Grupo Nº 6

TÉCNICO LISBOA

Inteligência Artificial

1.º Semestre 2014/2015

Fill-a-Pix

Relatório de Projecto

Índice

1 Implementação Tipos e Representação Problema PSR

1.1 Tipos Abstractos de Informação

O tipo Restrição utilizado no projeto foi definido através de uma estrutura constituída por dois atributos: uma lista, denominada **lista_vars**; e uma função lambda, com o nome de **predicado**. O primeiro elemento, como o nome sugere, é utilizado para guardar todas as variáveis que participam diretamente no predicado em causa. No que toca ao segundo elemento, pretende-se que guarde a função que constitui a restrição propriamente dita. Mais tarde foi adicionada um novo atributo à estrutura, **res**, um inteiro com o resultado que a soma das atribuições de todas as variáveis deve corresponder de forma à restrição ser satisfeita. Esta variável é apenas utilizada para a função de resolução resolve-best e o seu constributo para a performance do resolve-best será discutida mais tarde.

Para a criação deste tipo de estruturas, foram definidos dois construtores, nomeadamente: faz--restricao; e faz-res. A diferença entre os dois prende-se na quantidade e tipo de atributos que recebem por *input* para a criação da estrutura, sendo que o segundo, para além de uma lista de variáveis e uma função lambda, recebe também um inteiro.

A invocação dos diferentes construtores é feita igualmente por duas funções, sendo que uma delas, a cria-restricao, faz parte da interface especificada no enunciado do projeto, recebendo dois argumentos, uma lista de variáveis e um predicado como *input* e devolve a estrutura de restrição descrita e a cria-res-best, a qual, para além dos dois argumentos referidos anteriormente, recebe igualmente o inteiro correspondente ao número de quadrados que devem ser pintados numa única restrição, como evidenciado no problema *fill-a-pix*.

Quanto ao tipo PSR, também definido através de uma estrutura, é constituído por cinco listas, sendo elas uma lista com todas as variáveis presentes no problema, **lista_vars**; uma lista com os domínios de cada variável, **lista_dominios**, sendo que cada domínio é também ele representado por uma lista; uma terceira lista com todas as restrições de cada variável, **restricoes_vars**; uma lista com todas as restrições existentes no PSR, **lista_restricoes**; e, por fim, uma lista com os valores atribuídos a cada variável, denominada **lista_atribuicoes**. É importante referir que foi decidida, durante a implementação, a existência e uma correspondência entre as posições das listas de variáveis e domínios, ou seja, a primeira posição na lista de domínios, corresponde à primeira variável na lista de variáveis. Esta mesma correspondência occore entre a lista de variáveis e a lista de atribuições. Tal como o tipo Restrição, a estrutura PSR contém um elemento apenas utilizado no caso do resolve-best, o **restricoes_vars**, já que, na implementação para a primeira parte do projeto, esta é criada como uma lista vazia, tendo só uma função efetiva na otimização para *fill-a-pix*, onde passa a ser definido como uma *hashtable*, onde as chaves são as variáveis e o valor é uma lista com todas as restrições em que a variável está envolvida.

Foram definidos, de novo, duas funções para a criação deste tipo de dados. Foi definida primeiramente a cria-psr, a qual cria as listas de variáveis, domínios, restrições e atribuições, inerentes à resolução do PSR. Posteriormente, foi criada uma nova função com o intuito de redefinir o PSR, cria-psr-best, alterando assim o tipo de dados utilizado, ou seja, no caso da lista de domínios e na lista de

atribuições, passaram a ser utilizadas *hashtables*. Em ambos os casos a chave é uma variável, e o valor é o domínio ou a atribuição correspondende a essa variável. Tal como mencionado e explicado anteriormente, a **restricoes_vars** é neste momento criada.

A alteração do PSR em relação à versão criada inicialmente deve-se a questões de performance, uma vez que se pretende um acesso mais rápido tanto à atribuição de uma variável (caso esta existisse), como ao domínio da mesma. Ao converter para *hashtables*, eliminou-se a procura em tempo linear que acompanha o uso das listas, o qual, neste caso, era um problema recorrente.

1.2 Representação do problema Fill-a-Pix como PSR

Foi decidido resolver o problema *fill-a-pix* através de uma lista que contém todas as variáveis do problema, onde uma variável apresenta uma correspondência directa com uma casa no puzzle. Como no puzzle, cada casa pode estar preenchida ou não, assume-se que cada variável terá um domínio constituído por (0, 1), assumindo que uma casa vazia será um 0, e uma casa preenchida será um 1. Assim, é criada uma *hashtable* com as variáveis utilizadas como chave, e uma lista (0, 1) sendo o valor associado à chave.

As variáveis, sendo que cada uma corresponde a uma casa no puzzle, obtou-se por dar um nome consistente com a sua posição. Assim, o nome de cada variável depende da sua posição, podendo ser, por exemplo, “0-0” no caso de estar localizada no canto superior esquerdo, ou seja, a sua posição (linha,coluna) ser (0,0). O nome é, por isso, constituido pela linha e pela coluna, separadas por um hífen. Esta representação é útil no contexto do problema, uma vez que é bastante intuitiva no caso da representação e permite que se acompanhe o funcionamento dos algoritmos, em casos em que o puzzle é mais simples, para ter a certeza do funcionamento correcto do algoritmo.

Em relação às restrições, são utilizados dois elementos no PSR para guardar as restrições do problema. Em primeiro lugar, uma lista de restrições. Obtámos por criar um segundo tipo de dados para guardar as mesmas restrições, com uma organização diferente. Uma *hashtable* em que as variáveis eram novamente utilizadas como chave, e a cada uma estava associado uma lista com todas as restrições em que a variável estava envolvida. Uma variável estar envolvida numa restrição quer dizer que, para a verificação do predicado dessa restrição, a variável tem um papel direto.

Estas restrições são criadas através de uma *string*, que quando avaliada, é junta à lista de restrições. Como a estrutura das restrições é sempre a mesma no contexto do problema, criou-se um modelo de função a ser utilizado, igualmente numa *string*, à qual são juntas as variáveis envolvidas, assim como uma função que permite ir buscar o valor dessas mesmas variáveis. Assim, com o valor de todas as variáveis, e caso todas elas estejam atribuídas, verifica-se a consistência da restrição, somando o valor de cada variável, e avalia-se se a restrição está a ser satisfeita, ou não, ou seja, se o seu predicado se verifica. Para o contexto do problema, apenas é considerada inconsistente nos casos em que é impossível chegar ao resultado esperado com as variáveis já atribuídas.

Como alternativa à função que avalia da string devolvida pelo criador de restrições, podíamos ter utilizado directamente uma closure, em vez de criar uma string para ser mais tarde avaliada, podendo talvez ser um pouco mais intuitiva.

Comparativamente a utilizar apenas listas contra a nossa utilização de *hashtables*, temos a vantagem de tempo de procura de um elemento específico, que é linear no caso das listas. Não era necessariamente obrigatório utilizar uma lista com todas as variáveis presentes no problema, uma vez

que ambas as *hashtables* existentes no PSR tinham as variáveis todas presentes no campo da chave, mas optámos por o fazer.

2 Implementação Procuras e Funções Obrigatórias

2.1 Fill-a-pix→psr

A função fill-a-pix->psr trata de transformar um problema fill-a-pix dado como *input* e convertê-lo num problema de restrições, o *output*, de forma a que os nossos algoritmos de resolução e procura possam resolvê-lo.

Para a primeira entrega do projeto, a função fill-a-pix->psr recebe como argumento um *array* bidimensional e retorna uma estrutura do tipo PSR. A sua execução é um conjunto de ciclos *dotimes* que tratam de visitar cada elemento do referido *array*. Para cada um, este é adicionado a uma lista que contém todas as variáveis, sendo que a variável que corresponde a um elemento é, tal como anteriormente referido, designada por uma *string* "linha-coluna", sendo que linha e coluna correspondem aos índices dos ciclos *dotimes*.

Para além disso, como num problema fill-a-pix cada variável só pode ter dois valores distintos no seu domínio (0 ou 1), é também adicionada uma lista com estes dois elementos à lista que trata de armazenar todos os domínios. Note-se que, devido à execução dos ciclos, o índice nas listas de variáveis e domínios tem um mapeamento direto, ou seja, a variável com índice n na lista de variáveis tem o seu domínio associado no índice n da lista de domínios.

Caso o elemento que seja lido seja o termo NIL, o algoritmo sabe que não é necessário criar uma restrição. Caso contrário, é um número de 0 a 9 que está presente na posição (linha,coluna) do array e, nesse caso, é criada uma restrição envolvendo esse elemento e todos os outros elementos que o rodeiam no puzzle. Dado que para criar uma restrição é necessário um conjunto de variáveis que participam na mesma, criou-se uma função denominada cria-variaveis, a qual recebe como *input* o tipo de restrição que é e as linha e coluna da posição do elemento.

Este tipo de restrição serve para diferenciar quando o elemento lido com número está ou no canto do puzzle, ou nas margens, ou no centro. Esta diferenciação é necessária de forma a saber a quantidade e quais as variáveis que devem ser criadas, dado que, no primeiro caso, são quatro, nas margens são seis e no centro do puzzle contamos com nove variáveis a participar na restrição. A função que trata deste problema chama-se tipo-variavel.

O predicado da restrição é criado pela função cria-funcao-linear. Esta recebe como *input* a lista de variáveis que participam na restrição, bem como o valor do elemento. A sua execução constrói uma *string*, sendo esta avaliada como uma função lambda, a qual tem em si chamadas de funções para conseguir adquirir o valor atribuído a cada variável e verificar se a restrição é ou não satisfeita. Após a restrição ser criada, é junta a uma lista que contém todas as restrições desse PSR.

Após o *array* ser todo preenchido, é devolvida uma estrutura do tipo PSR composta pelas listas de variáveis, domínios e restrições criadas durante a execução dos ciclos.

2.2 Psr->Fill-a-pix

Esta função trata simplesmente de receber uma estrutura do tipo PSR e as dimensões do *array* final e devolve um *array* bidimensional preenchido conforme as atribuições das variáveis.

Primeiramente, é criado um array com as dimensões pretendidas e todo preenchido a 0. De seguida, com um ciclo *dotimes* que percorre toda a lista de atribuições do PSR, caso uma variável dessa

lista tenha a sua atribuição com valor 1, isto é, para, por exemplo, a variável "0-0", se estiver presente na lista de atribuições o par ("0-0" . 1), deve ser preenchida a posição (0,0), sendo o tuplo (linha,coluna), do *array* de retorno com o valor 1. Assim, tal como evidenciado no problema fill-a-pix, essa posição trata-se de um quadrado pintado. Caso a variável tenha o valor 0, isto é, para o exemplo anterior, estar presente na lista de atribuições o par ("0-0" . 0), sabemos que é uma posição que não deve ser pintada e, assim, não alteramos o conteúdo dessa posição no *array*.

Por último, a função trata de retornar o dito.

2.3 Heurística de Grau

A heurística de grau é assegurada por uma função denominada var-maior-grau. Esta recebe como argumento uma estrutura do tipo PSR e devolve a variável que participa num maior número de restrições.

A sua linha de execução começa com a chamada à função psr-variaveis-nao-atribuidas presente na interface requerida na primeira parte do projeto. Após obter a lista de variáveis não atribuídas com um ciclo *dotimes*, a função trata de adquirir a lista de restrições em que a variável participa, utilizando desta vez a função psr-variavel-restricoes.

Esta função tem também como variável interna uma lista de inteiros, denominada grau, onde guarda o grau de cada variável com um mapeamento direto de índices entre a lista de variáveis não atribuídas e aquela. À medida que verifica que uma variável está presente numa restrição e que há, pelo menos, mais uma variável constituinte dessa restrição que tenha valor atribuído NIL, o algoritmo incrementa a posição de índice correspondente à variável não atribuída corrente de análise do seu grau.

Após a análise da lista de variáveis não atribuídas, o algoritmo trata de percorrer a lista grau e encontrar o máximo grau. Após percorrer toda, devolve o a variável não atribuída cujo índice é o mesmo que o do grau máximo.

2.4 Heurística MRV

A função mrv trata de executar a heurística de *Minimum Remaining Values*. Primeiramente, começa por escolher como mínimo o domínio da primeira variável não atribuída da lista de variáveis não atribuídas.

Seguidamente, executa um ciclo para percorrer toda a lista de variáveis não atribuídas e, caso alguma delas tenha um domínio com menor *length*, essa passará a ser designada como a variável que o algoritmo retornará.

Por último, retorna a variável cujo tamanho do domínio é o mínimo entre elas todas. Note-se que caso haja mais que uma variável com o domínio do mesmo tamanho, é escolhida a que aparece primeiro na lista de variáveis não atribuídas.

2.5 Procura-Retrocesso e Inferência

As três procuras de retrocesso implementadas na segunda fase do projeto seguem todas a mesma estrutura, isto é, cada uma é constituída por uma função principal e uma outra recursiva, chamada pela principal. Esta função recursiva recebe, quando chamada pela primeira vez, um contador de testes com o valor de 0, o qual é atualizado pelo algoritmo e por cada retorno da função auxiliar de retrocesso.

O aspeto que teve mais relevância na tradução do pseudocódigo para Lisp nesta segunda parte é a assunção de que o *False* é diferente de NIL na função forward-checking, uma vez que, assumindo que *False* seria o mesmo que NIL, não nos seria possível distinguir entre uma lista de inferências vazia e o valor False aquando da execução das funções procura-retrocesso-fc-mrv-aux e procura-retrocesso-mac-mrv-aux, dado que uma lista vazia toma o valor NIL.

A implementação das inferências é uma lista de listas em que cada uma destas listas tem dois elementos: o primeiro é uma variável; e o segundo é uma lista que identifica o novo domínio dessa mesma variável proveniente das inferências.

As funções que operam diretamente com as inferências são as seguintes:

- antigo-dom, a qual trata de guardar os domínios de cada variável do PSR na lista de inferências de forma a que estes sejam preservados como eram antes de serem manipulados e, assim, poderem ser utilizados caso sejam necessários;
- troca-dominio, que ao receber o PSR e uma lista de domínios, substitui os domínios das variáveis no PSR pelos domínios das variáveis presentes na lista que recebe. É utilizada para substituir os domínios no PSR pelas inferências feitas, ou, no caso de se detetar uma inconsistência, reverter as alterações com o antigo domínio que estava guardado.
- dom-inferencias, a qual devolve o respetivo domínio da variável dada como *input* presente na lista de inferências.

3 Optimizações, Heurísticas e Técnicas adicionais utilizadas

3.1 Optimizações especificas para o problema Fill-a-Pix

Redefiniu-se os tipos PSR e restrição, de modo a serem representados através de *hashtables* em tipos que se esperava serem acedidos demasiadas vezes e que uma procura linear estaria a comprometer a performance da resolução. Assim, as listas de atribuições e a de domínios passaram a ser do novo tipo mencionado anteriormente. Com esta redefinição, acrescentou-se uma nova *hashtable* ao tipo PSR, a **restricoes_vars,** com o intuito de melhorar os tempos de pesquisa das restrições de cada variável.

Por fim, como a variável que guarda as inferências feitas pode ser constantemente utilizada em certos *puzzles*, optou-se por utilizar também uma *hashtable* para representar as inferências, antes guardadas apenas numa lista.

3.2 Criação/Combinação de Heurísticas

Com o objectivo de melhorar a performance do algoritmo de procura, obtou-se pela combinação das heurísticas MRV, *Minimum Remaining Value*, e de grau, de forma a não escolher varíaveis sem grande impacto para a resolução do problema enquanto não existem inferências. Assim, no ínicio da resolução, enquanto as inferências se encontram vazias e todos os domínios estão definidos como uma lista de valores 0 e 1, a heurística irá escolher variáveis com maior impacto na resolução do problema, isto é, variáveis envolvidas num maior número de variáveis. Há medida que o PSR é resolvido e as inferências são utilizadas, a heurística de grau deixa de ser utilizada uma vez que se dá privilégio à primeira variável que se encontre com domínio de tamanho um.

3.3 Utilização de técnicas adicionais

De modo a melhorar o desempenho do *resolve-best*, optou-se por correr um pré-processamento no PSR, depois deste ser criado. Primeiramente, durante a leitura do *array* dado como *input*, são guardadas as variáveis que participam em restrições triviais, ou seja, restrições do tipo 9 ou 0, ou 4, mas que estejam localizadas num canto, ou, por fim, um 6 que esteja presente numa borda. Com esta lista, são preenchidas todas as casas com os valores apropriados e que se sabe que estão corretos.

Depois desta rápida análise, volta-se a percorrer todas as restrições existentes no problema, analisando as variáveis que estão atribuídas a 0, a 1, ou que ainda não apresentem qualquer valor atribuído. Com estes dados, é possível completar certas restrições no PSR, e, voltando a correr todas as restrições, é possível, em alguns casos, resolver o problema na íntegra sem recorrer ao algoritmo de procura. Assim, com este *loop* a completar restrições que vão sendo consideradas triviais à medida que ele é executado, poupa-se um grande tempo e espaços de computação, dado que são menos variáveis com que a procura tem de manipular.

4 Estudo Comparativo

4.1 Critérios a analisar

Para comparar as diversas variantes dos algoritmos adotou-se como critérios o tempo de execução do programa para encontrar a solução pretendida, o espaço que a sua execução ocupou e a quantidade de testes de consistência efetuados.

Escolheu-se o tempo e o espaço de forma a analisar a variação das complexidades temporais e espaciais, respetivamente, com o crescimento das dimensões dos puzzles. Como é óbvio, será privilegiado o programa com menores complexidades temporais e espaciais, isto é, com um crescimento mais lento, e que executem em menores tempo e espaço.

Quanto à quantidade de testes de consistência efetuados, tem o propósito de entender o quão inteligente é o algoritmo, uma vez que se espera que o melhor algoritmo consiga encontrar a solução com o menor número de testes possíveis.

4.2 Testes Efectuados

Os problemas de fill-a-pix utilizados para testar o nosso algoritmo são os mesmo que foram disponibilizados aos alunos no ficheiro exemplos.lisp.

A sua escolha baseia-se no facto de, entre eles, ser possível testar diversas variantes de problemas fill-a-pix, sendo que uns têm diversas condições de restrições triviais, enquanto que outros não. Para além disso, a sua gama compreende puzzles com dimensões diversas. Assim, é possível comparar eficazmente os nossos critérios entre todas as variantes.

O puzzle e0 é um 3x3 apenas com uma restrição, possibilitando a observação de um número de testes mínimos para uma só restrição e a rapidez com que esta é completada.

No que toca aos puzzles e1 e e1_1, estes são 5x5 e, enquanto que o e1_1 tem três restrições triviais, as restrições do e1 só conseguem ser resolvidas testando todas as combinações das atribuições de 0 ou 1 às variáveis.

Os fill-a-pix e2, e3 e e4 são 10x10 e têm uma ligeira variância no que toca ao número de restrições e à forma como elas estão distanciadas no puzzle. O e2 tem 36 restrições, é o que as tem mais dispersas entre os três e tem poucas restrições triviais, sendo que, entre elas, o número dentro dos quadrados é relativamente baixo em média, perto de 4. Os puzzles e3 e e4 têm, respetivamente, 38 e 41 restrições, mas, ao contrário do e2, estas estão mais concentradas.

Quanto aos puzzles e5 e e6, estes são de 15x15, sendo que o e5 tem 102 restrições e o e6 84. No entanto, o segundo tem uma maior zona repleta de restrições com o valor 0, o que, naturalmente, irá conduzir a uma execução mais rápida do algoritmo de procura em comparação com o e5, uma vez que essas restrições são trivialmente completadas.

O puzzle e7 tem 121 restrições com dimensões 20x20. Apenas 13 dessas 121 são restrições triviais, portanto, este puzzle tem o propósito de avaliar, no geral, a performance das procuras.

4.3 Resultados Obtidos

As seguintes tabelas representam os resultados obtidos para as diversas variantes do algoritmo de resolução. A função usada para determinar o *run time* e o espaço ocupado pela execução da função de resolução foi a *time*.

Primeiro, foram efetuados testes com a procura cega, isto é, o algoritmo resolve-simples.

Puzzle	Tempo (s)	Espaço (Megabytes)	Número de Testes
e0	0.001898	0.05196	18
e1	0.009446	0.26622	67
e1_1	0.009551	0.3016	69
e2	172.66025	2 447.08845	440 475
e3	603.69275	5 377.0212	1 577 577
e4	1.836812	35.38514	4 873
e5	10 683.892	320 736.74738	7 582 175
e6	68.96207	249.15772	74 577
e7	5 177.8433	54 623.4678	1 872 827

Tabela 1 – Resultados do algoritmo resolve-simples.

De seguida, testou-se as procuras informadas com as diferentes heurísticas e métodos de propagação de inferências abordados no enunciado do projeto.

Puzzle	Tempo (s)	Espaço (Megabytes)	Número de Testes
e0	0.002699	0.05267	18
e1	0.016491	0.2718	60
e1_1	0.022764	0.31116	77
e2	4 651.6304	10 137.5979	1 897 405
e3	∞	∞	∞
e4	∞	∞	∞
e5	∞	∞	∞
e6	∞	∞	∞
e7	∞	∞	∞

Tabela 2 – Resultados do algoritmo com procura baseada na heurística de grau e sem método de propagação de inferências.

Dos testes e3 ao e7, como o algoritmo de procura demorou mais que três horas e meia a tentar resolver os ditos, atribui-se o valor de ∞ a todos os critérios. Esta má prestação levou a que a heurística de grau fosse logo posta de parte e, por isso, ignorada daqui em diante.

Puzzle	Tempo (s)	Espaço (Megabytes)	Número de Testes
e0	0.003334	0.08533	54
e1	0.03	0.49408	322
e1_1	0.033333	0.53557	294
e2	4.88	57.99437	11 423
e3	687.6566	3 512.43188	1 790 170
e4	1.860001	22.52174	4 301
e5	3 534.563	36 935.49267	2 355 841
e6	9.673332	125.77131	5 952
e7	2 751.2498	8 909.2487	664 327

Tabela 3 – Resultados do algoritmo com procura baseada na heurística de MRV e com método de *Forward Checking* para propagação das inferências.

Puzzle	Tempo (s)	Espaço (Megabytes)	Número de Testes
e0	0.003333	0.10749	110
e1	0.04	0.53918	438
e1_1	0.043334	0.70076	518
e2	9.256666	79.98289	23 183
e3	1 090.44	4 697.02829	2 979 354
e4	3.319999	31.33629	8 577
e5	7 222.1494	59 338.71766	5 700 974
e6	16.976665	170.54453	12 516
e7	3 782.0596	10 933.63696	1 058 684

Tabela 4 – Resultados do algoritmo com procura baseada na heurística de MRV e com método de MAC para propagação das inferências.

4.4 Comparação dos Resultados Obtidos

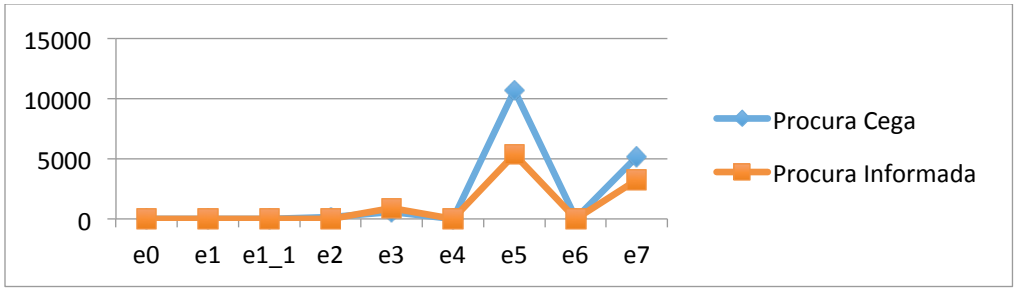
- Procura Cega vs. Procura Informada

Primeiramente, comparou-se as procuras Cega e Informada, sendo esta última apenas respeitante aos algoritmos com propagação de inferências, tal como evidenciado na secção anterior, a heurística de grau é ignorada.

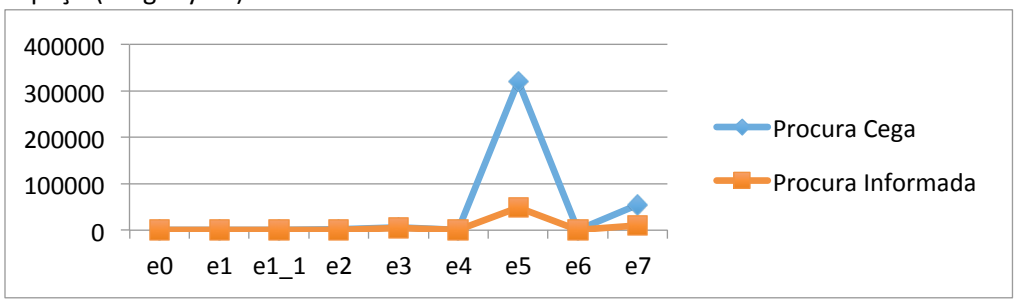
Verifica-se que a procura informada é mais rápida que a procura cega, bem como também ocupa menos espaço nos testes de maiores dimensões e dificuldade. Quanto ao número de testes, verifica-se que promovem a mais rápida chegada a um estado resultado, como acontece numa procura informada, existindo o corte de ramos da árvore de procura que sabemos que são maus logo à partida.

Também a escolha da próxima variável tem impacto, denotando a boa performance dos algoritmos informados.

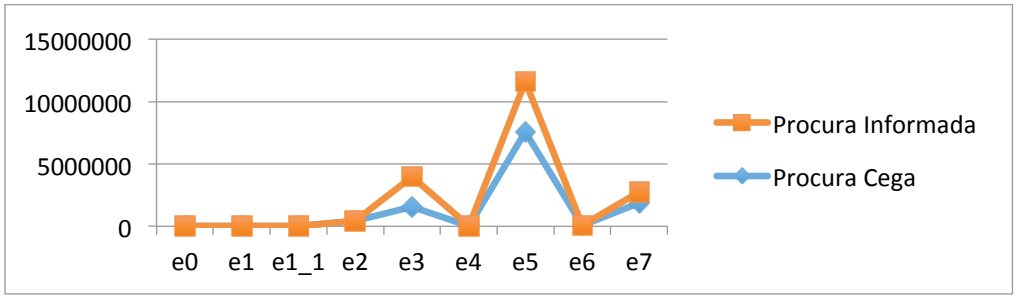
- Tempo (s)



- Espaço (Megabytes)



- Testes



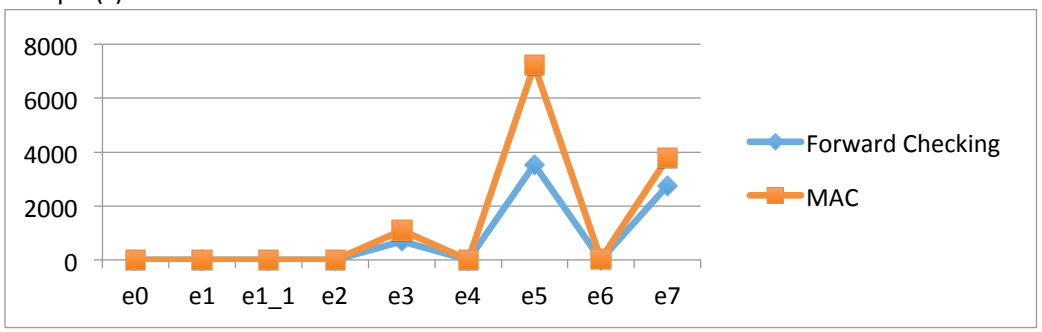
- Procura sem inferências vs. Procura com inferências

  Dado que a procura sem inferências com a heurística de grau apresentou resultados de tais dimensões exorbitantes, conclui-se que, com propagação de inferências, o algoritmo é mais eficiente. Dessa forma, não foram efetuadas quaisquer comparações entre as duas formas algorítmicas.

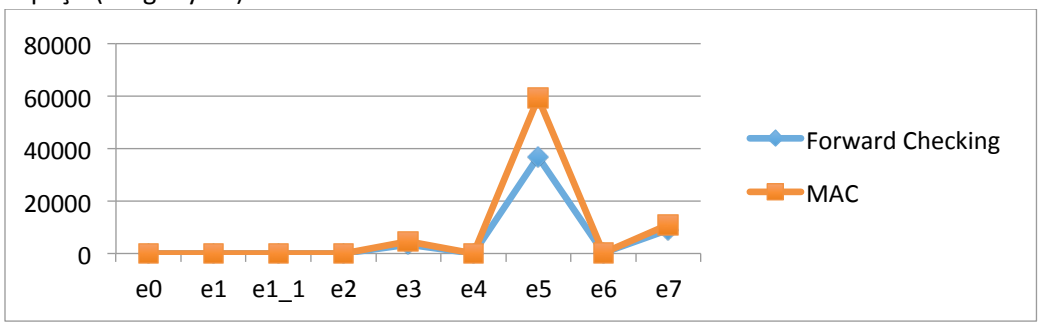
- Procura com propagação *Forward Checking* vs. Procura com propagação MAC

  Comparando os dois métodos de propagação de inferências, verifica-se que, nos casos dos puzzles que são mais fáceis de resolver, a performance de ambos os algoritmos é similar. No entanto, caso o caminho para a solução seja mais complexo e envolva mais testes, o FC tem melhores resultados temporais e espaciais. Apesar de o MAC fazer mais testes e assegurar uma maior consistência ao longo da execução do algoritmo, esta não compensa na complexidade temporal com que resolve o problema.

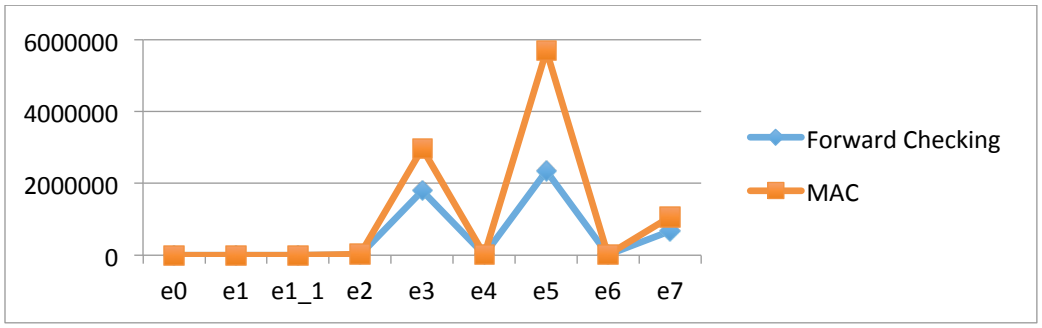
- Tempo (s)



- Espaço (Megabytes)



- Testes



4.5 Escolha do resolve-best

Após uma análise cuidada dos dados adquiridos, iniciou-se a escolha da heurística e do método de propagação de inferências para o algoritmo resolve-best.

No que toca à escolha da heurística, optou-se pela MRV, mas com uma variação: no caso de a heurística de MRV não der prioridade a qualquer variável, ou seja, no caso de elas todas terem o tamanho do domínio igual, a heurística em utilização passa a ser a de grau e, assim, é devolvida a variável que participa num maior número de restrições. Dado que o nosso algoritmo de procura recebe um problema PSR que já passou por um pré-processamento de completude de restrições trivialmente atribuídas, no início da execução do algoritmo, todas as variáveis não atribuídas têm domínio igual de tamanho dois. Só após serem feitas as primeiras inferências é que o domínio de algumas é reduzido para tamanho um e, a partir desse ponto, é a heurística MRV que prevalece.

Quanto ao método de propagação de inferências, escolheu-se o *Forward Checking*, uma vez que, em comparação com a execução quer espacial, quer temporal, quer em qualidade do número de testes, esta ganha sempre frente à MAC. Apesar de o MAC, à medida que se executa, ter uma performance mais sólida na procura da solução através do maior número de testes, o espaço e tempo que a função utiliza não compensam esta característica.

De forma a verificar os nossos resultados obtidos, apresenta-se de seguida a tabela com os resultados do algoritmo resolve-best em relação aos mesmos critérios designados no ponto 4.1.

Puzzle	Tempo (s)	Espaço (bytes)	Número de Testes
e0	4.63E-4	0.08427	0
e1	0.013333	0.37251	286
e1_1	0.001303	0.38268	0
e2	0.02	1.85264	2
e3	0.22	3.25977	4466
e4	0.02	2.07774	8
e5	0.056667	5.81944	0
e6	0.046667	4.88836	0
e7	1130.3466	4218.03393	5257610

Tabela 5 - Resultados do algoritmo resolve-best.

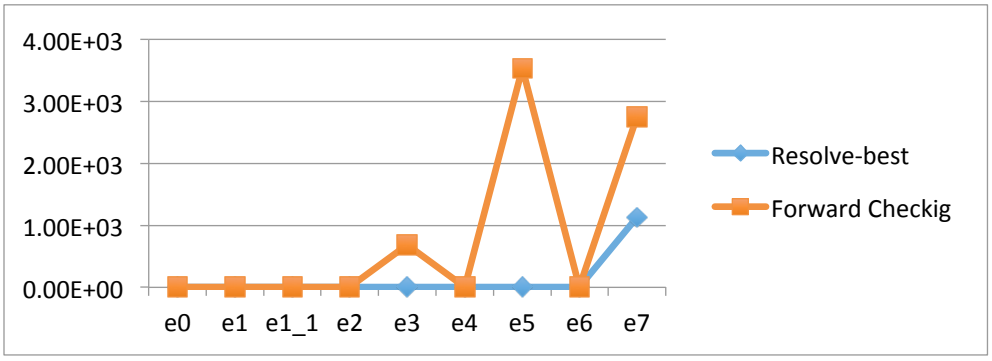
- Procura com propagação *Forward Checking* vs. Procura resolve-best

  De forma a testar o quão eficiente é o algoritmo resolve-best, escolheu-se fazer comparações entre o mesmo e o melhor algoritmo de procura desenvolvido na segunda parte do projeto, a procura com inferências por FC e heurística de MRV.

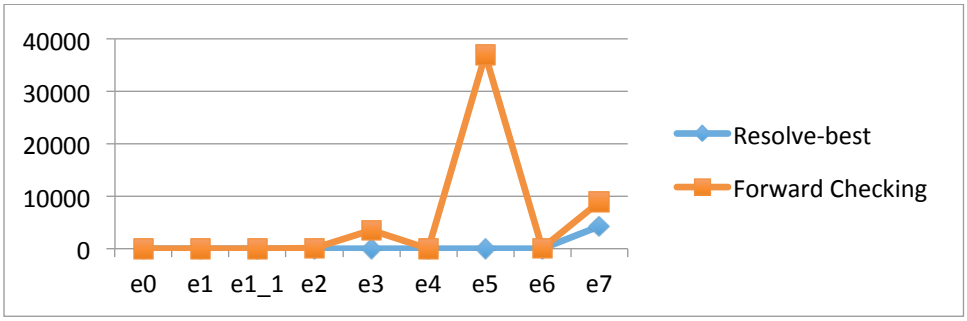
  Tal como esperado, o algoritmo *best* é extremamente melhor em todos os testes que o algoritmo por FC tem um pico no gráfico, tanto a nível temporal e espacial. Quanto a nível de testes de consistência, pelo facto de, tal como explicado anteriormente, o algoritmo usar a heurística de grau ao início verifica-se que este aumento do número de testes compensa a performance do algoritmo, visto que a nível temporal, o resolve-best leva metade do tempo.

  Devido ao pré-processamento que o *array* sofre antes mesmo de ser alvo da execução do resolve-best, existem mesmo puzzles que são trivialmente completados com um número mínimo de testes de consistência. No caso do e7, o pico de execução do resolve-best explica-se devido ao aumento exponencial do número de variáveis a considerar no puzzle e pelo facto de serem poucas as restrições que podem ser trivialmente satisfeitas logo de início.

  - Tempo (s)



- Espaço (Megabytes)



- Testes