A CHART PARSING REALISATION OF DYNAMIC PROGRAMMING, WITH BEST-FIRST ENUMERATION OF PATHS IN A LATTICE

Henry S. Thompson

Department of Artificial Intelligence; Centre for Cognitive Science; Centre for Speech Technology Research
University of Edinburgh; 80 South Bridge; Edinburgh EH1 1HN; SCOTLAND

Abstract
Active chart parsing offers a clean and flexible way of implementing dynamic programming to find the best path through a cyclic directed lattices. This paper describes how this comes about and what the general form of a chart parsing realisation of dynamic programming takes. Advantage is taken of the resulting flexibility to produce a system which not only finds the best path, but enumerates paths in order. Finally we exemplify the process for finding paths through a word lattice given bi-class probabilities, and gives a few results from some experiments.

Acknowledgements
The algorithm presented in this paper was developed at the Edinburgh University Speech Input Project, in the course of discussions with Anne Johnstone and John Bridle, and the first implementation was done by Greg Filz. EUSIP was part of the Alvey Large Scale Demonstrator called Speech Input Work Station and Wordprocessor, whose managing participant was the Plessey Company. Funding was provided by Alvey via SERC contracts D/29611, D/29628 and D/29604. This project has been subsumed into the IED/SERC Large Scale Integrated Speech Technology Demonstrator Project, whose industrial partner is Marconi Speech and Information Systems.

I. Introduction
Finding the highest-scoring or lowest-cost path through a lattice of scored edges is a well-known problem whose dynamic programming solution has been taken up both in Artificial Intelligence and Signal Processing (see e.g. Bellman 1957, Nilsson 1971, Pearl 1984, Viterbi 1967). After John Bridle (personal communication) pointed out the relevance of this approach to the EUSIP problem of finding the most probable path through a lattice of labelled edges given a table of pairwise transition probabilities between labels, I designed a modification of the algorithm which not only finds the best path, but in fact enumerates paths in order of score in linear time per path. This is possible because the modified algorithm is constructed in the context of an active chart parsing framework (Thompson 1983), which supports the construction of customised parsers for particular tasks. The algorithm depends on the explicit representation of partial paths within the active chart parsing approach to retain information about alternatives while searching for the best path. This allows the process to be re-started after a solution is found, in order to find the next best solution, and so on. The rest of this paper describes the problem, such rudiments of active chart parsing as are necessary, the aptness of active chart parsing to dynamic programming problems and the ordered path enumeration algorithm.

II. The General Problem
Consider a directed a-cyclic lattice (a graph with a single root, a single end-point and no loops) whose edges are labelled with scored items. There is a partial ordering on the vertices, notionally from left to right, such that vertex $i$ is ordered before vertex $j$ if there is a properly directed path from $i$ to $j$. There is some function from paths (properly connected sequences of edges in the lattice) to scores, which is non-decreasing under concatenation. In particular, there is an inductive version of the scoring function, which can compute the score of any single edge extension to a (possibly empty) scored path. Given these assumptions, dynamic programming/Viterbi search provides an optimal $O(n)$ (where $n$ is the number of edges in the lattice) solution to the problem of finding the best path from one end of the lattice to the other. The approach is inductive. If we know the best path to each vertex one edge from the end we can compute the best path overall by comparing the best extension of each of those. And of course we can find the best path to each vertex one edge from the end by doing the same thing with respect to each vertex one edge back from them, and so on. What makes this less than complete enumeration of paths is the fact that we are working with a lattice. If we consider the following example this becomes clear:

![Figure 1. Simple lattice](image)

Edges have scores—path scores are just the sum of the scores of the edges composing the path. To perform the first step in the dynamic programming process, we need best paths to vertices 5 and 6. To perform the second step, for vertex 5, we need the best path from vertex 4. But this requirement is shared by the second step for vertex 6. It follows that the thing to do is turn the process around, and compute the best paths in left to right order, so that at each step all the prerequisite computations one edge back have already been done. It should be clear that the time complexity of this process is indeed $O(n)$, as we need to consider each
edge only once, in the course of extending the best path to its left end to get the best path to its right end.

If all we care about in the way of results is the score of the best path, we need only keep track of the best score to each vertex as we go. But if the path itself is also required, then we must build backpointers, recording for each vertex as we reach it the reverse of the edge we traversed to get there. Then at the end we can read off the best path back to front (see e.g. Nilsson 1971). This is at this point that the similarity to a parsing problem becomes apparent, and the appropriateness of chart parsing evident.

We can illustrate the progress of the dynamic programming process by a simple tableau. Each step in the process is recorded in a section of the tableau. Each line of each section records a path—the first column gives the end vertex, the second the score and the third the sequence of vertices used to get there. The lines are ordered by vertex. A line is parenthesised if there is another line in the section for a path to the same vertex with a higher score.

The way one step is computed from the previous one should be clear. Take the first unparenthesised path, extend it in all possible ways given the actual lattice at hand, and make new entries for the new paths thus generated, parenthesising as necessary. We treat scores as probabilities, so score combination is by multiplication.

<table>
<thead>
<tr>
<th>End Vertex</th>
<th>Score</th>
<th>Path so far</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>&lt;&gt;</td>
</tr>
<tr>
<td>2</td>
<td>.3</td>
<td>&lt;1&gt;</td>
</tr>
<tr>
<td>4</td>
<td>.25</td>
<td>&lt;1&gt;</td>
</tr>
<tr>
<td>6</td>
<td>.2</td>
<td>&lt;1&gt;</td>
</tr>
</tbody>
</table>

| 4          | .25   | <1>         |
| (4)        | .15   | <1,2>       |
| 6          | .2    | <1>         |
| (6)        | .012  | <1,2,3>     |

| 5          | .125  | <1,4>       |
| 6          | .2    | <1>         |
| (6)        | .05   | <1,4>       |
| (6)        | .012  | <1,2,3>     |

| 6          | .2    | <1>         |
| (6)        | .05   | <1,4>       |
| (6)        | .012  | <1,2,3>     |

| 7          | .0875 | <1,4,5>     |
| (7)        | .0875 | <1,4,5>     |

Figure 2. Tableau of Dynamic Programming construction of best path

Note that there are in fact 10 paths constructed, as per the remark on complexity above.

III. Active Chart Parsing

Active chart parsing is a flexible and elegant methodological framework for the construction of parsers for a wide range of grammatical formalisms. There are a number of strands to its history, with both Earley (1970) and Colmerauer (1970) among its antecedents, but the original synthesis was Kay's (1973). For a more extended introduction, see Thompson & Ritchie (1984) and other references cited there.

The essential notion of active chart parsing is to use one data structure, the chart, to record the input, the intermediate hypotheses and results and the final output of the parsing process. The chart is a directed graph, composed of labelled edges and partially ordered vertices. Edges are of two types, active and inactive. Inactive edges represent complete constituents, whether input as such, or discovered in the course of parsing. The sub-graph composed of inactive edges alone is typically a directed a-cyclic lattice. Active edges represent incomplete constituents, that is, hypotheses about the possible location and partial contents of potential constituents. Edges are never changed—the parsing process consists in the construction and addition of new edges to the chart on the basis of its prior contents and the nature and contents of the grammar with respect to which the input is being analysed.

This process has three essential components:

1) The fundamental rule, which says that whenever the right end of an active edge and the left end of an inactive edge first meet, one or more new edges may be constructed, running from the left end of the active edge to the right end of the inactive edge, provided the label of the inactive edge satisfies the conditions for extension recorded in the label of the active edge. The contents of this new edge are a function of the contents of the two edges which provoked it.

2) The notion of rule invocation, which provides for various schemes governing the way in which initial hypotheses are introduced, e.g. in a data-driven (bottom up) or hypothesis-driven (top down) manner.

3) The notion of scheduling, which separates the creation of edges (by either the fundamental rule or some rule invocation scheme) from their addition into the chart, thereby allowing them to be added in an order which differs from that in which they are created. Since it is the addition of edges to the chart which invokes the fundamental rule and the rule invocation scheme, this allows for flexible control of the search strategy. For example, first-in-first-out scheduling gives rise to breadth-first search of the space of possible analyses, whereas last-in-first-out gives depth-first search.

The matter of rule invocation is not really relevant in the particular case at hand, as there is only one rule, namely that a valid path is made up of contiguous edges which span the lattice (Path \( Path \leq Edge^* \)), and can therefore be built into a specialisation of the fundamental rule. What is relevant is the matter of scheduling, for on this will depend most of the sophistication of the algorithms I am describing here.

IV. A Chart Parsing Realisation of Dynamic Programming

The crucial observation is that if we simply implement a chart parser which would find and score all paths through a lattice, and then impose a modified version
of breadth-first search with pruning, the result is dynamic programming. To find all paths is trivial. The lattice is represented as a collection of inactive edges in a chart. Active edges represent partial paths through the lattice, and contain the score so far and backpointers to the inactive edges incorporated so far. An active and an inactive edge combine to produce a new active edge, with score the product of the edge scores (treating them as probabilities) and contents updated to include the new edge. All the action is in the queueing. Instead of a simple FIFO queue, we have a two level structure, with edges grouped together by right-hand vertex, and those groups sorted in left-to-right order. (The order of edges within a group is irrelevant.) It should be clear that running this way will build spanning edges for all paths, and will do so left-to-right, one vertex at a time.

To get only the best paths is straightforward. Instead of one edge queue, we have two. The higher priority one is a simple FIFO queue, which we initialise with an empty active edge with score 1 at the first vertex in the chart to get things going. All edges produced by the operation of the fundamental rule, however, are queued on another, lower priority, queue, which operates as above, grouping by right-hand vertex. When the time comes to add an edge to the chart, it comes from the higher priority queue, or if that is empty, we remove the highest scoring edge (or edges) from the group with the leftmost right-hand vertex from the other queue and queue it on the higher priority queue. We discard any other edges in that group. (Two queues are not strictly speaking necessary, but doing it this way makes the changes required for the enumeration version easier to follow.) It should be clear that this will exactly reproduce the dynamic programming search described in section II. In particular note that each inactive edge will be processed only once (assuming no equal-score paths). An inactive edge is processed only when an active edge is added which has right-hand end at its left. But since all active edges with that right-hand end will have been accumulated in the queue, and only the best one added to the chart, this will happen only once.

V. Path Enumeration

Moving from finding the best path to the ordered enumeration of paths is now easy, because of the reification of partial paths which is central to the parsing approach. All that changes is what happens when we consider the group of edges with the leftmost right-hand end. We still re-queue the highest scoring edge(s) onto the high priority queue as before. However instead of discarding the remaining edges we move them as a group to a new queue with even lower priority, where the edge groups are queued preserving the ordering by right-hand end.

It should be clear that up to the end of the first pass, when the best path has been found, the operation of this algorithm is very similar to the previously described one. But once the top two queues are empty, when the best path has been found, the third queue, which has accumulated the partial paths which were not chosen for extension in the first pass, will then be invoked. What happens as a result is simply that the whole contents of this queue are transferred to the middle queue, from which the best scoring edges for each final tag in the group for the leftmost right-hand end will then be re-queued on the top queue for processing, and the rest from that group demoted to the bottom queue, and so on. The net result of this second pass will thus be to find the next best path, using some combination of partial paths left in the bottom queue from the previous pass, and new paths constructed from them. This process can obviously continue in score order until all paths have been found.

The time complexity of each pass is at worst as before plus a factor for the time to find the best edges from each group during each re-queueing step. This is at worst the number of edges times the number of passes, since every active edge still in the middle queue has to be examined once per pass to see if it is the highest in its group, and at worst we get one new edge per edge per pass. The 'at worst' qualifications are because as the process continues the left end of the lattice effectively moves rightwards. As all initial sub-paths up to some point have been explored the number of edges involved in each pass diminishes accordingly. In any case the time for each pass is still linear in the number of edges, being the sum of the two linear sub-processes, one associated with the top queue and one with the middle one.

For more implementation details and examples of operation, see Thompson 1989.

VI. The Particular Problem—Tagged Word Lattices

We now move from the general to the particular, using an example from speech recognition work at Edinburgh.

The items in the lattice are words. Each word belongs to one or more categories, denoted by tags, chosen from a finite set. The first order transition probability is known for each pair of tags. An example might be displayed as follows, where the ordering is implicit in the vertex labels, and the edges are labelled with words and their tags, whose transition probabilities are given in the accompanying matrix (omitted in this version of the paper):

Figure 3. Sample lattice of tagged words

There are six paths through this lattice, with 21 possible tag sequences, ranging from the best, "recognise[V] beach[N]" with probability .237, to the worst, "wreck[N] an[D] [PN] speech[N]" with probability .00001. Dynamic programming is the obvious way to avoid constructing all twenty-one tag sequences, computing the product of their constituent transition probabilities and sorting the result, a method which would become prohibitive in our application in which a typical lattice may have 1010 paths and who knows how many tag sequences.

It is slightly more complicated to apply our chart parsing solution for this case. Instead of simple scores...
on edges, we have words, with one or more tags associated with them. Each word has a score based on the strength of the acoustic evidence—an estimate of the conditional probability of the word given the acoustic hypotheses available. Hereafter I will refer to this as the word score. Each word/tag pair has a probability associated with it—the conditional probability of the tag given the word (e.g. man is much more likely to be a noun than a verb). I will call this the tag/word score. We also know the probability of all possible tag/tag transitions, which I will call the tag>tag score. (In our case both these sets of probabilities are frequency derived from a relevant corpus (Foster and Hurford 1986.) This means that for any given (partial) path through the lattice, there are a large number of possible scores, depending on which of the many possible tag sequences represented by that path is considered. But it is not too difficult to modify the simple algorithm described above to accommodate this—we simply need to process the best score up to a given vertex for every possible path-final tag. This is because the inductive step in finding the best path depends not only on the score of the next shorter path, but also the contribution made to the new score by the transition probability. To understand this, consider the following example:

![Figure 3. Lattice fragment](image)

Here we see that even if the best path to $v_n$ is via the $w_1$ edge, the best path to $v_{n+1}$ may be via the $w_2$ edge, if the tag>tag score from $t_3$ to $t_4$ is sufficiently better than that from $t_1$ or $t_2$.

The required modification is easy to implement. Active edges record not only their score, but also the last tag in the path they represent. When an active and an inactive edge meet, one new edge is produced for each word/tag pair contained in the inactive edge, with new score the product of the active edge score, the word score, the tag/word score and the tag>tag score from the tag of the active edge to the tag of the word/tag pair. The central queue now groups edges for each occurring path-final tag for each right-hand vertex. For the above example, this would mean three different groups for the four edges incident on $v_n$, one for each of $t_1$, $t_2$ and $t_3$. Re-queuing then re-queues the best edge(s) stored for each final tag of the left-most right-hand vertex, and demotes the rest.

The result will be similar to that of the simpler case, except of course that in the end there will be as many spanning paths as there are tags incident on the final vertex.

It should be clear that processing time is still linear in the number of edges, but the constant term has increased by something like the product of the average number of tags per word times the total number of tags. The factor for the total number of tags is because each edge will be processed by the fundamental rule with respect to all the edges incoming at its right hand end, and while in the simple case there would only be one such edge, now there might be as many as one per possible tag. The factor for the average number of tags per word in the lattice is because doing the fundamental rule now requires iteration over the word/tag pairs represented by an inactive edge.

**VII. Results**

In practice, with an input lattice composed of 31 edges containing 39 word/tag pairs which required 3184 passes to enumerate the 14076 possible tagged paths, the time for the slowest pass, the 792nd, was only 5 times slower than the first. Thus it appears that the worst-case increase in processing time per pass is not actually anywhere near realised—in fact we see a roughly linear increase over the first 500 passes, then a stable period over the next 500 passes, and finally a decline through the last 2100 passes as no new paths are found. By the time this last phase is entered, after the 1035th pass, the algorithm has in fact become quite inefficient, as it has actually enumerated all the remaining paths and is simply sorting them in a very inefficient manner. We could detect this case by noting when the middle queue is reduced to only one group, that for the right-most vertex, and then simply sorting that group by score, since once we reach that state there will never be any new paths created.

**References**


