Tiago Koji Castro Shibata (8988730)

PCS - 3838
Inteligência Artificial

Relatório:
Exercício prático

Introdução

Nesse exercício prático, foi implementado um programa que resolve problemas de *Sudoku*.

O *sudoku* é um jogo baseado na colocação lógica de números em um tabuleiro, satisfazendo uma série de restrições. Na sua versão mais comum, um tabuleiro com 9x9 células é utilizado (outros tamanhos são também possíveis). Algumas células já estão preenchidas, dando pistas iniciais. As casas vazias devem ser deduzidas pelo jogador utilizando lógica, observando as restrições do jogo. As seguinte restrições precisam ser consideradas:

- O tabuleiro precisa ser todo preenchido com números de 1 a 9
- Cada coluna deve conter uma (e apenas uma) ocorrência de cada número de 1 a 9
- Cada linha deve conter uma (e apenas uma) ocorrência de cada número de 1 a 9
- Ao dividir o tabuleiro em 9 regiões 3x3, cada região deve conter uma (e apenas uma) ocorrência de cada número de 1 a 9

O *sudoku* é importante pela sua popularidade, sendo um passatempo bastante comum e que pode ser encontrado em qualquer banca de jornais.

Implementei um programa em Prolog que, dado um tabuleiro incompleto, resolve o *Sudoku* utilizando a abordagem de satisfação de restrições. Há muita importância prática em um programa que resolva o *Sudoku* por essa abordagem, como:

- Verificar a validade de problemas: Tabuleiros de *Sudoku* costumam ser considerados válidos se e apenas se há uma única solução possível do tabuleiro completo. Um programa de resolução pode verificar se um dado problema pode ser resolvido, identificando configurações iniciais impossíveis ou que geram mais que uma solução possível
- Auxiliar na criação de problemas: Durante a criação de problemas, se um tabuleiro inválido for detectado, o criador pode observar quais casas não estão restringidas ainda e quais restrições precisam ser sanadas, para assim decidir como alterar o quebra cabeça para que se torne válido.

Abordagem proposta

Implementei um programa em Prolog que, dado um tabuleiro incompleto, resolve o *Sudoku* utilizando a abordagem de satisfação de restrições. A biblioteca ***clpfd*** (Constraint Logic Programming over Finite Domains) foi utilizada, que possui predicados prontos para facilitar a criação de programas usando satisfação de restrições.

O código foi baseado na implementação dada na documentação do SWI-Prolog [1].

Estrutura do software

O seguinte programa foi utilizado:

```
% Baseado em http://www.swi-prolog.org/pldoc/man?section=clpfd-sudoku
:- use_module(library(clpfd)).

/*
 * Verificar que:
 * * Há 9 linhas;
 * * Cada linha possui mesmo comprimento de 9;
 * * Cada elemento está entre 1 e 9;
 * * Em cada linha, todos os elementos são distintos;
 * * Em cada coluna, todos os elementos são distintos;
 * * Blocos 3x3 são válidos;
 */
sudoku(Rows) :-
        length(Rows, 9), maplist(same_length(Rows), Rows),
        append(Rows, Vs), Vs ins 1..9,
        maplist(all_distinct, Rows),
        transpose(Rows, Columns),
        maplist(all_distinct, Columns),
        Rows = [As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is],
        blocks(As, Bs, Cs),
        blocks(Ds, Es, Fs),
        blocks(Gs, Hs, Is).

blocks([], [], []).
blocks([N1,N2,N3|Ns1], [N4,N5,N6|Ns2], [N7,N8,N9|Ns3]) :-
        all_distinct([N1,N2,N3,N4,N5,N6,N7,N8,N9]),
        blocks(Ns1, Ns2, Ns3).
```

Em partes:

```
sudoku(Rows) :-
```

Essa regra define o predicado *sudoku*, que será verdadeiro se as próximas condições forem verdadeiras. Esperamos que *Rows* seja dado como um lista de listas (lista contendo 9 listas, onde cada lista representa uma linha, e cada linha deve conter 9 elementos, sendo números ou variáveis não instanciadas).

```
        length(Rows, 9), maplist(same_length(Rows), Rows),
```

Validamos que há 9 linhas e cada uma delas possui o mesmo comprimento que a lista *Rows* (ou seja, 9 elementos).

```
        append(Rows, Vs), Vs ins 1..9,
```

append gera uma lista (*Vs*) a partir da concatenação das listas em *Rows*, e verifica se todos os elementos estão no domínio *1..9*. O predicado *ins* está na biblioteca ***clpfd***.

```
        maplist(all_distinct, Rows),
```

Para cada elemento de *Rows*, chamamos *all_distinct* para verificar que nenhuma linha possui elementos repetidos.

```
        transpose(Rows, Columns),
        maplist(all_distinct, Columns),
```

Realizamos transposição de *Rows* em *Columns* e verificamos que todos os elementos são distintos.

```
        Rows = [As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is],
        blocks(As, Bs, Cs),
        blocks(Ds, Es, Fs),
        blocks(Gs, Hs, Is).
```

Instanciamos *As, …, Is* contendo as linhas de *Rows*, e verificamos de 3 em 3 linhas se seus blocos são válidos.

```
blocks([], [], []).
blocks([N1,N2,N3|Ns1], [N4,N5,N6|Ns2], [N7,N8,N9|Ns3]) :-
        all_distinct([N1,N2,N3,N4,N5,N6,N7,N8,N9]),
        blocks(Ns1, Ns2, Ns3).
```

Utilizamos recursão para verificar os blocos. Para cada trio de linhas recebido, extrai-se as 3 primeiras colunas da cabeça da lista e verifica-se se todos os elementos são distintos, continuando a recursão no restante da linha. A recursão acaba se não houver mais elementos na linha.

Descrição dos experimentos

Muitos testes poderiam ser realizados para validar diferentes *corner-cases* do problema. Eu escolhi 7 configurações de teste, tomadas de diferentes fontes *online* [1], [2], [3].

2 testes são de configurações válidas e 5 de inválidas:

```
% Casos de teste
% Válido
problem(1, [[_,_,_,_,_,_,_,_,_],
            [_,_,_,_,_,3,_,8,5],
            [_,_,1,_,2,_,_,_,_],
            [_,_,_,5,_,7,_,_,_],
            [_,_,4,_,_,_,1,_,_],
            [_,9,_,_,_,_,_,_,_],
            [5,_,_,_,_,_,_,7,3],
            [_,_,2,_,1,_,_,_,_],
            [_,_,_,_,4,_,_,_,9]]).

% Inválido (elemento fora de 1..9)
problem(2, [[_,_,_,_,_,_,_,_,_],
            [_,_,_,_,_,3,_,10,5],
            [_,_,1,_,2,_,_,_,_],
            [_,_,_,5,_,7,_,_,_],
            [_,_,4,_,_,_,1,_,_],
            [_,9,_,_,_,_,_,_,_],
            [5,_,_,_,_,_,_,7,3],
            [_,_,2,_,1,_,_,_,_],
            [_,_,_,_,4,_,_,_,9]]).

% Inválido (sem soluções)
problem(3, [[_,_,_,_,_,_,_,_,_],
            [_,_,_,_,_,3,2,8,5],
            [_,_,1,_,2,_,_,_,_],
            [_,_,_,5,_,7,_,_,_],
            [_,_,4,_,_,_,1,_,_],
            [_,9,_,_,_,_,_,_,_],
            [5,_,_,_,_,_,_,7,3],
            [_,_,2,_,1,_,_,_,_],
            [_,_,_,_,4,_,_,_,9]]).

% Inválido (múltiplas soluções)
problem(4, [[_,_,_,_,_,_,_,_,_],
            [_,_,_,_,_,_,_,_,_],
            [_,_,_,_,_,_,_,_,_],
            [_,_,_,_,_,_,_,_,_],
            [_,_,_,_,_,_,_,_,_],
            [_,_,_,_,_,_,_,_,_],
            [_,_,_,_,_,_,_,_,_],
            [_,_,_,_,_,_,_,_,_],
            [_,_,_,_,_,_,_,_,_]]).

% Inválido (múltiplas soluções)
problem(5, [[_,8,_,_,_,9,7,4,3],
            [_,5,_,_,_,8,_,1,_],
            [_,1,_,_,_,_,_,_,_],
            [8,_,_,_,_,5,_,_,_],
```

```
            [_,_,_,8,9,4,_,_,_],
            [_,_,_,3,_,_,_,_,6],
            [_,_,_,_,_,_,_,7,_],
            [_,3,_,5,_,_,_,8,_],
            [9,7,2,4,_,_,_,5,_]]).

% Inválido (2 soluções)
problem(6, [[_,3,9,_,_,_,1,2,_],
            [_,_,_,9,_,7,_,_,_],
            [8,_,_,4,_,1,_,_,6],
            [_,4,2,_,_,_,7,9,_],
            [_,_,_,_,_,_,_,_,_],
            [_,9,1,_,_,_,5,4,_],
            [5,_,_,1,_,9,_,_,3],
            [_,_,_,8,_,5,_,_,_],
            [_,1,4,_,_,_,8,7,_]]).

% Válido
problem(7, [[_,_,_,2,6,_,7,_,1],
            [6,8,_,_,7,_,_,9,_],
            [1,9,_,_,_,4,5,_,_],
            [8,2,_,1,_,_,_,4,_],
            [_,_,4,6,_,2,9,_,_],
            [_,5,_,_,_,3,_,2,8],
            [_,_,9,3,_,_,_,7,4],
            [_,4,_,_,5,_,_,3,6],
            [7,_,3,_,1,8,_,_,_]]).
```

Temos um teste com elemento fora do domínio, um sem solução, um com tabuleiro vazio, um com múltiplas soluções, e um com 2 soluções. _ representa uma variável não instanciada.

Os problemas foram executados em SWI-Prolog 7.6.4:

```
$ swipl --version
SWI-Prolog version 7.6.4 for x86_64-linux
```

Problemas de teste podem ser rodados via:

```
problem(1, Rows), sudoku(Rows), maplist(portray_clause, Rows).
```

Aqui, há o predicado *problem(1, Rows)*, que instanciará *Rows* com os valores previamente dados no predicado *problem(1, descrição do problema)*. Depois, *sudoku* é chamado e para cada linha é chamado *portray_clause*, que realiza *pretty-print* da linha. O resultado é:

```
[9, 8, 7, 6, 5, 4, 3, 2, 1].
[2, 4, 6, 1, 7, 3, 9, 8, 5].
[3, 5, 1, 9, 2, 8, 7, 4, 6].
[1, 2, 8, 5, 3, 7, 6, 9, 4].
[6, 3, 4, 8, 9, 2, 1, 5, 7].
```

```
[7, 9, 5, 4, 6, 1, 8, 3, 2].
[5, 1, 9, 2, 8, 6, 4, 7, 3].
[4, 7, 2, 3, 1, 9, 5, 6, 8].
[8, 6, 3, 7, 4, 5, 2, 1, 9].
Rows = [[9, 8, 7, 6, 5, 4, 3, 2|...], [2, 4, 6, 1, 7, 3, 9|...], [3,
5, 1, 9, 2, 8|...], [1, 2, 8, 5, 3|...], [6, 3, 4, 8|...], [7, 9,
5|...], [5, 1|...], [4|...], [...|...]].
```

Em problemas sem solução (como 2 ou 3), *false* é indicado. Em problemas com múltiplas soluções, os elementos que puderam ser instanciados são mostrados, e as restrições sobre os elementos restantes são indicadas. Por exemplo, no problema 6, que possui 2 soluções:

```
problem(6, Rows), sudoku(Rows), maplist(portray_clause, Rows).
```

Retorna:

```
[4, 3, 9, 6, 5, 8, 1, 2, 7].
[1, 5, 6, 9, 2, 7, 3, 8, 4].
[8, 2, 7, 4, 3, 1, 9, 5, 6].
[_, 4, 2, 5, 1, _, 7, 9, 8].
[7, 8, 5, 2, 9, 4, 6, 3, 1].
[_, 9, 1, 7, 8, _, 5, 4, 2].
[5, 7, 8, 1, 4, 9, 2, 6, 3].
[2, 6, 3, 8, 7, 5, 4, 1, 9].
[9, 1, 4, 3, 6, 2, 8, 7, 5].
Rows = [[4, 3, 9, 6, 5, 8, 1, 2|...], [1, 5, 6, 9, 2, 7, 3|...], [8,
2, 7, 4, 3, 1|...], [_31504, 4, 2, 5, 1|...], [7, 8, 5, 2|...],
[_31624, 9, 1|...], [5, 7|...], [2|...], [...|...]],
_31504 in 3\/6,
all_distinct([_31504, 4, 2, 7, 8, 5, _31624, 9|...]),
all_distinct([4, 1, 8, _31504, 7, _31624, 5, 2|...]),
all_distinct([_31504, 4, 2, 5, 1, _31534, 7, 9|...]),
_31624 in 3\/6,
all_distinct([_31624, 9, 1, 7, 8, _31654, 5, 4|...]),
_31654 in 3\/6,
all_distinct([5, 1, _31534, 2, 9, 4, 7, 8|...]),
all_distinct([8, 7, 1, _31534, 4, _31654, 9, 5|...]),
_31534 in 3\/6.
```

Vemos o tabuleiro e as 4 casas que não puderam ser instanciadas. Vemos também as restrições que caem sobre essas 4 casas. Em posse desse conhecimento, poderíamos mudar o *puzzle* de forma que uma dessas 4 casas ficasse bem definida.

Tempo de execução

As resoluções foram geradas de maneira surpreendentemente rápida, visualmente instantânea para o usuário. Em um teste simples, realizei medição de tempo. Primeiro compilei o programa para um executável, para que o tempo de compilação das regras não entrasse no teste, e rodei alguns dos testes:

```
$ swipl -o sudoku -c sudoku.pl
$ echo 'problem(7, Rows), sudoku(Rows), maplist(portray_clause, Rows).' |
time ./sudoku
```

Testes foram executados em um Intel(R) Core(TM) i5-8500. No carregamento e execução de todo o programa, o tempo médio adquirido foi de 0,03s no problema 7 (válido). O problema 1, que também é válido mas é mais difícil de resolver (menos restrições iniciais), toma 0,07s em média. O problema 2 (trivial pelo elemento fora do domínio) tomou 0,01s, e o 3 (sem soluções) 0,06s. Em todos os testes, a execução tomou menos que 0,1s.

Alternativamente, usando o comando *time* da biblioteca *statistics*, tive o seguinte resultado no problema 7:

```
160,523 inferences, 0.015 CPU in 0.015 seconds (100% CPU, 10803175
Lips)
```

Para forçar o tempo de execução, testei com o *puzzle* mais difícil que encontrei na internet [4]. Sua definição é:

```
problem(8, [[1,_,_,_,_,7,_,9,_],
            [_,3,_,_,2,_,_,_,8],
            [_,_,9,6,_,_,5,_,_],
            [_,_,5,3,_,_,9,_,_],
            [_,1,_,_,8,_,_,_,2],
            [6,_,_,_,_,4,_,_,_],
            [3,_,_,_,_,_,_,1,_],
            [_,4,_,_,_,_,_,_,7],
            [_,_,7,_,_,_,3,_,_]]).
```

Nele, não havia uma solução encontrada apenas observando as restrições e foi necessário utilizar busca e *backtracking* (forçando instanciação com o uso de *label*):

```
problem(8, Rows), time(sudoku(Rows)), time(maplist(label, Rows)),
maplist(portray_clause, Rows).
```

Os dois comandos *time* geraram:

```
% 523,374 inferences, 0.034 CPU in 0.034 seconds (100% CPU, 15399409
Lips)
% 2,131,561 inferences, 0.138 CPU in 0.138 seconds (100% CPU, 15490960
Lips)
```

O resultado ainda é bastante satisfatório.

Análise dos resultados

O programa foi implementado e testado com sucesso. A execução do algoritmo implementado por satisfação de restrições é surpreendentemente rápida, com tempos de execução na ordem de dezenas ou centenas de milésimos de segundo em todos os testes realizados. Testes passaram com sucesso, e casos sem solução tiveram as casas não restringidas indicadas.

Conclusões e trabalhos futuros

O trabalho foi implementado com sucesso e pude aprender um pouco sobre a linguagem Prolog e sobre o programação utilizando satisfação de restrições.

Como melhoria do trabalho, a implementação poderia ser generalizada para outros números de dimensões. A implementação provavelmente poderia ser otimizada mudando um pouco a ordem de execução das regras, de maneira a reduzir a árvore de busca; no entanto, esses resultados de tempo já são bastante promissores e aplicáveis em problemas reais de *Sudoku*, inclusive de dificuldade bastante elevada.

Referências externas

[1]: http://www.swi-prolog.org/pldoc/man?section=clpfd-sudoku
[2]: http://www.sudokudragon.com/unsolvable.htm
[3]: http://sudopedia.enjoysudoku.com/Invalid_Test_Cases.html
[4]: https://www.kristanix.com/sudokuepic/worlds-hardest-sudoku.php