Solucionando Sudoku como PSR

Guilherme Zamberlam Pomini[1]

[1]Departamento de Informática – Universidade Estadual de Maringá (UEM)
Maringá – PR – Brasil

ra99345@uem.br

Resumo. *Este artigo visa a modelagem e implementação do puzzle Sudoku utilizando técnicas Inteligência Artificial. O artigo trata o Sudoku como um Problema de Satisfação de Restrição (PSR) e utiliza heurísticas para reduzir a árvore de busca do problema. A implementação proposta é comparada com a abordagem de força bruta e confirmada como superior computacionalmente.*

1. Sudoku

Sudoku é um jogo de quebra-cabeça baseado em colocação lógica de números em um tabuleiro. Tradicionalmente o Sudoku é jogado em um tabuleiro com 81 células divididas em 9 linhas, 9 colunas e 9 blocos. O jogo tem como objetivo preencher o tabuleiro de forma que nenhuma linha, coluna ou bloco possua números repetidos, que variam de 1 a 9.

A resolução do Sudoku é feita através de um estado inicial do tabuleiro parcialmente preenchido de forma que seja possível induzir os próximos passos a partir do estado existente.

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) Tabuleiro Sudoku Inicial

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

(b) Tabuleiro Sudoku Resolvido

Figura 1. Exemplo de Sudoku 9x9

Computacionalmente resolver um Sudoku de ordem N^2xN^2 é um problema NP-Completo, não havendo uma solução em tempo polinominal conhecida. Uma solução simples para um Sudoku é avaliar através de força bruta todas as possibilidades de preenchimento para as células vazias.

2. Problemas de Satisfação de Restrição

Um Problema de Satisfação de Restrição (PSR) é todo problema que possui um conjunto de variáveis, um conjunto de valores possíveis para as variáveis (o domínio de cada variável) e um conjunto de restrições, possuindo como objetivo encontrar um conjunto de atribuições que satisfaçam todas as restrições.

Tipicamente, um PSR possui como atributos básicos:

1. **Estado Inicial:** uma associação de valores às variáveis, podendo ser parcial ou total.
2. **Função Sucessor:** responsável por atribuir um valor a uma variável desde que não entre em conflito com as restrições.
3. **Teste de Objetivo:** verificar se o estado atual é uma solução do problema.
4. **Custo do Caminho:** constante para cada passo.

Os PSR podem ser resolvidos de várias formas, existindo inúmeras heurísticas e técnicas aplicáveis. No entanto, tipicamente um PSR é resolvido como um problema de busca, sendo as abordagens mais comuns são variações de Backtracking, Propagação de Restrições e Busca Local.

2.1. Aplicações de PSR

Problemas de PSR são amplamente utilizados no dia-a-dia já que grande parte dos problemas humanos possuem restrições. Exemplos podem ser dados em atividades de escalonamento e alocação de recursos, mapeamentos, divisão de elementos em conjuntos, entre muitos outros.

3. Sudoku visto como PSR

Tendo firmado a ideia de o que é um PSR e como funciona um Sudoku é fácil notar como os dois problemas se encaixam. Para um Sudoku qualquer a sua representação como PSR pode ser definida como:

- **Conjunto de Variáveis:** cada célula vazia do tabuleiro.
- **Domínio das Variáveis:** um número variando de 1 a 9 (no caso do sudoku 9x9)
- **Conjunto de Restrições:**
    1. Não pode haver números repetidos nas linhas
    2. Não pode haver números repetidos nas colunas
    3. Não pode haver números repetidos nos blocos
- **Estado:** uma das possíveis atribuições de valores para as células
- **Solução:** um tabuleiro totalmente preenchido que satisfaça as restrições

Definido como um Sudoku se encaixa em PSR resta a implementação das técnicas. A base da resolução continua sendo uma busca utilizando Backtracking, contudo aqui ela é uma busca ”inteligente”que reduz consideravelmente a árvore de busca através de heurísticas.

3.1. Escolha da Variável

Primeiramente deseja-se otimizar a escolha variável (uma célula vazia) que terá seus valores testados a cada chamada do algoritmo. Um algoritmo de força bruta normalmente segue sequencialmente a ordem do tabuleiro, partindo do canto superior direito até o fim do tabuleiro. É fácil ver como essa escolha de variáveis não leva em consideração a quantidade de valores disponível para cada variável ou as restrições em si.

Aplicando a Heurística da variável mais restrita deseja-se testar as variáveis que possuem a menor quantidade de valores no seu domínio primeiro, reduzindo a quantidade de nós da busca. A heurística leva em conta que faz mais sentido explorar uma árvore que possua menos galhos primeiro do que uma árvore maior.

A Figura 3.1 exemplifica a heurística, se a célula (1,1) for escolhida como variável atual a árvore de busca a partir dela terá 9 ramos, 1 para cada valor do domínio. No entanto, se a célula (3,3) for escolhida a quantidade de ramos será somente 4, reduzindo pela metade o espaço de busca.

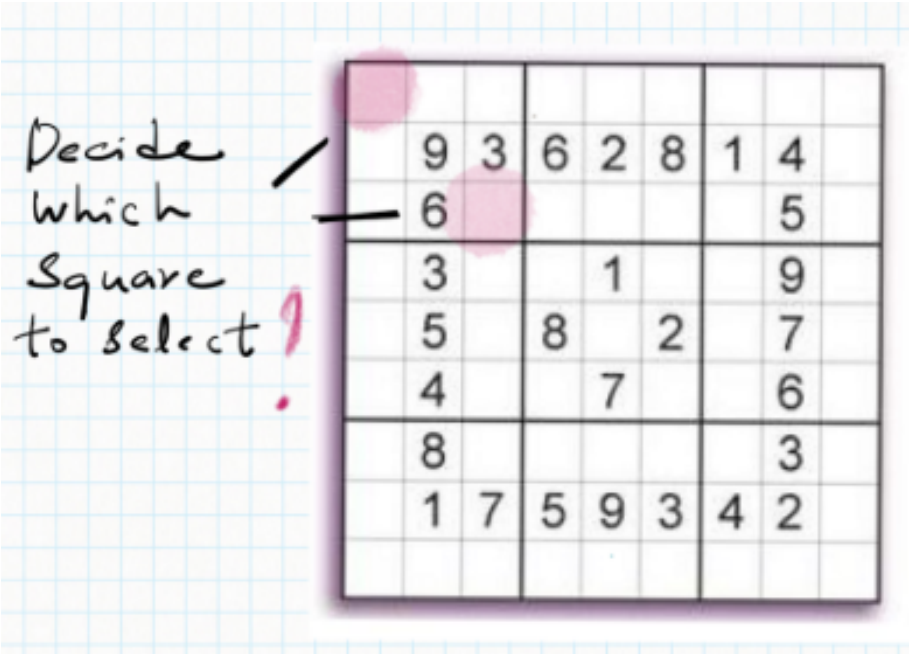

Figura 2. Escolha de variável no Sudoku

3.2. Escolha do Valor

Além da escolha da melhor variável para cada estado também é interessante uma escolha ótima do valor do domínio para cada variável. A partir da própria estrutra do Sudoku é possível notar que cada valor do domínio das variáveis em um sudoku 9x9 só aparecerá 9 vezes no tabuleiro.

Tendo em vista o máximo de vezes fixo que cada valor do domínio pode aparecer, se uma variável possui em seu domínio um valor que apareceu mais do que os outros é lógico induzir que a chance deste valor ser o correto para a variável é maior do que a dos outros valores. Utilizando a Heurística do valor mais restritivo, deseja-se diminuir a chance de falha na atribuição de uma variável, evitando galhos que não levem a solução.

4. Implementação

Como dito anteriormente, a base da implementação é um algoritmo de backtracking. A diferença é que para cada estado do problema, a variável escolhida será a que possui o maior número de restrições no seu domínio (no caso de haver mais de uma será escolhido uma aleatória entre as que possuem a maior restrição). A outra diferença será na ordem de atribuição dos valores, ao invés de seguir sequencialmente os valores do domínio, serão testados os valores mais restritos primeiro.

Tal implementação toma proveito da estrutra do jogo para propagar as restrições para as variáveis a cada iteração. Quando um valor é atribuido para uma variável, todas as outras que possuem uma dependência a variável tem seu domínio afetado, diminuindo o espaço de busca.

A implementação foi feita na linguagem Python, versão 3, com a utilização da biblioteca Random.

4.1. A Classe Sudoku

Foi feita uma classe que possui toda a implementação de um jogo de Sudoku da seguinte forma:

```
class Sudoku:
    def __init__(self,n):
        self.n = n
        self.tabuleiro = []
        self.preenchidos = 0
        self.feitos = []
        self.quantidade = []
        for i in range(n**2 + 1):
            self.quantidade.append(0)
```

Onde o valor **n** é a ordem do sudoku, **tabuleiro** é uma matriz que armazena os valores do tabuleiro, **preenchidos** guarda a quantidade de variáveis (ou células) que possuem um valor atribuido, **feitos** armazena as células já preenchidas de inicialização e **quantidade** armazena a quantidade de cada valor do domínio das variáveis presente no tabuleiro.

A classe ainda possui funções auxiliares para verificar se um valor está presente na linha, coluna ou bloco de uma célula:

```
def valorLinhaColuna(self, celula, valor):
        for x in self.tabuleiro[celula[0]]:
            if x == valor:
                return True
        for x in range(self.n ** 2):
            if self.tabuleiro[x][celula[1]] == valor:
                return True
        return False

    def valorBloco(self, celula, valor):
        linhaInicial = celula[0] - (celula[0] % self.n)
        colunaInicial = celula[1] - (celula[1] % self.n)
        for i in range(linhaInicial, linhaInicial + self.n):
            for j in range(colunaInicial, colunaInicial + self.n):
                if self.tabuleiro[i][j] == valor:
                    return True
        return False
```

E por fim as funções para retornar a lista de valores do domínio de uma variável e a próxima variável a ser escolhida com a maior quantidade de conflitos (menor quantidade de valores no domínio):

```
def valoresCelula(self, celula):
        aux = [x for x in range(1, self.n **2 + 1)]
        for x in range(1, self.n ** 2 + 1):
            if self.valorLinhaColuna(celula, x) or
```

```
            self.valorBloco(celula, x):
                aux.remove(x)
        return aux

    def proximaCelula(self):
        menorValor = self.n ** 2
        listaMenores = []
        for i in range(self.n ** 2):
            for j in range(self.n ** 2):
                if self.tabuleiro[i][j] == 0:
                    qtd = len(self.valoresCelula((i,j)))
                    if qtd == 0:
                        return False
                    if qtd < menorValor:
                        menorValor = qtd
                        listaMenores.clear()
                        listaMenores.append((i,j))
                    elif qtd == menorValor:
                        listaMenores.append((i,j))
        aux = random.randint(0, len(listaMenores) -1)
        return listaMenores[aux]
```

4.2. Algortimo PSR

A função de resolução do Sudoku com PSR foi implementada da seguinte forma:

```
def resolveSudokuPSR(sudoku, celula = None):
    if sudoku.preenchidos == sudoku.n ** 4:
        return True
    if celula == None:
        celula = sudoku.proximaCelula()
    valoresCelula = sudoku.valoresCelula(celula)
    valoresCelula.sort(key = lambda x: sudoku.quantidade[x],
        reverse = True)
    for x in valoresCelula:
        sudoku.tabuleiro[celula[0]][celula[1]] = x
        sudoku.preenchidos += 1
        sudoku.quantidade[x] += 1
        if sudoku.preenchidos == sudoku.n ** 4:
            return True
        proximaCelula = sudoku.proximaCelula()
        if proximaCelula != False:
            if(resolveSudokuPSR(sudoku, proximaCelula)):
                return True
        sudoku.tabuleiro[celula[0]][celula[1]] = 0
        sudoku.preenchidos -= 1
        sudoku.quantidade[x] -= 1
    return False
```

A função se baseia em recursão até que todas as células sejam preenchidas, a variável (célula) de cada estado é obtida pela função proximaCelula que devolve a variável mais restrita naquele estado.

Os valores do domínio da variável escolhida são filtrados para somente os valores possíveis, utilizando a propagação de restrições dos estados anteriores. Os valores do domínio são ordenados conforme os valores mais restritos de forma a minimizar a chance de escolher um valor errado.

5. Resultados

Foram feitos testes de resolução de diferentes Sudokus usando as abordagens PSR e de Força Bruta como forma de comparação. Para 20 tabuleiros de Sudoku diferentes com cada um sendo resolvido duas vezes, uma para cada abordagem, foi possível gerar o seguinte gráfico.

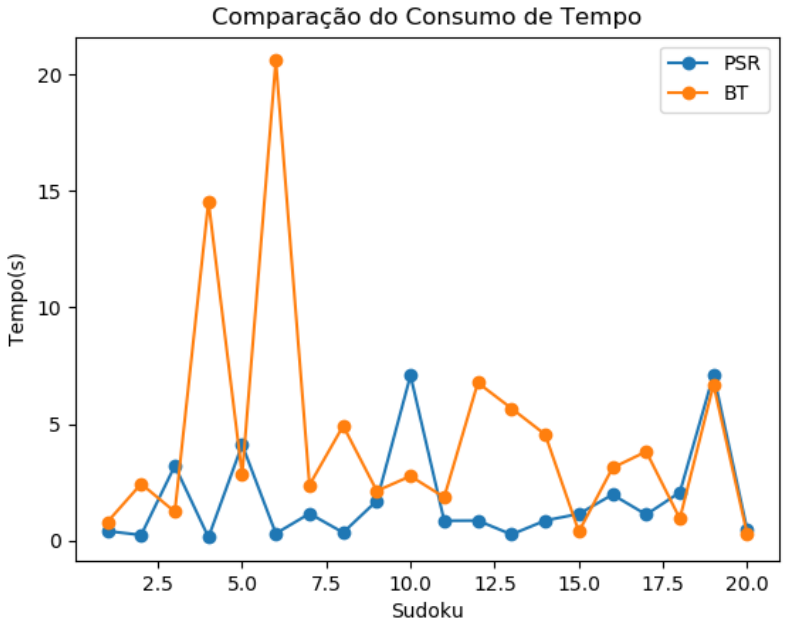

Figura 3. Grafico de Tempo de Execução

É possível notar que embora a abordagem PSR acabe não sendo superior em 100% dos casos, o seu ganho computacional é inegável. No exemplo da Figura 5 a abordagem PSR é superior em 13 dos 20 casos, tendo ganhos de até 200% quando comparado com a abordagem de força bruta.

6. Conclusão

Neste artigo foi exposta uma forma de implementar a resolução de um Sudoku tratando o jogo como um Problema de Satisfação de Restrição. Foram utilizadas heurísticas para uma seleção inteligente de cada célula e de cada valor do domínio da célula. Também foi feita uma implementação da resolução por força bruta como comparação.

A partir da implementação foi possível notar como a diferença na modelagem do problema faz uma grande diferença no tempo de execução. Pela modelagem de PSR diminuir a árvore de busca do problema, o custo computacional foi drasticamente reduzido quando comparado com a abordagem de força bruta padrão. É possível induzir que conforme a ordem do Sudoku aumenta, a diferença entre as duas abordagens aumentará ainda mais, com o ganho da modelagem PSR sendo ainda mais significativo.

Referências

[Haralick and Elliott 1980] Haralick, R. M. and Elliott, G. L. (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artificial intelligence*, 14(3):263–313.

[Kemisetti 2018] Kemisetti, A. (2018). Solving sudoku ... think constraint satisfaction problem.

[Kumar 1992] Kumar, V. (1992). Algorithms for constraint-satisfaction problems: A survey. *AI magazine*, 13(1):32–32.

[Seif 2017] Seif, G. (2017). Solving sudoku using a simple search algorithm.