37_T

Inteligência Artificial

1.º Semestre 2014/2015

Fill-a-Pix

Relatório de Projecto

João Ferreira 76390

Raquel Cristóvão 76513

Mariana Vicente 76527

Índice

1 Implementação Tipos e Representação Problema PSR 3
1.1 Tipos Abstractos de Informação 3
1.2 Representação do problema Fill-a-Pix como PSR 4
2 Implementação Procuras e Funções Obrigatórias 6
2.1 Fill-a-pix⟶psr 6
2.2 Psr->Fill-a-pix 7
2.3 Heurística de Grau 7
2.4 Heurística MRV 8
2.5 Procura-Retrocesso e Inferência 8
3 Optimizações, Heurísticas e Técnicas adicionais utilizadas 9
3.1 Optimizações especificas para o problema Fill-a-Pix 9
3.2 Criação/Combinação de Heurísticas 10
4 Estudo Comparativo 11
4.1 Critérios a analisar 11
4.2 Testes Efectuados 11
4.3 Resultados Obtidos 12
4.4 Comparação dos Resultados Obtidos 12
4.5 Escolha do resolve-best 14
4.6 Testes em Código Lisp 14

1. Implementação Tipos e Representação Problema PSR

1.1 Tipos Abstractos de Informação

Para encontrar uma resolução para o problema do fill-a-pix foram criados dois tipos diferentes de estruturas de dados: **restricao** e **psr**.

A primeira estrutura de dados, **restricao,** é uma estrutura que é definida da seguinte forma:

(defstruct restricao lista-variaveis

funcao-verifica)

Desta forma, a cada restrição é atribuída uma lista de variáveis (que se verá no ponto seguinte o que significam ao certo) e uma função que verifica se a restrição é verdadeira. Com este tipo de estrutura de dados pretende-se facilitar a implementação do tipo seguinte, o **PSR**.

A estrutura de dados **PSR** é definida da seguinte forma:

(defstruct psr variaveis

dominios

restricoes

*atribuicoes-var)**

Para cada psr, existirá um conjunto de variáveis, onde cada uma tem o seu domínio (valores que cada variável pode ter). Existem ainda as restrições, que afetam os valores das variáveis. Este último argumento é derivado da estrutura de dados acima, **restricao**.

Um exemplo fácil para perceber este mecanismo é imaginar o seguinte caso: Um psr tem a lista de variáveis (x,y), ambas com domínio (0,1,2). Se o psr tiver, por exemplo, duas restrições que informem que x=1 e y>x, facilmente se consegue determinar os valores de ambas as variáveis. Neste caso x=1 e y=2. A primeira restrição receberia apenas a variável x e a segunda ambas as variáveis, executando uma função para verificar que valores do domínio tornam a restrição verdadeira, sabendo já que x=1. Este exemplo de pequeno grau de dificuldade permite perceber que estas duas estruturas de dados poderão ser uma boa forma de implementar a resolução de um problema fill-a-pix, como se irá demonstrar no ponto 1.2.

Habitualmente, apenas os três primeiros argumentos (variáveis, domínios e restrições) fazem parte de uma estrutura **PSR.** Porém decidimos incluir também um argumento atribuicoes-var, onde se guarda o valor atribuído a cada variável. Funciona assim, como uma lista de tuplos. Recorrendo ao exemplo anterior, seria uma lista formada pelos tuplos (x,1) e (y,2).

O problema fill-a-pix poderia ser representado de outra forma, por exemplo, colocando as variáveis do problema numa lista isolada e ir testando todas as combinações possíveis até chegar a uma que fosse aceite. Apesar de ser uma implementação possível só permitiria resolver, em tempo aceitável, problemas muito simples, onde o limite de restrições fosse pequeno. A utilização da estrutura **PSR** permite utilizar métodos de procura eficientes e mecanismos capazes de resolver estes problemas num espaço muito menor de tempo e até mesmo de memória, como se irá demonstrar mais a frente. Sendo este um problema que lida com um número muito elevado de restrições, a estrutura **restricao** assume também uma grande importância, pois guarda toda a informação referente a estas. O uso de estruturas de dados **PSR** é assim muito utilizado no que toca a problemas que têm de obedecer a um elevado número de limitações.

1.2 Representação do problema Fill-a-Pix como PSR

O puzzle fill-a-pix pode ser representado como um PSR da seguinte forma:

Variáveis: São as “casas” do puzzle. Por exemplo, um puzzle 3x3 tem 9 casas. Neste projecto a decisão tomada foi a de representar cada variável pela linha e coluna que ocupam no tabuleiro, como é demonstrado no seguinte exemplo:

(0,0)	(0,1)	(0,2)
(1,0)	(1,1)	(1,2)
(2,0)	(2,1)	(2,2)

Domínios: Cada domínio representa os valores possíveis para cada variável, por isso decidiu-se que os domínios de todas as variáveis seriam (0,1). 0 indica que a casa é pintada com a cor branca e 1 que é pintada com a cor preta. Não existe mais nenhum valor possível, por isso não fazia sentido acrescentar mais nenhum valor ao domínio. Ao mesmo tempo, todas as variáveis partem em “igualdade”, ou seja, no inicio todas podem ser pintadas de uma cor ou de outra, enquanto ainda não foram vistas as restrições.

Restrições: Cada restrição recebe uma lista de variáveis (as que participam nessa mesma restrição) e executa uma função que vai verificar os valores possíveis para as variáveis. É guardado numa lista, que está definida como variável global, um valor entre 0 e 9 para cada variável. Este valor vai ser utilizado para calcular os valores possíveis para a variável. Por exemplo, se a casa (1,1) vier com valor 9, sabemos que todas as casas à volta são pretas e é adicionado a esta lista um tuplo ((1,1),9). As casas que não trazem valores (valor NIL) não são adicionadas à lista.

Representar as variáveis pela sua posição no tabuleiro, pareceu a escolha mais lógica de imediato, pois esta é a característica que melhor identifica cada variável. Os domínios, tratando-se dos valores possíveis que esta pode ocupar, apenas podem ser 0 ou 1(branco ou preto), já que mais nenhum valor é admissível. As restrições guardam o valor entre 0 e 9 pois este é necessário para verificar as limitações do problema, sendo assim este o local mais correcto para guardar e trabalhar com tal informação.

Como referido no ponto 1.1, foi adicionado um argumento **atribuicões** que é útil para se saber quais as variáveis que já têm um valor atribuído, não precisando assim de ir à lista de variáveis procurar o seu valor. É uma vantagem em termos de tempo, porque basta procurar se a variável está presente nesta lista de atribuicões para saber se já tem um valor do domínio, embora signifique um aumento da memória usada.

Julgamos assim, que esta é a forma mais eficiente de formular um problema fill-a-pix como um PSR.

2. Implementação Procuras e Funções Obrigatórias

2.1 Fill-a-pix->psr

A função Fill-a-pix->psr recebe como argumento um array com as restrições de cada variável (ou seja, neste caso, de cada casa do tabuleiro). Depois de retiradas as dimensões do array, a lista de variáveis e de domínios é preenchida. Na lista de variáveis, para cada linha e coluna, é adicionado um tuplo (linha,coluna) na lista enquanto que a lista de domínios é apenas uma lista com tuplos (0,1) do tamanho da lista de variáveis, pois todas têm o mesmo domínio.

Guardar as restrições que afectam uma variável é mais complexo e optou-se pelo seguinte método: foi criada uma lista que vai conter várias listas de variáveis lá dentro. A primeira variável de cada sub-lista é aquela que está a ser afectada pelas restantes variáveis dessa mesma sub-lista. Assim, pegando numa variável inicial, vê-se quais são as casas, que estando à volta, têm uma restrição. Por exemplo, começando pela variável (0,0), iriam ser examinadas as casas à volta desta à procura de restrições, neste caso as casas (0,1), (1,0) e (1,1). Imaginando que apenas a casa (1,1) tem uma restrição, nesta nova lista seria adicionada uma sub-lista com ((0,0);(1,1)). Já sabemos assim que a única variável que afecta (0,0) é a (1,1). Neste passo ainda não nos preocupamos com o valor da restrição em si.

Depois de construir todas as sub-listas, é guardada como variável global uma lista que contém tuplos (variável; valor de restrição). Ou seja, se a casa (1,1) tiver uma restrição de valor 9, será guardado na lista o tuplo ((1,1);9). Desta forma consegue-se saber sempre qual o valor de restrição associado a cada variável, em todos os momentos.

A função de teste às restrições pega na lista global e vai percorre-la, para ver se a atribuição que está a ser feita a uma variável é permitida pela restrição que é passada a esta função (esta restrição é também guardada numa variável global quando é chamada por exemplo, na função psr-atribuicao-consistente-p). Recorrendo a um exemplo: se estivermos a testar o valor 0 para a casa (0,0), no caso em que a única restrição existente está na casa (1,1) e tem valor 9. A lista de variáveis que contém restrições e que afectam (0,0) vai ter apenas a variável (1,1). De modo a permitir que a função termine mal encontre uma violação da restrição é calculado o número máximo possível de zeros e de uns. O número máximo de uns que pode haver é o valor da restrição (neste caso pode haver 9). O número máximo de 0 é o contrário, é o número de casas à volta da variável em questão

menos o valor da restrição (ou seja, neste caso, 9 casas à volta de (1,1) menos 9=0, logo não pode haver zeros). Como estávamos a testar o valor de 0 para a casa (0,0), percebemos que este não vai funcionar, pois ultrapassa desde logo o limite de zeros que pode haver à volta de (1,1). Quando se testar o valor 1, não vai haver qualquer problema pois vai estar sempre dentro do limite de uns, já que tudo à volta vai ter que ficar pintado. O raciocínio deste exemplo mais básico, é o mesmo para todos os valores de restrições existentes, sendo uma questão de mais ou menos passagens pela função de teste de restrições.

2.2 Psr->Fill-a-pix

Esta função recebe um psr e transforma-o num array cujo tamanho também é passado como argumento. Para fazer esta passagem basta percorrer a lista de variáveis do psr e fazer um parse-integer à linha e à coluna. Sabendo que a estrutura de uma variável é (linha coluna) a posição 0 da string é a linha e a posição 2 é a coluna (a posição 1 é do espaço). Tendo o valor da linha e da coluna, preenche-se o array na posição dada pelos dois argumentos anteriores com o valor dessa mesma variável. Com o exemplo da variável (0,2) ter valor 1, o valor da linha é 0 e da coluna é 2, ou seja, no novo array, na posição (0,2) vai colocar-se o valor 1.

2.3 Heurística de Grau

A heurística de grau tem como objectivo devolver a variável não atribuída que está envolvida no maior número de restrições com outras variáveis não atribuídas. Para isto, é percorrida a lista de variáveis não atribuídas na sua totalidade, e para cada variável visitam-se também todas as restrições. Cada vez que uma variável é encontrada na lista de variáveis de cada restrição, um contador é aumentado com o número de outras variáveis presentes nessa lista, que ainda não foram atribuídas. No final, a variável com o contador mais alto é o resultado da execução desta heurística. Em caso de empate, escolhe-se a primeira variável na lista das não atribuídas.

2.4 Heurística MRV

A heurística MRV tem como objectivo devolver a variável não atribuída que tem o menor domínio. Para isto, é percorrida a lista de variáveis não atribuídas e para cada variável, um contador é incrementado com o número de valores possíveis no domínio. No final, a variável com o contador

mais baixo é o resultado da execução desta heurística. Em caso de empate, escolhe-se a primeira variável na lista das não atribuídas.

2.5 Procura-Retrocesso e Inferência

Tanto a procura-retrocesso-simples como a procura-retrocesso-grau foram feitas sem uso de inferências, sendo que a única diferença entre elas é a forma como a próxima variável é escolhida (primeira da lista de não atribuídas e através da heurística grau, respectivamente).

Já nas procuras que usam forward checking e mac, com a heurística mrv, são feitas inferências que ajudam a reduzir o domínio, sendo que esta é uma vantagem, já que esta heurística usa o tamanho do domínio para escolher a próxima variável a ser atribuída. Para além disso, quanto menor o domínio, mais fácil é dar um valor a uma variável.

Decidiu-se criar as inferências como uma lista de tuplos, onde o primeiro elemento é a variável sobre a qual se está a inferir e o segundo elemento é o novo domínio da variável, ou seja uma lista de valores actualizada. Usando a função fornecida *revise* é feita a tentativa de diminuir o domínio de uma variável, ou seja, uma inferência. Quando de facto se verifica que naquele momento se pode alterar o domínio, é guardado o novo domínio na lista de inferências. Voltando à função de procura, o domínio da variável é substituído pelo domínio presente na lista de inferências (é feito um back-up dos domínios iniciais, caso a inferência esteja errada). Se a inferência estiver errada, mais tarde o domínio gravado no back-up volta a ser associado à variável.

3. Optimizações, Heurísticas e Técnicas adicionais utilizadas

3.1 Optimizações especificas para o problema Fill-a-Pix

Algumas optimizações que poderiam ser realizadas nas funções de modo a tornar a procura por retrocesso mais eficiente para um puzzle Fill-a-pix são:

- Definir os domínios das variáveis logo como (0,1), no momento em que estas são criadas, não necessitando de um novo ciclo sobre a lista de variáveis para lhes atribuir um domínio;

- Nas funções em que não se realizam inferências e que é necessário percorrer a lista de domínios de uma variável, não seria preciso chamar a função que o faz, porque sabe-se à partida que basta percorrer uma lista formada por 0 e 1.

- Tornar a função *psr-variaveis-nao-atribuidas* mais eficiente, não mostrando as variáveis por ordem. Quando esta função é executada, vai ter que percorrer sempre a lista de atribuições, onde estão todas as variáveis e os seus valores, e ver quais os valores que estão a NIL. Como a lista é visitada sempre pela ordem inicial, a ordem inicial das variáveis é mantida. Seria mais eficiente que sempre que uma variável voltasse a ter valor NIL, fosse adicionada ao fim da lista das não atribuídas, nunca sendo necessário voltar a percorrer a lista de atribuições para encontrar as que não têm valor.

- A implementação do último ponto, permitiria que várias funções que chamam a função referida no ponto anterior fossem mais eficientes, como consequência da maior rapidez da função que retorna as variáveis não atribuídas. Exemplo disso é a função *arcos-vizinhos-nao-atribuidos*, que necessita de percorrer essas variáveis para verificar se os dois “vizinhos” são variáveis não atribuídas.

3.2 Criação/Combinação de Heurísticas

A única heurística criada (que não foi implementada na segunda parte do projecto mas criada posteriormente, para ser testada neste relatório final) foi a seguinte:

- Heuristica1: As variáveis são seleccionadas pelo valor de restrição que possuem. As variáveis não atribuídas que trazem consigo restrições de valor 9 e 0 vão ser as primeiras a ser escolhidas (são os melhores casos, já que ou tudo está branco ou tudo está preto). De seguida são escolhidas as variáveis não atribuídas com valor de restrições 8 e 1 e assim sucessivamente. Em caso de empate é escolhida a primeira variável na lista das não atribuídas. Por exemplo, se a variável (0,0) tiver restrição

2, a variável (0,2) valor de restrição 1 e a variável (1,1) valor de restrição 6, a primeira a ser escolhida é a (0,2), seguida pela (0,0) e por fim a (1,1). Só depois destas, são seleccionadas todas as outras. Esta heurística tem um defeito que poderia ser melhorado. Nos cantos do puzzle sabemos que o valor máximo não é 9, mas sim 6 ou 4, dependendo da posição, por isso estes deviam ser os primeiros valores a ser tratados e não deviam ser tão adiados, como são segundo esta heurística.

4. Estudo Comparativo

4.1 Critérios a analisar

O principal critério para comparar as várias formas de resolver um problema Fill-a-pix foi o tempo que cada forma demora a dar uma solução ou se é capaz de a dar num tempo mínimo. Assim, é feita a análise de várias funções heurísticas, que utilizadas juntamente com funções de procura, permitem obter tempos diferentes. Neste projecto, foram utilizadas três funções heurísticas diferentes: grau, mrv e a heurística especificada no ponto 3.2 do relatório. Quanto às funções de procura, baseiam-se todas em métodos de retrocesso, onde em alguns casos são realizadas inferências.

4.2 Testes Efectuados

Foi utilizado um conjunto de 8 testes para verificar a qualidade das funções. Estes testes utilizam tabuleiros de diversos tamanhos, com mais ou menos restrições, o que permite ter uma noção se estes dois pontos interferem na qualidade das funções de procura, e se sim, em quais. Os testes, que estão especificados em código Lisp no ponto 4.6 do relatório, têm as seguintes características:

	Tamanho do Tabuleiro	**Número de Casas com Restrições**	**Número de Casas com Restrições de Valor 0 ou 9**
Teste 1	3 x 3 (9 casas)	1	1
Teste 2	3 x 3 (9 casas)	4	3
Teste 3	3 x 3 (9 casas)	3	0
Teste 4	5 x 5 (25 casas)	8	1
Teste 5	5 x 5 (25 casas)	9	3
Teste 6	10 x 10 (100 casas)	34	1
Teste 7	15 x 15 (225 casas)	102	8
Teste 8	20 x 20 (400 casas)	133	9

4.3 Resultados Obtidos

	P. Retrocesso-Simples	P. Retrocesso-Grau	P. Retrocesso-FC-MRV	P. Retrocesso-MAC-MRV	P. Retrocesso-Heuristica1
Teste 1	0	0	0	0	0
Teste 2	0	0	0	0	0
Teste 3	0	0	0	0	0
Teste 4	0,50	0,53	0,82	0,98	0,45
Teste 5	0,64	0,54	1,14	1,30	0,59
Teste 6	INFINITO	INFINITO	46,18	150 (2min 30s)	INFINITO
Teste 7	INFINITO	INFINITO	INFINITO	INFINITO	INFINITO
Teste 8	INFINITO	INFINITO	INFINITO	INFINITO	INFINITO

4.4 Comparação dos Resultados Obtidos

Os três primeiros testes obtiveram resultados iguais em todas as variantes das procuras. Sendo testes de reduzida dimensão, mesmo procuras que não fazem inferências conseguem resultados excelentes, pois o tempo que demora a percorrer todas as variáveis é muito pequeno, o que faz com que a ordem pela qual se escolhe a próxima variável a ser tratada, seja pouco relevante.

O quarto teste, de dimensão um pouco maior, regista bons resultados mas agora com algumas diferenças consoante a procura usada. O facto de ser um puzzle de tamanho reduzido permite continuar a dar bons resultados à procura simples, que não faz inferências e limita-se a escolher a próxima variável na lista de não atribuídas. Por ter que percorrer mais variáveis, o tempo de execução acaba por ser aumentado, mas numa escala muito pequena. A procura que utiliza a heurística de grau encontra-se nos mesmos moldes. Apesar de se tratar de uma boa heurística, escolher a variável que está envolvida no maior número de restrições, o custo de executar a função que permite saber o número de restrições que afectam essa variável acaba por ficar equivalente ao custo de executar a função retrocesso-simples. No teste 5 já se vê que há um ganho de tempo na função que utiliza a heurística de grau relativamente à procura por retrocesso simples, já que esta

última é bastante inconstante e depende bastante da formulação do puzzle, já que a última casa, por exemplo, pode "destruir" toda a procura feita por não ter sido utilizada uma boa heurística. A procura com a heuristica1 consegue resultados satisfatórios, nomeadamente no teste 4. No teste 5 como tem que ir à procura de mais variáveis com restrições acaba por demorar um pouco mais de tempo mas sem grandes consequências. É facilmente observável que as procuras que utilizam a heurística MRV, fazendo também inferências, são as que demoram mais tempo. Isto deve-se ao facto de serem funções que, ao fazerem inferências, perdem mais tempo a testar os valores possíveis para a variável e a colocá-los numa lista de inferências, actualizando mais tarde o psr com esses valores. Estes testes permitem, contudo, que não seja necessário refazer tantas vezes os valores atribuídos às variáveis, pois através das inferências consegue logo detectar-se quais os valores aceitáveis ou não consoante as restrições vistas. Apesar de ser uma perda nos testes de reduzida dimensão, é um ganho nos testes com tabuleiros maiores.

Como é observável pela tabela, no teste com tabuleiro 10x10, apenas as funções que realizam inferências conseguem resolver o puzzle num tempo aceitável. Fizeram-se testes até 20 minutos, sem que nenhuma das outras procura o tenha resolvido nesse limite temporal. O que nos testes de dimensão mais pequena, era uma ligeira desvantagem é agora bastante importante. Ao realizar a inferência, as funções de procura conseguem descobrir valores aceitáveis ou não para as variáveis muito mais rapidamente, enquanto as que não realizam tendem a repetir demasiadas vezes os procedimentos, pois cada vez que encontrar uma incoerência com as restrições, podem ter que voltar a executar a função de procura em grande parte do psr. Apesar disso, os tempos que a procura com forward checking e com mac alcançam não são satisfatórias, pois ultrapassam largamente o tempo que era pretendido.

Os testes com maior dimensão não chegam a ser resolvidos, nem sequer com o uso de inferência. Uma solução para isto seria tentar melhorar as heurísticas de modo a que pudessem escolher as variáveis mais indicadas primeiro. As funções que usam heurísticas, quando tinham um empate, escolhiam a variável que aparecia primeiro na lista de não atribuídas, o que também não é um bom método. Uma possível solução seria executar a heurística mrv e em caso de empate executar, por exemplo, a heurística grau.

4.5 Escolha do resolve-best

A função resolve-best trata-se apenas de uma chamada à função procura-retrocesso-fc-mrv para resolver o problema fill-a-pix, pois foi a que conseguiu melhores resultados nos nossos testes, apesar de estarem longe do ideal.

4.6 Testes em Código Lisp

```
(defparameter teste1 (make-array (list 3 3) :initial-contents
        '((NIL NIL NIL)
          (NIL 9 NIL)
          (NIL NIL NIL))))

(defparameter teste2 (make-array (list 3 3) :initial-contents
        '((1 NIL 0)
          (NIL NIL NIL)
          (0 NIL 0))))

(defparameter teste3 (make-array (list 3 3) :initial-contents
        '((4 NIL NIL)
          (NIL 4 NIL)
          (2 NIL NIL))))

(defparameter teste4 (make-array (list 5 5) :initial-contents
        '((NIL NIL 1 NIL NIL)
          (NIL 1 NIL NIL 5)
          (1 NIL NIL NIL 6)
          (NIL NIL NIL 9 NIL)
          (NIL 5 6 NIL NIL))))

 (defparameter teste5 (make-array (list 5 5) :initial-contents
        '((0 NIL 2 NIL 0)
          (NIL NIL NIL NIL NIL)
```

```
        (2 NIL 5 NIL 2)
        (NIL NIL NIL NIL NIL)
        (0 NIL 2 NIL 0))))

(defparameter teste6 (make-array (list 10 10) :initial-contents
        '((NIL 2 3 NIL NIL 0 NIL NIL NIL NIL)
        (NIL NIL NIL NIL 3 NIL 2 NIL NIL 6)
        (NIL NIL 5 NIL 5 3 NIL 5 7 4)
        (NIL 4 NIL 5 NIL 5 NIL 6 NIL 3)
        (NIL NIL 4 NIL 5 NIL 6 NIL NIL 3)
        (NIL NIL NIL 2 NIL 5 NIL NIL NIL NIL)
        (4 NIL 1 NIL NIL NIL 1 1 NIL NIL)
        (4 NIL 1 NIL NIL NIL 1 NIL 4 NIL)
        (NIL NIL NIL NIL 6 NIL NIL NIL NIL 4)
        (NIL 4 4 NIL NIL NIL NIL 4 NIL NIL))))

(defparameter teste7 (make-array (list 15 15) :initial-contents
        '((0 NIL NIL 4 3 2 1 NIL NIL NIL NIL NIL 3 NIL NIL)
        (NIL NIL 5 NIL NIL 4 NIL NIL 4 4 NIL NIL NIL NIL 3)
        (NIL 5 4 5 4 5 5 NIL 5 3 NIL 1 2 NIL 3)
        (4 NIL NIL NIL 4 NIL NIL 4 2 NIL 1 NIL NIL NIL NIL)
        (NIL NIL 5 4 NIL 2 2 NIL 1 0 NIL NIL 7 5 NIL)
        (NIL NIL NIL 5 NIL NIL 0 NIL NIL NIL NIL 4 5 NIL 2)
        (4 NIL NIL 5 4 2 0 0 NIL NIL NIL 5 6 NIL NIL)
        (5 NIL NIL 6 5 NIL NIL NIL NIL NIL 3 3 3 NIL 3)
        (NIL NIL 5 NIL 5 3 NIL NIL NIL NIL NIL NIL 3 NIL NIL)
        (5 NIL NIL 6 5 NIL 3 5 NIL 6 NIL NIL 0 NIL 0)
        (NIL NIL 5 NIL 4 3 2 4 5 NIL 4 NIL NIL 1 NIL)
        (NIL 7 NIL NIL 5 NIL NIL 1 NIL 5 5 5 NIL NIL NIL)
        (NIL NIL 6 4 4 4 3 1 2 4 NIL NIL 6 4 NIL)
        (NIL 5 NIL 6 NIL NIL NIL NIL NIL 4 6 NIL NIL NIL NIL)
        (NIL NIL NIL NIL NIL NIL 3 2 0 NIL 4 4 3 NIL 2))))
```

```
(defparameter teste8 (make-array (list 20 20) :initial-contents
        '((4 NIL NIL 6 NIL NIL NIL 6 NIL NIL 4 NIL 4 NIL NIL 1 NIL NIL 3 NIL)
          (NIL NIL 9 NIL 8 NIL 9 8 NIL NIL NIL NIL NIL 2 2 NIL 6 NIL NIL 3)
          (6 NIL NIL NIL NIL 7 7 NIL NIL 2 NIL 1 2 NIL 2 4 6 8 NIL 4)
          (5 NIL NIL NIL 3 NIL NIL NIL 3 NIL NIL NIL 0 NIL NIL NIL NIL NIL 7 NIL)
          (NIL NIL 1 NIL NIL NIL NIL NIL NIL 0 NIL NIL NIL NIL NIL NIL 5 5 NIL 4)
          (NIL NIL NIL NIL NIL 0 NIL NIL 1 NIL NIL NIL NIL NIL NIL 4 NIL 6 NIL 4)
          (NIL 0 NIL NIL NIL 3 NIL NIL 3 NIL 2 NIL 2 NIL 1 3 4 NIL NIL NIL)
          (NIL NIL NIL 6 NIL NIL NIL NIL 3 4 NIL NIL 5 NIL NIL NIL 6 NIL NIL NIL)
          (0 NIL NIL 8 NIL NIL 3 NIL 4 NIL 6 8 6 NIL NIL NIL NIL 8 NIL 3)
          (NIL NIL NIL NIL 6 4 NIL 2 3 NIL NIL NIL NIL 7 6 NIL 7 9 NIL NIL)
          (NIL NIL NIL NIL NIL 5 NIL NIL NIL 5 NIL 6 7 NIL NIL NIL 5 7 NIL NIL)
          (NIL 7 8 NIL 8 NIL NIL NIL NIL NIL 4 NIL NIL 6 NIL NIL NIL 6 NIL NIL)
          (NIL NIL NIL 7 NIL NIL 7 NIL NIL 5 NIL NIL NIL NIL NIL NIL 1 NIL NIL 6 NIL)
          (5 NIL NIL NIL NIL NIL NIL NIL 3 3 NIL 6 NIL NIL 5 NIL NIL NIL NIL NIL)
          (NIL Nil 7 NIL 7 NIL NIL NIL 5 NIL NIL NIL 7 NIL 3 NIL 3 NIL NIL 5)
          (NIL 5 NIL 7 NIL NIL NIL 1 NIL 1 NIL NIL 7 NIL NIL NIL NIL NIL NIL NIL)
          (NIL 5 NIL NIL NIL 4 5 NIL NIL NIL NIL NIL NIL NIL 1 NIL 2 5 NIL 3)
          (NIL 7 9 NIL 8 NIL NIL 1 NIL NIL NIL NIL 7 NIL NIL 6 NIL NIL 5 NIL)
          (NIL 5 NIL 7 NIL NIL NIL NIL NIL 5 NIL NIL NIL NIL 3 NIL NIL NIL NIL 3)
          (NIL NIL NIL NIL 6 5 NIL NIL 4 5 5 NIL NIL 3 NIL NIL NIL 3 NIL NIL)))
```