A Structural Query System for Han Characters

Matthew Skala
Theoretical Computer Science Group, IT University of Copenhagen,
Rued Langgaards Vej, 2300 København S, Denmark.
mskala@ansuz.sooke.bc.ca

Abstract

The IDSgrep structural query system for Han character dictionaries is presented. This dictionary search system represents the spatial structure of Han characters using Extended Ideographic Description Sequences (EIDSes), a data model and syntax based on the Unicode IDS concept. It includes a query language for EIDS databases, with a freely available implementation and format translation from popular third-party IDS and XML character databases. The system is designed to suit the needs of font developers and foreign language learners. The search algorithm includes a bit vector index inspired by Bloom filters to support faster query operations. Experimental results are presented, evaluating the effect of the indexing on query performance.

Keywords

Han script, character description, font, radical, grep, tree matching

1. Introduction

Han script carries semantic information at a level finer than single characters. If we write the Japanese word *yuki* (``snow'') phonetically as ゆき, it has no obvious morphology. If we write it with the Han character 雪, then we can compare it to 雲 (*kumo*, ``cloud''); 電 (*inazuma*, ``lightning''); and 霊 (*rei*, ``spirit''). These characters have something in common: they each consist of 雨 above something. Computational study of what they have in common requires a computational way of describing it.

In this paper we describe the IDSgrep structural query system. IDSgrep answers questions about Han characters and their spatial structures by searching for matching patterns in dictionary databases, using a query language designed for the purpose. The data model covers the entire structure of characters, not only their general layout, and the query language permits description of arbitrarily complex Boolean criteria. These novel features allow IDSgrep to answer queries that previous systems could not.

Systematic efforts to describe and analyze the structure of Han characters date back as far as the Second Century (Creamer 1989). Creamer describes *Shuowen Jiezi*, a very early Chinese dictionary, which divided its character set into 540 headings, most corresponding to semantic components (radicals) that might appear in the characters. The 雨 component in our example of 雪 is one such. Details like the list of radicals, and the process of choosing a single component as the index radical for each character, have varied over time and with the differ-

ing purposes for which dictionaries have been compiled. But that general scheme remained the standard for hardcopy dictionaries of Han characters until the Twentieth Century. A user looking up an unknown character would start by identifying the radical. They would then search the section corresponding to that radical, which might be further organized by total number of strokes in the character.

Radicals in such dictionaries usually correlate with semantics, but serve primarily as index keys. The constraints of the dictionaries require choosing exactly one radical from the list as the index key for each character. Sub-character semantics requires more than just answering the question ``Are these two characters indexed by the same radical?''; the other components, and their spatial relationships, are of interest. Dictionary lookups also benefit from more detailed descriptions of spatial structure, as the desire for radicals to reflect semantics forces compromise of their use as purely visual descriptions.

The most recent hardcopy dictionaries, especially those aimed at foreign language learners, use more detailed and visual structural descriptions. For instance, the SKIP method (Halpern 2013) uses an easily-memorized numerical description defined by the visual appearance, not the semantics, of the character. A user can identify the dictionary head for 明 as ``1--4--4'', meaning ``divided into left and right parts, with four strokes on the left and four on the right'', and find the character in the dictionary without needing to know which side means ``moon'', nor which component is the radical.

WWWJDIC (Breen 2014b) is one of the best-known computerized dictionaries. As a Web-based resource it is constantly updated, and it offers queries by traditional radical and stroke count; by SKIP code; and several other classification schemes. Its interactive searches are especially convenient for users who do not know the Han character set well. In multi-radical mode, the user chooses one component at a time, from a list that is roughly the few hundred radicals of traditional dictionaries. Any components that occur in the character may be used; there is no requirement to identify the single official radical of the character. A dynamically-updated list shows all characters in the database that contain all (by simple Boolean AND) of the chosen components. Handwriting recognition mode offers a similar interactive search process, updating the list as the user writes the first few strokes of the character. It depends on the user having enough knowledge of the writing system to guess the stroke order. These searches allow for more flexibility than traditional radical search, but still cannot answer queries with precisely specified geometric constraints.

IDSgrep, the subject of the current work, originated as an internal development tool for the Tsukurimashou Project's Japanese-language parametric font family (Skala 2014b). Tsukurimashou represents characters as programming language subroutines, with the structure of the code echoing the visual, not semantic, organization of the characters. For instance, the code for 明 invokes subroutines for 日 and 月 and a subroutine that abstracts the operation of placing components in a left-right configuration. When writing new character definitions, the developer must be able to find other characters with similar structures that could share code.

Queries like ``which other characters, if any, have the same right-hand side as this one?'' are not easy to answer with other dictionaries. Native experts, if available at all, can also only help to a point. Negative search results are of interest, and a human expert can at best say ``I can't think of a character fitting this description.'' To be sure *there exists no* character fitting a given description we need a precise semantics of character descriptions, a database we can trust to contain all characters of interest, and a tool for querying the database. IDSgrep is the query tool; it defines at least a syntax for the descriptions; and it can make use of existing databases that are complete enough to be useful.

Much current work in computational linguistics and natural language processing can benefit from this new resource. For instance, Chu et al. (2013) enhance Chinese-Japanese machine translation by making use of the similarities between the Simplified Chinese, Traditional Chinese, and Japanese character sets. As they describe, visual similarity is one of the clues that indicates possible semantic equivalency. However, all their sources, and their completed mapping tables, stop at the level of whole characters. The linguistic processes leading early characters to develop similarly or differently in different languages actually take place at the level of character components. These authors demonstrate an improvement in machine translation by exploiting knowledge of similar characters. It is a research question worth asking, whether a machine translation system informed directly by components could show even greater improvement. IDSgrep can support that research.

Hao and Zhu (2013) study conversion between Simplified and Traditional Chinese text. As in the Chinese-Japanese translation work, they are limited by the nature of their data sources to consideration of whole characters. Although ``this component usually simplifies to that component, in any character'' is a typical description of the linguistic processes underlying simplification, it cannot be conveniently represented as a single rule at the whole-character level. IDSgrep's data model provides a way to describe such rules, and its search capability supports their discovery and implementation.

Liu et al. (2011) study mistakes made by students in written Chinese, in particular the substitution of incorrect characters that may be visually or phonologically similar to the correct characters. As they write, there are ``no obvious ways to determine algorithmically whether two Chinese characters are visually similar yet.'' They survey several possibilities, and choose Cangjie codes. They are forced to extend the codes ``with the help of computer programs and subjective judgments'' to cover their full problem domain, a process they describe as ``labor-intensive''. Shared structures in Cangjie must be detected by plain substring search, and that does not always correctly detect all shared structures. IDSgrep offers a better compromise. With it, work like that of Liu et al. (2011) can focus primarily on determining which shared structures are or are not relevant to student mistakes, rather than spending intensive labor on imperfectly describing the shared structures in the first place.

The main contributions of the present work are:

- a data model and query language for the spatial structure of Han characters;
- algorithmic techniques for efficient implementation of the query language; and
- experimental evaluation of a practical implementation.

The software is freely available from the Tsukurimashou Project's Web site on Sourceforge Japan (Skala 2014c). In its original form, studied here, IDSgrep is a command-line utility. Third-party users have adapted it to other environments, such as an experimental Web interface (Fasih 2015).

1.1. Character description languages

Computer typesetting projects for Han-script languages have long used descriptions of the character glyphs in terms of smaller components, with varying degrees of complexity and formal specification in how those components may be combined. Some work in this area has focused on the METAFONT system (Knuth 1986), in which glyphs to be typeset are described using a fully powered computer programming language and components and combining operations can be invoked as subroutines. Many authors have worked on METAFONT-related Han script projects over the course of more than three decades, with the Tsukurimashou Project that gave birth to IDSgrep as one of the most recent contributions (Mei 1980; Hobby and

Guoan 1984; Hosek 1989; Yiu and Wong 2003; Laguna 2005; Skala 2014b). The Wadalab font project (Tanaka et al. 1995) implemented similar concepts using LISP instead of METAFONT, and was one of the most successful projects of its kind; fonts it generated are in wide use in the free software community to this day. Any such project implicitly extends the programming language used into a language for describing Han characters, but most do not treat the descriptions as separate entities from the software code. HanGlyph (Yiu and Wong 2003) is one exception: it defines a formal syntax for a description language that is translated by separate and character-independent software.

Several projects use XML rather than a programming language to describe characters, and these projects often emphasize dictionary and database applications instead of primarily font creation. Font creation may nonetheless be included as one intended application of the data. Such projects include Structural Character Modeling Language (SCML) (Peebles 2007), Character Description Language (CDL) (Wu and Zheng 2009), GlyphWiki (Kamichi 2014), and KanjiVG (Apel 2014). Here the focus is often on providing high-quality data in a convenient form for application development, with such details as user interface and query language left to the application developers to determine. Although IDSgrep does not query XML directly, it is one such query application. The possibility of using the popular XML databases, and KanjiVG in particular, was one factor motivating its design.

The Prolog-based Han character description and query system described by Dürst (1996) may be the closest previous work to IDSgrep's Prolog-inspired data model. It was a proof of concept designed to illustrate the general power of Prolog for internationalization, not designed and not easily able to be scaled to complete dictionaries.

1.2. Tree searching

The general problem of searching for a pattern in a large input is one of the most thoroughly studied in computer science. Searching utilities like GNU grep (Free Software Foundation 2014) are widely used. At least among expert users, grep-like regular expression search is regarded as the standard for flexible text searching and is expected as a feature of text editors, database software, and programming languages or libraries. Considered as a general-purpose searching utility, IDSgrep does something much like regular expression matching on tree structures. Regular expression matching generalized to trees, and other kinds of tree pattern matching, have been studied both as abstract problems (Polách 2011) and with specific application to searching parse trees in computational linguistics applications (Lai and Bird 2010; Choi 2011). The Tregex utility (Levy and Andrew 2006) is a popular implementation in the computational linguistics domain, used for comparison in the experimental section of the present work.

Although the system can process other kinds of queries too, many important IDSgrep queries take the form of an example tree with some parts left as match-anything wildcards. The matching operation on such a query is equivalent to the *unification* operation on terms in logic programming languages like Prolog (Clocksin and Mellish 1987), and algorithmic techniques applicable to unification are of interest for IDSgrep and IDSgrep-like tree matching.

Unification can also be defined in a lattice of types, and one well-known technique for unification in type lattices represents the types as bit vectors with bitwise AND and zero-testing to represent the unification operation (Aït-Kaci et al. 1989). The bit vector approach to type unification has been extended to generalize the zero value, which permits the use of shorter vectors and thus faster processing (Skala et al. 2010). Permitting approximate results via the Bloom filter concept (Bloom 1970; Skala and Penn 2011) allows further speed

improvement. The present work applies similar ideas to speed improvement for tree matching. The work of Kaneta et al. (2012) on unordered pseudo-tree matching with bit vectors is also of interest; it considers a very different tree-matching problem, but it uses some similar bit-vector techniques, and has a strong theoretical analysis.

2. The EIDS data model and syntax

Unicode defines a simple grammar for describing Han characters as strings called *Ideographic Description Sequences* (IDSes) (Unicode Consortium 2011). An IDS is one of the following:

- a single character chosen from a set that includes the Unicode-encoded Han characters, strokes for building up Han characters, and radicals or components that may occur in Han characters;
- one of the prefix binary operators ⿰⿱⿴⿵⿶⿷⿸⿹⿺⿻ followed by two more IDSes, defined recursively; or
- one of the prefix ternary operators ⿲⿳ followed by three IDSes, defined recursively.

The binary and ternary operators are special characters defined for this purpose, with code points in the range U+2FF0 to U+2FFB. Example Unicode IDSes include ``⿰日月" for ``明"; ``⿰言⿱五口" for ``語"; and ``⿴口⿱⿰木山丁" for an unencoded nonsense character. These are shown in Figure 1. Note that Japanese character forms are used in the examples in this paper; for most characters these resemble the Traditional Chinese forms, but there are also many characters for which the Japanese form is closer to the Simplified Chinese form.

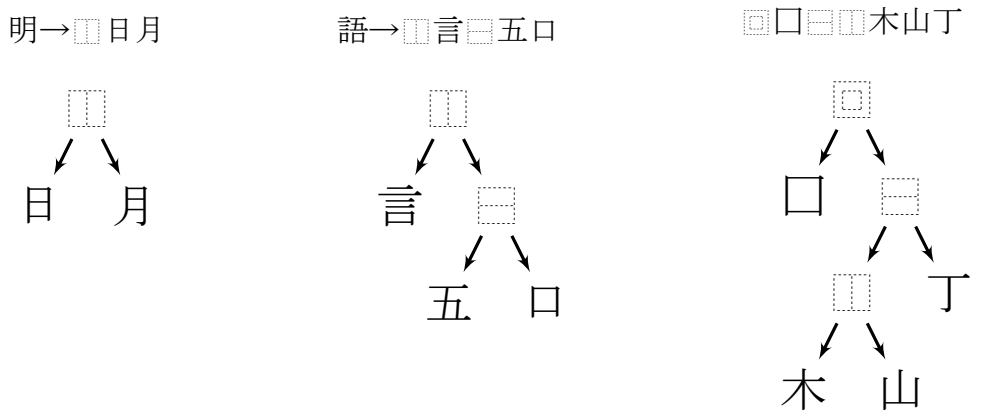

Figure 1: Sample Unicode IDSes and their associated trees.

IDSgrep describes characters using *Extended Ideographic Description Sequences* (EIDSes), which are strings of Unicode characters expressing abstract data structures called *EIDS trees* (Skala 2014a). An EIDS tree is a tree data structure with the following properties.

- Each node has a *functor*, which is a nonempty string of Unicode characters.
- Each node may optionally have a *head*, which if present is a nonempty string of Unicode characters.
- Each node has a sequence of between zero and three children, which are EIDS trees defined recursively.

The number of children of a node is called its *arity*. Functors, and heads where present, usually consist of single characters, but that is not a requirement.

The most explicit EIDS character string for a given EIDS tree consists of the head of the root, if any; the functor of the root; and then the EIDSes for all the root's children, recursively. Heads and functors are marked as such, and the arity of the node is indicated, by enclosing the strings in specific delimiting characters. ASCII angle brackets <> indicate a head. Parentheses (), dots . . , square brackets [], and curly braces {} are used for functors with arity zero, one, two, and three respectively. For instance, an EIDS in this explicit syntax might be written ``[pq].x.<head of a>(a)(b)". That starts with the functor of the root, which is ``pq", enclosed in square brackets to show that the root has two children. The first child is ``.x.<head of a>(a)". Dots around the functor ``x" indicate that this child is a unary node, with one child of its own. That child is ``<head of a>(a)", a nullary (leaf) node with a head attached. On the other side of the root, the second child is the headless leaf ``(b)". The associated EIDS tree is shown in Figure 2.

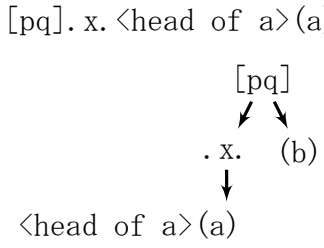

Figure 2: A sample EIDS and its EIDS tree.

The syntax includes several additional features designed both to make it easier to use and to allow valid Unicode IDSes to be valid IDSgrep EIDSes. The fully bracketed form would rarely be used in practice. First, all the Unicode IDS operator characters such as ⿲ and ⿰, and some special characters used in IDSgrep pattern matching, are considered to have implicit brackets of the appropriate type when they occur where an opening bracket would otherwise appear. These are called *sugary implicit brackets* (from the term ``syntactic sugar" (Landin 1964)). For instance, ``⿰(a)(b)" expresses the same EIDS tree as ``[⿰](a)(b)".

If a character does not have a special function in the syntax, then by default the character is considered to have implicit <> head brackets and also be followed by ``(;)", a *syrupy implicit semicolon*. Han characters and their components fall into this category. Thus a single character like ``語" is a valid EIDS as well as a valid Unicode IDS, and parsing it produces the same EIDS tree as the explicitly bracketed ``<語>(;)".

A few other syntax rules exist, covering issues like backslash escapes and ASCII abbreviations used to make typing easier. These points are beyond the scope of the current discussion, but described in the IDSgrep documentation (Skala 2014a). One remaining rule significant to the current work comes about because neither a head nor a functor may be empty. If the closing bracket that would otherwise end a bracketed string occurs immediately after the opening bracket, then it does *not* end the string but becomes the first character of the string. The important, and motivating, consequence of this rule is that ``..." is valid syntax for the functor of a unary node consisting of a single ASCII period. That is the ``match anywhere" query operator.

A Unicode IDS maps naturally to the EIDS tree formed by parsing it as an EIDS. The IDS operators like ⿰ and ⿲ become the functors of binary and ternary nodes in the tree under the sugary-bracket rule. The Han characters, strokes, and components become the heads of

leaf nodes, with semicolons as their functors, under the syrupy-semicolon rule. However, it is also possible to insert heads at other levels of the tree just by inserting each head in ASCII angle brackets at the appropriate point in the Unicode IDS. For instance, a dictionary entry for the character 語 might look like ``⟨語⟩⿰言⟨吾⟩⿱五口'': the internal nodes are marked with the characters that represent the subtrees at those locations even though they are also broken down further. If a subtree happened not to be an encoded character in itself, it could be left anonymous with no head. A search for this dictionary entry could use the complete low-level decomposition, or match a subtree or the entire entry by matching the appropriate head. The EIDS tree is shown in Figure 3.

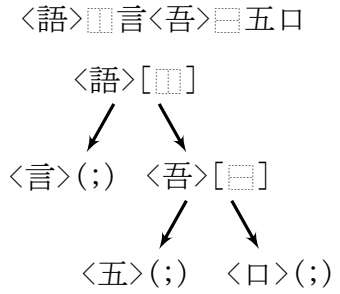

Figure 3: The EIDS tree for the dictionary entry ⟨語⟩⿰言⟨吾⟩⿱五口. Note the implicit brackets and semicolons.

3. The IDSgrep query language

To offer search functions on a database of character descriptions, we need a way for users to describe the search criteria. IDSgrep implements a *query language* capable of expressing a wide variety of criteria on the structures of characters. The IDSgrep query language was inspired by Unix regular expressions; it is intended to provide a similar combination of simplicity and expressive power. But because of the context-free non-regular nature of EIDS syntax, regular expressions as such would not be appropriate. IDSgrep's query language is based instead on matching trees against templates that may contain wildcards and other special matching operators.

3.1. Query language definition

We will define a function $\mathit{match}(N, H)$, where N and H are EIDS trees called the *needle* and *haystack* respectively. This function returns a truth value, T if the trees are considered to match and F otherwise, according to the following rules.

- If N and H both have heads, then $\mathit{match}(N, H) = \mathsf{T}$ if and only if those heads are identical. No other rules are applied.
- If N and H do not both have heads, but the functor and arity of N are in the set of matching operators $\{(?), \ldots, .*., .!., [\&], [|], .=., .@., ./., .\#.\}$ then $\mathit{match}(N, H)$ is determined by rules specific to the operator, as described below. Note that arities are indicated by the brackets around the operators, and ... is one of the operators, not an indication of omitted items.
- Otherwise, $\mathit{match}(N, H) = \mathsf{T}$ if and only if N and H have identical functors and arities and $\mathit{match}(N_i, H_i) = \mathsf{T}$ recursively for each pair (N_i, H_i) of corresponding children

of N and H.

The nullary question mark $(?)$ is a match-everything wildcard: $\mathit{match}((?), H) = \mathsf{T}$ for all H. Three dots (syntax for a unary functor containing a single dot) match anywhere: $\mathit{match}(\ldots N, H) = \mathsf{T}$ if some subtree of H (possibly all of H) is matched by N. The asterisk allows reordering of children at the top level: $\mathit{match}(.*. N, H) = \mathsf{T}$ if and only if there is some permutation of the children of N that would match H.

The basic Boolean operations of NOT, AND, and OR are available through .!., [&], and [|] respectively. We have $\mathit{match}(.!. N, H) = \mathsf{T}$ if and only if $\mathit{match}(N, H) = \mathsf{F}$; $\mathit{match}([\&]MN, H) = \mathsf{T}$ if and only if $\mathit{match}(M, H) = \mathsf{T}$ and $\mathit{match}(N, H) = \mathsf{T}$; and $\mathit{match}([|]MN, H) = \mathsf{T}$ if and only if $\mathit{match}(M, H) = \mathsf{T}$ or $\mathit{match}(N, H) = \mathsf{T}$.

The equals sign performs literal matching of functors that would otherwise be interpreted as special. If N and H both have heads, then $\mathit{match}(.=. N, H) = \mathsf{T}$ if and only if the heads are identical. Otherwise, $\mathit{match}(.=. N, H) = \mathsf{T}$ if and only if N and H have identical functors, identical arities, and $\mathit{match}(N_i, H_i) = \mathsf{T}$ is true for their corresponding children. Those are the same rules as for tree matching without .=. , except that any special matching behavior of the functor of N is ignored. This operator is seldom needed for Han character databases, but included to allow the use of IDSgrep on other kinds of data.

The at-sign does rearranged matching for operations governed by an associative law. Consider the case of three character components side by side, as in the character 側. This structure could be represented as ⿰⿰亻貝刂, ⿰亻⿰貝刂, or ⿲亻貝刂, as shown in Figure 4. A query written to match one of these could miss the others.

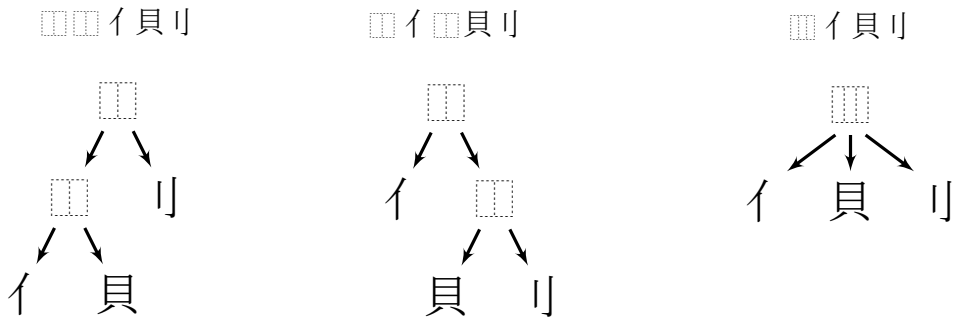

Figure 4: Multiple representations of side-by-side components.

Kawabata (2012) proposes normalizing all IDSes into a canonical form (which in this case would be ⿰亻⿰貝刂) to make matching easier. That is fine if we have complete control of the input data, and we are sure we will always want to match all alternatives in this kind of case. But in some applications, such as describing the structure of code in the Tsukurimashou Project, there may be meaningful differences between the tree structures of Figure 4, such that we might sometimes want a query to match one and *not* the others. If the EIDS trees are used to represent semantic information, then the different trees may reflect different analyses of the semantic value of 側. Thus, rather than imposing a normalization requirement on the database, IDSgrep implements a special operator for associative matching. If associative matching is desired, the user can invoke this operator.

Evaluation of $\mathit{match}(.@. N, H)$ starts from the roots of N and H and descends recursively through all children whose functors and arities match those of the root. The remaining subtrees

below the matching nodes are treated as children of notional nodes with unlimited arity; and then those nodes are compared literally as with the `.=.` operator (functor and arity must match, and all corresponding children). Thus `@⿰⿰AB⿰CD` will match all five cases of `A`, `B`, `C`, and `D` combined in that order by three ⿰ nodes. This matching operator does not convert ternary to binary IDS operators, as the normalization approach would.

The remaining special matching operators provide escape from IDSgrep to other pattern matching systems. Slash invokes the PCRE library to perform Perl-compatible regular expression matching (Hazel 2014). This operator is included to support use of IDSgrep on expanded dictionaries that combine structural information with pronunciation and word definitions. Finally, the hash operator is for invoking user-defined matching predicates. In IDSgrep version 0.5.1 the user-defined predicates test characters against the coverage of font files; they are not described further here.

3.2. Examples

In typical use, the IDSgrep utility's main input is a dictionary containing the decompositions of characters, with each tree having a head at the root level containing the character being decomposed, and then some decomposition below that. For instance, the IDSgrep dictionary derived from KanjiVG includes the entry 〈結〉⿰糸〈吉〉⿱士口. The utility accepts a query from the user and displays all dictionary entries that match the query.

The simplest kind of query is a single character like 結. Under the parsing rules, that is translated to a tree consisting of a single nullary (leaf) node with 結 as its head and semicolon as its functor, as if the query had been the explicitly-bracketed `〈結〉(;)`. Since all the dictionary entries have heads, matching proceeds by simply comparing heads for identity; the search will return 〈結〉⿰糸〈吉〉⿱士口 and any other entries that have heads identical to 結. Used this way, IDSgrep performs a simple lookup function. Figure 5 shows the head-to-head matching in this example. Because both the needle and the haystack have heads, the match result is determined solely by testing for the heads to be equal.

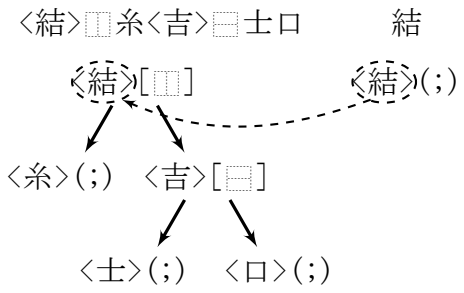

Figure 5: Basic head-to-head matching.

A more complicated query might specify the complete structure of the character. For instance, ⿰糸⿱士口 (note no heads on the non-leaf nodes) will match 〈結〉⿰糸〈吉〉⿱士口 by recursive matching of subtrees. In this way IDSgrep might serve to augment an input method: a user might know the pronunciation or other information needed to type 糸, 士, and 口, without knowing how to type 結. Figure 6 shows this query. At the root, because the needle and haystack do not both have heads, matching tests for the functors ⿰ and arities to be equal, then proceeds recursively through the subtrees. The left subtrees are identical and

match by the head-to-head rule, and the right subtrees (<吉>㠯士口 and 㠯士口) match by recursively examining their structure.

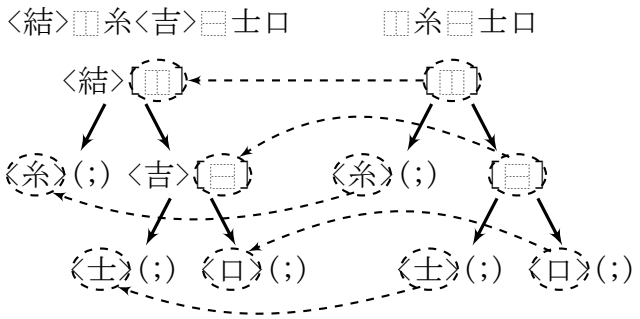

Figure 6: Matching subtrees recursively.

In those examples a single dictionary entry was more or less completely specified by the query. But the greater flexibility of IDSgrep's query language appears in cases where the query specifies only partial information, and may match many entries. For instance, the query . . . 士 matches all characters that contain 士 anywhere, with 70 hits in the KanjiVG database. The query &. . . 士. . . 口 matches all characters containing both 士 and 口, with 25 hits in KanjiVG. These searches mimic the multi-radical search of many computerized character dictionaries.

IDSgrep can go a step further yet by capturing spatial information in the query. For instance, ⿰?. . . 士 matches characters that contain 士 as or within the right side (not just anywhere; 31 hits in KanjiVG), using the wildcard operator on the left; this query is illustrated in Figure 7. At the top level, the needle root [⿰] matches the root of the haystack (by functor and arity) and triggers recursive examination of the subtrees. The left child of the needle is (?), which matches unconditionally. On the right, the match-anywhere operator . . . allows its single child to match anywhere within the haystack's right subtree <吉>㠯士口. The child of the match-anywhere operator is 士, which matches within <吉>㠯士口 at the second level. Note that the match-anywhere operator is restricted by its application inside the right child of [⿰]: this query would not match the dictionary entry <頡>⿰<吉>㠯士口頁, which contains 士 but does not contain it within the right child of the root. Query restrictions of this kind are not available in systems that treat the character as an undifferentiated bag of components.

The query ⿰?㠯士口 matches characters that contain 㠯士口 as the right side (6 hits in KanjiVG). That query might come from a language learner who is unsure about 糸 but recognizes and can type the other components in 結. A handwriting recognition query would be difficult here because 糸 comes first in the stroke order, requiring the user to write it correctly before starting to specify the known components.

Boolean operations allow for fine-tuning of query results. For instance, the available databases often analyze components like 貝 as composites having 八 at the bottom. Such a description of the visual layout may not always give useful insight into the semantic value of the larger component. A user interested in 八 but not when it occurs inside components like 貝 might use the query &. . . 八!. . . 㠯?八. To match that query, each dictionary entry would have to match . . . 八 (therefore containing 八 somewhere), but *not* match . . . 㠯?八. Anything containing 八 as the right (lower) child of [㠯], for instance because it contains 貝, would be

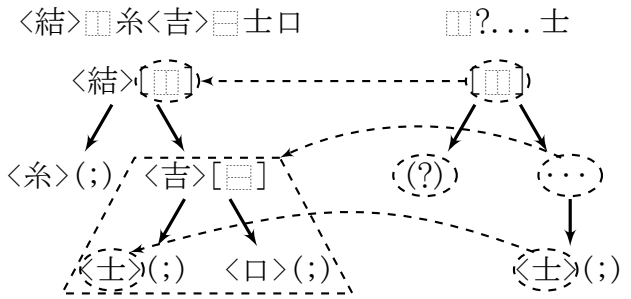

Figure 7: Matching with wildcard operators.

excluded, as in the negative example of Figure 8. In that example, the overall match fails because !... ⿱?八 fails, which in turn is because ... ⿱?八 succeeds. The successful match in the left subtree of [&] does not affect the result.

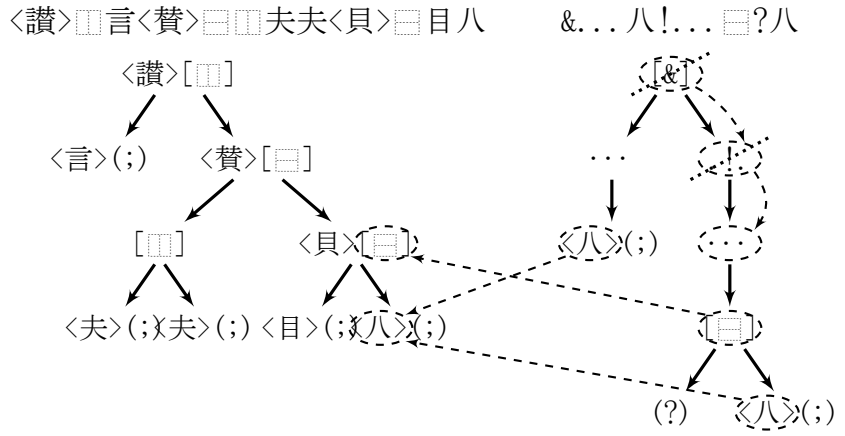

Figure 8: Use of Boolean operators (failed match).

4. Implementing the query language

A straightforward implementation of the IDSgrep command-line utility would parse every EIDS tree in its input and then match the trees against the user-specified matching pattern by recursive descent. Figure 9 shows the organization of such a system. The current version of IDSgrep is somewhat more complicated, but this basic design underlies the enhancements to be described later.

Dictionaries come from third-party sources in a variety of formats. For instance, KanjiVG (Apel 2014) is a collection of XML files describing the control-point coordinates for drawing the strokes of each character. As illustrated at the top left of Figure 9, this data must be converted to IDSgrep's EIDS format for use with the search utility. The conversion process will vary depending on the original format. The IDSgrep build system includes Perl scripts to automate such conversions from several popular Han character databases, and it is expected that users with data in other formats will find it reasonably easy to do similar conversion-

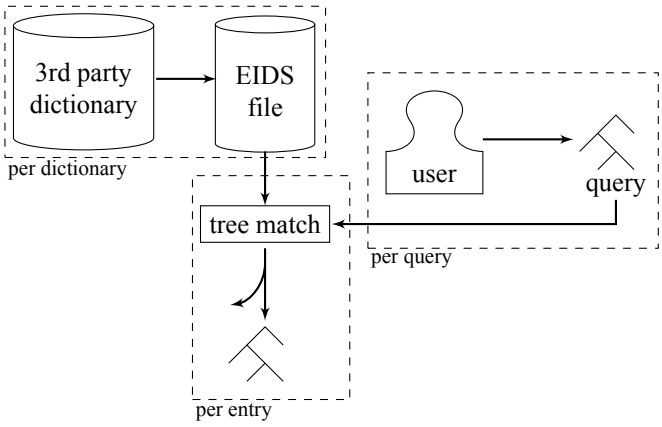

Figure 9: Answering queries with tree matching.

s on their own data. The EIDS file is generated once for each dictionary, during software installation; it does not need to be re-converted during use of the system.

The command-line utility starts its work when the user enters a query. Like the Unix grep utility that inspired it, the IDSgrep utility accepts a single query on its command line, displays the query's results, and then terminates.

Per-query processing is shown at the right of Figure 9. At its simplest, this processing consists only of parsing the user's query into an internal representation of the corresponding EIDS tree. Then the command-line utility starts parsing one or more EIDS files as its main input, specified by command-line options or a default configuration. For each EIDS tree in the input, it tests whether the input tree matches the query tree, according to the matching rules of the previous section. Formally, this is an evaluation of the $match(N, H)$ function defined in Section 3, where the needle N is the user's query and the haystack H iterates through every tree in the database. Any trees for which *match* returns T are displayed as query results.

Early versions of IDSgrep used a design like this one, matching the query against every tree in the database by recursive descent. In typical real-life cases, it gives reasonable performance, typically a few seconds to answer a query on a desktop PC with a large dictionary. That is as fast as one human user can type queries on a command line. However, the worst-case performance of the recursive tree match with a complicated query is potentially very bad (at least exponential-time) because it may recurse with a large branching factor. Even with realistic non-worst-case queries, applications like Web services, computational linguistics research, and resource-constrained smart phone dictionaries may require better performance.

The straightforward recursive descent approach may be unacceptably slow for two reasons: the command-line utility runs the full tree match operation on every dictionary entry for every query, and the tree match operation is itself slow. In the following subsections we describe algorithmic enhancements in the current IDSgrep implementation, which address both those points. The current implementation performs fewer full tree matches, by attempting to guess their results using cheaper analysis of precomputed index data. If IDSgrep can prove what the result of the tree match would be, it can skip actually performing the full recursive test. That technique is called match filtering, and described in Subsections 4.1 through 4.3. When it is necessary to perform a tree match test after all, IDSgrep also attempts to make the matching

operation faster, using dynamic programming in the form of memoization. Memoization reduces the superpolynomial worst-case time bound to $O(n^3)$ per tree match, subject to some limitations. That technique is described in Subsection 4.4.

Figure 10 illustrates the design of the full IDSgrep system, with the details that were omitted from Figure 9.

4.1. Match filtering

The events described in the boxes labeled ``per entry'' in Figures 9 and 10 occur tens or hundreds of thousands of times for every query. Operations in the ``per query'' and ``per dictionary'' boxes are performed much less often. Even if achieving a small reduction (microseconds) in the average per-entry processing time requires a relatively large amount of additional work (milliseconds) in per-query setup, it will still improve the user-visible performance of the system. The per-dictionary preprocessing is not visible to the user in normal operation at all, and effectively comes for free. To a first-order approximation, the per-entry processing time is all that matters to performance.

It makes sense, then, that to improve the performance of the system we should address the slowest thing in the ``per entry'' box. That is the tree match operation, which computes $\mathit{match}(N, H)$ for a search pattern N and dictionary entry H by an expensive recursive algorithm. Match *filtering* attempts to reduce the expense of computing $\mathit{match}(N, H)$ by computing it on fewer values of H. We hope that the time saved by the avoided calls to *match* will more than compensate for whatever additional work may be required to recognize the ruled-out entries. This general approach is the foundation of the well-known *Bloom filtering* technique (Bloom 1970).

Let $\mathcal{E}$ be the set of all EIDS trees, let $\mathcal{F}$ and $\mathcal{V}$ be sets called the *filters* and the *vectors* respectively, and define functions $\mathit{filt} : \mathcal{E} \to \mathcal{F}$, $\mathit{vec} : \mathcal{E} \to \mathcal{V}$, and $\mathit{check} : \mathcal{F} \times \mathcal{V} \to \{\mathsf{T}, \mathsf{F}\}$ such that for all $N, H \in \mathcal{E}$, this property holds:

$$\mathit{match}(N, H) = \mathsf{T} \Rightarrow \mathit{check}(\mathit{filt}(N), \mathit{vec}(H)) = \mathsf{T}\,. \tag{1}$$

We precompute and store, per dictionary, the values of $\mathit{vec}(H)$ for each H in the dictionary. These go in a separate file stored with the dictionary and labeled ``bit vector file'' in Figure 10. To answer a user query N, we compute $\mathit{filt}(N)$, once per query; then $\mathit{check}(\mathit{filt}(N), \mathit{vec}(H))$, for every query and every entry. When $\mathit{check}(\mathit{filt}(N), \mathit{vec}(H)) = \mathsf{F}$, we can skip to the next entry. The property (1) guarantees that in such a case, we know $\mathit{match}(N, H) = \mathsf{F}$ without calculating it explicitly. Only when $\mathit{check}(\mathit{filt}(N), \mathit{vec}(H)) = \mathsf{T}$ do we invoke the more complicated algorithm to compute $\mathit{match}(N, H)$, and return H as a match to N should that return T. The computation of *check* is time-critical because it happens for every dictionary entry and every query; the other steps occur in preprocessing, many fewer times.

If (1) holds, then the algorithm is correct in the sense of returning the same set of match results that we would get without filtering. Such filtering schemes clearly exist; having *check* return T unconditionally is a trivial example. However, for filtering to be of benefit, the following properties (paraphrased from the IDSgrep user manual (Skala 2014a)) are desirable. Note that unlike (1), which must be absolutely true, it is acceptable for these properties to hold only on average in common cases. When they fail, the system becomes less efficient but remains correct.

- Although *vec* may be expensive to compute, the elements of $\mathcal{V}$ it produces as output are small enough that we can afford to store them for all dictionary entries.
- Although *filt* may be expensive to compute in comparison to *check*, it is still fast enough

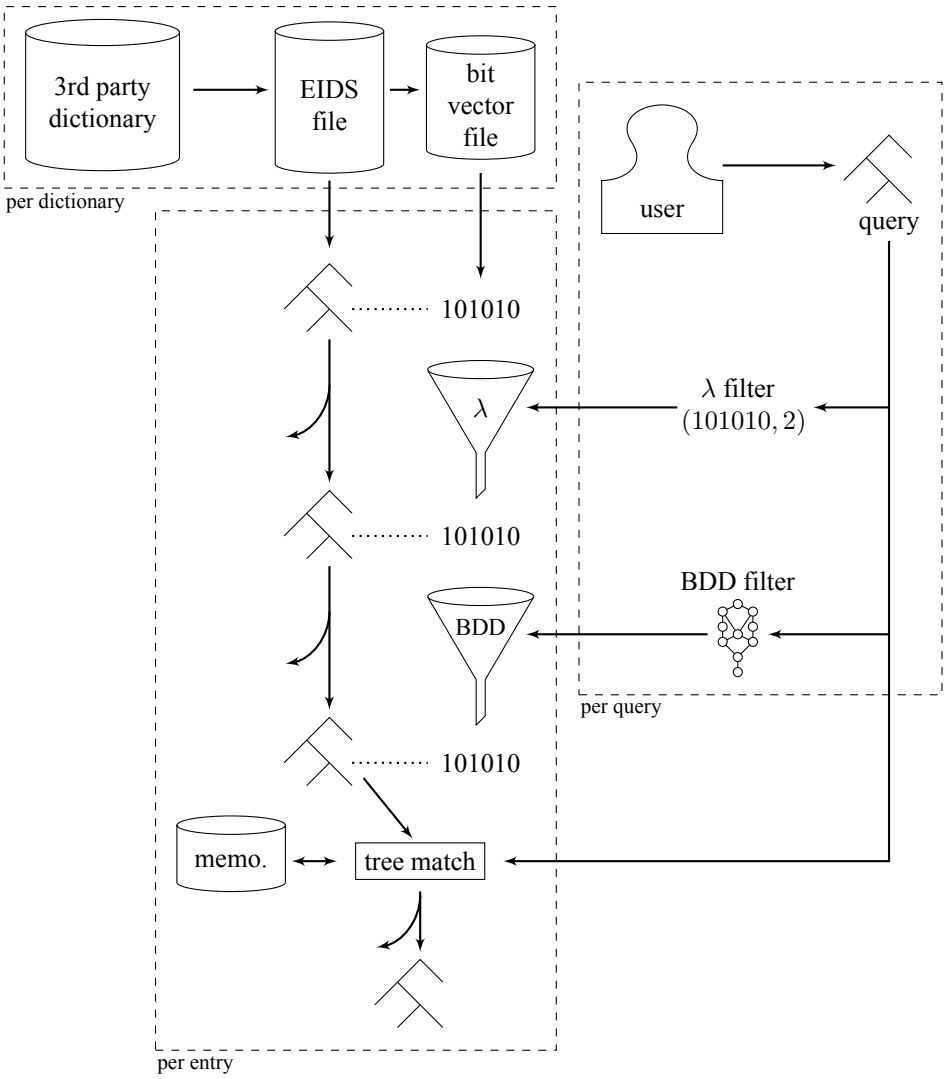

Figure 10: IDSgrep processing with algorithmic enhancements.

that we can reasonably afford to compute it once for each user-initiated query (each value of N).
- The *check* function is very fast.
- The converse of (1) is usually true, on the distribution of search patterns and dictionary entries we expect to see in practice.

IDSgrep uses two layers of match filtering, which share their definition of *vec* but differ in their definitions of *filt* and *check*. As shown in Figure 10, EIDS trees from the dictionary are joined with their vectors from the bit vector file as they pass through the filtering layers. If the lambda filter proves that a bit vector will not match, then the entry is discarded without further processing, as symbolized by the arrow curling off to the side. Any vectors that pass the lambda filter test are checked by the BDD filter, with another opportunity for them to be discarded. Only those entries whose bit vectors survive both filters pass to the EIDS syntax parser and recursive-descent tree match. The tree match is by far the most expensive single operation in per-query processing, so skipping it as many times as possible makes a real difference in overall performance.

4.2. Bit vectors and lambda filters

Classical Bloom filtering is a match filtering scheme much as described here, applied to subset membership tests. It is desired to quickly test whether objects may be elements of some set that was fixed in advance, without the cost of storing and searching the entire set. The Bloom filter applies a small constant number of hash functions to an input object and uses them as indices into an array of bits. Each bit is set to 1 if and only if any of the hashes applied to any of the objects in the set would produce that hash value. When testing an unknown object, we check all its corresponding bits and return it as a possible match if and only if they are all 1. The scheme may produce some false positives (possible matches that were not in the set) but no false negatives. Any object that is in the set will necessarily return the ``possible match'' result. For any other object, we are checking multiple bits that are effectively chosen at random. As long as the array is large enough to contain a significant fraction of zero bits, it is reasonably likely that at least one of the bits checked will be zero and we can return ``definitely no match''. The desired properties hold of recognizing all objects in the set, and not too many others. Bloom (1970) gives a detailed analysis, which has become well-known.

It is also well-known that some algebraic operations can be applied to Bloom filters with useful results: for instance, the bitwise AND of two Bloom filter bit arrays is a Bloom filter that recognizes the intersection of the sets they recognize. Guo et al. (2010) give a good summary of results on algebraic combinations of standard Bloom filters, in the context of introducing an enhanced version of their own. IDSgrep uses this algebraic view of Bloom filters to create a *filter calculus* in which the filter approximating a complicated EIDS-match query is calculated from filters that approximate matching its subtrees. The notion of *generalized zero* (Skala et al. 2010) detected by counting bits and testing against a threshold is also applied.

Let $\mathcal{V}$, the set of vectors for match filtering, be $\{0,1\}^{128}$; that is, the set of 128-bit binary vectors. Let $\mathcal{F}$, the set of possible filters, be $\mathcal{V} \times \mathbb{Z}$; each filter is a pair (m, λ) of a vector m from $\mathcal{V}$ (called the *mask*) and an integer λ. We call filters of this type *lambda filters*. Let *check*$((m, \lambda), v) = \mathsf{T}$ if and only if strictly more than λ bits are 1 in the bitwise AND of m and v. Where these filters come from (the function *filt*) will be discussed later. For now, note that we can create a match-everything filter by setting $\lambda = -1$, regardless of the vectors m and v. Therefore with an appropriate definition of *filt* these definitions are capable of describing

a filtering scheme that is at least correct if not highly efficient.

The function *vec* : $\mathcal{E} \to \mathcal{V}$, which associates a 128-bit vector with an EIDS tree, is defined as follows. Let T be the input tree. The 128-bit result is divided into four 32-bit words; call them v_1, v_2, v_3, v_4. A hash function chooses three distinct bits in v_1 to be set to 1, depending on the head of T, or three bits representing the hash of the empty string if there is no head. These bits must be distinct; it is a uniform choice among the 4960 combinations of three out of 32 bits. Then another hash function sets three more bits (distinct from each other but not necessarily from the three representing the head) depending on the arity and functor of T. Thus, v_1 will contain between three and six 1 bits.

If we are looking for trees that exactly match a specific head at the root, we can say with certainty that any tree having the desired head will have three specific bits in its vector equal to 1, namely the three bits corresponding to the hash of the head. A filter $(m, 2)$ with m selecting exactly those three bits will match all such trees. It will not match many others, because with only at most six bits of 32 set, the chances are good that at least one of the three bits chosen from v_1 will be 0 for any tree that does not have the desired head. This filter foreshadows the more complicated filters we will use to approximate the full EIDS *match* operation.

In the case of a nullary tree, the calculation of *vec*(T) stops at this point, leaving $v_2 = v_3 = v_4 = 0$. For higher arities, we set bits to 1 in the other three words of the vector. The head, functor, and arity of the first child of the root are hashed to choose two sets of three bits in v_2. Similarly, the last child of the root sets two sets of three bits in v_3. All other nodes below the root, including the middle child of a ternary root and all lower-level descendants, hash into v_4.

Figure 11 illustrates the bit vector calculation for a dictionary entry, with values in hexadecimal and binary notation as indicated by the subscripts. Words in the vector, and bits in the words, are indexed in one-based little-endian order; the least significant bit of the vector is bit 1 of v_1. Each node in the tree selects three bits with its head (or lack of a head) and three bits with its functor/arity pair, as shown by the indices on the dashed arrows; these bits are set to 1 in a word selected by the location of the node in the tree. All three of the nullary nodes with semicolon functors select the bit combination 7, 13, 25 in their respective words. In the case of a unary root (rare in Han character dictionary entries, but they occur in search patterns), the single child would set bits to 1 in both v_2 and v_3. Both of the two grandchildren of the root select v_4, so it ends up with a greater density of 1 bits than the other words.

We mentioned that using hashing to choose three bits in the first 32-bit word of a filter mask, and requiring all three to be 1, gives a filter for the query that matches that head. Lambda filters are constructed the same way for matching the state of having no head at the root, and for matching a functor/arity pair. Starting from these *atomic* lambda filters, the filter calculus for lambda filters combines them to produce lambda filters that approximate more complicated queries.

Given two lambda filters, we can construct a new lambda filter guaranteed to match if at least one of them matches. For instance, if at least one of the eight-bit filters $(10011000, 2)$ and $(00000111, 2)$ matches, then $(10011111, 2)$ also matches. The new filter is found by taking the bitwise OR of the masks, and the minimum of the λ values. Some precision is lost, because $(10011111, 2)$ will also match in some cases where neither of the original filters would match. This is the Boolean OR operation in the filter calculus of lambda filters. The IDSgrep implementation also includes an optimization for the case where the two λ values are not equal. In that case we can show that it is possible to remove bits from the mask of the filter with greater λ before combining them, to achieve a slightly more precise filter as

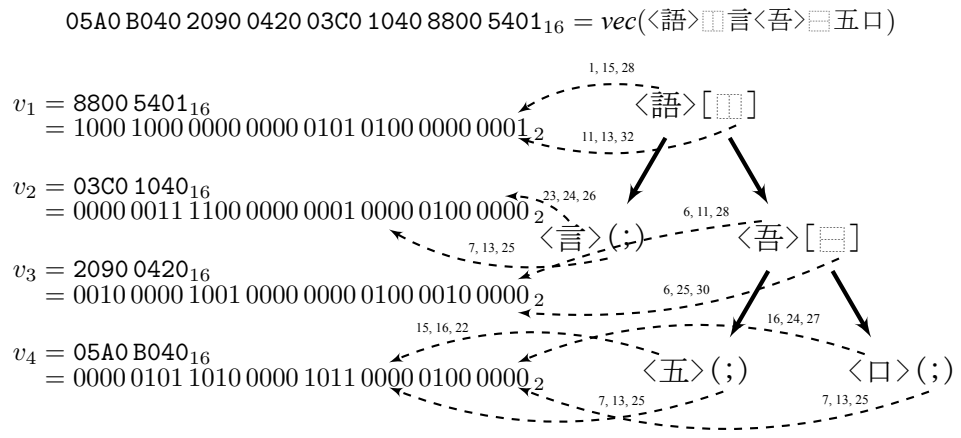

Figure 11: Calculating the *vec* function.

the result of the OR while preserving correctness. A similar operation is defined for Boolean AND on lambda filters.

One more filter calculus operation is necessary to do basic EIDS matching: computing a lambda filter for matching a child, given that we have a lambda filter for matching at the root. This operation makes use of the word structure of the *vec* function. If the lambda filter based at the root has a mask that checks certain bits in v_1, then a filter for the same tree pattern in the first child should check the same bits in v_2. More than one bit in the root filter may correspond to the same bit in the child filter. For instance, a filter that would require 1 bits at equivalent positions in v_2 and v_4 when checking for a pattern at the root, can only look at that bit position in v_4 once, when checking for the same pattern in a child of the root. The value of λ must then be reduced accordingly. IDSgrep's implementation of the child-matching filter calculus operation computes the worst-case number of such collisions and subtracts it from λ, as well as performing the necessary rearrangement of bits in the filter mask.

These operations suffice to implement lambda filters for the *basic_match* function. Recall that *basic_match*$(N, H) = \mathsf{T}$ if and only if either N and H both have heads and those heads are identical, or N and H do not both have heads but they have the same functor and arity and all corresponding children match recursively. This defines EIDS matching without special matching operators; and each operation in that logical statement corresponds to a filter calculus operation. Starting with the atomic lambda filters for matching heads, head absence, and functor/arity, we can use OR, AND, and the ``match child'' transformation to calculate a lambda filter *filt*(N) with the desired property that *check*(*filt*(N), *vec*$(H)) = \mathsf{T}$ whenever N and H match under basic matching, and not often otherwise.

It takes a little more work to handle the special matching operators of IDSgrep. When calculating a filter for a query, IDSgrep follows the same logic as in the definition of the *match* function: check whether the functor and arity of the root of the query EIDS tree describe a special matching operator, follow rules specific to the operator if one is recognized, and apply basic matching otherwise.

The Boolean OR and AND operations [|] and [&] are handled by straightforward application of the OR and AND rules already defined by the filter calculus. The Boolean NOT

operator . !. is more difficult. Any case of a filter matching might be a false positive for which the full query would not match, and would become a forbidden false negative if we attempted to invert it. Therefore, if we attempt to evaluate the . !. operator in pure filter calculus where given a filter for a query x we must find a filter for . !. x, the only correct result will be a match-everything filter, regardless of x. IDSgrep processes the . !. operator by temporarily breaking out of the filter calculus to apply Boolean algebra to the underlying EIDS matching queries, which contain more useful information than their lambda filters. It applies double negation (NOT NOT x equivalent to x), de Morgan's theorem (NOT (x OR y) equivalent to (NOT x) AND (NOT y)), and recognizes the special cases of the match-everything and match-nothing queries (?) and . !. (?) . Only if the negation cannot be removed or postponed by algebra does IDSgrep resort to returning the match-everything filter.

Only a few other special operators are handled by the lambda filtering scheme. The filter for the literal-match operator . =. is just the filter for its child under basic matching, ignoring any special meaning of the child's functor and echoing the definition of . =. . The unordered-match operator . *. is expanded into an equivalent construction using Boolean OR on all permutations of children, before calculating the lambda filter on the expansion. Similarly, the match-anywhere operator ... is expanded into an equivalent OR of four queries: one each for matching at the root, first child, last child, and all other descendants. These cases correspond neatly to the four 32-bit words in the bit vector.

One can imagine a similar expansion of an associative query using . @. into an equivalent query without . @. , but such a construction would suffer from a combinatorial explosion, containing one subquery for each way of parenthesizing the original, all combined with Boolean OR. IDSgrep avoids this possibility by just using the match-everything filter for . @. queries, giving behavior that is at least correct and not significantly worse than no filtering at all. Similarly, the user-predicate and regular-expression matching operators, which escape to other matching functions that defy algebraic analysis, are always assigned match-everything lambda filters.

By applying these rules, IDSgrep can calculate a lambda filter for any query, having the desired properties of no false negatives and reasonably few false positives. That is the first layer of filtering, used to avoid both explicit calculation of the *match* function and evaluation of the somewhat more expensive second layer of filtering, which is described next.

4.3. BDD filters

The precision of lambda filtering is limited by the implicit requirement of the filter calculus that the result of an operation on lambda filters must itself be a lambda filter. IDSgrep's *BDD filters* attempt to retain more precision, by using a more powerful formalism for expressing the filters. They are named for the *binary decision diagram* (BDD), which is a well-known data structure for representing Boolean functions of bit vectors. The data structure is well described in standard references (Knuth 2009) and we do not explain its inner workings beyond IDSgrep's perspective. IDSgrep uses a third-party open-source BDD library named BuDDy (Lind-Nielsen 2014) as a black box implementation of BDDs.

BuDDy provides Boolean functions as objects for the software to manipulate. These objects support simple operations like ``compute the function that is the Boolean OR of these two functions''. They also support much more complicated functions, like ``count the number of distinct input vectors on which this function is true''. Some of these operations are NP-hard and cannot be performed in reasonable time in the worst case; but the algorithms and the implementation include many optimizations for the cases expected in practice.

IDSgrep applies BDDs directly to the match filtering problem. Let $\mathcal{V}$, the set of vectors for match filtering, be $\{0,1\}^{128}$, the 128-bit binary vectors just as in lambda filtering. Similarly, let *vec* be the same function used in the lambda filtering of the previous section. BDD filtering operates on the same vectors. It differs in the definition of $\mathcal{F}$, the set of filters: here, $\mathcal{F}$ is the set of all *monotonic* Boolean functions on 128-bit binary vectors. Monotonic Boolean functions of binary vectors are those where, if the function's value for a given input is true, changing a 0 bit in the input to a 1 can never cause the function's value to change to false. This requirement limits the complexity of the functions somewhat, but $\mathcal{F}$ remains a huge set, and we will later apply a further constraint on the complexity of the functions that will actually be used. Elements of $\mathcal{F}$ are represented by binary decision diagrams, and the *check* function on a BDD and a vector simply evaluates the function that the BDD represents, using the vector as input.

The calculus of BDD filters starts with atomic filters and applies operations to create filters for arbitrary EIDS matching queries, much like the calculus of lambda filters. Just as with lambda filters, when a query has a given head at the root, there are three bits in the first word of its vector that must be 1. It is easy to create a BDD for the function true if and only if all those bits are 1, and that is the atomic BDD filter for matching that head value. So far, it represents the same function that the equivalent lambda filter would represent. Similar atomic BDDs are easy to define for matching the absence of a head at the root, and any given functor/arity pair.

Boolean OR and AND use the relevant BDD operations directly. Here is the first significant difference from lambda filtering. The lambda filter for the OR of two lambda filters may also match on some vectors that would not have been matched by either input, representing a loss of precision. The OR of two BDDs instead represents *exactly* the function that is true if and only if at least one of the input functions is true. The BDD for AND is, similarly, an exact representation of that operation. There is no loss of precision in these simplest BDD filter calculus operations.

Matching a child may involve some loss of precision because of bit collisions, just as in the case of lambda filters. The definition of the child-matching transformation on BDD filters is also somewhat more complicated. But as with lambda filters, it is possible given a BDD filter for matching a pattern at the root, to compute a BDD filter that examines a rearranged subset of the vector bits to match the same pattern as a specified child of the root.

Applying these operations to the atomic filters, as in lambda filtering, gives BDD filters for basic EIDS matching. Filters for special matching operators are also constructed using similar techniques to those used for lambda filtering. Boolean OR and AND, and the literal match operator .=., are straightforward. Boolean NOT is handled by examining the underlying EIDS-match queries and applying Boolean algebra, as in lambda filtering, with a match-everything BDD filter used as a fallback where necessary.

Unordered match .*. is expanded into a Boolean OR of the matched permutations, and match-anywhere ... into an OR of four expressions for matching at the root, as first child, as last child, or as any other descendant. Finally, the .@., ./., and .#. operators are assigned match-everything BDD filters, as in the lambda filtering case.

One significant issue remains: the complexity of the calculated BDD filters. The BuDDy library is quite efficient, containing most of the usual optimizations expected of a BDD library. But even with good constants, any BDD library must repeatedly solve NP-hard problems to maintain the data structure, and there is a potential for both time and space requirements to become exponential. We could imagine a pathological query that would cause IDSgrep to spend so much time in the per-query preprocessing as to outweigh any possible advantage in

the per-entry scanning.

IDSgrep addresses that concern by enforcing a constraint on the complexity of any BDD returned by filter calculus operations. Recall that adding false positives to a filter will never cause incorrect results from the overall filtering and matching algorithm; it will only reduce efficiency by requiring more full EIDS tree matches. We can always change a BDD filter to one that returns the possible-match result on more vectors, as long as we do not cause it to stop returning possible-match on any vectors for which it already does so. Furthermore, the BuDDy library can provide an estimate of the cost of a BDD in time and space, in the form of a count of the nodes in an internal data structure.

With these facts in mind, IDSgrep has a simple way of avoiding excessive resource consumption in BDD filter calculus operations: after each operation, it checks whether the result is too complicated, and if so, it applies an *existential quantification* operation to the BDD. That has the effect of reducing the complexity of the internal representation, possibly losing some precision, without rendering the filtering result incorrect. By applying existential quantification whenever necessary (which is rarely, in practice), IDSgrep enforces an upper limit on the complexity of the BDD filter and prevents the per-query preprocessing from running out of control.

4.4. Match memoization

Straightforward recursive descent evaluation of the IDSgrep matching function takes exponential time in the worst case. The definition of *match* recurses more than once into its children in the cases of the `...` operator (each subtree of the haystack against the needle) and the `.*.` operator (the haystack against as many as six permutations of the needle). A matching pattern with many nested instances of these may take a very long time to evaluate.

However, the straightforward recursive descent algorithm lends itself to dynamic programming via memoization. The needle and haystack each contain a linear number of subtrees, and each pair of subtrees deterministically does or does not match. We can store and re-use the result of *match*(N, H) for each pair (N, H) in a table of size $O(n^2)$.

Computing the *match* function given the table entries for all subtrees of its arguments is a linear-time operation in the worst case implemented within IDSgrep, which is the `.@.` operator; that operator potentially requires comparing node lists of linear length. IDSgrep stores strings using a hashed symbol table for constant-time equality tests on strings, so `.@.` can be implemented in $O(n)$ time, plus recursion into the subtrees. The other operators are constant-time after recursion is paid for. Multiplying $O(n^2)$ subproblems with $O(n)$ time per subproblem gives $O(n^3)$ time overall. This analysis excludes the `./.` regular-expression matching operator. That operator connects IDSgrep to the external PCRE library, which does not offer time guarantees; but $O(n^2)$ remains as a bound on the number of calls IDSgrep makes to PCRE.

In the practical implementation, on commonly-occurring queries, match memoization is rarely beneficial. Users seldom construct queries with more than one or two instances of `...` or `.*.`, rarely nesting them even then. The additional constant factors associated with hashing before and after each node-to-node matching test, increased memory working set size resulting from random accesses to the hash table, and so on, are considerable. But to guard against pathological or malicious queries, the IDSgrep utility implements memoization conditional on the matching pattern. When the matching pattern includes more than two instances of `...` or `.*.`, IDSgrep will memoize *match*, giving a $O(n^3)$ time bound while still avoiding the overhead of maintaining the hash table in the usual case of simpler queries.

5. Experimental evaluation

This section presents experimental evaluation of IDSgrep version 0.5.1, with BuDDy 2.4 and dictionaries from CJKVI as supplied by the IDSgrep distribution; CHISE-IDS 0.25; the September 1, 2013 released version of KanjiVG; and Tsukurimashou 0.8. Speed results are user CPU time on a MacBook Pro equipped with a 2.3GHz Intel iCore i7 CPU and 8G of RAM. The operating system was Mac OS X 10.9.5.

5.1. Match filtering

The main experimental question of interest here was how the algorithmic enhancements (both kinds of match filtering, and match memoization) affect query speed. The speed test queries were chosen to be similar to those users typically make in practice, and to exercise the relevant features of the query language. There was an emphasis on queries involving wildcards and Boolean logic, which are more challenging to search algorithms. Some queries returning no hits, and some returning large numbers of hits, were tested. However, artificial pathological cases that users would not be expected to create in actual use were not included in the main speed evaluation.

We started with the 160 Grade Two Jōyō Kanji characters as taught in the Japanese school system, and found their entries in the CJKVI Japanese-language character structure dictionary generated by the IDSgrep installer. That dictionary excludes characters with no breakdown into smaller components, according to its own rules for determining what qualifies as an atomic component; other dictionaries do have entries for some of the characters that CJKVI excludes. For 144 of the Grade Two characters, CJKVI provided an entry; and for each of those we removed all heads from the EIDS tree except at the leaves, to create a tree that might be further modified to form a test query. For instance, from the dictionary entry 【数】⿰<娄>⿱米女攵, removing the non-leaf heads gave ⿰⿱米女攵.

The test query set contained 1642 queries and was constructed as follows:

- All 160 Grade Two kanji as single characters for head-to-head matching.
- Match-anywhere applied to each of the 160 Grade Two kanji.
- The 144 dictionary entries with heads removed.
- The 144 headless dictionary queries with each leaf in turn replaced by the wildcard (?). For instance, ⿰⿱米女攵 generated ⿰⿱?女攵, ⿰⿱米?攵, and ⿰⿱米女?. This process created 536 queries, reduced to 524 by removing duplicates.
- Unordered-match applied to the root of each of the 144 headless dictionary entries, for instance *⿰⿱米女攵 from ⿰⿱米女攵.
- For all headless dictionary entries that included the necessary structure for associative match to be meaningful, such as ⿰日⿱⿱十一寸, the same tree with associative match inserted, such as ⿰日@⿱⿱十一寸. There were 53 of these, including three where it was possible to apply @ to two different associative structures in the same tree.
- For all headless dictionary entries with binary roots, the same tree with the root replaced by the Boolean OR operator, for instance |口儿 from ⿱口儿. There were 137 of these. The seven headless dictionary entries without binary roots all had ⿱ as root functor.
- For each x chosen from among the 160 Grade Two kanji, the queries &...x...日 and &...x!...日. That makes 320 queries, intended to test Boolean AND and NOT with match-anywhere in usage patterns similar to multi-radical search; since 日 occurs within some of the Grade Two kanji, some of these queries will necessarily return no results.

The literal-match, regular-expression, and user-defined predicate operators, which exist for

special purposes not directly relevant to structural query of Han characters, were excluded from the test query list.

Four dictionaries of character decompositions were used for the speed test: CJKVI (Japanese version, supplied in the IDSgrep 0.5.1 package) with 74361 entries totaling 4461882 bytes; CHISE IDS version 0.25, with 133606 entries totaling 5555303 bytes; the KanjiVG release of September 1, 2013, with 6666 entries totaling 175257 bytes; and Tsukurimashou 0.8, with 2655 entries totaling 106021 bytes. This makes a total of 217288 dictionary entries. Although CHISE IDS supplies more than half the entries, the other dictionaries often use different structural descriptions of frequently-occurring characters and components, and so they add some diversity in the trees to be searched.

The IDSgrep 0.5.1 installer is also capable of building a dictionary from the EDICT2 file (Breen 2014a), but that was not included in the present experiment because it is a dictionary of word meanings and pronunciations, intended to be searched primarily with PCRE. Since it contains many large entries that would tend to be skipped by the test queries aimed at single characters, its inclusion in the timing results would tend to overstate the advantages of bit filtering in the character dictionary applications studied here.

Table 1 summarizes the test queries, test dictionaries, and numbers of hits returned (final tree matches, which are the same regardless of filtering). The mean number of hits per query was 101.72. The top three queries by number of hits were . . . 心, &. . . 心!. . . 日, and . . . 止, returning 5353, 5152, and 4074 hits respectively. There were 67 queries that returned no hits, and 560 that returned one hit each. Note that the total hit counts for Boolean AND and match-anywhere queries are the same because of the design of the test query set. Each match-anywhere test query corresponds to a pair of Boolean AND test queries. Their results are disjoint and when unified are the same hits returned by the match-anywhere query.

	queries	CJKVI-J	CHISE	KanjiVG	Tsuku.	TOTAL
dictionary size		74361	133606	6666	2655	217288
Grade Two kanji	160	144	123	160	320	747
match-anywhere Gr. 2	160	28757	37394	1903	634	68688
headless entries	144	152	67	18	16	253
wildcard leaves	524	16237	9998	1212	357	27804
unordered match	144	157	67	18	16	258
associative match	53	54	9	0	2	65
Boolean OR	137	145	31	129	213	518
Boolean AND	320	28757	37394	1903	634	68688
TOTAL	1642	74403	85083	5343	2192	167021

Table 1: Test queries, test dictionaries, and tree-match hit counts.

We ran 20 loops of the 1642 test queries against the 217288 entries of the test dictionaries in each of four treatments: the default configuration with both filters; each filter alone; and no match filtering. Filtering treatments were selected using IDSgrep's built-in command-line options, and times were collected using its statistics option. The resulting hit counts are in Table 2. Percentages refer to the input of each filtering or matching layer; for instance, when both filters were used the 30980198 BDD hits per loop represented 19.8% of the 156617732 trees that had already passed the lambda filter. All the results shown are per loop of $1642 \times 217288 = 356786896$ matching tests. Because the filtering and tree match algorithms are deterministic, the filter hit counts are the same for all loops of each condition.

filters	λ hits		BDD hits		tree hits	
both	156617732	(43.9%)	30980198	(19.8%)	167021	(0.54%)
BDD			30980206	(8.7%)	167021	(0.54%)
λ	156617732	(43.9%)			167021	(0.11%)
none					167021	(0.05%)

Table 2: Hit count results for the filtering layers.

Timing results are in Table 3. These are means and sample standard deviations in seconds for the same 20 loops, measured in user CPU seconds using the MacOS `getrusage` system call. The columns marked ``overall'' represent the entire user-visible query time, which includes both per-query preprocessing and per-entry matching and filtering. Because users always execute complete queries consisting of both preprocessing and search, that is the most useful measurement of query performance.

	overall		preprocessing	
filters	mean	st. dev.	mean	st. dev
both	110.46	0.62	1.8377	0.0315
BDD	118.81	1.24	1.8333	0.0410
λ	321.29	1.78	0.0170	0.0007
none	583.72	2.42	0.0062	0.0005

Table 3: Timing results for the filtering layers (seconds).

However, for the sake of better understanding the algorithm's performance, per-query preprocessing time is also measured separately. Those figures include creating the filters, parsing the query string into an internal tree structure, and a few miscellaneous tasks performed once per query, such as allocating memory.

The design of the software does not allow for creating a BDD filter without at least partially creating a lambda filter, because those two operations are done simultaneously, sharing much of their code. Only the BDD creation can be removed entirely by appropriate compile-time options. As a result, the preprocessing times for the ``BDD only'' experimental condition also include whatever lambda filter preprocessing could not be disabled at run time. The ``lambda only'' results were measured on a separately-compiled IDSgrep binary with no BDD support at all. It is clear from those measurements that the preprocessing time for lambda filtering alone is so much less than for BDD filtering as to be negligible when they are combined. The ``no filtering'' condition also has nonzero preprocessing time, because of the parsing and memory allocation common to all configurations.

The times for the ``both'' and ``BDD'' conditions seem close enough for a statistical test to be appropriate. One-factor ANOVA is the obvious choice. The four treatments are considered as a single factor because of the close interaction between the BDD and lambda filters when both are used. That is properly a single treatment, not two treatments independently applied. The data (especially for preprocessing time) violate the necessary assumption of homoscedasticity, with standard deviations varying greatly between the different treatments. Since the standard deviations increase with the means, a logarithmic transformation is appropriate. Sample means and standard deviations for the base-10 logarithms of the timing measurements are shown in Table 4.

	overall		preprocessing	
filters	mean	st. dev.	mean	st. dev
both	2.0432	0.0024	0.2642	0.0073
BDD	2.0748	0.0045	0.2631	0.0096
λ	2.5069	0.0024	-1.7696	0.0173
none	2.7662	0.0018	-2.2107	0.0324

Table 4: Transformed timing results ($\log_{10}$ of seconds).

Applying one-way ANOVA to the transformed timing data for overall query time gives $F(3, 76) > 2.7 \times 10^5$, $p < 0.0001$, so we reject the null hypothesis that the mean overall times are the same for the four filtering treatments. The Tukey HSD (Honestly Significant Difference) test applied to the pairwise differences in transformed overall time gives $HSD(0.01) = 0.0058$, less than any of the pairwise differences in sample means, so all the differences are statistically significant with $p < 0.01$.

On the transformed processing times, one-way ANOVA gives $F(3, 76) > 9.2 \times 10^4$, $p < 0.0001$, so we reject the null hypothesis that the mean preprocessing times are the same for the four filtering treatments. The Tukey HSD test applied to these pairwise differences gives $HSD(0.01) = 0.0039$, less than any of the pairwise differences except between the ``both'' and ``BDD'' treatments, so all the pairwise differences except that pair are statistically significant with $p < 0.01$. The difference in preprocessing time between ``both'' and ``BDD'' also is not significant at the 95% or 90% levels ($HSD(0.05) = 0.0032$, $HSD(0.10) = 0.0028$).

5.2. Memoization

To check the effects of memoization, we compiled a modified version of the IDSgrep command-line utility in which the test for whether to use memoization was disabled. The configuration was otherwise default; in particular, both layers of bit vector filtering were active. We then ran queries against the combined test dictionaries for the character component 寺 nested inside k match-anywhere operators, for k from 1 to 10. These queries each return the same set of 261 results, but would be expected to become slower as k increases.

Table 5 and Figure 12 show the time in seconds per query for each value of k, with sample mean and standard deviation for 100 trials. Also shown in the figure is a linear function fit by least squares to the default-configuration query times, and an exponential function fit to the no-memoization times. The exponential function was found by least squares fitting a line to the logarithms of the data, to avoid overemphasis on the larger numbers.

5.3. Bit vector generation

Although the cost of creating the bit vectors for IDSgrep is not directly visible to dictionary users, it may be of interest in evaluating the algorithm. Table 6 shows the time required to generate bit vectors for each of the test dictionaries, in units of user CPU seconds with sample mean and standard deviation over a sample of 20 trials. Other tasks included in the software installation process but not properly part of the IDSgrep system, such as decompressing archived data, compiling C source code, and converting foreign formats into EIDS, are excluded from these results.

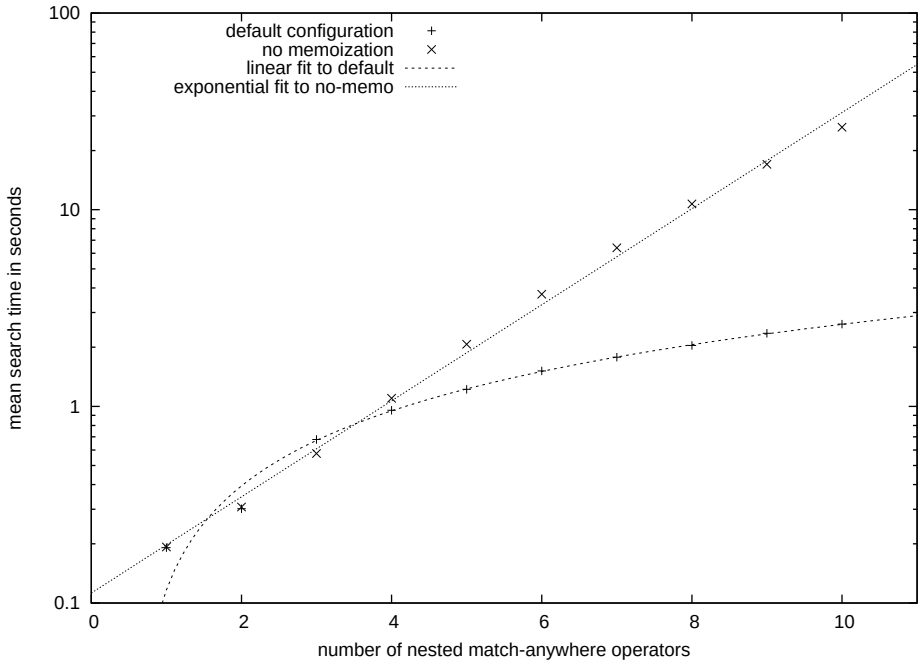

Figure 12: Query times for nested match-anywhere with and without memoization.

5.4. Comparison to other software

Because IDSgrep is currently the unique implementation of its own query language, there is nothing else quite like it that could serve as the perfect benchmark for comparison. Examining the effect of bit vector indexing was the main goal of our evaluation, and for that it suffices to compare IDSgrep using bit vectors against IDSgrep not using bit vectors. But to place IDSgrep in perspective with other software, we used the following criteria to select comparable systems to test against. Software packages compared to IDSgrep should be:

1. software packages;
2. capable of searching for, not only describing, Han characters;
3. capable of at least one kind of structural query more sophisticated than just looking up an exact match to a single character;
4. available to the general public;
5. suitable for automated performance testing;
6. typical of the systems that users choose in actual practice;
7. the best implementations available; and
8. collectively representative of the range of systems in use.

We found two systems that met all the criteria: GNU grep (Free Software Foundation 2014) and Tregex (Levy and Andrew 2006).

GNU grep is a popular version of the standard command-line grep utility. Its basic function is to take a text file as input and pass through to the output all lines that match a query pattern---much as IDSgrep does for EIDS trees. EIDS dictionary files are easily converted into a form with exactly one tree on each line, so that grep's line-based matching will correspond

	default config		no memoization	
k	mean	st. dev.	mean	st. dev
1	0.191	0.003	0.193	0.002
2	0.302	0.003	0.308	0.004
3	0.678	0.004	0.576	0.008
4	0.955	0.021	1.099	0.014
5	1.220	0.019	2.068	0.030
6	1.513	0.026	3.719	0.051
7	1.779	0.037	6.398	0.070
8	2.040	0.039	10.697	0.123
9	2.354	0.039	17.007	0.235
10	2.618	0.050	26.285	0.273

Table 5: Query times for nested match-anywhere with and without memoization.

dictionary	entries	bytes	mean	st. dev
CJKVI-J	74361	4461882	0.1748	0.0027
CHISE	133606	5555303	0.2144	0.0026
KanjiVG	6666	175257	0.0088	0.0007
Tsukurimashou	2655	106021	0.0056	0.0006
TOTAL	217288	10298463	0.4036	

Table 6: Time cost of bit vector generation (seconds).

to matching of EIDS trees. The query patterns for grep are usually described as *regular expressions*, and regular expressions as such cannot be used to recognize non-regular languages (the language of balanced parentheses being the classical example). EIDS and Unicode IDS, as context-free non-regular languages, are not well suited to sophisticated queries with grep. However, GNU grep, like many recent implementations of grep, supports back-references and other extensions of regular expression syntax that allow it to recognize a limited class of non-regular languages. As one of its authors describes in an electronic mailing list posting (Haertel 2010), it is a heavily optimized implementation of standard DFA-based string search techniques.

Many users seeking functionality similar to that of IDSgrep would reach for GNU grep in particular. Many more would use something similar to GNU grep, such as another implementation of Unix grep, or the regular expression search built into a text editor. The CHISE project (Morioka 2014), in particular, offers integration with XEmacs (Wing et al. 2014) for grep-like regular expression and substring search. As such, GNU grep is a good candidate for comparison to IDSgrep. It can be, and actually is in practice, used to serve many of the same needs as IDSgrep, and it is amenable to automated performance testing.

Not all IDSgrep queries can reasonably be translated into grep-like string search queries, but two important kinds of IDSgrep queries easily can be. Looking up a single character with a query like 語, which should return exactly those dictionary entries that have that character as head, is one. Looking for a single component anywhere in the entry with a query like . . . 言 is the other. We can translate these two queries into GNU grep queries 【語】 and 言 respectively. In IDSgrep's default databases (after some processing to normalize the format), each entry is one text line, lenticular brackets (synonymous with ASCII angle brackets in EIDS

syntax) occur only in entry heads, and characters like 言 do not occur in unusual contexts. As a result these GNU grep queries return the same entries as the original IDSgrep queries despite not having technically identical semantics. Simple Boolean queries involving AND and NOT operations can also be performed easily with grep, by passing the output of one grep instance through another.

In our test query set, 640 queries are thus of a form that can easily be processed with GNU grep. We ran 20 loops of those 640 queries against the combined test databases, using IDSgrep in its default configuration, IDSgrep with bit vector filtering turned off (for a possibly fairer comparison to grep, which uses no pre-computed index), and GNU grep version 2.20 in its ``fast'' mode (fgrep), which does simple string matching without the more sophisticated features of extended regular expressions. The timing results in user CPU seconds, with means and standard deviations over the sample of 20 loops, are shown in Table 7.

search software	mean	std. dev.
IDSgrep (default)	94.50	0.28
IDSgrep (no filtering)	262.45	0.20
GNU grep	5.48	0.03

Table 7: Timing comparison between IDSgrep and GNU grep (seconds).

The crucial disadvantage of GNU grep is that it cannot do the complicated subtree-matching queries for which IDSgrep is intended. Stanford Tregex (Levy and Andrew 2006) is a more powerful tree-matching program originating in the computational linguistics community, and one of the nearest pre-existing equivalents to IDSgrep in terms of expressive power and application domain. It is intended for use with parse trees of sentences in databases like the Penn Treebank (Marcus et al. 1993), and it supports a query language based on describing constraints between nodes. The available constraints are chosen based on the community's experience with what kinds of queries users wish to make on parse trees. In general, Tregex has more emphasis on longer-scale ancestry and predecessor/successor relationships, and less emphasis on fixed-arity nodes and the sequence of children, compared to IDSgrep. Tregex, like GNU grep, is a popular implementation that many users would naturally choose in situations where IDSgrep might be applicable. Tregex is also typical of a larger class of tree matching software packages.

To make EIDS trees searchable with Tregex, it was necessary to translate the trees into the syntax used by tree bank files, which expresses variable-arity trees using nested parentheses and alphanumeric labels. We used a Perl script to do this translation, using identifier names that encoded the EIDS tree node information (arities, heads, and functors) into alphanumeric strings. For example, the EIDS 〈明〉⿰日月 was translated to the tree bank-style tree (R (C2FF0 H660E (A3B H65E5) (A3B H6708))). That can be read as a root R with one child, which in turn is binary with functor U+2FF0 (⿰) and head U+660E (明). The next two children in the tree bank-style tree are both nullary with functor U+003B (semicolon, which is implicit in the EIDS syntax), and heads U+65E5 (日) and U+6708 (月) respectively.

With the trees encoded into tree bank syntax, almost all of our test query set could be translated into Tregex queries by recursively expressing the match condition at each node in Tregex terms. Only the associative-match queries were omitted. To exactly match the semantics of IDSgrep's associative-match operator would involve many additional Tregex constraints to exclude exotic cases. It is not clear how to perform a fair comparison between the two systems on such queries. Writing a query with exactly identical semantics to IDSgrep would

seem to penalize Tregex by making it do a great deal of superfluous computation, but tuning the queries closely to the data would render the comparison meaningless on general data.

We ran 20 loops of all test queries except the 53 associative-match queries against the combined test dictionaries, using IDSgrep in its default configuration, IDSgrep with match filtering turned off, and Tregex. We used Tregex version 3.4.1, with the Sun Mac JDK 8, update 20. The timing results (measured in seconds, with the means and sample standard deviations over the 20 loops) are in Table 8.

search software	mean	std. dev.
IDSgrep (default)	102.42	0.35
IDSgrep (no filtering)	568.52	1.27
Tregex	8430.66	69.29

Table 8: Timing comparison between IDSgrep and Tregex (seconds).

5.5. Discussion

Table 1 illustrates the differences between the four test dictionaries. On the 160 single-character searches, the CHISE and CJKVI-J dictionaries each return fewer than 160 results, because these dictionaries only contain entries for characters when they have nontrivial decompositions. The KanjiVG dictionary, however, derives from a data source primarily concerned with the strokes rather than the component breakdown. It includes an entry for every character in its scope even where the decomposition is trivial. Tsukurimashou includes two entries (giving decomposition and source code information in separate entries) for every character.

Bearing in mind that our test queries are derived from CJKVI-J entries, the headless-entry and unordered-match queries return only a few results in the other databases because of differences in how the dictionaries break down the same characters. That effect shows up more strongly with the associative-match queries. The 53 associative queries return 54 results from CJKVI-J despite its canonicalization intended to make associative matching unnecessary, because it contains separate entries for U+66F8 and U+2F8CC, both of which look like 書 and have the same decomposition. Even with associative matching, only a few of the queries in this class return results from the other databases, because of differing breakdowns. Finally, note the similarity in all databases (nearly identical match counts) between the headless-entry and unordered-match queries. It appears to be a property of the Han character set that there are very few pairs of characters differing only by a reordering of subtrees at the root level (for instance, swapping the left and right of a left-right character).

The timing and filter hit results in Tables 2 and 3 show the effect of filtering. Lambda filtering eliminates a little over half of the tree tests given the distribution of queries and dictionary data we used. With tree tests accounting for most of the running time, lambda filtering gives a factor of approximately 1.8 speed-up overall. BDD filtering eliminates 91.3% of the tree tests and gives a factor of approximately 4.9 speed-up.

However, there is little additional benefit to using both filters at once. Although we found the difference in overall time to be statistically significant, the sample mean running time for both filters together is just 7.1% faster than for BDD filtering alone. Note that the difference in raw number of BDD hits per loop is only eight hits, on a total of almost 31 million. Any tree match avoided by the lambda filter would almost certainly be eliminated by the BDD filter anyway. The speed benefit from the lambda filter in this configuration can be attributed to

avoiding the BDD checks themselves. Both filter implementations exist in the current version of IDSgrep because of the history of its development, but a new implementation might omit the lambda filters without any important loss of speed. On the other hand, because it does not require an external BDD library, the lambda filter implementation may still be useful in installations where the external dependency is undesirable. Its much smaller preprocessing time could also be of benefit if the dictionaries were very small.

The correlation between the filters can be understood by considering how they share their vector definitions. We can imagine an ideal exact filtering function of bit vectors that returns true exactly on those bit vectors, and only those, which could possibly be associated with matching trees. Such a function would extract all possible information from the bit vectors and give the best possible filtering given our vector-creating function. The lambda filtering function is a coarse one-sided approximation of the ideal filtering function, but the BDD filter as implemented almost perfectly approximates the ideal. It gives false positives *relative to the ideal filtering function* only in the relatively rare cases where implementation compromises force a loss of precision. To see a tree check eliminated by the lambda filter and not the BDD filter, the tree check would first have to be among the roughly 56% of non-hits that the lambda filter is able to eliminate at all. But it would also have to be in the very small set of hits that differ between the implementation of BDD filters and the hypothetical ideal bit-vector filter. As long as the definition of the vectors remains the same, not much improvement in filtering is possible.

The preprocessing time results show that with dictionaries of the size used in this experiment, per-query preprocessing is not an important part of the overall query cost. It is less than 2% of the overall search time in the case of using both filters, even less in the other treatments. The cost of per-query preprocessing would only have an important effect on overall query time when searching much smaller dictionaries. Preprocessing for lambda filters is also insignificant compared to preprocessing for BDD filters.

Per-dictionary preprocessing times are not directly relevant to the user experience, but at less than half a second for all dictionaries combined, these times seem reasonable and should present no impediment to frequent dictionary updates. Dividing the no-filtering query loop time of 583.72 seconds by 1642 queries per loop gives a mean per-query time of 0.355 seconds, which can be compared to the bit vector generation time of 0.403 seconds. Both those numbers are for processing the entire dictionary set once. We can say that generating the bit vectors for dictionary entries is only slightly slower than doing unfiltered searches on them.

Tree-match memoization is not expected to make much difference to practical applications, but the experimental results on it illustrate the asymptotic behavior of the algorithm. Applying increasing numbers of nested match-anywhere operators slows down the matching linearly in the default configuration (with memoization on demand). Despite the worst-case bound of $O(n^3)$ for the algorithm, the case tested in our experiment involves checking a linearly-increasing query against a database that does not change, with constant-time tests for each pair of nodes. Then $\Theta(n)$ performance is what we might expect. With memoization, the matching time increases exponentially, also as we would expect from theory, although as seen in Figure 12, the fit there is less close.

GNU grep and Tregex are typical of what someone without IDSgrep might use for similar purposes. The CHISE project (Morioka 2014), in particular, offers an online grep-like substring search of its IDS database, as well as editor plugins to support local regular expression search on the same data. For the simple match-anywhere and head-to-head single character queries we tested, GNU grep is unquestionably much faster than IDSgrep, by a factor of 47.9

(in sample mean user CPU time per loop) compared to IDSgrep without filtering, or 17.2 with filtering. The comparison without filtering may be more fair because GNU grep does not benefit from a precomputed index. On the other hand, IDSgrep is not really intended for this kind of query; its goal is to answer detailed structural queries which GNU grep cannot do at all. It remains that IDSgrep might benefit from switching to a faster string search algorithm from its current filtered tree match, when it can detect that a query is of a simple form that could be answered by string search.

Tregex is roughly comparable to IDSgrep for more advanced structural queries. Both are specialized to their application domains, and they have different application domains, so they are not perfectly comparable. In our comparison, covering almost all of our original query speed test set, IDSgrep was found to be 82.3 times as fast as Tregex if allowed to use its precomputed bit vector indices, or 14.8 times as fast without them. Factors contributing to the speed difference may include:

- a basic speed difference between IDSgrep's compiled C code and Tregex's Java;
- differences between our test databases and the kind of data Tregex more commonly uses; and
- the fact that Tregex solves a harder problem.

Although our test queries did not use this feature directly, the Tregex query language allows binding named variables to nodes in the tree and applying Boolean constraints to them. It is not difficult to prove that this feature makes Tregex's matching problem NP-hard, in contrast to IDSgrep's matching problem, which has a $O(n^3)$ time bound.

To summarize the comparison between programs, it would be reasonable to say that GNU grep is designed for speed in preference to expressive power; Tregex is designed for expressive power in preference to speed; and IDSgrep falls somewhere in between.

6. Conclusions and future work

We have described the IDSgrep structural query system for Han character dictionaries: its data model, query language, and details of the algorithms it uses, with experimental results.

IDSgrep was first developed to support Japanese-language font development in the Tsukurimashou Project. The user base in that application is small and highly trained. But the concept of structural queries on Han characters is of potential interest to other dictionary users, including many who are not computer scientists. It is an open question to what extent language learners and other dictionary users may find IDSgrep useful. A study comparing the experience of human users when using various dictionary systems including IDSgrep would be of interest. Use of this new resource to support computational linguistics research on Han character sets is also a direction for future study.

The algorithmic ideas in IDSgrep have more general application. In particular, BDD filtering of Bloom-style bit vectors is novel, at least to the computational linguistics domain, and may be a useful extension to existing bit vector techniques for parsing of unification-based grammars. Application of the filtering technique to problems beyond character dictionaries is another possible direction for future work.

References

Aït-Kaci, H., Boyer, R. S., Lincoln, P. and Nasr, R., 1989, Efficient implementation of lattice operations, *ACM Transactions on Programming Languages and Systems*, vol. 11, no. 1, pp. 115--146.

Apel, U., 2014, KanjiVG. Retrieved April 21, 2014 from `http://kanjivg.tagaini.net/`.

Bloom, B. H., 1970, Space/time trade-offs in hash coding with allowable errors, *Communications of the ACM*, vol. 13, no. 7, pp. 422--426.

Breen, J., 2014a, The EDICT dictionary file. Retrieved April 21, 2014 from `http://www.csse.monash.edu.au/~jwb/edict.html`.

Breen, J., 2014b, WWWJDIC: Online Japanese Dictionary Service. Retrieved April 21, 2014 from `http://www.csse.monash.edu.au/~jwb/cgi-bin/wwwjdic.cgi`.

Choi, Y. S., 2011, Tree pattern expression for extracting information from syntactically parsed text corpora, *Data Mining and Knowledge Discovery*, vol. 22, no. 1-2, pp. 211--231.

Chu, C., Nakazawa, T., Kawahara, D. and Kurohashi, S., 2013, Chinese-Japanese machine translation exploiting Chinese characters, *ACM Transactions on Asian Language Information Processing*, vol. 12, no. 4, pp. 16:1--16:25.

Clocksin, W. F. and Mellish, C. S., 1987, *Programming in Prolog*, Springer.

Creamer, T. B. I., 1989, Shuowen Jiezi and textual criticism in China, *International Journal of Lexicography*, vol. 2, no. 3, pp. 176--187.

Dürst, M. J., 1996, Prolog for structured character description and font design, *Journal of Logic Programming*, vol. 26, no. 2, pp. 133--146.

Fasih, A., 2015, IDSgrep on the Web. Retrieved February 11, 2015 from `http://fasiha.github.io/idsgrep-emscripten/`.

Free Software Foundation, 2014, GNU Grep 2.18. Retrieved April 21, 2014 from `http://www.gnu.org/software/grep/manual/grep.html`.

Guo, D., Wu, J., Chen, H., Yuan, Y. and Luo, X., 2010, The dynamic Bloom filters, *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 1, pp. 120--133.

Haertel, M., 2010, why GNU grep is fast. Mailing list posting. Retrieved April 21, 2014 from `http://lists.freebsd.org/pipermail/freebsd-current/2010-August/019310.html`.

Halpern, J., editor, 2013, *The Kodansha Kanji Learner's Dictionary: Revised and Expanded*, Kodansha.

Hao, T. and Zhu, C., 2013, Toward a professional platform for Chinese character conversion, *ACM Transactions on Asian Language Information Processing*, vol. 12, no. 1, pp. 1:1--1:22.

Hazel, P., 2014, Pcre---Perl compatible regular expressions. Retrieved April 21, 2014 from `http://www.pcre.org/`.

Hobby, J. D. and Guoan, G., 1984, A Chinese meta-font, *TUGboat*, vol. 5, no. 2, pp. 119--136.

Hosek, D., 1989, Design of Oriental characters with metafont, *TUGboat*, vol. 10, no. 4, pp. 499--502.

Kamichi, K., 2014, GlyphWiki. Retrieved April 21, 2014 from `http://en.glyphwiki.org/wiki/GlyphWiki:MainPage`.

Kaneta, Y., Arimura, H. and Raman, R., 2012, Faster bit-parallel algorithms for unordered pseudo-tree matching and tree homeomorphism, *Journal of Discrete Algorithms*, vol. 14, no. 0, pp. 119--135.

Kawabata, T., 2012, Normalization of ideographic description sequence, in *36th Internationalization and Unicode Conference (IUC36), Santa Clara, California, USA, October 22--24, 2012*. Conference without published proceedings.

Knuth, D. E., 1986, *The Metafont Book*, Addison-Wesley.

Knuth, D. E., 2009, *The Art of Computer Programming*, vol. 4, pre-fascicle 1B, Addison-Wesley.

Laguna, J. R., 2005, Hóng-Zì: A Chinese metafont, *TUGboat*, vol. 26, no. 2, pp. 125--128.

Lai, C. and Bird, S., 2010, Querying linguistic trees, *Journal of Logic, Language and Infor-*

mation, vol. 19, no. 1, pp. 53--73.

Landin, P. J., 1964, The mechanical evaluation of expressions, *The Computer Journal*, vol. 6, no. 4, pp. 308--320.

Levy, R. and Andrew, G., 2006, Tregex and tsurgeon: Tools for querying and manipulating tree data structures, in Calzolari, N., Choukri, K., Gangemi, A., Maegaard, B., Mariani, J., Odijk, J. and Tapias, D., editors, *5th International Conference on Language Resources and Evaluation (LREC 2006), Genoa, Italy, 22--28 May 2006*.

Lind-Nielsen, J., 2014, BuDDy: A BDD package. Retrieved April 21, 2014 from `http://buddy.sourceforge.net/manual/main.html`.

Liu, C.-L., Lai, M.-H., Tien, K.-W., Chuang, Y.-H., Wu, S.-H. and Lee, C.-Y., 2011, Visually and phonologically similar characters in incorrect Chinese words: Analyses, identification, and applications, *ACM Transactions on Asian Language Information Processing*, vol. 10, no. 2, pp. 10:1--10:39.

Marcus, M. P., Marcinkiewicz, M. A. and Santorini, B., 1993, Building a large annotated corpus of English: The Penn Treebank, *Computational Linguistics*, vol. 19, no. 2, pp. 313--330.

Mei, T. Y., 1980, LCCD, a language for Chinese character design, Report STAN-CS-80-824, Stanford University, Department of Computer Science.

Morioka, T., 2014, CHISE project. Retrieved April 21, 2014 from `http://www.chise.org/`.

Peebles, D. G., 2007, Scml: A structural representation for Chinese characters, Tech. Rep. TR2007--592, Dartmouth College.

Polách, R., 2011, *Tree Pattern Matching and Tree Expressions*, Master's thesis, Czech Technical University in Prague.

Skala, M., 2014a, *IDSgrep, version 0.5.1*. Retrieved April 21, 2014 from `http://tsukurimashou.sourceforge.jp/idsgrep.pdf`.

Skala, M., 2014b, Tsukurimashou: a Japanese-language font meta-family, *TUGboat*, vol. 34, no. 3, pp. 269--278.

Skala, M., 2014c, Tsukurimashou Font Family and IDSgrep. Retrieved April 21, 2014 from `http://tsukurimashou.sourceforge.jp/`.

Skala, M., Krakovna, V., Kramár, J. and Penn, G., 2010, A generalized-zero-preserving method for compact encoding of concept lattices, in Hajic, J., Carberry, S. and Clark, S., editors, *48th Annual Meeting of the Association for Computational Linguistics (ACL 2010), Uppsala, Sweden, July 11--16, 2010*, pp. 1512--1521, Association for Computational Linguistics.

Skala, M. and Penn, G., 2011, Approximate bit vectors for fast unification, in Kanazawa, M., Kornai, A., Kracht, M. and Seki, H., editors, *The Mathematics of Language: 12th Biennial Conference (MOL 12), Nara, Japan, September 6--8, 2011*, vol. 6878 of *Lecture Notes in Artificial Intelligence*, pp. 158--173, Springer.

Tanaka, T., Iwasaki, H., Nagahashi, K. and Wada, E., 1995, Making kanji skeleton fonts through compositing parts [Japanese], *Transactions of the Information Processing Society of Japan*, vol. 36, no. 9, pp. 2122--2131.

Unicode Consortium, 2011, Ideographic description characters, in *The Unicode Standard, Version 6.0.0*, section 12.2, The Unicode Consortium, Mountain View, USA.

Wing, B. et al., 2014, XEmacs: The next generation of Emacs. Retrieved April 21, 2014 from `http://www.xemacs.org/`.

Wu, S. and Zheng, S., 2009, A structure character modeling for Chinese character glyph de-

scription, in Mahmoud, S. S., Jusoff, K. and Li, K., editors, *2009 International Conference on Electronic Computer Technology, Macau, China, February 20--22, 2009*, pp. 245--248, IEEE Computer Society.

Yiu, C. L. K. and Wong, W., 2003, Chinese character synthesis using Metapost, *TUGboat*, vol. 24, no. 1, pp. 85--93.