Reverse Engineering Sublime Text's Fuzzy Match

March 28, 2016

Sublime Text is my favorite text editor for programming.

One of my favorite features of Sublime Text is it's fuzzy search algorithm. It's blistering fast at navigating to files and functions. Many people on the internet have asked how it works. None of the answers were satisfying. So I decided to figure it out myself.

(Note: Updated on 2/18/2017 with improved exhaustive search. See bottom for details.)

Interactive Demo

I've put an interactive demo that shows off my results. This versions uses Unreal Engine 4 filenames as it's dataset.

Give it a whirl. Then keep reading to learn how it works.

Search Pattern

abc

Results (Top 10)

1374 matches in 535.6 milliseconds

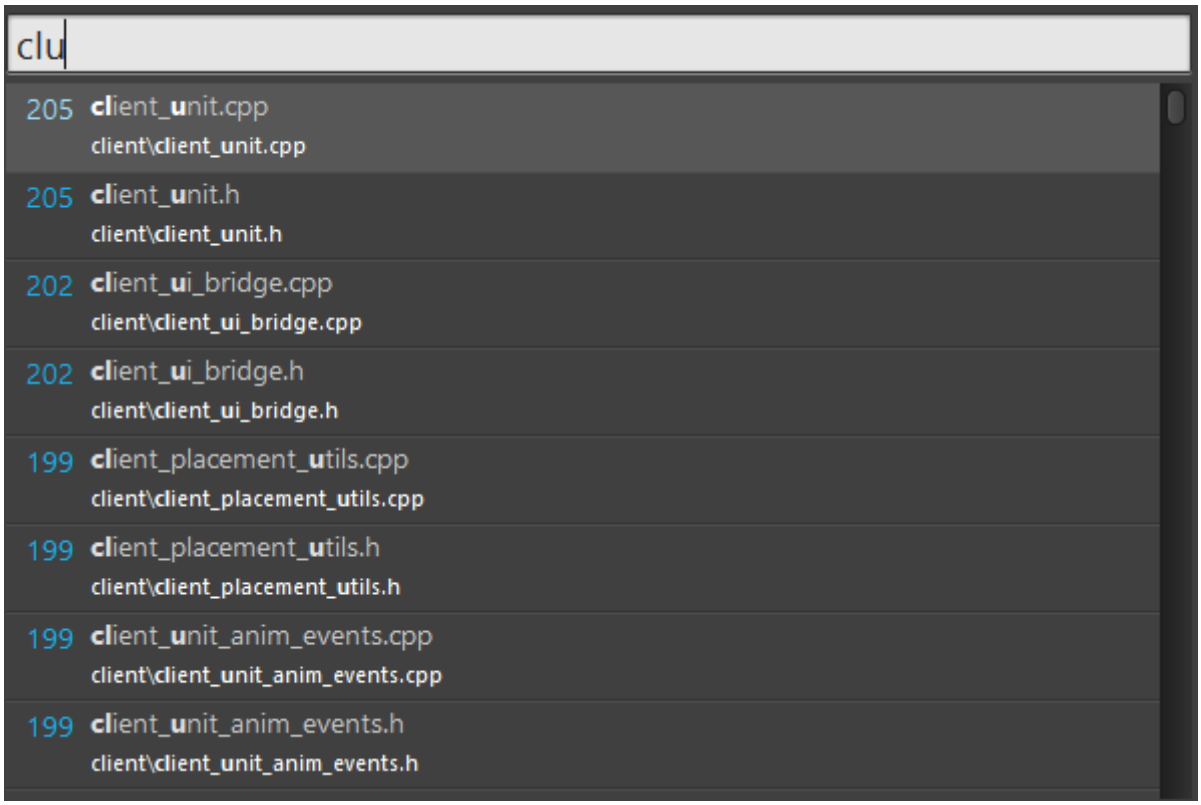
- 15 - **A**lpha**B**lend.**c**pp
- 10 - **A**nim**B**lueprint**C**ompiler.h
- 9 - **A**nim**B**lueprint**C**ompiler.cpp
- 7 - **A**LAudio**B**uffer.**c**pp
- 7 - **A**ISystem**B**ase.**c**pp
- 7 - **A**nimNode**B**ase.**c**pp
- 7 - T**ab**Commands.h
- 6 - **A**PIVaria**b**le.**c**s
- 6 - **A**nim**B**lueprint.**c**pp
- 6 - T**ab**Commands.cpp

Click here for a more featureful demo with multiple datasets and search options.

Sublime Workings

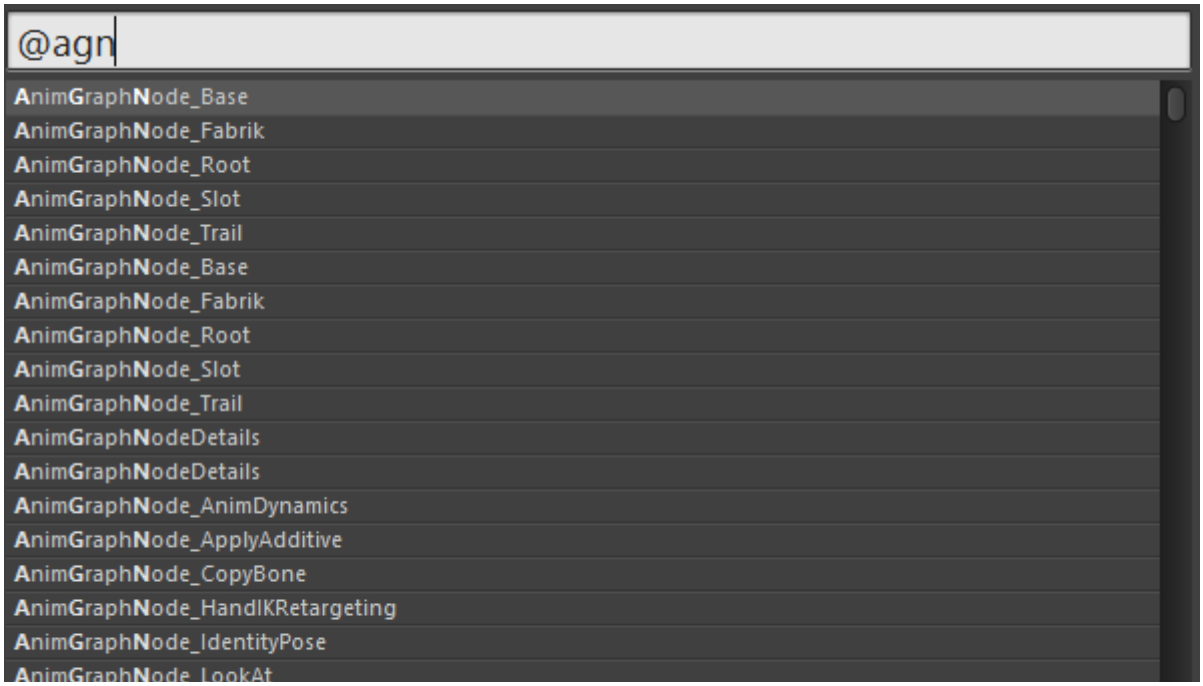
What is fuzzy matching in Sublime Text? And what makes it so cool? I'm glad you asked.

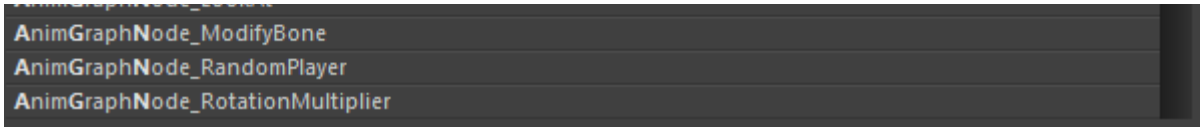
Sublime has two super handy navigation functions. One for finding files and one for finding symbols (functions, class names, etc). Both work the same way. Instead of having to type the file name exactly right you can just type a few letters. Those letters are cleverly matched to create a ranked results list. Here's an example.



This is a file search. I entered the search pattern 'clu'. The top most result is '**cl**ient_**u**nit.cpp'. The matched characters in each result are in bold.

Here's another example.





Typing 'agn' reveals many AnimGraphNode types. Most items can be uniquely identified with only a handful of key strokes.

Inspiration

Sublime's fuzzy matching is great. It's super great. I love it to pieces. Tragically, nothing else uses it. Not other text editors. Not IDEs. Not websites with search fields. Nothing. It should be everywhere but it's nowhere.

I'd like to change that. First, I want to first unravel the mystery of the fuzzy match. Second, I want to provide source code that can be used by existing projects to improve their search.

I have a few specific use cases in mind. I want to support programming. File names, classes, functions, etc. But that's not all.

I'm an avid Hearthstone player. It's common to search for cards. You search in-game when building decks. Many players also search online. Sites like HearthArena help in draft mode. I'm also a big fan of card database sites like Hearth.cards.

Most Hearthstone related sites only perform basic substring matching. Does the card name "Ragnaros the Firelord" contain the substring 'rag'? Yes it does. But so too does "Inner Rage", "Faerie Dragon", "Magma Rager", and many more. Being able to type "rtf" or "ragrs" would be so much faster and better.

It also needs to be fast. It should run interactively while testing against tens of thousands of entries.

Functionality

If we experiment with Sublime Text there are two things which are immediately clear.

1. Fuzzy matching tries to match each character in sequence.
2. There's a hidden score where some matched characters are worth more points than others.

We can implement the first part easily. Let's do it!

```
// Returns true if each character in pattern is found sequentially within str
static bool fuzzy_match(char const * pattern, char const * str)
{
    while (*pattern != '\0' && *str != '\0')  {
        if (tolower(*pattern) == tolower(*str))
            ++pattern;
        ++str;
    }

    return *pattern == '\0' ? true : false;
}
```

Voila! In my library I've included this simple version for both C++ and JavaScript. I did so for a very specific reason. It can trivially replace simple substring matching. (cough cough. Slack emoji search. cough cough.)

Scoring

The fun part is the hidden score. What factors get checked and how many points are they worth? First, here are the factors that I currently check for:

- Matched letter
- Unmatched letter
- Consecutively matched letters
- Proximity to start
- Letter following a separator (space, underscore)
- Uppercase letter following lowercase (aka CamelCase)

This part is straight forward. Matched letters are good. Unmatched letters are bad. Matching near the start is good. Matching the first letter in the middle of a phrase is

good. Matching the uppercase letters in camel case entries is good.

The tricky part is how many points these factors are worth. I believe there isn't a single correct answer. Weights depend on your expected data set. File paths are different from file names. File extensions may be ignorable. Single words care about consecutive matches but not separators or camel case.

That said I think I found a decent balance. It works great against several different data sets. I highly suggest looking at the source code.

- Score starts at 0
- Matched letter: +0 points
- Unmatched letter: -1 point
- Consecutive match bonus: +5 points
- Separator bonus: +10 points
- Camel case bonus: +10 points
- Unmatched leading letter: -3 points (max: -9)

There is some nuance to this. Scores don't have an intrinsic meaning. The score range isn't 0 to 100. It's roughly [-50, 50]. Longer words have a lower minimum score due to unmatched letter penalty. Longer search patterns have a higher maximum score due to match bonuses.

Separator and camel case bonus is worth a LOT. Consecutive matches are worth quite a bit.

There is a penalty if you DON'T match the first three letters. Which effectively rewards matching near the start. However there's no difference in matching between the middle and end.

There is not an explicit bonus for an exact match. Unmatched letters receive a penalty. So shorter strings and closer matches are worth more.

That's almost it. For a single search pattern results can be sorted by score. It works well. I encourage you to check out the demo if you haven't already.

Performance

Performance

Grep is fast. Really fast. It's highly optimized and doesn't need to test each letter. It can skip ahead.

Fuzzy match is not as fast as grep. It needs to test every letter in the search string. And while I've written what I would consider clean code it has not been ruthlessly optimized. There's a certain focus on readability for educational purposes.

My home CPU is an Intel i5–4670 Haswell @ 3.4Ghz. Matching a pattern against 13,164 file names found in Unreal Engine 4 takes ~5 milliseconds on a single thread. Testing against an English word list with 355,000 words takes ~50 milliseconds. (It was 30ms until the secret sauce improvement.)

JavaScript is not as fast as C++. In fact it seems to be about 25x slower. I am a video game programmer who knows nothing about webdev. There may be some obvious room for improvement. An async helper is provided so script doesn't block on slow searches.

Closing Thoughts

I love Sublime Text and it's fuzzy match algorithm. My first goal was to create something equally effective. I think I achieved that goal.

My second goal was to package that solution onto GitHub in such a way that other people can benefit. I don't know if I achieved that goal. I hope so. If you found this post useful or informative please let me know. If you use this code in any way I'd love to hear about it. If you want to fork to specialize the code to a specific use case by all means.

Interactive Demo: Click Here
Source Code: C++ ; JavaScript
GitHub: lib_fts

Thanks for reading.

Update — 2/18/2017

This project has been updated based on feedback from Sublime Text's Jon Skinner. The algorithm now performs an exhaustive search to find all possible matches and returns the match with the highest score.

Consider the string " SVisualLoggerLogsList.h" and the search pattern "LLL". There are four L's so they can be matched several different ways. The naive approach might match the first three L's. A higher scoring match would skip the first L and match the three CamelCase L's.

```
String: SVisualLoggerLogsList.h
    Pattern: LLL

Possible Matches (in bold):
    SVisualLoggerLogsList.h
    SVisualLoggerLogsList.h
    SVisualLoggerLogsList.h
    SVisualLoggerLogsList.h
```

The new exhaustive method finds all matches and returns the one with the highest score. Performing an exhaustive search is slower than finding just the first match. However the small decrease in speed is more than made up for by the increase in result quality.