FSM aplicada à resolução de Sudokus

Caio Graça Gomes
Instituito Tecnológico de Aeronáutica
São José dos Campos – SP, Brasil
caio.graca@gmail.com

***Resumo*—Esse projeto teve como objetivo implementar uma máquina de estados finita (*FSM*) que pudesse resolver sudokus das mais variadas dificuldades sem o uso de "força bruta", isto é, a inteligência artificial implementada não usa tentativa e erro para completar os sudokus, apenas inferências lógicas.**

Palavras-chave—máquina de estados finita, sudoku, inteligência artificial.

I. Introdução

O Sudoku é um jogo que se baseia na colocação lógica de números. No caso de um Sudoku 9x9, o jogador tem por objetivo colocar números de 1 a 9 em cada uma das células vazias de uma grade 9x9 dividida em 9 quadrados 3x3 (subgrades), de maneira a completar a grade de números.

O Sudoku vem com algumas pistas iniciais, que são números já preenchendo algumas das células da grade 9x9, estes números estão dispostos de tal forma que o Sudoku possuirá solução e esta será única. O preenchimento das células vazias com números de 1 a 9 deve ser feito de tal forma que não pode haver repetição de números:

- Em uma mesma linha da grade 9x9;
- Em uma mesma coluna da grade 9x9;
- Em um mesmo quadrado 3x3 (subgrade).

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

Fig. 1. Exemplo de um jogo de Sudoku 9x9.

Além disso, os Sudokus podem ter N^2xN^2 células vazias ($N \geq 2$), as regras do jogo para esse caso mais geral serão análogas às do 9x9.

Para este projeto, implementou-se uma máquina de estados para tentar resolver a maior parte dos sudokus existentes, para isso, a máquina de estados simulará o comportamento de um humano (com certa experiência em Sudokus) ao se deparar com esse quebra-cabeça. Os estados serão ações a ser tomadas sobre o sudoku, que mudarão para outros estados se não conseguiu realizar nada sobre a grade ou se consegui realizar.

Vale salientar que a FSM resolve a grande maioria, mas não é capaz de resolver todos os sudokus possíveis, em alguns casos bem improváveis seria necessário implementar uma lógica mais profunda, mas na maioria dos sudokus que não são resolvidos pela FSM, é fundamental e estritamente necessário o uso de tentativa e erro, o que foi confirmado pelo autor ao fazer alguns destes Sudokus manualmente. Esses casos não resolvíveis são geralmente considerados de dificuldade extrema ou muito difícil pelas revistas/aplicativos de Sudoku.

Ademais, é possível demonstrar que para um Sudoku 9x9 possuir solução e esta ser única, o número mínimo de pistas iniciais é 17, esses casos serão testados na FSM

II. Metodologia

Para a implementação do Código, inicialmente criou-se a classe `class Sudoku()` em `sudoku.py`, que armazena informações do *grid* do Sudoku e possui algumas funções a serem aplicadas sobre a grade do Sudoku nos estados.

Após isso, fez-se a FSM em `state_machine.py`, que contém as ações que o Código deve tomar para resolver o Sudoku em ordem decrescente de probabilidade da necessidade do uso delas.

Por fim, usou-se um gerador de sudokus implementado pelo sueco Kjell Ericson e disposível na internet [2] para testar a implementação.

A. *Implementação da classe Sudoku*

Primeiramente, a classe Sudoku receberá em sua função `_init_()` dois argumentos serão eles:

- `starting_grid`: que contém o *grid* inicial do Sudoku a ser resolvido;
- `behavior`: que recebe o comportamento inicial da máquina de estados, no caso, o `Fill_Possibilities_State`.

Além disso, é de interesse armazenar informações do tamanho do sudoku (definindo o `self.dimension` e o `self.type`) e outros atributos que virão a ser utilizados na máquina de estados, são eles:

- `self.possibilities`: Armazena todos os números que são possíveis de estar em cada uma das células do *grid*, considerando o conhecimento atual;
- `self.possibilities_line`: Armazena os números que são possíveis em cada uma das linhas de cada um dos quadrados (*subgrids*), considerando o conhecimento atual;
- `self.possibilities_column`: Armazena os números que são possíveis em cada uma das colunas de cada um dos quadrados (*subgrids*), considerando o conhecimento atual;

Ademais, a classe possui algumas funções que serão bastante úteis durante a máquina de estados são elas:

XXX-X-XXXX-XXXX-X/XX/$XX.00 ©20XX IEEE

- `square_of_the_cell(self, cell)`: Retorna a posição do quadrado (*subgrid*) de uma determinada célula no grid;
- `is_number_valid(self, number, cell)`: Verifica se determinado número é válido em uma determinada célula, para isto, apenas observa se o número em questão já está na linha, coluna ou *subgrid* da célula;
- `possible_numbers(self, cell)`: Retorna um *array* de *booleanas* com os números possíveis em uma determinada célula baseado na função `is_number_valid`;
- `number_of_possible_numbers(self, cell)`: Retorna a quantidade de possíveis números em uma determinada célula com base na função `is_number_valid`;
- `update_possibilities(self, number, cell)`: Após a inserção de um número no grid numa determinada célula, o `self.possibilities` irá se alterar na linha, coluna e quadrado da célula, para evitar a repetição do número, assim, essa função atualiza o `self.possibilities`;
- `line_possibility(self, square)`: Atualiza o `self.possibilities_line` com base no `self.possibilities` em um determinado quadrado (*subgrid*);
- `column_possibility(self, square)`: Atualiza o `self.possibilities_line` com base no `self.possibilities` em um determinado quadrado (*subgrid*);
- `update(self)`: Atualiza o estado do sudoku.

B. *Implementação da FSM*

A implementação da FSM consistiu em descrever estados que apontam o que o código deve realizar quando em uma determinada situação, cada estado tentará realizar algo sobre o *grid*, se obtiver sucesso, a FSM voltará a um estado inicial mais básico (`Fill_Numbers_State()`), caso contrário, irá para outro estado para tentar algo diferente, de modo que os próximos estados são cada vez mais improváveis de serem necessários, esse processo continua até que a máquina resolva o sudoku ou não saiba mais o que fazer. Assim, temos os seguintes estados:

1) `Fill_Possibilities_State()`: É o estado inicial, preenche o `self.possibilities` do grid inicial, quando terminado vai para o estado fundamental `Fill_Numbers_State()`;

2) `Fill_Numbers_State()`: É o estado fundamental, todos os estados após este retornarão para este em caso de sucesso. Preenche os números no *grid* caso apenas um número seja possível em uma determinada célula (analisando o `self.possibilities`, assim como todos os próximos estados analisarão). Ao final, segue para o próximo estado;

3) `Fill_Line_State()`: Preenche os números no *grid* em caso de, em uma determinada linha, tal número só possa estar em uma determinada célula. Segue para o próximo em caso de não fazer nada, assim como todos os próximos seguirão para seus consecutivos em caso de falha;

4) `Fill_Column_State()`: Análogo ao `Fill_Line_State()`, mas para colunas;

5) `Fill_Square_State()`: Análogo ao `Fill_Line_State()`, mas para quadrados (*subgrids*);

6) `Possibilities_Line_State()`: Caso em um determinado *subgrid*, determinado número seja possível apenas em uma determinada linha, então, nos subgrids da mesma horizontal não poderá haver esse número nessa mesma linha, assim, esse estado atualiza o `self.possibilities` com base nisso. Exemplo:

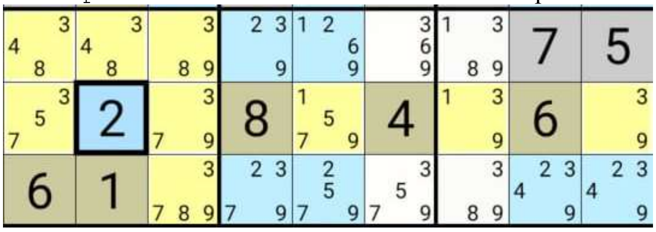

Fig. 2. Exemplo de situação para o atual estado.

Veja que no quadrado da direita, o número 2 só pode estar na Terceira linha, isto implica que haverá um 2 nessa linha e, portanto, não poderá haver um 2 na Terceira linha do quadrado do meio, o que permite restringir as possibilidades dessas células;

7) `Possibilities_Column_State()`: Análogo ao anterior, mas para colunas;

8) `Possibilities_Line2_State()`: Semelhante ao `Possibilities_Line_State`, mas agora, analisa se um número numa determinada linha, só pode estar em um quadrado. Caso positivo, o número não poderá estar em outra linhas deste mesmo quadrado. Exemplo:

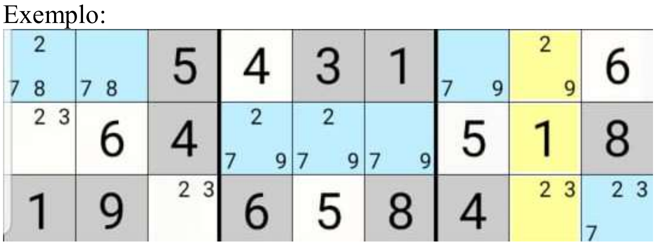

Fig. 3. Exemplo de situação para o atual estado.

Veja que o 7 da Terceira linha só pode estar no terceiro quadrado, logo, o 7 desses quadrado só pode estar nessa linha, o que nos permite eliminar a possibilidade do 7 nas outras linhas desse quadrado;

9) `Possibilities_Column2_State()`: Análogo ao anterior, mas para colunas;

10) `Possibilities_Pair_Line_State()`: Se, em uma mesma linha, há duas células em que é possível dois números, e apenas estes dois, então esse par de números está nesse par de células, logo, é possível eliminar a possibilidade desses estarem em outra célula desta linha. Exemplo:

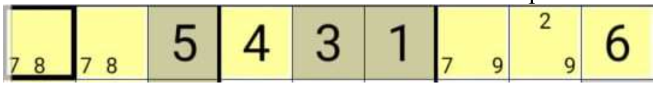

Fig. 4. Exemplo de situação para o atual estado.

Veja que na linha, o 7 e 8 são possíveis, e somente eles são possíveis, na primeira e na segunda célula, assim, 7 e 8 não podem estar em qualquer outra das células desta linha, o que elimina a possibilidade do 7 na sétima célula;

11) `Possibilities_Pair_Column_State()`: Análogo ao anterior, mas para colunas;

12) `Possibilities_Pair_Square_State()`: Análogo ao anterior, mas para quadrados (*subgrids*);

13) `Possibilities_Pair_Line2_State()`: Semelhante ao `Possibilities_Pair_Line_State`, mas agora, analisa se um par de pontos só pode estar presente em uma determinada linha em um par de células, então podemos eliminar a possibilidade de outros números estarem nesse par de células. Exemplo:

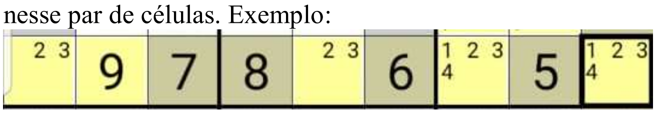

Fig. 5. Exemplo de situação para o atual estado.

Veja que os números 1 e 4 desta linha só podem estar nas sétima e nona células, assim, o 2 e o 3 não podem estar nessas células;

14) `Possibilities_Pair_Column2_State()`: Análogo ao anterior, mas para colunas;

15) `Possibilities_Pair_Square2_State()`: Análogo ao anterior, mas para quadrados (*subgrids*);

16) `End_State()`: Estado final, o Sudoku aqui já deve estar completo, caso negativo, a inteligência foi incapaz de resolvê-lo e só fez até certo ponto.

É válido ressaltar que os estados implementados relacionados à pares de números (10 a 15) são um caso particular de algo mais geral, o mesmo pode ocorrer com trios de números, quartetos, e assim por diante, mas isto não foi implementado pois é muito rara a necessidade de considerar grupos maiores que pares.

C. *Teste da máquina*

Após completa implementação do Código, testou-se o algoritmo com um gerador de sudokus on-line, que provém sudokus de variadas dificuldades. Testou-se o algoritmo nas cinco maiores dificuldades do puzzle disponíveis, dificuldades estas que disponibilizavam 17 dicas iniciais o mínimo para a resolução de um Sudoku, as dificuldades eram intituladas de, por ordem de dificuldade: 17 (super_hard), 17 (super_level2), 17(super_level3), 17(super_level4) e 17(extreme).

Os testes eram 12 Sudokus com cada dificuldade, ao final analisou-se quantos sudokus foram completos. É importante ressaltar que os Sudokus em dificuldades não tão elevadas quanto estas eram absolutamente sempre resolvidos, o que não é muito útil à análise do desempenho da máquina.

O site disponibiliza uma função que já fornece os 12 sudokus em determinada dificuldade, mas foi necessário adaptar a formatação por meio de uma função criada em `state_machine_test.py`.

III. Resultados e Discussão

A partir dos testes realizados com os 12 Sudokus em cada uma das dificuldades, obtiveram-se os seguintes resultados:

Tabela 1

Dificuldade	Sudokus Completos
17 (super_hard)	12/12
17 (super_level2)	6/12
17 (super_level3)	9/12
17 (super_level4)	9/12
17 (extreme)	6/12

Assim, é notória a eficiência do algoritmo implantado, pois ele foi capaz de resolver boa parte dos sudokus considerados mais difíceis que existem, é bem verdade que não consegui realizar todos, mas demonstra uma ótima eficiência.

Além disso, foi verificado à mão que a maioria dos sudokus não realizados pela máquina, o preenchimento do sudoku chegou a um ponto em que era absolutamente necessário o uso da tentativa e erro, pois existem esses tipos de sudoku. Isso só reforça que a lógica utilizada foi muito eficiente na resolução do problema.

O algoritmo, mesmo que não resolva por completo o sudoku, retorna uma resolução parcial deste (geralmente a maior parte das células já estão preenchidas). Assim, ainda seria possível implementar uma força bruta ao final desse algoritmo para conseguir finalizar qualquer sudoku em um tempo hábil.

IV. Conclusão

A partir dos resultados obtidos, conclui-se que foi obtida uma forma bem mais inteligente de se realizar Sudokus, pois não envolve "força bruta". Ademais, apesar da lógica não conseguir concluir absolutamente todos os Sudokus, ainda é possível implementar uma força bruta ao final da resolução parcial, o que diminuiria consideravelmente o tempo de resolução do sudoku. Em suma, a máquina de estados finita foi uma boa alternativa ao problema do Sudoku.

Referências

[1] Sudoku. Disponível em: https://pt.wikipedia.org/wiki/Sudoku. Acesso em 4 de junho de 2019.

[2] Generate and solve Sudoku. Disponível em: https://kjell.haxx.se/sudoku/. Acesso em 4 de junho de 2019.