FELINE: UM MÉTODO DE INDEXAÇÃO PARA CONSULTAS DE ALCANÇABILIDADE EM GRANDES GRAFOS ESTÁTICOS E DINÂMICOS

RENÊ RODRIGUES VELOSO

FELINE: UM MÉTODO DE INDEXAÇÃO PARA CONSULTAS DE ALCANÇABILIDADE EM GRANDES GRAFOS ESTÁTICOS E DINÂMICOS

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

Orientador: Wagner Meira Jr.
Coorientador: Loïc Pascal Gilles Cerf

Belo Horizonte
Julho de 2015

Rodrigues Veloso, Renê

V443f Feline: Um Método de Indexação para Consultas de Alcançabilidade em Grandes Grafos Estáticos e Dinâmicos / Renê Rodrigues Veloso. — Belo Horizonte, 2015
xxix, 114 f. : il. ; 29cm

Tese (doutorado) — Universidade Federal de Minas Gerais
Orientador: Wagner Meira Jr.

1. Computação - Teses. 2. Indexação - Teses.
3. Teoria dos grafos - Teses. 4. Algoritmos - Teses.
I. Título.

CDU 519.6*52(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Feline: um método de indexação para consultas de alcançabilidade em grandes grafos estáticos e dinâmicos

RENÊ RODRIGUES VELOSO

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. WAGNER MEIRA JÚNIOR - Orientador
Departamento de Ciência da Computação - UFMG

PROF. LOIC PASCAL GILLES CERF - Coorientador
Departamento Ciência da Computação - UFMG

PROF. CAETANO TRAINA JÚNIOR
Instituto de Ciências Matemática e de Computação - USP

PROF. JAYME LUIZ SZWARCFITER
Núcleo de Computação Eletrônica - UFRJ

PROF. NIVIO ZIVIANI
Departamento de Ciência da Computação - UFMG

PROF. SEBASTIÁN ALBERTO URRUTIA
Departamento de Ciência da Computação - UFMG

PROF. MOHAMMED JAVEED ZAKI
Rensselaer Polytechnic Institute

Belo Horizonte, 17 de julho de 2015.

Aos meus familiares e amigos.

Ao Departamento de Ciências da Computação da Universidade Estadual de Montes Claros (UNIMONTES).

"Tudo o que colhi até aqui fui eu que plantei. Portanto, tenho nada a reclamar. Somente a agradecer, porque até para plantar exige-se aprendizado."

(Autor desconhecido)

Resumo

Um problema-chave em diversas aplicações baseadas em grafos direcionados é a necessidade de responder rapidamente se existe um caminho entre dois vértices u e v quaisquer, isto é, se u alcança v, o que é denominado consulta de alcançabilidade. Esse problema é particularmente desafiador no caso de grafos muito grandes.

Uma abordagem comumente aplicada é o pré-processamento dos grafos, de forma a produzir uma estrutura de índice eficiente e que permita o rápido acesso às informações de alcançabilidade entre os vértices. No entanto, a maioria dos métodos de indexação existentes não são escaláveis. Dessa forma, a necessidade de métodos eficientes e escaláveis tem ganhado destaque nos últimos anos.

Pode ser necessário indexar tanto grafos estáticos quanto dinâmicos. A indexação de grafos estáticos, i.e., grafos que não se alteram com o decorrer do tempo, deve ser tal que os tempos para construir o índice de alcançabilidade e para responder às consultas sejam os menores possíveis. A indexação em grafos dinâmicos, i.e., grafos que podem sofrer alterações ao longo do tempo, é um desafio maior. Neles, além do tempo de construção, o tempo de atualização do índice frente a inserções e remoções de vértices e arestas deve ser o menor possível (e muito menor do que reconstruir todo o índice em cada atualização do grafo), sem que o tempo para responder às consultas aumente. Há ainda a necessidade de gerenciar a ocorrência de ciclos, lidando com os componentes fortemente conectados.

É proposto, então, neste trabalho, um novo método de indexação denominado Feline (*Fast rEfined onLINE search*). Esse método constrói um índice a partir da representação do grafo em um plano bidimensional, da qual são extraídas as informações de alcançabilidade em tempo constante para uma porção significativa de consultas. Experimentos demonstram a eficiência do método em relação às abordagens estado da arte.

Como extensão do Feline, propomos também um método para a manipulação de índices para grafos dinâmicos. Essa extensão tem como base um algoritmo de Ordenação Topológica Dinâmica (DTO), o qual realiza atualizações no índice a cada

modificação do respectivo grafo. Estudos comparativos são realizados e um estudo preliminar para o suporte à inserção em lotes de arestas é apresentado. Em seguida, as conclusões e oportunidades de trabalhos futuros finalizam esta tese.

Palavras-chave: consulta de alcançabilidade, grafos direcionados estáticos, grafos direcionados dinâmicos.

Abstract

A key problem in many applications based on directed graphs is the need to quickly know if there is a path between two given vertices u and v, i.e., if u reaches v, which is termed *reachability query*. This problem is particularly challenging in the case of very large graphs.

One commonly applied approach is the preprocessing of the graphs in order to produce an efficient index structure allowing quick access to the reachability information between all the vertices. However, most existing indexing strategies are not scalable. Thus, the need for new efficient and scalable strategies have gained prominence in recent years.

It may be necessary to index both static and dynamic graphs. The indexing of static graphs , i.e., graphs that do not change with the passage of time, should be such that the time to build the reachability index and to answer the queries are kept to a minimum. Indexing in dynamic graphs, i.e., graphs which may change over time, is a bigger challenge. In them, besides the construction time, the time to update the index (due to the insertions and deletions of vertices and edges) should be as small as possible and much less than rebuilding the whole index on each update, without increasing the time to answer the queries. Also, there is the need to manage the occurrence of cycles, dealing with the strongly connected components.

It is proposed, then, in this research, a new indexing strategy termed Feline (*Fast Refined onLINE search*). This approach constructs an index from the representation of the graph in a two-dimensional plane, from which are extracted reachability informations in constant time for a significant portion of queries. Experiments demonstrate the efficiency of the Feline compared to state of the art approaches.

As an extension of Feline, we also propose a method for handling indexes for dynamic graphs. This extension is based on a Dynamic Topological Ordering algorithm (DTO), which updates the index in every modification of the respective graph. Comparative studies are conducted and a preliminary study to support the batch insertion of edges is also presented. Then the conclusions and future work opportunities finalize

this work.

Keywords: reachability query, static directed graphs, dynamic directed graphs.

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Lista de Notações

Notação	**Descrição**
V	vértices do dígrafo
E	arestas do dígrafo
E_{DAG}	arestas do DAG associado
SCC	função $G \mapsto 2^V$ dando os componentes fortemente conectados
r	função *representante* $V \mapsto V_{\text{DAG}}$
$\prec$	ordenação topológica do DAG
$\prec_x$ ($\prec_y$)	ordenação topológica referente ao eixo X (respec. Y), onde $\prec$ é implementado como um vetor.
$\prec^u$	ordem topológica (posição) do vértice u em $\prec$
$a \prec_= \text{b}$	denota $\prec^a \leq \prec^b$
$in(S)$ ($out(S)$)	conjunto de vértices vizinhos de $S \subseteq r(u) \mid u \in V$
$\deg_{in}(u)$ ($\deg_{out}(u)$)	grau de entrada (grau de saída) de u
$u\ E^*\ v$	u alcança v em E, onde $*$ é a *estrela de Kleene*
δ	o conjunto de vértices que terão suas posições alteradas

Sumário

Capítulo 1

Introdução

Apresenta-se neste capítulo uma breve introdução ao tema de pesquisa, bem como as motivações que justificam a realização desta tese. Conceitos básicos e algumas definições necessárias ao entendimento do tema são elucidados e, em seguida, os objetivos de pesquisa são discutidos.

1.1 Motivação, Definições e Proposta

Diversos problemas de pesquisa da atualidade estão relacionados à manipulação de grandes conjuntos de dados [Cuzzocrea et al., 2013]. Tratando-se da Web, esses conjuntos de dados são provenientes de diversas fontes como as Redes Sociais, sites pessoais (Blogs/Vlogs) e de entretenimento, além de bibliotecas digitais. Uma forma bastante utilizada de estruturar as informações de fontes como essas é por meio de grafos.

Kumar et al. [2000] destaca que páginas na Web e seus *hyperlinks* podem ser modelados como vértices e arestas de um grafo direcionado, respectivamente. Outro exemplo do uso de grafos na Web é encontrado nas *Redes Sociais On-line* (RSO) [Arqué & Nettleton, 2012; Fan, 2012; Wilson et al., 2012] como, por exemplo, o Facebook, que possui mais de 1 bilhão de usuários e uma média de 130 amigos por usuário, resultando em um grafo social com mais de 1 bilhão de vértices e aproximadamente 97 bilhões de arestas. Outros exemplos incluem grafos de citações, redes de comunicação e aplicações usando RDF (*Resource Description Framework*).

Dado um grafo direcionado $G = (V, E)$, com V sendo um conjunto finito de vértices e $E \subseteq V^2$ sendo um conjunto de arestas, e dados $u, v \in V$, um problema-chave em muitas aplicações baseadas em grafos é a necessidade de saber se existe um caminho entre u e v em G, i.e., saber a ***alcançabilidade*** entre u e v. Esse problema é particularmente desafiador no caso de grafos muito grandes, principalmente

em aplicações onde há a necessidade de saber a alcançabilidade entre vértices a todo o instante [Jin et al., 2012; Seufert et al., 2013; Yildirim et al., 2010; van Schaik & de Moor, 2011; Cheng et al., 2013]. Essa necessidade pode ser averiguada no contexto de redes sociais [Dhia, 2012]. Nesse caso, por exemplo, para determinar políticas de privacidade como "somente minha família e meus amigos podem visualizar minhas fotos de aniversário", "somente meus filhos e seus amigos podem ter acesso a este texto que escrevi" ou "somente os meus vizinhos podem saber que estou on-line". Esse mesmo conceito é utilizado em aplicações de segurança eletrônica ou mesmo em marketing inteligente na Web ([Anyanwu & Sheth, 2003]). Na área de *link structure analysis*, esta consulta é usada para determinar alcance entre websites [Lempel & Moran, 2000].

Outro exemplo é encontrado na área de Compiladores com os algoritmos de *Program Slicing* (ou fatiamento de programas, em português). Uma fatia de programa consiste das partes ou componentes de um programa que potencialmente afetam os valores computados em algum ponto de interesse [Tip, 1995]. Há várias aplicações para o fatiamento, como um melhor entendimento do programa, auxílio à depuração, testes, manutenção, entre outras. Um método para calcular um fatiamento é mapear o programa em um grafo de dependências e usar consultas de alcançabilidade, onde as instruções do programa são os vértices e as dependências são as arestas. Por exemplo, dado o programa da Figura 1.1 e seu grafo de dependências, pode-se encontrar todas as instruções que são influenciadas pela inicialização da variável "*sum*", obtidas a partir do grafo de dependências, pela computação de todos os vértices que são alcançáveis a partir de $sum = 0$ (técnica denominada de *Forward Slice* [Horwitz et al., 1988; Tip, 1995]). Como o grafo de dependências cresce com o tamanho do programa, obter informações de alcançabilidade se torna uma tarefa custosa.

Em um contexto de aplicação onde frequentemente deseja-se saber a alcançabilidade entre vértices, uma abordagem comum é realizar um pré-processamento dos grafos. Esse pré-processamento produz um índice que permite o rápido acesso às informações de alcançabilidade. O que se denomina de índice, portanto, é um sumário sobre a alcançabilidade entre vértices do grafo em questão, permitindo que consultas sejam feitas de forma eficiente. Contudo, estudos recentes demonstram que o tempo de acesso a essas informações não é a única preocupação das abordagens existentes. Há também a preocupação com o tamanho do índice gerado e com o tempo necessário para construí-lo, como discutiremos ao longo desta tese.

```
void main () {
    int i = 1;
    int sum = 0;
    while (i<11) {
        sum = add(sum, i);
        i = add(i, 1);
    }
    printf("sum = %d\n", sum);
    printf("i = %d\n", i);
}

static int add(int a, int b)
{
    return(a+b);
}
```

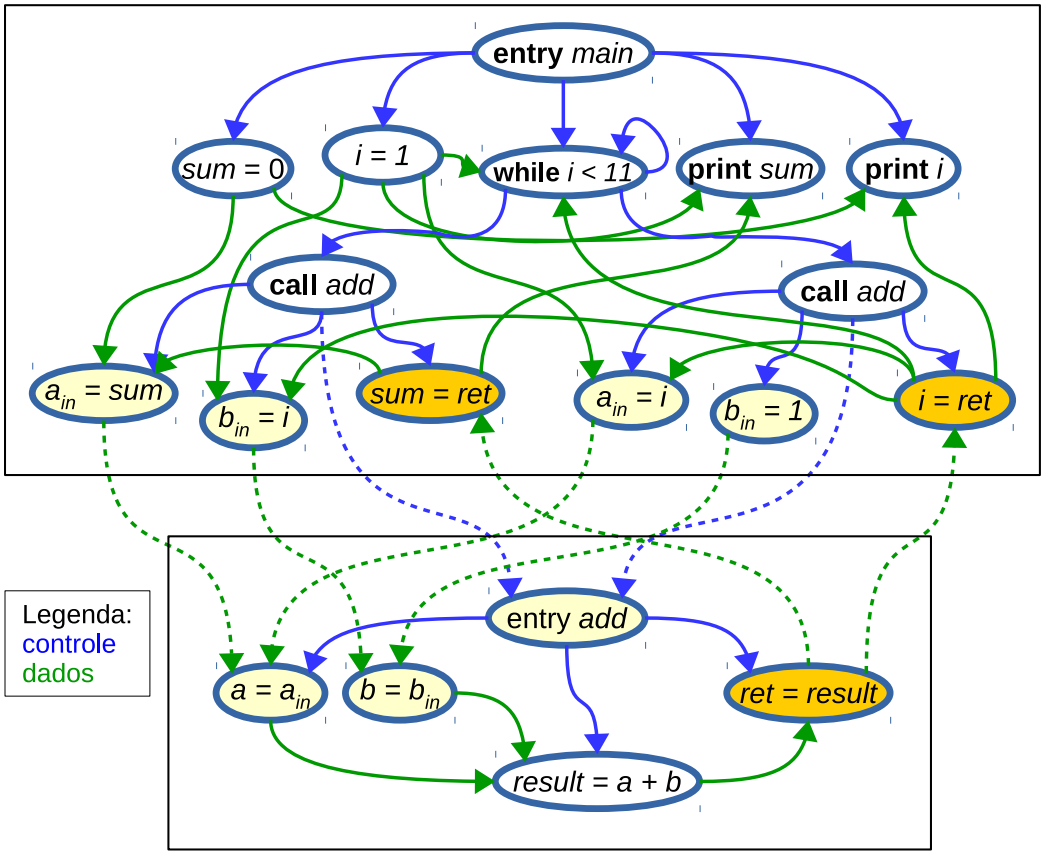


Figura 1.1. Exemplo de um programa e seu grafo de dependências. (Figura traduzida de [GrammaTech, 2014])

1.1.1 Consultas de Alcançabilidade em Dígrafos

Dado um grafo direcionado (ou dígrafo) $G = (V, E)$, uma *consulta de alcançabilidade* responde se existe um caminho nesse grafo partindo de $u \in V$ até $v \in V$, ou seja, se v é alcançável a partir de u. Neste trabalho, uma consulta de alcançabilidade será representada por $u \rightsquigarrow^{?} v$. Essa consulta retorna *verdadeiro* se existir um caminho que conecte os dois vértices ou *falso*, caso contrário.

Definição 1 (Alcançabilidade)**.**

$$\forall (u, v) \in V^2,\ v \textit{ é alcançável a partir de } u, \textit{ denotado por } uE^*v,$$
$$\textit{se e somente se} \begin{cases} u = v \\ \textit{ou} \\ \exists (u, w) \in E \ e\ w\ E^*\ v \end{cases}.$$

A relação de alcançabilidade E^* (*estrela de Kleene*) é reflexiva e transitiva. Ela não é antissimétrica, uma vez que não é uma ordem parcial, pois G pode conter ciclos. Portanto, dado um dígrafo G qualquer, é necessário construir um grafo direcionado acíclico (ou seja, o DAG[1]) $G_{\text{DAG}} = (V_{\text{DAG}}, E_{\text{DAG}})$ por meio do agrupamento de todo componente fortemente conectado (veja a Definição 2) de G em novos vértices e inserindo-os em V_{DAG}, retendo em E_{DAG} as arestas entre esses componentes. O algoritmo de Tarjan [Tarjan, 1972] pode ser utilizado para identificar os componentes fortemente conectados. Sua complexidade de tempo é $O(|V| + |E|)$. Responder uma consulta de alcançabilidade de $u \in V$ para $v \in V$ (em G) é, portanto, o mesmo que responder a uma consulta de $SCC(u) \in V_{\text{DAG}}$ para $SCC(v) \in V_{\text{DAG}}$. No entanto, a relação de alcançabilidade agora é uma ordem parcial. Assim, toda consulta de alcançabilidade no grafo original pode ser respondida no DAG.

Definição 2 (Componente Fortemente Conectado)**.** *Dados um dígrafo (V, E) e um vértice $u \in V$, o componente que inclui u, denotado $SCC(u)$ é definido por $\{v \in V \mid u\ E^*\ v\ \wedge v\ E^*\ u\}$.*

1.1.2 Indexação para Alcançabilidade

Um índice de alcançabilidade é uma rotulação dos vértices do grafo a que se pretende indexar, conforme a Definição 3. Por exemplo, considere o grafo da Fig. 1.2. Nele, o conjunto de vértices $V = \{a, b, c, d, e, f, g, h, i, j\}$ é mapeado para o conjunto de rótulo

[1]DAG significa *Directed Acyclic Graph*, ou grafo direcionado acíclico, em português.

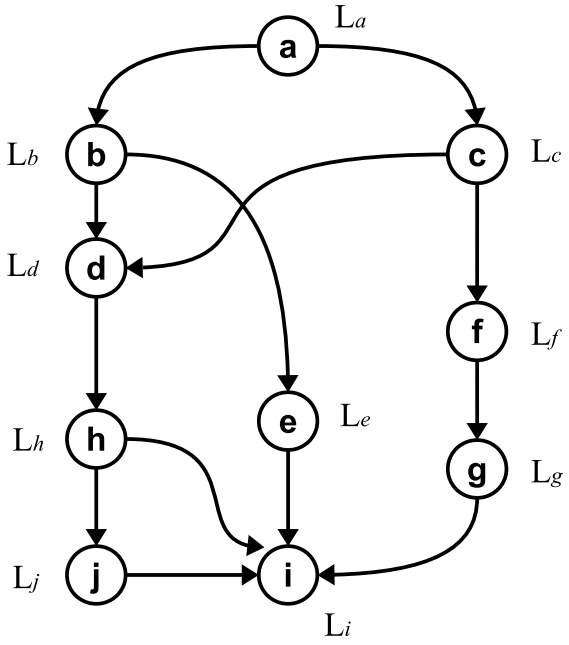


Figura 1.2. Exemplo de uma rotulação dos vértices do grafo.

$L = \{L_a, L_b, L_c, L_d, L_e, L_f, L_g, L_h, L_i, L_j\}$. Cada rótulo em L é usado para constatar a alcançabilidade dos vértices em V, isto é, para saber se o vértice a alcança o vértice f, basta verificar os seus rótulos.

Definição 3 (Índice de Alcançabilidade)**.** *Um índice de alcançabilidade de um DAG* $G_{DAG} = (V_{DAG}, E_{DAG})$ *é uma* **rotulação** $L = \{L_1, L_2, L_3, ..., L_{|V_{DAG}|}\}$*, onde, para dois vértices* i *e* j*, um rótulo* L_i *é atribuído a* i *e um rótulo* L_j *é atribuído a* j*, de forma que há uma relação entre* L_i *e* L_j *que define a alcançabilidade entre* i *e* j*.*

Definição 4 (Método de Indexação de Alcançabilidade)**.** *Um método de indexação de alcançabilidade* $I = (R, C)$ *é composto por duas funções, um rotulador* R *e um gerenciador de consultas* C*. Um* **rotulador** *é uma função* $R : G_{DAG} \rightarrow L$ *que atribui uma rotulação* L *para o grafo* G_{DAG}*. Um* **gerenciador de consultas** C *é uma função que, dada uma consulta* $u \rightsquigarrow^? v$*, mapeia os vértices* u *e* v *em seus respectivos rótulos* L_u *e* L_v *e, em seguida, verifica a alcançabilidade entre os vértices tomando como base os seus rótulos. Essa função retorna verdadeiro ou falso, atuando de forma* ***off-line*** *ou* ***on-line***[2]*. O* gerenciador de consultas C *é off-line quando requer apenas os rótulos dos vértices para inferir a alcançabilidade e é on-line quando precisa, além dos rótulos*

[2]O termo *on-line* foi primeiro utilizado no trabalho de Jin et al. [2012].

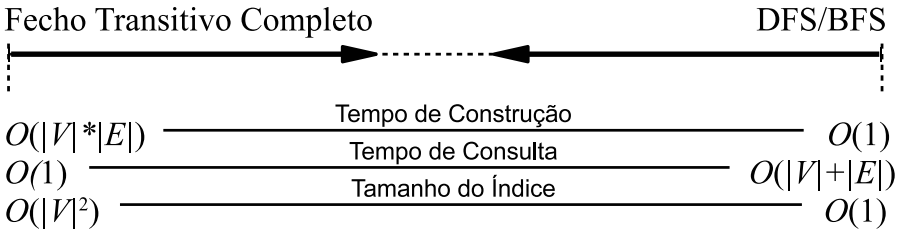


Figura 1.3. Compromisso entre o tempo de consulta, tempo de construção e tamanho do índice.

dos vértices, realizar uma busca no grafo (para todas ou somente para um subgrupo de consultas realizadas).

Historicamente, como descrito por Yildirim et al. [2010], há basicamente duas abordagens extremas para responder a consultas de alcançabilidade em DAGs, representando dois extremos de acordo com a estratégia adotada (veja a Figura 1.3 traduzida de [Yildirim et al., 2010]). O desempenho de cada estratégia varia de acordo com o compromisso entre três objetivos: *tempo de construção*, *tempo de consulta* e *tamanho do índice*.

Definição 5 (Tempo de Construção)**.** *Tempo de construção é o tempo exigido para extrair as informações do grafo, ou para pré-processá-lo, com o intuito de gerar um índice (ver Definição 3) que materialize o fecho transitivo do grafo*[3] *ou torne a busca menos custosa.*

Definição 6 (Tamanho do Índice)**.** *Refere-se à quantidade e tamanho dos rótulos, i. e., $|L|$ (Definição 3) em bytes.*

Definição 7 (Tempo de Consulta)**.** *É o tempo necessário para responder a alcançabilidade entre dois vértices u e v, dada uma consulta $u \rightsquigarrow^{?} v$.*

Referindo-se novamente à Figura 1.3, a estratégia à esquerda representa as técnicas que extraem e armazenam o fecho transitivo completo do grafo, criando o que se denomina de *índice de alcançabilidade*, conforme a Definição 3, pois resume as relações de alcançabilidade entre os vértices. O tempo necessário para extrair o fecho transitivo completo do grafo (i. e., o índice) é o tempo de construção. Embora esse índice permita respostas em tempo constante, ele possui complexidade de espaço quadrática

[3] O fecho transitivo $T(G_{\text{DAG}})$ de um grafo $G_{\text{DAG}} = (V_{\text{DAG}}, E_{\text{DAG}})$ é um grafo que contém uma aresta (u, v) sempre que existir um caminho de u até v em V_{DAG}.

no número de vértices (o que caracteriza o *tamanho do índice*) que, nesse caso, é impraticável para grafos muito grandes. O *tempo de consulta* é o tempo para verificar a alcançabilidade entre dois vértices, que é constante nesse caso.

No outro extremo da Figura 1.3 (lado direito), as técnicas são baseadas em uma busca em profundidade (ou mesmo em largura) para verificar a alcançabilidade entre dois vértices. Essa estratégia propõe-se a expandir novos vértices desde uma determinada origem até que o objetivo seja encontrado, respondendo, assim, se existe um caminho em tempo $O(|V_{\text{DAG}}| + |E_{\text{DAG}}|)$ para cada consulta (tempo de consulta) sem a necessidade de indexação (i. e., com o tempo de construção e o tamanho do índice constantes). Tomando como base essas estratégias, o objetivo dos métodos atuais, que compõem o estado da arte, é ser uma estratégia intermediária entre esses dois extremos. Tais métodos tentam manter um compromisso entre realizar consultas rápidas e gerar um índice pequeno (também diminuir o tempo de construção desse índice).

1.2 Tipos de Índices

Conforme definido anteriormente, um método de indexação de alcançabilidade é dividido em duas partes, um rotulador e um gerenciador de consultas, sendo este último de dois tipos: on-line e off-line.

É *off-line* quando requer apenas os rótulos dos vértices para inferir a alcançabilidade e é *on-line* quando precisa, além dos rótulos dos vértices, realizar uma busca no grafo em alguns casos. Consequentemente, um gerenciador *off-line* necessita de um índice cujos rótulos expressam o fecho transitivo completo do grafo. Por conseguinte, para grafos maiores, esses índices se tornam grandes (de fato, possui complexidade espacial quadrática) e necessitam ser comprimidos.

Já no gerenciador *on-line*[4], os rótulos são aplicados para guiar as buscas no grafo e realizar podas no espaço de busca. Por vezes, esses rótulos são suficientes para evitar que as buscas sejam feitas, respondendo de forma rápida às consultas. Contudo, em alguns casos, a busca no grafo é inevitável. Isso acontece, por exemplo, em consultas com respostas representando ***falsos-positivos***.

Definição 8 (Falso-Positivo)**.** *Dada uma consulta $u \rightsquigarrow^{?} v$, para dois vértices $u, v \in V_{DAG}$, o seu resultado é um falso-positivo quando os rótulos de u (L_u) e v (L_v) indicam a sua alcançabilidade, mas não há caminho em G_{DAG} entre u e v. Neste caso, os rótulos sozinhos não permitem decidir a alcançabilidade entre u e v.*

[4]Um método de indexação de alcançabilidade que usa um gerenciador on-line é mais conhecido por *Refined On-line Search Approach*, ou somente Método de Busca On-line [Jin et al., 2012].

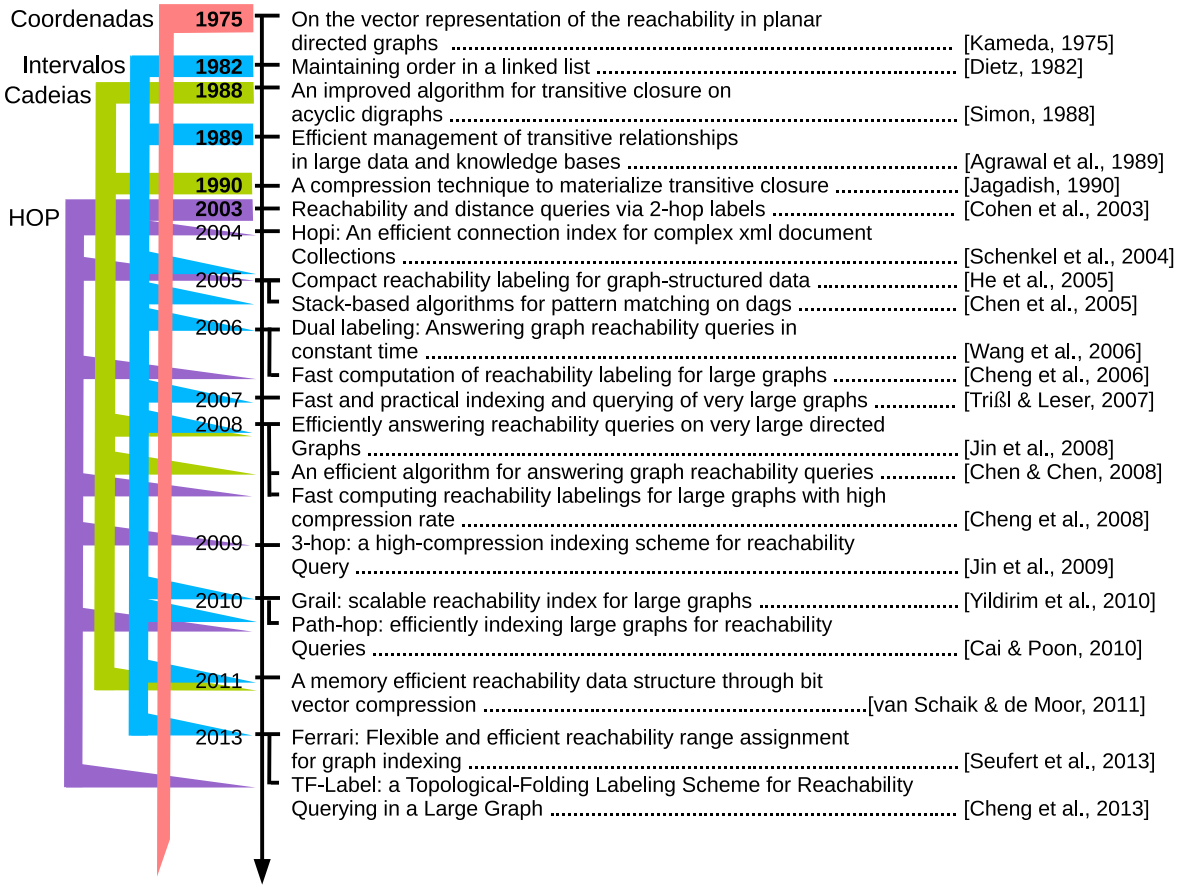


Figura 1.4. Linha do tempo mostrando as quatro famílias de métodos de indexação para alcançabilidade.

Os índices utilizados pelos métodos estão divididos em quatro famílias, a saber: *intervalos*, *cadeias*, *hop* e *coordenadas*. A Figura 1.4 apresenta uma linha do tempo mostrando os trabalhos seminais e os trabalhos influenciados por eles [Kameda, 1975; Dietz, 1982; Simon, 1988; Agrawal et al., 1989; Jagadish, 1990; Cohen et al., 2003; Schenkel et al., 2004; He et al., 2005; Chen et al., 2005; Wang et al., 2006; Cheng et al., 2006; Trißl & Leser, 2007; Jin et al., 2008; Chen & Chen, 2008; Cheng et al., 2008; Jin et al., 2009; Chen, 2009; Yildirim et al., 2010; Cai & Poon, 2010; van Schaik & de Moor, 2011; Chen & Chen, 2011; Seufert et al., 2013; Cheng et al., 2013]. Cada família é representada por uma cor e múltiplas setas indicam abordagens híbridas.

A família dos intervalos contém os métodos que rotulam os vértices utilizando intervalos obtidos pelo caminhamento no DAG, por exemplo, em pós-ordem ou pré-ordem. Dessa forma, esse caminhamento é feito a partir de uma árvore de cobertura extraída dos grafos. Grande parte dos métodos dessa família são do tipo *on-line*.

A família das cadeias possui métodos quase que predominantemente do tipo off-

line. A ideia básica é decompor o grafo em cadeias, i. e., ele é segmentado em vários conjuntos disjuntos de vértices e arestas (um vértice aparece em somente um conjunto). Em seguida uma rotulação é aplicada a cada vértice dentro das cadeias e entre as cadeias. Por vezes abordagens híbridas utilizando intervalos são encontradas.

Definição 9 (Fecho Transitivo Completo de um DAG)**.** *Dado um DAG* $G_{DAG} = (V_{DAG}, E_{DAG})$*, o fecho transitivo completo de* G_{DAG} *é um grafo* $G_{DAG}+ = (V_{DAG}, E_{DAG}+)$ *tal que, para todo* $u, v \in V_{DAG}$*, há uma aresta* $(u, v) \in E_{DAG}+$ *se e somente se existe um caminho partindo de* u *até* v *em* G_{DAG}*.*

A família *hop*, que possui somente métodos *off-line*, comprime o fecho transitivo do grafo por meio da definição de conjuntos de entrada e saída para cada vértice. Em geral, cada vértice mantém uma lista de vértices intermediários que ele pode alcançar e uma lista de vértices intermediários que podem alcançá-lo. Consequentemente, para responder às consultas, uma operação de junção (*join*) é feita entre listas para determinar a ocorrência de vértices em comum.

Os métodos da família das *coordenadas* buscam fazer um mapeamento dos vértices em um plano Cartesiano (espaço n-dimensional), rotulando os vértices com as coordenadas no plano. Tais métodos representam uma vertente pouco explorada em indexação para alcançabilidade. De fato, somente uma única proposta foi encontrada em [Kameda, 1975], mas sem resultados experimentais. Um outro método, denominado PathTree [Jin et al., 2008] da família das cadeias, utiliza somente uma coordenada (eixo-x) de um plano bidimensional para compor os seus rótulos.

O algoritmo de Kameda [1975] representa uma das primeiras tentativas de indexação de grafos, aplicável de forma restrita aos DAGs planares. Ele constrói uma representação de um DAG planar num espaço bidimensional, seguindo uma abordagem *upward*, onde cada aresta é uma curva monotonicamente crescente na direção vertical (i. e., eixo-y no plano). Esse algoritmo permite que a alcançabilidade seja inferida em tempo constante (para DAGs planares). O leitor encontra maiores detalhes sobre o algoritmo no próximo capítulo.

1.3 Grafos Dinâmicos

Conforme comentado anteriormente, dígrafos muito grandes são encontrados em diversos tipos de aplicações incluindo, por exemplo, redes sociais, redes de sensores, redes biológicas e diversas bases de dados da Web. Tais grafos não surgiram grandes, mas se tornaram grandes de forma incremental, como é o caso das redes sociais onde novos usuários fazem novos amigos na rede (e perdem amigos também). Como consequência,

há uma emergente necessidade por algoritmos que processem de forma eficiente grandes dígrafos nas quais seus vértices e arestas podem aparecer e desaparecer ao longo do tempo.

Dado que o dígrafo sofre alterações ao longo do tempo, se a versão atual do dígrafo é considerada final, então ele pode ser indexado tal que todas as consultas subsequentes podem ser eficientemente respondidas. Contudo, não é comum haver uma versão "final" de um dígrafo dinâmico. O dígrafo continua evoluindo e as consultas devem ser respondidas corretamente. Quando um dígrafo é muito grande, o tempo para reconstruir um índice do início é frequentemente muitas ordens de magnitude maior que o tempo entre duas consultas ou mesmo entre duas atualizações. O índice, portanto, precisa ser incrementalmente atualizado. Se o dígrafo é acíclico (DAG), esse índice é uma ordenação topológica: caminhos de u até v não passam por vértices posicionados depois de v, i.e., a travessia no grafo a partir de u pode ser cancelada se tais vértices forem alcançados. No caso de uma única ordenação topológica de um DAG, Pearce & Kelly [2006] propuseram uma solução de nome PK.

Uma abordagem para atualização do índice de alcançabilidade é aquela denominada Dagger. Proposta por Yildirim et al. [2013], Dagger é escalável e detecta a ocorrência de ciclos quando há inserção de uma nova aresta no grafo, manipulando componentes fortemente conectados por meio de uma estrutura de dados que comporta o dígrafo cíclico e sua versão acíclica. Maiores detalhes sobre essa abordagem podem ser encontrados no próximo capítulo.

1.4 Definição do Problema

Quanto mais os métodos se aproximam da extração do fecho transitivo completo do grafo (vide Figura 1.3), mais demorada é a fase de construção do índice. Isso representa uma clara desvantagem dos métodos off-line. Um tempo de construção menor é um fator importante, principalmente em contextos onde o grafo possa ter a sua estrutura alterada ao longo do tempo, exigindo uma nova indexação. Por exemplo, em grafos sociais, onde novas conexões entre usuários são feitas (ou desfeitas) a todo o momento. Nessas situações, os métodos on-line apresentam vantagens, pois possuem um processo de indexação mais simples, uma vez que o grafo é mantido em memória. Contudo, há poucos métodos on-line que suportam grafos grandes [Yildirim et al., 2010].

De acordo com a linha do tempo exposta na Figura 1.4, pouco se fez em relação à família de métodos baseados em coordenadas. De fato, o trabalho seminal de Kameda [1975] não apresenta resultados experimentais, além de ser uma proposta aplicável

somente a grafos planares. Por esse motivo, as perguntas que definem o problema abordado neste trabalho, a respeito de grafos estáticos, são:

- Quais adaptações são necessárias ao algoritmo de Kameda [1975] para permitir a indexação de DAGs não planares?

- Um índice da família das coordenadas, combinado com um método de busca on-line, representa uma solução eficiente para o problema de alcançabilidade em grafos não planares? Onde, por eficiente, entende-se:

  - Tal abordagem traz ganhos no tempo de construção do índice?
  - Tal abordagem reduz o tamanho dos índices construídos?
  - Para os casos em que uma busca no grafo é necessária, é possível reduzir o tempo das consultas?

Este trabalho dedica-se a responder a esses questionamentos e, pela proposta de um método on-line da família das coordenadas, apresentar uma solução com melhor compromisso entre tempo de construção do índice, tamanho do índice e tempo de consulta. Um objetivo deste trabalho é, portanto, demonstrar a possibilidade de obter tal método, sendo uma variante do algoritmo de Kameda, onde as consultas do tipo on-line resolvam os casos de exceção para os grafos não planares.

Esta tese trata também da manutenção de informações sobre alcançabilidade em grafos direcionados dinâmicos, com detecção de ciclos e manutenção de componentes fortemente conectados. Para tanto, investigou-se algoritmos eficientes que nos permitem inserir e remover arestas de um grafo mantendo o índice atualizado. Sendo assim, uma questão importante a ser resolvida é: *Como atualizar de forma eficiente o índice de alcançabilidade sem reconstruí-lo do início em toda atualização de seu grafo?* Para tanto, após a verificação de semelhanças entre o problema de DTO (*Dynamic Topological Ordering*) e alcançabilidade, o algoritmos Feline-PK foi proposto, permitindo a inserção em lote de arestas. Avaliou-se experimentalmente essa abordagem utilizando datasets tradicionais com milhões de arestas, demonstrando a superioridade de Feline-PK quando comparado ao estado-da-arte.

1.5 Organização do Texto

O restante deste trabalho está organizado da seguinte forma:

- O Capítulo 2 apresenta o referencial teórico e estado da arte em indexação para buscas de alcançabilidade. Nele, o conteúdo possui foco nos métodos recentes e que são escaláveis, uma vez que o tema de pesquisa assim o exige.
- O Capítulo 3 aborda o método Feline, bem como definições importantes e os seus algoritmos. O capítulo também apresenta os experimentos realizados e os resultados que comprovam a eficiência do método.
- O Capítulo 4 dedica-se a formalizar a proposta de extensão do método Feline para indexar grafos com atualizações. Experimentos são apresentados e, em seguida, um estudo preliminar em inserções de arestas em lotes é mostrado.
- O Capítulo 5 apresenta as conclusões desta tese e algumas propostas de trabalhos futuros.

Capítulo 2

Referencial Teórico

Apresenta-se neste capítulo os trabalhos relacionados à indexação e consultas de alcançabilidade. Para tanto, está organizado da seguinte forma:

- O trabalho seminal das abordagens da família das coordenadas, o algoritmo de Kameda [1975], é apresentado na Seção 2.1.

- Na Seção 2.2, os trabalhos relacionados são discutidos. Para tanto, eles são divididos em abordagens escaláveis e não-escaláveis. O critério para essa divisão foi a avaliação dos resultados experimentais publicados em trabalhos recentes e em *surveys* da área. Os principais métodos considerados escaláveis pela literatura são apresentados.

- A indexação de grafos dinâmicos é brevemente apresentada na Seção 2.3, seguida do método que compõe o estado da arte: Dagger.

2.1 O algoritmo de Kameda

Quando um grafo é representado num plano sem que as suas arestas se cruzem (*grafo planar*), ele divide esse plano em regiões denominadas *faces* [Nishizeki & Rahman, 2004; Nishizeki & Chiba, 2008]. Cada região, ou face, é caracterizada pelas arestas que a contornam. Por exemplo, o grafo planar da Figura 2.1 possui 3 faces, $F1$, $F2$ e $F3$ (também chamada de face externa ou infinita). Essa noção se aplica aos DAGs planares, sem perda de generalidade.

O trabalho de Kameda [1975] aparece como um precursor em indexação para alcançabilidade. Contudo, o método empregado por esse trabalho é aplicável somente

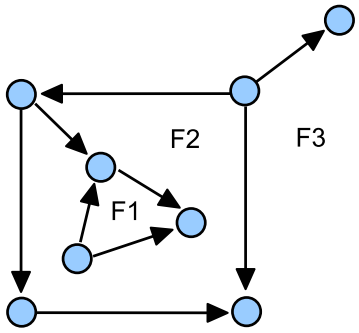


Figura 2.1. Exemplo de dígrafo com 3 faces: F1, F2 e F3 (face externa).

a DAGs planares e com uma restrição, o grafo deve ter uma representação do tipo ***upward* planar**.

Definição 10 (Fontes e Sumidouros)**.** *Um vértice u é uma **fonte** se ele não possui arestas que chegam nele. Um vértice u é um **sumidouro** se ele não possui arestas que partem dele.*

Definição 11 (Representação *upward* planar)**.** *Um DAG planar G_{DAG} admite uma representação do tipo upward planar, se todas as suas fontes e sumidouros estão na face externa e todas as arestas são representadas por curvas monotonicamente crescentes na direção vertical.*

A Figura 2.2-(A) mostra um DAG que obedece a essa restrição, de acordo com a Definição 11. Essa restrição do algoritmo de Kameda assemelha-se a uma técnica usada na área de visualização de grafos (*graph drawing*). Conforme Bertolazzi et al. [1998]; Hopcroft & Tarjan [1974]; Papakostas [1994]; Healy & Lynch [2005]; Bertolazzi et al. [2002], uma representação do tipo *upward* planar de um DAG planar G_{DAG}, é uma representação de G_{DAG} num plano bidimensional, criando um posicionamento dos vértices de tal forma que os sumidouros e as fontes devem estar na mesma face (externa) em lados opostos.

O objetivo do algoritmo de Kameda é rotular vértices utilizando vetores, de tal maneira que u alcança v se e somente se $L_u < L_v$, onde L_u é um vetor bidimensional[1] atribuído ao vértice u e onde L_v é um vetor bidimensional atribuído ao vértice v, cada um representando as coordenadas x e y no plano, onde $L_u = (x_u, y_u)$ e $L_v = (x_v, y_v)$. Dado um DAG $G_{\mathrm{DAG}} = (V_{\mathrm{DAG}}, E_{\mathrm{DAG}})$ planar, que possua uma representação do tipo *upward* planar, seja $n = |V_{\mathrm{DAG}}|$, o algoritmo realiza uma busca em profundidade mais à

[1]Originalmente definido como um vetor d-dimensional, porém o autor postula que 2 dimensões é suficiente para um DAG planar.

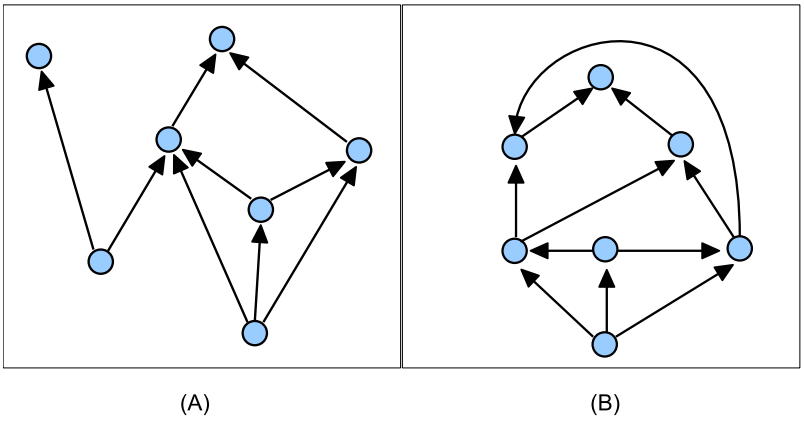


Figura 2.2. (A) Exemplo de grafo direcionado planar em uma representação *upward*. (B) Exemplo de grafo direcionado planar que não possui uma representação *upward* planar.

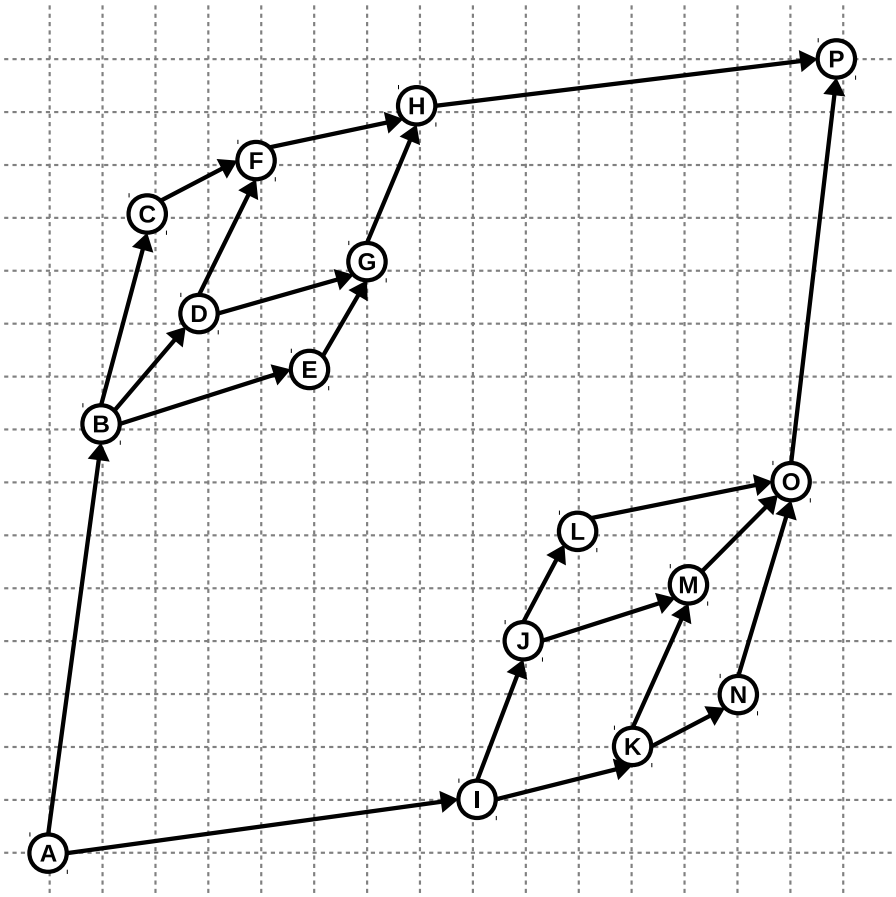


Figura 2.3. Exemplo de grafo planar.

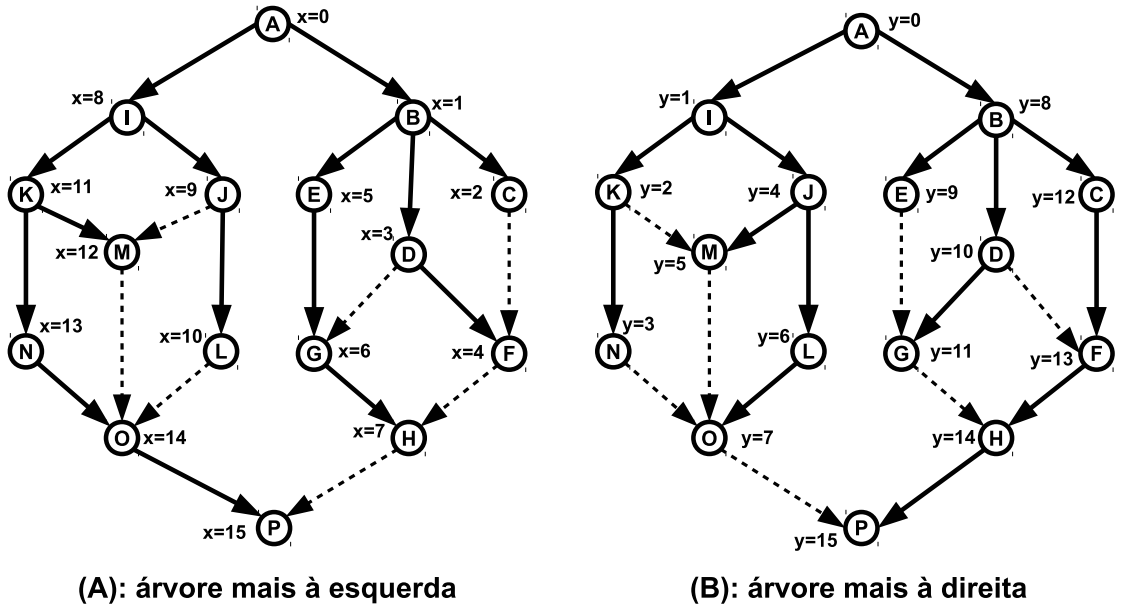


Figura 2.4. Exemplo: (A) árvore de cobertura à esquerda (arestas em negrito) e coordenadas X (primeiro componente do vetor). (B) árvore de cobertura à direita e coordenadas Y (segundo componente do vetor).

esquerda (em relação aos vértices da lista de adjacências) em G_{DAG}, com um contador inicial $i = n$. Quando um vértice é desempilhado, atribuímos o valor de i ao primeiro componente do rótulo L desse vértice e i é decrementado. O mesmo é feito com uma busca mais à direita. Por exemplo, suponha o DAG planar da Figura 2.3, onde o resultado dessa primeira fase do algoritmo de Kameda pode ser visto na Figura 2.4(A).

Definição 12 (Ordenação topológica)**.** *Um ordenação topológica de um DAG* $G_{DAG} = (V_{DAG}, E_{DAG})$*, é uma ordenação* $\prec$ *em* V_{DAG} *tal que* $\forall (u, v) \in E_{DAG}$*,* $u \prec v$*.*

Definição 13 (Ordenação topológica $\prec_x$)**.** *Uma ordenação topológica* $\prec_x$ *obtida por uma busca em profundidade mais à esquerda em* G_{DAG}*.*

Definição 14 (Ordenação topológica $\prec_y$)**.** *Uma ordenação topológica* $\prec_y$ *obtida por uma busca em profundidade mais à direita em* G_{DAG}*.*

A árvore de cobertura[2] resultante é chamada de *árvore de cobertura à esquerda.* O segundo componente dos vetores que rotulam cada vértice são igualmente computados, mas com uma busca mais à direita (Figura. 2.4(B)) e o resultado é denominado

[2]Uma árvore de cobertura de um grafo G_{DAG} é um grafo acíclico que contém todos os vértices de G_{DAG} e é um subgrafo de G_{DAG} [Cormen et al., 2001].

de *árvore de cobertura à direita*. Cada componente dos vetores L são ordenações topológicas (Definições 13 e 14). Essas ordenações obtidas ($\prec_x$ e $\prec_y$) são complementares. O tempo de construção desse índice é $O(n)$ e o tempo de consulta é constante, uma vez que o grafo é planar.

Teorema 1. *Seja $G_{DAG} = (V_{DAG}, E_{DAG})$ planar e admitindo uma representação do tipo upward planar. Existem duas ordenações topológicas complementares de V_{DAG}, denotadas $\prec_x$ e $\prec_y$ (chamadas ordenações mais à esquerda e mais à direita), tal que, para $u, v \in V_{DAG}$, existe um caminho direcionado de u até v se e somente se $u \prec_x v$ e $u \prec_y v$. [Kameda, 1975; Sairam et al., 1990]*

Na figura, os rótulos de cada vértice são formados pelo par (x, y), na qual o valor de x é encontrado na árvore de cobertura à esquerda e o valor de y encontrado na árvore de cobertura à direita. Por exemplo, pela Figura. 2.4, temos que o vértice J possui o rótulo $(9, 4)$. Já o vértice P, possui o rótulo $(15, 15)$. Dados dois vértices distintos u e v, e seus rótulos $L_u = (x_u, y_u)$ e $L_v = (x_v, y_v)$, u alcança v se e somente se $(x_u, y_u) < (x_v, y_v)$, isto é, se $x_u < x_v$ e $y_u < y_v$ (se $u = v$ então u alcança v). Assim, para a consulta $J \rightsquigarrow^? P$[3], basta comparar os rótulos de cada um, isto é, ao verificar que $9 < 15$ e $4 < 15$, sabe-se que J alcança P em tempo constante. Já para a consulta $E \rightsquigarrow^? O$, tem-se que E não alcança O, pois $(5, 9) \not< (14, 7)$, i.e., $5 < 14$, mas $9 \not< 7$ (Teorema 1).

2.1.1 Visualização de grafos baseada em dominância fraca

A abordagem de Kameda (e da representação do tipo *upward*) é restrita aos grafos planares. Esse algoritmo rotula os vértices com coordenadas no plano bidimensional, de forma que as relações de dominância entre as coordenadas dos vértices expressem diretamente a alcançabilidade entre eles. No exemplo anterior, dizemos que J domina P, pois $(9, 4) < (15, 15)$.

De acordo com as pesquisas realizadas, não há registros de métodos derivados da abordagem de Kameda para grafos não planares. Sendo assim, uma hipótese levantada pelo presente trabalho, é que o trabalho de Kameda pode ser estendido para grafos não planares por meio de um método on-line.

Autores da área de visualização de grafos publicaram um método de visualização que é aplicável a grafos não planares. Esse método, denominado *Weak dominance drawing* ([Kornaropoulos & Tollis, 2011, 2012]), mantém a restrição de Kameda, porém

[3] A notação $u \rightsquigarrow^? v$, definida no capítulo anterior, representa uma consulta de alcançabilidade entre os vértices u e v.

permite que as condições de dominância sejam relaxadas, criando **exceções**, i. e., uma relação de dominância não verdadeira entre alguns vértices (daí o termo *dominância fraca*). Isso permite uma visualização amigável aos humanos, compatível com a *upward*, ou seja, os vértices fontes e os sumidouros são posicionados em lados opostos (por exemplo, o grafo S_3^0 como mostrado na Figura 2.5). Ele não admite uma representação num plano $2D$ que seja livre de exceções [Eades et al., 1994; Kornaropoulos & Tollis, 2011]. Na figura, as coordenadas de u são menores do que as coordenadas de v, mas não há caminho entre esses dois vértices.

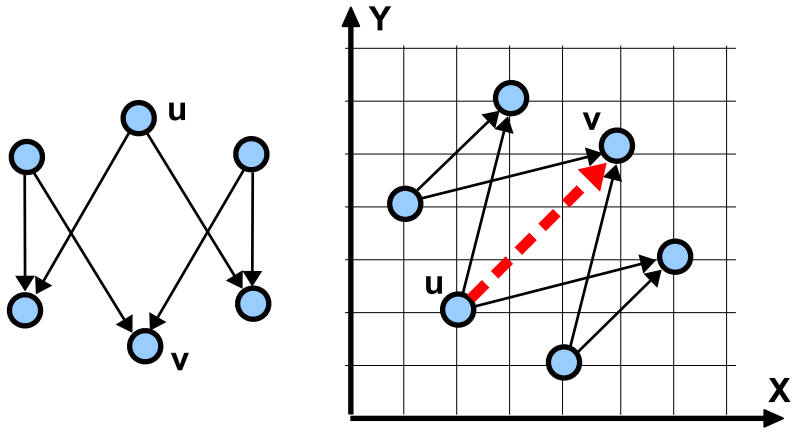


Figura 2.5. Exemplo de exceção entre u e v. A seta tracejada é uma exceção (ou, como em alguns textos, um *falsely implied path*), também chamado de *falso-positivo*.

O grande problema de utilizar o método de Kameda para indexar grafos não planares, é que, existindo um número exponencial de maneiras de se percorrer o grafo em profundidade [Knuth & Szwarcfiter, 1974], obter somente a árvore de cobertura à esquerda e à direita não oferece garantias de que o número de exceções seja mínimo. Infelizmente, o problema de criar uma representação de um DAG não planar, de forma a minimizar o número de exceções é conhecidamente NP-Difícil [Kornaropoulos & Tollis, 2011, 2012], o que faz com que os métodos existentes sejam heurísticos (no capítulo seguinte apresentar-se-á um desses métodos).

Para a abordagem de indexação proposta neste trabalho, estendemos o algoritmo de Kameda para ser aplicado a DAGs não planares. Para isso, utilizamos a ideia de dominância fraca obtida da área de visualização de grafos. Como será apresentado no capítulo seguinte, é possível com esse método responder uma grande parte das consultas em tempo constante, sendo necessária uma busca no grafo em alguns casos. Contudo, demonstra-se que essa busca é realizada geralmente em uma pequena parte

do grafo original, por meio de mecanismos eficientes de poda. Dado isso, a principal contribuição deste trabalho dedica-se a mostrar que a abordagem de Kameda [1975] pode ser generalizada para indexar grafos não-planares estáticos e dinâmicos, criando uma abordagem escalável, atual e com desempenho superior aos métodos que compõem o estado da arte em alcançabilidade.

2.2 Trabalhos relacionados

Apesar de todos os métodos publicados para grafos estáticos, Yildirim et al. [2010] mostrou que grafos muito grandes não são suportados pela maioria das abordagens existentes, i. e., esses grafos são indexáveis mas a computação de seus índices é inviável por diversos motivos, entre eles estão requisitos de memória quadrática no número de vértices e/ou arestas e tempo de indexação/consulta elevado. De fato, durante a nossa investigação, somente quatro métodos mostraram-se escaláveis podendo indexar grafos com mais de 100.000 arestas: Nuutila's Interval ([Nuutila, 1995; van Schaik & de Moor, 2011]), Grail ([Yildirim et al., 2010]), Ferrari ([Seufert et al., 2013]) e TF-Label ([Cheng et al., 2013]). A Tabela 2.1 mostra um sumário das complexidades de tempo e espaço para alguns desses métodos.

	Consulta	Construção	Tamanho do índice
Transitive closure	$O(1)$	$O(\|V\| * \|E\|)$	$O(\|V\|^2)$
Opt. Tree Cover	$O(\|V\|)$	$O(\|V\| * \|E\|)$	$O(\|V\|^2)$
Tree+SSPI	$O(\|E\| - \|V\|)$	$O(\|V\| + \|E\|)$	$O(\|V\| + \|E\|)$
GRIPP	$O(\|E\| - \|V\|)$	$O(\|V\| + \|E\|)$	$O(\|V\| + \|E\|)$
Dual labeling	$O(\|E\| - \|V\|)$	$O(\|V\| + \|E\| + t^3)$	$O(\|V\| + t^2)$
Path-tree	$O(\log^2 k)$	$O(\|E\| * k)$	$O(\|V\| * k)$
3-Hop	$O(\log \|V\| + k)$ ou $O(\|V\|)$	$O(k * \|V\|^2 \|Con(G)\|)$	$O(\|V\| * k)$
Nuutila's Interval	$O(\log_2 I)$	$O(\|V\| * \|E\|)$	$O(I)$
Grail	$O(d)$ até $O(\|V\| + \|E\|)$	$O(d(\|V\| + \|E\|))$	$O(d\|V\|)$
Ferrari	$O(\log d)$ até $O(\|V\| + \|E\|)$	$O(\|E\| * d^2)$	$O(d\|V\|)$

Tabela 2.1. Comparação entre as abordagens. $t = O(|E| - |V|)$ é o número de arestas fora da árvore de cobertura; k é o número de caminhos/cadeias; I é a quantidade de listas de intervalos; e d é o número de intervalos. Para simplificar a apresentação da tabela, exclusivamente nesta tabela, onde lê-se V entenda-se V_{DAG}, onde lê-se E entenda-se E_{DAG} e onde lê-se G entenda-se G_{DAG}.

2.2.1 Abordagens não escaláveis

Apresenta-se nesta seção os trabalhos relacionados ao tema de pesquisa e que são considerados como sendo abordagens não escaláveis, i. e., no contexto de grafos muito grandes os métodos a seguir não apresentam um bom desempenho de acordo com trabalhos recentes.

Como mostrado na Tabela 2.1, a abordagem *Transitive closure*, proposta por Simon [1988], refere-se ao algoritmo para computação do fecho transitivo de arestas de um DAG. O método pode construir um índice em tempo $O(|V_{\text{DAG}}| * |E_{\text{DAG}}|)$ e espaço $O(|V_{\text{DAG}}|^2)$ e responde a consultas em tempo constante. Nota-se que estes custos (de construção e tamanho do índice) são impraticáveis para grafos muito grandes, então outros estudos buscam reduzi-los, mantendo pequeno o custo para realizar as consultas. Por conseguinte, diferentes abordagens buscam explorar outras estruturas de dados, como árvores ou decomposição em caminhos (ou cadeias, que são conjuntos disjuntos de vértices interligados por um caminho), para computar e comprimir o fecho transitivo.

Métodos como os encontrados em [Chen & Chen, 2008; Jagadish, 1990] são baseados em decomposição em cadeias, onde, se um vértice alcança outro, então eles podem estar na mesma cadeia e mantém uma ordem de precedência entre si. Seguindo uma técnica diferente, Agrawal et al. [1989] propõe um índice gerado a partir de uma árvore de cobertura extraída do grafo, denominada *Optimum Tree Cover*. Esta abordagem sugere, como um primeiro e importante passo, encontrar a melhor árvore que maximiza a compressão do fecho transitivo. Para tanto, dada tal árvore, um intervalo é atribuído a cada vértice tal que, se um vértice u pode alcançar outro vértice v, então o intervalo de u contém o intervalo de v. Estes intervalos podem ser atribuídos por meio de caminhamentos em *pós-ordem* na árvore.

De forma semelhante, Tree+SSPI, de Chen et al. [2005], também usa uma árvore de cobertura e tem custo computacional de $O(|V_{\text{DAG}}|+|E_{\text{DAG}})$ para construir o índice de tamanho $O(|V_{\text{DAG}}| + |E_{\text{DAG}})$. No entanto, para evitar a ocorrência de falsos-negativos nas consultas, o método é associado a uma estrutura chamada SSPI (*Surrogate & Surplus Predecessor Index*). Outro método que também utiliza uma árvore de cobertura foi proposto por Trißl & Leser [2007] e é denominado GRIPP (*GRaph Indexing based on Pre and Postorder numbering*). No GRIPP, um intervalo é atribuído a cada vértice, mas alguns vértices alcançáveis por arestas que não aparecem na árvore são replicados para garantir a cobertura. Jin et al. [2008] propôs o *Path-Tree*, que extrai caminhos disjuntos (cadeias) de um DAG e então cria uma árvore na qual recebe uma rotulação por intervalos. Neste caso, o tempo para consultas e construção, além do tamanho do

índice, dependem do número de caminhos, expresso por k, encontrados no grafo.

Para Wang et al. [2006], a maioria dos grafos obtidos de aplicações do mundo real são esparsos. Isso significa que o número de arestas nesses grafos que não aparecem na árvore de cobertura (*spanning tree*) é pequeno, mas são importantes e podem auxiliar na construção de um índice mais robusto. Assim, seu trabalho propõe um método chamado *Dual-labeling*. Esse método considera o uso de um esquema de codificação para as arestas da árvore de cobertura e outra codificação para as arestas do grafo que não aparecem na árvore. Uma vez que o número de arestas que não estão na árvore é t, o método constrói um índice de tamanho $O(|V_{\mathrm{DAG}}| + t^2)$ em um tempo $O(|V_{\mathrm{DAG}}| + |E_{\mathrm{DAG}}| + t^3)$, com tempo de consulta de $O(|V_{\mathrm{DAG}}| * |E_{\mathrm{DAG}}|)$.

Os métodos da classe *Hop Labeling* (como [Cheng et al., 2006, 2008; Cohen et al., 2003; He et al., 2005; Schenkel et al., 2005]), mantém, para cada vértice e um certo *raio de alcance* (hops), um conjunto de outros vértices que podem alcançá-lo (antecessores) e outro conjunto de vértices que são alcançados (sucessores). Assim, dois vértices u e v são alcançáveis se a interseção entre os antecessores (conjunto L_{in}) de v e os sucessores (conjunto L_{out}) de u não é vazia. Se necessário, o processo de verificação pode ser repetido para os vértices intermediários em L_{in} e L_{out}. O método *3-Hop labeling*, proposto por Jin et al. [2009], tenta usar decomposições em cadeias combinadas com a estratégia de *Hop Labeling* para minimizar os conjuntos de sucessores e antecessores, gerando um índice menor. A ideia principal é, dada uma decomposição em cadeias extraída do grafo, uma representação concisa do fecho transitivo é primeiro derivada, a qual é chamada de contorno do grafo ($Con(G_{\mathrm{DAG}})$). Isso corresponde aos pontos de transição entre as cadeias encontradas. Esta técnica de indexação é análoga, como os autores dizem, a um mapa rodoviário. Dessa forma, para alcançar um destino (um vértice destino) a partir de uma dada origem, primeiro precisamos entrar na via apropriada (ou cadeia) e ir em direção à saída correta para encontrar o destino.

2.2.2 Abordagens escaláveis

2.2.2.1 NUUTILA's Interval

NUUTILA's Interval ([Nuutila, 1995; van Schaik & de Moor, 2011]) extrai o fecho transitivo do grafo e usa listas de intervalos para comprimir os segmentos de vértices consecutivos. Por exemplo, se o fecho transitivo de um determinado vértice u for o conjunto $v_1, v_2, v_3, v_4, v_6, v_7, v_8, v_9, v_{11}, v_{12}$, o seu conjunto de alcançabilidade será comprimido em três listas de intervalos: $[v_1, v_4]$, $[v_6, v_9]$ e $[v_{11}, v_{12}]$. Dessa forma, para constatar se v_1 alcança v_3, por exemplo, basta verificar os limites do primeiro intervalo

$[v_1, v_4]$, onde este pode ser encontrado em tempo logarítmico já que a lista de intervalos está ordenada.

O método proposto por Nuutila [1995] e depois revisto por van Schaik & de Moor [2011] utiliza, nessa proposta mais recente, uma estrutura de *bit-vectors* para representar cada vértice nos intervalos, onde podem ser computados por operações lógicas OU. Por exemplo, o algoritmo a seguir pode ser usado para computar o fecho transitivo de um DAG:

Algoritmo 1: Intervalos Alcançáveis

Dados: $V_{\text{DAG}}, E_{\text{DAG}}$
1 **início**
2 $[v_1, v_2, ..., v_n] \leftarrow$ OrdenaçãoTopológica(V_{DAG}, E_{DAG});
3 **para** $i \leftarrow n$ *decrementando* **até** *1* **faça**
4 alcançavel$[v_i] \leftarrow \bigcup\{\text{alcançavel}[v_j] \cup \{v_j\} | (v_i, v_j) \in E_{\text{DAG}}\}$;

Dessa forma, esse método implementa um esquema de compressão que facilita o uso das operações lógicas OU, agilizando a construção do fecho transitivo. Esse esquema utilizado é denominado PWAH (Partitioned Word Aligned Hybrid compression scheme) e pode ser melhor entendido nos trabalhos publicados por Wu et al. [2006] e van Schaik & de Moor [2011].

2.2.2.2 GRAIL

Proposto por Yildirim et al. [2010], o método de busca *on-line* denominado GRAIL (*Graph Reachability indexing via rAndomized Interval Labeling*) atribui a cada vértice u um rótulo $L_u = [r_x, r_u]$, onde r_u e r_x denotam valores obtidos de um caminhamento em pós-ordem no grafo. A intuição é que um vértice u alcança outro vértice v se $L_v \subseteq L_u$. Contudo, esse tipo de rotulação não garante que exceções não ocorram. Uma exceção é a ocorrência de resultado com falso-positivo, que é quando $L_v \subseteq L_u$ mas u não alcança v.

No GRAIL, o índice é formado por múltiplos intervalos obtidos por meio de caminhamentos em pós-ordem no DAG, seguindo a estratégia *min-post*. *Min-post* atribui um intervalo único $L_u = [s_u, e_u]$ a cada vértice u, onde s_u e e_u denotam o início e o fim do intervalo. O valor e_u é definido como $e_u = post(u)$, que é o rank em pós-ordem do vértice u, e o valor s_u é definido como $s_u = min\{s_x | x \in children(u)\}$ se u não é uma folha e $s_u = e_u$, se u é uma folha (i. e., um vértice sem sucessores). Grail utiliza ainda algumas otimizações para acelerar os tempos das consultas, como o chamado *positive-cut filter* e *level filter* (essas otimizações também são utilizadas pelo método proposto

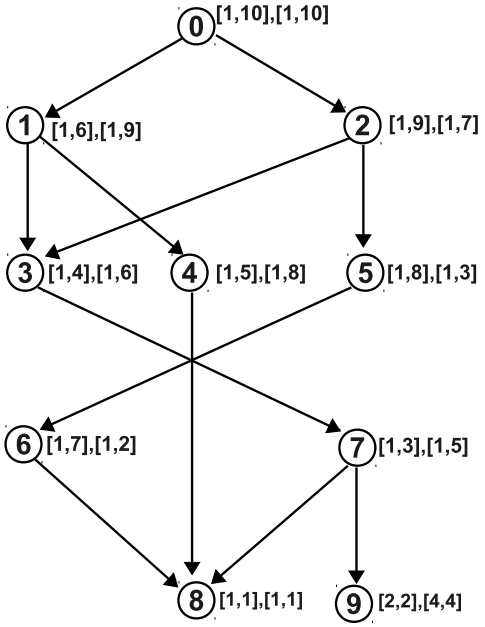


Figura 2.6. DAG com múltiplos intervalos [Yildirim et al., 2010].

neste trabalho e serão explicadas no próximo capítulo). O Algoritmo 2 apresenta um pseudo-código para *min-post*.

Algoritmo 2: MinPost

```
1  MinPostLabeling(G_DAG)
2  início
3      r ← 1
4      Roots ← {n : n ∈ roots(G_DAG)}
5      para cada x ∈ Root, em ordem aleatória faça
6          Call MinPostVisit(x, G_DAG)
7  MinPostVisit(x, G_DAG) início
8      se x foi visitado antes então
9          retorna
10     para cada y ∈ Children(x), em ordem aleatória faça
11         Call MinPostVisit(y, G_DAG)
12     r_c ← min{s_c : c ∈ Children(x)}
13     L_x ← [min(r, r_c), r]
14     r ← r + 1
```

A Figura 2.6, mostra um DAG depois da rotulação com dois intervalos. Com esses rótulos, para saber se o vértice 2 (em $[1,9],[1,7]$) alcança o vértice 4 (em $[1,5],[1,8]$), i. e., se 2 alcança 4, deve-se verificar se $[1,5] \subseteq [1,9]$ e $[1,8] \subseteq [1,7]$, que é falso,

significando que o vértice 2 não alcança o vértice 4, pois $[1,5] \subseteq [1,9]$ (i.e., o intervalo $[1,5]$ está contido no intervalo $[1,9]$), mas $[1,8] \not\subseteq [1,7]$. Da mesma forma, para verificar se o vértice 2 alcança o vértice 9 (em $[2,2],[4,4]$), precisamos verificar se $[2,2]] \subseteq [1,9]$ e $[4,4] \subseteq [1,7]$, que é verdadeiro. Contudo, não é possível saber de antemão se esse resultado é verdadeiro ou é um falso-positivo. Um exemplo disso é a consulta $4 \rightsquigarrow^{?} 3$, que tem resposta positiva mas o vértice 4 não alcança o vértice 3. Nesses casos, uma busca em profundidade no grafo é necessária, usando como esquema de poda e direcionamento da busca, explora somente os vértices que estão contidos no intervalo do vértice de início (no último exemplo, o vértice 4). É importante observar que GRAIL usa múltiplos intervalos (d intervalos) de forma a reduzir o número de exceções. Nesse caso, quanto mais intervalos, menores as chances de que ocorram exceções.

2.2.2.3 FERRARI

Muito similar ao GRAIL, FERRARI (*Flexible and Efficient Reachability Range Assignment for gRaph Indexind*) [Seufert et al., 2013] também utiliza múltiplos intervalos obtidos por meio de aplicações do algoritmo *min-post*. No entanto, FERRARI utiliza uma compressão seletiva do conjunto de intervalos, onde um subconjunto de intervalos adjacentes são agrupados, tornando mais ágil a verificação de alcançabilidade. Essa operação de agrupamento de intervalos é diferente da realizada pelo *NUUTILA's Interval* e tenta preencher as lacunas entre os intervalos atribuídos. Como o agrupamento é feito para cada aresta, então cada vértice u receberá pelo menos $deg(u) \times d$ intervalos (usando d intervalos). Assim o custo computacional para atribuir os intervalos é $O(\sum_v deg(u) \times d^2) = O(|E_{\mathrm{DAG}}| \times d^2)$, e, como os intervalos não agrupados são mantidos em ordem, o tempo de consulta é $O(\log_2 d \times (|V_{\mathrm{DAG}}| + |E_{\mathrm{DAG}}|)$.

FERRARI também atribui outras duas informações a cada vértice, que é um intervalo dito *exato* proveniente da árvore de cobertura extraído do DAG e a posição do vértice em uma ordenação topológica. Ambos são usados para resolver casos de exceções e poda do espaço de busca, onde todos os vértices que estão posicionados além do vértice objetivo na ordenação topológica podem ser descartados.

2.2.2.4 TF-Label

O trabalho recente de Cheng et al. [2013] propõe a abordagem denominada TF-Label (onde TF significa *Topological Folding*). Essa nova abordagem compacta o grafo de forma recursiva objetivando reduzir o índice final, para isso é utilizada uma técnica de rotulação de vértices similar às tradicionais abordagens da classe *Hop Labeling*.

TF-Label atribui os conjuntos $labels_{out}(u)$ e $labels_{in}(u)$ para cada vértice u do DAG de entrada, obtidos de um passo chamado de *topological folding* (i. e., dobra topológica, numa tradução livre). Para uma dada consulta $u \rightsquigarrow^{?} v$, o método necessita apenas verificar a interseção entre $labels_{out}(u)$ e $labels_{in}(v)$ para aferir a alcançabilidade entre u e v. Se um elemento em comum é encontrado na interseção, então u alcança v, caso contrário, u não alcança v.

O índice produzido por essa abordagem é limitado em tamanho pela quantidade de rótulos aplicados e o tempo de consulta é limitado pela operação de interseção. Contudo, o tempo de construção não é tão óbvio. Simplificando um pouco, mas sem perder de vista os detalhes, a complexidade é da ordem de $O(\Sigma_{1 \leq i \leq lgL(G_{\mathrm{DAG}})} \Sigma_{v \in V_{\mathrm{DAG}}(F_i(G_{\mathrm{DAG}}))} (deg_{in}(v, F_i(G_{\mathrm{DAG}})) * deg_{out}(v, F_i(G_{\mathrm{DAG}}))))$, onde $L(G_{\mathrm{DAG}})$ é um função que obtém o número de níveis topológicos do grafo de entrada G_{DAG}, $F_i(G_{\mathrm{DAG}})$ é a i-ésima dobra topológica de G_{DAG} e $deg_{in}(v, F_i(G_{\mathrm{DAG}}))$ é o grau de entrada do vértice na i-ésima dobra.

2.2.2.5 SCARAB Framework

Um trabalho recente publicado por Jin et al. [2012] apresenta um *framework* unificado para alcançabilidade denominado SCARAB (*standing for SCAlable ReachABility*). Esse framework possibilita acelerar (*boosting*) os métodos existentes e permite que métodos não escaláveis possam trabalhar com grafos grandes.

SCARAB propõe o uso de um grafo auxiliar extraído do grafo original, contendo o que se chama de "*backbone de alcançabilidade*" (ou estrutura de alcançabilidade). Esse grafo auxiliar em geral é menor e condensa toda a informação de alcançabilidade entre os principais vértices do grafo, isto é, aqueles vértices intermediários que fazem a conexão entre os demais. Isso significa que o backbone contem as principais vias de acesso entre todos os vértices do grafo, para determinar se existe caminho entre quaisquer dois vértices, basta verificar se os vértices estão conectados ao backbone. Em seguida, uma busca no backbone é suficiente para dizer se eles se alcançam. A Figura 2.7 mostra um exemplo onde o grafo original está à esquerda e o backbone à direita.

Como o grafo auxiliar é em geral menor, é possível acelerar métodos de busca on-line, como o proposto neste trabalho. Como observado por Jin et al. [2012], tais métodos alcançam melhor desempenho quando combinados com SCARAB. Embora os ganhos de desempenho para responder consultas de alcançabilidade aumentem tanto o tamanho do índice e quanto o tempo para construí-lo.

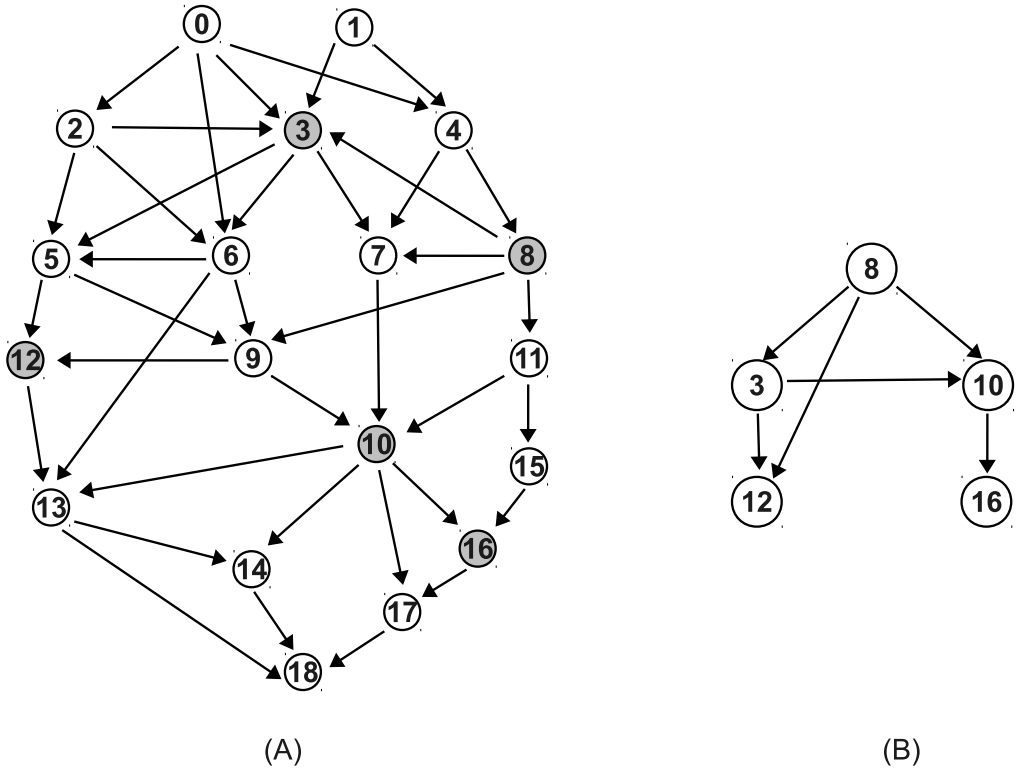


Figura 2.7. Exemplo: (A) grafo original; (B) Backbone de alcançabilidade. Figura coletada de Jin et al. [2012].

2.3 Alcançabilidade em Grafos Dinâmicos

O problema de indexação de grafos torna-se ainda mais complexo se esses grafos sofrerem atualizações com o passar do tempo, como acontece numa rede social, por exemplo. Numa rede social, a cada instante novos relacionamentos entre os participantes são estabelecidos, bem como a inserção de novos membros ou a saída de outros. Isso claramente faz com que a estrutura dos grafos mude.

Nesse contexto, a indexação estática não é adequada, pois para cada atualização uma nova e completa indexação deve ser realizada. Outros exemplos de grafos dinâmicos são encontrados em Compiladores, onde grafos de dependências são extraídos das instruções do programa para análise de fluxo. Na linguagem Prolog, a execução do programa depende de uma busca em profundidade na árvore de execução (construída dinamicamente) para avaliar uma consulta e respondê-la. Nesse contexto, implementações paralelas de Prolog se beneficiam de consultas de alcançabilidade [Gupta & Jayaraman, 1993].

Uma atualização no grafo é uma operação que insere ou remove arestas ou vértices

[Mehta & Sahni, 2004]. Sendo assim, entende-se por *grafo dinâmico* um grafo que está sujeito a uma sequência de atualizações. Por conseguinte, o objetivo de um algoritmo de indexação para tais grafos é atualizar eficientemente o índice após mudanças dinâmicas no grafo, ao invés de reconstruí-lo a todo instante [Yildirim et al., 2013]. Por exemplo, se o custo para indexar um grafo é $O(C)$, após cada uma de m modificações no grafo, uma solução dinâmica deve ter custo menor que $O(mC)$ e só será eficiente se puder amortizar o custo de atualizar o índice para um número grande de consultas.

Uma classificação de problemas de grafos dinâmicos pode ser encontrada em [Mehta & Sahni, 2004] e pode ser de dois tipos, de acordo com as atualizações permitidas. Um grafo dinâmico é dito ser *totalmente dinâmico* se as operações de atualização incluem a inserção e remoção de vértices e arestas. Um grafo *parcialmente dinâmico* permite somente um tipo de atualização: inserção (problema incremental) ou remoção (problema decremental).

Historicamente, a comunidade científica pouco tem se dedicado ao problema de alcançabilidade em grafos dinâmicos. Acerca do tema, há registros de soluções não escaláveis para a manutenção do fecho transitivo em meio a atualizações no grafo, como os trabalhos de King [1999]; Frigioni et al. [2001]; Demetrescu & Italiano [2005] e Roditty [2008]. Outro trabalho dedicado a consultas de alcançabilidade em grafos dinâmicos é o de Roditty & Zwick [2004]. Em pesquisas realizadas, um importante método escalável para o problema de alcançabilidade foi encontrado: o trabalho de Yildirim et al. [2013], denominado Dagger.

2.3.1 Dagger

O Dagger (para *Dynamic Graph REAchability*, com um 'G' extra) [Yildirim et al., 2013], é a versão dinâmica do GRAIL. Ele gera um índice com tamanho e tempo de construção lineares (na ordem do grafo), além de um tempo de consulta proporcional à versão estática (i.e., para grafos estáticos ou que não sofrem atualizações ao decorrer do tempo). O Dagger consegue atualizar o índice em tempo linear no tamanho do grafo de entrada e em tempo constante em alguns casos específicos.

Assim como no GRAIL, os componentes fortemente conectados são identificados e o DAG associado é extraído e indexado. Dagger indexa o DAG, mas mantém o grafo original, porque as atualizações podem afetar os componentes, criando novos ou separando os existentes. A Figura 2.8, extraída de [Yildirim et al., 2013], apresenta um grafo direcionado e seu respectivo DAG (após a identificação dos componentes fortemente conectados). Segundo Yildirim et al. [2013], toda atualização deve ser feita no grafo original, pois essas atualizações podem afetar a estrutura do DAG que será

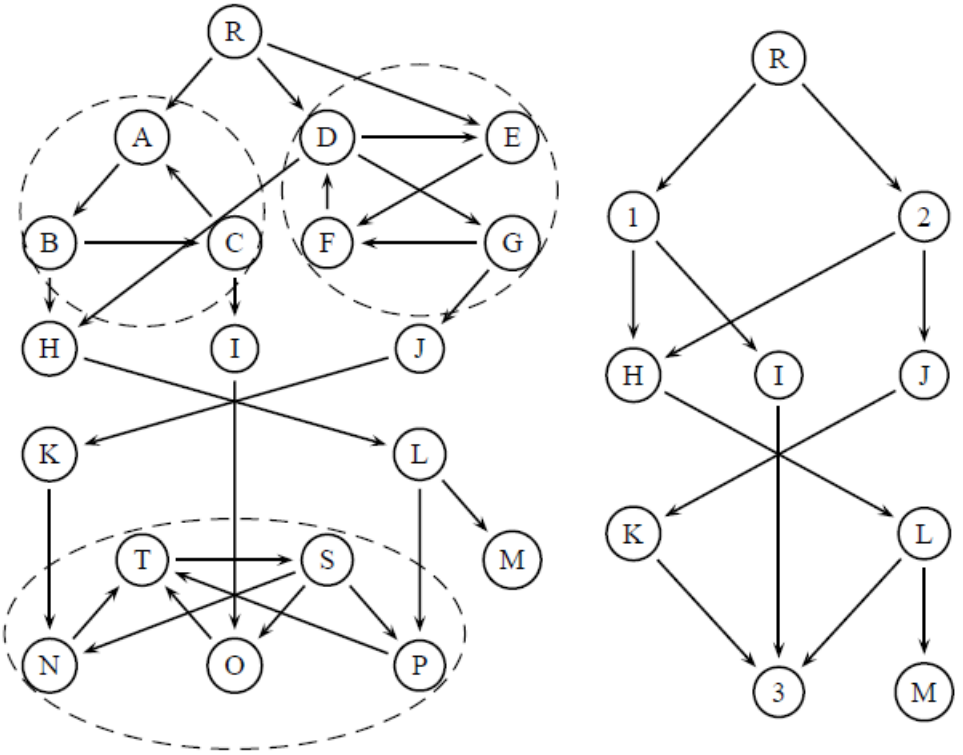


Figura 2.8. À esquerda o grafo direcionado de entrada e, à direita, o respectivo DAG

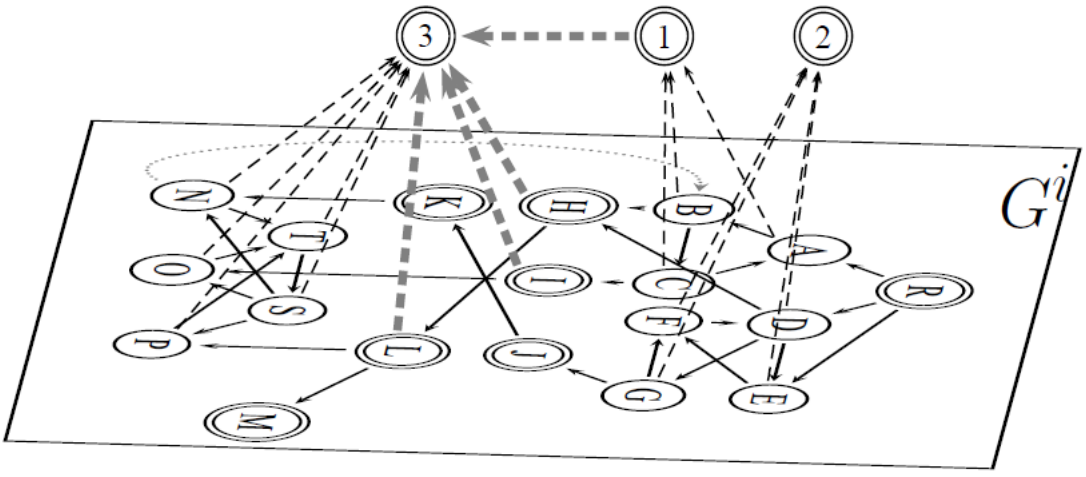


Figura 2.9. Exemplo: estrutura para grafo dinâmico.

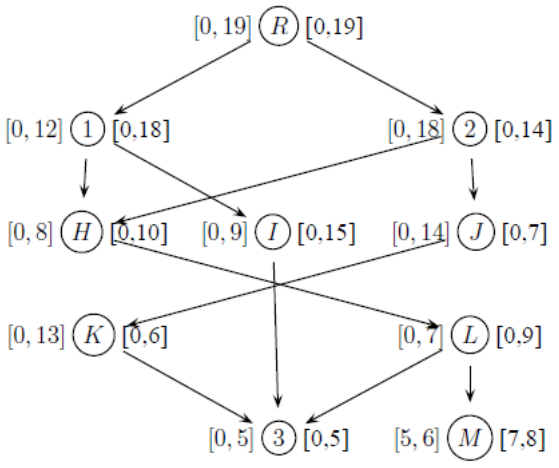


Figura 2.10. Rotulação de um DAG usando o *min-post modificado*. Os intervalos à esquerda e direita de cada vértice é resultado de uma travessia diferente.

indexado.

Dagger gerencia o grafo dinâmico por meio de uma estrutura contendo ambos o grafo normal (direcionado) e o DAG indexado. A Figura 2.9 apresenta a estrutura utilizada. Nela, os vértices desenhados com linhas duplas representam os componentes fortemente conectados. Os vértices acima do plano são componentes maiores e as setas tracejadas (mais finas) representam o mapeamento entre vértices do grafo original e os componentes.

O índice é construído pela rotulação dos vértices do DAG de maneira bem similar ao GRAIL, usando múltiplos intervalos, com uma modificação do algoritmo *min-post*. Para cada caminhamento (em pós-ordem), o algoritmo inicia da(s) raiz(es) do DAG e rotula cada vértice. Durante o caminhamento, um contador é mantido e incrementado com o tamanho do respectivo componente fortemente conectado, pois o intervalo atribuído deve ser suficientemente grande para conter alterações que possam ocorrer dentro do componente. A Figura 2.10 mostra esse esquema de rotulação com dois intervalos (figura extraída de [Yildirim et al., 2013]), onde, por exemplo, o vértice 3 que representa um componente fortemente conectado (vide Figura 2.8) contendo 5 vértices do grafo original, por isso o seu rótulo possui intervalos de 0 a 5.

Dagger suporta operações de inserção e remoção de vértices e arestas, mas algumas operações são menos custosas do que outras. Por exemplo, na inserção de uma nova aresta (u, v) no grafo original, caso o intervalo de v já estiver contido no intervalo de u, e nenhum novo componente for criado, nenhuma atualização precisa ser feita no

índice. Senão, será necessário aumentar a folga do intervalo de u de forma proporcional e propagar esse aumento para os vértices que antecedem u, recursivamente. No entanto, se a inserção/remoção de arestas (e/ou vértices) resultar na geração de novos componentes fortemente conectados, a operação é custosa, pois envolve recalcular alguns intervalos.

O custo de atualização do índice no caso de uma inserção de aresta é, no pior caso, proporcional a uma consulta de alcançabilidade e atualização dos rótulos dos vértices, isto é $O(d*|E_{\mathrm{DAG}}|)$. Uma inserção de vértice pode ser acompanhada de várias inserções de arestas, sendo assim o tempo para inserir um vértice (acompanhado de sua lista de arestas de entrada l_e e sua lista de arestas de saída l_s) demanda $O(d * |l_s| + |l_e| * C_{ai})$, onde C_{ai} é o tempo gasto para inserir uma aresta e d é a quantidade de intervalos usados, sendo necessário atualizar os índices dos vértices sucessores.

O custo para remover uma aresta é constante, se os vértices conectados estiverem em componentes diferentes. Caso contrário, para remover uma aresta (u, v), o custo é $O(|Ec'| - |Ec''| + |Vc''|)$ onde $|Ec'|$ é a quantidade de arestas do componente antes de ser separado, já $|Ec''|$ e $|Vc''|$ são as quantidades de arestas e vértices do componente que contém o vértice u depois de separado. Para remover um vértice, primeiro devem ser removidas todas as arestas de entrada e saída, a um custo de $O(|l_s| * C_{ad} + |l_e|)$, onde l_s é a quantidade de arestas de saída, l_e a quantidade de arestas de entrada e C_{ad} o custo de remover uma aresta. Se a remoção da aresta não separar o componente, ou seja, se mesmo depois da remoção da aresta houver caminho no grafo entre u e v, o custo para remover a aresta é constante.

O trabalho de Yildirim et al. [2013] apresenta duas questões ao tratarmos de grafos dinâmicos: primeiro, o custo de atualização do índice deve ser muito menor do que reconstruir o índice inteiro a cada atualização, o que justifica o uso do método; segundo, há a necessidade de mantermos o dígrafo original em memória, uma vez que toda alteração é feita nele. Portanto, a abordagem deve fazer uso de estruturas de dados mais elaboradas a fim de conter o dígrafo e seu DAG (a ser, de fato, indexado). Pensando nisso, Dagger escala bem, mas seu desempenho é afetado pelas estruturas de dados complexas e redundantes. Nesta tese, mostramos que nosso método é melhor do que o Dagger quanto ao tempo para atualizar o índice e quanto ao desempenho das consultas, que chegam a ser dezenas de vezes mais rápidas, em média. Nosso método também considera todas as possíveis atualizações em um dígrafo que não precisa ser acíclico.

2.3.2 Incremental 2-HOP

Incremental 2-HOP [Bramandia et al., 2010] deriva do 2-HOP para grafos estáticos, onde, para um DAG (V_{DAG}, E_{DAG}) cada vértice $v \in V_{DAG}$ é associado a um rótulo L, que compreende duas listas de vértices $L_{in}(v)$ e $L_{out}(v)$. Estas duas listas são chamadas de rótulos 2-HOP. Os vértices que são armazenados em $L_{in}(v)$ (e respectivamente em $L_{out}(v)$) são vértices que podem alcançar (respectivamente, são alcançados por) v. Assim, para todos os vértices u e v, $L_{in}(v) \cap L_{out}(u) \neq \emptyset$ se e somente se u alcança v. Dessa forma, se existe um vértice w tal que ele está em ambos $L_{in}(v)$ e $L_{out}(u)$, então u pode alcançar w, e w, por sua vez, pode alcançar v. Em seus estudos, Bramandia et al. [2010] melhoram o índice tradicional 2-HOP, que atualizam o índice quando um novo vértice é inserido. Eles propõem algumas funções heurísticas para encontrar os rótulos com base em uma propriedade de separação de vértices, na qual um conjunto de vértices X separa u e v se a remoção de X desconecta u e v. Logo, em todo caminho p que liga u à v haverá pelo menos um vértice em p escolhido como a interseção entre u e v. De acordo com os autores, suas heurísticas escolhem os vértices que aparecem no maior número de caminhos entre u e v (onde $u\ E^*_{DAG}\ v$), e então remove os vértices escolhidos e suas arestas do grafo. Tal remoção indica que todos os caminhos que passam por esses vértices foram processados. Na prática, o algoritmo que constrói o índice encontra esses vértices de forma gulosa. Assim, dado um conjunto de ancestrais de u, $A(u)$, e um conjunto de decendentes $D(u)$, ele adiciona u à L_{out} de $A(u)$ e L_{in} de $D(u)$. Uma vez que u é escolhido, todos os caminhos de $A(u)$ até $D(u)$ via u não são mais relevantes para construir o índice. O algoritmo remove u e suas arestas do grafo e procede para a próxima iteração. Ele termina quando não há mais arestas restantes no grafo. A complexidade de tempo desse método é $O(|V_{DAG}|^2 \times (|V_{DAG}| + |E_{DAG}|))$.

Dagger escala melhor do que Incremental 2-HOP e é aplicável também à dígrafos cíclicos. Contudo, o desempenho de Dagger é afetado por suas complexas estruturas de dados e o desempenho de suas consultas não é significantemente melhor do que uma busca em largura (BFS). Por outro lado, Incremental 2-HOP suporta apenas DAGs e realiza $O(|V|)$ atualizações nos rótulos 2-HOP em cada atualização do índice, i. e., uma complexidade de tempo pior do que Dagger. Esta tese mostra que Feline-PK é mais rápido do que Dagger na atualização do índice, onde esse índice é melhor que o gerado pelo Dagger: as consultas de alcançabilidade são respondidas de forma 10 vezes mais rápidas, em média. Assim como o Dagger, Feline-PK considera todas as possíveis atualizações no dígrafo, que pode ser cíclico. Incremental 2-HOP não suporta todas as operações de inserção e remoção de vértices e arestas. Não é possível com este algoritmo indexar um grafo que começa sem vértices e arestas ($G = (\emptyset, \emptyset)$) e incrementalmente vai

sendo atualizado com novos conjuntos de vértices e arestas. Neste caso, a indexação deve iniciar de um grafo não vazio e somente após o índice 2-HOP ser construído. Incremental 2-HOP não consegue atualizar o índice dada um remoção de arestas, uma vez que o grafo é desconstruido na indexação. Portanto, como este método é deficiente e ineficiente, não é considerado nos experimentos desta tese.

2.4 Conclusão

Apesar da grande quantidade de trabalhos publicados na área de alcançabilidade em grafos, a maioria das abordagens não são escaláveis. Por conseguinte, há uma necessidade de novas abordagens ou estudos que aprimorem abordagens antigas, priorizando não somente a questão da escalabilidade, mas também as modificações nas estruturas dos grafos ao longo do tempo.

No capítulo seguinte, apresentamos uma abordagem para indexação de grafos estáticos com melhor complexidade teórica do que os métodos relacionados aqui, além de ser mais simples de implementar e com melhor desempenho na prática. Em seguida, pensando na questão das modificações dinâmicas dos grafos, apresenta-se em capítulos posteriores uma proposta que contemple um índice que suporte tais modificações.

Capítulo 3

Alcançabilidade em Grafos Estáticos: Feline

Apresenta-se neste capítulo o método Feline (*Fast rEfined onLINE search*) para a construção de índices de alcançabilidade. Toda a discussão sobre a construção do índice e realização de consultas dar-se-á no contexto de métodos de busca online, onde o grafo original faz-se necessário em memória. Para tanto, este capítulo está organizado como a seguir:

- Após uma breve introdução, a Seção 3.2 apresenta as definições e propriedades do índice Feline, que é complementada logo em seguida pela Seção 3.3, descrevendo como o índice pode ser interpretado graficamente.

- A Seção 3.4 mostra como o índice de alcançabilidade é obtido, apresentando o algoritmo de construção.

- A Seção 3.5 introduz o algoritmo para realizar as consultas, discute a intuição a respeito da eficiência do método e apresenta a complexidade computacional do Feline.

- A Seção 3.6 apresenta duas otimizações implementadas.

- Os experimentos realizados e os resultados obtidos são discutidos na Seção 3.7.

- Por fim, a Seção 3.8 conclui este capítulo.

3.1 Introdução

Em geral, construir um índice de alcançabilidade para um dígrafo estático pode ser decomposto em: (1) agrupar os componentes fortemente conectados e (2) construir um índice que ajude a responder consultas de alcançabilidade no DAG resultante. O método de indexação que é apresentado neste capítulo, derivado da abordagem de Kameda [1975], foi definido a partir da observação de uma propriedade da ordenação topológica sobre um DAG $G_{\text{DAG}} = (V_{\text{DAG}}, E_{\text{DAG}})$ qualquer: dados dois vértices u e v em V_{DAG}, se v ocorre antes de u na ordenação, pode-se dizer com certeza que u não alcança v em G_{DAG}. Contudo, se v ocorre depois de u, nada pode-se afirmar a respeito da alcançabilidade entre u e v. Obviamente, se v ocorre depois de u, podemos determinar a alcançabilidade por uma busca (e. g., em profundidade) a partir de u no grafo, seguindo a orientação das arestas. Dessa forma, se existir um caminho entre u e v, a busca terá uma resposta positiva ao teste de alcançabilidade. Por exemplo, dado o DAG da Figura 3.1 e a ordenação topológica mostrada, podemos afirmar que o vértice h não alcança o vértice a, pois a vem antes de h na ordenação.

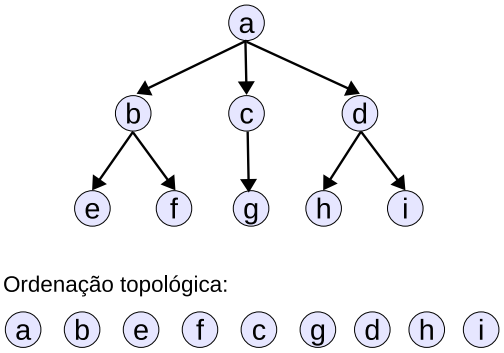


Figura 3.1. DAG e uma ordenação topológica.

A partir dessa observação, é razoável pensar que é possível reduzir o trabalho de geração de índice para um método de busca on-line, por meio de uma simples ordenação topológica do grafo com tempo $O(|V_{\text{DAG}}| + |E_{\text{DAG}}|)$. Dado um DAG $(V_{\text{DAG}}, E_{\text{DAG}})$, uma ordenação topológica $\prec$ é uma enumeração de V_{DAG} tal que para todo par $(u, v) \in E_{\text{DAG}}, u \prec v$ (ordem total). Uma consequência dessa definição é que, dado um DAG $(V_{\text{DAG}}, E_{\text{DAG}})$, sua ordenação topológica $\prec$ e dois vértices $(u, v) \in V^2_{\text{DAG}}, u\ E^*_{\text{DAG}}\ v \Rightarrow u = v$ ou existe $(u, k) \in E_{\text{DAG}}$ tal que $k \prec_= v$ e $k\ E^*_{\text{DAG}}\ v$. Isso significa que todos os caminhos partindo de u até v (se existir algum) passam pelos vértices que estão entre u e v seguindo a ordenação topológica. Contudo, a ordenação topológica não é única e, como consequência, na presença de muitas ordenações topológicas de um dado DAG,

precisamos considerar os vértices entre u e v na interseção de *todas* essas ordenações[1]. Se essa interseção é vazia, então podemos responder de forma negativa à consulta (sem a necessidade de percorrer o DAG). Isso acontece, por exemplo, quando v está antes de u em pelo menos uma das ordenações topológicas.

Por exemplo, a Figura 3.2 apresenta duas ordenações topológicas possíveis para o DAG anterior: uma priorizando os vértices mais à esquerda da raíz, outra priorizando os vértices mais à direita da raíz em uma busca em profundidade. Isto é, na ordenação (I), após o vértice a, o vértice b é visitado; já na ordenação (II), depois do vértice a, visita-se primeiro o vértice d.

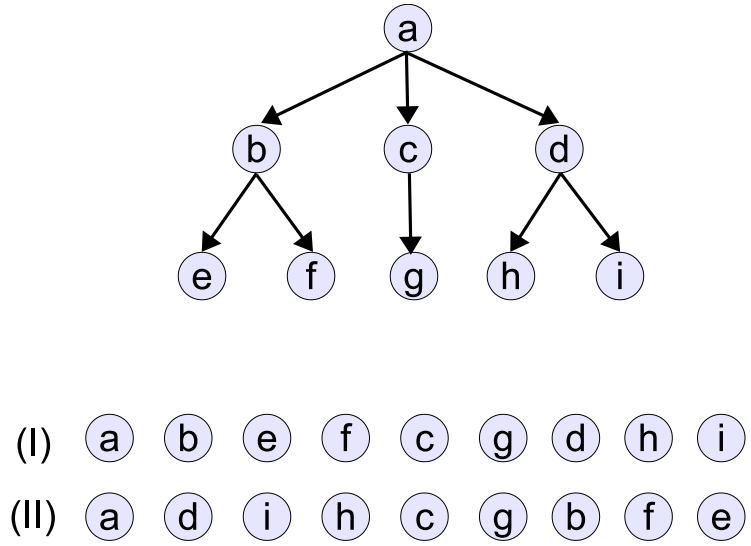


Figura 3.2. Dag e ordenações topológicas.

No caso da consulta $d \rightsquigarrow^{?} g$, considerando a ordenação (I), podemos dizer rapidamente que a resposta é negativa, pois g aparece antes de d, indicando que não pode haver um caminho entre os dois vértices iniciando em d. No entanto, considerando a ordenação (II), seria necessária uma busca no grafo para confirmar que não há caminho entre d e g, pois d aparece antes de g. Contudo, conforme Kameda [1975], essas duas ordenações topológicas são "complementares" e a resposta negativa em (I) é suficiente para responder de forma negativa a essa consulta.

A ideia principal do método elucidado neste capítulo é que a utilização de mais de uma ordenação topológica pode resultar em um mecanismo de poda mais eficiente do espaço de busca, pois é possível descartar vértices que seriam percorridos pela

[1] Conhecidamente, há várias ordenações topológicas possíveis a um mesmo DAG ([Knuth & Szwarcfiter, 1974; Kalvin & Varol, 1983]), e cada ordenação atribui *rankings* (i. e., um valor inteiro entre 0 e $|V_{\text{DAG}}|$) diferentes aos vértices.

busca apenas verificando a sua posição (ou *ranking*) em cada ordenação. Por exemplo, considerando o grafo da Figura 3.2, veja o caso da consulta $a \rightsquigarrow^{?} g$. Como as duas ordenações topológicas (I e II) indicam que a ocorre antes de g, uma busca no grafo é inevitável, de forma a confirmar a evidência de que a alcança g (i.e., $aE^{*}_{\mathrm{DAG}}g$). Contudo, pode-se reduzir a busca apenas descartando os vértices que nunca alcançam g em cada ordenação, o que pode diminuir o tempo de busca em muitos casos. Isso pode ser visto no exemplo da Figura 3.3, onde todos os vértices que aparecem depois de g em cada ordenação são removidos da busca (vide hachura nas ordens I e II), limitando a quantidade de arestas a serem percorridas (hachuras no grafo) e direcionando a busca.

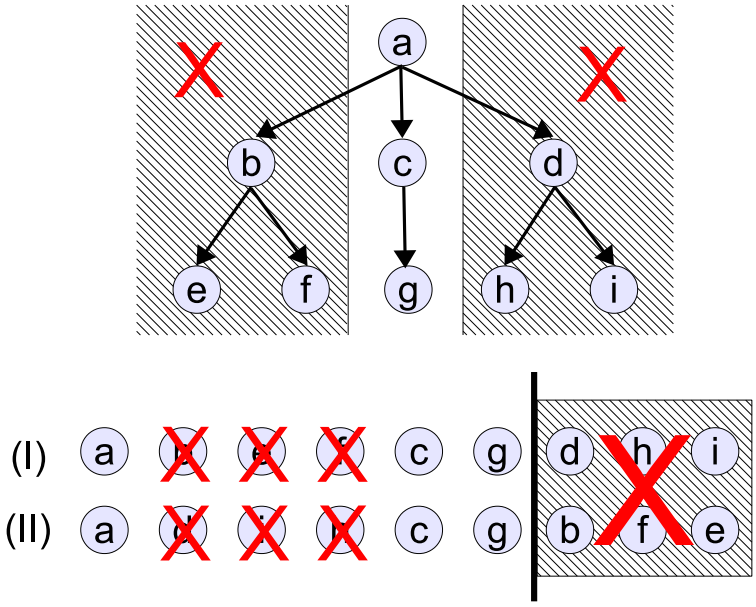


Figura 3.3. Dag e ordenações topológicas: Todos os vértices encontrados depois de g nas duas ordenações são descartados da busca (área hachurada), limitando a quantidade de arestas a serem visitadas. Na Figura, as áreas hachuradas no DAG representam o vértices descartados.

Como comentado anteriormente, as ordenações topológicas devem ser *complementares*. Assim, ao responder se $u\ E^{*}_{\mathrm{DAG}}\ v$ (para dois vértices arbitrários $u, v \in V^{2}_{\mathrm{DAG}}$) a interseção deve ser tão pequena quanto possível (o menor número de vértices que são candidatos a estarem no caminho entre os dois vértices), acelerando a busca. Essa mesma ideia é encontrada na área de *Graph Drawing* e é denominada *Weak Dominance Drawing* [Battista et al., 1998, 1992; Eades et al., 1994; Kornaropoulos & Tollis, 2011]. Uma *Weak Dominance Drawing*, definida por Kornaropoulos *et al.*, cria uma representação de um DAG não (necessariamente) planar em um plano bidimensional, relaxando as propriedades de dominância, i.e., permitindo algumas exceções quando

há arestas que se cruzam. Nele, cada vértice u recebe uma coordenada única no plano $\mathbb{N}^2$ por meio de duas ordenações topológicas (i.e., um par de inteiros $(\prec_x^u, \prec_y^u)$), preservando as relações de alcance entre a maioria dos pares de vértices. Embora esse método se dedique exclusivamente à visualização de grafos, não há relação publicada com o trabalho de Kameda [1975] e os seus algoritmos podem ser reaproveitados para gerar um índice de alcançabilidade on-line.

O análogo da relação de alcançabilidade na representação gráfica é a relação "está no canto superior direito". Mais precisamente, ambas as coordenadas de um vértice v, alcançáveis a partir de outro vértice u, devem ser maiores que as respectivas coordenadas de u, necessariamente. Quando essa relação geométrica é verificada, a alcançabilidade é decidida por uma busca na parte reduzida do grafo, identificada a partir do índice. Quando não é verificada, a não-alcançabilidade é decidida em tempo constante.

A utilização do algoritmo proveniente da área de visualização de grafos, além de representar uma interseção interessante entre áreas da Ciência da Computação, mostra que o trabalho original de Kameda [1975] para grafos planares, publicado há décadas, pode ser adaptado para grafos não planares. Por conseguinte, o presente capítulo mostra que este método, denominado Feline (para *Fast rEfined onLINE search*), é não apenas viável mas pode superar em desempenho os trabalhos mais relevantes publicados recentemente.

3.2 Definições

Considerando um grafo direcionado $G = (V, E)$ arbitrário a ser indexado, diz-se que um vértice $v \in V$ é alcançável por outro vértice $u \in V$, se e somente se existe um caminho entre u e v em G, de acordo com Definição 1.

Conforme comentado no capítulo de introdução, para simplificar as notações usadas neste texto, será considerado somente o grafo associado $G_{\text{DAG}} = (V_{\text{DAG}}, E_{\text{DAG}})$, i.e., o grafo direcionado acíclico (DAG) extraído de $G = (V, E)$ pelo agrupamento de todo componente fortemente conectado.

A ideia principal que fundamenta Feline é associar a todo vértice em V_{DAG} um par ordenado de números inteiros único em $\mathbb{N}^2$. Em outros termos, existe um mapeamento $i : V_{\text{DAG}} \to \mathbb{N}^2$, construído por Feline, associado a G_{DAG} onde, dados dois vértices distintos $(u, v) \in V_{\text{DAG}}^2$, v é dito alcançável a partir de u, denotado por $u \; E_{\text{DAG}}^* \; v$, se e somente se existe um caminho em G_{DAG} de u até v e $i(u) \preceq i(v)$. Em outras palavras, a todo vértice u é atribuída uma coordenada $(\prec_x^u, \prec_y^u)$ num plano cartesiano XY, onde

a abscissa X e a ordenada Y são as ordenações topológicas $\prec_x$ e $\prec_y$, respectivamente. Assim, $\prec_x^u$ é a posição de u em $\prec_x$ e $\prec_y^u$ é a posição de u em $\prec_y$, de forma que a relação de ordem parcial $\preceq$ é definida a seguir:

Definição 15 (Ordem parcial $\preceq$)**.** $\forall(\prec_x^u, \prec_y^u, \prec_x^v, \prec_y^v) \in \mathbb{N}^4$, $(\prec_x^u, \prec_y^u) \preceq (\prec_x^v, \prec_y^v) \Leftrightarrow u \prec_x v \wedge u \prec_y v$.

As propriedades de reflexividade, antissimetria e transitividade de $\preceq$ derivam diretamente das propriedades de $\leq$. Geometricamente, $(\prec_x^u, \prec_y^u) \preceq (\prec_x^v, \prec_y^v)$ significa que $(\prec_x^v, \prec_y^v)$ está no quadrante superior direito do sistema cartesiano bidimensional, com $(\prec_x^u, \prec_y^u)$ como origem. Diz-se que u domina v. Note que testar se essa relação é verdadeira requer tempo constante. Esse método garante que $\forall(u, w) \in E_{\text{DAG}}, i(u) \preceq i(w)$. Por conseguinte, uma consulta de alcançabilidade entre dois vértices u e v, denotada por $u \rightsquigarrow^? v$, retorna *verdadeiro* se $u = v$ ou $\exists w \in V_{\text{DAG}} \setminus \{u\} \mid i(u) \preceq i(w) \wedge (w\ E^*_{\text{DAG}}\ v)$. Essa afirmação, juntamente com a transitividade de $\preceq$, demonstram o seguinte teorema:

Teorema 2. $\forall(u, v) \in V^2_{DAG}$, $(u\ E^*_{DAG}\ v) \Rightarrow i(u) \preceq i(v)$.

A implicação $(u\ E^*_{\text{DAG}}\ v) \Rightarrow i(u) \preceq i(v)$ é sempre verdadeira (veja Definição 15). Por contraposição, se $i(u) \not\preceq i(v)$ então nenhum percurso no grafo é necessário para responder de forma negativa a consulta de alcançabilidade de u para v. Com esse teorema em mãos, pode-se complementar a definição de consulta de alcançabilidade:

Definição 16 (Consulta de alcançabilidade)**.**

$$\forall(u, v) \in V^2_{DAG},\ u \rightsquigarrow^? v \Rightarrow \begin{cases} u = v \\ ou \\ \exists(u, w) \in E_{DAG} \mid i(w) \preceq i(v) \wedge (w\ E^*_{DAG}\ v) \end{cases}.$$

De uma perspectiva lógica, escrever "$i(w) \preceq i(v) \wedge (w\ E^*_{\text{DAG}}\ v)$" ao invés de "$(w\ E^*_{\text{DAG}}\ v)$" não acrescenta informação alguma, uma vez que $(w\ E^*_{\text{DAG}}\ v) \Rightarrow i(w) \preceq i(v)$ (pelo Teorema 2). Todavia, é algoritmicamente interessante. Isso significa que a busca recursiva por um caminho desde u até v pode ser interrompida quando uma aresta levar a um vértice w tal que $i(w) \not\preceq i(v)$. Como consequência, somente um sub-grafo de G_{DAG} é percorrido.

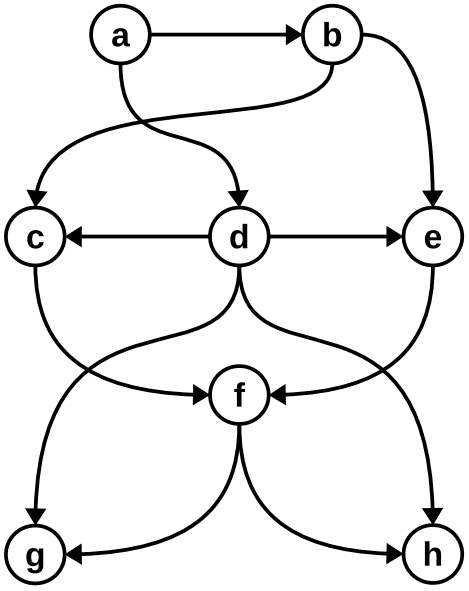
Com a ajuda do índice de Feline, responder a consultas de alcançabilidade $u \rightsquigarrow^? v$ torna-se um processo eficiente de apenas dois passos:

1. Testar se $i(u) \preceq i(v)$ e responder de forma negativa se $i(u) \not\preceq i(v)$ (pelo Teorema 2);

2. Caso contrário, procurar em G_{DAG} um caminho iniciando em u até v que passe somente por vértices cujas coordenadas sejam $\preceq i(v)$.

3.3 Interpretação Gráfica

Dado um DAG G_{DAG}, todo vértice u é associado a um par $(\prec_x^u, \prec_y^u) \in \mathbb{N}^2$ tal que, para quaisquer dois vértices $u, v \in G_{\text{DAG}}$, se existe um caminho de u até v, então a coordenada $\prec_x^u$ é menor ou igual à coordenada $\prec_x^v$ e a coordenada $\prec_y^u$ é menor ou igual à coordenada $\prec_y^v$, pois $u \prec_x v$ e $u \prec_y v$. A Figura 3.4 mostra um exemplo de um DAG e o seu índice, de tamanho $O(|V_{\text{DAG}}|)$.



	X	Y
a	1	1
b	2	3
c	4	5
d	3	2
e	5	4
f	6	6
g	7	8
h	8	7

(A) **(B)**

Figura 3.4. A visualização de um DAG: em (A) o DAG; em (B) o índice relacionado ao DAG apresentado, onde cada linha representa a coordenada de um vértice.

O índice resultante pode ser representado graficamente. O par de inteiros associado ao vértice é entendido como coordenadas no plano cartesiano.

O Teorema 2 significa graficamente que, para um dado vértice u, temos apenas que verificar seu quadrante superior direito, de forma a identificar os vértices que podem ser alcançados a partir de u. Se não, a alcançabilidade é respondida de forma negativa em tempo constante.

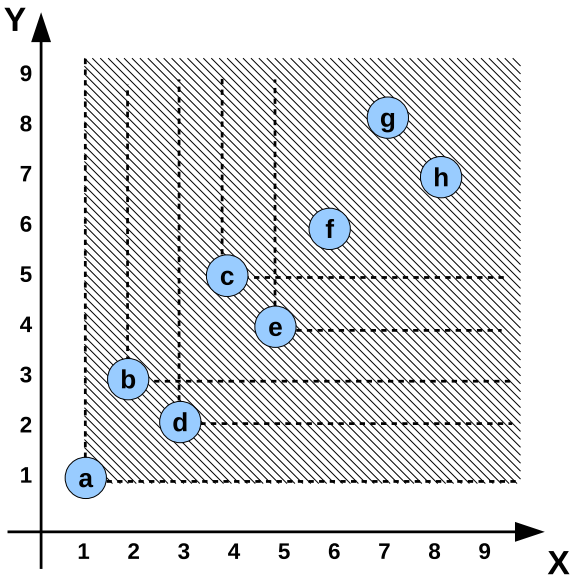


Figura 3.5. Exemplo de regiões de dominação.

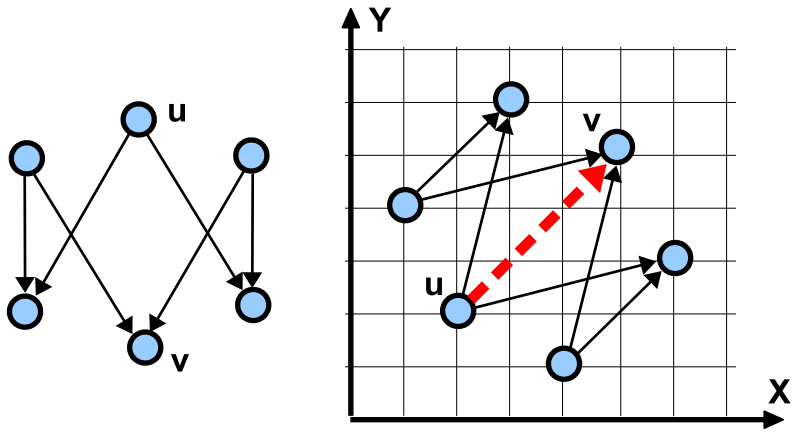


Figura 3.6. Exemplo de exceção entre u e v. A seta tracejada é uma exceção (ou, como em alguns textos, um *falsely implied path*), também chamado de *falso-positivo*.

As relações $\preceq$ entre os vértices (ou pontos no plano) são ilustradas na Figura 3.5, onde as linhas tracejadas expressam a área de alcançabilidade que se inicia em cada vértice. Por exemplo, considere os vértices a e h. Para $a\ E^*_{\text{DAG}}\ h$, necessariamente tem-se que $i(a) \preceq i(h)$ (pelo Teorema 2, e de fato a área hachurada iniciando de $i(a) = (1, 1)$ inclui o ponto $i(h) = (8, 7)$. Pode-se ver também na figura que d não está no canto superior direito de b, i. e., $i(b) \not\preceq i(d)$. Usando a contraposição do Teorema 2, conclui-se que $b \not\rightsquigarrow d$.

Infelizmente, o índice não permite que respostas positivas sejam dadas em tempo constante. Em outras palavras, o Teorema 2 deixa claro que $\forall(u,v) \in V^2_{\text{DAG}}$, $(u\ E^*_{\text{DAG}}\ v) \Rightarrow i(u) \preceq i(v)$, mas a implicação inversa nem sempre é verdadeira. A Figura 3.6 mostra um *crown DAG* conhecido como grafo S^0_3. No respectivo índice, tem-se que $i(u) \preceq i(v)$ mesmo se $u \rightsquigarrow^? v$ é falso, representado pela seta tracejada entre os vértices u e v (*falso-positivo*).

É importante notar que alguns grafos, tal como um S^0_3, não admitem um índice $2D$ que seja livre de falsos-positivos. De fato, tais grafos problemáticos existem com a construção de um índice nD com n arbitrariamente grande [Eades et al., 1994; Kornaropoulos & Tollis, 2011]. O problema de minimizar o número de falsos-positivos, isto é, dada uma ordenação topológica qualquer, encontrar uma ordenação complementar que minimize a quantidade de falsos-positivos, é conhecidamente NP-Difícil [Kornaropoulos & Tollis, 2011, 2012]. Contudo, pode-se gerar uma solução aproximada (localmente ótima), tal solução é apresentada a seguir.

3.4 Construção do Índice

Os passos para a construção do índice de alcançabilidade são apresentados nesta seção. Para tanto, o Algoritmo 3 gera as coordenadas (do plano XY) que irão compor o índice Feline. As coordenadas do eixo X (abscissas) são determinadas primeiro por um algoritmo de ordenação topológica (linha 2), resultando no conjunto $\prec_x$ com as posições (rank) de cada vértice. Qualquer algoritmo de ordenação topológica pode ser utilizado (algoritmos de tempo $O(|V_{\text{DAG}}|+|E_{\text{DAG}}|)$ são conhecidos [Kahn, 1962; Knuth, 1997]).

Para computar o conjunto de coordenadas Y (ordenadas), i. e., $\prec_y$, usa-se a heurística proposta por Kornaropoulos & Tollis [2012] (nas linhas 3 à 17). Cada passo da heurística toma uma decisão sobre a posição de um vértice, que minimizará o número de falsos-positivos. A heurística realiza repetidas remoções de vértices que não possuem antecessores (i. e., raízes) para gerar uma nova ordenação topológica $\prec_y$. Dado o conjunto inicial de raízes (linha 9), ela iterativamente escolhe a raíz com a maior posição (rank) em $\prec_y$ (linha 11), um caso particular do algoritmo de Kahn ([Kahn, 1962]).

O operador arg max (line 11) tenta evitar uma exceção (i. e., falso-positivo) entre o vértice escolhido e os vértices de iterações anteriores, escolhendo a raíz com a maior posição na ordenação topológica. Esta escolha foi demonstrada ser localmente ótima por Kornaropoulos & Tollis [2012]. Se uma raíz tem maior posição em $\prec_x$, então

Algoritmo 3: Construção do Índice

Dados: $(V_{\text{DAG}}, E_{\text{DAG}})$, um grafo direcionado acíclico cujo conjunto de vértices, V_{DAG}, é o conjunto de inteiros de 1 até $|V_{\text{DAG}}|$ e $E_{\text{DAG}} \subset V_{\text{DAG}}^2$

`// para cada` u, $\prec_x^u$ `é a posição de` u `em` $\prec_x$

1 **início**
2 $\quad \prec_x \leftarrow$ OrdenaçãoTopológica$(V_{\text{DAG}}, E_{\text{DAG}})$;
3 $\quad \prec_y \leftarrow \emptyset$;
$\quad$ `// Armazenando o DAG como um vetor de sucessores diretos de cada vértice e o vetor de seus graus de entrada`
4 $\quad$ heads $\leftarrow (\emptyset, \ldots, \emptyset)$;
5 $\quad d \leftarrow (0, \ldots, 0)$;
6 $\quad$ **para todo** $(u, v) \in E_{DAG}$ **faça**
7 $\quad\quad$ heads$_u \leftarrow$ heads$_u \cup \{v\}$;
8 $\quad\quad d_v \leftarrow d_v + 1$;
9 $\quad$ roots $\leftarrow \{v \in V_{\text{DAG}} \mid d_v = 0\}$;
10 $\quad$ **enquanto** *roots* $\neq \emptyset$ **faça**
11 $\quad\quad u \leftarrow \arg\max_{v \in \text{roots}}(\prec_x^v)$;
$\quad\quad$ `// Insere` u `ao final de` $\prec_y$
12 $\quad\quad \prec_y \leftarrow (\prec_y, u)$;
$\quad\quad$ `// Atualiza` d `e o conjunto roots`
13 $\quad\quad$ roots $\leftarrow$ roots $\setminus \{u\}$;
14 $\quad\quad$ **para todo** $v \in heads_u$ **faça**
15 $\quad\quad\quad d_v \leftarrow d_v - 1$;
16 $\quad\quad\quad$ **se** $d_v = 0$ **então**
17 $\quad\quad\quad\quad$ roots $\leftarrow$ roots $\cup \{v\}$;
18 $\quad$ **return** $(\prec_x, \prec_y)$

ela será atribuída a uma posição baixa em $\prec_y$. Depois que uma raíz é selecionada e removida de *roots*, novos vértices sem nenhum antecessor podem aparecer e o conjunto *roots* deve ser atualizado (linhas 13 à 17).

Por exemplo, considere o DAG da Figura 3.4-(A). Uma ordenação topológica pode gerar o conjunto $\prec_x$ com os vértices $\{a, b, d, c, e, f, h, g\}$, associados às coordenadas x com as posições (ranks) de 1 a 8. O algoritmo primeiro preenche o conjunto *roots* com o vértice $\{a\}$ (linha 9). Então, na linha 11, ele escolhe o vértice a do conjunto *roots* (com rank 1, em $\prec_x$) e coloca-o no conjunto $\prec_y$ (na primeira posição). Depois, na linha 13, a é removido de *roots*, que é atualizado com as novas raízes b e d, e agora $roots = \{b, d\}$. Então, como d é a raíz com a maior posição $\prec_x$, ele também é removido de *roots* e inserido na segunda posição de $\prec_y$. Repare que d possui $\prec_y^b = 2$ e que d não tem descendentes que se tornam novas raízes, e, no momento, $\prec_y = \{a, d\}$

e $roots = \{b\}$. O vértice b é o próximo escolhido e então tem-se que $\prec_y = \{a, d, b\}$ e $roots = \{c, e\}$. Na Figura 3.5, os vértices a, d, e b recebem as coordenadas y 1, 2 e 3, respectivamente. O algoritmo continua até que $\prec_y$ esteja completo.

3.5 Consultas de Alcançabilidade

O Algoritmo 4 apresenta a estratégia de busca do Feline. Como mencionado anteriormente, para dois vértices u e v, ele primeiro verifica se u é igual a v, devido à propriedade de reflexão ou em caso da busca alcançar o destino. Na linha 4, se $i(u) \not\preceq i(v)$ é verdadeiro, pode-se imediatamente parar a busca, pela contraposição do Teorema 2. Este último passo é conhecido como *negative-cut*, porque nenhuma busca é necessária para concluir que u não alcança v. Por outro lado, se $i(u) \preceq i(v)$, nenhuma conclusão pode ser tirada imediatamente. Neste caso, Feline precisa explorar todos os vértices dentro da região entre u e v recursivamente (em DFS).

Algoritmo 4: ConsAlcançabilidade

```
   Dados: (u, v) ∈ V²_DAG, dois vértices do DAG
1  início
2      se u = v então
3          retorna true;
4      se i(u) ⪯ i(v) então
5          para todo w ∈ heads_u faça
6              if ConsAlcançabilidade(w, v) then
7                  retorna true;
8      retorna false;
```

3.5.1 Discussão sobre o desempenho do Feline

Considerando as abordagens de busca on-line, no caso de consultas positivas[2] (ou mesmo falsos-positivos), o tempo de consulta pode aumentar enquanto novos vértices são percorridos por DFS. No entanto, devido às ordenações topológicas, o método de poda do Feline pode descartar os caminhos que não alcançam o vértice procurado. A Figura 3.7 ilustra a busca associada com uma consulta positiva $u \rightsquigarrow^? v$ no Grail, Ferrari e Feline. De forma mais específica, o triângulo representa o espaço de busca composto pelos vértices que são incluídos no espaço de busca de u.

[2]Usa-se o termo *consulta positiva* para referenciar aquelas consultas que recebem uma *resposta* positiva sobre a alcançabilidade, de forma oposta, usa-se o termo *consulta negativa*.

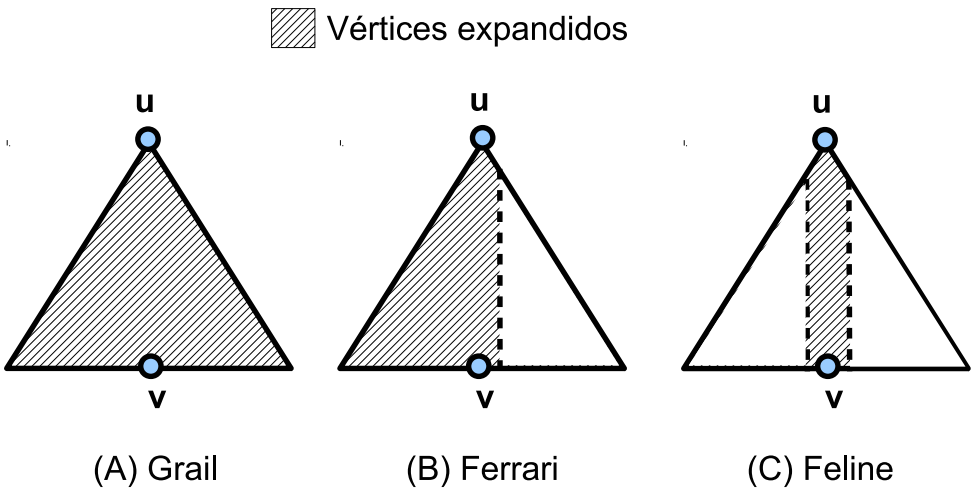


Figura 3.7. O espaço de busca de três abordagens on-line: Grail, Ferrari e Feline. Os triângulos representam os vértices expandidos por DFS. As áreas hachuradas são as ramificações não podadas na busca.

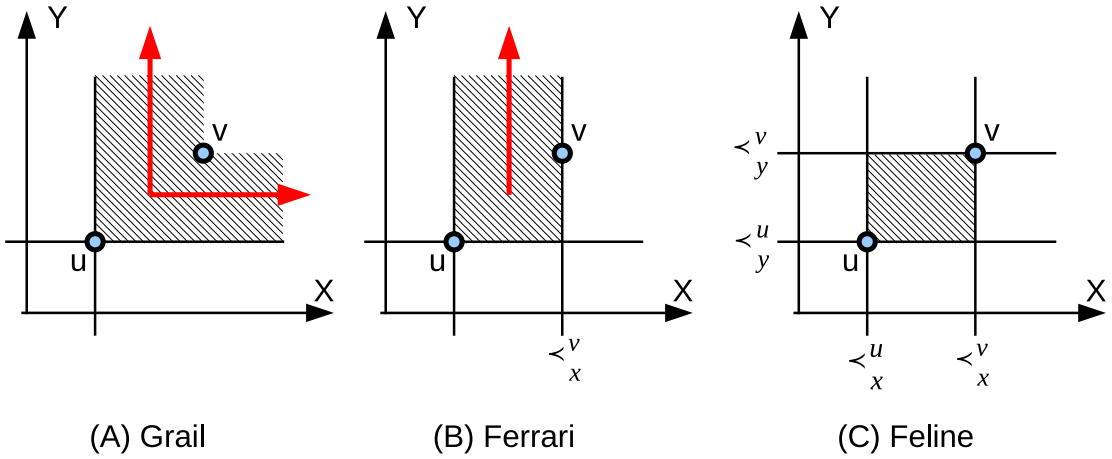


Figura 3.8. Comportamento da poda realizada pelo Feline quando emprega as mesmas estratégias de busca de Grail e Ferrari. As setas indicam a possibilidade de ramificações a serem exploradas no processo de busca (em DFS).

Em todas as abordagens, a busca irá expandir vértices até que não reste mais vértices a serem expandidos. Contudo, especialmente no caso da ocorrência de falsos-positivos, o vértice objetivo v nunca será encontrado pelo Grail e todos os vértices da subárvore contida nos intervalos de u serão expandidos (ou até que algum filtro interrompa a busca, como o filtro de nível apresentado a seguir). Ferrari evita esse caso podando vértices com ordem topológica maior do que v, limitando a busca a somente uma parte do respectivo espaço.

A Figura 3.8 mostra a equivalência entre os espaços de busca explorados por ambos Grail e Ferrari em relação ao Feline. Ela mostra o comportamento do mecanismo de busca do Feline se ele aplicar as mesmas estratégias usadas por Grail e Ferrari. O Ferrari possui um mecanismo de poda que se assemelha ao Feline usando apenas uma ordenação topológica (e. g., no eixo X). Embora Grail empregue uma busca direcionada, ele pode explorar caminhos que nunca poderão alcançar v, sendo equivalente ao Feline sem a verificação de limites em cada dimensão do plano.

3.5.2 Complexidade Computacional

É fácil ver que o Algoritmo 4 leva o tempo $O(1)$ quando, para dois vértices $u, v \in V_{\text{DAG}}$, ou u e v são o mesmo vértice, ou a relação de dominância fraca não é verdadeira (i. e., $i(u) \not\preceq i(v)$). No entanto, quando $i(u) \preceq i(v)$, a alcançabilidade é decidida por uma busca (em DFS) em, geralmente, uma parte pequena do grafo G identificada a partir do índice. O pior caso é ter que percorrer todo o grafo G_{DAG}, por exemplo, numa busca envolvendo dois vértices em extremidades opostas com uma fonte e um sumidouro, que resulta em um tempo $O(|V_{\text{DAG}}| + |E_{\text{DAG}}|)$.

Considerando o tempo de construção do índice, a complexidade de tempo do Algoritmo 3, desconsiderando o tempo para transformar G em G_{DAG}, é $O(|V_{\text{DAG}}| \log |V_{\text{DAG}}| + |E_{\text{DAG}}|)$, porque todas as arestas são enumeradas (uma vez na linha 6 e outra na linha 14) e *roots* é armazenado como uma estrutura *max-heap*, onde todo vértice é inserido uma única vez ($O(\log |V_{\text{DAG}}|)$), tal que a linha 11 tem um custo $O(1)$. $\prec_x$, $\prec_y$ e d são simples vetores e *heads* é um vetor de vetores. Uma vez que o algoritmo usa um algoritmo de ordenação topológica com tempo $O(|V_{\text{DAG}}| + |E_{\text{DAG}}|)$, a complexidade final é $O(|V_{\text{DAG}}| \log |V_{\text{DAG}}| + |E_{\text{DAG}}|)$. O tamanho do índice é $O(|V_{\text{DAG}}|)$.

3.6 Otimizações

3.6.1 Filtro Positive-Cut

Diversas abordagens descritas na literatura usam uma árvore de cobertura (*spanning-tree*) na construção do índice de alcançabilidade [Agrawal et al., 1989; Chen et al., 2005; Trißl & Leser, 2007; Wang et al., 2006; Yildirim et al., 2010]. De acordo com Yildirim et al. [2010], em uma árvore, a indexação usando apenas o algoritmo *min-post* (Algoritmo 2) é o bastante para responder a consultas positivas em tempo constante. Essa estratégia é chamada *positive-cut filter*.

Conforme mencionado no Capítulo 2, o algoritmo de Kameda para grafos planares cria uma indexação de alcançabilidade que permite responder consultas em tempo constante. Dessa forma, é possível utilizar esse algoritmo para acelerar algumas consultas de Feline, para aqueles caminhos restritos às arestas da árvore de cobertura (também chamada de árvore geradora ou *spanning tree*) extraída do grafo a ser indexado.

Após extrair a árvore de cobertura (ou floresta, no caso de dígrafos desconectados), aplica-se o algoritmo de Kameda e, com esse índice resultante, a alcançabilidade pode ser garantida para consultas com resultado positivo. No entanto, nada pode-se afirmar sobre as arestas que não aparecem na árvore. Esse é o motivo de usar essa estratégia apenas como um *filtro*, i. e., adicionando mais um passo de poda antes de uma possível busca no grafo.

A Figura 3.9 apresenta uma árvore de cobertura extraída do DAG da Figura 3.4 e indexada com o algoritmo de Kameda (os rótulos em cada vértice representam o índice). Por exemplo, considerando o vértice h, pode-se concluir que a consulta $a \rightsquigarrow^{?} h$ será positiva, sem a necessidade de busca, pois $(1,1) \preceq (7,6)$. No entanto, nada pode-se afirmar sobre a consulta $b \rightsquigarrow^{?} h$, uma vez que não existem arestas na árvore conectando os dois vértices.

Feline adiciona dois passos extras ao Algoritmo 3 para computar essas novas coordenadas. Primeiro, é feita a extração da árvore de cobertura (que pode ser feita diretamente pela ordenação topológica na linha 2), e, segundo, a aplicação do algoritmo de Kameda. Para usar o filtro *positive-cut*, o Algoritmo 4 precisa ser alterado para o Algoritmo 5, onde as novas linhas 1 e 2 verificam as coordenadas atribuídas para cada vértice, e onde $Ki(u)$ retorna a coordenada de u obtida com Kameda na respectiva árvore de cobertura.

3.6.2 Filtro de nível

Outro filtro utilizado no Feline é chamado de Filtro de Nível (*Level Filter* Yildirim et al. [2010]) e que também é utilizado em outras abordagens como o Grail e o Ferrari. Esse filtro leva em consideração o nível de cada vértice no DAG.

O nível de um vértice v (l_v) pode ser entendido como a sua profundidade [Bender et al., 2009]: se v não possui antecessores imediatos, então $l_v = 0$; caso contrário, $l_v = \max_{u:(u,v)\in E_{\text{DAG}}} l_u + 1$. De acordo com [Bender et al., 2009], o nível de um vértice induz sua ordem topológica, pois se u precede v no DAG G_{DAG} e $u \neq v$, então $l_u < l_v$. O Algoritmo 5 aplica esse filtro (linha 6).

A Figura 3.10 ajuda a entender a aplicação do filtro de nível. O exemplo mostra que uma consulta com falso-positivo $h \rightsquigarrow^{?} g$, no respectivo grafo, pode ser podada em

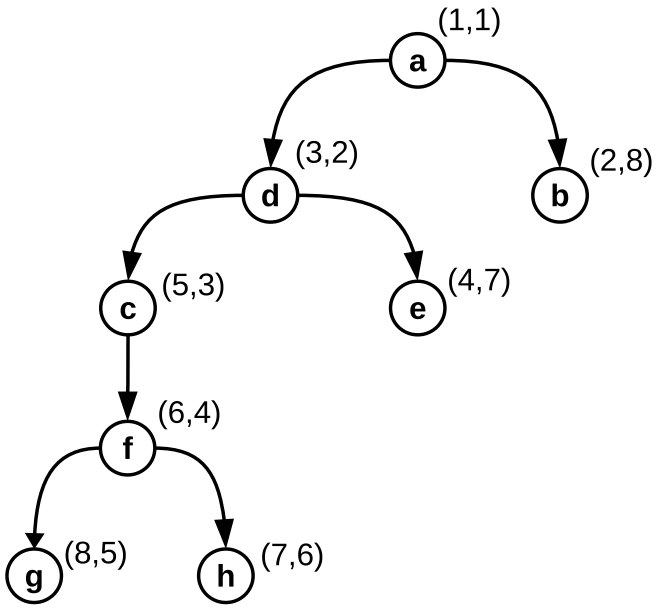


Figura 3.9. Indexação com o algoritmo de Kameda para uma árvore de cobertura extraída do DAG da Fig.3.4.

Algoritmo 5: ConsAlcançabilidade2

Dados: $(u, v) \in V^2_{\mathrm{DAG}}$, dois vértices do DAG

```
1  início
2      se Ki(u) ⪯ Ki(v) então
3          retorna true;
4      se u = v então
5          retorna true;
6      se i(u) ⪯ i(v) e l_u < l_v então
7          para todo w ∈ heads_u faça
8              se ConsAlcançabilidade2(w, v) então
9                  retorna true;
10     retorna false;
```

tempo constante apenas verificando que h e g estão no mesmo nível.

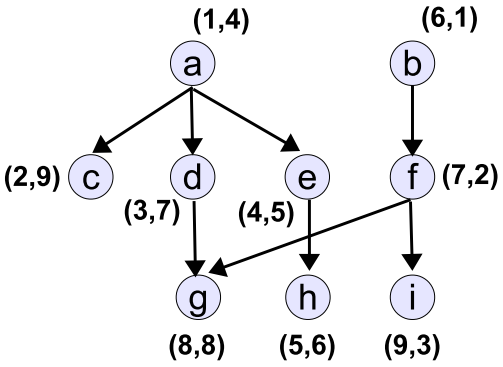


Figura 3.10. Exemplo onde o vértice h não alcança g, mas existe uma exceção entre eles (g está na área de dominação de h). No entanto, g e h estão no mesmo nível, e assim o *filtro de nível* interrompe a busca. Os números representam as coordenadas atribuídas pelo Feline.

3.7 Experimentos

3.7.1 Metodologia Experimental

Conduzimos experimentos para comparar a abordagem Feline com as abordagens do estado-da-arte: Grail [Yildirim et al., 2010], Ferrari, Nuutila's Interval [Nuutila, 1995; van Schaik & de Moor, 2011] e TF-Label [Cheng et al., 2013]. Assim como o Feline, Grail e Ferrari também empregam uma busca on-line onde o grafo deve permanecer em memória junto com o índice gerado. Interval e TF-Label são baseados na compressão do fecho transitivo, gerando um índice off-line, isto é, as consultas podem ser respondidas com base somente no índice, sem a necessidade de manter o grafo original na memória.

As versões otimizadas dos métodos *baseline* foram usadas nos experimentos (seguindo as recomendações dos autores). Todas as implementações (Grail, Ferrari, Interval e TF-Label) foram fornecidas pelos seus autores e estão codificadas em C++, assim como Feline.

Utilizamos dois conjuntos de grafos reais (esparsos e densos), separados em grafos pequenos (i. e., com menos de 100.000 arestas) e grandes, além de grafos sintéticos. Os experimentos com grafos pequenos foram realizados em uma máquina Intel(R) Xeon(R) CPU E5620 (8-core, 2.40GHz) com 32G de RAM. Para grafos grandes, usamos um Intel(R) Xeon(R) CPU E5620 (8-core, 2.40GHz) com 96G de RAM. No entanto, todas as implementações são *single threaded*.

3.7.1.1 Consultas

Geramos um conjunto de consultas diferente para cada grafo (real ou sintético) contendo 500k pares de vértices. Em seguida, submetemos o respectivo conjunto de consultas para cada grafo. Os resultados apresentados representam a média entre 10 execuções para cada grafo. Com esses conjuntos de consultas fomos capazes de estimar o desempenho em termos do tempo de consulta, tempo de construção dos índices e seus tamanhos. As respostas às consultas foram validadas por meio de buscas exaustivas nos grafos, i.e., para cada consulta, uma busca em profundidade no respectivo grafo foi realizada. Dessa forma, os algoritmos experimentados tiveram suas saídas comparadas, comprovando-se que são confiáveis.

Todos os grafos, códigos da abordagem proposta e os conjuntos de testes estão disponíveis em `http://www.dcc.ufmg.br/~renerv/feline`.

3.7.2 Grafos utilizados nos experimentos

Grafos	Vértices	Arestas	Coeficiente Agr.	Diâmetro efetivo	Raízes	Folhas
Arxiv	6000	66707	0,35	5,48	961	624
Yago	6642	42392	0,24	6,57	5176	263
Go	6793	13361	0,07	10,92	64	3087
Pubmed	9000	40028	0,10	6,32	2609	4702
citeseer	10720	44258	0,28	8,36	4572	1868
Uniprot22m	1595444	1595442	0,00	3,3	1556157	1
Cit-patents	3774768	16518947	0,09	10,5	515785	1685423
citeseerx	6540401	15011260	0,06	8,4	567149	5740710
Go-uniprot	6967956	34770235	0,00	4,8	6945721	4
Uniprot100m	16087295	16087293	0,00	4,1	14598959	1
Uniprot150m	25037600	25037598	0,00	4,4	21650056	1

Tabela 3.1. Conjunto de grafos para experimentos

3.7.2.1 Grafos Reais

Usamos 5 grafos reais densos e pequenos (i.e., com menos de 100.000 vértices): *Arxiv*[3], *Citeseer*[4], *Go*[5], *Pubmed*[6] e *Yago*[7]. Também usamos 6 grafos grandes (esparsos e densos): *Cit-Patents*[8], *Citeseerx*[9], *Uniprot22m*, *Go-uniprot*, *Uniprot100m* e *Uni-*

[3] arxiv.org
[4] citeseer.ist.psu.edu
[5] www.geneontology.org
[6] www.pubmedcentral.nih.gov
[7] www.mpi-inf.mpg.de/suchanek/downloads/yago
[8] snap.stanford.edu/data/cit-Patents.html
[9] citeseerx.ist.psu.edu

prot150m[10]. A Tabela 3.1 mostra algumas características desses grafos, tal como o número de vértices e arestas, coeficiente de agrupamento (*clustering*), diâmetro efetivo (ao invés do diâmetro convencional) disponíveis em [Yildirim et al., 2010] (os autores usaram o software SNAP [11] para computar esses valores). O diâmetro efetivo (ou excentricidade efetiva) é o tamanho estimado do caminho na qual 90% de todos os pares de vértices conectados são mutuamente alcançáveis [Chakrabarti & Faloutsos, 2012]. O diâmetro efetivo é mais robusto do que o diâmetro e tem sido usado em diversos trabalhos para analisar propriedades topológicas de grafos na internet [Chakrabarti & Faloutsos, 2012; Siganos et al., 2006].

3.7.2.2 Grafos Sintéticos

Os experimentos usando DAGs sintéticos aleatórios têm por objetivo avaliar a escalabilidade de Feline. A Tabela 3.2 apresenta esses datasets. Cada grafo foi gerado de acordo com o especificado em Yildirim et al. [2010]. Primeiro, dado um conjunto predeterminado de vértices, criou-se uma permutação desse conjunto, na prática, uma ordenação topológica diferente. Então, para um dado número de arestas, foram selecionados dois vértices de forma aleatória e conectados, respeitando a ordem topológica de cada um. As características desses grafos sintéticos não são mostradas, pois o tamanho de cada grafo inviabiliza a computação.

Grafo	Vértices	Arestas
50M	50M	50M
60M	60M	60M
70M	70M	70M
80M	80M	80M
90M	90M	90M
100M	100M	100M
200M	200M	200M
50M-5	50M	250M
50M-10	50M	500M
100M-5	100M	500M
100M-10	100M	1000M

Tabela 3.2. Grafos sintéticos

[10] www.uniprot.org
[11] snap.stanford.edu/snap/

	Tempo de Construção (ms)				
Grafos	Grail	Interval	Ferrari	TF-Label	Feline
Arxiv	13,607	48,136	30,085	7885,118	**5,533**
Yago	11,714	13,707	18,501	50,998	**4,206**
Go	8,160	7,504	10,583	30,715	**3,333**
Pubmed	13,967	27,392	32,609	86,405	**5,528**
Citeseer	17,403	31,409	34,591	154,177	**7,356**
Uniprot22m	7121,763	1189,403	1083,154	2546,216	**818,732**
Cit-patents	22575,605	504209,060	28157,440	253828,562	**6065,227**
Citeseerx	25400,748	8049,076	18134,930	104418,257	**5519,161**
Go-uniprot	45998,303	23002,793	29491,370	70984,078	**6051,333**
Uniprot100m	93990,593	13279,480	14223,680	48741,968	**10533,610**
Uniprot150m	155546,666	21756,457	25738,860	70321,609	**17745,300**

	Tempo de Consulta (ms)				
Grafos	Grail	Interval	Ferrari	TF-Label	Feline
Arxiv	745,230	**28,087**	166,103	164,076	493,092
Yago	27,810	16,099	33,119	11,689	**9,893**
Go	66,344	**18,193**	29,209	18,597	81,846
Pubmed	32,269	12,614	23,530	10,744	**9,997**
Citeseer	37,842	17,373	32,247	15,814	**13,226**
Uniprot22m	36,524	87,393	23,263	19,670	**18,115**
Cit-patents	91,998	74,917	87,360	62,005	**41,291**
Citeseerx	97,654	47,417	61,792	91,149	**40,292**
Go-uniprot	37,633	112,861	213,040	30,319	**22,817**
Uniprot100m	53,373	117,323	60,839	48,084	**26,507**
Uniprot150m	63,384	121,837	61,466	55,397	**27,764**

Tabela 3.3. Tempos de construção e de consulta para os grafos reais. Tempos médios (em milissegundos).

3.7.3 Resultados

3.7.3.1 Tempo de Construção do Índice

A Tabela 3.3 mostra os valores médios obtidos para os tempos de construção dos índices (em milissegundos). Os melhores resultados estão destacados com um fundo cinza. Posteriormente, são apresentados os resultados para os grafos sintéticos.

Para os grafos pequenos, os tempos de construção do Feline são pelo menos 3 vezes mais rápidos do que o Grail, Ferrari, Interval e TF-Label. Apesar do bom desempenho de Feline, para os grafos Arxiv e Go o algoritmo Interval obteve os melhores resultados. Acredita-se que a estrutura destes grafos favoreça o surgimento de falsos-positivos, forçando o algoritmo a percorrer o grafo em muitas consultas. Como é impraticável contabilizar a quantidade de falsos-positivos devido ao tamanho dos grafos, essa suposição pode ser constatada na Figura 3.11, que apresenta a quantidade de

consultas respondidas em tempo constante nos experimentos, i. e., sem a necessidade de percorrer o grafo. Nela, percebe-se que a quantidade de consultas que necessitam de uma busca no grafo é maior para os grafos Arxiv e Go. Para os grafos grandes, o desempenho de Feline é pelo menos 10 vezes melhor do que os outros.

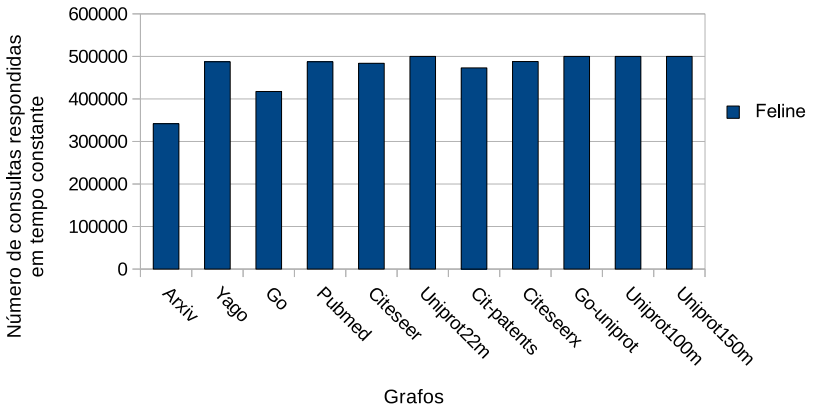


Figura 3.11. O gráfico mostra a quantidade de consultas que foram respondidas em tempo constante em cada experimento.

Aplicou-se o teste de Friedman [Friedman, 1937] a cada resultado para obter sua significância estatística. Em um nível de confiança de 0.1, o desempenho de Feline é melhor em todos os casos. Procedendo ao teste *post-hoc* de Nemenyi Hollander & Wolfe [1999], desenvolvemos o diagrama da Fig. 3.12, que representa a diferença crítica de desempenho entre as abordagens. No diagrama, o eixo representa os *rankings* médios dos métodos, que vai de 1 (significando o melhor) até 4 (pior). Quando comparados todos os métodos entre si, formamos grupos que não apresentam diferença significativa (ligados por uma linha em negrito). A diferença crítica (CD) é também mostrada e, se a distância entre os rankings de duas abordagens é maior que o valor de CD, então as abordagens não são agrupadas. Por exemplo, o grupo formado por Interval, Grail e Ferrari indica que as diferenças de seus desempenhos não são estatisticamente significativas.

3.7.3.2 Tempo de Consulta

Os tempos médios para responder uma consulta são também mostrados na Tabela 3.3. O método Interval alcançou bons resultados mas, como mostrar-se-á, não conseguiu-se construir os índices para grafos sintéticos muito grandes. Novamente, num nível de confiança de 0.1, o teste de Friedman mostra que os desempenhos são diferentes.

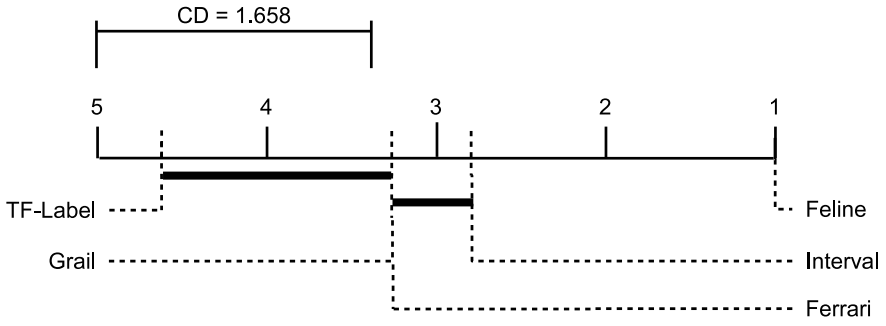


Figura 3.12. O diagrama de diferença crítica para os tempos de construção.

Contudo, como detalhado na Fig. 3.13, Feline é competitivo com Interval e TF-Label, uma vez que eles estão no mesmo grupo. Assim, como Feline é mais rápido do que Grail e Ferrari, podemos afirmar que Feline tem o melhor tempo de consulta (considerando os grafos reais). De fato, Feline é tipicamente 2 vezes mais rápido.

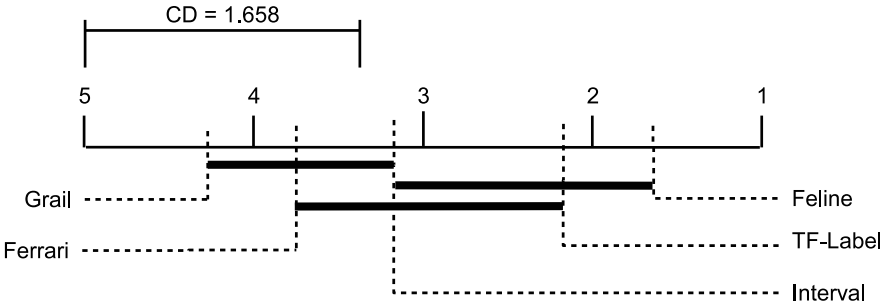


Figura 3.13. O diagrama de diferença crítica para os tempos de consulta.

3.7.3.3 Abordagem Feline-B (bidirecional)

Em muitos grafos, os graus de entrada e saída de cada vértice seguem distribuições diferentes. Para verificar se essa diferença pode influenciar o índice Feline, representamos graficamente os índices de alguns grafos e sua respectiva versão invertida (o grafo invertido), i. e., para cada DAG G_{DAG}, geramos outro DAG $G^T_{\text{DAG}} = (V_{\text{DAG}}, E^T_{\text{DAG}})$, onde E^T_{DAG} é o conjunto de arestas com as direções invertidas. Após verificar os gráficos, notamos que os vértices do grafo normal e invertido recebem coordenadas diferentes. Isso ocorre porque, com a inversão das arestas de um DAG, devido às distribuições dos graus de entrada e saída de seus vértices, o número de sucessores (ou antecessores) de cada vértice muda, bem como o número de fontes e sumidouros do DAG. A Figura 3.14 mostra as representações gráficas de quatro índices, para os grafos: Arxiv, Yago, Go

e Pubmed (seus tamanhos tornaram possível obter essas representações), onde cada ponto representa um vértice do respectivo grafo. Obviamente uma consulta $u \rightsquigarrow^? v$ no DAG G_{DAG} é equivalente a uma consulta $v \rightsquigarrow^? u$ no DAG G^T_{DAG}, mas, como os vértices possuem coordenadas diferentes em cada índice, cada um resulta em um desempenho diferente para a mesma consulta.

Considerando as representações gráficas, decidiu-se investigar se o uso do grafo invertido pode contribuir para melhorar o desempenho de Feline. Por isso, incluiu-se novos experimentos onde Feline gera um índice invertido. A versão que usa esse índice é chamada de Feline-I. A Tabela 3.4 resume os tempos de construção e consulta do Feline-I.

	Tempo de Construção			**Tempo de Consulta**		
Grafos	Feline	Feline-I	Feline-B	Feline	Feline-I	Feline-B
Arxiv	5,533	5,467	10,760	493,092	894,211	**420,806**
Yago	4,206	3,776	8,090	9,893	24,754	**9,435**
Go	3,333	3,737	6,706	81,846	**34,753**	44,390
Pubmed	5,528	5,465	10,700	9,997	**9,006**	9,517
Citeseer	7,356	6,821	14,102	13,226	19,508	**11,608**
Uniprot22m	818,732	740,511	1567,305	**18,115**	18,964	19,461
Cit-patents	6065,227	5994,798	11913,330	41,291	**27,894**	32,226
Citeseerx	5519,161	6242,018	11243,650	**40,292**	56,193	41,777
Go-uniprot	6051,333	6616,405	12202,540	**22,817**	53,354	24,392
Uniprot100m	10533,610	9417,537	20373,920	**26,507**	28,410	29,837
Uniprot150m	17745,300	15985,690	34359,940	**27,764**	29,749	31,570

Tabela 3.4. Resultados para Feline-B. Tempos médios (em milissegundos).

Os ganhos observados de Feline-I, para os mesmos grafos, indicam que o índice invertido pode ser usado para compor uma nova e mais eficiente estratégia de poda. Essa nova estratégia é baseada em ambos os índices (normal e invertido) ao mesmo tempo, i. e., na interseção das áreas alcançáveis (dois testes de dominância na linha 4 do Algoritmo 4). Feline-B (bidirecional) implementa essa nova estratégia. para uma consulta $u \rightsquigarrow^? v$, todos os vértices fora da área de (u, v) no índice normal e na área (v, u) do invertido, são descartados. O desempenho de Feline-B é comparado com o Feline e Feline-I na Tabela 3.4. Note que o tempo médio para a construção do índice é quase o dobro, mas os resultados de Feline-B são próximos ao melhor índice individual (Feline e Feline-I) para cada grafo, e melhor do que a média de ambos Feline e Feline-I.

3.7.3.4 Resultados para os Grafos Sintéticos

Os tempos de construção e de consultas são reportados nas Figuras 3.15 e 3.16, respectivamente. Note que para os grafos 200M, 50M-5, 50M-10, 100M-5 and 100M-10 mostrados, não obtivemos resultados para o Interval e nem para o TF-label, pois eles falharam durante os experimentos para esses grafos. Esses métodos são do tipo off-line,

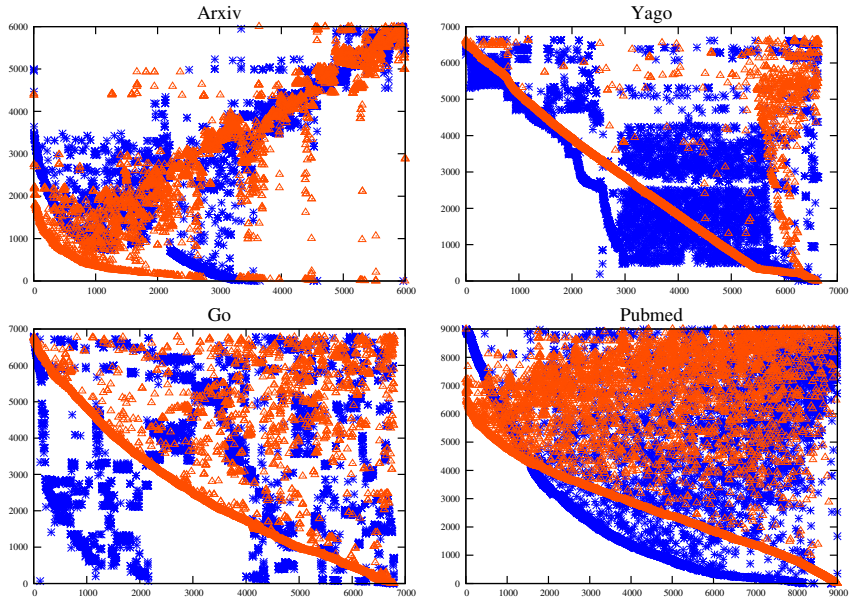


Figura 3.14. Representações gráficas dos índices. Legenda: $*$ para os normais e ▲ para os invertidos.

que fazem uso de estruturas de dados complexas para suportar o processo de extração de seus índices. Isso faz com que os requisitos de memória sejam elevados, como no caso do Interval e do TF-label, que necessitam de múltiplas fases de computação para compactar o fecho transitivo completo dos grafos.

Esses resultados mostram que Feline e Feline-B são escaláveis e verdadeiramente competitivos com as abordagens do estado da arte. Com os experimentos, mostrou-se que Feline possui alto desempenho e escalabilidade com relação ao tempo de construção, devido ao índice pequeno e seu algoritmo de indexação simplificado. Já o Feline-B possui os melhores tempos de consulta, pois seu mecanismo de poda, com o auxílio dos filtros, é eficiente para consultas positivas e negativas.

3.7.3.5 Tamanho do índice

As Figuras 3.17 e 3.18 mostram o tamanho de cada índice construído pelo Feline. As figuras não mostram os valores para Feline-I, porque o seu índice tem o mesmo tamanho do gerado pelo Feline.

Devido aos múltiplos intervalos, Grail gera índices que são maiores do que os gerados pelo Feline, sendo 2 vezes maior quando $d = 3$ e 4 vezes maior quando $d = 5$. O índice gerado pelo Feline-B é composto pelo índice normal e pelo invertido, sendo então duas vezes maior que o gerado pelo Feline ou Feline-I.

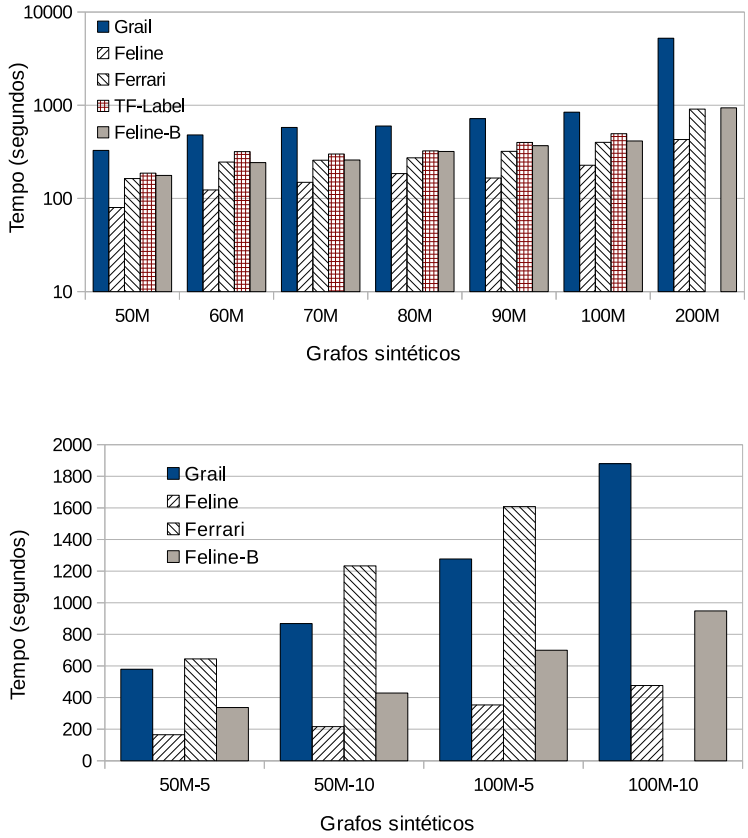


Figura 3.15. Tempos de construção para os grafos sintéticos.

3.7.4 Avaliação do SCARAB

Grafo	Feline-SCAR	Grail-SCAR
Arxiv	**203,761**	319,452
Yago	**8,691**	20,760
Pubmed	**8,703**	17,932
citeseer	**10,539**	19,595
uniprot22m	**23,033**	63,202
Cit-patents	**30,749**	52,908
citeseerx	**35,248**	171,752
Go-uniprot	**28,025**	74,649
uniprot100m	**30,970**	85,538
uniprot150m	**33,068**	91,111

Tabela 3.5. Tempos de consulta pra as implementações baseadas no SCARAB.

De acordo com Jin et al. [2012], o framework SCARAB pode acelerar os tempos de consulta de alguns métodos. Embora não seja objetivo deste trabalho a investigação da aplicação dos chamados métodos de *boosting*, usamos o framework para mostrar que a solução Feline pode obter vantagens de seu uso. Para isso, implementou-se uma versão

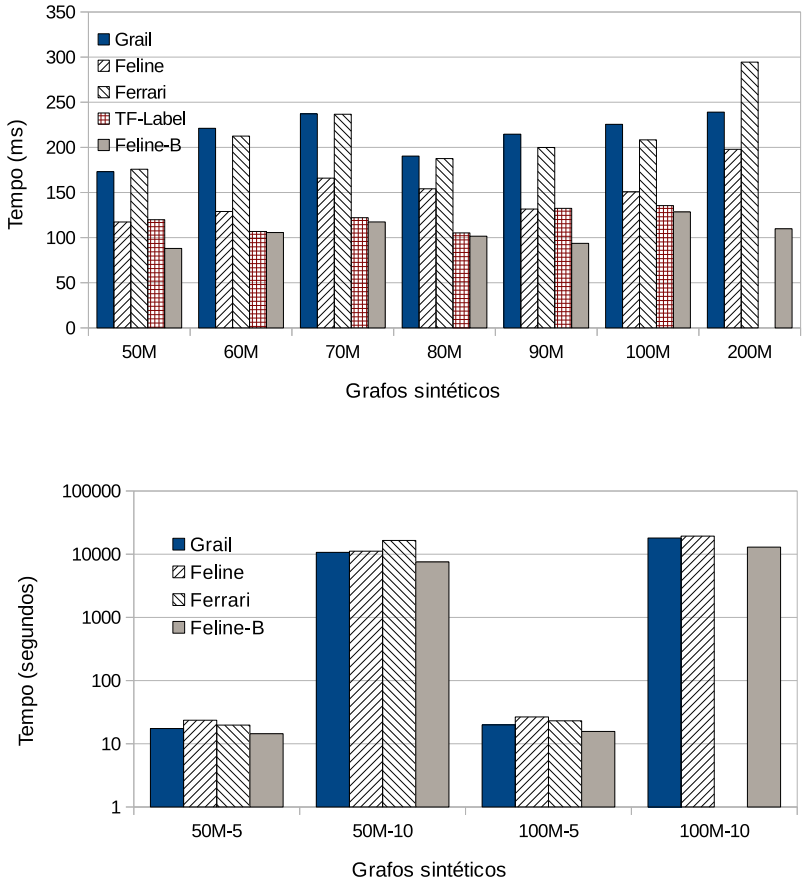


Figura 3.16. Tempos de consulta para os grafos sintéticos.

denominada Feline-SCAR, derivada do Feline-B. Em teoria, de acordo com os autores, qualquer abordagem existente pode ser acelerada pelo framework, apesar da dificuldade em implementar as respectivas versões do Interval e TF-label devido à complexidade de adaptação das abordagens do tipo off-line, pois é necessário modificá-las para que suas estruturas de dados e formatos de arquivos de saída sejam compatíveis com o framework. A versão denominada Grail-SCAR foi fornecida pelos autores e destacada como a melhor abordagem em Jin et al. [2012].

A Tabela 3.5 apresenta os resultados de alguns experimentos seguindo as recomendações dos autores do SCARAB. Embora alguns ganhos sejam observados, com um nível de confiança de 0.1, Feline-SCAR e Grail-SCAR são diferentes, como destacado na Figura 3.19.

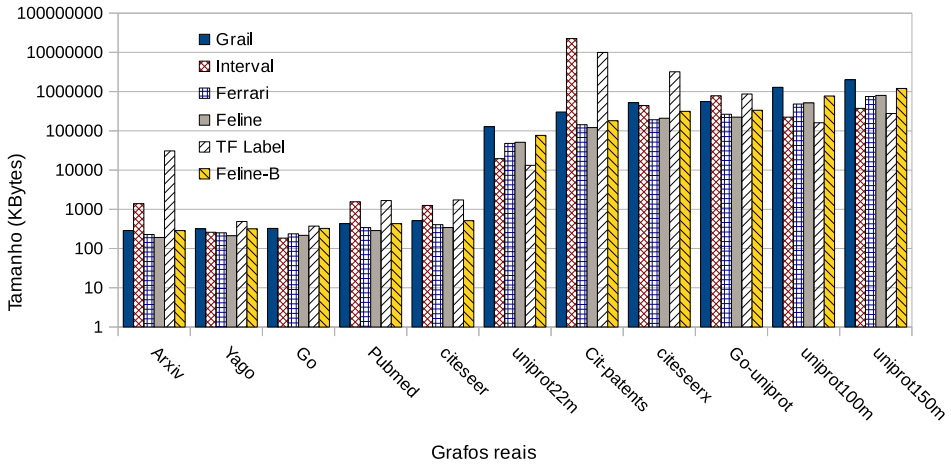


Figura 3.17. Tamanhos dos índices para os grafos reais.

3.7.5 Discussão

Para todos os grafos, pode-se notar que, apesar das características particulares de cada grafo, Feline alcançou os melhores resultados. Destaca-se aqui algumas avaliações:

- A abordagem Interval não escala bem, falhando para os grafos grandes. Acredita-se que o problema esteja nos requisitos de memória para o seu índice gerado. Os autores da abordagem não estipularam qual seria a relação entre quantidade de memória e tamanho do grafo.
- Ferrari e TF-Label realmente apresentam tempos de consulta e de construção melhores do que o Grail, isso é suficiente para que Grail não seja mais considerado o estado-da-arte, conforme Jin et al. [2012].
- Os estudos apresentados por [Cheng et al., 2013] não são abrangentes e o mesmo não explica os motivos que levaram a implementação a falhar para os grafos sintéticos grandes.
- Todas as abordagens de busca on-line estudadas podem realizar podas de consultas negativas (em tempo constante), como também aplicar alguma estratégia de *positive-cut*. Dessa forma, concluímos que a diferença entre seus desempenhos realmente vem da eficiência na busca para consultas não podadas previamente ou nos casos de consultas contendo falsos-positivos. Mostramos que Feline descarta mais ramificações na busca, evitando vértices que aparecem depois do vértice alvo nas ordenações topológicas.

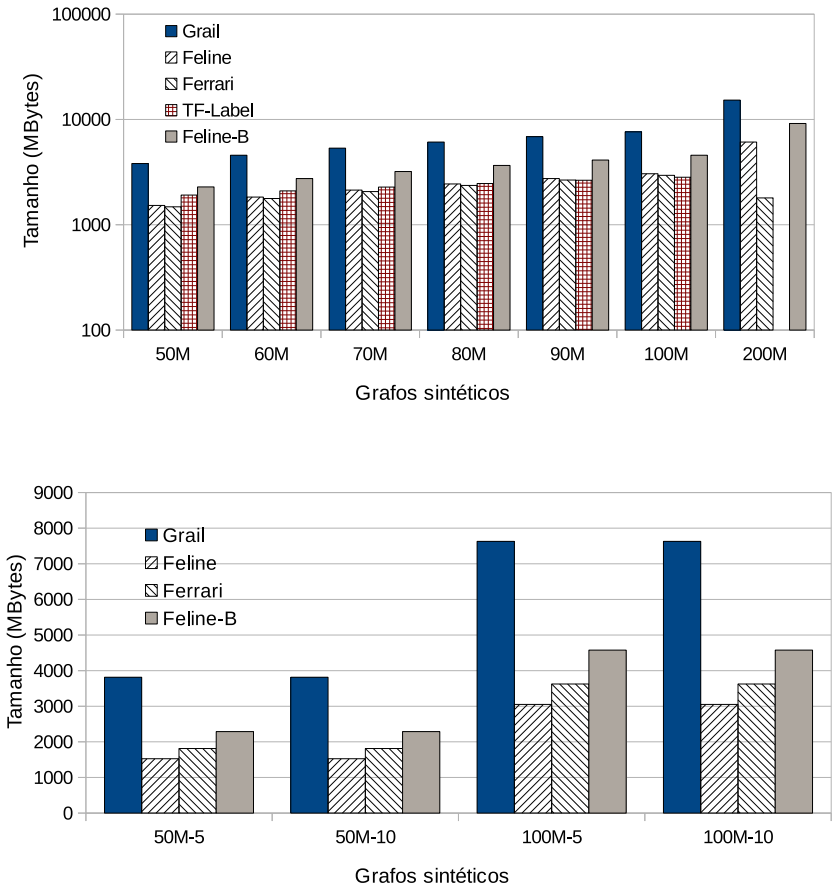


Figura 3.18. Tamanhos dos índices para os grafos sintéticos.

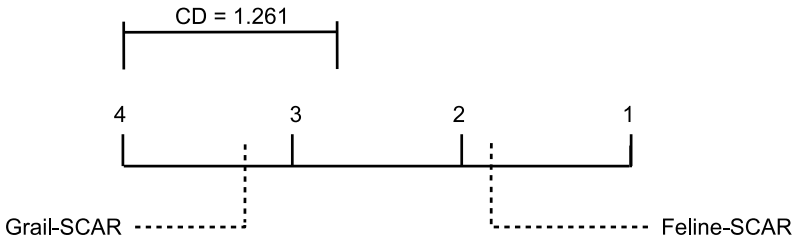


Figura 3.19. O diagrama de diferença crítica para os experimentos com o SCARAB.

- A abordagem relativamente simples de Feline para a indexação contribui para a geração de um índice pequeno. O tamanho linear do índice (com o número de vértices do DAG associado) evita a parametrização indesejável *a priori*, para considerar o número de intervalos que serão necessários. Por exemplo, no Grail, essa

parametrização visa otimizar seu tempo de consulta e tamanho do índice. Considerando o Interval, essa característica do Feline é ainda melhor, pois o número de intervalos é encontrado dinamicamente no Interval. Já o número de intervalos usados pelo Ferrari é predefinido como no Grail, mas algumas otimizações incrementam o índice dinamicamente também.

- Feline e Feline-B representam o melhor entre dois compromissos: tempo de construção e tempo de consulta. Feline alcança os melhores tempos de construção para todos os datasets, e o Feline-B alcança os melhores tempos de consulta, com um índice de tamanho competitivo.

3.8 Conclusão

Dado um grafo, considera-se neste trabalho a tarefa de responder eficientemente se existe um caminho que conecte dois vértices arbitrários. Esse é um problema importante e desafiador em muitas aplicações que utilizam grafos muito grandes.

Este capítulo apresentou um novo método de indexação de grafos para o problema de alcançabilidade. Inspirado na área de *Graph Drawing*, foi proposto o método Feline, que cumpre requisitos de escalabilidade e facilidade de implementação. Os experimentos realizados mostraram que Feline supera os métodos que compõem o estado-da-arte com referência ao tempo de consulta, tempo de construção e tamanho do índice gerado.

O método apresentado é aplicável somente a grafos estáticos. No próximo capítulo apresenta-se uma extensão do Feline para manipular modificações dinâmicas nos grafos ao mesmo tempo que o índice é mantido atualizado.

Capítulo 4

Alcançabilidade em Grafos Dinâmicos: Feline-PK

Este capítulo apresenta Feline-PK, cujo objetivo é atualizar de forma eficiente o índice em reação a uma atualização unitária (inserção ou remoção de um vértice ou aresta) de um dígrafo cíclico ou acíclico. Feline-PK é basedo em heurísticas de forma que muitos (se não todos, no caso de uma resposta negativa) vértices são descartados quando um caminhamento é feito no grafo para responder a uma consulta. O índice de Feline-PK suporta atualizações incrementais eficientes, i.e., permite identificar um número pequeno de vértices cujos índices devem mudar para considerar a última atualização. Demonstramos que atualizações incrementais são muito mais rápidas que uma reconstrução inteira.

O restante do capítulo está organizado como a seguir:

- Na Seção 4.1, apresenta-se os conceitos necessários e definições importantes para o entendimento dos algoritmos.
- A Seção 4.2 apresenta os algoritmos para atualizações dinâmicas, seguidos pela avaliação experimental na Seção 4.3.
- A Seção 4.4 apresenta um estudo preliminar sobre atualização do índice dada a inserção de conjuntos (lotes) de arestas.
- A Seção 4.5 conclui este capítulo.

4.1 Preliminares

Para facilitar a leitura deste capítulo e melhor entender conceitos importantes, o leitor encontrará no texto que se segue um resumo da abordagem Feline considerando as novas notações utilizadas que foram adaptadas para o contexto de grafos dinâmicos. Optou-se por separar algumas definições básicas deste capítulo em relação ao anterior, para que a leitura e compreensão de ambos os métodos siga com maior fluidez e independência. Este capítulo apresenta também uma visão geral sobre o algoritmo PK para ordenação topológica dinâmica.

Dado um DAG $(V_{\text{DAG}}, E_{\text{DAG}})$ e um subconjunto S de V_{DAG}, $in(S) = \{u \in V_{\text{DAG}} \setminus S \mid \exists v \in S \wedge uE_{\text{DAG}}v\}$ e $out(S) = \{v \in V_{\text{DAG}} \setminus S \mid \exists u \in S \wedge u\ E_{\text{DAG}}\ v\}$. O DAG é armazenado como listas de adjacências de seus vértices e o custo para computar $out(S)$ é $O(|S| + \sum_{s \in S} \deg_{out}(s))$. O DAG invertido $(V_{\text{DAG}}, \{(v,u) \mid (u,v) \in E_{\text{DAG}}\})$ também é armazenado como listas de adjacências de seus vértices e o custo para computar $in(S)$ é $O(|S| + \sum_{s \in S} \deg_{in}(s))$.

Em geral, construir um índice de alcançabilidade para um dígrafo estático pode ser decomposto em: (1) agrupar os componentes fortemente conectados e (2) construir um índice que ajude a responder consultas de alcançabilidade no DAG resultante. Relembramos aqui a definição de componentes fortemente conectados (chamados apenas de "componentes" neste texto) como consta na Definição 17.

Definição 17 (Componente Fortemente Conectado)**.** *Dados um dígrafo (V, E) e um vértice $u \in V$, o componente que inclui u, denotado $SCC(u)$ é $\{v \in V \mid u\ E^*\ v \wedge v\ E^*\ u\}$.*

Qualquer vértice em um componente pode ser escolhido como *representante* do componente. Formalmente, temos um subconjunto V_{DAG} de V e um mapeamento $r : V \to V_{\text{DAG}}$ tal que para todo $(u,v) \in V^2, SCC(u) = SCC(v) \Rightarrow \exists w \in SCC(u) \mid r(u) = r(v) = w$. A partir desse mapeamento r, o DAG (associado com o dígrafo), como exemplificado na Figura 4.1, é definido como:

Definição 18 (DAG Associado)**.** *Dado um dígrafo (V, E), o DAG associado (V, E_{DAG}) é $(V, \{(r(u), r(v)) \mid u\ E\ v\})$.*

Note que há distinção entre o conjunto de arestas do dígrafo E e o conjunto de arestas de seu DAG associado (E_{DAG}), mas não entre os vértices, uma vez que todos os vértices do dígrafo de entrada precisam ser mantidos em caso de uma atualização envolvendo-os. O uso da função r implementada como a estrutura *union-find* [Tarjan, 1975], é ideal para a manutenção de um particionamento de vértices (aqui SCC) e

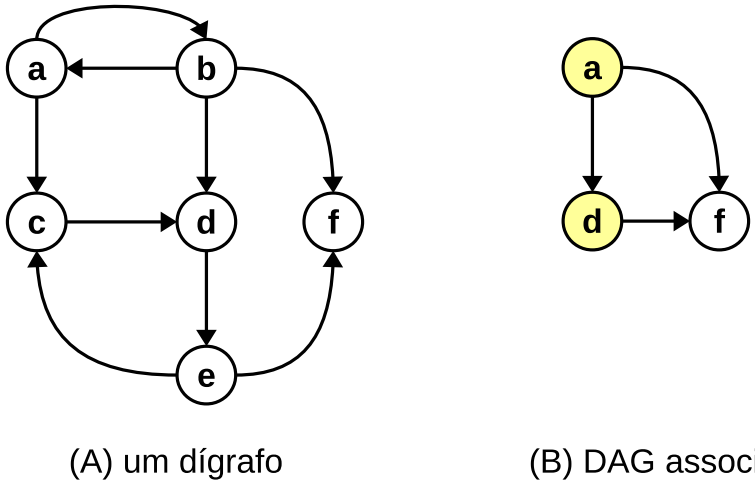


Figura 4.1. Em (A) um dígrafo e, em (B), o DAG associado onde $r(a) = r(b) = a$, $r(c) = r(d) = r(e) = d$ e $r(f) = f$.

fornecimento de um representante para toda partição. Os conjuntos E e E_{DAG} são implementados como *hashmaps* (as listas de adjacências de arestas normais e invertidas). No restante deste capítulo, todas as notações que são relacionadas aos vértices do DAG usam a função r de forma que, dado um vértice u, $r(u)$ pertence ao DAG e u ao dígrafo.

4.1.1 Feline

A extensão de Feline para grafos dinâmicos é aplicável a dígrafos genéricos (cíclicos ou não). Para tanto, considerando esse contexto, apresenta-se a seguir um resumo de Feline utilizando as novas notações, o que ajudará a compreender os algoritmos dinâmicos.

Feline é baseado em ordenações topológicas: dado um DAG (V, E_{DAG}), uma ordenação topológica $\prec$ é uma ordenação total de V tal que para todo par $(r(u), r(v)) \in E_{\text{DAG}}, r(u) \prec r(v)$. Uma consequência dessas definições é que, dado um DAG (V, E_{DAG}), sua ordenação topológica $\prec$ e dois vértices $u, v \in V^2, r(u)\ E^*_{\text{DAG}}\ r(v) \Rightarrow r(u) = r(v)$ ou existe $(r(u), r(k)) \in E_{\text{DAG}}$ tal que $r(k) \prec_= r(v)$ e $r(k)\ E^*_{\text{DAG}}\ r(v)$. Isso significa que todos os caminhos partindo de $r(u)$ até $r(v)$ (se existir algum) passam pelos vértices que estão entre $r(u)$ e $r(v)$ seguindo a ordenação topológica. Contudo, a ordenação topológica não é única e, como consequência, na presença de muitas ordenações topológicas de um dado DAG, precisamos considerar os vértices entre $r(u)$ e $r(v)$ na interseção de *todas* essas ordenações. Se essa interseção é vazia, então podemos responder de forma negativa a consulta (sem a necessidade de percorrer o DAG). Isso

acontece, por exemplo, quando $r(v)$ está antes de $r(u)$ em pelo menos uma das ordenações topológicas. O Algoritmo 6 constrói um índice Feline sobre um DAG associado.

Algoritmo 6: Constrói índice

```
1  constrói_indice(V, E_DAG)
2  início
       // Ordenação topológica por DFS
3      ≺_x ← TopologicalOrdering(V, E_DAG)
4      ≺_y ← ∅
5      heads ← (∅, ..., ∅)
6      d ← (0, ..., 0)
7      para todo (r(u), r(v)) ∈ E_DAG faça
8          heads_u ← heads_u ∪ {r(v)}
9          d_v ← d_v + 1
10     roots ← {r(v) | v ∈ V and d_v = 0}
11     while roots ≠ ∅ do
12         r(u) ← arg max_{r(v)∈ roots}(≺_x^{r(v)})
           // Adiciona r(u) ao final de ≺_y
13         ≺_y ← (≺_y, r(u))
           // Atualiza d e o conjunto roots
14         roots ← roots \{r(u)}
15         para todo r(v) ∈ heads_u faça
16             d_v ← d_v − 1
17             se d_v = 0 então
18                 roots ← roots ∪{r(v)}
19     retorna (≺_x, ≺_y)
```

Feline permite não somente responder consultas de forma rápida, mas também ajuda a encontrar um caminho entre dois vértices (se existir algum) ou retornar todos os vértices em pelo menos um dos caminhos entre dois vértices. Formalmente, dados dois vértices $(r(u), r(v))$ tal que $u, v \in V^2$, para retornar todos os vértices em pelo menos um dos caminhos entre eles, basta computar o conjunto $\{r(w) \mid w \in V$ e $r(u)$ E^*_{DAG} $r(w)$ e $r(w)$ E^*_{DAG} $r(v)\}$, denotado $P(r(u), r(v))$ no restante deste texto. Por exemplo, para o DAG da Figura 4.2, $P(c, i)$ retorna os vértices c, d, f, g, h, j, i.

O custo para computar $P(r(u), r(v))$ com Feline depende do tamanho do DAG reduzido entre $r(u)$ e $r(v)$ (o retângulo na Fig. 4.2-(B)), isto é, $(V_r, E_{\text{DAG\_}r}) = (\{r(w) \mid w \in V$ e $r(u) \prec_x r(w) \prec_x r(v)$ e $r(u) \prec_y r(w) \prec_y r(v)\}, E_{\text{DAG}} \cap {V_r}^2)$. Isso é $O(|V_r| + |E_{\text{DAG\_}r}| + \sum_{s \in V_r} \deg_{out}(s))$. O termo $\sum_{s \in V_r} \deg_{out}(s)$ vem das tentativas de seguir os caminhos que vão além do retângulo mostrado na Figure 4.2.

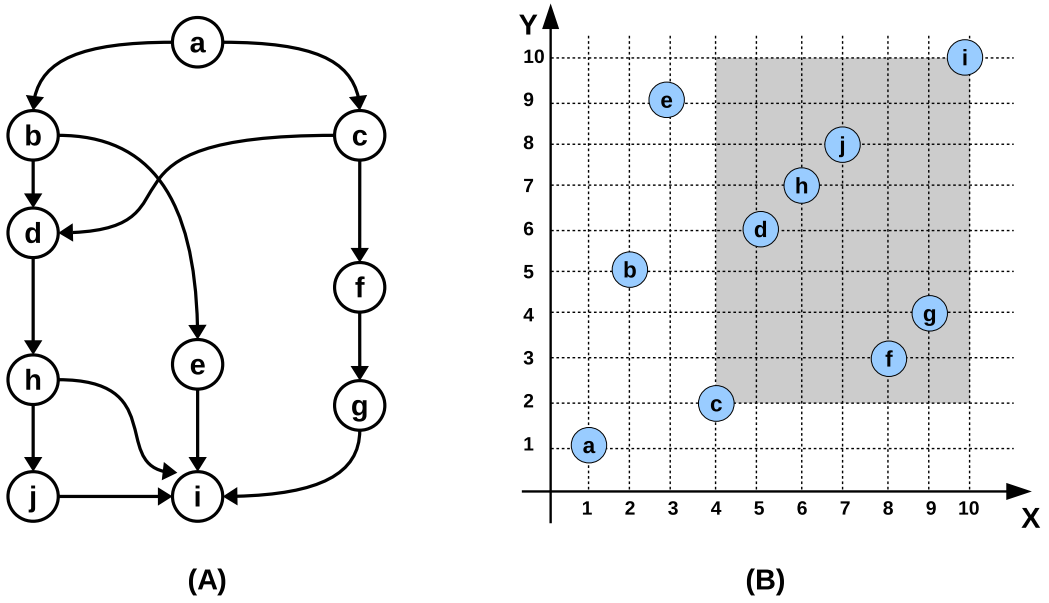


Figura 4.2. (A) DAG e (B) índice Feline. O retângulo mostra a região que pode ser percorrida na tentativa de responder positivamente à consulta: *c alcança i ?*.

De forma a, incrementalmente, atualizar o índice Feline, precisamos atualizar as ordenações topológicas. Neste capítulo, para facilitar a apresentação, consideraremos que o índice é sempre construído sobre o DAG associado ao dígrafo corrente.

4.1.2 Algoritmo PK

O algoritmo PK, cujo nome é uma referência aos autores Pearce & Kelly [2006], tem como base o fato de que, dado um DAG (V, E_{DAG}), quando uma nova aresta $(r(u), r(v))$ s.t. $u, v \in V^2$ é inserida e essa aresta invalida a ordenação topológica, uma nova ordem topológica do grafo é obtida por:

1. uma ordenação topológica do sub-DAG com os vértices em $\delta = \{r(k) \mid k \in V \text{ e } r(v) \prec_= r(k) \prec_= r(u) \text{ e } (r(v)\ E^*_{\text{DAG}}\ r(k) \text{ ou } r(k)\ E^*_{\text{DAG}}\ r(u))\}$;

2. permutando em $\prec$ os vértices em δ com relação à ordem encontrada no passo anterior (os vértices em $V \setminus \delta$ mantêm suas posições).

De forma mais específica, $\delta = \delta^{r(v)}_{out} \cup \delta^{r(u)}_{in}$, onde $\delta^{r(u)}_{in} = \{r(w) \mid w \in V \text{ e } r(v) \prec r(w) \text{ e } r(w)\ E^*_{\text{DAG}}\ r(u)\}$ e $\delta^{r(v)}_{out} = \{r(w) \mid w \in V \text{ e } r(w) \prec r(u) \text{ e } r(v) E^*_{\text{DAG}} r(w)\}$. Isto é, δ é formado por todos os vértices entre $r(u)$ e $r(v)$ que alcançam $r(u)$ (i. e.$\delta^{r(u)}_{in}$) e todos os vértices que são alcançáveis por $r(v)$ (i. e., $\delta^{r(v)}_{out}$ contém todos os vértices que são alcançáveis a partir de $r(v)$).

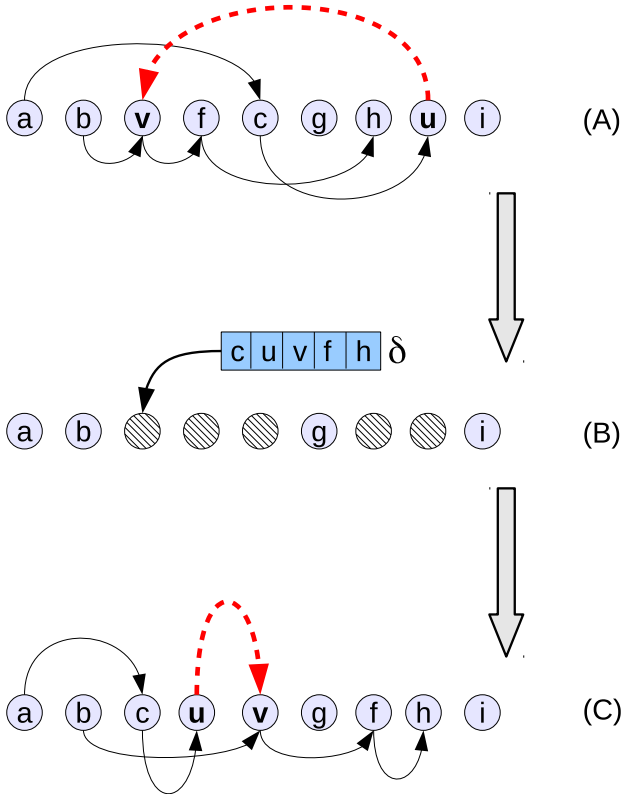


Figura 4.3. Exemplo 4.1.2: em (A) o conjunto δ é identificado e os conjuntos δ_{out} e δ_{in} são computados; em (B) o conjunto δ é reorganizado; em (C) a nova ordem topológica.

Notamos com isso que $\delta = \emptyset$ somente quando $r(u)$ e $r(v)$ já estão em posições corretas na ordem topológica (i. e., $r(u) \prec r(v)$). Mais ainda, nenhum membro de $\delta_{out}^{r(v)}$ alcança qualquer membro de $\delta_{in}^{r(u)}$ sem introduzir um ciclo no DAG. Uma representação gráfica para δ é mostrada na Figura 4.3.

Exemplo 1 - reordenando o conjunto δ Suponha uma ordem topológica como mostrada na Figura 4.3-(A). A inserção de uma aresta (u, v) invalida a ordem topológica, uma vez que $v \prec u$. Assim, como $\delta_{out}^{v} = \{v, f, h\}$ e $\delta_{in}^{u} = \{c, u\}$, segue-se que $\delta = \{c, u, v, f, h\}$. Esses são os vértices no conjunto a ser reordenado, o que significa que na nova ordem, depois de atualizar o grafo, os vértices em δ serão reposicionados usando como referência somente as suas posições anteriores. Portanto, todos os vértices restantes não são afetados.

A Figura 4.3 apresenta o conjunto δ após a reordenação de seus vértices tal que todos os vértices em $\delta_{in}^{r(u)}$ precedem aqueles em $\delta_{out}^{r(v)}$. Note que nenhuma posição topológica extra foi criada e nenhum vértice fora de δ foi alterado (como o vértice g

na figura). Com isso, o seguinte invariante é mantido: $\forall r(w) \in \delta_{out}^{r(v)}, r(v) \prec_= r(w)$ e $\forall r(w) \in \delta_{in}^{r(u)}, r(w) \prec_= r(u)$.

A complexidade de tempo de PK é $O(|\delta| \log |\delta| + \|\delta\|)$, onde $\|\delta\|$ denota a quantidade de vértices de δ e seus vizinhos, e $|\delta| \log |\delta|$ é o tempo para permutar os vértices em δ. É importante notar que Pearce & Kelly [2006] não descreveram uma solução para o caso em que a nova aresta $(r(u), r(v))$ cria um ciclo, i.e., o caso onde $r(v)\ E^*_{\mathrm{DAG}}\ r(u)$. Contudo, consideramos esse caso neste trabalho.

4.2 Atualização Dinâmica do Índice Feline

Detalhamos aqui o processo de atualização do índice Feline em um contexto dinâmico. Para tanto, descrevemos quatro algoritmos que realizam a inserção/remoção de um vértice desconectado (Algs. 7 e 8) e a inserção/remoção de uma aresta (Algs. 10 e 11). Devido a impossibilidade de ordenar topologicamente os vértices de um grafo cíclico, o índice $(\prec_x, \prec_y)$ é obtido do DAG (V, E_{DAG}) associado com o dígrafo (V, E) por meio de uma função denominada *representante* r (Veja a Def. 18). Em resumo, seis variáveis, $\prec_x, \prec_y, E_{\mathrm{DAG}}, V, E$ e r, são utilizadas pelos algoritmos recebendo-as como argumentos e retornando-as atualizadas.

4.2.1 Inserção de um Vértice Desconectado

O Algoritmo 7 insere um vértice desconectado u em tempo constante, dessa forma u é adicionado a V (linha 3) e mapeado para ele mesmo através de r (linha 4). De fato, sendo desconectado do resto do dígrafo, u é o único representante possível para seu componente, $\{u\}$. Considerando o índice, $\prec_x$ e $\prec_y$ devem continuar válidos independentemente das posições em que u venha a ser inserido. De fato, como nenhuma aresta está associada a u ainda, u é inserido no final de $\prec_x$ (linha 5) e no início de $\prec_y$ (linha 6), i. e., no canto inferior-direito do plano que representa o índice. Esta é uma boa escolha: até que uma aresta conecte u a outro vértice, qualquer consulta de alcançabilidade envolvendo u seria devidamente respondida de forma negativa sem qualquer travessia no DAG (V, E_{DAG}).

4.2.2 Remoção de um Vértice Desconectado

O Algoritmo 8 remove um vértice desconectado u em tempo constante. Esse algoritmo segue os passos do Alg. 7 porém em ordem inversa: u é removido de V (linha 3), seu mapeamento através de r é removido (linha 4), e é removido de $\prec_x$ (linha 5) e

Algoritmo 7: insere um vértice desconectado

```
1 insere_vértice_desconectado(≺_x, ≺_y, E_DAG, V, E, r, u)
2 início
3 |  V ← V ∪ {u}
4 |  r(u) ← u
5 |  insere u no fim de ≺_x
6 |  insere u no início de ≺_y
7 |  retorna (≺_x, ≺_y, E_DAG, V, E, r)
```

$\prec_y$ (linha 6). Nada mais é necessário: sendo u desconectado do restante do dígrafo, ele necessariamente representa seu componente $\{u\}$ (e nenhum outro vértice) e suas posições em $\prec_x$ e $\prec_y$ são irrelevantes.

Algoritmo 8: remove um vértice desconectado

```
1 remove_vértice_desconectado(≺_x, ≺_y, E_DAG, V, E, r, u ∈ V)
2 início
3 |  V ← V \ {u}
4 |  apaga r(u) = u de r
5 |  apaga u de ≺_x
6 |  apaga u de ≺_y
7 |  retorna (≺_x, ≺_y, E_DAG, V, E, r)
```

4.2.3 Inserção de uma Aresta

A inserção de uma aresta (u, v) exige alguns passos a mais, como apresentado no Alg. 10. Quando $r(v)$ não alcança $r(u)$, $(V, E_{\text{DAG}} \cup (r(u), r(v)))$ permanece um DAG, SCC não é alterado e ambas as ordens topológicas $\prec_x, \prec_y$ são atualizadas com Feline. O caso mais complicado para a inserção é portanto associado com $r(v)$ E^* $r(u)$. Nesse caso, SCC sofre alterações. Contudo, não precisamos executar o algoritmo de Tarjan para encontrar os componentes fortemente conectados sempre que isso acontece.

Ao invés do teste $r(v)$ E^* $r(u)$, o conjunto de vértices $V_{\text{cycles}} = P(r(v), r(u))$ é computado no tempo $O(|V_r| + |E_{\text{DAG\_}r}| + \sum_{s \in V_r} \deg_{out}(s))$ como explicado na Sec. 4.1. Obviamente temos $r(v)$ E^* $r(u)$ se e somente se V_{cycles} não é vazio. Neste caso, SCC pode ser alterado de uma única forma: muitos componentes do SCC atual são combinados em um único componente. O novo componente contém os vértices em V, que são representados por algum vértice em V_{cycles}, i. e., os vértices no caminho de $r(v)$ até $r(u)$ no DAG associado. Qualquer vértice em tal caminho pode atuar como o novo

representante para o componente que foi recentemente agrupado. Mais precisamente, a atualização de r pode ser realizada de acordo com o Algoritmo 9.

Algoritmo 9: Atualiza r

```
1 atualiza_r(V_cycles, r)
2 início
3 |  escolhe r' em V_cycles
4 |  para todo w ∈ V_cycles faça
5 |  |  r(w) ← r'
6 |  retorna (r, r')
```

Note o leitor, no entanto, que na implementação para a computação de V_{cycles}, a escolha de r' (o novo representante do componente) e a atualização de r são realizadas de forma eficiente, uma vez que r é armazenado em uma estrutura union-find [Tarjan, 1975]. Graças a isso e ao Feline, a atualização de r depois da inserção de uma aresta no dígrafo possui complexidade de tempo de pior caso igual a $O(|V_r| \log^* |V_r|)$, onde $\log^*$ é o número de vezes que devemos iterar a função log em $|V_r|$ antes de obtermos um número menor ou igual a 1. Por exemplo, $\log_2^* 65536 = 4$ pois $\log_2 65536 = 16$, $\log_2 16 = 4$, $\log_2 4 = 2$ e $\log_2 2 = 1$. Note que a função $\log^*$ cresce de forma muito lenta.

A atualização das duas ordens topológicas $\prec_x$ e $\prec_y$ é realizada pelo Algoritmo 10, que conclui a atualização pelo reuso de algoritmos apresentados anteriormente. Para que isso aconteça, precisamos determinar o sub-DAG com os vértices que serão permutados em $\prec_x$ e $\prec_y$. Esse sub-DAG $(\delta, E^2_{\text{DAG}} \cap \delta^2)$, onde δ é computado na linha 11, é submetido ao Feline (Alg. 3), que retorna as novas ordens topológicas $(\prec'_x, \prec'_y)$. As linhas 14 e 15 organizam os vértices em $\prec_x$ e $\prec_y$ de acordo com $\prec'_x$ e $\prec'_y$, respectivamente. A linha 5 requer um tempo $O(|V_r| + |E_{\text{DAG\_}r}| + \sum_{s \in V_r} \deg_{out}(s))$ e as linhas 6-9 requerem um tempo $O(|V_r| \log |V_r| + \|\delta\|)$ como explicado na Sec. 4.1. A complexidade da linha 17 requer $O(|V_{\text{cycles}}|)$ de acordo com o Alg. 9, e as linhas 18-20 requerem $O(\sum_{s \in V_{\text{cycles}}} \deg(s) + |in(V_{\text{cycles}})| + |out(|Vcycles)|)$, logo, requer $O(\sum_{s \in V_{\text{cycles}}} \deg(s))$. As linhas 21 e 22 requerem $O(|\delta \setminus V_{\text{cycles}}|)$, uma vez que elas mudam somente as posições dos vértices restantes. Portanto, a complexidade final do Algoritmo 10 é $O(|V_r| \log |V_r| + |E_{\text{DAG\_}r}| + \sum_{s \in V_r} \deg(s))$.

Exemplo 2 - atualizando o índice: SCC não alterado Considerando o DAG e seu índice mostrados na Fig. 4.4, suponha a inserção de uma nova aresta (j, f) como indicado na Fig. 4.4-(A). De acordo com o Alg. 10, $V_{\text{cycles}} = \emptyset$, pois essa nova aresta não cria um ciclo. A união dos conjuntos $\delta^{r(j)}_{in\_x} = \emptyset$, $\delta^{r(f)}_{out\_x} = \emptyset$, $\delta^{r(j)}_{in\_y} = \{d, h, j\}$ e

Algoritmo 10: insere aresta

1 **insere_aresta**($\prec_x, \prec_y, E_{\mathbf{DAG}}, V, E, r, (u,v)$)
2 **início**
3 $E \leftarrow E \cup \{(u,v)\}$
4 **se** $r(u) \neq r(v)$ **então**
5 $V_{\text{cycles}} \leftarrow P(r(v), r(u))$
6 $\delta^{r(u)}_{in\_x} \leftarrow \{w \in V \mid r(v) \prec_x w \wedge wE^*_{\text{DAG}} r(u)\}$
7 $\delta^{r(v)}_{out\_x} \leftarrow \{w \in V \mid w \prec_x r(u) \wedge r(v)E^*_{\text{DAG}} w\}$
8 $\delta^{r(u)}_{in\_y} \leftarrow \{w \in V \mid r(v) \prec_y w \wedge wE^*_{\text{DAG}} r(u)\}$
9 $\delta^{r(v)}_{out\_y} \leftarrow \{w \in V \mid w \prec_y r(u) \wedge r(v)E^*_{\text{DAG}} w\}$
10 **se** $V_{cycles} = \emptyset$ **então**
`// SCC não alterado`
11 $\delta \leftarrow \delta^{r(u)}_{in\_x} \cup \delta^{r(v)}_{out\_x} \cup \delta^{r(u)}_{in\_y} \cup \delta^{r(v)}_{out\_y}$
12 $E_{\text{DAG}} \leftarrow E_{\text{DAG}} \cup \{(r(u), r(v))\}$
13 $(\prec'_x, \prec'_y) \leftarrow$ constrói_índice$(\delta, E_{\text{DAG}} \cap \delta^2)$
14 permuta em $\prec_x$ os vértices em δ de acordo com $\prec'_x$
15 permuta em $\prec_y$ os vértices em δ de acordo com $\prec'_y$
16 **senão**
`// SCC alterado`
17 $(r, r') \leftarrow$atualiza_r(V_{cycles}, r)
18 $I \leftarrow in(V_{\text{cycles}})$
19 $O \leftarrow out(V_{\text{cycles}})$
20 $E_{\text{DAG}} \leftarrow (E_{\text{DAG}} \setminus (I \times V_{\text{cycles}} \cup V_{\text{cycles}} \times O)) \cup (I \times \{r'\} \cup \{r'\} \times O)$
21 move r' em $\prec_x$ entre $\max_{\prec_x}(\delta^{r(u)}_{in\_x} \setminus V_{\text{cycles}})$ e $\min_{\prec_x}(\delta^{r(v)}_{out\_x} \setminus V_{\text{cycles}})$
22 move r' em $\prec_y$ entre $\max_{\prec_y}(\delta^{r(u)}_{in\_y} \setminus V_{\text{cycles}})$ e $\min_{\prec_y}(\delta^{r(v)}_{out\_y} \setminus V_{\text{cycles}})$
23 **retorna** $(\prec_x, \prec_y, E_{\text{DAG}}, V, E, r)$

$\delta^{r(f)}_{out\_y} = \{f, g\}$ é destacada na Fig. 4.4-(B), na qual compõe o sub-DAG $(\delta, E_{\text{DAG}} \cap \delta^2)$ computado nas linhas 11 e 12 do Algoritmo 10. Depois que Feline reorganiza as ordens topológicas, os vértices δ em $\prec_x$ e $\prec_y$ são recolocados em suas posições apropriadas. O índice resultante é mostrado na Figura 4.5.

Exemplo 3 - atualizando o índice: SCC alterado Considere agora o DAG e seu índice mostrado na Fig. 4.6. Suponha que uma nova aresta (h, b) é inserida. A Figura 4.6-(A) mostra o DAG e a nova aresta, onde $V_{\text{cycles}} = \{b, d, h\}$ está em destaque. Seguindo o Alg. 10, temos os conjuntos $\delta^{r(h)}_{in\_x} = \{c, d, h\}$, $\delta^{r(b)}_{out\_x} = \{b, e, d\}$, $\delta^{r(h)}_{in\_y} = \{d, h\}$ e $\delta^{r(b)}_{out\_y} = \{b, d\}$ como representados na Figura 4.7-(A). Na linha 17, o novo vértice representativo é eleito, e.g., o vértice d. Nas linhas 18, 19 e 20, removemos as arestas

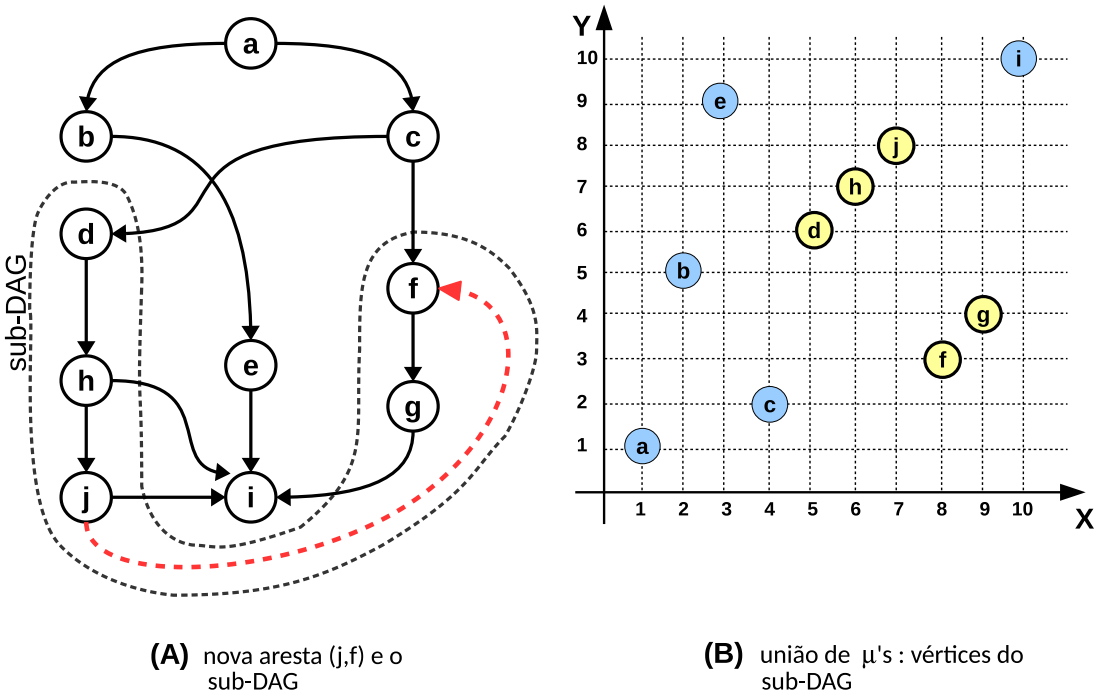


Figura 4.4. Exemplo 23: caso "SCC não alterado". Esta figura mostra em (A) o DAG, a nova aresta (f, j) e o sub-DAG obtido de δ, em (B) a área está em destaque com os vértices do sub-DAG.

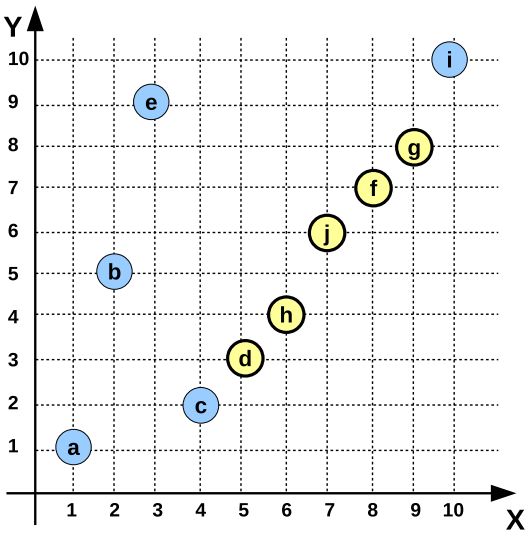


Figura 4.5. Exemplo 23: novo índice. Os vértices indexados por Feline e recolocados em posições corretas estão em destaque.

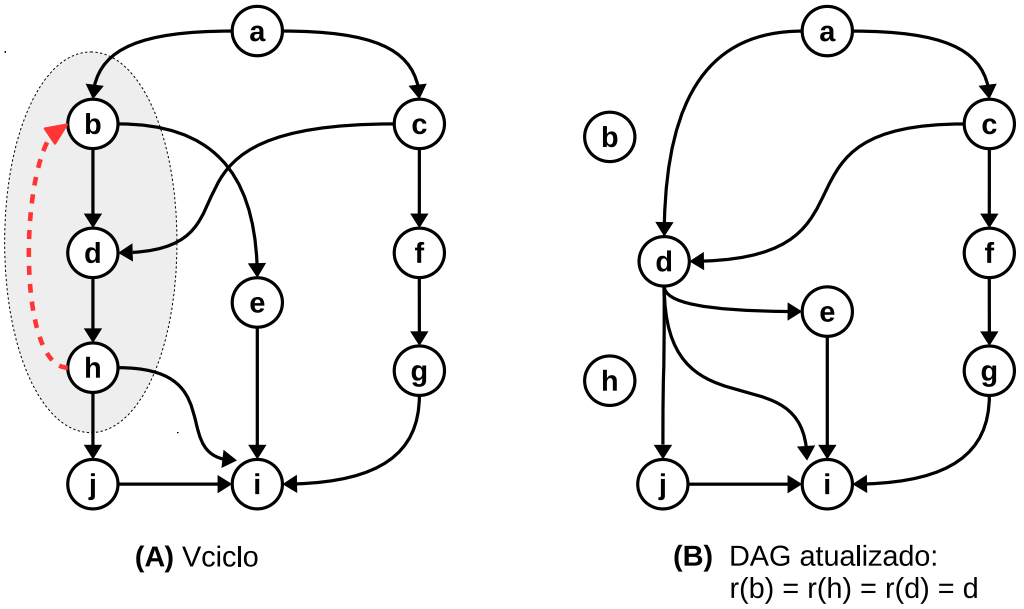


Figura 4.6. Exemplo 23: em (A) os vértices em V_{cycles}. Em (B), o DAG atualizado (arestas em E_{DAG}).

(a,b), (b,d), (b,e), (c,d), (d,h), (h,i) e (h,j), e então inserimos (a,d), (c,d), (d,e), (d,i) e (d,j). Os vértices que não são representantes são sempre desconectados do DAG (veja a Definição 18) e podem estar em qualquer posição nas ordens topológicas. De fato, as ordens antigas são mantidas uma vez que elas são, provavelmente, melhores que aquelas atribuídas pelo Alg. 7, i. e., se o vértice é eleito como representante novamente (depois de uma remoção de aresta), ele pode ser colocado próximo à sua antiga posição e a atualização terá custo menor. Em seguida, o índice é atualizado nas linhas 21 e 22. Na linha 21, $\max_{\prec_x}(\delta^{r(h)}_{in\_x} \setminus V_{\text{cycles}}) = \{c\}$ e $\min_{\prec_x}(\delta^{r(b)}_{out\_x} \setminus V_{\text{cycles}}) = \{e\}$, então na nova $\prec_x$ teremos $c \prec_x d \prec_x e$. A linha 22 não resultará em modificações em $\prec_y$ pois d já está em sua posição correta. A Figura 4.7-(B) apresenta o índice resultante.

Pearce & Kelly [2006] demonstraram que a atualização da ordem topológica ($\prec_x$ ou $\prec_y$) no caso de "SCC não alterado" requer somente permutar os vértices em δ, mantendo válidas as ordenações topológicas. Essa estratégia faz uso do Alg. 3. É preciso demonstrar que todas as arestas no DAG atualizado têm suas direções concordando com a ordem topológica ("forward edges") $\prec_x$ (a demonstração para $\prec_y$ é idêntica). Existem os seguintes casos:

1. Comparação entre dois vértices conectados, nenhum deles sendo r': a comparação é trivial.

2. Comparação entre dois vértices conectados, um deles sendo r'. Como eles são

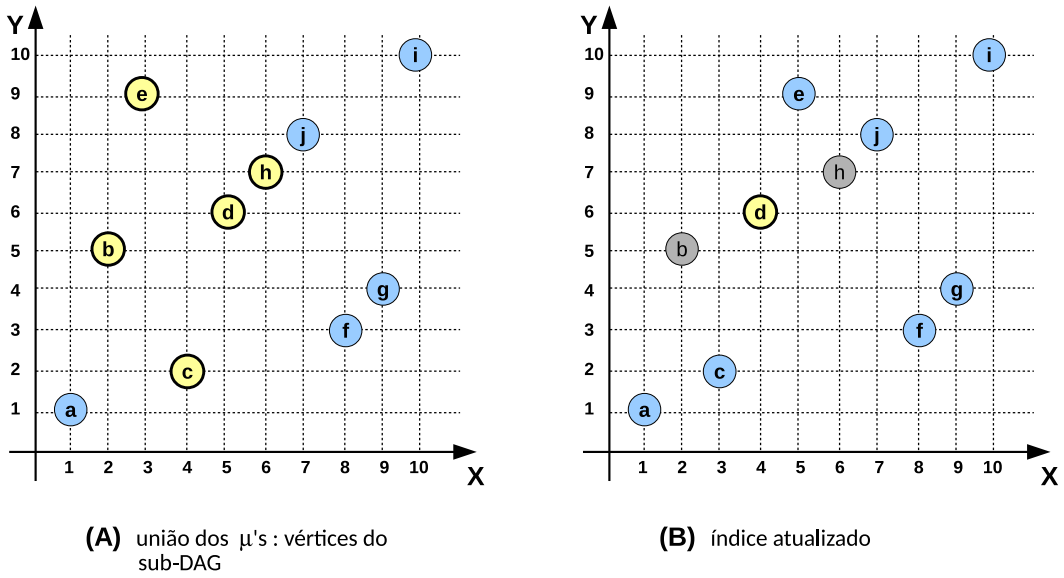


Figura 4.7. Exemplo 23: novo índice após o agrupamento $\{b, d, h\}$. Em (B), o representante d e os vértices b e h fora do índice.

conectados, o outro vértice está em $in(\{r'\})$ ou em $out(\{r'\})$:

a) O outro vértice está em $in(\{r'\})$. Ele não pode estar depois de $r(u)$ na ordenação original (caso contrário a ordem estaria incorreta). Dada a definição de $\delta_{in}^{r(u)}$, o outro vértice está antes de $r(v)$ ou em $\delta_{in}^{r(u)}$. Em ambos os casos, r' está depois na nova ordenação.

b) O outro vértice está em $out(\{r'\})$. Ele não pode estar antes de $r(v)$ na ordenação original (caso contrário a ordenação estaria incorreta). Dada a definição de $\delta_{out}^{r(v)}$, o outro vértice está depois de $r(u)$ ou em $\delta_{out}^{r(v)}$. Em ambos os casos, r' está antes na nova ordenação.

4.2.4 Remoção de uma aresta

O Algoritmo 11 remove uma aresta (u, v) do dígrafo, i. e., de E (linha 3). Os dois vértices u e v estão no mesmo componente (o teste na linha 4 passa) ou em componentes diferentes (o teste na linha 4 falha). O último caso é fácil de entender: uma vez que (u, v) conecta dois componentes, esses componentes não são modificados, i. e., r não é alterado. O que pode (ou não) alterar E_{DAG}. De fato, (u, v) refere-se a aresta $(r(u), r(v))$ no DAG associado (V, E_{DAG}). A aresta $(r(u), r(v))$ pode (ou não) ter sido removida de E_{DAG}. De acordo com a Def 18, isso depende de E ainda possuir uma aresta que parte do componente de u para o componente de v. Se possuir (o

teste na linha 12 falha), $(r(u), r(v))$ permanece em E_{DAG}. Caso contrário (o teste passa), $(r(u), r(v))$ é removido de E_{DAG} (linha 13). Pelo motivo de qualquer ordenação topológica de um DAG permanecer válida depois de uma remoção de aresta, nem $\prec_x$ e nem $\prec_y$ sofrem mudanças.

Algoritmo 11: remove aresta

```
1  remove_aresta(≺_x, ≺_y, E_DAG, V, E, r, (u, v))
2  início
3      E ← E \ {(u, v)}
4      se r(u) = r(v) então
5          C ← {w ∈ V | r(w) = r(u)}
6          se ¬(u(E ∩ C²)*v) então
7              E_DAG ← E_DAG \ (V × {r(u)} ∪ {r(u)} × V)
8              r ← Tarjan(C, E ∩ C², r)
9              para todo (w, k) ∈ E ∩ (V × C ∪ C × V) faça
10                 insere_aresta(≺_x, ≺_y, E_DAG, V, E, r, (w, k))
11     senão
12         se para todo (a, b) ∈ E, (r(a), r(b)) ≠ (r(u), r(v)) então
13             E_DAG ← E_DAG \ {(r(u), r(v))}
14     retorna (≺_x, ≺_y, E_DAG, V, E, r)
```

Considera-se agora o caso mais complicado, onde u e v pertenciam ao mesmo componente (o teste da linha 4 passa). Primeiro, esse componente, um conjunto C de vértices é computado (linha 5). Uma travessia no dígrafo então inicia em u para descobrir se v continua alcançável apesar da remoção de (u, v) (linha 3). Essa travessia (em DFS) é seguramente abortada assim que ela deixa o conjunto C de vértices (porque C era um componente, nenhum caminho de u até v pode passar por vértices em $V \setminus C$). Se v é alcançável (o teste na linha 6 falha), então o componente fica como está, i. e., r não é alterado, bem como o DAG associado (V, E_{DAG}) e o índice relacionado $(\prec_x, \prec_y)$. Ao contrário, se a remoção de (u, v) agora impede que u alcance v (o teste na linha 6 passa), então todas as variáveis precisam ser atualizadas. De acordo com a Def. 17, u e v agora estão em diferentes componentes. De fato, C pode agora ter que ser particionado em qualquer número de componentes entre 2 e $|C|$. Dessa forma, r precisa refletir essa alteração e o DAG associado (V, E_{DAG}) também precisa ser modificado. Antes de conectar os novos vértices representantes (ou mesmo decidir qual vértice é um novo representante), o vértice que representava todos os vértices em C é desconectado do resto do DAG, i. e., todas as arestas envolvendo-o são removidas de E_{DAG} (linha 7). O algoritmo de Tarjan é responsável por identificar o particionamento de C e compor

os novos componentes menores (linha 8). Ele não precisa considerar os vértices fora de C pois a remoção de $(u, v) \in C^2$ não afeta seus componentes. Consideramos aqui que o algoritmo de Tarjan também escolhe um representante para cada componente, i. e., atualiza r. Todas as arestas envolvendo um vértice em C (linha 9) são então reinseridas. Para facilitar a apresentação, a função *insere_ aresta* é chamada (linha 10). No entanto, o trabalho a ser feito por ela é reduzido pois sabemos que não há nenhum novo ciclo.

Quanto à complexidade de *remove_ aresta*, quando u e v estão em diferentes componentes, o custo do teste na linha 12 domina, mas é apenas $O(|SCC(u)|)$. Isso é possível graças à estrutura *union-find* e a escolha da implementação das listas de adjacências como *hashmaps*. Quando u e v estão no mesmo componente, esse componente é recuperado em tempo $O(|C|)$, graças novamente à estrutura *union-find*. Ele é percorrido (por DFS) em tempo $O(\sum_{c \in C} \deg_{out}(c))$, que é um custo alto e depende dos graus de saída ao invés de $|E \cap C^2|$ devido às tentativas de deixar o componente. Esse é o custo de pior caso para a atualização se o componente não for dividido. Caso contrário, o custo é mais alto, pois ele depende do conjunto M de componentes encontrados pelo algoritmo de Tarjan. O algoritmo de Tarjan, por ele mesmo, executa em um tempo que é $O(\sum_{c \in C} \deg_{out}(c))$. O número de arestas que precisam ser reinseridas para atualizar o índice é $(\sum_{m \in M}(|\text{in}(m)|+|\text{out}(m)|))$, portanto, o algoritmo executa em tempo $O(\sum_{m \in M}((|\text{in}(m)| + |\text{out}(m)|) \times I_{wk}))$, onde I_{wk} é o tempo para inserir a aresta (w, k) usando o Alg. 10. No entanto, embora escrevemos *insere_ aresta*, a função que é realmente executada não precisa computar $P(r(v), r(u))$ (na linha 5 de Alg. 10) pois já sabemos que a aresta reintroduzida não cria ciclo (graças ao algoritmo de Tarjan). Em outros termos, toda inserção sempre cai no caso "SCC não alterado" (nenhum teste é necessário na linha 10).

Exemplo 4 - removendo uma aresta: separando o componente Considere outro DAG da Fig. 4.8-(A), onde o vértice b é o representante do componente, como mostrado na Fig. 4.8-(B). Suponha que a aresta (l, d) é removida (de dentro do componente representado por b como mostrado na Fig. 4.8-(B)). Se este é o caso, l pode não mais alcançar d, significando que o componente foi dividido. Então, precisamos remover do DAG todas as arestas que conectam o componente representado por b, i. e., as arestas (a, b), (c, d), (k, e), (h, i) e (h, j) da Fig. 4.8-(A). Os novos componentes encontrados são mostrados na Fig. 4.9-(A), onde todas as arestas de cada vértice nos novos componentes são restauradas e os representantes b, d e h escolhidos (algoritmo de Tarjan's modificado na linha 8 do Alg. 11). Finalmente, as arestas (a, b), (c, d), (k, e), (h, i), (h, j) e (d, h) são reinseridas, uma a uma, para atualizar o índice.

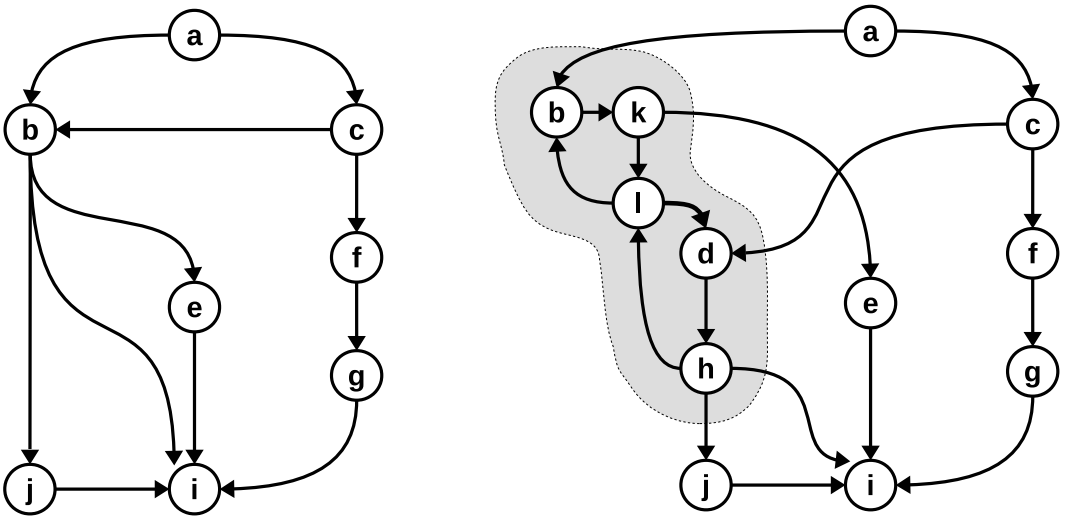


Figura 4.8. Exemplo 14: Em (A), o DAG e, em (B), todos os vértices do componente representado por b são mostrados.

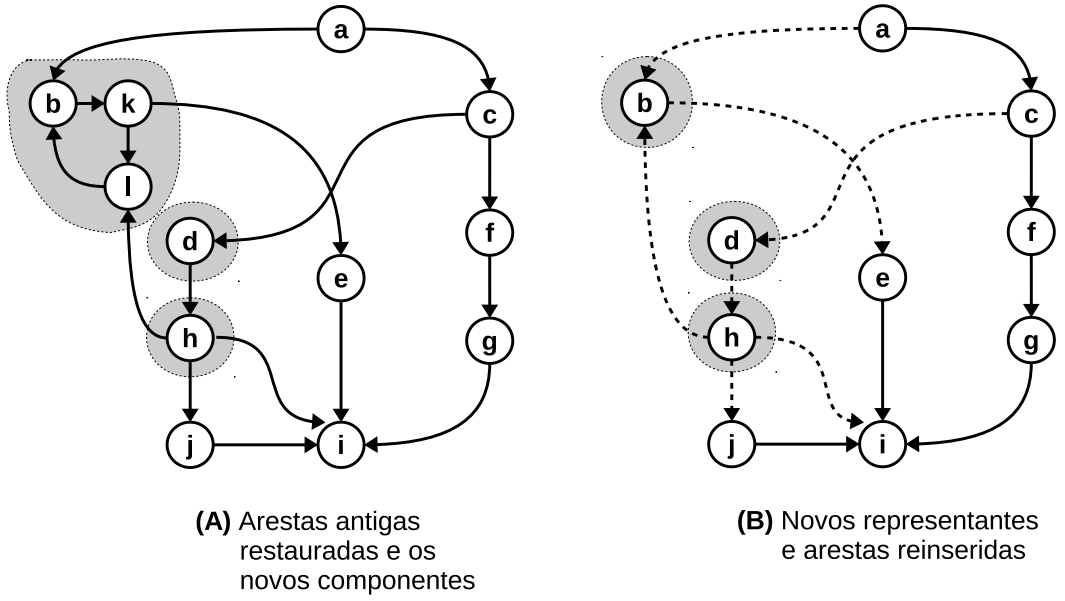


Figura 4.9. Exemplo 14: Em (A) a aresta (l, d) removida, os novos componentes identificados e as antigas arestas restauradas. Em (B), os componentes são agrupados (em inglês, "folded") e as arestas (a, b), (c, d), (k, e), (h, i), (h, j) e (d, h) reinseridas (linhas pontilhadas).

4.3 Experimentos

Esta seção avalia o desempenho de Feline-PK comparado à abordagem do estado-da-arte. Implementou-se os algoritmos usando linguagem C++, tal como as implementações do competidor fornecidas pelos respectivos autores. Todos os experimentos foram realizados em uma máquina Intel(R) Core(TM) i5-3330 CPU 3.00GHz e 8GB RAM, com sistema operacional GNU/Linux. Todas as implementações são *single threaded*.

Grafo	$\|V\|$	$\|E\|$	**cíclico?**
Arxiv	6000	66707	não
Yago	6642	42392	não
Go	6793	13361	não
Pubmed	9000	40028	não
citeseer	10720	44258	não
p2p-Gnutella31	62586	147892	sim
soc-Epinions1	75879	508837	sim
Uniprot22mA	100000	99998	não
Cit-PatentsA	100000	88520	não
CiteseerxA	100000	152544	não
Go-UniprotA	100000	582298	não
Email-EuAll	265214	420045	sim
Uniprot22mB	300000	299998	não
Cit-PatentsB	300000	297686	não
CiteseerxB	300000	533289	não
Go-UniprotB	300000	2168728	não
web-NotreDame	325729	1497134	sim
Uniprot22m	1595444	1595442	não
Cit-patents	3774768	16518947	não
Go-uniprot	6967956	34770235	não

Tabela 4.1. Bases de Dados

Bases de dados: Usou-se um conjunto de DAGs tradicionais (benchmark) usados em diversos trabalhos como [Bramandia et al., 2010; Yildirim et al., 2010, 2013; Veloso et al., 2014; Zhu et al., 2014], e quatro dígrafos (cíclicos) coletados de snap.stanford.edu, que são: p2p-Gnutella31, soc-Epinions1, Email-EuAll e web-NotreDame. Os DAGs *Go-UniProt*, *Cit-Patents*, *CiteSeerX* e *Uniprot22m* estão duplicados e diferem entre si pelo número de vértices, onde um conjunto tem 100k vértices e o outro tem 300k vértices, aleatoriamente selecionados. Esses grafos estão na Tabela 4.1.

Primeiro foi necessário verificar se usar Feline-PK é melhor que reconstruir o índice do princípio sempre que um grafo é atualizado. Assim, repetiram-se os experimentos de Feline (estático), que foram comparados com os tempos médios de atualização

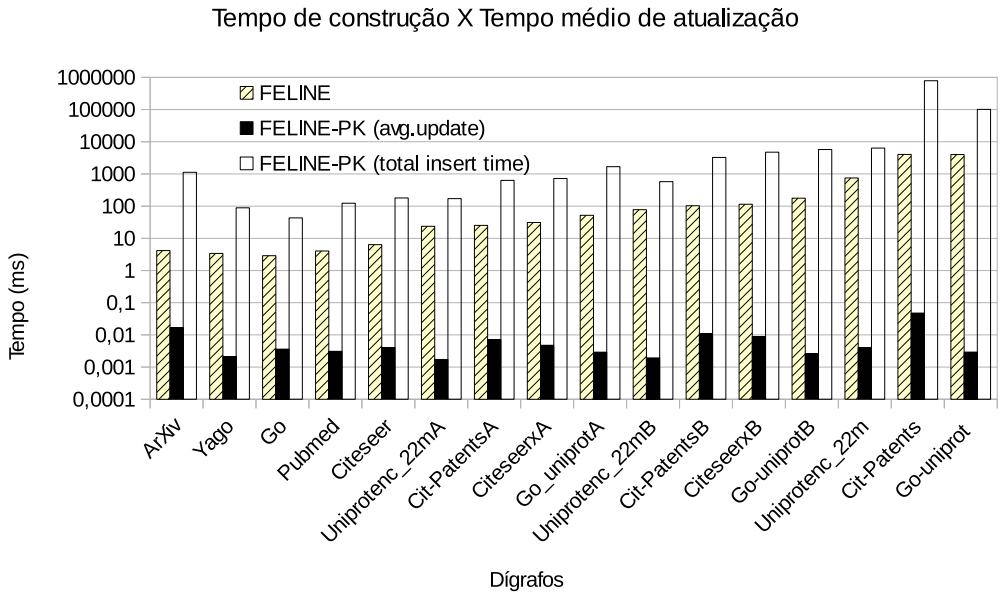


Figura 4.10. Comparação entre Feline (tempos de construção) e Feline-PK (tempos de atualização). Esta figura mostra que inserir uma aresta e atualizar o índice usando Feline-PK resulta em melhor desempenho do que reconstruir o índice com uma abordagem estática. A terceira barra é o tempo de inserção total para todas as arestas com Feline-PK.

após a inserção de arestas com Feline-PK. Após isso, comparou-se também os resultados de Feline-PK com a abordagem do estado-da-arte, verificando seus tempos de atualização quando há inserção e remoção de arestas, como também os tempos para responder às consultas. O terceiro experimento mostra a proporção de respostas negativas fornecidas sem qualquer caminhamento nos grafos. Acredita-se, portanto, que esses experimentos permitem dizer que Feline-PK é a melhor solução. Alguns resultados não foram reportados devido aos seus tempos de execução serem maiores do que 5 horas.

4.3.1 Resultados

A Figura 4.10 apresenta uma comparação entre a solução estática Feline (construção do índice) e a dinâmica Feline-PK (atualizações após cada inserção). Os tempos médios de atualização são obtidos após as arestas serem inseridas no grafo, uma a uma, de forma aleatória. Como pode-se ver na figura, o tempo para atualizar o índice dada uma inserção de aresta é menor que reconstruir o índice inteiro. A terceira barra no gráfico mostra o tempo total para a inserção de todas as arestas usando Feline-PK.

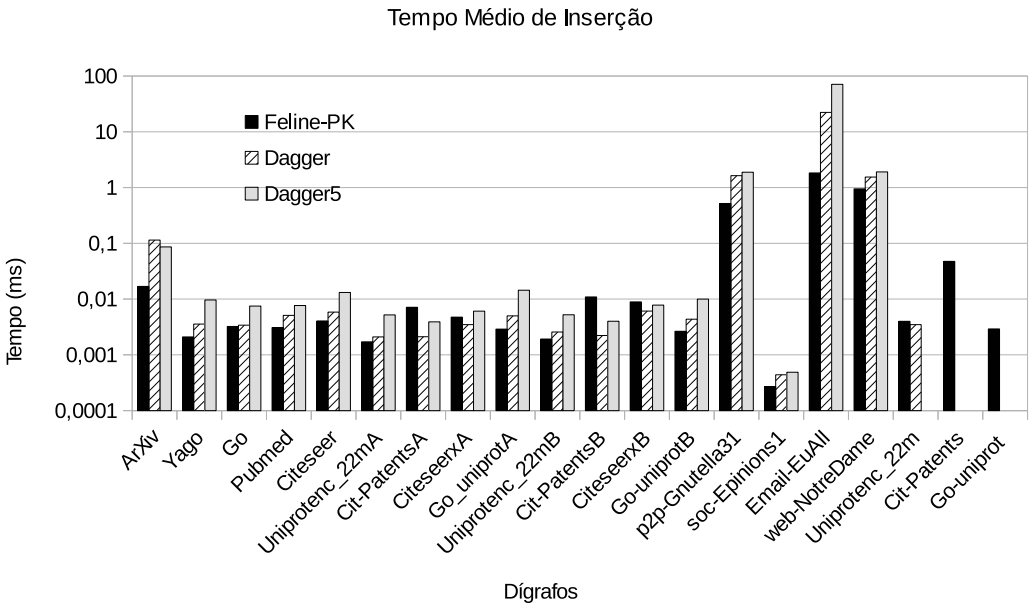


Figura 4.11. Comparação entre Feline-PK, Dagger e Dagger5 (com 5 intervalos). Os tempos computados estão em milissegundos.

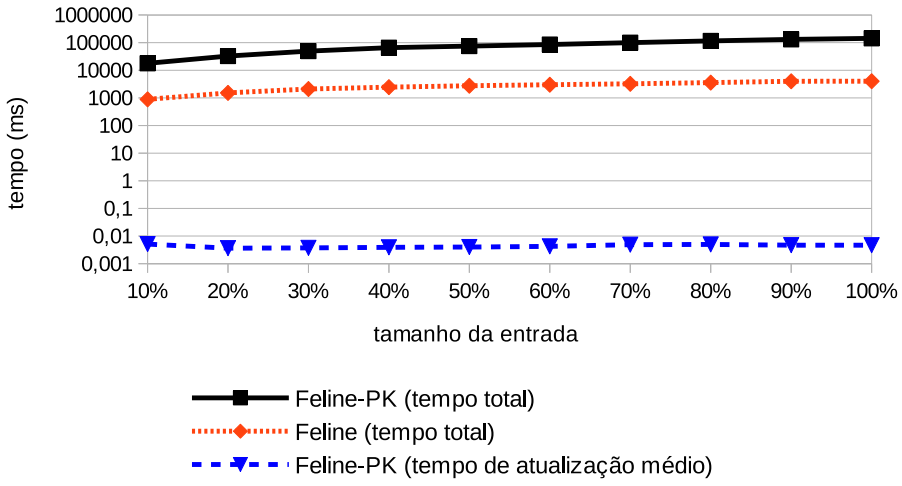


Figura 4.12. Comparação entre Feline-PK e Feline enquanto o grafo Go-Uniprot cresce (10%, 20%,..., 100%). As curvas representam os tempos totais de atualização utilizando Feline-PK e Feline, como também os tempos médios de atualização utilizando Feline-PK. Os tempos mostrados estão em milissegundos.

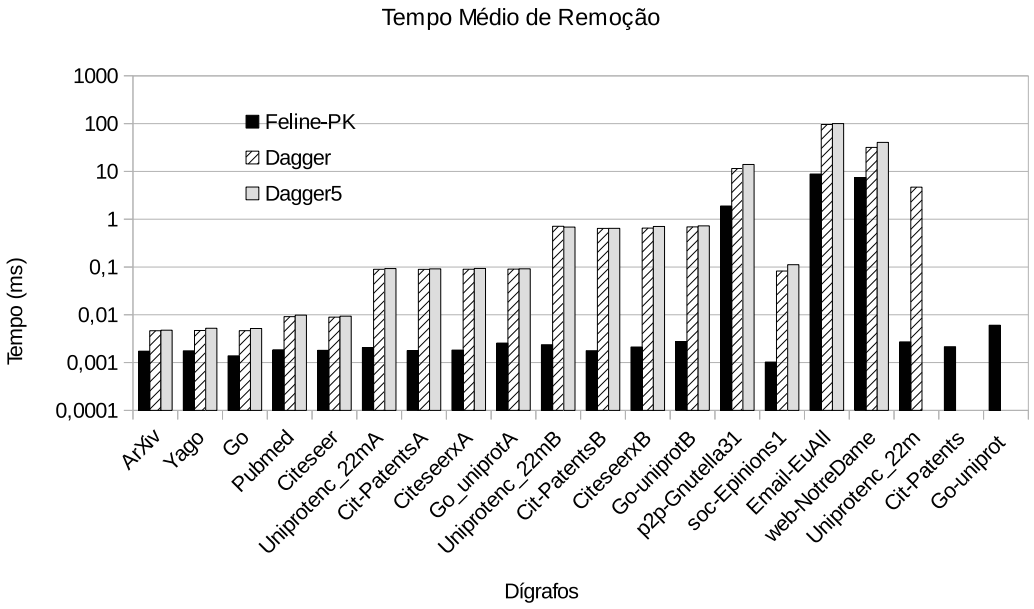


Figura 4.13. Comparação entre Feline-PK, Dagger e Dagger5 (com 5 intervalos). Os tempos mostrados estão em milissegundos.

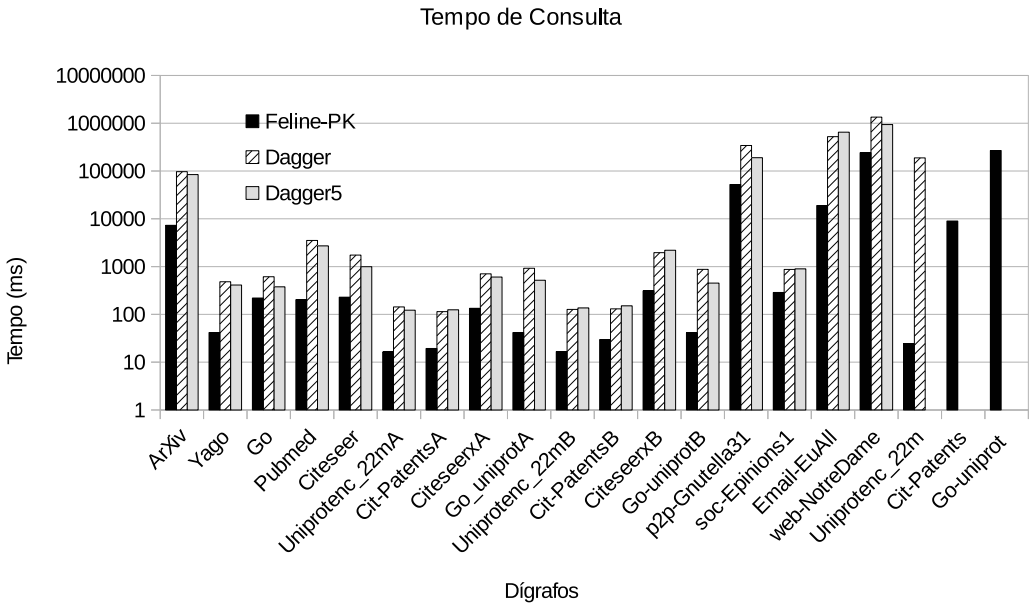


Figura 4.14. Tempo total (em ms) para 100k consultas.

Note que inserir uma aresta está entre um mil e um milhão de vezes mais rápido do que reconstruir o índice inteiro, sendo que esses ganhos aumentam com o tamanho do grafo. Portanto, dado um grafo existente, parece uma boa ideia primeiro construir o índice com Feline e então atualizá-lo com Feline-PK (Figura 4.10 mostra somente grafos acíclicos uma vez que o tempo para agrupar os componentes não afete os resultados). Essa vantagem de Feline-PK pode ser vista na Figura 4.12, que mostra como o desempenho degrada, permitindo comparar o custo de, incrementalmente, inserir uma aresta com o custo de reconstruir o índice do princípio. A figura apresenta três curvas para o grafo Go-Uniprot: uma para o tempo médio para inserir uma das arestas entre os primeiros 10% do grafo, então uma para inserir uma aresta depois dos primeiros 10% mas antes dos primeiros 20% etc; outra curva para o tempo total de inserção, uma a uma, de todas as arestas entre os primeiros 10%, então todas as arestas entre os primeiros 20% etc; e uma para o tempo de reconstrução com Feline do índice inteiro para o DAG com somente as arestas nos primeiros 10%, então com somente as arestas nos primeiros 20% etc. Esses resultados mostram que o tempo médio de atualização tem pouca variação ao longo da construção do grafo enquanto o tempo total aumenta, confirmando que Feline-PK é eficiente para atualizações dinâmicas em grafos muito grandes.

No experimento seguinte, removeu-se de forma aleatória as arestas (uma por uma) dos grafos e mediu-se os tempos médios para removê-las por cada abordagem (Figura 4.13). O desempenho de Dagger é explicado por sua estratégia de propagação de rótulos, que é, normalmente, quando uma aresta é removida, os rótulos dos vértices ancestrais devem ser atualizados. Em Feline-PK, nada precisa ser feito pois uma remoção de aresta não invalida as ordens topológicas, ou, no caso de manutenção dos componentes, apenas uma atualização local. Como resultado, Feline-PK é quase 1000 vezes mais rápido que Dagger quando atualiza grafos muito grandes depois de uma remoção de aresta. Mais ainda, os grafos maiores (Go-Uniprot e Cit-Patents) podem finalmente ser atualizados em tempo razoável. As figuras também mostram que, embora o *overhead* da manutenção dos componentes afete seu desempenho, Feline-PK ainda possui melhor desempenho que Dagger em grafos cíclicos (p2p-Gnutella31, soc-Epinions1, Email-EuAll and web-NotreDame).

Considerando o desempenho das consultas, mediu-se também os tempos para respondê-las. Assim, todos os métodos foram submetidos aos mesmos conjuntos de consultas contendo 10^5 pares de vértices escolhidos de forma aleatória entre o conjunto de vértices de cada grafo. A Figura 4.14 apresenta os resultados, onde podemos ver que Feline-PK tem melhor desempenho do que o Dagger. Esses resultados são justificados pela observação da Figura 4.15, que mostra o comportamento do índice enquanto novas

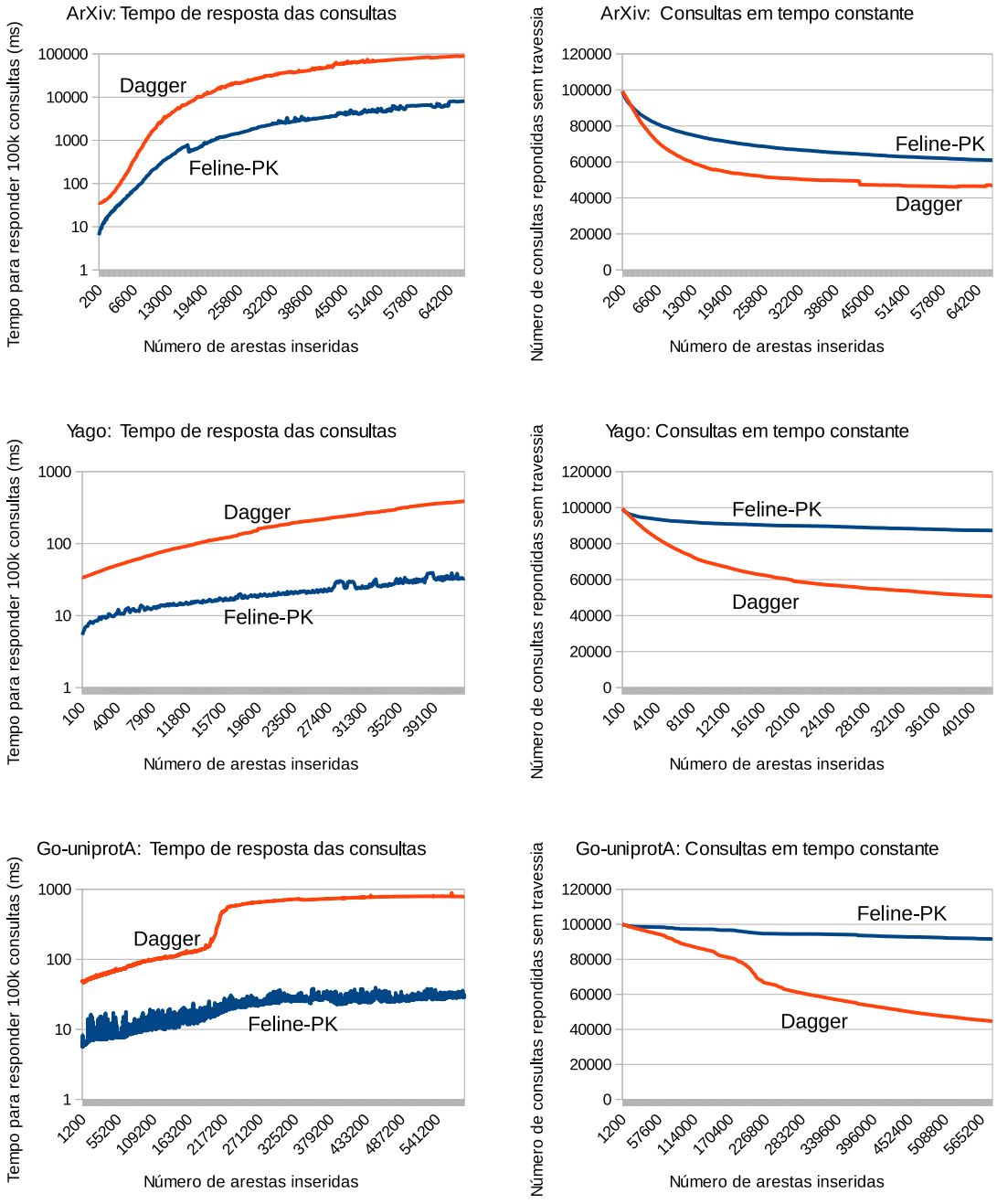


Figura 4.15. Os gráficos mostram uma comparação entre Feline-PK e Dagger. Comparamos o *número de respostas negativas obtidas sem qualquer travessia do grafo* (figuras da esquerda) e *os tempos para responder as consultas* (figuras da direita). Cada ponto representa uma média de 100 execuções. Outros resultados estão disponíveis no Apêndice A.

arestas são inseridas no grafo.

A Figura 4.15 destaca os tempos de Feline-PK e Dagger para responder as consultas, comparados ao número de respostas negativas obtidas sem qualquer travessia no grafo. As figuras mostram que, enquanto novas arestas são inseridas, o índice do Dagger apresenta uma maior redução na quantidade dessas respostas negativas do que Feline-PK, o que faz com o Dagger seja em torno de 10 vezes mais lento do que Feline-PK.

4.4 Indexação Dinâmica com Inserção de Arestas em Lote

Foi apresentado anteriormente um método para a manutenção do índice de alcançabilidade dada a inserção e remoção de arestas no grafo, onde o índice é atualizado a cada inserção ou remoção de uma aresta (*unit-change problem*). Em um contexto de redes sociais, por exemplo, onde centenas de alterações são realizadas a todo momento em grafos muito grandes, essa abordagem não é a ideal. Outro problema é quanto à sequência de atualizações no grafo que podem causar um efeito "sanfona". Por exemplo, suponha que uma atualização separe um componente em dois, em seguida outra atualização junte os dois componentes novamente, depois uma terceira atualização torne a separá-lo. Nesse contexto, seria melhor tratar o índice de forma estática (utilizando o Feline, por exemplo) e refazê-lo por completo a cada período de tempo, após todas as atualizações nesse período serem feitas.

Uma alternativa para resolver esse problema é atualizar o índice após a inserção de um grupo de arestas ou lote (*batch*). Com isso, o efeito "sanfona" poderia ser evitado, bastando apenas analisar a sequência de operações em cada lote e depois remover aquelas que são inúteis para o índice final. Contudo, a despeito de pesquisas realizadas, não foram encontrados métodos para manutenção de índices de alcançabilidade na presença de atualizações em lote. Por conseguinte, apresenta-se aqui um estudo inicial sobre a inserção de arestas em lotes (*batch-change problem*), considerando um algoritmo de ordenação topológica dinâmica denominado PK-2 [Pearce & Kelly, 2010]

Os experimentos realizados aqui consideram somente atualizações em DAGs, ficando como trabalho futuro (I) a comparação com outros algoritmos de ordenação topológica dinâmica para a inserção de lotes de arestas, e (II) adaptação dos algoritmos para a manutenção de componentes fortemente conectados.

4.4.1 Inserção de Arestas em Lotes

Dentre as várias abordagens para o problema, uma destaca-se pela simplicidade e baixa complexidade computacional na prática. Chamada de PK-2 [Pearce & Kelly, 2010], trata-se de uma extensão do algoritmo apresentado na seção anterior e possui complexidade de tempo igual a $O(\min\{k.(|V_{\text{DAG}}|+|E_{\text{DAG}}|), |V_{\text{DAG}}||E_{\text{DAG}}|\})$ para processar qualquer sequência de k inserções de lotes de arestas.

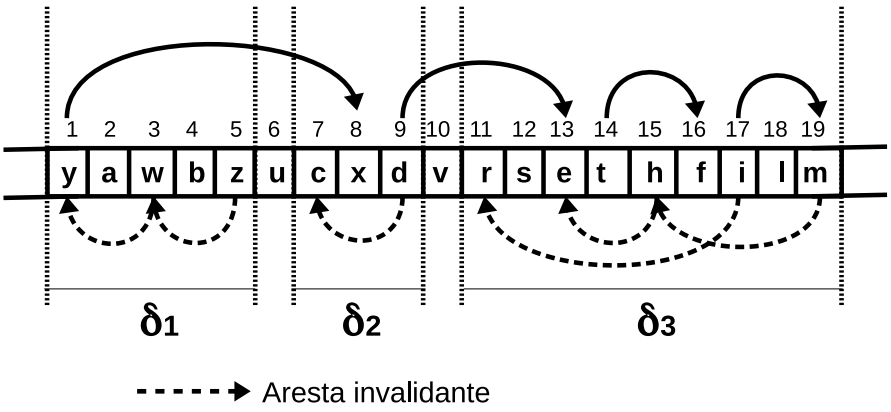


Figura 4.16. Exemplo regiões afetadas e suas arestas invalidantes em um ordenação topológica qualquer.

O algoritmo PK-2 recebe um conjunto B de arestas invalidantes (ou seja, arestas que invalidam a ordenação topológica dos vértices) e rearranja-as tornando válida a ordenação topológica. Cada grupo de arestas invalidantes pode separar a ordenação topológica em diferentes subconjuntos δ disjuntos ou não. A Figura 4.16 mostra quatro arestas invalidantes, que dividem a ordenação topológica em três subconjuntos δ_1, δ_2 e δ_3. Cada subconjunto δ_i (aqui chamado de subconjunto inválido) é tratado de forma independente dos outros (podem ser rearranjados em paralelo). Os subconjuntos inválidos são processados em uma única passada, iniciando-se do vértice com maior ordem e movimentando cada vértice apenas uma vez. Cada aresta invalidante que se sobrepõe pode ser agrupada em um único subconjunto inválido como definido a seguir:

Definição 19 (Subconjunto inválido δ_i)**.** *Seja $G_{DAG} = (V_{DAG}, E_{DAG})$ e $\prec$ uma ordenação topológica válida. Para um conjunto B de arestas invalidantes (recém inseridas em G_{DAG}), um subconjunto inválido é denotado por δ_i e definida como $\{k \in V_{DAG} \mid u \prec_= k \prec_= v$ e $(v,k) \in B$ ou $(k,u) \in B\}$, onde $\prec^u$ é mínimo e $\prec^v$ é máximo*[1].

[1]Usamos a notação $\prec^u$ denotando o ranking de u na ordenação topológica $\prec$. A notação não é um abuso uma vez que $\prec$ é implementado por um vetor.

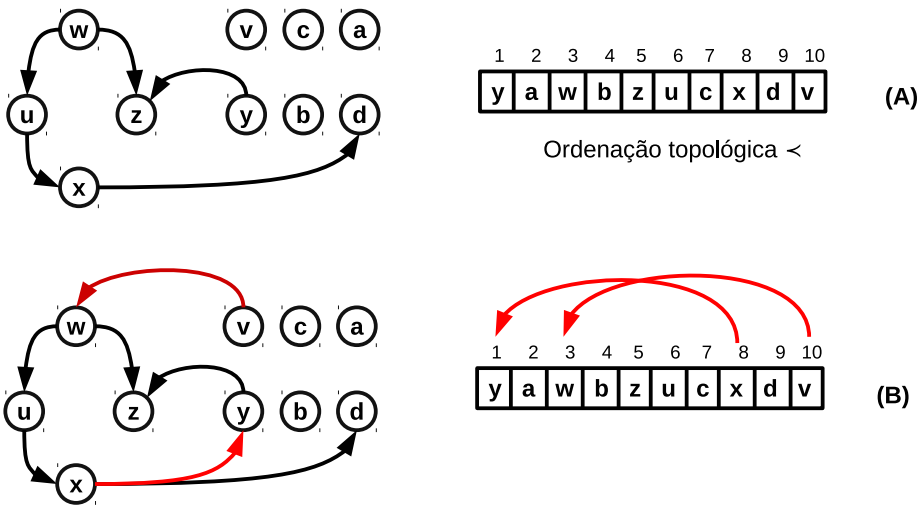


Figura 4.17. Grafo e sua região afetada. Em (A), o grafo antes da inserção de arestas invalidantes e , em (B), após a inserção das arestas (v, w) e (x, y).

O exemplo da Figura 4.17-(B) mostra a região afetada e duas arestas invalidantes ((v, w) e (x, y)) inseridas no grafo em (A). Para exemplificar o funcionamento básico do Algoritmo 12, suponha que o conjunto B contendo o lote de arestas contém somente essas duas arestas invalidantes (setas na cor vermelha na figura), dada uma ordenação topológica válida, a função *addBatch* insere essas arestas no grafo e invoca o algoritmo *addBatch*. Na linha 2 do algoritmo, todas as arestas válidas são removidas do lote, pois elas não interferem na ordenação (no caso do exemplo utilizado elas não existem).

Cada grupo de arestas sobrepostas inicia-se no vértice de maior índice do grupo (cauda da primeira aresta) e termina no vértice que é a cabeça da última aresta. Na Figura 4.17-(B), o vértice que inicia a região é o v de índice 10 e o vértice que termina é o y, com índice 1. O algoritmo começa a avaliar o vértice de maior índice primeiro, pois este encabeçará o novo grupo a ser organizado. Sendo assim, o grupo é processado da direita para a esquerda. A linha 4 do algoritmo garante que as arestas sejam processadas nessa ordem. O algoritmo, após isso, pode ser dividido em duas funções: *shift* (Algoritmo 13) e *discover* (Algoritmo 14).

Na função *discover* todo o conjunto B é percorrido, onde o objetivo é identificar os vértices que serão movidos (i.e., receberão novos índices) visitando-os apenas uma vez. Essa função inicia da aresta invalidante (u, v) com maior índice $\prec_x^u$ e, seguindo uma busca em profundidade, todo par (k, u), significando que k é alcançável a partir de u dentro do grupo, é colocado em uma pilha. Esse passo é repetido para todas as arestas invalidantes e o fato de escolher a aresta invalidante com maior valor $\prec_x^u$ assegura que a ordem final esteja correta para cada vértice visitado.

Algoritmo 12: addBatch

```
1  addBatch(B,V) início
2      Remova de B todas as arestas válidas;
3      se |B| > 0 então
           // ordena as arestas invalidantes em ordem decrescente
              pelo índice da cauda.
4          ordena(B);
5          lb ← |V|;
6          s ← 0;
7          para todo i ← 0...|B| − 1 faça
8              (u, v) ← B[i];
9              se ≺_x^u < lb ∧ i ≠ 0 então
10                 Q ← discover({B[s]...B[i − 1]});
11                 shift(lb, Q);
12                 s ← i;
13             lb ← min(≺_x^v, lb);
           // processa a última região afetada
14         Q ← discover({B[s]...B[|B| − 1]});
15         shift(lb, Q);
```

Para exemplificar, na Figura 4.18-(A), essa função insere em uma pilha todos os vértices alcançáveis a partir das cabeças da arestas invalidantes, com uma indicação da cauda da aresta invalidante que o alcança. A pilha Q, então, é um conjunto cujos membros estão corretamente ordenados tal que $\forall(x_1, d_1)_i, (x_2, d_2)_j \in Q, (d_2 \prec_x d_1 \vee (d_1 = d_2 \wedge x_1 E^*_{\mathrm{DAG}} x_2)) \Rightarrow i < j)$, onde i e j são posições na pilha.

No exemplo da figura, todos os vértices dentro da região afetada são visitados e aqueles que não são alcançáveis pelas arestas invalidantes são movidos para as primeiras posições da nova ordenação (veja a Figura 4.18-(B)). Neste exemplo, os vértices a, b e c, vão para as posições 1, 2 e 3, respectivamente. Assim que um vértice, que é a cauda da aresta invalidante mais interna, é visitado, todos os vértices que formam par com ele na pilha são desempilhados e adicionados à nova ordenação em sequência (fase de *shift*). Isso pode ser verificado no exemplo da Figura 4.18-(A) onde, no topo da pilha Q, há o par (y, x).

Continuando com o exemplo, o algoritmo *shift* começa pelo par no topo da pilha. Dessa forma, o vértice x é colocado na posição livre seguinte (posição 4). Em seguida todos os vértices alcançáveis a partir de x são adicionados às próximas posições. Neste caso, o y e o d. O mesmo é feito para o par (w, v) que é agora o topo da pilha. Nas funções *discover* e *shift*, $topoOrd^{-1}$ retorna o vértice, dada a sua posição na ordenação

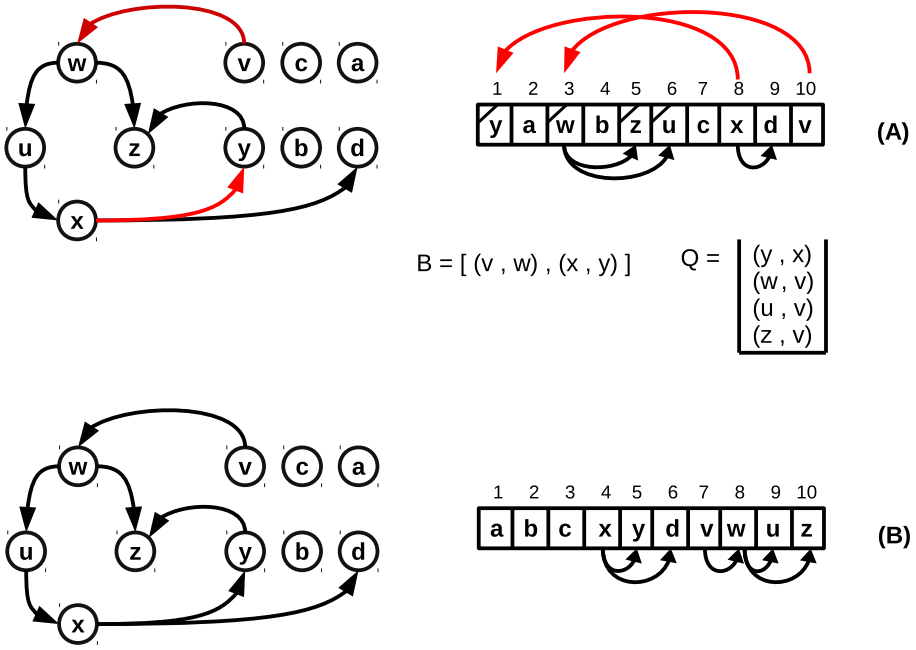


Figura 4.18. Aplicação das funções *discover* e *shift*.

topológica.

Quanto à ordenação topológica referente à segunda coordenada Y do índice de alcançabilidade (i. e., $\prec_y$), basta repetir todo o processo considerando as arestas invalidantes dessa ordenação topológica.

Algoritmo 13: Shift

```
1  shift(i, Q) início
2      n ← 0;
3      enquanto Q ≠ ∅ faça
           // topoOrd^{-1}(i) retorna o vértice localizado no índice i
4          w ← topoOrd^{-1}(i);
5          se livre(w) então
6              n ← n + 1; livre(w) ← false;
7          senão
8              ≺_x^w ← (i − n); topoOrd^{-1}(i − n) ← w;
9          (v, d) ← top(Q);
10         enquanto Q ≠ ∅ ∧ w = d faça
11             n ← n − 1;
12             ≺_x^v ← (i − n); topoOrd^{-1}(i − n) ← v; pop(Q);
13             (v, d) ← top(Q);
14         i ← i + 1;
```

Algoritmo 14: Discover

```
1  discover(B) início
2      Q ← ∅;
       // ordena as arestas em ordem decrescente pelo índice da
          cauda.
3      sort(B);
4      para todo i ← 0...|B| faça
5          (u, v) ← B[i];
6          se ¬livre(v) então
7              dfs(v, ≺_x^u));
8      retorna Q;
9  dfs(v, ub) início
10     livre(v) ← true;
11     onDFS(v) ← true;
12     para todo (v, s) ∈ E faça
13         se onDFS(s) então
14             retorna aborta;
15         se ¬livre(s)∧ ≺_x^s< ub então
16             dfs(s, ub);
17     onDFS(v) ← false;
18     push((v, topoOrd^{-1}(ub)), Q);
```

4.4.2 Experimentos

Conforme mencionado previamente, não foram encontradas referências de índices de alcançabilidade baseados em atualizações em lote nos trabalhos relacionados. Portanto, os experimentos aqui apresentados visam demonstrar o desempenho do método proposto, em contraste com a realização de atualizações unitárias. Pesquisas futuras podem considerar também outras abordagens de ordenação topológica, bem como a manutenção de componentes fortemente conectados.

O primeiro experimento foi realizado para verificar se há vantagens em inserir arestas em lote ao invés de uma por vez. Para isso, medimos o tempo de inserção (que inclui o tempo de atualizar o índice) para construir cada grafo.

A Figura 4.19 resume os tempos de inserção de arestas nos grafos indicados. Para cada grafo, foram inseridas as arestas uma a uma, enquanto que essas mesmas arestas foram agrupadas em um lote único e inserido nos grafos, i.e., um grupo de tamanho $|E_{\mathrm{DAG}}|$. Apesar de os tempos da inserção em lote serem até 10% menores, não é de se esperar que essa seja uma situação prática, pois é um caso típico para uma abordagem estática. Contudo, esse experimento mostra, em termos gerais, que

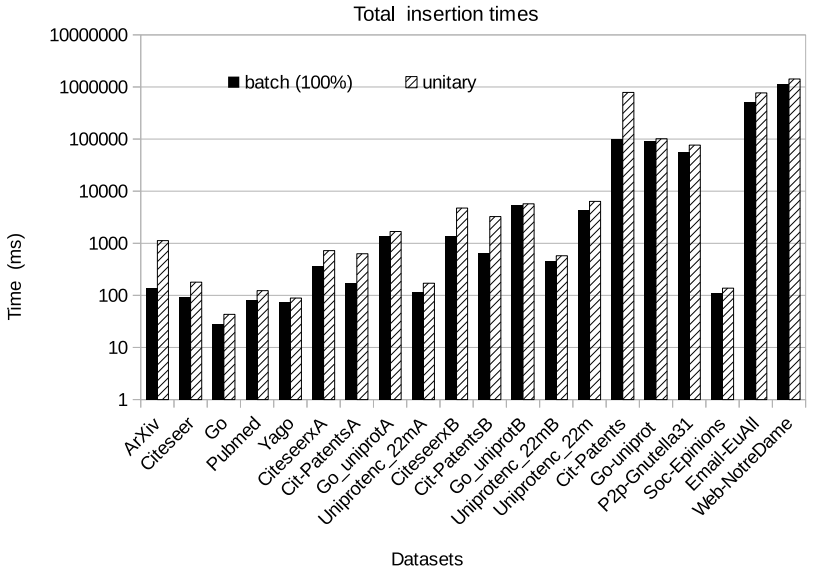


Figura 4.19. Tempos de inserção de arestas. O gráfico apresenta a relação entre o tempo para inserir arestas de forma unitária e em lote. O tempo para a inserção em lote é para todas as arestas do grafo, agrupadas em um único lote.

mesmo numa situação extrema a abordagem de inserção em lote oferece ganhos reais em comparação com as inserções unitárias.

É razoável supor que em situações reais os lotes não sejam tão grandes. Por isso, o segundo experimento verifica como o método se comporta com tamanhos de lotes variados.

Os gráficos da Figura 4.20 mostram os tempos de inserção totais para alguns grafos variando-se os tamanhos dos lotes (para os outros grafos, consulte o Apêndice A). Cada conjunto de arestas inserido foi dividido em partes que correspondem a 1%, 2%, 3%, ..., 100% do tamanho total de cada um. Cada lote foi inserido no grafo até completá-lo. Por exemplo, o grafo ArXiv possui 66707 arestas, no experimento correspondente à inserção de lotes com tamanho de 1%, cada lote tem aproximadamente 667 arestas. Logo, esses lotes foram inseridos até completar o grafo. O mesmo foi feito para os outros tamanhos.

Os tempos de inserção são compostos pelo tempo para inserir as arestas no grafo (onde lê-se *normal* na legenda dos gráficos) e o tempo para organizar as arestas invalidantes (*inval* na legenda). Este, por sua vez, é dominado por duas partes críticas no algoritmo, a fase em que as arestas invalidantes são ordenadas (Algoritmo 12) e a fase denominada *shift* (nos gráficos, referente aos algoritmos *shift* e *discover*), que está

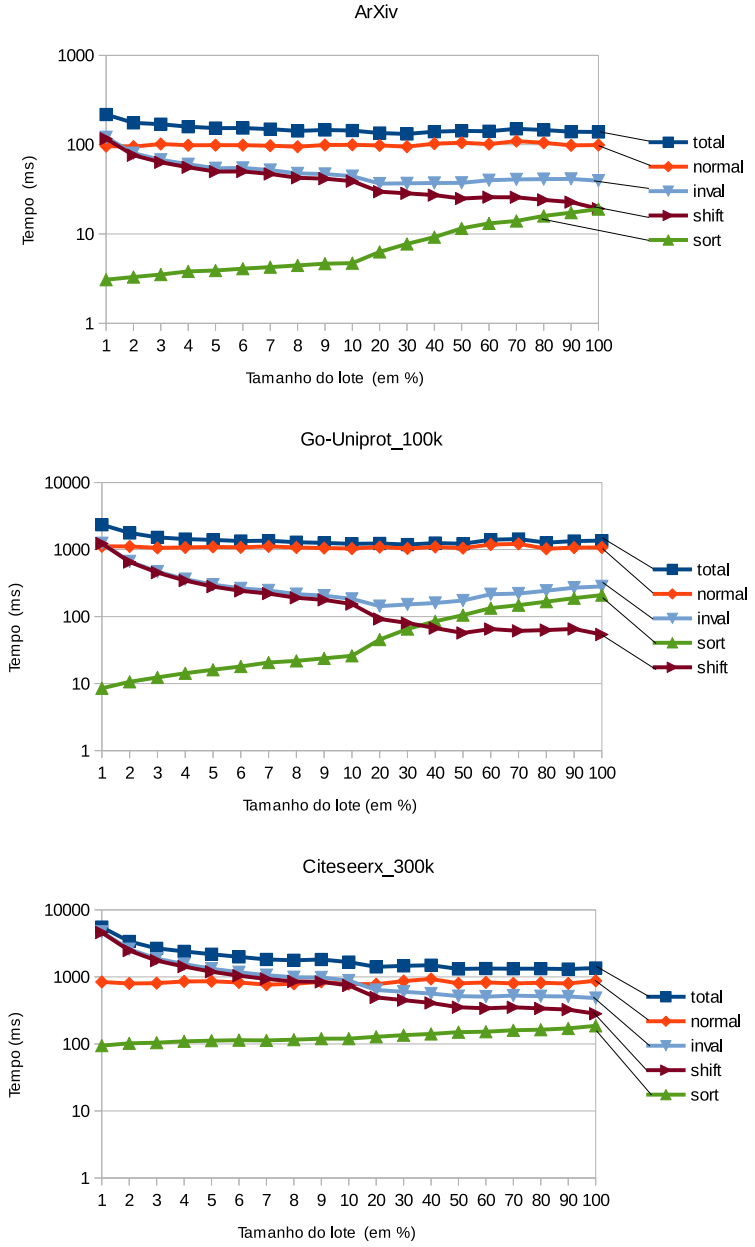


Figura 4.20. Tempos de inserção total dos lotes e a contribuição de cada fase do algoritmo de inserção nesse tempo. Na legenda: "normal" refere-se ao tempo para inserção de arestas no grafo. "inval" é tempo necessário para organizar as arestas invalidantes e atualizar o índice; esse último é composto por "sort" e "shift", que são os tempos para ordenar as arestas invalidantes, juntamente com as funções *discover* e *shift*. Outros resultados podem ser encontrados no apêndice.

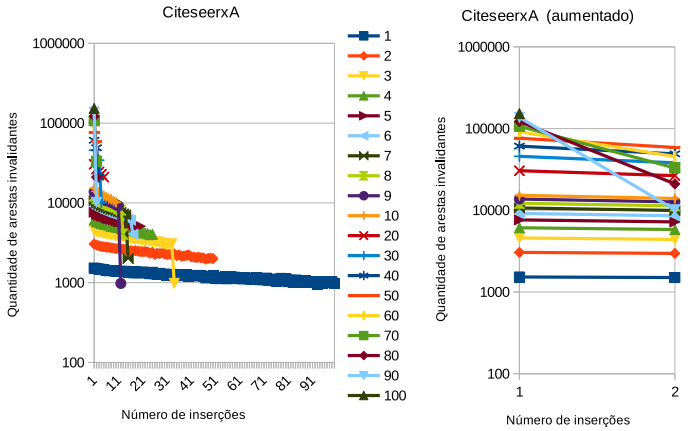


Figura 4.21. Quantidade de arestas invalidantes à medida que os lotes são inseridos no grafo Citeseerx. Na legenda, os tamanhos estão em %. O gráfico menor à direita dá destaque para os lotes de tamanhos 1% e 2%. Os gráficos para os outros grafos estão no Apêndice.

na legenda dos gráficos. A figura permite entender que, embora os tempos de inserção permaneçam com pouca variação ao longo das inserções, cada fase do algoritmo influencia no desempenho de acordo com o tamanho dos lotes.

Percebe-se que o tempo para atualizar o índice sofre bastante influência da fase *shift* quando o tamanho do lote é pequeno. Cada lote é independente dos demais, contudo quando as arestas invalidantes de um lote são organizadas, as arestas do próximo lote podem diminuir, pois os vértices são movidos para outras posições. Essa característica pode ser percebida no gráfico da Figura 4.21, para o grafo Citeseerx, que mostra que a quantidade de arestas invalidantes (conjunto B) diminui com o tempo, para cada porcentagem. Isso explica a queda do tempo de execução na fase *shift* do grafo Citeseerx na Figura 4.20. Contudo, para esse mesmo grafo, à medida que o tamanho do lote aumenta (de 1% para 2%, depois para 3% e assim por diante), a ordenação das arestas invalidantes torna-se mais custosa, contribuindo para manter o tempo de inserção total com pouca variação.

4.5 Sumário de Resultados do Capítulo

Este capítulo apresentou um índice eficiente para alcançabilidade em grafos dinâmicos muito grandes. Foi proposta aqui uma nova solução para o processamento de atualizações em índices de alcançabilidade baseado no problema de Ordenação Topológica Dinâmica (ou DTO - *Dynamic Topological Ordering*).

Foi demonstrado experimentalmente que Feline-PK atualiza de forma eficiente o índice. As Figs. 4.10-4.12 mostram, para 16 dígrafos diferentes (com milhões de vértices e arestas), que inserir uma aresta leva entre poucos μs e algumas dezenas de μs. A remoção de arestas exige tempos similares se o dígrafo é um DAG, enquanto que no caso de dígrafos cíclicos o tempo para atualizar o índice é algo em torno de 1 ms (devido a separações nos componentes). Esses tempos estão entre 100 vezes (para dígrafos pequenos) e um milhão de vezes (para dígrafos grandes) menores que os tempos que Feline leva para reconstruir o índice. As Figs. 4.11 e 4.13 também mostram que, não importando o dígrafo, os tempos de atualização são menores que os obtidos pela solução do estado-da-arte, Dagger. O tempo médio de remoção pode chegar a ser cerca de 100 vezes menor para alguns dígrafos. Mais importante, o índice de Feline-PK resulta em tempos de consulta significantemente melhores do que o índice do Dagger. De fato, isso pode ser observado na Fig. 4.14, onde as respostas às consultas de alcançabilidade do Feline-PK chegam a ser 10000 vezes mais rápidas (mas normalmente são cerca de 10 vezes mais rápidas) do que Dagger. De acordo com a Fig. 4.15, esses tempos crescem com o tamanho do dígrafo, mas esse parâmetro afeta menos Feline-PK do que Dagger. Em outras palavras, Feline-PK escala melhor.

Este capítulo considerou também a inserção de lotes de arestas e a atualização do índice com base nesses lotes. Para isso, uma extensão do PK foi utilizada e verificou-se que pode haver ganhos de desempenho. Contudo, uma pesquisa mais aprofundada no tema fica como trabalho futuro.

Capítulo 5

Conclusões e Trabalhos Futuros

Este capítulo apresenta um sumário das contribuições de pesquisa desta tese. Em seguida, algumas limitações e propostas de trabalhos futuros são discutidos.

5.1 Sumário de Resultados e Contribuições

O problema atacado por esta tese é conhecido como *alcançabilidade*, o qual possui como entradas um grafo direcionado e uma consulta envolvendo dois vértices u e v (representada por $u \rightsquigarrow^{?} v$). A saída deve ser uma reposta positiva, significando que o vértice u alcança o vértice v, ou negativa, significando que u não alcança v. Examinou-se este problema em duas versões, para grafos estáticos e para grafos dinâmicos. A versão para grafos estáticos considera que os grafos não sofrem alterações ao longo do tempo, não exigindo que os índices de alcançabilidade sejam atualizados. A versão para grafos dinâmicos é o oposto disso. O grafos ditos dinâmicos podem sofrer atualizações de tempos em tempos, no conjunto de vértices ou arestas, fazendo com que o índice acompanhe essas atualizações para refletir corretamente a alcançabilidade entre os vértices. Esta última versão do problema é mais complexa e, na presente data, há poucos trabalhos publicados, ao contrário da primeira versão. Esta tese apresenta, então, algoritmos eficientes para cada tipo de problema. Esses algoritmos são chamados de *Feline* e *Feline-PK*. Avaliou-se experimentalmente cada algoritmo, demonstrando que, tanto Feline quanto Feline-PK, são escaláveis e produzem índices eficientes considerando o tempo de construção do índice, o tamanho desse índice e o tempo para responder às consultas.

O algoritmo Feline (*Fast rEfined onLINE search*) é inspirado na área de visualização de grafos (*graph drawing*) e no algoritmo de Kameda [Kameda, 1975] para grafos planares. Feline constrói um índice a partir da representação do grafo em um plano

bidimensional, da qual são extraídas as informações de alcançabilidade em tempo constante para uma porção significativa de consultas. Experimentos demonstram a eficiência do método em relação às abordagens do estado da arte, sendo que a característica mais importante de Feline é a sua capacidade de descartar algumas busca, evitando a expansão daqueles vértices que aparecem depois do vértice objetivo nas ordenações topológicas. O índice gerado por Feline, que é linear no número de vértices dos grafos, evita a etapa de otimzação dos parâmetros (que, em geral, referem-se à quantidade de rótulos de cada vértice) presente em muitas técnicas que constam na literatura relacionada. Esse índice, mais simples e pequeno, permite também que Feline utilize o grafo invertido para extrair novos rótulos, agilizando ainda mais as respostas às consultas.

Como extensão do Feline, Feline-PK é voltado para grafos dinâmicos. Essa extensão utiliza algoritmos de *Ordenação Topológica Dinâmica* para realizar as atualizações a cada modificação no respectivo grafo. Feline-PK utiliza ainda estruturas de dados e representações que facilitam a indexação de grafos direcionados cíclicos, uma vez as atualizações podem afetar componentes fortemente conectados. Experimentos demonstram que utilizar Feline-PK é melhor do que atualizar o índice inteiro a cada modificação no grafo, sendo que quando arestas ou vértices são removidos do grafo, nada precisa ser feito. Um aprimoramento preliminar dessa extensão, uma abordagem incremental em lotes, também foi apresentada. Com ela, é possível melhorar o tempo de atualização do índice dada a inserção de um conjunto grande de arestas. Com isso, conclui-se que a abordagem Feline e suas extensões representam uma ótima alternativa escalável para problema de indexação de alcançabilidade, com algumas limitações, que são apresentadas a seguir.

5.1.1 Limitações

Os algoritmos propostos possuem algumas limitações quando comparados a outras técnicas, que incluem:

1. Quantidade de ordenações topológicas: os algoritmos propostos utilizam ordenações topológicas que devem ser complementares para que o índice seja eficiente. Contudo, não se sabe se a utilização de mais ordenações aumenta o desempenho do índice e nem como determinar a complementaridade dessas ordenações. Esse problema também é apontado pela abordagem de Kameda e segue em aberto.

2. Falsos positivos: a indexação de grafos não-planares leva à ocorrência de falsos-positivos nas consultas, gerando, em muitos casos, caminhamentos no grafo e o aumento do tempo para responder às consultas. Os métodos propostos não

indicam como detectar e nem como contornar o aparecimento de falsos-positivos, principalmente no caso de grafos dinâmicos.

3. Memória principal: em todos os algoritmos, os grafos e índices têm que estar na memória principal. Contudo, isso não é razoável na prática, uma vez que, se o grafo e o seu índice forem maiores do que a quantidade de memória disponível, os resultados apresentados podem sofrer alterações. Dessa forma, é importante considerar estudos sobre a localidade de referência. Nenhum dos trabalhos encontrados na literatura realizam tal estudo.

5.2 Trabalhos Futuros

5.2.1 Utilização de espaço multidimensional

Sendo um falso-positivo um posicionamento ambíguo de um vértice no plano, isto é, a atribuição de uma coordenada a um vértice v que se encontra dentro da área de dominação de outro vértice u e sem a existência de um caminho entre u e v no respectivo grafo. Tal posicionamento se dá pela representação (ainda inadequada) de um grafo não planar em um espaço planar (bidimensional).

A partir dos experimentos documentados no Capítulo 3, percebemos que o uso de duas ordenações topológicas, juntamente com o uso de um grafo invertido, ofereceu uma redução da quantidade de falsos-positivos no índice obtido, o que resultou em uma diminuição do tempo médio de consultas. Esses experimentos apresentam, portanto, evidências que essa ambiguidade pode ser reduzida com o uso de mais de uma ordenação sobre o mesmo grafo. Dessa forma, ao invés de usarmos um grafo invertido, poderíamos pensar em usar mais dimensões (espaço multidimensional) para diminuir a quantidade de falsos-positivos.

Um exemplo pode ser visto na Figura 5.1, que mostra um DAG e sua representação em um plano bidimensional. Na figura, o vértice 3 não é alcançável a partir do vértice 2. No entanto, no plano, o vértice 3 está na região de dominação do vértice 2. A exceção entre esses vértices não pode ser resolvida em um plano bidimensional. Logo, a solução mais óbvia seria resolver a exceção em outra dimensão, criando um plano tridimensional. A Figura 5.2 apresenta uma solução, pois os pontos em exceção são deslocados no plano 3D, fazendo com que o vértice 3 saia da região de dominação do vértice 2.

O resultado desse estudo é apresentar recomendações a respeito da quantidade mínima de dimensões que indexem a alcançabilidade em cada tipo de grafo. O problema

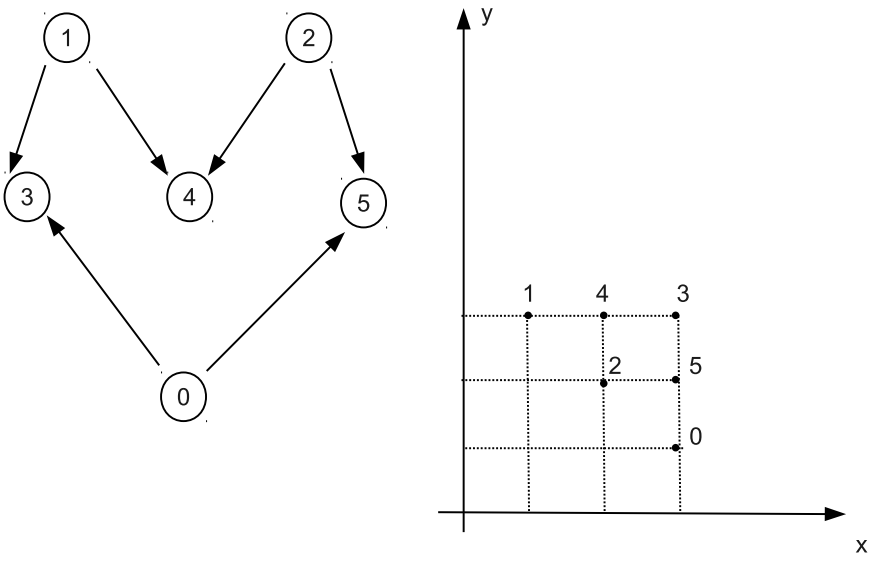


Figura 5.1. Exemplo de vértices em exceção num plano bidimensional.

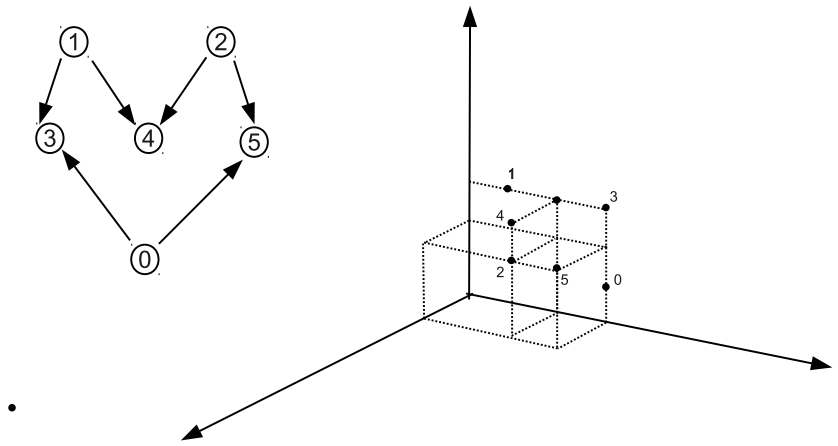


Figura 5.2. Resolução de exceção utilizando um plano tridimensional.

em questão ainda é um problema em aberto na Computação, conforme exposto nos trabalhos de [Kameda, 1975] e [Yildirim et al., 2010].

5.2.2 Indexação sob demanda

A pesquisa em alcançabilidade em grafos, até o momento presente, tem foco somente na melhoria de desempenho da indexação e desempenho das consultas. Contudo, pelo que se sabe, nada foi feito em direção a uma caracterização da frequência com que cada vértice do grafo é consultado, dado o contexto de aplicação do grafo. Essa caracterização das consultas permitiria a otimização do índice, pela utilização de cache ou outras técnicas, de forma a responder algumas consultas mais rapidamente (para aqueles vértices mais frequentemente consultados). Outra possibilidade, seria a realização de uma indexação por demanda, dado que alguns vértices podem nunca aparecer em consultas.

Na indexação sob demanda, os vértices são indexados somente no surgimento de uma consulta envolvendo esses vértices. Por conseguinte, o índice cresce de acordo com a necessidade, gerando uma representação casada com a distribuição de frequência das consultas. Essa representação poderia ser entendida como uma nova condensação do grafo, incorporando uma redução transitiva.

5.2.3 Abordagem paralela e distribuída

Uma complementação imediata do trabalho é a definição de uma arquitetura paralela e distribuída para a computação e o armazenamento do índice. É razoável supor que em um DAG muito grande pode haver vários subgrafos do tipo *upward*. Pensando nessa hipótese, uma solução paralela e distribuída poderia ser composta pela distribuição de partições do DAG (subgrafos) em uma arquitetura de cluster, sendo que, dada uma consulta de alcançabilidade, uma pesquisa nos subgrafos possa ser feita em paralelo. De fato, nos subgrafos do tipo *upward* essa consulta seria em tempo constante. Contudo, naqueles subgrafos não planares, a abordagem Feline poderia ser utilizada.

A Figura 5.3 ilustra um exemplo de um DAG e três de seus subgrafos (circulados). Cada subgrafo destacado pode ser indexado considerando a sua planaridade. Dessa forma, na ocorrência de uma consulta de alcançabilidade, o índice individual de cada subgrafo pode ser consultado em paralelo.

Contudo, há alguns problemas dessa abordagem. O primeiro é quanto a identificação dos subgrafos e os critérios utilizados para isso. Devemos pensar na quantidade de subgrafos e se a identificação dos mesmos será parametrizada ou realizada de forma dinâmica, sendo a mesma preocupação para o tamanho dos subgrafos, de forma a ma-

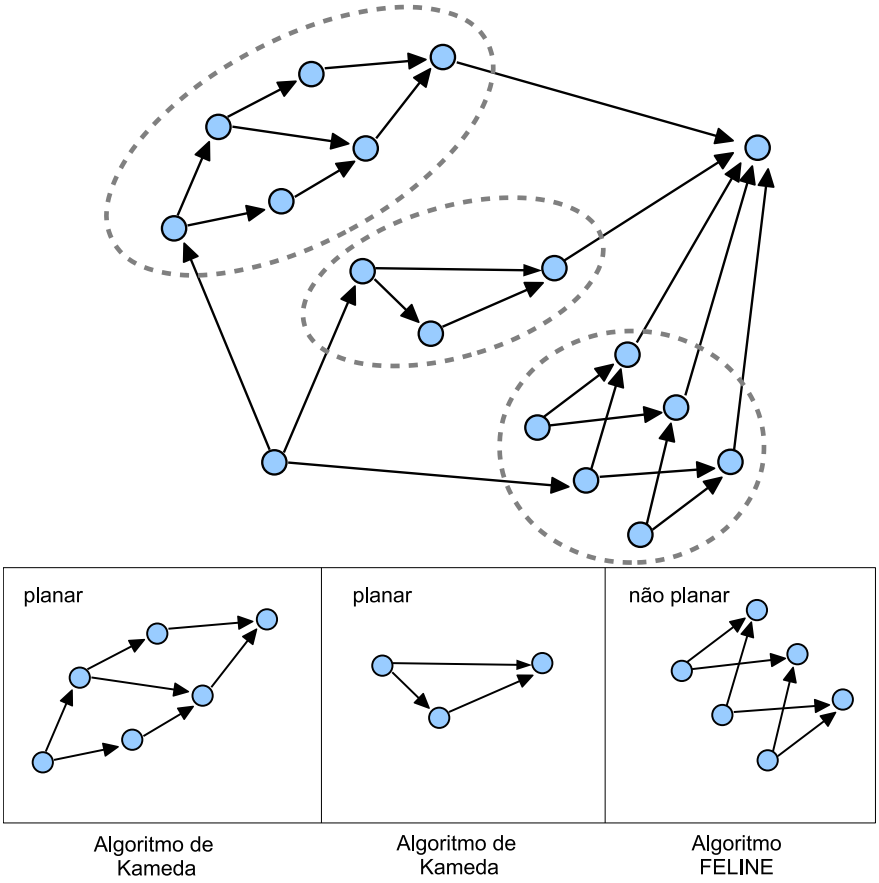


Figura 5.3. DAG de exemplo para indexação paralela e distribuída. Os três subgrafos identificados podem ser indexados utilizando uma abordagem diferente, de acordo com a sua planaridade.

ximizar o desempenho das buscas em paralelo. Outro problema é quanto a planaridade dos grafos, pois sabemos que testar se um dado DAG é do tipo *upward* planar é um problema NP-Completo [Garg & Tamassia, 1995].

5.2.4 Alcançabilidade em grafos com restrições

O trabalho apresentado neste texto dedica-se à explorar indexação de alcançabilidade em grafos sem restrições, i.e., em grafos não rotulados. Contudo, conforme Jin et al. [2010], a maioria dos grafos do mundo real são rotulados (as arestas possuem rótulo que denotam o relacionamento entre os vértices). Um problema de pesquisa ligado a esse tipo de grafo é saber se dois vértices são alcançáveis entre si por um caminho cujos rótulos obedecem a alguma restrição.

Jin et al. [2010] e Yildirim et al. [2013] apresentam um exemplo encontrado nas redes sociais, onde cada pessoa é representada por um vértice e duas pessoas estão

relacionadas por arestas indicando o tipo do relacionamento: amigo-de, irmão-de, filho-de, empregado-de, etc. Dado isso, uma consulta de alcançabilidade poderia levar em consideração essas restrições. Por exemplo, se quisermos saber se um dado usuário u alcança outro usuário v por relacionamentos do tipo "amigo-de", ou mesmo se esses dois usuários possuem algum grau de parentesco.

Uma proposta de trabalho futuro é adaptar o método FELINE para grafos com restrições, uma vez que há poucos trabalhos nessa direção.

Referências Bibliográficas

Agrawal, R.; Borgida, A. & Jagadish, H. V. (1989). Efficient management of transitive relationships in large data and knowledge bases. Em *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pp. 253--262, New York, NY, USA.

Anyanwu, K. & Sheth, A. (2003). p-queries: Enabling querying for semantic associations on the semantic web. Em *In Proceedings of the Twelfth International World-Wide Web Conference*, pp. 690--699. ACM Press.

Arqué, N. M. & Nettleton, D. F. (2012). Analysis of on-line social networks represented as graphs — extraction of an approximation of community structure using sampling. Em *Proceedings of the 9th international conference on Modeling Decisions for Artificial Intelligence*, MDAI'12, pp. 149--160, Berlin, Heidelberg. Springer-Verlag.

Battista, G. D.; Eades, P.; Tamassia, R. & Tollis, I. G. (1998). *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall.

Battista, G. D.; Tamassia, R. & Tollis, I. G. (1992). Area requirement and symmetry display of planar upward drawings. *Discrete Comput. Geom.*, 7(4):381--401. ISSN 0179-5376.

Bender, M. A.; Fineman, J. T. & Gilbert, S. (2009). A new approach to incremental topological ordering. Em *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, pp. 1108--1115, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

Bertolazzi, P.; Battista, G. D. & Didimo, W. (2002). Quasi-upward planarity. *Algorithmica*, 32(3):474–506.

Bertolazzi, P.; Battista, G. D.; Mannino, C. & Tamassia, R. (1998). Optimal upward planarity testing of single-source digraphs. *SIAM Journal on Computing*.

Bramandia, R.; Choi, B. & Ng, W. K. (2010). Incremental maintenance of 2-hop labeling of large graphs. *Knowledge and Data Engineering, IEEE Transactions on*, 22(5):682–698. ISSN 1041-4347.

Cai, J. & Poon, C. K. (2010). Path-hop: Efficiently indexing large graphs for reachability queries. Em *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, CIKM '10, pp. 119--128, New York, NY, USA. ACM.

Chakrabarti, D. & Faloutsos, C. (2012). *Graph Mining: Laws, Tools, and Case Studies*. Synthesis Lectures on Data Mining and Knowledge Discovery. Morgan-Claypool Publishers.

Chen, L.; Gupta, A. & Kurul, M. E. (2005). Stack-based algorithms for pattern matching on dags. Em *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pp. 493--504. VLDB Endowment.

Chen, Y. (2009). General spanning trees and reachability query evaluation. Em *Proceedings of the 2Nd Canadian Conference on Computer Science and Software Engineering*, C3S2E '09, pp. 243--252, New York, NY, USA. ACM.

Chen, Y. & Chen, Y. (2008). An efficient algorithm for answering graph reachability queries. Em *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pp. 893--902, Washington, DC, USA. IEEE Computer Society.

Chen, Y. & Chen, Y. (2011). Decomposing dags into spanning trees: A new way to compress transitive closures. Em *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pp. 1007–1018. ISSN 1063-6382.

Cheng, J.; Huang, S.; Wu, H. & Fu, A. W.-C. (2013). Tf-label: A topological-folding labeling scheme for reachability querying in a large graph. Em *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pp. 193--204, New York, NY, USA. ACM.

Cheng, J.; Yu, J. X.; Lin, X.; Wang, H. & Yu, P. S. (2006). Fast computation of reachability labeling for large graphs. Em *In Proc. of EDBT'06*.

Cheng, J.; Yu, J. X.; Lin, X.; Wang, H. & Yu, P. S. (2008). Fast computing reachability labelings for large graphs with high compression rate. Em *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*, pp. 193--204, New York, NY, USA. ACM.

Cohen, E.; Halperin, E.; Kaplan, H. & Zwick, U. (2003). Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5):1338--1355. ISSN 0097-5397.

Cormen, T. H.; Stein, C.; Rivest, R. L. & Leiserson, C. E. (2001). *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edição. ISBN 0070131511.

Cuzzocrea, A.; Saccà, D. & Ullman, J. D. (2013). Big data: a research agenda. Em *Proceedings of the 17th International Database Engineering & Applications Symposium*, IDEAS '13, pp. 198--203, New York, NY, USA. ACM.

Demetrescu, C. & Italiano, G. F. (2005). Trade-offs for fully dynamic transitive closure on dags: breaking through the o(n2 barrier. *J. ACM*, 52(2):147--156. ISSN 0004-5411.

Dhia, I. B. (2012). Access control in social networks: a reachability-based approach. Em *Proceedings of the 2012 Joint EDBT/ICDT Workshops*, pp. 227--232, New York, NY, USA. ACM.

Dietz, P. F. (1982). Maintaining order in a linked list. Em *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, STOC '82, pp. 122--127, New York, NY, USA. ACM.

Eades, P.; ElGindy, H.; Houle, M.; Lenhart, B.; Miller, M.; Rappaport, D. & Whitesides, S. (1994). Dominance drawings of bipartite graphs.

Fan, W. (2012). Graph pattern matching revised for social network analysis. Em *Proceedings of the 15th International Conference on Database Theory*, ICDT '12, pp. 8--21, New York, NY, USA. ACM.

Friedman, M. (1937). The Use of Ranks to Avoid the Assumption of Normality Implicit in the Analysis of Variance. *Journal of the American Statistical Association*, 32(200):675--701. ISSN 01621459.

Frigioni, D.; Miller, T.; Nanni, U. & Zaroliagis, C. (2001). An experimental study of dynamic algorithms for transitive closure. *J. Exp. Algorithmics*, 6. ISSN 1084-6654.

Garg, A. & Tamassia, R. (1995). Upward planarity testing. Em *SIAM Journal on Computing*, pp. 436--441.

GrammaTech, I. (2014). Dependence graphs and program slicing. http://www.grammatech.com/research/publications/dependence-graphs-and-program-slicing. Acessado em Dezembro de 2014.

Gupta, G. & Jayaraman, B. (1993). Analysis of or-parallel execution models. *ACM Trans. Program. Lang. Syst.*, 15(4):659--680. ISSN 0164-0925.

He, H.; Wang, H.; Yang, J. & Yu, P. S. (2005). Compact reachability labeling for graph-structured data. Em *Proceedings of the 14th ACM international conference on Information and knowledge management*, pp. 594--601, New York, NY, USA. ACM.

Healy, P. & Lynch, K. (2005). Fixed-parameter tractable algorithms for testing upward planarity. Em *Proceedings of the 31st International Conference on Theory and Practice of Computer Science*, SOFSEM'05, pp. 199--208, Berlin, Heidelberg. Springer-Verlag.

Hollander, M. & Wolfe, D. A. (1999). *Nonparametric Statistical Methods, 2nd Edition*. Wiley-Interscience, 2 edição. ISBN 0471190454.

Hopcroft, J. & Tarjan, R. (1974). Efficient planarity testing. *J. ACM*, 21(4):549--568. ISSN 0004-5411.

Horwitz, S.; Reps, T. & Binkley, D. (1988). Interprocedural slicing using dependence graphs. Em *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pp. 35--46, New York, NY, USA. ACM.

Jagadish, H. V. (1990). A compression technique to materialize transitive closure. *ACM Trans. Database Syst.*, 15(4):558--598. ISSN 0362-5915.

Jin, R.; Hong, H.; Wang, H.; Ruan, N. & Xiang, Y. (2010). Computing label-constraint reachability in graph databases. Em *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pp. 123--134, New York, NY, USA. ACM.

Jin, R.; Ruan, N.; Dey, S. & Xu, J. Y. (2012). Scarab: scaling reachability computation on large graphs. Em *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 169--180, New York, NY, USA. ACM.

Jin, R.; Xiang, Y.; Ruan, N. & Fuhry, D. (2009). 3-hop: a high-compression indexing scheme for reachability query. Em *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pp. 813--826, New York, NY, USA. ACM.

Jin, R.; Xiang, Y.; Ruan, N. & Wang, H. (2008). Efficiently answering reachability queries on very large directed graphs. Em *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 595--608, New York, NY, USA. ACM.

Kahn, A. B. (1962). Topological sorting of large networks. *Commun. ACM*, 5(11):558--562. ISSN 0001-0782.

Kalvin, A. D. & Varol, Y. L. (1983). On the generation of all topological sortings. *J. Algorithms*, 4(2):150–162.

Kameda, T. (1975). On the vector representation of the reachability in planar directed graphs. *Inf. Process. Lett.*, 3(3):75–77.

King, V. (1999). Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. Em *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, Washington, DC, USA. IEEE Computer Society.

Knuth, D. E. (1997). *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA. ISBN 0-201-89683-4.

Knuth, D. E. & Szwarcfiter, J. L. (1974). A structured program to generate all topological sorting arrangements. *Information Processing Letters - IPL*, 2(6):153--157.

Kornaropoulos, E. M. & Tollis, I. G. (2011). Weak dominance drawings and linear extension diameter. *CoRR*, abs/1108.1439.

Kornaropoulos, E. M. & Tollis, I. G. (2012). Overloaded orthogonal drawings. Em *Proceedings of the 19th international conference on Graph Drawing*, pp. 242--253, Berlin, Heidelberg. Springer-Verlag.

Kumar, R.; Raghavan, P.; Rajagopalan, S.; Sivakumar, D.; Tompkins, A. & Upfal, E. (2000). The web as a graph. Em *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '00, pp. 1--10, New York, NY, USA. ACM.

Lempel, R. & Moran, S. (2000). The stochastic approach for link-structure analysis (salsa) and the tkc effect. Em *Proceedings of the 9th international World Wide Web conference on Computer networks*, pp. 387--401, Amsterdam, The Netherlands, The Netherlands. North-Holland Publishing Co.

Mehta, D. P. & Sahni, S. (2004). *Handbook Of Data Structures And Applications (Chapman & Hall/Crc Computer and Information Science Series.)*. Chapman & Hall/CRC. ISBN 1584884355.

Nishizeki, T. & Chiba, N. (2008). *Planar Graphs: Theory and Algorithms*. Dover Books on Mathematics Series. Dover Publications. ISBN 9780486466712.

Nishizeki, T. & Rahman, M. (2004). *Planar Graph Drawing*. Lecture notes series on computing. World Scientific. ISBN 9789812560339.

Nuutila, E. (1995). *Efficient Transitive Closure Computation in Large Digraphs*. Tese de doutorado, Finnish Academy of Technology.

Papakostas, A. (1994). Upward planarity testing of outerplanar dags. Em Tamassia, R. & Tollis, I. G., editores, *Graph Drawing*, volume 894 of *Lecture Notes in Computer Science*, pp. 298–306. Springer.

Pearce, D. J. & Kelly, P. H. J. (2006). A dynamic topological sort algorithm for directed acyclic graphs. *J. Exp. Algorithmics*, 11:1.7. ISSN 1084-6654.

Pearce, D. J. & Kelly, P. H. J. (2010). A batch algorithm for maintaining a topological order. Em *Proceedings of the Thirty-Third Australasian Conferenc on Computer Science - Volume 102*, ACSC '10, pp. 79--88, Darlinghurst, Australia, Australia. Australian Computer Society, Inc.

Roditty, L. (2008). A faster and simpler fully dynamic transitive closure. *ACM Trans. Algorithms*, 4(1):6:1--6:16. ISSN 1549-6325.

Roditty, L. & Zwick, U. (2004). A fully dynamic reachability algorithm for directed graphs with an almost linear update time. Em *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, STOC '04, pp. 184--191, New York, NY, USA. ACM.

Sairam, S.; Cohen, R. F.; Tamassia, R. & Vitter, J. S. (1990). Fully dynamic techniques for reachability in planar st-graphs. Relatório técnico CS-91-02, Department of Computer Science, Brown University.

Schenkel, R.; Theobald, A. & Weikum, G. (2004). Hopi: An efficient connection index for complex xml document collections. Em *In 9th Int. Conference on Extending Database Technology (EDBT*, pp. 237--255.

Schenkel, R.; Theobald, A. & Weikum, G. (2005). Efficient creation and incremental maintenance of the hopi index for complex xml document collections. Em *Proceedings of the 21st International Conference on Data Engineering*, pp. 360--371, Washington, DC, USA. IEEE Computer Society.

Seufert, S.; Anand, A.; Bedathur, S. J. & Weikum, G. (2013). FERRARI: Flexible and Efficient Reachability Range Assignment for Graph Indexing. Em *ICDE'13: Proceedings of the 29th IEEE International Conference on Data Engineering*. IEEE.

Siganos, G.; Tauro, S. L. & Faloutsos, M. (2006). Jellyfish: A conceptual model for the as internet topology. *Journal of Communications and Networks*, 8(3):339–350.

Simon, K. (1988). An improved algorithm for transitive closure on acyclic digraphs. *Theor. Comput. Sci.*, 58(1-3):325--346. ISSN 0304-3975.

Tarjan, R. (1972). Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160.

Tarjan, R. E. (1975). Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215--225. ISSN 0004-5411.

Tip, F. (1995). A survey of program slicing techniques. *JOURNAL OF PROGRAMMING LANGUAGES*, 3:121--189.

Trißl, S. & Leser, U. (2007). Fast and practical indexing and querying of very large graphs. Em *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pp. 845--856, New York, NY, USA. ACM.

van Schaik, S. J. & de Moor, O. (2011). A memory efficient reachability data structure through bit vector compression. Em *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pp. 913--924, New York, NY, USA. ACM.

Veloso, R. R.; Cerf, L.; Junior, W. M. & Zaki, M. J. (2014). Reachability queries in very large graphs: A fast refined online search approach. Em *Proc. 17th International Conference on Extending Database Technology (EDBT), Athens, Greece, March 24-28, 2014.*, pp. 511--522.

Wang, H.; He, H.; Yang, J.; Yu, P. S. & Yu, J. X. (2006). Dual labeling: Answering graph reachability queries in constant time. Em *in Proc. 22nd International Conference on Data Engineering*, p. 75. IEEE Computer Society.

Wilson, C.; Sala, A.; Puttaswamy, K. P. N. & Zhao, B. Y. (2012). Beyond social graphs: User interactions in online social networks and their implications. *ACM Trans. Web*, 6(4):17:1--17:31. ISSN 1559-1131.

Wu, K.; Otoo, E. J. & Shoshani, A. (2006). Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 31(1):1--38. ISSN 0362-5915.

Yildirim, H.; Chaoji, V. & Zaki, M. J. (2010). Grail: Scalable reachability index for large graphs. *PVLDB*, 3(1):276–284.

Yildirim, H.; Chaoji, V. & Zaki, M. J. (2013). Dagger: A scalable index for reachability queries in large dynamic graphs. *CoRR*, abs/1301.0977.

Zhu, A. D.; Lin, W.; Wang, S. & Xiao, X. (2014). Reachability queries on large dynamic graphs: A total order approach. Em *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pp. 1323--1334, New York, NY, USA. ACM.

Apêndice A

Apêndice

A.1 Gráficos complementares do Capítulo 4

A.1.1 Complemento da Figura 4.15

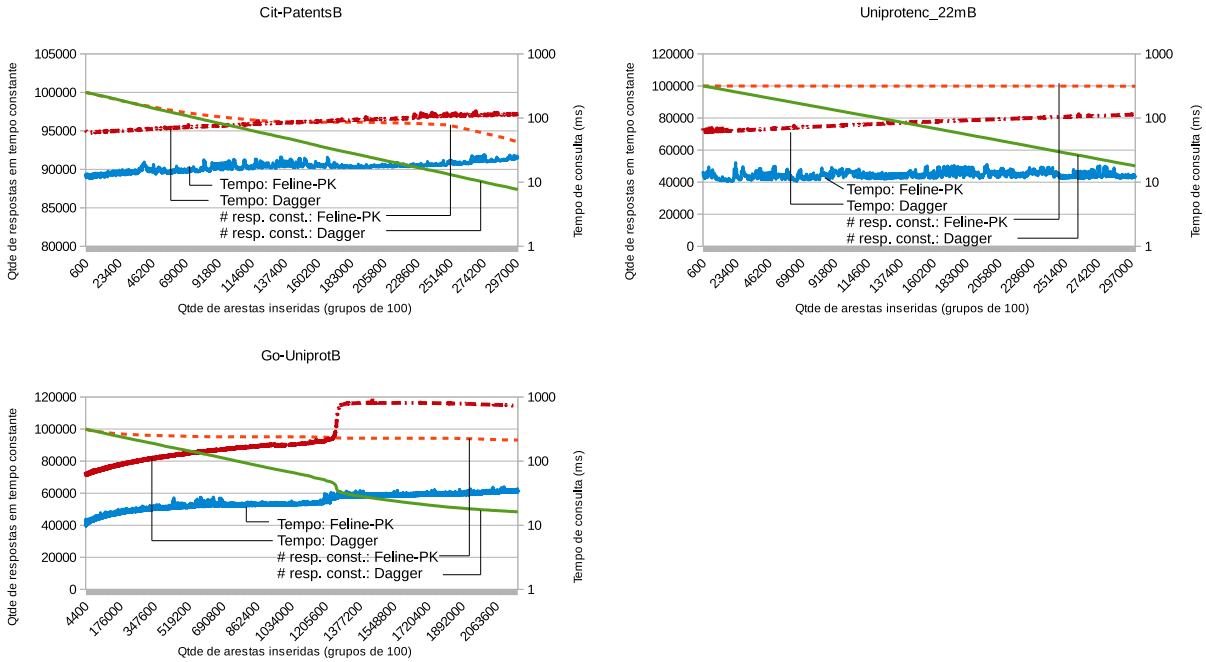


Figura A.1. Os gráficos mostram uma comparação no desempenho das consultas entre os métodos Feline e Dagger, levando em consideração o tempo de consulta e a quantidade de respostas em tempo constante. Cada gráfico apresenta os tempos e quantidade de respostas para cada 100 mil consultas realizadas a cada 100 inserções de novas arestas.

A.1.2 Complementos da Figura 4.20

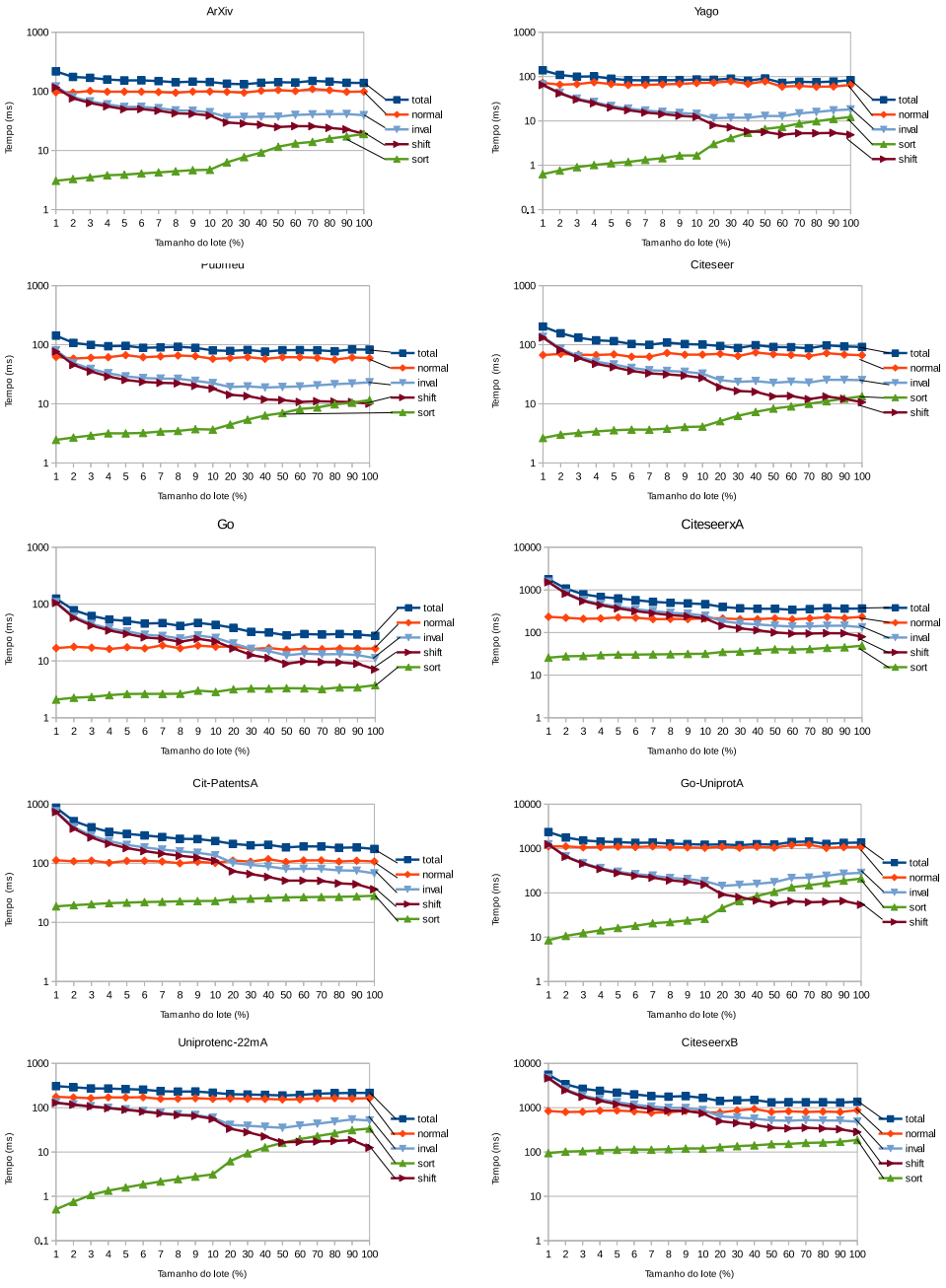


Figura A.2. Tempos de inserção total dos lotes e a contribuição de cada fase do algoritmo de inserção nesse tempo. Na legenda: "normal" refere-se ao tempo para inserção de arestas no grafo. "inval" é tempo necessário para organizar as arestas invalidantes e atualizar o índice; esse último é composto por "sort" e "shift", que são os tempos para ordenar as arestas invalidantes, juntamente com as funções *discorer* e *shift*.

A.1.3 Complementos da Figura 4.21

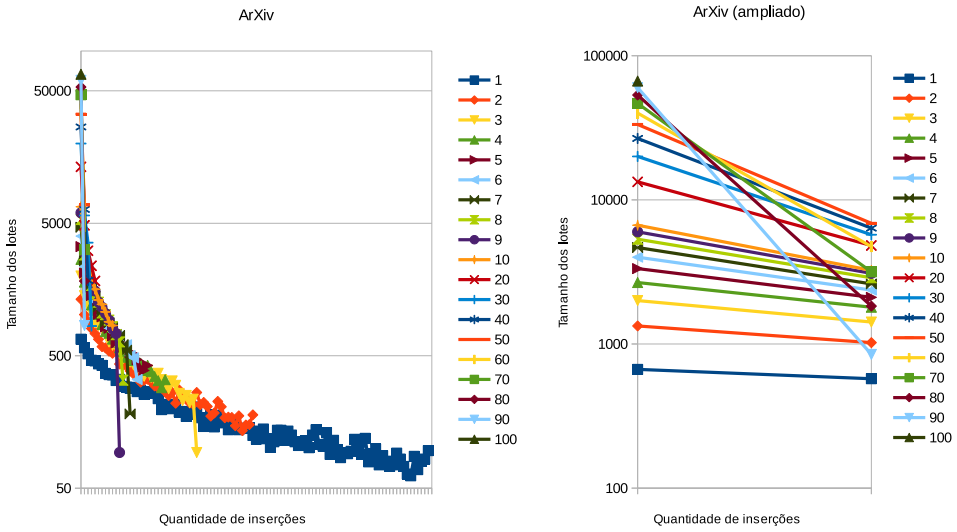


Figura A.3. Quantidade de arestas invalidantes à medida que os lotes são inseridos no grafo Citeseer. Na legenda, os tamanhos estão em %. O gráfico menor à esquerda dá destaque para os lotes de tamanhos 1% e 2%.

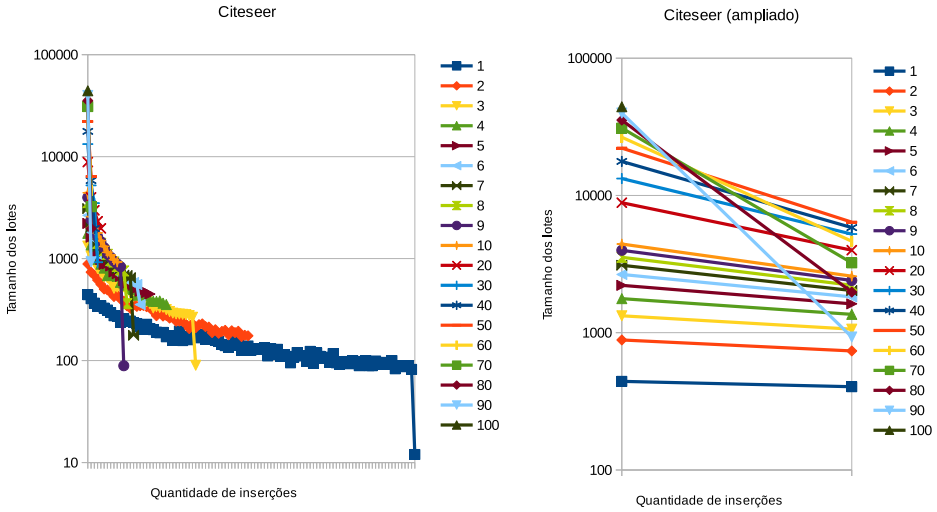


Figura A.4. Quantidade de arestas invalidantes à medida que os lotes são inseridos no grafo Citeseer. Na legenda, os tamanhos estão em %. O gráfico menor à esquerda dá destaque para os lotes de tamanhos 1% e 2%.

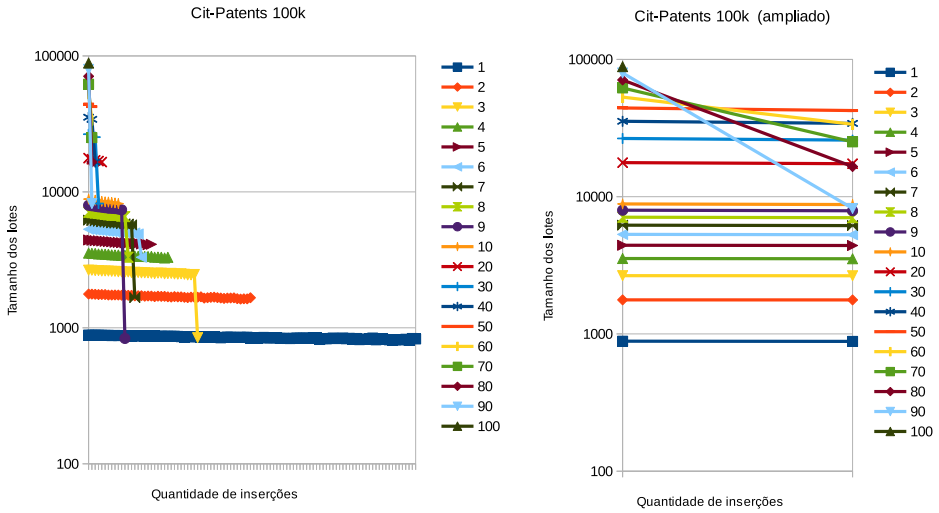


Figura A.5. Quantidade de arestas invalidantes à medida que os lotes são inseridos no grafo Cit-Patents. Na legenda, os tamanhos estão em %. O gráfico menor à esquerda dá destaque para os lotes de tamanhos 1% e 2%.

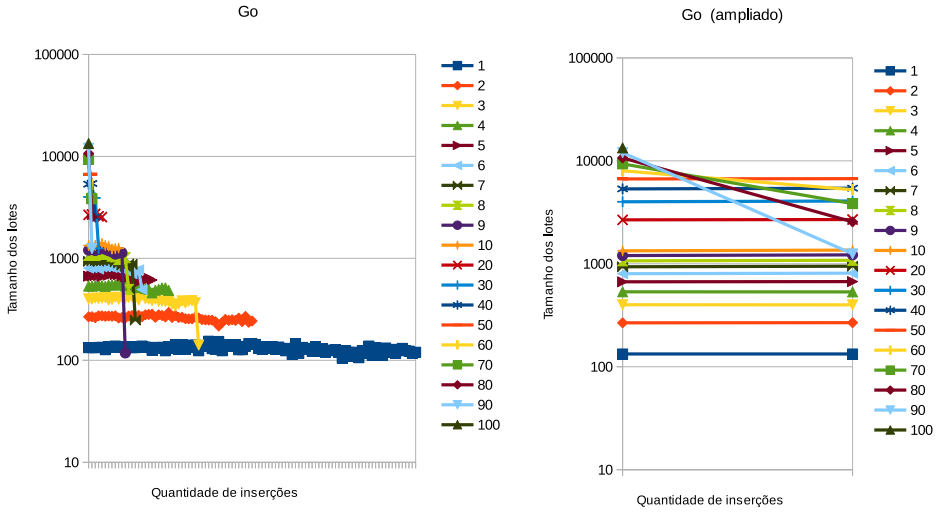


Figura A.6. Quantidade de arestas invalidantes à medida que os lotes são inseridos no grafo Go. Na legenda, os tamanhos estão em %. O gráfico menor à esquerda dá destaque para os lotes de tamanhos 1% e 2%.

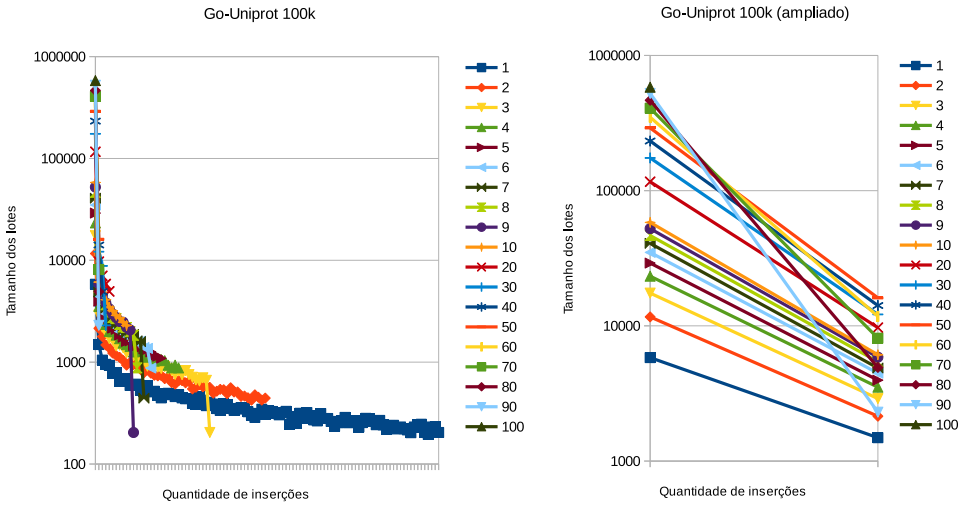


Figura A.7. Quantidade de arestas invalidantes à medida que os lotes são inseridos no grafo Go-Uniprot. Na legenda, os tamanhos estão em %. O gráfico menor à esquerda dá destaque para os lotes de tamanhos 1% e 2%.

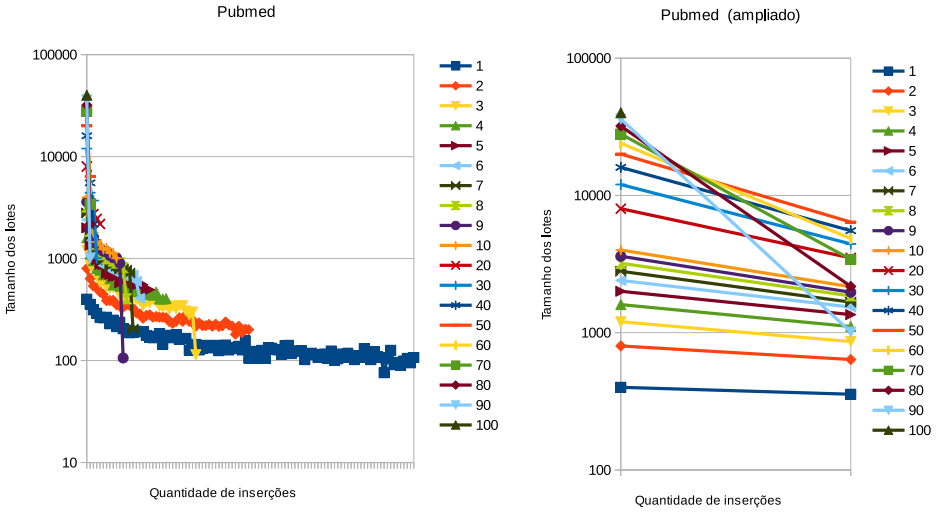


Figura A.8. Quantidade de arestas invalidantes à medida que os lotes são inseridos no grafo Pubmed. Na legenda, os tamanhos estão em %. O gráfico menor à esquerda dá destaque para os lotes de tamanhos 1% e 2%.

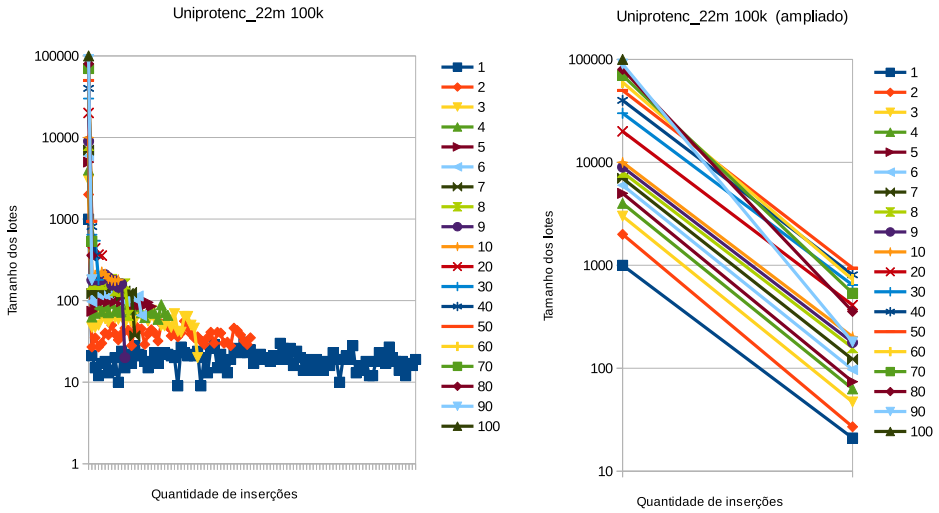


Figura A.9. Quantidade de arestas invalidantes à medida que os lotes são inseridos no grafo Uniprotenc-22m. Na legenda, os tamanhos estão em %. O gráfico menor à esquerda dá destaque para os lotes de tamanhos 1% e 2%.

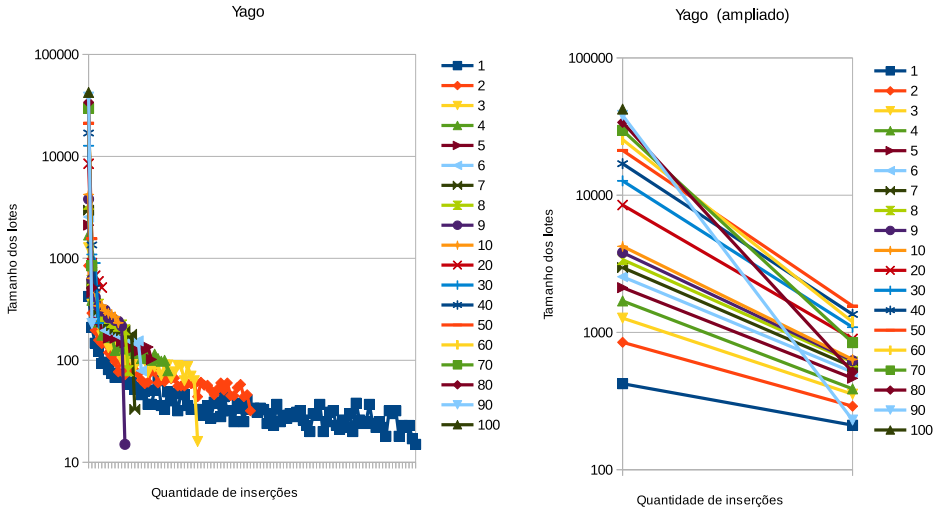


Figura A.10. Quantidade de arestas invalidantes à medida que os lotes são inseridos no grafo Yago. Na legenda, os tamanhos estão em %. O gráfico menor à esquerda dá destaque para os lotes de tamanhos 1% e 2%.