Trabalho Prático 3

Algoritmos 1

Lucas Paulo Martins Mariz

2018055016

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

lucaspaulom@hotmail.com

Resumo: *O trabalho a seguir se trata da aplicação de conceitos e algoritmos relacionados ao estudo de técnicas de programação bem como suas utilidades no mundo real. Além disso, questões como a discussão de melhores casos, acurácia de custos e maximizações, complexidade e performance são tratados com o intuito de concretizar e desenvolver uma capacidade analítica nos alunos para com o tema.*

Abstract: *The following work deals with the application of concepts and algorithms related to the study of programming techniques, as well as their usefulness in the real world. In addition, issues such as discussion of best cases, cost accuracy and maximizations, complexity and performance are presented in order to realize and develop an analytical capacity in students for the subject.*

1. Introdução

Foi proposto pela disciplina de Algoritmos 1 o desenvolvimento de uma aplicação capaz de fornecer soluções para uma dada instância de um *Sudoku.* O *Sudoku* é um jogo baseado na colocação lógica de números. Em sua forma padrão, células distribuídas em uma grade 9x9, constituída por 3x3 subgrades chamadas de região. Essa espécie de quebra cabeça contém algumas pistas iniciais, permitindo uma indução / dedução ao resultado. Via de regra, cada coluna, linha e região só pode ter um número de cada um dos 1 a 9, considerando uma grade do tamanho previamente mencionado.

Podemos definir o *Sudoku* com base em sua ordem. Um *sudoku* de ordem **n** consiste em uma grade n^2 x n^2 com subgrades n x n. Sua versão mais famosa consiste na ordem de n = 3, no entanto, existem versões que não são um quadrado perfeito que serão consideradas na implementação desse trabalho.

Sendo assim, o trabalho se dividiu em dois eixos específicos:

- Definir uma forma de conectar os o grafo que representará a instância do *sudoku* bem como determinar uma heurística para resolver o problema.
- Utilizando um algoritmo de coloração em grafos, determinar solução do problema. Caso ela seja encontrada, retornar para o usuário a solução. Caso contrário, informar que não tem solução e retornar a instância, mesmo que incompleta.

Ademais, a heurística escolhida tem grande importância na acurácia do programa, dado que um bom verificador pode conseguir solucionar mais instancias do que outros. Por outro lado, não é garantido que a implementação solucionará todas as entradas, devido ao fato de não existir *backtracking,* impossibilitando a correção de erros em tempo de execução. No entanto, tal fato não torna a implementação inválida, somente sugere que diferentes métodos atendem à diferentes instâncias.

Por fim, a padronização das entradas constitui um fator de suma importância para a veracidade da saída. Na primeira linha do arquivo de entrada lê-se o tamanho do *sudoku* N, a quantidade de linhas I e a quantidade de colunas J. Nas N linhas seguintes são encontrados os dados do mesmo, sendo uma instância contendo algumas pistas e 0's nas posições vazias, as quais devem ser preenchidas com o intuito de encontrar a solução.

2. Instruções de Compilação e Execução

O programa foi desenvolvido na linguagem C, compilado pelo GCC versão 7.4.0 em um sistema operacional Ubuntu 18.04. Todo o código foi executado em meu notebook cujas especificações principais estão descritas abaixo:

- Intel Core i5-7200U CPU 2.5GHZ x 4
- Nvidia G-Force GTX 840M
- 8GB RAM
- SSD Kingston A400 240GB
- Ubuntu 18.04.2 LTS

Portanto, recomenda-se que a execução do código ocorra em um sistema *Ubuntu* versão 14 ou superior contendo pelo menos a versão 7 do *GNU C Compiler (GCC)*. Foi utilizado um *makefile* que se encarrega de todas as operações de compilação dos arquivos '.c'.

A execução do projeto aguarda um argumento (*argv*) indicando o arquivo contendo os dados de entrada e deve seguir o seguinte padrão:

./tp3 path/to/file.txt

3. Implementação

A implementação do trabalho envolveu uma estrutura de dados simples, a qual armazenava as informações referentes ao *sudoku*. Através dessa estrutura, todas as manipulações e cálculos necessários foram feitos, de forma que toda a informação estivesse armazenada em um único ponto. Nas próximas seções serão discutidas as particularidades da abordagem bem como os métodos utilizados para a implementação dos algoritmos.

A ideia principal do algoritmo se baseia no problema de coloração em grafos. A instância desse problema consiste em um grafo G(V, A) no qual deve-se encontrar o menor número de cores que colorem o grafo de tal forma que vértices adjacentes tenham cores dissemelhantes. No entanto, não buscamos o menor número de cores possíveis uma vez que, dada a construção do problema, temos que $x(G) = N$, sendo $x(G)$ o número de cores necessárias para colorir o grafo G e N o tamanho do *sudoku.*

O problema de coloração em grafos é NP-Completo, possuindo uma solução exponencial. Deste modo, foram adotadas medidas com o intuito de buscar uma solução aproximada satisfatória.

3.1. Transformação do Problema

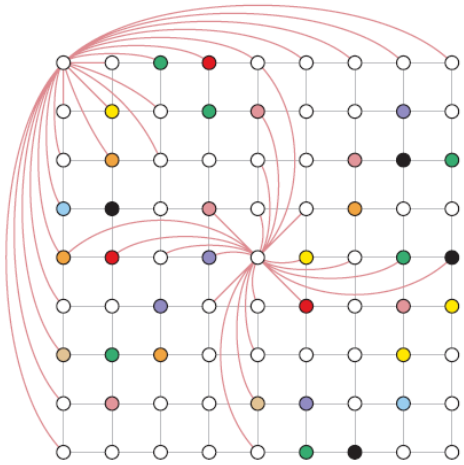

Figura 1: Esquema do grafo para um sudoku 9x9

Foi utilizada o problema da coloração de grafos para a montagem da solução, respeitando todas as regras do *sudoku* previamente mencionadas. Mapeando os conceitos, a não repetição de números nas linhas, colunas e blocos pode ser entendida como a não repetição de cores nos mesmos.

Em um primeiro momento, geramos um grafo com *NxN* vértices a partir de um *sudoku NxN*. Para cada vértice, o mesmo é conectado aos vértices de sua linha, coluna e bloco, mantendo a regra de que vértices adjacentes não podem ter cores iguais. Tais conexões foram representadas através de uma lista de adjacências. Com essa montagem a propriedades do problema em questão são respeitadas. O esquema do grafo descrito pode ser observado na figura 1. A partir dos dados lidos, o grafo é vai sendo montado dessa maneira, sendo que as pistas iniciais contribuem para a construção do vetor de possibilidades de cada vértice, o qual informa quais cores / números o mesmo não pode assumir. Tal vetor é atualizado toda vez que o grafo é modificado, de forma a manter a concisão das informações.

3.2. Caminhamento da Solução

A heurística utilizada no projeto é baseada na adoção de metodologias intuitivas. Durante a execução do programa, são armazenadas em um vetor informações sobre as cores que determinado vértice não pode assumir. Dessa forma, escolhemos o vértice que ainda não tenha sido preenchido e que tem a menor quantidade de opções de cores disponíveis entre todos os outros. Após essa escolha, caso tenha somente uma cor disponível, a mesma é escolhida. Por outro lado, caso tenham várias cores disponíveis, o vértice assume a primeira do vetor.

O processo citado anteriormente repete até que alguma condição de aceitação ou rejeição do *sudoku* seja alcançada. São elas:

- Caso não tenha mais vértices disponíveis para coloração, ou seja, todos os vértices foram preenchidos, temos que a solução foi alcançada. Isso indica que nenhum vértice do grafo foi colorido com a mesma cor de um vértice adjacente, ou seja, nenhum bloco, linha ou coluna possuem o mesmo número. Nesse caso retornamos a solução.

- Caso o algoritmo tente colorir um vértice sem opções de cores disponíveis. Isso indica que os vértices adjacentes já assumiram todas as a cores disponíveis, impossibilitando que a solução seja alcançada. Nesse caso retornamos a solução, mesmo que incompleta.

4. Análise Experimental

Uma massiva quantidade de dados gerados foi considerada na análise que se segue. Foram realizados diversos testes com o intuito de checar a veracidade dos dados. Os dados obtidos com os experimentos realizados estão armazenados no repositório do projeto no *Github* (https://github.com/LucasPMM/sudoku) e na própria pasta do projeto.

Além disso, todo o trabalho foi devidamente testado em diversos ambientes, tanto locais quanto remotos, com o intuito de adquirir os melhores resultados possíveis. Ademais, a utilização do depurador *Valgrind* foi fundamental no processo de correção de erros e prevenção de *memory leaks*

4.1. Testes

Utilizando alguns scripts, foram gerados diversos arquivos de entradas, utilizando *sudokus* de tamanhos variados. Foram realizados testes para valores de N variando de 4 a 81, sendo que cada teste foi executado 100 vezes com o intuito de obter um valor satisfatório para a média.

Inicialmente, fiz os testes na minha máquina local, no entanto, as constantes trocas de contexto existentes no processador prejudicaram os resultados finais. Ações como navegar na internet ou estar com outros programas abertos na máquina no momento da execução do trabalho implicou em inconstâncias nos resultados e na produção de diversos outliers (pontos fora da curva) na contagem do tempo de execução. Com o intuito de contornar essa situação, foi simulado um ambiente virtual *Ubuntu* na plataforma *Codenvy* e os mesmos testes foram refeitos.

4.2. Avaliação de Heurística

A partir das análises e testes realizados, é plausível avaliar a eficiência da heurística adotada, ou seja, queremos saber o quanto o algoritmo acerta / erra em determinados tamanhos de *sudokus.* Dessa forma, seremos capazes de responder as seguintes perguntas:

- *Quais foram os motivos que levaram a adoção de tal heurística?*
- *Quais formatos de tabela do Sudoku (4x4, 9x9, etc...) a heurística adotada obteve melhores soluções?*

A resposta para a primeira é bem simples, dado que a heurística foi escolhida com base na sua possível eficácia bem como na facilidade de sua implementação. A escolha de elementos de forma aleatória pode atender casos que outras heurísticas não vão conseguir produzir solução. No entanto, quando estamos falando de grandes vetores para *sudokus*

de ordem maior, a chance de acertar o elemento correto de forma aleatória é remota, sendo que caso acerte em uma determinada execução, muito provavelmente não acertará na próxima, causando uma inconsistência dos resultados gerados. Levando todos esses fatores em consideração, fui capaz de advogar que a melhor heurística, e a mais intuitiva, seria a escolha de um elemento fixo, no caso o primeiro elemento. É notável o impacto que a ausência do *backtracking* causa no cenário de escolhas inválidas, entretanto, tal quadro é plenamente explicável quando estamos trabalhando com soluções aproximadas para problemas NP-Completo.

Para a segunda pergunta alguns testes extras tiveram que ser realizados a fim de tornar a resposta mais consistente e coesa. Os dados obtidos estão contidos no gráfico da figura 2.

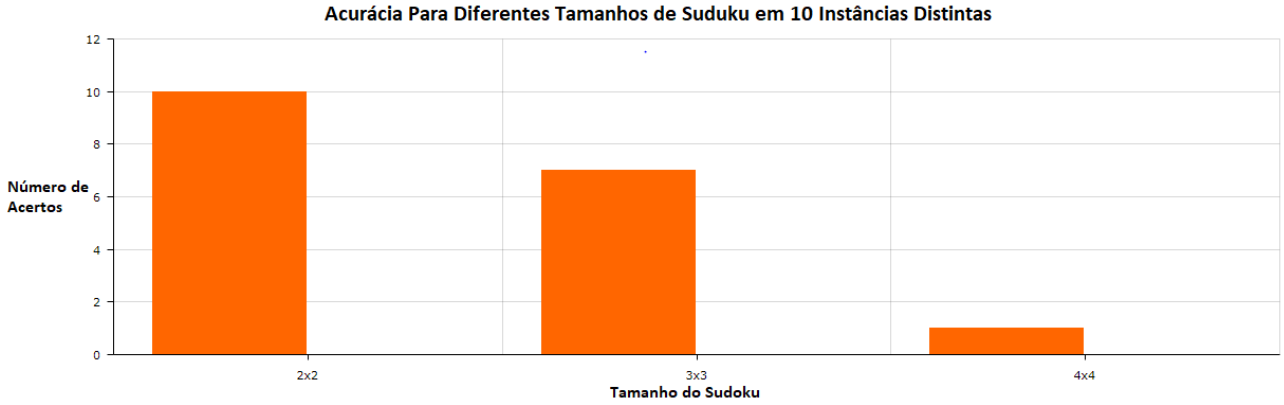

Figura 2: Gráfico Número de Acertos em 10 instâncias por Tamanho do Sudoku

Em suma, foram gerados 10 *sudokus* de diferentes tamanhos para avaliar a taxa de acerto do algoritmo, ou seja, a quantidade de vezes que o programa nos fornece uma solução válida. Os três tamanhos escolhidos foram:

- Subgrade de tamanho 2x2 em um *sudoku* 4x4;
- subgrade de tamanho 3x3 em um *sudoku* 9x9;
- subgrade de tamanho 4x4 em um *sudoku* 16x16.

Dessa forma, pode-se concluir que instâncias de tamanhos menores possuem taxas de acerto maiores do que àquelas de tamanhos maiores. Esse fato se repetiu de maneira geral durante toda a execução do trabalho, sendo que, por exemplo, não foi encontrado nenhuma entrada de tamanho 81x81 na qual o algoritmo produzia solução válida.

Essa circunstância pode ser facilmente esclarecida devido ao caso que, em instâncias menores, o vetor de cores não permitidas para cada vértice tem poucas posições, aumentando as chances de acerto. No caso oposto, as chances de acerto decaem de forma expressiva, fazendo com que o algoritmo não produza solução para a entrada.

4.3. Analise de Complexidade

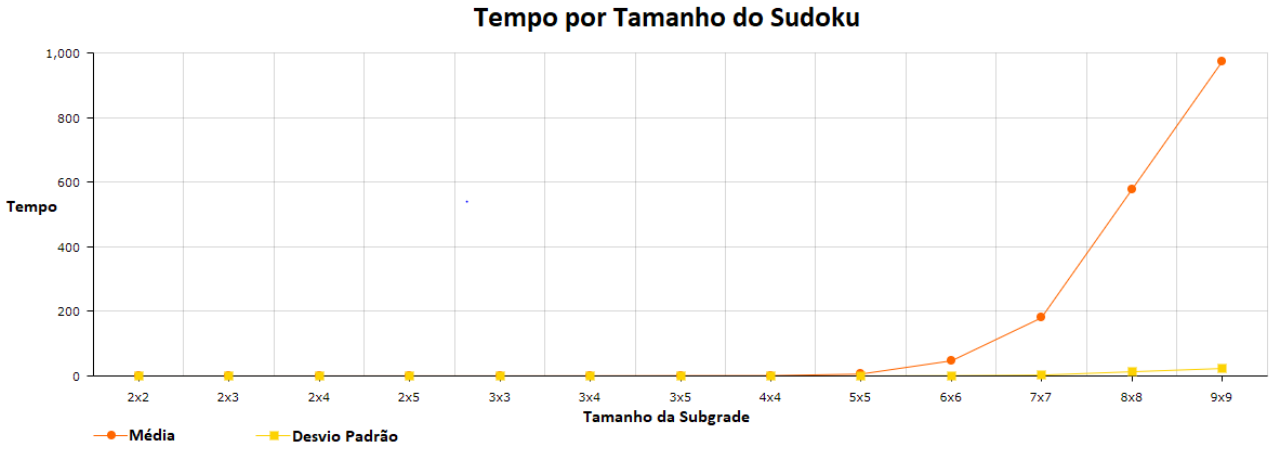

Figura 3: Gráfico Tempo (em milissegundos) por Tamanho do Sudoku

Analisando as saídas do algoritmo, disponível no diretório do trabalho e no projeto no *Github*, podemos ter várias conclusões. Em um primeiro momento, será analisada as questões de complexidade de tempo e espaço do algoritmo desenvolvido e, logo após, será feita a correlação com o gráfico obtido na figura 3, verificando se o mesmo está de acordo com a teoria abordada.

- **Complexidade de Espaço:**

A montagem do grafo envolve a construção de uma lista de adjacências. Dessa forma, temos uma complexidade de espaço equivalente a $O(N^3)$, uma vez que possuímos N^2 elementos com, no máximo, N elementos em cada lista. Além disso, durante a leitura inicial dos dados vindos do arquivo de entrada, a montagem inicial do *sudoku* é armazenada em uma matriz *NxN*. Dessa forma pode-se concluir que **o algoritmo é limitado superiormente por uma complexidade de $O(N^3)$ em relação ao espaço.**

- **Complexidade de Tempo:**

A analise da complexidade de tempo envolve algumas verificações mais detalhadas, sendo que consideraremos o pior caso para definir o limite superior. O pior caso não está somente relacionado ao tamanho da entrada e sim a configuração do *sudoku*. Para instâncias diferentes de entradas do mesmo tamanho, podemos ter casos em que uma execução é mais rápida do que a outra, dada que a disposição das pistas ajuda na solução e impacta diretamente na quantidade de iterações.

Grande parte das funções contidas no código são da ordem de N^2. Todavia, a função de calcular a solução do *sudoku* possui uma certa complexidade extra, interferindo diretamente na resultante global. Primeiro, temos um loop mais externo que é executado enquanto existirem 0's na lista de adjacências. No pior caso, teríamos uma instância contendo somente 0's, ocasionando N^2 execuções. Esse loop sempre *para* devido ao fato de ocorrerem verificações constantes em seu interior, as quais são responsáveis por finalizá-lo caso alguma regra do jogo seja quebrada. Dentro desse loop é feita outra iteração com o intuito de encontrar o vértice que tem a menor quantidade de cores disponíveis para a coloração. Em outras palavras, estamos procurando o vértice que possui poucas possibilidades de números dado o preenchimento atual do grafo. Iterar sob

todos os vértices do grafo tem um custo adicional de N^2, incrementando o custo global para N^4. Na próxima seção do loop foi feita outra iteração para encontrar a célula que atende tais possibilidades. Após isso, iteramos sob toda a lista de adjacências do vértice em questão a fim de encontrar a cor correta para o preenchimento, seguindo a heurística adotada. Esse processo tem um custo adicional de N^3, pois, ao realizar o preenchimento, verificamos se a lista já possui a cor, de forma a evitar duplicatas no vetor de controle de coloração, e a inserção é feita de forma ordenada, deixando a estrutura mais organizada. Dessa forma, podemos juntar todas esses processos e concluir que **o algoritmo é limitado superiormente por uma complexidade de $O(N^5)$ em relação ao tempo.**

Com isso, temos o gráfico da figura 3 representando o cenário descrito. Pode-se observar que para instâncias menores o tempo de execução é relativamente baixo. No entanto, com o crescimento das entradas, o tempo cresce rapidamente, caracterizando o fator N^5.

5. Conclusão

Com a realização dos testes e análise dos dados, é notável que diversos problemas reais podem ser solucionados utilizando diferentes abordagens de programação existentes e, dependendo do objetivo buscado, tratamentos distintos devem ser tomados. Além disso, é necessário destacar que alguns erros de medida podem ter ocorrido devido a aleatoriedade dos testes gerados e pela troca de contexto existente no processador do computador, fazendo com que processos percam ou ganhem prioridade na execução, oscilando o tempo final de execução do trabalho. No entanto, essa situação foi controlada em partes ao executar o trabalho em um sistema remoto livre, em grande parte, das trocas de contexto locais.

Além disso, a distinção dos resultados obtidos devido a metodologia adotada bem como a heurística utilizada foram de suma importância para as discussões das seções anteriores. Tal fato nos leva a concluir que não se pode afirmar que uma heurística é melhor que a outra, apenas podemos inferir que existem objetivos diferentes em seus procedimentos e métodos e, dado o fato que se trata de uma aproximação da solução real, temos que determinados tratamentos irão cobrir casos desiguais quando comparados.

Sendo assim, a análise qualitativa de um problema é de fundamental importância tendo em vista a procura pela melhor solução em termos de complexidade e velocidade da execução. Pode-se afirmar que o trabalho teve grande importância no processo pedagógico da disciplina, colocando em pratica e provando conceitos vistos em sala, bem como pode-se dizer que os objetivos foram cumpridos e as dificuldades superadas uma vez que as metas foram alcançadas.

6. Bibliografia

Sudoku Puzzles – Gerador de casos de teste. Disponível em <http://www.menneske.no/sudoku/eng/>. Acesso em 23 nov 2019.

Ziviani, Nívio. (2011), Projeto de Algoritmos com Implementações em Pascal e C, Cengage Learning, 3° Ed.

Slides da disciplina de Algoritmos 1 da professora Jussara.