Resolução de Problema de Decisão

Trid

Ricardo Miguel de Carvalho e Tiago Lascasas dos Santos

Faculdade de Engenharia da Universidade do Porto
FEUP-PLOG, Turma 3MIEIC03, Grupo Trid_1

Resumo: este artigo pretende ilustrar a resolução de puzzles Trid usando programação em lógica com restrições, sendo também contemplada a geração de puzzles com tamanhos variáveis. Para tal, o problema foi modelado em Prolog e foram aplicadas as restrições necessárias por forma a que todas as regras fossem cumpridas e, portanto, o puzzle pudesse ser corretamente resolvido, tendo também sido criado um gerador de puzzles nesta linguagem. O principal objetivo deste projeto, maximizar a eficiência do algoritmo e torná-lo escalável para qualquer dimensão do puzzle, foi então atingido com sucesso, tendo sido feitos estudos sobre a influência de certos factores sobre a eficiência da solução. Destes estudos concluiu-se que é possível obter soluções em tempo útil para dimensões para puzzles de tamanho na ordem dos 90, que o número de somas num puzzle Trid não têm influência significativa e que a opção de entiquetação ffc foi determinante na implementação eficiente desta solução.

Palavras-Chave: Prolog, Programação em lógica, Programação com restrições, Trid, Resolução de Problemas de Decisão.

1 Introdução

O presente artigo tem como objetivo a análise de um algoritmo recorrendo a lógica com restrições, de forma a resolver o puzzle Trid cujas regras são descritas na próxima secção. Neste artigo será também descrita a abordagem ao problema que permitiu alcançar o algoritmo de resolução e também o algoritmo de geração de puzzles. Posteriormente, será também descrito o processo seguido para a representação textual do puzzle resolvido e é também analisada a complexidade temporal do algoritmo calculada empiricamente. Por fim serão expostas as conclusões retiradas aquando da conclusão do trabalho e em anexo é disponibilizado o código fonte do projeto.

2 Descrição do Problema

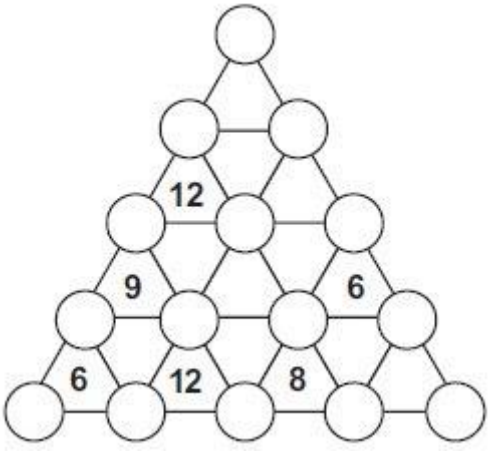

Fig. 1. Exemplo de um jogo de Trid de tamanho 5 em fase inicial.

Trid é um puzzle de lógica de forma triangular (**Fig. 1**). O objetivo do jogo é atribuir a todos os vértices (representados na imagem por círculos) valores que respeitem as seguintes regras:

- Os valores têm de variar entre 0 e o tamanho do lado do triângulo;
- Nos casos em que um triãngulo interno tem um valor especificado, a soma dos três vértices desse triângulo tem de ser igual a esse valor;
- Não pode haver valores nos vértices repetidos em qualquer linha reta.

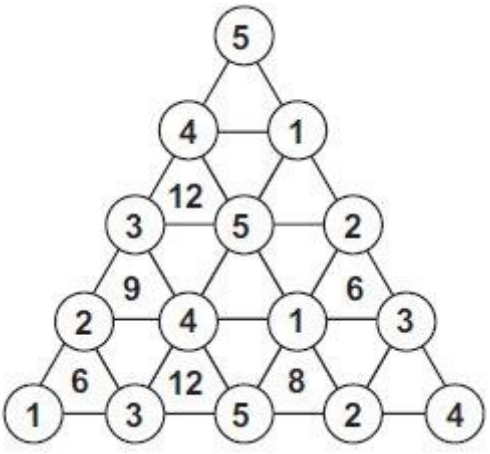

Fig. 2. Exemplo da **Fig. 1** resolvido.

3 Abordagem

3.1 Variáveis de Decisão

Na modelação do problema implementada foram considerados o valor de cada vértice como sendo as variáveis de decisão, sendo que cada variável representa o valor de um vértice. Estas variáveis estão dispostas numa lista de listas em que cada sublista representa um nível horizontal do triângulo (lista **V**). Esta modelação permite saber qual é o vértice a que cada variável se refere: o segundo vértice do terceiro nível será o segundo elemento da terceira sublista, por exemplo. A modelação do exemplo da **Fig. 1** segundo esta convenção é a seguinte:

V = [[V1], [V2, V3], [V4, V5, V6], [V7, V8, V9, V10], [V11, V12, V13, V14, V15]]

As somas dos triângulos internos são dadas também por uma lista de listas, em que cada sublista tem quatro elementos: os conteúdos dos três vértices e a soma, por esta ordem. Esta lista foi intitulada de lista **T**. A modelação das somas internas do exemplo da **Fig. 1** segundo esta convenção é a seguinte:

T = [[V2, V4, V5, 12], [V4, V7, V8, 9], [V7, V11, V12, 6], [V7, V11, V12, 6], [V8, V12, V13, 12], [V9, V13, V14, 8]]

Tal como mencionado na descrição do problema, cada vértice pode tomar um valor entre 1 e o número de linhas horizontais do triângulo, valor este que pode ser facilmente obtido em run-time calculando o tamanho da lista **V**. Portanto, o domínio de cada variável é [1, length(**V**)]. No contexto da implementação da solução este domínio é estabelecido pelo predicado **setDomain/2**.

3.2 Restrições

Na implementação da solução foram consideradas quatro tipos de restrições principais, cada uma implementada por um predicado específico. Três destas restrições prendem-se com o facto de que cada linha tem de ter valores diferentes, havendo um predicado para cada tipo de linha (horizontais ou diagonais em ambos os sentidos), sendo que o quarto envolve as somas de triângulos internos:

1. **horizontal/1** – este predicado implementa a restrição de que cada linha horizontal deve ter valores distintos: para cada sublista de **V**, ou seja, para cada linha horizontal, é aplicado o predicado **all_distinct/1** da biblioteca CLPFD;

2. **diagonalsRight/1** – este predicado implementa a restrição de que cada linha diagonal "\" deve ter valores distintos. De modo a implementar essa restrição, o predicado realiza um varrimento do triângulo da direita para a esquerda, aglomerando as variáveis que correspondem a uma diagonal a cada iteração e aplicando o predicado **all_distinct/1** a cada uma dessas diagonais. Como tal, a primeira diagonal a ser considerada tem um tamanho igual à altura do triângulo (com os vértice mais superior do triângulo e o vértice inferior direito como extremos),

enquanto que a última possui apenas um único elemento (o vértice do canto inferior esquerdo).

3. **diagonalsLeft/1** - este predicado implementa a restrição de que cada linha diagonal "/" deve ter valores distintos. Adota uma estratégia em tudo semelhante à do predicado anterior, sendo que realiza um varrimento da esquerda para a direita em vez do contrário.

4. **innerSums/1** – este predicado implementa a restrição de que caso o valor da soma de um triângulo interno exista, então a soma dos seus vértices tem de ser igual a essa soma. Esta restrição é estabelecida percorrendo a lista **T** e, para cada sublista [T1, T2, T3, Sum] nessa lista aplicar a seguinte restrição:

$$T1 + T2 + T3 \#= Sum.$$

3.3 Estratégia de Pesquisa

A etiquetação das variáveis é feita usando o predicado **labeling/2** da biblioteca CLPFD mas, dado que este predicado necessita de receber uma lista simples de variáveis, é primeiro necessário converter a lista **V** (que é uma lista de listas) numa lista uniforme, o que é feito usado o predicado **flatten/2**. A nível das opções de entiquetação foi utilizada a opção **ffc**, cuja vantagem se encontra posteriormente demonstrada. O tempo da etiquetação é medido usando os predicado **statistics/1** da biblioteca CLPFD.

3.4 Geração de puzzles de dimensão variável

Foi implementado um gerador de puzzles Trid, que cria uma lista **V** correspondente a um triângulo gerado com o número de níveis especificado, e uma lista **T** com um número de somas de triângulos internos também especificados. As somas internas geradas são aleatórias, podendo gerar puzzles sem solução. Contudo, o valor destas somas é realista, sendo que o seu valor máximo é dado pela seguinte fórmula:

$$\mathbf{Max} = 3 * \text{length}(\mathbf{V}) - 3$$

O valor máximo corresponde, então, à situação em que um dos vértices tem o valor length(**V**), outro vértice o valor length(**V**) – 1 e o outro o valor length(**V**) – 2. Quaisquer valores superiores a estes dois últimos em cada vértice estariam a induzir valores iguais numa mesma linha, o que não pode acontecer. O valor mínimo corresponde, da mesma forma, à soma em que os vértices têm os três menores valores possíveis (1, 2 e 3), ou seja, 6. Não há repetição de triângulos, como seria de esperar.

4 Visualização da Solução

Para a visualização da solução foram criados os predicados **printTrid/1**, **printTrid/2**, **printLine/1** e **printIntrLine/1**.

Para visualizar a solução é chamado o predicado **printTrid/1**, cujo argumento é uma lista de listas com os valores dos vértices, em que cada lista representa uma linha a partir do topo e cujos elementos são valores da esquerda para a direita, dentro da mesma linha. Este predicado calcula o tamanho do triângulo (número de vértices na maior linha) que é equivalente o número de linhas com vértices, o comprimento da lista de vértices e que é enviado decrementado em uma unidade juntamente com os vértices ao predicado **printTrid/2**. Este valor é posteriormente decrementado em uma unidade a cada linha representada e é usado para calcular o número necessário de espaço antes de cada linha para que os triângulos interiores fiquem alinhados com os vértices correspondentes, sendo que esse valor é dado pela expressão:

$$R \times 3 \tag{1}$$

Em que R representa o número de linhas entre a linha atual e o fundo do triângulo. De seguida é chamado o predicado **printLine/1** e o predicado **printIntrLine/1**. O caso base identificado foi a linha inferior do triângulo caso em que não é chamado o predicado **printIntrLine/1** por já não haver mais linhas, deixa de ser necessária uma linha intermédia.

O predicado **printLine/1** imprime recursivamente os valores dos vértices de uma linha com dois algarismos (adiciona um zero nas dezenas caso necessário) separados entre si por quatro hífens. Para estre predicado foi identificado como caso base o ultimo elemento da linha ao qual não são adicionados os hífens por não ter elemento seguinte. O predicado **printIntrLine/1** imprime no ecrã a linha intermédia que contém os caracteres ‘/’ e ‘\’ de forma a complementar o desenho dos triângulos internos. De forma análoga à impressão de uma linha, a impressão de uma linha intermédia tem como caso base o último elemento de forma a não imprimir espaços no fim da linha.

Foram também criados os predicados auxiliares **numberLength/2** que calcula o número de dígitos de um número dado, **printZeros/1**, **printSpaces/1** e **printHifen/1** que imprimem no ecrã zeros, espaços e hífens, respetivamente, tantas vezes quantas indicadas em argumento.

No caso de um puzzle ser gerado em vez de ser fornecido como input, é tamném realizada a escrita no ecrã das várias somas de triângulos internos geradas. Esta escrita é feita pelo predicado **printSums/2**, que recebe uma lista com identificadores dos vários vértices de cada soma e as próprias somas em si, imprimindo uma soma por linha. Estes identificadores têm o formato **line-pos**, em que **line** é o número da linha (a começar em 0) e **pos** é a posição dentro dessa linha (a começar em 1).

```
| ?- trid(10, 5).

Generated inner sums:
Vertex 4-2 + vertex 5-2 + vertex 5-3 = 25
Vertex 7-2 + vertex 8-2 + vertex 8-3 = 24
Vertex 6-5 + vertex 7-5 + vertex 7-6 = 22
Vertex 8-1 + vertex 9-1 + vertex 9-2 = 17
Vertex 8-5 + vertex 9-5 + vertex 9-6 = 24

Time: 0.0s
```

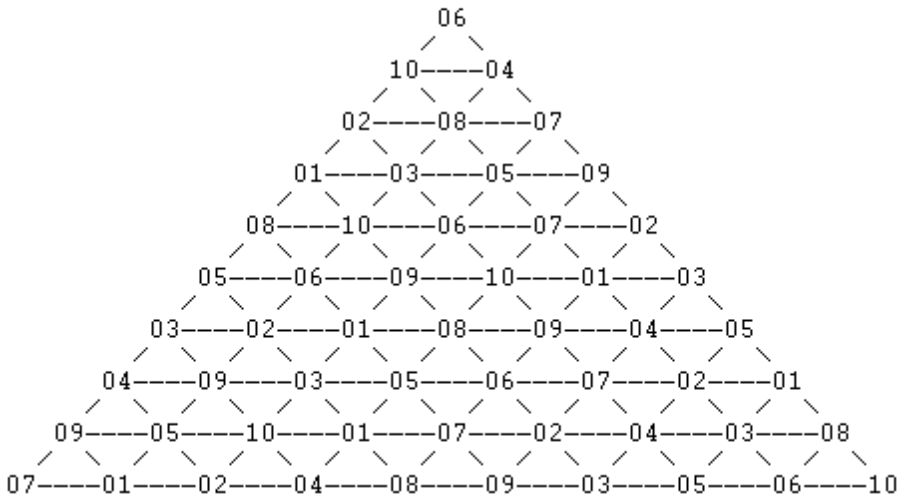

Fig. 3. Exemplo da visualização de um triângulo Trid gerado com 10 níveis e 5 somas.

5 Resultados

De modo a estudar a complexidade temporal da resolução do problema usando triângulos de diferentes tamanhos foram realizadas várias medições do tempo de etiquetação fazendo variar o tamanho da pirâmide (**V**) e o número de somas de triângulos internos (**T**), sendo que se usou o gerador acima especificado de modo a gerar estes puzzles. Os resultados obtidos encontram-se na **Table 1**.

Foi também realizado um estudo de eficiência da opção **ffc** utilizada em relação a uma etiquetação sem esta opção, e cujos resultados se encontram na **Table 2**.

Table 1. Tempo de entiquetação, em segundos, para diferentes dimensões de **V** e de **T**.

	V									
T		10	20	30	40	50	60	70	80	90
	0	0.0	0.01	0.07	0.23	0.62	1.55	3.21	6.04	11.29
	10	0.0	0.01	0.07	0.23	0.69	1.51	3.45	5.97	11.4
	20		0.01	0.06	0.25	0.66	1.58	3.51	6.19	11.88
	30		0.01	0.06	0.22	0.63	1.54	3.27	6.22	11.98
	40			0.06	0.22	0.66	1.51	3.45	6.33	12.31
	50				0.23	0.64	1.56	3.07	6.91	12.22

Table 2. Comparação dos tempos de entiquetação com e sem a opção **ffc**, em segundos (? = sem solução em tempo útil).

V	**T**	Tempo com **ffc**	Tempo sem **ffc**
12	0	0.0	156.4
	5	0.0	25.94
	10	0.0	0.05
13	0	0.0	?
	5	0.0	?
	10	0.0	?
14	0	0.0	?
	5	0.0	?
	10	0.0	?

6 Conclusões

Com base nas medições feitas foi possível inferir duas conclusões principais: a primeira prende-se com a evolução do tempo de etiquetação para valores variáveis de **V** e de **T**, e a segunda com a influência da opção **fcc** nos tempos de entiquetação.

Em relação ao primeiro ponto é de notar que só se começa a sentir uma diferenças significativa no tempo de etiquetação para valores de **V** superiores a 50, sendo que valores inferiores produzem resultados inferiores a 1 segundo, e mesmo para o valor mais alto testado, 90, a solução foi calculada em tempo útil, o que demonstra que a solução encontrada é eficiente para puzzles de dimensão muito elevada. Para valores na ordem dos 100 não foi possível calcular valores, visto que um erro de falta de memória é gerado antes de ser encontrada uma solução. Quanto ao número de somas **T**

podemos concluir que estas podem ter um efeito positivo ou negativo no tempo de etiquetação. Este efeito, contudo, é negligenciável no contexto do problema, visto estes incrementos ou decrementos serem insignificantes, não chegando sequer a 1 segundo no caso mais extremo encontrado. É também de notar que, dada a aleatoriedade das somas geradas, para uma mesma dimensão de **V** e **T** o valor do tempo pode ser diferente, visto que, dependendo da circunstância, as somas geradas podem ser mais ou menos benéficas para a obtenção da solução.

Em relação ao segundo ponto, é muito evidente de que a opção de entiquetação **ffc** influencia de forma drástica os tempos de entiquetação: sem esta opção só foi possível obter soluções em tempo útil para dimesões de **V** iguais ou inferiores a 12, sendo que com a opção **ffc** foi possível obter soluções até valores na ordem dos 90, já para não falar de que os resultados até 12 são instantâneos, ao contrário dos tempos bastante grandes obtidos sem **ffc**. È também de notar que o número de somas teve um efeito na eficiência da versão sem **ffc**, onde um maior número de somas leva a tempos menores, ao contrário da versão com **ffc**, em que, tal como mencionado no ponto anterior, é irrelevante.

7 Bibliografia

1. The Logical World of Puzzles, http://rohanrao.blogspot.pt/2009/05/rules-of-trid.html, acedido pela ultima vez em 2017/12/22
2. www.sachsentext.de, http://www.sachsentext.de/en/taxonomy/term/403, acedido pela ultima vez em 2017/12/22

8 Anexos

8.1 Código fonte

trid.pl

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

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Trid Entrypoints
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Generates a Trid board of size N with NT triangle sums, solves it and prints the solution
trid(N, NT) :-
        integer(N),
        integer(NT),
        nl, write('Generated inner sums:'), nl,
                generateTrid(N, NT, V, T),
                solveTrid(V, T),
                printTrid(V).

%Solves a trid board V with inner sums T and prints the solution
trid(V, T) :-
        solveTrid(V, T),
        printTrid(V).

%Tests the solver with a predetermined board.
%Solution in http://rohanrao.blogspot.pt/2009/05/rules-of-trid.html
testSolver :-
        V = [[V1],[V2, V3],[V4, V5, V6],[V7, V8, V9, V10],[V11, V12, V13, V14, V15]],
        T = [[V2, V4, V5, 12],[V4, V7, V8, 9],[V7, V11, V12, 6],[V7, V11, V12, 6],[V8, V12, V13, 12],[V9, V13, V14, 8]],
        trid(V, T).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Trid Solver
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Trid solver:
% V - list of lists with the vertices
% T - list of liss with the triangles and inner sums
solveTrid(V, T) :-
        length(V, Range),
                setDomain(V, Range),
                horizontal(V),
        diagonalsRight(V),
                diagonalsLeft(V),
                innerSums(T),
                flatten(V, Vf),
        reset_timer,
                labeling([ffc], Vf),
        print_time.

%Sets the domain of all vertices between 1 and the specified value R
setDomain([], _).
setDomain([L|Ls], R) :-
                domain(L, 1, R),
                setDomain(Ls, R).

%Sets each horizontal line of the board to have distinct elements within that line
horizontal([]).
horizontal([L|Lr]) :-
                all_distinct(L),
                horizontal(Lr).

%Sets each "\" diagonal of the board to have distinct elements within that diagonal
```

```
62 diagonalsRight(V) :-
63         length(V, N),
64         diagonalsRight(V, N).
65 diagonalsRight(_, 0).
66 diagonalsRight([V|Vx], N) :-
67         restrictRight([V|Vx], N),
68         M is N - 1,
69         diagonalsRight(Vx, M).
70
71 restrictRight(V, N) :- restrictRight(V, N, [], 0).
72 restrictRight([], _, Diagonal, _) :- all_distinct(Diagonal).
73 restrictRight([X|Xs], N, Acc, R) :-
74         element(Rn, X, Elem),
75         append(Acc, [Elem], App),
76         Rn is R + 1,
77         restrictRight(Xs, N, App, Rn).
78
79 %Sets each "/" diagonal of the board to have distinct elements within that diagonal
80 diagonalsLeft(V) :-
81         length(V, N),
82         diagonalsLeft(V, N).
83 diagonalsLeft(_, 0).
84 diagonalsLeft(V, N) :-
85         restrictLeft(V, N),
86         M is N - 1,
87         diagonalsLeft(V, M).
88
89 restrictLeft(V, N) :- restrictLeft(V, N, []).
90 restrictLeft([], _, Diagonal) :- all_distinct(Diagonal).
91 restrictLeft([X|Xs], N, Acc) :-
92         element(N, X, Elem),
93         append(Acc, [Elem], App),
94         restrictLeft(Xs, N, App).
95 restrictLeft([_|Xs], N, Acc) :-
96         restrictLeft(Xs, N, Acc).
97
98 %imposes the restrictions regarding the sum of three vertices
99 innerSums([]).
100 innerSums([X|Xs]) :-
101         innerSum(X),
102         innerSums(Xs).
103 innerSum([T1, T2, T3, Sum]) :-
104         T1 + T2 + T3 #= Sum.
105
106 %flattens the list of lists of vertices into a list of vertices
107 flatten(List, Flat) :- flatten(List, Flat, []).
108 flatten([], Flat, Flat).
109 flatten([X|Xs], Flat, Acc) :-
110                 append(Acc, X, App),
111                 flatten(Xs, Flat, App).
112
113 %predicates to calculate the labeling time
114 reset_timer :- statistics(walltime, _).
115 print_time :-
116         statistics(walltime,[_,T]),
117         TS is ((T//10)*10)/1000,
118         nl, write('Time: '), write(TS), write('s'), nl, nl.
119
120 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
121 % Trid Generator
122 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
123
```

trid.pl

```
%Generates a Trid board
% N - the number of rows
% NT - the number of inner triangle sums
% V - the generated board
% T - the generated inner sums
generateTrid(N, NT, V, T) :-
        N > 0,
        createBoard(N, V),
        createInnerSums(NT, V, T).

%creates a board of the specified dimension
createBoard(N, V) :- createBoard(N, V, [], 0).
createBoard(N, V, V, N).
createBoard(N, V, Acc, Index) :-
        I is Index + 1,
        length(Row, I),
        append(Acc, [Row], App),
        createBoard(N, V, App, I).

%creates inner triangle sums
createInnerSums(NT, V, T) :- createInnerSums(NT, V, T, [], 0, []).
createInnerSums(NT, _, T, T, NT, Tuples) :- writeSums(T, Tuples).
createInnerSums(NT, V, T, Acc, Count, AccTuple) :-
        generateInnerSum(V, AccTuple, Sum, Tuple),
        append(Acc, [Sum], App),
        append(AccTuple, [Tuple], AppTuple),
        CountInc is Count + 1,
        createInnerSums(NT, V, T, App, CountInc, AppTuple).

generateInnerSum(V, ExistingSums, Sum, Tuple) :-
        makeSum(V, Sum, Tuple),
        \+ member(Tuple, ExistingSums).
generateInnerSum(V, ExistingSums, Sum, Tuple) :-
        generateInnerSum(V, ExistingSums, Sum, Tuple).

makeSum(V, [TopVertex, LeftVertex, RightVertex, SumValue],
   TopRow-TopVertexPos-BottomRow-LeftVertexPos-BottomRow-RightVertexPos) :-
        length(V, Size),
        Limit is Size - 1,
        random(1, Limit, TopRow),
        BottomRow is TopRow + 1,
        random(1, TopRow, TopVertexPos),
        LeftVertexPos is TopVertexPos,
        RightVertexPos is LeftVertexPos + 1,
        getVertex(V, TopRow, TopVertexPos, TopVertex),
        getVertex(V, BottomRow, LeftVertexPos, LeftVertex),
        getVertex(V, BottomRow, RightVertexPos, RightVertex),
        SumLimit is Size * 3,
        random(6, SumLimit, SumValue).

getVertex(V, Row, Pos, Vertex) :- getVertex(V, Row, Pos, Vertex, 0).
getVertex([V|Vx], Row, Pos, Vertex, Row) :-
        nth1(Pos, V, Vertex).
getVertex([], _, _, _, _).
getVertex([_|Vx], Row, Pos, Vertex, Index) :-
        IndexInc is Index + 1,
        getVertex(Vx, Row, Pos, Vertex, IndexInc).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Trid Displayer
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

trid.pl

```
writeSums([], []).
writeSums([[_, _, _, SumValue] | Ts],
   [TopRow-TopVertexPos-BottomRow-LeftVertexPos-BottomRow-RightVertexPos | Tts]) :-
    write('Vertex '), write(TopRow-TopVertexPos), write(' + '),
    write('vertex '), write(BottomRow-LeftVertexPos), write(' + '),
    write('vertex '), write(BottomRow-RightVertexPos), write(' = '),
    write(SumValue), nl,
    writeSums(Ts, Tts).

printTrid(V) :- length(V, R), R1 is R - 1,
                printTrid(V, R1).

printTrid([Line], R):-
                Amount is 3 * R, printSpaces(Amount),
                printLine(Line), nl.

printTrid([Line | Rest], R):-
                Amount is 3 * R, printSpaces(Amount),
                printLine(Line), nl, R1 is R - 1,
                IntrAmount is (R1 * 3) + 2,printSpaces(IntrAmount),
                printIntrLine(Line), nl,
                printTrid(Rest, R1).

printLine([Element]) :- numberLength(Element, ElementLength),
                Amount is 2 - ElementLength,
                printZeros(Amount), print(Element).
printLine([Element|Rest]) :- numberLength(Element, ElementLength),
                Amount is 2 - ElementLength,
                printZeros(Amount), print(Element),
                printHifen(4), printLine(Rest).

printIntrLine([_LineStart]):-write('/  \\').
printIntrLine([_LineStart|Rest]):- write('/  \\  '), printIntrLine(Rest).

%prints the given amount of zeros
printZeros(0).
printZeros(Amount):- write(0), A is Amount - 1, printSpaces(A).

%prints the given amount of spaces
printSpaces(0).
printSpaces(Amount):- write(' '), A is Amount - 1, printSpaces(A).

%prints the given amount of hifens
printHifen(0).
printHifen(Amount):- write('-'), A is Amount - 1, printHifen(A).

%counts the number of digits in a number
numberLength(0, 0).
numberLength(Number, Length):- X is Number//10, numberLength(X, L), Length is L + 1.

```