BELO HORIZONTE - MG
OUTUBRO DE 2019

BACHARELADO EM SISTEMAS DE INFORMAÇÃO

UNIVERSIDADE FEDERAL DE MINAS GERAIS(UFMG)

TRABALHO PRÁTICO 3
SUDOKU

ICARO HENRIQUE VIEIRA PINHEIRO
2018046556

1) INTRODUÇÃO

O Objetivo deste trabalho é propor um algoritmo capaz de solucionar o problema conhecido como “Sudoku”, que consiste em um jogo de lógica com números. A ideia do jogo é preencher a tabela, composta por linhas e colunas, de forma que não existam números repetidos na mesma linha e nem na mesma coluna, e também não podem existir números repetidos na mesma região.

Inicialmente, o jogo vem com algumas posições previamente preenchidas, dessa forma essas posições servem como um guia para o jogador se orientar e a partir disso definir quais valores podem ir para quais posições.

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

A ideia principal é aplicar a teoria aprendida em sala de aula, durantes as aulas ministradas ao longo da disciplina de Algoritmos 1 para determinar se determinada instância do Sudoku possui solução ou não.

A ideia ao longo dessa documentação é apresentar uma breve descrição do desafio proposto, explanar possíveis heurísticas para a solução do problema, defender a heurística adotada e explicar seu funcionamento, analisar a complexidade do algoritmo, através dos testes efetuados descrever a avaliação experimental e por fim propor uma visão geral após a conclusão deste trabalho.

2) IMPLEMENTAÇÃO

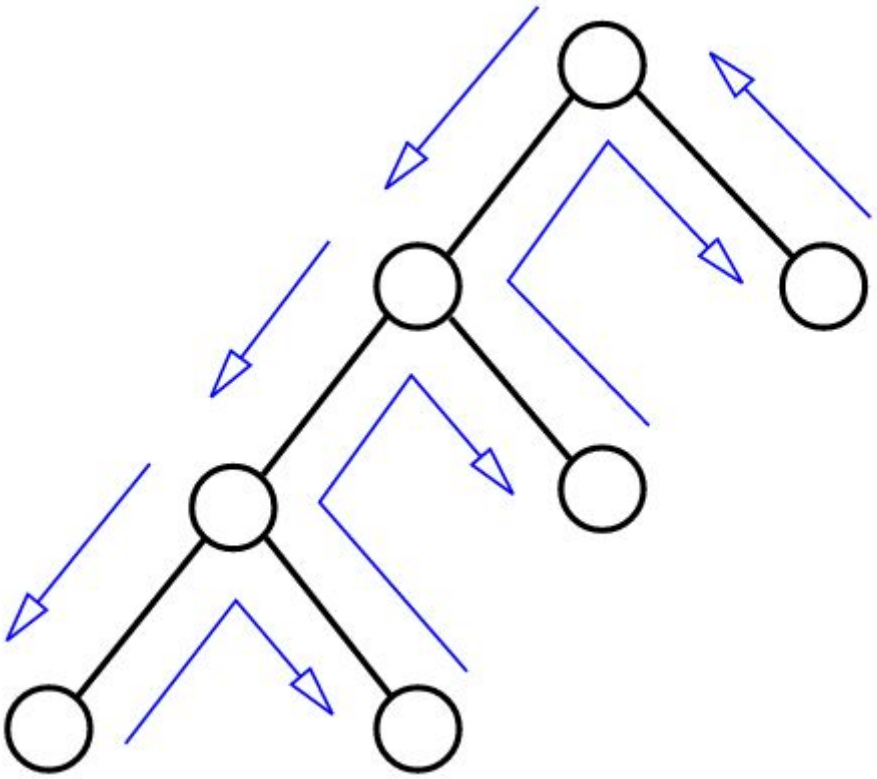
Para esse desafio em específico, optou-se por implementar a solução na linguagem de programação C++ para que dessa forma fosse possível usar o paradigma de programação conhecido como orientação a objetos, pois a outra opção seria a linguagem de programação antecessora C, que infelizmente não possui suporte para o uso de programação orientada a objetos.

Além da linguagem, a técnica escolhida para realizar o algoritmo foi a do “Backtracking”. Além do Backtracking resolver o problema de forma simples e eficaz, temos um custo de memória menor que o custo caso a implementação fosse realizada utilizando grafos, visto que ao armazenar em um grafo, devido ao formato quadrado ser perdido, seria necessário armazenar dentro dos nós os índices das posições, além de ser necessário criar listas de adjacência, entre outros padrões de referência. Por fim, ficou claro que utilizando uma matriz para armazenar o Sudoku, seria mais fácil comparar nós que possuem conflitos e como já citado anteriormente teríamos um custo de armazenamento na memória menor e uma maior facilidade para manipular a estrutura de dados.

A **técnica do Backtracking** funciona da seguinte maneira, a ideia é atribuir números um a um para cada célula vazia. Antes que esse número seja atribuído, é necessário verificar se ele não vai entrar em conflito,ou seja, entrará em conflito caso a linha, a coluna, ou a região já possua o mesmo número. Caso a atribuição possa ocorrer sem que nenhum conflito aconteça, podemos prosseguir com o algoritmo, repetindo-o, através de uma chamada recursiva, e verificar se ele leva a uma solução ou não.

De uma forma geral, o Backtracking é uma tentativa de refinar o algoritmo de força bruta, porque através dessa técnica podemos eliminar caminhos sem que seja necessários examiná-los por completo, a ideia principal parte da premissa de montar uma árvore e percorrer através dela usando chamadas recursivas, mas caso seja constatado que um caminho não levará a uma solução o algoritmo descarta aquele caminho, retrocede ao caminho anterior e então continua executando o algoritmo. Para facilitar o entendimento da técnica, observe o diagrama abaixo. As chamadas e os retornos aos nós anteriores ocorrem conforme o algoritmo percorre os nós.

É importante ressaltar que essa técnica não se aplica apenas ao problema do sudoku, e sim a vários outros problema NP-Completos. Como por exemplo o problema da mochila, o problema do caixeiro viajante dentre outros.



- **CLASSE SUDOKU**
  - A classe Sudoku está armazenada dentro do arquivo Sudoku.hpp e foi implementada com o intuito de fornecer uma estrutura de dados para o conjuntos de informações que compõem o Sudoku, ela conta com os seguintes componentes:
    - **int **matriz:** estrutura para armazenar a tabela que compõe o sudoku
    - **int tamanho:** estrutura para armazenar o tamanho da tabela do sudoku
    - **int I:** estrutura para armazenar as colunas
    - **int J:** estrutura para armazenar as linhas

    - **Sudoku():** Essa função possui a responsabilidade de ser o construtor da classe, ela atribui para a classe o número de linhas, de colunas, o tamanho total da tabela e a matriz que foi montada na main.

    - **solucionarSudoku**(): A função possui a responsabilidade de implementar o Backtracking para encontrar as possíveis soluções para a matriz, respeitando as regras do sudoku. Para isso ela se utiliza da recursão proposta de forma que ela permanece executando até que o sudoku tenha sido resolvido, caso contrário, ela retorna false, pois não é possível encontrar uma solução. A recursão e

verificação ocorre exatamente como já explicado acima no parágrafo que descreve como o algoritmo de Backtracking funciona.

- **verificarConflitos()**: A função possui a responsabilidade de conferir se existe algum conflito na linha da posição em questão, na coluna da posição em questão. Para isso ela percorre a linha e a coluna através de um laço for e verifica se o valor que se deseja atribuir a aquela posição já está presente na linha ou na coluna. Além disso, ela também verifica se o valor está presente na região da tabela.

- **verificarSeExisteElementoNaPosicao():** A função possui a responsabilidade de verificar se já não existe um valor previamente definido naquela posição em questão. Para que dessa forma um conflito seja disparado e não seja atribuído um valor naquela posição.

- **sudokuAindaNaoFoiSolucionado():** A função possui a responsabilidade de verificar se a tabela do sudoku já foi resolvida ou não, para isso ela percorre todas as posições em busca de alguma posição que ainda permaneça com o valor 0. Caso uma posição desse tipo tenha sido encontrada, ela retorna que a tabela de sudoku ainda não foi resolvida.

3) INSTRUÇÕES DE COMPILAÇÃO

Acesse a pasta inicial dos arquivos e digite no terminal o comando para que o makefile seja executado:

```
C:\Users\MasterPC\Desktop\UFMG\4 semestre\alg1\tp3>make
```

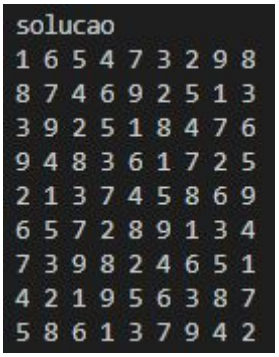
Ou caso não possua o Makefile instalado, utilize a seguinte linha de comando

```
C:\Users\MasterPC\Desktop\UFMG\4 semestre\alg1\tp3>g++ main.cpp Sudoku.cpp -Wall -o  tp3
```

Em seguida, para que o programa seja executado digite o nome do executável seguido do nome da entrada.txt

```
C:\Users\MasterPC\Desktop\UFMG\4 semestre\alg1\tp3>tp3 993.txt
```

Dessa forma, o programa produzirá o resultado esperado



4) ANÁLISE DE COMPLEXIDADE

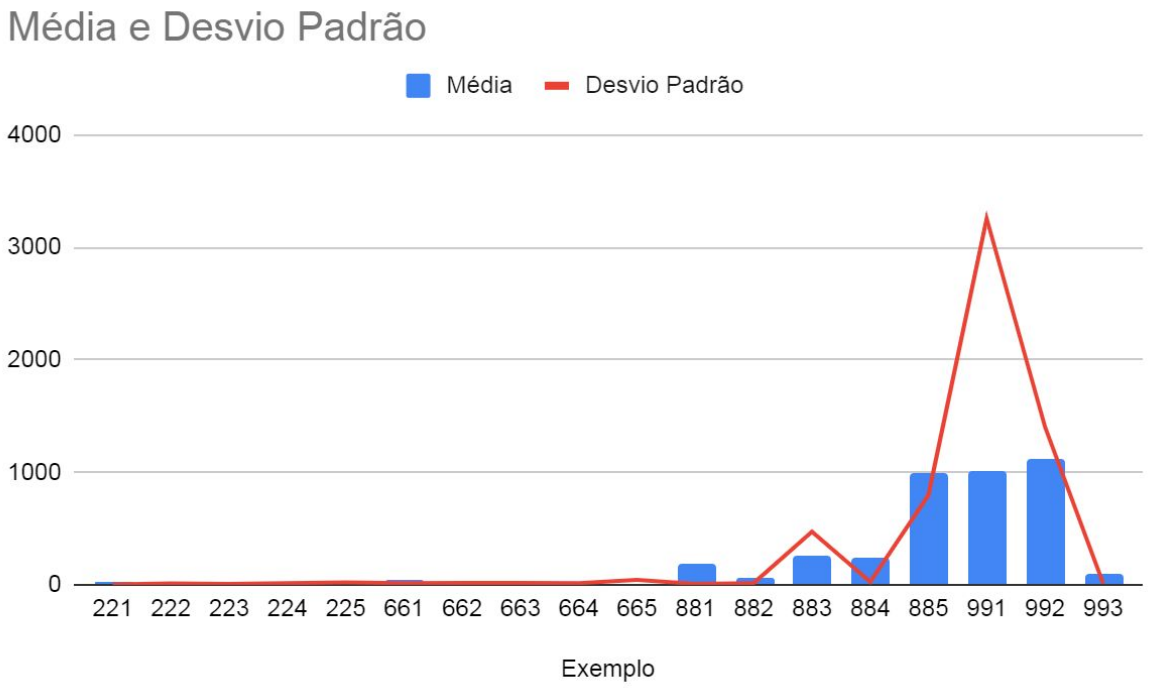
O custo para armazenar a nossa matriz, responsável por representar a tabela do sudoku é da ordem de O($\mathbf{n}^2$), pois utilizamos o espaço em memória de N linhas X N colunas, totalizando $\mathbf{n}^2$.

Além disso, é necessário observar em relação ao tempo é esperado que a complexidade seja exponencial, pois se trata de um problema já conhecido NP-Completo. Dessa forma, temos que a nossa equação de recorrência será O(tamanho da tabela elevado ao nosso custo verificado anteriormente de $\mathbf{n}^2$. Ou seja, O(**T^n²**).

5) AVALIAÇÃO EXPERIMENTAL

O objetivo desta análise é verificar o desempenho da solução proposta, busca-se encontrar o pior caso. Foram utilizadas diferentes entradas para servirem de amostras para o problema a fim de garantir a integridade dos dados, utilizou-se técnicas estatísticas para o cálculo da média, como o uso de desvio padrão.

Exemplo	Média	Desvio Padrão
221	19,15	4
222	12,75	12,25
223	21	8,32
224	22,4	13,7
225	22	22,4
661	35,8	14,8
662	30,4	17,1
663	25,6	16,2
664	33,9	14,4
665	14,8	43,02
881	184,12	7
882	62,4	14,12
883	252,7	471,8
884	234	25,14
885	986,7	796,5
991	1012,4	3254,4
992	1114,5	1405,18
993	102,1	11,86



Verificamos que entradas maiores possuem um custo maior em comparação com entradas menores, isso reforça o elevado custo de um problema NP-Completo. Foi constatado o poder que um algoritmo de custo exponencial possui de elevar o custo rapidamente. Verifica-se que a solução que foi implementada conseguiu resolver de forma eficiente os desafios propostos, considerando as limitações de um problema NP-Completo.

6) CONCLUSÃO

Ao final desse trabalho, pode-se concluir que apesar de existirem diferentes heurísticas capazes de solucionar o mesmo problema, o Backtracking se mostrou bastante eficaz nesse processo. Através desse trabalho foi possível verificar de fato e entender na prática quais são as dificuldades e os desafios para se solucionar um problema NP-Completo. Isso Aguça ainda mais a curiosidade a respeito dos problemas NP-Completo, no caso de estudo apresentado neste trabalho constatamos que de fato a solução ocorre em tempo exponencial, e é possível verificá-la em tempo polinomial, mas em nenhum a ideia de “E se fosse possível solucionar problema NP-Completo em tempo polinomial, e caso fosse, como ela seria”, ou seja, a ideia de se caso algum dia prove-se que P=NP então todo o paradigma apresentado nesse projeto mudaria.

Durante o processo para realizar esse trabalho tive a oportunidade de melhorar tanto meu lado profissional como programador, fui desafiado a propor soluções para problemas, uma tarefa que requer dedicação e pesquisa, além disso, executar o trabalho prático foi uma oportunidade de aplicar na prática os conteúdos aprendidos em sala de aula, e dessa forma complementar o aprendizado.

7) REFERÊNCIAS BIBLIOGRÁFICAS

[+] Geeks For Geeks. Disponível em < https://www.geeksforgeeks.org/sudoku-backtracking-7/> Acesso em 16 nov 2019.
[+] Wikipedia. Disponivel em <https://pt.wikipedia.org/wiki/Sudoku> Acesso em 23 nov 2019.
[+]Decom UFOP. Disponível em <http://www3.decom.ufop.br/toffolo/site_media/uploads/2011-1/bcc402/slides/10._backtracking.pdf> Acesso 15 nov 2019.
[+] Youtube. Disponivel em <https://youtu.be/l7f9-GNH1j8> Acesso em 22 nov 2019.