Crypto Product Solver

João Renato Pinto[1,2,3[up201705547]] and Tiago Gonçalves Gomes[1,2,3[up201806658]]

[1] Faculdade de Engenharia da Universidade do Porto - PLOG
[2] Turma 3MIEIC06
[3] Grupo Crypto_Product_05

Resumo. No contexto da Unidade Curricular de Programação Lógica foi nos proposta a escrita de um artigo relativo ao desenvolvimento de um *Solver* para os Puzzles desenvolvidos pelo professor Erich Friedman [1], *Crypto Product*, usando a Programação em Lógica por Restrições. Serão apresentadas as restrições do problema e a abordagem escolhida para o resolver.

1 Introdução

Este artigo irá apresentar um programa escrito em SICSTUS Prolog com o intuito de resolver Puzzles *Crypto Product* [2], começando por fazer uma descrição das regras dos puzzles, para de seguida proceder a uma explicitação da abordagem utilizada para a resolução dos problemas.

Será também demonstrada a visualização do problema e os respetivos resultados, terminando com uma sucinta conclusão de todo o desenvolvimento.

2 Descrição do Problema

Crypto Product é um problema de decisão. Cada puzzle é uma multiplicação de círculo com 2 operandos e 1 resultado e segue uma série de regras:

- Círculos de cores semelhantes contém o mesmo dígito.
- Cada puzzle tem uma solução única.
- Cada círculo tem 1 e 1 só dígito.
- Multiplicação tem de ser verdadeira.
- O resultado tem de ser positivo.

Fig. 1. Exemplo de puzzle *Crypto Product* resolvido

3 Abordagem

3.1 Variáveis de Decisão

Um puzzle *Crypto Product* é representado por 4 listas, uma para cada um dos operandos, outra para o resultado, e a última representa as variáveis. Cada uma das cores utilizadas é representada por uma variável. Para resolver o problema cada uma das variáveis tem de ser substituída com um inteiro de domínio 0 a 9.

Para otimizar o *Solver*, são calculados os domínios possíveis para cada um dos operandos e para o resultado, sendo que cada operando não pode começar por 0, o seu valor mínimo é dado pela fórmula **OperandMin = $10^{(\text{LengthOperand} - 1)}$** e o valor máximo por **OperandMax = $10^{\text{LengthOperand}}$**.

```
crypto_product(Operand1, Operand2, Result, Variables, PostingConstrainsTime, LabelingTime) :-

    % ------------ declarar variáveis e domínios ---------------------------
    domain(Variables, 0, 9),

    length(Operand1, LOperand1),
    Operand1Max is 10^LOperand1,
    Operand1Min is 10^(LOperand1-1),
    domain([Operand1Result], Operand1Min , Operand1Max),

    length(Operand2, LOperand2),
    Operand2Max is 10^LOperand2,
    Operand2Min is 10^(LOperand2-1),
    domain([Operand2Result], Operand2Min , Operand2Max),

    length(Result, LResult),
    ResultMax is 10^LResult,
    ResultMin is 10^(LResult-1),
    domain([ResultScalar], ResultMin , ResultMax),
```

Fig. 2. Declaração de variáveis e respetivos domínios

Exemplificando, um puzzle com [B,G] como primeiro operando, [B,R] como segundo operando e [G,B,R] como resultado e [R,G,B] como variáveis.

```
Operand1 = [B,G],
Operand2 = [B,R],
Result = [G,B,R],
Variables = [R,G,B].
```

Fig. 3. Representação em Prolog de puzzle da Fig. 1 por resolver

3.2 Restrições

Para a resolução deste problema, apenas foram utilizadas restrições rígidas, de forma a que a resposta resultante fosse, efetivamente, a resolução do puzzle. Numa abordagem de tradução das restrições dos puzzles para Prolog, calculamos os valores máximos de cada operando e do resultado, conforme o seu número de dígitos, o resultado positivo e restringimos a multiplicação.

Abordagem de tradução direta. Como explicado em cima, traduzimos as restrições do puzzle, criamos também multiplicadores baseados no tamanho dos operandos (e.g. Operando = RGB, Multiplicadores = [100,10,1]). Calculando assim o produto escalar de forma a obter cada um dos valores para os operandos e resultado.

Por fim, restringimos o valor da multiplicação, obtendo assim a solução.

```
% ------------ restrições ----------------------------------------------

all_distinct(Variables),

generate_multipliers(LOperand1, MultipliersOp1),!,
scalar_product(MultipliersOp1, Operand1, #=, Operand1Result),

!,generate_multipliers(LOperand2, MultipliersOp2),!,
scalar_product(MultipliersOp2, Operand2, #=, Operand2Result),

!,generate_multipliers(LResult, MultipliersResult),!,
scalar_product(MultipliersResult, Result, #=, ResultScalar),!,

Operand1Result * Operand2Result #= ResultScalar,
```

Fig. 4. Implementação em Prolog da abordagem anterior

4 Visualização da Solução

Para mostrar o resultado da resolução do puzzle ao utilizador, utilizamos o predicado *printResult*/5:

```
% Imprime no ecrã a solução do puzzle
printResult(Operand1, Operand2, Result, PostingConstrainsTime, LabelingTime) :-
    nl,nl,
    write('--> Result:'),nl,nl,
    printNumberList(Operand1),write(' x '),
    printNumberList(Operand2),write(' = '),
    printNumberList(Result),nl,nl,nl,nl,
    write('--> Statistics:'), nl,nl,
    print_time('Posting Constraints: ',PostingConstrainsTime),
    print_time('Labeling Time: ',LabelingTime),nl,nl,nl,nl,
    pressEnterToContinue,
    play.
```

Fig. 5. Implementação em Prolog da visualização de resultados

Este predicado recebe os operandos e o resultado da multiplicação em forma de listas, que contêm os dígitos que foram calculados.

Para mostrar cada lista de dígitos como um número, fazemos uso do predicado *printNumberList*/2:

```
% Imprime uma lista de inteiros como um número inteiro
printNumberList(List):-
    length(List,Len),
    printNumberList(List,Len).
printNumberList([],0).
printNumberList([H|T],Len):-
    Len > 0,
    Len1 is Len - 1,
    write(H),
    printNumberList(T,Len1).
```

Fig. 6. Implementação em Prolog de predicado auxiliar para visualização de resultados

Além da solução do problema, também são mostradas algumas estatísticas sobre a geração da solução, nomeadamente o tempo de imposição de restrições e de pesquisa de solução, o número de *backtracks* e o número de restrições impostas:

Fig. 7. Visualização da Solução apresentada na Fig. 1

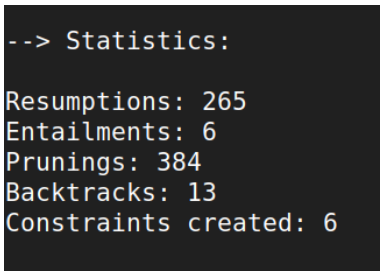

Fig. 8. Visualização das estatísticas para o problema da Fig. 1

5 Geração

Para ser possível ao utilizador inserir novos puzzles, este tem de os inserir num formato específico: uma lista que contém 4 listas compostas por variáveis: as duas primeiras listas contêm os operandos, a terceira contém o resultado e a última as variáveis utilizadas.

Fig. 9. Exemplo de puzzle

Para facilitar a criação de novos puzzles, desenvolvemos também um *script* em python, facultado nos anexos, que dado dois operandos, nos dá uma lista no formato especificado para o input do puzzle no programa Prolog. Desta forma, é facilitada a criação de novos puzzles com dimensões elevadas, não tendo o utilizador que criar os puzzles manualmente.

Fig. 10. Exemplo de Execução do Script de Geração

6 Resultados

6.1 Análise Dimensional

Devido à natureza do nosso problema, o tempo de execução do nosso solver praticamente não varia com o tamanho dos inputs. No entanto, é de notar que, tendo a biblioteca clpfd um limite no domínio, não podemos testar o nosso *solver* para inputs acima desse domínio, o que nos limitou no estudo das estatísticas da resolução do puzzle.

Fig. 11. Solução de Puzzle com 3 variáveis

```
--> Statistics:

Resumptions: 61
Entailments: 10
Prunings: 81
Backtracks: 2
Constraints created: 6
```

Fig. 12. Estatísticas referentes à solução do Puzzle da fig. 11

Fig. 13. Solução de Puzzle com 10 variáveis

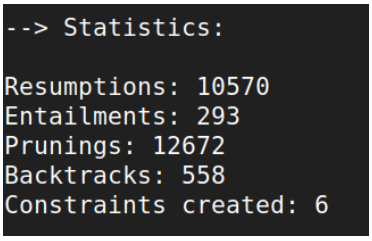

Fig. 14. Estatísticas referentes à solução do Puzzle da fig. 13

6.2 Estratégias de Pesquisa

No contexto do nosso problema, consideramos que a melhor estratégia de pesquisa era a *default*, portanto utilizamos o predicado *labeling* com a lista de opções vazia (‘*labeling*([],*Variables’*). Isto equivale a ter uma lista de opções com os elementos:

- *leftmost* – é selecionada a variável mais à esquerda
- *step* - escolha binária entre X #= B e X #\= B, onde B é a *lower* ou *upper bound* de X
- *up* – o domínio é explorado de forma ascendente
- *all* – todas as soluções são enumeradas

7 Conclusões e Trabalho Futuro

Como futuros aperfeiçoamentos ao projeto poderíamos criar um gerador de puzzles aleatórios com algumas modificações ao script de python e encontrar uma solução para a limitação da biblioteca clpfd quanto aos limites de domínio.

Analisando os resultados obtidos, podemos concluir que a biblioteca clpfd é extremamente eficiente na resolução deste tipo de problemas, não sendo detetada qualquer diferença de tempo de execução entre um puzzle simples e um puzzle mais complexo, sendo ambas quase instantâneas, o que no paradigma das linguagens de programação imperativa seria impossível.

Referências

1. Erich's Page, https://erich-friedman.github.io/. Acedido pela última vez a 2 Jan 2021.
2. Crypto Product, https://erich-friedman.github.io/puzzle/crypto/. Acedido pela última vez a 2 Jan 2021.
3. Arithmetic Constraints, https://sicstus.sics.se/sicstus/docs/latest4/html/sicstus.html/Arithmetic-Constraints.html#Arithmetic-Constraints. Acedido pela última vez a 2 Jan 2021.