Algoritmos de Busca para Resolução do Cubo de Rubik

Autor: Carlos Matheus R Martins
Matrícula: 514690
Disciplina: Inteligência Artificial

1. Introdução

Basicamente o código está dividido em 2 duas classes, a primeira é RubiksCube responsável pela representação do cubo e suas ações onde foi inspirada nas representações eficientes no estado A.I. A classe EstatisticasBusca é responsável por guardar informações da solução encontrada e as métricas de performance do código.

Para representações na matriz usamos a seguinte notação:

- **Face 0:** Frontal (F)
- **Face 1:** Direita (R)
- **Face 2:** Traseira (B)
- **Face 3:** Esquerda (L)
- **Face 4:** Superior (U)
- **Face 5:** Inferior (D)

Movimento	Descrição	Movimento Inverso
F	Face frontal horário	F'
R	Face direita horário	R'
U	Face superior horário	U'
B	Face traseira horário	B'
L	Face esquerda horário	L'
D	Face inferior horário	D'

Os movimentos básicos foram baseados na mecânica descrita por Joyner, D. (2008) "Adventures in Group Theory". Já nos movimentos inversos foi usada uma estratégia de movimento anti-horário = 3 x movimento horário. Já que matematicamente $R^{-1} = R^3$ (anti-horário = 3× horário).

A Heurística foi aplicado o Padrão Cruzado, estratégia muito utilizada na cultura no speedcubing, desenvolvida por Jessica Fridrich. Onde o centro recebe o peso um valor maior que as bordas para facilitar as rotações.

2. Algoritmos de Busca Implementados

2.1 Busca em Largura (BFS - Breadth-First Search)

Funcionamento:

O algoritmo BFS explora o espaço de estados de forma sistemática, visitando todos os nós de um nível antes de prosseguir para o próximo nível. No contexto do Cubo de Rubik:

1. **Inicialização:** Coloca o estado inicial do cubo em uma fila FIFO
2. **Expansão:** Remove o primeiro estado da fila e gera todos os 12 possíveis sucessores (aplicando cada movimento F, R, U, B, L, D, F', R', U', B', L', D')
3. **Verificação:** Para cada sucessor, verifica se é o estado resolvido
4. **Controle de ciclos:** Mantém um conjunto de estados visitados para evitar loops infinitos
5. **Continuação:** Adiciona novos estados à fila e repete até encontrar solução

Vantagens:

- Garante encontrar a solução ótima (menor número de movimentos)
- Completude: sempre encontra solução se ela existir
- Simplicidade de implementação

Desvantagens:

- Alto consumo de memória (exponencial)
- Pode ser lento para problemas complexos

Complexidade:

- **Temporal:** O(b^d), onde b=12 (fator de ramificação) e d=profundidade da solução
- **Espacial:** O(b^d) - armazena todos os nós dos níveis explorados

2.2 Busca em Profundidade com Aprofundamento Iterativo (IDDFS)

Funcionamento:

IDDFS combina as vantagens do DFS (baixo uso de memória) com a garantia de otimalidade do BFS:

1. **Iteração por níveis:** Executa busca em profundidade limitada para d=0, 1, 2, ..., até encontrar solução
2. **DFS limitado:** Para cada limite, aplica busca em profundidade recursiva

3. **Backtracking:** Quando atinge o limite ou encontra estado já visitado, retrocede
4. **Controle de ciclos:** Usa conjunto de visitados que é limpo a cada nova iteração de profundidade

Processo recursivo:

```
função DFS_limitado(estado, caminho, limite):
    se estado.resolvido(): retorna caminho
    se len(caminho) >= limite: retorna NULL
    se estado em visitados: retorna NULL

    marca estado como visitado
    para cada movimento em [F, R, U, B, L, D, F', R', U', B', L', D']:
        novo_estado = aplicar_movimento(estado, movimento)
        resultado = DFS_limitado(novo_estado, caminho + [movimento], limite)
        se resultado != NULL: retorna resultado

    remove estado dos visitados (backtracking)

    retorna NULL
```

Vantagens:

- Uso de memória linear O(d)
- Garante solução ótima
- Completude

Desvantagens:

- Recomputa estados em iterações anteriores (overhead temporal)

Complexidade:

- **Temporal:** O(b^d) - mas com overhead de recomputação
- **Espacial:** O(d) - apenas armazena o caminho atual

2.3 Algoritmo A* com Heurística

Funcionamento:

A* é um algoritmo de busca informada que usa uma função de avaliação f(n) = g(n) + h(n):

- **g(n):** Custo real do caminho do estado inicial até n
- **h(n):** Estimativa heurística do custo de n até o objetivo
- **f(n):** Estimativa do custo total do caminho através de n

Processo:

1. **Lista aberta:** Mantém nós a serem explorados ordenados por f(n) (min-heap)
2. **Lista fechada:** Conjunto de nós já processados
3. **Expansão:** Sempre expande o nó com menor f(n)
4. **Atualização:** Calcula g, h e f para cada sucessor
5. **Terminação:** Para quando encontra o estado objetivo

Heurística do Padrão Cruzado:

python

```
def heuristica_padrao_cruzado(cubo):
    pontuacao = 0
    for face in range(6):
        # Penaliza centro incorreto (peso 3)
        if centro_incorreto: pontuacao += 3
        # Penaliza bordas incorretas (peso 2 cada)
        bordas_incorretas = conta_bordas_erradas(face)
        pontuacao += bordas_incorretas * 2
    return pontuacao
```

Esta heurística é **admissível** (nunca superestima o custo real) e **consistente**, garantindo que A* encontre a solução ótima.

Vantagens:

- Otimalidade garantida (com heurística admissível)
- Eficiência guiada pela heurística
- Flexibilidade para diferentes heurísticas

Desvantagens:

- Performance dependente da qualidade da heurística
- Alto uso de memória

Complexidade:

- **Temporal:** O(b^d) no pior caso, mas pode ser melhor com boa heurística
- **Espacial:** O(b^d) - armazena lista aberta e fechada

2.4 Busca Bidirecional

Funcionamento:

Executa duas buscas simultâneas que se encontram no meio:

1. **Busca progressiva:** Do estado inicial em direção ao objetivo
2. **Busca regressiva:** Do estado resolvido em direção ao inicial (aplicando movimentos inversos)
3. **Expansão alternada:** A cada nível, expande ambas as fronteiras
4. **Teste de interseção:** Verifica se algum estado foi visitado por ambas as buscas
5. **Construção da solução:** Concatena os caminhos quando há interseção

Processo de movimentos reversos:

codigo:

 # Busca regressiva aplica movimentos inversos

for movimento in [F, R, U, B, L, D, F', R', U', B', L', D']:

movimento_reverso = inverter(movimento) # F → F', R → R'

novo_estado = aplicar(estado_atual, movimento_reverso)

Construção da solução final:

caminho_completo = caminho_progressivo + inverter_sequencia(caminho_regressivo)

Vantagens:

- Redução dramática do espaço de busca: O(b^(d/2)) vs O(b^d)
- Eficiência superior para problemas complexos
- Mantém otimalidade

Desvantagens:

- Implementação mais complexa
- Requer função de movimentos inversos
- Overhead de manter duas estruturas

Complexidade:

- **Temporal:** O(b^(d/2)) - redução exponencial
- **Espacial:** O(b^(d/2)) - significativamente menor que métodos tradicionais

3. Análise Comparativa de Complexidade

3.1 Resumo das Complexidades

Algoritmo	Tempo	Espaço	Otimalidade	Completude
BFS	O(b^d)	O(b^d)	✓	✓
IDDFS	O(b^d)	O(d)	✓	✓
A*	O(b^d)*	O(b^d)	✓**	✓**
Bidirecional	O(b^(d/2))	O(b^(d/2))	✓	✓

*Com boa heurística pode ser significativamente melhor
**Com heurística admissível e consistente

3.2 Análise Teórica e Prática

Para o Cubo de Rubik 3x3x3:

- **Fator de ramificação (b):** 12 movimentos possíveis
- **Profundidade típica (d):** 2-20 movimentos dependendo do embaralhamento
- **Estados possíveis:** ~4.3 × 10^19 (número de Deus = 20)

Crescimento exponencial:

- d=5: 12^5 = 248,832 estados
- d=10: 12^10 ≈ 6.2 × 10^10 estados
- d=15: 12^15 ≈ 1.5 × 10^16 estados

3.3 Performance Observada nos Testes

Com base nos resultados experimentais obtidos:

Problemas simples (2-3 movimentos):

- Todos os algoritmos performam similarmente
- Diferença negligível em tempo e memória
- BFS e A* ligeiramente mais rápidos

Problemas médios (4-5 movimentos):

- **Bidirecional** mostra superioridade clara
- **IDDFS** mantém baixo uso de memória
- **A*** performance dependente da qualidade da heurística
- **BFS** começa a mostrar limitações de memória

Projeção para problemas complexos (>10 movimentos):

- **Bidirecional** seria o único viável
- **BFS** e **A*** inviáveis por memória
- **IDDFS** viável mas extremamente lento

---

4. Implementação e Otimizações

4.1 Representação do Estado

- **NumPy arrays:** Uso de `dtype=np.int8` para otimizar memória e operações matriciais.
- **Hash eficiente:** `estado.tobytes()` para identificação única de estados
- **Cópia profunda:** `np.copy()` para evitar efeitos colaterais

4.2 Controle de Ciclos

- **Sets em Python:** Operações O(1) para verificação de pertencimento
- **Hash de estados:** Conversão de arrays para bytes para usar como chave

4.3 Otimizações Específicas

BFS:

- `collections.deque` para operações FIFO eficientes
- Monitoramento de pico de memória

IDDFS:

- Backtracking automático via recursão
- Limpeza de visitados entre iterações

A*:

- `heapq` para fila de prioridade eficiente
- Contador para desempate no heap

Bidirecional:

- Alternância entre expansões para balanceamento
- Cálculo eficiente de movimentos inversos

5. Resultados e Conclusões

5.1 Principais Descobertas

1. **Eficácia da Busca Bidirecional:** Demonstrou superioridade para problemas complexos
2. **Trade-off Tempo vs Memória:** IDDFS oferece melhor uso de memória à custa de tempo
3. **Importância da Heurística:** A* pode ser muito eficiente com heurística bem calibrada
4. **Limite prático do BFS:** Inviável para problemas com >6-7 movimentos

5.2 Recomendações de Uso

Para implementações práticas:

- **Problemas simples (≤4 movimentos):** BFS ou A*
- **Recursos limitados de memória:** IDDFS
- **Problemas complexos (>5 movimentos):** Busca Bidirecional
- **Quando otimalidade é crítica:** BFS ou A* com heurística admissível

5.3 Ideias de otimização

Usar processamento em paralelo pode ter um ganho muito significativo de performance, uma heurística que não precise de um grande quantidade de memória também é de grande ajuda.

6. Referências

- Joyner, D. (2008). *Adventures in Group Theory: Rubik's Cube, Merlin's Machine, and Other Mathematical Toys*. Baltimore: Johns Hopkins University Press.
- Fridrich, J. (1997). *Fridrich Method*. Speedcubing community contribution.
- Russell, S., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.
- Korf, R. E. (1997). Finding optimal solutions to Rubik's Cube using pattern databases. *AAAI-97 Proceedings*.