به نام خالق زیبایی‌ها

نرم افزار حل بازی سودوکو در پرولوگ

تحقیق درس آزمایشگاه هوش مصنوعی

فهرست

مقدمه

پرولوگ یکی از بهترین نمونه و مثال یک زبان برنامه نویسی منطقی است. یک برنامه منطقی دارای یک سری ویژگیهای قانون و منطق است . PROLOG از محاسبهٔ اولیه استفاده می کند. در حقیقت خود این نام از برنامه نویسی PRO در LOGIC می آید یک مفسر برنامه را بر اساس یک منطق می نویسد. پرولوگ یکی از زبان‌های برنامه‌نویسی کامپیوتر است که می‌توان گفت با تمام زبان‌های دیگر متفاوت است زیرا که برنامه‌های آن دستوری نیست، بلکه شامل یک رشته گزاره‌ها و قواعد است. در واقع، گزاره‌ها همان مفهوم گزاره‌های منطقی را دارند و قواعد هم در حکم گزاره‌نماها هستند، یعنی گزاره‌هایی که مشتمل بر متغیرها هستند.

برنامه نویس در پایان برنامه، یک هدف قرار می‌دهد که برنامه سعی می‌کند از طریق درست‌آزمایی گزاره‌هایی که دارد و از طریق انطباق دادن مقادیر معلوم با متغیرها، آن را تامین نماید.

در صورتی که در جایی از برنامه امکان انطباق و تأمین هدف فراهم نشد، برنامه برگشت می‌کند و از نزدیک‌ترین انشعاب ممکن، مسیر دیگری را دنبال می‌کند.

این ویژگی های پرولوگ این زبان را برای حل بازی سودوکو که یک عمل منطقی محسوب میشود، مناسب کرده است. لازم به ذکر است که در اینجا ما از SWI-Prolog استفاده میکنیم

لازم به ذکر است که به دلیل خوانایی بیشتر کد و عدم بهم ریختگی متن، در سورس کد از کامنت های انگلیسی استفاده شده است.

همچنین فایل sudoku.pl نیز به همراه این سند، داخل پوشه ارسالی موجود می‌باشد.

سودوکو چیست؟

سودوکو (به ژاپنی: 数独) جدول اعدادی است که یکی از بهترین تمرین‌های مغز و تقویت آی کیو است. دانشمندان می‌گویند :با حل سودوکو، مغز انسان ده سال جوان می‌شود. امروزه یکی از سرگرمی‌های رایج در کشورهای مختلف جهان به‌شمار می‌آید.

قانون بازی

نوع متداول سودوکو یک جدول ۹x۹ است که کل جدول هم به ۹ جدول کوچک‌تر ۳x۳ تقسیم شده‌است. در این جدول چند عدد به طور پیش فرض قرار داده شده که باید باقی اعداد را با رعایت سه قانون زیر یافت:

- قانون اول: در هر سطر جدول اعداد ۱ الی ۹ بدون تکرار قرار گیرد.
- قانون دوم: در هر ستون جدول اعداد ۱ الی ۹ بدون تکرار قرار گیرد.
- قانون سوم: در هر ناحیه ۳x۳ جدول اعداد ۱ الی ۹ بدون تکرار قرار گیرد.

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

یک صفحه‌ی سودوکوی حل نشده

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

بعد از حل شدن

حل مسئله

برای حل یک پازل سودوکو توسط ، می‌توانیم از تکنیکی به نام برنامه‌نویسی منطقی محدودیت (CLP) استفاده کنیم. CLP یک رویکرد قدرتمند برای حل مسائل است که شامل محدودیت های پیچیده بین متغیرها است. در مورد یک پازل سودوکو، می‌توانیم پازل را به‌عنوان فهرستی از فهرست‌ها نشان دهیم که هر فهرست داخلی یک ردیف از پازل را نشان می‌دهد.

این برنامه از کتابخانه clpfd استفاده می کند، که اگرچه هسته prolog نیست،اما یک ابزار بسیار قدرتمند است که اغلب در برنامه های prolog استفاده می شود. نام clpfd را می توان به "برنامه نویسی منطقی محدودیت در دامنه های محدود" یا "Constraint Logic Programming over Finite Domains" گسترش داد. بیایید بررسی کنیم که این به چه معناست. یک محدودیت، رابطه‌ای منطقی و ساده بین متغیرهای متعدد (و ناشناخته) است، که در آن هر یک می تواند مقداری را از یک دامنه (یک مجموعه) بگیرد. به عنوان مثال: "یک دایره در داخل یک مربع است" به ما یک محدودیت می دهد بدون اینکه موقعیت دقیقی برای مربع یا دایره ارائه دهد. می‌توانیم یک شی سوم، مثلاً یک مثلث، اضافه کنیم و یک محدودیت دیگر ایجاد کنیم - "مربع در سمت راست یک مثلث است". در اینجا دامنه‌ی محدود به معنای مجموعه‌ی محدود است.

در SWI-Prologبرنامه نویسی محدودیت با استفاده از مجموعه ای از کتابخانه ها تحقق می یابد:
clp/clp_distinct، clp/simplex، clp/clpfd، clpqr.

برای حل پازل با استفاده از CLP، ابتدا باید پازل را در یک لیست واحد از متغیرها پهن کنیم. ما می‌توانیم این کار را با استفاده از گزاره flatten/2انجام دهیم، که فهرستی از لیست‌ها را می‌گیرد و یک لیست واحد حاوی تمام عناصر فهرست‌های داخلی را برمی‌گرداند. برای مثال، اگر ما یک پازل را به صورت [[1,2,3],[4,5,6],[7,8,9]]نشان دهیم، آنگاه فراخوانی به flatten/2 نتیجه‌ی مقابل را برمیگرداند: .[1,2,3,4,5,6,7,8,9]

```
% Flatten the puzzle into a single list of variables.
flatten(Puzzle, Tmp), Tmp ins 1..9,
```

هنگامی که پازل صاف شد، می‌توانیم از قید `ins/2` **برای محدود کردن مقادیر متغیرها به اعداد صحیح از 1 تا 9 استفاده کنیم. این تضمین می‌کند که هر متغیر در پازل فقط می‌تواند مقداری از مجموعه مقادیر ممکن را بگیرد. یک پازل سودوکو به عنوان مثال، فراخوانی** `Var ins 1..9` **متغیر** `Var` **را محدود می کند تا فقط مقادیری از مجموعه**`{1, 2, 3, 4, 5, 6, 7, 8, 9}` **بگیرد.**

```
% Restrict the values of the variables to the integers from 1
to 9.
flatten(Puzzle, Tmp), Tmp ins 1..9,
```

در مرحله بعد، از گزاره `transpose/2` **برای جابجایی ردیف های پازل به ستون ها استفاده می کنیم. این به ما اجازه می دهد تا محدودیت** `all_distinct/1` **را برای هر سطر، ستون و بلوک از پازل اعمال کنیم. گزاره** `transpose/2` **فهرستی از لیست ها را می گیرد و فهرست جدیدی از لیست ها را که در آن ردیف ها و ستون های لیست اصلی جابجا شده اند، برمی گرداند. به عنوان مثال، اگر ما یک پازل را به صورت** `[[1,2,3],[4,5,6],[7,8,9]]` **نشان دهیم، آنگاه فراخوانی برای** `transpose/2` **نتیجه مقابل را برمیگرداند:** `[[1,4,7],[2,5,8],[3,6,9]]`.

```
% Transpose the rows of the puzzle into columns.
transpose(Rows, Columns),
```

محدودیت all_distinct/1 برای اطمینان از متمایز بودن هر عنصر در یک سطر، ستون یا بلوک استفاده می شود و از کتابخانه‌ی clpfd می آید. این لازمه یک پازل سودوکو معتبر است، زیرا هر سطر، ستون و بلوک باید شامل اعداد 1 تا 9 بدون تکرار باشد. برای اعمال قید all_distinct/1 برای هر سطر، ستون و بلوک پازل، می‌توانیم از گزاره maplist/2 استفاده کنیم که یک قید معین را برای هر عنصر اعمال می‌کند. به عنوان مثال، فراخوانی maplist(all_distinct,Rows) محدودیت all_distinct/1را برای هر لیست داخلی در لیست لیست ردیف ها اعمال می کند.

```
% Ensure that each row, column, and block contains distinct elements.
                                        maplist(all_distinct, Rows),
                                     maplist(all_distinct, Columns),
                                      maplist(all_distinct, Blocks),
```

برای تقسیم پازل به بلوک‌های 3×3، می‌توانیم از قید blocks/2استفاده کنیم. این گزاره فهرستی از لیست‌هایی را می‌گیرد که ردیف‌های پازل را نشان می‌دهند و فهرستی از فهرست‌ها را برمی‌گرداند که در آن هر فهرست داخلی یک بلوک 3×3 از پازل را نشان می‌دهد. به عنوان مثال، اگر ما یک پازل را به صورت [[1,2,3,4],[5,6,7,8],[9,1,2,3],[4,5,6,7]]نشان دهیم.

سپس فراخوانی تابع blocks/2 نتیجه‌ی زیر را برمیگرداند:

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

```
                                        % Split the puzzle into 3x3 blocks.
                                                    blocks(Rows, Blocks),
```

در نهایت، از گزاره `label/1` برای اختصاص یک مقدار به هر متغیر در پازل استفاده می کنیم. این گزاره پازل را به طور موثر با یافتن برچسب متغیرهایی که تمام محدودیت هایی را که ما تعریف کرده ایم، حل می کند. گزاره `label/1` یک گزاره جستجو است که مقادیری را به متغیرها نسبت می دهد به گونه ای که همه محدودیت ها رعایت شوند و از یک الگوریتم جستجوی عمق اول برای کشف فضای راه‌حل‌های ممکن استفاده می‌کند و اولین راه‌حلی را که تمام محدودیت‌ها را رعایت می‌کند، برمی‌گرداند.

```
% Solve the puzzle by assigning values to the variables.
maplist(label, Rows).
```

به طور کلی، این رویکرد برای حل یک پازل سودوکو با استفاده از CLP موثر است زیرا به ما اجازه می‌دهد به راحتی محدودیت‌های بین متغیرها را در پازل مشخص کنیم. با استفاده از گزاره‌های `ins/2`، `maplist`، `transpose/2` و `label/1`، می‌توانیم به طور خلاصه محدودیت‌هایی را که باید رعایت شوند تا معما راه‌حلی داشته باشد، بیان کنیم.

علاوه بر این، CLP یک رویکرد اعلامی برای حل مسئله است، به این معنی که ما فقط باید محدودیت ها را مشخص کنیم و به حل کننده اجازه دهیم راه حلی را پیدا کند که آن محدودیت ها را رعایت کند. این کار خواندن و درک کد را آسان می کند و به ما امکان می دهد به جای نگرانی در مورد جزئیات الگوریتم جستجو، روی تعریف محدودیت ها تمرکز کنیم.

در پایان، استفاده از CLP برای حل یک پازل سودوکو یک راه موثر و مختصر برای حل این نوع مشکلات است. با نمایش پازل به عنوان لیستی از لیست ها و اعمال محدودیت های مناسب، می توانیم به حل کننده اجازه دهیم راه حلی پیدا کند که تمام الزامات یک پازل معتبر سودوکو را رعایت کند.

تست ها

نسخه‌های پرولوگ:

SWI-Prolog 9.0.2-1 for MacOSX 10.14 (Mojave) and later on x86_64 and arm64

SWI-Prolog 9.0.2-1 for Microsoft Windows (64 bit)

برای اجرای کد، داخل دایرکتوری برنامه از دستور زیر در ترمینال(zsh و Powershell) استفاده شد:

```
$ swipl -f sudoku.pl
```

البته میتوان از طریق برنامه‌ی SWI-Prolog و با استفاده از دستور زیر هم به نتیجه مشابه رسید:

```
?- consult(sudoku).
```

سپس، طبق این الگو، پازل را وارد برنامه میکنیم:

```
$ Puzzle = [
[2,_,_,_,9,_,_,_,1],
[3,_,9,_,_,_,7,_,_],
[_,_,1,_,4,_,_,7,_],
[_,6,_,_,_,_,_,_,_],
[_,_,_,_,_,_,3,_,_],
[_,_,8,6,_,_,7,9,_],
[6,_,_,7,_,_,8,_,_],
[1,2,3,_,_,8,_,_,_],
[_,8,7,_,_,4,3,_,_]], Puzzle = [A,B,C,D,E,F,G,H,I],
sudoku(Puzzle).
```

که برنامه پاسخ زیر را برمیگرداند:

```
                                        Puzzle = [...],
                                A = [4, 9, 7, 1, 5, 3, 2, 8, 6],
                                B = [5, 1, 8, 2, 6, 7, 3, 9, 4],
                                C = [3, 2, 6, 9, 8, 4, 7, 5, 1],
                                D = [6, 7, 2, 8, 3, 9, 4, 1, 5],
                                E = [8, 5, 4, 7, 1, 2, 6, 3, 9],
                                F = [1, 3, 9, 5, 4, 6, 8, 7, 2],
                                G = [9, 4, 1, 3, 2, 8, 5, 6, 7],
                                H = [2, 8, 5, 6, 7, 1, 9, 4, 3],
                                I = [7, 6, 3, 4, 9, 5, 1, 2, 8].
```

یک نمونه‌ی دیگر:

```
                                          ?- Puzzle = [
                                        [_,_,6,_,_,_,9,_,_],
                                        [_,_,_,6,_,_,_,7,5],
                                        [_,5,8,_,_,7,_,1,4],
                                        [_,6,_,_,1,_,_,5,7],
                                        [_,_,_,7,5,_,_,6,_],
                                        [_,_,_,_,_,_,_,3,_],
                                        [_,_,_,1,_,_,7,8,3],
                                        [_,1,_,_,_,3,_,4,_],
[_,_,_,_,6,_,5,_,_]], Puzzle = [A,B,C,D,E,F,G,H,I],
                                        sudoku(Puzzle).
                                        Puzzle = [...],
                                A = [7, 4, 6, 5, 8, 1, 9, 3, 2],
                                B = [1, 3, 2, 6, 4, 9, 8, 7, 5],
                                C = [9, 5, 8, 2, 3, 7, 6, 1, 4],
                                D = [2, 6, 9, 3, 1, 8, 4, 5, 7],
                                E = [4, 8, 3, 7, 5, 2, 1, 6, 9],
                                F = [5, 7, 1, 4, 9, 6, 3, 2, 8],
                                G = [6, 9, 4, 1, 2, 5, 7, 8, 3],
                                H = [8, 1, 5, 9, 7, 3, 2, 4, 6],
                                I = [3, 2, 7, 8, 6, 4, 5, 9, 1].
```

سورس کد نهایی

```
                                                    % sudoku.pl
                                                    %
  % This program defines a `sudoku/1` predicate that solves a sudoku
                                                    puzzle. The
    % puzzle is represented as a list of lists, with each inner list
                                                    representing a
                                                % row of the puzzle.
                                                    %
                                                    %
                          % Author: Your Name (PouyaHallaj@Gmail.com)
                                                    %
                        % License: This code is in the public domain.
                                        :- use_module(library(clpfd)).
                                                % sudoku(Puzzle)
                                                    %
% Solves the sudoku puzzle represented by Puzzle. Puzzle should be a
                                                    list of
 % lists, with each inner list representing a row of the puzzle. The
                                                    solution
                    % will be a labeling of the variables in Puzzle.
                                                    %

                                                sudoku(Puzzle) :-
                % Flatten the puzzle into a single list of variables
                                                flatten(Puzzle, Tmp),
    % Restrict the values of the variables to the integers from 1 to
                                                    9
                                                Tmp ins 1..9,
                    % Transpose the rows of the puzzle into columns
                                                Rows = Puzzle,
                                            transpose(Rows, Columns),
                                % Split the puzzle into 3x3 blocks
                                                blocks(Rows, Blocks),
      % Apply the all_distinct/1 constraint to each row, column, and
                                                    block
                                            maplist(all_distinct, Rows),
```

```
                                    maplist(all_distinct, Columns),
                                    maplist(all_distinct, Blocks),
        % Find a labeling of the variables that satisfies all of the
                                                          constraints
                                                maplist(label, Rows).

                                               % blocks(Rows, Blocks)
                                                                    %
% Splits the rows of a sudoku puzzle into 3x3 blocks. Rows should be
                                                              a list of
     % lists, with each inner list representing a row of the puzzle.
                                                        Blocks will be
 % a list of lists, with each inner list representing a 3x3 block of
                                                             the puzzle.
                                                                    %
                                blocks([A,B,C,D,E,F,G,H,I], Blocks) :-
                                        % Split the rows into 3 blocks
   blocks(A,B,C,Block1), blocks(D,E,F,Block2), blocks(G,H,I,Block3),
                                % Combine the blocks into a single list
                               append([Block1, Block2, Block3], Blocks).

                                     % blocks(Row1, Row2, Row3, Block)
                                                 blocks([], [], [], []).
      blocks([A,B,C|Bs1],[D,E,F|Bs2],[G,H,I|Bs3], [Block|Blocks]) :-
                                % Combine the rows into a single block
                                          Block = [A,B,C,D,E,F,G,H,I],
                                % Split the remaining rows into blocks
                                        blocks(Bs1, Bs2, Bs3, Blocks).
```