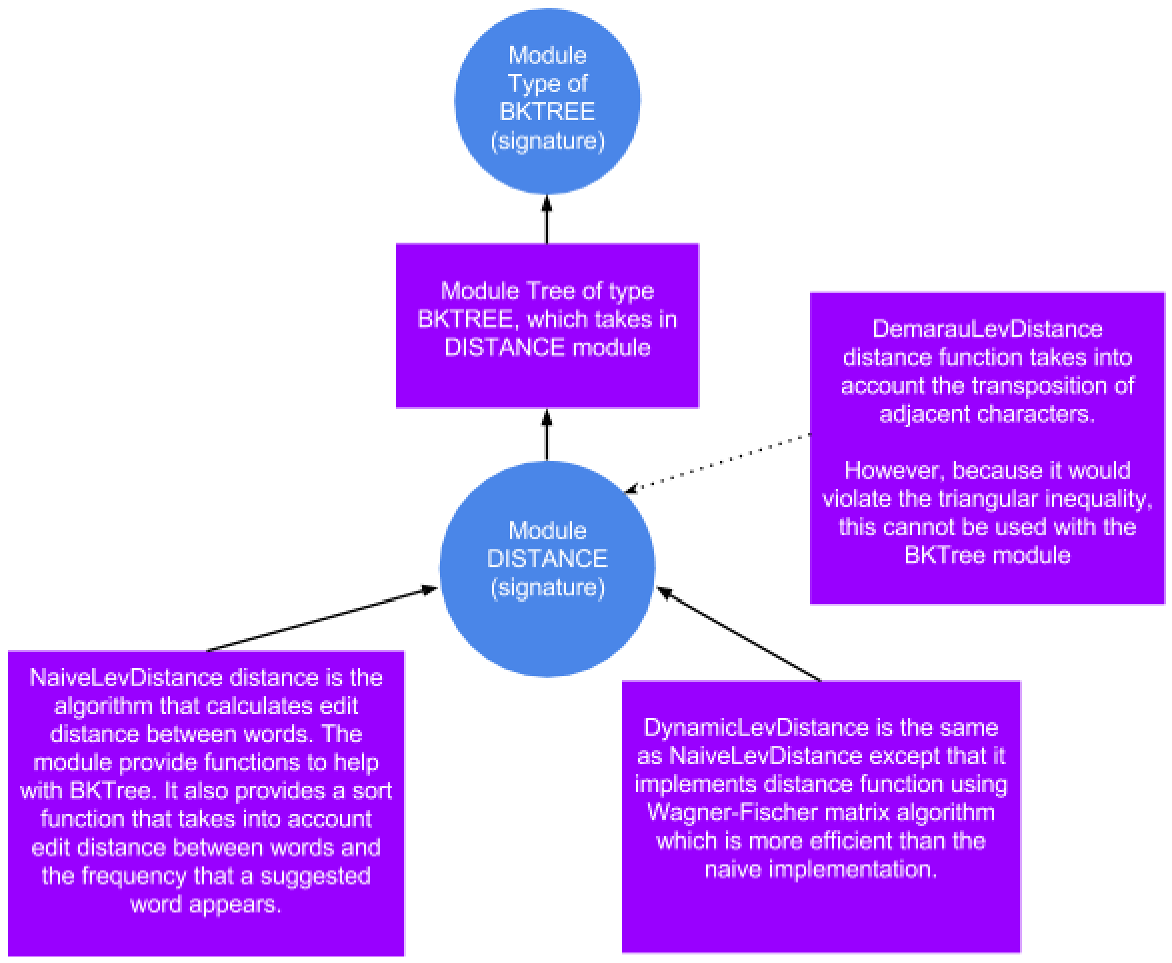
2. Tasks 2 is accomplished in 1.
3. We increase the tolerance level used in search function when a word is longer (default is set at +1 tolerance every 5 character)
4. We collaborated to finish writing up the documentation.
5. We created the video and then added in narration.

Instructions for Running and Testing the Spell-Checker

1. Unzip the submitted ZIP file.
2. You should be able see the following files:
    - README.md
    - info.txt
    - code [folder]
        - Makefile
        - project.ml
        - project.native
    - data [folder]
        - raw_data [folder]
            1. frequency_dict.txt
            2. scowl_dict.txt
        - cleaned_dict.txt
        - test_dict.txt
3. "cd" into the folder called "code".
4. Type "make clean; make" into your terminal.
5. Then run "./project.native".
6. The terminal will print the runtimes for loading test_dict.txt which is a 500-word dictionary and then read "Loading dictionary, please wait … "
7. Now an input line will appear. You can enter a string of word here and expect it to return the maximum of 10 suggested spellings for your input. The suggested words are ranked based on how likely a suggested word is intended given the input that you type. If your word is correctly spelled and is in our dictionary, it should return as the first of the list.

8. **Note**. Some misspellings will not return any results. The reason for this is our current set tolerance for misspellings. The tolerance level increases by 1 for every 5 characters. If your word is < 5 characters, the tolerance is 1, meaning the input can be up to 1 letter off from the correct spelling. If your word's length is between 5 and 9 inclusive characters, the tolerance is 2, meaning the input can be up to 2 letters off from any correct spelling in the dictionary.
9. **Note 2** Keep in mind that "correct spellings" depends on obtained dictionary files, which though extensive, might not include some specific terms and might include some non-proper English term
10. Some words we suggest for testing out:
    - smil3 [smile] → returns "smile"
    - hardbard [Harvard] → returns "hardware"
    - shandeliar [chandelier] → returns "chandelier"
    - raneboe [rainbow] → will not return any results because it exceeds the tolerance of 2
    - chrystanthemum [chrysanthemum] → returns "chrysanthemum"
    - moogle [moogle] → returns "google"

Design and Implementation


Module Type of BKTREE (signature)
Module Tree of type BKTREE, which takes in DISTANCE module
DemarauLevDistance distance function takes into account the transposition of adjacent characters.
However, because it would violate the triangular inequality, this cannot be used with the BKTree module
Module DISTANCE (signature)
NaiveLevDistance distance is the algorithm that calculates edit distance between words. The module provide functions to help with BKTree. It also provides a sort function that takes into account edit distance between words and the frequency that a suggested word appears.
DynamicLevDistance is the same as NaiveLevDistance except that it implements distance function using Wagner-Fischer matrix algorithm which is more efficient than the naive implementation.

The Levenshtein Distance Modulus

Testing: Every Distance module has several tests in it (most of which is inherit from NaiveLevDistance module) and the tests run every time that the program is run. To add more tests, you can add them at NaiveLevDistance module if they can also be used in all the three modules. However, tests for DamarauLevDistance module are different because, unlike the other two implementations, we need to take into account the transpositions of adjacent characters.

NaiveLevDistance
Calculates the edit distance (which becomes edge weights for the BK Tree) with a brute force approach. (Please see: https://web.stanford.edu/class/cs124/lec/med.pdf)

DynamicLevDistance

Calculates the edit distance using Wagner-Fischer matrix to help with run time at triviale expense of memory. (Please see: https://web.stanford.edu/class/cs124/lec/med.pdf)

DemarauLevDistance

Calculates edge weights for the tree while taking the transpoisition of words into account. However, because it would violate the triangular inequality of d(x,y) + d(y,z) >= d(x,z), this cannot be used with the BK Tree module. Click for more information

Using and Testing the Burkhard-Keller Tree

We built the tree following this article:
https://nullwords.wordpress.com/2013/03/13/the-bk-tree-a-data-structure-for-spell-checking/

However, our tree has the following type

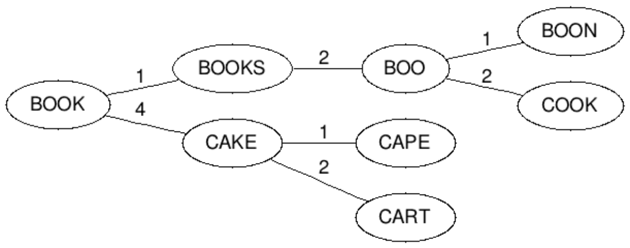
```
type branch = Single of d * (string * float)
                | Mult of (d * (string * float) * branch list)

type tree = Empty | Branch of branch
```

Where Single is a node with no child, where Mult is a node with least one children (children represents by branch list)

Our tree preserves 2 invariant principles

1. A parent node can not have more than 1 immediate child with the same edit distance.
    a. For example in illustration below, both "books" and "boo" , both have edit distance of 1 when compared to "book". However, both "books" and "boo" cannot be the immediate children of "book" (which is parent node). Instead, in this illustration, it shows that "books" was inserted first (and thus become the immediate child of book). Thus, "boo" which is inserted after has to traverse into "books" and become a more distant children of "book"



2. A Mult's branch list (list of its children) is sorted in ascending order based on edit distance between its child and itself.
   ex.
   ```
         Mult of ( 0, ("a" , some prob), [ (node of "aa"), (node of "aaa"),
   (node of "aaaaaa")...]
   ```

To use the modules and set things up, we call this tree **dict** for dictionary [See **module BKTree** of type **BKtree**] [**BKtree** is a functor that takes in one of the distance modulus we created and returns a **BKTREE**.] Then while we read inputs from user's keyboard we use **print_result** which in turn uses `search` to return a list of suggested words.

Other functions in our **BKtree** module include `insert` [for the admin side], which allows someone to manually insert words into the tree, with branches weighted by whatever distance module we have passed into **BKtree.** We also added a helper function `is_member` to help administrators to check if a word is in a tree.

Testing:
All tests (including that of helper functions) of BKTree runs everytime the program runs. To add more test, please modifies the `run_tests ()` function inside the BKTree module.
On our part, we wrote our tests by first mapping a sample tree, calculating Levenshtein distance for each tree's endges by hand and if `insert` build the tree as we intended. In order to test `search`, `is_member` and some other function, we picked sample words to test certain situations such as a distance of 0 or an outrageous distance of greater than 20 with words that were clearly not in our dict.txt file.

Reflection

How good was your original planning?
Our original planning was pretty decent. We ended up having the signature and implementation modules like we planned in the initial draft. We also have accomplished all of the extra tasks that we wanted to get to in the initial draft. Although there were some trouble in creating the tree type, we recovered from that just in time.

In terms of schedule and time planning. At first, we were a bit crunched since we had type-ups due at the exact same time as psets. It was really difficult balancing finishing the last few psets and implementing the project at the same time. We had some unexpected things that came up in people's schedules, but we were able to handle it pretty smoothly by distributing more work to those who are not as busy.

How did your milestones go?

We were able to meet all of our milestones and follow the implementation schedule that we set for ourselves. For the final project, we were also able to implement all of the extra-features that we wished to have

What was your experience with design, interfaces, languages, systems, testing, etc.?

We use OCaml for our project, and it has becomes a very handy language when we are creating both Distance module and the Tree module. The initial design for our signature or interfaces that we had were well planned, by the final version almost all of the interface functions are still there to help with testing and usage.

What surprises, pleasant or otherwise, did you encounter on the way?

One thing that surprised us was that the first implementation of Burkhard-Keller Tree, we realized the the tree is an n-ary tree. Specifically, in our n-ary BKtree, a parent node can have an infinite number of children, but each edges from this parent node must all have different weight (or edit distance). Look above for invariant principles explanation.

Because it is an n-ary tree, we need to think of a new type tree to represent it (so far we hav only had experience with binary tree). This is one of the most rewarding learning process (maybe because it was difficult for us), because we ended up with the types and invariant principles (look above) which keeps the tree organized and efficient.

What choices did you make that worked out well or badly?

Other than initially mistaking the tree to be binary tree and not n-ary tree, our original planning and divisions of work did not fall through as we have planned. Some of the group members ended up being more busy than the other and to ensure that we reach our milestones, we fixed the problem by re-distributing the work so that those who are currently free can work more, while those who can join later would also be able to contribute to something.

What would you like to do if there were more time?

If we had more time, we would further find a way to optimize the run time of our program. We would look around for a data structure that we could apply the Damerau-Levenshtein algorithm, or maybe just a better structure that helps with retrieval and insert time. Possibly, we might find a way to further enhance the search results to be more representative of what the user intended.

Next time, we would do things differently by …
We would probably form our groups earlier, and brainstorm ideas before the project even started. This would allow us to avoid the time crunch we experienced in choosing a suitable topic during that first week of the project.

The most important thing we learned from the project was…
We are glad to have some experience with implement an algorithm that we randomly find interesting, also the fact that we have learned a bit more about dynamic programing. And in general, It is more difficult to work on coding projects just because of the nature of the project itself, where multiple people are making complicated edits to the project files at the same time. We found that it was best to split the project into smaller chunks and have different people work on designated chunks would be more efficient.

Advice for Future Students

One thing we learned from this project is that it is really important to start early and do thorough research on potential algorithms and methods of implementation. It's important to really understand the algorithm you are planning to implement before you implement it, or else you might have to spend a while fixing and changing things that you've already spent a while on.

Starting early allows you time to really understand what you are trying to implement, and also time to change the overall plan if your project ends up being too complicated, too small in scope, or anything else.

Another minor but essential thing is to really figure out how Github works as early as possible. It is important that everyone is following the correct procedures when committing, pushing, or pulling from the repository. Although we did not have this problem, we can see how messing this up could create massive headaches for any group. Keep each other updated on when you push new code… and always commit then pull before you start working on a new part of the project. ALWAYS have a backup copy of your code. Put it on Dropbox, Google Drive, a flash-drive, anywhere. Do not risk losing the code you have worked so hard on!

Appendix I: Probability formula

This is based on Peter Norvig's article: http://norvig.com/spell-correct.html

Let $P(c)$ = Probability that the correct word 'c' (from dictionary) is intended.
Let $P(w)$ = Probability that the input word is type or given by user

We are trying to maximize $P(c|w)$ or the probabily that c is the word intended given that 'w' is typed in

Accoring to Bay's rule:

$$P(c|w) = \frac{P(w|c)P(c)}{P(w)}$$

However, since P(w) is the same for every word we can ignore that (Norvig) an instead get

$$P(c|w) = P(w|c)P(c)$$

We obtain $P(c)$ from dividing frequency at which 'c' appears by the total times every word in Gutenberg dictionary

For $P(w|c)$ or the probability that 'w' is type given that 'c' is intended, this estimates to the error model, and thus we can calculate this by turning edit distance between w and c into some sort of probability.

Using Norvig's implied method at which *"words of edit distance 1 are infinitely more probable than known words of edit distance 2, and infinitely less probable than a known word of edit distance 0"*, we are trying the scale diffrent edit distance at different level of probability.

Let d = distance between w and c
Let t = tolerance

According to our BKtree's search function, we know that the distance between input and each suggested word that it returns will not exceed a tolerance level (t) which we set.

Therefore,

$$\sum_{i=0}^{t} P(d = i) = 1$$

Where P(d=i) is probability assigned to be $P(w|c)$ when distance between w and c is i.

Next, we come up with a model to assign various sensible probabily to diffrent edit distance. For example the one that we used in our code is,

$$\begin{aligned}
P(d = t) &= a \\
P(d = t - 1) &= 100a \\
P(d = t - 2) &= 100^2 a \\
&\vdots \\
P(d = 1) &= 100^{t-1} a \\
P(d = 0) &= 100^t a
\end{aligned}$$

In here specifically, we assigned the word that has the most distance (t) to have some probability value a (which is really small), meanwhile those with distance of $t - 1$ to have probabilty of $100a$ which is a lot more likely to happen. This go on so that for those words that have distance of 0 (or exact same word) would have a lot, lot more probabily than that of distance equals to t.

To figure out a, we use

$$\begin{aligned}
\sum_{i=0}^{t} P(d = i) &= 1 \\
a(1 + 100 + 100^2 + ... + 100^t) &= 1 \\
a(\frac{100^{t+1} - 1}{99}) &= 1 && \text{(by Geometric series)} \\
a &= \frac{99}{100^{t+1} - 1}
\end{aligned}$$