Resolução do Puzzle de Magellán

Pedro Vieira de Castro, João David Gonçalves Baião
[1] Faculdade de Engenharia da Universidade do Porto, Rua Roberto Frias 4200-465, Portugal

FEUP-PLOG, Grupo Magellan_1

Resumo. Implementação de um algoritmo que resolva o puzzle de Magellán e puzzles aleatoriamente criados com base na mesma lógica no âmbito do paradigma de Programação em Lógica com Restrições.

1 Introdução

O objetivo do segundo projeto no âmbito da cadeira de Programação em Lógica é o desenvolvimento de um programa em Programação em Lógica com Restrições cujo seu propósito é resolver um dos problemas de otimização ou decisão combinatória sugeridos no enunciado.

Decidimos escolher o puzzle Magellán. O puzzle original consiste numa caixa em forma de paralelepípedo (como um VHS) em que as duas maiores faces da caixa estão divididas em regiões de diferentes cores. O objetivo é distribuir as cores de modo a que nenhuma região adjacente tenha a mesma cor. No puzzle físico, estas cores são escolhidas com rodas que têm as cores nelas indicadas.

Para dificultar um pouco mais algumas destas rodas são visíveis de ambos os lados da caixa. Deste modo, a escolha da cor de algumas regiões de um lado da caixa bloqueiam as do outro e vice-versa.

A principal razão pela qual escolhemos este puzzle foi a intriga e a procura do desafio que o puzzle parecia oferecer. Ao analisarmos brevemente foi nos claro que estávamos deparados com uma aplicação prática do ***Four Colour Theorem*.**

O ***Four Colour Theorem*** diz que: "*Dado um mapa plano, dividido em regiões, quatro cores são suficientes para colori-lo de forma a que regiões vizinhas não partilhem a mesma cor.*".

O teorema afirma também que regiões que só partilhem um ponto não são consideradas vizinhas, o que vai de acordo com o que acontece no puzzle.

2 Descrição do Problema

O puzzle Magellan consiste num retângulo com 33 regiões coloridas e numeradas. A parte frontal do retângulo está numerada de 1 a 33, enquanto a parte traseira está numerada de 34 a 66. Cada região tem uma roda de 6 faces, que pode ser girada, estando cada face da roda colorida de diferentes formas.

O objetivo do puzzle é ter todas as regiões com diferentes cores nas suas proximidades. O puzzle original é suposto ser resolvido usando as 6 cores, ou menos. No entanto, decidimos ter 2 opções diferentes iniciais, resolver o puzzle original, ou criar um puzzle novo e resolvê-lo também.

Estas regiões podem ser então analisadas como um grafo. Cada nó será uma região e as arestas do grafo vão representar a adjacência dessa região em relação a outras.

É possível otimizar o resultado, usando menos cores. Como referido acima, como o Teorema das Quatro Cores assim postula, é possível colorir todos os mapas usando apenas 4 cores, não havendo cores repetidas nas imediações. Por vezes será também possível colorir com menos de 4 cores.

3 Abordagem

3.1 Variáveis de Decisão

A única variável de decisão que usamos é *ListNodes*, a lista de nós, em que o valor de cada nó representa a cor da região, tal como no tabuleiro original. O seu domínio é uma das 6 cores possíveis, branco, azul, verde, amarelo, vermelho e preto.

3.2 Restrições

Usamos 2 restrições, sendo a primeira para definir o facto de nós adjacentes não poderem ter cores iguais, através do uso do predicado **restrictColorsOfGraph (ListNodes, List, Term)**. Aqui restringe-se então que se existe uma adjacência entre dois nós então esses dois nós têm que ter um valor para a cor diferente.

A segunda restrição tem base no facto de o puzzle ter uma componente frontal e traseira. Tendo isto em conta, existem rodas que interagem com ambas as faces e, por exemplo, se a roda for [Amarelo, Azul, Verde, Branco, Preto, Vermelho] e no tabuleiro frontal estiver a cor Azul na região da roda, então na face traseira terá obrigatoriamente a cor Preto.

Isto foi implementado no predicado **restrictDoubleWheels(ListNodes, List)**.

3.3 Função de Avaliação

Na implementação da nossa solução não criamos uma função de avaliação, não vendo necessidade na altura da implementação da mesma.

3.4 Estratégia de Pesquisa

Utilizamos labeling nos seguintes predicados: **originalGame(Type, TimeOut)** e **randomGraphing(ListNodes, Type, TimeOut)**. O **originalGame** faz o labeling para o puzzle original e o **randomGraphing** para os puzzles gerados aleatoriamente.

Foram implementados menus que permitem adicionar diferentes opções ao labeling. Para além dos resultados obtidos apresentados no ponto 5 deste artigo, também chegamos às seguintes conclusões:

Quanto ao resultado obtido a única opção que influencia a **qualidade** é a minimização do valor de **nvalue** da lista de nós. O predicado **nvalue** recebe uma lista de variáveis de domínio e devolve o número de valores diferentes dessa lista. Ao minimizar o número de valores diferentes é possível chegar a uma solução ótima que usa o menor número de cores possível.

Quanto à rapidez a opção que torna a busca do resultado mais rápido é o **ffc.** Ao usar **ffc** o *labeling* vai procurar atribuir valores às variáveis com menor domínio primeiro de modo a reduzir o número de impraticabilidades que depois seriam introduzidas.

Em termos de validade das soluções que são apresentadas, todas estão corretas. A única variação é a cor atribuída.

Como ponto adicional que nós achamos que traria mais valor ao projeto, decidimos implementar uma função de seleção de variáveis. O critério desta função seria atribuir o *labeling* a uma variável aleatória. Esta função foi puramente experimental não tendo qualquer uso exceto no estudo do *backtracking* efetuado pelo *labeling* no caso das anteriormente faladas impraticabilidades.

4 Visualização da Solução

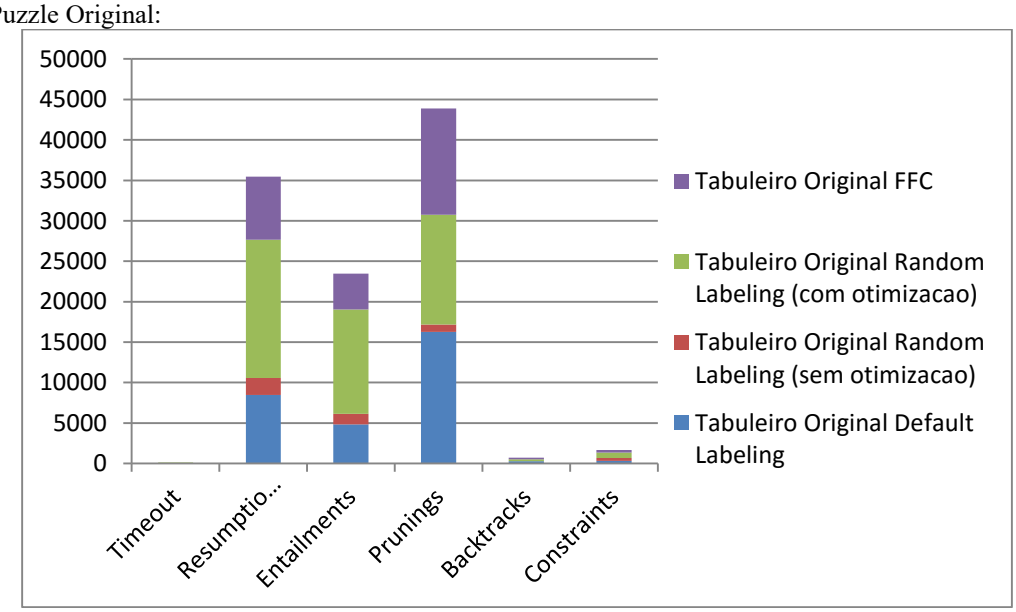
A solução é visualizada recorrendo a vários predicados que mostram a alteração nas regiões coloridas e a estatística do *labeling* usado. Estes predicados são:

- **fd_statistics**: Este predicado é nativo de SICStus PROLOG e imprime o número de vezes que uma restrição foi resumida, o número de vezes que uma restrição detetou um *entailment*, o número de vezes que o domínio foi "cortado", o número de *backtrackings* feito e o número de restrições criado.
- **printGraph(ListNodes, List, Term)**: Este predicado chama recursivamente o predicado **printNode(ListNodes, List, Index, Term)** que chama por sua vez o predicado **printRelation(ListNode, List, Node, SecondNode, Index)**.
  O predicado **printRelation** imprime as cores de duas regiões adjacentes. É possível visualizar que a cor dos nós é sempre diferente.

5 Resultados

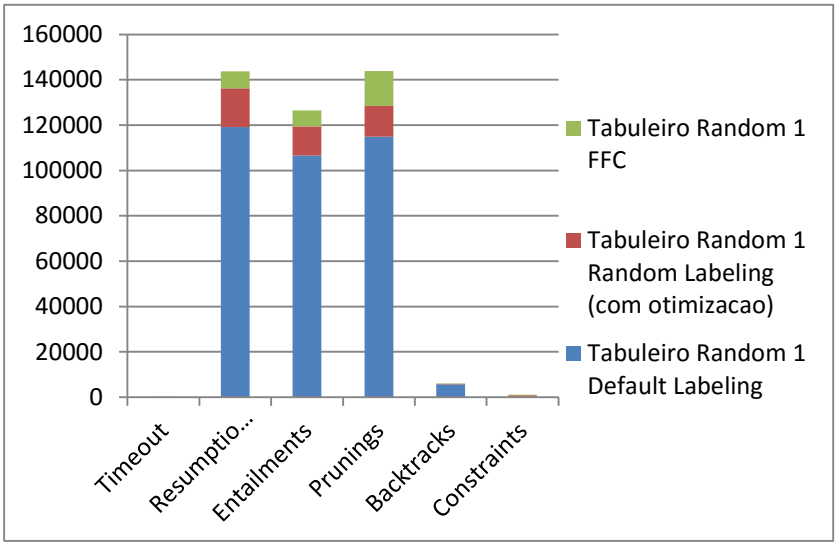
Todas as estatísticas foram geradas através do uso da nossa aplicação com *timeout* de 100 e não é possível guardar um puzzle *random*, logo, sempre que chamamos a aplicação para um puzzle *random*, é gerado um *random* novo.

Puzzle Original:

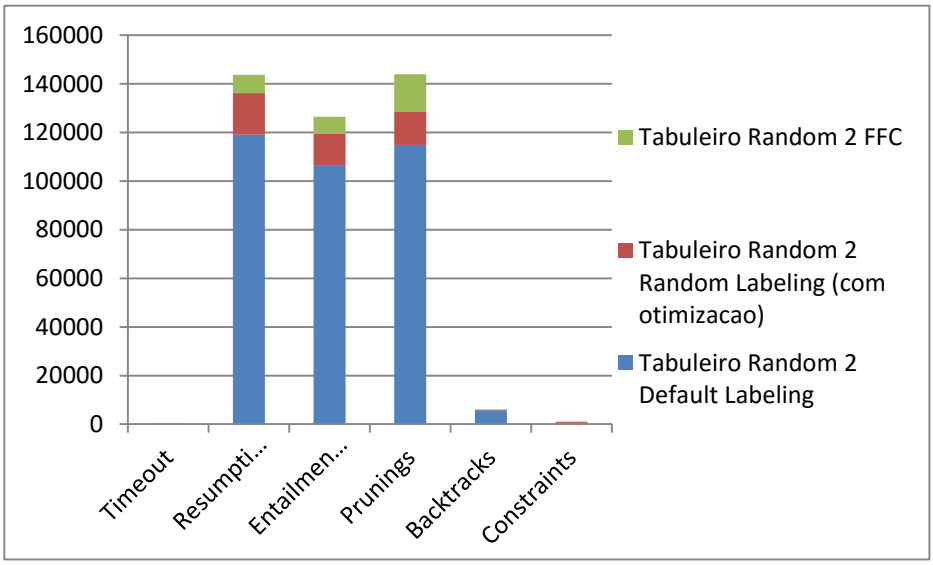


Resolução do Puzzle Magellan

Puzzle 1:



Puzzle 2:



Após a análise das estatísticas geradas variadas vezes, concluímos que usando um *labeling* definido o programa corre sem problemas e rapidamente, virtualmente sem *timeout*, mesmo com otimizações. No entanto, usando *default labeling* ou até mesmo um puzzle diferente do puzzle original, a aplicação pode demorar imenso tempo até arranjar uma solução e, por exemplo, usando *random labeling* num puzzle *random* e pedindo otimizações, deixamos a aplicação a correr durante a noite e passado 8 horas, ainda estava a correr.

6 Conclusões e Trabalho Futuro

Analisando o projeto que desenvolvemos, apercebemo-nos das possibilidades que o uso de restrições e etiquetamento (*labeling*) nos oferecem, tornando a solução de certos problemas muito mais fácil de alcançar.

Olhando para a solução que apresentamos, reparamos que tem alguns fatores limitantes. Por exemplo, ao criar um puzzle *random* não é tido em conta o facto de poder existir uma face traseira e uma frontal, mas estando implementado no puzzle original.

Por outro lado, o facto do grafo criado a partir do puzzle *random* poder não ser planar (um grafo planar é aquele que pode ser desenhado num plano de forma a que as adjacências não se cruzem), é possível não existir solução, visto que assim o **Teorema das Quatro Cores** não se aplica.

Contudo, não existem apenas limitações. O facto de termos implementado o *random labeling*, mesmo não sendo pedido, é um fator de valorização e uma vantagem. Desenvolvemos também a possibilidade de resolver o problema de forma otimizada (usando ou não as 6 cores disponíveis, como visto no Teorema das Quatro Cores).

Em conclusão, se pudéssemos alterar mais algo no nosso trabalho, teríamos adicionado a possibilidade de guardar os tabuleiros que geramos aleatoriamente. Além disso, implementávamos um tabuleiro frontal e um traseiro ao gerar um puzzle *random*.

Bibliografia

1. http://www.jaapsch.net/puzzles/magellan.htm - Magellán
2. https://www.puzzlemaster.ca/browse/wood/european/3386-magellan-puzzle - Magellán
3. https://sicstus.sics.se/sicstus/docs/latest4/html/sicstus.html/ - SICStus Documentation
4. https://pt.wikipedia.org/wiki/Grafo_planar - Grafo Planar
5. http://www.britannica.com/biography/Henry-Dudeney/images-videos/dudeney-henry-gas-water-electricity-problem/91872 - Grafo Planar
6. http://projecteuclid.org/euclid.ijm/1256049011 - Teorema das Quatro Cores
7. http://research.microsoft.com/en-us/um/people/gonthier/4colproof.pdf - Teorema das Quatro Cores

Anexo

Em anexo a este ficheiro está incluído o código fonte.