Resolução de Problema de Decisão usando Programação em Lógica com Restrições: Carpooling

Bárbara Sofia Silva e Julieta Frade
FEUP-PLOG, Turma 3MIEIC01, Grupo Carpooling_4

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 carpooling, este, tem como finalidade obter o menor número de carros atendendo às especificações do grupo de viajantes. Assim, através da linguagem de Prolog, foi possível a resolução deste mesmo problema, que será abordada detalhadamente neste artigo.

Keywords: Carpooling, SICStus, Prolog, FEUP.

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 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 *Carpooling*.

O problema de otimização escolhido consiste na atribuição de carros e na formação de grupos que iram viajar nestes carros, isto é, visto que um grupo de pessoas forneçam informações como, se possuem carro, se desejam usar o seu carro, com quem gostariam de viajar e com quem não gostariam de viajar, nosso programa tem como objetivo minimizar o numero de carros necessários e criar grupos de viajem que satisfaçam os integrantes da viajem.

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.
    - **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.

2 Descrição do Problema

Carpooling é um problema de otimização. Este problema retrata a situação na qual um grupo de estudantes planejam viajar e desejam minimizar custos de transportes. Também é considerável que alguns grupos de pessoas, por exemplo, amigos, desejam viajar juntos e não com desconhecidos.

Portanto, pretende-se obter o menor número de carros necessários de modo a que o máximo de pessoas possíveis fiquem com aqueles que desejam viajar em conjunto. Também é de prioridade relevar que os carros são disponibilizados por aqueles que vão viajar e estes informam se desejam levar o carro ou não.

3 Abordagem

Na resolução deste problema na linguagem *Prolog* foi utilizada uma lista de listas com a estrutura apresentada abaixo. Cada sublista é composta por quatro elementos: id da pessoa, booleano para posse de carro, booleano para desejo de usar o próprio carro, id do grupo que deseja pertencer, id do grupo que deseja não pertencer.

Lista 1. Exemplo da Lista Inicial

```
/*nome | tem carro? (0,1) | deseja levar carro? (0,1) | grupo desejado (id) | grupo indesejado (id)*/
list(
[
[1,  0, 0, 1, 2],
[2,  1, 1, 2, 3],
[3,  1, 0, 3, 4],
[4,  0, 0, 4, 5],
[5,  1, 1, 5, 1],
[6,  1, 0, 1, 2],
[7,  1, 1, 1, 2],
[8,  0, 0, 3, 4],
[9,  0, 0, 2, 3],
[10, 0, 0, 4, 5],
[11, 1, 1, 4, 1],
[12, 0, 0, 2, 3],
[13, 1, 0, 2, 3],
[14, 0, 0, 3, 4],
[15, 0, 0, 1, 2],
[16, 0, 0, 4, 5],
[17, 1, 1, 3, 4],
[18, 0, 0, 5, 1],
[19, 1, 1, 2, 3],
[20, 0, 0, 3, 4],
[21, 0, 0, 4, 5],
[22, 0, 0, 1, 2]
]
).
```

3.1 Variáveis de Decisão

A solução do problema vem na forma de uma lista de listas, na qual cada lista interna corresponde a um carro, no qual o primeiro elemento é o motorista e seus próximos elementos são os integrantes do carro.

Tabela 2. Exemplo de Atribuição Final

```
L = [[2,19,13,12,9],[5,18],[7,22,15,6,1],[11,21,16,10,4],[17,20,14,8,3]]
```

Nota-se que o tamanho da solução é diretamente proporcional ao número de pessoas que pretendem viajar, visto que uma quantidade mínima de carros é necessário para a viajem. Em relação ao domínio, este vai de 1 até o id da última pessoa registrada.

3.2 Restrições

Os elementos da lista inicial têm que ser todos distintos. Como cada posição da lista inicial representa uma pessoa distinta, cada id deve ser único a fim de evitar conflitos. Isso é garantido através da função *all_distinct()*.

Na escolha dos motoristas, primeiro deve-se contabilizar aqueles que tem carro e desejam levar o mesmo. É necessário garantir que haja o mínimo de carros o suficiente, para tanto

verificamos quantas pessoas tem e desejam levar o próprio carro(*getDrivers(INPUT,NUMBER,DL)*), caso não seja o suficiente verificamos aqueles que tem carro e não desejam levar(getDriversExtra(INPUT,N2,DL1)).

```
solve(INPUT,NUMBER,OUTPUT):-
    length(INPUT, NAUX),
    NUMBER is ceiling(NAUX/5),
    getDrivers(INPUT,NUMBER,DL),
    length(DL,I2),
    (I2 #= N ->
        addWanted(INPUT,DL,OUTPUT)
    ;
        N2 is N-I2,
        getDriversExtra(INPUT,N2,DL1),
        append(DL,DL1,DLF),
        addWanted(INPUT,DLF,OUTPUT)
    ).
```

3.3 Estratégia de Pesquisa

A estratégia adotada foi a Heurística comum de *firts fail principle,* na qual tratamos os casos mais difíceis primeiro, evitando que eles piorem caso sejam colocados a parte e tratados posteriormente. Assim, escolhemos tratar das variáveis com o maior número de restrição por ordem de chegada das mesmas.

4 Visualização da Solução

O programa permite resolver o problema de otimização de *Carpooling* e para a demonstração da sua resolução é feita de acordo com o predicado apresentado abaixo.

De forma ao problema ser instanciado, deverá ser inserido através de um aquivo de texto uma lista com estrutura comentada anteriormente. Assim, o predicado c*arpooling/1* pode ser chamado de forma a apresentar cada um dos motoristas designados, o total de carros e suas respetivas composições.

```
| ?- carpooling(L).
Driver : 2
Driver : 5
Driver : 7
Driver : 11
Driver : 17
Total Cars: 5
L = [[2,19,13,12,9],[5,18],[7,22,15,6,1],[11,21,16,10,4],[17,20,14,8,3]]
```

5 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 que irão viajar.** O tempo de resolução do problema e o número de retrocessos variam exponencialmente com o aumento do número de pessoas, enquanto que o número de restrições criadas varia linearmente com este aumento. 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 conflitos entre os interesses dos viajantes.** Diferentemente do caso anterior, o principal fator que altera a execução do código é a quantidade de restrições. Ao aumentar o número de conflitos, torna-se necessário trata-los o que aumenta exponencialmente o tempo de execução.

6 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 algumas destas 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. Também é percetível como número de restrições é muito maior em um caso real, sendo que estas foram escolhidas não serem representadas por uma questão de praticidade, mas também poderia ser levado em conta carros de tamanhos diferentes, gastos de combustível, necessidade de transporte para deficientes físicos, entre outros.

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.

7 Bibliografia

- Rossi, F., van Beek, P. and Walsh, T. (Eds.): “Handbook of Constraint Programming”, Elsevier, 2006
- Dechter, R.: “Constraint Processing”, Morgan Kaufmann, 2003
- Roman Barták: “On-line Guide to Constraint Programming”, http://kti.mff.cuni.cz/~bartak/constraints/
- Documentação on-line do SICStus Prolog: https://sicstus.sics.se/documentation.html

8 Anexo

8.1 Código Fonte

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

/*nome | tem carro? (0,1) | deseja levar carro? (0,1) | grupo desejado (id) | grupo indesejado (id)*/
list(
[
[1,  0, 0, 1, 2],
[2,  1, 1, 2, 3],
[3,  1, 0, 3, 4],
[4,  0, 0, 4, 5],
[5,  1, 1, 5, 1],
[6,  1, 0, 1, 2],
[7,  1, 1, 1, 2],
[8,  0, 0, 3, 4],
[9,  0, 0, 2, 3],
[10, 0, 0, 4, 5],
[11, 1, 1, 4, 1],
[12, 0, 0, 2, 3],
[13, 1, 0, 2, 3],
[14, 0, 0, 3, 4],
[15, 0, 0, 1, 2],
[16, 0, 0, 4, 5],
[17, 1, 1, 3, 4],
[18, 0, 0, 5, 1],
[19, 1, 1, 2, 3],
[20, 0, 0, 3, 4],
[21, 0, 0, 4, 5],
[22, 0, 0, 1, 2]
]
).

carpooling(L):-
   list(X),
   solve(X,N,L), !,
   print(N).

solve(INPUT,NUMBER,OUTPUT):-
   length(INPUT, NAUX),
   NUMBER is ceiling(NAUX/5),
```

```
    getDrivers(INPUT,NUMBER,DL),
    length(DL,I2),
    (I2 #= N ->
        addWanted(INPUT,DL,OUTPUT)
    ;
        N2 is N-I2,
        getDriversExtra(INPUT,N2,DL1),
        append(DL,DL1,DLF),
        addWanted(INPUT,DLF,OUTPUT)
    ).

/*adiciona wanteds de cada driver*/
addWanted(_,[],_).

addWanted(X,[H|T],L):-
    getWantedGroup(X,H,G),
    getWantedIds(X,G,W),
    nonmember(W, [H|T]),
    append([H],W,CAR),
    all_distinct(CAR),
    addWanted(X,T,LAUX),
    append(LAUX,[CAR],L).

/*getWanted Ids*/
getWantedIds([],_,_).

getWantedIds([H|T], G, L):-
    getWantedIds(T,G,LAUX),
    match(H,3,N),
    checkId(H,N,G,I),
    append(LAUX,I,L).

checkId(H,G,G,[I]):-
    match(H,0,I).

checkId(_,_,_,[]).

/*getWanted Group*/
getWantedGroup([],_,_).

getWantedGroup([H|T], G, L):-
    match(H,0,N),
    checkGroup(H,N,G,L),
    getWantedGroup(T,G,L).

checkGroup(H,G,G,L):-
    match(H,3,L).

checkGroup(_,_,_,_).

/*pessoas que querem levar o carro*/
getDrivers([],_,_).

getDrivers(_,0,_).

getDrivers([H|T],N,DL):-
```

```
    match(H, 1, AUX1),
    match(H, 2, AUX2),
    match(H, 0, AUX3),
    checkDriver(N,N2,AUX3,AUX1,AUX2,D),
    getDrivers(T,N2,DAUX),
    append(DAUX,D,DL).

checkDriver(N,N2,AUX3,1,1,[AUX3]):-
    N2 is N-1,
    write('Driver : '),
    write(AUX3),nl.

checkDriver(N,N2,_,_,_,[]):-
    N2 is N.

/*pessoas que não querem levar o carro, mas tem*/
getDriversExtra([],_,_).

getDriversExtra(_,0,_).

getDriversExtra([H|T],N,DL):-
    match(H, 1, AUX1),
    match(H, 2, AUX2),
    match(H, 0, AUX3),
    checkDriverExtra(N,N2,AUX3,AUX1,AUX2,D),
    getDriversExtra(T,N2,DAUX),
    append(DAUX,D,DL).

checkDriverExtra(N,N2,AUX3,1,0,[AUX3]):-
    N2 is N-1,
    write('Driver : '),
    write(AUX3),nl.

checkDriverExtra(N,N2,_,_,_,[]):-
    N2 is N.

/*Get nth element*/
match([H|_],0,H).

match([_|T],N,H) :-
    N > 0,
    N1 is N-1,
    match(T,N1,H).
```