



UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL REI
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO - DCOMP
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO
UNIDADE CURRICULAR: PROJETO E ANÁLISE DE ALGORITMOS
DOCENTE: LEONARDO CHAVES DUTRA DA ROCHA

RELATÓRIO DE PROJETO E ANÁLISE DE ALGORITMOS

Este relatório apresenta uma análise dos dados coletados em um desenvolvimento de um algoritmo de força bruta e uma outra versão utilizando uma heurística própria para a empresa Xinguiling Entertainment Games Inc. .

BERNARDO MAIA DETOMI - 202050054
BRENO ESTEVES DOS SANTOS - 222050055

RELATÓRIO DE PROJETO E ANÁLISE DE ALGORITMOS:

Relatório apresentado à unidade curricular
projeto e análise de algoritmos do
curso de Ciência da Computação, da
Universidade Federal de São João del-Rei.

Prof. Dr. Leonardo Chaves Dutra da Rocha

1 Introdução

Sudoku é um quebra-cabeça lógico que desafia os jogadores a preencherem uma grade 9x9 com números de 1 a 9, respeitando três restrições fundamentais: os números não podem se repetir nas linhas, colunas e subgrades de 3x3. Criado na década de 1970, o jogo ganhou popularidade global no início dos anos 2000 e, desde então, tornou-se um dos mais amados e desafiadores problemas lógicos do mundo.

Embora seja amplamente conhecido como um passatempo, a resolução de Sudokus também tem aplicações em problemas reais. Sua natureza NP-completa o torna um campo de estudo relevante para algoritmos que lidam com problemas de otimização, como planejamento logístico, alocação de recursos e soluções de quebra-cabeças em sistemas embarcados. Além disso, sua popularidade resultou no desenvolvimento de aplicativos de resolução automática, plataformas digitais e campeonatos, demonstrando a conexão entre jogos matemáticos e avanços tecnológicos.

Resolver um Sudoku é um problema que pode variar de simples para muito complexo, dependendo da quantidade de células preenchidas inicialmente e de suas disposições. Este problema apresenta uma rica aplicação em computação, especialmente na área de algoritmos, pois permite explorar diferentes abordagens de solução, como força bruta e heurísticas otimizadas. Estas abordagens ilustram como é possível balancear simplicidade, eficiência computacional e recursos necessários.

Neste trabalho, dois algoritmos foram desenvolvidos para resolver instâncias de Sudoku: o primeiro utiliza um método de força bruta baseado em backtracking, enquanto o segundo aplica uma heurística que prioriza células com menos possibilidades válidas (“Minimum Remaining Values” - MRV). O objetivo principal é analisar a eficiência de ambas as abordagens, avaliando seus tempos de execução e comportamento em diferentes cenários de entrada. Além disso, este trabalho busca destacar como a escolha de um algoritmo adequado pode influenciar a resolução de problemas computacionais, proporcionando uma compreensão mais profunda dos paradigmas de projeto de algoritmos.

Através desta documentação, detalharemos a implementação dos algoritmos, a análise de suas complexidades, os resultados dos testes realizados e as conclusões obtidas. Este estudo não só reforça a importância do entendimento teórico, como também evidencia os desafios práticos na implementação de soluções para problemas NP-completos.

2 Estrutura do Código

Para abordar o problema de forma organizada e eficiente, é essencial uma boa estruturação do código. O programa segue um fluxo claro, descrito no fluxograma da Figura 1, que inclui leitura de entrada, representação do Sudoku, resolução do problema e geração de saída.

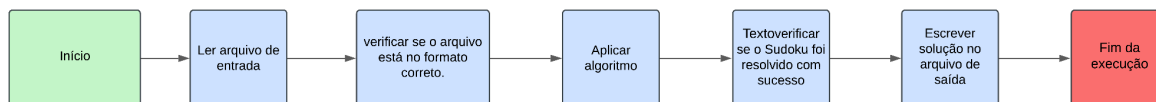


Figura 1: Fluxograma de linha do tempo do código.

2.1 Leitura de Arquivos

O programa utiliza um arquivo de entrada que representa o quebra-cabeça do Sudoku. Cada linha do arquivo contém os valores das células da grade, separados por espaços, com zeros indicando células vazias. Um exemplo de entrada está ilustrado na Figura 2.

```
./backtracking entrada.txt saida.txt  
./heuristica entrada.txt saida2.txt
```

Figura 2: Exemplo da linha de comando para a execução.

A leitura é feita com a função `fscanf()`, que popula uma matriz 9x9 representando o estado inicial do jogo. Um arquivo de saída é gerado ao final do processo, contendo a solução do Sudoku.

2.2 Representação do Sudoku

O Sudoku é representado por uma matriz bidimensional de inteiros, onde cada posição (i, j) corresponde a uma célula da grade. Esta representação permite acessar e modificar os valores de forma eficiente durante o processo de resolução.

2.3 Algoritmos de Resolução

Para resolver o Sudoku, foram implementadas diferentes abordagens, incluindo:

2.3.1 Backtracking

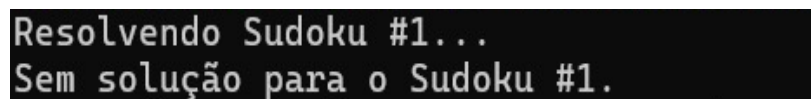
O algoritmo de backtracking é a solução principal, explorando todas as possibilidades de preenchimento de células vazias. O processo verifica recursivamente cada combinação, retornando ao estado anterior quando uma violação das regras é detectada.

2.3.2 Heurísticas de Busca

Heurísticas como o "Menor Domínio Primeiro" (MRV - Minimum Remaining Values) foram incorporadas para priorizar células com menos opções de preenchimento, reduzindo o espaço de busca e otimizando o tempo de execução.

2.3.3 Algoritmos de Resolução

A solução encontrada é gravada no arquivo de saída utilizando a função `fprintf()`. Caso o quebra-cabeça não possua solução, uma mensagem apropriada é registrada, como é mostrado na Figura 3.



```
Resolvendo Sudoku #1...  
Sem solução para o Sudoku #1.
```

Figura 3: Exemplo de mensagem apropriada para erro.

3 Estruturas de Dados

3.1 Matriz de Inteiros

A representação do Sudoku como uma matriz 9x9 permite acessar rapidamente células específicas, verificar valores existentes e validar as regras do jogo durante o preenchimento.

3.2- Listas de Opções

Listas dinâmicas foram utilizadas para rastrear as opções válidas para cada célula, atualizando-se conforme valores são atribuídos ou removidos.

4 Algoritmo de Resolução

4.1 Backtracking

O algoritmo de backtracking é composto pelos seguintes passos principais:

1. Encontrar a primeira célula vazia.
2. Para cada valor possível (1 a 9):
 - Verificar se o valor é válido na linha, coluna e bloco correspondente.
 - Preencher a célula e chamar recursivamente o algoritmo para resolver o restante do tabuleiro.
 - Se nenhuma solução for encontrada, desfazer a atribuição (backtrack).
3. Retornar verdadeiro se uma solução for encontrada; caso contrário, retornar falso.

4.1 Heurística MRV

O algoritmo de heurística baseado em MRV (Minimum Remaining Values) segue os seguintes passos principais:

1. Encontrar a célula com menos possibilidades válidas (MRV):
 - Percorrer o tabuleiro e identificar a célula vazia com o menor número de valores possíveis, priorizando a resolução das células mais restritas.
2. Para cada valor possível na célula selecionada:
 - Verificar se o valor é válido na linha, coluna e bloco correspondente.
 - Preencher a célula com o valor e chamar recursivamente o algoritmo para resolver o restante do tabuleiro.
 - Caso nenhuma solução seja encontrada, desfazer a atribuição (backtrack).
3. Retornar verdadeiro se uma solução for encontrada; caso contrário, retornar falso.

5 Análise de complexidade

O objetivo é detalhar as complexidades de tempo e espaço dos algoritmos implementados, considerando os piores casos, melhores casos e casos médios. O foco será nas funções principais: `solve_sudoku` (backtracking) e `heuristic_solve` (com heurística MRV).

5.1 Análise do algoritmo Backtracking (solve_sudoku)

Descrição do Algoritmo

- O algoritmo percorre o tabuleiro em busca da próxima célula vazia.
- Para cada célula vazia, tenta números de 1 a 9.
- Valida cada tentativa utilizando a função `is_valid`.
- Caso a tentativa seja válida, avança para a próxima célula recursivamente.
- Se nenhuma tentativa funcionar, retorna ao estado anterior (backtrack).

Complexidade de Tempo

- Validação (`is_valid`):
 - Para cada tentativa de número, verifica:
 - Linha: $O(n)$
 - Coluna: $O(n)$
 - Subgrade 3x3: $O(\sqrt{n})^2 = O(n)$
 - Custo total para `is_valid`: $O(n)+O(n)+O(n)=O(n)$.
- Backtracking Recursivo:
 - No pior caso, cada célula vazia tem n possibilidades.
 - Supondo m células vazias:
 - O número total de estados explorados é n^m
 - Para cada estado, executa a validação $O(n)$.
 - Complexidade total: $O(n^m \cdot n)=O(n^{m+1})$.

Complexidade de Espaço

- O algoritmo utiliza a pilha de chamadas recursivas:
 - Profundidade máxima: m (número de células vazias).
 - Complexidade de espaço: $O(m)$.

5.2 Análise do algoritmo Heurística (heuristic_solve)

Descrição do Algoritmo

- Baseado no MRV, seleciona célula com o menor número de possibilidades antes de avançar.
- Reduz significativamente o número de estados analisados.
- A lógica restante é similar ao backtracking.

Complexidade de Tempo

- Escolha da melhor célula (`find_best_cell`):
 - Percorre todas as células para calcular as possibilidades.
 - Para n^2 células:

- Custo de count_possibilities para cada células: $O(n)$ (usa is_valid).
- Custo total: $O(n^3)$.
- Resolução com Heurística:
 - Reduz o número de estados analisados pela escolha ótima de células.
 - Se p é a profundidade média da pilha recursiva:
 - Complexidade total: $O(p \cdot n^3)$, onde $p < m$.

Complexidade de Espaço

- Similar ao backtracking, mas com overhead adicional para armazenar possibilidades:
 - Espaço total: $O(m+n^2)$, onde n^2 é o espaço para rastrear possibilidades.

5.3 Comparação prática

Backtracking:

- Tempo (Pior caso): $O(n^{m+1})$
- Espaço: $O(m)$

Heurística:

- Tempo (Pior caso): $O(p \cdot n^3)$, com $p < m$.
- Espaço: $O(m+n^2)$

5.4 Análise matemática baseada nos testes

Com base nos gráficos das figuras 4-9 e tabela:

- Para tabuleiros 9x9, o tempo médio foi drasticamente reduzido pela heurística.
- Para tabuleiros maiores (12x12 e 16x16), a heurística mostrou ganhos lineares em relação ao crescimento exponencial do backtracking.

Gráficos indicam:

- A heurística reduz a ordem de complexidade prática.
- O ganho é especialmente visível em tabuleiros maiores e mais complexos.

6 Teste de software

6.1 Verificação de solução

Foram testados puzzles de Sudoku de diferentes níveis de dificuldade para garantir a corretude das soluções geradas. Os resultados foram comparados com soluções conhecidas.

6.2 Medição de desempenho

O tempo de execução foi medido utilizando funções de temporização como `gettimeofday()`. Os testes mostraram que o algoritmo é eficiente para puzzles com solução única e um número moderado de células vazias.

À medida que aumentamos o tamanho do tabuleiro de Sudoku, o tempo necessário para resolver o problema tende a aumentar também. Isso ocorre porque, com um tabuleiro maior, há um número maior de células a serem preenchidas e uma maior quantidade de possibilidades a serem analisadas.

No gráfico, podemos observar que, conforme o tamanho do tabuleiro cresce, o tempo de resolução também se expande de forma proporcional. Isso acontece porque a complexidade do problema aumenta, exigindo mais cálculos, verificações e estratégias de resolução. O aumento do número de quadrados, linhas e colunas em um tabuleiro maior gera mais interações entre essas unidades, resultando em um maior tempo de processamento.

Esse comportamento é característico de problemas computacionais mais complexos, onde a dificuldade de encontrar a solução cresce à medida que o problema aumenta de escala. O gráfico ilustra bem essa relação, mostrando que a solução de Sudoku em tabuleiros maiores exige mais tempo, conforme se torna mais desafiador resolver as interações entre as células.

6.3 Medição de tempo

O jogo de Sudoku é resolvido utilizando duas abordagens: *backtracking* e a heurística *MRV* (Most Restricted Variable). O *backtracking* é uma técnica de força bruta que tenta todas as possibilidades para preencher o tabuleiro, voltando atrás quando encontra uma solução inválida. Já a heurística *MRV* é aplicada para escolher a célula com menos opções disponíveis, priorizando as variáveis mais restritas, o que reduz o espaço de busca e acelera a resolução. Combinando essas técnicas, o jogo é resolvido de forma mais eficiente, especialmente em tabuleiros maiores. Nas figuras a seguir veremos os resultados.

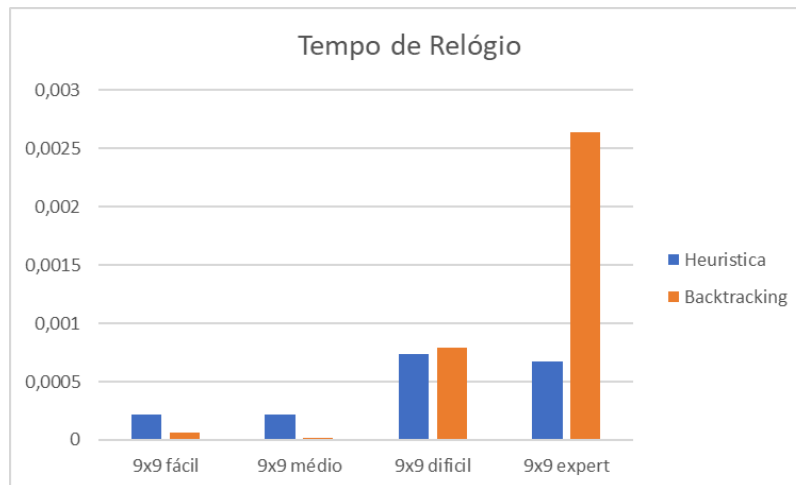


Figura 4: Gráfico de tempo de relógio 9x9.

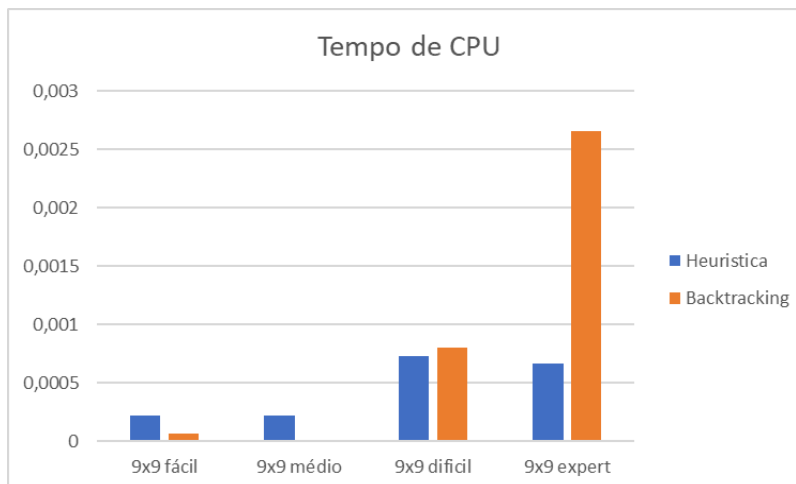


Figura 5: Gráfico de tempo de CPU 9x9.

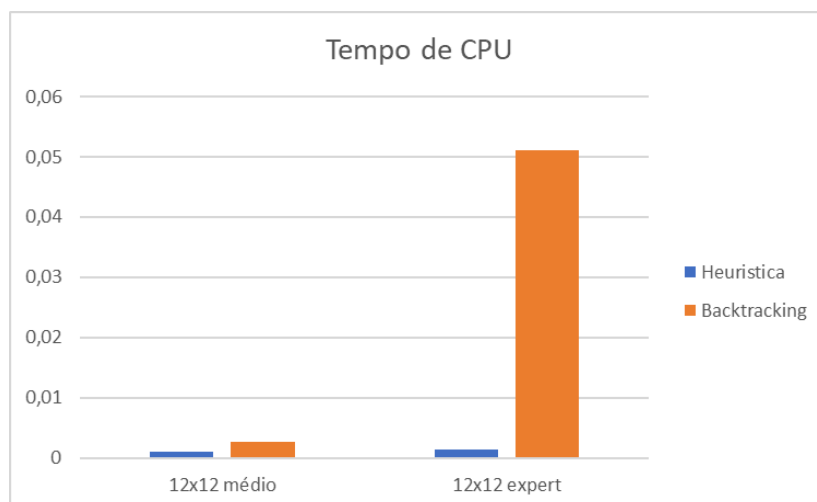


Figura 6: Gráfico de tempo de relógio 12x12.



Figura 7: Gráfico de tempo de CPU 12x12.



Figura 8: Gráfico de tempo de relógio 16x16.

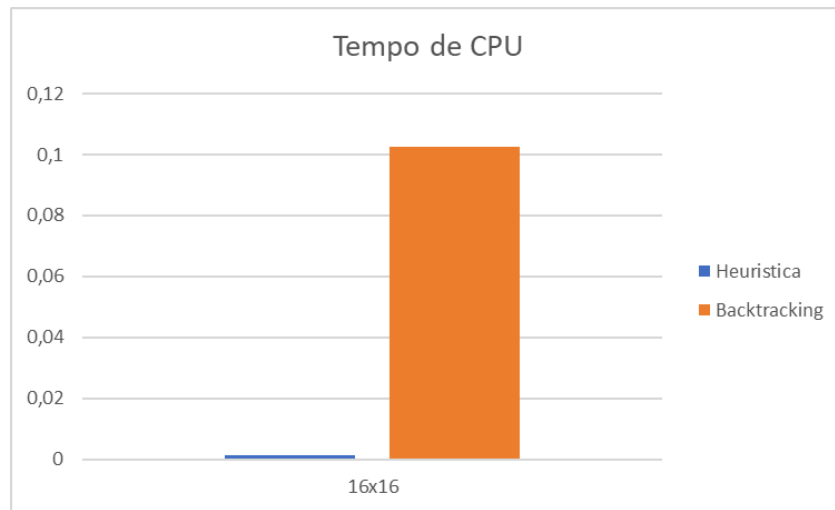


Figura 9: Gráfico de tempo de CPU 16x16.

Tempo em segundos	Tempo Relógio Backtracking	Tempo Relógio Heurística	Tempo CPU Backtracking	Tempo CPU Heurística
9x9 fácil	0,000063	0,000222	0,000068	0,000215
9x9 médio	0,000013	0,000217	0,000013	0,000215
9x9 difícil	0,000792	0,000735	0,000797	0,000728
9x9 expert	0,002641	0,000674	0,002656	0,000668
12x12 médio	0,0026	0,001091	0,002614	0,001082
12x12 expert	0,050848	0,001378	0,051116	0,001368
16x16	0,103229	0,001547	0,102574	0,001556

Tabela de representação de todos os tempos referente ao tamanho do tabuleiro de Sudoku.

7- Conclusão

Este trabalho demonstrou a complexidade envolvida na resolução de Sudokus, um problema aparentemente simples, mas que desafia os limites da computação devido ao seu caráter NP-completo. A implementação de dois algoritmos diferentes permitiu explorar abordagens contrastantes: o backtracking puro, com sua simplicidade e exaustividade, e o método heurístico, que emprega uma otimização inteligente para reduzir o espaço de busca.

Os testes realizados evidenciaram as limitações do algoritmo de força bruta, que se torna ineficiente à medida que a complexidade do Sudoku aumenta. Por outro lado, a heurística MRV provou ser uma solução viável para instâncias mais complexas, alcançando tempos de execução significativamente menores. Essa diferença reflete a importância de selecionar algoritmos adequados ao contexto do problema.

Além disso, o estudo ressalta a relevância da análise de desempenho na escolha de soluções computacionais. Entender as vantagens e limitações de diferentes abordagens é essencial para desenvolver soluções eficazes e escaláveis, especialmente em aplicações práticas onde o tempo de processamento é um recurso crítico.

Por fim, este trabalho reforça o valor do aprendizado de algoritmos e estruturas de dados no campo da computação, proporcionando uma base sólida para solução de problemas complexos e para avanços em diversas áreas tecnológicas. A resolução de Sudokus, embora seja um exemplo específico, destaca princípios aplicáveis a muitos desafios reais enfrentados por profissionais da computação.

Referências

Thomas H. Cormen. *Algoritmos: Teoria e Prática*. LTC, 2012.

Jayme L. Szwarcfiter and Lilian Markenzon. *Estruturas de Dados e Seus Algoritmos*, 3ª edição. GEN, 2015.

Documentação da função getopt em C:

https://www.gnu.org/software/libc/manual/html_node/Getopt.html.

Orunmila. "Modular code and how to structure an embedded C project." Disponível em: <https://www.microforum.cc/blogs/entry/46-modular-code-and-how-to-structure-an-embedded-c-project/>.

Nishtha Thakur. "getopt() function in C to parse command line arguments." Disponível em: <https://www.tutorialspoint.com/getopt-function-in-c-to-parse-command-line-arguments>.