Trabalho Prático 3 - Algoritmos 1

Guilherme de Abreu Lima Buitrago Miranda - 2018054788

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

guilhermemiranda@dcc.ufmg.br

1. Introdução

O trabalho prático 3 da disciplina de Algoritmos 1 consiste em implementar uma heurística para resolver o $puzzle\ Sudoku$. O mesmo é composto por diversos quadrantes, linhas e colunas, além de algumas células preenchidas. O objetivo do quebra-cabeça é preencher as células faltantes de forma a não repetir números em cada linha, coluna ou subquadrante.

Dessa forma, foi desenvolvido um algoritmo que preenche as posições vazias do $Sudoku$ com o número ausente quando possível. Caso contrário, ou seja, se existem duas ou mais opções para cada célula faltante do $puzzle$, o mesmo busca adivinhar a opção para uma delas e continua a tentar resolver o desafio.

Ademais, tal algoritmo faz uso de um grafo para representar a instância do problema e utiliza técnicas de coloração de grafos para escolher o melhor vértice a ser preenchido a depender da situação.

2. Implementação

2.1. Entrada e saída de dados

A entrada esperada é feita por meio de um arquivo de texto, passado como argumento na execução do programa. Esse arquivo deve ter a seguinte estrutura: na primeira linha, três inteiros N, I e J, representando, respectivamente, o tamanho (N x N) da tabela do $Sudoku$, a quantidade de colunas de cada quadrante e a quantidade de linhas dos mesmos.

Nas próximas N linhas serão recebidos N números, representando a instância N x N do $Sudoku$ em questão.

A saída gerada pelo programa é impressa no $stdout$ e sua primeira linha diz respeito ao sucesso ou não da heurística. Caso tenha encontrado uma solução final válida, o algoritmo imprime 'solução' e, nas próximas N linhas imprime N números referentes ao tabuleiro N x N solucionado. Caso contrário, é impressa a linha 'sem solução' e, nas seguintes N linhas são impressos os N números referentes ao tabuleiro incompleto gerado pelo algoritmo.

2.2. Estruturas de dados

Para a modelagem do problema, foi criada uma classe $Sudoku$, que implementa o grafo a ser processado e tem como atributos o número de células (N x N) no tabuleiro e o número de linhas e colunas por quadrante.

Além disso, conta também com dois arrays de tamanho N x N. O primeiro conta com um $vector$ em cada posição, representando a lista de adjacências de cada célula (nó

do grafo), ou seja, a lista de células que a célula raíz é ligada (células distintas que estão na mesma linha, coluna ou quadrante). Já o segundo tem, em cada posição, um mapa, que representa os números que aquela célula não pode se tornar, sendo esse preenchido pelo algoritmo a medida que o tabuleiro é povoado. A escolha pelo mapa se deve, principalmente, por seu acesso a elementos ter complexidade menor que no caso de um $vector$, conforme explicitado posteriormente na seção de Análise de Complexidade.

Por fim, a classe também conta com um array de tamanho N x N representando o tabuleiro do $puzzle$ que será impresso após o processamento.

2.3. Algoritmo

Conforme supracitado, o algoritmo desenvolvido transforma o $puzzle\ Sudoku$ no problema de coloração de Grafos. Assim, logo na construção do problema, são ligadas as células (nós do grafo) às outras de mesma coluna, linha ou quadrante.

Posteriormente, para cada célula lida, além de ser inserida na matriz que descreve o tabuleiro, se a mesma tiver um valor diferente de zero, ou seja, for uma célula já preenchida do $puzzle$, o método $addValue$ adiciona, nos elementos os quais aquela célula está ligada, seu número (análogo à cor, no problema de coloração) no mapa $cantBe$, representando que aquelas células não podem assumir esse valor.

Após a leitura de todas as células é invocado o método $solve$, contendo a heurística polinomial que tenta resolver o quebra-cabeça. Nela, enquanto o $Sudoku$ não é resolvido (ou chega a um ponto em que falha pois nenhuma célula vazia pode mais ser preenchida sem que fira as regras do jogo), é buscada a célula com menor número de opções possíveis a se preencher e, caso haja apenas uma opção, a mesma é escolhida. Caso contrário, é escolhida a opção de menor grau, ou seja, se temos 1 e 5 entre as opções, 1 será o número a ser escolhido para o preenchimento.

A posteriori, cada célula ligada àquela preenchida anteriormente tem seu mapa $cantBe$ modificado, já que essas não podem mais ter o valor mais recentemente atribuído. Além disso, é feita uma checagem no tabuleiro do $Sudoku$ verificando se o mesmo está totalmente preenchido. Caso seja verdadeiro, o método de solução se encerra e a palavra 'solução' é impressa no $main$. Caso contrário, o método continua a tentar resolver o quebra-cabeças. Não obstante, caso a heurística atinja um ponto em que mais nenhuma célula pode ser preenchida mas ainda existem posições vazias no tabuleiro, o $loop$ é encerrado, o método de solução retorna falso e é impresso 'sem solução'. Por fim, é também impresso o tabuleiro do $puzzle$ da maneira em que foi finalizado o processamento.

2.4. Instruções de compilação e execução

O compilador utilizado para o desenvolvimento do programa foi o g++/$gcc\ 8.3.0$, usando a flag $-std = c$++11 numa máquina com a versão 4.19.49 do Kernel Linux. Para executar o programa, ao entrar em seu diretório, no terminal, deve ser digitado o comando $make$. Com isso, será gerado um binário $tp3$ e o algoritmo pode ser executado passando um arquivo de texto como argumento, por exemplo: $./tp3\ entrada.txt$

3. Análise de Complexidade

3.1. Complexidade de Tempo

Para a heurística implementada, o primeiro passo é, no momento de se construir o $Sudoku$, a criação das conexões entre linhas, colunas e quadrantes. Tal criação tem complexidade $O(N^3)$, pois, para cada um dos vértices, o mesmo tem $n-1$ conexões na linha, $n-1$ na coluna e cerca de $n-1$ no quadrante, já que alguns foram previamente conectados nos passos anteriores. Assim, dado que isso é feito para cada um dos vértices, temos uma complexidade de $O(N^2) * O(N) = O(N^3)$.

Posteriormente, para cada uma das N x N células lidas no arquivo, é feita a inserção de seu valor no mapa $cantBe$ daquelas com que faz ligação se seu valor for diferente de 0. Assim, esse procedimento tem complexidade $O(N^2logN)$, já que, para manter a ordenação, a inserção em mapas tem complexidade $O(logN)$. Dessa forma, $O(N^2) * O(logN) = O(N^2logN)$.

No método que efetivamente resolve o $Sudoku$, o primeiro passo é encontrar o vértice com menor número de possibilidades para ser colorido. Essa operação percorre todos os N^2 vértices do grafo e analisa o tamanho do seu mapa $cantBe$, o que gasta um tempo constante, nos deixando com uma complexidade de $O(N^2)$. A seguir, devemos procurar qual é a primeira cor disponível para se colorir o vértice e, como temos no máximo N cores e a busca num mapa tem complexidade $O(logN)$ graças a sua ordenação prévia, tem-se uma complexidade de $O(NlogN)$.

Para atualizar a lista de adjacências do vértice colorido, deve-se considerar que, conforme supracitado, essa lista tem tamanho máximo $O(N)$ e, portanto, ao inserir a cor no mapa de cada um dos N elementos dessa lista, tem-se novamente a complexidade de $O(NlogN)$.

Assim, considerando que todo esse processo é feito para, no máximo, todos os N^2 vértices do grafo, teremos uma complexidade de: $O(N^2) * ((O(N^2) + O(NlogN) + O(NlogN)) = O(N^2) * O(N^2) = O(N^4)$.

Por fim, é impresso na saída padrão o tabuleiro final gerado, o que tem complexidade $O(N * N)$ = $O(N^2)$, já que, para cada linha do tabuleiro, itera por suas colunas.

Dessa forma, analisando todo o algoritmo, tem-se as seguintes complexidades: $O(N^3)+O(N^2logN)+O(N^4)+O(N^2)$, o que resulta uma complexidade final de $O(N^4)$.

3.2. Complexidade de Espaço

Para a complexidade de espaço, devemos considerar todas as esturutas de dados criadas para a resolução do problema. Dessa forma, tem-se a criação de uma matriz de tamanho N x N para representar o tabuleiro do $Sudoku$, o que tem complexidade de espaço $O(N^2)$.

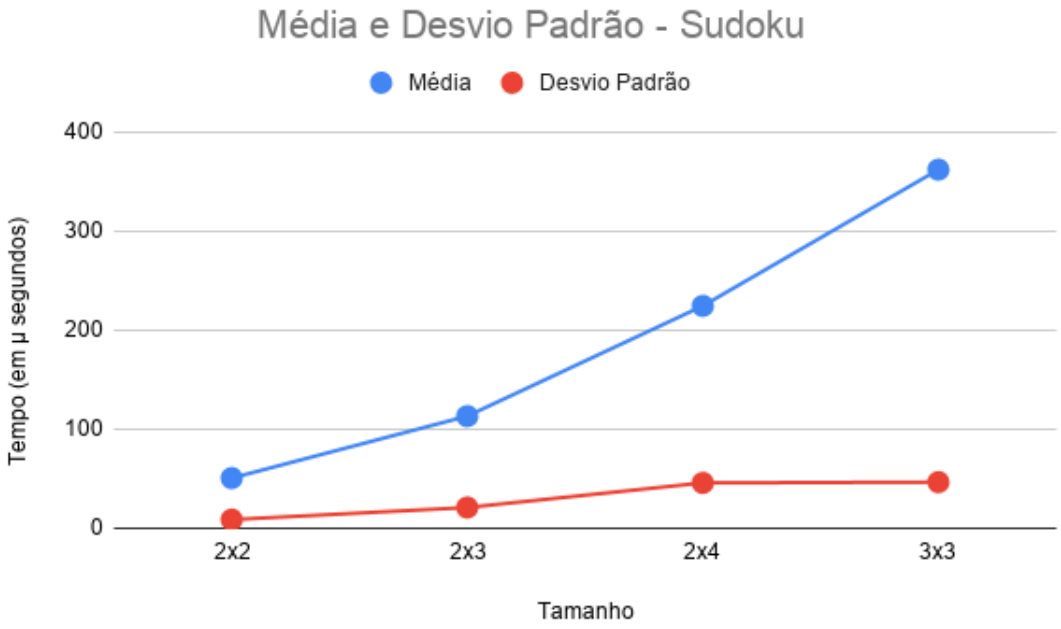
Além disso, para cada uma das N x N células do quebra-cabeça existe uma lista de adjacências e um mapa de valores(cores). Dessa forma, a lista será de, no máximo, $O(N)$, já que cada vértice estará conectado com $N-1$ células na mesma linha, $N-1$ células na mesma coluna e menos que N vértices no mesmo quadrante, a depender do tamanho do mesmo. Assim, como teremos N^2 células com as listas, a complexidade da mesma será de $O(N^3)$.

Não obstante, como existem no máximo N valores/cores a serem usados, para cada mapa individual teremos uma complexidade de $O(N)$, o que, considerando todas as células, gera uma complexidade de $O(N^3)$. Dessa forma, a complexidade de espaço final do algoritmo será de $O(N^2)$ + $O(N^3)$ + $O(N^3)$ = $O(N^3)$

4. Avaliação Experimental

Para a análise experimental, foram gerados 10 tabuleiros de tamanhos 2 x 2, 2 x 3, 2 x 4 e 3 x 3, totalizando 40 casos de teste. Posteriormente, a fim de evitar interferências devido à troca de contexto do processador junto ao sistema operacional, cada um deles foi executado 15 vezes e, com isso, calculou-se as médias e os desvios padrões.

O resultado foi condensado no gráfico abaixo para melhor visualização dos dados:



Como o esperado, vemos um rápido crescimento no tempo a medida que as dimensões do tabuleiro aumentam, o que corrobora com a complexidade $O(N^4)$ encontrada na seção anterior.

Além disso, foi calculada a taxa de acerto para cada um dos tamanhos do $Sudoku$. Assim, encontramos:

- 2x2 - 100%
- 2x3 - 100%
- 2x4 - 40%
- 3x3 - 80%

Como trata-se de uma heurística polinomial para resolver um problema de natureza não polinomial, é possível afirmar que os resultados são de bastante positivos, já que a taxa de acerta alcançada é bastante satisfatória.

5. Conclusão

A criação e implementação da heurística foi de suma importância para fixar os conceitos e paradigmas de redução de problemas não-polinomiais vistos em sala de aula, especialmente envolvendo a manipulação de grafos.

Seguramente a maior dificuldade encontrada foi na criação das estruturas necessárias no grafo para o funcionamento adequado da heurística proposta, como o uso de mapas e a manipulação de ponteiros de maneira geral. Além disso, ponderar sobre a complexidade da solução e sua taxa de acerto também representou um grande desafio durante a implementação.

Dessa forma, o trabalho foi, de fato, muito interessante, tanto para praticar o desenvolvimento na linguagem C++, tanto para melhorar a experiência na criação heurísticas pensando em maneiras viáveis de se trabalhar com problemas de natureza não-polinomial.

6. Referências

Cormen , T., Leiserson, C, Rivest R., Stein, C. Introduction to Algorithms, Third Edition, MIT Press, 2009. Versão Traduzida:Algoritmos – Teoria e Prática 3a. Edição, Elsevier, 2012.

Atílio Gomes Luiz. Coloração de grafos e suas aplicações. Universidade Federal do Céara - Campus Quixadá, 2015.

N. Ziviani. Projeto de Algoritmos com Implementações em Pascal e C. Cengage, 2011.