PCS3838 Inteligência Artificial – Exercício Prático

Eric Rodrigues Pires

Nesse relatório, será realizada a implementação em Prolog de um resolvedor de Sudokun por satisfação de restrições.

***Palavras-chave*—Prolog, Sudoku, satisfação de restrições, programação lógica.**

I. Introdução

Esse exercício prático tem como objetivo solucionar um tabuleiro qualquer de Sudoku utilizando-se a linguagem Prolog – mais especificamente, utilizando-se programação lógica de satisfação de restrições (CLP).

O Sudoku é um problema clássico muito conhecido e com regras simples. Por isso, ele é bastante popular. Ele possui um tabuleiro 9x9, aonde cada linha, coluna, e região 3x3 deve possuir exatamente um de cada número entre 1 e 9.

Devido à simplicidade do problema, ele pode ser utilizado para aplicar os conceitos de CLP no aprendizado da linguagem Prolog.

II. Abordagem Proposta

Como mencionado anteriormente, será utilizada a satisfação de restrições neste exercício. Assim, será apresentada a modelagem utilizada para as variáveis, o domínio, e as restrições.

Suponhamos um tabuleiro de Sudoku com valores A_{ij}, onde i representa uma linha de 1 a 9, e j representa uma coluna de 1 a 9. Baseado no enunciado do Sudoku, podemos elaborar as seguintes restrições para o problema:

- **Variáveis:** $A_{ij}, \forall i, j \in [1, 9]$
- **Domínio:** $A_{ij} \in [1, 9], \forall i, j \in [1, 9]$
- **Restrições:**
  $A_{ij} \neq A_{ik}, \forall i, j, k \in [1, 9], j \neq k$;
  $A_{ij} \neq A_{kj}, \forall i, j, k \in [1, 9], i \neq k$;
  $A_{ij} \neq A_{kl}, \forall i, j, k, l \in [1, 3], \langle i, j\rangle \neq \langle k, l\rangle$;
  $A_{ij} \neq A_{kl}, \forall ik \in [1, 3], \forall j, l \in [4, 6], \langle i, j\rangle \neq \langle k, l\rangle$;
  $A_{ij} \neq A_{kl}, \forall i, k \in [1, 3], \forall j, l \in [7, 9], \langle i, j\rangle \neq \langle k, l\rangle$;
  $A_{ij} \neq A_{kl}, \forall i, k \in [4, 6], \forall j, l \in [1, 3], \langle i, j\rangle \neq \langle k, l\rangle$;
  $A_{ij} \neq A_{kl}, \forall i, j, k, l \in [4, 6], \langle i, j\rangle \neq \langle k, l\rangle$;
  $A_{ij} \neq A_{kl}, \forall i, k \in [4, 6], \forall j, l \in [4, 6], \langle i, j\rangle \neq \langle k, l\rangle$;
  $A_{ij} \neq A_{kl}, \forall i, k \in [7, 9], \forall j, l \in [1, 3], \langle i, j\rangle \neq \langle k, l\rangle$;
  $A_{ij} \neq A_{kl}, \forall i, k \in [7, 9], \forall j, l \in [4, 6], \langle i, j\rangle \neq \langle k, l\rangle$;
  $A_{ij} \neq A_{kl}, \forall i, j, k, l \in [7, 9], \langle i, j\rangle \neq \langle k, l\rangle$

A formulação matemática para as restrições das linhas e colunas (as duas primeiras restrições) é trivial. Para as regiões 3x3 (demais restrições), é necessário enumerar cada uma delas. Com esses parâmetros, é possível realizar a satisfação de restrições do problema.

III. Estrutura do Software

O programa entregue foi completamente desenvolvido do zero, sem reutilizar código de implementações pré-existentes. Ele possui três módulos importantes:

1) Módulo de resolução de Sudoku `solver.pl`. Utilizou-se a biblioteca `clpfd` para a resolução de problemas de satisfação de restrições, como indicado na documentação [1]. As restrições foram separadas em três categorias, cada uma com sua função e implementação específicas:
   - Restrição de linhas: Simplesmente adicionar restrição de valores únicos (`all_distinct/1`) a cada 9 elementos da lista.
   - Restrição de colunas: A cada 9 elementos da lista, coletar cada um em uma nova lista distinta, representando as colunas respectivamente, e ao final, aplicar a restrição de valores únicos nestas listas de colunas.
   - Restrição de regiões: Coletar de 3 em 3 elementos da lista em três listas distintas (representando cada região), até terminar de ler 3 linhas consecutivas. Neste momento, aplicar a restrição de valores únicos nas listas de regiões, criar novas listas, e continuar iterando até varrer todos os elementos.
2) Módulo de leitura de arquivo `file.pl`. Dado um nome de arquivo, este módulo carrega os valores em uma variável representando um tabuleiro de Sudoku. Os arquivos devem ser tabuleiros no formato de uma lista do Prolog, para facilitar a leitura do programa, e as casas vazias devem ser variáveis anônimas _.
3) Módulo de linha de comando `main.pl`. Obtém uma lista de arquivos dos argumentos da linha de comando que invocou o programa e resolve, imprimindo na tela as respostas. Cada argumento será tratado recursivamente. Foi implementada também a lógica de impressão dos tabuleiros para exibição na tela de forma legível. A utilização de argumentos da linha de comando se baseou em uma implementação de servidor em Prolog [2]. Além disso, esse módulo realiza o `trace` da solução dos tabuleiros quando o Prolog roda em configuração de debug (por padrão).

Esses módulos foram integrados para resolver tabuleiros em arquivos e imprimir os resultados na tela. Os detalhes de implementação foram comentados nas fontes dos programas.

IV. Descrição dos Experimentos

Foram escolhidos dois tabuleiros de Sudoku para testes, criados por um aplicativo de celular [3]. O primeiro foi gerado na configuração “Easy”, e o segundo, na configuração “Nightmare”. Eles foram salvos respectivamente

como `sudoku1.txt` e `sudoku2.txt`, no formato de listas do Prolog esperado.

Feito isso, o programa foi executado no SWI-Prolog da linha de comando de um MacBook. Utilizando-se o utilitário do terminal `time`, obteve-se um tempo de execução de aproximadamente 330 ms para a resolução dos dois tabuleiros.

V. Análise dos Resultados

Abaixo, reproduziu-se os resultados obtidos na execução do programa:

```
$ swipl --nodebug main.pl sudoku1.txt sudoku2.txt

+---+---+---+
|843|621|795|
|691|735|248|
|572|498|613|
+---+---+---+
|365|247|981|
|219|586|437|
|487|319|562|
+---+---+---+
|758|164|329|
|124|953|876|
|936|872|154|
+---+---+---+

+---+---+---+
|716|493|528|
|452|178|693|
|389|265|417|
+---+---+---+
|891|527|346|
|523|946|871|
|674|831|952|
+---+---+---+
|937|614|285|
|268|759|134|
|145|382|769|
+---+---+---+
```

Os resultados obtidos foram comparados com os de uma ferramenta online de resolução de Sudoku [4], retornando os mesmos valores que os encontrados acima. Dessa forma, constatou-se a implementação correta do programa.

Também foi testada a execução do `trace` de solução [5]. Abaixo está um exemplo de uso da ferramenta para depuração:

```
$ swipl main.pl sudoku1.txt

Begin trace
   Call: (9) solver:solve_sudoku([8, _396, _402, _408,
     2, _420, 7|...]) ? creep
   Call: (10) clpfd:ins([8, _396, _402, _408, 2, _420,
     7|...], ..(1, 9)) ? creep
   Call: (11) clpfd:fd_must_be_list([8, _396, _402,
     _408, 2, _420, 7|...]) ? skip
   Exit: (11) clpfd:fd_must_be_list([8, _396, _402,
     _408, 2, _420, 7|...]) ? creep
   Call: (11) clpfd:'__aux_maplist/2_fd_variable+0'([
     8, _396, _402, _408, 2, _420, 7|...]) ? up
   Exit: (10) clpfd:ins([8, _974{clpfd = ...}, _1086{
     clpfd = ...}, _1184{clpfd = ...}, 2, _1282{clpfd
     = ...}, 7|...], ..(1, 9)) ? creep
   Call: (10) solver:row_constraint([8, _974{clpfd =
     ...}, _1086{clpfd = ...}, _1184{clpfd = ...}, 2,
     _1282{clpfd = ...}, 7|...]) ? creep
   Call: (11) clpfd:all_distinct([8, _974{clpfd = ...}
     , _1086{clpfd = ...}, _1184{clpfd = ...}, 2,
     _1282{clpfd = ...}, 7|...]) ? creep
   Call: (12) clpfd:fd_must_be_list([8, _974{clpfd =
     ...}, _1086{clpfd = ...}, _1184{clpfd = ...}, 2,
     _1282{clpfd = ...}, 7|...]) ? up
   Exit: (11) clpfd:all_distinct([8, _974{clpfd = ...
     }, _1086{clpfd = ...}, _1184{clpfd = ...}, 2,
     _1282{clpfd = ...}, 7|...]) ? up
   Exit: (10) solver:row_constraint([8, _568{clpfd =
     ...}, _636{clpfd = ...}, _698{clpfd = ...}, 2,
     _760{clpfd = ...}, 7|...]) ? creep
   Call: (10) solver:column_constraint([8, _568{clpfd
     = ...}, _636{clpfd = ...}, _698{clpfd = ...}, 2,
     _760{clpfd = ...}, 7|...]) ? skip
   Exit: (10) solver:column_constraint([8, 4, 3, 6, 2,
     1, 7|...]) ? creep
   Call: (10) solver:block_constraint([8, 4, 3, 6, 2,
     1, 7|...]) ? skip
   Exit: (10) solver:block_constraint([8, 4, 3, 6, 2,
     1, 7|...]) ? creep
   Exit: (9) solver:solve_sudoku([8, 4, 3, 6, 2,
     1, 7|...]) ? no debug
+---+---+---+
|843|621|795|
|691|735|248|
|572|498|613|
+---+---+---+
|365|247|981|
|219|586|437|
|487|319|562|
+---+---+---+
|758|164|329|
|124|953|876|
|936|872|154|
+---+---+---+
```

É possível verificar que as funções da biblioteca `clpfd` realizam operações a cada nova restrição adicionada para solucionar o problema.

VI. Conclusões e Trabalhos Futuros

O projeto obteve os resultados esperados pelo enunciado do exercício prático. Além de tudo, foi possível compreender como utilizar a satisfação de restrições na linguagem Prolog, e como utilizar o `trace` para examinar a execução de um programa em linguagem lógica.

No futuro, este projeto pode ser expandido para tratar um arquivo em formato não-Prolog, apresentar um passo-a-passo humanamente factível para resolução de Sudoku, ou até mesmo gerar novos tabuleiros válidos para jogar.

Referências

[1] SWI-Prolog. *Constraint Logic Programming*. Disponível em: http://www.swi-prolog.org/pldoc/man?section=clp. Acesso em 27 de outubro de 2018.

[2] GitHub. *SWI-Prolog / swish*. Disponível em: https://github.com/SWI-Prolog/swish. Acesso em 28 de outubro de 2018.

[3] STRUHAR, V. *Best Sudoku Free*. Disponível em: https://play.google.com/store/apps/details?id=cz.struharv.sudoku. Acesso em 5 de novembro de 2018.

[4] Sudoku Solutions. *Online Sudoku Solver and Helper*. Disponível em: http://www.sudoku-solutions.com. Acesso em 13 de novembro de 2018.

[5] SWI-Prolog. *Debugging and Tracing Programs*. Disponível em: http://www.swi-prolog.org/pldoc/man?section=debugger. Acesso em 20 de novembro de 2018.