Análise Comparativa de Algoritmos para Resolução de Sudoku

Melissa Oliveira da Conceição
Centro de Ciências Tecnológicas - Ciência da Computação
Universidade Estadual do Norte do Paraná (UENP)
Bandeirantes, Brasil
melissaoliveirac2@gmail.com

***Abstract*—Este trabalho apresenta uma análise comparativa entre diferentes algoritmos para a resolução de jogos de Sudoku em variados níveis de dificuldade. Foram realizados 115 testes, avaliando a eficácia e o tempo médio de execução de cada abordagem. As técnicas analisadas incluem *backtracking* otimizado e força bruta recursiva com *backtracking*. Os resultados indicam que ambos os algoritmos foram capazes de resolver os desafios, porém com diferenças significativas em eficiência. O *backtracking* otimizado apresentou desempenho superior, especialmente em instâncias mais complexas, mostrando sua maior viabilidade para problemas de alta dificuldade.**

***Index Terms*—*Backtracking*, Força Bruta, Análise de Desempenho, Tempo de Execução, Complexidade Computacional, Sudoku, Algoritmos de Busca**

I. Introdução

O Sudoku, é um popular jogo de lógica, frequentemente usado como um desafio computacional por conta de sua estrutura matemática e complexidade envolvida na resolução. Consiste em uma grade de 9x9, onde os jogadores devem preencher células vazias com números de 1 a 9, de modo que cada número apareça apenas uma vez em cada linha, coluna e subgrade 3x3, como mostra a figura 1. Ainda que simples de entender, a resolução de um Sudoku pode se tornar desafiadora à medida que a dificuldade do quebra-cabeça aumenta [1], [3], [4].

Com o avanço tecnológico, algoritmos de busca e otimização foram sendo aplicados para a resolução das grades de forma automática. Dentre os mais variados métodos, *backtracking* e força bruta são os mais comuns. O *backtracking*, é uma técnica de busca que visa a resolução do problema por tentativa e erro, é amplamente utilizado devido à sua eficiência em problemas de complexidade média a alta [4]. Já a força bruta, envolve testar todas as combinações possíveis até encontrar uma solução, esse método geralmente se torna impraticável em problemas grandes devido ao seu custo computacional elevado [4].

Esse trabalho teve como objetivo comparar a eficácia dos algoritmos na resolução de jogos de Sudoku em diversos níveis de dificuldade, logo, a análise considerou o tempo de execução e a eficiência de cada abordagem.

O artigo está estruturado da seguinte maneira: a Seção 2 descreve os algoritmos de *backtracking* e força bruta, enquanto a Seção 3 detalha a metodologia utilizada para implementação e avaliação dos algoritmos. A seção 4, apresenta os resultados dos experimentos realizados e a discussão dos resultados. Na Seção 5 conclui-se o estudo.

II. Descrição dos Algoritmos: Backtracking e Força Bruta

A. Algoritmo de Backtracking

O algoritmo conhecido como "*backtracking*" (retrocesso) é uma técnica de busca utilizada para a resolução de problemas por meio de tentativa e erro, ou seja, o algoritmo explora todas as possibilidades para resolver o problema, retrocedendo sempre que a escolha se prova incorreta. Esse tipo de algoritmo é comumente usado para problemas combinatórios, como quebra-cabeças e grafos [4], [5].

- **Funcionamento**
  - **Exploração Recursiva:** O algoritmo visa construir a solução passo a passo, adicionando uma nova decisão a cada etapa.
  - **Verificação:** A cada passo, é verificado se a decisão atual é válida conforme as restrições do problema. Caso não seja, a decisão é descartada.
  - **Retrocesso:** Se nenhuma decisão válida puder ser tomada a partir de uma determinada etapa, então o algoritmo retrocede para a etapa anterior e busca outra opção.
  - **Condição de Parada:** O processo continua até que:
    - Uma solução seja encontrada.
    - Todas as possibilidades tenham sido exploradas, o que indica que o problema não tem solução.
- **Recursão** O *backtracking* geralmente é implementado de forma recursiva, com uma função que tenta resolver o problema da seguinte maneira:
  - A função tenta uma decisão;
  - Caso a decisão seja válida, chama a si mesma recursivamente para o próximo passo;
  - Se encontrar uma solução ou um caminho inválido, desfaz a decisão e tenta outra.
- **Vantagens:** É mais eficiente que o método da força bruta em problemas que permitem a poda de ramos inviáveis. E também permite resolver problemas mais complexos com restrições.
- **Desvantagens:** Dependendo da estrutura do problema, o consumo de memória pode ser muito alto devido à

recursão. Assim como, a execução da busca tente a ser cansativa, a menos que se programem restrições. Também pode ser ineficiente com grandes problemas ou em casos onde há muitas possibilidades a serem exploradas.

B. Força Bruta

O algoritmo conhecido como força bruta, basicamente, é uma abordagem que tenta todas as combinações possíveis para resolver um problema. Em outras palavras, ele não utiliza de heurísticas ou técnicas de otimização, e confia no poder computacional para a exploração de todas as soluções até encontrar uma válida.

- **Funcionamento**
  - **Geração das Soluções:** Todas as combinações possíveis do espaço de soluções são geradas.
  - **Teste:** Cada combinação é testada visando verificar se atende às restrições do problema.
  - **Condição de Parada:** O algoritmo finaliza quando:
    - Uma solução válida é encontrada.
    - Todas as combinações foram testadas sem encontrar solução.
- **Implementação**
  - Geralmente é implementado em *loops* iterativos visando gerar todas as combinações possíveis.
  - Pode ocorrer adaptações visando a exploração do espaço de soluções de maneira sistemática ou aleatória.
- **Vantagens** É simples de implementar, faz com que todas as soluções possíveis sejam testadas. Não é dependente de heurísticas ou otimizações, e é útil em problemas pequenos ou bem definidos.
- **Desvantagens** É ineficiente em problemas com grandes espaços de soluções, já que o tempo de execução cresce exponencialmente com o tamanho do problema. Não aproveita das restrições do problema para otimizar a busca.

III. Metodologia

Este estudo teve como objetivo a comparação do desempenho de dois algoritmos, *backtracking* e força bruta recursiva, na resulução de Sudoku. O Sudoku como problema base se deve a estrutura bem definida do jogo e a complexidade do mesmo, que cresce de acordo com a dificuldade das instâncias, o que permitiu avaliar como cada uma das abordagens citadas se comporta diante de diferentes cenários.

O jogo é um quebra-cabeça matemático que tem como objetivo o preenchimento de uma grade 9x9 com números de 1 a 9, obedecendo três regras principais:

- Cada número deve aparecer uma única vez em cada linha;
- Cada número deve aparecer uma única vez em cada coluna;
- Cada número deve aparecer uma única vez em cada subgrade 3x3.

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

Fig. 1. Imagem de um tabuleiro de Sudoku - Fonte: Autora

A. Força Bruta e Limitações

A força bruta "pura" consiste em tentar todas as possibilidades sem a otimização do processo. Levando isso para o contexto do Sudoku, significaria tentar todos os números de 1 a 9 em cada célula sequencialmente até que todas as células estejam preenchidas [1], sem considerar restrições ou explorar soluções parciais, ou seja, tentar resolver o problema da forma mais próxima ao conceito de força bruta possível.

Um dos maiores problemas de se implementar o Sudoku seria o excesso de possibilidades, para cada célula, há 9 possibilidades de números, logo, um tabuleiro tradicional de Sudoku com 9x9 resultaria em aproximadamente $9^{81} \approx 1.97 \times 10^{77}$ possíveis combinações para preencher. Ainda que com as restrições do Sudoku (cada linha, coluna e subgrade 3x3 devem conter números de 1 a 9 sem repetições), o número de configurações válidas ficaria em torno de 6.67×10^{21} [2]. Ainda assim, o número de possibilidades é grande, o que pode tornar a busca por soluções muito ineficiente se feitas ingenuamente, por força bruta.

Outro ponto importante seria a ineficácia, mesmo tentando preencher todas as células, ainda há uma grande chance das combinações escolhidas na célula posterior violarem alguma restrição. Ou seja, seria necessário verificar e validar cada combinação em todas as células conforme avança, logo, sem o *backtracking*, que permite retroceder assim que encontra uma solução inválida, o processo se tornaria muito demorado, já que seria necessário revisar todo o tabuleiro após cada tentativa.

O uso do *backtracking* reduz drasticamente a quantidade de possibilidades que precisariam ser testadas, pois evita que o algoritmo siga por caminhos que não levariam a uma solução. Embora teoricamente possível, resolver o Sudoku sem o *backtracking* seria extremamente ineficiente e nada prático [5]. Uma abordagem de força bruta sem *backtracking* seria, de forma resumida, uma busca pouco inteligente. Logo, é recomendado o uso da técnica e, ainda outras mais visando otimização, como heurísticas ou algoritmos especializados para resolver o Sudoku de maneira satisfatória.

Para o presente estudo, as técnicas de força bruta e *backtracking* foram implementadas, sendo que o *backtracking* foi usado nos dois algoritmos mas de formas distintas, isso se justifica pela natureza do problema Sudoku, que como dito anteriormente possuí um grande número de combinações possíveis, o que o faz ter alto custo computacional, o uso do *backtracking* permite que o algoritmo reduza significativamente a quantidade de combinações testadas ao evitar explorar soluções que já são inválidas, característica essa que é fundamental para tratar a complexidade do Sudoku, pois evita que o algoritmo avance por caminhos que certamente não levarão a uma solução válida. A necessidade do uso de *backtracking* para ambos os algoritmos permite comparar diretamente a eficiência de diferentes abordagens: uma mais otimizada, que utiliza estratégias de otimização, e outra que, embora use também da técnica, se aproxima mais da força bruta ao não realizar técnicas de refinamento.

B. Especificações técnicas da máquina usada para implementação e testes

- **Hardware:**
  - **Processador (CPU):** 12th Gen Intel(R) Core(TM) i7-12700H 2.30 GHz
  - **Memória RAM:** 16 GB
  - **Armazenamento:** 1 TB
  - **Placa de Vídeo (GPU):** RTX 3060
- **Software:**
  - **Sistema Operacional:** Windows 11
  - **Linguagem de Programação:** Python 3.12.2
  - **IDE:** VSCode

C. Implementação

Como informado anteriormente, ambos os algoritmos fazem uso dos conceitos de *backtracking* e força bruta, porém um deles possuí otimização extra visando o desempenho e mais similaridade com o conceito de *backtracking*, enquanto o outro não, tornando-o mais próximo do conceito de força bruta. Todos os códigos-fonte aqui apresentados estão disponíveis no repositório GitHub [8].

D. Implementação de Backtracking

1) **Passo Um:** A função "Verifica" é responsável por verificar se um número pode ser inserido em uma posição do tabuleiro de Sudoku sem violar as regras do jogo. Visando otimização, foram utilizados conjuntos (*sets*) para armazenar os números já presentes em cada linha, coluna e subgrade 3x3, o que permite a verificação rápida de presença.

```
1  def verifica(tabuleiro, linha, coluna, num,
       linSet, colSet, subgradesSet):
2      if num in linSet[linha] or num in
           colSet[coluna] or num in
           subgradesSet[(linha // 3, coluna //
            3)]:
3          return False
4      return True
```

Listing 1. Código para resolver Sudoku com backtracking - parte 1

A função recebe o tabuleiro e os *sets* (linSet, colSet, subgradesSet) que por sua vez armazenam os valores já utilizados nas respectivas regiões. Caso o número já estiver presente em qualquer uma das estruturas, o mesmo não pode ser inserido e a função retorna *False*. Caso contrário, retorna *True*, o que permite a inserção.

2) **Passo Dois:** A principal função, chamada de "resSudoku" é responsável por implementar a abordagem de *backtracking* que explora recursivamente as possibilidades de preenchimento do tabuleiro. O algoritmo percorre o tabuleiro em busca de células vazias que foram representadas por "0". Para cada célula, ele testa números de 1 a 9, e verifica se a inserção é válida. Caso seja, o mesmo atualiza os conjuntos de controle e faz uma chamada recursiva buscando tentar resolver o restante do tabuleiro. Se em algum momento nenhuma opção válida for achada, o algoritmo retrocede, removendo o último número inserido e tentando outro valor.

```
1      def resSudoku(tabuleiro, linSet, colSet,
           subgradesSet):
2          for linha in range(9):
3              for coluna in range(9):
4                  if tabuleiro[linha][coluna] ==
                       0:
5                      for num in range(1, 10):
6                          if verifica(tabuleiro,
                               linha, coluna, num,
                                linSet, colSet,
                               subgradesSet):
7                              tabuleiro[linha][coluna
                                   ] = num
8                              linSet[linha].add(num)
9                              colSet[coluna].add(num)
10                             subgradesSet[(linha //
                                   3, coluna // 3)].
                                   add(num)
11
12                             if resSudoku(tabuleiro,
                                    linSet, colSet,
                                   subgradesSet):
13                                 return True
14
15                             tabuleiro[linha][coluna
                                   ] = 0
16                             linSet[linha].remove(
                                   num)
17                             colSet[coluna].remove(
                                   num)
18                             subgradesSet[(linha //
                                   3, coluna // 3)].
                                   remove(num)
19
20                 return False
21
22         return True
```

Listing 2. Código para resolver Sudoku com backtracking - parte 2

3) **Passo Três:** A função "mostrarTab" visa auxiliar a visualização do Sudoku antes e depois da resolução, ela formata a saída buscando torná-la legível e organizada. Ela percorre cada linha do tabuleiro e imprime no terminal os valores formatados.

```
1  def mostrarTab(tabuleiro):
2      for linha in tabuleiro:
```

```
            print("␣".join(str(num) for num in
                linha))
```

Listing 3. Código para resolver Sudoku com backtracking - parte 3

4) **Passo Quatro:** O código usa os dados de um arquivo .csv que contém instâncias de Sudoku para serem resolvidas, isso é feito dentro da função main(). Cada linha do arquivo contém um tabuleiro jogável representado como uma sequência de 81 números, que o código separa para formar uma matriz 9x9.

```
def main():
    with open(r"db\sudoku.csv", newline='')
         as arquivoCsv:
        leitor = csv.reader(arquivoCsv)
        for linha in leitor:
            desafio = linha[0]
            solucaoCorreta = linha[1]

            tabuleiro = [[int(desafio[i * 9
                 + j]) for j in range(9)]
                for i in range(9)]
```

Listing 4. Código para resolver Sudoku com backtracking - parte 4

5) **Passo Cinco:** Para otimizar as verificações de números, foram criadas três estruturas de conjuntos:

```
linSet = [set() for _ in range(9)]
colSet = [set() for _ in range(9)]
subgradesSet = { (i, j): set() for i in
    range(3) for j in range(3) }
```

Listing 5. Código para resolver Sudoku com backtracking - parte 5

Cada *Set* ajuda a garantir que os números inseridos não violem as regras do Sudoku, evitando o excesso de verificações.

6) **Passo Seis:** Para contabilizar o tempo de execução do algoritmo foi usado um contador que mede e garante precisão. Isso foi necessário para comparar a eficiência do algoritmo em diferentes níveis de dificuldades dos jogos.

```
inicioTempo = time.perf_counter()
    if resSudoku(tabuleiro, linSet, colSet,
         subgradesSet):
        fimTempo = time.perf_counter()
        tempoExecucao = fimTempo -
            inicioTempo
        print(f"\nTempo␣de␣execucao:␣{
            tempoExecucao:.9f}␣segundos")
```

Listing 6. Código para resolver Sudoku com backtracking - parte 6

7) **Passo Sete:** Após resolver o tabuleiro, a solução é comparada com a resposta esperada que está presente também no arquivo .csv. Essa validação assegura que o algoritmo produziu um resultado válido para o jogo.

```
tabuleiroResolvido = "".join(str(num) for
    linha in tabuleiro for num in linha)
if tabuleiroResolvido == solucaoCorreta:
    print("A␣solucao␣esta␣correta!\n")
else:
    print("A␣solucao␣esta␣incorreta!\n")
```

Listing 7. Código para resolver Sudoku com backtracking - parte 7

O código mostrado faz a implementação de uma solução usando *backtracking* e otimizações extras baseadas em conjuntos para jogos de Sudoku em diversos níveis de dificuldade, tais otimizações visaram reduzir as verificações desnecessárias.

E. Implementação de Força Bruta Recursiva

1) **Passo Um:** A função "verifica" é responsável por validar se um número pode ser colocado naquela posição do tabuleiro, para isso ela faz verificações se o número já existe na linha, coluna e subgrade 3x3. Caso o número não seja encontrado nas verificações, a função retorna *True*, indicando que o número pode ser colocado na posição.

```
def verifica(tabuleiro, linha, coluna, num):
    if num in tabuleiro[linha]:
        return False
    if num in [tabuleiro[i][coluna] for i in
        range(9)]:
        return False

    inicioLin, inicioCol = 3 * (linha // 3), 3
        * (coluna // 3)
    for i in range(inicioLin, inicioLin + 3):
        for j in range(inicioCol, inicioCol +
            3):
            if tabuleiro[i][j] == num:
                return False

    return True
```

Listing 8. Código para resolver Sudoku com Força Bruta - parte 1

2) **Passo Dois:** Já a função de validação "validaTabuleiro" é responsável pela validação final no tabuleiro, após a solução ser encontrada, visando garantir que todas as condições do jogo foram atendidas, ou seja, se todas as linhas, colunas e subgrades contém números de 1 a 9, sem repetições ou zeros. Caso alguma dessas condições falhe, então a função retorna *False*, que indica que o tabuleiro não é uma solução válida. Caso contrário, retorna *True*.

```
def validaTabuleiro(tabuleiro):
    for linha in tabuleiro:
        if len(set(linha)) != 9 or any(num == 0
             for num in linha):
            return False
    for coluna in range(9):
        if len(set(tabuleiro[linha][coluna] for
             linha in range(9))) != 9:
            return False
    for inicioLin in range(0, 9, 3):
        for inicioCol in range(0, 9, 3):
            subgrade = [
                tabuleiro[i][j]
                for i in range(inicioLin,
                    inicioLin + 3)
                for j in range(inicioCol,
                    inicioCol + 3)
            ]
            if len(set(subgrade)) != 9:
                return False

    return True
```

Listing 9. Código para resolver Sudoku com Força Bruta - parte 2

3) **Passo Três:** A função ”fbResSudoku” implementa o algoritmo para a resolução do jogo, que percorre cada célula do tabuleiro procurando por uma célula vazia (representada por ”0”), e para cada uma encontrada, tenta inserir um número de 1 a 9, validando-o com a função ”verifica”. Caso o número seja válido, ele é inserido no tabuleiro. Logo após isso, a função faz uma chamada recursiva a si mesma para tentar resolver o resto do tabuleiro, se bem-sucedida, o jogo é resolvido e a função retorna *True*. Se em algum momento o algoritmo encontrar uma situação onde não é possível preencher um número válido ele desfaz a alteração voltando o número para zero e tenta o próximo número. O algoritmo retorna *True* quando o Sudoku é resolvido e caso contrário *False*.

```
def fbResSudoku(tabuleiro):
    for linha in range(9):
        for coluna in range(9):
            if tabuleiro[linha][coluna] == 0:
                for num in range(1, 10):
                    if verifica(tabuleiro,
                        linha, coluna, num):
                        tabuleiro[linha][coluna
                            ] = num
                        if fbResSudoku(
                            tabuleiro):
                            return True
                        tabuleiro[linha][coluna
                            ] = 0
                return False
    return validaTabuleiro(tabuleiro)
```

Listing 10. Código para resolver Sudoku com Força Bruta - parte 3

4) **Passo Quatro:** A função ”mostrarTab” faz a impressão do tabuleiro de Sudoku no formato de matriz 9x9 visando torná-lo visualmente legível e organizado no terminal.

```
def mostrarTab(tabuleiro):
    for linha in tabuleiro:
        print("␣".join(str(num) for num in
            linha))
```

Listing 11. Código para resolver Sudoku com Força Bruta - parte 3

A função principal por sua vez, carrega o arquivo .csv que contém os jogos (os mesmos que foram testados pelo algoritmo anterior) e as respostas dos mesmos e converte em um tabuleiro 9x9. Ela também é responsável por medir o tempo de execução, verificar se o tabuleiro resolvido é correspondente a resposta correta e, por fim, fazer a impressão no terminal dos resultados.

```
def main():
    with open(r"db\sudoku.csv", newline='') as
        arquivoCsv:
        leitor = csv.reader(arquivoCsv)
        for linha in leitor:
            desafio = linha[0]
            solucaoCorreta = linha[1]
            tabuleiro = [[int(desafio[i * 9 + j
                ]) for j in range(9)] for i in
                range(9)]
            print("Tabuleiro␣inicial:")
            mostrarTab(tabuleiro)
            inicioTempo = time.perf_counter()
            if fbResSudoku(tabuleiro):
                fimTempo = time.perf_counter()
                tempoExecucao = fimTempo -
                    inicioTempo
                print("\nSolucao␣encontrada:")
                mostrarTab(tabuleiro)
                print(f"\nTempo␣de␣execucao:␣{
                    tempoExecucao:.9f}␣segundos
                    ")
                tabuleiroResolvido = "".join(
                    str(num) for linha in
                    tabuleiro for num in linha)
                if tabuleiroResolvido ==
                    solucaoCorreta:
                    print("A␣solucao␣esta␣
                        correta!\n")
                else:
                    print("A␣solucao␣esta␣
                        incorreta!\n")
            else:
                print("Nao␣foi␣possivel␣
                    resolver␣o␣Sudoku.\n")
```

Listing 12. Código para resolver Sudoku com Força Bruta - parte 4

O código apresentado utilizou de uma abordagem simples de força bruta com o auxilio da técnica de *backtracking* mas sem nenhuma otimização extra, buscando torná-lo o mais próximo da ideia original apresentada.

F. Base de dados

Os dados utilizados para testar os códigos, foram jogos de Sudoku que tinham de ser resolvidos, bem como suas respectivas respostas, que foram utilizadas para a comparação das saídas ao final da resolução de cada problema. Ao todo foram usados 115 jogos, sendo 100 jogos com níveis de dificuldade diversos, retirados de uma base de dados disponível no Kaggle [6], visando encontrar a média de tempo de cada um deles. Por fim, quinze outros jogos que tinham suas dificuldades divididas igualmente, sendo os cinco fáceis, cinco médios e cinco difíceis, todos retirados do site ”Sudoku.com” [7], bem como suas respectivas respostas para a comparação que é realizada ao final de cada código, o objetivo foi mostrar o comportamento em relação a cada nível de jogo.

IV. Análise e Discussão de Resultados

Nesta seção, foram analisados os resultados obtidos pelos dois algoritmos, o primeiro usando uma abordagem de *backtracking* com otimizações e o segundo mais parecido com a técnica de força bruta.

Primeiro foram apresentados os resultados de ambos individualmente e após isso a comparação entre os mesmos. O desempenho dos métodos foram avaliados com base no tempo médio de execução, eficácia na resolução dos tabuleiros e comparação com a outra abordagem.

A. Resultados - Backtracking

Os resultados dos testes usando 100 jogos com diversos níveis de dificuldade foi:

- **Entrada e tentativa de resolução:** De 100 jogos recebidos como entrada o algoritmo foi capaz de tentar resolver 100 deles, resultando em uma aproveitamento de tentativa de resolução de 100%.

- **Eficácia de resolução:** Todos as respostas geradas pelo algoritmo estavam iguais aos dados de resultado da tabela .csv, ou seja, a resposta desejada foi alcançada pelo método corretamente, logo, dos 100 jogos testados, todos foram resolvidos corretamente, o que significa que o algoritmo teve aproveitamento de resolução correta de 100%.
- **Tempo Médio:** A média de tempo por solução foi de 0.001398057 segundos.

Os resultados dos testes usando jogos com nível de dificuldade fácil, testando 5 jogos diferentes, foi:

- **Entrada e tentativa de resolução:** 100% de aproveitamento nas tentativa de resolução.
- **Eficácia de resolução:** As respostas geradas pelo algoritmo estavam iguais as da tabela, logo, o algoritmo teve aproveitamento de resolução correta de 100%.
- **Tempo Médio:** A média de tempo por solução foi de 0.000287160 segundos.

Os resultados dos testes usando jogos com nível de dificuldade médio, testando 5 jogos diferentes, foi:

- **Entrada e tentativa de resolução:** 100% de aproveitamento nas tentativa de resolução.
- **Eficácia de resolução:** As respostas geradas pelo algoritmo estavam iguais as da tabela, logo, o algoritmo teve aproveitamento de resolução correta de 100%.
- **Tempo Médio:** A média de tempo por solução foi de 0.001226700 segundos.

Os resultados dos testes usando jogos com nível de dificuldade difícil, testando 5 jogos diferentes, foi:

- **Entrada e tentativa de resolução:** 100% de aproveitamento nas tentativa de resolução.
- **Eficácia de resolução:** As respostas geradas pelo algoritmo estavam iguais as da tabela, logo, o algoritmo teve aproveitamento de resolução correta de 100%.
- **Tempo Médio:** A média de tempo por solução foi de 0.006934560 segundos.

Observações: O algoritmo foi capaz de resolver todos os jogos independente do nível de dificuldade. A análise de tempo médio demonstra que conforme a dificuldade do jogo aumenta, isto é, conforme menos pistas iniciais, mais necessidade de retroceder, logo, existe uma sensibilidade com a complexidade do jogo, pois quanto mais difícil o Sudoku mais o tempo de execução tende a aumentar.

B. Resultados - Força Bruta Recursiva

Os resultados dos testes usando 100 jogos com diversos níveis de dificuldade foi:

- **Entrada e tentativa de resolução:** De 100 jogos recebidos como entrada o algoritmo foi capaz de tentar resolver 100 deles, resultando em uma aproveitamento de tentativa de resolução de 100%.
- **Eficácia de resolução:** Todos as respostas geradas pelo algoritmo estavam iguais aos dados de resultado da tabela .csv, ou seja, a resposta desejada foi alcançada pelo método corretamente, logo, dos 100 jogos testados, todos foram resolvidos corretamente, o que significa que o algoritmo teve aproveitamento de resolução correta de 100%.
- **Tempo Médio:** A média de tempo por solução foi de 0.002247122 segundos.

Os resultados dos testes usando jogos com nível de dificuldade fácil, testando 5 jogos diferentes, foi:

- **Entrada e tentativa de resolução:** 100% de aproveitamento nas tentativa de resolução.
- **Eficácia de resolução:** As respostas geradas pelo algoritmo estavam iguais as da tabela, logo, o algoritmo teve aproveitamento de resolução correta de 100%.
- **Tempo Médio:** A média de tempo por solução foi de 0.000391460 segundos.

Os resultados dos testes usando jogos com nível de dificuldade médio, testando 5 jogos diferentes, foi:

- **Entrada e tentativa de resolução:** 100% de aproveitamento nas tentativa de resolução.
- **Eficácia de resolução:** As respostas geradas pelo algoritmo estavam iguais as da tabela, logo, o algoritmo teve aproveitamento de resolução correta de 100%.
- **Tempo Médio:** A média de tempo por solução foi de 0.001835940 segundos.

Os resultados dos testes usando jogos com nível de dificuldade difícil, testando 5 jogos diferentes, foi:

- **Entrada e tentativa de resolução:** 100% de aproveitamento nas tentativa de resolução.
- **Eficácia de resolução:** As respostas geradas pelo algoritmo estavam iguais as da tabela, logo, o algoritmo teve aproveitamento de resolução correta de 100%.
- **Tempo Médio:** A média de tempo por solução foi de 0.009660780 segundos.

Observações: Assim como o algoritmo anterior, o algoritmo de força bruta recursiva também foi capaz de solucionar todos os tabuleiros dos jogos e obteve resolução correta de 100%. Os tempos médios de resolução aumentam proporcionalmente à dificuldade do jogo, indicando que a técnica é sensível à complexidade da instância.

TABLE I
RESULTADOS DOS TESTES DOS ALGORITMOS

Algoritmo	Nível de Dificuldade	Tempo Médio (segundos)
Backtracking	Diversos	0.001398057
	Fácil	0.000287160
	Médio	0.001226700
	Difícil	0.006934560
Força Bruta	Diversos	0.002247122
	Fácil	0.000391460
	Médio	0.001835940
	Difícil	0.009660780

C. Comparação

Com os resultados obtidos e apresentados anteriormente, é possível realizar uma comparação entre os algoritmos implementados, com base em três aspectos principais: taxa de

sucesso, tempo de execução e escalabilidade de acordo com a dificuldade do problema.

1) Taxa de Sucesso: Os dois algoritmos apresentaram desempenho equivalente quanto à taxa de sucesso:

- **Tentativa de resolução:** Ambos os algoritmos conseguiram processar e resolver 100% das entradas.
- **Eficácia de resolução:** As soluções geradas por ambos os algoritmos corresponderam às respostas esperadas para todos os casos testados, o que garantiu o aproveitamento de 100% de resoluções corretas.

Logo, em termos de tentativa e exatidão nas resoluções dos jogos, ambos os algoritmos foram igualmente eficazes.

2) Tempo Médio de Execução: A maior diferença entre os algoritmos está no tempo necessário para encontrar a resposta. A tabela I mostra que o algoritmo de *backtracking* otimizado apresentou tempos médios de execução menores em comparação ao outro algoritmo testado

- Para jogos de dificuldade **fácil**, o *Backtracking* levou em média 0.000287160 segundos, enquanto a Força Bruta Recursiva levou 0.000391460 segundos.
- Para jogos de dificuldade **média**, o *Backtracking* teve um tempo médio de 0.001226700 segundos, menor que os 0.001835940 segundos da Força Bruta Recursiva.
- Para jogos de dificuldade **difícil**, o *Backtracking* teve um tempo médio de 0.006934560 segundos, enquanto a Força Bruta Recursiva precisou de 0.009660780 segundos.
- No conjunto geral de testes, o *Backtracking* obteve um tempo médio de 0.001398057 segundos, enquanto a Força Bruta Recursiva obteve 0.002247122 segundos.

Abaixo está o gráfico comparando os tempos médios de execução dos algoritmos, onde é possível ver a diferença de desempenho entre os dois métodos Imagem 2. É importante observar atentamente a diferença nos jogos com nível de dificuldade alto.

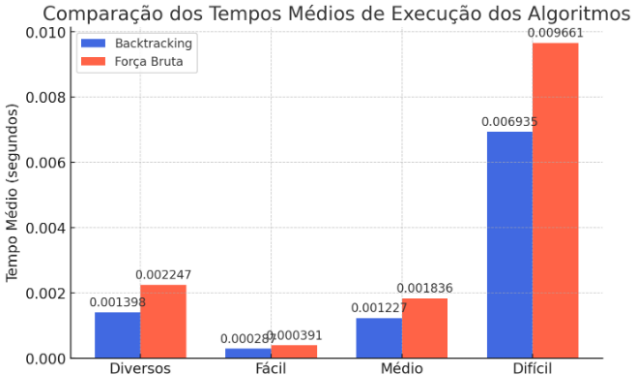

Fig. 2. Imagem de Comparação dos Tempos Médios de Execução dos Algoritmos - Fonte: Autora

3) Escalabilidade: Os resultados de ambos os algoritmos indicam que eles são sensíveis a complexidade do problema, apresentando tempos médios de execução crescentes conforme a dificuldade da entrada aumenta. Entretanto, o *backtracking* escala de forma mais eficiente, já que o tempo médio dele cresce a uma taxa menor em relação ao outro algoritmo.

Essa diferença existe por conta das podas inteligentes e otimizadas que o *backtracking* realiza, o que elimina rapidamente caminhos inválidos e reduz o espaço de busca, consequentemente, reduzindo o tempo.

4) Considerações finais: Ambos os algoritmos foram capazes de processar e resolver os problemas de maneira precisa. No entanto, ao considerar a eficiência, o *backtracking* com otimizações é a melhor escolha, ele apresentou tempos de execução menores para todos os níveis de dificuldade e sua escalabilidade também é superior, pois ele tem a capacidade de reduzir significativamente o número de recursões, o que o torna uma solução mais viável para instância mais complexas de Sudoku.

V. Conclusão

Este trabalho apresentou uma comparação entre algoritmos de busca na resolução de jogos de Sudoku com vários níveis de dificuldade. Ao todo foram feitos 115 testes para avaliar o desempenho das abordagens utilizadas, essas que foram baseadas em força bruta e *backtracking*.

Os resultados mostraram que ambas as técnicas foram capazes de solucionar os problemas, no entanto, o *backtracking* otimizado se destacou em desempenho em relação à força bruta recursiva, pois apresentou o menor tempo médio de processamento, especialmente para instâncias complexas. Por outro lado, o outro algoritmo demonstrou ser funcional, ainda que com um custo computacional maior.

Com isso, conclui-se que o uso de técnicas de otimização são essenciais para melhorar o desempenho nas resoluções de jogos de Sudoku, pois ao explorar o espaço de soluções de forma mais inteligente, essas técnicas superam a abordagem de força bruta, o que torna as resoluções de jogos mais complexos viável em tempos computacionais mais razoáveis.

Para trabalhos futuros recomenda-se explorar heurísticas avançadas, assim como aprendizado de máquina, visando aprimorar ainda mais a eficiência na resolução dos jogos de Sudoku.

References

[1] J. R. Coelho, "Análise matemática de jogos: Soluções, estratégias e aplicações," Monografia apresentada ao Instituto de Matemática, Estatística e Computação Científica da Universidade Estadual de Campinas, Campinas, SP, 2024.

[2] P. M. Martins and J. Picado, "Existe um Sudoku com 16 pistas?," CMAT, Departamento de Matemática e Aplicações, Universidade do Minho, e CMUC, Departamento de Matemática, Universidade de Coimbra, [Online]. Acesso em: 16/01/2025.

[3] J.-P. Delahaye, "The Science behind Sudoku," *Scientific American*, vol. 294, no. 6, pp. 1–8, 2006.

[4] D. Job and V. Paul, "Recursive Backtracking for Solving 9×9 Sudoku Puzzle," *Bonfring International Journal of Data Mining*, vol. 6, no. 1, pp. 1–6, Jan. 2016.

[5] S. Chatterjee, S. Paladhi, and R. Chakraborty, "A Comparative Study on the Performance Characteristics of Sudoku Solving Algorithms," *IOSR Journal of Computer Engineering (IOSR-JCE)*, vol. 16, no. 5, ver. VI, pp. 69–77, Sep.–Oct. 2014.

[6] R. Rao, "Sudoku - A Collection of 1M+ Puzzles," Kaggle, 2019. [Online]. Available: https://www.kaggle.com/datasets/rohanrao/sudoku. [Acesso: 20-Jan-2025].

[7] Easybrain, "Sudoku.com - Free Online Sudoku Puzzles," 2024. [Online]. Disponível em: https://sudoku.com/. [Acesso: 23-Jan-2025].

[8] M. O. da Conceição, “Repositorio para os códigos utilizados na resolução dos jogos de Sudoku,” *GitHub*. Disponível em: https://github.com/MelissaOliveirac/sudoku