

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Gustavo Nunes de Oliveira

**Avaliação Experimental de Estruturas de Dados
Compactas na Construção de Grafos Temporais
Compactos**

Uberlândia, Brasil

2021

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Gustavo Nunes de Oliveira

**Avaliação Experimental de Estruturas de Dados
Compactas na Construção de Grafos Temporais
Compactos**

Trabalho de conclusão de curso apresentado
à Faculdade de Computação da Universidade
Federal de Uberlândia, como parte dos requi-
sitos exigidos para a obtenção do título de
Bacharel em Ciência da Computação.

Orientador: Prof. Marcelo Keese Albertini

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Ciência da Computação

Uberlândia, Brasil

2021

Sumário

1	TABELA DE SÍMBOLOS	5
2	INTRODUÇÃO	6
3	CONCEITOS FUNDAMENTAIS	8
3.1	Array	8
3.1.1	Arrays fixos	9
3.1.1.1	Operação <i>read</i>	11
3.1.1.2	Operação <i>write</i>	12
3.1.2	Arrays variados	14
3.1.2.1	<i>Sampled pointers</i>	14
3.1.2.2	<i>Dense pointers</i>	15
3.2	Bitvector	16
3.2.1	Operação <i>rank</i>	17
3.2.2	Operação <i>select</i>	18
3.3	Wavelet tree	20
3.3.1	Construção	21
3.3.1.1	Análise	22
3.3.2	Operação <i>rank</i>	23
3.3.2.1	Análise	24
3.3.3	Operação <i>select</i>	24
3.3.3.1	Análise	25
4	GRAFOS TEMPORAIS COMPACTOS	26
4.1	Estrutura de dados <i>Adjacency Log of Events (EveLog)</i>	27
4.1.1	Construção	28
4.1.1.1	Análise	28
4.1.2	Operação <i>has_edge</i>	28
4.1.2.1	Análise	29
4.1.3	Operação <i>neighbours</i>	29
4.1.3.1	Análise	29
4.2	Estrutura de dados <i>Time-Interval Log Per Edge (EdgeLog)</i>	30
4.2.1	Construção	31
4.2.1.1	Análise	31
4.2.2	Operação <i>has_edge</i>	32
4.2.2.1	Análise	32

4.2.3	Operação <i>neighbours</i>	32
4.2.3.1	Análise	32
4.3	Estrutura de dados <i>Compact Adjacency Sequence (CAS)</i>	33
4.3.1	Construção	34
4.3.1.1	Análise	34
4.3.2	Operações extras para a <i>wavelet tree</i>	35
4.3.2.1	Operação <i>range_next_value_pos</i>	35
4.3.2.2	Operação <i>range_count</i>	36
4.3.2.3	Operação <i>range_report</i>	36
4.3.3	Operação <i>has_edge</i>	36
4.3.3.1	Análise	37
4.3.4	Operação <i>neighbours</i>	37
4.3.4.1	Análise	37
4.4	Considerações finais	37
5	EXPERIMENTAÇÃO	39
5.1	Detalhes da implementação	39
5.2	Resultados	40
5.2.1	Análise	41
6	CONCLUSÕES	44
	REFERÊNCIAS	45

Resumo

Com o aumento na geração de dados causado pelos avanços tecnológicos das últimas décadas, é crucial que seja possível armazenar e manipular tais dados em memória principal através de estruturas de dados que armazenam as informações de forma eficiente.

Diversas estruturas de dados foram criadas para representar grafos temporais utilizando estruturas de dados compactas, visando alcançar uma diminuição no uso de memória mantendo o desempenho em tempo para realizar consultas no grafo. Dentre as estruturas de dados para grafos temporais analisadas neste trabalho estão as listas de adjacência tradicionais, assim como as estruturas *Adjacency Log of Events (EveLog)* (CARO; RODRÍGUEZ; BRISABOA, 2015), *Time-Interval Log Per Edge (EdgeLog)* (XUAN; FERREIRA; JARRY, 2003) e *Compact Adjacency Sequence (CAS)* (CARO; RODRÍGUEZ; BRISABOA, 2015), que utilizam representações compactas de *arrays* e *bitvectors*, assim como *wavelet trees* para sua construção.

Devido à escassez de referências de implementações das estruturas de grafos temporais compactos analisadas neste trabalho, as mesmas foram implementadas sem um referencial prático, baseadas apenas nas suas respectivas propostas teóricas. Foi utilizado no projeto o paradigma de orientação a objetos, assim como testes unitários para garantir que as operações foram implementadas de maneira correta. Além disso, toda a implementação foi feita pensando em obter um bom desempenho, de forma modularizada, clara e de fácil manutenção.

Após a implementação das estruturas de dados analisadas no trabalho utilizando as técnicas escolhidas, foram executados experimentos para analisar a eficiência do uso de memória e tempo médio por consulta para cada estrutura de grafos temporais, utilizando geradores de grafos temporais aleatórios para gerar os dados de entrada para os testes. Os experimentos mostraram um resultado inferior das implementações de estruturas de grafos temporais compactos em relação à abordagem utilizando listas de adjacência tradicionais, tanto em uso de memória quanto em desempenho por operação, o que indica que as implementações feitas neste projeto não conseguiram alcançar na prática o resultado previsto pelas estruturas teoricamente.

Entretanto, ainda que os resultados obtidos não satisfizeram o que era proposto teoricamente, as implementações feitas podem servir como referencial para trabalhos futuros, para buscar alcançar o que foi proposto de forma teórica na prática ou analisar estruturas semelhantes que não fizeram parte do escopo deste trabalho.

Palavras-chave: estruturas de dados, grafo, temporal, compacto, experimental.

1 Tabela de símbolos

Símbolo	Significado
$\log X$	Equivalente a $\log_2 X$.
w	Tamanho da palavra utilizada pelo processador.
$[a, b]$	Intervalo fechado de números inteiros $[a, b]$.
$[a, b)$	Intervalo semi-aberto de números inteiros $[a, b)$.
$A[a, b]$	Sub-intervalo da sequência A , composto pelos elementos $A[i], \forall i \in [a, b]$.
$A[i]$	Elemento de índice i na sequência A . Considera-se $0 \leq i < n$, onde n indica o tamanho de A .
$O(f(n))$	Notação O de análise assintótica.
$o(g(n))$	Notação o de análise assintótica, tal que se $f(n) = o(g(n))$, então $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$.
$len(x)$	Indica quantos bits são necessários para representar o valor $x \in \mathbb{N}$ em binário. Equivalente a $\lfloor \log_2 x \rfloor + 1$ para $x > 0$ e $len(x) = 1$ para $x = 0$.

Tabela 1 – Tabela de símbolos utilizados no decorrer deste trabalho.

2 Introdução

A quantidade de dados produzida hoje em dia está aumentando mais rapidamente que a capacidade de processar tais dados. Seja um super computador processando dados sobre astronomia, ou um *smartwatch* processando dados do seu usuário em ambos os casos há uma grande quantidade de dados a serem tratados em proporção à capacidade de armazenamento e processamento dos dispositivos.

Uma abordagem desejável é a manipulação dos dados em memória principal por conta dos menores tempos de acesso. Entretanto, é comum não ser possível armazenar uma grande quantidade de dados em memória principal, sendo necessário realizar acessos mais frequentes em memória secundária, que possuem um tempo de acesso maior. Por exemplo, os discos rígidos operam na casa dos milisegundos por acesso (NEDEV; KAMENOV, 2018) enquanto que as memórias voláteis atuais como a SDRAM DDR4 realizam operações de entrada e saída com latência na casa dos nanosegundos¹ (aproximadamente um milhão de vezes mais eficiente).

Dentre as possíveis estratégias para diminuir o impacto desse problema, inclui-se o desenvolvimento de estruturas de dados compactas, onde o objetivo é armazenar e manipular a informação diretamente na forma compactada. Essas estruturas buscam manter o desempenho semelhante ao de estruturas tradicionais, porém reduzindo o uso de memória em várias ordens de magnitude. Por exemplo, teoricamente são necessários $Z \leq \log C_n = \log \frac{1}{N+1} + \log \binom{2N}{N}$ bits para representar uma árvore binária enraizada de n nós, onde C_n é o n -ésimo número de Catalão, que representa o número de árvores binárias enraizadas com n nós. Suponha que Z é o limite teórico de bits necessários para armazenar um certo conjunto de dados. Existem 3 classificações de estruturas de dados que visam o uso de menos memória, a primeira e mais econômica garante o uso de $Z + O(1)$ bits e é chamada de implícita. A segunda é chamada de sucinta e garante o uso de $Z + o(Z)$ bits e a terceira é chamada de compacta e garante o uso de $Z + O(Z)$ bits (JACOBSON, 1988).

Diversas estruturas de dados foram desenvolvidas para reduzir o uso de bits pelas estruturas de dados convencionais entre elas podemos citar: *arrays* (NAVARRO, 2016a), *bit vectors* (NAVARRO, 2016a), árvores (JACOBSON, 1989) e grafos (NAVARRO, 2016a). Recentemente, a comunidade científica tem proposto soluções para estruturas de dados compactas para grafos que registram informações de entidades e relacionamentos durante períodos de tempo. Um exemplo desse cenário é a modelagem de ligações entre usuários de telefonia, onde uma aresta entre dois indivíduos existe durante o tempo em que eles se encontram em uma ligação. Esse tipo de grafo temporal apresenta desafios para ser repre-

¹ Acessado em 15/10/2021: <<https://web.archive.org/web/20211015172302/https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/Secondary.html>>

sentado utilizando poucos bits e realizando as operações de forma eficiente. Um exemplo de desafio é reduzir número de ponteiros utilizados, o que em contrapartida resulta em uma perda de dinamicidade. Outro desafio é diminuir o espaço utilizado pela estrutura, sem que haja perda significativa de desempenho em tempo por consulta.

Dentre os estudos mais recentes sobre o tema podemos destacar o desenvolvimento de algoritmos como o TGCSA (SADAKANE, 2003; BRISABOA et al., 2014), cuja ideia principal é representar o grafo como uma *string* e construir um *suffix array* sobre ela, com o objetivo de reduzir uma operação no grafo a problemas de *string matching*. Também podemos destacar a utilização de estruturas de dados como a *wavelet tree* (GROSSI; GUPTA; VITTER, 2003), que permite realizar operações eficientes baseadas em *bitvectors*, possibilitando a representação de grafos temporais utilizando *compact adjacency sequences (CAS)* (CARO; RODRÍGUEZ; BRISABOA, 2015).

Neste trabalho de conclusão de curso, foram analisadas empiricamente as estruturas de dados compactas mais comuns utilizadas para representar grafos temporais. São elas: *EdgeLog - Time-interval Log per Edge* (XUAN; FERREIRA; JARRY, 2003) que se baseia em uma lista de adjacência em dois níveis, *EveLog - Adjacency Log of Events* (CARO; RODRÍGUEZ; BRISABOA, 2015) cuja abordagem considera arestas temporais na forma de eventos temporais e *CAS - Compact Adjacency Sequence* (CARO; RODRÍGUEZ; BRISABOA, 2015) que representa o grafo como um *array* manipulado por uma *wavelet tree*.

Dado o aumento na quantidade de estruturas de dados compactas sendo desenvolvidas, é difícil selecionar a mais adequada para o desenvolvimento de um projeto específico. Com o intuito de auxiliar na escolha da estrutura de dados ideal para uma aplicação, este trabalho teve como objetivo avaliar e comparar o desempenho de estruturas de dados compactas do ponto de vista prático, para a implementação de grafos temporais.

3 Conceitos Fundamentais

Para o estudo das estruturas de representação de grafos temporais apresentadas no Capítulo 4 é necessário primeiramente conhecer as estruturas de dados elementares que serão utilizadas. As estruturas aqui chamadas de elementares são: *arrays*, *bitvectors* e *wavelet trees*. Estas estruturas são dependentes entre si (veja Figura 1), portanto recomenda-se a leitura seguindo a ordem dessas dependências, para uma melhor compreensão das estruturas apresentadas.

$$Array \longrightarrow BitVector \longrightarrow WaveletTree$$

Figura 1 – Dependências entre as estruturas elementares utilizadas na construção de estruturas de representação de Grafos Temporais Compactos.

3.1 *Array*

Arrays são estruturas de dados fundamentais para a construção de outras estruturas mais complexas. O objetivo principal de um *array* é armazenar dados em sequência na memória principal, de forma que eles possam ser acessados posteriormente durante a execução do programa.

Um *array* suporta as seguintes operações:

- $read(A, x)$

A operação *read* permite o acesso ao valor do *array* A que está armazenado na posição $x \in [0, n)$, onde n é o número de elementos de A .

- $write(A, x, val)$

A operação *write* permite atribuir o valor $val \in \mathbb{N}$ para a posição $x \in [0, n)$ do *array* A , substituindo o valor anterior.

Os *arrays* comuns suportados pela maioria das linguagens de programação armazenam tipos de dados que ocupam uma quantidade de *bits* fixa pré-definida. Por exemplo, para a criação de um *array* com capacidade de armazenar n inteiros em C++ utiliza-se $32n$ *bits*, uma vez que cada inteiro ocupa 4 *bytes* = 32 *bits* de memória. O menor tipo de

dados é o tipo *char* que utiliza 1 *byte* = 8 *bits*, portanto a menor quantidade de memória necessária para armazenar um *array* de n elementos utilizando os métodos padrão das linguagens de programação é $8n$ *bits*.

É comum que os valores armazenados não ocupem exatamente a quantidade de *bits* atribuída pela linguagem de programação. Por exemplo, para armazenar um valor *booleano* é necessário apenas 1 *bit*, portanto seria possível criar um *array* de apenas n *bits* para esse propósito. Porém, como mencionado no parágrafo anterior, o menor tipo de dados presente na linguagem C++ ocupa 8 *bits*. Portanto na prática seriam necessários $8n$ *bits* para armazenar um *array booleano* com n elementos. Neste cenário 7 a cada 8 *bits* seriam desperdiçados, o que representa 87,5% de desperdício de espaço.

Além dos *arrays* de tamanho fixo (ou seja, cujas posições utilizam um número fixo de *bits*), podemos encontrar situações onde os valores armazenados no *array* necessitem quantidades de *bits* diferentes por elemento. Por exemplo, seja $A = [17, 1]$. $A[0] = 17$ precisa de 5 *bits* para ser armazenado enquanto que $A[1] = 1$ precisa de apenas 1 *bit*. Neste cenário, se A utilizar uma quantidade fixa de *bits* para armazenar cada posição, serão necessários no mínimo 5 *bits* (quantidade de *bits* necessária para armazenar o valor máximo de A), causando um desperdício de 4 *bits* para armazenar $A[1]$.

Portanto, seria interessante uma representação de *arrays* cujas posições utilizem exatamente a quantidade de *bits* necessária para armazenar os valores, reduzindo o desperdício de memória resultante.

Nas próximas seções serão discutidas algumas abordagens utilizadas neste trabalho para a representação de *arrays* que buscam trazer uma solução para o desperdício de memória apresentados nos parágrafos anteriores.

3.1.1 Arrays fixos

Para o contexto deste projeto, um *array* fixo k é uma sequência de elementos com um valor pré-determinado k , tal que cada elemento é representado em memória utilizando exatamente k *bits*. Existem *arrays* de tamanho fixo nas linguagens de programação mais populares, por exemplo o *array* do tipo *int* na linguagem C, que representa cada elemento utilizando 32 *bits*. O problema é que nem sempre é desejável utilizar $k = 32$ *bits*. Em alguns casos, pode ser mais interessante utilizar um valor $k < 32$.

Considere o *array* $A = [4, 6, 2]$. Se for utilizado um *array* do tipo *int*, serão gastos 96 *bits* enquanto que apenas 9 seriam o suficiente (apenas $k = 3$ *bits* para cada elemento), totalizando um desperdício de 90%.

Uma proposta apresentada em (NAVARRO, 2016a) introduz uma maneira de armazenar um *array* fixo k , onde $0 < k \leq w$, utilizando *arrays* fixos w tradicionais como base.

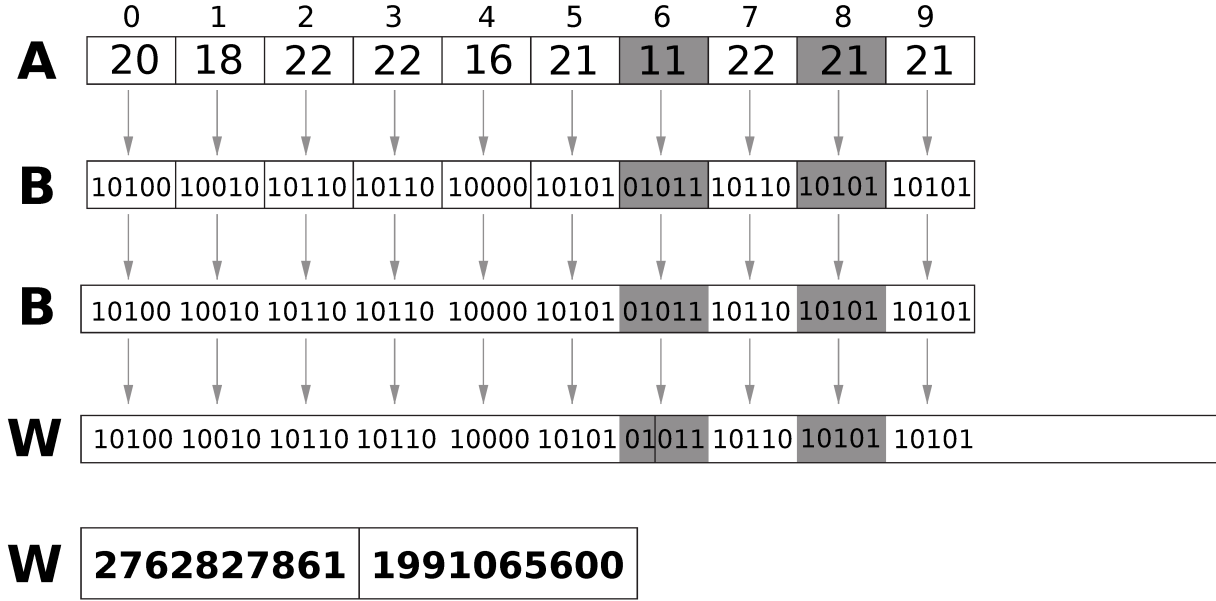


Figura 2 – Exemplo de representação de uma sequência A utilizando *arrays* fixos. O *array* B ilustra a representação lógica de A na forma de uma sequência de *bits* onde são utilizados $k = 5$ *bits* por elemento, que é armazenada utilizando o *array* W , em que cada posição armazena blocos de $w = 32$ *bits* de B . O elemento $A[8]$ ilustra o caso onde os *bits* de um valor $A[i]$ encontram-se em uma única posição de W , enquanto que o elemento $A[6]$ ilustra o cenário onde os *bits* de um valor encontram-se divididos em duas posições adjacentes de W . Imagem baseada na Figura 3.1 do livro *Compact Data Structures - A Practical Approach*([NAVARRO, 2016a](#)).

Considere por exemplo o *array* de dados A com n elementos, que serão representados utilizando k *bits* cada. Podemos representar este *array* como uma sequência B de nk *bits*, onde cada bloco de k *bits* representa um elemento de A e portanto existem n blocos. Dividindo B em blocos de tamanho w , é obtida uma nova sequência W , dessa vez com $\lceil \frac{nk}{w} \rceil$ blocos. Como cada bloco possui w *bits*, pode-se utilizar um *array* fixo w já disponível na linguagem de programação para representar W , onde cada bloco seria uma posição de W . A Figura 2 exemplifica essa ideia.

Seguindo essa abordagem, é possível armazenar os nk *bits* utilizando $w \lceil \frac{nk}{w} \rceil$ *bits* ao todo, reduzindo o desperdício de memória a uma quantidade constante $w - 1$ no pior dos casos.

É preciso definir as operações *read* e *write* para *arrays* fixos considerando elementos de k *bits* em W , que é organizado utilizando w *bits* por elemento. Para isso, utiliza-se manipulações de *bits* para alcançar um desempenho satisfatório em tempo sem aumentar o consumo de memória.

Existem dois casos de funcionamento para cada operação:

1. Os k *bits* desejados encontram-se completamente contidos dentro de uma posição

em W ;

2. Os k bits desejados encontram-se parcialmente contidos em duas posições adjacentes em W . Neste caso, uma quantidade $0 < k' < k$ dos k bits formam um sufixo no bloco da esquerda e o restante dos $k - k'$ bits formam um prefixo no bloco da direita.

Ambos os casos mencionados acima podem ser visualizados na Figura 2 e devem ser considerados durante a implementação.

Para identificar qual caso deverá ser tratado, é preciso primeiro identificar em quais blocos o elemento desejado encontra-se, utilizando-se o fato de que os *bits* em questão estão adjacentes na sequência. Para descobrir em qual bloco b o i -ésimo *bit* da sequência se encontra, utiliza-se a seguinte fórmula: $b = \lfloor i/w \rfloor$.

Além de descobrir o bloco b que um determinado *bit* i se encontra, também é importante descobrir qual a posição i' do *bit* i dentro de $W[b]$. Esse valor pode ser encontrado através da seguinte fórmula: $i' \equiv i \pmod{w}$.

3.1.1.1 Operação *read*

Sejam o par (i, j) os índices do primeiro e último *bit* de $A[x]$ em B , tais que $A[x] = B[i, j]$. Considere $W[bi]$ e $W[bj]$ os blocos em W que contém os *bits* i e j respectivamente, sendo (bi, bj) os índices de tais blocos.

Para $bi = bj$, os k bits de $A[x]$ formam um intervalo de tamanho k na máscara de *bits* do bloco bi . Para descobrir o intervalo relativo dentro da máscara de *bits*, utilizam-se as fórmulas: $i' \equiv i \pmod{w}$ e $j' \equiv j \pmod{w}$, onde (i', j') representam o intervalo de $A[x]$ na máscara de *bits* de $W[bi]$. Para acessar um intervalo (a, b) em uma máscara de *bits* m podemos utilizar o seguinte comando em linguagem C:

```
#define access_mask_interval(m, a, b) ((m >> a) & ((1 << (b-a+1)) - 1))
```

Portanto, para casos em que $bi = bj$, pode-se escrever $read(A, x)$ utilizando a *macro* acima: $read(A, x) = access_mask_interval(W[bi], i', j')$

A operação $access_mask_interval(m, a, b)$ é composta apenas por operações de manipulação de *bits*, que são implementadas em tempo constante pela linguagem de programação. Portanto, a complexidade em tempo da operação $read(A, x)$ quando $bi = bj$ é $O(1)$.

Quando $bi \neq bj$, tem-se o cenário onde o intervalo $[a, b]$ encontra-se entre os dois blocos. Sejam $[a_{esq}, w - 1]$ e $[0, b_{dir}]$ os intervalos em $W[bi]$ e $W[bj]$ que concatenados compõem o intervalo $[a, b]$. Nesse caso, pode-se utilizar a função $read_diff_blocks$ descrita a seguir para realizar a leitura do elemento $A[x]$:

```

int read_diff_blocks(int *W, int x) {
    // calcula-se bi, bj, a_esq, e b_dir em termos de x
    int val_esq = access_mask_interval(W[bi], a_esq, w-1);
    int val_dir = access_mask_interval(W[bj], 0, b_dir);
    int val_esq <= (b_dir + 1);
    return val_esq | val_dir;
}

```

Assim como no caso mais simples, a leitura para blocos diferentes também se resume a um número constante de manipulações de *bit* e portanto possui a mesma complexidade de tempo $O(1)$.

3.1.1.2 Operação *write*

Para executar a operação $write(A, x, val)$, assim como na operação $read(A, x)$ discutida na seção anterior, utilizaremos o *array* W para representar a sequência de *bits* B em blocos de tamanho w .

Para realizar a escrita, utiliza-se uma função semelhante à *macro* $access_mask_interval$, que será utilizada de maneira análoga para fazer a escrita de val na máscara de *bits* m (que neste caso será um bloco no *array* W).

Para realizar a escrita em uma máscara de *bits* utilizamos a operação lógica XOR , representada por $\oplus$. Esta operação é interessante pois possui a seguinte propriedade: $a \oplus a = 0$, que implica em: $a \oplus a \oplus b = b$.

Essa propriedade pode ser utilizada para sobrescrever um sub-intervalo $[a, b]$ de m com o valor val (assumindo-se que val é representado usando no máximo $b - a + 1$ *bits*) da seguinte forma:

1. $m = m \oplus (access_mask_interval(m, a, b) \ll a)$. Isso é equivalente a zerar o intervalo $[a, b]$ em m .
2. $m = m \oplus (val \ll a)$. Isso é equivalente a preencher o intervalo $[a, b]$ na máscara de *bits* m com os *bits* que representam val .

Os passos acima podem ser utilizados para definir a função $write_mask_interval(m, a, b, val)$:

```

// sobrescreve o intervalo [a, b] na máscara de bits m com o valor val
void write_mask_interval(int *m, int a, int b, int val) {
    *m = *m ^ (access_mask_interval(*m, a, b) << a) ^ (val << a);
}

```

A função acima pode ser utilizada para implementar a função $write(A, x, val)$. Assim como descrito para a operação $read(A, x)$, considera-se par (i, j) os primeiro e último *bits* de $A[x]$ em B . Da mesma forma, $W[bi]$ e $W[bj]$ são os blocos em W que contém os *bits* i e j respectivamente.

Assim como em $read(A, x)$, para os casos onde $bi = bj$, podemos definir a função $write(A, x, val)$ como:

```
/*
    executa A[x] = val, onde A está representado com o auxílio do array W, e
    bi == bj
*/
void write(int *W, int x, int val) {
    // calcula-se i' e j' e bi em termos de x
    write_mask_interval(W[bi], i', j', val);
}
```

De maneira análoga, para os casos onde $bi \neq bj$, o intervalo $[a, b]$ pode ser dividido em dois intervalos adjacentes $[a_{esq}, w - 1]$ e $[0, b_{dir}]$, tais que $[a_{esq}, w - 1]$ é um sufixo do bloco da esquerda e $[0, b_{dir}]$ é um prefixo do bloco da direita.

Sejam $d = b_{dir} + 1$, $d_val = val \& ((1 \ll d) - 1)$ e $e_val = (val \gg d)$. Os valores e_val e d_val representam as porções da máscara de bits do valor val que serão incluídas nos intervalos $[a_{esq}, w - 1]$ e $[0, b_{dir}]$ das máscaras de *bits* $W[bi]$ e $W[bj]$, respectivamente. Desta forma, pode-se definir a seguinte função $write$ para os casos onde $bi \neq bj$:

```
/*
    executa A[x] = val,
    onde A está representado com o auxílio do array W e bi != bj
*/
void write(int *W, int x, int val) {
    // calcula-se a_esq, b_dir, bi e bj em termos de x
    int d = b_dir + 1;
    int d_val = val & ((1 << d) - 1); // d_val contém os primeiros d bits de val
    int e_val = val >> d; // e_val contém os demais bits de val
    write_mask_interval(W[bi], a_esq, w-1, e_val);
    write_mask_interval(W[bj], 0, b_dir, d_val);
}
```

Assim como para a operação $read(A, x)$, a operação $write$ se baseia em um número constante de operações de manipulação de *bits* e portanto possui complexidade em tempo de $O(1)$ por operação.

3.1.2 Arrays variados

Na seção anterior foi apresentada uma abordagem bastante eficiente para lidar com *arrays* cujos elementos utilizam uma quantidade fixa de *bits*. Apesar das técnicas discutidas funcionarem bem com *arrays* cujos valores são semelhantes, ainda ocorre muito desperdício quando os dados armazenados possuem distribuição aleatória e variada. Por exemplo, para armazenar os valores $\{2, 1, 16\}$ em um *array* que utiliza células de tamanho fixo, seriam necessários pelo menos 5 *bits* para armazenar o valor 16 e, portanto, seriam utilizados 15 *bits* para armazenar o *array* todo (5 *bits* para cada elemento). Porém, o valor 2 precisa de apenas 2 *bits* para ser representado e o valor 1 utiliza apenas um único *bit*. Portanto, são necessários apenas 8 *bits* para representar os dados, apenas 53% dos 15 *bits* utilizados pela abordagem mencionada.

Pensando em reduzir o uso de memória, (NAVARRO, 2016a) apresenta uma abordagem que permite armazenar sequências de valores arbitrários, reduzindo o desperdício de memória e permitindo realizar acessos com um desempenho satisfatório. Essa abordagem, descrita a seguir, inviabiliza a escrita dos elementos no *array* por se tratar de uma abordagem estática. Entretanto, para o contexto de grafos temporais compactos tratado neste trabalho isso não será um problema.

3.1.2.1 Sampled pointers

Uma primeira abordagem para representar *arrays* variados é a abordagem chamada de *sampled pointers*. Ela consiste em construir os *arrays* B e W da mesma forma que é feito para os *arrays* fixos (Seção 3.1.1).

Para *array* fixos essa abordagem funciona sem complicações pois um determinado valor $A[i]$ é diretamente mapeado para o intervalo $[i_0, i_1] = [ik, (i + 1)k - 1]$ em B . Dessa forma, é possível manipular os *bits* desejados usando as técnicas discutidas na seção anterior.

Porém, para *arrays* cujos elementos utilizam um número variado de *bits*, não é possível saber diretamente qual é o intervalo ocupado por um valor em B . Portanto, é necessária uma abordagem diferente para encontrar o intervalo em que um determinado valor ocupa em B . É exatamente esse o problema que os *sampled pointers* resolvem.

A ideia é dividir o *array* A em blocos lógicos e criar um novo *array* $P[0, \lceil \frac{N}{k} \rceil - 1]$, onde N é a quantidade total de *bits* em B e k é um inteiro que representa o tamanho de cada bloco lógico. O valor $P[x]$ indicará o início do intervalo de $A[x \times k]$ em B . Dessa forma, para descobrir onde um determinado valor $A[i]$ está, basta identificar qual é o bloco $p_i = \lfloor \frac{i}{k} \rfloor$ em que ele se encontra. Após descoberto o bloco p_i , itera-se linearmente sobre os *bits* em B que compõem os $i' \equiv i \pmod{k}$ elementos do bloco lógico que aparecem antes de $A[i]$, para encontrar a posição inicial de $A[i]$ em B .

Para que seja possível iterar nos elementos do bloco, é necessário que os valores estejam codificados utilizando alguma técnica que permita realizar a decodificação por meio da leitura em sequência dos *bits*, como a codificação *Gamma* (γ) (ELIAS, 1975).

Uma vez utilizados os *sampled pointers* para encontrar o intervalo $B[i_0, i_1]$ que representam os *bits* do valor $A[i]$, pode-se aplicar as mesmas técnicas demonstradas na Seção 3.1.1 para realizar a leitura dos *bits* de B armazenados com o auxílio do *array* W .

Perceba que utilizando essa abordagem, caso seja necessário atualizar uma posição utilizando uma quantidade maior de *bits*, toda a estrutura deve ser reconstruída e, por isso, esta abordagem não suporta de maneira eficiente operações do tipo *write*. Por esse motivo a estrutura utilizando *sampled pointers* consiste em uma abordagem estática.

Portanto, operações do tipo $read(A, i)$ podem ser executadas em $O(k \times lmax)$ de tempo, onde $lmax$ é o número máximo de *bits* necessários para armazenar um valor $A[i]$. Em termos de memória, para construir a estrutura dos *sampled pointers*, além dos $w \lceil \frac{N}{w} \rceil$ *bits* necessários para armazenar W , é necessário armazenar $\lceil \frac{N}{k} \rceil$ valores, um para cada bloco, contendo $\lfloor \log N \rfloor + 1$ *bits* representando o índice do início do bloco em B . Para armazenar P , utiliza-se um *array* fixo que utiliza w *bits* por elemento e, portanto, a estrutura utiliza $w \lceil \frac{N}{k} \rceil + w \lceil \frac{N}{w} \rceil$ *bits* para ser armazenada.

3.1.2.2 Dense pointers

A abordagem utilizando *sampled pointers* é interessante pois permite trocar eficiência de tempo por economia de memória, simplesmente variando o tamanho do bloco k . Porém, de maneira geral, é interessante uma abordagem que não envolva iterar entre os elementos do bloco para conseguir acessar um determinado valor $A[i]$.

Para solucionar esse problema, pode-se estender a estrutura dos *sampled pointers*, de modo que além de armazenar os índices do início de cada bloco, também armazena-se o índice de cada elemento $A[i]$, relativo ao bloco no qual ele se encontra.

Formalmente, a ideia é criar um novo *array* $P'[0, n - 1]$, onde $P'[i] = \sum_{j=k \lfloor i/k \rfloor}^{i-1} len(j)$, onde $len(x)$ é o número de *bits* utilizados para representar o valor x na base binária. Portanto, pode-se encontrar o início do intervalo de $A[i]$ em B em tempo constante por meio da fórmula: $indice(i) = P[\lfloor i/k \rfloor] + P'[i]$. De maneira semelhante, pode-se calcular o intervalo $B[i_0, i_1]$ que representa $A[i]$, encontrando $i_0 = indice(i)$ e $i_1 = indice(i + 1) - 1$.

Dessa forma, é possível realizar a leitura de $A[i]$ em W utilizando as mesmas abordagens descritas na Seção 3.1.1 em tempo constante. Para isso, utiliza-se os *dense pointers* para encontrar o intervalo em B ocupado por $A[i]$, e os demais passos para fazer a leitura são os mesmos que foram utilizados para os *arrays* fixos.

Em termos de memória, para armazenar P' são necessários n valores, porém

cada valor representa um índice relativo ao bloco ao qual se encontra e por isso utiliza $\lfloor \log(k \times lmax) \rfloor + 1$ *bits* para ser armazenado. Portanto, ao todo são utilizados $w \left\lceil \frac{N}{k} \right\rceil + w \left\lceil \frac{N}{w} \right\rceil + n \log(k \times lmax)$ *bits* para armazenar os *sampled* e *dense pointers*. O valor escolhido para k no trabalho foi $k = w \times \ln 2$ e portanto a complexidade final de memória para armazenamento da estrutura ficou em $O(N + n \log(w \times lmax \times \ln 2)) = O(n \log w)$.

3.2 Bitvector

Na Seção 3.1.1 foi discutida uma forma de armazenar *arrays* utilizando um número fixo de *bits* para representar cada elemento. Essa abordagem se mostra muito eficiente para manusear *arrays* de *bits*, pois permite armazenar sequências de n *bits* com desperdício de memória constante. Porém, além de armazenar os dados, é comum que seja necessário extrair informações extras sobre a sequência de *bits* $B[0, n - 1]$, por exemplo:

1. Quantos *bits* de valor 1 existem no prefixo $B[0, i]$?
2. Qual a posição do i -ésimo *bit* de valor 1?
3. Quantos *bits* de valor 1 existem no *array*?

Essas operações podem ser respondidas de maneira trivial por meio de uma busca linear no *array*, veremos a seguir que é possível respondê-las em tempo constante, mantendo o uso de memória na ordem de $o(n)$ *bits*.

A estrutura que nos permite responder tais perguntas para uma sequência de *bits* B é chamada de *bitvector* e busca suportar as seguintes operações (que podem ser combinadas para responder as 3 perguntas acima):

- $access(B, i)$: retorna o valor do *bit* armazenado na posição i .
- $rank(B, i, x)$: retorna quantos *bits* iguais a x existem no intervalo $[0, i)$, com $0 \leq i < n$ e $x \in \{0, 1\}$. Quando omitido, assume-se $x = 1$.
- $select(B, i, x)$: retorna o índice do i -ésimo *bit* igual a x , com $0 \leq i < n$ e $x \in \{0, 1\}$. Também pode ser definido como a primeira posição j , tal que $rank(B, j, x) = i$. Para os casos onde i é maior ou igual ao número de *bits* iguais a x em B , retorna-se o valor n . Quando omitido, assume-se $x = 1$.

A operação *access* é suportada pelo *array* de *bits* e, por isso, o foco dessa seção será implementar as outras duas funcionalidades: *rank* e *select*.

3.2.1 Operação *rank*

Uma solução trivial para o problema do *rank* é realizar uma busca linear no *array* mantendo um contador dos *bits* de interesse até a posição desejada. Tal abordagem é simples, porém possui complexidade em tempo na ordem de $O(n)$, o que para a maioria dos casos é muito lenta.

Outra solução seria criar um *array* S de soma de prefixos, onde $S[0] = 0$ e $S[i] = \sum_{k=0}^{i-1} B[k]$, $\forall i > 0$. Com esse *array* criado, a operação $\text{rank}(B, i)$ é equivalente ao valor armazenado em $S[i]$. A construção desse *array* é feita em tempo linear, porém cada consulta é respondida em $O(1)$. O problema dessa abordagem é que a estrutura de soma acumulada utiliza $n \lfloor \log n \rfloor + n$ *bits* para ser armazenada (n valores que utilizam $\lfloor \log n \rfloor + 1$ *bits* para serem representados), o que é maior que a quantidade de memória utilizada pelos próprios dados.

Uma abordagem proposta por Jacobson ([JACOBSON, 1988](#)) utiliza ambas as abordagens propostas acima para solucionar o problema utilizando $o(n)$ de memória adicional, alcançando um desempenho de $O(1)$ de tempo por operação.

A ideia é dividir o *array* em blocos *grandes* de tamanho $\log^2 n$. O propósito com essa divisão é que, ao invés de pré-calcular o *rank* para cada posição utilizando soma de prefixos conforme proposto acima, pode-se pré-calcular o *rank* para cada **início** de bloco grande. Dessa forma, para calcular $\text{rank}(B, i)$, basta encontrar o *rank* relativo ao início do bloco que contém o elemento i e somar com o valor do *rank* no início do bloco, pré-calculado.

Essa abordagem utiliza $\log n$ *bits* para armazenar a resposta pré-calculada para cada um dos $\left\lceil \frac{n}{\log^2 n} \right\rceil$ blocos grandes existentes, totalizando $\frac{n \log n}{\log^2 n} = \frac{n}{\log n} = o(n)$. Porém, para calcular o *rank* dentro de um bloco grande é necessário realizar uma busca linear, gastando $O(\log^2 n)$ de tempo.

Entretanto, pode-se realizar a mesma técnica de divisão de blocos para calcular o *rank* dentro de cada bloco grande. A ideia é dividir o *array* também em blocos pequenos de $\log n$ *bits*. Esses blocos pequenos estarão contidos em um bloco grande e armazenarão a soma de prefixos dentro do bloco grande, até o início do bloco pequeno correspondente. Dessa forma, para calcular $\text{rank}(B, i)$, basta somar:

1. o valor do *rank* pré-calculado para o início do bloco grande,
2. o valor do *rank* pré-calculado do início do bloco grande, até o início do bloco pequeno,
3. o valor do *rank* do prefixo i dentro do bloco pequeno.

Dessa forma, é atingida uma complexidade de $O(\log n)$ de tempo para iterar linearmente nas posições do bloco pequeno. Porém, nos cenários reais $\log n$ normalmente é

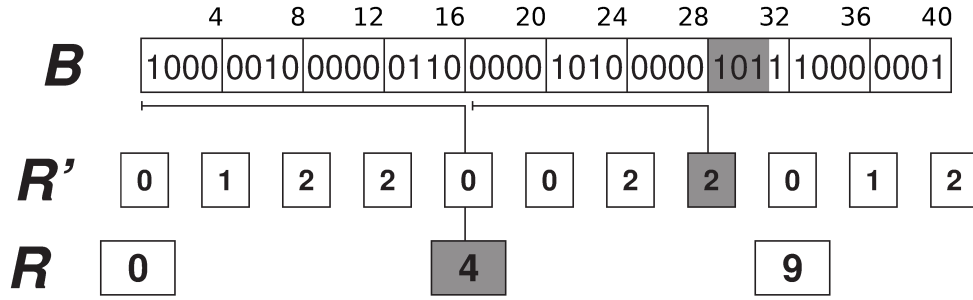


Figura 3 – Exemplo da representação da sequência de bits B utilizando a divisão proposta por Jacobson, utilizando blocos grandes e pequenos para pré-calculer o *rank* para o início de cada bloco. O *array R* armazena os valores pré-computados para o *rank* ao início de cada bloco grande, enquanto que o *array R'* armazena os valores do *rank* para cada posição de B , relativos ao início do bloco grande correspondente. Imagem baseada na Figura 4.3 do livro *Compact Data Structures - A Practical Approach* (NAVARRO, 2016a).

menor que w e portanto esse cálculo do *rank* dentro do bloco pequeno pode ser feito com manipulações de *bits* e utilizando ferramentas que nos permitem fazer a contagem de *bits* em uma palavra do sistema em $O(1)$, como a função `__builtin_popcount()` implementada pelo compilador *GNU/GCC* de linguagem C/C++, alcançando-se um desempenho em tempo de $O(1)$ por consulta.

Em relação à memória, como ainda são utilizados blocos grandes, inicialmente a estrutura consome $o(n)$ *bits* de espaço conforme analisado anteriormente. Para os blocos pequenos, é preciso armazenar apenas a soma dentro do bloco grande, portanto utilizando $O(\log(\log^2 n)) = O(\log \log n)$ *bits* para pré-calculer o *rank* de cada um dos $\left\lceil \frac{n}{\log n} \right\rceil$ blocos pequenos, totalizando um gasto de memória adicional de $O\left(\frac{n \log \log n}{\log n}\right) = o(n)$.

Portanto, utilizando a divisão em blocos de dois níveis descrita acima consegue-se implementar uma estrutura que suporta operações $\text{rank}(B, i)$ em $O(1)$ de tempo, utilizando $o(n)$ *bits* de memória adicional.

3.2.2 Operação *select*

Assim como a operação *rank*, é possível implementar a operação $\text{select}(B, i)$ de diversas maneiras. A mais trivial de todas é fazer uma busca linear no *array* para encontrar o i -ésimo *bit* desejado. Porém, essa abordagem teria complexidade de tempo linear, o que na maioria dos casos é mais lento que o desejado.

Outra abordagem seria pré-calculer a resposta para o $\text{select}(B, i)$ em uma tabela para cada *bit* 1 presente no *array*¹. O cálculo seria feito em tempo linear, porém cada consulta seria respondida em tempo constante, uma vez que os dados estariam pré-calculados. Porém, no pior caso, quando o *array* fosse composto majoritariamente de *bits* de valor 1,

¹ Aqui assume-se que o *bit* desejado possui valor 1, mas o mesmo pode ser abstraído para o *bit* 0.

seriam necessárias n posições na tabela, cada uma utilizando $\lceil \log n \rceil$ *bits*, o que resultaria em um consumo de memória de $n \lceil \log n \rceil$ *bits*, mais que o próprio *array*.

Entretanto, para *arrays* esparsos, ou seja, com poucos *bits* iguais a 1, seriam necessárias menos entradas na tabela e, portanto, menos memória seria utilizada para armazenar todos os possíveis pré-cálculos. De forma geral, se o *array* possuir m *bits* de valor 1, são necessários $m \lceil \log n \rceil$ *bits* para pré-calculas as respostas. Portanto, para casos onde m é pequeno, essa abordagem pode ser interessante.

No caso do *array* ser denso, isto é, com muitos *bits* iguais a 1, uma possível solução é usar busca binária combinada com *rank* para encontrar a última posição j , tal que $\text{rank}(B, j) = i$. Dessa forma, $\text{select}(B, i) = j$ e o cálculo é feito utilizando $o(n)$ de memória adicional, necessária para a implementação do *rank*, com uma complexidade de $O(\log n)$ de tempo para a execução da busca binária.

Uma abordagem que busca aproveitar os benefícios das duas anteriores foi proposta por Clark (CLARK, 1997) e baseia-se em dividir o *array* em superblocos, de forma que cada superbloco contenha s *bits* iguais a 1. Como os *bits* estão distribuídos aleatoriamente pelo *array*, esses superblocos terão tamanhos diferentes e portanto alguns serão densos, enquanto que outros serão esparsos. Para os superblocos esparsos são pré-calculadas as respostas, enquanto que os superblocos densos são tratados utilizando busca binária ou busca linear (caso o tamanho do superbloco seja bem pequeno).

Considere $s = \log^2 n$ e que um superbloco esparsos é aquele com pelo menos $s^2 = \log^4 n$ *bits*. De maneira análoga, um superbloco é denso se possuir menos de $s^2 = \log^4 n$ elementos. Para cada superbloco b , será calculado um valor $S[b]$ indicando o início de b na sequência de *bits* B .

No pior caso, existirão $\frac{n}{\log^4 n}$ superblocos esparsos, cada um utilizando $\log n$ *bits* para armazenar a resposta dos s possíveis *bits* iguais a 1 dentro do superbloco, totalizando um consumo de memória de $O\left(\frac{n \times \log n \times \log^2 n}{\log^4 n}\right) = O\left(\frac{n}{\log n}\right) = o(n)$ *bits*. Nos casos em que a resposta se encontra em um superbloco denso, é necessário fazer uma busca binária dentro do mesmo, com um desempenho de $O(\log \log^4 n) = O(\log \log n)$ por consulta.

Para diminuir o tempo da consulta, pode-se aplicar a mesma divisão de blocos novamente para criar sub-blocos, que serão utilizados para responder as consultas em superblocos densos.

Considere um sub-bloco sendo um bloco que contém $\sqrt{\log n}$ *bits* de valor 1 e está contido dentro de um superbloco denso. Um sub-bloco é chamado de esparsos se ele contiver pelo menos $\frac{\log n}{2}$ *bits* e denso caso contrário.

Para pré-calculas as respostas para os sub-blocos esparsos, será criado um *array* S' que irá armazenar o índice em B para cada um dos $\sqrt{\log n}$ *bits* iguais a 1 do sub-bloco, relativos ao superbloco denso ao qual ele pertence. Existem no máximo $\left\lceil \frac{2n}{\log n} \right\rceil$

sub-blocos esparsos. Para cada um dos $\sqrt{\log n}$ *bits* iguais a 1 pertencentes a um sub-bloco esparsos serão utilizados $\log \log^4 n = 4 \log \log n$ *bits* para armazenar o índice relativo ao superbloco. Portanto, serão gastos $O\left(\left\lceil \frac{2n}{\log n} \right\rceil \times \sqrt{\log n} \log \log n\right) = O\left(\frac{n \log \log n}{\sqrt{\log n}}\right) = o(n)$ *bits* de memória para pré-calcular as respostas para os *bits* que se encontram dentro de um sub-bloco esparsos.

Para calcular a resposta para os sub-blocos densos, armazena-se para cada bloco o índice de início em B relativo ao superbloco denso ao qual ele se encontra. Cada sub-bloco denso possui pelo menos $\sqrt{\log n}$ *bits*, logo existem $\left\lceil \frac{n}{\sqrt{\log n}} \right\rceil$ sub-blocos densos e cada índice relativo é representado utilizando $O(\log \log^4 n)$ *bits*. Dessa forma, é possível pré-calcular o índice do início de cada sub-bloco utilizando $O\left(\frac{n \log \log n}{\sqrt{\log n}}\right) = o(n)$ *bits*. Para calcular a resposta dentro de cada sub-bloco denso, pode-se pré-calcular todas as possíveis $2^{\frac{\log n}{2}}$ respostas em uma tabela. Uma segunda abordagem seria percorrer os *bits* iguais a 1 da máscara de *bits* do sub-bloco denso utilizando técnicas de manipulação de *bits*. A abordagem escolhida neste trabalho utiliza a segunda abordagem, com complexidade $O(x) = O(w)$, onde x é o número de *bits* iguais a 1 existentes do início do sub-bloco denso até o resultado de $\text{select}(B, i)$.

Portanto, ao todo são gastos $o(n)$ *bits* para pré-calcular as respostas para consultas em superblocos esparsos. Para os superblocos densos também são utilizados $o(n)$ *bits* para pré-calcular as respostas para sub-blocos esparsos, assim como para armazenar o início de cada sub-bloco denso. Dessa forma, são gastos $o(n)$ *bits* de memória no total, alcançando $O(1)$ de tempo por consulta, com exceção das consultas pertencentes a sub-blocos densos, que possuem complexidade $O(w)$. Na prática, consultas em sub-blocos densos são eficientes devido ao uso de máscaras de *bits*, portanto considera-se uma complexidade $O(1)$ amortizada para consultas do tipo *select*.

3.3 Wavelet tree

Conforme apresentado na Seção 3.2, os *bitvectors* são úteis quando é necessário manipular sequências de *bits*. Porém, nem sempre as sequências a serem manipuladas são compostas de valores binários. Uma extensão dos *bitvectors* seria realizar as operações *rank* e *select* em uma sequência cujos elementos pertencem a um intervalo $[0, \sigma)$. Uma das estruturas que permite fazer isso é chamada de *wavelet tree* (GROSSI; GUPTA; VITTER, 2003), descrita a seguir.

Seja um alfabeto Σ , que representa o intervalo $[0, \sigma)$. A *wavelet tree* permite realizar as seguintes consultas sobre elementos de uma sequência $A[0, n - 1]$ onde $A[i] \in \Sigma$:

- $\text{access}(A, i)$: retorna $A[i]$;

- $rank(A, i, c)$: retorna quantos índices j existem no intervalo $A[0, i - 1]$ tais que $A[j] = c$, para $c \in \Sigma$ e $j < i$.
- $select(A, i, c)$: retorna qual o maior índice j tal que $rank(A, j, c) = i$. Em outras palavras, retorna qual a posição do i -ésimo valor c na sequência, onde $c \in \Sigma$ e $0 \leq i < n$. Se o número de valores iguais a c em A for menor ou igual a i , retorna-se n .

Uma solução trivial para responder tais consultas seria construir um *array* de soma de prefixos para cada símbolo $c \in \Sigma$. Tal abordagem iria resultar em uma matriz M de dimensões $\sigma \times n$, onde $M[c][i] = rank(A, i, c)$ para cada par de valores (c, i) possíveis.

Utilizando tal abordagem, seria possível responder consultas do tipo $rank(A, i, c)$ em $O(1)$ pois estariam pré-calculadas e consultas do tipo $select(A, i, c)$ em $O(\log n)$, utilizando busca binária para encontrar o maior j tal que $rank(A, j, c) \geq i$. Como cada valor armazenado em M é menor ou igual a n , seriam utilizados $O(\sigma n \log n)$ *bits* para armazenar todas as possíveis respostas para o $rank(A, i, c)$.

A *wavelet tree* busca reduzir o consumo de memória a $O(w\sigma + n \log \sigma)$ *bits*, com complexidade $O(\log \sigma)$ de tempo por consulta.

3.3.1 Construção

A ideia da *wavelet tree* baseia-se em construir uma árvore binária sobre o alfabeto Σ , onde cada nó da árvore representa uma partição de elementos do *array* A , assim como um sub-intervalo de valores de Σ , de forma que todos os elementos $A[i]$ representados por um nó específico estão contidos no sub-intervalo do alfabeto representado pelo mesmo.

Seja X a raiz da árvore, que portanto representa o *array* A inteiro e o intervalo $[0, \sigma)$. Sejam também X_{esq} e X_{dir} os filhos da esquerda e direita de X . Os elementos representados por X são repassados aos filhos de forma que X_{esq} receberá os valores $0 \leq A[i] \leq \lfloor \frac{\sigma}{2} \rfloor$ e X_{dir} receberá os valores $\lfloor \frac{\sigma}{2} \rfloor < A[i] < \sigma$. Se um nó representa um intervalo de valores $[c, c]$, para $c \in \Sigma$, esse nó é uma folha na árvore e, portanto, não possui filhos.

Para implementar a operação $access(A, i)$, é necessário percorrer o caminho percorrido pela posição i da sequência até uma folha. Sejam X_f a folha que contém o elemento $A[i]$ e $[c, c]$ o intervalo do alfabeto representado por X_f , como o intervalo possui um único valor possível, conclui-se que $A[i] = c$.

Para percorrer o caminho de um índice i na árvore até um nó-folha, considere por exemplo a sequência de valores A' representada por X_{esq} . É necessário saber qual índice j em A' um determinado valor $A[i]$ encontra-se. Para conseguir essa informação, é mantido um *bitvector* B em cada nó X' indicando para cada elemento $A[i]$ representado

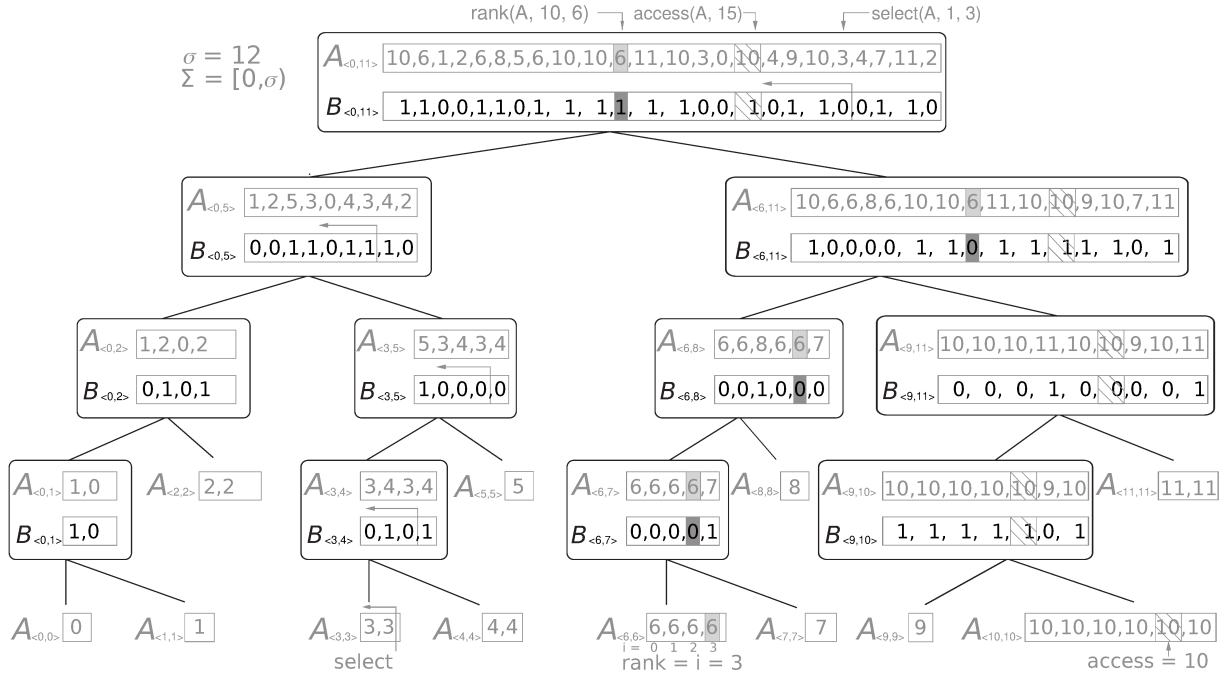


Figura 4 – Exemplo de construção da estrutura *wavelet tree* para uma sequência A com $\sigma = 12$ e $\Sigma = [0, 11]$, tal que cada nó representa um sub-intervalo $[a, b]$ do alfabeto Σ , indicado pela notação $A_{\langle a, b \rangle}$ e $B_{\langle a, b \rangle}$. Em cinza estão as sequências representadas logicamente por cada nó da árvore e, em preto, os elementos que são armazenados por cada nó como parte da construção da estrutura, são eles: *bitvectors*, intervalos do alfabeto e ponteiros para os filhos. Também é possível visualizar os nós e índices percorridos da raiz até as folhas para o cálculo das operações $\text{rank}(A, 10, 6)$ e $\text{access}(A, 15)$, assim como para a operação $\text{select}(A, 1, 3)$, cujo cálculo começa na folha e sobe em direção à raiz. Imagem baseada na Figura 6.4 do livro *Compact Data Structures - A Practical Approach* (NAVARRO, 2016a).

por X' , se o mesmo foi repassado para a esquerda ou para a direita durante a construção. Formalmente, $B[i] = 0$ se $A[i]$ foi repassado para a esquerda, ou $B[i] = 1$ se $A[i]$ foi repassado para a direita.

Dessa forma, se um determinado elemento $A[i]$ for tal que $A[i] \leq \left\lfloor \frac{\sigma}{2} \right\rfloor$, o mesmo será representado como o elemento $j = \text{rank}(B, i, 0)$ em X_{esq} . Caso contrário, $A[i]$ será representado como o elemento $j = \text{rank}(B, i, 1)$ em X_{dir} . O processo acima é repetido recursivamente até que o subconjunto de A representado pelo nó seja vazio ou até que o nó X' seja uma folha, conforme exemplificado pelo Algoritmo 1. A Figura 4 exemplifica uma *wavelet tree*.

3.3.1.1 Análise

Como cada nó irá repassar os elementos aos quais é responsável para os filhos, cada nível da *wavelet tree* será composto de n elementos. Consequentemente, a soma de todos os *bits* necessários para um determinado nível da árvore é igual a n . Como descrito

Algoritmo 1: $\text{access}(A, i)$ utilizando *wavelet tree*

```

início
   $X = \text{raiz da wavelet tree};$ 
   $X.L$  e  $X.R$  compõem o intervalo em  $\Sigma$  representado por  $X$ ;
  enquanto  $X.L \neq X.R$  faça
     $B = \text{bitvector}$  representando os elementos do nó  $X$ ;
     $b = \text{access}(B, i);$ 
     $i = \text{rank}(B, i, b);$ 
    se  $b = 0$  então
       $X = X_{\text{esq}};$ 
    fim
    senão
       $X = X_{\text{dir}};$ 
    fim
  fim
  retorna  $X.L;$ 
fim

```

na Seção 3.2, é possível implementar *bitvectors* utilizando $n + o(n)$ *bits* de memória. Logo, cada nível da *wavelet tree* também utilizará $n + o(n)$ *bits* para armazenar os *bitvectors* necessários. Além disso, serão armazenados dois inteiros em cada nó, relativos ao intervalo representado pelo mesmo, assim como dois ponteiros para os seus respectivos filhos, totalizando $4w$ *bits* por nó.

Em cada nível o intervalo do alfabeto representado por um nó é dividido pela metade, logo podem existir no máximo $\lceil \log \sigma \rceil$ níveis abaixo da raiz. Portanto, ao todo, os *bitvectors* utilizarão $n \lceil \log \sigma \rceil + o(n \log \sigma)$ *bits* para serem armazenados e, em geral, σ é constante.

Com a quantidade de níveis da árvore, pode-se calcular a quantidade de nós da mesma. No caso da árvore ser completa, cada nível k terá 2^k nós, de maneira que o número total de nós é menor ou igual a $\sum_{k=0}^{\lceil \log \sigma \rceil} 2^k = 2^{\lceil \log \sigma \rceil + 1} - 1 \leq 2 \times 2^{\lceil \log \sigma \rceil} - 1 \leq 4\sigma$. Para cada nó são armazenados $4w$ *bits* e, portanto, os nós da árvore utilizam no máximo $16w\sigma$ *bits* para armazenar as informações referentes ao alfabeto representado e os filhos correspondentes.

Portanto, somando-se o espaço ocupado pelos nós individualmente e pelos *bitvectors* totaliza-se um consumo de $n \lceil \log \sigma \rceil + o(n \log \sigma) + 16w\sigma$ *bits* para armazenar a *wavelet tree*.

3.3.2 Operação *rank*

Seguindo o Algoritmo 1, chega-se ao nó-folha da árvore que contém o elemento $A[i]$, logo representa o intervalo $\sigma[A[i], A[i]]$. Um fato interessante é que se $A[i]$ é representado

pelo índice $0 \leq i'$ no nó-folha, então existem i' elementos $A[j], j < i$ tais que $A[j] = A[i]$. Portanto, $\text{rank}(A, i, A[i]) = i'$.

Seguindo a mesma ideia, pode-se modificar o Algoritmo 1 para que ao final dele o nó-folha resultante represente o intervalo $[c, c]$ em Σ e, portanto, torna-se possível saber quantos elementos $A[j], j < i$ existem tais que $A[j] = c$, que é exatamente a resposta para a operação $\text{rank}(A, i, c)$. O Algoritmo 2 representa essa ideia.

Algoritmo 2: $\text{rank}(A, i, c)$ utilizando *wavelet tree*

```

início
   $X = \text{raiz da wavelet tree};$ 
   $X.L$  e  $X.R$  compõem o intervalo em  $\Sigma$  representado por  $X$ ;
  enquanto  $X.L \neq X.R$  faça
     $B = \text{bitvector}$  representando os elementos do nó  $X$ ;
    se  $c \leq \lfloor \frac{X.L+X.R}{2} \rfloor$  então
       $X = X_{\text{esq}};$ 
       $i = \text{rank}(B, i, 0);$ 
    fim
    senão
       $X = X_{\text{dir}};$ 
       $i = \text{rank}(B, i, 1);$ 
    fim
  fim
  retorna  $i$ ;
fim

```

3.3.2.1 Análise

Como a cada passo do algoritmo percorre-se 1 nível na árvore, são realizadas $\lceil \log \sigma \rceil$ iterações para calcular o rank . A cada iteração é necessário realizar consultas do tipo rank em um *bitvector*, que podem executadas em $O(1)$ conforme demonstrado na Seção 3.2.1. Portanto, a complexidade final da operação rank é de $O(\log \sigma)$ em tempo e $O(1)$ de memória uma vez que a *wavelet tree* já está construída, pois não é necessário criar nenhuma estrutura de dados adicional para o cálculo da operação.

3.3.3 Operação *select*

Assim como na operação rank , a operação *select* também é calculada por meio de uma busca na *wavelet tree*. Porém, a busca terá início em um nó-folha da árvore e irá subir até a raiz em busca de calcular a resposta esperada.

O objetivo é responder a consulta $\text{select}(A, i, c)$, que representa qual o índice do i -ésimo elemento de valor c em A . Seja X o nó-folha que representa o valor c do alfabeto

e X_{pai} o ancestral direto do nó X na *wavelet tree*, o cálculo do *select* pode ser feito com o Algoritmo 3.

Algoritmo 3: *select*(A, i, c) utilizando *wavelet tree*

```

início
   $X$  = Nó-folha que contém elementos de valor  $c$ ;
   $X.L$  e  $X.R$  compõem o intervalo em  $\Sigma$  representado por  $X$ ;
   $X_{pai}$  representa o ancestral direto de  $X$  na árvore;
  enquanto  $X$  não é raiz faça
     $X = X_{pai}$ ;
     $B$  = bitvector representando os elementos do nó  $X$ ;
    se  $c \leq \lfloor \frac{X.L+X.R}{2} \rfloor$  então
       $i = select(B, i, 0)$ ;
    fim
    senão
       $i = select(B, i, 1)$ ;
    fim
  fim
  retorna  $i$ ;
fim

```

3.3.3.1 Análise

Para executar o Algoritmo 3, é necessário primeiramente fazer uma busca da raiz até a folha, semelhante ao que é feito pelo Algoritmo 1. Como foi demonstrado, essa operação pode ser executada em tempo $O(\log \sigma)$, utilizando $O(1)$ *bits* de memória adicional.

Uma vez que foi encontrado o nó-folha, é preciso executar operações *select* nos *bitvectors* dos nós presentes no caminho da folha até a raiz. Para que não sejam armazenados ponteiros para os ancestrais de cada nó na árvore, armazena-se ponteiros para os nós presentes no caminho até a raiz, para que seja possível efetuar o cálculo de X_{pai} para os nós presentes no caminho. Outra abordagem, utilizada neste trabalho, é implementar o algoritmo 3 de maneira recursiva, armazenando os nós do caminho até a folha na pilha de recursão. Entretanto, ambas as abordagens utilizam $O(w \log \sigma)$ *bits* de memória adicional.

Uma vez construída a *wavelet tree* com *bitvectors* que utilizam a abordagem proposta por Clark demonstrada na Seção 3.2.2 para calcular a operação *select*, cada operação *select* nos *bitvectors* do caminho até a raiz pode ser executada em tempo $O(1)$ utilizando $O(1)$ *bits* adicionais ao espaço utilizado pela *wavelet tree*. Como existem $\lceil \log \sigma \rceil$ nós no caminho da folha até a raiz, no total a operação *select*(A, i, c) na *wavelet tree* possui complexidade $O(\log \sigma)$ de tempo e $O(w \log \sigma)$ *bits* de memória adicional.

4 Grafos temporais compactos

Um grafo temporal consiste em um conjunto V de vértices, numerados no intervalo $[0, |V| - 1]$ e um conjunto de E arestas $e_i = (u, v, t_{ini}, t_{fim})$, onde u, v são vértices do grafo, que estiveram conectados no intervalo de tempo $[t_{ini}, t_{fim}]$, para $0 \leq t_{ini} \leq t_{fim} \leq T$, para $t_{ini}, t_{fim}, T \in \mathbb{N}$, sendo T o valor máximo para os instantes de tempo. Para cada par de arestas $e_1 = (u, v, t_0, t_1)$ e $e_2 = (u, v, t_2, t_3)$ com $t_1 \leq t_2$, é considerado que os intervalos $[t_0, t_1]$ e $[t_2, t_3]$ não possuem interseção, a não ser nos instantes t_1 e t_2 .

Considera-se uma representação lógica para os grafos temporais utilizando listas de adjacência de eventos, como apresentado na Figura 5. Define-se um evento na lista de adjacência de um determinado vértice u o par (v, t) , que indica um instante no tempo simbolizando o início ou o fim da aresta (u, v) . A sequência $events[u]$ é composta pelos eventos que representam a lista de adjacência de um vértice u , para $u \in V$, enquanto que a ordem de um evento (v, t) em $events[u]$ indicará se o mesmo representa o início ou o fim da aresta (u, v) .

Neste trabalho, as estruturas de dados que representam os grafos temporais implementam as três operações básicas apresentadas na Tabela 2: *has_edge*, *neighbours* e *aggregate*, sendo as duas primeiras implementadas de maneira específica para cada estrutura de dados e a última sendo uma implementação comum a todas as estruturas analisadas.

$has_edge(u, v, t_{ini}, t_{fim})$	Retornar um valor <i>booleano</i> indicando se existe uma aresta do vértice u para o vértice v em algum instante do intervalo de tempo $[t_{ini}, t_{fim}]$.
$neighbours(u, t_{ini}, t_{fim})$	Para um determinado vértice u , retornar todos os vértices v tais que exista uma aresta de u para v em algum instante do intervalo de tempo $[t_{ini}, t_{fim}]$.
$aggregate(t_{ini}, t_{fim})$	Retornar todas as arestas do grafo que estão ativas em algum instante do intervalo de tempo $[t_{ini}, t_{fim}]$.

Tabela 2 – Operações básicas de grafos temporais que foram utilizadas para avaliar o desempenho de cada uma das abordagens analisadas neste trabalho.

Para a implementação da operação $aggregate(t_{ini}, t_{fim})$, constrói-se um *array* de pares (u', v') , composto pela união das sequências N_u , definidas como:

$$N_u = \{(u, v), \forall v \in neighbours(u, t_{ini}, t_{fim})\}, \forall u \in V. \quad (4.1)$$

Nas seções a seguir, descrevem-se as estruturas e correspondentes algoritmos utilizados para implementar as operações da Tabela 2.

Para cada operação é feita uma análise do consumo de tempo e memória, utilizando-se estruturas tradicionais da biblioteca padrão da linguagem C++ para computar os dados temporários necessários para cada uma das operações, por exemplo, listas de vértices e tabelas *hash* auxiliares. Portanto, a análise de memória do armazenamento de cada estrutura foi feita de maneira aprofundada, enquanto que o foco da análise de cada operação mencionada na Tabela 2 foi em desempenho de tempo.

4.1 Estrutura de dados *Adjacency Log of Events (EveLog)*

A estrutura *Adjacency Log of Events (EveLog)* (CARO; RODRÍGUEZ; BRISABOA, 2015) baseia-se em armazenar as arestas do grafo de forma semelhante a uma lista de adjacência tradicional. Para cada vértice $u \in V$, o que determina se um evento $e \in events[u]$ é o início ou a conclusão de uma conexão é a paridade do mesmo na sequência, onde aqueles que possuem paridade ímpar indicam o início, caso contrário representam o término de uma aresta.

Por exemplo, considere $u = 0$ e $events[0] = [(1, 1), (2, 1), (1, 4), (2, 5)]$ (veja Figura 5). O primeiro evento $(1, 1)$ representa o início de uma conexão entre os vértices $u = 0$ e $v = 1$ no tempo $t = 1$. Como o terceiro evento é referente ao vértice $v = 1$ novamente, considera-se que o mesmo representa o fim da conexão entre os vértices u e v no tempo $t' = 4$. Portanto, ambos os eventos juntos representam uma aresta (u, v) que ficou ativa no intervalo de tempo $[1, 4]$. De forma análoga, a lista também inclui uma aresta entre os vértices $u = 0$, $v' = 2$, no intervalo de tempo $[1, 5]$.

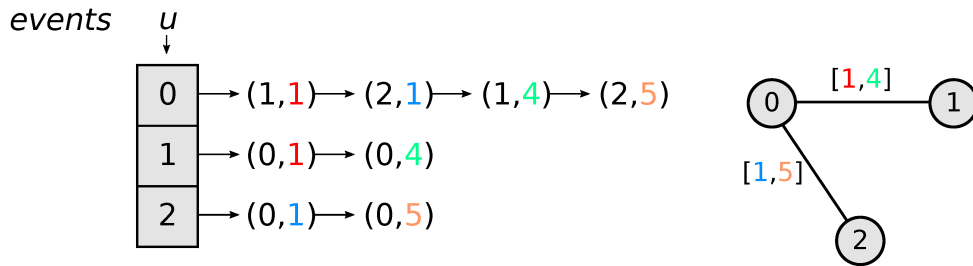


Figura 5 – Exemplo da organização lógica da lista de eventos da estrutura EveLog para um grafo bidirecional, ilustrado na direita. Em colorido encontram-se os instantes no tempo no qual cada evento ocorreu, onde eventos com a mesma cor na lista de eventos representam a mesma aresta no grafo. Por exemplo, em $events[0]$ encontra-se o evento $(2, 1)$ que equivale ao evento bidirecional $(0, 1)$ de mesma cor em $events[2]$.

4.1.1 Construção

Para o armazenamento de cada lista de eventos $events[u]$, são utilizados dois *arrays* variados com as técnicas abordadas em 3.1.2, denominados $labels[u]$ e $timestamps[u]$, onde $labels[u]$ armazena os vértices v que fazem parte dos eventos de u e $timestamps[u]$ armazena os instantes em que cada evento ocorreu. Portanto, se $e_i = (v, t)$, $e_i \in events[u]$, então $labels[u][i] = v$ e $timestamps[u][i] = t$. É importante notar também que os eventos são ordenados em ordem crescente por instante de tempo.

Visando diminuir a variabilidade dos dados e diminuir o consumo de memória, utiliza-se a técnica de compressão *delta gap* nos valores que formarão o *array* $timestamps[u]$. De maneira simplificada, para uma determinada sequência de inteiros $S[0, V - 1]$, a codificação *delta gap* baseia-se em criar uma outra sequência $S'[0, V - 1]$, de forma que $S'[0] = S[0]$ e $S'[i] = S[i] - S[i - 1]$, $\forall i > 0$.

4.1.1.1 Análise

Seja m o número de elementos em $events[u]$, $\forall u \in V$. Armazena-se ambos os *arrays* $labels[u]$ e $timestamps[u]$ utilizando *arrays* variados (veja 3.1.2) com *dense pointers*, utilizando $O(m \log w)$ *bits* de memória. Em termos de tempo, é aplicada a codificação *Delta Gap* nos tempos dos eventos para construir $timestamps[u]$ iterando-se nos elementos com complexidade $O(m)$.

Como existem $|E|$ arestas, tem-se que $\sum_{u \in V} m_u = |E|$ e portanto $\sum_{u \in V} m_u \log w = |E| \log w$. Dessa forma, pode-se armazenar a estrutura *EveLog* utilizando $O(|E| \log w)$ *bits*, com tempo de construção de $O(|E|)$.

4.1.2 Operação *has_edge*

Para executar a operação $has_edge(u, v, t_{ini}, t_{fim})$, calcula-se os seguintes valores:

1. i , que é o índice do primeiro evento $e = (v, t) \in events[u]$ que ocorreu após o início do intervalo $[t_{ini}, t_{fim}]$, tal que $t \geq t_{ini}$,
2. j , definido como o primeiro evento $e = (v, t) \in events[u]$ que ocorreu após o intervalo $[t_{ini}, t_{fim}]$, ou seja, $t > t_{fim}$,
3. f_{begin} , que é o número de eventos $e_k = (v, t) \in events[u]$ contendo v antes do início do intervalo, tais que $k < i$,
4. f_{end} , definido como o número de eventos $e_k = (v, t) \in events[u]$ contendo v antes do fim do intervalo, tais que $k < j$.

Uma vez calculados os valores acima, a aresta (u, v) estará ativa em algum instante do intervalo $[t_{ini}, t_{fim}]$ se f_{begin} for ímpar, indicando que a aresta (u, v) estava ativa no início do intervalo $[t_{ini}, t_{fim}]$, ou se $f_{end} > f_{begin}$, o que significa que algum evento $e = (v, t)$ aconteceu, tal que $t \in [t_{ini}, t_{fim}]$.

Caso nenhum dos dois cenários sejam verdadeiros, significa que a aresta estava desativada no início do intervalo de tempo desejado e não ficou ativa em nenhum momento. Portanto, se algum dos casos acima forem verdadeiros, tem-se que $has_edge(u, v, t_{ini}, t_{fim}) = true$, senão $has_edge(u, v, t_{ini}, t_{fim}) = false$.

4.1.2.1 Análise

Primeiramente, para acessar o *array* $timestamps[u]$ é necessário fazer a decodificação *Delta Gap*, que pode ser feito em $O(m)$ de tempo e memória, onde m é o número de elementos em $events[u]$. Para encontrar os índices i e j pode-se utilizar busca binária, alcançando um desempenho de $O(\log m)$ de tempo e consumo de memória constante.

Para realizar o cálculo de f_{begin} e f_{end} , é necessário iterar em $events[u]$ mantendo a frequência dos eventos (v, t) em um contador, consumindo $O(m)$ de tempo e $O(1)$ de memória.

Portanto, a complexidade final de tempo e memória adicional por consulta é $O(m) = O(|E|)$.

4.1.3 Operação *neighbours*

Para encontrar os vizinhos de um determinado vértice u que estiveram ativos em pelo menos um instante do intervalo de tempo $t = [t_{ini}, t_{fim}]$, é necessário calcular quais arestas (u, v) estavam ativas no início de t e quais ficaram ativas durante o mesmo intervalo de tempo.

Faz-se uma busca linear em $events[u]$ com o auxílio dos *arrays* $timestamps[u]$ e $labels[u]$ mantendo-se dois conjuntos de elementos S_{antes} e $S_{durante}$, que contêm os vértices v tais que: se $v \in S_{antes}$, então a aresta (u, v) estava ativa no início do intervalo de tempo t . Da mesma forma, se $v \in S_{durante}$, então a aresta (u, v) ficou ativa em pelo menos um instante de t .

A resposta para a operação $neighbours(u, v, t_{ini}, t_{fim})$ é $S_{antes} \cup S_{durante}$, que é calculada por meio de uma busca linear com o auxílio de uma tabela *hash*.

4.1.3.1 Análise

Assim como na operação has_edge , é necessário decodificar o *array* $timestamps[u]$ antes de realizar os cálculos necessários. Como visto, essa decodificação é feita em $O(m)$

de tempo e memória, onde m é o número de elementos em $events[u]$.

Para manter os conjuntos S_{antes} e $S_{durante}$ foi utilizada a estrutura `std::unordered_set`, que é uma implementação de conjuntos utilizando tabelas *hash*, disponibilizada pela linguagem C++.

Ao todo, utilizando-se os métodos descritos, tem-se uma complexidade de $O(m) = O(|E|)$ em tempo e memória para realizar o cálculo da operação *neighbours*.

4.2 Estrutura de dados *Time-Interval Log Per Edge (EdgeLog)*

A estrutura *Time-Interval Log Per Edge (EdgeLog)* (XUAN; FERREIRA; JARRY, 2003) é uma representação de grafos temporais compactos cuja ideia principal baseia-se em manter uma lista de adjacências em dois níveis para cada vértice. Para cada vértice u , mantém-se uma lista encadeada $L1_u$ com os identificadores dos vizinhos de u , em ordem crescente. Cada vizinho v presente nessa lista irá possuir um ponteiro para uma outra lista encadeada $L2_v$ com os pares de tempos de início e término da aresta (u, v) , ordenados por tempo de início. A Figura 6 exemplifica essa ideia.

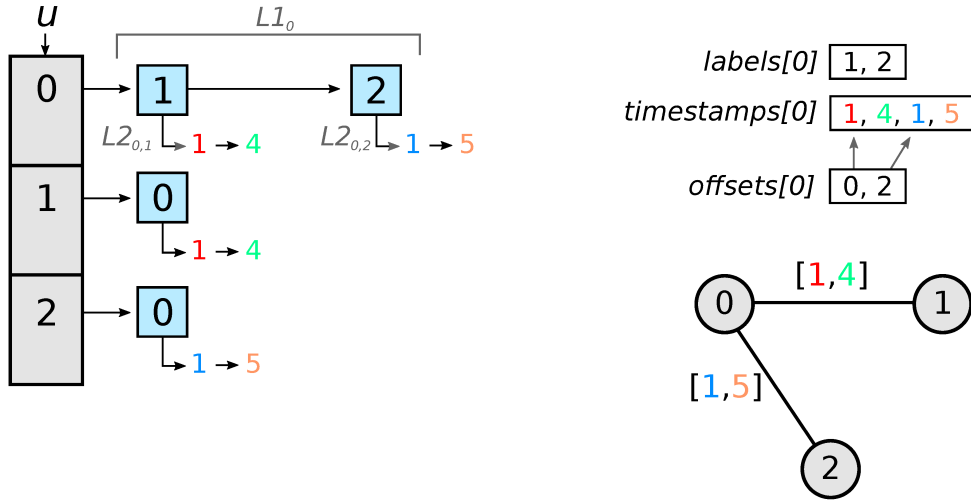


Figura 6 – Exemplo da organização lógica das listas e sub-listas que compõem a estrutura *EdgeLog* para o grafo temporal bidirecional ilustrado. As listas $L1_u$ em azul representam quem são os vizinhos de um determinado vértice u e cada vizinho possui uma sub-lista $L2_{u,v}$ com os pares de intervalos de tempo nos quais a aresta (u, v) ficou ativa. Na direita são ilustrados os arrays $labels[0]$, $timestamps[0]$ e $offsets[0]$ que são utilizados para armazenar a estrutura, onde $labels[0]$ armazena os elementos de $L1_0$, $timestamps[0]$ armazena a concatenação das listas $L2_{0,1}$ e $L2_{0,2}$, e $offsets[0]$ indica o início de cada sub-lista $L2$ em $timestamps[0]$.

4.2.1 Construção

Para evitar o uso de ponteiros, são criados 3 *arrays* que juntos representam a lista de adjacência de um determinado vértice u . São eles:

- $labels[0, P - 1]$, onde P é o número de vértices v tais que existe pelo menos 1 evento (u, v) na lista de adjacência. Esse *array* armazena os vizinhos de u na mesma ordem em que eles aparecem em $L1_u$.
- $timestamps[0, Q - 1]$, onde Q é o número de eventos existentes na lista de adjacência de u , armazena os instantes no tempo onde ocorreram cada evento de forma sequencial. Pode ser visto como a concatenação de $L2_{u,v}, \forall v \in labels[u]$. O valor $timestamps[x], 0 \leq x < Q - 1$, onde x é par, representa o início de um intervalo de tempo da aresta (u, v) , que teve fim no instante de tempo $timestamps[x + 1]$.
- $offsets[0, P - 1]$, onde P é o número de vértices v adjacentes a u , como definido para $labels[u]$. Armazena para cada vizinho v , o índice de início dos instantes de tempo dos eventos (v, t) no *array* $timestamps[u]$.

Para o exemplo da Figura 6, tem-se que $labels[0] = [1, 2]$, $timestamps[0] = [1, 4, 1, 5]$ e $offsets[0] = [0, 2]$. Visando diminuir a variabilidade dos dados e alcançar maior compactação, codifica-se os três *arrays* utilizando a codificação *Delta Gap*, assim como é feito para a estrutura de dados *EveLog* 4.1. Após essa primeira codificação, armazena-se os resultado utilizando codificação de *Huffman* (HUFFMAN, 1952).

4.2.1.1 Análise

Sejam S_{labels} , $S_{timestamps}$ e $S_{offsets}$ as sequências geradas para os três *arrays* mencionados anteriormente, codificados utilizando a codificação *delta gap*. Seja também o valor n igual à soma dos tamanhos das três sequências. Após a codificação *Huffman*, tem-se um consumo de memória menor que $O(n\mathcal{H}_0(S) + n)$ bits (NAVARRO, 2016b), onde S é a concatenação de S_{labels} , $S_{timestamps}$ e $S_{offsets}$, enquanto que $\mathcal{H}_0(S)$ é a entropia de ordem zero da sequência S .

Para realizar a construção das sequências S_{labels} , $S_{timestamps}$ e $S_{offsets}$, é preciso ordenar os eventos, com um desempenho de $O(|E| \log |E|)$ em tempo. Após ordenados os eventos, os mesmos são comprimidos utilizando a codificação *delta gap* em $O(|E|)$ de tempo e a codificação de *Huffman*, que também é calculada em $O(n\mathcal{H}_0(S)) = O(|E|)$ em tempo.

Portanto, armazena-se a estrutura *EdgeLog* utilizando $O(n\mathcal{H}_0(S)) = O(|E|)$ bits de memória.

4.2.2 Operação *has_edge*

Para responder a consulta $has_edge(u, v, t_{ini}, t_{fim})$, primeiramente é realizada a descompressão dos *arrays labels*, *timestamps* e *offsets*.

Seja pos_v o índice do vértice v em $labels[u]$, que pode ser calculado em $O(\log P)$ utilizando busca binária. Calcula-se $i = offsets[pos_v]$ e $j = offsets[pos_v + 1] - 1$, que representam o *sub-array timestamps* $[i, j]$ que contém informações sobre os valores dos intervalos de tempo em que a aresta (u, v) ficou ativa.

A operação tem resultado *true* se existe algum intervalo de tempo $[timestamps[x], timestamps[x + 1]]$, onde x é par e $i \leq x < j$, que possui interseção em pelo menos um instante com o intervalo $[t_{ini}, t_{fim}]$. Caso contrário, retorna-se *false* indicando que não existe aresta entre os vértices u e v no intervalo desejado.

4.2.2.1 Análise

Primeiramente é necessário descomprimir os *arrays labels*, *timestamps* e *offsets*, que é feito em $O(Q)$ de tempo e memória adicional, onde Q é o número de eventos na lista de adjacência de u . Após a descompressão, percorre-se os $\frac{Q}{2}$ intervalos de tempo t' em $L2_v$ verificando se há alguma interseção entre t' e o intervalo de tempo $[t_{ini}, t_{fim}]$ da consulta. Essa verificação pode ser feita em tempo constante e, portanto, essa etapa também possui complexidade em tempo $O(Q)$.

No pior dos casos Q é igual a $|E|$ e, portanto, a complexidade final em tempo e memória adicional por operação *has_edge* é $O(|E|)$.

4.2.3 Operação *neighbours*

A operação $neighbours(u, t_{ini}, t_{fim})$ retorna como resultado uma lista composta por todos os vértices $v \in labels[u]$, tais que $has_edge(u, v, t_{ini}, t_{fim}) = true$. Para evitar descompressão dos *arrays labels*, *timestamps* e *offsets* a cada chamada da função *has_edge*, tais *arrays* são decodificados uma única vez e utilizados pela função *has_edge* conforme apresentado na Seção 4.2.2.

4.2.3.1 Análise

Como cada chamada da operação *has_edge* irá percorrer apenas $L2_v$ para um vizinho v específico, ao fim do algoritmo cada lista $L2$ será percorrida uma única vez e, portanto, totalizando $O(Q)$ operações.

Dessa forma, assim como a operação *has_edge*, a operação *neighbours* possui desempenho de $O(|E|)$ em tempo e memória.

4.3 Estrutura de dados *Compact Adjacency Sequence (CAS)*

A *Compact Adjacency Sequence (CAS)* (CARO; RODRÍGUEZ; BRISABOA, 2015) é uma estrutura de dados utilizada para representar grafos temporais de maneira compacta. Ela se baseia em representar a lista de adjacência de um determinado vértice u como uma sequência de símbolos $c \in [0, |V| + T]$ representando para cada tempo t_i em questão, quem são os vértices v tais que existe um evento (v, t_i) na lista de adjacência de u . Por exemplo, seja $events[u] = [(a, 1), (b, 3), (a, 4), (b, 4)]$ a representação de duas arestas: (u, a) com intervalo de tempo $[1, 4]$ e (u, b) com intervalo de tempo $[3, 4]$. A sequência de símbolos S_u para essas arestas será $[1, a, 3, b, 4, a, b]$.

Para que seja possível distinguir os valores referentes aos vértices e aos instantes de tempo na sequência, cada instante t de um evento $(v, t) \in events[u]$ é mapeado para o valor $t + |V|$. Dessa forma, os valores presentes nas sequências pertencem ao alfabeto $\Sigma = [0, |V| + T]$, onde valores menores que $|V|$ representam vértices no grafo e valores $t \geq |V|$ representam o instante de tempo $t - |V|$ de algum evento presente em $events[u]$. É importante notar que esse mapeamento torna o grafo estático, não sendo possível adicionar vértices ao mesmo.

Logo, considerando $T = 4$ e $|V| = 2$, a sequência S_u utilizada como exemplo anteriormente é representada como $[3, 0, 5, 1, 6, 0, 1]$ após o mapeamento dos instantes de tempo, considerando $a = 0$ e $b = 1$.

Seja a sequência de símbolos para os eventos de um vértice $u \in V$ representada por S_u . A estrutura *CAS* irá representar o grafo como uma sequência S composta pela concatenação das subsequências S_u , juntamente com um *bitvector* B representando quais intervalos dessa concatenação representam a sequência de símbolos S_u de um vértice u específico.

Para uma sequência S_u de m_u elementos, cria-se uma sequência de *bits* $B_u[0, m_u]$, tal que $B_u[0] = 1$ e $B_u[i] = 0$ para $i \in [1, m_u]$. O *bitvector* B é definido como a concatenação das sequências $B_u, u \in V$, tal que cada *bit* 1 refere-se a um vértice e cada *bit* 0 refere-se a um elemento de S , de forma que se S possui m elementos e o grafo possui $|V|$ vértices, B possui $m + |V|$ *bits*. Essa construção de B permite encontrar em qual intervalo $[i, j]$ de S uma sequência S_u é representada, por meio das fórmulas $i = rank(B, select(B, u, 1), 0)$ e $j = rank(B, select(B, u + 1, 1), 0) - 1$. A Figura 7 exemplifica essa definição das sequências S e B .

Para realizar consultas no grafo utilizando esta representação, manipula-se a sequência S com o auxílio de uma *wavelet tree*, definida na Seção 3.3, que reduz operações complexas em uma sequência de símbolos em um determinado alfabeto a uma sequência de operações *rank* e *select* em *bitvectors*.

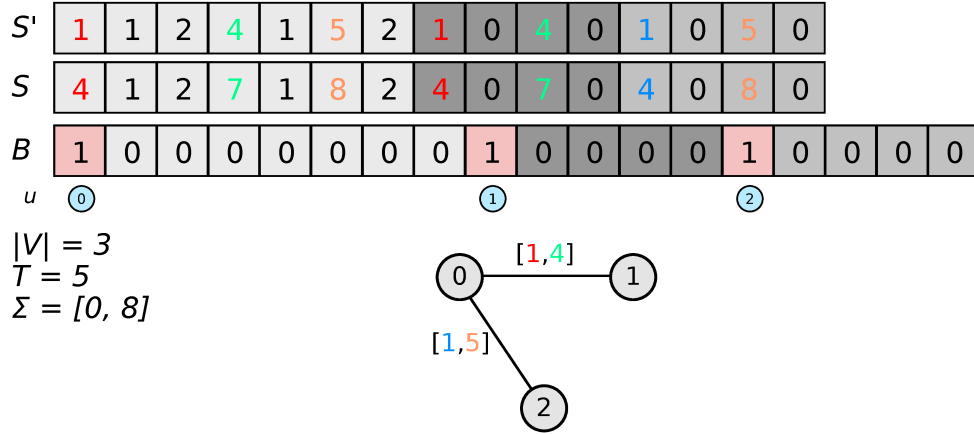


Figura 7 – Exemplo da organização da estrutura CAS. A sequência S' é o resultado da concatenação das subsequências $S_u, u \in V$, representadas em tons de cinza distintos para cada u . Após o mapeamento dos instantes de tempo t para $t \leftarrow t + |V|$, cria-se a sequência S que é armazenada por uma *wavelet tree* (descrita na Seção 3.3). Cada instante de tempo é ilustrado com cores diferentes e representa um evento do grafo temporal apresentado, tal que valores com a mesma cor representam o mesmo evento para uma determinada aresta bidirecional. O *bitvector* B é tal que o valor $B[i] = 0$ representa um elemento da sequência S_u para $u = \text{rank}(B, i, 1)$ e, portanto, cada sequência contígua de zeros em B representa uma sub-sequência S_u em S , enquanto que o valor $B[i] = 1$ indica o início da próxima sequência de zeros, referente ao vértice $u = \text{rank}(B, i, 1) + 1$.

4.3.1 Construção

Para construir a sequência S , primeiro ordena-se os eventos em $\text{events}[u]$ para cada vértice $u \in V$ em ordem crescente de tempo. Após ordenados os eventos, percorre-se $\text{events}[u]$ para todo $u \in V$ em ordem crescente, construindo-se a sequência S e o *bitvector* B .

Após a construção da sequência S , constrói-se uma *wavelet tree* sobre a mesma, que será utilizada para realizar as operações *has_edge* e *neighbours* conforme descrito nas Seções 4.3.3 e 4.3.4 a seguir.

4.3.1.1 Análise

Para ordenar todos os eventos para cada vértice, foi utilizada a função `std::sort` da biblioteca padrão da linguagem C++, com complexidade média $O(|E| \log |E|)$ para ordenar todos os $2|E|$ eventos, onde $|E|$ é o número de arestas do grafo.

A construção de S é feita realizando-se uma busca linear nos eventos. No pior dos casos, existirão $2|E|$ elementos em S , representando todos os eventos do grafo e totalizando uma complexidade de $O(|E|)$ em tempo e memória. S é utilizada para construir a *wavelet tree* no alfabeto $\Sigma = [0, |V| + T]$, com $\sigma = |V| + T + 1$ e desempenho de $O(|E| \log \sigma)$ de

tempo.

Como S é representada pela *wavelet tree*, não é necessário armazenar S após a construção da estrutura e, dessa forma, apenas o *bitvector* B é armazenado além da *wavelet tree*.

Portanto, a estrutura CAS ocupa $O(w\sigma + |E| \log \sigma + \sigma) = O(w\sigma + |E| \log \sigma)$ bits de espaço em memória para ser armazenada, onde w é o tamanho da palavra do sistema e pode ser construída em $O(|E| \log |E| + |E| \log \sigma)$ de tempo.

4.3.2 Operações extras para a *wavelet tree*

Para que seja possível realizar as operações *has_edge* e *neighbours* utilizando a *wavelet tree* construída sobre a sequência S é necessário definir 3 novas operações além das que foram definidas anteriormente na Seção 3.3:

- $range_next_value_pos(S, l, r, val)$

Retorna o menor índice j em S tal que $0 \leq l \leq j \leq r < n$ e $S[j] \geq val$, onde n é o número de elementos em S . Caso não exista nenhum j válido, retorna-se $r + 1$.

- $range_count(S, l, r, val)$

Retorna a frequência de val na subsequência $S[l, r]$. Formalmente, retorna quantos índices j existem em S tal que $0 \leq l \leq j \leq r < n$ e $S[j] = val$, onde n é o número de elementos em S .

- $range_report(S, l, r)$

Retorna um *array* de pares (x, f_x) indicando para cada valor $x \in S[l, r]$ qual a sua frequência f_x no intervalo.

4.3.2.1 Operação $range_next_value_pos$

Para implementar a operação $range_next_value_pos(S, l, r, val)$, utiliza-se uma abordagem recursiva semelhante à operação *select*, descrita na Seção 3.3.3.

Sejam X a raiz da *wavelet tree* e $[L, R]$ o intervalo em Σ representado por X . Caso val seja maior que $\lfloor \frac{L+R}{2} \rfloor$, a resposta, caso exista, encontra-se na sub-árvore de X_{dir} . Caso contrário, todo elemento presente em X_{dir} é válido, porém, podem existir elementos válidos em X_{left} . Nesse caso, é preciso calcular o mínimo entre a possível resposta em X_{left} e o índice do primeiro elemento presente em X_{dir} .

A implementação desse algoritmo pode ser encontrado na linha 85 do código no *github* em <https://web.archive.org/web/20211020193747/https://github.com/nunesg/compact-temporal-graphs/blob/main/lib/WaveletTree.h> e é implementado em $O(\log \sigma)$ em tempo e memória utilizando recursão.

4.3.2.2 Operação *range_count*

Essa operação é equivalente a $rank(S, r + 1, val) - rank(S, l, val)$. Dessa forma, o cálculo de $range_count(S, l, r, val)$ é feito com a mesma complexidade em tempo da operação *rank* descrita na Seção 3.3.2, com desempenho de $O(\log \sigma)$ em tempo e $O(1)$ de memória adicional à utilizada pela *wavelet tree*.

4.3.2.3 Operação *range_report*

Para armazenar os pares (x, f_x) da resposta para a operação $range_report(S, l, r)$, utiliza-se uma tabela *hash* implementada pela estrutura `std::unordered_map` padrão da linguagem C++.

Para encontrar os símbolos e suas respectivas frequências, realiza-se uma busca em profundidade na árvore, com complexidade $O(\sigma) = O(|V| + T)$ de tempo e $O(\log \sigma)$ de memória adicional para armazenar a pilha de recursão.

No pior dos casos haverão $O(|E|)$ resultados na resposta da operação e, portanto, utiliza-se $O(|E|)$ de memória para o armazenamento do resultado.

4.3.3 Operação *has_edge*

Para calcular $has_edge(u, v, t_{ini}, t_{fim})$, primeiramente calcula-se o intervalo $[i, j]$ em S que representa a sequência S_u utilizando-se o *bitvector* B por meio das fórmulas $i = rank(B, select(B, u, 1), 0)$ e $j = rank(B, select(B, u + 1, 1), 0) - 1$.

Calculado o intervalo $[i, j]$, calcula-se o intervalo $K = [k_{ini}, k_{fim})$ em S tal que $i \leq k_{ini} \leq k_{end} \leq j + 1$ que representa os eventos que ocorreram em algum instante no tempo pertencente ao intervalo $[t_{ini}, t_{fim}]$. O intervalo K pode ser calculado através das fórmulas $k_{ini} = range_next_value_pos(S, i, j, t_{ini})$ e $k_{fim} = range_next_value_pos(S, i, j, t_{fim} + 1)$, onde $range_next_value_pos$ é a operação descrita na Seção 4.3.2.1 executada pela *wavelet tree* construída sobre a sequência S .

Com ambos os intervalos $[i, j]$ e $[k_{ini}, k_{fim}]$ calculados, utiliza-se a operação $range_count$ da *wavelet tree* para calcular os seguintes valores:

- $f_{antes} = range_count(S, i, k_{ini} - 1, v)$

Representa a frequência de v em $S[i, j]$ antes do tempo t_{ini} .

- $f_{fim} = range_count(S, i, k_{fim} - 1, v)$

Representa a frequência de v em $S[i, j]$ até o tempo t_{fim} .

- $f_{durante} = f_{fim} - f_{antes}$

Representa a frequência de v no intervalo de tempo $[t_{ini}, t_{fim}]$.

A operação *has_edge* retorna o valor *true* se e somente se $f_{durante}$ é maior que 0, indicando que ocorreu um evento com o vértice v em $[t_{ini}, t_{fim}]$, ou se f_{antes} for ímpar, o que significa que havia uma aresta (u, v) ativa no início do intervalo de tempo desejado.

4.3.3.1 Análise

Como todas as operações utilizadas possuem desempenho $O(\log \sigma)$ em tempo e a operação *range_next_value_pos* também possui essa complexidade em memória, tem-se que a operação *has_edge* possui complexidade $O(\log \sigma)$ de tempo e memória.

4.3.4 Operação *neighbours*

Primeiramente calcula-se os intervalos $[i, j]$ e $[k_{ini}, k_{fim}]$ assim como descrito na Seção 4.3.3. Com esses intervalos computados, pode-se calcular os seguintes valores:

- $R_{antes} = \text{range_report}(S, i, k_{begin} - 1)$

Contém as frequências de todos os símbolos presentes em $S[i, j]$ antes do instante t_{ini} .

- $R_{durante} = \text{range_report}(S, k_{begin}, k_{end} - 1)$

Contém as frequências de todos os símbolos presentes em $S[i, j]$ durante o intervalo $[t_{ini}, t_{fim}]$.

Os vizinhos de u no intervalo $[t_{ini}, t_{fim}]$ são aqueles vértices v que aparecem em $R_{durante}$, ou que possuem frequência ímpar em R_{antes} indicando que a aresta (u, v) estava ativa no instante t_{ini} .

4.3.4.1 Análise

Para construir a lista de vizinhos v é necessário fazer uma busca linear nos conjuntos $R_{durante}$ e R_{antes} com o auxílio de uma tabela *hash* para evitar adicionar elementos repetidos à resposta. Isso é feito em $O(|V| + T)$ em tempo e utiliza $O(|E|)$ de memória adicional, assim como a operação *range_report* utilizada no algoritmo.

Dessa forma, a complexidade da operação *neighbours* é $O(|V| + T)$ em tempo e $O(|E|)$ em memória.

4.4 Considerações finais

Após analisar as estruturas de dados de representação de grafos temporais é possível comparar a eficiência de cada uma em tempo e consumo de memória, como demonstrado pelas Tabelas 3 e 4, respectivamente.

	<i>EveLog</i>	<i>EdgeLog</i>	<i>CAS</i>
<i>Construção</i>	$O(E)$	$O(E)$	$O(E \log(V + T) + E \log E)$
<i>Operação has_edge</i>	$O(E)$	$O(E)$	$O(\log(V + T))$
<i>Operação neighbours</i>	$O(E)$	$O(E)$	$O(V + E + T)$

Tabela 3 – Comparação das complexidades em tempo das operações *has_edge* e *neighbours*, assim como o tempo de construção para cada estrutura de representação de grafos temporais.

	<i>EveLog</i>	<i>EdgeLog</i>	<i>CAS</i>
<i>Armazenamento (bits)</i>	$O(E \log w)$	$O(E)$	$O(w(V + T) + E \log(V + T))$
<i>Operação has_edge</i>	$O(E)$	$O(E)$	$O(\log(V + T))$
<i>Operação neighbours</i>	$O(E)$	$O(E)$	$O(E)$

Tabela 4 – Comparação das complexidades em consumo de memória adicional pelas operações *has_edge* e *neighbours*, assim como a quantidade de *bits* utilizados para o armazenamento de cada estrutura de representação de grafos temporais.

5 Experimentação

Para cada representação de grafos temporais descritas na Seção 4, foram realizadas as seguintes etapas:

1. Implementação utilizando as técnicas descritas;
2. Execução das operações suportadas pelo grafo (veja Tabela 2) com valores de entrada aleatórios de diferentes ordens de grandeza;
3. Análise da eficiência da implementação utilizada, tanto em tempo médio por operação quanto em consumo de memória, baseado nos dados coletados na experimentação.

Os algoritmos foram executados utilizando bases de dados de diferentes ordens de grandeza criadas por geradores de grafos temporais artificiais com o intuito de obter o consumo e memória e desempenho médio por operação, para que seja possível comparar as estruturas entre si.

5.1 Detalhes da implementação

As estruturas analisadas neste trabalho foram implementadas na linguagem de programação C++11, utilizando o paradigma de orientação a objetos. O código pode ser acessado no diretório do *GitHub* pelo link <https://github.com/nunesg/compact-temporal-graphs>.

A ferramenta de compilação utilizada foi o programa *Bazel*¹, que possibilita a compilação e a execução de testes unitários para o projeto. Também foram utilizados *scripts* em linguagem *Python3* para criação dos grafos temporais, execução dos testes para cada estrutura de representação de grafos temporais e armazenamento dos resultados em um banco de dados *MySQL*.

Os grafos temporais foram gerados aleatoriamente e armazenados em um arquivo de texto. Uma vez gerado tal arquivo com o grafo de entrada, eram executados os experimentos para cada estrutura de grafo temporal, para que os resultados comparados fossem relativos à mesma entrada.

¹ Acessado em 20/10/2021: <https://web.archive.org/web/20211018114250/https://docs.bazel.build/versions/4.2.1/bazel-overview.html>

A medição do tempo por operação foi feita utilizando o método `high_resolution_clock::now()` da biblioteca `<chrono>` da linguagem C++, para calcular a diferença de tempo entre o início da operação até o final de seu cálculo.

Para medir o uso de memória por cada estrutura, foi necessário implementar um método `measure_memory` para cada estrutura de dados implementada, que mede o uso de memória alocado para a estrutura e suas sub-estruturas auxiliares através do uso da função `sizeof` da linguagem C. Por exemplo, para medir o uso de memória de um nó de uma *wavelet tree*, soma-se os *bytes* utilizados pelas constantes (ponteiros e variáveis indicando o sub-intervalo do alfabeto representado pelo nó), assim como o resultado do método `measure_memory` aplicado nos filhos recursivamente e no *bitvector* armazenado no nó em questão. Este algoritmo é implementado na linha 133 do *link* <https://web.archive.org/web/20211020193747/https://github.com/nunesg/compact-temporal-graphs/blob/main/lib/WaveletTree.h>.

5.2 Resultados

As tabelas abaixo contêm o tempo médio em milissegundos por operação para cada implementação de grafos temporais, considerando um grafo temporal gerado aleatoriamente com diferentes valores para $|E|$ e T . As tabelas de armazenamento indicam, para cada estrutura de dados, quantos *kilobytes* foram utilizados para armazenar o mesmo grafo de entrada.

<i>EveLog</i>					<i>EdgeLog</i>				
$ V $	<i>has_edge</i>	<i>neighbours</i>	<i>aggregate</i>		$ V $	<i>has_edge</i>	<i>neighbours</i>	<i>aggregate</i>	
10	0.0	0.1	5.8		10	0.03	0.3	6.0	
100	0.0	0.0	3.2		100	0.0	0.0	7.4	
1000	0.0	0.0	5.0		1000	0.0	0.0	8.2	
10000	0.0	0.0	11.2		10000	0.0	0.0	11.8	
100000	0.0	0.0	37.8		100000	0.0	0.0	42.6	
<i>CAS</i>					Lista de Adjacência				
$ V $	<i>has_edge</i>	<i>neighbours</i>	<i>aggregate</i>		$ V $	<i>has_edge</i>	<i>neighbours</i>	<i>aggregate</i>	
10	0.0	0.45	9.6		10	0.0	0.0	0.0	
100	0.0	0.0	45.0		100	0.0	0.0	0.0	
1000	0.0	0.0	62.2		1000	0.0	0.0	0.2	
10000	0.0	0.0	126.4		10000	0.0	0.0	5.0	
100000	0.0	0.0	510.4		100000	0.0	0.0	53.0	
Tempo de Construção (ms)					Armazenamento (kb)				
$ V $	Lista Adj.	<i>CAS</i>	<i>EdgeLog</i>	<i>EveLog</i>	$ V $	Lista Adj.	<i>CAS</i>	<i>EdgeLog</i>	<i>EveLog</i>
10	0.0	143.0	33.0	6.0	10	11	1271	4	8
100	0.0	161.0	33.0	12.0	100	14	1387	16	20
1000	0.0	190.0	31.0	12.0	1000	35	2216	113	129
10000	9.0	301.0	33.0	30.0	10000	246	4880	999	1151
100000	88.0	714.0	75.0	115.0	100000	2355	8536	9788	11347

Tabela 5 – $|E| = 1000$, $T = 1000$.

<i>EveLog</i>					<i>EdgeLog</i>				
$ V $	<i>has_edge</i>	<i>neighbours</i>	<i>aggregate</i>		$ V $	<i>has_edge</i>	<i>neighbours</i>	<i>aggregate</i>	
10	0.01	0.0	3.4		10	0.37	0.35	6.8	
100	0.0	0.0	4.2		100	0.0	0.0	16.8	
1000	0.0	0.0	6.6		1000	0.0	0.0	11.0	
10000	0.0	0.0	8.2		10000	0.0	0.0	20.2	
100000	0.0	0.0	47.8		100000	0.0	0.0	56.0	

<i>CAS</i>					Lista de Adjacência				
$ V $	<i>has_edge</i>	<i>neighbours</i>	<i>aggregate</i>		$ V $	<i>has_edge</i>	<i>neighbours</i>	<i>aggregate</i>	
10	0.0	1.7	18.6		10	0.0	0.0	0.0	
100	0.0	0.1	49.2		100	0.0	0.0	0.0	
1000	0.0	0.0	97.2		1000	0.0	0.0	1.0	
10000	0.0	0.0	141.8		10000	0.0	0.0	5.0	
100000	0.0	0.0	656.6		100000	0.0	0.0	50.4	

Tempo de Construção (ms)					Armazenamento (kb)				
$ V $	Lista Adj.	<i>CAS</i>	<i>EdgeLog</i>	<i>EveLog</i>	$ V $	Lista Adj.	<i>CAS</i>	<i>EdgeLog</i>	<i>EveLog</i>
10	0.0	384.0	47.0	16.0	10	11	11317	7	10
100	0.0	393.0	66.0	21.0	100	14	11354	18	22
1000	1.0	413.0	35.0	19.0	1000	35	12181	116	130
10000	7.0	478.0	66.0	37.0	10000	246	14675	1002	1152
100000	55.0	1141.0	95.0	121.0	100000	2355	18253	9792	11347

Tabela 6 – $|E| = 1000$, $T = 100000$.

<i>EveLog</i>					<i>EdgeLog</i>				
$ V $	<i>has_edge</i>	<i>neighbours</i>	<i>aggregate</i>		$ V $	<i>has_edge</i>	<i>neighbours</i>	<i>aggregate</i>	
10	25.97	43.7	478.6		10	6.42	5.95	54.6	
100	2.12	3.7	334.0		100	2.61	2.65	296.0	
1000	0.0	0.0	550.6		1000	0.68	0.8	824.4	
10000	0.0	0.0	405.4		10000	0.0	0.0	1068.6	
100000	0.0	0.0	578.0		100000	0.0	0.0	1097.6	

<i>CAS</i>					Lista de Adjacência				
$ V $	<i>has_edge</i>	<i>neighbours</i>	<i>aggregate</i>		$ V $	<i>has_edge</i>	<i>neighbours</i>	<i>aggregate</i>	
10	0.01	2.35	26.0		10	0.0	1.05	7.0	
100	0.0	3.7	343.8		100	0.0	0.0	26.0	
1000	0.0	1.9	2510.2		1000	0.0	0.0	34.2	
10000	0.0	0.2	6710.2		10000	0.0	0.0	52.0	
100000	0.0	0.0	13205.8		100000	0.0	0.0	121.0	

Tempo de Construção (ms)					Armazenamento (kb)				
$ V $	Lista Adj.	<i>CAS</i>	<i>EdgeLog</i>	<i>EveLog</i>	$ V $	Lista Adj.	<i>CAS</i>	<i>EdgeLog</i>	<i>EveLog</i>
10	10.0	7825.0	401.0	565.0	10	1172	15846	38	705
100	16.0	12191.0	1450.0	931.0	100	1174	22790	238	797
1000	10.0	18641.0	3812.0	1552.0	1000	1195	34581	683	1018
10000	32.0	35283.0	4337.0	2226.0	10000	1406	52550	1761	2234
100000	91.0	117552.0	4038.0	2271.0	100000	3515	140745	11497	12991

Tabela 7 – $|E| = 100000$, $T = 1000$.

5.2.1 Análise

Após a execução dos experimentos, pode-se perceber que as estruturas de representação de grafos temporais utilizando estruturas de dados compactas implementadas neste projeto não tiveram êxito em diminuir o uso de memória para grafos temporais de maior escala, quando comparadas à representação não-compacta que utiliza listas de

<i>EveLog</i>					<i>EdgeLog</i>				
$ V $	<i>has_edge</i>	<i>neighbours</i>	<i>aggregate</i>		$ V $	<i>has_edge</i>	<i>neighbours</i>	<i>aggregate</i>	
10	27.66	35.6	368.8		10	24.38	28.9	233.8	
100	2.36	3.1	496.8		100	9.12	9.0	917.0	
1000	0.0	0.0	475.4		1000	0.94	1.05	1029.0	
10000	0.0	0.0	588.4		10000	0.0	0.0	1297.6	
100000	0.0	0.0	706.2		100000	0.0	0.0	1304.8	
<i>CAS</i>					Lista de Adjacência				
$ V $	<i>has_edge</i>	<i>neighbours</i>	<i>aggregate</i>		$ V $	<i>has_edge</i>	<i>neighbours</i>	<i>aggregate</i>	
10	0.0	92.55	1091.0		10	0.0	2.0	24.8	
100	0.0	20.9	2278.4		100	0.0	0.0	14.6	
1000	0.0	3.8	5365.0		1000	0.0	0.0	48.2	
10000	0.0	0.4	8603.4		10000	0.0	0.0	35.4	
100000	0.0	0.0	8629.0		100000	0.0	0.0	106.8	
Tempo de Construção (ms)					Armazenamento (kb)				
$ V $	Lista Adj.	<i>CAS</i>	<i>EdgeLog</i>	<i>EveLog</i>	$ V $	Lista Adj.	<i>CAS</i>	<i>EdgeLog</i>	<i>EveLog</i>
10	16.0	134360.0	842.0	767.0	10	1172	139983	182	749
100	12.0	134777.0	5427.0	1636.0	100	1174	142822	725	895
1000	18.0	3833020.0	4578.0	2158.0	1000	1195	145283	878	1158
10000	19.0	3445520.0	4084.0	3087.0	10000	1406	155883	1928	2384
100000	143.0	188572.0	4106.0	2640.0	100000	3515	238686	11743	13099

Tabela 8 – $|E| = 100000$, $T = 100000$.

adjacência tradicionais.

Isso pode ser consequência de um alto número de estruturas auxiliares que são necessárias para construir cada estrutura de grafo temporal utilizando as técnicas descritas neste trabalho, o que aumenta a constante multiplicativa da complexidade em memória utilizada pelas estruturas. Por exemplo, percebe-se um alto consumo de memória pela estrutura de dados *CAS*, que se dá em grande parte pelo alto consumo de memória adicional necessária para responder consultas do tipo *select* em *bitvectors* com complexidade $O(1)$ em tempo, conforme descrito na Seção 3.2.2.

Outro fator que pode ter influenciado o desempenho foi o uso de grafos gerados aleatoriamente, visto que as técnicas de compressão utilizadas pelas estruturas de representação de grafos temporais analisadas podem obter um resultado superior quando aplicadas em entradas reais.

Apesar do alto consumo de memória, a estrutura de dados *CAS* se mostrou consistente em executar operações do tipo *has_edge* de maneira rápida, tanto em grafos esparsos, ou seja, cuja proporção $\frac{|E|}{|V|}$ é pequena, quanto em grafos densos, com $\frac{|E|}{|V|}$ grande. Porém, a abordagem utilizando listas de adjacência também entregou resultados excelentes, o que inviabiliza o uso da estrutura *CAS* para grafos dos tamanhos analisados neste projeto, devida à sua maior complexidade de implementação e manutenção.

Em relação às estruturas de dados *EveLog* e *EdgeLog*, ambas mostraram desempenho inferior à estrutura *CAS* para a execução da operação *has_edge*, apesar de serem mais eficientes tanto em tempo de construção quanto em tempo por operações dos ti-

pos *neighbours* e *aggregate*. Também percebe-se que a estrutura *EdgeLog*, é mais eficiente que a estrutura *EveLog* para executar operações do tipo *aggregate* em grafos densos, mas possui desempenho inferior em grafos esparsos.

Em termos de tempo de construção, percebe-se que em geral a estrutura *EveLog* possui um tempo de construção menor que a estrutura *EdgeLog*, enquanto que o tempo de construção da estrutura *CAS* é claramente superior, em alguns casos mais de mil vezes mais lento.

6 Conclusões

Durante o desenvolvimento deste projeto foram utilizados diversas referências teóricas de como implementar as estruturas de dados de representação de grafos temporais. Porém, percebeu-se uma dificuldade para encontrá-las implementadas em alguma linguagem de programação para ter uma referência do ponto de vista prático. Dessa forma, as implementações feitas neste trabalho podem servir de base para futuras implementações das mesmas estruturas, assim como de outras semelhantes.

Uma possível continuação para este projeto é realizar a análise experimental de outras estruturas de dados para representar grafos temporais, como as estruturas *Compact Events ordered by Time (CET)* (CARO; RODRÍGUEZ; BRISABOA, 2015) que faz o uso de *wavelet trees* em alfabetos bidimensionais e *Temporal Graph Compressed Suffix Array (TGCSA)* que utiliza *suffix arrays* compactos (SADAKANE, 2003) para realizar consultas em grafos temporais representados em forma de *strings* (BRISABOA et al., 2014).

Além disso, também pode ser interessante implementar as estruturas de representação de grafos temporais compactos utilizando outras tecnologias, linguagens de programação e algoritmos, comparando os resultados obtidos nessas outras abordagens com o resultado dos experimentos realizados neste trabalho.

Devido aos resultados não esperados após a experimentação (estruturas compactas utilizando mais memória que listas de adjacência tradicionais), percebe-se que devido à forma com que foram implementadas as estruturas, não foi possível obter na prática o desempenho teoricamente previsto pelas mesmas.

Uma decisão que influenciou o alto consumo de memória da estrutura *CAS* foi o uso de *bitvectors* que implementam operações do tipo *select* em tempo constante. Durante os testes foi constatado um alto consumo de memória pela implementação feita neste projeto da operação *select* proposta teoricamente por (CLARK, 1997). Uma possível solução para o alto consumo de memória é implementar a operação *select* utilizando uma combinação de *rank* e busca binária para realizar o cálculo da operação com pouca memória adicional. Outra possível estratégia que pode diminuir o uso de memória pelos *bitvectors* é utilizar *bitvectors* comprimidos, conforme apresentado em (NAVARRO, 2016a, Capítulo 4).

Referências

- BRISABOA, N. et al. A compressed suffix-array strategy for temporal-graph indexing. In: *International Symposium on String Processing and Information Retrieval*. [S.l.]: Springer, 2014. p. 77–88. Citado 2 vezes nas páginas 7 e 44.
- CARO, D.; RODRÍGUEZ, M.; BRISABOA, N. Data structures for temporal graphs based on compact sequence representations. *Information Systems*, v. 51, p. 1–26, 2015. Citado 5 vezes nas páginas 4, 7, 27, 33 e 44.
- CLARK, D. Compact pat trees. University of Waterloo, 1997. Citado 2 vezes nas páginas 19 e 44.
- ELIAS, P. Universal codeword sets and representations of the integers. *IEEE transactions on information theory*, IEEE, v. 21, n. 2, p. 194–203, 1975. Citado na página 15.
- GROSSI, R.; GUPTA, A.; VITTER, J. High-order entropy-compressed text indexes. In: *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*. [S.l.]: Society for Industrial and Applied Mathematics, 2003. p. 841–850. Citado 2 vezes nas páginas 7 e 20.
- HUFFMAN, D. A. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, IEEE, v. 40, n. 9, p. 1098–1101, 1952. Citado na página 31.
- JACOBSON, G. *Succinct static data structures*. Tese (Doutorado) — Carnegie Mellon University, Pittsburgh, PA, USA, 1988. Citado na página 6.
- JACOBSON, G. Space-efficient static trees and graphs. In: *Annual Symp. Foundations of Computer Science*. [S.l.]: IEEE Computer Society, 1989. p. 549–554. Citado na página 6.
- JACOBSON, G. J. *Succinct static data structures*. [S.l.]: Carnegie Mellon University, 1988. Citado na página 17.
- NAVARRO, G. *Compact data structures: A practical approach*. [S.l.]: Cambridge University Press, 2016. Citado 7 vezes nas páginas 6, 9, 10, 14, 18, 22 e 44.
- NAVARRO, G. *Compact data structures: A practical approach*. [S.l.]: Cambridge University Press, 2016. 39-39 p. Citado na página 31.
- NEDEV, S.; KAMENOV, V. Hdd performance research. In: . [S.l.: s.n.], 2018. Citado na página 6.
- SADAKANE, K. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, v. 48, p. 294–313, 2003. Citado 2 vezes nas páginas 7 e 44.
- XUAN, B.; FERREIRA, A.; JARRY, A. Computing shortest, fastest, and foremost journeys in dynamic networks. *International Journal of Foundations of Computer Science*, v. 14, p. 267–285, 2003. Citado 3 vezes nas páginas 4, 7 e 30.