Problema de Otimização em Carpooling

Leonor M. Sousa [up201705377] e Sílvia J. M. Rocha[up201704684]

3MIEIC01 – *Carpooling_1*

Resumo. Descrição da implementação de uma solução de modelação de um problema de otimização como um PLR explicando a abordagem utilizada e demonstrando os resultados obtidos.

Palavras-chave: Otimização, *carpooling*, restrições, variáveis de decisão, função de avaliação, *labeling*.

1 Introdução

O segundo projeto da unidade curricular PLOG tem como objetivo a construção de um programa em Programação em Lógica com Restrições (PLR) para a resolução de um problema de otimização/decisão.

O nosso grupo selecionou, de entre as opções disponíveis, o problema de otimização em *carpooling.* A descrição deste problema encontra-se mais à frente neste documento.

2 Descrição do Problema

O problema em análise ao longo deste documento é um problema de otimização em que se pretende obter a máxima satisfação dos participantes de uma viagem de fim de curso ao mesmo tempo que se minimiza o número de carros a utilizar.

Para que isto seja possível, cada estudante deve indicar se tem ou não carro, a sua vontade (ou não) de levar carro, uma lista de colegas com quer formar grupo de viagem, e uma lista de colegas com quem não quer formar grupo de viagem.

A resolução deste problema deve, por isso, ser modelada de forma a ser capaz de resolver problemas deste género com diferentes parâmetros de entrada. Esta modelação deverá tentar minimizar a quantidade de carros usados, maximizando ao mesmo tempo o grau de satisfação dos estudantes (de levar ou não o seu carro, e relativamente à composição dos grupos de viagem).

3 Abordagem

3.1 Variáveis de Decisão

As variáveis de decisão definidas para o problema já descrito foram as seguintes:

- **MaxGroups**. Esta variável corresponde ao número máximo de carros que é possível utilizar para resolver o problema. Esta variável toma como valor o mais baixo entre o total de condutores e o total de participantes a dividir por dois (considerou-se que carros com apenas uma pessoa não representam um benefício de satisfação alto o suficiente para compensar os gastos).
- **ParticipantsNo.** Corresponde a uma lista com todos os participantes da viagem (representados por um número) e cujo domínio varia entre 1 e o número total de participantes (admite-se que são atribuídos números às pessoas de forma ordenada e sem saltar valores).
- **Participants.** Representa uma lista com todas as informações de todos os elementos a participar na viagem. Cada elemento desta lista, é em si uma lista cujo primeiro elemento é o número representativo de cada participante, o segundo é uma lista de pessoas com quem gostaria de fazer a viagem e o terceiro é uma lista de pessoas com quem não quer efetuar a viagem. O tamanho desta lista corresponde à variável *TotalParticipants* que representa o número de participantes.
- **OutputGroups.** Esta variável apresenta a solução para o problema de otimização em questão. Isto é, contém uma lista em que cada elemento representa um carro diferente. O tamanho desta variável está compreendido entre 1 e a variável *MaxGroups.*
- **OutputScore.** Esta variável representa a pontuação final da solução obtida sendo usada como método de avaliação da qualidade da solução.

3.2 Restrições

As restrições do problema foram implementadas como descrito abaixo.

- O tamanho da lista *OutputGroups* não pode ultrapassar o valor *MaxGroups* cuja forma de obtenção já foi explicada acima.
- Num determinado grupo, devem verificar-se as seguintes condições:
    - Para cada novo elemento a adicionar a um determinado grupo, é obrigatório que este elemento não pertença à lista de pessoas com quem não viajar dos restantes elementos do grupo.
    - De igual modo, nenhum dos elementos já presentes naquele grupo pode estar presente na lista de pessoas com quem não viajar do elemento a entrar.
    - Todos os grupos têm de ter pelo menos um condutor.

- Nenhum elemento pode estar em mais do que um grupo, simultaneamente.
- Um grupo só é fechado quando já atingiu a capacidade máxima de 5 pessoas ou quando já não é possível adicionar mais pessoas ao veículo (quer porque já todos os participantes foram alocados quer porque todos os que restam pertencem à lista de pessoas com quem não viajar de algum dos elementos já pertencente ao grupo).

3.3 Função de Avaliação

Foi definida uma função de avaliação para a solução obtida. Esta função calcula uma pontuação para cada um dos grupos formados segundo os critérios seguidamente apresentados:

- Para cada elemento, é atribuído um ponto por cada pessoa do seu grupo que pertencesse à sua lista de pessoas com quem viajar e é retirado um ponto por cada pessoa que pertencesse à sua lista de pessoas com quem não viajar (nos casos em que os elementos não pertencem a nenhuma destas listas a pontuação não é alterada, isto é, são atribuídos 0 pontos.
- Para cada grupo, é analisada a presença de pessoas que tenham dado a indicação de ter carro. Caso exista pelo menos um elemento que tenha dado a indicação de ter carro e demonstrado vontade em levá-lo, é atribuído mais um ponto à pontuação total do grupo. Caso não exista nenhum destes, é procurado um elemento que tenha dado indicação de ter carro mas que não o queira levar. Caso exista, é retirado um ponto à pontuação total do grupo. Caso não exista, falha e é procurada uma nova solução.

3.4 Estratégia de Pesquisa

Para procurar a melhor solução para o problema de otimização no caso do *carpooling*, foi aplicada uma estratégia de seleção de variáveis em que é maximizada a pontuação total dos grupos e minimizado o número de carros utilizados.

4 Visualização da Solução

Para visualizar a solução do problema de otimização modelada como um PLR é necessário invocar o predicado *carpooling(+Participants, +CanDrive, +WillDrive).* Em que:

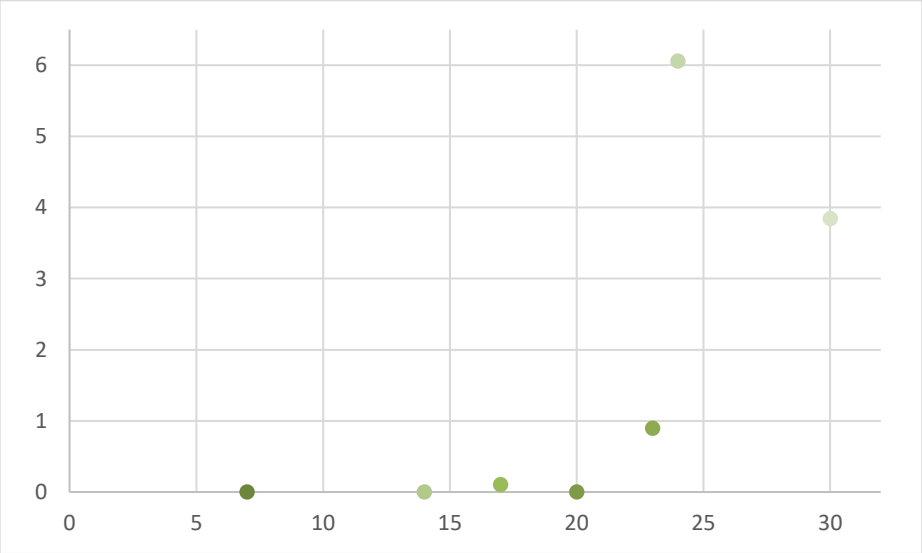
- *Participants* é uma lista de participantes em que cada elemento é do tipo *[número, [pessoas com quem viajar],[pessoas com quem não viajar]]*
- *CanDrive* é uma lista de participantes que deram indicação de ter carro mas não demonstraram vontade em levar carro.
- *WillDrive* é uma lista de participantes que deram indicação de ter carro e demonstraram vontade em levá-lo.

Exemplo de uma invocação do predicado descrito:

carpooling([[1, [2,3], [4]], [2, [5,6], [3]], [3, [1], []], [4 , [3,2], []], [5, [6,7], []], [6, [3], []], [7 ,[2], []]], [2,3], [7]).

5 Resultados

Executando a modelação como PLR que efetuamos com diferentes dimensões foi obtido o gráfico seguinte:



Como é possível denotar, o tempo de execução não varia linearmente com o tamanho dos grupos dado que, por vezes, acrescentar mais um elemento com as condições certas pode simplificar o problema ao invés de o complicar (exemplo: adicionar um elemento que tenha carro).

6 Conclusões e Trabalho Futuro

Finalizando o projeto e analisando o trabalho desenvolvido, pensamos que conseguimos modelar uma boa solução para o problema proposto dado que o programa encontra soluções que respeitam o número de condutores disponível tentando maximizar a satisfação dos participantes ao garantir que nenhum destes terá de viajar com algum dos elementos da sua lista de pessoas com quem não viajar.

Consideramos, no entanto, que existem algumas melhorias a serem efetuadas. A principal melhoria seria no sentido de procurar otimizar a função que cria os grupos de forma a avaliar a satisfação de uma forma mais global e não tão individualizada como se encontra atualmente.

7 Bibliografia

Consulta dos slides fornecidos no *moodle*

8 Anexos

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

displayRe-
sults(Participants, OutputGroups, TotalGroupsScore) :-
    write(' > PARTICIPANTS: '), write(Participants), nl,
    write(' > OUTPUT GROUPS: '), write(OutputGroups), nl,
    write(' > OUTPUT SCORE: '), write(TotalGroupsScore), nl
.

%Participants -> [number, [friends], [nemesis]].
getParticipants([],[]).
getParticipants([[Element | _] |Others], Participants):-
    getParticipants(Others, ParticipantsAux),
    append([Element], ParticipantsAux, Participants).

carpooling(Participants, CanDrive, WillDrive) :-
    length(Participants, TotalParticipants),
    solve(Participants, TotalParticipants, CanDrive, WillDr
ive, OutputGroups, TotalGroupsScore),
    getParticipants(Participants, Numbers),
    displayRe-
sults(Numbers, OutputGroups, TotalGroupsScore).

solve(Participants, TotalParticipants, CanDrive, WillDrive,
 OutputGroups, TotalGroupsScore) :-
    statistics(walltime, [Start,_]),

    %Variáveis de Decisão
    MaxGroups is TotalParticipants div 2,
    getParticipants(Participants, Numbers),
    domain(Numbers, 1, TotalParticipants),

    %Restrições
```

```
    get_groups(Participants, CanDrive, WillDrive, OutputGro
ups, []),
    length(OutputGroups, GroupSize),
    GroupSize #=< MaxGroups,
    append(OutputGroups, Vars),
    all_distinct(Vars),

    %Função de Avaliação
    calcu-
lateScore(Participants, CanDrive, WillDrive, OutputGroups,
GroupsScore),
    sum(GroupsScore, #=, TotalGroupsScore),

    label-
ing([maximize(TotalGroupsScore), minimize(GroupSize)], Vars
),

    statistics(walltime, [End,_]),
    Time is End - Start,
    format(' > Duration: ~3d s~n', [Time]).

delete_element([],_,[]).
de-
lete_element([[Element | T] | Others], ElementToDel, NewLis
t):-
    delete_element(Others, ElementToDel, NewListAux),
    (Element \= ElementToDel ->
        ap-
pend(NewListAux,[[Element|T]], NewList); NewList = NewListA
ux).

delete_occurencies([],_,[]).
de-
lete_occurencies([[Element, FriendsGroup, NemesisGroup] | O
thers], ElementToDel, NewOthers):-
    delete_occurencies(Others, ElementToDel, NewOthersAux),
    delete(FriendsGroup, ElementToDel, NewFriendsGroup),
```

```
    ap-
pend(NewOthersAux,[[Element, NewFriendsGroup, NemesisGroup]
], NewOthers).

delete_friends([],_,[]).
de-
lete_friends([ Element | Others], ElementToDel, NewList):-
    delete_friends(Others, ElementToDel, NewListAux),
    (Element \= ElementToDel ->
        ap-
pend(NewListAux,[Element], NewList); NewList = NewListAux).

findFriendsAndNeme-
sis([[Participant , Friends, Nemesis] | Others], Element, F
riendsGroup, NemesisGroup):-
    (Element == Participant ->
        FriendsGroup = Friends, NemesisGroup = Nemesis;
        findFriendsAndNeme-
sis(Others, Element, FriendsGroup, NemesisGroup)).

verifyNotMember(_,[],_,_).
verifyNotMem-
ber(Others, [Element | Group], NewElement, Nemesis):-
    findFriendsAndNeme-
sis(Others, Element, _Friends, NemesisElement),
    \+member(NewElement, NemesisElement),
    \+member(Element, Nemesis),
    verifyNotMember(Others, Group, NewElement, Nemesis).

create-
Group(_,_,Others,NewOthers,[],_,_,_, GroupAux, Group):-
 Group = GroupAux, NewOthers = Others.
create-
Group(_,_,Others,NewOthers,_,_,_,5, GroupAux, Group):-
 Group = GroupAux, NewOthers = Others.
create-
Group([Element, FriendsGroup, NemesisGroup], Others, Others
```

```
Aux, NewOthers, Participants, CanDrive, WillDrive, GroupSiz
e, GroupAux, Group):-
    member(NewElement, Participants),
    (FriendsGroup \= [] ->
        member(NewElement, FriendsGroup);
        \+member(NewElement, NemesisGroup)),
    findFriendsAndNeme-
sis(OthersAux, NewElement, _Friends, Nemesis),
    (\+verifyNotMember([[Element, FriendsGroup, NemesisGrou
p] | Others], GroupAux, NewElement, Nemesis) ->
        delete(FriendsGroup, NewElement, NewFriendsGroup),
        delete(Participants, NewElement, NewParticipants),
        create-
Group([Element, NewFriendsGroup, NemesisGroup], Others, Oth
ersAux, NewOthers, NewParticipants, CanDrive, WillDrive, Gr
oupSize, GroupAux, Group);
        de-
lete_element(OthersAux, NewElement, NewOthersAux),
        de-
lete_occurencies(NewOthersAux, NewElement, NewOthers2),
        de-
lete_friends(FriendsGroup, NewElement, NewFriendsGroup),
        delete(CanDrive, NewElement, NewCanDriveAux),
        delete(Participants, NewElement, NewParticipants),
        delete(WillDrive, NewElement, NewWillDriveAux),
        append(GroupAux, [NewElement], NewGroup),
        length(NewGroup, GroupSizeAux),
        create-
Group([Element, NewFriendsGroup, NemesisGroup], Others, New
Oth-
ers2, NewOthers, NewParticipants, NewCanDriveAux, NewWillDr
iveAux, GroupSizeAux, NewGroup, Group)).

delete_elements(_,[],Aux, Elements):- Elements = Aux.
de-
lete_elements(Participants, [Element | Group], NewElementsA
ux, NewElements):-
```

```
    de-
lete_element(NewElementsAux, Element, NewElementsAux2),
    de-
lete_elements(Participants, Group, NewElementsAux2, NewElem
ents).

delete_drivers([],_,[]).
delete_drivers([Driver | Others], Group, NewDrivers):-
    delete_drivers(Others, Group, NewDriversAux),
    (\+member(Driver, Group)->
        append(NewDriversAux, [Driver], NewDrivers);
        NewDrivers = NewDriversAux).

get_groups([],_,_, OutputGroups, OutputGroupsAux):-
 OutputGroups = OutputGroupsAux.
get_groups([ [Element, FriendsGroup, NemesisGroup] | Others
], CanDrive, WillDrive, OutputGroups, OutputGroupsAux):-
    getParticipants(Others, Participants),
    delete_occurencies(Others, Element, NewOthers2),
    create-
Group([Element ,FriendsGroup, NemesisGroup], NewOthers2, Ne
wOth-
ers2, NewOthers, Participants, CanDrive, WillDrive, 1, [Ele
ment], Group),
    delete_drivers(CanDrive, Group, NewCanDrive),
    delete_drivers(WillDrive, Group, NewWillDrive),
    append(OutputGroupsAux, [Group], OutputGroups2),
    get_groups(NewOthers, NewCanDrive, NewWillDrive, Output
Groups, OutputGroups2).

calculateElementScore(_,_,_,_,_,_,[],0).
calculateEl-
ementScore(Participants, CanDrive, WillDrive, Element, Frie
ndsGroups, NemesisGroups, [ Member | Group], Score):-
    calculateEl-
ementScore(Participants, CanDrive, WillDrive, Element, Frie
ndsGroups, NemesisGroups, Group, ScoreAux),
```

```
    (member(Member, FriendsGroups) -
> AuxScore = 1; (member(Member, NemesisGroups) -
> AuxScore = -1; AuxScore = 0)),
    Score is AuxScore + ScoreAux.

calculateGroupScore(_,_,_,[],0).
calculateGroup-
Score(Participants, CanDrive, WillDrive, [Element | Group],
  Score):-
    calculateGroup-
Score(Participants, CanDrive, WillDrive, Group, ScoreAux),
    findFriendsAndNeme-
sis(Participants, Element, FriendsGroups, NemesisGroups),
    calculateEl-
ementScore(Participants, CanDrive, WillDrive, Element, Frie
ndsGroups, NemesisGroups, Group, ElementScore),
    Score is ScoreAux + ElementScore.

findDriver([],_,_,0).
findDriv-
er([Element | Group], CanDrive, WillDrive, FinalScore):-
    findDriver(Group, CanDrive,WillDrive, FinalScoreAux),
    (member(Element, WillDrive) ->
        ScoreDriver = 1;
        (member(Element, CanDrive) ->
            ScoreDriver = 0 ;
            ScoreDriver = -1 )),
    FinalScore is FinalScoreAux + ScoreDriver.

calculateScore(_,_,_,[], []).
calcu-
lateScore(Participants , CanDrive, WillDrive,[OutputGroup |
 T], GroupsScore) :-
    calcu-
lateScore(Participants, CanDrive, WillDrive, T, GroupsScore
Aux),
```

```
    calculateGroup-
Score(Participants, CanDrive, WillDrive, OutputGroup, Score
),
    findDriv-
er(OutputGroup, CanDrive, WillDrive, ScoreDriverAux),
    length(OutputGroup, GroupSize),
    ComparableValue is 0 - GroupSize,
    NeutralValue is 1 - GroupSize,
    (ScoreDriverAux > ComparableValue ->
        (ScoreDriverAux > NeutralValue -
> ScoreDriver = 1; ScoreDriver = 0); fail),
    TotalScore is Score + ScoreDriver,
    append(GroupsScoreAux, [TotalScore], GroupsScore).
```