Trabalho Prático 3

Atualizações Problemáticas

Algoritmos e Estrutura de Dados III

Álvaro Augusto Alves de Carvalho

2015430231

1 Introdução

O objetivo deste trabalho prático é a resolução do problema proposto entitulado “Atualizações Problemáticas”. O problema consiste em determinar, a partir de uma determinada rede de servidores que precisam ser ocasionalmente desconectados para receberem novas atualizações, qual é a melhor forma de desativar estes servidores sem que haja muitos problemas para o tráfego de dados da rede. Para isto, o problema informa que servidores adjacentes não podem ser desconectados ao mesmo tempo, e deve-se determinar um número mínimo de rodadas para que o processo de atualização da rede inteira ocorra. Este problema pode ser diretamente representado por um dos problemas clássicos de Coloração de Grafo, que também é um problema conhecido NP-Completo.

.

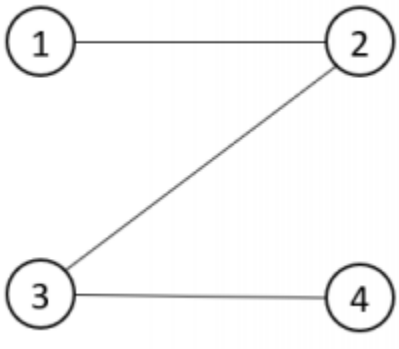

Figura 1: Uma rede de servidores. Esta rede pode ser diretamente representada por um grafo, conforme a própria figura.

Além disso, o problema proposto também deve ser resolvido de duas maneiras diferentes. Inicialmente deve ser implementado um algoritmo de força bruta que testa todas as candidatas a solução e retorna a melhor delas. Este método é no entanto extremamente ineficiente e começa a apresentar problemas quando a rede de servidores torna-se grande. Pensando nisso, o problema também pede para implementar uma solução aplicando uma heurística, que pode não encontrar a melhor solução, mas encontrará uma solução relativamente boa em tempo viável.

Com isto em mente, e utilizando conceitos de programação dinâmica aprendidos na disciplina de Algoritmos e Estrutura de Dados III, chegou-se na conclusão de que a melhor maneira de abordar o problema com heurística seria utilizando um algoritmo Guloso, que procura determinar em tempo de execução qual é a melhor cor a ser utilizada em determinado vértice do Grafo que representa a rede de servidores.

2 Modelagem do Problema e Implementação

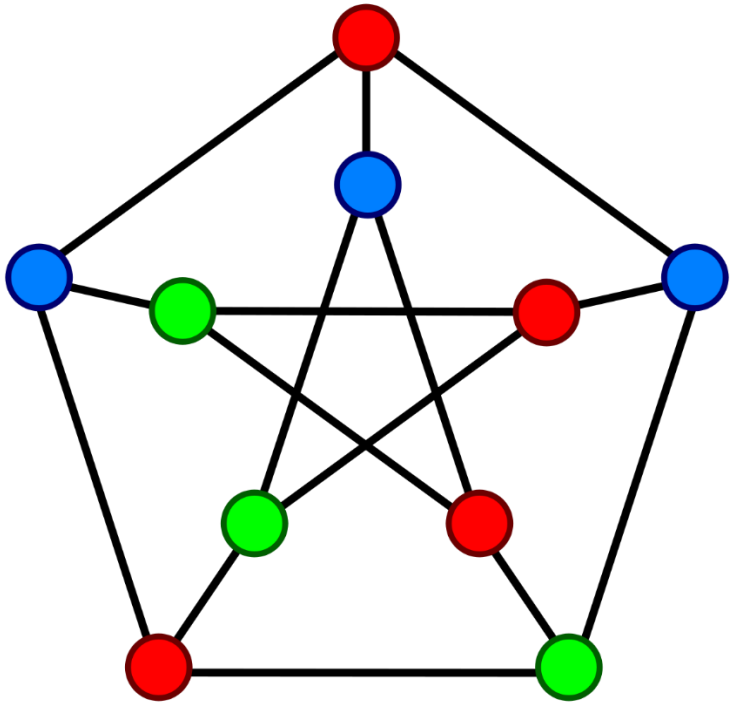

Figura 2: Exemplo clássico de coloração de grafo. O problema de coloração de grafo determina que nenhum vértice adjacente pode ter a mesma cor.

O problema de coloração de Grafo é um problema clássico da computação. Por definição, para colorir um grafo apropriadamente, nenhum vértice adjacente pode ter a mesma cor, sendo assim, um problema que surge é: qual o mínimo número de cores que pode-se utilizar para colorir um grafo G? O menor número diferente de cores que pode ser usado para colorir um grafo G é também chamado de número cromático de G. É importante entender que “cor” é uma noção abstrata, e isso pode ser representado por uma legenda, uma numeração, ou uma tipificação qualquer para um grafo. No problema em questão, a “cor” do grafo seria a rodada em que os servidores devem ser desconectados, e portanto o problema é que nenhum servidor adjacente pode ser desconectado na mesma rodada.

Este problema, para qualquer grafo usando mais de 2 cores diferentes, é um problema NP-Completo, ou seja, não existe um método conhecido para encontrar uma solução para o problema em tempo polinomial. No entanto, existem alguns métodos que podem ser utilizados para aproximar uma solução ótima em tempo viável.

O método provavelmente mais intuitivo e pouco eficiente para encontrar a solução deste problema é a força bruta. Testar cada possível candidato para solução e assim determinar qual é a solução que se encaixa no problema. É fácil de perceber que força bruta é bastante ineficiente, e seu tempo de execução cresce exponencialmente em relação ao tamanho do grafo. Portanto, é necessário também implementar uma segunda solução utilizando uma heurística de programação. A heurística escolhida para resolver este problema foi a utilização de um algoritmo Guloso bem básico.

O algoritmo Guloso implementado funciona da seguinte maneira:

- Colore-se o primeiro vértice com a primeira cor
- Para os vértices restantes:
    - Considerando o vértice atual, colore-se o vértice com a cor de menor número possível que não foi utilizada em nenhum dos outros vértices que estejam adjacentes a ele. Caso todas as outras cores já tiverem sido utilizadas, colore-se com uma nova cor.

```
/* colore o primeiro vertice do grafo com a primeira cor
   neste caso, utilizou-se um vetor de 1 a V para as cores, considerando que
   a cor representa um servidor em uma rede. */
   colors[1] = 1;

   for each u in grafo.vertices {
       for each w in adjacents[u] {
           if (w.color != -1)
               availableColors[colors[w]] = false;
       }
       for each color in availableColors {
           if (availableColors[color] == true) {
               colors[u] = color;
               break;
           }
       }
   }
```

Figura 3: Pseudocódigo do algoritmo Guloso utilizado.

Por fim, o algoritmo acaba com um vetor chamado **result** contendo todas as cores utilizadas nos vértices do grafo, além do número máximo de cores (ou rodadas) utilizadas. Então, as respostas são impressas nos arquivos de saída como solicitado, e o problema está resolvido.

3 Análise de Complexidade (teórica)

3.1 Análise de Complexidade de Tempo

Podemos realizar a análise de complexidade temporal baseando nas complexidades de cada algoritmo apresentado.

- O algoritmo utilizado para ler as entradas e construir o grafo tem complexidade fixa em O(|V| + E), onde V é o número de vértices e E é o número de arestas. O grafo foi implementado utilizando uma lista de adjacências, exatamente igual ao grafo implementado no Trabalho Prático 0 desta disciplina.
- O algoritmo guloso utilizado possui uma complexidade de O(V^2 + E), no pior caso.
- Os outros algoritmos implementados referem-se à destruição do grafo, impressão da resposta no arquivo e outros utilitários. Todos com complexidade de O(|V|), no máximo.

Por fim, temos que a complexidade do programa em si, considerando o algoritmo de maior complexidade, é:

$$O(V^2 + E)$$

3.2 Análise de Complexidade Espacial

Para a análise de complexidade espacial, vamos abordar a estrutura de dados do grafo em si, assim como os algoritmos utilizados.

- A estrutura utilizada é uma lista de adjacências, que possui uma complexidade espacial de O(|V| + E).
- O algoritmo Guloso percorre todo o grafo, portanto com uma complexidade espacial O(|V|). Não é necessário alocar nenhuma memória além do grafo.
- Os demais algoritmos também não alocam memória além do grafo em si, portanto não passam em complexidade dos algoritmos citados acima.

Por fim, temos uma complexidade espacial do programa, considerando as estruturas e algoritmos abordados: $O(|V| + E)$

4 Análise dos Experimentos

Para a análise experimental do trabalho, foi utilizado o comando *time* (para análise de complexidade temporal), e o *valgrind* (para análise do consumo de memória do programa). Foram utilizados os testes fornecidos pelo moodle da disciplina na análise. Todos os testes foram realizados em uma máquina virtual Ubuntu através do Oracle VM VirtualBox, com 4gb de RAM alocados. A máquina física dispõe de um processador i7-4700MQ @2.40GHz, com 16gb de RAM.

4.1 Análise Temporal

Para a análise temporal, cada teste foi executado 10 vezes utilizando o comando *time*. Ex.: time ./tp2 < teste_i.txt > output_i.txt

Os testes utilizados foram os dados no arquivo TestesHeuristica.zip

Os valores obtidos seguem na tabela 1.

Teste / Execucoes	1	2	3	4	5	6	7	8	9	10
1	0.002	0.002	0.003	0.003	0.003	0.002	0.002	0.002	0.002	0.002
2	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002
3	0.002	0.002	0.002	0.002	0.002	0.002	0.003	0.002	0.003	0.002
4	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.004	0.003
5	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.003
6	0.007	0.005	0.006	0.004	0.005	0.004	0.005	0.005	0.006	0.005
7	0.005	0.005	0.005	0.005	0.005	0.005	0.005	0.005	0.005	0.006
8	0.009	0.006	0.005	0.005	0.006	0.007	0.005	0.005	0.006	0.006
9	0.01	0.006	0.007	0.01	0.007	0.007	0.007	0.007	0.006	0.006
10	0.013	0.007	0.007	0.007	0.009	0.013	0.008	0.008	0.008	0.008

Tabela 1: Valores de tempo de execução obtido por teste, em um total de 10 execuções. Valores expressos em segundos.

A partir da média dos valores obtidos (tabela 2), e do número de vértices no grafo em cada teste, o gráfico da figura 3 foi construído.

Teste	Tempo Médio	V
1	0.0023	11
2	0.002	23
3	0.0022	47
4	0.0031	191
5	0.003	81
6	0.0052	169
7	0.0051	451
8	0.006	450
9	0.0073	621
10	0.0088	864

Tabela 2: Tempo médio de execução x número de vértices, por teste

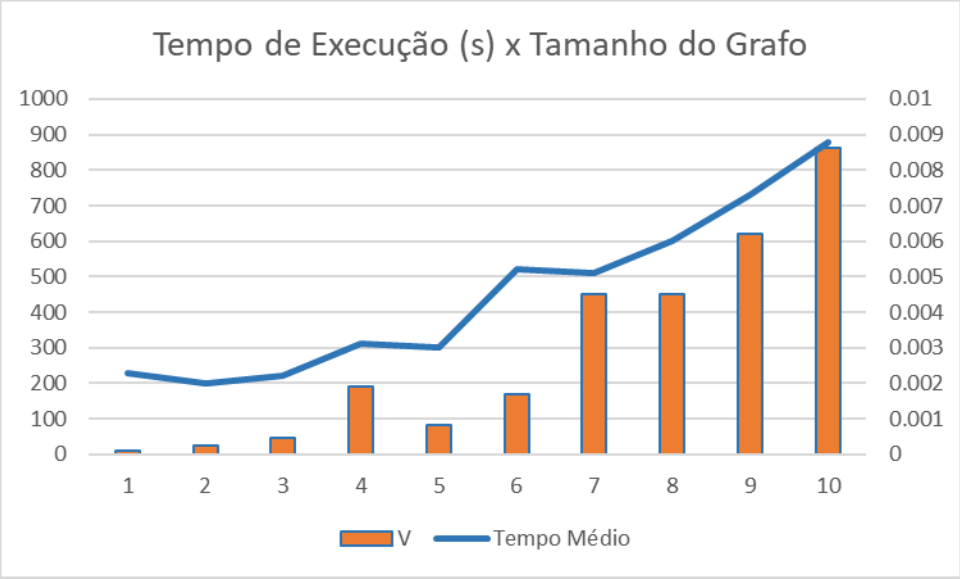

Figura 4: Gráfico de tempo de execução (em segundos) x Número de vértices no grafo, por teste

Podemos ver, principalmente nos últimos testes, que a complexidade do programa depende linearmente do número de vértices no grafo, mas o número de arestas ou relações entre vértices possue um grande peso também.

4.2 Análise Espacial

Para a análise espacial foi utilizado o Valgrind para cada caso de teste apresentado, e foi criada uma tabela com os valores de memória utilizados em cada execução em relação ao número de vértices. Posteriormente, foi criada uma representação gráfica dos dados. Os testes utilizados foram os mesmos (TestesHeuristica.zip)

Testes	kBytes	V
1	14.144	11
2	15.872	23
3	21.344	47
4	90.464	191
5	81.648	81
6	227.76	169
7	295.136	451
8	329.24	450
9	465.392	621
10	618.952	864

Tabela 3: Espaço de memória utilizado (kBytes) x Número de vértices no grafo, por teste

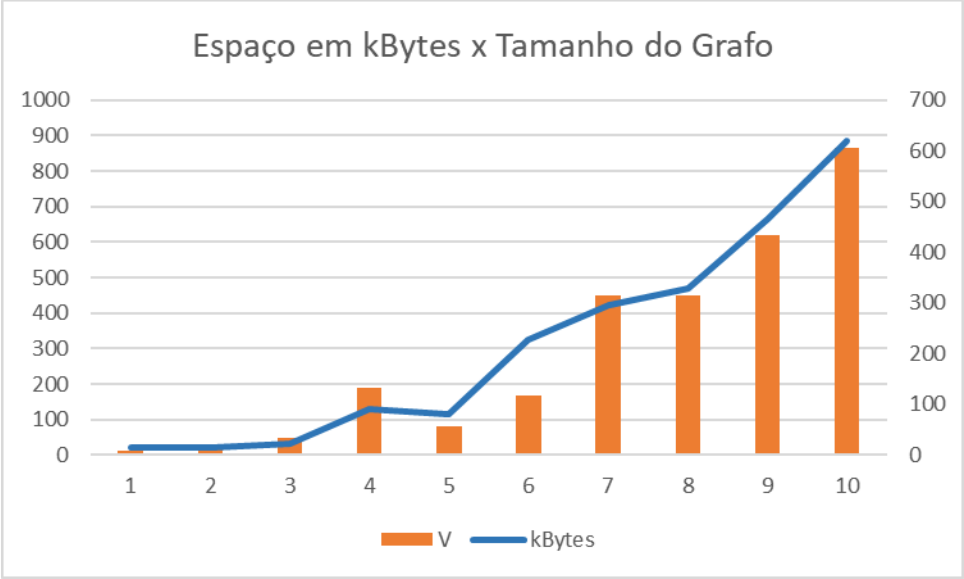

Figura 4: Gráfico de consumo de memória (kBytes) x tamanho do grafo, por teste

Através dos dados apresentados, podemos verificar que neste caso o consumo de memória é diretamente relacionado ao número de vértices no grafo.

5 Conclusão

O trabalho proposto teve como objetivo familiarizar o aluno com conceitos de problemas um pouco mais práticos de programação, além de aprofundar os conceitos de grafos, vistos na disciplina de Algoritmos e Estrutura de Dados III. Para a resolução do problema, foi implementada uma solução relativamente simples levando em consideração a popularidade do problema proposto, um clássico da teoria da computação, com algumas simples implementações para processamento dos dados, de forma a se adequar ao problema apresentado.

Esta solução aqui apresentada, apesar de não ser a mais eficiente, acredito que teve uma boa relação de “complexidade de implementação x eficiência do código”. Acredito que este foi um problema bastante direto, e a abordagem utilizada foi adequada à realidade do aluno, que ainda possui algumas limitações para implementação de algoritmos mais sofisticados, principalmente considerando o tempo disponível para realização do trabalho junto com as dificuldades encontradas em outras disciplinas durante o curso. As análises experimentais e teóricas, a princípio, também parecem condizentes com o problema proposto e a solução implementada.

É importante notar que o algoritmo de força bruta não foi implementado neste trabalho, devido à dificuldades encontradas somado com a falta de tempo. Esta documentação portanto está sendo entregue sem considerar o algoritmo de força bruta para experimentos e implementação, e caso ocorra alguma atualização neste sentido em tempo hábil, a submissão deste trabalho também será atualizada.