**Resolução de Problema de Decisão usando**

**Programação em Lógica com Restrições:**

**Redistribuição de Público**

Bárbara Sofia Silva e Julieta Frade

FEUP-PLOG, Turma 3MIEIC02, Grupo Redistribuição de Publico\_2

Faculdade de Engenharia da Universidade do Porto

Rua Dr. Roberto Frias, 4200-465, Porto, Portugal

**Resumo.** O projeto foi desenvolvido no Sistema de Desenvolvimento SICStus Prolog no âmbito da unidade curricular de Programação em Lógica, cujo objetivo é resolver um problema de decisão/otimização implementando restrições. O problema de otimização escolhido é o de redistribuição de público, este, tem como finalidade obter o menor conjunto de trocas necessárias para que todos os grupos de pessoas fiquem em lugares contíguos num concerto. Assim, através da linguagem de Prolog, foi possível a resolução deste mesmo problema, que será abordada detalhadamente neste artigo.

**Keywords:** Redistribuição de Público, SICStus, Prolog, FEUP

# 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 de Computação. Para tal, foi necessário implementar uma possível resolução para um problema de decisão ou otimização em Prolog, com restrições. O grupo escolheu um problema de otimização, denominado por Redistribuição de Público.

O problema de otimização escolhido consiste no ajuste da distribuição de lugares de um concerto, isto é, visto alguns grupos de pessoas, por exemplo amigos, não terem conseguido ficar juntos, o objetivo deste projeto é elaborar uma resolução que obtenha o menor conjunto de trocas necessárias para que todos estes grupos de pessoas fiquem em lugares contíguos.

Este artigo tem a seguinte estrutura:

* **Descrição do Problema:** descrição com detalhe o problema de otimização ou decisão em análise.
* **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.
  + **Restrições:** descrição das restrições rígidas e flexíveis do problema e a sua implementação utilizando o SICStus Prolog.
  + **Função de Avaliação:** descrição da forma de avaliar a solução obtida e a sua implementação utilizando o SICStus Prolog.
  + **Estratégia de Pesquisa:** descrição da estratégia de etiquetagem (*labeling*) utilizada ou implementada, nomeadamente no que diz respeito à ordenação de variáveis e valores.
* **Visualização da Solução:** explicação dos predicados que permitem visualizar a solução em modo de texto.
* **Resultados:** demonstração de exemplos de aplicação em instâncias do problema com diferentes complexidades e análise dos resultados obtidos.
* **Conclusões e Trabalho Futuro:** conclusões retiradas deste projeto, resultados obtidos, vantagens e limitações da solução proposta, aspetos a melhorar.
* **Bibliografia:** livros, artigos, páginas Web, utilizados para desenvolver o trabalho.
* **Anexo:** código fonte, ficheiros de dados e resultados, entre outros.

# Descrição do Problema

Redistribuição de Público é um problema de otimização. Este problema retrata a situação na qual foram vendidos todos os bilhetes disponíveis para um concerto na Casa da Música, e cada bilhete tem um lugar específico. Infelizmente, alguns grupos de pessoas, por exemplo, amigos ou familiares, não conseguiram lugares contíguos, sujeitando-se a ficarem em lugares dispersos da sala.

Portanto, pretende-se obter o menor conjunto de trocas necessárias de modo a que todos os grupos de pessoas fiquem em lugares contíguos. As mudanças a efetuar devem igualmente ter o menor impacto possível, isto é, as pessoas a mudar devem sê-lo para o lugar mais próximo possível que permita obter uma solução válida.

# Abordagem

Na resolução deste problema na linguagem *Prolog* foi utilizada uma lista para representar a distribuição de público inicial (***InputGroups***), pois para as pessoas do mesmo grupo ficarem em lugares contíguos não é necessário ter em consideração as restantes filas.

**Tabela 1.** Exemplo de Distribuição Inicial

|  |  |  |
| --- | --- | --- |
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |
| 10 | 11 | 12 |

Na posição correspondente ao número do lugar na lista *InputGroups*, está o número do grupo da pessoa inicialmente nesse lugar. Assim, se a Tabela 1 representar o público, os números em cada célula o número de cada lugar e as cores os diferentes grupos, a lista tida em consideração para resolver o problema será: *InputGroups=[1,1,2,2,3,3,3,2,2,1,3,3].*

## Variáveis de Decisão

A solução do problema vem na forma de 2 listas que representam a *distribuição de público final*: uma para representar a distribuição dos grupos (***OutputGroups***) e outra que nos diz qual o lugar inicial da pessoa atualmente naquele lugar (***OutputIndexs***) Assim a solução deste problema tendo em conta a distribuição inicial a cima seria: *OutputGroups=[1,1,1,2,2,2,2,3,3,3,3,3]* e *OutputIndexs=[1,2,10,4,3,8,9,5,6,7,11,12].*

**Tabela 2**. Exemplo de Distribuição Final

|  |  |  |
| --- | --- | --- |
| 1 | 2 | 10 |
| 4 | 3 | 8 |
| 9 | 5 | 6 |
| 7 | 11 | 12 |

Tanto o tamanho da lista *OutputGroups*, como o da lista *OutputIndexs* é igual ao tamanho da lista *InputGroups*. No que toca ao domínio, o da lista *OutputGroups* é de 1 até ao número de grupos e da lista *OutputIndexs* é de 1 até ao número de pessoas.

## Restrições

Os elementos da lista OutputIndexs têm que ser todos distintos**.**Como cada posição da lista *OutputIndexs* é constituída pelo número do lugar inicial da pessoa atualmente nessa posição e os números de lugares são únicos foi chamado o predicado de restrição *all\_distinct(OutputIndexs).*

**Na lista *OutputGroups* o elemento na posição i tem que ser o elemento que está na posição j da *InputGroups*, sendo a posição j o elemento na posição i da lista *OutputIndexs* (*OutputGroups[i] = InputGroups[j] AND j = OutputIndexs[i]*).**É necessário garantir que os grupos estão corretamente atribuídos a cada lugar na solução do problema. Para isso foi utilizado o predicado *get\_groups(InputGroups, OutputIndexs, OutputGroups).*

**get\_groups**(\_,[],[]).

**get\_groups**(*InputGroups*, [*OutputIndexsH*|*OutputIndexsT*], [*OutputGroupsH*|O*utputGroupsT*]):-

**element**(*OutputIndexsH*, *InputGroups*, *OutputGroupsH*),

**get\_groups**(*InputGroups*, *OutputIndexsT*, *OutputGroupsT*).

**A distância de um elemento até ao próximo do mesmo grupo (caso exista) tem que ser 0.** Como o objetivo do problema é juntar as pessoas do mesmo grupo, impõe-se que a distância de um elemento de um grupo até ao próximo elemento do mesmo grupo caso este exista seja nula, para isso é usado o predicado *approximate(OutputGroups).*

**approximate**([]).

**approximate**([*OutputGroupsH*|*OutputGroupsT*]):-

**get\_distance**(*Distance*, *NotUnique*, *OutputGroupsT*, *OutputGroupsH*),

*NotUnique* #=> *Distance* #=0,

**approximate**(*OutputGroupsT*).

O predicado *get\_distance,* com a ajuda de um autómato, retorna em *Distance* a distância até ao primeiro elemento com o valor *OutputGroupsH* e caso este exista o predicado retorna em *NotUnique* o valor 1, caso contrário o valor 0.

## Função de Avaliação

Este problema é um problema de otimização, ou seja, o objetivo não é somente encontrar uma solução, mas sim encontrar a melhor solução.

Neste caso, a melhor solução é a que implica:

1. Menor distância percorrida por cada pessoa na mudança de lugar.
2. Menor número de trocas.

Assim, para cada lugar na distribuição final é calculada a diferença desse mesmo lugar e do lugar inicial da pessoa, isto é possível pois a lista *OutputGroups* guarda o lugar inicial na posição final da pessoa. Para isso é usado o seguinte predicado *fill\_differences(OutputIndexs,OutputIndexs, Differences).*

**fill\_differences**(\_,[],[]).

**fill\_differences**(*OutputIndexs*, [*OutputIndexsH*|*OutputIndexsT*], [*DifferencesH*|*DifferencesT*]):-

**element**(*OutputPos*, *OutputIndexs*, *OutputIndexsH*),

*DifferencesH* #= **abs**(*OutputPos*-*OutputIndexsH*),

**fill\_differences**(*OutputIndexs*, *OutputIndexsT*, *DifferencesT*).

Depois da lista *Differences* ter sido obtida, somam-se todos os elementos desta lista obtendo-se a variável *TotalDifference*

Seguidamente conta-se o número de elementos não zero da lista *Differences*, ou seja, contam-se os elementos que se moveram com a ajuda do predicado *get\_changes(NumOfChanges, Differences),* que usa um autómato para retornar o contador desejado em *NumOfChanges.*

Assim, para minimizar com igual peso o deslocamento das pessoas e o número de deslocamentos significa otimizar uma função (linear) objetivo:

*Minimize F = 1 \* TotalDifference + 1 \* NumOfChanges*

## Estratégia de Pesquisa

Foram testadas várias opções de pesquisa para a resolução deste problema. Para se poder chegar a alguma conclusão com os testes teve que ser usada a mesma distribuição inicial. Esta distribuição é uma plateia de 10 pessoas de 5 grupos diferentes, tendo cada grupo 2 pessoas: *InputGroups = [1,2,3,4,5,1,2,3,4,5].* A tabela 3 em Anexo apresenta os dados desses mesmos testes. Podemos então concluir que a melhor estratégia de pesquisa é a utilização das opções ***step min*** e a pior estratégia é o uso das opções *bisect max*.

# Visualização da Solução

O programa permite resolver o problema de otimização de Redistribuição de Público e para uma melhor demonstração da sua resolução, existem três predicados que permitem visualizar a solução em modo de texto.

De forma ao problema ser instanciado, deverá ser inserido na consola o predicado *problem*. Este predicado pode ser chamado de duas formas diferentes, recebendo apenas um argumento, uma lista completa do público a reordenar, ou dois argumentos, o número total de lugares e de grupos de pessoas.

No caso de receber dois argumentos, irá ser gerada, de forma aleatória, uma lista. O predicado *generateList*trata de gerar esta mesma lista.

**generateList**(0, [], \_).

**generateList**(*Counter*, [*Head*|*Tail*], *TotalGroups*) :-

*Counter* > 0,

*Counter1* **is** *Counter* - 1,

**random**(1, *TotalGroups*, *Head*),

**generateList**(*Counter1*, *Tail*, *TotalGroups*).

**problem**(*TotalAudience*, *TotalGroups*) :-

*MaxGroups* **is** *TotalGroups* + 1,

**generateList**(*TotalAudience*, *InputGroups*, *MaxGroups*),

**write**(' > INPUT GROUPS:'), **write**(*InputGroups*), **nl**,

**solve**(*InputGroups*, *TotalAudience*, *TotalGroups*, *OutputGroups*, *OutputIndexs*, *TotalDifference*, *NumOfChanges*),

**displayOutput**(*OutputGroups*, *OutputIndexs*, *TotalDifference*, *NumOfChanges*).

**problem**(*InputGroups*) :-

**length**(*InputGroups*, *TotalAudience*),

**maximum**(*TotalGroups*, *InputGroups*),

**solve**(*InputGroups*, *TotalAudience*, *TotalGroups*, *OutputGroups*, *OutputIndexs*, *TotalDifference*, *NumOfChanges*),

**write**(' > INPUT GROUPS: '), **write**(*InputGroups*), **nl**,

**displayOutput**(*OutputGroups*, *OutputIndexs*, *TotalDifference*, *NumOfChanges*).

Após ser resolvido o problema, o predicado *displayOutput*mostra a lista de grupos ordenada, a lista de índices ordenada, o número total de mudanças e a distância total de todas essas mudanças.

**displayOutput**(*OutputGroups*, *OutputIndexs*, TotalDifference, NumOfChanges) :-

**write**(' > OUTPUT GROUPS: '), **write**(*OutputGroups*), **nl**,

**write**(' > OUTPUT INDEXS: '), **write**(*OutputIndexs*), **nl**,

**write**(' > Total Changes: '), **write**(*NumOfChanges*), **nl**,

**write**(' > Total Changes Value: '), **write**(*TotalDifference*), **nl**.

# Resultados

Para se poderem tirar conclusões dos resultados obtidos foram medidos o tempo de resolução, o número de retrocessos e o número de restrições criadas. Seguem-se as condições de teste e as respetivas conclusões:

* **Fez-se variar o número de pessoas na audiência, mantendo-se o número de grupos (Tabela 4, Figura 1, Figura 2 e Figura 3 em Anexo).** O tempo de resolução do problema e o número de retrocessos variam exponencialmente com o aumento do número de pessoas da audiência, enquanto que o número de restrições criadas varia linearmente com o aumento de pessoas da audiência. Pode-se então concluir que o tempo de resolução depende do número de retrocessos e não do número de restrições criadas.
* **Fez-se variar o número de grupos, mantendo-se o número de pessoas na audiência (Tabela 5, Figura 4, Figura 5 e Figura 6 em Anexo).** Tal como nas condições anteriores, o tempo e o número de retrocessos variam da mesma forma, confirmando a conclusão de que o tempo depende do número de retrocessos e não do número de restrições criadas.

O número de restrições criadas mantém-se para o mesmo número de pessoas da audiência, com a exceção de quando os elementos são todos do mesmo grupo. Isto deve-se ao facto de que quando os elementos são do mesmo grupo, o domínio da variável *OutputGroups* será de 1 a 1, sendo logo atribuídos os valores à lista não precisando das restantes restrições.

O tempo varia exponencialmente quando o número de grupos varia de 1 a 5. Quando varia entre 6 e 9, o tempo varia de forma não conclusiva. Provavelmente será pelo facto de que, quando o nº de grupos é maior que 5 e porque o número de elementos é 10, haverá grupos só de 1 elemento enquanto outros grupos têm 2 elementos, isto vai tornar as condições inconstantes. Quando o número de grupos é 10, o tempo é muito pequeno pois só há um elemento de cada grupo, não havendo necessidade para trocas.

# Conclusões e Trabalho Futuro

O projeto teve como principal objetivo aplicar o conhecimento adquirido nas aulas teóricas e práticas, e foi concluído que a linguagem de Prolog, em particular, o módulo de restrições, é bastante útil para determinadas situações, como na resolução de problemas de decisão e otimização.

Ao longo do desenvolvimento deste projeto, foram encontradas algumas dificuldades, nomeadamente a escolha das restrições e a sua implementação. Após uma longa análise da biblioteca *clpfd* e dos slides fornecidos foi possível superar estas mesmas dificuldades.

Note-se que existem aspetos que podiam ser melhorados, como a escolha de um método mais eficiente e otimizado, dado que a nossa solução se demonstrou ser um pouco limitada tendo em conta o tempo que a aplicação demora a resolver o problema dependendo da sua dimensão.

Em suma, o projeto foi concluído com sucesso, visto solucionar corretamente o problema proposto, e o seu desenvolvimento contribuiu positivamente para uma melhor compreensão do funcionamento do *labeling e* variáveis de decisão, assim como na aplicação de restrições.

# Anexo

## Tabelas e Gráficos

**Tabela 3.** Testes de Estratégia de Pesquisa

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **leftmost** | **min** | **max** | **first\_fail** | **anti\_first\_fail** | **occurrence** | **most\_constrained** | **max\_regret** |
| **step** | 3,347 | 1,718 | 18,072 | 18,708 | 49,706 | 2,877 | 16,060 | 21,570 |
| **enum** | 3,533 | 5,125 | 18,717 | 18,383 | 21,699 | 3,121 | 19,539 | 20,850 |
| **bisect** | 3,223 | 2,997 | 73,378 | 17,607 | 56,772 | 2,932 | 16,652 | 27,014 |
| **middle** | 3,409 | 4,655 | 28,249 | 30,042 | 71,761 | 2,957 | 28,405 | 25,004 |
| **median** | 3,534 | 5,056 | 28,257 | 23,818 | 69,325 | 3,250 | 25,992 | 25,295 |

A **tabela 3** apresenta as várias durações (em segundos) da resolução do problema registadas para várias combinações de opções de labeling. As opções em cada linha são as que controlam de que modo é que as escolhas são feitas para cada variável selecionada, enquanto que as opções em cada coluna são as opções que controlam a ordem em que a próxima variável é escolhida para atribuição.

**Tabela 4.** Variação da duração, retrocessos e restrições criadas em função do número de pessoas

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Número de Grupos** | 3 |  |  |  |
| **Número de Pessoas** | **Tempo(s)** | **Retrocessos** | **Restrições Criadas** | **InputGroups** |
| **3** | 0,002 | 2 | 78 | [1,2,3] |
| **5** | 0,006 | 20 | 186 | [1,2,3,1,2] |
| **7** | 0,029 | 148 | 342 | [1,2,3,1,2,3,1] |
| **9** | 0,118 | 512 | 546 | [1,2,3,1,2,3,1,2,3] |
| **11** | 0,497 | 2869 | 798 | [1,2,3,1,2,3,1,2,3,1,2] |
| **13** | 6,080 | 54511 | 1098 | [1,2,3,1,2,3,1,2,3,1,2,3,1] |
| **15** | 24,400 | 311614 | 1446 | [1,2,3,1,2,3,1,2,3,1,2,3,1,2,3] |

**Tabela 5.** Variação da duração, retrocessos e restrições criadas em função do número de grupos

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Número de Pessoas** | 10 |  |  |  |
| **Número de Grupos** | **Tempo(s)** | **Retrocessos** | **Restrições Criadas** | **InputGroups** |
| **1** | 0,013 | 9 | 468 | [1,1,1,1,1,1,1,1,1,1] |
| **2** | 0,192 | 1300 | 666 | [1,2,1,2,1,2,1,2,1,2] |
| **3** | 0,264 | 1723 | 666 | [1,2,3,1,2,3,1,2,3,1] |
| **4** | 0,580 | 2696 | 666 | [1,2,3,4,1,2,3,4,1,2] |
| **5** | 1,618 | 8693 | 666 | [1,2,3,4,5,1,2,3,4,5] |
| **6** | 0,918 | 4839 | 666 | [1,2,3,4,5,6,1,2,3,4] |
| **7** | 1,087 | 5406 | 666 | [1,2,3,4,5,6,7,1,2,3] |
| **8** | 0,352 | 1527 | 666 | [1,2,3,4,5,6,7,8,1,2] |
| **9** | 0,786 | 4710 | 666 | [1,2,3,4,5,6,7,8,9,1] |
| **10** | 0,018 | 9 | 666 | [1,2,3,4,5,6,7,8,9,10] |

**Figura 1.** Variação da duração de resolução em função do número de pessoas

**Figura 2.** Variação do número de retrocessos em função do número de pessoas

**Figura 3.** Variação do número de restrições criadas em função do número de pessoas

**Figura 4.** Variação da duração do tempo em função do número de grupos

**Figura 5.** Variação do número de retrocessos em função do número de grupos

**Figura 6.** Variação do número de restrições criadas em função do número de grupos

## Código Fonte

:- use\_module(library(clpfd)).

:- use\_module(library(lists)).

:- use\_module(library(random)).

displayOutput(OutputGroups, OutputIndexs, TotalDifference, NumOfChanges) :-

write(' > OUTPUT GROUPS: '), write(OutputGroups), nl,

write(' > OUTPUT INDEXS: '), write(OutputIndexs), nl,

write(' > Total Changes: '), write(NumOfChanges), nl,

write(' > Total Changes Value: '), write(TotalDifference), nl.

generateList(0, [], \_).

generateList(Counter, [Head|Tail], TotalGroups) :-

Counter > 0,

Counter1 is Counter - 1,

random(1, TotalGroups, Head),

generateList(Counter1, Tail, TotalGroups).

problem(TotalAudience, TotalGroups) :-

MaxGroups is TotalGroups + 1,

generateList(TotalAudience, InputGroups, MaxGroups),

write(' > INPUT GROUPS:'), write(InputGroups), nl,

solve(InputGroups, TotalAudience, TotalGroups, OutputGroups, OutputIndexs, TotalDifference, NumOfChanges),

displayOutput(OutputGroups, OutputIndexs, TotalDifference, NumOfChanges).

problem(InputGroups) :-

length(InputGroups, TotalAudience),

maximum(TotalGroups, InputGroups),

solve(InputGroups, TotalAudience, TotalGroups, OutputGroups, OutputIndexs, TotalDifference, NumOfChanges),

write(' > INPUT GROUPS: '), write(InputGroups), nl,

displayOutput(OutputGroups, OutputIndexs, TotalDifference, NumOfChanges).

solve(InputGroups, TotalAudience, TotalGroups, OutputGroups, OutputIndexs, TotalDifference, NumOfChanges) :-

statistics(walltime, [Start,\_]),

*%Variáveis de Decisão*

length(OutputGroups, TotalAudience),

length(OutputIndexs, TotalAudience),

domain(OutputGroups, 1, TotalGroups),

domain(OutputIndexs, 1, TotalAudience),

*%Restrições*

all\_distinct(OutputIndexs),

get\_groups(InputGroups, OutputIndexs, OutputGroups),

approximate(OutputGroups),

*%Função de Avaliação*

fill\_differences(OutputIndexs, OutputIndexs, Differences),

sum(Differences, #=, TotalDifference),

get\_changes(NumOfChanges, Differences),

Min #= NumOfChanges + TotalDifference,

*%Labelling*

append(OutputGroups, OutputIndexs, Vars),

labeling([minimize(Min), step, min], Vars),

statistics(walltime, [End,\_]),

    Time is End - Start,

format(' > Duration: ~3d s~n', [Time]).

*%fd\_statistics.*

get\_groups(\_,[],[]).

get\_groups(InputGroups, [OutputIndexsH|OutputIndexsT], [OutputGroupsH|OutputGroupsT]):-

element(OutputIndexsH, InputGroups, OutputGroupsH),

get\_groups(InputGroups, OutputIndexsT, OutputGroupsT).

get\_distance(Counter, NotUnique, OutputGroupsT, Value) :-

distance\_signature(OutputGroupsT, Sign, Value),

automaton(Sign, \_, Sign,

[source(i), sink(i), sink(j)],

[arc(i,0,i, [C+1, NU+0]), arc(i,1,j, [C+0, NU+1]),

arc(j,0,j, [C+0, NU+0]), arc(j,1,j, [C+0, NU+0])],

[C, NU], [0,0], [Counter,NotUnique]).

distance\_signature([],[], \_).

distance\_signature([X|Xs], [S|Ss], Value):-

S in 0..1,

X#\=Value #=> S#=0,

X#=Value #=> S#=1,

distance\_signature(Xs, Ss, Value).

approximate([]).

approximate([OutputGroupsH|OutputGroupsT]) :-

get\_distance(Distance, NotUnique, OutputGroupsT, OutputGroupsH),

NotUnique #=> Distance #=0,

approximate(OutputGroupsT).

fill\_differences(\_,[],[]).

fill\_differences(OutputIndexs, [OutputIndexsH|OutputIndexsT], [DifferencesH|DifferencesT]) :-

element(OutputPos, OutputIndexs, OutputIndexsH),

DifferencesH #= abs(OutputPos-OutputIndexsH),

fill\_differences(OutputIndexs, OutputIndexsT, DifferencesT).

get\_changes(Counter, Differences) :-

changes\_signature(Differences, Sign),

automaton(Sign, \_, Sign,

[source(i), sink(i)],

[arc(i,0,i,[C+0]), arc(i,1,i, [C+1])],

[C], [0], [Counter]).

changes\_signature([],[]).

changes\_signature([X|Xs], [S|Ss]) :-

S in 0..1,

X#\=0 #=> S#=1,

X#=0 #=> S#=0,

changes\_signature(Xs,Ss).