Trabalho Prático 3

Jean George Alves Evangelista - 2018047021

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

jeanalves@dcc.ufmg.br

1. Introdução

O Sudoku é um *puzzle* de lógica. Sua instância é composta por uma grade/matriz numérica, contendo linhas e colunas com algumas células já preenchidas. Solucionar um Sudoku consiste em preenchê-lo de tal forma a não repetir números nas linhas, colunas e blocos internos. Neste trabalho serão considerados tanto Sudokus que são quadrados perfeitos (os mais populares) como Sudokus que não são quadrados perfeitos. As figuras abaixo apresentam os dois tipos de Sudokus.

	2	4			7			
6								
		3	6	8		4	1	5
4	3	1			5			
5							3	2
7	9						6	
2		9	7	1		8		
	4			9	3			
3	1				4	7	5	

Figura 1. Instância de um Sudoku que é um quadrado perfeito .

2	3				1		
						8	
7			6				
					3		7
1		5		3			
			4	1		6	
	7					3	8
				5			4

Figura 2. Instância de um Sudoku que não é um quadrado perfeito .

Tal como descrito no enunciado, o objetivo do trabalho é implementar um programa na linguagem de programação C/C++ que receba como entrada um sudoku NxN, composto por sub quadrantes de tamanho IxJ. O programa deve de alguma forma tentar solucionar o sudoku, respeitando as regras do *puzzle*. No fim, deve ser mostrado se o sodoku possui

solução ou não, além de mostrar a solução - caso não tenha solução deve ser mostrado até onde o algoritmo conseguiu chegar.
O trabalho foi realizado utilizando a técnica de *backtracking*, que será melhor explicada na seção seguinte, bem como a lógica utilizada para resolução.

2. Implementação

Optou-se por realizar a implementação na linguagem de programação C++. Foi escolhida a implementação via *backtracking*, uma vez que foi permitido solucionar o problema sem utilização de Coloração de Grafos.

2.1 Classes

Sudoku: Classe principal do programa. Possui como atributos a dimensão total do sudoku (N), o número de linhas de cada bloco, o número de colunas de cada bloco e uma matriz NxN responsável por armazenar o sudoku propriamente dito. Existem dois métodos públicos na classe, um para mostrar o estado atual do Sudoku (matriz de números) e outro para tentar solucionar o Sudoku (método recursivo).

2.2 Descrição da execução, dos principais métodos e das escolhas de implementação

2.2.1 main.cpp

Como todo programa desenvolvido em C++, a execução inicia-se no na função *main* presente no arquivo *main.cpp*. Na função *main* a primeira coisa feita é tentar abrir o arquivo (cujo nome foi recebido como parâmetro na execução) contendo as entradas, caso o arquivo não seja aberto por qualquer motivo o programa se encerra. Em seguida, um objeto do tipo *Sudoku* é instanciado, passando o arquivo lido para o construtor da classe, que preenche os atributos do sudoku corretamente. Em seguida o arquivo que foi aberto para a leitura é fechado - caso não seja fechado por qualquer motivo o programa se encerra.
Nesse ponto da execução já foram lidos e armazenados os dados necessários, é chamado então o método *resolver* para tentar solucionar o sudoku. Caso *resolver* retorne true, significa que o sudoku foi resolvido e caso contrário significa que não possível resolvê-lo. Por fim, mostra-se o estado atual do sudoku por meio do método *mostrar*.

2.2.2 Sudoku

O arquivo *Sudoku.cpp* implementa os métodos da classe *Sudoku*, definidos em *Sudoku.hpp*.

2.2.2.1 Método construtor

O construtor recebe um arquivo já aberto. Por meio desse arquivo obtêm-se os dados necessários, como as dimensões do sudoku e a largura/altura dos blocos internos. Com

base nas dimensões aloca-se a memória necessária para a matriz que representa o sudoku, utilizando o operador *new* para alocar dinamicamente. Tendo alocado a memória, é possível começar a ler os números que constituem o sudoku e armazená-los na matriz. Os números vão de 1 até N, sendo N a dimensão do sudoku. 0 é utilizado para indicar que a célula ainda não está preenchida. A partir desse momento os dados estão organizados e podem ser utilizados para solucionar o sudoku.

2.2.2.2 Método resolver()

Método principal de todo o programa. Como o próprio nome diz, o método tenta resolver o sudoku. O método primeiramente cria uma variável para linha e uma variável para coluna. Em seguida, ele verifica se existe alguma célula vazia/não preenchida (0s). Se não existir nenhum 0 na matriz é concluído que o sudoku foi resolvido com sucesso e retorna-se true. Caso exista algum 0 na matriz as variáveis linha e coluna criadas anteriormente corresponderão aos índices da matriz onde é preciso preencher com um dígito.

É feito um loop, que vai de 1 até a dimensão do sudoku e cada dígito (índice do do laço *for*) é testado em cada iteração. Em cada iteração, caso não existam conflitos, isto é, não exista um dígito igual ao que está sendo testado na mesma linha, mesma coluna e mesmo bloco interno, a matriz na posição [linha][coluna] recebe o dígito. O método *resolver()* então é chamado recursivamente. Caso ele retorne false em algum momento, a matriz na posição [linha][coluna] recebe 0 e o próximo dígito é testado. Embora se assemelhe a um método de força bruta, este método se baseia no conceito de backtracking para aproveitar melhor da recursão e não fazer testes desnecessários.

Por último, gostaria de salientar que o método *resolver()* descrito faz uso de outros métodos privados da classe *Sudoku*, mas suas funcionalidades principais foram descritas acima e estão explicadas em comentários no código.

2.2.2.3 Método mostrar()

Não há muito o que dizer sobre este método. Como o próprio nome diz, ele mostra o estado atual do sudoku. Isso é feito iterando-se sob a matriz que representa o sudoku e printando todos os números na ordem correta.

2.3 Backtracking

Como dito, foi utilizado da técnica de *backtracking* para resolução do sudoku. Algoritmos de *backtracking* são comumente utilizados para encontrar soluções para determinado problema que cria candidatos para a solução de maneira incremental. Isto é, a partir do momento em que conclui-se que algum candidato é inválido (não faz parte do conjunto solução), ele é totalmente abandonado. O algoritmo então dá um "passo para trás" e volta a análise para outro candidato, avaliando a solução a partir dele.

É possível compreender o backtracking como uma árvore de possibilidades, onde o nó raiz corresponde ao problema original, cada nó é um candidato e as folhas são possíveis candidatos à solução. A árvore é percorrida em profundidade, da raiz até a folha, para solucionar o problema. A figura abaixo representa uma árvore de possibilidades.

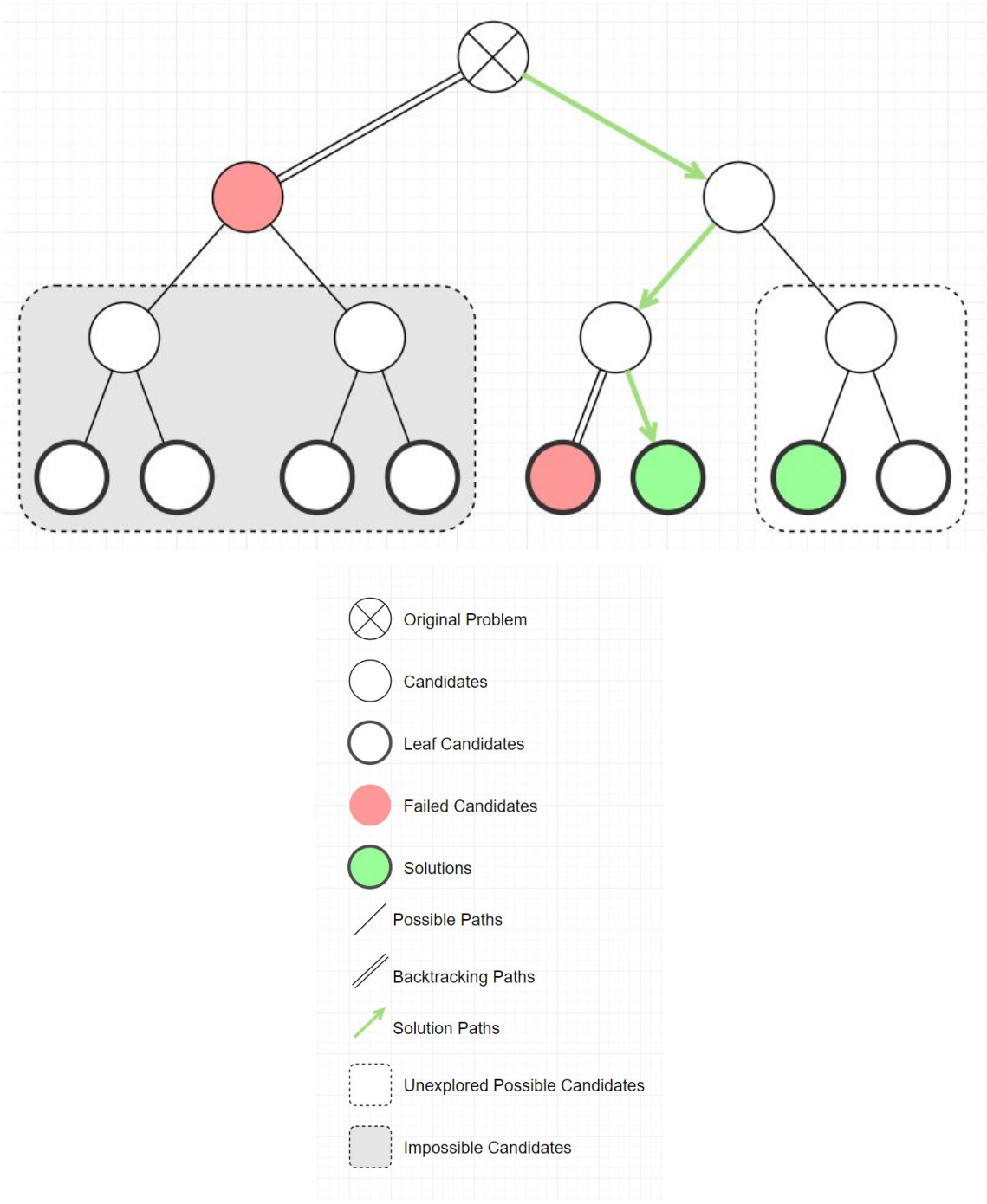

Figura 3. Representação de uma árvore de possibilidades.

Setas em verde representam um caminho que resultou numa solução válida. Linhas duplas indicam que houve *backtracking*, isto é, foi necessário voltar ao nó pai porque o nó em questão não resultará numa solução. Folhas em verde são soluções válidas, folhas/nós em vermelho são inválidas. Observa-se duas áreas na árvore, uma área tracejada com fundo cinza e outra tracejada com fundo branco. A área tracejada em cinza é um conjunto de candidatos impossíveis, tendo em vista que para alcançá-los seria necessário passar um nó que é um candidato inválido. A área tracejada com fundo branco é uma área não explorada, não sendo possível afirmar se ali existirá ou não uma ou mais soluções pelo simples fato dos vértices ali presentes não terem sido visitados. Para o problema do sudoku a área tracejada com fundo branco é particularmente interessante, porque ali podem estar presentes outras configurações para o sudoku que o solucionam corretamente, tendo em vista que um sudoku por ter múltiplas soluções.
A árvore de possibilidades para um sodoku seria semelhante ao da Figura 3, entretanto seria necessário que cada nó que não é folha tenha N filhos, sendo N a dimensão do sudoku.

3. Instruções de Compilação de Execução

Compile com make e rode com ./tp3 <nomedoarquivodeentrada>.

4. Análise de Complexidade

4.1 Complexidade de Espaço

É necessário preencher um sudoku de tamanho N*N, portanto a complexidade de espaço seria O(N*N) a princípio. Devido à recursão, temos um gasto com memória auxiliar para a pilha, mas esse gasto não aumenta em ordem a complexidade de $O(N^2)$.

4.1 Complexidade de Tempo

Para analisar a complexidade de tempo vamos partir de um exemplo, um sudoku 9x9 que não possui nenhuma célula preenchida inicialmente. Cada nó que não é folha terá 9 ramificações e o tamanho do problema é reduzido em 1. Sabemos que o problema é modelado num quadrado, então considerando $M = N^2$, a equação de recorrência será:
$T(M) = 9*T(M-1) + 1$, o que nos diz que a complexidade será $O(9^M) = O(9^{(N^2)})$.
Trata-se de uma complexidade exponencial, como esperado para um problema NP-Completo.

5. Avaliação Experimental

Foram realizados testes com todos as entradas disponibilizadas. Cada Sudoku foi solucionado 10 vezes e no fim calculou-se a média e o desvio padrão do tempo de execução. Abaixo estão os resultados encontrados em cada teste, individualmente e agregados por tamanho do sudoku.

TEMPO DE EXECUÇÃO		
SUDOKU	MÉDIA	DESVIO PADRÃO
SUDOKU	MÉDIA	DESVIO PADRÃO
221	21,40	15,37
222	10,20	8,31
223	21,00	14,14
224	23,70	22,58
225	23,2	17,09
661	41,9	19,79
662	29,1	15,2
663	28,5	15,72
664	37,6	39,82
665	13,4	8,06
881	201,6	12,85
882	47,4	16,27
883	2041,1	4342,05
884	224,7	28,09
885	1145,1	814,76
991	1488,2	3564,38
992	1310,9	1605,93
993	103,3	11,86

Tabela 1. Resultados obtidos para diferentes Sudokus.

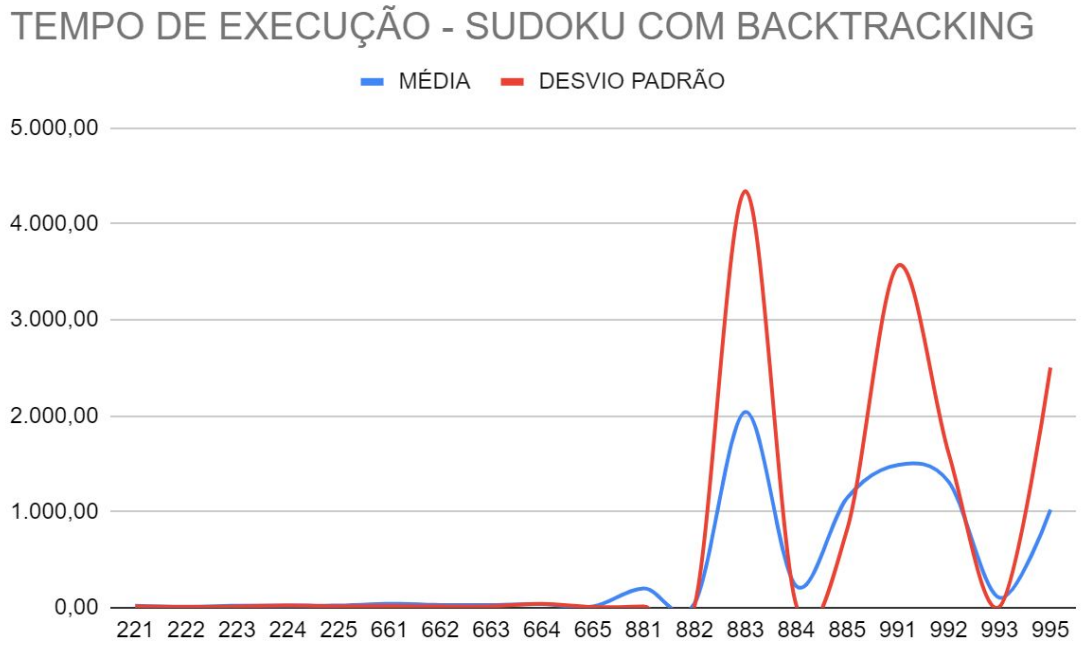

Gráfico 2. Resultados obtidos para diferentes Sudokus.

TEMPO DE EXECUÇÃO		
TAMANHO	MÉDIA	DESVIO PADRÃO
2	19,90	16,91
6	30,10	24,49
8	433,70	540,11
9	981,02	2.381,74

Tabela 2. Resultados agregados por tamanho.

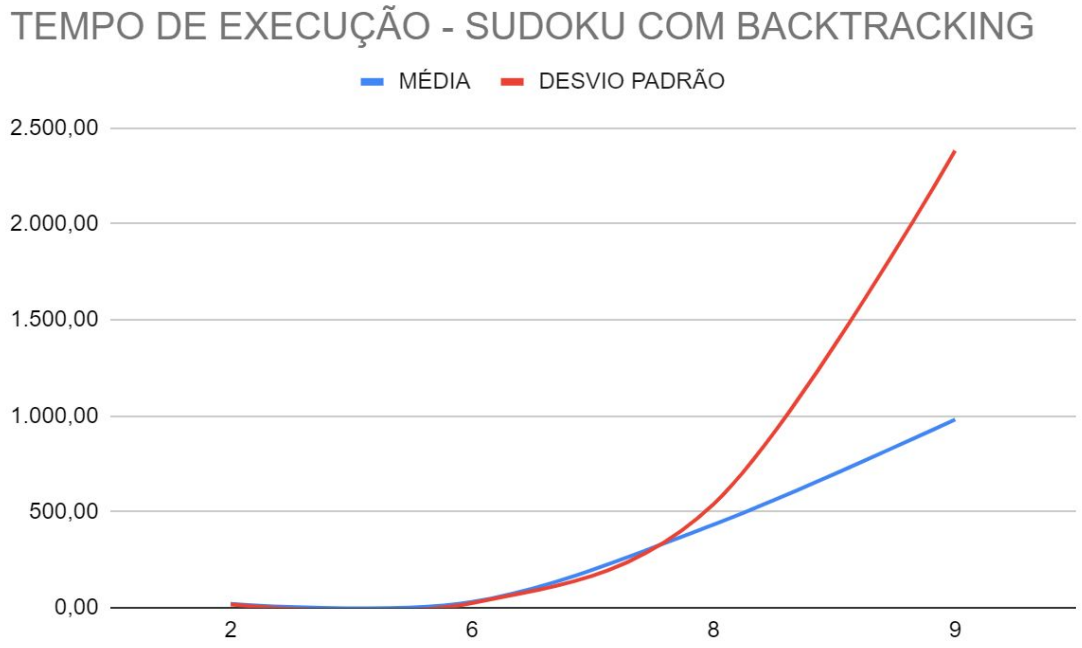

Gráfico 2. Resultados agregados por tamanho.

Primeiramente observa-se que alguns sudokus tiveram tempos de execução muito maiores do que outros, isso possivelmente se deve a terem uma solução que demora mais para ser encontrada via árvore de possibilidades. Ainda assim, observa-se o tempo de execução cresce à medida que o tamanho do sudoku aumenta.

6. Conclusão

Conclui-se que foi possível implementar um programa funcional que apresente uma solução para o problema descrito no enunciado. Por último, é possível afirmar que o trabalho foi bastante interessante e proveitoso, por permitir compreender e colocar em prática os conceitos relacionados ao backtracking para a resolução de um problema bastante interessante como o sudoku.

Referências

Hackernoon. Disponível em: <https://hackernoon.com/>. Acesso em: 24 nov. 2019.

C++ Language. Disponível em: <http://www.cplusplus.com/doc/tutorial/>. Acesso em: 24 nov. 2019.

Geeks for Geeks. Disponível em: <https://www.geeksforgeeks.org//>. Acesso em: 24 nov. 2019.

Alsuwaiyel MH: Algorithms: Design Techniques and Analysis World Scientific Publishing Company; 1998.