GraphBLAS para Rust

Robert Morais Santos Broketa[1], **Hamilton José Brumatto**[1], **Vânia Cordeiro da Silva**[1]

[1]Departamento de Ciência da Computação
Universidade Estadual de Santa Cruz (UESC) – Ilhéus, BA – Brasil

rmsbroketa.cic@uesc.br, hjbrumatto@uesc.br, vania@uesc.br

Abstract. *This work explores the implementation of GraphBLAS, a linear algebra-based library for graph algorithms, using the Rust programming language. GraphBLAS is a symbolic language that uses sparse matrix and vector operations to efficiently represent and manipulate graphs, allowing the development of algorithms related to them. Rust's strong guarantees for memory safety, concurrency, and performance make it a candidate for building a library like GraphBLAS. This study applies the theoretical foundations of GraphBLAS, including its algebraic structures such as monoids and semirings, using Rust's typing system and programming paradigms.*

Resumo. *Este trabalho explora a implementação do GraphBLAS, uma biblioteca baseada em álgebra linear para algoritmos de grafos, na linguagem de programação Rust. O GraphBLAS é uma linguagem simbólica que utiliza operações com matrizes esparsas e vetores para representar e manipular grafos de forma eficiente, permitindo o desenvolvimento de algoritmos relacionados aos mesmos. As fortes garantias do Rust para segurança de memória, concorrência e desempenho o torna um candidato para a construção de uma biblioteca como o GraphBLAS. Este estudo aplica os fundamentos teóricos do GraphBLAS, incluindo suas estruturas algébricas, como monoides e semianéis, utilizando o sistema de tipagem e paradigmas de programação do Rust.*

1. Introdução

O uso de grafos na modelagem de dados tem sido de extrema importância pois grafos conseguem relacionar instâncias, guardando informações sobre as instâncias bem como sobre as relações. Vários estudos em teoria de grafos foram realizados e destes decorrem importantes algoritmos [Bondy and Murty 2008]. Porém diferente de estruturas de dados lineares, como *Pilhas*, *Filas*, *Mapas* e outras que possuem um grande conjunto de bibliotecas prontas em várias linguagens, as linguagens de programação não definem um conjunto de bibliotecas para grafos e seus algoritmos.

Isto decorre do fato de que grafos possuem uma grande flexibilidade no tratamento de suas informações, pois não há um padrão claro de todas as informações que precisam ser armazenadas e os grafos podem ser representados em diversas formas, entre elas a matriz de adjacência, matriz de incidência e lista de adjacência [Bondy and Murty 2008].

Os grafos quando armazenados na forma de matriz, seja de adjacência ou seja de incidência, podem fazer o uso de operações da álgebra linear para implementação de seus algoritmos. Desta forma, já podemos contar com um conjunto de bibliotecas que são preparadas para tratar operações de matrizes e vetores em álgebra linear. Normalmente

esta classe de bibliotecas é denominada de BLAS (*Basic Linear Algebra Subprograms*). Nela encontramos operações como soma de matrizes produto entre matrizes e vetores, inversão de matrizes, decomposição de matrizes entre outras [Blackford et al. 2002].

Com isto surgiu a proposta de uma linguagem de implementação de alto nível baseada em operações da álgebra linear para implementar os algoritmos em grafos. Nasce então o GraphBLAS [Community]. O GraphBLAS representa um conjunto de operações da álgebra linear que podem ser encadeadas formando um algoritmo de grafo [Davis 2019]. Com isto começaram a ser construídas APIs (*Application Program Interface*) para traduzir ou implementar GraphBLAS em várias linguagens, como C/C++, MPI/C++, Java, entre outras.

Neste trabalho a proposta é dar um início a um projeto que venha a criar API do GraphBLAS para a linguagem Rust. Rust é uma linguagem de programação relativamente nova que permite desenvolver sistemas seguros e eficientes, sem recorrer a práticas como Coletores de Lixo (*Garbage Collectors*) [Klabnik and Nichols 2023].

Este trabalho está dividido da seguinte forma, na seção 2 são apresentados os conceitos relacionados ao GraphBLAS e a representação dos grafos com base na álgebra linear. Na seção 3 estão apresentadas as operações da álgebra linear com vistas às operações do GraphBLAS. Na seção 4 faz-se uma apresentação da linguagem Rust e na seção 5 propõe-se a implementação de elementos básicos do GraphBLAS e Álgebra Linear na linguagem Rust e na seção 6 uma demonstração de construção de algoritmo com esta implementação. Por fim, na seção 7 ponderamos as conclusões finais a respeito do trabalho.

2. GraphBLAS

GraphBLAS é um conjunto padronizado de APIs para executar análises e operações em grafos utilizando métodos baseados em matrizes. A biblioteca parte da ideia de que matrizes esparsas podem ser usadas para representar grafos como matrizes de adjacência ou de incidência. Dessa forma operações em grafos podem ser executadas como transformações lineares e outras operações da álgebra linear. Conceitualmente, vértices de um grafo são representados por linhas e colunas de uma matriz de adjacência, e arestas correspondem a valores não zero da mesma.

GraphBLAS se baseia no BLAS, ambos representam um conjunto de subrotinas ou funções e descrições de estruturas ou classes que oferecem um conjunto básico de operações da Álgebra Linear, como produto entre vetores, produto matriz-vetor, inversão de matrizes, e outras. O objetivo do BLAS é fornecer apoio às várias operações da álgebra linear, já no GraphBLAS é fornecer apoio às várias operações baseadas na álgebra linear que permitam construir algoritmos em grafos, como exemplificado na Figura (1). As principais aplicações de algoritmos com GraphBLAS se concentram em *datamining* e base de dados volumosas. Desta forma, soluções robustas, paralelas e eficientes são necessárias nas implementações desta biblioteca.

2.1. Matrizes de Adjacência

Um grafo pode ser modelados na forma de Matriz de Adjacência[Kepner et al. 2016]. Tal matriz geralmente quadrada, pois as linhas e

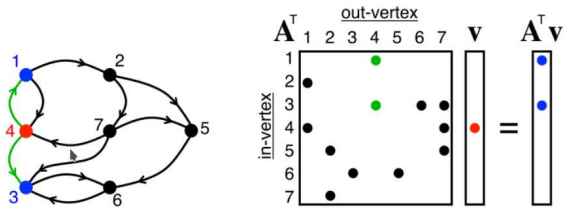


Figura 1. Representação de uma etapa de uma busca em largura
Fonte: The GraphBLAS community [Community]

colunas são o mesmo conjunto de vértices representam ligações entre vértices adjacentes. Mas podem ser conjuntos disjuntos de vértices, a exemplo de grafos bipartidos.

Dada uma matriz de adjacência $\mathbf{A}$ para um grafo orientado, se $\mathbf{A}(v_1, v_2) = 1$, então existe um arco do vértice v_1 para o vértice v_2, vide Figura (2). Observe que $\mathbf{A}(v_1, v_2)$ não é equivalente a $\mathbf{A}(v_2, v_1)$, pois seriam arcos distintos. Matrizes de adjacência também podem conter pesos associados aos seus arcos. Se $\mathbf{A}(v_1, v_2) = w_{12} \neq 0$, então o arco de v_1 para v_2 contém o peso w_{12} como podemos ver representado na Figura (2).

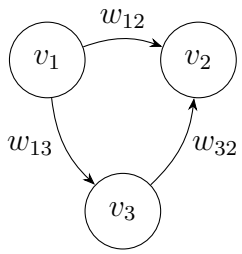


Figura 2. Exemplo de um grafo orientado com peso nos arcos

2.2. Matriz de Incidência

Matrizes de incidência podem ser usadas para representar grafos que possuam quaisquer das propriedades: orientados, k-partidos, multigrafos e/ou hipergrafos[Kepner et al. 2016].

Uma matriz de incidência $\mathbf{E}$ utiliza as linhas para representar cada aresta ou arco do grafo e as colunas representam cada vértice, o valor armazenado na matriz é o grau de incidência da aresta sobre o vértice.

Em um grafo não orientado, podemos considerar a relação de incidência $\Psi(a) = \{v_1, v_2\}$ representada na matriz de incidência com $\mathbf{E}(a, 1) = 1$ e $\mathbf{E}(a, 2) = 1$ como indicação de que aresta a é uma ligação entre v_1 e v_2, no caso de e, que é um laço em v_3, temos que $\mathbf{E}(e, 3) = 2$, como podemos ver na Figura (3).

Em um hipergrafo uma aresta pode representar a ligação entre mais de dois vértices. Da mesma forma que em um grafo comum, no hipergrafo cada vértice recebe o grau da incidência da aresta ou arco sobre ele. Exemplificando, para denotar um hiperarco a que possui uma ligação com caudas em v_1 e v_2 e cabeça em v_3 basta indicar que $\mathbf{E}(a, 1) = -1$, $\mathbf{E}(a, 2) = -1$ e $\mathbf{E}(a, 3) = 1$, como podemos ver na Figura (4). Por outro lado se o hipergrafo não é orientado, e a é uma hiperaresta entre v_1, v_2 e v_3, cada vértice possui o grau 1 de incidência a partir desta hiperaresta.

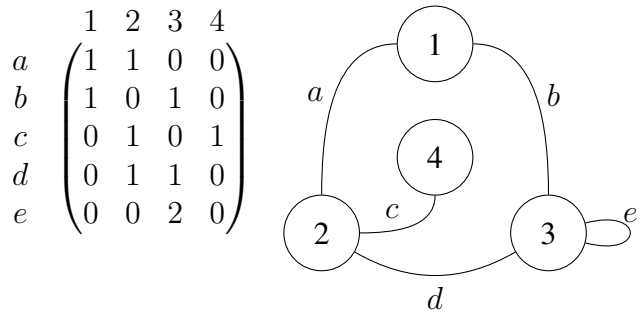


Figura 3. Matriz de incidência representando um grafo não orientado

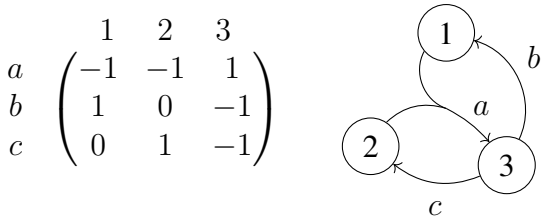


Figura 4. Matriz de incidência representando um hipergrafo orientado

3. Matrizes, Álgebra Linear e GraphBLAS [Kepner et al. 2016]

Apesar da notação formal de Matrizes de transformação em espaços vetoriais na Álgebra Linear, no uso destas no contexto da GraphBLAS seremos menos formais.

A definição canônica de matriz no contexto do GraphBLAS que possui m linhas e n colunas de números reais está representada na Eq. (1).

$$\mathbf{A} : \mathbb{R}^{m \times n} \tag{1}$$

Os índices das linhas e colunas de uma matriz $\mathbf{A}$ são, respectivamente, $i \in I = \{1, ..., m\}$ e $j \in J = \{1, ..., n\}$, de forma que qualquer valor de $\mathbf{A}$ pode ser denotado por $\mathbf{A}(i, j)$. Isto não está restrito ao conjunto dos reais, de forma semelhante a matriz $\mathbf{A}$ pode armazenar valores do conjunto dos complexos, números inteiros e números naturais.

Canonicamente índices de linhas e colunas são números naturais $I, J : \mathbb{N}$. Porém algumas implementações do GraphBLAS podem apresentar índices não negativos $I = \{0, ..., m-1\}$ e $J = \{0, ..., n-1\}$.

Para GraphBLAS uma matriz é definida pela seguinte notação em 2D:

$$\mathbf{A} : I \times J \to \mathbb{S} \tag{2}$$

onde índices $I, J : \mathbb{Z}$ são inteiros finitos não negativos com m e n elementos, respectivamente, e $\mathbb{S} \in \{\mathbb{R}, \mathbb{Z}, \mathbb{N}, ...\}$ é um conjunto de escalares. Sem perda de generalidade matrizes podem ser denotadas por:

$$\mathbf{A} : \mathbb{S}^{m \times n} \tag{3}$$

Um vetor puro é denotado por:

$$\mathbf{v} = \mathbb{S}^{m} \tag{4}$$

Tabela 1. Grupos e Operações

GRUPO	DOMÍNIO	$\oplus$	$\otimes$	ZERO	IDENTIDADE	ANIQUILADOR
Aritmética dos Reais	$a \in \mathbb{R}$	$+$	$\times$	0	$a + 0 = a$	$a \times 0 = 0$
Álgebra Min-Plus	$a \in \{\mathbb{R} \cup -\infty\}$	$\min$	$+$	∞	$\min(a, \infty) = a$	$a + \infty = \infty$
Álgebra Max-Min	$\mathbb{R}^+$	$\max$	$\min$	0	$\max(a, 0) = a$	$\min(a, 0) = 0$
Conjunto de Partes	$\mathcal{P}(\mathbb{Z})$	$\cup$	$\cap$	$\emptyset$	$a \cup \emptyset = a$	$a \cap \emptyset = \emptyset$

Um escalar é um elemento singular de um conjunto $s \in \mathbb{S}$ e não possui dimensões.

3.1. Operações e Propriedades Escalares

Operações de matrizes existentes no GraphBLAS são construídas a partir de operações que chamamos de escalares. As operações escalares primárias são adição aritmética ($1 + 1 = 2$) e multiplicação ($2 \times 2 = 4$). GraphBLAS também permite que essas operações de adição e multiplicação sejam definidas pelo implementador ou usuário. Para evitar confusão com as operações primárias, $\oplus$ será utilizado para denotar adição e $\otimes$ multiplicação.

Certas combinações de $\oplus$ e $\otimes$ sobre certos tipos de valores escalares são úteis pois eles preservam propriedades matemáticas desejadas, como associatividade e distributividade.

Tais propriedades são extremamente úteis para construir aplicações com grafos pois elas permitem a troca de operações sem alterações no resultado.

Algumas operações também preservam a comutatividade, porém podem existir definições para $\oplus$ e $\otimes$ que não as preservem. Dessa forma, tais operações acarretam em propriedades similares quando efetuadas sob matrizes e vetores.

3.2. Monoides e Semianéis

Em álgebra abstrata, monoides são conjuntos compostos por uma operação binária associativa e um elemento neutro, chamado de identidade[Chevalley 1956]. Anéis caracterizam-se por grupos compostos por duas operações binarias $\oplus$ e $\otimes$. Considerando a partir do conjunto R, o grupo (R, $\oplus$) possui uma operação associativa, comutativa, um elemento identidade, e um aditivo inverso. O grupo (R, $\otimes$) é um monoide, possui a distributividade da multiplicação em respeito a adição. A tabela 1 apresenta um conjunto de grupos e operações associadas, indicando a identidade associativa e o aniquilador multiplicativo.

Semianéis são generalizações de anéis que descartam o requerimento do aditivo inverso sobre $\oplus$.

3.3. Elemento 0: ausência de arestas

Matrizes esparsas têm uma grande importância em GraphBLAS. Muitas implementações de matrizes esparsas reduzem o armazenamento ao não guardar os elementos

0 de uma matriz.

GraphBLAS permite o elemento 0 a ser definido pela implementação ou usuário. Quando o elemento 0 possui certas propriedades com respeito com as operações escalares $\oplus$ e $\otimes$, então a esparsidade das operações das matrizes podem ser feitas eficientemente. Tais propriedades são a identidade aditiva e o aniquilador multiplicativo.

3.4. Máscaras

Máscaras ("masks") permitem a aplicação seletiva de operações a elementos específicos de uma matriz, com base em uma condição binária. Tal seletividade se prova particularmente útil em algoritmos de grafo e operações matriciais onde elementos de uma matriz são condicionalmente dependentes de valores da mesma ou de critérios externos.

Elas são matrizes ou vetores binários, tipicamente das mesmas dimensões dos elementos sendo operados. São utilizadas para controlar quais elementos da matriz devem ser atualizados durante uma operação, como exemplo típico, serve para não atualizar vértices "já visitados pelo algoritmo". Os elementos da máscara são **true** ou **false**. Ao efetuar uma operação, apenas elementos que correspondem a **true** da máscara são afetados. Essa aplicação seletiva de operações trazem ao GraphBLAS uma grande flexibilidade e eficiência na implementação de algoritmos.

Existem máscaras estruturais, que representam a existência de elementos não vazios e máscaras complementares, que representam a existência de elementos vazios.

4. Rust

Esta seção tem como base o livro *The Rust Programming Language* [Klabnik and Nichols 2023], este pode ser usado como uma referência mais completa sobre a linguagem.

Rust é uma linguagem de programação voltada a sistemas, fortemente tipada, desenvolvida com um foco em segurança, performance e concorrência. Possui muitos paradigmas, combinando características imperativas, funcionais, orientadas a objetos e concorrentes. Ela permite gerenciamento de memória de baixo nível, porém equipado com garantias de segurança que previnem bugs comuns em outras linguagens.

Tais seguranças são possíveis graças a um modelo novo de gerenciamento de memória, baseado em um sistema de *ownership* e *borrowing* (propriedade e empréstimo). Esse modelo elimina a necessidade de um coletor de memória, comum em linguagens como Java, garantindo que memórias sejam automaticamente liberadas quando não estão mais sendo usadas. Esse modelo é reforçado através da tipagem da linguagem, juntamente com *borrowing*, onde cada valor tem apenas um "dono", e referências a valores são mutáveis ou imutáveis, mas não os dois simultaneamente. Essas abstrações não incorrem nenhum custo extra em tempo de execução pois são checadas em tempo de compilação. Tais garantias fazem Rust ser comparado com linguagens como C e C++ em termos de performance.

O sistema de Traits permite ao Rust incorporar características como abstração e polimorfismo comuns em códigos reutilizáveis e permite também outro tipo de polimorfismo: a *composição*, que expande o significado de polimorfismo. Um Trait especifica

um contrato que um tipo deve implementar, semelhante a interfaces em Java. Este sistema tem um papel crucial na programação genérica. Rust também explora a construção de macros na construção de código, chamado de *metaprogramming* que são interpretados em tempo de compilação.

As bibliotecas associadas ao Rust, como Tokyo [Flitton and Morton 2025] e Rayon [Hohenheim and Durante 2018] permitem um paralelismo automático no uso de *iterators*, *locks* e outros mecanismos que simplificam as tarefas do programador, já que este não precisa criar e manipular os threads.

5. Implementação

Este projeto realiza a implementação de uma grande porção das definições impostas pela especificação 2.2.0 [Brock et al. 2021]. Não pretende-se aqui criar algoritmos em GraphBLAS, mas sim implementar as operações do GraphBLAS em Rust.

Nesta versão tem-se apenas a implementação serial dos códigos, apesar do Rust permitir uma paralelização automática no uso de *iterators* ao navegar ao longo de vetores, muitas paralelizações em operações de matrizes requerem algoritmos próprios que permitam *pipelines* [Grama et al. 2003]. Uma vez a versão serial correta e completa será explorada a paralelização.

As definições aqui mostradas serão apenas descrições das mesmas, e limitadas àquelas relevantes à implementação do algoritmo de caminhos mínimos (SSSP - *Single Source Shortest Paths*), afim de demonstrar a utilização da biblioteca para solucionar problemas relacionados a grafos. Para mais detalhes, a implementação inicial da biblioteca está disponível em [Morais 2025].

A biblioteca define e implementa uma base de operadores unários, binários, monoides, e semianeis. Ela incorpora também formas que permitem usuários criarem seus próprios operadores e operações, respeitando as definições impostas pela especificação. Todas essas definições são construídas em tempo de compilação.

É implementado uma definição de vetor, matriz e operações básicas que permitem o usuário implementar suas próprias estruturas de vetores, que podem ser usadas para efetuar as operações algébricas sem se importar com o tipo ou implementação do vetor utilizada. De forma semelhante é proposta uma interface que define máscaras, de forma que usuários são livres para trazer suas implementações de máscaras.

Buscando a facilidade de uso, a biblioteca provê também uma implementação simples das estruturas citadas (vetor e matriz esparsa, além de máscaras), permitindo o uso da mesma sem necessitar inicialmente de suas próprias versões.

5.1. Operações

São implementadas diversas operações algébricas que dizem respeito a vetores e matrizes. Exemplos, para A, B e C matrizes, u, v e w vetores e $\langle M \rangle$, $\langle m \rangle$ máscaras, a tabela 2 apresenta algumas operações implementadas na forma de métodos, entre vetores, matrizes e combinadas.

5.2. Descritores

Descritores especificam como os outros argumentos de entrada que correspondem a coleções (vetores, matrizes, máscaras) devem ser processados antes da operação princi-

Tabela 2. Operações do GraphBLAS

MÉTODO	NOME	NOTAÇÃO
`mxm`	matriz × matriz	$C\langle M\rangle = A \oplus . \otimes B$
`vxm`	vetor × matriz	$w\langle m\rangle = u \oplus . \otimes A$
`mxv`	matriz × vetor	$w\langle m\rangle = A \oplus . \otimes u$
`eWiseAdd`	adição elemento a elemento	$C\langle M\rangle = A \oplus B$
	conjunto de união de índices	$w\langle m\rangle = u \oplus v$
`eWiseMult`	multiplicação elemento a elemento	$C\langle M\rangle = A \otimes B$
	conjunto de interseção de índices	$w\langle m\rangle = u \otimes v$

pal do método ser executada.

6. Algoritmos

Como aplicação será apresentado a implementação do algoritmo do caminho mínimo de Bellman-Ford. No algoritmo usa-se o semi-anel da álgebra Min-Plus, o Algoritmo (1) mostra o resultado.

Algoritmo 1 SSSP: Caminho Mínimo - Algébrico Bellman-Ford

Entrada: Matriz de adjacência **A**, vértice de origem s, #vértices n
Saída: Vetor de distância d (real)
$\mathbf{d} = [\infty, \infty, ..., \infty]$
$\mathbf{d}(s) = 0$ ▷ Distância da origem para origem
for $k = 1$ para $n - 1$ **do**
  $\mathbf{d} = \mathbf{d} \oplus . \otimes \mathbf{A}$ ▷ Atualiza a distância para os vértices
end for

As Figuras (5) a (8) simulam a aplicação do algoritmo passo a passo no ciclo **for**, considerando o grafo orientado de 5 vértices da figura representado na forma da matriz A

Figura 5. Algoritmo SSSP Bellman-Ford, k = 1

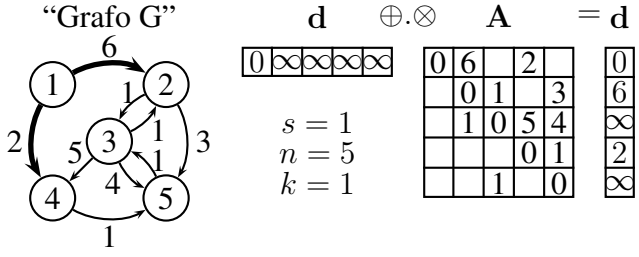


Figura 6. Algoritmo SSSP Bellman-Ford, k = 2

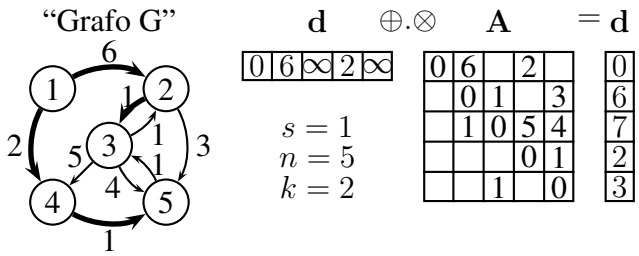


Figura 7. Algoritmo SSSP Bellman-Ford, k = 3

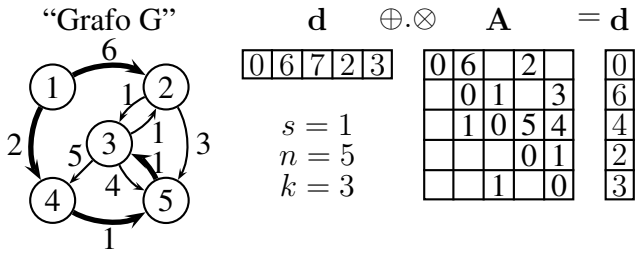
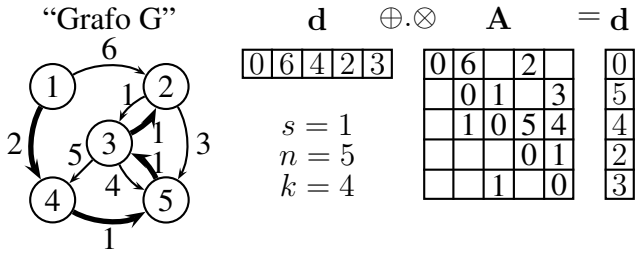


Figura 8. Algoritmo SSSP Bellman-Ford, k = 4



Esse algoritmo pode ser construído utilizando construções da biblioteca em Rust, como demonstrado na listagem 2.

Algoritmo 2 Algoritmo de SSSP utilizando construções do GraphBLAS.

```
fn sssp<Mt, Vc>(a: &Mt, s: IndexType, n: IndexType) -> GblasResult<Vc>
where Mt: Matrix<Scalar = f32>, Vc: VecOps<Scalar = f32, Mat = Mt> {
    if a.nrows() != a.ncols() {
        return Err(ApiError::DimensionMismatch.into());
    }
    // d = [inf, inf, ..., inf]
    let idxs = (0..a.nrows()).collect::<Vec<_>>();
    let vals = idxs.iter().map(|_| f32::INFINITY).collect::<Vec<_>>();
    let mut d = Vc::new(a.nrows())?.build(idxs, vals, n, First::new())?;
    // d(s) = 0
    d.set_element(s, 0.)?;
    // for k = 1 para n - 1 do
    for _ in 1..n - 1 {
        // d = d min.plus A
        let d_dup = d.clone();
        d.vxm(
            Option::<Vc>::None,
            Some(Minimum::new()),
            MinPlusSemiring::new(),
            &d_dup,
            a,
            None,
        )?;
    }
    Ok(d)
}
```

7. Considerações Finais

Esse estudo demonstra uma inicial implementação do GraphBLAS na linguagem de programação Rust, uma vez que a API do GraphBLAS é extensa. As operações implementadas oferecem uma base para futuras expansões de bibliotecas para processamento

de grafos. A integração com as operações algébricas oferecidas por GraphBLAS juntamente com o sistema de tipagem robusto de Rust e garantias de segurança fornecem um caminho promissor para o desenvolvimento de algoritmos de processamento de grafos escaláveis e de alta performance. A exploração dos recursos providos por Rust, como seu modelo de *ownership* e genéricos, destaca seu potencial para lidar com tarefas computacionais complexas concorrentes enquanto garantem segurança e eficiência.

Trabalhos futuros envolve a extensão da implementação atual com operações adicionais, otimização dos cálculos da matriz esparsa de forma a permitir implantar os vários algoritmos do GraphBLAS. Outra extensão é a exploração de implementações que façam uso extensivo das capacidades de concorrência da linguagem. Tal feito representaria uma grande evolução na viabilidade do uso dessa versão do GraphBLAS uma vez que paralelismo e concorrência são importantes nas principais aplicações do GraphBLAS e podem ser bem aproveitados no contexto de operações da álgebra linear.

Referências

Blackford, L. S., Petitet, A., Pozo, R., Remington, K., Whaley, R. C., Demmel, J., Dongarra, J., Duff, I., Hammarling, S., Henry, G., and others (2002). An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151. Publisher: Citeseer.

Bondy, J. A. and Murty, U. S. R. (2008). *Graph Teory*. Springer.

Brock, B., Bulu, A., Mattson, T., Mcmillan, S., and Moreira, J. (2021). The GraphBLAS C API Specification: Version 2.0.0. *The Carnegie Mellon University*.

Chevalley, C. (1956). *Fundamental concepts of algebra*. New York : Academic Press.

Community, G. The GraphBLAS. https://graphblas.org/. Publication Title: Welcome to the GraphBLAS Forum. Accessed: 2024-08-23.

Davis, T. A. (2019). Algorithm 1000: SuiteSparse: GraphBLAS: Graph algorithms in the language of sparse linear algebra. *ACM Transactions on Mathematical Software (TOMS)*, 45(4):1–25. Publisher: ACM New York, NY, USA.

Flitton, M. and Morton, C. (2025). *Async Rust: Unleashing the Power of Fearless Concurrency*. O'Reilly Media, Sebastopol.

Grama, A., Gupta, A., Karypis, G., and Kumar, V. (2003). *Introduction to Parallel Computing*. Addison-Wesley, Harlow, England ; New York.

Hohenheim, J. and Durante, D. (2018). *Rust Standard Library Cookbook: Over 75 recipes to leverage the power of Rust*. Packt Publishing, Birmingham, UK.

Kepner, J., Aaltonen, P., Bader, D., Buluç, A., Franchetti, F., Gilbert, J., Hutchison, D., Kumar, M., Lumsdaine, A., Meyerhenke, H., McMillan, S., Yang, C., Owens, J. D., Zalewski, M., Mattson, T., and Moreira, J. (2016). Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9.

Klabnik, S. and Nichols, C. (2023). *The Rust programming language*. No Starch Press.

Morais, R. (2025). GraphBLAS para Rust. https://github.com/rybertm/gblas.