Cyclic feeding interactions between finite-state mal-rules: an algorithm for the optimal grouping and ordering of mal-rules

Robert Reynolds[†‡], Laura Janda[†], Tore Nesset[†]
[†] *UiT–Arctic University of Norway*, *Tromsø, Norway*
[‡] *Brigham Young University*, *Provo, UT, USA*

Abstract

Intelligent Language Tutoring Systems typically attempt to automatically diagnose learner errors in order to provide individualized feedback. One common approach is the use of mal-rules to extend normative grammars by licensing specific types of learner errors. In finite-state morphologies, mal-rules can be implemented as two-level rules or replace rules. However, unlike the phonological rules of natural languages, mal-rules do not necessarily behave as a coherent system, especially with respect to feeding interactions. Using examples from learner errors attested in the RULEC corpus of Russian learner texts, we illustrate the problem of cyclic feeding interactions that can occur between mal-rules. We then describe a formal algorithm for identifying an optimal ordering for mal-rules to be applied to a transducer.

Keywords: Learner errors, mal-rules, Russian, rule ordering, finite-state transducer

1. Introduction

Intelligent Language Tutoring Systems (ILTS) automatically analyze language produced by a learner in order to provide individualized feedback. Examples of ILTS include E-Tutor (Heift 2010), Robo-Sensei (Nagata 2009), TAGARELA (Amaral and Meurers 2011), the i-tutor (Choi 2016), the FeedBook (Meurers et al. 2019). Unlike a spell-checker, which merely suggests correct forms, an ILTS can provide remedial explanation and examples to help the learner understand *why* a given form is incorrect, or *how* to correct the error. This article is connected to recent work on a new ILTS: Russian Mentor for Orthographic Rules (RuMOR) (Reynolds et al. 2022).[1]

In order to provide this information, an ILTS must "abstract away from the specific string entered by the learner to more general classes of properties by automatically analyzing the learner input using NLP algorithms and resources" (Meurers 2020). This article is concerned with one particular approach to learner language analysis: mal-rules. Mal-rules are purposefully mal-formed rules that produce the same kinds of errors that learners make. By applying such rules to existing analyzers or parsers, we license learner errors, which can then be recognized by the system. In this way, we are able to analyze structures that are absent from normative Natural Language Processing (NLP) systems.

Mal-rules can interact with other rules in ways that violate the coherence of the rule system as a whole. This can pose technical issues for systems that integrate a large number of mal-rules. A simple demonstration of this problem is misspellings caused by learners' failure to distinguish two sounds in the target language. For example, many learners of Russian do not distinguish between the letters ш and щ (*š* and *šč* in transcription) or their respective sounds, [ʂ] and [ɕː]. This confusion would lead to the following errors: *xoroščo* (c.f., *xorošo* 'good') or *piša* (c.f., *pišča* 'food'). Modeling these kinds of errors might require two rules, as shown in (1).

(1) a. šč → š (and add the tag +shch2sh)
b. š → šč (and add the tag +sh2shch)

Applying these rules sequentially to the word *pišča* would result in the analyses shown in (2).

(2) a. *pišča* : *pišča*
b. *pišča*+shch2sh : *piša*
c. *pišča*+shch2sh+sh2shch : *pišča*

As can be seen in (2), applying rule (1a) to (2a) results in (2b), which is the expected error, with the appropriate tag. However, because the first rule feeds into the second, when rule (1b) is subsequently applied to the lexicon, it

[1] https://icall.byu.edu/rumor

 Nordlyd 46.1: 219—229, *Morfologi, målstrev og maskinar: Trond Trosterud {fyller | täyttää | deavdá | turns} 60!*, edited by Lene Antonsen, Sjur Nørstebø Moshagen and Øystein A. Vangsnes. Published at UiT The Arctic University of Norway. http://septentrio.uit.no/index.php/nordlyd https://doi.org/10.7557/12.6306

converts (2b) back into the original, correct wordform (note that (2b) and (2c) have identical surface forms). The result is an analyzer that would return two analyses for the surface form *pišča*, one with its correct normative reading, and one with two contradictory error tags. Reversing the order of the rules would fix the problem for *pišča*, but then the word *xorošo* would have an analogous problem. We refer to this set of circumstances as a cyclic feeding interaction. Although two-rule cycles, such as this example, are the most common, cyclic feeding interactions can involve any number of rules. Furthermore, a given rule can be part of any number of cycles.

Our use of the term *cycle* is not connected to uses in Cyclic Phonology and related generative frameworks. Their use of the term *cyclic* stems from the work of Chomsky et al. (1956) in their analysis of English word stress. In research inspired by their work, cycles are iterative application of the same rule(s) at increasingly larger morphosyntactic units. In contrast, our use of the term *cycle* is taken from Graph Theory, as discussed in Section 2.2. We represent rules in a graph, where each rule is a node in the network, and feeding interactions between rules are represented as directed edges (arrows) between nodes. In this representation, a cycle is a relation between rules where feeding interactions ultimately lead back to the original form. As discussed in Section 2.1, some researchers name this a "mutual feeding" relation, but we avoid this term because it implies that only two rules are involved in the interaction.

1.1. Acyclic feeding interactions

We have seen that cyclic feeding relations can be problematic, but *acyclic* feeding interactions are also consequential for rule interactions. The toy rules in (3) are an example of rules in a feeding order. Because they are in a feeding order, the analyzer would recognize wordforms with both errors on the same word (e.g. cat+a2o+o2u : cut). However, if the order of the rules is reversed (a counterfeeding order), the analyzer would fail to recognize this interaction of errors. In order to model mal-rule interactions, it is important to optimize the order of all the rules to maximize the number of rules in feeding orders (and minimize the number of rules in counter-feeding orders). Given that a set of n rules can be arranged in $n!$ different permutations, an algorithm that can approach this task automatically would be a boon to rule authors.[2]

(3) a. a → o (and add the tag +a2o)

b. o → u (and add the tag +o2u)

1.2. Article structure

In this article, we present a formal algorithm, implemented as an open-source python script,[3] to block cyclic feeding interactions. In addition, our algorithm assesses the consequences of ordering *acyclic* feeding interactions, and suggests an ordering of errors to optimize the feeding interactions of such rules.

In Section 2, we discuss related research in three fields: generative phonology (§2.1), mal-rules (§2.3), and Graph Theory (§2.2). In Section 3, we describe the algorithm. In Section 4, we apply the algorithm to a real-world set of mal-rules modeling Russian learner orthographic and morphological errors. Finally, in Section 5, we summarize our contribution and outline a few paths for future work.

2. Related work

2.1. Generative approaches to feeding and bleeding orders

The terms *feeding order* and *bleeding order* were first introduced by Kiparsky (1968). A feeding order is an ordering of two rules such that the first rule generates new contexts in which the second order applies. Stated negatively, for some words the second rule would not apply if the first rule had not created the right context. When two rules could have a feeding order, but they are not in the right order for a feeding interaction to occur, they are said to be in a *counterfeeding*

[2] It is worth noting that rule interactions of this kind can rapidly explode the size of a transducer, which could strain computational resources. In this case, one might wish to *avoid* modeling mal-rule interactions in order to keep the transducer smaller. This can be achieved by reversing the optimal feeding order to produce the optimal counter-feeding order.

[3] https://github.com/reynoldsnlp/xfst_malrule_ordering

order. Whereas a feeding order is a *timely* application of a feeding interaction, a counterfeeding order can be said to be a *tardy* application of a feeding interaction.

A bleeding order is an ordering of two rules such that the first rule destroys contexts in which the second rule applies. In other words, for some words the second rule would apply if not for the first rule. Just as with feeding interactions, if two rules could have a bleeding order, but they are in the wrong order for a bleeding interaction to occur, they are in a *counterbleeding* order. A summary of these interactions is given in Table 1. In this article, we focus only on *feeding* interactions.

		Chronology	
		timely	tardy
Interference	excitatory	feeding	counterfeeding
	inhibitory	bleeding	counterbleeding

Table 1: Types of rule interaction

Generative linguists have wrestled with the problem of *cyclic* feeding orders, albeit for different reasons than the present article. For their theories, cyclic feeding interactions (also known as "mutual" feeding interactions) pose problems for explanatory power, parsimony (i.e. Occam's Razor), and learnability. For example, one mechanism that can be used to solve the problem of cyclic/mutual feeding is disjunctive ordering (Chomsky and Halle 1968), which uses braces/brackets/parentheses to define a schema from which only one rule can be applied. Similarly, the Elsewhere Condition (Kiparsky 1973) was put forth as an alternative to disjunctive ordering.

Pullum (1976) expresses suspicion of derivations of the type A → B → A, which he calls "Duke-of-York derivations."[4] He argues that in most cases, Duke-of-York derivations should be avoided on the grounds of parsimony, but he also claims that there are cases in which a Duke-of-York derivation is either simpler than the alternatives or the only possible explanation. However, McCarthy (2003) argues that the Duke-of-York phenomena described in the literature are vacuous because the intermediate step is not necessary. He claims that non-vacuous Duke-of-York phenomena do not exist in natural language.

It is worth emphasizing that the motivations behind the discussion of cyclic/mutual feeding orders of generative grammars are very different from those of the current article. First and foremost, mal-rules are not intended to behave as a coherent, parsimonious, learnable system of generalizations. Each individual mal-rule represents its own self-contained deviation from a normative grammar. As such, the output of each mal-rule is a spelled out, final wordform. To borrow from the analogy of the Duke of York referenced in Footnote 4, in a generative grammar the Duke of York's intermediate top-of-the-hill is just an inefficient detour. However, in a mal-rule approach to morphological analysis, the top-of-the-hill is an important waypoint along many possible paths an error analysis might take. The problem with mal-rules only arises if a cycle of waypoints leads back to where we came from, because the error tags, which record the path of errors taken, become both self-contradictory and redundant.

Researchers in finite-state morphology have developed a number of solutions to solve the same kinds of problems described by generativists. A two-level morphology (Koskenniemi 1983) compiles all phonological rules into one transducer and can thereby apply all of the rules to the lexicon simultaneously. For example, Karttunen (1993) demonstrates a number of two-level solutions, including one that functions similarly to the disjunctive ordering proposed by Chomsky and Halle (1968). Such applications of two-level morphology are not relevant to the problem of cyclic feeding interactions among mal-rules because they do not allow for keeping and tagging intermediate forms. Individual mal-rules can be implemented as two-level rules, as discussed in Section 2.3 below, but two-level rules cannot solve the problem of cyclic feeding interactions between mal-rules.

2.2. Cycle detection (Graph Theory)

Graph Theory is the study of pairwise relations between objects, typically as part of a network of such relations. We represent feeding relations as directed graphs, where edges go from a given rule's node to the nodes of rules that it feeds into. By using a graph representation, we are able to take advantage of previous research regarding the detection of cycles.

[4]The name comes from the eponymous poem that exhibits a fruitless round trip: *Oh, the grand old Duke of York, He had ten thousand men; He marched them up to the top of the hill, And he marched them down again.*

A cycle in a directed graph can be thought of as an infinite loop, where a given path leads back to its own beginning node. More formally, a directed cycle is a non-empty path in which the only repeated nodes are the first and last nodes, i.e. the first and last nodes in the path are the same node. When representing replace rule interactions as a directed graph, cyclic feeding interactions are conveniently manifested as graph cycles.

Although more efficient algorithms have been put forward, cycles in a directed graph can be detected using depth-first search. If a search finds an edge that points to an ancestor of the current node, the edge to that ancestor is called a *back edge*. All the back edges which depth-first search skips over are part of cycles. Several algorithms have been suggested that are more efficient than naive depth-first search, and we use the algorithm described in Johnson (1975), as implemented in the python package `networkx`[5] (Hagberg et al. 2008).

2.3. Mal-rules

Mal-rules are rules—orthographic, phonological, syntactic, etc.—that generate or license learner errors (Sleeman 1982, Matthews 1992, Antonsen 2012, Reynolds et al. 2022). In the domain of finite-state morphological analysis, mal-rules can be implemented in a number of ways. One example can be taken from Antonsen (2012), who implements mal-rules as part of a two-level ruleset in the following way. First, error tags are added to a path in the lexicon (`lexc`) on both the upper and lower sides. Then, the tag is removed from the lower side under specific conditions using `twolc` rules. The analyses with the error tag in both levels are then removed from the transducer by means of XFST-style regular expression rules. One drawback of this approach is that it does not always allow for combining multiple errors on the same wordform, without significantly complicating interactions between `twolc` rules.

The approach assumed in this article is taken from Reynolds et al. (2022), whose mal-rules are implemented as regular expression replace rules. Figure 1 illustrates this process. Importantly, this entire process is based on a pre-existing transducer that contains the lexicon with all normative surface forms of the target language, along with their morphosyntactic tags. The pre-existing transducer is used to initialize the `main` transducer discussed below.

A replace rule with the optional replacement operator (i.e. `(->)` or `(<-)`) is applied to the main transducer (`main`) to yield an intermediate transducer (`inter1`) which contains everything from the `main` transducer, as well as all forms generated by the rule.[6] The `main` transducer is then subtracted from `inter1` to yield an intermediate transducer (`inter2`) with only those forms that were generated by the rule. Another rule is applied to add an error tag to all readings in `inter2` to yield the last intermediate transducer (`inter3`) which has only the forms generated by the mal-rule, with error tags. The `main` transducer is then replaced by a disjunction of itself with `inter3`, and other mal-rules can then be applied in the same fashion. In this way, mal-rules naturally stack on one another to yield forms that are the combination of multiple errors.[7]

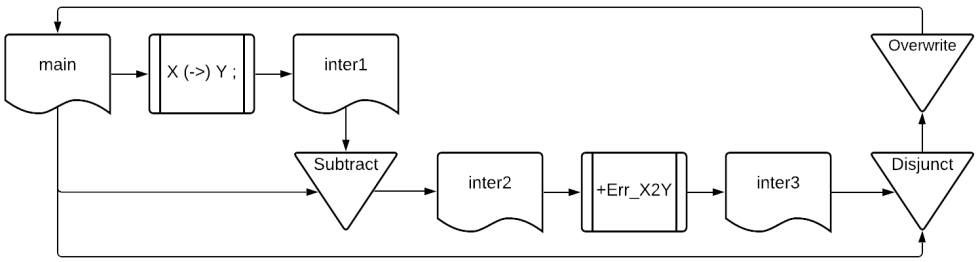

Figure 1: Workflow for adding error(s) to `main` transducer using regular expression replace rules

[5] https://networkx.org/

[6] If a wordform from the original lexicon or preceding mal-rules contains multiple instances of the letter(s) to be changed, the optional replacement operator results in multiple outputs, including every combination of optional changes. In our approach, all of these resulting wordforms are tagged identically, with only one error tag.

[7] It is possible to use `twolc` to achieve a similar result, where instead of applying a replace rule to a fully-compiled transducer, you apply a modified form of the "normative" `twolc` rule set (with only the changes necessary to produce one type of error) to the lexical transducer from `lexc`. Although this approach does give rules access to the underlying forms from the `lexc` transducer, it does not naturally allow for error stacking, since it is based on the output of `lexc`.

3. Methods: Blocking cyclic feeding interactions

While discussing our algorithm for blocking cyclic feeding interactions from our mal-rule application order and maximizing feeding interactions, we use the toy rules in (4) to illustrate the rule interactions.

(4) a. a simple acyclic feeding order (`i → j → k`)
b. a two-rule cyclic feeding order (`m ↔ n`)
c. a 3+-rule cyclic feeding order (`a → e → i → o → u → a`)

3.1. Generating feeding graph from regex rules

The first step of our algorithm is to convert our replace rules into a directed graph, with each node representing a rule, and each directed edge representing a feeding relation from one rule to another. Our method assumes that the replace rules are written as XFST regular expressions, one file per error type. In simple cases, each file may contain a single replacement, and in more complex cases, it may contain multiple parallel replacements or the composition of multiple replacements.

The replacement rule(s) for each error are extracted by reading each source file directly using a simple regular expression to identify every instance of a token followed by one of the optional replacement operators (`(->)` or `(<-)`), followed by another token. Currently, the contextual constraints of conditional replacement (e.g. the `|| L _ R` in the rule `A (->) B || L _ R`) are ignored.[8] This means that the algorithm identifies all true feeding relations (perfect recall), but in cases where constraints block rule interaction, the algorithm generates false positives (imperfect precision). Although we do not yet have a way to automatically detect which constraints block feeding interactions, our script can be run in interactive mode, where a human can manually override these false positives.

The graph is generated by creating a node for each error. Then, for each replace rule belonging to that error, a directed edge is added from that error to any error whose rules take as input that rule's output.[9] An example based on (4) is given in Figure 2. Note that in more complicated real-world examples with multiple replacements per error, there may be more than one edge leaving a node.

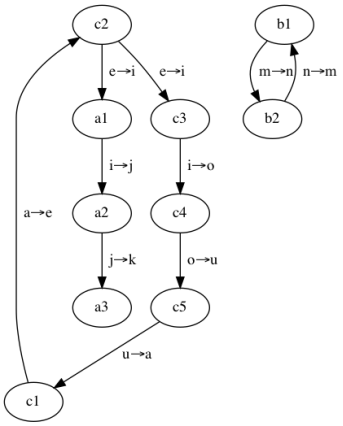

Figure 2: Directed graph generated from rules in (4)

3.2. Blocking cycles

Simple cycles in the rule graph can combine with other cycles to form more complex cycles. As discussed below, our algorithm eliminates cycles by removing all edges between two given nodes, and since removing simple cycles consequently removes their more complex derivatives, our algorithm can focus on removing only simple cycles in the rule graph. To do this, our algorithm starts by addressing cycles of shortest length first, then iteratively removing cycles of greater and greater length until no cycles remain.

As discussed in Section 2.3, we assume an approach that adds errors serially, one after the other. Traditionally, feeding interactions are blocked by putting rules in a counterfeeding order, but this solution does not work for two-rule cyclic feeding interactions.

3.2.1. Blocking two-rule cycles

Two-rule cyclic feeding interactions (a.k.a. "mutual feeding interactions") cannot be placed in a counterfeeding order. Instead, they are added in parallel so that neither rule can feed into the other, as shown in Figure 3. In this figure, both

[8] One reason for this is that transducer composition is non-commutative, so compositions of rule transducers are themselves dependent on the rule ordering. This means that potentially every possible permutation of the rules must be tested to determine whether the rules have a cyclic feeding interaction. This is left to future research, as discussed in Section 5.1.

[9] Errors that feed themselves are assumed to be false positives.

errors are based on the same version of `main`, and their outputs are added back into `main` at the same time, so they can no longer interact. In terms of the rule graph, when two rules are added in parallel, all edges between those two nodes are removed.

In the case of our toy example, the errors associated with rules `b1` and `b2` (`m` → `n` and `n` → `m`, respectively) would be added in parallel.

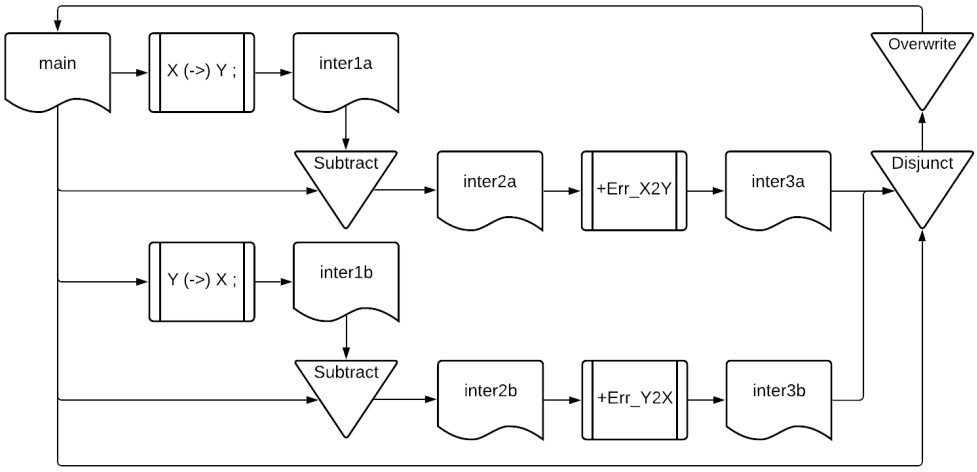

Figure 3: Workflow for adding multiple errors in parallel to `main` transducer using regular expression replace rules

3.2.2. Blocking cycles with more than two rules

In our toy example, the vowel cycle (`a` → `e` → `i` → `o` → `u` → `a`) represents a cycle of more than 2 rules. Cycles composed of more than 2 rules can be be broken by removing only one feeding interaction from the cycle. This can be achieved by putting two rules in a counterfeeding order. For example, if `e` → `i` were ordered before `a` → `e`, then they would be in a counterfeeding order, their feeding interaction would not be manifested, and the cycle would be broken. This is ideal for scenarios in which the optimal ordering described in Section 3.3 below naturally places any two of these rules in a counterfeeding order.

Another approach is to simply add the rules in parallel, as is done with two-rule cyclic feeding interactions. Depending on what parallelized rule sets already exist, parallelizing yet another pair of errors could have unwanted consequences, such as merging two existing parallelized rule sets. For example, if two rule sets already exist, `{A, B}` and `{C, D}`, then parallelizing `A` and `C` would result in merging the two sets into one, which would make it impossible for these errors to stack. The larger the existing rule sets, the greater the impact on stacking.

This scenario poses a dilemma which can only be resolved arbitrarily. Either merge the rule sets to add a large number of errors in parallel (which limits error stacking), or force a counterfeeding order to override the optimal feeding order discussed below is Section 3.3 (which also limits stacking). Here the user should weigh the relative impacts of these approaches on stacking.

Since we already use parallel addition for avoiding two-rule cycles, we default to the same mechanism for blocking cycles with more than two rules. The algorithm can randomly select which two rules to add in parallel, but in interactive mode, the user can manually select the rules (either to add in parallel or to force into a counterfeeding order). In theory, adding any two rules from the cycle in parallel would break the cycle, but in order to maximize the remaining feeding interactions (in the same spirit as Section 3.3 below) we limit the options to removing rules that are *adjacent* in the cycle.

3.3. Methods: Maximizing feeding order

As long as cyclic feeding interactions are effectively blocked, it is usually advantageous to maximize the feeding order of mal-rules so that all possible rule interactions are modeled. In order to estimate the order of rules that maximizes the feeding order, we leverage the work of Gansner et al. (1993), Ellson et al. (2002), whose graph visualization work is implemented in the popular `graphviz`[10] utility. We are most interested in the `dot` algorithm, which is used to render hierarchical drawings of directed graphs.

The first pass of the `dot` algorithm determines the optimal rank assignment of each node consistent with its edges. Roughly speaking, this means that nodes with the most incoming edges are placed at the bottom of the image and nodes with the fewest incoming edges are placed at the top. The result is that the overall flow of the graph runs from top to bottom. All nodes that share the same rank are assigned the same y-coordinates, so the y-coordinate in the output of the `dot` algorithm can be interpreted as a proxy for its rank. Therefore, ordering the mal-rules according to their y-coordinates from top to bottom will optimize their feeding interactions, since the top-most rules have the fewest incoming edges.

The final rule ordering and grouping output by the algorithm takes the order from the `dot` algorithm, and integrates each of the parallelized rule sets by replacing the first instance of one of its members with the entire set. For example, if the rank order from the `dot` algorithm for our toy example were `[b1, b2, c2, a1, c3, a2, c4, a3, c5, c1]` and the parallelized rule sets were `[{b1, b2}, {c1, c2}]`, then the final ordering/grouping would be `[{b1, b2}, {c1, c2}, a1, c3, a2, c4, a3, c5]`.

4. Example from Russian

In order to show a practical example of the algorithm described in Section 3, we apply it to those mal-rules described in Reynolds et al. (2022) that are implemented as regular expression replace rules. A summary of these rules is given in Table 2.

Tag	Tag explanation	Example (Correct form in parentheses)
a2o	Misspelling (о should be а)	озночает (означает)
e2je	Misspelling (е should be э)	ето (это)
Gem	Should be just single, not geminate, letter	рассширить (расширить)
H2S	Misspelling (ь should be ъ)	подьезд (подъезд)
i2j	Misspelling (й should be и)	миллйард (миллиард)
i2y	Misspelling (ы should be и)	блызко (близко)
Ikn	Ikanje (и should be е/я/а)	дитей (детей)
j2i	Misspelling (и should be й)	рабочии (рабочий)
je2e	Misspelling (э should be е)	проэкта (проекта)
NoGem	Geminate letter is missing	имено (именно)
NoSS	Misspelling (ь is missing)	болше (больше)
o2a	Akanje (а should be о)	каторый (который)
prijti	Misspelling the stem of прийти	прийду (приду)
revIkn	Reversed Ikanje (е, а, я should be и)	умерает (умирает)
sh2shch	Misspelling (щ should be ш)	лучще (лучше)
shch2sh	Misspelling (ш should be щ)	вообше (вообще)
ski	по-∼ский instead of по-∼ски	по-русский (по-русски)
SRc	Spelling Rule ы>и (after ц)	близнеци (близнецы)
SRy	Spelling Rule ы>и	книгы (книги)
y2i	Misspelling (и should be ы)	описивают (описывают)

Table 2: Russian mal-rules implemented as regular expression replace rules in Reynolds et al. (2022)

The graph generated from the feeding interactions of these mal-rules has 19 nodes and 139 edges, with 1347

[10] https://graphviz.org

cycles. A visualization of the graph, generated by the `dot` algorithm of `graphviz`, is shown in Figure 4 at a small scale for general reference, but note that digital versions of the document allow for zooming in.

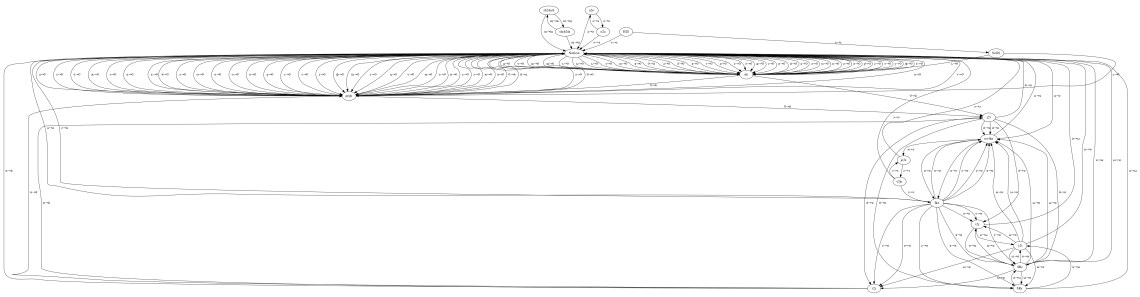

Figure 4: Visualization of the Russian graph based on Table 2 before removing cycles

The algorithm identifies 12 two-rule feeding cycles, and constructs sets of rules that should be added in parallel, as shown in (5). Although most of these pairs are obvious to humans, note that some of the sets have more than two errors because some rules are involved in more than one cyclic feeding interaction.

(5) a. {sh2shch, shch2sh}
b. {a2o, o2a}
c. {Ikn, revIkn}
d. {i2j, j2i}
e. {SRy, y2i, i2y, SRc}
f. {je2e, e2je}
g. {ski, prijti, NoGem}

After removing the 12 two-rule cyclic feeding orders, the remaining graph has 19 nodes and 46 edges, with only one cycle. Note that blocking 12 cycles resulted in the removal of 93 edges. This is because some errors include more than one replace rule that can interact with other rules. For example, the `NoGem` error models when learners fail to write both letters of a geminate, and includes 33 replaces rules, one for each letter of the alphabet.

Also note that although only 12 cycles (i.e., 93 edges) were removed, the total number of cycles in the graph was reduced from 1347 to 1. This is because the majority of the cycles were complex combinations of shorter, simpler cycles. By removing the edges of their component cycles, these complex cycles were also removed.

The only remaining cycle is `prijti` → `j2i` → `SRc` → `i2j` → `prijti`. The algorithm begins by checking whether this cycle is already broken, either because of existing parallelizations, or because the `dot` algorithm currently places the nodes in a counterfeeding order. In this case, the cycle is broken because two of the errors in this cycle are already parallelized: `i2j` and `j2i`. Therefore, no further action is needed to break this cycle.

Although this cycle is already broken, it is worth thinking through what would happen if this were not the case. Although one could randomly select an adjacent pair of nodes to add in parallel, we explore the consequences of this decision. Adding `prijti` and `j2i` in parallel would have the same effect as adding `i2j` and `prijti` in parallel: the sets in (5d) and (5g) would be merged into a set of five rules to be added in parallel. Similarly, blocking either `j2i` → `SRc` or `SRc` → `i2j` would have the same effect regardless: the sets in (5d) and (5e) would be merged into a set of six rules to be added in parallel.

In the spirit of maximizing non-cyclic feeding interactions, adding five errors in parallel could theoretically block fewer feeding interactions than adding six errors in parallel. Otherwise, it could be worth considering which errors in the existing sets are more or less likely to co-occur, preferably on the basis of empirical evidence, and to make the selection to keep co-occuring errors in different sets so that they can stack on top of each other.

In actuality, none of this was necessary, since this cycle was already broken because of pre-existing parallelizations.

Finally, the algorithm plotted the final graph with no feeding cycles using the `dot` algorithm, as shown in Figure 5. Using the y-coordinates as proxy for feeding rank, the algorithm output the following order with parallelized groups

in curly braces: `[H2S, NoSS, {NoGem, prijti, ski}, {e2je, je2e}, {Ikn, revIkn}, {i2j, j2i}, {SRc, SRy, y2i, i2y}, {a2o, o2a}, {shch2sh, sh2shch}]`.

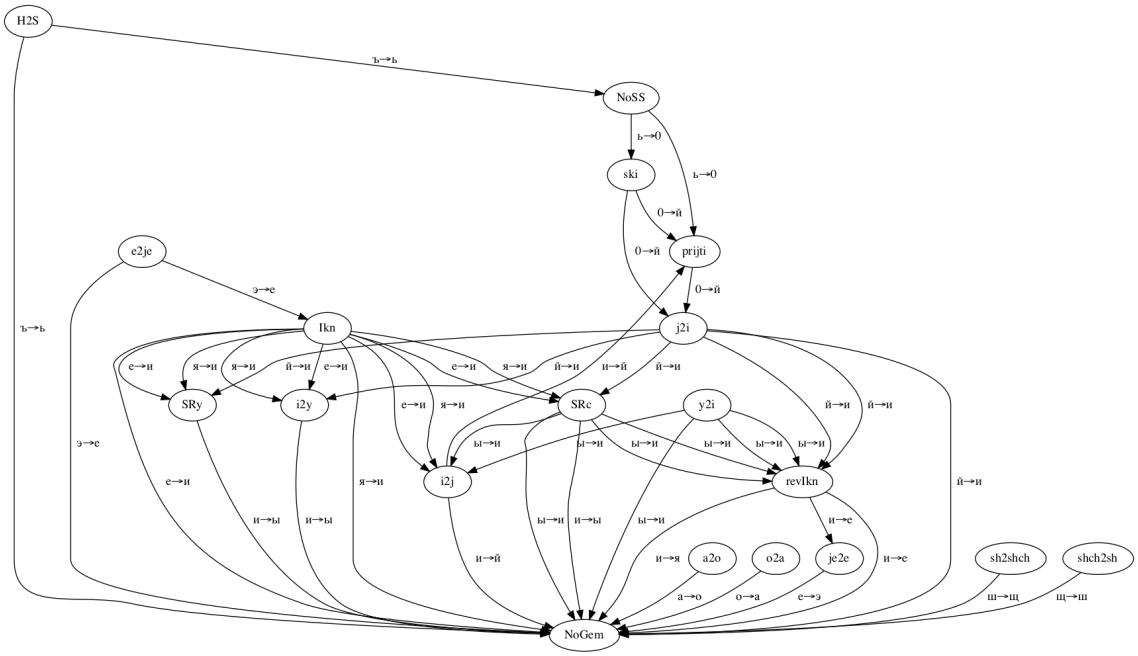

Figure 5: Visualization of the Russian graph based on Table 2 after removing cycles

5. Conclusions

We have discussed the problem of cyclic feeding interactions between mal-rules, and introduced a new algorithm for identifying an ordering (with parallelized groups) to block cyclic feeding interactions and maximize all other feeding interactions. The algorithm can be run in interactive mode, which allows the user to override the automatic behavior of the algorithm.

One unintended benefit of the algorithm is that by making feeding interactions explicit, the user may identify ways to edit replace rules to make them more precise. For example, the `NoGem` rule mentioned in Section 4 was originally written to delete geminate letters for every letter in the alphabet. However, some Russian letters never appear twice in a row, including ь, ъ, and й. By including them in the error rules, unnecessary rule interactions were introduced. The explicit inspection of feeding interactions, as facilitated by our algorithm, can lead to more precise rule writing. Although the intended application of our algorithm is with mal-rules, it could be used for analyzing the feeding interactions of any system of replace rules.

5.1. Future work

The proposed algorithm is focused on *feeding* interactions, but does not consider *bleeding* interactions. Future work can explore how to integrate awareness of bleeding and counterbleeding interactions in the algorithm to maximize error stacking.

As discussed in Section 3.1, our algorithm is not sensitive to conditional constraints on replace rules. The solution to this problem will most likely depend on the existing regular expression compilers for XFST/HFST. For example, one could compile transducers from the replace rules and then compose those transducers independent from any lexicon, and examine the structure of the resulting transducer. However, transducer composition is not a commutative operation,

so the composition of the rules is itself dependent on the order in which the rules are composed. More research is needed to determine whether this (or another) approach could be used to produce a graph of feeding interactions that is sensitive to the conditional constraints of each rule.

Currently, parallelized error sets are added to the final rule ordering at the first occurrence of one of its members in the ranks output by `dot`. Future work is needed to determine how this affects the feeding interactions of other members of the set, and how to optimize the set's position on that basis.

Lastly, the strategy of selecting which errors to parallelize in cyclic feeding interactions of more than two rules is multi-faceted and complex. As discussed in Section 4, this includes the potential merging of existing error sets, as well as the question of maximizing feeding interactions. It is difficult for users to know how a given decision would interact with existing parallelization groups, as well as how it affects future decisions. Future work is needed to help users quickly and easily assess the consequences of parallelizing each pair of errors with respect to these factors.

References

Amaral, Luiz and Detmar Meurers. 2011. On using intelligent computer-assisted language learning in real-life foreign language teaching and learning. *ReCALL* 23 1: 4–24. https://doi.org/10.1017/S0958344010000261.

Antonsen, Lene. 2012. Improving feedback on l2 misspellings-an fst approach. In *Proceedings of the SLTC 2012 workshop on NLP for CALL; Lund; 25th October; 2012*, 080, pp. 1–10. Linköping University Electronic Press.

Choi, Inn-Chull. 2016. Efficacy of an ICALL tutoring system and process-oriented corrective feedback. *Computer Assisted Language Learning* 29 2: 334–364. https://doi.org/10.1080/09588221.2014.960941.

Chomsky, Noam and Morris Halle. 1968. *The sound pattern of English*. Harper & Row.

Chomsky, Noam, Morris Halle, and Fred Lukoff. 1956. On accent and juncture in english. *For Roman Jakobson* 65: 80.

Ellson, John, Emden Gansner, Lefteris Koutsofios, Stephen C. North, and Gordon Woodhull. 2002. Graphviz— open source graph drawing tools. In *Graph Drawing*, edited by Petra Mutzel, Michael Jünger, and Sebastian Leipert, pp. 483–484. Springer Berlin Heidelberg, Berlin, Heidelberg. https://doi.org/10.1007/3-540-45848-4_57.

Gansner, E.R., E. Koutsofios, S.C. North, and K.-P. Vo. 1993. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering* 19 3: 214–230. https://doi.org/10.1109/32.221135.

Hagberg, Aric, Pieter Swart, and Daniel S Chult. 2008. Exploring network structure, dynamics, and function using networkx. Tech. rep., Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

Heift, Trude. 2010. Developing an intelligent language tutor. *CALICO Journal* 27 3: 443–459. https://doi.org/10.1558/cj.27.3.443-459.

Johnson, Donald B. 1975. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing* 4 1: 77–84.

Karttunen, Lauri. 1993. Finite-state constraints. *The last phonological rule* 6: 173–194.

Kiparsky, Paul. 1968. Linguistic universals and linguistic change. In *Universals in linguistic theory*, pp. 170–202. Holt, Rinehart and Winston, New York.

Kiparsky, Paul. 1973. Elsewhere in phonology. In *A Festschrift for Morris Halle*, pp. 93–106. New York: Holt, Rinehart and Winston.

Koskenniemi, Kimmo. 1983. Two-level morphology: A general computational model for word-form recognition and production. Tech. rep., University of Helsinki, Department of General Linguistics.

Matthews, Clive. 1992. Going AI: Foundations of ICALL. *Computer Assisted Language Learning* 5 1: 13–31. https://doi.org/10.1080/0958822920050103.

McCarthy, John J. 2003. Sympathy, cumulativity, and the duke-of-york gambit. *The syllable in optimality theory* pp. 23–76. https://doi.org/10.1017/CBO9780511497926.003.

Meurers, Detmar. 2020. Natural language processing and language learning. In *The Concise Encyclopedia of Applied Linguistics*, edited by Carol A. Chapelle, pp. 817–831. Wiley, Oxford.

Meurers, Detmar, Kordula De Kuthy, Florian Nuxoll, Björn Rudzewitz, and Ramon Ziai. 2019. Scaling up intervention studies to investigate real-life foreign language learning in school. *Annual Review of Applied Linguistics* 39: 161–188. https://doi.org/10.1017/S0267190519000126.

Nagata, Noriko. 2009. Robo-Sensei's NLP-based error detection and feedback generation. *CALICO Journal* 26 3: 562–579. https://doi.org/10.1558/cj.v26i3.562-579.

Pullum, Geoffrey K. 1976. The duke of york gambit. *Journal of linguistics* 12 1: 83–102. https://doi.org/10.1017/S0022226700004813.

Reynolds, Robert, Laura Janda, and Tore Nesset. 2022. RuMOR: Russian Mentor for Orthographic Rules: ICALL to

help learners of Russian become confident writers. *Computational Linguistics and Text Complexity: a special issue of the Russian Journal of Linguistics* p. 15.

Sleeman, D. 1982. Inferring (mal) rules from pupil's protocols. In *Proceedings of ECAI-82*, pp. 160–164. Orsay, France.