# SOLUÇÕES INTELIGENTES PARA O CUBO DE RUBIK

**Bruno Abrantes Esteves**

Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação
Orientador: Prof. Dr. Raul Fonseca Neto

Juiz de Fora, MG
Julho de 2008

# Agradecimentos

Primeiramente, agradeço a minha mãe por todo o carinho, meu pai por seu apoio e Diego por estar sempre presente. Aos amigos, pelo incentivo e força. A Patrícia, por todo amor que houver nessa vida.

# Sumário

# Lista de Reduções

| | |
|---|---|
| AG | Algoritmo Genético |
| B | Back, movimentos de rotação em relação à face |
| D | Down, movimentos de rotação em relação à face |
| F | Front, movimentos de rotação em relação à face |
| IA | Inteligência Artificial |
| ID | *Interative Deepening* |
| L | Left, movimentos de rotação em relação à face |
| R | Right, movimentos de rotação em relação à face |
| U | Up, movimentos de rotação em relação à face |

# Lista de Figuras

# Lista de Tabelas

# Resumo

O Cubo de Rubik continua a representar um desafio para cientistas de todo o mundo devido à complexidade de seus resultados. As principais soluções já encontradas foram abordadas, tendo como base a Teoria dos Conjuntos e a Inteligência Artificial. A aplicação de heurísticas vem sendo útil em métodos de busca, com o objetivo de evitar a exploração do espaço do problema por força bruta. Uma adaptação da distância de Manhattan foi desenvolvida e empregada em substituição à heurística comumente utilizada na solução do Cubo. A fim de representar com fidedignidade o jogo, o modelo desenvolvido foi comparado com um Cubo de Rubik real. Este trabalho propõe uma nova heurística para o Cubo de Rubik, aplicando-a em duas possíveis soluções: uma implementação do Algoritmo A* e um Algoritmo Genético. O A* foi escolhido por ser um algoritmo de eficácia garantida, produzindo resultados ótimos. Como alternativa, a solução através dos algoritmos genéticos foi proposta, trazendo uma inovação na perspectiva adotada no campo de pesquisa de jogos.

**Palavras-chave:** Inteligência Artificial, Cubo de Rubik, Algoritmo Genético, Heurística.

# Capítulo 1
# Introdução

Esse trabalho tem como objetivo apresentar um estudo sobre o cubo de Rubik. Por se mostrar um problema complexo e de difícil resolução, o jogo pode ser considerado um excelente alvo para a Inteligência Artificial, visto que possui simples representação, contando somente com dezoito movimentos possíveis bem como apresenta um espaço de busca razoavelmente grande - mais de $4{,}32 \times 10^{19}$ estados admissíveis. Essas duas características impedem a aplicação de métodos de força bruta como estratégia de busca.

Um obstáculo encontrado foi a modelagem específica e correta do cubo, pois nenhuma estrutura de dados existente é capaz de produzir um ambiente fidedigno ao jogo. A idéia inicial era representar o problema através de um conjunto de vetores inteiros, um para cada posição presente no cubo. Dessa forma, cada estado do problema ocuparia o equivalente a cinqüenta e quatro inteiros, uma vez que cada face contém nove cubies e o cubo é formado por seis faces. Porém, isso constitui uma estrutura cara para se armazenar aos milhares, como seria necessário na aplicação do A*. Destarte, uma nova estrutura foi aplicada de modo que uma cadeia de caracteres – uma *string* – representaria cada face; assim, seria necessário armazenar seis *strings*, resultando em uma economia considerável em relação à representação inicialmente planejada.

Para assegurar a corretude do modelo, testes exaustivos foram realizados e conferidos com um modelo real do cubo, garantindo, assim, que nenhum erro se mantivesse no desenvolvimento do trabalho. Os dezoito movimentos também foram intensamente avaliados e comparados com um exemplar físico do jogo, impedindo a propagação de erros.

O último obstáculo foi a busca por uma heurística consistente e eficaz. A heurística comumente empregada é aquela que leva em conta o número de peças fora do lugar e não a distância delas à sua posição original. Todavia, uma adaptação da distância de manhattan foi utilizada, pois a mesma apresentou resultados melhores que a heurística de uso mais comum. Muitas das heurísticas encontradas são úteis apenas para as soluções humanas como, por

exemplo, os regimes de sub-metas que foram muito utilizados nas soluções referentes ao terceiro capítulo.

Os algoritmos implementados foram o A* e um algoritmo genético. Nesse trabalho, foi feita uma comparação entre as duas técnicas, apresentando os pontos fortes e fracos de cada uma. Todo o código referente a esse trabalho foi desenvolvido usando a linguagem Java, implementada na IDE NetBeans.

Esse trabalho é acompanhado por um CD anexado à contra-capa em que estão disponíveis:

- Código compilado - pronto para a execução dos programas;
- Código fonte dos algoritmos implementados em Java;
- Versão digital do trabalho escrito;

## 1.1 Organização da Monografia

Esse trabalho é dividido em 7 capítulos, incluindo essa introdução. No segundo capítulo, é apresentada uma lista de qualidades dos jogos que lhes possibilitam a simulação de situações reais. Uma descrição detalhada do cubo de Rubik é apresentada, demonstrando o tamanho do espaço de busca e a complexidade do jogo.

O terceiro capítulo trata da Inteligência Artificial. A sua interdisciplinaridade aparece desde antes de sua formação, envolvendo conceitos de Filosofia, Psicologia e Matemática. Um pequeno histórico e os principais requisitos da Inteligência Artificial completam o capítulo.

No quarto capítulo, foi elaborado um resumo das principais resoluções em que são descritas as mais importantes e conhecidas soluções propostas para o cubo de Rubik. A maioria das antigas resoluções busca provar o menor número possível de movimentos necessários para se resolver qualquer instância do cubo, enquanto poucos trabalhos estão interessados em seu algoritmo propriamente.

No quinto capítulo, a primeira solução é proposta. Demonstrou-se como as heurísticas permitem a representação do conhecimento em termos computacionais. Ademais, a obtenção, as características e avaliação da qualidade das heurísticas foram explicadas bem como o primeiro algoritmo utilizado, com ilustrações de trechos do código.

Os Algoritmos Genéticos que serviram de base para a segunda solução proposta foram tratados no sexto capítulo. Ele tem início com um pequeno histórico da computação evolucionista e aborda seus conceitos e principais técnicas. Por fim, a segunda implementação é demonstrada, utilizando-se novamente de trechos do código para elucidar a explicação.

No sétimo capítulo, as técnicas utilizadas foram comparadas, sendo mostradas as vantagens e desvantagens de cada método assim como também alguns refinamentos que poderiam melhorar seu desempenho.

# Capítulo 2

# Jogos

## 2.1 Jogos como Objeto de Estudo

Os jogos são alvo de estudo de ciências, como a Inteligência Artificial (I.A.), por diversas razões. A fácil abstração e representação das situações de um jogo – que quase sempre podem ser resumidos em um conjunto de peças, um tabuleiro e outros objetos simples – permite a construção de modelos lógicos condizentes com o aspecto real do problema. Os jogos ajudaram a desenvolver a capacidade mental humana ao longo de sua evolução: competições mentais como o Xadrez e o Go apresentam uma competição abstrata em que o melhor jogador vence, forçando uma evolução da capacidade de raciocínio e aquisição de conhecimento através de experiências (partidas) passadas.

Como apresentam regras sólidas e bem estruturadas, os jogos são capazes de simplificar o cenário se comparados a situações reais em que uma infinidade de regras pode ser quebrada ou interpretada de maneira errônea. Além disso, eles possuem, em geral, poucos operadores válidos que consistem em movimentar ou substituir as peças entre si; caso as regras sejam seguidas, os operadores válidos só podem gerar estados válidos, mantendo sempre a consistência do problema.

A clareza e simplicidade dos objetivos de um jogo como, por exemplo, derrotar o rei adversário, no xadrez, ou conseguir ordenar as 15 peças no Jogo dos 15 facilita sua representação. Desse modo, a maioria das resoluções de jogos consiste em uma seqüência de jogadas a partir de um dado estado inicial até a solução. O problema é justamente a busca do estado final, pois jogos simples como o Jogo dos 15 chegam a ter $10^{13}$ estados válidos, segundo Korf e Schultze (2005). Portanto, torna-se necessário um artifício que viabilize a busca pelo estado último - especificamente, a Inteligência Artificial.

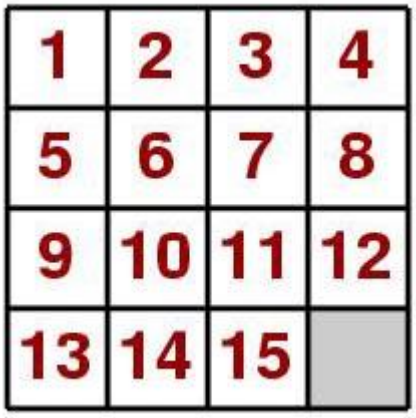


**Figura 2.1 Solução do jogo dos 15**

Jogos são representações simples, mas sua solução pode servir para problemas reais. A transição de uma equipe entre diferentes projetos pode ser generalizada como uma situação de travessia, como o clássico problema dos Canibais e Missionários assim como um caso de logística pode ser tratado como um exemplo do Caixeiro Viajante.

**Figura 2.2 Missionários e Canibais: Todos devem atravessar o rio; porém não pode haver um número de canibais superior ao de monges em nenhuma das bordas do rio.**

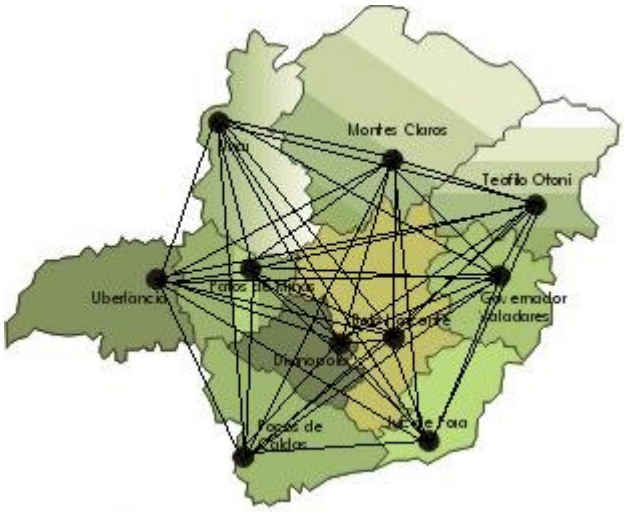


**Figura 2.3 Caixeiro Viajante: Exemplo baseado em Minas gerais, onde os traços representam as rotas comerciais. Devemos percorrer todas as cidades com o menor caminho possível e passando uma única vez por cada cidade.**

A programação de jogos surgiu concomitantemente com a Inteligência Artificial. No início da década de 50, Alan Turing e Claude Shannon programaram, de forma simultânea e independente, um jogo de Xadrez. Ele foi escolhido por motivos simples como o fato das regras serem simples de programar e interpretar, possibilitando a análise de muitos passos a frente bem como a evolução da habilidade do jogador só ser possível mediante aprendizado; logo se a máquina melhorasse, ela estaria adquirindo conhecimento.

Por fim, normalmente os jogos são problemas que apresentam respostas rápidas se comparados a outras áreas de estudo como a climatologia e a certas áreas da biologia. Por exemplo, semanas de computação paralela pode ser considerado um tempo curto se comparadas aos meses, ou até anos, necessários para o crescimento de uma colônia de fungos, ou às décadas de coletas e análise de dados para se caracterizar o clima de uma região.

## 2.2 Cubo de Rubik

O cubo mágico é formado por 26 pequenos cubos e um eixo central, que serve de base para a rotação das faces do jogo. Existem três tipos de classificação dos cubos que formam o jogo: os *corners cubies* - cubos que possuem três faces à mostra, são encontrados nas bordas mais externas (vértices) do cubo de Rubik -; os *edge cubies* - peças intermediárias entre dois

*corners*, ou seja, localizam-se nas arestas do cubo; e os *central cubies* - cubos que se localizam no centro de cada face e não se movem em relação a ela, podendo girar somente em sentido horário e anti-horário. Assim, o cubo de Rubik é formado por oito corners cubies, doze edge cubies e seis central cubies. Os termos *corners*, *edge* e *central* serão usados para representar os *corners cubies*, *edge cubies* e *central cubies*, respectivamente.

Os corner representam três faces, podendo ser apresentadas em três orientações diferentes entre si para cada um dos oito espaços que ocupem. Dessa forma, selecionando aleatoriamente um corner cubie e colocando-o em uma posição e uma orientação qualquer chega-se a vinte e quatro (três faces e oito espaços) configurações possíveis, um segundo cubie teria vinte e um estados, até o último corner cubie que, pelo cubo original, só poderia ocupar um espaço, pois, se o mesmo ocupar alguma das outras duas possíveis configurações, o problema deixa de ser gerado a partir da solução. Desse modo, o total de posições orientadas possíveis para os *corners* seria de oitenta e oito milhões cento e setenta e nove mil e oitocentos e quarenta estados (ou simplesmente $(8!)x(3^7)$).

Similarmente, pode-se calcular o total de posições válidas dos *edge*. Cada cubie pode preencher um dos doze espaços em duas posições distintas; logo, o primeiro pode possuir vinte e quatro estados, o segundo pode preencher um dos onze espaços restantes, totalizando vinte e duas configurações e assim por diante. Os dois últimos *cubies* a serem inseridos já têm uma orientação pré-definida; caso desrespeitem essa norma, o cubo não pode ser gerado a partir do cubo original e, conseqüentemente, nunca chegará ao resultado final. Assim, tem-se um total de novecentos e oitenta bilhões novecentos e noventa e cinco milhões duzentos e setenta e seis mil e oitocentas posições válidas (ou $(12!)x(2^{10})$).

Destarte, chega-se ao incrível número de estados diferentes entre si que um cubo de Rubik pode apresentar: quarenta e três qüinqüilhões, duzentos e cinqüenta e dois qüatrilhões, três trilhões, duzentos e setenta e quatro bilhões, quatrocentos e oitenta e nove milhões, oitocentos e cinqüenta e seis mil $((8!)x(3^7))x((12!)x(2^{10}))$ diferentes configurações; e somente uma é a solução. A chance de se encontrar aleatoriamente a solução do Cubo é muito menor que a de se ganhar na loteria, as chances da mega-sena, por exemplo, são de 1 jogo premiado para $3{,}6x10^{10}$ jogos possíveis, contra 1 solução em $4{,}32x10^{19}$ estados possíveis.

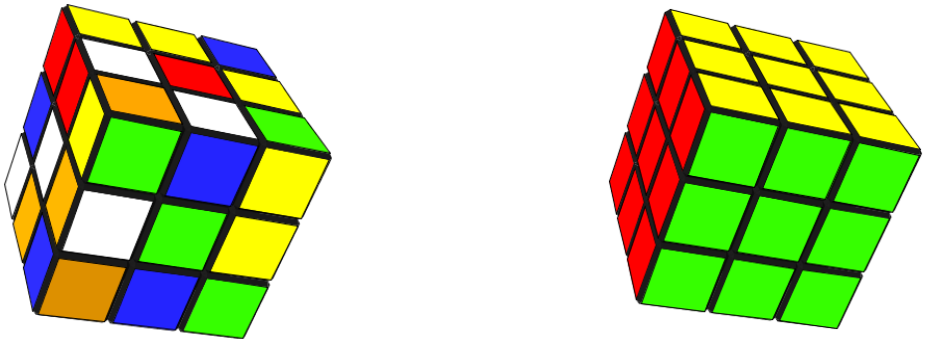

**Figura 2.4 Exemplos do Cubo: embaralhado (à esquerda) e o único estado solução (à direita)**

Para se ter uma noção do tamanho desse número, deve-se compará-lo ao número de avogrado. Ele equivale a aproximadamente um quatorze mil avos da constante física que representa o número de moléculas existente um uma molécula-grama de um corpo - aproximadamente $6{,}0251 \times 10^{23}$ contra $4{,}32 \times 10^{19}$ estados do Cubo mágico.

### 2.2.1 Histórico do Cubo

Ernö Rubik foi aluno de artes (posteriormente, mudou de curso para Arquitetura) e atua como professor em Budapeste. Percebendo que grande parte de seus alunos (mesmo os pós-graduados) possuía dificuldades de visualizar objetos tridimensionais, buscou novas técnicas e exercícios de visualização. Filho de um engenheiro da aeronáutica, Rubik sempre apresentou interesse em quebra-cabeças e jogos de raciocínio, fonte de sua inspiração. Patenteado em trinta de janeiro de 1975, seu primeiro Cubo foi apresentado à Trial (antiga empresa húngara de criação e distribuição de brinquedos) a qual produziu e vendeu inicialmente cerca de cinco mil cubos mágicos. O professor foi premiado em 1978 por sua invenção, fato que estimulou fortemente a demanda pelos Cubos.

Em meados de 1979, a gigante norte-americana Ideal Toy adquiriu a distribuição do brinquedo por toda a América, agora com o novo nome de ‘cubo de Rubik’. Ele foi remodelado para melhor se adaptar ao mercado americano, tendo suas cores se tornado mais vivas, suas faces mais fáceis de se girar bem como poderia ser desmontado e remontado para que pudesse voltar à solução. Essas modificações deram resultado, já que, em menos de dois anos, as vendas no mercado americano ultrapassaram a marca de quinze milhões de peças.

Ainda hoje, o cubo de Rubik possui seus aficcionados, que continuam a produzir mais e mais quebra-cabeças baseados no cubo. Como exemplos, pode-se citar o cubo de 5x5x5 (cinco cubículos, ao invés dos 3 originais) e um curioso Dodecaedro regular (um sólido de Platão constituído de 12 faces iguais).

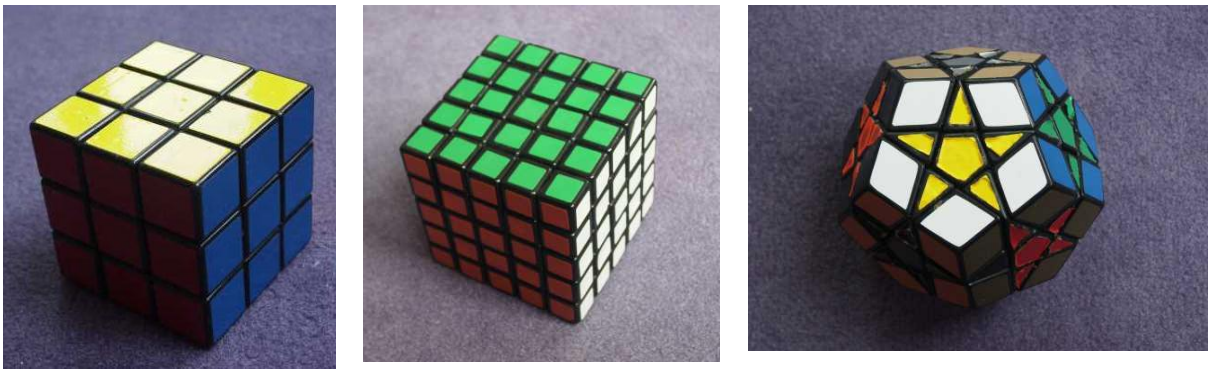

**Figura 2.5 Outros Exemplos do Cubo: original (imagem à esquerda), cubo 5x5x5 (centro) e um dodecaedro baseado no cubo de Rubik (à direita).**

### 2.2.2 Notação utilizada

Esse trabalho segue o padrão especificado em Frey e Singmaster (1982), um dos primeiros trabalhos a tratar o Cubo de Rubik como objeto de estudo e o primeiro a padronizar várias nomenclaturas usadas pelo meio científico. O número de Deus (God's number) e o algoritmo de Deus (God's algorithm) são referências ao número mínimo (ou suficiente) de movimentos que pode solucionar todos os estados do Cubo de Rubik. Os cientistas sempre supuseram que ele estava situado entre vinte e trinta movimentos. Através da teoria dos conjuntos, Rokicki (2008) provou que a marca atual é de vinte e cinco passos suficientes, mas muitos acreditam que o número de movimentos necessários deve ser ainda menor. São conhecidas mais de trinta e seis mil resoluções que necessitam de exatos 20 movimentos; porém não foi encontrado nenhum caso que exija para sua solução 21 ou mais movimentos necessários (ROKICKI, 2006). No entanto, esse conjunto é muito pequeno se considerarmos todo o espaço de busca gerado pelos movimentos a partir de um cubo de Rubik totalmente embaralhado. O algoritmo de Deus seria o algoritmo capaz de sempre encontrar a melhor solução, ou seja, encontraria a solução em, no máximo, um número de passos igual ao número de Deus.

Geralmente, os estudos atuais são baseados na métrica de rotação da face (*face turn metric*) que considera movimentos em qualquer face no sentido de noventa graus (horária e anti-horária) e cento e oitenta graus como um só movimento. Outras métricas, como a métrica de rotação quadrada da face (*quarter-turn metric*), consideram válidas somente a rotação em noventa graus como movimento; sendo que, para essa métrica, rotações de cento e oitenta graus são contadas como dois movimentos. Bons trabalhos podem ser encontrados utilizando a métrica de rotação quadrada, como os trabalhos de Radu (2007) e Passini (1991), porém é mais comum encontrar artigos sobre a primeira métrica, utilizada também no presente trabalho.

Assim como na convenção padrão adotada, as faces são nomeadas pela primeira letra de seu nome em inglês: ***U****p* (face de cima), ***D****own* (face de baixo), ***L****eft* (face esquerda), ***R****ight* (face direita), ***F****ront* (face mais próxima do usuário, ou a 'frente' do Cubo) e ***B****ack* (face 'escondida' do usuário, as 'costas' do cubo ou ainda a mais distante), ou simplesmente U, D, L, R, F e B. Os movimentos possíveis serão representados pela face girada mais um pós-fixo; desse modo, girar a face de cima do cubo no sentido horário em 90º será representado por U, U1 ou ainda U+. Uma rotação de 180º na face de baixo será representado por D2 ou D180, e um movimento da face esquerda em 90º no sentido anti-horário será representado por L', L3 ou L-. Essas nomenclaturas serão usadas ao longo de todo o artigo, sendo que o padrão numeral (U1, U2 e U3) será usado na implementação desenvolvida junto desse trabalho.

Um grupo específico de estados do cubo em que somente alguns movimentos são permitidos, é dado por G = <lista de operadores disponíveis>. Desse modo, G = <U , D2> só permitiria que a face de cima girasse em 90º no sentido horário e que a face de baixo girasse sempre 180º. Uma seqüência de operadores aplicados é chamada de *maneuver* como, por exemplo, F2 B2 R2 L2 U2 D2 (conhecida como *pons asinorum* – referência à obra Euclidiana) que produz, a partir do cubo inicial, uma espécie de 'xadrez' cúbico em que todas as faces ficam em um estado de quadriculado bicolor ou, ainda, U D' R L' F B' U D' (*six spots*) em que cada face tem todas as cores iguais, exceto o *central cubie*.

A distância entre dois estados do cubo representa o menor número de movimentos entre as duas instâncias. O diâmetro de um grupo é a menor distância que liga qualquer elemento ao estado mais distante do mesmo. Devido à simetria do Cubo, caso se considere

como um grupo todas as posições que o mesmo pode assumir, o diâmetro seria o número de Deus, tão buscado pelos cientistas.

# Capítulo 3
# Inteligência Artificial

## 3.1 Conceitos

Inteligência - do latim *intellegere*, capacidade de entendimento – é uma idéia antiga, nascida na Filosofia antiga e, ainda hoje, sem um conceito formal. Diferentes concepções sobre o que é inteligência existem na atualidade e tentativas de medição já foram elaboradas, como o famoso teste de QI; seus resultados, no entanto, não são aceitos por todos os estudiosos. A Psicologia possui duas grandes vertentes sobre a inteligência: uma prega que é uma característica individual que possui vários fatores determinantes, enquanto a segunda defende a idéia de que a inteligência é uma habilidade mental que envolve diversas capacidades.

Neisser *et al* (1996) é um dos mais representativos trabalhos que defende a inteligência como sendo uma característica pessoal e intransferível. O estudo mostra resultados obtidos em diversos testes aplicados a uma grande variedade de pessoas. Esse ramo defende a inteligência como capacidade mental de aprendizagem e entendimento, sendo que essas características dependem da combinação de vários fatores como o sexo - embora os homens tenham mais neurônios, a diferença entre os resultados é mínima -, fatores sócio-econômicos - alimentação adequada, acesso à cultura e tempo de estudo -, consumo de drogas - uso da mãe em período pré-natal e os resultados nos filhos, o extermínio de neurônios causado por consumo contínuo de drogas entre outros.

A segunda definição foi formalizada em *Mainstream Science on Intelligence* (1994), defendendo a idéia de que a aptidão intelectual é a soma de várias outras habilidades como a capacidade de resolver problemas, planejar ações e o aprendizado com experiências anteriores. Não consiste em aprendizado acadêmico ou literário, mas sim na capacidade mais ampla de compreensão do mundo à sua volta. Um novo estudo – muito aceito na Psicologia - divide a inteligência em sete diferentes tipos: lingüística (talento com linguagens escritas e faladas), raciocínio lógico-matemático (facilidade para investigação e manipulação numérica), visão espacial (capacidade de visualizar padrões e imagens), aptidão musical (competência em definir sons e suas características), corpóreo-cinestésica (habilidade caracterizada pela

expressão corporal), interpessoal (expressa na capacidade de se conhecer outras pessoas, seus sentimentos e motivações) e intrapessoal (auto-conhecimento, permitindo a auto-motivação).

Desse modo, pode-se perceber que conceituar inteligência não é tarefa nada fácil, mas ambas as correntes concordam que a inteligência está ligada à capacidade de aprendizagem e entendimento. Por outro lado, a idéia de 'artificial' é facilmente identificada como sendo tudo aquilo que não é natural. Um coração artificial, por exemplo, é um aparelho usado para substituir um coração, órgão naturalmente formado nos animais.

Um computador moderno é capaz de executar milhares de dados em poucos instantes; desse modo, ele pode resolver vários problemas demasiadamente complicados para o ser humano. A inteligência permite encontrar bons resultados, ou até mesmo o melhor, de forma razoavelmente barata em diversos problemas como num jogo de xadrez.. Unindo essas duas características, tem-se uma ferramenta muito poderosa, capaz de tomar decisões, aprender com experiências passadas e até mesmo jogar xadrez melhor que os seres humanos; isso e muito mais é chamado de inteligência artificial. O segundo conceito de inteligência, que engloba diversos campos, não é estudo somente da ciência da computação; já existem próteses robóticas capazes de imitar o movimento de um membro (inteligência corpórea-cinestésica) além dos conhecidos tradutores automáticos e corretores ortográficos (lingüística).

Embora não seja perfeita, a I.A. também vem sendo usada para simular diversos campos como, por exemplo, o aprendizado humano no qual vem apresentando avanços significativos. A máquina seria então capaz de raciocinar como o ser humano? Essa questão é discutida por filósofos, psicólogos e cientistas da computação há bastante tempo. Antes desses questionamentos, no entanto, Alan Turing propôs um teste que, mais tarde, seria conhecido como Teste de Turing o qual consiste em colocar uma pessoa para conversar ao mesmo tempo com dois desconhecidos através de um bate-papo instantâneo, por exemplo – sendo um deles um ser humano e o outro uma máquina. Através do teste, busca-se descobrir por quanto tempo a máquina consegue imitar o comportamento humano; se ela não puder ser descoberta, Turing considerava que ela poderia realmente pensar. O teste inclusive é destaque no filme Blade Runner em que um interrogatório era aplicado para determinar se o alvo era um humano ou andróide.

O silogismo de Aristóteles iniciou a chamada racionalidade, ou capacidade de raciocínio lógico, segundo a qual através de algumas premissas, é possível deduzir uma

conclusão. A capacidade de decisão pode ser simulada a partir da representação do conhecimento em forma de premissas. Ela pode ser implementada através dos princípios da lógica matemática como as tabelas verdade e as leis de De Morgan. Contudo, transformar informações reais em notação matemática não é simples, sobretudo quando o conhecimento não é totalmente preciso, mas representa o início da racionalidade artificial.

Premissa 01: O homem é um vertebrado.
Premissa 02: Todo vertebrado é um ser vivo.
Conclusão: O homem é um ser vivo.

**Figura 3.1 Exemplo de Silogismo.**

## 3.2 Interdisciplinaridade

Embora seja um campo recente, a Inteligência Artificial herdou princípios, idéias, conceitos e técnicas de várias outras ciências, sendo uma matéria interdisciplinar desde sua origem. Atualmente, ela vem sendo usada em domínios diversos como no diagnóstico de doenças (medicina), na exploração do código genético (biologia), na prova de teoremas matemáticos, na decodificação de símbolos (lingüística), na redução de custos determinando melhores rotas (logística) entre outros campos.

Sem a ciência da computação, a aplicabilidade da inteligência artificial seria bastante limitada, uma vez que sua aplicação depende basicamente da existência de dois fatores: inteligência e o artefato que a faz artificial.

Desde os primeiros programas feitos por Ada Lovelace para a *Analytical Engine* (AE) de Babbage, ela já imaginava se aquela máquina seria capaz de jogar xadrez. Décadas depois, os primeiros programas capazes de executar isso foram programados por Alan Turing e Claude Shanon, anos após o surgimento dos primeiros computadores. Essa questão foi finalizada em 1997, quando o computador Deep Blue venceu o campeão mundial Kasparov, provando que não só que o computador pode jogar como também derrotar o ser humano.

O silogismo é um dos princípios herdados do campo filosófico. Há também o conflito entre as posições ideológicas do dualismo e do materialismo. Enquanto a primeira corrente defende a divisão do cérebro entre espírito (ou raciocínio) e instinto, a segunda prega que toda forma de raciocínio pode ser reduzida a simples processos físicos. Contudo, é possível estabelecer um meio-termo entre o materialismo e o dualismo. O raciocínio é um processo natural-fisiológico, mas não pode ser tomado de forma tão trivial como no modelo de

pensamento puramente físico de Leibniz. Uma aproximação dessa ‘forma intermediária’ foi usada por Minsk e Edmonds em seu protótipo de rede neural.

George Boole contribuiu com a I.A. através de sua lógica de boole, uma tentativa de representar a lógica de Aristóteles formalmente. Outra contribuição matemática foi adquirida através dos problemas propostos por Hilbert, visto que, em resposta aos mesmos, Kurt Gödel provou que alguns problemas não podiam ser solucionados. Alan Turing, ao projetar sua máquina de estados computáveis (máquina de Turing), classificou os problemas em computáveis e não-computáveis; além desses dois tipos, existem os NP-completos, descobertos por Steven Cook e Richard Karp. A maioria dos problemas do I.A. se encaixa na classificação de NP-completos.

## 3.3 Histórico

McCulloch e Pitts publicaram o primeiro artigo que trata de conhecimento e computação. Eles discorreram sobre princípios fisiológicos neurais (referindo-se ao cérebro e aos neurônios), a lógica proposital e a teoria da computação de Turing. A dupla propôs a representação de um neurônio artificial que possuiria dois estados - ‘ligado’ e ‘desligado’. Estímulos mudariam o comportamento dos neurônios e seriam passados adiante, transmitindo a informação por toda uma ‘rede de neurônios’.

O trabalho dos cientistas foi usado simultaneamente por Alan Turing e Claude Shannon na tentativa de construir um programa capaz de jogar xadrez. Nessa mesma época, nasceu o primeiro protótipo de rede neural, construída por Marvim Minsk e Dean Edmonds. A estrutura era formada por mais de três mil tubos a vácuo e simulava uma rede de quarenta neurônios. Infelizmente, o trabalho não apresentou os resultados esperados para a época, não sendo creditado como deveria, mas foi muito importante no estudo formal das redes neurais pouco mais de duas décadas depois do projeto inicial.

O termo Inteligência Artificial foi adotado em uma convenção organizada por John McCarthy. O cientista reuniu grandes pesquisadores interessados em metadados, redes neurais e inteligência. Embora o encontro não tenha realizado nenhum trabalho sólido, ele serviu como ligação entre grandes centros de pesquisa, como a IBM, o *MIT* e a *Carnigie Tech*.

Ainda na convenção, foi apresentado o *Logic Theorist*, ou simplesmente LT, programa capaz de pensar através de uma representação anumeral, conseguindo resolver problemas

complexos. Allen Newell e Herbert Simon, criadores do LT, mostraram que o seu software podia resolver grande parte dos teoremas apresentados a ele - retirados do segundo capítulo do livro *Principia Mathematica*. Russel, um dos autores do livro, inclusive mostrou que o LT foi capaz de provar teoremas de forma mais sucinta.

O próximo programa que se destacou foi o *General Problem Solver* ou GPS. Seu comportamento era similar ao de um ser humano, sendo um grande pioneiro para a época. Devido a sérias limitações de hardware, grande parte dos problemas eram divididos em sub-metas, aproximando-o ainda mais do comportamento humano.

Herbert Gelernter construiu o *Geometry Theorem Prover* (GTP) que, de modo similar ao LT, conseguia deduzir e provar teoremas da geometria através de axiomas simples. O grande passo dado pelo GTP foi a criação de uma representação numérica para um diagrama como resultado da prova de um teorema. Essa representação foi adotada no próprio GTP por Gelernter, de modo que ele sempre verificava os diagramas já percorridos de modo a economizar passos intermediários presentes em outros teoremas.

O primeiro programa capaz de simular um jogo real foi o *checkers* (jogo de damas americano) de Arthur Samuel. O Software era capaz de disputar partidas com jogadores profissionais, provando que o comportamento do programa independe do conhecimento apresentado por seu criador, já que a sua habilidade de jogo era muito maior que a de seu programador.

John McCarthy, o idealizador do congresso que dá nome à Inteligência Artificial, contribuiu com duas grandes descobertas. A primeira foi a LISP, linguagem de programação de alto nível, que se tornou referência para a programação no campo da IA. A segunda foi a criação e aplicação de *time-sharing*, um modo de aproveitar o tempo de processamento dos computadores, já que os recursos eram muito caros. *Advice Taker* (AT), outra contribuição de McCarthy, foi outro programa que seguiu a mesma linha de criação de sentenças a partir de axiomas iniciais. De forma diferente de seus predecessores, o AT era capaz de lidar com ‘conhecimentos mundano’; assim, de posse de um certo conjunto de informações, ele era capaz de traçar o planejamento de um vôo desde a compra da passagem ao trajeto do embarque.

Minsky, grande companheiro de McCarthy, coordenou um grupo de estudos em que cada grupo de indivíduos se centrava em um certo campo, chamados de micro-mundos. Esse

estudo profundo de pequenas áreas gerou uma série de programas inteligentes como o *Analogy Program* de Tom Evan, capaz de solucionar problemas de analogia geométrica como aqueles presentes em testes de QI.

Duas décadas depois da apresentação, a proposta sobre redes neurais de Pitts e McCulloch foi reconhecida. Winograd e Cowan prosseguiram os estudos iniciados pela dupla anterior, mostrando que um grande número de elementos simples pode, coletivamente, representar um conceito individual forte. Essa idéia ampliava o trabalho de vinte anos antes, conferindo-lhe maior robustez e capacidade de paralelismo e, finalmente, reconhecendo o grande trabalho dos pesquisadores.

Como os recursos computacionais eram caros e escassos, os pesquisadores alimentaram a ilusão de que, com poder computacional infinito, grande parte dos problemas da IA seriam resolvidos. A teoria da *Machine Evolution* (que resultaria nos atuais Algoritmos Evolutivos) baseava-se na idéia de alterar pequenos trechos randomicamente e em seguida aplicá-los, buscando melhorias na performance. Todavia, ela foi responsável por derrubar o mito de que, mesmo com poder computacional infinito, o problema seria resolvido, uma vez que, com centenas de horas de processamento gastas, não houve nenhum aperfeiçoamento do programa.

A idéia de individualizar um problema para realizar uma determinada tarefa deu origem aos Sistemas Especialistas - programas especificamente projetados para um certo ambiente. O *Dendral* foi o primeiro software a contar com conhecimentos extremamente específicos em sua área e mostrou ótimos resultados; ele era capaz de inferir a estrutura molecular de uma substância a partir de resultados obtidos por um espectrômetro. Moléculas curtas tinham uma boa margem de acerto, enquanto que as maiores geravam muitos erros. Consultando um especialista químico, a equipe de desenvolvimento refinou o processo de identificação; assim, a partir da localização de certas estruturas moleculares - como o radical –OH presente nos álcoois -, muitos compostos eram eliminados, aumentando a acurácia do problema.

Com mais de quatrocentas e cinqüenta regras, o *Mycin* foi o próximo grande especialista. Ele era capaz de diagnosticar doenças a partir de amostras de sangue com uma taxa de acerto maior que a de médicos recém-formados. Como a maioria das doenças não

segue um padrão restrito, o Mycin não se mostrou tão forte quanto o Dendral, mas ainda assim foi um grande destaque.

Talvez o acontecimento recente de maior destaque no campo da AI seja o desafio do DeepBlue, o computador-jogador de xadrez da IBM e o campeão mundial, o russo Garry Kasparov. O primeiro embate aconteceu em 1996, com três vitórias do jogador russo contra uma do computador e dois empates. Cerca de um ano depois, ocorreu a revanche do DeepBlue; ele venceu duas partidas contra uma de Kasparov (e três empates). Esse evento foi mundialmente publicado como a primeira vez que a máquina vence o homem, embora o sucessor do *checkers* de Samuel também tenha se sagrado campeão de torneios bem antes. Os jogos estão disponíveis em Fischer (1996) e Fischer (1997), onde podem ser revividos a qualquer momento.

**Tabela 3.1 – Resumo das Partidas de 1996:** Kasparov **3 x 1** Deep Blue

| Jogo | Peças Brancas | Peças Pretas | Jogador Vitorioso |
|---|---|---|---|
| 1 | Deep Blue | Kasparov | Deep Blue |
| 2 | Kasparov | Deep Blue | Kasparov |
| 3 | Deep Blue | Kasparov | Empate |
| 4 | Kasparov | Deep Blue | Empate |
| 5 | Deep Blue | Kasparov | Kasparov |
| 6 | Kasparov | Deep Blue | Kasparov |

**Tabela 3.2 – Resumo das Partidas de 1997:** Deep Blue **2 x 1** Kasparov

| Jogo | Peças Brancas | Peças Pretas | Jogador Vitorioso |
|---|---|---|---|
| 1 | Kasparov | Deep Blue | Kasparov |
| 2 | Deep Blue | Kasparov | Deep Blue |
| 3 | Kasparov | Deep Blue | Empate |
| 4 | Deep Blue | Kasparov | Empate |
| 5 | Kasparov | Deep Blue | Empate |
| 6 | Deep Blue | Kasparov | Deep Blue |

## 3.4 Fundamentos

Para um entendimento mais profundo do trabalho proposto, é necessário abordar alguns pré-requisitos que a própria Inteligência Artificial possuiu. Um deles é a teoria dos grafos, importante principalmente como forma de notação dos estados do problema, e a sua

evolução no ambiente. Igualmente importantes são os métodos não-informados, muito usados antes da aplicação de conhecimento no processo de busca.

### 3.4.1 Teoria dos Grafos

Possivelmente, o maior pré-requisito computacional para a plena compreensão da Inteligência Artificial seja a Teoria dos Grafos. Um grafo normalmente é a representação de um estado, sendo constituído por um conjunto de pontos (vértices) ligados entre si por linhas (arestas), e é representado por G (V,A) em que V é um conjunto de vértices (também chamados de nós, nodos ou, em alguns casos, estados), A é o conjunto de pares de V, representando a aresta que liga os nós. Os elementos de A são representados através da notação v1~v2, onde v1 e v2 são os nodos interligados.

Dois vértices são chamados vizinhos (ou adjacentes) se possuem pelo menos uma aresta que una os dois. O grau de um vértice é dado pelo número de arestas que o mesmo possui. Em grafos direcionados, o grau é dividido em dois tipos: Grau de Emissão que representa o número de arestas que partem do vértice, enquanto o Grau de Recepção indica o número de arestas que chegam ao nó. Quando o grau de recepção de um vértice é zero, ele pode ser chamado de fonte; por outro lado, se o grau de emissão for nulo, o nodo é considerado um sumidouro.

Uma característica interessante que pode ser atribuída a uma aresta é um custo (ou peso). Desse modo, é possível traçar rotas diferentes pelo grafo, considerando os custos mais baratos. Uma possível metáfora aos custos poderia ser, por exemplo, o tráfego de carros em uma determinada cidade; dependendo do horário, pode ser mais proveitoso utilizar caminhos mais longos (passando por mais vértices) para escapar de engarrafamentos nas vias principais que tem o custo maior.

Alguns tipos específicos de grafos são mais interessantes para o estudo da Inteligência Artificial, sobretudo as árvores. Árvores são grafos acíclicos e conexos em que o nodo fonte é chamado de raiz, enquanto que os vértices sumidouros recebem o nome de folhas.

### 3.4.2 Métodos de Busca

A grande maioria dos problemas se resume a buscar uma solução no espaço de estados, que normalmente é representada por um caminho que conecta o estado-inicial do problema ao estado-final. É chamado de espaço de busca o conjunto de estados possíveis que um problema pode assumir; tomando-se como exemplo o jogo dos 15, o espaço do problema é formado por todas as permutações das dezesseis peças (quinze números e o espaço) no tabuleiro do jogo. Normalmente ele é representado por um grafo, enquanto a busca pelo espaço é representada através de uma arvore, o que possibilita uma fácil representação de todos os caminhos possíveis.

Os métodos de busca descritos ao longo deste capítulo são chamados de não-informados por não serem capazes de distinguir qual estado é melhor que o outro. Outros métodos explorados nesse trabalho são a busca heurística, tratada no quinto capítulo, e a exploratória, representada através dos algoritmos genéticos na sexta parte.

Dois parâmetros apresentados pela árvore de busca são muito importantes em problemas grandes, sobretudo em métodos de busca cegos: **b** e **d**. O fator de ramificação (branching factor), ou **b**, é a média de operadores que pode ser aplicado a qualquer estado, gerando B filhos; no cubo de Rubik, por exemplo, o fator de ramificação médio é 13,5 (KORF, 1997). **D**, por sua vez, é a profundidade (Deep) de um estado seja ele final ou não. O termo ‘gerar’ um nó significa criá-lo, enquanto o ‘expandir’ é a geração de todos os seus filhos, mas não seu percorrimento. Em geral, as técnicas não-informadas têm como grande falha o tempo, já que devem executar uma busca exaustiva por todo o espaço.

Os métodos de busca não-informados também são conhecidos como métodos de força-bruta, uma vez que percorrem todos os caminhos sem nenhuma informação, possuindo somente o estado inicial, uma condição de parada e um conjunto válido de operadores. Os principais algoritmos de exploração através da força bruta são as buscas em profundidade, em largura, em profundidade iterativa e bi-direcional.

### 3.4.2.1 Busca em Profundidade (*Depth-First Search*)

A estratégia usada pela busca em profundidade é a de explorar sempre o nó mais recentemente gerado. Quando encontra uma folha, ele volta para o nó mais profundo que ainda não foi explorado e recomeça a busca a partir dele. Essa técnica exige somente o armazenamento da rota atual, ou seja, armazena-se, no máximo, **b** x **d** nodos.

A busca em profundidade geralmente é implementada a partir da estrutura de uma pilha em que um nó é explorado e seus filhos são inseridos no topo da pilha para serem imediatamente percorridos. Também é comumente implementada através de estruturas recursivas.

Caso a árvore seja a representação de um grafo cíclico, existe uma grande chance da busca em profundidade ficar presa em um loop infinito. Para evitar essa possibilidade, pode ser imposto um D limite; no caso do cubo de Rubik, o ideal seria o valor de 26 níveis, uma vez que nenhum cubo precisa de mais que 25 movimentos para ser resolvido, conforme demonstrado por Rokicki (2008).

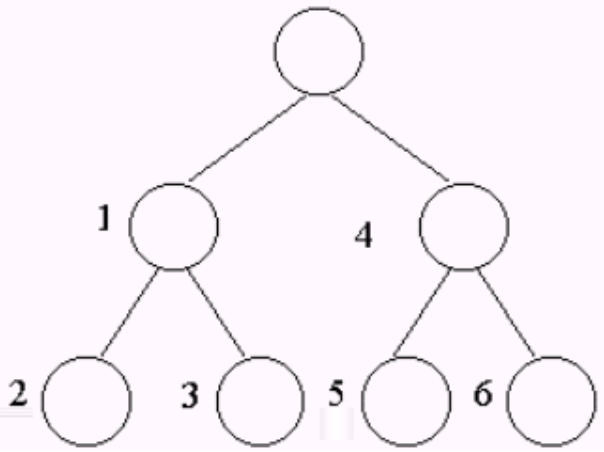


**Figura 3.2 Ordem de Exploração em Profundidade.**

### 3.4.2.2 Busca em Largura (*Breadth-First Search*)

A busca em largura possui um conceito simples capaz de garantir sempre a melhor solução. Com base nesse método, a expansão de um nó no nível **n** só pode ser efetuada quando todo o nível ‘n-1’ estiver completamente expandido. Esse comportamento é similar ao de uma fila, ou seja, ao explorar um nó, ele é retirado do inicio da fila e seus filhos são inseridos no fim da mesma. Em outras palavras, sempre é explorado o nó menos recentemente gerado.

O grande problema da busca em largura é o custo de memória, tendo em vista que é necessário armazenar todos os nodos de um nível, fazendo com que o gasto de memória cresça exponencialmente; para explorar o nível **n**, é necessário espaço para armazenar $\mathbf{b^n}$ estados, o que pode ser impossível. O gasto de tempo é relativo ao da geração e exploração de cada nó, o que deixa o algoritmo muitas vezes impraticável.

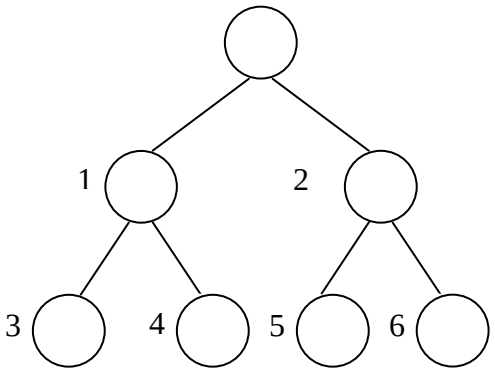


**Figura 3.3 Ordem de Exploração Largura.**

### 3.4.2.3 Busca em Profundidade Iterativa (*Iterative-Deepening Search*)

Esse método mescla as idéias e as qualidades das duas técnicas anteriormente citadas. Ele expande todos os nós até um certo nível a cada execução; dessa forma, economiza-se memória, pois só são armazenados os nodos não explorados e sempre se encontra a melhor solução. Por outro lado, o gasto de tempo é superior aos métodos anteriores, já que, a cada avanço de nível de profundidade, toda a arvore é novamente gerada, podendo tornar sua execução impraticável.

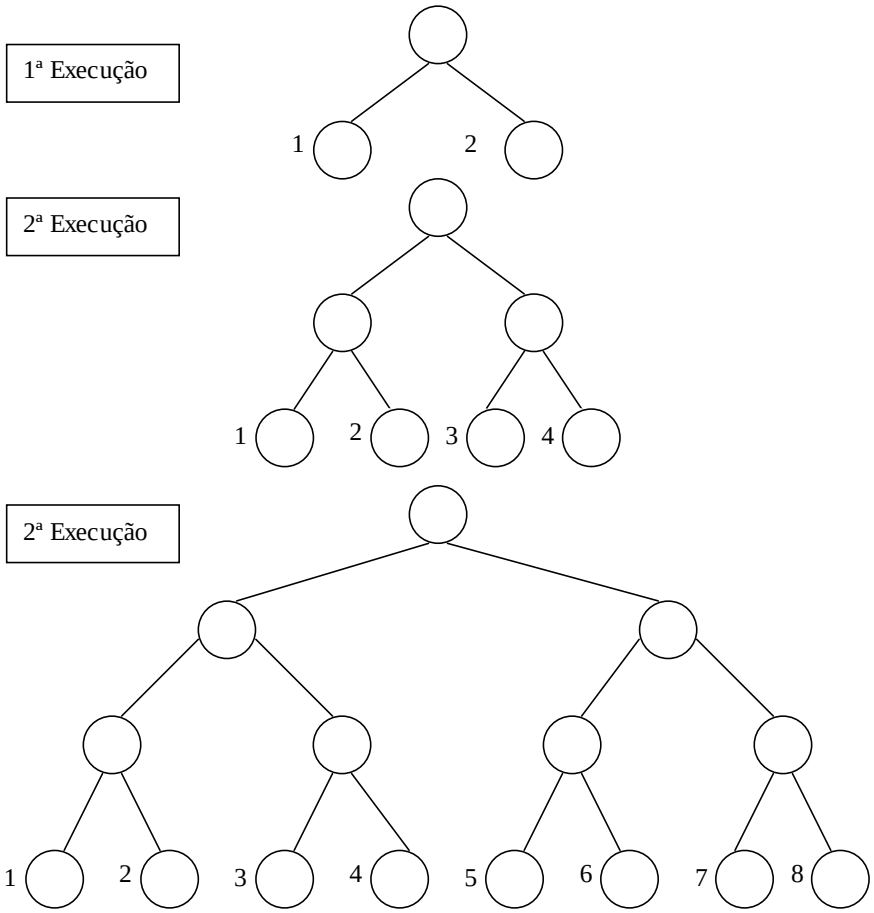


**Figura 3.4 Ordem de Exploração em Profundidade Iterativa.**

### 3.4.2.4 Busca em Bi-Direcional (*Bidirectional Search*)

A concepção da busca bi-direcional é simples, visto que são duas buscas simultâneas em largura, uma partindo do estado inicial e a outra da solução. Ela é mais barata que a busca em largura em termos de memória e tempo (já que $2x\ B^{n/2} < B^n$), mas não se deve esquecer que do gasto de tempo necessário para comparar os resultados já alcançados. Diferentemente da busca em largura, ela pode encontrar soluções não-ótimas, mas sempre serão valores próximos do melhor possível.

A busca bidirecional só pode ser aplicada quando já se conhece a solução do problema, o que impede seu uso em certos casos como, por exemplo, otimizações. Quando existem várias soluções, é possível tomar algumas providências como escolher uma única solução e, a partir dela, fazer várias buscas ou descobrir uma função de derivação em que se consiga ligar todas as soluções umas as outras.

Problemas constantemente executados podem ser otimizados de forma simples na medida em que ao invés de se executar toda vez as duas buscas em largura, uma delas poderia ser armazenada em disco, evitando a segunda busca sempre que o algoritmo for executado.

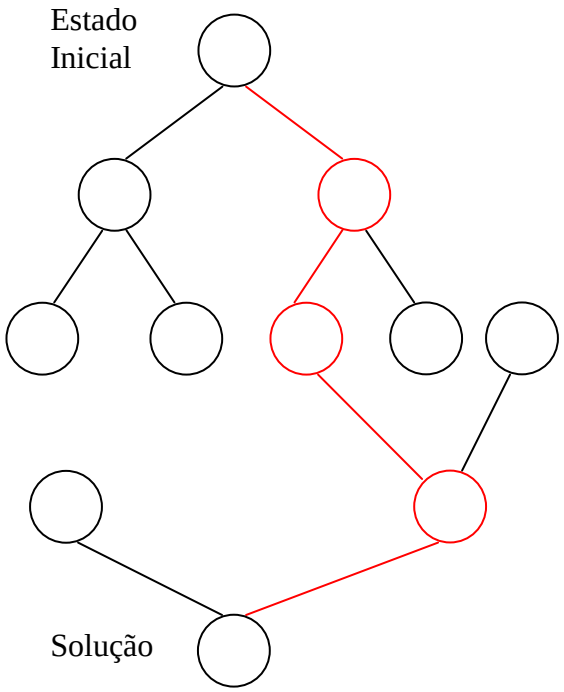


**Figura 3.5 Exploração da Busca Bi-Direcional: o caminho em vermelho é a solução encontrada.**

# Capítulo 4

# Resoluções do Cubo de Rubik

Com a criação do cubo de Rubik, surgiu uma dúvida que, até os dias atuais, aflige os cientistas e, principalmente, os cubistas: qual seria o menor número de movimentos necessários para se chegar de um estado qualquer até solução?

Uma busca em largura resolveria facilmente esse problema, mas o número de estados gerados pela tal busca é proibitivo para os atuais computadores; usando-se uma árvore em que cada cubo ocupe apenas um bit, uma busca em largura com vinte e quatro níveis de profundidade (representando um passo a menos que a melhor solução encontrada até o momento), seriam obtidos 161245862142086000000000000 bits, ou seja, mais de $2x\ 10^{14}$ terabytes.

**Tabela 4.1 Quantidade de nós gerados por uma busca em largura (KORF, 1997)**

| Profundidade | Nós gerados | Profundidade | Nós gerados |
|---|---|---|---|
| 1 | 18 | 13 | 581975750199168 |
| 2 | 243 | 14 | 7768485393179328 |
| 3 | 3240 | 15 | 103697388221736960 |
| 4 | 43254 | 16 | 1384201395738071424 |
| 5 | 577368 | 17 | 18476969736848122368 |
| 6 | 7706988 | 18 | 246639261965462754048 |
| 7 | 102876480 | 19 | 3329630036533747179648 |
| 8 | 1373243544 | 20 | 44950005493205586925248 |
| 9 | 18330699168 | 21 | 606825074158275423490848 |
| 10 | 244686773808 | 22 | 8192138501136718217126448 |
| 11 | 3266193870720 | 23 | 110593869765345695931207048 |
| 12 | 43598688377184 | 24 | 1493017241832166895071295148 |

Desse modo, não é possível simplesmente buscar por melhores soluções forçosamente; deve-se ter um método de busca efetivo, uma vez que é extremamente improvável que movimentos aleatórios alcancem a solução ótima. Ainda que a disponibilidade de artigos, em sua maioria antigos, seja escassa, foi elaborado um histórico das resoluções mais conhecidas que não correspondem à totalidade daqueles existentes.

É possível distinguir os pesquisadores do cubo de Rubik de acordo com o foco de sua pesquisa: os matemáticos e os cientistas da computação. Os primeiros procuram reduzir o número de Deus, utilizando-se principalmente da teoria dos conjuntos, enquanto os outros empregam métodos de busca informada, visando alcançar o algoritmo de Deus.

| **Tabela 4.2 Distinção entre grupos de pesquisadores** | | |
|---|---|---|
| | Matemáticos | Cientistas da Computação |
| Objetivos | Número de Deus | Algoritmo de Deus |
| Base teórica | Teoria dos Conjuntos | Busca Informada |

## 4.1 Primeira Resolução: O algoritmo de quatro fases

Hofstadter (1981) publicou o primeiro artigo científico em que provou um limite para a resolução do cubo de Rubik. O algoritmo foi desenvolvido por Morwen Thistlethwaite e é fortemente baseado em teoria dos grupos e provado por extensas buscas científicas.

O método de busca empregado pelo cientista funcionava com base na divisão da solução em quatro etapas em que cada uma possuía cada vez menos cubos e movimentos possíveis, o que gerava buscas mais ágeis e em conjuntos restritos de Cubos. A seqüência de buscas era dividida nos quatro grupos descritos a seguir:

- G0: O grupo inicial de buscas que contém todos os Cubos possíveis. Para dar prosseguimento, busca-se algum estado do Cubo que pertença ao Grupo 1, aplicando somente operadores de rotação de noventa graus (em sentido horário ou anti-horário).
- G1: O grupo um é constituído por cubos que podem ser resolvidos somente com os movimentos de noventa graus das faces Near, Far, Left e Right, e o giro de cento e oitenta graus das faces Up e Down. É um grupo restrito se comparado ao G0, mas, ainda assim, um grupo grande para a busca da solução; aplicando-se o conjunto de instruções permitidas, busca-se um estado que possa ser enquadrado dentro do Grupo G2, definido abaixo.
- G2: Conjunto de estados em que a resolução se dá somente através da rotação em noventa graus das Faces Left e Right e o giro de cento e oitenta graus das demais faces. A transição entre G1 e G2 é o processo que mais elimina estados, possuindo apenas 1082565 possíveis cubos - fração muito pequena se

comparado aos $4{,}32 \times 10^{19}$ totais do cubo de Rubik. Através dos movimentos possíveis, procura-se por um estado que se encaixe no Grupo G3 e segue-se para a próxima etapa.

- G3: No penúltimo grupo de cubos, só são possíveis movimentos de cento e oitenta graus em qualquer face, diminuindo drasticamente o número de operadores válidos se comparados aos do primeiro grupo. Aplicando-os, encontra-se o Conjunto G4 que possui somente um elemento.
- G4: O grupo final contém somente um elemento, ou seja, a solução do problema.

**Tabela 4.3 Relação entre o grupo e a quantidade de movimentos do mesmo.**

| Grupo de Estados | Quantidade de Movimentos | Possíveis Movimentos |
|---|---|---|
| G0 | 12 | U, U2, U', D, D2, D', L, L2, L', R, R2, R', F, F2, F', B, B2, B' |
| G1 | 10 | U2, D2, L, L2, L', R, R2, R', F, F2, F', B, B2, B' |
| G2 | 8 | U2, D2, L, L2, L', R, R2, R', F2, B2 |
| G3 | 6 | U2, D2, L2, R2, F2, B2 |
| G4 | 0 | Nenhum |

O processo de busca, dividido nessas quatro etapas, é mais rápido e barato que uma busca geral pelo cubo. Thistlethwaite provou que o número mínimo de movimentos possíveis para resolver 'o pior caso' do cubo é de cinqüenta e dois movimentos, sendo necessários sete passos para a transição entre os grupos G0 e G1, treze entre G1 e G2, quinze entre G2 e G3 e, finalmente, dezessete entre G3 e G4.

## 4.2 Segunda Resolução: O algoritmo de duas fases

Baseado na solução encontrada por Thistlethwaite, Kociemba (**2007**) desenvolveu outra metodologia que consistia em separar o algoritmo em duas fases. Esse método foi desenvolvido por ele no início da década de noventa, mas somente a sua última versão está disponível no site. Na primeira fase, eram gerados milhares de maneuvers (nome dado pelos cubistas para macro-operadores) que levariam o estado aleatório dado até um grupo específico de cubos em que somente oito movimentos eram permitidos, a saber: Up e Down girados em

90º (em sentido horário e anti-horário) e as demais faces somente em 180º - similar ao grupo G2 de Thistlethwaite.

A partir desse subgrupo, eram executadas somente as operações permitidas, reduzindo drasticamente o total de nós gerados. Com esse algoritmo, Kociemba conseguiu provar que qualquer cubo era solucionado em no, máximo. trinta passos, sendo que até doze passos eram necessários para encaixar o cubo alvo no subgrupo definido e mais dezoito passos para a solução final ser encontrada.

Reid (1995a) provou que o pior caso nunca aconteceria na segunda etapa do algoritmo de Kociemba, restringindo para dezessete passos o tamanho máximo da segunda etapa e, conseqüentemente, para a marca de vinte e nove passos para solucionar qualquer caso do Cubo.

## 4.3 Terceira Resolução: Próximo do Algoritmo de Deus

A partir da idéia dos *Pattern Databases*, Korf (1997) realizou uma busca inversa e, assim, conseguiu um banco de dados de estados gerados a partir da solução, armazenando os movimentos necessários para conseguir chegar até o estado-alvo. De posse desse conjunto de dados, ele gerou um pequeno número de exemplos de cubo mágico e, através de uma busca em profundidade (A*), conseguiu resolver todos em, no máximo, 18 movimentos; acreditando ser esse um número próximo ao Número de Deus.

O uso de um algoritmo bidirecional (caso que generaliza o método utilizado por Korf) diminui drasticamente o número de estados gerados em uma busca em largura (de $10^{27}$ para 'apenas' $10^{13}$ estados). No entanto, seriam necessárias inúmeras comparações para saber se o estado gerado é novo, se já foi fechado, se está mapeado na busca inversa entre outros casos, impossibilitando a computação do problema em tempo hábil. Além disso, ainda seriam necessários cerca de 2 terabytes de armazenamento.

O banco de dados de Korf também serviu de base para o cálculo da heurística utilizada; através das posições dos 'Corners Cubies', o algoritmo calculava a distância entre aquele estado e o estado final, dando preferência para o desenvolvimento de estados mais próximos da solução.

A técnica usada também sofreu uma limitação tecnológica, tendo em vista que as tabelas de seu Banco de Dados passavam dos 82 megabytes, gerando inúmeros acessos da memória cachê na memória principal, o que impedia a criação de tabelas ainda maiores, que tornariam o algoritmo mais lento. Essa restrição também limitou o cálculo do valor heurístico de cada estado, pois, ao invés de usar todo o Cubo, foram usados somente os Corners Cubies.

O algoritmo gerado por Korf (1997) é até hoje aceito como a melhor solução para qualquer estado do cubo, o seu único fator limitativo é o tempo, porque em uma máquina munida de um processador Core 2 Quad Q6600 (com quatro núcleos de processamento), a resolução de cada estado leva, em média, quinze minutos. Para solucionar mil cubos, seriam necessárias, em média, 250 horas, inviabilizando a resolução em tempo hábil de todos os $4{,}32x10^{19}$ estados para que se possa, finalmente, encontrar o menor número de passos. Diante dessa impossibilidade, novos algoritmos vêm sendo lançados, aproximando-se cada vez mais do número de Deus.

## 4.4 Quarta Resolução: Vinte e Sete movimentos

Publicado após a detecção de um pequeno problema no trabalho de Kunkle e Cooperman (2007), mas antes da solução do mesmo, Radu (2007) conseguiu resolver qualquer instância do cubo em até 27 movimentos. Inicialmente provando ser capaz de resolver o cubo em vinte e oito passos, o cientista se utilizou do algoritmo de Kociemba; tomando o pior caso na primeira fase do algoritmo (equivalente a doze movimentos), teria-se que reduzir para dezesseis o número máximo de passos da segunda etapa. Considerando somente as instâncias que precisam de dezessete passos e descartando as demais, Radu conseguiu chegar à marca desejada através de teoremas presentes na teoria dos grupos. Aplicando o software de Kociemba(2007) em vários exemplos, o cientista mostrou que a marca dos vinte e sete passos é tangível.

No entanto, um refinamento é apontado no próprio artigo, declarando que vinte e sete seria a marca mais próxima do número de Deus. Reorganizando o conjunto intermediário de movimentos possíveis de modo dinâmico e executando sempre o pior caso desse grupo, independente do resultado esperado, o cientista alcançou a nova marca de vinte e sete passos que, infelizmente, foi quebrada dias depois da publicação de seu trabalho com a correção do

artigo de Kunkle e Cooperman (2007). A técnica elaborada por Radu (2007) foi deixada de lado por apresentar resultados que podem nunca vir a acontecer como foi provado de maneira similar por Reid (1995a) em relação ao algoritmo de Kociemba (2007).

## 4.5 Quinta Resolução: Vinte e Seis movimentos

Utilizando-se de novas e mais velozes formas de multiplicação de elementos em grupo, Kunkle e Cooperman (2007) conseguiram reduzir para vinte e seis o menor número de passos suficientes para se resolver qualquer instância do cubo mágico. Tomando como base o trabalho de Kociemba (2007), os cientistas modificaram partes do algoritmo original baseados em teoria dos grupos, gerando um subgrupo H = <U2,L2,R2,F2,B2> alcançável com apenas quinze mil setecentos e cinqüenta e dois possíveis estados.

A primeira fase tem início com a construção de um grafo de Cayley gerado através de uma busca bidirecional dada da seguinte maneira: uma busca em largura, de profundidade sete, é gerada para todas as instâncias de H. Em seguida, gera-se uma busca em largura a partir da solução até que se encontre um estado comum às duas buscas. No pior caso, são necessárias 13 operações para transformar uma instância qualquer pertencente a H na solução do Cubo.

Por ser tratar de uma tarefa exaustiva e repetitiva, os cientistas construíram uma pequena base de dados para armazenar a primeira fase; desse modo, ao efetuar a busca por um exemplo pertencente ao subgrupo H, já será possível saber a distância do elemento à solução, diminuindo drasticamente a computação necessária.

A segunda fase é mais complexa, já que se busca gerar um elemento pertencente a H a partir de um exemplo aleatório do Cubo. Armazenadas em formato hash de 4 bits, todas as instâncias significativas (somente as que são sabidamente mais caras, não sendo resolvidas em menos de 18 passos) ocuparam mais de 685 GB – valor que extrapola a memória principal das máquinas modernas. Os cientistas buscaram então o processamento paralelo para resolver esse problema no qual a memória secundária poderia armazenar esse valor e funcionaria de forma similar à memória principal em um supercomputador.

O avanço da segunda fase também se deu por busca bidirecional, mas de um modo diferente dessa vez; pois cada nível era gerado separadamente e comparado ao mais profundo da outra fonte, em busca de elementos comuns. O processo ‘de volta’ – busca em largura a

partir dos elementos pertencentes a H - foi executado somente uma vez e armazenado assim como os grafos resultantes da primeira etapa a fim de obter economia de processamento. O pior caso encontrou dezesseis passos, totalizando vinte e nove, marca que Reid (1997a) já havia alcançado.

Os cientistas, baseados em princípios da teoria dos grupos e na simetria do cubo, resolveram percorrer todos os caminhos de cada grafo gerado na segunda fase com a finalidade de provar que atalhos poderiam ser encontrados na forma de simplificações. Eles foram bem sucedidos em sua execução, porque conseguiram diminuir em três o número máximo de passos dados na segunda etapa, realizando um total de vinte e seis passos, a nova marca do Cubo de Rubik.

Um pequeno problema na resolução foi identificado por Drupal (2007a) e está relacionado ao fato de que ao tentar provar que qualquer cubo que esteja a uma distância menor ou igual a três pode ser resolvido em até quatorze movimentos, os cientistas usaram exemplos reduzidos por simetria, infringindo leis da teoria dos grupos. Em resposta a Kociemba, Drupal (2007b) corrigiu o erro e executou novamente o algoritmo, encontrando resultados que seriam um pouco piores; assim, alguns casos precisavam de quinze movimentos e não quatorze conforme anunciado previamente. Por outro lado, esse erro não afetou o resto da solução, permitindo que a marca permanecesse em vinte e seis movimentos com número máximo para solucionar qualquer caso do cubo de Rubik.

## 4.6 Sexta Resolução: Vinte e Cinco movimentos

Reunindo novas idéias, hardware mais poderoso e uma otimização de conceitos já conhecidos, Rokicki (2008) foi capaz de resolver qualquer estado do cubo de Rubik em até 25 movimentos.

O cientista dividiu o espaço de busca em dois bilhões de conjuntos, contendo cerca de vinte bilhões de estados cada um, percorrendo todo o conjunto possível de instâncias. De forma similar ao algoritmo de Kociemba, Rokicki dividiu seu algoritmo em duas fases, sendo que a primeira consiste na geração de cubos pertencente a um subgrupo e, a partir desse subgrupo, a solução é encontrada.

O subgrupo H = <U, U2, U', D, D2, D', R2, L2, F2, B2> é diferente do subgrupo original de Kociemba, visto que permite dois movimentos a mais que o original, ampliando o poder de representação e, portanto, o número de elementos do grupo. A partir das arestas – instâncias pertencentes ao grupo mais distantes da solução –, o cientista criou seus dois bilhões de conjuntos. Assim, fazendo uma espécie de busca em largura a partir de cada cubo da aresta, Rokicki conseguiu agrupar elementos com características e distâncias similares. Dessa forma, ao conseguir solucionar um elemento do grupo, todos os outros elementos podem ser considerados solucionados, simplificando consideravelmente a busca por soluções.

O subgrupo H possui cerca de mais de dezenove milhões de estados, incluindo a solução. Todos os elementos são caracterizados da seguinte forma: todos os edge e corners estão orientados corretamente, ou seja, não há inversão de orientação; e todos os edge cubies estão em sua área de origem, isto é, cubies que contém uma face Up ou uma face Down estão em Up ou Down (não necessariamente na face correta), e todos os outros (*middle edge*) estão ocupando as arestas do meio: F/L , F/R, B/L e B/R. Essas características são preservadas pelos movimentos possíveis que limitam o subgrupo H.

Após a simplificação dos elementos por meio da simetria, todo o subgrupo H foi armazenado em uma tabela hash de dois bits, ocupando cerca de trezentos megabytes – valor pequeno o suficiente para ser mantido na memória principal da máquina utilizada por Rokicki para a solução do problema. Ao invés de armazenar a distância do elemento à solução, o cientista guardou a posição da instância no subgrupo e, a partir dessa posição, foi capaz de gerar os grupos, conforme anteriormente descrito.

O grafo formado por todos os conjuntos gerados mostra que cada vértice dá origem a pelo menos um subconjunto de vinte milhões de elementos; ao resolver todos os subconjuntos gerados pelo vértice, pode-se 'apagá-lo' de H, simplificando ainda mais o processo de busca. O processo de escolha do próximo vértice é determinado através de um método míope: o vértice mais complexo (que gera mais subconjuntos) e próximo é escolhido para ser percorrido.

Preocupado com a corretude de sua solução, o pesquisador usou três estruturas paralelas de resolução que são o algoritmo solucionador e dois auxiliares. O primeiro auxiliar comparava e testava se a solução encontrada era realmente válida, ou seja, se, através do cubo inicial dado e aplicando os movimentos sugeridos, a solução era realmente obtida. O segundo

auxiliar era um simples ‘banco de soluções’ em que cada solução encontrada era adicionada ao ‘banco’, aliviando o programa principal. Por fim, cada otimização (incluindo as simplificações por simetrias) foi ignorada em momentos posteriores de execução para garantir a funcionalidade do programa; nesses casos, não eram executadas todas as instâncias; na verdade, eram sorteados e executados três milhões de elementos diferentes, todos alcançando resultados similares ao pretendido. Finalmente, o cientista aplicou todos os resultados mais relevantes no Cube Explorer, simulador gratuito de Kociemba (2007), alcançando a solução com o maneuver adquirido pela solução em todos os casos.

Executada em uma máquina equipada com oito gigabytes de memória e um processador Core 2 Quad modelo Q6600 (quatro núcleos de processamento de 1,6 GHz), cada solução leva cerca de dezoito minutos para ser alcançada contra uma média de quinze do algoritmo de Korf. Por outro lado, a resolução de um subgrupo de dezoito movimentos (como o subgrupo formado por <R’, B2, U, B2, U’,R>) com vinte milhões de estados é obtida em, aproximadamente, 5 horas de execução ao passo que, se o algoritmo de Korf fosse utilizado, seriam necessárias cerca de cinco milhões de horas para executar todos os estados e garantir sua marca local de dezoito movimentos.

## 4.7 Solução Humana

Existem receitas para a solução ‘humana’ do problema, mas todas elas são dispendiosas caso se considere o número de movimentos. A mais difundida é dividida em 2 etapas, contando ainda com quatro ou cinco maneuvers. Com o advento da Internet, novas técnicas vêm sendo lançadas periodicamente, mas nenhuma envolve um menor número de movimentos médios que aquela que será descrita.

Inicialmente, deve-se solucionar os Edge Cubies de cada face. Normalmente, uma face é eleita para se iniciar as soluções dos Edge Cubies e segue-se resolvendo todos eles, de ‘cima’ para ‘baixo’. Para isso, são usados dois macro-operadores bem comuns aos cubistas, o trocador de bordas e o trocador de bordas com inversão.

**Figura 4.1 Primeira Etapa da Solução Humana do Cubo: inicialmente são resolvidos todos os *Edge Cubies* da face superior, então os *Cubies* do 'meio' são solucionados e, finalmente, fecham-se os *Edge Cubies* inferiores.**

As seqüências trocadoras de bordas permutam dois dos Edge Cubies 'vizinhos'de lugar, mas também implicam no movimento de dois Corners Cubies os quais serão resolvidos no final do processo. Para a troca de Edge não vizinhos, os métodos são usados de duas a quatro vezes.

**Figura 4.2 Troca de Bordas Sem Inversão.**

**Figura 4.3 Troca de Bordas Com Inversão.**

Após o correto posicionamento de todos os Edge Cubies, precisa-se solucionar os Corners Cubies. Para isso, o maneuver trocador de cantos é usado sem a preocupação com a orientação correta; esse será o último passo. Esse macro-operador resulta na troca de cantos opostos; para troca de cantos vizinhos, simplesmente se deve girar a face que possui os dois cantos e depois corrigi-se o edge cubie ‘modificado’ para sua posição inicial.

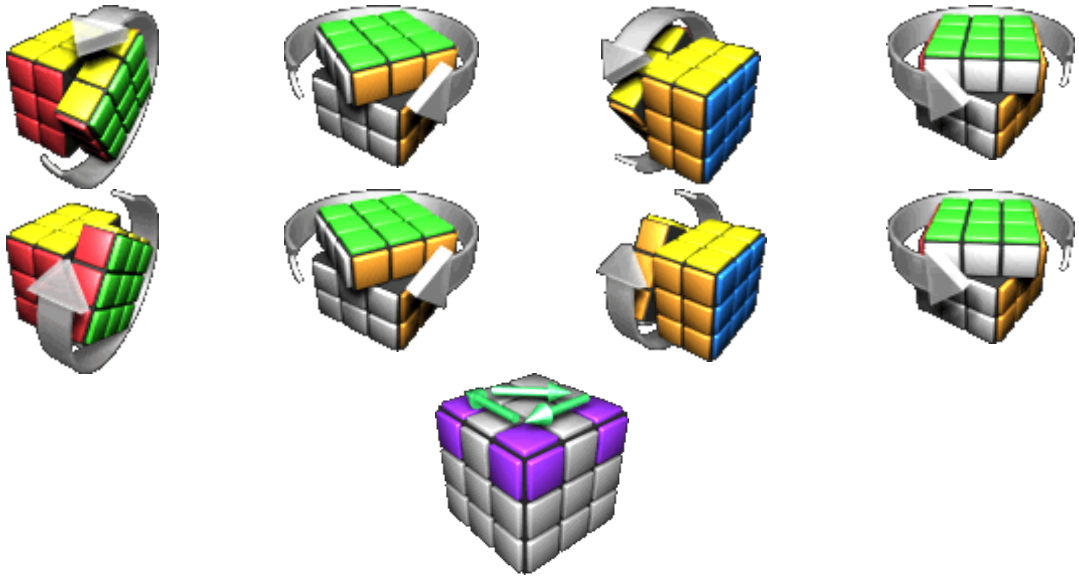

**Figura 4.4 Troca de Cantos.**

Após todos os Corners estarem em seu lugar certo, utiliza-se dos maneuvers giradores de canto para a direita ou para a esquerda. Note que executar um desses macro-operadores duas vezes seguidas equivale à execução da outra seqüência. Devido às características do Cubo, cada conjunto de correção de orientação opera sobre dois Corners vizinhos; desse modo, ele pode ‘atrapalhar’ um Canto já arrumado; porém, no final, dois estados serão arrumados com uma só seqüência de movimentos.

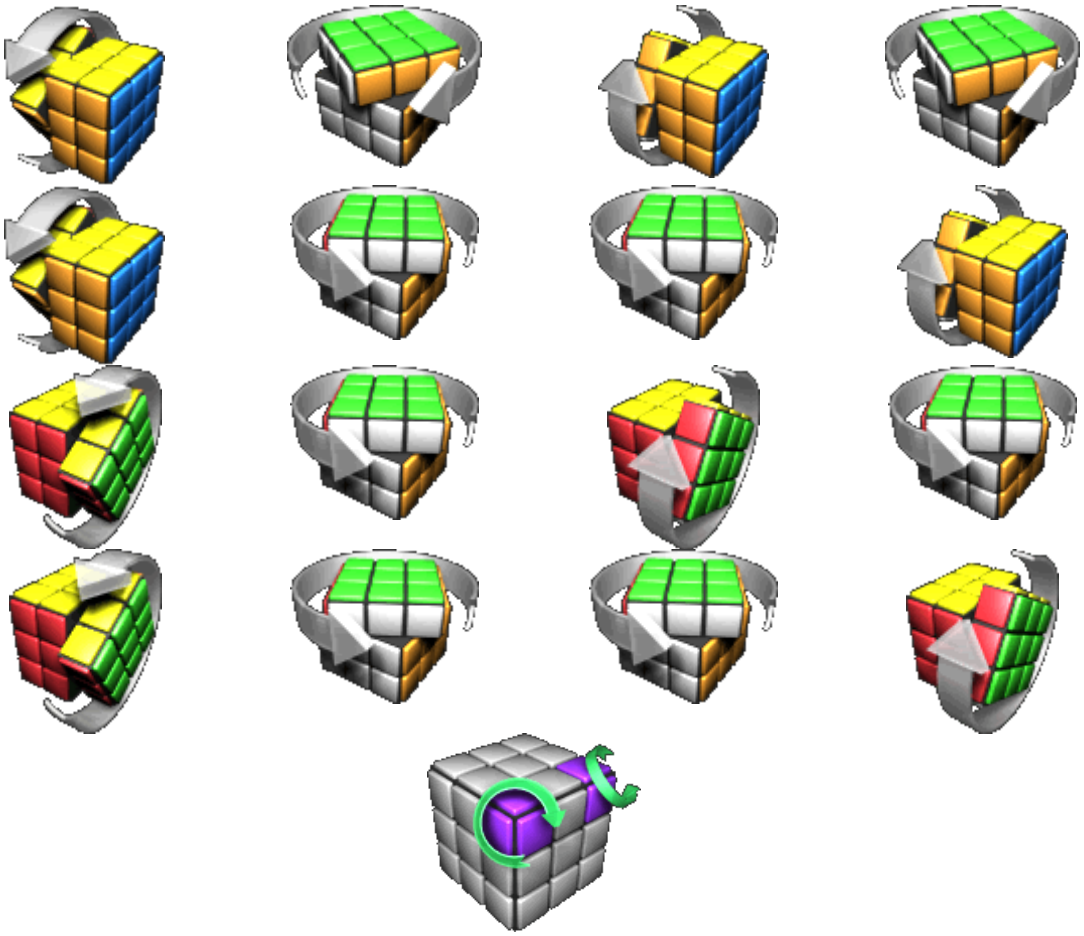

**Figura 4.5 Giro de Canto no Sentido Horário.**

**Figura 4.6 Giro de Canto no Sentido Anti-Horário.**

Dessa forma, observa-se a complexidade das ‘soluções humanas’ em relação aos vinte e cinco movimentos obtidos por Rokicki (2008). Embora sejam extremamente mais rápidas, as técnicas humanas demandam maior quantidade de movimentos. Um caso simples, como a rotação de uma única face, resulta na utilização de até três maneuvers, gerando um gasto de dezenas de movimentos contra somente um movimento computado por uma busca heurística.

Por outro lado, casos mais complexos, como faces aleatoriamente giradas por diversas vezes, exigem um enorme tempo bem como recursos computacionais na geração e análise de bilhões de estados diferentes, enquanto que, com o uso dos ‘métodos humanos’, a solução seria rápida, ou seja, algumas centenas de movimentos, sem a necessidade de se guardar qualquer estado que não seja o atual e a solução.

# Capítulo 5
# Heurísticas

O termo heurística vem do grego *heuriskein* cujo significado é descobrir, encontrar. A célebre frase atribuída a Arquimedes quando da descoberta do principio do empuxo (Eureka) seria grafada como Heureka, mas o ‘h’ acabou sendo abolido com o tempo. O termo se tornou popular a partir do livro *How to Solve It* do matemático George Pólya; o volume reúne varias técnicas de auxilio na solução de problemas, ou seja, uma coleção de heurísticas.

Destarte, pode-se chamar de heurística uma forma de auxiliar a resolução de um problema. Em termos computacionais, função heurística é o nome dado a uma expressão que calcula o custo aproximado da geração de um estado específico em relação ao estado atual. Em outras palavras, pode-se dizer que as funções heurísticas tendem a classificar diferentes estados, julgando quais são melhores que os outros.

Os primeiros sistemas especialistas usavam a heurística como base, não como ferramenta, o que deixava os programas à mercê das funções heurísticas que, normalmente, não são inteiramente precisas. Elas eram vistas como solução última, uma vez que apresentavam resultados razoáveis em muitos aspectos, inibindo a necessidade de buscas exaustivas que, em geral, eram caras e demoradas. Com o tempo, os programadores perceberam que era um erro tratar a heurística como base do problema, já que esse comportamento acabava por limitar as buscas. Ela começou a ser usada como instrumento cujo emprego, muitas vezes, pode ser regulado pelo usuário o qual pode aumentar, diminuir ou inibir o uso da função.

## 5.1 Criação de uma heurística

As funções heurísticas costumam estar atreladas ao objetivo do problema e medem quanto o estado atual está próximo dele. Assim, não existe uma fórmula exata de criação de heurísticas, embora alguns métodos sejam muito práticos e conhecidos.

George Pólya, em sua obra *How to Solve It* fornece várias dicas de como começar a resolver o problema, com sugestões para a criação de heurísticas como, por exemplo, fazer um

esquema de um problema com o objetivo de facilitar sua compreensão. Quando não se consegue resolver uma determinada questão, deve-se buscar solucionar problemas próximos ou similares. Se a solução é amplamente conhecida, mas ainda não foi possível encontrá-la, comece o problema a partir dela; esse método, inclusive, é o mesmo utilizado na busca bidirecional.

Outro método extremamente utilizado é o relaxamento do problema. Relaxar um problema (ou suas regras) equivale a diminuir ou até mesmo retirar algumas restrições que o problema encontra, de modo a simplificar o problema. O jogo dos oito, por exemplo, só possui uma regra de acordo com a qual pode-se mover a peça A para B se B for vazio (espaço) e eles são vizinhos. Relaxando o problema, são encontradas pelo menos três possibilidades:

1. *É possível mover a peça A para B se B for vazio*. Esse relaxamento retira a condição de 'vizinhança' do problema, permitindo que A migre para B em qualquer posição do tabuleiro que B esteja.
2. *É possível mover a peça A para B se eles são vizinhos*. Isso permite que duas peças troquem de lugar desde que sejam vizinhas, não existindo a obrigatoriedade de que uma delas seja o espaço vazio.
3. *Pode-se também mover a peça A para B*. Mais radical que os anteriores, nessa forma, as peças podem trocar de lugar como bem entenderem, não sendo limitadas por sua vizinhança ou por terem que ocupar o lugar vazio.

**Figura 5.1 Relaxamentos Típicos Obtidos a partir das Regras do jogos dos Oito**

Programas como o Absolver de Prieditis (1993) podem gerar heurísticas automaticamente através do relaxamento do problema. Esse software gerou a primeira heurística não-trivial para o cubo de Rubik. Infelizmente, apesar dessa heurística demandar a existência de um estado simples, este precisa de um a três movimentos para ser formado (uma face precisa ter todos os seus corners cubies no local correto) e, a partir dessa instância, é possível solucionar qualquer problema em até dezoito movimentos. Essa marca é ótima, mas em todos os exemplos resolvidos pelo Absolver foram encontrados outras soluções com menos de 20 movimentos, mostrando que, apesar de sua qualidade, a heurística não é admissível.

## 5.2 Características

A aplicação das heurísticas está sujeita a alguns de seus atributos. O primeiro e, comumente, mais importante é a admissibilidade, o que significa que uma heurística é dita admissível se o seu valor nunca é maior que o custo para se chegar até a solução pelo caminho ótimo; de outra forma, jamais seria encontrada.

A consistência, por sua vez, prega que nenhum caminho pode ter custo negativo, ou seja, o custo de um filho sempre deve ser maior ou igual ao de seu pai em relação ao estado inicial. O valor heurístico do filho pode ser maior, menor ou diferente do pai, já que ele pode estar dando 'um passo para trás'; contudo, seu custo de geração será sempre maior. Uma heurística que é consistente é automaticamente admissível.

Outra característica interessante sobre as heurísticas está ligada a sua especificidade. De fato, esta peculiaridade não necessariamente se associa à melhoria em sua qualidade, tendo em vista que elas apresentam ótimos resultados na modalidade em que se especializaram em detrimento de resultados ínfimos ou não existentes em problemas similares.

Por fim, a dominância também é qualidade que merece ser destacada. Dize-se que uma heurística domina as demais quando a qualidade apresentada pela mesma é superior às outras. Um ótimo exemplo de heurística dominante é a distância de Manhattan que é a melhor heurística 'humana' criada para resolver o problema.

## 5.3 Qualidade da Heurística

Medir a eficácia de uma heurística não é simples, já que elas costumam ser específicas demais. Algumas, como a famosa 'distância de Manhattan', são consideradas muito boas por serem admissíveis, consistentes e, sobremodo, uma boa medida. Essa heurística, muito usada em jogos como o jogo dos oito, propõe que se some a distância de todas as peças de sua posição atual até a solução. No caso específico dos jogos de 'peças deslizantes', como o jogo dos 8 e dos 15, só há movimentação nas direções horizontal e vertical similarmente ao do percurso ao longo dos quarteirões de uma grande cidade, daí o nome de Manhattan.

Embora medir um conceito abstrato possa parecer estranho, a forma mais utilizada de medição de heurística é feita através do Fator de Ramificação Efetivo **b*** (*Effective Branching*

*Factor*). Considerando **n** o número total de nodos gerados até encontrarmos a solução e **d** a profundidade da solução, encontra-se **b*** utilizando a seguinte fórmula:

$$\mathbf{n} = 1 + \mathbf{b^*} + (\mathbf{b^*})^2 + (\mathbf{b^*})^3 + \ldots + (\mathbf{b^*})^{\mathbf{d}}$$

**Figura 5.2 Formula para Calculo de Fator de Ramificação Efetivo.**

Quanto menor o valor de **b***, "melhor" será a heurística. Esse fator varia a cada solução, pois depende do estado inicial, número de nós gerados, profundidade da solução encontrada entre outros fatores. Para contornar a mudança de magnitude do fator, costuma-se executar o programa várias vezes, com diferentes estados, em busca de um valor médio para **b***.

## 5.4 A*

O uso de heurística foi uma das primeiras formas de se acrescentar conhecimento a uma busca por resultados. Um dos primeiros e mais simples algoritmos a usar a heurística foram os míopes. Eles simplesmente escolhem o melhor valor heurístico local, expandindo-o até encontrar a solução ou entrarem em *loop* ou se mostrarem ineficazes em termos de custo computacional. Por outro lado, muitos algoritmos míopes são usados por serem simples, rápidos e, geralmente, apresentarem um resultado aceitável. Assim, outros mecanismos de busca utilizando heurísticas precisavam ser descobertos, levando à descoberta do A*. Pode-se resumir o Algoritmo A* em uma busca ordenada, onde os nós que tenham a menor soma do custo estimado (H) com o custo adquirido (G) são primeiramente explorados.

Se a heurística empregada for consistente, o algoritmo sempre encontra a melhor solução, conforme foi demonstrado em Russel e Norvig (1995). Como o A* expande os nós por ordem de menor incremento de F, eventualmente a saída ótima é expandida. Esse comportamento só não é alcançado se um ou mais estados tiverem fator de ramificação infinita, o que é impossível.

Como toda função ótima, o custo em tempo do algoritmo A* cresce de forma exponencial, embora em uma razão menor em relação aos outros, como a busca em largura. O problema é que essa redução é proporcional à qualidade da heurística. O tempo de computação em si não é o maior problema do A* por ser obrigado a guardar todos os estados gerados, mesmo aqueles que não são explorados. A grande maioria das funções heurísticas carrega um

pequeno erro que tende a gerar estados fora do caminho ótimo; quanto melhor for a heurística, menos erros são gerados e mais rapidamente eles são corrigidos produzindo-se, assim, menos estados. Esses erros, se gerados em excesso, tendem a ocupar muito espaço em memória, podendo sobrepujar os recursos disponíveis.

## 5.5 Primeira Solução Proposta

Na primeira solução, partiu-se de um algoritmo conservador com uma base de estudos sólida. Desse modo, aplicou-se o A* na tentativa de solucionar o Cubo de Rubik. O primeiro passo para a sua aplicação seria a decisão pela heurística a ser usada.

Pesquisas apontam que a heurística mais utilizada pelos cientistas é aquela que soma quantos cubos estão em seu lugar, revelando-se uma heurística fraca, pois não sabe decidir entre dois estados diferentes que possuem três peças fora do lugar, sendo uma delas uma fileira inteira e a outra 3 *cubies* espalhados pelo objeto. Dessa forma, decidiu-se pela utilização de uma heurística diferente em conformidade com a idéia da distância de Manhattan, ótima para o estudo do jogo dos 8. A nova função apresenta mais valores que as anteriores: caso um *cubie* esteja em sua posição correta, seu valor é zero; se estiver a um movimento da solução, seu custo equivale a um; qualquer outra posição pode ser resolvida com dois passos. Anteriormente eram avaliados somente se o um *corner* ou *edge* estavam em seu lugar corretos, apresentando somente os valores zero e um.

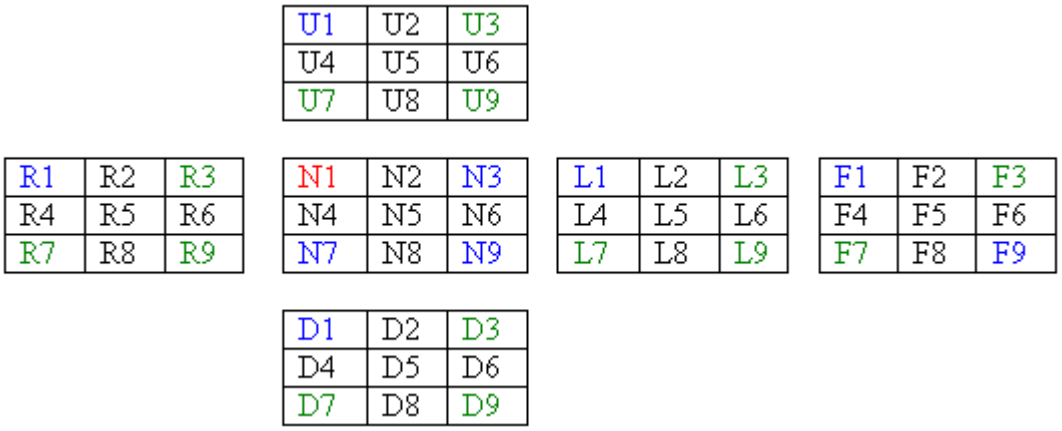


**Figura 5.3 Distância de Manhattan Aplicada ao Cubo de Rubik. No exemplo acima, caso o nó N1 esteja ocupando a posição vermelha, seu custo é zero; caso esteja nas posições azuis, o custo é um e, nas posições verdes, seu custo é dois. Se ele ocupar alguma outra posição, o problema apresenta um erro, pois a peça é um corner cubie e não pode freqüentar a posição de um edge cubie.**

Aplicando a idéia apresentada acima, obtém-se o seguinte:

```
1231  public int calculaHeuristica(){
1232      int heuristica = 0;
1233      String alvo = "";
1234
1235      alvo = this.getF(1);
1236      if (alvo.equals("F1")){
1237          heuristica = heuristica + 0;
1238      }else{
1239          if ((alvo.equals("F3"))|
1240             (alvo.equals("F7"))|(alvo.equals("F9"))|
1241             (alvo.equals("U1"))|(alvo.equals("L1"))|
1242             (alvo.equals("D1"))|(alvo.equals("R1"))|
1243             (alvo.equals("B1"))|(alvo.equals("B9"))){
1244              heuristica = heuristica + 1;
1245          }else{
1246              heuristica = heuristica + 2;
1247              System.out.print(alvo);
1248          }
1249      }
```

**Figura 5.4 Calculo da heurística de acordo com a distância de Manhattan.**

O cubo utilizado no trabalho é formado por seis faces, cada uma representada por uma cadeia de caracteres (string). Dessa forma, é possível verificar qualquer posição e definir precisamente onde está cada peça que compõe o cubo, facilitando o calculo da função

heurística. Caso o cubo fosse representado como um vetor de inteiros, a idéia inicial, cada estado ocuparia o equivalente a 55 inteiros, o que é muito maior que espaço requerido por seis *strings*. O método construtor da classe **Cubo** será mostrado a seguir:

```
20  public Cubo() {
21      F = "F1F2F3F4F5F6F7F8F9";
22      B = "B1B2B3B4B5B6B7B8B9";
23      L = "L1L2L3L4L5L6L7L8L9";
24      R = "R1R2R3R4R5R6R7R8R9";
25      D = "D1D2D3D4D5D6D7D8D9";
26      U = "U1U2U3U4U5U6U7U8U9";
27      h = this.calculaHeuristica();
28      f = this.calculaHeuristica();
29      g = 0;
30  }//construtor
```

**Figura 5.5 Método Construtor da Classe Cubo.**

A classe **Estrela** contém o método **realizaBusca**, que é a implementação do algoritmo em si.

```
23  void realizaBusca(Cubo c){
24      Cubo alvo;
25      int saida = 0;
26      aberto[0] = c;
27      nAberto = 0;
28
29      while((nAberto > -1) | (saida != 0)){
30          alvo = selecionaNo();
31          geraSucessores(alvo);
32      }//while
33  }//Estrela
```

**Figura 5.6 Implementação do A***

O primeiro passo a ser dado é inserir o estado inicial ao vetor de abertos, atualizando o número de elementos dentro do conjunto. Em seguida, executa-se uma seqüência de instruções enquanto a estrutura de abertos possuir elementos ou a saída for encontrada. A estrutura de abertos foi implementada através de uma arvore hierárquica, que nada mais é que uma arvore binária que segue algumas regras específicas, onde cada nodo pai tem seu custo menor ou igual ao de seus filhos. Essa estrutura permite uma grande redução em termos de tempo de

acesso aos melhores nós – que estão sempre na raiz da árvore – garantindo uma boa performance do programa em relação ao tempo.

O início do laço de comandos é a escolha do melhor estado presente na estrutura de abertos. Em seguida, geram-se os sucessores do nó escolhido para, então, transferir-se o nó da lista de abertos para a de fechados. Por fim, percorre-se a lista de sucessores (que contém sempre 18 elementos) para saber se algum deles é a solução – que, se encontrada, modifica o valor de saída – ou se eles pertencem ao estado de fechado. Caso contrário, eles são acrescentados à fila de abertos.

A estrutura de seleção do melhor nodo é simples, uma vez que o melhor nodo sempre está presente na raiz da árvore. Após a retirada do melhor nó da árvore o último nodo é colocado na raiz, o tamanho do vetor de abertos é diminuído e rebaixado.

```
52  Cubo selecionaNo(){
53      Cubo retorno = new Cubo(aberto[0]);
54      nAberto--;
55      aberto[0]=aberto[nAberto+1];
56      rebaixa(0);
57      return retorno;
58  }//primeiro
```

**Figura 5.7 Seleção do primeiro nodo.**

O método **rebaixa** é responsável por organizar a árvore, de modo a obedecer as diretrizes da árvore hierárquica, ou seja, manter sempre os pais com custo menor ou igual ao dos filhos.

```
61  void rebaixa(int i){
62      int x,j,n;
63      boolean fim = false;
64      n = nAberto;
65      x = aberto[i].f;
66      j = 2*i;
67      while((j<=n) && (!fim)){
68          if((j<n)&(aberto[j].f > aberto[j+1].f)){
69              j++;
70          }
71          if(x <= aberto[j].f){
72              fim = true;
73          }else{
74              aberto[j/2] = aberto[j];
75              j = j*2;
76          }
77      }
78      aberto[j/2] = aberto[i];
79  }//rebaixa
```

**Figura 5.8 Rebaixamento;**

A próxima etapa a se explorar é a geração de sucessores. Depois de efetuada a escolha do melhor nodo, deve-se explorá-lo. Dessa forma, geram-se todos os 18 sucessores, salientando-se que mesmo os estados fechados são produzidos; por outro lado, a verificação de valores, mais tarde, deixará claro quem já pertence à lista de fechados, não permitindo que esses estados sejam inseridos na lista de abertos.

```
93  void geraSucessores(Cubo alvo){
94
95          sucessores[0] = new Cubo(alvo);
96          sucessores[0].FH();
97          sucessores[0].g = sucessores[0].g + 2;
98          sucessores[0].h = sucessores[0].calculaHeuristica();
99          sucessores[0].f = sucessores[0].h + sucessores[0].g;
100         if(sucessores[0].eSolucao()){
101             System.out.println("Solucao encontrada!");
102             System.out.println("Foram abertos "+nAberto+" nodos");
103             System.exit(0);
104         }
105         if(livre(sucessores[0])){
106             insere(sucessores[0]);
107         }
```

**Figura 5.9 Geração de um Sucessor;**

Antes de inserir  um estado recém criado à estrutura de abertos, devemos verificar se o estado não pertence a estrutura de abertos; se for o caso, ele não deve ser adicionado novamente.

```
83  boolean livre(Cubo c){
84      for(int i = 0; i<=nAberto; i++){
85          if(aberto[i].eIgual(c)){
86              return false;
87          }
88      }
89      return true;
90  }
```

**Figura 5.10 Função livre: verifica se um nó pode ou não ser inserido da arvore de abertos.**

### 5.5.1 Resultados

A maior dificuldade apresentada pelo algoritmo foi a limitação de memória presente na linguagem Java, já que, por se tratar de uma máquina virtual, ela possui um limite nativo de cento e vinte e oito mega de espaço em memória, restringindo consideravelmente a busca. Em termos práticos, o algoritmo fica restrito a cento e cinco mil estados, um número pequeno se comparado aos $10^{19}$ possíveis. Esse obstáculo foi superado com a inserção dos parâmetros "***-Xms -Xmx***" em que o primeiro representa o tamanho mínimo ocupado em memória e o segundo, o máximo. Ao ser executada, a máquina virtual reserva inicialmente o parâmetro definido por "***–Xms***" em memória. De acordo com a necessidade, mais memória é alocada até o valor definido por "***–Xmx***".

Dessa forma, deve-se inserir os parâmetros "***-Xms32m -Xmx512m***" na maquina virtual para o correto funcionamento do algoritmo. Isso pode ser feito de duas formas distintas: ou as variáveis são inseridas no campo de execução da IDE utilizada ou, simplesmente, através de linha de comando no prompt.

Após a correta inserção dos referidos parâmetros, a máquina virtual irá reservar, caso haja disponibilidade, até quinhentos e doze mega de memória RAM, expandindo consideravelmente o espaço do problema.

Mesmo com a ampliação de memória proveniente da inserção dos parâmetros anteriormente mencionados, o espaço de busca continua limitado. À medida que o algoritmo

continua a busca pela solução, mais memória é requerida. Caso se leve em conta o espaço total do problema, o limite de quinhentos e doze mega torna-se pequeno.

O armazenamento de abertos através de uma lista encadeada poderia tornar o tempo outro grande fator limitativo do A*. Dessa forma, a estrutura de abertos foi implementada através de uma árvore hierárquica, diminuindo o tempo de computação necessário.

A grande vantagem do algoritmo A* consiste na sua otimalidade, garantida pelo uso de uma heurística consistente. Além de sempre apresentar um valor superior ao custo real da solução, a distância de Manhattan adaptada ao problema jamais apresenta valores negativos. Assim, é possível assegurar a obtenção do menor caminho entre o estado inicial e o cubo solucionado.

Dado o consumo excessivo de memória por parte do A*, optou-se por propor uma segunda solução baseada  no conceito de Algoritmos Genéticos. Além de possuírem um gasto fixo em termos de memória, normalmente convergem para o resultado com relativa rapidez.

# Capítulo 6
# Algoritmo Genético

A computação evolucionista teve início na década de cinqüenta com os trabalhos de Nils Barricelli: *Esempi numerici di processi di evoluzione* e *Symbiogenetic evolution processes realized by artificial methods*. Esses artigos não obtiveram repercussão até a contribuição de Alex Fraser que publicou uma série de estudos sobre simulações de mecanismos de seleção artificial aplicados em organismos diversos. Os algoritmos genéticos hoje conhecidos contêm grande parte do estudo de Fraser como elemento principal.

Hans Bremermann, inspirado pelos resultados anteriores, conseguiu utilizar o estudo sobre populações para otimizar problemas, fortalecendo ainda mais a técnica recém-descoberta. Ingo Rechenberg e Hans-Paul Schwefel conseguiram resolver problemas complexos de engenharia através de algoritmos genéticos, usando mecanismos de recombinação gênica, mutação e seleção de indivíduos. Apesar de fiascos como a *Machine Evolution*, essa nova forma de buscar soluções cresceu consideravelmente e conjuntamente com o crescimento exponencial do poder computacional, a computação evolucionista ampliou seu espectro de novas técnicas, muitas das quais em desenvolvimento até os dias atuais.

Os algoritmos evolucionistas têm como fonte de inspiração, em sua maioria, fenômenos naturais. Um ótimo exemplo é a técnica de Arrefecimento Simulado (*Simulated Annealing)* que é baseada em um método de resfriamento de metais em estado liquido para a obtenção de sólidos que apresentem baixa energia. Outros exemplos famosos são as Colônias de Formigas (*Ant Colony*) em que a simulação do comportamento de insetos sociais (baixa inteligência individual, mas presentes em grande número) pode fornecer grandes otimizações de problemas, ou ainda as Redes Neurais Artificiais (*Artificial Neural Networks*) e os Sistemas Imunológicos Artificiais (*Artificial Immune System*), ambos inspirados na fisiologia do ser humano.

Uma característica marcante da computação evolutiva é a rapidez de seus resultados, pois quando ela é aplicada de forma correta, o problema converge rapidamente para a solução. Quando o espaço de busca é muito irregular, é possível que se acabe preso em ótimos locais, nome dado a uma posição que atinge o melhor valor dentro de um espaço limitado e específico; geralmente, é um valor próximo do ótimo absoluto, mas está distante do mesmo.

Em determinados casos, um ótimo local já é suficiente para resolver o problema como em uma questão de logística, por exemplo, na qual a simples adoção de um ‘ótimo local’ pode representar uma redução considerável de custos, dispensando gastos com a busca pelo ótimo absoluto, mais difícil de ser encontrado e com pouca diferença em relação à redução de custos já obtida.

Ao contrário de muitos métodos de otimização, os algoritmos genéticos não tendem a consumir memória demasiadamente. Normalmente, os recursos necessários equivalem somente ao armazenamento de duas populações – a parental e a gerada. Por outro lado, de acordo com a função de aptidão, pode-se incorrer em um processo dispendioso em termos de tempo, uma vez que a função pode demorar a convergir para a solução ou mesmo ficar estagnada em um resultado sub-ótimo. Função de aptidão é a nomenclatura dada a uma expressão que julga a qualidade de um indivíduo. Quanto melhor sua aptidão, melhor é o seu resultado. Em problemas de busca informada, muitas vezes a função aptidão é uma heurística, embora esse fato não seja obrigatório.

Assim como em qualquer outro problema de otimização, a sua modelagem é uma preocupação constante dos cientistas evolucionistas. Em termos de recursos computacionais e simulação do comportamento do problema real, quanto melhor for a representação, melhores e mais rápidos serão os resultados encontrados.

## 6.1 Construção do Algoritmo Genético

Os algoritmos genéticos são baseados na teoria de seleção natural de Darwin, inclusive com vários pontos em comum como o fato de ambos apresentarem uma população estável, com um número relativamente constante de indivíduos; a presença de hereditariedade, na forma de herança de material genético; e, por fim, o próprio mecanismo de seleção natural em que poucos indivíduos sobrevivem para a próxima geração graças a sua aptidão natural, representada por seus genes.

Indivíduo é o nome dado a uma instância do problema. No caixeiro viajante, por exemplo, uma rota seja válida ou não pode ser chamada de um individuo. Normalmente, eles são compostos por sua ‘estrutura genética’, o equivalente aos dados internos do problema. Em

geral, o cromossomo é representado por um vetor, sendo que cada elemento componente do mesmo é uma característica do indivíduo, de maneira similar ao código genético original. Novamente no problema do caixeiro, os genes do indivíduo podem ser compostos pela ordem das cidades visitadas.

População equivale a um conjunto de indivíduos diferentes entre si. Cada indivíduo carrega uma estrutura genética diferente nas gerações iniciais, mas a seleção tende a especializar os indivíduos, podendo trazer gerações em que o código genético de vários indivíduos é muito parecido ou até mesmo igual. Essa dominância de um código sobre os demais pode ser aumentada por elitismo – comportamento em que os melhores indivíduos possuem maiores chances de procriação em detrimento dos piores. O elitismo tende a acelerar o algoritmo através de uma diminuição no espaço de busca, o que pode acabar em ótimos locais se o ótimo geral não estiver dentro desse cenário.

O programa tem início com a criação da primeira geração de indivíduos, processo que, normalmente, é feito de forma aleatória. Segue-se então para a seleção de pais, reprodução, mutação e, finalmente, para as gerações seguintes até que os parâmetros de finalização do programa sejam atingidos.

## 6.2 Técnicas de Seleção

De forma similar à teoria da evolução de Darwin, a seleção natural restringe a sobrevivência do código genético somente àquele presente nos mais aptos seja porque eles 'morrem' antes seja por simplesmente não conseguirem um parceiro.

Após a constituição de uma geração, os indivíduos precisam se reproduzir para gerar descentes. Eles serão selecionados pelo algoritmo para se reproduzir entre si, gerando filhos para a próxima geração. Entre os métodos de seleção, destacam-se o torneio, a truncada, a proporcional à aptidão e a por rank.

1. **Torneio:** A seleção através de torneio é simples e eficaz; são selecionados **n** indivíduos do grupo de acesso, sendo o melhor deles 'promovido' a pai. Todos os indivíduos são novamente inseridos no grupo de acesso – inclusive o escolhido – para que sempre exista uma chance constante de seleção. É importante notar que o pior

indivíduo nunca sobreviverá a essa seleção, enquanto o melhor pode se reproduzir somente se for escolhido, mas todas as vezes que ele for escolhido, será elevado à condição de pai. Esse método é mais usado em populações grandes, uma vez que o mesmo dispensa a organização dos indivíduos. Adicionalmente, métodos de ordenação são caros, sobretudo em grandes vetores.

2. **Truncada:** Um grupo de indivíduos é demarcado e somente elementos desse conjunto podem ser selecionados. Essa política de seleção é extremamente elitista, pois somente os melhores indivíduos – ou os não-piores, dependendo da situação – têm chance de se reproduzirem. Esse método é similar ao empregado por criadores de rebanho, já que somente os melhores indivíduos são selecionados para a reprodução.
3. **Seleção Proporcional à Aptidão:** Nesse método, cada indivíduo possui uma chance de ser escolhido de acordo com a sua aptidão relativa. Ela é a razão entre sua aptidão e o somatório das aptidões de todos os indivíduos. O grande problema dessa política de seleção é que, se, em uma população recente (gerações baixas), existir um elemento próximo de um ótimo local, ele terá uma aptidão muito maior em relação à dos outros elementos, fazendo com que os resultados convirjam para o ótimo local.
4. **Seleção por 'Rank':** De modo similar ao item anterior, a seleção por 'rank' permite que todos os indivíduos possam ser selecionados. Por outro lado, a seleção não é diretamente proporcional à aptidão, mas sim à sua colocação num 'rank' de indivíduos. Por exemplo, pode-se dizer que o primeiro indivíduo tem 10% de chance de ser escolhido, o segundo 6%, o terceiro 4% até o último que possui 0,3%. Essa técnica, assim como a anterior, permite uma melhor exploração do espaço de busca, embora, como toda técnica de seleção, possua um certo comportamento elitista, visto que os primeiros colocados tendem a se reproduzir mais.

Não existe um método melhor ou pior que os demais para situações gerais. Problemas específicos podem apresentar melhores comportamentos adotando-se determinada técnica. Por exemplo, uma otimização parcial pode adotar a seleção proporcional à aptidão por estar interessada em um ótimo local, mas não necessariamente na melhor solução, cabendo ao usuário escolher qual método aplicar.

## 6.3 Recombinação

Selecionando dois indivíduos ao acaso, induz-se a recombinação deles, gerando filhos que são resultados da troca de material genético dos pais. Geralmente, os indivíduos não possuem características de sexo distinto, embora esse comportamento possa ser adicionado ao processo evolutivo, só permitindo que indivíduos gerem filhos quando se acasalarem com outros de sexo oposto.

A geração de filhos é representada pela permuta de material genético dos pais em um processo similar ao *crossing-over* (ou *crossover)* – processo de recombinação gênica celular. O *crossover* é, normalmente, representado pelo corte de um único ponto nos dois cromossomos, embora também existam variedades como os múltiplos pontos de *crossover* ou a geração de filhos com maior e menor carga genética.

| **I** | **II** | **III** |
|---|---|---|
| P1 : [C C G T C A T A] | P1 : [C C G T C A T A] | P1 : [C C G T C A T A] |
| P2 : [C G A T T G C T] | P2 : [C G A T T G C T] | P2 : [C G A T T G C T] |
| F1 : [C C G T C G C T] | F1 : [C C G T C G C A] | F1 : [C C G T T A C T] |
| F2 : [C G A T T A T A] | F2 : [C G A T T A T T] | F2 : [C G A T C G T A] |

**Figura 6.1 Exemplos de *Crossing-over:* os filhos F1 e F2 recebem parte do código genético de seus pais. Nos exemplos, é possível ver o comportamento da técnica de ponto de corte único (I), múltiplos pontos de corte (II) e diferentes cargas genéticas (III).**

A reprodução associada à mutação é um dos métodos que permitem buscas amplas em um grande espaço, pois a combinação de indivíduos diferentes pode gerar indivíduos diferentes de um dos pais ou até mesmo dos dois, explorando espaços ainda não percorridos.

## 6.4 Mutação

A mutação é o segundo método responsável pela expansão do espaço de busca. Ela irá alterar o valor de um ou mais alelos. Assim como na reprodução, esse procedimento é usado para inserir novos traços na população, mantendo a busca por melhores resultados em ambientes diversos.

Os algoritmos genéticos em geral possuem uma constante, chamada de taxa de mutação, que representa a chance de um alelo ser modificado. Se for decidido que o gene

sofrerá mutação, seu valor muda; em um algoritmo genético binário, por exemplo, o gene trocará seu valor de 1 para 0 ou de 0 para 1. Em outros casos, pode-se sortear um elemento dentro de um intervalo válido e substituir o valor do alelo selecionado por esse.


**Figura 6.2 Exemplos de mutação: ponto único (I) e múltiplos pontos (II).**

A mutação pode ocorrer somente uma vez ou em vários pontos separados do mesmo cromossomo. Além disso, caso se queira manter o elitismo, pode-se fazer com que o melhor ou os melhores indivíduos não sofram mutação nunca, mantendo, assim, sua ótima performance.

## 6.5 Próxima Geração

Após os processos de seleção, reprodução e mutação, finalmente se tem material suficiente para escolher a próxima geração. Os filhos são ordenados a partir de sua aptidão, sendo os melhores selecionados para compor a próxima geração. Depois de eleita a próxima geração, o ciclo se reinicia; assim, casais são sorteados, código genético é recombinado e alelos são modificados através de mutações até que um dos critérios de parada do algoritmo seja alcançado.

Mecanismos de seleção também poderiam ser utilizados como artifícios que permitiriam que o melhor pai fosse preservado da antiga geração para a atual. Em casos extremos, todos os pais serão inseridos no grupo de possíveis selecionados, gerando um comportamento elitista. Esse último cenário pode se tornar prejudicial, pois se os pais forem ótimos locais, dificilmente serão deixados de lado, podendo nunca encontrar a solução ótima. Indivíduos não selecionados para a próxima geração são sumariamente descartados.

## 6.6 Segunda Solução Proposta

A segunda solução proposta por esse trabalho é a resolução do cubo de Rubik baseada em técnicas da computação evolucionista, mais especificamente nos algoritmos genéticos. Para os pesquisadores, só possui relevância a completa resolução do cubo, não importando a obtenção de ótimos locais. Desse modo, deve-se tomar cuidado com a implementação do Algoritmo Genético, uma vez que ele tende a encontrar estados sub-ótimos com extrema eficiência em detrimento da localização de ótimos absolutos; especificamente, o cubo solucionado.

Um estudo sobre soluções do cubo de Rubik a partir de Arrefecimento Simulado feito por Rocha (2007) não apresentou bons resultados. Para simplificar o problema, foi usado um cubo de 2x2x2 ao invés do 3x3x3 padrão, que apresenta $3{,}6x10^6$ estados possíveis, tornando uma busca por força bruta possível. Além disso, ele possui o mesmo fator de ramificação do cubo original (dezoito movimentos, três para cada face). Mesmo com esse cubo simplificado, Rocha não logrou êxito, visto que suas soluções envolviam conjuntos de sessenta a seiscentos movimentos em média contra vinte e cinco da solução proposta por Rokicki (2008) na solução do cubo maior e mais complexo.

O primeiro passo para a criação de um AG é similar à criação de qualquer outro algoritmo, ou seja, a modelagem do problema em termos ‘técnicos’. O modelo de cubo que será utilizado é o mesmo presente na primeira solução, incluindo a heurística utilizada; nessa solução, porém, é necessário que se definam mais características, relativas exclusivamente à solução de AGs.

A escolha mais difícil consiste em como representar um estado do cubo em forma de vetor eficiente. O cubo poderia ser representado a partir de um vetor de cinqüenta e quatro posições (cada posição para um *cubie*), o que tornaria cada indivíduo longo e caro, além de tornar as operações demoradas. A idéia proposta então foi a de armazenar os movimentos do cubo, não o objeto em si. Dessa forma, teria-se um vetor de vinte e cinco elementos que, apesar de ainda grande, é muito prático para fins de reprodução e mutação as quais representam, no presente modelo, um novo direcionamento.

```
4    Integer[] cromossomo = new Integer[30];
```

**Figura 6.3 Representação de um Cromossomo.**

Na segunda solução proposta, cada alelo representa um movimento a ser executado no cubo inicial, buscando a solução do problema. Cada operador deve ser executado seguindo a ordem estabelecida por sua posição no cromossomo; desse modo, o movimento representado pelo segundo alelo irá operar sobre o cubo inicialmente dado após a operação da primeira posição do cromossomo. Um cromossomo ótimo seria representado por um vetor de vinte e cinco elementos; seu tamanho foi aumentado visando uma relaxação do problema, uma vez que com mais movimentos mais soluções estão disponíveis. Desse modo as soluções encontradas podem não ser sempre ótimas, mas podem ser consideradas razoáveis, já que apresentam até cinco movimentos a mais que a marca alcançada por Rokicki (2008).

A função de avaliação presente nesse algoritmo é a mesma da primeira solução: uma adaptação da distância de Manhattan. A avaliação funciona da seguinte forma: seleciona-se o primeiro operador, aplicando-o no estado inicial e calcula-se sua aptidão. Guarda-se esse valor e em seguida aplica-se o primeiro e o segundo operadores no estado inicial, comparando essa aptidão com a anterior; a que for melhor sobrevive, enquanto a outra é descartada. E assim se prossegue até que se tenha aplicado todo o cromossomo e se possua somente a melhor aptidão encontrada em todo o processo.

```
57  public void aval(Cubo c){
58   int melhorAval = 500;
59   for(int indice =0; indice < 30; indice++ ){
60      switch(this.cromossomo[indice]){
61          case 1:{c.FH();break;}
62          case 2:{c.F180();break;}
63          /*
64          AINDA EXISTEM OS CASES 3 ATÉ 16!
65          */
66          case 17:{c.D180();break;}
67          case 18:{c.DAH();break;}
68          default:{
69                      indice--;
70                      break;
71              }
72          }//fim switch
73          if (melhorAval > c.avaliacao()){
74              melhorAval = c.avaliacao();
75          }
76      }
77      this.avaliacao = melhorAval;
78  }//fim aval;
```

**Figura 6.4 Função de Aptidão: repare que somente o menor valor heurístico é conservado.**

Definidos os pontos mais importantes, deve-se gerar a população inicial. Ela é gerada de forma aleatória, de modo a forçar com que a busca ocorra em nenhum espaço específico. Em algumas variantes da computação evolucionista, como a evolução diferencial, o espaço de busca é limitado, aumentando a eficácia de alguns problemas e dificultando a execução de outros.

```
7   public Individuo(Cubo c){
8       Cubo auxCubo = new Cubo(c);
9       int aux = 0;
10      for (int i = 0; i < 30; i++){
11              aux = 1+(int)(Math.random()*18);
12              cromossomo[i] = aux;
13      }
14      auxCubo = new Cubo(c);
15      this.simplificacao(auxCubo);
16      this.aval(auxCubo);
17  }
```

**Figura 6.5 Criação de um Indivíduo Pertencente à Geração Parental.**

A técnica de seleção utilizada é a de torneio em que são escolhidos aleatoriamente dois grupos de dois indivíduos cada, sendo que o melhor indivíduo de cada grupo será submetido à recombinação com o melhor do grupo anterior. Ao final, todos os indivíduos selecionados (os escolhidos e não-escolhidos) são devolvidos ao grupo de possíveis pais para novas seleções até que o numero desejado de filhos seja alcançado.

```
179   for(int i = 0; i < 50; i=i+2){
180       cand1 = (int)(Math.random()*49);
181       cand2 = (int)(Math.random()*49);
182       if(otima[cand1].avaliacao > otima[cand2].avaliacao) {
183           pai = cand2;
184       }
185       else{
186           pai = cand1;
187       }
188
189       cand1 = (int)(Math.random()*49);
190       cand2 = (int)(Math.random()*49);
191       if(otima[cand1].avaliacao > otima[cand2].avaliacao) {
192           mae = cand2;
193       }
194       else{
195           mae = cand1;
196       }
```

**Figura 6.6 Seleção de Pais, Seguindo a Modalidade Torneio.**

A partir da carga genética dos pais, serão gerados dois filhos que herdarão parte do material de cada um, conforme visto no código. O tipo de crossover empregado é o de ponto único de corte, uma vez que os demais métodos não acrescentam nenhuma vantagem à solução do problema. O método que permite filhos com diferentes cargas genéticas seria pior ainda, tendo em vista que há a possibilidade de apresentar resultados com mais de vinte e cinco movimentos, resultado que não é o pretendido.

```
299   int divisor = (int)(Math.random()*29);
300   for (int i = 0; i<divisor; i++){
301           filho1.cromossomo[i] = pai.cromossomo[i];
302           filho2.cromossomo[i] = mae.cromossomo[i];
303   }
304   for (int i = divisor; i < 30; i++){
305           filho2.cromossomo[i] = pai.cromossomo[i];
306           filho1.cromossomo[i] = mae.cromossomo[i];
307   }
```

**Figura 6.7 Crossover Gerando dois filhos.**

Conforme observado no código, um número é sorteado aleatoriamente entre zero e vinte e nove, garantindo que pelo menos o último alelo sempre trocará de posição, diminuindo a chance de uma geração de indivíduos idênticos aos pais, embora esse fato ainda possa ocorrer caso os pais escolhidos sejam os mesmos.

O comportamento que foi aplicado no algoritmo genético do trabalho é a possibilidade de ocorrerem várias mutações em um mesmo cromossomo. A taxa de mutação fixada é de 4%, assegurando que, em média, um alelo troque de valor em cada cromossomo. Contudo, a fixação deste comportamento não constitui uma regra geral. Como essa troca de alelos é aleatória, há uma chance - mais especificamente, uma em dezoito - que o alelo não troque de valor, reduzindo as possibilidades de real mutação para, aproximadamente, 3,8%. Todos os filhos, depois de criados, passam pelo processo de mutação, não permitindo nenhum tipo de elitismo.

```
309   int chance = 0;
310   for(int i = 0; i < 30; i++){
311       chance = 1+(int)(Math.random()*100);
312       if (chance <= 4){
313           filho1.cromossomo[i] = 1+(int)(Math.random()*18);
314       }
315   }
316
317   for(int i = 0; i < 30; i++){
318       chance = 1+(int)(Math.random()*100);
319       if (chance <= 4){
320           filho2.cromossomo[i] = 1+(int)(Math.random()*18);
321       }
322   }
```

**Figura 6.8 Mecanismo de Mutação.**

A geração randômica de indivíduos, a reprodução dos mesmos e a mutação podem, ocasionalmente, gerar cromossomos inválidos que, normalmente, representam estados impossíveis de serem alcançados. No modelo proposto, optou-se por corrigir os cromossomos a partir de seus erros, evitando que estados inválidos, mesmo que penalizados, espalhem-se ou ainda, que a população fique estagnada por possuir estados inválidos em excesso.

```
338  public void simplificacao(Cubo c){
339    Cubo auxCubo = new Cubo(c);
340    for(int i = 0; i<26; i++){
341      switch(this.cromossomo[i]){
342        case 1:{
343           if(this.cromossomo[i+1] == 1){
344              this.cromossomo[i] = 2;
345              for(int j=i+1; j<25; j++){
346                  this.cromossomo[j] = this.cromossomo[j+1];
347              }
348              this.cromossomo[25]=1+(int)(Math.random()*18);
349              i--;
350              break;
351           }
```

**Figura 6.9 Simplificação do Cromossomo.**

O código da imagem 6.9 pertence ao método simplificação que percorre todo o cromossomo em busca de estados que possam ser simplificados. No trecho destacado, percebe-se que, se dois alelos vizinhos apresentam o comportamento sugerido (ambos alelos possuem o valor 1), é possível simplificar o cromossomo aplicando o valor 2 à primeira posição e adiantando os demais alelos. No caso, foram simplificados dois movimentos de rotação da face dianteira em 90º no sentido horário por um movimento de 180º.

Por fim, a seleção de indivíduos sobreviventes para a próxima geração é dada pela seleção dos melhores filhos da geração passada, não guardando nenhum elemento da população anterior.

```
205  //Proxima Geração
206  for(int i = 0; i < 50; i++){
207      for(int j = 0; j < 100; j++){
208          if (melhorAval > filho[j].avaliacao){
209              indice = j;
210              melhorAval = filho[j].avaliacao;
211          }
212      }
213      melhorAval = 100;
214      otima[i] = filho[indice];
215      filho[indice].avaliacao = 100;
216  }
```

**Figura 6.10 Sobrevivência de Indivíduos para a Próxima Geração**

### 6.6.1 – Resultados

Por apresentar uma topologia saturada de ótimos locais, a busca pelo melhor resultado se torna complexa, elevando o tempo necessário para a sua localização. Contrariando o comportamento padrão, o tempo de execução se mostrou bastante dependente da geração parental, visto que muitos casos não convergiram satisfatoriamente. Caso nenhum resultado seja encontrado em até três mil gerações, a busca é reiniciada com uma geração parental diferente das anteriores. Essa medida foi tomada a fim de contornar a distorção associada à dependência da geração inicial.

Esse comportamento não é raro, pois muitas aplicações também tendem a agir de forma similar. O emprego de outros métodos, como o crossover de tamanho variado ou de vários pontos, a mutação de vários pontos, as recombinações por rank ou mesmo uma taxa de mutação diferente, poderia corrigir essa conduta, mas seriam necessários exaustivos testes para a aplicação efetiva de tais técnicas.

Objetivando reduzir o tempo dispendido, um cromossomo maior foi criado. Embora isso permita a geração de resultados diferentes do ótimo, eles ainda podem ser considerados bons, uma vez que seu tamanho máximo é de trinta movimentos, cinco a mais que o defendido por Rokicki (2008).

Um grande benefício apresentado pelo Algoritmo Genético é seu custo fixo em termos de memória. O espaço ocupado por qualquer geração tende a ser constante. Contudo, ele poderia variar em algumas situações como, por exemplo, na utilização de uma política de

correção ou na existência de indivíduos de carga genética de tamanho variável. A constância do gasto de memória possibilita a projeção de um tamanho de população ótimo, consistente com o *hardware* disponível.

# Capítulo 7
# Considerações Finais

O objetivo deste trabalho é fornecer um estudo sobre o cubo de Rubik. Tendo em vista que as tentativas de força bruta ainda não são praticáveis com o hardware disponível, as resoluções apresentadas se basearam nas buscas informadas e no algoritmo genético. Será feita uma comparação entre os algoritmos implementados, identificando os pontos fortes e fracos de cada um. Além disso, serão sugeridas algumas modificações que poderiam implicar em melhores resultados.

## 7.1 Comparação das Técnicas

Ambas as técnicas empregadas nesse trabalho geraram boas soluções. Por se tratar de uma heurística consistente, todos os resultados apresentados pelo algoritmo A* se resumem a caminhos ótimos. A segunda solução, por sua vez, possui um limite de profundidade de trinta movimentos, superior ao provado por Rokicki (2008), garantindo sempre um bom resultado, ainda que não seja o ótimo.

A tabela 7.1 apresenta os resultados obtidos por meio da execução das soluções propostas. A fim de evitar qualquer comportamento elitista, os valores de todos os campos equivalem à média da solução de três problemas distintos, exceto os exemplos destacados na tabela. Cada problema foi gerado aleatoriamente através do método **estadoInicial**, mostrado na imagem 7.1.

O campo ‘Ordem’, presente na tabela 7.1, foi usado como parâmetro de entrada para o método **estadoInicial**; ele também é o número necessário de movimentos para que solução seja alcançada. A notação ‘Nodos’ equivale ao número de nós abertos e não explorados da árvore de busca até que o resultado seja encontrado.

O campo ‘Tempo’ foi calculado pela própria IDE NetBeans, que possui um medidor de tempo próprio. Esses fatores foram obtidos em um Pentium Dual-Core de 2GHz, munido de 1 Giga de memória e utilizando o Windows Xp SP2.

Os parâmetros ‘Profundidade’ e ‘Geração’ referem-se exclusivamente aos Algoritmos Genéticos. O primeiro corresponde ao número de movimentos necessários entre o estado inicial e o cubo resolvido, definindo, assim, a qualidade da solução encontrada. O segundo campo equivale ao número de gerações percorridas até que a solução seja encontrada, ou seja, corresponde ao número de recombinações, mutações e seleções executadas.

**Tabela 7.1 – Resultados encontrados**

| | A* | | Genético | | |
|---|---|---|---|---|---|
| **Ordem** | Nodos | Tempo | Profundidade | Geração | Tempo |
| **1** | 9 | 1 s | 1 | 0 | 1s |
| **2** | 31 | 1 s | 2 | 31 | 2s |
| **3** | 213 | 1 s | 3 | 96 | 3s |
| **6** | 9805 | 25m 6s | **6** | 247 | 15s |
| **10** | 14793(1) | 1h 27m 6s | 11 | 5142 | 16m |
| **13** | (2) | - | 14 | 43627 | 41m |
| **20** | (2) | - | **23**(3) | $10^9$ | 1h 53m 27s |

1 – Somente um caso em seis execuções gerou resultado;

2 – Nenhum resultado foi encontrado (Falta de Memória);

3 – Foi executado somente uma instância, devido ao custo (tempo).

No A*, por exemplo, a solução de ordem 6 gerou, em média, 9805 nodos até que o cubo fosse totalmente resolvido em pouco mais de vinte e cinco minutos. O Algoritmo Genético, por sua vez, alcançou em 15 segundos a resolução do cubo, sendo necessários seis movimentos assim como produção de 247 gerações.

Somente uma execução em seis tentativas gerou resultados de ordem 10 no A*. Possivelmente essa instância foi gerada por uma seqüência de movimentos não redundantes, não configurando, de fato, um problema de ordem 10. Dada a limitação de memória existente, nenhum problema de ordem superior a 10 pode ser solucionado no A*. Adicionalmente, o tempo de busca torna proibitivo a execução de várias instâncias de grande ordem.

Por outro lado, o comportamento apresentado pelo Algoritmo Genético possibilitou a busca de instâncias de maior ordem. Por exemplo, a única solução encontrada para um caso de

ordem 20 demorou quase duas horas de execução e não gerou um resultado ótimo, tendo em vista que o número de movimentos necessários para sua solução equivale a sua ordem, ou seja, 20 é menor que o alcançado (profundidade 23).

```
179  public void estadoInicial(int k){
180      int face = 0;
181      for (int i = 0; i<k; i++){
182              face = 1+(int)(Math.random()*18);
183              switch(face){
184                      case 1:{
185                              this.FH();break;
186                      }
187                      case 2:{
188                              this.F180();break;
189                      }
190                      case 3:{
191                              this.FAH();break;
192                      }
```

**Figura 7.1 Geração de Cubo Embaralhado**

### 7.1.1 A*

O algoritmo A* foi utilizado nesse trabalho por vários motivos. O primeiro foi possibilitar a aplicação da nova heurística proposta em um algoritmos de busca. Adicionalmente, outro fator que justificou a escolha desse método foi a sua otimalidade.

Se comparado aos algoritmos genéticos, seu tempo de execução e o consumo de memória podem restringir o uso do A*. Mesmo com a adoção de uma árvore hierárquica o tempo acabou se tornando um fator limitativo. Por outro lado, sempre teremos a melhor solução existente.

### 7.1.2 Algoritmo Genético

Geralmente, os algoritmos genéticos tendem a refinar os problemas, encontrando-se, dessa maneira, boas soluções. Não obstante, ótimas soluções são mais caras e difíceis de se conseguir, comportamento apresentado com a aplicação usada no trabalho.

A resolução através do AG independe da complexidade do estado inicial. Dessa forma, qualquer que seja estado, o esforço computacional médio depende diretamente da primeira população e do desenvolvimento do algoritmo, tornando este excelente para a resolução de problemas complexos.

Uma das grandes vantagens do algoritmo genético sobre a outra resolução proposta é justamente o seu custo em memória, visto que o consumo desse recurso é constante durante toda a aplicação, sendo equivalente ao tamanho da população inicial.

## 7.2 Trabalhos futuros

Embora a tecnologia atual ainda restrinja os resultados esperados como, por exemplo, o custo, ainda alto, de memória no algoritmo A*, o presente trabalho apresentou os resultados esperados. Em tese, algumas modificações podem ser feitas nos algoritmos para, posteriormente, terem seus resultados comparados aos aqui relatados.

A eficácia da primeira solução proposta é diretamente proporcional à qualidade da heurística; dessa maneira, quando uma nova e melhor forma de conhecimento sobre a solução do cubo for descoberta, talvez a busca seja mais barata de localizar. Outra alternativa seria a utilização de diferentes algoritmos com uso de heurística, como uma busca bidirecional informada, ou mesmo o IDA* (*Iterative-Deepening A**). Possivelmente, o uso de linguagens mais ágeis, como o C, poderia acelerar o processo de busca.

O emprego de penalidades pode ser uma alternativa às correções aplicadas aos cromossomos que apresentarem redundância. Sob essa forma, o algoritmo poderia convergir com maior rapidez e com maior grau de acurácia em direção ao ótimo total. O uso de um cromossomo maior, possibilitando um maior número de movimentos, facilitaria a busca por um bom resultado; possivelmente, ele não será o melhor, já que quanto maior o número de movimentos, pior é a solução; mas, ainda assim, é um caminho razoável. Por fim, o uso de diferentes técnicas evolucionistas como a evolução diferencial (*Differential Evolution*) poderia apresentar melhores resultados que aqueles obtidos no modelo proposto.

# Referências Bibliográficas


DEITEL, H. M.; DEITEL, P.J. Java: Como Programar. 4. ed. São Paulo: Bookman, 2004. 1386 p.

DRUPAL. ***26f Now Claimed Proven Sufficient.*** **(2007a).** Disponível em <http://cubezzz.homelinux.org/drupal/?q=node/view/86#comment-346 >. Último acesso em maio de 2008.

_____. ***26f Now Claimed Proven Sufficient.*** **(2007b).** Disponível em <http://cubezzz.homelinux.org/drupal/?q=node/view/86#comment-372 >. Último acesso em maio de 2008.

FISCHER, R. *Garry Kasparov vs Deep Blue 1996 Games.* 1996. Disponível em: <http://www.bobby-fischer.net/Garry_Kasparov_vs_Deep_Blue_1996_games.html> Último acesso em maio de 2008.

_____. *Garry Kasparov vs Deep Blue 1997 Games.* 1997. Disponível em: <http://www.bobby-fischer.net/Garry_Kasparov_vs_Deep_Blue_1997_games.html> Último acesso em maio de 2008.

FREY, A. H.; SINGMASTER, D. *Handbook of Cubik Math.* 1.ed. Enslow Publishers, 1982. 204 p.

HOFSTADTER, D.R. Metamagical Themas: The Magic Cube's Cubies are Twiddled by Cubists and Solved by Cubemeisters. **Scientific American**, v. 244, p. 20-39, mar. 1981.

KOCIEMBA, H. *Cube Explorer*. 2007. Disponível em: <http://kociemba.org/download.htm> Último acesso em maio de 2008.

KORF, R.E. Macro-operators: A Weak Method for Learning. **Artificial Intelligence**, v. 26, n.1, pp. 35-77, 1985.

_____. *Finding optimal solutions to Rubik's Cube using pattern databases*. 1997. In: AAAI-97 Fourteenth National Conference on Artificial Intelligence. Providence, United States of America, 1997.

KORF, R.E.; SCHULTZE, P. *Large-Scale Parallel Breadth-First Search*. 2005. In: AAAI-05 Twentieth National Conference on Artificial Intelligence. Pittsburgh, United States of America, 2005.

KUNKLE, D.; COOPERMAN, G. *Twenty-Six Moves Suffice for Rubik's Cube*. 2007. In: ISSAC'07 International Symposium on Symbolic and Algebraic Computation. Ontario, Canada, 2007.

*Mainstream Science on Intelligence*. Nova Iorque: Wall Street Journal, 13 de dezembro de 1994. Disponível em: <http://www.udel.edu/educ/gottfredson/reprints/1994WSJmainstream.pdf> Último acesso em maio de 2008.

NEISSER, U.; BOODOO, G.; BOUCHARD JR., T. J.; BOYKIN, A. W.; BRODY, N.; CECI, S. J.; HALPERN, D. F.; LOEHLIN, J. C.; PERLOFF, R.; STERNBERG, R. J.; URBINA, S. Intelligence: Knowns and Unknowns. **American Psychologist**, v. 51, n. 2, p. 77-101, fev. 1996.

PASSINI, M. Um Estudo Quantitativo de Estratégias de Busca Heurística na Resolução de Problemas. Juiz de Fora, 1991. Trabalho de Conclusão de Curso (Bacharelado em Matemática/ Informática). UFJF – MG.

PRIEDITIS, A. E. Machine discovery of effective admissible heuristics. **Machine Learning**, v. 12, n. 1-3, p. 117-141, ago. 1993.

RADU, S. *A New Upper Bounds on Rubik's Cube Group*. 2007. Disponível em: <http://www.risc.uni-linz.ac.at/publications/download/risc_3122/uppernew3.ps.> Último acesso em maio de 2008.

REID, M. ***New Upper Bounds*. (1995a)**. Disponível em: <http://www.math.rwth-aachen.de/~Martin.Schoenert/Cube-Lovers/michael_reid__new_upper_bounds.html> Último acesso em maio de 2008.

_____. ***Superflip requires 20 face turns*. (1995b)**. Disponível em: < http://www.math.rwth-aachen.de/~Martin.Schoenert/Cube-Lovers/michael_reid__superflip_requires_20_face_turns.html> Último acesso em maio de 2008.

_____. *Optimal Cube Solver*. 1997. Disponível em: <http://www.math.ucf.edu/~reid/Rubik/description.html> Último acesso em maio de 2008.

ROCHA, C. A. *Solving Rubik's Cube.* 2007. Disponível em: <http://fab.cba.mit.edu/classes/MIT/864.05/people/rocha/project/> Último acesso em maio de 2008.

ROKICKI, T. *In search of: 21f*s and 20f*s; a four month odyssey*. 2006. Disponível em <http://cubezzz.homelinux.org/drupal/?q=node/view/56> Último acesso em maio de 2008.

_____. *Twenty-Five Moves Suffice for Rubik's Cube*. 2008. Disponível em: < http://tomas.rokicki.com/Rubik25.pdf > Último acesso em maio de 2008.

RUSSEL, S.; NORVIG, P. *Artificial Intelligence: A Modern Approach*. 1. ed. Upper Saddle River: Prentice-Hall, 1995. 932 p.