LIII Simpósio Brasileiro de Pesquisa Operacional
João Pessoa - PB, 3 a 5 de novembro de 2021

ANÁLISE DA UTILIZAÇÃO DE ÁRVORES DINÂMICAS PARA A OTIMIZAÇÃO DO ALGORITMO DE DINIC EM PROBLEMAS DE FLUXO MÁXIMO EM GRAFOS

Felipe Kallás Silva
Universidade Federal de Itajubá - UNIFEI
Av. BPS, 1303 - Bairro Pinheirinho - Itajubá-MG
felipe.kallas@gmail.com

José Luiz da Silva Neto
Universidade Federal de Itajubá - UNIFEI
Av. BPS, 1303 - Bairro Pinheirinho - Itajubá-MG
jlsneto98@gmail.com

Alexandre Donizeti Alves
Universidade Federal do ABC - UFABC
Av. dos Estados, 5001 - Bairro Santa Terezinha - Santo André-SP
alexandre.donizeti@ufabc.edu.br

Edmilson Marmo Moreira
Universidade Federal de Itajubá - UNIFEI
Av. BPS, 1303 - Bairro Pinheirinho - Itajubá-MG
edmarmo@unifei.edu.br

RESUMO

Uma das principais aplicações das árvores dinâmicas é a otimização do algoritmo de Dinic, utilizado para resolver o problema do fluxo máximo em grafos. Neste artigo é apresentada uma análise da utilização destas estruturas e uma de suas soluções, denominada *Link-cut tree*, para a otimização deste algoritmo. Foram realizados testes para comparar o algoritmo original de Dinic com a versão otimizada. Os resultados mostram que o algoritmo mais rápido pode variar dependendo do grafo de entrada e a escolha deve ser baseada nas características dos grafos dados.

PALAVRAS CHAVE. ***Link-cut tree*****, Árvore** ***Splay*****, Problema do Fluxo máximo.**

TAG – Teoria e Algoritmos em Grafos

ABSTRACT

One of the main applications of dynamic trees is optimizing the algorithm of Dinic, used to solve the maximum flow problem in graphs. This article presents an analysis of the use of these structures and one of their solutions, named *Link-cut tree*, for optimization of this algorithm. In addition, the original Dinic algorithm was compared with the version optimized. The results show that the fastest algorithm can vary depending on the graph of input, and the choice must be based on the characteristics of the given graphs.

KEYWORDS. Link-cut tree, Splay tree, Maximum flow problem.

TAG – Theory and Algorithms in Graphs

1. Introdução

O problema das árvores dinâmicas [Sleator e Tarjan, 1983, 1985] é manter uma floresta arbitrária de n vértices que muda com o tempo por meio de inserções e exclusões de arestas. As operações típicas nesta estrutura incluem criar ou remover uma aresta entre dois vértices; aumentar o custo de todas as arestas da raiz até um vértice qualquer por um valor constante e alterar a raiz da árvore com o conteúdo de um determinado vértice da própria árvore. Também é necessário que a solução desenvolvida seja capaz de rapidamente responder questões sobre a conectividade de vértices; calcular o menor ancestral comum a dois vértices e encontrar a aresta de peso mínimo da raiz até algum vértice [Werneck, 2016].

Uma das principais aplicações das árvores dinâmicas é a otimização do algoritmo de Dinic, utilizado para resolver o problema do fluxo máximo em grafos. Neste cenário, este artigo apresenta uma análise da utilização destas estruturas e uma de suas soluções, a *Link-cut tree*, para a otimização do algoritmo de Dinic. Nesta análise, foram realizadas comparações de desempenho entre o algoritmo de Dinic tradicional e sua versão otimizada.

Nas demais seções deste artigo, serão apresentados: as árvores *Splay* e a *Link-cut tree*; uma discussão sobre as implementações de duas versões do algoritmo de Dinic que resolvem o problema de fluxo máximo; e a análise da eficiência destas implementações, comparadas através de uma série de testes com diferentes dados de entrada.

2. Conceitos Básicos

Dado um grafo direcionado $G = (V, E)$, onde cada aresta tem uma capacidade de fluxo positiva e cada vértice recebe o fluxo vindo dessa aresta, deseja-se saber qual é a quantidade de fluxo que é possível passar de um vértice S (*source*), até um vértice T (*target*), de tal forma que o fluxo em qualquer aresta seja menor ou igual à sua capacidade.

Buscando resolver esse problema de forma computacional, Ford e Fulkerson [1956] criaram o primeiro algoritmo. Depois vieram outros trabalhos [Edmonds e Karp, 1972; Sleator e Tarjan, 1983], com destaque para o algoritmo de Dinic [1970] que se tornou um dos algoritmos de grafos mais utilizados, principalmente devido à sua complexidade, que é $O(V^2E)$.

2.1. Algoritmo de Dinic

O algoritmo constrói um grafo em camadas utilizando *Breadth-First Search* (BFS), a partir do grafo original. O grafo em camadas é construído da seguinte maneira: para cada vértice v, calcula-se o $level[v]$, como sendo o tamanho do menor caminho de S até v, onde as arestas desse caminho apresentam uma capacidade maior que 0. Então, mantém-se essas arestas (v, u), onde $level[v] + 1 = level[u]$. Se o vértice T não pode ser alcançado com um único BFS, o algoritmo termina e retorna o fluxo máximo calculado. Caso contrário, usando somente arestas válidas no grafo em camadas, são realizadas várias *Depth-First Search* (DFS) de S até T, até atingir o fluxo de bloqueio, somando o valor do gargalo no fluxo de cada aresta presente no caminho encontrado. Este procedimento é repetido até que o fluxo máximo seja encontrado.

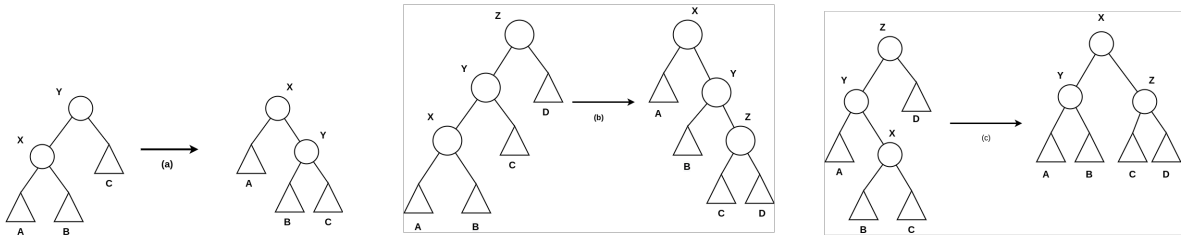
2.2. Árvore Splay

A árvore *Splay* foi introduzida por Sleator e Tarjan [1983] e se trata de uma árvore binária de busca auto ajustada, mas, ao contrário de uma árvore binária balanceada normal, o processo de ajuste ocorre toda vez que um item é acessado. Com isso, o tempo para consultar os itens que foram acessados mais recentemente torna-se menor. A medida que as operações são realizadas na estrutura, a tendência é que a árvore fique mais próxima de uma árvore balanceada. Desta forma, a complexidade amortizada das operações é próximo de $O(\log n)$.

O objetivo na elaboração desta estrutura foi criar uma forma mais simples de reestruturar a árvore depois de cada acesso, de forma a mover o item acessado mais próximo à raiz da árvore, assumindo que aquele item será acessado novamente, processo denominado *splaying*.

Para realizar o *splay* em um nó x, repete-se o processo de *splaying* até que este se torne a raiz da árvore. Este processo considera os seguintes casos:

- *Caso 1 (zig)*: Se $p(x)$, o pai de x, é a raiz da árvore, rotaciona-se a aresta unindo x com $p(x)$, conforme ilustra a Figura 1(a).
- *Caso 2 (zig-zig)*: Se $p(x)$ não é a raiz e x e $p(x)$ estão ambos na esquerda ou ambos na direita, rotaciona-se a aresta unindo $p(x)$ com o avô $g(x)$ e rotaciona-se a aresta juntando x com $p(x)$. A Figura 1(b) mostra um exemplo desta operação.
- *Caso 3 (zig-zag)*: Se $p(x)$ não é a raiz e x é o filho à esquerda e $p(x)$ é o filho à direita, ou vice-versa, rotaciona-se a aresta juntando x com $p(x)$ e, então, rotaciona-se a aresta juntando x com o novo $p(x)$. A Figura 1(c) ilustra um exemplo desta operação.



(a) Zig: Rotação simples (b) Zig-zig: Duas rotações simples (c) Zig-zag: Rotação dupla

Figura 1: Processos de *Splaying*

2.3. O problema das árvores dinâmicas

Dada uma floresta de N vértices de árvores enraizadas com custos em suas arestas, este problema consiste em modificar a árvore de acordo com as seguintes operações básicas:

- $link(v, w, c)$: insere uma aresta entre os vértices v e w com um custo c se v e w fazem parte de componentes conexas diferentes;
- $cut(v)$: remove a aresta entre v e o seu parente $p(v)$, se v não for a raiz da árvore à qual ele pertence;
- $makeRoot(v)$: transforma o vértice v na raiz da árvore que o contém.

Estas operações podem ser feitas em tempo constante. Agora suponha que seja necessário encontrar a aresta de menor custo do caminho de v até a raiz da árvore que o contém, ou então adicionar uma constante c ao custo de todas as arestas contidas neste caminho. Uma solução ingênua teria uma complexidade de tempo proporcional ao tamanho do caminho considerado ($\Theta(n)$). Este problema específico aparece no contexto de algoritmos de fluxo máximo e, por este motivo, há o interesse em construir uma estrutura de dados mais eficiente para resolver este problema.

Este problema é intitulado o *problema das árvores dinâmicas*. Existem algumas variações que necessitam responder outras perguntas e suportar outras operações, como por exemplo, identificar a conectividade entre vértices ou a soma dos custos das arestas de um caminho e a possibilidade de se adicionar valores aos vértices ao invés das arestas [Werneck, 2006].

Este problema foi originalmente introduzido por Sleator e Tarjan [1983], que apresentaram uma estrutura de dados capaz de resolver uma versão um pouco mais simples do que a descrição apresentada, em tempo $O(\log n)$. Em seguida, Tarjan [1984] apresentou outra estrutura de dados, denominada *Link-cut tree*. Esta estrutura reutilizou muitas das ideias contidas em seu trabalho anterior, com alterações que simplificaram a implementação e melhoraram o tempo de execução.

Sleator e Tarjan demonstraram que, com a *Link-cut tree*, é possível solucionar mais eficientemente vários problemas, tais como: a identificação do menor ancestral comum de dois vértices em uma floresta dinâmica ($O(n \log n)$); calcular vários tipos de fluxos de rede, como fluxo máximo ($O(nm \log n)$), fluxo de bloqueio ($O(m \log n)$) e fluxo acíclico ($O(m \log n)$); computar certos tipos de árvores geradoras mínimas com restrições; e implementar o algoritmo simplex [Nelder e Mead, 1965] de tal forma que a atualização de uma árvore de solução possível possa ser realizada em tempo $O(\log n)$ por pivô.

Além disso, outros estudos demonstraram que as *Link-cut trees* podem ser utilizadas como meio para implementar outras estruturas de dados. Andrade [2020] demonstrou que elas podem ser utilizadas para acelerar a implementação de filas de prioridade completamente retroativas que, segundo Sunita e Garg [2018], resolvem versões dinâmicas do algoritmo de Dijkstra [1959].

2.4. Link-cut tree

A implementação da *Link-cut tree* [Tarjan, 1984] não possui todas as funcionalidades descritas, como a inclusão de custos nas arestas, que pode ser realizada fazendo pequenas alterações nas árvores *Splay* e implementando *lazy propagation* na estrutura. A *Link-cut tree*, descrita nesta subseção, foi baseada na implementação de Demaine [2012].

Para realizar as operações da *Link-cut tree* de forma eficiente, é feita uma decomposição da árvore representada em um conjunto de caminhos. Vale notar que esta decomposição de caminhos será modificada em quase todas as operações realizadas, mesmo que a árvore representada não sofra modificações. Para realizar esta decomposição, foram introduzidos os seguintes conceitos:

- O **filho preferido** de um vértice v será *nulo*, se v foi o último vértice acessado na subárvore de v; ou w, se w é um filho de v e o último vértice acessado na subárvore de v está na subárvore de w.

- Um **caminho preferido** é um caminho maximal que consiste somente de arestas que levam de um vértice ao seu filho preferido. Um exemplo de uma possível decomposição de caminhos é ilustrada na Figura 2.

- Cada caminho preferido é representado por uma árvore *Splay*, indexando os vértices pela distância da raiz. Estas árvores são chamadas de **árvores auxiliares**.

- A raiz de cada árvore auxiliar armazena o **parente de caminho**, que é o pai do vértice mais alto em cada caminho preferido.

- A **árvore de árvores auxiliares** é o conjunto de todas as árvores auxiliares com seus respectivos parentes de caminho. A Figura 3 ilustra uma árvore de árvores auxiliares.

2.5. Implementação da Link-cut tree

Para a implementação da *Link-cut tree*, foi utilizada a estrutura apresentada no Algoritmo 1 para representar um vértice.

A principal operação que auxilia na modificação da estrutura de dados é o método $access(v)$. Todas as outras operações usam esta função de acesso como base e todas são simples de serem

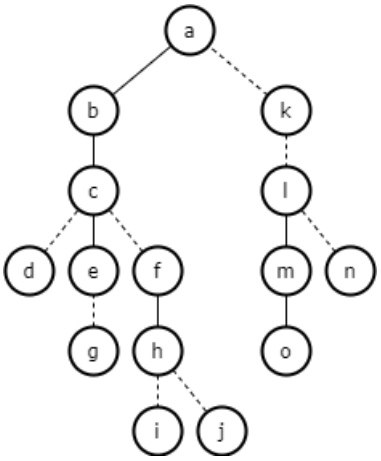


Figura 2: Exemplo de uma decomposição de caminhos de uma árvore. Arestas completas representam filhos preferidos e arestas pontilhadas representam os demais filhos.

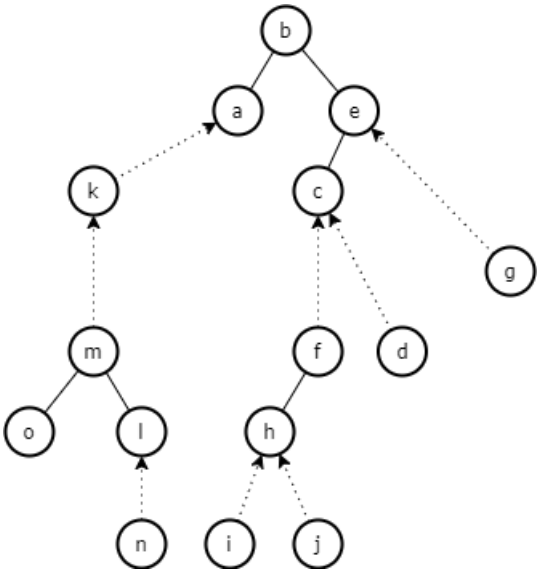


Figura 3: Exemplo de uma árvore de árvores auxiliares dada a decomposição de caminhos apresentada na Figura 2. As arestas completas indicam ligações dentro das árvores auxiliares enquanto ligações pontilhadas e com setas indicam os parentes de caminho.

realizadas depois que o acesso é definido. O papel da função $access(v)$ é fazer com que o caminho da raiz da árvore representada até v se torne um caminho preferido, e fazer com que o vértice v se torne a raiz da sua árvore auxiliar. Isso faz com que todos os vértices de v até a raiz da árvore representada façam parte de uma única árvore auxiliar. Desta forma, realizar as demais operações após a realização de uma operação de acesso é reduzido ao problema de fazer uma consulta em uma única árvore *Splay*.

Para se realizar a função $access(v)$, deve-se primeiro realizar uma operação de *splay* no vértice v para colocá-lo na raiz da sua árvore auxiliar. Note que, ao fazer isso, o ponteiro do parente de caminho também deve ser transferido para a nova raiz da árvore. A forma mais simples

Algoritmo 1 Estrutura do vértice

```
1: struct NODE
2:     left, right : Node        ▷ Filhos da esquerda e direita, respectivamente
3:     parent : Node             ▷ Pai do nó atual na mesma árvore auxiliar
4:     pathparent : Node         ▷ Parente de caminho do vértice atual
5: end struct
```

e direta de implementar isso é incorporar tal mudança diretamente na rotação realizada pela árvore *Splay*. Uma vez que v está na raiz da sua árvore auxiliar, é necessário retirar todos os vértices de profundidade maior que v. Porém, como a árvore *Splay* está indexada pela altura dos vértices na árvore representada, esta operação é equivalente a remover o filho preferido de v.

Em seguida, deve-se fazer com que o filho preferido do parente de caminho de v seja o próprio vértice v. Este processo deve ser repetido várias vezes até que v seja a raiz da árvore. Neste ponto, a árvore auxiliar do vértice v terá apenas os vértices do caminho da raiz até v. O Algoritmo 2 realiza ambos os passos da operação $access(v)$.

A operação $findroot(v)$ tem o papel de encontrar a raiz da árvore na qual o vértice v pertence. Note que isto não é o mesmo que encontrar a raiz da árvore de árvores auxiliares. Uma vez que a operação $access(v)$ já foi definida, para descobrir a raiz da árvore representada, basta realizar uma operação de $access(v)$ para que o vértice v pertença à mesma árvore *Splay* que a raiz da subárvore. Depois disso, basta descobrir qual é o vértice de menor índice dentro desta árvore *Splay* para descobrir qual vértice é a raiz. O Algoritmo 3 apresenta a implementação da operação $findroot(v)$.

Algoritmo 2 Access

```
1: procedure ACCESS (V)
2:     splay(v)

       Removendo o filho preferido de v
3:     v.right.pathparent ← v
4:     v.right.parent ← null
5:     v.right ← null

6:     while v.pahtparent ≠ null do
7:         w ← v.pathparent
8:         splay(w)

           Trocando o filho preferido de w por v
9:         w.right.pathparent ← w
10:        w.right.parent ← null
11:        w.right ← v
12:        v.parent ← w
13:        v.pathparent ← null
14:        splay(v)
```

Assim como a operação $findroot(v)$, a operação $findcost(v)$ é significativamente simplificada em uma *Link-cut tree* após a definição da operação $access(v)$. Sem perda de generalidade,

foi considerado que a operação $findcost(v)$ precisa retornar o mínimo do caminho da raiz até o vértice v, mas isso pode ser alterado para que, ao invés do mínimo, seja computado o máximo ou a soma dos valores em questão.

Para computar o custo mínimo da raiz até um vértice v, basta realizar uma operação $access(v)$ para fazer com que o caminho da raiz até v seja preferido. Depois de feito isso, tem-se uma árvore *Splay* que contém todos os valores relevantes para o cálculo. Para encontrar o mínimo destes valores, pode-se implementar a árvore *Splay* de tal forma que todos os vértices guardem o valor mínimo presente em sua subárvore. Desta forma, é possível manter estes valores atualizados durante a realização rotações, inserções e remoções sem a necessidade de iterar pela árvore. Como se deseja encontrar o mínimo global da árvore *Splay*, basta consultar o valor do mínimo da subárvore da raiz da árvore auxiliar, que é o próprio vértice v. A implementação da operação $findcost(v)$ é apresentada no Algoritmo 4.

Algoritmo 3 FindRoot

```
1: function FINDROOT (V)
2:     access (v)
3:     root ← v
4:     while root.l ≠ null do
5:         root ← root.l
6:     splay (root)
7:     return root
```

Algoritmo 4 FindCost

```
1: function FINDCOST (V)
2:     access (v)
3:     return v.subtreeMin
```

3. Otimização do algoritmo de Dinic usando árvores dinâmicas

Conforme visto anteriormente, o algoritmo desenvolvido por Dinic [1970] é utilizado para encontrar o fluxo máximo de um grafo, sendo um dos mais eficientes. Sleator e Tarjan [1983] desenvolveram um algoritmo baseado no algoritmo de Dinic que reduz o problema de fluxo em N problemas de fluxo de bloqueio, onde é possível encontrar este fluxo em $O(E \log V)$ utilizando árvores dinâmicas. Com isso, a solução deles tem complexidade $O(VE \log V)$, onde V é a quantidade de vértices e E a quantidade de arestas. As árvores dinâmicas são utilizadas para calcular os caminhos de aumento e manter a capacidade das arestas. Inicialmente, cada vértice é considerado como uma árvore separada. Em seguida, executa-se o algoritmo de Dinic da seguinte maneira:

- *Passo 1*: Seja $v = raiz(S)$. Se $v = T$, executa-se o *passo 4*. Caso contrário, executa-se o *passo 2*.
- *Passo 2*: $v \neq T$. Se nenhuma aresta sai de v, continua-se a execução a partir do *passo 3*. Senão, seleciona-se uma aresta (v, w) e realiza-se a operação $link\ (v, w, capacidade((v, w)))$. A execução retorna para o *passo 1*.
- *Passo 3*: Todos os caminhos de v até T estão bloqueados. Se $v = S$, calcula-se a capacidade não utilizada para cada aresta da árvore usando o custo. Senão, deleta-se do grafo todas as

arestas chegando em v. Para cada aresta (u, v) que é aresta da árvore, realiza-se a operação $cut(u)$, salvando a capacidade não utilizada. Retorna-se para o *passo 1*.

- *Passo 4*: $v = T$; o caminho da árvore de S até T é aumentativo. Seja $v = mincost(S)$. (a aresta $(v, pai(v))$ é a menor capacidade do caminho aumentativo). Realiza-se a operação $update(s, -c)$ e continua-se a execução a partir do passo 5.

- *Passo 5*: deleta as arestas sem capacidade restante. Seja $v = mincost(S)$. Se $cost(v) = 0$, deleta-se $(v, pai(v))$ do grafo e realiza-se a operação $cut(v)$, salvando uma capacidade não utilizada de zero, e repete-se o *passo 5*. Se $(cost(v) > 0)$, retorna-se para o *passo 1*.

Quando o algoritmo termina a sua execução, o fluxo final é calculado a partir do original e das capacidades não utilizadas. O algoritmo requer $O(V)$ árvores dinâmicas e cada uma leva $O(E \log V)$, totalizando uma complexidade $O(VE \log V)$.

4. Implementação dos algoritmos

Com base nos estudos realizados, foram implementadas as estruturas de dados *Splay* e *Link-cut tree*. Também foi implementado o algoritmo de Dinic, em sua definição original e uma versão com o cálculo do fluxo de bloqueio usando a *Link-cut tree*, permitindo que ambos sejam comparados.

Para garantir a corretude dos algoritmos implementados, estes foram testados com arquivos de testes tipicamente utilizados em competições de programação. Estes arquivos podem ser encontrados em diversas plataformas e a sua validade é garantida devido à grande quantidade de pessoas que os utilizam. Os resultados obtidos pelos algoritmos foram iguais aos resultados esperados em todos os testes realizados.

4.1. Metodologia

Foram realizados vários testes para comparar a eficiência dos algoritmos desenvolvidos. Os códigos foram desenvolvidos na linguagem C++17 com o compilador GNU G++17 7.3.0. Os testes foram executados em uma máquina com o sistema operacional Windows 10, um processador Intel(R) Core(TM) i7-4710HQ CPU @ 2.5GHz e 16,0GB de memória RAM.

Para comparar os algoritmos de fluxo máximo desenvolvidos, foram criadas diversas baterias de teste, cada uma contendo 20 testes e todos foram executados 5 vezes. Em cada caso, considerou-se que o tempo de execução da bateria foi a média dentre todas as $20 \times 5 = 100$ execuções.

Uma vez que o desempenho destes algoritmos varia drasticamente dependendo do tipo do grafo fornecido, os testes foram separados em seis classes. Cada classe representou grafos com propriedades distintas. Além disso, para cada classe de grafos, foram criadas três baterias de teste, cada uma com grafos com tamanho e ordem diferentes. Desta forma, foi possível verificar o desempenho de cada algoritmo em situações diferentes e compará-los. Os testes foram separados nas seguintes classes:

1. Grafos esparsos randômicos com capacidades baixas:
   - $|E| = 5 \cdot |V|$;
   - as arestas foram geradas entre pares de vértices escolhidos de forma aleatória;
   - existe pelo menos um caminho de S até T;
   - as capacidades têm um valor randômico no intervalo $[1, 10]$.

2. Grafos esparsos randômicos com capacidades altas:
   - $|E| = 5 \cdot |V|$;
   - as arestas foram geradas entre pares de vértices escolhidos de forma aleatória;
   - existe pelo menos um caminho de S até T;
   - as capacidades têm um valor randômico no intervalo $[1, 10^6]$.
3. Grafos esparsos com capacidades baixas e distância alta entre S e T:
   - $|E| = 5 \cdot |V|$;
   - a menor distância entre S e T é maior que $|V|/20$;
   - existe pelo menos um caminho de S até T;
   - as capacidades têm um valor randômico no intervalo $[1, 10]$.
4. Grafos esparsos com capacidades altas e distância alta entre S e T:
   - $|E| = 5 \cdot |V|$;
   - a menor distância entre S e T é maior que $|V|/20$;
   - existe pelo menos um caminho de S até T;
   - as capacidades têm um valor randômico no intervalo $[1, 10^6]$.
5. Grafos densos randômicos com capacidades baixas:
   - $|E| = |V|^2/10$;
   - arestas foram geradas entre pares de vértices escolhidos de forma aleatória;
   - existe pelo menos um caminho de S até T;
   - as capacidades têm um valor randômico no intervalo $[1, 10]$.
6. Grafos densos randômicos com capacidades altas:
   - $|E| = |V|^2/10$;
   - Arestas foram geradas entre pares de vértices escolhidos de forma aleatória;
   - Existe pelo menos um caminho de S até T;
   - As capacidades têm um valor randômico no intervalo $[1, 10^4]$.

5. Discussão dos resultados

Esta seção contém os resultados dos experimentos descritos na seção anterior. Os resultados estão separados de acordo com as classes definidas anteriormente. Para cada classe, foram realizados três ensaios diferentes, cada um com um grafo de ordem e tamanho diferentes.

5.1. Classes 1 e 2

Os tempos de execução dos ensaios das classes 1 e 2 podem ser encontrados nas Tabelas 1 e 2, respectivamente. Nos ensaios destas classes, notou-se que, em todos os casos, o tempo de execução do Dinic tradicional foi consideravelmente menor que o tempo de execução do Dinic com a otimização da *Link-cut tree*. Este resultado foi inesperado, uma vez que a complexidade do algoritmo de Dinic com a otimização mencionada é menor que a complexidade do Dinic tradicional.

O motivo pelo qual o resultado foi diferente do esperado é que o fluxo total nestes casos não é grande e a maioria dos caminhos aumentativos encontrados pelo algoritmo são pequenos.

$\|V\|$	10^4	10^5	10^6
Dinic tradicional	(12 ± 3) ms	(92 ± 23) ms	(804 ± 240) ms
Dinic com *Link-cut tree*	(31 ± 8) ms	(211 ± 37) ms	(1804 ± 369) ms

Tabela 1: Resultados dos ensaios de Classe 1

$\|V\|$	10^4	10^5	10^6
Dinic tradicional	(15 ± 4) ms	(114 ± 33) ms	(839 ± 211) ms
Dinic com *Link-cut tree*	(37 ± 7) ms	(255 ± 49) ms	(2104 ± 414) ms

Tabela 2: Resultados dos ensaios de Classe 2

Isso significa que iterar pelos vértices acaba sendo mais eficiente que fazer uma única atualização de custo em caminho na *Link-cut tree*, uma vez que a estrutura em questão possui uma alta constante de tempo para a execução das operações, já que ela usa como base as árvores *Splay*. Neste contexto, conclui-se que o uso do algorimto Dinic tradicional é a melhor escolha para grafos esparsos com caminhos aumentativos pequenos.

5.2. Classes 3 e 4

Os tempos de execução dos ensaios das classe 3 e 4 podem ser encontrados nas Tabelas 3 e 4, respectivamente. Antes de discutir os resultados obtidos por estes ensaios, vale notar que não foi possível concluir os testes do ensaio 4. Isso se deve ao fato de que todos os casos de teste estavam demorando muito tempo, para o caso $|V| = 10^6$, durante a sua execução pelo algoritmo Dinic tradicional. Como os ensaios continham 100 testes e todos estes necessitavam de mais de 15 minutos para se chegar na solução, concluiu-se que era inviável completar os testes para esta classe de grafos.

$\|V\|$	10^4	10^5	10^6
Dinic tradicional	(22 ± 4) ms	(318 ± 59) ms	(2774 ± 511) ms
Dinic com *Link-cut tree*	(17 ± 4) ms	(240 ± 41) ms	(1959 ± 359) ms

Tabela 3: Resultados dos ensaios de Classe 3

$\|V\|$	10^4	10^5	10^6
Dinic tradicional	(110 ± 25) ms	(7280 ± 1943) ms	> 15 minutos
Dinic com *Link-cut tree*	(45 ± 8) ms	(2113 ± 413) ms	(30321 ± 5653) ms

Tabela 4: Resultados dos ensaios de Classe 4

Notou-se que, nestes testes, a execução do algoritmo Dinic otimizado com a *Link-cut tree* foi consideravelmente mais rápido que o algoritmo tradicional. A diferença fica especialmente visível a medida que se aumenta o fluxo máximo contido nas arestas do grafo e a quantidade de vértices. Isto acontece, pois todos os caminhos aumentativos do grafo são grandes quando comparados à quantidade de vértices. Isso faz com que a etapa do algoritmo, que atualiza o fluxo contido nas arestas do caminho aumentativo, tenha uma execução muito demorada comparado à modificação de custos de um caminho realizado em uma única operação na *Link-cut tree*.

5.3. Classes 5 e 6

Os tempos de execução dos ensaios de classe 5 e 6 podem ser encontrados nas Tabelas 5 e 6, respectivamente. Como a quantidade de arestas em cada grafo era muito grande (grafos densos) e crescia de forma quadrática com relação à quantidade de vértices, os testes foram gerados com grafos de ordem menor.

$\|V\|$	2000	5000	8000
Dinic tradicional	(17 ± 4) ms	(144 ± 31) ms	(350 ± 91) ms
Dinic com *Link-cut tree*	(61 ± 17) ms	(504 ± 120) ms	(1122 ± 235) ms

Tabela 5: Resultados dos ensaios de Classe 5

$\|V\|$	2000	5000	8000
Dinic tradicional	(25 ± 6) ms	(190 ± 47) ms	(442 ± 92) ms
Dinic com *Link-cut tree*	(103 ± 27) ms	(692 ± 186) ms	(1445 ± 349) ms

Tabela 6: Resultados dos ensaios de Classe 6

Observou-se que o tempo de execução do algoritmo Dinic com a *Link-cut tree* foi consideravelmente maior que o algoritmo tradicional. Como já explicado na análise dos testes de classe 1 e 2, a maioria dos caminhos aumentativos encontrados pelo algoritmo são pequenos e, portanto, é mais rápido iterar pelos vértices que realizar uma modificação na estrutura de dados.

6. Considerações finais

Através da análise dos dados obtidos, percebe-se que a solução do algoritmo de Dinic com a otimização da *Link-cut tree* não deve ser usada em todos os casos, uma vez que, para a maioria dos grafos, ela aumenta o tempo de execução do algoritmo. O motivo é que, apesar *Link-cut tree* diminuir a complexidade do algoritmo, ela introduz uma constante de tempo muito alta. A exceção são os grafos que têm uma grande distância entre os vértices *S* e *T*. Neste caso, o processo de calcular a aresta de corte dos caminhos aumentativos e fazer a atualização do fluxo nestas arestas tem um alto custo para o algoritmo original, enquanto a *Link-cut tree* consegue fazer isso rapidamente para caminhos longos.

Desta forma, para decidir qual algoritmo de fluxo máximo utilizar, deve-se fazer um estudo da aplicação e entender precisamente o formato dos grafos com os quais se está trabalhando. Outra possível solução é implementar ambos os algoritmos e, antes de executá-los, fazer uma rápida estimativa de qual algoritmo terá o melhor desempenho. Essa estimativa deve considerar vários fatores, como por exemplo, a relação da quantidade de vértices e arestas no grafo e o tamanho do menor caminho entre S e T.

Vale ressaltar que a estrutura *Link-cut tree* possui outras aplicações e ainda é estudada em pesquisas das áreas de Algoritmos, Estrutura de Dados e Teoria dos Grafos. Alguns exemplos de estudos recentes [Sunita e Garg, 2018; Andrade, 2020] incorporaram esta estrutura para melhorar o desempenho na resolução de outros problemas. Desta forma, é possível que trabalhos futuros explorem outras aplicações de estruturas que resolvem o problema das árvores dinâmicas.

As implementações desenvolvidas para a realização desta análise estão disponíveis na plataforma *GitHub*[1], permitindo que outros pesquisadores possam utilizá-las em suas pesquisas.

[1] https://github.com/felipe-kallas/LinkCutTree

Referências

Andrade, J. W. J. (2020). Estruturas de dados retroativas: Aplicações na dinamização de algoritmos. Master's thesis, Universidade Federal de Itajubá.

Demaine, E. (2012). *Dynamic Graphs I*. MIT 6.851 Advanced Data Structures.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271.

Dinic, E. A. (1970). Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Math. Dokl.*, 11:1277–1280.

Edmonds, J. e Karp, R. M. (1972). Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264. ISSN 0004-5411. URL https://doi.org/10.1145/321694.321699.

Ford, L. R. e Fulkerson, D. R. (1956). Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404.

Nelder, J. A. e Mead, R. (1965). A Simplex Method for Function Minimization. *The Computer Journal*, 7(4):308–313. URL https://doi.org/10.1093/comjnl/7.4.308.

Sleator, D. D. e Tarjan, R. E. (1983). A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(2):362–391.

Sleator, D. D. e Tarjan, R. E. (1985). Self-adjusting binary search trees. *J. ACM*, 32(3):652–686. ISSN 0004-5411. URL https://doi.org/10.1145/3828.3835.

Sunita e Garg, D. (2018). Dynamizing dijkstra: A solution to dynamic shortest path problem through retroactive priority queue. *Journal of King Saud University - Computer and Information Sciences*. ISSN 1319-1578. URL http://www.sciencedirect.com/science/article/pii/S1319157817303828.

Tarjan, R. E. (1984). *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, USA. ISBN 0898711878.

Werneck, R. F. (2006). *Design and Analysis of Data Structures for Dynamic Trees*. PhD thesis, Princeton University.

Werneck, R. F. (2016). *Dynamic Trees*, p. 605–609. Springer New York, New York, NY. ISBN 978-1-4939-2864-4. URL https://doi.org/10.1007/978-1-4939-2864-4_121.