

Processamento aproximado de grafos

Renato David Silva Rosa

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática e de Computadores

Orientador: Prof. Luís Manuel Antunes Veiga

Júri

Presidente:	Doutor Alberto Manuel Rodrigues da Silva
Vogal:	Doutor Luís Manuel Antunes Veiga
Vogal:	Doutor Pável Pereira Calado

Outubro de 2016

Agradecimentos

Agradeço ao meu orientador, o Prof. Luís Veiga, por, no meio da sua intensa vida de ensino e investigação, ter estado presente durante a elaboração deste trabalho e ter dado um contributo sempre atempado e oportuno.

Agradeço ao Sérgio Esteves pela co-orientação, pelo interesse e pelo auxílio sempre pronto.

Por fim, agradeço a todos os meus amigos e familiares que me apoiaram e incentivaram na elaboração deste trabalho.

Lisboa, 23 de Novembro de 2016

Renato Rosa

Resumo

Os grafos são utilizados para representar variadas realidades do mundo atual, e apresentam desafios específicos em termos do seu processamento em grande escala, sobretudo devido à sua dimensão e contínua mutabilidade.

Começa por examinar-se o trabalho relacionado em termos de sistemas de processamento em bloco e de *streams*, técnicas especializadas para grafos e processamento aproximado, com a adequada categorização e contextualização.

Nesse contexto, apresenta-se a biblioteca GraphApprox, concebida para ser utilizada em conjunto com o Apache Flink. O seu objetivo consiste em utilizar resultados aproximados e computação diferida como meio de otimizar o uso dos recursos computacionais e os tempos de resposta, aumentando a escalabilidade do sistema.

O utilizador define um grafo inicial e uma *stream* de atualizações e consultas. Quando uma consulta é recebida, o utilizador, por meio de funções *callback*, pode definir o tipo de processamento a efetuar, exato, aproximado, ou a repetição da resposta anterior. Para essa decisão, tem acesso ao estado atual do grafo, bem como às atualizações entretanto recebidas e a estatísticas sobre as mesmas. As atualizações ao grafo são monitorizadas, a fim de poderem eficientemente obter-se estatísticas e conjuntos de vértices de interesse do grafo.

Foi ainda implementado um algoritmo de aproximação ao PageRank, baseado na sumarização do grafo original.

Por fim, é apresentada a arquitetura da solução implementada, as estruturas de dados e algoritmos definidos, bem como pormenores de implementação, concluindo-se com uma avaliação qualitativa e quantitativa da solução desenvolvida.

Palavras-chave

Grafo

PageRank

Processamento aproximado

Apache Flink

Abstract

Graphs are used to represent different realities of the current world, and offer particular challenges in terms of its large-scale processing, especially due to their size and continuous changeability.

At first, related work in terms of batch and stream processing, techniques for graphs, and approximate processing, is revised, with proper categorization and contextualization.

In this context, GraphApprox library, designed to be used with Apache Flink, is presented. Its goal is to use approximate results and deferred computing as a way to optimize the use of computing resources and response times, increasing system scalability.

The user defines an initial graph and a stream of updates and queries. When a query is received, the user, through callback functions, can define the type of processing to perform: exact, approximate, or the repetition of the previous answer. For this decision, the current state of the graph, as well as the updates and statistics on them, are provided. The updates to the graph are monitored, in order to efficiently obtain statistics and sets of vertices of interest from the graph.

An approximation algorithm for PageRank, based on the summarization of the original graph, has been implemented.

Finally, the architecture of the designed solution is presented, with the data structures and algorithms defined, as well as details of implementation. The work concludes with a qualitative and quantitative evaluation of the developed solution.

Keywords

Graph

PageRank

Approximate processing

Apache Flink

Índice

1	Introdução	10
1.1	Motivação	10
1.1.1	Casos de uso	11
1.2	Desafios e dificuldades	11
1.2.1	Representação	12
1.2.2	Dimensão	12
1.2.3	Heterogeneidade	12
1.2.4	Mutabilidade	13
1.3	Limitações das soluções existentes	13
1.4	Objetivos	13
1.5	Contribuições	14
1.6	Organização do documento	15
2	Trabalho relacionado	16
2.1	Processamento em bloco e em tempo real	16
2.1.1	Processamento em bloco: MapReduce	16
2.1.2	Processamento iterativo	18
2.1.3	<i>Streams</i>	18
2.1.4	Conclusões	20
2.2	Processamento de grafos	21
2.2.1	Computação centrada no vértice	22
2.2.2	<i>Streams</i> em grafos	28
2.2.3	Outras abordagens	28
2.3	Processamento aproximado	29
2.3.1	Conceito	30
2.3.2	Técnicas	31
3	Arquitetura da solução	34
3.1	Processamento aproximado	34
3.1.1	PageRank	34
3.2	Informação/dados	35
3.3	Requisitos	35
3.4	Arquitetura	36
3.4.1	Sistema de processamento de grafos	36
3.4.2	<i>Stream</i> de atualizações	37
3.4.3	GraphApprox: biblioteca de processamento aproximado de grafos	38
3.4.4	Utilizador	38

3.5	Estruturas de dados	38
3.5.1	Vértices, arcos e grafos	39
3.5.2	UpdateInfo	39
3.5.3	GraphUpdates	39
3.5.4	GraphUpdateStatistics	40
3.5.5	Config	40
3.6	Algoritmos base	41
3.6.1	Monitorização de atualizações ao grafo	41
3.6.2	Seleção de vértices e expansão à vizinhança	42
3.6.3	Grafo sumário	43
3.6.4	PageRank aproximado	44
3.7	A API	44
3.7.1	Inicialização	45
3.7.2	Funções <i>callback</i> definidas pelo utilizador	46
3.8	Algoritmo global de processamento aproximado	50
4	Implementação	52
4.1	Apache Flink	52
4.1.1	API Flink/Gelly	53
4.1.2	Questões práticas de desempenho	53
4.2	Organização da solução	54
4.2.1	GraphApprox	55
4.2.2	GraphAlgorithms	56
4.2.3	Tester	57
5	Avaliação	59
5.1	Aspetos qualitativos	59
5.2	Datasets de validação	60
5.2.1	PolBlogs	60
5.2.2	Cit-HepPh	60
5.2.3	Facebook	61
5.3	Cenário experimental	62
5.4	Metodologia de avaliação de <i>rankings</i>	62
5.5	Inicialização e monitorização de atualizações	63
5.6	Qualidade da resposta	64
5.6.1	Impacto do Valor mínimo da alteração de grau	65
5.6.2	Impacto da dimensão da vizinhança	66
5.7	Dimensão do grafo sumário	66
5.8	Desempenho	67
5.9	Repetição da última resposta	69
5.10	Conclusão	71
6	Conclusões e trabalho futuro	75

Lista de Figuras

2.1	PageRank em Pregel	24
2.2	PageRank no modelo GAS	25
3.1	Arquitetura geral da solução	37
3.2	Algoritmo global de processamento aproximado	51
4.1	<i>Stack</i> do Apache Flink	53
4.2	Modelo de processo do Apache Flink	54
4.3	Diagrama de classes da biblioteca GraphApprox	55
4.4	SocketStreamProvider	56
4.5	Exemplo de utilização do GraphApprox para processamento dum grafo real	58
5.1	Tempo acumulado despendido com a monitorização, entre cada consulta ao grafo	65
5.2	Valores de RBO para $n = 0$ e diferentes t	66
5.3	Valores de RBO para $n = 1$ e diferentes t	67
5.4	Valores de RBO para $t = 0,00$ e diferentes n	68
5.5	Valores de RBO para $t = 0,20$ e diferentes n	69
5.6	Fração de arcos do grafo sumário para $n = 0$ e diferentes t , no <i>dataset</i> Facebook	70
5.7	Fração de arcos do grafo sumário para $t = 0,10$ e diferentes n , no <i>dataset</i> Facebook	70
5.8	Medidas de desempenho que ilustram uma boa relação qualidade-uso de recursos	71
5.9	Ganhos em desempenho à custa de qualidade dos resultados	72
5.10	Alteração do desempenho no decorrer do processamento	73
5.11	Exemplo de insucesso da abordagem	73
5.12	Qualidade e desempenho com estratégia de repetição da resposta anterior	74

Lista de Algoritmos

1	Adicionar arco	41
2	Remover arco	41
3	Reinicializar registo	42
4	PageRank	44
5	PageRank aproximado	45
6	Cálculo do PageRank aproximado	45
7	Processamento aproximado	50

Acrónimos

API	Application Programming Interface
BD	base de dados
BSP	Bulk Synchronous Parallel
CPU	Central Processing Unit
CRUD	Create-Read-Update-Delete
CSV	Comma Separated Values
DAG	grafo orientado acíclico (<i>directed acyclic graph</i>)
FDU	função definida pelo utilizador
GAS	Gather-Apply-Scatter
GML	Graph Modelling Language
HDFS	Hadoop Distributed File System
I/O	Input/Output
IP	Internet Protocol
JAR	Java Archive
JVM	Java Virtual Machine
POJO	Plain Old Java Object
QoD	Quality-of-Data
QoS	Quality-of-Service
RBO	Rank-biased Overlap
RDD	Resilient Distributed Dataset
SSSP	Single source shortest path
WWW	World Wide Web

Capítulo 1

Introdução

Muitos objetos e realidades do nosso quotidiano, quer naturais quer criados pelo homem, assumem a forma de redes, com elementos individuais que se relacionam entre si [19]. Estas redes podem ser representadas formalmente na forma de grafos.

Os grafos e o seu estudo estão presentes em muitos campos do conhecimento: não só na matemática, nomeadamente na teoria de grafos, que os estuda exaustivamente, mas também nas ciências sociais e naturais, como a biologia e a física. É na teoria matemática de grafos que encontramos porém a formalização do seu estudo, com importantes resultados teóricos e aplicações práticas.

A necessidade de resolver problemas que envolvem grafos em cada domínio levou, porém, a que outras ciências contribuíssem para enriquecer o domínio teórico com intuições-chave, conceitos inovadores e importantes aplicações [19, pp. 1-2].

Existem, de facto, abundantes aplicações dos grafos, bem como diversas funções e medidas que podem ser computadas com base neles: caminhos entre vértices, distâncias, componentes, cortes, medidas de centralidade, identificação de comunidades (*clusters*), entre muitas outras.

1.1 Motivação

O processamento de grafos envolve estruturas de grande dimensão e, mais recentemente, também em constante mutação. Os sistemas distribuídos são uma ferramenta bastante utilizada para o processamento em larga escala, com resultados em tempo útil. No entanto, em qualquer um destes sistemas, existe permanentemente a necessidade de melhorar e otimizar a utilização de recursos computacionais e o tempo de resposta, o que se aplica também ao caso concreto dos grafos.

Consideremos, para efeitos de definição do nosso objeto de estudo, sistemas distribuídos de processamento de grafos com suporte a atualização dos mesmos. O caso típico é aquele em que o sistema possui uma representação do grafo e recebe continuamente atualizações, que aplica ao grafo. Consideramos que existe simultaneamente uma função de interesse, que deve ser computada a partir do grafo, e que as alterações ao grafo implicam a sua recomputação, seja ela realizada de raiz ou incrementalmente, sobre o resultado anterior.

Em *workflows* de processamento em grande escala, bem como nos componentes individuais que os constituem, existe consumo de CPU, memória, I/O, e também de tempo necessário para computar uma resposta atualizada. Ora, acontece que, por vezes, a diferença entre o resultado da recomputação e o resultado obtido anteriormente não é significativa, não obstante todo o gasto decorrente do processamento. Em certos domínios, todavia, é tolerável admitir uma resposta não exata, mas suficientemente aproximada, se isso significar uma poupança significativa em termos de recursos computacionais e

maior rapidez. Neste caso, o sistema poderia calcular um resultado aproximado, ou limitar-se a devolver a resposta anterior, adiando a recomputação completa para quando fosse expectável que a mesma produzisse resultados significativos.

Estamos perante uma necessidade de compromisso entre a correção da resposta dada pelo sistema e a otimização do uso dos recursos utilizados para obter a mesma resposta [32, 34], o que pode ser de interesse para problemas muito concretos, como vemos ee seguida.

1.1.1 Casos de uso

Apresentamos agora alguns exemplos de casos de uso em que o processamento aproximado de grafos pode fornecer uma resposta mais eficaz, sem comprometer a funcionalidade desejada.

Trending topics Numa rede social *online*, tal como o Facebook, o Twitter, ou um *feed* de notícias, encontramos grafos que integram, entre outras coisas, conteúdos relacionados entre si, os temas que interessam a determinado utilizador, e os utilizadores e as relações entre eles. Com este grafo, é possível conceber um sistema que, em tempo real, sugere a determinado utilizador conteúdos que potencialmente lhe possam interessar. A determinação destes conteúdos exige processar um grafo mas, caso a computação demore demasiado tempo, a resposta pode ser já inútil ou obsoleta.

Por outro lado, havendo milhares, ou mesmo milhões, de utilizadores *online*, este processo tem de ser escalável e altamente otimizado. Note-se ainda que uma resposta apenas aproximada é tolerável: o facto de um determinado conteúdo não ser sugerido em primeiro lugar, como deveria, mas em segundo ou terceiro, não traz necessariamente consequências de monta, ou é, pelo menos, muito preferível à ausência de resposta atempada.

Sistema de recomendação Este caso de uso é semelhante ao anterior, mas tem agora como alvo sites de compras *online*. Neste caso o grafo é formado pelos produtos disponíveis e respetivas categorias, pelos utilizadores, os seus interesses e as conexões com outros utilizadores, e ainda pelos produtos que são comprados em conjunto (um grafo pesado é uma boa forma de representar a frequência com que determinado par de produtos é adquirido em conjunto). A partir de tal grafo, é possível, por exemplo, recomendar outros itens que potencialmente interessem ao cliente, o que deverá ser feito de forma rápida e eficiente, caso contrário pode falhar-se o objetivo.

Deteção de fraude A deteção de fraude, num domínio que assuma a forma dum grafo, por exemplo em contexto bancário, tendo em conta clientes, moradas, números telefónicos, endereços IP, etc. [66], a ser feita em tempo real, requer rapidez de resposta, incompatível com longos tempos de processamento. Além disso, é exemplo dum caso em que os requisitos de exatidão se apresentam de forma assimétrica: não devem admitir-se falsos negativos, ao passo que alguns falsos positivos são toleráveis.

Posto isto, o processamento de grafos apresenta também particulares desafios computacionais, como veremos agora.

1.2 Desafios e dificuldades

Os grafos que encontramos no mundo real comportam importantes desafios em termos de representação, armazenamento e processamento. As suas características, bem como a sua importância, motivaram o aparecimento de grande número de técnicas, algoritmos e sistemas especialmente desenhados tendo em vista a eficiência e a rapidez de resposta [4, 43].

Alguns dos desafios concretos colocados pelos grafos reais são as diversas formas possíveis de os representar e armazenar, a sua dimensão, a sua heterogeneidade e a sua mutabilidade, como veremos de seguida.

1.2.1 Representação

Os grafos são constituídos por vértices e arcos. Uns e outros podem ter associados valores ou objetos. Contudo, existem muitas formas diferentes de representar um grafo, e cada uma apresenta as suas vantagens e desvantagens. Vejamos alguns breves exemplos.

A forma porventura mais simplificada de representar um grafo consiste em enumerar os seus arcos. Esta representação é simples e económica em termos de espaço, mas é bastante inconveniente quando se pretende, por exemplo, iterar sobre os vértices, encontrar a vizinhança de cada um, etc.

No extremo oposto, a matriz de adjacências torna certas operações bastante simples, tais como iteração de vértices ou testar se determinado arco existe, permitindo também a aplicação de algoritmos matriciais a grafos, mas já não é tão conveniente, por exemplo, para atravessar a vizinhança dum vértice.

Outra representação comum, a lista de adjacências, torna a iteração sobre a vizinhança dum vértice bastante eficiente, mas para testar a existência de arcos no grafo tem claramente um desempenho inferior [26, Cap. 22.1].

A conclusão que se pode tirar é que a representação dum grafo está intimamente ligada com o problema que se pretende resolver e com as ferramentas algorítmicas utilizadas para o efeito. Este facto levou ao surgimento de representações híbridas e otimizações dedicadas. Alguns sistemas de processamento de grafos possuem os seus próprios esquemas de representação, mais ou menos flexíveis para o utilizador.

1.2.2 Dimensão

Intimamente ligada à questão da representação está a da dimensão dos grafos quotidianos.

Alguns grafos da vida real são enormes para a escala humana. Talvez o melhor exemplo seja a World Wide Web (WWW), com um tamanho aproximado de 50 mil milhões de vértices (páginas web), somente na sua porção possível de indexar pelos motores de busca [85]. Mesmo as redes humanas/sociais podem ser bastante grandes. A rede de amigos do Facebook, ou os *retweets* e respostas no Twitter, são hoje alguns exemplos.

A teoria computacional de grafos contempla diferentes formas de representação, com diferentes implicações em termos do espaço ocupado e de eficiência das operações comuns num grafo. Por exemplo, as matrizes de adjacências apresentam algumas propriedades bastante desejáveis, mas também um elevado preço a pagar em termos de espaço utilizado: para um grafo com n vértices requer-se um espaço de $O(n^2)$, o que para a WWW é impraticável. A sua dimensão requer de facto técnicas de representação avançadas [23, 24].

A importância dos grafos e a necessidade de os armazenar e processar de forma rápida e eficiente veio dar origem a bases de dados especializadas, a sistemas distribuídos dedicados ao seu processamento em larga escala e em tempo real, e a *frameworks* ou bibliotecas para tratamento de grafos em sistemas de processamento distribuído mais genéricos, como veremos na Sec. 2.2.

1.2.3 Heterogeneidade

À grande dimensão das redes que encontramos na natureza e na sociedade acresce uma característica que lança importantes desafios: a sua heterogeneidade.

Heterogeneidade aqui significa a diversidade que existe no interior do próprio grafo. Em muitos grafos reais, como por exemplo, mais uma vez, a WWW, encontramos um grande número de vértices com grau reduzido, mas também um número reduzido de vértices, conhecidos por *hubs*, com um número muito elevado de ligações. Neste tipo de grafos, a distribuição de grau segue uma *lei de potência* nestes grafos, que são frequentemente *livres de escala* [17, Sec. 4.2-4.4]. Foi descoberto há relativamente pouco tempo que muitos grafos da vida real importantes e relevantes apresentam estas características, pelo que os grafos livres de escala são um hoje um vasto campo de estudo, o que contrasta com a teoria de grafos tradicional [18].

Esta heterogeneidade potencia alguns problemas para um sistema distribuído de processamento de grafos. Por exemplo, seguindo uma estratégia simples de partição de vértices pelo nós do sistema, a distribuição da carga de trabalho pode facilmente apresentar grandes desequilíbrios, pela presença de *hubs*, aos quais se associa, tipicamente, uma necessidade de processamento muito superior.

1.2.4 Mutabilidade

Um outro desafio prende-se com o facto de que a maior parte dos grafos evolui ao longo do tempo, quer na sua estrutura, quer nas propriedades que estão associadas aos seus elementos. Um objeto em constante mudança requer cuidados específicos. Podemos facilmente encontrar exemplos: páginas e ligações na WWW são criadas a cada momento; mensagens, *retweets* e respostas no Twitter são constantes; compras em lojas *online* estão permanentemente a ser realizadas.

Existem muitas técnicas e algoritmos para grafos, mas supõem que temos um grafo imutável, ou pelo menos relativamente estável. A sua adaptação para grafos que se alteram continuamente nem sempre é trivial. O problema da volatilidade dos grafos reais é uma das motivações para a contribuição que esperamos fornecer com este trabalho.

1.3 Limitações das soluções existentes

Como veremos no capítulo 2, existe ampla investigação no campo do processamento em grande escala, com especialização nos grafos. Também existe trabalho na área do processamento aproximado. A par da investigação, existe também considerável oferta de soluções funcionais, de âmbito académico ou industrial, comercial ou *open source*.

Não se encontram porém, pelo menos facilmente, respostas especializadas que ofereçam, de forma integrada, todas estas funcionalidades. Como veremos, existe resposta para a necessidade de processar grafos em grande escala e iterativamente, contudo o processamento aproximado não se encontra incluído, usualmente. Por outro lado, podem encontrar-se soluções de processamento aproximado inseridas em sistemas de processamento em grande escala, mas sem que as mesmas sejam especializadas em grafos.

1.4 Objetivos

O objetivo do nosso trabalho é pois conceber e implementar uma ferramenta computacional que sirva de meio para otimizar o uso de recursos computacionais e também o tempo de resposta em sistemas de processamento de grafos.

Partindo dum conceito já aplicado a armazenamento replicado e processamento na *nuvem* [33] e a *workflows* de processamento [32], que consiste na avaliação do impacto de novos dados e na recomputação adiada, segundo um modelo de Quality-of-Data (QoD) [34] definido pelo utilizador, propõe-se, no

presente trabalho, fornecer as ferramentas que permitam aplicar este mesmo conceito a um novo nível e num âmbito mais específico, o do processamento de grafos.

Partimos da base do processamento incremental de grafos, ou seja, o processamento de atualizações a grafos de forma incremental, e que pretendemos agora estender de forma a que realize processamento aproximado.

Como vimos, os grafos, além de relevantes na representação de problemas, apresentam especificidades que os distinguem doutras classes de problemas, tanto que motivaram a criação de sistemas dedicados ao seu processamento. Assim, a aplicação deste modelo a grafos comporta desafios que justificam uma abordagem especializada.

Contudo, cada problema real tem características diferentes, e o mesmo se aplica aos grafos concretos. Ao falarmos de grafos em constante atualização, torna-se claro que não é possível ter, de forma prática, uma solução universalmente eficaz e eficiente. Assim, o nosso objetivo é proporcionar os meios, sob a forma de biblioteca ou API, que permitam raciocinar sobre as alterações aos grafos e os resultados anteriores, bem como os requisitos em termos de QoD, e tomar decisões sobre o processamento a fazer.

Uma tal API poderá então ser conectada, por exemplo, a um sistema inteligente de qualquer género, a fim de implementar de forma completa um sistema de processamento para grafos, com computação aproximada, e adaptado a cada caso. O sistema poderá então estar apto a prever quando a recomputação não vai produzir alterações significativas no resultado, e caso contrário, decidir efetuar o processamento.

Isto implica a existência dum critério para decidir o que são ou não alterações significativas num grafo, e com que medida de relevância. Pode definir-se por exemplo, uma diferença máxima relativa entre os dados de entrada anteriores e os mais recentes ou, em alternativa, uma diferença máxima entre o estado atual e uma previsão dos novos resultados, a partir da qual a computação tem de ser repetida.

Para esse efeito, há que encontrar um modo de estimar a diferença relativa entre o resultado anterior e o resultado que se obteria se a recomputação fosse realizada ou, por outras palavras, estimar o erro acumulado com confiança suficiente. Para isso, porém, é necessário poder efetuar uma previsão da diferença relativa conhecendo apenas o *input*, ou seja, a *stream* de alterações ao grafo desde a última computação.

O utilizador do sistema deve poder exprimir o que considera um desvio aceitável em relação aos valores exatos, e estabelecer os seus objetivos em termos do compromisso entre correção e otimização, tendo por base que são como inversas uma da outra: maior otimização do uso de recursos implica maior erro e incerteza, e exatidão dos resultados exige maior processamento.

1.5 Contribuições

O presente trabalho insere-se no contexto do processamento de grafos em tempo real, e oferece as seguintes contribuições:

1. Análise da literatura sobre a temática do processamento de dados em grande escala e de *streams*;
2. Uma visão de conjunto e pormenorizada sobre diferentes estratégias para resolução de problemas em grafos;
3. Discussão sobre várias técnicas de processamento aproximado e da sua aplicabilidade;
4. Apresentação duma arquitetura/API genérica que permite utilizar dados acerca das alterações a um grafo e dos resultados anteriores para tomar decisões relativas ao processamento;

5. Implementação da API como uma biblioteca/sistema associado ao Apache Flink, que se intitulou GraphApprox;
6. Desenho e implementação duma versão aproximada do algoritmo PageRank, adaptando um modelo assente em sumarização do grafo;
7. Avaliação da solução com diferentes grafos reais e algoritmos, fazendo uso do processamento aproximado como ferramenta de otimização de recursos.

1.6 Organização do documento

O restante do presente documento organiza-se como a seguir se descreve.

No capítulo 2 descreve-se e analisa-se de forma estruturada o trabalho relacionado com este tema, desde o processamento de dados em grande escala até ao processamento especializado em grafos, nas suas abordagens e meta-algoritmos específicos, e ainda o processamento aproximado, em geral e específico para grafos.

No capítulo 3 são enunciados os requisitos da solução, após o que é descrita a arquitetura desenvolvida, em alto nível e nas suas partes. São ainda definidas as estruturas de dados utilizadas, e explicados os algoritmos utilizados e o fluxo de dados e de controlo.

De seguida, o capítulo 4 explicita a forma como a solução descrita foi implementada na forma de biblioteca ligada a um sistema de processamento de grafos já existente.

O capítulo 5 começa por descrever o ambiente, os algoritmos e os dados (grafos) utilizados para avaliar a solução implementada. Apresentam-se os testes realizados, bem como os seus resultados quantitativos e qualitativos, segundo critérios devidamente descritos. Os resultados servem de suporte a uma discussão sobre os méritos e potencialidades da abordagem seguida, bem como das suas insuficiências e pontos a explorar e melhorar.

Por fim, o capítulo 6 apresenta as conclusões a tirar de todo o trabalho desenvolvido, bem como pontos para o desenvolvimento futuro.

Capítulo 2

Trabalho relacionado

Neste capítulo pretende-se fornecer uma visão global e crítica sobre o trabalho relacionado com os objetivos propostos. Esta análise pretende enquadrar a problemática do processamento de grafos em tempo real no seu contexto, apresentando as várias formas de encarar e resolver a questão. Pretende extrair-se desta apresentação uma tentativa de sistematização que permita compreender e valorizar as contribuições valiosas para os nossos objetivos, mas também aferir os motivos por que nenhuma abordagem resolve cabalmente o problema.

Este percurso divide-se em três grandes partes, numa aproximação gradual à temática do processamento aproximado de grafos. Na secção 2.1 apresentam-se as principais abordagens ao tratamento de grandes quantidades de dados, em bloco, iterativas ou em *stream*. É sobre os princípios estabelecidos por este tipo de abordagens que assentam as técnicas especializadas no processamento de grafos exploradas na secção 2.2. Por fim, entramos no campo do processamento aproximado, na secção 2.3, onde, ao mesmo tempo que introduzimos o contexto do processamento aproximado em geral, centramos a nossa atenção mais uma vez nos grafos.

2.1 Processamento em bloco e em tempo real

Nesta secção começaremos por ver o paradigma MapReduce, que dalguma forma constitui hoje a base do processamento de dados em grande escala, passando de seguida para as técnicas empregadas para, ainda dentro deste paradigma, dar suporte a computação iterativa. Daí, segue-se para o processamento de *streams*.

2.1.1 Processamento em bloco: MapReduce

A necessidade processar grandes quantidades de dados de forma rápida, fiável e eficiente deu origem a estratégias novas para abordar os problemas. A paralelização surgiu como o grande paradigma para enfrentar as tarefas que os problemas envolvem.

O modelo/sistema MapReduce [29] foi pioneiro na difusão deste paradigma, e constitui ainda um ponto de referência. Os diferentes sistemas nele inspirados distinguem-se sobretudo pela forma como suportam computação iterativa e com dependências, e como potenciam a reutilização dos resultados intermédios.

O MapReduce é um modelo de programação destinado ao processamento e geração de grandes conjuntos de dados. O nome também designa a implementação deste modelo, criada pela Google. Dado um conjunto de dados sob a forma de pares chave-valor, o utilizador define duas funções: a

primeira, *map*, processa cada par original, gerando um conjunto de pares chave-valor intermediários; a segunda, *reduce*, funde, ou sumariza, todos os valores intermediários associados à mesma chave.

Este modelo pode exprimir uma grande quantidade de tarefas reais de processamento de dados, de forma simplificada e especialmente apta à introdução de processamento em paralelo, por não estabelecer dependências entre os dados. O utilizador pode concentrar-se apenas na lógica específica do problema que pretende solucionar, abstraindo dos pormenores relativos ao paralelismo, distribuição, sincronização, tolerância a falhas e comunicação, que são assegurados pelo sistema.

Este paradigma deu origem a variados projetos, que implementam MapReduce ou disponibilizam funcionalidades/APIs nele baseados.

Implementações

O Hadoop¹, uma das mais conhecidas e utilizadas implementações de MapReduce, é uma *framework open-source* que permite o processamento distribuído de grandes conjuntos de dados em *clusters*. Apresentando um modelo de programação simples, permite estender a computação a milhares de nós, garantindo tolerância a falhas e distribuição de carga, implementadas na camada de aplicação [84].

Do Hadoop fazem parte o Hadoop Distributed File System (HDFS), sistema de ficheiros distribuído onde os dados podem ser acedidos por todos os nós do sistema, o YARN, para agendamento de tarefas e gestão de recursos do *cluster*, e o sistema MapReduce propriamente dito. Atua ainda como uma base comum a um conjunto de projetos significativos no âmbito da computação distribuída.

Outros produtos que disponibilizam um modelo de processamento baseado em MapReduce incluem Infinispan², CouchDB³, Disco⁴, entre muitas outras. A Stream API introduzida na versão 8 do Java [79] é um exemplo da implementação de conceitos, e mesmo nomenclatura, provenientes do MapReduce numa linguagem amplamente utilizada.

Críticas

A abordagem MapReduce foi criticada por alegadamente representar um retrocesso em relação ao modelo de processamento oferecido pelas bases de dados (BDs) relacionais, pela ausência dum modelo de dados definido e por consistir na manipulação de *records* em baixo nível. Foi também apontado que a estratégia que propõe não constitui propriamente novidade, pois utiliza técnicas conhecidas e utilizadas há vários anos [60]. Deve notar-se, porém, que esta comparação é inadequada, pois o MapReduce nunca foi concebido para ser utilizado como uma base de dados.

Uma outra limitação prende-se com o facto de as computações, no modelo MapReduce, para poderem tirar o máximo partido da otimização possibilitada por esta técnica, deverem ser expressas como um fluxo de dados acíclico, ou seja, sem dependências entre os dados e entre diferentes pontos do fluxo de execução. Na verdade, quando existem dependências cíclicas, isto é, quando determinados pontos da execução são repetidos com dados obtidos em fases posteriores, surge uma dificuldade acrescida em exprimir o processo como um conjunto de computações independentes que podem ser executadas de forma paralela e assíncrona.

Em problemas que envolvem computação iterativa sobre o mesmo conjunto de dados, e a análise interativa, *ad-hoc*, de *datasets*, o Hadoop, por exemplo, volta a recarregar todos os dados a partir do HDFS, incorrendo numa latência desnecessária[92]. No caso dos algoritmos em grafos, que são frequentemente de natureza iterativa, esta questão coloca-se de forma premente. Por este motivo, o modelo de processamento MapReduce simples não é o mais indicado para o tratamento de grafos.

¹<http://hadoop.apache.org>

²<http://infinispan.org>

³<http://couchdb.apache.org>

⁴<http://discoproject.org>

2.1.2 Processamento iterativo

Como resposta a algumas das limitações referidas anteriormente, surgiram abordagens que pretendem tirar proveito do modelo de computação expresso pelo MapReduce, mas ao mesmo tempo oferecer mais garantias de consistência e tolerância a falhas, um modelo de programação mais alto-nível, com operadores construídos a partir das operações básicas de forma otimizada, e sobretudo suporte a computação iterativa e repetida sobre o mesmo conjunto de dados. Este último requisito é conseguido através do uso de *caching* agressivo dos resultados intermédios, que são mantidos em memória ao longo da sua utilização. Outra importante funcionalidade consiste num mecanismo de otimização das operações sobre os dados (semelhante à otimização de consultas em BDs), incluindo as funções definidas pelo utilizador (FDUs).

Implementações

O Apache Spark⁵, para responder a estas questões, utiliza uma abstração denominada Resilient Distributed Datasets (RDDs), que são estruturas de dados paralelas, tolerantes a falhas, que permitem manter explicitamente resultados intermédios em memória, a fim de serem reutilizados e manipulados por uma série de operadores. Os RDDs apenas podem ser obtidos por operações determinísticas a partir dos dados ou de outros RDDs. Desta forma, ao invés de se representarem os dados intermédios explicitamente, são armazenadas as operações que permitem obter o RDD a partir dos dados anteriores. Deste modo, mesmo tornando-se um pouco mais complexa a reconstrução dum conjunto de dados, logra evitar-se a replicação direta, que é dispendiosa, sobretudo em termos de espaço, e mais difícil de manter, e ao mesmo tempo assegura-se uma eficaz recuperação de falhas [91].

Os RDDs, juntamente com as operações suportadas sobre eles e uma forma de variáveis partilhadas entre as várias tarefas do programa, permitem que o Spark, com o seu modelo predominantemente em memória, possa correr programas 100 vezes mais rapidamente que o Hadoop, em memória, e 10 vezes mais rapidamente em disco[14].

O Apache Flink⁶ é uma plataforma *open-source* para processamento distribuído de *streams* e dados em bloco. Tem a sua origem no projeto Stratosphere [5], criado na Universidade de Berlim. O seu núcleo é um sistema de processamento de *streams*, sendo o processamento em bloco tratado internamente como o caso especial duma *stream* limitada. Por outras palavras, todos os dados são *streams*, de forma subjacente, e o processamento em bloco é efetuado sobre essas mesmas *streams*. Esta estratégia permite unificar o tratamento de blocos e *streams* na mesma *framework*.

Entre as funcionalidades suportadas, contam-se o processamento de eventos fora de ordem, janelas de *streaming* flexíveis, processamento contínuo e computações *stateful*. Quanto a tolerância a falhas, utiliza *snapshots* distribuídos e afirma garantir processamento *exactly-once*. No que respeita a otimizações, é de notar o suporte a computação iterativa e baseada em *deltas* (útil no caso dos grafos) e ainda uma gestão de memória própria dentro da Java Virtual Machine (JVM) [13].

O Flink afirma apresentar um alto desempenho em termos de taxa de débito, aliado a uma baixa latência, o que é confirmado por alguns *benchmarks* realizados recentemente [38].

2.1.3 Streams

O esquema algorítmico utilizado pelo MapReduce é apropriado para processamento de grandes blocos de dados estáticos, mas não tanto de fluxos de dados contínuos, vulgo *streams* de dados. Isso ocorre

⁵<http://spark.apache.org>

⁶<http://flink.apache.org>

porque, como vimos (Sec. 2.1.1), o modelo MapReduce divide a computação em fases individuais e independentes, não estando otimizado para computação iterativa ou contínua.

Como forma de mitigar esta desvantagem, foram estudados modelos incrementais de MapReduce [45], dos quais um dos pioneiros foi o Google Percolator [61]. O processamento incremental consiste em computar resultados atualizados a partir de resultados obtidos anteriormente e dos novos dados entretanto recebidos, sem computar a função novamente no conjunto de dados inteiro, o que é frequentemente impraticável ou muito dispendioso. Se para certas funções uma versão incremental é trivial, para outras é muito difícil encontrar uma boa versão incremental, mesmo quando existe um algoritmo eficiente para dados estáticos [21].

Note-se, porém, que processamento incremental não é sinónimo de processamento de *streams*. Por um lado, pode existir processamento incremental sem haver propriamente uma *stream* de atualizações, o que acontece quando os dados mais recentes chegam em blocos, com intervalos, e não em contínuo; por outro, uma *stream* não tem de ser processada incrementalmente: pode recolher-se uma porção razoável da *stream* e proceder depois a uma recomputação de raiz. Contudo, quer num caso quer noutro, existe uma considerável latência no processamento, tal como períodos de tempo em que os resultados apresentados pelo sistema não correspondem aos dados mais atuais.

Por este motivo, foram desenvolvidos sistemas que procuram aproximar-se o mais possível da computação em tempo real, fornecendo resultados com a menor latência possível em relação à chegada dos dados.

Um sistema de processamento de *streams* recebe habitualmente dados de fontes muito heterogêneas, que comunicam por diferentes protocolos e podem apresentar uma acessibilidade irregular. Por este motivo, qualquer sistema de processamento de *streams* possui uma “porta de entrada” que traduz toda esta informação da forma mais eficiente possível e a encaminha para o sistema. Outro ponto comum a todos os sistemas deste género é algum tipo de filas de mensagens, fundamentais para compensar as diferenças entre as fontes de informação (clientes) e os sistemas de processamento, a nível de taxas de produção/consumo, servindo ainda de filtro e de encaminhador da informação para o local adequado. O sistema de mensagens em específico pode utilizar mensagens puramente temporárias, ou apoiar-se na escrita prévia das mensagens em armazenamento físico, alternativas que apresentam cada uma as suas vantagens em diferentes cenários [42].

Uma aplicação para processamento duma *stream* de dados consiste num grafo de processamento, o qual tem a forma de grafo orientado acíclico (DAG), e que nalguns sistemas pode ser definido explicitamente pelo utilizador, e noutros implicitamente, através de representações mais alto-nível. O grafo de processamento é depois transformado pelo sistema num grafo de execução, de acordo com o paralelismo, a disponibilidade de recursos e potenciais otimizações, podendo a partir daí consumir os dados da *stream* de forma contínua [42].

Posto isto, um qualquer sistema de processamento de *streams* consiste basicamente numa API destinada a definir o grafo de processamento, que consiste nas fontes de dados, nas operações internas e nas saídas, bem como na implementação do grafo de execução a partir do grafo de processamento definido pelo utilizador [42].

As diferenças entre sistemas/implementações inserem-se essencialmente nas seguintes categorias: forma de definição do grafo de processamento; garantias em termos de entrega das mensagens e de tolerância a falhas. Vejamos como algumas implementações concretas ilustram estas diferenças.

Implementações

O Apache Storm⁷ foi originalmente criado pelo Twitter e é um sistema distribuído de processamento em tempo real de *streams* ilimitadas de dados. Apresenta uma latência muito baixa nos resultados. O Storm é um exemplo de sistema em que o grafo de processamento da *stream* (intitulado *topology*) é definido diretamente. Os vértices podem ser *spouts* (fontes de *streams*) ou *bolts* (onde é realizada toda a computação) [78]. Já sistemas como o Spark Streaming (uma biblioteca para processamento de *streams* sobre o Spark) e o Flink permitem definir *fontes de dados* e *operadores*: o grafo é construído implicitamente.

Outra diferença significativa ilustrada por este conjunto de implementações são as garantias em termos de processamento de mensagens: o Storm garante o processamento de cada elemento *at-least-once* ou *at-most-once* [78], por meio de nós especiais que verificam, através de *checksums*, se o processamento representado pelo DAG posterior foi realizado com sucesso. Por outro lado, o Spark Streaming e o Flink permitem garantir *exactly-once* [90], o Spark através dos RDDs (cf. Sec. 2.1.2) e o Flink por meio de *checkpoints*.

Alguns *benchmarks* realizados [20, 38] comprovam que o Storm apresenta uma latência muito baixa quando comparado com outros sistemas (Spark e Flink), mas que em contrapartida possui taxa de débito (*throughput*) menos vantajosa.

Existe porém uma diferença significativa entre o Spark Streaming e os restantes. A técnica base utilizada são as *discretized streams*, que consistem em dividir a computação da *stream* numa série de *mini-batches*, pequenos blocos, como o nome indica, que são computados de forma determinística e *stateless*, em pequenos intervalos de tempo, tipicamente inferiores a um segundo. Cada um dos blocos em que a *stream* é dividida é transformado num RDD.

Desta forma, são suportadas todas as operações padrão em *streams*, ao mesmo tempo que as questões relativas à consistência são clarificadas pelo facto de se utilizar uma noção discretizada de tempo. O uso de RDDs apresenta ainda a vantagem de permitir a integração com outros tipos de tarefas do Spark, nomeadamente o processamento em blocos tradicional ou consultas interativas [90].

A principal limitação desta abordagem consiste na latência que surge inevitavelmente por se efetuar *batch* dos dados, ainda que em intervalos pequenos [90]. Apesar de em boa parte das aplicações essa latência ser negligenciável, existem outras para as quais tal demora não é desejável.

Já no caso do Flink, como foi já referido, o núcleo é um sistema de processamento de *streams*, sendo o processamento em bloco tratado internamente como o caso especial duma *stream* limitada. Neste sentido, o Flink segue uma abordagem oposta ao Spark, e mais próxima do Storm. A vantagem é permitir evitar as latências que são inevitáveis no caso da estratégia de discretização seguida pelo Spark [13].

2.1.4 Conclusões

Existem três grandes propriedades que se deseja que um sistema analítico para grandes quantidades de dados possua: correção; baixa latência; elevado débito. Ora, as técnicas e sistemas que vimos até agora não logram satisfazer simultaneamente todas estas propriedades. Sistemas de processamento em bloco, como o Hadoop, conseguem correção máxima e bom débito, mas com latências elevadas; sistemas de processamento de *streams*, como o Storm, apresentam latências mínimas, mas falham no débito, na presença de grandes quantidades de dados, a não ser que recorram a processamento aproximado, sacrificando a correção.

Da exposição até agora efetuada pode tirar-se uma conclusão imediata: a escolha do algoritmo,

⁷<http://storm.apache.org>

estratégia, sistema, ou implementação a utilizar, está fortemente dependente do problema concreto que se pretende resolver, e das características dos dados, sobretudo em termos temporais. Se para o processamento dum bloco de dados estático, numa só passagem o MapReduce/Hadoop seria uma escolha acertada, para cálculo iterativo envolvendo, por exemplo, matrizes, as soluções otimizadas para computação iterativa permitiriam um uso de recursos muito mais eficiente. Mas um e outro apresentam desvantagens no caso de os dados a processar se apresentarem em fluxo contínuo (*stream*), para o que existem técnicas e implementações dedicadas.

Existem, porém, problemas que não se enquadram em nenhuma destas categorias ou que, melhor, apresentam características de todas elas. A estratégia, nesse caso, passa por abordagens híbridas, que combinam várias das estratégias referidas anteriormente.

Nesse campo, um exemplo de proposta estruturada é a Arquitetura Lambda [52], que consiste num meta-sistema/arquitetura de alto nível, baseada na conceção dum sistema em camadas (*layers*), em que os requisitos em termos de correção, latência e débito são separados entre si, e satisfeitos por diferentes camadas do sistema [52, pp. 14-15].

O *batch layer* [52, p. 16] armazena um grande conjunto de dados em bruto, imutáveis e em constante crescimento, e computa funções arbitrárias sobre o mesmo. Os resultados são recomputados regularmente, à medida que chegam novos dados, e são produzidas *batch views*, conjuntos de resultados pré-calculados, que são utilizados pelo *serving layer* para responder em tempo mínimo a consultas sobre os dados [52, p. 17]. A recomputação a partir dos dados é usualmente demorada, pelo que as consultas, nesse intermédio, apresentarão resultados desatualizados. Para compensar esta falha, é introduzida uma terceira camada, o *speed layer*, que tem como função processar apenas os dados mais recentes que chegam continuamente, de forma incremental, gerando *realtime views* em constante atualização, e possivelmente apenas aproximadas [52, pp. 18-20].

No que diz respeito aos objetivos do nosso trabalho (Sec. 1.4), a nossa atenção concentra-se, sobretudo, porém, no processamento iterativo de dados e de *streams*.

2.2 Processamento de grafos

Veremos, nesta secção, de que forma se procurou dar uma resposta à necessidade de processamento em grande escala de grafos. Daí, expomos o modelo algorítmico utilizado em praticamente todas as soluções atuais, o processamento centrado nos vértices, que definiremos, para de seguida identificar os principais eixos que diferenciam as variadas aplicações deste modelo, com exemplos em sistemas reais, e apontando também algumas das principais críticas e alternativas a este modelo.

As soluções que vimos até agora não são especializadas no processamento de grafos, ainda que possam ser utilizadas para esse fim.

Por um lado, não oferecem formas de representação de grafos para além das que são utilizadas para representar dados genericamente. Para responder a esta necessidade, surgiram BDs especializadas em grafos, como por exemplo o Neo4j⁸ ou o Titan⁹ permitem armazenar grafos através de estruturas especializadas. São indicadas quando é importante uma lógica transacional e com foco nas operações CRUD, e para consultas *ad-hoc*, sobretudo quando estas se centram na própria estrutura do grafo. Contudo, não oferecem normalmente capacidades de processamento em bloco nem distribuído.

Por outro lado, o *MapReduce* não permite representar as dependências computacionais próprias da generalidade dos algoritmos para grafos. As soluções para processamento iterativo e de *streams*, por sua vez, não oferecem suporte direto para uma expressão dos algoritmos a executar em grafos que seja simultaneamente intuitiva e apropriada à paralelização [49]. Ainda quando conseguida, tal abordagem

⁸<http://neo4j.com>

⁹<http://thinkaurelius.github.io/titan>

acaba frequentemente por envolver demasiado movimento de dados, sem aproveitar as oportunidades de otimização decorrentes da estrutura peculiar dos grafos [36].

Estas insuficiências levaram ao desenvolvimento duma abordagem ao processamento de grafos que permite tanto exprimir os algoritmos a aplicar de forma natural, como utilizar estratégias de paralelização do processamento de modo mais facilitado, como veremos agora.

2.2.1 Computação centrada no vértice

A maior parte dos algoritmos clássicos sobre grafos baseia-se numa visão de conjunto sobre o grafo, com estruturas globais e dependências entre vértices e arcos que são mantidas ao longo de todo o algoritmo. Pense-se, por exemplo, no algoritmo de Dijkstra [30], que utiliza uma fila de prioridades, global, para executar o algoritmo, e cuja execução se processa sincronamente, um passo de cada vez. Ora, tais características tornam difícil, se não mesmo impossível, a paralelização e distribuição do algoritmo, pois as interdependências e as necessidades de sincronização tornariam a implementação e execução demasiado complexas, inutilizando eventuais benefícios.

Como forma de responder à necessidade de exprimir algoritmos e computações sobre grafos ao mesmo tempo intuitiva, relativamente fácil de implementar e propícia à paralelização e tolerância a falhas, procurou desenvolver-se um modelo de programação que favorecesse ao mesmo tempo a localidade do algoritmo (computações locais e simultâneas) e a minimização de dependências entre os elementos do grafo.

É neste contexto que surge o modelo de *computação centrada no vértice*, conhecido em inglês por *vertex-centric* ou *vertex-oriented* e ainda pelo lema “*think like a vertex*”, que exprime bem o sentido deste modelo de programação: os programadores são encorajados a deixar uma visão global do grafo para se concentrarem na forma de resolver o problema a partir de computações iterativas e paralelas centradas em cada vértice. As implementações deste modelo partem duma FDU sobre os vértices do grafo, que recebe tipicamente como entrada dados dos vértices e arcos adjacentes, produzindo saídas que são comunicadas através dos arcos de saída. A FDU é executada iterativamente até que uma determinada condição seja observada. Chamaremos a esta FDU *função do vértice*.

Por exemplo, no caso do problema Single source shortest path (SSSP) em grafos pesados, o mesmo que é resolvido pelo algoritmo de Dijkstra, um algoritmo centrado no vértice utiliza uma FDU que, a cada iteração do algoritmo e para cada vértice, seleciona o vizinho com menor distância à origem, define a soma dessa distância com o peso do arco que o liga a esse vizinho como a sua própria distância, se a referida soma for inferior à sua distância atual, e finalmente propaga a sua nova distância a todos os vizinhos (no caso dum grafo orientado, o conjunto de vizinhos a considerar é diferente num e noutro caso [53]).

Este modelo é, dalgum modo, uma adaptação do *MapReduce*, na medida em que se foca na computação local a cada vértice, tomando o sistema a responsabilidade de integrar e unificar tudo da forma adequada. Por outro lado, este modelo centrado no vértice e baseado em mensagens presta-se a exprimir boa parte dos algoritmos de forma intuitiva e acessível.

Este modelo de computação distribuída para grafos está na base de muitas propostas e soluções existentes. Existem todavia diferenças fundamentais na sua implementação concreta, algumas com implicações importantes em termos de lógica e de desempenho. Os principais eixos diferenciadores são [53]: a forma de partilha de informação entre vértices; o funcionamento síncrono ou assíncrono; o modelo de implementação da função de computação para cada vértice; o modo de particionamento/distribuição do grafo pelos vários nós do sistema distribuído. Faremos agora luz sobre essas diferenças.

Partilha da informação entre vértices

O modelo centrado no vértice pressupõe que cada vértice tem acesso a dados da sua vizinhança, vértices e arcos, e que, no fim de cada iteração, cada vértice disponibiliza essa mesma informação. Há duas grandes formas de implementar esta partilha: por meio de mensagens ou por meio de memória partilhada.

No primeiro caso, a função do vértice recebe como entrada as mensagens a ele enviadas na iteração anterior. Utiliza-as na computação do resultado e, por fim, envia mensagens a outros vértices, tipicamente os vizinhos. No segundo caso, cada vértice tem acesso direto aos dados dos seus vizinhos. Neste caso, não é necessário recolher nem enviar mensagens, mas surgem problemas acrescidos devido à necessidade de sincronização de escritas e acessos à mesma zona de memória.

Um exemplo da primeira abordagem é o Pregel [50], um sistema destinado ao processamento distribuído de grafos em larga escala, criado e implementado pela Google. O modelo de computação do Pregel foi implementado de forma *open-source* através do projeto Apache Giraph¹⁰, um dos sistemas de processamento de grafos com maior popularidade.

O processo de computação de grafos utilizado pelo Pregel desenvolve-se numa sequência de passos, denominados *supersteps*. Em cada um desses passos, é executada, para cada vértice v , uma função (*vertex-program*), definida pelo utilizador. Essa função pode modificar o estado de v e dos arcos nele originados. Para além disso, em cada passo s , a função tem acesso às mensagens enviadas a V no passo $s - 1$, e tem a possibilidade de enviar mensagens destinadas a outros vértices (normalmente os vértices adjacentes, mas não necessariamente), que serão processadas no passo $s + 1$. O Pregel, tal como outras implementações, não limita o envio de mensagens aos vértices adjacentes; é possível enviar uma mensagem a qualquer vértice cujo ID seja conhecido. Outros sistemas são mais limitados neste aspeto, apenas permitindo partilhar informação com a vizinhança ou parte dela.

No fim de cada passo, cada vértice pode desativar-se, *votando para parar*. Um vértice inativo não é computado nos passos seguintes, a não ser que receba uma mensagem. O algoritmo, no seu conjunto, termina quando todos os vértices votarem para parar e não existirem mensagens pendentes.

Relacionado com o tema do uso de mensagens para partilha de informação estão algumas otimizações que permitem reduzir o tráfego e a complexidade da computação. Uma das principais, que é preconizada no Pregel, é o recurso a funções de agregação de mensagens, ou *combiners*, definidas pelo utilizador, e que, como o nome indica, se destinam a combinar as diferentes mensagens destinadas ao mesmo vértice. Tais funções deverão ser comutativas e associativas, de forma a que a ordem de execução não interfira no resultado final. Existem várias funções de sumarização possíveis, consoante o objetivo pretendido, entre as quais soma, máximo, mínimo, ou simplesmente a união de todos os valores. Existem também os *aggregators*, outro tipo de funções que computam, em cada passo, estado global visível para todos os vértices. Além disso, é ainda possível modificar a topologia do grafo durante o processamento.

Na figura 2.1 estão representados os elementos principais duma implementação do algoritmo PageRank em Pregel. A função *Combiner* efetua a fusão de mensagens. A função *PregelPageRank*, o *vertex-program*, computa o novo valor do *rank* ao fim de cada iteração. A paragem do algoritmo pode ser sinalizada de duas formas: votar para parar após um número determinado de iterações; votar para parar quando não existir variação significativa no *rank*.

Já como exemplo duma estratégia diferente temos o GraphLab [48], que se distingue do Pregel pelo facto de não utilizar mensagens, mas assentar a comunicação num modelo de memória partilhada. Cada vértice tem total acesso aos seus dados, bem como aos dos vértices adjacentes e dos arcos, independentemente da sua orientação. Pode ainda agendar programas a serem executados por vértices

¹⁰<http://giraph.apache.org>

```

Combiner(Msg m1, Msg m2) :
    return Msg(m1.value + m2.value)

PregelPageRank(Msg m) :
    total = m.value
    vertex.rank = 0.15 + 0.85 * total
    foreach(nbr in out_neighbors) :
        SendMsg(nbr, vertex.rank / num_out_nbrs)

```

Figura 2.1: PageRank em Pregel

vizinhos. Desta forma, é possível utilizar um modelo de comunicação assíncrono, em que o momento em que o fluxo de dados se realiza é definido no programa do utilizador, o que nos leva ao eixo diferenciador que se segue. A estratégia de uso de memória partilhada tem ainda importantes implicações no particionamento dos dados, neste caso vértices e arcos, pelos vários nós do sistema distribuído, em termos de replicação e consistência, como veremos também adiante.

Sincronismo e assincronismo

O Pregel é um exemplo claro de sistema síncrono. Em cada *superstep* a computação é paralela, mas no final de cada um existe uma barreira de sincronização, que garante que todos os vértices terminaram a sua computação antes de se avançar para o *superstep* seguinte. O sincronismo, neste caso, garante também que todas as mensagens são emitidas e que são recebidas na iteração seguinte. O modelo síncrono tem também, porém, as suas desvantagens: existe um considerável *overhead* em termos de recursos de tempo necessário para manter a sincronização; o sistema torna-se suscetível ao problema dos *stragglers*, ou seja, nós de processamento que requerem significativamente mais tempo que os restantes, seja por motivos de falha, seja por desproporção da distribuição da computação (algo relativamente fácil de acontecer com grafos livres de escala) [73].

Este modelo, computação síncrona com comunicação baseada em mensagens, é conhecido, aliás, há algumas décadas, nos alvares da computação paralela. Trata-se do modelo conhecido como Bulk Synchronous Parallel (BSP), que utiliza as mensagens como forma de evitar boa parte dos problemas de sincronização, como leituras e escritas concorrentes e consequentes *deadlocks* e *race conditions* [80]. O Pregel foi, na verdade, uma implementação de BSP especializada em algoritmos em grafos.

O facto de como exemplo dum sistema assíncrono termos mais uma vez o GraphLab mostra como a questão do (as)sincronismo está intimamente ligada com o modelo de comunicação utilizado. Em computação assíncrona, não é necessário um vértice aguardar que todos os outros terminem um passo da computação para avançar para o seguinte, o que torna a utilização de mensagens excessivamente complexa, ao invés dum esquema de partilha de memória e leitura direta.

O assincronismo permite uma melhor utilização de recursos, e pode mesmo acelerar a convergência dalguns algoritmos, mas introduz problemas relacionados com a emergência de não-determinismo na computação. Para fazer face a esta dificuldade, o GraphLab, por exemplo, assegura que qualquer execução é serializável, ou seja, que possui uma execução sequencial correspondente. Para esse efeito, utiliza um mecanismo de sincronização que impede vértices adjacentes de correrem simultaneamente o programa [37].

Existem também abordagens mistas ou híbridas, que utilizam uma combinação de estratégias síncronas e assíncronas, ou que contornam os problemas de uma ou outra abordagem com técnicas mais especializadas [53].

Computação no vértice

A forma de concreta de conceber e implementar a computação que é realizada em cada vértice constitui mais uma diferença significativa entre diferentes propostas.

Sistemas como o Pregel não distinguem conceptualmente, pelo menos de forma explícita, a computação do vértice em si e os aspetos ligados à comunicação de dados e à sumarização. Existe uma única função (no Pregel, a função *Compute*) para ser executada de forma iterativa em cada vértice, e nessa função são lidas as mensagens recebidas, atualizado o valor do vértice, e emitidas as mensagens para o passo seguinte.

Este esquema é um pouco inflexível, pelo que surgiram propostas diferentes, com o objetivo de tomar partido duma efetiva separação entre as fases conceptuais da função de vértice neste modelo. É nesse contexto que surge uma proposta de abstração a nível superior dos algoritmos em grafos: o modelo Gather-Apply-Scatter (GAS), assim chamado a partir das iniciais das três fases que o compõem [37].

Gather É recolhida, para cada vértice, informação dos vértices e arcos a ele adjacentes. Essa informação é sumarizada através duma FDU.

Apply Utiliza-se o resultado da fase anterior para atualizar o valor associado ao vértice que está a ser processado.

Scatter Usa-se o novo valor associado ao vértice para atualizar a informação associada aos arcos que lhe são adjacentes.

Este modelo abstrato foi introduzido no âmbito dum outro sistema de processamento distribuído de grafos, o PowerGraph [37], que se propõe reunir as vantagens do Pregel e do GraphLab. A decomposição GAS permite pensar a computação das mensagens duma forma *pull-based*, ou seja, é o sistema que se encarrega, através da invocação da fase *Scatter*, de obter as mensagens destinadas aos vértices adjacentes a determinado vértice, ao invés de ser o programa do utilizador a enviar as mensagens diretamente. Deste modo, é possível gerir a distribuição de trabalho e o fluxo de mensagens de forma mais eficiente e oportuna, permitindo uma série de otimizações.

Por outro lado, o PowerGraph, mantendo uma lógica de programação orientada aos vértices, procura eliminar a dependência em relação ao grau dos vértices através da distribuição da execução do programa *pelos arcos*, divididos pelos vários nós do sistema. A fase *Gather* é invocada paralelamente em cada arco e os resultados intermédios são reunidos e fundidos, passando o resultado final à fase seguinte. Do mesmo modo, a função *Scatter* é também executada em paralelo.

A figura 2.2 apresenta o algoritmo PageRank, segundo o modelo GAS, numa lógica de iteração por arcos. As funções *gather* e *scatter* são invocadas para cada arco (u, v) . A função *apply* é aplicada a cada vértice, recebendo o parâmetro *acc*, resultante da sumarização dos valores obtidos na fase anterior, através da função *sum*. No caso do PageRank, para cada vértice, a função *gather* apenas tem em conta os arcos de entrada, e a fase *scatter* apenas os de saída. O valor da variação do *rank* entre iterações, representado por *u.delta*, é utilizado para ativar um vértice vizinho para a iteração seguinte.

```
gather(u, (u,v), v):                                     sum(a, b): return a + b
    return v.rank / #outNbrs(v)

apply(u, acc):                                           scatter(u, (u,v), v):
    rnew = 0.15 + 0.85 * acc                             if(|u.delta| > eps) Activate(v)
    u.delta = (rnew - u.rank) / #outNbrs(u)              return delta
    u.rank = rnew
```

Figura 2.2: PageRank no modelo GAS

Um outro exemplo de sistema que segue a mesma estratégia de distribuição da computação pelos arcos é o Chaos¹¹ que resulta do desenvolvimento dum outro sistema, o X-Stream [51, 65].

¹¹<http://labos.epfl.ch/hpgp#chaos>

A estratégia de iteração sobre os arcos representa uma abordagem distinta da iteração sobre vértices, vista até agora, mas deve contudo ser englobada na mesma categoria de *think-like-a-vertex*. Na verdade, o que se altera é a entidade sobre a qual a iteração é realizada, com as implicações em termos de distribuição de carga e de particionamento dos dados, contudo a lógica de programação permanece idêntica: o programa incide sobre os vértices de origem e destino de cada arco [53].

Existem ainda outras variantes, conforme o modo como são distinguidas ou fundidas as diferentes fases conceptuais da função de vértice. Assim, temos modelos de duas fases, como o *Signal/Collect* [71] ou o *Scatter/Gather* [65], proposto pelo já referido sistema X-Stream. Os modelos de programação em duas fases são, aliás, comuns em implementações que se baseiam na iteração de arcos [53].

Estes diferentes modelos de programação de algoritmos em grafos são usualmente oferecidos sob a forma de APIs em sistemas de armazenamento e processamento de grafos genéricos. Assim, por exemplo, a biblioteca de grafos do Apache Spark, GraphX, disponibiliza um operador denominado *pregel*¹². Por outro lado, o Apache Flink, na sua biblioteca de processamento de grafos, denominada Gelly, oferece três formas diferentes de exprimir computações iterativas em grafos [11]:

Vertex-centric Idêntica à *Pregel API* do GraphX, funciona como o Pregel, através de *supersteps* e de mensagens para qualquer vértice, oferecendo a maior versatilidade em termos de poder expressivo.

Scatter-Gather Modelo centrado no vértice em duas fases semelhante ao *Signal/Collect*, e que utiliza uma função (*scatter*) para produzir mensagens destinadas a qualquer outro vértice, e outra (*gather*) para atualizar o valor do vértice com base nas mensagens recebidas, tudo dentro do mesmo *superstep*.

Gather-Sum-Apply Neste caso a iteração faz-se por arcos, e em cada *superstep* uma função *gather* é invocada em paralelo nos arcos e vizinhos de cada vértice, produzindo resultados parciais, que são agregados num valor único através duma função *sum*, comutativa e associativa. Por fim, a função *apply*, tendo em conta o resultado da operação *sum* e o valor anterior presente no vértice, atualiza o valor do mesmo. Este último modelo permite otimizar a execução dos algoritmos à custa dalgumas perdas no poder expressivo: cada vértice apenas tem acesso à sua vizinhança.

Particionamento e distribuição

Pretendendo-se tomar partido do processamento dum grafo em paralelo, como forma de obter resultados mais rapidamente, coloca-se de imediato a questão de como dividir os elementos do grafo e a sua computação pelos vários componentes do sistema. Qualquer estratégia para este fim tem três grandes objetivos: dividir a computação e o uso de recursos de forma equitativa pelos nós do sistema; minimizar a comunicação entre diferentes máquinas; minimizar a sobrecarga causada pela sincronização. Como é de esperar, estes diferentes requisitos não são fáceis de assegurar simultaneamente.

No que diz respeito às estratégias para particionamento dum grafo, podemos ensaiar várias classificações: baseadas em modelos/algoritmos matemáticos ou em heurísticas mais eficientes; *offline*, efetuadas após a leitura completa do grafo, ou *online*, realizadas ao mesmo tempo que o mesmo é lido; estáticas ou com possibilidade de reparticionamento dinâmico; baseadas nos vértices ou nos arcos [53]. Esta última classificação, pela ligação próxima à questão dos modelos algorítmicos abordados na secção anterior, é agora analisada com mais pormenor.

Edge-cuts A distribuição de carga e paralelização é efetuada através da partição dos vértices do grafo pelos vários nós do sistema. Cada nó recebe um igual número de vértices, que são computados em

¹²<https://spark.apache.org/docs/1.2.0/graphx-programming-guide.html#pregel-api>

paralelo, assegurando-se a sincronização, comunicação e tolerância a falhas. A fim de otimizar o fluxo de dados entre diferentes nós, a partição de vértices deve ser tal que minimize o número de arcos divididos entre diferentes máquinas, no que é conhecido como *edge-cut*.

Existem algumas ferramentas para efetuar a partição do grafo descrita mas, como observado em [37], o seu desempenho não é satisfatório para grafos cuja distribuição de grau segue uma lei de potência. Nesses casos, sistemas como o Pregel como o GraphLab, por exemplo, acabam por utilizar *hashing* para distribuir os vértices de forma aleatória pelos nós. Este método, sendo rápido e fácil de implementar, tem o grave inconveniente de, em termos de valor esperado, cortar a maior parte dos arcos, ou seja, colocar a origem e o destino do arco em nós diferentes. Se a carga em termos de computação fica razoavelmente bem distribuída, o *overhead* em termos de comunicação aproxima-se do pior cenário possível [86], quer em relação à quantidade de mensagens entre nós, quer no que diz respeito à presença de *bottlenecks*, introduzidos pela assimetria da comunicação. O mesmo acontece com o armazenamento dos dados, pois cada nó tem de manter a informação acerca das adjacências e, em muitos casos, cópias locais dos dados associados a vértices e arcos.

Em modelos que tratam todos os vértices de forma simétrica, a distribuição de grau em forma de lei de potência introduz desequilíbrio na distribuição de trabalho, visto que as fases *Gather* e *Scatter* apresentam uma complexidade que é linear no grau do vértice. Por fim, há a notar que este tipo de modelo, embora permita executar o programa para muitos vértices em simultâneo, não prevê a paralelização dentro do próprio programa, o que compromete a escalabilidade do sistema na presença de vértices com grau elevado [37].

Vertex-cuts Já as estratégias baseadas na iteração sobre os arcos induzem outra forma de distribuir o processamento e os dados pelos vários nós do sistema. Agora, ao invés dos vértices, são os arcos que são distribuídos uniformemente pelas diversas máquinas. Disso resulta também que são agora os vértices que podem ser divididos por diferentes nós. Assim, alterações aos dados dum vértice têm de ser comunicadas a todos os nós que possuem uma cópia do mesmo.

A título de exemplo, a gestão das várias réplicas do mesmo vértice passa por, no caso concreto do PowerGraph, selecionar aleatoriamente uma das réplicas como *master*, tornando-se as restantes *mirrors* que mantêm uma cópia de leitura dos dados. Assim, alterações aos dados dum vértice são efetuadas apenas no *master*, que as propaga imediatamente para os *mirrors*.

O objetivo passa a ser portanto minimizar o número de vértices divididos entre máquinas diferentes. Conforme exposto em [37], este objetivo pode ser formalizado pela noção de *balanced p-way vertex-cut*: cada arco é atribuído a um nó do sistema de modo a minimizar o número de vértices divididos, mas sem que o desequilíbrio entre o número de arcos atribuídos a diferentes nós ultrapasse um fator constante predefinido.

Existem diferentes técnicas para efetuar um *vertex-cut* nas condições descritas. Em [37] apresenta-se o valor esperado para a replicação de vértices quando os arcos são atribuídos aos nós de forma aleatória. Mostra-se ainda que, em grafos que seguem uma lei de potência, se obtém um desempenho superior ao obtido com *edge-cuts*.

No GraphX, o particionamento do grafo pelos vários nós do sistema distribuído é realizada também através de *vertex-cuts*, na esteira da estratégia seguida pelo PowerGraph, a fim de minimizar o *overhead* de comunicação. É possível porém controlar a forma como os arcos são distribuídos pelos nós, através de funções de particionamento expressamente definidas, ou utilizar a estratégia por omissão, que tem em conta a forma como o próprio *input* se encontra particionado [86, 36].

2.2.2 *Streams* em grafos

A questão da mutabilidade dos grafos motiva a utilização de *streams* no contexto do seu processamento.

Uma boa forma de representar a evolução do grafo é através duma *stream* de atualizações que vão alterando propriedades do mesmo. As alterações tanto podem manter intacta a estrutura da rede, modificando apenas os dados associados a vértices e arcos, como proceder a alterações topológicas ao grafo, introduzindo e removendo vértices e arcos. Estes dois tipos de atualizações têm implicações diferentes em termos de processamento distribuído, representação, equilíbrio de carga e otimização.

Ao invés de se construir o grafo estaticamente para depois o analisar, recebe-se uma *stream* de atualizações incrementais. Redes sociais bem conhecidas, tais como Facebook, Twitter, *blogs*, apresentam estas características, constituindo bons casos de uso de processamento de *streams* de grafos em tempo real [31].

Existe um outro tipo de *stream* de grafos na literatura sobre esta temática. Quando um grafo é demasiado grande para ser mantido em memória, existem algoritmos e sistemas desenhados para receber apenas uma *stream* dos arcos do grafo. Os arcos podem ser recebidos numa ordem particular (por exemplo, recebendo todos os arcos conectados a determinado vértice sequencialmente), ou a ordem pode ser arbitrária ou expressamente aleatória. Estas diferentes possibilidades de ordem têm claramente diferentes implicações para o desenho dos algoritmos.

Neste caso, estamos perante o processamento de grafos linearizados, sem representação completa em memória, e sem processamento de atualizações. Como se pode depreender facilmente, este tipo de *streams* em grafos não está relacionado com o facto de certos gráficos serem mutáveis; este modelo de computação de grafos pode aplicar-se também a grafos estáticos.

A abordagem clássica a este tipo de *streams* de grafos requer que o processamento seja realizado com apenas uma passagem da *stream*: assim que um elemento é visto, não volta a ser visto de novo. Para além disso, não pode manter-se o grafo inteiro em memória, caso contrário não haveria distinção entre esta e a abordagem normal. Toda a computação, seja exata ou aproximada, tem de ser realizada nestas condições e, como foi referido, existem bastantes algoritmos especializados neste tipo de cenário.

Também existem variantes a este modelo de computação de grafos. Algumas delas permitem múltiplas passagens da *stream*, usualmente um número linear ou polinomial delas (relativamente ao tamanho do grafo). Outros algoritmos baseiam-se na possibilidade de “anotar” a *stream* entre passagens sucessivas. Sem representar o grafo na sua totalidade, outras técnicas permitem representar uma parte dele, tal como os vértices, ou outras estruturas convenientes [54].

Ao longo do presente trabalho, quando se faz referência a *streams* de grafos, pretende referir-se o primeiro significado: existe um grafo com uma estrutura e propriedades bem definidas, e que evolui no tempo, facto representado por uma *stream* de atualizações.

2.2.3 Outras abordagens

Uma das críticas que pode fazer-se aos paradigmas e sistemas expostos é que reduzem a expressividade dos algoritmos que podem ser expressos, e consequentemente dos problemas que podem ser resolvidos eficientemente em grafos. Um sintoma significativo é o facto de grande parte da investigação nesta área recorrer aos mesmos tipos de algoritmos para avaliar as soluções.

Pode ainda observar-se que, no modelo centrado nos vértices, não existe controlo explícito sobre a forma como o grafo é particionado pelos vários nós de processamento, não se tomando assim partido de possíveis otimizações algorítmicas caso esse particionamento pudesse ser controlado pelo programador [77].

Existem ainda classes de problemas em grafos que não se enquadram nos paradigmas vistos até agora, centrados em vértices ou em arcos. Exemplo desse tipo de problemas são os de *graph mining*

(detecção de subgrafos frequentes, a contagem de padrões ou a enumeração de cliques, por exemplo). Problemas como estes envolvem a enumeração e a exploração exaustiva de subgrafos, o que torna abordagens como a do Pregel ineficientes pelo facto de envolverem demasiado movimento de mensagens e causarem sobrecarga em vértices de grau elevado. Por este motivo surgiram abordagens especializadas, tais como as que têm como unidade conceptual de computação motivos, ou estruturas repetitivas presentes nos grafos, das quais um exemplo é o Arabesque [76]. Relacionadas com estas, outras propostas sublinham que, muitas vezes, é necessária informação não só de cada vértice isoladamente ou dos seus vizinhos imediatos, mas de toda a vizinhança, e por esse motivo propõem meta-algoritmos que têm como alvo subgrafos, ou vizinhanças, quer como modelo de programação único ou a par do modelo centrado no vértice. Os exemplos incluem implementações como Giraph++ [77], GoFFish [69], NScale [62] e Blogel (centrado em *blocos*, que mais não são que subgrafos) [88].

Outra observação pertinente é a que dá conta de que frequentemente se utilizam algoritmos que percorrem arcos do grafo, acedendo aos vértices ao longo de caminhos. Esta observação deu origem a estratégias *path-centric*, como no caso do PathGraph [89]. Outros algoritmos recorrem frequentemente a subconjuntos de vértices, o que motivou uma abordagem especializada no Ligra [68].

2.3 Processamento aproximado

A otimização do uso de recursos de processamento, em bases de dados, por exemplo, inseria-se normalmente num contexto em que os dados são relativamente estáveis e o elemento instável consistia em diferentes análises e consultas a essa mesma informação. Nestes casos, havendo conhecimento completo dos dados envolvidos, os sistemas podem formular *query plans* a fim de otimizar a forma como as respostas são obtidas.

Já no caso do processamento de *streams*, estamos perante a situação inversa: muitas vezes as consultas são sempre as mesmas, processadas de forma permanente, e o que varia é o afluxo de novos dados, frequentemente imprevisível. Facilmente surgem picos de afluência de informação na *stream*, que podem tornar o sistema instável e mesmo incapaz de efetuar o processamento requerido no tempo desejável [87]. Mesmo que não se utilizem *streams*, o facto de o objeto alvo das análises e consultas estar em constante mutação invalida estratégias de otimização *a priori*, que sendo válidas para um determinado estado do objeto, não o são necessariamente para outro.

A resposta mais imediata a este problema, reservar mais recursos computacionais, sejam eles físicos ou virtuais, não é escalável, pois em última análise exigiria uma capacidade de expansão infinita, além de que leva a uma utilização sub-ótima dos recursos, bem como a custos indesejáveis.

Assim, assumindo um contexto em que os recursos são fixos, vislumbram-se de imediato duas formas de responder a esta problemática: otimização do uso dos recursos existentes, por meio duma distribuição inteligente do processamento e/ou alteração dinâmica de parâmetros da computação (*tuning*); e o recurso ao processamento aproximado, descartando parte do trabalho excessivo em troca dum erro aceitável.

Face à imprevisibilidade do afluxo de dados, mas também à natureza heterogénea das computações a efetuar, dos dados e dos recursos disponíveis [63], surgiram diferentes abordagens de resposta. A nível de plataformas que integram sistemas e tecnologias heterogéneos, geralmente em ambiente de *cluster*, existem soluções para gerir eficientemente recursos entre diferentes sistemas e *frameworks* analíticas [40].

Ao nível de sistemas individuais, existe um grupo de estratégias que assenta na distribuição, ou redistribuição, da carga pelos nós do sistema e respetivos recursos computacionais associados da forma que mais favoreça o débito e a estabilidade como um todo. Analisando o grafo de tarefas e nós, bem

como o fluxo de dados, podem utilizar-se algoritmos que determinam, em tempo real, a alocação de recursos mais eficiente [87, 6]. O uso de técnicas que permitem ao sistema adaptar-se rapidamente a mudanças no fluxo de dados, e tomar partido dos recursos disponíveis é também conhecido como *computational elasticity* [67]

Outras soluções recorrem à alteração de parâmetros da computação (*tuning*) durante a execução, como forma de promover a estabilidade e o desempenho do sistema. Um dos exemplos paradigmáticos é o tamanho do *batch* em que os dados são processados, em sistemas de processamento em bloco ou de *streams* em *mini-batch* [27].

Estas técnicas são úteis para fazer face a picos súbitos do fluxo de dados e manter a estabilidade e a capacidade de resposta do sistema, mas, além de não serem infalíveis, têm em conta apenas o aspeto quantitativo do fluxo de dados. Não oferecem resposta satisfatória à questão qualitativa, que formulámos nos início deste trabalho, de permitir ter em conta as alterações entretanto introduzidas no objeto da computação a fim de evitar que o custo da computação seja superior ao benefício da resposta. Isto leva-nos ao processamento aproximado, a alternativa que mereceu, no presente trabalho, o nosso foco de atenção. Na medida em que permite ter em conta não apenas os aspetos quantitativos, mas também os qualitativos, dos dados, o processamento aproximado constituiu a nossa linha de investigação, pelo que é agora examinado mais de perto.

2.3.1 Conceito

O processamento aproximado consiste em admitir uma determinada quantidade de erro no resultado que se obtém do processamento da informação em troca de vantagens significativas, tais como a rapidez na resposta ou poupança/otimização no uso de recursos utilizados para a obter. Existem inúmeras aplicações reais em que a completa exatidão da resposta não é tão importante como a rapidez da obtenção da mesma, tais como os exemplos que foram apresentados a título casos de uso do presente trabalho, na sec. 1.1.1.

O processamento aproximado surgiu no contexto de sistemas de apoio à decisão [47] e inclui diferentes técnicas, conforme o problema concreto a resolver e as estratégias empregadas na sua resolução. Podemos considerar, *grosso modo*, duas grandes formas de utilização de processamento aproximado. A primeira delas ocorre quando os resultados são sempre aproximados, isto é, quando o sistema fornece sempre aproximações aos resultados reais, com erro controlado, o que pode acontecer quando a obtenção da resposta exata é muito dispendiosa, demorada, ou mesmo impossível. Uma segunda forma de processamento aproximado insere-se em sistemas que garantem *correção futura* (*eventual accuracy*) dos resultados: o sistema fornece regularmente resultados exatos, mas nos intervalos da recomputação da resposta exata apresenta apenas resultados com uma certa aproximação (à semelhança do que foi apresentado na sec. 2.1.4 a propósito da arquitetura Lambda).

No que diz respeito ao processamento aproximado em si, podemos distinguir dois grupos de técnicas/algoritmos: a aproximação por meio de observação da informação a ser processada, e possivelmente seleção/filtragem de dados; a aproximação algorítmica propriamente dita, isto é, o uso de algoritmos que produzem respostas aproximadas. No primeiro grupo abordaremos a amostragem, o *load shedding*, o *task dropping* e técnicas baseadas em inteligência artificial; de seguida veremos alguns conceitos sobre algoritmos de aproximação em grafos.

2.3.2 Técnicas

Amostragem

A amostragem é possivelmente uma das técnicas mais utilizadas para aproximação, nos mais variados domínios. Foi proposta já há décadas como meio de otimizar o tempo de resposta de consultas a bases de dados, ou como forma de previsão de resultados. No caso dos sistemas distribuídos, a amostragem é uma técnica útil para reduzir, por vezes de forma considerável, os recursos computacionais e o tempo utilizados na obtenção da resposta, à custa dum erro que se considere tolerável.

O erro é usualmente apresentado na forma de margens de erro para um determinado intervalo de confiança. O utilizador tanto pode estabelecer o erro máximo admissível, devendo o sistema selecionar a amostra conveniente para essa finalidade, como, de modo inverso, determinar o tamanho da amostra, sendo o erro estimado apresentado pelo sistema [35].

Um problema a considerar, por conseguinte, neste âmbito, prende-se com o cálculo do erro e a confiança que a esse cálculo também está associada. Não existe um método de estimação do erro ideal, e algumas experiências mostraram que o erro reportado, em cenários reais, peca frequentemente ora por otimismo (devolvendo resultados na verdade mais distantes do resultado correto do que seria expectável), ora por pessimismo (selecionando amostras demasiado grandes e alocando recursos desnecessariamente, para o nível de erro que seria aceitável). Para fazer face a este problema, têm sido também desenvolvidas técnicas de diagnóstico para determinar a confiança que pode ser dada ao erro calculado por determinado método [3].

Em bases de dados, ou sistemas em que os dados são estáveis, podem pré-computar-se amostras de diferentes tamanhos e estratificações diferentes, a selecionar conforme a consulta pretendida e o erro máximo admissível [2]. Quando os dados sofrem constante atualização, como é o caso dum sistema de processamento de *streams*, esta estratégia não pode ser aplicada diretamente, mas podem encontrar-se soluções mistas, com atualização de amostras pré-computadas, por exemplo. O mesmo tipo de solução pode servir para um sistema em arquitetura Lambda.

Amostragem de grafos Existem diferentes formas de efetuar amostragem em sistemas de processamento em larga escala. No nosso caso concreto, o de grafos que recebem atualizações, pode efetuar-se sempre nova amostragem, depois de aplicadas as atualizações ao grafo, ou partir duma amostra inicial, que é atualizada da forma conveniente, à medida que chegam novos dados. A amostragem de grafos constitui em si mesma um tema autónomo, com ampla investigação em aberto.

Existem diferentes cenários em que a amostragem de grafos pode ser utilizada: o grafo completo pode ser conhecido, e nesse caso o que se pretende é utilizar a amostra como um resumo, um representante, ou o grafo completo pode ser desconhecido, caso em que a amostra pode servir também como uma forma de exploração do próprio grafo.

O objetivo da amostragem dum grafo também pode ser variável. Por exemplo, no caso de grafos utilizados em ciências sociais, o interesse é frequentemente obter um conjunto representativo de vértices (os indivíduos da população). Mas noutros casos, o objetivo pode ser manter certas propriedades do grafo original. Entre as propriedades que se deseja manter podem enumerar-se algumas relativamente simples, como a distribuição de grau, mas também propriedades mais complexas, tais como coeficiente de *clustering*, medidas de centralidade, cortes, densidade, etc. Por vezes, garantir a preservação destas propriedades em termos de amostragem não é trivial, e pode requerer técnicas específicas, ou garantias, em termos analíticos, da correção da abordagem utilizada. A amostra obtida, quando se garante preservar a propriedade pretendida no grafo original, tanto pode servir como um estimador para a dita propriedade, como servir de objeto para algoritmos específicos que dependem da propriedade preservada.

Existem variadas técnicas de amostragem para grafos. As mais básicas são as que consistem em obter uma amostra dos vértices através das técnicas usuais de amostragem, mantendo os arcos entre os vértices selecionados. O mesmo pode ser aplicado para efetuar amostragem aos arcos. A amostragem tanto pode ser aleatória como recorrer a ferramentas estatísticas e ao conhecimento do domínio para efetuar amostras estratificadas, baseadas em propriedades associadas aos vértices ou arcos. Uma variante consiste em selecionar vértices juntamente com a sua vizinhança, abordagem relevante, por exemplo, em redes sociais, em que o foco é usualmente um indivíduo juntamente com as suas ligações.

Existem todavia casos, como por exemplo quando o grafo não pode ser obtido na sua totalidade, em que estas técnicas simples não podem ser utilizadas. Nesses casos recorre-se sobretudo a uma família de técnicas de amostragem baseadas em passeios aleatórios (*random walks*). Algumas das formas de obter amostras segundo estes métodos oferecem as mesmas garantias, em termos estatísticos, que a amostragem direta de vértices ou arcos; outras permitem preservar certas propriedades do grafo original. Diferentes técnicas de amostragem permitem preservar diferentes propriedades, ou são mais indicadas para distintos tipos de grafos [41].

Task dropping

Para além da amostragem do *input* de dados, é possível também recorrer a *task dropping*, quando se descartam, ao invés do *input*, algumas das tarefas em que o processamento global é particionado (por exemplo, por nó). Esta técnica pode servir também como estratégia de tolerância a falhas: descartam-se as tarefas que falharam por algum motivo ou as que estão demasiado demoradas (*stragglers*), estimando o erro introduzido [64]. Existem também técnicas mais avançadas, ajustadas à lógica do problema, e que recorrem a FDU's [35]. Técnicas e modelos estatísticos, tais como amostragem multi-nível ou teoria de valores extremos permitem efetuar o cálculo a partir da amostra e extrapolar o resultado para o conjunto dos dados, com um erro máximo previsto que pode ser calculado [35].

Load shedding

No contexto de processamento de *streams*, o *load shedding* consiste em descartar uma parte dos dados a processar, quando, por via dum súbito aumento do fluxo de dados, o sistema não tem capacidade de os processar a todos, ou quando fazê-lo introduz uma latência inaceitável. O problema de *load shedding* pode ser assim formalizado como um problema de otimização em que se procura maximizar a exatidão dos resultados, com a restrição de que a taxa de entrada de novos dados não pode ser superior à taxa a que eles são processados [15]. No caso dos sistemas distribuídos, a questão apresenta complexidade adicional, pois cada um dos nós pode ter os seus próprios requisitos e limitações. Podem conceber-se estratégias centralizadas, em que os diversos nós concordam num plano de *load shedding* comum, ou locais, em que cada nó procede, de forma controlada, à seleção de *input* a desprezar [74].

O *load shedding* aleatório é muito semelhante, na sua técnica, à amostragem do *input*, com a possível diferença de que, quando se fala em amostragem, a amostra ter uma dimensão geralmente bastante inferior ao conjunto de dados original; no caso do *load shedding*, são os dados descartados a minoria. Por outro lado, *load shedding* faz sentido apenas no contexto de *streams*, ao contrário da amostragem, que é uma técnica mais geral e que é aplicada frequentemente a dados estáticos ou pouco variáveis.

Esta técnica também é utilizada em contexto de BDs/*data warehouses*, quando para responder a determinada consulta não se tem em conta uma parte dos dados. Neste contexto, esta estratégia recebe o nome de *data skipping*, sendo porém idêntica, na sua essência [72].

As questões que se impõem são *quando* descartar dados (detetar as condições em que o mesmo deve de ser realizado), *onde* os descartar (em que ponto da *pipeline* de processamento devem ser os dados/resultados descartados), e que *quantidade* e *quais* os dados a desprezar [75]. A deteção do

excesso de carga do sistema envolve algoritmos específicos que não importa discutir aqui. Quanto ao lugar onde os dados devem ser descartados, é fácil de ver que quanto mais cedo na *pipeline* analítica forem eliminados, menos processamento desnecessário terá sido realizado; contudo, frequentemente o fluxo de dados não é linear entre *streams*, e uma eliminação precoce pode implicar erros intoleráveis [75].

Quanto à quantidade e quais os dados a não ter em conta, a abordagem mais simples é meramente quantitativa e consiste em descartar *input* de forma aleatória. Com esta técnica, conhecendo a distribuição dos dados, pode estimar-se e limitar-se o erro resultante através de métodos estatísticos [15].

Existem porém técnicas mais avançadas, que têm em conta a importância dos dados para a computação em causa, e procuram efetuar um *load shedding semântico*, inspirado no conceito de Quality-of-Service (QoS), bastante utilizado no contexto de tráfego em redes. Para esse efeito, há que determinar a importância relativa dos dados e seus valores para os resultados finais, seja através de regras decorrentes da lógica do problema, seja através de métodos estatísticos ou de aprendizagem computacional baseados no histórico de dados processados. Estas técnicas permitem obter predicados que determinam que dados remover do processamento [75].

Outras técnicas

Para além da aproximação por métodos estatísticos, existem ainda outras abordagens, baseadas por exemplo em aprendizagem automática. O objetivo é, a partir do histórico de processamento, inferir a correlação entre os novos dados de entrada e as diferenças no *output* calculado. Quando essa relação não é direta, existem métodos de aprendizagem, tais como redes neurais, *fuzzy logic*, métodos probabilísticos, etc., que permitem obter eficazmente previsões mais acertadas sobre as potenciais alterações que novos dados introduzem nos resultados [34].

Algoritmos de aproximação em grafos

Enquanto as técnicas apresentadas anteriormente consistem em aproximar resultados através de alguma forma de seleção ou filtragem do grafo de *input*, os algoritmos de aproximação para grafos têm uma base bastante diferente. Neste caso, o grafo é, ou pode ser, o grafo na sua totalidade, sem amostragem ou outro tipo de seleção; o algoritmo em si é que produz resultados aproximados.

Existem os algoritmos de aproximação a problemas em grafos da classe de complexidade $NP-hard$. Este tipo de problema é considerado intrinsecamente difícil, com tempo de execução exponencial, no pior caso, e portanto inviável de ser resolvido de forma exata e ao mesmo tempo eficiente e em tempo útil. Contudo, é necessária uma resposta, pelo que foram desenvolvidos algoritmos que devolvem uma resposta apenas próxima da exata, mas em tempo idealmente polinomial, no pior dos casos. No caso dos problemas de otimização, o desempenho destes algoritmos é medido através da razão entre a solução aproximada obtida e a solução ótima. Alguns destes algoritmos garantem que esta razão é sempre inferior a um determinado $\rho(n)$, em que n é o tamanho do *input*. Outros recebem como *input* não só o problema a resolver mas ainda a aproximação desejada, sendo que quanto maior a aproximação, maior o tempo necessário para computar a resposta [26, Cap. 35].

Alguns problemas em grafos inserem-se nesta categoria: *vertex cover* [26, Cap. 35.1]; *minimal k-cut* [81, Cap. 4]; problema do caixeiro viajante [26, Cap. 35.2].

Podem também conceber-se algoritmos de aproximação para problemas em grafos que, não sendo da classe $NP-hard$, são ainda dispendiosos em termos de tempo e recursos.

Capítulo 3

Arquitetura da solução

Neste capítulo começa por se apresentar um exemplo motivador de processamento aproximado, que permite enquadrar e justificar o que se descreve de seguida. Identificada a informação envolvida no processo, são enunciados os requisitos que a solução deve cumprir. Apresenta-se então a proposta final de arquitetura para uma biblioteca de processamento aproximado de grafos, com os seus vários elementos, e são descritos as estruturas de dados que utiliza e os algoritmos em que se fundamenta. A API disponibilizada ao utilizador é apresentada de seguida, concluindo-se com o algoritmo geral de processamento aproximado veiculado pela solução que se propõe.

Neste ponto, a finalidade é fornecer uma visão de conjunto que permitam entender o sistema como um todo e os princípios que presidem à sua construção.

3.1 Processamento aproximado

No presente trabalho apresenta-se uma estratégia que se situa entre a recomputação completa e a devolução da resposta anterior sem computação adicional: o processamento aproximado do grafo. Deste modo, fornece-se uma forma de obter resultados aproximados expectavelmente mais fiáveis do que a mera repetição do resultado anterior, e também, espera-se, com menor uso de recursos computacionais que a computação exata.

Para este processamento aproximado podem utilizar-se as técnicas anteriormente referidas, nomeadamente algoritmos de aproximação (sec. 2.3.2), amostragem/filtragem do grafo (sec. 2.3.2) e outras técnicas híbridas, que combinam elementos de diferentes propostas.

A fim de ilustrar, de forma concreta, o que é possível fazer em termos de processamento aproximado com as ferramentas que a nossa solução disponibiliza, apresenta-se como exemplo motivador o algoritmo PageRank, para depois se descrever a forma como se toma partido da informação sobre as atualizações ao grafo para obter uma aproximação.

3.1.1 PageRank

O PageRank é uma medida de centralidade em grafos, popularizada pela sua utilização pelo Google como forma de ordenar as páginas web pela sua importância [59]. Baseia-se na intuição de que as páginas mais importantes são aquelas para as quais mais páginas importantes diferentes apontam. Uma outra forma de definir esta medida é modelando um utilizador da WWW que, começando numa página aleatória, segue um dos links dessa página com probabilidade β , e com probabilidade $1 - \beta$ escolhe aleatoriamente uma página diferente. O PageRank é assim a probabilidade de o utilizador se encontrar numa determinada página após um número k de iterações [28, Cap. 14.3].

O algoritmo PageRank situa-se no seguimento doutros algoritmos de centralidade, e possui uma definição precisa em termos matriciais [57, Cap. 7.4]. Contudo, uma forma mais intuitiva de o definir, e mais apropriada para implementação, é através dum algoritmo iterativo centrado no vértice.

Todos os vértices iniciam com um valor igual de PageRank. Em cada iteração, cada vértice envia, através de cada um dos seus arcos de saída, o valor do seu PageRank dividido pelo seu grau de saída. De seguida, redefine o valor do seu PageRank como a soma de todos os valores que recebeu através dos seus arcos de entrada, multiplicado por um fator β , $0 \leq \beta \leq 1$, a que soma um valor constante de $1 - \beta$. O processo termina ao fim dum número determinado de iterações, ou quando os valores convergirem dentro dum intervalo máximo pré-definido [59][28, Cap. 14.3].

Este algoritmo é facilmente exprimível em termos de qualquer um dos modelos explanados na sec. 2.2.1 (onde foram apresentados exemplos), o que, aliado à sua popularidade, leva a que muitos dos sistemas de processamento de grafos disponíveis o disponibilizem nas suas bibliotecas.

3.2 Informação/dados

Para entender a forma como o sistema foi concebido, há que estabelecer de forma precisa que dados, que informação, está envolvida em todo o processo, para seguidamente poder entender o seu fluxo. Para isso, identificamos agora quais os dados que são recebidos e produzidos ao longo do processo. Note-se que, neste ponto, apenas interessa identificar a informação que é visível do exterior, e não dados intermédios que não são acessíveis ao utilizador.

Grafo inicial Existe um grafo inicial, que pode até ser um grafo vazio, que representa o estado inicial do domínio por ele representado, e que será objeto, ao longo do processo, de atualizações que interferem na sua estrutura e propriedades, e sobre o qual é necessário extrair informação por meio de computação intensiva.

Atualizações ao grafo Uma *stream* contínua com atualizações ao grafo, que podem consistir em adição ou remoção de vértices e arcos, bem como em alterações a propriedades associadas aos vértices ou arcos, sem modificar topologicamente o grafo. Assume-se que esta *stream* tem um fluxo imprevisível, e que os dados por ela fornecidos não respeitam qualquer ordem ou regularidade pré-definida. Assume-se que por entre as atualizações, e em instantes arbitrários, são também solicitadas consultas ao grafo, que são respondidas recorrendo à referida computação intensiva.

Resultados Informação produzida pelo sistema, com os dados resultantes da computação intensiva sobre o grafo, tendo em conta as atualizações que foi recebendo, e que responde às consultas recebidas através da *stream*.

3.3 Requisitos

De acordo com os objetivos deste trabalho, descrita a informação envolvida em todo o processo, e tendo em conta o que fomos expondo até agora, podemos definir alguns requisitos funcionais que pretendemos que a nossa solução ofereça.

O utilizador deve poder:

1. definir a localização duma instância do Flink, que será utilizada para o processamento;
2. definir um grafo inicial;

3. definir a origem duma *stream* de atualizações;
4. definir uma transformação aos elementos da *stream*, se necessário, antes do seu processamento pelo GraphApprox;
5. selecionar o algoritmo/medida de interesse a computar, entre as oferecidas pelo GraphApprox;
6. definir critérios para decidir se, para cada vértice, as alterações que recebeu são de dimensão ou impacto relevantes;
7. aceder ao estado atual do grafo, às atualizações pendentes, a estatísticas e medidas sobre as atualizações e à configuração corrente do algoritmo;
8. determinar, possivelmente com base nas informações referidas no ponto 7, através duma função definida pelo utilizador (FDU), se, antes de dar resposta a uma consulta ao grafo, as atualizações devem ser aplicadas, ou se são armazenadas para aplicação posterior;
9. decidir, possivelmente com base nas informações referidas no ponto 7, através duma FDU, qual o tipo de processamento a efetuar, de modo a responder a uma consulta: nenhum (repetição da resposta anterior), aproximado ou exato (completo);
10. modificar a configuração do algoritmo antes de efetuar o processamento;
11. determinar a forma do *output* da resposta, bem como a sua dimensão, se aplicável (número de elementos do *rank*, por exemplo);
12. aceder ao resultado da computação, a estatísticas e medidas sobre o mesmo e sobre o desempenho do sistema na sua obtenção, assim como a dados concretos relacionados com a computação efetuada (estruturas de dados, resultados intermédios. . .).

É com base nestes requisitos que apresentamos, de seguida, a arquitetura geral da solução, as estruturas de dados definidas, os algoritmos utilizados e a API disponibilizada, finalizando com o algoritmo geral de processamento aproximado.

3.4 Arquitetura

Como pode depreender-se dos requisitos expostos, e dos objetivos do trabalho, existem quatro elementos principais no cenário em que a presente solução se insere. Para além da biblioteca de processamento aproximado de grafos, a que chamaremos GraphApprox, temos um sistema de processamento de grafos (neste caso o Apache Flink e a sua biblioteca Gelly), a *stream* de atualizações e, por fim, o utilizador do sistema. A figura 3.1 ilustra a forma como estes elementos se relacionam.

Esta figura representa uma visão geral, bastante independente dos pormenores de implementação, que serão descritos no capítulo 4. Descrevemos agora mais pormenorizadamente cada um destes elementos arquiteturais.

3.4.1 Sistema de processamento de grafos

Dado que o nosso objetivo não é criar de raiz um sistema que processe grafos, a solução passa pela possibilidade de utilização dum sistema já existente e acessível para efetuar esse processamento.

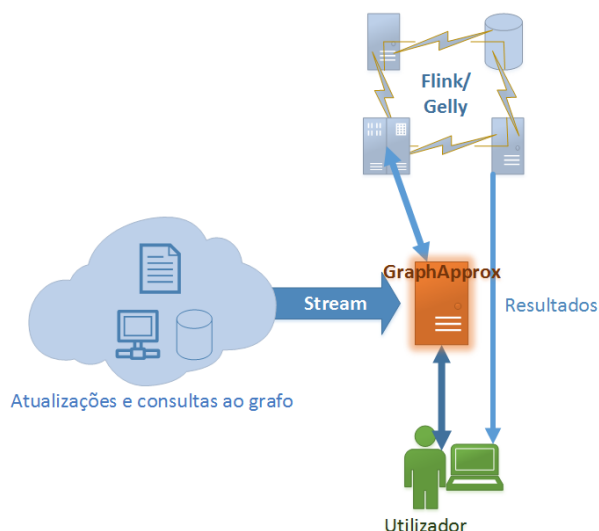


Figura 3.1: Arquitetura geral da solução

Pretende-se ainda que a questão do processamento aproximado, e até mesmo da existência duma *stream* de atualizações, seja transparente para este sistema. Nesse aspeto, qualquer sistema de processamento de grafos, com suporte para a representação e manipulação dos mesmos, e com possibilidade de expressão de algoritmos, seria viável como infraestrutura a utilizar na presente solução.

Uma boa parte dos sistemas referidos na sec. 2.2.1 encontra-se nessas condições. A escolha da solução concreta a utilizar, porém, afeta profundamente o modo como os restantes componentes são implementados, e isso foi tido em conta na seleção.

Na presente solução, o sistema de processamento de grafos selecionado foi o Apache Flink, com a sua biblioteca de grafos, Gelly. Esta escolha cumpre com os requisitos enunciados, permitindo representar grafos, manipulá-los topologicamente e ainda representar e executar algoritmos segundo três abordagens diferentes, conforme referido na sec. 2.2.1, o que representa uma mais-valia em termos de poder expressivo e de otimização.

O Flink/Gelly, onde o efetivo processamento dos grafos é efetuado, estará, num cenário de produção, normalmente num *cluster*, acessível através da rede, ideia representada na figura. Contudo, poderá também ser uma instância *standalone* ou até correr localmente.

3.4.2 Stream de atualizações

Esta é uma fonte de dados externa, cuja origem e forma não está determinada à partida. Sabe-se que contém alterações ao grafo, bem como consultas a serem respondidas, num qualquer formato pré-determinado.

A *stream* de atualizações poderá ter diferentes naturezas e fontes: ficheiros, *sockets*, bases de dados. O ponto essencial é o facto de ser consumida pela biblioteca GraphApprox, que interceta as atualizações ao grafo de modo a diferir a sua aplicação e o processamento subsequente para o momento mais oportuno.

Conforme o problema, grafo, *dataset* concreto, a *stream* pode apresentar mais ou menos regularidade. Por exemplo, quando o grafo representa uma realidade que evolui no tempo, as atualizações podem refletir o fluxo temporal; por outro lado, sobretudo em grafos artificiais criados expressamente para servir de teste, a dimensão temporal pode estar ausente. Do mesmo modo, não se pode prever se a *stream* vai ter picos de intensidade, ou se pelo contrário, podem passar intervalos consideráveis sem que sejam recebidos quaisquer dados.

Assume-se também que cada atualização ou consulta é uma unidade de informação individual. Qualquer agregação ou acumulação de atualizações é realizada *a posteriori*.

3.4.3 GraphApprox: biblioteca de processamento aproximado de grafos

Esta biblioteca de processamento aproximado, como referido, constitui o cerne da solução que apresentamos. Foi concebida para funcionar em conjunto com um sistema genérico de processamento de grafos, fornecendo a lógica de suporte a processamento aproximado.

Para esse efeito, recebe um grafo inicial, bem como a *stream* de atualizações ao mesmo grafo. Disponibiliza ainda uma API para determinar, sempre que uma consulta ao grafo é recebida, se é efetuada a computação exata ou aproximada. Note-se que, por computação aproximada, a este nível, podemos incluir várias realidades diferentes, tais como um algoritmo de aproximação, técnicas de filtragem/amostragem do grafo ou a repetição da última resposta dada.

A biblioteca GraphApprox implementa a lógica de processamento aproximado de acordo com o determinado pelo utilizador. Para esse efeito, procede a um registo das atualizações ao grafo, disponibilizando-as às FDU's. Note-se que a separação entre a biblioteca GraphApprox e o *cluster* Flink é conceptual, mas não necessariamente física. O GraphApprox tanto pode estar no mesmo ambiente virtual que o sistema de processamento de grafos, como residir num outro nó e comunicar remotamente com este. A coexistência do GraphApprox e do Flink na mesma máquina traz vantagens em termos de comunicação. No entanto, os dois são independentes, o que significa que é possível utilizar para o efeito um *cluster* Flink separado, correndo o GraphApprox do lado do utilizador ou num nó intermédio (a opção representada na figura). Importa sublinhar que, para o Flink, o GraphApprox é como um qualquer cliente, sendo pois transparente a sua finalidade de proporcionar processamento aproximado de grafos.

3.4.4 Utilizador

Visto que cada grafo apresenta a sua especificidade, assim como cada algoritmo, a única forma de efetuar processamento aproximado de forma realista consiste em disponibilizar ao utilizador a possibilidade de exprimir a sua própria lógica, baseada no conhecimento que possui do domínio. Note-se que o utilizador, neste caso, se trata normalmente dum programador, de alguém que faz uso do sistema/biblioteca/API GraphApprox no seu próprio programa.

O utilizador não só define a localização do Flink/Gelly e da *stream*, como também recebe os resultados em tempo real, à medida que vão sendo produzidos em resposta às consultas recebidas na *stream*. O GraphApprox disponibiliza uma API que permite definir o modo como o processamento é efetuado, de acordo com os requisitos anteriormente enunciados.

3.5 Estruturas de dados

Descrevem-se agora as estruturas de dados especializadas criadas no âmbito da biblioteca GraphApprox e que permitem que cumpra duas grandes finalidades: o registo das atualizações e a sua disponibilização, bem como de estatísticas e medidas sobre elas; possibilitar que o utilizador defina a configuração do algoritmo de processamento aproximado utilizado. Na primeira categoria temos as estruturas `UpdateInfo`, `GraphUpdates` e `GraphUpdateStatistics`; na segunda, temos a estrutura `Config`. Previamente, dá-se conta da forma como são representados vértices, arcos e grafos, contexto necessário para se poder introduzir algumas das estruturas que se seguem, bem como as questões algorítmicas subsequentes.

3.5.1 Vértices, arcos e grafos

Na esteira da representação utilizada na plataforma concreta selecionada para a construção desta solução, o Gelly, e por ser, de resto, a representação mais simples possível, os vértices são representados pelo seu identificador (normalmente um número inteiro) e os arcos por um par ordenado de identificadores de vértices (*origem, destino*) (assumindo, sem perda de generalidade, que o grafo é orientado). Um grafo pode assim ser representado por um conjunto de vértices e um conjunto de arcos.

No caso de vértices ou arcos terem valores associados (o que sucede frequentemente na solução que apresentamos, como se verá), é adicionado mais um elemento ao tuplo que os representa. Nesse caso, um vértice é um par ordenado (*id, valor*) e um arco um triplo ordenado (*origem, destino, valor*).

3.5.2 UpdateInfo

Esta estrutura serve para possibilitar a monitorização das alterações de grau a um vértice e é apenas um conjunto de campos, (em linguagem C seria uma `struct`, em Java um Plain Old Java Object (POJO)), todos do tipo número inteiro, que agora se descrevem.

nUpdates O número de atualizações ao vértice desde o início do registo, ou da sua reinicialização;

prevInDegree O grau de entrada do vértice aquando da última computação/(re)início do registo;

currInDegree O grau de entrada do vértice no momento atual;

prevOutDegree O grau de saída do vértice aquando da última computação/(re)início do registo;

currOutDegree O grau de saída do vértice no momento atual.

A estrutura pode ser reiniciada, sempre que, após uma computação sobre o grafo, as alterações que a `UpdateInfo` representava tenham sido integradas no grafo, e as alterações futuras sejam monitorizadas a partir do seu estado mais recente.

A estrutura `UpdateInfo` Serve para ser utilizada numa lista associativa, que a cada vértice (ou ao seu identificador) faz corresponder uma estrutura deste tipo. Esta lista associativa será doravante referida como `infoMap`.

3.5.3 GraphUpdates

Estrutura que serve como contentor para as atualizações pendentes ao grafo, e é composta por quatro conjuntos:

verticesToAdd Vértices a adicionar ao grafo;

verticesToRemove Vértices a remover do grafo;

edgesToAdd Arcos a adicionar ao grafo;

edgesToRemove Arcos a remover do grafo.

O uso desta estrutura permite fornecer cópias independentes das atualizações a diferentes intervenientes do processo, sem estabelecer uma dependência com o objeto que efetua a monitorização das atualizações.

3.5.4 GraphUpdateStatistics

Esta estrutura contém estatísticas acerca das atualizações recebidas desde o (re)início da monitorização, e é formada pelos seguintes campos do tipo inteiro:

addedVertices Número de vértices a adicionar;

removedVertices Número de vértices a remover;

addedEdges Número de arcos a adicionar;

removedEdges Número de arcos a remover;

updatedVertices Número de vértices cujo grau foi alterado;

totalVertices Número total de vértices do grafo, após aplicação das atualizações;

totalEdges Número total de arcos do grafo, após aplicação das atualizações.

O facto de estes campos serem meros contadores permite que a sua atualização seja efetuada *online*, ou seja, à medida que as atualizações são recebidas, o que permite que as estatísticas estejam imediatamente disponíveis, quando necessárias.

3.5.5 Config

A parametrização do algoritmo de processamento aproximado constitui um foco de adaptação por parte do utilizador. A estrutura inclui os seguintes campos:

iterations Número máximo de iterações do algoritmo. Tratando-se de algoritmos iterativos, conforme apresentado na sec. 2.2.1, o número total/máximo de iterações tem marcada influência na correção dos resultados e na quantidade de recursos utilizada.

updatedRatioThreshold Limite mínimo da taxa de alteração do grau dos vértices. Este parâmetro controla que vértices são selecionados para a computação aproximada, por terem sofrido alteração do seu grau acima do limite fornecido. A forma como a taxa de alteração é calculada e os vértices são selecionados é descrita na sec. 3.6.2.

neighborhoodSize Tamanho da vizinhança a considerar: selecionado um conjunto de vértices, pode expandir-se a seleção para incluir também os vértices que se encontram até determinada distância, em termos de número de arcos que os separam. Esta expansão é descrita na sec. 3.6.2.

outputSize Número de elementos a apresentar como *output*. Este parâmetro é aplicável, por exemplo, no caso de *rankings*, quando apenas interessam os elementos que se encontram no topo do mesmo, e como tal mais relevantes.

Parâmetros específicos Alguns algoritmos requerem parâmetros concretos para a sua execução, como é o caso, por exemplo, do PageRank, cujo parâmetro β determina a probabilidade de o passeio aleatório continuar para um dos vértices da vizinhança.

Apesar de estarmos a conceber uma solução o mais possível genérica, é inevitável porém que, chegando a aspetos mais concretos como estes, alguns pormenores reflitam as opções em termos de algoritmos concretos e de estratégias para processamento aproximado utilizadas.

3.6 Algoritmos base

Nesta secção apresentam-se os algoritmos que foram desenvolvidos na presente solução de processamento aproximado de grafos. Começa-se pelos mais simples, dedicados à atualização das estruturas de monitorização de atualizações e à sua reinicialização, passando depois para as definições e algoritmos destinados ao cálculo do PageRank aproximado.

3.6.1 Monitorização de atualizações ao grafo

O algoritmo 1 é utilizado para a adição dum novo arco. Utilizam-se estruturas de dados auxiliares, nomeadamente os conjuntos *edgesToAdd*, *edgesToRemove*, *verticesToAdd* e *verticesToRemove* bem como a lista associativa *infoMap* (sec. 3.5.2), que a cada vértice faz corresponder uma estrutura *UpdateInfo*.

Algoritmo 1 Adicionar arco

```
1: edgesToAdd  $\leftarrow$  edgesToAdd  $\cup$  {edge}
2: edgesToRemove  $\leftarrow$  edgesToRemove  $\setminus$  {edge}
3: if edge.source  $\notin$  infoMap then
4:   verticesToAdd  $\leftarrow$  verticesToAdd  $\cup$  {edge.source}
5:   verticesToRemove  $\leftarrow$  verticesToRemove  $\setminus$  {edge.source}
6:   infoMap[edge.source]  $\leftarrow$  new UpdateInfo
7: end if
8: infoMap[edge.source].updates  $\leftarrow$  infoMap[edge.source].updates + 1
9: infoMap[edge.source].currentOutDegree  $\leftarrow$  infoMap[edge.source].currentOutDegree + 1

10: if edge.target  $\notin$  infoMap then
11:   verticesToAdd  $\leftarrow$  verticesToAdd  $\cup$  {edge.target}
12:   verticesToRemove  $\leftarrow$  verticesToRemove  $\setminus$  {edge.target}
13:   infoMap[edge.target]  $\leftarrow$  new UpdateInfo
14: end if
15: infoMap[edge.target].updates  $\leftarrow$  infoMap[edge.target].updates + 1
16: infoMap[edge.target].currentInDegree  $\leftarrow$  infoMap[edge.target].currentInDegree + 1
```

Este algoritmo consiste em manter uma lista de arcos a adicionar, verificando, para o vértice de origem e para o de destino, se o mesmo já existe no grafo, adicionado-o a uma lista de vértices a adicionar, caso contrário. Para além disso, incrementa um contador de atualizações para cada vértice e atualiza os seus valores de grau de entrada e de saída, da forma adequada.

O algoritmo 2 é aplicado para a eliminação dum arco.

Algoritmo 2 Remover arco

```
1: edgesToRemove  $\leftarrow$  edgesToRemove  $\cup$  {edge}
2: edgesToAdd  $\leftarrow$  edgesToAdd  $\setminus$  {edge}
3: if edge.source  $\in$  infoMap then
4:   infoMap[edge.source].updates  $\leftarrow$  infoMap[edge.source].updates + 1
5:   infoMap[edge.source].currentOutDegree  $\leftarrow$  infoMap[edge.source].currentOutDegree - 1
6: end if

7: if edge.target  $\in$  infoMap then
8:   infoMap[edge.target].updates  $\leftarrow$  infoMap[edge.target].updates + 1
9:   infoMap[edge.target].currentInDegree  $\leftarrow$  infoMap[edge.target].currentInDegree - 1
10: end if
```

Este segundo algoritmo é semelhante ao anterior, agora para a remoção de arcos do grafo, procedendo também ao incremento do contador de atualizações e à atualização dos graus de entrada e de saída.

Este registo de atualizações destina-se a ser utilizado entre computações distintas, realizadas para resposta a uma consulta ao grafo, e como tal é reinicializado no final de cada recomputação. As estruturas destinadas aos vértices e arcos a adicionar e a remover são também reinicializadas após as atualizações serem efetivamente aplicadas, sendo utilizado para essa reinicialização o algoritmo 3.

Algoritmo 3 Reinicializar registo

```

1: procedure RESET(updateInfo)
2:   updateInfo.nUpdates  $\leftarrow$  0
3:   updateInfo.prevInDegree  $\leftarrow$  updateInfo.currInDegree
4:   updateInfo.prevOutDegree  $\leftarrow$  updateInfo.currOutDegree
5: end procedure
    $\triangleright$  verticesToReset é uma lista dos vértices cuja monitorização se pretende reinicializar
6: for all  $v \in$  verticesToReset do RESET(infoMap[v])
7: end for

```

3.6.2 Seleção de vértices e expansão à vizinhança

A seleção dos vértices que se julga virem a ter a mais relevante alteração de *rank* é controlada pelo parâmetro *updatedRatioThreshold* (sec. 3.7.1). O grau é, no caso do PageRank, a única característica que pode influenciar o resultado. Como vimos (sec. 3.6.1), o registo das atualizações recebidas permite determinar o desvio, em termos de grau, em relação à última vez que o PageRank foi computado, e como tal é possível saber quais os vértices que, por apresentarem mais alterações, são mais suscetíveis de ter o seu *rank* alterado. O parâmetro *updatedRatioThreshold* determina o nível de alteração que se julga mínimo para o vértice ser selecionado. Assim, por exemplo, é possível selecionar todos os vértices cuja alteração de grau é superior a 10%, ou todos os vértices que foram alterados.

Dado um *threshold* $t \geq 0$, o conjunto K_0 dos vértices selecionados é dado por:

$$K_0 = \left\{ x : \left| \frac{\text{infoMap}[x].\text{currInDegree}}{\text{infoMap}[x].\text{prevInDegree}} - 1 \right| > \text{updatedRatioThreshold} \right\} \quad (3.1)$$

Apresenta-se o caso em que apenas o grau de entrada é tido em conta (no caso do PageRank é de facto esse que é mais relevante para o *rank* dum vértice), mas a definição, para o caso do grau de saída ou para o grau total é trivial de obter a partir desta.

Contudo, não são geralmente apenas os vértices diretamente alterados que são suscetíveis de apresentar maiores alterações na medida que se pretende computar. Em muitos cenários, incluindo o das medidas de centralidade, caso do PageRank, que nos serve de exemplo e motivação, os vizinhos diretos dos vértices mais alterados acabam também por ter o seu *rank* provavelmente alterado. Desta forma, pode haver interesse em incluir, nos vértices a recomputar, não apenas os vértices acima do *threshold* de atualização referido, mas também a sua vizinhança. O parâmetro *neighborhoodSize* determina a dimensão da vizinhança a considerar.

Tomando o conjunto K_0 de vértices acima do *threshold*, selecionado segundo a definição anterior, e definindo $\text{dist}(u, v)$ como o tamanho, em número de arcos, do caminho mínimo entre u e v num grafo orientado (e com $\text{dist}(u, u) = 0$), o conjunto de vértices K_n a considerar, após expansão à vizinhança, é dado por

$$K_n = \{v : \text{dist}(u, v) \leq \text{neighborhoodSize}, u \in K_0\} \quad (3.2)$$

Note-se que esta definição corresponde apenas a uma expansão no sentido dos arcos de saída. Mais uma vez, para o caso do PageRank, é este o caso que interessa. Para uma expansão que tenha em conta o outro tipo de vizinhança, arcos de entrada, a definição deve ser adaptada da forma adequada.

3.6.3 Grafo sumário

A aproximação ao PageRank que apresentamos na nossa solução baseia-se na sumarização do grafo, constituindo uma adaptação dum procedimento já estudado e aplicado [44]. O critério que preside a essa sumarização é bastante intuitivo, e deriva da própria definição de PageRank e do modo como é calculado em termos de mensagens entre vértices.

Como vimos, os vértices que recebem mais atualizações apresentam mais probabilidade de ter o seu *rank* alterado entre iterações; os seus vizinhos imediatos, mesmo que não tenham recebido tantas atualizações, também terão provavelmente o seu *rank* modificado, mas à medida que nos afastamos desse conjunto, o *rank* tende a permanecer igual [25, 16].

A partir desta ideia, o algoritmo de PageRank aproximado que aplicámos começa por dividir os vértices do grafo em dois grupos: os que se supõe que terão o seu *rank* alterado; os que se supõe que manterão o seu *rank*. Estes últimos são então fundidos num vértice único, que os representa a todos, e o PageRank é então calculado neste novo grafo-sumário, de dimensão inferior ao original. Por fim, enquanto que para os vértices que permaneceram se calcula um novo *rank*, para os vértices que foram fundidos atribui-se o mesmo *rank* da iteração anterior.

Definimos agora o grafo sumário e a forma como é obtido do grafo original.

Dado o grafo original $G = (V, E)$ e o conjunto dos vértices K_n , obtido da forma anteriormente descrita, o grafo sumário define-se como $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ em que $\mathcal{V} = K_n \cup \{B\}$. B é o vértice único (*bigVertex*) que representa todos os vértices não incluídos no conjunto K_n .

Para determinar os arcos de \mathcal{G} , começamos por definir $internalEdges = \{(u, v) \in E : u, v \in K_n\}$, ou seja, os arcos cuja origem e destino são vértices contidos em K_n , e $edgesToInside = \{(w, z) \in E : w \notin K_n, z \in K_n\}$, ou seja, arcos em que apenas o vértice de destino se encontra em K_n .

Desta forma, temos:

$$\mathcal{E} = internalEdges \cup \{(B, z) : (w, z) \in edgesToInside\} \quad (3.3)$$

Isto corresponde a substituir todos os vértices externos a K_n por B e a modificar os arcos da forma adequada.

Note-se que todos os arcos entre vértices exteriores a K_n são descartados. O mesmo acontece com os vértices com origem em K_n e destino no exterior, pela seguinte razão: no caso do PageRank, em cada iteração, o novo *rank* dum vértice é calculado a partir dos *ranks* recebidos nos seus arcos de entrada. Ora, o *rank* do vértice B é irrelevante, pois o mesmo representa todos os vértices cujo *rank* não se espera que se altere significativamente. Por esse motivo, os arcos com destino em B podem ser eliminados.

No entanto, para a correção do algoritmo, é necessário armazenar informação adicional no grafo sumário, nomeadamente nos seus arcos.

Em primeiro lugar, ao eliminarem-se arcos de saída dos vértices em K_n , é necessário preservar o seu grau de saída original, visto que este é utilizado para o cálculo do *rank* a enviar em cada iteração do algoritmo. Deste modo, para $(u, v) \in internalEdges$ define-se:

$$val((u, v)) = \frac{1}{deg_{out}(u)} \quad (3.4)$$

Agora, apesar de substituímos os vértices exteriores a K_n por um único vértice B , o *rank* total recebido em cada iteração do algoritmo pelos vértices em K_n tem de ser calculado como se tal substituição não tivesse ocorrido. A diferença é que, agora, o valor proveniente dos vértices exteriores é constante durante o algoritmo, pelo que pode ser registado no início e utilizado subsequentemente. Para esse efeito, antes da fusão dos vértices exteriores em B começa por se registar, em cada arco com destino

em K_n e origem fora desse conjunto, o *rank* que seria enviado originalmente:

$$\text{val}((w, z)) = \frac{\text{rank}(u)}{\text{deg}_{\text{out}}(u)} \quad (3.5)$$

Com base nisto, após a fusão dos vértices exteriores em \mathcal{B} e da conversão dos arcos, tem-se:

$$\text{val}((\mathcal{B}, z)) = \sum_w \text{val}((w, z)), \quad (w, z) \in \text{edgesToInside} \quad (3.6)$$

Desta forma, preserva-se o *rank* recebido dos vértices exteriores, agregado através da soma, a operação de agregação utilizada no PageRank por cada vértice sobre os valores recebidos através dos arcos de entrada.

3.6.4 PageRank aproximado

Definido o grafo sumário, o algoritmo de PageRank tem de ser adaptado da forma adequada para ter em conta as características do grafo sumário e poder calcular uma aproximação.

O algoritmo 4 exprime o PageRank em termos duma estratégia Scatter-Gather (sec. 2.2.1), através de dois procedimentos, utilizados respetivamente para enviar mensagens aos outros vértices e para recolher as mensagens recebidas e calcular um novo valor para o vértice. Este é, a menos de pormenores de implementação, um dos algoritmos oferecidos pela biblioteca de grafos do Flink, Gelly, para computação de PageRank (classe `PageRank<K>`, da *package* `org.apache.flink.graph.library`)

Algoritmo 4 PageRank

```

1: procedure SENDMESSAGES(vertex)
2:   for all  $e \in \text{vertex.outEdges}$  do SENDMESSAGETo( $e.target$ ,  $\frac{\text{vertex.rank}}{\text{vertex.outDegree}}$ )
3:   end for
4: end procedure

5: procedure UPDATEVERTEX(vertex, messages)
6:    $sum \leftarrow \sum_{m \in \text{messages}} m.value$ 
7:    $\text{vertex.rank} \leftarrow (\beta \times sum) + (1 - \beta)$ 
8: end procedure

```

O algoritmo 5 é utilizado na versão aproximada do PageRank que empregamos, sendo executado no grafo sumário anteriormente descrito. Como pode verificar-se, é uma adaptação do algoritmo anterior, para ter em conta que existe um vértice destacado no grafo, para o qual a computação assume uma forma diferente. Para isso, o algoritmo possui a informação da identidade do vértice \mathcal{B} .

Com base nestas definições, e assumindo que o PageRank aproximado, ilustrado no algoritmo 5, pode ser obtido através da função `SUMMARIZEDGRAPHpagerank`, apresenta-se o algoritmo 6, que integra os passos anteriormente explicitados.

Antes de apresentar o algoritmo geral de processamento aproximado, configurável pelo utilizador, é necessário introduzir a API que o `GraphApprox` disponibiliza.

3.7 A API

Sendo o elemento central do presente trabalho, analisamos de perto a estrutura da biblioteca, a relação com as fontes de informação definidas na sec. 3.2 e a sua integração com os outros componentes do sistema. Esta análise é realizada à medida que descrevemos os vários elementos da API, ou seja, os

Algoritmo 5 PageRank aproximado

```
1: procedure SENDMESSAGES(vertex)
2:   if vertex =  $\mathcal{B}$  then
3:     for all  $e \in \mathcal{B}.\text{outEdges}$  do SENDMESSAGETo(e.target, e.val)
4:   end for
5:   else
6:     for all  $e \in \text{vertex.outEdges}$  do SENDMESSAGETo(e.target, vertex.rank  $\times$  e.val)
7:   end for
8:   end if
9: end procedure

10: procedure UPDATEVERTEX(vertex, messages)
11:   if vertex  $\neq \mathcal{B}$  then
12:      $sum \leftarrow \sum_{m \in \text{messages}} m.\text{value}$ 
13:     vertex.rank  $\leftarrow (\beta \times sum) + (1 - \beta)$ 
14:   end if
15: end procedure
```

Algoritmo 6 Cálculo do PageRank aproximado

```
1: function COMPUTEAPPROXIMATE(originalGraph, previousRanks)
2:   updated  $\leftarrow$  UPDATEDABOVE THRESHOLD(updateTracker, config)
3:   expanded  $\leftarrow$  EXPANDTO NEIGHBORHOOD(originalGraph, updated, config)
4:   summaryGraph  $\leftarrow$  BUILDSUMMARYGRAPH(originalGraph, expanded, previousRanks)
5:   ranks  $\leftarrow$  SUMMARIZEDGRAPHPAGE RANK(summaryGraph, config)
6:   newRanks  $\leftarrow$  ranks  $\cup$  previousRanks  $\setminus$  ranks
7:   RESET(updateTracker, expanded)
8:   return newRanks
9: end function
```

elementos que requerem definição ou parametrização por parte do utilizador, e que permitem que a solução genérica seja particularizada para cada problema a resolver.

As funcionalidades disponibilizadas por esta API dividem-se em dois grupos: os parâmetros de inicialização e as funções *callback* que são invocadas em pontos chave do processamento.

3.7.1 Inicialização

Aquando da inicialização são fornecidos, por parte do utilizador, alguns elementos chave para o processamento, que agora descrevemos.

Ambiente de execução

O utilizador define qual o ambiente de execução a ser utilizado, ou seja, em termos práticos, a localização da instância ou *cluster* do sistema de processamento de grafos (Flink, no nosso caso concreto). Como referido, é transparente para todo o processo o facto de esse ambiente ser remoto ou local, ser distribuído ou ser um nó isolado.

Grafo inicial

O módulo de processamento aproximado é inicializado com o grafo inicial, antes de começarem a receber-se as atualizações através da *stream*. Esse grafo é de imediato processado para se extraírem as informações necessárias à avaliação da magnitude e relevância das atualizações que vão ser recebidas.

Um exemplo relevante deste tipo de informação a ser extraída é o grau dos vértices, quer de entrada quer de saída. Podem ser consideradas outras medidas pertinentes.

Este grafo inicial pode assumir diferentes formas e dimensões, dependendo do domínio que representa. Pode representar a realidade inicial que é subsequentemente alterada ou, como foi já referido, pode até ser um grafo vazio, que é o caso quando o grafo em estudo é totalmente representado pelas atualizações, que o vão construindo.

Stream de atualizações

A *stream* de atualizações é outro parâmetro de inicialização do módulo. As atualizações são recebidas e processadas à medida da sua chegada, sendo realizado um registo das mesmas, conforme as estruturas de dados *UpdateInfo* (sec. 3.5.2 e *GraphUpdates* (sec. 3.5.3).

Conforme as características do sistema de processamento de grafos, nomeadamente da forma como os grafos são representados, e de como é tratada a sua manipulação estrutural, pode aplicar-se cada atualização de imediato, ou efetuar algum tipo de *buffering*, de modo a aplicar um conjunto de atualizações duma só vez. Um momento típico em que esta atualização em bloco pode ser realizada é imediatamente antes duma consulta ao grafo, que tem naturalmente de ser realizada sobre os dados mais recentes. Podem contudo aplicar-se outras estratégias, como por exemplo aplicação periódica, salvaguardando a necessidade, quando aplicável, de as consultas serem realizadas tendo em conta todas as atualizações até então recebidas.

Na solução que apresentamos, as atualizações são armazenadas e aplicadas duma só vez, antes da realização duma consulta, como veremos a propósito da implementação.

Seja qual for a política de atualização seguida, o registo de atualizações ao grafo, integrado na biblioteca de processamento aproximado, garante informação atualizada sobre medidas relevantes acerca do grafo, determinantes para poder decidir qual o tipo de processamento a realizar, se exato, se aproximado, se repetição do último resultado.

Formato de saída

Ao utilizador cabe também definir a forma como os resultados da computação são apresentados e/ou armazenados, ou seja, de que forma o resultado no formato próprio do sistema de processamento de grafos selecionado é materializado num formato legível para o seu utilizador final, seja ele humano ou um outro nó de processamento numa cadeia mais alargada. Os sistemas oferecem suporte a vários formatos de saída, de modo que aqui toma-se partido das possibilidades oferecidas pelas suas próprias APIs.

Deste modo, os resultados podem ser escritos num ficheiro CSV, armazenados numa BD, enviados através dum *socket*, gravados em HDFS, ou até simplesmente descartados, para dar alguns exemplos.

Configuração

A parametrização do algoritmo de processamento aproximado é realizada através da definição dos valores duma estrutura *Config* (sec. 3.5.5), que permite configurar aspetos relativos à aproximação dos resultados e ainda do algoritmo concreto que irá ser utilizado.

3.7.2 Funções *callback* definidas pelo utilizador

O cerne da API consiste num conjunto de cinco FDUs, que implementam, para cada caso, a lógica que se pretende imprimir ao processamento aproximado. Descrevemos agora cada uma delas.

void onStart()

Esta função não recebe quaisquer parâmetros. Destina-se a efetuar ações que devam ser prévias a todo o processamento subsequente, e como tal é invocada apenas uma vez, no início do processo, após definir-se o grafo inicial, mas antes de começarem a receber-se os dados provenientes da *stream*.

Esta função pode ser utilizada, por exemplo, para inicializar recursos, como ficheiros e BDs, ou efetuar outras tarefas de preparação.

bool beforeUpdates(updates, statistics)

Trata-se duma função que é invocada após a receção da consulta, mas antes de as atualizações ao grafo serem aplicadas. Recebe os seguintes parâmetros, que expõem o estado da aplicação, para análise por parte do utilizador.

Atualizações As atualizações ao grafo desde a última computação: vértices e arcos a adicionar e a remover.

Estatísticas Estatísticas acerca do grafo e das atualizações: número de vértices e arcos a adicionar e a remover; número de vértices alterados; número total de vértices e arcos do grafo.

A finalidade desta função é dar oportunidade de avaliar a magnitude das alterações pendentes, de modo a decidir se é oportuno aplicar as alterações ao grafo de imediato. Pode haver critérios para dizer, por exemplo, que não serão aplicadas de imediato as atualizações, visto estas serem em número reduzido, pelo que se aguardarão as atualizações subseqüentes para, se for oportuno então, as aplicar todas duma só vez. Esta estratégia permite poupar, se se entender justificado, o uso de recursos e tempo que sempre estão envolvidos na aplicação das atualizações.

É claro que a não-aplicação de atualizações é uma opção que apresenta maior justificação quando a resposta à consulta recebida passa por repetir o resultado anterior, mas nada impede que se tomem outras opções, como por exemplo efetuar a computação sobre os dados anteriores, mas com diferentes parâmetros.

A função devolve como resultado um valor booleano que determina se as atualizações devem ser aplicadas ao grafo antes de se efetuar a computação: *verdadeiro*, se se pretende que sejam aplicadas; *falso*, caso contrário.

A aplicação das atualizações terá como consequência a reinicialização das estruturas envolvidas no registo de atualizações ao grafo.

Action onQuery(id, query, graph, updates, statistics, updateInfos, config)

A função é invocada sempre que é recebida, na *stream*, uma consulta ao grafo, após lhe serem aplicadas as alterações entretanto pendentes, se assim tiver sido determinado pelo valor devolvido pela função *beforeUpdates()*. Recebe os seguintes parâmetros:

Id Um identificador único da consulta, para identificar univocamente a invocação corrente da função.

Consulta O conteúdo da consulta recebida.

Este parâmetro permite que o processamento seja personalizado não só em termos do estado atual dos dados, mas também da própria consulta em si; na verdade, existem cenários em que, para o mesmo estado dos objetos envolvidos, a decisão de processamento é diferente quando é tido em conta o conteúdo da consulta efetuada.

Grafo O grafo que é objeto da consulta, tal como se encontra no momento, aplicadas já as atualizações, se o valor devolvido pela função *beforeUpdates()* assim o tiver indicado.

A disponibilização do grafo permite que se efetuem análises mais completas do que as estatísticas e dados acerca das alterações no grau dos vértices. Compete ao utilizador, ao fornecer esta FDU, decidir sobre o que é mais oportuno. Em determinados cenários, pode justificar-se efetuar algum tipo de computação, por exemplo, para obter uma medida de relevo sobre o grafo, com o objetivo de tomar uma decisão mais sustentada acerca do tipo de processamento a efetuar. Em todo este processo, compete também ao utilizador que define a função ter presente o equilíbrio entre os recursos necessários para efetuar essa pré-computação e os que serão poupados como consequência desta.

Atualizações As atualizações ao grafo desde a última computação: vértices e arcos a adicionar e a remover. Permite-se assim aceder não apenas a informação acerca das atualizações, mas também ao próprio conteúdo material das mesmas.

Estatísticas Estatísticas acerca do grafo e das atualizações: número de vértices e arcos a adicionar e a remover; número de vértices alterados; número total de vértices e arcos do grafo. Trata-se de valores que vão sendo atualizados à medida que as atualizações ao grafo são recebidas, de modo que se encontram já computadas nesta fase. É certo que a maior parte destas estatísticas podia ser obtida por análise do parâmetro anterior, mas neste caso poderia já estar envolvida computação adicional.

Alterações de grau Uma lista associativa que a cada vértice faz corresponder uma estrutura *UpdateInfo* (definida na sec. 3.6.1), com informação dos graus dos vértices no momento corrente e aquando da última computação.

Este parâmetro permite ter informação mais pormenorizada, e também imediatamente acessível, acerca da efetiva alteração de grau dos vértices do grafo. É possível, com esta estrutura, computar, por exemplo, o número ou a razão dos vértices com grau superior a um determinado valor; ou os que sofreram alteração ao seu grau acima de determinada magnitude. É certo que estas informações poderiam ser obtidas através do grafo e das atualizações; mas mais uma vez isso implicaria processamento adicional, ao passo que esta estrutura permite aceder às mesmas informações de modo mais imediato. Na medida em que se prevê que seja um conjunto de informação potencialmente de acesso intensivo, justifica-se, pelo menos conceptualmente, a sua materialização em memória, através deste parâmetro.

Configuração A configuração do processamento aproximado (sec. 3.7.1). Esta estrutura pode ser analisada, mas, mais importante que isso, alterada. Isto é, com base na análise efetuada ao grafo e às alterações que receba, é possível alterar os parâmetros da configuração da forma adequada, o que fornece um grau adicional de liberdade no controlo do processamento que será realizado subsequentemente.

Note-se que a modificação dos parâmetros da configuração corresponde a uma alteração do estado da aplicação que pode ser obtida como efeito secundário na invocação da função *onQuery()*, para além do valor que a mesma devolve.

Com destes dados, o utilizador possui elementos para tomar uma decisão quanto ao tipo de processamento a efetuar, podendo para isso estabelecer critérios ou socorrer-se de técnicas diversas a fim de obter os melhores resultados.

Esta função devolve uma *Action*, um de três valores, correspondendo às seguintes ações subsequentes:

- Devolver a última resposta calculada, sem efetuar qualquer recomputação;
- Efetuar processamento aproximado, devolvendo por conseguinte um resultado também aproximado;
- Dar uma resposta exata, após uma recomputação completa do resultado.

Uma das potencialidades desta API reside no facto de dar inteira liberdade no que diz respeito ao valor a devolver, e que vai determinar a ação ulterior do sistema. A forma de chegar ao valor a devolver pode ir desde heurísticas simples, que têm conta apenas a dimensão/número das atualizações e a natureza do problema, até soluções complexas, baseadas por exemplo em aprendizagem automática (*machine learning*) ou outras técnicas no domínio da inteligência artificial, como redes neuronais, árvores e regras de decisão, métodos estatísticos, etc.

void onQueryResult(id, query, action, graph, result, statistics, ...)

Esta função é invocada após a resposta à consulta ao grafo ter sido obtida. Recebe os seguintes parâmetros:

Id Identificador único da consulta.

Consulta O conteúdo da consulta recebida.

Ação Trata-se do valor devolvido pela função *onQuery*, referida acima, que determinou o tipo de processamento (repetição, aproximado ou exato) que deu origem aos resultados que são agora reportados.

Grafo O grafo que foi objeto da consulta.

Resultados Os resultados obtidos e fornecidos como resposta.

Este parâmetro permite ter acesso aos resultados materiais da computação, de modo a dar a possibilidade de estes serem tidos em conta em decisões futuras. Pode, por exemplo, efetuar-se algum tipo de correlação com as atualizações que os antecederam, ou com os parâmetros passados na configuração do algoritmo. Os resultados podem ainda ser utilizados para efetuar uma avaliação da qualidade, por um qualquer meio acessível, de forma a validar a estratégia de processamento aproximado seguida.

Estatísticas de execução Estatísticas relativas ao desempenho global do sistema na computação do resultado, tais como dados relativos ao tempo total de execução, espaço físico utilizado, tráfego de rede, ou outros.

Estas estatísticas podem ser utilizadas para uma validação do equilíbrio entre aproximação de resultados e poupança de recursos, com a consequente alteração de parâmetros ou de estratégias, se necessário.

Dados relativos ao algoritmo concreto utilizado Conforme o algoritmo/técnica utilizada para efetuar a aproximação de resultados, pode justificar-se fornecer mais parâmetros que proporcionem uma visão mais completa de todas as questões envolvidas no processamento e na qualidade dos resultados obtidos

Este é um parâmetro genérico que deve ser dum tipo que o algoritmo em concreto possa processar, e relevante para a caracterização do desempenho. No caso do PageRank aproximado, o parâmetro que aqui é fornecido é o grafo sumário sobre o qual o processamento foi efetuado,

caso tenha sido utilizado, ou *NULL*, no caso de se ter calculado o PageRank exato, ou repetida simplesmente a resposta anterior.

A função não devolve nada.

Note-se que qualquer uma destas funções, para além dos parâmetros que recebe, pode ter acesso livre a outras fontes de dados, ao critério do utilizador, sejam elas dados de execuções anteriores, resultados computados em fases anteriores do algoritmo ou dados externos.

void onStop()

Esta função, complementar à função *onStart* destina-se a, se necessário, proceder à finalização de recursos e outras tarefas necessárias no final do processo, assumindo que o mesmo não dura indefinidamente.

3.8 Algoritmo global de processamento aproximado

Podemos agora, a título de conclusão, apresentar o algoritmo 7, que é executado pela biblioteca Graph-Approx a fim de disponibilizar processamento aproximado de grafos, configurável pelo utilizador. Este algoritmo utiliza as estruturas de dados anteriormente descritas, bem como os outros algoritmos que foram definidos e os métodos *callback* definidos pelo utilizador, e insere-se no esquema arquitetural previamente apresentado.

Algoritmo 7 Processamento aproximado

```
1: ONSTART
2: repeat
3:   msg ← TAKEMESSAGE(stream)
4:   if msg is Add then REGISTERADDEDGE(msg)
5:   else if msg is Remove then REGISTERREMOVEEDGE(msg)
6:   else if msg is Query then
7:     update? ← BEFOREUPDATES(graphUpdates, graphUpdateStatistics)
8:     if update? then
9:       APPLYUPDATES(graphUpdates)
10:    end if
11:    response ← ONQUERY(id, msg, graph, updates, statistics, infoMap, config)
12:    if response = Repeat-last-answer then
13:      newRanks ← previousRanks
14:    else if response = Compute-approximate then
15:      newRanks ← COMPUTEAPPROXIMATE(graph, previousRanks)
16:    else if response = Compute-exact then
17:      newRanks ← COMPUTEEXACT(graph)
18:    end if
19:    OUTPUTRESULT(newRanks)
20:    ONQUERYRESULT(id, msg, response, graph, summaryGraph, newRanks, jobStatistics)
21:  end if
22: until stopped
23: ONSTOP
```

A figura 3.2 ilustra o mesmo algoritmo, agora sob a forma de fluxograma.

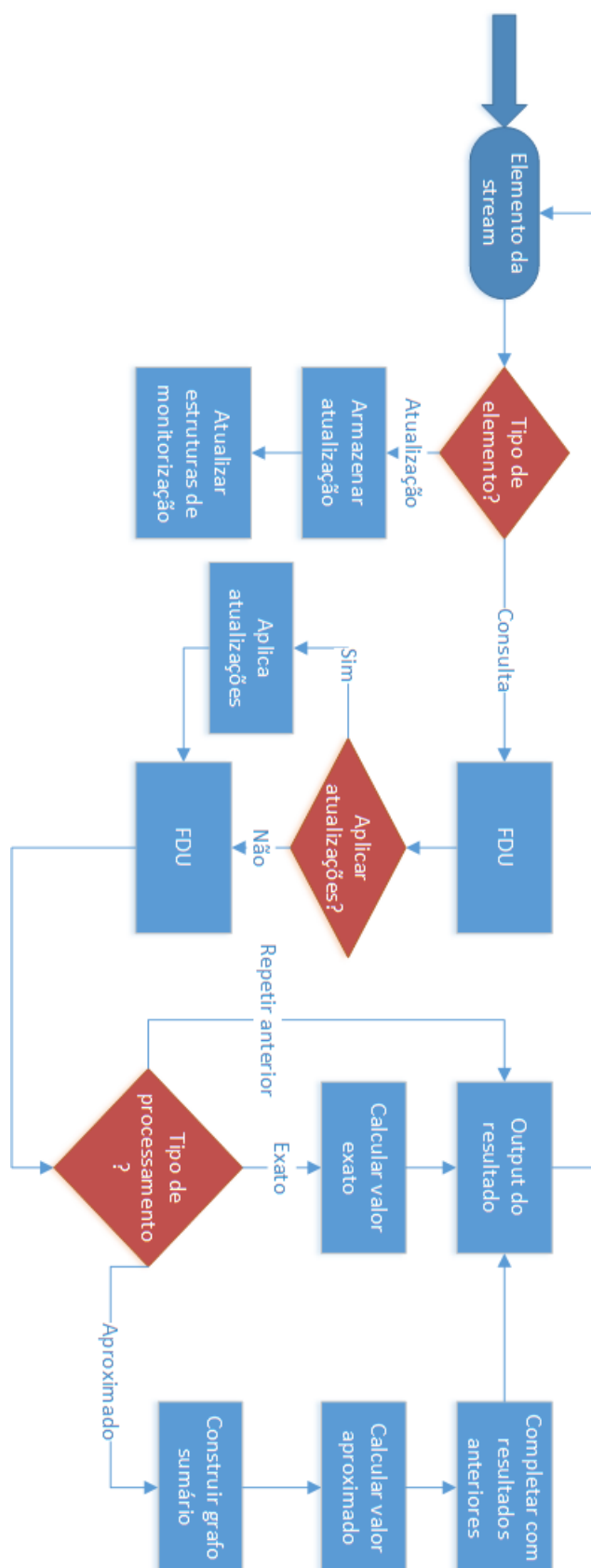


Figura 3.2: Algoritmo global de processamento aproximado

Capítulo 4

Implementação

Neste capítulo apresentam-se pormenores concretos relativos à implementação do GraphApprox, necessários para entender algumas limitações e constrangimentos, bem como para justificar as opções tomadas na conversão duma arquitetura abstrata num protótipo funcional.

Começamos por situar a implementação no seu contexto em termos de tecnologias concretas utilizadas, com as consequências que daí derivam, após o que se apresenta a organização do material produzido num diagrama de classes, descrevendo de seguida a forma como a solução está organizada.

4.1 Apache Flink

O Apache Flink foi selecionado como o sistema de processamento, com suporte a grafos, no qual a nossa solução se insere. Foi utilizada a versão mais recente disponível no repositório público, à data 1.2-SNAPSHOT.

O Flink caracteriza-se por assentar a sua funcionalidade no processamento de *streams*, mas disponibilizar duas grandes APIs, uma dedicada às *streams* e outra aos dados em bloco. Sobre cada uma dessas APIs genéricas, são disponibilizadas bibliotecas especializadas, entre as quais o Gelly, a biblioteca de processamento de grafos, que está baseada na DataSet API. O Flink possui também vários modos de funcionamento, desde o local ao baseado na *nuvem* [7]. A figura 4.1 apresenta os vários componentes do Flink e a forma como se organizam.

A definição dum trabalho (*job*) no Flink é incremental. Começam por se definir os dados iniciais (*DataSource*), que tanto podem ser carregados a partir dum qualquer suporte exterior, como criados expressamente. Sobre esses dados iniciais são então definidas transformações, sob a forma de operadores, que podem servir para filtrar, transformar ou reduzir os dados, obtendo-se sucessivamente diferentes *DataSets*. A cadeia de operadores termina com um operador especial de saída, denominado *DataSink*, que determina o destino dos dados finais [8].

Definido o fluxo de dados e de processamento, pode ser dada ordem para a execução. Nesse momento, a definição do trabalho a realizar é submetida a um otimizador, que gera um DAG denominado *JobGraph*, que é então submetido ao *JobManager* do Flink definido no programa [10]. A figura 4.2, extraída da documentação do Flink, ilustra o modo como este processo se estrutura.

A escolha deste sistema levou a algumas opções concretas, como por exemplo o uso da API Java disponibilizada pelo Flink, a mais desenvolvida até à data e mais completa em termos de funcionalidade. Por esta razão, o trabalho foi desenvolvido utilizando Java 8. O uso desta versão do Java, bem como o facto de a API do Flink estar diretamente ligada à nossa solução, e acessível a partir das bibliotecas criadas, trouxe algumas especificidades ao trabalho, como agora se explicita.

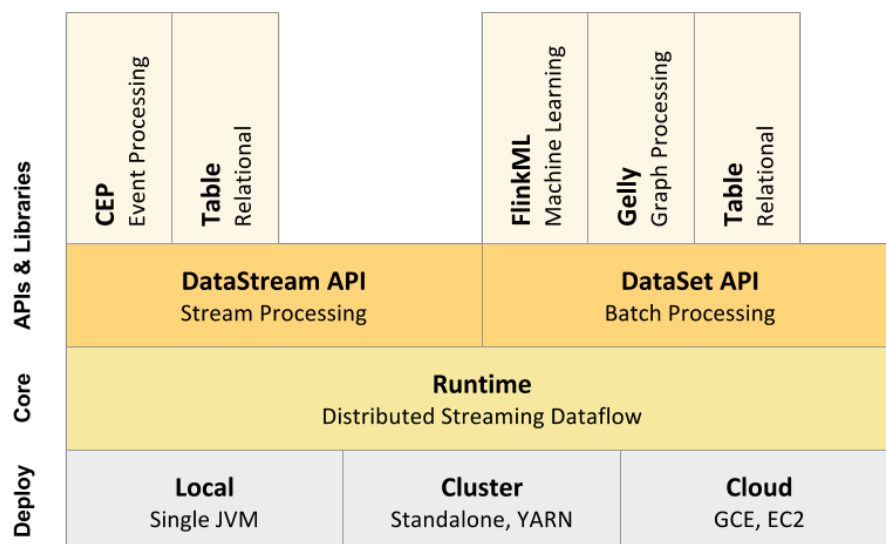


Figura 4.1: *Stack* do Apache Flink

4.1.1 API Flink/Gelly

O facto de a biblioteca GraphApprox estar intimamente ligada a uma instância do Flink motivou o uso das suas funcionalidades e do seu modelo de programação. Assim, algumas operações foram implementadas recorrendo aos *operadores* que essa plataforma disponibiliza (subclasses da classe abstrata `Operator`, da *package* `org.apache.flink.api.common.operators`), e que constituem o modo de exprimir fluxos de trabalho/algoritmos sobre os mais diversos conjuntos de dados. Procurou-se assim tomar partido da existência dum sistema de processamento em bloco para efetuar algum do processamento potencialmente mais oneroso. Tal é o caso, por exemplo, da construção do grafo sumário que serve de suporte ao cálculo do PageRank aproximado.

O uso dos operadores do Flink imprime um estilo funcional à expressão dos algoritmos, tornando o conhecimento dos operadores necessário para entender as operações que estão envolvidas. No entanto, os operadores utilizam nomes comuns no âmbito da programação funcional em geral e dos sistemas da família MapReduce em particular (tais como *map*, *flatMap*, *reduce*, *filter*, *join*, *coGroup*, *groupBy*, *union*).

Procurou-se tomar partido dalgumas otimizações oferecidas pelo Flink, nomeadamente as chamadas *semantic annotations*, que servem para fornecer indicações ao otimizador sobre os dados que, num determinado operador, são transformados e quais os que são simplesmente passados sem alteração. O bom uso desta funcionalidade pode permitir ganhos em termos de desempenho [9].

Contudo, nem sempre se justifica o uso do Flink para efetuar computações e transformações de dados. Na verdade, à utilização desse sistema está sempre associado um atraso inevitável, relacionado com a necessidade de transmitir o código a executar na instância/*cluster* Flink (normalmente sob a forma dum ficheiro JAR), bem como os dados iniciais a tratar. Ora, por vezes é preferível efetuar uma computação local, quando os dados a tratar não são particularmente extensos nem o processamento complexo, e são necessários resultados céleres. É o que acontece, por exemplo, com a monitorização de atualizações ao grafo e a obtenção de estatísticas.

4.1.2 Questões práticas de desempenho

A forma como, no Gelly, são suportadas as alterações ao grafo ditou a política de aplicação das atualizações. Na verdade, por cada atualização são gerados novos *DataSets*, pelo facto de, de acordo com

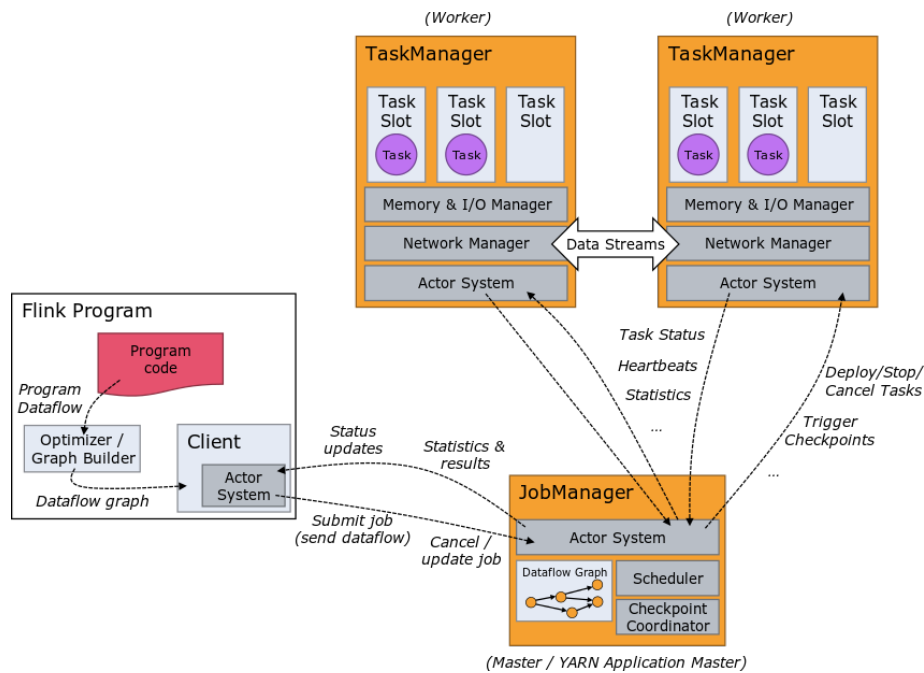


Figura 4.2: Modelo de processo do Apache Flink

os princípios da programação funcional, os mesmos serem imutáveis. Assim, a aplicação das atualizações uma por uma geraria uma quantidade desmesurada de novos *DataSets*, com as consequências em termos de carga para o sistema a isso associadas. Assim, optou-se pelo armazenamento das atualizações até à efetiva necessidade de as aplicar, o que é feito então duma vez só, com um fluxo de dados consideravelmente inferior.

Outra característica do Flink consiste no facto de um *DataSet* poder ser não apenas um conjunto de dados, mas também um conjunto de dados iniciais (*DataSource*) e uma cadeia de operadores (de facto, a aplicação dum operador produz sempre um novo *DataSet*), à semelhança do que acontece também com os RDDs do Apache Spark. Ora, isto significa que, se um determinado *DataSet* assim definido for diretamente reutilizado, voltarão a ser efetuadas todas as operações definidas pela cadeia de operadores, desde a leitura do *DataSource* inicial.

Ora, na solução que desenvolvemos, a reutilização de *DataSets* provenientes de execuções anteriores é constante (o grafo evolui de forma incremental, os *ranks* iniciais são os da execução anterior, por exemplo), pelo que logo no início do desenvolvimento foi necessário delinear uma estratégia. Esta passou pela *materialização* dos *DataSets* a reutilizar, para permitir que, ao serem reutilizados no início duma nova cadeia de operadores, sejam tomados como *DataSources*. Para esse efeito, no final de cada execução, os *DataSets* resultantes foram gravados como ficheiros binários em *cache*, a fim de serem lidos na execução seguinte. Por razões relacionadas com concorrência, a escrita nos ficheiros é rotativa, ou seja, uma execução lê de determinado ficheiro mas escreve num outro diferente, sendo que após algumas execuções o primeiro ficheiro é reaproveitado, e assim sucessivamente.

4.2 Organização da solução

O protótipo que acompanha o presente trabalho está dividido em três componentes distintos.

Em primeiro lugar, a biblioteca *GraphApprox*, o elemento central, onde é implementada a lógica de monitorização das atualizações e de processamento aproximado, bem como a disponibilização da API descrita anteriormente.

Na biblioteca GraphAlgorithms implementaram-se algoritmos para uso por parte das restantes bibliotecas, nomeadamente o PageRank na sua versão exata e na sua versão aproximada.

4.2.1 GraphApprox

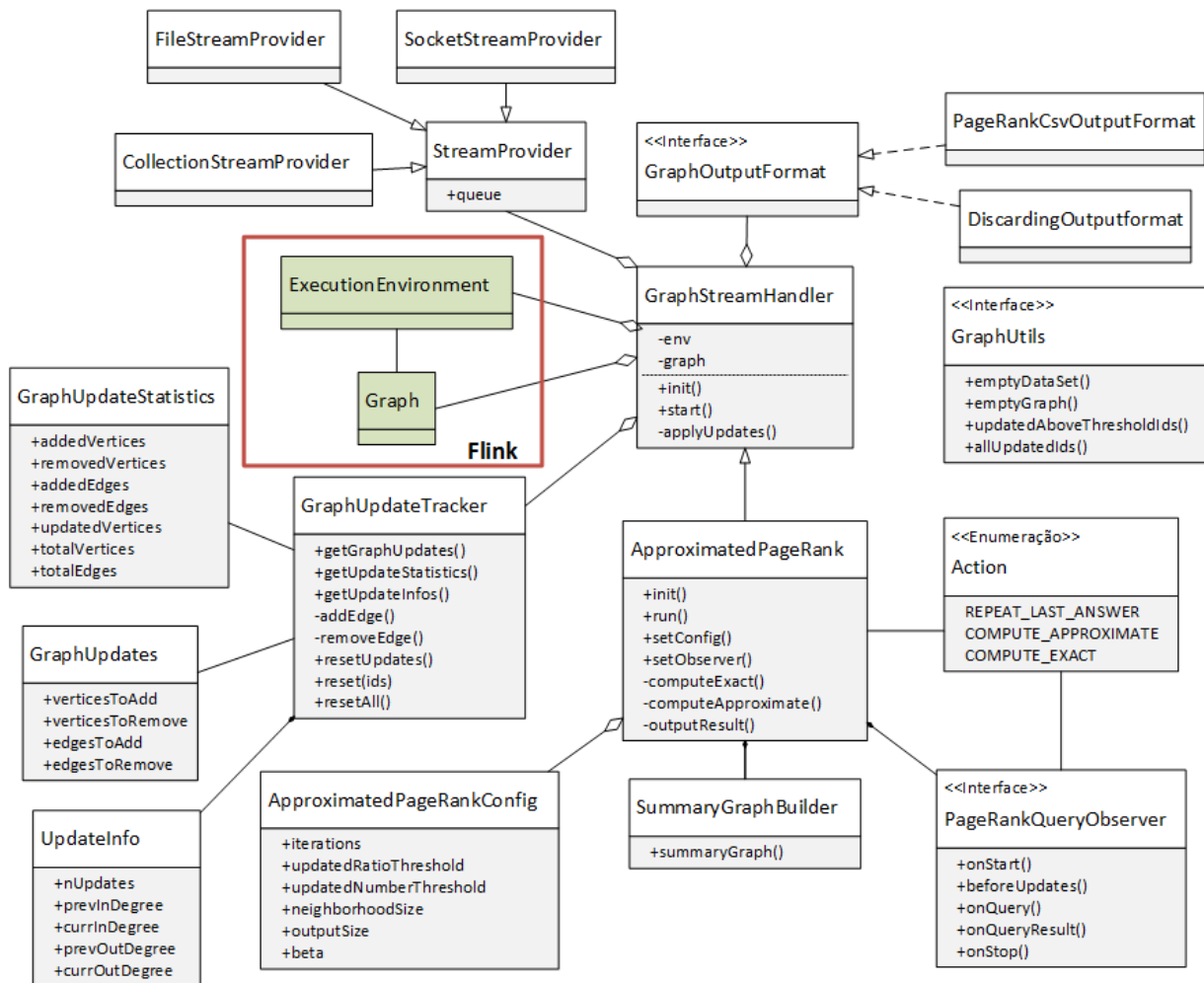


Figura 4.3: Diagrama de classes da biblioteca GraphApprox

A implementação de algoritmos/classes concretas que providenciem o tratamento de alterações a um grafo, bem como consultas ao mesmo, provenientes duma *stream* é abstraída na classe `GraphStreamHandler`.

Esta classe contém referências para um grafo inicial, expresso em termos dos tipos utilizados pelo Flink/Gelly (Graph, composto por um DataSet de vértices e outro de arcos). O ambiente de execução Flink a utilizar vem associado ao grafo. Estas classes, destacadas no diagrama, estabelecem a ligação entre o GraphApprox e a *framework* do Flink. Para poderem utilizar-se as funcionalidades oferecidas pelo Flink e pelo Gelly, foi necessário incluir as respetivas dependências (geridas através do Apache Maven).

Também contém uma referência para um `GraphOutputFormat`, que mais não é do que uma espe-

cialização do `OutputFormat` regular do Flink, a fim de suportar de forma mais ágil a natureza iterativa dos resultados (nomeadamente através da atribuição dum número de versão sequencial e de etiquetas opcionais que permitam distinguir os diferentes resultados produzidos). `GraphOutputFormat` é uma *interface*, permitindo a definição das mais diversas formas de saída, de que se apresentam dois exemplos: o `DiscardingGraphOutputFormat`, como o nome indica, é uma implementação *dummy* que descarta todo o *output* (para servir de formato de saída por omissão), e o `PageRankCsvOutputFormat`, que produz um ficheiro CSV que apresenta os identificadores dos vértices por ordem decrescente do *rank*, apresentando também opcionalmente o próprio *rank* calculado.

A ligação à *stream* de atualizações e consultas é efetuada através dum objeto `StreamProvider`. Este objeto disponibiliza uma `BlockingQueue`, estrutura pertencente à *Java Collections Framework*, e que se destina a providenciar uma fila produtor-consumidor *thread-safe* [22]. As implementações concretas de `StreamProvider` podem então adaptar *streams* de qualquer origem e tipo a objetos Java que são inseridos na fila, e consumidos por um interveniente à partida não especificado (no nosso caso, um `GraphStreamHandler`). São oferecidas algumas implementações de `StreamProvider`: `SocketStreamProvider`, que lê strings dum *socket* e as coloca na fila, como pode verificar-se no excerto de código da figura 4.4; `FileStreamProvider`, que faz o mesmo com linhas dum ficheiro de texto; `CollectionStreamProvider`, que insere na *stream* elementos duma coleção Java.

```
try (Socket s = new Socket(host, port);
    BufferedReader br =
        new BufferedReader(new InputStreamReader(s.getInputStream())) {
    String line;
    while ((line = br.readLine()) != null) {
        queue.put(line);
    }
} catch...
```

Figura 4.4: `SocketStreamProvider`

A monitorização das atualizações ao grafo é realizada pela classe `GraphUpdateTracker`, que mantém todo o registo das atualizações em si e das estatísticas relativas às mesmas. Para isso utiliza outro grupo de classes, que implementam as estruturas de dados definidas na sec. 3.5 (`GraphUpdates`, `GraphUpdateStatistics`, `UpdateInfo`).

Os métodos *callback* a ser fornecidos pelo utilizador, e que constituem uma parte importante da API, estão declarados na interface `PageRankQueryObserver`, para o caso do PageRank aproximado. Outros algoritmos terão as suas próprias declarações dos métodos *callback*, visto que para cada um podem existir parâmetros específicos. No caso do PageRank aproximado, por exemplo, é passado no método *onQueryResult* o grafo sumário construído especificamente para este algoritmo. Também a configuração é específica: a classe `ApproximatePageRankConfig` implementa a estrutura *Config* da sec. 3.5, fornecendo adicionalmente o parâmetro β próprio do PageRank.

4.2.2 GraphAlgorithms

Nesta biblioteca foram implementados os algoritmos que se destinam a ser utilizados pela biblioteca `GraphApprox`, e eventualmente por quaisquer clientes que utilizem o Gelly. O PageRank foi o algoritmo especificamente reimplementado para servir de cenário e caso de uso do `GraphApprox`.

PageRank

O Gelly disponibiliza uma série de algoritmos em grafos, entre os quais duas versões de PageRank, uma segundo o modelo Scatter-Gather e outra no modelo Gather-Sum-Apply. Estas implementações, contudo, sofrem duma limitação que impediu a sua utilização nos grafos reais a testar: assumem que cada vértice possui pelo menos um arco de entrada e um de saída [12]. Ora, na maioria dos grafos reais isso não pode ser garantido. Por esse motivo, o algoritmo original, na versão Gather-Scatter foi adaptado, na classe `SimplePageRank`, a fim de calcular o PageRank correto para qualquer tipo de grafo, podendo ainda ser executado sobre grafos com qualquer tipo de vértices e arcos, devolvendo apenas um `DataSet` que a cada identificador de vértice faz corresponder o *rank* calculado.

A classe `SummarizedGraphPageRank` calcula o PageRank para um grafo sumário, de acordo com o algoritmo 5 (sec. 3.6.4), ou seja, tendo em conta que existe um vértice destacado que deve ser tratado de forma diferenciada.

4.2.3 Tester

Neste módulo são incluídas várias classes que fazem utilização das bibliotecas anteriores com *datasets* reais, que são utilizadas para validar e avaliar a solução, e concretamente a biblioteca `GraphApprox` (sec. 5). Tratam-se de implementações de `PageRankQueryObserver`, ou seja, dos métodos *callback* definidos na API do `GraphApprox`, e programas Java diretamente executáveis (com método `main` estático) que executam todos os passos de inicialização e de utilização da biblioteca.

A figura 4.5 apresenta a parte substancial dum desses programas Java diretamente executáveis, que cria um ambiente Flink remoto, determina um grafo inicial e uma *stream* de atualizações, define o algoritmo a utilizar, configura o processamento aproximado com esse mesmo algoritmo, estabelece a forma de saída dos resultados e, por fim, dá início à computação.

```

ExecutionEnvironment env =
    ExecutionEnvironment.createRemoteEnvironment("146.193.41.145", 6123,
        "flink-graph-approx-0.3.jar", "flink-graph-algorithms-0.3.jar");

try {
    Graph<Long, NullValue, NullValue> graph =
        Graph.fromCsvReader("~/Datasets/Cit-HepPh/Cit-HepPh-init.txt", env)
            .ignoreCommentsEdges("#")
            .fieldDelimiterEdges("\t")
            .keyType(Long.class);

    ApproximatedPageRankConfig config = new ApproximatedPageRankConfig()
        .setBeta(0.85)
        .setIterations(iterations)
        .setUpdatedRatioThreshold(threshold)
        .setNeighborhoodSize(neighborhoodSize)
        .setOutputSize(outputSize);

    String outputDir = String.format("%s/Results/PR/CitHepPh-%02.2f-%d",
        remoteDir, threshold, neighborhoodSize);
    PageRankCsvOutputFormat outputFormat =
        new PageRankCsvOutputFormat(outputDir, System.lineSeparator(),
            ";", false, true);
    outputFormat.setName("approx_PR");

    ApproximatedPageRank approximatedPageRank =
        new ApproximatedPageRank(new SocketStreamProvider("localhost", 1234), graph);
    approximatedPageRank.setConfig(config);
    approximatedPageRank.setOutputFormat(outputFormat);

    String dir = localDir + "/Statistics/PR/CitHepPh";
    approximatedPageRank.setObserver(new ApproximatedPRStatistics(dir, args[6]));

    approximatedPageRank.start();
} catch (Exception e) {
    e.printStackTrace();
}

```

Figura 4.5: Exemplo de utilização do GraphApprox para processamento dum grafo real

Capítulo 5

Avaliação

Neste capítulo procede-se a uma avaliação quantitativa e qualitativa da biblioteca GraphApprox. Após considerações de carácter descritivo relativamente à solução implementada, são descritos os *datasets* com que a mesma foi avaliada. É caracterizado o cenário experimental, após o que são apresentados e discutidos os resultados obtidos, segundo vários eixos de avaliação.

5.1 Aspetos qualitativos

A avaliação da biblioteca GraphApprox constituiu também uma forma de validar a própria implementação, visto que partiu da utilização das API por ela disponibilizada, e anteriormente descrita, para processar dados efetivos e obter resultados.

O uso de diferentes *datasets*, com dimensões distintas, visa demonstrar como o GraphApprox permite utilizar grafos de diferentes naturezas e fontes, tomando partido da própria versatilidade do Gelly, a biblioteca de grafos do Flink. Já o suporte à definição de streams de atualizações constituiu um trabalho independente. Na verdade, procurou-se abstrair os pormenores relativos à natureza da stream, utilizando, conforme descrito, estruturas de dados comuns para representar dados que chegam de forma imprevisível.

O GraphApprox, porém, não se limita a oferecer alguns adaptadores para os tipos de *streams* mais comuns (como ficheiros e *sockets*), mas disponibiliza, através de abstração, uma forma de definir novos adaptadores para diferentes tipos de *streams*, que têm em comum apenas o facto de colocarem objetos numa zona de memória partilhada, para serem consumidos pelo GraphApprox. Todos os pormenores relativos à obtenção dos mesmos objetos, e de eventuais transformações que sejam necessárias, podem ser definidos pelo utilizador.

No que diz respeito à saída de dados, o GraphApprox também oferece abstração suficiente para definir diferentes formas de *output*, com suporte à definição de informações adicionais, como por exemplo marcas temporais, úteis quando se trata de computação iterativa.

A utilização de métodos *callback*, ao mesmo tempo que permite uma resposta já de si dinâmica a diferentes dados e estados da computação, possibilita um nível adicional de dinamismo e flexibilidade: nada impede que, por exemplo, o próprio conjunto de métodos *callback*, que se encontra encapsulado num objeto, seja alterado durante a computação, de forma a responder de forma mais adequada às características dos dados ou das consultas que vão sendo realizadas. O mesmo pode acontecer, como vimos, com a configuração, passível de ser alterada, se não durante a execução do algoritmo, entre diferentes execuções do mesmo.

Visto que não há limitações em relação a fontes externas que podem ser utilizadas ou consultadas, este modelo torna-se apto para a conexão com sistemas de mais alto nível, que tomem partido, por

exemplo, de ferramentas na área da análise estatística ou da aprendizagem automática, no intuito de tomar melhores decisões para um processamento mais eficiente e eficaz.

5.2 *Datasets* de validação

Os *datasets* escolhidos para validar e avaliar a nossa solução foram escolhidos entre grafos do mundo real, disponíveis na literatura especializada nesta temática. A escolha de *datasets* reais vem trazer uma verosimilhança acrescida às conclusões que se possam extrair da avaliação realizada. A alternativa seria o uso de grafos gerados especificamente para o efeito, que seria, sem dúvida, também uma opção viável.

No entanto, apesar de serem reais, os grafos selecionados, como veremos, não refletem necessariamente, só por si, a carga de trabalho, em termos de sucessão temporal, em tempo real, para a qual a nossa solução de processamento aproximado se dirige. A utilização de *datasets* de grande dimensão e elevado débito exige-se para o estudo da implementação real, uma fase posterior, e é mais exigente em termos da infraestrutura e meios necessários para efetuar uma avaliação.

Ainda assim, é necessário que a avaliação reflita dalgum modo a realidade alvo onde a solução se insere, e por esse motivo os conjuntos de dados utilizados foram adaptados de modo a simular, no que diz respeito ao fluxo temporal, a realidade do processamento em tempo real. Assim, o foco está não tanto no interesse da computação, dos resultados obtidos, em si, como resposta a um problema real, mas nas medidas quantitativas e qualitativas que podemos extrair da avaliação.

Descrevemos agora as características de cada um dos *datasets* utilizados, e ainda a forma como cada um deles foi dividido num grafo inicial e em atualizações, bem como o modo como as mesmas atualizações foram tornadas em *streams*.

5.2.1 PolBlogs

Este *dataset* consiste num grafo orientado de blogs acerca de política dos Estados Unidos da América e das hiperligações entre eles, recolhido em 2005 por Lada Adamic e Natalie Glance [1, 56]. Possui 1490 vértices e 18091 arcos.

Este grafo é disponibilizado em formato Graph Modelling Language (GML) [39], pelo que teve de ser transformado numa lista de arcos em formato CSV. A transformação foi realizada através dum *script* em linguagem Python, com recurso a uma biblioteca de grafos nessa linguagem, denominada *networkx*¹, com suporte à leitura de grafos em formato GML.

Foram selecionadas os primeiros 1000 arcos para definir o grafo inicial, e os restantes foram utilizados como atualizações. Entre as atualizações, neste caso adição de novos arcos, e correspondentes vértices, no caso de ainda não existirem, foram introduzidas consultas, um total de 61, distribuídas aleatoriamente.

Esta *stream* é obtida diretamente dum ficheiro (utilizando a classe `FileStreamProvider`), e os elementos são fornecidos de forma contínua, sem intervalos.

5.2.2 Cit-HepPh

Este é o grafo de citações entre artigos presentes no arXiv² na área da fenomenologia de Física de alta energia (*High energy physics phenomenology*). Possui 34546 vértices (artigos) e 421578 arcos (citações) [46, 70]. Cada artigo é representado por um identificador único no formato de número inteiro.

¹<https://networkx.github.io>

²<https://arxiv.org/>

Os dados cobrem o período temporal entre janeiro de 1993 e abril de 2003. Os dados disponíveis permitem associar, a cada vértice, a data de publicação do artigo que lhe corresponde, pelo que este grafo possui uma evolução temporal possível de determinar com exatidão. O grafo é disponibilizado no formato de lista de arcos em formato CSV (separado por espaço).

Foi definido como grafo inicial aquele formado pelos primeiros 40000 arcos do *dataset*. Os restantes foram utilizados como atualizações.

A *stream* foi fornecida sob a forma de *socket*, e portanto consumida pelo GraphApprox com recurso à classe `SocketStreamProvider`. A disponibilização do *socket* e o envio dos elementos foram realizados através dum *script* em Python.

Neste caso, visto que havia informação temporal disponível acerca dos vértices, tomou-se partida da mesma, com o objetivo de imprimir, de forma realista, alguma irregularidade à *stream*. Para esse efeito, começou por se associar, ao identificador de cada artigo, a lista de identificadores dos artigos que cita. De seguida, foi associada a cada data de publicação a lista dos identificadores dos artigos publicados naquela data. Na posse destas listas associativas, percorreu-se cada dia entre a data mínima e a data máxima presente no *dataset* e, no caso de haver artigos publicados nesse dia, foram enviados através da *stream* todos os arcos com origem nos vértices respetivos (o que corresponde a enviar o conjunto de citações de cada artigo).

Entre cada dia percorrido efetua-se uma pausa de 0,01 segundos. Desta forma, foi possível imprimir maior dinâmica à *stream*, simulando a passagem do tempo real, com períodos de maior débito e outros de pausa, conforme a distribuição real dos artigos pelos dias em questão.

Foi inserida uma consulta na *stream* a cada 61 dias percorridos, o que corresponde, *grosso modo*, a uma atualização do *ranking* de artigos a cada 2 meses de tempo real.

5.2.3 Facebook

Este grafo contém os utilizadores do Facebook de Nova Orleães (vértices), com arcos entre utilizadores ligados por amizade nesta rede social. Possui um total de 63731 vértices e 1545686 arcos. Cobre o intervalo entre 5 de setembro de 2006 e 21 de janeiro de 2009. Para algumas das ligações (arcos), foi possível determinar o momento (*timestamp*) em que foi estabelecida; para outros, essa informação não pôde ser determinada [82, 58]. Também neste caso o grafo é disponibilizado no formato de lista de arcos em formato CSV (separado por espaço), cada um com o *timestamp* respetivo (com granularidade de um segundo), ou com a informação de que o mesmo não pôde ser determinado.

Tomando partido desta diferença, tomou-se como grafo inicial o que é formado por todos os arcos que não possuem informação temporal, reservando-se os restantes para a *stream* de atualizações. Dessa forma, o grafo inicial é formado por 640121 arcos. A *stream* foi também neste caso apresentada na forma de *socket*.

À semelhança do que foi feito para o *dataset* anterior, tomou-se partido da informação temporal associada aos arcos para definir o fluxo da informação na *stream*. Começou por se associar, a cada *timestamp* presente no *dataset*, a lista dos arcos que o têm associado. De seguida, percorreu-se, de forma contínua, todos os *timestamps* entre o valor mínimo e o máximo, emitindo para a *stream* a lista dos arcos a ele associados.

Foi inserida uma consulta a cada 604800 segundos do *dataset*, o que corresponde a um intervalo de 7 dias, o que totalizou 125 consultas. Após cada consulta foi efetuada uma pausa de 20 segundos antes de retomar o envio de elementos.

A resposta a cada consulta, para este e para os *datasets* anteriores, consiste numa lista dos 1000 vértices com mais importância no grafo, por ordem decrescente do seu *rank*, obtido através do algoritmo PageRank.

A tabela 5.1 sumariza as características dos três *datasets* utilizados.

Dataset	Vértices	Arcos	Referências
PolBlogs	1 490	18 091	[1, 56]
Cit-HepPh	34 546	421 578	[46, 70]
Facebook	63 731	1 545 686	[82, 58]

Tabela 5.1: Datasets de validação

5.3 Cenário experimental

Foi criado um conjunto de classes Java que testam cada um dos *datasets* mencionados com diferentes parametrizações. Estas classes constituem, na prática, a utilização da biblioteca GraphApprox e da sua API com grafos concretos, servindo também para ilustrar e validar o seu poder expressivo.

Para além destas classes, um conjunto de *scripts* permite automatizar o processo de testes, através da utilização de diferentes classes, com diferentes parametrizações, e ainda auxiliar no processo de recolha e sistematização dos resultados.

Os programas e análises foram executados numa máquina do *cluster* Sigma, com processador Intel Core i7-2600K, com 3.4 GHz e 4 cores (8 *threads*), e com 11,6 GB de memória RAM.

Para cada *dataset*, com as respetivas consultas inseridas entre as atualizações, da forma descrita, começou por se efetuar o cálculo do PageRank exato³, recolhendo o resultado (lista ordenada de vértices), assim como estatísticas relativas à execução.

As estatísticas recolhidas são de dois tipos: a dimensão do grafo, isto é, o número de vértices e de arcos, com os quais a computação, em termos de quantidade de mensagens enviadas no decorrer do algoritmo tem uma relação direta; o tempo total de execução do algoritmo, valor que é possível obter diretamente através do Flink.

Na posse destes dados, a avaliação dos resultados processou-se em dois grandes eixos: a qualidade dos resultados, isto é, a aproximação entre os resultados aproximados obtidos segundo as técnicas de processamento aproximado utilizadas e os resultados reais; e a diferença relativa em termos do tamanho do grafo (total ou sumariado) que foi necessário computar e do tempo despendido na computação.

Estabelecido este cenário, procurou-se, num primeiro instante, validar as estratégias selecionadas para processamento aproximado, bem como a sua implementação, ou seja, verificar que os resultados obtidos apresentam efetivamente qualidade pela sua aproximação aos resultados reais. Num segundo instante, o nosso foco foi verificar o efeito de diferentes parametrizações do algoritmo na qualidade dos resultados e no tamanho do grafo/tempo necessário para a computação.

Antes, porém, de apresentar resultados, é necessário descrever a forma como a qualidade dos resultados foi avaliada.

5.4 Metodologia de avaliação de *rankings*

A fim de avaliar a qualidade dos *rankings* obtidos, ou seja, da lista de vértices por ordem decrescente de importância, é necessário encontrar uma medida que, dado o resultado esperado e o obtido, indique a semelhança, ou a diferença, entre ambas, numa forma normalizada.

Existem bastantes medidas para obter a chamada *rank correlation*, contudo, na avaliação deste género de *rankings*, há que ter em conta algumas peculiaridades.

³Chamamos PageRank exato ao obtido com o algoritmo iterativo aplicado ao grafo na sua totalidade, apesar de esse resultado ser também, frequentemente, uma aproximação ao PageRank conforme definido matematicamente.

Frequentemente este tipo de listas ordenadas não é exaustivo, ou seja, não inclui todos os elementos do domínio, mas efetua um corte nalgum ponto mais ou menos arbitrário. Por exemplo, no nosso caso, podemos avaliar apenas os primeiros 100 ou 1000 resultados. Isto tem como consequência que as listas a comparar podem não ser compostas pelos mesmos elementos, o que leva a que tenham de se pôr de parte, neste problema concreto, medidas amplamente utilizadas como o τ de Kendall ou o σ de Spearman, que assumem que os elementos em ambos os *rankings* são os mesmos, ainda que noutra ordem. Por outro lado, como o corte pode ser efetuado num ponto arbitrário, a medida não pode assumir qualquer *cutoff* específico.

Ligado ao facto de estas listas poderem ser truncadas em pontos arbitrários está o facto de que os seus elementos não possuem todos a mesma importância: os elementos do topo da lista são muito mais importantes que os dos últimos lugares. Assim, uma medida de semelhança entre listas ordenadas ideal deverá possuir a propriedade de atribuir consideravelmente mais valor a diferenças (ou semelhanças) entre o 1º e o 2º lugar das listas do que entre o 49º e o 50º, por exemplo.

Com base nestas características, foi selecionada como medida de avaliação da qualidade dos *rankings* o Rank-biased Overlap (RBO), apresentado pelos seus autores como a primeira medida que satisfaz todas as condições anteriormente enunciadas [83].

Esta medida de semelhança baseia-se no cálculo da taxa de sobreposição (*overlap*) entre prefixos progressivamente maiores das listas a comparar, e está ligado a um modelo probabilístico bastante intuitivo: considere-se um utilizador que compara as duas listas e que, após comparar o primeiro elemento de cada uma, continua, com probabilidade p a comparar o seguinte. A dado momento, assim, desiste de comparar mais elementos, e é então calculada a sobreposição das listas até ao ponto em que o utilizador desistiu. O RBO é, assim, o valor esperado da sobreposição das listas nesta experiência aleatória, que varia entre 0 e 1.

O RBO recebe um parâmetro p , com $0 \leq p \leq 1$ que modela a persistência do utilizador, ou seja, a probabilidade de continuar a observar elementos das listas. Assim, com $p = 0$, apenas o primeiro elemento de cada lista é comparado; com $p = 1$, as listas são comparadas exaustivamente. O parâmetro p permite assim atribuir menor ou maior importância aos primeiros elementos das listas. Com $p = 0,9$, os primeiros 10 elementos têm um peso de cerca de 86% na avaliação; com $p = 0,98$, o mesmo peso é atribuídos aos primeiros 50 elementos [83].

Para o cálculo do RBO foi utilizada uma implementação em Python disponível *online* [55]. Foi empregado um valor de $p = 0,99$.

5.5 Inicialização e monitorização de atualizações

A monitorização das atualizações ao grafo, peça-chave da estratégia de processamento aproximado concebida e implementada, introduz necessariamente um *overhead* em termos de espaço em memória e tempo despendido.

A inicialização da estrutura `infoMap` implica a obtenção dos graus de todos os vértices do grafo inicial, o que na prática é realizado através dum *map-reduce* sobre os arcos do grafo, sendo de complexidade linear no número de arcos do grafo inicial. Sublinhe-se que, ao contrário da monitorização contínua, só é realizado uma vez, no início do processamento.

Realizaram-se experiências para determinar a quantidade de tempo necessária para pré-processar o grafo inicial, para cada um dos *datasets*. A tabela 5.2 apresenta a média dos resultados obtidos em 100 repetições. Ainda que significativos, sobretudo no caso do *dataset* Facebook, este tempo só é despendido na inicialização, uma vez durante o processo, pelo que não afeta de forma notável a escalabilidade da solução.

Dataset	Tempo de inicialização
PolBlogs	755 ms
CithecPh	959 ms
Facebook	3406 ms

Tabela 5.2: Tempo de inicialização da estrutura de monitorização de atualizações

A monitorização contínua das atualizações recorre a uma lista associativa que faz corresponder o identificador de cada vértice com estrutura do tipo `UpdateInfo`. Em termos do espaço necessário, visto que cada estrutura `UpdateInfo` é composta por cinco campos do tipo `long`, a que se deve acrescentar o espaço ocupado pelo identificador do vértice, que podemos assumir, sem perda de generalidade, ser também do tipo `long`, obtemos 48 bytes por registo. Assumindo, por hipótese, um grafo com 1 milhão de vértices, e considerando o uso dum `HashMap` com um fator de carga de 0,5, podemos estimar cerca de 100 MB utilizados nesta estrutura, como limite superior. Este valor, ainda que não seja propriamente desprezável, é no entanto viável na maior parte dos cenários, bastando recordar que é suportável na generalidade dos *laptops* comuns, muitos deles, hoje, já com 8 GB de memória RAM disponível. Com 10 milhões de vértices, a estimativa é de 1GB, mas nesse caso, para uma carga de trabalho com estas dimensões, já é necessário, num cenário realista, ter em conta máquinas, ou conjuntos de máquinas em *cluster*, com maiores capacidades em termos de memória de trabalho disponível.

Tendo em conta que, expectavelmente, apenas uma pequena parte dos vértices sofrerá atualizações entre consultas consecutivas, pode ensaiar-se uma solução intermédia, em que o resultado do processamento inicial, destinado a obter os graus dos vértices, é mantido em armazenamento secundário (disco, HDFS, etc.) e a estrutura de monitorização apenas contém os vértices atualizados. Quando fosse necessário introduzir na lista associativa um vértice até então ainda não monitorizado, teria de se obter o seu grau inicial a partir do armazenamento secundário, o que, obviamente, traria custos acrescidos. Com esta técnica, porém, poderia encontrar-se um compromisso entre a poupança no espaço em memória ocupado e o acréscimo de processamento decorrente.

Quanto ao tempo despendido com a monitorização das atualizações, ou seja, com o incremento dos contadores de atualizações e registo dos novos graus de entrada e de saída, foi medido o tempo acumulado, para todas as atualizações recebidas entre cada consulta efetuada ao grafo, para cada um dos *datasets*. Entre consultas sucessivas, é recebido um determinado número de atualizações, e para cada uma delas é necessário tempo para efetuar o registo. Para obter uma noção exata da dimensão deste tempo, o mesmo foi medido individualmente para cada atualização. A figura 5.1 apresenta, para cada consulta efetuada ao grafo, a soma dos tempos gastos com as atualizações desde a consulta imediatamente anterior. Como pode verificar-se, esse tempo situa-se, nos exemplos utilizados, entre os 0 e cerca de 20 ms, o que é razoável neste contexto, e confirma a viabilidade desta abordagem.

5.6 Qualidade da resposta

Verificamos agora o efeito que diferentes parametrizações do nosso algoritmo têm na qualidade dos resultados obtidos. Os parâmetros livres do modelo são o valor mínimo de alteração relativa de grau para um vértice ser selecionado (`updatedRatioThreshold`) e o tamanho da vizinhança a considerar quando esse conjunto inicial de vértices é expandido (`neighborhoodSize`).

Nos gráficos subsequentes, denominou-se “iteração” cada uma das consultas ao grafo, na medida em que cada uma delas corresponde a uma nova execução do algoritmo de PageRank aproximado.

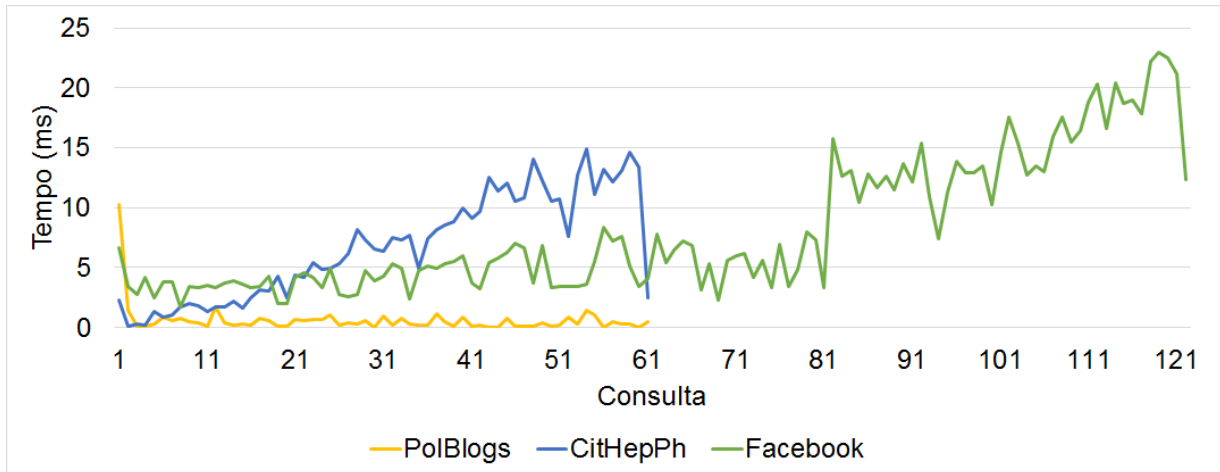


Figura 5.1: Tempo acumulado despendido com a monitorização, entre cada consulta ao grafo

5.6.1 Impacto do Valor mínimo da alteração de grau

Para esta experiência, mantendo-se um tamanho de vizinhança fixo, aqui representado por n , fez-se variar valor mínimo de alteração relativa de grau (parâmetro *updatedRatioThreshold*, aqui representado por t), a fim de observar o impacto desse parâmetro na qualidade dos resultados obtidos, em termos de PageRank. Como referido, a qualidade foi avaliada através da medida RBO, tendo em conta resultados de PageRank exatos obtidos previamente através do GraphApprox.

A fim de validar e avaliar o algoritmo de processamento aproximado de PageRank proposto, calculou-se, sempre que era recebida uma consulta ao grafo, o PageRank aproximado, segundo o método anteriormente exposto, após serem aplicadas todas as atualizações pendentes.

A figura 5.2 apresenta os valores de RBO obtidos, para cada uma das consultas realizadas, para um valor fixo $n = 0$ e t variável.

Em primeiro lugar, é nítido que valores mais elevados de t produzem resultados com pior qualidade. Isto vai de encontro ao que seria expectável, pois t mais elevado corresponde a critérios mais restritos para seleção dum vértice para recomputação, o que conduz à seleção de um menor número de vértices. Isto tem como consequência uma maior acumulação de erro proveniente de consultas anteriores, pois menos vértices têm o seu *rank* atualizado.

Em segundo lugar, observa-se uma tendência geral para a degradação da qualidade do *ranking* obtido à medida que a computação avança. Isto acontece pelo facto de a computação, nesta experiência, ser sempre aproximada, pelo que os desvios em relação aos valores reais se vão acumulando de consulta para consulta. Os resultados obtidos para o *dataset* Facebook (fig. 5.2c), ilustram, porém, uma inversão nesta tendência. Isto explica-se pelas características desta *dataset*. A partir aproximadamente da 90ª consulta, as alterações ao grafo recebidas afetam um maior número de vértices, pelo que são mais aqueles cujo *rank* é efetivamente recomputado. Este facto leva a que seja acumulado menor erro de respostas anteriores e a qualidade global da resposta melhor.

A figura 5.3 apresenta os resultados obtidos para a qualidade dos resultados com $n = 1$. Como pode verificar-se, a qualidade dos resultados melhorou de forma visível (atente-se à diferença na escala relativamente aos gráficos anteriores), ao passo que a diferença entre diferentes valores de t é agora praticamente desprezável. A explicação para este fenómeno reside no facto de agora ser efetuada uma expansão do conjunto dos vértices selecionados à sua vizinhança, com $n = 1$. Esta expansão vem diluir os efeitos de falta de representatividade que se observam quando é aumentado o valor de t .

Neste ensejo, estudamos agora mais de perto o efeito de diferentes valores n de expansão à vizinhança na qualidade dos resultados.

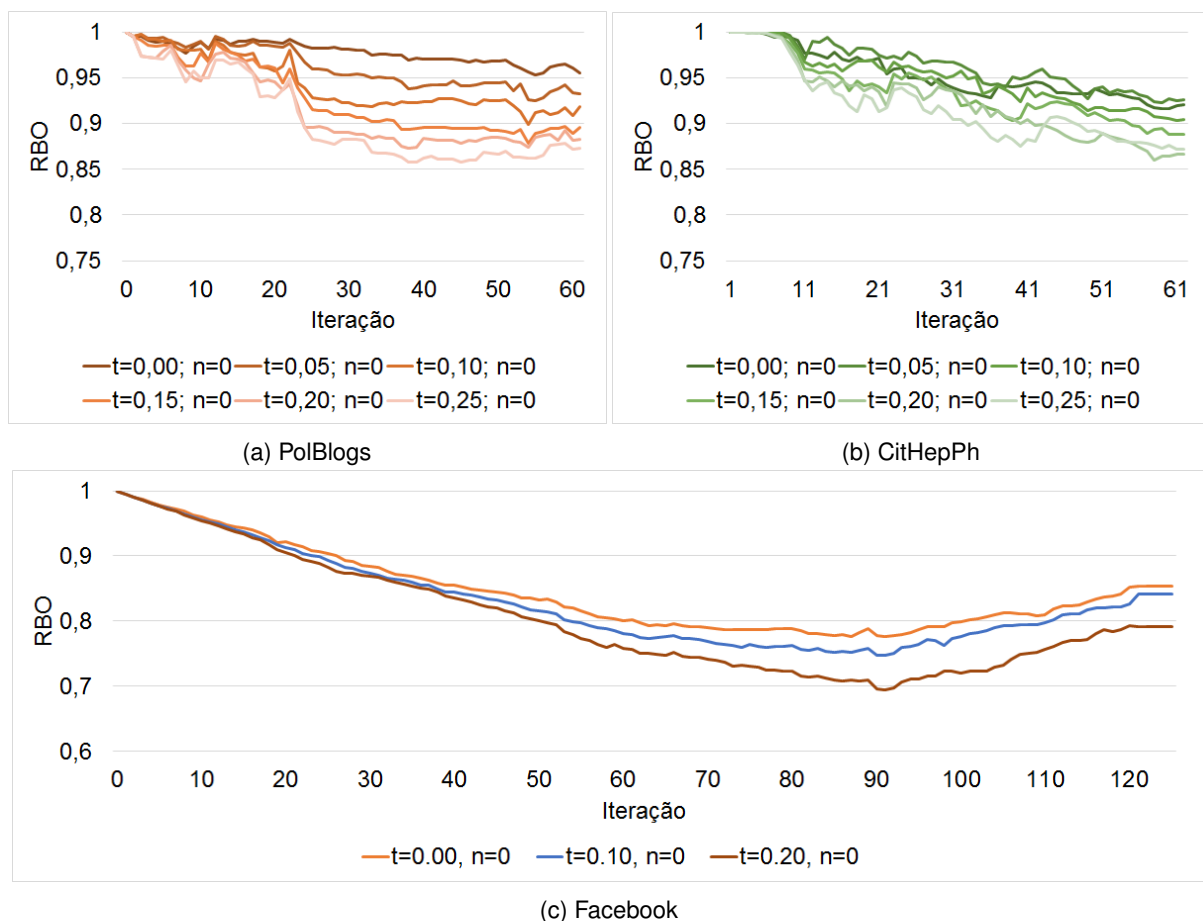


Figura 5.2: Valores de RBO para $n = 0$ e diferentes t

5.6.2 Impacto da dimensão da vizinhança

Para observar o impacto deste parâmetro, mantemos agora t fixo, fazendo variar o valor de n , e procedendo à avaliação do *ranking* obtido. As condições e cenário, no que diz respeito ao restante, são idênticos aos da experiência anterior.

A figuras 5.4 e 5.5 apresentam os resultados obtidos. O que de imediato se pode concluir é que existe uma diferença muito significativa entre $n = 0$ e $n > 0$, no que diz respeito à qualidade dos resultados obtidos, e que, pelo contrário, o impacto de diferentes valores de $n \geq 1$ é reduzido. Isto vem confirmar a intuição/assumpção de que, no caso do PageRank, não só os vértices com maior quantidade de alterações, mas também os seus vizinhos imediatos, terão provavelmente o seu *rank* modificado.

5.7 Dimensão do grafo sumário

Verificamos agora o efeito de diferentes parametrizações no tamanho do grafo sumário utilizado para efetuar a computação do PageRank aproximado, conforme exposto na sec. 3.6.4. A dimensão do grafo sumário tem evidentemente uma relação direta com a quantidade de recursos necessários para a computação, e com o tempo necessário para a completar.

Consideramos, para esta avaliação, a fração do número de arcos, em relação ao grafo original. O número de arcos tem, como se pode concluir da análise do algoritmo de PageRank utilizado, uma relação linear como número de mensagens emitidas em cada iteração do algoritmo, e é como tal uma forma de, indiretamente, avaliar o dispêndio de recursos em termos da quantidade de informação circulante.

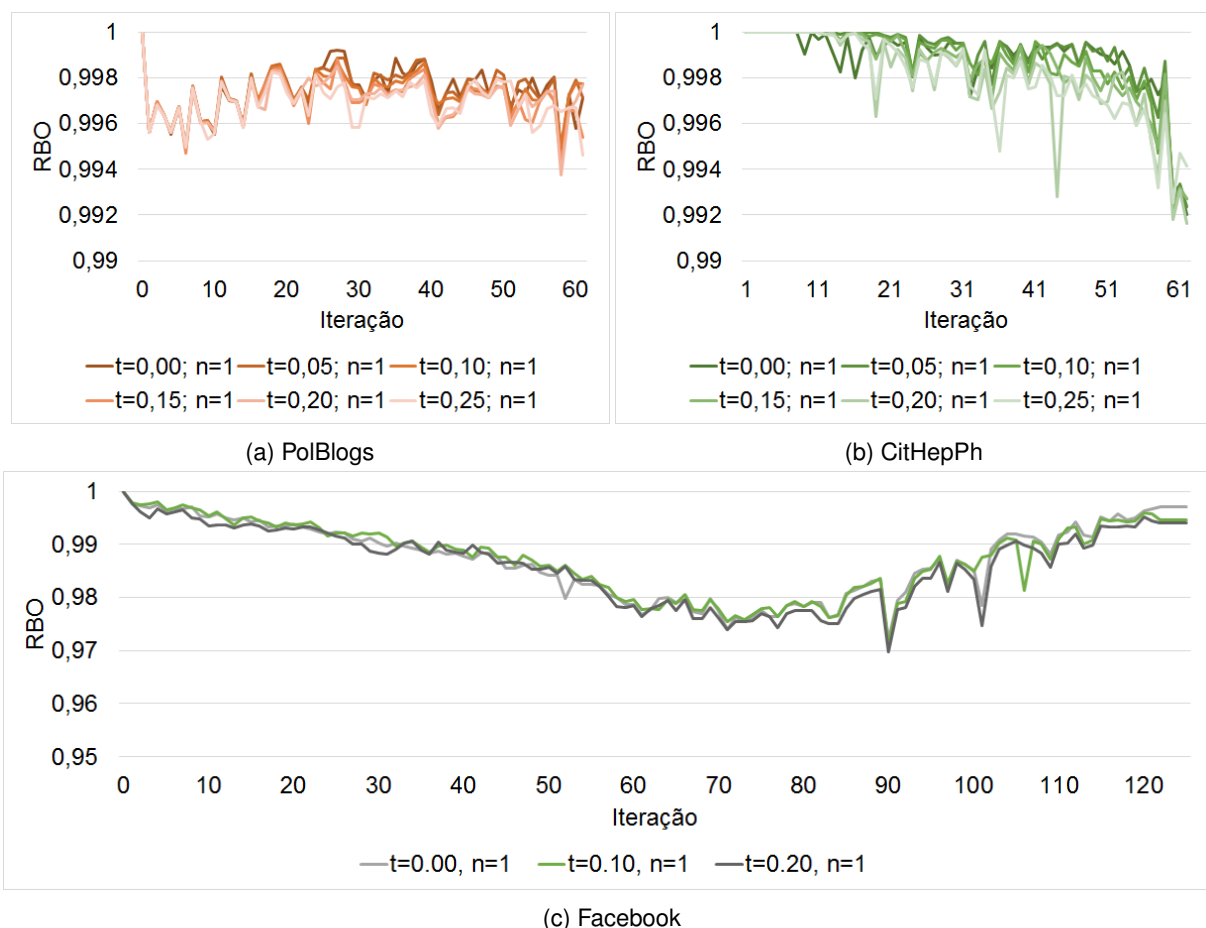


Figura 5.3: Valores de RBO para $n = 1$ e diferentes t

A figura 5.6 ilustra o impacto que diferentes valores de t têm na fração de arcos do grafo-sumário, relativamente ao grafo original, com $n = 0$, para o *dataset* Facebook, o maior dos três utilizados.

Podemos retirar duas conclusões desta observação. Em primeiro lugar, mesmo considerando todos os vértices que receberam atualizações ($t = 0,00$), verifica-se que o número de arcos do grafo sumário é consideravelmente menor que o do original, no máximo 25%, neste exemplo. Em segundo lugar, a utilização de valores superiores de t tem um efeito claro na dimensão do grafo sumário, e como tal nos recursos utilizados na computação do PageRank.

A figura 5.7 apresenta também a fração do número de arcos, mas agora para diferentes dimensões de vizinhança n , com t fixo.

O gráfico deixa clara a influência deste parâmetro na dimensão do grafo a partir do qual se efetua a computação aproximada. Se para $n = 0$ a fração de arcos conservada é bastante diminuta, o que como vimos, tem as suas consequências na qualidade da resposta, para $n = 2$ a fração de arcos é já próxima de 1, o que evidentemente terá como consequência um ganho diminuto em termos de eficiência da computação.

5.8 Desempenho

Neste ponto, observamos finalmente o efeito da estratégia de processamento aproximado seguida, em termos de tempo de processamento das respostas, e na relação entre qualidade e desempenho computacional, aqui medido através do tempo de computação e da dimensão do grafo processado em número

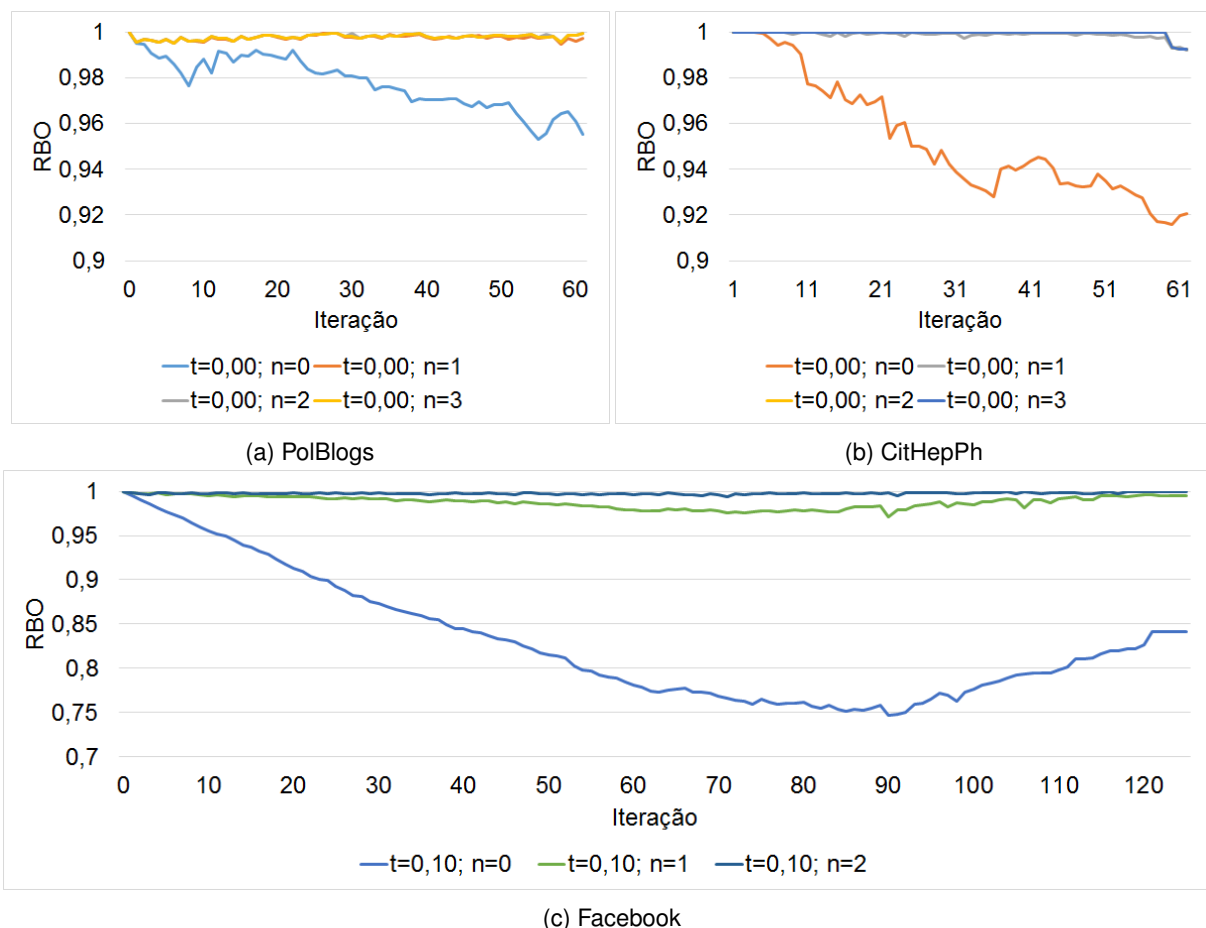


Figura 5.4: Valores de RBO para $t = 0,00$ e diferentes n

de vértices e de arcos. Os resultados apresentam essas medidas em termos de valor relativo ao obtido por computação exata do PageRank.

Na figura 5.8 podemos encontrar dois exemplos em que a técnica empregada produziu resultados com elevada qualidade, ao mesmo tempo que permitiu uma poupança significativa de recursos.

Podemos verificar nestes exemplos uma redução do processamento na casa dos 50% e mais ainda, com qualidade da solução, medida em RBO, superior a 0,9. Estes valores confirmam a potencialidade desta abordagem de redução do uso de recursos computacionais à custa dum erro tolerável na resposta.

Já a figura 5.9 ilustra como, em determinados cenários, é possível obter ganhos consideráveis em termos de tempo/uso de recurso computacionais, à custa de qualidade da resposta, que é, ainda assim, elevada, podendo ser suficiente para a aplicação solicitada, dependendo dos requisitos.

Como vemos, no caso do *dataset* Facebook (figuras 5.9c e 5.9d), o valor de RBO desce agora a valores apenas razoáveis, entre 0,7 e 0,8, mas o tempo de processamento mantém-se de forma consistente, respetivamente, entre 30 e 45% do tempo que demoraria a computar a resposta exata. Os ganhos são ainda mais evidentes se se tiver em conta a fração de vértices e de arcos. Os outros dois exemplos, para os restantes *datasets*, exibem um ganho sustentado em termos de desempenho à medida que a computação avança, com valores de RBO superiores a 0,85.

Na figura 5.10 podemos verificar como a parametrização fixa nem sempre produz resultados homogêneos e consistentes. Como vemos, a partir aproximadamente da 80ª consulta, o tempo de computação é já demasiado próximo do tempo necessário para a computação exaustiva, pelo que o processamento aproximado, neste caso, deixa de fazer sentido. Estes resultados mostram a adequação de estratégias de parametrização dinâmica, eventualmente ligadas a sistemas de mais alto nível, capazes

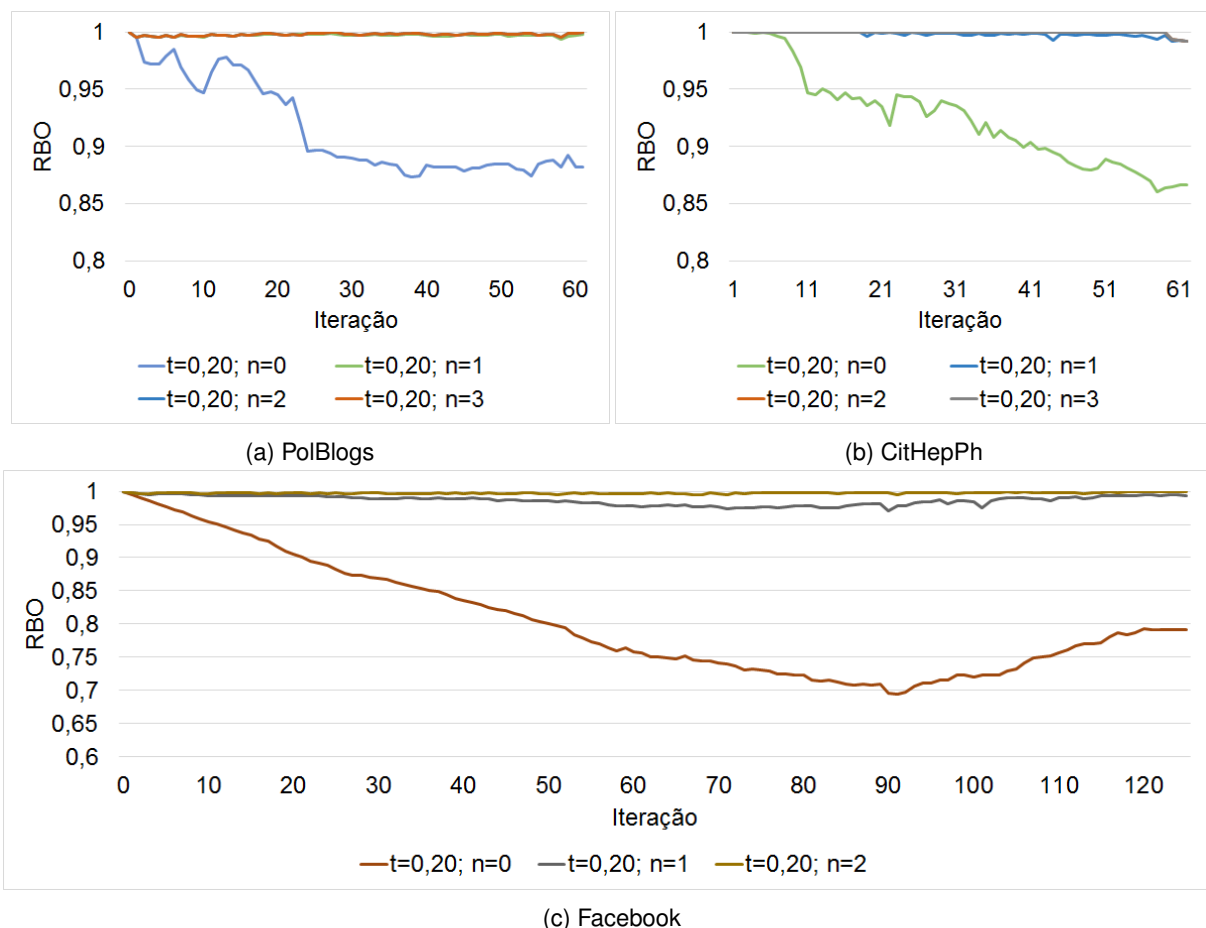


Figura 5.5: Valores de RBO para $t = 0,20$ e diferentes n

de reagir a alterações de desempenho.

Por fim, na figura 5.11 encontramos um exemplo em que a estratégia de processamento aproximado que empregamos se revela inadequada. Neste caso, o tempo de processamento é aproximadamente idêntico ao do processamento exaustivo. Neste caso ocorreu uma deficiente seleção de parâmetros. Apesar de a fração de arcos, e sobretudo de vértices, ser inferior à do grafo original, o tempo de processamento é penalizado pela necessidade de proceder à seleção dos vértices, expansão à vizinhança e construção do grafo sumário, tarefas cujos requisitos em termos de processamento acabam por absorver os já por si marginais ganhos potenciais pelo uso de processamento aproximado assim parametrizado.

5.9 Repetição da última resposta

O processamento aproximado do grafo, efetuado, no caso do PageRank, através da seleção dos vértices mais relevantes e da sumarização do grafo, não é a única alternativa oferecida pelo GraphApprox para efeitos de aproximação aos resultados reais. Uma outra hipótese consiste na simples repetição da resposta anterior, viável sobretudo quando as alterações ao grafo registadas não parecem justificar uma recomputação completa.

A fim de avaliar as potencialidades desta alternativa, foi realizada uma experiência simples, com os mesmos *datasets* e as mesmas consultas até aqui utilizados, e que consistiu em, alternadamente, como resposta às consultas, efetuar a recomputação completa e repetir o resultado anterior.

Na figura 5.12 podemos encontrar os resultados obtidos em termos de qualidade das respostas, medida através de RBO, comparado com a resposta exata, e em termos do tempo despendido, aqui

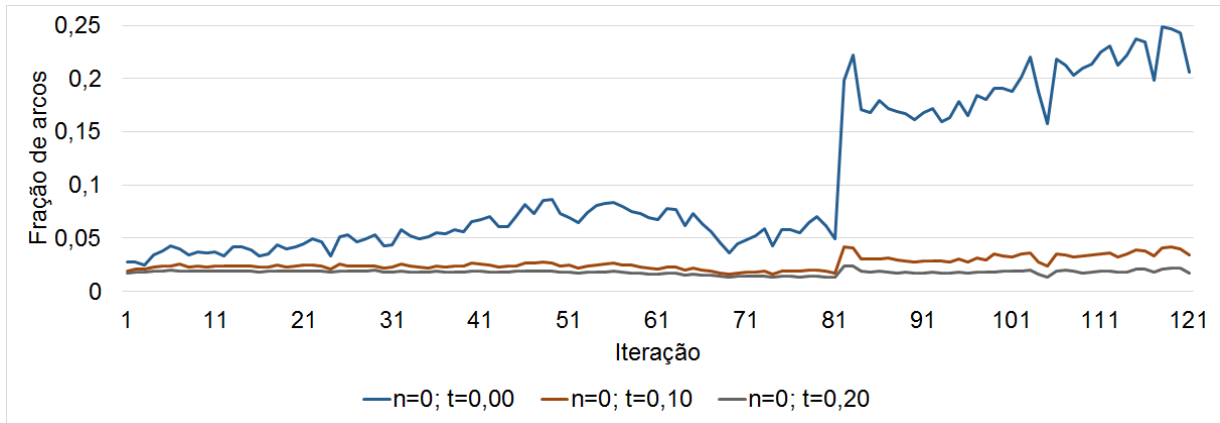


Figura 5.6: Fração de arcos do grafo sumário para $n = 0$ e diferentes t , no *dataset* Facebook

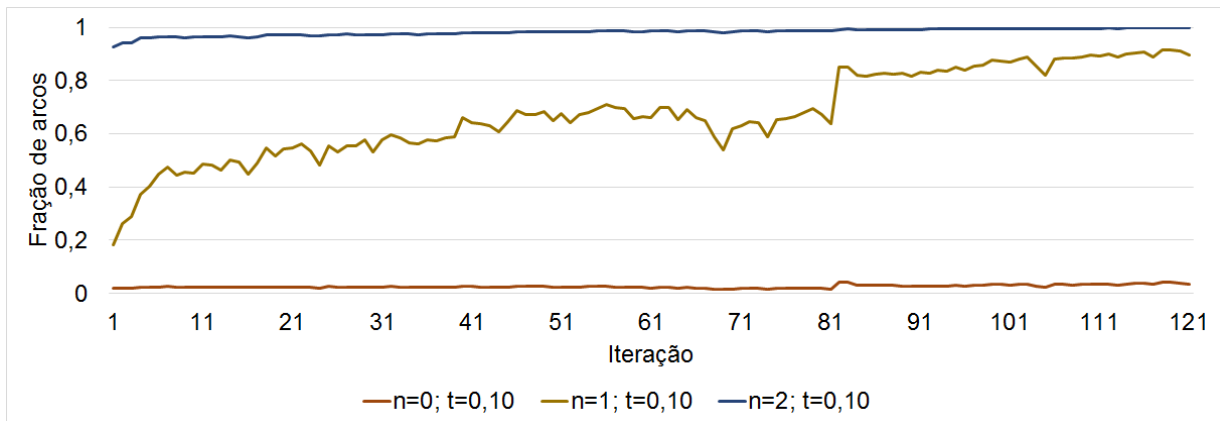


Figura 5.7: Fração de arcos do grafo sumário para $t = 0,10$ e diferentes n , no *dataset* Facebook

apresentado na forma de fração do tempo da execução imediatamente anterior (que foi uma computação exata). Os gráficos apresentam apenas os valores respeitantes às consultas cuja resposta foi a mera repetição da resposta anterior.

Da observação destes dados podemos extrair de imediato três conclusões.

Em primeiro lugar, a simples repetição de respostas permite obter uma elevada qualidade nos *rankings*, superior a 0,9, no caso do CitHepPh, e a 0,95, no caso do Facebook. Este facto espelha a natureza dos *datasets* e das suas atualizações, nomeadamente o facto de o impacto destas últimas ser reduzido quando se está em presença de um grafo já por si de grandes dimensões.

Em segundo lugar, podemos observar que o tempo necessário para devolver a resposta é consideravelmente reduzido, inferior a 10%, para o CitHepPh, e a 4%, para o Facebook. Considerando estes valores majorantes, obtém-se, para este exemplo, no conjunto da computação, tendo em conta que se alternou entre computação exata e repetição, uma redução do tempo total de computação a menos de 55%, no primeiro caso, e a menos de 52%, no segundo.

Em terceiro lugar, é visível uma tendência de redução da fração de tempo necessária para devolver a resposta. Isto explica-se facilmente pelo facto de que o tempo envolvido neste tipo de resposta é aproximadamente constante, ao passo que, à medida que o grafo se vai tornando maior, a computação exata, pelo contrário, vai exigindo progressivamente mais tempo para ser completada.

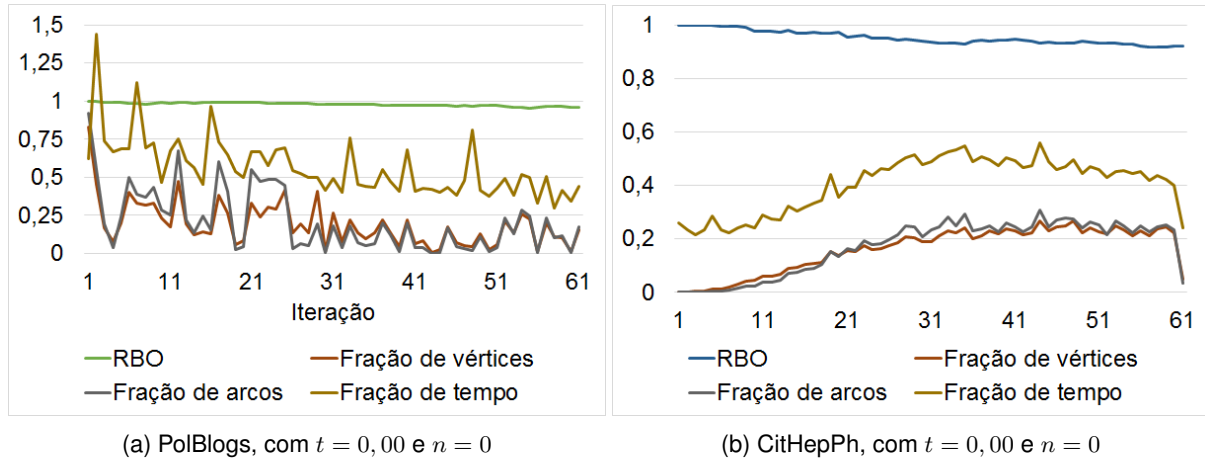


Figura 5.8: Medidas de desempenho que ilustram uma boa relação qualidade-uso de recursos

5.10 Conclusão

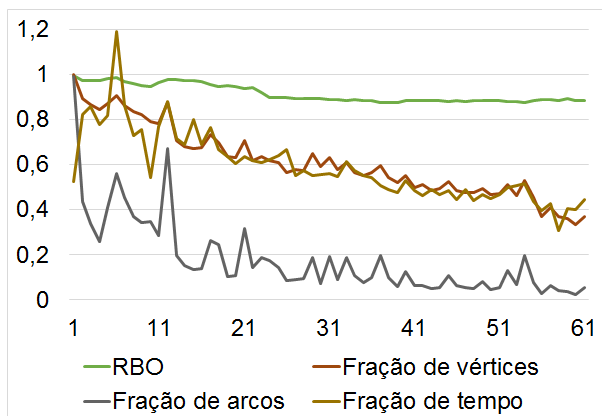
As experiências realizadas com o objetivo de avaliação mostram, antes de mais, a flexibilidade da API na criação de diferentes cenários de utilização. Os *datasets* utilizados apresentam dimensões diferenciadas, o que ilustra também a capacidade de acomodar cargas de trabalho mais ou menos exigentes.

Os resultados obtidos permitem concluir que o processamento aproximado que foi implementado produz resultados corretos, porque próximos dos resultados exatos. Boa parte da degradação de qualidade que se observa tem a ver com a acumulação de erro ao longo das iterações, pois a técnica utilizada envolve a utilização de parte dos resultados anteriores. A intercalação de processamento aproximado com processamento exato permitiria travar a acumulação de erro e produzir resultados de qualidade superior, à custa, claro, de processamento adicional.

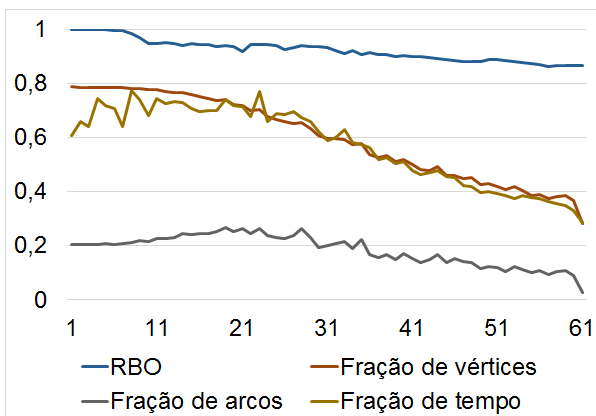
No caso de existir alguma flexibilidade quanto ao grau de aproximação dos resultados, as experiências mostram que é possível obter ganhos significativos em termos de desempenho, tendo em conta quer a dimensão do grafo efetivamente computado, quer o tempo necessário para a sua computação.

No que diz respeito à parametrização do processamento aproximado, verificou-se que os valores mais apropriados estão em grande parte dependentes do grafo concreto em causa e das características das atualizações que recebe. Por esse motivo, a possibilidade de parametrização dinâmica, oferecida pela API, é uma mais-valia considerável.

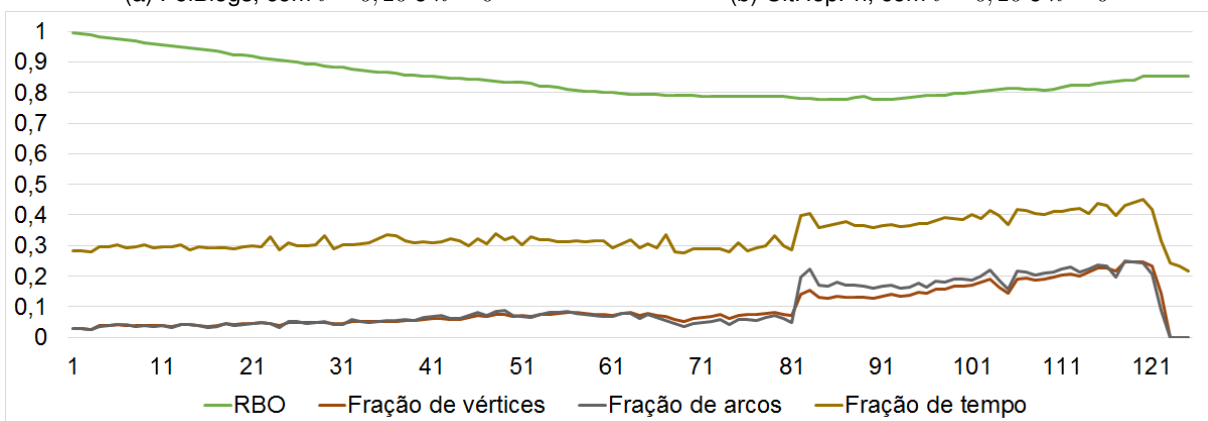
Em suma, este trabalho de utilização da biblioteca GraphApprox e da sua avaliação permitiu, para além de validar a implementação realizada, ilustrar as potencialidades do processamento aproximado de grafos.



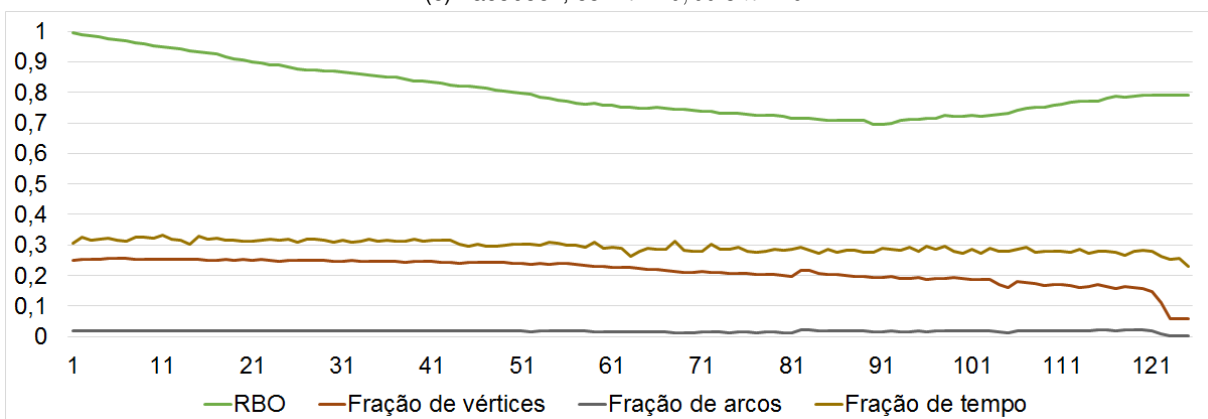
(a) PolBlogs, com $t = 0, 20$ e $n = 0$



(b) CitHepPh, com $t = 0, 20$ e $n = 0$

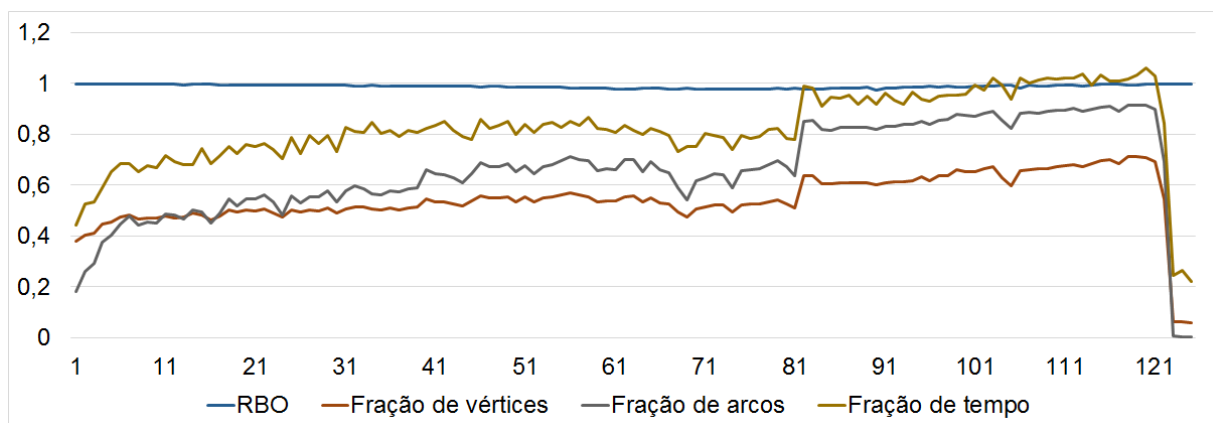


(c) Facebook, com $t = 0, 00$ e $n = 0$

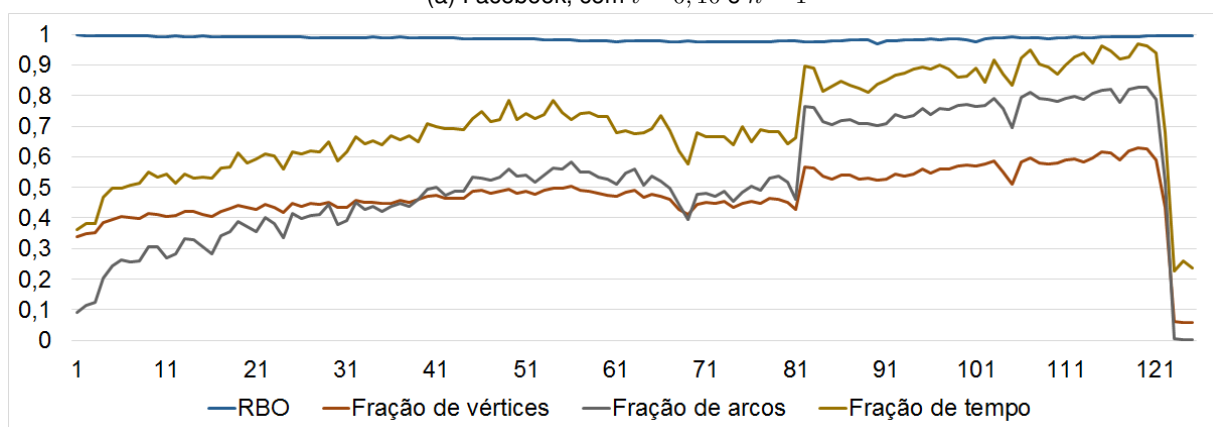


(d) Facebook, com $t = 0, 20$ e $n = 0$

Figura 5.9: Ganhos em desempenho à custa de qualidade dos resultados



(a) Facebook, com $t = 0, 10$ e $n = 1$



(b) Facebook, com $t = 0, 20$ e $n = 1$

Figura 5.10: Alteração do desempenho no decorrer do processamento

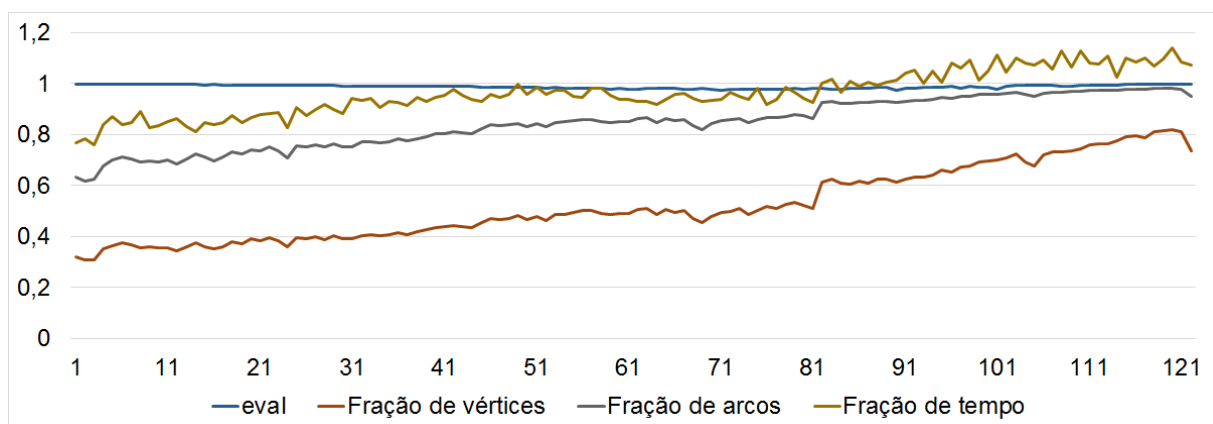
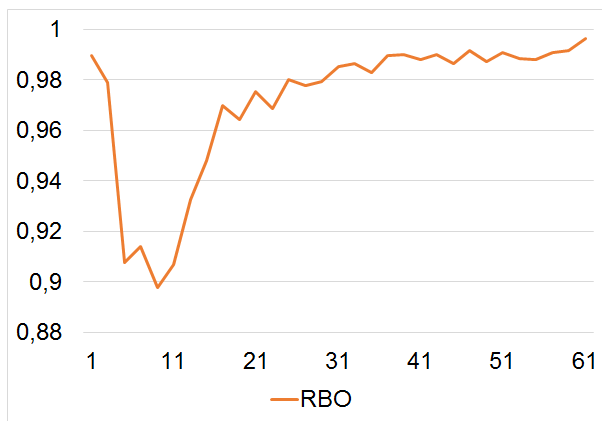
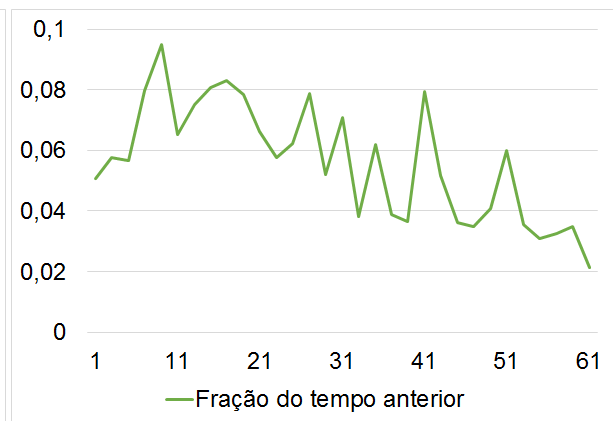


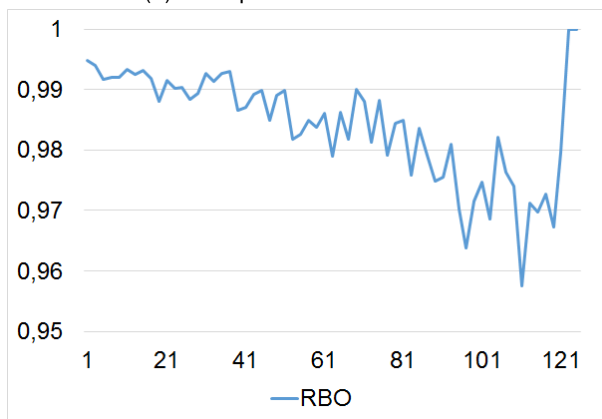
Figura 5.11: Exemplo em que o processamento aproximado não traz vantagens (*dataset* Facebook, com $t = 0, 00$ e $n = 1$)



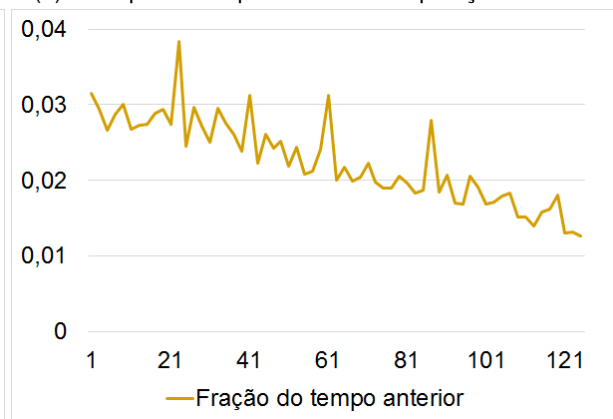
(a) CitHepPh - Valores de RBO



(b) CitHepPh - Tempo relativo à computação anterior



(c) Facebook - Valores de RBO



(d) Facebook - Tempo relativo à computação anterior

Figura 5.12: Qualidade e desempenho com estratégia de repetição da resposta anterior

Capítulo 6

Conclusões e trabalho futuro

Esta dissertação teve como tema o processamento aproximado de grafos de grande dimensão e em constante atualização. Grafos com estas características modelam muitas situações reais, e têm nos últimos tempos recebido atenção por parte da comunidade científica e empresarial.

Existem já muitas e variadas soluções dirigidas ao processamento em grande escala e em tempo real, mas os grafos, pela sua especificidade, justificam abordagens dedicadas. Na esteira do trabalho que vem sendo desenvolvido na área do processamento aproximado, como forma de racionalização de recursos computacionais, propusemo-nos, neste trabalho, estender essa estratégia à realidade dos grafos em atualização permanente. Esta necessidade nasce da verificação de que, frequentemente, o volume, ou a relevância, das atualizações a um grafo pode não justificar a recomputação completa de determinada medida ou informação que se pretende extrair do grafo; acontece bastantes vezes que essa medida ou informação pouco difere da que foi obtida anteriormente, não obstante todo o esforço computacional utilizado para a determinar. Neste contexto, o processamento aproximado e a computação adiada aparecem como meios de encontrar um equilíbrio.

As dificuldades que surgem neste contexto estão ligadas precisamente às características dos grafos, nomeadamente a sua dimensão e a sua mutabilidade, mas também a sua heterogeneidade, a diversidade de formas de representação e tratamento, o facto de servirem para representar interdependências e, por esse facto, não serem facilmente divisíveis e tratáveis pelas ferramentas usuais de processamento de dados em grande escala. Para a dificuldade na abordagem a esta problemática contribui também o facto de ser difícil conjugar, de forma harmoniosa e integrada, processamento em grande escala, representação e processamento de grafos, suporte a atualizações contínuas sob a forma de *stream* e ainda processamento aproximado. A revisão que efetuámos ao trabalho relacionado com este tema mostra a abundância de propostas teóricas e práticas para cada um destes itens, com implementações de sucesso, mas também que, apesar da diversidade de soluções, a integração de todos estes objetivos é difícil de conseguir.

Deste modo, esta dissertação teve como objetivos, para além de situar o tema no seu contexto, ensaiar uma proposta de solução efetiva de otimização do uso de recursos computacionais no processamento de grafos em constante atualização, por meio de ferramentas de processamento aproximado. Visto que dificilmente poderá existir uma solução universalmente adequada a todos os problemas envolvendo grafos, dada a sua diversidade e diferentes requisitos, assumiu-se o objetivo de fornecer, sob a forma duma API, as ferramentas para poder tomar decisões relativas ao processamento aproximado com base em informação acerca do estado atual do grafo e das atualizações recebidas desde a última computação. Das contribuições do presente trabalho fazem também parte a implementação dessa solução e a sua avaliação.

Para esse efeito, foi concebida uma biblioteca denominada GraphApprox, destinada a ser utilizada

juntamente com o Apache Flink e a sua biblioteca de grafos, Gelly, bem como com uma *stream* de atualizações ao grafo, de natureza genérica. O GraphApprox situa-se entre o utilizador e o Flink, servindo de ponte para poder efetuar o processamento aproximado. Para esse efeito, recebe a *stream* de atualizações e procede ao registo das mesmas, podendo a qualquer momento fornecer estatísticas e informações pertinentes sobre o estado atual do grafo e das atualizações. Com base nessa informação, o utilizador pode tomar decisões relativas ao processamento, bem como configurá-lo da forma mais adequada.

Para esse efeito, foram definidas várias estruturas de dados, bem como algoritmos adequados ao registo da informação necessária e ao processamento aproximado em si. Como exemplo motivador e implementação concreta foi selecionado o algoritmo PageRank, uma bem conhecida medida de centralidade de vértices.

A API disponibilizada ao utilizador consiste nos parâmetros de inicialização e de configuração do processamento a desenvolver (grafo, *stream*, formato de saída) e ainda num conjunto de métodos *callback* que permitem, no momento duma consulta ao grafo, ter acesso a toda a informação relativa ao grafo e às atualizações e dar respostas adequadas relativamente ao tipo de processamento a efetuar (exato ou aproximado), bem como alterar dinamicamente a sua configuração.

No capítulo relativo à implementação, foi apresentado o contexto tecnológico em que a mesma se inseriu, a saber, Flink e Java, esclarecendo-se alguns aspetos relativos ao funcionamento do Flink e da sua API, nomeadamente o seu estilo de programação funcional com operadores. Após algumas questões de pormenor, foi também esclarecida a organização do projeto implementado.

A solução foi avaliada, com diferentes grafos adaptados de situações reais, tendo em conta as questões da qualidade dos resultados e do desempenho do sistema na obtenção dos mesmos. Estudou-se ainda o impacto de diferentes parâmetros presentes na configuração num e noutro eixo de avaliação. O resultado da avaliação veio confirmar a versatilidade da API disponibilizada e ainda as potencialidades do processamento aproximado como resposta à problemática que motivou o presente trabalho.

Para concluir, podemos enumerar algumas pistas de trabalho futuro, que refletem algumas insuficiências da solução que propomos, mas também evidenciam o seu potencial de crescimento e de aplicação.

- Implementar e estudar os méritos e limitações da solução com outros algoritmos, quer diferentes algoritmos de centralidade, quer algoritmos de outra natureza;
- Disponibilizar através da API a definição de algoritmos de aproximação personalizados;
- Delinear estratégias e suportar alterações ao grafo não topológicas;
- Explorar e testar outras estratégias de decisão em relação ao processamento, com base nas atualizações e no estado atual do grafo;
- Explorar e testar as potencialidades de processamento paralelo e distribuído do Apache Flink;
- Alargar a avaliação a *datasets* diversos, com remoção de arcos e de vértices, e também de cargas de trabalho em tempo real, não adaptadas;
- Verificar a viabilidade de conexão do GraphApprox a um sistema externo que permita tomar decisões mais complexas com base na informação disponível.

Bibliografia

- [1] Lada A Adamic e Natalie Glance. «The Political Blogosphere and the 2004 U.S. Election: Divided They Blog». Em: *Proceedings of the 3rd International Workshop on Link Discovery*. LinkKDD '05. New York, NY, USA: ACM, 2005, pp. 36–43. ISBN: 1595932151. URL: <http://doi.acm.org/10.1145/1134271.1134277>.
- [2] Sameer Agarwal et al. «BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data». Em: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. New York, NY, USA: ACM, 2013, pp. 29–42. ISBN: 9781450319942. URL: <http://doi.acm.org/10.1145/2465351.2465355>.
- [3] Sameer Agarwal et al. «Knowing when You'Re Wrong: Building Fast and Reliable Approximate Query Processing Systems». Em: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. New York, NY, USA: ACM, 2014, pp. 481–492. ISBN: 9781450323765. URL: <http://doi.acm.org/10.1145/2588555.2593667>.
- [4] Charu Aggarwal e Karthik Subbian. «Evolutionary Network Analysis: A Survey». Em: *ACM Comput. Surv.* 47.1 (2014), 10:1–10:36. ISSN: 0360-0300. URL: <http://doi.acm.org/10.1145/2601412>.
- [5] Alexander Alexandrov et al. «The Stratosphere platform for big data analytics». Em: *The VLDB Journal* 23.6 (mai. de 2014), pp. 939–964. ISSN: 1066-8888. URL: <http://link.springer.com/10.1007/s00778-014-0357-y>.
- [6] L. Amini et al. «Adaptive Control of Extreme-scale Stream Processing Systems». Em: *26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*. IEEE, 2006, pp. 71–71. ISBN: 0769525407. URL: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1648858>.
- [7] *Apache Flink 1.2-SNAPSHOT Documentation: Apache Flink Documentation*. URL: <https://ci.apache.org/projects/flink/flink-docs-release-1.2/index.html#stack> (acedido em 12/10/2016).
- [8] *Apache Flink 1.2-SNAPSHOT Documentation: Flink DataSet API Programming Guide*. URL: <https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/batch/index.html> (acedido em 12/10/2016).
- [9] *Apache Flink 1.2-SNAPSHOT Documentation: Flink DataSet API Programming Guide - Semantic Annotations*. URL: <https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/batch/index.html#semantic-annotations> (acedido em 15/10/2016).
- [10] *Apache Flink 1.2-SNAPSHOT Documentation: General Architecture and Process Model*. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.2/internals/general_arch.html#general-architecture-and-process-model (acedido em 12/10/2016).
- [11] *Apache Flink 1.2-SNAPSHOT Documentation: Iterative Graph Processing*. URL: https://ci.apache.org/projects/flink/flink-docs-master/dev/libs/gelly/iterative_graph_processing.html (acedido em 13/09/2016).

- [12] *Apache Flink 1.2-SNAPSHOT Documentation: Library Methods*. URL: https://ci.apache.org/projects/flink/flink-docs-master/dev/libs/gelly/library_methods.html#pagerank (acedido em 05/10/2016).
- [13] *Apache Flink: Features*. URL: <http://flink.apache.org/features.html> (acedido em 14/10/2016).
- [14] *Apache Spark - Lightning-Fast Cluster Computing*. URL: <http://spark.apache.org> (acedido em 14/10/2016).
- [15] Brain Brian Babcock et al. «Load Shedding Techniques for Data Stream Systems». Em: *In Proc. of the 2003 Workshop on Management and Processing of Data Streams (MPDS)*. 2003, pp. 1–3. URL: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.5.1941>.
- [16] Bahman Bahmani et al. «PageRank on an Evolving Graph». Em: *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '12. New York, NY, USA: ACM, 2012, pp. 24–32. ISBN: 9781450314626. URL: <http://doi.acm.org/10.1145/2339530.2339539>.
- [17] Albert-László Barabási. *Network Science*. Cambridge University Press, 2016. ISBN: 9781107076266. URL: <http://barabasi.com/networksciencebook/>.
- [18] Albert-László Barabási e Réka Albert. «Emergence of Scaling in Random Networks». Em: *Science* 286.5439 (out. de 1999), pp. 509–512. URL: <http://www.sciencemag.org/content/286/5439/509.abstract>.
- [19] Alain Barrat, Marc Barthélemy e Alessandro Vespignani. *Dynamical Processes on Complex Networks*. 1st. New York, NY, USA: Cambridge University Press, 2008. ISBN: 9780521879507. URL: <http://www.cambridge.org/us/academic/subjects/physics/statistical-physics/dynamical-processes-complex-networks>.
- [20] *Benchmarking Streaming Computation Engines at... | Yahoo Engineering*. URL: <http://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at> (acedido em 14/10/2016).
- [21] Pramod Bhatotia et al. «Incoop: MapReduce for Incremental Computations». Em: *Proceedings of the 2Nd ACM Symposium on Cloud Computing*. SOCC '11. New York, NY, USA: ACM, 2011, 7:1–7:14. ISBN: 9781450309769. URL: <http://doi.acm.org/10.1145/2038916.2038923>.
- [22] *BlockingQueue (Java Platform SE 8)*. URL: <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html> (acedido em 04/10/2016).
- [23] P Boldi e S Vigna. «The Webgraph Framework I: Compression Techniques». Em: *Proceedings of the 13th International Conference on World Wide Web*. WWW '04. New York, NY, USA: ACM, 2004, pp. 595–602. ISBN: 158113844X. URL: <http://doi.acm.org/10.1145/988672.988752>.
- [24] Paolo Boldi e Sebastiano Vigna. «The webgraph framework II codes for the world-wide web». Em: *Data Compression Conference, 2004. Proceedings. DCC 2004*. IEEE, 2004, pp. 528–528. ISBN: 0769520820. URL: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1281504>.
- [25] Steve Chien et al. «Link Evolutions: Analysis and Algorithms». Em: *Internet Math*. 1.3 (2003), pp. 277–304. URL: <http://projecteuclid.org/euclid.im/1109190963>.
- [26] Thomas H Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844, 9780262033848. URL: <https://mitpress.mit.edu/books/introduction-algorithms>.
- [27] Tathagata Das et al. «Adaptive Stream Processing Using Dynamic Batch Sizing». Em: *Proceedings of the ACM Symposium on Cloud Computing*. SOCC '14. New York, NY, USA: ACM, 2014, 16:1–16:13. ISBN: 9781450332521. URL: <http://doi.acm.org/10.1145/2670979.2670995>.

- [28] Easley David e Kleinberg Jon. *Networks, Crowds, and Markets: Reasoning About a Highly Connected World*. New York, NY, USA: Cambridge University Press, 2010. ISBN: 9780521195331. URL: <http://www.cambridge.org/us/academic/subjects/computer-science/algorithmics-complexity-computer-algebra-and-computational-g/networks-crowds-and-markets-reasoning-about-highly-connected-world>.
- [29] Jeffrey Dean e Sanjay Ghemawat. «MapReduce: Simplified Data Processing on Large Clusters». Em: *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 1–13. URL: <http://dl.acm.org/citation.cfm?id=1251254.1251264>.
- [30] E W Dijkstra. «A note on two problems in connexion with graphs». Em: *Numerische Mathematik* 1.1 (1959), pp. 269–271. ISSN: 0945-3245. URL: <http://dx.doi.org/10.1007/BF01386390>.
- [31] D Ediger et al. «Massive streaming data analytics: A case study with clustering coefficients». Em: *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*. 2010, pp. 1–8. URL: <http://www.cc.gatech.edu/~bader/papers/StreamingCC.html>.
- [32] Sérgio Esteves, João Nuno Silva e Luís Veiga. «Fluchi: a quality-driven dataflow model for data intensive computing». Em: *Journal of Internet Services and Applications* 4.1 (2013), p. 12. ISSN: 1869-0238. URL: <http://www.jisajournal.com/content/4/1/12>.
- [33] Sérgio Esteves e Luís Veiga. «WaaS: Workflow-as-a-Service for the Cloud with Scheduling of Continuous and Data-Intensive Workflows». Em: *The Computer Journal* (2015). URL: <http://comjnl.oxfordjournals.org/content/early/2015/01/08/comjnl.bxu158.abstract>.
- [34] Sérgio Esteves et al. «Incremental dataflow execution, resource efficiency and probabilistic guarantees with Fuzzy Boolean nets». Em: *Journal of Parallel and Distributed Computing* 79-80 (mai. de 2015), pp. 52–66. ISSN: 07437315. URL: <http://www.sciencedirect.com/science/article/pii/S0743731515000507>.
- [35] Inigo Goiri et al. «ApproxHadoop: Bringing Approximations to MapReduce Frameworks». Em: *SIGPLAN Not.* 50.4 (2015), pp. 383–397. ISSN: 0362-1340. URL: <http://doi.acm.org/10.1145/2775054.2694351>.
- [36] Joseph E Gonzalez et al. «GraphX: Graph Processing in a Distributed Dataflow Framework». Em: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 599–613. ISBN: 9781931971164. URL: <http://dl.acm.org/citation.cfm?id=2685048.2685096>.
- [37] Joseph E Gonzalez et al. «PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs». Em: *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX, 2012, pp. 17–30. ISBN: 9781931971966. URL: <https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez>.
- [38] *High-throughput, low-latency, and exactly-once stream processing with Apache Flink | data Artisans*. URL: <http://data-artisans.com/high-throughput-low-latency-and-exactly-once-stream-processing-with-apache-flink> (acedido em 14/10/2016).
- [39] Michael Himsolt. *GML: A portable Graph File Format*. Rel. téc. 94030 Passau, Germany: Universität Passau, 1999. URL: https://www.researchgate.net/publication/228572038_Gml_A_portable_graph_file_format.

- [40] Benjamin Hindman et al. «Mesos: A Platform for Fine-grained Resource Sharing in the Data Center». Em: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 295–308. URL: <http://dl.acm.org/citation.cfm?id=1972457.1972488>.
- [41] Pili Hu e Wing Cheong Lau. «A Survey and Taxonomy of Graph Sampling». Em: *CoRR* abs/1308.5 (ago. de 2013). URL: <https://arxiv.org/abs/1308.5865>.
- [42] Supun Kamburugamuve e Geoffrey Fox. *Survey of Distributed Stream Processing*. Rel. téc. February. Bloomington, IN, USA: School of Informatics e Computing, Indiana University, 2016. URL: https://www.researchgate.net/publication/299411481_Survey_of_Distributed_Stream_Processing.
- [43] Jannis Koch et al. «Complex Network Analysis on Distributed Systems: An Empirical Comparison». Em: *Proceedings of the 2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2015*. ASONAM '15. New York, NY, USA: ACM, 2015, pp. 1169–1176. ISBN: 9781450338547. URL: <http://doi.acm.org/10.1145/2808797.2808923>.
- [44] Amy Nicole Langville e Carl Dean Meyer. «Updating Pagerank with Iterative Aggregation». Em: *Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers & Posters*. WWW Alt. '04. New York, NY, USA: ACM, 2004, pp. 392–393. ISBN: 1581139128. URL: <http://doi.acm.org/10.1145/1013367.1013491>.
- [45] Daewoo Lee, Jin-Soo Kim e Seungryoul Maeng. «Large-scale incremental processing with MapReduce». Em: *Future Generation Computer Systems* 36 (jul. de 2014), pp. 66–79. ISSN: 0167739X. URL: <http://www.sciencedirect.com/science/article/pii/S0167739X13001891>.
- [46] Jure Leskovec, Jon Kleinberg e Christos Faloutsos. «Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations». Em: *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*. KDD '05. New York, NY, USA: ACM, 2005, pp. 177–187. ISBN: 159593135X. URL: <http://doi.acm.org/10.1145/1081870.1081893>.
- [47] Qing Liu. *Approximate Query Processing*. English. Ed. por LING LIU e M.TAMER ÖZSU. Boston, MA, 2009. URL: http://www.springerlink.com/index/10.1007/978-0-387-39940-9_534.
- [48] Yucheng Low et al. «Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud». Em: *Proc. VLDB Endow.* 5.8 (2012), pp. 716–727. ISSN: 2150-8097. URL: <http://dx.doi.org/10.14778/2212351.2212354>.
- [49] Yucheng Low et al. «GraphLab: A New Parallel Framework for Machine Learning». Em: *Conference on Uncertainty in Artificial Intelligence (UAI)*. Catalina Island, California, 2010. URL: <http://arxiv.org/abs/1006.4990>.
- [50] Grzegorz Malewicz et al. «Pregel: A System for Large-scale Graph Processing». Em: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 135–146. ISBN: 9781450300322. URL: <http://doi.acm.org/10.1145/1807167.1807184>.
- [51] Jasmina Malicevic, Amitabha Roy e Willy Zwaenepoel. «Scale-up Graph Processing in the Cloud: Challenges and Solutions». Em: *Proceedings of the Fourth International Workshop on Cloud Data and Platforms*. CloudDP '14. New York, NY, USA: ACM, 2014, 5:1–5:6. ISBN: 9781450327145. URL: <http://doi.acm.org/10.1145/2592784.2592789>.
- [52] Nathan Marz e James Warren. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. 1ª ed. Greenwich, CT, USA: Manning Publications Co., 2015. ISBN: 9781617290343. URL: <https://www.manning.com/books/big-data>.

- [53] Robert Ryan McCune, Tim Weninger e Greg Madey. «Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing». Em: *ACM Comput. Surv.* 48.2 (2015), 25:1–25:39. ISSN: 0360-0300. URL: <http://doi.acm.org/10.1145/2818185>.
- [54] Andrew McGregor. *Graph Mining on Streams*. Ed. por LING LIU e M.TAMER ÖZSU. Boston, MA, 2009. URL: http://www.springerlink.com/index/10.1007/978-0-387-39940-9_184.
- [55] *measures/RBO.py at master · ragrawal/measures*. URL: <https://github.com/ragrawal/measures/blob/master/measures/rankedlist/RBO.py> (acedido em 14/10/2016).
- [56] *Network data*. URL: <http://www-personal.umich.edu/~mejn/netdata> (acedido em 14/10/2016).
- [57] Mark Newman. *Networks: An Introduction*. New York, NY, USA: Oxford University Press, Inc., 2010. ISBN: 9780199206650. URL: <https://global.oup.com/academic/product/networks-9780199206650>.
- [58] *Online Social Networks Research @ MPI-SWS*. URL: <http://socialnetworks.mpi-sws.org/data-wosn2009.html> (acedido em 07/10/2016).
- [59] Lawrence Page et al. *The PageRank Citation Ranking: Bringing Order to the Web*. Technical Report 1999-66. Stanford InfoLab, 1999. URL: <http://ilpubs.stanford.edu:8090/422/>.
- [60] Andrew Pavlo et al. «A Comparison of Approaches to Large-scale Data Analysis». Em: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. SIGMOD '09. New York, NY, USA: ACM, 2009, pp. 165–178. ISBN: 9781605585512. URL: <http://doi.acm.org/10.1145/1559845.1559865>.
- [61] Daniel Peng e Frank Dabek. «Large-scale Incremental Processing Using Distributed Transactions and Notifications». Em: *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*. 2010. URL: <http://research.google.com/pubs/pub36726.html>.
- [62] Abdul Quamar, Amol Deshpande e Jimmy Lin. «NScale: neighborhood-centric large-scale graph analytics in the cloud». Em: *The VLDB Journal* 25.2 (abr. de 2016), pp. 125–150. ISSN: 1066-8888. URL: <http://dx.doi.org/10.1007/s00778-015-0405-2>.
- [63] Charles Reiss et al. «Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis». Em: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC '12. New York, NY, USA: ACM, 2012, 7:1–7:13. ISBN: 9781450317610. URL: <http://doi.acm.org/10.1145/2391229.2391236>.
- [64] Martin Rinard. «Probabilistic Accuracy Bounds for Fault-tolerant Computations That Discard Tasks». Em: *Proceedings of the 20th Annual International Conference on Supercomputing*. ICS '06. New York, NY, USA: ACM, 2006, pp. 324–334. ISBN: 1595932828. URL: <http://doi.acm.org/10.1145/1183401.1183447>.
- [65] Amitabha Roy, Ivo Mihailovic e Willy Zwaenepoel. «X-Stream: Edge-centric Graph Processing Using Streaming Partitions». Em: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. New York, NY, USA: ACM, 2013, pp. 472–488. ISBN: 9781450323888. URL: <http://doi.acm.org/10.1145/2517349.2522740>.
- [66] Gorka Sadowksi e Philip Rathle. *Fraud Detection : Discovering Connections with Graph Databases*. Rel. téc. Neo4j, 2015. URL: <https://neo4j.com/resources/fraud-detection-white-paper/>.
- [67] S Schneider et al. «Elastic scaling of data parallel operators in stream processing». Em: *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. 2009, pp. 1–12. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5161036.
- [68] Julian Shun e Guy E Blelloch. «Ligra: A Lightweight Graph Processing Framework for Shared Memory». Em: *SIGPLAN Not.* 48.8 (2013), pp. 135–146. ISSN: 0362-1340. URL: <http://doi.acm.org/10.1145/2517327.2442530>.

- [69] Yogesh Simmhan et al. «GoFFish: A Sub-graph Centric Framework for Large-Scale Graph Analytics». Em: *Euro-Par 2014 Parallel Processing: 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings*. Ed. por Fernando Silva, Inês Dutra e Vítor Santos Costa. Cham: Springer International Publishing, 2014, pp. 451–462. ISBN: 9783319098739. URL: http://dx.doi.org/10.1007/978-3-319-09873-9_38.
- [70] *SNAP: Network datasets: High-energy physics Phenomenology citation network*. URL: <http://snap.stanford.edu/data/cit-HepPh.html> (acedido em 07/10/2016).
- [71] Philip Stutz, Abraham Bernstein e William Cohen. «Signal/Collect: Graph Algorithms for the (Semantic) Web». Em: *Proceedings of the 9th International Semantic Web Conference on The Semantic Web - Volume Part I*. ISWC'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 764–780. ISBN: 3-642-17745-X, 978-3-642-17745-3. URL: <http://dl.acm.org/citation.cfm?id=1940281.1940330>.
- [72] Liwen Sun et al. «Fine-grained partitioning for aggressive data skipping». Em: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data - SIGMOD '14*. SIGMOD '14. New York, New York, USA: ACM Press, 2014, pp. 1115–1126. ISBN: 9781450323765. URL: <http://dl.acm.org/citation.cfm?doid=2588555.2610515>.
- [73] Siddharth Suri e Sergei Vassilvitskii. «Counting Triangles and the Curse of the Last Reducer». Em: *Proceedings of the 20th International Conference on World Wide Web*. WWW '11. New York, NY, USA: ACM, 2011, pp. 607–614. ISBN: 9781450306324. URL: <http://doi.acm.org/10.1145/1963405.1963491>.
- [74] Nesime Tatbul, Uur Çetintemel e Stan Zdonik. «Staying FIT: Efficient Load Shedding Techniques for Distributed Stream Processing». Em: *Proceedings of the 33rd International Conference on Very Large Data Bases*. VLDB '07. VLDB Endowment, 2007, pp. 159–170. ISBN: 9781595936493. URL: <http://dl.acm.org/citation.cfm?id=1325851.1325873>.
- [75] Nesime Tatbul et al. «Load Shedding in a Data Stream Manager». Em: *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*. VLDB '03. VLDB Endowment, 2003, pp. 309–320. ISBN: 0127224424. URL: <http://dl.acm.org/citation.cfm?id=1315451.1315479>.
- [76] Carlos H C Teixeira et al. «Arabesque: A System for Distributed Graph Mining». Em: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. New York, NY, USA: ACM, 2015, pp. 425–440. ISBN: 9781450338349. URL: <http://doi.acm.org/10.1145/2815400.2815410>.
- [77] Yuanyuan Tian et al. «From 'think like a vertex' to 'think like a graph'». Em: *Proceedings of the VLDB Endowment* 7.3 (nov. de 2013), pp. 193–204. ISSN: 21508097. URL: <http://dl.acm.org/citation.cfm?doid=2732232.2732238>.
- [78] Ankit Toshniwal et al. «Storm@Twitter». Em: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. New York, NY, USA: ACM, 2014, pp. 147–156. ISBN: 9781450323765. URL: <http://doi.acm.org/10.1145/2588555.2595641>.
- [79] Raoul-Gabriel Urma. *Processing Data with Java SE 8 Streams, Part 1*. URL: <http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html> (acedido em 02/09/2016).
- [80] Leslie G. Valiant. «A bridging model for parallel computation». Em: *Communications of the ACM* 33.8 (1990), pp. 103–111. ISSN: 0001-0782. URL: <http://doi.acm.org/10.1145/79173.79181>.
- [81] Vijay V Vazirani. *Approximation Algorithms*. New York, NY, USA: Springer-Verlag New York, Inc., 2001. ISBN: 3540653678. URL: <http://www.springer.com/gp/book/9783540653677>.

- [82] Bimal Viswanath et al. «On the Evolution of User Interaction in Facebook». Em: *Proceedings of the 2Nd ACM Workshop on Online Social Networks*. WOSN '09. Dataset: <http://socialnetworks.mpi-sws.org/data-wosn2009.html>. New York, NY, USA: ACM, 2009, pp. 37–42. ISBN: 9781605584454. URL: <http://doi.acm.org/10.1145/1592665.1592675>.
- [83] William Webber, Alistair Moffat e Justin Zobel. «A Similarity Measure for Indefinite Rankings». Em: *ACM Trans. Inf. Syst.* 28.4 (2010), 20:1–20:38. ISSN: 1046-8188. URL: <http://doi.acm.org/10.1145/1852102.1852106>.
- [84] Tom White. *Hadoop: The Definitive Guide*. 4^a ed. O'Reilly Media, Inc., 2015. ISBN: 9781491901632. URL: <http://hadoopbook.com/>.
- [85] *WorldWideWebSize.com | The size of the World Wide Web (The Internet)*. URL: <http://www.worldwidewebsite.com/> (acedido em 30/09/2016).
- [86] Reynold S Xin et al. «GraphX: A Resilient Distributed Graph System on Spark». Em: *First International Workshop on Graph Data Management Experiences and Systems*. GRADES '13. New York, NY, USA: ACM, 2013, 2:1–2:6. ISBN: 9781450321884. URL: <http://doi.acm.org/10.1145/2484425.2484427>.
- [87] Ying Xing, Stan Zdonik e Jeong-Hyon Hwang. «Dynamic Load Distribution in the Borealis Stream Processor». Em: *Proceedings of the 21st International Conference on Data Engineering*. ICDE '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 791–802. ISBN: 0769522858. URL: <http://dx.doi.org/10.1109/ICDE.2005.53>.
- [88] Da Yan et al. «Blogel: A Block-centric Framework for Distributed Computation on Real-world Graphs». Em: *Proc. VLDB Endow.* 7.14 (2014), pp. 1981–1992. ISSN: 2150-8097. URL: <http://dx.doi.org/10.14778/2733085.2733103>.
- [89] Pingpeng Yuan et al. «Fast Iterative Graph Computation: A Path Centric Approach». Em: *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '14. Piscataway, NJ, USA: IEEE, nov. de 2014, pp. 401–412. ISBN: 9781479955008. URL: <http://dx.doi.org/10.1109/SC.2014.38>.
- [90] Matei Zaharia et al. «Discretized Streams: Fault-tolerant Streaming Computation at Scale». Em: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13 1. New York, NY, USA: ACM, 2013, pp. 423–438. ISBN: 9781450323888. URL: <http://doi.acm.org/10.1145/2517349.2522737>.
- [91] Matei Zaharia et al. «Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing». Em: *NSDI'12 Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (2012), pp. 2–2. ISSN: 00221112. URL: <http://dl.acm.org/citation.cfm?id=2228301>.
- [92] Matei Zaharia et al. «Spark : Cluster Computing with Working Sets». Em: *HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing* (2010), p. 10. ISSN: 03642348. URL: <http://dl.acm.org/citation.cfm?id=1863113>.