Accelerating Substructure Similarity Search for Formula Retrieval

Wei Zhong[1,✉], Shaurya Rohatgi[2], Jian Wu[3],
C. Lee Giles[2] and Richard Zanibbi[1,✉]

[1] Rochester Institute of Technology
{wxz8033,rxzvcs}@rit.edu
[2] Pennsylvania State University
[3] Old Dominion University

Abstract. Formula retrieval systems using substructure matching are effective, but suffer from slow retrieval times caused by the complexity of structure matching. We present a specialized inverted index and rank-safe dynamic pruning algorithm for faster substructure retrieval. Formulas are indexed from their Operator Tree (OPT) representations. Our model is evaluated using the NTCIR-12 Wikipedia Formula Browsing Task and a new formula corpus produced from Math StackExchange posts. Our approach preserves the effectiveness of structure matching while allowing queries to be executed in real-time.

Keywords: Math information retrieval, Query processing optimization, Dynamic pruning.

1 Introduction

In information retrieval, a great deal of research has gone into creating efficient search engines for large corpora. However, few have addressed substructure search in structural content, e.g., in Mathematical Information Retrieval (MIR) [21] where efficient substructure similarity search is needed to identify shared subexpressions effectively. For example, in math formula search, to discern that $a + b$ and $b + a$ are equivalent (by commutativity), but that $ab + cd$ and $a + bcd$ are different, applying tokenization and counting common token frequencies is insufficient. Instead, a hierarchical representation of mathematical operations is needed and we may want to identify shared substructures.

In the most recent math similarity search competition,[4] effective systems all take a tree-based approach by extracting query terms from tree representations. For example, an Operator Tree (OPT) is used in Figure 1 to represent math formulas where operands are represented by leaves and operators are located at internal nodes. This facilitates searching substructures shared by two math expressions. For example, we can extract paths from their tree representations and find their shared subtrees by matching their common paths grouped by subtree

[4] The NTCIR-12 Wikipedia Formula Browsing Task.

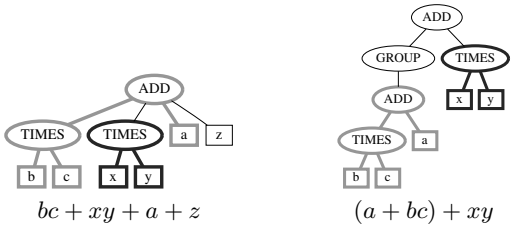

Fig. 1. Operator trees (OPTs) for two similar formulas. OPTs represent the application of operations (at internal nodes in circles) to operands (at the leaves in squares). Two common substructures are highlighted in black and gray.

root nodes. However, in order to carry structure information, it is common to see structural queries with over tens or even hundreds of path tokens which is unusual for normal fulltext search. This makes query processing costly for realistic math search tasks.

In text similarity search, query processing can be accelerated through dynamic pruning [18], which typically estimates score upperbounds to prune documents unlikely to be in the top K results. However, effective substructure search requires additional matching or alignment among query terms, and this makes it hard to get a good score estimation and it prevents us applying traditional dynamically pruning effectively. In fact, reportedly few state-of-the-art MIR systems have achieved practical query run times even when given a large amount of computing resources [11, 20]. In this paper we try to address this problem by introducing a specialized inverted index and we propose a dynamic pruning method based on this inverted index to boost formula retrieval efficiency.

2 Related Work

Recently there has been an increasing amount of research on similarity search for math formulas, with most focusing on search effectiveness [5, 7, 11, 23]. There are many emerging issues regarding effectiveness, including handling mathematical semantics, and identifying interchangeable symbols and common subexpressions. However, the efficiency of math formula search systems is often not addressed.

A number of MIR systems apply text search models to math retrieval, extracting sequential features from formulas and use variants of TF-IDF scoring [12, 14, 16]. These approaches incorporate a bag-of-words model, and use frequency to measure formula similarity. Inevitably, they need to index different combinations of sequences or substrings to handle operator commutativity and subexpression identification. This index augmentation results in a non-linearly increasing index size in the number of indexed "words" [12] and thus hurts efficiency for large corpora. On the other hand, recent results [10, 20, 23] reveal that effective systems for formula retrieval use tree-based approaches distinct from text-based methods. However, tree-based systems usually need to calculate

costly graph matching or edit distance metrics [9,22], which generally have non-linear time complexity. Recently, a path-based approach [23] was developed to search substructures in formula OPTs approximately by assuming that identical formulas have the same leaf-root path set. Although at the time of writing, it obtains the best effectiveness for the NTCIR-12 dataset, the typically large number of query paths means that query run times are not ideal - maximum run times can be a couple of seconds.

Dynamic pruning has been recognized as an effective way to reduce query processing times [2, 8, 13, 18]. Dynamic pruning speeds up query processing by skipping scoring calculations or avoiding unnecessary reads for documents which are unlikely to be ranked in the top K results. Pruning methods can be based on different query processing schemes: Document-at-a-time (DAAT) requires all relevant posting lists be merged simultaneously. Term-at-a-time (TAAT) or score-at-a-time (SAAT) processes one posting list at a time for each term, requiring additional memory to store partial scores, and posting lists in this case are usually sorted by document importance (e.g, impact score [1]), with promising documents placed at the front of inverted lists. Pruning strategies are *rank-safe* (or *safe up to rank K*) [19] if they guarantee that the top K documents are ranked in the same order before and after pruning. The most well-known rank-safe pruning strategies for DAAT are MaxScore [8,17,19] and WAND variants [3,6]. Shan et al. [15] show that MaxScore variants (e.g. BMM, LBMM) outperform other dynamic pruning strategies for long queries, and recently Mallia et al. [2] report a similar finding over a range of popular index encodings.

3 Preliminaries

Baseline Model This work is based on our previous work [23] which extracts prefixes from OPT leaf-root paths as index or query terms. The OPT is parsed from a formula in LaTeX. For indexed paths, they are mapped to corresponding posting lists in an inverted index where the IDs of expressions containing the path are appended. For query paths, the corresponding posting lists are merged and approximate matching is performed on candidates one expression at a time. The similarity score is measured from matched common subtree(s).

Because math symbols are interchangeable, paths are tokenized for better recall, e.g., variables such as a, b, c are tokenized into VAR. In our tokenized path representation uppercase words denote token types, which may be for operators as well as operands (e.g., TIMES for symbols representing multiplication). In Figure 1, when indexing "bc + xy + a + z," its expression ID (or ExpID) will be appended to posting lists associated with tokenized prefix paths from its OPT representation, i.e., VAR/TIMES, VAR/ADD and VAR/TIMES/ADD. At query processing, the shared structures highlighted in black and gray are found by matching these tokenized paths (two paths match if and only if they have the same tokenized paths, for example, "a/+" and "z/+" can be matched) and common subtree roots are identified by grouping paths by their root nodes. As

a result, the posting list entry also stores the root node ID for indexed paths, in order to reconstruct matches substructures at merge time.

At query time, the similarity score is given by the size of matched common subtrees. Specifically, the model chooses a number of "widest" matched subtree(s) (e.g., $a + bc$ is the widest matched in Figure 1 because it has 3 common leaves and is "wider" than the other choices) and measure formula similarity based on the size of these common subtrees.

The original Approach0 model [23] matches up to three widest common subtrees and scores similarity by a weighted sum of the number of matched leaves (operands) and operators from different common subtrees $\hat{T}_q^i, \hat{T}_d^i$ of a common forest π. Operators and operand (leaf) nodes weights are controlled by parameter α, while the weight of rooted substructures from largest to smallest are given by β_i. In the following, $|\cdot|$ indicates the size of a set:

$$\sum_{i=1}^{3} \beta_i \left(\alpha \cdot \left| \text{operators}(\hat{T}_d^i) \right| + (1-\alpha) \cdot \left| \text{leaves}(\hat{T}_d^i) \right| \right) , \qquad (\hat{T}_q^i, \hat{T}_d^i) \in \pi \tag{1}$$

Interestingly, while multiple subtree matching boosts effectiveness, using just the widest match still outperforms other systems in terms of highly relevant results [23]. The simplified similarity score based on widest common subtree between query and document OPTs T_q, T_d is the widest match $w^*_{Q,D}$, formally

$$w^*_{Q,D} = \max_{\hat{T}_q, \hat{T}_d \in \text{CFS}(T_q, T_d)} |\,\text{leaves}(\hat{T}_d)| \tag{2}$$

where $\text{CFS}(T_q, T_d)$ are all the *common formula subtrees* between T_q and T_d. In addition to subtree isomorphism, a *formula subtree* requires leaves in a subtree to match leaves in the counterpart, in other words, subtrees are matched bottom-up from operands in OPTs. In Figure 1, the value of $w^*_{Q,D}$ is 3, produced by the widest common subtrees shown in gray.

Dynamic Pruning In dynamic pruning, the top K scored hits are kept throughout the querying process, with the lowest score in the top K at a given point defining the threshold θ. Since at most K candidates will be returned, dynamic pruning strategies work by estimating score upperbounds before knowing the precise score of a hit so that candidate hits with a score upperbound less or equal to θ can be pruned safely, because they will not appear in the final top K results. Moreover, if a subset of posting lists alone cannot produce a top K result from their upperbounds, they are called a *non-requirement set*, the opposite being the *requirement set*. Posting lists in the non-requirement with IDs less than the currently evaluating IDs in the requirement set can be skipped safely, because posting lists in the non-requirement set alone will not produce a top K candidate.

4 Methodology

In this paper, we apply dynamic pruning to structural search. As structure search has more query terms in general, we focus on a MaxScore-like strategy suggested

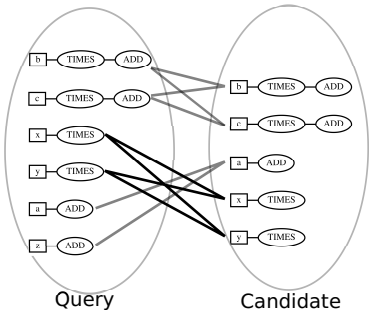

Fig. 2. Bipartite graph of hit path set for formulas in Figure 1 (original leaf symbol is used here to help identify paths). Edges are established if paths from the two sides are the same after tokenization. Edges with shared end points (i.e., same root-end nodes) in original OPTs have the same color (black or gray).

by [2,15], since they do not need to sort query terms at merge iterations (which is expensive for long queries). Our approach is different from the original MaxScore, as upperbound scores are also calculated from the query tree representation. We also use the simplified scoring equation (2) where a subset of query terms in the widest matched common subtrees $\hat{T}_q^*, \hat{T}_d^*$ contribute to the score. In contrast, typical TF-IDF scoring has all hit terms contribute to the rank score.

When we merge posting lists, a set of query paths match paths from a document expression one at a time, each time a *hit path set* for matched query and candidate paths are examined. Define $\mathcal{P}(T)$ to be all paths extracted from OPT T, i.e., $\mathcal{P}(T) = \{p : p \in \text{leafroot\_paths}(T^n), n \in T\}$ where T^n is the entire subtree of T rooted at n with all its descendants. We model the hit path set by a bipartite graph $G(Q, D, E)$ where $Q = \{q : q \in \mathcal{P}(T_q)\}, D = \{d : d \in \mathcal{P}(T_d)\}$ are query and document path sets, and edges are ordered pairs $E = \{(q, d) : \text{tokenized}(q) = \text{tokenized}(d), q \in Q, d \in D\}$ representing a potential match between a query path to a document path. Since an edge is established only for paths with the same token sequence, we can partition the graph into disconnected smaller bipartite graphs $G_t = G(Q_t, D_t, E_t)$, each identified by tokenized query path t:

$$
\begin{aligned}
Q_t &= \{q : q \in Q, \text{tokenized}(q) = t\} \\
D_t &= \{d : d \in D, \text{tokenized}(d) = t\} \\
E_t &= \{(q, d) : (q, d) \in E, \text{tokenized}(q) = \text{tokenized}(d)\}
\end{aligned}
$$

Figure 2 shows the hit path set of the example in Figure 1, this example can be partitioned into independent subgraphs associated with tokenized paths VAR/TIMES/ADD, VAR/TIMES and VAR/ADD. Each partition is actually a complete bipartite graph (fully connected) because for any edge between Q_t and D_t, it is in edge set E_t. And for each complete bipartite graph $G(Q_t, D_t, E_t)$, we can obtain their maximum matching sizes from $\min(|Q_t|, |D_t|)$ easily.

On the other hand, to calculate score $w^*_{Q,D}$, we need to find a pair of query and document nodes at which the widest common subtree $\hat{T}^*_q, \hat{T}^*_d$ are rooted (see equation 2), so we also define the matching candidate relations filtered by nodes. Let $G^{(m,n)} = G(Q^{(m)}, D^{(n)}, E^{(m,n)})$ be the subgraph matching between query subtree rooted at m and document subtree rooted at n where

$$\begin{aligned} Q^{(m)} &= \{q : q \in Q, \text{root\_end}(q) = m\} \\ D^{(n)} &= \{d : d \in D, \text{root\_end}(d) = n\} \\ E^{(m,n)} &= \{(q,d) : (q,d) \in E, \text{root\_end}(q) = m, \text{root\_end}(d) = n\} \end{aligned}$$

Then, similarity score $w^*_{Q,D}$ can be calculated from selecting the best matched node pairs and summing their partition matches. Specifically, define *token paths* of tree T rooted at n as set $\mathfrak{T}(n) = \{t : t = \text{tokenized}(p), p \in \text{leafroot\_paths}(T^n)\}$,

$$w^*_{Q,D} = \max_{m \in T_q, n \in T_d} \nu(G^{(m,n)}) \tag{3}$$

$$= \max_{m \in T_q, n \in T_d} \sum_{t \in \mathfrak{T}(m)} \nu(G^{(m,n)}_t) \tag{4}$$

$$= \max_{m \in T_q, n \in T_d} \sum_{t \in \mathfrak{T}(m)} \min(|Q^{(m)}_t|, |D^{(n)}_t|) \tag{5}$$

where $\nu(G)$ is the maximum matching size of bipartite graph G.

Denote $w_{m,t} = |Q^{(m)}_t|$, we call $w_{m,t} \geq \min(|Q^{(m)}_t|, |D^{(n)}_t|)$ as our (pre-computed) partial score upperbound. It is analogous to text search where each posting list has a partial score upperbound, the TF-IDF score upperbound is merely their sum. In our case, the sum for partial score upperbounds is only for one node or a subtree.

In the following we propose three strategies to compute $w^*_{Q,D}$ upperbound from partial score upperbounds and assign non-requirement set.

Max reference (MaxRef) strategy In MaxScore [17, 19], each posting list has a partial score upperbound, however, our scoring function implies each posting list can be involved with multiple partial score upperbounds. One way to select the non-requirement set in our case is using an upperbound score $MaxRef_t$ (for each posting list t) which is the maximum partial score from the query nodes by which this posting list gets "referenced", and if a set of posting lists alone has a sum of MaxRef scores less or equal to θ, they can be safely put into the non-requirement set.

The rank safety can be justified, since each posting list corresponds to a unique tokenized path t, and $\text{MaxRef}_t = \max_m w_{m,t}$. Then for $m \in T_q, n \in T_d$,

$$\sum_t \min(|Q^{(m)}_t|, |D^{(n)}_t|) \leq \sum_t w_{m,t} \leq \sum_t \text{MaxRef}_t \tag{6}$$

then the selection of non-requirement set (named **Skip** set for short) such that $\sum_{t \in \textbf{Skip}} \text{MaxRef}_t \leq \theta$ follows $w^*_{Q,D} \leq \theta$ for all non-requirement set posting lists.

Greedy binary programming (GBP) strategies Inequality (6) is relaxed twice, so it spurs the motivation to get tighter upperbound value by maximizing the number of posting lists in the non-requirement set, so that more posting lists are likely to be skipped. Define partial upperbound matrix $\mathbf{W} = \{w_{i,j}\}_{|T_q|\times|\mathfrak{T}|}$ where $\mathfrak{T} = \{\mathfrak{T}(m), m \in T_q\}$ are all the token paths from query OPT ($\mathfrak{T}$ is essentially the same as tokenized $\mathcal{P}(T_q)$), and a binary variable $\boldsymbol{x}_{|\mathfrak{T}|\times 1}$ indicating which corresponding posting lists are placed in the non-requirement set. One heuristic objective is to maximize the number of posting lists in the non-requirement set (GBP-NUM):

$$\text{maximize} \qquad \mathbf{1} \cdot \boldsymbol{x} \tag{7}$$

$$s.t. \qquad \mathbf{W}\boldsymbol{x} \leq \theta \tag{8}$$

However, maximizing the number of posting lists in the non-requirement set does not necessarily cause more items to be skipped, because the posting lists can be very short. Instead, we can maximize the total length of posting lists in the non-requirement set. In this case, the vector of ones in objective function (7) is replaced with posting list length vector $\mathbf{L} = \left[L_1, L_2, \ldots L_{|\mathfrak{T}|}\right]$, where L_i is the length of posting list i. We call this strategy GBP-LEN. The two GBP strategies are rank-safe since constraints in inequality (8) implies $\sum_{t\in\mathbf{Skip}} w_{m,t} \leq \theta$.

Both strategies require solving binary programming problems, which are known to be NP-complete and thus too intensive for long queries. Instead, we greedily follow one branch of the binary programming sub-problems to obtain a feasible (but not optimal) solution in $O(|T_q||\mathfrak{T}|^2)$.

5 Implementation

Figure 3 illustrates formula query processing using a modified inverted index for dynamic pruning. For each internal node m of the query OPT, we store the number of leaves of m as $w_m = |Q^{(m)}|$. Each query node points to tokenized path entries in a dictionary, where each reference is associated with $w_{m,t} = |Q_t^{(m)}|$ identified by tokenized path t (denoted as m/w_m of t). In Figure 3, node $q1$ from the query has 6 leaves, which is also the upperbound number of path matches for $q1$, i.e, $|Q^{(1)}|$. Since $q1$ consists of 2 tokenized leaf-root paths VAR/TIMES/ADD and VAR/ADD, $q1$ is linked to two posting lists, each associated with a partial score upperbound (5 and 1).

Each posting list maps to a token path $t \in \mathfrak{T}$ with a dynamic counter for the number of query nodes referring to it (initially $|Q_t|$). Query nodes are pruned by our algorithm when its subtree width is no longer greater than the current threshold, because the corresponding subexpression cannot be in the top-K results. In this case the reference counter decreases. A posting list is removed if its reference counter is less than one.

Each posting list entry identified by an ExpID stores n and $w_{n,t} = |D_t^{(n)}|$ values of subtree token path t rooted at n (denoted as n/w_n of t). As an example, in Figure 3, the hit OPT (of ExpID 12) has 5 paths tokenized as

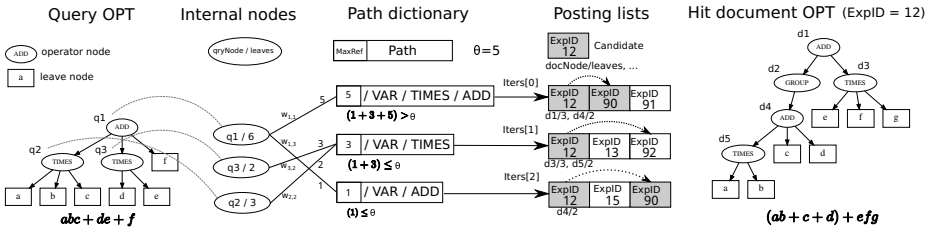

Fig. 3. Indices for formula search with dynamic pruning. For MaxRef strategy, the top posting list is the only one in the requirement set. The bottom two posting lists are advanced by skipping to next candidate ExpID.

t = VAR/TIMES/ADD, 2 rooted at $d4$ and 3 rooted at $d1$. The information $(d1/3, d4/2)$ is stored with corresponding posing list t. In our implementation, each posting list is traversed by an iterator ($iters[t]$), and its entries are read by $iters[t].read()$ from the current position accessed by iterator.

Query processing is described in Algorithm 1. RequirementSet returns selected iterators of the requirement set. Assignment according to different pruning strategies is described in Section 4. In the MaxRef strategy, we sort posting lists by descending MaxRef values, and take as many posting lists as possible into non-requirement set from the lowest MaxRef value. At merging, a *candidate* ID is assigned by the minimal ExpID of current posting list iterators in the requirement set. Requirement set iterators are advanced by one using the *next()* function, while iterators in the non-requirement set are advanced directly to the ID equal to or greater than the current candidate by the *skipTo()* function. In Figure 3 for example, the posting list corresponding to VAR/TIMES/ADD is in the requirement set under the MaxRef strategy, while the other two are not: Document expression 13 and 15 will be skipped if the next candidate is 90. For ease of testing termination, we append a special ExpID *MaxID* at the end of each posting list, which is larger than any ExpID in the collection.

At each iteration, a set of *hitNodes* is inferred containing query nodes associated with posting lists whose current ExpIDs are candidate ID. qryNodeMatch calculates matches for hit nodes according to equation 5, pruning nodes whose maximum matching size is smaller than than previously examined nodes. Given query hit node $q1$ in Figure 3, function qryNodeMatch returns

$$\max_{n \in T_d} \nu(G^{(1,n)}) = \max(\min(5,2) + \min(1,2),\ \min(5,3)) = 3$$

Then the algorithm selects the best matched query node and its matched width (i.e., *widest* in Algorithm 1) is our structural similarity $w^*_{Q,D}$.

After obtaining $w^*_{Q,D}$, we compute a metric for the similarity of symbols (e.g., to differentiate $E = mc^2$ and $y = ax^2$) and penalize larger formulas, to produce a final *overall similarity score* [23] for ranking. Because of this additional layer, we need to relax our upperbound further. According to the overall scoring

function in [23], our relaxing function u can be defined by assuming perfect symbol similarity score in overall scoring function, specifically

$$u(w) = \frac{w}{|\text{leaves}(T_q)| + w}\left[(1-\eta) + \eta\,\frac{1}{\log(1+n_d)}\right] \tag{9}$$

where in our setting, parameters $\eta = 0.05, n_d = 1$. Whenever threshold θ is updated, we will examine all the query nodes, if a query node m has an upperbound less or equal to the threshold, i.e., $u(m) \leq \theta$, then the corresponding subtree of this node is too "small" to make it into top K results. As a result, some of the posting lists (or iterators) may also be dropped due to zero reference.

Algorithm 1 Formula searching algorithm with pruning

```
1: function QRYNODEMATCH(iters, m, candidate, widest, θ)
2:     nodeMatch[ ] := 0; ℓ := | leaves(m)|          ▷ ℓ is the leftover estimated upperbound.
3:     for each m/w_m of tokenized path t rooted at m do
4:         Let i be the iterator index associated with t
5:         if iters[i].expID < candidate then
6:             iters[i].skipTo(candidate)
7:         if iters[i].expID = candidate then
8:             for each n/w_n of t from iters[i].read() do
9:                 nodeMatch[n] := nodeMatch[n] + min(w_m, w_n)
10:        ℓ := ℓ − w_m;
11:        estimate := max(nodeMatch) + ℓ                          ▷ Update estimation.
12:        if estimate ≤ widest or u(estimate) ≤ θ then
13:            return 0
14:    return max(nodeMatch)
15:
16: function FORMULASEARCH(iters, strategy)
17:    θ := 0; reqs := REQUIREMENTSET(θ, strategy)
18:    heap := data structure to hold top K results
19:    while true do
20:        candidate := minimal ID in current expIDs of reqs
21:        if candidate equals MaxID then                ▷ Search terminated, return results.
22:            return top K results
23:        Let G(Q, D, E) be the hit path set bipartite graph.
24:        widest := 0; hitNodes := {root_end(q) : (q, d) ∈ E}
25:        for m in hitNodes do          ▷ Calculate maximum match for each hit query node.
26:            if | leaves(m)| ≤ widest then
27:                continue
28:            maxMatch := QRYNODEMATCH(iters, m, candidate, widest, θ)
29:            if maxMatch > widest then widest := maxMatch          ▷ Find the widest width.
30:        if widest > 0 then
31:            score := calculate final score (including symbol similarity).          ▷ See [23].
32:            if heap is not full or score > θ then
33:                Push candidate or replace the lowest scored hit in heap.
34:                if heap is full then                          ▷ Update current threshold.
35:                    θ := minimal score in current top K results
36:                    Drop small query nodes and unreferenced iterators.
37:                    reqs := REQUIREMENTSET(θ, strategy)          ▷ Update requirement set.
38:        for iters[i] in reqs do                          ▷ Advance posting list iterators.
39:            if iters[i].expID = candidate then iters[i].next()
```

6 Evaluation

We first evaluate our system[5] on the NTCIR-12 Wikipedia Formula Browsing Task [20] (NTCIR-12 for short), which is the most current benchmark for formula-only retrieval. The dataset contains over 590,000 math expressions taken from English Wikipedia. Since work in formula retrieval is relatively new, there are only 40 queries in NTCIR-12 that can be compared with other published systems. However, these queries are well designed to cover a variety of math expressions in different complexity. There are 20 queries containing wildcards in this task (using wildcard specifier **\qvar** to match arbitrary subexpression or symbols, e.g., query "\qvar{a}^{2} + \qvar{b}^{3}" can match "$x^2 + (y + 1)^3$"). We add support for wildcards by simply treating internal nodes (representing a rooted subexpression) of formulas as additional "leaves" (by ignoring their descendants), and the wildcard specifiers in a query are treated as normal leaves to match those indexed wildcard paths.

Since the corpus of NTCIR-12 is not large enough to show the full impact of pruning, we also evaluate query run times on a corpus containing over 1 million math related documents/threads from Math StackExchange (MSE) Q&A website[6] and we run the same query set from NTCIR-12. Run times are shown for the posting list merging stage (e.g., time for parsing the query into OPT is excluded) and unless specified, posting lists are compressed and cached into memory. Each system had five independent runs, and we report results from overall distribution. The resulting uncompressed index size for NTCIR-12 and MSE corpora are around 2 GB and 16 GB in size, with 961,604 and 5,764,326 posting lists respectively. The (min, max, mean, standard deviation) for posting list lengths are (1, 262,309, 16.95, 737.84) and (1, 7,916,296, 73.74, 9736.72).

Table 1 reports run time statistics. Non-pruning (exhaustive search) baselines with K = 100 are also compared here. Almost consistently, GBP-LEN strategy achieves the best efficiency with smaller variance. This is expected since GBP-LEN models the skipping possibility better than GBP-NUM. Although GBP-NUM gives a tighter theoretic upperbound than MaxRef, it only maximizes the number of posting lists in the non-requirement set and may lead to bad performance when these posting lists are short.

There are a few times the best minimal run times are from other strategies, for those with meaningful gaps, i.e., in Wiki dataset of non-wildcard queries when K = 1000, MaxRef outperforms in standard deviation and maximum run time to a notable margin; however, it likely results from a small threshold due to large K, so that the efficiency on the small sized NTCIR dataset is less affected by pruning (small θ means less pruning potential) compared to the time complexity added from assigning to the requirement set. The latter is more dominant in GBP runs. In wildcard queries, however, many expressions can match the query thus the threshold value is expected to be larger than that in the non-wildcard case.

[5] Source code: `https://github.com/approach0/search-engine/tree/ecir2020`
[6] MSE corpus: `https://www.cs.rit.edu/~dprl/data/mse-corpus.tar.gz`

Table 1. Query merge time performance (in milliseconds) for different strategies.

	Runs		Non-wildcards					Wildcards				
	K	Strategy	μ	σ	median	min	max	μ	σ	median	min	max
Wiki Dataset	100	Baseline	540.12	569.44	360.50	7.00	2238.00	426.73	383.47	225.50	8.00	1338.00
	100	MaxRef	90.29	74.14	79.00	3.00	312.00	145.50	121.19	136.00	**7.00**	573.00
		GBP-NUM	84.90	80.44	52.50	3.00	321.00	138.82	102.55	135.00	9.00	428.00
		GBP-LEN	**67.49**	**61.40**	**45.00**	**2.00**	**218.00**	**125.27**	**97.28**	**103.50**	9.00	**404.00**
	200	MaxRef	107.71	**82.64**	102.00	**5.00**	**322.00**	160.10	121.40	149.00	9.00	583.00
		GBP-NUM	105.34	99.51	71.50	**5.00**	357.00	155.52	110.61	153.00	**8.00**	479.00
		GBP-LEN	**89.63**	83.20	**62.00**	**5.00**	330.00	**142.78**	**103.11**	**143.50**	9.00	**446.00**
	1000	MaxRef	154.51	**93.75**	157.50	**6.00**	**361.00**	211.86	140.01	186.00	10.00	662.00
		GBP-NUM	159.80	143.70	120.50	**6.00**	626.00	208.91	136.42	178.50	10.00	591.00
		GBP-LEN	**144.25**	126.95	**105.00**	**6.00**	622.00	**195.70**	**122.25**	**176.00**	**9.00**	**536.00**
MSE Dataset	100	Baseline	15134.10	15186.78	11161.00	157.00	55499.00	13450.57	12554.19	7075.50	304.00	47513.00
	100	MaxRef	1083.23	1274.23	745.50	28.00	5922.00	3188.66	2458.91	2925.00	85.00	10412.00
		GBP-NUM	1202.24	1240.21	815.00	37.00	4987.00	2943.79	2025.96	2987.00	**84.00**	8775.00
		GBP-LEN	**562.83**	**635.26**	**382.50**	**24.00**	**2313.00**	**2257.95**	**1491.59**	**2346.50**	86.00	**4494.00**
	200	MaxRef	1261.21	1368.93	1012.50	30.00	6439.00	3416.77	2753.09	3032.50	160.00	12412.00
		GBP-NUM	1378.19	1398.08	998.50	39.00	5863.00	3174.93	2283.05	3125.00	**159.00**	10099.00
		GBP-LEN	**697.32**	**739.11**	**478.00**	**27.00**	**2925.00**	**2504.90**	**1683.16**	**2382.50**	**159.00**	**6049.00**
	1000	MaxRef	2030.05	1746.17	1796.50	53.00	7816.00	4123.26	3510.01	3473.00	287.00	16981.00
		GBP-NUM	1952.52	1746.05	1530.50	60.00	7197.00	3786.89	2744.99	3493.50	**281.00**	11323.00
		GBP-LEN	**1217.16**	**1083.53**	**764.50**	**47.00**	**3756.00**	**3304.69**	**2403.09**	**2812.00**	285.00	**9895.00**

System	Non-Wildcard		Wildcard		All queries	
	Full	Partial	Full	Partial	Full	Partial
MCAT	.5678	.5698	**.4725**	.5015	.5202	.5356
Tangent-S	.6361	.5872	.4699	**.5368**	**.5530**	**.5620**
base-best	**.6726**	**.5950**	-	-	-	-
base-opd-only	.6586	.5153	-	-	-	-
Ours (pruning)	.6586	.5173	.3678	.3973	.5132	.4573
Ours (exhaustive)	.6586	.5173	.3678	.3973	.5132	.4573

Fig. 4. Bpref [4] scores. Bpref chosen because we did not participate in NTCIR-12 and did not contribute to the pooling.

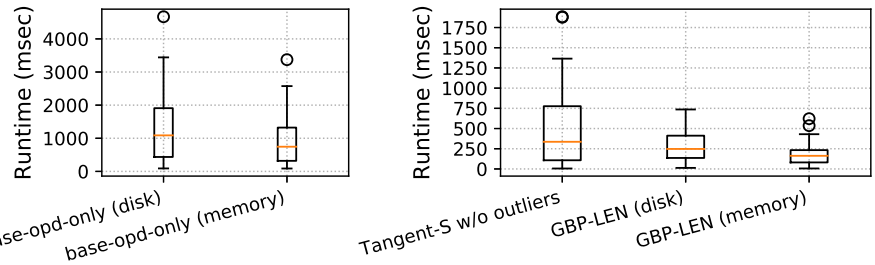

Fig. 5. Average run times on the same machine (Environment: Intel Core i5 @ 3.60GHz per core, 16 GB memory and SSD drive) for NTCIR-12 Wiki Formula Browsing Task.

Secondly, we have compared our system effectiveness (Figure 4) and efficiency (Figure 5) with Tangent-S [5], MCAT [11] and our baseline system without pruning [23], which are all structure-based formula search engines that have obtained the best published Bpref scores on NTCIR-12 dataset. In addition, ICST system [7] also obtains effective results for math and text mixed task, but they do training on previous Wiki dataset and their system is currently not available.

All systems are evaluated in a single thread for top-1000 results. We use our best performance strategy, i.e., GBP-LEN, having an on-disk version with posting lists uncompressed and always read from disk, and an in-memory version with compression. For the baseline system, only 20 non-wildcard queries are reported because it does not support wildcards. We compare the baseline best performed run (base-best) which uses costly multiple tree matching as well as its specialized version (base-opd-only) which considers only the largest matched tree width (see equation 2). Tangent-S has a few outliers as a result of its costly alignment algorithm to rerank structure and find the Maximum Subtree Similarity [22], its non-linear complexity makes it expensive for some long queries (especially in wildcard case). And MCAT reportedly has a median query execution time around 25 seconds, using a server machine and multi-threading [11]. So we remove Tangent-S outliers and MCAT from runtime boxplot. For space, we only include the faster base-opd-only baseline in Figure 5.

We outperform Tangent-S in efficiency even if we exclude their outlier queries, with higher Bpref in non-wildcard fully relevant results. Our efficiency is also better than the baseline system, even if the latter only considers less complex non-wildcard queries. However, our overall effectiveness is skewed by bad performance of wildcard queries because a much more expensive phase is introduced to boost accuracy by other systems to handle inherently difficult "structrual wildcards."

Our pruning strategies are rank-safe (pruning and exhaustive version shows the same Bpref scores) but there is a minor Bpref difference between ours and baseline (base-opd-only) due to parser changes we have applied to support wildcards (e.g., handle single left brace array as seen in a wildcard query) and they happen to slightly improve accuracy in partially relevant cases.

7 Conclusion

We have presented rank-safe dynamic pruning strategies that produce an upper-bound estimation of structural similarity in order to speedup formula search using subtree matching. Our dynamic pruning strategies and specialized inverted index are different from traditional linear text search pruning methods and they further associate query structure representation with posting lists. Our results show we can obtain substantial improvement in efficiency over the baseline model, while still generating highly relevant non-wildcard search results. Our approach can process a diverse set of structural queries in real time.

References

1. Anh, V.N., Moffat, A.: Pruned query evaluation using pre-computed impacts. In: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval. pp. 372–379. ACM (2006)
2. Antonio Mallia, M.S., Suel, T.: An experimental study of index compression and daat query processing methods. In: European Conference on Information Retrieval (ECIR 2019). Springer (2019)
3. Broder, A.Z., Carmel, D., Herscovici, M., Soffer, A., Zien, J.: Efficient query evaluation using a two-level retrieval process. In: Proceedings of the twelfth international conference on Information and knowledge management. pp. 426–434. ACM (2003)
4. Buckley, C., Voorhees, E.M.: Retrieval evaluation with incomplete information. In: Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval. pp. 25–32. ACM (2004)
5. Davila, K., Zanibbi, R.: Layout and semantics: Combining representations for mathematical formula search. In: Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval. pp. 1165–1168. ACM (2017)
6. Ding, S., Suel, T.: Faster top-k document retrieval using block-max indexes. In: Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval. pp. 993–1002. ACM (2011)
7. Gao, L., Yuan, K., Wang, Y., Jiang, Z., Tang, Z.: The math retrieval system of ICST for NTCIR-12 MathIR task. In: NTCIR (2016)
8. Jonassen, S., Bratsberg, S.E.: Efficient compressed inverted index skipping for disjunctive text-queries. In: European Conference on Information Retrieval. pp. 530–542. Springer (2011)
9. Kamali, S., Tompa, F.: Structural similarity search for mathematics retrieval. pp. 246–262 (07 2013)
10. Kenny Davila, Ritvik Joshi, S.S.V.G., Zanibbi, R.: Visual search using line-of-sight graphs: Application to math formula images. In: European Conference on Information Retrieval (ECIR 2019). Springer (2019)
11. Kristianto, G.Y., Topic, G., Aizawa, A.: Mcat math retrieval system for ntcir-12 mathir task. In: NTCIR (2016)
12. Lin, X., Gao, L., Hu, X., Tang, Z., Xiao, Y., Liu, X.: A mathematics retrieval system for formulae in layout presentations. In: Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval. SIGIR '14, ACM, New York, NY, USA (2014)
13. Macdonald, C., Ounis, I., Tonellotto, N.: Upper-bound approximations for dynamic pruning. ACM Transactions on Information Systems (TOIS) **29**(4), 17 (2011)
14. Miller, B.R., Youssef, A.: Technical Aspects of the Digital Library of Mathematical Functions. Annals of Mathematics and Artificial Intelligence **38**(1-3), 121–136 (2003), `https://link.springer.com/article/10.1023/A:1022967814992`
15. Shan, D., Ding, S., He, J., Yan, H., Li, X.: Optimized top-k processing with global page scores on block-max indexes. In: Proceedings of the Fifth ACM International Conference on Web Search and Data Mining. pp. 423–432. WSDM '12, ACM, New York, NY, USA (2012)
16. Sojka, P., Líška, M.: Indexing and searching mathematics in digital libraries. In: International Conference on Intelligent Computer Mathematics. pp. 228–243. Springer (2011)

17. Strohman, T., Turtle, H., Croft, W.B.: Optimization strategies for complex queries. In: Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval. pp. 219–225. ACM (2005)
18. Tonellotto, N., Macdonald, C., Ounis, I., et al.: Efficient query processing for scalable web search. Foundations and Trends® in Information Retrieval **12**(4-5), 319–500 (2018)
19. Turtle, H., Flood, J.: Query evaluation: strategies and optimizations. Information Processing & Management **31**(6), 831–850 (1995)
20. Zanibbi, R., Aizawa, A., Kohlhase, M., Ounis, I., Topic, G., Davila, K.: NTCIR-12 MathIR task overview. In: NTCIR (2016)
21. Zanibbi, R., Blostein, D.: Recognition and retrieval of mathematical expressions. Int. J. Doc. Anal. Recognit. **15**(4), 331–357 (Dec 2012)
22. Zanibbi, R., Davila, K., Kane, A., Tompa, F.W.: Multi-stage math formula search: Using appearance-based similarity metrics at scale. In: Proceedings of the 39th International ACM SIGIR Conference on Research & Development in Information Retrieval. SIGIR '16, ACM, NY, USA (2016)
23. Zhong, W., Zanibbi, R.: Structural similarity search for formulas using leaf-root paths in operator subtrees. In: European Conference on Information Retrieval (ECIR 2019). Springer (2019)