Damn Cool Algorithms, Part 1: BK-Trees

Posted by Nick Johnson | Filed under coding, tech, damn-cool-algorithms

This is the first post in (hopefully) a series of posts on Damn Cool Algorithms - essentially, any algorithm I think is really Damn Cool, particularly if it's simple but non-obvious.

BK-Trees, or Burkhard-Keller Trees are a tree-based data structure engineered for quickly finding near-matches to a string, for example, as used by a spelling checker, or when doing a 'fuzzy' search for a term. The aim is to return, for example, "seek" and "peek" if I search for "aeek". What makes BK-Trees so cool is that they take a problem which has no obvious solution besides brute-force search, and present a simple and elegant method for speeding up searches substantially.

BK-Trees were first proposed by Burkhard and Keller in 1973, in their paper "Some approaches to best match file searching". The only copy of this online seems to be in the ACM archive, which is subscription only. Further details, however, are provided in the excellent paper "Fast Approximate String Matching in a Dictionary".

Before we can define BK-Trees, we need to define a couple of preliminaries. In order to index and search our dictionary, we need a way to compare strings. The canonical method for this is the Levenshtein Distance, which takes two strings, and returns a number representing the minimum number of insertions, deletions and replacements required to translate one string into the other. Other string functions are also acceptable (for example, one incorportating the concept of transpositions as an atomic operation could be used), as long as they meet the criteria defined below.

Now we can make a particularly useful observation about the Levenshtein Distance: It forms a Metric Space. Put simply, a metric space is any relationship that adheres to three basic criteria:

- d(x,y) = 0 <-> x = y *(If the distance between x and y is 0, then x = y)*
- d(x,y) = d(y,x) *(The distance from x to y is the same as the distance from y to x)*
- d(x,y) + d(y,z) >= d(x,z)

The last of these criteria is called the Triangle Inequality. The Triangle Inequality states that the path from x to z must be no longer than any path that goes through another intermediate point (the path from x to y to z). Look at a triangle, for example: it's not possible to draw a triangle such that it's quicker to get from one point to another by going along two sides than it is by going along the other side.

These three criteria, basic as they are, are all that's required for something such as the Levenshtein Distance to qualify as a Metric Space. Note that this is far more general than, for example, a Euclidian Space - a Euclidian Space is metric, but many Metric Spaces (such as the Levenshtein Distance) are not Euclidian. Now that we know that the Levenshtein Distance (and other similar string distance functions) embodies a Metric Space, we come to the key observation of Burkhard and Keller.

Assume for a moment we have two parameters, *query*, the string we are using in our search, and *n* the maximum distance a string can be from *query* and still be returned. Say we take an arbitary string, *test* and compare it to *query*. Call the resultant distance *d*. Because we know the triangle inequality holds, all our results must have at most distance *d+n* and at least distance *d-n* from *test*.

From here, the construction of a BK-Tree is simple: Each node has a arbitrary number of children, and each edge has a number corresponding to a Levenshtein distance. All the subnodes on the edge numbered n have a Levenshtein distance of exactly n to the parent node. So, for example, if we have a tree with parent node "book" and two child nodes "rook" and "nooks", the edge from "book" to "rook" is numbered 1, and the edge from "book" to "nooks" is numbered 2.

To build the tree from a dictionary, take an arbitrary word and make it the root of your tree. Whenever you want to insert a word, take the Levenshtein distance between your word and the root of the tree, and find the edge with number d(newword,root). Recurse, comparing your query with the child node on that edge, and so on, until there is no child node, at which point you create a new child node and store your new word there. For example, to insert "boon" into the example tree above, we would examine the root, find that d("book", "boon") = 1, and so examine the child on the edge numbered 1, which is the word "rook". We would then calculate the distance d("rook", "boon"), which is 2, and so insert the new word under "rook", with an edge numbered 2.

To query the tree, take the Levenshtein distance from your term to the root, and recursively query every child node numbered between d-n and d+n (inclusive). If the node you are examining is within d of your search term, return it and continue your query.

The tree is N-ary and irregular (but generally well-balanced). Tests show that searching with a distance of 1 queries no more than 5-8% of the tree, and searching with two errors queries no more than 17-25% of the tree - a substantial improvement over checking every node! Note that exact searching can also be performed fairly efficiently by simply setting n to 0.

Looking back on this, the post is rather longer and seems more involved than I had anticipated. Hopefully, you will agree after reading it that the insight behind BK-Trees is indeed elegant and remarkably simple.

02 April, 2007

Comments

Like and 41 others liked this.

Add New Comment

Login

Showing 87 comments

Sort by popular now

Adam Parkin

Personally I really like the blog post, simple to follow, good descriptions & motivation. Diagrams are always cool, but not necessary to convey meaning and/or intent. One tip I would give is to avoid the "block of text" when explaining an algorithm. For example:

"To build the tree from a dictionary, take an arbitrary word and make it
the root of your tree. Whenever you want to insert a word, take the
Levenshtein distance between your word and the root of the tree, and
find the edge with number d(newword,root). Recurse, comparing your query
with the child node on that edge, and so on, until there is no child
node, at which point you create a new child node and store your new word
there. For example, to insert "boon" into the example tree above, we
would examine the root, find that d("book", "boon") = 1, and so examine
the child on the edge numbered 1, which is the word "rook". We would
then calculate the distance d("rook", "boon"), which is 2, and so insert
the new word under "rook", with an edge numbered"

This whole paragraph would be better served in point form, or even pseudocode. Makes it easier to pull out the step-by-step procedure of the algorithm.

5 months ago 3 Likes Like Reply

Wbrehaut

I agree almost completely except that a few diagrams are essential to anything but a linear structure.

Your point on presenting algorithms in algorithm format instead of long, dense narrative paragraph format is most important, though.

5 months ago in reply to Adam Parkin Like Reply

NItin

A terrible post for an Algorithm Description. Wasted 10 minutes.

5 months ago 3 Likes Like Reply

Nick Johnson

A terrible comment lacking in constructive criticism. Wasted 30 seconds.

5 months ago in reply to NItin 17 Likes Like Reply

Mason

Here's my two cents:

The article started off great, a good definition of where the foundations for the idea come from.

A) Levistien distance isn't explained at all.

B) Once you get into the actual algorithm, it's all words. I, like many engineers I've talked to, tend to be bad at parsing large parts of chunks. To solve this, adding diagrams which repeat the information in these paragraphs would help immensely.

C) Your paragraphs imply a large amount of 'random' behavior in insertion. In particular, insertion and querying should have some sort of pseudocode which concisely describes the operations which need to occur, and in which order.

Thanks for putting in the effort for this series, and I hope to see more from you.

5 months ago in reply to Nick Johnson 1 Like Like Reply

Nick Johnson

Thanks for the feedback. I don't think it's in scope to explain how levenshtein works, and that's why I link to the Wikipedia article on it. If I tried to explain every relevant computer science topic in one of my articles, I'd never get to the actual point.

I wrote this article a long time ago, and I think if you check out my more recent articles you'll agree that my writing style has improved since.

Insertion behaviour _is_ random, and nobody's been able to find a better way to pick nodes when constructing a tree than random selection, funnily enough.

5 months ago in reply to Mason Like Reply

Philip Tellis

I for one thought it covered the topic well. Just broad and deep enough to understand what it's about and links to point to more information for the interested reader.

5 months ago in reply to Nick Johnson 2 Likes Like Reply

mr majs

You seem like a very nice person, but I also thought that the description was as fuzzy as the algorithm. Perhaps you should add a few colorful pictures! That would really brighten up the page.

5 months ago in reply to Nick Johnson Like Reply

Michael

He provided a data point. Constructive criticism is optional.

5 months ago in reply to Nick Johnson Like Reply

Nathan Ramsey

Manners are not optional.

5 months ago in reply to Michael 8 Likes Like Reply

Bryan Corell

This Data structure isn't mean for idiots. I mean, the metric algorithm is a dynamic programming problem, which is then wrapped into a non-trivial tree structure. Of course its hard you morons.

5 months ago 2 Likes Like Reply

Eli

I'm not sure if anyone's mentioned this yet, but I think it's really important to point that in the paragraph:

"Assume for a moment we have two parameters, query, the string we are using in our search, and n the maximum distance a string can be from query and still be returned. Say we take an arbitary string, test and compare it to query. Call the resultant distance d. Because we know the triangle inequality holds, all our results must have at most distance d+n and at least distance d-n from test."

d-n should be abs(d-n) since you can set which one is d and which is n arbitrarily. This should save you a ton of time in your search of the tree. Assuming the tree is approximately well balanced, it means you can find iterate through everything within a distance of w from some search term in O(t/(t-w)) time where t = the total number of words in the tree.

5 months ago 2 Likes Like Reply

MrTrick

Needs more diagrams.
I lost track after 'From here, the construction of a BK-Tree is simple.'

5 years ago 9 Likes Like Reply

Nick Johnson

Good point. I'll add some if I get the chance.

5 years ago in reply to MrTrick Like Reply

jqb

No chance in four years?

9 months ago in reply to Nick Johnson 10 Likes Like Reply

MrTrick

Hrm... must be caching problems, I don't see your update...

5 years ago in reply to Nick Johnson 8 Likes Like Reply

Barry

Thanks for this. I have some questions though:
1) How long does it take on average to build a tree from a dictionary
2) Assume I want to find best matches with an incorrectly spelt English word (lets assume the dictionary we will be using is the old familiar English dictionary), how would I be able to calculate its best match quickly (what would I use a root)?
3) Lastly, how big (in terms of memory) is say a BK tree used for an average spell checker (I assume you have this face readily available).
Thanks again for the fascinating topic.

4 years ago 2 Likes Like Reply

MrTrick

Holy WTF.

4 years ago in reply to Barry 2 Likes Like Reply

Ilya1113

Great Article!
I'm building this for OCAML if anyone needs it.

1 month ago Like Reply

jhhga

Interesting post - I've been playing with some related algorithms for word game solvers.

So here's a potential tweak on the algorithm above - what if we weighted the answers by a measure of "distance" from the original key (assume a standard american keyboard). An answer of teek is more likely to be reek or geek (correct words) than peek or meek.

Can think of a number of potential applications of this approach...

3 months ago Like Reply

Nick Johnson

Yup! That approach will work just fine as long as the distance is integral, and obeys the triangle inequality (which it will if you're just defining a set of valid edits like that).

3 months ago in reply to jhhga Like Reply

jhhga

Joking aside - got an implementation question, since you're also a python guy. What is the "smart" way to implement a graph / tree data structure behind a web application? Assume we're talking about something where it's ok if it take a few cycles to "assemble" (eg. can do it at server startup) but needs to run quickly for each page view...

This doesn't seem to fit neatly into a SQL database....

3 months ago in reply to Nick Johnson Like Reply

Nick Johnson

That depends how you intend to query and update it. I don't think the comments thread on BK-Trees is the right place to have this conversation, though - post it to SO with more details and we can have a conversation about it there. :)

3 months ago in reply to jhhga Like Reply

jhhga

Hah... such an approach can be used for good or evil (eval?). Either the ultimate spelling corrector or an automated process for finding long tail SEO keywords with an above-average expectation of traffic....

3 months ago in reply to Nick Johnson Like Reply

Ross

Nice article. Well-written and clear.

4 months ago Like Reply

MasonBially

This actually helped me solve a problem I have been having with fuzzy matching in our StarCraft AI project. And while I realize this is a bit of an old post, it was very helpful what I was doing, thank you!

4 months ago Like Reply

Kiranstalin

very useful blog. A few diagrams would have simplified it even more but still its easy to understand.

5 months ago Like Reply

John Hart

Nice article. It might help some readers to understand _why_ BK trees are useful here is that you can precompute a BK tree for your entire dictionary using a random root word, and then use that BK tree for any search term you want. That is, you don't have to generate a new BK tree for each new search term (which of course would be slower than a brute-force search).

(I realize this is obvious to some, but may not be obvious to all).

Depending on the desired edit distance threshold, it might be simpler to generate all strings of that edit distance (per Norvig's "spell corrector in 21 lines") and then compare these against your corpus.

5 months ago Like Reply

Wbrehaut

If you had a diagram or two I would havegiven you an A+!

5 months ago Like Reply

Chris K

Nitpick: it's "recur", not "recurse".

5 months ago Like Reply

Derek Rhodes

great post, keep them coming.

5 months ago Like Reply

Song Zheng, My name is Song Zheng

It was hard for me to follow, my attention span got me through only the first half of the article. However, I bookmarked this for future read. I'm very interested in algorithms.

I'm really glad you're doing this, but I feel like algorithms are better taught via video or diagrams, visual description of each steps will be alot easier to follow.
Also, it would be really helpful if you post a table of contents of all the algorithms you plan to talk about, it would be helpful to satisfy curious readers like me =D

=D

5 months ago Like Reply

Anonymous

BK Trees are okay... but use w/ edit distance is still rather expensive. For nearest neighbor searches using Levenshtein, a trie or DFA works much better. The cost of comparison at each node adds up. Granted, it's better than brute-force, but not by much.

5 months ago Like Reply

Nick Johnson

Have you read my other post, on Levenshtein automata? It also quantifies the speedups. BK-Trees are significantly faster than brute-force - the original BK tree paper demonstrates that.

5 months ago in reply to Anonymous Like Reply

Steve Hanov

What if you are searching things that have floating point distances? It is worth mentioning that Vantage Point (VP) trees are a generalization of BK trees. VP trees are slightly simpler to implement and can deal with items that have non-integer distances.

To build a VP tree node: 1. Choose an arbitrary item. 2. Each child of that node has a left and right child. The left contains all of the items that are closer than the median distance away, and the right contains items that are farther. Recurse.

To search a node, we calculate the distance from the search query to the node. In most cases we can discard the left or the right if we are too far from the "middle".

VP Trees can search items that have thousands of dimensions -- for example, images or music.

10 months ago Like Reply

Nick Johnson

VP trees are interesting, but not a generalization of BK trees. Being able to find the median of a set presumes a total order, which BK trees don't require. What is the median element of a dictionary with respect to Levenshtein distance between words?

10 months ago in reply to Steve Hanov Like Reply

Steve Hanov

Sorry, I was being too brief. To build one node of the VP tree we first choose an arbitrary item, and then sort all of the items by distance from the item that we chose (similar to the BK Tree). This does not require a total ordering of the items themselves, only on their distances from the distinguished item (called the "vantage point)". Then, instead of putting them into integer buckets as in the BK tree, they are simply partitioned into two sets -- those farther away than X, and those closer than X. It is most efficient to choose X as the median of the distances. X is then stored in the tree node and used in the search.

It is best explained in the paper: http://pnylab.com/pny/papers/v... . There is an implementation for levenshtein distances here: http://www.logarithmic.net/pfh....

I am writing because I loved your article when I found it, but I deal with high dimensional floating point data. I struggled for some time through all kinds of trees and VP turned out to work best for me due to its simplicity, speed, and low storage requirements.

10 months ago in reply to Nick Johnson 1 Like Like Reply

Nick Johnson

Ah, I misunderstood. Yes, that would certainly work with non-dimensional data. Thanks for the link!

10 months ago in reply to Steve Hanov 1 Like Like Reply

Nicholas Miller

Nice work buddy, came in real handy.

11 months ago Like Reply

sarah

The paper "Some approaches to best match file searching" is availble as a PDF here: http://citeseerx.ist.psu.edu/v...

1 year ago Like Reply

Shrederroy

Hi, I don't know if anyone else has pointed this out, but there are a couple of small technical errors. The Levenshtein distance is not a metric space; it is a metric. A metric space with the Levenshtein distance as the underlying metric would be a set, such as the set of strings in the English language, together with the Levenshtein distance. In other words, the metric space would be the ordered pair M = (W, L), where W = {w | w is a word in the English language} and L: W x W -> N U {0} is the Levenshtein distance mapping an ordered pair of two words into the natural numbers together with zero.

1 year ago Like Reply

Anonymous

Wow, that's incredibly helpful and bizarrely timely (I'm working on a project that this will help alot). Thanks to both you and StumbleUpon for the BK-Trees algorithm!

2 years ago Like Reply

Fuad Efendi, Two Beer Or Not Two Beer!

So many coments today, what's up?

Look for "Deterministic Finite Automaton" (DFA) and "Trie" (reTRIEval) articles at Wikipedia. BK-Tree is toy for kids in comparison: http://issues.apache.org/jira/...

2 years ago Like Reply

Maxime Henrion

Great post! I couldn't resist implementing those trees in Haskell, so here we go:

http://gist.github.com/324580

The query function implements approximate matching the way you are describing it. I discovered afterwards that there's an existing package on Hackage that implements BK trees, but surprisingly, it doesn't seem to provide a function for approximate matching.

2 years ago Like Reply

Maxime Henrion

Actually, there is such a function in the bktrees package, it's called elemsDistance, I had missed it.

2 years ago in reply to Maxime Henrion Like Reply

Tomas

Nice post.

2 years ago Like Reply

Vojislav Stojkovic

Great post! Is there any recommendation on how to choose the root word?

2 years ago Like Reply

Nick Johnson

I tried several methods to pick nodes when I was implementing this, and without exception they started off promising, but had terrible worst cases. Random selection seems to be the best option - finding a better one seems to be an open subject for research.

2 years ago in reply to Vojislav Stojkovic Like Reply

Claus Sørensen

Thank you for an interesting post :) It seems to be a really neat and useful structure!

2 years ago Like Reply

ducky

I have implemented Porter-Stemming and spellchecking through pSpell in a Search engine, and am looking to make it more robust.

This article is excellent, and I will now be investigating BK-Trees and how I can incorporate them in my search engine.

Thank you so much for an informative article.

2 years ago Like Reply

Abednego

Cool. Thanks. I thought I'd implement this.
http://shgypsy.com/tools/BkTre...

2 years ago Like Reply

Fuad Efendi, Two Beer Or Not Two Beer!

https://issues.apache.org/jira...
https://issues.apache.org/jira...

- based on some scientific papers, DFA/DNA/Automaton, and seems to be much faster than BKTree for some use cases (RegEx, wildcard queries, etc), but I haven't tried it yet...

2 years ago Like Reply

Fuad Efendi, Two Beer Or Not Two Beer!

NFA/DFA are proper terms... I am wondering if Google uses it:
http://rcmuir.wordpress.com/20...
http://www.mitpressjournals.or...
http://citeseerx.ist.psu.edu/v...

2 years ago in reply to Fuad Efendi Like Reply

Jonas Kress

Well done, thanks!

2 years ago Like Reply

Fuad Efendi

I believe this simple code in Java is self explanatory: http://www.tokenizer.ca/commun...

It can be used with any kind of distance function, not just Levenshtein. And, with any kind of objects, not just String.

Thanks for inspiring me!

2 years ago Like Reply

Nick Johnson

Neat! It's always good to see practical implementations. You should probably clarify that it will work with any distance function, as long as that function obeys the triangle inequality, though. :)

2 years ago in reply to Fuad Efendi Like Reply

Fuad Efendi, Two Beer Or Not Two Beer!

Thanks! I contributed code to http://issues.apache.org/jira/...
- Java-based full-text indexing framework
And I did some research: looks like http://www.catalysoft.com/arti... is 5x times faster and logically more appropriate than Levenstein algo for many "text" use cases

2 years ago in reply to Nick Johnson Like Reply

thejay

The article from catalysoft is describing the well known q-gram algorithm for fuzzy string matching. But there's no mention of it whatsoever in the article.

2 years ago in reply to Fuad Efendi Like Reply

MrTrick

C# implementation:
[url]http://www.smartcorner.eu/post...[/url]

3 years ago Like Reply

MrTrick

C# implementation:
http://www.smartcorner.eu/post...

3 years ago Like Reply

MrTrick

BK-Trees implementation

3 years ago Like Reply

regsitesph@yahoo.com

thanks for describing it !

4 years ago Like Reply

Paul Battley

After reading this post, I wrote a simple BK-Tree implementation in Ruby, and did a talk about it at Euruko this year.
The source code is available via Subversion: svn co http://paulbattley.googlecode.... bktree
[url=http://po-ru.com/presentations...]My slides[/url] (which probably won't look good in anything other than Firefox, I'm afraid) are also online.

4 years ago Like Reply

tonys

Very neat algorithm - thanks for describing it !
I've just tried out a slightly optimized version of it that reduces the maximum distance every time it finds a better 'match' so that the tree query function returns only a list of the best possible matches.
Rough testing seems to indicate that the optimization only offers an occasional/slight improvement when the query distance is 2 but then gets progressively more useful as

the distance gets bigger.
The optimization is pretty obvious but doesn't seem to be in the "Fast Approximate.." paper - do you perhaps know if there are lurking problems ?

4 years ago Like Reply

dan

It's good to have some diagrams and pictures when explaining such things. Otherwise good post :)

4 years ago Like Reply

russ

Sounds great but a quick search for a C++ implementation didn't brig up anything useful. Do you have any links to C/C++ implementations of BK-Trees?

4 years ago Like Reply

DaveG

Also lost it at "From here, the construction of a BK-Tree is simple"; you introduce 'edge' suddenly with no explanation, which appears to be critical as the article progresses. Prior to that point it was very well explained :)

4 years ago Like Reply

KiLVaiDeN

Cool information, thank you ! A little bit of code would have been particularly nice to enhance your article though, but well already explaining the principle is good enough :)
The major lack of this algorithm is speed... The Levenshtein distance algorithm is pretty complex.
If you don't know about them, sometimes it's easier to use SOUNDEX kind of algorithms, which are based on phonetics and can provide good results. Lot of variants exist, and most databases include those by default !
Cheers
K

4 years ago Like Reply

Volkan YAZICI

It'd be really awesome if somebody would implement BK-Tree's in PostgreSQL using GiST.

4 years ago Like Reply

MrTrick

Nice read... Keep it up!

4 years ago Like Reply

manthrax

Awesome article. Thank you!

4 years ago Like Reply

marcos

Really interesting. Thank you.

4 years ago Like Reply

pp

Sexy! Thanks.

4 years ago Like Reply

MrTrick

"distance a string can be from query and still be returned. Say we take an arbitary string, test and compare it to query. Call the resultant distance d. Because we know the triangle inequality holds, all our results must have at most distance d+n and at least distance d-n from test."
Eh, shouldn't that be:
"all our results must have at most distance *n-d* and at least distance d-n from test."
?

4 years ago Like Reply

klausnrooster

Pseudocode would clarify the build process. Damn interesting either way though!

4 years ago Like Reply

Alexander Jerusalem

Thanks for that! It's very useful and well written. Obviously this could be used with any distance measure, not just Levenshtein. For instance, it seems interesting to use a hierarchical distance function based on unicode categories, so that distance values derived from diacritics and case are always smaller than those derived from variance in base characters.

4 years ago Like Reply

MrTrick

Hi Nick,
very nice explanation. I enjoyed reading it.
Thanks.

4 years ago Like Reply

John

Thanks! That was a great introduction to a structure i had not yet come across, but one that helps with a project i'm working on now! Cheers :)

4 years ago Like Reply

MrTrick

nitpicking
[quote]Now we can make a particularly useful observation about the Levenshtein Distance: It forms a Metric Space. Put simply, a metric space is any relationship that adheres to three basic criteria: (...) [/quote]
These criteria define a *distance*. A metric space is a *tuple* of (S,d) where S is a set and d a distance.

4 years ago Like Reply

Dom

So Google doesn't want smart people to know about search engine theory? I wonder why????

4 years ago Like Reply

Close Protection World

diagrams as mentioned would help alot

4 years ago Like Reply

MrTrick

I also don't follow. Maybe some illustrations of the data structure and handling a query would help.

4 years ago Like Reply

robert

hopefully providers such as c5 or powercollections will pick up on this and push it in as well....
or maybe we will offer...
great post sir...
btw: http://www.itu.dk/research/c5/

4 years ago Like Reply

MrTrick

Real good blog. Not a math person, but love to here about algo's like this. Worded well also. Easy to undestand. Keep up the good work

4 years ago Like Reply

MrTrick

they are also called vp-trees vector space trees

4 years ago Like Reply

Ole

Well done!, I have to say that information about bk-trees structure is extremly scarce on the Internet. Most web pages indexed by google :-) speaks about "using bk-trees" but not "how they are build".
Congratulations.

5 years ago Like Reply

MrTrick

[url=http://well-adjusted.de/~jrsch...]BKTree by Jochen Schulz[/url] is a Python implementation of this algorithm, as a part of a larger library. Comments at the url cited above, source code available at [url]http://well-adjusted.de/~jrsch...[/url] and beyond.

4 years ago in reply to Ole Like Reply

Subscribe by email RSS

Reactions

blog comments powered by **DISQUS**

Blogroll

- Nick Johnsonz
- Bill Katz
- Coding Horror
- Craphound
- Neopythonic
- Schneier on Security