UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Mateus Ferreira Silva

Desenvolvimento e Avaliação do TGCSA para Redução de Consumo de Memória em Grafos Temporais

Uberlândia, Brasil

2023

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Mateus Ferreira Silva

Desenvolvimento e Avaliação do TGCSA para Redução de Consumo de Memória em Grafos Temporais

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Sistemas de Informação.

Orientador: Prof. Dr. Marcelo Keese Albertini

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Sistemas de Informação

Uberlândia, Brasil

2023

Resumo

A cada dia, a quantidade de dados gerados aumenta, exigindo ferramentas mais eficientes para o uso de memória RAM. Para isso, estruturas de dados compactas são uma alternativa, pois, possuem um consumo menor de memória RAM em detrimento da eficiência no tempo de execução de consultas. Assim, é possível realizar processamentos em conjuntos de dados que antes eram inviáveis devido ao seu grande volume. Neste trabalho, temos como objetivo implementar a estrutura de dados compacta TGCSA para realizar a operação de vizinhos diretos e estudar o seu desempenho no consumo de memória RAM, por meio de uma avaliação prática. Os resultados provenientes dos experimentos realizados indicam que o TGCSA apresentou um menor consumo de memória RAM em comparação com as outras estruturas de dados para grafos temporais testadas. Entretanto, ainda é necessário incorporar novas técnicas ao TGCSA implementado neste trabalho, visando otimizar o tempo de execução das consultas e aprimorar ainda mais a compactação das estruturas internas.

Palavras-chave: Estrutura de Dados Compacta, Grafo Temporal, Suffix Array, TGCSA, Memória RAM

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

TCC	Trabalho de Conclusão de Curso
CSA	Compressed Suffix Array
iCSA	integer-based CSA
TGCSA	Temporal Graph Compressed Suffix Array

Lista de símbolos

$[a, b]$	Intervalo fechado de números inteiros $[a, b]$.
$S[i]$	Elemento de índice i na sequência S, onde $1 \leq i \leq n$, tal que n é igual a quantidade de elementos de S.
$S[a, b]$	Sub-intervalo da sequência S, composto por elementos $S[i]$, $\forall i \in [a, b]$.
O	Notação O de análise assintótica.
Θ	Notação Θ de análise assintótica.
$\in$	Pertence.
$\Leftrightarrow$	Se e somente se.
$\subset$	Está contido; é subconjunto próprio de.
$\subseteq$	É subconjunto de.
$\prec$	Precede na ordem lexicográfica.
Σ	Alfabeto.
Ψ	Sequência Ψ.

Sumário

1 Introdução

Um grafo é um tipo de dado abstrato que pode ser utilizado para representar inúmeras situações do mundo real. Por exemplo, um grafo pode ser aplicado para modelar os contatos entre indivíduos em um conjunto de pessoas, onde as pessoas são representadas como um conjunto de vértices V e os contatos entre elas são representados como um conjunto de arestas E. Nesse contexto, considera-se que um contato ocorreu sempre que duas pessoas estiveram a uma distância mínima d uma da outra. Uma das variações do grafo é o grafo temporal, onde a dimensão do tempo também é levada em consideração. Entre as operações possíveis em um grafo temporal, destaca-se a identificação dos vizinhos diretos. Essa operação possibilita verificar se um vértice u está conectado diretamente a outro vértice v, levando em consideração o intervalo de tempo de duração de cada conexão. Seguindo o exemplo anterior, ao adicionarmos a dimensão do tempo, agora não só saberíamos quais indivíduos entraram em contato entre si, mas também quando estes contatos ocorreram.

Para exemplificar a aplicação de um grafo temporal, considere o seguinte cenário. Imagine uma situação em que um vírus altamente contagioso se espalha entre pessoas, onde qualquer pessoa que se aproxime a uma distância d de um indivíduo infectado corre o risco de ser contaminada. Esse tipo de cenário pode ser representado eficazmente por um grafo temporal, no qual a operação de vizinhos diretos pode ser empregada para identificar todas as pessoas que tiveram contato com um indivíduo infectado. Em outras palavras, essa operação nos permite encontrar novos possíveis casos de infecção com base no indivíduo infectado e no período de tempo em que ele estava transmitindo o vírus. Além desse cenário específico, inúmeras outras situações podem ser adequadamente modeladas como grafos temporais, permitindo um armazenamento e consulta eficientes de seus dados.

Grafos podem ser representados de várias formas e cada representação pode ter uma complexidade de espaço distinta. As três formas mais comuns de armazenar grafos são: listas de adjacências, matrizes de adjacência e lista de arestas. Em listas de adjacência o custo de espaço é $\Theta(|V|+|E|)$. Matrizes de adjacência custam $\Theta(|V|^2)$ de espaço. Listas de arestas custam $\Theta(|E|)$ de espaço. Para estimar o custo de um grafo temporal, podemos multiplicar o uso de espaço de cada estrutura de dados pela quantidade de elementos no conjunto de instantes de tempo T.

Considerando o exemplo que foi citado anteriormente, supondo que há 100.000 pessoas e que por dia cada pessoa entra em contato com outras 20 pessoas durante um intervalo de tempo de 365 dias, temos $\nu = |V| = 100.000$, $\epsilon = |E| = 2.000.000$ e $\tau = |T| = 365$. Ao utilizar números inteiros de 32 *bits* (4 *bytes*) para representar V e E, obteríamos

as seguintes estimativas de uso de memória por estrutura de dados:

- Listas de adjacência $(\nu + \epsilon) \times \tau \times 4$ *bytes* = 3,066 *Gigabytes*.
- Matrizes de adjacência $\nu^2 \times \tau \times 4$ *bytes* = 14.600 *Gigabytes*.
- Listas de arestas $\epsilon \times \tau \times 4$ *bytes* = 2,92 *Gigabytes*.

Uma das formas de se implementar grafo temporal é o *Temporal Graph Compressed Suffix Array* (TGCSA), conforme descrito por (BRISABOA et al., 2014). O TGCSA é uma estrutura de dados compacta, isto é, uma estrutura de dados que busca reduzir o consumo de memória sem que haja grande perda no tempo de execução original. Estruturas de dados compactas nos ajudam no armazenamento e computação de conjuntos de dados que antes eram impraticáveis devido ao seu volume, como explicado por (NAVARRO, 2016).

1.1 Objetivos

O objetivo geral desse trabalho de conclusão de curso é realizar um estudo e implementação do TGCSA, e analisar o seu desempenho na solução de problemas de conectividade em grafos temporais. Sendo assim, os objetivos específicos desse trabalho são: implementar diversas versões de grafos temporais não compactos, implementar o TGCSA e comparar os resultados obtidos.

2 Conceitos básicos

2.1 Grafo

O grafo é uma estrutura de dados vastamente utilizada devido a sua capacidade de modelar diversos problemas do mundo real. Segundo Navarro (2016), um grafo é um conjunto de nós, ou vértices, conectados em pares na forma de arestas. Em um grafo direcionado, as arestas possuem direção, ou seja, as arestas vão do vértice de origem u para o vértice de destino v, portanto, é dito que v é adjacente a u. Grafos não-direcionados são grafos onde as arestas são simétricas entre u e v, sendo assim, u e v são adjacentes entre si. Um grafo direcionado pode ser denotado por $G = (V, E)$, onde V é o conjunto dos vértices e $E \subseteq V \times V$ é o conjunto das arestas. Neste trabalho serão utilizados grafos direcionados.

2.2 Grafo temporal

De acordo com Brisaboa et al. (2014) um grafo temporal pode ser defino formalmente como um conjunto C de contatos entre um conjunto de vértices V durante um conjunto de instantes de tempo T, representando duração do grafo. Um contato é uma tupla de quatro elementos (u, v, t, t'), onde $[t, t') \subset T \times T$ representa o intervalo de tempo onde a aresta $(u, v) \in E \subseteq V \times V$ está ativa. Dizemos que (u, v) está ativo em um tempo t se existe um contato $(u, v, t_s, t_e) \in C$ tal que $t \in [t_s, t_e)$. A Figura 1 mostra um exemplo de grafo temporal composto pelos contatos: (b, d, t_1, t_3), (c, d, t_2, t_3), (a, c, t_3, t_5) e (a, b, t_4, t_5).

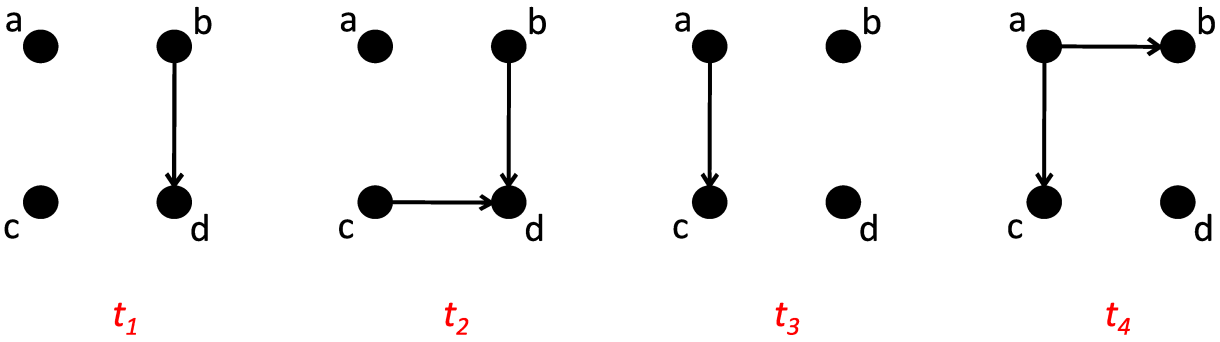

Figura 1 – Representação de um grafo temporal composto por quatro vértices e quatro instantes de tempo de duração. Figura adaptada de (BRISABOA et al., 2018).

2.3 Estruturas de dados compactas

Segundo Navarro (2016), uma estrutura de dados compacta mantém os dados e outras estruturas de dados internas em um formato que não apenas utiliza menos espaço, mas que também permite o acesso de forma compacta. Portanto, uma estrutura de dados compacta nos permite armazenar e consultar de maneira eficiente, conjuntos de dados muito maiores na memória principal do que seria possível utilizando os dados representados em sua forma original com estruturas de dados clássicas.

2.4 Bitvector

De acordo com Navarro (2016), *bitvectors* são fundamentais na implementação de diversas estruturas de dados compactas. Um *bitvector* pode ser descrito como um *array* de *bits* $B[1, n]$ que suporta as seguintes operações:

- $access(B, i)$: retorna o valor do *bit* $B[i]$, para qualquer $1 \leq i \leq n$.
- $rank_v(B, i)$: retorna o número de ocorrências de um *bit* $v \in \{0, 1\}$ em $B[1, i]$ para qualquer $0 \leq i \leq n$. Considere $rank_v(B, 0) = 0$. Caso v seja omitido considere $v = 1$.
- $select_v(B, i)$: retorna a posição em B da i-ésima ocorrência do *bit* $v \in \{0, 1\}$, para qualquer $i \geq 0$. Considere $select_v(B, 0) = 0$ e $select_v(B, i) = n + 1$ se $i > rank_v(B, n)$. Caso v seja omitido considere $v = 1$.

2.5 Suffix Array

Conforme o que foi explicado por Manber e Myers (1993), um *suffix array* pode ser descrito da seguinte maneira: considere $S[1, n]$ como um texto composto por caracteres do alfabeto Σ, onde $\Sigma = \{A, B, C, ..., Z\}$. Além disso, considere $S[n + 1] = \$$ como um terminador único, enfatizando que, $\forall\sigma \in \{A, B, C, ..., Z\}$, $\$ < \sigma$. Uma *substring* $S[i, n + 1]$ é denominada sufixo de S, onde $i \in \{1, 2, ..., n + 1\}$. Adiante, considere $A[1, n + 1]$ o *suffix array* de S; A é um *array* de números inteiros i representando os sufixos $S[i, n + 1]$. Os inteiros no *suffix array* A são ordenados conforme a ordem lexicográfica dos sufixos que eles representam. A ordem entre dois sufixos pode ser definida como $S[x, n + 1] < S[y, n + 1] \Leftrightarrow S[x] < S[y]$. Se $S[x] = S[y]$, então recursivamente verifica-se se $S[x + 1, n + 1] < S[y + 1, n + 1]$, onde $x \in \{1, 2, ..., n\}$ e $y \in \{1, 2, ..., n\}$. A singularidade de $S[n + 1]$ garante que nenhum outro sufixo compartilhe a mesma ordem lexicográfica. Adicionalmente, é importante notar que $A[1] = n + 1$, uma vez que $S[n + 1]$ é menor que qualquer outro caractere pertencente ao alfabeto Σ.

i	Sufixo ($S[i, n+1]$)
1	GATAGACA$
2	ATAGACA$
3	TAGACA$
4	AGACA$
5	GACA$
6	ACA$
7	CA$
8	A$
9	$

Tabela 1 – Sufixos do texto $S =$ `"GATAGACA$"`.

i	$A[i]$	Sufixo ($S[A[i], n+1]$)
1	9	$
2	8	A$
3	6	ACA$
4	4	AGACA$
5	2	ATAGACA$
6	7	CA$
7	5	GACA$
8	1	GATAGACA$
9	3	TAGACA$

Tabela 2 – *Suffix array* A construído a partir do texto $S =$ `"GATAGACA$"`.

Dentre as consultas que um *suffix array* consegue realizar, neste trabalho destaca-se a consulta que verifica se um padrão P de tamanho m existe no texto S retornando a posição em A onde o padrão é localizado. Essa operação realiza uma busca binária no *array* A utilizando o texto S para realizar as comparações do sufixo apontado por A com o padrão P. Essa operação é realizada em tempo $O(m \log n)$. Como os sufixos que correspondem ao padrão P estão dispostos de forma consecutiva em A, então é possível retornar um intervalo $[l, r]$ onde o padrão é encontrado, para $l \in \{1, 2, ..., n\}$, $r \in \{1, 2, ..., n\}$ e $l \leq r$.

Resumidamente, um *suffix array* mantém em memória um texto $S[1, n+1]$ e um *array* de inteiros $A[1, n+1]$, onde os inteiros em A apontam para posições em S. Em conclusão, o *suffix array* destaca-se como uma estrutura de dados altamente relevante, possibilitando buscas eficientes de padrões em textos.

2.6 Compressed Suffix Array

De acordo com o que foi explicado por Brisaboa et al. (2018) e proposto por Sadakane (2003) um *Compressed Suffix Array* (CSA) é uma representação compacta para o *suffix array*. A seguir, será apresentada uma explicação sobre a construção do CSA.

Considere um *suffix array* $A[1, n+1]$ associado a um texto $S[1, n+1]$. Para reduzir o armazenamento necessário para representar S e A, o CSA introduz uma *array* de inteiros $\Psi[1, n+1]$. Essa nova sequência é uma permutação de A, onde $z = \Psi[i]$, para $i \in \{1, 2, ..., n+1\}$, adicionalmente, $A[z] = A[i] + 1 = A[\Psi[i]]$. No caso em que $A[i] = n$, então $\Psi[i] = 1$.

Para que Ψ seja utilizado, duas novas estruturas se fazem necessárias, V e D. O *array* $V[1, \sigma']$ representa o vocabulário, onde V contém todos os símbolos utilizados em S ordenados lexicograficamente. Nesse contexto, σ' representa o tamanho do alfabeto de S. Adicionalmente, também é introduzido um *bitvector* $D[1, n+1]$, que indica o início dos intervalos apontados por A nos quais os primeiros símbolos desses sufixos coincidem. De maneira formal, $D[i] = 1$ se $i = 1$ ou $S[A[i]] \neq S[A[i+1]]$ para $i \in \{1, 2, ..., n\}$; $D[i] = 0$ caso contrário.

Em resumo, o CSA consiste em um *array* Ψ, um *bitvector* D e outro *array* V. Esses componentes possibilitam a realização de buscas binárias sem a necessidade de armazenar A ou S, pois $S[A[i]] = V[\text{rank}_1(D, i)]$. Além disso, observe que um símbolo $S[A[i]+1] = V[\text{rank}_1(D, \Psi[i])]$, $S[A[i]+2] = V[\text{rank}_1(D, \Psi[\Psi[i]])]$ e assim por diante. Dessa forma, utilizando apenas Ψ, D e V, é possível obter todos os símbolos de S necessários para a comparação com o padrão P em cada passo de uma busca binária. Vale destacar que, embora Ψ tenha o mesmo custo de espaço que A, sua alta compressibilidade permite uma significativa redução do consumo de memória, conforme descrito por (BRISABOA et al., 2018).

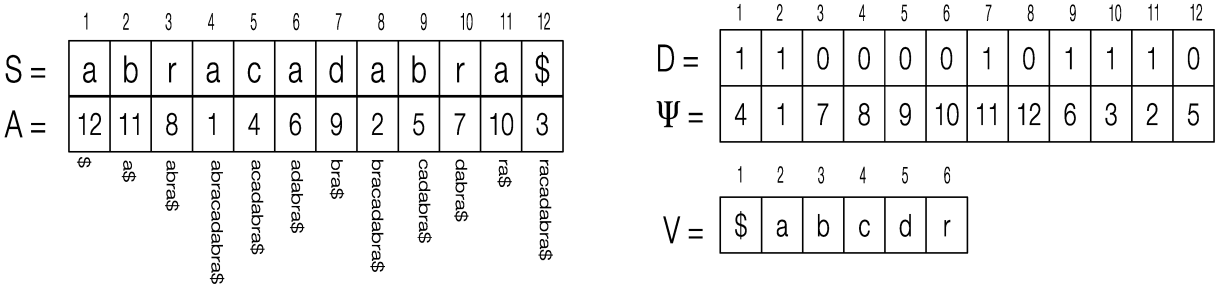

Figura 2 – Exemplo de um CSA construído para o texto $S =$ "abracadabra". A parte da esquerda mostra o *suffix array* A. Na direita é possível ver os componentes do CSA: a sequência Ψ, o *bitvector* D e o vocabulário V. Figura adaptada de (BRISABOA et al., 2018).

3 Desenvolvimento

3.1 Tecnologias

3.1.1 C++

Para o desenvolvimento deste TCC, a linguagem de programação C++ na sua versão 20 foi escolhida. Informações detalhadas sobre essa linguagem podem ser encontradas na fonte de referência C++20 (2023), que abrange os aspectos específicos relacionados à utilização do C++20. A escolha pelo C++ se deu por diversos motivos dentre os quais vale-se destacar:

- Eficiência de Desempenho: O C++ é uma linguagem de alto desempenho, o que é crucial para tarefas que envolvam um grande número de operações complexas, como estruturas de dados compactas e algoritmos em grafos temporais.
- Gerenciamento de memória manual: O C++ permite a alocação de memória manual, sem o uso de estratégias automáticas de *Garbage Collection*, o que é de grande valor para otimizar o consumo de RAM. Isso é particularmente relevante para este trabalho, onde um controle rigoroso sobre a alocação de memória é fundamental.
- Orientação a Objetos: Ao contrário da linguagem C, o C++ suporta a programação orientada a objetos, o que é vantajoso ao modelar estruturas de dados complexas, simplificando o processo de implementação.
- Ampla Biblioteca Padrão: C++ possui uma rica biblioteca padrão que oferece recursos prontos para uso, desta forma tornando o processo de desenvolvimento muito mais ágil.

3.1.2 CMake

De acordo com CMake Reference Documentation (2023), o CMake é uma ferramenta de código aberto amplamente utilizada para automatizar o processo de compilação e geração de projetos em linguagens como C, C++, e outras. Ele é empregado neste TCC devido à sua capacidade de proporcionar um sistema de compilação flexível e altamente portátil. Com o CMake, é possível definir as dependências, opções de compilação e configurações de projeto de forma independente da plataforma, simplificando a compilação em diferentes sistemas operacionais.

3.1.3 GoogleTest

O Google Test, também conhecido como *googletest*, é uma biblioteca de teste de código aberto amplamente utilizada em projetos de desenvolvimento de software, especialmente em C e C++. Informações detalhadas sobre o *googletest* podem ser encontradas no repositório da biblioteca (Google LLC, 2023). Essa biblioteca fornece uma estrutura flexível e eficaz para escrever testes unitários e de integração, permitindo que os desenvolvedores verifiquem a qualidade e o desempenho de seu código de maneira sistemática. A escolha de utilizar o *googletest* neste TCC se deve à sua sólida reputação e à capacidade de automatizar testes complexos, o que é fundamental para garantir a confiabilidade e a robustez das estruturas de dados desenvolvidas neste trabalho. O *googletest* simplifica o processo de criação e execução de testes, tornando-o essencial para a validação e verificação deste trabalho.

3.1.4 Valgrind

Conforme explicado por Nethercote, Walsh e Fitzhardinge (2006), o Valgrind é um *framework* de instrumentação binária dinâmica para a construção de ferramentas de análise de programas. Das diversas ferramentas que o Valgrind possui vale se destacar o Massif. O Massif é um *heap profiler*, com essa ferramenta é possível gerar um perfil detalhado da alocação de memória dinâmica, tirando capturas regulares da memória *heap* de um programa. O Massif gera um gráfico que mostra o uso de memória *heap* ao longo do tempo, incluindo informações sobre quais partes do programa são responsáveis pelas maiores alocações de memória. O gráfico é complementado por um arquivo de texto ou HTML que fornece informações adicionais para identificar onde a maior alocação de memória está ocorrendo. Vale ressaltar que o uso do Massif resulta em um aumento de aproximadamente 20 vezes no tempo de execução dos programas. Essa ferramenta desempenha um papel fundamental neste TCC para a análise da eficiência de memória dos programas desenvolvidos.

3.1.5 Shell Script

Segundo Parker (2011), o *shell* é a principal forma de comunicação com os sistemas Unix e Linux, oferecendo um meio direto de programação ao automatizar tarefas simples a intermediárias. *Shell scripts* são arquivos de texto que contêm uma sequência de comandos que podem ser executados em um ambiente de *shell*. Nesse TCC os scripts *shell* são aplicados para automatizar o processo de *benchmarking* das estruturas de dados desenvolvidas.

3.2 Implementação do TGCSA

Nesta seção serão apresentados os passos para a implementação do TGCSA, seguindo o que foi descrito por (BRISABOA et al., 2014). O código correspondente a essa implementação pode ser encontrado no repositório (SILVA, 2023).

3.2.1 Modificando CSA para o TGCSA

O principal componente de um TGCSA é o *Compressed Suffix Array* (CSA), mais especificamente um CSA baseado em números inteiros ou iCSA, entretanto, há uma importante diferença entre o CSA padrão e o CSA que é utilizado pelo TGCSA, a seguir estes detalhes serão descritos. Assuma que todos os termos em um contato estão contidos em quatro alfabetos disjuntos Σ_1, Σ_2, Σ_3 e Σ_4, de forma que $\Sigma_1 \prec \Sigma_2 \prec \Sigma_3 \prec \Sigma_4$, a notação $\prec$ denota a ordem lexicográfica entre os alfabetos, estabelecendo que os símbolos de um alfabeto são lexicograficamente menores aos do próximo alfabeto. O procedimento se inicia criando uma lista ordenada de n contatos, de modo que os contatos são ordenados pelo seu primeiro termo e em seguida (caso tenham o mesmo primeiro termo), pelo segundo termo e assim por diante. Esses contatos ordenados são representados como uma sequência com $4n$ elementos e um *suffix array* $A[1, 4n]$ é construído sobre essa representação. Note que como os valores de $\Sigma_i \prec \Sigma_j$ $(\forall i < j)$, as primeiras 25% das entradas em A $(A[1, n])$ apontarão para os primeiros termos de todos os contatos, as próximas n entradas $(A[n + 1, 2n])$ para os segundos termos e assim por diante. Consequentemente, as primeiras 25% das entradas de Ψ $(\Psi[1,\ n])$ apontarão para uma posição no intervalo $[n + 1, 2n]$, pois, na sequência indexada cada símbolo $u \in \Sigma_1$ é seguido por um símbolo $v \in \Sigma_2$ e assim por diante.

Observe que no CSA padrão se $A[i]$, onde $i \in [3n + 1, 4n]$, aponta para o último termo do j-ésimo contato, então $\Psi[i]$ armazena a posição em A que aponta para o primeiro termo do contato $(j + 1)$-ésimo, presente no intervalo $[1, n]$. No entanto, esses ponteiros serão modificados na última parte de Ψ $(\Psi[3n + 1, 4n])$, de modo que $A[\Psi[i]]$ deixe de apontar para o primeiro termo do contato seguinte e passe a apontar para o primeiro termo do mesmo contato.

Em consequência do que foi definido no parágrafo anterior é possível notar que começando por qualquer entrada i em Ψ e seguindo os ponteiros $\Psi[\Psi[\Psi[\Psi[i]]]]$, todos os elementos do contato atual podem ser recuperados, mas nenhum elemento de qualquer outra tupla será alcançada. Com esta modificação, não é mais possível percorrer todo o CSA apenas utilizando Ψ, pois, aplicações consecutivas de Ψ sempre obterão ciclicamente os quatro elementos do mesmo contato.

3.2.2 Construção detalhada do TGCSA

O primeiro passo para a construção do TGCSA é a criação de uma sequência S com n contatos ordenados. Portanto, considere $S[1, 4n] = \langle u^1, v^1, t_s^1, t_e^1, u^2, v^2, t_s^2, t_e^2, ..., u^n, v^n, t_s^n, t_e^n \rangle$.

Continuando, assuma que existem $\nu = |V|$ diferentes vértices e $\tau = |T|$ períodos de tempo. É possível definir uma função de mapeamento reversível que mapeia os termos de qualquer contato original $c = (u, v, t_s, t_e)$ para $c' = (u, v + \nu, t_s + 2\nu, t_e + 2\nu + \tau)$. Para alcançar isso, será preciso definir um *array* $gaps[1, 4] \leftarrow [0, \nu, 2\nu, 2\nu + \tau]$, dessa forma $c'[i] = c[i] + gaps[i]\ \forall i = 1...4$. Esse processo cria quatro intervalos no novo alfabeto Σ'. Perceba que com esse mapeamento um vértice i é mapeado para um inteiro i ou para um inteiro $i + \nu$ dependendo se é um vértice de origem, ou um vértice de destino. Similarmente, o instante de tempo de t é mapeado para t+$gaps[3]$ ou t+$gaps[4]$, lembrando que $gaps[3] = 2\nu$ e $gaps[4] = 2\nu + \tau$. Com esse mapeamento é possível distinguir entre vértices de origem e destino como também tempos de início e fim simplesmente verificando o intervalo onde o valor se encontra.

Note que o alfabeto Σ' possui lacunas, isto é, considere o conjunto ordenado $\alpha = \{a_1, a_2, a_3, ..., a_n\}$ construído a partir de Σ', onde $i \in \{2, 3, ..., n\}$ tal que $a_i - a_{i-1} > 1$. Para evitar as lacunas de Σ', um *bitvector* $B[1, 2\nu + 2\tau]$ é criado. No *bitvector* $B[i] \leftarrow 1$ se o símbolo i do alfabeto Σ' ocorre em um contato e $B[i] \leftarrow 0$ caso o contrário. Desta forma, cada um dos quatro termos de um contato (u, v, t_s, t_e) correspondem a um bit 1 em B. Para cada $\sigma \in \Sigma'$ uma função $mapID(\sigma)$ é definida, essa função retorna um inteiro id, tal que $id \leftarrow mapID(\sigma) = rank_1(B, \sigma)$ se $B[\sigma] = 1$; se $B[\sigma] = 0$ então $0 \leftarrow mapID(\sigma)$. O mapeamento inverso pode ser obtido pela função $unmapID(id)$, onde $\sigma = unmapID(id) = select_1(B, id)$.

Neste ponto é possível definir a sequência de *ids* $Sid[1, 4n]$, a criação de Sid se dá na seguinte forma, $Sid[i] \leftarrow mapID(S[i] + gaps[((i-1) \mod 4) + 1])\ \forall i = 1...4n$. Considere a variável *type* que representa os diferentes tipos de informações de um contato. Nesse contexto, $type = 1$ refere-se ao vértice de origem, $type = 2$ ao vértice de destino, $type = 3$ ao tempo de início e $type = 4$ ao tempo de fim. Dessa forma qualquer símbolo s de S pode ser mapeado para o valor corresponde em Sid pela função $getmap(valor, tipo)$, onde $id = getmap(s, type) \leftarrow rank_1(B, s + gaps[type])$. Similarmente, o mapeamento inverso é providenciado pela função $getunmap(id, type) \leftarrow select_1(B, id) - gaps[type]$.

Finalmente, um iCSA será construído sobre Sid. Observe que, uma vez que os alfabetos dos *ids* associados aos quatro termos de qualquer contato são disjuntos, o *array* de sufixos correspondente, A, possui quatro intervalos de comprimento n tal que $A[(j-1)n + 1, jn]$ para $j = 1...4$. Os ponteiros em cada um dos intervalos apontarão para os sufixos de vértice de origem, vértice de destino, tempo de início e tempo de fim, respectivamente. Igualmente, os valores em $\Psi[1, n]$ que estão no intervalo de vérti-

ces de origem, apontarão para o intervalo dos vértices de destino $[n+1, 2n]$. Valores em $\Psi[n+1, 2n]$ que estão no intervalo de vértices de destino, apontarão para o intervalo dos tempos de início $[2n+1, 3n]$. Da mesma forma, os valores em $\Psi[2n+1, 3n]$ que estão no intervalo de tempos de início, apontarão para o intervalo de tempos de fim $[3n+1, 4n]$. Por fim, os valores em $\Psi[3n+1, 4n]$ que estão no intervalo de tempos de fim, apontarão para o intervalo de vértices de origem $[1, n]$.

O *array* Ψ será modificado para o TGCSA, a fim de permitir movimentos cíclicos em Ψ de um termo para o próximo termo de um mesmo contato. Para fazer isso é necessário modificar os valores do Ψ originalmente definido, tal que, $\forall i = 3n+1...4n$, $\Psi[i] \leftarrow ((\Psi[i] - 2) \mod n) + 1$. Essa pequena alteração traz a interessante propriedade que possibilita realizar consultas por qualquer termo em um contato. O iCSA será utilizado para realizar buscas binárias, por qualquer termo de um contato, obtendo o intervalo $A[l, r]$ e então Ψ será aplicado circularmente até três vezes para obter os outros termos do contato.

Observe que, no contexto do iCSA, não é necessário armazenar o vocabulário de símbolos V, como ocorre em implementações tradicionais do CSA. Conforme discutido por Farina et al. (2012), no iCSA, o vocabulário torna-se implícito, eliminando a necessidade de sua preservação. Devido à ausência de lacunas no alfabeto utilizado para a construção do *array* Sid, podemos obter os seus valores apenas utilizando o *bitvector* D. Lembre-se que, se $D[i] = 1$ então $D[i+1] = 1$ caso $Sid[A[i]] \neq Sid[A[i+1]]$ e $D[i+1] = 0$ caso o contrário, ou seja, D é o *bitvector* que indica que as mudanças dos símbolos apontados por A, portanto, os valores originais de Sid podem ser obtidos da seguinte forma, $Sid[A[i]] = rank_1(D, i)$.

Em síntese, a representação do TGCSA consiste em um *bitvector* B, um *bitvector* D e um *array* de inteiros Ψ. Na implementação do TGCSA que foi realizada neste trabalho, apesar dos benefícios em relação à redução de consumo de memória, não foram empregadas técnicas de compactação para os *bitvectors* B e D. Da mesma forma, o *array* Ψ não foi implementado utilizando técnicas de compactação. Assim sendo, B e D foram implementados utilizando recursos nativos da linguagem C++ para representação de sequências de *bits*, assim como Ψ é armazenado em um contêiner padrão da linguagem. Um ponto que merece destaque é que os *bitvectors* B e D não utilizam nenhuma outra estrutura para auxiliar nas consultas de *rank* e *select*, o que implica que essas consultas são realizadas em tempo linear.

A figura a seguir ilustra todas as estruturas envolvidas na criação do TGCSA. Para o exemplo abaixo assuma um grafo temporal com $\nu = 5$ vértices enumerados de 1 a 5 e $\tau = 8$ instantes de tempos enumerados de 1 a 8. O grafo temporal contém os seguintes contatos: $(1, 3, 1, 8)$, $(1, 3, 5, 8)$, $(2, 1, 1, 5)$, $(4, 3, 7, 8)$ e $(4, 5, 5, 7)$. A Figura 3 também mostra o processo de recuperação do primeiro contato.

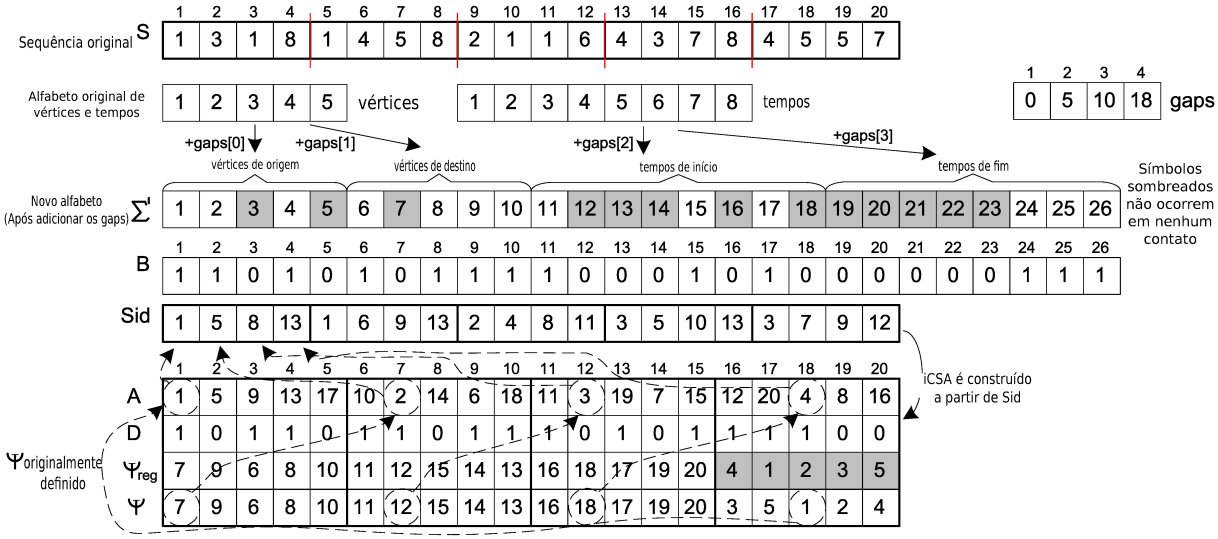

Figura 3 – Estruturas envolvidas no processo de criação do TGCSA. Figura adaptada de (BRISABOA et al., 2014).

3.2.3 Consultas no TGCSA

Dentre os diversos tipos de consultas suportadas por grafos temporais, este trabalho concentra-se na implementação e análise das consultas de vizinhos diretos e vizinhos inversos, a seguir é realizado um detalhamento sobre o funcionamento de tais consultas. A consulta de vizinhos diretos pode ser definida na forma da função $DirectNeighbors(vrtx, t)$, que aceita um vértice de origem $vrtx$ e um instante de tempo t como parâmetros, devolvendo todos os vértices de destino conectados a $vrtx$ durante o instante t. Em outras palavras, a função busca contatos $(vrtx, v, t_s, t_e)$, onde $t_s \leq t < t_e$. Neste ponto é necessário introduzir a função $CSA\_binSearch(pattern)$ que realiza uma busca binária pelo padrão $pattern$ utilizando o iCSA do TGCSA para obter o intervalo $A[l, r]$ onde o padrão ocorre. A consulta de vizinhos diretos é implementada obtendo inicialmente os intervalos $[lu, ru] \leftarrow CSA\_binSearch(getmap(vrtx, 1))$, $[lt_s, rt_s] \leftarrow CSA\_binSearch(getmap(t, 3))$ e $[lt_e, rt_e] \leftarrow CSA\_binSearch(getmap(t, 4))$. Note que a função $getmap$ que está sendo usada para mapear o valor de $vrtx$ do alfabeto original para o alfabeto disjunto Σ'. Finalmente, Ψ é aplicado circularmente, até duas ou três vezes para cada posição $i \in [lu, ru]$ e por meio dos intervalos $[lt_s, rt_s]$ e $[lt_e, rt_e]$ as condições de tempo de início e tempo de fim são verificadas. A seguir, a operação de vizinhos diretos é detalhada por meio de um pseudocódigo, baseando-se nas instruções fornecidas por (BRISABOA et al., 2018).

Vizinhos diretos (TGCSA)

Input: vértice de origem $vrtx$, tempo t
Output: *vizinhos* (v) de $vrtx$ no contato $(vrtx, v, t_1, t_2)$ tal que $t_1 \leq t < t_2$

```
1:  function DirectNeighbors(vrtx, t)
2:      u ← getmap(vrtx, type = 1);                ▷ mapeia para o alfabeto sem lacunas.
3:      if u = 0 then return ∅;      ▷ se vrtx não é encontrado como vértice de origem.
4:      end if

5:      vizinhos ← ∅;
6:      t_s ← getmap(t, type = 3);
7:      t_e ← getmap(t, type = 4);

8:      [lu, ru] ← CSA_binSearch(u);   ▷ intervalo A[lu, ru] para o vértice de origem u.
9:      [lt_s, rt_s] ← CSA_binSearch(t_s);  ▷ intervalo A[lt_s, rt_s] para o tempo de início t_s.
10:     [lt_e, rt_e] ← CSA_binSearch(t_e);    ▷ intervalo A[lt_e, rt_e] para o tempo de fim t_e.

11:     for i ← lu to ru do  ▷ verifica os intervalos de tempo para cada ocorrência de u.
12:         x ← Ψ[i];                                  ▷ x = posição do vértice de destino.
13:         y ← Ψ[x];                                    ▷ y = posição do tempo de início.
14:         if y ≤ rt_s then
15:             z ← Ψ[y];                                  ▷ z = posição do tempo de fim.
16:             if z > rt_e then
17:                 vizinhos ← vizinhos ∪ {getunmap(x, type = 2)};
18:             end if
19:         end if
20:     end for

21:     return vizinhos;
22: end function
```

O algoritmo para consulta de vizinhos diretos possui complexidade assintótica de $O(n \log n)$. Isso ocorre porque a parte do pseudocódigo que apresenta a maior complexidade é a função $CSA\_binSearch(pattern)$. Essa função realiza um busca binária no CSA por um padrão, a complexidade dessa operação é de $O(m \log n)$, onde n é o tamanho da sequência e m é o tamanho do padrão buscado. No entanto, para as funções de vizinhos diretos e vizinhos inversos, o padrão buscado sempre terá tamanho 1, resultando em uma complexidade de $O(\log n)$. Em cada passo da busca binária, uma consulta de *rank* é realizada no *bitvector* D. Essa é consulta linear e contribui para o custo total da função $CSA\_binSearch(pattern)$, tornando-a $O(n \log n)$, uma vez que D tem tamanho n. Vale destacar que neste caso intervalo retornado pela função $CSA\_binSearch(pattern)$ nunca será maior que $n/4$. Isso implica que a complexidade assintótica do laço executado sobre esse intervalo será sempre inferior a $O(n \log n)$.

A operação de vizinhos inversos é muito semelhante à operação de vizinhos diretos.

Para a operação de vizinhos inversos a função $ReverseNeighbors(vrtx, t)$ é definida, essa função recebe como parâmetro um vértice de destino $vrtx$ e um instante de tempo t e retorna todos os vértices de origem que estão conectados a $vrtx$ durante o instante t. Resumidamente, a função busca os contatos $(u, vrtx, t_s, t_e)$ tal que $t_s \leq t < t_e$. O procedimento inicial para a consulta dos vizinhos inversos requer a obtenção dos seguintes intervalos $[lv, rv] \leftarrow CSA\_binSearch(getmap(vrtx, 2))$, $[lt_s, rt_s] \leftarrow CSA\_binSearch(getmap(t, 3))$ e $[lt_e, rt_e] \leftarrow CSA\_binSearch(getmap(t, 4))$. Em seguida, Ψ é aplicado de maneira circular, até duas ou três vezes para cada posição $i \in [lv, rv]$ e por meio dos intervalos $[lt_s, rt_s]$ e $[lt_e, rt_e]$ as condições de tempo de início e tempo de fim são verificadas. A seguir, a operação de vizinhos inversos é detalhada por meio de um pseudocódigo, de acordo com o que foi apresentado por (BRISABOA et al., 2018).

Vizinhos inversos (TGCSA)

Input: vértice de destino $vrtx$, tempo t
Output: $vizinhos\_inv$ (u) de $vrtx$ no contato $(u, vrtx, t_1, t_2)$ tal que $t_1 \leq t < t_2$

```
1:  function ReverseNeighbors(vrtx, t)
2:      v ← getmap(vrtx, type = 2);                  ▷ mapeia para o alfabeto sem lacunas.
3:      if v = 0 then return ∅;      ▷ se vrtx não é encontrado como vértice de destino.
4:      end if

5:      vizinhos_inv ← ∅;
6:      t_s ← getmap(t, type = 3);
7:      t_e ← getmap(t, type = 4);

8:      [lv, rv] ← CSA_binSearch(v);   ▷ intervalo A[lv, rv] para o vértice de destino v.
9:      [lt_s, rt_s] ← CSA_binSearch(t_s);  ▷ intervalo A[lt_s, rt_s] para o tempo de início t_s.
10:     [lt_e, rt_e] ← CSA_binSearch(t_e);    ▷ intervalo A[lt_e, rt_e] para o tempo de fim t_e.

11:     for i ← lv to rv do  ▷ verifica os intervalos de tempo para cada ocorrência de v.
12:         y ← Ψ[i];                                  ▷ y = posição do tempo de início.
13:         if y ≤ rt_s then
14:             z ← Ψ[y];                                ▷ z = posição do tempo de fim.
15:             if z > rt_e then
16:                 u ← Ψ[z];                          ▷ u = posição do vértice de origem.
17:                 vizinhos_inv ← vizinhos_inv ∪ {getunmap(u, type = 1)};
18:             end if
19:         end if
20:     end for

21:     return vizinhos_inv;
22: end function
```

A complexidade assintótica para a consulta de vizinhos inversos é de $O(n \log n)$, seguindo as mesmas considerações elucidadas para a consulta de vizinhos diretos.

3.3 Grafos temporais utilizando estruturas de dados convencionais

Para estabelecer uma base de comparação, neste estudo foram desenvolvidas versões de grafos temporais utilizando estruturas de dados convencionais comumente empregadas na representação de grafos. As representações escolhidas incluem a lista de adjacência e a lista de arestas, enquanto a matriz de adjacência foi excluída da comparação devido ao seu significativo consumo de memória em relação às outras duas estruturas. A implementação da versão temporal dessas estruturas de dados é relativamente simples, exigindo apenas a adição da dimensão do tempo a tais estruturas. Nas subseções a seguir, as implementações dessas estruturas serão abordadas em mais detalhes.

3.3.1 Grafo temporal com listas de adjacência

De acordo com Halim et al. (2013), em uma lista de adjacência, temos um *array* de *array* que armazena uma lista de vizinhos para cada vértice do grafo. Adicionando a dimensão do tempo a essa estrutura, um *array* de *array* de *array* é obtido, onde cada momento no tempo é uma entrada no primeiro *array*, cada vértice do grafo é uma entrada do segundo *array* e o terceiro *array* corresponde à lista de vizinhos de um determinado vértice. A seguir, apresenta-se um pseudocódigo que demonstra a operação de vizinhos diretos para um grafo temporal com listas de adjacência.

Vizinhos diretos (listas de adjacência)

Input: vértice de origem $vrtx$, tempo t
Output: $vizinhos$ (v) de $vrtx$ no contato $(vrtx, v, t_1, t_2)$ tal que $t_1 \leq t < t_2$

```
1: function DirectNeighbors(vrtx, t)
2:     vizinhos ← ∅;
3:     N ← lenght(graph[t][vrtx]);  ▷ Considere graph[T][U][V] como o grafo temporal
       com listas de adjacência. Adicionalmente, considere lenght(x) como uma função que
       retorna a quantidade de elementos de um array x.
4:     for i ← 1 to N do
5:         vizinhos ← vizinhos ∪ graph[t][vrtx][i];
6:     end for
7:     return vizinhos;
8: end function
```

A complexidade assintótica do algoritmo apresentado anteriormente é de $O(\nu)$. Pois, no pior dos casos o vértice $vrtx$ vai estar conectado a todos os outros vértices durante o instante de tempo t, isto é, $vrtx$ possui $\nu - 1$ vizinhos diretos, lembrando que $\nu = |V|$.

3.3.2 Grafo temporal com listas de arestas

Segundo Halim et al. (2013), uma lista de arestas pode ser armazenada na forma de uma lista de pares de inteiros, ou seja, um *array* de tuplas. Ao adicionar a dimensão do tempo a essa estrutura, um *array* de *array* de tuplas é obtido, onde cada momento no tempo é uma entrada no primeiro *array*, e o segundo *array* corresponde à lista de tuplas, onde cada tupla representa uma aresta ativa para um determinado momento do tempo. A seguir, um pseudocódigo que demonstra a operação de vizinhos diretos para um grafo temporal com listas de arestas é apresentado a seguir.

Vizinhos diretos (lista de arestas)

Input: vértice de origem $vrtx$, tempo t
Output: $vizinhos$ (v) de $vrtx$ no contato $(vrtx, v, t_1, t_2)$ tal que $t_1 \leq t < t_2$

1: **function** DIRECTNEIGHBORS($vrtx, t$)
2: $vizinhos \leftarrow \emptyset$;
▷ Considere $graph[T][E]$ como o grafo temporal com lista de arestas. Adicionalmente, considere as funções $first(x)$ e $second(x)$ que respectivamente retornam o primeiro e segundo elemento de um tupla x.
3: **for** $i \in graph[t]$ **do**
4: **if** $first(i) = vrtx$ **then**
5: $vizinhos \leftarrow vizinhos \cup second(i)$;
6: **end if**
7: **end for**
8: **return** $vizinhos$;
9: **end function**

A complexidade assintótica do algoritmo apresentado anteriormente é de $O(\epsilon)$. Visto que é preciso verificar cada uma das arestas ativas no instante de tempo t, lembrando que $\epsilon = |E|$.

4 Resultados

Este capítulo descreve todos os resultados derivados dos testes conduzidos. Além disso, serão fornecidas informações detalhadas sobre a infraestrutura utilizada para conduzir os testes de desempenho comparativo, assim como uma explicação abrangente do método de teste adotado.

4.1 Ambiente de testes

Os testes foram conduzidos utilizando o programa desenvolvido neste trabalho, que, como mencionado anteriormente, está disponível no repositório (SILVA, 2023). Para a realização dos testes, a máquina utilizada estava equipada com um processador AMD Ryzen 7 5700X (16) @ 3.400GHz e 32GB de memória RAM com frequência de 3200 MHz. O sistema operacional utilizado foi o Pop!_OS 22.04 LTS x86_64, com o *kernel* 6.5.6-76060506-generic, juntamente com o *shell* zsh 5.8.1 e o compilador g++ (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0.

4.2 Metodologia de testes

Para conduzir os testes, um gerador foi implementado, que gera um conjunto de contatos e consultas de maneira aleatória. O gerador opera por meio de uma interface de linha de comando, onde é possível especificar o número de contatos a serem gerados, o número de consultas, assim como os valores máximos para um vértice e tempo. Tanto os contatos quanto as consultas geradas foram salvos em arquivos, possibilitando sua reutilização para realizar diversos testes. Em relação às consultas, estas são criadas utilizando os vértices existentes no conjunto de contatos aleatórios gerados anteriormente. As consultas consistem nas operações de vizinhos diretos e vizinhos inversos.

Para realizar o *benchmark*, o programa desenvolvido é executado utilizando a ferramenta Massif do Valgrind, com os contatos e as consultas aleatórias geradas previamente pelo gerador. Adicionalmente, um *script shell* é utilizado para automatizar o processo de *benchmarking* e facilitar os testes.

4.3 Resultados dos testes

A seguir, serão apresentados os resultados de dois testes realizados. O primeiro utilizou um conjunto de dados menor, resultando em tempos de execução inferiores a 1

minuto. O segundo teste fez uso de um conjunto de dados relativamente maior, proporcionando uma análise mais abrangente do desempenho das estruturas.

Grafo temporal	Tempo de execução (ms)	*Heap max* (*bytes*)	*Heap med* (*bytes*)
Lista de adjacência	419	1.305.444	740.000
Lista de arestas	356	633.081	222.660
TGCSA	44.800	722.517	179.260

Tabela 3 – Resultados dos testes com 3.000 contatos, 300 consultas, onde os valores dos vértices variam entre 1 e 1.000 e os valores dos tempos variam entre 1 e 40. O termo *Heap max* representa o consumo máximo em *bytes* da memória *heap*, enquanto *Heap med* representa a média em *bytes* da memória da memória *heap* utilizada durante a execução do teste.

Figura 4 – Consumo de memória *heap* ao longo do tempo (lista de adjacência) referente a Tabela 3.

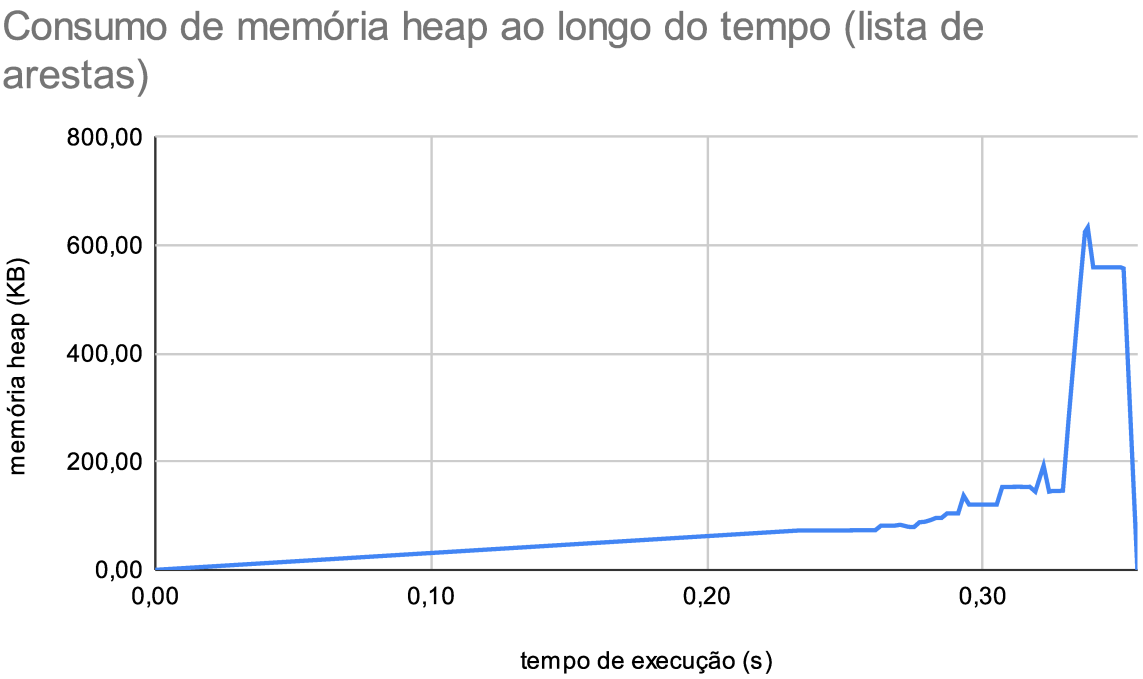

Figura 5 – Consumo de memória *heap* ao longo do tempo (lista de arestas) referente a Tabela 3.

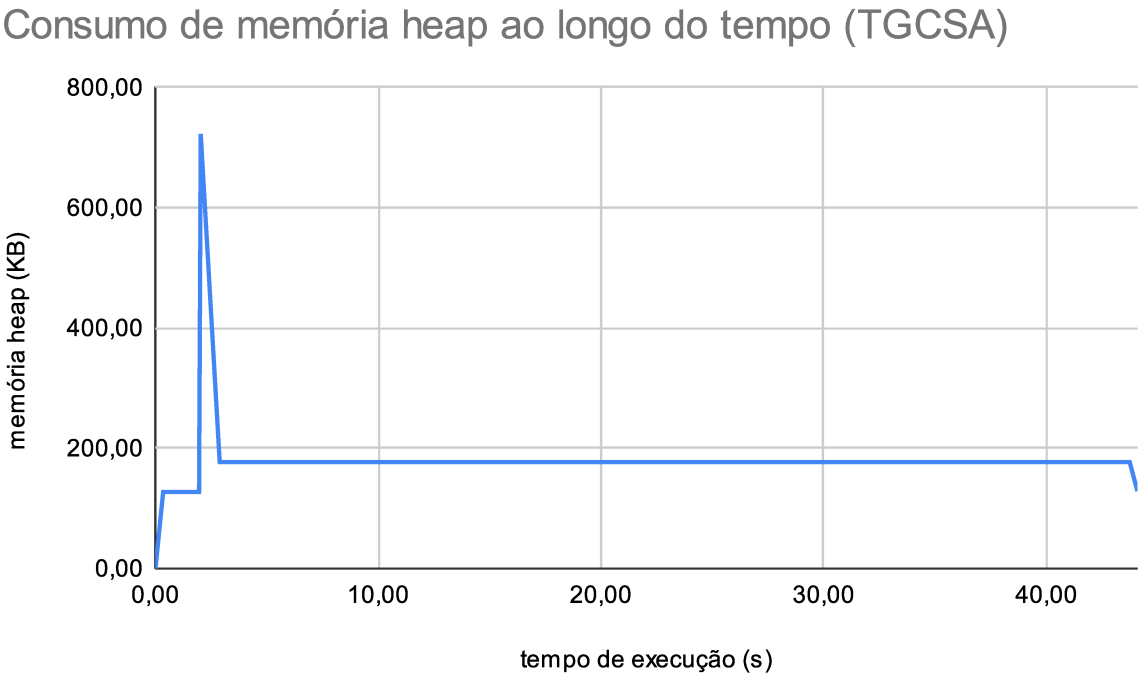

Figura 6 – Consumo de memória *heap* ao longo do tempo (TGCSA) referente a Tabela 3.

Grafo temporal	Tempo de execução (ms)	*Heap max* (*bytes*)	*Heap med* (*bytes*)
Lista de adjacência	10.680	114.438.176	85.110.000
Lista de arestas	2.416	42.021.561	26.380.000
TGCSA	5.101.200	6.424.125	1.150.000

Tabela 4 – Resultados dos testes com 30.000 contatos, 3.000 consultas, onde os valores dos vértices variam entre 1 e 10.000 e os valores dos tempos variam entre 1 e 400. O termo *Heap max* representa o consumo máximo em *bytes* da memória *heap*, enquanto *Heap med* representa a média em *bytes* da memória da memória *heap* utilizada durante a execução do teste.

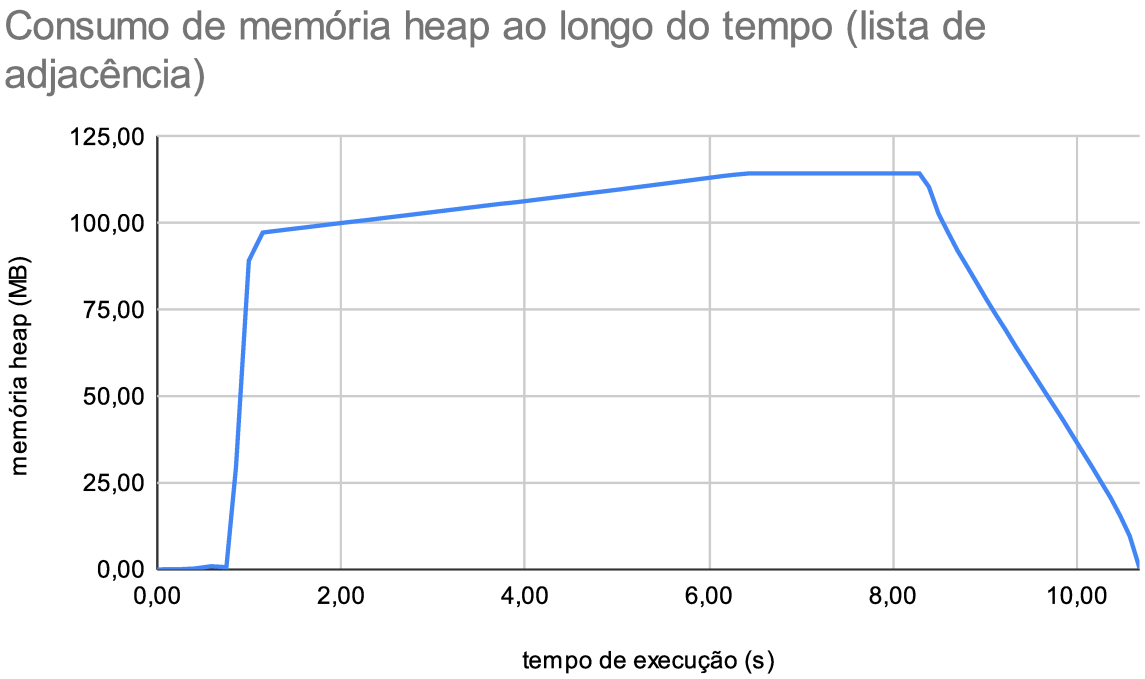

Figura 7 – Consumo de memória *heap* ao longo do tempo (lista de adjacência) referente a Tabela 4.

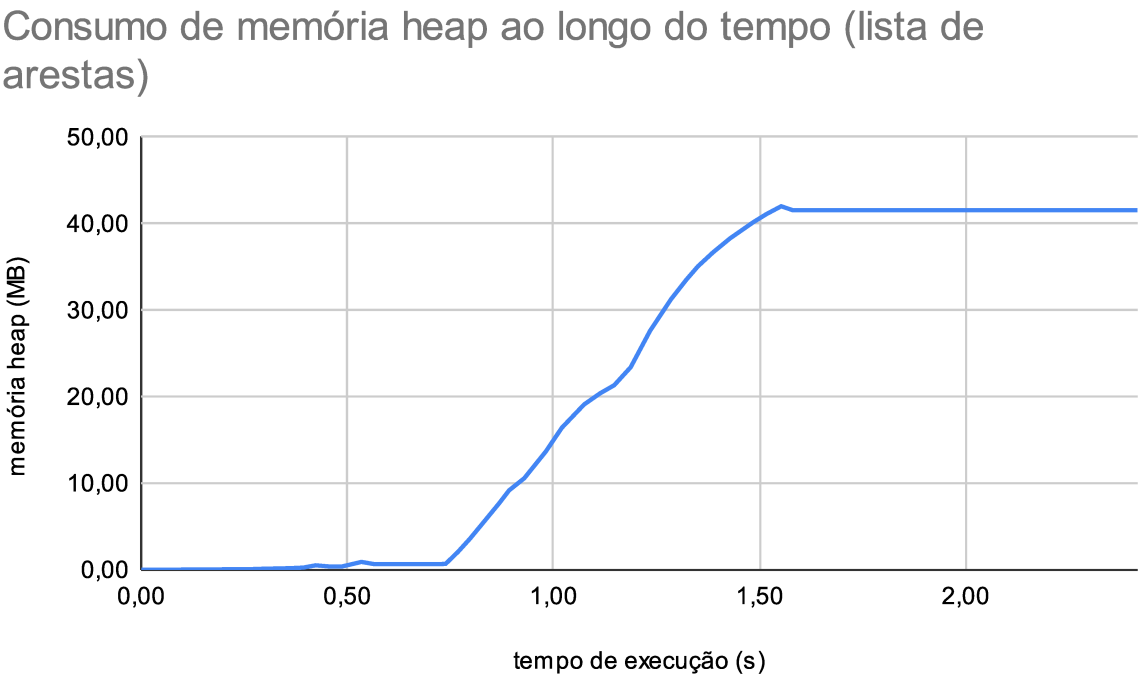

Figura 8 – Consumo de memória *heap* ao longo do tempo (lista de arestas) referente a Tabela 4.

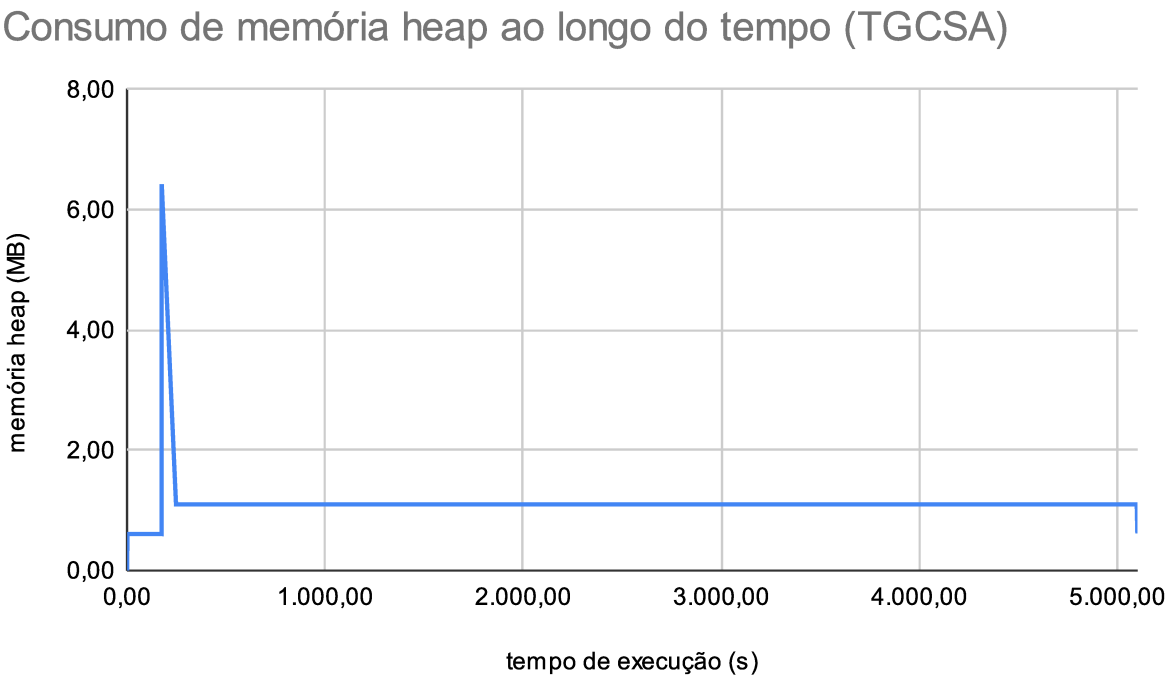

Figura 9 – Consumo de memória *heap* ao longo tempo do (TGCSA) referente a Tabela 4.

4.4 Considerações finais acerca dos testes

Primeiramente é importante observar que no caso do TGCSA, há um pico no consumo de memória durante a criação da estrutura de dados, após isso os dados desnecessários são desalocados, resultando em um consumo de memória significativamente menor durante a execução das consultas. Dessa forma, a média do consumo de memória da *heap* torna-se uma métrica mais interessante para a realização de comparações. Além disso, também é possível notar que o grafo temporal baseado em listas de adjacência apresentou um desempenho em memória significativamente inferior em comparação com as outras duas estruturas implementadas nesse trabalho. Com o objetivo de concisão, as comparações subsequentes se concentrarão principalmente no TGCSA e no grafo temporal baseado em listas de arestas.

No primeiro teste, devido ao conjunto de dados pequeno, o TGCSA apresentou um consumo máximo de memória aproximadamente 14% maior que o grafo temporal baseado em listas de arestas. Entretanto, ao considerar a média do consumo de memória, o TGCSA mostrou-se cerca de 19% mais eficiente que o grafo temporal baseado em listas de arestas. Quanto ao tempo de execução, o TGCSA apresentou um tempo aproximadamente 11 vezes maior em relação as outras duas estruturas testadas.

No segundo teste, conduzido com um conjunto de dados maior, o TGCSA surge como uma opção mais atrativa para a economia no uso de memória. Nesse contexto, o TGCSA apresenta um consumo médio de memória quase 23 vezes inferior ao espaço consumido pela estrutura baseada na lista de arestas. Em contrapartida, é importante notar que o TGCSA possui um tempo de execução aproximadamente 2.111 vezes superior ao tempo de execução do grafo temporal baseado em listas de arestas.

5 Conclusão e trabalhos futuros

Considerando os resultados obtidos, conclui-se que o TGCSA se destaca pela eficiência no consumo de memória em relação a outras estruturas de dados para grafos temporais utilizadas no teste. Contudo, é crucial reconhecer que essa vantagem vem acompanhada de um comprometimento no tempo de execução.

Para futuras pesquisas é recomendável o aprimoramento da implementação do TGCSA realizada neste trabalho, visando aumentar sua eficiência de memória por meio da compactação do *array* Ψ e a aplicação de outras técnicas relacionadas a estruturas de dados compactas. Além disso, seria importante explorar maneiras de aprimorar a eficiência do tempo de execução, dando especial ênfase ao uso de outras estruturas de *bitvectors* capazes de realizar a operação de *rank* em tempo constante, como também a priorização do uso de algoritmos mais eficazes para a ordenação de sufixos. Essas melhorias têm o potencial de elevar ainda mais o desempenho do TGCSA, contribuindo para uma implementação mais robusta e eficaz em diferentes cenários de aplicação.

Referências

BRISABOA, N. R.; CARO, D.; FARIÑA, A.; RODRÍGUEZ, M. A. A compressed suffix-array strategy for temporal-graph indexing. In: MOURA, E.; CROCHEMORE, M. (Ed.). **String Processing and Information Retrieval**. Cham: Springer International Publishing, 2014. p. 77–88. ISBN 978-3-319-11918-2. Disponível em: <https://doi.org/10.1007/978-3-319-11918-2_8>. Citado 4 vezes nas páginas 9, 10, 16 e 19.

BRISABOA, N. R.; CARO, D.; FARINA, A.; RODRIGUEZ, M. A. Using compressed suffix-arrays for a compact representation of temporal-graphs. **Information Sciences**, Elsevier, v. 465, p. 459–483, 2018. Disponível em: <https://doi.org/10.1016/j.ins.2018.07.023>. Citado 5 vezes nas páginas 10, 12, 13, 19 e 21.

C++20. **C++20**. 2023. Acessado em: 03 de novembro de 2023. Disponível em: <https://en.cppreference.com/w/cpp/20>. Citado na página 14.

CMake Reference Documentation. **CMake Reference Documentation**. 2023. Acessado em: 03 de novembro de 2023. Disponível em: <https://cmake.org/cmake/help/latest/index.html>. Citado na página 14.

FARINA, A.; BRISABOA, N. R.; NAVARRO, G.; CLAUDE, F.; PLACES, A. S.; RODRÍGUEZ, E. Word-based self-indexes for natural language text. **ACM Transactions on Information Systems (TOIS)**, ACM New York, NY, USA, v. 30, n. 1, p. 1–34, 2012. Disponível em: <https://doi.org/10.1145/2094072.2094073>. Citado na página 18.

Google LLC. **googletest**. 2023. Acessado em: 03 de novembro de 2023. Disponível em: <https://github.com/google/googletest>. Citado na página 15.

HALIM, S.; HALIM, F.; SKIENA, S. S.; REVILLA, M. A. **Competitive programming 3**. [S.l.]: Citeseer, 2013. Citado 2 vezes nas páginas 22 e 23.

MANBER, U.; MYERS, G. Suffix arrays: a new method for on-line string searches. **siam Journal on Computing**, SIAM, v. 22, n. 5, p. 935–948, 1993. Disponível em: <https://doi.org/10.1137/0222058>. Citado na página 11.

NAVARRO, G. **Compact data structures: A practical approach**. Cambridge University Press, 2016. ISBN: 9781316588284. Disponível em: <https://doi.org/10.1017/CBO9781316588284>. Citado 3 vezes nas páginas 9, 10 e 11.

NETHERCOTE, N.; WALSH, R.; FITZHARDINGE, J. Building workload characterization tools with valgrind. In: IEEE. **2006 IEEE International Symposium on Workload Characterization**. 2006. p. 2–2. Disponível em: <https://doi.org/10.1109/IISWC.2006.302723>. Citado na página 15.

PARKER, S. **Shell Scripting: Expert Recipes for Linux, Bash, and more**. 1. ed. [S.l.]: Wrox, 2011. ISBN: 978-1118024485. Citado na página 15.

SADAKANE, K. New text indexing functionalities of the compressed suffix arrays. **Journal of Algorithms**, Elsevier, v. 48, n. 2, p. 294–313, 2003. Disponível em: <https://doi.org/10.1016/S0196-6774(03)00087-7>. Citado na página 12.

SILVA, M. **Desenvolvimento e Avaliação do TGCSA para Redução de Consumo de Memória em Grafos Temporais**. 2023. Disponível em: <https://github.com/MateusFerreiraSilva/TCC-TGCSA>. Citado 2 vezes nas páginas 16 e 24.