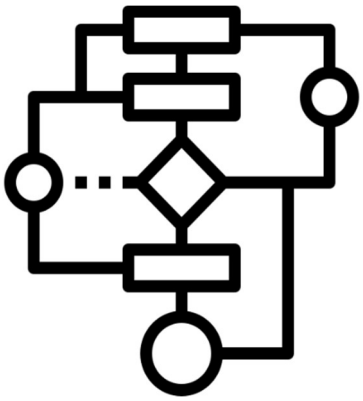
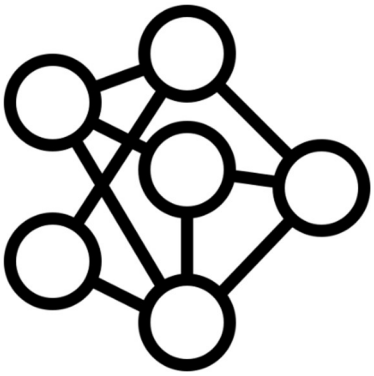
Relatório de Algoritmia Avançada

SPRINT B – Integração do módulo de pesquisas no contexto do projecto integrador





Turma 3NA
1180562 Diogo Vieira
1210811 Gustavo Couto
1040271 Bruno Santos
1210803 Salomé Teixeira

Data: 26/11/2023

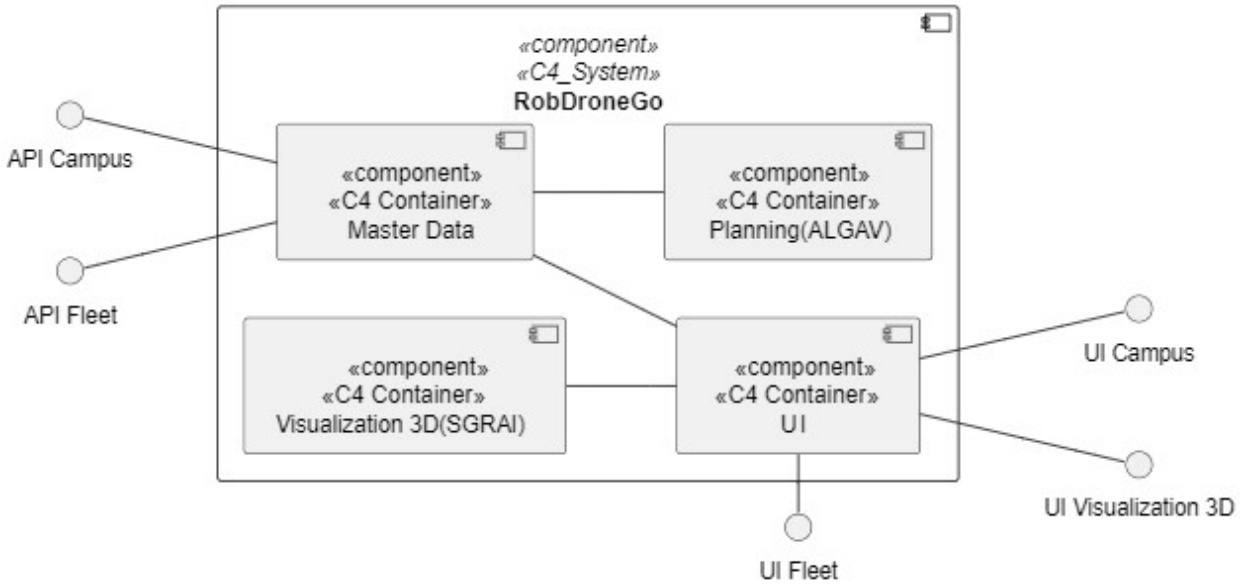
Indíce:

1. Representação do contexto do domínio

Este relatório visa expôr o trabalho realizado para a unidade curricular de Algoritmia Avançada no contexto do projeto integrador do 5º Semestre da licenciatura de Engenharia Informática. O domínio do projeto é o desenvolvimento de um sistema de gestão de campus e de uma frota de robôs e drones que desempenharão determinadas funções a pedido dos utentes do sistema. A componente de Algoritmia Avançada é responsável pela obtenção dos melhores percursos dentro do campus, de menção os dentro de cada piso e entre pisos/edifícios, contemplando os espaços interiores dos pisos assim como as ligações entre edifícios, de forma a promover uma melhor otimização dos recursos energéticos dos dispositivos assim como para promover a celeridade da realização dos serviços.

As componentes criadas procuram enquadrar algoritmos de pesquisa desenvolvidos em Prolog, integradas no projeto integrador via protocolo http fornecido pelos docentes, nomeadamente os de pesquisa em profundidade, em largura e o algoritmo A-Star. Dentro da organização estrutural do projeto, foi criado um módulo para desenvolvimento das bases de conhecimento assim como dos algoritmos(predicados) propriamente ditos.

Para a solução global foram contemplados 4 edificios cada um entre 2 a 4 pisos, conforme requerido .



2. Solução ótima para planeamento de movimentação entre pisos

De acordo com a solução de questão de exame fornecida foi-nos possível adaptar para o nosso caso de uso, uma vez que englobava as lógicas de negócio que nos eram requeridas, tais como a travessia entre pisos por elevadores e a travessia de corredores.

A solução passa por vários predicados acessórios, entre os quais, o predicado primeiro em profundidade com retrocesso **`caminho_edificios(EdOr,EdDest,LEDCam),`** tomando **`EdOr`** como o edifício de origem como parâmetro, **`EdDest`** como o edifício de destino, sendo **`LEDCam`** a lista de edifícios percorrida a ser retornada entre os dois edifícios.

```
caminho_edificios(EdOr,EdDest,LEdCam):-caminho_edificios2(EdOr,EdDest,[EdOr],LEdCam).

caminho_edificios2(EdX,EdX,LEdInv,LEdCam):-!,reverse(LEdInv,LEdCam).
caminho_edificios2(EdAct,EdDest,LEdPassou,LEdCam):-(liga(EdAct,EdInt);liga(EdInt,EdAct)),\+member(EdInt,
LEdPassou),
                                                    caminho_edificios2(EdInt,EdDest,[EdInt|
                                                    LEdPassou],LEdCam).
```

Para a definição da base de conhecimento deste predicado acessório foram definidas ligações bidireccionais entre edifícios sob a forma de **`liga(<códigoEdifício1>,<códigoEdifício2>)`**, para obtermos a representação dos nós e vértices do grafo do problema considerado.

O predicado **`caminho_edificios2(EdX,EdX,LEdInv,LEdCam):-!,reverse(LEdInv,LEdCam).`** é o caso base do predicado recursivo, uma vez que o edifício de origem seja igual ao edifício de destino, o percurso **`LedInv`** é invertido para obter a ordem correta do início ao fim e sendo retornado como **`LedCam.`**

Relativamente à parte recursiva, ou seja **`caminho_edificios2 (EdAct,EdDest,LEdPassou,LEdCam),`** esta verifica se existe uma ligação entre o edifício atual **`EdAct`** com um edifício intermédio **`EdInt,`** tal que este último não esteja dentro da lista de edifícios visitados **`LedPassou.`** A condição **`(liga(EdAct,EdInt);liga(EdInt,EdAct))`** garante a bidireccionalidade da ligação entre os edifícios e a condição **`\+member(EdInt,LEdPassou)`** garante que não visitamos o mesmo edifício duas vezes.

Em suma, este algoritmo implementa uma pesquisa primeiro em profundidade para obter um percurso entre dois edifício num grafo, iniciando num edifício de origem, explorando cada caminho o mais longe possível antes de efetuar o backtracking, garantindo que os edifícios não sejam visitados mais que uma vez, de forma a evitar ciclos infinitos.

O predicado anterior é utilisado noutro predicado acessório para obter a solução, sendo este o **`caminho_pisos(PisoOr,PisoDest,LEdCam,LLig),`** que associado aos factos adicionados à base

de conhecimento com informações sobre os pisos de cada edifício, os pisos servidos por cada elevador, assim como os corredores que unem edifícios, com a informação de que pisos são unidos por estes.

```
caminho_pisos(PisoOr,PisoDest,LEdCam,LLig):-pisos(EdOr,LPisosOr),member(PisoOr,LPisosOr),
                                             pisos(EdDest,LPisosDest),member(PisoDest,LPisosDest),
                                             caminho_edificios(EdOr,EdDest,LEdCam),
                                             segue_pisos(PisoOr,PisoDest,LEdCam,LLig).
```

O objetivo principal deste predicado é de fornecer um percurso entre pisos, um de origem **PisoOr** e um de destino **PisoDest**, através de vários edifícios interligados, entrando mais em detalhe, teremos para o primeiro predicado **pisos(EdOr,LPisosOr), member(PisoOr,LPisosOr)** que identifica o edifício **EdOr** que contem o piso de origem **PisoOr**, e o mesmo é feito de seguida para o edifício de destino através do **pisos(EdDest,LPisosDest), member(PisoDest,LPisosDest)** , depois o **caminho_edificios(EdOr,EdDest,LEdCam)** vai encontrar o percurso entre dois edifícios conforme explicado anteriormente.

Quanto ao predicado **segue_pisos**, este serve para determinar um percurso específico entre dois pisos, dado o percurso entre edifícios **LedCam** utilisado no predicado anterior.

Entrando em mais detalhe neste predicado, temos **segue_pisos (PisoDest,PisoDest,_,[])** como o caso base do mesmo , parando a recursividade quando o piso atual é o piso de destino, retornando uma lista vazia, significando que não há mais movimentos necessários a efetuar.

Posteriormente temos o caso de análise dos elevadores, em que primeiro é avaliado se os pisos de destino e de origem são diferentes, depois através do **elevador(EdDest,LPisos), member(PisoDest1,LPisos), member(PisoDest,LPisos)** é analisado se o edifício de destino possuí um elevador que sirva tanto o piso de destino como o de origem. O resultado deste predicado é uma lista com um único elemento **elev(PisoDest1, PisoDest),** indicando que uma viagem de elevador é necessária para movimentar entre os 2 pisos.

De seguida temos analisado o caso dos corredores, onde a cláusula segue_pisos**(PisoAct,PisoDest,[EdAct,EdSeg|LOutrosEd],[cor(PisoAct,PisoSeg)|LOutrasLig]):** lida com o cenário onde podemos mover de um piso para outro através de um corredor, através de **(corredor(EdAct,EdSeg,PisoAct,PisoSeg);corredor(EdSeg,EdAct,PisoSeg,PisoAct))** , onde é feita a verificação se existe um corredor que ligue o edifício atual **EdAct** no piso **PisoAct** com o edifício seguinte **EdSeg** no piso **PisoSeg**. O predicado chama-se novamente recursivamente com o edifício seguinte e os edifícios restantes do processo.

No final temos a análise mista (corredores e elevadores), onde neste contexto usamos **segue_pisos(PisoAct,PisoDest,[EdAct,EdSeg|LOutrosEd],[elev(PisoAct,PisoAct1),cor(PisoAct1,PisoSeg)|LOutrasLig]):.**

```
segue_pisos(PisoDest,PisoDest,_,[]).
segue_pisos(PisoDest1,PisoDest,[EdDest],[elev(PisoDest1,PisoDest)]):-
    PisoDest\==PisoDest1,
    elevador(EdDest,LPisos), member(PisoDest1,LPisos), member(PisoDest,LPisos).
segue_pisos(PisoAct,PisoDest,[EdAct,EdSeg|LOutrosEd],[cor(PisoAct,PisoSeg)|LOutrasLig]):-
    (corredor(EdAct,EdSeg,PisoAct,PisoSeg);corredor(EdSeg,EdAct,PisoSeg,PisoAct)),
    segue_pisos(PisoSeg,PisoDest,[EdSeg|LOutrosEd],LOutrasLig).
segue_pisos(PisoAct,PisoDest,[EdAct,EdSeg|LOutrosEd],[elev(PisoAct,PisoAct1),cor(PisoAct1,PisoSeg)|
LOutrasLig]):-
    (corredor(EdAct,EdSeg,PisoAct1,PisoSeg);corredor(EdSeg,EdAct,PisoSeg,PisoAct1)),PisoAct1\==PisoAct,
    elevador(EdAct,LPisos),member(PisoAct,LPisos),member(PisoAct1,LPisos),
    segue_pisos(PisoSeg,PisoDest,[EdSeg|LOutrosEd],LOutrasLig).
```

Em suma, `segue_pisos` contrói metodicamente um percurso consistido de movimentos de elevadores e de corredores entre edifícios.

Por fim, o predicado final **melhor_caminho_pisos(PisoOr,PisoDest,LLigMelhor)** que vai utilisar o predicado **caminho_pisos** previamente descrito , vai procurar o "melhor" caminho entre dois dados pisos num complexo de edifícios interligados, sendo que o critério para ser o melhor envolve usar o menor número de elevadores e/ou corredores possível.

```
melhor_caminho_pisos(PisoOr,PisoDest,LLigMelhor):-
    findall(LLig,caminho_pisos(PisoOr,PisoDest,_,LLig),LLLig),
    menos_elevadores(LLLig,LLigMelhor,_,_).
```

No predicado **menos_elevadores** , entrando em mais detalhe, temos como caso base **menos_elevadores([LLig],LLig,NElev,NCor)** que havendo apenas um percurso de sobra, este é retornado assim como o número de contagens de movimentos **NElev** de elevador de de movimentos **NCor** de corredores, tendo de seguida o caso recursivo **menos_elevadores ([LLig|OutrosLLig],LLigR,NElevR,NCorR)** que itera para cada percurso, comparando o número de movimentos de elevador e de corredor de forma a obter o melhor percurso, através de **conta(LLig,NElev1,NCor1)** que conta o número de elevadores e corredores utilizados para cada. O predicado de seguida compara o percurso obtido com os do resto da lista **LligM**, atualizando o melhor percurso **LligR**, com número de elevadores e corredores caso o percurso seja "melhor".

```
menos_elevadores([LLig],LLig,NElev,NCor):-conta(LLig,NElev,NCor).
menos_elevadores([LLig|OutrosLLig],LLigR,NElevR,NCorR):-
    menos_elevadores(OutrosLLig,LLigM,NElev,NCor),
    conta(LLig,NElev1,NCor1),
    (((NElev1<NElev;(NElev1==NElev,NCor1<NCor)),!,NElevR is NElev1, NCorR is NCor1,LLigR=LLig);
     (NElevR is NElev,NCorR is NCor,LLigR=LLigM)).
```

3. Soluções para movimentação do robô dentro de um piso

3.1 Consideração dos movimentos diagonais

De forma a criar as soluções para a movimentação do robot dentro de um piso, primeiro foi criada uma base de conhecimento onde definimos os valores de **1** e **0** para cada célula, de uma matriz que pode ir de 4 por 4 até 7 por 7 para depois vir a ser utilisada para o predicado de criação de grafo seguinte:

```
cria_grafo(_,0):-!.
cria_grafo(Col,Lin):-cria_grafo_lin(Col,Lin),Lin1 is Lin-
1,cria_grafo(Col,Lin1).

cria_grafo_lin(0,_):-!.
cria_grafo_lin(Col,Lin):-m(Col,Lin,0),!,
        ColS is Col+1, ColA is Col-1, LinS is Lin+1,LinA is Lin-1,
    ((m(ColS,Lin,0),assertz(ligacel(cel(Col,Lin),cel(ColS,Lin)));true)),
    ((m(ColA,Lin,0),assertz(ligacel(cel(Col,Lin),cel(ColA,Lin)));true)),
    ((m(Col,LinS,0),assertz(ligacel(cel(Col,Lin),cel(Col,LinS)));true)),
    ((m(Col,LinA,0),assertz(ligacel(cel(Col,Lin),cel(Col,LinA)));true)),

    ((m(ColS, LinS, 0), m(ColS, Lin, 0), m(Col, LinS, 0), assertz(ligacelm(cel(Col, Lin), cel(ColS, LinS)));true)),
    ((m(ColA, LinS, 0), m(ColA, Lin, 0), m(Col, LinS, 0), assertz(ligacelm(cel(Col, Lin), cel(ColA, LinS)));true)),
    ((m(ColS, LinA, 0), m(ColS, Lin, 0), m(Col, LinA, 0), assertz(ligacelm(cel(Col, Lin), cel(ColS, LinA)));true)),
    ((m(ColA, LinA, 0), m(ColA, Lin, 0), m(Col, LinA, 0), assertz(ligacelm(cel(Col, Lin), cel(ColA, LinA)));true)),
    Col1 is Col-1,
    cria_grafo_lin(Col1,Lin).
cria_grafo_lin(Col,Lin):-Col1 is Col-1,cria_grafo_lin(Col1,Lin).
```

Este predicado é um predicado recursivo que itera sobre as linhas da grelha da matriz, através do predicado **`cria_grafo_lin`** . O caso base do predicado de criação de matriz **`cria_grafo(_,0)`** utiliza o operador cut **`!`** para parar a recursão assim que o segundo argumento obtenha o valor **`0`**. Quanto ao predicado recursivo, o predicado **`cria_grafo_lin`** é então chamado para a linha atual, e cria_grafo vai ser recursivamente chamado, decrementando o número de linhas com **`Lin1 is Lin - 1`**.

Entrando em mais detalhe no predicado **`cria_grafo_lin`**, este processa cada célula por linha, verificando se esta possui um valor de **`0`** através de **`m(Col,Lin,0)`** ,e, sendo esse o caso, cria os vértices para com as suas células adjacentes com o valor **`0`**, através do predicado de Prolog **`assertz`** para adicionar ao longo do percurso factos de ligações das células, considerando os movimentos

horizontais e verticais antes e de seguida os movimentos diagonais permitidos, para a base de conhecimento, factos estes cruciais para a criação correta das ligações.

Esta recursão prossegue até todas as colunas numa dada linha serem processadas, verificado por **`cria_grafo_lin(0,_).`** O predicado **`ligacel`** é declarado como **`dynamic`** que permite ao programa de adicionar ou retirar factos desta natureza à base conhecimento durante a execução do programa, que é em suma utilizado para construir o grafo baseado na matriz criada na base de conhecimento inicial.

É ainda utilizado um outro predicado para a criação da matriz baseada no input de números e colunas fornecidos pelo utilizador:

```
:-dynamic ligacel/2.
:-dynamic m/3.
:-dynamic nlin/1.
cria_matriz:-
    retractall(m(_,_,_)),
    retractall(ligacel(_,_)),
    write('Numero de Colunas: '),read(NCol),nl,
    write('Numero de Linhas: '),read(NLin),nl,asserta(nlin(NLin)),
    cria_matriz_0(NCol,NLin),cria_grafo(NCol,NLin),retract(nlin(_)).

cria_matriz_0(1,1):-!,asserta(m(1,1,0)).
cria_matriz_0(NCol,1):-!,asserta(m(NCol,1,0)),NCol1 is NCol-1,nlin(NLin),cria_matriz_0(NCol1,NLin).
cria_matriz_0(NCol,NLin):-asserta(m(NCol,NLin,0)),NLin1 is NLin-1,cria_matriz_0(NCol,NLin1).
```

Os predicados **`ligacel`** e **`nlin`** são declarados como dinâmicos, com as implicações anteriormente mencionadas, depois por sua vez, o predicado **`cria_matriz`** inicializa o processo retirando todos os factos existentes **`m(_,_,_)`** e **`ligacel(_,_)`** da base de conhecimento, de forma a garantir uma inicialização sem dados corruptos, de seguida, o predicado passa por ler o número de colunas **`Ncol`** e linhas **`Nlin`** a partir do input do utilisador e guarda para a base de conhecimento **`nlin(NLin).`**

O predicado então chama cria_matriz_0(NCol, NLin) de forma a criar a matriz e **`cria_grafo(NCol, NLin)`** de seguida para criar o grafo baseado nesta mesma matriz, seguido da remoção **`nlin(_)`** para limpeza.

Quanto ao predicado **`cria_matriz0`**, este cria a matriz recursivamente inserindo dados do tipo **`m`** para a base de conhecimento, este por sua vez possui um predicado para o caso base **`cria_matriz_0(1,1)`** que gere a criação da última célula da matriz.

Para a primeira linha através de **`cria_matriz_0(NCol,1),`** o predicado insere na base de conhecimento a célula **`m(NCol,1,0)`** e de seguida chama-se a si próprio recursivamente com **`NCol-1`** de forma a processar as colunas restantes da primeira linha, já para as outras linhas, através de **`cria_matriz_0(NCol,NLin`**), este insere para a base de conhecimento a célula **`m(NCol,NLin,0)`** e de seguida chama-se recursivamente a si próprio com **`NLin-1`** com o fim de processar a linha anterior, continuando o processo até todas as células da matriz estarem criadas.

3.2 Primeiro em profundidade

Relativamente ao predicado de pesquisa de primeiro em profundidade (Depth-First-Search):

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

dfs2(Dest,Dest,LA,Cam):-
        reverse(LA,Cam).

dfs2(Act,Dest,LA,Cam):-
        ligacel(Act,X),
        \+ member(X,LA),
        dfs2(X,Dest,[X|LA],Cam).

all_dfs(Orig,Dest,LCam):-findall(Cam,dfs(Orig,Dest,Cam),LCam).

better_dfs(Orig,Dest,Cam):-all_dfs(Orig,Dest,LCam), shortlist(LCam,Cam,_).

shortlist([L],L,N):-!,length(L,N).
shortlist([L|LL],Lm,Nm):-shortlist(LL,Lm1,Nm1),
                          length(L,NL),
                    ((NL<Nm1,!,Lm=L,Nm is NL);(Lm=Lm1,Nm is Nm1)).
```

Neste predicado, este é iniciado com **`dfs(Orig, Dest, Cam)`** a partir de um nó de origem **`Orig`**, para um nó de destino **`Dest`** e procura um percurso **`Cam`**, este chama o predicado **`dfs2`** com a origem, o destino, uma lista contendo o nó de origem e uma variável para o percurso.

Quanto ao predicado **`dfs2`**, este é um predicado auxiliar para o **`dfs`**, tendo como caso base **`dfs2(Dest, Dest, LA, Cam)`** que verifica se o nó atual **`Act`** é o nó de destino , caso o seja, reverte a lista **`LA`**, que vai acumular o percurso em ordem inversa para obter o resultado final **`Cam`**. De seguida o caso recursivo **`dfs2(Act, Dest, LA, Cam)`** atravessa o grafo , e de forma a evitar ciclos, caso encontre um nó **`X`** ligado ao nó em curso **`Act`**, mais uma vez através do predicado **`ligacel`** verifica se **`X`** não está novamente na lista **`LA`**, e depois repete a recursão com **`X`** como o novo nó atual e com **`[X|LA]`** como sendo o percurso atualizado.

Relativamente ao predicado **`all_dfs(Orig, Dest, LCam)`**, este pura e simplesmente utiliza ao predicado **`findall`** para obter todas as soluções do predicado de pesquisa em profundidade anterior.

Quanto ao predicado **`better_dfs`**, este por sua vez vai procurar o caminho mais curto dentro de todos os percursos possíveis através do auxílio do predicado **`shortlist`**, cujo caso base **`shortlist([L], L, N)`** gere a situação onde apenas existe uma única lista, retornando ela mesma assim como o seu comprimento **N**. No seu predicado recursivo **`shortlist([L|LL], Lm, Nm)`** é feita a comparação entre o comprimento da lista atual, com a da lista mais curta encontrada nas listas reminiscentes **LL**, mantendo a mais pequena assim como o seu comprimento.

Ainda relativamente a Depth-First-Search temos mais um predicados, o **`betterdfs1`**:

```
:-dynamic melhor_sol_dfs/2.
better_dfs1(Orig,Dest,LCaminho_minlig):-
        get_time(Ti),
        (better_dfs11(Orig,Dest);true),
        retract(melhor_sol_dfs(LCaminho_minlig,_)),
        get_time(Tf),
        T is Tf-Ti,
        write('Tempo de geracao da solucao:'),write(T),nl.

better_dfs11(Orig,Dest):-
        asserta(melhor_sol_dfs(_,10000)),
        dfs(Orig,Dest,LCaminho),
        atualiza_melhor_dfs(LCaminho),
        fail.

atualiza_melhor_dfs(LCaminho):-
        melhor_sol_dfs(_,N),
        length(LCaminho,C),
        C<N,retract(melhor_sol_dfs(_,_)),
        asserta(melhor_sol_dfs(LCaminho,C)).
```

Iniciado pelo **`dynamic melhor_sol_dfs`**, como um predicado dinâmico , depois passamos pelo predicado **`better_dfs1(Orig, Dest, LCaminho_minlig),`** nesta instrução, são passados mais uma vez origem **`Orig`** e destino **`Dest`** e a variável de retorno **`LCaminho_minlig`** que retornará o caminho mínimo encontrado Na instrução **`retract(melhor_sol_dfs(LCaminho_minlig,_))`** removes a melhor solução atual e unimos **`LCaminho_minlig`** com o percurso encontrado. De seguida no predicado **`asserta(melhor_sol_dfs(_, 10000)),`** este adiciona um pior cenário artificial com um valor arbitrário elevado (10000) para estabelecer um termo de comparação inicial.

De seguida invocamos o predicado detalhado mais acima **`dfs(Orig, Dest, LCaminho),`** retornando o percurso e finalmente através do predicado acessório **`atualiza_melhor_dfs`**, é comparada a solução com a melhor até ao momento, caso seja melhor, o predicado remove a melhor solução atual e substitui pela recentemente descoberta com asserta(**`melhor_sol_dfs(LCaminho, C))`** onde **C** é o comprimento de **`Lcaminho.`**

3.3 Primeiro em largura

No que toca ao predicado de pesquisa em largura (Breadth-First-Search) foi utilisado o predicado seguinte:

```
bfs(Orig,Dest,Cam):-bfs2(Dest,[[Orig]],Cam).

bfs2(Dest,[[Dest|T]|_],Cam):-
        reverse([Dest|T],Cam).

bfs2(Dest,[LA|Outros],Cam):-
        LA=[Act|_],
        findall([X|LA],
                (Dest\==Act,ligacel(Act,X),\+ member(X,LA)),
                Novos),
        append(Outros,Novos,Todos),
        bfs2(Dest,Todos,Cam).
```

Onde em **`bfs(Orig, Dest, Cam)`** inicializa-se propriamente o predicado, onde, em semelhança ao predicado de pesquisa primeiro em profundidade, vai iniciar a pesquisa de um dado nó de origem **`Orig`** e um dado nó de destino **`Dest`**, retornando o percurso **`Cam`**.

Quanto ao predicado **`bfs2`**, este processa uma lista de percursos, cada um representado por uma lista de nós, na sua primeira cláusula, **`bfs2(Dest, [[Dest|T]|_], Cam`**), é verificado se o primeiro percurso na lista de percursos consegue chegar à solução, ou seja ao destino **`Dest`**, caso tal aconteça, vai reverter o percurso e unir ao **`Cam`**, uma vez que noa pesquisa em largura o percurso é construído em ordem inversa.

No que toca à segunda cláusula, **`bfs2(Dest, [LA|Outros], Cam),`** esta processa o primeiro percurso **`LA`** a partir da lista de percursos **`[LA|Outros`**], sendo o nó atual **`Act`** , cabeça da lista **`LA`**. Com a cláusula **`findall`** são reunidos todos os nós **`X`** onde haja ligações com o nó atual **`Act`** através do **`ligacel(Act, X)`**, e que não estejam no percurso **`LA`** como gerido por **`\+ member(X, LA)`**, evitando revisitar nós e prevenindo ciclos, de seguida, cada novo percurso **`[X|LA]`** é adicionado à lista de novos percursos Novos, posterioremente com **`append(Outros, Novos, Todos),`** é juntado os novos percursos para o fim da lista de percursos existentes, sendo de seguida chamado a ele próprio recursivamente, atualizando a lista de percursos **`Todos.`**

3.4 A*

Para a solução de pesquisa do algoritmo A-Star, foi criado o predicado seguinte:

```
aStar(Orig,Dest,Cam,Custo):-
aStar2(Dest,[(_,0,[Orig])],Cam,Custo).
aStar2(Dest,[(_,Custo,[Dest|T])|_],Cam,Custo):-
reverse([Dest|T],Cam).
aStar2(Dest,[(_,Ca,LA)|Outros],Cam,Custo):-
LA=[Act|_],
findall((CEX,CaX,[X|LA]),
(Dest\==Act,edge(Act,X,CustoX),\+ member(X,LA),CaX is CustoX + Ca, estimativa(X,Dest,EstX),CEX is CaX +EstX),Novos),
append(Outros,Novos,Todos),
sort(Todos,TodosOrd),
aStar2(Dest,TodosOrd,Cam,Custo).
```

Onde podemos verificar que se inicia pelo predicado **aStar**, chamando **aStar2** com um percurso inicial contendo apenas um nó, o de origem, e com um custo de percurso igual a 0. De seguida entramos no caso base do predicado, em que é verificado se o o percurso na lista contem o nó de destino **Dest** na sua cabeça em **aStar2(Dest,[(_,Custo,[Dest|T])|_],Cam,Custo)**, caso onde reverte a lista de nós para que seja fornecida na ordem correta, caso seja verificada a condição.

Na parcela recursiva do predicado verificamos a expansão dos percursos a partir do nó atual **Act**, gerando novos percursos para todos os nós adjacentes que não estejam no percurso atual, sendo que para cada percurso, é calculado **CaX** como o custo de alcance de um novo nó, **EstX** , como uma estimativa do custo a partir do novo nó para o destino (variável heurística) e **CEX** como a soma de **CaX** e **EstX** representando o custo total estimado do percurso caso atravesse por este novo nó.

De seguida, este acrescenta estes novos percursos para a lista de percursos existentes, organizando-os pelo seu custo total estimado e recursa novamente.

Este predicado utilisa um outro predicado, **estimativa**, que calcula a estimativa heurística do custo de um nó para outro, que é crucial para a eficiência do algoritmo A-Star, uma vez que direcciona a pesquisa para o nó de destino.

```
estimativa(Nodo1,Nodo2,Estimativa):-
node(Nodo1,X1,Y1),
node(Nodo2,X2,Y2),
Estimativa is sqrt((X1-X2)^2+(Y1-Y2)^2).
```

Foi ainda acrescentada uma base de conhecimento para poder testar o funcionamento deste predicado, onde foram adicionados nós para os pesos heurísticos dos edifícios, foram criados vértices entre os edifícios, assim como nós para cada sala dentro de cada piso, assim como vértices para as mesmas, contemplando igualmente os pesos heurísticos.

4. Integração das soluções de movimento dentro de pisos e entre pisos

Para a integração de ambas as soluções, foi criada uma base de conhecimento com informações das células de cada piso, usado o predicado seguinte para criar um grafo:

```
% Definindo dinamicamente o predicado ligacel para poder adicionar conexões
:- dynamic ligacel/2.

% Inicia a criação do grafo
cria_grafo :-
    cria_grafo(4, 4).

% Caso base: quando Lin é 0, apenas cria a conexão específica
cria_grafo(_, 0) :-
    estabelece_conexao, !.


% Caso recursivo: cria as ligações para uma linha e depois passa para a anterior
cria_grafo(Col, Lin) :-
    cria_grafo_lin(Col, Lin),
    Lin1 is Lin - 1,
    cria_grafo(Col, Lin1).


% Cria as conexões para uma linha específica
cria_grafo_lin(0, _) :- !.
cria_grafo_lin(Col, Lin) :-
    conecta_celulas(Col, Lin),
    Col1 is Col - 1,
    cria_grafo_lin(Col1, Lin).
  %    cria_grafo_lin(Col,Lin):-Col1 is Col-1,cria_grafo_lin(Col1,Lin).
```

Neste predicado para criação de grafo adaptado do das outras soluções, neste caso o caso base **`cria_grafo(_, 0) :- estabelece_conexao, !.`** estabelece ligações com corredores e elevadores, através do **`assertz`**. De seguida, **`cria_graf_lin`** , recursivamente vai iterar para cada coluna em cada linha, criando ligações em cada célula, de seguida, conecta_celulas garante a lógica de conexão, verificando certas condições para cada célula. Por final no **`conecta_adjacentes`**, liga a célula às suas células adjacentes na mesma matriz, calculando as coordenadas das células adjacentes (norte, sul, este, oeste e diagonais).

Para cada direção verifica a condição **`call(Matriz, ...)`** que caso se verifica como verdadeira, grava um facto para a base de conhecimento com **`assertz(ligacel(cel(Matriz, Col, Lin), cel(Matriz, ..., ...)))`** . O **`true`** a seguir ao **`;`** é para assegurar que o predicado tem sucesso mesmo que a condição falhe, permitindo ao processo de prosseguir.

5. Conclusões:

No contexto deste relatório, foi-nos possível tomar várias conclusões em vários campos:

Eficiência dos Algoritmos de Pesquisa:

No contexto do projeto de gestão de campus com robôs e drones, os algoritmos de pesquisa como Depth-First Search (DFS), Breadth-First Search (BFS) e A* desempenham um papel crucial. Estes algoritmos são essenciais para a navegação eficiente dos robôs dentro de um piso e entre diferentes pisos do edifício. Foi-nos possível detalhar como cada um destes algoritmos funciona e sua adequação para diferentes cenários dentro do campus. Por exemplo, o DFS pode ser mais adequado para exploração de áreas desconhecidas, enquanto o BFS é útil para encontrar o caminho mais curto num ambiente mapeado.

Análise de Tempo de Execução:

A análise dos tempos de execução desses algoritmos foi fundamental para entender a sua viabilidade no contexto requerido. O nosso trabalho mostra que, enquanto o algoritmo A* tende a ser mais rápido devido à sua natureza heurística, o BFS e o DFS têm os seus próprios méritos em termos de simplicidade e previsibilidade. Essa análise de tempo de execução ajuda a identificar qual algoritmo é mais adequado para uma tarefa específica, considerando os trade-offs entre rapidez e precisão.

Implicações Práticas:

A aplicação prática destes algoritmos no projeto de gestão de campus revela informações valiosas sobre como a teoria da computação pode ser aplicada num contexto realista. Por exemplo, a eficiência do A* em encontrar rotas ótimas rapidamente é contrabalançada pela sua complexidade aumentada em comparação com o BFS e o DFS. Essa compreensão ajuda a equipa do projeto a tomar decisões informadas sobre qual algoritmo implementar, dependendo das necessidades específicas da tarefa e das características do ambiente do campus.

Desafios e Oportunidades:

Este relatório também demonstra os desafios associados à implementação destes algoritmos, como a necessidade de otimizar a memória e o tempo de execução para evitar atrasos na movimentação dos robôs. Além disso, abre oportunidades para futuras melhorias, como a integração de algoritmos mais sofisticados ou a otimização dos algoritmos existentes para melhorar ainda mais a eficiência e a eficácia do sistema de gestão de campus, uma vez que os resultados apesar de não serem maus, caso houvesse mais tempo, seria possível uma boa otimização dos algoritmos.

6. Anexos

6.1. **Análise da Complexidade**

6.1.1. **Primeiro em profundidade**

Começando pelo predicados de pesquisa em profundidade **`dfs`** e **`dfs2`**, neste caso, cada aresta e nó do grafo serão visitados, no worst case scenario, cada vértice e cada nó serão visitados. A complexidade da pesquisa em profundidade é **O(V+E)**, sendo **V** os vértices (nós) e **E** o número de arestas, isto pois cada vértice é explorado pelo menos uma vez, e cada aresta é considerada uma vez.

Relativamente ao predicado **`all_dfs`**, este procura todos os percursos em profundidade possíveis, de um ponto de partida até ao destino, a complexidade será proporcional ao número de percursos, que pode ser bastante elevado, especialmente no caso de grafos altamente densos. Nessa situação de pior caso, o número de percursos pode variar exponencialmente com o número de nós, elevando o predicado para uma complexidade de **O(V!)**, sendo uma complexidade bastante elevada, tornando a execução lenta para casos de uma proporção amostral elevada.

Já relativamente à determinação do percurso mais curto, pelo meio de **`better_dfs`** e **`all_dfs`** conseguimos gerar todos os percursos e encontrar o efectivamente mais curto, a complexidade destes predicados é dominada pela complexidade de encontrar todos os percursos (**`all_dfs`**) .

Em última instância, o predicado shortlist itera por todos os percursos para encontrar o mais curto, que vai ser naturalmente da ordem **O(N)**, sendo N o número de percursos encontrados, no entanto, como N pode ser extremamente elevado, notavelmente pelos métodos anteriores, da ordem **O(V!).**

Em suma, para este predicado obtemos uma complexidade O(V!).

6.1.2. **Primeiro em largura**

No que toca ao predicado de pesquisa em largura, este iniciado pelo predicado **`bfs`** por si só não traz grande impacto na complexidade, no entanto no que toca ao **`bfs2`**, através da operação do **`findall`**, verificando cada aresta do nó atual, para cada nó, o algoritmo explora todos os vértices adjacentes, sendo então por consequência, no worst case scenario todas as arestas serão visitadas.

Como de forma geral, todos as arestas e nós irão ser visitados, então teremos uma complexidade da ordem **O(V+E)**, com **V** como os nós e **E** as arestas, onde contemplamos a visita de todos os nós e arestas pelo menos uma vez.

Em suma, para este predicado obtemos uma complexidade O(V+E).

6.1.3. **A***

Para o caso do A-Star, a sua complexidade temporal é dependente da lógica heurística implementada, no caso de a heurística ser eficiente na previsão do custo para o objetivo, a complexidade será da ordem **O(V)**, sendo **V** o número de vértices. No pior cenário, caso a lógica heurística não seja útil, a complexidade pode ascender para **O(V^2)** para uma matriz de adjacências **ou O(V + E log V)** no caso de uma lista de adjacências com uma fila de prioridade, sendo **E** o número de arestas.

Para o caso da nossa implementação, a exploração dos nós no predicado aStar é feita pelo `findall`, sendo que para cada nó, este considera todos os seus vizinhos e calcula o custo total (custo do percurso + custo heurístico). No predicado estimativa, é estimado o valor heurístico, que neste caso é a distância euclídiana entre nós, sendo que esta operação individual é de ordem **O(1)** para cada par de nós. No que toca ao predicado **append**, este combina os percursos recentemente descobertos com os existentes, a sua eficiência poderá variar, mas deverá ser da ordem **O(N)**, sendo **N** o comprimento da primeira lista. Relativamente ao `sort`, este filtra os percursos baseados no seu custo estimado total, a complexidade de filtrar pode ser de **O(N log N)** onde **N** é o número de percursos na lista.

No que toca à complexidade global, do algoritmo A-Star, num pior cenário será de ordem O(V^2) para má eficiência heurística e O(V + E log V) no caso de uma boa eficiência heurística

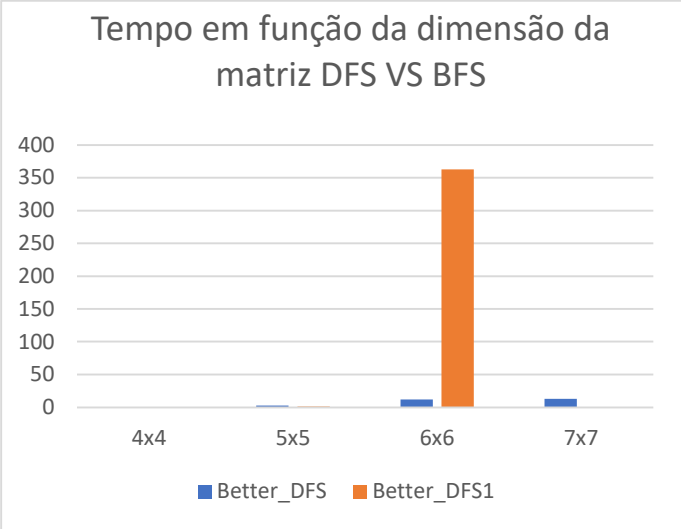
6.2. **Gráficos e tabelas de Performance**

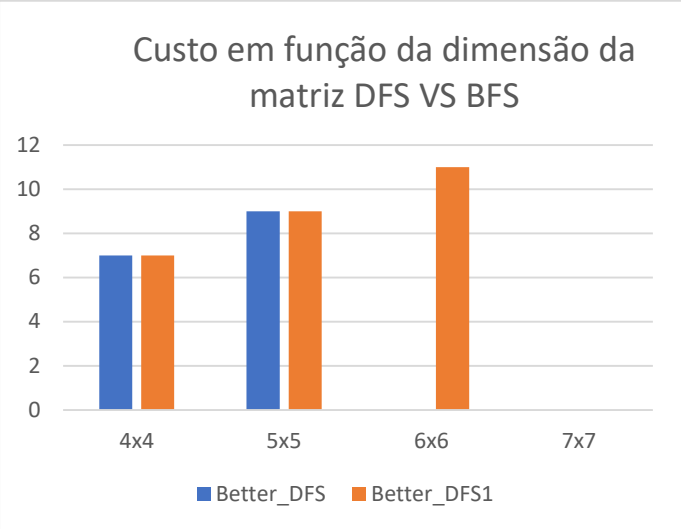
Estudou-se a viabilidade e a melhor solução obtida considerando os algoritmos de implementação primeiro em profundidade **`dfs`**, primeiro em largura **`bfs`**, melhor solução em profundidade **`better_dfs`** e melhor solução em profundidade sem **`find_all`** com o cálculo de tempo de geração **`TSol (better_dfs1+Tsol).`**

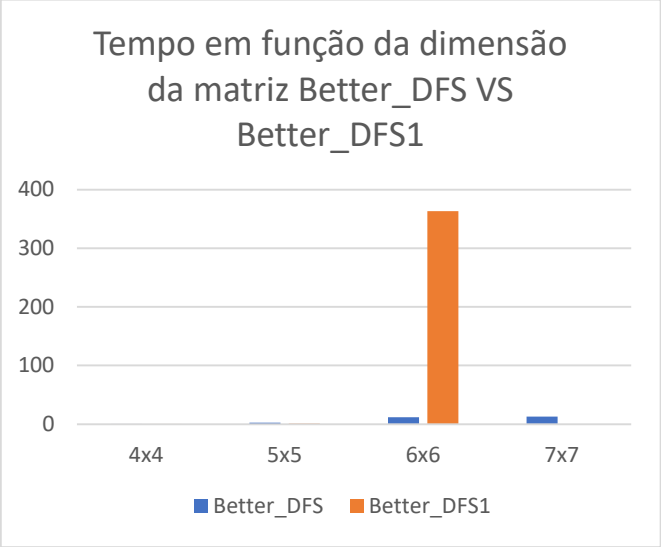
Na solução primeiro em profundidade **`dfs`**, primeiro em largura **`bfs`** e melhor solução em profundidade com o **`find_all`** e sem o algoritmo **`Tsol`**, o tempo foi cronometrado sendo considerada a sua contabilização para o tempo inicial a partir do momento da execução do algoritmo até ao tempo final pelo seu output. Já na solução melhor solução em profundidade sem o **`find_all`** o tempo foi calculado com recurso ao algoritmo **`TSol`**.

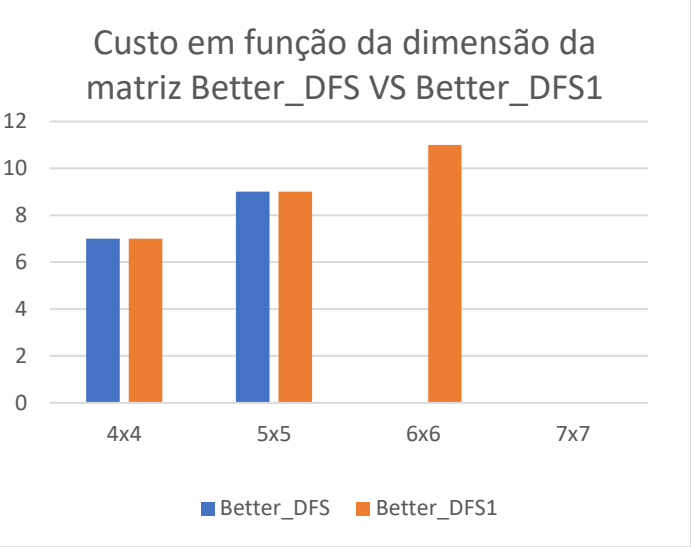
Quando este tudo foi realizado ainda não estavam a ser considerados os movimentos diagonais, contudo este movimento foi considerado mais tarde nos algoritmos considerados e desenvolvidos.

NColx NLin	Nº nós	Nº lig	Solução 1ºProf (dfs)	Custo e Tempo	Solução 1.º em Largura (bfs)	Custo e Tempo	Melhor Solução - Todas 1. Profundidade (better_dfs)	Custo e Tempo	Melhor Solução - Todas 1. Profundidade (better_dfs1+TSol)	Custo e Tempo
4*4	16	48	[cel(1, 1), cel(2, 1), cel(3, 1), cel(4, 1), cel(4, 2), cel(3, 2), cel(2, 2), cel(1, 2), cel(1, 3),cel(2, 3),cel(3, 3),cel(4, 3),cel(4, 4)]	Custo=13 Tempo=0s	[cel(1, 1), cel(2, 1), cel(3, 1), cel(4, 1), cel(4, 2), cel(4, 3), cel(4, 4)]	Custo=7 Tempo=0,001s	[cel(1, 1), cel(1, 2), cel(1, 3), cel(1, 4), cel(2, 4), cel(3, 4), cel(4, 4)].	Custo=7 Tempo=0,080s	[cel(1, 1), cel(1, 2), cel(1, 3), cel(1, 4), cel(2, 4), cel(3, 4), cel(4, 4)].	Custo=7 Tempo=0,014s
5*5	25	75	Cam=[cel(1, 1),cel(2, 1),cel(3, 1),cel(4, 1),cel(5, 1),cel(5, 2),cel(4, 2),cel(3, 2),cel(2, 2),cel(1, 2),cel(1, 3),cel(2, 3),cel(3, 3),cel(4, 3),cel(5, 3),cel(5, 4),cel(4, 4),cel(3, 4),cel(2, 4),cel(1, 4),cel(1, 5),cel(2, 5),cel(3, 5),cel(4, 5),cel(5, 5)]	Custo=25 Tempo=0,163s	[cel(1, 1),cel(2, 1),cel(3, 1),cel(4, 1),cel(5, 1),cel(5, 2),cel(5, 3),cel(5, 4),cel(5, 5)]	Custo=9 Tempo=0,641s	[cel(1, 1),cel(1, 2),cel(1, 3),cel(1, 4),cel(1, 5),cel(2, 5),cel(3, 5),cel(4, 5),cel(5, 5)].	Custo=9 Tempo=2,377s	[cel(1, 1), cel(2, 1), cel(3, 1), cel(4, 1), cel(5, 1), cel(5, 2), cel(5, 3), cel(5, 4), cel(5, 5)]	Custo=9 Tempo=0,980s
6*6	36	108	Cam=[cel(1, 1),cel(2, 1),cel(3, 1),cel(4, 1),cel(5, 1),cel(6, 1),cel(6, 2),cel(5, 2),cel(4, 2),cel(3, 2),cel(2, 2),cel(1, 2),cel(1, 3),cel(2, 3),cel(3, 3),cel(4, 3),cel(5, 3),cel(6, 3),cel(6, 4),cel(5, 4),cel(4, 4),cel(3, 4),cel(2, 4),cel(1, 4),cel(1, 5),cel(2, 5),cel(3, 5),cel(4, 5),cel(5, 5),cel(6, 5),cel(6, 6)]	Custo=31 Tempo=0,353s	[cel(1, 1), cel(2, 1), cel(3, 1), cel(4, 1), cel(5, 1), cel(6, 1), cel(6, 2), cel(6, 3), cel(6, 4), cel(6, 5), cel(6, 6)]	Custo= 11 Tempo=2,171s	Erro de stack	Custo=Erro Tempo=11,778 s	[cel(1, 1), cel(2, 1), cel(3, 1), cel(4, 1), cel(5, 1), cel(6, 1), cel(6, 2), cel(6, 3), cel(..., ...)|...].	Custo=11 Tempo=362.991 s
7*7	49	147	Cam = [cel(1, 1), cel(2, 1), cel(3, 1), cel(4, 1), cel(5, 1), cel(6, 1), cel(7, 1), cel(7, 2), cel(..., ...)|...]	Custo= Tempo=0,422s	Erro de stack	Custo=Erro Tempo=8s	Erro de stack	Custo=Erro Tempo=12,944 s	Não devolveu resultado. Esteve 2,5h a processar.	-









6 .3**. Outputs das Soluções**

BFS + DFS (*3_us515_analise_complexidade.pl*)

4x4:

```
?- cria_matriz.
Numero de Colunas: 4.

Numero de Linhas: |: 4.

true .

?- dfs(cel(1,1),cel(4,4),Cam).
Cam = [cel(1, 1), cel(2, 1), cel(3, 1), cel(4, 1), cel(4, 2), cel(3, 2), cel(2, 2), cel(1, 2), cel(..., ...)|...] .

?- bfs(cel(1,1),cel(4,4),Cam).
Cam = [cel(1, 1), cel(2, 1), cel(3, 1), cel(4, 1), cel(4, 2), cel(4, 3), cel(4, 4)] .

?- better_dfs(cel(1,1),cel(4,4),Cam).
Cam = [cel(1, 1), cel(1, 2), cel(1, 3), cel(1, 4), cel(2, 4), cel(3, 4), cel(4, 4)].

?- better_dfs1(cel(1,1),cel(4,4),Cam).
Tempo de geracao da solucao:0.0040040016174316406
Cam = [cel(1, 1), cel(2, 1), cel(3, 1), cel(4, 1), cel(4, 2), cel(4, 3), cel(4, 4)].
```

5x5:

```
?- cria_matriz.
Numero de Colunas: 5.

Numero de Linhas: |: 5.

true .

?- dfs(cel(1,1),cel(5,5),Cam).
Cam = [cel(1, 1), cel(2, 1), cel(3, 1), cel(4, 1), cel(5, 1), cel(5, 2), cel(4, 2), cel(3, 2), cel(..., ...)|...] .

?- bfs(cel(1,1),cel(5,5),Cam).
Cam = [cel(1, 1), cel(2, 1), cel(3, 1), cel(4, 1), cel(5, 1), cel(5, 2), cel(5, 3), cel(5, 4), cel(..., ...)] .

?- better_dfs(cel(1,1),cel(5,5),Cam).
Cam = [cel(1, 1), cel(1, 2), cel(1, 3), cel(1, 4), cel(1, 5), cel(2, 5), cel(3, 5), cel(4, 5), cel(..., ...)].

?- better_dfs1(cel(1,1),cel(5,5),Cam).
Tempo de geracao da solucao:0.36486291885375977
Cam = [cel(1, 1), cel(2, 1), cel(3, 1), cel(4, 1), cel(5, 1), cel(5, 2), cel(5, 3), cel(5, 4), cel(..., ...)].
```

6x6:

```
?- cria_matriz.
Numero de Colunas: 6.

Numero de Linhas: |: 6.

true .

?- dfs(cel(1,1),cel(6,6),Cam).
Cam = [cel(1, 1), cel(2, 1), cel(3, 1), cel(4, 1), cel(5, 1), cel(6, 1), cel(6, 2), cel(5, 2), cel(..., ...)|...] .

?- bfs(cel(1,1),cel(6,6),Cam).
Cam = [cel(1, 1), cel(2, 1), cel(3, 1), cel(4, 1), cel(5, 1), cel(6, 1), cel(6, 2), cel(6, 3), cel(..., ...)|...] .

?- better_dfs(cel(1,1),cel(6,6),Cam).
ERROR: Stack limit (1.0Gb) exceeded
ERROR:   Stack sizes: local: 9Kb, global: 5.6Mb, trail: 0Kb
ERROR:   Stack depth: 17, last-call: 29%, Choice points: 34
ERROR:   In:
ERROR:     [17] system:'$add_findall_bag'([length:31])
ERROR:     [16] '$bags':findall_loop([length:31], <compound (:)/2>, _1481016, [])
ERROR:     [15] system:setup_call_catcher_cleanup(<compound (:)/2>, <compound (:)/2>, _1481052, <compound (:)/2>)
ERROR:     [10] user:better_dfs(<compound cel/2>, <compound cel/2>, _1481094)
ERROR:     [9] '$toplevel':toplevel_call(<compound (:)/2>)
ERROR:
ERROR: Use the --stack_limit=size[KMG] command line option or
ERROR: ?- set_prolog_flag(stack_limit, 2_147_483_648). to double the limit.
^  Exception: (15) setup_call_catcher_cleanup('$bags':'$new_findall_bag', '$bags':findall_loop(_1480064, user:dfs(cel(1, 1), cel(6, 6), _1480064), _1480062, []), _1485540, '$bags':'$destroy_findall_bag') ? break
etter_dfs1(cel(1,1),cel(6,6),Cam).% Break level 1
[1]  ?- better_dfs1(cel(1,1),cel(6,6),Cam).
Tempo de geracao da solucao:106.41375803947449
Cam = [cel(1, 1), cel(2, 1), cel(3, 1), cel(4, 1), cel(5, 1), cel(6, 1), cel(6, 2), cel(6, 3), cel(..., ...)|...].
```

7x7:

```
[1]  ?- cria_grafo(7,7).
true .

[1]  ?- cria_matriz.
Numero de Colunas: 7.

Numero de Linhas: |: 7.

true .

[1]  ?- dfs(cel(1,1),cel(7,7),Cam).
Cam = [cel(1, 1), cel(2, 1), cel(3, 1), cel(4, 1), cel(5, 1), cel(6, 1), cel(7, 1), cel(7, 2), cel(..., ...)|...] .

[1]  ?- bfs(cel(1,1),cel(7,7),Cam).
Cam = [cel(1, 1), cel(2, 1), cel(3, 1), cel(4, 1), cel(5, 1), cel(6, 1), cel(7, 1), cel(7, 2), cel(..., ...)|...] .

[1]  ?- better_dfs(cel(1,1),cel(7,7),Cam).
ERROR: Stack limit (1.0Gb) exceeded
ERROR:   Stack sizes: local: 13Kb, global: 26.8Mb, trail: 0Kb
ERROR:   Stack depth: 31, last-call: 35%, Choice points: 47
ERROR:   In:
ERROR:     [31] system:'$add_findall_bag'([length:39])
ERROR:     [30] '$bags':findall_loop([length:39], <compound (:)/2>, _7022548, [])
ERROR:     [29] system:setup_call_catcher_cleanup(<compound (:)/2>, <compound (:)/2>, _7022584, <compound (:)/2>)
ERROR:     [24] user:better_dfs(<compound cel/2>, <compound cel/2>, _7022626)
ERROR:     [23] '$toplevel':toplevel_call(<compound (:)/2>)
ERROR:
ERROR: Use the --stack_limit=size[KMG] command line option or
ERROR: ?- set_prolog_flag(stack_limit, 2_147_483_648). to double the limit.
^  Exception: (29) setup_call_catcher_cleanup('$bags':'$new_findall_bag', '$bags':findall_loop(_7021388, user:dfs(cel(1, 1), cel(7, 7), _7021388), _7021386, []), _7027072, '$bags':'$destroy_findall_bag') ? break
etter_dfs1(cel(1,1),cel(7,7),Cam).% Break level 2
[2]  ?- better_dfs1(cel(1,1),cel(7,7),Cam).
```

A-Star (*4_us515_aStar.pl*)

```
?-
|    aStar(d105, d106, Caminho, Custo).
Caminho = [d105, c101, d102, d101, d107, d106],
Custo = 49
```

Solução entre pisos e edifícios (*4_us510_mov_robot_entre_ed_pisos.pl*)

```
?-
|    cria_grafo.
true .

?- dfs(cel(a1, 4, 3), cel(b2, 3, 3),L).
Iniciando DFS de cel(a1,4,3) para cel(b2,3,3)
Visitando: cel(a1,4,3), movendo para: cel(a1,3,3)
Visitando: cel(a1,3,3), movendo para: cel(a1,2,3)
Visitando: cel(a1,2,3), movendo para: cel(a1,1,3)
Visitando: cel(a1,1,3), movendo para: cel(a1,1,4)
Visitando: cel(a1,1,4), movendo para: cel(a1,2,4)
Visitando: cel(a1,2,4), movendo para: cel(a1,3,4)
Visitando: cel(a1,3,4), movendo para: cel(a1,4,4)
Visitando: cel(a1,1,3), movendo para: cel(a1,2,4)
Visitando: cel(a1,2,4), movendo para: cel(a1,3,4)
Visitando: cel(a1,3,4), movendo para: cel(a1,4,4)
Visitando: cel(a1,2,4), movendo para: cel(a1,1,4)
Visitando: cel(a1,1,3), movendo para: cel(a2,1,3)
Visitando: cel(a2,1,3), movendo para: cel(a2,2,3)
Visitando: cel(a2,2,3), movendo para: cel(a2,3,3)
Visitando: cel(a2,3,3), movendo para: cel(a2,4,3)
Visitando: cel(a2,4,3), movendo para: cel(a2,4,4)
?-
|    dfs(cel(c2, 4, 3), cel(c1, 3, 3),L).
Iniciando DFS de cel(c2,4,3) para cel(c1,3,3)
Visitando: cel(c2,4,3), movendo para: cel(c2,3,3)
Visitando: cel(c2,3,3), movendo para: cel(c2,2,3)
Visitando: cel(c2,2,3), movendo para: cel(c2,1,3)
Visitando: cel(c2,1,3), movendo para: cel(c2,1,4)
Visitando: cel(c2,1,4), movendo para: cel(c1,1,4)
Visitando: cel(c1,1,4), movendo para: cel(c1,2,4)
Visitando: cel(c1,2,4), movendo para: cel(c1,3,4)
Visitando: cel(c1,3,4), movendo para: cel(c1,4,4)
Visitando: cel(c1,4,4), movendo para: cel(c1,4,3)
Visitando: cel(c1,4,3), movendo para: cel(c1,3,3)
Caminho encontrado: [cel(c2,4,3),cel(c2,3,3),cel(c2,2,3),cel(c2,1,3),cel(c2,1,4),cel(c1,1,4),cel(c1,2,4),cel(c1,3,4),cel(c1,4,4),cel(c1,4,3),cel(c1,3,3)]
DFS finalizado.
L = [cel(c2, 4, 3), cel(c2, 3, 3), cel(c2, 2, 3), cel(c2, 1, 3), cel(c2, 1, 4), cel(c1, 1, 4), cel(c1, 2, 4), cel(c1, 3, 4), cel(..., ..., ...)|...]
```