Diferentes abordagens para implementação de estruturas de union find com operação de exclusão

Heitor Santos[1], **Paulo G. S. da Fonseca**[1] **(orientador)**

[1]Centro de Informática – Universidade Federal de Pernambuco (UFPE)
Caixa Postal 7803 – 50.740-530 – Recife – PE – Brasil

{hss2,paguso}@cin.ufpe.br

Resumo. *Um problema recorrente na computação é particionar os nós de um grafo em diferentes conjuntos, utilizando estruturas conhecidas como union-find. Uma implementação padrão dessa estrutura oferece operações para criar um conjunto, procurar a qual conjunto um elemento pertence e unir dois conjuntos. Contudo, para algumas aplicações específicas, é necessário excluir um elemento de um determinado conjunto, o que não é contemplado nas implementações tradicionais. Este trabalho explora e compara duas abordagens distintas para implementar a operação de exclusão de elementos em estruturas union-find. Analisamos o uso de memória, o tempo de execução e a aplicabilidade de cada método, fornecendo uma visão abrangente sobre suas vantagens e desvantagens. Através de experimentos e análise teórica, visamos aprimorar as implementações de union-find, tornando-as mais versáteis para diferentes cenários, usando uma aplicação da Biologia Computacional como estudo de caso.*

Trabalho de Graduação apresentado como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

1. Introdução

A metagenômica é uma área emergente da bioinformática que envolve o estudo de genomas de microrganismos obtidos diretamente de amostras ambientais. O sequenciamento metagenômico extrai sequências curtas (ou *leituras)* de genomas agregados de comunidades de microrganismos inteiras. Num ambiente natural complexo, como solo ou intestino humano, amostras metagenômicas podem conter potencialmente milhares ou mesmo milhões de espécies. Antes da análise, os genomas devem ser reconstruídos parcial ou completamente a partir das leituras. Essa reconstrução pode ser feita através de *re-sequenciamento*, processo no qual leituras são mapeadas para genomas de referência. Outra forma de fazer isso é utilizando a montagem *de novo*, onde os genomas são reconstruídos encontrando sequências de leituras sobrepostas. Por exemplo, no caso onde temos as leituras A, B e C, sendo A = ATCGGCTACGA**TACGCGATCG**, B = **TACGCGATCG**GCTACGTACGAA e C = CCCCCCCCCCCCCCGGGGTT, teríamos dois organismos, a saber ATCGGCTACGATACGCGATCGCTACGTACGAA e CCCCCCCCCCCCCCGGGGTT. Um problema dessa abordagem é que o processo de montagem torna-se altamente intensivo em memória e computação à medida que o tamanho e a complexidade do conjunto de dados aumentam. [Flick et al. 2017].

Em [Zheng et al. 2023], os autores discutem um dispositivo móvel capaz de sequenciar moléculas de DNA e transmitir leituras de DNA em tempo real, utilizado para

estudos *in situ*. Como esses dispositivos geralmente possuem memória e energia limitadas, a análise bioinformática precisa ser ou transferida para um serviço em nuvem (o que nem sempre é viável) ou adiada até que recursos computacionais suficientes estejam disponíveis. As Unidades Taxonômicas Operacionais (OTUs), representadas por clusters de leituras de DNA altamente semelhantes, são utilizadas como uma aproximação da composição microbiana da amostra sequenciada. Cada OTU agrupa leituras que são suficientemente semelhantes entre si, e o número de OTUs pode ser usado como uma estimativa do número de organismos distintos na amostra.

Para representar essas relações, os autores modelam as leituras de DNA como vértices em um grafo, onde vértices conectados representam leituras que se sobrepõem. Essas conexões formam componentes conexos, que correspondem aos OTUs. Esse método de identificar OTUs como componentes conexos de grafos também já foi demonstrado na literatura de maneira satisfatória previamente.[Flick et al. 2017]

No processo de encontrar as sobreposições, pode ser necessário a remoção de trechos de DNA repetidos, presentes na sobreposição de duas leituras. Os autores de [Zheng et al. 2023] propõem uma estrutura union-find que fornece suporte para a operação de remoção, porém que utiliza muita memória para armazenar os trechos do genoma, e realiza essa operação em tempo linear. Em contrapartida, temos uma estrutura com remoções em tempo constante proposta em [Alstrup et al. 2005]. Neste trabalho, iremos implementar as duas abordagens dessa estrutura, submetê-las a testes e comparar qual se comporta melhor em termos de uso de memória e tempo de execução em diferentes casos.

2. Background

Sequenciadores de DNA baseados em nanoporos podem sequenciar moléculas de DNA diretamente e transmitir as leituras de DNA resultantes quase em tempo real. [Jain et al. 2015], [Zheng et al. 2023], [Gardy et al. 2015]. Os autores de [Zheng et al. 2023] propõem uma abordagem para identificar unidades taxonômicas operacionais, tratando-as como componentes conexos em um grafo de sobreposição de leituras provindas desses sequenciadores. Nesse contexto, cada leitura é representada como um vértice do grafo, e dois vértices u e v têm uma aresta os conectando se há uma sobreposição entre o sufixo de u e o prefixo de v. Retomando o exemplo apresentado na seção 1, teríamos três vértices: A = ATCGGCTACGATACGCGATCG, B = TACGCGATCGGCTACGTACGAA e C = CCCCCCCCCCCCCCCGGGGTT. Há uma sobreposição entre A e B, como mostra a Figura 1. Isso implica na existência de uma aresta conectando A a B no nosso grafo, conforme a Figura 2. Ao identificar os componentes conexos desse grafo, observamos que existem dois, ou seja, temos duas unidades taxonômicas operacionais. Na computação, componentes conexos são frequentemente representados por estruturas do tipo union-find. Nessas estruturas, dado um elemento, podemos identificar o representante do componente ao qual ele pertence. Ao identificar os representantes de todos os componentes, determinamos quantos componentes estão presentes no grafo.

A implementação padrão da estrutura union-find dá suporte para as operações de $makeset$, $union$ e $find$. Uma operação $makeset$ gera um conjunto com um único elemento. Uma operação de $union$ recebe dois conjuntos e os une, destruindo os dois

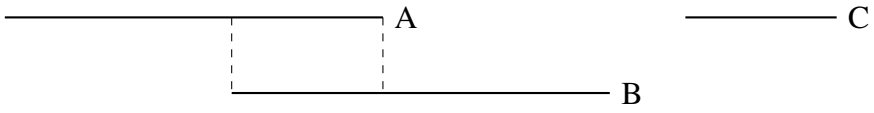

Figura 1. Sobreposições das leituras

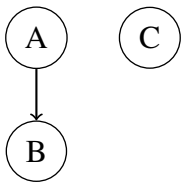

Figura 2. Grafo das sobreposições de leituras

conjuntos originais. Uma operação $find$ recebe um elemento e retorna uma referência ao conjunto que atualmente o contém.

Para abordar o problema de identificação de unidades taxonômicas operacionais em dispositivos móveis, é necessário também considerar a remoção de elementos de um componente. Isso se deve ao fato de que o grafo pode se tornar muito grande, ocupando uma quantidade significativa de memória em dispositivos com capacidade computacional limitada. Em certos componentes, podem existir leituras que vêm da mesma região do genoma e compartilham sobreposições, e que assim proveem informações redundantes. Um exemplo são as leituras na Figura 3. E tem uma sobreposição com F e G. F também tem uma sobreposição com G, então podemos remover F do grafo da Figura 4 sem perder nenhuma informação crítica e assim diminuir o número de leituras que temos que manter na memória, resultando no grafo da Figura 5. Portanto, é essencial implementar uma nova estrutura union-find que suporte a operação de exclusão. Os autores de [Zheng et al. 2023] propõem uma estrutura baseada em tabelas de dispersão.

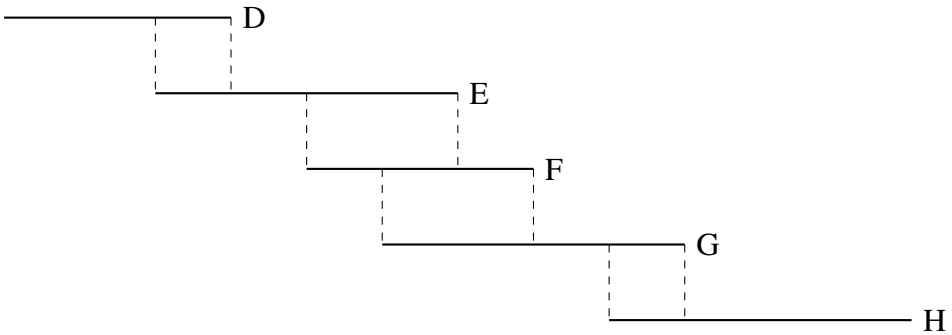

Figura 3. Leituras com informação redundante

2.1. Estrutura baseada em tabelas de dispersão

Um componente conexo é um conjunto de vértices em um grafo onde cada par de vértices está conectado por pelo menos um caminho. Podemos ver então que no grafo da Figura 6 temos três componentes conexos. Para cada componente, temos um representante r, que nada mais é do que um vértice que funciona como uma referência a um componente.

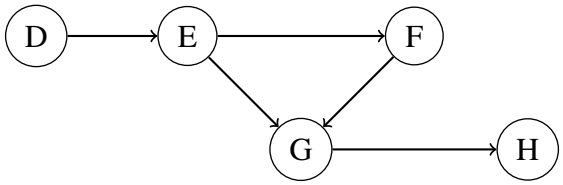

Figura 4. Grafo com informação redundante

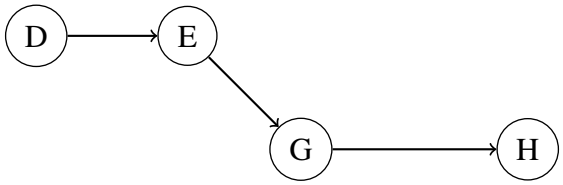

Figura 5. Grafo com informação redundante removida

Para esse exemplo, vamos escolher que os representantes dos componentes sejam 1, 7 e 6. Se nos perguntarmos "qual o representante do componente que contém 2?"ou "qual o representante do componente que contém 3?"teríamos a mesma resposta: 1; o que nos diz que ambos os vértices 1, 2 e 3 estão no mesmo componente.

A estrutura proposta em [Zheng et al. 2023], aqui chamada de estrutura 1, se baseia em utilizar duas tabelas de dispersão. A primeira tabela de dispersão, chamada de `mapping-components`, mapeia de um vértice v para o vértice r, sendo r o representante do componente ao qual v pertence. A segunda tabela de dispersão, `component-members`, mapeia um vértice r para o conjunto A de vértices que fazem parte do componente cujo representante é r. Por exemplo, para o grafo da Figura 6, teríamos as tabelas 1 e 2. Olhando a tabela 1, podemos responder as perguntas feitas no parágrafo anterior. Apenas com essa tabela, em uma implementação padrão da estrutura union-find, poderíamos ter as operações de $makeset$, $union$ e $find$. Porém, ao realizar a operação $delete$, por exemplo, para o vértice 1 dessa estrutura, teríamos outros 4 vértices apontando como representante de seu componente um vértice que não existe mais.

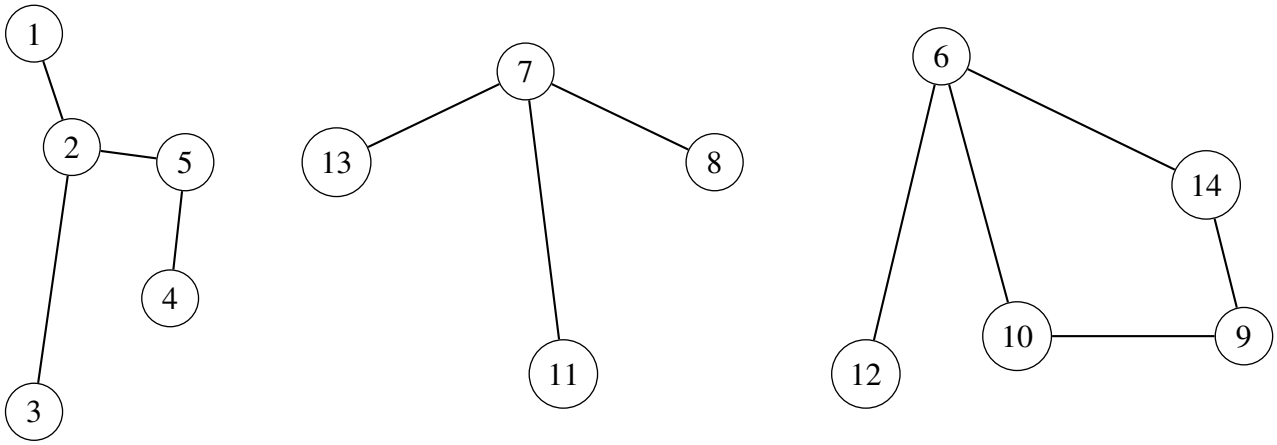

Figura 6.

Para a operação de $makeset$, primeiro definimos v como o representante do con-

v	r
1	1
2	1
3	1
4	1
5	1
6	6
7	7
8	7
9	6
10	6
11	7
12	6
13	7
14	6

Tabela 1. `mapping-components`

r	A
1	$\{1, 2, 3, 4, 5\}$
7	$\{7, 8, 11, 3\}$
6	$\{6, 9, 12, 14\}$

Tabela 2. `component-members`

junto que contém v , definindo como v o valor de `mapping-components` na posição v. Depois criamos um conjunto vazio em `component-members` na posição v e adicionamos v esse conjunto. O acesso a uma posição de uma tabela de dispersão tem complexidade assintótica $\Theta(1)$, a criação do conjunto também tem complexidade $\Theta(1)$, No entanto, a inserção de um valor no conjunto tem complexidade logarítmica, ou seja, $\Theta(\log n)$ onde n representa o tamanho do conjunto, então $makeset$ tem complexidade $\Theta(\log n)$. Para o $find$, apenas consultamos `mapping-components` na posição v, então o custo dessa operação é $\Theta(1)$.

Para a $union$ de dois conjuntos, primeiro, encontramos os representantes dos conjuntos, `root-a` e `root-b`; como já vimos, essa operação tem complexidade $\Theta(1)$. Em seguida, determinamos qual é o representante com o valor numérico menor (o que será usado como novo representante do conjunto unido) e qual com valor numérico maior (que será removido). Em seguida, movemos todos os m elementos do conjunto com o representante maior para o conjunto com o representante menor, atualizando seus representantes em `mapping-components`. A inserção no conjunto tem complexidade $\Theta(\log n)$. Porém a cada elemento que adicionamos, o conjunto aumenta de tamanho de n para $n + 1$; então a complexidade é $\Theta(m \log(n + m - 1))$. Finalmente, removemos a entrada do representante maior de `component-members`, o que tem complexidade $\Theta(1)$. Então a complexidade dessa operação é $\Theta(m \log(n + m - 1))$.

Exemplo 3.1: consideremos os seguintes conjuntos, $A = \{1, 2, 3\}$ e $B = \{4, 5\}$, onde o representante de A é 1 e o representante de B é 4. Suponhamos que desejamos

unir esses dois conjuntos. Primeiro, encontramos os representantes de cada conjunto: `root-a`$= 1$ e `root-b`$= 4$. Comparando os representantes, 1 é menor que 4, então 1 será o novo representante do conjunto unido. Movemos todos os elementos do conjunto com o representante maior B para o conjunto com o representante menor A, atualizando os representantes em `mapping-components` (ou seja, $4 \rightarrow 1$, $5 \rightarrow 1$). Finalmente, removemos a entrada de B em `component-members`. Após a união, o conjunto resultante será $\{1, 2, 3, 4, 5\}$ com 1 como representante.

Para o $delete$ de um elemento: se o elemento não for representante de um conjunto, ele simplesmente é removido das duas tabelas de dispersão. Como já vimos, a remoção de um elemento da tabela de dispersão tem complexidade $\Theta(1)$. Se ele for representante, os membros desse conjunto são movidos para um novo conjunto cujo representante é o primeiro elemento encontrado no conjunto do elemento a ser removido. Depois removemos o antigo representante das duas tabelas de dispersão. Essa movimentação de elementos tem complexidade $\Theta(m \log(n + m - 1))$. Mas como cada componente só tem um representante, vamos assumir o caso de não representante como o caso médio. Logo, essa operação tem complexidade $\Theta(1)$.

Exemplo 3.2: vamos considerar o conjunto $C = \{6, 7, 8, 9\}$ onde 6 é o representante. Vamos supor que queremos remover o elemento 6. Primeiro, verificamos se 6 é o representante do conjunto. Como 6 é o representante, precisamos mover os membros do conjunto para um novo conjunto com um novo representante. Selecionamos o primeiro elemento encontrado no conjunto, digamos 7, como o novo representante. Atualizamos os representantes dos elementos restantes para 7 em `mapping-components` (ou seja, $7 \rightarrow 7$, $8 \rightarrow 7$, $9 \rightarrow 7$). Em seguida, removemos 6 das tabelas de dispersão. Após a exclusão, o conjunto resultante será $\{7, 8, 9\}$ com 7 como o novo representante.

3. Uma union-find alternativa: estrutura baseada em árvores

A estrutura proposta em [Alstrup et al. 2005], aqui chamada de estrutura 2, utiliza uma estrutura em árvore. Cada nó contém as seguintes informações: o valor, que é o vértice do grafo que ele representa; se ele está ocupado ou não, ou seja, se o vértice que ele representa ainda está presente no grafo ou se já foi removido; seu rank, que representa seu nível na árvore; um ponteiro para o nó pai; um ponteiro para o último dos nós filhos e ponteiros para os nós "irmãos", (nós que têm o mesmo pai) anterior e posterior. Iterando recursivamente pelos irmãos anteriores e posteriores é possível ter acesso a todos os filhos de um nó a partir de qualquer nó filho.

Sempre que criamos um novo componente, estamos criando uma árvore com um único nó. Quando fazemos a união de dois componentes, estamos transformando duas árvores em apenas uma árvore. Nessa estrutura a raiz da árvore funciona como o representante do componente conexo. Além dos nós, também criamos uma tabela de dispersão, aqui chamada de `mapping-nodes` que mapeia de um vértice v presente no grafo para um nó n presente na árvore, que é o nó que contém o vértice v.

Por exemplo, para o grafo da Figura 6 teríamos 3 árvores, assim como na Figura 7, representando três componentes conexos. Se seguíssemos o caminho do nó para a raiz a partir de 2 chegaríamos em 1, o mesmo aconteceria com 3. Logo sabemos que 1, 2 e 3 estão no mesmo componente. Os ponteiros para os irmãos e o ponteiro para o último filho são úteis para manipular os nós filhos de um nó, o que é necessário para a operação

de $delete$, como veremos a seguir.

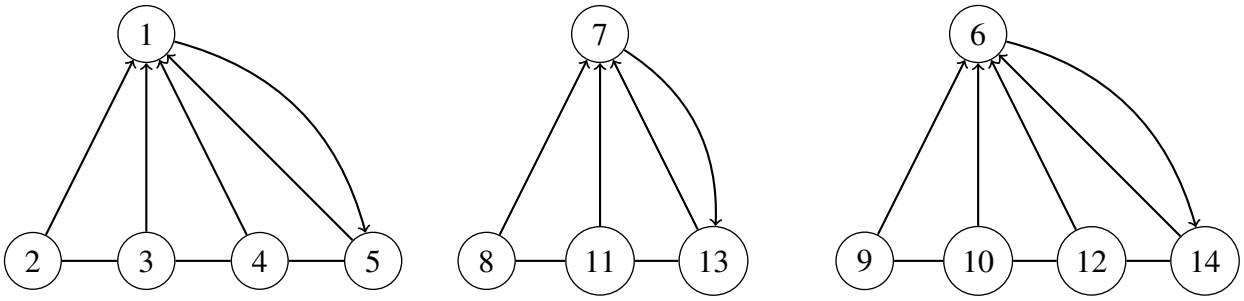

Figura 7. Estrutura em árvore

3.1. Operações na estrutura

Para a operação de $makeset$ de um vértice v, criamos um nó r com $rank$ 0 e valor v. O ponteiro que aponta para o pai desse nó é nulo; também marcamos o nó como ocupado. Após isso inserimos um ponteiro para r em `mapping-nodes` na posição v. Essa operação tem complexidade $\Theta(1)$. Para o $find$, primeiro obtemos o nó com valor v, consultando `mapping-nodes` na posição v. Depois seguimos os ponteiros para o pai do nó recursivamente até a raiz, e então retornamos a raiz como representante do conjunto.

Para fazer a $union$ dos nós a e b, primeiro obtemos os representantes dos conjuntos que contêm a e b usando a função $find$. Vamos chamar esses representantes de `root-a` e `root-b`. Se $rank($`root-a`$) < rank($`root-b`$)$, fazemos um $swap$ entre `root-a` e `root-b`, de forma que $rank($`root-a`$) \geq rank($`root-b`$)$. Então `root-b` passa a ser um filho de `root-a`. Para isso atualizamos o ponteiro pai de `root-b` para `root-a` e atualizamos a lista de filhos de `root-a` para incluir `root-b`. Se $rank($`root-a`$) = rank($`root-b`$)$, o $rank$ de `root-a` é incrementado em 1.

Dado que o $rank$ estritamente aumenta ao seguir ponteiros para o pai, o tempo de uma operação de $find$ aplicada a um elemento em um conjunto A é proporcional ao $rank(A)$. Usando indução, mostramos que $rank(A) < \log_2 |A|$, ou equivalentemente, que $|A| >= 2^{rank(A)}$. Quando A é criado com $makeset(x)$, ele possui $rank = 0$ e $2^0 = 1$ elementos. Se C é o conjunto criado por $union(A, B)$, então $|C| = |A| + |B|$. Se C tiver a mesmo $rank$ de A ou B, terminamos trivialmente. Caso contrário, temos $rank(A) = rank(B) = k$ e $rank(C) = k + 1$, e então $|C| = |A| + |B| >= 2^k + 2^k = 2^{k+1}$. Logo, podemos dizer que a operação $find$ tem complexidade $\mathcal{O}(\log_2 |A|)$. A operação $union$ tem o tempo predominado pela operação $find$, então tem a mesma complexidade.

Exemplo 3.3: vamos considerar a $union$ dos dois primeiros conjuntos do grafo da Figura 6. Vamos chamar os conjuntos de E e F, e seus representantes `root-e` e `root-f`, respectivamente. Como $rank($`root-e`$) \geq rank($`root-f`$)$, `root-f` passa a ser um filho de `root-e`. Nesse caso, também temos que $rank($`root-e`$) = rank($`root-f`$)$, então incrementamos o $rank$ de `root-e` em 1. Após a união, o representante do conjunto unido é 1, e a estrutura de hierarquia reflete que 7 é um filho de 1. Assim, o novo conjunto pode ser representado como $EF = \{1, 2, 3, 4, 5, 7, 8, 11, 13\}$ com 1 como representante.

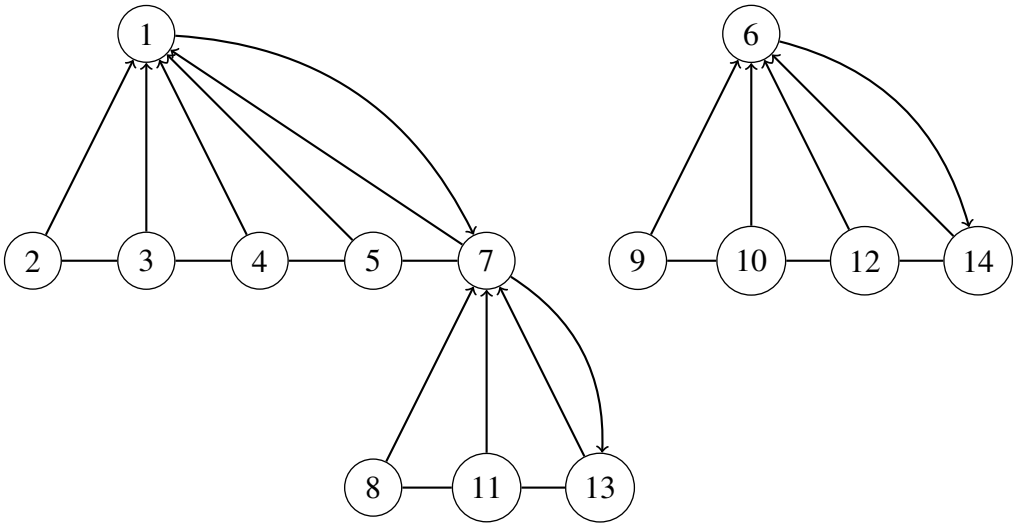

Figura 8. Estrutura em árvore após a $union$ de E e F

Para o $delete$ de um valor x, fazemos o seguinte: obtemos u, o nó que contém x, acessando `mapping-nodes` na posição x. A primeira coisa a fazer é marcar u como desocupado. Se u tiver mais de um filho não precisamos fazer mais nada. Se u tem apenas um filho, a única coisa necessária é fazer o $bypass$ de u. Ou seja, o nó filho de u, (v) passa a ser filho do pai de u, (p). Para isso, definimos p como o ponteiro pai de v, retiramos v da lista de filhos de u e o adicionamos à lista de filhos de p, depois podemos remover u. Se invés disso, u for uma folha, primeiro o retiramos da lista de filhos de p e então removemos u. Se p for agora uma folha, seu $rank$ será reduzido a 0. Se p está desocupado e agora tem apenas um filho, nós fazemos o $bypass$ de p. Essa operação tem complexidade $\mathcal{O}(1)$.

Exemplo 3.4: Vamos considerar remoção de 13 da árvore da Figura 8. Podemos notar que 13 é uma folha, então o retiramos da lista de filhos de 7 e depois o removemos da estrutura. Então temos uma estrutura como a da Figura 9.

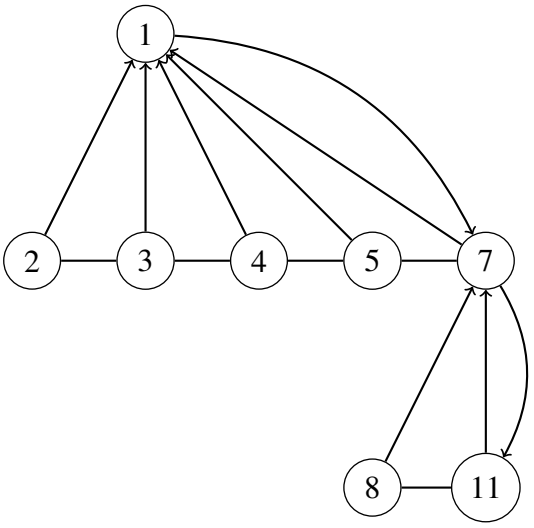

Figura 9. Estrutura em árvore após a remoção de 13

Exemplo 3.5: Agora vamos considerar remoção de 7 da árvore da Figura 9. 7 tem dois filhos, então continua na árvore. A única coisa que temos que fazer é marcá-lo como ocupado e temos uma estrutura como a da Figura 10.

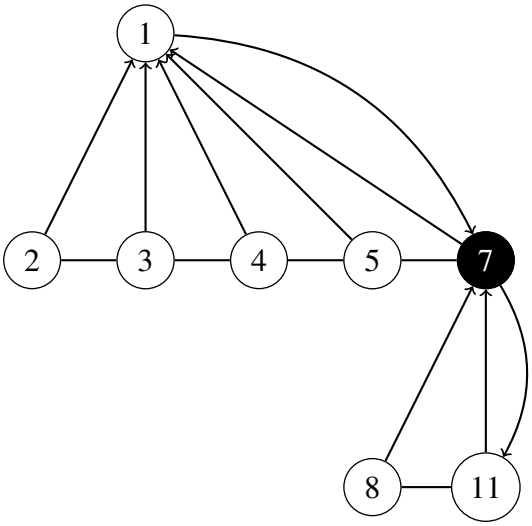

Figura 10. Estrutura em árvore após a remoção de 7

Exemplo 3.6: Agora vamos considerar remoção de 11 da árvore da Figura 10. 11 é uma folha, então assim como 13, nós o removemos da estrutura. Porém agora 7 está desocupado e tem apenas um filho. Então fazemos um $bypass$ e 8 passa a ser filho direto de 1. Sabendo os nós anterior e posterior de 8, eu consigo retirar 8 da lista de irmãos. Sabendo o último nó filho de 1 eu consigo ter acesso a toda a lista de filhos de 1 e inserir 8 nessa lista, resultando em uma estrutura como a da Figura 11.

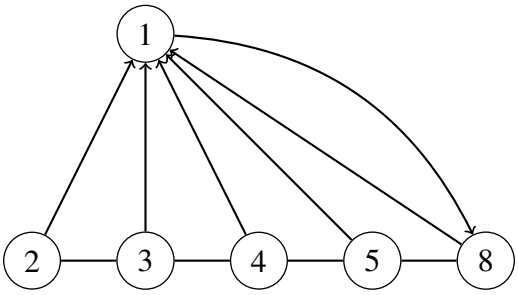

Figura 11. Estrutura em árvore após a remoção de 11

4. Testes

Ambas as implementações foram feitas utilizando C++. A implementação da estrutura 1 foi feita com base na implementação dos autores de [Zheng et al. 2023], que pode ser encontrada em [Zheng 2020]. A implementação da estrutura 2 foi feita com base na descrição fornecida pelos autores de [Alstrup et al. 2005]. As duas implementações criadas para este trabalho podem ser encontradas em [Santos 2024]. Todos os testes foram executados em um computador octacore com processador i7, com 8GB de memória RAM no sistema operacional Ubuntu 20.04. Para testar o desempenho das implementações,

nós criamos grafos e os submetemos às operações disponíveis na union-find, isso é, $makeset$, $union$, $find$ e $delete$. Para os primeiros testes, nós utilizamos os seguintes parâmetros: $N \in \bigcup_{i=1}^{10}\{1000 * i\}$ como o número de nós presentes inicialmente no grafo, $A \in \bigcup_{i=1}^{10}\{1000 * i\}$ o número de arestas e $R \in \bigcup_{i=1}^{10}\{500 * i\}$ o número de nós removidos. Os nós que serão conectados por arestas são escolhidos de forma aleatória, seguindo uma distribuição uniforme. Para cada entrada do programa, definimos quantos nós e arestas queremos no nosso grafo e quantos desses nós serão removidos e construímos a entrada da seguinte maneira:

1. Embaralhamos a ordem original dos nós.
2. Iteramos pelos nós, realizando a operação de $makeset$ para cada nó x e, em seguida, fazemos uma exploração em profundidade pelos nós vizinhos. Para cada vizinho y, realizamos $makeset$ e depois $union$ entre x e y.
3. Após cada exploração, removemos um número aleatório de nós entre 0 e 5.
4. Depois de visitar cada nó, removemos o número restante de nós conforme especificado.

Exemplo 4.1: Vamos considerar um exemplo para um grafo com 5 nós e 4 arestas. Poderíamos acabar com um grafo como o da Figura 12. Suponhamos que queremos remover 2 desses nós durante o processo. Poderíamos ter uma entrada como a da listagem 1.

Listing 1. Entrada exemplo

```
makeset 3
makeset 1
union   3 1
makeset 4
union   1 4
makeset 2
union   4 2
delete  1
delete  2
makeset 0
union   0 4
```

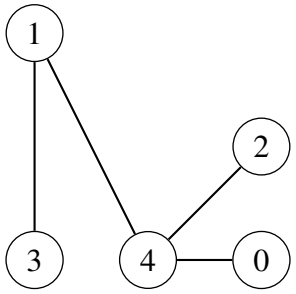

Figura 12. Gráfico da entrada exemplo

Fizemos então três testes. Em cada testes variamos um dos parâmetros, enquanto mantemos os outros dois fixos em seu menor valor, conforme definido por N, A e R. Ou seja, para cada teste, nós tivemos dez entradas. Para cada teste, os aspectos observados foram tempo e memória.

4.1. Tempo

Para observar o tempo consumido, utilizamos a ferramenta *time* [tim 2018] presente no Linux. Podemos observar que a estrutura 2 mantém-se consistentemente mais eficiente em todos os cenários. Embora a implementação da estrutura 1 assegure um tempo constante para a operação de $find$, enquanto a estrutura 2 opera em tempo logarítmico, é importante notar que a $union$ da Estrutura 2 (cujo desempenho é influenciado principalmente pelo $find$) também é realizada em tempo logarítmico. Em contraste, a união proposta em [Zheng et al. 2023] requer iteração por todos os elementos do conjunto, e como já vimos, tem complexidade $\Theta(m \log(n + m - 1))$.

Similarmente, no caso da operação de $delete$, o tempo da estrutura 2 é constante, apenas marcando um elemento como desocupado ou redirecionando um ponteiro. Por outro lado, a estrutura 1 demanda a movimentação de todos os elementos de um conjunto para outro, o que é consideravelmente mais oneroso em termos de tempo de execução.

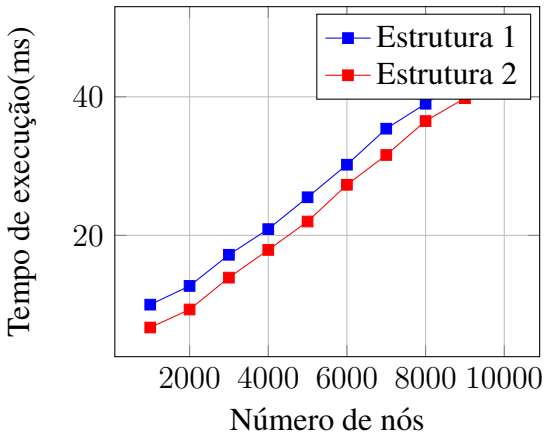

Figura 13. Tempo consumido por operações na union-find variando o número de nós adicionados, com 1000 arestas e 500 nós removidos

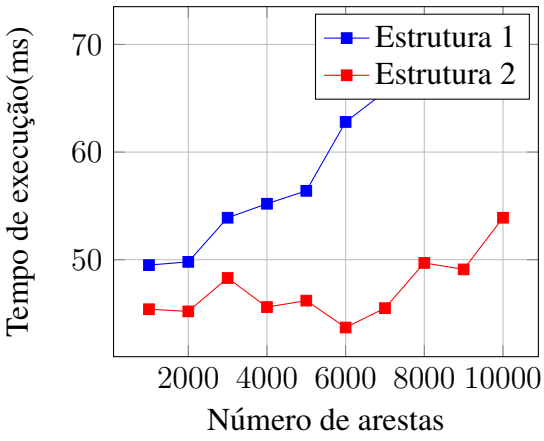

Figura 14. Tempo consumido por operações na union-find variando o número de arestas, com 1000 nós adicionados e 500 nós removidos

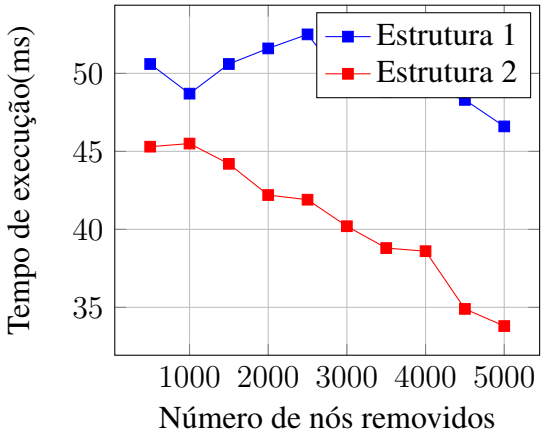

Figura 15. Tempo consumido por operações na union-find variando o número de nós removidos, com 10000 nós adicionados e 1000 arestas

4.2. Memória

Para observar a quantidade de memória utilizada durante a execução do processo, usamos a ferramenta *valgrind* [val 2024]. Assim como para o tempo de execução, notamos que a Estrutura 2 se comporta melhor também no quesito memória utilizada. Isso porque na implementação de [Zheng et al. 2023] é necessário manter uma cópia de cada nó nas duas tabelas de dispersão, além do espaço preciso para armazenar os conjuntos de `component-members`. Na segunda implementação, cada nó está presente somente uma vez na árvore, com toda a travessia sendo feita apenas com ponteiros.

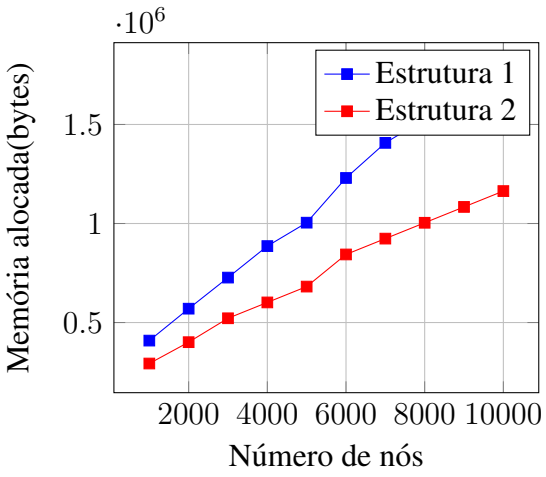

Figura 16. Memória alocada para operações na union-find variando o número de nós adicionados, com 1000 arestas e 500 nós removidos

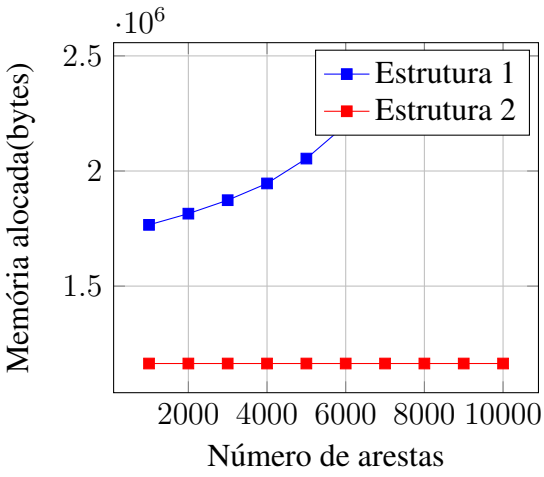

Figura 17. Memória alocada para operações na union-find variando o número de arestas, com 1000 nós adicionados e 500 nós removidos

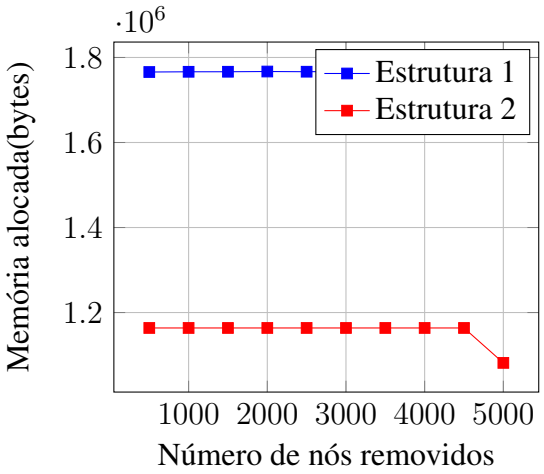

Figura 18. Memória alocada para operações na union-find variando o número de nós removidos, com 1000 nós adicionados e 1000 arestas

Algo importante de ser notado é que o observado durante os testes foram os bytes alocados durante a execução, independente de serem descartados posteriormente ou não. Por isso, para ambas as implementações, o uso de memória se mantém quase constante ao variar o número de nós removidos; já que o número de nós criados é o mesmo, o número de bytes alocados também é o mesmo. A seguir mostramos a memória alocada no momento da saída do programa.

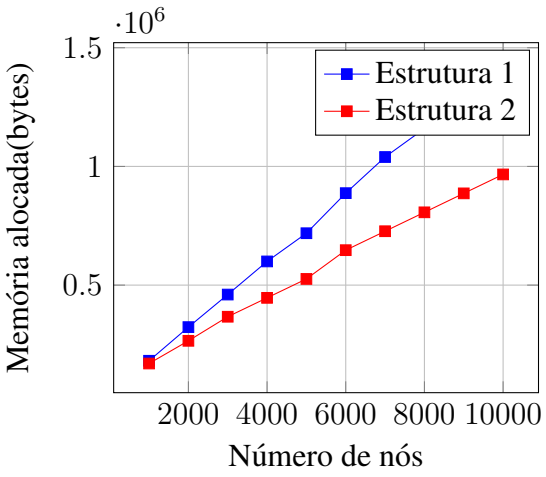

Figura 19. Memória alocada ao fim de operações na union-find variando o número de nós adicionados, com 1000 arestas e 500 nós removidos

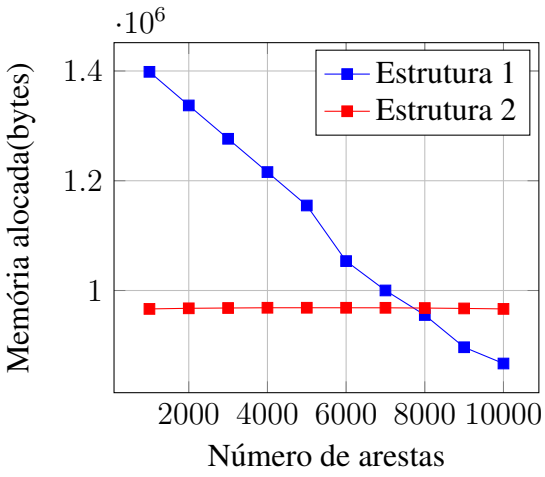

Figura 20. Memória alocada ao fim de operações na union-find variando o número de arestas, com 1000 nós adicionados e 500 nós removidos

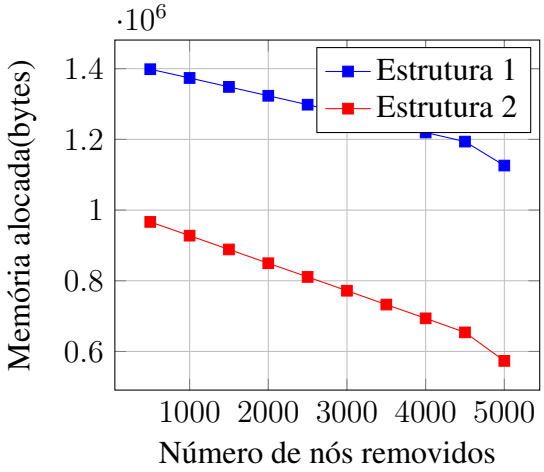

Figura 21. Memória alocada ao fim de operações na union-find variando o número de nós removidos, com 1000 nós adicionados e 1000 arestas

Uma coisa interessante é que para a implementação de [Alstrup et al. 2005] a memória utilizada permanece constante ao variar o número de arestas. Isso ocorre porque não é necessária nenhuma estrutura auxiliar para representar os componentes e a ligação entre os nós além dos ponteiros, que ocupam um espaço muito pequeno na memória. Quanto ao declínio da memória utilizada na saída do programa de [Zheng et al. 2023] quando aumentamos o número de arestas, nossa principal hipótese é de que quanto mais arestas, é preciso alocar mais memória para guardar essas arestas. Porém, conforme mais arestas criadas, maior é a probabilidade de diminuir o número de componentes, assim temos uma perda de memória utilizada. Por exemplo, na Figura 22, temos 20 nós e 10 componentes. Assim, temos que armazenar em `component-members` 10 conjuntos com 2 elementos cada. Enquanto que, na Figura 23, temos mais arestas, consequentemente menos componentes, e assim armazenamos apenas 2 conjuntos em `component-members`. Nos dois casos, temos o mesmo número de nós armazenados, mas no segundo caso, não temos a carga adicional de armazenar tantos conjuntos.

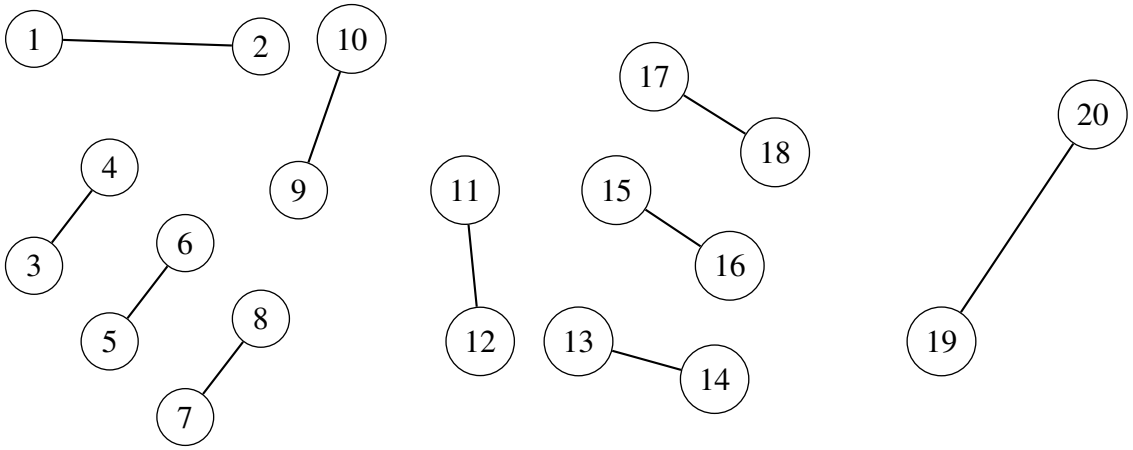

Figura 22. Componentes em pares

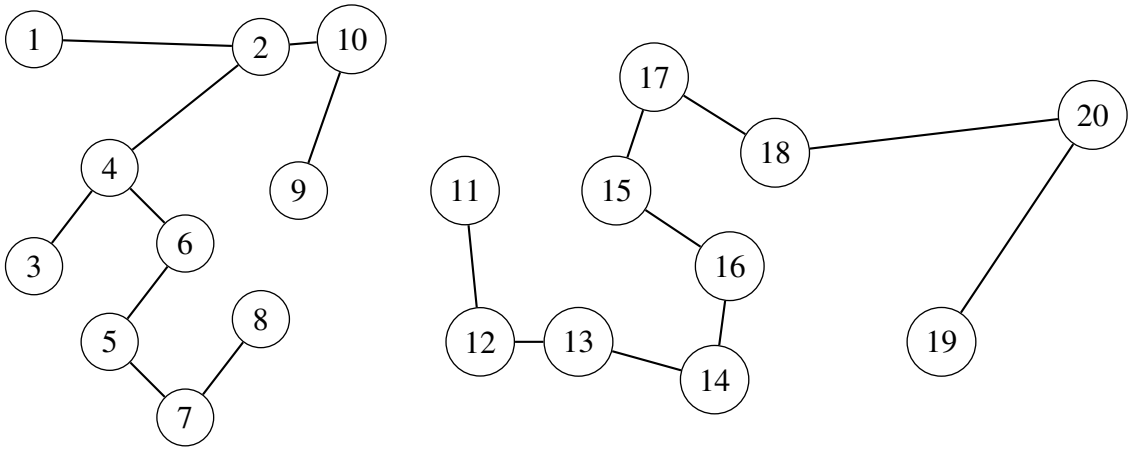

Figura 23. Componentes com muitos elementos

Nós também fizemos alguns testes variando apenas o número de nós e arestas no grafo, sem deletar nenhum nó, para observar como a memória se comporta nesse caso. Podemos ver que nos dois casos, a estrutura 2 se comporta melhor.

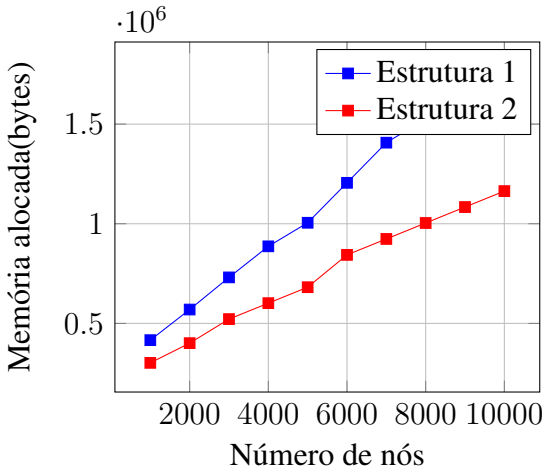

Figura 24. Memória alocada ao fim de operações na union-find variando o número de nós adicionados, com 1000 arestas e sem remover nós

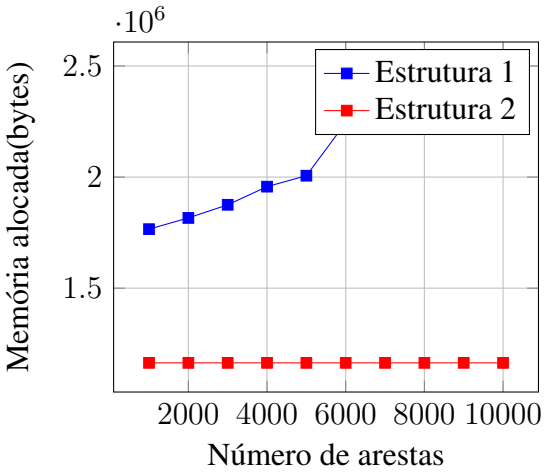

Figura 25. Memória alocada ao fim de operações na union-find variando o número de arestas, com 1000 nós adicionados e sem remover nós

5. Conclusão

Neste estudo, foram exploradas e comparadas duas abordagens distintas para a implementação da operação de exclusão em estruturas de union-find. A análise abrangente considerou aspectos como uso de memória, tempo de execução e aplicabilidade de cada método. Por meio de experimentos e análises teóricas, o objetivo foi aprimorar as implementações de union-find, tornando-as mais versáteis e eficientes para diversas aplicações.

Os experimentos demonstraram que, na maioria dos casos testados, a implementação que fornece a operação de exclusão em tempo constante oferece melhores resultados tanto em termos de tempo de execução quanto de uso de memória. Isso se manteve verdadeiro ao variar o número de nós no grafo, o número de arestas e o número de nós removidos. No entanto, um ponto negativo da implementação de [Alstrup et al. 2005] foi observado ao aumentar o número de arestas. A memória utilizada por essa implementação permanece constante, enquanto na outra implementação a quantidade de memória utilizada diminui conforme o número de arestas aumenta. Isso indica que, para o caso médio, a implementação em tempo constante é a mais indicada. No entanto, em aplicações onde se esperam componentes densos com um número muito alto de arestas, como em sistemas de transportes, redes sociais e grafos da web, a implementação de [Zheng et al. 2023] pode ser mais eficiente.

Para trabalhos futuros, seria interessante aplicar as duas implementações em cenários reais. Por exemplo, a implementação de [Alstrup et al. 2005] poderia ser utilizada no algoritmo de identificação de unidades taxonômicas, conforme proposto por [Zheng et al. 2023], com dados reais de sequências de DNA. Isso proporcionaria uma

visão mais clara das capacidades e limitações de cada implementação em situações práticas.

Referências

[tim 2018] (2018). Gnu time. https://www.gnu.org/software/time/. Acessado em: 19 jun 2024.

[val 2024] (2024). Valgrind documentation. https://valgrind.org/docs/manual/index.html. Acessado em: 19 jun 2024.

[Alstrup et al. 2005] Alstrup, S., Gørtz, I., Rauhe, T., Thorup, M., and Zwick, U. (2005). Union-find with constant time deletions. volume 11, pages 78–89.

[Flick et al. 2017] Flick, P., Jain, C., Pan, T., and Aluru, S. (2017). Reprint of "a parallel connectivity algorithm for de bruijn graphs in metagenomic applications". Parallel Computing, 70:54–65. SC16 Student Cluster Competition Reproducibility Initiative.

[Gardy et al. 2015] Gardy, J., Loman, N. J., and Rambaut, A. (2015). Real-time digital pathogen surveillance — the time is now. Genome Biology, 16(1):155.

[Jain et al. 2015] Jain, M., Fiddles, I. T., Miga, K. H., et al. (2015). Improved data analysis for the minion nanopore sequencer. Nature methods, 12(4):351–356.

[Santos 2024] Santos, H. (2024). union-find. https://github.com/Heitor-Santos/union-find.

[Zheng 2020] Zheng, V. (2020). transclosures. https://github.com/vickymzheng/transclosures.

[Zheng et al. 2023] Zheng, V., Sariyuce, A. E., and Zola, J. (2023). Identifying taxonomic units in metagenomic dna streams on mobile devices. IEEE/ACM Trans Comput Biol Bioinform.