

Shapely Squares

Rui Guedes [up201603854] and João Barbosa [up201604156]

Faculdade de Engenharia da Universidade do Porto
www.fe.up.pt

FEUP-PLOG, Turma 3MIEIC3, Grupo Shapely_Squares_2

Abstract. No âmbito da disciplina de Programação em Lógica, pretendeu-se desenvolver um programa, em Prolog, com restrições sobre domínios finitos para a resolução de um dos problemas de otimização ou decisão combinatória fornecidos. Shapely Squares foi o problema de decisão combinatória escolhido. Este problema consiste num puzzle 2D cuja representação é feita através de uma matriz bidimensional de tamanho variável. Cada célula desta matriz deve ser preenchida por um dígito e é identificada por um determinado elemento que indica as restrições a que cada célula está sujeita. Através da linguagem Prolog e fazendo uso da biblioteca *clpfd* do SICStus, desenvolveu-se uma aplicação capaz de resolver este problema, apresentando uma solução.

Keywords: SICStus · Prolog · Restrições · *Shapely Squares*.

1 Introdução

O desenvolvimento deste projeto tem como objetivo introduzir à programação em lógica com restrições, permitindo assim implementar soluções mais eficientes comparativamente com, por exemplo, uma abordagem do estilo *generate and test*. Para tal, foi necessário a resolução de um dos problemas de otimização ou decisão combinatória fornecidos, cuja decisão recaiu sobre o puzzle 2D, Shapely Squares.

O problema escolhido corresponde a um puzzle cuja representação é feita através de uma matriz de tamanho variável. Cada célula desta matriz é identificada por um determinado elemento que indica as restrições para a respetiva célula. Para a resolução do puzzle é necessário que a totalidade das restrições presentes no mesmo sejam verificadas, tratando-se, assim, de um problema de satisfação de restrições.

De modo a detalhar o problema em si bem como a sua resolução o presente artigo encontra-se estruturado da seguinte forma: apresentação detalhada do problema acompanhada com a abordagem ao mesmo, visualização da solução e resultados obtidos, conclusões e trabalho futuro, e por fim referências bibliográficas seguidas dos anexos onde se encontra o código fonte do programa desenvolvido.

2 Descrição do Problema

Shapely Squares é um problema de decisão combinatória. Este problema consiste num puzzle 2D cuja solução está dependente de um certo conjunto de restrições. Este problema é representado através de uma matriz de tamanho variável onde em cada célula encontra-se um determinado elemento que é responsável por definir as suas restrições. De seguida encontra-se uma possível representação do puzzle.











































							23
							20
							26
							20
							26
							19
							22

Fig. 1. Representação possível de um puzzle Shapely Squares

Cada célula do puzzle deve ser preenchida com um valor entre 0 e 9. A soma dos dígitos de cada linha deve ser igual ao valor da última coluna dessa mesma linha. A cada célula, como referido, estão também associadas restrições dependendo do elemento lá presente. Seguem-se as restrições para os diferentes elementos:

- **Estrela** - Deve ser primo, no mínimo 2 e não pode ter vizinhos que sejam primos ou iguais a 1.
- **Quadrado** - Deve ser 0 ou 5 e não pode ter o mesmo dígito que os seus vizinhos a menos que esse vizinho seja um diamante.
- **Diamante** - É ímpar e é igual à soma de todos os elementos à sua esquerda, na linha correspondente.
- **Triângulo** - Não pode ser 0, tem de estar localizado imediatamente abaixo de um dígito par e deve ser menor do que ele.
- **Círculo** - Não pode ser múltiplo de 3 e todos os círculos devem ter o mesmo dígito.
- **Cavalo de Xadrez** - Corresponde ao número de dígitos pares no seu alcance de ataque.
- **Coração** - A soma do seu valor com corações vizinhos deve ser igual a 10.

Uma solução a este problema, corresponde a uma matriz completamente preenchida com dígitos em que todas as restrições definidas são satisfeitas.

3 Abordagem

O problema foi abordado de forma iterativa, de forma a conseguir fornecer a melhor solução possível. Inicialmente, efetuou-se uma análise detalhada acerca do funcionamento do puzzle. Compreendidos todos estes detalhes, procedeu-se à análise da representação do problema e da respetiva solução de forma visualmente apelativa, bem como à representação interna das variáveis de decisão. Esta abordagem encontra-se detalhadamente descrita nas seguintes subsecções.

3.1 Variáveis de decisão

O programa desenvolvido recorre a duas estruturas distintas para obter a solução de uma dado puzzle. Estas estruturas correspondem a uma lista de listas cujo conteúdo difere de uma para a outra. Enquanto uma das estruturas é responsável por conter a informação acerca do puzzle, isto é, os elementos presentes em cada célula, a outra contém os dígitos a preencher, ou seja, as variáveis de decisão, cujo domínio varia entre 0 e 9. São estas variáveis de decisão que são sujeitas à aplicação de restrições e que, por sua vez, são transmitidas ao *labeling* que pesquisa por uma solução que satisfaz todas as restrições definidas, correspondendo, dessa forma, a uma solução ao puzzle.

3.2 Restrições

As restrições a aplicar foram já previamente descritas na descrição do problema (soma das linhas e restrições a cada elemento). Na implementação das restrições, foi necessário ter o cuidado de escolher restrições mais globais, permitindo à biblioteca *clpfd* do *SICStus* fazer melhores otimizações. Por exemplo, para a soma de vários dígitos, necessária para as restrições da soma das linhas, do diamante e do coração, foi utilizado o predicado *sum/3*.

A aplicação das restrições é efetuada, começando pela iteração das linhas da matriz do puzzle de forma a aplicar a restrição da soma das linhas, seguido pelas restrições a cada célula do puzzle, cuja restrição inerente é dependente do elemento presente nessa mesma célula.

3.3 Estratégia de pesquisa

Definidas todas as restrições ao problema, é efetuada a pesquisa de uma solução ao problema proposto. Para essa função, a biblioteca *clpfd* do *SICStus* disponibiliza um predicado - *labeling/2* - que etiqueta as variáveis de decisão fornecidas, usando um conjunto de opções à escolha do utilizador que permite especificar a estratégia de pesquisa.

A estratégia mais apropriada de pesquisa depende da instância do problema a resolver, logo a aplicação desenvolvida permite ao utilizador escolher ele próprio as opções de pesquisa. De uma forma geral, no que diz respeito à ordem de seleção das variáveis, é uma boa heurística selecionar a variável mais à esquerda (*leftmost - default*), tendo em conta que a etiquetagem de variáveis propaga restrições para o resto da linha (devido à soma da linha), sendo nesse caso preferível pesquisar linha a linha. Quanto à escolha do valor, a seleção do valor não é relevante (entre *step* e *bisect* p. ex.), porque o intervalo de valores do domínio é reduzido (10 valores). Para a escolha dos valores (forma ascendente - *up* - ou forma descendente - *down*), o valor das somas das linhas são uma boa heurística para perceber se a solução terá dígitos menores ou maiores.

4 Visualização da solução

O programa desenvolvido permite ao utilizador resolver dois tipos de puzzles: puzzles gerados dinamicamente e puzzles previamente definidos. Em qualquer uns dos casos, a visualização quer do problema quer da solução é disponibilizada ao utilizador. Inicialmente, é efetuada a representação visual do problema ao utilizador seguida da informação acerca da resolução do mesmo: tempo de resolução e respetivas estatísticas.

```
| ?- solve_puzzle([].1).  
#####  
#  
# ##### # # ##### ##### # # # # ##### # # ##### ##### #  
# ##### # # # # # # # ##### # # # # ##### # # # # ##### #  
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #  
# ##### ##### ##### ##### # # # # ##### # # # # ##### #  
# ##### ##### # # # # ##### # # # # ##### # # # # ##### #  
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #  
# ##### # # # # # # # # ##### ##### # # ##### # # # # ##### #  
# ##### # # # # # # # # ##### ##### # # ##### # # # # ##### #  
# ##### # # # # # # # # ##### ##### # # ##### # # # # ##### #  
#####  
  
Legend: Empty [ ], Star [*], Square [#], Diamond [+], Triangle [&], Circle [@], Chess Knight [♠], Heart [♥].  
  
|-----|-----|-----|-----|-----|-----|-----|-----|  
| # | ♠ | | + | # | @ | | 23  
|-----|-----|-----|-----|-----|-----|-----|-----|  
| # | # | ♤ | @ | # | # | | 20  
|-----|-----|-----|-----|-----|-----|-----|-----|  
| @ | | ♤ | + | ♡ | # | ♤ | 26  
|-----|-----|-----|-----|-----|-----|-----|-----|  
| ? | # | ♤ | | + | | # | 20  
|-----|-----|-----|-----|-----|-----|-----|-----|  
| ? | @ | + | @ | * | # | # | 26  
|-----|-----|-----|-----|-----|-----|-----|-----|  
| # | ♤ | # | # | # | # | ♡ | 19  
|-----|-----|-----|-----|-----|-----|-----|-----|  
| @ | ♤ | + | | @ | # | ♤ | 22  
|-----|-----|-----|-----|-----|-----|-----|-----|
```

Solving Time: 2ms

Statistics:

Resutations: 1763
Entailments: 267
Prunings: 1036
Backtracks: 0
Constraints created: 258

Press any key to show solution . . .

Fig. 2. Representação do problema e estatísticas de resolução.

Efetuada a representação do problema, é dada a possibilidade ao utilizador de o tentar resolver manualmente, ficando à escolha do utilizador o momento em que pretende visualizar a solução.

Press any key to show solution ...

0	3	6	9	0	4	1	23
5	0	4	4	5	0	2	20
4	1	2	7	6	5	1	26
7	0	1	1	9	2	0	20
3	4	7	4	3	0	5	26
5	2	0	5	0	5	2	19
4	1	5	7	4	0	1	22

yes
| ?=

Fig. 3. Representação da solução do problema.

Finalmente, após o sinal do utilizador, é representado a solução. Para efetuar ambas estas representações, recorreu-se à criação do predicado *display_puzzle/2* que recebe uma lista de listas (matriz bidimensional) e um estado que pode ser inicial ou final. No estado inicial, o predicado percorre todos os elementos e efetua a conversão destes na sua versão visual através do predicado *model_to_view* (representação do problema). No segundo estado, o predicado limita-se simplesmente a efetuar a representação do conteúdo da matriz, que por sua vez corresponde à solução do problema.

5 Resultados

Usando três instâncias predefinidas do problema, conseguimos comprovar com alguma confiança a exatidão da aplicação na resolução do problema. Para expandir o leque de problemas, foi desenvolvido um gerador de instâncias do problema. Ao contrário das instâncias predefinidas, as instâncias geradas não são tão completas, podem ter várias soluções e não há garantia de que vários problemas com o mesmo tamanho tenham a mesma complexidade. As instâncias geradas podem ser divididas em dois grupos; aquelas rapidamente resolvidas e aquelas cuja fase de pesquisa é demorada (podendo não existir solução). À medida que N aumenta, o segundo grupo ganha cada vez mais prevalência. Considerando o primeiro grupo, seguem-se dados que mostram a evolução de várias estatísticas relacionadas com a resolução do puzzle à medida a que se aumenta o tamanho do problema, usando as opções de pesquisa *default* (dimensão $N \rightarrow$ matriz $N \times N$):

Dimensões (N)	Tempo (ms)	Retomadas	Acarretamentos	Podas	Retrocessos	Restrições
5	0	395	68	669	0	68
10	1	2941	649	4125	17	345
15	4	3735	671	8374	1	627
20	17	8572	1457	16090	1	1373
25	27	13472	2423	25638	4	2256

Fig. 4. Evolução das estatísticas à medida que N aumenta.

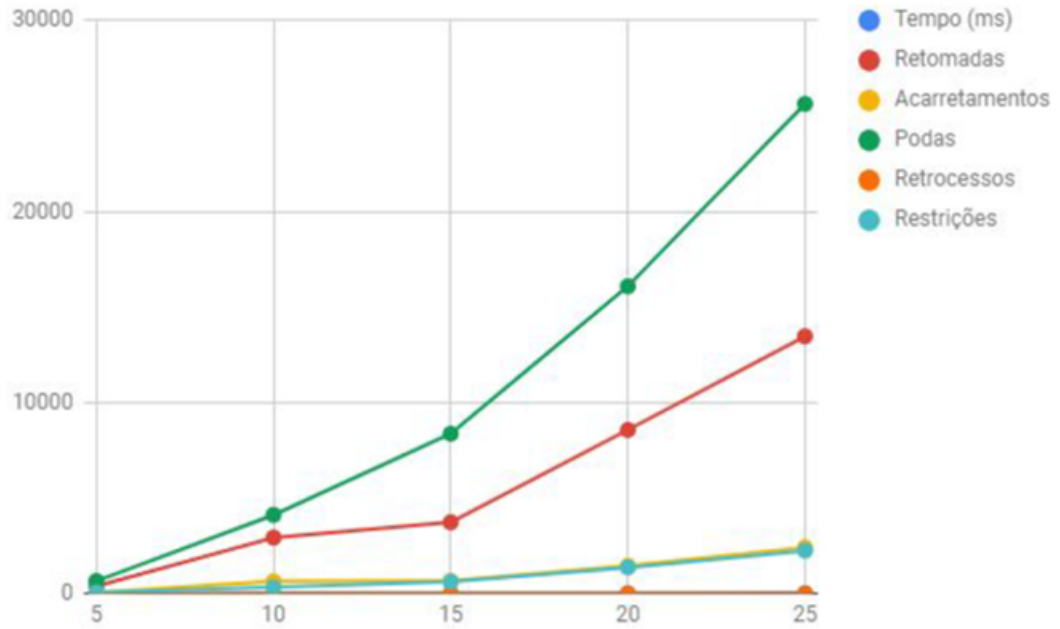


Fig. 5. Representação gráfica da evolução das estatísticas.

6 Conclusões e Trabalho Futuro

Tendo em conta as instâncias do problema predefinidas e geradas resolvidas, concluímos que a aplicação desenvolvida é capaz de resolver o problema proposto, caso exista resolução, apresentando uma das suas soluções. Quanto à eficiência, torna-se difícil chegar a uma conclusão dado que as soluções têm uma gama muito extrema de tempo de resolução. Esta gama já foi atenuada pela deteção de incoerências na geração em relação às restrições do problema (gerando instâncias potencialmente impossíveis que causavam numa pesquisa exaustiva de todas as possibilidades) entretanto resolvidas. É necessário analisar melhor o comportamento do gerador e o problema de forma mais teórica por forma a identificar a causa deste grupo extremo e concluir se se trata duma geração de problemas impossíveis e/ou se a resolução do problema é incapaz de propagar de forma significativa as restrições em certos tipos de problemas.

7 Bibliografia

1. Shapely Squares Problem, https://thegriddle.net/puzzledir/shapelysquares_2010_06_15.pdf.
2. SICStus Documentation, <https://sicstus.sics.se/documentation.html>.
3. LNCS Homepage, <http://www.springer.com/lncs>.

8 Anexos

8.1 Código fonte

[illegible]

[illegible]


```

% Dynamically generates a new puzzle

% -PuzzleInfo: Puzzle generated
% +Size: Size of the puzzle
generate_puzzle(PuzzleInfo, Size):- generate_numbers(PuzzleNums, Size, Size, []), !,
                                   generate_elements(PuzzleNums, PuzzleInfo, PuzzleNums, Size, Size).

% Generates random numbers for the whole puzzle

% -PuzzleNums: Puzzle with numbers generated
% +LineSize: Number of lines
% +ColSize: Number of columns
% +Sums: Last Line
generate_numbers([RowSums], 0, _, RevRowSums):- reverse(RevRowSums, RowSums).
generate_numbers([H|T], LineSize, ColSize, Sums):- generate_num_row(H, ColSize, 0, Sum),
                                                    DecLineSize is LineSize - 1,
                                                    generate_numbers(T, DecLineSize, ColSize,
[Sum|Sums]).

% Generates random numbers for a certain line

% +Line: Line to be generated
% +ColSize: Number of elements in line
% +CurrCol: Current column
% -Sum: Sum of all elements
generate_num_row([], 0, Sum, Sum).
generate_num_row([H|T], ColSize, Acc, Sum):- random(0, 10, H),
                                                    DecColSize is ColSize - 1,
                                                    NewAcc is Acc + H,
                                                    generate_num_row(T, DecColSize, NewAcc, Sum).

% Generates elements for the new puzzle

% +PuzzleNums: Puzzle with numbers generated
% -PuzzleInfo: Generated puzzle
% +PuzzleNums: Puzzle with numbers generated
% +LineSize: Number of lines
% +ColSize: Number of columns
generate_elements([RowSums], [RowSums], _, 0, _).
generate_elements([N|NS], [E|ES], PuzzleNums, LineSize, ColSize):- generate_row_elements(N, E, PuzzleNums, LineSize, ColSize), !,
                                                                    DecLineSize is LineSize - 1,
                                                                    generate_elements(NS, ES,
PuzzleNums, DecLineSize, ColSize).

% Generates line elements for the new puzzle

% +PuzzleNums: Puzzle with numbers generated
% -PuzzleInfo: Generated puzzle
% +PuzzleNums: Puzzle with numbers generated
% +LineSize: Number of lines
% +ColSize: Number of columns
generate_row_elements([], [], _, 0).
generate_row_elements([N|NS], [E|ES], PuzzleNums, LineSize, ColSize):- generate_element(N, E, PuzzleNums, LineSize, ColSize), !,
                                                                    DecColSize is ColSize
- 1,
                                                                    generate_row_elements
ES, PuzzleNums, LineSize, DecColSize).

% Generates Diamond Element

% -N: New value
% +Type: Element type
% +PuzzleNums: Puzzle with numbers generated
% +L: Number of lines
% +C: Number of columns
generate_element(N, 3, PuzzleNums, L, C):- member(N, [1, 3, 5, 7, 9]),
                                                    length(PuzzleNums, IncLS),
                                                    LI is IncLS - L,
                                                    nth1(LI, PuzzleNums, Line),
                                                    sublist(Line, LeftNums, 0, _, C),
                                                    sumlist(LeftNums, N).

% Generates Star Element

% -N: New value
% +Type: Element type
% +PuzzleNums: Puzzle with numbers generated
% +L: Number of lines
% +C: Number of columns
generate_element(N, 1, PuzzleNums, L, C):- member(N, [2, 3, 5, 7]),
                                                    PuzzleNums = [Line | Rest], length(Rest, LS), length(Line, CS),
                                                    star_validate(PuzzleNums, L, C, LS, CS, 0).

% Generates Chess Element

```

```

% -N: New value
% +Type: Element type
% +PuzzleNums: Puzzle with numbers generated
% +L: Number of lines
% +C: Number of columns
generate_element(N, 6, PuzzleNums, L, C):- PuzzleNums = [Line | Rest], length(Rest, LS), length(Line, CS),
chess_validate(N, PuzzleNums, L, C, LS, CS, 0, 0).

% Generates Square Element

% -N: New value
% +Type: Element type
% +PuzzleNums: Puzzle with numbers generated
% +L: Number of lines
% +C: Number of columns
generate_element(N, 2, PuzzleNums, L, C):- member(N, [0, 5]),

PuzzleNums = [Line | Rest], length(Rest, LS), length(Line, CS),
square_validate(N, PuzzleNums, L, C, LS, CS, 0).

% Generates Triangle Element

% -N: New value
% +Type: Element type
% +PuzzleNums: Puzzle with numbers generated
% +L: Number of lines
% +C: Number of columns
generate_element(N, 4, PuzzleNums, L, C):- N > 0,

PuzzleNums = [Line | Rest], length(Rest, LS), length(Line, CS),
get_num_by_direction(Above, PuzzleNums, L, C, -1, 0, LS, CS),
0 =:= Above mod 2,
N < Above.

% Generates Circle Element

% -N: New value
% +Type: Element type
% +PuzzleNums: Puzzle with numbers generated
% +L: Number of lines
% +C: Number of columns
generate_element(1, 5, _, _, _).

% Generates No Element

% -N: New value
% +Type: Element type
% +PuzzleNums: Puzzle with numbers generated
% +L: Number of lines
% +C: Number of columns
generate_element(_, 0, _, _, _).

% Star validation

% +PuzzleNums: Puzzle with numbers generated
% +L: Number of lines
% +C: Number of columns
% +LS: Line size
% +CS: Column size
% +Dir: Direction
star_validate(PuzzleNums, L, C, LS, CS, Dir):- neighbor(Dir, LI, CI),

get_num_by_direction(Number, PuzzleNums, L, C, LI, CI, LS,
CS),

!,
member(Number, [0, 4, 6, 8, 9]),
NewDir is Dir + 1,
star_validate(PuzzleNums, L, C, LS, CS, NewDir).

star_validate(4):- !.
star_validate(PuzzleNums, L, C, LS, CS, Dir):- !, Dir < 3,

NewDir is Dir + 1,
star_validate(PuzzleNums, L, C, LS, CS, NewDir).

% Chess knight validation

% -N: New value
% +PuzzleNums: Puzzle with numbers generated
% +L: Number of lines
% +C: Number of columns
% +LS: Line size
% +CS: Column size
% +Dir: Direction
% +Sum: Total sum
chess_validate(N, PuzzleNums, L, C, LS, CS, Dir, Sum):- attack_range(Dir, LI, CI),

get_num_by_direction(Number, PuzzleNums, L,
C, LI, CI, LS, CS),
member(Number, [0, 2, 4, 6, 8]),

```

```

NewDir, IncSum).
chess_validate(N, _ , _ , _ , 8, N):-!.
chess_validate(N, PuzzleNums, L, C, LS, CS, Dir, Sum):-!, Dir < 7,

NewDir, Sum).

% Square knight validation

% -N: New value
% +PuzzleNums: Puzzle with numbers generated
% +L: Number of lines
% +C: Number of columns
% +LS: Line size
% +CS: Column size
% +Dir: Direction
square_validate(N, PuzzleNums, L, C, LS, CS, Dir):-neighbor(Dir, LI, CI),

CI, LS, CS),

square_validate(_ , _ , _ , _ , 4):-!.
square_validate(N, PuzzleNums, L, C, LS, CS, Dir):-!, Dir < 3,

NewDir).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% API: solve_puzzle/2, solve_gen_puzzle/2, solve/5
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Solves a certain puzzle with specific ID

% +Options: Labeling options to optimize solution
% +Puzzle: Puzzle to be solved
solve_puzzle(Options, PuzzleID):-display_puzzle_info,

% ---- Puzzle information - INIT ----
puzzle(PuzzleID, PuzzleInfo),
PuzzleInfo = [FirstLine | _],
length(PuzzleInfo, TmpLineSize), LineSize is TmpLineSize - 1,
length(FirstLine, ColSize),
% ---- Puzzle information - END ----

solve(Options, PuzzleInfo, LineSize, ColSize, TmpLineSize).

% Solves a certain puzzle dynamically generated

% +Options: Labeling options to optimize solution
% +Size: Desired size for the generated puzzle
solve_gen_puzzle(Options, Size):-display_puzzle_info,

generate_puzzle(PuzzleInfo, Size),!,
TmpLineSize is Size + 1,
solve(Options, PuzzleInfo, Size, Size, TmpLineSize).

% Solves a certain puzzle

% +Options: Labeling options to optimize solution
% +PuzzleInfo: Internal Representation of the puzzle (facts)
% +LineSize: Number of lines
% +ColSize: Number of columns
% +TmpLineSize: Last line identifier
solve(Options, PuzzleInfo, LineSize, ColSize, TmpLineSize):-% ---- Variable puzzle generation - INIT ----

ColSize, Puzzle),

LastLine),

END ----

nl,

% ---- Solving puzzle - INIT ----

```

```

reset_timer, !,
apply_puzzle_constraints(Options,

Vars, Puzzle, PuzzleInfo, LineSize, ColSize, LastLine),

display_time,
% ---- Solving puzzle - END ----

% ---- Statistics ----
display_statistics,

% ---- Show solution ---- %
write('Press any key to show

solution ...'),

get_char(_),

!,
append(Puzzle, [LastLine], TMP),

nl,
display_puzzle(TMP, 'final'),

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Constraints
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% API: apply_puzzle_constraints/7, apply_row_constraints/4, apply_constraints/5, apply_line_constraints/7, apply_circle_remaining_constraint/2

% Applies the necessary constraints to the puzzle

% +Options: Options to optimize labeling
% +Vars: Domain variables to be constraint
% +Puzzle: Internal Representation of the puzzle (variables)
% +PuzzleInfo: Internal Representation of the puzzle (facts)
% +LineSize: Number of lines
% +ColSize: Number of columns
% +LastLine: Internal Representation of the last line of the puzzle to be displayed (fact)
apply_puzzle_constraints(Options, Vars, Puzzle, PuzzleInfo, LineSize, ColSize, LastLine):- domain(Vars, 0, 9), !,

LineSize, LastLine, 0), !,

PuzzleInfo, 0, LineSize, ColSize), !,

PuzzleInfo), !,

Vars). % Labeling options are the default

% Applies row row constraint by making sure that the sum of all elements in each row is equal to a certain value

% +Puzzle: Internal Representation of the puzzle (variables)
% +LineSize: Number of lines
% +LastLine: Internal Representation of the last line of the puzzle to be displayed (fact)
% +CurrLine: Current line
apply_row_constraints([], LastLine, _, LastLine):- !.
apply_row_constraints([H|T], LineSize, LastLine, CurrLine):- nth0(CurrLine, LastLine, LineSum),

safe_sum(H, #=, LineSum), !,
NewCurrLine is CurrLine + 1,
apply_row_constraints(T, LineSize,

LastLine, NewCurrLine).

% Applies to each element it's corresponding constraint (puzzle)

% +Puzzle: Internal Representation of the puzzle (variables)
% +PuzzleInfo: Internal Representation of the puzzle (facts)
% +CurrLine: Current line
% +LineSize: Number of lines
% +ColSize: Number of columns
apply_constraints(_, _, LineSize, LineSize, _):- !.
apply_constraints(Puzzle, PuzzleInfo, CurrLine, LineSize, ColSize):- nth0(CurrLine, Puzzle, Line), nth0(CurrLine, PuzzleInfo, LineInfo), !,

PuzzleInfo, Line, LineInfo, CurrLine, 0, ColSize),

apply_line_constraints(P,

NextLine is CurrLine +

1,

apply_constraints(Puzzle,

PuzzleInfo, NextLine, LineSize, ColSize).

% Applies to each element it's corresponding constraint (puzzle line)

% +Puzzle: Internal Representation of the puzzle (variables)
% +PuzzleInfo: Internal Representation of the puzzle (facts)
% +Line: A certain line of the puzzle
% +LineInfo: Information about the puzzle line
% +CurrLine: Current line
% +CurrCol: Current column
% +ColSize: Number of columns
apply_line_constraints(_, _, _, ColSize, ColSize):- !.
apply_line_constraints(Puzzle, PuzzleInfo, Line, LineInfo, CurrLine, CurrCol, ColSize):- NextCol is CurrCol + 1,

```

```

LineInfo, VarInfo),

Line, Var), !,

Var, Puzzle, PuzzleInfo, CurrLine, CurrCol), !,

PuzzleInfo, Line, LineInfo, CurrLine, NextCol, ColSize).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Element Constraints
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% API: apply_element_constraint/6

% Neighbors clockwise access - neighbor(Direction, LineInc, ColInc)
neighbor(0, -1, 0). % Up
neighbor(1, 0, 1). % Right
neighbor(2, 1, 0). % Down
neighbor(3, 0, -1). % Left

% Empty element constraints

% +Type: Specifies the type of the variable
% +Var: Variable to apply the constraints
% +Puzzle: Internal Representation of the puzzle (variables)
% +PuzzleInfo: Internal Representation of the puzzle (facts)
% +LineNo: Line number
% +ColNo: Column number
apply_element_constraint(0, _, _, _, _).

% Star element constraints

% +Type: Specifies the type of the variable
% +Var: Variable to apply the constraints
% +Puzzle: Internal Representation of the puzzle (variables)
% +LineNo: Line number
% +ColNo: Column number
apply_element_constraint(1, Var, Puzzle, _, LineNo, ColNo):- Var in {2, 3, 5, 7}, % Var is prime

LineSize), length(Line, ColSize), !,

LineSize, ColSize, LineNo, ColNo, 0).

% Square element constraints

% +Type: Specifies the type of the variable
% +Var: Variable to apply the constraints
% +Puzzle: Internal Representation of the puzzle (variables)
% +PuzzleInfo: Internal Representation of the puzzle (facts)
% +LineNo: Line number
% +ColNo: Column number
apply_element_constraint(2, Var, Puzzle, PuzzleInfo, LineNo, ColNo):- Var mod 5 #= 0, % Var is either 0 or 5

length(Puzzle, LineSize), length(Line, ColSize), !,

PuzzleInfo, Var, LineSize, ColSize, LineNo, ColNo, 0).

% Diamond element constraints

% +Type: Specifies the type of the variable
% +Var: Variable to apply the constraints
% +Puzzle: Internal Representation of the puzzle (variables)
% +LineNo: Line number
% +ColNo: Column number
apply_element_constraint(3, Var, Puzzle, _, LineNo, ColNo):- Var mod 2 #= 0, % Var is even

Var, LineNo, ColNo).

% Triangle element constraints

% +Type: Specifies the type of the variable
% +Var: Variable to apply the constraints
% +Puzzle: Internal Representation of the puzzle (variables)
% +LineNo: Line number
% +ColNo: Column number
apply_element_constraint(4, Var, Puzzle, _, LineNo, ColNo):- Var #= 0,

Retrieves the element directly above

LineInc, NewColNo is ColNo + ColInc, !,

LineSize), length(Line, ColSize), !,

```

```

Puzzle = [Line | _], length(Puzzle,

apply_star_constraints(Puzzle,

Puzzle = [Line | _],

apply_square_constraint

apply_diamond_constraints(Puzzle,

neighbor(0, LineInc, ColInc), %

NewLineNo is LineNo +

Puzzle = [Line | _], length(Puzzle,

apply_triangle_constraints(Puzzle,

```

```
Var, LineSize, ColSize, NewLineNo, NewColNo).

% Circle element constraints

% +Type: Specifies the type of the variable
% +Var: Variable to apply the constraints
apply_element_constraint(5, Var, _, _, _):- Var mod 3 #= 0. % Var is not multiple of 3

% Chess Knight element constraints

% +Type: Specifies the type of the variable
% +Var: Variable to apply the constraints
% +Puzzle: Internal Representation of the puzzle (variables)
% +LineNo: Line number
% +ColNo: Column number
apply_element_constraint(6, Var, Puzzle, _, LineNo, ColNo):- Puzzle = [Line | _], length(Puzzle, LineSize), length(Line, ColSize), !,
                                                                    apply_chess_knight_constraints(Var,
                                                                    Puzzle, LineSize, ColSize, LineNo, ColNo).

% Heart element constraints

% +Type: Specifies the type of the variable
% +Var: Variable to apply the constraints
% +Puzzle: Internal Representation of the puzzle (variables)
% +PuzzleInfo: Internal Representation of the puzzle (facts)
% +LineNo: Line number
% +ColNo: Column number
apply_element_constraint(7, Var, Puzzle, PuzzleInfo, LineNo, ColNo):- Puzzle = [Line | _], length(Puzzle, LineSize), length(Line, ColSize), !,
                                                                    apply_heart_constraints(
                                                                    Puzzle, PuzzleInfo, LineSize, ColSize, LineNo, ColNo).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Star Constraints
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% API: apply_star_constraints/6, star_constraint/5
% Auxiliary Predicates: check_boundaries, get_element

% Applies remaining constraints to the star element

% +Puzzle: Internal Representation of the puzzle (variables)
% +LineSize: Number of lines
% +ColSize: Number of columns
% +LineNo: Line number
% +ColNo: Column number
% +Direction: Direction where it should apply the constraint
apply_star_constraints(_, _, _, _, _, 4):- !.
apply_star_constraints(Puzzle, LineSize, ColSize, LineNo, ColNo, Direction):- neighbor(Direction, LineInc, ColInc),
                                                                    NewLineNo
                                                                    is LineNo + LineInc,
                                                                    NewColNo
                                                                    is ColNo + ColInc, !,
                                                                    star_constr
                                                                    NewDirecti
                                                                    apply_star_

                                                                    LineSize, ColSize, NewLineNo, NewColNo), !,

                                                                    is Direction + 1,

                                                                    LineSize, ColSize, LineNo, ColNo, NewDirection).

% Star constraint

% +Puzzle: Internal Representation of the puzzle (variables)
% +LineSize: Number of lines
% +ColSize: Number of columns
% +LineNo: Line number
% +ColNo: Column number
star_constraint(Puzzle, LineSize, ColSize, LineNo, ColNo):- check_boundaries(LineSize, ColSize, LineNo, ColNo),
                                                                    get_element(Puzzle, LineNo, ColNo,
                                                                    Neighbor),
                                                                    Neighbor in {0, 4, 6, 8, 9}.

star_constraint(_, _, _, _).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Square Constraints
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% API: apply_square_constraints/8, square_constraint/7
% Auxiliary Predicates: check_boundaries, get_element

% Applies remaining constraints to the square element

% +Puzzle: Internal Representation of the puzzle (variables)
% +PuzzleInfo: Internal Representation of the puzzle (facts)
% +Var: Variable to apply the constraint
```



```
%+LineSize: Number of lines
% +ColSize: Number of columns
% +LineNo: Line number
% +ColNo: Column number
% +Direction: Direction where it should apply the constraint
apply_square_constraints(_ _ _ _ 4):-!.
apply_square_constraints(Puzzle, PuzzleInfo, Var, LineSize, ColSize, LineNo, ColNo, Direction):- neighbor(Direction, LineInc, ColInc),
is LineNo + LineInc, NewColNo is ColNo + ColInc, !,
PuzzleInfo, Var, LineSize, ColSize, NewLineNo, NewColNo), !,
is Direction + 1,
PuzzleInfo, Var, LineSize, ColSize, LineNo, ColNo, NewDirection).

% Square constraint

% +Puzzle: Internal Representation of the puzzle (variables)
% +PuzzleInfo: Internal Representation of the puzzle (facts)
% +Var: Variable to apply the constraint
% +LineSize: Number of lines
% +ColSize: Number of columns
% +LineNo: Line number
% +ColNo: Column number
square_constraint(Puzzle, PuzzleInfo, Var, LineSize, ColSize, LineNo, ColNo):- check_boundaries(LineSize, ColSize, LineNo, ColNo),
get_element(
LineNo, ColNo, Neighbor),
get_element(
LineNo, ColNo, NeighborInfo),
NeighborIn
Var #!=
\= 3, % Neighbor is not a diamond
Var #!=

Neighbor .
square_constraint(_ _ _ _ _ _).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Diamond Constraints
% API: apply_diamond_constraints/4, get_left_elements/4

% Applies remaining constraints to the diamond element

% +Puzzle: Internal Representation of the puzzle (variables)
% +Var: Variable to apply the constraint
% +LineNo: Line number
% +ColNo: Column number
apply_diamond_constraints(Puzzle, Var, LineNo, ColNo):- nth0(LineNo, Puzzle, Line),
get_left_elements(Line, 0, ColNo, LeftElements),
safe_sum(LeftElements, #=, Var).

apply_diamond_constraints(_ , Var, _ _):- Var #= 9.

% Get all elements that are left of a certain element

% +PuzzleLine: Internal Representation of the puzzle line (variables)
% +CurrCol: Current column number
% +ColNo: Element column number
% -LeftElements: Elements on the left
get_left_elements(_ , ColNo, ColNo, []):-!.
get_left_elements([_], CurrCol, ColNo, [_|Rest]):- NewCurrCol is CurrCol + 1, !,
get_left_elements(T, NewCurrCol, ColNo,
Rest).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Triangle Constraints
% API: apply_triangle_constraints/6
% Auxiliary Predicates: check_boundaries, get_element

% Applies remaining constraints to the diamond element

% +Puzzle: Internal Representation of the puzzle (variables)
% +Var: Variable to apply the constraint
% +LineSize: Number of lines
% +ColSize: Number of columns
% +LineNo: Line number
% +ColNo: Column number
apply_triangle_constraints(Puzzle, Var, LineSize, ColSize, LineNo, ColNo):- check_boundaries(LineSize, ColSize, LineNo, ColNo),
get_element(Puzz
LineNo, ColNo, Neighbor),
Neighbor mod 2
```

Var #<

Neighbor.
apply_triangle_constraints(_ , _ , _ , _ , _).

%%
%% Circle Constraints
%%
% API: apply_circle_remaining_constraint/2, get_all_circles/4
% Auxiliary predicates: all_equal/1

% Applies remaining constraints to the circle element

% +Puzzle: Internal Representation of the puzzle (variables)
% +PuzzleInfo: Internal Representation of the puzzle (facts)
apply_circle_remaining_constraint(Puzzle, PuzzleInfo):- flatten(Puzzle, PuzzleFlatten), flatten(PuzzleInfo, PuzzleInfoFlatten),
get_all_circles(PuzzleFlatten, PuzzleInfoFlatten,
AllCircles, 5),
all_equal(AllCircles).

% Retrieves all circle variables present on the puzzle

% +Puzzle: Internal Representation of the puzzle (variables)
% +PuzzleInfo: Internal Representation of the puzzle (facts)
% -Circles: List of all circles
% +Circle: Circle type
get_all_circles([], [], _):- !.
get_all_circles([H1|T1], [Circle|T2], [H1|Rest], Circle):- !, get_all_circles(T1, T2, Rest, Circle).
get_all_circles([_|T1], [_|T2], AllCircles, Circle):- !, get_all_circles(T1, T2, AllCircles, Circle).

% Applies constraint that ensures that all elements are equal

% List: List containing all elements where constraint will be applied
all_equal([]):- !.
all_equal([_]):- !.
all_equal([H1,H2|T]):- H1 #= H2, !,
all_equal([H2|T]).

%%
%% Chess Knight Constraints
%%
% API: apply_chess_knight_constraints/6, get_attack_range_elements/7

% Chess Knight attack range clockwise access - attack_range(Direction, LineInc, ColInc)
attack_range(0, -2, 1). % Up-Right
attack_range(1, -1, 2). % Mid-Up-Right
attack_range(2, 1, 2). % Mid-Down-Right
attack_range(3, 2, 1). % Down-Right
attack_range(4, 2, -1). % Down-Left
attack_range(5, 1, -2). % Mid-Down-Left
attack_range(6, -1, -2). % Mid-Up-Left
attack_range(7, -2, -1). % Up-Left

% Applies constraints to the chess knight element

% +Var: Variable to apply the constraint
% +Puzzle: Internal Representation of the puzzle (variables)
% +LineSize: Number of lines
% +ColSize: Number of columns
% +LineNo: Line number
% +ColNo: Column number
apply_chess_knight_constraints(Var, Puzzle, LineSize, ColSize, LineNo, ColNo):- get_attack_range_elements(Puzzle, LineSize, ColSize,
LineNo, ColNo, Elements, 0), !,
safe_sum(!
#=: Var).

% Retrieves chess knight attack range elements

% +Puzzle: Internal Representation of the puzzle (variables)
% +LineSize: Number of lines
% +ColSize: Number of columns
% +LineNo: Line number
% +ColNo: Column number
% -Elements: Attack range elements
% -Direction: Direction where it should apply the constraint
get_attack_range_elements(_ , _ , _ , _ , [], 8):- !.
get_attack_range_elements(Puzzle, LineSize, ColSize, LineNo, ColNo, [Result|Rest], Direction):- attack_range(Direction, LineInc, ColInc),
is LineNo + LineInc, NewColNo is ColNo + ColInc,
ColSize, NewLineNo, NewColNo),

```

NewLineNo, NewColNo, Element),

mod 2 #= 0 #<=> Result, % If element is even then Result will be equal to 1. Otherwise Result is equal to 0

is Direction + 1,

LineSize, ColSize, LineNo, ColNo, Rest, NewDirection).
get_attack_range_elements(Puzzle, LineSize, ColSize, LineNo, ColNo, Elements, Direction):- NewDirection is Direction + 1,

LineSize, ColSize, LineNo, ColNo, Elements, NewDirection).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Heart Constraints
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% API: apply_heart_constraints/7, get_neighboring_hearts/8

% Applies constraints to the heart element

% +Var:      Variable to apply the constraint
% +Puzzle:   Internal Representation of the puzzle (variables)
% +PuzzleInfo: Internal Representation of the puzzle (facts)
% +LineSize: Number of lines
% +ColSize:  Number of columns
% +LineNo:   Line number
% +ColNo:    Column number
apply_heart_constraints(Var, Puzzle, PuzzleInfo, LineSize, ColSize, LineNo, ColNo):- get_neighboring_hearts(Puzzle, PuzzleInfo, LineSize,
ColSize, LineNo, ColNo, Neighbors, 0),

[Var], Elements), !,

#=: 10).

% Retrieves all heart neighbors

% +Puzzle:   Internal Representation of the puzzle (variables)
% +PuzzleInfo: Internal Representation of the puzzle (facts)
% +LineSize: Number of lines
% +ColSize:  Number of columns
% +LineNo:   Line number
% +ColNo:    Column number
% -Neighbors: Heart neighbors
% +Direction: Direction where it should apply the constraint
get_neighboring_hearts(, , , , , , 4):- !.
get_neighboring_hearts(Puzzle, PuzzleInfo, LineSize, ColSize, LineNo, ColNo, [Neighbor|Rest], Direction):- neighbor(Direction, LineInc, ColInc),

is LineNo + LineInc, NewColNo is ColNo + ColInc,

ColSize, NewLineNo, NewColNo),

NewLineNo, NewColNo, NeighborInfo),

#:= 7, % Neighbor is a heart

NewLineNo, NewColNo, Neighbor),

is Direction + 1,

PuzzleInfo, LineSize, ColSize, LineNo, ColNo, Rest, NewDirection).
get_neighboring_hearts(Puzzle, PuzzleInfo, LineSize, ColSize, LineNo, ColNo, Neighbors, Direction):- NewDirection is Direction + 1,

PuzzleInfo, LineSize, ColSize, LineNo, ColNo, Neighbors, NewDirection).

```