Searching in a DVI File

Nigel Chapman

1. Introduction

Most, if not all, DVI previewers and printer drivers provide a facility for selecting a subset of the pages of a document; this subset is specified using the contents of the `\count0` to `\count9` registers that TEX outputs to identify each page of the file. This makes it easy to preview just pages 7, 8 and 9, but what if you know you want to look at the page with the paragraph about Katzenellenbogen by the Sea? If you're not sure how the page makeup worked out, you won't know where that is. Trial and error will find the right page sooner or later, but it would be more convenient if there was a facility for selecting a page by its content, that is, the occurrence on it of a particular string.

Many efficient string searching algorithms already exist; they are used routinely in text editors and other programs. These algorithms take as their input a string of characters — the target — and a pattern. The pattern specifies a set of strings. The task of the searching algorithm is to find the location, if any, within the target of a substring that belongs to the set specified by the pattern. The pattern may simply be a single string, specifying itself, or it may use metacharacters and some formalism such as regular expressions to specify a larger set of strings. In general, the more elaborate the language permitted for specifying patterns, the more elaborate the search algorithm will be. There are well known efficient algorithms for searching for single strings [4] and for sets specified by regular expressions [1].

A DVI file is a sequence of typesetting commands, some of which may have parameters. (DVI commands are fully described in [3,§§583–590].) A user specifying a pattern to search for will want to type that pattern at a terminal using the subset of ASCII that corresponds to printable characters. Thus, before one of the standard string searching algorithms can be employed, either the pattern must be converted to a sequence of DVI commands, the DVI file must be mapped into a string of printable ASCII characters, or both DVI file and pattern must be mapped into some other common representation.

Leaving aside the possibility of using anything more elaborate than simple strings as patterns, a possible approach based on the first of these options is to use TEX to convert the pattern into DVI. Using this approach, it would be possible for patterns to be specified in the TEX language, and thus to make use of macros and to carry out searches on all features of a document, including math mode material and even rules and spaces. Search patterns could be extracted directly from the TEX source of a document. (In fact, for non-trivial search strings they would probably have to be, because of the difficulty of deciding exactly what TEX commands produced some particular output.) However, the problems of interfacing TEX are considerable, and the overheads of running it to process every search string are unlikely to be acceptable. Furthermore, the actual DVI produced by TEX for a particular string will depend on the context in which that string appears. Such elements as interword spacing, line breaking and hyphenation may be very different when the string appears in the middle of a paragraph and when it is typeset in isolation. Thus, even after a pattern was converted to DVI, it would not be possible to apply simple string matching: some sort of fuzzy matching would be necessary.

If converting the pattern to DVI is problematical, what about converting the DVI file to ASCII? This is essentially the same task as that performed by DVI previewers for dumb terminals, and it suffers from the same limitations: only text material can be dealt with properly, and spacing must be approximated. It has the great virtue of being simple, and, once the transformation has been done, any string matching algorithm can be used, including those that support regular expressions. This approach will be looked at further in the next section.

Finally, there is the possibility of converting both the DVI file and the pattern to some common representation. The obvious choice here is the extended character code set used by TEX to specify characters in its math symbol and extension fonts. This requires some means of specifying characters in the pattern other than printable ASCII characters; the obvious way of doing this is by permitting a suitable subset of TEX commands to be used in patterns. Matching can then be done on text and math mode material. This approach is further described in section 3.

2. Text Searching

The text parts of a TEX document will be typeset using TEX's text fonts. Computer Modern text fonts each contain 128 characters and these are laid out in such a way that the printable ASCII characters occur in the positions corresponding to their ASCII codes. The positions corresponding to the ASCII non-printable (control) characters are used for ligatures, accents and Greek letters (see the

font layout tables in Appendix F of [2]). A search pattern typed by a user will only use printable ASCII characters, but it is necessary to ensure that, for example, the pattern `ffi` matches the ligature ffi.

A DVI file can be thought of as a program in the machine code of a virtual typesetting machine. The instructions of this machine perform primitive typesetting operations, such as *set a character* or *select a new font*. These instructions are held in the DVI file as bit patterns. For reading by people, they can be represented by mnemonics, just as conventional machine code instructions are given mnemonic form in assembly language. DVI instructions may have parameters, specifying, for example, the length of a rule to be set, or the code for a character. There are two sets of DVI commands that cause a character to be typeset: the *set* commands, with mnemonics `set_char_0` through `set_char_127`, `set1`, `set2`, `set3` and `set4`, and the *put* commands, `put1`, `put2`, `put3` and `put4`. The first 128 of these are unparameterized and directly specify a character code, the others take an argument of between one and four bytes. TEX always uses `set_char_`n for a character with code n, if $0 \leq n \leq 127$, so any ASCII character will be set with either a `set_char_`n or `put1` (see [3,§585]). Because of kerning, commands for spacing may appear between character setting commands, even within a word. For example, the DVI commands corresponding to the word 'Katzenellenbogen', as it appears near the beginning of the file containing this paper, have the following mnemonic representation.

```
setchar75
setchar97
setchar116
setchar122
setchar101
setchar110
setchar101
setchar108
setchar108
setchar101
setchar110
right2 -18205
setchar98
x0 18205
setchar111
setchar103
setchar101
setchar110
```

(The `right2` and `x0` commands perform horizontal spacing.)

A possible strategy for converting a DVI file into a stream of characters is to scan the file, ignoring everything except *set* and *put* commands, and return the corresponding ASCII code. But this doesn't do quite what is required. In the first place, all characters not in text fonts should also be skipped. To do this, it is necessary to keep track of the current font by interpreting the additional DVI commands `fnt_def`, `fnt` and `fnt_num`. The external name of each font used can be found among the parameters of its `fnt_def`; it is necessary to have *a priori* knowledge of which of these names correspond to text fonts — for Computer Modern fonts this knowledge is obtainable in Appendix F of *The TEXbook* [2].

Secondly, ligatures should be expanded into their component letters, so, for example, when a `set_char_14` command is found, it should be converted to the three letters `f f i`. Dashes should also be expanded, en-dash to `--` and em-dash to `---`. Dotless i and j should be replaced by ordinary i and j. All other non-printable characters will have to be skipped, as should the Spanish punctuation marks ¡ and ¿ and hyphens, the last since a pattern should match irrespective of whether the corresponding occurrence in the DVI file is split across lines. Strictly speaking, to skip hyphens it is necessary to know the `\hyphenchar` of the current font, and this can only be determined by examining the format used in TEXing the document. It will usually be safe, though, to assume it is `'\-`.

If the scheme described is used, a stream of characters can be produced from the DVI file, corresponding to the textual parts of the document, and these can be matched against a search string made up of printable ASCII characters. However, spacing in the DVI file will be entirely skipped, so ASCII space characters in the search string will not match anything. One response to this is to insert space characters in places that correspond to word breaks in the DVI file. Because of what TEX does with glue, however, these cannot be identified with certainty (as the spacing produced by ASCII DVI previewers such as `dvitty` testifies). The easier alternative is to remove all spaces from the search pattern. This will produce spurious matches, e.g., `pullover` will match `pull over`, and vice versa, but this is preferable to failing to match because of incorrectly inserted spaces.

By allowing an escape character in the search pattern, this scheme can be extended to cope with the full range of accents in the Computer Modern text fonts. The obvious scheme is to use plain

TEX's control sequences for accents and accented characters and expand them in the pattern.

Use of negative glue, `\llap`, characters with negative width, and so on can lead to the commands for setting characters occurring in the DVI file in a very different order from that in which the characters appear in the typeset document. Thus, the matching obtained can only ever be approximate unless DVI commands are first sorted by x and y coordinates. For most applications, this probably wouldn't matter, since most searches will be for obviously identifiable words.

3. Searching in Mathematics

A DVI searching program based on the previous section will suffice to find the page with the text 'Katzenellenbogen by the Sea', but what about the page with

$$\left(\sum_{i=1}^{n} \alpha_i^{m_1} + \beta_i^{m_2}\right)^k ?$$

This is much more of a challenge. To begin with: how should the search pattern be specified? The obvious way is to use the TEX language, but this is not, in fact, an answer, because of TEX's powerful macro facilities and its almost unbounded possibilities for redefining the meaning of any character. One must assume at least part of a format in interpreting a search pattern. The obvious choice is the plain format, so that a search string for the above could be `$(\sum_{i=1}^n \alpha_i^{m_1} + \beta_i^{m_2})^k$`. This illustrates several of the problems that must be overcome when searching in math mode material.

Firstly, `\alpha` and `\beta` represent characters from TEX's math italic font; these occur in the same place as the ff and ffi ligatures in a text font and the two have to be distinguished. TEX does this by using mathcodes, which specify a font family and the position in the font (they also specify a type, but that is irrelevant here) and defining control sequences such as `\alpha` using `\mathchardef`. The mathcode values for such symbols can be used by the searching algorithm, provided the DVI file can be turned into a stream of mathcodes, rather than a stream of ASCII characters as described in the previous section. This is not much more difficult than determining when a character occurs in a text font. By keeping track of the external name of the current font it is possible to deduce the current font family; to do this properly, it is necessary to look at the format used to typeset the document and interpret any assignments to `\textfont`k. The lazy alternative is to assume the assignments of plain TEX. As all text fonts are assigned to family 0, the text matching using ASCII codes will still work.

Not all the special symbols available in math mode are defined by `\mathchardef`. Some are built up as a combination of other symbols, for example $\longmapsto$. Such composite symbols must be expanded into their components.† Some of these combinations are defined in quite an elaborate way, but it is relatively easy to deduce the sequence of character setting DVI instructions they will produce.

Multiple sub- and superscripts present a different sort of problem. A term such as x_i^2 can be produced in TEX by either `$x_i^2$` or `$x^2_i$`. Either pattern should match the DVI for x_i^2, irrespective of how that was specified in the original TEX source. Study of Appendix G of *The TEXbook* reveals that when an atom has both a subscript and a superscript, TEX always sets the superscript first, followed by the subscript. Thus, a pattern with both can be normalized to a form with the superscript first. The `_` and `^` can be ignored and the sequence of component characters, in the appropriate families, can be searched for. Again, this may produce spurious matches (`$x_i^2$` will match $x2i$, for example) but will not fail to match when it should.

The last major complication of searching in math mode is caused by those characters that can change their size. (How big are the parentheses in the example at the beginning of this section?) These include the things defined by `\delimiter`, `\mathaccent` and `\radical`, which may give two different mathcode values for the same control sequence. Matters are made more complicated by the fact that any character may be the first in a linked sequence of characters or may be constructed out of several pieces. These series and extensible characters are considered part of the font, and information about them has to be taken from its TFM file.

A control sequence defined by `\delimiter`, or a mathcode that specifies a character that is part of a sequence can be thought of as a pattern that describes a set of alternatives. An extensible character also specifies a set, but, in theory at least, it is an infinite set of strings of the form TR^kMR^kB or TR^kB where T, R, M and B are the top, repeating, middle and bottom pieces of the extensible character, and $k \geq 0$. All these possibilities can be described by regular expressions.

† A seemingly intractable problem here is the underline symbol, represented by `\_` in plain TEX, which is actually a rule.

$$
\begin{aligned}
\mathit{pattern} &\rightarrow \mathit{pterm}\ \{\,\underline{|}\ \mathit{pterm}\}\\
\mathit{pterm} &\rightarrow \mathit{pfactor}\ \{\,\mathit{pfactor}\}\\
\mathit{pfactor} &\rightarrow \mathit{pprimary}\ [\,\underline{*} \mid \underline{?}\,]\\
\mathit{pprimary} &\rightarrow \underline{\{}\,[\,\mathit{pattern}\,]\,\underline{\}} \mid \underline{\$}\ \mathit{mathpattern}\ \underline{\$}\\
&\quad \mid \mathit{pelement}\\
\mathit{pelement} &\rightarrow \underline{\mathrm{char}} \mid \underline{\mathrm{cs}}\\
\mathit{mathpattern} &\rightarrow \mathit{mathelement}\ \{\,\mathit{mathelement}\}\\
\mathit{mathelement} &\rightarrow \mathit{pelement}\,[\,\mathit{scription}\,]\\
&\quad \mid \underline{\{}\ \mathit{mathpat}\,[\,\mathit{scription}\,]\,\underline{\}}\\
\mathit{scription} &\rightarrow \underline{\uparrow}\ \mathit{scriptelement}\,[\,\underline{=}\mathit{scriptelement}\,]\\
&\quad \mid \underline{=}\ \mathit{scriptelement}\,[\,\underline{\uparrow}\mathit{scriptelement}\,]\\
\mathit{scriptelement} &\rightarrow \mathit{pelement} \mid \underline{\{}\,[\,\mathit{mathpat}\,]\,\underline{\}}\\
\mathit{mathpat} &\rightarrow \mathit{mathelement}\ \{\,\mathit{mathelement}\}
\end{aligned}
$$

Figure 1. Grammar for search patterns.

A delimiter specifies $d_1 \mid d_2$ where d_1 and d_2 are the small and large versions of the symbol; a series of characters is just $c_1 \mid c_2 \mid \ldots \mid c_n$, and an extensible recipe is $(TR^*MR^*B) \mid (TR^*B)$, which may be simplified to $TR^*(M \mid \epsilon)R^*B$ (ϵ is the empty string). Thus if patterns are specified as regular expressions, and searching is done using finite state machines, control sequences and characters corresponding to symbols that can change size one way or another can be treated as shorthands for these regular expressions. Note that the fact that the regular expression for an extensible character does not enforce the restriction that the number of repeatable segments above and below the middle must be the same doesn't matter, because no other combination can occur.

In summary, to deal with math mode, the DVI file must be converted to mathcodes specifying font family and character position and the pattern must be modified so that characters and control sequences in it are mapped into regular expressions matching such codes, taking into account the font family characters will be typeset in, delimiter codes, character series and extensible characters.

4. An Implementation

A prototype DVI searching facility based on the ideas in the previous sections has been implemented as part of a TEX-based hypertext system I am developing. Search patterns may be written in a language defined by the extended BNF grammar in Figure 1. (The symbols [,], {, } and | are grammar metacharacters: items enclosed in [and] are optional, those in { and } may occur zero or more times, | separates alternatives. The terminal symbols — the symbols of the pattern language — are underlined. Note the difference between the metacharacters {, } and | and the terminals $\underline{\{}$, $\underline{\}}$ and $\underline{|}$.) In patterns, the $\underline{|}$ operator separates alternatives, the postfix $\underline{*}$ means that the preceding pattern element may occur zero or more times, and the $\underline{?}$ that it is optional. Curly brackets are used to group items: the syntax of the pattern language means that they work as expected to delimit complex sub- and superscripts, but they are also used as brackets to override the default associativity of regular expression operators, in the usual way. The terminals $\underline{\mathrm{char}}$ and $\underline{\mathrm{cs}}$ represent lexical classes consisting of single characters other than pattern metacharacters and TEX-style control sequences, respectively. The metacharacters are represented in patterns by \\, *, \?, etc.

A pattern input by the user is parsed by a simple recursive descent parser, and a data structure for a nondeterministic finite state machine with ϵ transitions is constructed in a syntax-directed manner using the conventional construction (see [1]). When the lowest level parsing function recognizes a control sequence or an extensible character, it returns a primitive finite state machine to recognize the strings described by it. For most control sequences defined by mathchardefs this is just a two state, one transition machine that recognizes the corresponding mathcode. For delimiters and other characters that change their size in different ways, and for composite characters, a more elaborate machine, corresponding to the appropriate regular expressions, as described in section 3, are produced. For any single, non-extensible character, a machine to recognize the character in the appropriate family is constructed. A special control sequence `\any` is recognized; it matches any single symbol.

The code to construct the finite state machines for control sequences and so on is derived from the definitions in `plain.tex` and PL files for the Computer Modern fonts. If a more general facility were desired, tables would have to be constructed automatically from any user-specified TEX format and TFM or PL files, a task requiring non-trivial analysis of these files.

Once the nondeterministic finite state machine with ϵ-transitions has been constructed, the ϵ-transitions are removed, but the machine is not made deterministic. The nondeterministic machine is used directly in the searching process, by keeping track of a set of active states. State transitions are performed by calling a function that returns the next symbol from the DVI file. This function performs the mapping to a sequence of mathcode values as described previously. It is assumed that only Computer Modern fonts are used, and the font

family assignments of plain TEX are wired into the code.

A problem not previously mentioned is that of displaying the location of the matching string if one is found. This facility was felt to be useful. In order to do it, it is necessary to interpret enough of the DVI commands to keep track of the x and y coordinates at which each character would be displayed. When the finite state machine accepts a string, the coordinates of its end are known and an arrow can be displayed pointing back at the matched string. A more elegant method, such as highlighting the entire match in reverse video, requires finding the coordinates of the start of the string too, which is more difficult.

The majority of the code implementing DVI searching is concerned with parsing patterns, constructing nondeterministic finite state machines and removing ϵ-transitions — that is, the code that any searching program based on finite state machines would require. Most of the overhead specifically resulting from the application to DVI searching is in the code to produce primitive machines for control sequences and extensible characters. The code required to transform the DVI into a stream of mathcodes is relatively simple.

5. Conclusion

The implementation described demonstrates that it is possible to search in a DVI file, using the ideas presented here. Consideration of this implementation suggests that text searching is a feature well worth implementing, but that searching in math mode is less clearly worthwhile.

As should be evident, searching in mathematics adds a good deal of complexity and requires large amounts of code for dealing with mathematical control sequences. This in turn requires the manual or automatic processing of TEX formats and PL or TFM files. Extensible characters pretty much dictate the use of finite state machines for searching, rather than some simpler algorithm such as Knuth-Morris-Pratt. Searching in maths also introduces an undesirable feature into the searching interface: some mathematical control sequences are recognized, but others are not. It might seem capricious to a user that it is all right to use `\alpha` in a search pattern, but not, for example, `\pmatrix`. Furthermore, if you actually did want to search for the page containing

$$\begin{pmatrix} x-\lambda & 1 & 0 \\ 0 & x-\lambda & 1 \\ 0 & 0 & x-\lambda \end{pmatrix}$$

it would be no trivial task to construct a suitable pattern out of the facilities available.

However, searching in text is quite another matter. Implementing the transformation of the textual parts of the DVI file to ASCII is very simple. This can easily be combined with any string searching algorithm that scans from left to right to produce an efficient text searching facility. In practice, this is likely to be adequate to select any page of a document by content. As well as the experimental system described in section 4, the `dviscr` previewer distributed with `emtex` already supports basic text searching, correctly handling accents and ligatures. It is to be hoped that, in future, text searching will become a common enhancement to DVI previewers and printer drivers.

References

1. Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms.* Reading, Mass.: Addison-Wesley, 1974.
2. Donald E. Knuth, *Computers and Typesetting, Volume A, The TEXbook.* Reading, Mass.: Addison-Wesley, 1986.
3. Donald E. Knuth, *Computers and Typesetting, Volume B, TEX: The Program.* Reading, Mass.: Addison-Wesley, 1986.
4. Klaus Pirklbauer, 'A study of pattern-matching algorithms', *Structured Programming* **13**(2), 89–98, 1992.

◇ Nigel Chapman
Department of Computer Science,
University College London,
Gower Street,
London, WC1E 6BT
U.K.
Janet: `N.Chapman@uk.ac.ucl.cs`