Conexidade Dinâmica

Projeto de Pesquisa para Iniciação Científica

Orientadora: Cristina Gomes Fernandes
Aluno: Daniel Angelo Esteves Lawand

Este projeto de pesquisa acompanha a requisição de bolsa de Iniciação Científica para o aluno Daniel Angelo Esteves Lawand.

1 Introdução

O objetivo deste projeto 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 [3].

O problema central será 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* [9], que provê resposta às consultas em tempo amortizado aproximadamente constante.

O problema de conexidade decremental foi resolvido por Even e Shiloach [4]. O método usa uma tabela onde se mantem 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* [7] ou das chamadas *Euler tour trees* [5] para representar a floresta. Estas estruturas implementam uma rotina chamada FindRoot(x), semelhante à rotina 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 FindRoot(x) = FindRoot(y). O tempo amortizado de atualização e consulta é $O(\lg n)$.

Para conexidade totalmente dinâmica em florestas com n vértices, podemos usar uma coleção das chamadas *Link-cut trees* [7] para representar a floresta. Esta estrutura implementa uma rotina chamada FindRoot(x), semelhante à rotina 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 FindRoot(x) = FindRoot(y). O tempo amortizado de atualização e consulta é $O(\lg n)$.Parágrafo igual ao de cima, porém retirei apenas a referência às Euler tour trees

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 *Euler tour 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, que planejamos estudar também, como parte deste projeto [2].Não sei o que fazer com esse parágrafo.

2 Splay trees

Splay tree é uma árvore binária de busca (ABB) com a propriedade de ser autoajustável, em que o último elemento acessado se torna a raiz da árvore [8]. Existem diferentes tipos de implementação, mas para o propósito deste trabalho utilizaremos a(s) seguinte(s) interface(s):

```
// splay.h
Node makeSplay();
void splay(Node);
void join(Node, Node);
void split(Node);
Node maxSplay(Node);
Node minSplay(Node);
void reflectTree(Node);
```

- `makeSplay()`: retorna a raíz de uma splay tree com apenas um nó, tendo limite de tempo de $O(1)$.
- `splay(v)`: recebe um nó `v`, tornando-o raíz da splay tree, com complexidade de $O(log_2 n)$ em tempo.
- `join(v, w)`: recebe duas splay trees, `v` e `w`, fazendo com que `v` seja filho direito de `w`. (Coloco a seguinte parte?) Sabe-se que de início `w` não tem filho direito. Tal rotina tem complexidade $O(1)$ em tempo.
- `split(v)`: recebe o nó `v` de uma splay tree e o separa de seu filho direito, gerando uma nova splay tree enraizada no filho direito de `v`. A complexidade dessa rotina em tempo leva $O(1)$.
- `maxSplay(v)`: retorna o nó mais profundo da splay tree em que `v` pertence, tendo complexidade de $O(log_2 n)$ em tempo.
- `minSplay(v)`: retorna o nó menos profundo da splay tree em que `v` pertence, tendo complexidade de $O(log_2 n)$ em tempo.
- `reflectTree(v)`: rotina em que inverte reciprocamente o filho direito com o esquerdo, tendo complexidade $O(1)$ em tempo.

Os nós (`Node`), nessa implementação, não possuem campo chave (`Key`), tratando-se de uma ABB com chaves implícitas, ou seja, não utilizamos as chaves para realizar qualquer tipo de operação. Em uma splay tree tradicional, as chaves da sub-árvore esquerda são menores do que a de seu pai e as chaves da sub-árvore direita são maiores do que a de seu pai, e para encontrar determinado nó, passamos a chave 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.

Nessa implementação, não há inserção ou deleção de nós, pois as splay trees crescem com a rotina `join` e diminuem através da rotina `split`, que quebra a árvore em duas.

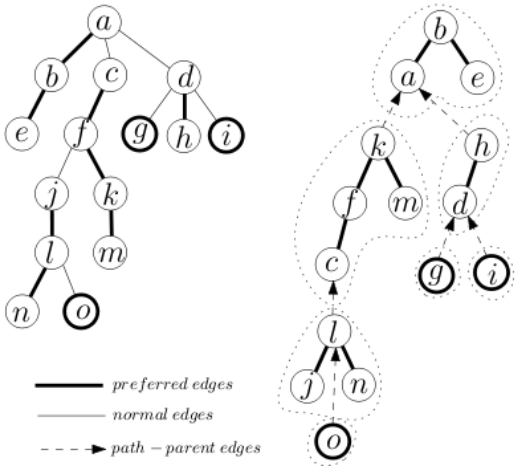

Figura 1: Exemplo de Link cut tree. (Pega no wikipedia)

3 Link-cut trees

Link cut tree é uma estrutura de dados que representa uma floresta de árvores enraizadas. Cada árvore na floresta é um caminho disjunto de vértices (caminho preferencial), onde cada caminho é representado por uma esturtura de dados auxiliar, no nosso caso usaremos a Splay tree.

Os nós na estrutura de dados auxiliar são ordenados de acordo a sua profundidade na Link cut tree. Desta forma, em uma splay tree podemos verificar que um nó é mais profundo que seu pai se for filho direito, e será mais superficial se for filho esquerdo. [10]

Na imagem a seguir temos como a Link cut tree representa os caminhos disjuntos de vértices em uma floresta de árvores enraizadas.

As rotinas usadas na implementação da Link cut tree foram:

```
Node maketree();
void link(Node, Node);
void cut(Node);
void access(Node);
Node findRoot(Node);
void evert(Node);
```

- `maketree()`: retorna a raíz de uma link cut tree com apenas um nó, tendo complexidade de $O(1)$ em tempo.

- `link(v, w)`: assumimos que `v` e `w` estão em árvores distintas. Nesta rotina acrescentaremos a aresta `vw` (ou `wv`??), fazendo com que `v` seja pai de `w`. Esta rotina tem complexidade O(access).

- `cut(v)`: dado que `v` não é a raiz da Link cut tree, removemos a aresta de `v` com seu pai, obtendo duas Link cut trees, uma enraizada em `v` e outra a qual o pai de `v` pertence. A complexidade desta rotina em tempo é O(access).

- `access(v)`: cria-se um caminho disjunto de vértices (caminho preferencial) da raiz da Link cut tree até o nó `v`, com complexidade em tempo de $O(??)$.

- `findRoot(v)`: retorna a raiz da Link cut tree em que o nó `v` pertence, levando $O(log_2 n + access)$ em tempo.

- `evert(v)`: modifica a Link cut tree, fazendo com que o nó `v` seja a raiz. Essa operação leva $O(access)$ em tempo.

4 Justificativa

O estudo de estruturas de dados e algoritmos mais sofisticados ou com análises mais complexas proporciona uma oportunidade de aprofundamento de conhecimentos importantes adquiridos durante um bom curso de graduação em Ciência da Computação.

Problemas dinâmicos em grafos dependem de estruturas de dados não triviais, para serem resolvidos de maneira eficiente. Várias destas estruturas não são abordadas nas disciplinas obrigatórias de uma graduação em Ciência da Computação.

Redes como a internet são extremamente dinâmicas, e ampliaram dramaticamente o interesse no estudo de problemas dinâmicos em grafos. O estudo da conexidade de tais redes é uma das primeiras etapas no entendimento da dinâmica da rede, por isso este estudo tem um papel central nessa área.

O tema escolhido para este projeto serve como motivação para o estudo de tópicos centrais de áreas importantes da Ciência da Computação, como projeto e análise de algoritmos e estruturas de dados.

O Daniel é um aluno do terceiro ano do Bacharelado em Ciência da Computação (BCC) do IME-USP. Embora suas notas no primeiro ano não sejam tão boas, levando-o a uma média 7,0 nas disciplinas que cursou, o Daniel não obteve nenhuma reprovação, e melhorou significativamente seu aproveitamento no segundo ano, obtendo notas todas acima de 7,0. O seu interesse em fazer iniciação científica demonstra esse amadurecimento e certamente vai consolidar ainda mais essa melhora do seu aproveitamento no curso.

As disciplinas do BCC que já foram concluídas pelo Daniel são suficientes para levar adiante este projeto. Dentre as disciplinas que ele cursou, destacamos MAC0121 Estruturas de Dados e Algoritmos I e MAC0323 Estruturas de Dados e Algoritmos II, que dão uma base essencial para o desenvolvimento desse projeto.

5 Objetivos

Neste projeto, o plano é estudar e implementar estruturas de dados e algoritmos usados no contexto de conexidade dinâmica de grafos. Além do funcionamento destas estruturas e algoritmos, serão estudados também suas análises de correção e seu consumo assintótico de tempo. Ou seja, serão abordados também conceitos e técnicas de análise de algoritmos.

O Daniel fez um rápido estudo preliminar do problema da conexidade dinâmica em grafos, que deve ser aprofundado durante os próximos meses. Ele também já implementou uma primeira estrutura de dados, baseando-se no livro de Sedgewick e Wayne [6], que será usada como base da implementação das chamadas *splay trees* [8]. Estas por sua vez são usadas na implementação das *link-cut trees*. No presente momento, o Daniel está estudando as splay trees, e deve começar a implementá-las nas próximas semanas.

O objetivo dessa iniciação científica é o aprendizado de várias estruturas de dados que possuem aplicações não apenas em conexidade dinâmica, mas em diversas outras áreas da Ciência da Computação. Vale destacar que as estruturas de dados que serão abordadas são não triviais e, sendo este um projeto de iniciação científica, é possível que apenas parte do material mencionado seja completamente estudado e implementado.

Como subproduto da iniciação científica, o Daniel deve também produzir um texto com tudo o que foi estudado.

6 Plano de trabalho e cronograma

O Daniel começou recentemente a trabalhar nesse projeto. Numa primeira fase, ele estudou a seção 1.1 do capítulo *Dynamic Graphs* no livro de Demetrescu et al. [3] sobre os conceitos básicos de grafos dinâmicos e as suas possíveis estruturas de dados para a implementação. Após isto, estudou o funcionamento das link-cut trees [1], implementou uma Árvore Binária de Busca e está estudando as splay trees.

A nossa intenção é inicialmente estudar e implementar os algoritmos e as estruturas de dados que são base para as link-cut trees. Tendo passado pelos conceitos fundamentais, inicia-se a implementação de tal estrutura. Junto a isso, se faz a análise de desempenho dos algoritmos envolvidos, delimitando o seu consumo assintótico de tempo. Ao término desta primeira etapa, o Daniel irá elaborar um texto descrevendo o que foi necessário para a implementação das link-cut trees e expor os resultados obtidos das análises. Estimamos que levaremos cerca de 4 meses para concluir esta etapa.

Ao passo que escreve o texto referente à primeira etapa, ele seguirá para o próximo tópico, que é o estudo e implementação das Euler tour trees. Primeiramente, ele irá estudar o funcionamento da estrutura de dados e verficar se esta faz uso de outras estruturas, caso positivo, ele irá estudar e implementar as estruturas de dados base para as Euler tour trees. Ao término deste estudo, o Daniel iniciará a implementação e análise das Euler tour trees.

Ao fim desta segunda etapa, o Daniel escreverá um texto descrevendo como foi a implementação das Euler tour trees e os resultados obtidos das análises. A partir disso, ele irá escrever um relatório intermediário contendo as informações dos textos já escritos e irá comparar o comportamento de cada estrutura de dados. Esperamos que esta etapa leve aproximadamente o mesmo tempo que a primeira fase do projeto, ou seja, cerca de 4 meses, embora ainda não tenhamos estudado esta estrutura de dados para ter uma noção mais concreta sobre esta previsão.

Se conseguirmos seguir este cronograma inicial, teremos ainda tempo para estudar mais um tópico relacionado ao tema. Como há bastante material disponível, pretendemos, a medida que nos aprofundamos nos estudos planejados para a primeira e segunda etapas, escolher mais algum tópico relacionado para estudar, se houver tempo após completarmos estas primeiras etapas do projeto.

7 Material e métodos

Há muito material sobre grafos dinâmicos na literatura. Primeiramente, utilizamos o livro de Demetrescu, Finnochi e Italiano [3] para ter uma familiaridade com os conceitos de grafos dinâmicos. Utilizaremos as notas de aula do Professor Demaine [1, 2], do MIT, e os artigos de Sleator e Tarjan [7] e de Henzinger e King [5] para nos aprofundarmos nas *link-cut trees* e nas *Euler tour trees*. Utilizaremos o livro de Sedgewick e Wayne [6] e um outro artigo de Sleator e Tarjan [8] para estudar as estruturas de dados que são base para implementação das estruturas de grafos dinâmicos.

O método que usaremos para conduzir essa iniciação científica é tradicional. O aluno estudará cuidadosamente os diversos resultados, e irá implementando parte do que for estudado, e teremos reuniões a cada duas semanas para discutir os assuntos estudados e as implementações. Ao mesmo tempo que estuda e implementa parte dos tópicos, o aluno escreverá um texto, o que possibilitará uma melhor avaliação de quão bem o material estudado está sendo absorvido.

8 Forma e análise dos resultados

Durante todo o período de estudo, além das implementações, o aluno estará preparando um texto, que, ao final do trabalho, conterá tudo que foi estudado na iniciação científica. Este é o principal objeto que pode ser usado na análise do trabalho que estará sendo desenvolvido. Fora isso, evidentemente esperamos que o aluno mantenha o bom desempenho (ou até melhore) no BCC.

Referências

[1] 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.

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

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

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

[5] 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.

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

[7] 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.

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

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

[10] Wikipedia. Link Cut Tree. Wikipedia Description.