TÉCNICO LISBOA

Fundamentos da Programação

Ano lectivo 2015-16

Segundo Projeto

Picross

O jogo *Picross* é um quebra-cabeças sobre uma imagem binária, em que as células de uma grelha devem ser coloridas ou deixadas em branco de acordo com os números na parte lateral e superior da grelha (*especificação das linhas e das colunas*, respetivamente) para revelar um quadro escondido. Neste tipo de quebra-cabeças, os números são uma forma de tomografia discreta que mede quantas linhas inteiras de quadrados preenchidos existem em qualquer linha ou coluna. Por exemplo, uma especificação de *6* 3 1 significa que existem conjuntos contíguos de seis, três e um quadrados preenchidos, por essa ordem, com pelo menos um quadrado branco entre grupos sucessivos.

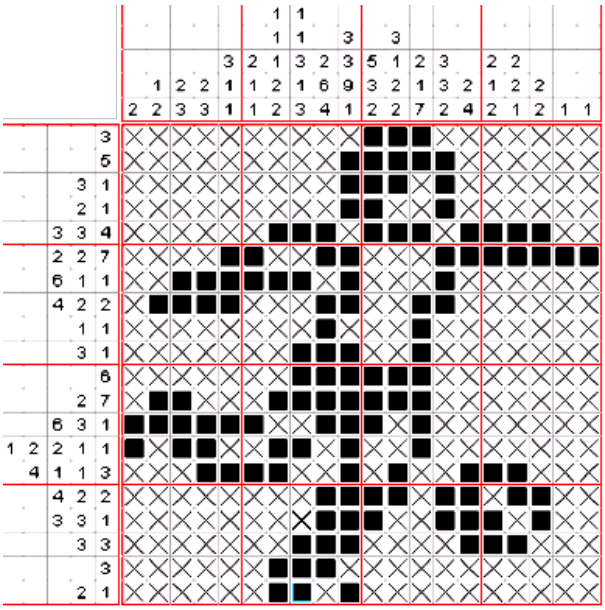


Figura 1: Exemplo de um tabuleiro Picross 20x20 resolvido.

O objetivo do jogo é descobrir a imagem descrita, identificando as células preenchidas e as células em branco da grelha de acordo com as especificações das linhas e colunas. Por exemplo, na Fig. 1 a célula (1 : 1) (célula no canto superior esquerdo do quadro) está em *branco* e a célula (1 : 11) está preenchida, correspondendo a uma *caixa*.

Para resolver um enigma, é preciso determinar quais as células que devem ficar em branco e as que devem ser preenchidas. Determinar quais as células em branco é tão importante como determinar quais as células a preencher. Durante o processo de resolução, as células em branco ajudam a determinar onde se situam os blocos contínuos de caixas. Apenas as células que podem ser determinadas pela lógica devem ser preenchidas, pois um erro fruto de uma suposição pode arruinar completamente a solução. O quadro escondido não desempenha nenhum papel no processo de resolução; a imagem, no entanto, pode ajudar a encontrar e eliminar um erro.

Os quebra-cabeças mais simples geralmente podem ser resolvidos por resolução de cada linha (ou coluna) isoladamente, sendo apenas necessário a identificação do conteúdo das células dessa linha (ou coluna). A resolução do quebra-cabeças resume-se assim ao conjunto das resoluções individuais de cada linha e coluna.

A resolução de alguns quebra-cabeças mais difíceis exigem a consideração dos vários cenários possíveis, procurando-se identificar alguma contradição: quando uma célula não pode ser uma caixa, porque outra célula iria produzir um erro, ela será definitivamente uma célula em branco. E vice versa.

1 Trabalho a Realizar

O objetivo do projeto é escrever um programa em Python que permita a um utilizador jogar o jogo *Picross* no computador. Para tal, deverá definir um conjunto de tipos de informação que deverão ser utilizados para manipular a informação necessária ao decorrer do jogo, bem como um conjunto de funções adicionais que permitirão jogar o jogo propriamente dito.

1.1 Tipos Abstratos de Dados (TAD)

TAD *coordenada* (2 val.)

O TAD *coordenada* será utilizado para indexar as várias células do tabuleiro. Cada célula do tabuleiro é indexada através da linha (um inteiro entre 1 e o número de linhas do tabuleiro) e da coluna respetiva (um inteiro entre 1 e o número de colunas do tabuleiro), em que a célula $(1 : 1)$ corresponde ao canto superior esquerdo do tabuleiro.

O TAD *coordenada* deverá pois ser um tipo **imutável** que armazena dois inteiros correspondentes a uma linha e uma coluna do tabuleiro.

As operações básicas associadas a este TAD são:

- *cria_coordenada* : $int \times int \rightarrow coordenada$

  Esta função corresponde ao construtor do tipo *coordenada*. Recebe dois argumentos do tipo inteiro, o primeiro dos quais corresponde a uma linha l (um

inteiro positivo) e o segundo a uma coluna c (um inteiro positivo), e deve devolver um elemento do tipo *coordenada* correspondente à célula $(l : c)$. A função deve verificar a validade dos seus argumentos, gerando um `ValueError` com a mensagem `'cria_coordenada: argumentos invalidos'` caso algum dos argumentos introduzidos seja inválido.

- *coordenada_linha* : *coordenada* → *inteiro*

  Este seletor recebe como argumento um elemento do tipo *coordenada* e devolve a linha respetiva.

- *coordenada_coluna* : *coordenada* → *inteiro*

  Este seletor recebe como argumento um elemento do tipo *coordenada* e devolve a coluna respetiva.

- *e_coordenada* : *universal* → *lógico*

  Este reconhecedor recebe um único argumento e devolve `True` caso esse argumento seja do tipo *coordenada*, e `False` em caso contrário.

- *coordenadas_iguais* : *coordenada* × *coordenada* → *lógico*

  Este teste recebe como argumentos dois elementos do tipo *coordenada* e devolve `True` caso esses argumentos correspondam à mesma célula do tabuleiro, e `False` em caso contrário.

- *coordenada_para_cadeia* : *coordenada* → *cad. caracteres*

  Esta função recebe como argumento um elemento do tipo *coordenada* e devolve uma cadeia de caracteres que a represente de acordo com o exemplo em baixo.

Exemplo de interação:

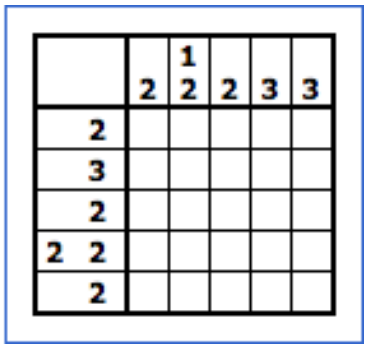
```
>>> C1 = cria_coordenada(-1, 1)
[...]
builtins.ValueError: cria_coordenada: argumentos invalidos
>>> C1 = cria_coordenada(0, 0)
[...]
builtins.ValueError: cria_coordenada: argumentos invalidos
>>> C1 = cria_coordenada(1, 2)
>>> coordenada_linha(C1)
1
>>> coordenada_coluna(C1)
2
>>> e_coordenada(0)
False
>>> e_coordenada(C1)
True
>>> C2 = cria_coordenada(1, 2)
>>> coordenadas_iguais(C1, C2)
True
>>> C3 = cria_coordenada(2, 1)
```

```
>>> coordenadas_iguais(C1, C3)
False
>>> coordenada_para_cadeia(C3)
"(2 : 1)"
```

TAD *tabuleiro* (6 val.)

O TAD *tabuleiro* será utilizado para representar o tabuleiro. Este TAD deverá permitir:

- (i) representar um tabuleiro de $Picross$ (uma grelha de $n \times n$ células);
- (ii) aceder a cada uma das células do tabuleiro;
- (iii) modificar o conteúdo de cada uma das células.



(a) Tabuleiro inicial.

Figura 2: Exemplo de tabuleiro inicial.

A Fig. 2 apresenta um exemplo de um tabuleiro 5×5, em que 2 na primeira linha estabelece o número de células que devem estar preenchidas consecutivamente nessa linha, e em que 3 na última coluna estabelece o número de células que devem estar preenchidas consecutivamente nessa coluna. A especificação $2\ 2$ na 4ª linha, estabelece que a 4ª linha tem duas caixas, seguidas de pelo menos uma célula em branco, seguidas de mais duas caixas.

As operações básicas associadas a este TAD são:

- *cria_tabuleiro* : *tuplo* $\rightarrow$ *tabuleiro*

  Esta função corresponde ao construtor do tipo *tabuleiro*. Recebe como argumento um elemento t do tipo *tuplo* descrevendo a especificação das linhas e das colunas do tabuleiro, e deverá devolver um elemento do tipo tabuleiro de acordo com a representação interna escolhida. O tuplo t deve ser composto por dois tuplos de tuplos de inteiros, em que o primeiro corresponde à especificação das linhas e o segundo à especificação das colunas. Por exemplo, o tuplo

$(((2,),(3,),(2,),(2,2),(2,)),((2,),(1,2),(2,),(3,),(3,)))$

contém a especificação para o jogo representado na Fig. 2.

O tabuleiro criado deverá estar vazio, isto é, todas as células deverão conter o valor 0, representando as células do quadro vazias (como desconhecidas).

A função deve verificar a validade dos seus argumentos, gerando um `ValueError` com a mensagem `'cria_tabuleiro: argumentos invalidos'` caso o argumento introduzido seja inválido.

- *tabuleiro_dimensoes* : *tabuleiro* → *tuplo*

  Este seletor recebe como argumento um elemento t do tipo *tabuleiro* e devolve um *tuplo* com dois elementos, cujo primeiro elemento é o número de linhas do tabuleiro e o segundo o número de colunas do mesmo.

- *tabuleiro_especificacoes* : *tabuleiro* → *tuplo*

  Este seletor recebe como argumento um elemento t do tipo *tabuleiro* e devolve um *tuplo* composto por dois tuplos de tuplos de inteiros, cujo primeiro elemento corresponde à especificação das linhas e o segundo à especificação das colunas.

- *tabuleiro_celula* : *tabuleiro* × *coordenada* → *{0, 1, 2}*

  Este seletor recebe como argumentos um elemento t do tipo *tabuleiro* e um elemento c do tipo *coordenada* e devolve um elemento do tipo *inteiro* entre 0 e 2, que corresponde ao valor contido na célula do tabuleiro referente à coordenada c. Caso a célula correspondente a c esteja vazia, deverá devolver o valor 0, caso corresponda a uma célula em branco deve devolver o valor 1 e caso esteja preenchida deve devolver o valor 2. A função deve verificar a validade dos argumentos para o tabuleiro em causa, e gerar um `ValueError` com a mensagem `'tabuleiro_celula: argumentos invalidos'` caso não se verifique.

- *tabuleiro_preenche_celula* : *tabuleiro* × *coordenada* × *{0, 1, 2}* → *tabuleiro*

  Este modificador recebe como argumentos um elemento t do tipo *tabuleiro*, um elemento c do tipo *coordenada* e um *inteiro* e entre 0 e 2, e **modifica** o tabuleiro t, preenchendo a célula referente à coordenada c com o elemento e, que pode ser 0, 1 ou 2, para representar o vazio, uma célula em branco ou uma caixa, respetivamente. A função deve devolver o tabuleiro modificado. Deve ainda verificar a validade dos argumentos e gerar um `ValueError` com a mensagem `'tabuleiro_preenche_celula: argumentos invalidos'` caso algum dos argumentos introduzidos não seja válido.

- *e_tabuleiro* : *universal* → *lógico*

  Este reconhecedor recebe um único argumento, devendo devolver `True` se o seu argumento for do tipo *tabuleiro*, e `False` em caso contrário.

- *tabuleiro_completo* : *tabuleiro* → *lógico*

Este reconhecedor recebe como argumento um elemento t do tipo *tabuleiro* e devolve `True` caso o tabuleiro t esteja totalmente preenchido corretamente de acordo com as suas especificações, e `False` em caso contrário.

- *tabuleiros_iguais* : *tabuleiro* $\times$ *tabuleiro* $\rightarrow$ *lógico*

  Este teste recebe como argumentos dois elementos t_1 e t_2 do tipo *tabuleiro* e devolve `True` caso t_1 e t_2 correspondam a dois tabuleiros com as mesmas especificações e quadros com o mesmo conteúdo, e `False` em caso contrário.

Deve ainda implementar o seguinte transformador de saída:

- *escreve_tabuleiro* : *tabuleiro* $\rightarrow$ {}

  A função *escreve_tabuleiro* recebe como argumento um elemento t do tipo *tabuleiro* e escreve para o ecrã a representação externa de um tabuleiro de Picross, apresentada no exemplo abaixo. Deve ainda verificar se t é um tabuleiro válido e, caso o argumento introduzido seja inválido, deve gerar um `ValueError` com a mensagem `'escreve_tabuleiro: argumento invalido'`.

Exemplo de interação:

```
>>> e=(((2,), (3,), (2,), (2, 2), (2,)), ((2,), (1, 2), (2,), (3,), (3,)))
>>> t = cria_tabuleiro(e)
>>> tabuleiro_dimensoes(t)
(5, 5)
>>> tabuleiro_especificacoes(t)
(((2,), (3,), (2,), (2, 2), (2,)), ((2,), (1, 2), (2,), (3,), (3,)))
>>> escreve_tabuleiro(t)
       1
  2    2    2    3    3
[ ? ][ ? ][ ? ][ ? ][ ? ] 2  |
[ ? ][ ? ][ ? ][ ? ][ ? ] 3  |
[ ? ][ ? ][ ? ][ ? ][ ? ] 2  |
[ ? ][ ? ][ ? ][ ? ][ ? ] 2 2|
[ ? ][ ? ][ ? ][ ? ][ ? ] 2  |

>>> t1 = tabuleiro_preenche_celula(t, cria_coordenada(4, 2), 2)
>>> escreve_tabuleiro(t)
       1
  2    2    2    3    3
[ ? ][ ? ][ ? ][ ? ][ ? ] 2  |
[ ? ][ ? ][ ? ][ ? ][ ? ] 3  |
[ ? ][ ? ][ ? ][ ? ][ ? ] 2  |
[ ? ][ x ][ ? ][ ? ][ ? ] 2 2|
[ ? ][ ? ][ ? ][ ? ][ ? ] 2  |

>>> tabuleiros_iguais(t, t1)
True
>>> t2 = tabuleiro_preenche_celula(t, cria_coordenada(4, 3), 1)
>>> escreve_tabuleiro(t)
       1
```

```
  2    2    2    3    3
[ ? ][ ? ][ ? ][ ? ][ ? ] 2  |
[ ? ][ ? ][ ? ][ ? ][ ? ] 3  |
[ ? ][ ? ][ ? ][ ? ][ ? ] 2  |
[ ? ][ x ][ . ][ ? ][ ? ] 2 2|
[ ? ][ ? ][ ? ][ ? ][ ? ] 2  |

>>> tabuleiros_iguais(t, t1)
False
>>> tabuleiros_iguais(t, t2)
True
>>> tabuleiro_celula(t, 0)
[...]
builtins.ValueError: tabuleiro_celula: argumentos invalidos
>>> tabuleiro_celula(t, cria_coordenada(4, 1))
0
>>> tabuleiro_celula(t, cria_coordenada(4, 2))
2
>>> tabuleiro_celula(t, cria_coordenada(4, 3))
1
>>> e_tabuleiro(t)
True
>>> e_tabuleiro("a")
False
>>> tabuleiro_completo(t)
False
>>> t2 = tabuleiro_preenche_celula(t, cria_coordenada(1, 1), 1)
>>> t2 = tabuleiro_preenche_celula(t, cria_coordenada(1, 2), 2)
>>> t2 = tabuleiro_preenche_celula(t, cria_coordenada(1, 3), 2)
>>> t2 = tabuleiro_preenche_celula(t, cria_coordenada(1, 4), 1)
>>> t2 = tabuleiro_preenche_celula(t, cria_coordenada(1, 5), 1)
>>> t2 = tabuleiro_preenche_celula(t, cria_coordenada(2, 1), 1)
>>> t2 = tabuleiro_preenche_celula(t, cria_coordenada(2, 2), 1)
>>> t2 = tabuleiro_preenche_celula(t, cria_coordenada(2, 3), 2)
>>> t2 = tabuleiro_preenche_celula(t, cria_coordenada(2, 4), 2)
>>> t2 = tabuleiro_preenche_celula(t, cria_coordenada(2, 5), 2)
>>> t2 = tabuleiro_preenche_celula(t, cria_coordenada(3, 1), 1)
>>> t2 = tabuleiro_preenche_celula(t, cria_coordenada(3, 2), 1)
>>> t2 = tabuleiro_preenche_celula(t, cria_coordenada(3, 3), 1)
>>> t2 = tabuleiro_preenche_celula(t, cria_coordenada(3, 4), 2)
>>> t2 = tabuleiro_preenche_celula(t, cria_coordenada(3, 5), 2)
>>> t2 = tabuleiro_preenche_celula(t, cria_coordenada(4, 1), 2)
>>> t2 = tabuleiro_preenche_celula(t, cria_coordenada(4, 4), 2)
>>> t2 = tabuleiro_preenche_celula(t, cria_coordenada(4, 5), 2)
>>> t2 = tabuleiro_preenche_celula(t, cria_coordenada(5, 1), 2)
>>> t2 = tabuleiro_preenche_celula(t, cria_coordenada(5, 2), 2)
>>> t2 = tabuleiro_preenche_celula(t, cria_coordenada(5, 3), 1)
>>> t2 = tabuleiro_preenche_celula(t, cria_coordenada(5, 4), 1)
>>> t2 = tabuleiro_preenche_celula(t, cria_coordenada(5, 5), 1)
>>>escreve_tabuleiro(t)
      1
  2    2    2    3    3
```

```
[ . ][ x ][ x ][ . ][ . ] 2  |
[ . ][ . ][ x ][ x ][ x ] 3  |
[ . ][ . ][ . ][ x ][ x ] 2  |
[ x ][ x ][ . ][ x ][ x ] 2 2|
[ x ][ x ][ . ][ . ][ . ] 2  |

>>> tabuleiro_completo(t)
True
```

TAD *jogada* (1 val.)

O TAD *jogada* será utilizado para representar a jogada a efetuar sobre um tabuleiro. Cada jogada é composta por uma coordenada e um valor igual a 1 ou 2, que representa o conteúdo de uma célula de um tabuleiro de picross (uma célula em branco ou uma caixa, respetivamente).

As operações básicas associadas a este TAD são:

- *cria_jogada* : *coordenada* $\times \{1, 2\} \rightarrow$ *jogada*

  Esta função corresponde ao construtor do tipo *jogada*. Recebe como argumento um elemento do tipo *coordenada* e um *inteiro* com valor 1 ou 2. A função deve verificar a validade dos seus argumentos, gerando um `ValueError` com a mensagem `'cria_jogada: argumentos invalidos'` caso algum dos argumentos introduzidos seja inválido.

- *jogada_coordenada* : *jogada* $\rightarrow$ *coordenada*

  Este seletor recebe como argumento um elemento do tipo *jogada* e devolve a coordenada respetiva.

- *jogada_valor* : *jogada* $\rightarrow \{1, 2\}$

  Este seletor recebe como argumento um elemento do tipo *jogada* e devolve o valor respetivo.

- *e_jogada* : *universal* $\rightarrow$ *lógico*

  Este reconhecedor recebe um único argumento e devolve `True` caso esse argumento seja do tipo *jogada*, e `False` em caso contrário.

- *jogadas_iguais* : *jogada* $\times$ *jogada* $\rightarrow$ *lógico*

  Este teste recebe como argumentos dois elementos do tipo *jogada* e devolve `True` caso esses argumentos correspondam à mesma jogada, e `False` caso contrário.

- *jogada_para_cadeia* : *jogada* $\rightarrow$ *cad. caracteres*

  Esta função recebe como argumento um elemento do tipo *jogada* e devolve uma cadeia de caracteres que a represente, de acordo com o exemplo em baixo.

Exemplo de interação:

```
>>> J = cria_jogada(cria_coordenada(1, 1), 4)
[...]
builtins.ValueError: cria_jogada: argumentos invalidos
>>> J = cria_jogada(cria_coordenada(1, 1), 2)
>>> jogada_para_cadeia(J)
"(1 : 1) --> 2"
>>> coordenada_para_cadeia(jogada_coordenada(J))
"(1 : 1)"
>>> jogada_valor(J)
2
>>> e_jogada(0)
False
>>> e_jogada(J)
True
>>> J2 = cria_jogada(cria_coordenada(1, 1), 1)
>>> jogadas_iguais(J, J2)
False
>>>  J2 = cria_jogada(cria_coordenada(1, 1), 2)
>>> jogadas_iguais(J, J2)
True
```

1.2 Funções Adicionais

- *le_tabuleiro* : *cad. caracteres* → *tuplo* **(1 val.)**

  Esta função recebe uma *cadeia de caracteres* que corresponde ao nome do ficheiro com os dados de especificação do jogo, e devolve um *tuplo* de dois tuplos com a especificação das linhas e colunas, respetivamente. Este tuplo pode ser usado para criar um tabuleiro, como no exemplo que se segue. A função deve conseguir ler ficheiros que contenham as especificações referidas, semelhantes aos ficheiros de exemplo fornecidos.

  Exemplo de interação:

```
>>> le_tabuleiro("jogo_fig2.txt")
(((2,), (3,), (2,), (2, 2), (2,)), ((2,), (1, 2), (2,), (3,), (3,)))
>>> T = cria_tabuleiro(le_tabuleiro("jogo_fig2.txt"))
>>> escreve_tabuleiro(T)
       1
  2    2    2    3    3
[ ? ][ ? ][ ? ][ ? ][ ? ] 2  |
[ ? ][ ? ][ ? ][ ? ][ ? ] 3  |
[ ? ][ ? ][ ? ][ ? ][ ? ] 2  |
[ ? ][ ? ][ ? ][ ? ][ ? ] 2 2|
[ ? ][ ? ][ ? ][ ? ][ ? ] 2  |

>>>
```

- *pede_jogada* : *tabuleiro* → *jogada* **(2 val.)**

  Esta função recebe o tabuleiro do jogo como argumento e devolve a jogada que o jogador pretende executar.

  A função deve começar por pedir a coordenada da célula a preencher c, seguida do valor v, de acordo com o exemplo de interação a seguir. A cadeia de caracteres introduzida para representar a coordenada c deve seguir o mesmo formato que a função $coordenada\_para\_cadeia$. A função devolve o resultado de invocar a função `cria_jogada(c, v)` sempre que possível, e o valor $False$ caso o utilizador não introduza uma coordenada válida para o tabuleiro em causa.

  No caso em que o utilizador não introduz uma cadeia de caracteres possível de ser transformada numa coordenada e num inteiro entre 1 e 2, o comportamento da função não está definido.

  Exemplo:

```
>>> T = cria_tabuleiro(le_tabuleiro("jogo_fig2.txt"))
>>> J = pede_jogada(T)
Introduza uma jogada
- coordenada entre (1 : 1) e (5 : 5)) >> (1 : 5)
- valor >> 2
>>> jogada_para_cadeia(J)
"(1 : 5) --> 2"
>>> pede_jogada(T)
Introduza uma jogada
- coordenada entre (1 : 1) e (5 : 5)) >> (6 : 6)
- valor >> 2
False
>>> J = pede_jogada(T)
Introduza uma jogada
- coordenada entre (1 : 1) e (5 : 5)) >> (5 : 5)
- valor >> 2
>>> jogada_para_cadeia(J)
"(5 : 5) --> 2"
>>>
```

- *jogo_picross*: *cad. caracteres* → *lógico* **(2 val.)**

  Esta função corresponde à função principal do jogo e permite a um utilizador jogar um jogo completo de $Picross$. Recebe como argumento uma *cadeia de caracteres* representando o nome do ficheiro com a especificação do tabuleiro, e devolve $True$ caso o tabuleiro resultante do jogo esteja $completo$ (quadro completo e de acordo com as especificações) e $False$ caso contrário.

  Após cada jogada, a função deve desenhar o tabuleiro resultante no ecrã e pedir ao utilizador uma nova jogada. Caso a jogada seja válida, deverá atualizar o tabuleiro e repetir este processo até o jogo terminar. Caso contrário, deverá escrever para o ecrã a indicação "`Jogada invalida.`" e solicitar nova jogada ao utilizador.

O jogo termina quando o tabuleiro já não tiver células vazias. Note que o jogador pode alterar o conteúdo de uma célula mais do que uma vez, mas só o pode fazer enquanto existirem células vazias no tabuleiro. Quando todas as células têm valor conhecido (branco ou caixa), ou seja já não existem células vazias, o jogo termina.

No Anexo encontra dois exemplos de execução completa da função $jogo\_picross$.

Sugestão: Poderá ainda ser útil implementar duas funções adicionais, nomeadamente `tabuleiro_celulas_vazias` e `linha_completa`. A primeira recebe como argumento um elemento do tipo tabuleiro e devolve uma lista com as coordenadas das células do tabuleiro que estão vazias. A segunda recebe como argumento um tuplo com a especificação de uma linha ou coluna, e uma lista com os conteúdos das células de uma linha / coluna, e verifica se a linha / coluna em questão satisfaz a especificação recebida. Estas sugestões, no entanto, não são obrigatórias.

1.3 Sugestões

1. Leia o enunciado completo, procurando perceber o objetivo das várias funções pedidas. Em caso de dúvida de interpretação, utilize o horário de dúvidas para esclarecer a sua questão.
2. No processo de desenvolvimento do projeto, comece por implementar os vários TADs pela ordem apresentada no enunciado, seguindo a metodologia respetiva. Em particular, comece por escolher uma representação interna antes de começar a implementar as operações básicas. Só depois deverá desenvolver as funções do jogo, seguindo também a ordem pela qual foram apresentadas no enunciado. Ao desenvolver cada uma das funções pedidas, comece por perceber se pode usar alguma das anteriores.
3. Para verificar a funcionalidade das suas funções, utilize os exemplos fornecidos como casos de teste.
4. Tenha o cuidado de reproduzir fielmente as mensagens de erro e restantes *outputs*, conforme ilustrado nos vários exemplos.

2 Aspetos a Evitar

Os seguintes aspetos correspondem a sugestões para evitar maus hábitos de trabalho (e, consequentemente, más notas no projeto):

1. Não pense que o projeto se pode fazer nos últimos dias. Se apenas iniciar o seu trabalho neste período irá ver a Lei de Murphy em funcionamento (todos os problemas são mais difíceis do que parecem; tudo demora mais tempo do que nós pensamos; e se alguma coisa puder correr mal, ela vai correr mal, na pior das alturas possíveis).

2. Não pense que um dos elementos do seu grupo fará o trabalho por todos. Este é um trabalho de grupo e deverá ser feito em estreita colaboração (comunicação e controle) entre os vários elementos do grupo, cada um dos quais com as suas responsabilidades. Tanto uma possível oral como perguntas nos testes sobre o projeto, servem para despistar estas situações.
3. *Não duplique código*. Se duas funções são muito semelhantes é natural que estas possam ser fundidas numa única, eventualmente com mais argumentos.
4. Não se esqueça que as funções excessivamente grandes são penalizadas no que respeita ao estilo de programação.
5. A atitude "vou pôr agora o programa a correr de qualquer maneira e depois preocupo-me com o estilo" é totalmente errada.
6. Quando o programa gerar um erro, preocupe-se em descobrir qual a causa do erro. As "marteladas" no código têm o efeito de distorcer cada vez mais o código.

3 Classificação

A avaliação da execução será feita através do sistema *Mooshak*. Tal como na primeira parte do projeto, existem vários testes configurados no sistema. O tempo de execução de cada teste está limitado, bem como a memória utilizada. Só poderá efetuar uma nova submissão pelo menos 15 minutos depois da submissão anterior. Só são permitidas 10 submissões em simultâneo no sistema, pelo que uma submissão poderá ser recusada se este limite for excedido. Nesse caso tente mais tarde.

Os testes considerados para efeitos de avaliação podem incluir ou não os exemplos disponibilizados, além de um conjunto de testes adicionais. O facto de um projeto completar com sucesso os exemplos fornecidos não implica, pois, que esse projeto esteja totalmente correto, pois o conjunto de exemplos fornecido não é exaustivo. É da responsabilidade de cada grupo garantir que o código produzido está correto.

Não será disponibilizado qualquer tipo de informação sobre os casos de teste utilizados pelo sistema de avaliação automática. Os ficheiros de teste usados na avaliação do projeto serão disponibilizados na página da disciplina após a data de entrega.

A nota do projeto será baseada nos seguintes aspetos:

1. Execução correta (70%).

   Esta parte da avaliação é feita recorrendo ao sistema *Mooshak* que sugere uma nota face aos vários aspetos considerados.

2. Estilo de programação e facilidade de leitura, nomeadamente a abstração procedimental, a abstração de dados, nomes bem escolhidos, qualidade (e não quantidade) dos comentários e tamanho das funções (30%). Os seus comentários deverão incluir, entre outros, uma descrição da representação interna adotada em

cada um dos TAIs definidos.

4 Condições de Realização e Prazos

A entrega do 2º projeto será efetuada exclusivamente por via eletrónica. Deverá submeter o seu projeto através do sistema *Mooshak*, até às **23:59 do dia 9 de Dezembro de 2015**. Depois desta hora, não serão aceites projetos sob pretexto algum.[1]

Deverá submeter um único ficheiro com extensão `.py` contendo todo o código do seu projeto. O ficheiro de código deve conter em comentário, na primeira linha, os números e os nomes dos alunos do grupo, bem como o número do grupo.

No seu ficheiro de código não devem ser utilizados caracteres acentuados ou qualquer carácter que não pertença à tabela ASCII. Isto inclui comentários e cadeias de caracteres. Programas que não cumpram este requisito serão penalizados em três valores.

Duas semanas antes do prazo, serão publicadas na página da cadeira as instruções necessárias para a submissão do código no Mooshak. *Apenas a partir dessa altura será possível a submissão por via eletrónica. Nessa altura serão também fornecidas a cada um as necessárias credenciais de acesso. Até ao prazo de entrega poderá efetuar o número de entregas que desejar, sendo utilizada para efeitos de avaliação a última entrega efetuada. Deverá portanto verificar cuidadosamente que a última entrega realizada corresponde à versão do projeto que pretende que seja avaliada. Não serão abertas exceções.*

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

Projetos iguais, ou muito semelhantes, serão penalizados com a reprovação na disciplina. O corpo docente da cadeira será o único juiz do que se considera ou não copiar num projeto.

[1] 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 grupos poderão não conseguir submeter nessa altura e verem-se, por isso, impossibilitados de submeter o código dentro do prazo.