Universidade de São Paulo
Instituto de Matemática e Estatística
Bachalerado em Ciência da Computação

Gabriel de Russo e Carmo

O problema da conectividade dinâmica em grafos

São Paulo
Novembro de 2018

O problema da conectividade dinâmica em grafos

Monografia final da disciplina
MAC0499 – Trabalho de Formatura Supervisionado.

Supervisor: Prof. Dr. José Coelho de Pina Junior

São Paulo
Novembro de 2018

Agradecimentos

Agradeço muito ao professor Coelho pelo seu tempo, por sua paciência, por me ajudar a contornar os problemas mais difíceis que surgiram pelo caminho e por me ensinar um jeito diferente de estudar.

Agradeço também a professora Yoshiko Wakabayashi, que ministrou a disciplina Introdução à Teoria dos Grafos de forma brilhante, contribuindo muito para meu interesse e entendimento da teoria dos grafos de forma geral.

Por fim, agradeço à todos os colegas do grupo de estudos para programação competitiva, o MaratonIME, pois sempre me motivaram a continuar estudando e me ensinaram que nenhum problema é difícil demais.

Resumo

No problema da conectividade dinâmica estamos interessados em manter um grafo sujeito a atualizações e consultas de conectividade. Uma atualização é uma adição ou remoção de aresta. Uma consulta de conectividade deseja saber se existe um caminho entre dois vértices. A solução apresentada mantém diversas florestas geradoras do grafo. Cada árvore das florestas geradoras é representada por uma árvore de busca binária balanceada. O consumo de tempo amortizado das operações é polilogarítmico. A solução foi implementada em C++. Seu desempenho foi testado na prática.

Palavras-chave: grafo, conectividade dinâmica, HDT, árvore de trilha eureliana.

Abstract

In the dynamic connectivity problem we are interested in maintaining a graph subject to updates and connectivity queries. An update is an insertion or deletion of an edge. A connectivity query asks if there is a path between two vertices. The presented solution keeps multiple spanning forests of the graph. Each tree of the spanning forests is represented by a balanced binary search tree. The amortized time consumption of each operation is polylogarithmic. The solution was implemented in C++. Its performance was tested in practice.

Keywords: graph, dynamic connectivity, HDT, euler tour tree.

Sumário

Capítulo 1

Introdução

No problema da conectividade dinâmica em grafos, estamos interessados em manter um grafo que está sujeito a atualizações e consultas. Uma atualização é uma adição ou remoção de aresta, enquanto uma consulta deseja saber se dois vértices estão conectados.

O problema vem sendo estudado por diversos acadêmicos ao longo das últimas décadas. Acredita-se que a primeira solução não-trivial do problema foi publicada há mais de 30 anos (Frederickson, 1983). Uma das soluções mais recentes foi publicada há pouco mais de dois anos (Huang *et al.*, 2016).

Nesta dissertação, analisamos a solução de Holm *et al.* (2001). Sem deixar a teoria de lado, a solução foi completamente implementada e seu desempenho foi testado na prática.

1.1 Objetivos

O trabalho tem como objetivo principal entender o problema e apresentar uma solução de forma detalhada. Grande parte dos artigos que tratam do problema são de alto nível e deixam de lado detalhes importantes da implementação. Por esse motivo, há um enfoque nas particularidades das estruturas de dados utilizadas e uma implementação da solução numa linguagem de programação moderna.

A criação de um material de estruturas de dados avançadas em língua portuguesa também é levada em consideração, uma vez que a maioria dos livros e artigos sobre o assunto só está disponível em outras línguas.

1.2 Estrutura do texto

O texto foi dividido em múltiplos capítulos e está organizado do maior nível de abstração para o menor. O leitor é aconselhado a seguir o trabalho na ordem proposta.

O capítulo 2 trata de noções básicas da teoria dos grafos, tópico essencial para o entendimento do problema e da solução. Nele são apresentadas definições, notações e algumas propriedades importantes que serão utilizadas ao longo do trabalho.

O capítulo 3 introduz o problema de forma exemplificada. Apresenta os conceitos mais importantes da solução, tem enfoque teórico e pseudocódigo. Alguns problemas são deixados como pendência e são resolvidos em outros capítulos.

O capítulo 4 apresenta o problema da conectividade dinâmica em florestas, uma versão restrita do problema. Sua solução é uma peça fundamental para a solução do problema original. Também tem enfoque teórico, pseudocódigo e algumas pendências que são resolvidas mais tarde.

O capítulo 5 apresenta soluções para as pendências em aberto dos capítulos anteriores. Trata de árvores de busca binária e das modificações necessárias nessa estrutura para resolver o problema original de forma eficiente.

O capítulo 6 recapitula as informações dos capítulos anteriores e dá detalhes da implementação de todas as estruturas de dados discutidas ao longo do trabalho. Nele encontra-se uma implementação em C++.

Por fim, o capítulo 7 trata dos testes que foram realizados durante o desenvolvimento da solução e do desempenho da implementação.

Capítulo 2

Grafos

Este capítulo formaliza definições importantes sobre grafos e introduz conceitos que serão usados extensivamente ao longo do trabalho.

2.1 Vértices e arestas

Um grafo é um par ordenado (V, A), onde V e A são conjuntos disjuntos finitos. Cada elemento de A é um par não-ordenado de elementos de V. Os elementos de V são chamados de **vértices**. Os elementos de A são chamados de **arestas**.

Se v e w são vértices em V e $\{v, w\}$, ou simplesmente vw, é uma aresta em A, então v e w são as **pontas da aresta** vw. Dizemos que v e w são **vizinhos** ou **adjacentes**.

Se $G = (V, A)$ é um grafo, dizemos que $G' = (V', A')$ é um **subgrafo** de G, ou simplesmente $G' \subseteq G$, se V' é subconjunto de V e A' é subconjunto de A. Um subgrafo $G' = (V', A')$ é dito **gerador** de $G = (V, A)$ se $G' \subseteq G$ e $V = V'$.

2.2 Caminhos e circuitos

Um caminho em um grafo $G = (V, A)$ é uma sequência $v_0a_1v_1a_2v_2 \dots a_pv_p$ tal que v_i é um vértice em V para todo $i = 0, 1, \dots, p$, a_i é uma aresta em A para todo $i = 1, 2, \dots, p$, os vértices são distintos dois a dois e $a_i = v_{i-1}v_i$. Os vértices v_0 e v_p são chamados de **pontas do caminho**. Dizemos que dois vértices v e w estão **conectados** se existe um caminho com as pontas v e w.

Um **circuito** em um grafo $G = (V, A)$ é uma sequência $v_1a_1v_2a_2v_3 \dots v_pa_p$ tal que v_i é um vértice de V para todo $i = 1, 2, \dots, p$, os vértices são distintos dois a dois, $a_i = v_iv_{i+1}$ para todo $i = 1, 2, \dots, p-1$ e $a_p = v_pv_1$.

Um grafo é dito **conexo** se para todo v e w em V existe um caminho com pontas v e w. Um subgrafo conexo maximal é um **componente** do grafo. Vale ressaltar que um grafo conexo só tem um componente.

2.3 Florestas e árvores

Chamamos um grafo sem circuitos de **floresta**. Uma **árvore** é uma floresta conexa. Logo, cada componente de uma floresta é uma árvore.

Afirmamos sem prova que todo grafo conexo tem uma **árvore geradora**, isto é, um subgrafo gerador que é uma árvore. Se o grafo não é conexo, ele tem uma **floresta geradora**, isto é, um subgrafo que contém uma árvore geradora de cada um de seus componentes.

Equivalentemente, uma floresta geradora é um subgrafo acíclico maximal. Um exemplo é dado na figura 2.1, onde arestas pontilhadas fazem parte do grafo e arestas destacadas fazem parte da floresta.

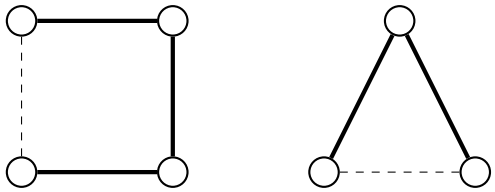

Figura 2.1: *Exemplo de floresta geradora.*

Florestas geradoras preservam informações de conectividade. Dado um grafo G, dois vértices v e w e uma floresta geradora F, temos que v e w estão conectados em G se e somente se estão conectados em F. De fato, se v e w estão conectados em G, então v e w fazem parte da mesma componente de G, que tem uma árvore geradora em F. Por outro lado, se v e w estão conectados em F, devem estar conectados em G já que $F \subseteq G$.

Capítulo 3

Conectividade dinâmica

No problema da **conectividade dinâmica** estamos interessados em manter um grafo G sujeito a atualizações e consultas. O conjunto de vértices de G é fixo. Uma **atualização** adiciona ou remove uma aresta. Uma **consulta** é uma pergunta do tipo "os vértices v e w estão conectados em G?". Concretamente, desejamos permitir as seguintes operações:

- GrafoDinâmico(n): devolve um grafo G com n vértices e nenhuma aresta.
- Conectado(G, v, w): devolve Sim se v, w estão conectados em G e Não caso contrário.
- Adicione(G, vw): adiciona a aresta vw em G. Supõe que vw não está em G.
- Remova(G, vw): remove a aresta vw de G. Supõe que vw está em G.

Se $G :=$ GrafoDinâmico(4) e seus vértices são rotulados como a, b, c e d, as operações Adicione(G, ab) e Adicione(G, cd) resultam no grafo da figura 3.1. Nesse momento, Conectado(G, a, d) devolve Não, enquanto Conectado(G, a, b) devolve Sim.

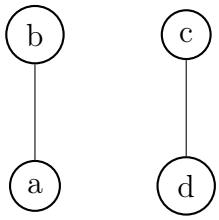

Figura 3.1: *Exemplo de grafo com 4 vértices.*

Já as operações Adicione(G, ac) e Adicione(G, bc) resultam no grafo da figura 3.2 e a operação Conectado(G, a, d) agora devolve Sim.

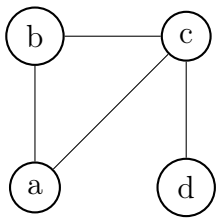

Figura 3.2: *Adição das arestas ac e bc.*

Por outro lado, REMOVA(G, cd) faz com que CONECTADO(G, a, d) passe a devolver NÃO, como ilustrado na figura 3.3.

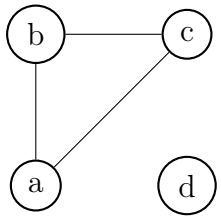

Figura 3.3: *Remoção da aresta cd.*

3.1 Estrutura de HDT

A partir deste ponto, denotamos por n o parâmetro de GRAFODINÂMICO, isto é, o número de vértices do grafo G. Uma solução do problema da conectividade dinâmica sobre um grafo G mantém várias **florestas geradoras** (Holm *et al.*, 2001). Intuitivamente, desejamos uma floresta geradora F de G para fazer consultas, já que os vértices estão conectados em G se e somente se estão conectados em F. Como toda componente de F é uma árvore, podemos representá-las de forma a realizar consultas em tempo $O(\lg n)$.

Nesse contexto, a dificuldade aparece quando adicionamos ou removemos arestas, pois a floresta geradora F pode mudar. Alguns casos são mais simples pois não acarretam tal mudança: adicionar uma aresta entre dois vértices que já estão conectados em F ou remover uma aresta que não pertence a F.

Já quando adicionamos uma aresta entre dois vértices que não estão conectados em F, basta adicionar a aresta à F para preservar a condição de floresta geradora. Finalmente, quando removemos uma aresta que está em F, além de remover a aresta da floresta, devemos nos atentar a possibilidade de F ter deixado de ser geradora. Nesse caso, devemos procurar por uma aresta **substituta** em G que reconecta as recém-criadas componentes de F, que pode não existir.

3.2 Níveis, subgrafos e invariantes

Vamos definir alguns conceitos importantes sobre um grafo G. O **nível** de uma aresta vw é número inteiro entre 0 e $\lg n$, inclusive. Se $\lg n$ não é inteiro, tomamos $\lceil \lg n \rceil$, porém escrevemos $\lg n$ para não sobrecarregar o texto. O nível de uma aresta recém-inserida é $\lg n$ e pode eventualmente diminuir, mas nunca aumentar.

Definimos G_i como o **subgrafo induzido** pelas arestas de nível menor ou igual a i. Dessa forma, $G_{\lg n} = G$. Sobre esses grafos, mantemos a seguinte relação invariante:

Invariante 1. *Todo componente de G_i tem no máximo 2^i vértices.*

Veja que a relação invariante vale logo após o grafo ter sido criado por GRAFODINÂMICO(n), pois toda componente de G_i tem exatamente 1 vértice, para $0 \leq i \leq \lg n$. Outra observação é que todo componente de G_0 tem no máximo $2^0 = 1$ vértice, portanto nenhuma aresta atinge o nível 0.

Definimos também F_i como uma floresta geradora de G_i. Logo, $F_{\lg n}$ é uma floresta geradora de G, a qual usaremos para consultas. Sobre essas florestas, mantemos ainda a seguinte relação invariante:

Invariante 2. *Para $i = 0, 1, \ldots, \lg n$ temos que $F_i = F_{\lg n} \cap G_i$. De maneira equivalente, $F_0 \subseteq F_1 \subseteq \cdots \subseteq F_{\lg n}$.*

Neste capítulo em específico, convencionamos o tamanho de um grafo G como o número de vértices de G e o denotamos por $|G|$.

3.3 Representação de grafos e florestas

Representamos grafos por **listas de adjacência**. Mantemos também as $\lg n$ florestas, denotadas por $F_1, \ldots, F_{\lg n}$. Arestas que pertencem a $F_{\lg n}$ são chamadas de **arestas de árvore**. Todas as outras arestas são chamadas de **arestas reservas**. Neste ponto, nos limitamos a supor uma representação das florestas que permite que a inserção e remoção de arestas e consultas de conectividade sejam realizadas com consumo de tempo $O(\lg n)$. Detalhes são dados no capítulo 4.

3.4 Operação GrafoDinâmico

A operação GrafoDinâmico apenas inicializa as listas de adjacência e florestas do grafo. Como não há nenhuma aresta em G, o invariante 1 é respeitado. Similarmente, como não há nenhuma aresta em F_i para $0 \leq i \leq \lg n$, o invariante 2 também é respeitado.

Tanto as florestas quanto listas podem ser inicializadas em tempo $O(1)$, portanto a operação consome tempo $O(n + \lg n)$ para inicializar as n listas de adjacência e as $\lg n$ florestas.

3.5 Operação Adicione

A operação Adicione(G, vw) apenas insere a aresta vw nas listas de adjacência de v e w e define seu nível como $\lg n$. Se v e w não estão conectados em $F_{\lg n}$, vw deve ser adicionada em $F_{\lg n}$.

Adicione(G, vw)

```
1  adicione vw nas listas de adjacência de v e w
2  vw.nivel = lg n
3  if v e w não estão conectados em F_{lg n}
4        adicione vw em F_{lg n}
```

Note que apenas modificamos $G_{\lg n}$ e $F_{\lg n}$, portanto ambos invariantes são respeitados. Também é importante notar que $F_{\lg n}$ continua sendo uma floresta geradora de G.

As operaçõpe de inserção em listas de adjacência podem ser todas realizadas em tempo constante se implementadas com listas ligadas. As operações nas linhas 3 e 4 consomem tempo $O(\lg n)$, como detalhado no capítulo 4. Logo, o consumo de tempo da operação Adicioneé $O(\lg n)$.

3.6 Operação Conectado

A operação Conectado é a mais simples. Basta verificar se os vértices estão conectados em $F_{\lg n}$.

CONECTADO(G, v, w)

1 **return** se v e w estão conectados em $F_{\lg n}$

O consumo de tempo é $O(\lg n)$, o mesmo da operação em florestas descrita no capítulo 4.

3.7 Operação REMOVA

A remoção de arestas é a operação mais complicada, pois ao remover uma aresta vw, devemos procurar uma aresta **substituta** que mantém $F_{\lg n}$ na condição de floresta geradora (possivelmente inexistente). Felizmente, $F_{\lg n}$ só perde essa condição se contém vw. Logo, se $F_{\lg n}$ não contém vw, basta removê-la das listas de adjacência de v e w.

Já no caso que $F_{\lg n}$ contém vw, além de removê-la das listas de adjacência, devemos também removê-la de todas as florestas que a contém, isto é, de $F_l, \ldots, F_{\lg n}$, onde l é seu nível. Na procura por uma aresta substituta, o invariante 2 nos garante que só precisamos considerar as arestas reservas com nível maior ou igual a l. De fato, imediatamente antes da remoção de vw, os vértices v e w estão conectados em F_l e desconectados em todas as florestas de nível menor que l. Após a remoção de vw, uma aresta com nível menor que l que reconecta suas componentes em F_l não pode existir pois todas as florestas de nível menor já são geradoras. Com isso em vista, vamos considerar os níveis em ordem crescente de l até $\lg n$.

Para um dado nível $i \geq l$, sejam T_v e T_w as árvores de F_i que contém v e w logo após a remoção de vw, respectivamente. É claro que $T_v \neq T_w$. Suponha sem perda de generalidade que $|T_v| \leq |T_w|$. Rebaixamos o nível de todas as arestas de T_v de nível i. Note que rebaixá-las modifica o tamanho das componentes de G_{i-1}. Entretanto, logo antes da remoção de vw, T_v e T_w faziam parte de uma única componente em G_i que respeitava o invariante 1, isto é, $|T_v| + |T_w| \leq 2^i$. Como $|T_v| \leq |T_w|$, segue que $|T_v| \leq 2^{i-1}$, ou seja, podemos rebaixar arestas de T_v sem violar o invariante 1.

Devemos agora considerar as arestas reservas. Se existe uma aresta reserva de nível i que reconecta T_v e T_w, com certeza uma de suas pontas está em T_v. Dessa forma, consideramos todas as arestas reservas de nível i incidentes a T_v. Para uma aresta xy nessas condições, há apenas duas possibilidades: xy reconecta T_v e T_w ou xy tem as duas pontas em T_v. No primeiro caso, encontramos uma aresta substituta. Basta adicionar xy em $F_i, \ldots, F_{\lg n}$ e parar a busca. No segundo caso, rebaixamos o nível de xy para $i-1$. Esse rebaixamento não altera o tamanho de nenhuma componente de G_{i-1} pois já rebaixamos todas as arestas de árvore de nível i.

REMOVA(G, vw)

```
1  remova vw das listas de adjacência de v e w
2  if vw ∈ F_{lg n}
3      for i = vw.nivel to lg n
4          remova vw de F_i
5          sejam T_v e T_w árvores de F_i que contém v e w, respectivamente
6          if |T_v| > |T_w|
7              T_v ↔ T_w
8          for xy ∈ T_v, xy.nivel = i
9              xy.nivel = i − 1
10             adicione xy em F_{i−1}
11         for xy ∈ G, xy.nivel = i, x ∈ T_v
12             if y ∈ T_w
13                 adicione xy em F_i ..., F_{lg n}
14                 stop
15             else
16                 xy.nivel = i − 1
```

A remoção das arestas das listas de adjacência consome tempo constante se implementada com listas ligadas. O laço da linha 3 executa no máximo $O(\lg n)$ vezes. Todas as operações de floresta consomem tempo $O(\lg n)$. Também é possível obter todas as arestas do laço da linha 8 em tempo $O(k \lg n)$, onde k é o número de arestas que satisfazem a condição. O mesmo pode ser dito da linha 11. Essas operações são detalhadas no capítulo 4.

Dessa forma, o consumo de tempo total da operação é $O(\lg^2 n + \lg \#rebaixamentos)$. É difícil analisar o número de rebaixamentos de uma execução específica de REMOVA, porém podemos olhar o que acontece ao longo de m operações ADICIONE e REMOVA. Cada aresta pode ser rebaixada no máximo $\lg n$ vezes. Por outro lado, uma aresta só pode ser rebaixada se tiver sido inserida pela operação ADICIONE previamente. Portanto, $\#rebaixamentos \leq \lg n \#insercoes$. Assim, podemos amortizar os rebaixamentos da operação REMOVA na operação ADICIONE, concluindo que ambas consomem tempo $O(\lg^2 n)$ amortizado.

Capítulo 4

Florestas dinâmicas

No problema da floresta dinâmica estamos interessados em manter uma coleção de árvores disjuntas que são modificadas ao longo do tempo pela adição e remoção de arestas. De forma objetiva, queremos realizar as seguintes operações:

- FlorestaDinâmica(n): devolve uma floresta F com n árvores disjuntas, todas com apenas um vértice.
- Conectado(F, v, w): devolve Sim se v e w estão conectados em F e Não caso contrário.
- Ligue(F, v, w): combina as árvores de F contendo v e w pela adição da aresta vw. Supõe que v e w estão em árvores diferentes.
- Corte(F, vw): remove a aresta vw de F. Supõe que a aresta vw existe.

Ao longo deste capítulo, denotamos por n o parâmetro da operação FlorestaDinâmica, isto é, o número de vértices da floresta. Se representarmos a floresta numa matriz de adjacência, é possível resolver o problema de modo que as operações Ligue e Corte consomem tempo $O(1)$, mas a operação Conectado consome tempo $O(n)$ (por meio de uma busca em largura).

Representando a floresta de outra forma, reduziremos o consumo de tempo da operação Conectado para $O(\lg n)$, pagando o preço de aumentar o consumo de tempo das operações Ligue e Corte para $O(\lg n)$.

4.1 Operações em sequências

Antes de introduzir a forma que representaremos uma floresta, apresentamos alguns conceitos importantes sobre sequências. Dada uma sequência S, escrevemos $S[i]$ para denotar seu i-ésimo elemento, onde $1 \leq i \leq |S|$. Também denotamos por $S[i..j]$ sua subsequência contínua de elementos no intervalo $[i, j]$.

Por fim, definimos duas operações sobre sequências:

- Fatie(S, k): Fatia S no k-ésimo elemento, isto é, devolve $S[1..k]$ e $S[(k+1)..|S|]$.
- Concatene(S_1, S_2): Concatena S_1 e S_2 numa mesma sequência, isto é, devolve S tal que $S[1..|S_1|] = S_1$ e $S[(|S_1|+1)..|S|] = S_2$.

4.2 Sequências eurelianas

Definimos a **sequência eureliana** (Henzinger e King, 1995) de uma árvore T enraizada num vértice r pela saída de $\mathrm{ET}(T, r)$, onde ET é definida como:

```
ET(T, v)
 1  acrescente vv
 2  for w vizinho de v
 3        acrescente vw
 4        if w não foi visitado
 5              ET(T, w)
```

Ao aplicar o procedimento no vértice $r =$ `A` da figura 4.1, obtemos a sequência eureliana `AA AB BB BE EE EB BA AC CC CF FF FC CG GG GC CA AD DD DA`. Dizemos que o vértice r escolhido é a raiz da árvore. Note que `rr` sempre é o primeiro elemento da sequência.

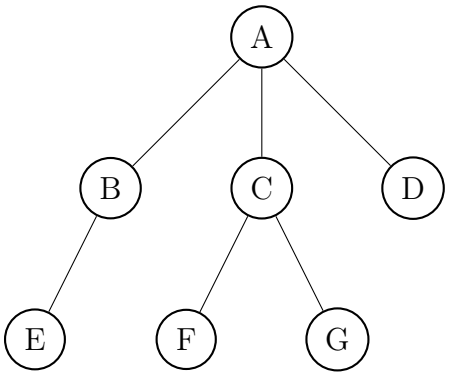

Figura 4.1: *Árvore enraizada no vértice A*

Uma sequência eureliana de uma árvore é na verdade uma trilha eureliana do grafo gerado pela duplicação de suas arestas e inclusão de laços em todos seus vértices (veja a figura 4.2).

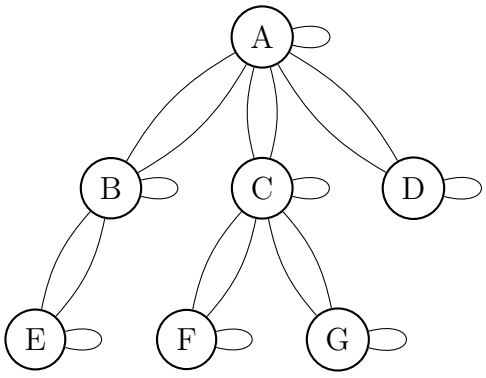

Figura 4.2: *Grafo induzido pela duplicação e adição de laços na árvore da figura 4.1*

Uma árvore tem mais de uma sequência eureliana, pois podemos escolher qualquer um de seus vértices como raiz. Entretanto, uma sequência eureliana representa exatamente uma árvore, já que contém precisamente todos seus vértices e arestas. Essa representação é enxuta e nos permite realizar as operações Ligue, Corte e Conectado de forma eficiente, como veremos adiante.

Proposição 4.3. *Uma sequência eureliana de uma árvore com n vértices tem tamanho $3n - 2$.*

Demonstração. Da definição, temos que cada aresta aparece duas vezes na sequência e cada vértice aparece uma vez na forma de um laço, portanto temos $2(n-1) + n = 3n - 2$ elementos. □

Vamos analisar o que acontece com a sequência eureliana da árvore da figura 4.1 quando removemos a aresta AC. A figura 4.4 ilustra o resultado da operação.

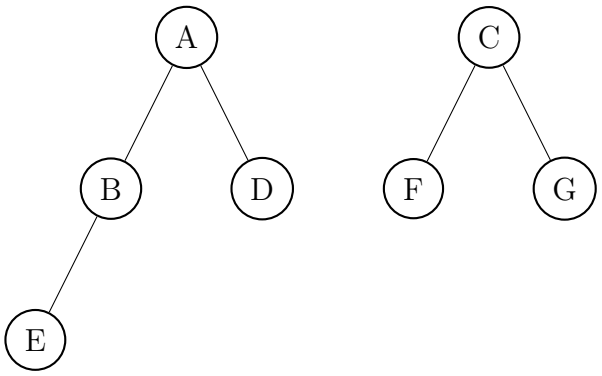

Figura 4.4: *Resultado da remoção da aresta AC.*

Veja o que acontece com sua sequência após a remoção das ocorrências de `AC` e `CA`:

`AA AB BB BE EE EB BA AC CC CF FF FC CG GG GC CA AD DD DA`
`AA AB BB BE EE EB BA  CC CF FF FC CG GG GC  AD DD DA`

Se tomarmos os vértices A e C como raízes de suas novas árvores, obtemos as sequências `AA AB BB BE EE EB BA AD DD DA` e `CC CF FF FC CG GG GC`, respectivamente. Nesse caso, vemos que a sequência da árvore que contém C é uma subsequência contínua da sequência original. Similarmente, a sequência da árvore que contém A é uma concatenação de duas subsequências contínuas.

Proposição 4.5. *Sejam T uma árvore e S uma sequência eureliana de T. Se vw é uma aresta de T, é possível remover vw e obter sequências eurelianas das novas componentes de v e w com quatro chamadas de* Fatie *e uma de* Concatene.

Demonstração. Seja vw uma aresta de T. Chame de T_v e T_w as novas componentes de v e w após a remoção de vw, respectivamente. Suponha sem perda de generalidade que `vw` ocorre antes de `wv` em S. Logo, S é da forma $S_1\,\texttt{vw}\,S_w\,\texttt{wv}\,S_2$, onde S_w é a sequência eureliana T_w enraizada em w. Com quatro chamadas de Fatie, extraímos S_w e removemos `vw` e `wv`. Uma chamada de Concatene(S_1, S_2) nos dá a sequência eureliana de T_v enraizada em v. □

Cabe agora analisar o que acontece quando adicionamos uma aresta. Por exemplo, vamos adicionar a aresta DG na floresta da figura 4.4. É conveniente fazer com que as pontas da nova aresta sejam raízes de suas respectivas árvores. Nesse caso, D e G passam a ser raízes, como visto na figura 4.6.

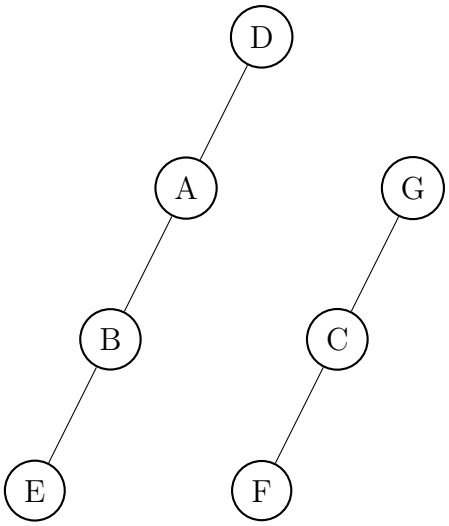

Figura 4.6: *Árvores re-enraizadas nos vértices D e G.*

Suas novas sequências são `DD DA AA AB BB BE EE EB BA AD` e `GG GC CC CF FF FC CG`, ambas rotações das sequências originais. Imediatamente após a adição da aresta DG, sua nova sequência é `DD DA AA AB BB BE EE EB BA AD DG GG GC CC CF FF FC CG GD`, simples concatenações das sequências e da nova aresta. O resultado é ilustrado na figura 4.7.

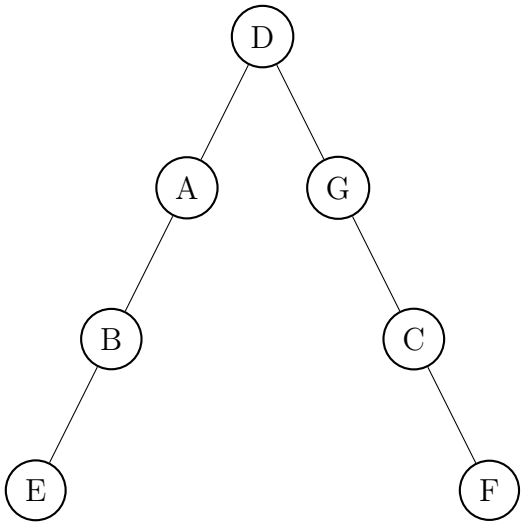

Figura 4.7: *Adição da aresta DG nas árvores re-enraizadas nos vértices D e G.*

Proposição 4.8. *Sejam T_v e T_w duas árvores disjuntas, v e w vértices de T_v e T_w, respectivamente. Sejam também S e R sequências eurelianas de T_v e T_w, respectivamente. É possível obter uma sequência eureliana de $(T_1 \cup T_2) + vw$ com cinco chamadas à* Concatene *e duas à* Fatie.

Demonstração. Primeiro note que $S = S_1\,\mathtt{vv}\,S_2$ e $R = R_1\,\mathtt{ww}\,R_2$. Definimos $S' := \mathtt{vv}\,S_2S_1$ e $R' := \mathtt{ww}\,R_2R_1$. Cada uma pode ser obtida por uma chamada à Fatie seguida de uma chamada à Concatene. Logo, $S'\,\mathtt{vw}\,R'\,\mathtt{wv}$ é uma sequência eureliana de $(T_1 \cup T_2) + vw$ enraizada em v, obtida por três chamadas à Concatene. □

As proposições apresentadas nos permitem definir as operações FlorestaDinâmica, Ligue, Corte e Conectado apenas em função de operações em sequências, isto é, de Fatie e Concatene. Entretanto, falta uma maneira de identificar a sequência que representa

a árvore que contém um dado vértice, assim como a posição desse vértice na sequência. Essas operações consomem tempo $O(\lg n)$ e serão descritas no capítulo 5.

4.3 Operação FlorestaDinâmica

A operação de FlorestaDinâmica apenas cria n árvores com um vértice cada, ou seja, n sequências com apenas um elemento. Consome tempo $O(n)$.

4.4 Operação Conectado

A operação Conectado é bastante simples. Para verificar se dois vértices v e w estão conectados em F, basta identificar suas sequências e verificar se são a mesma.

```
Conectado(F, v, w)
1  seja S a sequência de F que contém vv
2  seja R a sequência de F que contém ww
3  return se S = R
```

4.5 Operação Ligue

A operação Ligue(F, v, w) adiciona a aresta vw em F. Como visto na proposição 4.8, basta fazer alguns fatiamentos e concatenações nas sequências de F.

```
Ligue(F, v, w)
1  seja S a sequência de F que contém vv
2  seja R a sequência de F que contém ww
3  S' = TragaParaFrente(S, vv)
4  R' = TragaParaFrente(R, ww)
5  Concatene(S', uv, R', vu)
```

Note que na linha 5 estamos passando mais argumentos para a operação Concatene. Estamos apenas simplificando a notação. Na prática, devemos chamar a operação três vezes, de forma encadeada.

A operação TragaParaFrente é apenas um envólucro para evitar repetições. É definida da seguinte forma:

```
TragaParaFrente(S, x)
1  seja k a posição de x em S
2  S1, S2 = Fatie(S, k − 1)
3  Concatene(S2, S1)
```

Como fazemos um número constante de chamadas à Fatie e Concatene, o consumo de tempo é limitado pelo consumo de tempo dessas operações, isto é, por $O(\lg n)$.

4.6 Operação Corte

A operação Corte também faz uso direito de Fatie e Concatene, como visto na proposição 4.5.

Corte(F, vw)

```
1  seja S a sequência de F que contém vw
2  seja k a posição de vw em S
3  S_1, R = Fatie(S, k)
4  seja q a posição de wv em R
5  S_w, S_2 = Fatie(R, q − 1)
6  S_1, vw = Fatie(S_1, |S_1| − 1)
7  wv, S_2 = Fatie(S_2, 1)
8  Concatene(S_1, S_2)
9  apague vw
10 apague wv
```

A operação supõe que vw ocorre antes de wv na sequência. É fácil adaptá-la para funcionar quando isso não acontece. Novamente, o consumo de tempo está limitado pelo consumo de tempo de Fatie e Concatene, isto é, por $O(\lg n)$.

4.7 Operações adicionais

Se quisermos resolver o problema da conectividade dinâmica restrito para florestas, a solução apresentada é suficiente. Entretanto, a motivação para se resolver a versão restrita veio do problema da conectividade dinâmica em grafos genéricos e portanto requer algumas modificações.

No problema original para grafos, precisamos consultar arestas com propriedades específicas. Para tanto, vamos definir algumas operações adicionais que nos permitem rapidamente encontrar tais arestas.

- MarqueBranco(F, v): adiciona uma marca branca no vértice v da floresta F.
- DesmarqueBranco(F, v): remove uma marca branca no vértice v da floresta F.
- MarquePreto(F, v): adiciona uma marca preta no vértice v da floresta F.
- DesmarquePreto(F, v): remova uma marca preta no vértice v da floresta F.
- VérticesBrancos(F, v): devolve todos os vértices com marcas brancas da componente de v na floresta F.
- VérticesPretos(F, v): devolve todos os vértices com marcas pretas da componente de v na floresta F.

A ideia é que para uma aresta de um certo tipo, adicionamos marcas em suas pontas. Neste trabalho, temos apenas duas características relevantes, as quais abstraímos para **preto** e **branco**.

Como nossas florestas são representadas por sequências, essas operações dependem da representação das sequências. Apresentamos uma representação eficiente no capítulo 5, que permite que as operações consumam tempo $O(\lg n)$, com exceção de VérticesBrancos e VérticesPretos, que consomem tempo $O(k \lg n)$, onde k é o número de vértices brancos/pretos da componente em questão.

Capítulo 5

Representação de sequências

Como visto no capítulo 4, desejamos uma representação de sequências que suporte as operações de concatenação e fatiamento eficientemente. Se representarmos sequências por listas ligadas, é possível realizar as operações em $O(n)$, onde n é o tamanho da maior sequência. Entretanto, apresentamos uma maneira que utiliza árvores de busca binária balanceadas, reduzindo o tempo de ambas operações para $O(\log n)$.

5.1 Árvores de busca binária implícitas

Para todo nó x de uma árvore de busca binária, ou ABB, denotamos por $x.chave$ sua chave de comparação e por $x.valor$ seu valor. Representamos uma sequência por uma árvore de busca binária, tomando como valores seus próprios elementos e como chaves suas respectivas posições. Por exemplo, a sequência `BRASIL` terá os valores da tabela 5.1 e pode ser representada pela árvore de busca binária da figura 5.1.

chave	`123456`
valor	`BRASIL`

Tabela 5.1: *Chaves e valores da sequência* `BRASIL`.

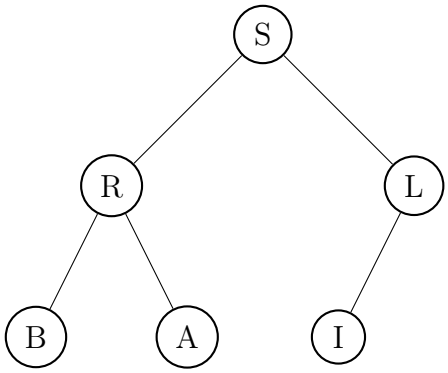

Figura 5.1: *Árvore de busca binária da sequência* `BRASIL`.

Na prática, os nós da árvore não guardam explicitamente suas chaves, como já ilustra a figura 5.1. Em vez disso, cada nó x guarda o tamanho de sua subárvore, denotado por $x.tamanho$. A figura 5.2 ilustra a mesma árvore explicitando esses tamanhos.

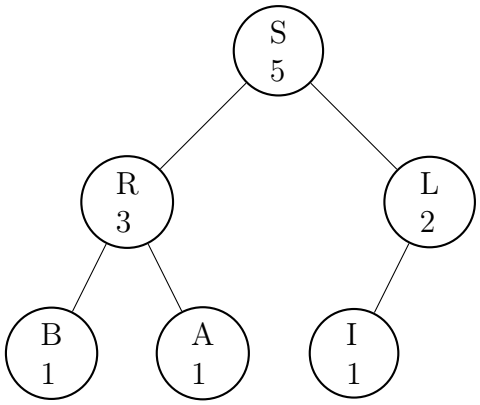

Figura 5.2: *Árvore da figura 5.1 com os tamanhos de cada subárvore.*

Tais árvores são chamadas de árvores de busca binária implícitas, pois não guardam suas chaves explicitamente. Sobre essas árvores, definimos as operações de **estatística de ordem** (Cormen *et al.*, 2009). São elas:

- BUSCA(T, k): devolve o k-ésimo nó da árvore T.
- ORDEM(x): devolve a posição (chave) do nó x.

Intuitivamente, a operação BUSCA(T, k) procura o k-ésimo nó com base nos tamanhos das subárvores. Começando da raiz, verificamos se o filho da esquerda tem pelo menos k elementos. Em caso positivo, o k-ésimo elemento deve estar nesse ramo e a busca prossegue recursivamente nessa direção. Caso contrário, o nó pode ser a própria raiz (quando há exatos $k-1$ elementos na subárvore do filho esquerdo) ou pode estar na subárvore do filho direito. No último caso, continuamos a busca recursivamente no ramo direito, carregando a informação de que já desconsideramos toda subárvore do filho esquerdo e a raiz.

Já a operação ORDEM(x) é calculada na outra direção, isto é, subindo na árvore até a raiz. Como o nó x tem chave maior do que todas as chaves em sua subárvore esquerda, sua chave (posição) é pelo menos o tamanho da subárvore esquerda mais um. Continuamos a busca em direção a raiz, carregando se viemos de um filho direito ou esquerdo. Se viemos de um filho direito, sabemos que nossa posição deve ser acrescida do tamanho da subárvore esquerda mais um.

As operações BUSCA e ORDEM podem ser implementadas em qualquer árvore de busca binária. O consumo de tempo de ambas é proporcional a altura da árvore. Dessa forma, ambas consomem tempo $O(\lg n)$ numa árvore rubro-negra.

Árvores rubros-negras suportam as operações JUNTE e DIVIDA (Tarjan, 1983), definidas da seguinte forma:

- JUNTE(T_1, T_2): Dado duas árvores disjuntas T_1 e T_2 onde $x.chave < y.chave$ para todo $x \in T_1$ e $y \in T_2$, devolve a união de T_1 e T_2.
- DIVIDA(T, k): Devolve duas árvores T_1 e T_2 tal que $x.chave \leq k$ para todo $x \in T_1$ e $y.chave > k$ para todo $y \in T_2$.

Ambas consomem tempo proporcional a altura da árvore, isto é, $O(\lg n)$. Se tratando de árvores de busca binária implícitas, essas operações são equivalentes às operações FATIE e CONCATENE.

5.2 Operações adicionais

Como mencionado na seção 4.7, desejamos colocar marcas brancas e pretas nos vértices das florestas. Os vértices são elementos de uma sequência, que por sua vez são nós de uma árvore de busca binária. Assim, podemos carregar mais informação nos nós da ABB: a quantidade de marcas brancas e pretas na subárvore de cada nó. Para adicionar uma marca num vértice da árvore, basta adicionar uma marca no nó correspondente da ABB e atualizar todos os nós no caminho desse nó até a raiz da ABB. Esse processo consome tempo proporcional à altura da ABB. No caso de uma árvore rubro-negra, consome tempo $O(\lg n)$.

Para obter todos os vértices de uma dada cor, usamos a informação extra da quantidade de marcas na subárvore. Começando pela raiz da ABB, só precisamos considerar subárvores com pelo menos uma marca da cor desejada. Dessa forma, atingimos todos os nós desejados. No caso de uma árvore rubro-negra, se temos k nós da cor desejada, o consumo de tempo total é de $O(k \lg n)$, pois cada nó é atingido em número proporcional a altura da ABB.

Capítulo 6

Implementação

6.1 Recapitulação

O problema da conectividade dinâmica usa uma representação complexa com várias camadas para um grafo. Um grafo de n vértices é representado por $\lg n$ florestas. Cada floresta é representada por várias sequências eurelianas, uma para uma de suas componentes (árvores). Cada sequência eureliana é representada por uma árvore de busca binária balanceada (ABBB). A figura 6.1 sintetiza a representação.

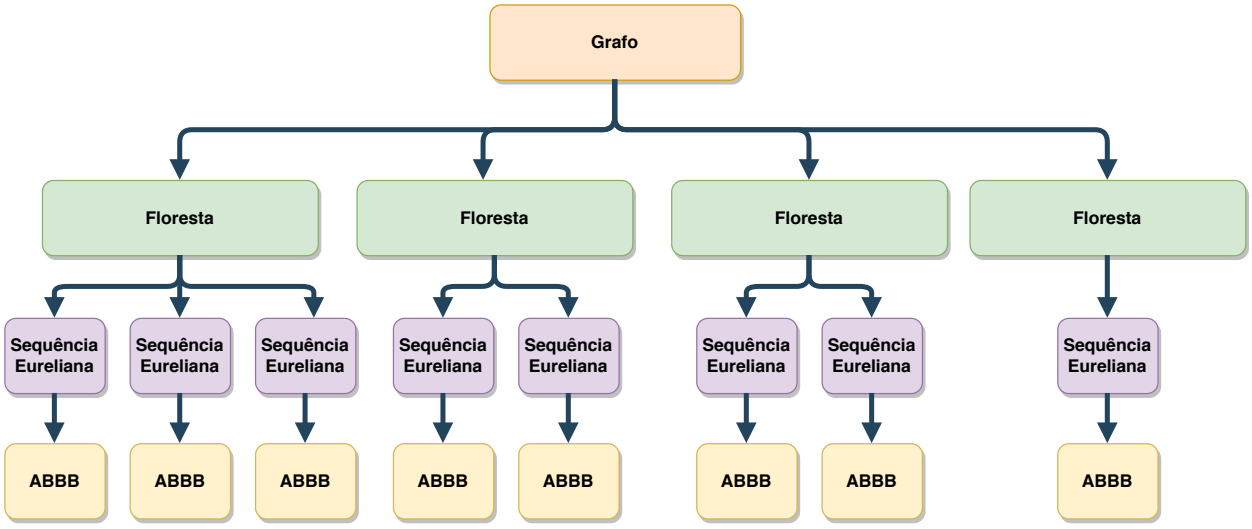

Figura 6.1: *Diagrama da representação do grafo no problema da conectividade dinâmica*

Apesar da complexidade, cada camada tem uma tarefa bem definida, o que facilita a implementação e uma eventual depuração.

6.2 Florestas

Vamos começar com os detalhes de implementação das florestas dinâmicas. Apesar das diversas camadas embaixo das florestas, na prática a implementação gira em torno da ABBB. Neste trabalho, escolhemos implementar uma treap (Aragon e Seidel, 1989), uma árvore de busca binária que usa um esquema de balanceamento aleatorizado. A vantagem é que as operações FATIE e CONCATENE de sequências são facilmente implementadas, pois a treap usa operações muito semelhantes como operações primitivas. Em teoria, o consumo de tempo

no pior caso é significativamente maior, mas na prática o desempenho é próximo ao de árvores rubros-negras.

Uma treap permite as seguintes operações:

- DIVIDA(T, x): Devolve duas treaps T_1 e T_2 de modo que todos os nós de T_1 tenham chave menor ou igual a x e todos os nós de T_2 tenham chave maior que x.
- JUNTE(T_1, T_2): Devolve a união das treaps T_1 e T_2. Supõe que todos os nós de T_1 tem chave menor que o todos os nós de T_2.

Como mencionado no capítulo 5, nossas árvores de busca binária são implícitas. Logo, as chaves das treaps são as posições dos elementos da sequência eureliana. Assim, as operações DIVIDA e JUNTE são equivalentes as operações FATIE e CONCATENE, respectivamente.

Outra peça importante na implementação de uma floresta dinâmica é uma tabela de espalhamento, que mapeia pares ordenados para nós de treap. As treaps representam sequências e queremos acessar os elementos da sequência (pares ordenados) de forma rápida. Dado um nó, podemos conseguir sua posição da sequência com auxílio da operação ORDEM. Essa tabela também é útil para verificar se uma aresta existe na floresta.

Quando criamos uma floresta com n vértices, devemos criar n treaps, todas com exatamente um nó, um para cada vértice da floresta. Feito isso, devemos mapear todos os elementos da forma vv para seus respectivos nós de treap.

Acima de tudo, treaps ainda são árvores de busca binária. Portanto podemos adicionar mais informações em seus nós, como o tamanho de sua subárvore e a quantidade de nós brancos e pretos em sua subárvore. Dessa forma, as operações VÉRTICESBRANCOS e VÉRTICESPRETOS podem se limitar a buscar apenas em subárvores que tem pelo menos um nó branco ou preto, respectivamente.

Para as consultas de conectividade, devemos verificar se dois vértices fazem parte da mesma sequência. Para tanto, basta comparar as raízes das treaps que contém os vértices da consulta. Para obter as raízes, também guardamos no nó da treap um ponteiro para seu pai.

6.3 Grafo

É na camada do grafo que devemos nos preocupar em manter as relações invariantes, colorir os vértices de forma correta e atualizar os níveis das arestas. Para tanto, mantemos um vetor de $\lg n$ florestas dinâmicas. De modo geral, essa camada orquestra diversas operações em florestas.

Também mantemos suas listas de adjacência. Para maximizar a eficiência, dividimos as listas de adjacência em dois tipos: para arestas de árvore e para arestas reservas. Dentro desses tipos, dividimos ainda em $\lg n$ listas, uma para cada nível de aresta. Dessa forma, podemos facilmente iterar pelas arestas de árvore ou reservas de um nível específico. Assim, temos duas matrizes de tamanho $n \times \lg n$, onde cada entrada é uma lista duplamente encadeada. Para permitir remoções das listas de adjacência em tempo constante amortizado, também guardamos uma tabela de espalhamento que mapeia arestas para suas ocorrências nas listas encadeadas.

Convencionamos que todo vértice que tem uma aresta de árvore adjacente a si é tem ao menos uma marca branca. Similarmente, todo vértice que tem uma aresta reserva adjacente a si tem ao menos uma marca preta. Isso é importante para saber quais arestas fazem parte de uma certa árvore ou quais arestas reservas são adjacentes a uma certa árvore. Como

temos as listas de adjacência separadas por nível e tipo, podemos considerar somente as arestas relevantes. Por exemplo, para iterar por todas as arestas de nível l de uma árvore T, podemos pegar todos os vértices brancos dessa árvore e iterar por suas listas de adjacência de árvore de nível l.

Outro ponto importante é que ao rebaixar uma aresta, muitas operações devem ser realizadas. Deve-se removê-la da lista encadeada corrente e colocá-la numa de nível mais baixo. No caso de ser uma aresta de árvore, também deve-se adicioná-la na floresta de nível inferior. As marcas dos vértices também devem ser ajustadas. As marcas das pontas da aresta devem ser removidas no nível atual e adicionadas no mesmo vértice, porém no nível menor.

6.4 C++

Uma implementação está disponível em https://github.com/gabrielrussoc/mac499/. Todo código foi escrito em C++ no padrão C++14. Não há nenhuma dependência externa, exceto a biblioteca *Google Test* [1] para testes. O código utiliza o sistema de compilação *Bazel*[2].

A biblioteca considera que num grafo de n vértices, esses são numerados de 0 até $n-1$. Um exemplo de uso é dado no código 6.1.

```
1 #include <cstdio>"
2 #include "DynamicGraph/DynamicGraph.hpp"
3
4 using usp::DynamicGraph;
5
6 int main(int argc, char **argv) {
7   DynamicGraph dg(3);
8   dg.insert(0, 1);
9   dg.insert(1, 2);
10  dg.insert(2, 0);
11  puts("%s" ? dg.is_connected(0, 1) : "Sim" ? "Nao"); // imprime Sim
12  dg.remove(0, 1);
13  puts("%s" ? dg.is_connected(0, 1) : "Sim" ? "Nao"); // imprime Sim
14 }
```

Código 6.1: *Exemplo de uso da biblioteca*

[1] https://github.com/abseil/googletest
[2] https://bazel.build/

Capítulo 7

Testes

O código da solução para o problema da conectividade dinâmica apresentado neste trabalho não é trivial. Erros são comuns e difíceis de serem detectados. Dessa forma, uma boa bateria de testes é essencial para verificar a implementação.

7.1 Problemas de programação competitiva

Uma maneira de testar o código é por meio de juízes virtuais de programação competitiva. O problema é recorrente no cenário competitivo e há vários desafios que o envolvem. Alguns exemplos são:

- https://www.spoj.com/problems/DYNACON1/
- https://www.spoj.com/problems/DYNACON2/
- https://codeforces.com/gym/100551/problem/A

Em geral, desafios de programação competitiva têm baterias de testes muito fortes. Resolvê-los é um grande indício de que a implementação está correta.

7.2 Teste de estresse

Outra maneira de testar a implementação é por meio de um **teste de estresse**. A ideia é pegar uma implementação de uma solução simples do problema e comparar os resultados das consultas de conectividade em ambas implementações de forma exaustiva.

Primeiro, geramos um grafo num formato específico, como por exemplo uma grade (veja a figura 7.1). O formato de grade é bom porque é muito sensível a atualizações, no sentido de que é facilmente desconectado e tem algumas arestas reservas. Esse grafo servirá de referência para atualizações aleatórias.

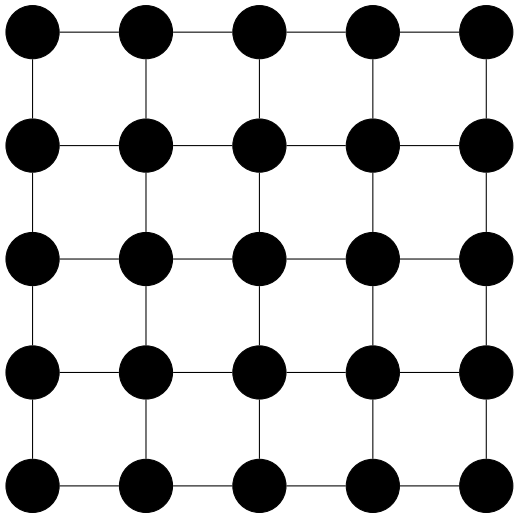

Figura 7.1: *Exemplo de grafo conhecido como grade.*

Além da grade, começamos com dois grafos vazios. Um deles é implementado de maneira direta, com matrizes de adjacência e fazendo buscas para consultas de conectividade. O outro é implementado como descrito no capítulo 6. Também escolhemos dois números k e m. O primeiro é a quantidade de atualizações aleatórias que faremos nos dois grafos. O segundo é o intervalo que faremos consultas de conectividade exaustivas, isto é, a cada m atualizações verificamos se o resultado de todas as consultas de conectividade entre dois vértices têm o mesmo resultado nas duas implementações. Como a consulta exaustiva é cara em termos de tempo, é ideal que m não seja muito pequeno.

Para conduzir as atualizações aleatórias, usaremos a grade como referência. Começamos com uma escolha aleatória de aresta da grade. Se a aresta não existe na implementação simples, adicionamos a aresta em ambas implementações. Caso contrário, removemos a aresta de ambas implementações. Isso garante que nunca removemos uma aresta que não existe ou que adicionamos uma aresta que já existe.

7.3 Desempenho

Para analisar o desempenho da solução de forma empírica, testes foram realizados com grades de diferentes tamanhos. Uma estratégia semelhante à do teste de estresse foi utilizada. Começando com um grade aleatória, arestas foram escolhidas de forma aleatória e uma implementação simples foi responsável por decidir se uma aresta deve ser adicionada ou removida na implementação eficiente. Entretanto, apenas os tempos da implementação eficiente foram medidos.

Como visto nas seções 3.5 e 3.6, as operações ADICIONE e CONECTADO tem um consumo de tempo muito baixo, pois acabam se reduzindo à operações em árvores de busca binária balanceadas. Lembre-se que para um grafo de n vértices, seu consumo de tempo é $O(\lg n)$. A figura 7.2 mostra o desempenho médio, em microssegundos, das operações. Foram realizadas cerca de 5000 operações de cada tipo para cada tamanho de grafo.

Figura 7.2: *Desempenho das operações* CONECTADO *e* ADICIONE.

A figura 7.2 sugere que apesar do consumo de tempo das operações crescerem num passo semelhante, a operação ADICIONE consome mais tempo que a operação CONECTADO. Essa diferença era esperada, afinal a operação ADICIONE também é responsável pela manutenção das listas de adjacência, assim como a coloração dos vértices.

Por outro lado, como visto na seção 3.7, a operação REMOVA tem um consumo de tempo maior, que depende do número de rebaixamentos realizados. A figura 7.3 mostra o desempenho médio, em microssegundos, da operação. Foram realizadas cerca de 5000 operações para cada tamanho de grafo.

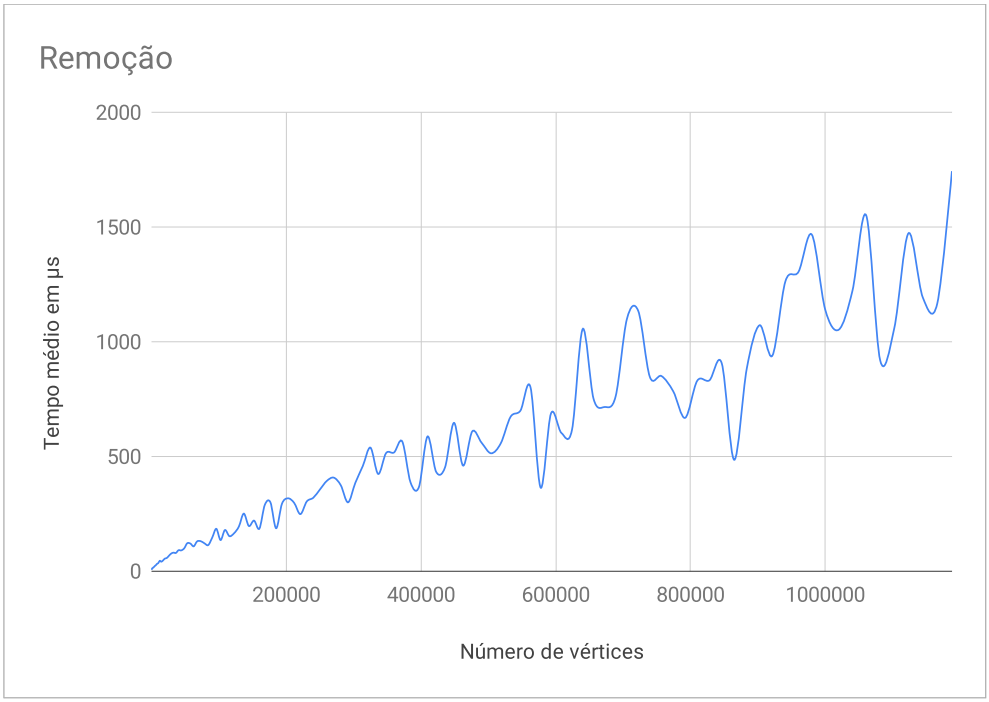

Figura 7.3: *Desempenho da operação* REMOVA.

A figura 7.3 sugere um consumo de tempo maior da operação REMOVA quando comparada com as operações ADICIONE e CONECTADO. Outro ponto que pode ser notado é uma maior variabilidade no consumo de tempo para os diferentes tamanhos de grafo. Os resultados não são surpreendentes, uma vez que a operação é muito dependente da estrutura do grafo e do tipo de aresta que foi removida.

Capítulo 8

Conclusões

Terminamos aqui a análise do problema da conectividade dinâmica. Detalhamos as estruturas de dados necessárias para a implementação de sua solução numa linguagem de programação moderna.

Diferente de boa parte do material sobre o assunto, a solução do problema foi completamente implementada e testada exaustivamente. Testamos seu consumo de tempo na prática e pudemos comprovar seu desempenho de forma empírica.

A solução se mostrou bastante trabalhosa e requer um bom conhecimento de estruturas de dados e análise de algoritmos para ser completamente compreendida. Dividida em vários níveis de abstração, é fácil se perder em suas várias camadas. Apesar do lado prático, a teoria não foi deixada de lado e os algoritmos apresentados foram detalhados e analisados.

Esta dissertação tratou apenas um dos diversos problemas dinâmicos estudados na acadêmia. Há outros problemas interessantes, como o problema da árvore geradora mínima dinâmica e o problema da bi-conectividade dinâmica.

Referências Bibliográficas

Aragon e Seidel(1989) C. R. Aragon e R. G. Seidel. Randomized search trees. Em *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, SFCS '89, páginas 540–545, Washington, DC, USA. IEEE Computer Society. ISBN 0-8186-1982-1. doi: 10.1109/SFCS.1989.63531. URL https://doi.org/10.1109/SFCS.1989.63531. Citado na pág. 21

Cormen *et al.***(2009)** Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edição. ISBN 0262033844, 9780262033848. Citado na pág. 18

Frederickson(1983) Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees. Em *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, páginas 252–257, New York, NY, USA. ACM. ISBN 0-89791-099-0. doi: 10.1145/800061.808754. URL http://doi.acm.org/10.1145/800061.808754. Citado na pág. 1

Henzinger e King(1995) Monika Rauch Henzinger e Valerie King. Randomized dynamic graph algorithms with polylogarithmic time per operation. Em *Proceedings of the Twenty-seventh Annual ACM Symposium on Theory of Computing*, STOC '95, páginas 519–527, New York, NY, USA. ACM. ISBN 0-89791-718-9. doi: 10.1145/225058.225269. URL http://doi.acm.org/10.1145/225058.225269. Citado na pág. 12

Holm *et al.***(2001)** Jacob Holm, Kristian de Lichtenberg e Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760. ISSN 0004-5411. doi: 10.1145/502090.502095. URL http://doi.acm.org/10.1145/502090.502095. Citado na pág. 1, 6

Huang *et al.***(2016)** Shang-En Huang, Dawei Huang, Tsvi Kopelowitz e Seth Pettie. Fully dynamic connectivity in o(log n(log log n)^2) amortized expected time. *CoRR*, abs/1609.05867. URL http://arxiv.org/abs/1609.05867. Citado na pág. 1

Tarjan(1983) Robert Endre Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA. ISBN 0-89871-187-8. Citado na pág. 18