Universidade do Algarve
Faculdade de Ciências e Tecnologias
Departamento de Engenharia Eletrónica e Informática

Relatório Técnico-Cientifico

Problem 1: Troika`s Cube

Adriano Fernandes Vaz, nº 51542
Natáliya Mykytas, nº52192

Relatório submetido no âmbito da
Disciplina de Inteligência Artificial
Professor Fernando Lobo

Abril de 2017

Índice

Introdução

Com este relatório pretende-se demonstrar de uma forma clara e sucinta o problema *"Troika`s Cube"*, descrever como foi interpretado, a maneira como foi solucionado e as metodologias abordadas.

O problema consiste em calcular a solução para o cubo no formato 3x3x1 num mínimo número de passos possível. Para calcular a solução mínima para este problema, precisamos de representar o cubo pelas suas cores. As cores que existem no cubo são as seguintes: Verde (G), Branco (W), Amarelo (Y), Vermelho (R), Laranja (O) e Azul (B). Introduzindo um conjunto de cores aleatório, seguindo a representação de 5 linhas por 8 colunas, o programa tem que verificar todas as combinações possíveis, geradas pelos movimentos, e retornar o número mínimo de passos necessários para a resolução do cubo.

O Cubo será representado por uma matriz com letras, onde cada letra corresponderá a uma cor. A matriz será 5x8, mas apenas imprimirá as cores, não imprimindo as posições vazias. Isto pode ser visualizado de um ponto de vista gráfico, como na seguinte Figura 1:

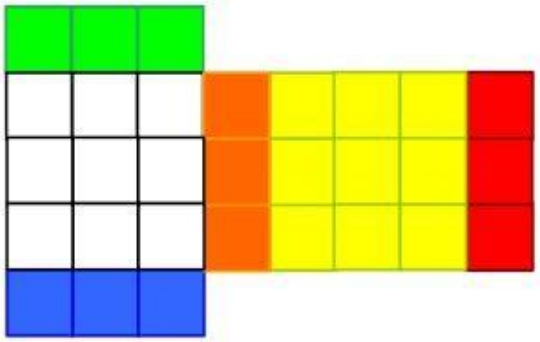

Figura 1 – Exemplo de uma solução para o Cubo

A solução do cubo consiste, em que cada face (laterais, superior e inferior) possua a mesma cor em todos os "blocos".
Esta solução só pode ser obtida, girando em torno de 180º em 6 combinações distintas. As 6 combinações possíveis são as seguintes (baseando na figura 1, facilitando assim a explicação):

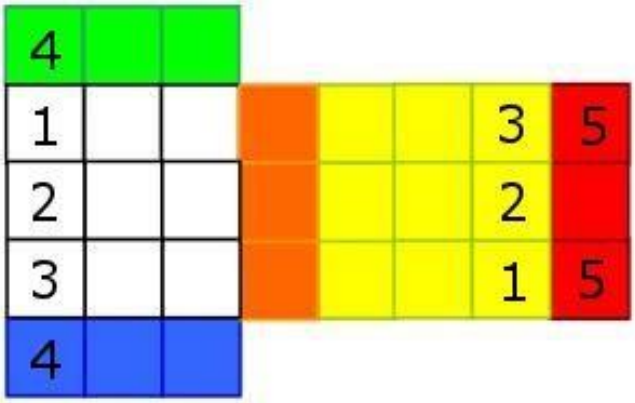

Figura 2 - Rotação Vertical à Esquerda

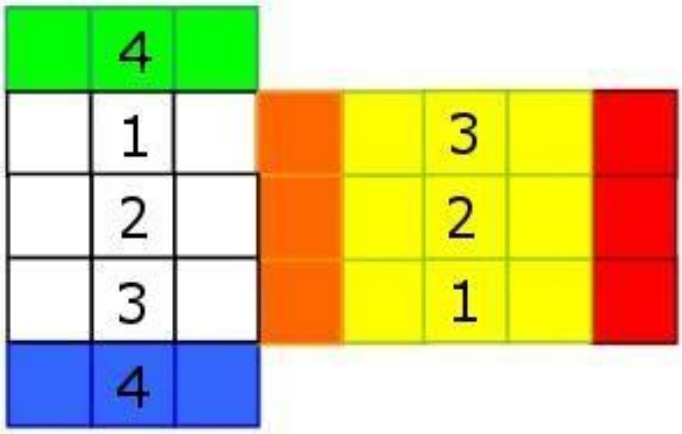

Figura 3 - Rotação Vertical ao Meio

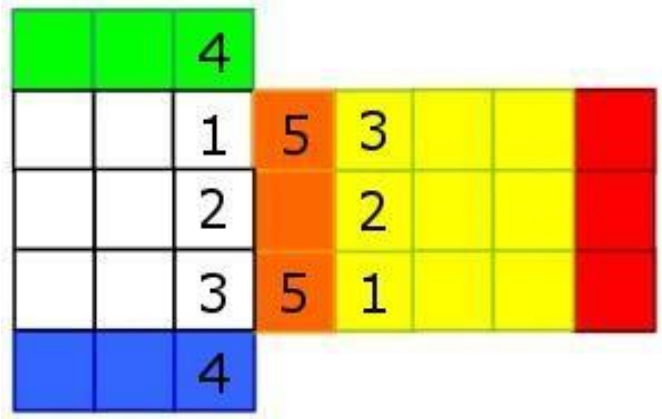

Figura 4 – Rotação Vertical à Direita

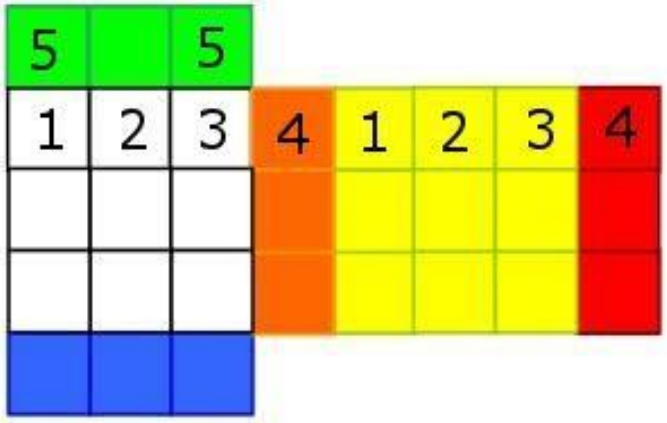

Figura 5 - Rotação Horizontal em Cima

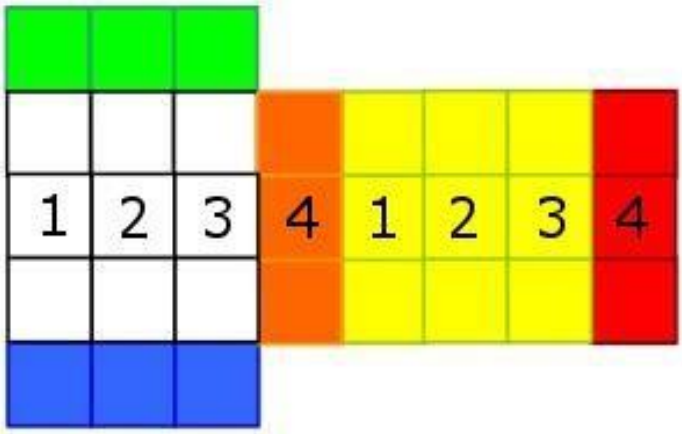

Figura 6 - Rotação Horizontal no Meio

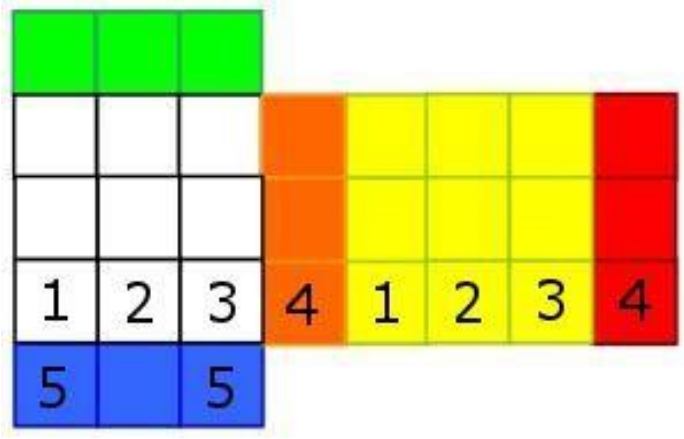

Figura 7 - Rotação Horizontal em Baixo

Os esquemas anteriores podem ser explicados da seguinte maneira: as cores que vão trocar entre si, estão numeradas de 1-4/5 (consoante o movimento). Por exemplo os 1's trocam entre si, os 2's entre si, etc. Sabendo-se o método de troca de cores, a resolução deste problema fica bastante facilitada, visto que só será preciso ter a preocupação em gerar as sucessivas combinações, através dos diferentes movimentos, transformando assim uma dada sequência de cores numa possível solução.

Uma instância do problema pode ser representada pela seguinte figura:

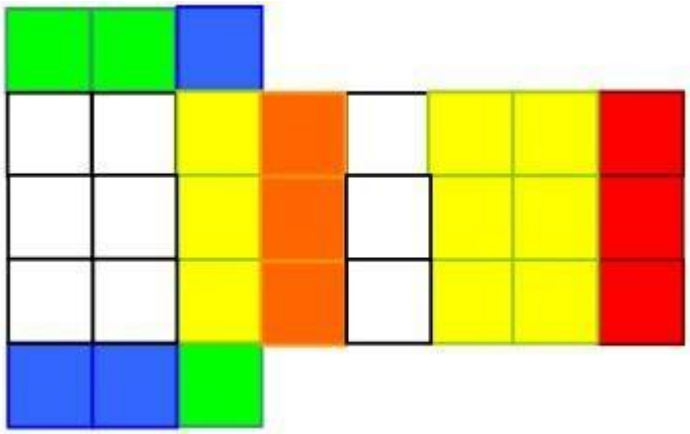

Figura 8 – Exemplo de uma instância do problema

O objetivo é dado um esquema do cubo que é introduzido pelo utilizador, calcular o número mínimo de passos necessários para obter uma possível solução do *Troika`s Cube*, usando a heurística apropriada. O resultado será um número inteiro que representa o número mínimo de passos necessários para obter a solução.

Neste caso específico do cubo 3x3x1, o *God's number* é 8. O *God's number* consiste no maior número de passos que qualquer configuração pode necessitar para atingir a configuração resolvida. Já o máximo número de configurações possíveis é 192, obtido através da seguinte equação:

$$\frac{2^4 \times 4!}{2} = 192 \qquad \textit{(Equação 1)}$$

A relação de configurações possíveis - passos necessários para atingir a configuração desejada é dada na seguinte Tabela 1:

n	0	1	2	3	4	5	6	7	8	**Total**
p	1	4	10	24	53	64	31	4	1	**192**

Tabela 1 - Número de configurações p que requerem n passos para atingir a configuração "resolvida".

UML

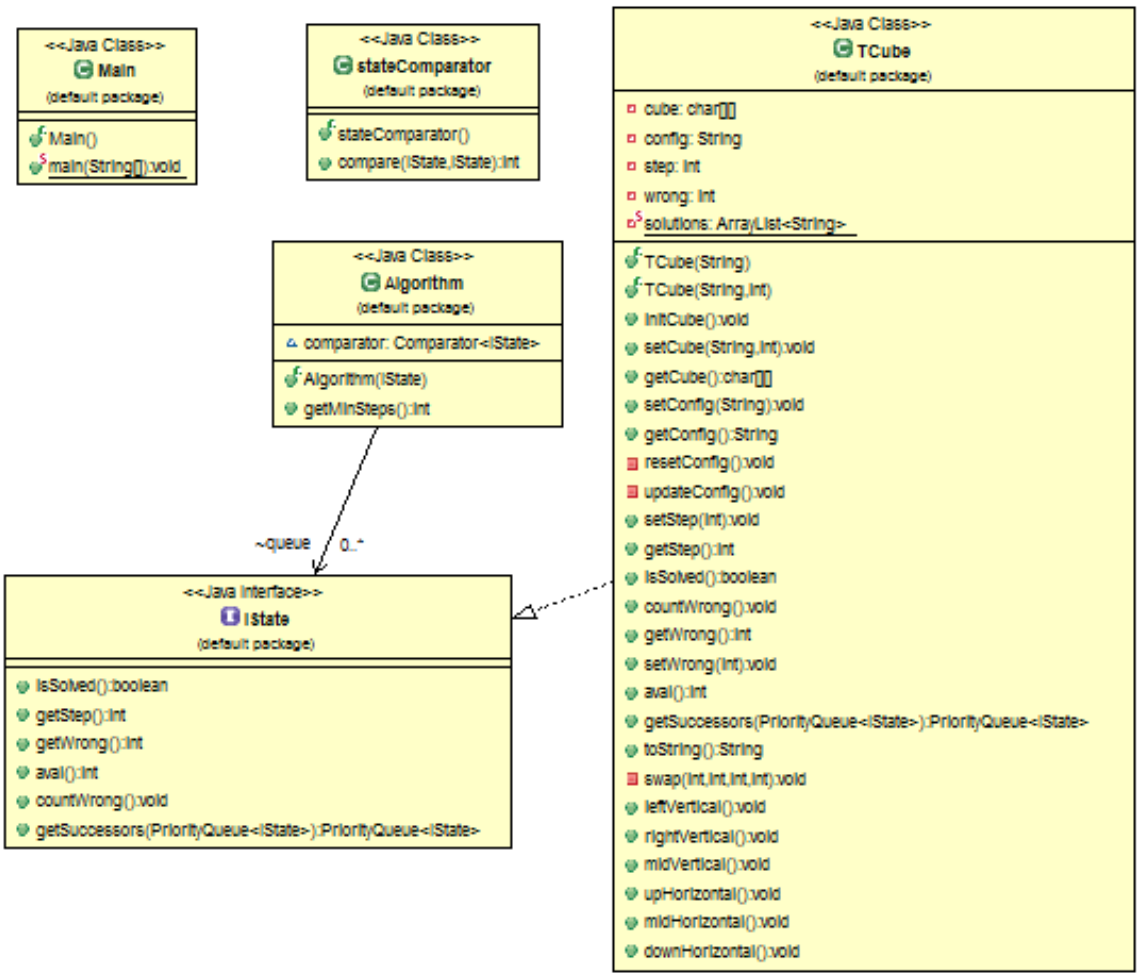

Figura 9 - Diagrama do Problema em UML

Ao interpretar o enunciado do problema *"Troika`s Cube"* conseguiu-se identificar que classes poderiam ser necessárias para a sua resolução.

Analisando concluímos que iriam ser precisas as classes *Algorithm e TCube.* Ao efetuar uma segunda leitura do problema, com mais atenção e cuidado, observamos que independentemente do problema apresentado, o algoritmo criado para calcular o número mínimo de passos para obter a configuração desejada, logo este teria de ser o mais geral possível. Por essa razão, seria necessário implementar uma interface. A interface garante com que qualquer outra classe possa utilizar o algoritmo desde de que a implemente e instancie os métodos que lhe estão associados. Outro problema que surgiu durante a análise do enunciado esteve relacionado com o tipo de estrutura de dados que melhor representaria a sequência de cores do cubo.

Para resolver a primeira questão decidiu-se criar uma interface, chamada *IState*. Esta

interface, fica responsável por implementar os seguintes métodos: *isSolved(), getWrong(), getStep(), getSuccessors() e countWrong()* .O método *isSolved(),* neste caso específico, tem como função verificar se uma determinada sequência de cores do cubo está correta, é um método do tipo booleano que retorna a condição *true* se a sequência em que estamos a trabalhar é igual a uma das soluções possíveis para o problema. O método *getSuccessors()* tem como função criar os cubos sucessores, com todos os movimentos possíveis e adiciona-los todos à *PriorityQueue*. O método *getStep()* devolve o número de movimentos necessários para chegar ao estado atual em que está o cubo. O método *getWrong(),*é o que cálcula a heuristica, que será explicada mais a seguir. E por fim o método *countWrong()* tem como função verificar quantas linhas incorretas tem o cubo atual.

A classe *TCube* constrói um esquema do cubo através de uma *String*, que contêm a sequência de cores introduzida pelo utilizador. Esta classe é responsável por implementar os métodos que estão na interface *IState* e os métodos que permitem realizar a rotação dos componentes do cubo.

A classe *Algorithm* é responsável por calcular o número mínimo de passos necessários para obter uma possível solução e enviar esse valor, do tipo inteiro, para a classe *Main*, que irá escrever esse valor no *output.*

A classe *StateComparator* é responsável por determinar qual o estado que terá prioridade sobre os outros, usando a heurística, em que o aval() é a soma do passo atual + número de erradas e que é utilizado para definir os estados que têm prioridade na *PriorityQueue*.

A nossa heurística verifica a face frontal do cubo e assim verifica quantas colunas não têm as 3 cores iguais. É também verificado se a parte superior e esquerda do cubo se encontram corretas. Não há necessidade de ver as partes opostas do cubo visto que têm exatamente o mesmo número de colunas erradas e o movimento corrige ambas. Por cada coluna errada acrescenta-se um movimento necessário para o número total de movimentos estimados necessários para a solução do cubo. Achamos que fosse a mais correta e eficiente, pois so verifica uma metade do cubo, o que é mais rápido que verificar o cubo inteiro.

Durante a análise do problema houve alguma dificuldade em perceber qual a melhor estrutura de dados a utilizar para a sua resolução. Após alguma discussão interna acerca do assunto decidiu-se colocar a sequência de cores do cubo, numa matriz de caracteres.

A matriz de caracteres pareceu-nos a melhor opção, porque permite uma melhor manipulação das posições do cubo facilitando a implementação dos métodos responsáveis pela rotação dos diversos componentes do cubo. Esta estrutura de dados, neste problema em concreto, poderá desperdiçar memória por haver alguns espaços vazios. No entanto, a facilidade de manipulação compensa essa desvantagem que a

nosso ver, não tem qualquer tipo de repercussão no resultado final, pois essas posições mantêm-se intocáveis.

Para explicar melhor como foi feita a matriz de caracteres, ver na figura apresentada abaixo:

Figura 10 Matriz de caracteres

Metodologias

Inicialmente houve uma indecisão relativamente à utilização de uma classe abstrata e uma interface.
A metodologia adotada pelo nosso grupo consiste em criar uma interface *IState*, a qual contém vários métodos, que poderão ser utilizados por qualquer outra classe, que a implemente. A interface *IState*, contêm os seguintes métodos: *isSolved(), getWrong() e getStep(), getSuccessors() e countWrong().* No caso do nosso problema, a classe *TCube*, tem de implementar estes métodos para que seja possível calcular o número mínimo de passos, de modo a resolver uma dada sequência do cubo. Decidimos seguir esta metodologia de criar uma interface, pois pretendemos partilhar os métodos *isSolved(), getWrong(), getStep(), getSuccessors() e countWrong(),* entre várias subclasses que poderão não ter nada de semelhante entre si, como por exemplo o problema do carteiro chinês e este problema do cubo 3x3x1.
A vantagem da utilização de uma destas abordagens é principalmente a partilha de métodos entre várias subclasses.

Visto que a resolução do cubo, depende o número de rotações necessárias, na classe *TCube*, estão implementados todos os métodos responsáveis pela rotação Os métodos que permitem efetuar as rotações do cubo são os seguintes: *leftVertical(), rightVertical(), midVertical(), upHorizontal(), midHorizontal() e downHorizontal()*. Como o nome dos métodos indica, cada um fica responsável pelos movimentos do cubo, através do método de troca de posições das cores *swap(int, int, int, int)*. Dentro de cada método a rotação é feita por uma simples troca de elementos da matriz que contêm uma dada sequência. Depois de efetuada a troca a sequência gerada pelo método é guardada no mesmo objeto da classe que posteriormente irá ser utilizado no método *getSuccessors()*. Este método é responsável por criar novas sequências. Que será utilizado pela classe *Algorithm* para gerar configurações até encontrar a configuração final através do método *isSolved().*

Já nesta classe *Algorithm*, classe universal para todo o tipo de problemas idênticos a este (*i.e.* que pretendam chegar a um determinado estado final), possui apenas um método, que é o *getMinSteps()*. Este método percorre uma *PriorityQueue* com os sucessores da configuração.

Durante a implementação da resolução do problema, foram criados os métodos *setConfig(), updateConfig(), resetConfig() e swap()*. Estes métodos têm como objetivo melhorar a utilização de código evitando acesso indevido as variáveis de instância por parte dos métodos que controlam a rotação dos vários componentes do cubo.

Para além do mencionado, durante a resolução do problema houve necessidade de efetuar uma generalização na criação dos métodos e algoritmos, em particular no código que está presente na classe *Algortihm*. Este código tem de ser o mais geral possível para

que outras classes que implementem a interface *IState*, possam a utilizar. A solução encontrada para o problema tenta ser a mais genérica possível de modo a responder as situações dadas no enunciado do trabalho, bem como a outras possíveis situações que poderão ser introduzidas pelo utilizador.

Algoritmo Utilizado

O algoritmo utilizado para resolver este problema consiste no seguinte:

1 – Lê-se uma dada sequência de cores existente no cubo. Coloca-sea sequência numa *String* que mais tarde irá ser convertida em matriz de 5 linhas com 8 colunas. As posições que não tem qualquer cor são consideradas com o valor 0.

2 – Instancia-se um objeto da classe *TCube*, com a sequência introduzida pelo utilizador e automaticamente são efetuadas as rotações necessárias.

> 2.1 – Existem 6 rotações disponíveis e uma rotação é uma simples troca de elementos da matriz onde está representada a sequência de cores possíveis do cubo.

3 – No método *getSuccessors()*, são criados 6 estados com a mesma configuração que o pai e com o mesmo movimento que o pai +1. Depois de ser inicializado é realizada um rotação unica por cubo, criando assim um outro estado para cada sucessores do pai e depois de cada rotação, o estado é adicionado à *PriorityQueue*, que já foi explicado anteriormente

4 - Gerados os sucessores, estes seguirão para uma *PriorityQueue*. O objetivo desta *Queue* é juntar os sucessores que foram anteriormente criados de modo a facilitar a procura da solução.

5 – Verifica-se na *PriorityQueue* qual das sequências geradas anteriormente é uma possível solução para o problema, ou seja que tem o menor número de linhas erradas. Enquanto a solução não é encontrada é incrementado um contador de passos cada vez que uma nova rotação é efetuada.

6 – Quando uma possível solução para o problema for encontrada, é retornado o resultado do contador que está no formato do tipo Inteiro.

Effective Branching Factor

O effective branching factor é um valor que determina em média quantos nós sucessores são gerados por cada nó. Como pedido calculamos o effective branching factor, para a nossa implementação do problema com heurísticas e sem heurísticas. Primeiro imprimimos o número de nós abertos em para o input 3, cujos números são 331 e 13620, do programa com heurísticas e sem, respetivamente.

Usando a formula:

x^0+x^1+x^2+x^3+x^4+x^5+x^6=331 ⇔ x= 2.40566

x^0+x^1+x^2+x^3+x^4+x^5+x^6=13620 ⇔ x= 4.69566

Sendo x o effective branching factor do programa, respetivamente, com heurísticas e sem heurísticas, podemos verificar que usando heurísticas são abertos muito menos nós, o que torna o algoritmo muito mais rápido ao procurar a solução com menor custo.

Conclusão

Os objetivos alcançados, com a realização deste trabalho prático foram os seguintes: 1) Quais são as heurísticas que existem e como as implementar.2) Criar um algoritmo que consiga responder ao problema o mais geral possível tendo em conta a sua utilização em outras situações aproveitando a reutilização de código. 3) Aprender o que é o effective branching factor, e como calcular.

Embora utilizar uma heurística nem sempre seja um método perfeito ou otimizado, é indicado para soluções que sejam imediatas. Com uma heurística é possível "conduzir" o programa de forma a obter resultados mais satisfatórios, de maneira mais rápida. No entanto, para criar uma heurística é necessário ter experiência em resolver o problema, muitas vezes por tentativa-erro.

Referências

I. https://en.wikipedia.org/wiki/Floppy_Cube
II. http://www.jaapsch.net/puzzles/floppy.htm
III. https://ruwix.com/twisty-puzzles/super-floppy-cube/
IV. https://en.wikipedia.org/wiki/God's_algorithm
V. http://ozark.hendrix.edu/~ferrer/courses/335/f11/lectures/effective-branching.htm