Procesamiento del Lenguaje Natural, núm. 43 (2009), pp. 57-64 recibido 1-05-2009; aceptado 5-06-2009

Fast approximate string matching with finite automata

Rapida búsqueda aproximada con autómatas de estado finito

Mans Hulden
University of Arizona
mhulden@email.arizona.edu

Resumen: En este artículo se presenta un algoritmo eficiente para dada una cadena de caracteres extraer las cadenas más cercanas de un autómata de estado finitos según alguna métrica de distancia. El algoritmo puede ser adaptado con el fin de beneficiarse de una variedad de métricas para determinar la similitud entre palabras.
Palabras clave: búsqueda aproximada, autómata

Abstract: We present a fast algorithm for finding approximate matches of a string in a finite-state automaton, given some metric of similarity. The algorithm can be adapted to use a variety of metrics for determining the distance between two words.
Keywords: approximate search, finite automata

1 *Introduction*

In this paper we shall present a promising approach to a classic search problem: given a single word w and a large set of words W, quickly deciding which of the words in W most closely resembles w, measured by some metric of similarity, such as minimum edit distance.

Performing this type of search quickly is important for many applications, not least in natural language processing. Spelling correctors, Optical Character Recognition applications and syntactic parsers, among others, all rely on quick approximate matching of some string against a pattern of strings.

The standard edit distance algorithm given in most textbooks on the subject does not scale well to large search problems. Certainly, calculating the edit distance between two individual words can be done quickly, even with different costs for different character substitutions and insertions. The ubiquitous textbook algorithm for finding the minimum edit distance (MED) between two words based on the dynamic programming method takes quadratic time in the length of the longer word.

However, finding the closest match between an input word w and a list of, say 1,000,000 words, is a much more demanding task. The strategy of calculating the edit distance between every word on the list and the input word is likely to take too much time.

Naturally, if we perform the search by calculating the edit distance between each word on a list and our target word w, there are a number of possible optimizations that can be made, such as aborting the calculation for a word-pair as soon as the MED comparison exceeds the lowest MED encountered so far. Even so, the search space remains too large for this strategy to be of practical use.

1.1 A more general problem

In addressing this problem, we shall investigate a much more general problem: that of finding the closest approximate match between a word w and words encoded in finite-state automaton $\mathcal{A}$. The problem is more general because a finite automaton can not only encode a finite number of words (as a deterministic acyclic automaton), but an infinite number of words, or a 'pattern.'

Considering the problem of finding an approximate match to word w against some automaton $\mathcal{A}$ instead of a word list L has many advantages. First, a finite word list can always be converted into an acyclic deterministic finite automaton. This means that a good solution to the problem of matching against an automaton is also a good solution to the word list case. Second, a finite automaton can represent an infinite number of strings—i.e. a pattern of strings—by virtue of the

ISSN: 1135-5948 © 2009 Sociedad Española para el Procesamiento del Lenguaje Natural

fact that it can contain cycles. Some natural languages, for instance, allow very long compound words, the patterns of which can be compactly modeled as a cyclic automaton. Third: natural language morphological analyzers are often implemented as finite-state transducers (Beesley and Karttunen, 2003). Creating a deterministic minimal finite-state automaton by extracting the domain or range from a finite-state transducer is trivial: one simply ignores the input or the output labels, and determinizes and minimizes the resulting automaton. This means, for instance, that if one has access to a morphology of a language represented as a transducer, that morphology can easily be used as a spelling corrector.

2 Solution based on informed search

Most previous solutions to this problem have been based on simple brute-force breadth or depth-first search through the automaton $\mathcal{A}$, comparing paths in the automaton with the word at hand and aborting searches on a path as soon as the required number of changes in the word at a point in the search reaches some specified cutoff.[1]

Our observation, however, has been that finite-state automata contain useful information that can be extracted cheaply and used profitably as a reliable guide in the search process, avoiding a brute-force strategy and the exponential growth of the search complexity it entails. In particular, as we shall show, one piece of information that can be extracted from an automaton with little effort is knowledge about what kinds of symbols can be encountered in the future. That is, for each state in an automaton $\mathcal{A}$, we can extract information about all the possible future symbols that *can* be encountered for the next n steps, given any n.

2.1 A*-search

When performing searches with some type of additional information to guide the search process, the preferred family of algorithms to use is usually some variant of the A^*-algorithm (Hart, Nilsson, and Raphael, 1968). This was an obvious choice for us as well. The additional information we use—the possible symbols that can be seen in future paths that extend out from some state in the automaton—can function as a guess about profitable paths to explore, and thus fits well in the A^*-paradigm.

The A^*-algorithm essentially requires that we have access to two types of costs during our search: a cost of the path in a partial exploration (g), and a guess about the future cost (h), which may not be an overestimate for the heuristic to be admissible. At every step of choosing which partial path to expand next, we take into account the combined actual cost so far (g), and the guess (h), yielding $f = g + h$.

Searching an automaton for matches against a word with edit distance naturally yields g, which is the number of changes made so far in reaching a state s in an automaton comparing against the word w. The guess h is based on the heuristic we already briefly introduced.

2.2 The search algorithm

For our initial experiments, we have considered the restricted problem of finding the shortest Levenshtein distance between w and paths in an automaton $\mathcal{A}$. This is the case of MED where insertion, substitution, and deletion all cost 1 unit. However, the algorithm can easily be extended to capture varying insertion, substitution, and deletion costs through so-called confusion matrices, and even to context-dependent costs.

Essentially, the search begins with a single node represented by the start state and the word at position 0 (the starting position). We then consider each possible edge in the automaton from the state v. If the edge matches the current word position symbol, we create a new node by advancing to the target state v', advancing *pos* (the word position counter *pos*) by 1, recalculating the costs $f = g+h$ and storing this as a new node to the agenda marked as x:x (which indicates no change was made). We also consider the cases where we insert a symbol (0:x), delete a symbol (x:0), and (if the edge currently inspected does not match the symbol in the word at position *pos*) substituting a symbol (x:y) with another one. When we are done with the node, we find the node with the lowest score so far, expand that node, and keep going until we find a solution. See figure 1 for a partially expanded search of the word

[1] The most prominent ones given in the literature are Oflazer (1996) who presents a depth-first search algorithm, and Schulz and Mihov (2002), which is essentially the same algorithm.

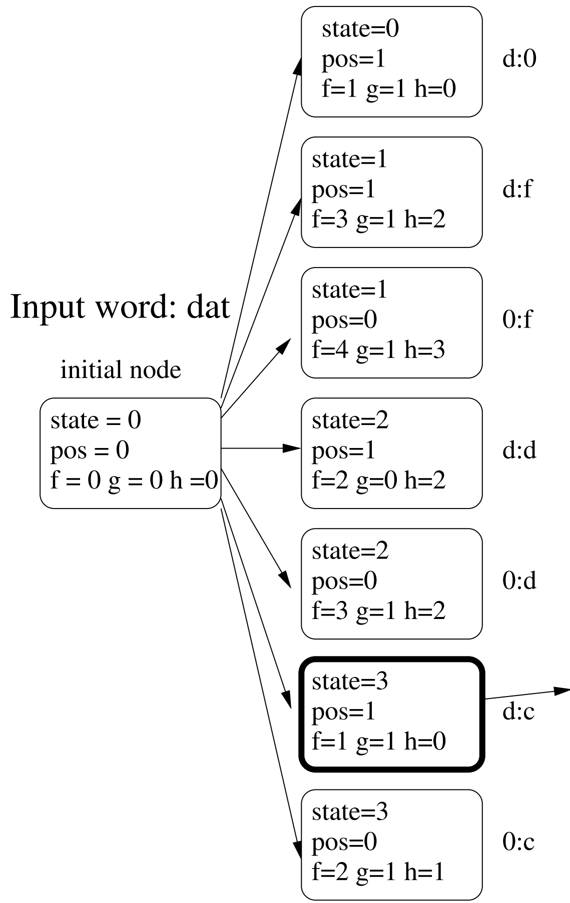

Figure 1: *Initial steps in searching for the approximate match against the word **dat** and the automaton depicted in figure 3. The node with the outgoing arrow will be expanded next. To reach that node from the initial node, we moved from state 0 to state 3 in the automaton, and thus had to substitute a **d** in the input for a **c** in the automaton, costing one unit, reflected in the g-value. The h value for that node is 0 since all the subsequent letters in the remainder of the word (**a** and **t** are in the state's symbol table. For this illustration, we assume the heuristic h(∞), seen in figure 3.*

dat against the automata in figure 3.

3 The heuristics

There are two obvious criteria for creating a useful heuristic h for the approximate search through a graph: the heuristic must be fast to calculate, either beforehand or on the fly, and, if calculated prior to the search, it must be compact—i.e. take up a linear amount of space proportional to the number of states in $\mathcal{A}$ we want to search.

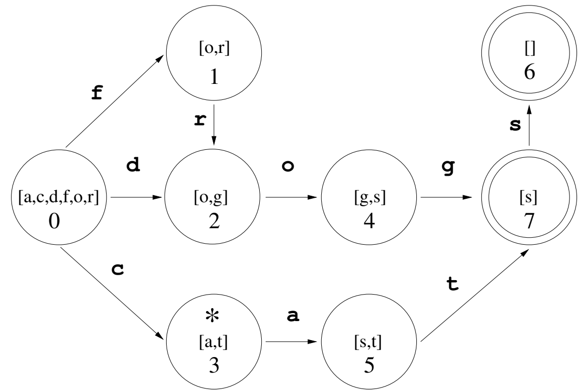

Figure 2: *An automaton where information about the possible symbols of future paths of length n (for the case $n = 2$) is stored in each state.*

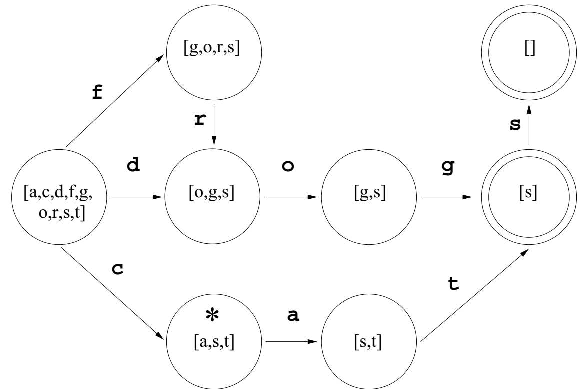

Figure 3: *An automaton where information about the possible symbols of future paths of length n (for the case $n = \infty$) is stored in each state. Calculating this for the set of states is accomplished in linear time by algorithm 2.1.*

As already mentioned, we have chosen as our heuristic function to store, for each state, a list of symbols that can be seen somewhere along a path of length n starting from that state. Figures 2 and 3 illustrate this by two acyclic finite automata that encode a number of words, and where every state has been marked with information about all possible future symbols n steps ahead, where n is 2 (figure 2), and ∞ (figure 3).

During the search of the graph this list is consulted and compared against the symbols in the current word we have yet to match. The discrepancies between the symbols in the

state and the symbols yet to be matched in the current word is used as our cost heuristic. That is, for each of a maximum of n symbols remaining to be matched in the word w at the current position of the search, for any symbol not found stored in the current state, we add a cost of 1 to the heuristic function h. This reflects an estimate (which is correct) that some time in the future any symbols not present in the path we are exploring will have to be either considered substitutions or deletions with respect to the input word.

For example, suppose we were matching the word *cagr* at position 1, i.e. the symbols remaining to be matched are *agr*, and suppose we are in the state marked with an asterisk in figures 2 and 3. Now using our heuristic function where n is 2 (figure 2), gives us an estimated h-value of 1, since the two following symbols to be matched are a and g, and the state marked with an asterisk does not contain g. For the same position, using the heuristic where n is ∞ (figure 3), the h-value becomes 2, since neither g nor r are found in subsequent paths, as is seen from the list of symbols stored in the state.

3.1 Consistency of h

It is easy to see that h for any n is an admissible heuristic for A^*-search in that it never overestimates the cost to reach the goal.[2] Certainly if there is a discrepancy of i symbols between future paths and the remainder of the word for some number of steps, those symbols will have to be produced by insertion or substitution (each of which cost 1 in our model), and hence h cannot overestimate the cost to the goal.[3]

This means that goals in the search will be found in order of cost, which is of course desirable, since we know that the first solution we find is the shortest one. Naturally, we can keep exploring and find more solutions if desired in increasing order of cost. For a spell checking application, for instance, we would keep the search going until either we reach some cutoff where the cost has gotten too high, or we find some desired number of approximate matches to suggest as corrected spellings.

[2] See e.g. Pearl (1984) for extensive discussions about what kinds of heuristics are suitable for A^*.

[3] The estimate h is also *consistent* in that it fulfils the triangle equality $h(p) \leq \text{cost}(p, a, p') + h(p')$, i.e. the estimated cost to the goal is always smaller than the combined actual cost to any point p' reachable from p and the new estimated cost from p' to the goal.

4 Precalculating h

For any serious application we will want to precalculate, for each state, the symbols that can be subsequently matched along some path of length n.

For any finite n this is a straightforward task: for each state we simply perform a depth-first search with a depth threshold of n states, and store all symbols we encounter on an edge in the state we started the search from. This procedure is given in Algorithm 2.2.

The case where $n = \infty$ is more difficult. Here we want to know, for each state, all the symbols that can possibly be encountered in the future, no matter how long the path. If we knew that the automaton we are dealing with were acyclic, this could be solved—although at a cost of some computation time—by the same algorithm as for finite n by limiting searches to the number of states in $\mathcal{A}$. Since we do not want to limit ourselves to searching acyclic graphs only, we have developed a separate algorithm for marking future symbols on the states in the case where $n = \infty$.

4.1 The special case n=∞

The solution for this case is based on the observation that we can divide an automaton into its strongly connected components—groups of states where each state has a path to each other in the group. Naturally, all states in the same strongly connected component (SCC) share the same future symbols for $n = \infty$.

Interestingly, we do not need to split the graph into its SCCs first, and then mark the states. We can in fact do both at the same time—calculate the SCC of the graph, and mark each state with the possible symbols that can follow. We do so by an adaptation of the well-known algorithm by Tarjan (1972) for dividing a graph into SCCs. The core of Tarjan's original algorithm is to perform, by recursion, a depth-first search (DFS) on the graph, while keeping a separate stack to store the vertices of the search performed so far. In addition, each vertex is given an index (low) to mark when it was discovered.[4]

[4] A very thorough analysis of this remarkably simple algorithm is given in Aho, Hopcroft, and Ullman

We now turn to the question of adapting this algorithm to mark each state with all its possible future symbols, presented in Algorithm 2.1. The crux to the marking the states at the same time as performing the SCC DFS is that a) for each edge between v and v' discovered, the parent vertex v inherits the symbols at v' as well as the edge symbol $e.sym$ used to get from v to v', and b) after we encounter the root of a SCC, we copy the properties of the root vertex v to all the child vertices v'.

5 *Choosing* n

Having now various possible n at our disposal to use as the guiding heuristic in the search of solutions, the question about which n, or which combinations of n to use as a heuristic is largely an empirical one. Our tests indicate that using a combination of $n = 2$ and $n = \infty$ is far superior to any other combination of n-values. When used in combination, the algorithm always chooses the larger of the two guesses, $\textsc{Max}(h(2), h(\infty))$, as the h-value.

Naturally, using two heuristics requires two symbol vectors to be stored for each state, where each vector takes up $|\Sigma|$ bits of space, for a sum total of $2|\Sigma||\mathcal{A}|$ bits, where $|\mathcal{A}|$ is the number of states in the automaton and $|\Sigma|$ the alphabet size of the automaton. Given that word lists can be compressed into a very small number of states, the combined h is probably the best choice even though it takes up twice as much storage as a single heuristic. If a single heuristic is needed because of space concerns, our results indicate that $h(\infty)$ is the best overall choice, although for applications where a small number of errors are expected, such as spelling correction, $n = 2$ or $n = 3$ may be slightly better.

Table 1 shows the number of nodes expanded as well as the number of nodes inserted on the agenda of the search graph (by Algorithm 2.3) for 100 randomly chosen misspelled words (all with shortest edit distance 3 or less), using various combinations of h, against a dictionary of 556,368 words.

We used these results (because they most likely reflect average use of the algorithm) to settle for using the combination of $\textsc{Max}(h(2), h(\infty))$ as our best heuristic h.

(1974).

	h_0	h_1	h_2	h_3	h_4	h_5
NI	6092	1892	1548	1772	1904	622
NE	3295	1143	909	1049	1193	89

Table 1: *Average number of nodes inserted on the agenda (NI) and nodes expanded (NE) when performing a search of 100 randomly perturbed words against a dictionary of Spanish containing 556,368 words, encoded as a finite-state automaton. Each search terminated after finding the 5 closest matches. The average length of the words was 6.6 letters and the average edit distance to each match was 2.18. The different heuristics used were* h_0*: no heuristic, i.e.* h *is always zero;* h_1*: only consider* $n = \infty$*;* h_2*: only consider* $n = 2$*;* h_3*: only consider* $n = 3$*;* h_4*, only consider* $n = 4$*, and* h_5*: the heuristic was* $max(n = 2, n = \infty)$ *and ties were resolved in the priority queue by giving priority to the highest word position. The priority queue strategy for all except* h_5 *was purposely adversarial to the search: i.e. nodes expanded were extracted LILO in the case of* f*-score ties. Note that the nodes expanded measure is much more important since the time to insert a node in a priority queue is generally constant, while the time taken to extract the minimum is on the order of* $log(n)$*, with* n *being the number of elements in the queue.*

5.1 Priority queue strategy

The number of nodes explored before finding solutions is affected not only by the particular heuristic used, but also by the choice of tiebreaking in the search strategy. In many cases, we will find unexpanded fringe nodes in the priority queue with the exact same f-value. How to choose a node for expansion in the case of ties has great bearing on the speed of the search as well, as is seen in table 1. In our experiments, the strategy of breaking ties so that the node that has advanced farthest in the word to be matched is given priority has proven to be far superior to any of the tiebreaking strategies we explored.

The reason for this seems natural: if we are exploring possible approximate matches, then, other things being equal (i.e. the f-score), one should follow the path that has moved the longest toward the end of the word.

```
foma[0]: regex Spanish2;
47432 states, 126535 arcs, 556368 paths.
foma[1]: apply med
apply med> wuighuiwrwegfwfw

-sig-uiér-e--mos
wuighuiwrwegfwfw
Cost[f]: 10

elapsed time: 0.100000s
Nodes inserted: 56293
Nodes expanded: 20549
apply med>
```

Figure 4: *Example search against a dictionary in our implementation.*

6 Results

We have implemented the algorithms described as an extension to the freely available finite state toolkit, *foma* (Hulden, 2009). The algorithms were written in C. We tested them against various automata, including a lexicon of English (represented as a cyclic finite-state automaton), and several acyclic lexicons of Spanish of between 500,000 and 1,000,000 words. The timing tests in figure 5 were achieved by generating 100 random words of each minimum edit distance 1 through 10 from one of our lexica, and then searching for the closest match with the algorithm. Although discovery of closest matches of edit distance more than 4 are probably not useful for spelling correction applications, other applications (such as Optical Character Recognition) may need to find closest matches that take into account far more perturbances than spelling correction applications need. For instance, figure 4 depicts our interface in the search of a word that results in a match of edit distance 10, where clearly the match would not be useful for spelling correction since the input word is mangled beyond recognition from its closest match in the dictionary.

7 Discussion and further work

The main result that the algorithm points to is that an informed search strategy such as we have outlined above is clearly superior to any uninformed strategy when searching word graphs such as finite automata for approximate matches. In particular, the heuristic of the strategy we have used is fast to compute ahead of time—taking only linear

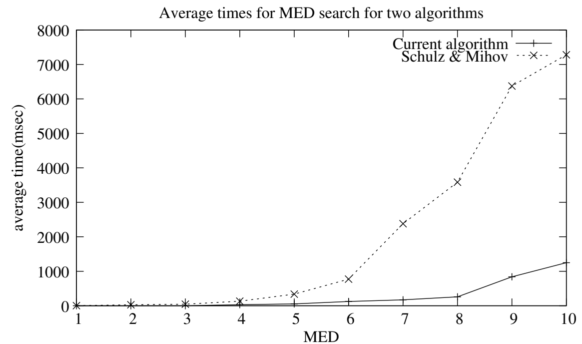

Figure 5: *Average wall clock times in our implementation of the new algorithm for finding minimum edit distances for 1000 words of edit distance, compared to running a Schulz and Mihov (2002) search on the same words. The 1000 words consisted of 100 words of each edit distance, i.e. 100 words in each group 1 through 10. The automaton which the randomly generated words were matched against was the FreeLing Spanish dictionary of 556,368 words which we first converted into a deterministic acyclic automaton.*

time and space in the size of the automaton—and often results in a dramatic reduction of the search space that needs to be explored in order to find approximate matches. For applications such as spelling correction, the method is very space and time-efficient, since one can encode large finite lists of words quite compactly as a finite automaton. Finding matches for minimum edit distance 5 in all our experiments against large dictionaries never took more than 70 milliseconds. For edit distances 1–3, the granularity of the system clock (one millisecond) was such that the timing results were always 0. As such, our implementation is already practically usable for many applications.

It also appears that the new strategy scales well to much larger problems. The seemingly exponential growth in the search time seen toward the end of the graph in figure 5 is probably partly an artifact resulting from the large number of errors in the word and the fact that most of the words in the automaton we matched against were much shorter than the input words were. That is, since almost every character in the words we tested that were of MED 9 or 10 was an error, there seems to be no way to avoid exploring a very large part of the entire search space. However, in preliminary tests against lexica that contain much longer words, this growth

does not occur as quickly.

7.1 Comparison

The fastest previously known method (to us) for performing the same type of approximate search, that of Schulz and Mihov (2002), which we had used before and was easily available in *foma*, shows much poorer performance as seen in the graph in figure 5. Implementations may of course vary in practical details and so an exact comparison is difficult. However, in comparing our algorithm to Schulz & Mihov we tried to minimize this disturbance in two ways: 1) Schulz & Mihov perform a preliminary step of constructing what they call a Levenshtein automaton for some edit distance n and the input word beforehand; we have not included this construction time in our figures; 2) Schulz & Mihov is essentially a simultaneous depth-first search of two graphs, the Levenshtein automaton and the lexicon automaton $\mathcal{A}$, and this search is performed on the exact same data structure as in the new algorithm (the internal automaton data structure of *foma*). These two points we believe make the algorithm comparison more meaningful than it would under other circumstances.

7.2 Complex distance metrics

All our experiments with the approximate matching were done with Levenshtein edit distance. But the algorithm can easily be adapted to a variety of distance metrics. One of the more useful metrics is to calculate costs from a confusion matrix where each individual substitution, insertion, or deletion can have a different cost. For instance, for Spanish spelling correction, one would want to specify lower costs than ordinarily for substituting **c** for **s**, and **b** for **v** because of their phonetic proximity or equality for speakers.

One can also concoct context-dependent cost schemes. Again, drawing on an example for Spanish, it may be beneficial to give a low cost to substituting **l** as **y**, but only if preceded by a deletion of an **l**. This would represent the replacement of **ll** with **y**, as these are phonetically similar for many speakers. For all such extensions, the only modification the basic algorithm needs is to keeping track of the h-score estimate given that the costs may differ for different operations of word perturbation.

8 Conclusion

We have presented an algorithm for finding approximate matches of a word to paths in a finite automaton. The algorithm is a variant of the popular A^* search algorithm where the main heuristic to guide the search is to record, for each state in the automaton, every symbol that can be encountered in paths extending from that state for n steps. Precalculating this for the special case where $n = \infty$ proved to be an interesting problem in its own right, which we solved by adapting Tarjan's (1972) algorithm for finding strongly connected components in a graph in linear time.

We expect the algorithm to be useful for a number of purposes, and have shown that it performs very efficiently in suggesting spelling corrections for misspelled word, even against very large dictionaries.

References

Aho, Alfred V., John E. Hopcroft, and Jeffrey D. Ullman. 1974. *The Design and Analysis of Computer Algorithms.* Addison-Wesley.

Beesley, Kenneth and Lauri Karttunen. 2003. *Finite-State Morphology.* CSLI, Stanford.

Hart, P.E., N.J. Nilsson, and B. Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107.

Hulden, Mans. 2009. Foma: a finite-state compiler and library. In *EACL 2009 Proceedings*, pages 29–32.

Oflazer, Kemal. 1996. Error-tolerant finite-state recognition with applications to morphological analysis and spelling correction. *Computational Linguistics*, 22(1):73–89.

Pearl, Judea. 1984. *Heuristics: intelligent search strategies for computer problem solving.* Addison-Wesley.

Schulz, Klaus and Stoyan Mihov. 2002. Fast string correction with Levenshtein automata. *International Journal on Document Analysis and Recognition*, 5(1):67–85.

Tarjan, Robert E. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160.

Algorithm 2.1: MARKVINF(v)

$v.index \leftarrow index$
$v.low \leftarrow index$
$index \leftarrow index + 1$
PUSH(v)
for each edge($v \rightarrow v'$)
do
- $v.syms \leftarrow v.syms \cup v'.syms \cup$ edge.sym
- **if** $v.index = 0$
  - **then** MARKVINF(v'); $v.low \leftarrow$ MIN($v.low, v'.low$)
  - **else if** v is on stack
  - **then** $v.low \leftarrow$ MIN($v.low, v'.index$)
- **if** $v.low = v.index$
  - **then**
    - **repeat**
      - POP(v')
      - $v'.syms \leftarrow v.syms$
    - **until** $v = v'$

Algorithm 2.2: MARKVFIN(V, n)

for each vhead $\in V$
do
- $d \leftarrow 0$
- LIMITDFS(vhead)

LIMITDFS(v)
$d \leftarrow d + 1$
if $d > n$
- **then** $d \leftarrow d - 1$; RETURN

for each edge($v \rightarrow v'$)
do
- vhead.syms $\leftarrow$ vhead.syms $\cup$ edge
- LIMITDFS(v')
- $d \leftarrow d - 1$

Algorithm 2.3: SEARCH($word, A$)

agenda(v,pos,g,h) $\leftarrow$ (start state, $0, 0$, *calculate_h*())
while agenda not empty
do
- (pos,v,cost) $\leftarrow$ remove-cheapest(agenda)
- **for each** edge($v \rightarrow v'$)with sym e
  do
  - **if** v is final and pos is end of word
    - **then** SOLUTION(*node*)
  - **if** word(pos) = edge
    - **then** ADD($v', pos + 1, cost$)
    - **else** ADD($v', pos + 1, cost + substcost$, *calculate_h*())
  - ADD($v', pos, cost + delcost$, *calculate_h*())
- ADD($v, pos + 1, cost + inscost$, *calculate_h*())