Serve este documento para esclarecer algumas dúvidas frequentes sobre o conceito de consistência utilizado no projecto da cadeira de IA, bem como as funções de procura e heurísticas que vão ser testadas explicitamente.

1) Restrições/Funções de validação

Uma abordagem possível à validação de restrições é testar a função de validação apenas se todas as variáveis referenciadas na restrição estiverem atribuídas. No entanto, no caso particular do problema fill-a-pix e outros problemas com restrições globais esta **não é** a abordagem **mais eficiente**, e portanto **não** é a abordagem que **deve ser seguida** no projecto.

A **abordagem** que deve ser **seguida** no projecto de IA baseia-se em dois princípios:

1. **Uma restrição deve ser considerada verificada/validada até se conseguir provar o contrário**.
2. **Se for possível concluir que a restrição não é verificada mesmo sem todas as variáveis estarem atribuídas, a função de validação deve retornar NIL**. Por exemplo, numa restrição do problema Fill-a-pix com o número 9, representando que todas as casas à volta têm que ser pretas, se a primeira casa atribuída corresponder ao valor branco, então consegue-se perceber que a restrição irá sempre falhar independentemente do valor das casas restantes (pois no máximo iriamos ter 8 casas pretas e não as 9 pretendidas), e deverá ser retornado NIL. Reparem que se isto não for feito, o algoritmo de procura retrocesso ia ser obrigado a fazer 8 atribuições seguintes (e tentar estupidamente todos os valores possíveis para elas) para chegar sempre à mesma conclusão. Embora fosse possível usar inferência com AC3 para chegar à conclusão que a atribuição é inválida, este tipo de inferência é muito mais lenta (requer a enumeração explicita de todos os valores possíveis do domínio para as variáveis envolvidas), do que o tipo de inferência local utilizado internamente na função de validação (no caso do fill-a-pix determinada eficientemente por uma expressão matemática).

2) Funções de consistência

Tendo em conta os princípios discutidos acima, as funções de consistência deverão ter o seguinte comportamento:

psr-consistente-p

```
function psr-consistente-p (psr)
    testes = 0
    for each restricao r in restricoes(psr)
      testes = testes + 1
      if r is not verified then return False,testes
    return True, testes
```

Assumindo que as restrições podem ser testadas mesmo sem variáveis atribuídas, temos de testar todas as restrições para testar se um psr é consistente. Se alguma das restrições falhar, podemos retornar imediatamente e não precisamos de testar as restantes. Se chegarmos ao fim sem nenhuma restrição falhar, então o psr é necessariamente consistente.

psr-variavel-consistente-p

Esta função é semelhante à anterior com a única diferença de que apenas queremos saber se uma variável é consistente. Portanto, apenas nos precisamos de preocupar em testar as

restrições da variável, pois uma restrição que envolva apenas "y" e "z" nunca irá afectar a consistência de "x".

```
function psr-variavel-consistente-p (psr,var)
    …
    for each restricao r in restricoes(psr,var)
      …
```

psr-atribuicao-consistente-p

Esta função deve ter o comportamento equivalente a atribuirmos temporariamente o valor à variável, testarmos a função psr-variavel-consistente-p, e repormos o psr tal como estava.

```
function psr-atribuicao-consistente-p (psr,var,valor)
    backup dos valores necessários do psr
    adicionar var=valor ao psr
    result,testes = psr-variavel-consistente-p(psr,var)
    repor psr no estado original
    return result,testes
```

psr-atribuicoes-consistentes-arco-p

Nesta função pretende-se verificar se duas atribuições são consistentes uma com a outra. Para tal, e de modo a sermos o mais eficientes possível, apenas temos de testar as restrições que referenciam ambas as variáveis.

```
function psr-atribuicoes-consistentes-arco-p(psr,var1,valor1,var2,valor2)
    backup dos valores necessários do psr
    adicionar var1=valor1 ao psr
    adicionar var2=valor2 ao psr
    testes=0
    for each restricao r in restrições(psr,var1)
        if var2 in variaveis(r)
            testes = testes + 1
            if r is not verified then
                repor psr no estado original
                return False,testes
    repor psr no estado original
    return True,testes
```

3) Funções Procura, Heurísticas e Inferência

procura-retrocesso-simples

Como na procura por retrocesso fazemos uma atribuição de cada vez, podemos usar como teste de consistência a função psr-atribuicao-consistente-p. O número de testes totais de uma

procura por retrocesso simples corresponde à soma de todos os testes feitos pela função de consistência mais a soma de todos os testes feitos nas chamadas recursivas.

```
function procura-retrocesso-simples(psr)
    testesTotais = 0
    if completo(psr) return psr,testesTotais

    var = primeira(variaveis-não-atribuidas(psr))
    foreach valor in domínio(psr,var)
        consistente,testes = psr-atribuicao-consistente-p(psr,var,valor)
        testesTotais += testes
        if consistente
            adiciona var=valor ao psr
            resultado,testes = procura-retrocesso-simples(psr)
            testesTotais += testes
            if resultado return resultado,testesTotais
            remove var=valor do psr
    return NIL,testesTotais
```

Procura-retrocesso-grau

Esta função é praticamente igual à procura-retrocesso-simples, com a única diferença de que a próxima variável a ser atribuída é determinada pela heurística de grau, em vez de ser a primeira da lista de variáveis não atribuídas. A heurística de grau consiste em escolher como próxima variável a que tiver maior grau, ou seja a variável envolvida no maior número de restrições com **outras variáveis não atribuídas**. Se houver mais que uma variável com o grau máximo, deve ser retornada a que aparecer primeiro na lista de variáveis não atribuídas.

Procura-retrocesso-fc-mrv

Esta função escolhe como a próxima variável a ser escolhida aquela que tiver o domínio mais pequeno. Se houver mais que uma variável com o domínio mínimo deve ser escolhida a que aparecer primeiro na lista de variáveis não atribuídas.

```
function procura-retrocesso-fc-mrv(psr)
    testesTotais = 0
    if completo(psr) return psr,testesTotais

    var = MRV(psr)
    foreach valor in domínio(psr,var)
        consistente,testes = psr-atribuicao-consistente-p(psr,var,valor)
        testesTotais += testes
        if consistente
            adiciona var=valor ao psr
            inferências,testes = forward-checking(psr,var,valor)
            testesTotais += testes
            if inferências
                adiciona inferências ao psr
                resultado,testes = procura-retrocesso-fc-mrv(psr)
                testesTotais += testes
                if resultado return resultado,testesTotais
                remove inferências do psr
            remove var=valor do psr
    return NIL,testesTotais
```

Como se pode ver no pseudocódigo, o número de testes totais corresponde à soma do número de testes de consistência, do número de testes feitos no processo de forward checking, e ao número de testes feitos nas chamadas recursivas.

Existem várias maneiras de implementar o algoritmo de forward checking, no entanto, no projecto pretende-se que implementem o algoritmo de forward checking como uma simplificação do AC3/MAC. Assim quando tiverem que implementar o algoritmo MAC, poderão reutilizar grande parte da funcionalidade. A variável inferências corresponde a um conjunto de domínios alterados. Por exemplo, a inferência {x={0,1,2], y={1},z={}}, indica que o novo domínio de x será {0,1,2}, o novo domínio de y será {1} e o novo domínio de z será {}. Se o domínio de uma variável não se alterar, não se deve colocar nada referente a essa variável no conjunto de inferências.

```
function forward-checking(psr,var)
    testesTotais = 0
    inferências = {}
    lista-arcos = arcos-vizinhos-não-atribuidos(psr,var)

    foreach arco (v2,v1) in lista-arcos
        revise,testes = revise(psr,v2,v1,inferencias)
        testesTotais += testes
        if revise
            if size(domínio(inferências,v2)) == 0
                return False,testesTotais

    return inferências,testesTotais
```

A função arcos-vizinhos-não-atribuidos recebe um psr e uma variável e retorna uma lista de arcos (**sem arcos repetidos**) que correspondem às **outras variáveis ainda não atribuídas** que estão envolvidas numa restrição com a variável recebida. Por exemplo, num psr com 4 variáveis x,y,z,w, onde a variável x está atribuída, e existe uma restrição entre todas as variáveis, a função arcos-vizinhos-não-atribuidos(psr,y) irá retornar a seguinte lista de arcos: ((z,y),(w,y)). **Atenção,**

```
function arcos-vizinhos-nao-atribuidos(psr,var)
    lista-arcos = {}

    foreach var-natribuida in psr-variaveis-nao-atribuidas(psr)
        if var != var-natribuida
            if var-natribuida esta envolvida numa restricao com var
                  inserir arco (var-natribuida,var) no fim da lista
    return lista-arcos
```

a implementação usada para a função arcos-vizinhos-não-atribuidos não é a mais eficiente para o problema fill-a-pix, mas é a única maneira de garantir que a ordem dos arcos corresponde à ordem inicial das variáveis.

A função revise tem como objectivo actualizar o conjunto de inferências ao tentar tornar uma variável v2 consistente em arco para v1. Reparem também que se o domínio de uma variável v2 ficar vazio depois da função revise ter calculado o seu domínio, o algoritmo de forward checking deve retornar imediatamente False, pois não é possível encontrar uma solução.

Para a especificação da função procura-retrocesso-fc-mrv ficar completa, falta-nos apenas descrever em mais detalhe a função revise. Esta função tem algumas particularidades, como por exemplo preferir ir buscar o domínio de uma variável ao conjunto de inferências, e só se não existir é que deve ir buscar o domínio de uma variável ao psr. Embora este problema não se coloque no algoritmo forward-checking, pois para cada variável só é feito revise uma vez, na versão MAC pode acontecer que uma variável seja revista mais do que uma vez, e portanto pode acontecer que o domínio no conjunto de inferências estaja mais actualizado que o domínio no psr. Como a função revise vai ser exactamente igual para o algoritmo MAC e forward-checking decidiu-se manter este mecanismo. Pode acontecer também que a segunda variável, y, já esteja atribuída. Neste caso não vamos iterar no domínio da variável, mas vamos criar um domínio auxiliar apenas com o valor da atribuição como único valor do domínio.

```
function revise(psr,x,y,inferencias)
    testesTotais = 0
    revised = False
    domínio-x = buscar domínio da var ao cj. de inferências (se existir), ou ao psr
    novo-dominio-x = domínio-x
    if y está atribuída
        domínio-y = {valor atribuído a y}
    else
        domínio-y = buscar domínio da var ao cj. de inf. (se existir), ou ao psr

    foreach valor vx in domínio-x
        foundConsistentValue = False
        foreach valor vy in domínio-y
            consistente,testes = psr-atribuicoes-consistentes-arco-p(psr,x,vx,y,vy)
            testesTotais += testes
            if consistente
                foundConsistentValue = True
                break from closest loop
        if foundConsistentValue == False
            revised = True
            remover vx de novo-dominio-x

    if revised
        adicionar/actualizar novo-dominio-x no conjunto de inferências
    return revised,testesTotais
```

Procura-retrocesso-MAC-mrv

Embora tenha dado ligeiramente mais trabalho implementar o algoritmo de forward-checking como uma variante do AC3/MAC, a vantagem é que a implementação desta procura vai ser praticamente igual à implementação da procura com forward-checking. O algoritmo de procura-retrocesso fica igual, com a única excepção de chamarmos uma função MAC em vez de forward-checking na inferência (obviamente). Mas mesmo a função MAC vai ser extremamente parecida

```
function MAC(psr,var)
    testesTotais = 0
    inferências = {}
    lista-arcos = arcos-vizinhos-não-atribuidos(psr,var)

    foreach arco (v2,v1) in lista-arcos
        revise,testes = revise(psr,v2,v1,inferencias)
        testesTotais += testes
        if revise
            if size(domínio(inferências,v2)) == 0
                return False,testesTotais

            ;;única coisa que muda do fc para o MAC
            novos-arcos = arcos-vizinhos-não-atribuidos(psr,v2)
            remover (v1,v2) de novos-arcos
            adicionar novos-arcos à lista-arcos

    return inferências,testesTotais
```

à função de forward-checking, com a diferença de que se a função revise alterar o domínio, teremos que acrescentar novos arcos à lista de arcos.

4) Considerações finais

Estes algoritmos devem ser os implementados tal como estão para passarem os testes no sistema Mooshak. No entanto, os algoritmos apresentados neste documento podem não ser necessariamente os mais eficientes. Cabe aos alunos perceber se poderão ou não fazer pequenas alterações de modo a torna-los mais eficientes quando os forem usar no algoritmo resolve-best. Por exemplo, devem pensar em que situações é que pode ser desnecessário fazer o teste de atribuição-consistente-p quando vão fazer inferência a seguir. Outra questão que pode ser relevante pode ter a ver com o tipo de teste usado na função revise. Eventualmente, poderá (ou não) fazer sentido usar um teste que teste mais restrições (do que as entre x e y) para tentar detectar uma possível inconsistência mais cedo. Deixamos esta análise e exploração de alternativas para vocês.