Relatório Busca Heurística

Aplicação de Heurísticas na Resolução de Sudoku

Autores: Dayany Lima, Deise Santana e Jefeson Martins

25 de novembro de 2024

Sumário

1 Introdução

O Sudoku é um jogo de lógica matemática que foi bastante popular no Japão em 1986 e se popularizou pelo mundo a partir de 2005, ele contém um tabuleiro (matriz ou grade) de tamanho $n \times n$, onde cada célula pode ser preenchida com números de 1 a n, formando n sub-grades (blocos) de tamanho $k \times k$, com $k = \sqrt{n}$.

Algumas células começam preenchidas e não pode ser alteradas, esse preenchimento inicial determina o nível de dificuldade do jogo, tornando-o mais fácil ou mais difícil 1. O objetivo final é que o jogador preencha todo o tabuleiro de forma que em nenhuma das linhas, colunas ou sub-grades $k \times k$ tenha repetição de um número 2.

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

Figura 1: Tabuleiro Inicial

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

Figura 2: Tabuleiro Resolvido

2 Definição do Problema

O Sudoku escolhido é o de 9×9, que contém 9 sub-grades 3×3. Este formato é considerado um "clássico", sendo conhecido por todo o mundo, com uma vasta quantidade de instâncias disponíveis para estudo. Foi levado em consideração Sudokus que têm somente uma solução e, para garantir essa restrição, pelo menos 17 células precisam estar preenchidas no tabuleiro [4].

A Figura 3 ilustra o processo de resolução de um Sudoku. Na grade à esquerda, tendo em vista a primeira subgrade, identificamos que a célula *(a)* é a única posição válida para o número 3, pois qualquer outra escolha infringiria as regras do jogo. De maneira semelhante, na grade à direita, ao considerar a segunda subgrade, fica claro que a célula *(b)* é o único local possível para o número 7, garantindo o cumprimento das restrições do Sudoku.

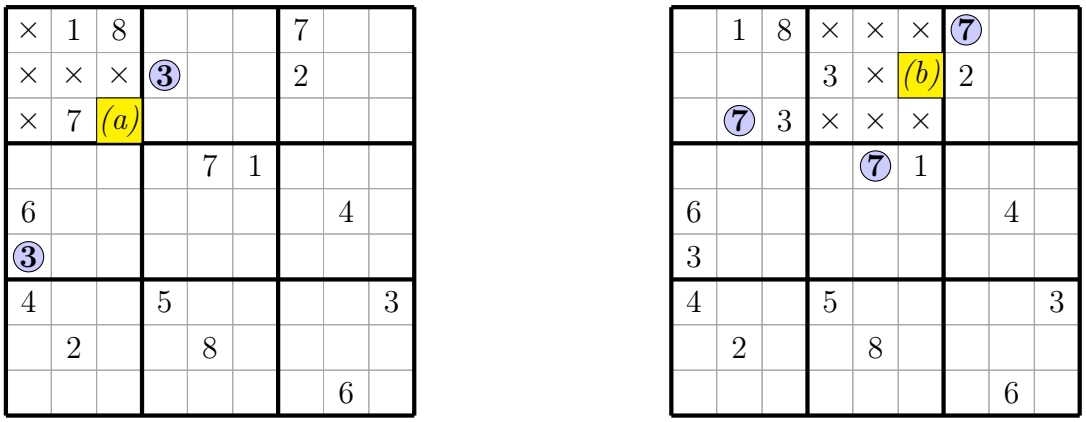

Figura 3: Resolução Sudoku

O tema foi escolhido devido à sua relevância em diversas áreas da Ciência da Computação, especialmente em contextos que demandam o uso de algoritmos heurísticos. O estudo do Sudoku tem gerado inúmeras pesquisas e ferramentas relacionadas à otimização, destacando sua importância acadêmica e prática.

3 Revisão Bibliográfica

3.1 Complexidade Computacional e NP-completude

Provar a complexidade computacional de um problema é fundamental para determinar a abordagem mais adequada para sua solução. O Sudoku, sendo um problema amplamente conhecido e utilizado em aplicações de heurísticas, tem sido objeto de diversos estudos que visam identificar a classe de complexidade à qual ele pertence. Uma das pesquisas mais importantes nessa área foi realizada por Yato and Seta (2003), que provou que o Sudoku é um problema NP-completo por meio da redução polinomial do problema do quadrado latino.

Um quadrado latino é uma matriz de ordem $n \times n$, preenchida com n elementos, onde cada elemento aparece exatamente uma vez em cada linha e em cada coluna. O Sudoku 9×9 pode ser visto como uma versão especializada de um quadrado latino, com uma restrição adicional: além de garantir que os elementos não se repitam nas linhas e colunas, também é necessário que não haja repetição de elementos nas subgrids 3×3. Como o problema do quadrado latino é NP-completo [1], e dado que ele pode ser reduzido polinomialmente ao Sudoku, conclui-se que o Sudoku também pertence à classe dos problemas NP-completos.

3.2 Sudoku como um problema de restrição

O Sudoku pode ser modelado como um problema de programação por restrições, com o objetivo de preencher um tabuleiro 9×9 com números de 1 a 9, respeitando restrições de unicidade em linhas, colunas e blocos 3×3. Na pesquisa de Simonis (2005), o Sudoku é tratado como um problema de programação por restrições, utilizando modelagem para resolver o quebra-cabeça sem o uso de busca. O autor faz uma comparação com o problema de *quasi-group completion*, um problema algébrico explorado no contexto da programação por restrições.

A pesquisa explora o uso de restrições *alldifferent*, que garante que os números sejam distintos em cada linha, coluna e bloco 3×3. A uso desse tipo de restrição em blocos maiores aumenta as possibilidades de identificar restrições redundantes, o que melhora o processo de solução. O estudo também apresenta técnicas como o *shaving*, que elimina valores inconsistentes das variáveis, permitindo resolver quebra-cabeças mais complexos sem busca.

Além disso, as técnicas propostas podem ser aplicadas na geração de quebra-cabeças com níveis específicos de dificuldade, baseadas em abordagens do *quasi-group completion*. O Sudoku, portanto, não só se torna uma aplicação útil para a programação por restrições, mas também uma ferramenta para aumentar o interesse por essa área de estudo.

3.3 Sudoku na Inteligência Artificial

O Sudoku é um puzzle muito utilizado como um caso de estudo na aplicação de técnicas de Inteligência Artificial (IA), pois proporciona um ambiente eficaz para testar, otimizar e avaliar algoritmos voltados para a solução de problemas. O estudo de Mantere and Koljonen (2007) explora a aplicação de algoritmos genéticos (GA) no Sudoku. Os principais objetivos da pesquisa é testar a eficiência da GA na resolução de quebra-cabeças de Sudoku, investigar sua capacidade de gerar novos puzzles e avaliar sua eficácia como ferramenta de classificação de dificuldade. O autor adota como abordagem o uso de uma GA pura, sem a inclusão de muitas regras específicas para o problema, com a finalidade de avaliar a eficiência do algoritmo de forma geral.

Embora existam algoritmos mais rápidos, os resultados demonstram que a GA pode ser eficaz na classificação da dificuldade de quebra-cabeças de Sudoku, uma vez que seus resultados se alinham bem com as classificações de dificuldade encontradas em jornais. Além disso, o estudo descreve como a GA pode gerar novos quebra-cabeças, removendo números de uma solução previamente encontrada e testando a unicidade da solução. A pesquisa sugere que uma solução pode ser considerada única quando é repetidamente encontrada em sucessivas tentativas, e o software *SudokuExplainer* foi utilizado para confirmar isso.

A pesquisa também compara o desempenho da GA com outros métodos de otimização, mostrando que, em algumas situações, a GA superou outros algoritmos evolutivos. O estudo conclui que futuras investigações poderiam explorar a combinação da GA com algoritmos mais específicos para Sudoku, bem como investigar métodos para minimizar o número de pistas fornecidas, garantindo a unicidade da solução.

4 Descrição da Heurística

O algoritmo utilizado para a resolução do problema foi o *Backtracking*1 em conjunto com as heurísticas de *Minimum Remaining Values*2 e *Candidate Reduction*3.

O *Backtracking* é uma abordagem poderosa para resolver problemas complexos, pois explora diferentes combinações até encontrar uma solução. Quando combinado com heurísticas, como as mencionadas, ele se torna ainda mais eficiente, já que essas estratégias ajudam a reduzir o espaço de busca e a acelerar o processo.

Com essa combinação, é possível garantir que, se existir uma solução válida, ela será encontrada de forma mais rápida e otimizada.

Algorithm 1 Backtracking Genérico

```
1: Função backtracking(estadoAtual)
2: if estadoAtual é uma solução then
3:    Exibir solução
4:    return verdadeiro
5: end if
6: for cada escolha em escolhasPossíveis no estadoAtual do
7:    if escolha é válida then
8:       Faça a escolha (atualize o estado)
9:       if ResolverProblema(estadoAtual) then
10:         return verdadeiro
11:      end if
12:      Desfaça a escolha (volte ao estado anterior)
13:   end if
14: end for
15: return falso
```

4.1 Minimum Remaining Values

A heurística *Minimum Remaining Values*, ou "Valores Restantes Mínimos", prioriza a seleção de variáveis com o menor número de opções disponíveis em seu domínio. Essas variáveis são consideradas mais restritivas e, consequentemente, mais suscetíveis a gerar conflitos futuros. Ao adotar essa abordagem, é possível antecipar e resolver inconsistências de forma eficiente, otimizando o processo de busca e reduzindo o espaço explorado pelo algoritmo. No contexto do Sudoku, essa estratégia foca na atribuição de valores às células com o menor número de possibilidades, acelerando a resolução do quebra-cabeça.

Algorithm 2 Minimum Remaining Values

Require: sudoku: Estrutura contendo as células e suas possibilidades
Ensure: Lista de células ordenadas pelo menor número de valores restantes

```
1: referenciaValoresRestantes ← lista vazia
2: for i = 0 to sudoku.linhas − 1 do
3:    for j = 0 to sudoku.colunas − 1 do
4:       if sudoku.possibilidades[i][j] ≠ ∅ then
5:          Adicionar (i, j, sudoku.possibilidades[i][j]) à referenciaValoresRestantes
6:       end if
7:    end for
8: end for
9: {Ordenar células com base no número de valores restantes}
10: Ordenar referenciaValoresRestantes de forma crescente pelo tamanho de
    possibilidades
11: return referenciaValoresRestantes
```

4.2 Candidate Reduction

A heurística *Candidate Reduction* ou "Redução de Candidatos" busca reduzir o conjunto de candidatos em um espaço de busca, melhorando a eficiência em processos como classificação, otimização ou busca em grandes bases de dados, através de algumas estratégias. No contexto do Sudoku, cada célula do tabuleiro possui um conjunto de possíveis valores que podem ser atribuídos, chamados de candidatos. A *Candidate Reduction* atua na diminuição desse conjunto de candidatos, eliminando valores impossíveis à medida que mais células são preenchidas, combinado com o *Backtracking*, a sua eficiência é aumentada, já que minimiza a necessidade de fazer exploração de soluções desnecessárias.

Algorithm 3 candidateReduction

Require: row, col, sudoku {Entrada: linha, coluna, Sudoku}
Ensure: Lista de candidatos válidos para a célula na posição (`row`, `col`)

candidatos ← lista vazia
sqrtRows ← $\sqrt{\texttt{sudoku.linhas}}$
sqrtCols ← $\sqrt{\texttt{sudoku.colunas}}$
usados ← vetor de booleanos inicializado como `falso` com tamanho (`sudoku.linhas`+ 1)
{Marcar números já usados na linha}
for c ← 0 até `sudoku.colunas` − 1 **do**
  if `sudoku.celulas[row][c]` ≠ 0 **then**
    usados[sudoku.celulas[row][c]] ← verdadeiro
  end if
end for
{Marcar números já usados na coluna}
for r ← 0 até `sudoku.linhas` − 1 **do**
  if `sudoku.celulas[r][col]` ≠ 0 **then**
    usados[sudoku.celulas[r][col]] ← verdadeiro
  end if
end for
{Marcar números já usados no subgrid}
startRow ← (`row` ÷ `sqrtRows`) · `sqrtRows`
startCol ← (`col` ÷ `sqrtCols`) · `sqrtCols`
for r ← `startRow` até `startRow` + `sqrtRows` − 1 **do**
  for c ← `startCol` até `startCol` + `sqrtCols` − 1 **do**
    if `sudoku.celulas[r][c]` ≠ 0 **then**
      usados[sudoku.celulas[r][c]] ← verdadeiro
    end if
  end for
end for
{Adicionar números não usados à lista de candidatos}
for num ← 1 até `sudoku.linhas` **do**
  if `usados[num]` = `falso` **then**
    Adicionar `num` à candidatos
  end if
end for
return candidatos

5 Resultados

Heurística	Instâncias	Menor tempo(ms)	Maior tempo(ms)	Média(ms)
Backtracking puro	95	0.0696	12728.2057	450.72650105
BackTracking + Minimum Remaining Values	95	0.2167	16111.6688	1061.50412105
Backtracking + Candidate Reduction	95	0.0963	9313.6346	331.97826526

Tabela 1: Comparação de heurísticas para resolução de Sudoku.

Resultados de cada técnica aplicada:

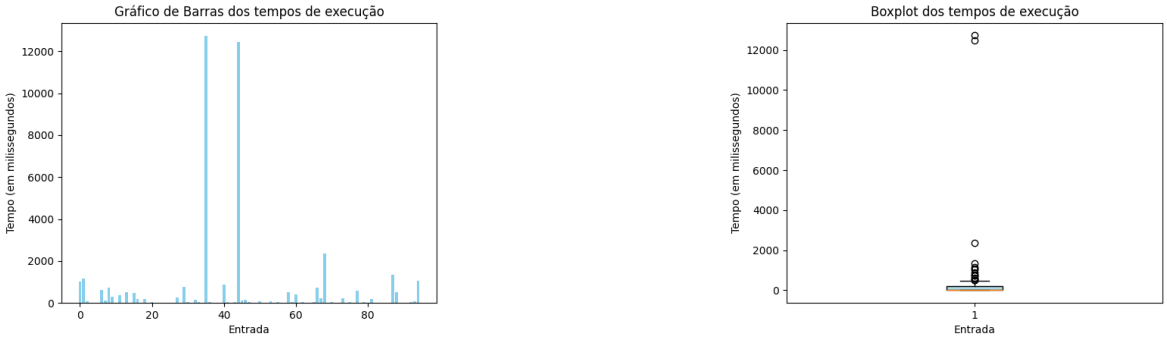

Figura 4: Backtracking Puro (barras e boxplot)

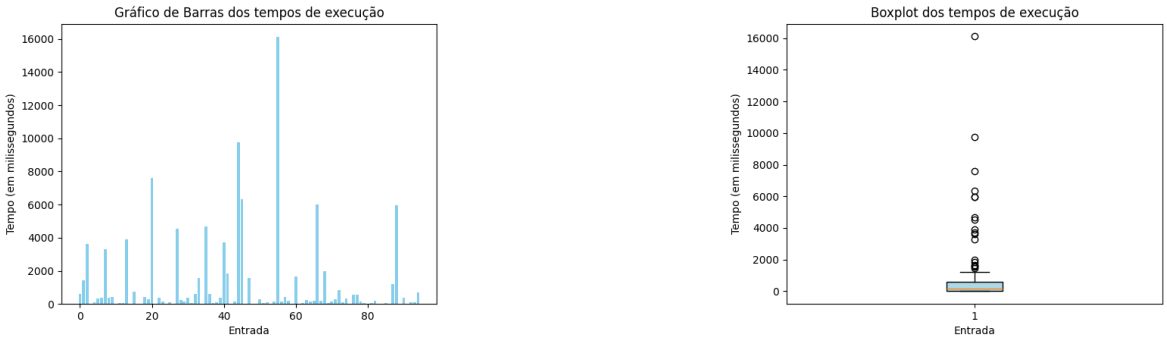

Figura 5: Backtracking + Minimum Remaining Values (barras e boxplot)

Figura 6: Backtracking + Candidate Reduction (barras e boxplot)

6 Conclusão

O algoritmo *backtracking* demonstrou ser eficiente sem a combinação com heurísticas, especialmente em instâncias menores, como os puzzles de 9×9. No entanto, à medida que o tamanho do tabuleiro aumenta, a aplicação de heurísticas poderia potencialmente melhorar ainda mais a desempenho do algoritmo.

Infelizmente, há uma escassez de um conjunto adequado de instâncias para realizar testes com tamanhos maiores, impossibilitando que fosse testado os algoritmos escolhidos em situações mais complexas.

Para trabalhos futuros, seria interessante desenvolver um estudo utilizando instâncias de tabuleiros maiores (com $n \geq 4$) aplicando as heurísticas abordadas, e comparar a eficiência entre as duas.

Com a possibilidade de um trabalho que crie essas instâncias e outro com a aplicação das mesmas. Isso implicaria na melhora da seção de resultados do trabalho desenvolvido nesse relatório, fornecendo uma análise mais robusta da aplicação dos algoritmos escolhidos.

Referências

[1] Charles J Colbourn. The complexity of completing partial latin squares. *Discrete Applied Mathematics*, 8(1):25–30, 1984.

[2] Joël Ouaknine Inês Lynce. Sudoku as a sat problem. In *AI&M*, 2006.

[3] Timo Mantere and Janne Koljonen. Solving, rating and generating sudoku puzzles with ga. In *2007 IEEE congress on evolutionary computation*, pages 1382–1389. IEEE, 2007.

[4] Gary McGuire, Bastian Tugemann, and Gilles Civario. There is no 16-clue sudoku: Solving the sudoku minimum number of clues problem via hitting set enumeration. *Experimental Mathematics*, 23(2):190–217, 2014. doi: 10.1080/10586458.2013.870056. URL `https://doi.org/10.1080/10586458.2013.870056`.

[5] Helmut Simonis. Sudoku as a constraint problem. *CP Workshop on Modeling and Reformulating Constraint Satisfaction Problems*, pages 13–27, 2005.

[6] Kevin Takano, Rosiane de Freitas, and Vinícius de Sá. O jogo de lógica sudoku: modelagem teórica, np-completude e estratégias algorítmicas exatas e heurísticas. In *Anais do XXXIV Concurso de Trabalhos de Iniciação Científica da SBC*, pages 71–80, Porto Alegre, RS, Brasil, 2015. SBC. URL `https://sol.sbc.org.br/index.php/ctic/article/view/10020`.

[7] Takayuki Yato and Takahiro Seta. Complexity and completeness of finding another solution and its application to puzzles. *IEICE transactions on fundamentals of electronics, communications and computer sciences*, 86(5):1052–1060, 2003.