Lógica para Programação

Projecto

2023-2024

A arte de escolher boas sombras (em Prolog)

Conteúdo

Uma pessoa amiga pede-te ajuda para organizar um festival de Verão: precisa de um programa que indique onde é que se devem localizar as tendas das pessoas que vão participar no evento. As regras são as seguintes: para que os participantes não fiquem todos amontoados, cada árvore deve ser associada a uma única tenda e as tendas não devem estar ao lado umas das outras.

A descrição do problema faz-te lembrar uma *app* chamada (nem de propósito) "Tendas e árvores" e decides começar por implementar um programa que resolva esses puzzles; depois não será complicado estender o programa para um cenário real. Claro que, estando a aprender o maravilhoso Prolog, decides que a implementação será em Prolog.

Assim, estudas as regras do jogo "Tendas e árvores" (Secção 1) e, apesar de ainda não teres tido a cadeira de estruturas de dados, defines a estrutura de dados que utilizarás (Secção 2). Crias ainda um programa em Prolog (primeiras linhas na Secção 3) e resolves os predicados que te permitem resolver os puzzles dos jogos (Secção 4). Sobre as condições de realização do projecto, a sua avaliação e recomendações, vê, sff, as Secções 5 e 6. Obrigada e diverte-te!

1 O puzzle "Tendas e Árvores"

É-te dado um tabuleiro (assuma-se uma matriz de dimensão NxN), que tem árvores em algumas posições. São também dados valores inteiros, indicado quantas tendas devem estar em cada linha/coluna. A Figura 1 mostra um puzzle inicial. Os valores da primeira linha e coluna indicam o número de tendas que deve estar em cada coluna e linha.

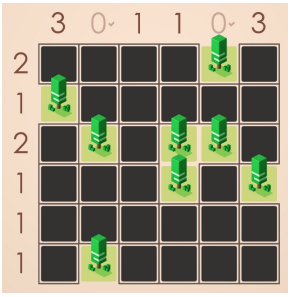

Figura 1: Exemplo de um puzzle inicial

O objectivo é colocar uma tenda na **vizinhança** – posição imediatamente acima, imediatamente abaixo, imediatamente à esquerda ou imediatamente à direita – de cada árvore, respeitando o número dado de tendas por linha e por coluna, e tendo em conta que não pode ficar uma tenda na **vizinhança alargada** de outra tenda. Isto é, após ser colocada uma tenda, não pode existir outra nem na posição imediatamente acima, nem na posição imediatamente abaixo, nem na posição imediatamente à esquerda, nem na posição imediatamente à direita, **nem nas posições diagonais**.

Nota: pode existir mais do que uma tenda na vizinhança de uma árvore; o que tem de ser garantido é que, no final, é possível atribuir uma e uma única tenda a cada árvore, estando essa tenda na sua vizinhança.

A Figura 2 mostra um puzzle quase resolvido. Nota que, por exemplo, existindo uma tenda na posição (1,1) não pode existir uma tenda nas posições (1, 2) e (2, 2) (nem na posição (2, 1) que, de qualquer modo, neste caso, já está ocupada por uma árvore).

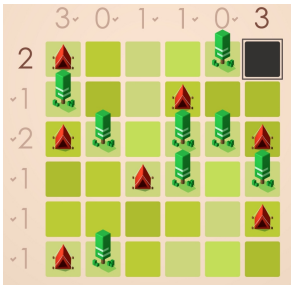

Figura 2: Exemplo de um puzzle quase resolvido

2 Estruturas de dados

Um puzzle vai ser representado por um triplo, contendo:

- Um tabuleiro, constituído por uma matriz (uma lista de listas), em que cada lista representa uma linha do puzzle;
- Uma lista que representa o número exacto de tendas que deverá existir por linha;
- Uma lista que representa o número exacto de tendas que deverá existir por coluna.

Assim, os puzzles da Figura 1 e da Figura 2 serão representados, respectivamente, do seguinte modo (“a” para representar “árvore”, “t” para representar “tenda” e “r” para representar “relva”):

```
([
[_, _, _, _, a, _],
[a, _, _, _, _, _],
[_, a, _, a, a, _],
[_, _, _, a, _, a],
[_, _, _, _, _, _],
[_, a, _, _, _, _]],
[2,1,2,1,1,1],
[3,0,1,1,0,3])

([
[t, r, r, r, a, _],
[a, r, r, t, r, r],
[t, a, r, a, a, t],
[r, r, t, a, r, a],
[r, r ,r, r, r, t],
[t, a, r, r, r, r]],
[2,1,2,1,1,1],
[3,0,1,1,0,3]).
```

Nota: no ficheiro puzzlesAcampar.pl, que te é dado, tens os puzzles que se seguem, que são os usados neste enunciado (bem como todos os exemplos apresentados neste enunciado). **Nota que não deves copiar NUNCA bocados de texto a partir deste pdf, correndo o risco de ficares com um input mal formatado (o que muitas vezes não é visível).**

```
puzzle(6-13,
([
[_, _, _, _, a, _],
[a, _, _, _, _, a],
[_, _, _, a, _, _],
[_, _, _, _, a, _],
[_, _, a, _, _, _],
[_, _, a, _, _, _]],
[2, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 2])).

puzzle(6-14,
([
[_, a, _, a, _, _],
[a, _, _, _, _, _],
[_, _, _, _, _, _],
[_, _, a, a, _, _],
[_, _, _, _, _, _],
[_, a, _, _, a, _]],
[3, 0, 1, 1, 1, 1], [2, 1, 1, 1, 2, 0])).

puzzle(8-1,
([
[_, _, _, _, a, _, a, _],
[a, _, _, _, _, _, _, a],
[_, _, _, _, _, _, _, _],
[_, a, _, _, a, _, _, _],
[_, _, _, a, _, a, a, a],
[a, _, _, _, _, _, _, _],
[_, _, a, _, _, _, _, _],
[_, _, _, a, _, _, _, _]],
[4, 0, 1, 2, 1, 3, 0, 2], [2, 0, 2, 2, 2, 2, 1, 2])).
```

3 O programa em Prolog

O teu ficheiro em Prolog (extensão pl) deverá ter as seguintes linhas iniciais:

```
% Escrever aqui o numero e o nome do aluno
:- use_module(library(clpfd)). % para poder usar transpose/2
:- set_prolog_flag(answer_write_options,[max_depth(0)]). % ver listas completas
:- [‘puzzlesAcampar.pl’]. % Ficheiro dado. No Mooshak tera mais puzzles.
% Atencao: nao deves copiar nunca os puzzles para o teu ficheiro de codigo
% Segue-se o codigo
```

4 Predicados a implementar

4.1 Consultas

Percebes que uma coisa importante a fazer é conseguir aceder rapidamente à informação contida no puzzle. Assim, decides implementar algo que te permita identificar:

- a **vizinhança** de uma dada posição de coordenadas (L, C), isto é, as coordenadas da posição imediatamente acima/abaixo/à esquerda/à direita de uma posição de coordenadas (L, C);
- a **vizinhança alargada** de uma dada posição de coordenadas (L, C), isto é, as coordenadas da vizinhança, tais como definidas anteriormente, bem como as coordenadas das diagonais (portanto, vizinhança alargada = vizinhança + diagonais);
- todas as coordenadas de um tabuleiro;
- todas as coordenadas que contêm um dado tipo de **objectos** (neste contexto, objecto é uma tenda (t), relva (r), árvore (a) ou ainda uma variável (por exemplo X), para indicar os espaços não preenchidos).

Arregaças as mangas e implementas os predicados `vizinhanca/2`, `vizinhancaAlargada/2`, `todasCelulas/2` e `todasCelulas/3`, tais que (respectivamente):

– ~~`vizinhanca`~~`((L, C), Vizinhanca)` é verdade se `Vizinhanca` é uma lista ordenada de cima para baixo e da esquerda para a direita, sem elementos repetidos, com as coordenadas das posições imediatamente acima, imediatamente à esquerda, imediatamente à direita e imediatamente abaixo da coordenada `(L, C)`;
– ~~`vizinhancaAlargada`~~`((L, C), VizinhancaAlargada)` é verdade se `VizinhancaAlargada` é uma lista ordenada de cima para baixo e da esquerda para a direita, sem elementos repetidos, com as coordenadas anteriores e ainda as diagonais da coordenada `(L, C)`;
– ~~`todasCelulas(Tabuleiro, TodasCelulas`~~`)` é verdade se `TodasCelulas` é uma lista ordenada de cima para baixo e da esquerda para a direita, sem elementos repetidos, com todas as coordenadas do ~~tabuleiro~~ `Tabuleiro`;
– ~~`todasCelulas(Tabuleiro, TodasCelulas, Objecto`~~`)` é verdade se `TodasCelulas` é uma lista ordenada de cima para baixo e da esquerda para a direita, sem elementos repetidos, com todas as coordenadas do tabuleiro `Tabuleiro` em que existe um objecto do tipo `Objecto` (neste contexto (tal como no anterior) objecto é uma tenda (t), relva (r), árvore (a) ou ainda uma variável (por exemplo X), para indicar os espaços não preenchidos).

Nota: os primeiros dois predicados poderão ter na sua lista coordenadas que podem estar fora das coordenadas do tabuleiro. Para este projecto, NÃO os retires da lista.

Por exemplo,

```
?- vizinhanca((3, 4), L).
...
L = [(2,4),(3,3),(3,5),(4,4)].

?- vizinhanca((3, 1), L).
...
L = [(2,1),(3,0),(3,2),(4,1)].

?- vizinhancaAlargada((3, 4), L).
...
L = [(2,3),(2,4),(2,5),(3,3),(3,5),(4,3),(4,4),(4,5)].

?- puzzle(6-13, (T, _, _)), todasCelulas(T, TodasCelulas).
...
TodasCelulas = [(1,1),(1,2),(1,3),(1,4),(1,5),(1,6),
(2,1),(2,2),(2,3),(2,4),(2,5),(2,6),
(3,1),(3,2),(3,3),(3,4),(3,5),(3,6),
```

```
(4,1),(4,2),(4,3),(4,4),(4,5),(4,6),
(5,1),(5,2),(5,3),(5,4),(5,5),(5,6),
(6,1),(6,2),(6,3),(6,4),(6,5),(6,6)].

?- puzzle(6-13, (T, _, _)), todasCelulas(T, TodasCelulas, a).
...
TodasCelulas = [(1,5),(2,1),(2,6),(3,4),(4,5),(5,3),(6,3)].

?- puzzle(6-13, (T, _, _)), todasCelulas(T, TodasCelulas, Z).
...
TodasCelulas = [(1,1),(1,2),(1,3),(1,4),(1,6),(2,2),(2,3),(2,4),(2,5),
(3,1),(3,2),(3,3),(3,5),(3,6),(4,1),(4,2),(4,3),(4,4),(4,6),
(5,1),(5,2),(5,4),(5,5),(5,6),(6,1),(6,2),(6,4),(6,5),(6,6)].
```

De seguida, decides que pode vir a ser útil contar os objectos de um dado tipo (incluindo variáveis) do tabuleiro, por linhas/colunas. Assim decides implementar o predicado `calculaObjectosTabuleiro/4`, tal que:

~~`calculaObjectosTabuleiro(Tabuleiro, ContagemLinhas, ContagemColunas, Objecto`~~`)` é verdade se `Tabuleiro` for um tabuleiro, `Objecto` for o tipo de objecto que se procura, e `ContagemLinhas` e `ContagemColunas` forem, respectivamente, listas com o número desses objectos por linha e por coluna.

Por exemplo (em que X e Y representam os espaços não preenchidos; podiam estar outras variáveis):

```
?- puzzle(6-13, (T, _, _)), calculaObjectosTabuleiro(T, CLinhas, CColunas, a).
...
CLinhas = [1,2,1,1,1,1],
CColunas = [1,0,2,1,2,1].
?- puzzle(6-13, (T, _, _)), calculaObjectosTabuleiro(T, CLinhas, CColunas, X).
...
CLinhas = [5,4,5,5,5,5],
CColunas = [5,6,4,5,4,5].
?- puzzle(6-13, (T, _, _)), calculaObjectosTabuleiro(T, CLinhas, CColunas, Y).
...
CLinhas = [5,4,5,5,5,5],
CColunas = [5,6,4,5,4,5].
```

Sugestão: Pensas que uma solução será implementar um predicado auxiliar que determina o número de um dado objecto numa lista; depois de o aplicar a todas as linhas do tabuleiro, pode-se usar o predicado `transpose/2` para o aplicar às colunas. Pensas que é fundamental distinguir na perfeição a utilização do "=" e do "==" em Prolog.

Para terminar esta etapa lembras-te que deve ser importante implementar um predicado que, dado um tabuleiro e uma coordenada, te diga se esta está vazia ou tem relva. Assim, decides implementar o predicado `celulaVazia/2`, definido tal como se segue:

~~`celulaVazia(Tabuleiro, (L, C))`~~ é verdade se `Tabuleiro` for um tabuleiro que não tem nada ou tem relva nas coordenadas (L, C). De notar que se as coordenadas não fizerem parte do tabuleiro, o predicado não deve falhar.

Por exemplo[1],

[1] Isto é, apenas no segundo caso é avaliado (explicitamente) para "false". Nos outros casos, não falha (mas não o "diz" explicitamente). Nota também que as variáveis serão diferentes em cada execução do predicado.

```
?- puzzle(6-13, (T, _, _)), celulaVazia(T, (1, 2)).
T = [[_17192,_17198,_17204,_17210,a,_17222],
...,
[_17402,_17408,a,_17420,_17426,_17432]].
?- puzzle(6-13, (T, _, _)), celulaVazia(T, (1, 5)).
false.
?- puzzle(6-13, (T, _, _)), celulaVazia(T, (0, 5)).
T = [[_19528,_19534,_19540,_19546,a,_19558],
...,
[_19738,_19744,a,_19756,_19762,_19768]].
?- puzzle(6-13, (T, _, _)), celulaVazia(T, (1, 7)).
T = [[_20870,_20876,_20882,_20888,a,_20900],
....
[_21080,_21086,a,_21098,_21104,_21110]].
```

4.2 Inserção de tendas e relva

E pronto, já consegues ter um conjunto de funcionalidades que te permitem descobrir coordenadas, se uma célula está vazia, etc. Mas ainda não implementaste nada que te permita inserir tendas e relva no teu tabuleiro. Mais uma vez, arregaças as mangas e voltas ao trabalho. Decides implementar os predicados `insereObjectoCelula/3` e `insereObjectoEntrePosicoes/4`, definidos como se segue (em ambos os seguintes casos, entenda-se[2] objecto como referindo uma tenda (t) ou relva (r)).

– ~~`insereObjectoCelula(Tabuleiro, TendaOuRelva, (L, C))`~~ é verdade se `Tabuleiro` é um tabuleiro e `(L, C)` são as coordenadas onde queremos inserir o objecto `TendaOuRelva`.
– ~~`insereObjectoEntrePosicoes(Tabuleiro, TendaOuRelva, (L, C1), (L, C2))`~~ é verdade se `Tabuleiro` é um tabuleiro, e `(L, C1)` e `(L, C2)` são as coordenadas, na Linha `L`, entre as quais (incluindo) se insere o objecto `TendaOuRelva`.

Por exemplo (os seguintes tabuleiros são apenas ilustrativos; podem nem ter solução),

```
?- T = [[_, _, a, _], [_, _, _, _], [a, a, a, a], [_, _, a, _]],
insereObjectoCelula(T, r, (1,1)).
T = [[r,_28482,a,_28494],
[_28506,_28512,_28518,_28524],
[a,a,a,a],
[_28566,_28572,a,_28584]].

?- T = [[_, _, a, _], [_, _, _, _], [a, a, a, a], [_, _, a, _]],
insereObjectoCelula(T, r, (1,3)).
T = [[_34140,_34146,a,_34158],
[_34170,_34176,_34182,_34188],
[a,a,a,a],
[_34230,_34236,a,_34248]].

?- T = [[_, _, a, _], [_, _, _, _], [a, a, a, a], [_, _, a, _]],
insereObjectoEntrePosicoes(T, r, (1,1), (1,4)).
T = [[r,r,a,r],
[_41134,_41140,_41146,_41152],
[a,a,a,a],
[_41194,_41200,a,_41212]].
```

[2] En**tenda**-se. Boa piada! Ahahaha

Sugestão: lembra que, mais tarde, podes usar o predicado `transpose` para inserir objectos quer nas linhas quer nas colunas de um tabuleiro, bastando, neste último caso fazer o "transpose" do tabuleiro original.

4.3 Estratégias

Chegou o momento mais esperado: implementar algumas estratégias para resolver os puzzles. Ocorre-te o seguinte[3]:

- Tens de implementar um predicado que enche de relva uma linha/coluna sempre que esta está completa (por completa entenda-se que tinha N tendas a colocar e foram colocadas as N tendas). Este predicado será o `relva/1`.
- Também faz sentido um predicado que põe relva em todas as posições inacessíveis de um tabuleiro, isto é, posições que não estão na **vizinhança** de nenhuma árvore (isto é, não estão em cima, em baixo, à esquerda ou à direita de árvore alguma). Este predicado será o `inacessiveis/1`.
- É preciso implementar um predicado que ponha as N tendas em falta numa linha/coluna quando existem exactamente N posições vazias nessa linha/coluna. Baptizas de `aproveita/1` esse predicado.
- Será certamente relevante um predicado que põe relva na **vizinhança alargada** de uma tenda, de modo a garantir que mais nenhuma tenda será aí colocada. Este predicado é o `limpaVizinhancas/1`.
- Finalmente, pensas que é igualmente importante definir um predicado que detecta quando apenas existe uma posição única que permite que uma dada árvore tenha uma tenda associada, e que coloca a tenda nessa posição. Será o `unicaHipotese/1`.

Tendo em conta, respectivamente, os predicados anteriores, implementas o seguinte:

– ~~`relva(Puzzle)`~~ é verdade se `Puzzle` é um puzzle que, após a aplicação do predicado, tem relva em todas as linhas/colunas cujo número de tendas já atingiu o número de tendas possível nessas linhas/colunas;
– ~~`inacessiveis(Tabuleiro)`~~ é verdade se `Tabuleiro` é um tabuleiro que, após a aplicação do predicado, tem relva em todas as posições inacessíveis;
– ~~`aproveita(Puzzle)`~~ é verdade se `Puzzle` é um puzzle que, após a aplicação do predicado, tem tendas em todas as linhas e colunas às quais faltavam colocar X tendas e que tinham exactamente X posições livres. De notar que este predicado deve ser implementado resolvendo as linhas, fazendo novas contagens, e resolvendo as colunas; não é recursivo;
– `limpaVizinhancas(Puzzle)` é verdade se `Puzzle` é um puzzle que, após a aplicação do predicado, tem relva em todas as posições à volta de uma tenda;
– ~~`unicaHipotese(Puzzle)`~~ é verdade se `Puzzle` é um puzzle que, após a aplicação do predicado, todas as árvores que tinham apenas uma posição livre na sua **vizinhança** que lhes permitia ficar ligadas a uma tenda, têm agora uma tenda nessa posição.

Por exemplo,

```
?- puzzle(6-14, P), relva(P).
P =  ([[_78632,a,_78644,a,_78656,r],
[a,r,r,r,r,r],
```

[3] É possível que não te ocorra nada disto ou que te ocorram coisas mais inteligentes. Mas estas funcionalidades são o que deves implementar.

```
[_78716,_78722,_78728,_78734,_78740,r],
[_78758,_78764,a,a,_78782,r],
[_78800,_78806,_78812,_78818,_78824,r],
[_78842,a,_78854,_78860,a,r]],
[3,0,1,1,1,1],[2,1,1,1,2,0]).

?- puzzle(6-14, (T, _, _)), inacessiveis(T).
T = [[_4730,a,_4742,a,_4754,r],
[a,_4778,r,_4790,r,r],
[_4814,r,_4826,_4832,r,r],
[r,_4862,a,a,_4880,r],
[r,_4904,_4910,_4916,_4922,r],
[_4940,a,_4952,_4958,a,_4970]].

?- puzzle(6-14, P), relva(P), aproveita(P).
P =  ([[t,a,t,a,t,r],
[a,r,r,r,r,r],
[_8472,_8478,_8484,_8490,_8496,r],
[_8514,_8520,a,a,_8538,r],
[_8556,_8562,_8568,_8574,_8580,r],
[_8598,a,_8610,_8616,a,r]],
[3,0,1,1,1,1],[2,1,1,1,2,0]).

?- puzzle(6-14, P), relva(P), aproveita(P), relva(P), unicaHipotese(P).
P =  ([[t,a,t,a,t,r],
[a,r,r,r,r,r],
[_6318,_6324,r,_6336,_6342,r],
[_6360,t,a,a,_6384,r],   <--- tenda em (4, 2) era a unica hipótese
[_6402,_6408,r,_6420,_6426,r],
[_6444,a,r,_6462,a,r]],
[3,0,1,1,1,1],[2,1,1,1,2,0]).

?- puzzle(6-14, P), relva(P), aproveita(P), relva(P),
unicaHipotese(P), limpaVizinhancas(P).
P =  ([[t,a,t,a,t,r],
[a,r,r,r,r,r],
[r,r,r,_20682,_20688,r],  <-- a tenda de (4, 2)
[r,t,a,a,_20730,r],       tem agora relva à volta
[r,r,r,_20766,_20772,r],
[_20790,a,r,_20808,a,r]],
[3,0,1,1,1,1],[2,1,1,1,2,0]).
```

4.4 Tentativa e Erro

Entretanto já estás eventualmente um bocadinho farto de implementar predicados e lembras-te que o Prolog faz uma coisa incrível: explora outros ramos quando o ramo em que está falha. Isto permite-te, como aprendeste nas aulas, fazer uma abordagem por tentativa e erro (que, como sabes, nem sempre é adequada para resolver alguns problemas): quando os predicados que implementaste anteriormente já não te permitem evoluir na resolução do teu tabuleiro, podes colocar uma tenda no tabuleiro, numa posição qualquer não preenchida, e voltar a chamar os teus predicados de novo; se falhar, isto é, se o número de tendas por linha/coluna não corresponder ao esperado (já implementaste predicados para isto) ou se não existir uma relação bijectiva entre árvores/tendas (o que garantirás com o predicado `valida/2` descrito

de seguida), falha. Assim, atacas os seguintes predicados:

- `valida/2`, que verifica se todas as árvores podem ser associadas a uma e a uma única tenda na sua vizinhança;
- `resolve/1`, que recebe um puzzle e devolve o puzzle resolvido.

Finalmente, implementas os seguintes predicados:

– `valida(LArv, LTen)` é verdade se `LArv` e `LTen` são listas com todas as coordenadas em que existem, respectivamente, árvores e tendas, e é avaliado para verdade se for possível estabelecer uma relação em que existe uma e uma única tenda para cada árvore nas suas vizinhanças.
– `resolve(Puzzle)` é verdade se `Puzzle` é um puzzle que, após a aplicação do predicado, fica resolvido.

Por exemplo,

```
?- valida([(1,2),(1,4),(2,1),(4,3),(4,4),(6,2),(6,5)],
[(1,1),(1,3),(1,5),(3,4),(4,2),(5,5),(6,1)]).
true.

?- valida([(1,1),(1,3)], [(1,2),(1,4)]).
true.

?- puzzle(6-14, P), resolve(P).
...
P =  ([[t,a,t,a,t,r],
[a,r,r,r,r,r],
[r,r,r,t,r,r],
[r,t,a,a,r,r],
[r,r,r,r,t,r],
[t,a,r,r,a,r]],
[3,0,1,1,1,1],
[2,1,1,1,2,0]).
```

Nota: se quiseres implementar mais estratégias antes de avançar para a tentativa e erro, estás à vontade. Nota só que tal não pode colidir com os testes de avaliação automática.

Entretanto, findo o projecto, mostras o teu trabalho à pessoa tua amiga, ligada à organização do evento. Fica encantada com o teu incrível trabalho e oferece-te 10 bilhetes para o festival de Verão e (cereja no topo do bolo) deixa-te escolher os grupos que vão participar no festival de Verão[4]. Vais imediatamente fazer a uma lista (risca o que não te interessa e/ou completa os espaços em branco): Imagine Dragons | Coldplay | Olivia Rodrigo | Xutos | Nick Cave | Twenty one Pilots | Muse | Maroon 5 | Taylor Swift | Sam Gomes[5] | Harry Styles | Billie Eilish | Adele | Anastacia | Guano Apes | _____ | _____ | _____ | _____ | _____.

[4] Imagina o teu cartaz de sonho. Sonhar não faz mal.
[5] `https://www.youtube.com/@samuelgomesmusic590`

5 Entrega e avaliação

5.1 Condições de realização e prazos

O projecto é realizado individualmente. O código do projecto deve ser entregue obrigatoriamente por via electrónica até às **23h59 de dia 8 de janeiro 2024**, através do sistema Mooshak. Depois desta hora, não serão aceites projectos sob pretexto algum[6]. A ter em conta:

- Cada aluno deverá submeter um ficheiro .pl contendo o seu código. O ficheiro de código deve conter em comentário, na primeira linha, o número e o nome do aluno;
- Não devem ser utilizados caracteres acentuados ou qualquer caractere que não pertença à tabela ASCII, mesmo em comentários;
- Não esquecer de remover/comentar as mensagens escritas no ecrã;
- A avaliação da execução do código do projecto será feita automaticamente através do sistema Mooshak, usando vários testes configurados no sistema. O tempo de execução de cada teste está limitado, bem como a memória utilizada. Tipicamente, só se poderá efectuar uma nova submissão 15 minutos depois da submissão anterior[7]. Só são permitidas 10 submissões em simultâneo no sistema, pelo que uma submissão poderá ser recusada se este limite for excedido. Nesse caso tentar mais tarde;
- Os testes considerados para efeitos de avaliação podem incluir ou não os exemplos disponibilizados, além de um conjunto de testes adicionais.

Serão publicadas na página da cadeira as instruções necessárias para a submissão do código no Mooshak e a partir dessa altura será possível a submissão por via electrónica. Até ao prazo de entrega poderá efectuar o número de entregas que desejar (por favor, não usar o Mooshak para debug), sendo utilizada para efeitos de avaliação a última entrega efectuada.

Pode ou não haver uma discussão oral do projecto e/ou uma demonstração do funcionamento do programa (será decidido caso a caso).

5.2 Cotação

A nota do projecto será baseada no seguinte:

- Execução correcta – 16 valores distribuídos da seguinte forma:
  1. Consultas (5.5 valores)
     (a) `vizinhança`: 0.75 valores;
     (b) `vizinhancaAlargada`: 0.75 valores;
     (c) `todasCelulas/2`: 1.0 valor;
     (d) `todasCelulas/3`: 1.0 valor;
     (e) `calculaObjectosTabuleiro`: 1.0 valor;

[6] Note que o limite de 10 submissões simultâneas no sistema Mooshak implica que, caso haja um número elevado de tentativas de submissão sobre o prazo de entrega, alguns alunos poderão ver-se impossibilitados de submeter o código dentro do prazo.

[7] De notar que, se for feita uma submissão no Mooshak a menos de 15 minutos do prazo de entrega, o aluno fica impossibilitado de efectuar qualquer outra submissão posterior.

   (f) `celulaVazia`: 1.0 valor;
2. Insere (2.0 valores)
   (a) `insereObjectoCelula`: 1 valor
   (b) `insereObjectoEntrePosicoes`: 1 valor;
3. Estratégias (6.0 valores)
   (a) `relva`: 1.2 valores;
   (b) `inacessiveis`: 1.2 valores;
   (c) `aproveita`: 1.2 valores;
   (d) `limpaVizinhancas`: 1.2 valores;
   (e) `unicaHipotese`: 1.2 valores;
4. Tentativa e Erro (2.5 valores)
   (a) `valida`: 1.0 valor.
   (b) `resolve`: 1.5 valor.

- Estilo de programação e facilidade de leitura – 4 valores assim distribuídos:
  - Comentários (1.0 valor): deverão ser incluídos comentários para o utilizador (descrição sumária do predicado); deverão também ser incluídos, quando se justifique, comentários para o programador.
  - Boas práticas (3.0 valores):
    * Integração de conhecimento adquirido durante a UC (1.0 valor).
    * Implementação de predicados não excessivamente longos. O facto de usar um predicado recursivo, desde que bem feito, não é penalizado em relação ao uso de predicados funcionais; no entanto, uma má abstração procedimental e duplicações de código serão penalizados (1.0 valor).
    * Escolha de nomes dos predicados auxiliares e das variáveis (1.0 valor).

Presença de *warnings* serão penalizadas (-2 valores).

5.3 Sobre as cópias

Os alunos deverão fazer o seu próprio código. Se por acaso tirarem partido do ChatGPT ou outro *Large Language Model* (LLM) qualquer, para resolver um ou mais predicados, deverão preencher um formulário (detalhes em breve) a explicar como decorreram as interações. Projectos muito semelhantes levarão à reprovação na disciplina e levantamento de processo disciplinar. Caso dois alunos tenham usado um LLM, não tenham preenchido o formulário e os projectos forem considerados muito semelhantes, serão considerados cópia. O corpo docente da disciplina será o único juiz do que se considera copiar.

6 Recomendações

- Usem o SWI PROLOG, que vai ser usado para a avaliação do projecto.
- Durante o desenvolvimento do programa não se esqueçam da Lei de Murphy:
  - Todos os problemas são mais difíceis do que parecem;
  - Tudo demora mais tempo do que pensamos;
  - Se alguma coisa puder correr mal, vai correr mal, na pior das alturas possíveis.