

1º Trabalho de Grafos

Patrícia de Andrade Kovalski e Victor Hernandez Silva Maia

Linguagem utilizada para implementação da biblioteca de grafos: JavaScript

Motivos dessa escolha:

- Performance satisfatória (mesma ordem de magnitude de C, graças a técnicas de JIT);
- Alto nível (*closures*, *first-class functions*);
- Praticidade obtida por rodar no *browser*;
- Qualidade dos *package managers*, npm;
- Disponibilidade de libs: amplamente utilizada no GitHub.

Estudos de Caso

1º Caso: Memória utilizada

	Matriz de Adjacência	Vetor de Adjacência
as_graph	19.7 MB	16.9 MB
subdblp	279.7 MB	66.2 MB
dblp	---	392.9 MB

Não foi possível criar a matriz de adjacência para o grafo dblp.txt no computador utilizado. Teoricamente seriam necessários aproximadamente 113.68GB para armazenar seus 1397510 vértices.

Implementação

Para a matriz de adjacência utilizamos apenas um bit por célula ao invés de um byte assim como utilizamos apenas metade da matriz por ela ser simétrica. Além disso, armazenamos os dados em um vetor dinâmico, o qual acessamos através de uma combinação da posição desejada da matriz. Essa implementação é *cache-friendly* e otimiza o acesso aos dados.

2º e 3º Casos: Tempo de execução de 10 BFSs/DFSs

BFS	Matriz de Adjacência	Vetor de Adjacência	DFS	Matriz de Adjacência	Vetor de Adjacência
as_graph	435.28 s	0,01 s	as_graph	432.18 s	0.02 s
subdblp	1533,25 s	0.17 s	subdblp	14430,27 s	0.29 s
dblp	---	2.07 s	dblp	---	7.49 s

Seria possível reduzir consideravelmente o tempo de execução matriz de adjacência utilizando um cache dos vizinhos de cada vértice da matriz; o que nada mais seria do que um vetor de adjacência. Acreditamos, porém, que isto acabaria com o propósito de comparação entre as estruturas. Preferimos, portanto, manter a versão não-otimizada.

4º Caso: Obtendo o pai dos vértices

	BFS (* VI/V = Vértice Inicial / Vértice)						DFS (* VI/V = Vértice Inicial / Vértice)					
as_graph	VI/V*	10	20	30	40	50	VI/V*	10	20	30	40	50
	1	1	1	1	1	1	1	3	3606	1945	9722	102
	2	2	10	35	2	35	2	3	247	1258	9722	102
	3	3	1	1	1	1	3	6	37	1258	2293	2693
	4	4	10	5573	6458	27	4	4	44	1258	9722	102
	5	5	5	8	2	5	5	3	37	1258	2293	2693
subdblp	VI/V*	10	20	30	40	50	VI/V*	10	20	30	40	50
	1	1	1	1	2	2	1	428	914	1102	1643	21
	2	1	1	1	2	2	2	428	914	1102	1643	21
	3	1	1	1	2	2	3	428	914	1095	1643	21
	4	1	1	1	2	2	4	428	914	1095	1643	21
	5	5	1	1	2	2	5	428	917	1102	1643	21
dblp	VI/V*	10	20	30	40	50	VI/V*	10	20	30	40	50
	1	1226753	843078	80251	1039901	291354	1	1226753	843078	80251	325983	291354
	2	1226753	843078	367956	1009777	766662	2	1226753	762595	80251	325983	291354
	3	1226753	843078	367956	1009777	766662	3	1226753	762595	80251	325983	291354
	4	1226753	843078	367956	1009777	766662	4	1226753	762595	80251	325983	291354
	5	1226753	843078	367956	1009777	766662	5	1226753	762595	80251	325983	291354

Implementação

Tanto o pai quanto o grau de cada vértice é obtido ao rodarmos uma BFS. Os valores são atualizados em um array referente ao grafo em questão.

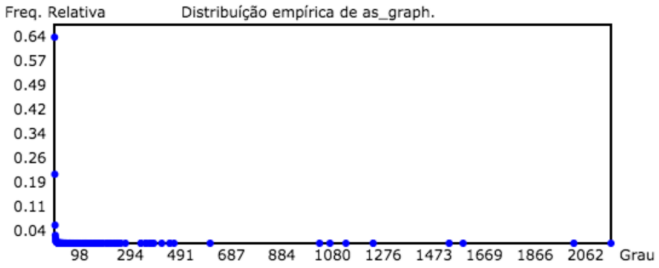
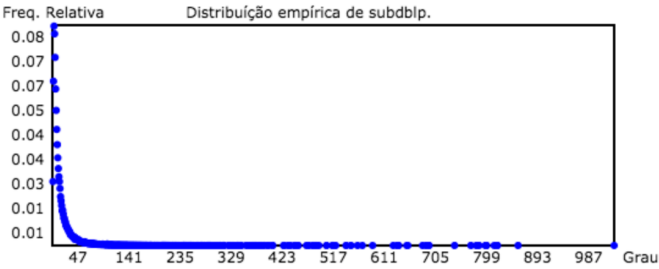
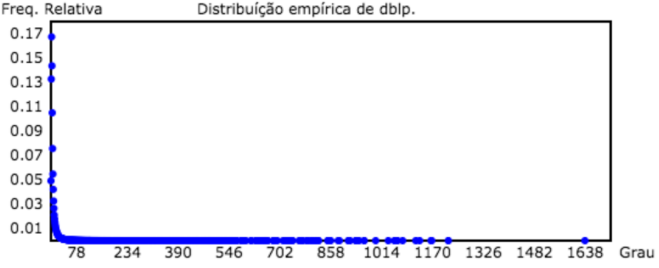
5º Caso: Componentes conexas

	as_graph	subdblp	dblp
Total de componentes conexas:	1	10	116442
Maior componente conexa:	32385	100001	1183247
Menor componente conexa:	32385	2	1

Implementação

Para otimizarmos a escolha do 'proximo vértice não marcado' utilizamos dois vetores. Ao rodarmos a BFS um vetor é preenchido com o último vértice não marcado e outro com o próximo vértice não marcado de cada vértice visitado. Ao final de cada BFS obtemos o 'próximo vértice não marcado' apenas manipulando vetores, que são eficientes para acesso aleatório.

6º Caso: Grau dos vértices

Grafo	Menor grau	Maior grau	Gráfico
as_graph	1	2161	
subdblp	1	1035	
dblp	0	1638	

7º Caso: Diâmetro do grafo

Grafo	Diâmetro	Tempo de Execução *	Gráfico
as_graph	11	8s	<p>Qt.Vértices</p> <p>Diâmetro de as_graph = 11</p> <p>Eccentricidade</p>
subdblp	7	5m 10s	<p>Qt.Vértices</p> <p>Diâmetro de subdblp = 7</p> <p>Eccentricidade</p>
dblp	24	9h	<p>Qt.Vértices</p> <p>Diâmetro de dblp = 24</p> <p>Eccentricidade</p>

* Tempo de execução medido utilizando o computador de diâmetro distribuído implementado para este trabalho. Se fossem utilizadas as versões single-core, o tempo seria maior.

Implementação

Esse algoritmo possui complexidade $O(n^2)$ e, portanto, sua execução se torna impraticável conforme o número de vértices aumenta. Calculamos que o tempo necessário para obter o diâmetro do grafo de 1.4kk vértices seria de, aproximadamente, 2 dias utilizando a máquina disponível. Para agilizar este processamento, implementamos uma arquitetura de computação distribuída.

Neste sistema, uma máquina mestra é encarregada de distribuir o trabalho entre várias máquinas trabalhadoras. Um servidor HTTP/TCP é utilizado para comunicação e, como o código está em JavaScript, para que um computador entre na rede, basta acessar o um site utilizando qualquer navegador, não sendo, portanto, necessária a instalação de qualquer tipo de software nas máquinas trabalhadoras.

Utilizando esse sistema com 4 computadores, com um total de 10 núcleos de processamento, o tempo pôde ser reduzido a cerca de 6 horas.

Código: <http://github.com/viclib/grafos>

Universidade Federal do Rio de Janeiro - UFRJ

Teoria dos Grafos

2014/2