Proposta de pré-processamento para fins de padronização de fórmulas matemáticas

Matheus A. Silveira[1], **Rodrigo de O. Figueiredo**[1], **Flavio B. Gonzaga**[1]

[1] Departamento de Ciência da Computação — Universidade Federal de Alfenas (UNIFAL-MG)
CEP 37.133-840 — Alfenas — MG — Brasil

{a16041,a14026,fbgonzaga}@bcc.unifal-mg.edu.br

Abstract. *Due to great content search on the internet, tools for such a requirement need constant development and refinement for a better user experience. Many file types as texts, media, articles, extending to mathematical expressions are fetched in great volume. The latter require more research to develop algorithms that allow a retrieval of mathematical expressions with more efficiency. To test the mathematical content retrieval algorithms developed, large-scale entries are required, consistent with real-time search. The paper's propose is to produce entries for the Structure Based Longest Common Subsequence (SLCS) algorithm in its parallel version that uses CUDA (Compute Unified Device Architecture), in order to find the feasibility of this mathematical's form content retrieval in the future, when joining the algorithm to the inputs. For this purpose, the JAVA language was used to preprocess mathematical expressions written in TEX from seven databases and to build a standardized information structure that will be used by the algorithm related to this work.*

Resumo. *Com crescentes buscas realizadas na internet, ferramentas para tal necessidade precisam de constante desenvolvimento e aperfeiçoamento para uma melhor experiência do usuário. Diversos tipos de arquivos, dentre eles textos, mídias, artigos, se estendendo até fórmulas matemáticas, são buscados em grande volume. Estas últimas necessitam de maiores pesquisas para o desenvolvimento de algoritmos que possibilitem recuperar expressões matemáticas com maior eficiência. Para testar os algoritmos de recuperação de conteúdo matemático desenvolvidos, são necessárias entradas em larga escala, condizente com a busca em tempo real. A proposta deste trabalho consiste em produzir entradas para o algoritmo Structure Based Longest Common Subsequence (SLCS) em sua versão paralela que utiliza CUDA (Compute Unified Device Architecture), com a finalidade de, no futuro, ao unir o algoritmo às entradas, descobrir a viabilidade desta forma de recuperação de conteúdo matemático. Para isto utilizou-se da linguagem JAVA com o objetivo de pré-processar fórmulas matemáticas escritas em TEX provindas de sete bases de dados e construir uma estrutura de informação padronizada, que será utilizada pelo algoritmo relacionado à este presente trabalho.*

1. Introdução

Com a crescente procura por informações na Internet, mecanismos de pesquisas são aperfeiçoados constantemente, visando levar resultados de forma rápida e concisa ao

usuário. Neste contexto, são buscadas informações que se estendem desde textos, imagens, vídeos, artigos científicos, até fórmulas matemáticas, sendo esta última o foco deste trabalho.

Para propiciar melhores resultados, diversos métodos e algoritmos para recuperação de informações foram desenvolvidos ao longo da evolução dos mecanismos de buscas. Um dos principais algoritmos é o *Longest Common Subsequence* (LCS) [Cormen et al. 2009], o qual propõe um mecanismo de comparação de *strings* que, dadas duas sequências, seu objetivo é encontrar a subsequência comum mais longa entre as mesmas. Por definição, uma subsequência Y é obtida a partir de uma sequência X, omitindo alguns dos elementos de X. Assim sendo, a subsequência Y contém apenas os elementos que também estão presentes na sequência X e a ordenação original de tais elementos também é mantida.

O algoritmo *Structure Based Longest Common Subsequence* (SLCS) [Kumar et al. 2012], é uma versão modificada do algoritmo LCS. No referido trabalho, as fórmulas matemáticas são primeiro pré-processadas, com o objetivo de realizar uma padronização (variáveis são trocadas por V, números por N, dentre algumas outras conversões). Uma vez realizada essa etapa, o SLCS compara as *strings* resultantes, com o objetivo de descobrir, dada uma consulta, quais seriam as fórmulas mais semelhantes existentes no banco de dados. O algoritmo recebe o nome de *Structure Based* porque considera no cálculo da semelhança a estrutura da fórmula, como elementos em subscrito e sobrescrito. Por exemplo, em $\sum_{i=0}^{10}$, o somatório está no nível inicial, e os seus limites inferior e superior aparecem como elementos subscrito e sobrescrito respectivamente.

No trabalho de [Miya et al. 2018] foi implementada uma versão paralela do algoritmo SLCS, utilizando a plataforma CUDA (*Compute Unified Device Architecture*) da NVIDIA Corporation. No trabalho mencionado, no entanto, a base de dados a ser buscada foi construída de maneira artificial. Seus autores selecionaram cinco fórmulas matemáticas, e as replicaram um número grande de vezes, com o objetivo de simular uma base de dados suficientemente grande. Nosso trabalho teve por objetivo construir uma base de dados real, com fórmulas obtidas a partir de sete bases de dados. Após a obtenção, é proposta ainda uma etapa de pré-processamento das fórmulas, para que as mesmas possam (em um trabalho futuro) serem testadas utilizando-se do algoritmo em CUDA.

2. Referencial Teórico

Esta seção oferece detalhes sobre o funcionamento dos algoritmos desenvolvidos por [Cormen et al. 2009], [Kumar et al. 2012] e por [Miya et al. 2018], bem como sobre o método proposto para a realização do pré-processamento das fórmulas matemáticas descrito em [Kumar et al. 2012].

2.1. Algoritmos

2.1.1. *Longest Common Subsequence* (LCS)

O algoritmo LCS [Cormen et al. 2009] é um algoritmo em que, passadas duas sequências de *strings*, sua finalidade é encontrar a subsequência comum mais longa entre as mesmas.

Uma sequência Z será uma subsequência comum entre duas sequências X e Y, se Z for subsequência de X e Y simultaneamente. Como exemplo citado pelos autores, dadas duas sequências X=(A,B,C,B,D,A,B) e Y=(B,D,C,A,B,A), a sequência (B,C,A) é uma subsequência comum entre X e Y. Porém não é a subsequência comum mais longa(LCS), pois possui comprimento 3, enquanto as sequências (B,C,B,A) e (B,D,A,B) possuem comprimento 4, e também são subsequências de X e Y. Como não há outra subsequência de comprimento 5 ou maior entre X e Y, as duas subsequências de comprimento 4 são ditas as LCS entre X e Y.

O algoritmo utiliza duas matrizes para realizar suas operações e encontrar a subsequência comum mais longa entre duas sequências X e Y. A matriz c[0 .. m, 0 .. n], onde n e m são as quantidades de elementos de X e Y respectivamente, é usada para encontrar o tamanho da maior subsequência comum, armazenando os valores obtidos através da equação F1. Por outro lado, a matriz b[1 .. m, 1 .. n] auxilia na construção da subsequência propriamente dita, indicando os elementos que são comuns.

$$c[i,j] = \begin{cases} 0 & se\ i = 0\ ou\ j = 0, \\ c[i-1, j-1] + 1 & se\ X[i] = Y[j], \\ max(c[i, j-1],\ c[i-1, j]) & se\ X[i] \neq Y[j]. \end{cases} \quad \text{(F1)}$$

Como visto na equação F1 a primeira linha e a primeira coluna da matriz c é preenchida com zeros. Iniciando a comparação das sequências, da segunda condição infere-se que, se os caracteres são iguais, ou seja, fazem parte da LCS, c[i,j] receberá o valor da posição anterior diagonal mais um (c[i-1,j-1] + 1). Em contrapartida, se os caracteres comparados forem diferentes, c[i,j] receberá o valor máximo entre o valor da posição anterior da linha i (c[i,j-1]) e o valor da posição anterior da coluna j (c[i-1,j]).

De modo semelhante, a matriz b utiliza as mesmas condições da equação F1 para armazenar as direções das origens das atribuições. Se X[i] = Y[j], então b[i,j]=↖, se não, se c[i-1,j] $\geq$ c[i,j-1], então b[i,j]=↑, se não, então b[i,j]=←. Ao final desse processo, iniciando em b[m,n], é possível montar a subsequência encontrada seguindo o caminho indicado pelas setas.

A Figura 1 ajuda à ilustrar as explicações sobre o algoritmo LCS, juntamente com o exemplo citado previamente.

2.1.2. *Structure Based* LCS (SLCS)

A abordagem apresentada por [Kumar et al. 2012] propõe utilizar o algoritmo SLCS, para realizar o ranqueamento de expressões matemáticas. O autor propõe que as fórmulas sejam pré-processadas, de modo que elementos semelhantes, como variáveis, números, funções, etc, e seus níveis sejam padronizados. Após essa etapa, é gerada então a base de dados que será usada como entrada para o algoritmo SLCS.

No algoritmo SLCS, dois termos são ditos correspondentes, se forem iguais entre si e compartilharem do mesmo nível. A função *score*, mostrada em F2, obtém a pontuação das fórmulas matemáticas segundo os seus níveis. Ao analisarmos, observamos a existência da função $l(x)$, a qual corresponde ao nível do termo x. Se os níveis forem iguais, então $|l(Q[i]) - l(D[j])| = 0$. Desse modo, o valor retornado pela função

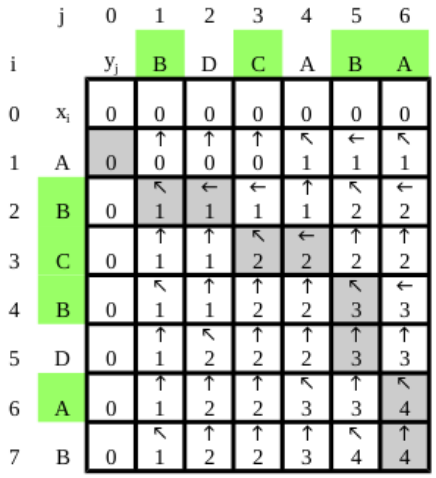

Figura 1. Cálculo das matrizes c e b sobre as sequências X = (A, B, C, B, D, A, B) e Y = (B, D, C, A, B, A). A matriz c é representada pelos algarismos, e a matriz b pelas setas. As duas matrizes foram unificadas em uma única imagem. As células sombreadas representam o caminho percorrido no processo de construção da LCS. Os elementos destacados em verde são os que compõem a LCS. A matriz na célula da linha 7 e coluna 6 contém o comprimento da LCS.

Fonte: retirado de [Miya et al. 2018] baseado em [Cormen et al. 2009].

será 1. Por outro lado, quando os níveis dos termos $Q[i]$ e $D[j]$ forem diferentes, então: $0 < score(Q[i], D[j]) \leq 0,5$.

$$score(Q[i], D[j]) = \frac{1}{|l(Q[i]) - l(D[j])| + 1} \tag{F2}$$

Pela função do *score*, conclui-se que, no algoritmo SLCS, quanto maior a diferença entre os termos $Q[i]$ e $D[j]$ das fórmulas matemáticas, menor será o valor retornado. Além disso, os níveis de cada termo das fórmulas matemáticas também são considerados, juntamente com a fórmula matemática propriamente dita. O SLCS é dado por F3.

$$SLCS[i,j] = \begin{cases} 0 & se\ i = 0\ ou\ j = 0, \\ SLCS[i-1, j-1] + score(Q[i], D[j]) & se\ Q[i] = D[j], \\ max(SLCS[i, j-1],\ SLCS[i-1, j]) & se\ Q[i] \neq D[j]. \end{cases} \tag{F3}$$

Como observado na equação F3, a primeira e a terceira operação são iguais às operações da F1, do LCS, assim como as suas condições. A diferença está na segunda operação, a qual é adicionada o valor da função *score*, ao invés do valor 1 como feito no LCS. Isto propicia levar em consideração os termos, juntamente com os seus respectivos níveis, em uma tarefa de medir qual a semelhança entre duas fórmulas matemáticas.

2.1.3. SLCS em CUDA

Em [Miya et al. 2018] foi realizada a implementação do algoritmo SLCS de modo paralelo, utilizando-se da técnica GPGPU (*General Purpose Graphics Processing Unit*),

que possibilita tirar proveito do *hardware* de GPU's (Graphics Processing Unit) para computação de grande volume de informações sobre dados genéricos, pois GPU's possuem milhares de núcleos, projetados para operar de forma paralela. A plataforma de computação paralela CUDA da proprietária NVIDIA foi a escolhida pelos autores para o desenvolvimento do referido trabalho.

Na abordagem de [Miya et al. 2018], foram apresentadas três implementações diferentes para o algoritmo SLCS paralelo. A primeira implementação, segue conceitos citados em [Kumar et al. 2012] e [Balhaf et al. 2016]. Este último, construiu um algoritmo utilizando a plataforma CUDA para o cálculo da distância de Levenshtein, que consiste na distância de edição entre duas sequências de caracteres, ou seja, visa descobrir qual é o número mínimo de inserções, remoções e substituições a serem realizadas para que uma cadeia de caracteres A seja igual a outra cadeia de caracteres B [Wikipedia contributors 2019]. Visando eliminar a dependência de dados existente na matriz de programação dinâmica, o autor propôs a técnica de rastreamento em diagonal da matriz, que em síntese, consiste em: a cada iteração, todos os elementos de uma diagonal específica são obtidos em paralelo, como ilustrado na Figura 2. Utilizando-se destes conceitos, a primeira implementação tem como foco a paralelização interna de uma única tarefa, aplicando a técnica do rastreamento em diagonal na comparação de duas fórmulas matemáticas.

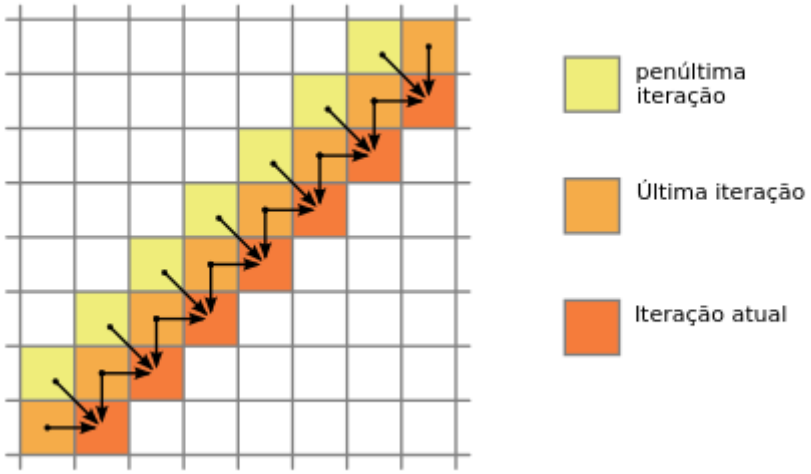

Figura 2. Dependências na matriz e as diagonais.
Fonte: retirado de [Miya et al. 2018] baseado em [Bednárek et al. 2017].

Em contraste, a segunda implementação tem a finalidade de paralelizar o algoritmo SLCS a nível de tarefas. Dessa forma, dada uma consulta Q, esta é comparada com cada expressão do banco de dados, sendo cada comparação considerada uma única tarefa. Todas estas tarefas são independentes umas das outras, e em decorrência disso, é aplicado o algoritmo SLCS de forma sequencial sobre cada tarefa, ou seja, o preenchimento da matriz dinâmica é feito linha por linha, partindo da primeira posição à esquerda.

A terceira implementação visa o aumento de paralelismo, e com o intuito de obtê-lo, é aplicado o paralelismo em cada tarefa separadamente e entre tarefas, unindo assim, as duas implementações anteriores. Para promover tal evento, é utilizado o aninhamento de

Kernel, onde o *Kernel* pai, que contém todas as comparações entre fórmulas matemáticas a serem realizadas, possui vários *Kernel* filhos, cada um contendo uma comparação a ser realizada. Todas as comparações entre o *Kernel* pai e os *Kernel* filhos podem ser executados simultâneamente, já que são independentes, gerando assim, o paralelismo a nível de tarefas, conceito utilizado na segunda implementação. Dentro de cada *Kernel* filho, é utilizada a técnica do rastreamento em diagonal, que promove o paralelismo interno da respectiva tarefa, conceito este utilizado na primeira implementação.

Segundo os testes realizados, a segunda implementação, que utiliza de paralelismo a nível de tarefas, obteve melhores resultados. O menor desempenho das outras implementações ocorreu devido à dependência de dados causada pela recursividade da técnica LCS apresentada por [Kumar et al. 2012] e à sobrecarga gerada pela técnica de rastreamento em diagonal. Apesar de ter sido incluída para amenizar a dependência de dados, a técnica de [Balhaf et al. 2016] não se mostrou satisfatória no cenário do trabalho apresentado.

2.2. Pré-processamento das fórmulas

No artigo "A Structure Based Approach for Mathematical Expression Retrieval" [Kumar et al. 2012], é apresentada uma estrutura de informação padrão para a entrada do algoritmo SLCS, obtida na etapa de pré-processamento. Essa estrutura padronizada foi a base para o desenvolvimento deste trabalho. Tal mecanismo visa padronizar as fórmulas matemáticas de entrada, escritas em TEX, para posteriormente maximizar a eficiência do algoritmo em medir a semelhança entre fórmulas diferentes.

Na abordagem apresentada pelos autores do artigo citado, as expressões utilizadas têm formato TEX. Portanto, o primeiro passo para transformação das fórmulas foi eliminar palavras com pouco ou nenhum significado matemático, como \displaystyle, \begin{array}, etc.

Posteriormente, foi criada uma tabela mapeando funções escritas em TEX para números. Por exemplo, o termo \sqrt, que representa a função da raiz quadrada, foi mapeado para o número 301. Assim também ocorre com todas as funções relevantes padronizadas em TEX.

Para que elementos semelhantes sejam encontrados dentro das fórmulas matemáticas, variáveis e constantes foram mapeados para as letras V e N, respectivamente. Por exemplo, as fórmulas $cos^2x + sin^2x = 1$ e $cos^2y + sin^2y = 1$ representam a mesma fórmula, porém, possuem variáveis diferentes (x e y). Desse modo, substituindo as variáveis x e y por V, verificamos que as duas fórmulas são equivalentes. Isso também pode ocorrer com letras gregas, além das outras letras de outros alfabetos. Em outro exemplo, temos $tan(30)$, que é uma instância de $tan(x)$, para $x = 30$. Se substituímos 30 por N, isso permite que a expressão citada seja correspondente a qualquer instância da função $tan(x)$.

Outro fator que foi levado em consideração, são as informações de estrutura de uma fórmula matemática, que são: sobrescrito e subscrito. Em TEX essas possibilidades são representadas pelo acento circunflexo e *underline*, respectivamente. Para denotá-los, foram utilizados quatro termos de estrutura: P_s e P_e, para o início e o fim de sobrescritos, bem como, B_s e B_e para o início e o fim de subscritos. Por

exemplo, a expressão $\sqrt[n]{x_1}$, cujo código em TEX é \sqrt[n]{x_1}, foi traduzida para: $301, P_s, V, P_e, B_s, V, B_s, N, B_e, B_e$.

Os últimos elementos tratados foram as linhas e colunas, que são essencias em matrizes. Assim sendo, cada linha foi representada por R e cada adição de uma coluna na linha foi trocada por C. Por exemplo, dada a matriz $\begin{bmatrix} x & y \\ x_1 & y_2 \end{bmatrix}$, cujo código em TEX é \left [\begin{array}{cc} x & y \\x_1 & y_2 \end{array} \right], temos como *string* pré-processada: $201, V, C, V, R, V, B_s, N, B_e, C, V, B_s, N, B_e, R, 202$

Como o que diferencia expressões matemáticas de *strings* é a bidimensionalidade das expressões, [Kumar et al. 2012] utilizaram os níveis das expressões como critério de comparação em seu algoritmo. Para obter tais níveis, basta observar a seguinte característica: se um elemento está sobrescrito ou subscrito ele possui nível maior ou igual a 1, se não, ele possui nível 0. No exemplo citado anteriormente, a expressão matemática $\sqrt[n]{x_1}$, cuja *string* pré-processada é dada por: $301, P_s, V, P_e, B_s, V, B_s, N, B_e, B_e$, dispõe dos seguintes níveis: 0,1,1,1,1,1,2,2,1.

Apesar de não terem sido disponibilizadas as bases de dados utilizadas na execução do algoritmo, tanto a de termos em TEX como a de fórmulas matemáticas, a qual o artigo cita como 829 expressões matemáticas em formato TEX, no decorrer do artigo os autores apresentam vários exemplos de fórmulas pré-processadas e seus devidos procedimentos de padronização. Alguns deles foram mencionados nesta sessão.

3. Metodologia e Resultados

Baseando-se na estrutura de informação padrão proposta em [Kumar et al. 2012], nosso trabalho teve como objetivo a produção de entradas padronizadas para o algoritmo SLCS em sua versão paralela [Miya et al. 2018]. Sua implementação foi realizada na linguagem JAVA, devido à praticidade no tratamento de *strings* que a linguagem fornece.

Para facilitar o entendimento dos processos de transformação das fórmulas matemáticas escritas em TEX para o resultado produzido após o pré-processamento aqui proposto, é importante apresentar algumas informações utilizadas na padronização das fórmulas:

- Variáveis foram representadas pela letra V;
- números inteiros foram representados pela letra N, e números não inteiros foram representados por R;
- funções e símbolos matemáticos obtidos através de termos escritos em TEX, relevantes no contexto matemático, foram representados por números inteiros;
- por último, em matrizes, vetores e outros elementos matemáticos que utilizam dos mesmos artifícios, L indica a presença de uma nova linha e C o início de uma nova coluna.

A etapa inicial realizada para possibilitar a padronização das fórmulas consistiu na coleta de termos TEX que são importantes no contexto matemático, como funções, símbolos matemáticos, operadores, etc. Organizando-os em uma tabela, ao todo, foram contabilizados 811 termos matemáticos distintos, os quais foram extraídos das seguintes referências: [Carlisle et al. 2001], [Sermutlu 2006], [Carlisle 1992], [Gratzer 1995].

Tais termos foram separados em três categorias:

- 1ª categoria: Letras do alfabeto Grego e Hebraico como α (\alpha), $\aleph$(\aleph) e outras, que geralmente são utilizadas para representar variáveis no contexto matemático;
- 2ª categoria: Termos importantes no contexto matemático, que são utilizados em áreas como o Cálculo, Geometria, Álgebra, Lógica e a matemática geral. Alguns exemplos são: Limite (\lim), Seno (\sin), Para Todo (\forall) e Raiz (\sqrt);
- 3ª categoria: Termos geralmente utilizados em contexto matemático, porém que não acrescentam significado na fórmula final pré-processada. Como exemplo, podem ser citados os termos \left e \right.

Cada termo da primeira categoria foi mapeado para a letra V, representando variáveis. Os da segunda categoria foram mapeados para números inteiros e os da terceira, para espaço em branco, como por exemplo "\Big", que vem sempre acompanhado de um delimitador, como o colchete ("\Big["), e que não agregaria significado ao resultado final da fórmula matemática pré-processada. Para evitar a ocorrência de traduções errôneas de termos que possuem prefixos idênticos, os mesmos foram ordenados de forma decrescente em relação ao comprimento. Como exemplo de tradução errônea, considerando que letras são substituídas por "V", o termo \subseteq por 279 e o termo \subset por 401, a expressão: $A \subseteq B$, cujo código TEX corresponde é "A \subseteq B", e sua tradução correta se dá por "V 279 V", poderia ser erroneamente traduzida para "V 401V V", se o termo "\subset", de comprimento menor ocorresse antes de "\subseteq"na tabela de termos.

A fim de auxiliar no processamento das fórmulas, foram criadas duas classes JAVA, representadas na Figura 3. Uma delas, chamada "EstruturaTermo.java", tem como atributos: "termo", que armazena um termo no formato TEX; e "id", que recebe o código referente ao termo armazenado. Ambos são do tipo *String*, e seus dados são extraídos da tabela de termos matemáticos construído previamente. A outra classe, nomeada "Formula.java", possui os atributos "formula", que armazena uma fórmula escrita em TEX e é do tipo *String*, e "vetorMarcacao", um vetor do tipo inteiro que auxilia na troca de elementos das fórmulas. Com isso, foi criado também um vetor o qual possui o tipo "EstruturaTermo", e armazena os termos matemáticos catalogados para serem utilizados no programa.

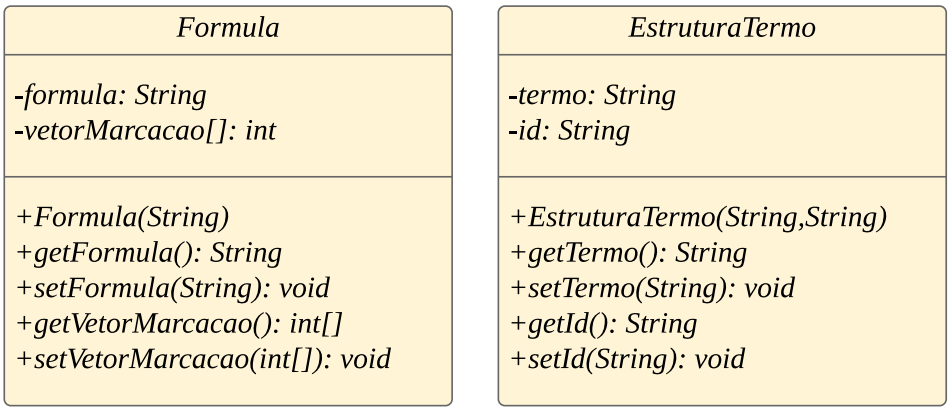

Figura 3. Diagramas UML das classes "Formula.java"e "EstruturaTermo.java".
Fonte: dos próprios autores.

3.1. Padronização das expressões matemáticas

Para facilitar a compreensão de como foi feita a padronização das fórmulas, podemos dividir o processo em três fases, onde a fase posterior é dependente da anterior.

3.1.1. Pré-tratamento da *string*

O primeiro passo para o tratamento da fórmula advinda do banco de dados foi a retirada dos elementos que não são importantes no contexto matemático e/ou que não agregariam informação na versão final da fórmula pré-processada. Uma das rotinas implementadas para este fim, realizou operações de tratamento através de expressões regulares, com relação aos espaços em branco. Desse modo, foram removidos os termos TEX que equivalem a espaço, tais como: \; e \quad, e em contrapartida, foi adicionado um espaço antes de cada barra invertida (\), para evitar problemas de junção de termos TEX consecutivos.

Em seguida, foram tratadas as fórmulas que possuem código para estrutura de matrizes e vetores, ou seja, as que continham termos como\begin{array}{}. Nestas, foi retirado o segundo argumento do termo \begin, o qual tem como função efetuar ajustes de alinhamento das colunas na matriz, característica não importante para o significado da fórmula em si. Em uma expressão iniciada com \begin{array}{lcr}, por exemplo, só restaria a *substring*: \begin{array}.

O próximo tratamento realizado diz respeito às palavras-chaves referentes à formatação e exibição de textos, que geralmente trazem conteúdo não relevante. Estas incluem \textstyle, \text, \textit, \color, \mbox, entre outras. Tais palavras foram removidas das expressões, assim como seu conteúdo. Um bom exemplo, é o da fórmula: \color{blue}S \color{black} + \color{red}S', que, após tal tratamento, foi reduzida para: S + S'. Contudo, nos casos em que havia conteúdo matemático incluído no argumento destes termos, indicado pela presença de pelo menos dois cifrões ($) não precedidos de barra invertida (\), tal conteúdo foi mantido, e apenas a palavra-chave e seus respectivos delimitadores (chaves) foram apagados.

De modo similar, também foram tratados termos como \mathbb, \mathop, \mathrm, que representam a presença de conteúdo matemático. Neste caso, porém, tudo o que estava entre os delimitadores dessas palavras-chave foi mantido, assim como na fórmula: x_0 \in \mathbb{C}^n, que foi alterada para: x_0 \in C^n.

3.1.2. Preenchimento da estrutura auxiliar de marcação

Para auxiliar no processo de tradução, foi introduzido no algoritmo um vetor de inteiros nomeado "vetorMarcacao", de tamanho n, em que n é o número de caracteres da fórmula vinda das etapas descritas anteriormente. Sua representação pode ser observada no exemplo da Figura 4, que mostra também a fórmula relacionada. Este vetor teve a finalidade de enumerar cada elemento, para posteriormente serem identificados como números, variáveis ou termos TEX.

Isto foi necessário para o algoritmo diferenciar as letras que eram identificadas como variáveis, das que faziam parte de uma palavra-chave TEX. Ao mesmo tempo, o

vetor fez com que a troca de elementos fosse mais organizada e explícita. Ele foi inicializado com zero em todas as posições, valor este que representa qualquer outro caractere presente na fórmula, exceto números, variáveis ou termos TEX.

Quando era encontrada uma barra invertida na análise da *string*, isto significava que havia um termo TEX presente na consulta, então o vetor de termos TEX era consultado. Caso o termo com início marcado pela barra invertida era encontrado no vetor de termos, cada posição do vetor correspondente à posição do caractere do termo na *string* foi marcada com o número 1, indicando a existência de uma variável ou algum termo TEX importante no contexto matemático, dependendo da categoria pertencente. Em seguida, os elementos do "vetorMarcacao" com a mesma posição de elementos numéricos da fórmula foram marcados com 2. Por último, as letras que não foram marcadas com 1, ou seja, não faziam parte de um termo TEX, foram todas marcadas com 3, representando variáveis. A identificação desse números e letras foi feita através de expressões regulares.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
F		\	l	i	m	_	{	s		\	t	o		1	}	(	s	-	1	)		\	z	e	t	a	(	s	)	=	1	.
V	0	1	1	1	1	0	0	3	0	1	1	1	0	2	0	0	3	0	2	0	0	1	1	1	1	1	0	3	0	0	2	0

Figura 4. Fórmula F: $\lim_{s\to 1}(s-1)\zeta(s) = 1.$ **mapeada para o vetor de marcação V.**
Fonte: dos próprios autores.

3.1.3. Substituição de elementos da fórmula matemática

Para realizar a troca de elementos matemáticos das fórmulas, foi implementada uma rotina na qual uma nova *string*, que conteria a saída do algoritmo, era formada a medida que os inteiros do "vetorMarcacao" de cada fórmula eram consultados. Essa rotina, descrita também no Algoritmo 1, o qual é um pseudocódigo da mesma, possuia quatro condições e suas respectivas instruções, e percorria o "vetorMarcacao"da posição $i = 0$ até $n - 1$. Tais condições eram:

- Se $vetorMarcacao[i] = 0$, a nova *string* recebia o caractere i da fórmula, sem alteração alguma;
- se $vetorMarcacao[i] = 1$, o "vetorMarcacao" era percorrido até a última ocorrência seguida do número 1, simultaneamente, cada caractere i da fórmula era concatenado em uma *string* auxiliar, que possuía o termo TEX presente, com isso, era adicionado na nova *string*, o código referente ao termo encontrado, presente na tabela;
- se $vetorMarcacao[i] = 2$, o contador i era incrementado até encontrar a última ocorrência do número 2, e apenas uma letra N era concatenada à nova *string*;
- por fim, se $vetorMarcacao[i] = 3$, a letra V era concatenada à nova *string*.

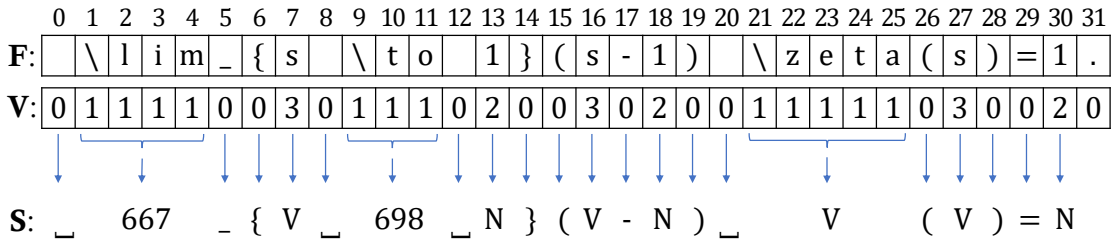

Figura 5. Saída padronizada (S) gerada a partir da fórmula $\lim_{s\to 1}(s-1)\zeta(s)=1$. (F) com o auxílio do vetor de marcação (V). Podemos notar que o ponto final (.) da fórmula original não foi incluído na versão final da saída padronizada.

Fonte: dos próprios autores.

Algoritmo 1: SUBSTITUIÇÃO DOS ELEMENTOS DA FÓRMULA

Entrada: $Formula\ \ f$
Saída: $String\ \ novaString$

```
1  início
2      String termo, novaSring ← ""
3      Inteiro i ← 0
4      enquanto i < tamanho da fórmula f faça
5          selecione f.vetorMarcacao[i] faça
6              caso 0 faça
7                  novaString ← novaString + f.formula[i]
8              fim
9              caso 1 faça
10                 termo ← ""
11                 enquanto f.vetorMarcacao[i] = 1 faça
12                     termo ← termo + f.formula[i]
13                     i ← i + 1
14                 fim
15                 novaString ← novaString + código do termo
16                 i ← i − 1
17             fim
18             caso 2 faça
19                 enquanto f.vetorMarcacao[i] = 2 faça
20                     i ← i + 1
21                 fim
22                 novaString ← novaString + N
23                 i ← i − 1
24             fim
25             caso 3 faça
26                 novaString ← novaString + V
27             fim
28         fim
29         i ← i + 1
30     fim
31 fim
32 retorna novaString
```

Após concluídas as etapas descritas, foram realizadas mais algumas substituições finais. Se a fórmula possuia um ponto (.) no seu final, este ponto foi retirado, com exceção das reticências (...). Toda *substring* "N.N", que indicava a presença de um número não inteiro (real), foi trocada por "R". O caractere "&" foi substituído por "C", por representar separação entre colunas de matrizes; e "\\" foi trocado por "L", representando nova linha. Também, foram retiradas as barras invertidas (\) restantes, as quais podiam representar espaço no formato TEX, e por último, dois ou mais espaços consecutivos foram convertidos para apenas um espaço.

Algumas dessas transformações podem ser notadas na Figura 5, a qual é um complemento para a Figura 4.

3.2. Resultados

O algoritmo implementado foi aplicado sobre sete bases de dados de fórmulas matemáticas no formato TEX : Mathematics[1], Socratic[2], MathOverflow[3], Wikipedia[4], PlanetMath[5], Wolfram MathWorld[6] e DLMF[7] (*Digital Library of Mathematical Functions*). Estas foram armazenadas em um banco de dados MariaDB. O sistema utilizado para executar o algoritmo segue as seguintes especificações: CPU Intel Core i7-6500U @ 2.50GHz, de quatro núcleos físicos; 8 GB de memória RAM; sistema operacional Ubuntu 18.04.1 LTS. O tem3.2. O tempo total de execução do algoritmo sobre cada base de dados, levando em conta as operações de consulta e atualização no banco de dados está mostrado na Tabela 1.

Base de dados	Número de fórmulas	Tempo de execução
Mathematics	8 488 806	22 minutos e 24 segundos
Socratic	1 693 153	3 minutos e 30 segundos
MathOverflow	929 909	2 minutos e 2 segundos
Wikipedia	398 889	1 minuto e 3 segundos
PlanetMath	67 418	13 segundos
Wolfram MathWorld	60 231	8 segundos
DLMF	25 827	6 segundos

Tabela 1. Tempo de execução das bases de dados pré processadas.

As bases de fórmulas pré-processadas estarão disponíveis para sua utilização no Laboratório de Redes e Sistemas Distribuídos (LaReS), no Campus Santa Clara, na UNIFAL-MG, tendo como principal objetivo, o seu uso como entradas para o algoritmo SLCS em sua versão paralela produzida por [Miya et al. 2018].

4. Conclusão e trabalho futuro

O trabalho descrito neste presente artigo, que padronizou as entradas para o algoritmo SLCS Paralelo construído em CUDA por [Miya et al. 2018], utilizou-se da linguagem

[1] https://math.stackexchange.com/
[2] https://socratic.org/
[3] https://mathoverflow.net/
[4] https://dumps.wikimedia.org/enwiki/
[5] https://planetmath.org/
[6] http://mathworld.wolfram.com/
[7] https://dlmf.nist.gov/

JAVA para melhor tratar *strings* que representam expressões matemáticas escritas em TEX. Para isto, foi criada uma tabela contendo palavras-chave TEX que abrangem áreas da matemática como Cálculo, Álgebra, Trigonometria, etc, encontradas em sites, e posteriormente foi definida uma estrutura de informação padrão, baseada na estrutura de [Kumar et al. 2012]. Após estes passos, utilizou-se de sete bases da dados, e nestas foi realizado um pré-tratamento que retira elementos irrelevantes no contexto matemático. Posteriormente, foi construída uma estrutura de marcação que auxiliou na substituição de elementos matemáticos das fórmulas, para assim, produzir as entradas para o algoritmo que é complementado por este trabalho.

Como trabalho futuro, será realizada a união destes dois trabalhos para avaliar a viabilidade da relação custo-desempenho de tal modo de recuperação desenvolvido para expressões matemáticas. Por conseguinte, esperam-se resultados que possam indicar se esta forma de recuperar conteúdo matemático possui boa performance em relação aos modelos de recuperação por fórmulas já existentes.

Referências

Balhaf, K., Shehab, M. A., Wala'a, T., Al-Ayyoub, M., Al-Saleh, M., and Jararweh, Y. (2016). Using gpus to speed-up levenshtein edit distance computation. In *2016 7th International Conference on Information and Communication Systems (ICICS)*, pages 80–84. IEEE.

Bednárek, D., Brabec, M., and Kruliš, M. (2017). Improving matrix-based dynamic programming on massively parallel accelerators. *Information Systems*, 64:175–193.

Carlisle, D. (1992). Latex math symbols. `http://web.ift.uib.no/Teori/KURS/WRK/TeX/symALL.html`. [Online, accessed 19-July-2019].

Carlisle, D., Pakin, S., and Holt, A. (2001). The great, big list of latex symbols.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to algorithms*. MIT press.

Gratzer, G. (1995). *Math into LATEX, An Introduction to LATEX and AMS-LATEX*. Birkhauser Boston, Inc.

Kumar, P. P., Agarwal, A., and Bhagvati, C. (2012). A structure based approach for mathematical expression retrieval. In *International Workshop on Multi-disciplinary Trends in Artificial Intelligence*, pages 23–34. Springer.

Miya, A. W., Gomes, G. C., and Gonzaga, F. B. (2018). Implementações do algoritmo SLCS em CUDA e aplicações em recuperação de informações matemáticas. 16f. Trabalho de Conclusão de Curso (Graduação)-Universidade Federal de Alfenas, Alfenas, 2018.

Sermutlu, E. (2006). Latex and ams-latex symbols.

Wikipedia contributors (2019). Levenshtein distance. `https://en.wikipedia.org/wiki/Levenshtein_distance`. [Online, accessed 29-July-2019].