Today, I am going to talk about structure similarity search for formulas, using leaf-root paths in operator subtrees.
My name is Wei, this is joint work with my advisor Prof. Zanibbi.

As it may be unfamiliar to mainstream IR people,
Our study domain is focusing on formula, it belongs to MIR. Which includes ...

What is searching by similarity in a math context? To give you an idea, i think it is best to give an example.
So here is one of the systems I have published earlier. The system accepts text keywords combined with math formulas, and retrieve relevant document.
Unlike the computational search engine Wolfra Alpha, these systems require very little math knowledge, except for some very simple commutativity, associativity rules embedded, we almost rely on purely visual looking.

So, in this field, we have some problems that are unique.
For example, the bag of words model may not be applied directly because math content are highly structural.
At least in our view, identifying the common subexpression is the key of math similarity search, but bag-of-words model is clearly not sufficient to know one subset of symbols belongs to one subexpression, and another set belongs to another subexpression.
Also, the frequency based scoring which most text search has based on, is also not suitable here, because symbols in formula can be replaced interchangeably. And again due to highly structural content, proximity may not imply two symbols are "near" to each other, because they can belong to subexpression far away from each other.
And the list will go on as you can see there are many fundamental differences compared to text search.

This field has already been developed for quite some years, the state of art approach now are all taking tree representations for math formula search and similarity scoring. Because it captures the nature structure of formulas.
There are two commonly used representations, on the left is SLT, it captures the topology and layout of formula, it has little ambiguity, but it is hard to capture order mutations of operands.
On the contrary, the one on the Right is operator tree, it lifts the operators to the top, so that you may use leaf to root paths to identify the expression without worrying about the reordering of operands. However, it needs to be parsed with the presence of much more ambiguity cases.
In this paper we are focusing operator tree.

Again, to give you a sense of what we do in the paper, here is an example. Don't worry if you cannot see it clearly, this is a real life example where we have identified three common subtrees between two expressions. In the following slides we will tell you how we identify them and how we are going to use those found common subtrees to score their similarity.

What we do first is we define what do we mean by common subtree, in our context, it means common formula subtree in particular.
What is common formula subtree? You can think it as common subtree with additional constraints that require them matching

from bottom up, and must start from the leaves.
And we define our [formula tree similarity] to be: find the common forest of two OPTs, such that it maximize a given scoring function.
The score of maximizer is our defined similarity.
In this paper we simply count the number of nodes matched, and apply different weights on different subtrees (beta weights) and differentiate operators and operands in the tree (the alpha weights).

OKay, we have defined the similarity, but how to calculate it? It actually requires finding subtree isomorphism.
I know this audience are mainly in IR field, but it is fine, what you need to know is just finding general isomorphism in tree is at least product of polynomial time, there are, however, an exact linear algorithm, but it requires matching the subtree outdegree, which means a+b+c will not match a+b, which is not desired in our case.

So given such a non-trivial problem, if we want to search in hundreds of thousands of documents in one second, we will have to simplify the problem. We are going to use leaf-root path as our search units, and we assume that if leaf-root paths match, the induced tree are identical in structure.

This assumption may fail as you can see in this example, highlighted in green are two trees with identical leaf-root path set, but they are structurally different, because the right one has an additional internal node. But nevertheless, we made this assumption.

Apart from that, after finding the identical subtree, we are also looking for whether or not there are other common subtrees.
Look at this example, by identifying another common subtree highlighted in red, we can know that T1 and T2 are more similar than T1 and T3, because of that additional red tree.
To find all the common trees, we assume the score of a tree is written as a ordered tuple, where each element is the "width" of corresponding subtree. So here in this example, score of T1 and T2 are 3 leaves matched in green, and 2 leaves matched in red.
Comparing the sore is essentially a tuple comparison, where you compare the first value first, if it ties, compare the second and so on. So this tuple score is essentially setting alpha to zero and care only about the number of leaves, and letting beta1 much greater than beta 2 and so on.
Here we only consider 3 subtrees, you may adjust it to other constants, but this is a efficiency consideration.
As an additional note, our intuition of this assumption is now that we have the path as search unit, the easiest thing we can find out of matched path is the width of tree, so having this assumption makes it easier to identify the best matched common forest.

However, just like the first assumption, this assumption will fail sometimes. Suppose you have two trees like this, and you choose to match the two green trees here because you cannot find a wider common subtree, and you find you are down.
However, if you choose to align this green tree, you may be able to match another tree here highlighted in red.

So, in reality, the greedy widest match may not yield defined tree similarity exactly.
But nevertheless we will use this assumption.

Under these two assumption, we can easily identify structural matching, and we can easily find the best common forest that represents our defined similarity.
The process is like this:
****
we will find the widest match by counting the number of matched path grouped by root-end node.
****
And the harder part is counting the operators, once we have found all matched leaves, we go through the hit paths again, this time we will identify paths "inside" one of the matched subtrees. And count the unique root-end node again, and the number should be the number of operators.
Then we will plugin every number we get into a scoring formula used for ranking, which I will show in the next couple slides.

So, once you decompose a tree into paths, we construct an inverted index, the hit paths are those paths from query tree, and those paths from document tree that are the same and of same document expression ID.
For example in this figure, ****

After identify the hit paths, we will find the matched paths grouped by the root-end node. Here we visualize it in a table.
The first column, ***, the first row ***
And what you have in the table, each table cell then represents a matched common subtree.
According to your second assumption, you just need to find the cell with most number of matches, and that one corresponds to the best matched single tree. And then you try to find another non-conflict matched tree.
You now have the best tuple score as maximizer, by the second assumption again, you can say any subtree on top of these leaves are the common forest we wanna find.

So what left is the operators, then go through the cells again, if one cell is intersect its leaves with matched tree, then it is inside of that tree, we will map the root-end node as matched. In this example, the light green cell is obviously intersect its leaves with the dark green cell on leaf b and c. So we know the light green subtree sits inside of the dark green tree.
In this way we will find all the internal node mappings.
As additional note, we can use bitmap to test the intersection of leaves if we assume the number of leaves are less to a constants.

Because we are counting the number of nodes to measure the matched size, some nodes we find will greatly bias our model, in particular, the subscript and superscript are very noisy. In our grammar, they consist 4 nodes in total, but they do not get rendered in the expression and thus we avoid counting them.

Now let's look at our ranking score, we already have what we call formula tree similarity, right?
**
And we have so far all using structure to measure similarity, but

we still want E=mc^2 when we search the exact symbol right?
So we also want symbol set similarity incorporated into our final ranking score.
We put very thing into a F-measure form and penalize the document formula size by a theta parameter.

So in evaluation we use the most recent math similarity search contest,
**

The effectiveness of different parameters are summarized in this figure. It is what I extracted from the paper.
Highlighted in green are the run labels assigning uniform contribution credits to different matched trees, we find it very ineffective. This also implies to assign more credits to larger match makes a lot sense.

And we also find in single tree matching, if we put more weights but not entire weights on operands match, we get a very high performance.

Even if our best results are achieved in the multi matching tree in 2-beta* and 3-beta runs, it has very small gap over the single tree matching.

And we also tried to evaluate how many tree matching in a single expression is sufficient, we distribute even weights on trees with the number of matches to the maximum from 1 to 5, and we observe very different performance behaviors among different queries, some queries benefit from multi-tree, some do not.
This may help to explain a little bit about why the multi tree matching does have a small gap above single tree matching, if you look at each query and average the performance, the increasing and decreasing queries kind of cancelled out. So it really depends on queries, what we find when we pull out all the queries from the table is, those complex queries tend to be hurt from multi-part matching and those simple and composite queries tend to benefit from multi-part matching.

When we compare with other systems, our bpref is able to outperform others, please notice that even single tree matching can outperform others, which implies even our simplified version can achieve very good results using path-based model.
And ***

As for efficiency, we are not quite satisfying at this point, it takes still multi-seconds to run some queries, but compared with Tangent-S, which has a non-linear time aligning algorithm (as you may see in the left-most figure), we have far less outliers in long queries, and because we are not doing any pruning so far, the improvement potential is very huge, we believe.

So, conclusion, we have shown path-based methods, even single tree matching, can achieve very good results, and we have observed some interesting effects of different parameters.
Our efficiency is not good, but it has potential to improve especially if you consider there are very long query in math search, you may find dynamic pruning algorithm very good to try.
In fact, we will improve efficiency as the next step **