KenKen - Programação em Lógica
Resolução de Problema de decisão/optimização usando Restrições

André Duarte e João Santos

Faculdade de Engenharia da Universidade do Porto,
Rua Roberto Frias, sn, 4200-465 Porto, Portugal
FEUP-PLOG, Turma 3MIEIC03, Grupo 41
ei11044@fe.up.pt,ei11126@fe.up.pt
http://www.fe.up.pt

Resumo KenKen é um puzzle baseado numa grelha que usa operações aritméticas básicas - soma, subtração, multiplicação e divisão - para criar desafios lógicos. A grelha de um jogo KenKen é constituída por áreas chamadas *cages* às quais é atribuído um resultado e uma operação aritmética. O jogador tem que preencher as *cages* de forma a que os números nela contidos, quando associados à operação especificada, dêem os resultados esperados, garantido que em cada linha ou coluna o mesmo número não seja repetido. O objectivo deste trabalho é criar um algoritmo que crie e subsequentemente resolva grelhas kenken recorrendo à linguagem CLP utilizando o ambiente SICSTUS.

Keywords: KenKen, Puzzle, Restrições, Domínios, Prolog, CLP

1 Introdução

Com este trabalho pretende-se obter conhecimentos de boas práticas de programação em lógica recorrendo a restrições e domínios para resolver problemas de decisão. Foi também uma oportunidade para experimentarmos funções nativas do SICSTUS das quais não tínhamos conhecimento até agora. O objectivo deste trabalho é gerar e resolver um puzzle KenKen. O programa por nós desenvolvido gera um tabuleiro de tamanho variável NxN em que N pode ser qualquer inteiro de 3 a 9. O Tabuleiro é depois gerado pelo programa de acordo com as especificações fornecidas e posteriormente resolvido. As condicionantes serão melhor explicadas na secção 2 deste documento. No final, o programa mostra o problema gerado e a sua resolução.

2 Descrição do Problema

O nosso trabalho está dividido em dois módulos principais, que foram desenvolvidos separadamente e depois chamados em conjunto para resolver o problema geral.

2.1 Criação aleatória de tabuleiros KenKen

O tabuleiro é quadrado e está dividido em várias regiões, denominadas *cages*, com formas variadas. Cada linha deve conter todos os números de 1 a N (em que N é o lado do tabuleiro) exactamente uma vez. O mesmo deve ser garantido para cada coluna. A cada cage está atribuída uma operação aritmética e um resultado.

Para a criação de um tabuleiro aleatório tivemos que lidar com uma situação não prevista na organização inicial do projecto: o Prolog responde sempre da mesma forma ao mesmo problema de restrições se as únicas condicionantes forem os predicados domain/3 e all_different/1. Deste modo tivemos que arranjar uma maneira de fornecer números aleatórios sem quebrar as regras do tabuleiro. A nossa solução foi preencher a diagonal do tabuleiro com números aleatórios e desta forma garantimos que preenchendo o resto do tabuleiro recorrendo a restrições representativas das propriedades do tabuleiro obtemos sempre tabuleiros diferentes.

Para gerar as várias *cages* tivemos que garantir que as operações de subtração e divisão eram apenas atribuídas a *cages* com 2 elementos. No processo de criação da própria cage usamos um algoritmo semelhante ao de Prim. Começamos num elemento aleatório que não faça parte de nenhuma cage e a partir de aí construimos um caminho de tamanho aleatório e vamos acrescentando casas adjacentes aleatórias até chegar ao tamanho definido ou até não haverem mais possibilidades de caminho. É criada então uma grelha vazia e esta é então associada às cages para ser resolvida pelo programa, que depois devolve a nova grelha resolvida.

2.2 Resolução de um qualquer tabuleiro KenKen

Para resolver o problema tivemos apenas que seguir as restrições acima impostas e adicionar para cada *cage* uma restrição relativa aos seus membros, operação e resultado, tendo em conta o número de casas de cada *cage*.

3 Ficheiros de Dados

Apesar de não ser necessário, visto que geramos problemas aleatórios, definimos alguns problemas exemplo para poderem ser testados repetidamente.

Executando *kenken(X)*, sendo X um número de 3 a 8 inclusive, o programa resolve estes problemas exemplo. Para gerar problemas aleatórios, deverá ser executado *kenken(0, X)*, onde X é o tamanho do lado do tabuleiro.

Tanto a geração de problemas como os problemas exemplo encontram-se no ficheiro problems.pl.

4 Variáveis de Decisão

Cada elemento do Tabuleiro é uma variável de decisão e, visto que usamos uma lista de listas para representarmos o Tabuleiro, para fazermos o labeling usamos o append da seguinte forma:

```
% Build unified list
append(Board, UniBoard),
...
% Label Variables
labeling([], UniBoard).
```

5 Restrições

No nosso problema todas as restrições são rígidas, visto que as regras são todas inflexíveis.

5.1 Resolução do problema

Para resolver o jogo, restringimos todos os elementos de cada linha e cada coluna a serem diferentes:

```
% SETUP each row different
setupDifferent(Board),

% SETUP each column different
transpose(Board, TBoard),
setupDifferent(TBoard),
```

```
setupDifferent([]).
setupDifferent([H|T]):-
   all_distinct(H),
   setupDifferent(T).
```

Também restringimos a multiplicação/soma/subtração/divisão dos elementos de cada *cage* a ser igual ao resultado esperado:

```
% SETUP groups
( foreach(P,Problem),
  param(Board)
  do
   setupGroup(Board, P)
),
```

Grupos unitários:

```
setupGroup(Board, [Operation, Spaces, Result]) :-
    length(Spaces, Length),
    Length =:= 1,
    [[Row,Column]] = Spaces,
    getElement(Board, Row, Column, Position),
    Position is Result.
```

Grupos de tamanho 2:

```
setupGroup(Board, [Operation, Spaces, Result]) :-
    length(Spaces, Length),
    Length =:= 2,
    [[Row1,Column1],[Row2,Column2]] = Spaces,
    getElement(Board, Row1, Column1, Position1),
    getElement(Board, Row2, Column2, Position2),
    checkGroup(Op, Position1, Position2, Result).
```

```
checkGroup(1, Position1, Position2, Result) :- Position1 +
    Position2 #= Result.
checkGroup(2, Position1, Position2, Result) :- Position1 *
    Position2 #= Result.
checkGroup(3, Position1, Position2, Result) :- Position1 -
    Position2 #= Result.
checkGroup(4, Position1, Position2, Result) :- Position2 *
    Result #= Position1.
```

Grupos de tamanho maior que 2:

```
setupGroup(Board, [Operation, Spaces, Result]) :-
    length(Spaces, Length),
    Length > 2,
    (foreach([Row,Column],Spaces),
     foreach(Value, UniSpaces),
     param(Board)
     do
         getElement(Board, Row, Column, Value)
    ),
    checkGroup(Operation, UniSpaces, Result).
```

```
checkGroup(1, Positions, Result) :- sum(Positions,#=,Result).
checkGroup(2, Positions, Result) :- prod(Positions,Result).
```

```
prod(List, Result) :-
        (foreach(L, List),
         fromto(1, In, Out, Result) do
             Out #= In * L
        ).
```

5.2 Criação do problema

As restrições usadas na fase de criação do tabuleiro base (sem *cages*) são idênticas no que diz respeito aos elementos serem diferentes em cada linha e coluna. No entanto, de modo a garantir a aleatoriedade, colocamos elementos aleatórios na diagonal antes do *labeling*.

6 Estratégia de Pesquisa

Recorrendo ao método experimental verificamos que para o nosso problema ao especificar apenas a opção bissect obtemos os resultados mais rápidos e com menor utilização de recursos.

7 Visualização do Problema

O problema é representado *cage* a *cage*, sendo cada *cage* uma lista com o operador, lista de casas e resultado.

```
Problem:
[+,[[1,1],[1,2],[1,3]],7]
[+,[[1,4],[2,4],[2,3]],9]
[+,[[1,5],[2,5],[3,5]],10]
[-,[[2,1],[3,1]],1]
[+,[[2,2],[3,2],[4,2],[4,3]],13]
[*,[[3,3],[3,4],[4,4],[4,5],[5,5]],96]
[*,[[4,1],[5,1],[5,2],[5,3],[5,4]],600]
```

8 Visualização da Solução

De forma a representar a solução ao problema, apresentamos o tabuleiro com os números respectivos em cada posição. Ficando da seguinte maneira:

```
Resolution:
[4,1,2,3,5]
[2,4,5,1,3]
[1,5,3,4,2]
[5,3,1,2,4]
[3,2,4,5,1]
```

9 Visualização das Estatíscas

Para que pudéssemos comparar a execução com as diferentes optimizações recorremos à biblioteca *system* para determinar o tempo de execução e a *fd_statistics* para obter informações sobre o processamento das restrições necessárias para a resolução do problema.

```
Statistics:
Execution time: 0 seconds
Resumptions: 3807
Entailments: 555
Prunings: 2993
Backtracks: 48
Constraints created: 62
```

10 Resultados

Os resultados obtidos foram bastante positivos, tanto na geração de problemas aleatórios como na resolução dos mesmos.

Conseguimos atingir um baixo tempo de execução, através da diminuição do *Backtracks* executados e o menos uso possível de *constraints*.

O tempo de execução varia dependendo da complexidade do problema e do tamanho do mesmo, como também poderá variar de computador para computador. Mesmo assim deixamos aqui alguns dos resultados obtidos como referência.

10.1 Problema Exemplo nº 5

```
Problem:
[+,[[1,1],[1,2],[1,3]],7]
[+,[[1,4],[2,4],[2,3]],9]
[+,[[1,5],[2,5],[3,5]],10]
[-,[[2,1],[3,1]],1]
[+,[[2,2],[3,2],[4,2],[4,3]],13]
[*,[[3,3],[3,4],[4,4],[4,5],[5,5]],96]
[*,[[4,1],[5,1],[5,2],[5,3],[5,4]],600]

Resolution:
[4,1,2,3,5]
[2,4,5,1,3]
[1,5,3,4,2]
[5,3,1,2,4]
[3,2,4,5,1]

Statistics:
Execution time: 0 seconds
Resumptions: 3807
Entailments: 555
Prunings: 2993
Backtracks: 48
Constraints created: 62
```

10.2 Problema Gerado com tamanho 7

```
Problem:
[+,[[1,1],[2,1],[3,1],[4,1]],13]
[+,[[1,2],[2,2],[3,2],[3,3],[2,3]],14]
[*,[[1,3],[1,4],[1,5],[2,5],[2,6]],540]
[+,[[1,6],[1,7],[2,7],[3,7],[4,7],[5,7],[5,6]],30]
[*,[[2,4],[3,4],[3,5],[4,5],[4,6],[3,6]],350]
[*,[[4,2],[5,2],[5,1],[6,1],[7,1]],1575]
[+,[[4,3],[5,3],[5,4],[6,4],[6,3],[7,3]],28]
[+,[[4,4]],6]
[*,[[5,5],[6,5],[7,5],[7,6],[6,6],[6,7],[7,7]],2880]
[*,[[6,2],[7,2]],42]
[+,[[7,4]],4]

Resolution:
[1,4,2,3,5,7,6]
[2,1,4,5,3,6,7]
[6,2,3,1,7,5,4]
[4,5,7,6,1,2,3]
[5,3,6,7,4,1,2]
[3,7,1,2,6,4,5]
[7,6,5,4,2,3,1]

Statistics:
Execution time: 0 seconds
Resumptions: 55348
Entailments: 5992
Prunings: 49973
Backtracks: 715
Constraints created: 652
```

10.3 Problema Exemplo nº 8

```
Problem:
[+,[[1,1],[2,1],[2,2],[1,2],[1,3],[2,3],[2,4],[3,4]],19]
[*,[[1,4],[1,5],[2,5],[2,6],[2,7],[2,8],[3,8],[3,7]],645120]
[+,[[1,6],[1,7]],13]
[*,[[1,8]],8]
[*,[[3,1],[4,1],[5,1],[5,2],[5,3],[5,4],[4,4],[4,3]],241920]
[+,[[3,2],[3,3]],6]
[*,[[3,5],[4,5],[4,6],[5,6],[6,6],[6,7],[7,7],[7,8]],40320]
[*,[[3,6]],8]
[+,[[4,2]],3]
[*,[[4,7],[4,8],[5,8],[5,7]],70]
[+,[[5,5],[6,5],[6,4],[7,4],[7,3],[6,3],[6,2]],35]
[*,[[6,1],[7,1],[8,1],[8,2],[7,2]],13440]
[*,[[6,8]],2]
```

```
[-,[[7,5],[8,5]],-1]
[+,[[7,6],[8,6],[8,7],[8,8]],11]
[-,[[8,3],[8,4]],-3]

Resolution:
[1,2,3,4,5,6,7,8]
[2,1,5,3,4,7,8,6]
[3,5,1,2,7,8,6,4]
[4,3,2,6,8,1,5,7]
[7,6,8,5,3,4,2,1]
[8,4,7,1,6,5,3,2]
[5,7,6,8,1,2,4,3]
[6,8,4,7,2,3,1,5]

Statistics:
Execution time: 0 seconds
Resumptions: 2324
Entailments: 264
Prunings: 1729
Backtracks: 15
Constraints created: 140
Constraints created: 652
```

10.4 Problema Exemplo nº 9

```
Problem:
[-,[[1,1],[1,2]],2]
[+,[[1,3],[2,3],[3,3]],8]
[*,[[1,4],[2,4],[2,5],[2,6],[1,6],[1,7]],3780]
[+,[[1,5]],5]
[+,[[1,8],[2,8],[3,8],[3,7],[4,7],[4,8],[5,8]],38]
[+,[[1,9],[2,9],[3,9]],23]
[*,[[2,1],[3,1],[3,2],[2,2]],24]
[+,[[2,7]],8]
[+,[[3,4],[4,4]],7]
[*,[[3,5],[4,5]],56]
[+,[[3,6],[4,6]],17]
[+,[[4,1],[4,2],[5,2],[5,1],[6,1],[6,2],[6,3],[5,3]],44]
[*,[[4,3]],6]
[*,[[4,9],[5,9],[6,9],[7,9],[7,8],[7,7],[8,7],[8,8],[8,9]],33600]
[+,[[5,4],[5,5],[6,5]],14]
[+,[[5,6],[6,6],[6,7],[5,7]],18]
[+,[[6,4],[7,4],[7,5],[7,6],[8,6],[9,6],[9,5],[9,4],[9,3]],43]
[*,[[6,8]],3]
[*,[[7,1],[8,1],[9,1],[9,2]],3456]
[+,[[7,2],[8,2],[8,3],[7,3]],25]
[-,[[8,4],[8,5]],1]
[*,[[9,7],[9,8],[9,9]],72]
```

```
24
25 Resolution:
26 [4,2,1,3,5,6,7,9,8]
27 [2,3,4,1,6,5,8,7,9]
28 [1,4,3,2,7,8,9,5,6]
29 [3,1,6,5,8,9,2,4,7]
30 [5,6,8,9,4,7,1,2,3]
31 [7,5,9,8,1,4,6,3,2]
32 [8,7,2,6,9,3,4,1,5]
33 [6,9,7,4,3,2,5,8,1]
34 [9,8,5,7,2,1,3,6,4]
35
36 Statistics:
37 Execution time: 0 seconds
38 Resumptions: 268155
39 Entailments: 23844
40 Prunings: 257699
41 Backtracks: 2938
42 Constraints created: 569
```

11 Conclusões

Com este trabalho fomos capazes de perceber o quão utéis poderão ser as restrições em Prolog e como estas poderão simplificar a resolução de alguns problemas, que noutra linguagem seria muito mais complexa. Melhoramos substancialmente o conhecimento acerca desta linguagem, o que fez com que o nosso trabalho fosse estruturado e pensado desde o início, ao contrário do que aconteceu com o 1º trabalho, o que nos permitiu melhorar significativamente tanto a performance da aplicação como a nossa produtividade neste projeto.

Conseguimos assim criar uma solução fiável de resolução de problemas KenKen e uma solução totalmente aleatória, à prova de erros, de geração de problemas deste jogo. Pelo que estamos bastante satisfeitos com as nossas melhorias e orgulhosos do nosso trabalho.

Referências

1. KenKen Math Puzzles, http://www.kenken.com