INSTITUTO SUPERIOR TÉCNICO

Lógica para Programação

Projecto

12 de Abril de 2019

Solucionador de Puzzles Binários

Conteúdo

O objetivo deste projecto é escrever um programa em PROLOG para resolver puzzles binários.

Um puzzle binário é uma matriz $n \times n$, em que n é um inteiro par, e $n \geq 4$. A n chamamos a dimensão do puzzle. Cada posição pode estar preenchida com 0 ou 1, ou estar vazia.

Uma solução para um puzzle Puz é uma matriz da mesma dimensão de Puz, em que todas as posições vazias de Puz foram preenchidas com 0 ou 1, de forma a respeitar as seguintes regras:

Regra 1 Não existem três zeros ou três uns seguidos, em nenhuma linha ou coluna.

Regra 2 Todas as linhas e colunas têm o mesmo número de zeros e uns. Este número será metade da dimensão do puzzle.

Regra 3 Não existem linhas repetidas, nem colunas repetidas.

Na Figura 1, apresenta-se um puzzle de dimensão 6.

		0			1
					1
				0	0
	1				
0		0			

Figura 1: Puzzle de dimensão 6.

1 Abordagem

Nesta secção apresentamos informalmente o algoritmo que o seu programa deve usar na resolução de puzzles binários.

1.1 Inicialização

O primeiro passo na resolução de um puzzle binário consiste na sua inicialização. Em alguns casos, este passo é suficiente para resolver o puzzle.

Para inicializar um puzzle devemos aplicar as regras 1 e 2, a cada linha e coluna, até não serem preenchidas novas posições.

Na Fig. 2 mostra-se a inicialização do puzzle da Figura 1.

		0			1
					1
					0
1	0	1	1	0	0
	1		0		1
0	1	0	1	1	0

Figura 2: Resultado da inicialização do puzzle da Figura 1.

Após a inicialização, deve ser verificado se o resultado não viola a regra 3 , isto é, se todas as linhas são diferentes umas das outras, e o mesmo para as colunas. Se o puzzle inicializado violar a regra 3, então o puzzle inicial é impossível.

1.2 Teste de posições e propagação de mudanças

Se após a inicialização de um puzzle restarem posições por preencher, é escolhida uma destas posições (i, j), e essa posição é testada com um valor $v \in \{0, 1\}$. Isto significa que o valor da posição (i, j) passa a ser v. Em seguida é necessário propagar esta mudança à linha i e à coluna j, ou seja, aplicar as regras 1 e 2 à linha i e à coluna j. Se forem alteradas novas posições, estas regras devem também ser aplicadas às linhas e colunas das posições alteradas, e assim sucessivamente.

Enquanto existirem posições por preencher é escolhida uma posição e testada com um valor. Se a escolha de um valor para uma posição provocar a violação de alguma das regras, é feita uma escolha diferente. Se já tiverem sido testados todos os valores para essa posição, retrocede-se até à última posição onde foi feita uma escolha. Note que o mecanismo de retrocesso do PROLOG faz precisamente isto.

Para exemplificar o que foi dito consideremos o puzzle da Figura 2. Suponhamos que era escolhida a posição $(1, 1)$ para testar e que esta posição era testada com o valor 0. Como alterámos a posição $(1, 1)$, temos de aplicar as regras 1 e 2 à linha 1 e à coluna 1. A aplicação destas regras faz com que a posição $(1, 2)$ seja alterada para 1. Em consequência, temos de aplicar as regras 1 e 2 à linha 1 e à coluna 2. A aplicação destas regras faz com que as posições $(2, 2)$ e $(3, 2)$ sejam alteradas para 0, o que leva a uma contradição, pois a coluna 2 passa a ter três zeros seguidos. Esta situação está representada na Figura 3.

0	1	0			1
	0				1
	0				0
1	0	1	1	0	0
	1		0		1
0	1	0	1	1	0

Figura 3: Resultado do teste da posição $(1, 1)$ com o valor 0 e da propagação da mudança.

Como o teste da posição $(1, 1)$ com o valor 0 levou a uma contradição, este valor e todos os que resultaram da sua propagação são removidos, voltando à situação representada na Figura 2.

Vamos agora testar a posição $(1, 1)$ com o valor 1. A propagação desta mudança à linha 1 e à coluna 1 não provoca qualquer alteração. A situação resultante encontra-se representada na Figura 4.

1		0			1
					1
					0
1	0	1	1	0	0
	1		0		1
0	1	0	1	1	0

Figura 4: Resultado do teste da posição $(1, 1)$ com o valor 1 e da propagação da mudança.

Suponhamos que é agora escolhida a posição $(1, 2)$ para ser testada com o valor 0. O resultado desta mudança e da sua propagação está representado na Figura 5.

1	0	0	1	0	1
			0		1
			0	1	0
1	0	1	1	0	0
	1		0		1
0	1	0	1	1	0

Figura 5: Resultado do teste da posição $(1, 2)$ com o valor 0 e da propagação da mudança.

De cada vez que é feita uma escolha e esta escolha é propagada, deve verificar-se se o resultado não viola a regra 3, isto é, se todas as linhas são diferentes umas das outras, e o mesmo para as colunas.

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 variável, se a posição correspondente do puzzle não estiver preenchida,
- o valor da posição correspondente do puzzle, se esta estiver preenchida.

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

```
[[_, _, 0, _, _, 1],
 [_, _, _, _, _, 1],
 [_, _, _, _, _, _],
 [_, _, _, _, 0, 0],
 [_, 1, _, _, _, _],
 [0, _, 0, _, _, _]]
```

3 Trabalho a desenvolver

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

Na implementação dos seus predicados, poderá usar os predicados fornecidos no ficheiro `codigo_comum.pl`. Para tal, deve colocar o comando `:- consult(codigo_comum).` no início do ficheiro que contém o seu projecto.

3.1 Predicados para a inicialização de puzzles

Nesta secção são definidos os predicados necessários para a inicialização de um puzzle. Recorde-se que, para inicializar um puzzle, devemos aplicar as regras 1 e 2, a cada linha e coluna, até não serem preenchidas novas posições. Assim, começamos por definir os predicados que permitem a aplicação das regras 1 e 2 a linhas e colunas.

3.1.1 Predicado `aplica_R1_triplo/2`

Implemente o predicado `aplica_R1_triplo/2`, tal que:

`aplica_R1_triplo(Triplo, N_Triplo)`, em que `Triplo` é uma lista de 3 elementos, em que cada elemento é 0, 1, ou uma variável, significa que `N_Triplo` é a lista resultante de aplicar a regra 1 ao triplo `Triplo`. Isto significa que se `Triplo` tiver dois zeros/uns e uma variável, esta deve ser preenchida com um/zero. Se o `Triplo` tiver três zeros (uns), o predicado deve devolver `false`.

Por exemplo,

```
?- aplica_R1_triplo([1,_,1],R).
R = [1, 0, 1].

?- aplica_R1_triplo([1,1,0],R).
R = [1, 1, 0].

?- aplica_R1_triplo([_,0,0],R).
R = [1, 0, 0].

?- aplica_R1_triplo([1,_,0],R).
R = [1, _G396, 0].

?- aplica_R1_triplo([0,0,0],R).
false.
```

3.1.2 Predicado `aplica_R1_fila_aux/2`

Implemente o predicado `aplica_R1_fila_aux/2`, tal que:

`aplica_R1_fila_aux(Fila, N_Fila)`, em que `Fila` é uma fila (linha ou coluna) de um puzzle, significa que `N_Fila` é a fila resultante de aplicar a regra 1 à fila `Fila`, uma só vez. Isto significa que `Fila` deve ser percorrida uma vez, do início para o fim, aplicando o predicado `aplica_R1_triplo/2` a cada sub-fila de 3 elementos. Se a `Fila` tiver três zeros (uns) seguidos, o predicado deve devolver `false`.

Por exemplo,

```
?- Fila = [1,_,0,_,1,0], aplica_R1_fila_aux(Fila, N_Fila).
Fila = N_Fila, N_Fila = [1, _310, 0, _322, 1, 0].

?- Fila = [0,_,0,1,1,_], aplica_R1_fila_aux(Fila, N_Fila).
Fila = [0, _310, 0, 1, 1, _334],
N_Fila = [0, 1, 0, 1, 1, 0].

?- Fila = [_,_,0,_,1,1], aplica_R1_fila_aux(Fila, N_Fila).
Fila = [_304, _310, 0, _322, 1, 1],
N_Fila = [_G555, _G558, 0, 0, 1, 1].

?- Fila = [_,_,0,1,1,1], aplica_R1_fila_aux(Fila, N_Fila).
false.
```

3.1.3 Predicado `aplica_R1_fila/2`

Implemente o predicado `aplica_R1_fila/2`, tal que:

`aplica_R1_fila(Fila, N_Fila)`, em que `Fila` é uma fila (linha ou coluna) de um puzzle, significa que `N_Fila` é a fila resultante de aplicar a regra 1 à fila `Fila`. Isto significa que todas as posições vazias de `Fila` que possam ser preenchidas para respeitar a regra 1 se encontram preenchidas em `N_Fila`. Para tal, o predicado `aplica_R1_fila_aux` deve ser repetidamente aplicado a `Fila`, até que não sejam preenchidas novas posições. Se a `Fila` tiver três zeros (uns) seguidos, o predicado deve devolver `false`.

Por exemplo,

```
?- Fila = [1,_,0,_,1,0], aplica_R1_fila(Fila, N_Fila).
Fila = N_Fila, N_Fila = [1, _310, 0, _322, 1, 0].

?- Fila = [0,_,0,1,1,_], aplica_R1_fila(Fila, N_Fila).
Fila = [0, _310, 0, 1, 1, _334],
N_Fila = [0, 1, 0, 1, 1, 0].

?- Fila = [_,_,0,_,1,1], aplica_R1_fila(Fila, N_Fila).
Fila = [_304, _310, 0, _322, 1, 1],
N_Fila = [_304, 1, 0, 0, 1, 1].

?- Fila = [_,_,0,1,1,1], aplica_R1_fila(Fila, N_Fila).
false.
```

3.1.4 Predicado `aplica_R2_fila/2`

Implemente o predicado `aplica_R2_fila/2`, tal que:

`aplica_R2_fila(Fila, N_Fila)`, em que `Fila` é uma fila (linha ou coluna) de um puzzle, significa que `N_Fila` é a fila resultante de aplicar a regra 2 à fila `Fila`. Seja N metade do número de elementos de `Fila`. Aplicar a regra 2 significa que se `Fila` já contiver N zeros (uns), todas as posições vazias de `Fila` devem ser preenchidas com uns (zeros) em `N_Fila`. Se o número de zeros ou uns ultrapassar N, o predicado deve devolver `false`.

Por exemplo,

```
?- Fila = [0,_,_,0,1,_], aplica_R2_fila(Fila, N_Fila).
Fila = N_Fila, N_Fila = [0, _310, _316, 0, 1, _334].

?- Fila = [0,_,_,0,1,0], aplica_R2_fila(Fila, N_Fila).
Fila = [0, _310, _316, 0, 1, 0],
N_Fila = [0, 1, 1, 0, 1, 0].
```

```
?- Fila = [0,0,_,0,1,0], aplica_R2_fila(Fila, N_Fila).
false.
```

3.1.5 Predicado `aplica_R1_R2_fila/2`

Implemente o predicado `aplica_R1_R2_fila/2`, tal que:

`aplica_R1_R2_fila(Fila, N_Fila)`, em que `Fila` é uma fila (linha ou coluna) de um puzzle, significa que `N_Fila` é a fila resultante de aplicar as regras 1 e 2 à fila `Fila`, por esta ordem.

Embora a ordem de aplicação das regras seja irrelevante para a resolução do puzzle, não o é para efeitos de avaliação do predicado. Assim, deve respeitar a ordem indicada. Por exemplo, dada a fila `[1,1,_,0,_,_]` se aplicarmos primeiro a regra 1 e depois a regra 2, obtemos `[1,1,0,0,1,0]`; se aplicarmos as regras por ordem inversa obtemos `[1,1,0,0,1,_]`.

Por exemplo,

```
?- Fila = [1,0,_,_,0,_], aplica_R1_R2_fila(Fila, N_Fila) .
Fila = N_Fila, N_Fila = [1, 0, _1452, _1458, 0, _1470].

?- Fila = [1,1,_,0,_,_], aplica_R1_R2_fila(Fila, N_Fila) .
Fila = [1, 1, _1452, 0, _1464, _1470],
N_Fila = [1, 1, 0, 0, 1, 0].

?- Fila = [0,0,_,0,1,0], aplica_R1_R2_fila(Fila, N_Fila).
false.
```

3.1.6 Predicado `aplica_R1_R2_puzzle/2`

Implemente o predicado `aplica_R1_R2_puzzle/2` , tal que

`aplica_R1_R2_puzzle(Puz, N_Puz)`, em que `Puz` é um puzzle, significa que `N_Puz` é o puzzle resultante de aplicar o predicado `aplica_R1_R2_fila`, às linhas e às colunas de `Puz`, por esta ordem.

Embora a ordem de aplicação seja irrelevante para a resolução do puzzle, não o é para efeitos de avaliação do predicado. Assim, deve respeitar a ordem indicada: primeiro aplicar o predicado `aplica_R1_R2_fila` às linhas, e depois aplicar este predicado às colunas.

Por exemplo, sendo `Puz` o puzzle representado na Figura 1, temos[1]

```
?- ..., aplica_R1_R2_puzzle(Puz, N_Puz), escreve_Puzzle(N_Puz).
- - 0 - - 1
```

[1] O predicado `escreve_Puzzle` está definido no ficheiro "`codigo_comum.pl`".

```
- - - - - 1
- - - - - 0
- 0 - 1 0 0
- 1 - - - 1
0 1 0 - - 0
```

3.1.7 Predicado `inicializa/2`

Implemente o predicado `inicializa/2` , tal que

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

Recorda-se que para inicializar um puzzle, devemos aplicar as regras 1 e 2, a cada linha e coluna do puzzle, até não serem preenchidas novas posições.

Por exemplo, sendo `Puz` o puzzle representado na Figura 1, temos

```
?- ..., inicializa(Puz, N_Puz) , escreve_Puzzle(N_Puz).
- - 0 - - 1
- - - - - 1
- - - - - 0
1 0 1 1 0 0
- 1 - 0 - 1
0 1 0 1 1 0
```

Note que o resultado anterior corresponde ao puzzle da Figura 2.

3.2 Predicado para a verificação da regra 3

Implemente o predicado `verifica_R3/1`, tal que

`verifica_R3(Puz)` significa que no puzzle `Puz` todas as linhas são diferentes entre si e todas as colunas são diferentes entre si.

Note que uma linha ou coluna contendo variáveis nunca é igual a nenhuma outra. Por exemplo:

```
?- Puz = [[0,_,_,1],
          [0,_,_,1],
          [1,_,_,_],
          [1,0,_,_]], verifica_R3(Puz).
Puz = ...

?- Puz = [[0,1,0,1],
          [1,_,_,_],
          [0,_,_,1],
```

```
          [1,0,_,_]], verifica_R3(Puz).
Puz = ...

?- Puz = [[0,1,0,1],
          [0,1,0,1],
          [1,_,_,_],
          [1,0,_,_]], verifica_R3(Puz).
false.
```

3.3 Predicado para a propagação de mudanças

Segundo a abordagem apresentada na Secção 1, de cada vez que é mudado o valor de uma posição (L, C) de um puzzle, essa mudança deve ser propagada à linha L e à coluna C. Esta propagação consiste em aplicar as regras 1 e 2 à linha L e à coluna C, e propagar, recursivamente, as mudanças que daí resultem.

Nesta secção vamos definir o predicado `propaga_posicoes` que recebe uma lista de posições a propagar e um puzzle, e determina o puzzle resultante da propagação.

Por exemplo, considere-se que no puzzle da Figura 4 é testada a posição $(1, 2)$ com o valor 0:

1	0	0			1
					1
					0
1	0	1	1	0	0
	1		0		1
0	1	0	1	1	0

Apresentamos de seguida o rastreio das chamadas ao predicado `propaga_posicoes` (apenas é representado o 1º argumento, isto é, a lista de posições a propagar).

`propaga_posicoes([(1,2)], ...)` origina o puzzle

1	0	0	1	0	1
					1
					0
1	0	1	1	0	0
	1		0		1
0	1	0	1	1	0

e a chamada `propaga_posicoes([(1,4), (1,5)], ...)`.

`propaga_posicoes([(1,4), (1,5)], ...)` origina o puzzle

1	0	0	1	0	1
			0		1
			0		0
1	0	1	1	0	0
	1		0		1
0	1	0	1	1	0

e a chamada `propaga_posicoes([(2,4), (3,4), (1,5)], ...)`.

`propaga_posicoes([(2,4), (3,4), (1,5)], ...)` não altera o puzzle e origina a chamada `propaga_posicoes([(3,4), (1,5)], ...)`.

`propaga_posicoes([(3,4), (1,5)], ...)` origina o puzzle

1	0	0	1	0	1
			0		1
			0	1	0
1	0	1	1	0	0
	1		0		1
0	1	0	1	1	0

e a chamada `propaga_posicoes([(3,5), (1,5)], ...)`.

`propaga_posicoes([(3,5), (1,5)], ...)` não altera o puzzle e origina a chamada `propaga_posicoes([(1,5)], ...)`.

`propaga_posicoes([(1,5)], ...)` não altera o puzzle e origina a chamada `propaga_posicoes([], ...)`.

Implemente o predicado `propaga_posicoes/3`, tal que

`propaga_posicoes(Posicoes, Puz, N_Puz)`, em que `Posicoes` é uma lista de posicoes e `Puz` é um puzzle, significa que `N_Puz` é o resultado de propagar, recursivamente, (as mudanças de) as posições de `Posicoes`.

3.4 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.

Por exemplo, sendo `Puzzle` o puzzle da Figura 1

```
?- ..., resolve(Puzzle, Solucao), escreve_Puzzle(Solucao).
1 0 0 1 0 1
0 0 1 0 1 1
1 1 0 0 1 0
1 0 1 1 0 0
0 1 1 0 0 1
0 1 0 1 1 0
Puzzle = [[_G20, _G23, 0, _G29, _G32, 1],  ...,
Solucao = [[1, 0, 0, 1, 0, 1], [0, 0, 1, 0, 1, 1], ...,
   [0, 1, 1, 0|...], [0, 1, 0|...]] .
```

4 Avaliação

A nota do projecto será baseada nos seguintes aspectos:

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

`aplica_R1_triplo`	0.5 val.
`aplica_R1_fila_aux`	1.0 val.
`aplica_R1_fila`	1.0 val.
`aplica_R2_fila`	2.0 val.
`aplica_R1_R2_fila`	1.0 val.
`aplica_R1_R2_puzzle`	1.0 val.
`inicializa`	2.0 val.
`verifica_R3`	2.0 val.
`propaga_posicoes`	3.5 val.
`resolve`	2.0 val.

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

5 Penalizações

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

6 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 4 de Maio** de 2019, 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.

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.

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).

7 Cópias

Projectos iguais, ou muito semelhantes, originarão a reprovação na disciplina e, eventualmente, o levantamento de um processo disciplinar. Os programas entregues serão

[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.

[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.

testados em relação a 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.

8 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.