Conexidade Dinâmica

Relatório de Atividades da Iniciação Científica

Orientadora: Cristina Gomes Fernandes
Aluno: Daniel Angelo Esteves Lawand

Este relatório refere-se à bolsa de Iniciação Científica
para o aluno Daniel Angelo Esteves Lawand,
e cobre o período de setembro de 2021 a agosto de 2022.

1 Introdução

O objetivo desta iniciação científica é o estudo e implementação de estruturas de dados e algoritmos para problemas de conexidade em grafos num contexto onde o grafo pode sofrer modificações [5].

O problema central é o de manter informações sobre as componentes conexas do grafo, ou mais exatamente, o problema de responder eficientemente consultas do tipo:

- Quantas componentes têm o grafo?
- Os vértices u e v estão numa mesma componente do grafo?

Num contexto estático, em que o grafo não sofre alterações, há algoritmos lineares que determinam as componentes conexas do grafo e respondem de maneira ótima consultas como estas.

No entanto, se o grafo em questão está sendo alterado, podendo ganhar ou perder arestas, o problema torna-se mais desafiador. Os tipos de modificações permitidas determinam a dificuldade do problema. Podemos considerar as três seguintes possibilidades:

- **conexidade incremental:** arestas podem ser acrescentadas ao grafo;
- **conexidade decremental:** arestas podem ser removidas do grafo;
- **conexidade totalmente dinâmica:** arestas podem ser acrescidas ou removidas do grafo.

O problema de conexidade incremental é resolvido de maneira bastante eficiente por meio de uma estrutura de dados conhecida como *union-find* [12], que provê resposta às consultas em tempo amortizado aproximadamente constante.

O problema de conexidade decremental foi resolvido por Even e Shiloach [6]. O método usa uma tabela onde se mantém o identificador da componente de cada vértice do grafo. A grande questão é como atualizar essa tabela quando da remoção de uma aresta.

Se o grafo em questão é uma floresta, ou seja, não tem circuitos, o problema é mais simples. A ideia é que a remoção de uma aresta sempre quebrará uma componente em duas, e atualizaremos os identificadores do menor dos dois pedaços resultantes. Ao remover por exemplo a aresta uv, devemos determinar se o pedaço em que u ficou é menor ou maior que o pedaço em que v ficou. Isso pode ser feito por exemplo usando-se duas buscas, uma a partir de u e uma a partir de v, que são executadas paralelamente (ou de maneira intercalada) e, ao chegarmos ao final de uma delas, abortamos a segunda, pois determinamos já qual é o menor dos pedaços. O tempo gasto desta maneira será proporcional ao tamanho do menor dos pedaços, implicando que o tempo para a atualização, amortizado pelo número de consultas, é $O(\lg n)$, onde n é o número de vértices do grafo.

Para grafos arbitrários, é necessário determinar se a aresta uv removida quebra ou não uma componente em duas. A ideia de novo é executar dois processos em paralelo (ou intercaladamente): um para determinar se u e v estão ainda na mesma componente após a remoção de uv, e o outro para decidir se uma componente foi quebrada em duas. Esse segundo processo é semelhante ao que foi usado no caso em que o grafo é uma floresta. Já o procedimento envolvido no primeiro processo é um pouco mais sofisticado, e utiliza uma estrutura de dados construída a partir de uma BFS do grafo original, que vai sendo atualizada a medida que o grafo vai sofrendo remoções de arestas.

Para conexidade totalmente dinâmica em florestas com n vértices, podemos usar uma coleção das chamadas *link-cut trees* [10] ou das chamadas *Euler tour trees* [7] para representar a floresta. Estas estruturas implementam uma rotina chamada $\textsc{FindRoot}(x)$, semelhante à rotina $\textsc{FindSet}(x)$ do union-find, que devolve um representante da componente em que o vértice x se encontra. Com isso, podemos responder às consultas facilmente, pois dois vértices x e y estão na mesma componente da floresta se e somente se $\textsc{FindRoot}(x) = \textsc{FindRoot}(y)$. O tempo amortizado de atualização e consulta é $\mathrm{O}(\lg n)$.

Para o caso geral da conexidade totalmente dinâmica, podemos representar um grafo arbitrário por uma floresta geradora maximal do grafo. Se chamarmos uma tal floresta de F, podemos usar por exemplo *link-cut trees* para armazená-la. Com isso, inserções e consultas podem ser implementadas diretamente usando as operações correspondentes destas árvores. Por outro lado, as remoções são mais desafiadoras. Em especial, se a aresta removida fizer parte da floresta F, isso pode requerer encontrar, de forma eficiente, uma outra aresta do grafo para ser adicionada a F. Esta operação é implementada por meio de uma estrutura de dados mais complexa [4].

No período de agosto de 2021 a março de 2022, completamos a implementação das *link-cut trees* e a solução do problema de conexidade totalmente dinâmica para florestas. Essa implementação envolve, entre outras coisas, a implementação de uma versão particular das chamadas *splay trees*, que são árvores de busca binária de busca que se auto-balanceiam.

No período de abril de 2022 a agosto de 2022, estudamos o problema mais geral da conexidade totalmente dinâmica em grafos arbitrários. Uma das soluções da literatura para este problema utiliza a biblioteca de conexidade para florestas [8]. Trata-se de um algoritmo sofisticado, que mantém várias florestas geradoras do grafo corrente, cada uma representa diferentes níveis do grafo, e estas florestas são mantidas por meio de *link-cut trees* como no problema da conexidade dinâmica em florestas.

No que segue, descrevemos brevemente as *splay trees* implementadas, depois descrevemos a implementação das *link-cut trees* utilizando as *splay trees* e finalizamos descrevendo a implementação da biblioteca do problema da conexidade totalmente dinâmica para florestas, usando as *link-cut trees*.

2 Splay trees

Splay tree é uma árvore binária de busca (ABB), em que o último elemento acessado se torna a raiz da árvore [11]. Isso é feito de uma maneira particular que garante que as operações na árvore tenham custo amortizado logarítmico no número de nós da árvore [9]. Existem diferentes tipos de implementação das *splay trees*.

Para o propósito desse trabalho, a chave dos elementos armazenados na *splay tree* não será armazenada explicitamente. Ou seja, os nós (`Node`), na nossa implementação, não possuem campo chave (`Key`), tratando-se de uma ABB com chaves implícitas, e não utilizamos as chaves para realizar qualquer tipo de operação. Em uma ABB tradicional, as chaves da subárvore esquerda são menores do que a do nó e as chaves da subárvore direita são maiores do que a do nó, e para encontrar determinada chave, a passamos como argumento. Em uma *splay tree* com chaves implícitas, considera-se que a chave de um nó é a posição dele num percurso in-ordem da árvore. Assim, por exemplo ao criar uma *splay tree* com um único elemento, não há necessidade de passar uma chave pois essa será automaticamente 1. Assim utilizamos a seguinte interface para as *splay trees*.

- `makeSplay(l)`: recebe um inteiro l que determina o nível do nó no grafo e retorna a raiz de uma *splay tree* com apenas um nó, com consumo de tempo de $\mathrm{O}(1)$.
- `splay(Node v)`: recebe um nó `v` de uma *splay tree*, tornando-o raiz desta *splay tree*, com consumo amortizado de tempo de $\mathrm{O}(\lg n)$.
- `join(Node v, Node w)`: junta as *splay trees* com raiz em `v` e `w`. Assume-se que `v` é o nó com chave máxima em sua *splay tree* e as chaves da *splay tree* de `w` se tornarão maiores que a chave de `v`. Tal rotina consome tempo $\mathrm{O}(1)$.
- `split(Node v)`: recebe um nó `v` de uma *splay tree* e quebra essa *splay tree* em duas: uma com todos os nós com chave menor ou igual a `v` e outra com os nós com chave maior que `v`. Tal rotina consome tempo $\mathrm{O}(1)$.
- `maxSplay(Node v)`: retorna o nó com chave máxima da *splay tree* a que `v` pertence, com consumo amortizado de tempo de $\mathrm{O}(\lg n)$.
- `minSplay(Node v)`: retorna o nó com chave mínima da *splay tree* a que `v` pertence, com consumo amortizado de tempo de $\mathrm{O}(\lg n)$.
- `reflectTree(Node v)`: inverte o percurso em in-ordem da *splay tree* enraizada em `v`. O efeito dessa operação numa ABB com chaves implícitas é a alteração de todas as chaves dos nós desta árvore de $0, 1, \cdots, t-1, t$ para $t, t-1 \cdots, 1, 0$. Essa rotina consome tempo $\mathrm{O}(1)$.

Nessa implementação, não há inserção ou remoção de nós, pois as *splay trees* crescem com a rotina `join` e diminuem através da rotina `split`, que quebra a árvore em duas. Também não faremos buscas nessas árvores. Note também que as operações `join`, `split` e `reflectTree` alteram implicitamente a chave de vários nós das árvores manipuladas. O uso de chaves implícitas é essencial para que o consumo de tempo dessas operações seja constante.

3 Link-cut trees

As *link-cut trees* são uma estrutura de dados usada para dar suporte a operações sobre florestas enraizadas [3].

Em nossa aplicação, queremos utilizá-las para manipular florestas simples (não enraizadas). Para isso, teremos que acrescentar à biblioteca básica das *link-cut trees* uma operação extra.

Começaremos descrevendo a biblioteca básica, voltada a florestas enraizadas [2]. A interface básica neste caso contém as seguintes rotinas:

- `maketree(l)`: recebe um inteiro l e cria uma árvore enraizada com um único nó de nível l e o devolve.
- `link(Node u, Node v)`: recebe um nó `u` que é raiz de uma árvore enraizada e um nó `v` de uma outra árvore enraizada e adiciona o arco de `u` para `v` juntando as duas árvores enraizadas.
- `cut(Node v)`: recebe um nó `v` de uma árvore enraizada em que `v` não é a raiz e remove o arco de `v` para seu pai na árvore, resultando em duas árvores enraizadas: uma com `v` e seus descendentes e outra com os demais nós da árvore enraizada original de `v`.
- `findroot(Node u)`: recebe um nó `u` de uma árvore enraizada e devolve a raiz da árvore.

Para manipular florestas simples, não enraizadas, adicionamos à interface uma rotina que altera a raiz de uma árvore enraizada para um outro nó arbitrário da árvore:

- `evert(Node v)`: recebe um nó `v` de uma árvore enraizada e modifica a árvore tornado `v` a raiz da árvore. Mais precisamente, inverte a orientação dos arcos no caminho de `v` até a raiz da árvore em que `v` se encontra.

Essa biblioteca de funções pode ser implementada por meio de *splay trees*, de modo que o consumo de tempo do `maketree` seja $O(1)$, e o consumo de tempo do `link`, `cut`, `findroot` e `evert` seja $O(\lg n)$ amortizado por operação, onde n é o número de chamadas a `maketree`.

Para implementar as rotinas das *link-cut trees* eficientemente, mantemos para cada nó da floresta o chamado filho preferido do momento: cada nó rastreia o filho que foi acessado por último nas operações executadas na floresta. O nó mais profundo acessado fica sem filho preferencial. Essa informação particiona cada árvore da floresta nos chamados caminhos preferenciais. Veja o exemplo da Figura 1(a).

A implementação mantém os nós de cada caminho preferencial em uma *splay tree*, considerando a profundidade do nó no caminho como a chave (implícita). As várias *splay trees* dos caminhos preferenciais de uma mesma árvore enraizada estão organizadas de acordo com os arcos da floresta que ligam estes caminhos. Veja a Figura 1(b).

A operação `maketree` simplesmente aciona a operação `makeSplay`. Para descrever o funcionamento das operações `link`, `cut`, `findroot` e `evert`, utilizamos a seguinte operação interna das *link-cut trees*:

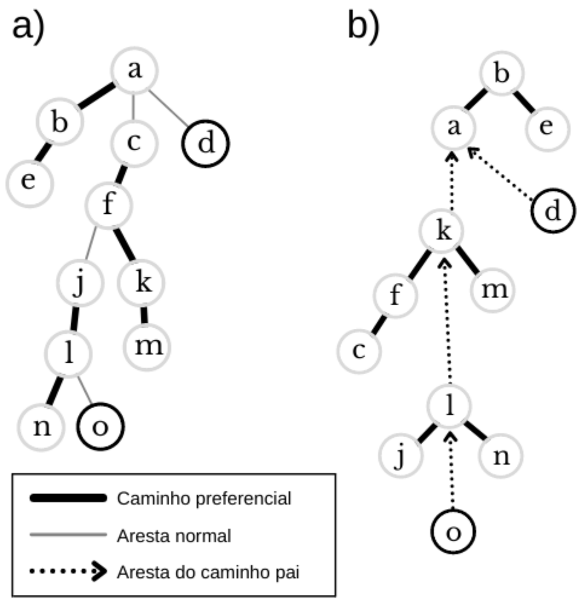

Figura 1: Exemplo de link-cut tree.

- `access(Node v)`: reorganiza os caminhos preferenciais estabelecendo `v` e cada ascendente de `v` como filho preferido de seu pai, e deixando `v` sem filho preferencial. Veja um exemplo do efeito da operação `access` na Figura 2.

Com essa operação, conseguimos descrever as operações `link`, `cut` e `evert` em termos das operações das *splay trees*.

No `link(u, v)`, acionamos o `access` [1] em `u` e em `v`, e depois acionamos um `join` das *splay trees* dos caminhos preferenciais de `u` e de `v`.

[comentário]O cut sofreu algumas alterações, mas que ainda não estão 100% validadas, como faço? Mostro as alterações aqui ou não as mostro?

No `cut(v)`, acionamos o `access` em `v` e o `split` no predecessor do `v` na *splay tree* do seu caminho preferencial.

No `evert(v)`, acionamos o `access` em `v` e em seguida o `reflectTree` na *splay tree* do caminho preferencial de `v`.

A operação `access` é a mais complexa, pois modifica várias das *splay trees* que compõem a *link-cut tree* de uma árvore enraizada. Devido tal complexidade, decidimos, nesse relatório, por omitir a descrição dessa operação em termos das operações das *splay trees*.

4 Conexidade totalmente dinâmica em florestas

Nesta seção descreveremos como utilizar *link-cut trees* para implementar florestas dinâmicas. Especificamente queremos implementar a seguinte interface para manutenção de uma floresta dinâmica:

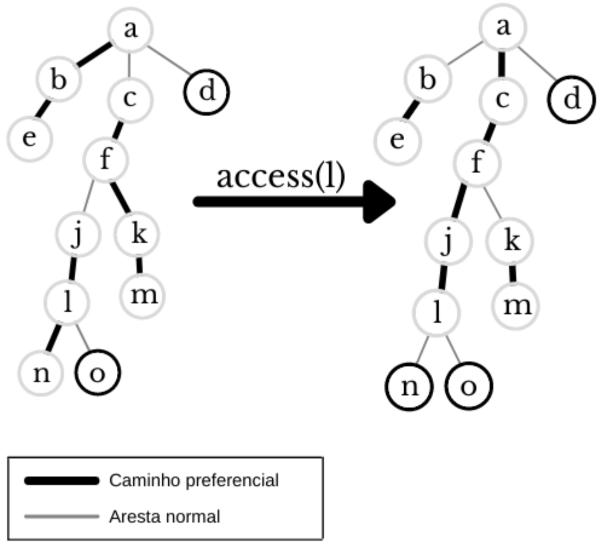

Figura 2: Exemplo da operação `access`.

- `dynamicForest(n)`: cria uma floresta com n vértices sem arestas; os vértices são identificados pelos números de 0 a $n-1$.

- `addEdge(LCT,i, j)`: recebe uma floresta LCT e dois vértices i e j em componentes distintos da floresta e adiciona a aresta ij na floresta. [Comentario] Devo colocar aqui que agora se cria um nó vértice, coisa que antes não ocorria?

- `deleteEdge(LCT,i, j)`: recebe uma floresta LCT e dois vértices i e j que correspondem a uma aresta da floresta e remove a aresta ij da floresta. [Comentario] Devo colocar aqui que agora se remove um nó vértice, coisa que antes não ocorria?

- `connected(LCT, i, j)`: devolve verdadeiro se i e j estão na mesma componente da floresta LCT, falso caso contrário.

- `inorderTraversal()`

Podemos implementar essas rotinas utilizando *link-cut trees* da seguinte maneira.

O `dynamicForest(n)` aciona o `maketree` n vezes a um custo O(n), retornando as referências das n *link-cut trees* criadas.

O `connected(LCT, i, j)` aciona o `findroot` para i e para j pertencentes à floresta LCT, e a partir dos resultados devolvidos decide se i e j estão ou não na mesma componente da floresta. O custo amortizado desta operação é O($\lg n$).

[Comentario] As rotinas seguintes sofreram alterações significativas, mas não estão funcionando 100%, explico o que se está fazendo no código mesmo não estando funcionando sempre?

O `addEdge(LCT, i, j)` aciona o `evert` em i e depois aciona o `link` nos nós correspondentes a i e j nas *link-cut trees*. O custo amortizado da operação é $O(\lg n)$.

O `deleteEdge(LCT, i, j)` aciona o `cut` nos nós correspondentes a i e j nas *link-cut trees*. O custo amortizado da operação é $O(\lg n)$.

5 Planos para o período de abril de 2022 a agosto de 2022

O primeiro passo estabelecido para o período de abril de 2022 até agosto de 2022, foi a criação de mais testes para fazer a análise de robustez e desempenho das *link-cut trees*. As atualizações no código podem ser encontradas em: `https://github.com/danlawand/conexidade-dinamica`.

O segundo passo estabelecido foi de ao final do estudo das *link-cut trees*, passar a estudar o problema mais geral, da conexidade totalmente dinâmica em grafos arbitrários. Uma das soluções da literatura para este problema utiliza a biblioteca de conexidade para florestas [8].

Trata-se de um algoritmo sofisticado, que mantém várias florestas geradoras do grafo corrente, representando níveis do grafo, e estas florestas são mantidas como no problema da conexidade dinâmica em florestas, por meio de *link-cut trees*. Projetávamos que esse estudo e a implementação deste algoritmo durasse cerca de 6 meses.

Um terceiro passo seria a possibilidade de estudar as *Euler tour trees*, uma outra estrutura de dados que pode ser usada de maneira semelhante às *link-cut trees* na resolução do problema de conexidade totalmente dinâmica. Mas a prioridade é o estudo do problema geral.

6 Atividades executadas no período de abril de 2022 a agosto de 2022

Durante a análise de robustez, quando submetemos a implementação que tínhamos a mais testes, foram encontrados alguns pequenos problemas e gastamos mais tempo do que previamos até ter uma versão robusta da implementação. O maior problema foi no esquema utilizado para implementar a rotina `reflectTree` e a rotina `evert`.

Depois disso começamos a estudar o algoritmo de Holm et al [8] que resolve o problema da conexidade dinâmica. Ao final desse estudo, definimos algumas fases para o projeto, e no atual momento conseguimos implementar parte dessas fases. O nosso programa ficou desta maneira:

1. Incluir os nós das arestas na implementação;
2. Criar percursos que identifiquem os nós das arestas;
3. Fazer testes do percurso em três níveis:

3.1. Atribuir um nível às arestas igual ao nível da árvore, e verificar se o percurso mostra todas as arestas da floresta;

3.2. Atribuir um nível às arestas diferente do nível da árvore, e verificar se o percurso mostra a floresta sem nenhuma aresta;

3.3. Atribuir níveis diferentes a cada aresta, e verificar se o percurso mostra apenas as arestas de mesmo nível da árvore;

4. Incluir a remoção dos nós arestas na implementação;

5. Refazer os testes e verificar se está tudo nos conformes;

Desse nosso programa, nós conseguimos implementar o item 1 integralmente, já o item 2 foi implementado, mas está no processo de validação, pois será no item 3 que saberemos se está robusto ou não. Dessa forma, o item 3 está parcialmente completo, visto que temos o item 3.1 completo, mas os itens subsequentes não. E por fim, os itens 4 e 5, não estão implementados, apesar de termos tentado incluir a remoção na implementação, que não funcionou.

Referências

[1] Link-cut tree. Wikipédia: a enciclopédia livre, March 2022. `https://en.wikipedia.org/wiki/Link/cut_tree`.

[2] R. Apte. Link-cut tree tutorial. CodeForces, March 2022. `https://codeforces.com/blog/entry/80383`.

[3] E. Demaine, J. Holmgren (scriber), J. Jian (scriber), M. Stepanenko (scriber), and M. Ishaque (scriber). Link-cut trees problems. Lecture Notes in Advanced Data Structures, 2012.

[4] E. Demaine and K. Lai (scriber). Dynamic graph problems. Lecture Notes in Advanced Data Structures, 2007.

[5] C. Demetrescu, I. Finocchi, and G. F. Italiano. *Handbook of Data Structures and Applications*, chapter Dynamic Graphs. Chapman and Hall/CRC, 2004.

[6] S. Even and Y. Shiloach. An on-line edge-deletion problem. *Journal of the ACM*, 28(1):1–4, 1981.

[7] M. R. Henzinger and V. King. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing (STOC)*, page 519, 1995.

[8] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM*, 48(4):723, 2001.

[9] R. Sedgewick and K. Wayne. *Algorithms*. Addison-Wesley, 4 edition, 2011.

[10] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing (STOC)*, page 114, 1983.

[11] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[12] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.