RELATÓRIO DO SPRINT 1

Turma 2DH _ Grupo 43
1190929 _ Patrícia Barbosa
1190947 _ Pedro Fraga
1190956 _ Pedro Garcia
1190963 _ Pedro Preto

Professora:
Brígida Teixeira, BCT

Unidade Curricular:
Algoritmia Avançada

Data: 04/12/2021

ÍNDICE

Determinar o Tamanho da Rede de um utilizador (até um determinado nível): 3

Caminho mais forte: 4

Caminho seguro: 5

Users com X tags em comum 5

Sugerir conexões com outros utilizadores tendo por base as tags e conexões partilhadas (até determinado nível) 7

Caminho mais curto 10

Determinar o Tamanho da Rede de um utilizador (até um determinado nível):

(Explicação da complexidade e do raciocínio encontram-se nas imagens)

```
/* O(n * log^2(n)) */
net_size_up_to_level(UserId, Level):- /* Este predicado existe para facilitar a interação com o utilizador */
    net_size_up_to_level_aux(UserId, Level, AllInNet).
```

```
/* considerando que o findall é O(n), o sort, O(log^2(n)), e o length, O(n),
como o flatten é O(log^2(n)) e o dfscl é O(n * log^2(n)),
o net_size_up_to_level_aux é O(n * log^2(n)) */
net_size_up_to_level_aux(UserId, Level, AllInNet):-
    findall(Path, dfscl(UserId, Dest, Path, Level), AllPaths),  /* Encontra todos os caminhos
    (uma lista de listas de utilizadores, cada caminho é uma lista de utilizadores) até um certo nivel de ligações */
    flatten(AllPaths, Aux), /* Transforma a lista de listas de utilizadores em uma única lista de utilizadores */
    sort(Aux, AllInNet), /* O sort é utilizado únicamente para remover os duplicados da lista */
    length(AllInNet, Size), /* Obtém o tamanho da lista de utilizadores, ou seja,
    o número de utilizadores na rede até ao nível escolhido */
    nl,write('All Users in Net Up to Level: '), write(AllInNet),nl, /* Escreve todos os utilizadores na rede até ao nível escolhido */
    write('Size of Net up to selected Level: '), write(Size),nl. /* Escreve o tamanho da rede até ao nível escolhido */
```

Métodos Complementares:

```
/* O(n * log^2(n)) */
dfscl(Orig, Dest, Cam, N):-
    dfscl2(Orig, Dest, [Orig], Cam, N).

/* O(n), presumindo que reverse é O(n) */
dfscl2(Dest,Dest,LA,Cam, _):-
    reverse(LA,Cam).

/* O(log^2(n)), presumindo que member é O(n) */
dfscl2(Act, Dest, LA, Cam, N):-
    N > 0, is_connected(Act, X), \+ member(X, LA), N1 is N - 1, dfscl2(X, Dest, [X|LA], Cam, N1).
```

```
/* O(n) */
is_connected(X,Y):- ligacao(X,Y,_,_); ligacao(Y,X,_,_). /* Verifica se dois users estão conectados,
sendo que as ligações são unidirecionais na base de conhecimento*/
```

```
/* O(log^2(n)), presumindo que o append é O(log(n)) */
flatten([],[]).
flatten([[H|T]|L], LF):- !, append([H|T],L,L1), flatten(L1,LF). /* Concatena todas as listas de uma lista em uma só lista */
flatten([X|L],[X|LF]):- flatten(L,LF).
```

Caminho mais forte:

```
:-dynamic caminho_somatorio/2. /*variavel dinamica com 2 argumentos*/
```

```
caminho_mais_forte(Orig,Dest,A,B):-
        asserta(caminho_somatorio([],0)),       /*inicializa a 0 o segundo parametro que representa o somatorio do caminho*/
        (verifica_caminho(Orig,Dest);true),
        caminho_somatorio(A,B), nl,             /*vai buscar os dados finais da variavel para os imprimir*/
        write("Caminho mais forte: "),write(A),write(" "),nl, write("Connection Strength: "),write(B),nl,!.
```

Método caminho_mais_forte vai ter complexidade O($log^2(n)$), uma vez que chama o método

verifica_caminho com complexidade de O($log^2(n)$).

```
verifica_caminho(Orig,Dest):-
    dfs(Orig,Dest,LCaminho),        /*chamada da funcao dfs que vai buscar os caminhos possiveis entre a origem e destino*/
    verifica_forca(LCaminho),       /*chamda de funcao*/
    fail.
```

Método verifica_caminho é recursivo e chama os métodos, dfs que apresenta uma complexidade de O(log(n)), e o método verifica_forca, que por sua vez chama o método get_somatorio com complexidade O(log(n)). Desta forma vai apresentar uma complexidade de O($log^2(n)$).

```
verifica_forca(LCaminho):-
    write('a verificar caminho '), write(LCaminho), nl,
    auxiliar_list(LCaminho,LResult),              /*Criar lista auxiliar para o caso de ser necessário atualizar a variavel global com o caminho e com o somatorio*/
    get_somatorio(LCaminho,Somatorio),                /*chamada da funcao get_somatorio*/
    caminho_somatorio(_,S),
    Somatorio>S,                                  /*verifica se o somatorio do caminho guardado é maior do que o somatório do caminho em questão*/
    retract(caminho_somatorio(_,_)),              /*caso o somatorio seja maior que o segundo argumento retiro o valor da variavel "caminho_somatorio"*/
    asserta(caminho_somatorio(LResult,Somatorio)).            /*coloco o novo caminho com o respetivo somatorio*/
```

Método verifica_forca chama o método get_somatorio com complexidade de log(n), logo também vai a apresentar uma complexidade de O(log(n)).

```
get_somatorio([],0).                        /*verifica se esta vazio*/
get_somatorio([_],0):-!.                    /*verifica se tem 1 elemento, seja ele qual for*/
get_somatorio([A,B|LCaminho],Somatorio):-
    no(Id1,A,_),                            /*tirar o id apartir do nome*/
    no(Id2,B,_),                            /*tirar o id apartir do nome*/
    is_bidirectional(Id1,Id2,F),                /*ir buscar a força*/
    get_somatorio([B|LCaminho], Somatorio2),        /*devolve B ao caminho*/
    Somatorio is Somatorio2 + F.                /*atualizar a variavel Total*/
```

Método get_somatorio é recursivo logo vai a apresentar uma complexidade de O(log(n)).

```
/*METODOS AUXILIARES*/

is_bidirectional(X,Y,Z):- ligacao(X,Y,Z,_); ligacao(Y,X,_,Z).   /*predicado para verificar se é bidirecional*/

auxiliar_list(LCaminho,LResult) :- cp(LCaminho,LResult).        /*predicado para criar a lista auxiliar*/
cp([],[]).
cp([H|T1],[H|T2]) :- cp(T1,T2).
```

Método is_bidirectional tem uma complexidade de O(n) pois percorre a base de dados até encontrar uma correspondência.

Método auxiliar_list é um método recursivo, logo apresenta complexidade de O(log(n)).

Caminho seguro:

```
:-dynamic caminho_seguro/2. /*variavel dinamica com 2 argumentos*/
```

```
caminho_mais_seguro(Orig,Dest,A,B,Xval):-
        asserta(caminho_seguro([],0)),      /*inicializa a 0 o segundo parametro que representa o somatorio do caminho*/
        (verifica_caminho(Orig,Dest,Xval);true),
        caminho_seguro(A,B),    nl,             /*vai buscar os dados finais da variavel para os imprimir*/
        write("Caminho mais forte: "),write(A),write(" "),nl, write("Connection Strength: "),write(B),nl,!.
```

Método caminho_mais_seguro vai ter complexidade O($log^2(n)$), uma vez que chama o método verifica_caminho com complexidade de O($log^2(n)$).

```
verifica_caminho(Orig,Dest,Xval):-
    dfs(Orig,Dest,LCaminho),                /*chamada da funcao dfs que vai buscar os caminhos possiveis entre a origem e destino*/
    verifica_seguranca(LCaminho,Xval),      /*chamada de funcao*/
    fail.
```

Método verifica_caminho é recursivo e chama os métodos, dfs que apresenta uma complexidade de O(log(n)), e o método verifica_forca, que por sua vez chama o método get_somatorio com complexidade O(log(n)). Desta forma vai apresentar uma complexidade de O($log^2(n)$).

```
verifica_seguranca(LCaminho,Xval):-nl,
    write('a verificar caminho '), write(LCaminho),
    auxiliar_list(LCaminho,LResult),                /*Criar lista auxiliar para o caso de ser necessário atualizar a variavel global com o caminho e com o somatorio*/
    get_somatorio(LCaminho,Somatorio,Xval),         /*chamada da funcao get_somatorio*/
    nl,write("SOMATORIO:"),write(Somatorio),nl,write(var(Somatorio)),nl,
    (var(Somatorio)==false,                         /*caso a variavel esteja inicializada prossegue caso contrário passa para o caminho seguinte */
    write(Somatorio),write(" > "),write(S),nl,
    caminho_seguro(_,S),
    Somatorio>S,                                    /*verifica se o somatorio do caminho guardado é maior do que o somatório do caminho em questão*/
    retract(caminho_seguro(_,_)),                   /*caso o somatorio seja maior que o segundo argumento retiro o valor da variavel "caminho_somatorio"*/
    asserta(caminho_seguro(LResult,Somatorio))).            /*coloco o novo caminho com o respetivo somatorio*/
```

Método verifica_seguranca chama o método get_somatorio com complexidade de log(n), logo também vai a apresentar uma complexidade de O(log(n)).

```
get_somatorio([],0,_).                          /*verifica se esta vazio*/
get_somatorio([_],0,_):-!.                      /*verifica se tem 1 elemento, seja ele qual for*/
get_somatorio([A,B|LCaminho],Somatorio,Xval):-
    no(Id1,A,_),                                /*tirar o id apartir do nome*/
    no(Id2,B,_),                                /*tirar o id apartir do nome*/
    is_bidirectional(Id1,Id2,F),nl,write("Forca: "),write(F),nl,
    (Xval>F->write("SOU INFERIOR"),get_somatorio([],0,Xval);write("SOU SUPERIOR"),get_somatorio([B|LCaminho],Somatorio2,Xval),Somatorio is Somatorio2 + F). /*se a força for inferior à escolhida
                                                                                                                                                        por parametro devolve a variavel somatorio por inicializar
                                                                                                                                                        caso contrário devolve o somatorio do caminho*/
```

Método get_somatorio é recursivo logo vai a apresentar uma complexidade de O(log(n)).

Users com X tags em comum

Declarar os factos dinâmicos, os users_com_x_tags_em_comum/1 serve para armazenar uma lista de listas de user que partilham das mesmas tags. O var_aux/1 serve para armazenar

```
:-dynamic var_aux/1.
:-dynamic users_com_X_tags_em_comum/1.
```

Metodo3, recebe por parametro LTags (lista de tags), X (numero de tags que os users têm de ter em comum) e LUsers (onde vai ser guardado o resultado).

Faz todas as combinações X a X possiveis de LTags e guarda-as na LcomXTags, coloca uma lista vazia no facto e chama o metodo4. Posteriormente guarda o valor de users_com_x_tags_em_comum em LUsers 1 e enviar a lista das tags combinadas X a X e a lista de listas de users que partilham dessas tags para o metodo imptime_Resultado de forma a escrever na consola. Por fim faz um retract colocando qualquer coisa nesse espaço em memoria.

Complexidade: O(log^2(n))

```
metodo3(LTags, X, LUsers):-
        todas_combinacoes(X, LTags, LcombXTags),
        asserta(users_com_X_tags_em_comum([])),
        (metodo4(LcombXTags, X);true),
        users_com_X_tags_em_comum(LUsers1),
        imprime_resultado(LcombXTags,LUsers1),
        retract(users_com_X_tags_em_comum(_)),
        write('Fim'), nl.
```

Vai percorrer todos os nos e guarda o user e a respetiv alista de tags e vai enviar juntamente com a LcombXTags para o metodo5. Este metodo obriga o falhanço com o fail de forma a percorrer todos os nos existentes. Quando não existir mais falha e juntamente com o fail entra na condição do metodo que o chamou (metodo4(LcomXTags, X);true).

Complexidade: O(n)

```
metodo4(LcombXTags, X):-
        no(Y, _, LOutroUser),
        write('User = '), write(Y),nl,
        ( metodo5(LcombXTags, Y, LOutroUser, X) ; true ),
        fail.
```

Recebe a lista com as listas de tags combinadas X a X, o user Y e a respetiva lista de tags LOutroUser e o X (numero de tags que os users têm de ter em comum). Coloca no facto var_aux uma lista vazia e chama o metodo6. Posterirmente guarda o valor do facto var_aux em LUsersCombinação e o valor do facto users_com_x_tags_em_comum de forma a poder acrescentar a esse mesmo facto a lista de users que partilham a combinação em questão.

Complexidade: O(n)

```
metodo5(LTags, Y, LOutroUser, X):-
        write('entrei'), nl,
        asserta(var_aux([])),
        (metodo6(LTags, Y, LOutroUser, X, LAux);true),
        var_aux(LUsersCombinacao),
        users_com_X_tags_em_comum(L),
        L1 = [LUsersCombinacao|L],
        asserta(users_com_X_tags_em_comum(L1)),
        retract(var_aux(_)),
        asserta(var_aux([])),
        fail.
```

Por fim o método 6 faz a interseção da cada lista de tags combinadas, com a lista do user Y e verifica se o seu tamanho é igual ou superior ao X e se assim for adiciona o user a lista auxiliar que vai ser guardada no facto var_aux.

Complexidade: O(n)

```
metodo6(LTags, Y, LOutroUser, X, LAux):-                          %O(n)
                        Lista = [LX|LTags],
                        intersecao(LX,LOutroUser, LIntersecao),
                        write('LIntersecao='), write(LIntersecao),nl,
                        length(LIntersecao, Tamanho),
                        %write('length='), write(Tamanho),nl,
                        Tamanho < X, fail;
                        write('encontrei o user: '), write(Y), nl,
                        var_aux(LUsersComXTagsPartilhadas),
                        %write('passei'), nl, nl,
                        LAux = [Y|LUsersComXTagsPartilhadas],
                        write('Guardei na LUsersComXTagsPartilhadas: '), write(LAux), nl,nl,
                        asserta(var_aux(LAux)).
```

Recebe duas listas por parâmetro e retorna a lista interseção de ambas. Verifica elemento a elemento da primeira lista se é membro da outra, se for chama novamente a interseço adicionando esse elemento à lista resultado , se não verifica se primeiro é sinónimo e faz o mesmo caso fosse membro ou então, caso não seja membro nem sinonimo chama o mÉtodo interseção mas sem adicionar o valor em questão a lista resultado.

Complexidade: O(log(n))

```
intersecao([ ],_,[ ]).
intersecao([X|L1],L2,[X|LI]):-membro(X,L2),!,intersecao(L1,L2,LI).
intersecao([X|L1],L2,[X|LI]):- ver_sinonimo(X,L2,LI),!, intersecao(L1,L2,LI).
intersecao([_|L1],L2, LI):- intersecao(L1,L2,LI).
```

Funciona de forma similar à interseção mas em vez de verificar se é membro, verifica se existe na base de conhecimento algum sinonimo.

Complexidade: O(log(n))

```
ver_sinonimo(_,[],_).
ver_sinonimo(X,[Tag|L2],[Tag|LI]):- sinonimo(X, Tag),!, ver_sinonimo(X,L2,LI).
ver_sinonimo(X,[Tag|L2],LI):- ver_sinonimo(X,L2,LI).
```

Complexidade: O(log(n))

```
todas_combinacoes(X,LTags,LcombXTags):-findall(L,combinacao(X,LTags,L),LcombXTags).
combinacao(0,_,[]):-!.
combinacao(X,[Tag|L],[Tag|T]):-X1 is X-1, combinacao(X1,L,T).
combinacao(X,[_|L],T):- combinacao(X,L,T).
```

Imprime o resultado final que resulta na combinação dos elementos de ambas as listas.

Complexidade: O(log(n))

```
imprime_resultado([],[]).
imprime_resultado([A|LcombXTags],[B|LUsers1]):- write('A combinacao '), write(A),       %O
                                                write(' é partilhada por '), write(B), nl,
                                                imprime_resultado(LcombXTags,LUsers1).
```

Sugerir conexões com outros utilizadores tendo por base as tags e conexões partilhadas (até determinado nível)

Para realizar a UC, foi criado o predicado suggest_users/3, cujos parâmetros são o ID do user a quem serão sugeridas conexões (UserId), o nível até qual poderão ser sugeridas as conexões (Level), e a lista a ser devolvida com os utilizadores sugeridos (ListaDeUsersSugeridos).

```
suggest_users(UserId, Level, ListaDeUsersSugeridos):-
  net_size_up_to_level(UserId, Level, Users),
  remove_item_from_list(UserId,Users,ListaDeUsers), /*Remover o próprio utilizador da lista de elementos*/
  remove_users_with_direct_connection(UserId,ListaDeUsers, UsersNearby),!, /**/
  no(UserId,_,ListaDeTags),suggest_users_2(UsersNearby,ListaDeTags,ListaDeUsersSugeridos),!.
```

O predicado começa por chamar o predicado net_size_up_to_level/3, elaborado no caso de uso "Determinar o tamanho da rede de um utilizador (até um determinado nível)", enviando o UserId, o Level, e uma lista (Users) que vai ser preenchida com os ID's de todos os utilizadores conectados até ao nível escolhido para o utilizador com o UserID enviado.

```
net_size_up_to_level(UserId, Level, List):-
    findall(Path, dfscl(UserId, Dest, Path, Level), AllPaths),
    flatten(AllPaths, Aux),
    sort(Aux, List).

flatten([],[]).
flatten([[H|T]|L], LF):- !, append([H|T],L,L1), flatten(L1,LF).
flatten([X|L],[X|LF]):- flatten(L,LF).

dfscl(Orig, Dest, Cam, N):-
    dfscl2(Orig, Dest, [Orig], Cam, N).

dfscl2(Dest,Dest,LA,Cam, _):-
    reverse(LA,Cam).

dfscl2(Act, Dest, LA, Cam, N):-
    N > 0, is_connected(Act, X), \+ member(X, LA), N1 is N - 1, dfscl2(X, Dest, [X|LA], Cam, N1).
    /**/
```

Contudo, a lista devolvida (Users) contém o ID do próprio utilizador, pelo que é necessário ser retirado, já que não faria sentido sere recomendada uma conexão consigo próprio. Assim, foi criado um predicado para poder remover items de uma lista, remove_item_from_list/3, que recebe como argumento o item a ser retirado da lista, neste caso o UserID, a lista do qual se quer retirar o elemento (Users), e uma lista que devolve a lista anterior sem o item enviado (ListaDeUsers).

```
remove_item_from_list(_,[],[]).
remove_item_from_list(Id,[Id|Users],List):-
    remove_item_from_list(Id,Users,List).
remove_item_from_list(Id,[User|Users],[User|List]):-
    remove_item_from_list(Id,Users,List).
```

Com o ID do próprio utilizador retirado, deparamo-nos com outro problema: a lista (ListaDeUsers) contém utilizadores que já estão conectados ao utilizador a quem vão ser sugeridas conexões. De modo a resolver este problema, foi criado o predicado remove_users_with_direct_connection/3. De maneira semelhante aos predicados anteriores, recebe 3 parâmetros: um ID e duas listas. O ID pertence ao utilizador a quem vão ser sugeridas conexões (UserID), a primeira lista é a devolvida no terceiro argumento do predicado remove_item_from_list/3, e o terceiro argumento é a lista de retorno (UsersNearby), que vai conter os ID's de todos os utilizadores até um certo nível, sem os de primeiro nível (conexão já existente) e sem o próprio utilizador a quem vão ser sugeridas conexões.

```
remove_users_with_direct_connection(_,[],[]).
remove_users_with_direct_connection(UserId,[HeadL1|TailL1],Lista2):- /*Lista 1 -> Lista de users ; */
  is_connected(UserId,HeadL1), remove_users_with_direct_connection(UserId,TailL1,Lista2)./*Lista 2 -> Lista sem utilizadores nivel 1 (já com cone
remove_users_with_direct_connection(UserId,[HeadL1|TailL1],[HeadL1 |Lista2]):-
  remove_users_with_direct_connection(UserId,TailL1,Lista2).

is_connected(X,Y):- ligacao(X,Y,_,_); ligacao(Y,X,_,_).
```

No predicado acima, cada utilizador presente na lista recebida no segundo argumento (através do seu id) é verificado se tem uma conexão com o utilizador com ID recebido no primeiro argumento, através do predicado auxiliar is_connected/2. Se for verificada a ligação, o id do utilizador que está na cabeça da lista do segundo argumento não é adicionado à lista de retorno. Caso não exista conexão, o id do utilizador é adicionado.

De seguida, voltando ao predicado suggest_users/3, busca-se a lista de tags do utilizador recebido no primeiro parâmetro, através do no/3, implementado na base de conhecimentos fornecida.

Por fim, é chamado o predicado auxiliar suggest_users_2/3, que tem como objetivo verificar se existem utilizadores (UsersNearby) com 1 ou mais tags em comum com a lista de tags recebida no segundo parâmetro (ListaDeTags) e, caso existam, são retornados na lista do terceiro argumento. Para tal, é percorrida recursivamente a lista de utilizadores próximos (UsersNearby) e busca-se a lista de tags do elemento à cabeça; de seguida recorrendo ao predicado intersecao/3, faz-se uma interseção entre as duas listas de tags (a ListaDeTags e do primeiro elemento do UsersNearby), e coloca-se numa lista auxiliar o resultado (X). Caso a length de X seja maior que 0, o utilizador cujo ID está à cabeça da lista do UsersNearby é adicionado à lista de retorno.

```
suggest_users_2([],_,[]). /*Se a lista de users a comparar estiver vazia*/
suggest_users_2([UserAtual|UsersNearby], ListaDeTags, ListaDeUsersSugeridos):-
  no(UserAtual,_,TagsUser2), intersecao(TagsUser2,ListaDeTags, X), length(X, Size),
  Size < 1, suggest_users_2(UsersNearby,ListaDeTags,ListaDeUsersSugeridos).
suggest_users_2([UserAtual|UsersNearby],ListaDeTags,[UserAtual|ListaDeUsersSugeridos]):-
  suggest_users_2(UsersNearby,ListaDeTags,ListaDeUsersSugeridos).

intersecao([],_,[]).
intersecao([X|L],L1,[X|LI]):-member(X,L1),!,intersecao(L,L1,LI).
intersecao([X|L],L1,LI):-intersecao(L,L1,LI).
```

Desta forma, se por exemplo, tentarmos sugerir utilizadores ao utilizador de ID 21, até ao nível 2, acontece o seguinte:

```
?- suggest_users(21, 2, UsersSugeridos).
UsersSugeridos = [1, 22, 23, 24, 41, 42, 43, 44].
```

Em termos de complexidade, o suggest_users/3 apresenta O(n * log ^{2}(n)), visto que chama o predicado net_size_up_to_level/3, de igual complexidade, e de todos os restantes predicados chamados apresentarem menor complexidade, nomeadamente:

- Suggest_users_2/3 ->
- intersecao/3 -> O(log(n))
- remove_item_from_list/3 -> O(log(n))
- remove_users_with_direct_connections/3->O(n * log(n))
- is_connected/2-> O(1)

Caminho mais curto

Facto com 2 argumentos que representam o caminho entre a origem e o destino e a distância entre eles.

```
:-dynamic melhor_sol_minlig/2.
```

O método plan_minlig(Orig, Dest, LCaminho_minlig) recebe dois users, um de origem e outro de destino e retorna em LCaminho_minlig o caminho mais curto entre eles. Começar por chamar o metodo melhor_caminho_minlig(Orig,Dest) que vai ser onde vai ser encontrada o caminho mais curto e posteriormente faz o retract do facto guardando o resultado em LCaminho_minlig.

Complexidade: O(log(n))

```
plan_minlig(Orig,Dest,LCaminho_minlig):-
        get_time(Ti),
        (melhor_caminho_minlig(Orig,Dest);true),
        retract(melhor_sol_minlig(LCaminho_minlig,_)),
        get_time(Tf),
        T is Tf-Ti,
        write('Tempo de geracao da solucao:'),write(T),nl.
```

Recebe dois users e chama o dfs enviando o caminho retornado pelo dfs para outro metodo que vai verificar se este é entao menor que ao que já está guardado no facto. Ao provocar o falhanço vai fazer o dfs até não existir mais soluções.

Complexidade O(n)

```
melhor_caminho_minlig(Orig,Dest):-
        asserta(melhor_sol_minlig(_,10000)),
        dfs(Orig,Dest,LCaminho),
        atualiza_melhor_minlig(LCaminho),
        fail.
```

Este método recebe um caminho e compara o seu tamanho com aquele que já esta guardado como caminho mais curto, se for menor, faz um asserta com um novo valor.

Complexidade: O(1)

```
atualiza_melhor_minlig(LCaminho):-
        melhor_sol_minlig(_,N),
        length(LCaminho,C),
        C<N,retract(melhor_sol_minlig(_,_)),
        asserta(melhor_sol_minlig(LCaminho,C)).
```

Complexidade: O(log(n))

```
dfs(Orig,Dest,Cam):-dfs2(Orig,Dest,[Orig],Cam).

dfs2(Dest,Dest,LA,Cam):-!,reverse(LA,Cam).
dfs2(Act,Dest,LA,Cam):-no(NAct,Act,_),(ligacao(NAct,NX,_,_);ligacao(NX,NAct,_,_)),
    no(NX,X,_),\+ member(X,LA),dfs2(X,Dest,[X|LA],Cam).
```