Grupo Nº 38



**Inteligência Artificial**

1.º Semestre 2014/2015

**Fill-a-Pix**

Relatório de Projecto

**Índice**

[1 Implementação Tipos e Representação Problema PSR 3](#_Toc405474606)

[1.1 Tipos Abstractos de Informação 3](#_Toc405474607)

[1.2 Representação do problema Fill-a-Pix como PSR 4](#_Toc405474608)

[2 Implementação Procuras e Funções Obrigatórias 5](#_Toc405474609)

[2.1 Fill-a-pix🡪psr 5](#_Toc405474610)

[2.2 Psr->Fill-a-pix 6](#_Toc405474611)

[2.3 Heurística de Grau 6](#_Toc405474612)

[2.4 Heurística MRV 6](#_Toc405474613)

[2.5 Procura-Retrocesso e Inferência 6](#_Toc405474614)

[3 Optimizações, Heurísticas e Técnicas adicionais utilizadas 7](#_Toc405474615)

[3.1 Optimizações especificas para o problema Fill-a-Pix 7](#_Toc405474616)

[3.2 Criação/Combinação de Heurísticas 7](#_Toc405474617)

[3.3 Utilização de técnicas adicionais 7](#_Toc405474618)

[4 Estudo Comparativo 8](#_Toc405474619)

[4.1 Critérios a analisar 8](#_Toc405474620)

[4.2 Testes Efetuados 8](#_Toc405474621)

[4.2.1 Descrição dos testes 9](#_Toc405474622)

[4.3 Resultados Obtidos 9](#_Toc405474623)

[4.4 Comparação dos Resultados Obtidos 10](#_Toc405474624)

[4.5 Escolha do resolve-best 12](#_Toc405474625)

# Implementação Tipos e Representação Problema PSR

## Tipos Abstractos de Informação

Neste ponto devem ser abordados, pelo menos, os seguintes tópicos:

•Descrição detalhada das estruturas de dados utilizadas no programa.

•Justificação das escolhas efectuadas em relação a possíveis alternativas e às operações implementadas.

Se implementarem mais do que um tipo psr para o projecto, deverão descrever os vários tipos psr implementados e justificar a sua implementação.

**Restrição**

O tipo restrição é representado por uma estrutura composta por 2 campos:

1. Campo guarda a lista de variáveis envolvidas na restrição.
2. Campo guarda o predicado que é usado para verificar se a restrição está ou não a ser cumprida.

**Var**

Tipo abstracto que guarda informação sobre cada variável do PSR representado por uma estrutura Var que guarda:

1. Campo guarda o nome da variável *String*.
2. Campo que guarda o valor atribuído a essa variável.
3. Campo que guarda uma lista correspondente ao domínio da variável.
4. Campo com lista de restrições em que essa variável esta envolvida.

**PSR**

O tipo PSR é representado por uma estrutura composta por 3 campos:

1. Campo que guarda uma hash-table em que a chave é o nome da variável *String* e o valor para a entrada que é a estrutura **Var** que representa essa variável.
2. Campo que guarda uma lista dos nomes das variáveis pela mesma ordem usada quando se constrói o psr.
3. Campo que guarda a lista de todas restrições pela mesma ordem que é fornecida ao construtor do psr.

No início da implementação efetuou-se a implementação do PSR de uma maneira mais simples. Em vez de usar uma hash-table para guardar as estruturas **Var** usou-seuma lista, mas para aumentar a velocidade dos acessos do tipo aceder unicamente aos atributos de uma variável (ex: **psr-altera-dominio!**), em vez de ter que percorrer a lista para encontrar a estrutura **Var** correspondente os acessos pela *hash-table* são praticamente em tempo contante. A lista de variáveis guardada no 2º campo do PSR serve para conservarmos a ordem original, para quando for preciso devolver lista em que a ordem tem que respeitar a original podemos usar esta como auxiliar, isto é um *trade-off* de alguma memória por tempo nos acessos.

Cada variável guarda as restrições a que pertence o que faz com que as procuras pelas restrições de uma determinada variável sejam tempo constante, se for preciso listar todas as restrições de uma variável basta percorrer a lista de restrições dessa variável em vez de todas as restrições do PSR. Caso contrário tínhamos que percorrer a lista toda de restrições para encontrar as respectivas de uma determinada variável.

Na função da interface do PSR que devolve todas as variáveis não atribuídas (**psr-variáveis-não-atribuídas**) nós percorremos todas as variáveis para ver quais estão atribuídas, se em vez disso tivéssemos uma lista com todas as variáveis não atribuídas seria mais rápido porque só era necessário devolver a lista. Esta estratégia teria um custo ao ter que remover uma variável da lista de variáveis não atribuídas quando se fizesse uma atribuição.

## Representação do problema Fill-a-Pix como PSR

Nesta secção deverão descrever como é que decidiram representar o puzzle Fill-a-Pix como um problema de Satisfação de Restrições. Ou seja, deverão indicar as variáveis, os domínios, e as restrições escolhidas para representar um puzzle Fill-a-Pix. Deverão também justificar a escolha efectuada, comparando com possíveis alternativas de representação.

**Variáveis**

As variáveis correspondentes a cada quadrícula do puzzle são representadas por uma *string* no formato “L C” em que L corresponde á linha e C á coluna respectiva da quadrícula, deste modo é fácil transformar a *String* em dois números para indexar a quadrícula.

**Domínios**

Os domínios correspondem a uma lista de valores no caso do Fill-a-Pix os valores possível são 0 ou 1, 0 caso a quadricula não deva ser pintada e 1 caso essa quadricula deva ser pintada.

**Restrições**

# Implementação Procuras e Funções Obrigatórias

Nesta secção deverão descrever a implementação das várias funções/heurísticas e mecanismos de inferência pedidos explicitamente no enunciado. No entanto, deverão apenas focar-se na descrição de particularidades que não estejam especificadas no documento de esclarecimento. Ou seja, **não queremos ler o pseudocódigo e respectivos comentários** criados por nós. Em particular devem descrever as seguintes funções/algoritmos:

## Fill-a-pix🡪psr

Deverão descrever brevemente o algoritmo de conversão de um tabuleiro Fill-a-Pix para o tipo PSR. Devido à sua complexidade, é importante descrever o processo de criação e teste de restrições.

O algoritmo percorre o *array* dado de cima para baixo e da esquerda para a direita começando no canto superior esquerdo, para cada posição/quadrícula analisa o conteúdo e decide o que fazer.

Ao percorrer o tabuleiro Fill-a-Pix as quadrículas que tiverem **NIL** como valor são ignoradas as que tiverem um valor diferente de **NIL,** ou seja(0 – 9) nessa altura é criado uma restrição correspondente com as variáveis correspondentes e um predicado dependente do valor da quadrícula.

Para implementarmos as restrições usámos 3 predicados de validação distintos, todos os tipos de predicado aquando da sua criação guardam a lista de variáveis a que correspondem:

* O predicado **pred-1** é usado para quando se tem a certeza que todas as variáveis dessa restrição devem ter obrigatoriamente todos os valores iguais a 1. Ao chamar este predicado caso uma variável tenha o valor 0 ele devolve imediatamente **NIL** caso só encontre **NIL’s** e 1’s ele devolve **T** .
  + Qualquer quadrícula que tenha o valor 9 deverá criar um **pred-1.**
  + As quadrículas nas bordas do tabuleiro à excepção dos cantos caso tenham o valor 6 também deve ser criado um **pred-1** porque sabemos que as 6 variáveis correspondentes às quadrículas que envolvem devem ter o valor 1.
  + As quadrículas nos cantos do puzzle que tiverem o valor 4 também devem criar um **pred-1**

Porque no canto devido às regras do Fill-a-Pix a quadrícula só afecta as 3 quadrículas em redor dessa mais ela própria.

* O predicado **pred-0** é usado quando se sabe que todas as variáveis dessa restrição devem ter obrigatoriamente 0 como valor, ou seja quando se encontra uma quadrícula com o número 0 deve-se criar um **pred-0.** Ao chamar este predicado caso uma variável tenha o valor 1 ele devolve imediatamente **NIL** caso só encontre **NIL’s** e 0’s ele devolve **T**.
* O predicado **pred-geral** é usado em todas as outras situações que não são abrangidas pelo **pred-0** e pelo **pred-1**. Este predicado tem algumas particularidades, para além de guardar não só a lista de variáveis guarda também o valor da quadrícula que o originou. Ao chamar este predicado ele conta o número de variáveis não atribuídas e o número de variáveis atribuídas do seu grupo de variáveis, caso o número de variáveis não atribuídas mais o número de variáveis a 1 seja igual ou superior ao valor da quadrícula (que gerou a restrição) e a contagem das variáveis com valor 1 não ultrapasse o valor da quadrícula ele devolve **T** caso uma das condições anteriores falhe ele retorna **NIL**.

## Psr->Fill-a-pix

Aqui deverão descrever o algoritmo oposto, i.e. a conversão de um PSR resolvido para um tabuleiro Fill-a-Pix.

O algoritmo de conversão de **PSR** para Fill-a-Pix percorre todas as atribuições (correspondem ao par (var . valor)) dadas pela função (**psr-atribuições**), analisa a *string* que corresponde á variável traduze-a para 2 inteiros correspondente à linha e coluna e assim altera o valor no Fill-a-Pix com o valor da atribuição.

## Heurística de Grau

Deverão descrever brevemente como é que implementaram a heurística de grau.

Para implementar a heurística de grau implementou-se uma função (**maximum-degree**) que recebe um **PSR** que percorre todas as variáveis não atribuídas e para cada restrição dessa variável verifica se existe pelo menos uma variável não atribuída que esteja a ser restringida por ela, se encontrar uma incrementa o grau da variável e passa para a próxima restrição. Conforme vai percorrendo as variáveis vai verificando se tem maior grau que a que está guardada para ser retornada só se o grau for maior é que coloca essa na variável que vai ser devolvida.

## Heurística MRV

Deverão descrever brevemente como é que implementaram a heurística MRV.

A heurística MRV à semelhança da heurística de grau foi implementada com uma função (**MRV**) que também percorre todas as variáveis não atribuídas mas esta vai guardando a que tiver um domínio mais pequeno (domínio com menos valores possíveis) que a que estiver guardada para ser devolvida, caso encontre uma com domínio menor coloca-a na variável que vai ser devolvida.

## Procura-Retrocesso e Inferência

Nesta secção deverão descrever alguns detalhes que considerem relevante na implementação das procuras por retrocesso com os vários tipos de inferência. Mais uma vez, aqui não é pretendido que façam copy-paste do pseudocódigo, mas sim que descrevam algum pormenor que achem importante. Por exemplo, o que usaram para representar uma inferência, e como é que essa é aplicada/removida ao psr.

**Inferência**

1. *Hash-table* que guarda domínios das variáveis para cada variável.

A inferência foi implementada por uma estrutura que contém uma *hash-table* que para cada variável guarda na entrada o domínio correspondente, deste modo os acessos aos domínios de inferência são em tempo-constante. Como primeira implementação tínhamos uma lista de pares (variável . domínio) mas essa era mais ineficiente a encontrar o par caso as inferências tivessem muitas variáveis. Em puzzles que é necessário fazer muitas inferências este custo associado à procura na lista seria muito pesado.

Esta estrutura tem 3 funções associadas:

* **Adiciona-inferências** que recebe um PSR e a estrutura inferência copia os domínios das respectivas variáveis para o PSR e ao mesmo tempo guarda os valores dos domínios que estavam no PSR na inferência, assim se for preciso repor os valores antigos basta chamar a função outra vez. Ela actua como um *toggle* em relação aos domínios.
* **Set-domínio-inferências** recebe uma estrutura inferência, uma variável e um domínio, actualiza o domínio dessa variável na inferência com o recebido, caso essa variável ainda não esteja presente na inferência adiciona-se uma entrada para ela com o domínio recebido.
* **Get-dominio-inferencias** recebe uma estrutura do tipo inferência e uma variável e devolve o domínio da variável, caso não exista nenhuma entrada para essa variável retorna -1 como erro para poder distinguir caso o domínio da variável a devolver seja **NIL**.

# Optimizações, Heurísticas e Técnicas adicionais utilizadas

Neste capítulo deverão descrever com algum detalhe 3 tipos de estratégias usadas para tornar o vosso algoritmo de procura mais eficiente.

## Optimizações especificas para o problema Fill-a-Pix

Nesta secção deverão descrever optimizações efectuadas nas funções de modo a tornar a procura por retrocesso mais eficiente para um puzzle especifico do Fill-a-Pix. Podem também descrever outros tipos de optimizações efectuadas ao pseudocódigo original fornecido. Por exemplo, se não houver a restrição de que as coisas têm que ser feitas por uma determinada ordem, é possível criar versões ligeiramente mais eficientes da função vizinhos-não-atribuidos, da função psr-variaveis-não-atribuidas, etc.

Para a função **Fill-a-pix->PSR** criou-se uma versão optimizada (**Fill-a-pix->PSR-best**) que em vez de criar restrições para os casos triviais explicados na secção 2.1, mais concretamente os abrangidos pelos **pred-1** e **pred-0** atribui logo os valores respectivos ás variáveis abrangidas ou seja 0 se fosse **o pred-0** e 1 se fosse o **pred-1**, isto porque não há outra hipótese para estas variáveis e assim poupa-se tempo significativo na procura.

## Criação/Combinação de Heurísticas

Se criarem alguma heurística adicional para a escolha de variável/valor, ou se usarem alguma combinação de heurísticas deverão descrevê-las nesta secção.

Construímos uma versão melhorada da função (**arcos-vizinhos-não-atribuídos**) chamada (**arcos-vizinhos-não-atribuídos-best**) em vez de percorrer todas as variáveis não atribuídas percorre-se só as variaveis que estão nas restrições dessa variável reduzindo o tempo de procura significativamente.

## Utilização de técnicas adicionais

Nesta secção final deverão descrever outras técnicas adicionais utilizadas para tentar tornar o algoritmo de procura mais eficiente. Alguns exemplos (que podem funcionar ou não) são:

* Conflict-directed backjumping.
* Execução algoritmo MR3 antes de iniciar a procura.
* Utilização de restrições globais para testes de consistência e inferência.

# Estudo Comparativo

## Critérios a analisar

Os critérios em estudo são a memória alocada e o tempo de execução. Adicionalmente, a comparação de procuras é baseada no número de testes de consistência.

As procuras com menor número de testes serão mais inteligentes.

A nossa solução terá sucesso se resolver os problemas Fill-a-Pix em tempo aceitável, sem comprometer o normal funcionamento do computador (i.e. utilizar todos os recursos).

## Testes Efetuados

Considerem-se os seguintes Puzzles:

**Puzzle 1** – 10X10 genérico – corresponde ao exemplo 2 fornecido

**Puzzle 2** – 20X20 com abundancia de casos triviais (0,9,4,6) – desenvolvido pelo grupo

**Puzzle 3** – 10X10 genérico – corresponde ao exemplo 4 fornecido

**Puzzle 4** – 20X20 genérico – corresponde ao exemplo 7 fornecido

**Puzzle 5** – 20X20 genérico e mais complexo – construído com base no level3/puzzle10 disponível em [*http://www.kongregate.com/games/conceptis/basic-logic-fill-a-pix-light-vol-1*](http://www.kongregate.com/games/conceptis/basic-logic-fill-a-pix-light-vol-1)

## Descrição dos testes

|  |  |  |
| --- | --- | --- |
| Teste | Descrição | Objetivo |
| Teste 1 | Utilizar a função resolve-simples para resolver o puzzle 1, usando o código de ambas as entregas. | Implementação com listas (1ª entrega) vs *hashtables* (2º entrega) |
| Teste2 | Utilizar a função resolve-simples para resolver o puzzle 2, usando o código da 2ª entrega (mas sem as atribuições iniciais dos casos particulares) | Importância de atribuições prévias |
| Teste 3 | Correr todas as procuras desenvolvidas (simples, grau, fc-mrv, mac-mrv, best), usando o puzzle 3 | Averiguar qual a procura com menos testes consistência |
| Teste 4 | Resolver os puzzles 4 e 5 com o resolve-best | Validar a optimalidade da solução final |

**Tabela 1 –** Descrição e objetivo de cada teste realizado

## Resultados Obtidos

Resultados obtidos com a função time, disponível no Common Lisp

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Teste** | | **Time** | **Memória** | | **# testes de consistência** |
| **#** | **Especificidade** | **Real** | **Cons cells** | **Other bytes** |
| 1 | Listas | 4min 30.32s | 536,947,475 | 1,960 | n.a. |
| Hashtables | 0.640000s | 2,905,626 | 0 | n.a. |
| **2** | Sem atribuições prévias | 0.110000s | 334,483 | 390,432 | n.a. |
| Com atribuições prévias | 0.099000s | 334,483 | 390,400 | n.a. |
| **3** | Procura-retrocesso-simples | 3.208000s | 16,592,563 | 56 | 341289 |
| procura-retrocesso-grau | Superior a 4h | [teste abortado] | [teste abortado] | [teste abortado] |
| procura-retrocesso-fc-mrv | 1.439000s | 3,117,419 | 6,185,888 | 152358 |
| procura-retrocesso-mac-mrv | 2.528000s | 4,849,142 | 5,541,824 | 301562 |
| Procura-retrocesso-best | 1.037000 | 2,274,418 | 6,185,920 | 151959 |
| **4** | Puzzle 4 | 2.235000s | 5,565,791 | 10,596,496 | n.a. |
| Puzzle 5 | 8min 33.446s | 1,724,021,534 | 1,884,058,160 | n.a. |

**Tabela 2 –** Resultados de cada teste/variante

## Comparação dos Resultados Obtidos

**Teste 1**

A versão com o uso de *hashtables* foi substancialmente mais rápida e o uso de memória foi menor.

Este tipo de estrutura de dados associa Chaves a Valores, sendo que conhecendo a Chave, o acesso ao Valor é direto (redução do tempo de execução). Mais, caso os Valores sejam repetidos, apenas uma cópia é guardada (redução da memória utilizada).

**Teste 2**

Notou-se uma ligeira melhoria na execução com as atribuições adicionais nos casos triviais. A procura fica menos extensa, diminuindo o tempo de execução.

**Teste 3**

Constatou-se que existe uma relação direta entre o tempo de execução de uma procura e o nº de testes de consistência realizados.

**Gráfico 1 -** Comparação entre as procuras [a procura-retrocesso-grau foi omitida] com base no tempo de execução

**Gráfico 2** - Comparação entre as procuras [a procura-retrocesso-grau foi omitida] com base no número de testes de consistência realizados

Por ordem decrescente de tempo:

**Procura-retrocesso-grau** – esta pesquisa foi abortada ao fim de 4 horas. Apenas foi implementada a heurística de maior grau, sem qualquer propagação de restrições. Conclui-se que esta procura não é adequada ao tipo de problema Fill-a-pix.

**Procura-retrocesso-simples** – A procura é bastante mais rápida do que a anterior pois não é necessário o cálculo da heurística. No entanto, o nº de testes é muito elevado, o que dificultaria uma procura num puzzle superior a 10X 10.

**Procura-retrocesso-mac-mrv** – O tempo de execução e o nº de testes é ligeiramente inferior pois foi introduzido um mecanismo de propagação de restrições (MAC) e a uma heurística (MRV). O MAC garante consistência em todos os arcos.

**Procura-retrocesso-fc-mrv** – Esta procura consiste num mecanismo de propagação de restrições (Forward Checking) e a uma heurística (MRV). O mecanismo de FC apenas assegura consistência entre a variável atual. Dada a especificidade do problema Fill-a-pix (muitas variáveis), esta procura permitiu minimizar o número de teste de consistência efetuados.

As procuras anteriores são aceitáveis em problemas de pequenas dimensões mas testes adicionais revelaram que já não o são em problemas mais complexos. Este motivo levou ao desenvolvimento da procura-retrocesso-best.

**Procura-retrocesso-best** – Baseia-se na procura-retrocesso-fc-mrv mas cada variável apenas admite os arcos-vizinhos-não-atribuídos entre ela própria. Esta alteração reduz o número de testes de consistência efetuados.

**Teste 4**

Os resultados do puzzle 5 são inferiores ao puzzle 4 dado que se tratava de um problema mais complexo, no entanto é aceitável.

**Testes Mooshak**

A solução apresentou sucesso em todos os testes avaliados, exceto o teste 28. O algoritmo resolveu apenas 2 dos 3 puzzles apresentados.

Conclui-se que ainda poderiam ser feitas melhorias ao projeto, principalmente na técnica de procura e as suas heurísticas.

## Escolha do resolve-best

O algoritmo resolve-best consiste na conversão de um array para o tipo PSR e a sua posterior resolução.

A conversão é realizada no procedimento Fill-a-pix->psr-best que: construi o PSR (versão já otimizada) e realiza o pré processamento de variáveis com casos triviais (4 – cantos e 6- limites do mapa).

A resolução do PSR é feita pela procura-resolve-best. Esta procura consiste no forward checking e na heurística de Mininum Reamaining Value. No entanto, apenas são considerados arcos-vizinhos-não-atribuídos entre uma dada variável e ela própria.