Trabalho Prático III - Algoritmos I

Gabriel Victor Carvalho Rocha

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG – Brasil

gabrielcarvalho@dcc.ufmg.br - 2018054907

1. Introdução

O problema abordado foi a implementação de um algoritmo que resolvesse o puzzle Sudoku utilizando uma heurística de grafos. Como esse problema é NP-Completo, é necessário então a utilização de um algoritmo Guloso que dará uma solução aproximada, ou seja, nem sempre haverá solução.

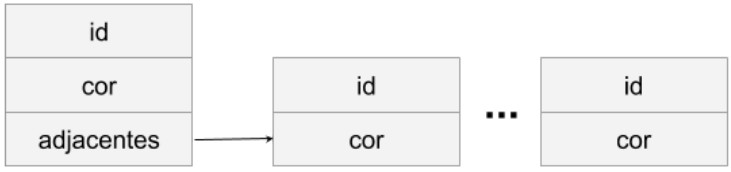
Entretanto, há diversas heurísticas que podem ser utilizadas, mas, ao longo dos tópicos seguintes será explicado o uso para resolvê-lo utilizando a coloração de grafos, onde temos que colorir cada célula de modo que não conflite com seus adjacentes.

2. Implementação

Todo o programa foi desenvolvido na linguagem C, utilizando o compilador GCC (GNU Compiler Collection).

2.1. Estruturas de dados

A estrutura de dados utilizada foi uma matriz NxN (N = dimensão do Sudoku) de células que possuem atributos id (número do vértice), cor e um vetor de struct adjacente com tamanho M (M = número de vértices adjacentes). O struct adjacente possui como atributo id (número do vértice) e sua cor. O esquema de uma única célula pode ser visto abaixo:



2.2. Heurística

A heurística adotada foi a transformação do Sudoku em uma estrutura de grafo e utilizar um algoritmo de coloração de grafos para gerar a solução. Como cada vértice possuirá um número fixo de vizinhos e não precisamos alterar ou remover vértices, foi adotada então a estrutura de vetor para armazenar os vértices adjacentes em cada célula da matriz.

Essa heurística foi escolhida por ser bastante intuitiva, pois adota uma maneira de escolher um vértice de forma com que nós resolvemos na maioria das vezes, ou seja, ver

aquele vértice que possui apenas uma opção para se colocar uma cor.

O algoritmo então consiste em: Dado o Sudoku, é colocado em cada vetor de cada célula seus vértices adjacentes, o maior detalhe que utilizaremos é o grau que é definido como o número de cores dos vértices adjacentes diferentes e diferente de 0, ou seja, quantas cores diferentes estão incidentes a ele. Isso é a base do algoritmo, pois iremos começar pegando o vértice que possui o maior grau e que possua cor = 0, que com sorte em muitas vezes é o que há apenas uma opção de cor, entretanto, em casos em que falta mais de uma cor, é atribuído a ele a primeira cor disponível, o que pode acarretar em casos sem solução, retornando em algum momento a cor 0, ou seja, o problema não terá solução nesse algoritmo.

Vejamos um exemplo: Dado o Sudoku 4 x 4 com quadrantes 2 x 2. O maior grau de um vértice é 3 (dimensão - 1). Considerando linha e coluna começando de 1.

$$\begin{bmatrix} 0 & 3 & 0 & 4 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 3 & 0 & 2 & 0 \end{bmatrix}$$

Para facilitar a execução, iremos pegar logo o primeiro vértice que possui maior grau.

O vértice[1][3] possui como adjacente 2, 3 e 4 (grau = 3), logo, podemos colori-lo com 1.

$$\begin{bmatrix} 0 & 3 & 1* & 4 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 3 & 0 & 2 & 0 \end{bmatrix}$$

O vértice[1][1] possui os adjacentes 1, 3 e 4 (grau = 3), logo, podemos colori-lo com 2.

$$\begin{bmatrix} 2* & 3 & 1 & 4 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 3 & 0 & 2 & 0 \end{bmatrix}$$

O vértice[2][3] possui os adjacentes 1, 2 e 4 (grau = 3), logo, podemos colori-lo com 3.

$$\begin{bmatrix} 2 & 3 & 1 & 4 \\ 0 & 0 & 3* & 0 \\ 0 & 0 & 0 & 0 \\ 3 & 0 & 2 & 0 \end{bmatrix}$$

O vértice[2][4] possui os adjacentes 1, 3 e 4 (grau = 3), logo, podemos colori-lo com 2.

$$\begin{bmatrix} 2 & 3 & 1 & 4 \\ 0 & 0 & 3 & 2* \\ 0 & 0 & 0 & 0 \\ 3 & 0 & 2 & 0 \end{bmatrix}$$

O vértice[3][3] possui os adjacentes 1, 2 e 3 (grau = 3), logo, podemos colori-lo com 4.

$$\begin{bmatrix} 2 & 3 & 1 & 4 \\ 0 & 0 & 3 & 2 \\ 0 & 0 & 4* & 0 \\ 3 & 0 & 2 & 0 \end{bmatrix}$$

O vértice[3][1] possui os adjacentes 2, 3 e 4 (grau = 3), logo, podemos colori-lo com 1.

$$\begin{bmatrix} 2 & 3 & 1 & 4 \\ 0 & 0 & 3 & 2 \\ 1* & 0 & 4 & 0 \\ 3 & 0 & 2 & 0 \end{bmatrix}$$

O vértice [2][1] possui os adjacentes 1, 2 e 3 (grau = 3), logo, podemos colori-lo com 4.

$$\begin{bmatrix} 2 & 3 & 1 & 4 \\ 4* & 0 & 3 & 2 \\ 1 & 0 & 4 & 0 \\ 3 & 0 & 2 & 0 \end{bmatrix}$$

O vértice[2][2] possui os adjacentes 2, 3 e 4 (grau = 3), logo, podemos colori-lo com 1.

$$\begin{bmatrix} 2 & 3 & 1 & 4 \\ 4 & 1* & 3 & 2 \\ 1 & 0 & 4 & 0 \\ 3 & 0 & 2 & 0 \end{bmatrix}$$

O vértice[3][2] possui os adjacentes 1, 3 e 4 (grau = 3), logo, podemos colori-lo com 2.

$$\begin{bmatrix} 2 & 3 & 1 & 4 \\ 4 & 1 & 3 & 2 \\ 1 & 2* & 4 & 0 \\ 3 & 0 & 2 & 0 \end{bmatrix}$$

O vértice[3][4] possui os adjacentes 1, 2 e 4 (grau = 3), logo, podemos colori-lo com 3.

$$\begin{bmatrix} 2 & 3 & 1 & 4 \\ 4 & 1 & 3 & 2 \\ 1 & 2 & 4 & 3* \\ 3 & 0 & 2 & 0 \end{bmatrix}$$

O vértice [4][2] possui os adjacentes 1, 2 e 3 (grau = 3), logo, podemos colori-lo com 4.

$$\begin{bmatrix} 2 & 3 & 1 & 4 \\ 4 & 1 & 3 & 2 \\ 1 & 2 & 4 & 3 \\ 3 & 4* & 2 & 0 \end{bmatrix}$$

E por fim, o vértice[4][4] possui os adjacentes 2, 3 e 4 (grau = 3), logo, podemos colori-lo com 1.

$$\begin{bmatrix} 2 & 3 & 1 & 4 \\ 4 & 1 & 3 & 2 \\ 1 & 2 & 4 & 3 \\ 3 & 4 & 2 & 1* \end{bmatrix}$$

Solução final:

$$\begin{bmatrix} 2 & 3 & 1 & 4 \\ 4 & 1 & 3 & 2 \\ 1 & 2 & 4 & 3 \\ 3 & 4 & 2 & 1 \end{bmatrix}$$

Claramente, é um exemplo simples e que pode ser resolvido mais facilmente, quando aumentamos o tamanho do Sudoku, nem sempre teremos apenas uma escolha (vértices com grau máximo), então precisaremos escolher alguma disponível (que foi determinado no algoritmo como a primeira que estiver disponível) e isso pode fazer com que a solução não possa ser achada.

3. Instruções de compilação e execução

A compilação é feita através de um arquivo Makefile, digitando "*make*" em um terminal Linux é gerado o executável "*tp*3" . Já a execução é realizada da seguinte forma:

*./tp*3 *nomeArquivo.txt*

4. Análise de complexidade

Na complexidade em relação ao tempo, iremos analisar o código principal *main* e assim percorreremos todas as funções na ordem em que aparecerem.

Primeiramente, definindo N como a dimensão do Sudoku e M como o número de vértices adjacentes (dado pela fórmula: $2 \star (N-1) + (LQ * CQ) - LQ - CQ + 1$, sendo LQ e CQ a linha e coluna do quadrante, respectivamente), após ler os dados iniciais, alocamos a mémoria da matriz na função alocaMatriz com custo $O(N^2)$ de tempo e $O(N^2M)$ de espaço. Então lemos os dados do Sudoku do arquivo de entrada com custo $O(N^2)$ de tempo.

Em seguida entramos na função colocaAdjacentes, que irá transformar o Sudoku em uma estrutura de grafo. Iniciamos ela chamando a função zeraGrausCores que possui custo de tempo $O(N^2M)$ e logo após, iremos adicionar todas relações de adjacências com custo $O(N^4)$ e dentro disso, há a chamada a função verificaCorExiste, que possui custo $O(M)$. Logo, possui custo de tempo total $O(N^4M)$.

Agora iremos utilizar a função coloreGrafo que gera a solução do Sudoku, aqui iremos rodar no loop principal N^2 vezes, mais adentro, iremos checar o maior grau da matriz com os outros 2 for's com custo $O(N^2)$, sendo então essa checagem $O(N^4)$, mas fora desses 2 for's também checamos a cor pela função verificaCor com custo de tempo no pior caso $O(NM)$, entretanto o peso maior na complexidade de tempo será chamada da função colocaAdjacentes para atualizar toda alteração feita, que como explicado anteriormente, tem custo $O(N^4M)$ e está dentro do primeiro for com N^2 iteração possuindo então custo $O(N^6M)$. A função de imprimir possui custo $O(N^2)$, assim esta função fica com custo final de tempo $O(N^6M)$.

No fim, liberamos a memória da matriz com custo de tempo $O(N^2)$.

Somando todas complexidades das funções na ordem em que aparecem no main: $O(N^2 + N^2 + N^4M + N^6M + N^2)$ em relação ao tempo e $O(N^2M)$ em relação ao espaço.

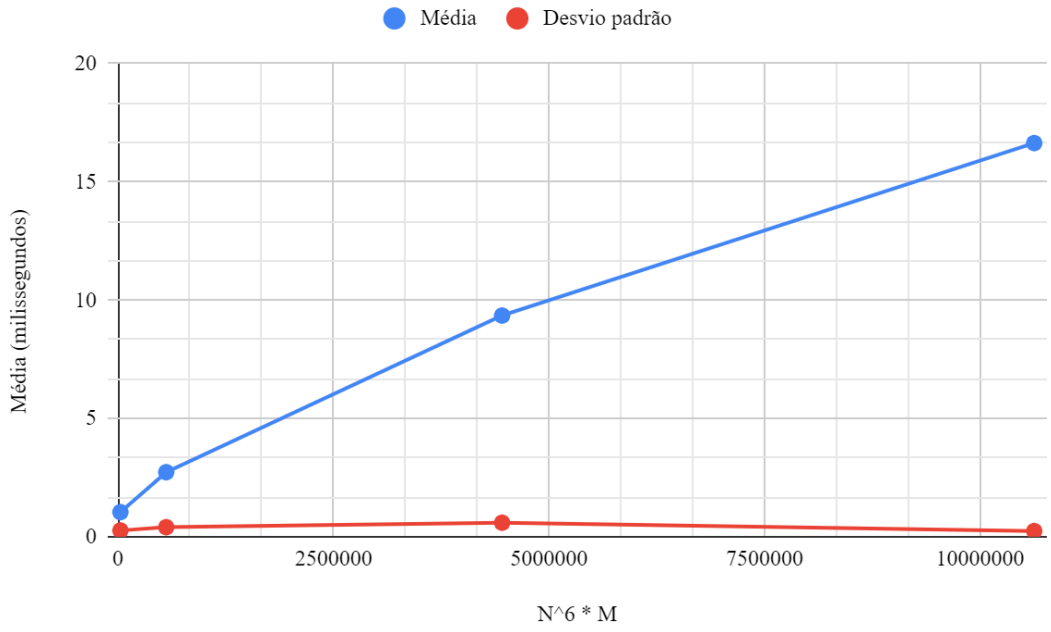
Portanto, a complexidade total do código é $O(N^6M)$ em relação ao tempo e $O(N^2M)$ em relação ao espaço.

5. Análise experimental

O pior caso do algoritmo ocorre quando de fato ele encontrará uma solução, pois isso significa que ele terá q percorrer por todas as células da matriz, fazendo sempre todas as checagens e atualizações. Quando não há solução, isso significa que antes que o algoritmo percorra toda a matriz ocorrerá um retorno 0 para a cor, retornando sem solução.

Já a memória utilizada, dado a dimensão N, independente se há ou não solução e o quanto o Sudoku está preenchido, o algoritmo sempre irá utilizar a mesma quantidade, já que aloca inicialmente uma matriz e um vetor (N^2M) e isso não se altera ao longo do código.

Para a realização do experimento, foram executados 20 vezes cada tamanho do Sudoku (4x4, 6x6, 8x8 e 9x9) que havia solução no algoritmo, guardando sua média e desvio padrão em milissegundos, gerando então o gráfico abaixo:



Como pode ser analisado, é possível perceber que a média está linear em relação ao número N^6M, ou seja, podemos certificar que a complexidade é $O(N^6M)$ como demonstrado na Análise de Complexidade.

Nos testes realizados utilizando o dataset fornecido, a heurística adotada obteve 100% de acerto (5 em 5) nos tamanhos 4x4 e 6x6. Já nos casos 8x8 e 9x9 obteve 60% de acerto (3 em 5). Logo, o Sudoku obteve um melhor resultado para os casos menores (4x4 e 6x6). Além disso, quanto mais o sudoku estiver inicialmente preenchido haverá mais chances do algoritmo gerar solução, já que esses preenchimentos podem acarretar em vértices com maiores graus.

6. Conclusão

Dessa forma, a partir da implementação da heurística de coloração de grafo podemos perceber que para Sudokus que possuem dimensões menores, temos uma taxa de acerto alta, mas aumentando a dimensão, começam a surgir instâncias nas quais não é possível gerar a solução. Isso ocorre, pois, à medida que a dimensão cresce, temos cada vez mais números de cores e isso por consequência faz com que fique cada vez mais difícil ter vértices com grau máximo $(N-1)$ pelo preenchimento inicial, ou seja, não há como escolher uma cor com 100% de chance de acerto, entrando então em uma questão probabilística para se acertar uma cor.

7. Bibliografia

KLEINBERG, Jon.; TARDOS, Éva. **Algorithm Design**.

ELISSA, Mostafa. **A Sudoku Solver using Graph Coloring.** Disponível em:<https://www.codeproject.com/Articles/801268/A-Sudoku-Solver-using-Graph-Coloring>.