Documentação TP3 - AEDS 3

Atualizações Problemáticas

Luciano Otoni Milen [2012079754]

1. Introdução

Neste trabalho prático, devemos implementar um sistema de gerenciamento de servidores de um aplicativo de música. A ideia é que os servidores estão conectados entre si e há a necessidade de atualizá-los de tempos em tempos. O problema é que para atualizar um servidor é necessário que seu tráfego seja direcionado para seus vizinhos, logo não é possível atualizar todos ao mesmo tempo. Devemos, então, implementar um algoritmo de alocação de servidores em *rounds*, onde cada servidor é atualizado em determinado *round*. Esta é uma adaptação de um problema clássico na computação, a coloração de grafos. É um requisito que o sistema seja implementado em 2 versões: uma utilizando força bruta e a outra alguma heurística para aproximar da solução em um tempo hábil.

2. Modelagem do problema

A forma mais simples de modelar este problema é utilizando a estrutura de grafos, onde cada servidor é um vértice e as conexões são representadas pelas arestas. Devemos, então, encontrar uma forma de percorrer o grafo, identificando os servidores que serão atualizados em determinado *round*.

Este problema é muito semelhante à coloração de grafos, onde a ideia é colorir os vértices em que um dado vértice seja colorido com uma cor diferente de seus vizinhos. A questão é, quantas cores são necessárias para colorir o grafo? Sabe-se que este é um problema na classe NP-Completo. Pertencer a esta classe significa que o problema não possui solução polinomial determinístico, ou seja, todos os algoritmos para resolvê-lo não são eficientes. Pode-se reduzir o problema da coloração no SAT, provando que o problema de fato faz parte da classe NP-Completo.

Sabendo das complicações associadas à resolução do problema, devemos implementar a resolução de duas formas: força bruta e heurística. A primeira, como já se sabe, é muito ineficiente, de classe exponencial. A ideia é testar todas as combinações de cores para os vértices e identificar aquela que obedece às restrições do problema. A heurística por sua vez é um tipo de aproximação. Basicamente tenta-se de forma inteligente estimar quantas cores serão necessárias a partir da estrutura do grafo.

É trivial perceber a semelhança dos dois problemas: enquanto um se trata de cores dos vértices, o outro é sobre diferentes *rounds* de atualização.

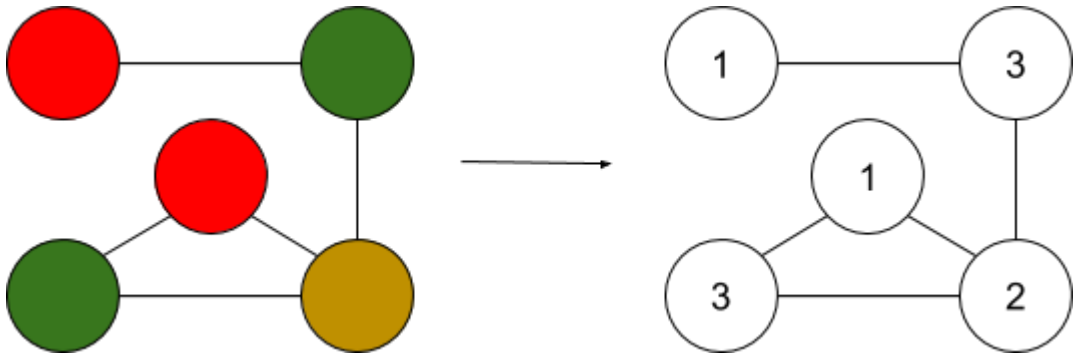

Figura. Transformação do problema de coloração de grafos no trabalho prático

Força Bruta

O algoritmo de força bruta é uma forma mais direta de resolver o problema. A cada atribuição dos *rounds* de atualização a configuração atual é verificada. Pode-se considerar que esta é uma melhoria à força bruta, através de *backtracking*. Invés de gerar todas as possibilidades e depois verificar uma por uma qual respeita as restrições do problema, pode-se verificar as soluções em tempo de atribuição dos rounds para economizar o considerável consumo de recursos do algoritmo de força bruta. Com isso, mais testes puderam ser realizados utilizando o código implementado. A solução via *backtracking* é bem mais eficiente, conforme podemos observar. Mesmo assim, possui alta complexidade, conforme veremos na próxima sessão.

Heurística

Conforme já visto, o problema de coloração de grafos é NP-Completo. Isto significa que todas as soluções exatas possuem um tempo ruim de execução, apresentando alta complexidade. Uma forma de driblar esta condição é através de uma implementação de uma heurística para o problema.

A forma mais simples de solucionar a alocação de *rounds* é através da implementação gulosa. É importante notar que não é uma solução ótima, já que a complexidade de tempo para calculá-la é altíssima, mas chega-se perto do valor ideal. O algoritmo guloso consegue resolver o problema garantindo um número máximo de *rounds* atribuídos, onde tal número é o maior grau dos vértices existentes.

A verificação para garantir que a solução é válida é bem simples. Basicamente escolhe o número mínimo de *rounds* tal que os servidores adjacentes possuam *rounds* escolhidos diferentes do definido para o servidor atual. Ou seja,

garante-se que vértices adjacentes não estejam no mesmo *round*. A definição do algoritmo pode ser vista a seguir.

Algoritmo guloso

```
para o primeiro servidor, atualize-o no round 1
escolha o round 1 para todos os próximos servidores
        | comece a caminhar no grafo:
        |       para o vértice atual i:
        |       |     verifique se os adjacentes possuem mesma cor
        |       |     se possuírem,
        |       |     |     escolha uma cor diferente para o vértice i
        |       |     senão,
        |       |     \     continue com o valor escolhido anterior
```

3. Análise de complexidade assintótica

Tempo

Força Bruta

As funções mais significativas são a `percorre`, para percorrer o grafo e a `geraValorRounds`, que a partir do vértice atual, verifica os adjacentes para validar o *round* escolhido.

Considerando que para cada vértice o grafo verifica todas as arestas e vértices adjacentes, temos *O*(nVertices * $nArestas^{nVertices}$), já que devemos sempre verificar cada vértice de cada aresta. Conforme podemos ver, o algoritmo é muito custoso. Mesmo assim, utilizando backtracking podemos obter um resultado melhor que o esperado.

Heurística

No pior caso, ou seja, todos os vértices estão conectados e assim precisa-se de um *round* por vértice, a heurística escolhida possui complexidade alta. Se pensarmos que todos os vértices deveriam ser verificados e para isto devemos percorrer todas as arestas, temos *O*(nVertices * nVertices + nArestas). Por mais que não possamos garantir o melhor tempo, já que se trata de uma heurística e este é um problema NP-Completo, o caso médio é bem atingido, já que o algoritmo inicia com todos os servidores setados no primeiro *round* e altera o valor definido somente se as restrições de mesmo *round* para vértices adjacentes seja descumprida.

Espaço

Força Bruta

A complexidade de espaço está mais relacionada à estrutura utilizada. Para o algoritmo de força bruta o grafo é armazenado numa matriz de adjacências, de tamanho nVertices * nVertices, pois assim é possível relacionar as arestas possíveis.

O *backtracking* se mostrou mais fácil de implementar utilizando a matriz de adjacências. Desta forma, a complexidade de espaço mais significativa do problema é *O*(nVertices * nVertices).

Heurística

A estrutura escolhida para a heurística foi diferente da força bruta. Para o algoritmo guloso não havia maior dificuldade de fazê-lo utilizando uma lista de adjacências, que no fim das contas é mais econômica que a matriz vista na força bruta. Para cada vértice, tem-se um vetor de nArestas posições. Ou seja, a complexidade de espaço está relacionada ao número de vértices pelo número de arestas, ou ainda, *O*(nVertices * nArestas).

4. Análise experimental

Para fazer a análise de tempo gasto pelo programa foi utilizado o Gnomon, software desenvolvido sob licença do MIT pelo usuário *paypal* do GitHub. Basta incluir " | gnomon" após o executável no terminal para contar o tempo gasto. A análise de memória foi feita com o Valgrind, que mostra a quantidade de *bytes* utilizado pelo programa. O programa foi elaborado e testado em uma máquina Ubuntu Linux 18 Intel® CoreTM i7-3610QM CPU @ 2.30GHz × 8 com 8gb de memória RAM. A tabela a seguir associa alguns testes feitos com o gasto de memória e de tempo.

Teste	Vértices	Arestas	Tempo (s)	Memória (KB)
t1fb	3	2	0.0056	13,488
t1h			0.0058	13,576
t2fb	3	3	0.0060	13,488
t2h			0.0059	13,608
t3fb	4	4	0.0059	13,532
t3h			0.0058	13,672

t4fb	191	2360	0.0089	162,384
t4h			0.0069	95,048
t5fb	81	2112	0.0129	40,944
t5h			0.0060	83,592
t6fb	169	6656	0.0211	130,352
t6h			0.0090	231,816

Os testes correspondem aos testes propostos pelos monitores. Foram feitos cerca de 10 execuções de cada teste para obter uma média dos valores de tempo e memória consumidos pelo programa. O sufixo “fb” significa que o teste é utilizando força bruta e “h” para heurística.

5. Conclusão

Comparado com os trabalhos anteriores, este foi de maior complexidade, já que foi exigida a implementação de dois algoritmos para resolver o mesmo problema. O primeiro, de força bruta, não ficou muito claro como implementar, logo decidi invés de verificar todas as soluções no final, avaliá-las conforme fossem sendo construídas. Desta forma, consegui implementar um algoritmo ainda mais eficiente que o esperado, utilizando *backtracking*. Para esta parte, o código roda perfeitamente. Não há sobra de memória e o tempo gasto é bem razoável. A heurística é muito mais elegante. Por meio de uma estratégia gulosa é possível aproximar bastante do resultado ótimo. O algoritmo funciona muito bem, o único problema está na hora de liberar a memória alocada para o grafo. Utilizei a mesma estrutura do TP0, uma lista de adjacências. Sobraram alguns poucos blocos de memória alocada no final, que não consegui liberar.

O trabalho foi bem difícil de ser concluído. Ainda que entregue com atraso, acredito que foi bem executado, e o aprendizado muito relevante: entender as diferenças entre força bruta e heurística e avaliar como o impacto pode ser grande quando devemos resolver problemas que estão na classe NP-Completo. O uso de servidores em atualização é claramente uma analogia às cores no problema de coloração de grafos.

6. Referências

[1] Design And Analysis Of Algorithms, Puntambekar, A.A.

[2] https://en.wikipedia.org/wiki/Graph_coloring

[3] https://www.geeksforgeeks.org

[4] Slides do professor Nívio e Introduction to Algorithms, Thomas Cormen