ALGAV – Sprint B

2021 / 2022

1191454 - Bruno Pereira

1180658 - Rafael Faísca

1191059 - Sérgio Balio

1170385 - Rui Mendes

Representação do conhecimento do domínio

Para o desenvolvimento do projeto foram utilizados os seguintes factos:

- **Nós:**
  Para os factos nó foi definido o seguinte formato:
  nó(userId, userName, tagsList).

  Ex: *nó(1,ana,[natureza, pintura, musica, sw, porto]).*

- **Ligações:**
  Para os factos ligação foi definido o seguinte formato:
  ligacao(userAId, userBId, connectionStrengthA-B, connectionStrengthB-A).

  Ex: *ligacao(1,11,10,8).*

Tamanho da rede do utilizador

Determinar o tamanho da rede de um utilizador até determinado nível.

O objetivo deste algoritmo é calcular o tamanho da rede de um utilizador e retornar uma lista com essa rede.

São adicionados os utilizadores que estão diretamente ligados ao utilizador origem, é verificado se existem utilizadores repetidos e a rede é atualizada removendo os repetidos, este processo é repetido para cada utilizador de cada nível até o nível pretendido.

```
tamanhoRedeUser2(Nivel, [UserAtual|QueueUsers], ProxQueue, Rede, ListaResultado):-
    % X será utilizadores conectados ao primeiro Utilizador na queue e que não se encontre na Rede
    findall(X,(
                ligacao(UserAtual, X,_,_),
                \+member(X, Rede)
            ),
            ListaUserNivel),

    % Adicionar lista de amigos ja visitados à Rede
    append(ListaUserNivel, Rede, ListaCompleta),

    % Criar lista de amigos visitados sem os amigos que já se encontrem em queue
    findall(Y,(
                member(Y, ListaUserNivel),
                \+member(Y, ProxQueue)
            ), UserNivelSemRepetidos),

    % Adicionar os amigos visitados que ainda não estão na queue à proxima queue
    append(UserNivelSemRepetidos, ProxQueue, NovaQueue),

    tamanhoRedeUser2(Nivel, QueueUtilizadores, NovaQueue, ListaCompleta, ListaResultado).
```

Sugerir conexões com tags em comum

Sugestão das conexões com outros utilizadores tendo por base as tags e conexões partilhadas até determinado nível.

O objetivo deste algoritmo é sugerir todas as conexões possíveis com utilizadores de determinado nível, onde todos os nós que fazem ligação até o utilizador destino partilham a mesma tag em comum com o utilizador origem.

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

sugerirConexoesPorTagComum(UserOrigem, Nivel, Resultado):-
    % Retorna a rede do utilizador ate ao nivel pretendido
    tamanhoRedeUser(UserOrigem, Nivel, Rede, _),

    % Remove os utilizadores da rede que tenham 0 tags em comum com o user
    redeTagsComuns(UserOrigem, Rede, UsersSugeridos),
    write('Possiveis Users Sugeridos: '),
    write(UsersSugeridos),
    nl,

    findall([UserDestino,Caminhos],(
                member(UserDestino, UsersSugeridos),
                UserDestino \== UserOrigem,

                % Lista de tags em comum entre o user origem e destino
                tagsComuns(UserDestino, UserOrigem, TagsComuns),

                write('User '), write(UserDestino),

                % Todos os caminhos possiveis do user origem ate o user destino onde todos os users do caminho teem
                dfsTagsComuns(UserOrigem, UserDestino, TagsComuns, Caminhos),

                write(': '), write(Caminhos),nl),
                Resultado),nl.
```

Inicialmente é gerada a rede do utilizador até o nível pretendido, feito no exercício anterior do tamanho da rede de um utilizador, de seguida são removidos os utilizadores da rede que não partilhem nenhuma tag em comum com o utilizador origem.

```
redeTagsComuns(User1, Rede, NovaRede):-
    no(User1, _, Tags),
    findall(User2, (
                member(User2, Rede),
                no(User2, _, Tags2),
                validaTagsComuns(Tags, Tags2)),
                NovaRede).
```

Depois de obtermos a rede do utilizador filtrada, é usado o findall para obter todas as soluções possíveis. Para cada utilizador destino possível pesquisamos as tags em comum entre o utilizador origem e destino, de seguida é chamado o DFS que verifica se cada utilizador do caminho partilha a mesma tag da lista de tags em comum nas ligações entre o utilizador origem e destino.

```
dfsTagsComuns(Orig, Dest, Tags, Caminhos):-
    findall(Caminho,(
                dfsTagsComuns2(Orig, Dest, [Orig], Caminho),
                % write('Caminho: '),
                % write(Caminho),nl,
                validaCaminhoTags(Caminho, Tags)),
                Caminhos).

dfsTagsComuns2(Dest, Dest, Lista, Caminho):-
    !,
    reverse(Lista, Caminho).

dfsTagsComuns2(UserAtual, Dest, Lista, Caminho):-
    % verifica ligaçao entre o UserAtual e um outro user X
    ligacao(UserAtual, X,_,_),

    % testar se X ja pertence à lista de users visitados
    \+member(X, Lista),

    dfsTagsComuns2(X, Dest, [X|Lista], Caminho).
```

Consultar o caminho mais curto

Consultar o caminho mais curto, ou seja, com menor número de ligações, para chegar a um determinado utilizador.

O objetivo deste algoritmo é determinar o caminho com menor número de ligações para do utilizador origem chegar ao utilizador destino.

Explicação do algoritmo

`:-dynamic melhor_sol_minlig/2.` aqui definimos melhor_sol_minlig(caminho,numero_ligacoes) na nossa base de conhecimentos para mais tarde a utilizarmos para guardar a solução mais curta ate ao momento.

```
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.
```

O predicado plan_minlig é o predicado chamado pelo utilizador para se determinar o caminho mais curto. Na 3° linha chama o predicado melhor_caminho_minlig que ira colocar a melhor solução na base de conhecimentos(melhor_sol_minlig).Na 4° linha colocamos a melhor solução no LCaminho_minlig para ser retornado ao utilizador e apagamos essa entrada da base de conhecimentos.

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

O predicado melhor_caminho_minlig comeca por adicionar a base de conhecimentos a melhor solução, sendo esta sem caminho e com o valor 10000 no número de ligações com o objetivo de ser a pior solução para depois poder ser alterada (só e alterada quando a ligação encontrada e melhor do que a q já se encontra na base de conhecimentos). Depois utiliza o predicado dfs, disponibilizado nas tps, para fazer uma busca em profundidade, ficando o caminho em LCaminho que depois e enviado por parâmetro para o predicado atualiza_melhor_minlig que ira atualizar na base de conhecimentos a melhor solução (entre a que já se encontra lá e o LCaminho).

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

Por fim, o predicado atualiza_melhor_minlig vai buscar a base de conhecimentos o número de ligações que a atual melhor solução tem, colocando-o em N e conta o número de ligações que o LCaminho, caminho recebido por parâmetro, colocando-o

em C. Depois se C for menor que N então retira a melhor solução que estava na base de conhecimentos e insere a nova melhor solução (LCaminho).

Consultar o caminho mais forte

Consultar o caminho mais forte, ou seja, com maior soma das forças de ligação em ambos os sentidos, para chegar a um determinado utilizador.

O objetivo deste algoritmo é determinar o caminho com o maior somatório das forças de ligação para o utilizador origem chegar ao utilizador destino.

Explicação do algoritmo

`:-dynamic melhor_sol_forte/3.` aqui definimos melhor_sol_forte (caminho, lista_forças_ligaçao, força_ligaçoes) na nossa base de conhecimentos para mais tarde a utilizarmos para guardar a solução mais forte até ao momento.

```
plan_forte(Orig,Dest,LCaminho_forte,Forca):- get_time(Ti),
    (melhor_caminho_forte(Orig,Dest);true),
    retract(melhor_sol_forte(LCaminho_forte,_,Forca)),
    get_time(Tf),
    T is Tf-Ti,
    write('Tempo de geracao da solucao: '), write(T),nl.
```

O predicado plan_forte é o predicado chamado pelo utilizador para se determinar o caminho mais forte. Na 2° linha chama o predicado melhor_caminho_forte que ira colocar a melhor solução na base de conhecimentos (melhor_sol_forte). Na 3° linha colocamos a melhor solução no LCaminho_minlig para ser retornado ao utilizador e apagamos essa entrada da base de conhecimentos.

```
melhor_caminho_forte(Orig,Dest):-
    asserta(melhor_sol_forte(_,_,-10000)),
    dfs_forte(Orig,Dest,LCaminho,LF),
    atualiza_melhor_forte(LCaminho,LF),
    fail.
```

O predicado melhor_caminho_forte começa por adicionar a base de conhecimentos a melhor solução, sendo esta sem caminho e lista de forças e com o valor -10000 na força de ligações com o objetivo de ser a pior solução para depois poder ser alterada (só é alterada quando a ligação encontrada é melhor do que a que já se encontra na base de conhecimentos). Depois utiliza o predicado dfs_forte, disponibilizado nas tps, para fazer uma busca em profundidade, ficando o caminho em

LCaminho e a lista de forças de ligação em LF que depois são enviados por parâmetro para o predicado atualiza_melhor_forte que ira atualizar na base de conhecimentos a melhor solução (entre a que já se encontra la e o LCaminho).

```
atualiza_melhor_forte(LCaminho,LF):-
    melhor_sol_forte(_,_,N),
    sumlist(LF,SF),
    SF>N, retract(melhor_sol_forte(_,_,_)),
    asserta(melhor_sol_forte(LCaminho,_,SF)).
```

Por fim, o predicado atualiza_melhor_forte vai buscar a base de conhecimentos e a força de ligações que a atual melhor solução tem, colocando-a em N e soma as forças de ligações do LCaminho, recorrendo a LF, colocando-a em SF. Depois se SF for maior que N então retira a melhor solução que estava na base de conhecimentos e insere a nova melhor solução (LCaminho).

Consultar o caminho mais seguro

Consultar o caminho mais seguro, ou seja, garante que não há uma força de ligação inferior a X considerando as forças nos dois sentidos da ligação, para chegar a um determinado utilizador.

O objetivo deste algoritmo é determinar o caminho com maior somatório das forças de ligação para o utilizador origem chegar ao utilizador destino garantindo que as forças de ligação em ambos os sentidos são maiores ou iguais a X.

Explicação do algoritmo

Para este predicado utilizamos o algoritmo do caminho mais forte com umas pequenas alterações.

```
plan_secure(Orig,Dest,LCaminho_forte,Forca,SEC):- get_time(Ti),
    (melhor_caminho_secure(Orig,Dest,SEC);true),
    retract(melhor_sol_forte(LCaminho_forte,LForca,Forca)),
    get_time(Tf),
    T is Tf-Ti,
    write('Tempo de geracao da solucao: '), write(T),nl.
```

O predicado plan_secure é o predicado chamado pelo utilizador para se determinar o caminho mais seguro. Na 2° linha chama o predicado melhor_caminho_secure que ira colocar a melhor solução na base de conhecimentos (melhor_sol_forte). Na 3° linha colocamos a melhor solução no LCaminho_forte para ser retornado ao utilizador e apagamos essa entrada da base de conhecimentos.

```
melhor_caminho_secure(Orig,Dest,SEC):-
    asserta(melhor_sol_forte(_,_,-10000)),
    dfs_forte(Orig,Dest,LCaminho,LF),
    min_list(LF,Min), Min >= SEC,
    atualiza_melhor_forte(LCaminho,LF),
    fail.
```

O predicado melhor_caminho_secure é imuito semelhante ao melhor_caminho_forte, tendo apenas a adição da condição que nenhuma das forças pode ser inferior à força mínima (SEC). Esta condição encontra-se na 4° linha.

Estudo da complexidade da determinação de caminhos

Nesta entrega do projeto de ALGAV analisamos a complexidade O(n) da determinação do caminho mais forte, curto e seguro.

Nº de camadas intermédias (sem nó 1 e 200)	Nº de nós por camada	Nº de Soluções	Tempo
1	1	1	0.0
2	2	4	0.0
3	3	27	0.0
4	4	256	0.0
5	5	3125	0.031176090240478516
6	6	46656	0.3593480587005615
7	7	823543	6.187011003494263
8	8	16777216	140.1806378364563
9	9	387420489	3398.5170300006866
10	10	10000000000	

FIGURA 1 – TABELA PARA LIGAÇÕES UNIDIRECIONAIS

Nº de camadas intermédias (sem nó 1 e 200)	Nº de nós por camada	Nº de Soluções	Tempo
1	1	1	0.0
2	2	4	0.0
3	3	27	0.02498602867126465
4	4	256	199.4347369670868
5	5	3125	
6	6	46656	
7	7	823543	
8	8	16777216	
9	9	387420489	
10	10	10000000000	

FIGURA 2 - TABELA PARA LIGAÇÕES BIDIRECIONAIS

Com estes dados podemos concluir que com o aumento do número de nós o número de soluções e o tempo de execução também aumenta, sendo que para as ligações bidirecionais a diferença é mais espontânea.

Se considerarmos o número de nós por camada com a letra N e o número de camadas intermédias, com a expressão: C^N conseguimos chegar ao número de soluções por nível.

Podemos assim concluir que os nossos algoritmos têm uma complexidade de O(C^N).