UNIVERSITY OF CALIFORNIA

Los Angeles

Generation, Recognition, and Learning in Finite State Optimality Theory

A dissertation submitted in partial satisfaction of the

requirements for the degree Doctor of Philosophy

in Linguistics

by

Jason Alan Riggle

2004

The dissertation of Jason Alan Riggle is approved.

Colin Wilson

Kie Zuraw

Charles E. Taylor

Edward P. Stabler, Committee Chair

University of California, Los Angeles

2004

ACKNOWLEDGEMENTS

When I met Ed Stabler I was electrified by the types questions that he was asking of linguistic theories and even more so by the fact that he seemed to have an idea of how to answer those questions. Without his insight and generous assistance this dissertation would not have been written. I have also been extremely lucky to have Colin Wilson and Kie Zuraw on my committee. Colin has been an outstanding sounding board for ideas and has an uncanny ability to crash brittle models and find holes in theories. Kie's superlative expository advice and willingness to work through the details of the most gory proofs have made this work much better and definitely clearer. My external member, Chuck Taylor, was also a great asset in encouraging me to take perspectives on linguistic issues that I otherwise wouldn't have considered.

UCLA has been a great place to work on a PhD. The community of students and faculty is always stimulating and supportive. Among the faculty that I owe the most thanks for making my time here rewarding and fun are: Bruce Hayes, for his numerous insightful comments on my work, both computational and otherwise; Pamela Munro, for being a friend and mentor and for showing me that languages aren't quite as tidy as you might think just from thumbing through grammars in the library; Virgil Lewis, for teaching me about Pima and putting up with countless hours of interrogation about one tiny aspect of his language; and Donca Steriade for her seemingly instant grasp of the ramifications of phonological proposals. So many students listened to so many versions of the proposals here and gave me so many great comments that I can't possibly thank them all – but I'll try. First and foremost my office mate, friend, co-author, and traveling companion Greg "the Kobelix" Kobele deserves great thanks for talking through many of the ideas here and offering indispensable advice on their implementation and exposition; Greg deserves credit as the 'invisible' 5th member of my committee (smelled but not seen). Harold Torrence and Leston Buell have been good friends and lunch companions throughout my time at UCLA. Marcus Smith has been a great colleague and fellow Pima-researcher (even if he never would admit I was right about the infixation). Sarah Van Wagenen-Foxe, Ben Keil, Katya Pertsova, and especially Jeff Heinz kept me enthusiastic by constantly expressing interest in my computational work and by being willing to give me feedback

on whatever hair-brained scheme I was cooking-up in a given week. All the Pima-kids – Brook, Eric, Heriberto, Sahyang, Shannon, Haiyong, Shabnam, Jill, and Melissa made our field methods class great. And, finally, thanks to Manola, Julia, and Luca for keeping me from working all the time, for buying me drinks, and for listening to me sing Elvis out at karaoke (oh yeah, and thanks to the King!).

None of this would have been possible if the folks at UC Santa Cruz hadn't hooked me on linguistics and taught me to love playing with formal models. I am most indebted to my first phonology teachers, Junko Itô, Armin Mester, and Jaye Padgett, for giving me the skill-base and tools to tackle any phonological problem, to Judith Aissen, whose Syntax 1 class is probably the reason that I am a linguist today, and to Donka Farkas for always providing a sympathetic ear and for introducing me to Suzanne.

There are many linguists to whom I am indebted for insight and support, through help with this work, help in becoming a linguist myself, or for the inspiration their work has provided. Some people I can't fail to thank here are Paul Smolensky, Bruce Tesar, John McCarthy, Geoff Pullum, Adam Albright, Eric Baković, Rachel Walker, Chris Potts, Dan Albro, Alan Yu, Nicole Nelson, Maria Gouskova, Adam Ussishkin, Andrew Wedel, and Nathan Sanders.

I also want to thank some of my crucial formative influences, in basically reverse chronological order, the Oak Grove School, especially Lee Nichols and Larry Johnson; the kids I misspent my youth with, Shana, Steve, Katy, Jesse, Dylan, Stefan, CJ, Greg, Jason, and, of course, Bert – see I told you those hours spent modeling the world with numbers and dice (playing D&D) would ultimately pay off! I am grateful to my mom and dad for setting me on this path in the first place by encouraging me to always ask questions, follow my passion, and do what I love. There is no one that I have known longer, or has influenced my life more greatly, that my twin brother Mathew. Thanks, Mat, for being my first fellow-scientist and colleague in the enterprise of figuring out the world.

Finally, thanks go to my wife Suzanne whose love and support have made life enjoyable throughout this process, whose keenness for inquiry and sense of humor make life enjoyable period, and whose indefatigable editorial assistance has made this work (more) grammatical, and not completely lacking in commas.

TABLE OF CONTENTS

1 Introduction

Like many phonological models that preceded it, Optimality Theory (OT; Prince and Smolensky, 1993) characterizes phonological grammars as relations between inputs (underlying forms) and outputs (pronounced surface forms). Unlike earlier theories, OT does not give an explicit derivational scheme for producing outputs from inputs, but rather provides a set of ranked constraints governing the input-output relation and states that the actual output is the one (or ones) among the infinite range of possible output forms that optimally satisfies the ranked constraints. Moreover, the same universal set of constraints is assumed to govern every language and the differences among the phonological patterns that are observed across languages are assumed to emerge solely from differences in the ranking of the constraints in individual languages.

There are three fundamental computational problems/questions that must be addressed within Optimality Theory, or indeed in any theory of generative grammar. The first question is how optimal output forms are to be found and whether or not it is the case that they can be found efficiently. This is the generation problem.

Also at issue is whether or not it's possible to work backwards from a surface form to the to the set of inputs that yield that form given some information about the ranking of the constraints. This is the recognition problem.

Finally, of central importance is whether or not it's possible to learn an optimality theoretic grammar (a ranking of the constraints) after having seen only a finite sample of the language defined by that grammar and, of course, whether this can be done efficiently. This is the learning problem.

1.1 Finite State Optimality Theory

In this dissertation I will develop a computational implementation of Optimality Theory based on Ellison's (1994) proposal that the optimization problem in OT should be modeled graph-theoretically as a "shortest paths" problem. Ellison showed that if the functions that generate the candidates and the constraints are both regular (representable with finite state machines) then the evaluation of the infinite range of possible output candidates for a given input can be given a concise representation as a single finite state machine that generates and evaluates candidates. The essential premise of this proposal is that optimization should be carried out on the representation of the function that generates and evaluates the candidates, which is finite, rather than on the set of actual candidates, which can be infinite. This essential premise of Ellison's model has been used, explored, and extended by many researchers in a program that one might call Finite State Optimality Theory (FSOT).

In reviewing various computational models of Optimality Theory, I'll focus on three properties in particular. The first of these properties is the nature of optimization. Optimization in many models is "bounded" in the sense that the grammar is only sensitive to differences in the numbers of violations for various candidates when those numbers fall below a preset upper bound. This has obvious empirical consequences, because the grammars cease to adjudicate between candidates whenever the violations for the optimal candidate exceed the preset upper bound.

The second property that I will attend to is the use of a priori restrictions on the candidate set. Many models restrict the action of the candidate generator, GEN, so that

only finitely many candidates may be generated for a given input. Sometimes this move is intended to allow brute force optimization via exhaustive search of the finite candidate set and other times this move is intended mainly to simplify the computation. If the restriction on the candidate set is principled – that is, if it provably reflects the action of undominated constraints then there will be no loss of empirical coverage. If, on the other hand, the restriction on the candidate set is just a stipulation that guarantees finitude then there will be a loss of empirical coverage for cases where the truly optimal candidates lie outside the finite candidate set.

The third property that I'll attend to is the way that optimization is carried out in the various models. Some models (like Ellison's) evaluate candidates with respect to all constraints simultaneously and other models use cascades of evaluators, one for each constraint. Though this aspect of the models doesn't have serious empirical ramifications, I'll show here that this difference in the representational scheme for the grammars does have significant ramifications with respect to how much information can be extracted from a single derivation. Specifically, I'll show that the use of a monolithic evaluator for all constraints simultaneously makes it possible to generate the set of optimal candidates for all possible rankings in a single derivation.

Throughout this work I'll focus solely on Optimality Theoretic phonology, and completely set aside any consideration of OT approaches to syntax or semantics. Though there is some excellent computational analysis of OT as applied to these domains (e.g. Jäger 1999, 2000), the restriction to phonology will help rein in the set of issues under consideration here.

1.2 FSOT models and variants

Eisner (1997a, 1997b, 1997c, 1999) adopts Ellison's assumptions that constraints are finite state, that there's no bound on the number of violations to which constraints are sensitive, and that there are no a priori bounds on (the size of) the candidate set. Eisner departs from Ellison's model in two central respects.

Eisner formulates a model which he calls Primitive Optimality Theory (OTP). In OTP Eisner takes Goldsmith's (1976, 1990) autosegmental representations and strips out the association lines, instead representing features as stacked overlapping constituents on a set of independent tiers that are synchronized by reference to a "constituent time-line". Thus the constraints reference the overlap of features across the various tiers in a gestural-score-like representation (c.f. Browman and Goldstein 1989). In OTP constraints are constructed from a limited set of primitives and fall into one of two classes depending on whether they dictate that prosodic/morphological/featural constituents must or must not overlap temporally across the tiers.

The second point of departure in Eisner's model is the introduction of a "cascade" approach to evaluation in which the constraints are applied sequentially from highest to lowest ranked as a series of filters that winnow out suboptimal candidates. Eisner introduces the cascade approach to rein in the amount of computation by avoiding the need to intersect all of the constraints and GEN to produce a single large evaluator. This is especially necessary in Eisner's tier-based model because the application of the generating function GEN to a particular input yields a finite state machine with as many as is $2^{|\mathbf{Tiers}|}$

states. This can be quite large because there are distinct tiers for prosodic constituents, features, and morphological constituents.

Eisner (1997b: 6-7) proves that optimization in his model is NP-hard in the number of tiers. That is, the amount of computation required to do optimization is worse than a polynomial function of the number of tiers. It is important to note, however, that this hardness result is for the complexity of the representations (the number of tiers) and not for the number of constraints or the size of the inputs to be optimized. Though Eisner acknowledges that in a fixed grammar this complexity is reduced to a constant factor, he observes that the constant factor might be prohibitively large and that this complexity will become highly relevant in the learning scenario if learners are free to hypothesize grammar models with varying numbers of tiers.

It's not surprising that the optimization problem in OT should be intractable in the dimension of representational complexity. Fortunately, however, even if we retreat from the position that the constraint set is fixed and universal, it would still be plausible to maintain the position that the set of building blocks (be they tiers or something else) that make up the representations over which the constraints are stated is finite and relatively small. This seems a reasonable position given that the search for simple and elegant representations has always been a central theme in linguistic theory. If, in modeling learning, we retreat even further and allow the learner to hypothesize arbitrary constraints over representations of arbitrary complexity, we still aren't necessarily sunk. If the difficulty in using a model for generating outputs helps determine the attractiveness of that model for the learner then the increase in computational complexity could actually be

seen as a pressure on the learner to pursue models with simpler representations. These issues and conjectures are well beyond the scope of this dissertation as I will be assuming a fixed (non tier-based) representational scheme and a fixed universal set of constraints. What I'll be more concerned with are issues of computational complexity in a range of "typical" OT analyses and how to make the representations and the grammar as efficient as possible in such cases.

Albro (1998a, 1998b) explores and extends OTP, providing an analysis of Turkish vowel harmony and disharmony and introducing to OTP a method for formulating faithfulness constraints that can be satisfied at a distance, a representation for disjunction of constraints (Crowhurst and Hewitt 1997), and a representation for conjunction of constraints (Smolensky 1995). Albro (2000) augments the basic finite-state approach to OT with a non-finite-state module designed to be able to capture the phenomenon of reduplication and implement the Base-Reduplicant correspondence constraints of Correspondence Theory (McCarthy and Prince 1995). Albro (2003) uses this extension to provide a large-scale implementation of an OT analysis of the phonology of Malagasy (including reduplication).

Frank and Satta (1998) propose a variant of OT in which every constraint makes only a binary distinction between candidates that are violators and candidates that are not. It's important to distinguish this from Ellison's (1994) restriction that each mark laid down by a constraint be 1 or 0, which nonetheless allows a single constraint to be violated multiple times. To encode something like multiple violability in Frank and Satta's model several constraints are used: one for a single violation, another for two violations, and so

on up to a constraint for ***n*** violations. One result of this restriction is that the grammar won't make distinctions among candidates with more than ***n*** violations.

Frank and Satta prove that with the binarity restriction in place OT grammars are limited so that they can only define rational i/o relations (relations that are representable with finite state machines). Following a idea from Marcus Hiller, they observe that, without the binarity restriction, optimization for a single constraint can turn a rational relation into an irrational one. To see how this can occur, imagine that GEN mapped underlying forms of ***a*b**** (some number of ***a***'s followed by some number of ***b***'s) to either themselves (the identity map) or to surface forms in which ***a***'s are replaced with ***b***'s and ***b***'s are replaced with ***a***'s (the unfaithful map). Optimization for a constraint penalizing each occurrence of the segment ***a*** would result either the identity map if there were more ***b***'s than ***a***'s in the input, the unfaithful map if there were more ***a***'s than ***b***'s, or both mappings just in case there were equal numbers of ***a***'s and ***b***'s in the input string. Relations involving this kind of unbounded counting are strictly beyond the power of finite state machines and therefore the addition of the ****a*** constraint has rendered the relation irrational.[1]

Eisner (2002) observes that this same kind non-rational relation can arise when agreement constraints interact with faithfulness to generate an unattested pattern that Baković (1999, 2000) calls "majority rule". Specifically, if a string of surface segments is obliged to agree on the value α for some feature *f* and the only thing that dictates which value is chosen is the action of faithfulness constraints, then the relative prevalence of the

[1] To state this a bit more technically "[a] relation *R* that realizes such a function is not rational, since its right restriction to the regular language $\{ a^n b^m \mid n, m \in \mathbb{N}\}$ does not have a regular left projection, namely $\{a^n b^m \mid n \geq m\}$" (Frank and Satta 1998:8).

of $+f$ and $-f$ features in the input will dictate which surface value is realized. As with Hiller's case, optimization in this kind of harmony system gives rise to an unbounded counting dependency and thus defines a relation that is irrational. I'll return to this issue in §7.5 but for now I'll simply observe that many phonologists claim that this kind of unbounded counting dependency is not seen in human phonology (e.g. Johnson 1972, Koskenniemi 1983, Kaplan and Kay 1981 1994, Karttunen, Koskenniemi, and Kay 1987).

Karttunen (1998) capitalizes on Frank and Satta's (1998) proposed violation-bound to formulate an OT model in which the computation of optimal forms uses strictly finite state means without invoking any non-finite techniques like the dynamic programming used in shortest-paths algorithms. To achieve this effect Karttunen uses a cascade of binary finite state constraints as filters to successively winnow out suboptimal candidates. His constraints are binary in Frank and Satta's sense – there's a constraint against one violation, another against two violations and so on up to a bound of $\boldsymbol{n}$ violations. To use the constraints as filters Karttunen formulates an operation called "lenient composition." When a constraint is leniently composed with a machine representing the candidate set either all violators are eliminated or, if eliminating all violators would eliminate all of the candidates, nothing is eliminated. Successively leniently composing constraints of varying degrees of strictness (the 1-violation version, the 2-violation version, etc.) with the candidate generator yields a machine that produces candidates with the minimal number of violations. Karttunen notes that, of course, this

only holds when, for all crucial comparisons, the optimal candidate gets fewer violations than the preset upper bound built into the constraints.

Karttunen's system is entirely finite state and as such obviously won't give rise to irrational relations – there can't be an unbounded counting dependency if the grammar is only sensitive to ***n*** or fewer violations. There is however, reason to be concerned with the violation bound. The lack of experimental (or even anecdotal) evidence supporting the position that people cease to distinguish levels of violation above some upper bound suggests that violation-bounded models fit poorly with human performance. Of course, in the face of such qualms, one could always suppose that the bound was so high that it was never exceeded in real-world situations. But such a response calls attention to an entirely different problem with the violation bounded model. That is, in order to implement constraints for the increased levels of violation the size of the machines representing the constraints must be multiplied. For instance, encoding the demand that there be no PARSE violations at all requires 66 states while encoding the demand there be fewer than five PARSE violations requires 248 states (Karttunen 1998:15). This suggests that any attempt to set the violation bound so high that it isn't observed in practice might yield machines too cumbersome to use for generating optimal outputs.

Gerdemann and Van Noord (2000) propose a model that is similar to Karttunen's in which they work around the size issue by revising way that the constraints are used to filter out suboptimal candidates. That is, rather than simply killing off candidates with ***n*** constraint violations (as in lenient composition), they propose a model in which the function that generates the candidates is turned into a "comparative" filter that winnows

out suboptimal candidates. Like Karttunen's, Gerdemann and Van Noord's model is completely finite state. To turn a constraint into a filter for suboptimal candidates, they compose the current candidate generator with a constraint, add one or more violation marks to each candidate that it generates, and then turn the complement of the range of this machine into an identity transducer. This filter can then be composed with the composition of the candidate generator and the constraint to eliminate candidates that have supersets of the violations incurred by the candidates with the smallest sets of violations.

Because the filter kills off candidates with supersets of the violations incurred by the optimal candidate, suboptimal competitors will only be eliminated if their violations "line-up" with the violations in the optimal candidate (i.e. the violations occur in the same positions in the string). For this reason Gerdemann and Van Noord call their strategy a "matching" approach. To allow the elimination of sub-optimal candidates whose violations don't match up with the violations in the optimal candidates, they introduce an operation of "mark permutation" in the construction of the filter. This operation allows a present number of violations, ***n***, to be shuffled around in the string when constructing the filter. The bound ***n*** comes from the fact that finite state machines can only implement bounded permutation.

The permutation bound improves on Karttunen's (1998) violation bound because there are grammars where a small amount of permutation allows a true implementation of optimization for arbitrarily large inputs with arbitrarily many violations. Gerdemann and Van Noord call the resulting grammar for such cases an "exact" implementation of OT.

They show that the basic CV syllable theory of Prince and Smolensky (1993: chapter six) presents just such a case. There are, however, two points of concern. In their system Gerdemann and Van Noord limit epenthesis to one at most one segment before each underlying segment and one segment at the end of the word. This is not generally tenable because grammars can require more than one segment of epenthesis. The worry is that adding the possibility for unbounded epenthesis might make it harder to match up violations and thus give rise to (more) cases where the permutation bound is exceeded. The second concern is with the permutation bound itself. As with the violation bounded model, one might claim that the bound is so high as to be unobservable in practice. Alas, also like the violation bounded model, the cost of increasing the bound grows quite rapidly, so this doesn't seem like a generally tenable solution.[2] As with the violation bounded model, when the upper bound (on permutation) is exceeded the model simply fails to distinguish among candidates. Gerdemann and Van Noord call the resulting grammars for such cases "approximate" implementations of OT.

The most tantalizing property of Karttunen's and of Gerdemann and Van Noord's models is that they produce transducers which directly map input forms to optimal output forms without the need to do optimization for each individual input. Because transducers can be inverted this will allow output forms to be mapped to inputs thereby opening the door for doing recognition in OT. Fosler (1996) suggests that a similar result might be obtained in Ellison's (1994) model by adding correspondence constraints (McCarthy and

[2] If it were possible to do so, proving that "reasonable" human phonological grammars never needed more than some particular bound of mark permutation would present a more appealing solution to this problem.

Prince 1994) and using heuristics to search through the candidate space but isn't explicit as to how the recognition is to be implemented.

In pursuit of transducers that directly encode optimal i/o mappings Eisner (2000, 2002) introduces an OT-variant in which all constraints are evaluated directionally. Under directional evaluation, violations closer to one specified edge of a form are strictly worse than all violations further from that edge. The prime advantage of directional evaluation is that it allows the immediate resolution of all conflicts between candidates. This locality property allows the whole system to be recast as a transducer that maps inputs directly to optimal outputs. As noted above, the availability of such transducers will allow efficient generation of outputs and will allow recognition to be done by inverting the transducers. There are, however, some odd empirical predictions that arise in the directional proposal. First, the premise that all constraints are evaluated directionally seems to predict that directional phonological phenomena should be ubiquitous when, in fact, they're relatively rare. Second, as Wilson (2004) points out, the use of directional constraint evaluation in generating harmony patterns with feature spreading, a domain to which it seems ideally suited, predicts odd and unattested harmonic patterns in which features spread towards the edge of the word just in case they can reach the edge, but if they cannot, there is no spreading at all.

The modifications of optimization introduced in the violation-bounded, matching, and directional variants of OT are motivated by the desire to restrict the generative power of OT grammars to rational relations and by the desire to allow phonological grammars to be recast as transducers that directly map input forms to optimal output forms. Though

restricting the generative power of the models and allowing them to perform recognition tasks is indeed a worthwhile goal, each of these variants of OT makes significant changes to the way the theory works and to the range of empirical predictions the theory makes.

The question that must be asked is whether these same results can be obtained without significantly modifying the nature of optimization. In this work I'll argue that the answer to this question is a qualified yes. That is, I'll show that a range of OT grammars define rational relations and can be recast as transducers capable of doing recognition. Rather than fundamentally altering optimization in OT, what I'll propose is an algorithm for constructing transducers that simply fails when given constraints that don't define a rational input/output relation. In chapter seven I'll propose that the absence of irrational relations in phonology might be attributed to the fact that learners fail to make simple generalizations (i.e. fail to construct transducers) that describe languages that arise under the constraint rankings that define irrational relations. In this sense transducer construction can be seen as a filter that produces gaps in the factorial typology of a constraint set. The upshot of this proposal is that optimization will be left intact in the system I'll use here.

1.3 Other computational approaches to OT

There has been too much computational and mathematical analysis of OT for me to comprehensively review all of it here. In this section I will briefly discuss some of the non-finite-state analyses of Optimality Theory that are germane to the proposals that I'll consider in this dissertation.

Tesar (1995ab, 1996b) presents a model of Optimality Theory in which generation is done by optimizing over "parse trees" that are built by matching the input string to the terminals of a context-free position structure grammar. Tesar uses dynamic programming techniques to evaluate sets of competing partial candidate structures with the constraints.

When comparing various computational models of OT, a point that immediately stands out in Tesar's implementation is the use of a principled bound on the amount of epenthesis that can occur in a single candidate. Unlike some researchers, who invoke an arbitrary upper bound, Tesar bounds epenthesis at a single syllable and proves that no optimal candidate could ever require more epenthesis than this in the grammars that he uses for the basic CV syllable theory of Prince and Smolensky (1993: chapter 6). Though this move renders the candidate set finite, it does so in a principled way, by eliminating infinitely many candidates that are guaranteed to be suboptimal.

In chapter six, I'll present a general automatic strategy for obtaining this kind of result with an algorithm that operates directly on the candidate generator and evaluation function to eliminate "harmonically bounded epenthesis" – the addition of epenthetic material that doesn't improve the candidates under any ranking of the constraints. Thus I'll show that rather than limiting epenthesis to "no more than ***n*** segments in each position ***p***" it's possible to limit epenthesis to those additions that can actually improve a candidate.

Walther (1996) uses a scheme akin to Tesar's (1996b) position structure grammar in which context-free grammars delimit the space of possible tree-structures (candidates) that can be built for an input. Walther explicitly rejects Ellison's (1994) proposal for

optimization over the candidate generating function, what Walther calls "an intensional description of the entire candidate set", opting to "firmly reside on the object rather than on the description level." To this end he proposes an algorithm that sorts finite sets of candidates according to their harmony and guarantees that that the sets will always be finite by imposing a bound on the amount of epenthesis allowed in any given candidate.

From this perspective, Walther's (2001) criticism that augmenting OT with the mechanisms of Correspondence Theory (CT; McCarthy and Prince 1995) gives rise to more candidates than there are atoms in the universe makes sense – if the candidate set were infinite to begin with this claim would be meaningless. Walther's computation of the size of the candidate set is, however, somewhat odd. In order to ensure that the set is finite in the first place, he limits the number of epenthetic and reduplicated segments to "a reasonable default value" but then, to compute the range of possible candidates, he allows arbitrarily fusion, fission, permutation, deletion, and changes to segments. The permutation alone multiplies the candidate set by a factor of $n!$ for inputs of length n. The candidate set would shrink drastically if the same kind of "reasonable" restriction was placed on permutation (to the inversion of pairs of adjacent segments for instance). An even more drastic reduction would be obtained if all unfaithful mappings were subject to the same sort of "reasonable" restriction. This is not to say that restricting Correspondence Theory to "reasonable" operations would make a brute-force search of the candidate space appealing. If anything, this foray into combinatorics should show the undesirability of brute-force approaches to optimization. I will come back to this point in §1.5, but first I'll briefly mention two more OT models that make crucial use of finite candidate sets.

Heiberg (1999) presents a computational model of Optimality Theory specifically designed to handle constraints on autosegmental representations (Goldsmith 1976, 2000). In her system the candidate set is always guaranteed to be finite (and relatively small) by virtue of the action of the Obligatory Contour Principle (Leben 1973, McCarthy 1979, 1986) and because she doesn't allow epenthesis. For generation Heiberg's model winnows suboptimal elements out of the finite candidate set with a cascade of optimizations one constraint at a time down through the ranking hierarchy.

Hammond (1995, 1997) presents a computational model for a fragment of OT that is designed to do syllabification. Hammond's model keeps the candidate set finite (and rather small) by ignoring deletion and epenthesis. Suboptimal candidates are eliminated from the finite candidate in a cascade-style evaluation one constraint at a time down through the constraint hierarchy.

1.4 Learning and harmonic bounding

Tesar (1995ab, 1996ab, 1997ab, 1998, 2000), Tesar and Smolensky (1998, 2000), and Prince and Tesar (1999) discuss the problem of inferring constraint rankings in OT from observed input/output pairs (and sometimes from outputs alone). To tackle this problem they propose a range of algorithms that rank constraints in a manner that is consistent with observed data. These algorithms constitute a variety of refinements and extensions of Tesar's original (1995a b) recursive-constraint-demotion (RCD) algorithm.

Of particular interest to the work in this thesis, Tesar (2000) shows how RCD can be used to efficiently detect inconsistency among sets of ranking arguments. This will

become quite relevant in chapter five where I'll incorporate Tesar's strategy into Ellison's (1994) finite state model to make it possible to detect and eliminate all the harmonically bounded candidates in a single derivation and thus produce exactly the set of candidates that can win under some permutation of the constraints.

An alternative strategy for ranking constraints consistently with observed data is presented in Boersma and Hayes (2001) and Boersma (1997, 1998, 2001). Boersma and Hayes propose a constraint ranking method that they call the Gradual Learning Algorithm (GLA) that responds to observed i/o pairs by perturbing the ranking of constraints along a continuous scale (see also Hayes and MacEachern 1998 and Hayes 1999). This model has the advantage that it is robust in the face of noise (errors) and can capture "free variation" by allowing constraints with overlapping probability distributions on the continuous scale.

Attempting to rank constraints in a manner consistent with observations is made difficult by the fact that the data can be highly ambiguous. Both RCD and the GLA must contend with the fact that a single datum (i/o pair) can often be explained by any member of a disjunction of partial orderings of the constraints. In this work I'll argue that the focus on actual constraint rankings is unnecessary. Instead, I'll propose an alternative model of learning in OT in which the learner's hypotheses are stated and manipulated using exactly the kinds of disjunctions of partial orderings that arise in observations.

Prince and Smolensky (1993) show that when one candidate α has a strict superset of the violations incurred by another candidate β there is no constraint ranking under which α can ever triumph over β. In this state of affairs α is said to be "harmonically bounded" by β. Samek-Lodovici and Prince (1999, 2002) develop and extend this notion to include

cases where a α is "collectively bounded" by the presence of a set of competitors even when α is not harmonically bounded by any member of the set alone. In this work I will refer to candidates that are not harmonically bounded as "contenders." Contenders play a crucial role in the model of learning in OT that I'll present here, for they constitute exactly the set of candidates whose failure is most informative to the learner.

Prince (2002a b) presents a scheme for encoding information about constraint rankings in the form of an Elementary Ranking Condition (ERC). This representational scheme based on Prince's (1998, 2000) proposed reformulation of tableau notation and encodes the same information as Tesar's (1995a *et seq.*) "mark-data pairs". ERCs have two properties that will make them vital to the work presented here. First, they give us a concise representation of information about constraint rankings and can be manipulated and combined in a variety of ways to draw inferences, and second, an RCD-like algorithm can be used to efficiently identify candidates that are harmonically bounded by detecting "inconsistency" among sets of Elementary Ranking Conditions (see chapter six).

1.5 My proposal

The system I develop here will be based essentially on Ellison's original finite state model of OT. I won't adopt Frank and Satta's (1998) binarity restriction on constraint evaluation, Walther's (1996) bound on epenthesis, Eisner's (2001) directional evaluation, or Gerdemann and van Noord's (1998) violation matching and permutation technique. Instead of a cascade of evaluators for the various constraints I'll build one monolithic evaluator that evaluates candidates with respect to all constraints simultaneously. Finally,

I'll state constraints directly over segments rather than features, tiers, or autosegmental representations. I summarize these properties below in (1).

(1) Basic properties of my model:
   a) no epenthesis bound – the candidate set is infinite
   b) no violation bound – constraints assess arbitrarily many violations
   c) standard evaluation – evaluation is neither directional nor matching
   d) monolithic *Eval* – constraints are applied in one fell swoop (not a cascade)
   e) segmental representations – not tier-based or autosegmental

Properties (a) through (c) above constitute substantive empirical claims about the nature of (computational) Optimality Theory. Properties (d) and (e) are merely implementational but they nonetheless have significant ramifications for the complexity of the computations and for the amount of information that can be extracted from a single derivation. I'll go through these points one at a time below.

While epenthesis-bounded models may serve to illuminate some phonological phenomena and types of constraint interaction, as a general model of OT the use of an arbitrary upper bound on epenthesis is linguistically unenlightening. Moreover the hope that such a bound could make a brute-force search of the candidate space tenable in the general case is doomed in the face of the combinatory complexity that can arise with even a finite set of options. In chapter six I'll show how the action of constraints like DEP guarantee the finitude of the set of contenders (those candidates that are not harmonically bounded). I'll argue that restricting our attention to the relevant candidates, the contenders, is ultimately more sensible (and more efficient) than placing restrictions like epenthesis

bounds directly on the candidate generator. Crucially, the issue at hand is not whether the candidate space is infinite, but rather whether the structure of the problem is such that it's possible to formulate an efficient strategy for finding optimal candidates.

I avoid the violation bounded, matching, and directional variants of OT for two reasons. First, they do not seem, in principle, to fit well with the empirical facts. Both the bounded violation set-up and the permutation technique are unable to adjudicate between competing candidates once the upper bound on violations/permutations is exceeded. The directional evaluation variant of OT has different empirical problems as it makes odd (and seemingly unattested) predictions about the prevalence of directional phonological phenomena and generates some odd harmony patterns. The second, and more important, reason that I do not adopt one of these variants of OT in this work is the hope to develop tools that provide insight into the workings of OT grammars as they are typically used by phonologists.

In this work I'll show that the use of a monolithic evaluator that encodes all of the constraints of the grammar has several advantages. Most of the computational proposals described above eschew such a technique out of concern that the representation of the evaluating function might grow explosively as constraints are added to the grammar (e.g. see Eisner 1997b and Albro 1998a for explicit discussion of this idea). I will argue here that this fear is not warranted. I'll show in chapter two that the size of the representation of the grammar resulting from combining all of the constraints is bounded by the number of unique phonological environments to which the grammar is sensitive. While this may

be huge, it isn't on the order of the billions and billions that would make the evaluator too cumbersome to be used in computing optimal forms.

On the contrary, putting all of the constraints together into one evaluating function has several advantageous consequences. Given a fairly concise representation of the entire grammar it is possible to do several things that would not otherwise have seemed possible. First, it becomes possible to do optimization for all rankings simultaneously. That is, rather than generating a single optimal i/o pair for a single ranking it becomes possible to generate the contenders (the non-harmonically bounded candidates) in a single derivation that isn't that much more complex than finding a single optimal i/o pair.

Combining all of the constraints into a single evaluator has other advantages as well. Given such a concise representation of the entire grammar it is possible to detect and eliminate sub-optimal fragments of parses in the evaluation function itself before considering a single input. This process, which I call preoptimization, provides numerous benefits. The most relevant of these benefits is that after preoptimization all subsequent optimization tasks can be done in linear time.

Finally, with the entire grammar represented as a single preoptimized evaluation function it is possible to detect and generalize recurring patterns across the optimization of various input strings and thereby construct a transducer that defines the same input/ output mappings as optimization. Unlike the OT-variants discussed above, this proposal doesn't involve modifying the nature of optimization to guarantee that transducers can be constructed for all rankings, but rather discovers transducers that are equivalent to OT

grammars in cases where such transducers exist. There will still be OT grammars that cannot be represented with transducers; for such grammars the algorithm simply fails.

If it is indeed the case that irrational phonological relations are absent in human phonologies then the process of transducer construction could explain these gaps in the factorial typology without a fundamental change to the nature of optimization in OT.

The final point worthy of comment in my implementation is the character of the phonological representations. Throughout this work constraints will be stated over segments and optimization will be carried out on functions that generate candidates that are strings of segments. Whether autosegmental representations like Heiberg's or tier-based representations like those of OTP ultimately yield substantively different empirical predictions is an issue that I will return to in chapter eight.

1.6 Why model OT computationally?

Optimization problems are notoriously hard. Finding the optimal candidate among an infinite set of possibilities via exhaustive search is obviously not possible. Even if the range of candidates is somehow restricted so that it's finite, if it grows geometrically with the size of the input then an exhaustive search is untenable in the general case.

A plausible response to this state of affairs might be to regard computability issues as an irrelevant distraction in the task of devising illuminating models of human linguistic faculties. Indeed, Prince and Smolensky's statements quoted in (2) might be seen as an instance of this attitude.

(2) Prince and Smolensky 1993: 215-216

> It is not incumbent upon a grammar to compute ... A grammar is a function that assigns structural descriptions to sentences; what matters formally is that the function is well-defined. ... [T]here are neither grounds of principle nor grounds of practicality for assuming that computational complexity considerations, applied directly to grammatical formalisms, will be informative.

Prince and Smolensky's comments in (2) highlight the difference between a well defined function and the algorithm that computes that function. This is especially necessary as a response to the often expressed, yet misguided, notion that the only way to do optimization is to exhaustively search a huge (or infinite) space of possibilities.

Optimization problems are not easy, but all good (tractable) solutions to nontrivial optimization problems involve techniques more subtle than brute-force search. Surely, if we find an algorithmic characterization of OT that is true to the spirit of OT and allows for efficient generation, then at the very least it should be adopted on pragmatic grounds, for without an efficient generation strategy it's impossible to actually test the predictions of the theory for any but the simplest "toy" grammars.

In other words, while the availability of an efficient computational implementation may not be required of phonological models, it is an invaluable asset to researchers using those models because it gives us some hope of understanding the predictions the models make and of understanding how changes to the basic ingredients of the models change those predictions.

Phonological grammars built from competing interacting constraints can become hideously complicated, thereby rendering our intuitions about the effects of a particular

constraint or constraint ranking dubious at best. To support this claim with a personal anecdote, I'm often quite surprised by the predictions the models make once implemented. For instance, having an algorithm generate a vowel harmony-like pattern that is achieved via deletion and epenthesis is initially shocking. Such a pattern could surely be predicted from introspection about the constraints and their interaction, but the vast number of such possibilities all but guarantees that introspection alone will miss interesting predictions. All too often an argument in favor of one OT analysis over another that is based on a few illustrative examples comes crashing down when just one relevant candidate is added to the set of forms being compared. With a fully implemented model of generation in OT it is easy to avoid such calamities.

Algorithmic methods are even more important when it comes to understanding the typological predictions made by the theory. Given that there could, in principle, be factorially many different input/output relations defined by a set of constraints, it is practically impossible to get a good idea of the typological predictions by simply reasoning about the constraint interactions.

Finally, the most relevant advantage conferred by actually implementing a model is that it makes it necessary to be totally explicit about every detail of the mechanics of the system. Though it would seem that when implementing a model one can get bogged down in minutiae, it is often the minute details that are left vague in most models that end up having unexpected global and systemic consequences.

1.7 Computation and computers

The work presented herein is computational in the mathematical sense, not because it requires the use of a computer. Computers serve as vital aids in performing the complex calculations required here but are not an integral part of any of the proposals.

Nonetheless, even the simple systems presented here would test the limits of patience if the calculations were carried out with pencil and paper. All through this work I've sought to use the smallest and simplest possible examples so that it is possible to follow the computations described (and so that the graphs fit on single pages).

Two free software packages have been absolutely invaluable in this work. SWI-prolog is a free distribution of Prolog, the logical programming language in which all of the algorithms presented here are implemented. SWI-prolog is freely available at **`www.swi-prolog.org`**. The graphs used throughout this work to illustrate the finite state machines were drawn with Graphviz. Graphviz is a free graph-drawing tool that is available from AT&T labs at **`www.research.att.com/sw/tools/graphviz`**.

My prolog implementations of the algorithms presented throughout this work can be downloaded from my homepage. Given that affiliations and web-addresses can change, the best way to find my homepage is to follow the link from the alumni page of the UCLA linguistics department at **`www.linguistics.ucla.edu`**.

1.8 Overview

In this work I provide a formal characterization of Optimality Theory in which I address complexity and the learnability issues. I represent the constraints of OT as finite

state transducers and phonological grammars as intersections of these transducers. In this system, a phonological grammar is a function that maps input strings to sets of output strings paired with the numbers of constraint violations incurred by each such mapping.

By putting all of the information of the grammar into one finite representation I am able to formulate the following algorithms.

The CONTENDERS algorithm takes an *unranked* set of constraints and an input and finds the set of input/output pairings that could win under *any* ranking of the constraints. With such information the learner is then able to compare the actually observed form with the entire (finite) set of informative losers.

The Optimality Transducer Construction algorithm (OTCA) takes a ranked set of constraints and from them constructs a transducer that directly generates optimal output forms from underlying forms. In effect, this move obviates the need for word-by-word optimization since all of the optimization is done in the process of the creation of the transducer. This proposal will yield a grammar that does not suffer from a geometric explosion in the amount of work to derive longer forms. Moreover, transducers can be inverted and therefore the grammar will be usable for comprehension (mapping surface to underlying forms) as well as for production.

2 Building Eval

In Optimality Theory (OT; Prince and Smolensky, 1993) the sound pattern of a given language is generated by the interaction of a set of competing universal drives (constraints) that vie to exert their influence on the phonology of the language. As with many phonological theories that preceded it, in OT it is assumed that "underlying forms" that are specified in the mental lexicon serve as the input to the phonology and the job of the phonological grammar is to mediate the relation between the underlying forms and the pronounced "surface forms." In OT this task is stated as an optimization problem. That is, the surface forms that are derived from a given input are those that optimally satisfy a set of ranked constraints governing the input-output relation. Optimal satisfaction of the ranked constraints is achieved by a given input-output pair, $\langle i, o \rangle$, just in case for any other possible output o' such that some constraint C_1 prefers $\langle i, o' \rangle$ over $\langle i, o \rangle$, there is another constraint C_2 ranked higher than C_1 that prefers $\langle i, o \rangle$ over $\langle i, o' \rangle$. While it is fairly easy to concisely state what it means for a given output to be optimal, the task of finding an optimal output among the infinite range of possible outputs can in principle be quite hard.[3]

One way to ensure that the input-output relation defined by the phonological grammar is a computable one is to restrict the formal power of the constraints that make up the grammar. In this chapter I'll present a general scheme for representing constraints on the input-output relation with finite state machines that encode functions from input-

[3] Or even impossible. If, for example, constraints are allowed to be arbitrary functions from i/o pairs to numbers of violations then the use of noncomputable functions as constraints would result in a noncomputable grammar.

output mappings to violations. I will begin by presenting a single constraint and showing how it can be used to evaluate the range of possible output candidates for a given input. I will then present an operation that combines a set of constraints into a single finite state machine defining a function that evaluates the candidates with respect to the whole set of constraints simultaneously. I'll call the combined set of constraints *Eval*. In this chapter I will illustrate the basic mechanics of this system with some very simple constraints. I will close the chapter by showing that *Eval* can be treated just like an individual constraint in evaluating the range of possible output candidates for a given input.

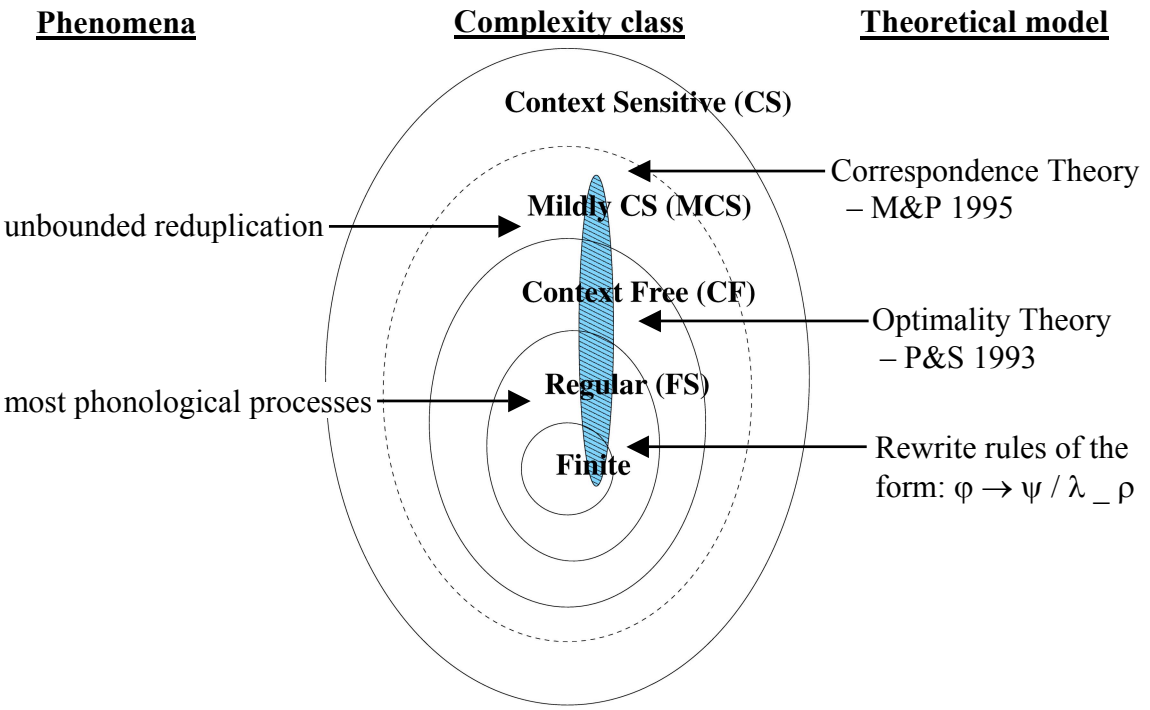
2.1 The finite state restriction

Ellison (1994) showed that if OT constraints are required to be regular (i.e. finite-state transducers that map $\langle i, o \rangle$ pairs to numbers of violations) then optimal outputs can be computed by representing the set of possible parses of an input as a graph and using a shortest-paths algorithm to find the output candidates that minimally violate the ranked set of constraints. This approach has been taken up in much subsequent research in finite-state Optimality Theory (Eisner 1997a, 1997b, 1997c, Frank and Satta 1998, Karttunen 1998, Albro 1998a, 2000, Gerdemann and Van Noord 2000, and others).

Unlike some other finite-state implementations of Optimality Theory, in this work I'll attempt to retain all the basic assumptions of OT with the single caveat that constraints must be representable as finite state machines. The motivation here is to explore the basic computational properties of standard Optimality Theory, not to find some approximation or modification of OT that has desirable computational properties. Given the finite state

caveat on the theory it's worth trying to understand what it buys us and what phenomena it allows us to describe. In (3) I lay out Chomsky's hierarchy of language complexity and situate a couple of phonological phenomena and three theoretical models in the hierarchy.

(3) Chomsky hierarchy of language complexity:



Kaplan and Kay (1994) show that rewrite rules of the form $\varphi \rightarrow \psi / \lambda \_ \rho$ can only define rational relations if the rules aren't allowed to recursively rewrite their own output. This covers directional iterative application, simultaneous application, and application in any finite number of cycles.[4] Rule-based phonological models are often touted as fitting well with the facts because most phonological processes seem to fall squarely in this class of relations. Grammars using optimization with ranked violable constraints (as in OT) are

[4] The only potential for irrational relations with rules comes if cycles can be iterated without bound. Persistent rules as proposed in Myers (1991) would provide an architecture where this could occur.

slightly more powerful because optimization can be used to generate unbounded counting phenomena (Frank and Satta 1998). The phenomenon of reduplication is slightly more complex still, requiring something like the constraints of Correspondence Theory (McCarthy and Prince 1995).

In this work, all of my constraints will be expressed as finite state machines. This restriction will put the generative power of the model presented here at the level of Prince and Smolensky's original (1993) model of Optimality Theory. In chapter seven I'll return to the issue of generative power and make a proposal for restricting the generative capacity of my model to just the rational relations. For the time being, however, I'll present my basic system with the acknowledgement that there are ways in which it over-generates (by defining irrational relations) and ways in which it under-generates (by failing to model reduplication).

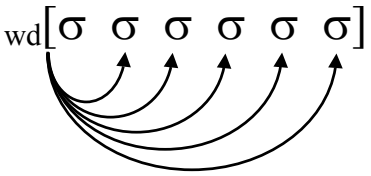
The imposition of the finite-state restriction on constraints rules out a handful of non-regular constraints in the OT literature. For instance, consider in (4) and (5) two alignment constraints using McCarthy and Prince's (1993) generalized alignment schema.

(4) ALL-FEET-LEFT:
The left edge of *every* foot must be aligned to the left edge of a prosodic word. (McCarthy and Prince 1993, 1994)

(5) ALL-σ-LEFT:
The left edge of *every* syllable must be aligned to the left edge of a prosodic word. (Mester and Padgett 1994)

These constraints aren't regular because they impose overlapping "counting dependencies," measuring the distances between multiple syllables/feet and the edge of the word. In (6) I illustrate evaluation with the constraint ALL-σ-LEFT.

(6) Evaluation with ALL-σ-LEFT:



The second syllable gets one violation (it's misaligned from the left edge by one syllable), the third syllable gets two violations, the fourth gets three violations, and so on. The total violations for the form in (6) are 1+2+3+4+5 = 15. It is easy to see how the violations will keep climbing as more syllables are added to the form.

Eisner (1997b) and Bíró (2004) discuss the fact that that this type of constraint which Bíró calls a "quadratic" alignment constraint cannot be modeled with finite state means.[5] Put simply, finite state methods fail here because there can be arbitrarily many syllables in a word, and thus there is no bound on how many simultaneous measurements might need to be made.

Because the constraints in (4) and (5) are generally better satisfied by forms with fewer syllables/feet, they have been used to act as "size restrictors" in reduplication (cf. Spaelti 1997). Eliminating alignment constraints of this type doesn't seem to pose a problem for the empirical coverage of the theory as there exist strategies for restricting reduplicant size and requiring adjacency of elements that don't rely on non-regular

[5] For a form with n syllables there will be $(n^2 - n)/2$ violations.

constraints. Moreover, there may be empirical reasons to eliminate such non-regular constraints as they have come under attack recently for problems with over-generation (cf. Eisner 1997b, McCarthy 2002).

The finite state restriction also makes it impossible to express the base/reduplicant faithfulness constraints of Correspondence Theory (McCarthy and Prince 1995). With their ability to enforce unbounded copying in reduplication, such constraints can't be evaluated with finite state machinery.[6]

For this initial examination I will also ignore the possibility of segments being reordered in the input-output relation (metathesis) because unbounded permutation of the input segments can't be described with finite-state methods. Note, however, that metathesis that inverts pairs of segments (or metathesis with a fixed upper bound) can be described with finite state methods.

The question of exactly what class of phonological phenomena the finite-state restriction excludes from consideration is an interesting one but one I won't go into here. For now I'll simply accept the possibility that the finite state restriction may exclude some interesting phonological phenomena from analysis but press onward because it seems a reasonable enough place to start.

[6] For discussion of how to incorporate some non-finite-state components to handle reduplication into an otherwise finite state implementation of OT, see Albro (2004).

2.2 Finite-state constraints

For this initial illustration of the mechanics of my implementation of Finite state Optimality Theory I will use the simple hypothetical language described in (7).[7]

(7) **Baa:**

Baa has only two phonemes, one consonant 'b' and one vowel 'a'. Baa does not permit consonant-consonant sequences in surface forms (vowel-vowel sequences are okay). Baa speakers avoid uttering CC-clusters that occur in underlying forms by deleting consonants.

To further simplify this scenario, I will assume that the only changes that can be made in mapping the input (the underlying form) to the output (the surface form) are the deletion of underlying segments and the insertion of new segments not present in the input. Even though this set up is simplified in the extreme, it is still of sufficient complexity to illustrate the basic mechanics of Optimality Theory.

All regular OT constraints can be characterized as finite state transducers that evaluate candidates by assigning violations to certain input-output mappings. Basically, each constraint is a function that maps input-output mappings to the positive integers. Finite state constraints can be described formally as in (8).

[7] I borrow this type of simple scenario from Prince and Smolensky's (1993) work but use 'b' and 'a' as the symbols in the languages under consideration in place of the abstract markers 'C' and 'V'.

(8) Finite state constraints are 5-tuples: $(Q, \Sigma, \delta, q_0, F)$ where

i) Q is the nonempty set of states in the constraint,

ii) Σ is the set of symbols in the language,

iii) δ is a transition function describing the input-output-violation triples that label the arcs leading from one state to the next: $\delta \subseteq Q \times \Sigma^+ \times \Sigma^+ \times \{0,1\} \times Q$, where Σ^+ is the union of Σ with the special symbols {-, •} which denote the empty string and a wildcard that will be explained below,

iv) $q_0 \in Q$ is the unique start state, and

v) $F \subseteq Q$, is the nonempty set of final states.

The easiest way to get an idea of what a given finite state constraint does is to examine its graphical representation. Consider in (10) an illustration of the finite-state version of the familiar constraint DEP, which penalizes epenthesis by assigning one violation per segment that is added to the output (McCarthy and Prince 1995).

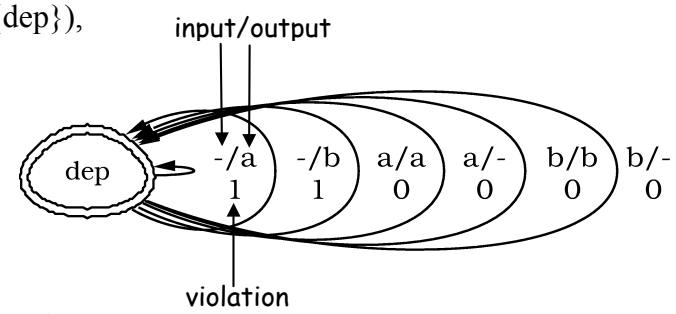
(9) **DEP**: every segment in the output must have an input correspondent

(10) **DEP** = ({dep}, {a, b}, δ, dep, {dep}),

(states, alphabet, arcs, start, finals)

δ = { (dep, -, a, 1, dep),
(dep, -, b, 1, dep),
(dep, a, -, 0, dep),
(dep, a, a, 0, dep),
(dep, b, -, 0, dep),
(dep, b, b, 0, dep)}.

arc = (origin, input, output, cost, terminus)



As notational conventions in the graphs presented in this work the start state is egg-shaped, the final states are double circled, and the dash "-" is used to represent the empty string.[8] The set δ makes up a list of the arcs in the machine. Each arc is a 5-tuple that begins with the state that is the **origin** of the arc, ends with the state that is the **terminus** of the arc, and whose middle three terms make up the **label** of the arc. The three terms constituting the label of each arc are an **input**, an **output**, and a **cost**, where the cost expresses the number of constraint violations that are incurred in mapping the input to the output.

2.3 Generating the candidates

To use a constraint, like DEP in (10), to evaluate candidates, a representation of the candidate set is needed. To build the candidate set we can simply extend the construction of the finite state representation for an input string given in (11).

(11) **Input acceptor:**

$A(s_1 \ldots s_n) = (\{0, \ldots, n\}, \Sigma, \delta, 0, \{n\})$, where

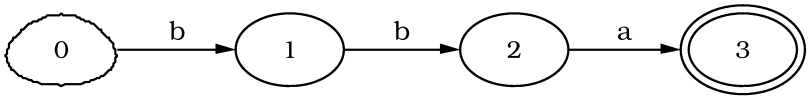
$\delta = \{(q, s_i, \bullet, \langle\rangle, i) | \text{either } 0 \leq q = i \leq n \text{ or } s_i \in s_1 \ldots s_n, \text{ and } q = i - 1\}$

The arcs defined by δ in (11) are described with (origin, input-symbol, •, ⟨⟩, terminus) 5-tuples. The use of the wildcard • as the output term will allow the labels on the arcs of the input acceptor to unify with arcs of other machines that share a common input – I'll explain this below. The 4th coordinate on each arc is the empty sequence ⟨⟩. This is where violations will be recorded once we add constraints to the picture.

[8] I reserve epsilon to represent open-mid front vowels.

The linear acceptor for the input string /bba/ is a simple machine with four states {0, 1, 2, 3}, one arc per input segment, and one arc looping from each state to itself accepting the empty string. The start state "0" corresponds to none of the input having been accepted and the final state "3" indicates that the entire input string has been accepted. In general when graphing constraints and input acceptors I won't include arcs that loop from states to themselves accepting empty strings, the wildcards for undefined segments or the empty cost vectors because they don't contribute any unique information. The acceptor for the input string /bba/ is given in (12).

(12) A(bba):

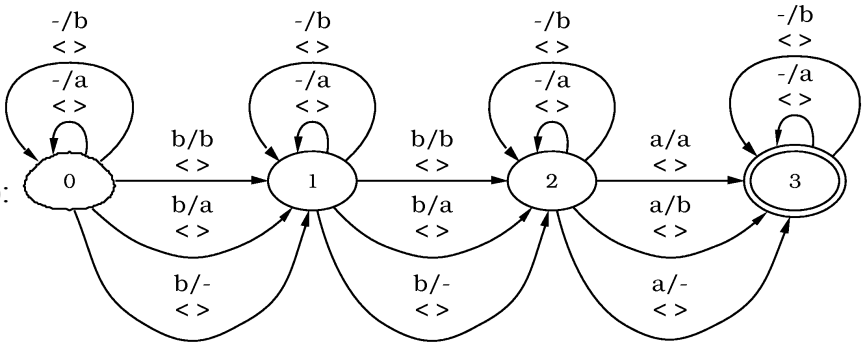


(states, alphabet, arcs, start, finals)

$A(\text{bba}) = (\{0, 1, 2, 3\}, \{a, b\}, \delta, 0, \{3\})$, $\delta = \{(0, -, \bullet, \langle\rangle, 0), (0, b, \bullet, \langle\rangle, 1),$
$(1, -, \bullet, \langle\rangle, 1), (1, b, \bullet, \langle\rangle, 2),$
$(2, -, \bullet, \langle\rangle, 2), (2, a, \bullet, \langle\rangle, 3),$
$(3, -, \bullet, \langle\rangle, 3)\}$

To obtain a finite representation of the infinite set of possible output candidates for a particular input string, the wildcards in the output coordinates of the arcs of the input acceptor can be filled in with each possible output symbol in Σ. I'll call the machine with this information filled in $g(x)$ or "GEN of input x." For example, GEN of input /bba/ is represented below in (13).

(13) g(bba):



- to be maximally explicit I've marked the empty cost-slots on the arcs

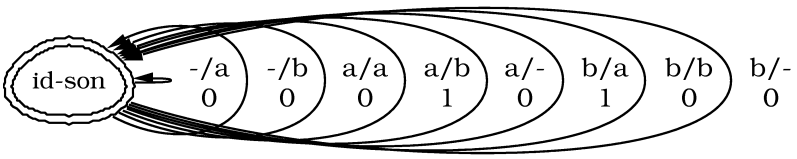
g(bba) = ({0, 1, 2, 3}, {a, b}, δ, 0, {*3*}), where

δ= {(0, -, a, ⟨⟩, 0), (0, -, b, ⟨⟩, 0), (0, b, b, ⟨⟩, 1), (0, b, a, ⟨⟩, 1), (0, b, 1, ⟨⟩, 1),
(1, -, a, ⟨⟩, 1), (1, -, b, ⟨⟩, 1), (1, b, b, ⟨⟩, 2), (1, b, a, ⟨⟩, 2), (1, b, 1, ⟨⟩, 2),
(2, -, a, ⟨⟩, 2), (2, -, b, ⟨⟩, 2), (2, b, b, ⟨⟩, 3), (2, b, a, ⟨⟩, 3), (2, b, 1, ⟨⟩, 3),
(3, -, a, ⟨⟩, 3), (3, -, b, ⟨⟩, 3)}

This machine represents an infinite set of i/o mappings from input string /bba/ to outputs drawn from {a, b}*. Every path from the start state to a final state is a pairing of /bba/ with a particular output string.

In this initial examination the only unfaithful i/o mappings that I'll consider will be those involving deletion and insertion. To restrict the candidate set in this way, an undominated faithfulness constraint can be used. Consider in (14) a member of McCarthy and Prince's (1995) IDENT family of constraints which militates against input-output mappings that change the value of the feature [± sonorant].

(14) IDENT(SON):

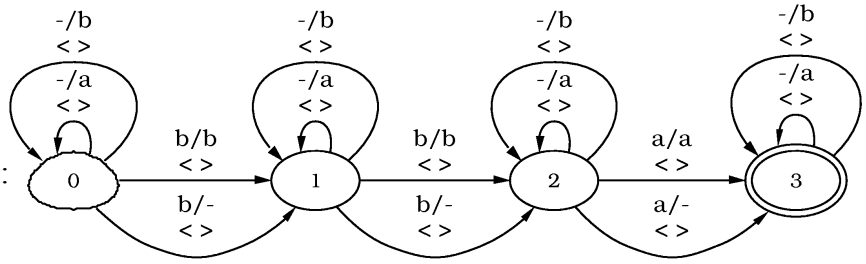


IDENT(SON) = ({id-son}, {a, b}, δ, id-son, {id-son})

δ = { (id-son, -, a, 0, id-son), (id-son, -, b, 0, id-son), (id-son, a, a, 0, id-son), (id-son, a, b, 1, id-son), (id-son, a, -, 0, id-son), (id-son, b, a, 1, id-son), (id-son, b, b, 0, id-son), (id-son, b, -, 0, id-son)}

If IDENT(SON) is ranked at the top of the constraint hierarchy then it will always be obeyed and only deletion, insertion, and faithful parsing will be allowed. I'll present a method for encoding undominated constraints in the grammar in chapter five; for now I'll simplify things by omitting arcs that violate IDENT(SON) from the machines under consideration. The candidate set generated for the input string /bba/ with this restriction in place is represented in (15).

(15) g(bba):



The machine in (15) encodes every possible mapping from the input string /bba/ to an output string where each input segment may be faithfully parsed or deleted and any amount of epenthesis is allowed.

This same technique can be used to represent output strings. A minor modification of the definition in (11) produces the definition of output acceptors given in (16).

(16) **Output acceptor:**

$A_{out}(s_1 \ldots s_n) = (\{0, \ldots, n\}, \Sigma, \delta, 0, \{n\})$, where

$\delta = \{(q, \bullet, s_i, \langle\rangle, i) | \text{either } 0 \le q = i \le n \text{ or } s_i \in s_1 \ldots s_n, \text{ and } q = i - 1\}$

The thing that distinguishes input from output acceptors is the position of the wildcards. Specifying every arc label in every machine with all three values input, output, and cost gives all the machines the same structure and helps keep the values straight.

2.4 Evaluating candidates

The machine $A(in)$ encodes input string *in*. To evaluate output candidates for *in* with a constraint, $A(in)$ is used to "restrict" the range of *i*/*o* pairs under consideration. To impose this restriction $A(in)$ is "intersected" with a the machine for the constraint. This operation, which I'll call "machine intersection" or simply *M*-intersection is defined in (18). First, in (17), I define the unification of symbols so that anything unifies with itself or the wildcard and everything else unifies to zero.

(17) ***def***: symbol unification

$$x \sqcup y = \begin{cases} x \text{ if } x = y \\ y \text{ if } x = \bullet \\ x \text{ if } y = \bullet \\ 0 \text{ otherwise} \end{cases}$$

(18) ***def***: *M*-intersection

$\langle M_1, M_2 \rangle^{\otimes} = M_3$ for $M_1 = (Q_1, \Sigma_1, \delta_1, S_1, F_1)$ and $M_2 = (Q_2, \Sigma_2, \delta_2, S_2, F_2)$,

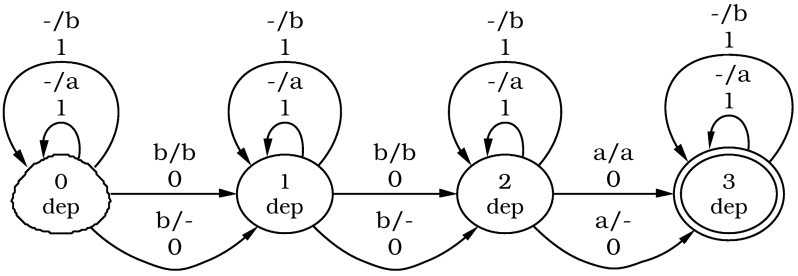
$M_3 = (Q_1 \times Q_2, \{\Sigma_1 \cup \Sigma_2\}, \delta, S_1 \times S_2, F_1 \times F_2)$ where

$\delta = \{(qx, i, o, \langle v, w\rangle, ry) \mid (q, i_1, o_1, v, r) \in \delta_1, (x, i_2, o_2, w, y) \in \delta_2,$

$i = i_1 \sqcup i_2 \neq 0, \text{ and } o = o_1 \sqcup o_2 \neq 0\}$

I call the ⊠ operation *M*-intersection because it creates an arc in the new machine for each pair of arcs with unifiable labels in the old machine. This operation isn't quite intersection in the classical sense because the cost vectors labeling the arcs are built up by concatenation as constraints are intersected – the operation isn't commutative because the ordering in the cost vectors depends on the order in which machines are combined via *M*-intersection. This operation does essentially the same thing as Ellison's (1994) algorithm for generating the "augmented product" of two machines. I illustrate *M*-intersection with *A*(bba) and DEP in (19).

(19) $\langle A(\text{bba}), \text{DEP}\rangle^{\boxtimes}$:



$\langle A(\text{bba}), \text{DEP}\rangle^{\boxtimes} = (\{0\text{dep}, 1\text{dep}, 2\text{dep}, 3\text{dep}\}, \{a,b\}, \delta, 0\text{dep}, \{3\text{dep}\})$,

where $\delta =$ {(0dep, -, a, ⟨1⟩, 0dep), (0dep, -, b, ⟨1⟩, 0dep), (0dep, b, b, ⟨0⟩, 1dep), (0dep, b, -, ⟨0⟩, 1dep), (1dep, -, a, ⟨1⟩, 1dep), (1dep, -, b, ⟨1⟩, 1dep), (1dep, b, b, ⟨0⟩, 2dep), (1dep, b, -, ⟨0⟩, 2dep), (2dep, -, a, ⟨1⟩, 2dep), (2dep, -, b, ⟨1⟩, 2dep), (2dep, b, b, ⟨0⟩, 3dep), (2dep, b, -, ⟨0⟩, 3dep), (3dep, -, a, ⟨1⟩, 3dep), (3dep, -, b, ⟨1⟩, 3dep) }.

Because the costs on the arcs of *A*(bba) are empty, when it is intersected with DEP and the costs on the arcs are concatenated, only the costs from the arcs of DEP show up on the arcs of the intersected machine.

Each path through (19) encodes the evaluation of a single candidate with respect to the constraint DEP. To describe this the definitions in (20) will be useful.

(20) **Paths and machines:**

a. A **path** is a sequence of arcs $\langle a_1, \ldots, a_n \rangle$, where for each pair of arcs a_i and a_{i+1} the terminus of a_i is the origin of a_{i+1}.

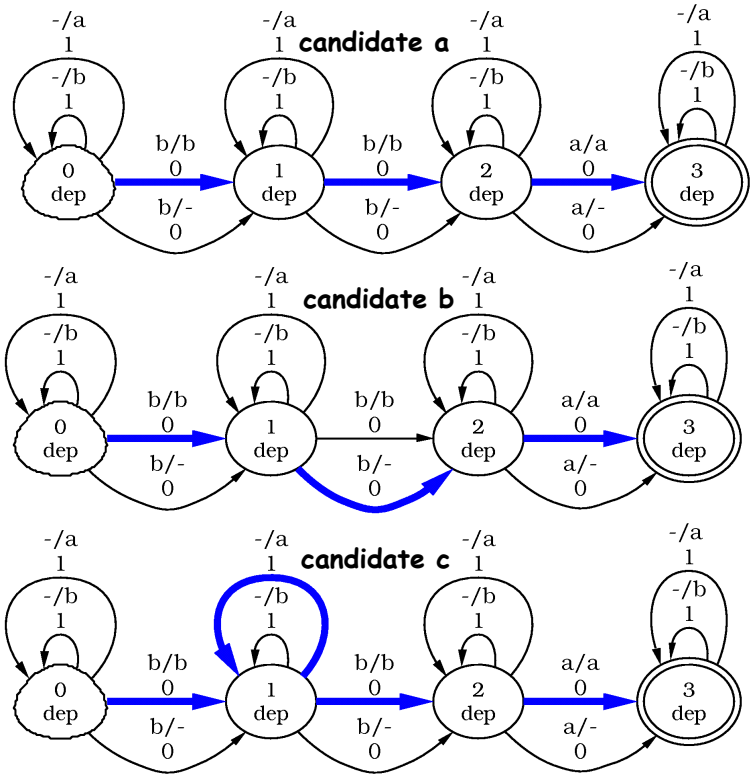
b. A path p is a path **between** q_1 and q_2 if the first arc in p has q_1 as its origin and the last arc in p has q_2 as its terminus.

c. A path p is a path **through** machine $M = (Q, \Sigma, \delta, q_0, F)$ if p is a path between q_0 and one of the final states of M.

The machine given in (19) represents the evaluation of an infinite candidate set with respect to the constraint DEP. Each path through (19) corresponds to one row in an infinite tableau. Three of these paths are illustrated in (21).

(21) Three candidates:

/bba/	DEP
a. bba	
b. ba	
c. baba	*!



Optimization for a single constraint can be quite easy. In the case above, removing all of the arcs marked with DEP violations will eliminate all of the candidates that violate DEP. When dealing with several constraints simultaneously things can, of course, become quite a bit more complicated. Now that the basic method for using finite state constraints to evaluate candidates has been laid-out I will turn to the issue of constraint interaction and introduce a few more constraints.

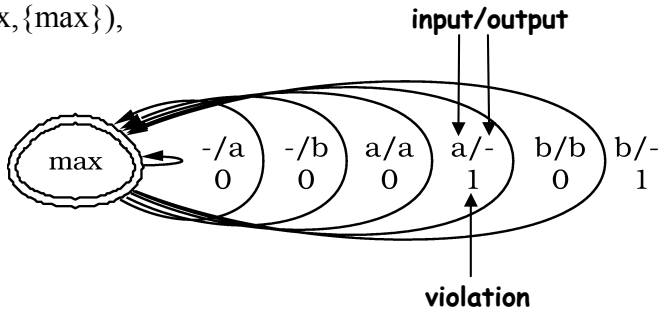
2.5 Three more constraints

Consider in (22) the constraint MAX which penalizes the deletion of material in the mapping from the underlying to the surface form (McCarthy and Prince 1995).

(22) MAX: input segments may not be deleted

MAX = ({max},{a, b}, δ, max,{max}),

δ = {(max, -, a, 0, max),
(max, -, b, 0, max),
(max, a, -, 1, max),
(max, a, a, 0, max),
(max, b, -, 1, max),
(max, b, b, 0, max)}.



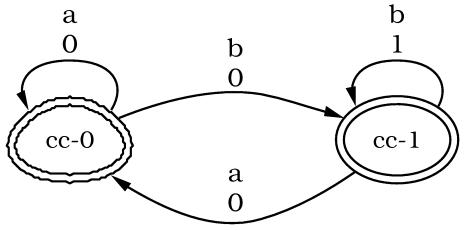
Basic faithfulness constraints like DEP and MAX only require one state because they assign violations irrespective of any particular phonological environment. I won't consider faithfulness constraints with environments in this work for simplicity's sake, but note that multi-state faithfulness constraints can be used capture "positional" faithfulness constraints (cf. Beckman 1995, 1998, Casali 1997).

Unlike faithfulness constraints, markedness constraints usually require multiple states to pick out the environments in which violations an occur.[9] Consider, for instance, the constraint *CC presented in (23) which assigns one violation per consonant-consonant sequence that occurs in a surface form.

(23) *CC: consonant-consonant sequences are not allowed

*CC = ({cc-0,cc-1},{a, b}, δ, cc-0, {cc-0, cc-1}),

δ = { (cc-0, •, a, 0, cc-0),
(cc-0, •, b, 0, cc-1),
(cc-0, •, -, 0, cc-0),
(cc-1, •, b, 1, cc-1),
(cc-1, •, a, 0, cc-0),
(cc-1, •, -, 0, cc-1)}.



In the finite state representation of *CC in (23), state cc-1 encodes the fact that a consonant has just been written as output. From this state, if another consonant is written as output then a violation of *CC occurs. In (23) and in the graphs throughout this work I won't bother to denote wildcards on the arcs and I won't write the arc looping from each node to itself accepting the empty string.

In general, the input slots on the arcs of markedness constraints will be filled with wildcards because markedness constraints aren't sensitive to properties of the input and thus needn't specify any information about the input. By specifying information about the input on a markedness constraint like the one in (23) it's possible to allow material that isn't present in the output to dictate whether or not the constraint is violated and to use

[9] Simple markedness constraints that penalize every occurrence of a particular segment can be represented with a single state.

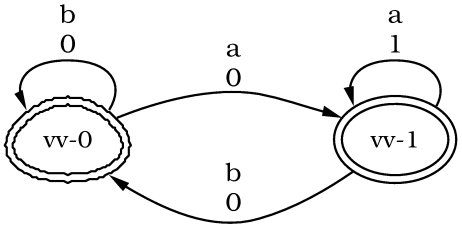
particular unfaithful i/o mappings to specify the environments where the constraint can be violated. Such constraints blur the line between markedness and faithfulness but allow a straightforward account of certain types of opacity and derived environment effects. This issue is worth further study, but for the current implementation I'll keep markedness and faithfulness strictly separate by disallowing markedness constraints that specify anything about the input.

In (24) I present the counterpart ot *CC, the constraint *VV, which penalizes vowel-vowel sequences in output strings.

(24) *VV: vowel-vowel sequences are not allowed

*VV = ({vv-0,vv-1},{a, b}, δ, vv-0, {vv-0, vv-1}),

δ = { (vv-0, •, a, 0, vv-1),
(vv-0, •, b, 0, vv-0),
(vv-0, •, -, 0, vv-0),
(vv-1, •, b, 0, vv-0),
(vv-1, •, a, 1, vv-1),
(vv-1, •, -, 0, vv-1)}.



This quartet of constraints, *VV, *CC, DEP, and MAX, makes up the basic suite of constraints that will be used to illustrate most of the algorithms presented in this work. Larger and more complicated sets of constraints will be discussed and examined in order to assess the scalability of the proposals presented here but I'll keep coming back to this simple case when illustrating new ideas. This suite of constraints will make it easier to explain the algorithms used in this work because the set is small enough and the constraints are simple enough that it will remain possible to rely on intuitions about their interactions to aid in understanding how the algorithms work.

2.6 Recursive *M*-intersection

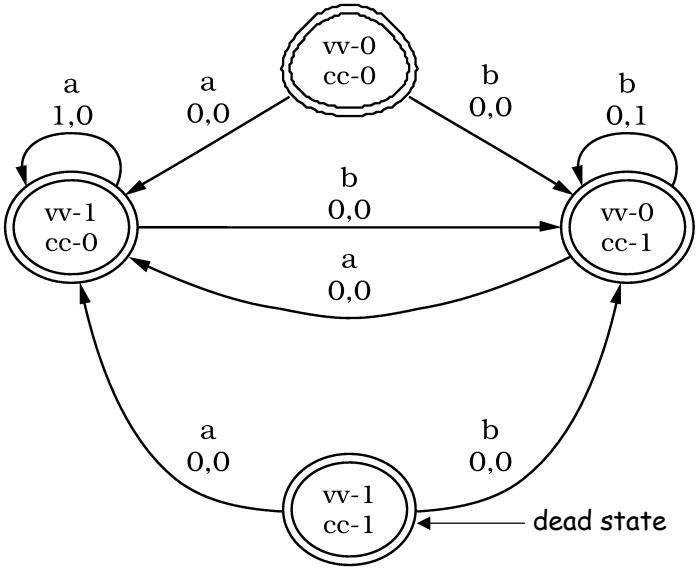
Ellison (1994) shows how candidates can be evaluated with respect to multiple constraints simultaneously by building a single evaluator for the entire set of constraints. To do this we can simply extend the definition of *M*-intersection recursively as in (25).

(25) ***def*: *M*-intersection – *recursive definition***

$\langle M_1, M_2, \ldots M_k \rangle^{\boxtimes} = \langle \langle M_1, M_2, \ldots M_{k-1} \rangle^{\boxtimes}, M_k \rangle^{\boxtimes}$

When two markedness constraints are brought together no input is specified, so the wildcards are retained in the input slots on the arcs. If the constraints have multiple states then there is the possibility that "dead" states will be created by intersection. For instance, in the intersection of *VV and *CC, there is one state that is in the cross product of *VV and *CC that could never be reached. This is illustrated in (26).

(26) A dead state:



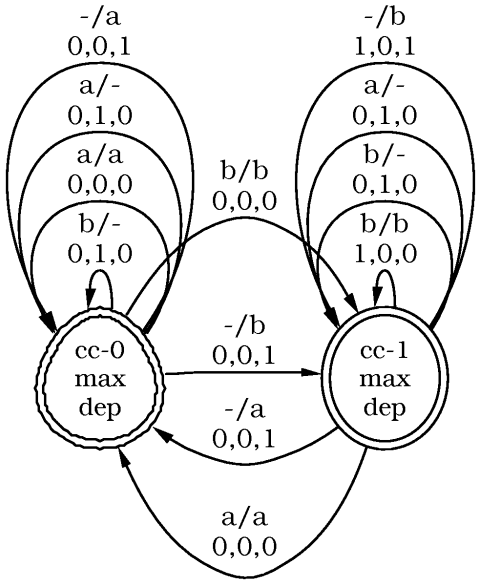
State (vv-1,cc-1) corresponds to state (vv-1) in *VV and state (cc-1) in *CC. State (vv-1,cc-1) could only be reached if the previous output symbol were simultaneously 'b' and 'a'. Thus there is no path from the start state (vv-0,cc-0) to (vv-1,cc-1) making it a dead (unreachable) state. Because dead states don't contribute any complete paths to the machine, I'll generally omit them when graphing intersected constraints.

The number of nodes in the machine representing the intersected set of constraints depends on the number of possible phonological environments in the intersection of the environments specified by those constraints. In (26), for instance, there could be $2 \times 2 = 4$ states in the intersected machine but only three states need be considered because there are only three phonological environments to which the grammar is sensitive. That is, either nothing has been seen yet, an 'a' has just been seen, or a 'b' has just been seen.

I'll return to the issue of the size of the machine representing *Eval* in chapter five. The point to take away from (26) is that the size of the intersected machine grows with the intersection of the environments specified by the intersected constraints and, because some environments overlap and others are mutually exclusive, the number of nodes in the machine resulting from intersection will often be smaller than the product of the number of nodes in the constraints used to build it.

For an illustration of *M*-intersection that uses both markedness and faithfulness constraints, consider in (27), the machine resulting from intersecting *CC, DEP and MAX.

(27) ⟨*CC, DEP, MAX⟩$^{\boxtimes}$:



⟨*CC, DEP, MAX⟩$^{\boxtimes}$ = (Q, {a,b}, δ, (cc-0,dep,max), F), where

Q = {(cc-0,dep,max), (cc-1,dep,max)} = F, and δ =

{((cc-0,dep,max), -, a, ⟨0,1,0⟩, (cc-0,dep,max)), ((cc-0,dep,max), a, -, ⟨0,0,1⟩, (cc-0,dep,max)),
((cc-0,dep,max), a, a, ⟨0,0,0⟩, (cc-0,dep,max)), ((cc-0,dep,max), a, -, ⟨0,0,1⟩, (cc-0,dep,max)),
((cc-0,dep,max), -, b, ⟨0,1,0⟩, (cc-1,dep,max)), ((cc-0,dep,max), b, b, ⟨0,0,0⟩, (cc-1,dep,max)),
((cc-1,dep,max), -, b, ⟨1,1,0⟩, (cc-1,dep,max)), ((cc-1,dep,max), a, -, ⟨0,0,1⟩, (cc-1,dep,max)),
((cc-1,dep,max), b, -, ⟨0,0,1⟩, (cc-1,dep,max)), ((cc-1,dep,max), b, b, ⟨1,0,0⟩, (cc-1,dep,max)),
((cc-1,dep,max), a, a, ⟨0,0,0⟩, (cc-0,dep,max)), ((cc-1,dep,max), -, a, ⟨0,1,0⟩, (cc-0,dep,max))}.

All of the structure in ⟨*CC, DEP, MAX⟩$^{\boxtimes}$ comes from the constraint *CC. State (cc-1,dep,max) encodes the fact that a consonant has just been written in the output and that if another consonant is written a violation of *CC will occur. Because MAX and DEP have only one state they don't increase the complexity of the machine by adding more states; they just add more detail to the cost vectors that label the arcs. In section 2.7 we'll consider how *Eval* grows in complexity as more constraints are added, but first I'll show how *Eval* can be used to evaluate candidate sets.

2.7 Using *Eval*

Intersecting a set of constraints produces a finite state machine that defines an infinite set of ⟨*in*, *out*, *cost*⟩ triples. Each triple in the set corresponds to a path through the machine and encodes the *cost* in constraint violations of mapping *in* to *out*. The cost of a path in a machine like (27) is the sum of the costs of the arcs that make up the path. The sum of two cost-vectors is simply the pairwise sum of the corresponding coordinates and the difference of two cost-vectors is the pairwise difference of their coordinates. For instance ⟨0, 2, 1⟩ plus ⟨0, 1, 1⟩ equals ⟨0, 3, 2⟩ and ⟨0, 2, 1⟩ minus ⟨0, 1, 1⟩ equals ⟨0, 1, 0⟩. This is defined in (28).

(28) **Cost vector arithmetic:**

$\langle v_1, v_2, ..., v_k\rangle + \langle w_1, w_2, ..., w_k\rangle = \langle(v_1 + w_1), (v_2 + w_2), ..., (v_k + w_k)\rangle$

$\langle v_1, v_2, ..., v_k\rangle - \langle w_1, w_2, ..., w_k\rangle = \langle(v_1 - w_1), (v_2 - w_2), ..., (v_k - w_k)\rangle$

When it's clear from the context what machine I am referring to, or when the specific machine doesn't matter I'll refer to the intersected set of constraints simply as *Eval*. The set of ⟨*in*, *out*, *cost*⟩ triples produced by *Eval* can be defined as in (29).

(29) Defining *Eval* in terms of ⟨*in*, *out*, *cost*⟩ triples:

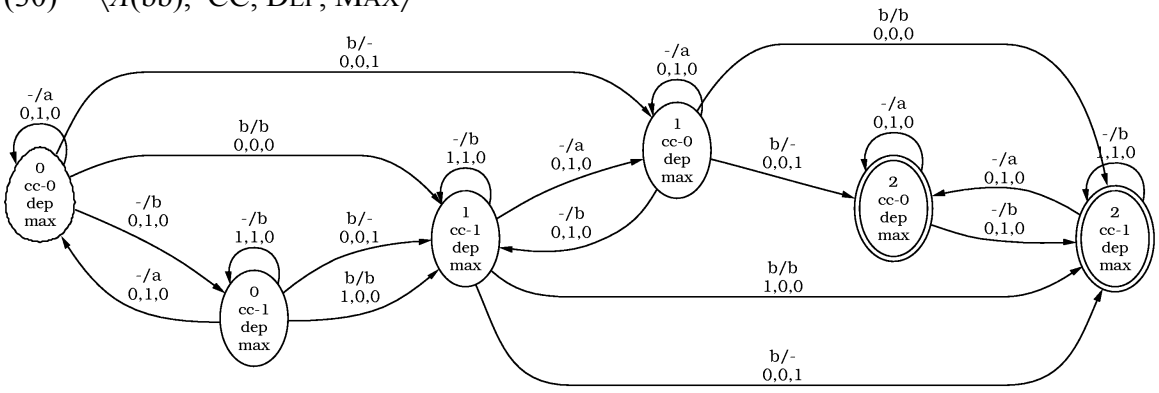
$\langle C_1, \ldots, C_k\rangle^{\otimes} = \{\langle i, o, c\rangle$ | there is a path p through $\langle C_1, \ldots, C_k\rangle^{\otimes}$ where

$p = \langle\langle q_1, i_1, o_1, v_1, r_1\rangle, ..., \langle q_n, i_n, o_n, v_n, r_n\rangle\rangle$,

$i = (i_1 i_2 \ldots i_n)$, $o = (o_1 o_2 \ldots o_n)$, and $c = (v_1 + ... + v_n)\}$

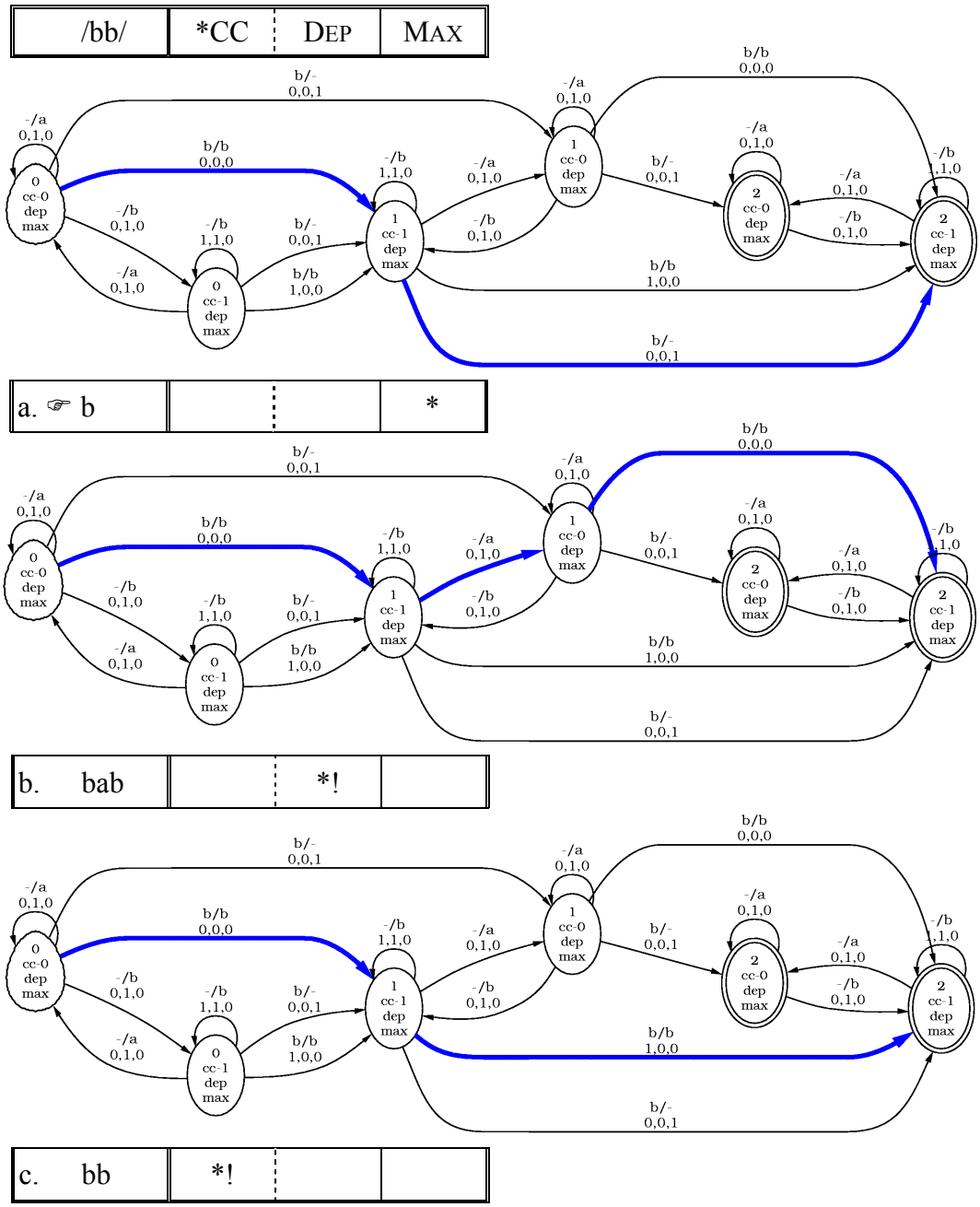
To evaluate all of the candidates for the input string *in*, the input acceptor $A(in)$ is intersected with the intersected set of constraints. For example, intersecting A(bb) with $\langle$*CC, DEP, MAX$\rangle^{\otimes}$ produces the machine represented in (30). This machine defines an infinite tableau encoding the evaluation of every candidate i/o pair for the input /bb/ under the constraints $\langle$*CC, DEP, MAX$\rangle^{\otimes}$.

(30) $\langle A(\text{bb}),$*CC, DEP, MAX$\rangle^{\otimes}$



Each member of the infinite set of paths though (30) corresponds to one row of a standard OT tableau. Each path accepts the input /bb/ along the first elements of its arc labels, writes an output candidate along the second elements of its arc labels, and marks violations along the third element of its arc labels. Consider the three candidates in (31).

(31) Three candidates:

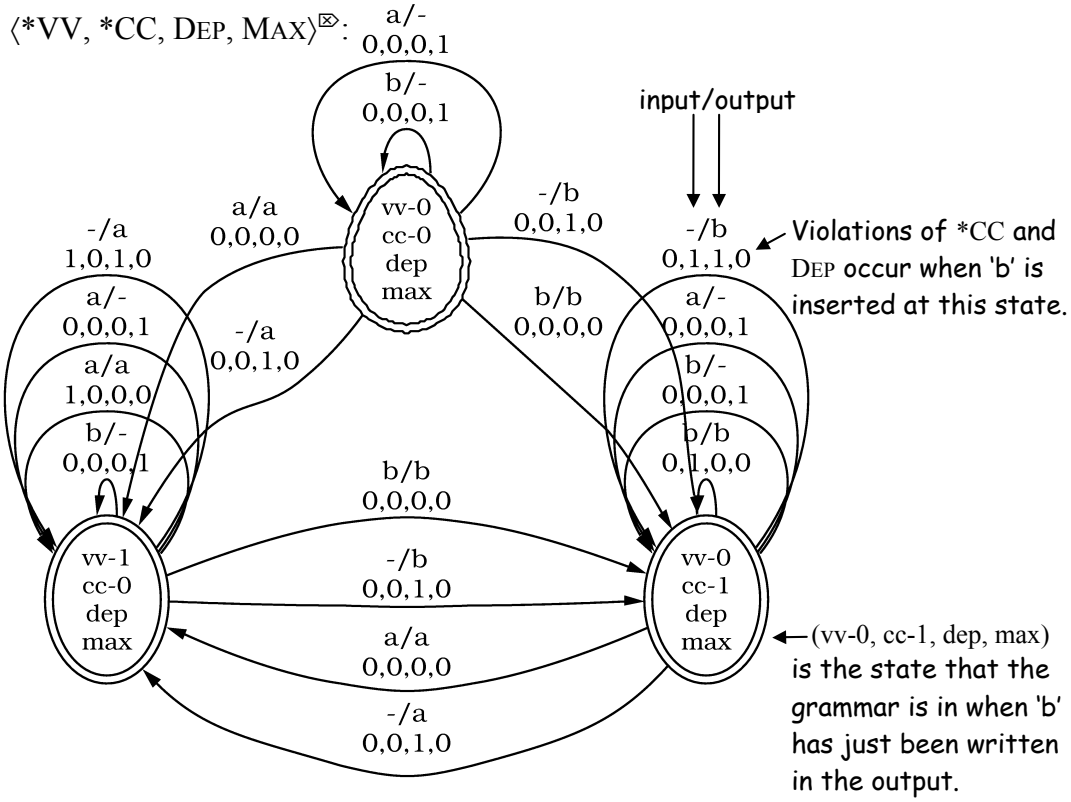


With simple graphs like the ones above it isn't hard to visually examine the paths through the machine and find the best ones (e.g. candidate a in (31)). With more constraints and longer input strings this task can become quite difficult.

2.8 The complexity of *Eval*

Adding *VV to *Eval* as presented in (27) could double its size. But when *VV is added the number of nodes in the machine is only increased by one because, as noted in (26), one of the nodes in the intersection is a dead state that can't ever be reached.

(32) ⟨*VV, *CC, DEP, MAX⟩$^{\boxtimes}$:



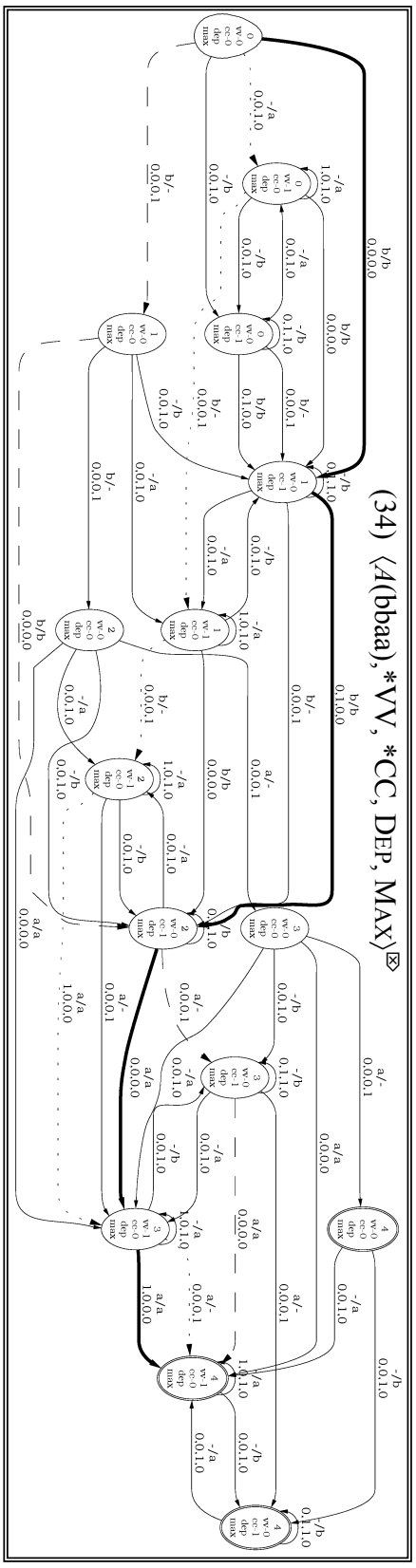
(33) ⟨*VV, *CC, DEP, MAX⟩$^{\boxtimes}$= (Q, {a, b}, δ, (vv-0, cc-0, dep, max), F)

where Q = {(vv-0, cc-0, dep, max), (cc-0, vv-1, dep, max), (cc-1, vv-0, dep, max)} = F,

δ = {((vv-0, cc-0), a, -, ⟨0,0,0,1⟩, (vv-0, cc-0)), ((vv-0, cc-0), b, -, ⟨0,0,0,1⟩, (vv-0, cc-0)),
((vv-0, cc-0), a, a, ⟨0,0,0,0⟩, (vv-1, cc-0)), ((vv-0, cc-0), -, a, ⟨0,0,1,0⟩, (vv-1, cc-0)),
((vv-0, cc-0), b, b, ⟨0,0,0,0⟩, (vv-0, cc-1)), ((vv-0, cc-0), -, b, ⟨0,0,1,0⟩, (vv-0, cc-1)),
((vv-1, cc-0), a, -, ⟨0,0,0,1⟩, (vv-1, cc-0)), ((vv-1, cc-0), b, -, ⟨0,0,0,1⟩, (vv-1, cc-0)),
((vv-1, cc-0), a, a, ⟨1,0,0,0⟩, (vv-1, cc-0)), ((vv-1, cc-0), -, a, ⟨1,0,1,0⟩, (vv-1, cc-0)),
((vv-1, cc-0), b, b, ⟨0,0,0,0⟩, (vv-0, cc-1)), ((vv-1, cc-0), -, b, ⟨0,0,1,0⟩, (vv-0, cc-1)),
((vv-0, cc-1), a, -, ⟨0,0,0,1⟩, (vv-0, cc-1)), ((vv-0, cc-1), b, -, ⟨0,0,0,1⟩, (vv-0, cc-1)),
((vv-0, cc-1), a, a, ⟨0,0,0,0⟩, (vv-1, cc-0)), ((vv-0, cc-1), -, a, ⟨0,0,1,0⟩, (vv-1, cc-0)),
((vv-0, cc-1), b, b, ⟨0,1,0,0⟩, (vv-0, cc-1)), ((vv-0, cc-1), -, b, ⟨0,1,1,0⟩, (vv-0, cc-1))}.

– dep and max are omitted from the node-names in δ in (33).

Eval in (33) evaluates candidates with respect to *VV, *CC, DEP, and MAX simultaneously. Because the structure-building aspect of *M*-intersection is commutative, the only difference among the machines for the twenty-four rankings of these four constraints will be in the order of ones and zeroes on the arcs. (34) shows ⟨*VV, *CC, DEP, MAX⟩$^{\boxtimes}$ intersected with *A*(bbaa) to evaluate all output candidates for /bbaa/. Three of the candidates are presented in (35).



(35) Three candidates:

	/bbaa/	*VV	*CC	DEP	MAX
bold →	a. ba				**
dashed →	b. bbaa	*!	*!		
dotted →	c. aa	*!		*!	***

Given the rapid increase in the complexity of these machines, it's obvious that visual examination is impractical in the search for optimal candidates. In the next chapter I will show how the fact that these graphs encode finite representations of the infinite candidate space can be exploited to allow us to efficiently find optimal candidates among the infinite range of possible outputs.

3 Optimization

Following Ellison's (1994) observation that the optimization problem in OT can be cast as a shortest paths problem in a graph-theoretic representation of the input/output mapping, I will present here an algorithm that takes the finite state machine resulting from intersecting an input acceptor with a set of finite state constraints and finds the cost of the most harmonic path to each node in that machine. This information will then be used to eliminate all suboptimal paths through the machine and thereby eliminate all suboptimal candidates. In other words, this algorithm will take a machine that generates an infinite set of candidates and return a sub-machine that generates only optimal candidates.

3.1 Harmony

In defining exactly what it means for one candidate to be more harmonic than another candidate, Grimshaw (1997) puts it most concisely: "[a] form which, for every pairwise competition involving it, best satisfies the highest-ranking constraint on which the competitors conflict, is *optimal*." Following Samek-Lodovici and Prince (1999, 2002), in this work, I'll abstract away from actual candidates and express harmony in terms of cost vectors.[10] Given two cost vectors v and w, the former is more harmonic than the latter, written $v \succ w$, just in case for every constraint C_j for which w has fewer violations than v there is some constraint C_i ranked that's above C_j for which v has fewer violations than w.

[10] Several candidates might share the same cost vector (set of violations) and it is the cost vectors, not the candidates, that compete for optimality.

In the cost vectors on the arcs of the machines, dominance among the constraints corresponds to precedence in the vectors. I'll make use of this fact in defining harmony in (36). In (36) I introduce the term INF to denote the infinite cost and the term $\overline{0}$ to serve as the all-zero cost. These will come in handy in defining optimization.

(36) ***def***: harmony

$v \succ w$ iff $v = \overline{0} \neq w$, $w = \text{INF} \neq v$, or

for $v = \langle v_1, \ldots, v_k \rangle$ and $w = \langle w_1, \ldots, w_k \rangle$, there is an i such that $w_i > v_i$ and, for every j such that $v_j > w_j$, it is the case that $j > i$.

The all-zero cost $\overline{0}$ doesn't add or take anything away from other cost vectors when they are summed (e.g. $v \pm \overline{0} = v$). When representing machines containing k-length cost vectors I'll usually represent $\overline{0}$ simply as a k-length sequence of zeroes.

In (37) I give a function *harm*(V) that picks out the single most harmonic member among a set of cost vectors V.

(37) $harm(V) = v$ such that $v \in V$ and there is no $w \in V$ for which $w \succ v$.

Because the definition of harmony in (36) guarantees that no two distinct vectors can be equally harmonic, there will always be a single unique "most harmonic vector" in any set of cost vectors.

Throughout this work I'll refer to the violations incurred by various paths through machines (candidates) as their cost. Concomitantly I'll sometimes use the term "cheaper" to mean more harmonic and the term "cheapest" to mean most harmonic.

3.2 The OPTIMIZE algorithm

In this section I present OPTIMIZE, an algorithm based on Dijkstra's (1959) single source shortest paths algorithm that basically combines Ellison's (1994) LABEL-NODES and PRUNE algorithms. This algorithm takes a machine whose arcs are "weighted" with cost-vectors and determines, for each node in a machine, the cost of the most harmonic (cheapest) path to that node from the unique start state. After gathering this information, the algorithm returns a new machine that contains all and only the most harmonic paths through the original machine. In this fashion it is possible to remove all suboptimal paths through the machine and thereby create a new machine that generates all and only the optimal candidates.

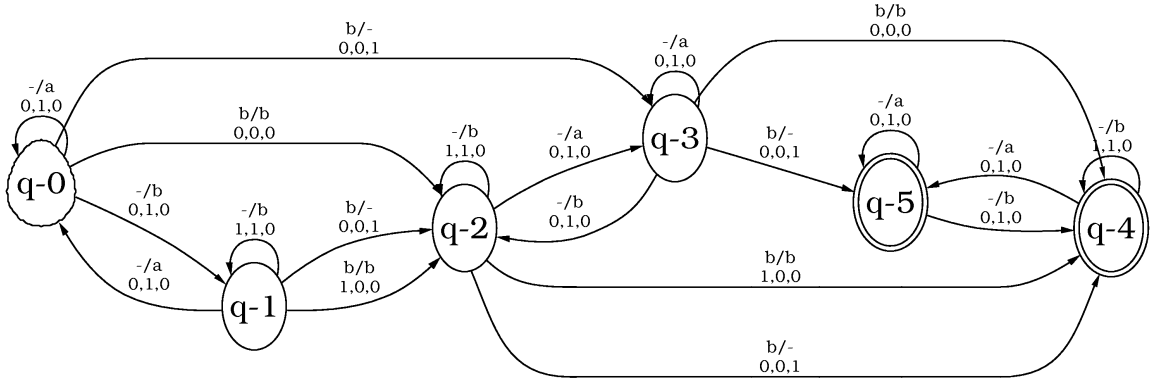
OPTIMIZE works by maintaining for each node q in the machine an attribute $o[q]$ that expresses the upper bound on the cost of the most harmonic path from the start state q_0 to q. This "cost estimate" for each node is set initially to INF and then gradually lowered as more information becomes available. At the outset of the algorithm, all of the nodes of the machine are placed in the set H. The set H can be thought of as "to do" list containing nodes for which the current value of $o[q]$ hasn't yet necessarily converged on the lowest (most harmonic) possible value for q. Over the course of the algorithm nodes in H are extracted one at a time as stable final estimates for $o[q]$ are reached. Finally, once the cost of the most harmonic path to each node has been found, this information is used to eliminate all of the suboptimal paths through the machine. The OPTIMIZE algorithm is presented in pseudo-code with comments in (38).

(38) OPTIMIZE(Q, Σ, δ, q_0, F)

1	$H \leftarrow Q$	- Put all of the nodes in the set H.
2	**for** each $q \in H$	
3	**do** $o[q] \leftarrow$ INF	- Set all of the cost attributes to INF.
4	$o[q_0] \leftarrow \overline{0}$	- Set the cost attribute of q_0 to $\overline{0}$.
5	**while** $H \neq \varnothing$	- While H is not empty loop as follows:
6	**do** $q_u \leftarrow q_u \in H$, and $\neg\exists q'$ such that $q' \in H$, $o[q'] \succ o[q_u]$	get one of the cheapest nodes in H,
7	$H \leftarrow H$ - $\{q_u\}$	remove that node, q_u, from H and
8	**for** each $(q_u, i, o, w, q_v) \in \delta$, $q_v \in H$	for each arc from q_u to a $q_v \in H$,
9	**do** $o[q_v] \leftarrow harm(\{o[q_v], (o[q_u] + w)\})$	replace the cost-estimate for q_v with $o[q_u] + w$ if the latter is cheaper.
10	**for** each $f \in F$	- Delete from the set of final nodes any node less harmonic than the most harmonic final.
11	**do** $f \leftarrow \varnothing$ if there is an $f' \in F$ such that $o[f] \prec o[f']$	
12	**for** each $a = (q_u, i, o, w, q_v) \in \delta$	- Delete any arc whose cost is not equal to the difference between the cost-estimate for its terminus and the estimate for its origin.
13	**do** $a \leftarrow \varnothing$ if $(c[q_u] + w) \neq c[q_v]$	

For an illustration of OPTIMIZE in action, consider in (39) the intersection of input /bb/ with *Eval* = *CC ≫ DEP ≫ MAX, repeated from (30). In (39) I've relabeled the nodes in the machine to $\{q_0, q_1, q_2, q_3, q_4, q_5\}$ to make them easier to refer to and keep track of.

(39) $\langle A(\text{bb}), *\text{CC}, \text{DEP}, \text{MAX}\rangle^{\boxtimes}$



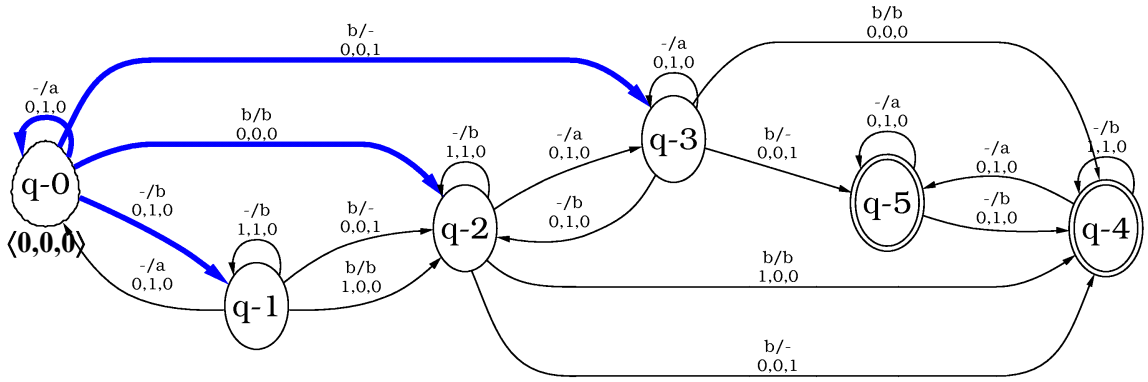
The algorithm is initialized by setting H to $\{q_0, q_1, q_2, q_3, q_4, q_5\}$ (**step one**) and setting all of the costs estimates to INF (**step two** and **step three**). The cost of the start state is set to zero (**step four**). In (40) I give a table indicating the current value of $o[q]$ for each node. These values serve as upper bounds on the cost of the most harmonic path from the start state to each node. Other than the start state (which can be reached from itself for free) all of the values are infinite. As paths to the nodes are found, these estimates will be lowered.

(40) The cost estimates:

	q_0	q_1	q_2	q_3	q_4	q_5
0.	⟨0,0,0⟩	INF	INF	INF	INF	INF

Since H is nonempty (**step five**), the node in H with the most harmonic cost is selected (**step six**). The only node whose cost is not infinite is q_0, so it is selected and removed from H (**step seven**) leaving $H = \{q_1, q_2, q_3, q_4, q_5\}$. Next, for each arc in δ with an origin at q_0 (**step eight**) the cost of the most harmonic path to that arc's terminus is reassessed.

(41) Four arcs originate at q_0:



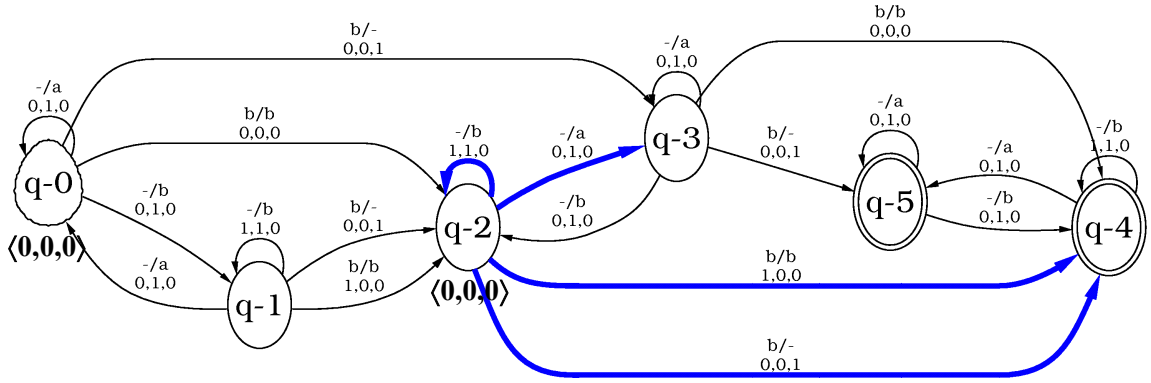
There are four arcs to be checked that have origins q_0. By traversing arc (q_0, -, a, ⟨0,1,0⟩, q_0) node q_0 can be reached at a cost of ⟨0,1,0⟩. This is the cost of the arc ⟨0,1,0⟩, plus $o[q_0]$ = ⟨0,0,0⟩. Since node q_0 is not in H (it was removed in step seven) no attempt is made to update the value of q_0 (even if q_0 was still in H, its value wouldn't be updated because ⟨0,1,0⟩ isn't more harmonic than the current cost estimate of ⟨0,0,0⟩ for q_0). Taking arc (q_0, -, b, ⟨0,1,0⟩, q_1), node q_1 can reached at a cost of ⟨0,1,0⟩. Since this is more harmonic than the current estimate of INF for $o[q_1]$, the value of $o[q_1]$ is updated to ⟨0,1,0⟩ (**step nine**). Checking arcs (q_0, b, b, ⟨0,0,0⟩, q_2) and (q_0, b, (), ⟨0,0,1⟩, q_3) in the same fashion yields an update of the values of q_2 and q_3 to ⟨0,0,0⟩ and ⟨0,0,1⟩ respectively (**step nine** two more times). The updated cost-estimates are given in the table in (42).

(42) The cost estimates:

	q_0	q_1	q_2	q_3	q_4	q_5
0.	⟨0,0,0⟩	INF	INF	INF	INF	INF
1.	⟨0,0,0⟩	⟨0,1,0⟩	⟨0,0,0⟩	⟨0,0,1⟩	INF	INF

In (42) I've shaded in the cell with the cost estimate for q_0 because it has been removed from H. Since H is nonempty, the node with the most harmonic cost is selected (this is the second iteration of **step six**). Node q_2 has the most harmonic cost estimate of any node remaining in H so it is removed, leaving $H = \{q_1, q_3, q_4, q_5\}$. For each arc in δ with an origin at q_2, the cost of the most harmonic path to that arc's terminus is reassessed.

(43) Four arcs originate at q_2:



Node q_2 has already been removed from H, so arc (q_2, -, b, ⟨1,1,0⟩, q_2) does not need to be checked.[11] Arc (q_2, b, -, ⟨0,0,1⟩, q_4) reaches node q_4 at a cost of ⟨0,0,1⟩, which is more harmonic than the current cost estimate for q_4, so $o[q_4]$ is updated. Arc (q_2, b, b, ⟨1,0,0⟩, q_4) reaches node q_4 at a cost of ⟨1,0,0⟩, which is less harmonic than the estimate we just made for q_4, so $o[q_4]$ is not updated. And finally, arc (q_2, -, a, ⟨0,1,0⟩, q_3) reaches node q_3 at a cost of ⟨0,1,0⟩, which is less harmonic than the current estimate for the most harmonic path to q_3, so $o[q_3]$ is not updated. The updated cost-estimates are given in (44).

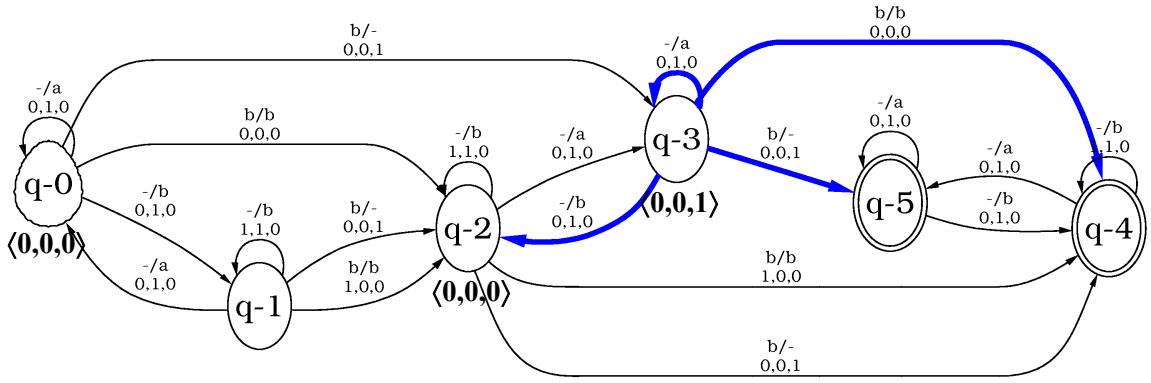
[11] If q_2 were still in H arc (q_2, -, b, ⟨1,1,0⟩, q_2) could reach it at a cost of ⟨1,1,0⟩, which is less harmonic than the current cost-estimate for the most harmonic path to q_2, so $o[q_2]$ wouldn't be updated. It is fairly easy to see here that epenthetic loops will never be optimal. Unless epenthesis changes the state in the machine it is merely gratuitous and can't possibly remedy violations of any other constraint.

(44) Updated cost estimates

	q_0	q_1	q_2	q_3	q_4	q_5
0.	⟨0,0,0⟩	INF	INF	INF	INF	INF
1.	⟨0,0,0⟩	⟨0,1,0⟩	⟨0,0,0⟩	⟨0,0,1⟩	INF	INF
2.	⟨0,0,0⟩	⟨0,1,0⟩	⟨0,0,0⟩	⟨0,0,1⟩	⟨0,0,1⟩	INF

Since H is nonempty, the node with the most harmonic cost is selected (this is the third iteration of **step six**). The unshaded cells show the nodes that are still in H; of these the estimates for q_3 and q_4 are equally harmonic so either one may be selected and removed from H (it doesn't matter which). If q_3 is removed then $H = \{q_1, q_4, q_5\}$. For each arc in δ originating at q_3, the cost of the most harmonic path to that arc's terminus is reassessed.

(45) Four arcs originate at q_3:



Arc (q_3, -, a, ⟨0,1,0⟩, q_3) reaches node q_3 at a cost of ⟨0,1,1⟩, which is less harmonic than the current cost estimate for q_3, so $o[q_3]$ is not updated. Arc (q_3, -, b, ⟨0,1,0⟩, q_2) reaches node q_2 at a cost of ⟨0,1,1⟩, which is less harmonic than the current cost-estimate for q_2, so $o[q_2]$ is not updated. Arc (q_3, b, b, ⟨0,0,0⟩, q_4) reaches q_4 at a cost of ⟨0,0,1⟩, which is equally harmonic to the current cost estimate for q_4, so $o[q_4]$ is not updated. And finally,

the arc (q_3, b, -, ⟨0,0,1⟩, q_5) reaches node q_5 at a cost of ⟨0,0,2⟩, which is less harmonic than the current cost estimate for the most harmonic path to q_5, so $o[q_5]$ is not updated. The updated cost estimates are given in the table in (46).

(46) Updated cost estimates

	q_0	q_1	q_2	q_3	q_4	q_5
0.	⟨0,0,0⟩	INF	INF	INF	INF	INF
1.	⟨0,0,0⟩	⟨0,1,0⟩	⟨0,0,0⟩	⟨0,0,1⟩	INF	INF
2.	⟨0,0,0⟩	⟨0,1,0⟩	⟨0,0,0⟩	⟨0,0,1⟩	⟨0,0,1⟩	INF
3.	⟨0,0,0⟩	⟨0,1,0⟩	⟨0,0,0⟩	⟨0,0,1⟩	⟨0,0,1⟩	⟨0,0,2⟩

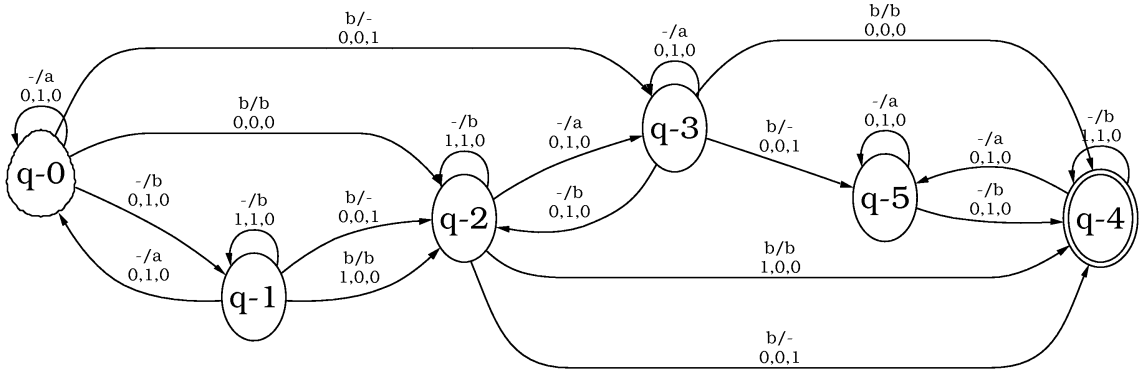
Three more iterations of the while loop in **step five** of the algorithm remove nodes q_1, q_4, and q_5 from H leaving $H = \varnothing$. These last three iterations don't yield new cost estimates for any of the nodes in the machine. The final estimates for the nodes are given in (47).

(47) Final cost estimates

	q_0	q_1	q_2	q_3	q_4	q_5
6.	⟨0,0,0⟩	⟨0,1,0⟩	⟨0,0,0⟩	⟨0,0,1⟩	⟨0,0,1⟩	⟨0,0,2⟩

After the cost of the most harmonic path to each node in the machine has been computed in steps one through nine of the algorithm, steps ten and eleven render nonfinal all but the most harmonic final states in F. Consulting (47), we see that $o[q_4] = ⟨0,0,1⟩$ and $o[q_5] = ⟨0,0,2⟩$. Thus, because $o[q_4] \succ o[q_5]$ node q_5 is removed from the set of final states. The result of removing q_5 from the set of final nodes is presented in (48).

(48) **Suboptimal finals eliminated:**

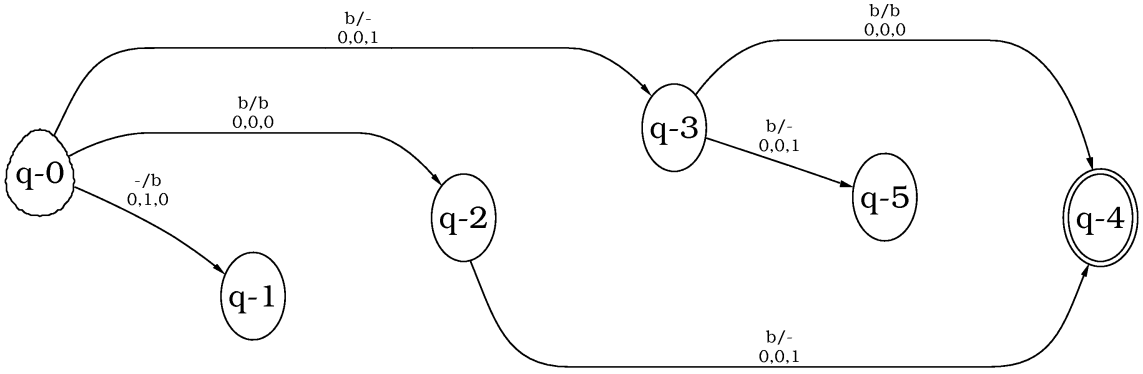


In **step twelve** and **step thirteen** of the algorithm, any arc whose cost is not equal to the cost estimate for its terminus minus the cost estimate for its origin is eliminated. This is illustrated for a few arcs in (49). In (50) I give result of removing of all suboptimal arcs.

(49) Evaluation of arcs:

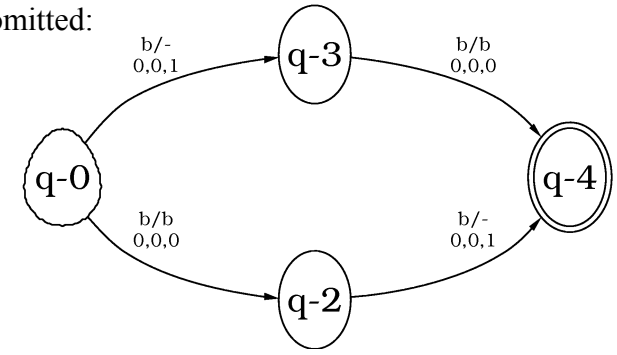
arc	**origin cost**	**terminus cost**	**status**
(q_1, b, -, ⟨0,0,1⟩, q_2)	$o[q_1]$ = ⟨0,1,0⟩	$o[q_2]$ = ⟨0,0,0⟩	deleted!
(q_2, -, a, ⟨0,1,0⟩, q_3)	$o[q_2]$ = ⟨0,0,0⟩	$o[q_3]$ = ⟨0,0,1⟩	deleted!
(q_0, b, -, ⟨0,0,1⟩, q_3)	$o[q_2]$ = ⟨0,0,0⟩	$o[q_3]$ = ⟨0,0,1⟩	left intact

(50) OPTIMIZE(⟨A(bb), *CC, DEP, MAX⟩$^{\boxtimes}$):



At this point the algorithm is finished and only optimal paths/candidates remain. In (51) I omit the dead-ends to make clear exactly which candidates are generated.

(51) Dead-ends omitted:



In (51) it is clear that there are actually two homophonous output candidates for input /bb/ that are optimal under *CC ≫ DEP ≫ MAX. In one candidate the first /b/ in the input is deleted and the second is faithfully parsed (the top path in (51)) and in the other candidate, the first /b/ is faithfully parsed and the second is deleted (the bottom path in (51)).

3.3 The Optimal Subpath lemma

Before proving that OPTIMIZE produces machines that generate all and only optimal candidates, I'll lay out a few definitions and a lemma that will be useful.

The **cost of an arc** is listed as the third element of its label. The cost of a path is the sum of the costs of its constituent arcs. I'll refer to a path's cost with the function in (52). This function takes a sequence of arcs and returns the coordinate-wise sum of the violation vectors along those arcs.

(52) The cost of a path:

$$c(p) = \sum_{i=0}^{t} v_i \quad \text{for } p = \langle (q_0, i_0, o_0, v_0, r_0), (q_1, i_1, o_1, v_1, r_1), \ldots, (q_t, i_t, o_t, v_t, q_t) \rangle$$

To denote a path from node q_u to q_v I'll write $p(q_u, q_v)$. Given a machine containing nodes q_u and q_v, the **cost of the optimal paths** from q_u to q_v can be defined as in (53).

(53) The cost of an optimal path:

$$opt(q_u, q_v) = \begin{cases} harm\{cst \mid c(p(q_u, q_v)) = cst\} & \text{if there is a path from } q_u \text{ to } q_v, \\ \text{INF} & \text{otherwise.} \end{cases}$$

A path p from node q_u to node q_v is an **optimal path** just in case $c(p) = opt(q_u, q_v)$.

(54) ***def***: optimal path through a machine

A path p is an **optimal path through** $\boldsymbol{M} = (Q, \Sigma, \delta, q_0, F)$ iff p is an optimal path from q_0 to $q_f \in F$ and there is no path p' from q_0 to $q' \in F$ such that $c(p) \prec c(p')$.

The definition in (54) takes into account the fact that there may be multiple final states in any given machine. To be optimal, a path must be among the most harmonic of all of the paths through the machine. To prove the correctness of OPTIMIZE I'll show that, at the termination of the algorithm, every complete path that remains in the machine is an optimal path through the machine.

Following Dijkstra's (1959) observation that shortest paths are made up of shortest paths, I observe here that optimal paths are made up of optimal paths. This gives us the lemma in (55).

(55) **Optimal subpath lemma:**

Subpaths of optimal paths are optimal paths. Given p, one of the most harmonic paths from node q_1 to q_t, for any i and j such that $1 \leq i \leq j \leq t$, if p_{ij} is a subpath of p from node q_i to node q_j, then p_{ij} is an optimal path from node q_i to node q_j.

proof: p can be seen as the concatenation of three subpaths $p = p_{1i} \cdot p_{ij} \cdot p_{jt}$ where p_{1i} is the path from q_1 to q_i, p_{ij} is the path from q_i to q_j and p_{jt} is the path from q_j to q_t and the cost of p is the sum of the costs of the subpaths, $c(p) = c(p_{1i}) + c(p_{ij}) + c(p_{jt})$.[12] To derive a contradiction assume there is a path p'_{ij} from node q_i to node q_j such that $c(p'_{ij}) \succ c(p_{ij})$.

If this were the case then there would also be a path p' from node q_1 to node q_t such that $c(p') = c(p_{1i})+c(p'_{ij})+c(p_{jt}) \succ c(p)$. But this contradicts the assumption that p is the most harmonic path from q_1 to q_t. ■

With the optimal subpath lemma in hand, the correctness of the OPTIMIZE algorithm can be shown to follow fairly straightforwardly.

[12] Since there's always the possibility of a zero-length path from a node to itself this statement covers the degenerate case where i equals 1 or j can equals t.

3.4 Correctness of OPTIMIZE

In this section I prove the correctness of the OPTIMIZE algorithm. The proof comes in two parts. I'll show first that at line ten of the algorithm (i.e. the point at which the cost-attribute table is finished) the value of $o[q]$ for each node is equal to $opt(q_0, q)$. Then I'll show that, given $opt(q_0, q)$ for each node, steps ten through thirteen of the algorithm leave intact all and only the optimal paths through the machine.

(56) Theorem: correctness of OPTIMIZE

Given a machine $M = (Q, \Sigma, \delta, q_0, F)$, OPTIMIZE($M$) returns a machine N in which there is a path p through N iff p was one of the most harmonic paths through M.

proof: First, I'll show that for every $q \in Q$ it's the case that $o[q] = opt(q_0, q)$ at line ten of the algorithm. Then I'll show that this guarantees that the biconditional holds.

part 1: I show here that $o[q] = opt(q_0, q)$ when q is removed from H and that this equality is preserved until the termination of the algorithm.

For the purposes of a contradiction, assume that q is the first node removed from H for which $o[q] \neq opt(q_0, q)$. I'll consider the point at which q is about to be removed from H and derive the contradiction that if q is indeed removed from H, $o[q]$ must in fact equal $opt(q_0, q)$, by examining an optimal path from q_0 to q.

Node q must not be q_0 because $o[q_0] = \overline{0}$ which must be equal to $opt(q_0, q_0)$. There must be some path from q_0 to q or else $o[q] = \text{INF} = opt(q_0, q)$, which violates the assumption that $o[q] \neq opt(q_0, q)$. Because there is at least one path from q_0 to q, there must be an optimal path p from q_0 to q. Consider q_y, the first

node along p that is still in H and q_x, the predecessor of q_y. The path p can be decomposed into the subpath p_x from q_0 to q_x, the arc (p_x, i, o, w, p_y), and the subpath p_y from q_y to q – in the minimal case p_x and p_y are empty and $q_x = q_0$ and $q_y = q$.

At this point the fact that $o[q_y] = opt(q_0, q_y)$ is entailed by the following facts. First, $o[q_x]=opt(q_0, q_x)$ because q was the *first* node that was removed from H where $o[q] \neq opt(q_0, q)$ and q_x has already been removed from H because it precedes q_y in p and q_y is the first node along p that is still in H. Second, arc (q_x, i, o, w, q_y) is the most harmonic path from q_x to q_y by the assumption that p is an optimal path and the optimal subpath lemma. Thus $o[q_y]$ is equal to $o[q_x] + w$ because line nine of OPTIMIZE updated the value of q_y when q_x was removed from H. Moreover because (q_x, i, o, w, q_y) lies along an optimal path $opt(q_0, q_x) + w$ equals $opt(q_0, q_y)$. Thus $o[q_y] = (o[q_x]+w) = (opt(q_0, q_x)+w) = opt(q_0, q_y)$, or simply $o[q_y] = opt(q_0, q_y)$.

Now the contradiction is evident. Because node q_y can't occur after q on a path from q_0 to q and all of the costs are nonnegative, it must be the case that $opt(q_0, q_y) \succeq opt(q_0, q)$ and thus because $o[q_y] = opt(q_0, q_y)$, it must be the case that $o[q_y] \succeq opt(q_0, q)$. Because $o[q]$ descends from infinity towards $opt(q_0, q)$ as the algorithm proceeds, it must be the case that $opt(q_0, q) \succeq o[q]$. Putting these facts together we have $o[q_y] \succeq opt(q_0, q) \succeq o[q]$ or simply $o[q_y] \succeq o[q]$. But because both q_y and q were still in H and q was not selected for removal after q_y it must be the case that $o[q] \succeq o[q_y]$. Finally, if $o[q] \succeq o[p_y]$ and $o[q_y] \succeq o[q]$ then

$o[q_y] = o[q]$, and from the facts above $o[q_y] = opt(q_0, q_y) = opt(q_0, q) = o[q]$. But this is contrary to the assumption that $o[q] \neq opt(q_0, q)$.

Thus when q is removed from H, it's true that $o[q] = opt(q_0, q)$ and because $o[q]$ doesn't change after this point the equality holds until the termination of the algorithm.

part 2: I'll show first that given $o[q] = opt(q_0, q)$ for each $q \in Q$ at the outset of step ten in OPTIMIZE, it's the case that if p is an optimal path through M then p is retained in N.

For a contradiction assume that p was an optimal path through M and that either its final state q_f was rendered nonfinal in line eleven or one of its arcs a_x was deleted in line thirteen. Let's try first, the possibility that q_f was rendered nonfinal. For this to occur there must be some $o[q_f'] \succ o[q_f]$. For $o[q_f']$ to be more harmonic than $o[q_f]$ there must be some path p' from q_0 to q_f' through M and $c(p') \succ c(p)$. But this contradicts the assumption that p was an optimal path through M. What about the possibility that one of p's arcs was deleted? For this to occur p must contain an arc a_x whose cost exceeds the difference in the cost-estimates for its origin and terminus (they are the only arcs deleted in step thirteen). If this were so then there would be a path p_x from q_0 to a_x's terminus that was cheaper than the subpath of p from q_0 to a_x's terminus. But the optimal subpath lemma tells us that if such a path existed then p would not be an optimal path, contradicting our assumption to the contrary.

I show second that given $o[q] = opt(q_0, q)$ for each $q \in Q$ at the outset of line ten in OPTIMIZE, it is the case that only optimal paths through M are retained in N.

There are two cases to consider. If p is not a path through M (it does not reach a final state) then it is not a path through N because no new finals or paths are created when building N. On the other hand, if p is a path through M but not an optimal path through M it won't survive in N. To see that this is so, assume for a contradiction that p did survive as a path through N. Because p is not an optimal path in M there must be another path p' through M such that $c(p') \succ c(p)$. There are two possibilities: if p is a path from q_0 to q_f and p' is a path from q_0 to q_f' then either $q_f = q_f'$ or $q_f \neq q_f'$. If $q_f = q_f'$ then $o[q_f] = o[q_f'] = c(p') \succ c(p)$, which means that somewhere along p there is an arc whose cost exceeds the difference in the cost estimates for its origin and terminus. But, because line thirteen of the algorithm deletes such arcs this contradicts the assumption that p was left intact by OPTIMIZE. Alternatively, if $q_f \neq q_f'$ then $o[q_f'] \succ o[q_f]$. If this were the case then q_f would be rendered nonfinal by line eleven of OPTIMIZE contradicting the assumption that p is left intact by the algorithm. ■

The OPTIMIZE algorithm given above in (38) will serve as the basis for several algorithms to be presented in the next two chapters. In chapter four I will examine optimization in detail and show how regularities across the optimization of various inputs can be generalized and utilized to make optimization more efficient, and in chapter five I will show how to generalize optimization so as to cover multiple rankings simultaneously.

4 Contenders

Until now we have been mainly concerned with the task of finding optimal output candidates with a particular ranking of the constraints in mind. In this chapter we'll take a broader perspective and turn to the task of finding all of the output candidates for a given input that can emerge as optimal under *any* ranking of a set constraints.

Prince and Smolensky (1993) refer to candidates that can never triumph under any permutation of the constraint set as "harmonically bounded." The simple case of harmonic bounding arises when one the violations for candidate are coordinate-wise-superset of the violations for another candidate (e.g. ⟨0,0,1,2⟩ is harmonically bounded by ⟨0,0,1,1⟩). Samek-Lodovici and Prince (1999, 2002) show that "complex harmonic bounding" can occur when two or more candidates gang-up to guarantee that a third candidate can never be optimal. Rather than calling the members of the complement of the set of harmonically bounded candidates "non-harmonically-bounded" or simply "winners" as in Samek-Lodovici and Prince (1999), I'll call the candidates that can win under some permutation of the constraints the "contenders."

To begin, let's review the definition of harmony as it is expressed relative to the cost vectors that label the arcs of the machines.

(57) **Harmony:**

For $v = \langle v_1, \ldots, v_n \rangle$ and $w = \langle w_1, \ldots, w_n \rangle$, v is more harmonic than w, written $v \succ w$, iff for each j such that $v_j > w_j$ there's an $i < j$ such that $v_i < w_i$.

Put simply, *v* is more harmonic than *w* just in case each constraint that *v* violates more than *w* is dominated (preceded) by a constraint that *w* violates more than *v*. Under this definition any given ranking of the constraints (ordering of the coordinates in the vectors) will select exactly one cost vector in any set as the most harmonic. From this observation Samek-Lodovici and Prince (1999) show that it readily follows that with *n* constraints there are at most *n*! contenders among any set of cost vectors (one contender per ranking).

4.1 Contenders defined

In (59) I give a recursive function that examines a cost vector *w* in a set of vectors *V* and returns true just in case there is some ranking of the constraints under which *w* is more harmonic every other member of *V*. First, to facilitate the definition of contenders, in (58) I give a function to pick out the lowest value at a particular coordinate among the vector in a set of cost vectors.

(58) $min_i(V)$ is the lowest value at the i^{th} coordinate for any vector in the set *V*.

(59) $$contender(w,V)\begin{cases}\text{true if } V=\{w\},\\ contender(w,V') \text{ if } V'=\{v\,|\,v\in V,\forall i(w_i=min_i(V)\rightarrow w_i=v_i)\}\neq V,\\ \text{false otherwise}\end{cases}$$

The first clause of (59) states that *w* is a contender in the vector-set containing only itself. The second clause of (59) is a bit more complicated. It says that *w* is a contender in *V* if *w* is a contender in the proper subset of *V* containing just the vectors that, if *w* has the lowest

i^{th}-coordinate value in V, are tied with w on the i^{th} coordinate.[13] If there is no such proper subset then there is no ranking that selects w and the *contender* function returns false.

In (60) I give a function *cont*(V) that takes a set of cost vectors and returns the subset consisting of just the vectors that are contenders in that set.

(60) $cont(V) = \{v \mid v \in V \text{ and } contender(v, V)\}$

The contenders can be quickly identified among any finite (and relatively small) set of cost vectors with the *contenders* function. As usual, the problem we face in trying to find the set of all contenders for a particular input is that the set of possible candidates (and thus the set of possible cost vectors) is not finite. In the next section I'll show how the same kind of strategy that was used to find optimal paths in the OPTIMIZE algorithm can be used to find contenders.

4.2 The CONTENDERS algorithm

Here I present CONTENDERS, a relatively straightforward extension of the OPTIMIZE algorithm presented in chapter three. In the CONTENDERS algorithm the gradient property of harmony that served as the "distance" metric in the OPTIMIZE algorithm is replaced with a binary distinction between cost vectors that are harmonically bounded and cost vectors that are contenders.

[13] This recursive step is similar to Prince's (2002b:20) operation of "Tableau Reduction" but is stated numerically rather than over Elementary Ranking Conditions. I'll come back to this point in chapter 5 where I will use Elementary Ranking Conditions in the search for contenders.

In this algorithm we aren't trying to find a single shortest path through the machine but rather all of the paths with costs that are contenders. Unlike the OPTIMIZE algorithm, CONTENDERS proceeds by initially setting a cost attribute for each node in the graph to the empty set ∅ and then gradually adding the costs of paths that reach that node as they are discovered (provided that they are contenders). It isn't necessary to start by setting the cost attribute for each node to INF because there is no point in the algorithm at which the set of cost attributes for the nodes is searched to find the cheapest one.

Any node that cannot be reached from the start state will end up with the empty set as its cost attribute termination of the algorithm. This outcome simply indicates that there is no cost at which that node can be reached from the start state.

In CONTENDERS, the cost attribute for each node is a set rather than a single value. This set may be made up of many different cost vectors, the only requirement being that are all contenders when compared to one another.

In order to compare and evaluate various paths through the machine it will be necessary to add the cost vectors of individual arcs to the sets of cost vectors that make up the cost attributes for the nodes. Extending the definition of cost-vector-sums as in (61) will facilitate this comparison.

(61) **Vector-set sum:**

$$V + w = \begin{cases} \{w\} \text{ if } V = \varnothing, \text{ else} \\ \{v' \mid v \in V \text{ and } v + w = v'\} \end{cases}$$

e.g. $\{\langle 0,1,0\rangle, \langle 0,0,0\rangle, \langle 2,1,0\rangle\}$

$$\underline{\qquad\qquad + \langle 1,1,1\rangle}$$

$$\{\langle 1,2,1\rangle, \langle 1,1,1\rangle, \langle 3,2,1\rangle\}$$

In (62) I present the CONTENDERS algorithm in pseudo-code with comments. As usual, I'll follow the presentation of the algorithm by stepping through its application to one optimization problem in detail. After running the algorithm to find the costs of the contenders we'll need a function to generate the actual candidates corresponding to those costs. I'll hold off on giving this function until §5.4 where it will be spelled out in detail.

(62) CONTENDERS(Q, Σ, δ, q_0, F) = CC

1	**for** each $q \in Q$	- Set the cost attribute for each node to the null set.
2	**do** $o[q] \leftarrow \varnothing$	
3	$o[q_0] \leftarrow \{\overline{0}\}$	- Set the cost of the start to $\{\overline{0}\}$
4	$H \leftarrow \{q_0\}$	- Put the start state in the set H.
5	**while** $H \neq \varnothing$	- While H is not empty loop as follows:
6	$H \leftarrow H - \{q_u\}$	remove a node, q_u, from H and
7	**for** each $(q_u, i, o, w, q_v) \in \delta$	for each arc from q_u to q_v,
8	**if** $cont(o[q_v] \cup o[q_u]+w) \neq o[q_v]$	if there are any new contenders,
9	**do** $o[q_v] \leftarrow cont(o[q_v] \cup o[q_u]+w)$ $H \leftarrow H \cup \{q_v\}$	update the cost attributes for $o[q_v]$ and add q_v to H.
10	$Cn \leftarrow cont\left(\bigcup_{q \in F} o[q]\right)$	- Select the contenders from the union of the cost attributes for the finals
11	**for** each $q_x \in F$	for the cost attribute of each final
12	**do** $o[q_x] \leftarrow o[q_x] \cap Cn$	keep only values that are contenders.
13	$CC = \{(q, cst) \mid q \in Q \text{ and } o[q] = cst\}$	- build a set of nodes paired with their contender-costs.

4.3 Finding contender-costs

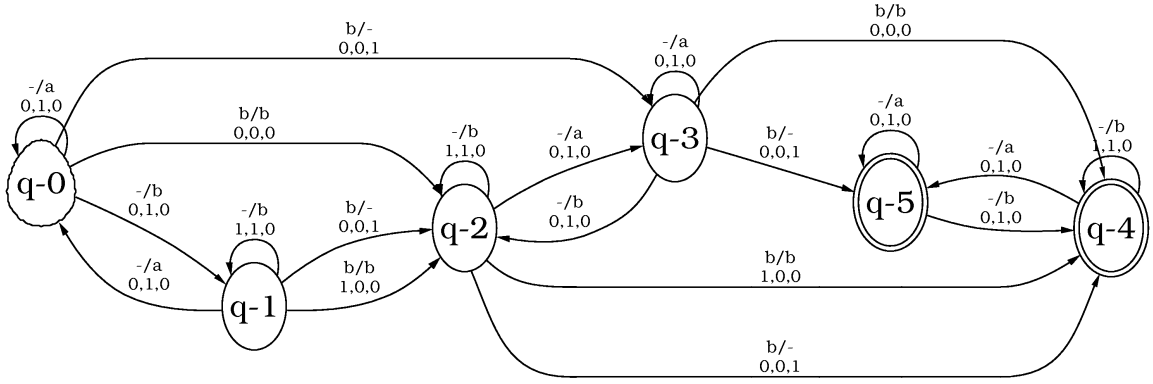
In this section we'll examine in detail steps 1-10 in the algorithm. These first ten steps examine the set of paths to each node and construct a table containing potential contender-costs for each node. The last three steps will then compare the costs at all of the final states and eliminate non-contenders. In (63) I define the contender-costs.

(63) Cost *c* is a **contender-cost** for node *n* iff
c is the cost of a path *p* from the start state to *n*,
there's a ranking *R* under which *p* is among the most harmonic paths to *n*,
and if *n* is final *p* is among the most harmonic paths to any final state under *R*.

In §5.4 I'll show how a table containing all of the contender costs be used to construct the set of candidates that are contenders.

In (64) I present ⟨*A*(bb), *CC, DEP, MAX⟩$^{\rtimes}$. This was first seen in chapter three in the search for optimal outputs for input /bb/ under the ranking *CC ≫ DEP ≫ MAX.

(64) ⟨*A*(bb), *CC, DEP, MAX⟩$^{\rtimes}$



At **step 1** the cost attribute of each node is set to null. Next the cost attribute of the start state is set to {⟨0,0,0⟩} and the start state is added to *H* (**step 3** and **step 4**).

In (65) I give the set of values in the cost attribute of each node. At this point the set $H = \{q_0\}$ – I've shaded the cells of the nodes that are not currently members of H.

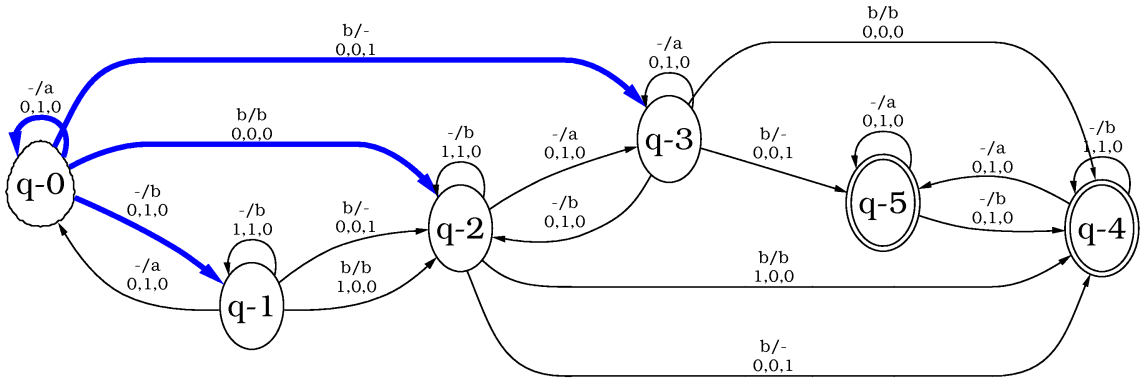
(65) The cost attribute table:

	q_0	q_1	q_2	q_3	q_4	q_5
0.	$\{\langle 0,0,0\rangle\}$	∅	∅	∅	∅	∅

Since H is not empty (**step 5**) we select a node at random from H (**step 6**) and remove it. The node in question must be q_0 since it's the only node currently in the set H.

In **step seven** the four arcs originating at q_0 are examined to see if adding $o[q_0]$ to any arc's weight reveals a new contender-costs for the node at its terminus (**step 8**).

(66) Four arcs to check:



Taking arc (q_0, -, a, ⟨0,1,0⟩, q_0) and adding $o[q_0]$={⟨0,0,0⟩} to its cost ⟨0,1,0⟩ yields {⟨0,1,0⟩}. The union of this set with the cost attribute of the arc's terminus is the set {⟨0,0,0⟩, ⟨0,1,0⟩}. The subset of this set containing only contenders is {⟨0,0,0⟩}, which is no different from the current cost attribute for the arc's terminus so the value of $o[q_0]$ is not updated and node q_0 is not added to H.

Taking arc (q_0, -, b, ⟨0,1,0⟩, q_1) and adding $o[q_0]$={⟨0,0,0⟩} to its cost ⟨0,1,0⟩ yields {⟨0,1,0⟩}, whose union with the cost attribute for the arc's terminus yields the set {⟨0,1,0⟩}. This set has one contender {⟨0,1,0⟩}, which is an improvement over the current cost attribute for the arc's terminus ($o[q_1]$ is currently ∅), so the value of $o[q_1]$ is updated to {⟨0,1,0⟩} and q_1 is added to H.

Taking arc (q_0, b, b, ⟨0,0,0⟩, q_2) and adding $o[q_0]$ to its cost yields {⟨0,0,0⟩}, whose union with the current value for $o[q_2]$ is {⟨0,0,0⟩}. This gives us one contender cost and is an improvement over the current value of ∅ for $o[q_2]$, so $o[q_2]$ is updated to {⟨0,0,0⟩} and q_2 is added to H.

Taking arc (q_0, b, -, ⟨0,0,1⟩, q_3) and adding $o[q_0]$ to its cost yields {⟨0,0,1⟩}, whose union with $o[q_3]$ is {⟨0,0,1⟩}. This reveals one contender, which is an improvement over the current value of ∅ for $o[q_3]$, so $o[q_3]$ is updated to {⟨0,0,1⟩} and q_3 is added to H. The updated cost attributes for the nodes are given in the table in (67).

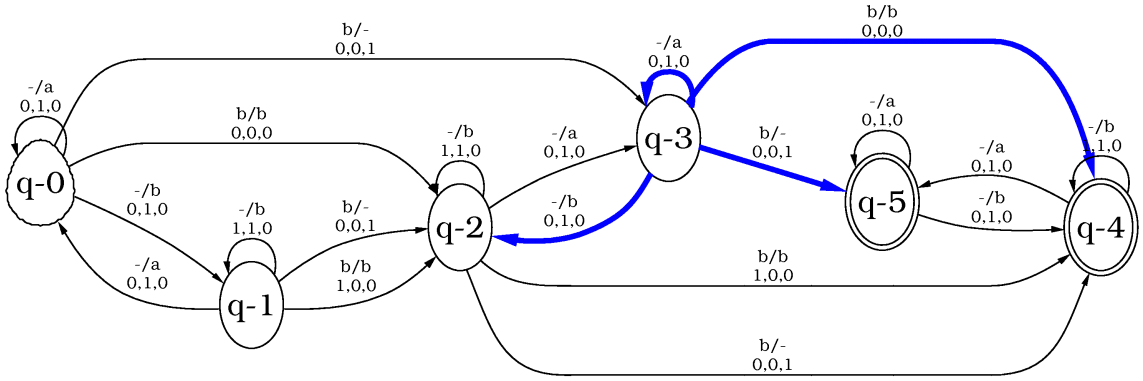
(67) Updated cost attributes:

	q_0	q_1	q_2	q_3	q_4	q_5
0.	{⟨0,0,0⟩}	∅	∅	∅	∅	∅
1.	{⟨0,0,0⟩}	{⟨0,1,0⟩}	{⟨0,0,0⟩}	{⟨0,0,1⟩}	∅	∅

H isn't empty, so another node is selected at random and removed (the 2nd iteration of **step 6**). The selection is random because the algorithm is not searching for the single cheapest (most harmonic) path, but rather for any paths that have costs that are contenders.

Choosing q_3 and removing it from H gives us four arcs originating at q_3 to examine. The arcs to be examined are indicated in bold in (68).

(68) Four arcs to check:



Taking (q_3, -, a, ⟨0,1,0⟩, q_3) and adding $o[q_3] = \{\langle 0,0,1\rangle\}$ to its cost yields $\{\langle 0,1,1\rangle\}$, whose union with $o[q_3]$ contains no new contenders, so $o[q_3]$ isn't updated and q_3 is not added back into H.

Taking (q_3, -, b, ⟨0,1,0⟩, q_2) and adding $o[q_3] = \{\langle 0,0,1\rangle\}$ to its cost yields $\{\langle 0,1,1\rangle\}$, whose union with $o[q_2]$ contains no new contenders, so $o[q_2]$ isn't updated and q_2 isn't added back into H.

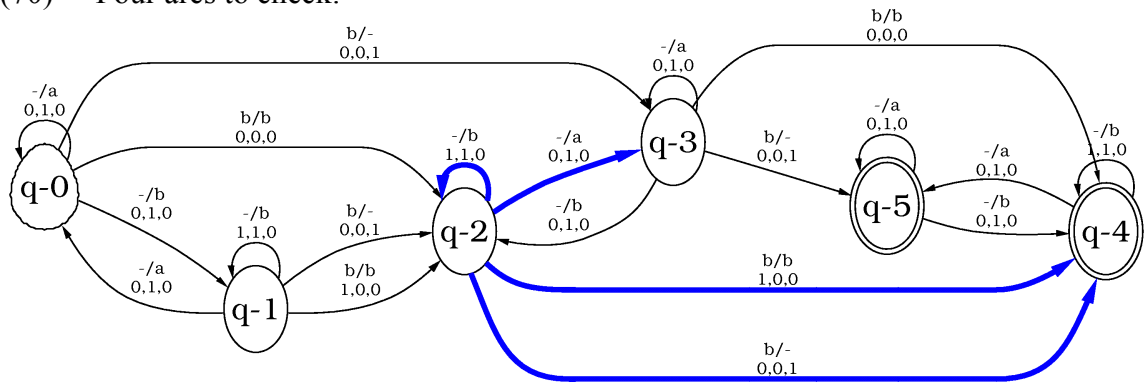
The current cost attributes for nodes q_4 and q_5 are ∅, so arcs (q_3, b, b, ⟨0,0,0⟩, q_4) and (q_3, b, -, ⟨0,0,1⟩, q_5) yield new values of $\{\langle 0,0,1\rangle\}$ for $o[q_4]$ and $\{\langle 0,0,2\rangle\}$ for $o[q_5]$. Because they have been updated, $o[q_4]$ and $o[q_5]$ are added to H. The new values for the cost attributes are given in the table in (69).

(69) Updated cost attributes:

	q_0	q_1	q_2	q_3	q_4	q_5
0.	{⟨0,0,0⟩}	∅	∅	∅	∅	∅
1.	{⟨0,0,0⟩}	{⟨0,1,0⟩}	{⟨0,0,0⟩}	{⟨0,0,1⟩}	∅	∅
2.	{⟨0,0,0⟩}	{⟨0,1,0⟩}	{⟨0,0,0⟩}	{⟨0,0,1⟩}	{⟨0,0,1⟩}	{⟨0,0,2⟩}

Removing node q_2 from H (the 3rd iteration of **step 6**) gives us the four arcs in bold in (70) to examine.

(70) Four arcs to check:



Taking (q_2, -, b, ⟨1,1,0⟩, q_2) and adding $o[q_2] = \{\langle 0,0,0\rangle\}$ to its cost yields $\{\langle 1,1,0\rangle\}$. There are no new contenders in the union of this set with $o[q_2]$.

Taking (q_2, -, a, ⟨0,1,0⟩, q_3) and adding $o[q_2]$ to its cost yields $\{\langle 0,1,0\rangle\}$, whose union with $o[q_3]$ is $\{\langle 0,0,1\rangle, \langle 0,1,0\rangle\}$. This differs from the current value of $o[q_3]$, so $o[q_3]$ is updated to $\{\langle 0,0,1\rangle, \langle 0,1,0\rangle\}$ and q_3 is added back into H. This step gives some cause for concern because we must be certain that nodes can only be added to H finitely many times or there's a danger that the algorithm will fail to terminate. I'll return to this issue in §4.5.

Taking (q_2, b, b, ⟨1,0,0⟩, q_4) and adding $o[q_2]$ to its cost yields {⟨1,0,0⟩}, whose union with $o[q_4]$ reveals new contenders, so $o[q_4]$ is updated to {⟨0,0,1⟩, ⟨1,0,0⟩}.

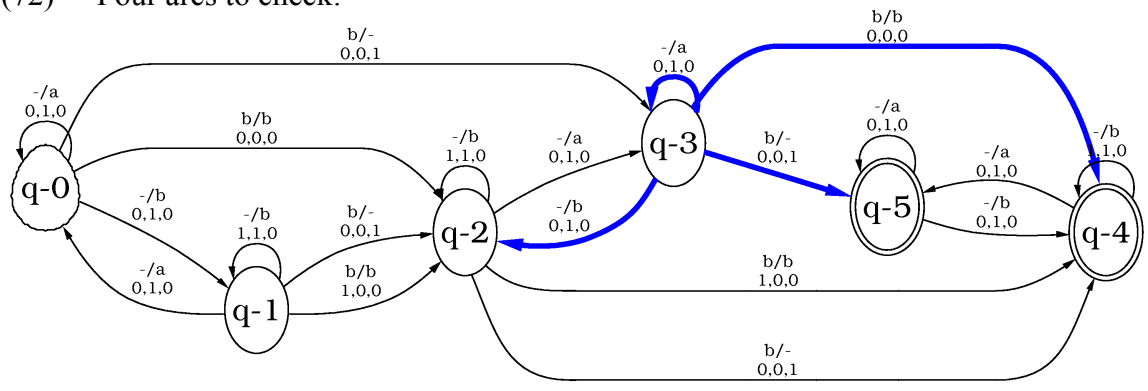
Taking (q_2, b, -, ⟨0,0,1⟩, q_4) and adding $o[q_2]$ to its cost yields {⟨0,0,1⟩}, whose union with o[q_4] has the same set of contenders as o[q4], so it isn't updated. The new values for the cost attributes are given in (71).

(71) Updated cost attributes:

	q_0	q_1	q_2	q_3	q_4	q_5
0.	{⟨0,0,0⟩}	∅	∅	∅	∅	∅
1.	{⟨0,0,0⟩}	{⟨0,1,0⟩}	{⟨0,0,0⟩}	{⟨0,0,1⟩}	∅	∅
2.	{⟨0,0,0⟩}	{⟨0,1,0⟩}	{⟨0,0,0⟩}	{⟨0,0,1⟩}	{⟨0,0,1⟩}	{⟨0,0,2⟩}
3.	{⟨0,0,0⟩}	{⟨0,1,0⟩}	{⟨0,0,0⟩}	{⟨0,0,1⟩, ⟨0,1,0⟩}	{⟨0,0,1⟩, ⟨1,0,0⟩}	{⟨0,0,2⟩}

Removing node q_3 from H (the 4th iteration of **step 6**) gives us the four arcs in bold in (72) to examine.

(72) Four arcs to check:



Arcs (q_3, -, b, ⟨0,1,0⟩, q_2) and (q_3, -, a, ⟨0,1,0⟩, q_3) don't yield any new contenders, so $o[q_3]$ and $o[q_4]$ are not updated.

Taking (q_3, b, b, ⟨0,0,0⟩, q_4) and adding $o[q_3]$ = {⟨0,0,1⟩, ⟨0,1,0⟩} to its cost yields {⟨0,0,1⟩, ⟨0,1,0⟩}, whose union with $o[q_4]$ is {⟨0,0,1⟩, ⟨0,1,0⟩, ⟨1,0,0⟩}. This set contains new contenders, so $o[q_4]$ is updated to {⟨0,0,1⟩, ⟨0,1,0⟩, ⟨1,0,0⟩} and q_4 is added to H.

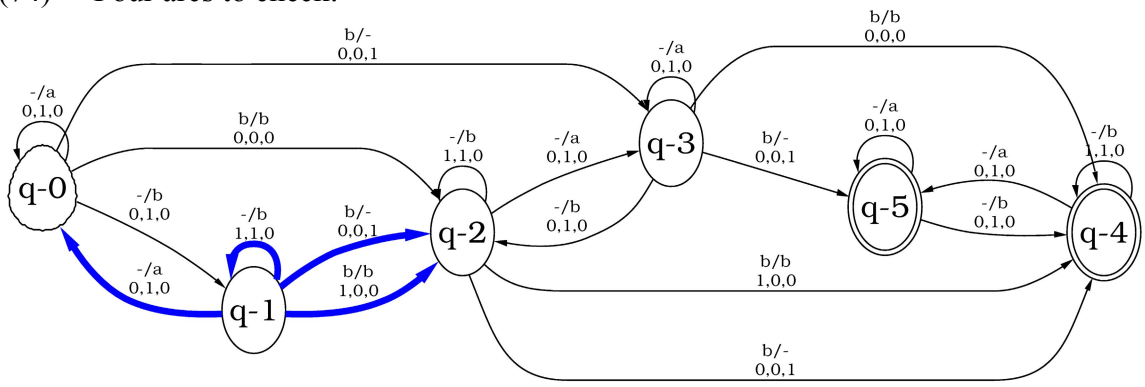
Taking (q_3, b, -, ⟨0,0,1⟩, q_5) and adding $o[q_3]$ to its cost yields {⟨0,0,2⟩, ⟨0,1,1⟩}, whose union $o[q_5]$ is {⟨0,0,2⟩, ⟨0,1,1⟩}, which is different than the current value for $o[q_5]$, so $o[q_5]$ is updated. The updated cost attributes are given in the table in (73).

(73) Updated cost attributes:

	q_0	q_1	q_2	q_3	q_4	q_5
0.	{⟨0,0,0⟩}	∅	∅	∅	∅	∅
1.	{⟨0,0,0⟩}	{⟨0,1,0⟩}	{⟨0,0,0⟩}	{⟨0,0,1⟩}	∅	∅
2.	{⟨0,0,0⟩}	{⟨0,1,0⟩}	{⟨0,0,0⟩}	{⟨0,0,1⟩}	{⟨0,0,1⟩}	{⟨0,0,2⟩}
3.	{⟨0,0,0⟩}	{⟨0,1,0⟩}	{⟨0,0,0⟩}	{⟨0,0,1⟩, ⟨0,1,0⟩}	{⟨0,0,1⟩, ⟨1,0,0⟩}	{⟨0,0,2⟩}
4.	{⟨0,0,0⟩}	{⟨0,1,0⟩}	{⟨0,0,0⟩}	{⟨0,0,1⟩, ⟨0,1,0⟩}	{⟨0,0,1⟩, ⟨0,1,0⟩ ⟨1,0,0⟩}	{⟨0,0,2⟩, ⟨0,1,1⟩}

Removing node q_1 from H (the 5th iteration of **step 6**) gives us the four bold arcs in (74) to examine.

(74) Four arcs to check:



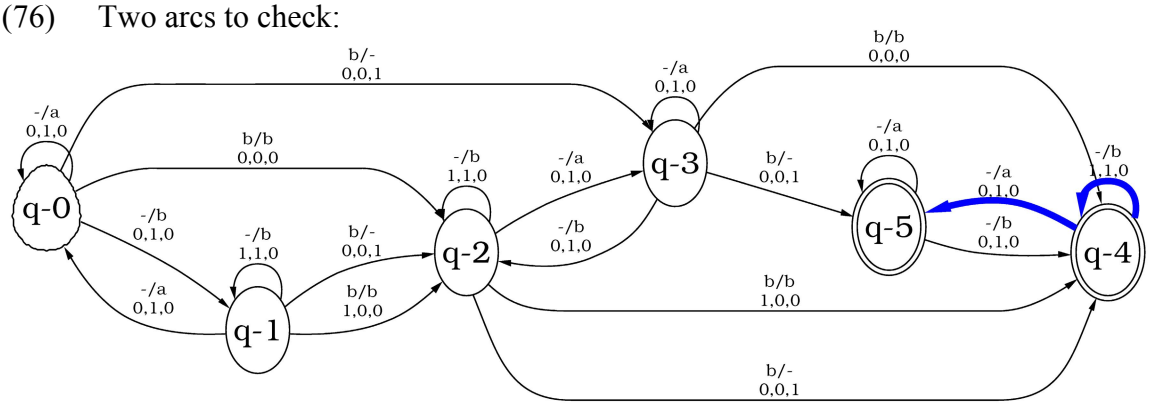
None of these arcs necessitate an update of $o[q_0]$, $o[q_1]$ or $o[q_2]$. The updated table of cost attributes is given in (75) – the only change is that q_1 has been removed from H.

(75) Updated cost attributes:

	q_0	q_1	q_2	q_3	q_4	q_5
5.	{⟨0,0,0⟩}	{⟨0,1,0⟩}	{⟨0,0,0⟩}	{⟨0,0,1⟩, ⟨0,1,0⟩}	{⟨0,0,1⟩, ⟨0,1,0⟩ ⟨1,0,0⟩}	{⟨0,0,2⟩, ⟨0,1,1⟩}

Removing node q_5 from H (the 6th iteration of **step 6**) gives us two arcs to examine.

(76) Two arcs to check:



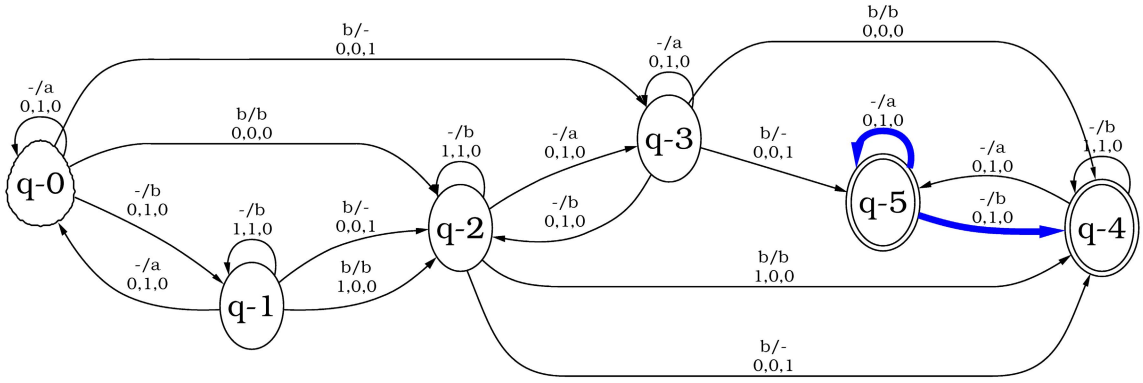
Arc (q_4, -, b, ⟨1,1,0⟩, q_4) doesn't necessitate an update for $o[q_4]$. Taking (q_4, -, a, ⟨0,1,0⟩, q_5) and adding $o[q_4]$ to its cost yields {⟨0,1,1⟩, ⟨0,2,0⟩, ⟨1,1,0⟩}. The union of this set with $o[q5]$ is {⟨0,1,1⟩, ⟨0,2,0⟩, ⟨1,1,0⟩, ⟨0,0,2⟩}. Selecting the contenders in this set yields {⟨0,2,0⟩, ⟨1,1,0⟩, ⟨0,0,2⟩} for $o[q5]$. The newly updated values for the set of cost attributes are given in the table in (77).

(77) Updated cost attributes:

	q_0	q_1	q_2	q_3	q_4	q_5
6.	{⟨0,0,0⟩}	{⟨0,1,0⟩}	{⟨0,0,0⟩}	{⟨0,0,1⟩, ⟨0,1,0⟩}	{⟨0,0,1⟩, ⟨0,1,0⟩ ⟨1,0,0⟩}	{⟨0,0,2⟩, ⟨0,2,0⟩, ⟨1,1,0⟩}

Removing node q_5 from H (the 7th iteration of **step 6**) gives us the two bold arcs in (78) to examine.

(78) Two arcs to check:



Neither (q_5, -, b, ⟨1,1,0⟩, q_5) nor (q_5, -, b, ⟨0,1,0⟩, q_4) necessitate updates of the cost attribute for $o[q_4]$ or $o[q_5]$. At this point H is empty so we move on to step 11 of the algorithm. The updated cost attributes for the nodes are given in (79).

(79) Updated cost attributes:

	q_0	q_1	q_2	q_3	q_4	q_5
7.	{⟨0,0,0⟩}	{⟨0,1,0⟩}	{⟨0,0,0⟩}	{⟨0,0,1⟩, ⟨0,1,0⟩}	{⟨0,0,1⟩, ⟨0,1,0⟩, ⟨1,0,0⟩}	{⟨0,0,2⟩, ⟨0,2,0⟩, ⟨1,1,0⟩}

4.4 Generating the contender candidates

In **step 11** of the algorithm, the union of the cost attributes for the final nodes is taken and the contenders are selected from this set. The union of the contender-costs for each of the final states is given in (80).

(80) Union of the final costs:

$$\bigcup_{q\in F} o[q] = \{\langle 0,0,1\rangle, \langle 0,1,0\rangle, \langle 1,0,0\rangle, \langle 0,0,2\rangle, \langle 0,2,0\rangle, \langle 1,1,0\rangle\}$$

Selecting the contenders from the union of the contender-costs for the final states gives us the set of contenders in (81).

(81) Contenders in the union of the final costs:

$$cont\left(\bigcup_{q\in F} o[q]\right) = \{\langle 0,0,1\rangle, \langle 0,1,0\rangle, \langle 1,0,0\rangle\}$$

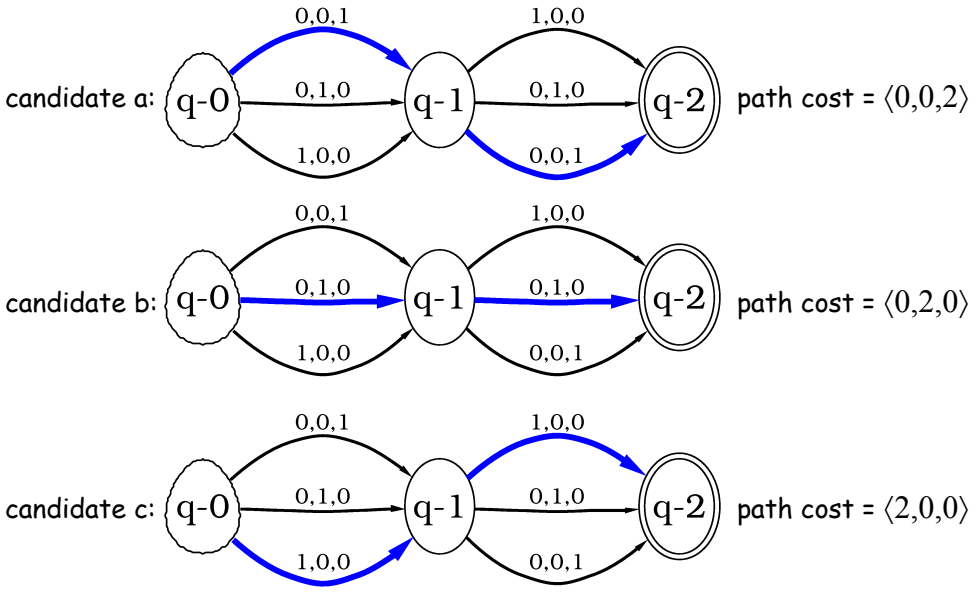
In **step 12** the cost attributes of nodes in the set of final states are updated so as to include only costs that are contenders when compared with all of the possible costs at the finals. Because none of the costs associated with q_5 are contenders in this comparison, the value of $o[q_5]$ is set to $\varnothing$. The final values for the cost attributes are given in (82).

(82) The final values for the cost attributes:

	q_0	q_1	q_2	q_3	q_4	q_5
8.	{⟨0,0,0⟩}	{⟨0,1,0⟩}	{⟨0,0,0⟩}	{⟨0,0,1⟩, ⟨0,1,0⟩}	{⟨0,0,1⟩, ⟨0,1,0⟩, ⟨1,0,0⟩}	∅

At this point, it would be nice if we could use the information in (82) to trim down the machine so that it generated only candidates that are contenders (like we did in the step 13 of OPTIMIZE). Unfortunately, the fact that we're using multiple rankings simultaneously makes this impossible. To see why this is so, consider the hypothetical case illustrated in (83) – I've suppressed the i/o-labels because they aren't relevant to the point at hand.

(83) Three contenders



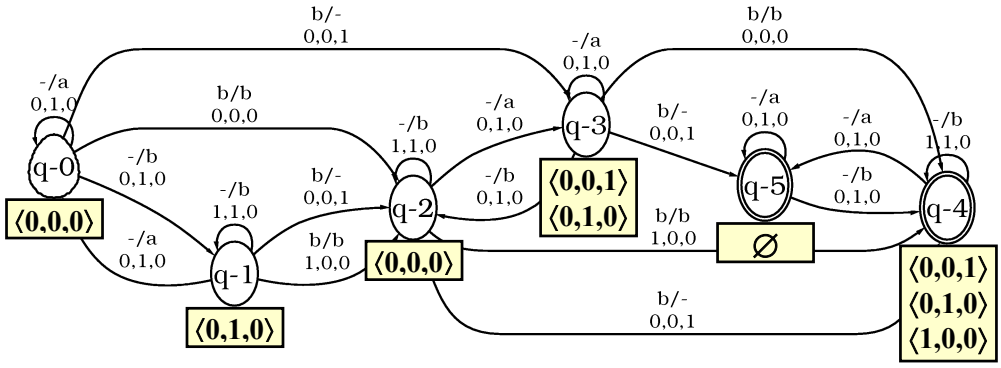
Every arc in (83) participates in path that generates a contender but it's not the case that every path through (83) generates a contender. In fact, of the nine paths, only the ones in bold generate contenders. The contenders can be extracted from (83) by collecting paths

one arc at a time while checking that the cost of the path collected so far is present in the cost attribute for the terminus of the path. This task can be performed with the recursive function *ccand*(q, s, w). When q is the start state, s is the empty string and w is the all-zero cost vector the function returns a contender-candidate paired with its violation vector. I'll go through some concrete examples of the application of this function below in (86).

(84) $$ccand(q,s,w) = \begin{cases} \langle s,w\rangle \text{ if } q \in F, \text{ else} \\ ccand(r,so,w') \text{ where } (q,i,o,v,r) \in \delta \text{ and } w+v=w' \in o[r] \end{cases}$$

In **step 15** of the CONTENDERS algorithm the function *ccand* is used to obtain the set of ⟨*output*, *cost*⟩ pairs for the paths that are contenders. In (85) I mark the nodes with contender-costs and in (86) and (87) I step through three applications of $ccand\left(q_0,\text{-},\overline{0}\right)$.

(85) Nodes annotated with contender-costs:



(86) $ccand\left(q_0,\text{-},\overline{0}\right)$ ← q_0 isn't a final state, arc (q_0, b, -, ⟨0,0,1⟩, q_3) yields

$= ccand\left(q_3,\text{-},\langle 0,0,1\rangle\right)$ ← q_3 isn't a final state, arc (q_3, b, b, ⟨0,0,0⟩, q_4) yields

$= ccand\left(q_4,\text{b},\langle 0,0,1\rangle\right)$ ← q_4 is final,

$= \left\langle \text{b}, \langle 0,0,1\rangle \right\rangle$ ← so a value of $ccand\left(q_0,\text{-},\overline{0}\right)$ is found.

(87) $ccand\left(q_0, \text{-}, \overline{0}\right)$
$= ccand\left(q_2, \text{b}, \langle 0,0,0\rangle\right)$
$= ccand\left(q_3, \text{ba}, \langle 0,1,0\rangle\right)$
$= ccand\left(q_4, \text{bab}, \langle 0,1,0\rangle\right)$
$= \left\langle \text{bab}, \langle 0,1,0\rangle \right\rangle$

$ccand\left(q_0, \text{-}, \overline{0}\right)$
$= ccand\left(q_2, \text{b}, \langle 0,0,0\rangle\right)$
$= ccand\left(q_4, \text{bb}, \langle 1,0,0\rangle\right)$
$= \left\langle \text{bb}, \langle 1,0,0\rangle \right\rangle$

In (88) I give a more familiar representation of the three contenders generated by (85).

(88)

/bb/	*CC	DEP	MAX
a. b			*
b. bab		*	
c. bb	*		

Now that the operation of the algorithm has been laid out I'll turn to the task of proving that it's correct. In §5.5 I'll show that the algorithm is guaranteed to terminate, and once the termination of the algorithm is established I'll show in §5.6 that it is correct.

4.5 Termination of the CONTENDERS algorithm

In the OPTIMIZE algorithm each node in the machine was removed from the set *H* at most one time. This removal occurred each time the cost of the most harmonic path to a node was found. In the CONTENDERS algorithm things are a bit more complicated. Each time a new contender-cost is found for *o*[*q*], node *q* is added to the set *H*. This ensures that the arcs leading away from *q* are assessed to see if any new contender-costs are revealed for the nodes that can be reached from *q* in a step of a single arc.

This is cause for concern because, unless there is an upper bound on how many times a given node can be added to the set *H*, the algorithm could go in circles forever and fail to terminate. In (89) I provide a proof that there is just such an upper bound.

(89) **Theorem**: termination of the CONTENDERS algorithm

The CONTENDERS algorithm can only add a given node to H finitely many times.

proof: Node q is put into H each time a path is found from q_0 to q with a cost c_i where c_i isn't a member of the current value of $o[q]$ but c_i is a contender in $\{c_i\} \cup o[q]$.

Because the node that makes the two endpoints of a cycle can be reached from itself for free, we'll find the cost of a cyclic path only after finding the cost of the corresponding acyclic one. Moreover, because there are no negative costs (a basic premise of OT), no path with a cycle can be more harmonic (have fewer violations) than the corresponding path without the cycle. Thus, the examination of a cycle can never reveal a cost for the node at the cycle's terminus that is any better than the previously known costs. Thus, examination of a cycle can never reveal a new contender and can't require adding the node to H.

There are only finitely many acyclic paths from the start state to any given node in a finite machine. Because there are only finitely many paths from q_0 to q that can contribute an as yet unseen contender value for $o[q]$, node q can only be added to H finitely many times. ■

Even if there are infinitely many actual candidates (input/output mappings) that all get the same set of violations, a state of affairs that can come about if there are zero-cost cycles in the machine, the CONTENDERS algorithm will terminate. This is so because the algorithm isn't computing the actual candidates, but rather the cost vectors associated with the set of candidates that are contenders.

4.6 Correctness of the CONTENDERS algorithm

In (94) I give a correctness proof for the CONTENDERS algorithm. To prove that the algorithm is correct I show that once *H* is empty, the *last* time that any given node was removed from *H* the cost attribute for that node must have contained all contender-costs. First, I will present two lemmas that will be helpful in the proof.

(90) **Harmonic relativity lemma:**

If $v \succ w$ then $v + c \succ w + c$ and $v - c \succ w - c$.

proof: By the definition of harmony, $v \succ w$ just in case for each j such that $v_j > w_j$ there is an $i < j$ such that $v_i < w_i$, and by the definition of the sum of cost vectors, c will add or subtract the same amount to each coordinate in v and w. Thus it is the case that if $v \succ w$ then, because relative inequality is preserved under addition and subtraction, it must be the case that $v + c \succ w + c$ and $v - c \succ w - c$. ■

Two corollaries follow readily from harmonic relativity.

(91) **Corollary 1** of harmonic relativity:

If v is a contender in V then $v + w$ is a contender in $V + w$.

proof: If v is a contender in V then v is among the most harmonic members of V under some ranking of the constraints. Harmonic relativity tells us that adding a fixed value w to every member of V doesn't change their relative harmony. ■

(92) **Corollary 2** of harmonic relativity:

If v is harmonically bounded in V then $v + w$ is harmonically bounded in $V + w$.

If v is harmonically bounded in V then for each ranking there is some $v' \in V$ such that $v' \succ v$. By harmonic relativity we know that adding w to both v' and v doesn't change their relative harmony, so $v + w$ is still harmonically bounded. ■

If there is a path p from the start state q_0 to node q and p's cost w is a contender among the costs of paths to node p, but the cost attribute for q doesn't contain w then I'll say that node q is "missing a contender-cost." Corollaries 1 and 2 will be used to show that if a node q is missing a given contender-cost w then the penultimate node on the path that costs w from q_0 to q must also be missing a contender-cost.

In (93) I give one more lemma that nails down the observation that if a vector is harmonically bounded in a set of vectors it is harmonically bounded in any superset of that set of vectors.

(93) **Bounding lemma:**

If v is harmonically bounded in V then v is harmonically bounded in $W \supseteq V$.

proof: If v is harmonically bounded in V then there is some subset of V consisting of cost vectors that are preferred to v under every constraint ranking. Adding more cost vectors to V can't change this fact. ■

With the lemmas and corollaries established in (90) through (93) in hand, the proof of the correctness of the CONTENDERS algorithm is actually relatively simple.

(94) **Theorem**: correctness of the CONTENDERS algorithm

CONTENDERS(M) associates all and only the contender-costs with each node in M.

proof: First I'll show that at the outset of step 10 the cost attributes for the nodes contain all possible contender-costs for each node. Then I'll show that steps 11-13 of the algorithm eliminate any costs that aren't contenders at the final states.

To derive a contradiction, suppose that the last time that node r was removed from H it was the case that the cost attribute $o[r]$ was missing a contender-cost c_r.

The first and only time that q_0 is removed from H $o[q_0]$ is set to $\{\overline{0}\}$. Because $\overline{0}$ is optimal under all rankings $o[q_0]$ contains all contender-costs for q_0. Thus, by the assumption that $o[r]$ is missing a contender-cost, r is not q_0.

For c_r to be a contender-cost at r there must be a path p from q_0 to r that costs c_r. If q is the penultimate node in p and (q, i, o, w, r) is p's final arc, then $o[q]$ must be missing the contender-cost $(c_r - w)$. Cost $(c_r - w)$ must be a contender at q because if it was harmonically bounded there then $(c_r - w) + w = c_r$ would be bounded at r (corollary 2). Furthermore, $(c_r - w)$ must be missing from $o[q]$ or else c_r would have been discovered when q was removed from H.

Applying this reasoning to p's penultimate arc reveals that p's antepenultimate node must also be missing a contender-cost. Following this reasoning back to p's origin reveals the contradiction because $o[q_0]$ can't be missing any contender-costs.

Once the possible contender-costs are obtained, line 10 pools the costs for the final and lines 11 and 12 delete costs on finals that aren't contenders. ■

The mechanism used here to find the set of contenders is a surprisingly minimal addition to the mechanism used to find a single optimal candidate. This is feat is made possible because we are using a single monolithic evaluator for all of the constraints in the grammar that encodes the preferences of all of the constraints into a single machine. Cascade-style analyses (e.g. Hammond 1995, Eisner 1997abc, Albro 1998ab, Karttunen 1998, Heiberg 1999, Gerdemann and Van Noord 2000) in which one constraint at a time is applied to the candidates to winnow out suboptimal elements can't replicate this feat because there's no single representation of the preferences of all of the constraints.

The amount of work involved in finding the contenders will go up with the number of contenders in contention because the number of cost vectors filling the cells of the cost attribute table will be larger. Crucially, however, this increase in workload is tied directly to the number of actual contenders and not to the number of constraints in the grammar. Though it is indeed possible for there to be $n!$ contenders in a grammar with n constraints the CONTENDERS algorithm will only consider all $n!$ rankings if there is a contender for each ranking. This means that, in general, this algorithm will provide massive savings over the brute-force approach of checking each of the factorially many rankings.

In the next chapter I'll show the value of finding contenders with a more realistic grammar and I'll reformulate CONTENDERS to use Prince's (2002) Elementary Ranking Conditions to represent hypotheses about constraint rankings. This reformulation of the algorithm will make it possible to specify some partial ranking information R in advance and then determine the set of contenders modulo the partial rankings in R.

5 Contenders and Learning

The ability to generate sets of contenders has important ramifications for learning in Optimality Theory, for the contenders are exactly the candidates whose failure can most inform a learner about the structure of the grammar. Tesar (1995a *et seq*.) explains how the "implicit negative evidence" provided by the failure of possible alternative candidates in competition with an actually observed output form can be used to infer rankings among constraints in Optimality Theoretic grammars.

Comparing an observed datum to the entire set of contenders allows the maximal number of inferences to be drawn from each observation. Moreover this can ensure that no piece of relevant information is missed or ignored the first time that it is presented.

In this chapter I will show of how learning algorithms (and presumably human learners) can benefit from access to contenders. In addition to this I'll address the larger questions of what the contenders show us about typology and what the contenders show us about how the difficulty of the learning problem scales when grammars with larger numbers of constraints are considered.

5.1 More constraints: the basic CV syllable theory

So far we've been considering extremely simple grammars with only three or four constraints over an alphabet with only two symbols. In this section we will expand our horizons to the basic CV syllable theory of Prince and Smolensky's (1993) chapter six (henceforth P&S). Since its presentation P&S's basic CV syllable theory it has been the subject of much computational analysis and has served as a sort of benchmark and test-

case for computational implementations of Optimality Theory (Tesar 1995a, Karttunen 1998, Gerdemann and Van Noord 2000, to name a few). For the basic CV syllable theory we will need an alphabet with the following three symbols.

(95) Our new alphabet:

C – a consonant

V – a vowel

x – the syllable boundary

Many implementations of basic CV syllable theory explicitly encode the onset and coda positions into the alphabet itself or assign bracketing and/or tree-like structures to the syllables. For the simple take on this system presented here, the three alphabet symbols given in (95) will suffice.

To implement the basic CV syllable theory we'll need finite-state versions of the basic markedness constraints on syllable structure ONSET and NOCODA (Prince and Smolensky 1993).

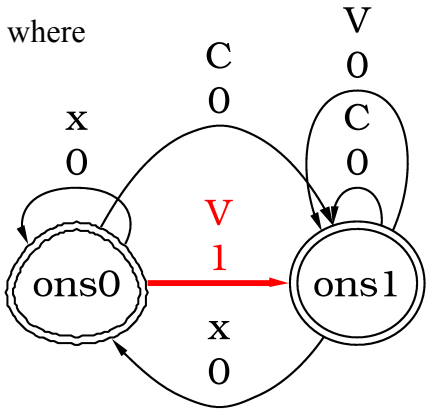
(96) **ONSET:** penalizes syllables without onsets

ONSET = (Q, {c, v, x}, δ, ons-0, F), where

Q = {ons-0, ons-1},

F = {ons-0, ons-1},

δ = {(ons-0, •, x, 0, ons-0),
(ons-0, •, v, 0, ons-0),
(ons-0, •, c, 0, ons-1),
(ons-1, •, c, 0, ons-1),
(ons-1, •, v, 0, ons-0),
(ons-1, •, x, 1, ons-0)}.



The start state of ONSET is the egg-shaped state **ons0** on the left in (96); this is also the state that the constraint is in if a syllable boundary has just been written in the surface form. From this state, writing a consonant takes the constraint to state **ons1** on the right (out of danger of a violation), but writing a vowel garners one violation. That is, a vowel at the beginning of a surface string or immediately after a syllable boundary incurs a violation of ONSET. Given this set-up, it's unnecessary to explicitly encode the syllabic affiliation of the segments in the alphabet.

NOCODA is the partner of ONSET in the basic CV syllable theory. In terms of the representations here NOCODA penalizes syllable boundaries that occur immediately after a consonant. The finite-state version of NOCODA is given in (97).

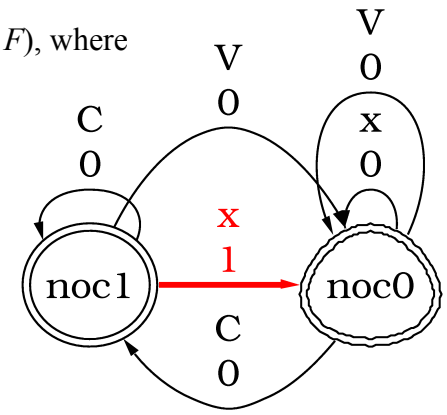
(97) **NOCODA:** penalizes syllables with codas

NOCODA = (Q, {c, v, x}, δ, noc-0, F), where

Q = {noc-0, noc-1},

F = {noc-0, noc-1},

δ = {(noc-0, •, x, 0, noc-0),
(noc-0, •, v, 0, noc-0),
(noc-0, •, c, 0, noc-1),
(noc-1, •, c, 0, noc-1),
(noc-1, •, v, 0, noc-0),
(noc-1, •, x, 1, noc-0)}.



In (97) the start-state of NOCODA is the egg-shaped node **noc0** on the right; this is also the state that the constraint is in whenever a vowel or syllable boundary has just been written in the surface form. From this state, writing a consonant in the surface form takes the constraint to state **noc1** on the left. From state **noc1,** writing a syllable boundary in

the surface form garners a violation of NOCODA. Put simply, this constraint penalizes the sequence '... C x ...'. With this formulation, NOCODA classifies syllables consisting of nothing but lone consonants as violators. It would be easy to formulate a more complex version of the constraint that would only be violated by consonants occurring immediately after a vowel and before a syllable boundary, but this simple version of the constraint will suffice for the current examination.

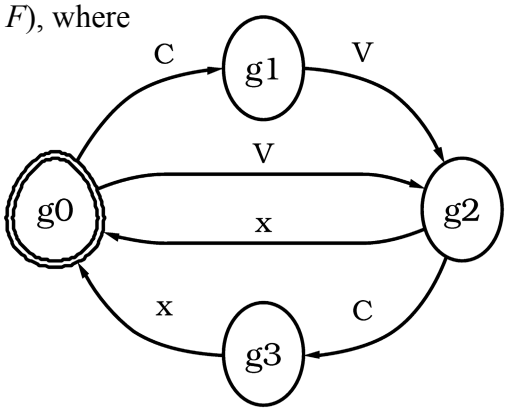
One of the factors that makes the basic CV syllable theory so straightforward is the assumption of a sort of "filter" that limits the scope of the forms under consideration to strings of the form ((C)V(C) x)*. That is, every candidate under consideration is a (possibly empty) string of syllables each of which must contain a vowel and may or may not have a single consonant onset or a single consonant coda. This filter can be given the following finite-state representation.

(98) **g** = (Q, {a, b, x}, δ, noc-0, F), where

Q = {g0, g1, g2, g3},

F = {g0},

δ = {(g0, •, C, ⟨⟩, g1),
(g0, •, V, ⟨⟩, g2),
(g1, •, V, ⟨⟩, g2),
(g2, •, x, ⟨⟩, g0),
(g2, •, C, ⟨⟩, g3),
(g3, •, x, ⟨⟩, g0)}.

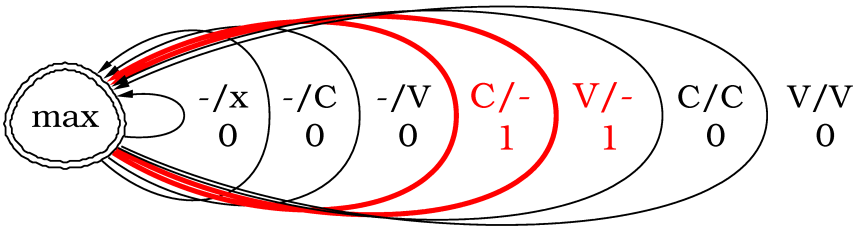


I call this filter **g** because it restricts the set of candidates generated by GEN for each input form. The machine in (98) is like a markedness constraint in that its arcs refer to surface strings and not to underlying segments (thus the wild-cards in the input slots), but it's also

unlike a markedness constraint in that it doesn't assign any violations (thus the empty cost vectors). This filter achieves its intended effect by simply lacking arcs that would allow the generation of the surface strings that it prohibits. Later in this chapter, I'll describe how the filter can be replaced with constraints, but this simplification will suffice for the time being.

The final ingredients for the basic CV syllable theory are some familiar faithfulness constraints. MAX is analogous to Prince and Smolensky's PARSE constraint in that it penalizes deletion. In (99) I repeat the MAX constraint from chapter two with the change that it is now specified over the alphabet $\Sigma = \{C, V, x\}$.

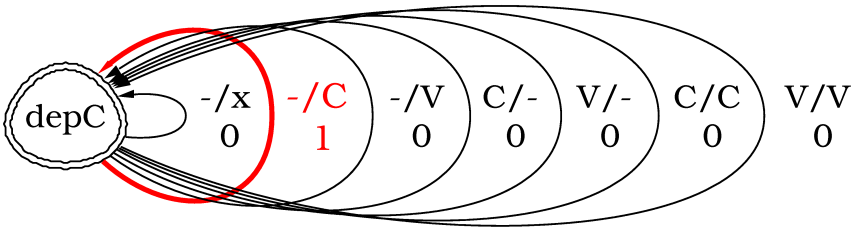
(99) **MAX:** penalizes the deletion of segments



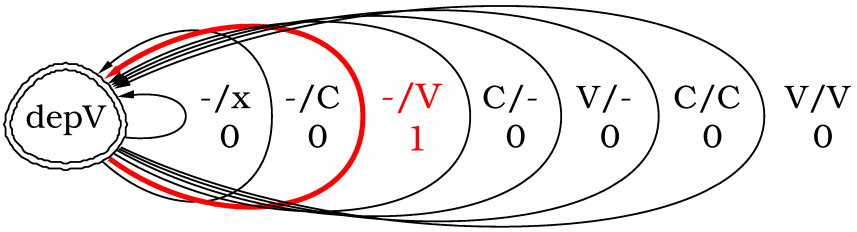
In this presentation I will make the simplifying assumption that inputs are not syllabified. That is, I'll assume that the input alphabet consists of 'C' and 'V'. With this assumption the syllable boundary 'x' can be inserted (the mapping -/x) but there is no need for arcs on which where 'x' is deleted (the mapping x/-).

The constraint DEP is analogous to Prince and Smolensky's FILL constraint in that it penalizes the addition of segments that aren't present in the input. The range of cases covered by P&S's $\text{FILL}^{\text{Onset}}$ and $\text{FILL}^{\text{Nucleus}}$ will be covered here by the consonant-specific and vowel-specific versions of the DEP constraint given in (100) and (101).

(100) **DEPC**: penalizes the insertion of consonants



(101) **DEPV**: penalizes the insertion of vowels

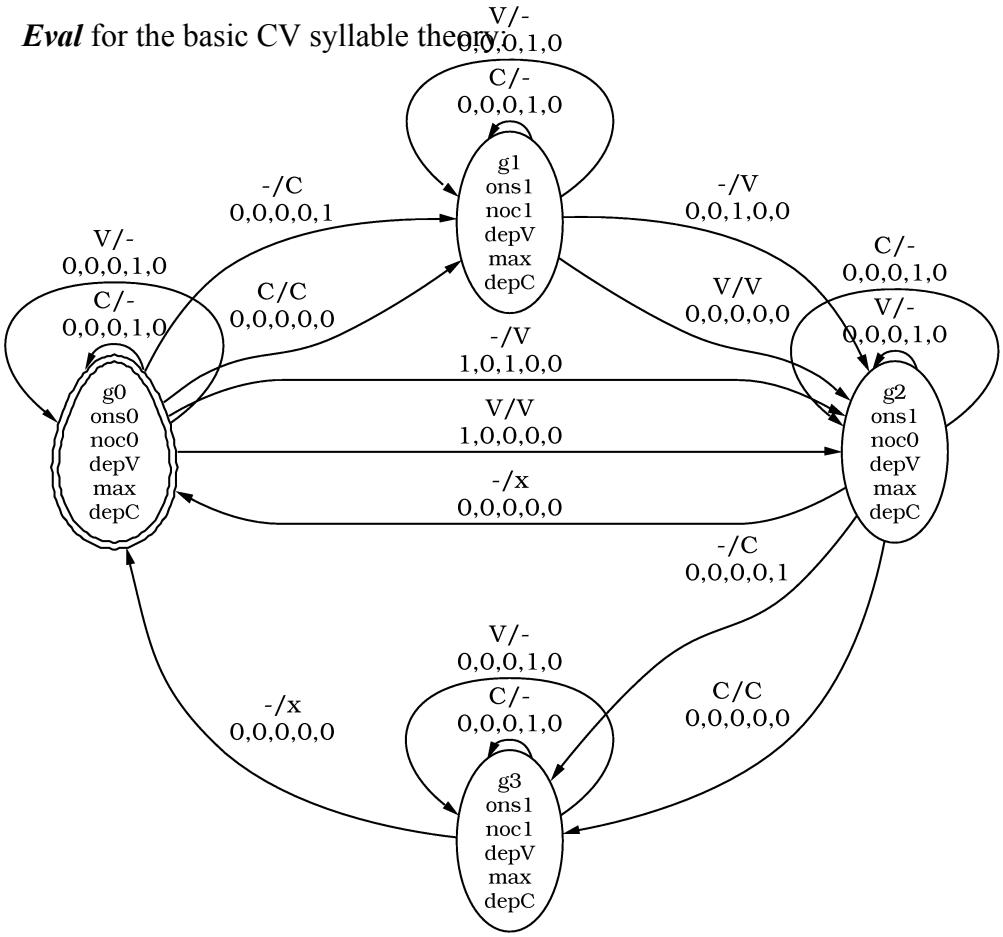


Unlike P&S's PARSE and FILL model, this set-up is not a "containment theory" in which all of the information about which segments have been deleted and inserted can be read off of the surface string. Nor is this system like Correspondence Theory (McCarthy and Prince 1995) in which the segments of the surface forms are indexed to indicate which underlying segments they correspond to.

Because violations of faithfulness constraints are assessed directly on the function that maps the input to the output, there is no need to use correspondence theoretic indexing to assess the effects of basic faithfulness constraints. Of course, the indexing of CT has other uses like enforcing base-reduplicant identity and penalizing the rearrangement of segments in the input/output mapping that aren't readily captured with finite state means.

Eval for languages of the basic CV syllable theory is then obtained from the intersection $\langle \mathbf{g}, \text{ONSET}, \text{NOCODA}, \text{DEPV}, \text{MAX}, \text{DEPC} \rangle^{\boxtimes}$. This is given in (102).

(102) ***Eval*** for the basic CV syllable theory:



In chapter three the grammars for various constraint rankings were derived by altering the order in which the constraints were intersected to build *Eval*. Recall that the order of intersection is reflected in the cost vectors on the arcs of the machine. In this chapter the CONTENDERS algorithm will be redefined so as to find the set of contenders modulo some pre-specified ranking information. With this redefinition, the order in the vectors will no longer be relevant because it will be possible to feed the CONTENDERS algorithm a total ordering of constraints and get the set containing the contenders (which will simply be the optimal forms) under that total ordering of the constraints.

In this fashion the redefined CONTENDERS algorithm will be able to do the duties of both the previous version of the CONTENDERS algorithm and the OPTIMIZE algorithm from chapter two. Moreover, with the redefined CONTENDERS algorithm, it won't be necessary to use different versions of *Eval* for the different rankings of the constraints. In this sense, the machine in (102) is a representation of the grammar for the entire class of languages defined by the basic CV syllable theory.

5.2 Matters of size

As we are consider grammars with more constraints, the issue of the size of the machine resulting from intersecting all of the constraints becomes more relevant. The potential for intersection to multiply the number of states in of great concern, for if the grammars grow explosively as more constraints are added it might become infeasible to work with machines representing entire grammars.

It is certainly true that artificial grammars built from hypothetical constraints of arbitrary complexity can grow geometrically under constraint intersection. Luckily, however, the object of study here is actual human phonological grammars which, at first blush, seem to be fairly well behaved in this regard. Understanding the difference between the worst case scenario and what seems to be the real-world problem can be illuminating, as will become clearer when we consider some approximations to real-world examples.

The grammar resulting from $\langle \mathbf{g}, \text{ONS}, \text{NOCODA}, \text{DEPV}, \text{MAX}, \text{DEPC} \rangle^{\otimes}$ could have 16 states in it (the product of the number of states in machines going into the intersection), but, as we see in (102), there are actually only 4 states. This is so because the phonological

environments distinguished by ONSET and NOCODA are the same as the phonological environments distinguished by the filter **g**. The number of states in *Eval* is exactly the number of unique phonological environments to which the grammar is sensitive. Single-state markedness and faithfulness constraints don't increase the size of the machine at all, and the only way for a multi-state constraint to truly multiply the number of states in the grammar is for it to draw a new distinction in every environment that was distinguished by the grammar before the new constraint was added. This happens readily when just a few constraints are considered in isolation, but as more (real) constraints are considered their environments will overlap whenever they draw distinctions that are being made by constraints already present in the grammar.

Even with the strong position that all of the constraints of Universal Grammar are present in the grammar of every language, the number of states in UG-*Eval* is bounded by the number of unique phonological environments distinguished by the set of constraints. Though this number might be huge, it isn't on the order of the billions and billions that could arise from the geometric nature of intersection. The hope then, is that that by using maximally simple representations of real phonological constraints the grammars won't become so huge and unwieldy that the search for optimal forms and contenders becomes intractable. Even if real grammars are generally too large to work with, the ability to find contenders for fragments or simplifications of those grammars will still be a useful tool for understanding constraint interaction and typology.

5.3 Contenders in the CV syllable theory

To see what kind of insight the CONTENDERS algorithm provides into the typology of the basic CV syllable theory, I constructed a lexicon of the 62 possible input strings made up of C's and V's from one to five segments in length – I'll call this lexicon CV^5. The set of contenders for each input string in CV^5 is given in the tableaux presented in (103) through (109). In each tableau the candidates (contenders) are listed in descending order of harmony for the ranking ONSET ≫ NOCODA ≫ MAX ≫ DEPV ≫ DEPC. Because they are all contenders, at least one of the 120 possible constraint rankings selects each candidate as optimal. To improve the readability of the tableaux I've used a dot '.' to mark medial syllable boundaries and omitted word-final syllable boundaries (the filter **g** ensures that there are no unsyllabified candidates).

(103) Inputs of length 1:

1. /c/	ons	noc	max	depV	depC
a. cv	0	0	0	1	0
b. -	0	0	1	0	0

2. /v/	ons	noc	max	depV	depC
a. cv	0	0	0	0	1
b. -	0	0	1	0	0
c. v	1	0	0	0	0

(104) Inputs of length 2:

3. /cv/	ons	noc	max	depV	depC
a. cv	0	0	0	0	0

4. /vc/	ons	noc	max	depV	depC
a. cv.cv	0	0	0	1	1
b. cv	0	0	1	0	1
c. cv	0	0	1	1	0
d. -	0	0	2	0	0
e. cvc	0	1	0	0	1
f. v.cv	1	0	0	1	0
g. v	1	0	1	0	0
h. vc	1	1	0	0	0

5. /cc/	ons	noc	max	depV	depC
a. cv.cv	0	0	0	2	0
b. -	0	0	2	0	0
c. cvc	0	1	0	1	0

6. /vv/	ons	noc	max	depV	depC
a. cv.cv	0	0	0	0	2
b. -	0	0	2	0	0
c. v.v	2	0	0	0	0

Input #3 in (104) has only one contender because all rankings prefer CV syllables. Seeing how an unknown grammar handles input #4 can be quite informative given the 8-way distinction among contenders for this input. Note, however, that candidates b and c

are homophonous. The former deletes the underlying vowel at the beginning of the input and adds a vowel after the consonant while the latter deletes the underlying consonant and inserts a consonant before the vowel. In this sense, candidates b and c are ambiguous.

(105) Inputs of length 3:

7. /ccc/	ons	noc	max	depV	depC
a. cv.cv.cv	0	0	0	3	0
b. -	0	0	3	0	0
c. cvc.cv, cv.cvc	0	1	0	2	0

8. /vcc/	ons	noc	max	depV	depC
a. cv.cv.cv	0	0	0	2	1
b. cv.cv	0	0	1	2	0
c. cv	0	0	2	0	1
d. -	0	0	3	0	0
e. cvc.cv, cv.cvc	0	1	0	1	1
f. cvc	0	1	1	0	1
g. cvc	0	1	1	1	0
h. v.cv.cv	1	0	0	2	0
i. v	1	0	2	0	0
j. vc.cv, v.cvc	1	1	0	1	0
k. vc	1	1	1	0	0

9. /cvc/	ons	noc	max	depV	depC
a. cv.cv	0	0	0	1	0
b. cv	0	0	1	0	0
c. cvc	0	1	0	0	0

10. /vvc/	ons	noc	max	depV	depC
a. cv.cv.cv	0	0	0	1	2
b. cv.cv	0	0	1	0	2
c. cv	0	0	2	1	0
d. -	0	0	3	0	0
e. cv.cvc	0	1	0	0	2
f. v.v.cv	2	0	0	1	0
g. v.v	2	0	1	0	0
h. v.vc	2	1	0	0	0

11. /ccv/	ons	noc	max	depV	depC
a. cv.cv	0	0	0	1	0
b. cv	0	0	1	0	0

12. /vcv/	ons	noc	max	depV	depC
a. cv.cv	0	0	0	0	1
b. cv	0	0	1	0	0
c. v.cv	1	0	0	0	0

13. /cvv/	ons	noc	max	depV	depC
a. cv.cv	0	0	0	0	1
b. cv	0	0	1	0	0
c. cv.v	1	0	0	0	0

14. /vvv/	ons	noc	max	depV	depC
a. cv.cv.cv	0	0	0	0	3
b. -	0	0	3	0	0
c. v.v.v	3	0	0	0	0

The flipside of ambiguity is a tie, this can be seen between the two forms in row c for input #7. In ambiguity multiple violation sets produce the same output form but in a tie multiple outputs produce the same violation set. The two forms listed as candidate 7c differ only in where the NoCoda violation occurs.

Input #8 /vcc/ shows an eleven-way distinction among the contenders and shows both ambiguity and a tie. The potential for ambiguity and ties can make the problem of inferring information about the grammar from observed i/o pairs even more difficult.

(106) Inputs of length 4:

15. /cccc/	ons	noc	max	depV	depC
a. cv.cv.cv.cv	0	0	0	4	0
b. -	0	0	4	0	0
c. cvc.cvc	0	2	0	2	0

16. /cvcc/	ons	noc	max	depV	depC
a. cv.cv.cv	0	0	0	2	0
b. cv	0	0	2	0	0
c. cvc.cv	0	1	0	1	0
d. cvc	0	1	1	0	0

17. /vccc/	ons	noc	max	depV	depC
a. cv.cv.cv.cv	0	0	0	3	1
b. cv.cv.cv	0	0	1	3	0
c. cv	0	0	3	0	1
d. -	0	0	4	0	0
e. cvc.cv, cv.cvc	0	1	1	2	0
f. cvc	0	1	2	0	1
g. cvc.cvc	0	2	0	1	1
h. v.cv.cv.cv	1	0	0	3	0
i. v	1	0	3	0	0
j. vc	1	1	2	0	0
k. vc.cvc	1	2	0	1	0

18. /vvcc/	ons	noc	max	depV	depC
a. cv.cv.cv.cv	0	0	0	2	2
b. cv.cv	0	0	2	0	2
c. cv.cv	0	0	2	2	0
d. -	0	0	4	0	0
e. cv.cvc.cv, cv.cv.cvc	0	1	0	1	2
f. cv.cvc	0	1	1	0	2
g. cvc	0	1	2	1	0
h. v.v.cv.cv	2	0	0	2	0
i. v.v	2	0	2	0	0
j. v.vc.cv, v.v.cvc	2	1	0	1	0
k. v.vc	2	1	1	0	0

19. /ccvc/	ons	noc	max	depV	depC
a. cv.cv.cv	0	0	0	2	0
b. cv	0	0	2	0	0
c. cv.cvc	0	1	0	1	0
d. cvc	0	1	1	0	0

20. /vcvc/	ons	noc	max	depV	depC
a. cv.cv.cv	0	0	0	1	1
b. cv.cv	0	0	1	0	1
c. cv.cv	0	0	1	1	0
d. cv	0	0	2	0	0
e. cv.cvc	0	1	0	0	1
f. cvc	0	1	1	0	0
g. v.cv.cv	1	0	0	1	0
h. v.cv	1	0	1	0	0
i. v.cvc	1	1	0	0	0

21. /cvvc/	ons	noc	max	depV	depC
a. cv.cv.cv	0	0	0	1	1
b. cv.cv	0	0	1	0	1
c. cv.cv	0	0	1	1	0
d. cv	0	0	2	0	0
e. cv.cvc	0	1	0	0	1
f. cvc	0	1	1	0	0
g. cv.v.cv	1	0	0	1	0
h. cv.v	1	0	1	0	0
i. cv.vc	1	1	0	0	0

22. /vvvc/	ons	noc	max	depV	depC
a. cv.cv.cv.cv	0	0	0	1	3
b. cv.cv.cv	0	0	1	0	3
c. cv	0	0	3	1	0
d. -	0	0	4	0	0
e. cv.cv.cvc	0	1	0	0	3
f. v.v.v.cv	3	0	0	1	0
g. v.v.v	3	0	1	0	0
h. v.v.vc	3	1	0	0	0

23. /cccv/	ons	noc	max	depV	depC
a. cv.cv.cv	0	0	0	2	0
b. cv	0	0	2	0	0
c. cvc.cv	0	1	0	1	0

24. /vccv/	ons	noc	max	depV	depC
a. cv.cv.cv	0	0	0	1	1
b. cv.cv	0	0	1	0	1
c. cv.cv	0	0	1	1	0
d. cv	0	0	2	0	0
e. cvc.cv	0	1	0	0	1
f. v.cv.cv	1	0	0	1	0
g. v.cv	1	0	1	0	0
h. vc.cv	1	1	0	0	0

25. /cvcv/	ons	noc	max	depV	depC
a. cv.cv	0	0	0	0	0

26. /vvcv/	ons	noc	max	depV	depC
a. cv.cv.cv	0	0	0	0	2
b. cv	0	0	2	0	0
c. v.v.cv	2	0	0	0	0

27. /ccvv/	ons	noc	max	depV	depC
a. cv.cv.cv	0	0	0	1	1
b. cv.cv	0	0	1	0	1
c. cv.cv	0	0	1	1	0
d. cv	0	0	2	0	0
e. cv.cv.v	1	0	0	1	0
f. cv.v	1	0	1	0	0

28. /vcvv/	ons	noc	max	depV	depC
a. cv.cv.cv	0	0	0	0	2
b. cv	0	0	2	0	0
c. v.cv.v	2	0	0	0	0

29. /cvvv/	ons	noc	max	depV	depC
a. cv.cv.cv	0	0	0	0	2
b. cv	0	0	2	0	0
c. cv.v.v	2	0	0	0	0

30. /vvvv/	ons	noc	max	depV	depC
a. cv.cv.cv.cv	0	0	0	0	4
b. -	0	0	4	0	0
c. v.v.v.v	4	0	0	0	0

The length 4 inputs show more ambiguity, more ties, and more forms with eleven-way distinctions among their contenders. Like #3, input #25 CVCV has only one contender.

(107) Inputs of length 5:

31. /ccccc/	ons	noc	max	depV	depC
a. cv.cv.cv.cv.cv	0	0	0	e.	0
b. -	0	0	e.	0	0
c. cvc.cvc.cv, cvc.cv.cvc, cv.cvc.cvc	0	2	0	3	0

32. /vcccc/	ons	noc	max	depV	depC
a. cv.cv.cv.cv.cv	0	0	0	d.	1
b. cv.cv.cv.cv	0	0	1	d.	0
c. cv	0	0	d.	0	1
d.	0	0	e.	0	0
e. cvc	0	1	3	0	1
f. cvc.cvc.cv, cvc.cv.cvc, cv.cvc.cvc	0	2	0	2	1
g. cvc.cvc	0	2	1	2	0
h. v.cv.cv.cv.cv	1	0	0	d.	0
i. v	1	0	d.	0	0
j. vc	1	1	3	0	0
k. vc.cvc.cv, vc.cv.cvc, v.cvc.cvc	1	2	0	2	0

33. /cvccc/	ons	noc	max	depV	depC
a. cv.cv.cv.cv	0	0	0	3	0
b. cv	0	0	3	0	0
c. cvc	0	1	2	0	0
d. cvc.cvc	0	2	0	1	0

34. /vvccc/	ons	noc	max	depV	depC
a. cv.cv.cv.cv.cv	0	0	0	3	2
b. cv.cv.cv	0	0	2	3	0
c. cv.cv	0	0	3	0	2
d. -	0	0	e.	0	0
e. cv.cvc	0	1	2	0	2
f. cvc.cv, cv.cvc	0	1	2	2	0
g. cv.cvc.cvc	0	2	0	1	2
h. v.v.cv.cv.cv	2	0	0	3	0
i. v.v	2	0	3	0	0
j. v.vc	2	1	2	0	0
k. v.vc.cvc	2	2	0	1	0

35. /ccvcc/	ons	noc	max	depV	depC
a. cv.cv.cv.cv	0	0	0	3	0
b. cv	0	0	3	0	0
c. cv.cvc.cv, cv.cv.cvc	0	1	0	2	0
d. cvc	0	1	2	0	0

36. /vcvcc/	ons	noc	max	depV	depC
a. cv.cv.cv.cv	0	0	0	2	1
b. cv.cv.cv	0	0	1	2	0
c. cv.cv	0	0	2	0	1
d. cv	0	0	3	0	0
e. cv.cvc.cv, cv.cv.cvc	0	1	0	1	1
f. cv.cvc	0	1	1	0	1
g. cvc.cv, cv.cvc	0	1	1	1	0
h. cvc	0	1	2	0	0
i. v.cv.cv.cv	1	0	0	2	0
j. v.cv	1	0	2	0	0
k. v.cvc.cv, v.cv.cvc	1	1	0	1	0
l. v.cvc	1	1	1	0	0

37. /cvvcc/	ons	noc	max	depV	depC
a. cv.cv.cv.cv	0	0	0	2	1
b. cv.cv.cv	0	0	1	2	0
c. cv.cv	0	0	2	0	1
d. cv	0	0	3	0	0
e. cv.cvc.cv, cv.cv.cvc	0	1	0	1	1
f. cv.cvc	0	1	1	0	1
g. cvc.cv, cv.cvc	0	1	1	1	0
h. cvc	0	1	2	0	0
i. cv.v.cv.cv	1	0	0	2	0
j. cv.v	1	0	2	0	0
k. cv.vc.cv, cv.v.cvc	1	1	0	1	0
l. cv.vc	1	1	1	0	0

38. /vvvcc/	ons	noc	max	depV	depC
a. cv.cv.cv.cv.cv	0	0	0	2	3
b. cv.cv.cv	0	0	2	0	3
c. cv.cv	0	0	3	2	0
d.	0	0	e.	0	0
e. cv.cv.cvc.cv, cv.cv.cv.cvc	0	1	0	1	3
f. cv.cv.cvc	0	1	1	0	3
g. cvc	0	1	3	1	0
h. v.v.v.cv.cv	3	0	0	2	0
i. v.v.v	3	0	2	0	0
j. v.v.vc.cv, v.v.v.cvc	3	1	0	1	0
k. v.v.vc	3	1	1	0	0

39. /cccvc/	ons	noc	max	depV	depC
a. cv.cv.cv.cv	0	0	0	3	0
b. cv	0	0	3	0	0
c. cvc	0	1	2	0	0
d. cvc.cvc	0	2	0	1	0

(108) Inputs of length 5: -*continued*

40. /vccvc/	ons	noc	max	depV	depC
a. cv.cv.cv.cv	0	0	0	2	1
b. cv.cv.cv	0	0	1	2	0
c. cv.cv	0	0	2	0	1
d. cv	0	0	3	0	0
e. cv.cvc	0	1	1	1	0
f. cvc	0	1	2	0	0
g. cvc.cvc	0	2	0	0	1
h. v.cv.cv.cv	1	0	0	2	0
i. v.cv	1	0	2	0	0
j. vc.cvc	1	2	0	0	0

41. /cvcvc/	ons	noc	max	depV	depC
a. cv.cv.cv	0	0	0	1	0
b. cv.cv	0	0	1	0	0
c. cv.cvc	0	1	0	0	0

42. /vvcvc/	ons	noc	max	depV	depC
a. cv.cv.cv.cv	0	0	0	1	2
b. cv.cv.cv	0	0	1	0	2
c. cv.cv	0	0	2	1	0
d. cv	0	0	3	0	0
e. cv.cv.cvc	0	1	0	0	2
f. cvc	0	1	2	0	0
g. v.v.cv.cv	2	0	0	1	0
h. v.v.cv	2	0	1	0	0
i. v.v.cvc	2	1	0	0	0

43. /ccvvc/	ons	noc	max	depV	depC
a. cv.cv.cv.cv	0	0	0	2	1
b. cv.cv.cv	0	0	1	2	0
c. cv.cv	0	0	2	0	1
d. cv	0	0	3	0	0
e. cv.cv.cvc	0	1	0	1	1
f. cv.cvc	0	1	1	0	1
g. cv.cvc	0	1	1	1	0
h. cvc	0	1	2	0	0
i. cv.cv.v.cv	1	0	0	2	0
j. cv.v	1	0	2	0	0
k. cv.cv.vc	1	1	0	1	0
l. cv.vc	1	1	1	0	0

44. /vcvvc/	ons	noc	max	depV	depC
a. cv.cv.cv.cv	0	0	0	1	2
b. cv.cv.cv	0	0	1	0	2
c. cv.cv	0	0	2	1	0
d. cv	0	0	3	0	0
e. cv.cv.cvc	0	1	0	0	2
f. cvc	0	1	2	0	0
g. v.cv.v.cv	2	0	0	1	0
h. v.cv.v	2	0	1	0	0
i. v.cv.vc	2	1	0	0	0

45. /cvvvc/	ons	noc	max	depV	depC
a. cv.cv.cv.cv	0	0	0	1	2
b. cv.cv.cv	0	0	1	0	2
c. cv.cv	0	0	2	1	0
d. cv	0	0	3	0	0
e. cv.cv.cvc	0	1	0	0	2
f. cvc	0	1	2	0	0
g. cv.v.v.cv	2	0	0	1	0
h. cv.v.v	2	0	1	0	0
i. cv.v.vc	2	1	0	0	0

46. /vvvvc/	ons	noc	max	depV	depC
a. cv.cv.cv.cv.cv	0	0	0	1	d.
b. cv.cv.cv.cv	0	0	1	0	d.
c. cv	0	0	d.	1	0
d. -	0	0	e.	0	0
e. cv.cv.cv.cvc	0	1	0	0	d.
f. v.v.v.v.cv	d.	0	0	1	0
g. v.v.v.v	d.	0	1	0	0
h. v.v.v.vc	d.	1	0	0	0

47. /ccccv/	ons	noc	max	depV	depC
a. cv.cv.cv.cv	0	0	0	3	0
b. cv	0	0	3	0	0
c. cvc.cv.cv, cv.cvc.cv	0	1	0	2	0

48. /vcccv/	ons	noc	max	depV	depC
a. cv.cv.cv.cv	0	0	0	2	1
b. cv.cv.cv	0	0	1	2	0
c. cv.cv	0	0	2	0	1
d. cv	0	0	3	0	0
e. cvc.cv.cv, cv.cvc.cv	0	1	0	1	1
f. cvc.cv	0	1	1	0	1
g. cvc.cv	0	1	1	1	0
h. v.cv.cv.cv	1	0	0	2	0
i. v.cv	1	0	2	0	0
j. vc.cv.cv, v.cvc.cv	1	1	0	1	0
k. vc.cv	1	1	1	0	0

49. /cvccv/	ons	noc	max	depV	depC
a. cv.cv.cv	0	0	0	1	0
b. cv.cv	0	0	1	0	0
c. cvc.cv	0	1	0	0	0

50. /vvccv/	ons	noc	max	depV	depC
a. cv.cv.cv.cv	0	0	0	1	2
b. cv.cv.cv	0	0	1	0	2
c. cv.cv	0	0	2	1	0
d. cv	0	0	3	0	0
e. cv.cvc.cv	0	1	0	0	2
f. v.v.cv.cv	2	0	0	1	0
g. v.v.cv	2	0	1	0	0
h. v.vc.cv	2	1	0	0	0

Note the 10-way distinction among contenders with no ambiguity and no ties for input #40.

(109) Inputs of length 5: *-fin*

51. /ccvcv/	ons	noc	max	depV	depC
a. cv.cv.cv	0	0	0	1	0
b. cv.cv	0	0	1	0	0

52. /vcvcv/	ons	noc	max	depV	depC
a. cv.cv.cv	0	0	0	0	1
b. cv.cv	0	0	1	0	0
c. v.cv.cv	1	0	0	0	0

53. /cvvcv/	ons	noc	max	depV	depC
a. cv.cv.cv	0	0	0	0	1
b. cv.cv	0	0	1	0	0
c. cv.v.cv	1	0	0	0	0

54. /vvvcv/	ons	noc	max	depV	depC
a. cv.cv.cv.cv	0	0	0	0	3
b. cv	0	0	3	0	0
c. v.v.v.cv	3	0	0	0	0

55. /cccvv/	ons	noc	max	depV	depC
a. cv.cv.cv.cv	0	0	0	2	1
b. cv.cv.cv	0	0	1	2	0
c. cv.cv	0	0	2	0	1
d. cv	0	0	3	0	0
e. cvc.cv.cv	0	1	0	1	1
f. cvc.cv	0	1	1	1	0
g. cv.cv.cv.v	1	0	0	2	0
h. cv.v	1	0	2	0	0
i. cvc.cv.v	1	1	0	1	0

56. /vccvv/	ons	noc	max	depV	depC
a. cv.cv.cv.cv	0	0	0	1	2
b. cv.cv.cv	0	0	1	0	2
c. cv.cv	0	0	2	1	0
d. cv	0	0	3	0	0
e. cvc.cv.cv	0	1	0	0	2
f. v.cv.cv.v	2	0	0	1	0
g. v.cv.v	2	0	1	0	0
h. vc.cv.v	2	1	0	0	0

57. /cvcvv/	ons	noc	max	depV	depC
a. cv.cv.cv	0	0	0	0	1
b. cv.cv	0	0	1	0	0
c. cv.cv.v	1	0	0	0	0

58. /vvcvv/	ons	noc	max	depV	depC
a. cv.cv.cv.cv	0	0	0	0	3
b. cv	0	0	3	0	0
c. v.v.cv.v	3	0	0	0	0

59. /ccvvv/	ons	noc	max	depV	depC
a. cv.cv.cv.cv	0	0	0	1	2
b. cv.cv.cv	0	0	1	0	2
c. cv.cv	0	0	2	1	0
d. cv	0	0	3	0	0
e. cv.cv.v.v	2	0	0	1	0
f. cv.v.v	2	0	1	0	0

60. /vcvvv/	ons	noc	max	depV	depC
a. cv.cv.cv.cv	0	0	0	0	3
b. cv	0	0	3	0	0
c. v.cv.v.v	3	0	0	0	0

61. /cvvvv/	ons	noc	max	depV	depC
a. cv.cv.cv.cv	0	0	0	0	3
b. cv	0	0	3	0	0
c. cv.v.v.v	3	0	0	0	0

62. /vvvvv/	ons	noc	max	depV	depC
a. cv.cv.cv.cv.cv	0	0	0	0	5
b. -	0	0	5	0	0
c. v.v.v.v.v	5	0	0	0	0

Perhaps, the most interesting bit of information gleaned about the basic CV syllable theory by running the forms of CV^5 through the CONTENDERS algorithm, is the twelve-way distinction among contenders for some of the input forms. This twelve-way distinction is only a tenth of variety we might have expected given that each of the 120 rankings could define a unique language. Nonetheless, these distinctions are slightly finer grained than the typology that is usually described for the basic CV syllable theory.

5.4 Contenders and typology in the CV syllable theory

Prince and Smolensky (1993: 104) illustrate how various rankings of the ONSET, NOCODA, PARSE, FILL^{Onset}, and FILL^{Nucleus} constraints generate the CV syllable structure typology. In (110) I recreate Prince and Smolensky's typology table replacing PARSE and FILL with MAX and DEP respectively. The notations "del." and "ep." in the cells indicate whether requirements on onsets and codas are achieved via epenthesis or deletion.

(110) Basic syllable typology:

		onset required		not required
		ONS, MAX ≫ DEPC	ONS, DEPC ≫ MAX	MAX, DEPC ≫ ONS
coda forbidden	NOC, MAX ≫ DEPV	**1.** Σ^{CV} ep. ep.	**2.** Σ^{CV} del. ep.	**8.** $\Sigma^{(C)VC}$ ep.
	NOC, DEPV ≫ MAX	**3.** Σ^{CV} ep. del.	**4.** Σ^{CV} del. del.	**7.** $\Sigma^{(C)VC}$ del.
coda allowed	MAX, DEPV ≫ NOC	**5.** $\Sigma^{CV(C)}$ ep.	**6.** $\Sigma^{CV(C)}$ del.	**9.** $\Sigma^{(C)V(C)}$

Though there are nine points in the typology described in (110), it's possible for a twelve-way distinction to arise among the contenders for a given input based on differences in the way consonant clusters are dealt with under various constraint rankings. Consider, for example, the contenders for the input /cvvcc/ presented in (111) below.

(111) Twelve contenders:

43. /ccvvc/	ons	noc	max	depV	depC
a. cv.cv.cv.cv	0	0	0	2	1
b. cv.cv.cv	0	0	1	2	0
c. cv.cv	0	0	2	0	1
d. cv	0	0	3	0	0
e. cv.cv.cvc	0	1	0	1	1
f. cv.cvc	0	1	1	0	1
g. cv.cvc	0	1	1	1	0
h. cvc	0	1	2	0	0
i. cv.cv.v.cv	1	0	0	2	0
j. cv.v	1	0	2	0	0
k. cv.cv.vc	1	1	0	1	0
L. cv.vc	1	1	1	0	0

Candidates a, b, c, and d in (111) arise under rankings in which onsets are required and codas are forbidden. According to the numbers I've given the languages in the table in (110), candidate a is an instance of language 1, candidate b is an instance of language 2, candidate c is an instance of language 3, and candidate d is an instance of language 4.

(112) Language five:

/ccvvc/	ons	noc	max	depV	depC	
e. cv.cv.cvc	0	1	0	1	1	≈ language 5
f. cv.cvc	0	1	1	0	1	

Candidates e and f both avoid the potential ONSET violation word-initially with an epenthetic consonant. They differ in that candidate e breaks up the consonant cluster at the end of the input string with an epenthetic vowel but candidate f eliminates the final cluster by deleting one of the final consonants. These changes are necessitated by the fact that **g** requires all surface forms to be of the shape ((C)V(C)x)*. If MAX dominates DEPV then candidate e is selected if DEPV dominates MAX then candidate f is selected.

(113) Language six:

/ccvvc/	ons	noc	max	depV	depC	
g. cv.cvc	0	1	1	1	0	≈ language 6
h. cvc	0	1	2	0	0	

Candidates g and h both avoid a potential ONSET violation by deleting the initial vowel of the input. They differ in that g separates the cluster at the end of the input with an epenthetic vowel and h deletes one of the consonants in the final cluster. Again, the choice between these candidates will be determined by the ranking of MAX and DEPV.

(114) Languages seven and eight:

/ccvvc/	ons	noc	max	depV	depC	
i. cv.cv.v.cv	1	0	0	2	0	= language 7
j. cv.v	1	0	2	0	0	= language 8

Candidates i and j both allow onsetless syllables but don't allow codas. The former avoids NOCODA violations with vowel epenthesis and the latter avoids NOCODA violations by deleting consonants.

(115) Language nine:

/ccvvc/	ons	noc	max	depV	depC	
k. cv.cv.vc	1	1	0	1	0	≈ language 9
L. cv.vc	1	1	1	0	0	

Candidates k and l avoid faithfulness violations by allowing ONSET and NOCODA to be violated. Like the candidate pairs e & f and g & h, these candidates differ in whether the consonant cluster at the end of the input is repaired via epenthesis or via deletion.

Adding one row to (110) so that the ranking MAX ≫ DEPV ≫ NOC is distinguished from DEPV ≫ MAX ≫ NOC would cover the twelve-way distinction that the contenders

in (109) show. This isn't an omission on the part of P&S as they explicitly set aside treatment of forms with clusters. It is, however, illuminating to see exactly what the addition of an inviolable constraint, like the prohibition against clusters, does to the typology.

Given that the twelve-way distinction first arises for inputs of length five, one might reasonably ask whether longer inputs could reveal still more distinctions. In this case, a logical assessment of the constraint interactions (and letting the algorithm churn through the 131 thousand inputs up to length 16) can reassure us that we have identified all of the distinctions made by the constraints. For more complex problems with more constraints such an approach would, of course, not be feasible.

5.5 Elementary Ranking Conditions

An Elementary Ranking Condition or ERC is a representation of the disparities in the violations incurred by two candidates (Prince 2002a, 2002b). ERCs are fundamentally comparative; one candidate is designated as the winner and the ERC describes, for each constraint, whether it prefers the designated winner, the other candidate, or neither.

Elementary Ranking Conditions are recorded as a vectors of the symbols **`W`**, **`L`**, and **`e`**. The order of the terms in the vector corresponds to an arbitrary (but fixed) ordering of the constraints. I'll call this fixed constraint-order the "key" because it allows us to read the ERCs. The occurrence the symbol **`W`** in the i^{th} coordinate of an ERC indicates that the i^{th} constraint in the key prefers the designated winner, likewise the occurrence of **`L`** in the i^{th} coordinate of an ERC indicates that the i^{th} constraint in the key prefers the designated

loser, and the occurrence of the symbol **e** in the i^{th} coordinate of an ERC indicates that the i^{th} constraint in the key prefers neither the designated winner nor the designated loser.

In (116) I give a function that takes two vectors, the first of which is assumed to be the winner and the second the loser, and yields an ERC encoding the rankings under which the first cost vector triumphs over the second.

(116) Building ERCs by comparing cost vectors:

$$erc(\langle w_1,...,w_n\rangle,\langle v_1,...,v_n\rangle)\begin{cases}\varnothing \text{ if } \langle w_1,...,w_n\rangle = \langle v_1,...,v_n\rangle, \text{ else}\\ \langle e_1,...,e_n\rangle \text{ where } e_i = \mathbf{e} \text{ if } w_i = v_i,\\ \qquad\qquad\qquad\qquad e_i = \mathbf{L} \text{ if } w_i > v_i, \text{ and}\\ \qquad\qquad\qquad\qquad e_i = \mathbf{W} \text{ if } w_i < v_i.\end{cases}$$

The convention that the *erc* function returns null when an ERC is compared to itself will allow us to ignore the "degenerate" ERC whose coordinates are all **e**.

For the sake of illustration, let's assume that we know that the input /vc/ surfaces as the fully faithful output [vc] under some unknown constraint ranking and ask what we can glean about the unknown ranking from this fact. In (117) I give the set of contenders for the input /vc/. For each candidate I give the ERC that's derived from comparing that candidate to the observed output (candidate a). The order of the terms in the ERCs (the key) corresponds to the order of constraints in the columns in (117). Note that this order is <u>not</u> the ranking of the unknown grammar that selects the input/output pair /vc/→ [vc].

(117)

/vc/	ons	noc	max	depV	depC	ERCs
a. ☞ vc	1	1	0	0	0	∅
b. v	1	0	1	0	0	⟨e,L,W,e,e⟩
c. v.cv	1	0	0	1	0	⟨e,L,e,W,e⟩
d. cvc	0	1	0	0	1	⟨L,e,e,e,W⟩
e. -	0	0	2	0	0	⟨L,L,W,e,e⟩
f. cv	0	0	1	1	0	⟨L,L,W,W,e⟩
g. cv	0	0	1	0	1	⟨L,L,W,e,W⟩
h. cv.cv	0	0	0	1	1	⟨L,L,e,W,W⟩

In (118) through (122) I'll go through the candidates in (117) one by one, comparing each to the observed winner a and discussing the ERC derived from this comparison.

(118)

/vc/	ons	noc	max	depV	depC	
a. ☞ vc	1	1	0	0	0	ERC
b. v	1	0	1	0	0	⟨e,L,W,e,e⟩

The ERC for candidate b is ⟨**e**,**L**,**W**,**e**,**e**⟩. The value at the first coordinate in the ERC is **e** because both candidate b and candidate a get one violation of ONSET – ONSET does not prefer either candidate. The value at the second coordinate in the ERC is **L** because candidate a, the winner, gets one violation of NOCODA while candidate b gets none – that is, NOCODA prefers the loser b. The value at the third coordinate is **W** because candidate a gets no violations of MAX while candidate b gets one – MAX prefers the winner a. The values at the fourth and fifth coordinates in the ERC are both **e** because candidates a and b get the same number of violations of DEPV and DEPC.

The ERC ⟨**e**,**L**,**W**,**e**,**e**⟩ tells us that the constraint in the third coordinate, MAX, prefers candidate a while the constraint in the second coordinate, NOCODA, prefers b. Thus, in order to select candidate a over candidate b, MAX must dominate NOCODA.

(119)

/vc/	ons	noc	max	depV	depC	
a. ☞ vc	1	1	0	0	0	ERC
c. v.cv	1	0	0	1	0	⟨e,L,e,W,e⟩

The ERC for candidate c is ⟨**e,L,e,W,e**⟩. This indicates that candidate a is preferred DEPV while candidate c is preferred by NOCODA. Thus DEPV must dominate NOCODA.

(120)

/vc/	ons	noc	max	depV	depC	
a. ☞ vc	1	1	0	0	0	ERC
d. cvc	0	1	0	0	1	⟨L,e,e,e,W⟩

The ERC for candidate d is ⟨**L,e,e,e,W**⟩. This tells us that DEPC prefers a while ONSET prefers d. Thus DEPC must dominate ONSET.

(121)

/vc/	ons	noc	max	depV	depC	
a. ☞ vc	1	1	0	0	0	ERC
e. -	0	0	2	0	0	⟨L,L,W,e,e⟩

The ERC for candidate e is ⟨**L,L,W,e,e**⟩. This is more informative than the previous three ERCs, it indicates that MAX a while ONSET and NOCODA both prefer candidate e. This means that MAX must dominate both ONSET and NOCODA.

(122)

/vc/	ons	noc	max	depV	depC	
a. ☞ vc	1	1	0	0	0	ERC
f. cv	0	0	1	1	0	⟨L,L,W,W,e⟩

The ERC for candidate f is ⟨**L,L,W,W,e**⟩. This indicates that MAX and DEPV both prefer candidate a but ONSET and NOCODA both prefer candidate f. Thus either MAX or DEPV must dominate both ONSET and NOCODA.

The ERCs for candidates g and h encode the same kind of disjunction seen in the ERC for candidate f.

ERCs are extremely useful tools for reasoning about ranking arguments. Their value is twofold: first, they are generalizations that abstract away from specific violations and encode just relevant differences between candidates, and second, as we'll see in the next section, they can be logically manipulated to draw nonobvious inferences.

5.6 Reasoning with Elementary Ranking Conditions

Prince (2002a) shows that Elementary Ranking Conditions are analogous to statements of three-valued relevance logics. The reader is referred to Prince's detailed analysis for discussion of this fact. For our current purposes there are two logical operations on ERCs that can facilitate their use in reasoning about rankings.

Fusion is an operation that combines two Elementary Ranking Conditions to produce a third ERC that is entailed by the truth of the first two. In fusion the coordinates of the ERCs are combined as in (123).

(123) **Fusion of Entries**: (Prince 2002a: 8)

$X \circ X = X$ – Fusing X with itself yields X; this is idempotence

$X \circ \mathbf{e} = X$
$\mathbf{e} \circ X = X$
} Fusing X with **e** yields X; **e** is the identity operator

$X \circ \mathbf{L} = \mathbf{L}$
$\mathbf{L} \circ X = \mathbf{L}$
} Fusing anything with **L** yields **L**; **L** is dominant

The fusion of ERC_1 and ERC_2, denoted $ERC_1 \circ ERC_2$, is simply the coordinate-wise fusion of the terms that make up ERC_1 and ERC_2. This is defined in (124).

(124) **Fusion of ERCs**:

$$\langle x_1, x_2, ..., x_n \rangle \circ \langle y_1, y_2, ..., y_n \rangle = \langle x_1 \circ y_1, x_2 \circ y_2, ..., x_n \circ y_n \rangle$$

For an illustration of fusion, consider ⟨**W,L,e**⟩ and ⟨**e,W,L**⟩. Fusing these ERCs yields ⟨**W,L,L**⟩. This straightforwardly capturing the transitivity of domination – if constraint 1 must dominate constraint 2 which in turn must dominate 3 then 1 must dominate 2 and 3.

Fusion is even more useful in its ability to eliminate disjunctions. For instance, given ERC_1 = ⟨**W,W,L**⟩ which says that either constraint 1 or constraint 2 must dominate constraint 3 and given ERC_2 = ⟨**W,L,W**⟩ which says that either constraint 1 or constraint 3 must dominate constraint 2 we can easily obtain $ERC_1 \circ ERC_2$ = ⟨**W,L,L**⟩ which says that constraint 1 must dominate both constraints 2 and 3.

While fusion determines what third ERC is entailed by the truth of two others, it is also the case that an ERC can be entailed from the truth of a single other ERC. In (125) I give Prince's definition of entailment for nontrivial ERCs. An ERC is nontrivial if it contains at least one **W** and one **L**. Because the ERCs that we are working with here come from contenders, they are all nontrivial. For further discussion of trivial vs. nontrivial ERCs, see Prince (2002a: 2).

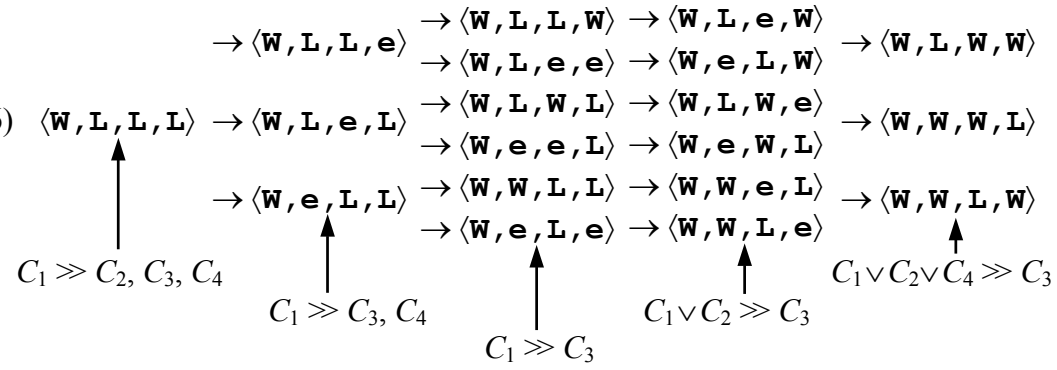
(125) A nontrivial ERC = $\langle x_1, ..., x_n \rangle$ entails E' = $\langle y_1, ..., y_n \rangle$ iff

$$\{ i \mid x_i = \mathbf{W} \} \subseteq \{ i \mid y_i = \mathbf{W} \} \text{ and } \{ j \mid x_j = \mathbf{L} \} \supseteq \{ j \mid y_j = \mathbf{L} \}$$

In prose, E_1 entails E_2 if the coordinates containing **W** in E_1 are a subset of those containing **W** in E_2 and the coordinates containing **L** in E_1 are a superset of those containing **L** in E_2.

If we consider **L** to be the most informative (strongest) bit of information, **W** to be the least informative (weakest) and **e** to intermediate then ERCs simply entail weaker ERCs. In (126) I illustrate the 18 nontrivial ERCs that are entailed by ⟨**W,L,L,L**⟩.[14]

(126)



A set of Elementary Ranking Conditions constitutes a hypothesis about how the constraints of a grammar are ranked. Entailment relations among ERCs make it possible to reduce a set of ERCs to a smaller set that encodes the same information. This is handy because it can reduce the number of ERCs that we need to keep track of. This will become relevant in the next section when we turn to the problem of determining whether a set of ERCs is internally consistent. For now note in (127) how the set of ERCs from the losing contenders in (117) can be reduced from seven down to three statements.

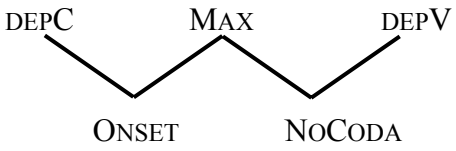
(127)
- **(b)** ⟨**e,L,W,e,e**⟩ ← entailed by **(e)**
- **(c)** ⟨**e,L,e,W,e**⟩
- **(d)** ⟨**L,e,e,e,W**⟩
- **(e)** ⟨**L,L,W,e,e**⟩

  } all of the information is contained in this subset
- **(f)** ⟨**L,L,W,W,e**⟩ ← entailed by **(e)**
- **(g)** ⟨**L,L,W,e,W**⟩ ← entailed by **(e)**
- **(h)** ⟨**L,L,e,W,W**⟩ ← result of **(c)** ∘ **(d)**

[14] Recall that a nontrivial ERC is one with at least one **W** and one **L**.

Removing the entailed ERCs in (127) eliminates the disjunctive ones **(f)**, **(g)**, and **(h)**, leaving only their simpler brethren in **(c)**, **(d)**, and **(e)**. Expressing the information encoded by these three ERCs with a conventional ranking diagram gives us (128).

(128)



5.7 Recursive Constraint Demotion and contenders

Tesar's (1995a) Recursive Constraint Demotion algorithm is used, modified, and discussed in a large body of work (RCD; Tesar 1995a, 1996ab, 1997ab, 1998, 2000, and Tesar and Smolensky 1998, 2000 for example. In addition to being useful as a method for finding constraint rankings that are consistent with observed winner/loser pairs, RCD can be used to check a set of Elementary Ranking Conditions for internal consistency. In (129) I provide a recursive function that captures the consistency detecting aspect of Recursive Constraint Demotion based on Prince's (2002b) discussion of ERCs and RCD.

(129) $$consistent(E) = \begin{cases} \text{true} & \text{if } E = \varnothing, \\ consistent(F) & \text{if } F = \{v \mid v \in E,\ v_i = \mathbf{e},\ \text{and } w \in E \rightarrow w_i \neq \mathbf{W}\} \neq E \\ \text{otherwise false} & \end{cases}$$

The first clause in (129) establishes the base case that an empty set of ERCs is consistent. The second clause is where the recursion comes in; it says that given subset F of E whose i^{th} coordinates are **e** where there's no member of E whose i^{th} coordinate is **W**, if set F is consistent and (not identical to E) then E is consistent, otherwise the function returns false.

It is easy to recast the *contenders* function from the previous chapter in terms of ERCs and RCD. To determine whether a vector v in a set of vectors V is a contender in V, every vector in the set is compared to v and represented as an ERC. If the resulting set of ERCs is consistent then v is a contender.

(130) $contenders(V) = \{v \mid v \in V \text{ and } consistent(\{erc(v, w) \mid w \in V\})\}$

To illustrate the redefined version of the *contenders* function, let's consider the problem of finding the contenders among $V = \{\langle 0,0,2\rangle, \langle 1,0,1\rangle, \langle 0,2,0\rangle, \langle 0,1,1\rangle\}$.

(131) Illustration of a consistency check:

To check whether the vector $\langle 0,0,2\rangle$ is among the contenders in V we express V as a set of ERCs relative to the vector $\langle 0,0,2\rangle$ and check that set for consistency.

$erc(\langle 0,0,2\rangle, \langle 0,0,2\rangle) = \varnothing$
$erc(\langle 0,0,2\rangle, \langle 1,0,1\rangle) = \langle W,e,L\rangle$
$erc(\langle 0,0,2\rangle, \langle 0,2,0\rangle) = \langle e,W,L\rangle$
$erc(\langle 0,0,2\rangle, \langle 0,1,1\rangle) = \langle e,W,L\rangle$
$consistent(\{\langle W,e,L\rangle, \langle e,W,L\rangle\})$ = true, so $\langle 0,0,2\rangle$ is a contender in E.

Repeating this check for the vectors $\langle 1,0,1\rangle$ and $\langle 0,2,0\rangle$ reveals that they are also contenders in the set V. Moving on to the last vector $\langle 0,1,1\rangle$ we detect inconsistency.

(132) Expressing V as ERCs relative to $\langle 0,1,1\rangle$ gives us:

$erc(\langle 0,1,1\rangle, \langle 0,0,2\rangle) = \langle e,L,W\rangle$
$erc(\langle 0,1,1\rangle, \langle 1,0,1\rangle) = \langle W,L,e\rangle$
$erc(\langle 0,1,1\rangle, \langle 0,2,0\rangle) = \langle e,W,L\rangle$
$erc(\langle 0,1,1\rangle, \langle 0,1,1\rangle) = \varnothing$
$consistent(\{\langle e,L,W\rangle, \langle W,L,e\rangle, \langle e,W,L\rangle\})$ = false

The non-contending vector ⟨0, 1, 1⟩ shows an instance of collective harmonic bounding (Samek-Lodovici and Prince 1999). That is, ⟨0, 1, 1⟩ is not bounded by any particular member of *V* but rather by the combination of ⟨0, 0, 2⟩ and ⟨0, 2, 0⟩ as alternatives.

In the recursive test for consistency, once the set of ERCs is reduced to the set *E* = {⟨e, L, W⟩, ⟨e, W, L⟩} only the 1st coordinate contains no instances of **L**, but the set of ERCs with an **e** in the 1st coordinate is {⟨e, L, W⟩, ⟨e, W, L⟩} which is identical to *E*, so the second clause of *consistent* fails and the function returns false.

With the redefinition of the *contenders* function in (130) in terms of Elementary Ranking Conditions, it's now the case that the information that is being manipulated to find the contenders among a set of cost vectors is exactly the kind of ranking information that is gleaned from comparing losing contender-candidates to an observed output. This suggests one final reformulation of the *contenders* function.

In (133) I reformulate *contenders* as a function over a set of vectors *V* and a set of elementary ranking conditions *E*. This definition is just like the one in (130) with the addition that before each set of ERCs is checked for consistency, the set *E* is added to it. This allows us to give the *contenders* function some partial information about the ranking that we have in mind when generating contenders.

(133) $contenders(V, E) = \{v \mid v \in V \text{ and } consistent(E \cup \{erc(v, w) \mid w \in V\})\}$

If *E* is empty then no ranking information is pre-specified and the function is just like the initial formulation in (130). On the other hand, if *E* contains some information about the ranking then *contenders* will return only those cost vectors that are possibly optimal

given that information. Finally, if E defines a total ordering of the constraints then the function will simply return the vector in V that is optimal under the ranking in E.

5.8 Beyond error-driven learning

Having reformulated the *contenders* function to take into account a set of known ERCs we can modify the CONTENDERS algorithm to take utilize the same information. In (134) I redefine CONTENDERS so that it takes an input, as set of constraints CON, and a set of ERCs and returns the set of contenders modulo the pre-specified ERCs.

(134) CONTENDERS(in, CON, $ERCs$) = Cn

0	$\langle A(in), \text{CON}\rangle^{\otimes} = (Q, \Sigma, \delta, q_0, F)$	- Intersect the input & constraints
1	**for** each $q \in Q$	- Set the cost attribute for each node in the machine to null.
2	**do** $o[q] \leftarrow \varnothing$	
3	$o[q_0] \leftarrow \{\overline{0}\}$ where $k = \lvert\text{CON}\rvert$	- Set the cost of the start to $\{\overline{0}\}$
4	$H \leftarrow \{q_0\}$	- Put the start state in the set H.
5	**while** $H \neq \varnothing$	- While H is not empty loop as follows:
6	$H \leftarrow H - \{q_u\}$	remove a node, q_u, from H and
7	**for** each $(q_u, i, o, w, q_v) \in \delta$	for each arc from q_u to q_v,
8	**if** $V=\{o[q_v] \cup o[q_u]+w\}$ and $contenders(V, ERCs) \neq o[q_v]$	if there are any new contenders,
9	**do** $o[q_v] \leftarrow contenders(V, ERCs)$ $H \leftarrow H \cup \{q_v\}$	update the cost attributes for $o[q_v]$ and add q_v to H.
10	$Cn \leftarrow contenders\left(\bigcup_{q\in F} o[q],\ E\right)$	- Select the contenders from the union of the cost attributes for the finals

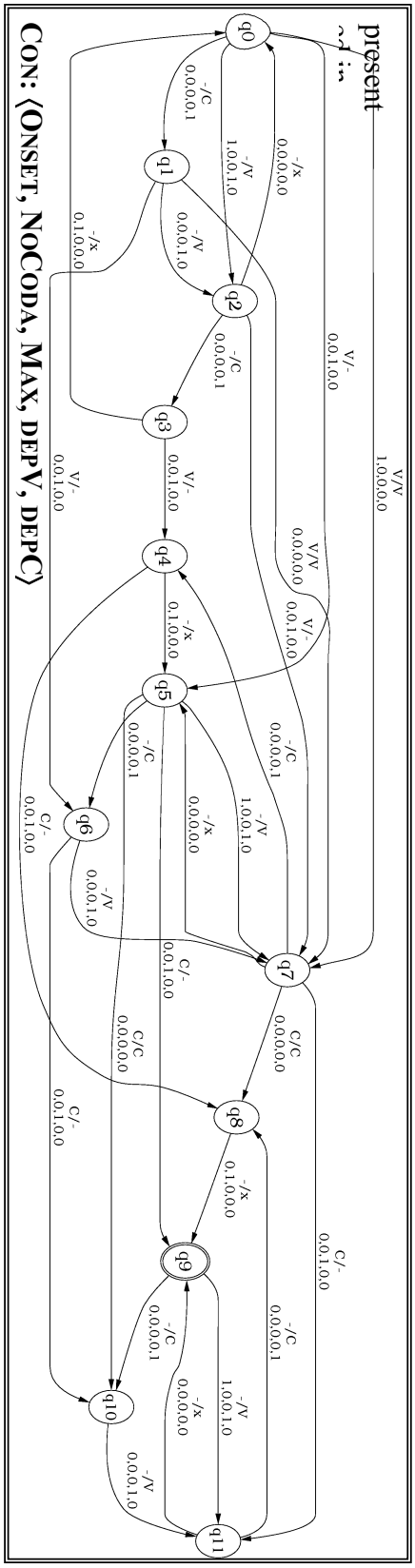
When no elementary ranking conditions are specified this new version of the algorithm outputs the same candidate set as the original CONTENDERS algorithm presented in chapter five. For instance, if ⟨**g**, ONSET, NOCODA, MAX, DEPV, DEPC⟩ is CON then CONTENDERS(vv, CON, ∅) = {⟨0,0,0,0,2⟩, ⟨0,0,2,00⟩, ⟨2,0,0,0,0⟩} which gives us the candidates in (135).

(135) Three contenders:

/vv/	ons	noc	max	depV	depC
a. cv.cv	0	0	0	0	2
b. -	0	0	2	0	0
c. v.v	2	0	0	0	0

Observing candidate c in (135) as the output allows us to turn the failed contenders into ERCs. Candidate b yields ⟨L, e, W, e, e⟩ and a yields ⟨L, e, e, e, W⟩.

When predicting the optimal output for another input it's now possible to constrain the predictions to those that are viable given the currently known ERCs. So, given E = {⟨L, e, W, e, e⟩, ⟨L, e, e, e, W⟩}, we can ask what does CONTENDERS(vc, CON, E) return? To answer this consider the machine ⟨A(vc), CON⟩$^{\otimes}$ presented in (136).



The revised CONTENDERS algorithm is much the same as the version presented in chapter five. As such, I won't illustrate the algorithm in detail again but rather will focus on the points where the presence of the pre-specified ERCs change the outcome.

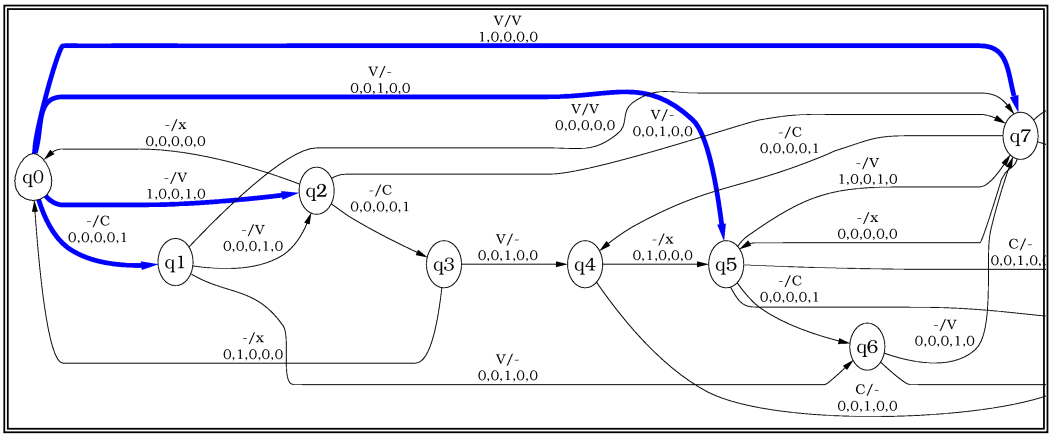
In (137) I give the cost table that results after the cost of the start-state has been set to $\overline{0}$ in **step 3** of the algorithm. In the cost tables shown here I won't bother to write brackets around the set that make up the contents of each cell. As in chapter five, unshaded cells in the cost table indicate nodes that are currently in set *H*.

(137)

	q_0	q_1	q_2	q_3	q_4	q_5	...	q_{11}
1.	⟨0,0,0,0,0⟩	∅	∅	∅	∅	∅		∅

In **step 4** of the algorithm *H* was set to $\{q_0\}$, so there is only one node in *H* to remove. Taking q_0 out of *H* and checking the arcs originating at it gives us the bold arcs in (138).

(138) There are four arcs originating at q_0:



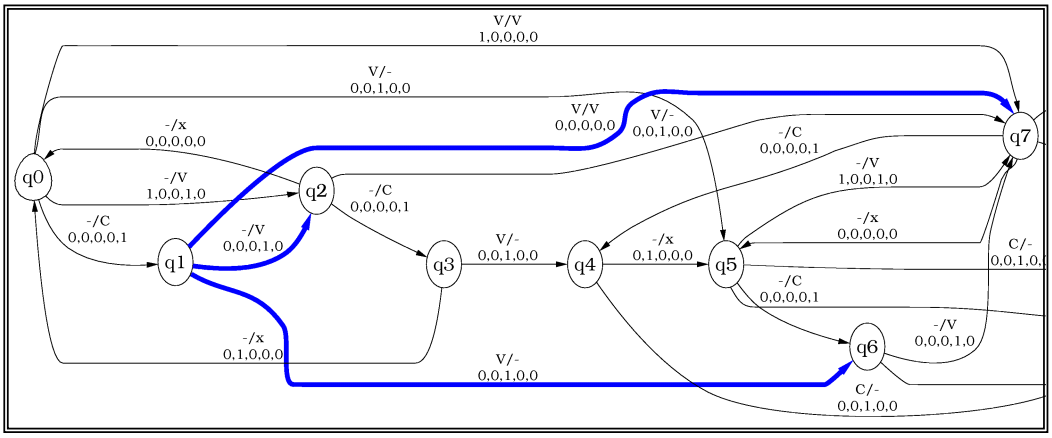
At this point there's no competition at the nodes, so each arc supplies a new contender-cost for the node at its terminus, and each node is added to *H*. In (139) I update the table.

(139)

2.	q_0	q_1	q_2	...	q_5	...	q_7	...	q_{11}
	⟨0,0,0,0,0⟩	⟨0,0,0,0,1⟩	⟨1,0,0,1,0⟩		⟨0,0,1,0,0⟩		⟨1,0,0,0,0⟩		∅

Next a node is selected at random from *H* and the arcs originating at it are checked. Selecting node q_1 gives us the three arcs in bold in (140) to check.

(140) There are three arcs originating at q_1:



Now we have some competition. The arc from q_1 to q_2 reveals a new potential cost of ⟨0,0,0,1,1⟩ for q_2. Taking this cost together with the current estimate for q_2 gives us the set {⟨1,0,0,1,0⟩, ⟨0,0,0,1,1⟩}. Feeding this set to the function *cont* from chapter five would return both values as contenders, but feeding this set along with our pre-specified ERCs to the *contenders* function returns only ⟨1,0,0,1,0⟩ as a contender because ⟨0,0,0,1,1⟩ cannot ever beat ⟨1,0,0,1,0⟩ under the pre-specified ERCs. This is illustrated in (141).

(141) *contenders*({⟨1,0,0,1,0⟩, ⟨0,0,0,1,1⟩},{⟨L,e,W,e,e⟩, ⟨L,e,e,e,W⟩})

i) checking ⟨1,0,0,1,0⟩:
erc(⟨1,0,0,1,0⟩, ⟨0,0,0,1,1⟩) = ⟨L,e,e,e,W⟩
consistent({⟨L,e,e,e,W⟩} ∪ {⟨L,e,W,e,e⟩, ⟨L,e,e,e,W⟩}) = true

ii) checking ⟨1,0,0,1,0⟩:
erc(⟨0,0,0,1,1⟩, ⟨1,0,0,1,0⟩) = ⟨W,e,e,e,L⟩
consistent({⟨W,e,e,e,L⟩} ∪ {⟨L,e,W,e,e⟩, ⟨L,e,e,e,W⟩}) = false

We have basically the same result for the arc from q_1 to q_7. This arc reveals a new potential cost of ⟨0,0,0,0,1⟩ for q_7. Taking this new cost together with the current cost attribute for node q_7 gives us {⟨0,0,0,0,1⟩, ⟨1,0,0,0,0⟩}. Obviously, given no specification of the ranking, these costs are both contenders – they each encode a single violation of one constraint. However, when they are expressed as ERCs, ⟨0,0,0,0,1⟩ gives us ⟨L,e,e,e,W⟩ and ⟨1,0,0,0,0⟩ gives us ⟨W,e,e,e,L⟩. The former is one of the pre-specified ERCs and the latter is its antithesis. Thus the former is trivially consistent and the latter inconsistent, so ⟨1,0,0,0,0⟩ is not a viable cost for q_7 and $o[q_7]$ is not updated.

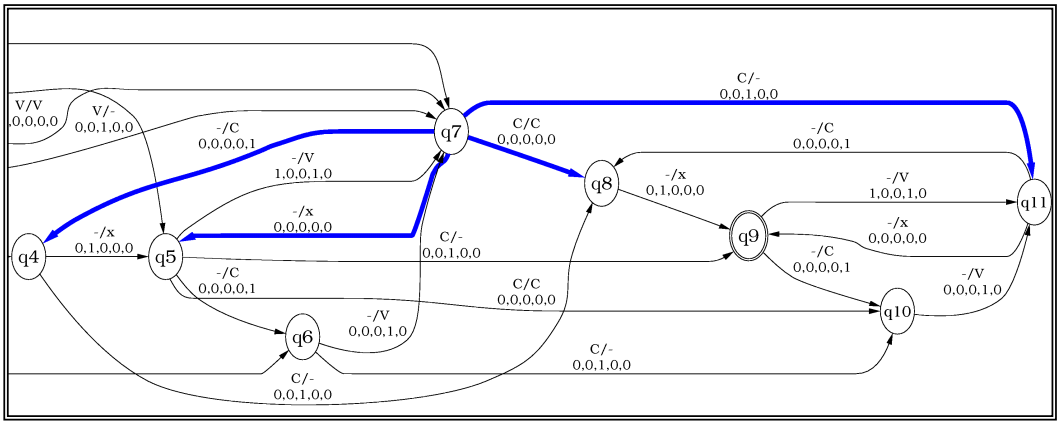
Checking the arc from q_1 to q_6 allows us to replace ∅ in the cost attribute for node q_6 with the estimate ⟨0,0,1,0,1⟩. The updated table of cost attributes is given in (142).

(142) Updated table:

3.	q_0	q_1	q_2	q_3	q_4	q_5
	⟨0,0,0,0,0⟩	⟨0,0,0,0,1⟩	⟨1,0,0,1,0⟩	∅	∅	⟨0,0,1,0,0⟩
	q_6	q_7	q_8	q_9	q_{10}	q_{11}
	⟨0,0,1,0,1⟩	⟨1,0,0,0,0⟩	∅	∅	∅	∅

Since H is not empty, a random node is removed from H and the arcs originating at it are checked. If q_7 is the node removed we have the four bold arcs in (143) to check.

(143) There are four arcs originating at q_7:



The arc from q_7 to q_5 reveals a new potential cost of ⟨1,0,0,0,0⟩ for node q_4 (that's the cost of the arc plus ⟨1,0,0,0,0⟩, the cost attribute for q_7). Taking this new cost together with the current cost attribute for node q_4 gives us the set {⟨1,0,0,0,0⟩, ⟨0,0,1,0,0⟩}. The result of feeding this set to the contenders function is illustrated in (144).

(144) *contenders*({⟨1,0,0,0,0⟩, ⟨0,0,1,0,0⟩},{⟨L,e,W,e,e⟩, ⟨L,e,e,e,W⟩})

i) checking ⟨1,0,0,0,0⟩:
erc(⟨1,0,0,0,0⟩, ⟨0,0,1,0,0⟩) = ⟨L,e,W,e,e⟩
consistent({⟨L,e,W,e,e⟩} ∪ {⟨L,e,W,e,e⟩, ⟨L,e,e,e,W⟩}) = true

ii) checking ⟨0,0,1,0,0⟩:
erc(⟨0,0,1,0,0⟩, ⟨1,0,0,0,0⟩) = ⟨W,e,L,e,e⟩
consistent({⟨W,e,L,e,e⟩} ∪ {⟨L,e,W,e,e⟩, ⟨L,e,e,e,W⟩}) = false

The *contenders* function returns only the cost vector ⟨1,0,0,0,0⟩ as a contender for q_5; this is a change from the previous value for $o[q_5]$ so the cost attribute for q_5 is updated.

The current cost attributes for nodes q_4, q_8 and q_{11} are all ∅, so the arcs terminating at them arcs reveal new values for the cost attributes for those nodes. The updated values are given in (145).

(145) Updated table:

4.	q_0	q_1	q_2	q_3	q_4	q_5
	⟨0,0,0,0,0⟩	⟨0,0,0,0,1⟩	⟨1,0,0,1,0⟩	∅	⟨1,0,0,0,1⟩	⟨1,0,0,0,0⟩
	q_6	q_7	q_8	q_9	q_{10}	q_{11}
	⟨0,0,1,0,1⟩	⟨1,0,0,0,0⟩	⟨1,0,0,0,0⟩	∅	∅	⟨1,0,1,0,0⟩

Eleven more passes through the while loop in **step 5** of the algorithm result in the removal of all of the nodes from *H* and gives us the finished cost table in (146).

(146) Finished table:

15.	q_0	q_1	q_2	q_3	q_4	q_5
	⟨0,0,0,0,0⟩	⟨0,0,0,0,1⟩	⟨1,0,0,1,0⟩	⟨1,0,0,1,1⟩	⟨1,0,0,0,1⟩	⟨1,0,0,0,0⟩
	q_6	q_7	q_8	q_9	q_{10}	q_{11}
	⟨1,0,0,0,1⟩	⟨1,0,0,0,0⟩	⟨1,0,0,0,0⟩	⟨1,0,0,1,0⟩ ⟨1,0,1,0,0⟩ ⟨1,1,0,0,0⟩	⟨1,0,0,0,0⟩	⟨1,0,1,0,0⟩ ⟨1,0,0,1,0⟩

Once the set *H* is empty **step 10** pools the costs at the final nodes and removes any costs that aren't contenders in this set. Since q_9 is the only final state, this check is redundant for this particular machine.

The output of the algorithm is {⟨1,0,0,1,0⟩, ⟨1,0,1,0,0⟩, ⟨1,1,0,0,0⟩}. Feeding the cost table along with the machine to the *ccand* function (presented in §4.4) produces the set of candidates. In (147) I give the three candidates that are produced by the modified

CONTENDERS algorithm for the input /vc/ when ⟨L,e,W,e,e⟩, and ⟨L,e,e,e,W⟩ are pre-specified as conditions on which candidates are to be generated.

(147) Three contenders:

	/vc/	ons	noc	max	depV	depC
a.	vc	1	1	0	0	0
b.	v	1	0	1	0	0
c.	v.cv	1	0	0	1	0

Using the new CONTENDERS algorithm it's possible to use the observation that /vv/ maps to [v.v] under some unknown grammar to reduce the set of predictions for a subsequent input like /vc/ from the eight contenders in (117) to just the three candidates In (147).

Running the CONTENDERS algorithm with some ERCs pre-specified can radically reduce the amount of work involved in finding the contenders because the pre-specified ERCs reduce the size of the set of viable contender-costs for each node. This reduces both the number of times that a node will be added into *H* and the number of vectors that have to be summed and kept track of.

5.9 Simulation

The redefined CONTENDERS algorithm provides a natural method for assessing the success of hypotheses about grammars in cases where the lexicon is fixed. That is, if we are dealing with a finite lexicon then the success of a hypothesis (set of ERCs) can be stated as the portion of the lexicon for which it reduces the number of viable contenders to exactly one. This measurement is a worst-case assessment that classifies a hypothesis

as correct for a given input form just in case every linearization of the constraints that is consistent with the hypothesized ERCs predicts the right output for that input. In other words, a hypothesis will only be rated as 100% correct once every linearization of the constraints that respects the ERCs in the hypothesis generates the right output for every input form in the lexicon.

With this tool I return now to the 62-form lexicon presented in §5.3. To assess the difficulty of learning languages in the class obtained from the interaction the CV^5 lexicon with the grammars defined by permutation of {**g**, ONSET, NOCODA, MAX, DEPV, DEPC}, I use the simulation described in (148). In step (ii) in the measurement, *cv* is defined as the number of forms in the lexicon for which CONTENDERS produces exactly one cost vector when given the set of ERCs that makes up the current hypothesis.

(148) **TRIAL(CON, *Lex*):**

1 Randomly linearize the constraints in CON to obtain ***R***.
2 $cv = |\{ in \mid in \in CV^5, |\text{CONTENDERS}(in, \boldsymbol{R}, \varnothing)| = 1 \}|$, output$\rightarrow(0, cv)$
3 Run TRIALOOP(***R***, CV^5, 0, ∅).

(149) **TRIALOOP(*R*, *Lex*, *n*, E_0)**

1 Draw a random input *in* from CV^5 and for *v* the cost vector of an optimal form in OPTIMIZE(*in*, ***R***), and for *V* the set of cost vectors obtained from CONTENDERS(*in*, ***R***, ∅), obtain $E_1 = \{ e \mid erc(v, v') , v' \in V\}$.
2 $cv = |\{ in \mid in \in CV^5, |\text{CONTENDERS}(in, \boldsymbol{R}, E_1 \cup E_0)| = 1\}|$, output$\rightarrow(n, cv)$
3 if $k = |CV^5|$ halt, else run TRIALOOP(*R*, CV^5, $n + 1$, $ERCs \cup E_0$).

The clause "output → (*n*, *cv*)" in **step 2** of (148) tells the algorithm to write out the number of lexical items covered after the n^{th} observation. In **step 2** of TRIAL there haven't yet been any observations so $n = 0$, with each subsequent observation *n* goes up by one.

For the basic CV syllable theory the value of *cv* at the first coverage check in **step 2** of TRIAL will always be 2. This is because there are two forms, /cv/ and /cvcv/, for which CONTENDERS produces exactly one prediction when fed no ERCs at all. In (150), I present two iterations of the TRIAL algorithm.

(150) **Trial 1** - random ranking: ⟨NOCODA, MAX, ONSET, DEPC, DEPV⟩

Observation 0: Lexicon coverage at 0 observations = 2

Observation 1: /cvv/ → CVxCVx

	/cvv/	NOC	MAX	ONS	DEPC	DEPV	ERCs
1	☞ cv.cv	0	0	0	1	0	∅
2	cv.v	0	0	1	0	0	⟨e,e,W,L,e⟩
3	cv	0	1	0	0	0	⟨e,W,e,L,e⟩

Total ERC set: E = {⟨e,e,W,L,e⟩, ⟨e,W,e,L,e⟩}

Lex coverage: 19 = |{ *in* | *in* ∈ CV^5, |CONTENDERS(*in*, ***R***, *E*)| = 1 }|

Observation 2: /cvv/ → CVxCVx

	/cvv/	NOC	MAX	ONS	DEPC	DEPV	ERCs
1	☞ cv.cv	0	0	0	1	0	∅
2	cv.v	0	0	1	0	0	⟨e,e,W,L,e⟩
3	cv	0	1	0	0	0	⟨e,W,e,L,e⟩

Total ERC set: E = {⟨e,e,W,L,e⟩, ⟨e,W,e,L,e⟩}

Lex coverage: 19 = |{ *in* | *in* ∈ CV^5, |CONTENDERS(*in*, ***R***, *E*)| = 1 }|

Observation 3: /cc/ → CVxCVx

	/cc/	NOC	MAX	ONS	DEPC	DEPV	ERCs
1	☞ cv.cv	0	0	0	0	2	∅
2	-	0	2	0	0	0	⟨e,W,e,e,L⟩
3	cvc	1	0	0	0	1	⟨W,e,e,e,L⟩

Total ERC set: E = {⟨e,e,W,L,e⟩, ⟨e,W,e,L,e⟩, ⟨e,W,e,e,L⟩, ⟨W,e,e,e,L⟩}

Lex coverage: 62 = |{ *in* | *in* ∈ CV^5, |CONTENDERS(*in*, ***R***, *E*)| = 1 }|

In trial 1 we see that after just three observations (actually only two because the second was identical to the first), the set of four ERCs collected is sufficient to make the right output prediction for every single form in the lexicon. Sometimes convergence on the correct grammar occurs after only one observation. This is illustrated in (151).

(151) **Trial 2** - random ranking: ⟨DEPC, MAX, NOCODA, DEPV, ONSET⟩

Observation 0: Lexicon coverage at 0 observations: 2

Observation 1: /cvv/ → CVxCVx

	/vc/	DEPC	MAX	NOC	DEPV	ONS	ERCs
1 ☞	v.cv	0	0	0	1	1	∅
2	vc	0	0	1	0	1	⟨e,e,W,L,e⟩
3	v	0	1	0	0	1	⟨e,W,e,L,e⟩
4	cv	0	1	0	1	0	⟨e,W,e,e,L⟩
5	-	0	2	0	0	0	⟨e,W,e,L,L⟩
6	cv.cv	1	0	0	1	0	⟨W,e,e,e,L⟩
7	cvc	1	0	1	0	0	⟨W,e,W,L,L⟩
8	cv	1	1	0	0	0	⟨W,W,e,L,L⟩

Total ERC set: E = {⟨e,e,W,L,e⟩, ⟨e,W,e,L,e⟩, ⟨e,W,e,e,L⟩, ⟨e,W,e,L,L⟩, ⟨W,e,e,e,L⟩, ⟨W,e,W,L,L⟩, ⟨W,W,e,L,L⟩}

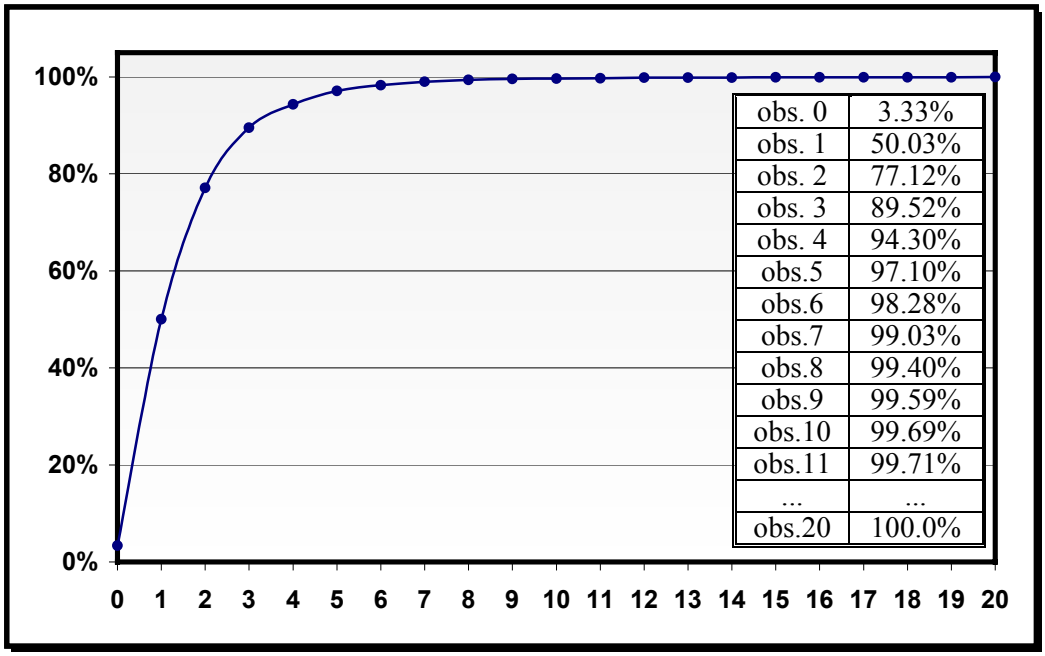
Lex coverage: 64 = |{ *in* | *in* ∈ *Lex*, |CONTENDERS(*in*, ***R***, *E*)| = 1 }|

The i/o pair observed in (151) constitutes what Gibson and Wexler (1994) call a global trigger. That is, this single observed datum reveals enough information about the language for the learner to distinguish it from all others in the space of possible languages. In this case the set of possible languages is just the set of input/output relations over CV^5 defined by permutation of the constraints {DEPC, MAX, NOCODA, DEPV, ONSET}.

Though it's often the case that one or two observations reveal enough information to make the right predictions thereafter for the entire lexicon (this happened in almost half of the trials I ran), in some trials fate is not so kind. For instance, observing the outcome for inputs /cv/ or /cvcv/ reveals nothing about the grammar, and observing the outcome for inputs that have been previously seen reveals no new information.

To get a general idea of how easily this simple class of languages can be learned I ran one thousand iterations of the TRIAL algorithm, recording for each iteration the portion of lexicon coverage after each of the observations. In (152) I graph the lexicon coverage after *n* observations averaged over the thousand trials.

(152) Average lexicon coverage after ***n*** observations (1,000 trials)



obs. 0	3.33%
obs. 1	50.03%
obs. 2	77.12%
obs. 3	89.52%
obs. 4	94.30%
obs.5	97.10%
obs.6	98.28%
obs.7	99.03%
obs.8	99.40%
obs.9	99.59%
obs.10	99.69%
obs.11	99.71%
...	...
obs.20	100.0%

To assess the advantage gained by access to the entire set of contenders I devised a simple error-driven version of the TRIAL algorithm that I'll call ED-TRIAL. In (153) I

give ED-TRIAL in detail. As with the version of the algorithm in (148), a random ranking ***R*** is selected and then input forms are randomly drawn from the lexicon and mapped to optimal outputs under ***R***. If all linearizations of the constraints that are consistent with the current hypothesis (set of ERCs) generated by the observed i/o mapping then no errors are possible and thus no information is gained. On the other hand, if any linearization that is consistent with the current hypothesis generates a wrong prediction then a linearization is selected at random and if it makes an erroneous prediction then the failed candidate allows one ERC to be deduced and added to the hypothesis.

(153) **ED-TRIAL (CON, *Lex*):**

1 Randomly linearize the constraints in CON to obtain ***R***.

2 $cv = |\{ in \mid in \in Lex, |\text{CONTENDERS}(in, \boldsymbol{R}, \varnothing)| = 1 \}|$, output $(0, cv)$

3 Run ED-TRIALOOP(***R***, *Lex*, 0, ∅).

(154) **ED-TRIALOOP(*R*, *Lex*, *n*, E_0)**

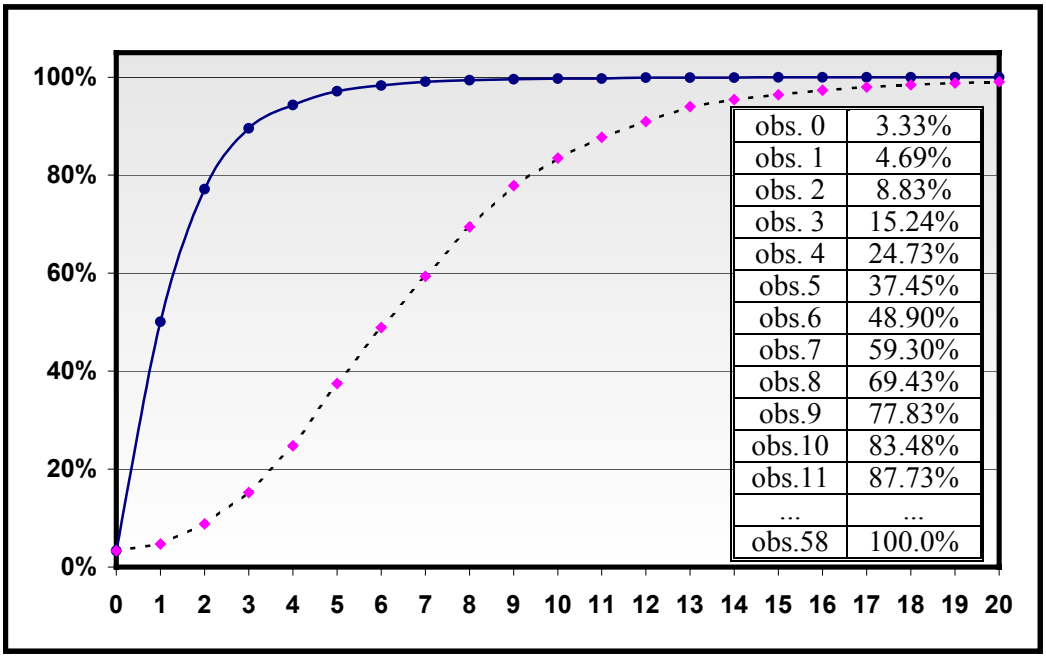
1 Draw a random input *in* from *Lex* and for CONTENDERS(*in*, ***R***, E_0) = *C* if $|C| > 1$ then pick a random $w \in C$ and for *w* the cost vector of an optimal form in OPTIMIZE(*in*, ***R***), obtain $E = | erc(w, v)$.

2 $cv = |\{ in \mid in \in Lex, |\text{CONTENDERS}(in, \boldsymbol{R}, E \cup E_0)| = 1 \}|$, output (n, cv)

3 if $k = |Lex|$ terminate, else run ED-TRIALOOP(*R*, *Lex*, $n + 1$, $E \cup E_0$).

In (155) I graph the lexicon coverage after *n* observations averaged over one thousand trials with the constraint set {**g**, ONSET, NOCODA, MAX, DEPV, DEPC}.

(155) Average lexicon coverage after ***n*** observations (1,000 trials)



– *The dashed line shows the* ED *trials, the solid line is repeated from* (152) *above.*

5.9 The (slightly) extended CV syllable theory

In this section I'll present a class of grammars with twice the number of constraints used in the basic CV syllable theory grammars.[15] Surprisingly, this increase in the number of constraints won't greatly change the rate at which languages can be learned with the simple strategy of gathering ERCs from observed i/o-pairs. The constraint set that I'll use for this exploration is given in (156) and (157). In (156) I start with four constraints that will be held undominated at the top of the hierarchy in every ranking considered and then in (157) I present the constraints whose rankings will be allowed to vary.

[15] Actually these new grammars will have 14 constraints, but 4 of them will be undominated in all rankings.

(156) Constraints held undominated in all rankings:

a) *CCC: sequences of three consonants are forbidden
– the sequence CCC gets one violation. This constraint has 3 states.

b) *VVV: sequences of three vowels are forbidden
– the sequence VVV gets one violation. This constraint has 3 states.

c) *HNUC: every syllable must have a nucleus (must contain at least one V)
– the sequence xC*x gets one violation. This constraint has 2 states.
(Prince and Smolensky 1993)

d) OCP: the sequence $VC^{+}V$ is not permitted within a syllable
– the sequence $VC^{+}V$ gets one violation. This constraint has 3 states.
(cf. Leben 1973, McCarthy 1979, 1986)

Keeping the four constraints in (156) undominated will rein in the set of candidates considered in each derivation to consist of (possibly empty) sequences of syllables each of which consists of 0-2 consonants followed by 1-2 vowels followed by 0-2 consonants. As previously, I'll assume that all surface segments are syllabified and that inputs are not specified for syllabification.

In (157) I give ten constraints whose ranking will be allowed to vary between trials. The first four are from the basic CV syllable theory and the last six round out what I'll call the extended CV syllable theory.

(157) Ranked constraints:

Basic CV syllable theory

a) ONSET: penalizes vowels word-initially and immediately following an 'x'
b) NOCODA: penalizes syllable boundaries occurring immediately after a 'C'
c) DEPC: penalizes consonant epenthesis
d) DEPV: penalizes vowel epenthesis

Extended CV syllable theory

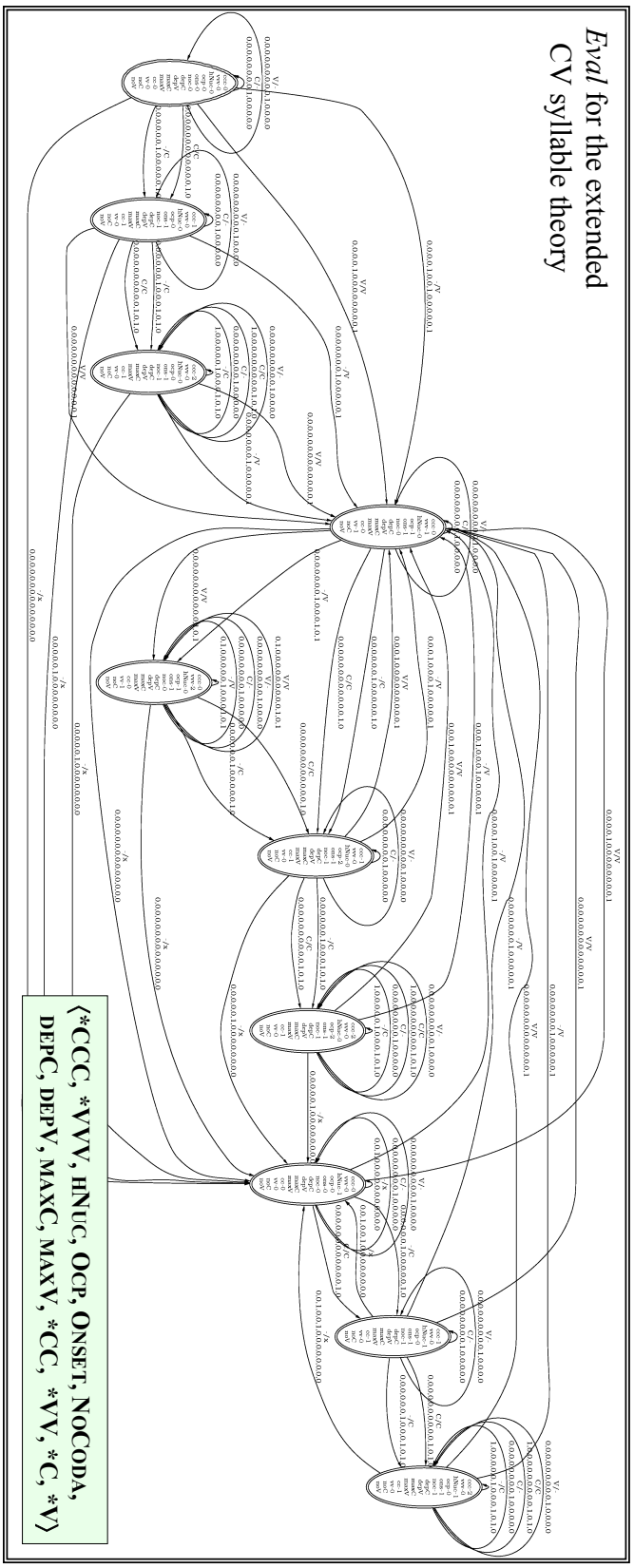
e) MAXC: penalizes deletion of consonants
f) MAXV: penalizes deletion of vowels
g) *CC: penalizes consonant-consonant sequences
h) *VV: penalizes vowel-vowel sequences
i) *C: penalizes each occurrence of a consonant
j) *V: penalizes each occurrence of a vowel

Constraints a through d were presented in §6.1. Recall that ONSET and NOCODA each have two states. The new constraints in e through j aren't too different from the constraints that we've seen thus far. The two versions of MAX in e and f distinguish consonant and vowel deletion respectively. Like all faithfulness constraints used here, each has one state. Constraints *CC and *VV penalizes pairs of adjacent consonants and vowels and each one has two states. At this point our alphabet has only one C and one V, so there is no need to distinguish good clusters and diphthongs from bad ones. Obviously, these constraints can be refined for larger alphabets. In i and j we have a pair of simple *STRUCTURE-style markedness constraints that penalize every occurrence of a consonant or vowel respectively.

Intersecting the 14 constraints in (156) and (157) could result in a machine with: 3 × 3 × 2 × 3 × 2 × 2 × 1 × 1 × 1 × 1 × 2 × 2 × 1 × 1 = 846 states. However, because many of the environments specified by the constraints overlap, there are only ten states in *Eval* for the extended CV syllable theory.

Eval for the extended CV syllable theory, is given in (158) at the right. Though a bit hard to read, this graph shows that the grammar doesn't get that much more complex when the new constraints are added.

By varying the ranking of the ten constraints in (157) we obtain 10! = 3,628,800 permutations. The pertinent question is, then, how many different i/o-relations are defined by permutation of these ten constraints. That is, how big is the typology?



In the lexicon CV^5, the input /vvccc/ has 39 contenders under the ten constraints in (157). This establishes at least 39 different i/o-relations defined by these constraints on the lexicon CV^5. There could easily be more than 39 languages, for it is entirely possible that no single input in the lexicon will show every possible distinction. In fact, the string /vccvvvccc/ has 78 contenders under the constraints in (157).

I'll come back to the larger question of how many different i/o-relations are defined by a set of constraints over the range of all possible inputs from Σ^* in chapter seven. For now I'll continue to focus on the finite lexicon CV^5 and ask how quickly the TRIAL and ED-TRIAL algorithms converge on the languages that are defined by the application of different permutations of the constraints in (157) to the input forms in CV^5.

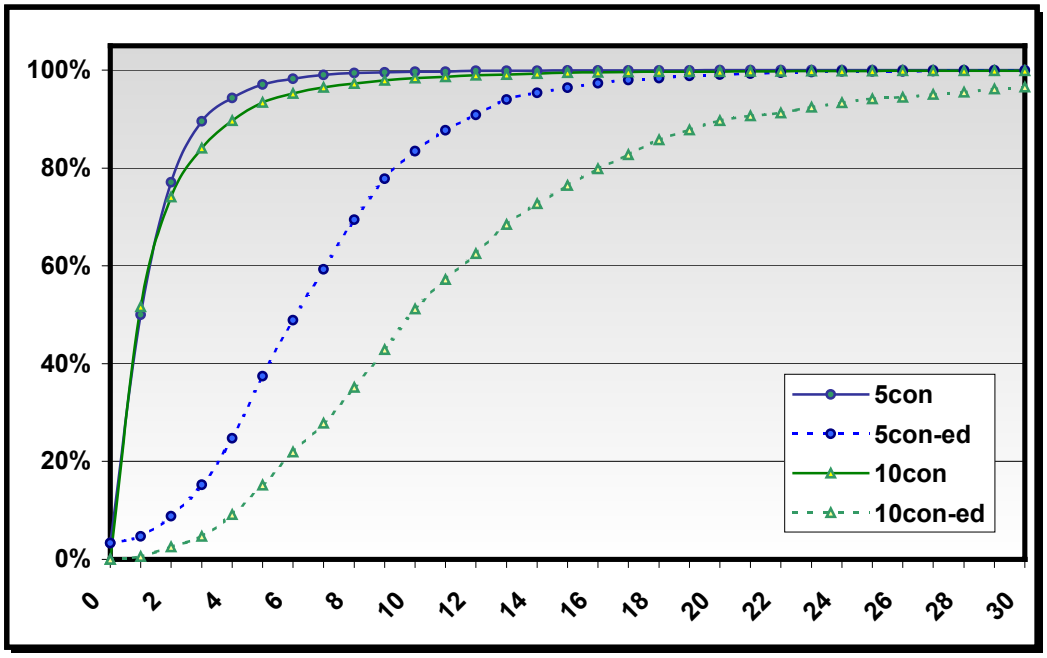
As with the 5-constraint scenario, I ran one thousand trials with each algorithm and calculated the average lexicon-coverage after each observation in each trial. In (159) I give a table with averaged results over the first 30 observations for both the contender-driven and error-driven versions of TRIAL in both the five and ten constraint scenarios.

(159) Average results across types of trials:

obs.	5con	5con-ed	10con	10con-ed
0	3.33%	3.33%	0%	0.00%
1	50.03%	4.69%	51.64%	0.61%
2	77.12%	8.83%	74.12%	2.54%
3	89.52%	15.24%	84.08%	4.72%
4	94.30%	24.73%	89.68%	9.16%
5	97.10%	37.45%	93.39%	15.21%
6	98.28%	48.90%	95.23%	21.92%
7	99.03%	59.30%	96.51%	27.79%
8	99.40%	69.43%	97.30%	35.14%
9	99.59%	77.83%	97.91%	42.85%
10	99.69%	83.48%	98.38%	51.17%
11	99.71%	87.73%	98.63%	57.19%
12	99.89%	90.89%	98.97%	62.54%
13	99.89%	93.99%	99.13%	68.48%
14	99.89%	95.40%	99.33%	72.72%
15	99.95%	96.40%	99.48%	76.45%
16	99.95%	97.31%	99.59%	79.84%
17	99.95%	97.98%	99.63%	82.75%
18	99.95%	98.40%	99.69%	85.72%
19	99.95%	98.82%	99.71%	87.81%
20	100%	99.09%	99.72%	89.68%
21	100%	99.32%	99.75%	90.69%
22	100%	99.50%	99.79%	91.28%
23	100%	99.62%	99.83%	92.43%
24	100%	99.75%	99.83%	93.36%
25	100%	99.78%	99.84%	94.13%
26	100%	99.78%	99.87%	94.45%
27	100%	99.84%	99.87%	95.06%
28	100%	99.89%	99.89%	95.49%
29	100%	99.89%	99.89%	96.18%
30	100%	99.92%	99.90%	96.56%

Unlike the 5-constraint trials, the 10-constraint trials start at 0% coverage. This is because, with the addition of basic markedness constraints like *V and *C, it's no longer the case that for inputs like /cv/ and /cvcv/ the identity map is optimal under all constraint rankings. In (160) I present a graph of the results for all four sets of trials.

(160) Average lexicon coverage after ***n*** observations (1,000 trials)



There are several interesting things illustrated in (160). The most surprising is the fact that doubling the number of constraints doesn't have much of an effect on the rate of convergence for contender-driven learning even though the number of possible rankings goes from 120 to 3.6 million and the number of possible i/o-relations goes from 12 to 39. The error-driven algorithm, on the other hand, does significantly worse when the number of constraints is doubled and accordingly the disparity between the rates of convergence for the contender-driven and error-driven trials is even greater in the 10-constraint trials.

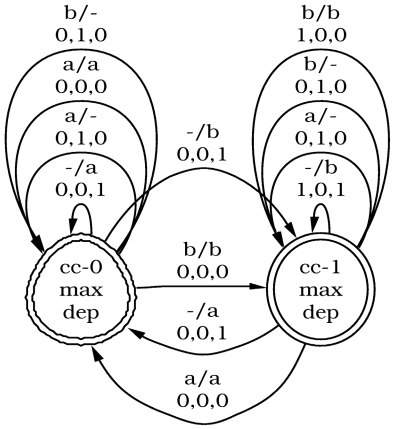
As an initial examination of learning with contenders, these results are promising. The fact that, as more constraints are added, the size of *Eval* does not grow explosively and the rate of learning doesn't drop too quickly suggests that the difficulties in learning real-world OT grammars may be far less than the imaginable worst case scenario.

6 OT in linear time

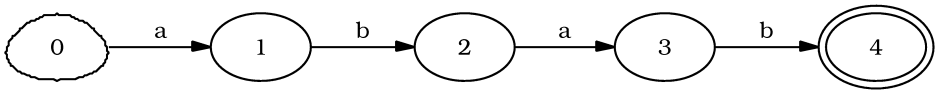
In chapter three I presented a method for finding optimal parses in the intersection of an input string with *Eval*. In this chapter I'll show how recurring structures across the evaluation of various inputs can be generalized and how suboptimal chunks of these structures can be excised ahead of time to make later optimization tasks more efficient.

Notice that when *Eval* in (161) is intersected with the acceptor for input /abab/ in (162) to produce (163), the state-structure of *Eval* is iterated over each segment of the input string.

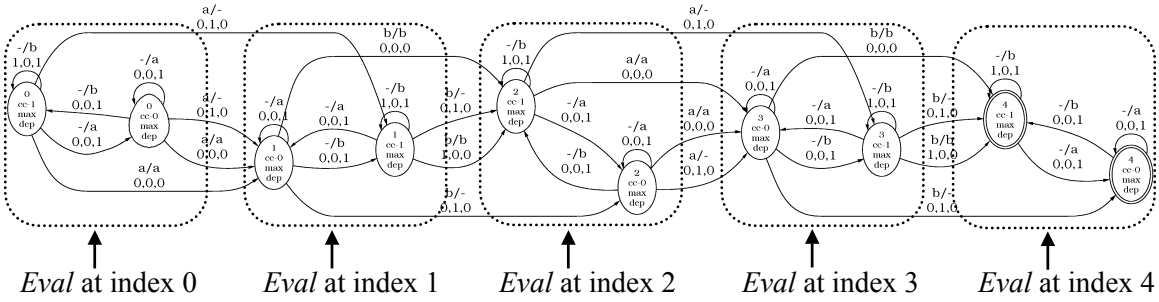
(161) ***Eval***:
⟨*CC, MAX, DEP⟩⊠



(162) *A*(abab):



(163) ⟨*A*(abab),*CC, MAX, DEP⟩⊠



When examining just one of the boxed iterations of *Eval* in (163) it's not possible to know which of the arcs are in a globally optimal path through the machine. This is so because optimality is a global property of an entire path relative to an entire machine. On the other hand, it is possible to detect many arcs that could *never* participate in an optimal path. This is because suboptimality can be a strictly local property (cf. the optimal subpath lemma in §3.3).

A path from point A to point B reading in input string S is locally optimal just in case there is no cheaper way to get from point A to point B reading in S. Conversely a path is locally suboptimal just in case there is a cheaper way to get from its origin to its terminus reading in the same input string that it does. It follows straightforwardly from the optimal subpath lemma that all subpaths of optimal paths are locally optimal paths.

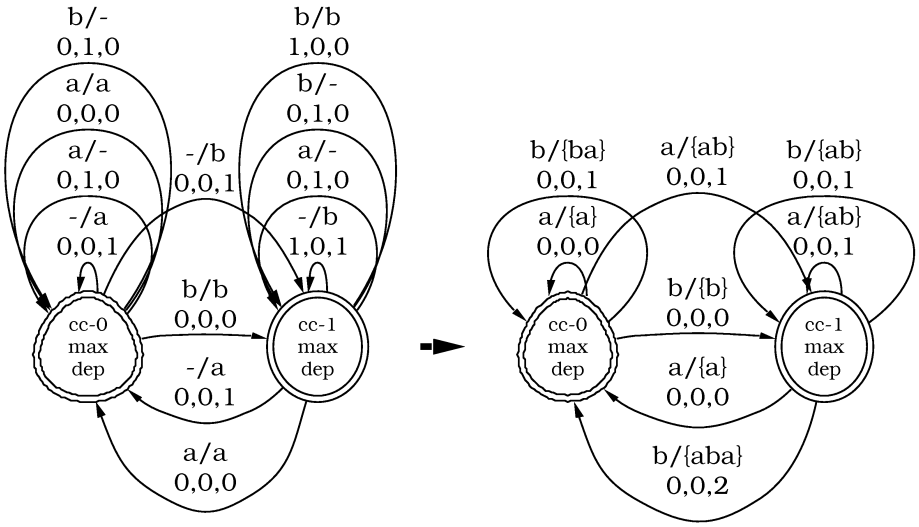
In this chapter I will present an algorithm that capitalizes on the fact that the structure of *Eval* is repeated many times throughout the assessment of various inputs, by doing optimization on *Eval* itself. I'll show that by removing locally suboptimal paths in *Eval* we obtain a "preoptimized" version of *Eval* that has many desirable properties. The most important of these properties is that the amount of work required to optimize the intersection of an input string with preoptimized *Eval* is a linear function of the length of the input string. In other words, the machine can be optimized in linear time.

6.1 Overview of preoptimization

Preoptimization takes *Eval* and returns a machine *E* with the same set of nodes, the same alphabet, the same start state, and the same final states. The only difference between

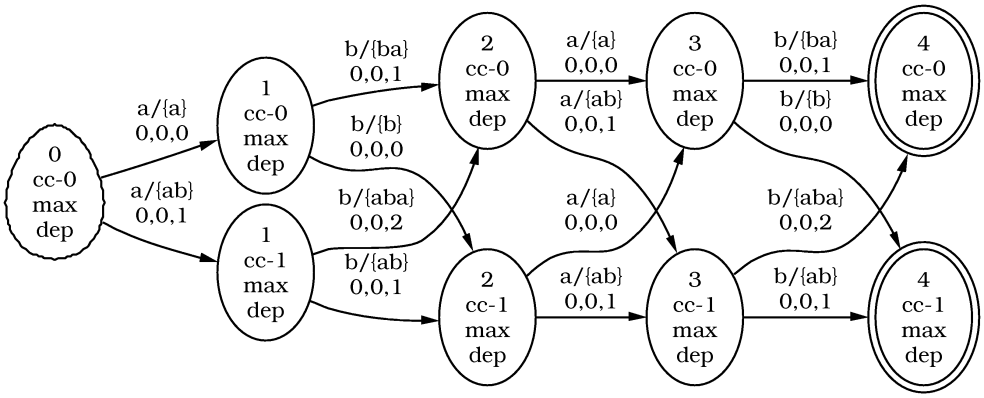
Eval and *E* is that for every pair of nodes *q* and *r*, if *w* is the cost of an optimal from *q* to *r* in *Eval* that accepts exactly one input symbol $i \in \Sigma$, then there is an arc (*q*, *i*, *Out, w*, *r*) in *E* where *Out* is the set of outputs written by the paths from *q* to *r* in *M* at cost *w* reading the input *i*. In this new machine *E*, each arc corresponds to a set of optimal paths in *Eval* that accept exactly one segment from the input string. In (164) I illustrate the effects of preoptimization on *Eval*.

(164) Preoptimization:



Consider in (165) the intersection of preoptimized *Eval* with the input /abab/ and contrast this with the evaluation of the same input without preoptimization, presented above in (163).

(165) ⟨*A*(abab), PREOPT(⟨*CC, MAX, DEP⟩)⟩$^{\boxtimes}$



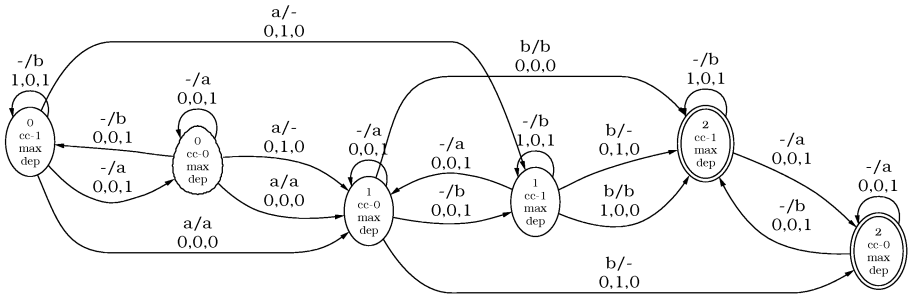
The machine in (165) has many interesting properties. The most salient of these, when compared with (163), is that (165) has no cycles (loops). Computationally, the fact that (165) is a directed acyclic graph (DAG), means that the amount of work required to optimize it is a linear function of its size. Empirically, the lack of cycles means that (165) generates only a finite set of candidates

Now that I've shown where we are going, the remainder of the chapter will be spent showing how to get there and proving that the result is sound in the sense that no member of the infinite set of candidates eliminated by the preoptimization algorithm could ever be optimal. I will close with a discussion of the computational complexity of the preoptimization algorithm.

6.2 Isomorphism across evaluations

Across the evaluation of various inputs, any given substring always contributes the same structure to the various machines in which it occurs. For example, consider in (166) the machine that results from intersecting *Eval* = ⟨*CC, MAX, DEP⟩$^{\boxtimes}$ with *A*(ab).

(166) ⟨*A*(ab),*CC, MAX, DEP⟩$^{\boxtimes}$



Notice that (166) is identical to the first half of ⟨*A*(abab),*CC, MAX, DEP⟩$^{\boxtimes}$ given in (163). In fact, modulo the differences in the indices at the tops of the nodes, (166) is also the same as the second half of (163); that is, they are isomorphic. Two graphs that contain the same number of nodes connected to one another in the same ways (with arcs with the same labels) are isomorphic. In the machines we are considering here, the only difference between the graphs is in the indices that make up the first part of the node-names.

The two halves of (163) are isomorphic to each other and will be isomorphic to a subgraph of ⟨A(*in*),*CC, MAX, DEP⟩$^{\boxtimes}$ for any input string *in* that has 'ab' as a substring. This can be put formally with the following lemma.

(167) ***Eval* isomorphism lemma:**

For $M_1 = \langle A(\langle r_1, ..., r_t\rangle), E\rangle^{\boxtimes}$ and $M_2 = \langle A(\langle s_1, ..., s_u\rangle), E\rangle^{\boxtimes}$ if $\langle r_i, ..., r_m\rangle = \langle s_j, ..., s_n\rangle$ then the subgraph of M_1 from index i to index m is the same as the subgraph of M_2 from index j to index n modulo point-wise substitution of the indices.

proof: From the definition of linear acceptors we know that $A(\langle r_i, ..., r_m\rangle) \equiv A(\langle s_j, ..., s_n\rangle)$. They both consist of a string of arcs that accepts the same string of input segments

but differ in that the first node of the former is named i (r_i is the i^{th} segment of r), and the first node of the latter is named j (s_j is the j^{th} segment of s).

Because M-intersection is not sensitive to the names of the nodes but only to the labeling of the arcs, and both $A(r)$ and $A(s)$ are intersected with E, the portion of $\langle A(r), E\rangle^{\otimes}$ that accepts $\langle r_i, ..., r_m\rangle$ will contain the same arcs as the portion of $\langle A(s), E\rangle^{\otimes}$ that accepts $\langle s_j, ..., s_n\rangle$ modulo the differences in the indices on the nodes. Where the former has nodes numbered $i-m$ the latter has nodes numbered $j-n$. ■

Given that the same structures will arise over and over again in the evaluation of various input strings, we can save a lot of work by modifying *Eval* itself so that chunks of parses that are locally suboptimal aren't generated in the first place. I'll call this process "preoptimization" of *Eval*. Preoptimization could, in principle, be done for strings of any length, but for our current purposes it will be most useful to preoptimize for individual segments. In the next section I will present the preoptimization algorithm and step through it in detail.

6.3 The preoptimization algorithm

In preoptimization, optimal paths accepting individual input segments will be fused into single arcs. To facilitate this process I introduce in (168) a function *out*(M) that yields the set of output strings generated by a machine.

(168) $out(M) = \{o_1...o_n \mid \langle(q_1, i_1, o_1, w_1, r_1), ..., (q_n, i_n, o_n, w_n, r_n)\rangle$ is a path through $M\}$

When the machines we are dealing with are optimized *out*(*M*) will always yield a finite set of strings. This is so because optimization will always remove epenthetic cycles because they incur extraneous violations of DEP. This observation can be captured with the theorem in (169).

(169) **Theorem**: optimal parse acyclicity

If epenthetic material (any arc with the empty string as input) is penalized in *Eval* then optimal paths in the intersection of *Eval* with an input never contain cycles.

proof: Suppose p is an optimal path through $M = \langle A(i), Eval \rangle^{\rtimes}$ and, for a contradiction, that p contains a cycle c_p. Node names in M have two parts, first a numeric index contributed by $A(i)$ and second a position in *Eval* contributed by *Eval*.

Because only arcs that don't accept input segments (epenthesis) don't advance one state in $A(i)$, c_p must consist entirely of epenthetic arcs. Furthermore, by the assumption that epenthesis is penalized in *Eval*, c_p must have a non-zero cost.

Considering the path p' that's identical to p except that it lacks the cycle c_p it is the case that p' must be more harmonic than p because the cycle is not free. This contradicts the assumption that p was an optimal path thereby demonstrating that if all epenthetic material is penalized then optimal paths must be acyclic. ■

Because the constraint DEP penalizing epenthetic material is universally present under standard OT assumptions and because DEP will be used in all grammars considered here, the set of output strings produced by the optimized machines can always be given as finite sets. If, for some reason, it were desirable to consider grammars in which material

could be epenthesized for free, then the output of the function *out*(*M*) could just as easily be represented with a finite state machine defining a set of strings.

A theorem similar to the one presented in (169) can be constructed based on the constraints of the *STRUC family. Put simply, whenever it's the case that material cannot be inserted for free it will follow that optimal paths are acyclic. This is true regardless of the ranking of the constraints that penalize such additions. In terms of the algorithms presented here, the acyclicity theorem yields the following corollary for the OPTIMIZE algorithm.

(170) **corollary 1** of optimal parse acyclicity:
OPTIMIZE($\langle A(in), Eval\rangle^{\boxtimes}$) yields acyclic machines.

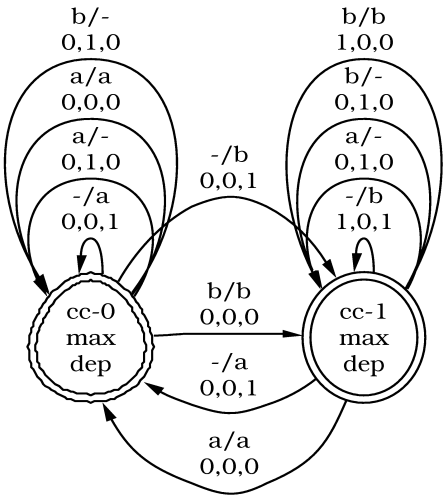
The preoptimization algorithm PREOPT is presented in (171) in pseudo-code with comments. Basically, PREOPT takes a machine, *Eval*, and returns a new machine identical to the original except for the fact that all of *Eval*'s arcs have been replaced with new arcs corresponding to optimal paths between pairs of nodes that accept single segments.

(171) PREOPT($\langle C_1, ..., C_k\rangle$) = $(Q, \Sigma, \delta_P, q_0, F)$

0	$\langle C_1, ..., C_k\rangle^{\boxtimes} = (Q, \Sigma, \delta, q_0, F)$	-Intersect the constraints.
1	$\delta_P \leftarrow \varnothing$	- Initialize δ_P, the preoptimized arc set, to null.
2	**for each** $\langle i, q, r\rangle \in \Sigma \times Q \times Q$	- For each ⟨segment, node, node⟩ trio
3	**do** $(Q', \Sigma, \delta', s, F') \leftarrow \langle A(i), (Q, \Sigma, \delta, q_0, F)\rangle^{\boxtimes}$	intersect the acceptor for that segment with *Eval*, then
4	**do** $M \leftarrow$ OPTIMIZE$(Q', \Sigma, \delta', (0, q), \{(1, r)\})$	optimize, but use $(0, q)$ and $(1, r)$ as the start and final states, and
5	**do** $\delta_P \leftarrow \{\delta_P \cup (q, i, out(M), w, r)\}$ where $w = c(p)$ for p, a path through M.	build an arc from q to r reading i and writing *out*(M) at a cost of w.

In (172) through (180) I'll use *Eval* = *CC ≫ MAX ≫ DEP to illustrate the action of the PREOPT algorithm.

(172) *Eval* = ⟨*CC, MAX, DEP⟩⊠

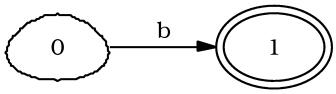


Eval = (*Q*, {a,b}, *δ*, (cc-0, max, dep), *F*),

Q = *F* = {(cc-0, max, dep), (cc-1, max, dep)},

δ = {((cc-0, max, dep), -, a, ⟨0,0,1⟩, (cc-0, max, dep)),
((cc-0, max, dep), a, -, ⟨0,1,0⟩, (cc-0, max, dep)),
((cc-0, max, dep), a, a, ⟨0,0,0⟩, (cc-0, max, dep)),
((cc-0, max, dep), b, -, ⟨0,1,0⟩, (cc-0, max, dep)),
((cc-0, max, dep), -, b, ⟨0,0,1⟩, (cc-1, max, dep)),
((cc-0, max, dep), b, b, ⟨0,0,0⟩, (cc-1, max, dep)),
((cc-1, max, dep), -, b, ⟨1,0,1⟩, (cc-1, max, dep)),
((cc-1, max, dep), a, -, ⟨0,1,0⟩, (cc-1, max, dep)),
((cc-1, max, dep), b, -, ⟨0,1,0⟩, (cc-1, max, dep)),
((cc-1, max, dep), b, b, ⟨1,0,0⟩, (cc-1, max, dep)),
((cc-1, max, dep), -, a, ⟨0,0,1⟩, (cc-0, max, dep)),
((cc-1, max, dep), a, a, ⟨0,0,0⟩, (cc-0, max, dep))}

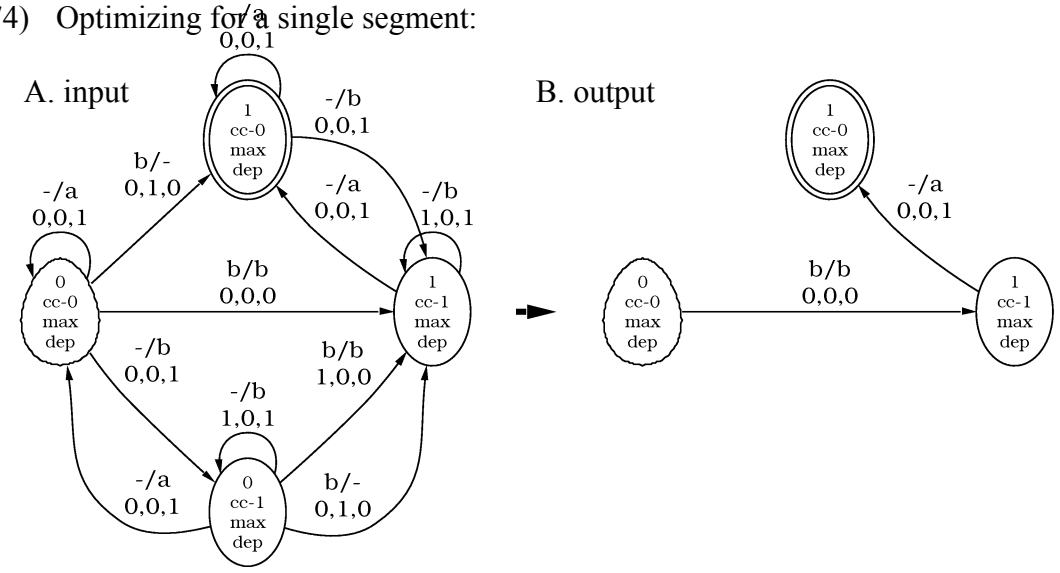
Paths that accept exactly one input segment can be assessed by intersecting the acceptor for that segment with *Eval*. For instance, the acceptor for input /b/ is given in (173).

(173) *A*(b): 

Instead of intersecting (173) with *Eval* as given in (172), which would allow us to determine the optimal routes from the start in (172) to the final states in (172) accepting input /b/, let us chose a pair of nodes (cc-0, max, dep) and (cc-0, max dep) in (172) to act as the start and final states respectively. That is, we'll be finding optimal paths from the left-hand node of (172) to itself. Along with the selection of /b/ as the input symbol this is **step 2** of the algorithm. In **step 3** *A*(b) is intersected with a machine identical to (172) modulo the specification of the start and final states that have been selected in step 2.

In order to find the optimal paths from state (cc-0, max, dep) to (cc-0, max, dep) reading the input symbol /b/ we'll assume that (cc-0, max, dep) is the start state of (172) and also its final state. With this specification (172) is as in **Error! Reference source not found.** – this is the set-up for **step 4** of the algorithm. Intersecting the acceptor for /b/ in (173) with **Error! Reference source not found.** modulo the specification of (cc-0, max, dep) as both the start and final states produces (174). Running OPTIMIZE on the machine in (174) produces (174)-B.

(174) Optimizing for a single segment:

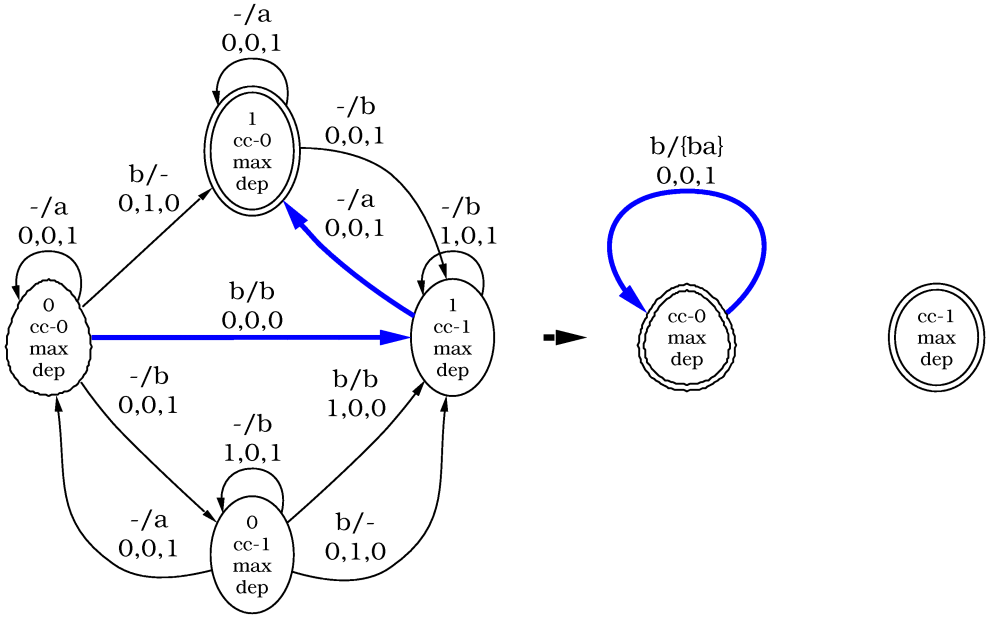


The machine presented in (174)-B. has just one path *p* that writes the output string 'ba' at a cost of ⟨0,0,1⟩. Thus, referring to the machine in (174)-B. as *M*, it's the case that *out*(*M*) = {ba} and *c*(*p*) = ⟨0,0,1⟩. With this information we are ready to construct an arc for the preoptimized machine (**step 5**). The arc is given in (175).

(175) *a* = ((cc-0, max, dep), b, {ba}, ⟨0,0,1⟩, (cc-0, max, dep))

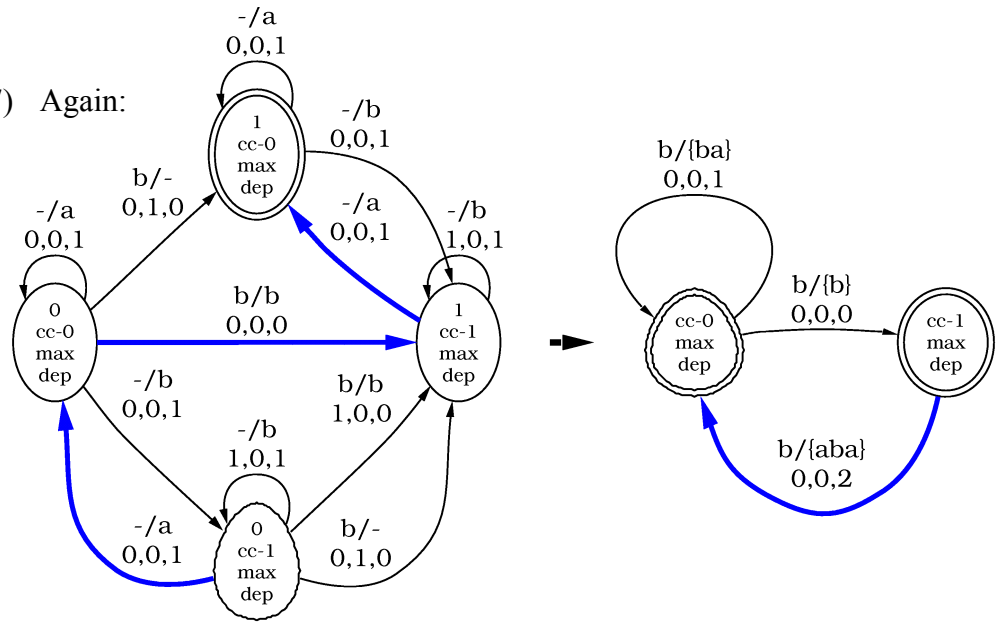
In (176), on the left I outline the arcs left intact by OPTIMIZE in bold, and on the right I show the arc that these arcs contribute to the preoptimized version of *Eval*.

(176) Optimizing for a single segment:



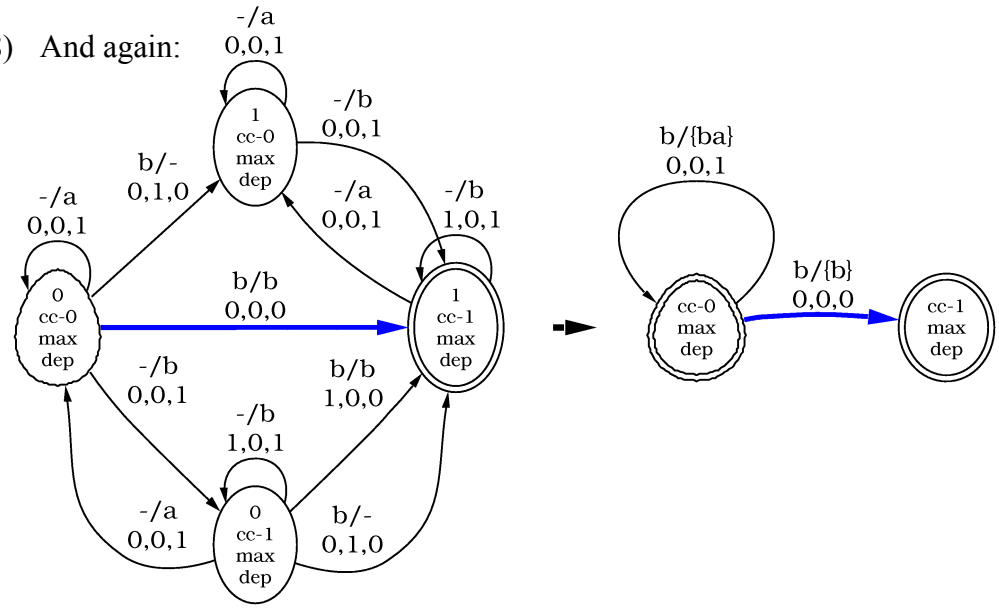
Sticking with the input /b/, another pair of nodes is chosen to act as the start and the final states and the procedure is repeated – the second iteration of **step 2**. This is illustrated in (177) with (cc-0, max, dep) and (cc-1, max, dep) serving as the start and final states respectively.

(177) Again:



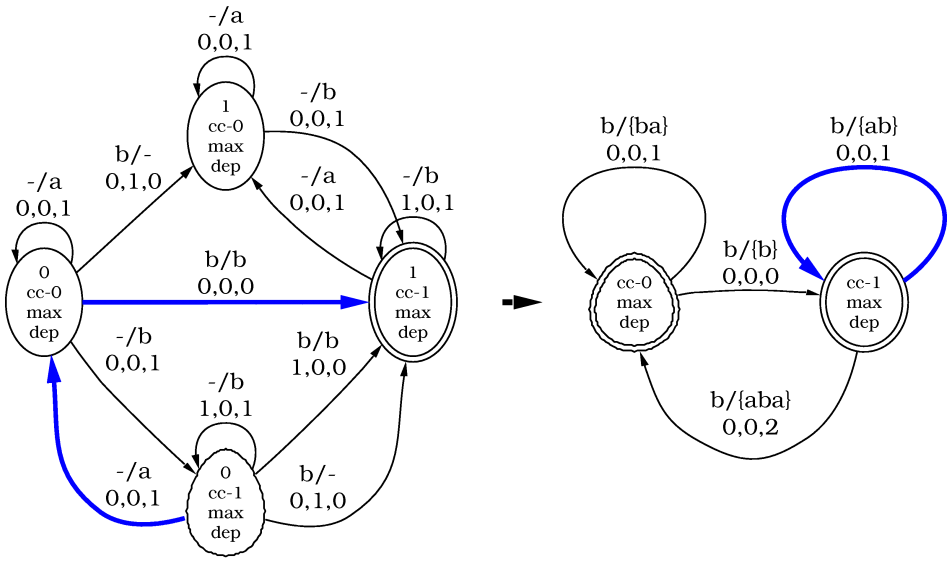
Next (cc-1, max, dep) and (cc-0, max, dep) are taken to serve the start and final states respectively – the third iteration of **step 2** of the algorithm. This is shown in (178).

(178) And again:



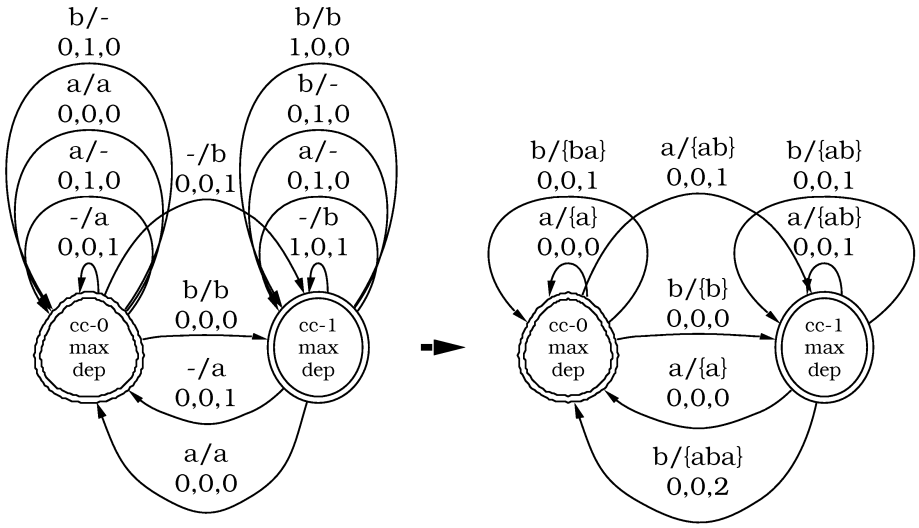
Finishing off the ⟨segment, node, node⟩ triples beginning with the input segment /b/, we take states (cc-1, max, dep) and (cc-1, max, dep) to be the start and final states respectively – the fourth iteration of **step 2** of the algorithm. This is shown in (179).

(179)



At this point the arcs with /b/ as input are finished and four more iterations of **step 2** in the algorithm are needed to cover the ⟨segment, node, node⟩ triples that begin with the input /a/. These last four steps fill out the remaining arcs in preoptimized *Eval* producing the machine on the right in (180), repeated from (164) above.

(180)



The number of arcs in preoptimized *Eval* is determined by the size of the set of symbols that make up Σ and whether or not every state in *Eval* can be reached from every

other state by reading every input symbol. If the constraints are restricted so that only markedness constraints have multiple states (to distinguish different surface environments) then it will generally be the case that every state can be reached from every other state by virtue of some sequence of epenthetic arcs. For *Eval* in which every state can be reached from every other state reading any given input symbol, if e is the number of nodes in *Eval* and s is the number of symbols in the alphabet, there will be $e \times s$ arcs originating at each node. With e nodes in total there will be se^2 arcs.

6.4 Correctness of PREOPT

To show that the preoptimization algorithm is correct I'll show that for any input *in*, if the string *out* is an output candidate from an optimal path in $\langle A(in), E\rangle^{\bowtie}$ then there is an optimal path in $\langle A(in), \text{PREOPT}(E)\rangle^{\bowtie}$ that also outputs the candidate *out*.

To do this I will show first that the language of PREOPT(*Eval*) is a subset of the language of *Eval*. Then I'll show that every candidate generated by $\langle A(in), E\rangle^{\bowtie}$ that is not generated by $\langle A(in), \text{PREOPT}(E)\rangle^{\bowtie}$ is guaranteed to be suboptimal. In (181) I repeat the definition of the language of a machine as the ⟨*input*, *output*, *cost*⟩ triples it produces.

(181) The language of a machine:

$$L(M) = \{\langle i, o, c\rangle \mid \text{there is a path } p = \langle\langle q_1, i_1, o_1, v_1, r_1\rangle, ..., \langle q_n, i_n, o_n, v_n, r_n\rangle\rangle,$$
$$\text{through } M \text{ where } i = (i_1 i_2 \ldots i_n),\ o = (o_1 o_2 \ldots o_n), \text{ and } c = (v_1 + ... + v_n)\}$$

With this description of the machines in mind it is easy to show that the language of the preoptimized machine is a subset of the language of the original machine. This is given as PREOPT lemma 1 in (182).

(182) **PREOPT lemma 1:**

$L(\text{PREOPT}(M)) \subseteq L(M)$ – preoptimized machines generate subsets of the languages generated by their non-preoptimized counterparts.

proof: By the definition of PREOPT, for every arc (q, i, o, w, r) in PREOPT(M) there is a path from q to r in M that reads i and writes o at a cost of c. Thus it follows from the definition of $L(X)$ that if $x \in L(\text{PREOPT}(M))$ then $x \in L(\text{PREOPT}(M))$. ■

In (183) I provide another lemma demonstrating that every optimal candidate generated by $\langle A(in), Eval\rangle^{\circledast}$ is also generated by $\langle A(in), \text{PREOPT}(Eval)\rangle^{\circledast}$.

(183) **PREOPT lemma 2:**

If p is an optimal path through $\langle A(in), Eval\rangle^{\circledast}$ that writes the candidate *out* then there's an optimal path p' through $\langle A(in), \text{PREOPT}(Eval)\rangle^{\circledast}$ that also writes *out*.

proof: PREOPT uses OPTIMIZE (whose correctness was shown in chapter three) to take a path p_i from node q to node r accepting the input segment i and writing the output string o at cost c and build an arc from q to r accepting i and writing o at cost c iff p_i was among the most harmonic paths from q to r accepting i.

Thus if p is a path from node q_0 to node r reading *in* and writing *out* at a cost of c in $\langle A(in), Eval\rangle^{\circledast}$, the only way for there **not** to be a path p' from node q_0 to

node *r* reading *in* and writing *out* at a cost of *c* in $\langle A(in), \text{PREOPT}(Eval)\rangle^{\boxtimes}$ would be for *p* to contain a subpath from q_i to q_j accepting one input segment for which there was a more harmonic path from q_i to q_j accepting that same input segment. But, by the optimal subpath lemma and the assumption that *p* is an optimal path, this is not possible. ■

From these two lemmas the correctness of PREOPT follows straightforwardly. Lemma 1 tells us that PREOPT doesn't introduce any new parses, and lemma 2 tells us that, though preoptimization may destroy (infinitely) many parses, all of the parses that are destroyed by preoptimization in are guaranteed to have been suboptimal. In (184) I use these two lemmas to provide a correctness proof for PREOPT.

(184) **Theorem**: correctness of PREOPT

For any input *in*, if the string *out* is an output candidate from an optimal path through $\langle A(in), E\rangle^{\boxtimes}$ then there is an optimal path through $\langle A(in), \text{PREOPT}(E)\rangle^{\boxtimes}$ that also outputs the candidate *out*.

proof: By PREOPT lemma 2, if *cand* is the output candidate from an optimal path through $\langle A(in), E\rangle^{\boxtimes}$ it is an output candidate from a path through $\langle A(in), \text{PREOPT}(E)\rangle^{\boxtimes}$. By PREOPT lemma 1, there are no competitor candidates in $\langle A(in), \text{PREOPT}(E)\rangle^{\boxtimes}$ that weren't present in $\langle A(in), E\rangle^{\boxtimes}$ thus *cand* is optimal in $\langle A(in), \text{PREOPT}(E)\rangle^{\boxtimes}$. ■

Assessing the complexity of PREOPT is relatively straightforward. As defined in (171) PREOPT requires that OPTIMIZE be run once for each ⟨segment, node, node⟩ trio. If *e*

is the number of nodes in *Eval* and *s* is the number of segments in the inventory then we have se^2 runs of OPTIMIZE. Each run of OPTIMIZE covers the intersection of a single input segment with *Eval* which yields a machine with at most 2*e* nodes. The complexity of OPTIMIZE is roughly quadratic, so the complexity of PREOPT is basically $s \times e^2 \times (2e)^2$ or simply $\Theta(e^4)$.

We can reduce the complexity of preoptimization from quartic to cubic by using an all-pairs shortest paths algorithm like the Floyd-Warshall algorithm (cf. Cormen et al. 2001: 620) once per input segment to build the arcs of the preoptimized machine. Since the complexity of the Floyd-Warshall algorithm is cubic in the number of nodes and we'll need *s* runs of the algorithm on machines with *e* nodes apiece, the overall complexity is $s \times e^3$ or simply $\Theta(e^3)$. This difference isn't terribly relevant in practice because the PREOPT algorithm only needs to be run once for a given grammar and doesn't need to be run on-line to produce optimal forms.

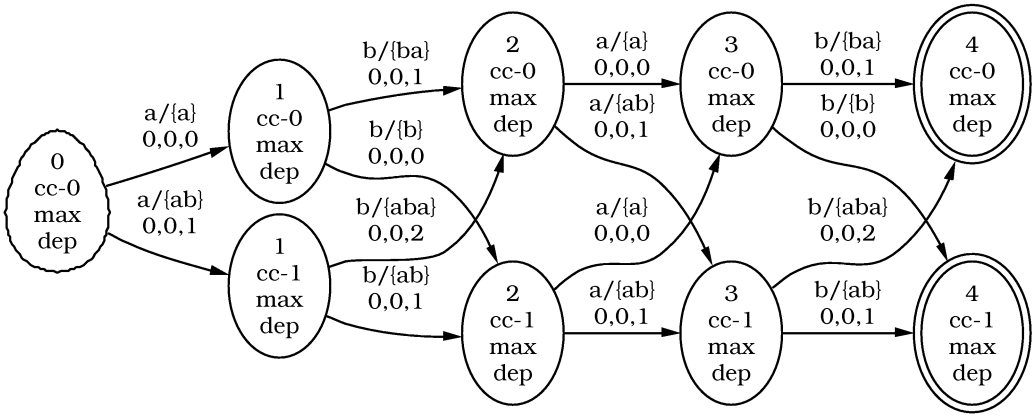
6.5 Optimization after PREOPT

In the previous section I showed that any optimal parse (input output pairing) in the intersection of an input with *Eval* will also be an optimal parse in the intersection of that input with preoptimized *Eval* and vice versa. This means that OPTIMIZE could be run on the intersection of an input with preoptimized *Eval* and it would yield the same result as running it on the intersection of that same input with *Eval* without preoptimization.

Nonetheless, the point of preoptimization is to obviate the need for the OPTIMIZE algorithm altogether and enable us to use something simpler and more efficient to generate

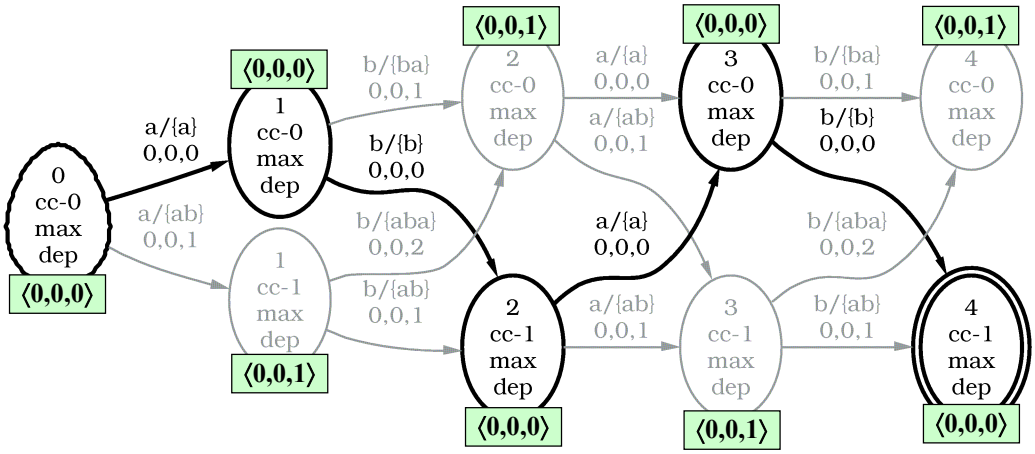
optimal forms. Consider in (185) the evaluation of the input string /abab/ with preoptimized *Eval* for the ranking *CC ≫ MAX ≫ DEP, repeated from §6.1 above.

(185) ⟨*A*(abab), PREOPT(⟨*CC, MAX, DEP⟩)⟩⋉



Running OPTIMIZE on the machine in (185) reveals the cost of the most harmonic paths to each node. With this information, finding the optimal paths through the machine is trivial. In (186) I annotate the nodes with their cost attributes and outline the optimal path in bold.

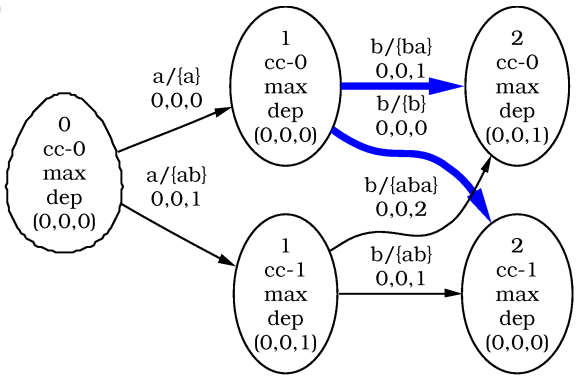
(186) OPTIMIZE(⟨*A*(abab), PREOPT(⟨*CC, MAX, DEP⟩)⟩⋉)



As a consequence of the linear structure of the input acceptor and the fact that preoptimization removes epenthetic cycles in *Eval*, it is the case that in (185) nodes with index *i* are ancestors of nodes with index *j* just in case *i* is less than *j*.

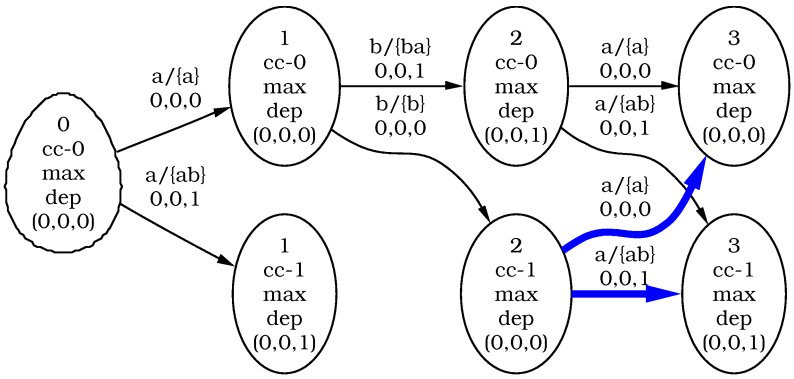
The linear structure of these machines makes it possible to optimize them without considering the structure of the entire machine. By building the machines in chunks that accept single input segments and proceeding left-to-right one segment at a time through the input, it's possible to build machines encoding all and only optimal parses. Consider, in (187), the acceptance of the second segment of the input string /abab/.

(187)



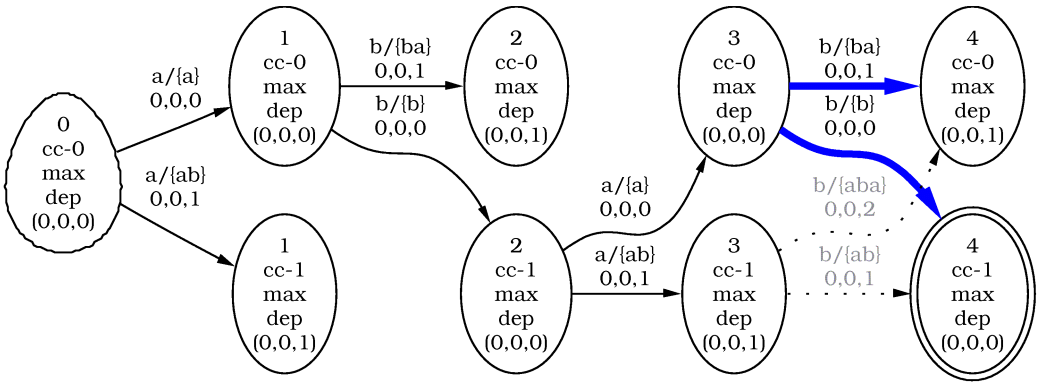
In (187) I've annotated each node with the cost of the most harmonic path to it. From the nodes at index 1 there are four arcs in *Eval* that could be used to reach the nodes at index 2. At this point, however, we know that only the arcs in bold can participate in an optimal path. This is so because the bold arcs reveal paths from the start state to the nodes at index 2 that are more harmonic than any path through (1, (cc-1, max, dep), ⟨0,0,1⟩). By simply failing to build the arcs originating at (1, (cc-1, max, dep), ⟨0,0,1⟩), the suboptimal paths that they encode are avoided. Consider in (188), arcs accepting the next segment.

(188)



Just as with the nodes at index 1, of the four arcs from *Eval* that might be used to reach the nodes at index 3 from the nodes at index 2, only the two in bold could possibly participate in an optimal path. Again, by simply failing to build the suboptimal arcs, the suboptimal paths that they encode are avoided. Consider in (189) the completion of the machine with the last segment of the input.

(189)



The same comparison made at indices 1 and 2 reveals that the bold arcs in (189) can be part of optimal paths and that the dotted arcs in (189) can't. By building only the bold arcs and allowing only the cheaper of the two possible final nodes to be final, we finish a machine that encodes only the optimal parses of /abab/ under *Eval*.

In (190) I present a function that performs the comparison of the arcs illustrated in (187) through (189). This function takes the set of states currently in the machine, the index of the rightmost states, an input segment, and the arc-set of preoptimized *Eval* and yields the set of optimal arcs accepting the next segment of the input.

(190) $bestArcs(Q, id, in, \delta^{E}) = \{((id, q, c), in, o, (id+1, r, c+w)) \mid (id, q, c) \in Q,$
$(q, in, o, w, r) \in \delta^{E}$ and there are no $(id, q', c') \in Q$, and
$(q', i, o', w', r) \in \delta^{E}$ such that $(c'+w') \succ (c+w)\}$

In this function I assume that the nodes are annotated with the costs of the most harmonic paths that reach them. With this function in hand it is possible to formulate an algorithm that uses *bestArcs* to build machines that generate all and only optimal parses of an input segment under preoptimized *Eval*. In (191) I present the algorithm OPT(*in*, *Ev*) in pseudo-code with comments.

(191) OPT$(in, (Q^{E}, \Sigma, \delta^{E}, q_0, F^{E})) = (Q, \Sigma, \delta, (0, q_0, \overline{0}), F)$

1	$Q \leftarrow \{(0, q_0, \overline{0})\}$	- Put the start state in Q.
2	$id \leftarrow 0$	- Set the starting index to 0.
3	**while** $in = \langle i, \langle in' \rangle \rangle$	- While segments remain to be done:
4	**do** $in \leftarrow in'$	remove the 1st segment from *in* and
5	**for** $A = bestArcs(Q, id, in, \delta^{E})$	get the best arcs from *id* reading *in*
6	$\delta \leftarrow \delta \cup A$	add the new arcs to δ
7	$Q \leftarrow Q \cup \{r \mid (q, i, o, r) \in A\}$	add the new nodes to Q
8	$id \leftarrow id + 1$	increment the index counter by 1.
9	$F = \{(id, q, c) \mid (id, q, c) \in Q, q \in F^{E},$ and there is no $(id', q', c') \in Q,$ such that $q' \in F^{E}$ and $c' \succ c\}$	- Keep only the cheapest possible finals.

To show that this algorithm is correct I'll show that the machine that it creates for any given input is the same as that derived from running OPTIMIZE on the intersection of that input with *Eval*. The correctness follows straightforwardly from the similarity of the comparison steps that are used to construct the arcs in the OPT algorithm and the arc removing portion of OPTIMIZE.

(192) **Theorem**: correctness of OPT

Given preoptimized *Ev*, every path in OPT(*in*, *Ev*) is identical, save for the costs annotating the node names, to a path in OPTIMIZE($\langle A(in), Ev\rangle^{\boxtimes}$) and vice versa.

proof: I will show that this is so by proving that the costs annotating the nodes are the same as the cost attributes in the cost table built by OPTIMIZE. From there it follows that the arcs of the two machines are the same because in both machines the arcs are drawn from *Ev* and are present just in case their cost is equal to the cost associated with their terminus minus the cost associated with their origin.

For each node (*id*, *ev*, *c*) in OPT(*in*, *Ev*) the cost attribute $o[(id, ev)]$ for the node (*id*, *ev*) in OPTIMIZE($\langle A(in), Ev\rangle^{\boxtimes}$) equals *c*. I'll show that this is true by induction on the indices. For the base case, it is easy to see that at the start states $(0, q_0, \overline{0})$ and $(0, q_0)$ this holds trivially. This is so because $o[(0, q_0)]$ is set to $\overline{0}$ at the outset of the OPTIMIZE algorithm.

If the similarity holds at index *n* it must hold at index *n*+1. This is so because the two machines use the exactly the same set of arcs *A* from *Ev* to connect nodes at indices *n* and *n*+1 and because the costs associated with nodes at *n*+1 in both

machines are the most harmonic combination of the cost of an arc in A and the cost for that arc's origin at index n. Thus since the costs at n are the same and the costs of the arcs in A are the same, the costs at n+1 must be the same as well. ■

The OPT algorithm does not require very much computation. The algorithm goes through the input one segment at a time comparing at most e^2 arcs to one another for each segment (at most one arc from each node of *Eval* at index n to each node of *Eval* at index n+1). Thus the complexity is basically $|in| \times e^2$. Because e is a constant of the grammar, the computation required in constructing OPT(*in*, *Eval*) goes up as a linear function of the length of *in*. Thus with preoptimized *Eval*, optimization can be done in linear time.

7 Transducing Optimality

How complex is phonology? Given the fact that most phonological phenomena are describable with simple rewrite rules, the answer must be a qualified "not very." There are, of course, complexities like reduplication, directionality, opacity, optionality, lexical strata, paradigm uniformity, ordering paradoxes, iterativity, and exceptionality that present varied difficulties for various phonological models. But, nonetheless, the "vanilla" phonological phenomena that constitute the bulk of what's observed in phonological grammars cross-linguistically seem to be fairly simple.

In rule-based phonological frameworks this simplicity is reflected in the fact that rules can be represented as transducers that take underlying forms containing the rule's environment of application and map them to surface forms in which the rule has been applied (Johnson 1972). The transducers can then be composed into a single transducer that encodes the serial application of a sequence of phonological rules (Kaplan and Kay 1994). If the whole grammar (the sequence of rules) is represented as a single transducer it can then be inverted and fed output forms to generate the set of input forms that map to those outputs. Such a set-up has the advantage that, once the transducer has been constructed, using the grammar for recognition is no more complex than using it for generation.

In constraint-based phonological frameworks things don't seem so straightforward. This is a bit odd, given that the object of study for both the rule-based and constraint-based frameworks is the same. The differences arise partly from the fact that the phenomena that are describable by optimization with grammars of ranked violable constraints are more complex than those describable with ordered rules. Indeed, Frank and Satta (1998) show

that even if constraints are restricted to those expressible with finite state transducers, the very process optimization can generate i/o-relations that are more complex than those that can be generated with finite state transducers alone. Before retuning to this issue in §7.5, I'll show that it is possible to find finite state transducers that define a large portion of the simple phonological patterns that are describable in constraint-based systems.

Specifically, I'll present here an algorithm for constructing transducers that map input forms directly to the output forms that would be selected as optimal under a given ranking of a set of constraints. With such transducers it will be possible to generate optimal output forms without the need to do optimization on each input/output pair. By encoding optimal parses directly into a finite state transducer it will thus be possible to circumvent the need for on-line optimization.

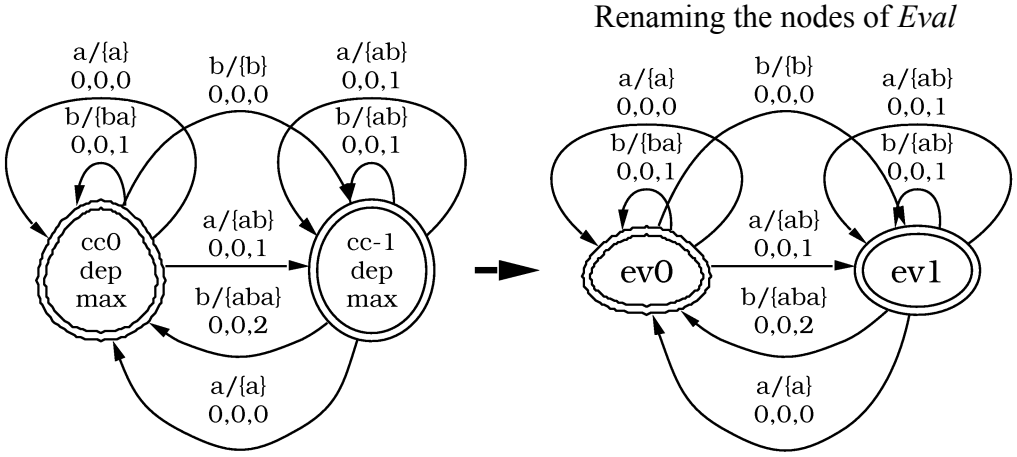
Because transducers can be inverted, once we have a transducer that maps inputs directly to optimal outputs, it is relatively easy to generate the input strings that map to a known output under a given grammar. Thus transducers will allow us to do recognition.

7.1 Relativity

The core insight that was behind the preoptimization technique presented in chapter six lies at the heart of transducer construction. The crucial observation is that it's possible to detect and eliminate chunks of parses (partial i/o-mappings) that are guaranteed to be suboptimal without access to global information about the range of all possible parses for a given input.

In (193) I present PREOPT(⟨*CC, MAX, DEP⟩). To make the nodes easier to refer to I'll rename them ev0 and ev1. Through §7.3 I'll use the term *Eval* to refer to the machine presented in (193).

(193) PREOPT(⟨*CC, MAX, DEP⟩)



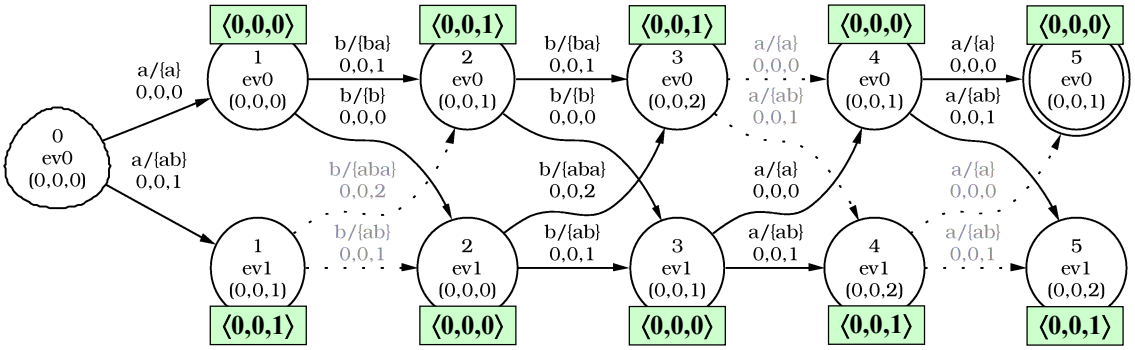
Consider in (195) and (196) the machines OPT(abbaa, *Eval*) and OPT(baabb, *Eval*). For these machines I've indicated in a box next to each node the cost of the most harmonic path that reaches that node expressed relative to the costs of the most harmonic paths to the other nodes at the same index.

To obtain these relative costs I "normalized" the costs annotating the nodes by re-expressing each coordinate of each vector relative to the minimal value at that coordinate for any cost vector in the set of nodes sharing the same index. The function used to create the normalized costs for the nodes is given in (194).

(194) $norm(V) = \{\langle (v_1 - b_1),...,(v_n - b_n)\rangle \mid \langle v_1,...,v_n\rangle \in V \text{ and } b_i = min\{w_i \mid \langle ... \ w_i \ ...\rangle \in V\}\}$

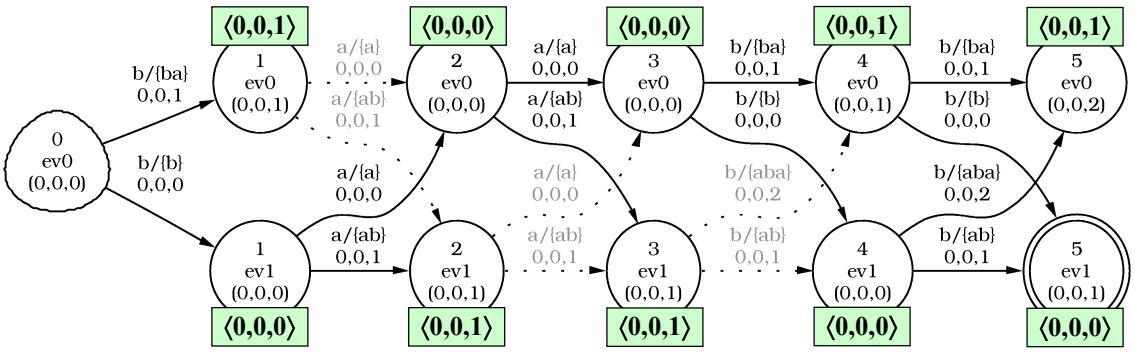
Attending to the relative costs reveals several recurring patterns in (195) and (196).

(195) OPT(abbaa, *Eval*)



Subtracting ⟨0,0,1⟩ from both nodes at index 3 shows their relative costs to be the same as the relative costs for the nodes at index 2. The process of normalization can be seen as an instance of "mark cancellation" whereby any violations that are shared by all competitors are factored out of the computation of optimality (Prince and Smolensky 1993: 42). This removal of violations is valid because every single candidate generated by (195) comes from a path that passes through exactly one of the nodes at index 3. Normalization reveals the same kind of recurring pattern in OPT(baabb, *Eval*).

(196) OPT(baabb, *Eval*)



The examples in (195) and (196) strikingly demonstrate that, for many of the arcs in the machine, the determination of whether they are among the potentially optimal acceptors for segment *i* at index *id* depends only on the relative costs of the nodes at index *id*. That is, whenever a particular set of relative costs occurs, the same arcs will be locally optimal in accepting the next input segment regardless of what comes before or after that segment.

Another interesting pattern that (195) and (196) illustrate is the fact that the set of relative costs for the nodes at index *id*+1 depends entirely on the relative costs at index *id* and on the input segment that labels arcs from nodes at *id* to *id*+1. This local dependence can be exploited in constructing machines that encode optimal parses.

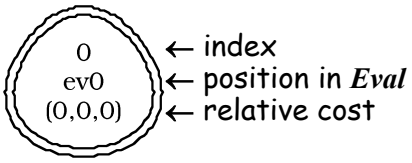
7.2 Infinite generalization: the simple case

When assessing various paths through a machine, it is the differences among the costs of those paths, not their absolute costs, that are relevant in determining which paths are optimal. By abstracting away from the absolute costs of the paths it is possible to see the grammar defined by PREOPT(⟨*CC, MAX, DEP⟩) as having essentially two states. In one state the relative cost of ev0 is ⟨0,0,1⟩ and the relative cost of ev1 is ⟨0,0,0⟩ and in the other state the relative cost of ev0 is ⟨0,0,0⟩ and the relative cost of ev1 is ⟨0,0,1⟩.

By capitalizing on the regularities that arise in machines like (195) and (196) it is possible to construct a new machine that contains all and only the paths (candidates) that the OPT algorithm generates for any input. This can be done incrementally by building the machine one segment at a time and asking with each added segment which arcs the OPT

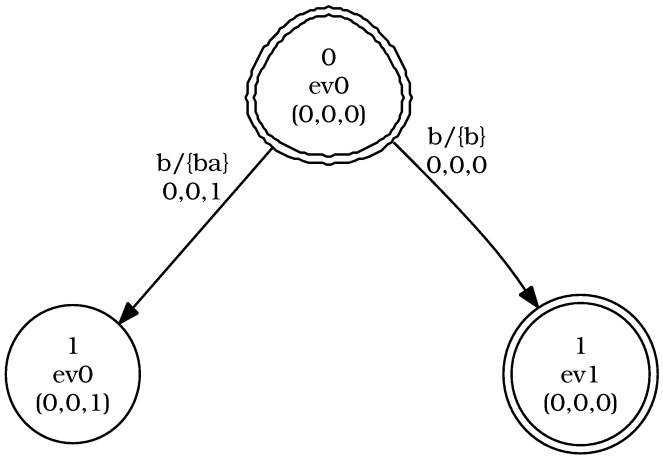
algorithm would provide to accept that input segment. Consider in (197) the start state of OPT(*s*, *Eval*) – the start state will be the same regardless of the input string *s*.

(197) The start-state:



Because the machine in (197) accepts no input strings at all, it is trivially true that it generates all and only optimal parses for every string it accepts. From this starting point we ask which arcs could possibly be optimal if the input /b/ were accepted at this state. In (198) I extend the machine with arcs that accept the input symbol /b/.

(198) Extending the machine with the segment /b/:

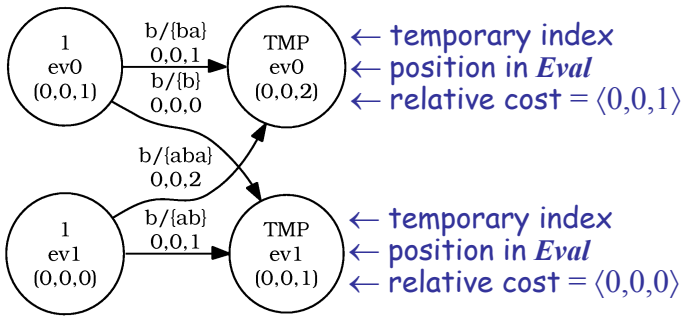


Though both ev0 and ev1 are final states in *Eval*, the fact that (1, ev1, (0,0,0)) can be reached more harmonically than (1, ev0, (0,0,1)) while reading the same input segment means that the former is a viable final state but the latter is not. At this point the machine

is exactly the same as OPT(b, *Eval*) thus it follows that (198) generates all and only the optimal parses for the input /b/ under the grammar defined by *Eval*.

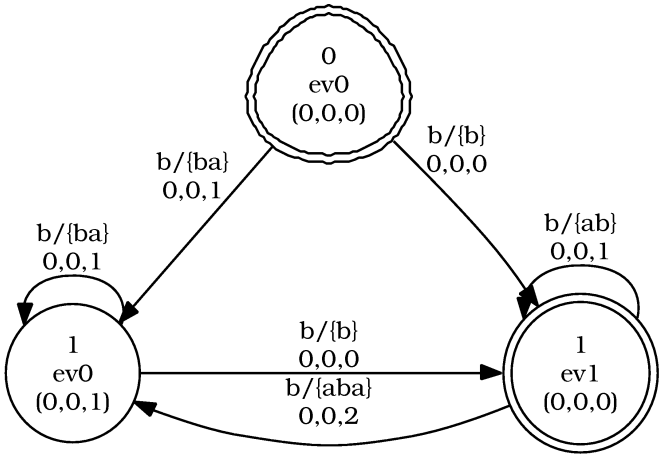
In (199) we ask what arcs could be optimal if another /b/ were accepted from the nodes at index 1. If we were constructing the machine according to the OPT algorithm of chapter six, another set of arcs and nodes with the index 2 would be added to the machine to accept the additional input segment. Instead of introducing the index 2, at this point in (199) I give the newly created nodes a "temporary" index TMP.

(199) Accepting /b/ :



Since we know that accepting more /b/'s will keep giving us at the same pair of relative costs (⟨0,0,1⟩ for ev0 and ⟨0,0,0⟩ for ev1), it's possible to make a generalization at this point. That is, rather than building a new pair of nodes at a new index we can give the nodes at the termini of the newly created arcs the index 1, which is already present in the machine. This creates the loop shown in (200).

(200) Extending the coverage with another /b/



In this single move we've made an infinite generalization. That is, the machine in (200) now accepts any string of b* and maps it to its optimal output under *Eval*. This will be proven below, but first I'll explain in detail the construction step used to create (200).

The first ingredient is the *bestArcs* function from chapter six with the modification that instead of assigning the nodes at the termini of the newly created arcs an index one higher than the index of their origins, it assigns them a temporary index TMP. The revised *bestArcs* function is given in (201).

(201) Revised *bestArcs* function:

$$\begin{aligned} bestArcs(Q, id, in, \delta^{E}) = \{((id, q, c), in, o, w, (\text{TMP}, r, c+w)) \mid (id, q, c) \in Q, \\ (q, in, o, w, r) \in \delta^{E} \text{ and there are no } (id, q', c') \in Q, \text{ and} \\ (q', i, o', w', r) \in \delta^{E} \text{ such that } (c'+w') \succ (c+w) \} \end{aligned}$$

To facilitate generalization, the costs annotating the termini of the newly created arcs will be normalized with the function in (202).

(202) A function for normalizing the node-costs in newly created arcs:

$$norma(\delta) = \{(q,i,o,c,(id,r,\langle v_1,...,v_n\rangle)) \mid (q,i,o,c,(id,r,\langle w_1,...,w_n\rangle) \in \delta \text{ and}$$
$$v_i = (w_i - b_i) \text{ for } min\{b_i \mid (q',i',o',c',(id,r',\langle b_1 ... \ b_i \ ... \ b_n\rangle) \in \delta\}\}$$

Once the arcs have been created and the costs annotating their termini have been normalized it's possible to check whether the set of relative costs for their termini is already present at some index in the machine. If so, the nodes are coindexed with the nodes that share their relative costs; if not, the nodes are given a new index that hasn't yet been used in the machine. This move is accomplished with the *index* function defined in (204). First, in (203) I give a term to refer to the set of nodes in a machine that share a particular index.

(203) $Q_{id} = \{(id,ev,cst) \mid (id,ev,cst) \in Q\}$ – the subset of Q with the index id.

(204) $index(Q, \delta) = (Q', \delta')$ where

if there is an id such that $\{(id,ev,cst) \mid (q,i,o,(\text{TMP},ev,cst)) \in \delta\} = Q_{id}$ then

$$\delta' = \{(q,i,o,c,(id,ev,cst)) \mid (q,i,o,c,(\text{TMP},ev,cst)) \in \delta\},$$

else for $id' \in \mathbb{N}$ such that $(id',ev',cst') \notin Q$

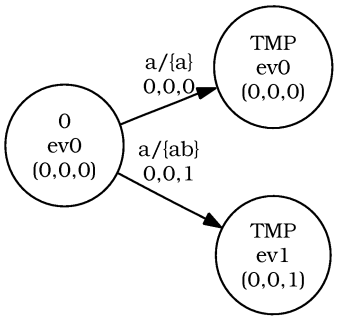
$$\delta' = \{(q,i,o,c,(id',ev,cst)) \mid (q,i,o,c,(\text{TMP},ev,cst)) \in \delta\}.$$

$$Q' = Q \cup \{r \mid (q,i,o,r) \in A'\}.$$

The first clause of (204) indexes new arcs with an existing index if there's already a set of nodes in the machine with the same set of relative costs. Barring this, a new index is created for the termini of the new arcs. Either way, the node-set Q is updated to include the termini of the recently created arcs.

Now that the machine accepts any sequence of /b/'s as input, we ask what happens if the segment /a/ is accepted at the start state. In (205) I give the arcs that accept /a/.

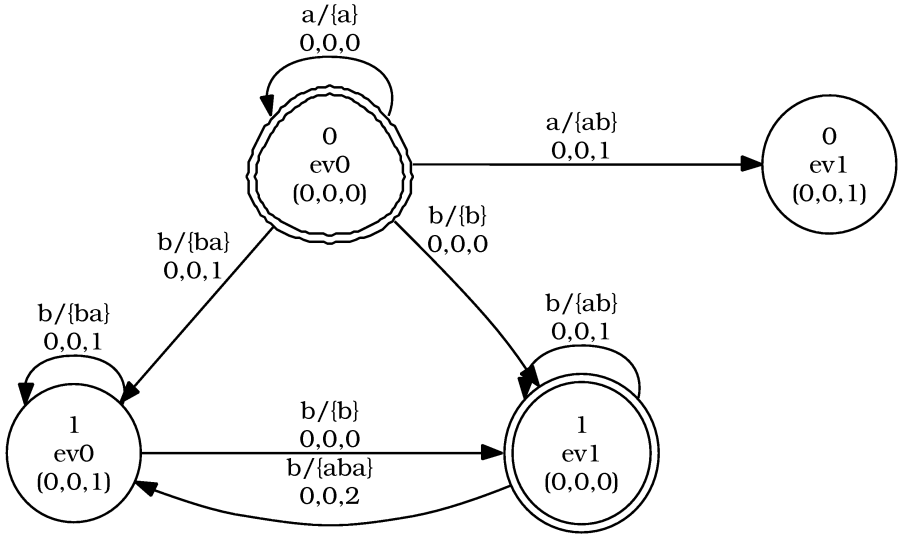
(205)



Accepting /a/ from the start yields a pair of nodes with the relative costs of ⟨0,0,0⟩ and ⟨0,0,1⟩. The first of these costs is already expressed on the start state. This situation suggests that we might index the new nodes with 0, thus making another generalization.

In (206) I give the machine resulting from assigning index 0 to the termini of the new arcs. A revised version of the indexing function will be given below in (207).

(206) Extending the coverage with the input /a/:



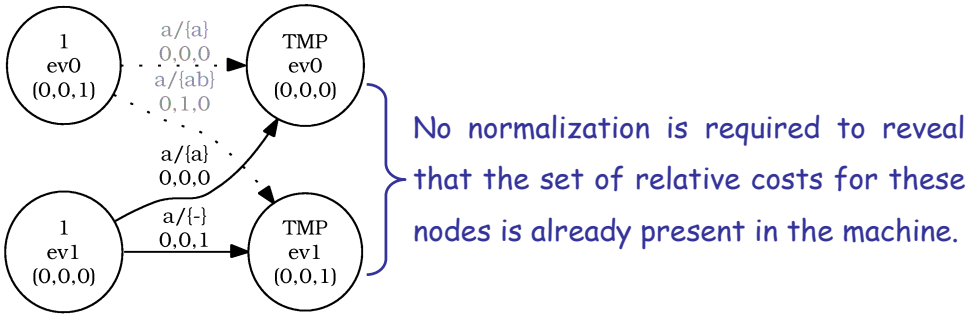
In (206) I've constructed a new set of nodes at index 0 by linking arcs accepting /a/ at index 0 back to nodes at index 0. This is acceptable because the arcs that currently originate at index 0 will remain optimal with this addition. The following revision to the indexation function will cover this case.

(207) $index(Q, \delta) = (Q', \delta')$ where if there's an *id* such that

$Q_{id} \neq \varnothing$, $T = \{(id, ev, cst) \mid (q, i, o, c, (\text{TMP}, ev, cst)) \in \delta\}$, $Q_{id} \subseteq T$, and
for each $in \in \Sigma$ $bestArcs(Q_{id}, id, in, \delta^{E}) = bestArcs(T, id, in, \delta^{E})$
then $\delta' = \{(q, i, o, c, (id, ev, cst)) \mid (q, i, o, c, (\text{TMP}, ev, cst)) \in \delta\}$,
else for $id' \in \mathbb{N}$ such that $(id', ev', cst') \notin Q$
$\delta' = \{(q, i, o, c, (id', ev, cst)) \mid (q, i, o, c, (\text{TMP}, ev, cst)) \in \delta\}$.
$Q' = Q \cup \{r \mid (q, i, o, r) \in A'\}$.

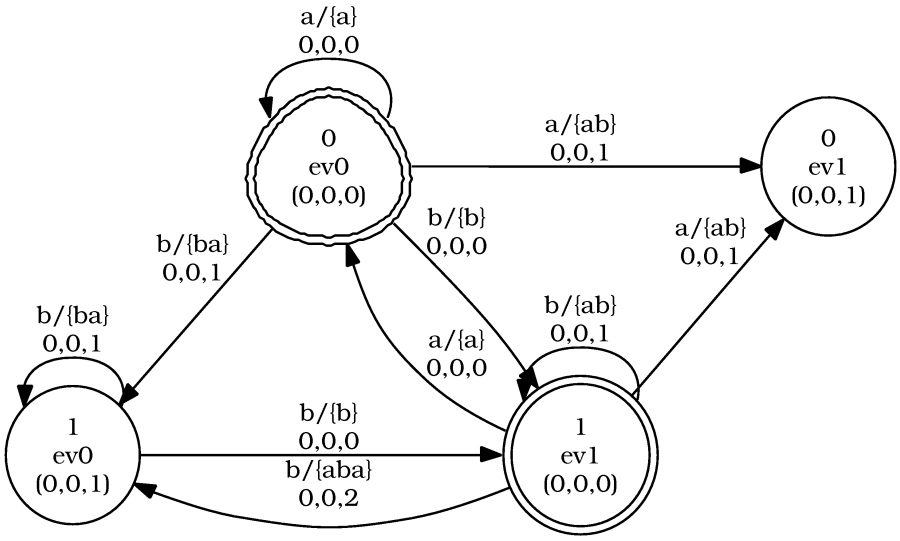
Next we ask which arcs optimally accept the symbol /a/ as the next input segment from the nodes at index 1. The potential new arcs are given in (208).

(208) Arcs accepting /a/ from index 1:



Giving the nodes at the termini of the arcs in (208) index 0 extends the machine to accept input /a/ from the nodes at index 1. After this step the machine is finished because every input is accepted from nodes at every index. The finished product is given in (209).

(209) The finished transducer:



The machine in (209) now has arcs accepting every input segment at every index. Because every input string is accepted by the nodes at exactly one index and is accepted by all of the nodes at that index, the machine accepts every one of the infinite set of input strings drawn from {a, b}*. Moreover, this machine maps every input string that it accepts to all and only its optimal output forms. This will be proven in §7.3, but first I'll present the transducer construction algorithm in a bit more detail.

7.3 The Optimality Transducer Construction Algorithm

In (210) I present the Optimality Transducer Construction Algorithm or OTCA in pseudo-code with comments. After proving that this version of the algorithm is correct I'll move on to slightly more complex cases of generalization.

(210) OTCA(Σ^I , (Q^E, Σ^E, δ^E, $q_0{}^E$, F^E)) = (Q, Σ, δ, q_0, F)

1	$q_0 \leftarrow (0, q_0{}^E, \overline{0})$	- Build the start state.
2	$Q \leftarrow \{q_0\}$	- Put the start state in Q.
3	$\delta \leftarrow \varnothing$	- Initialize the arc-set to null.
4	$T \leftarrow \{0\}$	- Add the index 0 to T – the "To do" list
5	**while** $T \neq \varnothing$	- while there are indices in the To do list
6	$T \leftarrow T - \{id\}$	remove one index from the list,
7	**for** each $i \in \Sigma$	for each segment in the inventory,
8	**do** $A \leftarrow bestArcs(Q, id, i, \delta^E)$	find the best arcs from id accepting i,
9	**do** $A' \leftarrow norma(A)$	normalize the costs on the termini, and
10	**do** $Q \leftarrow Q'$, $index(Q, A') = (Q', A'')$ $\delta \leftarrow \delta \cup A''$	add the newly created arcs and nodes to the machine.
11	$F = \{(id, ev, c) \mid (id, ev, c) \in Q, ev \in F^E,$ and there's no $(id', ev', c') \in Q$ such that $ev' \in F^E$ and $c' \succ c\}$	- Keep only the cheapest final at each index as a final in the finished machine.

To facilitate the proof of the correctness of the OTCA I'll abstract away from the algorithm to just the change made by each iteration of the while loop in **step 3**. I'll call the change in question a "best extension" just in case it meets the following criteria.

(211) **Best extension:**

def: (Q^+, Σ, δ^+, q_0, F^+) is a best extension of (Q, Σ, δ, q_0, F) under (Q^E, Σ, δ^E, $q_0{}^E$, F^E) iff it's the case that for every arc ((id, ev_1, c), i, o, w, (id_x, ev_2, c_x)) in (Q^+ - Q) there are no (id, q', c') $\in Q$ and (ev_1, i, o', w', ev_1) $\in \delta^E$ such that $(c' + w') \succ (c + w)$.

The fact that each set of arcs added to the machine makes a best extension follows directly from the definition of the *bestArcs* function. To show that the algorithm is correct I'll prove that if a machine is built from best extensions then every path in that machine accepting string *s* is an optimal *s*-accepting path between the start state and some node.

In (212) I define *E*-equivalence, this will come in handy in showing that the set of paths in the result of the OTCA are the same as those that occur under optimization.

(212) $(id, ev, c) \equiv_E (id', ev, c')$ – nodes with the same *Eval* component are *E*-equivalent.

Arcs (*q*, *i*, *o*, *r*) and (*q*′, *i*, *o*, *r*′) are *E*-equivalent iff $q \equiv_E q'$ and $r \equiv_E r'$.

Two paths are *E*-equivalent iff they consist entirely of *E*-equivalent arcs.

Two sets of paths A and B are *E*-equivalent iff every path in A is *E*-equivalent to a path in B, and every path in B is *E*-equivalent to a path in A.

Having defined *E*-equivalence, it's easy to say exactly what it means for a machine (the transducer under construction) to generate exactly the same parses (i/o mappings) as optimization for any input string. The property that we are after is defined in (213).

(213) A machine *M* is **OPT equivalent** to *Ev* iff for any string *s* the set of paths through *M* that accept *s* are *E*-equivalent to the set of paths through OPT⟨*s*, *Ev*⟩ that accept *s*.

With this definition in hand I will show that transducer construction is correct by proving that each step in transducer construction preserves OPT equivalence. Because the machine resulting from first step in the OTCA (the start state) is trivially OPT equivalent we have a good basis for induction. If it can be shown that the OPT equivalence is invariant throughout transducer construction then the correctness will be established.

(214) **Theorem**: best extensions preserve OPT equivalence

Any best extension of *M* under *Ev* preserves OPT equivalence to *Ev*.

proof: For a contradiction, assume that after a best extension under *Ev* the paths through *M* accepting *s* are not equivalent to the paths through OPT⟨*s*, *Ev*⟩ for some *s*.

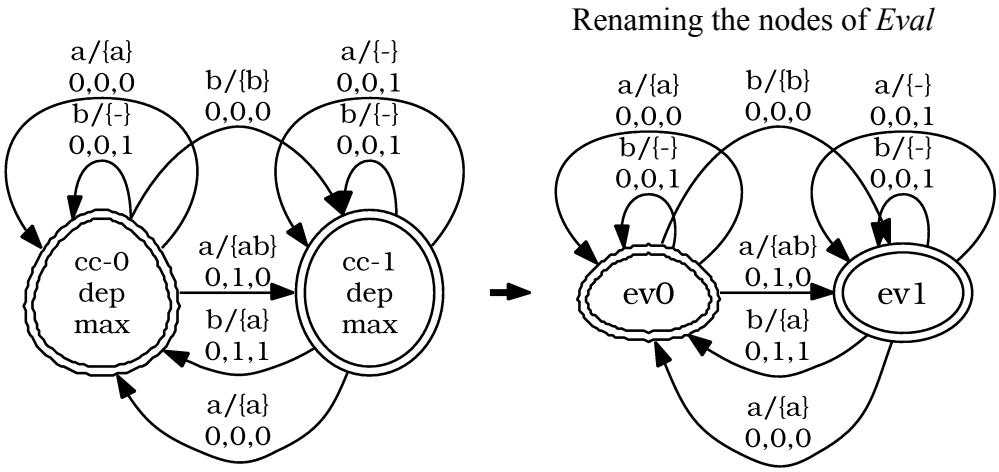
Thus there must be some first segment s_i of *s* where the arcs accepting s_i from index *a* to index *b* in *M* are not equivalent to the arcs accepting s_i from index *x* to index *y* in OPT⟨*s*, *Ev*⟩. But, since s_i was the first point of non-equivalence, the same set of relative best costs must annotate the nodes at index *a* and index *x*. Moreover, because *Ev* contributes the same set of possible arcs to both machines, the set of possible arcs from *a* to *b* are equivalent to the set of possible arcs from *x* to *y*. Thus, because the same arcs in both sets yield the set of cheapest relative best costs for the nodes at index *b* and index *y*, the arcs accepting s_i must be equivalent, and thus we have a contradiction. ■

The above theorem shows that the OTCA is sound. That is, that every i/o-mapping the transducer generates is an optimal parse. Given this, all that's needed to establish the correctness of the algorithm is proof that it encodes an i/o-mapping for every input string. The completeness of the OTCA follows trivially from the fact that the algorithm only terminates once every input symbol can be accepted from every index taken together with the fact that every input string accepted leads to all of the nodes at exactly one index. Thus, the OTCA is sound and complete.

7.4 Generalization with variables

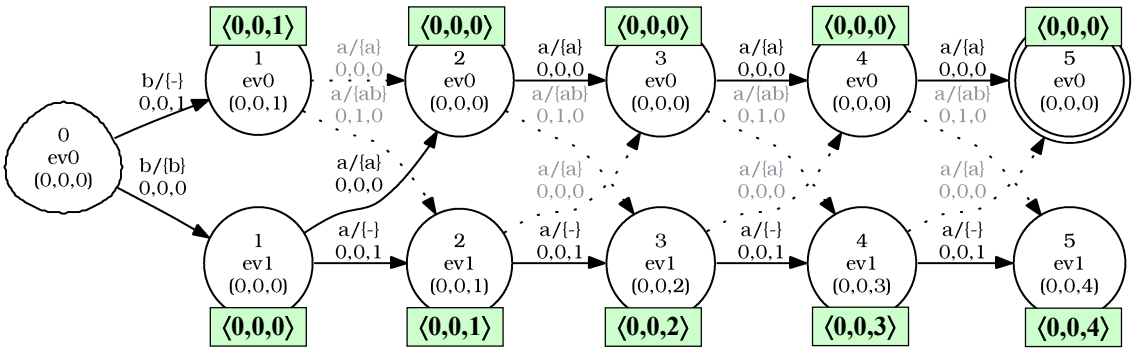
Transducer construction isn't always as straightforward as the case given in §7.2. For a slightly more complex case that uses exactly the same constraints, consider in (215) *Eval* for the ranking *CC ≫ DEP ≫ MAX.

(215) PREOPT(⟨*CC, DEP, MAX⟩)



For the remainder of this section I'll refer to PREOPT(⟨*CC, DEP, MAX⟩) as *Eval*. Consider in (216) the machine resulting from OPT(baaa, *Eval*). Next to each node in the machine I've indicated the relative cost of the most harmonic path to that node.

(216) OPT(baaa, *Eval*)

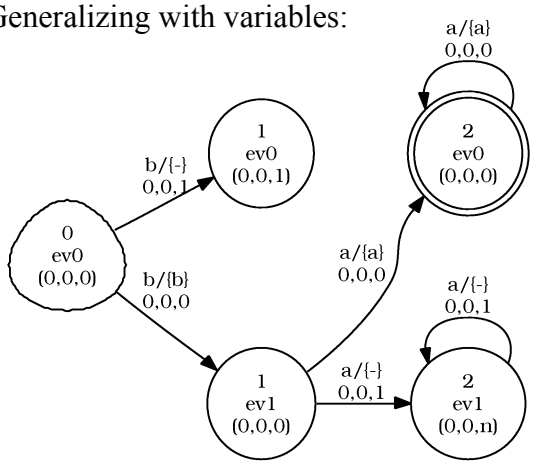


Unlike the machines in §7.3, the patterns in (216) cannot be captured by simply normalizing the costs at each index. This is so because the relative costs of ev0 and ev1 nodes can grow ever more disparate as longer and longer sequences of /a/'s are considered.

Nonetheless, there is a very obvious pattern playing out in (216) that is fairly easy to generalize. With each successive /a/ the cost of the best path to the node at ev1 goes up by ⟨0,0,1⟩ and the cost of the best path to the node at ev0 does not change at all. More importantly, the exact disparity between the costs of the nodes at ev0 and ev1 is never relevant in choosing which arcs to extend the machine with. No matter how many /a/'s are added, ev1 will always cost more than ev0, ev1 will not be a viable final state, and the same set of arcs will be used to accept the next input segment at each successive iteration.

To capture this pattern I'll introduce variables into the costs annotating the nodes. Consider in (217) a machine just like the one in (216) but with the change that the nodes at indices 2 through 5 have been collapsed down to a single pair of nodes at index 2.

(217) Generalizing with variables:



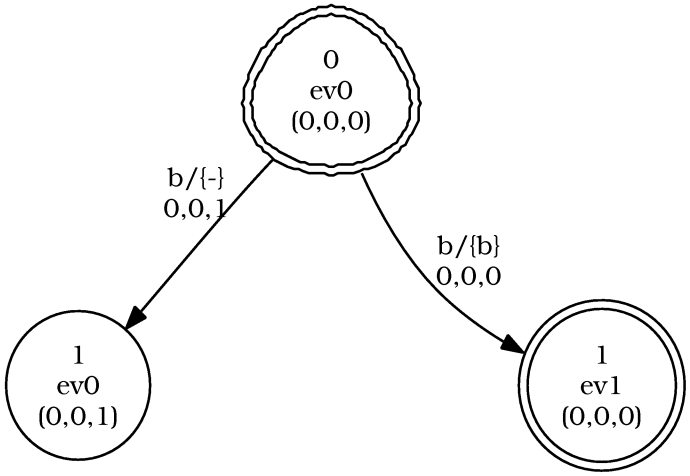
In (217) the variable *n* in the cost annotating node (2, ev1, ⟨0,0,*n*⟩) stands for any number greater than one. The collapsing of the nodes at indices 2-5 is licensed in this case

by the fact that, for all values of *n* greater than one, the same set of arcs will be used to accept the input /a/. For this generalization to be valid, all daughters of the node with the variable must inherit the variable, and it must remain true that for all values of *n* greater than one the same arcs are optimal in accepting the next input segment. I will lay this out more explicitly below; for now let's go through transducer construction with the additional possibility of variable introduction to allow generalization. The OTCA always begins with the start state at index 0. This is given in (218).

(218) The start sate:

In (219) the machine is extended to accept the input symbol /b/. At this point the choice of which arcs to add to the machine to accept the input /b/ is simple because there is only one arc leading away from ev0 to each of the other nodes in *Eval* that accepts /b/.

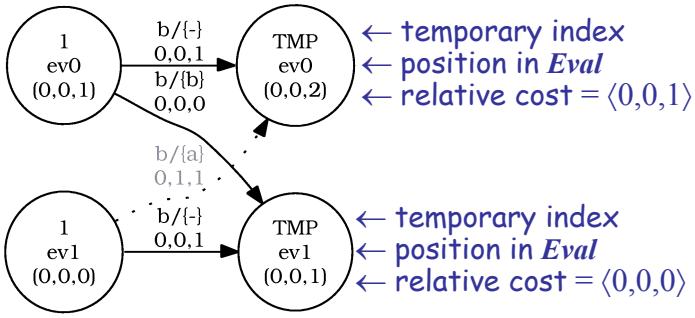
(219)



Though both ev0 and ev1 are final states in *Eval*, the fact that (1, ev1, (0,0,0)) can be reached more harmonically means that only it will be final in the transducer. At this point (219) is exactly the same as OPT(b, *Eval*) and as such we know that it generates all and only the optimal parses for the input /b/ under *Eval*.

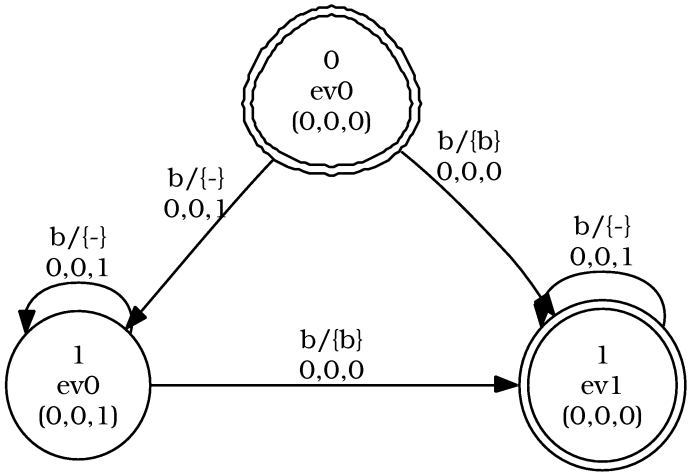
The prospect of accepting another /b/ as input from the nodes at index 1 suggests a generalization. Consider in (220) the arcs that might be used to extend the machine.

(220) Accepting another /b/:



Normalizing the costs at the termini of the arcs reveals the same set of relative costs as those at index 1. By giving the termini at the ends of the newly created arcs the index 1 we generalize. The resulting machine is given in (221).

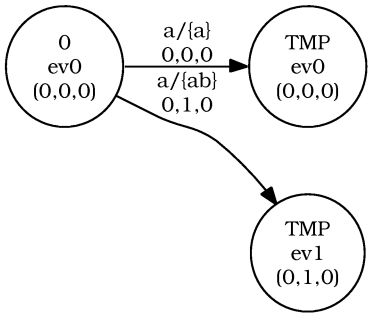
(221)



The machine in (221) currently generates an infinite range of optimal parses by deleting all but one /b/ when fed any input sequence consisting of nothing but /b/'s.

Next we ask what happens if an /a/ is accepted from the start state. In (222) I give the arcs that accept /a/ from ev0.

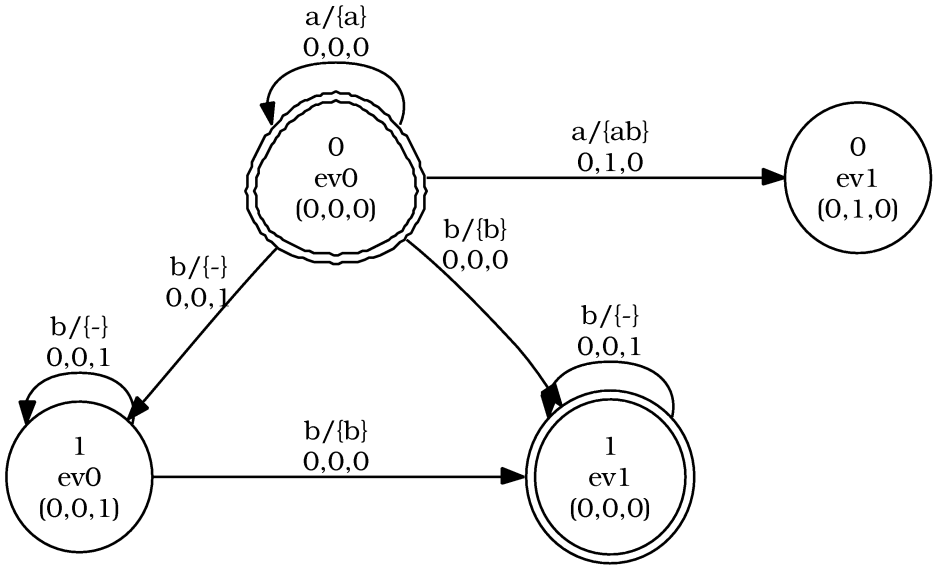
(222) Extension with /a/:



The definition of the *index* function in (207) suggests that the new nodes might be given the index 0 because the nodes currently indexed with 0 express a subset of the costs associated with the newly created nodes. For this indexation to go through it must be the case that with the assignment of index 0 to the new nodes, the arcs already in the machine originating at index 0 are still correct. That is, the addition of a new node at index 0 must not create new and more harmonic paths leading away from the nodes at index 0 than the ones already discovered.

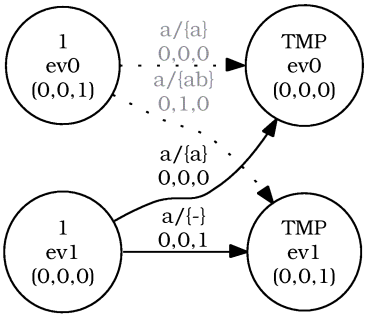
In the case at hand, the /b/ accepting arcs will still be linked to node (0, ev0, (0,0,0)) with the addition of the node (1, ev1, (0,0,1)), thus the generalization is legitimate. The resulting machine is given in (223).

(223) Generalization on /a/ at index 0:



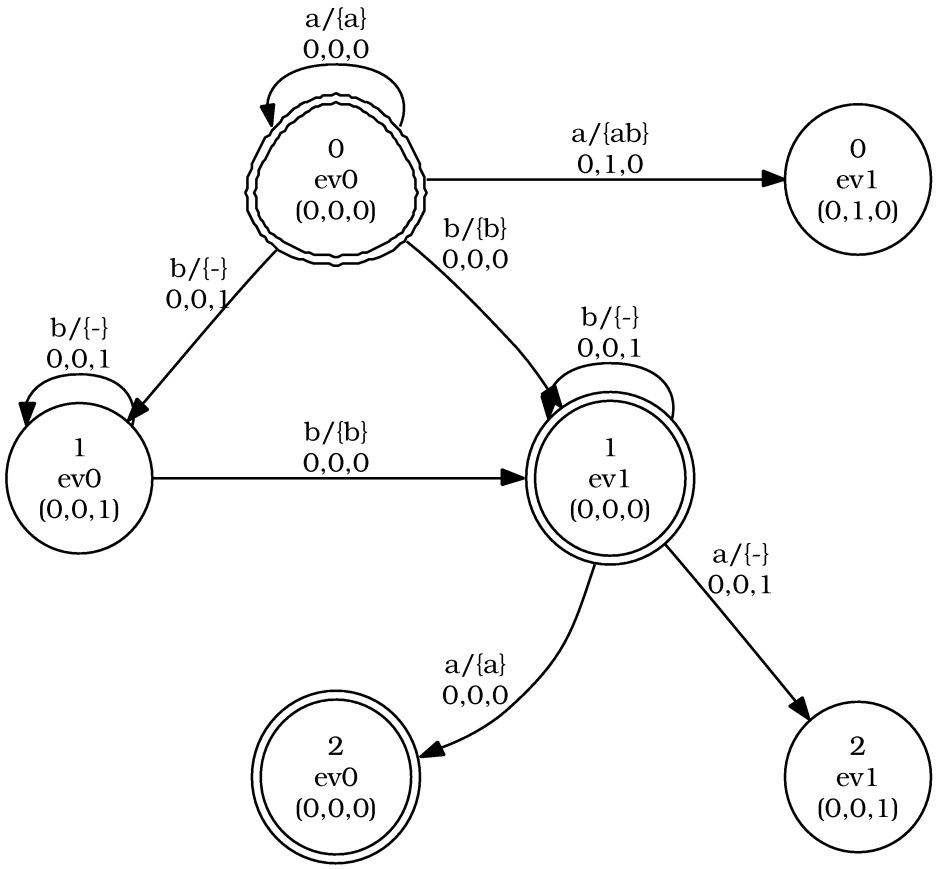
Turning to the acceptance of an input of /a/ at index 1 gives us the arcs in (224) to consider.

(224) Extending the machine with an /a/:



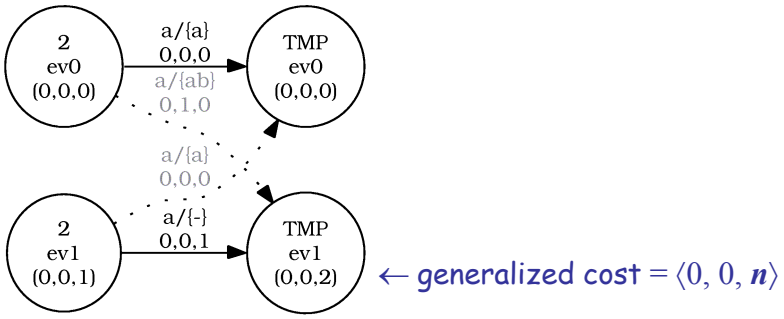
This set of relative costs is not already present in the machine, so a new index is created and arcs are added to the machine. The result is given in (225).

(225)



When the check is made to see which arcs optimally accept an input of /a/ at index 2, a pattern emerges that can be generalized. Consider in (226) the arcs that accept /a/.

(226)

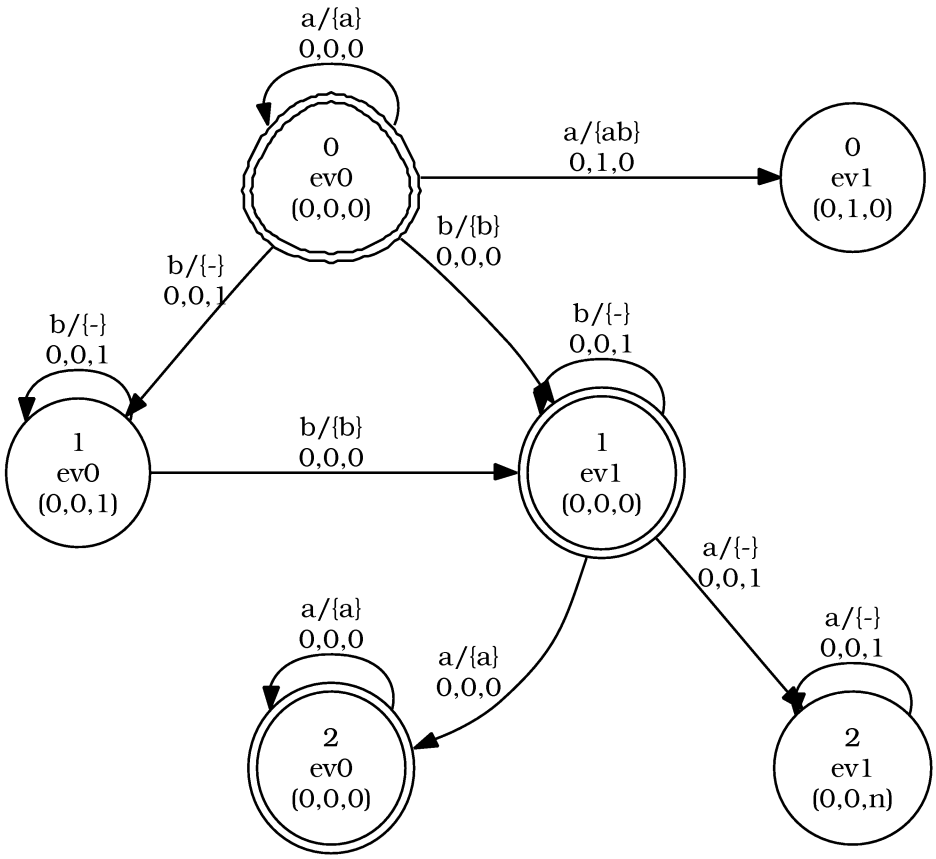


When checking the nodes already in the machine, we don't find the set of relative costs ⟨0,0,0⟩ for ev0 and ⟨0,0,2⟩ for ev1 but we do find the highly similar costs ⟨0,0,0⟩ and

⟨0,0,1⟩ at index 2. When such a similarity exists an attempt can be made at generalization. Both ⟨0,01⟩ and ⟨0,0,2⟩ for ev1 can be seen as instances of ⟨0,0,***n***⟩ for ***n*** ≥ 1. If we replace ⟨0,0,1⟩ and ⟨0,0,2⟩ with ⟨0,0,***n***⟩ then the new nodes can be indexed with 2.

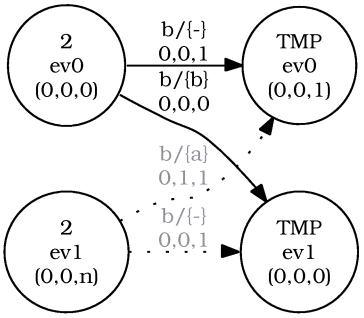
The generalization step is licit for the nodes at a given index just in case the exact value of ***n*** (provided that it's greater than one) never matters in determining which arcs are optimal in accepting an input segment from that index. Furthermore, for the generalization step to be licit for the whole machine, every daughter of a node with a variable must inherit the variable and the generalization must be licit at each index where the variable has been inherited. In (227) I give the machine resulting from this generalization.

(227)



For the generalization with the variable to have been legitimate at index 2, it must be the case that when /b/ is accepted at index 2 the value of ***n*** has no effect on which arcs optimally accept /b/. Consider in (228) the arcs that could be taken with /b/ from index 2.

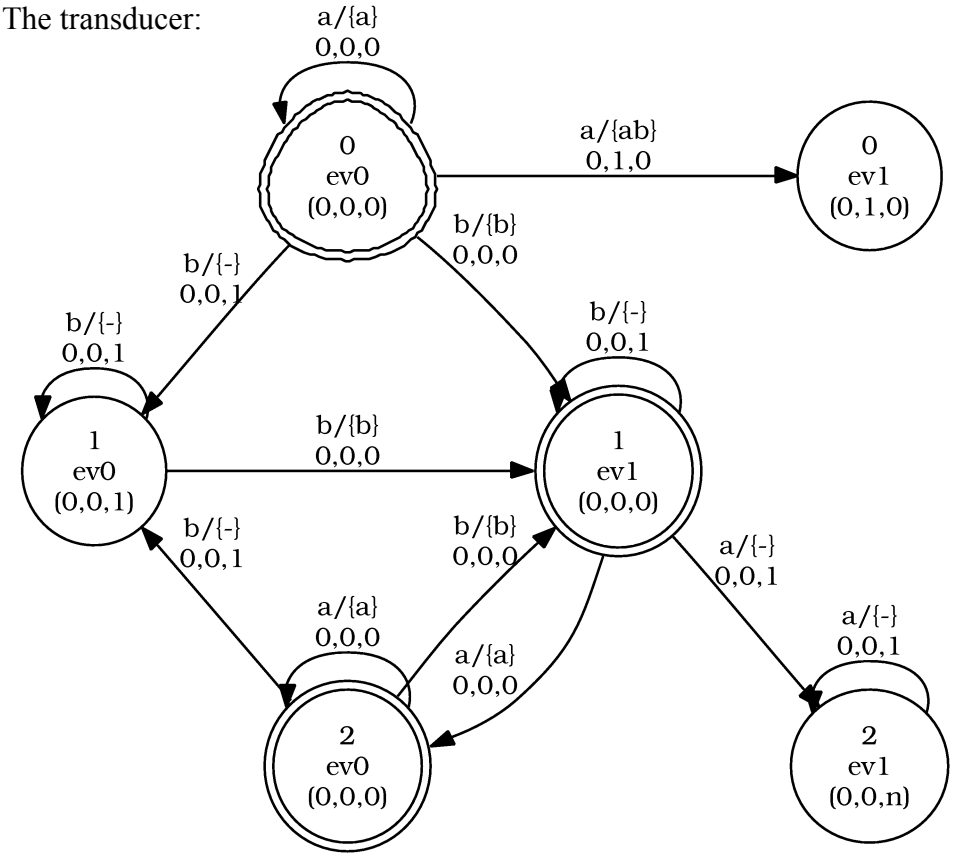
(228) Accepting /b/ at index 2:



In (228) it's clearly the case that for all values of ***n*** greater than one the same set of arcs will be used to accept /b/. Since no locally optimal arcs originate at (2, ev1, (0, 0, n)), no nodes inherit the variable. If some nodes had inherited the variable we would have had to repeat the same check again to ensure that the variable introduction was licit.

The nodes at index 1 already express the set of relative costs annotating the termini of the arcs in (228). By indexing the new arcs with 1, the input /b/ is accepted from index 2 and the machine is completed. In (229) I give the completed transducer.

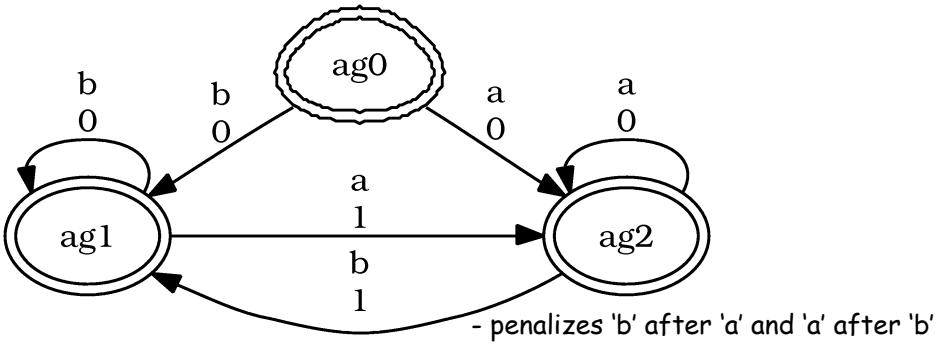
(229) The transducer:



7.5 Majority rule

The generalizations that are allowed by normalizing costs and introducing variables capitalize on recurring patterns across optimizations. These generalization work by re-expressing the infinite set of possible absolute costs as finitely many patterns of relevant dissimilarity. There are, however, OT grammars that cannot be rendered finite in this way because they do not have finitely many relevant patterns of dissimilarity. In this section I will show how the OTCA fails when it encounters such patterns. For this scenario I'll add the constraint given in (230) to the grammar.

(230) AGREE: adjacent segments must agree on the ±*cons* feature (Lombardi 1996, 1999)



This isn't a terribly realistic constraint on vowel and consonant cooccurrence but it will suffice to provide a nice simple illustration of the mechanics of the system. Regardless of the merits of this particular constraint, the patterns predicted by constraints of this type must be taken into account because constraints like (230) are often used in OT analyses of agreement, assimilation, harmony and other similar phenomena.

If the AGREE constraint in (230) dominates INDENT(CONS) the following patterns will emerge. Because AGREE is dominant, every surface string will consist either solely of [a]'s or solely of [b]'s. The deciding factor will be whether there were more instances of /a/ or /b/ in the input string. To minimize IDENT violations, whichever segment type is relatively less prevalent will assimilate to the more prevalent segment type.

Lombardi (1996, 1999) argues that this sort of dependency on relative frequencies of segment types in the input is unheard of in the phonology of natural languages and offers a couple of proposals for how it might be avoided. Baković (1999, 2000) dubs this sort of counting prediction "majority rule" and proposes yet another strategy for it's elimination. Lombardi suggests that one way to avoid the majority rule prediction would be to modify

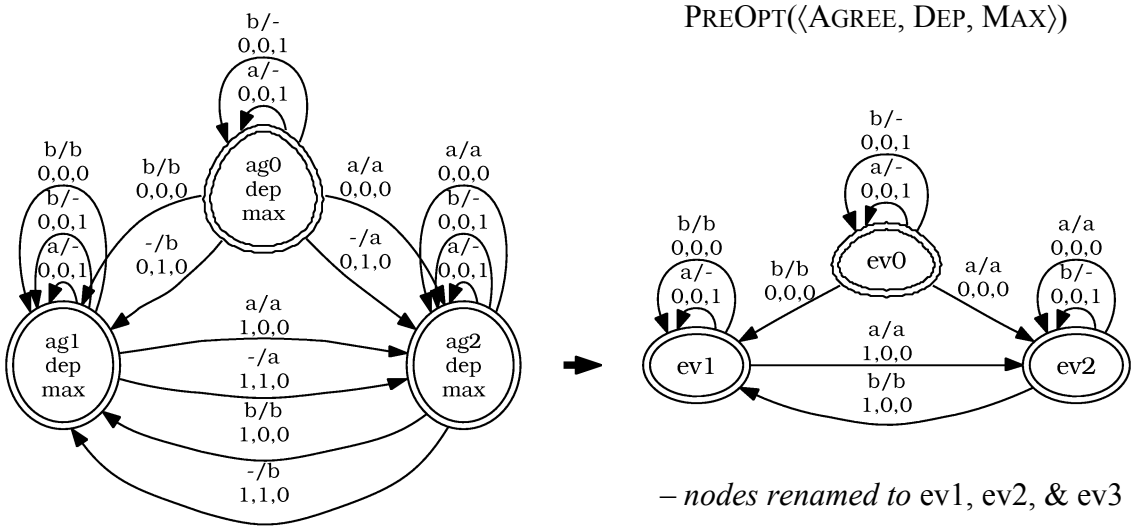
faithfulness constraints so that sequences of featural changes get exactly one faithfulness violation regardless of how many features are changed. Baković argues that this proposal gives rise to an altogether different problematic typological prediction in which features are especially vulnerable to change in the vicinity of other segments that have surfaced unfaithfully. Baković calls this new problem "contrast dependency" and argues that it is just as bad as majority rule.

Another way that the majority rule prediction might be avoided is through the use of faithfulness constraints that, unlike typical IDENT constraints, are sensitive to specific feature values. This is, in fact, exactly the way that Prince and Smolensky's (1993) PARSE and FILL faithfulness constraints are applied to features (Kirchner 1993, Itô, Mester and Padgett 1995). In Correspondence theory (McCarthy and Prince 1995) this same effect can be achieved with Dep and Max constraints that refer to specific features (Lombardi 1995, Walker 1997). Alternatively, to avoid the need for a mechanism to keep the features from floating around (Itô, Mester and Padgett 1995), Baković suggests that McCarthy and Prince's (1995) feature-value-specific versions of IDENT constraints could be used.

Baković own proposed remedy for majority rule is that local constraint conjunction (Smolensky 1993, 1995, 1997) be used to combine markedness and faithfulness constraints (Lubowicz 1998) to create constraints that are violated by unfaithful input/output-mappings that increase markedness. Given Smolensky's (1993) basic tenet that it's universally the case that $(C_1\&C_2) \gg C_1, C_2$, it follows that the constraint against assimilating to the marked feature value will universally dominate the basic constraint against assimilation (IDENT).

Lombardi's and Baković's proposals can remedy the majority rule problem for the interaction of IDENT and AGREE constraints. But alas, as we know from Frank and Satta's (1998) work, counting dependencies arise from the fundamentally comparative nature of OT and aren't the fault of a few ill behaved constraints. Consider, for instance, the rather perverse grammar in (231) that enforces agreement by deleting offending segments.

(231) ⟨AGREE, DEP, MAX⟩$^{\boxtimes}$



In this grammar surface strings will consist entirely of [a]'s or [b]'s because AGREE is dominant. If the vowel and consonant specific MAX constraints are ranked below the general version of MAX then the determination of whether the surface string is all [a]'s or all [b]'s will be made on the basis of which segment type is more prevalent in the input – MAX will demand that the less prevalent type be the one that's deleted to satisfy AGREE.

This is majority rule all over again and, due to the unbounded counting dependency, another instance of an OT grammar that doesn't define a rational i/o-relation. It is worth noting here that though Frank and Satta's (1998), Karttunen's (1998), and Gerdemann

and Van Noord's (2000) modified versions of OT eliminate the possibility of unbounded counting dependencies they actually don't solve the majority rule problem. In these OT variants there is an upper bound on the number of violations to which the grammar is sensitive (or for Gerdemann and Van Noord an upper bound on the number of violations that can be permuted to eliminate suboptimal competitors). If the bound in these cases is set to ***n*** then it will still be possible to generate majority rule phenomena when the majority in question is smaller than ***n***.

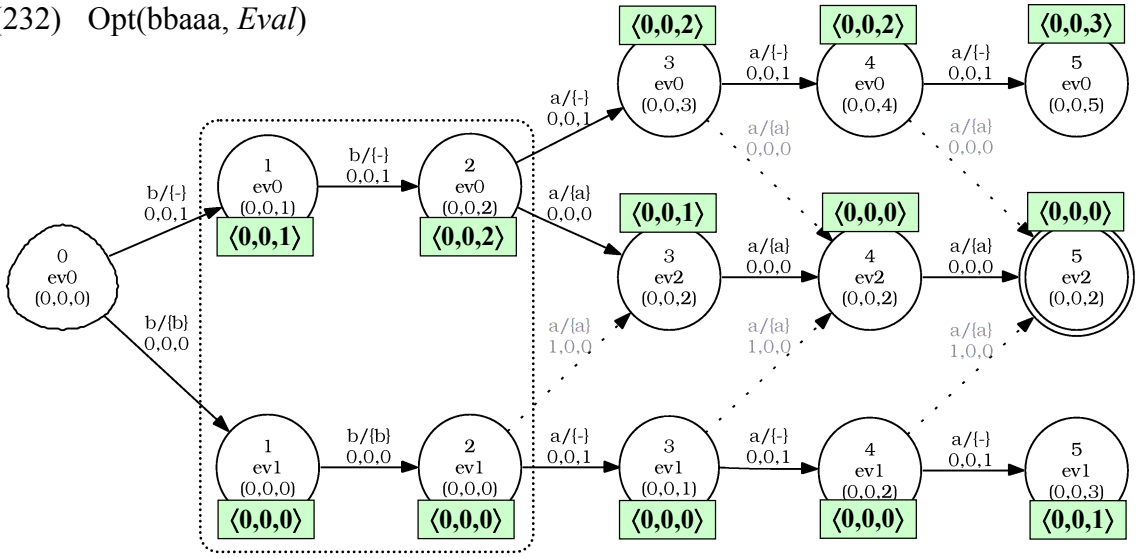
The only proposed variant of OT to date that completely eliminates the majority rule problem is Eisner's (2002) directional OT variant. In Eisner's model every constraint is evaluated directionally in such a way that violations closer to one specified edge of the form are strictly worse than any violations further from that edge. In this system majority rule can't arise because all decisions about optimality are strictly local. Eisner's proposal also has the advantage that, as with Karttunen's and Gerdemann and Van Noord's models, his OT grammars can be turned into transducers and thus used for recognition.

Eisner's proposed directional variant of Optimality Theory does, however, have one major drawback. Specifically, it predicts that directional preferences in phonological phenomena should be ubiquitous rather than a relative rarity. While not pathological, this typological prediction seems to be a poor fit with observation. Furthermore, Wilson (2004) points out that when Eisner's directional evaluation is used to generate directional harmony systems, it predicts an anomalous harmony pattern in which harmony occurs only if the harmonizing feature can spread all the way to the edge of the word.

What I would like to propose here is that rather than attempting to rid the theory of constraints that can interact to generate majority rule and rather than changing the nature of optimization so that majority rule cannot arise, we can instead eliminate majority rule as a prediction on the basis of the fact that grammars evidencing it do not define rational i/o-relations. That is, because grammars with majority rule can't be described as finite state transducers, if part of the phonological acquisition process involves extracting regular (i.e. finite-state) generalizations about the phonology, then this requirement can act as a sort of filter that blocks the grammars that generate majority rule. Below, I'll show just how the OTCA fails when it counters a grammar defining an i/o-relation that is not rational.

Consider in (232) the effect of optimizing /babab/ with the grammar in (231). I've indicated the relative costs of the most harmonic paths to each node in a box next to it.

(232) Opt(bbaaa, *Eval*)



If the transducer were being constructed incrementally, then, upon encountering the boxed portion of (232), the algorithm would attempt introduce a variable to capture the similarity between the relative costs of the nodes at indices 1 and 2. Indeed, if more /b/'s

are encountered at this point the disparity between the relative cost of ev0 and ev1 can grow without bound. Therefore, unless variables can be introduced, there will be no way to recast the range of possible relative costs as a finite set.

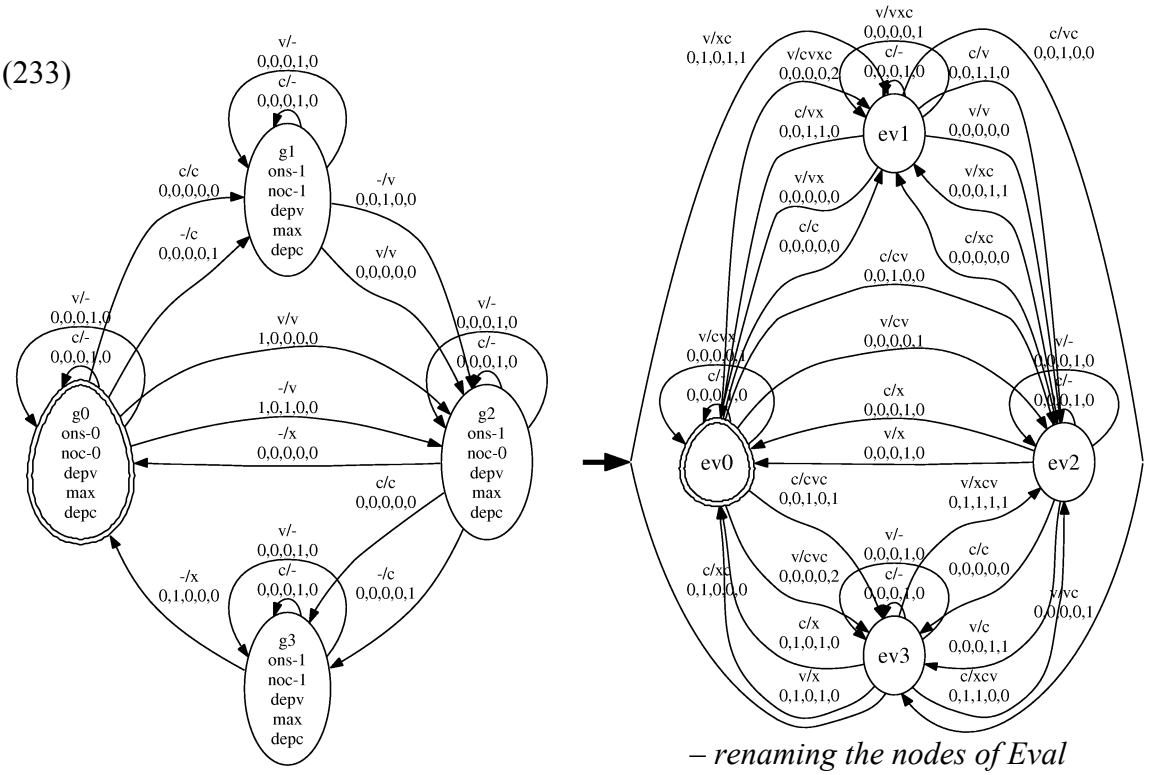
If a variable is introduced at this point, however, its presence will obliterate any ability to make relevant distinctions among the paths in the machine downstream from the node where the variable is added. This is so because, unlike the example in §7.4, the variable would be introduced on a "live" path and once the daughters of the node with the variable inherited it from their mothers it would become impossible to adjudicate between the various paths.

In practical terms, variable introduction fails in the OTCA because, at the very next index, the value of ***n*** is critical in determining which arcs should be added to the machine. Considering the bigger picture, variable introduction will always fail in grammars that generate majority rule precisely because it isn't possible to abstract away from the exact value of the disparity in the cost of the various paths. This is so because it's the number of underlying segments bearing a particular feature that determines which segment-type is in the majority.

7.6 Recognition

Construction of transducers for the languages of the basic CV syllable theory is fairly straightforward. As with the cases presented in §7.3 and §7.4 the first step is the preoptimization of *Eval*. In (233) I present the results of preoptimization for the ranking ONSET ≫ NOCODA ≫ DEPV ≫ MAX ≫ DEPC.

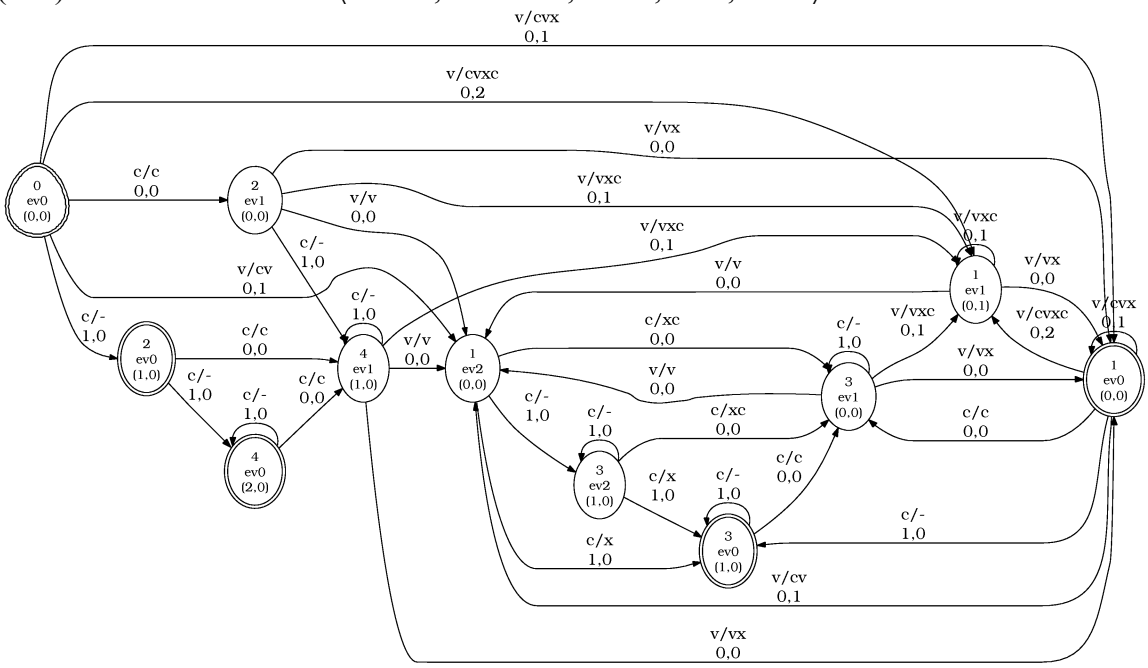
(233)



In (233) preoptimization creates a machine with quite a few more arcs than *Eval* before preoptimization. This happens because every node in *Eval* can be reached from every other node by adding some sequence of epenthetic segments. Thus, because there are two input symbols and four nodes, there are eight arcs originating at each node. In (233) and in (234) below, I'll omit the set-brackets around the output strings on the arc labels because all of the sets are singletons.

Constructing a transducer from the preoptimized machine in (233) proceeds exactly as in §7.3 and §7.4. In (234) I present the results of the OTCA for the input alphabet {c, v} and the machine in (233).

(234) The transducer for ⟨ONSET, NOCODA, DEPV, MAX, DEPC⟩:



In (234) I've removed dead ends and the arcs that lead to them to make the graph easier to read. I have also omitted the first three coordinates of each cost vector because they were zeroes in all cases. This comes as no surprise as the only constraints that will ever be violated in an optimal i/o-mapping are the two lowest ranked, MAX and DEPC.

Using the transducer in (234) for recognition is relatively trivial. By intersecting an output string with the machine, it is possible to derive the set of input strings that are mapped to that output under the grammar used to construct the transducer. The only novel point is that, because the outputs labeling the arcs are strings, we'll need a new definition of *M*-intersection to allow an output string to be intersected with a machine like the one in (234). This new version of *M*-intersection is given in (235).

(235) ***def***: *M*-intersection

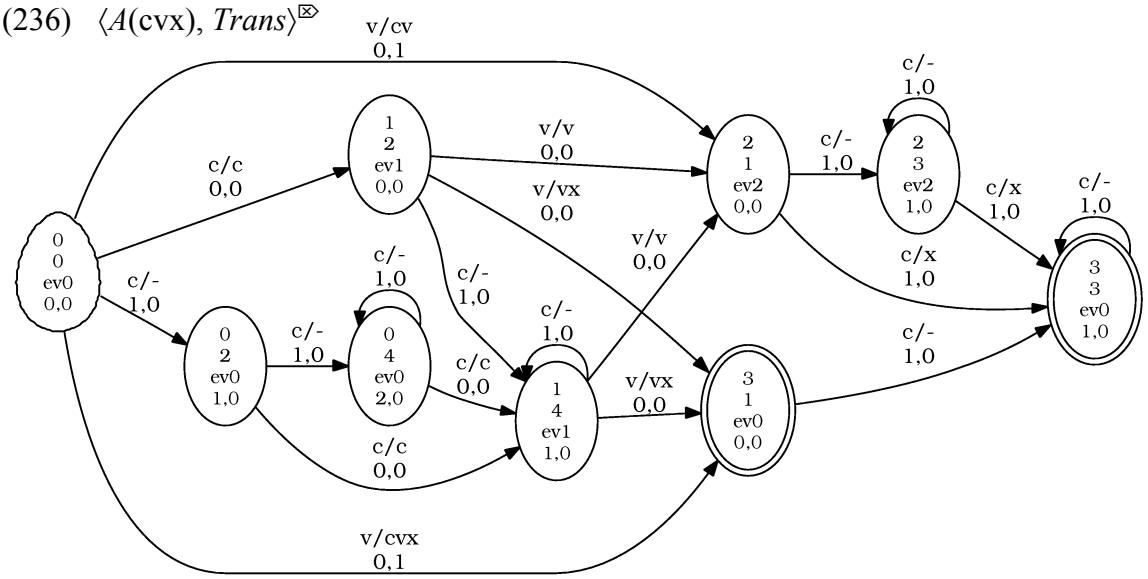
$\langle M_1, M_2 \rangle^{\otimes} = M_3$ for $M_1 = (Q_1, \Sigma_1, \delta_1, S_1, F_1)$ and $M_2 = (Q_2, \Sigma_2, \delta_2, S_2, F_2)$,

$M_3 = (Q_1 \times Q_2, \{\Sigma_1 \cup \Sigma_2\}, \delta, S_1 \times S_2, F_1 \times F_2)$ where

$\delta = \{(qx, i, o, v, ry) \mid$ there's a path from q to r in M_1 accepting o and $(x, i, o, w, y) \in \delta_2\}$

By intersect the linear acceptor for an output string with the transducer in (234), which I'll simply call *Trans*, we derive the set of inputs that yield that output string. In (236) I give the intersection of the linear acceptor for a simple CV-syllable with *Trans*.

(236) $\langle A(\text{cvx}), \textit{Trans} \rangle^{\otimes}$



The machine in (236) defines an infinite set of i/o-mappings that are optimal under the ranking ONSET ≫ NOCODA ≫ DEPV ≫ MAX ≫ DEPC. Examining the paths through this machine reveals that every string of /c*vc*/ will be mapped to the output [cv].

Transducers like (236) have the nice property that if an attempt is made to intersect them with an output string that can't be generated by the grammar, the result is a machine that contains no complete paths from the start state to a final state. For instance, there are

no arcs originating at the start state of (236) that are labeled with output strings beginning with a vowel. Because ONSET is dominant in the grammar, such strings simply cannot be generated.

7.7 Learning and generative power

I have proposed here an algorithm that generates a transducers that are equivalent to optimization for ranked sets of constraints just in case the relation that's determined by optimization is rational. Transducer construction can be done only once 'off-line' and does not need to be repeated for every input string. This method will yield a parser that is quite fast. Furthermore, since the grammar is represented with a transducer it will be reversible, thereby enabling its use as a perception grammar as well as a production grammar.

This approach offers a novel take on the issue of generative power. Rather than placing some overarching restriction on optimization itself to keep the patterns generated strictly regular, we could suppose that transducer construction was part of the process of learning phonological grammars. Taking this approach would constrain the languages that the learner actually learns to the regular subset of the set of languages generated under constraint permutation.

8 Conclusions

In this dissertation I have implemented and assessed various properties of a finite-state model of Optimality Theory. Following a growing body of work including that of Ellison (1994, 1995), Walther (1996), Eisner (1997a, 1997b, 1997c, 2000, 2002), Albro (1998a, 1998b, 2000, 2003), Karttunen (1998), and Gerdemann and Van Noord (2000), I model the constraints of Optimality Theory as finite-state machines. There are, however, four main differences between the model proposed here and these previous proposals.

In the model presented here optimization is left intact and not modified by placing upper bounds on the number of violations that can distinguish competing candidates or by adding directionality to evaluation so that violations near one edge are strictly worse than violations further from that edge.

In the model presented here constraints can be multiply violated by an individual candidate thereby distinguishing different levels of violation rather than simply making a binary distinction between candidates that are violators and candidates that are not.

In the model presented here no arbitrary upper bounds are placed on the candidate set vis a vis its size or the amount of epenthesis that can occur in a particular candidate.

These three points affect the empirical predictions and coverage of the theory. By leaving optimization intact, allowing multiple violation, and allowing an infinite candidate set, the model presented here is much closer to Optimality Theory as usually practiced by phonologists. The fourth point of departure between the model presented here and other finite-state models of OT is not a deep property of the model with empirical ramifications but rather a property of the implementation. Nonetheless, it is this fourth point of difference

from the previous models that sets the stage for almost all of the results presented in this dissertation.

In the FSOT-model that I formulate here all of the constraints of the grammar are combined into a single evaluation function (a finite state machine) that is used in one fell swoop to perform optimization. This is unlike most other finite state models of OT where evaluation is carried out one constraint at a time using a cascade of finite-state transducers. This aspect of the model has many felicitous consequences.

8.1 Contenders and the utility of monolithic *Eval*

The strategy of performing evaluation one constraint at a time with a cascade of transducers is motivated by concerns about the potential size of the representation of the evaluator obtained by combining all of the constraints of the grammar into a single finite state machine. Indeed this is a valid concern given the potential for explosive growth when machines are intersected. I have, however, argued here that this fear is not warranted because the number of states in the evaluator obtained by intersecting of all constraints cannot exceed the number of unique phonological environments to which the grammar is sensitive. Though this number may be quite large it doesn't seem that it will grow so large as to make the machines impossible to work with.

Intersecting a set of constraints to produce a single machine creates a finite and relatively concise representation of the entire class of grammars defined by permutation of those constraints. Given such a machine it's possible to take an input and find the set of outputs that can emerge as optimal under any ranking of the constraints (the contenders)

without the necessity of performing factorially many calculations. Such a feat is obviously impossible in any model that requires building a different cascade of evaluators to find the optimal output for each particular ranking of the constraints.

8.2 Preoptimization and efficient generation

Having a single finite representation of the evaluation function *Eval* also makes it possible to examine the structure of *Eval* itself to detect and eliminate (infinitely many) suboptimal parses before any input us considered. *Eval* only needs to be preoptimized one time to drastically reduce the complexity of all subsequent optimization tasks. Moreover, preoptimization of *Eval* yields a machine that can be used to evaluate candidates in linear time and, if every i/o-mapping that adds structure to the output is penalized by some constraint then the candidate set is always guaranteed to be finite. This is quite promising given the standard OT assumption that such constraints (e.g. DEP or even Zoll's (1993) *STRUC) are present, albeit possibly lowly ranked, in all phonological grammars.

In a sense, preoptimization factors epenthesis out of the system, leaving epenthetic material as an option only when it increases the harmony of the output form. This idea has been around for some time. Tesar (1995a) restricts the amount of epenthetic material that is allowed to occur in candidates to less than a syllable arguing that if it were the case that an entire epenthetic syllable could increase harmony then unbounded syllable epenthesis would be predicted. Explicit upper bounds on the amount of epenthesis allowed in candidates have been either proposed or discussed by many researchers (Hammond 1997, Karttunen 1998, Walther 1996, and Lubowicz 2003, to name a few). What is new in the

proposal here is that rather than picking a particular quantity of epenthesis and simply disallowing candidates that exceed that limit, epenthetic material can be restricted in a principled way to only those additions that actually have some chance of increasing the harmony of a candidate. The question 'what is the longest sequence of epenthetic material that can ever occur in an optimal candidate' can be answered for a given class of grammars (set of constraints) by intersecting the constraints and doing preoptimization.

The most relevant ramification of the fact that preoptimization factors epenthesis out of the system is that it creates an evaluation function that when intersected with an input string produces a directed acyclic graph. This means that optimization tasks can be carried out in linear time and should put to rest all qualms about the "computational feasibility" of Optimality Theory.

Even if it is possible to use a principled method to obtain a finite candidate set, the quest for a trick that makes the candidate set finite is misguided. Far more promising is the possibility of generating the set of contenders which are not only finite but also exactly the relevant candidates.

8.3 Transducers and recognition

In chapter seven I showed how transducers can be constructed that perform the same mapping as optimization with a ranked set of constraints. Constructing a transducer is more costly in terms of computation than preoptimization, but it creates a machine that can perform generation tasks even faster than optimization with preoptimized grammars.

The most relevant property of the transducers, however, is the fact that they can be inverted to do recognition. I haven't devoted much space to the recognition problem in this work because once the transducers have been created it is a relatively trivial matter to run them backwards to do recognition. Of course, this presupposes that the ranking of the constraints is already known (it was used to build the transducer). A more interesting and realistic problem arises if we consider the task learning from outputs rather than i/o-pairs. I'll return to this issue in §8.6.

8.4 The finite state restriction

The only significant departure in the finite-state OT model developed in this work from OT as it is practiced by phonologists at large is the imposition of the restriction that constraints must be able to be represented as finite-state machines. This restriction buys us a tremendous amount. It allows optimization to be very naturally characterized graph-theoretically as a shortest paths problem, it allows efficient generation, it allows us to find contenders, and for many rankings it makes possible the construction of transducers that can be inverted to do recognition.

There are, however, two domains where the finite-state restriction is a poor fit with OT analyses that appear throughout the literature. These two domains are alignment and reduplication. The ill-suitedness of finite state methods to some constraints on alignment may actually turn out to be an asset to the theory, but the failure of finite-state models to readily handle reduplication presents an open problem.

As Eisner (1997b) and Bíró (2004) have discussed, many alignment constraints cannot be modeled with finite-state machines. For instance ALL-SYLLABLES-LEFT, which assigns to each syllable in the word a number of violations dependent on its distance from the left edge, cannot be modeled because the violations it assesses grows as a quadratic function of the length of the word. Interestingly, such constraints have recently come under attack on independent empirical grounds. Eisner (1997c) observes that constraints of this type make odd and unattested typological predictions like the centering of floating tones in words. McCarthy (2002) also argues that this type of constraint should be eliminated from the theory citing a variety of problematic empirical predictions that arise with these so called "gradient" constraints.

The other domain to which finite state methods are ill-suited is more of a problem. Reduplication, with its apparently unbounded crossing dependencies, presents a canonical instance of a pattern that is strictly more complex than that which can be generated using finite-state machines. There are several plausible responses to this state of affairs that do not involve abandoning the finite state approach to phonology.

One tack that might be taken here would be to relegate to morphology some of the computation in reduplication. It's well known that there are morpho-syntactic patterns that are well beyond those describable with finite state methods, so calling on the morphology to play a role in candidate generation or base/reduplicant coindexation might factor out some of the complexity in reduplication. Such a strategy would require that the cases of over-application and under-application be dealt with carefully because they do seem to be truly part of the phonology and not the morphology.

Another approach to the complexity of reduplication is laid out by Albro (2003). He suggests that a non-finite-state module can be incorporated into an otherwise finite-state grammar expressly for the purposes of handling reduplication.

8.5 Representations

Unlike some previous computational models of Optimality Theory which modify the nature of optimization to achieve desirable computational properties, I've tried to stay as true as possible to OT as it is usually practiced. I have, however, made representational choices in this work that aren't totally standard.

For the most part the decision to formulate the constraints over segments rather than features, autosegmental representations (cf. Heiberg 1999), or tiers (cf. Eisner 1997b and Albro 1998) is motivated by simplicity and is not an essential property of the model that I've developed in this work. These representations have the advantage that the constraints are usually quite small (only one or two states) and the input and output strings can be read directly off of the paths through the machines.

A perfectly viable alternative would be to use features on the arcs of the machines rather than segments. Indeed, when considering languages with large numbers of phonemes it may become quite cumbersome to have an individual arc for every pair of segments that show a particular featural change in the input/output mapping.

Whether this is a trivial or significant aspect of the implementation hangs on what the featural/autosegmental/tier-based representations buy us in terms of codifying possibly universal properties of phonological grammars like locality, strict-adjacency, and natural class behavior. It is certainly the case that any restriction that autosegmental or tier-based

representations impose on the system as a whole can also be captured via meta-constraints on the statement of segmental constraints. This alternative may, however, come with a serious loss of elegance in the statement of the generalizations.

On the other hand, grammars built from segmental constraints seem, at least for the simple cases, to be much easier to work with. This suggests the possibility that, while autosegmental representations make some generalizations more elegant, the off-loading of the generalizations into the structure of the representations themselves may ultimately come at a high price in terms of our ability to manipulate grammars stated over those representations.

Another aspect of the model developed here that should receive further scrutiny is the representation of OT grammars as preoptimized evaluators or as transducers. There is some tension between ranked sets of universal constraints on one hand and finite state machines on the other. The former is far more elegant in terms of describing the range of possible languages and the relatedness of languages to one another while the latter is far more transparent in terms of describing the input/output mapping defined by a particular grammar. One might ask if one of these representations of phonological grammars was more "right" or more illuminating than the other.

Considering my algorithm for turning ranked sets of constraints into finite state transducers that map inputs directly to optimal outputs, it might seem as if I've taken the OT out of Optimality Theory. That is, one might think that the low-level description of the computation of optimality is somehow closer to the "true" phonology and should therefore obviate the need for constraints and rankings. Such a conclusion would be mistaken, first

and foremost, because ranked constraints provide elegant descriptions of the relations that are defined by phonological grammars and of the relationships among various grammars (languages), while the algorithms that I present here represent merely one method for computing these relations.

Even if it's the case that the algorithms make substantive predictions about human phonology (e.g. commitment to finite state computation eliminates irrational i/o relations), there's no sense in which the utility of the constraint-based representation of the grammars is diminished. After presenting algorithms for creating transducers that define the same i/o-relations as sequences of phonological rules, Kaplan and Kay (1994) stress a similar point.

> "The common data structures that our programs manipulate are clearly states, transitions, labels, and label pairs--the building blocks of finite automata and transducers. But many of our initial mistakes and failures arose from attempting also to think in terms of these objects. The automata required to implement even the simplest examples are large and involve considerable subtlety for their construction. To view them from the perspective of states and transitions is much like predicting weather patterns by studying the movements of atoms and molecules or inverting a matrix with a Turing machine." – Kaplan and Kay 1994:376.

Kaplan and Kay stress the need for tools that allow high-level reasoning about grammars and languages. This is a general point that holds for all linguistic models and, while a formal idea of how the computation for a particular model might be carried out can aid in understanding that model, we still need an appropriate high-level description of the range of possible grammars in that model.

8.6 The future

The finite state model of Optimality Theory presented in this work is somewhat preliminary in several respects. Though the results for the toy grammars that I have

considered are promising, the only way to convincingly address the issue of scalability in real-world phonology will be to apply this system to a large range of phonological problems. There are numerous details of the implementation and representations that might be tweaked as needed when addressing new problems, but the basic strategy of building a single evaluator to allow generation of contenders, preoptimization, and transducer construction should be applicable to a wide range of problems.

The significance of the model and algorithms presented here does not lie in the deepening of our understanding of the basic CV syllable theory but rather in the possibility that the tools developed here can be applied to Optimality Theory as it is generally practiced to deepen our understanding of the interactions among the various components of the theory and of the predictions made by our phonological models.

One of the most promising areas for further research in this model is the possibility of suspending the assumption that the constraint set is fixed and universal and instead exploring methods of constraint construction and discovery. Not only does the finite-state assumption for constrains, provide a natural complexity metric and a scheme for formulating new constraints, but the representation of the entire grammar as a single evaluating function makes it possible to ask how the addition of a new constraint affects the complexity of the whole grammar.

REFERENCES

Albro, Daniel M. 1998a. *Evaluation, implementation, and extension of Primitive Optimality Theory*. MA Thesis, UCLA.

Albro, Daniel M. 1998b. Three formal extensions to primitive optimality theory. *Proceedings of the 38th ACL*.

Albro, Daniel M. 2000. Taking Primitive Optimality Theory beyond the finite state. In *Proceedings of the 18th International Conference on Computational Linguistics, Special Interest Group in Computational Phonology*.

Albro, Daniel M. 2003. *A Large-Scale, Computerized Phonological Analysis of Malagasy*. Talk given at the 2003 meeting of the LSA, Atlanta, GA.

Baković, Eric. 1999. *Assimilation to the unmarked*. ROA-340.

Baković, Eric. 2000. *Harmony Dominance and Control*. PhD Dissertation, Rutgers University.

Beckman, Jill. 1995. Shona height harmony: Markedness and Positional Identity. In Jill N. Beckman, Laura Walsh Dickey and Suzanne Urbanczyk (eds.), *Papers in Optimality Theory, University of Massachusetts Occasional Papers 18*, Amherst, Massachusetts. 53-76.

Beckman, Jill. 1998. *Positional Faithfulness*. PhD Dissertation, University of Massachusetts, Amherst.

Bíró, Tamás. 2003. Quadratic Alignment Constraints and Finite State Optimality Theory. In *Proceedings of the Workshop on Finite-State Methods in Natural Language Processing (FSMNLP), 10th Conference of the European Chapter of the ACL*, Budapest, Hungary.

Boersma, Paul. 1997. How we learn variation, optionality, and probability. *Proceedings of the Institute of Phonetic Sciences of the University of Amsterdam* 21:43–58.

Boersma, Paul. 1998. *Functional Phonology. Formalizing the interactions between articulatory and perceptual drives.* Doctoral dissertation, University of Amsterdam. The Hague: Holland Academic Graphics.

Boersma, Paul. 2000. "Learning a grammar in functional phonology". In J. Dekkers, F. van der Leeuw, and J. van de Weijer (eds.), *Optimality Theory: Phonology, Syntax, and Acquisition.* 465-523. Oxford University Press.

Boersma, Paul and Bruce Hayes. 2001. Empirical tests of the Gradual Learning Algorithm. *Linguistic Inquiry* 32: 45–86.

Browman, Catherine. and Goldstein, Louis. 1989. Articulatory gestures as phonological units. *Phonology* 6, 201-251.

Casali, Roderic. 1997. Vowel Elision in Hiatus Contexts: Which Vowel Goes? *Language* 73: 493-533.

Cole, Jennifer and Charles Kisseberth. 1995. An optimal domains theory of vowel harmony. *FLSM V*. Urbana: University of Illinois. 101–114.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. *Introduction to Algorithms, Second Edition*. Cambridge: MIT Press.

Crowhurst, Megan J., and Mark Hewitt. 1997. *Boolean operations and constraint interactions in Optimality Theory*. ROA-229-1197.

Dijkstra, Edsger W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik,* 1: 269-271.

Eisner, Jason. 1997a. Decomposing FootForm: Primitive constraints in OT. In *Proceedings of Student Conference in Linguistics, SCLI VIII*, MIT Working Papers in Linguistics.

Eisner, Jason. 1997b. *Efficient generation in primitive Optimality Theory*. Ms., University of Pennsylvania. ROA-206-0797.

Eisner, Jason. 1997c. *What constraints should OT allow?* Ms., University of Pennsylvania. ROA 204-0797.

Eisner, Jason. 1999. *Doing OT in a straightjacket*. Talk presented at UCLA.

Eisner, Jason. 2000. Directional constraint evaluation in Optimality Theory. In *Proceedings of COLING*.

Eisner, Jason. 2002. Comprehension and Compilation in Optimality Theory. In *Proceedings of the 40th ACL*.

Ellison, T. Mark. 1994. Phonological derivation in Optimality Theory. In *Proceedings of the Fifteenth International Conference on Computational Linguistics*, 1007-1013. ROA-75-0000.

Ellison, T. Mark. 1995. OT, finite-state representations and procedurality. In *Proceedings of the Conference on Formal Grammar*, Barcelona.

Fosler, Eric. "On Reversing the Generation Process in Optimality Theory," Proceedings of the Association for Computational Linguistics, Santa Cruz, California, 1996.

Frank, Robert, and Giorgio Satta. 1998. Optimality Theory and the generative complexity of constraint violability. *Computational Linguistics*, 24: 307-315.

Gerdemann, Dale, and Gertjan v. Noord. 2000. Approximation and exactness in finite state optimality theory. In Jason Eisner, Lauri Karttunen, and Alain Thriault (eds.), *SIGPHON 2000, Finite State Phonology*.

Gibson, Edward, and Kenneth Wexler. 1994. Triggers. *Linguistic Inquiry* 25(3): 407-454.

Goldsmith, John. 1976. *Autosegmental Phonology*. PhD Dissertation, Massachusetts Institute of Technology. Distributed by Indiana University Linguistics Club, Bloomington, Indiana.

Goldsmith, John. 1990. *Autosegmental and metrical phonology*. Oxford: Blackwell.

Grimshaw, Jane. 1997. Projection, Heads, and Optimality. *Linguistic Inquiry* 28(4): 373-422. ROA-68.

Hammond, Michael. 1995. *Syllable Parsing in English and French*. Ms., University of Arizona. ROA-58.

Hammond, Michael. 1997. *Parsing in OT*. Ms., University of Arizona. ROA-222-1097.

Hayes, Bruce. 1999. *Phonological acquisition in Optimality Theory: The early stages.* Ms. UCLA. ROA-327

Hayes, Bruce. 2000. Gradient Well-formedness in Optimality Theory. In Joost Dekkers, Frank van der Leeuw and Jeroen van de Weijer, eds., *Optimality Theory: Phonology, Syntax, and Acquisition*, Oxford University Press, pp. 88-120.

Hayes, Bruce and Margaret MacEachern. 1998. Quatrain form in English Folk Verse. *Language* 64, 473-507.

Heiberg, Andrea Jeanine. 1999. *Features in Optimality Theory: A Computational Model*. PhD Dissertation, University of Arizona.

Itô, Junko, R. Armin Mester, and Jaye Padgett. 1995. Licensing and Underspecification in Optimality Theory. LI 26, 571-613.

Jäger, Gerhard. 1999. *Optimal syntax and optimal semantics*. Handout for a talk at DIP-colloquium, University of Amsterdam.

Jäger, Gerhard. 2000. "Some Notes on the Formal Properties of Bidirectional Optimality Theory", *Linguistics in Potsdam* **8**: 41-63.

Kaplan, Ronald M., and Martin Kay. 1981. Phonological rules and finite-state transducers. Paper presented at ACL/LSA conference, New York.

Kaplan, Ronald M., and Martin Kay. 1994. Regular models of phonological rule systems. *Computational Linguistics*, 20(3): 331–378.

Karttunen, Lauri. 1998. The proper treatment of optimality theory in computational phonology. In *Finite-state Methods in Natural Language Processing*. 1–12.

Karttunen, Lauri; Koskenniemi, Kimmo; and Kaplan, Ronald M. 1987. A compiler for two-level phonological rules. In Report No. CSLI-87-108, Center for the Study of Language and Information, Stanford University.

Koskenniemi, Kimmo. 1983. Two-level morphology: A general computational model for word-form recognition and production. *Publication No. 11, Department of General Linguistics,* University of Helsinki.

Leben, William R. 1973. *Suprasegmental Phonology*. PhD Dissertation, Massachusetts Institute of Technology.

Lombardi, Linda. 1995. Why Place and Voice are Different: Constraint-specific alternations in Optimality Theory. Ms., University of Maryland, College Park.

Lombardi, Linda. 1996. Positional Faithfulness and Voicing Assimilation in Optimality Theory. Ms., University of Maryland, College Park.

Lombardi, Linda. 1998. Evidence for MaxFeature Constraints from Japanese. *University of Maryland Working Papers in Linguistics* 7, 41-62.

Lombardi, Linda. 1999. Positional faithfulness and voicing assimilation in Optimality Theory. *Natural Language and Linguistic Theory*, 17: 267–302.

Lubowicz, Anna. 2003. *Contrast Preservation in Phonological Mappings*. PhD Dissertation, University of Massachusetts, Amherst.

McCarthy, John J. 1979. *Formal Problems in Semitic Phonology and Morphology*. PhD Dissertation, Massachusetts Institute of Technology. Published 1985 by Garland, New York.

McCarthy, John J. 1986. OCP Effects: Gemination and Antigemination. *Linguistic Inquiry* 17: 207-264.

McCarthy, John J. 2002. *Against gradience*. ROA-510.

McCarthy, John J., and Alan S. Prince. 1993. Generalized alignment. In *Yearbook of Morphology*, pages 79–153. Dordrecht: Kluwer, also: ROA-7.

McCarthy, John J., and Alan S. Prince. 1994. *The emergence of the unmarked: Optimality in prosodic morphology*. Ms., University of Massachusetts, Amherst, and Rutgers University. 249-384.

McCarthy, John J., and Alan S. Prince. 1995. Faithfulness and reduplicative identity. In J. Beckman, S. Urbanczyk, and L. Walsh (eds.), *Papers in Optimality Theory, University of Massachusetts Occasional Papers 18*, Amherst, Massachusetts.

Myers, Scott. 1991. "Persistent Rules," *Linguistic Inquiry* 22, 315-344.

Mester, Armin, and Jaye Padgett. 1994. Directional syllabification in Generalized Alignment. In Merchant, Jason, Jaye Padgett and Rachel Walker (eds.), *Phonology at Santa Cruz 3*, 79-85. Santa Cruz, California.

Prince, Alan S. 2002a. Entailed Ranking Arguments. ROA-500.

Prince, Alan S. 2002b. Arguing Optimality. In Coetzee, Andries, Angela Carpenter and Paul de Lacy (eds.), *Papers in Optimality Theory II*. Amherst, Massachusetts.

Prince, Alan S., and Paul Smolensky. 1993. *Optimality theory: constraint interaction in generative grammar*. To appear, MIT Press. TR-2, Rutgers University Cognitive Science Center.

Prince, Alan and Bruce Tesar. 1999. Learning Phonotactic Distributions. Technical Report RuCCS-TR-54, Rutgers Center for Cognitive Science, Rutgers University, New Brunswick, NJ.

Samek-Lodovici, Vieri, and Alan S. Prince. 1999. *Optima.* ROA-363.

Samek-Lodovici, Vieri, and Alan S. Prince. 2002. *Fundamental Properties of Harmonic Bounding*. RuCCS-TR-71.

Smolensky, Paul. 1993. *Harmony, Markedness, and Phonological Activity*. Paper presented at the Rutgers Optimality Workshop, Rutgers University. ROA-87.

Smolensky, Paul. 1995. *On the structure of the constraint component Con of UG*. ROA-86.

Smolensky, Paul. 1995. On the Internal Structure of the Constraint Component *Con* of UG. Handout of talk given at the University of Arizona.

Smolensky, Paul. 1997. Constraint Interaction in Generative Grammar II: Local Conjunction. Paper presented at the Hopkins Optimality Theory Workshop/Maryland Mayfest 1997, Baltimore, MD.

Spaelti, Philip. 1997. *Dimensions of Variation in Multi-Pattern Reduplication*. PhD Dissertation, University of California, Santa Cruz.

Tesar, Bruce. 1995a. *Computational Optimality Theory*. PhD Dissertation, University of Colorado. ROA-90-0000.

Tesar, Bruce. 1995b. *Computing optimal forms in Optimality Theory: Basic syllabification*. Ms., University of Colorado at Boulder. ROA-52-0295.

Tesar, Bruce. 1996a. Error-driven learning in Optimality Theory via the efficient computation of optimal forms. In *The proceedings of the Workshop on Optimality Theory in Syntax: Is the Best Good Enough?*. Cambridge, MA: MIT Press.

Tesar, Bruce. 1996b. Computing optimal descriptions for Optimality Theory: Grammars with context-free position structures. *Proceedings of the 34th Annual Meeting of the ACL*.

Tesar, Bruce. 1997a. *Multi-recursive constraint demotion*. ROA-197.

Tesar, Bruce. 1997b. An iterative strategy for learning metrical stress in Optimality Theory. In *Proceedings of the Twenty-First Annual Boston University Conference on Language Acquisition*, 615-626.

Tesar, Bruce. 1998. *Robust interpretive parsing in metrical stress theory*. Ms., Rutgers University. ROA-262-0598.

Tesar, Bruce. 2000. Using inconsistency detection to overcome structural ambiguity in language learning. Technical Report RuCCS-TR-58, Rutgers Center for Cognitive Science, Rutgers University, New Brunswick NJ. ROA-426

Tesar, Bruce, John Alderete, Graham Horwood, Nazarre Merchant, Koichi Nishitani, and Alan Prince. 2003. Surgery in language learning. In *The Proceedings of WCCFL 22*, pp. 477-490. ROA-619.

Tesar, Bruce, and Paul Smolensky. 1998. Learnability in Optimality Theory. *Linguistic Inquiry* 29(2): 229-268.

Tesar, Bruce, and Paul Smolensky. 2000. *Learnability in Optimality Theory*. Cambridge: MIT Press.

Walker, Rachel. 1997. *Faith and Markedness in Esimbi Feature Transfer*. PASC 5, 103-115.

Walther, Markus. 1996. *OT SIMPLE—A construction kit approach to Optimality Theory implementation*. Ms., Heinrich-Heine-Universität, Düsseldorf. ROA-152-1096.

Walther, Markus. 2001. *Correspondence Theory: More Candidates Than Atoms in the Universe*. Ms., Philipps-Universität, Marburg. ROA-459-0801.

Wilson, Colin. 2004. *Analyzing unbounded spreading with constraints: marks, targets, and derivations*. Ms., UCLA.

Zoll, Cheryl. 1993. *Directionless syllabification and ghosts in Yawelmani*. Ms., University of California, Berkeley. ROA-28.