Optimising the lexical representation to improve A* lexical search

Nikko Ström

Abstract

The A algorithm is defined in a directed graph formalism. Pruning, path merging and modification of the algorithm to output word graphs rather than N-best lists are discussed. The concept of quotient graphs is utilised to improve the lexical graph. Results from experiments with the continuous speech recognition task of the WAXHOLM project are reported. The use of a particular quotient graph in the first pass of the A* algorithm is shown to speed up the search significantly without degrading the accuracy.*

Introduction

In addition to the acoustical analysis, lexical constraints are perhaps the most obvious knowledge source available for automatic speech recognition. Lately, much effort has been directed at incorporating lexical knowledge early in the recognition process (Zue et al. 1991, Schwartz et al. 1992, Murveit et al. 1993). This reduces the search space for other, possibly more computationally intensive components of the system, such as context-dependent acoustical analysis and syntactic analysis. However, to optimally combine different knowledge sources, “hard” decisions must not be taken early in the recognition process. This implies that the output of the lexical search component must be a representation of a collection of possible word strings. A popular representation is simply an N-best list of word strings (Schwartz et al. 1992, Zue et al. 1991). More compact representations are word graphs (Hetherington et al. 1993) or word lattices (Murveit et al. 1993). Other components may then analyse the word strings further and re-score them.

Searching for multiple word strings is a much more complex task than just searching for the best string. Several algorithms have been proposed, including the N-best algorithm (Schwartz & Chow 1990) and the A* algorithm (Soong & Huang 1991). This paper treats the application of the A* algorithm and how to use quotient graphs to improve the performance. The A* algorithm is a two-pass algorithm where the first pass is an exhaustive best-path Viterbi search. The first pass is usually where most of the computation is done. By replacing the original lexical representation by a quotient graph in the first pass, the computational effort can be considerably reduced (Kenny et al. 1991). In contrast to (Kenny et al. 1991), we construct quotient graphs that generate exactly the same language as the original lexical graph. Thus, the speed-up in the Viterbi pass is not traded against degraded performance of the second search pass.

The following two sections treat the A* algorithm in a graph formalism where both acoustic input data and the lexicon are represented as directed graphs. This is followed by a section on speeding up the search using quotient graphs and a concluding discussion.

Directed graph representation

In the present formulation, directed graphs are used to represent both the acoustical input data and the lexicon. Further, if word graphs are used to represent the output word strings, then all input and output data are represented as directed graphs.

Let us formally define a directed graph as the 6-tuple {**S**, $\mathbf{S}^i$, $\mathbf{S}^h$, **Σ**, **A**, **B**} where **S** is the set $\{s_i\}$ of nodes, $\mathbf{S}^i$ is the initial probabilities for all nodes and $\mathbf{S}^h$ halting probabilities for all nodes. **Σ** is the alphabet $\{\sigma_i\}$ including the no-output symbol ε. **A** is the set $\{a_{ij}\}$ of transition probabilities from node s_i to s_j. **B** is the set $\{b_{ijk}\}$ of output probability distributions for symbol σ_k and transition from node s_i to s_j.

Formally, the above definition of a directed graph is equivalent to a Hidden Markov Model (HMM), but to use the term "model" for the graph representing the input would not be appropriate. Therefore we use a more neutral term for all graphs. Note that unlike the terminology in much of the HMM literature, the output distributions are on the arcs.

An important characteristic of a directed graph is whether it is left-to-right, i. e. if there is some ordering of the nodes such that all arcs flow in the same direction (with the possible exception of self-loops). Dynamic programming techniques such as the Viterbi algorithm works only on left-to-right graphs. In continuous speech recognition applications, the graph representing the lexicon is not left-to-right because the words are connected. This property is most easily realised from the fact that the same word can occur several times in the same utterance. However, we will show that if we restrict the graphs representing the acoustical input to left-to-right graphs then there exists a graph, the product graph, that is left-to-right where the lexical search can be performed.

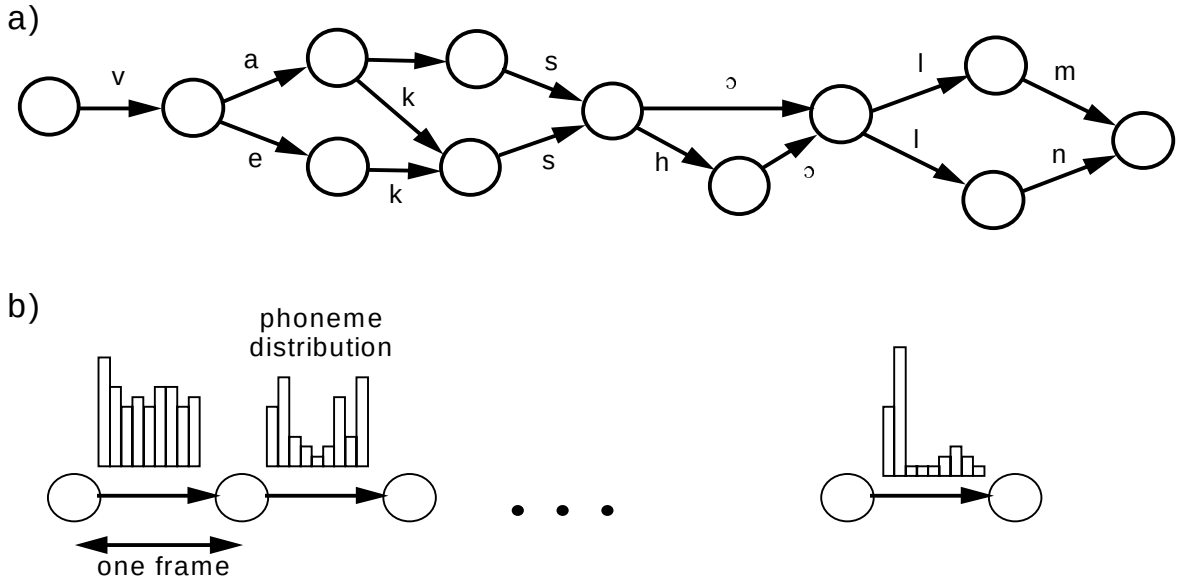

Figure 1. Two types of observation graphs.
a) Phonetic graph for the utterance of the Swedish word "Waxholm". The output distributions on the arcs have all their mass concentrated to one phoneme each.
b) Observation graph based on frame-wise acoustic analysis. The topology is very simple but each arc has an output phoneme distribution over all phonemes.

Observation graphs

An observation graph is a representation of the observation of an utterance at some level. The arcs represent speech units and the nodes represent the boundaries between units. Observation graphs are left-to-right.

Figure 1a) is a phonetic observation graph where the speech units on the arcs are phonemes. Figure 1b) shows an observation graph based on speech frames; the speech units are frames and the nodes are the time points between frames. Each arc has a discrete output distribution with one probability for each phoneme.

Figure 2 shows a word graph where the speech units are words. This structure is a possible type of output from a lexical search.

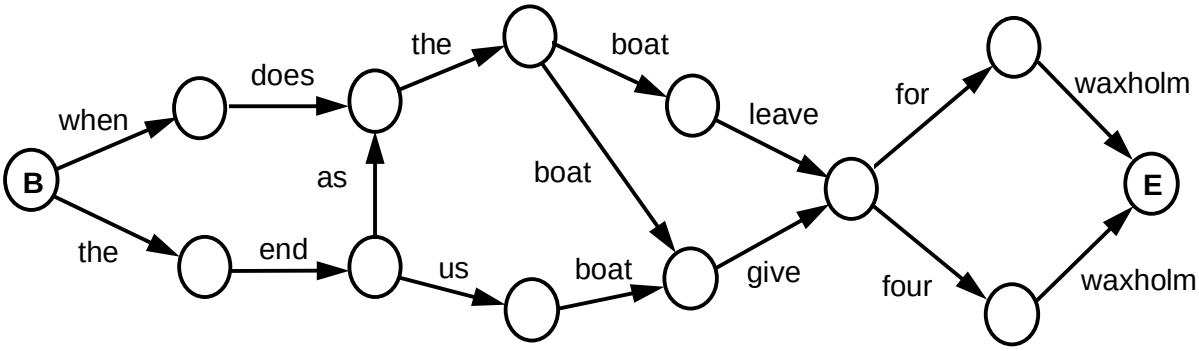

Figure 2. A word graph.

Model graphs

Model graphs represent a model of the language at some level. The lexical graphs treated in this paper model the words with their phoneme realisations and the connections between words. An arc represents either a phoneme or a word connection. Figure 3 shows the model of a single word in a lexical graph. The total lexical graph is the set of all word models and the word-connecting arcs. The word connecting arcs flow from word end nodes to word start nodes and have no output. As mentioned above, model graphs are generally not left-to-right.

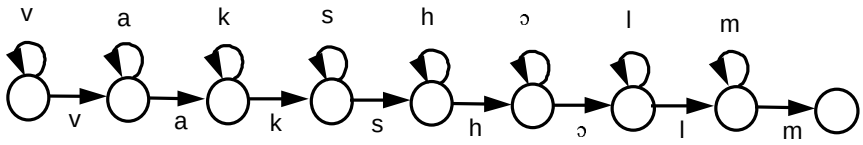

Figure 3. A word model. The self-loops are necessary if the observation graph is of the type of Figure 1b).

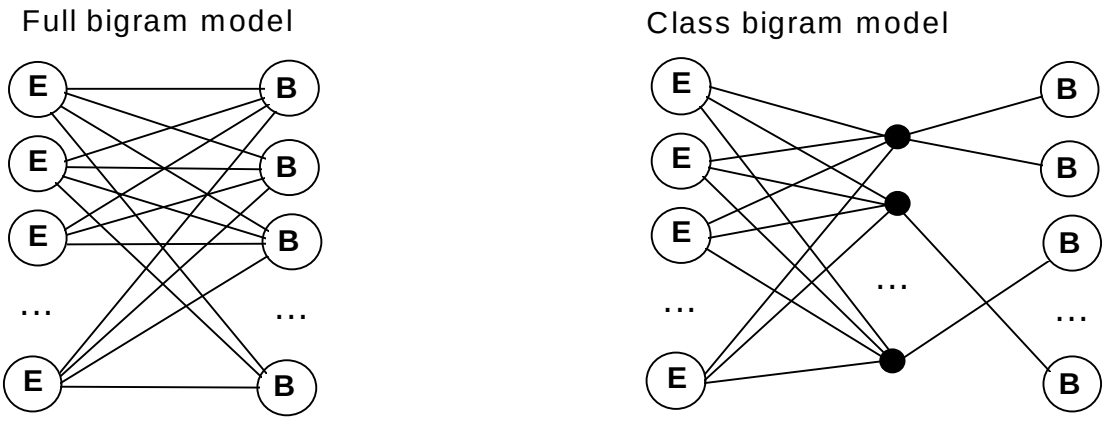

Figure 4. Full bigram model and class bigram model. The filled nodes are class nodes. All word end nodes (E) have word connecting arcs to all class nodes. The word-start nodes (B) have a word connecting arc only from its respective class node.

Class bigram grammar

The transition probabilities of the word connecting arcs in the lexical graph effectively implements a bigram grammar. If the lexicon is large, the number of word connecting arcs (proportional to the square of the number of words) becomes large and dominates the size of the graph. To overcome this, class nodes can be introduced. Class nodes have word connecting arcs flowing to one word class only, but word connecting arcs from all words flowing into them. Figure 4 is an illustration of this construction. The new number of word connecting arcs is linearly proportional to the number of words. Obviously, this constraint turns the grammar into a class bigram grammar.

The product graph

Lexical search is a parallel search for output strings in the lexical graph L and in an observation graph I. This seems to be more complicated than searching for paths in one graph only. In this section we introduce the product graph, a concept that reduces this more complicated problem to the simpler one-graph problem. The search matrix in Figure 5b is a visualisation of the product graph. The lexical graph and the observation graph are the factor graphs.

If the alphabet is the same in two graphs L and I we can define the product graph, $P = L \times I$, as follows. For each pair of nodes $s(L)_i$ and $s(I)_j$ in L and I, define a node $s(P)_{ij}$ in P. For each pair of nodes $s(P)_{ij}$ and $s(P)_{kl}$, define an arc in P from $s(P)_{ij}$ to $s(P)_{kl}$ if and only if, for some symbol σ_k, there exists both an arc from $s(L)_i$ to $s(L)_k$ in L and an arc from $s(I)_j$ to $s(I)_l$ in I, both with output probability greater than zero for σ_k. The output probability for the symbol σ_k and transition probability of the arc in P are the products of the respective probabilities of the arcs in L and I.

An extension is necessary to account for word connecting arcs in L. We introduce virtual self-loops with unit probability for ε and unit transition probability, on all nodes in I (this does not change the set of possible output strings from I or their probability).

The new ε-self-loops together with the word connecting arcs in *L* give the appropriate arcs in *P*.

From Figure 5a it is easy to see that any path in *P* uniquely defines one path in *L* and one path in *I*. Further, the observation probability of a path in *P* is the product of the observation probabilities of the corresponding paths in *L* and *I*. It is also obvious that if *I* is left-to-right, then *P* is also left-to-right (In Figure 5b all arcs flow from left to right).

From the above we get the useful result that if *L* is a lexical graph and *I* is an observation graph, then the output sequences from $P = L \times I$ are the same as the output sequences from *I* constrained by the model *L*. Also, *P* is left-to-right so we can apply the Viterbi algorithm. The two-dimensional array of nodes in *P*, ordered by the indices of the nodes in *L* and *I*, is often called the search matrix (see Figure 5b), and will play an important role in the next section.

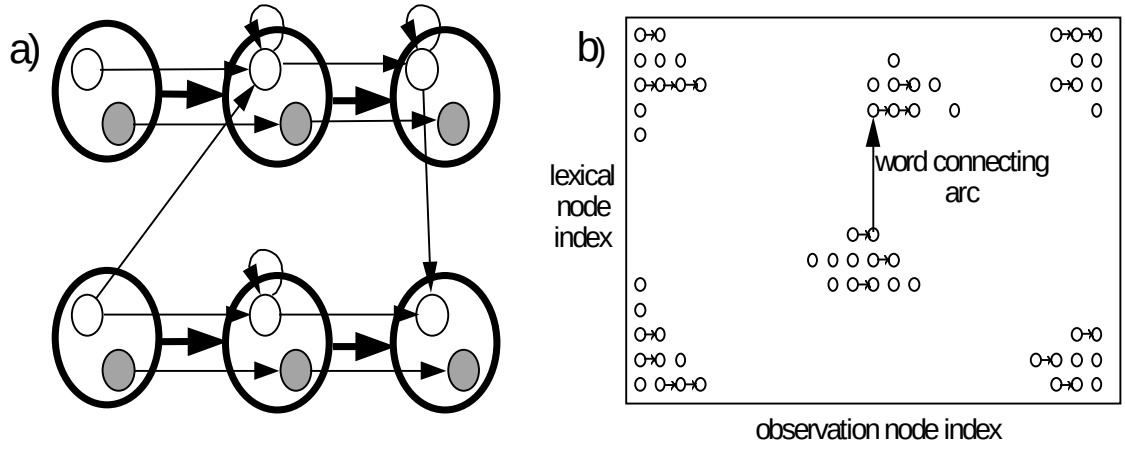

Figure 5.

a) Illustration of the construction of the product graph. The nodes and arcs of the product graph are drawn with thick lines and the two factor graphs are drawn with thin lines. There is an arc in the product graph iff there are arcs in both factor graphs between two nodes in the product graph.

b) The search matrix. The nodes of the product graph are arranged in a two dimensional grid where columns correspond to nodes constructed from the same observation node and rows correspond to the same lexical node. Some of the arcs and nodes and one word connecting arc are shown. Because the arcs of the observation graph are left-to-right, the arcs of the product graph will also be left-to-right regardless of the arcs in the model graph. The word-connections in the product graph can be justified by introducing self-loops in the input graph with the no-output symbol ε*.*

Path-searching in graphs

With the construction of the product graph, we have reduced the problem of lexical search to searching for paths in a directed left-to-right graph. Although the product graph is not explicitly constructed in an efficient implementation, we use this formulation here for notational convenience.

Viterbi best path search

Simply searching for the best path in a graph is efficiently implemented using dynamic programming (DP) – also known as the Viterbi algorithm. Let the nodes of a graph be ordered in left-to-right order (no arcs flow from one node to a node with a lower index). Also, assume that all arcs have all their probability distributions concentrated to only one output symbol. This simplifies the notation and is in fact the case for the product graph defined above. Now, the Viterbi algorithm can be described as follows:

i) Initialise a vector $\{h^*_i\}$, with one entry for each node, to the initial probabilities $\mathbf{S}^i$. Also initialise a back-trace vector $\{x_i\}$, of the same size.

ii) For each node s_i (in left-to-right order) do:
For each arc flowing into s_i find the value of $x_j + a_{ij}$, where s_j is the origin of the arc. Let h^*_i be the largest of the values and store the index of the arc giving this value in x_i. Thus, h* is the observation probability for the best path to s_i.

iii) Find the node *e* with the largest value of $s^h_i\, h^*_i$ – the observation probability for the best path ending in this node (recall that s^h_i is the halting probablitity for node i). Back-trace from *e* using the arcs in $\{x_i\}$ to get the best path.

As mentioned above, the best path only is not enough for our purposes, but the vector h* will be an essential ingredient of the following formulation of the A* algorithm.

Stack search

The Viterbi algorithm is polynomial in both time- and space complexity. Searching for multiple paths in a graph is a harder and in general an exponential problem. This observation forces us to choose a best-first strategy. Starting from the possible halting nodes of the graph, a search tree is constructed where the partial path with the

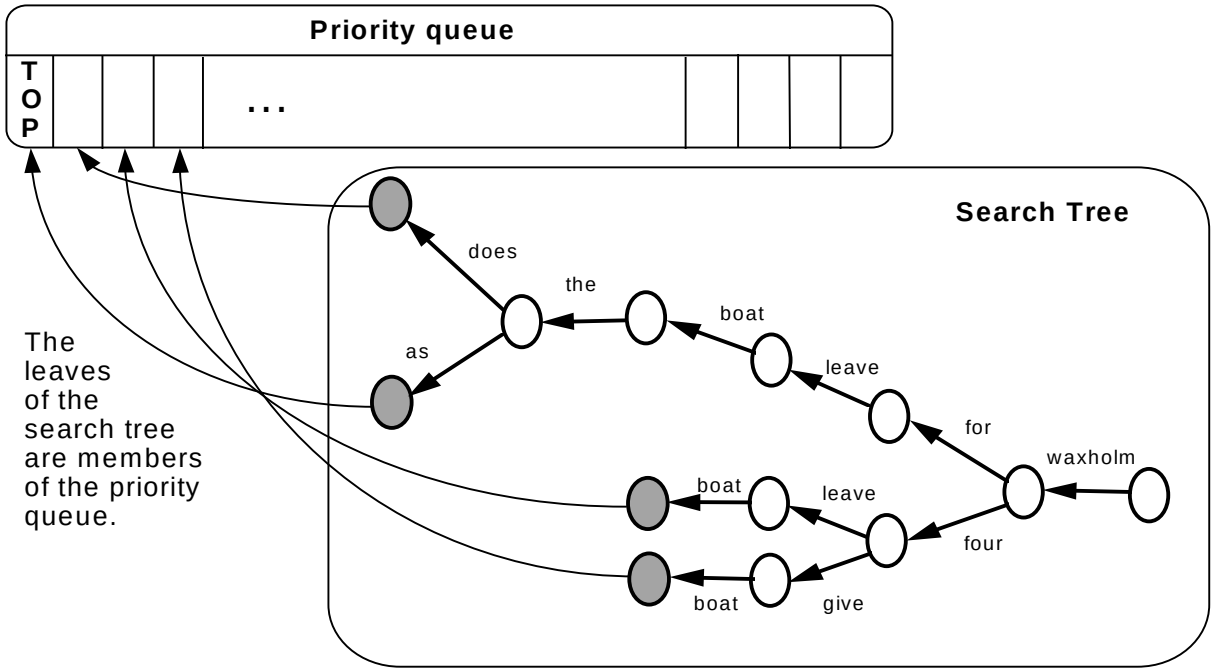

Figure 6. Stack search. A search tree is built by successively expanding the leave of the partial path with the highest observation probability.

currently best probability is always expanded first. When a suitable number, N, of paths have reached a starting node, the search is terminated. To keep track of the currently best path in the search tree, a priority queue (also known as a stack) is needed. The stack can be efficiently implemented using the heap data structure which requires only logarithmic time both for insertions and for retrieval of the best path (Sedgewick 1988). For obvious reasons this search is known as “Stack search”. Stack search is illustrated in Figure 6.

A serious drawback of the above algorithm emanates from the fact that any extension will lower the observation probability of the path. Therefore short paths will be expanded before long paths making the search resemble a breadth-first search that is too inefficient. A more sophisticated strategy to choose which path to expand is necessary.

The A* heuristic

To be able to compare partial paths of different lengths it is not enough to know the observation probability of the partial paths. A fair comparison would also include some estimation of the observation probability of the continuation of the paths. The success of an A* algorithm depends on this estimation called the A* heuristic.

Assume a partial path ending in a halting node and starting in some interior node s_i of the graph (this will always be the case since paths are built backwards). Assume also that the observation probability of the partial path is g. The best path from an initial node to s_i is h^*_i, where h* is computed in the Viterbi search described above. Now we can define an A* heuristic, $f^* = g + h^*_i$. f^* has the property that it is greater than or equal to any other complete path containing the partial path. This property is called A*-admissibility.

A*-admissible heuristics has the advantage that paths passing through a given node will be expanded in order of likelihood. A*-admissible heuristics guarantees that when a complete path is chosen for extension, it has a higher observation probability than any complete path that will be constructed later and it is safe to put it in the list of N-best hypotheses. Consequently, when N complete paths have been constructed, the search can be terminated.

Pruning

In continuous speech recognition tasks, the lexicon and therefore the size of the product graph is relatively large. To speed up the search procedure, pruning is often used.

In the Viterbi pass, pruning is relative to the best h* value at the respective time point (nodes in the same column of the search matrix have the same time). All nodes with an h^*_i below a threshold below the maximum h* are pruned and arcs from these nodes are not considered in the continued search.

In the stack search, pruning is relative to the score of the best path computed in the Viterbi pass. Partial paths with an A* heuristic below a threshold below the best score are not extended further.

Path merging

If the above search algorithm would be implemented, it would produce N-best lists with many word strings with the same wording but slightly different word alignment. If only the N best word strings are desired, a lot of computational effort can be saved by keeping information at each node about what word strings have been expanded from the node. Since paths are expanded in order, it is necessary to expand only one path with a particular word string from each node regardless of word alignment.

Further, if the object is to produce word graphs rather than N-best lists, even more paths can be merged.

Generation of word graphs

The word strings in the N-best list often differ only in one or a few words with the rest of the words identical. This makes a graph representation more efficient (see Figure 2). Generation of such a representation can easily be incorporated in the stack search (Hetherington et al. 1993). Each node in the search matrix is a potential node in the word graph. Each time a path is extended, an arc from the start node to the end node of the extension is created in the word graph.

Note that in this case it is necessary to expand only once from each node regardless of word string. In fact, this property improves the efficiency to the extent that even if an N-best list is the desired output, it is advantageous to first search for a word graph and then use a second A* search to find the N best word strings from the word graph.

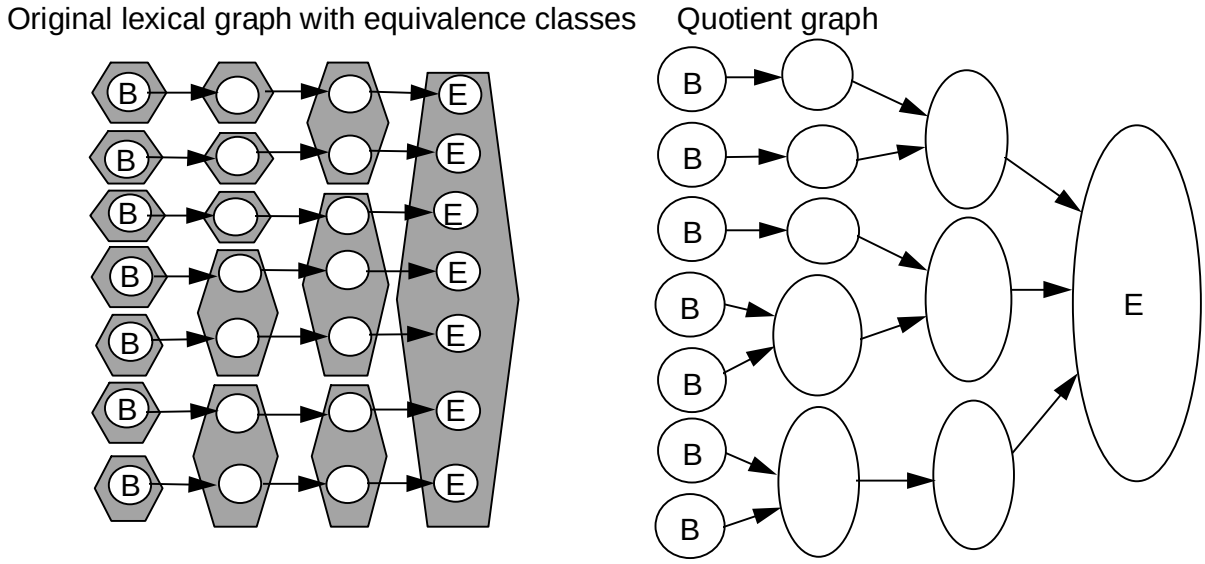

Figure 7. Construction of a lexical tree. To the left the original graph is shown with the shaded areas indicating equivalence classes of the nodes. Two nodes are equivalent iff the paths of arcs forward to the word-end node have the same symbols. To the right is the quotient graph with tree structure.

Quotient graphs

Definition

Given a graph L and an equivalence relation ~ on the nodes of L, we can construct a quotient graph $L^\bullet$ called the quotient of L by ~. Identify the nodes $\{n^\bullet_i\}$ of $L^\bullet$ with the equivalence classes of L by ~. Following the terminology of (Kenny et al. 1991), let (n_1, a, n_2) be a branch in L if n_1 and n_2 are nodes in L and a is an arc from n_1 to n_2. Let $(n^\bullet_1, a, n^\bullet_2)$ be a branch in $L^\bullet$ if (n_1, a, n_2) is a branch in L and $n^\bullet_1$ is the equivalence class of n_1 and $n^\bullet_2$ is the equivalence class of n_2. The output distribution of a is carried over to $L^\bullet$. In Figure 7 an example of equivalence classes and the construction of the quotient graph is given.

Lexicon tree

Instead of the straight-forward representation of each word as a string of phones (see Figure 3), the lexicon can be represented as a lexicon tree (Murveit et al. 1994). In a backward lexicon tree, words ending in the same phones share nodes and arcs in the graph. Note that partial paths are extended backwards in the stack search phase, and the word identity is always known at the word-start node where the extension terminates. As shown in Figure 7, the lexicon tree can be constructed as a quotient graph of the original lexical graph.

As pointed out by (Murveit et al. 1994) some modelling power is lost in the tree representation, because:

i) Triphone modelling will not be possible in all positions of the tree since the left phonetic context can be ambiguous. This problem will not be addressed here, but a full context dependent model can of course be used to re-score the hypotheses generated by the A* search.

ii) The word identity of the root node is ambiguous so the bigram grammar will be lost. However if a class bigram grammar is used, a lexical tree can be built for each word class making the word class identity for root nodes unambiguous. The division into class trees reduces the sizes of the equivalence classes. However, in many practical applications, the number of classes is small compared to the number of words and a few of the classes contain a large majority of the words. In these cases the reduction of the size of the graph will still be significant.

A lexicon from the WAXHOLM project (Blomberg et al. 1993) is used to give some quantitative figures. The lexicon has 643 words in 23 different word classes. The original word graph has 5 398 nodes and 4 755 phone arcs. The lexical graph with one tree for each word class has 3 424 nodes and 3 401 phone arcs. The word connecting arcs are reduced from 15 432 ($23 \times 643 + 643$) to 1 172 ($23 \times 23 + 643$).

If no pruning is used the speed-up in the Viterbi search is proportional to the graph-size reduction. Informal tests indicate that this relation holds approximately also with pruning. The stack search phase is also faster with the lexicon tree representation, but not with a simple relation to graph size.

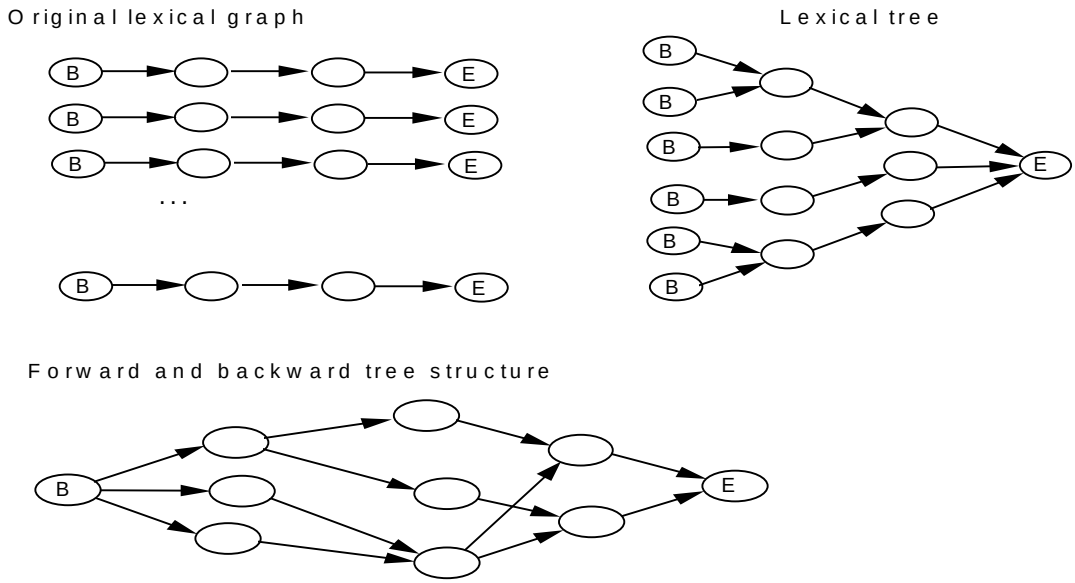

Figure 8. Different lexical representations. The bottom graph can be constructed as a quotient graph to the lexical tree in the same way as the lexical tree was constructed from the original graph. In the two top graphs there is no problem with word identity since each word has a unique starting node (B) but this information is lost in the bottom graph.

A quotient graph for h* computation

The lexical tree generates exactly the same language as the original graph. It is tempting to merge nodes in the same way also at the beginning of the words (see Figure 8, bottom). This can be done without changing the language generated by the graph but the word identities are lost. However, the word identities are not used in the Viterbi search and the equivalence classes in the quotient graph definition can be a bridge from the reduced graph to a larger graph used in the stack search where the word identities are necessary.

Assume L is a lexical tree representation and $L^\bullet$ is its quotient graph reduced both forwards and backwards. The Viterbi search is performed on $L^\bullet$. Then the stack search is performed on L and the h* estimate for node n is equal to h* for node $n^\bullet$ in $L^\bullet$, where $n^\bullet$ is the equivalence class of n. As pointed out by (Kenny et al. 1991), any quotient graph will give an A*-admissible heuristic, but this particular quotient graph reduces the size of the Viterbi search without degrading the performance of the stack search phase.

Again the WAXHOLM project lexicon serves as an example. The number of nodes is now reduced from 5 398 to 2 028 and the number of phone arcs is reduced from 4 755 to 2 627. The word connecting arcs are reduced from 15 432 to 529 (23×23). Informal tests have indicated that the size reduction makes the algorithm run at least two times faster.

Summary and discussion

Table I illustrates the graph size reductions for the different lexical representations. The final quotient graph used for the Viterbi search has roughly half as many arcs and nodes as the original. The number of word connections is reduced even more. This reduction reduces the search time to less than half of that of the original system. The lexicon used is this paper is relatively small (643 words). A larger lexicon would be reduced even more by using quotient graphs. This is because the number of words relative to the number of phonemes and word classes would increase, resulting in more nodes and arcs that could be shared.

The gained efficiency is achieved by reducing the modelling power of the lexical graph. The ability to model context dependencies is reduced and the bigram grammar has to be reduced to a class bigram grammar with relatively few classes. The trade-off between modelling power and search speed is intuitively clear and quotient graphs are a general mechanism to control and utilise it. By selecting an appropriate equivalence relation, the desired trade-of can be achieved.

Table I. Graph sizes for the different lexical representations (compare Figure 8).

	Type of object		
	Nodes	Phone arcs	Word connecting arcs
Original lexical graph	5 398	4 755	15 432
Lexical tree	3 424	3 401	1 172
Forward and backward tree structure	2 028	2 627	529

Acknowledgements

I would like to thank Michael Phillips for sharing his experience and initiating my motivation for this work. The author is supported by a donation from VOLVO AB.

References

Blomberg M., Carlson R., Elenius K., Granström B., Gustafson J., Hunnicut S., Lindell R., & Neovius L. (1993): "An Experimental Dialogue System: Waxholm," *Proc EUROSPEECH '93*, pp. 1867-1870.

Hetherington I. Lee, Phillips Michael S., Glass James R. & Zue Victor W. (1993): “A* Word Network Search For Continuous Speech Recognition,” *Proc IEEE ICASSP '93*, pp. 1533-1536.

Kenny P., Hollan V., Gupta V., Lennig M., Mermelstiein P. & O’Shaughnessy D. (1991): “A* - Admissible Heuristics For Rapid Lexical Access,” *Proc IEEE ICASSP '91*, pp. 689-692.

Murveit H., Butzberger J., Digalakis V. & Wientraub M. (1993): “Large-Vocabulary Dictation Using SRI’s Decipher™ Speech Recognition System: Progressive Seach Techniques,” *Proc IEEE ICASSP '93*, pp. II-319 - II-322.

Murveit H., Monaco P., Digalakis V. & Butzberger J. (1994): “Techniques To Achieve An Accurate Real-Time Large Vocabulary Speech Recognition System,” *Proc ARPA Workshop on Human Language Technology*, March ’94, Plainsboro, NJ, pp369-373.

Nilsson N. (1971): *Problem-Solving Methods in Artificial Intelligence*, Morgan Kaufman Publishers, Inc., ISBN 0-934613-10-9.

Sedgewick Robert (1988): *Algorithms*, 2nd edition, Addison-Wesley Publishing Company Inc., ISBN 0-201-06673-4.

Schwartz Richard & Chow Yen-Lu (1990): “The N-best Algorithm: An Efficient And Exact Procedure For Finding The Most Likely Sentence Hypotheses,” *Proc IEEE ICASSP '90*, pp. 81-84.

Schwartz R., Austin S., Kubala F., Makhoul J., Nguyen L., Placeway P. & Zavaliagkos G. (1992): “New Uses For The N-best Sentence Hypotheses Within The Byblos Speech Recognition System,” *Proc IEEE ICASSP '92*, **Vol I,** pp. 1-4

Soong F. & Huang E. (1991): “A Tree-Trellis Based Fast Search for Finding the N Best Sentence Hypotheses in Continuous Speech Recogniton,” *Proc IEEE ICASSP '91*, pp. 705-708.

Zue V., Glass J., Goodine D., Leung H., Phillips M., Polifroni J. & Seneff S. (1991): "Integration Of Speech Recognition And Natural Language Processing In The MIT Voyager System," *Proc IEEE ICASSP '91*, pp. 713-716.