TP1 - Grafos

José Leal Domingues Neto

Date: April 9, 2014
PAA
DCC - UFMG

1 Algoritmos

1.1 BFS

O BFS foi implementado de acordo com o algoritmo do livro. Um detalhe e dificuldade em particular foi como implementar o históarico de arestas visitadas. Como esse dado não é normalmente dado pelo algoritmo, foi implementado um Map<Node, List<Edge>> onde para cada nó a aresta visitada era adicionada, justamente com todas as arestas visitadas do nó anterior. O BFS está implementado na BFS.java, e implementa a interface AllPairs.

1.1.1 Big-O

Runtime: $O(V + E)$ para o BFS, $O(V)$ para cada nó, o que é $O(V * (V + E))$. Mas como temos $O(E)$ dentro do for para procurar a aresta de menor peso, temos $O(V * (V + 2 * E))$.
=>$O(V * E)$

Memória: Começamos com $O(V + E)$ para lista de adjacência. Porem adicionalmente guardamos os caminhos, que é $O(V * E)$.
=>$O(V * E)$

1.2 Repeated Squaring

A parte teórica foi baseada no livro, com mínimas modific
oes na implementação em si. É utilizada a matrix de adjacência para guardar e calcular o RS. Para guardar os caminhos mínimos, utiliza-se o uma matrix de listas List<Edge>[][] para guardar para cada **i** e **j** no algoritmo. Como antes, adiciona-se todas as arestas do nó anterior mais a seguinte. Ela implementa a interface AllPairs e extende a classe AbstractAdjMatrixTraverse que é dividida com o Floyd-Warshall.

1.2.1 Big-O

Runtime: $O(V^4)$ para o RS.
=>$O(V^4)$

Memória: Começamos com $O(V^2)$ para a matrix de adjacência. Porem adicionalmente guardamos os caminhos para cada **i** e **j**, que é $O(V^2 * E)$.
=>$O(V^2 * E)$

1.3 Floyd-Warshal

Para o Floyd-Warshall muito da metodologia foi reutilizada do RS para implementar o algoritmo. Já que tudo como matrix de adjacência já havia sido feito, foi criado uma classe que os dois compartilharam a AbstractAdjMatrixTraverse. Com o Floyd-Warshall, também foi usado uma matrix de arestas para cada **i** e **j**.

1.3.1 Big-O

Runtime: $O(V^3)$ para o Floyd-Warshall.
$=>O(V^3)$

Memória: Começamos com $O(V^2)$ para a matrix de adjacência. Porem adicionalmente guardamos os caminhos para cada **i** e **j**, que é $O(V^2 * E)$.
$=>O(V^2 * E)$

1.4 Johnson

O algoritmo de Johnson é complexo por haver a necessidade de implementar v'arios algoritmos em um só. Ou seja, foi implementado o Dijkstra juntamente com Bellman-Ford que não possuem fácil lógica. Para aumentar a complexidade, há a necessidade de repesar os pesos dos nós após o Bellman-Ford. Para sua implementação O j foi usada uma lista de adjacência contando os caminhos encontrados em um mapa Map<Node, List<Edge>> para cada nó. Após isso, a conta das arestas utilizadas é feita e o betweeness é determinado.

1.4.1 Big-O

Runtime: $O(V^2 lgV)$ para o Johnson com HEAP, $O(V^3)$ para o Johnson com List,
$=>O(V^3)$

Memória: Começamos com $O(V + E)$ para lista de adjacência. Porem adicionalmente guardamos os caminhos, que é $O(V * E)$.
$=>O(V * E)$

2 Johnson vs. Floyd-Warshall

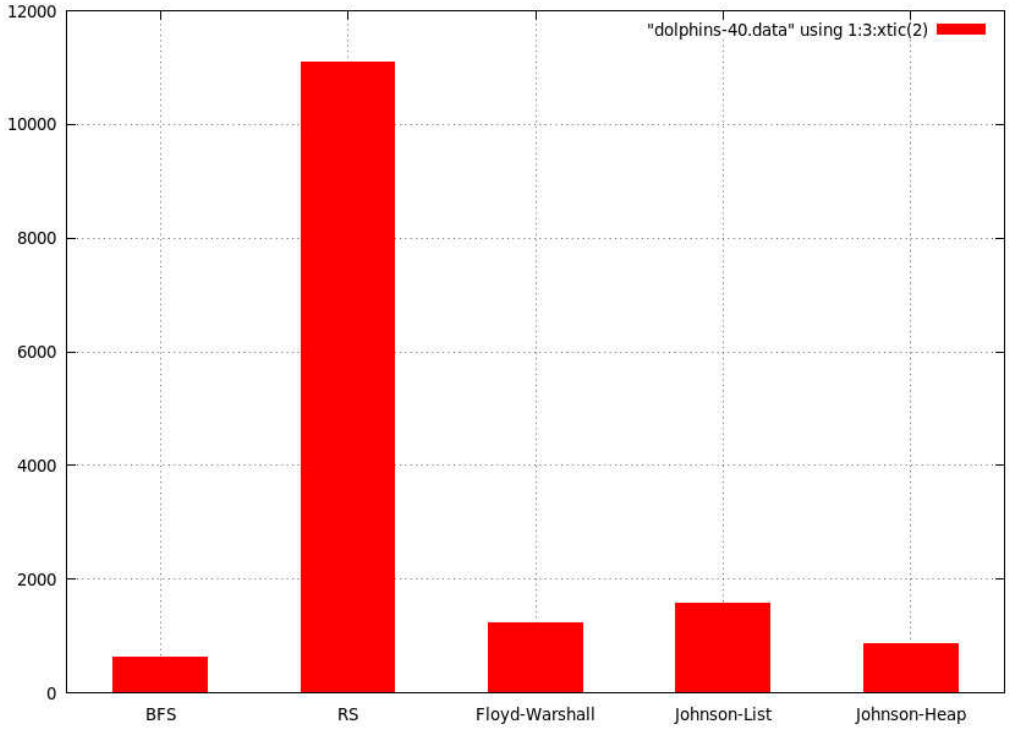
Os dois algoritmos cumprem o mesmo trabalho, porém o algoritmo de Johnson é mais recomendado quando se lida com grafos do tipo sparse. Como pode-se usar

um algoritmo mais rápido para determinar o menor caminho nos casos de poucas arestas, evita-se o uso do Floyd-Warshall.

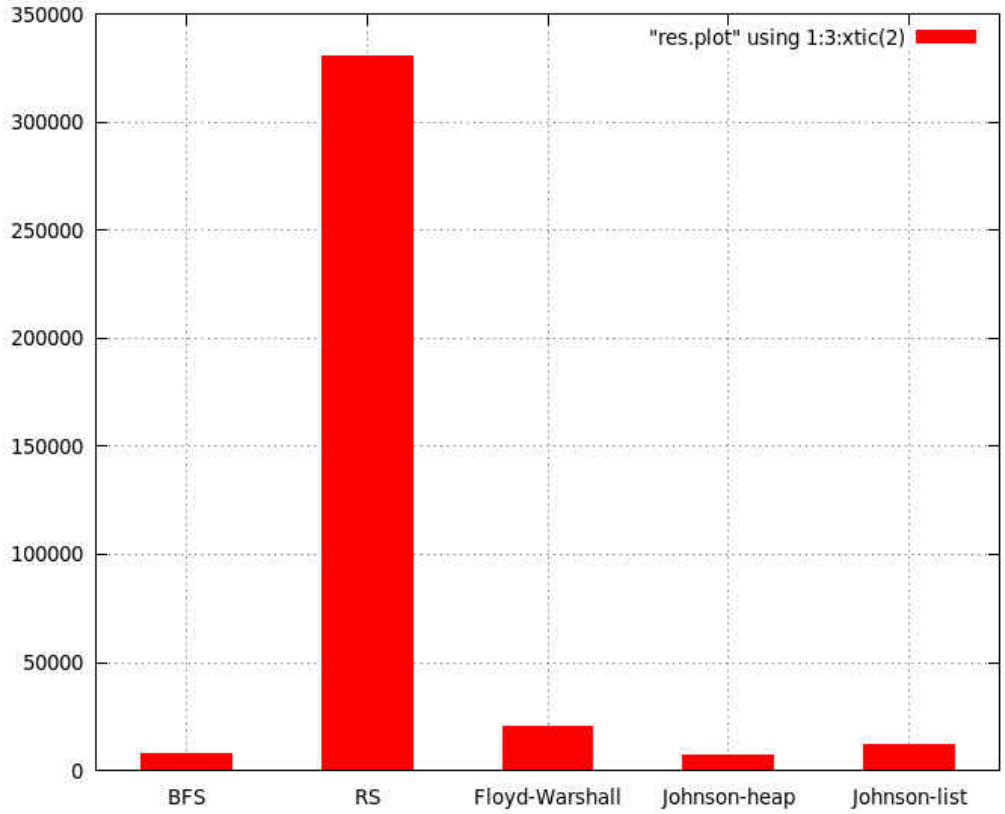
3 Tempo de execução e memória

3.1 Tempo de execução

Alguns testes foram feitos, e temos os seguintes resultados.



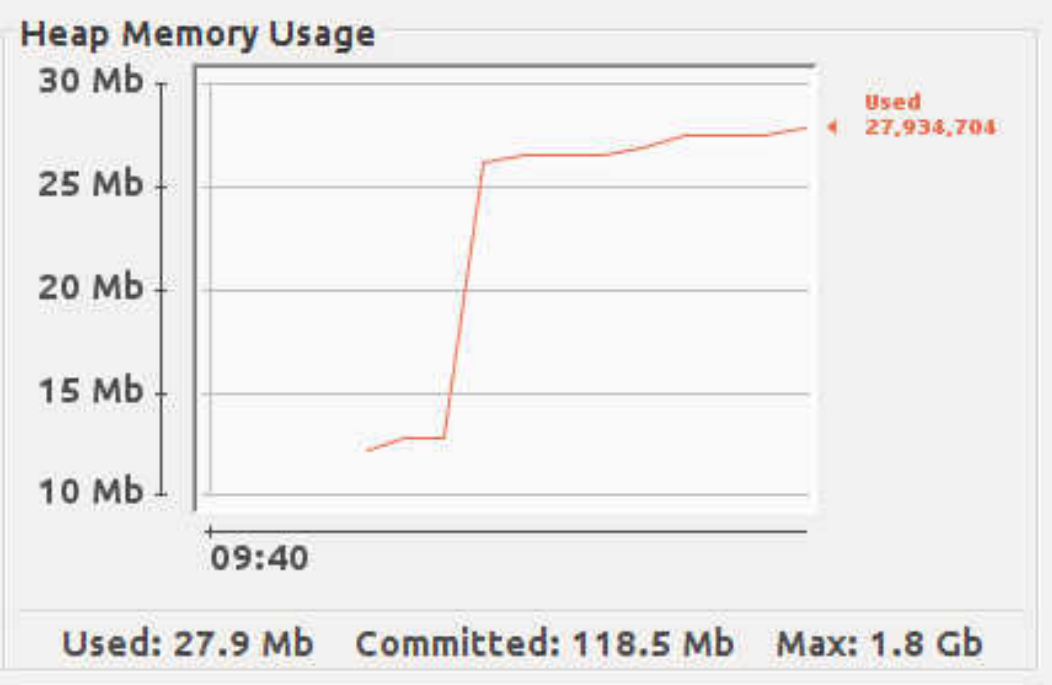
Este exemplo foi rodado em um grafo teoricamente pequeno o dolphin.gml. Ele possui por volta de 100 nóas.
Como podemos ver, o algoritmo de Repeated Squaring possui um tempo de execução muito maior do que os outros. Mesmo neste exemplo, onde o eixo Y é dado em milisegundos, vemos que já se torna impossível rodarmos o algoritmo com eficiência. Temos a seguir um exemplo com um tempo de execução maior:



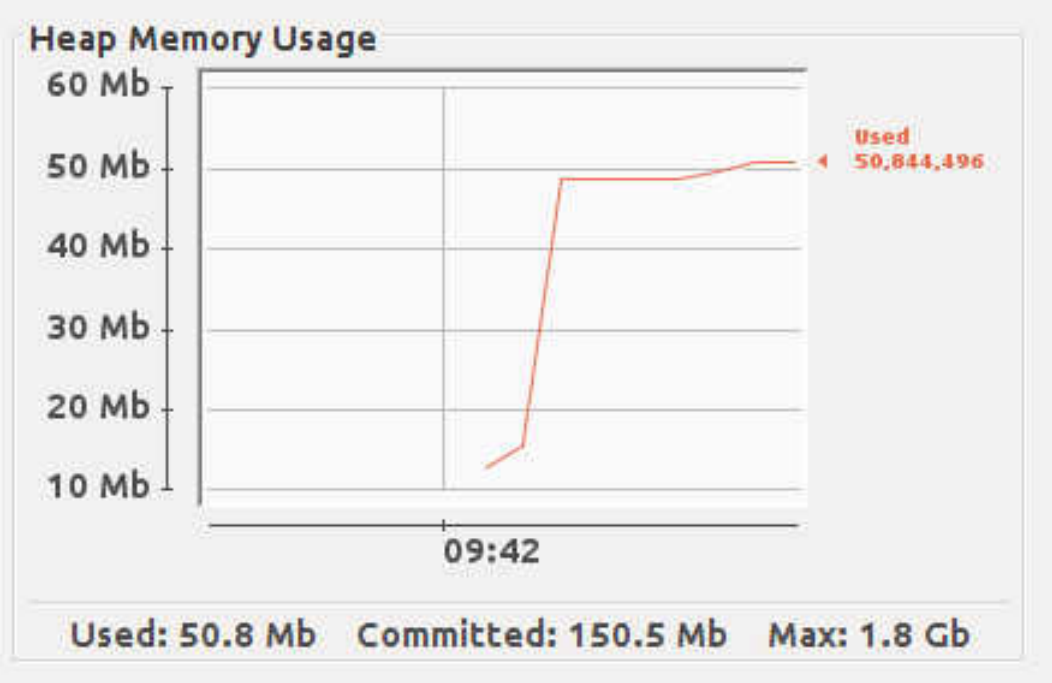
Este exemplo foi rodado no grafo adjnoun.gml, que é um pouco maior, contendo 110 nóas.
Como vemos, o RS rodou por mais de 3 minutos, enquanto os outros algoritmos demoraram menos de 30 segundos.

3.2 Memória

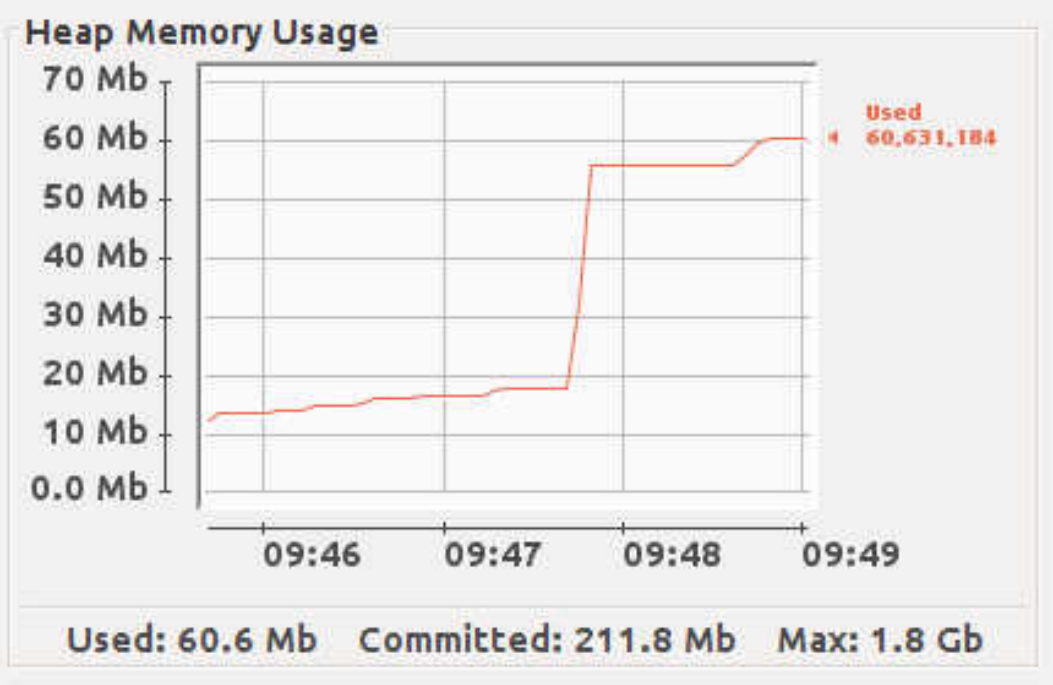
Muito se foi aprendido com este experimento. Foi visto que na verdade em cada algoritmo se troca muitas vezes espaço por velocidade. O jconsole do Java foi utilizado para apurar estes dados.



Aqui foi rodado um BFS com um o grafo dolphin.gml. A memória utilizada foi pouca e o algoritmo foi performante.



Aqui vemos o algoritmo de Floyd-Warshall rodando sob as mesmas condições. Como se vê ele necessita de quase o dobro da memória do anterior.



E por fim este é o Johnson. Vê se claramente que apesar do algoritmo de Johnson ser mais rápido, ele faz uma troca por espaço.

3.3 Ambiente testado

O Ambiente testado foi em um Dell XPS13, Intel i7 Haswell com 8Gb de memória.