Trabalho Linguagens Formais e Autômatos - Parte 2

Guilherme Guimarães
Eduardo Altmann de Bem
Leonardo Greco Fin

Introdução

Foi desenvolvido na primeira etapa do trabalho um sistema de cadastro que reconhece senhas consideradas “fortes”, usando os estados de um autômato finito para guardar os requisitos alcançados. Agora, iremos expandir esse sistema para que este seja necessariamente modelado por uma linguagem livre de contexto (GLC), provando que o novo cenário não pode ser descrito por uma linguagem regular.

O Sistema

O objetivo do novo sistema é reconhecer entradas em que a senha de login é a mesma da senha de cadastro, fazendo o papel de um autenticador. Infelizmente, linguagens livres de contexto não são poderosas o suficiente para descrever palavras do tipo ww (uma palavra repetida duas vezes), então será preciso a formulação de uma técnica para simular/aproximar essa classe de linguagens sem alterarmos a entrada do sistema.

Outras Tentativas

Para contornar esse problema, seria possível definir o sistema de forma que a entrada contenha a primeira ou segunda ocorrência da senha ao contrário, formando a palavra $w^r w$ / ww^r. Essas entradas são reconhecidas por mecanismos livres de contexto. Entretanto, isso afetaria a usabilidade do sistema, dado que o usuário precisará manualmente escrever uma das senhas inversas. Dessa forma, decidimos pela estratégia a seguir.

A Estratégia

Como visto, GLCs são incapazes de gerar palavras do tipo ww, mas conseguem gerar palavras do tipo $a^n b^n$. Então, se conseguirmos achar um modo de interpretar a entrada como uma palavra balanceada, será possível modelar o sistema usando uma GLC.

A ideia básica dessa estratégia é a transformação $w \to a^n$. Ou seja, é preciso criar um mapeamento de todas as possíveis senhas para sequências de um único símbolo. Se interpretarmos a palavra a^n como o número n em base unária, vemos que essa transformação é um *hash* de w, onde mapeamos uma *string* arbitrária para um número. Assim, é preciso designar um algoritmo de *hashing* que seja implementável por um autômato de pilha.

O Algoritmo

O nosso algoritmo de *hashing* possui alguns requisitos que precisam ser cumpridos para poder ser usado pelo nosso sistema:

1. palavras iguais devem resultar no mesmo número (*hash*)
2. colisões de *hash* devem ser mínimas (palavras diferentes com mesmo *hash*)

3. o algoritmo precisa fazer um único passe pela palavra

Com esses requisitos em mente, iremos começar com um algoritmo básico e o aprimorar aos poucos, até chegarmos em uma solução aceitável.

Primeira Iteração
O algoritmo mais básico implementável por um autômato de pilha é um que associa um número distinto a cada símbolo e faz a soma dos números correpondentes aos símbolos da palavra de entrada. Expressando em termos matemáticos, com uma palavra sendo uma sequência $w = x_1x_2x_3...x_n$, $h(w)$ sendo a função de hash, e $p(x)$ sendo o mapeamento de um símbolo para um número:

$$h(w) = \sum_{k=1}^{n} p(x_k)$$

Esse algoritmo é simples de implementar e garante que palavras iguais resultem no mesmo *hash*. Entretanto, o número de colisões é muito alto, considerando que permutações (anagramas) da mesma palavra criam *hashes* iguais, devido à natureza comutativa da adição.

Segunda Iteração
Para melhorarmos o problema das colisões, precisamos que a posição do símbolo na palavra influencie o seu valor na soma do *hash*. Um método fácil seria multiplicar o $p(x)$ pela posição do símbolo antes de adicionar à soma:

$$h(w) = \sum_{k=1}^{n} k * p(x_k)$$

Infelizmente, a taxa de colisões ainda permanece alta. Considere o alfabeto de entrada $\Sigma = \{a..z\}$, $p(x) = \{\text{posição do símbolo no alfabeto latino}\}$ e palavras $w_1 = \text{cba}$ e $w_2 = \text{j}$:

$$\begin{aligned} h(\text{"cba"}) &= 1 * p(c) + 2 * p(b) + 3 * p(a) \\ &= 1 * 3 + 2 * 2 + 3 * 1 = 10 \end{aligned}$$

$$h(\text{"j"}) = 1 * p(j) = 1 * 10 = 10$$

Terceira Iteração
Um modo de resolver completamente o problema das colisões seria ao invés de multiplicar o $p(x)$ pela posição do símbolo, tratar a palavra de entrada como um número e fazer uma mudança de base da palavra para a base unária.

Se definirmos a base da palavra de maneira correta, a operação de conversão de bases se torna uma função injetora, garantindo um mapeamento único de palavras para *hashes*.

Primeiro é preciso definir qual a base da palavra de entrada. Sabendo que a base de qualquer sistema numérico é o tamanho do conjunto de dígitos (ex: a base decimal contém {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} = 10 dígitos), poderiamos definir a base da nossa palavra como o tamanho do alfabeto de entrada Σ. Nesse caso, iremos definir como $|\Sigma| + 1$, pois nenhum símbolo terá o valor 0, para evitar problemas como $1 = 01 = 001$.

Com a base definida, podemos fazer a operação de mudança de bases calculando a soma de todos os valores numéricos dos caracteres da palavra multiplicados pela base elevada à posição do caracter. Definindo a base do alfabeto de entrada como b, temos:

$$h(w) = \sum_{k=1}^{n} b^{k-1} * p(x_k)$$

Interpretando a palavra como um número de base b, se nota que os dígitos do número estão invertidos, com o digíto menos significativo mais à esquerda. Entretanto, isso se torna irrelevante visto que o valor em si do número não importa, mas apenas que as propriedades do *hash* sejam mantidas.

Também vemos que o valor do número cresce de forma exponencial em relação ao tamanho da palavra. Isso não é um problema dado que o modelo teórico de um autômato de pilha possui memória infinita.

Para terminar a transformação, convertemos $h(w)$ para uma base unária que é processável pelo autômato (usando um símbolo arbitrário a):

$$h(w) \to a^{h(w)}$$

O Problema

Como a pilha já está sendo utilizada, precisamos codificar a informação da posição do símbolo nos estados. Isso traz um problema: os estados são finitos, enquanto as palavras têm tamanho arbitrário. Similarmente, autômatos com pilha não conseguem calcular exponenciais nem multiplicações por números arbitrários. Então é nesse momento que é preciso introduzir uma aproximação do comportamento ideal.

Algorítmo Final

O caminho encontrado foi dividir a palavra em blocos de tamanho m, e calcular os exponenciais módulo m. Como m é finito, podemos precalcular esses valores, e transformar as operações do algorítmo em uma multiplicação por uma constante (o que sabemos que é possível fazer). Assim, o algorítmo final seria:

$$h(w) = \sum_{k=1}^{n} b^{(k-1) \bmod m} * p(x_k)$$

Um exemplo desse algorítmo usando um alfabeto $\Sigma = \{a, b, c, d, e, f\}$ e base 7:

Primeiro definimos o mapeamento $p(x) = \{(a, 1), (b, 2), (c, 3), (d, 4), (e, 5), (f, 6)\}$, a palavra $w = \text{abcd}$, e o tamanho de bloco $m = 4$:

$$\begin{aligned} h(w) &= p(a) * 7^{0 \bmod 4} + p(b) * 7^{1 \bmod 4} + p(c) * 7^{2 \bmod 4} + p(d) * 7^{3 \bmod 4} \\ &= 1 * 7^0 + 2 * 7^1 + 3 * 7^2 + 4 * 7^3 \\ &= 1 * 1 + 2 * 7 + 3 * 49 + 4 * 343 \\ &= 1534 \end{aligned}$$

Outro exemplo com $w = \text{bcefaa}$ e $m = 3$:

$$
\begin{aligned}
h(w) &= p(b) * 7^{0 \bmod 3} + p(c) * 7^{1 \bmod 3} + p(e) * 7^{2 \bmod 3} + p(f) * 7^{3 \bmod 3} + p(a) * 7^{4 \bmod 3} + p(a) * 7^{5 \bmod 3} \\
&= 2 * 7^0 + 3 * 7^1 + 5 * 7^2 + 6 * 7^0 + 1 * 7^1 + 1 * 7^2 \\
&= 2 * 1 + 3 * 7 + 5 * 49 + 6 * 1 + 1 * 7 + 1 * 49 \\
&= 330
\end{aligned}
$$

Análise

Essa aproximação transforma a linguagem reconhecida de uma que reconhece palavras iguais para uma que reconhece palavras com a mesma soma total de seus blocos de tamanho m. Por exemplo, para $m = 3$, abcdef = defabc. Assim, quanto menor for m, maior será a taxa de colisões, com $m = 1$ sendo equivalente à primeira iteração do algoritmo. Nesse mesmo sentido, na medida que $m \to \inf$, a linguagem se aproxima do comportamento ideal ww^r. Vale notar que também seria possível um algoritmo que enumera as possíveis permutações de um bloco através de estados, mas esse resultaria em uma explosão combinatória do número de estados. Veremos que o valor de m influencia de forma linear a quantidade de estados da nossa solução.

Implementação

Iremos modelar um sistema simples de cadastro de senha, na qual é preciso primeiro definir a senha e depois confirmá-la. O sistema possui apenas duas ações: escrever a senha, utilizando os símbolos do alfabeto de entrada, e o *enter*, para sinalizar o final da senha, e será representado por um símbolo extra.

Um alfabeto de entrada simples será utilizado por motivos de praticidade, mas é possível integrar esse sistema à primeira etapa do trabalho para serem permitidas apenas senhas fortes, visto que a interseção entre uma LLC e uma linguagem regular sempre resulta em outra LLC. Também é preciso escolher um valor de m, que definirá a complexidade e precisão do sistema. Será mostrado um método de construção de um autômato para um m arbitrário, mas a implementação usará um m pequeno, também por motivos de simplicidade.

Serão usadas as letras *a* a *i* como alfabeto de entrada + um símbolo para denotar o fim da palavra / ação de registrar a senha (usaremos o símbolo '#'), então: $\Sigma = \{a, b, c, d, e, f, g, h, i, \#\}$, a base $b = 10$, $m = 3$ e $p(x) =$ posição de x no alfabeto. Os estados serão divididos em duas fases: cadastro e confirmação. A primeira fase será responsável por ler os símbolos da primeira palavra e empilhar seu valor correspondente, enquanto a segunda fase lerá os símbolos da segunda palavra e desempilhar seus valores. A condição de aceite será ter uma pilha vazia ao fim da entrada. Assim, a linguagem ideal que queremos aproximar é $L = \{w\#w\#\}$. O alfabeto da pilha será apenas o símbolo para representar o número unário e um símbolo para o início da pilha: $\Gamma = \{x, Z\}$.

O número de estados será $2m + 1$ (guardar a posição atual do bloco em cada fase, mais um estado final). Em ambas as fases, cada estado q_i lerá o próximo símbolo da palavra, empilhará/desempilhará o número de dígitos unários correspondentes, e passará para o estado $q_{(i+1) \bmod m}$. Caso encontrem o símbolo '#', a primeira fase passará para a segunda, e a segunda irá aceitar ou rejeitar a entrada, a depender do estado da pilha. Os números a serem manipulados, após serem precalculados, podem ser modelados com a seguinte tabela:

q_i	a	b	c	d	e	f	g	h	i
0	x^1	x^2	x^3	x^4	x^5	x^6	x^7	x^8	x^9
1	x^{10}	x^{20}	x^{30}	x^{40}	x^{50}	x^{60}	x^{70}	x^{80}	x^{90}
2	x^{100}	x^{200}	x^{300}	x^{400}	x^{500}	x^{600}	x^{700}	x^{800}	x^{900}

Table 1: Exemplo de tabela de números precalculados para $m = 3$

Algumas Palavras Aceitas e Rejeitadas

Aceitas:

- abc#abc#
- abcdef#abcdef#
- abcdef#defabc#
- gabi#gabi#
- aaaaaa#bbb#
- ccc#bbbaaa#
- abcdefghi#abcdefghi#

Rejeitadas:

- abc#cba#
- abc#def#
- aaabbb#ababab#
- abcdefghi#aacdefghi#
- abcdefghi#abdefghi#

Prova de Irregularidade por Bombeamento

Provaremos a irregularidade dessa linguagem L através do lema do bombeamento:

1. Assumimos que existe uma constante $p \geq 1$ como o lema pede.
2. Utilizaremos $w \in L = a^p \# a^p \#$, que é maior que p.
3. Como as duas palavras possuem apenas o mesmo símbolo repetido a, suas somas só serão iguais caso o número de a's sejam iguais, independente de m.
4. Pelo lema do bombeamento, deve existir uma decomposição $w = xyz$ com $|xy| \leq p$ e $|y| \geq 1$ tal que $xy^i z \in L$ para todo $i \geq 0$.
5. Como $|xy| \leq p$, y contém apenas a's pertencentes à primeira palavra.
6. Bombeando y para termos xy^2z nos dá um resultado com mais a's na primeira palavra do que na segunda, pois a segunda palavra permanece inalterada.
7. Logo, xy^2z não está em L, pois a soma das duas palavras não é a mesma.
8. Logo, L não é regular.

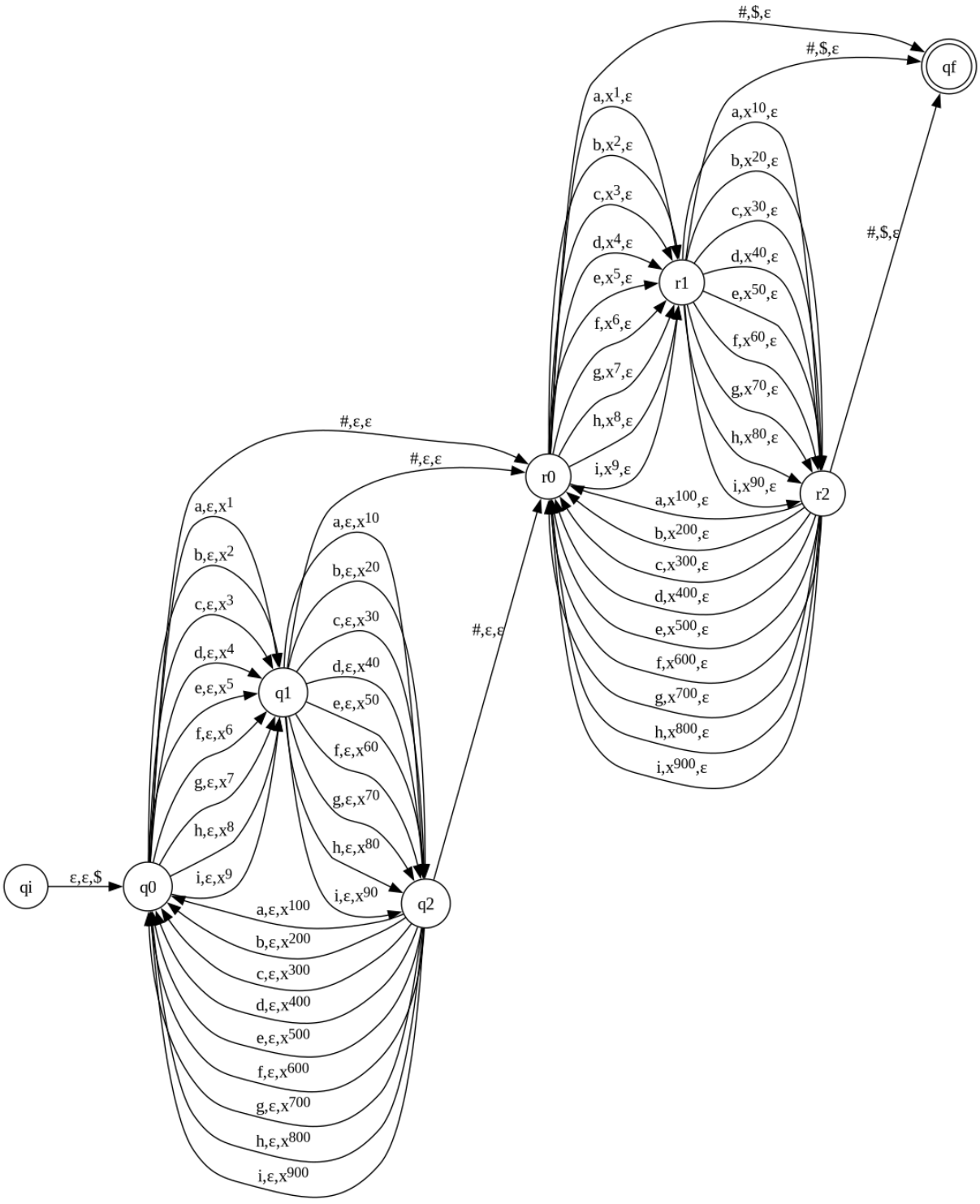

Figure 1: Diagrama do autômato reconhecedor para $m = 3$

A Gramática

Em seguida será definida a GLC que descreve o sistema. A ideia por trás da gramática é utilizar as variáveis para guardar a diferença atual entre os valores das duas palavras. As únicas variáveis com uma produção terminal são as que possuem uma diferença de 0, representando uma entrada balanceada.

Para casos de $m > 1$, precisamos também guardar o expoente do próximo símbolo de cada palavra. As variáveis terão o formato $V_{\Delta,i,j}$, onde $\Delta = h(w_2) - h(w_1)$ (a diferença entre os valores das palavras) e i, j são os "contadores" módulo m dos expoentes das palavras w_1 e w_2. A variável inicial é $V_{0,0,m-1}$.

Cada variável terá uma produção para cada símbolo "dígito" da entrada. As produções são construídas de forma que Δ se movimente sempre em direção à 0, então variáveis da forma $V_{\Delta>0,i,j}$ terão produções que adicionam à w_1, e vice-versa. A magnitude desse "movimento" depende do dígito que será produzido e o contador do expoente atual. As produções então têm um dos seguintes formatos:

$$\Delta < 0 : \begin{cases} V_{\Delta,i,j} \to V_{\Delta',i,j'}a \\ \Delta' = \Delta + p(a) * b^j \\ j' = (j-1) \bmod m \\ a \in T - \# \end{cases} \qquad \Delta \geq 0 : \begin{cases} V_{\Delta,i,j} \to aV_{\Delta',i',j} \\ \Delta' = \Delta - p(a) * b^i \\ i' = (i+1) \bmod m \\ a \in T - \# \end{cases}$$

Dessa forma, a diferença entre as duas palavras nunca ultrapassará b^m para uma base b. Assim conseguimos manter o número de variáveis finito. O tamanho da gramática final é de aproximadamente $2m * b^m$ variáveis e $2m * b^m(b-1)$ produções. Por exemplo, uma gramática com base $b = 10$ e tamanho de bloco $m = 3$ terá ~ 6.000 variáveis e ~ 60.000 produções. Por causa disso, juntamente com o fato de que a ferramenta JFLAP não suporta gramáticas com mais de 26 variáveis, o arquivo *.jff* da GLC será feito com base $b = 10$ e $m = 1$. Entretanto, como o autômato possui uma estrutura mais simples, também será anexado um arquivo *.jff* do autômato feito com base $b = 10$ e $m = 3$.

Vale notar que esse método constrói uma gramática deterministica, o que não é surpreendente, dado que o autômato reconhecedor também é determinístico. Um algorítmo como o reconhecedor de *Earley* é então capaz de processar uma palavra dessa gramática em *O(n)* passos.

Segue a gramática resultante para base $b = 4$ e tamanho de bloco $m = 2$ com terminais $T = \{1, 2, 3, \#\}$, $p(x) = x$, e variável inicial $V_{0,0,1}$:

$V_{0,0,1} \to 1V_{-1,1,1} \mid 3V_{-3,1,1} \mid 2V_{-2,1,1} \mid \#$
$V_{0,1,1} := 3V_{-12,0,1} \mid 1V_{-4,0,1} \mid 2V_{-8,0,1} \mid \#$
$V_{0,1,0} := 3V_{-12,0,0} \mid 2V_{-8,0,0} \mid 1V_{-4,0,0}$
$V_{0,0,0} := 1V_{-1,1,0} \mid 2V_{-2,1,0} \mid 3V_{-3,1,0}$
$V_{-12,0,0} := V_{-11,0,1}1 \mid V_{-9,0,1}3 \mid V_{-10,0,1}2$
$V_{-12,0,1} := V_{-8,0,0}1 \mid V_{-4,0,0}2 \mid V_{0,0,0}3$
$V_{-11,0,0} := V_{-9,0,1}2 \mid V_{-8,0,1}3 \mid V_{-10,0,1}1$
$V_{-11,0,1} := V_{-7,0,0}1 \mid V_{-3,0,0}2 \mid V_{1,0,0}3$
$V_{-10,0,0} := V_{-8,0,1}2 \mid V_{-9,0,1}1 \mid V_{-7,0,1}3$
$V_{-10,0,1} := V_{2,0,0}3 \mid V_{-6,0,0}1 \mid V_{-2,0,0}2$
$V_{-9,0,1} := V_{-1,0,0}2 \mid V_{-5,0,0}1 \mid V_{3,0,0}3$
$V_{-9,0,0} := V_{-6,0,1}3 \mid V_{-8,0,1}1 \mid V_{-7,0,1}2$

$V_{-8,0,0} := V_{-7,0,1}1 \mid V_{-6,0,1}2 \mid V_{-5,0,1}3$
$V_{-8,0,1} := V_{4,0,0}3 \mid V_{-4,0,0}1 \mid V_{0,0,0}2$
$V_{-7,0,0} := V_{-5,0,1}2 \mid V_{-4,0,1}3 \mid V_{-6,0,1}1$
$V_{-7,0,1} := V_{-3,0,0}1 \mid V_{1,0,0}2 \mid V_{5,0,0}3$
$V_{-6,0,0} := V_{-5,0,1}1 \mid V_{-3,0,1}3 \mid V_{-4,0,1}2$
$V_{-6,0,1} := V_{-2,0,0}1 \mid V_{2,0,0}2 \mid V_{6,0,0}3$
$V_{-5,0,0} := V_{-3,0,1}2 \mid V_{-4,0,1}1 \mid V_{-2,0,1}3$
$V_{-5,0,1} := V_{7,0,0}3 \mid V_{3,0,0}2 \mid V_{-1,0,0}1$
$V_{-4,0,1} := V_{0,0,0}1 \mid V_{4,0,0}2 \mid V_{8,0,0}3$
$V_{-4,0,0} := V_{-2,0,1}2 \mid V_{-1,0,1}3 \mid V_{-3,0,1}1$
$V_{-3,1,0} := V_{-1,1,1}2 \mid V_{0,1,1}3 \mid V_{-2,1,1}1$
$V_{-3,1,1} := V_{1,1,0}1 \mid V_{5,1,0}2 \mid V_{9,1,0}3$
$V_{-3,0,0} := V_{-1,0,1}2 \mid V_{-2,0,1}1 \mid V_{0,0,1}3$
$V_{-3,0,1} := V_{9,0,0}3 \mid V_{1,0,0}1 \mid V_{5,0,0}2$
$V_{-2,1,1} := V_{6,1,0}2 \mid V_{2,1,0}1 \mid V_{10,1,0}3$
$V_{-2,0,1} := V_{2,0,0}1 \mid V_{10,0,0}3 \mid V_{6,0,0}2$
$V_{-2,1,0} := V_{-1,1,1}1 \mid V_{0,1,1}2 \mid V_{1,1,1}3$
$V_{-2,0,0} := V_{-1,0,1}1 \mid V_{1,0,1}3 \mid V_{0,0,1}2$
$V_{-1,1,0} := V_{1,1,1}2 \mid V_{0,1,1}1 \mid V_{2,1,1}3$
$V_{-1,0,1} := V_{11,0,0}3 \mid V_{7,0,0}2 \mid V_{3,0,0}1$
$V_{-1,0,0} := V_{2,0,1}3 \mid V_{1,0,1}2 \mid V_{0,0,1}1$
$V_{-1,1,1} := V_{7,1,0}2 \mid V_{11,1,0}3 \mid V_{3,1,0}1$
$V_{1,1,1} := 3V_{-11,0,1} \mid 1V_{-3,0,1} \mid 2V_{-7,0,1}$
$V_{1,1,0} := 1V_{-3,0,0} \mid 2V_{-7,0,0} \mid 3V_{-11,0,0}$
$V_{1,0,1} := 1V_{0,1,1} \mid 2V_{-1,1,1} \mid 3V_{-2,1,1}$
$V_{1,0,0} := 2V_{-1,1,0} \mid 3V_{-2,1,0} \mid 1V_{0,1,0}$
$V_{2,0,0} := 1V_{1,1,0} \mid 2V_{0,1,0} \mid 3V_{-1,1,0}$
$V_{2,0,1} := 2V_{0,1,1} \mid 3V_{-1,1,1} \mid 1V_{1,1,1}$
$V_{2,1,0} := 3V_{-10,0,0} \mid 1V_{-2,0,0} \mid 2V_{-6,0,0}$
$V_{2,1,1} := 1V_{-2,0,1} \mid 3V_{-10,0,1} \mid 2V_{-6,0,1}$
$V_{3,1,0} := 2V_{-5,0,0} \mid 3V_{-9,0,0} \mid 1V_{-1,0,0}$
$V_{3,0,0} := 3V_{0,1,0} \mid 1V_{2,1,0} \mid 2V_{1,1,0}$
$V_{4,0,0} := 1V_{3,1,0} \mid 3V_{1,1,0} \mid 2V_{2,1,0}$
$V_{4,1,0} := 1V_{0,0,0} \mid 2V_{-4,0,0} \mid 3V_{-8,0,0}$
$V_{5,0,0} := 1V_{4,1,0} \mid 2V_{3,1,0} \mid 3V_{2,1,0}$
$V_{5,1,0} := 2V_{-3,0,0} \mid 3V_{-7,0,0} \mid 1V_{1,0,0}$
$V_{6,1,0} := 2V_{-2,0,0} \mid 3V_{-6,0,0} \mid 1V_{2,0,0}$
$V_{6,0,0} := 1V_{5,1,0} \mid 3V_{3,1,0} \mid 2V_{4,1,0}$

$V_{7,1,0} \coloneqq 2V_{-1,0,0} \mid 1V_{3,0,0} \mid 3V_{-5,0,0}$
$V_{7,0,0} \coloneqq 1V_{6,1,0} \mid 2V_{5,1,0} \mid 3V_{4,1,0}$
$V_{8,0,0} \coloneqq 1V_{7,1,0} \mid 2V_{6,1,0} \mid 3V_{5,1,0}$
$V_{8,1,0} \coloneqq 2V_{0,0,0} \mid 3V_{-4,0,0} \mid 1V_{4,0,0}$
$V_{9,0,0} \coloneqq 3V_{6,1,0} \mid 1V_{8,1,0} \mid 2V_{7,1,0}$
$V_{9,1,0} \coloneqq 1V_{5,0,0} \mid 2V_{1,0,0} \mid 3V_{-3,0,0}$
$V_{10,0,0} \coloneqq 1V_{9,1,0} \mid 3V_{7,1,0} \mid 2V_{8,1,0}$
$V_{10,1,0} \coloneqq 2V_{2,0,0} \mid 3V_{-2,0,0} \mid 1V_{6,0,0}$
$V_{11,1,0} \coloneqq 2V_{3,0,0} \mid 3V_{-1,0,0} \mid 1V_{7,0,0}$
$V_{11,0,0} \coloneqq 2V_{9,1,0} \mid 1V_{10,1,0} \mid 3V_{8,1,0}$