

Universidade de Brasília – UnB
Faculdade de Ciências e Tecnologia em Engenharia – FCTE
Engenharia de Software

Grafos Notáveis e Casos de Borda em Programação Competitiva

Autor: André Macedo Rodrigues Alves
Orientador: Prof. Dr. Edson Alves da Costa Júnior

Brasília, DF
2026



André Macedo Rodrigues Alves

Grafos Notáveis e Casos de Borda em Programação Competitiva

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília – UnB

Faculdade de Ciências e Tecnologia em Engenharia – FCTE

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Brasília, DF

2026

André Macedo Rodrigues Alves
Grafos Notáveis e Casos de Borda em Programação Competitiva/ André Macedo Rodrigues Alves. – Brasília, DF, 2026-
121 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Faculdade de Ciências e Tecnologia em Engenharia – FCTE , 2026.

1. Grafos. 2. Programação Competitiva. I. Prof. Dr. Edson Alves da Costa Júnior. II. Universidade de Brasília. III. Faculdade de Ciências e Tecnologia em Engenharia. IV. Grafos Notáveis e Casos de Borda em Programação Competitiva

CDU 02:141:005.6

André Macedo Rodrigues Alves

Grafos Notáveis e Casos de Borda em Programação Competitiva

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 17 de julho de 2026:

Prof. Dr. Edson Alves da Costa Júnior
Orientador

Prof. Dr. Bruno César Ribas
Convidado 1

Prof. Dr. Daniel de Paula Porto
Convidado 2

Brasília, DF
2026

Agradecimentos

Agradeço à minha família pelo apoio constante ao longo de toda a graduação. Registro também minha gratidão aos docentes e à comunidade acadêmica que incentivam a programação competitiva e as maratonas no Distrito Federal, em especial por meio de iniciativas como o Maratonas DF, e aos que contribuem com a organização de competições, com o apoio a equipes e com a infraestrutura necessária para esses eventos e para a participação em competições nacionais.

"Program testing can be used to show the presence of bugs,
but never to show their absence."

Edsger W. Dijkstra

Resumo

Este trabalho analisa casos de borda em problemas de grafos no contexto da programação competitiva, integrando fundamentação teórica, geração automatizada de instâncias e discussão de situações extremas que afetam implementações. Inicialmente, apresenta-se uma revisão de conceitos de programação competitiva, teoria dos grafos e geração de casos de teste, seguida da formalização de grafos notáveis recorrentes em competições. Na etapa prática, foram implementados geradores com a biblioteca Testlib para construir instâncias representativas e extremas dessas estruturas. Em seguida, problemas de programação competitiva foram analisados para identificar situações que comprometem a correção e a robustez das implementações, especialmente em cenários de conectividade atípica, limites de entrada e padrões estruturais específicos, com exemplificação dos geradores em cada problema selecionado. A discussão sugere que falhas recorrentes estão associadas, em grande parte, a hipóteses implícitas sobre a entrada e que a construção sistemática de casos de teste favorece a robustez das soluções. Como contribuição, o trabalho oferece um referencial teórico-prático para apoiar o desenvolvimento e a verificação de implementações em problemas de grafos.

Palavras-chave: grafos. programação competitiva. casos de borda. geração de testes.

Abstract

This work analyzes edge cases in graph problems within competitive programming by integrating theoretical foundations, automated instance generation, and discussion of extreme situations that affect implementations. First, it reviews core concepts from competitive programming, graph theory, and test-case generation. Then, it formalizes notable graph families frequently found in contests. In the practical stage, generators were implemented with the Testlib library to build representative and extreme instances of those structures. Next, competitive programming problems were examined to identify scenarios that compromise implementation correctness and robustness, especially under unusual connectivity patterns, input limits, and specific structural configurations, with generator examples for each selected problem. The discussion suggests that recurrent failures are often linked to implicit assumptions about input data and that systematic edge-case construction favors solution robustness. As a contribution, this work provides a theoretical-practical reference to support the development and verification of implementations for graph problems in competitive environments.

Key-words: graphs; competitive programming; edge cases; test generation.

Lista de ilustrações

Figura 1 – Exemplo de grafo $G = (V, E)$ com quatro vértices. 20
Figura 2 – Exemplo de grafo direcionado. 22
Figura 3 – Exemplo de grafo ponderado, com o peso indicado sobre cada aresta. . 22
Figura 4 – Matriz de adjacências do grafo da Figura 1. 23
Figura 5 – Lista de adjacências do grafo da Figura 1. 24
Figura 6 – Lista de arestas do grafo da Figura 1. 24
Figura 7 – Progressão da BFS a partir do vértice 1; cada cor representa um nível. 27
Figura 8 – Progressão da DFS a partir do vértice 1; em laranja, o vértice recém-descoberto. 29
Figura 9 – Grafo não conexo, com duas componentes conexas. 30
Figura 10 – Ciclo incremental de desenvolvimento dos grafos notáveis 40
Figura 11 – Exemplos de grafos completos não direcionados com 3 e 5 vértices. . . 42
Figura 12 – Exemplo de grafo completo direcionado com 4 vértices. 42
Figura 13 – Grafo com 6 vértices contendo uma 4-clique destacada (vértices 1 a 4). 44
Figura 14 – Exemplo de grafo bipartido com os conjuntos de vértices representados pelas cores vermelha e azul. 45
Figura 15 – Exemplo de árvore com 7 vértices. 47
Figura 16 – Exemplo de grafo caminho P_7 (*bambu*). 48
Figura 17 – Exemplo de grafo ciclo C_6. 50
Figura 18 – Exemplo de grafo estrela com 7 vértices. 51
Figura 19 – Exemplo de floresta com três componentes (três árvores). 52
Figura 20 – Exemplo de grafo de sucessores com 4 vértices. Cada vértice possui exatamente um sucessor. 54
Figura 21 – Exemplo de grafo acíclico direcionado (DAG) com 6 vértices. 55

Lista de abreviaturas e siglas

UnB	Universidade de Brasília
FCTE	Faculdade de Ciências e Tecnologia em Engenharia
DF	Distrito Federal
TCC	Trabalho de Conclusão de Curso
BFS	*Breadth-First Search*
DFS	*Depth-First Search*
DAG	*Directed Acyclic Graph*
DSU	*Disjoint Set Union*
LIS	*Longest Increasing Subsequence*
LCA	*Lowest Common Ancestor*
MaxST	*Maximum Spanning Tree*
BIT	*Binary Indexed Tree*
AC	*Accepted*
WA	*Wrong Answer*
TLE	*Time Limit Exceeded*
MLE	*Memory Limit Exceeded*
RE	*Runtime Error*
CE	*Compilation Error*
ICPC	International Collegiate Programming Contest
OBI	Olimpíada Brasileira de Informática
SBC	Sociedade Brasileira de Computação
IOI	International Olympiad in Informatics
CSES	*Code Submission Evaluation System*

USACO	USA Computing Olympiad
BOCA	*BOCA Online Contest Administrator*
CNPq	Conselho Nacional de Desenvolvimento Científico e Tecnológico

Lista de símbolos

$G = (V, E)$	Grafo com conjunto de vértices V e conjunto de arestas E
n	Número de vértices de um grafo
m	Número de arestas de um grafo
$\|E(G)\|$	Número de arestas do grafo G
$\{u, v\}$	Aresta não direcionada entre os vértices u e v
(u, v)	Aresta direcionada de u para v
$N(v)$	Vizinhança do vértice v
$\deg(v)$	Grau do vértice v
$\text{dist}(u, v)$	Distância (em número de arestas) entre os vértices u e v
K_n	Grafo completo com n vértices
K_{n_1,n_2}	Grafo bipartido completo com partições de tamanhos n_1 e n_2
P_n	Grafo caminho com n vértices
C_n	Grafo ciclo com n vértices
U, V	Partições disjuntas de vértices em um grafo bipartido
$w : E \to \mathbb{R}$	Função de peso atribuída às arestas de um grafo ponderado
$\text{succ}(x, k)$	Sucessor de x após k passos em um grafo funcional
$O(f(n))$	Notação Big-O para complexidade assintótica
$\Theta(f(n))$	Notação Θ para complexidade assintótica justa
$\log k$	Logaritmo de k na base 2

Sumário

1 INTRODUÇÃO 14
1.1 Objetivos 15
1.2 Estrutura do Trabalho 15

2 FUNDAMENTAÇÃO TEÓRICA 16
2.1 Programação Competitiva 16
2.1.1 Estrutura de um problema 17
2.2 Conceitos Elementares de Grafos 19
2.2.1 Definição e terminologia básica 20
2.2.2 Tipos de grafos 21
2.2.3 Representações computacionais 23
2.2.4 Travessias em grafos 26
2.3 A Biblioteca Testlib 30
2.4 Trabalhos Correlatos 33

3 MATERIAIS E MÉTODOS 35
3.1 Pesquisa bibliográfica 35
3.2 Seleção de grafos notáveis e de problemas 36
3.3 Processo de desenvolvimento dos capítulos 38
3.3.1 Fundamentação teórica 38
3.3.2 Grafos notáveis 38
3.3.3 Análise de casos de borda em problemas de grafos 39
3.4 Ferramentas e metodologia de desenvolvimento 39

4 GRAFOS NOTÁVEIS 41
4.1 Grafo Completo 41
4.2 Clique 43
4.3 Grafo Bipartido 45
4.4 Árvore 46
4.5 Grafo Caminho (*Bambu*) 48
4.6 Grafo Ciclo 49
4.7 Grafo Estrela 51
4.8 Floresta 52
4.9 Grafo de Sucessores 53
4.10 Grafo Acíclico Direcionado (DAG) 55

5 ANÁLISE DE CASOS DE BORDA EM PROBLEMAS DE GRAFOS 58
5.1 Problema 1 - Bipartição por 2-Coloração 58
5.2 Problema 2 - Ordenação topológica (*Course Schedule*) 63
5.3 Problema 3 - Subsequência crescente em caminhos de árvore (*Caminhada na Montanha*) 67
5.4 Problema 4 - Componente conexa restrita a intervalo em árvore (*Empresa de Festas*) 72
5.5 Problema 5 - Menor gerente na cadeia de comando (*Boss*) 79
5.6 Problema 6 - Capacidade máxima em caminhos de entrega (*Trucks*) 85
5.7 Problema 7 - Ciclo simples (*Round Trip*) 91
5.8 Problema 8 - Contagem de sanduíches (Sanduíche) 95
5.9 Problema 9 - Conexão de componentes (*Building Roads*) 99
5.10 Problema 10 - Ciclos em grafo de sucessores (*Planets Cycles*) . . . 103

6 CONSIDERAÇÕES FINAIS 108
6.1 Trabalhos futuros 108

REFERÊNCIAS 109

APÊNDICES 111

APÊNDICE A – CÓDIGO-FONTE DA CLASSE GRAPH 112

1 Introdução

A programação competitiva combina o projeto de algoritmos com a sua implementação em programas cuja correção é avaliada automaticamente: a solução submetida é executada sobre um conjunto de casos de teste e comparada à saída esperada, dentro de limites de tempo e de memória (LAAKSONEN, 2018). Entre os tópicos fundamentais dessa modalidade, os grafos ocupam posição de destaque, com temas como travessias, conectividade, ciclos e caminhos mínimos (SKIENA; REVILLA, 2003). Em plataformas como Codeforces[1], AtCoder[2] e CSES (*Code Submission Evaluation System*)[3], é comum encontrar problemas cuja solução depende de arestas, caminhos, árvores, ciclos e outras estruturas representadas por grafos.

Como a correção é aferida por um conjunto de testes definido por quem propõe o problema, e não por uma prova formal, a robustez da implementação depende de sua capacidade de lidar com toda a variedade de entradas admitidas. Dominar a habilidade de testar o próprio código, verificando deliberadamente valores extremos e situações-limite (como entradas de tamanho mínimo ou máximo, valores nulos ou negativos e configurações degeneradas), é apontado como competência essencial ao competidor (HALIM; HALIM; EFFENDY, 2020). Essas situações-limite, os chamados casos de borda, costumam passar despercebidas: em problemas de grafos, incluem grafos com número mínimo ou máximo de vértices, grafos desconexos quando se supõe conectividade e estruturas atípicas, como árvores lineares (grafos caminho). Uma implementação pode funcionar nos casos comuns e, ainda assim, falhar diante dessas configurações.

A dificuldade de antecipar tais situações está associada às propriedades estruturais de cada tipo de grafo. Certas famílias admitem algoritmos especializados, mas também tensionam implementações genéricas: os grafos de sucessores, por exemplo, em que cada vértice tem grau de saída igual a um, decompõem-se em componentes que contêm exatamente um ciclo e caminhos que a ele conduzem (LAAKSONEN, 2018). Conhecer as famílias clássicas da teoria dos grafos (WEST, 2001; DIESTEL, 2017) e estruturas frequentes em programação competitiva, como o grafo acíclico direcionado (DAG) (HALIM; HALIM; EFFENDY, 2020), é um passo relevante para elaborar casos de teste que cubram tanto o comportamento típico quanto situações extremas. Para construir essas instâncias de forma sistemática, elaboradores de problemas e competidores recorrem a geradores automatizados, apoiados por bibliotecas como Testlib (MIRZAYANOV, 2005), jngen (SMIRNOV,) e tgen (MALETTA,). Compreender como os casos de borda surgem, quais estruturas os

[1] <https://codeforces.com>
[2] <https://atcoder.jp>
[3] <https://cses.fi>

tornam prováveis e como gerá-los automaticamente constitui, assim, etapa importante na preparação para competições e na validação de implementações.

1.1 Objetivos

O objetivo geral deste trabalho é analisar casos de borda em problemas de grafos no contexto da programação competitiva, articulando a fundamentação teórica dos grafos notáveis com a geração automatizada de instâncias e com a discussão de situações extremas que afetam a correção e a robustez das implementações.

Os objetivos específicos são:

(a) realizar revisão bibliográfica sobre programação competitiva, teoria dos grafos e geração de casos de teste;

(b) formalizar e ilustrar os grafos notáveis recorrentes em competições, reunidos no Quadro 3;

(c) implementar geradores de instâncias para grafos notáveis com apoio da biblioteca Testlib;

(d) analisar, em problemas representativos, casos de borda que comprometam a correção e a robustez das implementações;

(e) ilustrar a construção e o uso de instâncias extremas na verificação de implementações em problemas de grafos.

1.2 Estrutura do Trabalho

Este trabalho está estruturado em seis capítulos: no Capítulo 1 são apresentados a introdução ao tema, os objetivos e a motivação do estudo; o Capítulo 2 aborda a programação competitiva, os conceitos elementares de grafos, as travessias (busca em largura, *Breadth-First Search*, BFS; busca em profundidade, *Depth-First Search*, DFS; e conectividade), a biblioteca Testlib e a apresentação dos trabalhos correlatos; o Capítulo 3 detalha o levantamento bibliográfico, a seleção dos grafos notáveis e dos problemas, o processo de desenvolvimento e as ferramentas utilizadas; o Capítulo 4 descreve os grafos notáveis selecionados, com definição, ilustração e gerador para cada família; o Capítulo 5 apresenta problemas de competições e discute instâncias extremas e soluções; e, por fim, o Capítulo 6 traz as considerações finais do estudo.

2 Fundamentação Teórica

Neste capítulo são apresentados os fundamentos teóricos e os trabalhos correlatos essenciais ao desenvolvimento deste estudo. Na Seção 2.1 explora-se o universo da programação competitiva, abordando as plataformas e eventos que incentivam essa modalidade de competição. Em seguida, na Seção 2.2, define-se formalmente o conceito de grafo, suas classificações e algoritmos fundamentais de travessia. Na Seção 2.3 descreve-se a biblioteca Testlib, empregada para geração automatizada de casos de teste e validação de soluções em competições de programação. Por fim, na Seção 2.4, são apresentados alguns trabalhos que abordam temas comuns a este estudo, incluindo grafos, casos de borda, programação competitiva e geração de testes.

2.1 Programação Competitiva

A programação competitiva é uma modalidade de competição intelectual em que os participantes desenvolvem algoritmos para resolver corretamente um conjunto de problemas predeterminados (HALIM; HALIM; EFFENDY, 2020). Esses problemas, tipicamente de natureza lógica ou matemática, são apresentados pelos organizadores dentro de um prazo fixo, e a avaliação é realizada de forma automatizada: a solução é executada sobre um conjunto de casos de teste e comparada à saída esperada, dentro de limites de tempo e de memória; os critérios mais comuns englobam o número de problemas solucionados corretamente e o tempo total de resolução, podendo também considerar a eficiência sob esses limites (LAAKSONEN, 2018). As competições podem ocorrer presencialmente ou via redes locais e internet, contando com o patrocínio de grandes empresas de tecnologia, como Meta.

Dentre os principais eventos da área, destacam-se, em termos de relevância para a formação e progressão competitiva no contexto deste trabalho, a Olimpíada Brasileira de Informática (OBI)[1], a Maratona de Programação da Sociedade Brasileira de Computação (SBC)[2] e a International Olympiad in Informatics (IOI)[3]. A OBI cumpre papel formativo para estudantes do ensino fundamental, médio e técnico; a Maratona da SBC atua como etapa nacional universitária e classificatória para o International Collegiate Programming Contest (ICPC); e a IOI representa uma referência internacional para olimpíadas de informática no nível pré-universitário. Além delas, também se destacam o ICPC[4], a Meta

[1] <https://olimpiada.ic.unicamp.br/>
[2] <https://maratona.sbc.org.br/>
[3] <https://ioinformatics.org/>
[4] <https://icpc.global/>

Hacker Cup[5], o AtCoder Grand Contest (AGC)[6], o Codeforces Global Round[7] e o Topcoder Open (TCO)[8].

Além das competições presenciais, existem diversas plataformas online que permitem praticar e participar de torneios virtuais. Alguns dos principais juízes online incluem o Codeforces, uma plataforma russa com competições regulares, rankings globais e uma comunidade ativa[9]; o AtCoder, um site japonês similar ao Codeforces, com torneios regulares aos sábados[10]; o CSES (*Code Submission Evaluation System*), um repositório finlandês de problemas com foco educacional, organizado de modo progressivo[11]; e o CodeChef, uma plataforma indiana voltada tanto para competições quanto para o ensino de algoritmos[12].

Para competir nessas plataformas e eventos, é necessário cobrir uma variedade de tópicos da ciência da computação e da matemática. Muitos eventos, como a OBI[13] e a USACO (USA Computing Olympiad)[14], disponibilizam um *syllabus*. Isto é, um documento que organiza os conteúdos e competências esperados em cada nível da competição. Geralmente, esses conteúdos incluem estruturas de dados, algoritmos de ordenação, técnicas de busca e teoria de grafos, dentre outros. Em relação aos grafos, problemas envolvendo arestas, caminhos, ciclos e árvores são comuns, de modo que o conhecimento destes conceitos é fundamental para os competidores.

2.1.1 Estrutura de um problema

Em plataformas e competições de programação, cada problema segue uma estrutura relativamente padronizada: enunciado (*statement*), formato de entrada (*input*), formato de saída (*output*), exemplos (*samples*) e observações (*notes*). O enunciado define o contexto e a tarefa computacional; as seções de entrada e saída especificam rigorosamente o contrato de dados; os exemplos mostram pares entrada–saída válidos; e as observações registram detalhes que evitam ambiguidades de interpretação.

Como exemplo real, considere o problema “Divisores” da OBI 2015 (Fase 1)[15].

[5] <https://www.facebook.com/codingcompetitions/hacker-cup>
[6] <https://atcoder.jp/>
[7] <https://codeforces.com/>
[8] <https://www.topcoder.com/community/competitive-programming/tco>
[9] <https://codeforces.com>
[10] <https://atcoder.jp>
[11] <https://cses.fi>
[12] <https://www.codechef.com>
[13] <https://olimpiada.ic.unicamp.br/prepare/ementas/>
[14] <https://usaco.org/index.php?page=training>
[15] <https://olimpiada.ic.unicamp.br/pratique/ps/2015/f1/divisores/>

Enunciado

Um pesquisador precisa determinar o número de divisores de um inteiro positivo N. Por exemplo, 660 possui 24 divisores positivos. Como $660 = 2^2 \cdot 3 \cdot 5 \cdot 11$, a quantidade de divisores é calculada por $(2+1)(1+1)(1+1)(1+1) = 24$. De modo análogo, $50 = 2 \cdot 5^2$, portanto possui $(1+1)(2+1) = 6$ divisores.

Entrada

Uma linha contendo um único inteiro N.

Saída

Uma linha contendo um único inteiro, a quantidade de divisores positivos de N.

Restrições

$1 \leq N \leq 10^4$.

Exemplos

Entrada	**Saída**
660	24
50	6
9216	33

O Código 1 apresenta uma solução em C++ para esse problema. A rotina percorre os candidatos i de 1 até $\lfloor\sqrt{N}\rfloor$. Quando i divide N, o par $(i, N/i)$ contribui com dois divisores distintos; no caso em que $i = N/i$, conta-se apenas um divisor. Com isso, a implementação tem complexidade assintótica de tempo de execução $O(\sqrt{N})$ e complexidade de memória $O(1)$.

Código 1 – Contagem de divisores, baseado na OBI 2015 Fase 1 “Divisores”.

```cpp
#include <bits/stdc++.h>

using namespace std;

void solve() {
    int N;
    cin >> N;
    int divs = 0;
    for (int i = 1; i * i <= N; i++) {
        if (N % i == 0) {
            divs += (N / i != i) ? 2 : 1;
```

```
        }
    }
    cout << divs << "\n";
}

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);
    solve();
    return 0;
}
```

Fonte: o Autor.

Após implementar a solução, realiza-se a submissão no juiz online. No sistema BOCA adotado na Maratona SBC[16], por exemplo, o competidor escolhe o problema, envia o código-fonte e aguarda a avaliação automática. O sistema compila o programa, executa-o em um conjunto de casos de teste e compara a saída produzida com a saída esperada, respeitando limites de tempo e memória.

O Quadro 1 sintetiza vereditos frequentes desse processo de correção automática.

Quadro 1: Vereditos comuns em juízes online

Veredito	**Interpretação**
Accepted (AC)	A saída coincide com a esperada em todos os casos dentro dos limites de tempo e memória.
Wrong Answer (WA)	O programa executa, mas produz saída incorreta em pelo menos um caso de teste.
Time Limit Exceeded (TLE)	O tempo de execução excede o limite definido para ao menos um caso.
Memory Limit Exceeded (MLE)	O uso de memória ultrapassa o limite estabelecido pelo juiz.
Runtime Error (RE)	O programa termina com erro em execução, como acesso inválido de memória ou divisão por zero.
Compilation Error (CE)	O código não compila no ambiente oficial da plataforma.

Fonte: o Autor.

2.2 Conceitos Elementares de Grafos

Grafos são uma das estruturas unificadoras da computação: com um único formalismo, modelam redes de transporte, circuitos elétricos, o fluxo de execução de um programa, relações entre pessoas em uma rede social e inúmeras outras situações (SKIENA; REVILLA, 2003). Essa generalidade explica por que grafos aparecem com tanta frequência

[16] <https://maratona.sbc.org.br/latam-2025/info_sheet.pdf>

em problemas de programação competitiva (LAAKSONEN, 2018). Nesta seção, parte-se de uma única definição geral e, a partir dela, caracterizam-se os tipos de grafo segundo dimensões independentes (orientação, peso, presença de ciclos, multiplicidade de arestas e densidade). Em seguida, discutem-se as formas de representar um grafo em um computador e, por fim, os algoritmos de travessia, junto a algumas de suas aplicações.

2.2.1 Definição e terminologia básica

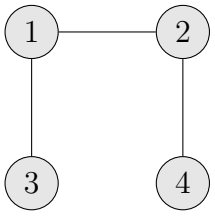
> **Definição 1** (Grafo)**.** *Um grafo G é uma estrutura composta por um conjunto finito de vértices V e um conjunto de arestas E, em que cada aresta representa uma ligação entre um par de vértices. Formalmente, escreve-se $G = (V, E)$, em que cada aresta é um par $\{u, v\}$ com $u, v \in V$ (CORMEN et al., 2009).*

Salvo menção em contrário, ao longo deste trabalho um grafo é considerado não direcionado (as arestas não têm orientação, de modo que $\{u, v\}$ e $\{v, u\}$ denotam a mesma ligação) e simples (sem laços nem arestas repetidas).[17] As demais variações são tratadas na Subseção 2.2.2.

A partir dessa definição, derivam-se os termos empregados no restante do texto. Dada uma aresta $\{u, v\}$, diz-se que u e v são suas extremidades, que u e v são adjacentes (ou vizinhos) e que a aresta incide sobre ambos. A vizinhança de um vértice v, denotada $N(v)$, é o conjunto de todos os vértices adjacentes a v; o grau de v, denotado $\deg(v)$, é o número de arestas que incidem sobre ele, ou seja, $\deg(v) = |N(v)|$ em um grafo simples.

A Figura 1 ilustra um grafo $G = (V, E)$. Nesse exemplo, tem-se $V = \{1, 2, 3, 4\}$ e $E = \{\{1, 2\}, \{1, 3\}, \{2, 4\}\}$. Cada elemento de V corresponde a um vértice (aqui identificado por um número), e cada par em E corresponde a uma aresta. Por exemplo, $N(1) = \{2, 3\}$ e $\deg(1) = 2$.

Figura 1 – Exemplo de grafo $G = (V, E)$ com quatro vértices.



Fonte: o Autor.

Para descrever como percorrer um grafo, são necessários ainda os conceitos de passeio, caminho e ciclo.

[17] Um grafo que admite laços (arestas $\{v, v\}$) ou arestas paralelas (mais de uma aresta ligando o mesmo par de vértices) é chamado de multigrafo. Nesse caso, E deixa de ser um conjunto e passa a ser um multiconjunto de pares, pois um mesmo par pode ocorrer mais de uma vez (DIESTEL, 2017). Todo grafo que não é simples é um multigrafo.

Definição 2 (Passeio, caminho e ciclo)**.** *Um passeio de v_0 a v_k é uma sequência de vértices $v_0, v_1, \ldots, v_k$ tal que $\{v_{i-1}, v_i\} \in E$ para todo $1 \leq i \leq k$. Um caminho é um passeio em que não há vértices repetidos. Um ciclo é um passeio com $k \geq 3$ em que $v_0 = v_k$ e os demais vértices não se repetem (DIESTEL, 2017).*

O número de arestas de um passeio é o seu comprimento. A distância entre dois vértices u e v, denotada $\text{dist}(u, v)$, é o comprimento do menor caminho entre eles; quando não há caminho ligando u e v, define-se $\text{dist}(u, v) = \infty$. Fixado um vértice de origem s, o nível de um vértice v é simplesmente $\text{dist}(s, v)$, isto é, o número mínimo de arestas necessário para alcançar v a partir de s.

Diz-se que um grafo não direcionado é conexo quando existe caminho entre todo par de vértices; caso contrário, ele se decompõe em componentes conexas (subconjuntos maximais dois a dois ligados por caminhos). A formalização desses conceitos aparece na Definição 8, após as travessias.

2.2.2 Tipos de grafos

A definição anterior fixa o caso mais comum, mas o primeiro passo em qualquer problema com grafos é identificar com qual tipo de grafo se está lidando, pois isso determina as estruturas de dados e os algoritmos adequados (SKIENA; REVILLA, 2003). As propriedades a seguir são independentes entre si: um grafo pode, por exemplo, ser simultaneamente direcionado, ponderado e esparso.

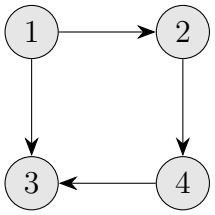
Direcionados × não direcionados

Em um grafo direcionado (ou dígrafo), cada aresta é um par ordenado (u, v) (aresta direcionada), que indica uma ligação orientada de u para v; assim, (u, v) e (v, u) representam arestas distintas. Distinguem-se, então, o grau de saída de um vértice (número de arestas que partem dele) e o grau de entrada (número de arestas que chegam a ele). Redes de ruas com mãos únicas e o fluxo de execução de um programa são naturalmente direcionados, ao passo que uma malha rodoviária de mão dupla é não direcionada (SKIENA; REVILLA, 2003). A Figura 2 apresenta um exemplo, em que a orientação de cada aresta é indicada por uma seta.

Ponderados × não ponderados

Em um grafo ponderado, associa-se a cada aresta um valor numérico, chamado de peso, por meio de uma função peso $w : E \to \mathbb{R}$. O peso costuma modelar grandezas como distância, custo ou capacidade. A distinção é particularmente relevante no problema do caminho mínimo: em grafos não ponderados, o caminho mínimo é o de menor número de

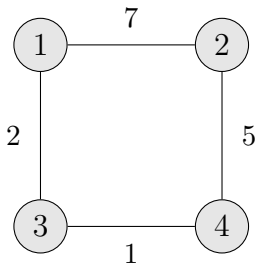
Figura 2 – Exemplo de grafo direcionado.



Fonte: o Autor.

arestas e pode ser obtido pela busca em largura; em grafos ponderados, o problema exige algoritmos mais sofisticados (SKIENA; REVILLA, 2003). A Figura 3 exibe um grafo não direcionado em que cada aresta recebe um peso.

Figura 3 – Exemplo de grafo ponderado, com o peso indicado sobre cada aresta.



Fonte: o Autor.

Cíclicos × acíclicos

Um grafo é acíclico quando não contém nenhum ciclo. Entre os grafos acíclicos não direcionados, destaca-se a árvore (um grafo conexo e acíclico), uma das estruturas mais simples e recorrentes, tratada em detalhe no Capítulo 4. No caso direcionado, um grafo acíclico é chamado de grafo acíclico direcionado (DAG, do inglês *Directed Acyclic Graph*), estrutura que aparece naturalmente em problemas de precedência e escalonamento, em que uma aresta (u, v) indica que u deve preceder v, e que é formalizada adiante na Definição 18 (SKIENA; REVILLA, 2003).

Simples × não simples

Conforme a convenção adotada, os grafos deste trabalho são simples: não possuem laços nem arestas paralelas. Quando essa restrição é relaxada, obtém-se um multigrafo, em que um mesmo par de vértices pode ser ligado por várias arestas e em que um vértice pode estar ligado a si próprio por um laço. Tais estruturas exigem cuidados adicionais na implementação de algoritmos; por isso, qualquer grafo que as evita é dito simples (SKIENA; REVILLA, 2003).

Esparsos × densos

O número de arestas de um grafo simples com n vértices varia de 0 a $\frac{n(n-1)}{2}$. Um grafo é dito denso quando $m = |E|$ é próximo desse máximo (da ordem de n^2) e esparso quando m é da ordem de n. Essa distinção é decisiva na escolha da representação: grafos densos favorecem a matriz de adjacências, enquanto grafos esparsos são representados de forma mais econômica por listas de adjacências, como se discute a seguir (SKIENA; REVILLA, 2003).

2.2.3 Representações computacionais

Para implementar algoritmos sobre grafos em um computador, é necessário escolher uma estrutura de dados que armazene V e E de forma eficiente. As representações mais usadas na literatura e na prática são a matriz de adjacências e a lista de adjacências (CORMEN et al., 2009); a lista de arestas também aparece com frequência, sobretudo quando o algoritmo itera diretamente sobre E (SKIENA; REVILLA, 2003). A seguir, cada uma é apresentada e ilustrada com o grafo da Figura 1, de modo a facilitar a comparação.

> **Definição 3** (Matriz de adjacências)**.** *Seja $G = (V, E)$ um grafo com $n = |V|$ vértices numerados de 1 a n. A matriz de adjacências é uma matriz A de ordem $n \times n$ em que $A[i][j] = 1$ se existe aresta entre os vértices i e j, e $A[i][j] = 0$ caso contrário. Em grafos não direcionados, A é simétrica; em grafos direcionados, $A[i][j]$ reflete apenas a aresta (i, j); e, em grafos ponderados, $A[i][j]$ pode armazenar o peso da aresta ou um valor sentinela quando não há ligação (CORMEN et al., 2009).*

A Figura 4 mostra a matriz de adjacências do grafo de exemplo. A entrada $A[1][2] = 1$ indica a aresta $\{1, 2\}$; entradas iguais a 0 correspondem a pares de vértices não adjacentes. O custo de espaço é $\Theta(n^2)$, independentemente do número de arestas, o que torna essa representação adequada para grafos densos ou quando consultas do tipo "i é adjacente a j?" devem ser respondidas em tempo constante.

Figura 4 – Matriz de adjacências do grafo da Figura 1.

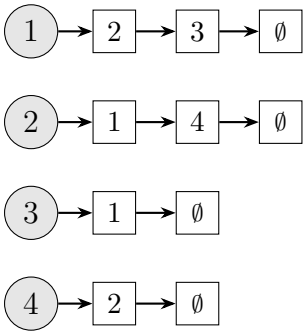
$$A = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 1 & 0 \\ 2 & 1 & 0 & 0 & 1 \\ 3 & 1 & 0 & 0 & 0 \\ 4 & 0 & 1 & 0 & 0 \end{array}$$

Fonte: o Autor.

Definição 4 (Lista de adjacências)**.** *A lista de adjacências associa a cada vértice $v \in V$ uma coleção (em geral, uma lista encadeada ou um vetor dinâmico) contendo todos os vértices adjacentes a v. Formalmente, trata-se de um arranjo* $\mathrm{Adj}[1..n]$*, em que* $\mathrm{Adj}[v]$ *armazena os vizinhos de v (CORMEN et al., 2009).*

A Figura 5 ilustra essa estrutura para o mesmo grafo: cada vértice aponta apenas para seus vizinhos, sem registrar explicitamente os demais pares. O espaço utilizado é $\Theta(n + m)$, com $m = |E|$, e percorrer todos os vizinhos de um vértice custa tempo proporcional ao seu grau. Por isso, a lista de adjacências é a escolha mais comum para grafos esparsos, em programação competitiva e em algoritmos de travessia como a busca em largura (*Breadth-First Search*, BFS) e a busca em profundidade (*Depth-First Search*, DFS).

Figura 5 – Lista de adjacências do grafo da Figura 1.



Fonte: o Autor.

Definição 5 (Lista de arestas)**.** *A lista de arestas armazena explicitamente cada elemento de E, em geral como um vetor de pares $\{u, v\}$ (ou triplas (u, v, w) em grafos ponderados), sem manter, para cada vértice, a relação direta com seus vizinhos (SKIENA; REVILLA, 2003).*

A Figura 6 reúne as três arestas do grafo de exemplo nessa forma compacta. Essa representação ocupa $\Theta(m)$ de espaço e é conveniente quando o algoritmo processa as arestas uma a uma (por exemplo, em estruturas de união e busca ou na ordenação de arestas para algoritmos de árvore geradora mínima, como o de Kruskal), mas consultar os vizinhos de um vértice específico exige percorrer toda a lista.

Figura 6 – Lista de arestas do grafo da Figura 1.

$$E = \{\{1, 2\},\ \{1, 3\},\ \{2, 4\}\}$$

Fonte: o Autor.

Em resumo, a matriz de adjacências privilegia consultas entre pares de vértices; a lista de adjacências equilibra espaço e travessia da vizinhança; e a lista de arestas minimiza o armazenamento de E quando o algoritmo itera sobre arestas. A escolha depende do grafo (denso ou esparso) e das operações exigidas pelo problema.

O Código 2 apresenta uma implementação em C++ de um grafo por meio de lista de adjacências, na forma de uma classe `Graph` construída sobre a Testlib. A classe é geral: o atributo `directed` permite representar tanto grafos direcionados quanto não direcionados; o número de vértices é mantido no atributo `n`; e o vetor de adjacências é dimensionado com `n + 1` posições, de modo que os vértices podem ser numerados de 1 a n. A implementação completa encontra-se no Apêndice A e em repositório público dedicado ao código[18].

Código 2 – Classe `Graph`.

```
17 class Graph {
18 public:
19     int n;
20     bool directed;
21     std::vector<std::vector<int>> adj;
22     // Vazio = sem enraizamento; tamanho n+1 = árvore enraizada em 1 (parent[1]=0).
23     std::vector<int> parent;
24
25     Graph(int n = 0, bool directed = false);
26
27     int m() const;
28
29     void addEdge(int u, int v);
30
31     static Graph complete(int n, bool directed = false);
32
33     static Graph completeBipartite(int n1, int n2);
34
35     static Graph randomBipartite(int n1, int n2, int m);
36
37     static Graph star(int n);
38
39     static Graph bamboo(int n);
40
41     static Graph cycle(int n);
42
43     static Graph tree(int n, int elongation = 0);
44
45     static Graph forest(int n, int components, int elongation = 0);
46
47     static Graph successors(int n);
48
49     static Graph dag(int n, int m);
50
51     static Graph withClique(int n, int k);
52
53     void printNM(std::ostream& out = std::cout) const;
```

[18] <https://github.com/andremralves/graphlib>

```
54
55      void printEdges(std::ostream& out = std::cout) const;
56
57      void printParents(std::ostream& out = std::cout) const;
58
59      std::vector<std::pair<int, int>> edges() const;
60
61  private:
62      void ensureInRange(int u) const;
63
64      void fillParentsFromRoot1();
65
66      static void addRandomTreeEdges(Graph& g, const std::vector<int>& verts, int elongation);
67  };
```

Fonte: o Autor.

2.2.4 Travessias em grafos

A operação básica da maioria dos algoritmos em grafos é percorrê-lo de forma sistemática, visitando cada vértice e examinando cada aresta um número constante de vezes. As duas estratégias fundamentais são a busca em largura e a busca em profundidade. Ambas partilham uma mesma ideia central (marcar os vértices já vistos, para não explorá-los novamente) e diferem apenas na ordem em que os vértices são descobertos (SKIENA; REVILLA, 2003). Para tanto, cada vértice passa por três estados ao longo da travessia: não descoberto, descoberto (já encontrado, mas com vizinhança ainda não totalmente explorada) e processado (após examinar todas as suas arestas).

Busca em largura

> **Definição 6** (Busca em largura, *Breadth-First Search* (BFS))**.** *Seja $G = (V, E)$ um grafo e $s \in V$ um vértice de origem. A busca em largura descobre os vértices em ordem não decrescente de nível: todos os vértices de nível k são descobertos antes de qualquer vértice de nível $k + 1$. Consequentemente, a BFS calcula a distância* $\mathrm{dist}(s, v)$ *de s a cada vértice alcançável v (CORMEN et al., 2009).*

Uma forma usual de implementar a BFS utiliza uma fila, que garante o processamento dos vértices na ordem em que são descobertos. O Código 3 apresenta essa implementação. A estrutura de fila é um detalhe da realização do algoritmo, e não parte da sua definição.

Código 3 – Implementação em C++ da busca em largura (BFS).

```
1  vector<int> bfs(int s, int N) {
2      vector<int> dist(N + 1, oo);
3      queue<int> q;
```

```
4
5       dist[s] = 0;
6       q.push(s);
7
8       while (not q.empty()) {
9           auto u = q.front();
10          q.pop();
11
12          // visita/processa u
13
14          for (auto v : adj[u]) {
15              if (dist[v] == oo) {
16                  dist[v] = dist[u] + 1;
17                  q.push(v);
18              }
19          }
20      }
21
22      return dist;
23  }
```

Fonte: Alves (2026).

Quando o grafo é representado por listas de adjacências, a BFS tem complexidade assintótica de tempo de execução $O(n + m)$, em que $n = |V|$ e $m = |E|$, pois cada vértice é processado no máximo uma vez e cada aresta é examinada um número constante de vezes (CORMEN et al., 2009).

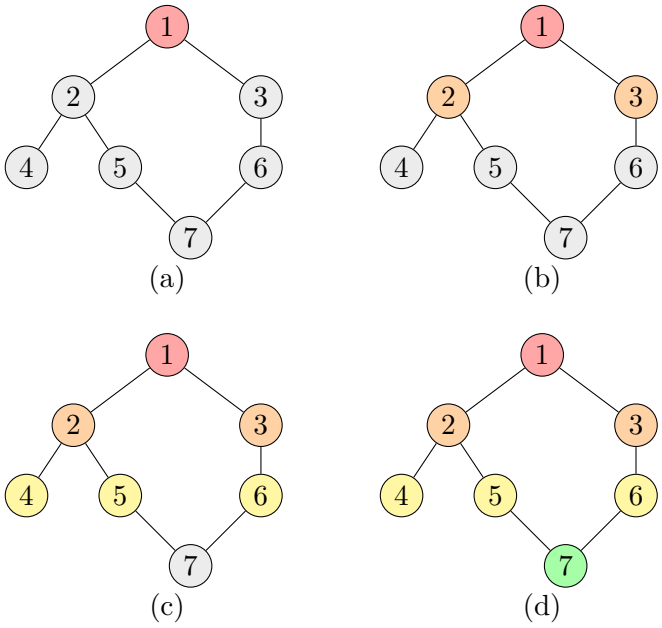
A Figura 7 ilustra a progressão da BFS a partir de $s = 1$ sobre um grafo com sete vértices. Cada quadro acrescenta os vértices descobertos em um novo nível, identificados por uma cor distinta: nível 0 (origem), nível 1, nível 2 e nível 3. Observe que os vértices mais distantes da origem são descobertos por último.

Na Figura 7, a ordem de descoberta dos vértices é $1, 2, 3, 4, 5, 6, 7$. Os vértices 2 e 3 (nível 1) são vizinhos diretos de 1; os vértices 4, 5 e 6 (nível 2) estão a duas arestas de distância; e o vértice 7 (nível 3), o mais afastado, é descoberto por último.

Aplicação: caminhos mínimos.

Ao descobrir um vértice v, registra-se em pai$[v]$ o vértice a partir do qual v foi alcançado. Esse registro define a árvore de descoberta da busca, enraizada em s. Como os vértices são descobertos em ordem crescente de nível, o caminho dessa árvore de s até qualquer vértice v usa o menor número possível de arestas; assim, basta seguir os ponteiros pai de v até s para recuperar um caminho mínimo em grafos não ponderados (SKIENA; REVILLA, 2003).

Figura 7 – Progressão da BFS a partir do vértice 1; cada cor representa um nível.



Fonte: o Autor.

Busca em profundidade

> **Definição 7** (Busca em profundidade, *Depth-First Search* (DFS))**.** *Seja $G = (V, E)$ um grafo e $s \in V$ um vértice de origem. A busca em profundidade avança o máximo possível ao longo de cada ramo antes de retroceder: ao descobrir um vértice ainda não visitado, a busca prossegue imediatamente por um de seus vizinhos não visitados, retornando ao vértice anterior apenas quando todos os seus vizinhos já tiverem sido descobertos (CORMEN et al., 2009).*

Diferentemente da BFS, a DFS alcança rapidamente vértices distantes da origem, pois aprofunda-se em um ramo antes de explorar alternativas. Sua formulação natural é recursiva, como mostra o Código 4; a recursão (ou, equivalentemente, uma pilha explícita) é um detalhe de implementação (SKIENA; REVILLA, 2003).

Código 4 – Implementação em C++ da busca em profundidade (DFS).

```cpp
const int MAX{200010};

bitset<MAX> visited;
vector<int> adj[MAX];

void dfs(int u) {
    if (visited[u])
        return;

    // processa/visita u

```

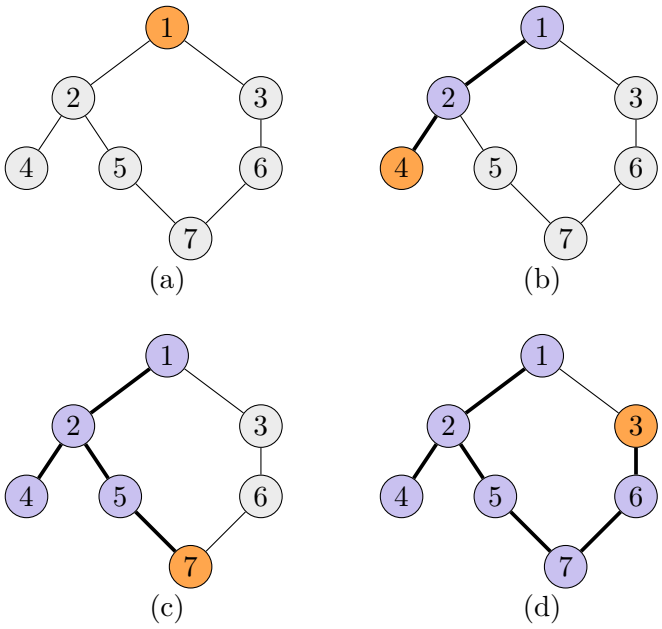
```
12      visited[u] = true;
13
14      for (auto v : adj[u])
15          dfs(v);
16  }
```

Fonte: Alves (2026).

Assim como a BFS, a DFS tem complexidade assintótica de tempo de execução $O(n + m)$ quando o grafo é representado por listas de adjacências, uma vez que visita cada vértice no máximo uma vez e percorre cada aresta um número constante de vezes (CORMEN et al., 2009).

A Figura 8 mostra a progressão da DFS a partir de $s = 1$ no mesmo grafo. Os vértices já visitados aparecem em azul e o vértice recém-descoberto em destaque (laranja); as arestas da árvore de descoberta são realçadas em traço espesso. Note como a busca desce até um vértice profundo antes de retroceder.

Figura 8 – Progressão da DFS a partir do vértice 1; em laranja, o vértice recém-descoberto.



Fonte: o Autor.

No exemplo, a ordem de descoberta da DFS é $1, 2, 4, 5, 7, 6, 3$, distinta da ordem produzida pela BFS no mesmo grafo $(1, 2, 3, 4, 5, 6, 7)$. A diferença evidencia o contraste entre as duas estratégias: a BFS expande-se por níveis, enquanto a DFS aprofunda-se em um ramo até esgotá-lo antes de explorar os demais.

Aplicação: detecção de ciclos.

A DFS sobre um grafo não direcionado classifica cada aresta em dois tipos: as arestas de árvore, que descobrem novos vértices (em destaque na Figura 8), e as arestas de retorno, cuja outra extremidade é um ancestral do vértice em exploração. Toda aresta de retorno fecha um ciclo com o caminho da árvore entre seus extremos; logo, um grafo é acíclico se, e somente se, sua DFS não produz nenhuma aresta de retorno (SKIENA; REVILLA, 2003). No grafo da Figura 8, a aresta $\{1, 3\}$, que não pertence à árvore de descoberta, é uma aresta de retorno: ela liga 3 ao seu ancestral 1 e fecha o ciclo $1, 2, 5, 7, 6, 3, 1$.

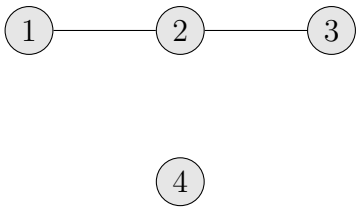
Aplicação: componentes conexas.

Antes de tratar dessa aplicação, é preciso definir os conceitos de conectividade e de componente.

> **Definição 8** (Conectividade e componentes)**.** *Um grafo não direcionado $G = (V, E)$ é conexo se, para todo par $u, v \in V$, existe um caminho ligando u a v. Uma componente conexa é um subconjunto maximal de vértices em que quaisquer dois estão ligados por algum caminho. Em grafos direcionados, G é fortemente conexo se, para todo par $u, v \in V$, há um caminho de u a v e outro de v a u (DIESTEL, 2017).*

Como a ordem de visita é irrelevante para esse fim, as componentes conexas podem ser obtidas com BFS ou DFS: inicia-se uma busca a partir de um vértice qualquer e todos os vértices alcançados formam uma componente; repete-se o processo a partir de algum vértice ainda não descoberto, e assim por diante, até esgotar V (SKIENA; REVILLA, 2003). A Figura 9 mostra um grafo com $V = \{1, 2, 3, 4\}$ e $E = \{\{1, 2\}, \{2, 3\}\}$: o vértice 4 está isolado; logo, G não é conexo e possui duas componentes, $\{1, 2, 3\}$ e $\{4\}$.

Figura 9 – Grafo não conexo, com duas componentes conexas.



Fonte: o Autor.

2.3 A Biblioteca Testlib

A Testlib, desenvolvida por Mirzayanov (2005), é uma biblioteca escrita em C++ utilizada em competições de programação para a construção de geradores, validadores e

checkers de casos de teste. Em problemas com entradas volumosas (por exemplo, vetores com dezenas ou centenas de milhares de elementos, ou grafos com muitos vértices) torna-se impraticável compor manualmente todas as instâncias de teste; nesses cenários, empregam-se *geradores*, programas auxiliares que produzem entradas válidas a partir de parâmetros de linha de comando.

Existem dois tipos principais de geradores (MIRZAYANOV, 2015). O *gerador de teste único* emite exatamente um caso por execução, geralmente na saída padrão; para obter vários testes, executa-se o programa repetidas vezes com parâmetros distintos. O *gerador de múltiplos testes*, por sua vez, produz vários casos em uma única execução, redirecionando cada um para um arquivo distinto. Neste trabalho, empregam-se geradores de teste único.

Embora seja possível escrever geradores sem biblioteca dedicada, a prática recomendada é evitar funções como `rand()` ou geradores da biblioteca padrão do C++ (por exemplo, `mt19937`), pois a sequência pseudoaleatória pode variar conforme o compilador e a plataforma. Um princípio básico na elaboração de geradores é que a mesma linha de comando deve produzir o mesmo teste em qualquer ambiente de compilação (MIRZAYANOV, 2015). A Testlib atende a esse requisito por meio do objeto `rnd`, inicializado a partir de um *hash* dos argumentos de linha de comando após `registerGen(argc, argv, 1)`, em que o terceiro parâmetro indica a versão do gerador pseudoaleatório.

O Código 5 apresenta um gerador elementar: recebe três parâmetros (o tamanho `n` e os extremos `a` e `b` de um intervalo) e imprime uma sequência de `n` inteiros aleatórios em $[a, b]$. O Quadro 2 resume as funções da Testlib efetivamente empregadas nos geradores deste trabalho.

Código 5 – Gerador elementar de sequência de inteiros com Testlib.

```
#include "testlib.h"

#include <iostream>

using namespace std;

int main(int argc, char* argv[]) {
    registerGen(argc, argv, 1);

    int n = opt<int>(1);
    int a = opt<int>(2);
    int b = opt<int>(3);

    println(n);
    for (int i = 0; i < n; ++i) {
        cout << rnd.next(a, b);
        cout << (i + 1 < n ? ' ' : '\n');
    }

```

```
20     return 0;
21 }
```

Fonte: o Autor.

Quadro 2: Funções da Testlib empregadas nos geradores deste trabalho

Função	**Descrição**
`registerGen(argc, argv, 1)`	Registra o programa como gerador e inicializa o gerador pseudoaleatório `rnd`
`opt<T>(i)`	Lê o i-ésimo argumento posicional da linha de comando, convertido para o tipo `T`
`rnd.next(a, b)`	Retorna inteiro equiprovável no intervalo $[a, b]$
`rnd.wnext(k, t)`	Retorna inteiro com distribuição enviesada; para $t > 0$, equivale ao máximo de t chamadas a `rnd.next(k)`
`rnd.perm(n, d)`	Retorna permutação aleatória de n elementos; o parâmetro d fixa o índice inicial (0 ou 1)
`rnd.partition(p, s)`	Particiona o inteiro s em p partes positivas de soma s
`shuffle(begin, end)`	Embaralha o intervalo [`begin`, `end`) de forma reproduzível, usando `rnd`
`println(...)`	Imprime argumentos separados por espaço e encerra a linha

Fonte: o Autor.

A Testlib também oferece suporte a validadores de entrada (`registerValidation`) e a *checkers* de saída (`registerTestlibCmd`), além de geradores de múltiplos testes (`startTest`). Esses papéis são comuns no ecossistema Polygon, plataforma de preparação de problemas utilizada em juízes como o Codeforces, mas não são empregados neste trabalho, que se concentra na geração automatizada de instâncias de grafos notáveis. Exemplos concretos de geradores de grafos com a classe `Graph` aparecem no Capítulo 4; o Código 5 serve apenas para introduzir o registro do gerador e a API `opt`/`rnd`.

Outras bibliotecas C++ seguem propósito semelhante, com ênfase na geração de casos de teste e em padrões recorrentes de instâncias, inclusive de grafos. O jngen (SMIRNOV,) é voltado a problemas de olimpíadas e oferece geradores de grafos e árvores (por exemplo, grafos aleatórios, árvores, bambu e estrela), além de utilitários para permutações, sequências, geometria e impressão formatada da saída. O tgen (MALETTA,) adota abordagem declarativa em um único *header*: o desenvolvedor descreve restrições (listas distintas, permutações com ciclos prescritos, árvores enviesadas, grafos conexos com número fixo de arestas) e a biblioteca amostra instâncias válidas, com suporte adicional a geração adversarial para casos extremos de algoritmos conhecidos. Ambas diferem da Testlib por se concentrarem na geração de dados estruturados, enquanto a Testlib

permanece mais genérica e integrada ao ecossistema Polygon (geradores, validadores e *checkers*).

2.4 Trabalhos Correlatos

Esta seção apresenta trabalhos da literatura que dialogam com os objetivos deste TCC: a identificação de casos de borda em algoritmos sobre grafos e a geração sistemática de instâncias de teste em contexto de programação competitiva. Foram selecionados estudos que tratam tanto da caracterização de situações extremas quanto de estratégias de validação de implementações em problemas envolvendo estruturas de grafos.

Na mesma instituição e orientação deste trabalho, Mattioli (2018) compara algoritmos guloso e de programação dinâmica para o problema do troco em sistemas de moedas gerados pseudoaleatoriamente, com implementações em C++. O estudo evidencia que a correção do algoritmo guloso depende de uma propriedade estrutural da entrada (a canonicidade do sistema de moedas), de modo que instâncias não canônicas funcionam como casos de borda capazes de invalidar uma estratégia aparentemente correta. Embora o domínio não seja o de grafos, a abordagem de avaliar algoritmos mediante geração sistemática de entradas com propriedades controladas dialoga diretamente com a proposta deste TCC.

O trabalho de Vater, Tanneberg e Maurer (2023) apresenta uma abordagem sistemática para a definição de *edge cases* e *corner cases* (casos de borda e casos extremos, na terminologia deste trabalho) no contexto de direção automatizada. Embora o domínio de aplicação seja distinto, o estudo compartilha com este trabalho a preocupação com situações que fogem do comportamento comum e que, se negligenciadas, podem comprometer a confiabilidade de sistemas. Enquanto os autores propõem uma taxonomia para fortalecer a segurança de sistemas embarcados em veículos autônomos, o presente estudo aplica a mesma ideia de rigor a problemas de grafos em competições, com o objetivo de detectar e explorar casos de borda para garantir a robustez das implementações.

Singh et al. (2025) investigam o *stress testing* em programação olímpica, técnica adotada por competidores experientes para validar soluções otimizadas contra implementações de referência em grande volume de entradas aleatórias. Os autores formalizam essa prática como estratégia de filtragem de candidatos incorretos e demonstram, no benchmark USACO, que a comparação sistemática entre soluções eficientes e soluções força bruta aumenta a taxa de acerto. Esse enfoque reforça a motivação deste trabalho ao evidenciar que casos de teste bem escolhidos, e não apenas entradas aleatórias genéricas, são decisivos para expor falhas em implementações de algoritmos sobre grafos. No presente TCC, essa preocupação materializa-se na geração estruturada de instâncias pelos geradores da classe `Graph` e nas discussões qualitativas do Capítulo 5.

Por fim, Yan et al. (2025) propõem o GraphFuzz, um *framework* de *fuzzing* orientado a feedback para testar implementações de algoritmos de grafos. O estudo destaca que o espaço de entradas em grafos é vasto e que poucos grafos gerados automaticamente já cobrem o código de muitas implementações, o que torna insuficiente depender apenas de cobertura de código ou de testes aleatórios. A abordagem combina *differential testing* com sinais de feedback específicos de algoritmos de grafos para produzir instâncias diversas capazes de revelar falhas lógicas e de execução. Essa perspectiva complementa a proposta deste trabalho ao mostrar que a construção deliberada de grafos com propriedades estruturais definidas, como as famílias tratadas aqui, constitui caminho promissor para fortalecer a validação de soluções em programação competitiva.

3 Materiais e Métodos

Este capítulo descreve os materiais e os procedimentos empregados no desenvolvimento do trabalho. A Seção 3.1 apresenta o levantamento bibliográfico, as bases consultadas e os critérios de seleção das obras. A Seção 3.2 explica como foram escolhidos os grafos notáveis e os problemas analisados, antecipando o conteúdo dos Capítulos 4 e 5. A Seção 3.3 comenta o processo de elaboração de cada bloco do texto. Por fim, a Seção 3.4 reúne as ferramentas utilizadas, a metodologia dos grafos notáveis, o procedimento da análise de casos de borda e a declaração de uso de inteligência artificial.

3.1 Pesquisa bibliográfica

Para a realização deste trabalho, foi conduzido um levantamento bibliográfico com o objetivo de identificar materiais que abordassem geração de casos de teste, casos de borda em algoritmos e estruturas de grafos utilizadas em programação competitiva. As bases de busca consultadas foram o Google Scholar e a Scopus. Complementarmente, foram examinados livros técnicos de algoritmos e de teoria dos grafos, documentação de bibliotecas de geração de testes e repositórios de código associados a competições.

Os termos de busca privilegiaram combinações em inglês alinhadas ao vocabulário da área. No Google Scholar, as consultas foram feitas com a frase entre aspas (correspondência aproximada da expressão completa); na Scopus, empregou-se a combinação booleana indicada. A Tabela 1 resume, para cada consulta, a ordem de grandeza dos resultados reportados pelo motor de busca (valores aproximados, registrados na ocasião do levantamento) e a quantidade de páginas de resultados efetivamente examinadas na triagem inicial. Esses totais não correspondem ao número de obras lidas: servem apenas para documentar a amplitude aparente de cada consulta e a necessidade de filtragem manual.

Tabela 1 – Consultas realizadas nas bases acadêmicas

Base	**Termo de busca**	**Resultados (ordem)**	**Pág. analisadas**
Google Scholar	“graph corner cases” (frase)	$\sim 10^5$–10^6	4
Google Scholar	“graph edge cases” (frase)	$\sim 10^6$	3
Scopus	corner AND cases AND graph AND algorithms	73	3

Fonte: o Autor.

A consulta “graph corner cases” retornou volume elevado de resultados, porém a maior parte situava-se em domínios distintos da programação competitiva, como segurança

veicular e direção automatizada. A expressão "graph edge cases" mostrou-se ainda menos específica, pois o termo *edge* coincide com a nomenclatura usual de arestas em grafos, o que dilui a relevância dos resultados. Na Scopus, a combinação booleana restringiu o conjunto a dezenas de trabalhos; mesmo assim, muitos tratavam de problemas do mundo real apenas modelados por grafos, sem ênfase em casos de borda de implementações algorítmicas.

Diante desse cenário, adotaram-se critérios explícitos para selecionar as obras a serem analisadas em profundidade: (i) aderência ao tema de casos de borda, geração de testes ou grafos em programação competitiva; (ii) preferência por livros clássicos e manuais da área como fontes primárias de definições e algoritmos; (iii) inclusão de artigos e trabalhos acadêmicos que dialogassem diretamente com a identificação de situações extremas ou com a validação sistemática de implementações. Com base nesses critérios, foram priorizados, entre outros, Cormen et al. (2009), Diestel (2017), West (2001), Laaksonen (2018) e Halim, Halim e Effendy (2020) para a fundamentação, além dos trabalhos correlatos discutidos na Seção 2.4.

3.2 Seleção de grafos notáveis e de problemas

Antes de apresentar as famílias de grafos e as análises de problemas nos capítulos seguintes, convém explicitar onde os conceitos e os enunciados foram localizados e quais critérios orientaram a inclusão ou a exclusão de cada item.

Os conceitos de grafos notáveis foram localizados em três frentes complementares. Na literatura clássica de teoria dos grafos, buscaram-se definições formais e propriedades estruturais em obras como as de West (2001), Diestel (2017) e Brandstädt, Le e Spinrad (1999). Na literatura de programação competitiva, observaram-se as famílias recorrentes em manuais e repositórios didáticos, em especial Laaksonen (2018) e Halim, Halim e Effendy (2020). Por fim, examinaram-se bibliotecas de geração de testes, como jngen (SMIRNOV,) e tgen (MALETTA,), e discussões de elaboradores de problemas sobre instâncias que tensionam implementações.

Adotou-se como critério de inclusão a conjunção de três condições: a estrutura é comum em problemas de programação competitiva; aparece, de forma explícita ou análoga, em geradores especializados ou em materiais de treinamento; e suas propriedades estruturais costumam originar casos de borda relevantes (por exemplo, densidade extrema, caminhos longos, hierarquias em árvore ou presença de ciclos direcionados). O Quadro 3 resume as dez famílias selecionadas.

Quadro 3: Grafos notáveis selecionados neste trabalho

Nž	**Nome**	**Resumo**
1	Completo	Todo par de vértices distintos é adjacente; densidade máxima.
2	Clique	Subconjunto de vértices dois a dois adjacentes.
3	Bipartido	Vértices particionáveis em dois conjuntos independentes.
4	Árvore	Grafo conexo acíclico.
5	Caminho (*bambu*)	Árvore linear; diâmetro máximo para n vértices.
6	Ciclo	Grafo 2-regular conexo; único ciclo.
7	Estrela	Árvore com um vértice central adjacente a todos os demais.
8	Floresta	Grafo acíclico; componentes conexas são árvores.
9	Sucessores	Cada vértice tem grau de saída igual a um.
10	DAG	Grafo direcionado acíclico.

Fonte: o Autor.

Nem toda noção frequente na teoria dos grafos foi elevada à condição de grafo notável neste trabalho. Grafos esparsos, por exemplo, foram estudados em rascunhos iniciais, mas a esparsidade é tratada aqui como característica transversal de densidade, aplicável a várias famílias, e não como classe estrutural autônoma com gerador próprio no Capítulo 4. De modo análogo, classes como grafos planares ou eulerianos, embora clássicas, foram deixadas de fora por aparecerem com menor frequência como famílias explícitas de casos de borda nas competições e nas bibliotecas de geração consideradas. Características como pesos nas arestas, conectividade e densidade foram, em vez disso, incorporadas à fundamentação teórica e às discussões transversais dos problemas.

Os problemas analisados foram extraídos de três fontes principais: o CSES Problem Set, escolhido pelo caráter educacional e pela cobertura de tópicos clássicos de grafos; a Maratona de Programação da SBC (etapa Sub-Regional Brasil do ICPC), escolhida pela relevância no contexto brasileiro e pela disponibilidade de enunciados com restrições realistas de competição; e a Olimpíada Brasileira de Informática (OBI), escolhida pelo caráter formativo e pela presença de enunciados acessíveis com modelagem explícita por grafos. A referência completa de cada enunciado permanece no Capítulo 5; aqui registra-se apenas a plataforma e o identificador.

A seleção privilegiou enunciados em que a modelagem por grafos é central e em que ao menos uma família do Quadro 3 (ou uma característica transversal associada, como conectividade ou pesos) pudesse ser explorada na discussão de casos de borda. O processo foi iterativo: quando o pareamento entre o enunciado e o gerador correspondente da classe `Graph` não se sustentava de forma natural, o problema era substituído por outro mais

alinhado à família em questão. Problemas puramente aritméticos, de geometria ou de estruturas de dados sem modelagem explícita por grafos foram descartados, ainda que disponíveis em grande volume nas mesmas plataformas. O Quadro 4 apresenta os dez problemas retidos.

Quadro 4: Problemas selecionados para análise de casos de borda

Nž	**Nome**	**Plataforma**	**ID**
1	Building Teams	CSES	1668
2	Course Schedule	CSES	1679
3	Caminhada na Montanha	Maratona SBC	2023-C
4	Empresa de Festas	Maratona SBC	2020-E
5	Boss	Maratona SBC	2013-C
6	Trucks	Maratona SBC	2013-J
7	Round Trip	CSES	1669
8	Sanduíche	OBI	2021-f2
9	Building Roads	CSES	1666
10	Planets Cycles	CSES	1751

Fonte: o Autor.

3.3 Processo de desenvolvimento dos capítulos

Esta seção descreve, de forma sintética, o processo seguido na elaboração dos principais blocos do trabalho.

3.3.1 Fundamentação teórica

A fundamentação teórica foi organizada em três eixos. No eixo de grafos, priorizaram-se definições, representações e travessias, além de características transversais que atravessam várias famílias, como pesos nas arestas, conectividade e densidade. No eixo de programação competitiva, construiu-se uma base sobre o funcionamento das competições e das plataformas, com exemplos de estrutura de problema e com a apresentação de um enunciado da OBI, competição formativa relevante no contexto brasileiro. No eixo da Testlib, apresentou-se a biblioteca empregada nos geradores do Capítulo 4, com exemplo de código e documentação das funções efetivamente utilizadas.

3.3.2 Grafos notáveis

Para cada grafo notável, o desenvolvimento acompanhou as etapas da metodologia da Seção 3.4: busca da definição formal em livros, elaboração de uma ilustração em LaTeX com TikZ, implementação do gerador correspondente na classe `Graph` com apoio da Testlib e redação do texto com propriedades e relevância para casos de borda. Os geradores destinam-se a elaboradores de problemas (*setters*) e a competidores que desejem construir instâncias de teste de forma sistemática.

3.3.3 Análise de casos de borda em problemas de grafos

No Capítulo 5, selecionaram-se problemas de grafos e, com os conceitos estudados na fundamentação e nos grafos notáveis em mente, buscou-se identificar casos de borda relevantes para cada enunciado. Cada problema foi resumido, modelado, implementado e discutido sob essa ótica; a solução foi validada quanto à entrada e à saída nas situações extremas consideradas. Os critérios de seleção dos enunciados constam na Seção 3.2.

3.4 Ferramentas e metodologia de desenvolvimento

No desenvolvimento prático, utilizou-se o Overleaf para escrita, edição e compilação do texto em LaTeX, o Git para controle de versões local, o GitHub como repositório remoto e plataforma de revisão por *pull requests*. A biblioteca Testlib (MIRZAYANOV, 2005) foi utilizada na implementação dos geradores de grafos notáveis da classe `Graph`. Os editores Vim e Visual Studio Code foram empregados na implementação dos códigos em C++ e na manutenção dos arquivos do trabalho. A compilação local do documento também foi realizada em ambiente Docker com TeX Live. As respectivas versões de cada ferramenta, assim como um breve resumo de seu principal uso, estão sintetizadas no Quadro 5.

Quadro 5: Ferramentas utilizadas no desenvolvimento

Ferramenta	**Versão**	**Uso principal**
Overleaf	online	Escrita e compilação colaborativa em LaTeX
Git	2.34	Controle de versões local
GitHub	–	Repositório remoto e revisão por *pull requests*
Testlib	`testlib.h`	Apoio aos geradores da classe `Graph`
Vim	8.2	Implementação manual das soluções em C++
Visual Studio Code	1.85+	Edição de código e arquivos do trabalho
g++	11.4	Compilação dos geradores e das soluções
Docker / TeX Live	2024	Compilação reproduzível do PDF

Fonte: o Autor.

Quanto à metodologia de desenvolvimento dos grafos notáveis, adotou-se o ciclo incremental ilustrado na Figura 10. As etapas são as seguintes.

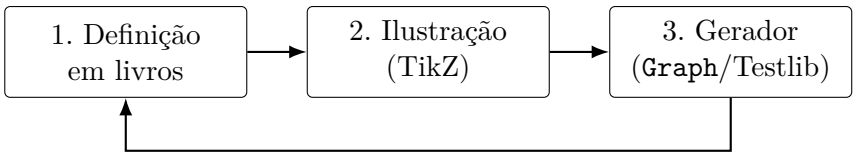
(1) **Definição.** Busca-se a definição formal do grafo em livros de teoria dos grafos e em manuais de programação competitiva, registrando propriedades estruturais relevantes para casos de borda.

(2) **Ilustração.** Elabora-se uma figura representativa em LaTeX com TikZ, com vértices numerados de 1 a n, para fixar a intuição visual da família.

(3) **Gerador.** Implementa-se o método correspondente na classe `Graph`, com apoio da Testlib, como artefato reutilizável para elaboradores de problemas (*setters*) e competidores.

Quando a definição, a figura ou o gerador se mostram inconsistentes entre si, retorna-se à etapa pertinente, fechando o ciclo da Figura 10.

Figura 10 – Ciclo incremental de desenvolvimento dos grafos notáveis



Fonte: o Autor.

Já na análise de casos de borda (Capítulo 5), o procedimento foi distinto: selecionaram-se problemas de grafos e, tendo em mente os conceitos da fundamentação e das famílias estudadas, identificaram-se casos de borda capazes de tensionar cada solução. Em seguida, as saídas esperadas para as instâncias ilustrativas foram obtidas por execução local das implementações apresentadas no capítulo, confrontadas, quando disponíveis, com os exemplos oficiais dos enunciados. Para cada família do Quadro 3, o Capítulo 5 inclui ainda um exemplo de geração com o método correspondente da classe `Graph`, ilustrando como elaboradores de problemas (*setters*) e competidores podem construir instâncias de teste de forma sistemática.

No apoio à redação, utilizaram-se recursos de inteligência artificial generativa integrados ao Overleaf e modelos de linguagem da OpenAI (ChatGPT), em conformidade com o dever de declaração previsto na Portaria CNPq nž 2.664/2026 (Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), 2026). Essas ferramentas foram empregadas para revisão textual, reestruturação de trechos, apoio à escrita e formatação em LaTeX, elaboração de esboços de figuras e tabelas e tradução. Estima-se que cerca de um terço do texto tenha contado com apoio de modelos de linguagem em alguma etapa (rascunho, reestruturação ou revisão), cabendo ao autor a revisão integral, a validação do conteúdo e a responsabilidade pelo resultado final, inclusive por eventuais imprecisões introduzidas pelas ferramentas.

4 Grafos Notáveis

Neste capítulo, apresentam-se os grafos notáveis utilizados ao longo do trabalho. Denominam-se grafos notáveis as famílias e estruturas de grafos que ocorrem com frequência em problemas de programação competitiva e que, por suas propriedades estruturais, costumam constituir casos de borda relevantes para a correção e o desempenho das implementações que os manipulam. O termo, de caráter didático, reúne sob um mesmo rótulo tanto classes clássicas da teoria dos grafos quanto características estruturais recorrentes em competições. Na literatura de teoria dos grafos, objetos dessa natureza são usualmente estudados sob a designação de classes ou famílias de grafos (BRANDSTÄDT; LE; SPINRAD, 1999; WEST, 2001).

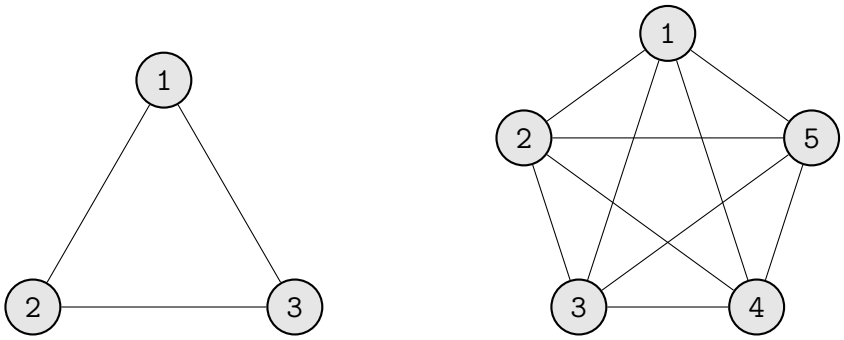
A opção por agrupá-los sob o rótulo de grafos notáveis justifica-se pelo objetivo deste trabalho: mais do que classificá-los formalmente, interessa reconhecer quais estruturas surgem com frequência nos enunciados e de que maneira elas tensionam os limites das implementações. Para cada grafo notável, apresentam-se a definição formal, exemplos visuais e métodos para a geração automática de instâncias com a biblioteca Testlib, viabilizando a construção sistemática de testes. As famílias tratadas são as do Quadro 3.

4.1 Grafo Completo

> **Definição 9** (Grafo Completo)**.** *Um grafo é dito completo se todo vértice é adjacente a todo outro vértice. Consequentemente, se um grafo contém ao menos um par de vértices não adjacentes, então o grafo não é completo (HARRIS; HIRST; MOSSINGHOFF, 2008). O grafo completo com n vértices é denotado por* K_n*.*

A Figura 11 apresenta, de acordo com a Definição 9, as visualizações de grafos não direcionados completos com 3 e 5 vértices, respectivamente.

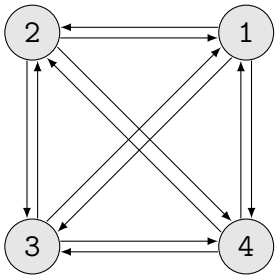
Figura 11 – Exemplos de grafos completos não direcionados com 3 e 5 vértices.



Fonte: o Autor.

A Figura 12 apresenta, de acordo com a Definição 9, as visualizações de um grafo direcionado completo com 4 vértices.

Figura 12 – Exemplo de grafo completo direcionado com 4 vértices.



Fonte: o Autor.

Em um grafo completo K_n, cada um dos n vértices está conectado aos outros $n-1$. Se K_n for não direcionado, então

$$|E(K_n)| = \frac{n(n-1)}{2}, \tag{4.1}$$

caso contrário, se K_n é direcionado, então

$$|E(K_n)| = n(n-1). \tag{4.2}$$

O Código 6 apresenta a implementação do método `complete` da classe `Graph`, que gera um grafo completo com n vértices. Se o grafo for não direcionado, o método percorre todos os pares (u, v) com $1 \leq u < v \leq n$ e insere cada aresta uma única vez por meio de `addEdge`. Se for direcionado, insere uma aresta direcionada para cada par ordenado distinto (u, v). A construção tem complexidade $O(n^2)$, em conformidade com a quantidade de arestas de K_n.

Código 6 – Implementação do método `complete` da classe `Graph`.

```cpp
/**
 * Grafo completo K_n.
 * @param n número de vértices (>= 1)
 * @param directed se verdadeiro, gera arestas direcionadas em ambos os sentidos entre pares distintos
 */
inline Graph Graph::complete(int n, bool directed) {
    if (n < 1)
        quitf(_fail, "Graph::complete requires n >= 1, got %d", n);

    Graph g(n, directed);

    if (directed) {
        for (int u = 1; u <= n; ++u)
            for (int v = 1; v <= n; ++v)
                if (u != v)
                    g.addEdge(u, v);
    } else {
        for (int u = 1; u <= n; ++u)
            for (int v = u + 1; v <= n; ++v)
                g.addEdge(u, v);
    }

    return g;
}
```

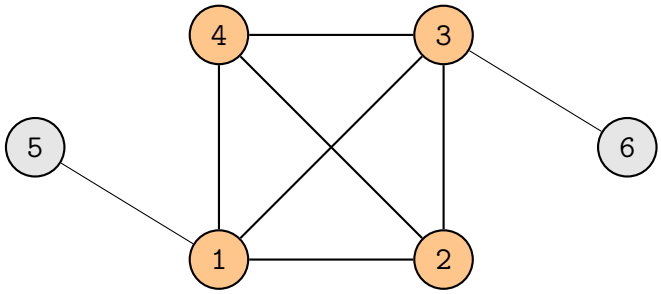
Fonte: o Autor.

4.2 Clique

> **Definição 10** (Clique)**.** *Uma clique em um grafo é um conjunto de vértices dois a dois adjacentes (WEST, 2001). Em outras palavras, todo par de vértices distintos desse conjunto está ligado por uma aresta. Uma clique de tamanho k (com k vértices) é chamada neste trabalho de k-clique.*

A Figura 13 ilustra um grafo com $n = 6$ vértices que contém uma 4-clique: os vértices $1, 2, 3$ e 4 (destacados) são dois a dois adjacentes, enquanto 5 e 6 apenas se ligam a esse conjunto. O grafo, portanto, não se reduz a um completo K_6; a clique é um subconjunto de V.

Figura 13 – Grafo com 6 vértices contendo uma 4-clique destacada (vértices 1 a 4).



Fonte: o Autor.

Cliques aparecem em problemas de teoria dos grafos e otimização combinatória, especialmente em tarefas de detecção de subestruturas densas e análise de conectividade entre grupos de vértices.

O Código 7 apresenta a implementação do método `withClique` da classe `Graph`, que gera um grafo conexo com n vértices contendo uma k-clique como subgrafo. Primeiro, aplica-se uma permutação aleatória dos rótulos $1, \ldots, n$; em seguida, conectam-se todos os pares entre os k primeiros vértices dessa ordem, produzindo K_k sobre um subconjunto de V. Quando $k < n$, cada vértice restante é ligado a um vértice já conectado sorteado uniformemente entre os anteriores na permutação, de modo que o grafo permaneça conexo sem se reduzir a um único caminho. Quando $k = n$, o resultado coincide com o grafo completo K_n gerado por `complete`.

Código 7 – Implementação do método `withClique` da classe `Graph`.

```
368 /**
369  * Grafo conexo com n vertices contendo uma k-clique.
370  * @param n número de vértices (>= 1)
371  * @param k tamanho da clique (1 <= k <= n)
372  */
373 inline Graph Graph::withClique(int n, int k) {
374     if (n < 1)
375         quitf(_fail, "Graph::withClique requires n >= 1, got %d", n);
376     if (k < 1 || k > n)
377         quitf(_fail, "Graph::withClique requires 1 <= k <= n, got k=%d, n=%d", k, n);
378
379     Graph g(n, false);
380     std::vector<int> order = rnd.perm(n, 1);
381
382     for (int i = 0; i < k; ++i)
383         for (int j = i + 1; j < k; ++j)
384             g.addEdge(order[i], order[j]);
385
386     for (int i = k; i < n; ++i)
387         g.addEdge(order[i], order[rnd.next(0, i - 1)]);
388
389     return g;
390 }
```

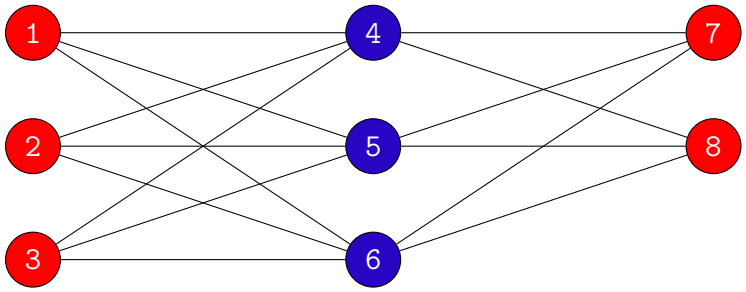
Fonte: o Autor.

4.3 Grafo Bipartido

> **Definição 11** (Grafo Bipartido)**.** *Um grafo bipartido é um grafo cujos vértices podem ser divididos em dois conjuntos de modo que todas as arestas conectem um vértice de um conjunto a um vértice do outro conjunto (SEDGEWICK; WAYNE, 2011).*

A Figura 14 ilustra um grafo bipartido cujos vértices estão divididos em dois conjuntos disjuntos: U (em vermelho) e V (em azul). Os vértices de U estão posicionados nas colunas das extremidades, enquanto os de V aparecem na coluna central. Todas as arestas conectam apenas vértices de conjuntos diferentes, como exige a Definição 11.

Figura 14 – Exemplo de grafo bipartido com os conjuntos de vértices representados pelas cores vermelha e azul.



Fonte: o Autor.

O Código 8 apresenta as implementações dos métodos `completeBipartite` e `randomBipartite` da classe `Graph`. A partição U ocupa os vértices $1, \ldots, n_1$ e a partição V ocupa $n_1{+}1, \ldots, n_1{+}n_2$. O primeiro método gera o grafo bipartido completo K_{n_1,n_2}; o segundo sorteia exatamente m arestas cruzadas por rejeição. A implementação tomou como referência a biblioteca jngen (SMIRNOV,).

Código 8 – Implementações dos métodos bipartidos da classe `Graph`.

```
133 /**
134  * Grafo bipartido completo K_{n1,n2}.
135  * Partição U: 1..n1; partição V: n1+1..n1+n2.
136  * @param n1 tamanho de U (>= 0)
137  * @param n2 tamanho de V (>= 0)
138  */
139 inline Graph Graph::completeBipartite(int n1, int n2) {
140     if (n1 < 0 || n2 < 0)
141         quitf(_fail, "Graph::completeBipartite requires n1, n2 >= 0, got (%d, %d)", n1, n2);
142 
143     Graph g(n1 + n2, false);
```

```
144     for (int u = 1; u <= n1; ++u)
145         for (int v = n1 + 1; v <= n1 + n2; ++v)
146             g.addEdge(u, v);
147 
148     return g;
149 }
150 
151 /**
152  * Grafo bipartido aleatório com exatamente m arestas entre as partições.
153  * @param n1 tamanho de U (>= 0)
154  * @param n2 tamanho de V (>= 0)
155  * @param m número de arestas (0 <= m <= n1*n2)
156  */
157 inline Graph Graph::randomBipartite(int n1, int n2, int m) {
158     if (n1 < 0 || n2 < 0)
159         quitf(_fail, "Graph::randomBipartite requires n1, n2 >= 0, got (%d, %d)", n1, n2);
160     if (m < 0)
161         quitf(_fail, "Graph::randomBipartite requires m >= 0, got %d", m);
162 
163     long long maxEdges = 1LL * n1 * n2;
164     if (m > maxEdges)
165         quitf(_fail, "Graph::randomBipartite: too many edges (%d > %lld)", m, maxEdges);
166 
167     Graph g(n1 + n2, false);
168     if (m == 0)
169         return g;
170 
171     if (n1 == 0 || n2 == 0)
172         quitf(_fail, "Graph::randomBipartite: cannot add edges when n1 or n2 is 0");
173 
174     std::set<std::pair<int, int>> used;
175     while ((int)used.size() < m) {
176         int u = rnd.next(1, n1);
177         int v = rnd.next(n1 + 1, n1 + n2);
178         used.emplace(u, v);
179     }
180 
181     for (const auto& [u, v] : used)
182         g.addEdge(u, v);
183 
184     return g;
185 }
```

Fonte: o Autor.

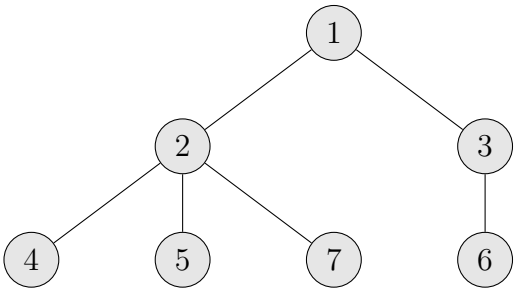
4.4 Árvore

Definição 12 (Árvore). *Uma árvore é um grafo não direcionado, conexo e acíclico. Equivalentemente, é um grafo no qual existe exatamente um caminho simples entre qualquer par de vértices. Ainda equivalentemente, um grafo conexo com exatamente* $n-1$

> *arestas, em que n é o número de vértices, é uma árvore; da mesma forma, um grafo acíclico com exatamente $n-1$ arestas também é uma árvore (DIESTEL, 2017; CORMEN et al., 2009).*

A Figura 15 ilustra um exemplo de árvore com 7 vértices. É possível observar que ela é um grafo conexo e que não existem ciclos. Além disso, qualquer par de vértices está conectado por exatamente um caminho simples.

Figura 15 – Exemplo de árvore com 7 vértices.



Fonte: o Autor.

As árvores são estruturas fundamentais tanto na teoria dos grafos quanto em diversas aplicações da ciência da computação, incluindo algoritmos de busca, compressão de dados, representação de hierarquias e estruturas de dados eficientes. Por serem grafos conexos e acíclicos, árvores possuem a propriedade de conter exatamente $n-1$ arestas, onde n representa o número de vértices.

Uma das propriedades importantes é que, ao remover qualquer aresta de uma árvore, o grafo resultante deixa de ser conexo. Da mesma forma, adicionar qualquer aresta a uma árvore cria necessariamente um ciclo.

O Código 9 apresenta a implementação do método `tree` da classe `Graph`, que gera uma árvore aleatória não direcionada com n vértices. A construção segue um processo análogo ao de Prim: fixam-se índices internos $0, \ldots, n-1$ e, para cada $v = 1, \ldots, n-1$, sorteia-se um pai em $\{0, \ldots, v-1\}$ por meio de `rnd.wnext(v, elongation)`. O parâmetro `elongation` controla o formato: na Testlib, valores positivos fazem `wnext` retornar o *máximo* entre vários sorteios uniformes, favorecendo índices maiores (pais *mais recentes* no processo, próximos de $v-1$) e alongando a estrutura; valores negativos retornam o *mínimo*, favorecendo índices menores (pais *mais antigos*, sobretudo o índice 0) e aproximando uma estrela; o valor 0 corresponde ao sorteio uniforme. Essa estratégia é a mesma adotada por `Tree::randomPrim` na biblioteca jngen (SMIRNOV,).

Antes de inserir as arestas, aplica-se uma permutação aleatória desses índices (`rnd.perm`), de modo que os rótulos $1, \ldots, n$ não fiquem acoplados à ordem de descoberta do processo Prim. Ao final, `fillParentsFromRoot1` preenche o vetor `parent` por busca

em largura a partir do vértice 1 (`parent[1] = 0`): como a permutação pode colocar o rótulo 1 em qualquer posição da construção, essa etapa reenraíza a árvore no vértice 1, de modo que `printParents` exponha sempre essa hierarquia.

Código 9 – Implementação do método `tree` da classe `Graph`.

```
263 /**
264  * Árvore aleatória não direcionada com n vértices (processo tipo Prim / jngen
    ↪  Tree::randomPrim).
265  * Usa rnd.wnext(v, elongation) e permuta índices antes de inserir arestas.
266  * elongation > 0: alongada; < 0: estrelada; 0: uniforme.
267  * @param n número de vértices (>= 1)
268  * @param elongation enviesamento de rnd.wnext
269  * Preenche parent enraizado em 1.
270  */
271 inline Graph Graph::tree(int n, int elongation) {
272     if (n < 1)
273         quitf(_fail, "Graph::tree requires n >= 1, got %d", n);
274
275     Graph g(n, false);
276     std::vector<int> verts(n);
277     for (int i = 0; i < n; ++i)
278         verts[i] = i + 1;
279     addRandomTreeEdges(g, verts, elongation);
280     g.fillParentsFromRoot1();
281     return g;
282 }
```

Fonte: o Autor.

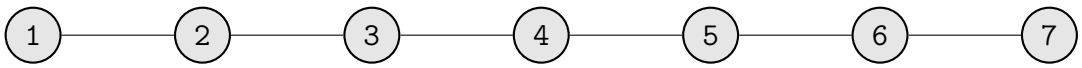
4.5 Grafo Caminho (*Bambu*)

> **Definição 13** (Grafo Caminho)**.** *Um grafo caminho é um grafo simples cujos vértices podem ser ordenados de modo que dois vértices são adjacentes se, e somente se, forem consecutivos nessa lista. O grafo caminho com n vértices é denotado por P_n (WEST, 2001).*

Em programação competitiva, o grafo P_n é frequentemente chamado de *bambu* ou de árvore linear: trata-se de uma árvore (Definição 12) na qual todos os vértices, exceto dois de grau 1, possuem grau 2. Essa estrutura contrasta com o grafo estrela (formalizado adiante na Definição 15), que concentra o grau em um único vértice central.

A Figura 16 ilustra um exemplo de grafo caminho P_7.

Figura 16 – Exemplo de grafo caminho P_7 (*bambu*).



Fonte: o Autor.

Grafos caminho são casos de borda recorrentes em competições. Com n vértices, P_n possui profundidade $O(n)$ em relação a qualquer extremidade, o que pode provocar estouro de pilha em implementações recursivas de busca em profundidade e degradar algoritmos cuja complexidade depende do diâmetro ou da altura da árvore. Por ser a árvore de diâmetro máximo entre as com n vértices, P_n é um dos primeiros formatos utilizados para *stress tests* em problemas que envolvem árvores.

O Código 10 apresenta a implementação do método `bamboo` da classe `Graph`, que gera o grafo caminho P_n não direcionado. Aplica-se uma permutação aleatória dos rótulos $1, \ldots, n$ (`rnd.perm`) e conectam-se os vértices consecutivos nessa ordem. Ao final, preenche-se o vetor `parent` com a árvore enraizada no vértice 1.

Código 10 – Implementação do método `bamboo` da classe `Graph`.

```
204 /**
205  * Grafo caminho P_n (bambu), não direcionado.
206  * Permuta os rótulos 1..n e liga pares consecutivos.
207  * @param n número de vértices (>= 1)
208  * Preenche parent enraizado em 1.
209  */
210 inline Graph Graph::bamboo(int n) {
211     if (n < 1)
212         quitf(_fail, "Graph::bamboo requires n >= 1, got %d", n);
213
214     Graph g(n, false);
215     std::vector<int> order = rnd.perm(n, 1);
216     for (int i = 0; i < n - 1; ++i)
217         g.addEdge(order[i], order[i + 1]);
218     g.fillParentsFromRoot1();
219
220     return g;
221 }
```

Fonte: o Autor.

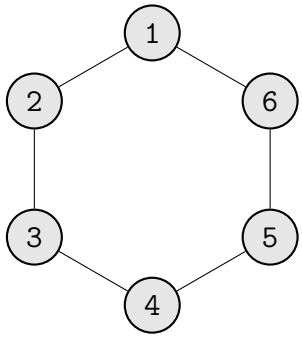
4.6 Grafo Ciclo

> **Definição 14** (Grafo Ciclo)**.** *Um grafo ciclo é um grafo com o mesmo número de vértices e arestas cujos vértices podem ser dispostos ao redor de um círculo de modo que dois vértices são adjacentes se, e somente se, forem consecutivos ao longo do círculo. O grafo ciclo com n vértices é denotado por C_n (WEST, 2001).*

O grafo C_n possui n vértices e n arestas; todo vértice tem grau 2. Em um grafo simples, a igualdade entre o número de vértices e o de arestas implica $n \geq 3$. C_n é o grafo conexo mínimo que contém um ciclo. Em relação à árvore (Definição 12), o ciclo é obtido ao acrescentar uma única aresta entre as extremidades de um caminho P_n, tornando o grafo conexo e não acíclico.

A Figura 17 ilustra um exemplo de grafo ciclo C_6.

Figura 17 – Exemplo de grafo ciclo C_6.



Fonte: o Autor.

Grafos ciclo aparecem com frequência em casos de teste de programação competitiva. Eles testam a detecção de ciclos em travessias, a verificação de bipartição (um ciclo ímpar, como C_3 ou C_5, é o contraexemplo canônico de grafo não bipartido) e algoritmos que assumem, incorretamente, que o grafo é uma árvore ou uma floresta (formalizada adiante na Definição 16).

O Código 11 apresenta a implementação do método `cycle` da classe `Graph`, que gera o grafo ciclo não direcionado C_n. Insere-se a aresta entre cada par de vértices consecutivos e fecha o ciclo entre n e 1. Exige-se $n \geq 3$, valor mínimo compatível com a Definição 14 em grafos simples e com a noção de ciclo de comprimento ao menos três (Definição 2).

Código 11 – Implementação do método `cycle` da classe `Graph`.

```
223 /**
224  * Grafo ciclo C_n não direcionado.
225  * @param n número de vértices (>= 3)
226  */
227 inline Graph Graph::cycle(int n) {
228     if (n < 3)
```

```
229         quitf(_fail, "Graph::cycle requires n >= 3, got %d", n);
230
231     Graph g(n, false);
232
233     for (int i = 1; i < n; ++i)
234         g.addEdge(i, i + 1);
235     g.addEdge(n, 1);
236
237     return g;
238 }
```

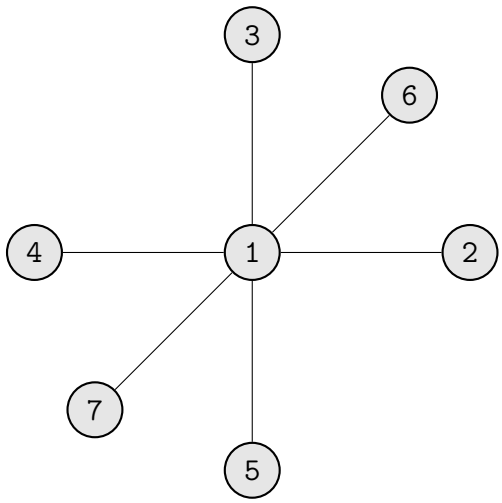
Fonte: o Autor.

4.7 Grafo Estrela

Definição 15 (Grafo Estrela)**.** *Uma estrela é uma árvore constituída por um vértice adjacente a todos os demais. A estrela com n vértices é o grafo bipartido completo $K_{1,n-1}$ (WEST, 2001; DIESTEL, 2017).*

A Figura 18 ilustra um exemplo de grafo estrela com 7 vértices.

Figura 18 – Exemplo de grafo estrela com 7 vértices.



Fonte: o Autor.

Grafos estrela aparecem com frequência em casos de teste de programação competitiva por representarem estruturas com alta concentração de grau em um único vértice. Esse formato é útil para avaliar algoritmos que podem degradar quando há grande desbalanceamento entre os graus dos vértices.

O Código 12 apresenta a implementação do método `star` da classe `Graph`, que gera um grafo estrela não direcionado com n vértices: o vértice 1 é o centro e os vértices $2, \ldots, n$ são as folhas. Ao final, preenche-se o vetor `parent` com a árvore enraizada no vértice 1, no formato $p_2, \ldots, p_n$.

Código 12 – Implementação do método `star` da classe `Graph`.

```
187 /**
188  * Grafo estrela não direcionado: vértice 1 no centro, 2..n folhas.
189  * @param n número de vértices (>= 1)
190  * Preenche parent enraizado em 1.
191  */
192 inline Graph Graph::star(int n) {
193     if (n < 1)
194         quitf(_fail, "Graph::star requires n >= 1, got %d", n);
195 
196     Graph g(n, false);
197     for (int i = 2; i <= n; ++i)
198         g.addEdge(1, i);
199     g.fillParentsFromRoot1();
200 
201     return g;
202 }
```

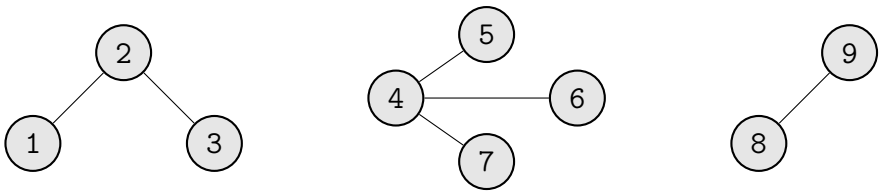
Fonte: o Autor.

4.8 Floresta

> **Definição 16** (Floresta)**.** *Uma floresta é um grafo não direcionado acíclico, isto é, um grafo que não contém ciclos. Assim, uma floresta é um grafo cujas componentes conexas são árvores (WEST, 2001; DIESTEL, 2017).*

A Figura 19 ilustra uma floresta com três componentes conexas, cada uma delas formando uma árvore.

Figura 19 – Exemplo de floresta com três componentes (três árvores).



Fonte: o Autor.

Florestas são úteis para modelar estruturas hierárquicas múltiplas e aparecem em problemas nos quais a entrada pode conter várias componentes desconexas sem ciclos. Em programação competitiva, esse tipo de grafo é importante para testar algoritmos que assumem, incorretamente, que o grafo é conexo.

O Código 13 apresenta a implementação do método `forest` da classe `Graph`, que gera uma floresta não direcionada com n vértices e exatamente c componentes conexas. Primeiro, `rnd.partition(c, n)` sorteia tamanhos positivos $n_1, \ldots, n_c$ que somam n; em seguida, `rnd.perm(n, 1)` define uma ordem aleatória dos rótulos $1, \ldots, n$, repartidos entre as componentes conforme esses tamanhos. Em cada parte, aplica-se o mesmo gerador de árvore aleatória usado pelo método `tree`.

Código 13 – Implementação do método `forest` da classe `Graph`.

```
284 /**
285  * Floresta aleatória não direcionada com n vértices e exatamente components componentes.
286  * Particiona rótulos (rnd.partition + rnd.perm) e gera uma árvore em cada parte.
287  * @param n número de vértices (>= 0)
288  * @param components número de componentes (1 <= components <= n; 0 se n = 0)
289  * @param elongation repassado a cada árvore
290  */
291 inline Graph Graph::forest(int n, int components, int elongation) {
292     if (n < 0)
293         quitf(_fail, "Graph::forest requires n >= 0, got %d", n);
294
295     if (n == 0) {
296         if (components != 0)
297             quitf(_fail, "Graph::forest: n = 0 requires components = 0, got %d", components);
298         return Graph(0, false);
299     }
300
301     if (components < 1 || components > n)
302         quitf(_fail, "Graph::forest requires 1 <= components <= n, got components=%d, n=%d",
303               components, n);
304
305     std::vector<int> sizes = rnd.partition(components, n);
306     std::vector<int> order = rnd.perm(n, 1);
307
308     Graph g(n, false);
309     int offset = 0;
310     for (int size : sizes) {
311         std::vector<int> verts(size);
312         for (int i = 0; i < size; ++i)
313             verts[i] = order[offset + i];
314         offset += size;
315         addRandomTreeEdges(g, verts, elongation);
316     }
317
318     return g;
319 }
```

Fonte: o Autor.

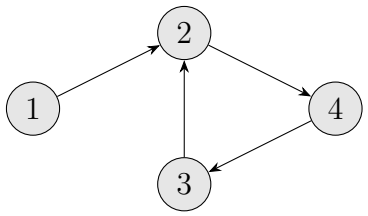
4.9 Grafo de Sucessores

> **Definição 17** (Grafo de Sucessores)**.** *Um grafo de sucessores é um grafo direcionado no qual cada vértice possui exatamente uma aresta de saída, ou seja, seu grau de saída é igual a 1. Dessa forma, cada vértice está conectado a um único sucessor (LAAKSONEN, 2018). Esse tipo de grafo é também chamado de grafo funcional, pois pode ser representado como uma função que associa cada vértice ao seu sucessor.*

Estruturalmente, um grafo de sucessores é composto por uma ou mais componentes, em que cada componente contém exatamente um ciclo, possivelmente com caminhos que levam até esse ciclo.

A Figura 20 apresenta um exemplo de grafo de sucessores com 4 vértices. Note que cada vértice possui exatamente uma aresta de saída, representando seu sucessor direto. É possível observar que o grafo é composto por um ciclo e por caminhos que eventualmente conduzem a esse ciclo.

Figura 20 – Exemplo de grafo de sucessores com 4 vértices. Cada vértice possui exatamente um sucessor.



Fonte: o Autor.

Grafos de sucessores aparecem frequentemente em problemas que envolvem simulações determinísticas, iteração funcional e análise de trajetórias. Devido ao fato de cada vértice possuir exatamente um sucessor, percorrer o grafo se torna equivalente a aplicar repetidamente a função de sucessão.

Uma das propriedades fundamentais desse tipo de grafo é que, inevitavelmente, qualquer sequência de sucessores termina em um ciclo. Isso ocorre porque, sendo finito o número de vértices e existindo exatamente uma aresta de saída por vértice, é garantido que, após um certo número de passos, se atinja um vértice previamente visitado.

Esse comportamento é explorado em algoritmos que envolvem, por exemplo, detecção de ciclos, cálculo de encontros (técnica do ponteiro rápido e lento, *Floyd's cycle detection*) e iteração acelerada. Também é comum definir a função $\text{succ}(x, k)$, que representa o vértice alcançado ao partir de x e percorrer k sucessores consecutivos. Com técnicas

de pré-processamento baseadas em exponenciação binária, é possível computar essa função em $O(\log k)$ (LAAKSONEN, 2018).

O Código 14 apresenta a implementação do método `successors` da classe `Graph`, que gera um grafo funcional com n vértices: para cada rótulo i, sorteia-se uniformemente um sucessor em $\{1, \ldots, n\}$ e insere-se a aresta $i \to \texttt{succ}(i)$.

Código 14 – Implementação do método `successors` da classe `Graph`.

```
321 /**
322  * Grafo de sucessores: cada vértice tem exatamente uma aresta de saída.
323  * @param n número de vértices (>= 1)
324  */
325 inline Graph Graph::successors(int n) {
326     if (n < 1)
327         quitf(_fail, "Graph::successors requires n >= 1, got %d", n);
328 
329     Graph g(n, true);
330     for (int i = 1; i <= n; ++i)
331         g.addEdge(i, rnd.next(1, n));
332 
333     return g;
334 }
```

Fonte: o Autor.

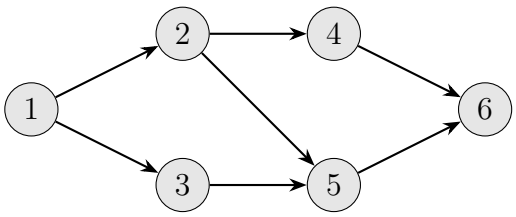
4.10 Grafo Acíclico Direcionado (DAG)

> **Definição 18** (Grafo acíclico direcionado, *Directed Acyclic Graph* (DAG))**.** *Um grafo acíclico direcionado (DAG) é um grafo direcionado que não contém ciclos (HALIM; HALIM; EFFENDY, 2020).*

Em um DAG, as arestas induzem uma relação de precedência entre vértices. Uma ordenação topológica dos vértices é uma disposição linear $v_1, v_2, \ldots, v_n$ tal que, para toda aresta (u, v), u aparece antes de v nessa disposição. Todo DAG admite ao menos uma ordenação topológica; a presença de um ciclo direcionado torna qualquer ordenação desse tipo impossível (HALIM; HALIM; EFFENDY, 2020).

A Figura 21 ilustra um DAG com 6 vértices. Uma ordenação topológica possível é $1, 2, 3, 4, 5, 6$: todas as arestas apontam de um vértice anterior para um posterior nessa sequência.

Figura 21 – Exemplo de grafo acíclico direcionado (DAG) com 6 vértices.



Fonte: o Autor.

DAGs aparecem com frequência em problemas de programação competitiva que envolvem precedência, escalonamento e programação dinâmica. Em escalonamento, uma aresta (u, v) indica que a tarefa u deve ser concluída antes de v; a ordenação topológica fornece, então, uma sequência válida de execução (SKIENA; REVILLA, 2003). Em programação dinâmica, os estados e as transições de uma recorrência formam um DAG implícito: a ausência de ciclos garante que cada subproblema possa ser resolvido uma única vez, após seus pré-requisitos (HALIM; HALIM; EFFENDY, 2020).

Em relação a outras famílias tratadas neste capítulo, o DAG distingue-se do grafo de sucessores (Definição 17): neste, cada vértice tem grau de saída exatamente igual a um e, em grafos finitos, toda trajetória termina em um ciclo; no DAG, o grau de saída é arbitrário e ciclos direcionados são proibidos. Também difere da árvore e da floresta (Definições 12 e 16), que são acíclicas no sentido não direcionado. Casos de borda típicos incluem o grafo sem arestas ($m = 0$), a cadeia $1 \to 2 \to \cdots \to n$, múltiplas fontes ou sumidouros e a inserção de uma única aresta que fecha um ciclo, invalidando a ordenação topológica.

O Código 15 apresenta a implementação do método `dag` da classe `Graph`, que gera um DAG com n vértices e exatamente m arestas. Sorteia-se uma permutação dos rótulos $1, \ldots, n$ (ordem topológica) e inserem-se m arestas distintas que respeitam essa ordem: se u precede v na permutação, a aresta é $u \to v$. O número máximo de arestas é $\frac{n(n-1)}{2}$. A estratégia segue a geração de grafos direcionados acíclicos da biblioteca jngen (SMIRNOV,).

Código 15 – Implementação do método `dag` da classe `Graph`.

```
336 /**
337  * DAG com n vertices e exatamente m arestas (estilo jngen).
338  * Permutação define ordem topológica; só aceita arestas que a respeitam.
339  * @param n número de vértices (>= 1)
340  * @param m número de arestas (0 <= m <= n*(n-1)/2)
341  */
342 inline Graph Graph::dag(int n, int m) {
343     if (n < 1)
344         quitf(_fail, "Graph::dag requires n >= 1, got %d", n);
345     if (m < 0)
```

```
346         quitf(_fail, "Graph::dag requires m >= 0, got %d", m);
347
348     long long maxEdges = 1LL * n * (n - 1) / 2;
349     if (m > maxEdges)
350         quitf(_fail, "Graph::dag: too many edges (%d > %lld)", m, maxEdges);
351
352     Graph g(n, true);
353     std::vector<int> order = rnd.perm(n, 1);
354     std::set<std::pair<int, int>> edges;
355
356     while ((int)edges.size() < m) {
357         int i = rnd.next(0, n - 2);
358         int j = rnd.next(i + 1, n - 1);
359         edges.emplace(order[i], order[j]);
360     }
361
362     for (const auto& [u, v] : edges)
363         g.addEdge(u, v);
364
365     return g;
366 }
```

Fonte: o Autor.

5 Análise de Casos de Borda em Problemas de Grafos

Problemas de grafos são recorrentes em competições de programação e exigem que o competidor domine algoritmos clássicos e também antecipe casos de borda capazes de provocar erros lógicos, estouro de tempo limite ou respostas incorretas. Em contextos de limite apertado de tempo e memória, detalhes como várias componentes conexas, vértices isolados, caminhos longos, estruturas em estrela, ciclos e grafos com diferentes densidades tornam-se decisivos para a correção da implementação.

Este capítulo discute problemas de grafos, identifica os tipos de grafos envolvidos e analisa os principais casos de borda que devem ser considerados na implementação da solução. Para cada problema, apresenta-se ainda um exemplo de geração de instância com o método correspondente da classe `Graph`. Os casos de borda levantados em cada seção são sintetizados em um quadro, com o comportamento esperado da solução. A tabela de resultados que encerra a seção instancia esses casos: cada linha associa a um item do quadro (ou a um exemplo oficial do enunciado, quando indicado) uma entrada concreta e a saída obtida por execução local da implementação apresentada.

5.1 Problema 1 - Bipartição por 2-Coloração

Enunciado resumido

Fonte. "*Building Teams*" (CSES Problem Set, task 1668).[1] Dado um grafo não direcionado simples $G = (V, E)$, que modela relações de amizade em uma turma, particione V em duas equipes de modo que nenhuma aresta conecte vértices da mesma equipe.

Entrada. A primeira linha contém dois inteiros n e m ($1 \le n \le 10^5$, $1 \le m \le 2 \cdot 10^5$), o número de alunos e de amizades. Cada uma das m linhas seguintes contém dois inteiros a e b ($1 \le a, b \le n$, $a \ne b$), indicando que os alunos a e b são amigos.

Saída. Se existir uma partição válida em duas equipes, imprima n inteiros separados por espaço, em que o i-ésimo é 1 ou 2 e indica a equipe do aluno i; caso contrário, imprima `IMPOSSIBLE`.

[1] Disponível em: <https://cses.fi/problemset/task/1668/>. Acesso em: 6 out. 2025.

Solução

Os alunos são modelados como vértices e as amizades como arestas simples. Diz-se 2-coloração uma função $c : V \to \{1, 2\}$ tal que, para toda aresta $\{u, v\} \in E$, vale $c(u) \neq c(v)$; isto é, vértices adjacentes recebem rótulos distintos. O problema equivale a verificar se G é bipartido (Definição 11) e, em caso afirmativo, produzir uma 2-coloração.

Uma abordagem possível percorre cada componente do grafo por meio de uma busca, seja em largura, seja em profundidade, atribuindo a cor 1 a um vértice inicial e alternando entre as cores 1 e 2 para seus vizinhos. Se, em alguma aresta $\{u, v\}$, $c(u) = c(v)$, então não há solução. A correção baseia-se no fato de que um grafo é bipartido se, e somente se, não possui ciclo de comprimento ímpar (LAAKSONEN, 2018). A complexidade assintótica de tempo de execução e a complexidade de memória são $O(n + m)$ quando se representa o grafo por meio de uma lista de adjacências.

Implementação

O Código 16 implementa a abordagem descrita utilizando uma DFS, reiniciando a busca em cada vértice ainda não colorido, a fim de percorrer componentes distintas. A implementação utiliza listas de adjacências, rótulos $\{1, 2\}$ e detecção de conflito quando uma aresta conecta vértices com a mesma cor, atingindo complexidade assintótica de tempo de execução $O(n + m)$.

Código 16 – Bipartição por 2-Coloração (DFS), baseado no CSES Building Teams

```cpp
#include <bits/stdc++.h>

using namespace std;

void solve() {
    int n, m;
    cin >> n >> m;
    vector<vector<int>> adj(n);

    while (m--) {
        int a, b;
        cin >> a >> b;
        a--, b--;
        adj[a].push_back(b);
        adj[b].push_back(a);
    }

    vector<int> teams(n);
    bool impossible = false;
    auto dfs = [&](auto&& self, int u, int color) -> void {
        if (teams[u]) {
            impossible |= (color != teams[u]);
            return;
        }
```

```
25
26          teams[u] = color;
27          for (auto v : adj[u])
28              self(self, v, 3 - color);
29      };
30
31      for (int i = 0; i < n; i++)
32          if (teams[i] == 0)
33              dfs(dfs, i, 1);
34
35      if (impossible)
36          cout << "IMPOSSIBLE";
37      else
38          for (auto a : teams)
39              cout << a << " ";
40      cout << '\n';
41  }
42
43  int main() {
44      ios_base::sync_with_stdio(0);
45      cin.tie(0);
46      solve();
47      return 0;
48  }
```

Fonte: o Autor.

Casos de borda e testes

A seguir, o Quadro 6 resume os principais casos de borda considerados para o Problema 1 e o comportamento esperado para cada cenário. A organização em casos permite verificar, de forma objetiva, se a implementação cobre situações mínimas, como grafos sem arestas, ou grafos notáveis, como estrelas e ciclos.

Observa-se, em especial, que a presença de qualquer componente com ciclo ímpar torna a instância globalmente inviável, exigindo a saída `IMPOSSIBLE`, mesmo quando outras componentes são válidas. Em contrapartida, grafos do tipo completo bipartido $K_{p,q}$ permanecem válidos quando os dois conjuntos recebem rótulos distintos.

Exemplo de geração

O caso de borda do grafo completo bipartido $K_{p,q}$ no Quadro 6 pode ser produzido de forma sistemática com o método `completeBipartite` da classe `Graph`. O Código 17 registra o programa como gerador via Testlib, lê os tamanhos das partições n_1 e n_2 da linha de comando e imprime a instância no formato do enunciado (n e m na primeira

Quadro 6: Casos de borda e comportamento esperado no Problema 1

Caso	**Comportamento esperado**
Grafo sem arestas (isolados)	Qualquer 2-coloração (ex.: todos 1)
Grafo desconexo	Colorir componente a componente
Estrela (um centro, várias folhas)	Válido (propagação alternada)
Ciclo par (ex.: 4-ciclo)	Válido (bipartido)
Ciclo ímpar (ex.: triângulo)	`IMPOSSIBLE`
Componente ímpar entre componentes válidas	`IMPOSSIBLE` (global)
Completo bipartido $K_{p,q}$	Válido (cores distintas para os dois conjuntos)

Fonte: o Autor.

linha, seguidos das arestas). Por exemplo, os parâmetros 3 e 3 geram um $K_{3,3}$.

Código 17 – Exemplo de gerador para o caso completo bipartido do Problema 1, com o método `completeBipartite` da classe `Graph`.

```cpp
#include "testlib.h"
#include "graphlib.h"

int main(int argc, char* argv[]) {
    registerGen(argc, argv, 1);

    int n1 = opt<int>(1);
    int n2 = opt<int>(2);

    Graph g = Graph::completeBipartite(n1, n2);
    g.printEdges();

    return 0;
}
```

Fonte: o Autor.

Resultados

A Tabela 2 instancia os casos de borda do Quadro 6: cada linha traz a entrada correspondente e a saída esperada. Os exemplos cobrem tanto cenários com solução, em que existe 2-coloração válida, quanto cenários inviáveis, nos quais a resposta correta é `IMPOSSIBLE`.

As instâncias ilustram o comportamento esperado em grafos com várias componentes conexas e em grafos bipartidos de diferentes estruturas, como estrelas e $K_{3,3}$. Além disso, evidenciam que a detecção de ciclos ímpares, mesmo quando ocorre em apenas uma componente, é suficiente para invalidar toda a instância.

Tabela 2 – Resultados - Problema 1 (bipartição por 2-coloração)

Caso	**Entrada**	**Saída**
Grafo sem arestas (isolados)	`3 0`	`1 1 1`
Grafo desconexo	`6 2` 1 2 4 5	`1 2 1 1 2 1`
Estrela (um centro, várias folhas)	`5 4` 1 2 1 3 1 4 1 5	`1 2 2 2 2`
Ciclo par (ex.: 4-ciclo)	`4 4` 1 2 2 3 3 4 4 1	`1 2 1 2`
Ciclo ímpar (ex.: triângulo)	`3 3` 1 2 2 3 1 3	`IMPOSSIBLE`
Componente ímpar entre componentes válidas	`6 4` 1 2 2 3 3 1 4 5	`IMPOSSIBLE`

Tabela 2 – Resultados - Problema 1 (bipartição por 2-coloração) (continuação)

Caso	**Entrada**	**Saída**
Completo bipartido $K_{3,3}$	`6 9` 1 4 1 5 1 6 2 4 2 5 2 6 3 4 3 5 3 6	`1 1 1 2 2 2`

Fonte: o Autor.

5.2 Problema 2 - Ordenação topológica (*Course Schedule*)

Enunciado resumido

Fonte.

"*Course Schedule*" (CSES Problem Set, task 1679).[2] Dado um grafo direcionado $G = (V, E)$ com n vértices (cursos) e m arestas, em que cada aresta da forma (a, b) indica que o curso a deve ser concluído antes de b, determine uma ordem linear dos cursos que respeite todas as restrições.

Entrada. A primeira linha contém dois inteiros n e m ($1 \leq n \leq 10^5$, $0 \leq m \leq 10^5$), o número de cursos e de pré-requisitos. Cada uma das m linhas seguintes contém dois inteiros a e b ($1 \leq a, b \leq n$), indicando que o curso a deve ser concluído antes do curso b.

Saída. Se existir uma ordenação válida, imprima n inteiros separados por espaço com os cursos em ordem topológica (rótulos de 1 a n); caso contrário, imprima `IMPOSSIBLE`.

Solução

Os cursos são modelados como vértices e os pré-requisitos como arestas direcionadas $a \to b$. Uma ordenação válida é uma ordenação topológica dos vértices: para toda aresta (u, v), o vértice u aparece antes de v na sequência. Tal ordenação existe se, e somente se, o

[2] Disponível em: <https://cses.fi/problemset/task/1679/>. Acesso em: 12 mai. 2026.

grafo é acíclico (DAG); a presença de um ciclo direcionado torna o problema impossível (LAAKSONEN, 2018).

Uma abordagem clássica é o algoritmo de Kahn: mantém-se o grau de entrada de cada vértice, inicia-se uma fila com todos os vértices de grau de entrada zero e, repetidamente, remove-se um vértice da fila, anexa-se o vértice removido à ordenação e decrementa-se o grau de entrada de seus sucessores; quando um sucessor atinge grau zero, ele entra na fila. Se, ao final, a ordenação contiver menos do que n vértices, o grafo possui um ciclo direcionado e a resposta é `IMPOSSIBLE`. A complexidade assintótica de tempo de execução e a complexidade de memória são $O(n + m)$ com listas de adjacências.

Implementação

O Código 18 implementa o algoritmo de Kahn com fila `queue`: após ler N e M, constrói-se o grafo `G` e o vetor `in` de graus de entrada (convertendo os rótulos para base 0 internamente); enfileiram-se os vértices com grau zero e, a cada remoção, o vértice é acrescentado a `topSort` e o grau de entrada de cada sucessor é decrementado. Se `topSort` contiver menos do que N vértices, imprime-se `IMPOSSIBLE`; caso contrário, imprime-se a ordenação obtida.

Código 18 – Ordenação topológica (Kahn), baseado no CSES “Course Schedule”

```cpp
#include <bits/stdc++.h>

using namespace std;

const int MAXN = 100001;
vector<int> G[MAXN];
int in[MAXN];

void solve() {
    int N, M;
    cin >> N >> M;

    for (int i = 0; i < M; i++) {
        int a, b;
        cin >> a >> b;
        a--, b--;
        G[a].push_back(b);
        in[b]++;
    }

    queue<int> q;
    for (int i = 0; i < N; i++) {
        if (in[i] == 0)
            q.push(i);
    }

    vector<int> topSort;
    while (!q.empty()) {
```

```
        auto u = q.front();
        q.pop();
        topSort.push_back(u);
        for (auto v : G[u]) {
            if (--in[v] == 0) {
                q.push(v);
            }
        }
    }

    if ((int)size(topSort) < N) {
        cout << "IMPOSSIBLE\n";
    } else {
        for (auto a : topSort) {
            cout << a + 1 << " ";
        }
        cout << "\n";
    }
}

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);
    solve();
    return 0;
}
```

Fonte: o Autor.

Casos de borda e testes

A seguir, o Quadro 7 resume casos de borda relevantes para o Problema 2: grafos sem arestas, componentes desconexas, cadeias e ciclos direcionados. Em particular, qualquer ciclo direcionado impede uma ordenação topológica, mesmo que o restante do grafo seja acíclico.

Quadro 7: Casos de borda e comportamento esperado no Problema 2

Caso	**Comportamento esperado**
Um único curso ($n = 1$, $m = 0$)	Ordem trivial `1`
Sem pré-requisitos ($m = 0$, $n > 1$)	Qualquer permutação de $1..n$
Grafo desconexo (várias cadeias)	Ordenação válida respeitando cada cadeia
Cadeia ($1 \to 2 \to \cdots \to n$)	Única ordem válida `1 2 ...n`
Ciclo direcionado	`IMPOSSIBLE`
Vários vértices com grau de entrada zero	Várias ordenações válidas

Fonte: o Autor.

Exemplo de geração

Instâncias acíclicas do Problema 2, nas quais existe ordenação topológica, podem ser produzidas com o método `dag` da classe `Graph`. O Código 19 registra o programa como gerador via Testlib, lê o número de vértices n e de arestas m da linha de comando e imprime a entrada no formato do enunciado (n e m na primeira linha, seguidos das arestas). Por exemplo, os parâmetros 4 e 3 geram um DAG com quatro cursos e três pré-requisitos.

Código 19 – Exemplo de gerador para instâncias acíclicas do Problema 2, com o método `dag` da classe `Graph`.

```cpp
#include "testlib.h"
#include "graphlib.h"

int main(int argc, char* argv[]) {
    registerGen(argc, argv, 1);

    int n = opt<int>(1);
    int m = opt<int>(2);

    Graph g = Graph::dag(n, m);
    g.printEdges();

    return 0;
}
```

Fonte: o Autor.

Resultados

A Tabela 3 instancia os casos de borda do Quadro 7, com a entrada correspondente e uma saída válida (quando existir). Para instâncias com várias ordenações corretas, qualquer permutação topológica é aceita pelo juiz.

Tabela 3 – Resultados - Problema 2 (*Course Schedule*)

Caso	**Entrada**	**Saída**
Um curso	`1 0`	`1`
Sem arestas ($n = 3$)	`3 0`	`1 2 3`

Exemplo do enunciado	`5 3`
	`1 2`
	`3 1`
	`4 5`
		`3 4 1 5 2`

Tabela 3 – Resultados - Problema 2 (*Course Schedule*) (continuação)

Caso	**Entrada**	**Saída**
Cadeia	`4 3` 1 2 2 3 3 4	`1 2 3 4`
Dois componentes acíclicos	`4 2` 1 2 3 4	`1 2 3 4`
Ciclo direcionado	`3 3` 1 2 2 3 3 1	`IMPOSSIBLE`

Fonte: o Autor.

5.3 Problema 3 - Subsequência crescente em caminhos de árvore (*Caminhada na Montanha*)

Enunciado resumido

Fonte. "Caminhada na Montanha" (Problema C da Maratona de Programação da SBC 2023, Sub-Regional Brasil do ICPC).[3] Uma montanha possui N marcos numerados de 1 a N e $N-1$ trilhas que os conectam, formando uma árvore (Definição 12) enraizada no marco 1. Cada marco i tem um valor v_i, correspondente ao número de curtidas que uma foto tirada nesse ponto terá na rede social. Partindo do marco 1 e seguindo sempre adiante pelas trilhas, sem retroceder, deseja-se postar fotos de modo que cada foto receba mais curtidas que a anterior. Para cada marco i, determine o número máximo de fotos que podem ser postadas ao iniciar a caminhada no marco 1 e terminar no marco i (não necessariamente fotografando o marco i).

Entrada. A primeira linha contém um inteiro N ($1 \leq N \leq 10^5$), o número de marcos. A segunda linha traz $N-1$ inteiros: o i-ésimo é o marco pai de $i+1$ na árvore enraizada em 1 ($1 \leq p_i \leq N$). A terceira linha contém N inteiros $v_1, \ldots, v_N$ ($1 \leq v_i \leq 10^9$), as curtidas de cada marco.

[3] Disponível em: <https://codeforces.com/gym/104555/attachments/download/21749/maratona.pdf>. Acesso em: 16 jun. 2026.

Saída. Imprima $N-1$ inteiros separados por espaço; o j-ésimo é a resposta para o marco $j+1$.

Solução

Os marcos são modelados como vértices e as trilhas como arestas de uma árvore enraizada em 1. Como existe exatamente um caminho simples entre quaisquer dois vértices, o trajeto do marco 1 até um marco i é único e corresponde ao caminho da raiz até i. A exigência de que cada foto receba mais curtidas que a anterior equivale a escolher uma subsequência estritamente crescente dos valores v ao longo desse caminho. O número máximo de fotos é, portanto, o comprimento da maior subsequência crescente (*Longest Increasing Subsequence*, LIS) da sequência de valores da raiz até cada vértice (LAAKSONEN, 2018).

A LIS de uma sequência pode ser obtida em $O(k \log k)$ por meio da técnica de *patience sorting*: mantém-se um vetor `res` no qual `res`$[j]$ guarda o menor valor possível para o final de uma subsequência crescente de comprimento $j+1$. Para cada novo valor, localiza-se por busca binária a primeira posição cujo valor é maior ou igual a ele (`lower_bound`): se o valor já ocorre em `res`, a posição é substituída e o comprimento não aumenta, o que garante o crescimento estrito; se nenhuma posição satisfaz, o valor estende o vetor. O comprimento de `res` é o tamanho da LIS corrente. A variante com `upper_bound` (primeiro elemento estritamente maior) corresponde à subsequência não decrescente e não se aplica a este problema.

Para responder a todos os marcos simultaneamente, executa-se uma DFS a partir da raiz, mantendo o vetor `res` como estado compartilhado do caminho atual. Ao entrar em um vértice u, aplica-se a operação de *patience sorting* com v_u, guarda-se o valor sobrescrito e registra-se a resposta de u como o tamanho atual de `res`. Após visitar toda a subárvore de u, desfaz-se a alteração, restaurando `res` ao estado anterior (e removendo o último elemento, caso tenha sido inserido nesta chamada). Esse retrocesso garante que cada vértice enxergue apenas os valores do seu próprio caminho até a raiz. A complexidade total é $O(N \log N)$ em tempo e $O(N)$ em memória.

Implementação

O Código 20 implementa a estratégia descrita. A entrada fornece, para cada marco $i+1$ com $i = 1, \ldots, N-1$, o pai na árvore; com isso, monta-se a lista de filhos `G` (vértices indexados em $0..N-1$, raiz no marco 1). Em seguida, leem-se as curtidas no vetor `likes`. A função `dfs` aplica `lower_bound` sobre `res` (vetor de pares *curtida, vértice* da técnica de *patience sorting*), grava `ans[u]` com o tamanho atual de `res` e restaura o estado com

`backup` ao retornar. Por fim, imprimem-se `ans[1]`, . . . , `ans[N-1]` (marcos $2..N$).

Código 20 – Subsequência crescente em caminhos de árvore, baseado na Maratona SBC 2023 “Caminhada na Montanha” (Problema C)

```cpp
#include <bits/stdc++.h>

using namespace std;

const int MAXN = 200001;

vector<int> G[MAXN];
int likes[MAXN], ans[MAXN];

vector<pair<int, int>> res;
void dfs(int u) {
    int pos = lower_bound(res.begin(), res.end(), pair(likes[u], 0)) - res.begin();
    if (pos == (int)size(res))
        res.emplace_back();
    pair<int, int> backup = res[pos];
    res[pos] = {likes[u], u};
    ans[u] = size(res);
    for (auto v : G[u]) {
        dfs(v);
    }

    res[pos] = backup;
    if (backup == pair(0, 0)) {
        res.pop_back();
    }
}

void solve() {
    int N;
    cin >> N;
    for (int i = 1; i < N; i++) {
        int p;
        cin >> p;
        p--;
        G[p].push_back(i);
    }
    for (int i = 0; i < N; i++)
        cin >> likes[i];
    dfs(0);
    for (int i = 1; i < N; i++) {
        cout << ans[i] << " ";
    }
    cout << "\n";
}

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);
    solve();
    return 0;
}
```

Fonte: o Autor.

Casos de borda e testes

A seguir, o Quadro 8 resume os principais casos de borda do Problema 3. Como a estrutura é sempre uma árvore enraizada, os casos exploram tanto formatos extremos de árvores (um único vértice, cadeias e estrelas) quanto configurações dos valores v que estressam o cálculo da LIS, como sequências constantes, estritamente crescentes e estritamente decrescentes.

Destaca-se que a exigência de crescimento estrito faz com que valores repetidos ao longo de um caminho não contribuam para o aumento da contagem de fotos: em um caminho com todos os valores iguais, a resposta é sempre 1. Já o caso $N = 1$ não possui marcos além da raiz, de modo que a saída é vazia.

Quadro 8: Casos de borda e comportamento esperado no Problema 3

Caso	**Comportamento esperado**
Único marco ($N = 1$)	Saída vazia (nenhum marco além da raiz)
Cadeia crescente	Resposta cresce a cada nível $(2, 3, \ldots, N)$
Cadeia decrescente	Resposta 1 para todos (LIS de comprimento 1)
Valores todos iguais	Resposta 1 (crescimento estrito)
Estrela (raiz com várias folhas)	Cada folha avaliada de forma independente
Árvore com ramos distintos	Cada caminho da raiz tem sua própria LIS

Fonte: o Autor.

Exemplo de geração

Os casos de borda em cadeia do Quadro 8 correspondem ao grafo caminho P_n (*bambu*) e podem ser produzidos com o método `bamboo` da classe `Graph`. O Código 21 registra o programa como gerador via Testlib, lê o número de marcos n da linha de comando, gera um P_n e imprime a instância no formato do enunciado: n, a linha de pais $p_2, \ldots, p_n$ via `printParents` (árvore enraizada no marco 1) e os valores $v_1, \ldots, v_n$ sorteados uniformemente. A permutação de rótulos de `bamboo` pode deixar o vértice 1 em uma extremidade (cadeia a partir da raiz) ou no interior do caminho (dois ramos a partir da raiz); em ambos os casos a estrutura permanece um P_n válido, mas apenas a primeira disposição realiza literalmente os casos "cadeia crescente/decrescente" do quadro. Para forçar a cadeia enraizada em uma extremidade, bastaria fixar a permutação sem

embaralhar o rótulo da raiz. Por exemplo, o parâmetro 4 gera um P_4.

Código 21 – Exemplo de gerador para o caso em cadeia do Problema 3, com o método `bamboo` da classe `Graph`.

```cpp
#include "testlib.h"
#include "graphlib.h"

#include <vector>

int main(int argc, char* argv[]) {
    registerGen(argc, argv, 1);

    int n = opt<int>(1);

    Graph g = Graph::bamboo(n);
    std::cout << n << '\n';
    g.printParents();

    for (int i = 1; i <= n; ++i) {
        if (i > 1)
            std::cout << ' ';
        std::cout << rnd.next(1, 1000000000);
    }
    std::cout << '\n';

    return 0;
}
```

Fonte: o Autor.

Resultados

A Tabela 4 instancia os casos de borda do Quadro 8 e inclui os dois exemplos do enunciado. Cada entrada lista, em sequência, o número de marcos N, os pais $p_2 \dots p_N$ e os valores $v_1 \dots v_N$; a saída traz os $N-1$ inteiros correspondentes aos marcos $2..N$.

Tabela 4 – Resultados - Problema 3 (*Caminhada na Montanha*)

Caso	**Entrada**	**Saída**
Exemplo 1 do enunciado	5 1 1 3 3 5 7 7 6 8	2 2 2 3
Exemplo 2 do enunciado	5 3 1 3 1 5 4 7 6 5	2 2 2 1

Tabela 4 – Resultados - Problema 3 (*Caminhada na Montanha*) (continuação)

Caso	**Entrada**	**Saída**
Cadeia crescente	4 1 2 3 1 2 3 4	2 3 4
Cadeia decrescente	4 1 2 3 4 3 2 1	1 1 1
Valores todos iguais	4 1 2 3 5 5 5 5	1 1 1
Estrela	5 1 1 1 1 1 5 4 3 2	2 2 2 2

Fonte: o Autor.

5.4 Problema 4 - Componente conexa restrita a intervalo em árvore (*Empresa de Festas*)

Enunciado resumido

Fonte. "Empresa de Festas" (Problema E da Maratona de Programação da SBC 2020, Sub-Regional Brasil do ICPC).[4] Uma empresa possui N empregados organizados de forma hierárquica: cada empregado, exceto o dono (empregado 1), tem um gerente direto, sem relações cíclicas de gerência. Cada empregado i tem uma idade A_i, e é garantido que a idade de um empregado nunca é maior que a de seu gerente direto ($A_i \leq A_{B_i}$). São organizadas M festas; a j-ésima tem um anfitrião O_j e um intervalo de idades $[L_j, R_j]$. Cada festa convida o maior conjunto de pessoas tal que: o anfitrião participa; todo convidado tem idade em $[L_j, R_j]$; e qualquer um, exceto o anfitrião, trabalha diretamente com (é gerente ou subordinado de) algum outro participante. Deve-se determinar de quantas festas cada empregado participou.

[4] Disponível em: <https://codeforces.com/gym/102861/attachments/download/12360/statements-pt.pdf>. Acesso em: 21 jun. 2026.

Entrada. A primeira linha contém dois inteiros N e M ($1 \leq N, M \leq 10^5$), o número de empregados e de festas. Cada uma das N linhas seguintes contém A_i e B_i ($1 \leq A_i \leq 10^5$, $1 \leq B_i \leq N$), a idade e o gerente direto do empregado i; os empregados são numerados de 1 a N, sendo 1 o dono (único com $B_i = i$), e vale $A_i \leq A_{B_i}$. Cada uma das M linhas finais contém O_j, L_j e R_j ($1 \leq L_j \leq A_{O_j} \leq R_j \leq 10^5$), o anfitrião e o intervalo de idades da j-ésima festa.

Saída. A saída consiste em uma única linha com N inteiros separados por espaço, em que o i-ésimo inteiro representa o número de festas das quais o empregado i participou.

Solução

A hierarquia é modelada como uma árvore (Definição 12) enraizada no dono. A exigência de que todo convidado trabalhe diretamente com outro participante significa que o conjunto de participantes deve ser conexo na árvore; somada à restrição de idade, isso equivale a tomar a componente conexa (Definição 8) que contém o anfitrião no subgrafo induzido pelos empregados com idade em $[L, R]$.

A propriedade $A_i \leq A_{B_i}$ torna as idades não decrescentes em direção à raiz; logo, ao longo de qualquer caminho da raiz até um vértice, elas são não crescentes de cima para baixo. Disso decorre a estrutura da componente: subindo a partir do anfitrião O, as idades só aumentam, de modo que se podem incluir ancestrais enquanto a idade permanecer $\leq R$. Seja a^* o ancestral mais alto de O com idade $\leq R$; ele é o topo da componente. Descendo a partir de a^*, as idades só diminuem, então todo descendente já satisfaz $A \leq A_{a^*} \leq R$, bastando que $A \geq L$. Como o menor valor ao longo do caminho de a^* a x é justamente A_x, um empregado x participa se, e somente se, está na subárvore de a^* e $A_x \geq L$.

Cada festa, portanto, é ancorada em a^*, e um empregado u participa dela quando a^* é ancestral de u e $L \leq A_u$. Para localizar a^*, nota-se que as idades ao longo do caminho da raiz até O já estão ordenadas (não crescentes), o que permite encontrá-lo por busca binária. Para a contagem, percorre-se a árvore com uma busca em profundidade (Definição 7), mantendo uma árvore de Fenwick (*Binary Indexed Tree*, BIT) indexada por idade: ao entrar em um vértice u, inserem-se as festas ancoradas em u, indexadas pela extremidade L; a resposta de u é o número de festas ativas com $L \leq A_u$, obtido por uma consulta de prefixo; ao retornar, desfazem-se as inserções, de modo que permaneçam ativas apenas as festas ancoradas no caminho atual da raiz até u. A complexidade é $O((N + M) \log N)$ em tempo.

Implementação

O Código 22 implementa a estratégia. A entrada fornece, para cada empregado i, a idade A_i e o gerente B_i, a partir dos quais se constrói a árvore (vértices indexados

em 0 internamente). A função `dfs1` mantém o caminho da raiz até o vértice atual e, para cada festa cujo anfitrião é u, aplica a busca binária descrita para encontrar o ancestral âncora a^*, registrando a festa nele. A função `dfs2` percorre a árvore com a BIT indexada por idade: insere as festas ancoradas no vértice (pela extremidade L), calcula `ans`$[u]$ como o número de festas ativas com $L \leq A_u$ e desfaz as inserções ao retornar. Por fim, imprime a contagem de todos os empregados.

Código 22 – Componente conexa restrita a intervalo em árvore, baseado na Maratona SBC 2020 “Empresa de Festas” (Problema E)

```cpp
#include <bits/stdc++.h>

using namespace std;

const char nl = '\n';
const int MAXN = 200001;

vector<int> G[MAXN];
vector<pair<int, int>> parties[MAXN], p2[MAXN];
int A[MAXN];

vector<int> path;
void dfs1(int u) {
    path.push_back(u);
    for (auto p : parties[u]) {
        auto& [l, r] = p;
        int lo = 0, hi = size(path);
        while (lo < hi) {
            int mid = (lo + hi) / 2;
            if (A[path[mid]] <= r) {
                hi = mid;
            } else {
                lo = mid + 1;
            }
        }
        p2[path[hi]].push_back(p);
    }
    for (auto v : G[u]) {
        dfs1(v);
    }
    path.pop_back();
}

int BIT[MAXN];
void update(int i, int v) {
    for (i++; i < MAXN; i += (i & -i)) {
        BIT[i] += v;
    }
}
int query(int i) {
    int res = 0;
    for (i++; i > 0; i -= (i & -i)) {
        res += BIT[i];
    }
```

```
45      return res;
46  }
47
48  int ans[MAXN];
49  void dfs2(int u) {
50      for (auto p : p2[u]) {
51          update(p.first, 1);
52      }
53      ans[u] = query(A[u]);
54      for (auto v : G[u]) {
55          dfs2(v);
56      }
57
58      for (auto p : p2[u]) {
59          update(p.first, -1);
60      }
61  }
62
63  void solve() {
64      int N, M;
65      cin >> N >> M;
66
67      for (int i = 0; i < N; i++) {
68          int b;
69          cin >> A[i] >> b;
70          b--;
71          if (b != i)
72              G[b].push_back(i);
73      }
74
75      for (int i = 0; i < M; i++) {
76          int o, l, r;
77          cin >> o >> l >> r;
78          o--;
79          parties[o].emplace_back(l, r);
80      }
81      dfs1(0);
82      dfs2(0);
83
84      for (int i = 0; i < N; i++) {
85          cout << ans[i] << " ";
86      }
87      cout << nl;
88  }
89
90  int main() {
91      ios_base::sync_with_stdio(0);
92      cin.tie(0);
93      solve();
94      return 0;
95  }
```

Fonte: o Autor.

Casos de borda e testes

A seguir, o Quadro 9 resume os principais casos de borda do Problema 4. Como a estrutura é uma árvore enraizada com idades não decrescentes rumo à raiz, os casos exploram formatos extremos da hierarquia (um único empregado, cadeias e estrelas) e, sobretudo, a interação entre o intervalo de idades e a exigência de conectividade.

Destaca-se que um empregado com idade dentro de $[L, R]$ pode, ainda assim, não ser convidado, caso esteja separado do anfitrião por um gerente cuja idade esteja fora do intervalo: a componente conexa é interrompida nesse ponto. No extremo oposto, quando todas as idades são iguais e pertencem ao intervalo, toda a subárvore alcançável é convidada.

Quadro 9: Casos de borda e comportamento esperado no Problema 4

Caso	**Comportamento esperado**
Empresa com um único empregado ($N = 1$)	Apenas o dono; cada festa conta 1 para ele
Cadeia (cada um gerencia o próximo)	Festa ampla cobre toda a cadeia; festa estreita no topo cobre só o dono
Estrela (dono com vários subordinados)	Festa de um filho sobe ao dono e desce aos irmãos no intervalo
Idades todas iguais	Toda a subárvore alcançável é convidada
Vértice no intervalo, mas desconexo do anfitrião	Não é convidado (componente conexa interrompida)
Intervalo mínimo ($L = R = A_O$)	Só o anfitrião e vizinhos de mesma idade

Fonte: o Autor.

Exemplo de geração

O caso de borda em estrela do Quadro 9 pode ser produzido com o método `star` da classe `Graph`. O Código 23 registra o programa como gerador via Testlib, lê o número de empregados n e de festas m da linha de comando, gera uma estrela enraizada no dono (vértice 1) e imprime a instância no formato do enunciado: para cada empregado, a idade A_i e o gerente B_i (obtido de `parent`), com $A_i \leq A_{B_i}$, seguidos das festas com intervalos compatíveis com a idade do anfitrião. Por exemplo, os parâmetros 4 e 2 geram uma estrela com quatro empregados e duas festas.

Código 23 – Exemplo de gerador para o caso em estrela do Problema 4, com o método `star` da classe `Graph`.

```
1 #include "testlib.h"
2 #include "graphlib.h"
3
4 #include <vector>
```

```
5
6  int main(int argc, char* argv[]) {
7      registerGen(argc, argv, 1);
8
9      int n = opt<int>(1);
10     int m = opt<int>(2);
11
12     Graph g = Graph::star(n);
13
14     std::vector<int> age(n + 1);
15     age[1] = rnd.next(1, 100000);
16     for (int i = 2; i <= n; ++i)
17         age[i] = rnd.next(1, age[1]);
18
19     std::cout << n << ' ' << m << '\n';
20     for (int i = 1; i <= n; ++i) {
21         int boss = (i == 1) ? 1 : g.parent[i];
22         std::cout << age[i] << ' ' << boss << '\n';
23     }
24
25     for (int j = 0; j < m; ++j) {
26         int o = rnd.next(1, n);
27         int l = rnd.next(1, age[o]);
28         int r = rnd.next(age[o], 100000);
29         std::cout << o << ' ' << l << ' ' << r << '\n';
30     }
31
32     return 0;
33 }
```

Fonte: o Autor.

Resultados

A Tabela 5 instancia os casos de borda do Quadro 9 e inclui o exemplo do enunciado, no formato de entrada e saída descrito anteriormente.

Tabela 5 – Resultados - Problema 4 (*Empresa de Festas*)

Caso	**Entrada**	**Saída**
Exemplo do enunciado	`10 3` 8 1 3 5 5 1 2 3 4 1 3 3 1 2 7 1 2 2 3 2 3 5 9 5 3 8 3 2 6	`2 1 3 1 1` 2 0 2 0 1
Cadeia	`4 2` 10 1 7 1 5 2 2 3 4 2 10 1 10 10	`2 1 1 1`
Estrela	`4 2` 10 1 5 1 3 1 8 1 2 3 10 2 5 6	`1 2 1 1`
Idades todas iguais	`3 1` 5 1 5 1 5 1 2 5 5	`1 1 1`

Tabela 5 – Resultados - Problema 4 (*Empresa de Festas*) (continuação)

Caso	**Entrada**	**Saída**
Desconexo do anfitrião	`3 1` 10 1 5 1 4 1 2 4 5	`0 1 0`

Fonte: o Autor.

5.5 Problema 5 - Menor gerente na cadeia de comando (*Boss*)

Enunciado resumido

Fonte. "Boss" (Problema C da Maratona de Programação da SBC 2013, Sub-Regional Brasil do ICPC).[5] Uma empresa possui N funcionários identificados de 1 a N, cada um com uma idade K_i. Existem M relações diretas de gerência: quando X gerencia Y, X é o superior imediato de Y. Um funcionário pode ser gerenciado direta ou indiretamente por outro. São dadas I instruções de dois tipos: `T` A B, que troca as posições de A e B na cadeia de comando (sem alterar a topologia das relações de gerência), e `P` E, que consulta a idade do gerente mais jovem de E (direto ou indireto).

Entrada. A primeira linha contém N, M e I ($1 \leq N \leq 500$, $0 \leq M \leq 6 \times 10^4$, $1 \leq I \leq 500$). A segunda linha traz as idades $K_1, \ldots, K_N$ ($1 \leq K_i \leq 100$). Cada uma das M linhas seguintes contém X e Y ($1 \leq X, Y \leq N$, $X \neq Y$), indicando que X gerencia Y diretamente. Cada uma das I linhas finais descreve uma instrução: `T` A B ($1 \leq A, B \leq N$) para troca de posições, ou `P` E ($1 \leq E \leq N$) para consulta. A última instrução é sempre do tipo `P`.

Saída. Para cada instrução `P`, imprima uma linha com a idade do gerente mais jovem do funcionário consultado; se ele não possui gerente, imprima `*`.

Solução

A cadeia de comando é modelada como um grafo direcionado cujos vértices representam posições fixas na estrutura organizacional. Para cada relação "X gerencia Y" constrói-se a aresta $Y \to X$, de modo que percorrer arestas a partir da posição de um

[5] Disponível em: <https://codeforces.com/gym/101473/attachments/download/5792/20132014-acmicpc-brazil-subregional-programming-contest-en.pdf>. Acesso em: 21 jun. 2026.

funcionário equivale a subir na hierarquia. Os funcionários ocupam essas posições de forma dinâmica: os vetores `where` e `who` mantêm a permutação entre funcionários e posições (`where`$[i]$ indica onde está o funcionário i; `who`$[p]$ indica quem ocupa a posição p). Uma operação `T` apenas troca quem ocupa duas posições, sem modificar o grafo de gerência.

Para uma consulta `P` E, o problema reduz-se a encontrar o menor valor entre as idades dos funcionários que ocupam posições alcançáveis a partir de `where`$[E]$ no grafo de gerência. Isso é feito com uma busca em largura (Definição 6) a partir de `where`$[E]$: a cada posição v visitada, atualiza-se o mínimo com $K_{\texttt{who}[v]}$. Se nenhum superior é alcançado, a resposta é `*`. Como $N \leq 500$ e $I \leq 500$, uma BFS de $O(N)$ por consulta é suficiente.

Implementação

O Código 24 lê as idades no vetor `K` e as relações de gerência, inicializando `where`$[i] = i$ e `who`$[i] = i$. Para cada instrução `T`, troca-se `who`[`where`$[A]$] com `who`[`where`$[B]$] e, em seguida, `where`$[A]$ com `where`$[B]$. Para cada instrução `P`, a função `bfs` parte de `where`$[E]$, percorre as arestas de subordinado para superior e retorna a menor idade encontrada; se nenhum superior é visitado, imprime `*`.

Código 24 – Menor gerente na cadeia de comando, baseado na Maratona SBC 2013 “Boss” (Problema C)

```cpp
#include <bits/stdc++.h>

using namespace std;

const int INF = INT_MAX; // sentinela ("infinito") para inicializar buscas de minimo
const int MAXN = 501;

int N, M, I;
vector<int> G[MAXN]; // lista de adjacencia
int K[MAXN];         // idades
int who[MAXN];       // who[p] -> quem ocupa a posicao p
int where[MAXN];     // where[i] -> onde esta o funcionario i

int bfs(int s) {
    queue<int> q;
    vector<int> used(N);
    used[s] = 1;
    q.push(s);
    int ans = INF;
    while (!q.empty()) {
        auto u = q.front();
        q.pop();
        for (auto v : G[u]) {
            if (!used[v]) {
                ans = min(ans, K[who[v]]);
                used[v] = 1;
                q.push(v);
            }
```

```
        }
    }
    return ans;
}

void solve() {
    cin >> N >> M >> I;

    for (int i = 0; i < N; i++) {
        cin >> K[i];
    }

    for (int i = 0; i < N; i++) {
        where[i] = i;
        who[i] = i;
    }

    for (int i = 0; i < M; i++) {
        int a, b;
        cin >> a >> b;
        a--, b--;
        G[b].push_back(a);
    }

    while (I--) {
        char op;
        cin >> op;
        if (op == 'T') {
            int a, b;
            cin >> a >> b;
            a--, b--;
            swap(who[where[a]], who[where[b]]);
            swap(where[a], where[b]);
        } else {
            int a;
            cin >> a;
            a--;
            int res = bfs(where[a]);
            if (res == INF) {
                cout << "*\n";
            } else {
                cout << res << "\n";
            }
        }
    }
}

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);
    solve();
    return 0;
}
```

Fonte: o Autor.

Casos de borda e testes

A seguir, o Quadro 10 resume os principais casos de borda do Problema 5. A estrutura combina um grafo direcionado de gerência com permutações dinâmicas de funcionários nas posições, de modo que os casos exploram tanto consultas sem superior quanto o efeito das trocas sobre quem é alcançado na subida pela hierarquia.

Destaca-se que um funcionário sem gerente direto ou indireto deve produzir `*`, e que uma troca `T` pode fazer com que um funcionário passe a ocupar uma posição sem superiores, mesmo tendo gerente antes da operação.

Quadro 10: Casos de borda e comportamento esperado no Problema 5

Caso	**Comportamento esperado**
Funcionário sem gerente ($N = 1$ ou topo da hierarquia)	Resposta `*`
Cadeia de gerência	Menor idade entre todos os superiores alcançáveis
Múltiplos caminhos até o mesmo funcionário	BFS considera todos os gerentes diretos e indiretos
Troca `T` entre funcionários	Idades nas posições mudam; consultas subsequentes refletem a nova ocupação
Troca leva funcionário a posição sem superiores	Consulta passa a retornar `*`
Última instrução sempre é `P`	Garante ao menos uma linha de saída por caso de teste

Fonte: o Autor.

Exemplo de geração

A hierarquia acíclica do Problema 5, incluindo o caso de cadeia do Quadro 10 como extremo alongado, pode ser produzida com o método `tree` da classe `Graph`. O enunciado admite, em geral, um digrafo de gerência com até M relações (não necessariamente uma árvore); o gerador a seguir cobre deliberadamente o subcaso em que a hierarquia é uma árvore enraizada, suficiente para exercitar cadeias e consultas sem superior. O Código 25 registra o programa como gerador via Testlib, lê o número de funcionários n e de instruções I da linha de comando, gera uma árvore enraizada no vértice 1 e imprime a instância no formato do enunciado: idades, $n - 1$ relações de gerência obtidas de `parent` (pai gerencia o filho) e I instruções cuja última é sempre uma consulta `P`. Por exemplo, os parâmetros 4 e 3 geram uma árvore com quatro funcionários, três relações e três instruções.

Código 25 – Exemplo de gerador para hierarquias em árvore do Problema 5, com o método `tree` da classe `Graph`.

```cpp
#include "testlib.h"
#include "graphlib.h"

int main(int argc, char* argv[]) {
    registerGen(argc, argv, 1);

    int n = opt<int>(1);
    int qi = opt<int>(2);

    Graph g = Graph::tree(n);
    int m = n - 1;

    std::cout << n << ' ' << m << ' ' << qi << '\n';
    for (int i = 1; i <= n; ++i) {
        if (i > 1)
            std::cout << ' ';
        std::cout << rnd.next(1, 100);
    }
    std::cout << '\n';

    for (int i = 2; i <= n; ++i)
        std::cout << g.parent[i] << ' ' << i << '\n';

    for (int j = 0; j < qi - 1; ++j) {
        if (n >= 2 && rnd.next(0, 1) == 0) {
            int a = rnd.next(1, n);
            int b = rnd.next(1, n);
            while (b == a)
                b = rnd.next(1, n);
            std::cout << "T " << a << ' ' << b << '\n';
        } else {
            std::cout << "P " << rnd.next(1, n) << '\n';
        }
    }
    std::cout << "P " << rnd.next(1, n) << '\n';

    return 0;
}
```

Fonte: o Autor.

Resultados

A Tabela 6 instancia os casos de borda do Quadro 10 e inclui os exemplos do enunciado, no formato de entrada e saída descrito anteriormente.

Tabela 6 – Resultados - Problema 5 (*Boss*)

Caso	**Entrada**	**Saída**
Exemplo 1 do enunciado	`7 8 9`	`18`
	21 33 33 18 42 22 26	21
	1 2	18
	1 3	18
	2 5	
	3 5	26
	3 6	
	4 6	
	4 7	
	6 7	
	P 7	
	T 4 2	
	P 7	
	P 5	
	T 1 4	
	P 7	
	T 4 7	
	P 2	
	P 6	
Exemplo 2 do enunciado	`6 5 6`	`*`
	10 20 30 40 50 60	10
	1 5	30
	1 4	30
	3 6	60
	2 5	
	4 5	
	P 1	
	P 5	
	P 6	
	T 1 6	
	P 1	
	P 4	
Funcionário isolado	`1 0 1`	`*`
	50	
	P 1	

Tabela 6 – Resultados - Problema 5 (*Boss*) (continuação)

Caso	**Entrada**	**Saída**
Cadeia de gerência	`3 2 3` 40 30 20 1 2 2 3 P 3 P 2 P 1	`30` 40
Efeito de troca	`2 1 3` 30 20 1 2 P 2 T 1 2 P 2	`30`

Fonte: o Autor.

5.6 Problema 6 - Capacidade máxima em caminhos de entrega (*Trucks*)

Enunciado resumido

Fonte. "Trucks" (Problema J da Maratona de Programação da SBC 2013, Sub-Regional Brasil do ICPC).[6] Nlogônia é um arquipélago de N ilhas conectadas por M pontes bidirecionais; cada ponte (A, B) suporta no máximo W toneladas. O grafo é conexo e há no máximo uma ponte entre cada par de ilhas. Para cada um dos S pares depósito/local de competição (L, H), deve-se determinar o maior peso que um caminhão pode transportar do depósito na ilha L até o local na ilha H sem exceder o limite de nenhuma ponte do trajeto.

Entrada. A primeira linha contém N, M e S ($2 \leq N \leq 2 \cdot 10^4$, $1 \leq M \leq 10^5$, $1 \leq S \leq 5 \cdot 10^4$). Cada uma das M linhas seguintes traz A, B e W ($1 \leq A, B \leq N$, $A \neq B$, $0 \leq W \leq 10^5$), descrevendo uma ponte e seu limite de peso. Cada uma das S linhas finais contém L e H ($1 \leq L, H \leq N$, $L \neq H$), indicando o depósito e o local da competição.

[6] Disponível em: <https://codeforces.com/gym/101473/attachments/download/5792/20132014-acmicpc-brazil-subregional-programming-contest-en.pdf>. Acesso em: 21 jun. 2026.

Saída. Para cada consulta, na ordem da entrada, imprima uma linha com o maior peso transportável entre L e H.

Solução

As ilhas são modeladas como vértices e as pontes como arestas ponderadas com capacidade máxima. Para um par (L, H), o peso transportável é o gargalo do melhor caminho: o máximo, entre todos os caminhos (Definição 2), do menor peso de aresta nesse caminho. Em outras palavras, busca-se o caminho que maximiza a capacidade mínima ao longo do trajeto (problema MaxiMin) (HALIM; HALIM; EFFENDY, 2020).

Esse valor coincide com o peso mínimo ao longo do único caminho entre L e H em uma árvore geradora de peso máximo (MaxST): ao processar arestas em ordem decrescente de capacidade, o algoritmo de Kruskal com união e busca (DSU, do inglês *Disjoint Set Union*) constrói tal árvore (HALIM; HALIM; EFFENDY, 2020; LAAKSONEN, 2018). A cada união de componentes por uma aresta de peso w, registra-se o filho na estrutura pai $\texttt{G}[\mathit{filho}] = (\mathit{pai}, w)$.

Para cada consulta, encontra-se um ancestral comum de L e H nessa árvore, e a resposta é o mínimo entre os pesos das arestas no caminho até esse ancestral. A implementação ilustrada percorre os ancestrais de L e, para cada um, verifica se H alcança o mesmo vértice subindo por `G`, atualizando o mínimo dos gargalos encontrados.

Implementação

O Código 26 ordena as arestas por peso decrescente e aplica Kruskal com DSU, preenchendo `G` a cada união bem-sucedida. A construção da MaxST tem complexidade assintótica de tempo de execução $O(M \log M)$. Para cada consulta (L, H), dois laços aninhados sobem pelos ponteiros pai em `G` até encontrar um vértice comum; no pior caso, essa consulta tem complexidade $O(N^2)$, de modo que o conjunto das S consultas fica em $O(S \cdot N^2)$. Sob as restrições do enunciado ($N \leq 2 \cdot 10^4$, $S \leq 5 \cdot 10^4$), uma versão de contest exigiria acelerar as consultas (por exemplo, com ancestral comum mais baixo, LCA); o código apresentado prioriza a clareza da redução à MaxST. Os índices são convertidos para base 0 internamente.

Código 26 – Capacidade máxima em caminhos de entrega, baseado na Maratona SBC 2013 “Trucks” (Problema J)

```cpp
#include <bits/stdc++.h>

using namespace std;

const int INF = INT_MAX; // sentinela ("infinito") para inicializar buscas de minimo
const char nl = '\n';
```

```
7
8  struct DSU {
9      vector<int> e;
10     DSU(int N) { e = vector<int>(N, -1); }
11     int get(int i) { return (e[i] < 0 ? i : e[i] = get(e[i])); }
12     bool unite(int i, int j) {
13         i = get(i);
14         j = get(j);
15         if (i == j)
16             return false;
17         if (e[j] < e[i])
18             swap(i, j);
19         e[i] += e[j];
20         e[j] = i;
21         return true;
22     }
23 };
24
25 void solve() {
26     int N, M, S;
27     cin >> N >> M >> S;
28
29     vector<pair<int, pair<int, int>>> edges;
30     for (int i = 0; i < M; i++) {
31         int a, b, w;
32         cin >> a >> b >> w;
33         a-, b-;
34         edges.emplace_back(w, pair(a, b));
35     }
36
37     sort(edges.begin(), edges.end());
38     reverse(edges.begin(), edges.end());
39
40     DSU dsu(N);
41
42     vector<pair<int, int>> G(N, {-1, INF});
43     for (int i = 0; i < M; i++) {
44         auto [w, p] = edges[i];
45         auto [a, b] = p;
46
47         a = dsu.get(a);
48         b = dsu.get(b);
49         if (a != b) {
50             dsu.unite(a, b);
51             if (a != dsu.get(a)) {
52                 G[a] = {b, w};
53             } else {
54                 G[b] = {a, w};
55             }
56         }
57     }
58     for (int i = 0; i < S; i++) {
59         int a, b;
60         cin >> a >> b;
61         a-, b-;
62
63         int ans = INF, ca = INF;
```

```
64          while (a != -1) {
65              int tmp = b;
66              int cb = INF;
67              while (tmp != -1) {
68                  if (a == tmp) {
69                      ans = min(ca, cb);
70                      goto done;
71                  }
72                  cb = G[tmp].second;
73                  tmp = G[tmp].first;
74              }
75              ca = G[a].second;
76              a = G[a].first;
77          }
78      done:
79          cout << ans << nl;
80      }
81  }
82
83  int main() {
84      ios_base::sync_with_stdio(0);
85      cin.tie(0);
86      solve();
87      return 0;
88  }
```

Fonte: o Autor.

Casos de borda e testes

A seguir, o Quadro 11 resume os principais casos de borda do Problema 6. Por se tratar de um grafo conexo (Definição 8) com limites de capacidade nas arestas, os casos exploram pontes com peso nulo, rotas alternativas e a escolha do caminho de maior gargalo.

Destaca-se que uma ponte com capacidade 0 não impede necessariamente o transporte, desde que exista outro caminho entre as ilhas com arestas de capacidade positiva; nesse caso, a resposta é o gargalo da melhor rota alternativa.

Exemplo de geração

O caso de borda do grafo completo no Quadro 11 pode ser produzido com o método `complete` da classe `Graph`. O Código 27 registra o programa como gerador via Testlib, lê o número de ilhas n e de consultas S da linha de comando, gera um K_n e imprime a instância no formato do enunciado: arestas com capacidades sorteadas uniformemente

Quadro 11: Casos de borda e comportamento esperado no Problema 6

Caso	**Comportamento esperado**
Duas ilhas com uma única ponte	Resposta igual à capacidade dessa ponte
Ponte com capacidade 0 e rota alternativa	Usa o caminho que evita a ponte frágil
Cadeia de pontes	Gargalo é o mínimo entre as capacidades do caminho
Múltiplas rotas possíveis	Escolhe o caminho de maior gargalo (MaxST)
Consulta entre ilhas adjacentes	Resposta é o peso da ponte direta ou de rota melhor
Grafo completo com pesos distintos	Caminho direto ou indireto de maior capacidade mínima

Fonte: o Autor.

em $\{0, \ldots, 10^5\}$ e S pares distintos de depósito e local. Por exemplo, os parâmetros 4 e 2 geram um K_4 com duas consultas.

Código 27 – Exemplo de gerador para o caso completo do Problema 6, com o método `complete` da classe `Graph`.

```cpp
#include "testlib.h"
#include "graphlib.h"

int main(int argc, char* argv[]) {
    registerGen(argc, argv, 1);

    int n = opt<int>(1);
    int s = opt<int>(2);

    Graph g = Graph::complete(n);
    int m = g.m();

    auto edges = g.edges();
    shuffle(edges.begin(), edges.end());

    std::cout << n << ' ' << m << ' ' << s << '\n';
    for (const auto& [u, v] : edges)
        std::cout << u << ' ' << v << ' ' << rnd.next(0, 100000) << '\n';

    for (int i = 0; i < s; ++i) {
        int l = rnd.next(1, n);
        int h = rnd.next(1, n);
        while (h == l)
            h = rnd.next(1, n);
        std::cout << l << ' ' << h << '\n';
    }

    return 0;
}
```

Fonte: o Autor.

Resultados

A Tabela 7 instancia os casos de borda do Quadro 11 e inclui os exemplos do enunciado, no formato de entrada e saída descrito anteriormente.

Tabela 7 – Resultados - Problema 6 (*Trucks*)

Caso	**Entrada**	**Saída**
Exemplo 1 do enunciado	`4 5 4` 1 2 9 1 3 0 2 3 8 2 4 7 3 4 4 1 4 2 1 3 1 4 3	`7` 9 8 7
Exemplo 2 do enunciado	`4 5 2` 1 2 30 2 3 20 3 4 10 4 1 40 2 4 50 1 3 1 2	`20` 40
Duas ilhas	`2 1 1` 1 2 5 1 2	`5`
Ponte com capacidade 0	`4 5 1` 1 2 9 1 3 0 2 3 8 2 4 7 3 4 4 1 3	`8`

Tabela 7 – Resultados - Problema 6 (*Trucks*) (continuação)

Caso	**Entrada**	**Saída**
Cadeia de pontes	`3 2 1` 1 2 10 2 3 5 1 3	`5`

Fonte: o Autor.

5.7 Problema 7 - Ciclo simples (*Round Trip*)

Enunciado resumido

Fonte. "*Round Trip*" (CSES Problem Set, task 1669).[7] Em Byteland há n cidades e m estradas bidirecionais. Deseja-se projetar um passeio circular que comece em uma cidade, visite duas ou mais outras cidades distintas e retorne ao início. Em outras palavras, deve-se encontrar um ciclo simples com pelo menos três vértices distintos.

Entrada. A primeira linha contém dois inteiros n e m ($1 \leq n \leq 10^5$, $1 \leq m \leq 2 \cdot 10^5$), o número de cidades e de estradas. Cada uma das m linhas seguintes contém dois inteiros a e b ($1 \leq a, b \leq n$), indicando uma estrada entre a e b. O grafo é simples: não há laços nem arestas paralelas.

Saída. Se existir um ciclo válido, imprima um inteiro k (o número de cidades na rota, incluindo a repetição da cidade inicial no fim) e, na linha seguinte, k inteiros com a sequência visitada. Caso contrário, imprima `IMPOSSIBLE`. Qualquer ciclo válido é aceito.

Solução

As cidades são modeladas como vértices e as estradas como arestas de um grafo não direcionado simples. O objetivo é decidir se o grafo contém um ciclo (Definição 2) e, em caso afirmativo, exibir uma sequência que o percorre e retorna à origem.

Uma busca em profundidade (Definição 7) com registro do pai detecta ciclos em grafos não direcionados: ao explorar a vizinhança de um vértice u, se algum vizinho v já estiver descoberto e v não for o pai de u na árvore de descoberta, então a aresta $\{u, v\}$ fecha um ciclo com o caminho de v até u. Mantém-se um vetor `path` com o caminho atual da DFS; ao detectar o ciclo, anexa-se v ao vetor e extrai-se o sufixo a partir da primeira

[7] Disponível em: <https://cses.fi/problemset/task/1669/>. Acesso em: 12 jul. 2026.

ocorrência de v, que é exatamente o ciclo a imprimir (com v repetido no fim). Se a busca esgotar todas as componentes sem encontrar tal aresta, o grafo é uma floresta e a resposta é `IMPOSSIBLE`. A complexidade assintótica de tempo de execução e a complexidade de memória são $O(n + m)$ com lista de adjacências.

Implementação

O Código 28 implementa a estratégia descrita. Após construir a lista de adjacências (vértices em base 0), a DFS recursiva marca vértices visitados, empilha o caminho corrente e, ao encontrar um vizinho já visitado distinto do pai, completa o ciclo e retorna verdadeiro. O laço externo reinicia a busca em cada componente ainda não explorada. Encontrado um ciclo, imprime-se o comprimento do sufixo e os rótulos convertidos de volta para $1..n$; caso contrário, imprime-se `IMPOSSIBLE`.

Código 28 – Detecção de ciclo simples (DFS), baseado no CSES “Round Trip”

```cpp
#include <bits/stdc++.h>

using namespace std;

void solve() {
    int n, m;
    cin >> n >> m;
    vector<vector<int>> adj(n);
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        --u;
        --v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }

    vector<bool> used(n);
    vector<int> path;
    auto dfs = [&](int pos, int parent, auto&& dfs) -> bool {
        used[pos] = true;
        path.push_back(pos);
        for (auto u : adj[pos]) {
            if (!used[u]) {
                if (dfs(u, pos, dfs))
                    return true;
            } else if (u != parent) {
                path.push_back(u);
                return true;
            }
        }
        path.pop_back();
        return false;
    };

    for (int i = 0; i < n; i++) {
```

```
        if (used[i])
            continue;
        if (dfs(i, -1, dfs)) {
            int last = path.back();
            auto it = find(path.begin(), path.end(), last);
            cout << path.end() - it << '\n';
            for (; it != path.end(); it++)
                cout << *it + 1 << ' ';
            cout << '\n';
            return;
        }
    }
    cout << "IMPOSSIBLE\n";
}

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);
    solve();
    return 0;
}
```

Fonte: o Autor.

Casos de borda e testes

A seguir, o Quadro 12 resume os principais casos de borda do Problema 7. Como a tarefa é encontrar um ciclo simples, os casos exploram a ausência de ciclos (florestas), ciclos mínimos C_3 e C_4, e a presença de um ciclo em apenas uma das componentes.

Destaca-se que um grafo acíclico (árvore ou floresta) deve produzir `IMPOSSIBLE`, e que basta um único ciclo em alguma componente para tornar a instância solúvel, independentemente das demais.

Quadro 12: Casos de borda e comportamento esperado no Problema 7

Caso	**Comportamento esperado**
$n = 1$ ou grafo com uma única aresta	`IMPOSSIBLE`
Floresta / árvore	`IMPOSSIBLE`
Ciclo C_3 (triângulo)	Ciclo de comprimento 4 na saída (início repetido)
Ciclo C_4	Ciclo de comprimento 5 na saída
Ciclo em uma componente, resto árvore	Qualquer ciclo da componente cíclica
Grafo desconexo com um ciclo	Ciclo encontrado na componente correspondente

Fonte: o Autor.

Exemplo de geração

O caso de borda do grafo ciclo C_n (Definição 14) no Quadro 12 pode ser produzido de forma sistemática com o método `cycle` da classe `Graph`. O Código 29 registra o programa como gerador via Testlib, lê o número de vértices n da linha de comando ($n \geq 3$) e imprime a instância no formato do enunciado (n e m na primeira linha, seguidos das arestas). Por exemplo, o parâmetro 4 gera um C_4.

Código 29 – Exemplo de gerador para o caso em ciclo do Problema 7, com o método `cycle` da classe `Graph`.

```cpp
#include "testlib.h"
#include "graphlib.h"

int main(int argc, char* argv[]) {
    registerGen(argc, argv, 1);

    int n = opt<int>(1);

    Graph g = Graph::cycle(n);
    g.printEdges();

    return 0;
}
```

Fonte: o Autor.

Resultados

A Tabela 8 instancia os casos de borda do Quadro 12 e inclui o exemplo do enunciado. Quando há vários ciclos válidos, indica-se uma saída possível aceita pelo juiz.

Tabela 8 – Resultados - Problema 7 (*Round Trip*)

Caso	**Entrada**	**Saída**
Exemplo do enunciado	5 6 1 3 1 2 5 3 1 5 2 4 4 5	4 1 3 5 1

Tabela 8 – Resultados - Problema 7 (*Round Trip*) (continuação)

Caso	**Entrada**	**Saída**
Árvore (caminho)	`4 3` 1 2 2 3 3 4	`IMPOSSIBLE`
Ciclo C_3	`3 3` 1 2 2 3 3 1	`4` 1 2 3 1
Ciclo C_4	`4 4` 1 2 2 3 3 4 4 1	`5` 1 2 3 4 1
Uma aresta ($n = 2$)	`2 1` 1 2	`IMPOSSIBLE`
Desconexo com ciclo	`5 4` 1 2 2 3 3 1 4 5	`4` 1 2 3 1

Fonte: o Autor.

5.8 Problema 8 - Contagem de sanduíches (Sanduíche)

Enunciado resumido

Fonte. "Sanduíche" (OBI 2021, Fase 2, Programação Nível Sênior).[8] A cada dia, o chef dispõe de n ingredientes distintos e de m pares de ingredientes que não podem aparecer juntos no mesmo sanduíche. Deseja-se contar quantos sanduíches diferentes podem ser montados. Dois sanduíches são distintos se os conjuntos de ingredientes diferem; todo sanduíche deve conter ao menos um ingrediente.

[8] Disponível em: <https://olimpiada.ic.unicamp.br/pratique/ps/2021/f2/sanduiche/>. Acesso em: 12 jul. 2026.

Entrada. A primeira linha contém dois inteiros n e m ($1 \leq n \leq 20$, $0 \leq m \leq 400$), o número de ingredientes e de pares incompatíveis. Os ingredientes são identificados por números de 1 a n. Cada uma das m linhas seguintes contém dois inteiros x e y ($1 \leq x < y \leq n$), indicando um par incompatível.

Saída. Imprima um único inteiro: o número de sanduíches distintos possíveis.

Solução

Os ingredientes são modelados como vértices e os pares incompatíveis como arestas de um grafo não direcionado simples $G = (V, E)$. Um sanduíche válido corresponde a um subconjunto não vazio $S \subseteq V$ em que nenhum par de vértices de S é adjacente, isto é, a um conjunto independente não vazio de G.

Como $n \leq 20$, é viável enumerar todos os 2^n subconjuntos por máscaras de bits. Define-se $\text{ok}[S] = 1$ se, e somente se, S é independente. O caso base é $\text{ok}[\emptyset] = 1$. Para $S \neq \emptyset$, escolhe-se um vértice $b \in S$ e escreve-se $S' = S \setminus \{b\}$; então $\text{ok}[S] = \text{ok}[S']$ e b não é vizinho de nenhum vértice de S'. A resposta é a soma de $\text{ok}[S]$ sobre todos os S não vazios. A complexidade assintótica de tempo de execução é $O(2^n + m)$ e a complexidade de memória é $O(2^n)$.

A presença de uma k-clique (Definição 10) restringe fortemente a contagem: de uma clique só é possível escolher no máximo um vértice. Em particular, se G for completo ($k = n$), restam apenas os n sanduíches com um único ingrediente.

Implementação

O Código 30 implementa a programação dinâmica descrita. Cada vértice i (em base 0) armazena em `adj[i]` a máscara de bits dos vizinhos. Para cada máscara não vazia, extrai-se o bit mais significativo b, recupera-se o estado do subconjunto sem b e verifica-se a ausência de conflito com `adj[b]`. Conta-se cada máscara independente.

Código 30 – Contagem de conjuntos independentes (DP em máscaras), baseado na OBI "Sanduíche"

```cpp
#include <bits/stdc++.h>

using namespace std;

void solve() {
    int n, m;
    cin >> n >> m;

    vector<int> adj(n);
    for (int i = 0; i < m; i++) {
        int x, y;
        cin >> x >> y;
```

```
        -x;
        -y;
        adj[x] |= (1 << y);
        adj[y] |= (1 << x);
    }

    vector<int> ok(1 << n);
    ok[0] = 1;
    int ans = 0;
    for (int mask = 1; mask < (1 << n); mask++) {
        int b = 31 - __builtin_clz(mask);
        int prev = mask - (1 << b);
        ok[mask] = ok[prev] && ((adj[b] & prev) == 0);
        if (ok[mask])
            ans++;
    }
    cout << ans << "\n";
}

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);
    solve();
    return 0;
}
```

Fonte: o Autor.

Casos de borda e testes

A seguir, o Quadro 13 resume os principais casos de borda do Problema 8. Como a tarefa é contar conjuntos independentes, os casos exploram ausência de restrições, grafos completos, cliques embutidas e instâncias mínimas.

Destaca-se que, sem pares incompatíveis, a resposta é $2^n - 1$, e que um grafo completo K_n reduz a resposta a exatamente n.

Quadro 13: Casos de borda e comportamento esperado no Problema 8

Caso	**Comportamento esperado**
$n = 1$, $m = 0$	1
Sem arestas ($m = 0$)	$2^n - 1$
Grafo completo K_n	n (apenas singletons)
Clique K_k embutida ($k < n$)	No máximo um vértice da clique por sanduíche
Triângulo (K_3) isolado em $n = 3$	3
Uma única aresta em $n = 2$	2 (os dois singletons)

Fonte: o Autor.

Exemplo de geração

O caso de borda com uma k-clique embutida no Quadro 13 pode ser produzido de forma sistemática com o método `withClique` da classe `Graph`. O Código 31 registra o programa como gerador via Testlib, lê n e k da linha de comando ($1 \leq k \leq n \leq 20$) e imprime a instância no formato do enunciado (n e m na primeira linha, seguidos das arestas). Quando $k = n$, o resultado coincide com o grafo completo K_n. Por exemplo, os parâmetros 5 e 3 geram um grafo conexo com 5 vértices contendo uma 3-clique.

Código 31 – Exemplo de gerador para o caso com clique do Problema 8, com o método `withClique` da classe `Graph`.

```cpp
#include "testlib.h"
#include "graphlib.h"

int main(int argc, char* argv[]) {
    registerGen(argc, argv, 1);

    int n = opt<int>(1);
    int k = opt<int>(2);

    Graph g = Graph::withClique(n, k);
    g.printEdges();

    return 0;
}
```

Fonte: o Autor.

Resultados

A Tabela 9 instancia os casos de borda do Quadro 13 e inclui os exemplos do enunciado.

Tabela 9 – Resultados - Problema 8 (Sanduíche)

Caso	**Entrada**	**Saída**

Exemplo 1 do enunciado	`3 2`
	`2 3`
	`1 2`
		`4`

Exemplo 2 do enunciado	`3 0`	`7`

Tabela 9 – Resultados - Problema 8 (Sanduíche) (continuação)

Caso	**Entrada**	**Saída**
Exemplo 3 do enunciado (K_3)	`3 3` 1 2 2 3 1 3	`3`
Um ingrediente	`1 0`	`1`
Uma aresta ($n = 2$)	`2 1` 1 2	`2`
Completo K_4	`4 6` 1 2 1 3 1 4 2 3 2 4 3 4	`4`

Fonte: o Autor.

5.9 Problema 9 - Conexão de componentes (*Building Roads*)

Enunciado resumido

Fonte. "*Building Roads*" (CSES Problem Set, task 1666).[9] Em Byteland há n cidades e m estradas bidirecionais. Deseja-se construir o menor número possível de novas estradas de modo que exista um caminho entre quaisquer duas cidades. Além do número mínimo, deve-se indicar quais estradas construir.

Entrada. A primeira linha contém dois inteiros n e m ($1 \leq n \leq 10^5$, $1 \leq m \leq 2 \cdot 10^5$), o número de cidades e de estradas. Cada uma das m linhas seguintes contém dois inteiros a e b ($1 \leq a, b \leq n$), indicando uma estrada entre a e b. O grafo é simples: não há laços nem arestas paralelas.

Saída. Imprima um inteiro k, o número mínimo de estradas a construir, e, nas k linhas seguintes, pares $u\,v$ descrevendo as novas estradas. Qualquer conjunto mínimo válido é aceito.

[9] Disponível em: <https://cses.fi/problemset/task/1666/>. Acesso em: 12 jul. 2026.

Solução

As cidades são modeladas como vértices e as estradas como arestas de um grafo não direcionado simples. O objetivo é tornar o grafo conexo (Definição 8) com o menor número de arestas adicionais.

Se o grafo possui c componentes conexas, bastam $c - 1$ arestas para conectá-las: escolhe-se um vértice representante em cada componente e liga-se esses representantes em cadeia. Uma busca em profundidade (ou em largura) identifica as componentes e coleta os representantes. A complexidade assintótica de tempo de execução e a complexidade de memória são $O(n + m)$ com lista de adjacências.

Quando a entrada já é uma floresta (Definição 16) com c árvores, a resposta permanece $c - 1$; esse formato é um caso de borda natural para testar a contagem de componentes.

Implementação

O Código 32 implementa a estratégia descrita. Após montar a lista de adjacências (vértices em base 0), a DFS marca vértices visitados; cada vértice ainda não marcado inicia uma nova componente e é registrado como representante. Em seguida, ligam-se representantes consecutivos e imprime-se o número de arestas adicionadas, seguido dos pares convertidos para a numeração $1..n$.

Código 32 – Conexão de componentes (DFS), baseado no CSES “Building Roads”

```cpp
#include <bits/stdc++.h>

using namespace std;

void solve() {
    int n, m;
    cin >> n >> m;

    vector<vector<int>> adj(n);
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        --u;
        --v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }

    vector<int> used(n);
    auto dfs = [&](auto&& dfs, int u) -> void {
        if (used[u])
            return;
        used[u] = true;
        for (auto v : adj[u])
```

```
            dfs(dfs, v);
    };

    vector<pair<int, int>> ans;
    vector<int> reps;
    for (int i = 0; i < n; i++) {
        if (!used[i]) {
            reps.push_back(i);
            dfs(dfs, i);
        }
    }
    for (int i = 1; i < (int)reps.size(); i++)
        ans.emplace_back(reps[i - 1], reps[i]);

    cout << ans.size() << "\n";
    for (auto [u, v] : ans)
        cout << u + 1 << " " << v + 1 << "\n";
}

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);
    solve();
    return 0;
}
```

Fonte: o Autor.

Casos de borda e testes

A seguir, o Quadro 14 resume os principais casos de borda do Problema 9. Como a tarefa é conectar componentes, os casos exploram grafos já conexos, florestas com várias árvores e vértices isolados.

Destaca-se que um grafo já conexo exige $k = 0$, e que c componentes isoladas (ou árvores disjuntas) exigem exatamente $c - 1$ estradas.

Quadro 14: Casos de borda e comportamento esperado no Problema 9

Caso	**Comportamento esperado**
Grafo já conexo ($c = 1$)	$k = 0$ (sem novas estradas)
Floresta com c componentes	$k = c - 1$
n vértices isolados	$k = n - 1$
Duas componentes (exemplo do enunciado)	$k = 1$
Uma aresta e vértices isolados	k igual ao número de componentes menos 1

Fonte: o Autor.

Exemplo de geração

O caso de borda em floresta do Quadro 14 pode ser produzido de forma sistemática com o método `forest` da classe `Graph`. O Código 33 registra o programa como gerador via Testlib, lê n e o número de componentes c da linha de comando ($1 \leq c \leq n$) e imprime a instância no formato do enunciado (n e m na primeira linha, seguidos das arestas). Por exemplo, os parâmetros 6 e 3 geram uma floresta com seis vértices e três componentes; a solução correspondente deve construir 2 estradas.

Código 33 – Exemplo de gerador para o caso em floresta do Problema 9, com o método `forest` da classe `Graph`.

```cpp
#include "testlib.h"
#include "graphlib.h"

int main(int argc, char* argv[]) {
    registerGen(argc, argv, 1);

    int n = opt<int>(1);
    int c = opt<int>(2);

    Graph g = Graph::forest(n, c);
    g.printEdges();

    return 0;
}
```

Fonte: o Autor.

Resultados

A Tabela 10 instancia os casos de borda do Quadro 14 e inclui o exemplo do enunciado. Quando há várias soluções mínimas, indica-se uma saída possível aceita pelo juiz.

Tabela 10 – Resultados - Problema 9 (*Building Roads*)

Caso	**Entrada**	**Saída**
Exemplo do enunciado	4 2 1 2 3 4	1 1 3
Já conexo (caminho)	4 3 1 2 2 3 3 4	0

Tabela 10 – Resultados - Problema 9 (*Building Roads*) (continuação)

Caso	**Entrada**	**Saída**
Três isolados	`3 0`	`2` 1 2 2 3
Uma aresta e um isolado	`3 1` 1 2	`1` 1 3

Fonte: o Autor.

5.10 Problema 10 - Ciclos em grafo de sucessores (*Planets Cycles*)

Enunciado resumido

Fonte. "*Planets Cycles*" (CSES Problem Set, task 1751).[10] Há n planetas e, em cada planeta i, um teletransportador que leva a um planeta t_i (possivelmente $t_i = i$). Partindo de um planeta, percorre-se a sequência de teletransportes até visitar novamente um planeta já encontrado. Deseja-se, para cada planeta de partida, o número de teletransportes efetuados até essa repetição.

Entrada. A primeira linha contém um inteiro n ($1 \leq n \leq 2 \cdot 10^5$), o número de planetas. A segunda linha contém n inteiros $t_1, t_2, \ldots, t_n$ ($1 \leq t_i \leq n$), o destino do teletransportador em cada planeta.

Saída. Imprima n inteiros: para cada planeta i, a quantidade de teletransportes a partir de i.

Solução

Os planetas são modelados como vértices e cada teletransportador como uma aresta $i \to t_i$. O resultado é um grafo de sucessores (Definição 17): todo vértice tem grau de saída exatamente igual a um. Em grafos finitos dessa família, cada componente contém exatamente um ciclo, eventualmente com caminhos (caudas) que conduzem a ele.

Para um vértice u no ciclo de comprimento ℓ, a resposta é ℓ: ao percorrer o ciclo, o primeiro planeta repetido é o próprio início após ℓ passos. Para um vértice na cauda, a resposta é a distância até o ciclo somada a ℓ. Uma DFS com memoização percorre cada vértice uma vez: ao encontrar um vértice já resolvido, propaga-se a resposta; ao detectar

[10] Disponível em: <https://cses.fi/problemset/task/1751/>. Acesso em: 12 jul. 2026.

um ciclo pelo vetor de distâncias no caminho atual, registra-se o comprimento do ciclo e preenche-se a resposta na volta da recursão. A complexidade assintótica de tempo de execução e a complexidade de memória são $O(n)$.

Implementação

O Código 34 implementa a estratégia descrita. Os vetores `succ`, `dis` e `ans` armazenam, respectivamente, o sucessor (em base 0), a profundidade no caminho da DFS atual e a resposta memoizada. A variável `startCycle` marca a profundidade do primeiro vértice do ciclo detectado, distinguindo na volta da recursão os vértices da cauda (que incrementam a resposta do sucessor) dos vértices do ciclo (que compartilham o mesmo valor ℓ).

Código 34 – Contagem de teletransportes em grafo de sucessores (DFS), baseado no CSES “Planets Cycles”

```cpp
#include <bits/stdc++.h>

using namespace std;

const int MAXN = 200001;

int dis[MAXN], succ[MAXN], ans[MAXN];

int startCycle = -1;

void dfs(int u) {
    int nxt = succ[u];
    if (ans[nxt] != 0) {
        ans[u] = ans[nxt] + 1;
        return;
    }

    if (dis[nxt] != 0) {
        ans[u] = dis[u] - dis[nxt] + 1;
        startCycle = dis[nxt];
        return;
    }

    dis[nxt] = dis[u] + 1;
    dfs(nxt);
    ans[u] = (dis[u] < startCycle || startCycle == -1) ? ans[nxt] + 1 : ans[nxt];
}

void solve() {
    int n;
    cin >> n;
    for (int i = 0; i < n; i++) {
        cin >> succ[i];
        -succ[i];
    }

```

```
37      for (int i = 0; i < n; i++) {
38          if (ans[i] == 0) {
39              dis[i] = 1;
40              startCycle = -1;
41              dfs(i);
42          }
43      }
44
45      for (int i = 0; i < n; i++)
46          cout << ans[i] << " ";
47      cout << "\n";
48  }
49
50  int main() {
51      ios_base::sync_with_stdio(0);
52      cin.tie(0);
53      solve();
54      return 0;
55  }
```

Fonte: o Autor.

Casos de borda e testes

A seguir, o Quadro 15 resume os principais casos de borda do Problema 10. Como a estrutura é um grafo funcional, os casos exploram laços, ciclos puros, caudas que alimentam um ciclo e múltiplas componentes.

Destaca-se que $t_i = i$ produz resposta 1, e que vértices em um mesmo ciclo compartilham a mesma resposta (o comprimento do ciclo).

Quadro 15: Casos de borda e comportamento esperado no Problema 10

Caso	**Comportamento esperado**
$n = 1$, $t_1 = 1$	1
Laço ($t_i = i$)	Resposta 1 naquele vértice
Ciclo puro de comprimento ℓ	ℓ em todos os vértices do ciclo
Cauda conduzindo a um ciclo	Distância ao ciclo + comprimento do ciclo
Várias componentes funcionais	Resolver independentemente por componente
Exemplo do enunciado	3 3 1 3 4

Fonte: o Autor.

Exemplo de geração

O grafo de sucessores do Quadro 15 pode ser produzido de forma sistemática com o método `successors` da classe `Graph`. O Código 35 registra o programa como gerador

via Testlib, lê o número de planetas n da linha de comando e imprime a instância no formato do enunciado (n na primeira linha e a lista de sucessores $t_1, \ldots, t_n$ na segunda). Por exemplo, o parâmetro 5 gera um grafo funcional aleatório com cinco vértices.

Código 35 – Exemplo de gerador para o caso de sucessores do Problema 10, com o método `successors` da classe `Graph`.

```cpp
#include "testlib.h"
#include "graphlib.h"

int main(int argc, char* argv[]) {
    registerGen(argc, argv, 1);

    int n = opt<int>(1);
    Graph g = Graph::successors(n);

    std::cout << n << '\n';
    for (int i = 1; i <= n; ++i) {
        if (i > 1)
            std::cout << ' ';
        std::cout << g.adj[i][0];
    }
    std::cout << '\n';

    return 0;
}
```

Fonte: o Autor.

Resultados

A Tabela 11 instancia os casos de borda do Quadro 15 e inclui o exemplo do enunciado.

Tabela 11 – Resultados - Problema 10 (*Planets Cycles*)

Caso	**Entrada**	**Saída**
Exemplo do enunciado	5 2 4 3 1 4	3 3 1 3 4
Um planeta (laço)	1 1	1
Ciclo de comprimento 2	2 2 1	2 2
Todos laços	3 1 2 3	1 1 1

Tabela 11 – Resultados - Problema 10 (*Planets Cycles*)
(continuação)

Caso	**Entrada**	**Saída**
Cauda e ciclo	`4` 2 3 4 3	`4 3 2 2`

Fonte: o Autor.

6 Considerações Finais

Neste trabalho, foram analisados casos de borda em problemas de grafos no contexto da programação competitiva e implementada a classe `Graph` para a geração de instâncias alinhadas a famílias notáveis. Entre as dificuldades encontradas, destacam-se a modelagem dessa classe de forma a atender a diversos tipos de grafos e a seleção de problemas em que um caso de borda pudesse ser associado de forma específica a um determinado tipo de grafo. Foi particularmente desafiador o pareamento entre cada gerador da classe `Graph` e um enunciado de competição: nem todo problema admitia, de modo natural, a exemplificação do método correspondente, o que levou à substituição de alguns enunciados por outros que se encaixavam melhor nas famílias do Quadro 3.

Foi possível constatar que a classe `Graph` atendeu ao propósito de gerar as famílias notáveis selecionadas e que ela pode auxiliar elaboradores de problemas (*setters*) na geração sistemática de testes. Para cada um dos dez problemas do Capítulo 5, documentou-se um exemplo de gerador com o método correspondente da classe, ilustrando a construção de instâncias de teste a partir das famílias notáveis. A aplicação aos problemas ocorreu em etapa posterior à implementação da classe e exigiu o realinhamento da seleção de enunciados descrito acima. Com a experiência acumulada, observa-se que a classe poderia ser estendida para cobrir características adicionais, como peso nas arestas, conectividade e ciclicidade.

Espera-se que este material auxilie competidores, elaboradores de problemas e técnicos a obter uma visão geral dos casos de borda em grafos, dos tipos de grafo a considerar na criação e na resolução de enunciados, e do uso dos trechos de código apresentados para a geração de instâncias.

6.1 Trabalhos futuros

Levando em conta as discussões apresentadas nos capítulos anteriores, a lista a seguir é uma sugestão de trabalhos futuros que podem complementar os resultados deste estudo:

- incrementar a classe `Graph` para gerar, sobre as famílias já implementadas, características transversais ainda sem gerador dedicado, como peso nas arestas, densidades controladas, conectividade e ciclicidade;
- acrescentar validadores e *checkers* com a Testlib, de modo a fechar o fluxo de geração, validação de entrada e verificação de saída nos moldes do ecossistema Polygon.

Referências

ALVES, E. *TEP: Material da disciplina Tópicos Especiais em Programação*. 2026. <https://github.com/edsomjr/TEP>. Acesso em: 18 jun. 2026. Citado 2 vezes nas páginas 27 e 28.

BRANDSTÄDT, A.; LE, V. B.; SPINRAD, J. P. *Graph Classes: A Survey*. Philadelphia: Society for Industrial and Applied Mathematics (SIAM), 1999. (SIAM Monographs on Discrete Mathematics and Applications). Citado 2 vezes nas páginas 36 e 41.

Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq). *Portaria CNPq Nž 2.664, de 6 de março de 2026: Institui a Política de Integridade na Atividade Científica do CNPq*. 2026. <http://memoria2.cnpq.br/web/guest/view/-/journal_content/56_INSTANCE_0oED/10157/23142775>. Publicado no DOU de 11 mar. 2026, Seção 1, p. 4. Acesso em: 10 jul. 2026. Citado na página 40.

CORMEN, T. H. et al. *Introduction to Algorithms*. 3. ed. [S.l.]: MIT press, 2009. Citado 9 vezes nas páginas 20, 23, 24, 26, 27, 28, 29, 36 e 47.

DIESTEL, R. *Graph Theory*. 5. ed. [S.l.]: Springer, 2017. Citado 8 vezes nas páginas 14, 20, 21, 30, 36, 47, 51 e 52.

HALIM, S.; HALIM, F.; EFFENDY, S. *Competitive Programming 4 – Book 1: The Lower Bound of Programming Contests in the 2020s*. [S.l.]: CP4 Books, 2020. ISBN 978-1716745529. Citado 6 vezes nas páginas 14, 16, 36, 55, 56 e 86.

HARRIS, J. M.; HIRST, J. L.; MOSSINGHOFF, M. J. *Combinatorics and graph theory*. New York, US: [s.n.], 2008. Citado na página 41.

LAAKSONEN, A. *Competitive Programmer's Handbook*. Self-published, 2018. Disponível em: <https://cses.fi/book.html>. Citado 10 vezes nas páginas 14, 16, 20, 36, 53, 54, 59, 64, 68 e 86.

MALETTA, B. *tgen – A C++ header for writing random testcase generators quickly and safely*. <https://github.com/brunomaletta/tgen>. Acesso em: 5 jul. 2026. Citado 3 vezes nas páginas 14, 32 e 36.

MATTIOLI, L. V. *Comparativo entre a estratégia gulosa e a programação dinâmica para o problema do troco com sistemas de moedas canônicos*. 59 p. Monografia (Trabalho de Conclusão de Curso (Bacharelado em Engenharia de Software)) — Universidade de Brasília, Brasília, 2018. Acesso em: 21 jun. 2026. Disponível em: <https://bdm.unb.br/handle/10483/21568>. Citado na página 33.

MIRZAYANOV, M. *Testlib – A C++ library for creating test generators and validators*. 2005. <https://github.com/MikeMirzayanov/testlib>. Accessed: 2025-05-26. Citado 3 vezes nas páginas 14, 30 e 39.

MIRZAYANOV, M. *Generators with testlib.h*. 2015. <https://codeforces.com/blog/entry/18291>. Acesso em: 9 jul. 2026. Citado 2 vezes nas páginas 30 e 31.

SEDGEWICK, R.; WAYNE, K. *Algorithms. 4. ed.* Boston: Addison-Wesley: [s.n.], 2011. Citado na página 45.

SINGH, K. et al. No stress no gain: Stress testing based self-consistency for olympiad programming. In: *ICLR 2025 Workshops: VerifAI*. [s.n.], 2025. Disponível em: <https://openreview.net/forum?id=7SlCSjhBsq>. Citado na página 33.

SKIENA, S. S.; REVILLA, M. A. *Programming Challenges: The Programming Contest Training Manual*. [S.l.]: Springer, 2003. Citado 11 vezes nas páginas 14, 19, 21, 22, 23, 24, 26, 28, 29, 30 e 56.

SMIRNOV, I. *jngen – Library for generating tests for olympiad problems*. <https://github.com/ifsmirnov/jngen>. Acesso em: 5 jul. 2026. Citado 6 vezes nas páginas 14, 32, 36, 45, 47 e 56.

VATER, L.; TANNEBERG, N.; MAURER, M. A systematic approach towards the definition of the terms edge case and corner case for automated driving. In: *Proceedings of the 2023 International Conference on Electrical, Computer and Communication Engineering (ICECCME)*. [S.l.]: IEEE, 2023. p. 1–6. Citado na página 33.

WEST, D. B. *Introduction to Graph Theory*. 2. ed. [S.l.]: Prentice Hall, 2001. Citado 8 vezes nas páginas 14, 36, 41, 43, 48, 49, 51 e 52.

YAN, W. et al. Graphfuzz: Automated testing of graph algorithm implementations with differential fuzzing and lightweight feedback. *arXiv preprint arXiv:2502.15160*, 2025. Disponível em: <https://arxiv.org/abs/2502.15160>. Citado na página 33.

Apêndices

APÊNDICE A – Código-fonte da classe Graph

Este apêndice reproduz a implementação completa da classe `Graph` empregada na geração de instâncias de grafos notáveis (Capítulo 4) e nos exemplos de geradores dos problemas analisados (Capítulo 5). No corpo do texto, apresentam-se apenas a interface da classe e os trechos correspondentes a cada método. O código-fonte correspondente, sem o texto desta monografia, está disponível publicamente em <https://github.com/andremralves/graphlib>.

Código 36 – Implementação completa da classe `Graph`.

```
#ifndef GRAPHLIB_H
#define GRAPHLIB_H

#include "testlib.h"

#include <algorithm>
#include <iostream>
#include <queue>
#include <set>
#include <utility>
#include <vector>

/**
 * Grafo por lista de adjacências, com vértices numerados de 1 a n.
 * Pensado para geradores de testes com Testlib.
 */
class Graph {
public:
    int n;
    bool directed;
    std::vector<std::vector<int>> adj;
    // Vazio = sem enraizamento; tamanho n+1 = árvore enraizada em 1 (parent[1]=0).
    std::vector<int> parent;

    Graph(int n = 0, bool directed = false);

    int m() const;

    void addEdge(int u, int v);

    static Graph complete(int n, bool directed = false);

    static Graph completeBipartite(int n1, int n2);

    static Graph randomBipartite(int n1, int n2, int m);

```

```
37      static Graph star(int n);
38
39      static Graph bamboo(int n);
40
41      static Graph cycle(int n);
42
43      static Graph tree(int n, int elongation = 0);
44
45      static Graph forest(int n, int components, int elongation = 0);
46
47      static Graph successors(int n);
48
49      static Graph dag(int n, int m);
50
51      static Graph withClique(int n, int k);
52
53      void printNM(std::ostream& out = std::cout) const;
54
55      void printEdges(std::ostream& out = std::cout) const;
56
57      void printParents(std::ostream& out = std::cout) const;
58
59      std::vector<std::pair<int, int>> edges() const;
60
61  private:
62      void ensureInRange(int u) const;
63
64      void fillParentsFromRoot1();
65
66      static void addRandomTreeEdges(Graph& g, const std::vector<int>& verts, int elongation);
67  };
68
69  // -- implementações --
70
71  /**
72   * Constrói um grafo com n vértices numerados de 1 a n.
73   * @param n número de vértices (>= 0)
74   * @param directed se verdadeiro, o grafo é direcionado
75   */
76  inline Graph::Graph(int n, bool directed) : n(n), directed(directed), adj(n + 1) {}
77
78  /**
79   * Retorna o número de arestas do grafo.
80   * Em grafos não direcionados, cada par de vértices adjacentes conta uma vez.
81   */
82  inline int Graph::m() const {
83      int edges = 0;
84      for (int u = 1; u <= n; ++u)
85          edges += (int)adj[u].size();
86      return directed ? edges : edges / 2;
87  }
88
89  /**
90   * Insere uma aresta entre u e v (ou uma aresta direcionada u -> v, se o grafo for
    ↪  direcionado).
91   * @param u vértice de origem (1 <= u <= n)
92   * @param v vértice de destino (1 <= v <= n)
```

```
93  */
94 inline void Graph::addEdge(int u, int v) {
95     ensureInRange(u);
96     ensureInRange(v);
97     adj[u].push_back(v);
98     if (!directed)
99         adj[v].push_back(u);
100    parent.clear();
101 }
102
103 inline void Graph::ensureInRange(int u) const {
104    if (u < 1 || u > n)
105        quitf(_fail, "vertex %d out of range [1, %d]", u, n);
106 }
107
108 /**
109  * Grafo completo K_n.
110  * @param n número de vértices (>= 1)
111  * @param directed se verdadeiro, gera arestas direcionadas em ambos os sentidos entre pares
    ↪  distintos
112  */
113 inline Graph Graph::complete(int n, bool directed) {
114    if (n < 1)
115        quitf(_fail, "Graph::complete requires n >= 1, got %d", n);
116
117    Graph g(n, directed);
118
119    if (directed) {
120        for (int u = 1; u <= n; ++u)
121            for (int v = 1; v <= n; ++v)
122                if (u != v)
123                    g.addEdge(u, v);
124    } else {
125        for (int u = 1; u <= n; ++u)
126            for (int v = u + 1; v <= n; ++v)
127                g.addEdge(u, v);
128    }
129
130    return g;
131 }
132
133 /**
134  * Grafo bipartido completo K_{n1,n2}.
135  * Partição U: 1..n1; partição V: n1+1..n1+n2.
136  * @param n1 tamanho de U (>= 0)
137  * @param n2 tamanho de V (>= 0)
138  */
139 inline Graph Graph::completeBipartite(int n1, int n2) {
140    if (n1 < 0 || n2 < 0)
141        quitf(_fail, "Graph::completeBipartite requires n1, n2 >= 0, got (%d, %d)", n1, n2);
142
143    Graph g(n1 + n2, false);
144    for (int u = 1; u <= n1; ++u)
145        for (int v = n1 + 1; v <= n1 + n2; ++v)
146            g.addEdge(u, v);
147
148    return g;
```

```
149 }
150
151 /**
152  * Grafo bipartido aleatório com exatamente m arestas entre as partições.
153  * @param n1 tamanho de U (>= 0)
154  * @param n2 tamanho de V (>= 0)
155  * @param m número de arestas (0 <= m <= n1*n2)
156  */
157 inline Graph Graph::randomBipartite(int n1, int n2, int m) {
158     if (n1 < 0 || n2 < 0)
159         quitf(_fail, "Graph::randomBipartite requires n1, n2 >= 0, got (%d, %d)", n1, n2);
160     if (m < 0)
161         quitf(_fail, "Graph::randomBipartite requires m >= 0, got %d", m);
162
163     long long maxEdges = 1LL * n1 * n2;
164     if (m > maxEdges)
165         quitf(_fail, "Graph::randomBipartite: too many edges (%d > %lld)", m, maxEdges);
166
167     Graph g(n1 + n2, false);
168     if (m == 0)
169         return g;
170
171     if (n1 == 0 || n2 == 0)
172         quitf(_fail, "Graph::randomBipartite: cannot add edges when n1 or n2 is 0");
173
174     std::set<std::pair<int, int>> used;
175     while ((int)used.size() < m) {
176         int u = rnd.next(1, n1);
177         int v = rnd.next(n1 + 1, n1 + n2);
178         used.emplace(u, v);
179     }
180
181     for (const auto& [u, v] : used)
182         g.addEdge(u, v);
183
184     return g;
185 }
186
187 /**
188  * Grafo estrela não direcionado: vértice 1 no centro, 2..n folhas.
189  * @param n número de vértices (>= 1)
190  * Preenche parent enraizado em 1.
191  */
192 inline Graph Graph::star(int n) {
193     if (n < 1)
194         quitf(_fail, "Graph::star requires n >= 1, got %d", n);
195
196     Graph g(n, false);
197     for (int i = 2; i <= n; ++i)
198         g.addEdge(1, i);
199     g.fillParentsFromRoot1();
200
201     return g;
202 }
203
204 /**
205  * Grafo caminho P_n (bambu), não direcionado.
```

```
206  * Permuta os rótulos 1..n e liga pares consecutivos.
207  * @param n número de vértices (>= 1)
208  * Preenche parent enraizado em 1.
209  */
210 inline Graph Graph::bamboo(int n) {
211     if (n < 1)
212         quitf(_fail, "Graph::bamboo requires n >= 1, got %d", n);
213 
214     Graph g(n, false);
215     std::vector<int> order = rnd.perm(n, 1);
216     for (int i = 0; i < n - 1; ++i)
217         g.addEdge(order[i], order[i + 1]);
218     g.fillParentsFromRoot1();
219 
220     return g;
221 }
222 
223 /**
224  * Grafo ciclo C_n não direcionado.
225  * @param n número de vértices (>= 3)
226  */
227 inline Graph Graph::cycle(int n) {
228     if (n < 3)
229         quitf(_fail, "Graph::cycle requires n >= 3, got %d", n);
230 
231     Graph g(n, false);
232 
233     for (int i = 1; i < n; ++i)
234         g.addEdge(i, i + 1);
235     g.addEdge(n, 1);
236 
237     return g;
238 }
239 
240 /**
241  * Insere em g as arestas de uma árvore aleatória não direcionada sobre os rótulos em verts.
242  * Processo tipo Prim (mesmo de tree).
243  */
244 inline void Graph::addRandomTreeEdges(Graph& g, const std::vector<int>& verts, int elongation) {
245     int k = (int)verts.size();
246     if (k <= 1)
247         return;
248 
249     std::vector<int> parent(k);
250     parent[0] = -1;
251     for (int v = 1; v < k; ++v)
252         parent[v] = rnd.wnext(v, elongation);
253 
254     std::vector<int> local = rnd.perm(k, 0);
255 
256     for (int v = 1; v < k; ++v) {
257         int u = verts[local[parent[v]]];
258         int w = verts[local[v]];
259         g.addEdge(u, w);
260     }
261 }
262 
```

```
263 /**
264  * Árvore aleatória não direcionada com n vértices (processo tipo Prim / jngen
    ↪  Tree::randomPrim).
265  * Usa rnd.wnext(v, elongation) e permuta índices antes de inserir arestas.
266  * elongation > 0: alongada; < 0: estrelada; 0: uniforme.
267  * @param n número de vértices (>= 1)
268  * @param elongation enviesamento de rnd.wnext
269  * Preenche parent enraizado em 1.
270  */
271 inline Graph Graph::tree(int n, int elongation) {
272     if (n < 1)
273         quitf(_fail, "Graph::tree requires n >= 1, got %d", n);
274 
275     Graph g(n, false);
276     std::vector<int> verts(n);
277     for (int i = 0; i < n; ++i)
278         verts[i] = i + 1;
279     addRandomTreeEdges(g, verts, elongation);
280     g.fillParentsFromRoot1();
281     return g;
282 }
283 
284 /**
285  * Floresta aleatória não direcionada com n vértices e exatamente components componentes.
286  * Particiona rótulos (rnd.partition + rnd.perm) e gera uma árvore em cada parte.
287  * @param n número de vértices (>= 0)
288  * @param components número de componentes (1 <= components <= n; 0 se n = 0)
289  * @param elongation repassado a cada árvore
290  */
291 inline Graph Graph::forest(int n, int components, int elongation) {
292     if (n < 0)
293         quitf(_fail, "Graph::forest requires n >= 0, got %d", n);
294 
295     if (n == 0) {
296         if (components != 0)
297             quitf(_fail, "Graph::forest: n = 0 requires components = 0, got %d", components);
298         return Graph(0, false);
299     }
300 
301     if (components < 1 || components > n)
302         quitf(_fail, "Graph::forest requires 1 <= components <= n, got components=%d, n=%d",
303               components, n);
304 
305     std::vector<int> sizes = rnd.partition(components, n);
306     std::vector<int> order = rnd.perm(n, 1);
307 
308     Graph g(n, false);
309     int offset = 0;
310     for (int size : sizes) {
311         std::vector<int> verts(size);
312         for (int i = 0; i < size; ++i)
313             verts[i] = order[offset + i];
314         offset += size;
315         addRandomTreeEdges(g, verts, elongation);
316     }
317 
318     return g;
```

```
}

/**
 * Grafo de sucessores: cada vértice tem exatamente uma aresta de saída.
 * @param n número de vértices (>= 1)
 */
inline Graph Graph::successors(int n) {
    if (n < 1)
        quitf(_fail, "Graph::successors requires n >= 1, got %d", n);

    Graph g(n, true);
    for (int i = 1; i <= n; ++i)
        g.addEdge(i, rnd.next(1, n));

    return g;
}

/**
 * DAG com n vertices e exatamente m arestas (estilo jngen).
 * Permutação define ordem topológica; só aceita arestas que a respeitam.
 * @param n número de vértices (>= 1)
 * @param m número de arestas (0 <= m <= n*(n-1)/2)
 */
inline Graph Graph::dag(int n, int m) {
    if (n < 1)
        quitf(_fail, "Graph::dag requires n >= 1, got %d", n);
    if (m < 0)
        quitf(_fail, "Graph::dag requires m >= 0, got %d", m);

    long long maxEdges = 1LL * n * (n - 1) / 2;
    if (m > maxEdges)
        quitf(_fail, "Graph::dag: too many edges (%d > %lld)", m, maxEdges);

    Graph g(n, true);
    std::vector<int> order = rnd.perm(n, 1);
    std::set<std::pair<int, int>> edges;

    while ((int)edges.size() < m) {
        int i = rnd.next(0, n - 2);
        int j = rnd.next(i + 1, n - 1);
        edges.emplace(order[i], order[j]);
    }

    for (const auto& [u, v] : edges)
        g.addEdge(u, v);

    return g;
}

/**
 * Grafo conexo com n vertices contendo uma k-clique.
 * @param n número de vértices (>= 1)
 * @param k tamanho da clique (1 <= k <= n)
 */
inline Graph Graph::withClique(int n, int k) {
    if (n < 1)
        quitf(_fail, "Graph::withClique requires n >= 1, got %d", n);
```

```
376     if (k < 1 || k > n)
377         quitf(_fail, "Graph::withClique requires 1 <= k <= n, got k=%d, n=%d", k, n);
378
379     Graph g(n, false);
380     std::vector<int> order = rnd.perm(n, 1);
381
382     for (int i = 0; i < k; ++i)
383         for (int j = i + 1; j < k; ++j)
384             g.addEdge(order[i], order[j]);
385
386     for (int i = k; i < n; ++i)
387         g.addEdge(order[i], order[rnd.next(0, i - 1)]);
388
389     return g;
390 }
391
392 /**
393  * Imprime n e m na primeira linha.
394  * @param out fluxo de saída (padrão: std::cout)
395  */
396 inline void Graph::printNM(std::ostream& out) const {
397     out << n << ' ' << m() << '\n';
398 }
399
400 /**
401  * Imprime n e m na primeira linha, seguido de uma aresta por linha (u v), em ordem aleatória.
402  * @param out fluxo de saída (padrão: std::cout)
403  */
404 inline void Graph::printEdges(std::ostream& out) const {
405     printNM(out);
406     std::vector<std::pair<int, int>> es = edges();
407     shuffle(es.begin(), es.end());
408     for (const auto& [u, v] : es)
409         out << u << ' ' << v << '\n';
410 }
411
412 /**
413  * Imprime parent[2]..parent[n] numa linha (árvore enraizada em 1).
414  * Exige que parent tenha sido preenchido (star, bamboo ou tree).
415  * @param out fluxo de saída (padrão: std::cout)
416  */
417 inline void Graph::printParents(std::ostream& out) const {
418     if (parent.empty())
419         quitf(_fail, "Graph::printParents requires a rooted tree (parent vector empty)");
420     for (int i = 2; i <= n; ++i) {
421         if (i > 2)
422             out << ' ';
423         out << parent[i];
424     }
425     out << '\n';
426 }
427
428 /**
429  * Lista de arestas sem duplicata.
430  * Não direcionado: uma entrada (u, v) com u < v por aresta.
431  * Direcionado: uma aresta (u, v) por entrada em adj.
432  */
```

```
433 inline std::vector<std::pair<int, int>> Graph::edges() const {
434     std::vector<std::pair<int, int>> es;
435     for (int u = 1; u <= n; ++u) {
436         for (int v : adj[u]) {
437             if (directed || u < v)
438                 es.emplace_back(u, v);
439         }
440     }
441     return es;
442 }
443 
444 /**
445  * Preenche parent com a árvore enraizada no vértice 1 (parent[1] = 0).
446  * Em grafos direcionados, trata arestas como bidirecionais só para a relação pai/filho.
447  */
448 inline void Graph::fillParentsFromRoot1() {
449     parent.assign(n + 1, 0);
450     if (n == 0)
451         return;
452 
453     std::vector<std::vector<int>> undirected(n + 1);
454     if (directed) {
455         for (const auto& [u, v] : edges()) {
456             undirected[u].push_back(v);
457             undirected[v].push_back(u);
458         }
459     } else {
460         undirected = adj;
461     }
462 
463     std::vector<char> visited(n + 1, 0);
464     std::queue<int> q;
465     visited[1] = 1;
466     q.push(1);
467     int seen = 1;
468     while (!q.empty()) {
469         int u = q.front();
470         q.pop();
471         for (int v : undirected[u]) {
472             if (visited[v])
473                 continue;
474             visited[v] = 1;
475             parent[v] = u;
476             q.push(v);
477             ++seen;
478         }
479     }
480 
481     if (seen != n)
482         quitf(_fail, "Graph::fillParentsFromRoot1: not a tree rooted at 1 (%d/%d reachable)",
483               seen, n);
484 }
485 
486 #endif
```

Fonte: o Autor.