TP3 - ALGORITMOS I

Universidade Federal de Minas Gerais

Luiz Felipe Couto Gontijo - 2018054524

- **Objetivo**

O objetivo do trabalho consiste em, dada uma instância do clássico problema do sudoku, utilizar uma heurística e desenvolver um algoritmo que complete as casas faltantes, ou seja, resolva o sudoku. Lembrando que o sudoku de entrada pode ser definido além do clássico 9x9, podendo ter tamanhos diferentes de quadrantes (mas que não ultrapasse o 9x9).

- **Ambiente de desenvolvimento e Compilação**

O projeto foi desenvolvido em C++.

Para compilar, basta rodar o comando *make* no diretório onde se encontra o arquivo *tp3.cpp.* Após gerado o executável, execute-o passando o arquivo texto de entrada como parâmetro. Segue um exemplo:

> make
> ./tp3 teste.txt

- **O problema**

O problema é simplesmente resolver a instância de sudoku recebida como entrada. Explicando brevemente, o sudoku é um jogo cujo objetivo é completar um grid (colunas verticais e linhas horizontais) preenchendo os espaços vazios com números de 1 a N(sendo N fornecido na entrada como o tamanho do "lado" de um sudoku NxN) , de forma a não repetir o mesmo número em uma mesma linha, coluna ou quadrante - a quantidade de linhas e colunas de cada quadrante também é definido na entrada - .

- **Implementação**

- **Sobre a heurística adotada**

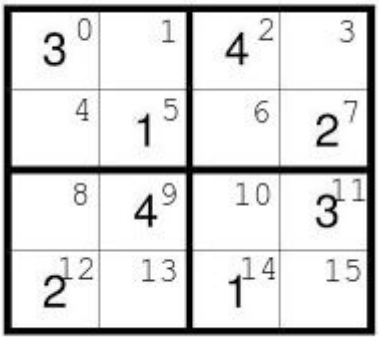
Para a implementação do problema, foi adotada uma heurística baseada em coloração de grafos. Cada vértice do grafo é definido como uma das casas da instância do sudoku recebida como entrada. Há uma aresta entre dois vértices caso ambas as casas que tais vértices representam não possam possuir o mesmo número. Ou seja, um vértice **V** qualquer possui aresta para todos aqueles que estão

em sua linha, coluna e quadrante. As “cores” são representadas pelos números de 1 a N (lembrando que N é o tamanho do “lado” do sudoku de NxN), e a resolução do problema se define em encontrar uma distribuição de cores em que nenhum vértice tenha a mesma cor de um vértice adjacente.

Tal heurística explicitada acima foi adotada, pois ela é capaz de definir o problema apresentado de forma mais clara e visível, fazendo com que os algoritmos posteriormente desenvolvidos sejam implementados baseando-se em uma estrutura já conhecida, a de grafos. Por exemplo, *fazer com que nenhuma casa tenha o mesmo número que outra casa em sua linha, coluna ou quadrante,* se reduz em *fazer com que um vértice não tenha a mesma cor que nenhum outro vértice conectado à ele.*

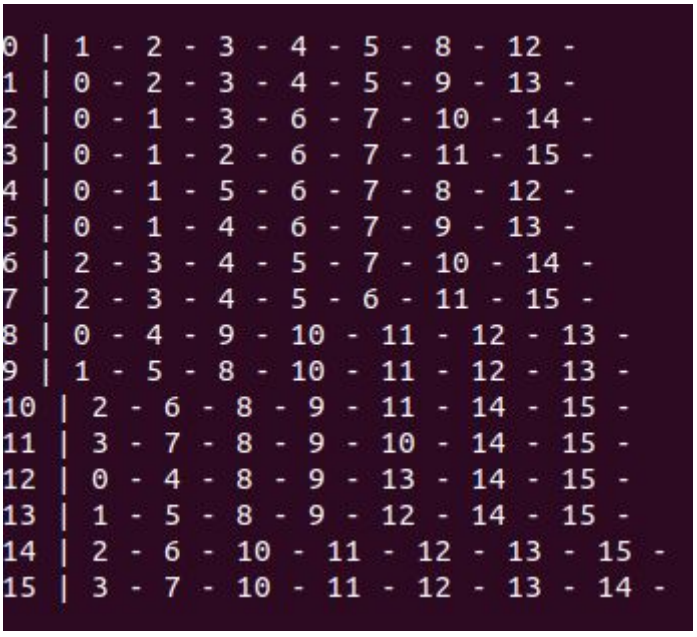
- **Criação do grafo e inicialização de variáveis**

Antes de tudo, todas as casas da instância do sudoku foram numeradas de 0 a (NxN) - 1 (pois o zero está incluso) sequencialmente da esquerda para a direita e de cima para baixo. A numeração de cada vértice/casa será chamada de índice do vértice/casa. Segue um exemplo de uma instância 4x4 e os índices de cada casa, para melhor compreensão:



É criada, então, a instância do grafo a partir da entrada. Para isso, foi feita uma lista de adjacência que possui cada um dos vértices/casas. Cada elemento dessa lista possui uma outra lista de adjacência que indica a quais vértices este elemento está conectado. Então, se há uma aresta de *A para B, B* está na lista de adjacência de *A*. Definir o índice do vértice anteriormente foi importante pois assim é possível identificar quais vértices estão em uma mesma coluna, linha ou quadrante. A linha é definida pelo quociente da divisão do índice do vértice pelo tamanho N do sudoku. Já a coluna, é definida pelo resto dessa divisão. Os quadrantes foram definidos como tuplas (x,y), em que x é a linha do vértice dividido pela quantidade de linhas do quadrante(fornecida como entrada), e y é a coluna do vértice dividido pela quantidade de colunas do quadrante. Assim, após a criação de cada vértice, já

é definido a qual linha, coluna e quadrante ele está inserido. Feito tudo isso, basta iterar por cada vértice, adicionando em sua lista de adjacência todos aqueles que possuem mesma linha, coluna ou quadrante (excluindo ele próprio). Para o exemplo dado anteriormente, temos a lista a seguir:



Para a definição de cores/número dos vértices, foi criado um vetor que possui todas as possibilidades de números de 1 até no máximo N que aquele vértice pode receber. Caso o vértice já esteja colorido, ou seja, já possui um número definido desde a entrada, esse vetor tem tamanho 1, com o número que ele está preenchido. Caso, contrário, o vetor é iniciado com todas as possibilidades. O restante do algoritmo pode ser dividido em *Primeira Tentativa de Preenchimento* e *Segunda Tentativa de Preenchimento*.

- **Primeira Tentativa de Preenchimento**

A primeira tentativa de preenchimento consiste em selecionar aqueles vértices que já possuem cores/números definidos e, então, retirar seu respectivo número/cor da lista de possíveis cores dos vértices que estão conectados a estes já preenchidos. Por exemplo, dado o vértice de índice = 0 do exemplo acima, sabemos que ele está preenchido com a cor 3. Assim, para os vértices de índices = 1,2,3,4,5,8 e 12, retiramos o número 3 da sua lista de possíveis cores. Caso o 3 já tivesse sido retirado ou o vértice também é um vértice já preenchido, não é necessário fazer nada. Fazendo isso, possivelmente são criados mais vértices que

possuem tamanho de sua lista igual a 1 (além daqueles dados como entrada), e então, eles estão agora preenchidos, e realizamos o mesmo procedimento para eles. O ideal seria que dessa forma o algoritmo conseguisse fazer com que todos os vértices tivessem tamanho igual a 1, ou seja, todos possuiriam apenas uma possibilidade de cor, e portanto, o problema estaria resolvido. De fato, isso ocorre para diversas instâncias do sudoku, e o algoritmo implementado é capaz de detectar caso isso ocorra, retornando a solução do problema. Porém, caso seja detectado que não está ocorrendo mudanças no estado do grafo, ou seja, não estão ocorrendo remoção de cores de vértices e ainda não há solução, o algoritmo interrompe a iteração atual e passa para a *Segunda tentativa de preenchimento.*

- **Segunda Tentativa de Preenchimento**

A segunda tentativa de preenchimento consiste em, primeiramente, pegar o resultado da primeira tentativa e criar uma "memória" para ele, ou seja, uma cópia. Dessa forma, é possível voltar para essa instância sempre que necessário. É realizado, então, um *backtracking.* O algoritmo consiste em selecionar o primeiro vértice que ainda não tem cor definida e definir uma cor para ele, dentre as cores presentes em sua lista de possíveis cores. E então, rodar o algoritmo da primeira tentativa de preenchimento com essa nova instância. Caso seja encontrada a solução, o algoritmo a retorna. Caso contrário, ou seja, caso algum vértice tenha ficado sem cores disponíveis, o algoritmo retorna para o estado de memória criado anteriormente (após a saída da primeira tentativa). Feito isso, é selecionado uma outra cor que não aquela selecionada anteriormente. Caso esse primeiro vértice não possua mais cores disponíveis, é passado para o próximo vértice que ainda não foi preenchido. E assim por diante.

Mesmo fazendo isso, e passando por todos os vértices e todas as cores possíveis, pode ser que o algoritmo não tenha encontrado uma solução, seja por que é necessário um backtracking em cima de um outro backtracking* (o que aumentaria muito a complexidade do algoritmo) ou seja porque realmente não há solução. Nesse caso, o algoritmo retorna "sem solução" e até onde ele conseguiu chegar.

* *Um backtracking em cima de outro backtracking* seria um backtracking em mais níveis, ou seja, supor a solução para um vértice e então, supor novamente a solução para um próximo vértice em cima de uma suposição, guardando na memória ambas as instâncias. Fazer isso para todos os vértices sem preenchimento aumentaria muito a complexidade de espaço/tempo do algoritmo e como todos os casos de testes fornecidos no dataset passaram com apenas um nível de backtracking, foi decidido não implementar essa solução.

- **Complexidade**

A análise de complexidade do algoritmo pode também ser dividida de acordo com as etapas apresentadas na implementação.

- **Criação do grafo e inicialização das variáveis**

Para criar o grafo, iteramos por todas as casas do sudoku, criando um vértice para cada uma. Além disso, para cada vértice **V**, iteramos em todos os outros vértices para criar a lista de adjacência de **V**. Assim a complexidade de tempo é $O(n^2)$ e a complexidade de espaço é O(2n), já que é criada uma instância do objeto vértice para cada casa do sudoku, e cada um possui uma lista de adjacência que não possui mais de N vértices.

- **Primeira tentativa de preenchimento**

A primeira tentativa de preenchimento itera quantas vezes for necessário para que todos os vértices tenham uma única possibilidade de cor ou que não haja mais mudanças no grafo. Nesse caso, o pior dos casos seria quando tivéssemos que passar para a segunda tentativa de preenchimento, mas ao mesmo tempo que preenchemos a maior quantidade de vértices possíveis (mas ainda sim continuamos sem solução). Quando avaliamos apenas para um vértice, a complexidade depende do número de vértices que este primeiro possui em sua lista de adjacência, mas que obviamente não ultrapassa n. Seja $O(n^2)$ então (não é um limite firme, mas é certamente um limite superior não absurdo) cada iteração para todo o grafo. O número de iterações necessárias não é grande o suficiente para ser considerado, e é possível perceber isso olhando para duas situações possíveis:

  - Caso haja muitas casas à serem preenchidas, o algoritmo logo ficará sem opções, e portanto o grafo não mudará, e assim passaremos para a próxima fase sem muitas iterações aqui.
  - Caso haja muitas casas que já estão ou que com certeza serão preenchidas, então o algoritmo já conseguirá a solução do puzzle, e não precisaríamos passar para a próxima fase.

Assim, um bom limite superior de tempo de execução para esta etapa é $O(n^2)$. Nessa etapa, não alocamos nova memória e, portanto, a complexidade de espaço não se aplica aqui.

- **Segunda tentativa de preenchimento**

A segunda tentativa de preenchimento será a mais custosa de toda a aplicação. Nela, precisamos iterar por cada possível cor de cada possível vértice a ser preenchido. Se possuímos uma instância em que poucas cores foram definidas na primeira etapa, então esta etapa será bastante custosa, já que no pior caso temos uma instância sem solução em que tentamos todas as cores de todos os vértices. Assim, como temos no máximo $\sqrt{n}\,(ou\ n^{0,5})$ cores para cada casa, e para cada iteração de cada vértice, de cada cor, teremos uma complexidade de aproximadamente $n^{2,5}$. Portanto, um limite superior e não absurdo pode ser $O(n^{2,5})$ para esta etapa. A heurística adotada interrompe o algoritmo quando percebe que não estão ocorrendo mais mudanças, e portanto, o caso explicitado acima é bem específico (mas possível). Esse limite é esperado pois estamos realizando apenas um nível de backtracking. Caso fossem feitas suposições "em cima" de suposições, teríamos uma complexidade bem maior.

A complexidade de tempo final do algoritmo é $O(n^{2,5})$, e a de espaço é de $O(n^2)$.

- **Testes**

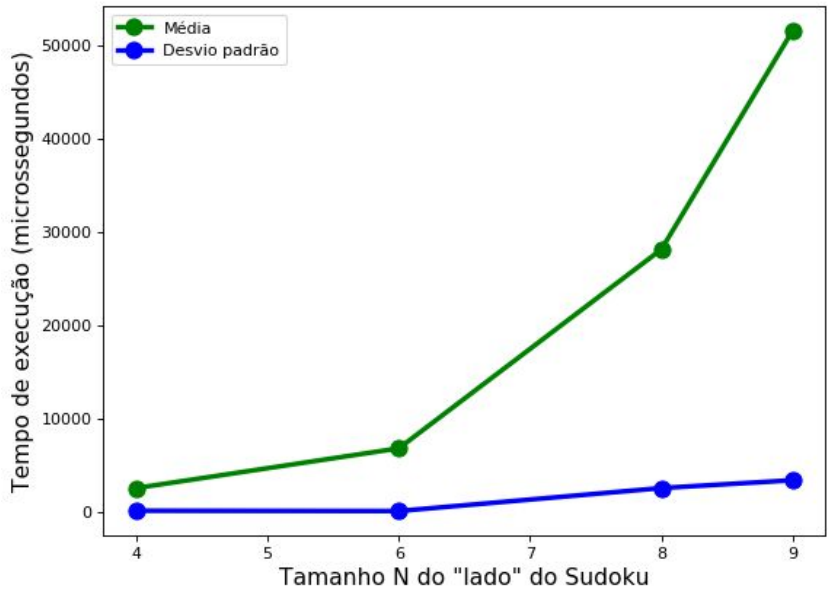
Para a realização dos testes, foram implementados 3 casos de teste para cada um dos sudokus de 4x4, 6x6, 8x8 e 9x9. Cada caso de teste foi executado 20 vezes, e então retirada a média e o desvio padrão. Os 3 casos de teste para cada um dos sudokus foram:

- Sudoku com múltiplas soluções no pior dos casos (ou seja, todas as casas estão vazias)
- Sudoku sem solução
- Sudoku com solução única

As tabelas e os gráficos abaixo mostram os resultados encontrados:

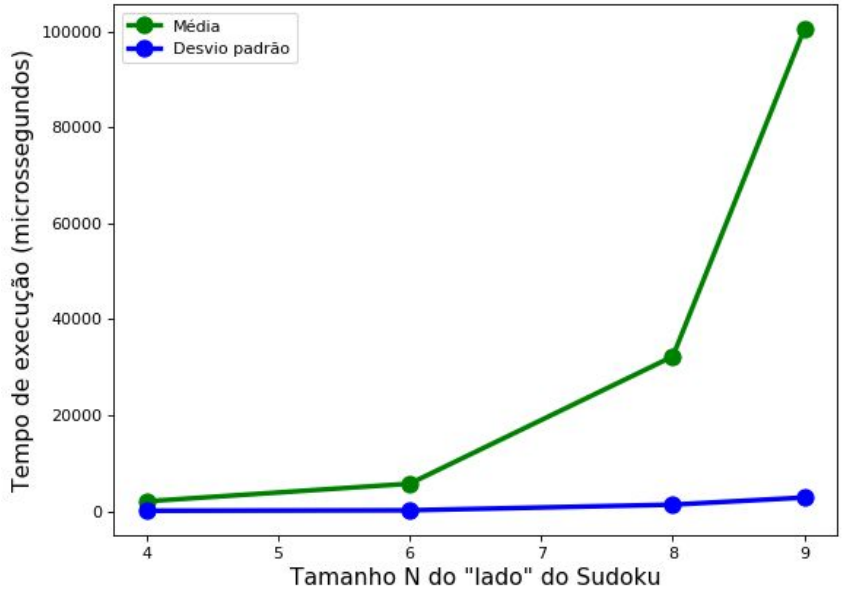
Múltiplas soluções

N	4	6	8	9
Média	2574,7	6831,3	28142,1	51575,75
Desvio Padrão	150,55	115,91	2576,52	3413,86



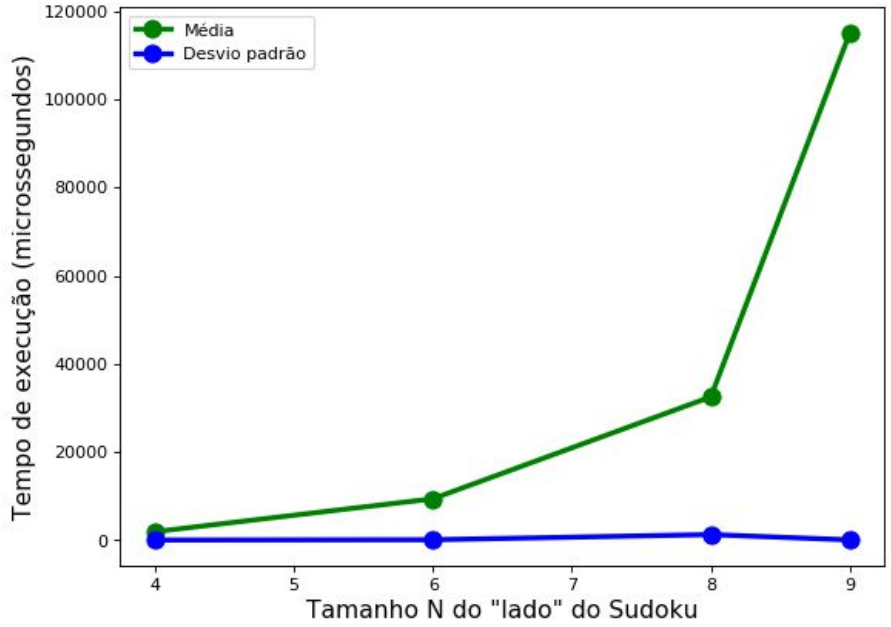
Sem solução

N	4	6	8	9
Média	2029,2	5697,75	32229,85	100526,25
Desvio Padrão	61,181	150,87	1330,29	2817,76



Única solução

N	4	6	8	9
Média	1946,45	9403,75	32528,55	115079,2
Desvio Padrão	53,25	113,1	1290,03	79,13



Os testes acima apresentados são capazes de evidenciar a natureza mais que quadrática do algoritmo de acordo com o tamanho N de entradas. Além disso, é possível perceber que a operação mais custosa se dá quando há solução, e que a menos custosa é quando a entrada possui inúmeras soluções. Isso ocorre pois a heurística implementada não retorna uma solução possível quando percebe que nem todos as casas podem ser preenchidas com apenas um número. O algoritmo só preenche casas vazias quando tem certeza que aquela cor é única para tal vértice. Assim, quando há uma única solução, o custo de preencher casas e remover cores se sobrepõe ao custo de procurar uma solução única para um caso em que há inúmeras soluções. Há ocasiões em que este custo possivelmente se inverteria, porém seriam instâncias bastante específicas do problema.

- **Questões**

Análise de pior caso do tempo de execução e da quantidade de memória utilizada pelo algoritmo.

O pior caso de tempo de execução ocorre na maior instância possível de entrada do sudoku (9x9) , e quando há uma única solução. Como explicado anteriormente, quando há uma única solução, o custo de preencher casas e remover cores se sobrepõe ao custo de procurar uma solução única para um caso em que há inúmeras soluções. Há ocasiões em que este custo possivelmente se inverteria, porém seriam instâncias bastante específicas do problema. Em teoria, porém, um caso com múltiplas soluções mas que o algoritmo necessitasse realizar o backtracking inúmeras vezes, seria mais custoso.

Quais os formatos de tabela do Sudoku (4x4, 9x9, etc..) a heurística adotada obteve melhores soluções?

A heurística adotada obteve melhores soluções em casos onde o formato da tabela do Sudoku é menor. Isso ocorre pois tabelas menores significam menos iterações além de menor possibilidades de cores para cada vértice do grafo e, portanto, um menor tempo de execução do algoritmo. Um caso onde temos uma tabela pequena (a exemplo da 4x4) e possuímos apenas zeros, o algoritmo logo detectaria que não há mudanças ocorrendo e retornaria que não há solução. Esse seria o melhor caso possível.

- **Referências para a realização do trabalho e Bibliografia**

Algorithm Design, by Jon Kleinberg, Eva Tardos & Iva Tardos.

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.