Chapter 5

Dasher As A Search Tool

5.1 Introduction

The previous chapter described a probabilistic approach to information retrieval in large collections. While it is clear from the results presented above that this approach is capable of providing very high performance in terms of the search results themselves, it is possible to take the idea further and use probability theory to make improvements to the way in which the search string is entered and the results are presented to the user. In Section 1.3.1, the Dasher user interface was described as a method of text entry making use of a probabilistic language model. In fact, it is possible to use Dasher in any context where navigation of data organised as a tree is necessary, and where estimates are available of the probability that the user will want to select a particular item.

This chapter investigates the application of the Dasher user interface to search tasks. For purposes of simplicity, the task is limited to that of 'exact search' — in other words, tasks where the user wishes to obtain information exactly matching a query. Two such tasks are considered — firstly searching for substrings within a text document, and secondly searching for matching records in a database. The analysis presented here concentrates on theoretical aspects of these applications.

5.2 Interactive search

Given a text document, it is often desirable to be able to locate all occurrences of a particular substring. Whilst many text processing applications provide this feature, in the majority of cases the search is non-interactive. In other words, the user provides the full substring, the search is performed, and the user then selects one of the possibly multiple matches. A few applications however provide an interactive search function. In this case, the search is performed after each symbol in the substring has been entered. Possible matches are indicated in the document, and the user may choose either to select a match from the current list, enter more symbols in order to refine the search, or delete symbols in order to make it less specific. Examples of applications providing this functionality include the GNU Emacs text editor

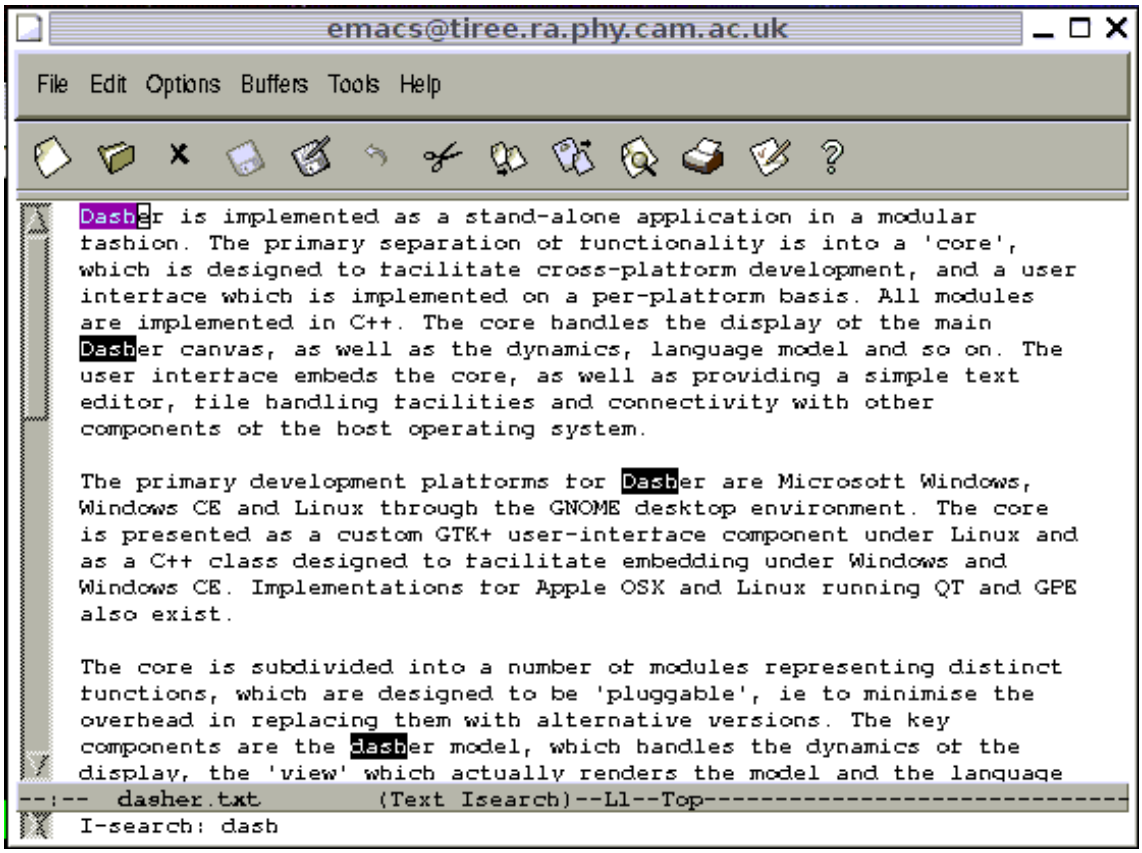


Figure 5.1: The interactive search (I-search) feature in the GNU Emacs text editor. Having entered the substring 'dash' all occurrences of that phrase in the document are highlighted.

[106] (see Figure 5.1) and the Mozilla Firefox web browser [109].

While such an interactive search feature is clearly preferable over the traditional non-interactive version, the use of free keyboard entry is limited in the sense that it offers no suggestion in advance as to which substrings occur in the document; when entering a new character to refine the search, the user is informed of the success of the search only after the character has been entered. This limitation can be addressed by using the Dasher interface.

In order to implement a search tool using Dasher, the problem must be recast into that of selecting a node from a tree structure whose branches are weighted by probabilities. This representation follows naturally from the ordering of all possible substrings in dictionary order. In other words, using Dasher as an interface for search can be considered equivalent to using it for text entry, except that the language model assigns zero probability to all strings which are not present in the document. It is initially assumed that the distribution over possible strings is uniform, so the probability associated with each node is simply proportional to the number of substrings for which that node is a prefix.

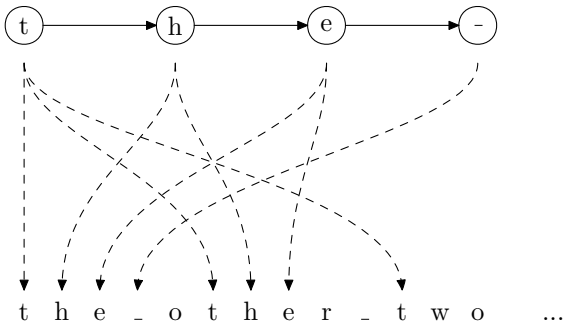


Figure 5.2: An illustration of the data structure used in search Dasher. The stored text, 'the_other_ two...' is shown along the bottom of the diagram, while the chain at the top represents one path from root to node through the trie. Each node stores a list of locations in the text corresponding to the end points of matches. From left to right these are 't', 'th', 'the' and 'the_' respectively. The leaf node, corresponding to '_', has only one item in its list meaning that the string is deterministic from this point on. Rather than increasing the depth of the trie, to find the subsequent symbols it is only necessary to follow the stored text from the position indicated by the unique pointer.

5.2.1 Data structure

The structure used to represent the document in a way which facilitates construction of the Dasher interface is essentially the same as that used in the PPM* text compression algorithm, a member of the PPM family described in Section 2.3.5 which makes use of unbounded context lengths [20]. Data is stored in a trie very similar to that used in Chapter 2, but where each node contains a list of references to locations in the text, which is also stored linearly in memory. The references point to locations in the text corresponding to the last symbol in instances of the N-gram represented by the location in the trie. There is no maximum depth for the trie, but it is terminated whenever a node is reached which contains only one pointer to the text. After this point, there is only one possible continuation of the substring, which can be found simply by following the text itself from the target of the last pointer. The data structure is illustrated in Figure 5.2. By maintaining a list of 'active' nodes in the trie, i.e. those which contain pointers to substrings terminating at the last character read, the structure can be generated in a single parse of the file.

Dasher itself maintains a trie structure corresponding to the hierarchy of boxes displayed on the screen. Each node in the Dasher trie represents one box, and the children of each node are the boxes contained within it. Thus there is a one to one correspondance between nodes in the data structure representing the text being searched, and those in the data structure used by Dasher. Each node in the Dasher trie needs to maintain a pointer to either a node in the data trie, or a position in the text. For Dasher nodes pointing to the data trie, the process of populating their children as the user zooms in is simply a matter of iterating over the children in the data trie. Each child Dasher node gets a pointer to the corresponding data node, or if that node only points to one location in the text, the Dasher node is given

Field	Description	Size (bytes)
Symbol	The symbol in the character set represented by the node.	1
Sibling	A pointer to the next child of the node's parent, or NULL to indicate that this is the last child.	4
Child	A pointer to the first child of the node, or NULL to indicate that no children exist.	4
Pointer	A pointer to a linked list of pointers to locations in the text.	4
	Total:	13

Table 5.1: Sizes of the fields required for storage in the data structure used in Search-Dasher, assuming a 4 byte pointer size.

a pointer to that location. Dasher nodes which already point to the text only have a single child, which has a pointer to the next location in the text[1].

We constructed a prototype implementation of this system, referred to as *Search-Dasher*. Screen captures of this prototype is shown in Figure 5.3. The figures show the sequence of events as the search term 'the' is entered one character at a time. After each character has been entered, all matches in the document are highlighted in the pane to the left of the screen.

5.2.2 Storage requirements

An important consideration when implementing a search facility is the size of storage required to maintain the data in a way which facilitates the search process. The method described above has two components. Firstly, each node in the trie contains a data structure needed to store the details of the trie itself. An illustration of the size of this structure is given in Table 5.1. Associated with each node is a variable length array of pointers into the text, which in order to facilitate construction is stored as a linked list. Each pointer will therefore use 8 bytes, with 4 for the pointer into the text itself and 4 for the link pointer. The total storage requirements are therefore

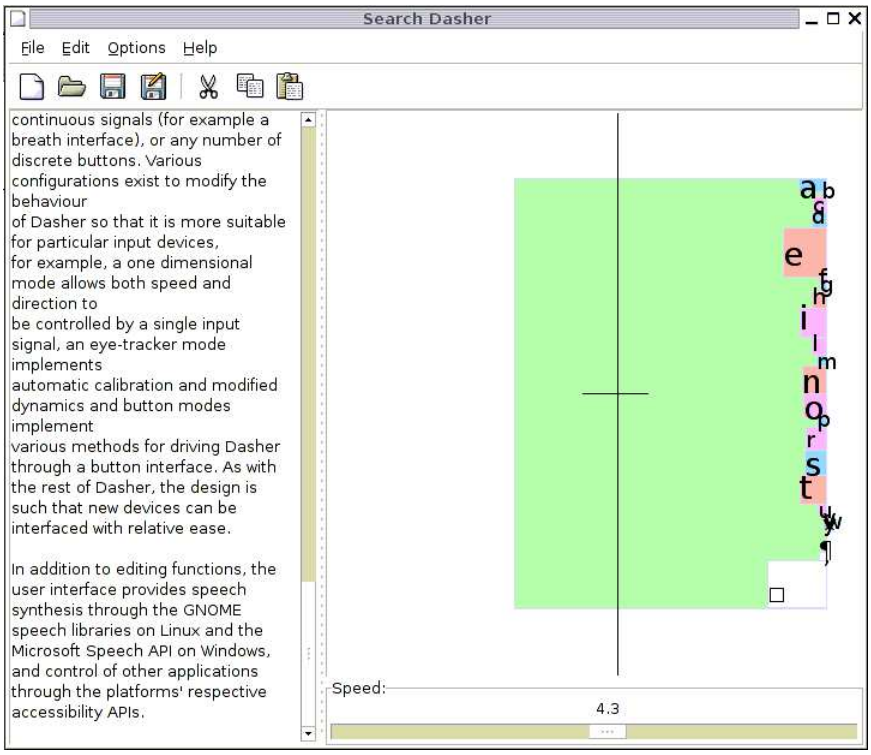
$$b_t = 13 \cdot n_n + 8 \cdot n_p \tag{5.1}$$

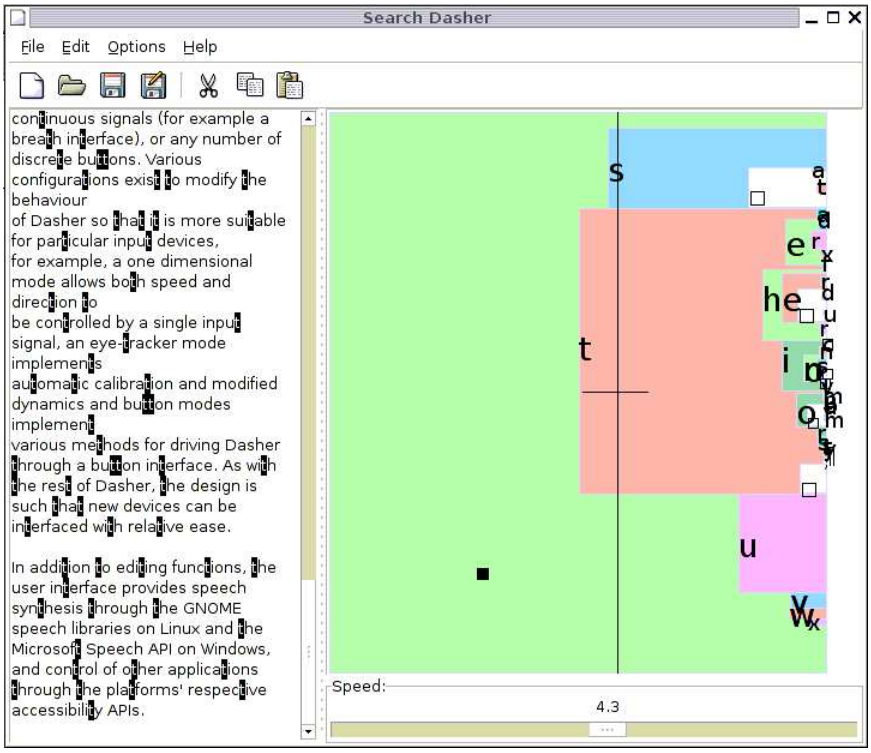
where n_n is the number of nodes and n_p is the number of pointers. Storage requirements are shown using experimentally obtained values for n_n and n_p in Figure 5.4.

The overall storage requirements are fairly large. Documents of 10kB in length will require of the order of 65 times their length in additional storage for the index. This does not necessarily limit the usefulness of this method however - a 10,000 byte document will require 640,000 bytes of storage of storage for the index, which is still quite small by the standards of contemporary desktop PCs. A document of length 100,000 will require 10^7 bytes of storage.

In some cases it is necessary to be able to search for only entire words. In this case,

[1] In practice additional nodes will be added to correspond to 'escaping' from search mode and performing some action on the strings which have been selected.


Search Dasher
File Edit Options Help
continuous signals (for example a breath interface), or any number of discrete buttons. Various configurations exist to modify the behaviour
of Dasher so that it is more suitable for particular input devices,
for example, a one dimensional mode allows both speed and direction to
be controlled by a single input signal, an eye-tracker mode implements
automatic calibration and modified dynamics and button modes implement
various methods for driving Dasher through a button interface. As with the rest of Dasher, the design is such that new devices can be interfaced with relative ease.
In addition to editing functions, the user interface provides speech synthesis through the GNOME speech libraries on Linux and the Microsoft Speech API on Windows, and control of other applications through the platforms' respective accessibility APIs.
Speed:
4.3


Search Dasher
File Edit Options Help
continuous signals (for example a breath interface), or any number of discrete buttons. Various configurations exist to modify the behaviour
of Dasher so that it is more suitable for particular input devices,
for example, a one dimensional mode allows both speed and direction to
be controlled by a single input signal, an eye-tracker mode implements
automatic calibration and modified dynamics and button modes implement
various methods for driving Dasher through a button interface. As with the rest of Dasher, the design is such that new devices can be interfaced with relative ease.
In addition to editing functions, the user interface provides speech synthesis through the GNOME speech libraries on Linux and the Microsoft Speech API on Windows, and control of other applications through the platforms' respective accessibility APIs.
Speed:
4.3

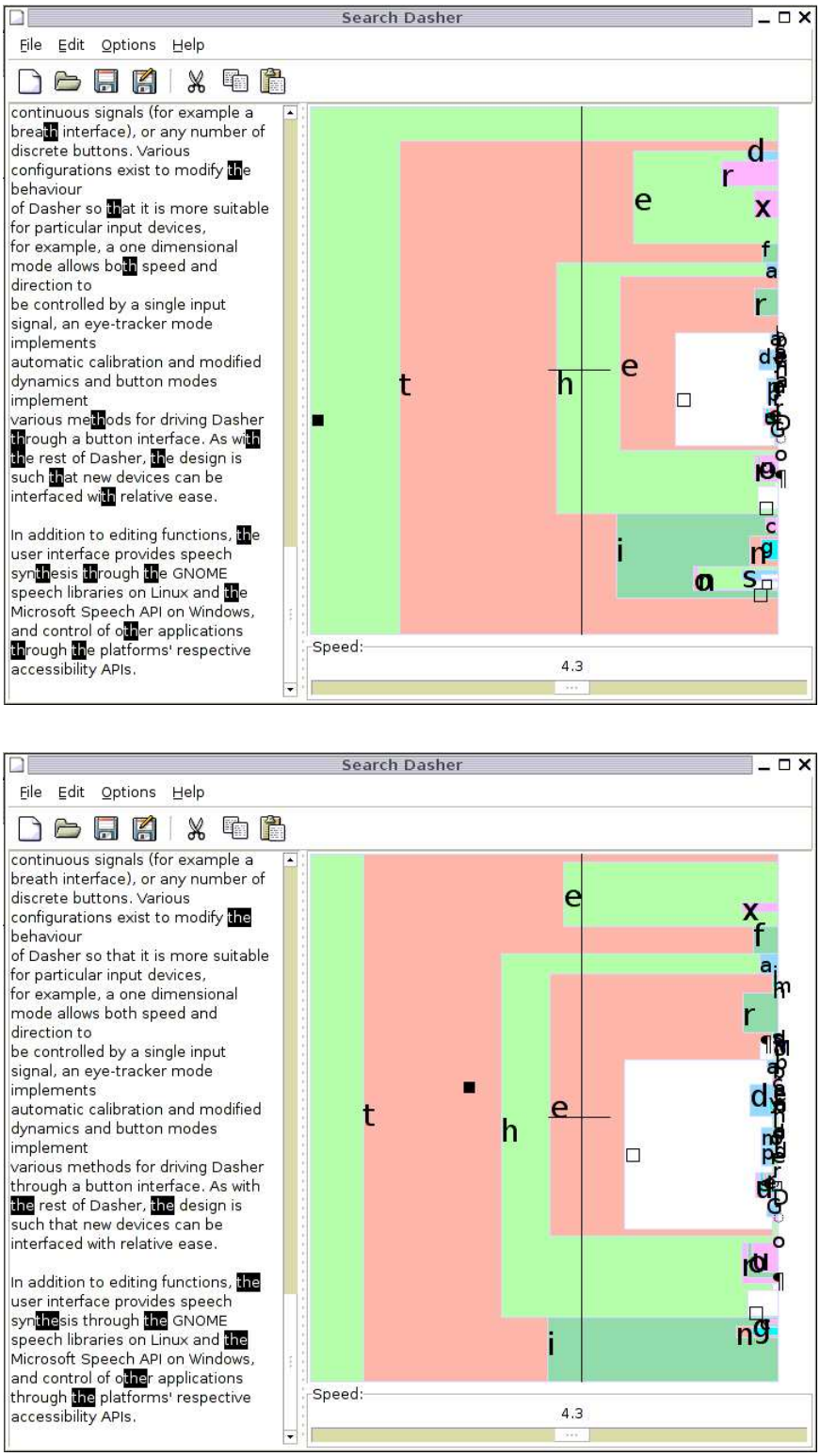


Figure 5.3: Screen captures showing the prototype Search-Dasher implementation

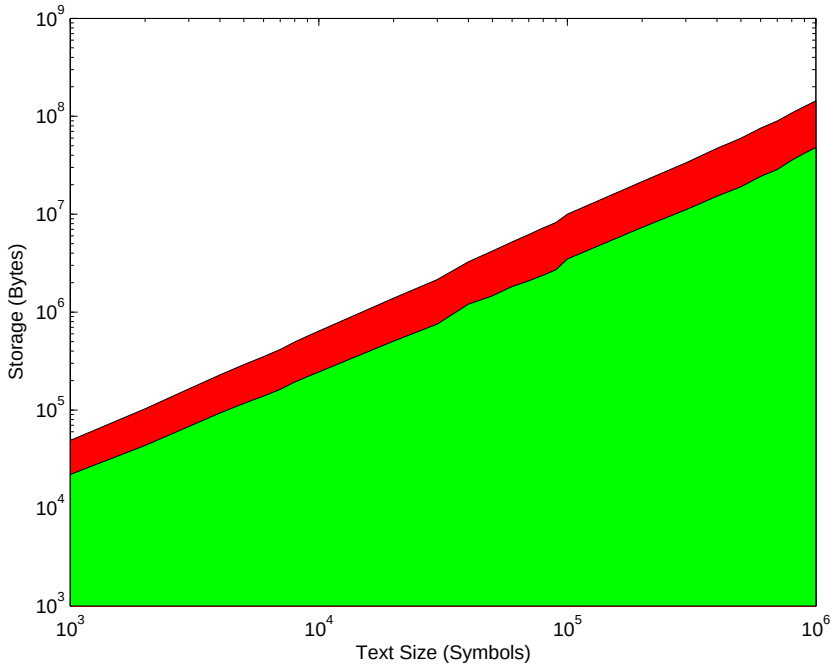


Figure 5.4: Storage requirements for the search index as a function of document length. The green area indicates the storage for the per-node data structure and the red area indicates the storage for the variable length pointer arrays.

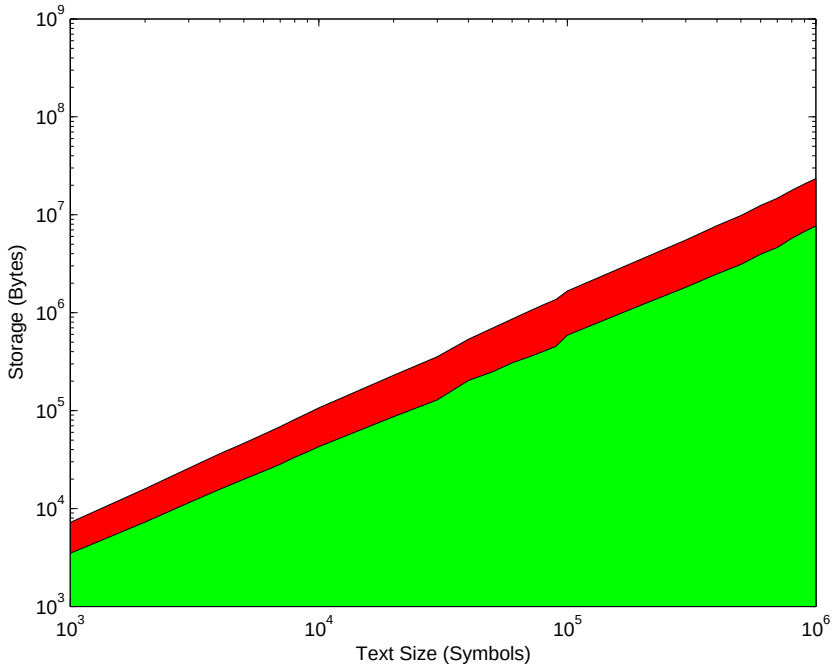


Figure 5.5: Storage requirements for the search index as a function of document length if only word prefixes are stored. The green area indicates the storage for the per-node data structure and the red area indicates the storage for the variable length pointer arrays.

substrings need be stored only if they are the prefix to words in the document. Performing the same calculation on this reduced set of strings yields significantly smaller requirements: a 10,000 byte document will produce an index of the order of 110,000 bytes and a 100,000 byte document of the order of 1.7×10^6 bytes. Storage requirements over a range of file sizes are shown in Figure 5.5.

5.2.3 Dynamic node allocation

It is possible to improve on the storage requirements by making a trade-off between pre-computation and computation at the time of Dasher node creation. To achieve this, rather than truncating the trie when the substring becomes unique, it is truncated at the point where the substring occurs no more than some small fixed number of times, $n_{\max}$. Each node in the Dasher trie must now be able to store either a pointer into the data trie, or a list of at most $n_{\max}$ pointers, each of which must be followed in the text when that node's children are created. Fortunately only Dasher nodes which are required to render the display are instantiated, so it is expected that the additional run time overhead will be low.

As an illustration, consider the case for $n_{\max} = 2$ on the data shown in Figure 5.2. In this case, the trie would be truncated below the node corresponding to h, which has 2 pointers into the data. To populate the children of the corresponding Dasher node, the text itself

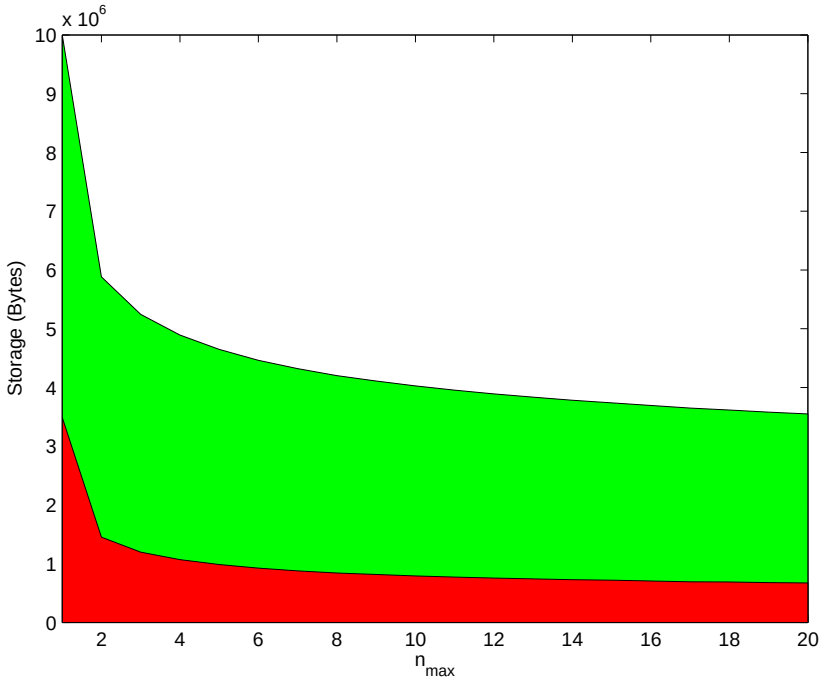


Figure 5.6: Storage requirements using dynamic node allocation as a function of $n_{\max}$ for a 100,000 byte text with full substring indexing. Storage for the trie itself is shown in green and that for the pointer lists is shown in red.

would be examined to find the possible continuations (both 'e' in this case). These pointers must both be maintained in order to populate the next level children, which then differ ('␣' and 'r').

Results for the possible storage savings based on experimental data are given in Figure 5.6 and numerically in Table 5.2. By setting $n_{\max} = 4$ it is possible to save more than 50% of the storage space, suggesting that this would give good performance in practice. Increasing $n_{\max}$ beyond this level results in diminishing returns, so the added computational cost will be less worthwhile.

5.2.4 Distributions over strings

The previous section assumed a uniform distribution over strings. It is of course, not necessary to make this assumption, and in many cases may be desirable to avoid it. In the simplest case, if the full text is indexed, it may be desirable to increase the probability of substrings which are word prefixes. In more structured documents, paragraph prefixes, headings and so on may also be more likely search candidates and can have their probability increased as appropriate. An example towards the more structured end of the scale is a computer source code editor, where it may be desirable to increase the probability of, for example, locations in the text corresponding to the definition of subroutines.

$n_{\max}$	**Storage Reduction**
1	0%
2	41%
3	48%
4	51%
5	54%
6	55%
7	57%
8	58%
9	59%
10	60%

Table 5.2: Percentage savings in storage requirements as a function of $n_{\max}$ for a 100,000 byte text with full substring indexing.

If each string is given a weight, corresponding to the un-normalised probability, this weight can be stored alongside each of the pointers. When populating children, the relative sizes of the boxes can be made proportional to the sums of the weights for all pointers in the corresponding child nodes. This will increase the storage size slightly, but depending on the resolution required it may only be necessary to add a few bits per entry.

5.3 Contact list lookup

One significant advantage of Dasher over many other text entry systems is that it requires only a pointing device or a small number of buttons, rather than a large keyboard. This property is particularly appropriate for text entry on small devices such as mobile telephones and personal digital assistants (PDAs). This section considers the task of selection from a database where each record is indexed by a short string. An example of an application involving this task might be the selection of a record from the contacts database stored on a telephone, where records are indexed by name.

A popular method of text entry on such devices is "predictive text entry" such as the T9 system [112]. In these systems, a small number of keys are mapped to the full alphabet, as well as numbers and some punctuation symbols. As the mapping is one-to-many, it is often necessary to disambiguate between multiple strings corresponding to the same key sequence. For free text entry this task is performed using a dictionary, possibly with some information about the relative frequencies of different words. Usually two additional keys are required for this task - one to cycle through the list of possible disambiguations and the other to make a selection. Disambiguation is often done on the level of words, so the select key can also carry an implied space. The disambiguation process is illustrated in Figure 5.7. In the selection task, rather than using a free dictionary, the contacts list itself can be used to perform disambiguation.

To analyse this method, we collected a list of the names of 460 members of the University

Key	Symbols
1	Punctuation, 1
2	A, B, C, 2
3	D, E, F, 3
4	G, H, I, 4
5	J, K, L, 5
6	M, N, O, 6
7	P, Q, R, S, 7
8	T, U, V, 8
9	W, X, Y, Z, 9
0	0

Figure 5.7: Left: Predictive text entry on the Sony Ericsson model k500i mobile telephone. The key sequence '739' has been entered, producing a list of possible disambiguations, as well as predictions for the complete word, in this case 'Text'. Right: Mappings from the numeric keypad to sets of symbols on this telephone. Mappings may vary between different models.

of Cambridge Physics Department. For a number of subsets of the list, representing various overall sizes, we calculated the minimum number of key presses required to unambiguously specify each name using T9. Names were stored as the first name followed by the second name, and alternative disambiguations were provided in dictionary order. Accented characters were replaced with their unaccented equivalents, and hyphens were replaced by spaces. In each case we assumed that one additional key press would be required to confirm the selection. We converted the results into times for entry assuming either 1 or 2 key presses per second (this range being obtained through informal observation of typical mobile telephone usage). For comparison, we also calculated the input time using Dasher assuming a uniform distribution over all contacts and a constant information entry rate of 4 bits per second (this figure represents the top end of the range of speeds obtained by users after one hour of experience with Dasher as reported in [96]) . A comparison of the results is shown graphically in Figure 5.8, which indicates that under these assumptions Dasher has the potential to allow significantly greater selection speeds.

An alternative viewpoint is to consider the potential information which could be entered using the same number of keypresses. Assuming a uniform distribution over presses of ten keys this is simply $\log_2(10)$ bits per key press. Multiplying by the mean number of keypresses gives the information which could potentially have been entered. This can be compared to the actual information expressed assuming a uniform distribution over contacts was also calculated. Results of this comparison are shown in Figure 5.9.

5.3.1 Model adaption

Traditional keyboard-based ambiguous text entry systems are essentially bound to a fixed encoding scheme. Although there is scope for some flexibility by varying the order in which possible disambiguations are presented, the mapping from keys to symbols remains fixed.

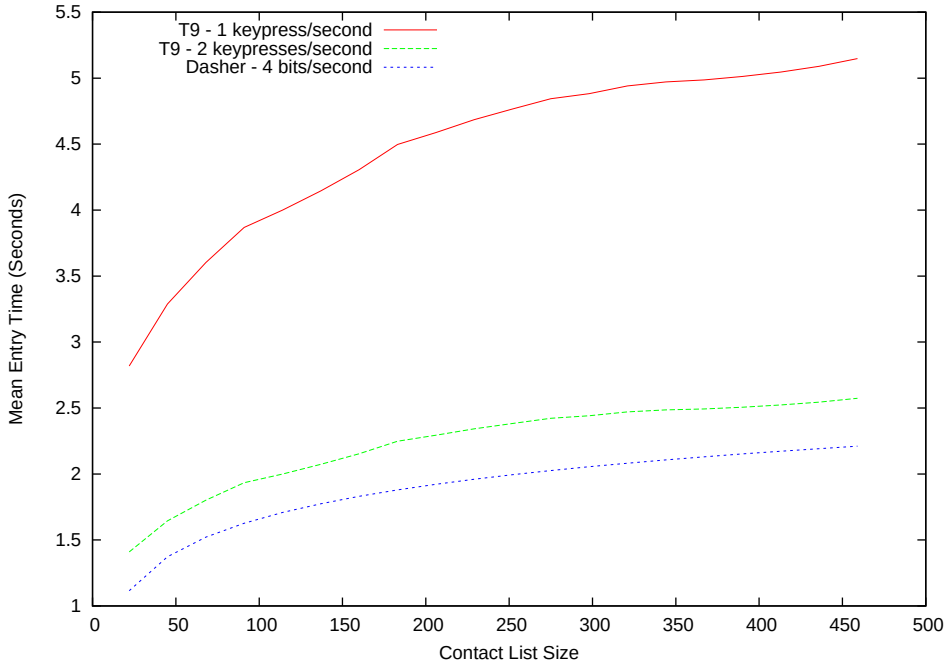


Figure 5.8: Average time required to unambiguously select entries from a contacts list. Values are plotted for contact lists sizes ranging from 22 to 460 entries. Rates are plotted assuming either 1 or 2 key presses per second for T9, or 4 bits per second for Dasher.

For Dasher however, the encoding scheme is dynamic, which allows for the possibility of adaptation. In particular, rather than assuming a uniform distribution over entries in the contacts list, any distribution may be used, for example based on the number of times that each entry has been retrieved in the past. Contacts which are accessed often would therefore be easier to recall. Adding a uniform Dirichlet prior with parameter α, this gives

$$P(x) = \frac{n(x) + \frac{\alpha}{q}}{N + \alpha} \tag{5.2}$$

where $n(x)$ is the number of times that entry x has been retrieved, $N = \sum_x n(x)$ is the total number of retrievals and q is the size of the contacts list. By associating a count with each node in the trie, predictions can be easily made using this model.

The net effect is that the entropy quoted in the previous section is an upper bound, and in practice it may be possible to significantly improve selection rates.

5.4 Further work

The analysis presented in this section has been theoretical in nature. The real utility of a novel user interface can only be determined by experiments conducted on human subjects using it for real tasks. It would therefore be appropriate to conduct such a test on the two interfaces described in this chapter. One quantity which would be revealing would be the

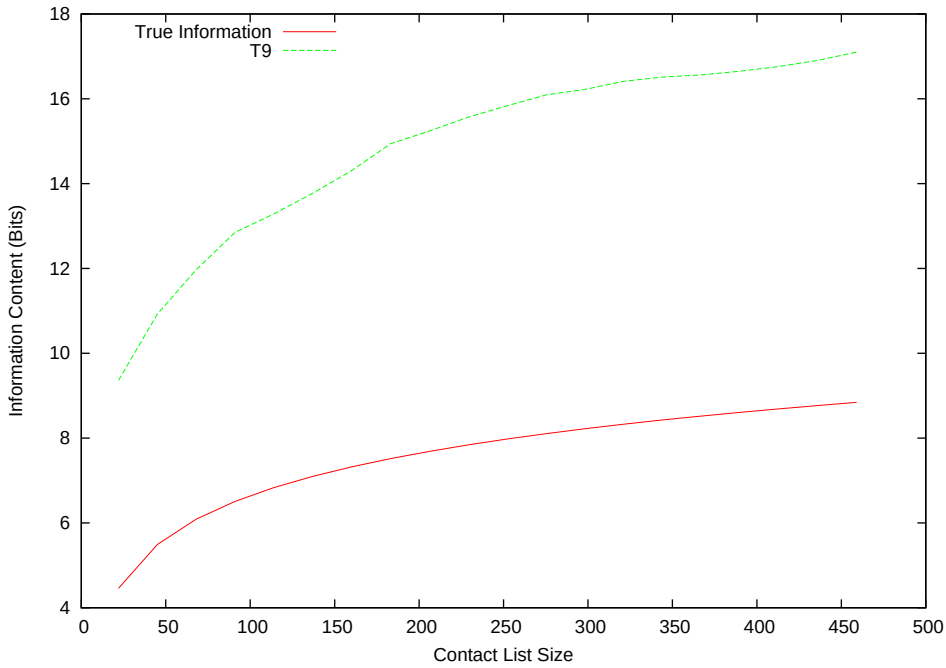


Figure 5.9: Average potential information content of keypresses required to select entries from the contact list, compared to the true information content assuming a uniform distribution over contacts. The potential information represents the number of bits which could be entered with the same number of keypresses were an optimal coding scheme to be used rather than T9.

time taken for users to perform tasks, such as looking up contacts using Dasher.

5.5 Conclusions

This chapter has presented an application for the Dasher user interface as a way of selecting items from an index, with the specific tasks of searching for substrings within a text document and selecting database records being considered. In the former case, a prototype application was developed to demonstrate the feasibility of the task, however, in its most basic form the method is quite demanding in terms of storage requirements. By making a trade-off between storage and computation at run time, it was shown that the storage requirements could be significantly reduced.

In the latter case, Dasher was compared to T9, a popular disambiguating text entry system used on mobile telephones. It was shown that Dasher provides a large improvement in terms of information efficiency, which can be further increased by using an adaptive interface whereby more probable entries become easier to select.