INSTITUTO SUPERIOR TÉCNICO

Lógica para Programação

Projecto

8 de Abril de 2017

Solucionador de problemas de Sudoku

O objetivo deste projecto é escrever um programa em PROLOG para resolver problemas de Sudoku.

Um problema de Sudoku de dimensão n (em que n é um quadrado perfeito) é uma grelha $n \times n$, em que cada posição tem um número entre 1 e n ou está vazia. Na Fig. 1, apresenta-se um exemplo de um problema de dimensão 9.

Uma grelha de Sudoku encontra-se dividida em linhas, colunas e blocos, todos numerados de 1 a n. Na Fig. 2 mostra-se esta numeração para um problema de dimensão 9.

Designa-se por grupo de posições, ou simplesmente grupo, o conjunto de posições correspondentes a uma linha, coluna ou bloco. O objectivo é preencher as posições vazias com números entre 1 e n, de modo a que cada grupo contenha todos os números entre 1 e n.

1 Abordagem

Nesta secção apresentamos o algoritmo que o seu programa deve usar na resolução de problemas de Sudoku, daqui em diante designados simplesmente por *puzzles*.

1.1 Inicialização

Quando um puzzle é inicializado, as posições que se encontravam vazias no puzzle original passam, no puzzle inicializado, a conter sequências ordenadas e sem elementos repetidos dos números possíveis. Note que um número é possível para uma posição se esse número não ocorrer numa sequência unitária na mesma linha, coluna ou bloco. De cada vez que é colocada uma sequência unitária numa posição, essa mudança deve ser propagada ao resto do puzzle (ver Secção 1.2).

Consideremos a Fig. 3. Nesta figura, parte a), está representado o puzzle inicial. Na parte b) mostra-se a inicialização deste puzzle até à posição $(3, 3)$.

	3				1			
		6					5	
5						9	8	3
	8				6	3		2
				5				
9		3	8				6	
7	1	4						9
	2					8		
			4				3	

Figura 1: Problema de dimensão 9.

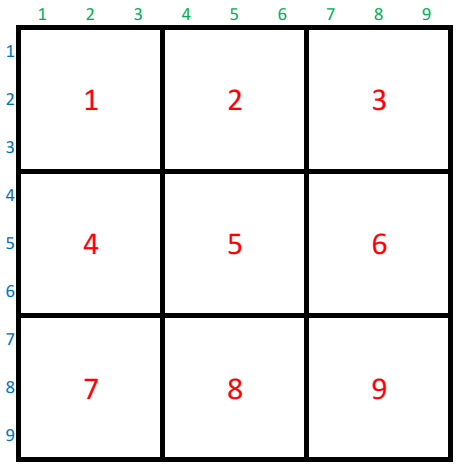

Figura 2: Numeração de linhas (azul), colunas (verde) e blocos (vermelho).

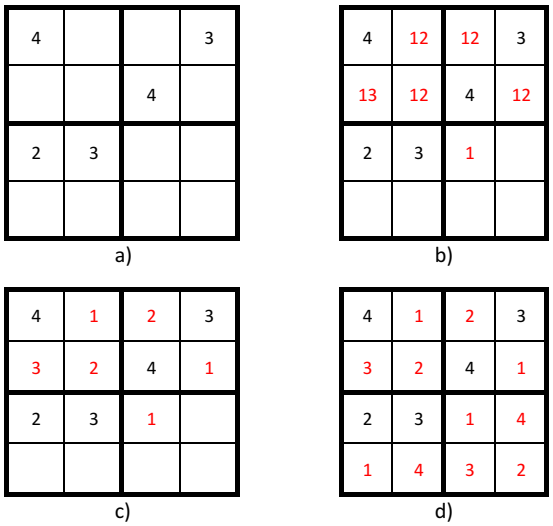

Figura 3: Inicialização de um puzzle de dimensão 4.

1.2 Propagação de mudanças

De cada vez que é colocada uma sequência unitária numa posição, o número dessa sequência deve ser retirado das sequências de todas as outras posições da mesma linha, coluna ou bloco. Se, em consequência, alguma sequência se tornar unitária, esta mudança deverá ser propagada da mesma forma.

Para exemplificar o que foi dito, consideremos a Fig. 3. Nesta figura, parte a), está representado o puzzle inicial. Na parte b) mostra-se a inicialização até à posição $(3, 3)$. A colocação nesta posição da sequência unitária contendo apenas o número 1 faz com que este número seja retirado da posição $(1, 3)$, ficando esta posição com a sequência contendo apenas o número 2.

A propagação desta nova mudança faz com que a posição $(2, 4)$ fique com a sequência contendo apenas o número 1 e que a posição $(1, 2)$ fique com a sequência contendo apenas o número 1.

A mudança da posição $(1, 2)$ faz com que a posição $(2, 1)$ fique com a sequência contendo apenas o número 3 e que a posição $(2, 2)$ fique com a sequência contendo apenas o número 2.

O resultado destas sucessivas propagações está representado na parte c). O resultado final da inicialização está representado na parte d).

Na Fig. 4 mostra-se a inicialização do puzzle correspondente ao problema da Fig. 1.

1.3 Inspecção de linhas, colunas e blocos

Após o puzzle inicializado, as suas linhas, colunas e blocos devem ser percorridos, verificando se existe algum número que apenas ocorre numa das posições de uma linha,

248	3	2789	25679	246789	1	2467	47	467
1248	479	6	2379	234789	234789	1247	5	147
5	47	127	267	2467	247	9	8	3
14	8	157	179	1479	6	3	1479	2
1246	467	127	12379	5	23479	147	1479	1478
9	457	3	8	1247	247	1457	6	1457
7	1	4	356	368	358	56	2	9
36	2	59	135679	13679	3579	8	147	14567
68	569	589	4	126789	25789	1567	3	1567

Figura 4: Resultado da inicialização do puzzle da Fig. 1.

coluna ou bloco. Se tal acontecer, a sequência correspondente deve passar a ter apenas esse número, e esta mudança deve ser propagada, tal como durante a inicialização.

Por exemplo, se inspeccionarmos a linha 3 do puzzle da Fig. 4, verificamos que o número 1 só ocorre na coluna 3. Isto faz com que a posição $(3, 3)$ fique apenas com o número 1. A propagação desta mudança faz com que os conteúdos das posições $(4, 3)$ e $(5, 3)$ sejam alterados. Como nenhuma destas posições fica com uma sequência unitária, nada mais há a propagar. Assim, o puzzle seria actualizado como se mostra na Fig. 5.

1.4 Obtenção de uma solução

Com os processos anteriores é possível obter, em alguns casos, a solução do problema. No entanto, pode acontecer que tal não seja suficiente. Nesta situação, é necessário fazer uma procura. Neste contexto, uma procura consiste em escolher uma posição que tem uma sequência não unitária, atribuir-lhe um dos números dessa sequência e propagar os seus efeitos. Este processo deve ser repetido até ser encontrada uma solução, isto é, até cada linha, coluna e bloco conter todos os números, de 1 a n. Caso não se encontre uma solução (o que significa que o número escolhido não se pode encontrar naquela posição), a procura deverá retroceder até à última escolha efectuada e optar por outro número.

2 Representação de posições e puzzles

Uma posição é representada por um par `(L,C)`, em que `L` representa a linha, e `C` representa a coluna.

Um puzzle de dimensão n é representado por uma lista de n listas de n elementos, em que cada uma das n listas representa uma linha do puzzle. Cada elemento é por sua vez uma lista de números. Na representação de um problema inicial esta lista pode ser vazia, ou conter um número entre 1 e n. Uma lista vazia indica que o contéudo da posição

248	3	2789	25679	246789	1	2467	47	467
248	479	6	2379	234789	234789	1247	5	147
5	47	1	267	2467	247	9	8	3
14	8	57	179	1479	6	3	1479	2
1246	467	27	12379	5	23479	147	1479	1478
9	457	3	8	1247	247	1457	6	1457
7	1	4	356	368	358	56	2	9
36	2	59	135679	13679	3579	8	147	14567
68	569	589	4	126789	25789	1567	3	1567

Figura 5: Puzzle após a inspecção da linha 3.

correspondente é desconhecido. Por exemplo, o puzzle da Fig. 1 é representado por

```
[[[ ],[3],[ ],[ ],[ ],[1],[ ],[ ],[ ]],
 [[ ],[ ],[6],[ ],[ ],[ ],[ ],[5],[ ]],
 [[5],[ ],[ ],[ ],[ ],[ ],[9],[8],[3]],
 [[ ],[8],[ ],[ ],[ ],[6],[3],[ ],[2]],
 [[ ],[ ],[ ],[ ],[5],[ ],[ ],[ ],[ ]],
 [[9],[ ],[3],[8],[ ],[ ],[ ],[6],[ ]],
 [[7],[1],[4],[ ],[ ],[ ],[ ],[ ],[9]],
 [[ ],[2],[ ],[ ],[ ],[ ],[8],[ ],[ ]],
 [[ ],[ ],[ ],[4],[ ],[ ],[ ],[3],[ ]]]
```

A partir do momento em que o puzzle é inicializado, as listas interiores deixam de poder ser vazias. Por exemplo, o puzzle da Fig. 4 é representado por

```
[[[2,4,8],[3],[2,7,8,9],[2,5,6,7,9],[2,4,6,7,8,9],[1],[2,4,6,7],[4,7],[4,6,7]],
[[1,2,4,8],[4,7,9],[6],[2,3,7,9],[2,3,4,7,8,9],[2,3,4,7,8,9],[1,2,4,7],[5],[1,4,7]],
[[5],[4,7],[1,2,7],[2,6,7],[2,4,6,7],[2,4,7],[9],[8],[3]],
[[1,4],[8],[1,5,7],[1,7,9],[1,4,7,9],[6],[3],[1,4,7,9],[2]],
[[1,2,4,6],[4,6,7],[1,2,7],[1,2,3,7,9],[5],[2,3,4,7,9],[1,4,7],[1,4,7,9],[1,4,7,8]],
[[9],[4,5,7],[3],[8],[1,2,4,7],[2,4,7],[1,4,5,7],[6],[1,4,5,7]],
[[7],[1],[4],[3,5,6],[3,6,8],[3,5,8],[5,6],[2],[9]],
[[3,6],[2],[5,9],[1,3,5,6,7,9],[1,3,6,7,9],[3,5,7,9],[8],[1,4,7],[1,4,5,6,7]],
[[6,8],[5,6,9],[5,8,9],[4],[1,2,6,7,8,9],[2,5,7,8,9],[1,5,6,7],[3],[1,5,6,7]]]
```

3 Trabalho a desenvolver

Nesta secção são descritos os predicados que deve desenvolver no seu projecto. Estes serão os predicados avaliados e, consequentemente, devem respeitar escrupulosamente as especificações apresentadas. Para além dos predicados descritos, poderá desenvolver todos os predicados que julgar necessários.

Na implementação dos seus predicados, poderá usar os predicados fornecidos no ficheiro `SUDOKU.pl`. Estes predicados são descritos na Secção 4.

As caixas de cor verde contêm sugestões para a implementação de alguns predicados. Poderá, ou não, seguir as sugestões dadas.

3.1 Predicados para a propagação de mudanças

Recorde-se que propagar uma mudança significa que, de cada vez que é colocada uma sequência unitária numa posição, o número dessa sequência deve ser retirado das sequências de todas as outras posições da mesma linha, coluna ou bloco. Se, em consequência, alguma sequência se tornar unitária, esta mudança deverá ser propagada da mesma forma.

Esta propagação é conseguida através de 2 predicados, `tira_num/4` e `puzzle_muda_propaga/4`, que se invocam mutuamente.

3.1.1 Predicado `tira_num/4`

Antes de implementar o predicado `tira_num/4`, deverá implementar o predicado auxiliar `tira_num_aux/4`:

`tira_num_aux(Num,Puz,Pos,N_Puz)` significa que `N_Puz` é o puzzle resultante de tirar o número `Num` da posição `Pos` do puzzle `Puz`.

Nota: se a sequência da posição `Pos` não contiver o número `Num`, nada é alterado.

Sugestão: utilize os predicados `puzzle_ref/3` descrito na Secção 4, e `puzzle_muda/4` descrito na próxima secção.

`tira_num(Num,Puz,Posicoes,N_Puz)` significa que `N_Puz` é o puzzle resultante de tirar o número `Num` de todas as posições em `Posicoes` do puzzle `Puz`.

Sugestão: use o predicado `percorre_muda_Puz/4` descrito na Secção 4. Este predicado permite-lhe aplicar `tira_num_aux/4` a uma lista de posições, que é exactamente o que faz o predicado `tira_num`.

3.1.2 Predicado `puzzle_muda_propaga/4`

`puzzle_muda_propaga(Puz, Pos, Cont, N_Puz)` faz o mesmo que o predicado `puzzle_muda/4`, mas, no caso de `Cont` ser uma lista unitária, propaga a mudança, isto é, retira o número em `Cont` de todas as posições relacionadas com `Pos`, isto é, todas as posições na mesma linha, coluna ou bloco.

Sugestão: utilize os predicados `puzzle_ref/3` , `puzzle_muda/4` e `posicoes_relacionadas/2` descritos na Secção 4.

Usando o exemplo da Fig. 3, temos:

```
?- Puz = [[[4],[1,2],[1,2],[3]],
          [[1,3],[1,2],[4],[1,2]],
          [[2],[3],[],[]],
          [[],[],[],[]]],
   puzzle_muda_propaga(Puz,(3,3),[1],N_Puz).
Puz = [[[4], [1, 2], [1, 2], [3]], ...,
N_Puz = [[[4], [1], [2], [3]], [[3], [2], [4], [1]],
         [[2], [3], [1], []], [[], [], [], []]].
```

Todos os predicados que se seguem devem usar `puzzle_muda_propaga` para mudar o conteúdo de posições de puzzles. Fica assim garantida a propagação automática de mudanças.

3.2 Predicados para a inicialização de puzzles

3.2.1 Predicado `possibilidades/3`

Este predicado será utilizado na inicialização de puzzles, para determinar os números possíveis para as posições do puzzle que não contêm sequências unitárias. Recorda-se que um número é possível para uma posição se esse número não ocorrer numa sequência unitária na mesma linha, coluna ou bloco.

`possibilidades(Pos,Puz,Poss)` significa que `Poss` é a lista de números possíveis para a posição `Pos`, do puzzle `Puz`. Nota: este predicado apenas deve ser usado para posições cujo conteúdo não é uma sequência unitária.

Por exemplo, sendo `Puz` o puzzle da Fig. 1:

```
?- ..., possibilidades((4,5),Puz,Poss).
Puz = [[[], [3], [], [], [], [1], [], []|...], ...
Poss = [1, 4, 7, 9].
```

Sugestão: Use os predicados `numeros/1` e `conteudos_posicoes/3`, descritos na Secção 4.

3.2.2 Predicado `inicializa/2`

Este predicado efectua a inicialização de puzzles (ver Secção 1.1).

Antes de implementar este predicado, deverá implementar o predicado auxiliar `inicializa_aux/3`, que inicializa uma posição de um puzzle:

`inicializa_aux(Puz,Pos,N_Puz)` significa que `N_Puz` é o puzzle resultante de colocar na posição `Pos` do puzzle `Puz` a lista com os números possíveis para essa posição. Note que, se o conteúdo da posição `Pos` de `Puz` já for uma lista unitária, nada é alterado.

Implemente agora o predicado `inicializa/2`:

`inicializa(Puz,N_Puz)` significa que `N_Puz` é o puzzle resultante de inicializar o puzzle `Puz`.

Assim, `inicializa/2` vai aplicar `inicializa_aux/3` a todas as posições do puzzle.

Sugestão: utilize os predicados `todas_posicoes/1` e `percorre_muda_Puz/4`, descritos na Secção 4.

3.3 Predicados para a inspecção de puzzles

3.3.1 Predicado `so_aparece_uma_vez/4`

Este predicado é usado na inspecção de puzzles, para verificar se existe algum número que apenas ocorre numa das posições de uma linha, coluna ou bloco.

`so_aparece_uma_vez(Puz,Num,Posicoes,Pos_Num)` significa que o número `Num` só aparece numa das posições da lista `Posicoes` do puzzle `Puz`, e que essa posição é `Pos_Num`.

Por exemplo, sendo `Puz` o puzzle da Fig. 4 e `Posicoes` a lista das posições da linha 3:

```
?- ..., so_aparece_uma_vez(Puz,1,Posicoes,Pos_Num).
...
Pos_Num = (3, 3).

?- ..., so_aparece_uma_vez(Puz,2,Posicoes,Pos_Num).
false.
```

3.3.2 Predicado `inspecciona/2`

Este predicado efectua a inspecção de puzzles (ver Secção 1.3). Assim, este predicado terá de verificar para cada grupo (linha, coluna ou bloco), e para cada número, se esse número só ocorre numa das posições desse grupo.

Antes de implementar este predicado, deverá implementar os predicados auxiliares `inspecciona_num/4` e `inspecciona_grupo/4`:

`inspecciona_num(Posicoes,Puz,Num,N_Puz)` significa que `N_Puz` é o resultado de inspeccionar o grupo cujas posições são `Posicoes`, para o número `Num`:

- se `Num` só ocorrer numa das posições de `Posicoes` e se o conteúdo dessa posição não for uma lista unitária, esse conteúdo é mudado para `[Num]` e esta mudança é propagada;
- caso contrário, `Puz = N_Puz`.

Por exemplo, sendo `Puz` o puzzle da Fig. 4 e `Posicoes` a lista das posições da linha 3:

- após a execução de `?- ..., inspecciona_num(Posicoes,Puz,1,N_Puz).`, `N_Puz` seria o puzzle da Fig. 5.
- após a execução de `?- ..., inspecciona_num(Posicoes,Puz,9,N_Puz).`, `N_Puz` seria igual a `Puz`.

`inspecciona_grupo(Puz,Gr,N_Puz)` inspecciona o grupo cujas posições são as da lista `Gr`, do puzzle `Puz` para cada um dos números possíveis, sendo o resultado o puzzle `N_Puz`.

`inspecciona(Puz,N_Puz)` inspecciona cada um dos grupos do puzzle `Puz`, para cada um dos números possíveis, sendo o resultado o puzzle `N_Puz`.

Sugestão: utilize os predicados `grupos/1`, `numeros/1` e `percorre_muda_Puz/4`, descritos na Secção 4.

3.4 Predicados para a verificação de soluções

`solucao(Puz)` significa que o puzzle `Puz` é uma solução, isto é, que todos os seus grupos contêm todos os números possíveis, sem repetições.

Antes de implementar este predicado, deverá implementar o predicado auxiliar `grupo_correcto/3`:

`grupo_correcto(Puz,Nums,Gr)`, em que `Puz` é um puzzle, significa que o grupo de `Puz` cujas posições são as da lista `Gr` está correcto, isto é, que contém todos os números da lista `Nums`, sem repetições.

Sugestão: use o predicado `conteudos_posicoes/3`, descrito na Secção 4.

Implemente agora o predicado `solucao/1`:

`solucao(Puz)` significa que o puzzle `Puz` é uma solução, isto é, que todos os seus grupos contêm todos os números possíveis, sem repetições.

Sugestão: utilize os predicados `grupos/1` e `numeros/1`, descritos na Secção 4.

3.5 Predicado `resolve/2`

Este é o predicado principal que, dado um puzzle, nos permite obter (um)a solução, se ela existir.

`resolve(Puz,Sol)` significa que o puzzle `Sol` é (um)a solução do puzzle `Puz`. Na obtenção da solução, deve ser utilizado o algoritmo apresentado na Secção 1: inicializar o puzzle, inspeccionar linhas, colunas e blocos, e só então procurar uma solução, tal como descrito na Secção 1.4.

Sugestão: Defina e implemente um predicado que, dado um puzzle, escolhe uma posição cujo conteúdo é uma lista não unitária.

4 Predicados fornecidos

No ficheiro `SUDOKU` encontra-se implementado um conjunto de predicados que poderão ser-lhe úteis na resolução do projecto. Nesta secção descrevem-se esses predicados.

- `dimensao(9)` - poderá alterar esta definição para considerar outras dimensões para os puzzles. Por exemplo, durante o desenvolvimento do código poderá considerar puzzles de dimensão 4, mais fáceis de testar.
- `numeros(L)` - `L` é a lista `[1,2,3,4,...,Dim]`, em que `Dim` é a dimensao do puzzle.
- `puzzle_ref(Puz, Pos, Cont)` - o conteúdo da posição `Pos` do puzzle `Puz` é `Cont`.
- `puzzle_muda(Puz, Pos, Cont, N_Puz)` - `N_Puz` é o puzzle resultante de mudar o conteúdo da posição `Pos` do puzzle `Puz` para `Cont`.
- `posicoes_relacionadas(Pos,Posicoes)` - `Posicoes` é a lista de posições relacionadas com a posicao `Pos`, isto é, posições na mesma linha, coluna ou bloco.
- `grupos(Gr)` - `Gr` é uma lista de listas: cada lista contem as posições de uma linha, coluna ou bloco, isto é, de um grupo.
- `todas_posicoes(Todas_Posicoes)` - `Todas_Posicoes` é uma lista com todas as posições do puzzle, ordenadas por linhas.
- `conteudos_posicoes(Puz,Posicoes,Conteudos)` - `Conteudos` é a lista com os conteudos do puzzle `Puz`, nas posições da lista `Posicoes`. Por exemplo,

```
?- Puz = [[[1,3],[3]],[[4,5],[3,7]]],Posicoes = [(2,1),(2,2)],
   conteudos_posicoes(Puz,Posicoes,Conteudos).
```

```
Puz = [[[1, 3], [3]], [[4, 5], [3, 7]]],
Posicoes = [ (2, 1), (2, 2)],
Conteudos = [[4, 5], [3, 7]].
```

- `percorre_muda_Puz(Puz,Accao,Lst,N_Puz)`: `Puz` é um puzzle; `Accao` é um predicado de argumentos `(Puz, <elemento de Lst>, N_Puz)`; `N_Puz` é o puzzle resultante de aplicar `Accao(Puz, <elemento de Lst>, N_Puz)` a cada elemento de `Lst`. No caso de `Accao` ter mais argumentos, para além de `(Puz, <elemento de Lst>, N_Puz)` a chamada a `percorre_muda_Puz` deve ser da forma `percorre_muda_Puz(Puz,Accao(Args),Lst,N_Puz)`.

  Para exemplificar a utilização deste predicado consideremos o predicado `junta_num_posicao/4`:[1]
  `junta_numero_posicao(Num,Puz,Pos,N_Puz)` significa que `N_Puz` é o puzzle resultante de juntar (no fim) o número `Num` ao conteúdo da posição `Pos` do puzzle `Puz`. Por exemplo, sendo `Puz` o puzzle da Fig. 3 a), teríamos:

```
?- ..., junta_numero_posicao(100,Puz,(1,1),N_Puz).
Puz = [[[4], [], [], [3]], [[], [], [4], []],
       [[2], [3], [], []], [[], [], [], []]],
N_Puz = [[[4, 100], [], [], [3]], [[], [], [4], []],
         [[2], [3], [], []], [[], [], [], []]].
```

  Suponhamos agora que pretendíamos juntar um número a várias posições de um puzzle. Para tal definamos o predicado `junta_numero_posicoes/4`.
  `junta_numero_posicoes(Num,Puz,Lista_posicoes,N_Puz)` significa que `N_Puz` é o puzzle resultante de juntar (no fim) o número `Num` ao conteúdo de cada uma das posições da lista `Lista_posicoes` do puzzle `Puz`. Note-se que este predicado corresponde a aplicar o predicado anterior várias vezes; uma vez por cada posição da lista `Lista_posicoes`. Então:

```
junta_numero_posicoes(Num,Puz,Posicoes,N_Puz) :-
    percorre_muda_Puz(Puz,junta_numero_posicao(Num),
                      Posicoes,N_Puz).
```

5 Avaliação

A nota do projecto será baseada nos seguintes aspectos:

- Execução correcta (80% - 16 val.). Estes 16 valores serão distribuídos da seguinte forma:

[1] No ficheiro `SUDOKU.pl` encontra-se a implementação deste e doutros exemplos.

`tira_num_aux`	1.0 val.
`tira_num`	1.0 val.
`puzzle_muda_propaga`	1.0 val.
`possibilidades`	1.0 val.
`inicializa_aux`	1.0 val.
`inicializa`	1.0 val.
`so_aparece_uma_vez`	2.0 val.
`inspecciona_num`	1.0 val.
`inspecciona_grupo`	1.0 val.
`inspecciona`	1.0 val.
`grupo_correcto`	1.0 val.
`solucao`	1.0 val.
`resolve`	3.0 val.

- Qualidade do código, a qual inclui abstracção relativa aos predicados implementados, nomes escolhidos, paragrafação e qualidade dos comentários (20% - 4 val.).

6 Penalizações

- Caracteres acentuados, cedilhas e outros semelhantes: 3 val.
- Presença de *warnings*: 2 val.

7 Condições de realização e prazos

O projecto deve ser realizado individualmente.

O código do projecto deve ser entregue obrigatoriamente por via electrónica até às **23:59 do dia 12 de Maio** de 2017, através do sistema Mooshak. Depois desta hora, não serão aceites projectos sob pretexto algum.[2]

Deverá ser submetido um ficheiro .pl contendo o código do seu projecto. O ficheiro de código deve conter em comentário, na primeira linha, o número e o nome do aluno. **Não deve incluir o ficheiro disponibilizado pelo corpo docente, pois este será carregado na execução.**

No seu ficheiro de código não devem ser utilizados caracteres acentuados ou qualquer caracter que não pertença à tabela ASCII, sob pena de falhar todos os testes automáticos. Isto inclui comentários e cadeias de caracteres.

É prática comum a escrita de mensagens para o ecrã, quando se está a implementar e a testar o código. Isto é ainda mais importante quando se estão a testar/comparar os algoritmos. No entanto, **não se esqueçam de remover/comentar as mensagens escritas no ecrã na versão final** do código entregue. Se não o fizerem, correm o risco dos testes automáticos falharem, e irão ter uma má nota na execução.

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

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. Só poderá efectuar uma nova submissão pelo menos 15 minutos depois da submissão anterior.[3] 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 projecto completar com sucesso os exemplos fornecidos não implica, pois, que esse projecto esteja totalmente correcto, pois o conjunto de exemplos fornecido não é exaustivo. É da responsabilidade de cada aluno garantir que o código produzido está correcto.

Duas semanas antes do prazo da entrega, 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 electrónica. Até ao prazo de entrega poderá efectuar o número de entregas que desejar, sendo utilizada para efeitos de avaliação a última entrega efectuada. Deverão portanto verificar cuidadosamente que a última entrega realizada corresponde à versão do projecto que pretendem que seja avaliada. Não serão abertas excepções.

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

8 Cópias

Projectos iguais, ou muito semelhantes, originarão a reprovação na disciplina e, eventualmente, o levantamento de um processo disciplinar. Os programas entregues serão testados em relação às várias soluções existentes na web. As analogias encontradas com os programas da web serão tratadas como cópias. O corpo docente da disciplina será o único juiz do que se considera ou não copiar num projecto.

9 Recomendações

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

[3] Note que, se efectuar uma submissão no Mooshak a menos de 15 minutos do prazo de entrega, fica impossibilitado de efectuar qualquer outra submissão posterior.