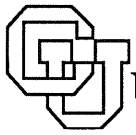
PARSING IN OPTIMALITY THEORY: A DYNAMIC PROGRAMMING APPROACH

Bruce Tesar

CU-CS-714-94



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

PARSING IN OPTIMALITY THEORY
A DYNAMIC PROGRAMMING APPROACH

Bruce Tesar

CU-CS-714-94 1994

Department of Computer Science
University of Colorado at Boulder
Campus Box 430
Boulder, Colorado 80309-0430 USA

Parsing in Optimality Theory:
A Dynamic Programming Approach

Bruce Tesar

Department of Computer Science
Campus Box 430
University of Colorado at Boulder
Boulder, CO 80309-0430
tesar@cs.colorado.edu

1. The Parsing Problem in Optimality Theory

In Optimality Theory (Prince & Smolensky 1991, 1993), grammaticality is defined by an optimization process. For any given linguistic input, the grammatical parse of that input is selected from a set of candidate parses. The grammatical parse is optimal in that it does the best job of satisfying a ranked set of universal constraints.

A grammar specifies a function. The grammar itself does not specify an algorithm, it simply assigns a grammatical structure to each input. However, one can ask the very interesting computational question of whether efficient algorithms exist to compute the structure assigned to a linguistic input. This is the parsing problem. In Optimality Theory, the parsing problem is easily understood as an optimization problem: search the space of candidate structures for the one that optimally satisfies the ranked constraints.

The general spirit of Optimality Theory is to have the GEN function generate a large and general space of candidate structural descriptions for an input, leaving much of the work to the constraints to determine grammaticality. This makes the parsing problem non-trivial; GEN is often envisioned as generating an infinite number of candidate structural descriptions, so simple exhaustive search is untenable. One alternative to exhaustive search is to attempt to exploit specific knowledge of the relationship between the constraints and the candidates generated by GEN, in order to limit the search.

Part of what makes characterizing the search a challenge is that several different phenomena may be involved in a structural description of an input. In addition to parsing input segments into different parts of structure, input segments may be deleted, and structural positions may be left unfilled, to have material inserted into them by later processes (e.g., phonetics). The next section will briefly examine some related problems, and existing solutions to those problems. The rest of the paper will then show how to combine the ideas from these other areas into a solution to the parsing problem in Optimality Theory.

2. Related Problems and Algorithms

The algorithms for parsing in Optimality Theory described below take inspiration from work done on several related problems.

2.1 Traditional Natural Language Parsing: Chart Parsing

Chart Parsing (Kay, 1980) is a class of algorithms for parsing context-free grammars common in natural language processing. It uses an approach known as dynamic programming. However, this approach only finds parses for which the input is precisely generated by the grammar; no deletions of inputs, or insertions of additional material, are permitted.

2.2 Sequence Comparison

Sequence comparsion (Sankoff & Kruskal 1983) is precisely concerned with editing operations. The solution of a sequence comparison problem is the minimum set of editing operations necessary to change one sequence into another. The editing operations of insertion, deletion, and transpositon, are shown to be sufficient for dealing with a wide range of applications. Again, the dynamic programming approach is the key to solving these problems efficiently.

However, the sequence comparsion problem looks at structures (sequences) two at a time. In Optimality Theory, we would like to compare an input to a large space of candidate structures. Also, the goal in Optimality Theory is not to minimize the number of editing operations, but to minimize violations of strictly ranked constraints, where the constraints will make the appropriate restrictions on editing operations (as well as other things).

2.3 Hidden Markov Models and Stochastic Context-Free Grammars

Hidden Markov Models (Rabiner 1989), and Stochastic Context-Free Grammars (Lari & Young 1990), can be thought of as mapping an input to a structure from a space of structures, and via an optimization process. The structures in Hidden Markov Models come from an underlying state machine, equivalent to a regular grammar. The Viterbi algorithm, again a dynamic programming-based algorithm, may be used to efficiently find the sequence of state transitions that maximizes the probability that the observed input would be generated. Optimality Theory parsing can be likened to this, with optimization with respect to constraint satisfaction, instead of probabilities. However, like chart parsing, the Viterbi algorithm assumes that no editing operations are allowed, such as insertion or deletion.

3.0 Parsing with a Regular Language GEN

The algorithm described in this section combines the approach to editing operations employed in Sequence Comparison with the approach to optimal structure assignment used with Hidden Markov Models. Since both of the source approaches are based on dynamic programming, it is perhaps no surprise that the algorithm combining the insights of the two is also a dynamic programming algorithm.

Recall that GEN maps an input to a set of candidate structural descriptions. Optimality Theory stipulates nothing further about the nature of the set of structural descriptions, or how one might describe the mapping. One simple strategy is to grant GEN a description of a space of *partial structures*, each partial structure having a set of slots (structural positions) that may accept input material. The set of candidate structural descriptions for a given input is then a set of specified ways of parsing the input into a partial structure, for all partial structures permitted by GEN. Any input-accepting slot not filled with an input segment is assumed to be epenthesized, and an input segment may be deleted by marking it as deleted within the structural description, rather than parsing it into a structure slot.

In this section, it is assumed that the set of partial structures used by GEN can be described by a regular grammar. Furthermore, the parsing of an input into a partial structure is restricted so that the linear order of the input is preserved. Deleted input segments will be so marked in the structural description, in the appropriate order. Based on what has been said so far, we may understand GEN to function as follows: for each partial structure PS generated by GEN's regular grammar G, generate an structural description S for each order-preserving way of parsing the input I into PS.

It should here be emphasized that the regular grammar just discussed is a descriptive formalism useful in understanding GEN; it is NOT a computational mechanism. The actual computational mechanism understandable in terms of the regular grammar is the set of operations contained in the Operations Table, described below.

Regular grammar productions have only one non-terminal on the right-hand side. As a result, step-by-step derivations of partial structures only have one non-terminal present at any given step, save for the last step which has none. This permits each non-terminal to be thought of as a kind of state: a step of a derivation is in the state corresponding to the single non-terminal present. The dynamic programming table will be based upon this view: for a given non-terminal X, table position $[i_j,X]$ will represent the optimal way of parsing input segments (1 thru j) into the best partially derived structure which is in state X.

The Basic CV Syllable Structure model will be used as an example. This model is described in chapter 6 of [Prince & Smolensky 1993]. The space of structures considered by GEN in this model are strings of syllables, where nuclei are mandatory, and onsets and codas are optional. NOTE: this is a statement of the universal set of structures to be considered, and NOT the inventory for any particular language. Partial structures will here be treated as strings of the symbols {E,O,N,C,o,n,c}. The non-terminal symbols {O, N, C} stand for onset, nucleus, and coda, respectively. The symbols {o, n, c} represent the input-accepting positions for onset, nucleus, and coda. For a given structural description, each such lower-case symbol is replaced with either an input segment or an epenthetic segment. NOTE: the use of this example should NOT to be taken to claim that regular grammars are sufficiently powerful to generate adequate representations for full, general phonology, any more than the simple CV Basic Syllable Structure model should be taken as a complete theory of phonology.

Here is a regular grammar describing this set of partial structures:

E -> e | oO | nN
O -> nN
N -> e | cC | oO | nN
C -> e | oO | nN

The non-terminals {E, O, N, C} may be thought of as corresponding to states. E is that starting state. O signifies that the last position (terminal) generated was an onset (o), N that a nucleus (n) was just generated, and C a coda (c). e signifies an empty string; those non-terminals which may evaluate to e correspond to possible finishing states. O is not a finishing state, because a syllable with an onset must also have a nucleus.

3.1 The Dynamic Programming Table

The table for parsing an input string $I = i_1 i_2 \ldots i_n$ is as follows:

	BOI	i_1	i_2	...	i_n
E	{}				
O					
N					
C					

This table has one row for each non-terminal in the grammar, and one column for each element in the input, plus a first column, BOI, standing for "beginning of input." This column is present because structure may be epenthesized at the beginning, before any of the input has been parsed. The table entry [N, i_2] represents the optimal way of parsing up through the second segment of the input, with a nucleus being the last structural position generated.

The parsing algorithm will proceed by filling in the columns of the table one at a time, left to right. After the best way of parsing the input through segment i_{j-1} ending in each non-terminal has been calculated (the entries of column i_{j-1}), those values are then used to determine the best way (for each non-terminal) of parsing the input through segment i_j. Once all the values for the last column are determined, the values in the table cells of the last column in rows corresponding to possible finishing states are compared. The cell value with the highest Harmony best satisfies the constraints, and gives the Harmony of the optimal parse of the input. The corresponding optimal structure may be represented by a set of pointers attached to the table: each cell contains both its harmony value and a pointer to the preceding cell in the derivation represented. The pointer will indicate what structure was added in moving from the preceding cell. Once the final column cell corresponding to the optimal parse has been determined, the structural description may be reconstructed by tracing backwards through the table via the pointers.

3.2 The Operations Table

An operation is here defined as a step in the assignment of structure to an input. Operations connect entries in the Dynamic Programming Table. Recall that the three main primitive actions involved in assigning structure to input are:

(a) parsing a segment of input into a piece of structure;
(b) deleting an input;
(c) inserting an unfilled piece of structure (epenthesis).

Primitive actions (a) and (c) involve generating pieces of structure, so they must be coordinated with productions in the structure generation grammar of GEN; (b) does not involve structure generation.

The Operations Table may be constructed from the regular grammar as follows. First, for any possible state [X,i_j] where more input remains to be parsed, one allowable operation is always to delete the next input segment. So, for each non-terminal, include a delete operation. For each grammar rule with a non-terminal X on the right-hand side, two operations are possible: the generated element of structure has the next input segment parsed into it, or the generated element of structure contains an inserted segment (an

insertion). So, for a given non-terminal X, for each rule generating it, create two operations, a parse operation and an insertion operation. Notice that any element of structure generated by the grammar must be phonetically realized, either through parsing or epenthesis. Thus, it is the grammar rules that determine what combinations of realized structural elements are and aren't candidates.

The Operations Table gives the set of operations that may lead to a possible entry in the Dynamic Programming Table. Below is shown the Operations Table for the CV syllable structure example.

New State	(Prev. State, Struc, {violations})	Grammar Rule	
[E, i_j]	([O,i_{j-1}], $\langle i_j \rangle$, {*PARSE})		DELETION
[O, i_j]	([O,i_{j-1}], $\langle i_j \rangle$, {*PARSE})		DELETION
	IF i_j=*Con* ([E,i_{j-1}], o/i_j, {})	E -> oO	PARSING
	IF i_j=*Con* ([N,i_{j-1}], o/i_j, {})	N -> oO	
	IF i_j=*Con* ([C,i_{j-1}], o/i_j, {})	C -> oO	
	([E,i_j], o/□, {*FILL^{ONS}})	E -> oO	INSERTION
	([N,i_j], o/□, {*FILL^{ONS}})	N -> oO	
	([C,i_j], o/□, {*FILL^{ONS}})	C -> oO	
[N, i_j]	([N,i_{j-1}], $\langle i_j \rangle$, {*PARSE})		DELETION
	IF i_j=*Vow* ([E,i_{j-1}], n/i_j, {})	E -> nN	PARSING
	IF i_j=*Vow* ([O,i_{j-1}], n/i_j, {})	O -> nN	
	IF i_j=*Vow* ([N,i_{j-1}], n/i_j, {})	N -> nN	
	IF i_j=*Vow* ([C,i_{j-1}], n/i_j, {})	C -> nN	
	([E,i_j], n/□́, {*FILL^{NUC}})	E -> nN	INSERTION
	([O,i_j], n/□́, {*FILL^{NUC}})	O -> nN	
	([N,i_j], n/□́, {*FILL^{NUC}})	N -> nN	
	([C,i_j], n/□́, {*FILL^{NUC}})	C -> nN	
[C, i_j]	([C,i_{j-1}], $\langle i_j \rangle$, {*PARSE})		DELETION
	IF i_j=*Con* ([N,i_{j-1}], c/i_j, {*-COD})	N -> cC	PARSING
	([N,i_j], c/□, {*-COD})	N -> cC	INSERTION

The Operations Table relates to the Dynamic Programming Table as follows. The Operations Table gives all of the possible operations that may fill a given cell in the Dynamic Programming Table. Each of the possible operations "competes" to fill in the table. The operation resulting in the most Harmonic substructure actually gets to put the Harmony of its substructure into the cell. The general formula for filling a Dynamic Programming Table cell (where X and Y represent non-terminals) is

$$[Y,j] = \max H_{op}\{ [X,k]_{op} + \{\text{violations}\}_{op} \}$$

where op ranges over the set of operations listed for Y. Here, "+" means "union", in that the result is the union of the sets of marks of the two operands. In English, select the operation with the most Harmonic sum of the Harmony of the previous state $[X,k]_{op}$ and the violations incurred by enacting the operation, $\{\text{violations}\}_{op}$.

One crucial aspect has not yet been explained about the application of this formula. The Parsing and Deletion operations have a previous state cell from the previous column in the DP Table, i_{j-1}. However, the Insertion operations refer to other cells in the same column of the DP Table as the cell being filled.

How, in general, can these cells be filled, if the value for each cell in the column depends upon the values in the other cells of the column? The answer involves some intricate details of the algorithm, and is addressed in the next section.

Notice that, in the Operations Table, the Parsing operations contain IF conditions. These are used to enforce constraints of the CV model that consonants (*Con*) may only fill onsets and codas, and vowels (*Vow*) only nuclei. These restrictions are assumed by the CV model to be part of GEN, and so are included here as IFs.

As an example, consider cell $[O,i_2]$ of the DP Table. The operations that may fill this cell are those with forms listed under $[O,i_j]$ in the Operations Table. One possibility is the deletion operation. The resulting Harmony of the operation will be the Harmony listed in cell $[O,i_1]$, which is a list of constraint violation marks, with the mark {*PARSE} added to it.

3.3 Limiting Structure: FSG Cycles

The insertion operations consume no input, and so they map one cell in a column to another. In principle, an unbounded number of such operations could apply, and in fact such structures are part of the formal definition of GEN. However, the algorithm need only explicitly consider a specified finite number of such operations within a column. This is because the constraints must explicitly ban "cycles" of insertions in order for the optimal value to be well-defined. Suppose there are M non-terminals. Then if M or more insertions take place consecutively, at least one non-terminal will be generated twice in the process; the section of the structure between these two occurrences is the cycle. If the constraints do not make a structure containing such a cycle more harmonic than the same structure with the cycle removed, then there is no optimal value, because one could always increase the Harmony by adding more insertions with such cycles. In fact, if such cycles have no Harmony consequences, then there will be an infinite number of Optimal structures, as any optimal structure can have more cycles of insertions added. Thus, for the optimization with respect to the constraints to be well-defined and reasonable, the constraints must strictly penalize insertion cycles. In the CV Syllable Structure model, an example of a cycle of epenthesis would be an entire epenthesized syllable. The FILL constraints serve to penalize structure in that model, by penalizing any structural positions unfilled by input segments.

Therefore, if there are M non-terminals, only at most M-1 consecutive insertion operations can apply in any optimal form. Applying the operations to the Dynamic Programming Table may be done by first filling in all cells of a column by considering only deletion and parsing operations (which only use values from the previous column). M-1 passes are then made through the column cells, considering the insertion operations: if the resulting Harmony of an insertion operation into a cell from another cell in the same column is higher than the Harmony already listed in the cell, replace the Harmony in the cell with that resulting from the considered insertion operation.

The ban on insertion cycles is the crucial observation that allows the algorithm to complete the search in a finite amount of time; although the space of structures to be searched is infinite, there is a provably correct (input-dependent) bound on the space of structures that actually need to be considered.

3.4 The Locality Restriction on Constraints

The Operations Table scheme described above contains a strong assumption about the universal constraints of the grammar: it assumes that constraint violations can be assessed within the context of a

single operation. This is a kind of locality constraint, in that the structure relevant to determining the assessment of a constraint is limited to the structure immediately referenced by the operation. The strength of that restriction on the constraints becomes even more apparent when the restrictions imposed upon operations are considered. An operation makes reference to the most recently generated piece of structure (the previous state), the most recently consumed segment of input, the next generated piece of structure if one is generated (the next state), and the next available segment of input. Constraints are restricted to considering two consecutive segments of input, two consecutive elements of structure, and the parsing relations between the inputs and the structure. It is this locality that allows the Operations Table to include for each operation the constraint violation marks incurred by that operation.

This locality restriction on the constraints is here shown to be a sufficient condition for the correctness of the algorithm. The algorithmic cost of relaxing this restriction in various ways is the subject of current research.

3.5 A Simple Example

Here is an example of the filled Dynamic Programming Table for the input /VC/, with the constraint ranking ONS ≫ -COD ≫ FILL^{NUC} ≫ PARSE ≫ FILL^{ONS}.

In this table, the arrow points in the direction of the previous cell. The label of the previous cell in given below the arrow. The abbreviations next to an arrow indicate the kind of operation that filled the cell: i for insertion, d for deletion, and p for parse. The right side of each cell shows the substructure represented by that cell (below), and the constraint violation marks assessed that substructure (above).

	BOI	i_1 = "V"	i_2 = "C"
E	START {} [E,BOI]	← d {*PARSE} [E,BOI] ⟨V⟩	← d {*PARSE *PARSE} [E,i_1] ⟨VC⟩
O	↑ i {*FILL^{ONS}} [E,BOI] □	←,↑ d,i {*FILL^{ONS} *PARSE} [O,BOI],[E,i_1] □⟨V⟩	↖ p {*PARSE} [E,i_1] ⟨V⟩C
N	↑ i {*FILL^{ONS} *FILL^{NUC}} [O,BOI] □□	↖ p {*FILL^{ONS}} [O,BOI] □V	← d {*FILL^{ONS} *PARSE} [N,i_1] .□V.⟨C⟩
C	↑ i {*FILL^{ONS} *FILL^{NUC} *COD} [N,BOI] .□□□.	↑ i {*FILL^{ONS} *-COD} [N,i_1] .□V□.	↖ p {*FILL^{ONS} *-COD} [N,i_1] .□VC.

Optimal Parse: .□V.⟨C⟩
This parse is represented in cell [N,i_2].

Cell [O,i_1] represents a true tie (although the two substructures are not part of the overall optimal structure). There are two ways to derive an essentially identical structure: first epenthesize and then delete, or first delete and then epenthesize. These are represented in the cell by the two arrows, with two

corresponding operations. In this case, the tie might be seen as a kind of anomaly, having no significance to the ultimate phonetic realization. However, it is possible that the order could be significant in other models (e.g., with the ALIGN constraint; see (Prince & Smolensky 1993)). Furthermore, if two truly different substructures for the same cell incurred identical marks, including a pointer for each permits all the optimal structures to be recovered from the table, if that cell should happen to figure in the set of structures ultimately found to be optimal.

4.0 Formal Statement and Analysis for Regular Grammar GEN

4.1 Summary of Assumptions

Assumptions about GEN:

(1) GEN has a space of partial structures described by a regular grammar G;
(2) each terminal of G may be replaced by a segment of input;
(3) restrictions contained in GEN (i.e., independent of the universal constraints) on the parsing of input segments into structure elements may be expressed in the form "input segment type A may be parsed into structure element type B";
(4) the linear order of the input segments must be preserved in all candidate structures generated by GEN;
(5) each structure element present in an structure must be either filled with an input segment or unfilled;
(6) each input segment must be either parsed into a structure element or marked as unparsed (deleted) in the structure.

Assumptions about the Universal Constraints:

(7) each constraint must be capable of being evaluated on the basis of a consecutive pair of structure elements and a consecutive pair of input segments.

Notice that assumptions (5) and (6) about GEN imply the three primitive actions parse, insertion, and deletion.

The locality restriction on constraints given in (7) is specific to the regular grammar case. A different kind of notion of locality will be needed for the context-free grammar case.

Given these assumptions, a description of GEN consists of the regular grammar G, plus an restrictions of the type described in assumption (3).

4.2 Construction of the Parser

Constructing the parser from a description of GEN consists of constructing the Operations Table.

Construction of the Operations Table:

For each non-terminal Y in G, construct the following:

A deletion rule $[Y,i_j]$: ($[Y,i_{j-1}]$, $\langle i_j \rangle$, {*violations})

For each rule in the grammar of the form X -> Yt,

An insertion rule $[Y,i_j]$: ($[X,i_j]$, $t/\square$, {*violations})

A parsing rule $[Y,i_j]$: ($[X,i_{j-1}]$, t/i_j, {*violations})

Any restrictions on the types of inputs that may be parsed into structure element t may be expressed as in IF condition preceding the parsing rule.

4.3 Operation of the Parser

Applying the parser to a particular input I consists of constructing the Dynamic Programming Table, and filling in the cells of the table.

Construction of the Dynamic Programming Table:
The table has a row for each non-terminal in G (including the start symbol E). The first column is labeled BOI, with a successive column for each segment of the input (in order).

Filling the Dynamic Programming Table:
For each column j, proceeding from left to right
  For each row X (except BOI)
    For the deletion operation D-OP listed for X
      Set $[X,i_j]$ = Harmony(D-OP)
    For each parsing operation P-OP listed for X
      If Harmony(P-OP) > $[X,i_j]$, then set $[X,i_j]$ = Harmony(P-OP)
  Repeat until no cell entries change
    For each row X
      For each insertion operation I-OP listed for X
        If Harmony(I-OP) > $[X,i_j]$, then set $[X,i_j]$ = Harmony(I-OP)

The expression "Harmony(OP)" here refers to $([X,k]_{op} + \{violations\}_{op})$, the harmony value that the given operation seeks to place into a cell.
The Harmony of the optimal parse is the maximum of the cell entries in the final column, and in rows corresponding to valid finishing states (non-terminals from which e may be derived in G).

4.4 Algorithm Correctness and Complexity

The problem meets both of the primary criteria for dynamic programming approaches: overlapping subproblems and optimal substructure [Cormen, Leiserson, & Rivest 1990]. The assumptions about GEN with a regular grammar give the overlapping subproblems, while the locality assumption about the constraints is sufficient for optimal substructure.

If M is the number of non-terminals in G, and N is the size of the input, the complexity of the parsing algorithm is $O(M^2N)$. Notice that this is linear in the input size, so for a fixed GEN (grammar), it is a linear algorithm.

5.0 Parsing with a Context-Free Language GEN

Assuming that GEN has a general context-free grammar generating partial structures makes things more complicated. The Dynamic Programming Table is larger: for each non-terminal X in the grammar, there will be a cell [X,j:k], for all j,k between 1 and n with $j <= k$. The cell [X,j:k] represents the optimal way of parsing input segments i_j thru i_k into a subtree with root X. Clearly, this will result in a complexity that is no longer linear in the input length.

An operation in the context-free case may refer to more than one previously filled cell. If X -> YZ is a grammar rule, then an operation based upon that rule to fill cell [X,j:k] may use, for example, [Y,j:m] and [Z,(m+1):k]. The general formulas for these rules would be based upon those used in Stochastic Context-Free Grammars (Lari & Young 1990).

Acknowledgements

Valuable guidance was provided by Paul Smolensky and Jim Martin. I also wish to thank Mark Liberman and David Haussler for valuable conversations. This work was supported by an NSF Graduate Fellowship to the author, and NSF grant IRI-9213894.

References

Corman, Thomas, Charles Leiserson, and Ronald Rivest. 1990. *Introduction to Algorithms.* MIT Press, Cambridge, MA.

Jakobson, Roman. 1962. *Selected writings 1: phonological studies.* The Hague: Mouton.

Kay, Martin. 1980. Algorithmic Schemata and Data Structures in Syntactic Processing. CSL-80-12, October 1980.

Lari, K., and S. J. Young. 1990. The Estimation of Stochastic Context-Free Grammars Using the Inside-Outside Algorithm. *Computer Speech and Language* 4:35-36.

Prince, Alan and Paul Smolensky. 1991. Notes on connectionism and Harmony Theory in linguistics. Technical Report CU-CS-533-91. Department of Computer Science, Univ. of Colorado, Boulder.

Prince, Alan and Paul Smolensky. 1993. *Optimality Theory: Constraint Interaction in Generative Grammar.* Ms. Rutgers University, New Brunswick, and University of Colorado, Boulder. NJ. To appear as Linguistic Inquiry Monograph, MIT Press, Cambridge, MA.

Rabiner, L.R. 1989. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Proc IEEE* 77(2):257-286.

Sankoff, David and Joseph Kruskal. 1983. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison.* Addison-Wesley, Reading, MA.