Trabalho de I.A.

Heurísticas para o Cubo Mágico

Disciplina: I.A./COS/UFRJ

Alunos:

Ricardo Fernandes Ribeiro
Paulo Coelho Ventura Pinto

Professora:

Inês Dutra

1) Introdução

Muitos problemas de importância prática e teórica possuem uma natureza combinatória. Os problemas combinatórios são intrigantes, pois muitos deles são fáceis de se definir, mas muitos deles são difíceis de serem resolvidos. Em geral, eles são problemas NP-hard.

O cubo mágico (ou cubo de Rubik) e jogo dos 8 (8-Puzzle) e a sua generalização (n^2-1)-Puzzle (ou jogo dos (n^2-1)) são problemas de natureza combinatória.

Esse dois puzzles serão utilizados para ilustrar o projeto de funções heurísticas mais acuradas que permitem encontrar, num tempo razoável, pela primeira, vez a solução ótima para o cubo mágico e o jogo dos 24 para inúmeras instâncias desses problemas.

Também será apresentada uma teoria que permite prever o tempo de execução de uma heurística admissível a partir de uma profundidade solução e da função heurística.

1.2) Cubo Mágico

O cubo mágico é um puzzle em forma de cubo. É conhecido também como cubo de Rubik, o seu inventor. Inventado em 1974 pelo húngaro Erno Rubik. Vendeu mais de 100 milhões de exemplares pelo mundo. É o puzzle combinatório mais famoso de todos os tempos.

O problema é difícil: o slogan nas embalagens comerciais diz que há bilhões de combinações possíveis, o que é um valor subestimado: Há mais de 4.3252×10^{19} diferentes configurações a partir da configuração inicial (o cubo com todas as faces com as mesmas cores).

Consiste de 27 cubículos que juntos formam um cubo 3x3x3:

Figura 1.1

O objetivo do jogo é, partindo de uma configuração válida qualquer, chegar a uma onde cada face possua somente uma cor:

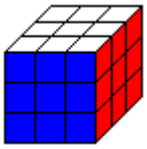


Figura 1.2

Existem configurações que são trivialmente inválidas, porém outras não. Isso se deve ao fato de que todo espaço do problema, do ponto de vista combinatorial, estar dividido em 12 sub-grafos isomórficos, onde o seu conjunto de vértices (estados configurações possíveis) é disjunto para cada sub-grafo dois a dois. Em outras palavras, nenhum movimento legal, dentro de uma configuração pertencente a um sub-grafo, levaria à outra configuração pertencente a outro sub-grafo. Esse fato é análogo ao que ocorre no jogo dos 15, onde há dois sub-grafos isomórficos separados.

Pode-se ainda visualizar a existência de cubos maiores como 5x5x5 e 10x10x10. Há o caso menor: 2x2x2.

Exemplos de cubos NxNxN

2x2x2 3x3x3 4x4x4 5x5x5

Figura 1.3

1.3) Regras de movimentação

- Cada movimento pode ser interpretado como uma rotação de elementos cúbicos.
- Cada rotação, de 90º, é aplicada em um conjunto de cubículos que estejam no mesmo plano.
- O eixo de rotação é sempre perpendicular a uma das faces.

Número de configurações possíveis

- É fatorial em N:

Nome Popular	*Instância*	*Número de configurações possíveis*
Rubik's Wahn	**5x5x5**	2.8 x $\mathbf{10}^{74}$
Rubik's Revenge	**4x4x4**	7.4 x $\mathbf{10}^{45}$
Rubik's Cube	**3x3x3**	4.3 x $\mathbf{10}^{19}$
Pocket Cube	**2x2x2**	3.6 x $\mathbf{10}^{6}$

Nome Popular	*Instância*	*Branching factor sem refinamentos*
Rubik's Wahn	**5x5x5**	30
Rubik's Revenge	**4x4x4**	24
Rubik's Cube	**3x3x3**	18
Pocket Cube	**2x2x2**	12

1.4) O Cubo de Rubik como um Problema Computacional

O Cubo de Rubik é considerado no campo da I.A. como um toy *benchmark*. Ele é usado para testar a eficiência de novas técnicas, possuindo características que são um teste de fogo:

- Um espaço de busca gigantesco;
- Cada movimento envolve mais de uma peça – o que não o corre na família do jogo dos 8, que também é uma família de *toy benchmarks* em IA;

Acredita-se que qualquer solução ótima para uma dada configuração possa ser encontrada em, no máximo, 20 movimentos e que a solução média seja de 18 movimentos. Essas são conjecturas presentes nos trabalhos do professor Richard Korf, que foi o primeiro a descobrir a primeira técnica viável para a busca de soluções ótimas para o Cubo de Rubik.

O método desenvolvido por Korf, usando a técnica de *patterns databases* e o *algoritmo de busca IDA**, foi capaz de encontrar a solução ótima para 10 cubos bem "embaralhados" (100 movimentos aleatórios de um cubo na configuração objetivo):

- Uma em 16 movimentos;
- Três em 17 movimentos;
- Seis em 18 movimentos;

1.5) Algoritmos de Busca

Em especial, nos deteremos no *algoritmo IDA**, que é a versão de espaço linear do *algoritmo A**. Esses dois algoritmos se valem de uma função heurística – h(n) – que estima o custo real de se alcançar uma solução a partir de uma determinada configuração. Se h(n) é admissível então ela não superestima o custo real e, assim, esses algoritmos sempre retornam a solução ótima, caso ela exista.

NxN Jogo dos N^2-1	**Ordem do Número de Estados**
3x3 Jogo dos 8	10^{5}
4x4 Jogo dos 15	10^{13}
5x5 Jogo dos 24	10^{25}

A função heurística clássica para a família do jogo dos 8 é a distância Manhattan. Ela é computada olhando-se cada uma das peças: contando-se o número de unidades da grade entre a sua posição atual e a posição objetivo da peça (mas sempre por movimentos horizontais e verticais) e, finalmente, somando-se todos esses resultados para todos as peças do jogo.

Essa heurística é um ***limite inferior*** no tamanho da solução ótima real, pois cada pedra deve ser movida a uma distância menor do que a sua distância Manhattan e cada movimento deve mover somente uma única pedra. Devido a essas características, a distância Manhattan é uma *heurísticas admissíveis*.

Infelizmente o *algoritmo A** não consegue resolver o Jogo dos 15, porque ele armazena cada nó gerado, exaurindo a memória disponível na maioria dos problemas antes de achar uma solução.

O *Interactive-Deepening A* (IDA*)* é uma versão do *algoritmo de busca A**, mas com complexidade de espaço linear na profundidade de busca. Assim como o A*, ele garante achar a solução ótima, se a função heurística usada for admissível. Usando a distância Manhattan, ele foi o primeiro algoritmo a encontrar a solução ótima para o jogo dos 15. Uma média 400 milhões de nós por segundo são gerados para cada instância do problema, requerendo 6 horas de tempo de execução em 1985.

2) Projeto de Funções Heurísticas com Patterns Databases

[Korf 2000] [KorfFel2000]

Uma explicação para a existência das funções heurísticas é que elas computam o custo exato da solução para uma versão mais relaxada ou simplificada do problema. Por exemplo, nos jogos do tipo do Jogo dos 8, se forem ignoradas as restrições de que só podemos mover uma peça para uma posição vazia, teremos um novo problema, onde todas as demais pedras podem ser movidas para qualquer direção, mesmo que ocorra uma sobreposição. O custo da solução ótima é exatamente da distância Manhattan de todas as pedras. É importante considerar que uma solução na instância inicial do problema é também uma solução da versão simplificada e, portanto, o custo da solução na versão simplificada é um limite inferior para o custo da solução ótima. A solução para o problema inicial é difícil porque as pedras interagem entre sim e as restrições de movimentações são grandes, o que não ocorre com a versão simplificada do problema. A principal desvantagem da heurística distância Manhattan é não "capturar" essas interações no cálculo da função heurística para uma determinada configuração de peças.

2.1) Pattern Databases

O conceito de *pattern databases* busca capturar algumas das interações entre as peças perdidas na distância Manhattan. Esses "bancos de dados" seriam usados para calcular uma estimativa da profundidade da solução ótima: um valor heurístico. O conceito de *pattern databases* surgiu através dos trabalhos de Culberson e Schaeffer . Por definição, uma função não precisa ser definida por uma equação algébrica ou algoritmo, mas sim por um tipo especial de relação. Algumas funções possuem um domínio finito e, portanto, caso haja disponibilidade de memória, os valores do seu conjunto-imagem podem ser armazenados em uma *look-up table*.

A idéia por trás dos pattern databases é pré-computar o custo da "solução" de alguns sub-padrões de uma determinada configuração e usar esses valores para computar uma função heurística mais acurada.

Os *pattern databases* podem ser classificados como:

- Não-Aditivos
- Aditivos
- Disjuntos

O primeiro tipo de *pattern database* na literatura foi o não-aditivo. Iremos apresentar cada um deles através dos problemas do jogo das peças deslizantes. Depois, esses conceitos serão aplicados ao cubo mágico em seções mais a frentes.

2.1.1) Pattern Databases Não-Aditivos

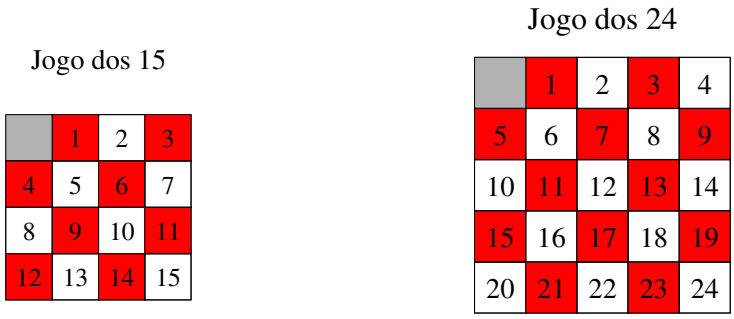


Figura 2.1

Considere um subconjunto de peças, tal que as sete pedras encontrem-se na coluna mais a direita e na linha mais abaixo no Jogo dos 15, quando estas estão na configuração solução. Esse subconjunto é chamado de *padrão crista* (fringe pattern). Suponha que temos agora uma instancia não-objetivo: o número mínimo de movimentos necessários para mover as peças da crista para sua posição objetivo, incluindo qualquer outra peça fora da crista, é, obviamente, um limite inferior no número mínimo de movimentações para se resolver o problema todo. Esse número é então calculado para todas as permutações possíveis das peças oriundas da crista. Ele depende somente das posições das peças de crista e da posição sem peça, e as demais peças são consideradas equivalentes para um padrão de crista específico. Como o número total de configurações possíveis é 16!/(16-8)!, ou mais explicitamente 518.918.400, e como um único byte é suficiente para contar o número de movimentações, teremos então, em memória, uma tabela com menos de 500 Mb.

Peças do Padrão Crista no Jogo dos 15

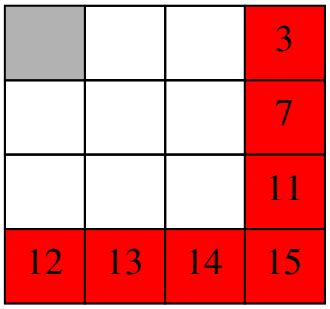


Figura. 2.2

Essa tabela é computada em uma única busca em profundidade, a partir da configuração objetivo. Nessa busca, as peças fora do padrão são equivalentes e um estado é identificado unicamente pela posição das peças do *padrão crista* e da região sem peça. Na primeira ocorrência do padrão, o custo é armazenando numa entrada da *look-up table*.

É importante lembrar que a construção do *pattern database* é realizada uma única vez: o custo da sua construção é amortizado na solução de várias instâncias do problema com o mesmo objetivo.

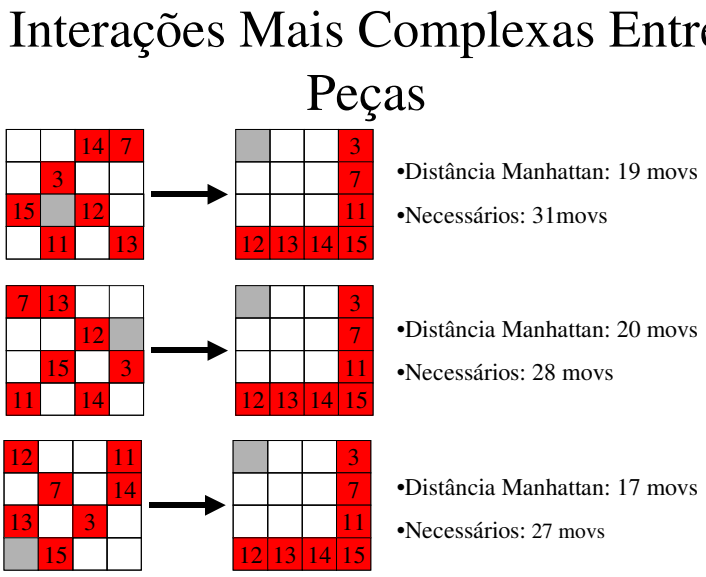


Figura. 2.3

Uma vez que o banco foi construído, o *algoritmo IDA** é utilizado para buscar a solução ótima para uma instância do problema. A função heurística pode buscar o máximo entre o valor de uma entrada na tabela (indexado por um padrão crista) e a distância Manhattan para todas as pedras fora do padrão da instância.

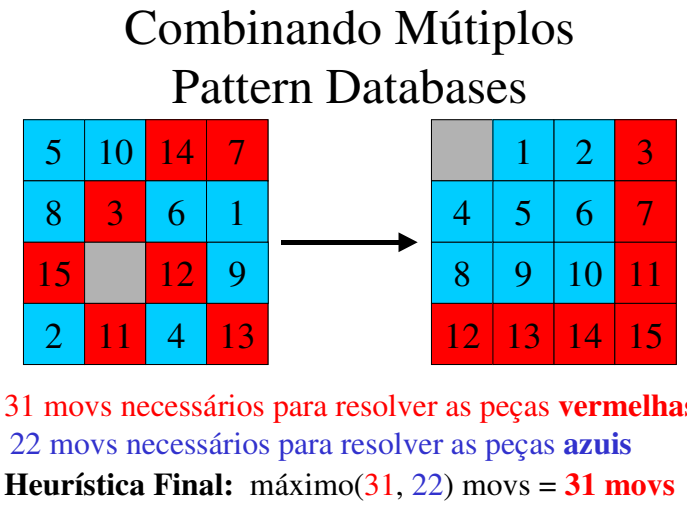


Figura. 2.4

É importante lembrar que não há impedimento em se ter outros *pattern databases* para o cálculo da função heurística, como, por exemplo, o máximo de um conjunto de *pattern databases*, pois o próprio *pattern database* é uma função heurística admissível.

Infelizmente não é permitido somar os valores da função heurística, pois corre-se o risco de contabilizar a movimentação de uma pedra mais de uma vez. Uma forma de impedir esse problema e melhorar a performance dos *pattern databases* será vista a seguir.

2.1.2) Pattern Databases Disjuntos

Uma forma para se tentar encontrar uma função heurística que combine o resultado de dois ou mais *pattern databases* sem superestimar o valor da função heurística, pode ser somando-se o valor das entradas dois *pattern databases* distintos, porém para esse procedimento resultar em uma função admissível, duas condições são necessárias:

- o conjunto peças de cada p*attern database* deve ser disjunto dois a dois;
- cada *pattern database* pode conter somente o número de movimentações das peças do conjunto.

Um exemplo trivial de *pattern database disjunto (ou database disjunto)* é a distância Manhattan. Cada distância Manhattan de uma peça pode ser vista como a soma de um conjunto individual de valores pattern databases, onde cada conjunto contém exatamente uma peça.

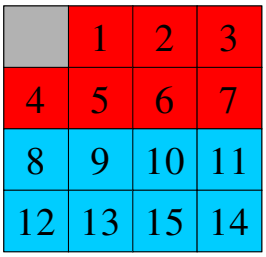
Um exemplo não trivial de *database disjunto* é dividir o Jogo dos 15 em duas metades horizontais num grupo de sete peças na parte de acima e oito peças na parte de baixo, assumindo que a posição sem peça fique no canto alto esquerdo para o estado objetivo.

O número de movimentos necessários para resolver cada peça em cada padrão a partir de todas as combinações possíveis são pré-computados, mas apenas contando a movimentação das pedras pertencentes ao padrão. Ao invés de explicitar a posição sem pedra, é armazenado o mínimo possível para todas as localizações possíveis da posição sem peça.

O database para o padrão das oito peças contém 16!/(16-8)! = 518.918.400 entradas. Para o das sete peças contém 16!/(16-7)! = 57.657.600 entradas. Como são necessários 4 bits para armazenar o número máximo de movimentos para cada padrão, 495Mb e 55 Mb de memória serão necessários, respectivamente, para armazenar cada tabela.

*Uma **regra geral** para particionar os conjuntos de peças é grupar as peças que estejam numa mesma região na configuração objetivo. Dessa forma, aumenta-se a probabilidade de a função heurística captura de interações entre peças.*

Exemplo Databases Aditivos - 1



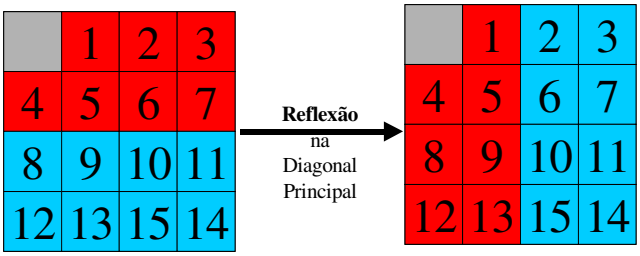
O database das 7-peças vermelhas contém **58 milhões** de entradas.

O database das 8-peças azuis contém **519 milhões** de entradas.

Figura. 2.5

Uma vez que os *databases disjuntos* acima foram computados, é possível se construir um novos databases tomando-se como base a reflexão das partições na diagonal principal do primeiro database. Então, uma nova função heurística é calculada a partir desses dois conjuntos de partições, como mostrado abaixo:

Exemplo Databases Aditivos - 2



Particionamento 1 Particionamento 2

Heurística Final = Máximo(h_1(**P**), h_2 (**P**))

OBS:

h_1 : heurística para ***particionamento 1*** computada através ***Database Aditivo 1***

h_2 : heurística para ***particionamento 2*** computada através ***Database Aditivo 2***

Figura. 2.6

Entretanto, seus valores somente podem ser combinados a partir dos seus valores máximos, pois os seus conjuntos de peças não são disjuntos.

Exemplo Databases Aditivos - 3

Partionamento para o Jogo dos 24

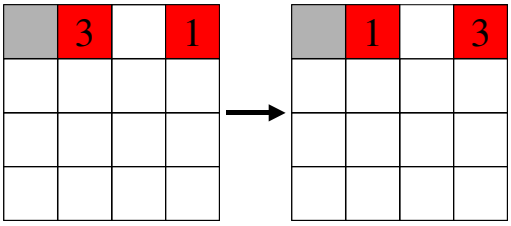
Figura. 2.7

2.1.3) Databases Pareados

Consideremos agora um *database* que contenha o número de movimentos necessários para corretamente posicionar cada par de peças a partir de qualquer posição possível. Esse valor conta que as duas peças consideradas não podem ocupar o mesmo lugar no espaço. Na maioria dos casos, o valor em uma entrada desse database será a soma da distância Manhattan, mas em alguns casos a excederá. Por exemplo, se duas peças estiverem na mesma linha, que também é a linha objetivo das peças, e ocupando a posição objetivo uma da outra. Para que as peças possam ser movimentadas para as suas posições objetivos, uma delas terá que dar passagem a outra para que o número de movimentos seja mínimo. Isso adiciona dois movimentos à soma das distâncias Manhattan.

Figura. 2.8

Exemplo: Conflito Linear



Distância Manhattan é 2+2 = 4 movs

Heurística com Conflito Linear é 2+2+2 = 6 movs

Exemplo de Dinâmica do Conflito Linear

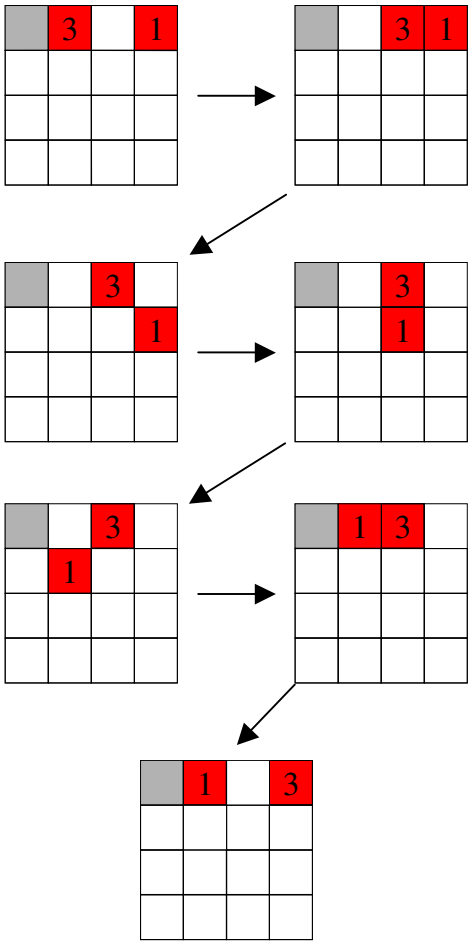


Figura. 2.9

A esses valores - o número de movimentos necessários para um par de peças ir para a sua posição correta - são chamados de *distância pareada*.

Seja *n* o número de pedras do jogo pedras, o tamanho de tabela construída seria da ordem de $O(n^4)$. Como nos demais *pattern databases*, esta tabela é computada uma única vez.

Seja uma tabela com a distância pareada e uma configuração de peças do jogo, a computação do valor da função heurística não deve somente somar os valores dos pares de peças que satisfaçam a configuração em questão, pois, certamente, a movimentação de uma determinada peça será contabilizada mais de uma vez, o que tornará a função heurística não admissível. Para evitar a perda da admissibilidade da função heurística, as n peças são particionadas em n/2 pares que não se sobreponham. A pergunta que entra em cena é: que pares de peças escolher? O número de partições possíveis com essa propriedade é da ordem de $\mathbf{O}(n!/(2^{n/2}(n/2)!))$. Na figura abaixo temos calculado o número exato de partições para os diferentes dos jogos de peças deslizantes:

JOGO	Número de Partições
3	3,0E+00
8	1,1E+02
15	2,0E+06
24	3,2E+11
35	1,6E+20
48	1,2E+30

O ideal é escolher os pares que maximizem o valor da função heurística. Uma forma de enxergar o problema é criar um grafo onde cada vértice é uma peça e cada aresta é uma aresta com um peso correspondente a distância pareada. O problema então se resume em achar um conjunto de arestas onde duas arestas escolhidas não incidam sobre o mesmo vértice e que a soma de seus pesos seja máxima. Esse problema de grafos é conhecido como *maximal weighted matching problem*, e pode ser resolvido em tempo polinomial $O(n^3)$, onde n é o número de vértices.

Esta técnica apresentada pode ser estendida para triplas de peças ou n-tuplas de maior ordem. Infelizmente para as triplas, esse problema é análogo ao *tree-dimensional matching problem* que é NP-completo, assim como os problemas de *matching* de ordens superiores.

Uma vantagem desse método é que ele produz valores heurísticos mais acurados e, portanto, menor geração de nós, comparado aos *databases disjuntos*. A razão para isso é que quando duas peças estão localizadas em partições diferentes de um *database disjunto,* suas interações não são capturadas, o que não ocorre em um database pareado. A sua principal desvantagem é o *overhead* computacional causado pela necessidade da resolução de um problema de *matching* para cada nó na busca (ou seja um pequeno problema de otimização para cada nó onde se deseja calcular o valor da função heurística). O que é muito mais custoso do que somar ou comparar valores obtidos de um database para cada grupo de peças.

3) Pattern Databases Aplicado ao Cubo Mágico

[Korf 1997]

Cubo Mágico
Cubo de Rubik

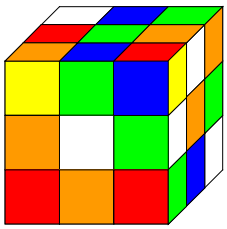


Figura. 3.1

Vamos apresentar a primeira técnica eficaz que obtém soluções ótimas para instâncias aleatórias para o cubo mágico 3x3x3, sendo que a média das mesmas aparenta estar em 18 movimentos.

Ela utiliza o algoritmo de busca IDA* com uma função heurística baseada em tabelas armazenadas em memória, também conhecidas como *pattern databases*. Essas tabelas contêm o número exato de movimentos requeridos para resolver vários sub-objetivos do problema, neste caso, o número de movimentos que solucionam subconjuntos de peças do cubo.

3.1) Cubo Mágico

O problema do *cubo mágico* 3x3x3 é extremamente difícil, dado que existem 4,3252 x 10^{19} estados diferentes que podem ser atingidos a partir de um determinada configuração. Para efeito de comparação, o jogo dos 15 possui 10^{13} estados, e o jogo dos 24 possui 10^{25} estados.

Para se resolver o *cubo mágico*, necessita-se de uma estratégia geral, que consiste em um conjunto de seqüências de movimentos, ou macro-operadores, que corrigem as posições de peças sem violar as que já estão devidamente posicionadas. Existem algoritmos que acham uma solução qualquer para o Cubo Mágico, porém, em geral, elas possuem comprimentos bem elevados (70 a 150 movimentos em média), se comparados os valores previstos pela conjetura do Professor Richard Korf: *qualquer instância aleatória encontra-se no máximo a 20 movimentações da configuração solução*.

3.2) O Espaço do problema

O primeiro passo é traduzir o jogo físico em um problema simbólico para ser manipulado por um computador. Dos 27 cubos, 26 são visíveis, com um no centro. Destes:

- Os seis cubos, um em cada centro de cada face rodam, mas não se movem, formando uma referência fixa, impedindo rotações do cubo inteiro.
- Oito cubos estão nos cantos, cada um com três faces visíveis.
- Doze cubos estão nas arestas, cada um com duas faces visíveis.

A próxima questão é definir os operadores primitivos. Embora, a princípio, as rotações de todas as fileiras de cubos, em qualquer sentido sejam válidas, muitas destas são redundantes. Por exemplo, se uma determinada fileira de cubos for movida em uma direção e, em seguida, a mesma fileira for movida no sentido oposto, teremos a configuração anterior de volta. *Tomando-se o cuidado de evitar estes movimentos redundantes, pode-se reduzir o fator médio de ramificação do Cubo Mágico de **18** para **13,34847***.

3.3) Uma primeira abordagem

Para se encontrar soluções ótimas, é necessário um algoritmo admissível de busca. Algoritmos com complexidade exponencial de espaço, como A* são impraticáveis em problemas grandes. O mais apropriado é o uso do IDA*, que executa ma busca em profundidade que por soluções com custos cada vez mais elevados, utilizando uma heurística para podar nós que ultrapassem o custo limite da iteração corrente.

Profundidade	Nós
1	18
2	243
3	$3{,}24 \times 10^{3}$
4	$4{,}325 \times 10^{4}$
5	$5{,}773 \times 10^{5}$
6	7,707 x 106
7	$1{,}029 \times 10^{8}$
8	$1{,}373 \times 10^{9}$
9	$1{,}833 \times 10^{10}$
10	$2{,}447 \times 10^{11}$
11	$3{,}266 \times 10^{12}$
12	$4{,}36 \times 10^{13}$
13	$5{,}82 \times 10^{14}$
14	$7{,}768 \times 10^{15}$
15	$1{,}037 \times 10^{17}$
16	$1{,}384 \times 10^{18}$
17	$1{,}848 \times 10^{19}$
18	$2{,}466 \times 10^{20}$

Então, precisa-se de uma função heurística. A heurística óbvia é a versão tridimensional da distância de Manhattan. Porém, após a mesma ser corrigida, para tornar-se admissível, o valor esperado da mesma é de 5,5, o que dá uma medida de sua eficácia, como será visto posteriormente.

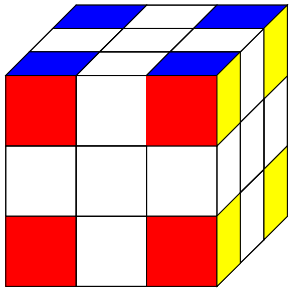
O algorítmo IDA*, executado em uma máquina Sun Ultra-Sparc Modelo 1, com esta heurística, pode procurar no nível 14 em cerca de três dias, mas resolver instâncias do problema no nível 18 levaria mais de 250 anos. Precisa-se de uma heurística melhor.

3.4) Pattern Databases

Enquanto normalmente pensamos em uma heurística como uma função computada por um algoritmo, esta mesma função pode ser armazenada em uma tabela, se houver memória suficiente. De fato, por razões de eficiência, funções heurísticas são geralmente pré-computadas e armazenadas em memória. Por exemplo, a distância de Manhattan acima pode ser computada com a ajuda de uma pequena tabela que contém as distâncias de Manhattan de cada peça individual para todas suas possíveis posições e orientações.

Se considerarmos apenas os oito cubos do canto, a posição e orientação da última peça podem ser determinadas pelas demais sete, existindo portanto $8! \times 3^7 = 88.179.840$ combinações possíveis. Usando uma busca em largura a partir do objetivo, é possível enumerar estes estados e armazenar em uma tabela o número de movimentos requeridos para resolver cada combinação de cubos nos cantos. Como estes valores vão de 0 a 11, são necessários quatro bits para cada entrada. Assim, esta tabela necessita de 44.089.920 bytes, ou 42 MB.

Pattern Database para os 8 Cubinhos do Cantos



Este database contém 88 milhões de entradas

Figura. 3.2

O valor esperado desta heurística é de 8,764, comparado com o 5,5 da distância de Manhattan. Durante a busca IDA* em cada estado gerado, um índice único é gerado, seguido de uma referência à tabela. O número de movimentos necessários para resolver os cubos dos cantos é recuperado, sendo um limite mínimo do número de movimentos para resolver o jogo inteiro.

Pode-se melhorar esta heurística, considerando-se os cubos das arestas. Nesse caso, os doze cubos das arestas são divididos em dois grupos de seis cubos. O número de combinações possíveis para cada grupo é de (12! / 6!) x 2^6 = 42.577.920. O número de movimentos necessários para resolver estes conjuntos varia de zero a dez, com um valor esperado de cerca de 7,668 movimentos. Utilizando-se quatro bits para cada entrada, esta tabela requer 21.288.960 bytes (~ 20 MB).

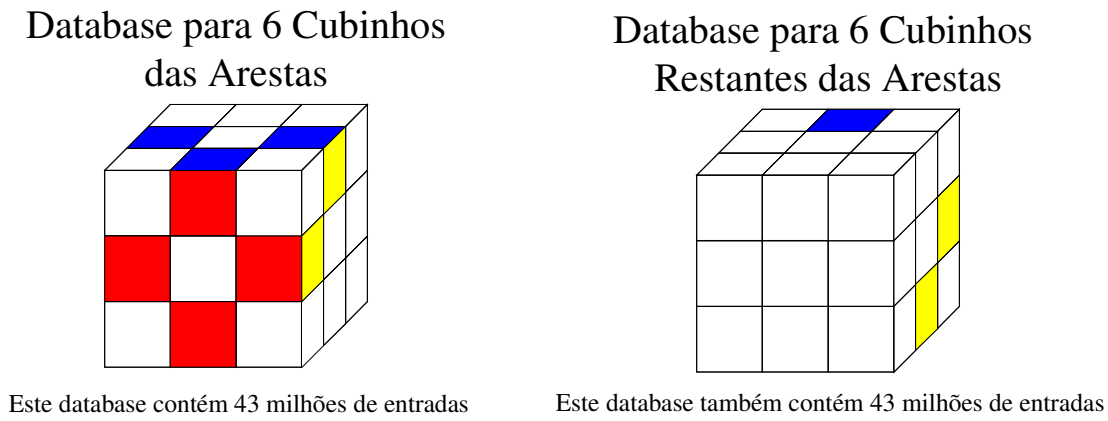


Figura. 3.3

O total requerido de memória para as três tabelas é de 82 MB, sendo que o tempo para gerar as tabelas das três heurísticas é de cerca de uma hora, sendo este custo amortizado para várias consultas com o mesmo estado final.

A única maneira admissível de combinar as heurísticas do canto e das arestas é, após a avaliação das mesmas, tomar o valor máximo. Em média, este valor é de 8,878. Embora este seja apenas um pouco superior à média de 8,764 obtida com a avaliação apenas dos cubos dos cantos, isto resulta em um aumento significativo do desempenho do algoritmo.

3.5) Resultados Experimentais

Foram geradas dez instâncias do Cubo Mágico, fazendo-se 100 movimentos randômicos em cada, partindo-se do estado final. Dado que cada estado gera 18 filhos, e com a conjectura de que o diâmetro do espaço de busca não ultrapassa 20 movimentos, acredita-se que a realização de 100 movimentos randômicos gera, efetivamente, instâncias aleatórias do problema.

Foi utilizado o algoritmo IDA* com a heurística descrita anteriormente, resolvendo todos os problemas de maneira ótima. Um dos problemas foi resolvido em 16 movimentos, com a geração de 3,72 bilhões de nós. Três foram resolvidos em 17 movimentos e os demais seis foram resolvidos em 18 movimentos, sendo que, no pior caso, foram gerados 1,02 trilhão de nós.

Isto foi feito em uma máquina Sun Ultra-Sparc Model 1, capaz de processar 700.000 nós por segundo, em 1997. Foi verificado que, em altas profundidades, o número de nós gerado por iteração é bastante estável para diferentes instâncias do problema. Por exemplo, nas seis iterações completadas no nível 17, o número de nós gerados variaram apenas de 116 a 127 bilhões de nós.

Buscas completas no nível 16 requerem uma média de 9,5 bilhões de nós, levando menos de quatro horas. Buscas completas no nível 17 geram uma média de 122 bilhões de nós, e levam cerca de dois dias. Uma busca completa no nível 18 deverá levar menos de quatro semanas. O fator de ramificação desta heurística, que é a razão do número de nós gerados em uma iteração, comparado com o número de nós gerados na iteração anterior é, aproximadamente, o fator de ramificação da busca de força bruta de 13,34847.

3.6) Análise de Desempenho

O número consistente de nós gerados pelo IDA* em uma dada iteração para diferentes instâncias do problema sugere que o desempenho do algoritmo é passível de ser analisado. A maioria das análises de buscas com heurísticas foi feita apenas com modelos analíticos, e não predisseram o desempenho de problemas reais. A discussão a seguir não é uma análise formal, mas um conjunto de observações de regularidades nos dados obtidos.

A maioria das análises de avaliações heurísticas relacionam o desempenho de um algoritmo de busca à acurácia da heurística como uma estimativa da distância exata ao objetivo. Uma dificuldade desta abordagem é a dificuldade de medir a acurácia de uma heurística, dado que determinar a distância exata para o objetivo em um dado problema é computacionalmente difícil para grandes problemas. Por acurácia, entende-se como a habilidade de selecionar estados que levarão a mais rapidamente ao estado objetivo. A heurística mais acurada é aquela que sempre seleciona o nó que pertence à solução mais próxima do estado objetivo.

Uma primeira observação é que se pode caracterizar a eficácia de uma heurística admissível pelo seu valor esperado sobre o espaço do problema. Isto pode ser determinado para um grau de acurácia exemplificando-se randomicamente o problema, e computando-se a heurística para cada estado. Para o cubo mágico, os valores da heurística foram enumerados em tabelas e seu valor esperado pode ser computado exatamente como a média dos valores presentes na tabela. Além disso, o valor esperado do máximo entre duas heurísticas pode ser computado a partir destas tabelas, assumindo que seus valores são independentes, o que é perfeitamente razoável aqui.

Se o valor da heurística de cada estado é igual ao seu valor esperado *e*, então a busca IDA* no nível *d* seria equivalente a busca em profundidade iterativa para a profundidade *d* - *e*, dado que f = g + h de cada estado seria sua profundidade menos *e*. Nestes experimentos, esta heurística tem um valor esperado de 8,878, que pode ser arredondado para 9. Uma busca IDA* completa no nível 17 gera aproximadamente 122 bilhões de nós. Todavia, uma busca força bruta para a profundidade 17 - 9 = 8 gera apenas cerca de 1,37 bilhões de nós, aproximadamente duas ordens de grandeza a menos.

A razão para esta discrepância é que os estados encontrados em uma busca IDA* não são um exemplo aleatório do espaço do problema. Estados com valores altos da função heurística são podados, e estados com valores baixos da função heurística geram mais filhos na mesma iteração. Assim, a busca é desviada em favor dos valores baixos da heurística, o que baixa a média dos valores da heurística dos estados encontrados pelo IDA* provocando uma maior geração de nós do que o previsto pelo valor esperado da heurística.

O próximo passo é predizer o valor esperado da heurística a partir da quantidade de memória usada e do fator de ramificação *b* do problema. Em geral, o limite mínimo do valor esperado de uma heurística é o logaritmo na base b do número de estado armazenados na tabela, dado que com *d* movimentos, pode-se gerar b^d estados. Porém, poderão haver estados duplicados na árvore de busca.

Como outro exemplo, dado que há cerca de 88 milhões de estados distintos de cubinhos dos cantos, isso sugere que o número de movimentos necessários para resolver o problema deveria ser cerca de sete. De fato, o número médio de movimentos é 8,764. Outra vez, a razão para a discrepância é que nem todos os nós do nível sete na árvore correspondem a estados únicos de cubos dos cantos, requerendo, então, uma exploração mais profunda para gerá-los em sua totalidade.

Em outras palavras, estimar o valor esperado da heurística a partir do fator de ramificação do espaço e do número de estado fornece um valor muito baixo ou pessimista. Por outro lado, estimar o número de nós gerados pelo IDA* a partir do valor esperado da heurística fornece um valor que também é muito baixo, que é otimista. Isto sugere que combinar os dois, para estimar o número de nós gerados pelo IDA* a partir do fator de ramificação do espaço e pelo número de estados na tabela heurística, os dois erros poderão ser mutuamente cancelados.

Formalmente:

- Seja *n* o número de estados no espaço do problema,
- Seja *b* o fator de ramificação do espaço,
- Seja *d* o comprimento médio da solução ótima para uma instância aleatória do problema,
- Seja *m* o total de memória utilizada, em termos de números de valores de heurística armazenados, *e*
- Seja *t* o tempo de execução do IDA*, em termos de nós gerados.

O comprimento médio *d* da solução ótima, que é a profundidade à qual o IDA* deverá buscar, pode ser estimada por $\log_b(n)$. Como argumentado anteriormente $e \sim \log_b(m)$ e $t \sim b^{(d-e)}$. Assim:

$$t \sim b^{(d-e)} \sim b^{(\log_b(n) - \log_b(m))} = n / m$$

Então, o tempo de execução do IDA* pode ser aproximado por O(n/m), o tamanho do problema dividido pela memória disponível. Usando os dados deste experimento:

- $n = 4{,}3252 * 10^{19}$
- $m = 173.335.680$
- $n / m = 249.527.409.904$
- $t = 352.656.042.894$

3.7) Usando Database Disjuntos e Pareados

Infelizmente as técnicas de *database disjuntos* e *database pareados* não podem ser usadas para solucionar o cubo mágico, pois cada movimentação envolve mais de uma peça e o seu uso poderia ocasionar perda da admissibilidade da nova função heurística.

3.8) Conclusões

Nesse trabalho, foram descobertas as primeiras soluções ótimas para instâncias randômicas do Cubo Mágico, um dos mais famosos problemas combinatórios deste tempo. O comprimento médio da solução ótima aparenta ser de 18 movimentos.

A idéia chave, devida o a Culberson e Schaeffer em 1996, é tomar um subconjunto de objetivos o problema original, e pré-computar e armazenar o número exato de movimentos para resolver estes sub-objetivos para todos os possíveis estados iniciais. Assim, a solução exata para estes sub-objetivos é usado como um limite mínimo para uma busca IDA* para o problema original.

Caracteriza-se aqui a eficácia de uma heurística admissível simplesmente pelo seu valor esperado. É também apresentada uma análise informal da técnica e elaborada uma hipótese de que seu desempenho é regido por aproximadamente t ~ n/m, onde t é o tempo de execução, m é a quantidade de memória utilizada e n o tamanho do espaço do problema. Esta aproximação é consistente com os resultados destes experimentos, e sugere que a velocidade do algoritmo cresce linearmente com a quantidade de memória disponível.

4) Complexidade de Tempo da busca IDA* para o Cubo Mágico

[Korf 2001]

Em trabalhos realizados anteriormente, era usado um modelo abstrato de árvore de espaço de busca, onde cada nó teria exatamente *b* filhos, e cada aresta teria um custo igual a uma unidade e que haveria um único estado objetivo à uma profundidade *d*. Esse modelo prevê que heurísticas com erro:

- absoluto constante resultam em uma complexidade de tempo linear;
- relativo constante resultam em uma complexidade de tempo exponencial;

Problemas com essa abordagem:

- assume que há um único caminho a partir do estado inicial até o estado objetivo;
- determina a acurácia da função heurística baseada num único estado;
- os resultados em geral são assintóticos

Devido a essas limitações é bastante difícil prever a performace do A* e IDA* em qualquer problema concreto.

4.1) Conceitos

Função Heurística Consistente

Seja h(*n*) a estimativa do custo até o estado objetivo a partir do nó *n.* Seja g(*n*) o custo do nó *n.*Seja *n'* um filho de *n.* Seja f(*n*) = g(*n*) + h(*n*). Uma função heurística h(*n*) é dita consistente se para todo n', f(*n*) ≤ f(*n'*). Supõe-se que para qualquer discussão a frente a função heurística seja consistente.

Função de Distribuição dos Valores Heurísticos: D(n)

D(*n*) é fração total dos estados to problema com valores heurísticos iguais ou inferiores a *n*.

Função de Equilíbrio dos Valores Heurísticos: P(n)

P(*n*) é probabilidade de um nó ser escolhido randomicamente e uniformemente entre todos os nós de uma profundidade *n*. P(*n*) **não** é uma *propriedade do problema*, mas sim do *espaço de busca do problema*. Para o cubo mágico P(n) = D(n), mas para a família do jogo dos 8 (jogo dos 15, jogo dos 24, ...) a probabilidade posição do espaço em branco deve ser considerada.

4.2) Resultado Principal

- ***b*** é o branching factor,
- ***d*** é a profundidade da solução ótima,
- ***P(n)*** é função de distribuição dos valores heurísticos.
- O número de nós expandidos na última iteração do IDA* é em média:

$$b^0P(d)+b^1P(d-1)+...b^dP(0)= \sum_{i=0}^{d} b^i\, P(d-i)$$

4.3) Teoria e Prática

Para o cubo mágico, discrepância entre predição do número de nós expandidos com o número de nós realmente expandidos foi 1% em média.

Para o jogo dos 15, discrepância entre predição do número de nós expandidos com o número de nós realmente expandidos foi 2,5 % em média.

5) Trabalho Prático

“Especificar e implementar o jogo do cubo mágico. Primeiro para cada face sendo uma matriz 2x2, depois 3x3 e por fim NxN. Criar uma heurística e comparar com uma busca completamente desinformada.”

Linguagem utilizada: Object Pascal / Delphi.

Modelagem

O cubo foi modelado como uma classe, com um conjunto de métodos para efetuar as rotações (operadores) e outras atividades custodiais.

A representação 3D não foi utilizada, pois seriam necessários 3 números inteiros para representar a posição de cada cubículo que compõe o cubo. Cada face do cubo foi representada como uma matriz de bytes NxN. Cada valor dos elementos de uma
matriz (face) é mapeado numa cor específica e vice-versa. Cada face é indexada por um número de 0 a 5.

Como são seis faces, a cardinalidade do conjunto de valores que cada posição na matriz pode assumir é 6. Sem perda de generalidade esses valores vão de 0 a 5. O número de elementos de um lado com uma determinada cor é sempre NxN, e o total de elementos no cubo é 6xNxN. Essas afirmações são válidas para qualquer cubo NxNxN.

Na modelagem adotamos os seguintes critérios;

- Sejam A, B o identificador de duas faces.

  A+B = 5 se, e somente se, as faces são opostas;

  Se uma posição da matriz contém um valor A a face onde esse valor foi iniciado é A;

Cada eixo de rotação perpendicular a uma face indexada por A dado por A mod 3;
Cada rotação numa face de índice A ocorre em um plano definido pelo seu eixo de rotação. Esse plano é indexado por valores que vão de 0 a N-1;

Cada rotação de um plano pode ter dois sentidos: 90ºe -90º. O sentido de rotação é definido pela regra da mão direita. Onde vetor de rotação tem o sentido da face de menor índice para a de maior. Ele é perpendicular ao eixo de rotação.

Um movimento então foi modelado como:

Cubo.Rotação(eixo,plano,sentido);

As rotações do tipo Rotação(A,0,sentido) e Rotação(A, N-1,sentido) modificam as faces que definem o vetor de rotação de forma que as matrizes que as representam sofram uma rotação interna de seus valores de mesma intensidade, mas de sentidos opostos. Tais operações preservam as propriedades das rotações mecânicas em um cubo real. Isso deve se à dependência de valores dos elementos das matrizes que se encontram em arestas de faces adjacentes num cubo real.

A partir da operação Rotação(eixo,plano,sentido) as demais funções relevantes ao problema são derivadas.

A rotação de cubo num eixo é realizada da seguinte forma

Cubo.RotaçãoEixo(eixo,sentido)

Para i assumindo os valores de 0 a Cubo.N-1 faça
Cubo.Rotação(eixo,i,sentido);

Isomorfismo de Cubos

Uma determinada configuração de cubo NxNxN possui 24xN configurações isomorfas.
Por isomorfismos definimos:

- Seja C e D cubos mágicos. Existe uma seqüência finita de aplicações de rotações de eixo em C que o torne idêntico a D, se e somente se, C e D são configurações isomorfas.

- Dois cubos possuem configurações isomórficas, se e somente se, são isomorfos.

- Todo cubo é isomorfo se comparado consigo mesmo.

- Dois cubo são isomórficos triviais, se C = D.

Rotações Isomórficas

As rotações isomórficas são aquelas que aplicadas a um determinado cubo em qualquer seqüência poderão gerar cubo isomorfos, se e somente se, o cubo isomorfo gerado é isomorfo trivial.

Implementação de uma Rotação Isomórfica:

Cubo.RotaçãoIsomórfica(eixo,plano,sentido);

```
Se plano<>0 então
        Cubo.Rotação (eixo,plano,sentido)
Senão
        Para i assumindo os valores de 1 a Cubo.N-1 faça
                Cubo.Rotação(eixo, i, -sentido);
```

Para a implementação acima um efeito geométrico curioso é observado:

- Seja C um cubo mágico, se a C for somente aplicada rotações isomórficas com a implementação acima então existe um cubículo localizado num de seus vértices cuja posição é inalterada.

Fatos das rotações isomórficas:

Existe uma seqüência de aplicações de rotações de tamanho M que resolve um determinado cubo se, e somente se, existe uma seqüência de aplicações de rotações isomorfas de tamanho M que o resolva.

Devido ao fato acima:

- não mais será trado o caso rotações simples, somente isomorfas. De agora em diante, rotação é sinônimo de rotação isomorfa.

Agora vale lembrar que rotações no eixo de cubos ficam proibidas! Pois destruiriam o isomorfismo!

Conseqüências do uso de rotações isomorfas

1. **Faz sentido dizer que cubículos estão fora de sua face;**
2. **Faz sentido o uso da distância Manhattam para cubículos fora de sua face;**
3. **Diminuição do overhead computacional, devido a comparações desnecessárias, pois a primeira falha na comparação entre dois cubos revela, definitivamente, que eles não são isomorfos.**
4. **Elimina a geração de cubos isomorfos que não sejam isomorfos triviais.**

Buscas

As buscas implementadas foram:

- Não informadas
  - Largura
  - Profundidade Limitada
  - Profundidade Interativa

- Informadas:
  - A*
    - Heurística Peças fora do lugar
    - Heurística Manhathan
    - Heurística Desordem do Cubo (experimental)
  - IDA*

    - Heurística Peças fora do lugar
    - Heurística Manhathan
    - Heurística Desordem do Cubo (experimental)

A manutenção de ponteiros para os nós-pais foi implementada.

Cada cubo possui variáveis internas para guardar informações sobre seu estado na busca:

- Quem é seu pai, caso não fosse o cubo inicial;
- Profundidade.
- Visitado ou não

Há ainda variáveis de controle e para evitar computação.

Eliminação de estados redundantes

Foi implementada uma tabela hash para realizar essa tarefa: excluir cubos isomorfos da busca. Porém ela mostrou nenhum ganho e gerou um consumo muito grande de memória, o que provocou um trashing prematuro do sistema.

Uma idéia menos radical foi a política de **avoidance**: procurar evitar, mas não garantir a existência de cubos isomorfos.

Política de avoidance baseia-se na existência de ponteiros para o nó pai: toda vez que um nó fosse expandido, todos os ancestrais diretos do nó seriam comparados com ele. Caso algum deles fosse isomorfos do descendente, este seria descartado.

Assinaturas

A comparação dos nós na tentativa de evitar estados repetidos poderia onerar desempenho da busca, o que aumentaria o custo off-line. Para evitar a comparação direta criou-se uma função, semelhante àquelas usadas em hashing, que gera uma assinatura para cada cubo, que tem como argumentos a disposição dos cubículos do cubo. A assinatura usada é um inteiro de 32-bits, armazenado na própria instância objeto cubo. Se dois cubos tivessem assinaturas diferentes, eles não seriam isomorfos, caso contrário, é efetuada, de fato, a comparação, cubículo a cubículo, para saber se são, ou não, isomorfos. Essa assinatura é sempre calculada sob demanda. Sabendo-se que o fator de ramificação de um cubo 3x3x3 é 18 e que seu fator de ramificação é 18 a maior árvore completa possui uma altura de 40 nós*. Com isso podemos inferir que o fator de ramificação caia para 17. Para soluções muito profundas não é de grande ajuda, mas para soluções mais próximas a raiz fornece uma melhoria. No caso do cubo 2x2x2 o fator de ramificação cai para 11.

***É claro que não se pode garantir tal fato, é apenas uma suposição até então que tal árvore exista.**

Buscas Não Informadas

Para as buscas não informadas foram implementadas filas e pilhas, para a realização das buscas em largura e profundidade limitada, respectivamente. A busca em profundidade interativa foi um incremento da profundidade limitada na qual essa era chamada na rotina daquela, mas uma diferença: a comparação com a solução só acontece na profundidade limite.

Buscas Informadas

Para a realização das buscas informadas foi implementada uma heap*, lista de prioridades. O custo da inserção e remoção de um nó nessa estrutura de dados é **O**(log(n)), onde n é o tamanho essa heap (a quantidade de nós nela contida). O nó cabeça seria sempre o nó com menor f*.

***Função heurística para Busca A*: f =h+g;**

*Um fato interessante é que as pilhas e as filas das buscas não informadas poderiam ter sido substituídas por uma heap implementando-se uma função f apropriada para cada caso.

Heurísticas Clássicas

- Peças fora do lugar:

  H = (Número de Peças fora da sua face) /(4 x N), para um cubo NxNxN

- Manhattam:

H = ((Número de peças fora da face e em face adjacente)+
2*(Número de peças fora da face e em face oposta))//(4 x N), para um cubo NxNxN

As duas heurísticas acima são admissíveis, pois afirmam que é possível chegar a solução realizando-se um número de rotações exatas, H, pois cada rotação moveria exatamente 4*N cubículos, sem levar em conta a eventual perturbação de tal operação nas demais peças. Se fosse possível efetuar a tarefa em com um número de passos menor que H, então existe alguma rotação que movimente um número de cubículos entre faces maior que 4*N, o que é um absurdo!

Essas heurísticas só puderam ser implementadas devido ao uso de rotações isomorfas. Uma atribuição dinâmica, por exemplo, da cor de um lado ao maior número de pedras de mesma cor existente nele, poderia fazer a busca oscilar e a solução nunca ser achada, pois a configuração do cubo é muito sensível às rotações.

Heurísticas Experimental: Desordem do Cubo

As heurísticas utilizadas para a resolução do Cubo Mágico como, por exemplo, o "Manhattan Distance", não podem ser utilizadas diretamente, dado que extrapolam (e muito) o número real de passos que faltam para se chegar à configuração final. Para usá-las, faz uma correção através de uma equação linear da forma:

$$h' = ah + b$$

Onde h é o valor inicial da heurística e h' é o valor corrigido. Tipicamente, temos os seguintes casos:

- Para h = 0 (cubo resolvido), h' também terá que ser zero.
- Quando falta um único movimento para se resolver o cubo, via de regra o valor de h é de 4N, sendo N o tamanho do cubo NxNxN.

Assim, a equação fica com a = 1/ 4N e b = 0.

Deste modo, a heurística seguramente será admissível, visto que nenhum movimento mexerá com mais de 4N faces.

Todavia, uma heurística é melhor quanto maior for o seu valor esperado [Korf 97]. Isto, na prática, significa que quanto mais próxima do valor real, melhor será a heurística. E esta forma de tornar a heurística admissível acaba por resultar em valores distantes do real.

Com isto, foi proposta uma forma de correção no formato:

$$h' = ah^2 + bh + c$$

Porém, para isto seria necessário o limite superior da heurística original, e associá-lo ao número máximo de passos para a solução ótima. Então, foi feita uma heurística que mede a desordem do cubo, contando, para cada face, em cada linha ou coluna, qual é a cor que mais ocorre, e quantas peças faltam para que toda a linha ou coluna fique com esta cor. Isto também é feito para a face inteira.

O algoritmo fica assim:

```
função desordem(C: cubo, N: inteiro): inteiro
início
        valor <- 0;

        para cada face em C fazer
        início
                para linha de 1 a N fazer
                início
                        conta <- Número de ocorrências da cor mais presente na linha;
                        valor <- valor + (N - conta);
                fim

                para coluna de 1 a N fazer
                início
                        conta <- Número de ocorrências da cor mais presente na coluna;
                        valor <- valor + (N - conta);
                fim
                conta <- Número de ocorrências da cor mais presente na face;
                valor <- valor + (N * N – conta);
        fim

        retornar valor;
fim
```

Os dois primeiros casos são similares ao que foi descrito no início:

- Quando o cubo está resolvido, em cada linha ou coluna de cada face, todas as peças têm a mesma cor. Assim, h = 0.

- Quando falta um único movimento para se resolver o cubo, todas as peças serão da mesma cor em duas faces. Nas quatro faces remanescentes, haverá uma peça de cor diferente em cada linha, e nas colunas todas terão a mesma cor (ou vice-versa). Da mesma forma, nestas faces, teremos N peças de cor diferente em cada face. Assim:

    - h = h1 + h2 = 4N + 4N
    - h = 8N

- O valor máximo da heurística ocorrerá quando, em todas as faces, as cores das peças forem diferentes em todas as linhas e colunas (para N > 6, isto não será mais

possível). Neste caso, teremos uma contribuição de N - 1 em cada linha ou coluna. Como são seis faces, e há N linhas e N colunas em cada face, temos:

- $h1 = 6 * 2 * N * (N - 1)$
- $h1 = 12 * (N^2 - N)$

- Vendo-se cada face como um todo, a maior desordem ocorrerá quando a contagem das cores for aproximadamente a mesma. Assim:

  - $h2 = 6 * (1 - 1/6) * N^2$
  - $h2 = (6 - 1) * N^2$
  - $h2 = 5N^2$

- Juntando-se os dois resultados:

  - $h = h1 + h2 = (12N^2 – 12N) + 5N^2$
  - $h = 17N^2 - 12N$

Assumiu-se como número máximo de movimentos 7N - 3, dado que para N = 2 são necessários no máximo 11 movimentos, e para N = 3 é preciso 18 movimentos. Não há valores para N >= 4.

Com isto, temos o seguinte mapeamento de valores de h e h':

H	***h'***
0	0
8N	1
$17N^2 - 12N$	7N - 3

O que resulta no seguinte sistema de equações:

$0^2 * a + 0 * b + c = 0$
$(8N)^2 * a + (8N) * b + c = 1$
$(17N^2 - 12N)^2 * a + (17N^2 - 12N) * b + c = 7N - 3$

Trivialmente, obtemos c = 0. Desenvolvendo-se as demais equações, obtemos:

$64N^2a + 8Nb = 1$
$(289N^4 - 408N^3 + 144N^2)a + (17N^2 - 12N)b = 7N - 3$

Resolvendo-se este conjunto de equações, obtivemos:

$a = (39 * N - 12) / (2312 * N^4 - 4352 * N^3 + 1920 * N^2)$
$b = (289 * N^2 - 856 * N + 336) / (2312 * N^3 - 4352 * N^2 + 1920 * N)$

Assim, podemos aplicar a correção necessária para obtermos um valor de h' mais próximo do real. A seguir, valores possíveis para a e b, dependendo de N.

N	***a***	***b***
2	0.00669643	-0.04464286
3	0.00120623	0.01271712
4	0.00041853	0.01785714
5	0.00019283	0.01728662

Resultados:

Foram feitos testes desta heurística, comparando-a com o Manhattan Distance. A seguir, os resultados:

Cubo 2x2x2:

Profundidade Esperada	A*				IDA*			
	Desordem		Manhattan		Desordem		Manhattan	
	Nós Gerados	Profundidade	Nós Gerados	Profundidade	Nós Gerados	Profundidade	Nós Gerados	Profundidade
6	1.572	6	6.252	6	25.005	6	206.853	6
6	1.080	6	8.388	6	6.666	6	37.050	6
7	16.056	9	19.476	7	956.337	9	810.018	7
7	9.504	7	26.376	7	552.269	7	848.357	7
8	9.516	8	129.648	8	1.268.306	8	15.467.342	8

Cubo 3x3x3:

Profundidade Esperada	A*				IDA*			
	Desordem		Manhattan		Desordem		Manhattan	
	Nós Gerados	Profundidade	Nós Gerados	Profundidade	Nós Gerados	Profundidade	Nós Gerados	Profundidade
4	756	4	1.710	4	182	5	1.478	4
5	7.650	5	52.812	5	39.518	8	102.914	5
5	3.366	5	2.070	5	1.865	7	6.707	5
5	13.860	7	8.190	5	71.572	7	5.470	5
6	5.778	6	286.272	6	1.825	8	325.357	6

Após estes testes, pode-se verificar que a heurística não é admissível, dado que em alguns casos retorna soluções sub-ótimas. Todavia, esta heurística efetua uma redução considerável no número de nós gerados, o que reduz o tempo total de processamento (e

também o risco de exaustão da memória, o que acarreta em paginação excessiva, deteriorando ainda mais o desempenho do programa).

Conclusões

Após alguns testes observamos:

A respeito das buscas informadas com heurísticas admissíveis podemos dizer que elas funcionaram para cubos 2x2x2 e 3x3x3 até a profundidade 5 e 4 respectivamente. Para profundidades maiores que 4 e cubos com fator N>4 elas degeneram-se em buscas em largura. Essas degenerações devem-se à natureza do problema e à qualidade das funções heurísticas empregadas: soluções muito profundas não nos informam, aparentemente, da sua natureza somente consideram as posições relativas entre cubículos e faces. É verdade que a recíproca não é verdadeira.

As técnicas para evitar estados repetidos foram bem sucedidas para cubos 2x2x2, mas para cubos maiores só retardou o inevitável: a falta de memória perante à soluções profundas (no caso do A*) e a baixa acurácia das funções heurísticas utilizadas.

Considerações finais: mudanças de técnicas pode fazer-se necessárias: a busca pela solução ótima pode não valer a pena, devido ao seu alto custo off-line.

Uso do Programa: MagicCube.exe

Autores: Paulo Coelho e Ricardo Fernandes Ribeiro

Dificuldade: indica o grau de “embaralhamento” de um cubo numa seqüência aleatória de até 100 rotações

Embaralha: realiza o embaralhamento com o grau indicado em Dificuldades

Busca !: efetua a busca selecionada à priori.

Profundidade Máxima: indica em que profundidade uma não-informada deve parar.

Ação: Usado após a realização de uma busca, ele efetua as rotações do cubo mais a esquerda, o qual é uma cópia do cubo principal.

NxNxN: informa a geometria do cubo.

Após a execução de uma busca, o programa fornece uma pequena “estatística”:

- **Nós Gerados**: Número de nós que foram fruto de uma expansão, através a aplicação de rotações;
- **Nós Expandidos**: Número de nós que sofreram expansão, ou seja que foram analisados;
- **Nós não Expandidos**: Nós que foram gerados, mas não foram expandidos;
- **Total de Nós** : Total de nós durante a execução da busca;
- **Profundidade Máxima**: é a profundidade do maior nó gerado ou expandido.

6) Referências

[Korf 1997] **Finding Optimal Solutions to Rubik's Cube Using Pattern Databases**, Richard E. Korf.
[Korf 2000] **Recent Progress in the Design and Analysis of Admissible Heuristic Functions**, Richard E. Korf
[Korf 2001] **Time Complexity of Iterative-Deepening-A***, Richard E. Korf, Michael Reid, Stefan Edelkamp.
[KorfFel2000] **Disjoint Pattern Database Heuristics**, Richard E. Korf, Ariel Felner

Outras Referências:

Planning with Pattern Databases, Stefan Edelkamp
Searching for Macro Operators with Automatically Generated Heuristics, István T. Hernádvölgyi.
Planning as Heuristic Search, Blai Bonet and Héctor Geffner