UNIVERSIDADE ESTADUAL DO SUDOESTE DA BAHIA
BACHALERADO EM CIÊNCIA DA COMPUTAÇÃO

GABRIEL SANTOS NERI

ESTRUTURA DE DADOS DISJOINT-SET

conceitos e aplicações

VITÓRIA DA CONQUISTA
2022

GABRIEL SANTOS NERI

ESTRUTURA DE DADOS DISJOINT-SET
conceitos e aplicações

Trabalho de conclusão de curso apresentado à Universidade Estadual do Sudoeste da Bahia, como requisito para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Roque Mendes Prado Trindade

VITÓRIA DA CONQUISTA
2022

GABRIEL SANTOS NERI

ESTRUTURA DE DADOS DISJOINT-SET

conceitos e aplicações

Trabalho de conclusão de curso apresentado à Universidade Estadual do Sudoeste da Bahia, como requisito para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA

____

Prof. Dr. Roque Mendes Prado Trindade - UESB

Orientador

____

Prof. Dr. Hélio Lopes dos Santos - UESB

Avaliador

____

Prof. Dra. Alzira Ferreira da Silva - UESB

Avaliador

AGRADECIMENTOS

Agradeço aos meus pais e minha família por todo o apoio e incentivo, aos meus colegas de turma por compartilhar momentos de diversão e aprendizado durante todo o curso.

Agradeço aos meus companheiros de Maratona de Programação, que estiveram no meu time, ou de alguma forma colaborou com o meu aprendizado em relação a algoritmos e estrutura de dados.

RESUMO

Algoritmos e estrutura de dados são dois temas de muita importância dentro do âmbito da Ciência da Computação. Este estudo tem como objetivo demonstrar como *disjoint-set* é utilizado para resolver problemas de competições de programação. *Disjoint-set* é uma estrutura de dados - também conhecida por *union-find* - que armazena uma coleção de conjuntos disjuntos e possui três operações: construção, união e busca. O presente trabalho discorre sobre essa estrutura, apresentando conceitos fundamentais para seu entendimento, como cada operação funciona e sua devida implementação. Duas otimizações são apresentadas a fim de melhorar a eficiência do algoritmo, otimizando a resolução dos problemas propostos. Por meio do trabalho realizado foi possível observar a importância da estrutura em relação aos problemas presentes em competições de programação.

Palavras-chave: estrutura de dados; algoritmos; *disjoint-set*; *union-find*; programação competitiva; resolução de problemas.

ABSTRACT

Algorithms and data structures are two subjects of great importance in computer science. This study aims to demonstrate how disjoint-set is used to solve programming competition problems. Disjoint-set is a data structure - also known as union-find - which stores a collection of disjoint sets and has three operations: make, union and find. This paper discusses about the structure, presenting fundamental concepts for its understanding, how each operation works and its implementation. Two optimizations are presented in order to improve the efficiency of the algorithm, optimazing the resolution of the proposed problems. Through the work carried out, it was possible to observe the importance of the structure in relation to the problems present in programming competitions.

Key-words: data structures; algorithms; disjoint-set; union-find; competitive programming; problem solving.

SUMÁRIO

1 INTRODUÇÃO **9**
1.1 Objetivos 9
1.1.1 Objetivos específicos 10
1.2 Programação competitiva 10
1.3 Juíz online 11
1.4 Estrutura do texto 11
2 CONCEITOS FUNDAMENTAIS **12**
2.1 Conjunto 12
2.2 Recursão 13
2.3 Complexidade de tempo 15
3 ESTRUTURA DE DADOS DISJOINT-SET **18**
3.1 História 18
3.2 A estrutura 18
3.2.1 Construção 19
3.2.2 União 19
3.2.2.1 Exemplo 19
3.2.3 Busca 20
3.2.3.1 Exemplo 20
4 IMPLEMENTAÇÃO **21**
4.1 Implementação ingênua 21
4.1.1 Construção 21
4.1.2 União 21
4.1.3 Busca 22
4.2 Otimizações 23
4.2.1 Compressão de caminho (path-compression) 23
4.2.2 União por tamanho/classificação (union by size/rank) 24
4.2.2.1 Exemplo de União por Tamanho (union by size) 25
4.2.2.2 Exemplo de União por Classificação (union by rank) 26
4.3 Complexidade 26
5 PROBLEMAS SELECIONADOS **27**
5.1 Fusões 27
5.1.1 Link do problema 27
5.1.2 Descrição geral 27
5.1.3 Descrição da entrada 27
5.1.4 Descrição da saída 28
5.1.5 Exemplo de caso de teste 28
5.1.6 Solução 28

5.2 Famílias de Troia 29
5.2.1 Link do problema 29
5.2.2 Descrição geral 29
5.2.3 Descrição da entrada 29
5.2.4 Descrição da saída 29
5.2.5 Exemplo de caso de teste 29
5.2.6 Solução 29
5.3 Reduzindo detalhes em um mapa 31
5.3.1 Link do problema 31
5.3.2 Descrição geral 31
5.3.3 Descrição da entrada 31
5.3.4 Descrição da saída 31
5.3.5 Exemplo de caso de teste 32
5.3.6 Solução 32
5.4 Distribuição de notícias 34
5.4.1 Link do problema 34
5.4.2 Descrição geral 34
5.4.3 Descrição da entrada 35
5.4.4 Descrição da saída 35
5.4.5 Exemplo de caso de teste 35
5.4.6 Solução 35

6 CONCLUSÃO 37

REFERÊNCIAS 38

1 INTRODUÇÃO

No âmbito da Ciência da Computação, saber resolver problemas é essencial. Desde os mais simples aos mais complexos, encontrar e implementar soluções eficientes para um problema é uma habilidade que o cientista da computação deve desenvolver ao longo de sua formação.

A utilização de algoritmos e estruturas de dados na resolução de problemas dentro da Ciência da Computação é de suma importância. Para o criador do núcleo Linux, Linus Torvalds (LWN, 2006):

> Eu vou, de fato, afirmar que a diferença entre um programador ruim e um bom é se ele considera mais importante seu código ou suas estruturas de dados. Programadores ruins se preocupam com o código. Bons programadores se preocupam com estruturas de dados e suas relações.

Diante das diversas estruturas de dados existentes dentro da computação, a união de conjuntos disjuntos - *disjoint-set union* - é uma das mais versáteis. A discussão sobre *disjoint-set* surgiu em 1964, quando o matemático e cientista da computação Bernard Galler e o cientista da computação Michael Fischer escreveram o artigo "*An Improved Equivalence Algorithm*". Desde então, outros artigos foram publicados e novas descobertas foram feitas.

Estudantes que participam de competições de programação tem como sua principal ferramenta os algoritmos e as estruturas de dados. Nesse âmbito, a estrutura de dados *disjoint-set* pode ser utilizada para resolver muitos dos problemas propostos nessas competições. Essa pesquisa tem caráter bibliográfico e propõe-se a responder a seguinte pergunta: "como a estrutura de dados *disjoint-set* é utilizada para resolver problemas de competições?".

Uma das dificuldades encontradas pelos competidores brasileiros é a falta de material em português sobre os assuntos recorrentes nas competições. Nesse sentido, esse trabalho tem como motivação o desejo de que estruturas de dados em geral e a união de conjuntos disjuntos sejam assuntos mais discutidos dentro do âmbito acadêmico, seja com o objetivo de participar de competições de programação ou para formar melhores cientistas da computação.

1.1 Objetivos

O presente trabalho tem como objetivo geral discorrer sobre a estrutura de dados *disjoint-set union* e aplicá-la na resolução de problemas de maratonas e olimpíadas de programação.

1.1.1 Objetivos específicos

- Entender conceitos fundamentais para o entendimento da estrutura;
- Compreender o funcionamento do *disjoint-set*;
- Implementar o *disjoint-set* com suas otimizações.

1.2 Programação competitiva

Programação competitiva pode ser definida como um "esporte" onde os competidores devem escrever programas para resolver problemas relacionados a lógica, algoritmos e estruturas de dados, dentro de um período limitado de tempo, seguindo restrições de espaço e processamento.

Uma das competições mais antigas é a *International Collegiate Programming Contest* (ICPC) que surgiu na década de 1970. Na edição de 2018, participaram 52.709 estudantes de 3.233 universidades e 110 países.

Grandes empresas como Google e Facebook também incentivam a programação competitiva. O Google é responsável pelo *Google Code Jam*, competição essa que começou em 2003 e já conta com quase 20 edições. Da mesma forma, o Facebook (Meta), organiza o *Facebook Hacker Cup*, que surgiu em 2011. Ambas as *big techs* buscam nas competições, encontrar talentos para um potencial emprego dentro da empresa.

No Brasil, duas competições se destacam: a Maratona de Programação e a Olimpíada Brasileira de Informática (OBI). Ambas realizadas pela Sociedade Brasileira de Computação (SBC).

A Maratona de Programação é um evento realizado pela SBC que surgiu no ano de 1996. Nasceu das competições regionais classificatórias para as finais mundiais do concurso de programação, o *International Collegiate Programming Contest* (ICPC), e é parte da regional sulamericana do concurso. É destinada à alunos de cursos de graduação e início de pós-graduação na área de computação e afins.

A Olimpíada Brasileira de Informática acontece desde 1999, e é destinada a alunos do quarto ano do ensino fundamental até o primeiro ano do ensino superior. É composta por duas modalidades: iniciação e programação. A modalidade programação se divide em quatro níveis (júnior, um, dois e sênior), sendo que os melhores alunos do nível 2 são convidados para um processo seletivo para a Olimpíada Internacional de Informática, onde quatro alunos brasileiros são escolhidos para representar o país na competição.

Segundo Halim e Halim (2013, p.1), “dados os problemas conhecidos da Ciência da Computação, resolva-os o mais rápido possível”. Para os autores do livro mais famoso do assunto, essa é a principal diretiva da programação competitiva.

1.3 Juíz *online*

Um juiz *online*, é um sistema *online* que testa programas em uma competição de programação. Problemas de competições como a Maratona de Programação e a Olimpíada Brasileira de Informática estão disponíveis nos juízes *online*.

Alguns exemplos de online judges são: Codeforces, URI, SPOJ, CodeChef, AtCoder, etc. Dentro dessas plataformas é possível treinar para competições resolvendo problemas e participar de competições *online*. Dado um programa, o juiz online pode responder:

- Aceito: seu programa está correto.
- Resposta errada: seu programa não está correto para todos os casos de testes.
- Tempo limite excedido: seu código demora demais para executar.
- Memória limite excedida: seu código utiliza memória demais.

Todos os problemas que serão apresentados nesse trabalho, podem ser submetidos em juízes *online*.

1.4 Estrutura do texto

No capítulo 2 é apresentado alguns conceitos fundamentais para o entendimento da estrutura de dados *disjoint-set*, entre eles, conjuntos, recursão e complexidade.

Os capítulos 3 e 4 são focados na estrutura. No 3, é apresentado brevemente a história e logo após os conceitos fundamentais, a ideia e as operações da união de conjuntos disjuntos. O capítulo 4 discorre sobre a implementação do algoritmo, partindo de uma implementação ingênua para uma com otimizações.

O capítulo 5 apresenta quatro problemas de competições de programação que são resolvidos utilizando *disjoint-set*. Cada um deles é composto por uma descrição, um exemplo de teste, uma explicação e por fim, a implementação em pseudocódigo.

2 CONCEITOS FUNDAMENTAIS

Alguns conceitos precisam ser compreendidos para que seja possível compreender o funcionamento da estrutura de dados de união de conjuntos disjuntos. Dentre eles: a definição básica de conjunto, recursão e um pouco sobre análise de complexidade.

2.1 Conjunto

Para Rosen (2009, p. 111), um conjunto é uma coleção não ordenada de objetos e esses objetos são chamados de elementos, ou membros, do conjunto. Diz-se que os elementos pertecem ao conjunto. Alguns exemplos de conjuntos são:

- conjunto das vogais
- conjunto dos números pares positivos

Cada membro que entra na formação do conjunto é chamado elemento. Para os exemplos anteriores, os elementos são, respectivamente:

- a, e, i, o, u
- 2, 4, 6, 8, 10, …

Em geral um conjunto é representado por uma letra maiúscula A, B, C, …, e um elemento com uma letra minúscula, a, b, c, x, y, ….

Seja G um conjunto e x um elemento. Se x pertence ao conjunto G, escrevemos:

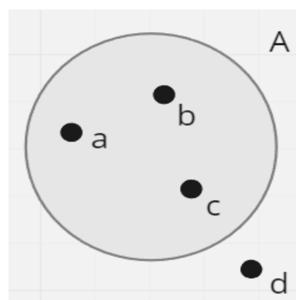
$$\mathbf{x} \in \mathbf{G}$$

Se x não é elemento do conjunto G, escrevemos:

$$\mathbf{x} \notin \mathbf{G}$$

Um conjunto pode ser representado através do diagrama de Euler-Veen.

Figura 1 - Diagrama de Euler-Veen



Fonte: Autoria própria, 2022.

No diagrama da figura 1, tem-se que:

$$\mathbf{a} \in \mathbf{A},\ \mathbf{b} \in \mathbf{A},\ \mathbf{c} \in \mathbf{A} \text{ e } \mathbf{d} \notin \mathbf{A}.$$

Algumas operações podem ser realizadas em conjuntos, para entendermos a estrutura *disjoint-set*, é necessário compreender as operações de união e interseção. As definições a

seguir foram retiradas do livro *Matemática Discreta e Suas Aplicações* do matemático Kenneth H. Rosen.

Definição 1: Sejam A e B conjuntos. A união dos conjuntos A e B, indicada por A ∪ B, é o conjunto que contém aqueles elementos que estão em A ou em B, ou em ambos.

Um elemento *x* pertence à união dos conjuntos A e B se e somente se *x* pertencer a A ou *x* pertencer a B. Então, tem-se que:

$$\mathbf{A \cup B = \{x \mid x \in A \lor x \in B\}}$$

Definição 2: Sejam A e B conjuntos. A interseção dos conjuntos A e B, indicada por A ∩ B, é conjunto que contém aqueles elementos que estão em A e em B, simultaneamente.

Um elemento *x* pertence à interseção dos conjuntos A e B se e somente se *x* pertencer a A e *x* pertencer a B. Então, tem-se que:

$$\mathbf{A \cap B = \{x \mid x \in A \land x \in B\}}$$

Definição 3: Dois conjuntos são chamados de disjuntos se sua interseção é um conjunto vazio.

2.2 Recursão

Uma função é dita recursiva, se ela é definida em termos de si mesma. Uma função recursiva pode ser uma ferramenta muito poderosa quando escreve-se um algoritmo. Na computação, uma função pode ser dita recursiva se apresenta os seguintes comportamentos:

- caso base: um cenário que não usa recursão para achar a resposta
- um passo recursivo: uma série de regras que reduz todos os casos para o caso base

O exemplo mais clássico de recursão é o da sequência de Fibonacci, cuja definição é a seguinte:

$$Fib(n) = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ Fib(n-1) + Fib(n-2) & \text{se } n >= 2 \end{cases}$$

É possível observar que para *Fib(n)* ser definida é preciso saber *Fib(n - 1)* e *Fib(n - 2)*, ou seja, a sequência de Fibonacci é definida em termos de si mesma. O pseudocódigo para a sequência é descrito abaixo:

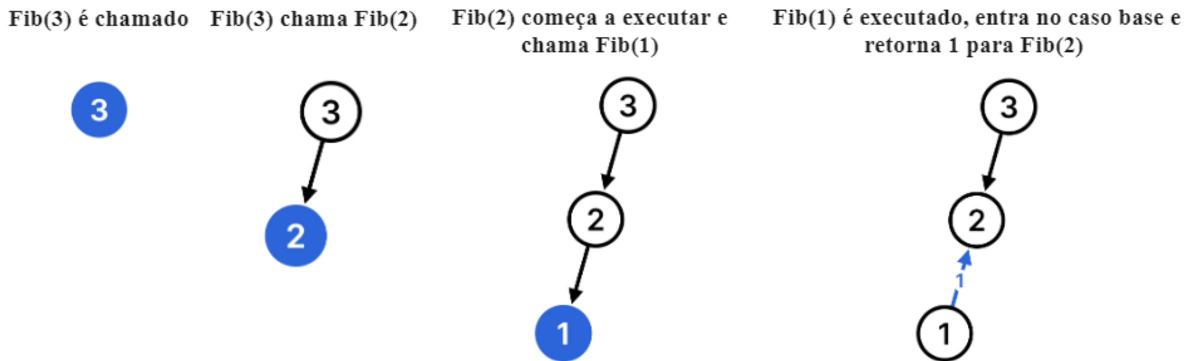
Algoritmo Sequência de Fibonacci

```
1: função FIB(n)
2:     se n <= 1 então                              ▷ caso base
3:         devolve n
4:     devolve FIB(n − 1) + FIB(n − 2)              ▷ passo recursivo
```

É sabido que a sequência de Fibonacci é como segue: 0, 1, 1, 2, 3, 5, 8, 13, …. Como para saber o *Fib(n)*, é necessário saber *Fib(n - 1)* e *Fib(n - 2)*, é definido como caso base *Fib(0) = 0* e *Fib(1) = 1*. Para o passo recursivo, é utilizado a fórmula já definida acima.

Para facilitar o entendimento de como uma função recursiva funciona, a figura abaixo ilustra a chamada de *Fib(3)* através de uma árvore de execução:

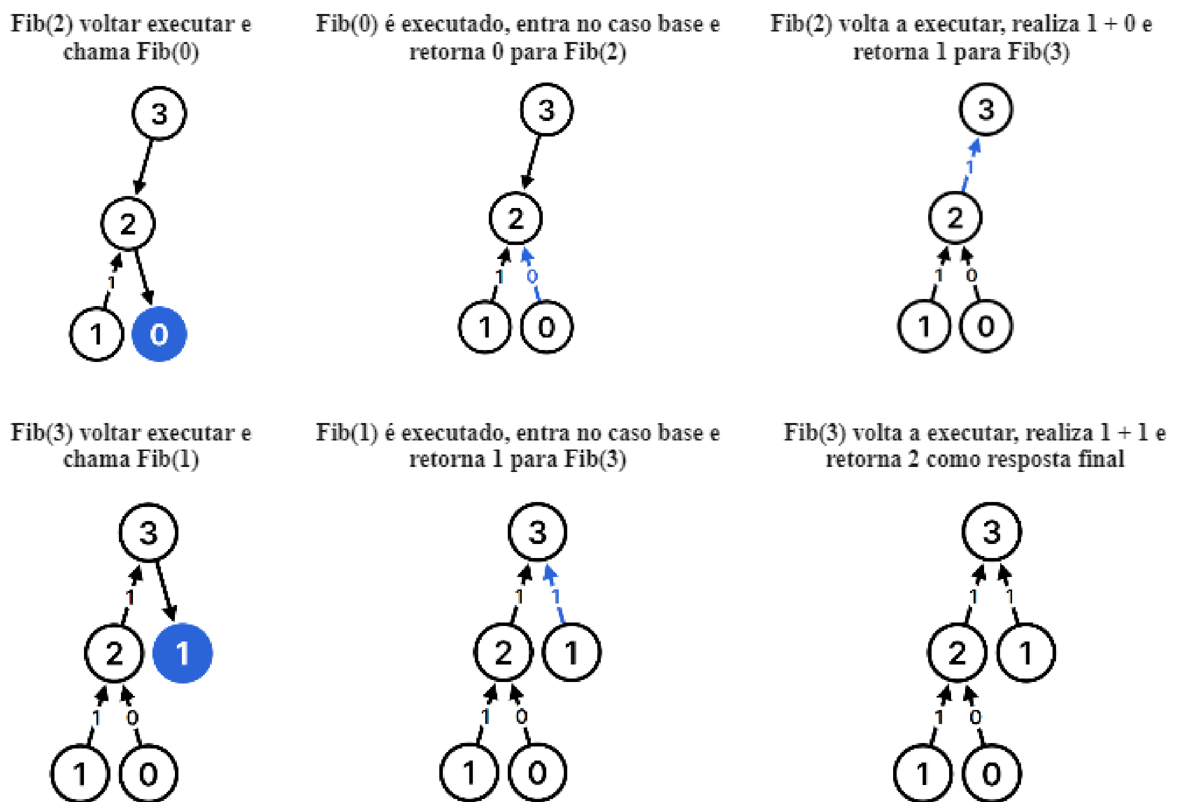
Figura 2 - Execução de Fib(3)



Fonte: Autoria própria, 2022.

Na figura 2, é possível observar que a recursão executa todas as chamadas de Fib(n - 1). A continuação é ilustrada na figura 3.

Figura 3 - Continuação da execução de Fib(3)



Fonte: Autoria própria, 2022.

2.3 Complexidade de tempo

Quando um problema é apresentado, pode ser fácil achar uma solução que resolva-o de forma lenta, na maioria dos casos, a maior dificuldade é encontrar uma solução eficiente.

Na programação competitiva, para resolver um problema de uma competição é necessário que o algoritmo seja rápido o suficiente para passar nos testes dentro do tempo estipulado. Se a solução for lenta, ela não será aceita.

A complexidade de tempo de um algoritmo estima quanto tempo ele irá utilizar para determinada entrada, ou seja, a eficiência de um algoritmo é representada por uma função onde o parâmetro é o tamanho da entrada.

Uma notação utilizada para representar esse tempo é a *big O*. A definição da notação O grande, segundo Rosen (2009, p. 180) é: “Sejam *f* e *g* as funções do conjunto de números inteiros ou dos números reais para o conjunto dos números reais. Dizemos que $f(x)$ é $O(g(x))$ se houver constantes C e k, tal que $|f(x)| \leq C\,|g(x)|$ sempre que $x > k$”.

Nessa notação, se olharmos para a função $an^2 + bn + c$ onde a, b e c são constantes, a complexidade de tempo seria $O(n^2)$, ou seja, quando a notação *big O* é utilizada, as constantes são ignoradas e só é considerado a maior potência de n.

Dentro do campo de complexidade algoritmos, também se faz presente a análise amortizada. Em um algoritmo em que o consumo de tempo de cada execução depende das execuções anteriores e de que cada execução lenta e precedida por execuções muito rápidas é utilizada a análise amortizada para fazer estimativas do consumo de tempo. Para Cormen, Leiserson, Riverst e Stein (2009, p. 451), na análise amortizada é analisado o tempo requerido para performar uma sequência de operações com estrutura de dados sobre todas as operações executadas. Abaixo, alguns exemplos de código e suas complexidade de tempo.

Exemplo 1: $O(n)$. O algoritmo faz n iterações.

Algoritmo Exemplo 1

```
1: x ←1
2: para i de 1 até n faça
3:     x ← i
```

Exemplo 2: $O(n^2)$. O algoritmo faz n iterações no primeiro laço e n iterações no segundo. Logo, multiplica-se n por n.

Algoritmo Exemplo 2

```
1: x ←1
2: para i de 1 até n faça
3:     para j de 1 até n faça
4:         x ← i + j
```

Exemplo 3: *O(n * m)*. O algoritmo faz n iterações no primeiro laço e m iterações no segundo. Logo, multiplica-se n por m.

Algoritmo Exemplo 3

```
1: x ←1
2: para i de 1 até n faça
3:     para j de 1 até m faça
4:         x ← i + j
```

Exemplo 4: *O(log n)*. Considerando n como o tamanho do array e observando seu tamanho a cada iteração, temos que: 1° iteração, tamanho do array = n; 2° iteração, tamanho do array = $\frac{n}{2}$; 3º iteração, tamanho do array = $\frac{\frac{n}{2}}{2} = \frac{n}{2^2}$; k-th iteração, tamanho do array = $\frac{n}{2^k}$.

Depois de k iterações, o tamanho do array se torna 1, ou seja: $\frac{n}{2^k} = 1$, logo $n = 2^k$.

Aplicando o $\log_2$ dos dois lados, obtém-se: $\log_2 n = \log_2 2^k$, sendo assim, $\log_2 n = k * \log_2 2$.

Como $\log_a a = 1$, o resultado é: $k = log_2(n)$.

Algoritmo Exemplo 4

```
1: função BuscaBinaria(arr, n, x)
2:     esq ← 1
3:     dir ← n
4:     enquanto esq <= dir
5:         meio ← (esq+dir)/2
6:         se arr[meio] == x então
7:             devolve 1
8:         senão se arr[meio] < x então
9:             esq ← meio + 1
10:        senão
11:            dir ← meio − 1
12:    devolve −1
```

Exemplo 5: *O(*$\sqrt{n}$*)*. O algoritmo faz $\sqrt{n}$ iterações.

Algoritmo Exemplo 5

```
1: função Primo(n)
2:     se n <= 1 então
3:         devolve False
4:     para i de 2 até √n faça
5:         se n mod i == 0 então
6:             devolve False
7:     devolve True
```

Calculando a complexidade de tempo de um algoritmo, é possível saber se ele é rápido o suficiente para resolver um problema, antes mesmo de codificar.

A figura 4, apresenta o tamanho de entrada *n* e qual deve ser a complexidade máxima para que um algoritmo seja aceito, assumindo que a CPU consegue computar 100 megabytes itens em 3 segundos.

Figura 4 - Tabela de complexidades aceitas em relação ao tamanho da entrada

n	Worst AC Algorithm	Comment
$\leq [10..11]$	$O(n!), O(n^6)$	e.g. Enumerating permutations (Section 3.2)
$\leq [15..18]$	$O(2^n \times n^2)$	e.g. DP TSP (Section 3.5.2)
$\leq [18..22]$	$O(2^n \times n)$	e.g. DP with bitmask technique (Section 8.3.1)
≤ 100	$O(n^4)$	e.g. DP with 3 dimensions + $O(n)$ loop, ${}_nC_{k=4}$
≤ 400	$O(n^3)$	e.g. Floyd Warshall's (Section 4.5)
$\leq 2K$	$O(n^2 \log_2 n)$	e.g. 2-nested loops + a tree-related DS (Section 2.3)
$\leq 10K$	$O(n^2)$	e.g. Bubble/Selection/Insertion Sort (Section 2.2)
$\leq 1M$	$O(n \log_2 n)$	e.g. Merge Sort, building Segment Tree (Section 2.3)
$\leq 100M$	$O(n), O(\log_2 n), O(1)$	Most contest problem has $n \leq 1M$ (I/O bottleneck)

Fonte: Competitive Programming 3, 2013.

3 ESTRUTURA DE DADOS *DISJOINT-SET*

Neste capítulo são apresentados os conceitos fundamentais para o entendimento da estrutura *disjoint-set*. As definições foram retiradas dos livros *Introduction to Algoritms 3, Competitive Programming 3* e do website *CP Algorithms.*

3.1 História

A discussão sobre *disjoint-set* surgiu em 1964, com o artigo "*An Improved Equivalence Algorithm*", escrito pelo matemático e cientista da computação Bernard Galler e o cientista da computação Michael Fischer.

Nove anos depois, em 1973, a complexidade do algoritmo foi limitada para $O(log^*(n))$ no estudo "*Set Merging Algorithms*" publicado pelos autores John Edward Hopcroft e Jeffrey David Ullman.

Em 1975, Robert Tarjan provou que o limite superior do algoritmo era a função inversa de Ackermann em "*Efficiency of a Good But Not Linear Set Union Algorithm*". Michael Fredman e Michael Sacks mostraram que a complexidade por operação do algoritmo era a função inversa de Ackermann amortizada no artigo "*The Cell Probe Complexity of Dynamic Data Structures*".

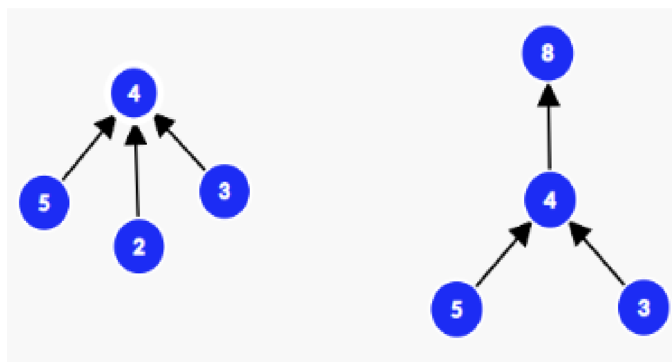
3.2 A estrutura

A estrutura de dados de união de conjuntos disjuntos (*disjoint-set*) mantém uma coleção $C = C_1, C_2, \ldots, C_k$ de conjuntos disjuntos, onde cada um tem seu próprio representante.

Os conjuntos são representados em forma de árvore, onde a raiz da mesma é chamada de representante do conjunto.

Veja o exemplo abaixo:

Figura 5 - Representação de conjuntos como árvore



Fonte: Autoria própria, 2022.

Observando a figura 5, na árvore da esquerda, os elementos do conjunto são {4, 5, 2, 3} e o representante do mesmo é o elemento {4}, raiz da árvore. Do mesmo modo, na árvore da direita, os elementos do conjunto são {8, 4, 5, 3} e o representante é o elemento {8}.

Utilizando a estrutura de dados disjoint-set é possível unir dois conjuntos disjuntos e determinar a qual conjunto um elemento pertence. Nesse sentido, a estrutura se baseia em três operações: construção, união e busca. Devido as suas operações a estrutura de dados *disjoint-set* é também conhecida por *union-find*.

3.2.1 Construção

A operação de construção recebe um elemento qualquer x e cria um novo conjunto, cujo o único membro é x, sendo assim, x é o próprio representante do conjunto. É necessário que x não esteja em nenhum outro conjunto já existente.

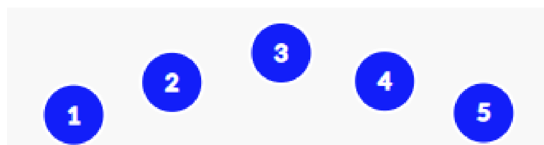
3.2.2 União

A união recebe dois parâmetros a e b e une os conjuntos S_a e S_b em um único. Só é possível realizar *union_sets(a, b)* se $S_a \neq S_b$.

3.2.2.1 Exemplo

Para ilustrar o funcionamento da operação *union_sets(a, b)*, inicialmente, temos 5 conjuntos disjuntos: {1}, {2}, {3}, {4} e {5}, onde cada um tem si próprio como representante. As árvores são como na figura abaixo:

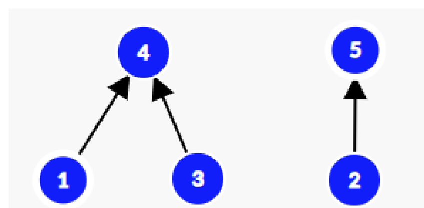
Figura 6 - Conjuntos disjuntos



Fonte: Autoria própria, 2022.

Ao realizar as operações *union_sets(1, 4), union_sets(2, 5)* e *union_sets(3, 4)*, o resultado obtido é mostrado na figura:

Figura 7 - Conjuntos após operações de união



Fonte: Autoria própria, 2022.

Quando uma operação de união é feita, o elemento que se tornará o novo representante do conjunto é definido de acordo a implementação do algoritmo. Esse tópico é discutido no próximo capítulo.

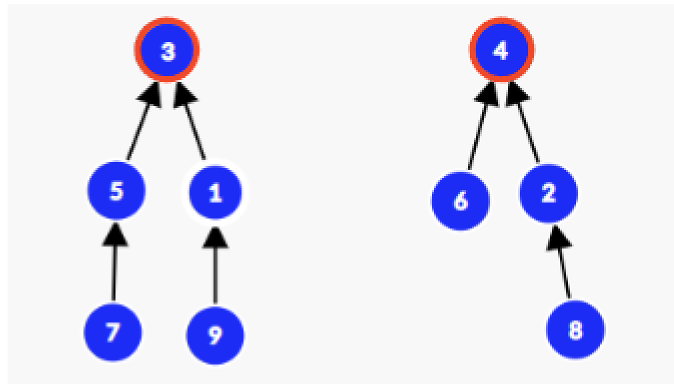
3.2.3 Busca

A operação busca recebe um parâmetro qualquer x e retorna o representante do conjunto S_x como resposta.

3.2.3.1 Exemplo

A imagem abaixo, apresenta a representação de dois conjuntos: {1, 3, 5, 7, 9} e {2, 4, 6, 8}:

Figura 8 - Dois conjuntos com seus representantes marcados



Fonte: Autoria própria, 2022.

No primeiro conjunto da figura 8, pode-se observar que o elemento representante é o 3, ou seja, para qualquer operação *find_sets(x)*, onde $x <= 9$ e x é ímpar, a resposta será 3.

Do mesmo modo, no segundo conjunto, para qualquer operação *find_sets(x)*, onde $x <= 8$ e x é par, a resposta será 4, pois é o nó representante do conjunto.

4 IMPLEMENTAÇÃO

Este capítulo tem como objetivo apresentar as implementações das operações da estrutura de dados *disjoint-set* na linguagem de programação C++.

Os conjuntos são representados em forma de árvore. Um array *pai* é utilizado para guardar as informações de um determinado conjunto.

Um elemento na posição a do array *pai* com valor b quer dizer que o ancestral de a na árvore é b.

4.1 Implementação ingênua

4.1.1 Construção

Para criação de um novo conjunto, precisamos chamar a função *construir*. Essa função recebe como parâmetro *x* e é criado uma árvore cujo o representante de *x* é ele próprio.

Algoritmo Construir

```
1: void CONSTRUIR(int x) {
2:     pai[x] = x
3: }
```

4.1.2 União

A função recebe dois parâmetros a e b e tem como objetivo unir os conjuntos S_a e S_b. Inicialmente é chamada a função *busca* para sabermos à que conjunto a e b pertecem. Essas informações são guardadas nas variáveis $repr_a$ e $repr_b$, ou seja, $repr_a$ guarda quem é o representante de S_a e $repr_b$ o representante de S_b.

Caso $repr_a = repr_b$ os dois elementos já estão no mesmo conjunto.

Se $repr_a \neq repr_b$ dizemos que $pai[repr_b] = repr_a$, o que significa que o ancestral de $repr_b$ agora é $repr_a$, desse modo, todo o conjunto que o elemento b estava contido se uniu com o conjunto S_a.

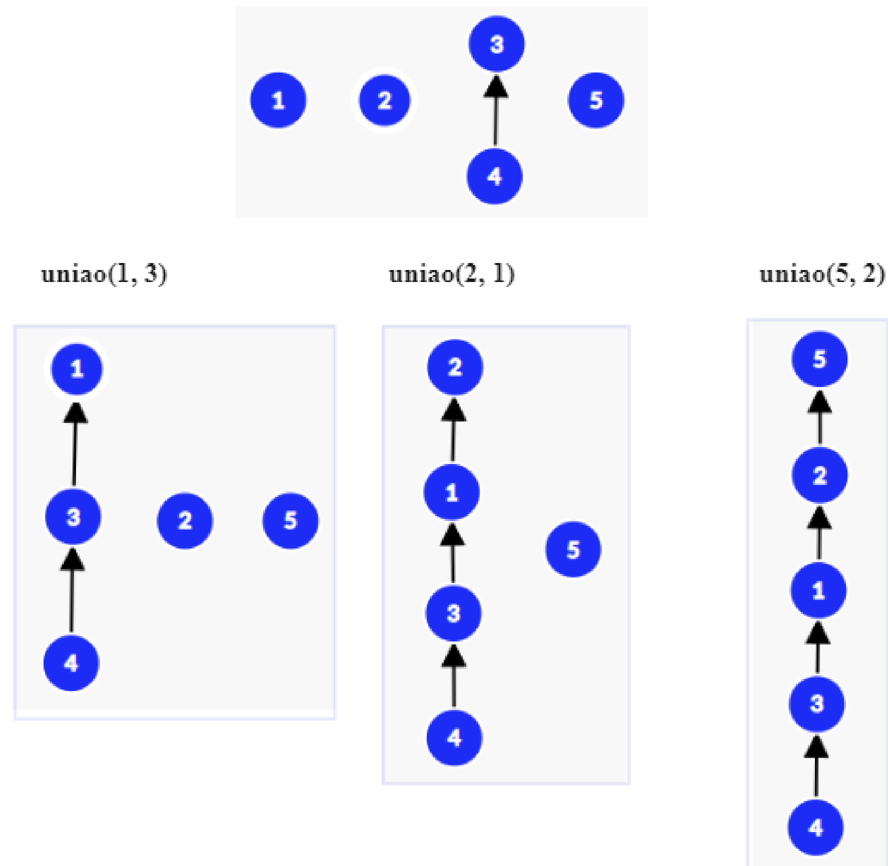
Algoritmo União - Implementação ingênua

```
1: void UNIAO(int a, int b) {
2:     repr_a = BUSCA(a);
3:     repr_b = BUSCA(b);
4:     if (repr_a ! = repr_b) {
5:         pai[repr_b] = repr_a;
6:     }
7: }
```

O problema dessa implementação é que, quando é feita a união do conjunto S_a com S_b, o conjunto resultante é sempre $S_a = S_a \cup S_b$. Isso pode gerar árvores desbalanceadas, como no exemplo abaixo:

Figura 9 - Árvores após operações de união

Conjuntos iniciais: {1}, {2}, {3, 4} e {5}



Fonte: Autoria própria, 2022.

É possível observar na figura 9, que a árvore resultante após as operações de união ficou desbalanceada devido ao conjunto resultante da união sempre ser atribuído a S_a na implementação ingênua.

4.1.3 Busca

Um parâmetro x é passado para a função e a mesma devolve o representante do conjunto que x está contido.

Algoritmo Busca - Implementação ingênua

```
1: int BUSCA(int x) {
2:     if (pai[x] == x) {
3:         return x;
4:     }
5:     return BUSCA(pai[x]);
6: }
```

Para encontrar o representante de uma árvore, a função busca tem como objetivo subir até a raiz da árvore. Nesse sentido, para cada vértice x, se ele não é a raiz, a função é chamada para o ancestral de x. Essa ideia é facilmente implementada recursivamente, tópico visto no capítulo 2.

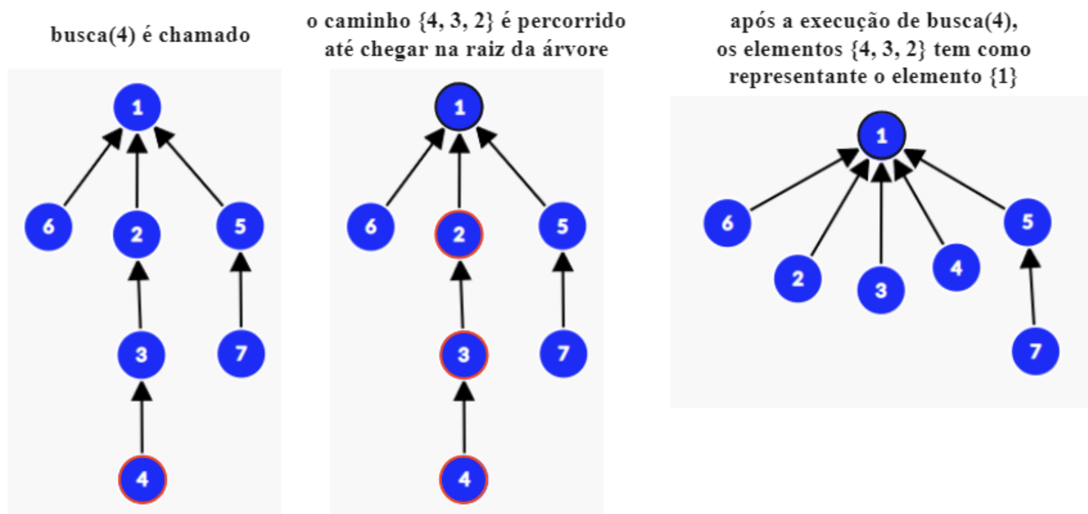
No entanto, essa implementação não é eficiente. Tomando como exemplo o conjunto resultante da Figura 8, se deseja-se saber qual o representante do elemento 4 é realizada a chamada *busca(4)*, cuja a complexidade para tal é de *O(n)*, onde n é o número de vértices do conjunto.

4.2 Otimizações

4.2.1 Compressão de caminho (path-compression)

A compressão de caminho tem como objetivo otimizar a função de busca. Quando a função é chamada para um vértice a deseja-se encontrar o representante $repr_a$ do conjunto em que a está contido. Com essa otimização, todo o caminho de a até $repr_a$ recebe como novo representante $repr_a$. O exemplo abaixo, ilustra essa ideia:

Figura 10 - Árvore após compressão de caminho



Fonte: Autoria própria, 2022.

A única mudança na implementação é associar *pai[x]* com o resultado da chamada de busca para *pai[x]*. Essa associação é feita no desempilhamento da recursão.

Algoritmo Busca - Compressão de caminho

```
1: int Busca(x) {
2:     if (pai[x] == x) {
3:         return x;
4:     }
5:     return pai[x] = Busca(pai[x]);
6: }
```

Na figura 10, o caminho {4, 3, 2} é percorrido através da chamada do passo recursivo, até a função encontrar o caso base, que é {1}, pois ele é o representante do conjunto. No desempilhamento da recursão todos os elementos que foram executados no passo recursivo recebem {1} como seu novo representante.

4.2.2 União por tamanho/classificação (union by size/rank)

Na união por tamanho, é utilizado um array para guardar o tamanho (quantidade de nós) de uma árvore. De outra forma, a união por classificação utiliza um array para guardar a classificação (profundidade) de uma árvore. Em ambos os casos, a árvore com maior tamanho/classificação é utilizada como representante após a união.

No código da união por tamanho, é necessário iniciar o array *tam* com 1, que é o tamanho inicial da árvore.

Algoritmo Disjoint-set Union - União por tamanho

```
1: void Construir(int x) {
2:     pai[x] = x;
3:     tam[x] = 1;
4: }
5:
6: void Uniao(int a, int b) {
7:     repr_a = Busca(a);
8:     repr_b = Busca(b);
9:     if (repr_a != repr_b) {
10:        if (tam[repr_a] > tam[repr_b]) {
11:            pai[repr_b] = repr_a;
12:            tam[repr_a] = tam[repr_a] + tam[repr_b];
13:        } else {
14:            pai[repr_a] = repr_b;
15:            tam[repr_b] = tam[repr_b] + tam[repr_a];
16:        }
17:    }
18: }
```

Na codificação da união por classificação, o array *class* é iniciado com 0, que é a profundidade inicial da árvore. É importante lembrar que, quando utilizado a compressão de caminho, a profundidade da árvore fica menor, portanto, o array *class* se torna um limite superior da profundidade da árvore.

Algoritmo Disjoint-set Union - União por classificação

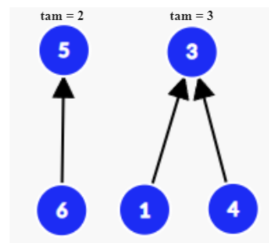
```
1:  void CONSTRUIR(int x) {
2:      pai[x] = x;
3:      class[x] = 0;
4:  }
5:
6:  void UNIAO(int a, int b) {
7:      repr_a = BUSCA(a);
8:      repr_b = BUSCA(b);
9:      if (repr_a != repr_b) {
10:         if (class[repr_a] > class[repr_b]) {
11:             pai[repr_b] = repr_a;
12:         } else {
13:             pai[repr_a] = repr_b;
14:             if (class[repr_a] == class[repr_b]) {
15:                 class[repr_b] = class[repr_b] + 1;
16:             }
17:         }
18:     }
19: }
```

Ambas as heurísticas de união, tem como objetivo melhorar o desempenho da busca.

4.2.2.1 Exemplo de União por Tamanho (union by size)

Dados dois conjuntos {5, 6} e {1, 4, 3}, o tamanho do primeiro conjunto é 2 e o do segundo 3.

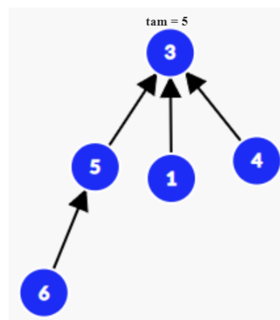
Figura 11 - Conjuntos iniciais - união por tamanho



Fonte: Autoria própria, 2022.

Após realizar a operação *union_by_size(5, 3)*, o resultado obtido é o mostrado abaixo:

Figura 12 - Conjunto de tamanho 5 após união

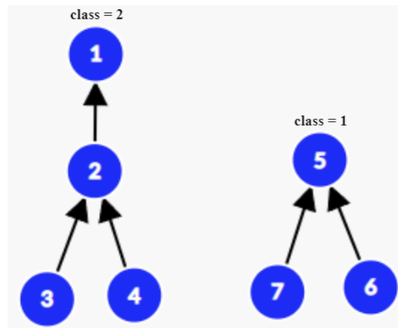


Fonte: Autoria própria, 2022.

4.2.2.2 Exemplo de União por Classificação (union by rank)

Dado os dois conjuntos {1, 2, 3, 4} e {5, 6, 7}, a profundidade do primeiro conjunto é 2 e do segundo 1.

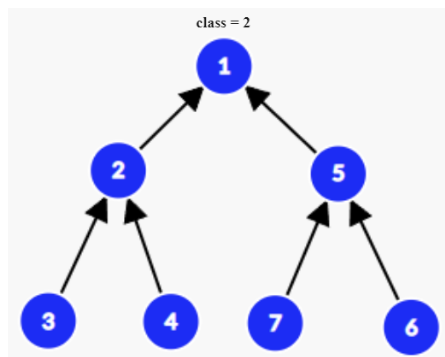
Figura 13 - Conjunto iniciais - união por classificação



Fonte: Autoria própria, 2022.

Após realizar a operação *union_by_rank(1, 5)*, a árvore resultante é mostrada na figura abaixo:

Figura 14 - Árvore após a união dos conjuntos



Fonte: Autoria própria, 2022.

4.3 Complexidade

As duas heurísticas de união apresentadas são semelhantes em termos de complexidade de espaço e tempo, ambas, executam em *O(log n)* onde *n* é o número de elementos, ou seja, pode-se utilizar tanto a união por tamanho, quanto por classificação.

A função *busca* implementada com a técnica de compressão de caminho tem complexidade de *O(log n)* por cada execução.

Quando utilizado a compressão de caminho juntamente com a união por tamanho ou classificação, o tempo para cada consulta é quase constante.

Cormen, Leiserson, Riverst e Stein (2009, p. 573) provam que a complexidade amortizada final é *O(a(n))*, onde *a(n)* é o inverso da função de Ackermann, tal qual cresce muito lentamente. Para um $n < 10^{600}$ esse número não ultrapassa 4.

5 PROBLEMAS SELECIONADOS

Neste capítulo são apresentados alguns problemas de programação competitiva que a solução envolve a estrutura de dados disjoint-set. Para cada problema é apresentado uma descrição geral, com objetivo de resumir a proposta do problema, a descrição da entrada e saída, exemplos de caso de teste e a solução do mesmo.

Além disso, todos os problemas apresentados estão em algum juiz online, podendo ser: SPOJ, Codeforces, AtCoder, NepsAcademy, PratiqueOBI, etc.

Os códigos apresentados no trabalho, estão em pseudocódigo, mas as soluções podem ser implementadas em uma linguagem de programação (C++, Java, Python, etc) e serem submetidas nos juízes online.

5.1 Fusões

Problema retirado da segunda fase da modalidade programação nível 1 da Olimpíada Brasileira de Informática (OBI) do ano de 2010.

5.1.1 Link do problema

https://olimpiada.ic.unicamp.br/pratique/p1/2010/f2/fusoes/

5.1.2 Descrição geral

Bancos estão cada vez mais informatizados, sendo assim, cada banco possui um identificador. Fusão é o nome dado a compra de um banco por outro, nesse caso, ambos bancos se tornam um só, mas podem ser referenciados por seus antigos identificadores.

Escreva um programa que, dada uma série de fusões entre bancos, responde a várias consultas perguntando se dois códigos bancários (identificadores) se referem ao mesmo banco.

5.1.3 Descrição da entrada

A primeira linha apresenta dois inteiros N (número de bancos) e K (número de operações efetuadas), onde $1 <= N <= 10^5$ e $1 <= K <= 10^5$. Os códigos de cada um dos N bancos, inicialmente, são inteiros de 1 até N. Cada uma das K linhas seguintes descreve uma fusão ou uma consulta.

Fusões são descritas com uma linha que começa com o caractere ‘F’, seguida por dois códigos bancários, que se referem aos dois bancos que estão sofrendo a fusão. Uma consulta é

descrita como uma linha que começa com o caractere ‘C’, seguida por dois códigos a serem consultados. Os códigos são sempre distintos.

É garantido que as fusões são sempre realizadas entre bancos diferentes e todos os códigos fornecidos na entrada são válidos.

5.1.4 Descrição da saída

O programa deve imprimir uma linha para cada consulta. Se os códigos se referem ao mesmo banco, a saída deve ser o caractere ‘S’, caso contrário, imprimir o caractere ‘N’.

5.1.5 Exemplo de caso de teste

ENTRADA			**SAÍDA**
4	5		
F	1	2	
F	2	3	
C	1	3	S
F	2	4	
C	1	4	S

5.1.6 Solução

Analisando a descrição geral do problema, é possível observar que a operação *união* da estrutura de dados *disjoint-set*, faz exatamente o que uma fusão de bancos precisa. Quando dois conjuntos *a* e *b* são unidos através da estrutura, é criado outro conjunto *z*, mas ainda é possível referenciar esse conjunto *z* através dos identificadores de *a* e *b*, para isso, basta utilizar a operação *busca*. É importante executar a função *construir* antes de realizar as consultas. Se utilizarmos união por tamanho ou por classificação, juntamente com a compressão de caminho, sabemos que para cada operação realizada a complexidade quase que constante. Portanto, as duas operações que mais custam no código é construir N vezes, O(N) e a leitura de K operações, O(K). A complexidade final é O(N + K). Abaixo, a implementação da solução:

Algoritmo Fusões

```
1:  char SOLUCAO(char caractere) {
2:      if (caractere ==' F') {                          ▷ operação de fusão
3:          UNIAO(banco1, banco2);
4:      } else {                                         ▷ operação de consulta
5:          if (BUSCA(banco1) == BUSCA(banco2)) {        ▷ se referem ao mesmo banco
6:              return 'S';
7:          } else {                                     ▷ bancos diferentes
8:              return 'N';
9:          }
10:     }
11: }
```

5.2 Famílias de Troia

Problema retirado da segunda fase da modalidade programação nível 2 da Olimpíada Brasileira de Informática (OBI) do ano de 2013.

5.2.1 Link do problema

https://olimpiada.ic.unicamp.br/pratique/p2/2013/f2/troia/

5.2.2 Descrição geral

Foram encontradas inscrições numa caverna a respeito de sobreviventes da Guerra de Troia. Arqueólogos descobriram que as inscrições descreviam relações de parentescos numa certa população. Cada item da inscrição indicavam duas pessoas que pertenciam a uma mesma família. Seu problema é determinar quantas famílias distintas existem.

5.2.3 Descrição da entrada

A primeira linha apresenta um inteiro N (número de pessoas) e M (número de relações de parentesco), onde $1 <= N <= 5 * 10^4$ e $1 <= M <= 10^5$. As M linhas seguintes contém dois inteiros X e Y, e indica que X tem relação de parentesco com Y.

5.2.4 Descrição da saída

A saída deve conter apenas uma linha contendo um único inteiro, que é o número de famílias.

5.2.5 Exemplo de caso de teste

ENTRADA		SAÍDA
9	8	3
1	2	
2	3	
3	6	
4	3	
6	5	
7	8	
1	4	
6	2	

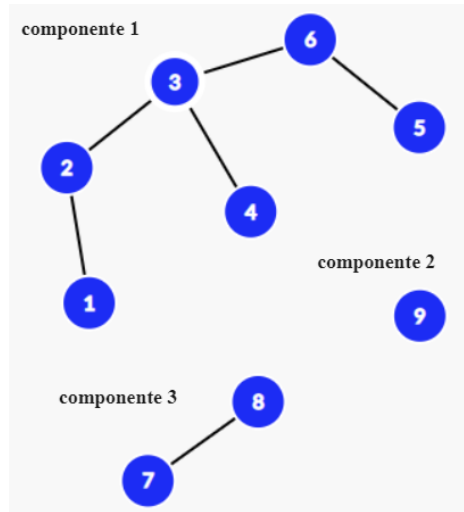
5.2.6 Solução

Esse é um problema clássico de contagem de componentes conexas. Em um grafo não dirigido uma componente conexa é um subgrafo no qual cada par de nós está conectado entre

si por meio de um caminho. A estrutura de dados disjoint-set pode ser utilizada para contar as componentes conexas de um grafo.

Para resolver o problema, inicialmente é necessário construir a árvore de 1 até N, em seguida, para cada relação de parentesco, a operação de *união* deve ser chamada. Para o exemplo do caso de teste apresentado, as árvores resultantes após as operações é ilustrada na imagem abaixo:

Figura 15 - Árvore após união de parentes



Fonte: Autoria própria, 2022.

Para contar o número de famílias (componentes conexas), é necessário saber o número de árvores restantes após as operações de *união*. Nesse sentido, basta iterar por todas as pessoas e para cada pessoa que tem como representante si mesmo, contar para resposta. Abaixo, a implementação da solução:

Algoritmo Famílias de Troia

```
1:  int SOLUCAO(int N, int M) {
2:      for (int i = 1; i <= N; i++) {
3:          CONSTRUIR(i);                      ▷ construindo árvore para cada pessoa
4:      }
5:      for (int i = 1; i <= M; i++) {
6:          UNIAO(x[i], y[i]);                 ▷ união de parentes
7:      }
8:      int familias = 0;
9:      for (int i = 1; i <= N; i++) {
10:         if (pai[i] == i) {                 ▷ i é raiz de uma árvore
11:             familias = familias + 1;
12:         }
13:     }
14:     return familias;
15: }
```

Algoritmos de busca como busca em profundidade e busca em largura também podem ser utilizados para resolver o problema de contagem de componentes conexas, ambas executam em O(V + E) onde V é o número de vértices e E o número de arestas.

A solução apresentada, tem complexidade O(N + M), já que as operações utilizando a *disjoint-set* são quase que constantes.

5.3 Reduzindo detalhes em um mapa

Problema retirado da segunda fase da modalidade programação nível 2 da Olimpíada Brasileira de Informática (OBI) do ano de 2011.

5.3.1 Link do problema

https://olimpiada.ic.unicamp.br/pratique/p2/2011/f2/rmapa/

5.3.2 Descrição geral

Dado um mapa de cidades e rodovias que as ligam, selecione um subconjunto das rodovias tal que entre qualquer par de cidades exista uma rota ligando-as e a soma dos comprimentos das rodovias é minímo. O problema é determinar a soma dos comprimentos das rodovias do subconjunto selecionado para um dado mapa.

5.3.3 Descrição da entrada

A primeira linha apresenta um inteiro N (número de cidades) e M (número de rodovias), onde $1 <= N <= 500$ e $1 <= M <= 124750$. As M linhas seguintes é composta por três inteiros U, V e C que indicam que exite uma rodovia de comprimento C que liga as cidades U e V.

5.3.4 Descrição da saída

A saída deve conter apenas uma linha apresentando a soma do comprimento das rodovias selecionadas.

5.3.5 Exemplo de caso de teste

ENTRADA			SAÍDA
5	6		34
1	2	15	
1	3	10	
2	3	1	
3	4	3	
2	4	5	
4	5	20	

5.3.6 Solução

Para resolver esse problema é necessário encontrar a árvore geradora mínima do grafo não-dirigido. Uma árvore geradora T de G é mínima se nenhuma outra árvore tem custo menor que o de T.

Uma das abordagens para encontrar a árvore geradora mínima é o algoritmo de Kruskal, tal qual, é um exemplo de algoritmo guloso. O funcionamento do mesmo é descrito a seguir:

- crie uma floresta (um conjunto de árvores), onde cada vértice no grafo é uma árvore separada;
- crie um conjunto S contendo todas as arestas do grafo;
- ordenar (crescente) o conjunto S em relação com o peso das arestas;
- iterar pelo conjunto S e para cada aresta que conecta árvores diferentes, unir os dois vértices da aresta em uma árvore.

No final do algoritmo, é obtido como resultado a árvore geradora mínima.

Para representar as arestas do grafo, é possível utilizar structs em C++, no código abaixo *u* e *v* são vértices de uma aresta cujo peso é *peso*.

Algoritmo Estrutura para armazenar as arestas

```
1: struct ARESTA {
2:     int u, v, peso;
3: };
```

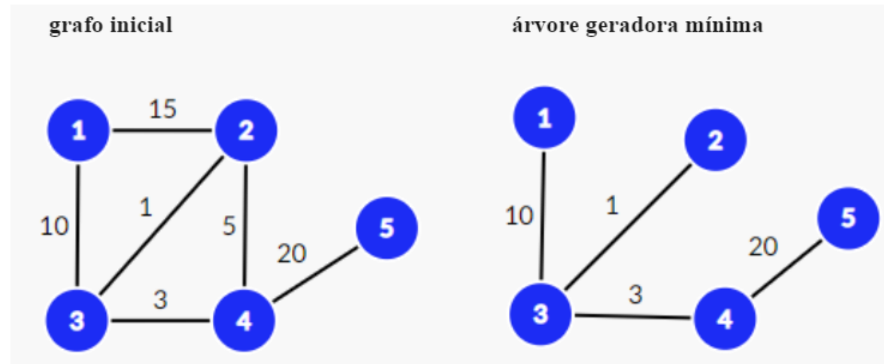
As arestas devem ser ordenadas de forma crescente em relação ao peso das arestas, a função abaixo é utilizada para comparar e ordenar os pesos nessa ordem.

Algoritmo Ordenação

```
1: bool COMPARAR(aresta A, aresta B) {
2:     return A.peso < B.peso;
3: }
```

Para a operação de união, pode-se utilizar um algoritmo de força bruta, mas a estrutura de dados *disjoint-set* é utilizada dentro do algoritmo de Kruskal para melhorar a eficiência do mesmo. É utilizado para criar uma floresta (*construir*), verificar se uma aresta conecta árvore diferentes (*busca*) e para unir dois vértices de uma aresta (*união*).

Figura 16 - Exemplo do caso de teste



Fonte: Autoria própria, 2022.

Implementação força bruta:

Algoritmo Reduzinho detalhes em um mapa - Força Bruta

```
1: int SOLUCAO(int N, int M, aresta Aresta[]) {
2:     int arvore_id[N + 2] = {};            ▷ armazena o identificador da árvore de cada vértice
3:     for (int i = 1; i <= N; i++) {
4:         arvore_id[i] = i;
5:     }
6:     SORT(Aresta + 1, Aresta + M + 1, comparar);     ▷ ordena em relação ao peso das arestas
7:     soma_compr = 0;
8:     for (int i = 1; i <= M; i++) {
9:         if (arvore_id[Aresta[i].u] != arvore_id[Aresta[i].v]) {          ▷ árvores diferentes
10:            soma_compr = soma_compr + Aresta[i].peso;     ▷ peso da aresta adicionado a resposta
11:            int id1 = arvore_id[Aresta[i].u];
12:            int id2 = arvore_id[Aresta[i].v];
13:            for (int j = 1; j <= N; j++) {
14:                if (arvore_id[j] == id1) {
15:                    arvore_id[j] = id2 ▷ representa a operação de união, id2 é o representante de u e v
16:                }
17:            }
18:        }
19:    }
20:    return soma_compr;
21: }
```

A complexidade para ordenar o vetor Aresta utilizando o método sort do C++ é de O(M log M), onde M é o número de arestas do grafo. M é no máximo N^2, e $\log N^2 = 2\log N = \log N$, ou seja O(M log M) = O(M log N). Na implementação acima a operação de união, tem complexidade de O(N). Como são feitas no máximo N-1 operações de união, a abordagem com força bruta tem complexidade final de $O(M \log N + N^2)$.

Implementação com *disjoint-set*:

Algoritmo Reduzinho detalhes em um mapa - Disjoint-set

```
1: int SOLUCAO(int N, int M, aresta Aresta[]) {
2:     for (int i = 1; i <= N; i++) {
3:         CONSTRUIR(i);                                   ▷ cada vértice é uma árvore
4:     }
5:     SORT(Aresta + 1, Aresta + M + 1, comparar);
6:     soma_compr = 0;
7:     for (int i = 1; i <= M; i++) {
8:         if (BUSCA(Aresta[i].u) != BUSCA(Aresta[i].v)) {  ▷ se os vértices estão em árvores diferentes
9:             UNIAO(Aresta[i].u, Aresta[i].v);             ▷ unir esses vértices
10:            soma_compr = soma_compr + Aresta[i].peso;    ▷ peso da aresta adicionado a resposta
11:        }
12:    }
13:    return soma_compr;
14: }
```

Utilizando *disjoint-set* as operações de união são constantes e é possível atingir a complexidade final igual a O(M log N).

5.4 Distribuição de notícias

Problema retirado da competição *Educational Codeforces Round 65* do juiz *online* Codeforces.

5.4.1 Link do problema

https://codeforces.com/contest/1167/problem/C

5.4.2 Descrição geral

Em uma rede social, existem N usuários comunicando entre si em M grupos diferentes. O problema é analisar o processo de distribuição de notícias entre os usuários. Inicialmente, algum usuário X recebe uma notícia de alguma fonte. Em seguida, ele manda essa notícia para seus amigos (dois usuários são amigos se existe pelo menos um grupo em que ambos pertencem). Amigos continuam mandando a notícia para outros amigos, esse processo só termina quando não existe um par de amigos que um sabe a notícia e o outro não.

Para cada usuário X, determinar qual é o número de usuários que saberá da notícia caso inicialmente apenas o usuário X comece a distribuí-la.

5.4.3 Descrição da entrada

A primeira linha apresenta um inteiro N (número de usuários) e M (número de grupos), onde $1 <= N, M <= 5 * 10^5$. Cada uma das M linhas seguintes descreve um grupo

de amigos. A M_i linha começa com um inteiro K (número de amigos no grupo), onde $0 <= K <= N$, em seguida, K inteiros diferentes são apresentados, descrevendo os usuários pertencentes ao grupo.

5.4.4 Descrição da saída

A saída deve conter N inteiros. O i-ésimo inteiro deve ser igual ao número de usuários que saberão da notícia caso o usuário i comece a distribuí-la.

5.4.5 Exemplo de caso de teste

ENTRADA				SAÍDA						
7	5			4	4	1	4	4	2	2
3	2	5	4							
0										
2	1	2								
1	1									
2	6	7								

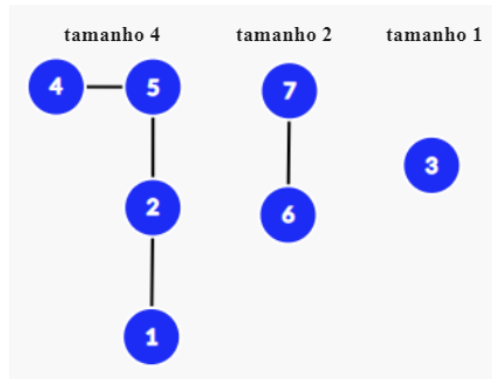
5.4.6 Solução

Baseando no fato de que quando um usuário recebe uma notícia ele a espalha para seus amigos, é possível resolver esse problema utilizando *disjoint-set*. Para unir amigos de um mesmo grupo, pode-se utilizar a operação de *união*. Também pode acontecer de dois ou mais grupos terem amigos em comum, nesse caso, a notícia que começou a ser distribuída em um grupo também chega ao outro.

Após todas as operações de *união*, cada conjunto representa uma rede de amigos onde qualquer usuário de um conjunto distribui a notícia para todos os elementos do mesmo. Nesse sentido, a quantidade de usuários que saberão da notícia caso algum usuário i comece distribuí-la é igual a quantidade de elementos no conjunto em que i está presente.

Para saber a quantidade de elementos de um conjunto na estrutura *disjoint-set*, pode-se utilizar o array *tam* para guardar o tamanho do conjunto, como visto na união por tamanho. Dessa forma, para um elemento i, a resposta final é *tam(busca(i))*.

Figura 17 - Exemplo do caso de teste após operações de *união*



Fonte: Autoria própria, 2022.

Implementação para a solução:

Algoritmo Distribuição de notícias

```
1:  void SOLUCAO(int N, int M) {
2:      for (int i = 1; i <= N; i++) {
3:          CONSTRUIR(i);                          ▷ cada usuário pertence a um conjunto
4:      }
5:      for (int i = 1; i <= M; i++) {
6:          CIN >> K;                              ▷ quantidade de usuários no grupo
7:          for (int j = 1; j <= K; j++) {
8:              CIN >> USUARIO[J];
9:          }
10:         for (int j = 1; j <= K − 1; j++) {     ▷ união de usuários de um grupo
11:             UNIAO(usuario[j], usuario[j + 1]);
12:         }
13:     }
14:     for (int i = 1; i <= N; i++) {
15:         COUT << TAM[BUSCA(I)] << ' ';          ▷ a resposta é o tamanho do conjunto
16:     }
17:     COUT << ENDL;
18: }
```

A solução apresentada tem complexidade final de O(N + M + $\sum_{i=1}^{m} k_i$), onde são feitas N chamadas de construção e para M grupos de amigos o número máximo de arestas do grafo é o somatório de 1 até m de k_i. Esse problema também pode ser resolvido com uma busca em grafos utilizando algoritmos como busca em largura ou busca em profundidade.

6 CONCLUSÃO

O desenvolvimento desse trabalho possibilitou demonstrar a importância da estrutura de dados *disjoint-set* para resolução de problemas de competições de programação.

Foram apresentadas as três operações da estrutura e a implementação das mesmas. Partindo da maneira mais ingênua e aplicando otimizações para melhorar o desempenho em relação a complexidade do algoritmo.

Sabendo da importância do estudo de estruturas de dados, o âmbito acadêmico deve incentivar cada vez mais o ensino de técnicas como *union-find*. Desenvolver o pensamento lógico e matemático para resolver problemas e aplicar assuntos do campo da computação como análise de algoritmos, é de fundamental importância para o desenvolvimento do cientista da computação.

O presente trabalho teve como objetivo principal apresentar como a estrutura de dados *disjoint-set* é utilizada para resolver problemas de competições de programação e fornecer um material em português sobre o assunto, visto que, a grande parte é encontrado em inglês. A presença de problemas da Olimpíada Brasileira de Informática aparecem como motivação para aqueles estudantes que participam da competição.

Nesse sentido, o trabalho pode ser utilizado por aqueles que desejam aprender sobre *disjoint-set* seja para participar de competições ou somente por curiosidade.

REFERÊNCIAS

HALIM, Steven; HALIM, Felix. **Competitive Programming 3**. 3.ed. 2013.

CORMEM, H. Thomas; LEISERSON, E. Charles; RIVEST, L. Ronald; STEIN, Cliford. **Introduction to Algorithms**. 3.ed. Londres: The Mith Press, 2009.

LAAKSONEN, Antti. **Competitive Programmer's Handbook**. 2018.

ROSEN, H. Kenneth. **Matemática Discreta e Suas Aplicações**. 6.ed. AMGH Editora, 2010.

IEZZI, Gelson; MURAKAMI, Carlos. **Fundamentos de Matemática Elementar**: Conjuntos e Funções. 7.ed. Atual Editora.

Disjoint set union. **CP Algorithms**. Disponível em: https://cp-algorithms.com/data_structures/disjoint_set_union.html. Acesso em: 03/07/2022.

Minimum Spanning Tree - Kruskal with Disjoint Set Union. **CP Algorithms**. Disponível em: https://cp-algorithms.com/graph/mst_kruskal_with_dsu.html. Acesso em: 30/07/2022.

Sobre a OBI. **Olimpíada Brasileira de Informática**. Disponível em: https://olimpiada.ic.unicamp.br/info/. Acesso em: 01/06/2022.

O que é?. **Maratona SBC de Programação**. Disponível em: http://maratona.sbc.org.br/sobre22.html. Acesso em: 01/06/2022.

Disjoint-set data structure. **Wikipedia**. Disponível em: https://en.wikipedia.org/wiki/Disjoint-set_data_structure. Acesso em: 28/06/2022.

Análise amortizada. **Ime USP**. Disponível em: https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/amortized.html. Acesso em: 25/09/2022.

TORVALDS, Linus. Re: Licensing and the library version of git. **LWN**. Disponível em: https://lwn.net/Articles/193245/. Acesso em: 25/09/2022.

GALLER, A. Galler; FISCHER, J. Michael. An improved equivalence algorithm. **Communications of the ACM**. Maio, 1964. Disponível em: https://dl.acm.org/doi/abs/10.1145/364099.364331. Acesso em: 06/06/2022.

HOPCROFT, E. John; ULLMAN, D. Jeffrey. Set Merging Algorithms. **SIAM Journal on Computing**. 1973. Disponível em: https://epubs.siam.org/doi/abs/10.1137/0202024. Acesso em: 06/06/2022.

TARJAN, E. Robert. Efficiency of a Good But Not Linear Set Union Algorithm. **Journal of the ACM**. 1975. Disponível em: https://ecommons.cornell.edu/handle/1813/5942. Acesso em: 06/06/2022.

FREDMAN, L. Michael; Saks, Michael. The Cell Probe Complexity of Dynamic Data Structures. Maio, 1989. Disponível em: https://dl.acm.org/doi/10.1145/73007.73040. Acesso em: 06/06/2022.