Programação em Logica – Aztec Math

David Dinis[up201706766] e José Gomes[up201707054]

Faculdade de Engenharia da Universidade do Porto
Mestrado Integrado em Engenharia Informática e Computação
FEUP-PLOG, Turma3MIEIC5, Grupo Aztec_Math_2

Resumo. Este projeto foi desenvolvido no âmbito da unidade curricular Programação em Logica e teve como objetivo a construção de um programa em Programação em Lógica com Restrições para a resolução de um problema de otimização/decisão combinatória. O puzzle escolhido tem o nome de Aztec Math e consiste em completar uma pirâmide de números na qual cada número resulta de uma de quatro operações aritméticas entre os dois números imediatamente abaixo.

Keywords: Aztec Math, Prolog, Restrições, SICStus.

1 Introdução

O projeto foi desenvolvido no âmbito da unidade curricular de Programação em Lógica de 3º ano do curso Mestrado Integrado em Engenharia Informática e Computação. Para tal, foi necessário implementar uma possível resolução para um problema de decisão ou otimização em *Prolog*, com restrições. Foi escolhido um problema de decisão chamado de "Aztec Math".

O problema escolhido consiste em preencher uma pirâmide de números inteiros de forma a que cada número seja resultado de uma operação de soma, subtração, multiplicação ou divisão entre os dois números imediatamente abaixo. Também existe o requisito de que, em cada linha da pirâmide, não podem existir números repetidos.

Este artigo tem a seguinte estrutura:

- **Descrição do Problema** – descrição em detalhe do problema de decisão.
- **Abordagem** – Descrição da modelação do problema como um PSR de acordo com as seguintes subsecções.
  - **Variáveis de Decisão –** Descrição das variáveis de decisão e os seus domínios e também os seus significados no contexto do problema.
  - **Restrições –** Descrever as restrições rígidas e flexíveis do problema e a sua implementação utilizando o *SICStus Prolog*.
  - **Estratégia de Pesquisa** – Descrever a estratégia de etiquetagem utilizada, nomeadamente heurísticas de ordenação de variáveis e valores.
- **Visualização da Solução –** Explicação os predicados que permitem visualizar a solução em modo de texto.

- **Gerador de Puzzles –** Explicação do funcionamento do gerador de puzzles e observações à cerca do mesmo
- **Resultados –** Demonstrações de vários exemplos de aplicação e analise dos resultados obtidos.
- **Conclusão –** Conclusões retiradas deste projeto, vantagens e limitações da solução obtida e aspetos a melhorar.
- **Anexo –** código fonte, ficheiros de dados e resultados, entre outros.

2 Descrição do Problema

O Puzzle "Aztec Math" é um problema de decisão. O puzzle consiste numa pirâmide de N (N <= 9) níveis em que cada linha da pirâmide tem mais um elemento que a linha imediatamente acima. A pirâmide está semipreenchida com números inteiros de 1 a 9, inclusive. O problema consiste em preencher os espaços em branco de forma a que **não existam** números repetidos em cada linha e a que cada elemento seja o **resultado de uma operação** de soma, subtração, multiplicação ou divisão **entre os dois números imediatamente abaixo**. A ordem das operações não é relevante, sendo que os dois números inferiores podem estar de qualquer lado da operação.

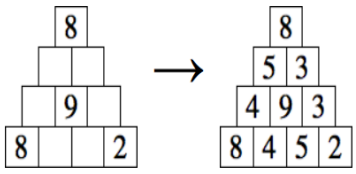

Fig. 1. Exemplo de um puzzle de 4 níveis

3 Abordagem

Para resolver este problema na linguagem *Prolog* foi utilizada uma lista de listas para representar a pirâmide do problema (*Board*). Cada linha da pirâmide corresponde a uma lista dentro da lista principal. Cada elemento das listas interiores representa um número inteiro pertencente à pirâmide.

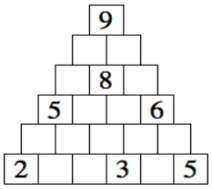

Fig. 2. Exemplo de um puzzle de 6 níveis

Assim, considerando o exemplo acima (**Fig.2)**, a representação em forma de lista de listas seria da forma: *Board* = [[9], [A,B], [C,8,D], [5,E,F,6], [G,H,I,J,K], [2,L,M,3,N,5]]. Para motivos de resolução do problema, os espaços em branco podem ser representados por "_" ou por qualquer átomo. Para resolver um puzzle é chamado o predicado *aztec/1*.

3.1 Variáveis de Decisão

A solução do problema vem na mesma lista de input (*Board*) e com o mesmo formato de lista de listas. Os espaços em branco ou átomos são preenchidos de acordo com as restrições aplicadas.

```
| ?- aztec([[7], [A, B], [C, D, 9], [2, E, F, G], [H, I, J, 7, K],[4,L,1,M,N,5]]).
A = 2,
B = 9,
C = 3,
D = 1,
E = 5,
F = 6,
G = 3,
H = 3,
I = 6,
J = 1,
K = 4,
L = 7,
M = 2,
N = 9 ?
```

Fig. 3. Exemplo da resolução de um puzzle de 6 níveis

O tamanho da lista é o mesmo tamanho da lista inicial. Cada elemento continua a representar uma linha da pirâmide do problema e cada elemento das listas interiores representa um número inteiro.

3.2 Restrições

Os elementos de cada lista interior têm de ser todos distintos (Restrição 1). Segundo as regras do puzzle, em cada linha da pirâmide, não podem existir números repetidos. Para cumprir esta regra foi aplicado o predicado de restrição *all_different/1* para todas as listas interiores.

Cada elemento resulta de uma operação aritmética entres os dois elementos imediatamente abaixo (Restrição 2). Para um dado elemento A e os elementos imediatamente abaixo B e C, o valor do elemento A resulta de uma das seguintes operações:

- A = B + C
- A = B – C ou A = C - B
- A = B * C

- A = B / C ou A = C / B.

Todas as restrições são aplicadas através do predicado *constraints_list/2* que recebe como primeiro argumento uma lista interior da *Board* e no segundo argumento a lista seguinte.

```
constraints_list([H|List1], [H1 | [H2 | List2]]) :-
    H in 1..9, H1 in 1..9, H2 in 1..9,
    all_different([H|List1]),
    all_different([H1 | [H2 | List2]]),
    (H #= H1 + H2 #\/ %SOMA
    H #= abs(H1 - H2) #\/ %SUBTRAÇÃO
    H #= H1 * H2 #\/ % MULTIPLICAÇÃO
    % DIVISÂO LEFT RIGHT
    ( H1 #> H2 #/\ 0 #= mod(H1, H2) #/\ H #= H1 / H2) #\/
    % DIVISÃO RIGHT LEFT
    ( H1 #< H2 #/\ 0 #= mod(H2, H1) #/\ H #= H2 / H1)),
    constraints_list(List1, [H2 | List2]).
```

O predicado “separa” a primeira lista em H e List1, sendo H o primeiro elemento da lista e List1 os restantes. “Separa” também a segunda lista no seu primeiro elemento (H1), no segundo (H2) e em List2 fica a restante lista.

A **Restrição1** é aplicada através duas chamadas ao predicado *all_different/1*, uma vez para cada lista e a **Restrição2** é aplicada nas linhas seguintes. Depois de aplicadas todas as restrições relativas ao primeiro elemento da primeira lista (H1) o predicado *constraints_list/2* é chamado outra vez, mas desta vez sem os primeiros elementos das duas listas. A recursividade termina quando a primeira lista está vazia.

3.3 Estratégia de Pesquisa

Para determinar qual a melhor estratégia de pesquisa, foram testados vários modos de ordenação de valores para o mesmo modo de seleção de valores *(“step”*). Os testes apresentados na **Tabela 1** foram efetuados todos com o mesmo puzzle de 6 níveis. A *Board* para o puzzle foi a seguinte: [[9], [A, B], [C, 8, D], [5, E, F, 6], [G, H, I, J, K], [2,L,M,3,N,5]]. Através dos resultados obtidos concluímos que a melhor estratégia a usar seria utilizar *“occurrence”* como modo de ordenação e *“step”* como modo de seleção de valores.

4 Visualização da solução

Para uma melhor visualização do puzzle resolvido foi criado o predicado *displayPuzzle/1* que imprime no ecrã o puzzle passado no seu argumento. Este predicado recebe no seu argumento uma lista de listas tal como o predicado *aztec/1* e percorre recursivamente a lista de listas imprimindo cada linha com "N-n" espaços a antecedê-la (sendo "N" o número de níveis da pirâmide e "n" o número da linha).

```
displayPuzzle(List):-
    length(List, Size),
    NewSize is Size -1,!, nl,
    displayPuzzleAux(List, NewSize).

displayPuzzleAux([Line| List], Size):-
    nTabs(Size),
    write(Line), nl,
    NewSize is Size -1, !,
    displayPuzzleAux(List, NewSize).

nTabs(X):-
    write(' '),
    X1 is X-1,
    nTabs(X1).
```

Os predicados *nTabs/1* e *displayPuzzleAux/2* são auxiliares ao *displayPuzzle/1*. Os predicados que param a recursividade não estão presentes no código acima.

5 Gerador de Puzzles

Foi criado um gerador de puzzles que, dado um número de níveis, gera um puzzle de **solução única** para ser resolvido. Para criar um puzzle é preciso chamar o predicado *generate/1* em que o primeiro argumento é o número de níveis da pirâmide pretendida.

O gerador utiliza uma abordagem de *bruteforce* para criar um puzzle de solução única. Ao chamar o predicado de geração, é criada uma lista de listas em que cada elemento das listas interiores tem 20% de probabilidade de ser um número aleatório de 1 a 9 e 80% de probabilidade de ser um espaço vazio. Após gerar a lista de listas é chamado o predicado *aztec/1* com a lista gerada e se existir apenas uma solução para essa *Board* então é apresentada como resposta. Caso esse puzzle tenha mais do que uma solução, é gerada outra lista aleatória e o predicado tenta de novo resolver o puzzle com apenas uma solução.

Os resultados obtidos com esta solução são muito inconsistentes pois depende de uma lista aleatória e não tem qualquer raciocínio lógico na decisão do puzzle.

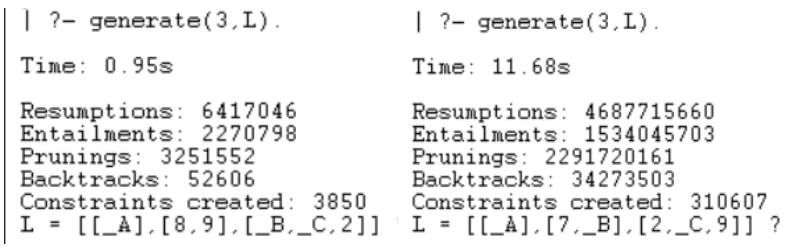

Fig. 4. Exemplos de geração de puzzles de 3 níveis

Como se pode ver na **Figura 4**, os tempos de geração são muito inconstantes. Durante os testes efetuados apenas foi conseguido gerar um puzzle de nível 4 que durou 135 segundos na sua geração.

6 Resultados

Foram medidos o número de retrocessos, número de restrições colocadas e o tempo de execução de puzzles de vários tamanhos. Os resultados estão presentes na **Tabela 2.**

Através dos dados obtidos, apenas é possível concluir que todas as variáveis estudadas (número de retrocessos, número de restrições colocadas e tempo de execução) aumentam de uma forma não linear com o número de níveis.

7 Conclusões e trabalho futuro

O projeto teve como principal objetivo aplicar o conhecimento adquirido nas aulas teóricas e práticas, e foi concluído que o módulo de restrições da linguagem Prolog é extremamente útil para problemas de decisão como o do "Aztec Math".

Durante o desenvolvimento deste projeto foram sentidas algumas dificuldades, nomeadamente no que diz respeito à utilização e modificação de listas em Prolog. Esta dificuldade fez com que o gerador de puzzles não ficasse tão eficiente como o esperado.

Embora concluído, o trabalho poderia ser melhorado, especialmente no que toca à abordagem *bruteforce* do gerador de puzzles. O predicado *aztec/1* também poderia ser melhorado arranjando uma alternativa para a dupla chamada ao predicado *all_different/1* cuja necessidade veio da dificuldade já referida de "manobrar" listas em Prolog".

Em suma, o projeto foi concluído com sucesso uma vez que soluciona corretamente e bastante rápido qualquer puzzle de "Aztec Math" e o desenvolvimento do mesmo contribuiu de forma positiva para uma melhor compreensão do funcionamento de Programação em Lógica com Restrições.

8 Bibliografia

1. Friedman, Erich. “Aztec Math Puzzles.” *Aztec Math Puzzles*, 2010, www2.stetson.edu/~efriedma/puzzle/aztec/. Acedido a 21 de dezembro de 2019.

9 Anexos

Tabela 1. Variação do modo de ordenação de valores em modo “step” para seleção de valores

Estratégia de Pesquisa	Tempo(s)
leftmost	0.02
min	0.14
max	0.06
first_fail	0.04
anti_first_fail	0.14
occurrence	0.00
ffc	0.01
max_regret	0.04

Tabela 2. Variação do tamanho do puzzle

Nº de Níveis	Tempo(s)	Retrocessos	Restrições
2	0.0	1	31
3	0.00	18	116
4	0.06	76	224
5	0.04	3568	561

9.1 Código fonte

```
:- use_module(library(clpfd)).
:- use_module(library(lists)).
:- use_module(library(random)).

appendlist([], Final, Final).

appendlist([H|T], List, Final) :-
    append(List,H, NewList),
    appendlist(T, NewList, Final).
```

```
aztec(List) :-
    appendlist(List, [], NewList),
    aux_aztec(List),
    catch(labeling([], NewList),_,fail),

aux_aztec([H|[]]).

aux_aztec([List|[ List1 | List2]]) :-
    constraints_list(List, List1),
    aux_aztec([List1 | List2]).

constraints_list([], List1).

constraints_list([H|List1], [H1 | [H2 | List2]]) :-
    H in 1..9, H1 in 1..9, H2 in 1..9,
    all_different([H|List1]),
    all_different([H1 | [H2 | List2]]),
    (H #= H1 + H2 #\/ %SOMA
    H #= abs(H1 - H2) #\/ %SUBTRAÇÃO
    H #= H1 * H2 #\/ % MULTIPLICAÇÃO
    ( H1 #> H2 #/\ 0 #= mod(H1, H2) #/\ H #= H1 / H2) #\/ %
DIVISÂO LEFT RIGHT
    ( H1 #< H2 #/\ 0 #= mod(H2, H1) #/\ H #= H2 / H1)), %
DIVISÃO RIGHt LEFT
    constraints_list(List1, [H2 | List2]).

nTabs(0).
nTabs(X):-
    write(' '),
    X1 is X-1,
    nTabs(X1).

displayPuzzle(List):-
    length(List, Size),
    NewSize is Size -1,!, nl,
    displayPuzzleAux(List, NewSize).

displayPuzzleAux([],_):-
    nl.

displayPuzzleAux([Line| List], Size):-
    nTabs(Size),
    write(Line), nl,
    NewSize is Size -1, !,
    displayPuzzleAux(List, NewSize).
```

```
randOrAtom(C):-
    maybe(0.2) -> random(0,10,C); true.

createPyramid(Size, CurrLevel, Board, FinalBoard):-
    CurrLevel =< Size,
    length(Row, CurrLevel), maplist(randOrAtom, Row),
    append(Board,[Row], Pyramid),
    NextLevel is CurrLevel + 1,
    createPyramid(Size, NextLevel, Pyramid, FinalBoard).

createPyramid(_,_,FinalBoard, FinalBoard).

try(Size, Res):-
    createPyramid(Size, 1, [], Board),
    findall(Board, aztec(Board), Possible),
    length(Possible, 1),
    length(Board, Size),
    append(Board,[], Res).

generate(Size, Board):-
    repeat,
    try(Size, Board).
```