

Avaliação das Buscas Informadas e Não Informadas Aplicadas ao Cubo de Rubik

Thiago Luiz Cabreira do Nascimento¹

¹Engenharia de Computação – Centro de Ciências Exatas e Tecnológicas
Universidade Católica Dom Bosco (UCDB)
CEP 79117-900 – Campo Grande – MS – Brasil

Abstract. *This paper describes how the main search techniques in the state space behave when applied in Rubik's Cube resolution. For some instances of the problem, each implemented search was applied and for each one, was observed parameters like how deep the goal state was found, how many nodes was expanded, how many nodes was evaluated, how many nodes there was in memory when the goal state was found and the time dispended until the algorithm had finished.*

Resumo. *Este artigo descreve como as principais técnicas de busca no espaço de estado se comportam quando aplicadas na resolução do Cubo de Rubik. Para várias instâncias do problema, cada busca implementada foi aplicada e para cada uma, foram observados parâmetros como em qual profundidade o estado meta foi encontrado, quantos nós foram expandidos, quantos sofreram expansão ou quantos nós foram avaliados, quantos nós existiam na memória quando encontrado o estado meta e o tempo gasto até o algoritmo ser finalizado.*

1. Introdução

O Cubo de Rubik, também conhecido como *Cubo Mágico* foi inventado pelo húngaro Erno Rubik na década de 70. Originalmente ele é um cubo 3x3x3, com diferentes cores em cada quadrado expostos dos subcubos. Cada plano 3x3x1 do cubo pode ser rotacionado 90, 180, 270 graus em relação ao resto do cubo. No estado meta, todos os quadrados em cada lado do cubo são da mesma cor. O quebra cabeça (puzzle) consiste em embaralhar o cubo com uma quantidade de movimentos aleatórios e o objetivo é restaurar o cubo ao estado inicial, ou seja, o estado meta.

A dificuldade do problema é que, em qualquer configuração que o cubo se encontre, 43,252,003,274,489,856,000 estados diferentes podem ser encontrados[Korf 1997], desencorajando, ou mesmo impossibilitando, o uso de algoritmos que empregam somente a força bruta, como as buscas cegas por exemplo.

Para resolver o problema do cubo, alguma estratégia geral é necessária, consistindo geralmente de um conjunto de sequência de movimento ou operadores que corretamente posicione subcubos individuais sem violar outros previamente posicionados. Estratégias típicas requerem em média entre 50 e 100 movimentos para resolver uma instância aleatoriamente gerada do cubo.

O presente trabalho demonstra como as diferentes técnicas de busca com diferentes heurísticas, no caso de busca informada, se comportam quando empregadas na resolução do problema do cubo mágico. Quais são as vantagens e desvantagens de cada abordagem do ponto de vista quantitativo e qualitativo.

2. Trabalhos Correlatos

Existem muitos trabalhos para resolver o problema do cubo mágico, embora a maioria esteja fora do mundo científico. A primeira aparição do *Cubo de Rubik* na literatura de Inteligência Artificial foi em 1985. Os experimentos podem ser divididos em dois grandes grupos: Um que apenas pretende resolver o problema, e outro que quer resolver o problema com uma solução ótima, ou seja, resolver com o menor número de passos possível.

Trabalhos que não se preocupam em resolver o problema com solução ótima geralmente usam a estratégia de decompor o problema em outros problemas menores, e usam heurística apropriadas para cada fase da busca pelo objeto. Alguns chegam perto de uma solução ótima, com a grande vantagem de requerer um tempo bem menor do que àqueles que empregam algoritmo ótimo.

Randal em seu artigo [Randall 1998] afirma resolver instâncias do cubo com uma média de 1 hora usando um computador com 4 processadores de 466Mhz e com 1.2 GB de memória. Para 200 instâncias, todas as soluções foram encontradas em profundidades inferiores a 22. O algoritmo utilizado, como o próprio autor descreve, é praticamente força bruta com grandes podas baseadas em heurísticas.

O primeiro trabalho que resolveu com solução ótima o problema do cubo mágico, foi produzido por Richard E. Korf em 1997, o procedimento empregado está descrito em seu artigo intitulado “Finding Optimal Solutions to Rubik’s Cube Using Pattern Databases” [Korf 1997], no experimento descrito resolveu-se 10 instâncias do cubo, sendo que 1 foi resolvida em 16 movimentos, 3 em 17 movimentos e as outras 3 em 18 movimentos. Executado em uma máquina capaz de avaliar 700,000 nós por segundos, encontrar um estado meta na profundidade 17 leva cerca de 2 dias. O algoritmo divide o problema em problemas menores que podem ser resolvidos com solução ótima, e, para eficiência, emprega também *pattern databases*, que consiste basicamente em pré-computar funções heurísticas de estados e armazenar na memória para consultas [Culberson and Schaeffer 1996].

Existem ainda trabalhos que empregam técnicas diferentes para abordar o problema sem sair da Inteligência Artificial, um exemplo é o *Aprendizado por Experiência* [Briesemeister et al. 1995].

3. Busca no Espaço de Estado

Muitos problemas podem ser vistos como encontrar um objetivo saindo de um ponto inicial e trilhar um caminho até um ponto final fazendo um quantidade finita de movimentos diferentes. Problemas assim podem ser resolvidos empregando-se a busca no estado de espaço [J. and Norvig 1995]. Problemas clássicos que podem ser resolvidos com essas técnicas são entre outros: missionários e canibais, problema das 8 peças, 8-rainhas, problema do vasilhame, e o abordado aqui, o cubo mágico.

Um **estado** é uma configuração do problema em um momento discreto de tempo. No caso do cubo um estado é o estado meta, onde cada face do cubo tem apenas uma cor. Outro estado é a configuração do cubo após ser rotacionado de alguma forma.

O **espaço de estado** é o conjunto de todos os estados possíveis. Para o cubo de Rubik, significa todas as permutações possíveis, ou seja 43,252,003,274,489,856,000

estados diferentes.

Um **operador** é uma operação que pode modificar um estado. Idealmente, um operador sempre deve causar movimento no estado em que foi aplicado. Para o problema do Cubo de Rubik, um operador pode ser: “rotacionar uma face no sentido horário”.

Busca no espaço de estado significa achar um caminho a partir do estado inicial até um estado objetivo. O espaço de estado é mais facilmente visualizado como estruturado em forma de árvore, onde cada nó representa um estado, e o fator de ramificação é dado pelo número de operadores que pode ser empregado.

Expandir um nó, significa aplicar todos os operadores possíveis a um estado (nó) para gerar novos estados, os nós sucessores.

Os tipos de busca podem ser divididos em duas, busca informada e não informada. A diferença é que na busca não informada, nenhum conhecimento específico do problema é usado para encontrar um caminho no espaço de estado, a única coisa que se sabe é se está em um estado meta ou não. Na busca informada existe uma função de avaliação, que usa o conhecimento do problema para avaliar qual é o melhor caminho a ser seguido. Essa função é chamada de função heurística.

Uma busca é completa quando sempre que houver um resultado possível ele for encontrado, a esse **critério** dá-se o nome **completude**. Outros critérios de avaliação das buscas são **complexidade de tempo**, que diz respeito ao tempo gasto para a busca ser finalizada, **complexidade de Espaço**, que mede a quantidade de memória utilizada e **otimalidade**, que verifica se o objetivo foi encontrado com um número mínimo de passos.

3.1. Busca Não Informada (Cega)

As *Buscas Não Informadas* são assim chamadas por não utilizar conhecimento do problema quando procurando o estado meta. Tudo o que sabe em determinado momento é se o estado é meta ou não. São também conhecidas como *Buscas Cegas*.

Na prática, não é viável para problemas com espaço de estado muito grandes, pois demandam muita memória e processamento. Nenhum trabalho que resolva qualquer instância do Cubo de Rubik foi desenvolvido até hoje usando apenas busca cega.

3.1.1. Busca em Largura

Para encontrar o estado meta, o algoritmo avalia todos os nós de menor profundidade primeiro, e por isso sempre encontra soluções ótimas, ou seja, soluções que partem do estado inicial e utilizam o menor número de operadores possíveis para chegar no estado meta. O problema com essa busca é o consumo de memória, pois mantém os nós gerados na memória.

3.1.2. Busca em Profundidade

A *Busca em Profundidade*, ao contrário da *Busca em Largura*(seção 3.1.1), avalia o nó de maior profundidade primeiro. A vantagem sobre a busca em largura é que apenas um ramo da árvore precisa ser mantida na memória. Em contrapartida, pode retornar soluções

que não são ótimas e, pior, entrar em loop infinito e nunca retornar uma solução. Para solucionar o problema do loop infinito, pode-se estipular um limite para a profundidade da busca (Busca em Profundidade Limitada), que é a que foi utilizada nesse trabalho.

3.1.3. Busca com Aprofundamento Iterativo

A *Busca com Aprofundamento Iterativo* é um misto da *Busca em Largura*(seção 3.1.1) e da *Busca em Profundidade*(seção 3.1.2), a avaliação dos nós é feita como na busca em profundidade, só que limitando a profundidade iterativamente, ou seja primeiro faz a busca em profundidade até a profundidade 1, depois inicia novamente e vai até a profundidade 2, depois até a profundidade 3 e assim sucessivamente. Embora o custo com processamento seja maior que as outras duas buscas, ela tira vantagem da melhor característica de cada uma: da busca em profundidade herda a necessidade de apenas guardar o caminho da raiz até o nó atual, e da busca em largura herda o fato de produzir sempre respostas ótimas.

3.1.4. Busca de Custo Uniforme

A *Busca de Custo Uniforme* procura minimizar o custo da solução a longo tempo calculando o custo do caminho atual. Esse cálculo retorna um valor exato e não é considerado “conhecimento do problema”.

O custo do caminho é geralmente denotado pela letra g , e ela sempre encontrará a solução ótima desde que se respeite a regra: $g(\text{sucessor}(n)) > g(n)$.

3.2. Busca Informada

As *Buscas Informadas* são aquelas que empregam conhecimento específico do problema na busca pelo estado meta. A eficiência de uma busca está intimamente ligada a *heurística*¹ utilizada, que é o valor que estima a distância do estado atual em relação ao meta. É geralmente denotada pela letra h : $h(n)$ = custo estimado da solução a partir do estado n .

Uma função de custo é a que calcula o custo real do estado atual até a raiz, é geralmente denotado pela letra g .

A *função de avaliação* é a função que calcula qual o melhor caminho a ser seguido, um algoritmo de busca informada geralmente estima o custo da solução e tenta minimizá-lo.

3.2.1. Melhor Primeiro (Best First)

Sempre escolhe o estado com menor custo *aparente*, isto é, utiliza a função heurística, que estima o custo da solução e não o seu valor exato. É conhecido também conhecida

¹A palavra heurística é derivada do verbo grego *heuriskein*, que significa encontrar, descobrir. É dito que Arquimédes correu nú pelas ruas gritando “Heureka” após ter descoberto o princípio da flutuação[J. and Norvig 1995].

como *Busca Gulosa* ou *Greedy Search* ².

A busca costuma encontrar soluções rapidamente, mas não é ótima, pois escolhe o melhor aparente e não o real melhor. Também não é completa, uma vez que pode entrar em loop infinito.

3.2.2. A Estrela (A Star ou A*)

Emprega as melhores características da *Busca Gulosa*(seção 3.2.1) e da *Busca com Custo Uniforme*(seção 3.1.4). Usa uma função heurística herdada da *Busca Gulosa* e a função de custo herdada da *Busca de Custo Uniforme* para escolher qual nó será expandido. Com essas duas funções cria-se uma nova função, chamada *função de avaliação* que nada mais é do que a soma das duas outras funções : $f(n) = g(n) + h(n)$.

A busca é ótima desde que a função heurística empregada seja *admissível*, ou seja, ela nunca pode superestimar o valor real da solução.

3.2.3. Subida de Encosta (Hill Climbing)

A busca *Subida de Encosta*³ faz parte da classe de algoritmos de *Melhoramento Iterativo* que, a grosso modo, sempre procura melhorar seu estado a partir do estado atual.

Não leva em consideração o caminho já percorrido. por causa desse comportamento pode cair em *máximos locais*, isto é, um estado que não é o meta e onde todos os vizinhos correspondem a um movimento pior.

3.2.4. Têmpera Simulada (Simulated Annealing)

Como a *Subida de Encosta*(seção 3.2.3), é uma busca de melhoramento iterativo. No entanto, pode retroceder para um estado pior para tentar melhorá-lo posteriormente. Nas iterações iniciais não escolhe o melhor estado e sim um movimento aleatório, se a situação melhora, sempre escolhe o estado, senão, associa ao movimento uma probabilidade menor do que 1. Essa probabilidade é dependente de dois parâmetros e decresce com o tempo:

$$P = e^{\frac{-\Delta E}{T}}$$

ΔE é a energia e corresponde ao módulo da diferença entre o nó atual e o nó candidato, T é a temperatura, que deve decrescer conforme o tempo, e P é a probabilidade que se procura.

4. Concepção do Software de Análise

O software desenvolvido foi totalmente escrito na linguagem Python, que é uma linguagem de alto nível considerada muito simples, o que permite um desenvolvimento ágil e de fácil manutenção. A utilização da linguagem é também um diferencial desse programa.

²Busca Gulosa ou *Greedy Search* vem do fato de ela sempre escolher a “maior” chance de encontrar a solução sem se importar com a mesma a longo prazo

³Essa busca leva esse nome devido a seguinte analogia: Encontrar o pico de uma montanha sob densa neblina e sofrendo de amnésia.

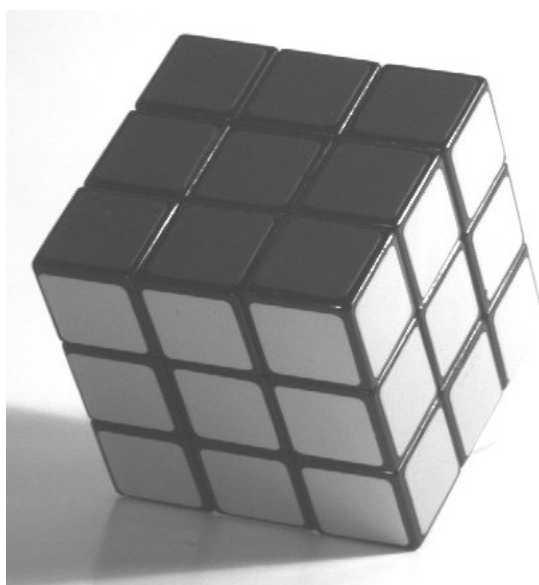


Figura 1. Cubo de Rubik

Escrito na plataforma linux, pode ser executado em qualquer sistema operacional que tenham disponíveis o interpretador python e as bibliotecas utilizadas. Para construir a interface com o usuário foi utilizado pyGTK.

4.1. Representação do Cubo

Um cubo mágico clássico⁴(figura: 1) é representado por 27 sub-cubos, dos quais 26 são visíveis. Cada face do cubo contém 9 quadrados que multiplicados pelas 6 faces obtém-se um total de 54 quadrados. Cada um desses quadrados podem ter uma de seis cores e há 9 quadrados exatamente com uma determinada cor. Na classe Cubo, cada face foi internamente representada como uma lista de nove inteiros correspondendo a quantidade de quadrados que cada face contém, os inteiros em cada lista podem ocupar seis valores diferentes, correspondendo às seis cores possíveis que um quadrado pode ter. Essas listas estão contidas em uma última lista que representa o Cubo de Rubik inteiro.

Os operadores foram definidos como sendo todas as rotações de 90 graus possíveis, ou seja 18 operadores, que são a interface básica para manipular o cubo. Internamente, as faces do cubo foram fixadas, dessa maneira, a qualquer tempo, sempre existirá apenas um eixo X, Y e Z para ser tratado. Na classe foram implementados 3 métodos, 1 para rotacionar cada um dos eixos, cada método recebe como parâmetro a direção da rotação e o plano de rotação no eixo, totalizando os 18 operadores possíveis.

4.2. Representação das Buscas

As buscas implementadas foram a *Busca em Largura*(seção 3.1.1), *Busca em Profundidade*(seção 3.1.2), *Busca com Aprofundamento Iterativo*(seção 3.1.3), *Busca Gulosa*(seção 3.2.1), *A Estrela*(seção 3.2.2), *Subida de Encosta*(seção 3.2.3) e *Anelamento Simulado*(seção 3.2.4). Nas buscas Informadas foram utilizadas duas heurísticas diferentes: *Distância Manhattan 3D* e *Peças fora da Face* totalizando assim um total de 11 buscas diferentes.

⁴retirado de <http://www.ece.cmu.edu/pueschel/examples/puzzles/rubik.jpg>

Nas buscas implementadas o espaço de estado é representado por uma lista que pode ser do tipo FILO (First In, Last Out) ou FIFO (First in, First Out). Cada nó da lista contém um objeto do tipo cubo e uma lista com o histórico de operadores aplicados ao cubo desde o estado inicial até o estado atual. Essa abordagem foi escolhida para podermos descartar os nós avaliados sem perder o caminho até o nó raiz, dessa forma podemos analisar também a quantidade de memória mínima necessária para a busca em um determinado tempo.

Como as buscas tem muito em comum entre si, apenas uma busca necessitou ser implementada inteiramente, todas as outras herdaram dessa primeira ou de uma de suas subclasses, necessitando apenas sobreescrever os métodos que diferem-nas.

Cada busca retorna a quantidade de nós avaliados, a quantidade de nós gerados, a quantidade de nós quando alcançada a meta, a profundidade quando alcançada a meta, o tempo total da busca, e os passos até o estado meta quando o mesmo é encontrado. E, para fins de estudo, pode-se ajustar individualmente a profundidade máxima que uma busca pode chegar.

5. Experimentos Realizados

As buscas implementadas foram aplicadas 10 vezes contra instâncias do cubo onde o estado meta se encontravam nas profundidades 1, 2, 3, 4 e 5. As buscas com melhoria iterativa tiveram sua profundidade limitada a 1000 e todas as demais a profundidade ajustada foi 4.

Assim, dada uma profundidade p , a quantidade de nós n que precisam ser avaliadas para se encontrar o estado meta no pior caso pode ser encontrado calculando-se:

$$n = o^p \quad (1)$$

Onde o é a quantidade de operadores aplicáveis, 18 no caso, que corresponde ao fator de ramificação da árvore de busca.

6. Resultado e Discussão

A partir dos experimentos podemos observar que as Buscas de Melhoramentos Iterativos não são uma boa escolha quando se trata de resolver o problema do Cubo de Rubik. Torna-se difícil encontrar um estado meta uma vez que um movimento que não seja bom seja efetuado. Mesmo em busca em profundidades rasas, a maioria das instâncias testadas não conseguiram se quer encontrar um resultado. De fato, não foi encontrado na literatura nenhuma solução que emprega esse tipo de busca na resolução do problema.

As Buscas Cegas sempre encontram uma solução nas profundidades rasas. Todavia, são as que mais gastam em recursos como memória e tempo, tornando-se inviável para resolver qualquer instância do problema. Em buscas onde o estado meta se encontra em profundidade superior a 5, não foi possível esperar por um estado meta. Uma parte foi por causa da máquina de teste utilizada possuir apenas 256 Mega Bytes de memória RAM, e, quando chega ao limite, a operação de swap no disco rígido reduz tanto o tempo de processamento que não foi possível obter uma resposta em tempo viável.

A Busca Gulosa demonstrou-se eficiente na resolução do problema onde o estado meta se encontre em profundidade inferior a 6. Mesmo assim, observou-se que na mai-

oria das vezes o estado meta poderia ser encontrado em níveis mais baixos do que os adquiridos.

A busca A Estrela demonstrou-se como sendo a melhor busca quando se leva em conta o custo benefício uma vez que encontra uma solução mais rapidamente em relação as buscas cegas e em profundidade menores do que a Busca Cega. Notou-se também que a maioria das abordagens ao problema do Cubo de Rubik emprega uma variante dessa busca denominada *A Estrela em Aprofundamento Iterativo* ou *Iterative Deepening A Star* (IDA*), que é uma mistura da *Busca com Aprofundamento Iterativo* e a própria *A Estrela*. A diferença é que a profundidade máxima é limitada pela função de avaliação empregada.

A tabela abaixo é um exemplo do desempenho de cada busca onde o estado meta se encontrava-se na profundidade 3. Esta é uma tabela como o programa implementado retorna:

Busca	Expandidos	Avaliados	Meta	Memória	tempo
Largura	16938	942	3	15997	2.161
Profundidade	74736	74699	4	38	8.653
A. Iterativo	5562	5543	3	20	0.696
Gulosa (1)	2718	1229	3	1490	0.574
Gulosa (2)	90	6	3	85	0.014
A* (1)	11466	11436	4	31	2,195
A* (2)	396	347	3	50	0.066
H. Climbing (1)	36	2	None	17	0.006
H. Climbing (2)	36	2	None	17	0.005
S. Annealing (1)	5382	582	None	0	0.921
S. Annealing (2)	18000	3853	None	0	3.260

Na primeira coluna o número entre parênteses especifica a heurística usada, 1 representa a heurística *Distância de Manhattan* e 2 representa a heurística *Peças no Lugar*. Na quarta coluna a palavra *None* significa que o estado meta não foi encontrado.

Na primeira coluna está o nome de cada busca implementada, na segunda é a quantidade de nós expandidos durante a busca, na terceira é a quantidade de nós avaliados, na quarta é profundidade em que o estado meta foi encontrado, na quinta está a quantidade de nós na memória quando a busca finalizou e na sexta é o tempo gasto pela busca.

7. Considerações Finais

Apesar de não resolver qualquer instância do Cubo de Rubik, pôde-se ver claramente como cada busca se comporta na resolução do Cubo de Rubik. Então, como previsto, o objetivo de observar como as buscas se comportam quando aplicado a esse tipo de problema foi efetuada com sucesso.

Pode-se incluir facilmente novas busca no software implementado, mesmo que ele não tenha sido concebido para suportar tal característica.

Entre as melhorias que poderiam ser implementadas estão uma quantidade maior de buscas, mais heurísticas e também dividir o problema em sub-problemas, de modo que seja possível resolver uma instância qualquer do cubo, não importando a quantidade de

movimentos de embaralhamento dado.

Referências

- Briesemeister, L., van Schewick, B., and Scheffer, T. (1995). Combination of problem solving and learning from experience (extended abstract).
- Culberson, J. C. and Schaeffer, J. (1996). Searching with pattern databases. In *Canadian Conference on AI*, pages 402–416.
- J., S. and Norvig, P. (1995). *Artificial Intelligence, A Modern Approach*. Prentice Hall, Nova Jersey, first edition.
- Korf, R. (1997). Finding optimal solutions to rubik’s cube using pattern databases. In *Proceedings of the Workshop on Computer Games (W31) at IJCAI-97*, pages 21–26, Nagoya, Japan.
- Randall, K. (1998). Solving rubik’s cube.