Trabalho de Grafos

Lucas Saraiva Ferreira

PAA - UFMG

Keywords: PAA, Grafos

1. Introdução

O problema previamente descrito no trabalho tinha como objetivo calcular os deputados mais influentes em um grafo, a saída é dada por uma lista dos IDs ordenados por ordem de influência, do mais influente para o menos influente.

A solução do problema foi construída utilizando a linguagem Java 1.8. Todos algoritmos e estruturas implementadas foram vistas em sala (*Dijkstra* e Lista de Adjacência. As próximas subseções visam explicar todo raciocínio usado para resolver o problema, as estruturas para armazenamento de dados e os algoritmos aplicados.

1.1. Representação computacional do grafo

A estrutura escolhida para salvar o grafo lido a partir do arquivo de entrada foi a lista de adjacência. Algumas das motivações para utilizar a lista de adjacência foram: a facilidade de trabalhar com listas e o fato de ocupar espaço O(V+E) em disco para grafos esparsos.

A lista de adjacência foi implementada de forma explicita, nenhuma estrutura pronta do java (e.g LinkedList, ArrayList, List) foi utilizada. Cada deputado (Classe *Deputy*) "aponta" para outro deputado, através de um atributo de classe. Além disso, cada deputado tem uma lista de Arestas (Classe *Connected*), que por sua vez ,"apontam" para as outras Arestas deste mesmo deputado. Cada objeto do tipo *Connected* salva o peso da aresta e quem é o deputado adjacente.

1.2. Betweenness Centrality

Para entender como computar a resposta do trabalho foi necessário estudar diferentes métricas de grafos. Com as informações do trabalho, é

possível concluir que: para computar o valor de influência de um deputado, seria necessário descobrir quantas vezes um deputado i era utilizado em interações entre outros dois deputados (J,k). As interações deveriam visar a maior chance de sucesso possível. Tudo isso é expressado pelo conceito de *Betweenness Centrality*.

A formula 1 descreve a métrica *Betweenness Centrality* proposta por Freeman [1]:

$$C_b(i) = \frac{g_{jk}(i)}{g_{jk}} \tag{1}$$

onde os parâmetros da formula são:

- $C_b(i)$ = Valor total de *Betweenness Centrality* de um vértice i;
- $g_{jk}(i)$ = Numero de caminhos ideais entre os vértices J e k que i é usado como intermediário;
- g_{jk} = Número total de caminhos ideais existentes entre os vértices J e k.

Exemplo de cálculo de *Betweenness Centrality*: Se existem dois caminhos, e um vértice i foi usado somente em um destes, logo ele recebe 0.5 de *Betweenness* para esse par de vértices.

O resultado final, é dado pela soma de todos valores de $C_b(i)$ calculados para cada vértice. Tendo estes valores para cada vértice, é possível inferir quem é a pessoa mais influente do grafo G, e ordenar elas pelo seu valor de *Betweenness Centrality*.

1.3. Caminhos mínimos/ideais

Sabemos que para computar o valor de *Betweenness Centrality* de todos vértices é necessário computar todos caminhos mínimos de todos vértices para todos vértices. Para realizar tal tarefa foi escolhido implementar o algoritmo de *Dijkstra* [2] com uma pequena modificação. O mesmo foi executado V vezes, uma vez para cada vértice. O algoritmo de *Dijkstra* foi escolhido por ser um algoritmo conhecido, de fácil entendimento e por ser funcional para o grafo de entrada (O grafo de entrada não tem arestas negativas). Apesar de não ser o mais rápido entre os algoritmos de *"All pairs shortest path"*, o autor deste trabalho ainda assim o considerou uma boa escolha.

Para este trabalho, foi necessário modificar a função de relaxamento do *Dijkstra*, tornando possível calcular e salvar N caminhos mínimos entre dois vértices, caso eles existam. A justificativa para essa modificação foi: certos grafos podem ter N caminhos mínimos entre dois vértices, ou seja, dois caminhos tem o mesmo valor total e este valor é mínimo. Para o cálculo de *Betweenness Centrality* todos N caminhos mínimos de um par de vértice devem ser computados.

O pseudocódigo 1 demonstra o relaxamento do *Dijkstra* modificado, onde, caso os pesos sejam iguais, o vértice é salvo como pai do vizinho e não é adicionado ainda a lista de futuras visitas:

```
begin
    influenciaComputada = atual.totalInfluence + vizinho.peso

    if influenciaComputada < vizinho.influenciaTotal() then

        vizinho.influenciaTotal() = influenciaComputada
        listaPais ⟵ vizinho.getLista()
        listaPais.add(atual)
        filaPrioridade.add(vizinho)

    else if influenciaComputada == vizinho.influenciaTotal()
     then
        listaPais ⟵ vizinho.getLista()
        listaPais.add(atual)

end
```

Algorithm 1: Relaxamento modificado

Além de modificar o *Dijkstra*, foram necessárias fazer outras duas modificações: a forma de armazenar os pais de cada vértice e o peso nas arestas.

1. Pais: uma vez que cada par de vértices (j,k) pode ter mais de um caminho mínimo, deve ser possível reconstruir todos os caminhos a partir dos pais salvos em cada vértice. Para possibilitar a reconstrução dos N caminhos, foi necessário criar uma Lista (Java List) de pais em cada vértice. Um método recursivo (computeAllPathsFromPrev) realiza a "reconstrução" dos caminhos.

2. Peso das Arestas: Para solucionar o problema é necessário computar os caminhos com maior chance de um acordo ser realizado entre dois vértices. O algoritmo de *Dijkstra* computa caminhos mínimos e para manter essa diretriz foi necessário modificar a ordem de grandeza das arestas. Se interpretarmos cada aresta como sendo a porcentagem de similaridade entre os vértices, e a similaridade máxima como 100%, ao subtrairmos 100 pelo peso da aresta, teremos o nível de dissimilaridade entre dois vértices. Com isso, podemos utilizar o *Dijkstra* para calcular o caminho que tenha o menor nível de dissimilaridade, sendo este o caminho com a maior chance de acordo entre dois deputados.

1.4. Solução

Em resumo, o algoritmo criado funciona da seguinte forma:

1. O arquivo de entrada é lido linha a linha. Para cada novo vértice, um objeto é criado e salvo na lista de adjacências. Os pesos e vértices adjacentes são salvos dentro da lista de Arestas de cada vértice.
2. Para cada vértice é computado todos os caminhos com menor dissimilaridade para todos os outros vértices
3. Para cada vértice e cada caminho encontrado, o valor de *Betweenness Centrality* é calculado. Ou seja, o *Betweenness Centrality* é calculado de forma parcial para cada vértice.
4. Os valores de *Betweenness Centrality* são ordenados de forma decrescente, em caso de empate, ordenado pelo ID do vértice de forma crescente. O resultado é gravado em um arquivo.

2. Analise de Complexidade

O algoritmo foi dividido em 4 partes para a análise de complexidade, sendo estas operações individuais, mas sequenciais. Trechos de código foram omitidos, uma pequena explicação acompanha a complexidade. Sumario das variáveis usadas: N - Tamanho da entrada em linhas, V - número de vértices, E - número de arestas, αij total de caminhos mínimos entre i e j.

1. Leitura da Entrada: O arquivo de entrada é lido linha a linha. Baseado nisso, a leitura do arquivo em si tem uma complexidade de:

$$O(N) \tag{2}$$

2. Construção do Grafo: Para inserir uma nova aresta na lista de adjacência é necessário verificar se o par de vértices desta aresta já existe na lista de adjacência. No pior caso, será necessário percorrer a lista inteira de vértices para inserir uma nova aresta. Podemos concluir que a complexidade desta parte é de:

$$O(V * E) \tag{3}$$

3. Calculo de caminho mínimo: O cálculo do caminho mínimo de todos vértices para todos vértices foi feito utilizando o algoritmo de *Dijkstra*. A implementação do *Dijkstra* utiliza uma fila de prioridade para salvar os vértices a serem visitados. As operações na fila são mais baratas que em uma lista comum. Além disso, é necessário rodar o *Dijkstra* V vezes para cada vértice. A complexidade final do algoritmo de *Dijkstra* pode ser dado por:

$$O(VElgV) \tag{4}$$

4. Calculo de *Betwenness Centrallity*: Um método recursivo é utilizado para calcular o valor de *Betwenness Centrallity* de cada vértice. Seja $\alpha v_i j$ o número total de vértices para chegar de i ate j. A equação de recorrência pode ser vista logo abaixo:

$$T(\alpha(v_{ij})) = T(\alpha(v_{ij}) - 1) + c \tag{5}$$
$$T(\alpha(v_{ij})) = O(\alpha(v_{ij})) \tag{6}$$
$$O(V^2\alpha(v_{ij})) \tag{7}$$

5. Ordenação da saída: O método de ordenação utilizado foi o pertencente a biblioteca *Collections* do java. De acordo com a documentação oficial, a complexidade do método *sort* é:

$$O(nlg(n)) \tag{8}$$

3. Testes

Os testes foram realizados numa máquina com as seguintes configurações:

- 8 Gb RAM
- Processador I7

- Sistema Linux Ubuntu 16.04
- Interpretador Java 1.8

Duas baterias de testes foram realizadas. A primeira bateria de testes foi utilizando arquivos pequenos, com o intuito de atestar o funcionamento do algoritmo. Dentre estes arquivos, foram usados os 5 exemplos fornecidos e outros 6 exemplos com grafos onde as arestas tem o mesmo peso. Foi possível obter o resultado desejado em todos os exemplos.

A segundo bateria de testes utilizou arquivos de entrada consideravelmente grandes em relação ao número de vértices e arestas. O objetivo destes testes era analisar o tempo de execução do algoritmo.

Foram gerados arquivos de 100, 300, 400 e 600 vértices. Cada qual com duas variações: Esparso (Poucas arestas) e denso (Aproximadamente $V^2/2$ arestas, sendo V o número de vértices).

A figura 1 mostra os testes para as entradas esparsas e densas ao rodar o algoritmo completo. É possível notar que o número de vértices tem uma influência na curva de crescimento, mas os de arestas causam maior impacto. A reta laranja faz referência as entradas dos grafos densos ($E = \frac{V^2}{2}$), é notável a diferença que as arestas causam no tempo de execução.

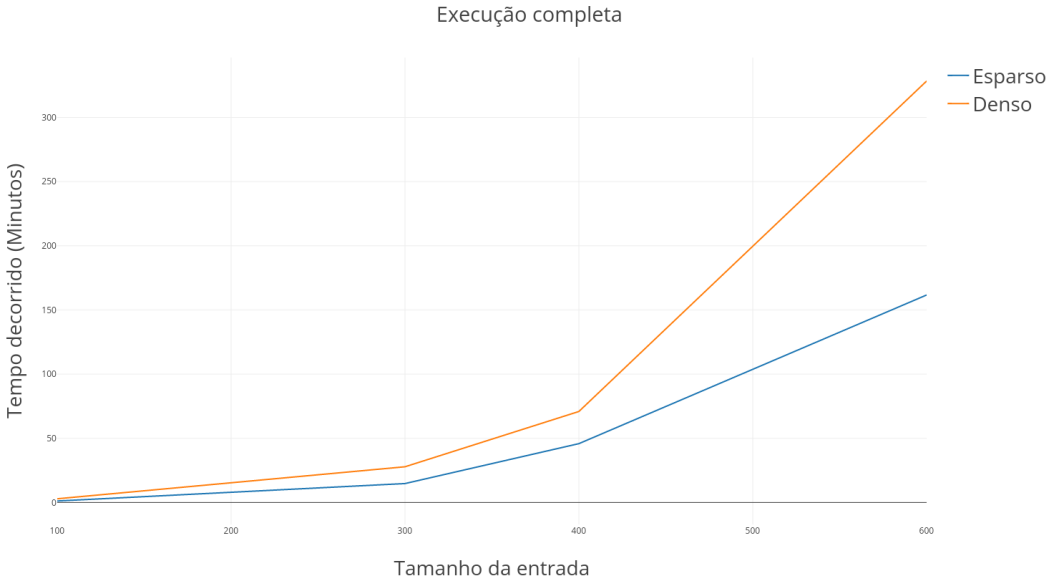

Figure 1: Execução completa

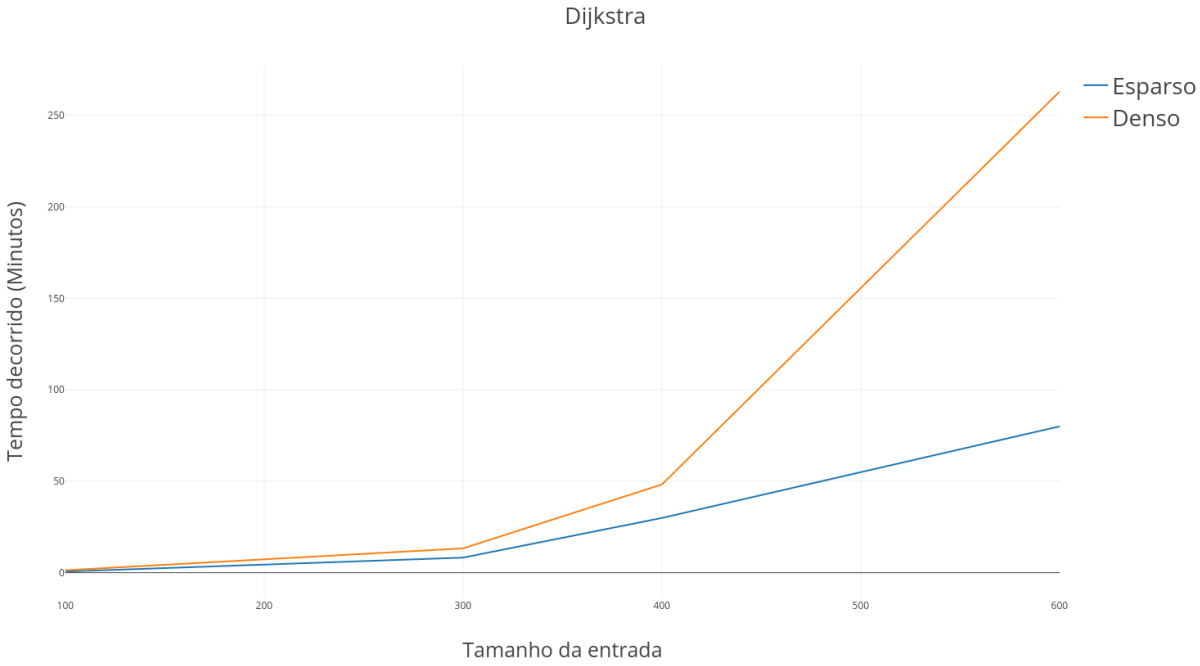

Figure 2: Execução Dijkstra

A figura 2 mostra o resultado do tempo ao executar somente o *Dijkstra*. É possível perceber por este gráfico que o *Dijkstra* é o causador do resultado no grafico 1. O algoritmo de *Dijkstra* precisa ser executado V vezes para os V vértices. Além disso, dentro dele existe um loop sobre as arestas de cada vértice. Para grafos densos, é esperado que o tempo seja consideravelmente maior em função do crescimento da entrada.

4. Conclusão dos Testes

Com os testes experimentais é possivel concluir que de fato a parte mais custosa do algoritmo é o *Dijkstra*. Para problemas de "all-pairs shortest path" roda o Dijkstra puro não é a melhor opçaõ no quesito tempo de execução. Além disso, a complexidade do *Dijkstra* depende muito da estrutura de dados usada nele. Heaps Binarios e de Fibbonnaci conseguem produzir tempos melhores, pois as operações nestes são mais baratas.

O uso de memória foi baixo, o programa salva na memória somente os vértices, as arestas e os pais de cada vértice. Todas outras informações são

descartadas assim que utilizadas.

O trabalho permitiu explorar métricas de grafos não estudadas em sala e aprender mais sobre como aplicar elas em problemas reais.

[1] L. C. Freeman, Centrality in social networks conceptual clarification, Social Networks (1978) 215.

[2] E. W. Dijkstra, A note on two problems in connexion with graphs, NUMERISCHE MATHEMATIK 1 (1959) 269–271.