

Mestrado Integrado em Engenharia Informática e Computação  
Programação em Lógica

Weight

3MIEIC06 - Weight1

Andreia Gouveia [up201706430@fe.up.pt](mailto:up201706430@fe.up.pt)  
João Filipe Carvalho de Araújo [up201705577@fe.up.pt](mailto:up201705577@fe.up.pt)

Janeiro 2019

# Resumo

Este projeto foi desenvolvido no Sistema de Desenvolvimento SICStus Prolog, no âmbito da unidade curricular de Programação em Lógica. O objectivo deste projecto é obter uma forma genérica de resolver um problema de decisão/otimização, utilizando restrições. O problema escolhido foi o puzzle Weight. Através da linguagem de Prolog, conseguimos chegar a uma resolução deste problema, que vai ser detalhadamente abordada neste artigo.

## Índice

<b>Resumo</b>	<b>2</b>
<b>Índice</b>	<b>2</b>
<b>1. Introdução</b>	<b>2</b>
<b>2. Descrição do Problema</b>	<b>3</b>
<b>3. Abordagem</b>	<b>3</b>
3.1 Variáveis de Decisão	4
3.2 Restrições	5
3.3 Função de Avaliação	5
3.4 Estratégia de Pesquisa	5
<b>4. Visualização da Solução</b>	<b>5</b>
<b>5. Resultados</b>	<b>8</b>
<b>6. Conclusões e Trabalho Futuro</b>	<b>8</b>
<b>Bibliografia</b>	<b>9</b>
<b>Anexo</b>	<b>9</b>

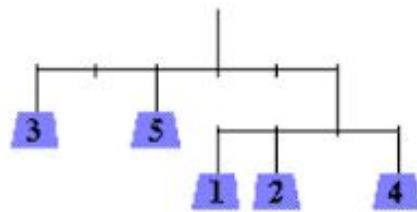
## 1. Introdução

Este trabalho foi desenvolvido no âmbito da cadeira curricular Programação em Lógica. O objetivo do mesmo focou-se na construção de um programa em Programação em Lógica com Restrições para a resolução de um dos problemas de otimização ou decisão combinatória. O grupo optou por um problema de decisão, o puzzle Weight.

Neste artigo pretende-se expor o problema proposto, tal como a resolução encontrada pelo grupo.

## 2. Descrição do Problema

O problema escolhido foi o *Weigth*. Este é um puzzle constituído por pesos. Cada peso possui massa de valor inteiro de 1 a N, sendo N o número total de pesos. Em cada caso, as linhas horizontais têm de ter o peso distribuído de modo a que, a parte direita tenha o mesmo peso (sendo o valor total de cada peso, a multiplicação do próprio peso com a distância a que se encontra do centro da “árvore”) que a parte esquerda (por ex:  $3*3 + 1*5 = 2*(1 + 2 + 4)$  e  $2*1 + 1*2 = 1*4$ ), como é possível verificar na seguinte figura:



É de notar que os valores dos pesos têm de ser únicos, ou seja, não existem 2 pesos com o mesmo valor.

Em suma, este projecto pretende encontrar uma forma genérica de resolver qualquer um destes puzzles de modo a cumprir com as regras do mesmo.

## 3. Abordagem

Na resolução do problema proposto, foi criada uma lista para representar o puzzle. Decidimos integrar nessa lista, os pesos, sendo o seu valor a respectiva distância ao centro da árvore, como também, no início de cada sublista/subárvore, a posição da mesma e o número de pesos que contém.

Quanto à distância, para distinguir se o peso se encontra do lado esquerdo ou direito da árvore, esta tem valor negativo se se encontrar à esquerda do centro da árvore (por ex: -1) e positivo se se encontrar à direita da mesma.

Na restante sublista, é representada a mesma informação que a da lista inicial (os pesos e a sua respectiva distância ao centro dessa árvore).

```
[-3, -1, [2, 3, -2, -1, 1]]
```

Neste projeto, a função responsável pela resolução do puzzle denomina-se de *sumWeights(Distances, Weights, Acumulator)*. Esta recebe a lista de distâncias (ou seja, a representação interna do puzzle escolhido), a lista *Weights* (mencionada na seção 3.1) e finalmente um acumulador (o valor inicial é 0).

```

sumWeights([],[],0).

sumWeights([H1|Distances],[H2|Weights],Acumulator):-%if H1 is not list
    \+ is_list(H1),
    sumWeights(Distances,Weights,AuxAcumulator),
    Acumulator #= H1 * H2 + AuxAcumulator.
sumWeights([[Position,NumWeights|ListDistAux]|Distances],Weights,Acumulator):-
%if first element of Distances is list
    first_n(NumWeights,Weights,FirstWeights),
    sumWeights(ListDistAux,FirstWeights,0),
    append(FirstWeights, RestWeights, Weights),
    multiply(Position,FirstWeights,Total),
    sumWeights(Distances,RestWeights,AuxAcumulator),
    Acumulator #= Total + AuxAcumulator.

```

Esta função vai ver se o primeiro elemento é uma lista.

- Caso não seja uma lista:

Volta a chamar `sumWeights` com os termos seguintes e depois guarda o valor da soma do acumulador desse `sumWeight` com o peso \* distância (o primeiro elemento de cada uma das listas) ao acumulador.

- Caso seja uma lista:

Ao ler a lista, vai receber a sua posição relativamente ao ramo que a originou (*Position*) e o número de pesos contidos na mesma (*NumWeights*). A seguir vai buscar os primeiros *NumWeights* pesos e chama `sumWeights` com o acumulador a 0 para a subárvore. Depois chama `sumWeights` para o resto da árvore.

No fim soma o acumulador desse último `sumWeights` com a soma de todos os pesos da subárvore multiplicados por *Position*.

## 3.1 Variáveis de Decisão

A solução do problema vem sob a forma de uma lista, onde são colocados os valores dos pesos do puzzle. Esta é representada, no código, pela variável *Weights*.

O tamanho desta variável é do tamanho do número de elementos a descobrir no puzzle, *N*. Os pesos estão ordenados na lista pela ordem em que aparecem no puzzle, da esquerda para a direita.

## 3.2 Restrições

Para resolver este problema, só recorreremos a duas restrições: os pesos têm de ser todos distintos e o seu domínio vai de 1 a *N*, sendo *N* o número de pesos.

Tal restrição foi feita devido a à especificação do jogo, ou seja, o puzzle teria que ser resolvido de modo a que os valores obtidos fossem todos distintos.

### 3.3 Função de Avaliação

Visto que só há uma resolução por puzzle, não foi necessário implementar uma função de avaliação, visto que, caso não existam soluções, o prolog responde com “no”. No caso de existir solução, este dá display da mesma.

### 3.4 Estratégia de Pesquisa

Para verificar a “satisfabilidade” de todas as restrições foi utilizado o predicado **labeling(Options , Variables)**. Mas especificamente **labeling([], Weights)**.

Não foram utilizados predicados de otimização.

## 4. Visualização da Solução

Para visualizar o puzzle escolhido e o resultado, é chamada a função *display(ListaDistancias,ListaPesos,AcumuladorDistancias,AcumuladorPesos)*.

```
printSum([T]):- write(T).
printSum([H|T]):-
    write(H), write(' + '), printSum(T).

display( [[H1,H2|H3]|H] ,P,Aux1,Aux2):-
    \+isEmpty(H),
    first_n(H2, P, AuxList),
    append(AuxList, RestP, P),
    append(Aux2, [AuxList], NewAux2),
    append(Aux1, [H3], NewAux1),
    write(H1),write(' * ('),printSum(AuxList),write(') + '),display(H,RestP,NewAux1,NewAux2).

display( [[H1,H2|H3]|H] ,P,Aux1,Aux2):-
    isEmpty(H),
    first_n(H2, P, AuxList),
    append(AuxList, [], P),
    append(Aux2, [AuxList], NewAux2),
    append(Aux1, [H3], NewAux1),
    write(H1),write(' * ('),printSum(AuxList),write(') = 0'),nl,display2(NewAux1,NewAux2).

display([H1|H2],[P1|P2],Aux,Aux2):-
    \+isEmpty(H2),
    write(H1),write(' * '),write(P1),write(' + '),
    display(H2,P2,Aux,Aux2).

display([H1],[P1],Aux1,Aux2):-
    write(H1),write(' * '),write(P1),write(' = 0'),nl,display2(Aux1,Aux2).

display2([],[]).

display2([Aux1|T],[Aux2|T1]):-
    display(Aux1,Aux2,T,T1).
```

No primeiro caso a função vai primeiramente verificar se não está no último elemento e se o primeiro elemento é uma lista. Depois adiciona aos acumuladores a subárvore e no fim escreve "Distancia \* (", imprimindo os pesos da subárvore a somarem-se e de seguida escreve ") + " e chama de novo o display.

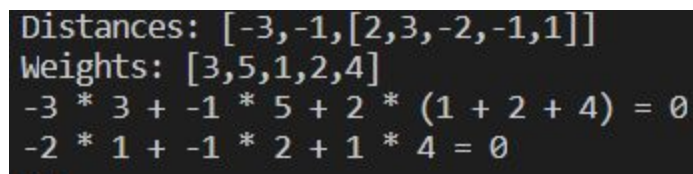
No segundo caso, a função vai verificar se está no último elemento e se o primeiro elemento é uma lista. Caso tal se confirme vai fazer o mesmo que o primeiro caso mas em vez de chamar de novo o display escreve "= 0" e procede a chamar a função display2.

No terceiro caso verifica se não está no último elemento e se o primeiro elemento não é uma lista. Caso se verifique, escreve " Distância \* Peso +" e chama de novo o display.

Finalmente, no quarto caso verifica se está no último elemento e se o primeiro elemento não é uma lista, escreve " Distância \* Peso = 0" e chama o display 2.

Por sua vez, o display2 chama o display para o primeiro elemento dos acumuladores das subárvores.

O output resultante apresenta-se na consola tal como indicado na figura seguinte:



```
Distances: [-3,-1,[2,3,-2,-1,1]]
Weights: [3,5,1,2,4]
-3 * 3 + -1 * 5 + 2 * (1 + 2 + 4) = 0
-2 * 1 + -1 * 2 + 1 * 4 = 0
```

Fizemos também a função weight(N) que testa rapidamente o problema com árvores de distâncias já criadas, sendo N o número de pesos a testar (N=5,6,7,8,9,10,14,19 or 20).

Também criamos um gerador de puzzles generate(X,Y), sendo X o número de pesos e Y o número de subárvores ( $X > Y * 2$  para ter o mínimo de pesos nas subárvores). Este gerador não devolve as distâncias ordenadas e os pesos estão ordenadas pelas distâncias. Segue-se o código responsável por esta funcionalidade:

```
game2(Size,N,Distances,Weights):-
    length(Weights, Size),
    length(Temp, Size),
    domain(Weights,1,Size),
    domain(Temp,-5,5),
    all_distinct(Weights),
    all_distinct(Temp),

    game2Aux(N,Temp,Distances,Weights),

    sumWeights(Distances, Weights, 0),
    append(Temp, Weights, NewList),
```

```

labeling([min], NewList).

game2Aux(0,Temp,Temp,_Weights).
game2Aux(N,Temp,Distances,Weights):-
    N>0,N1 is N-1,
    random_member(Elem, Temp),
    N2 is N*2,
    length(Weights,Size),
    random(N2, Size, X),%number of weights in subtree
    first_n(X,Weights,NewWeights),
    repeat,
    random_subseq(Temp,SubList,Rest),
    length(SubList,X),!,
    game2Aux(N1,SubList,SubDist,NewWeights),
    append([Elem,X], SubDist, SubDistFixed),
    append(Rest, [SubDistFixed], Distances).

generate(X,Y):-
    game2(X,Y,Distances,Weights),
    write('Distances: '),write(Distances),nl,write('Weights: '),write(Weights),
    nl,display(Distances,Weights,[],[]).

```

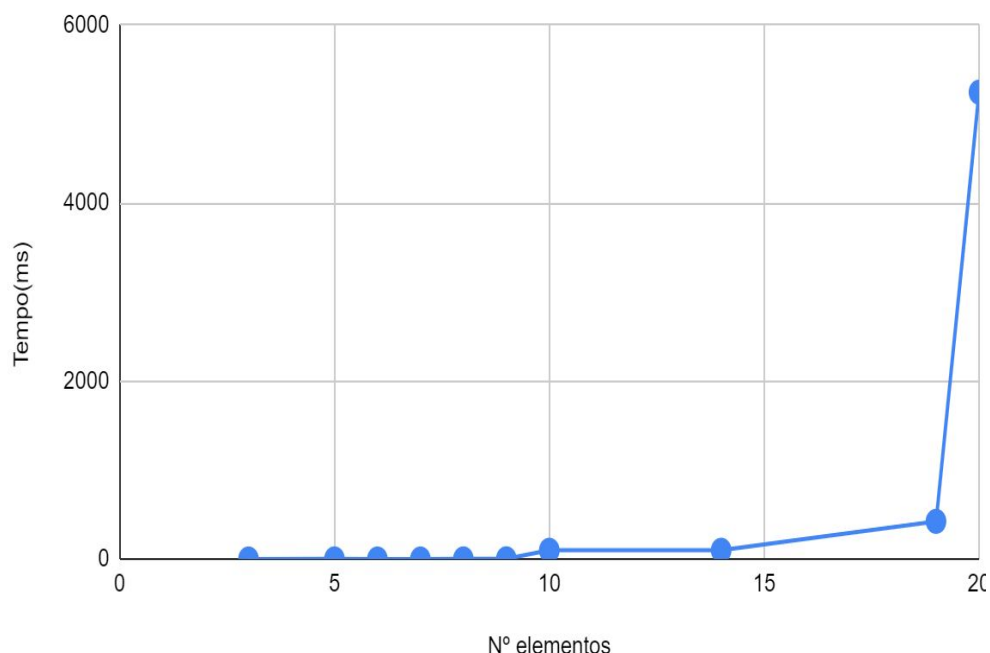
## 5. Resultados

Para retirar resultados desta experiência, foram obtidos os tempos de resolução dos puzzles. Seguem-se os resultados obtidos, com a respectiva análise:

- **Medição do tempo de execução, fazendo variar o número de elementos do puzzle:**

Medimos o tempo de 10 puzzles com diferentes números de elementos, sendo que o número de elementos testado para cada puzzle foi [ 4 , 5 , 6 , 7 , 8 , 9 , 10 , 14 , 19 , 20].

Foi obtido o seguinte gráfico:



Tempo(ms)	Nº de Elementos
0	3
1	5
0	6
0	7
1	8
2	9
98	10
98	14
423	19
5247	20

Com este gráfico podemos verificar que o algoritmo é muito eficiente até aos puzzles com 10 ou mais elementos, visto que o tempo de resolução não ultrapassa 1 ms. Após os 19 elementos, o tempo de resolução aumenta muito mais significativamente.

## 6. Conclusões e Trabalho Futuro

O principal objectivo deste projecto foi consolidar a matéria lecionada nas aulas teóricas e teórico-práticas, em particular o módulo das restrições.

Ao longo do desenvolvimento do projecto, foram encontradas algumas dificuldades, nomeadamente na escolha das restrições a usar e como as implementar. Mas, com ajuda das aulas, conseguimos superar essas dificuldades.

É de notar que existem aspetos que poderiam ser melhorados, como por exemplo, tornar o algoritmo de solução mais eficiente.

No entanto, achamos que conseguimos resolver o problema proposto com sucesso e que o seu desenvolvimento ajudou na consolidação da matéria lecionada.

## Bibliografia

- <https://www2.stetson.edu/~efriedma/puzzle/weight/>

## Anexo



```

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

%for sublist, first element is position, second element is number of weights
dist(3,[-2,-1,7]).%3 elements
dist(5,[-3,-1,[2,3,-2,-1,1]]).%5 elements
dist(6,[-3,-2,[1,3,[-1,2,-3,1],1],2]).%6 elements
dist(7,[-3,4,-3,-2,-1,2],[3,2,-2,1],5]).%7 elements
dist(8,[-1,4,[-1,3,-2,-1,1],2],[1,4,-3,-2,-1,2]).%8 elements
dist(9,[-3,-2,[-1,5,[-2,2,-2,1],[1,2,-1,3],2],2,4]).%9 elements
dist(10,[-1,5,[-1,4,[-1,3,-1,1,2],3],2],[1,3,-2,1,2],[2,2,-2,3])).%10 elements
dist(14,[-2,3,-2,-1,5],[1,11,[-1,9,[-1,7,-2,-1,[1,5,[-1,3,-2,[1,2,-1,2]],1,2]],1,2],1,3])).%14
dist(19,[-2,8,-3,-1,[1,6,[-1,4,-2,-1,[1,2,-1,2]],1,2]],1,[2,10,-3,-1,[1,8,[-1,5,[-3,2,-1,2],-2,-1,1],[1,3,-2,-1,1]]])).%19
dist(20,[-4,3,-3,-2,3],[-2,7,-2,-1,[1,5,[-1,3,-1,2,3],1,2]],1,[3,9,[-1,7,[-1,5,-1,[1,2,-2,1],2,3],1,2],1,2])).%20

weight(X):-dist(X,List),game(X,List,R),
write('Distances: '),write(List),nl,write('Weights:
'),write(R),nl,display(List,R,[],[]). % X=5,6,10,14,19 or 20
game(Size,Distances,Weights):-
    %Variaveis de decisao
    length(Weights, Size),
    domain(Weights,1,Size),
    %Restricoes
    all_distinct(Weights),
    %Funcao de avalicao
    sumWeights(Distances, Weights, 0),
    %Labelling
    labeling([], Weights).

sumWeights([],[],0).

```

```

sumWeights([H1|Distances],[H2|Weights],Acumulator):-%if H1 is not list
    \+ is_list(H1),
    sumWeights(Distances,Weights,AuxAcumulator),
    Acumulator #= H1 * H2 + AuxAcumulator.
sumWeights([[Position,NumWeights|ListDistAux]|Distances],Weights,Acumulator):-%if
first element of Distances is list
    first_n(NumWeights,Weights,FirstWeights),
    sumWeights(ListDistAux,FirstWeights,0),
    append(FirstWeights, RestWeights, Weights),
    multiply(Position,FirstWeights,Total),
    sumWeights(Distances,RestWeights,AuxAcumulator),
    Acumulator #= Total + AuxAcumulator.

%first_n(0,_,[]).
%first_n(N, [H|T], [H|T1]) :- N > 0, N1 is N-1, first_n(N1, T, T1).
first_n(N, List, AuxList) :-
    length(AuxList,N),
    append(AuxList, _, List).

multiply(_,[],0).

multiply(Dist,[H|Weights],Total):-
    multiply(Dist,Weights,AuxTotal),
    Total #= AuxTotal + Dist * H.

%display

isEmpty([]).

printSum([T]):- write(T).
printSum([H|T]):-
    write(H), write(' + '), printSum(T).

```

```

display( [[H1,H2|H3]|H] ,P,Aux1,Aux2):-
    \+isEmpty(H),
    first_n(H2, P, AuxList),
    append(AuxList, RestP, P),
    append(Aux2, [AuxList], NewAux2),
    append(Aux1, [H3], NewAux1),
    write(H1),write(' * '),printSum(AuxList),write(') +
'),display(H,RestP,NewAux1,NewAux2).

display( [[H1,H2|H3]|H] ,P,Aux1,Aux2):-
    isEmpty(H),
    first_n(H2, P, AuxList),
    append(AuxList, [], P),
    append(Aux2, [AuxList], NewAux2),
    append(Aux1, [H3], NewAux1),
    write(H1),write(' * '),printSum(AuxList),write(') =
0'),nl,display2(NewAux1,NewAux2).

display([H1|H2],[P1|P2],Aux,Aux2):-
    \+isEmpty(H2),
    write(H1),write(' * '),write(P1),write(' + '),
    display(H2,P2,Aux,Aux2).

display([H1],[P1],Aux1,Aux2):-
    write(H1),write(' * '),write(P1),write(' = 0'),nl,display2(Aux1,Aux2).

display2([],[]).

display2([Aux1|T],[Aux2|T1]):-
    display(Aux1,Aux2,T,T1).

game2(Size,N,Distances,Weights):-
    Size>N*2,
    length(Weights, Size),
    length(Temp, Size),
    domain(Weights,1,Size),
    domain(Temp,-5,5),

```

```

    all_distinct(Weights),
    all_distinct(Temp),

    game2Aux(N,Temp,Distances,Weights),

    sumWeights(Distances, Weights, 0),
    append(Temp, Weights, NewList),
    labeling([min], NewList).

game2Aux(0,Temp,Temp,_Weights).
game2Aux(N,Temp,Distances,Weights):-
    N>0,N1 is N-1,
    random_member(Elem, Temp),
    N2 is N*2,
    length(Weights,Size),
    random(N2, Size, X),%number of weights in subtree
    first_n(X,Weights,NewWeights),
    repeat,
    random_subseq(Temp,SubList,Rest),
    length(SubList,X),!,
    game2Aux(N1,SubList,SubDist,NewWeights),
    append([Elem,X], SubDist, SubDistFixed),
    append(Rest, [SubDistFixed], Distances).

generate(X,Y):-
    game2(X,Y,Distances,Weights),
    write('Distances: '),write(Distances),nl,write('Weights: '),write(Weights),
    nl,display(Distances,Weights,[],[]).

% time tests

test(X):-
    dist(X,List),game(X,List,R).

getTime(X):-
    statistics(walltime, [TimeSinceStart | [TimeSinceLastCall]]),

```

```
test(X),  
statistics(walltime, [NewTimeSinceStart | [ExecutionTime]]),  
write('Execution took '), write(ExecutionTime), write(' ms. '), nl.
```