Trabalho Individual 1 - Algoritmos de Busca para Resolução do Quebra Cabeça de Oito Peças

Thauany Moedano – RA: 92486
Departamento de Ciência e Tecnologia
Universidade Federal de São Paulo
São José dos Campos, Brasil
t.moedano@unifesp.br

***Resumo* — Este trabalho mostra a implementação de três algoritmos de busca para a resolução do problema clássico do quebra cabeça de oito peças e analisa otimalidade, completeza e eficiência de cada um dos algoritmos levantando as principais vantagens e desvantagens no uso de cada estratégia de busca.**

Palavras-chave* — *busca, profundidade, gulosa, largura, heurística.

I. Introdução

Buscas de maneira geral são muito utilizadas no ramo da computação. Diversos problemas podem ser resolvidos utilizando diferentes estratégias de buscas. As buscas cegas são um gênero de busca que utilizam apenas a informação fornecida no início do problema. Isto pode ser uma desvantagem quando a árvore do problema possui muitas ramificações e diferentes possibilidades. Já a busca com informação utiliza um dado que vai além da definição do problema, atribuindo este fator a uma heurística. Este trabalho faz uma análise de diferentes algoritmos de busca para resolver o clássico problema do quebra cabeça de oito peças e discute questões como otimalidade, completeza e gastos de memória e tempo além de dominância entre heurísticas.

II. Algoritmos De Busca

A. Busca em Largura

A busca em largura é um tipo de busca cega, ou seja, só utiliza a informação dada no início do problema. Esta estratégia de busca usa o princípio de uma fila para percorrer a árvore de possibilidades do problema.

Esta busca expande a árvore de busca por níveis, visitando cada filho de uma camada antes de seguir para a próxima. Portanto em um algoritmo de busca em largura, primeiramente são visitados todos os nós com a profundidade 1, depois todos os nós com profundidade 2 e assim por diante.

A cada nó visitado, seus nós filhos são armazenados em uma fila. Assim se faz necessário salvar o estado no momento em que um nó é visitado. Isso é um problema pois uma árvore com muitas ramificações demanda muita memória.

Assim um pseudo-código para representar a busca em largura seria da seguinte forma:

```
Busca em Largura
fila F;
enquanto(!F.vazia())
{
    s = F.front();
    para todo v vizinho de s faça
    {
        F.push(v);
    }
    F.pop();
}
```

B. Busca em Profundidade

A busca em profundidade é similar à busca em largura quanto ao seu gênero: busca sem informação. Entretanto sua estratégia de busca, diferentemente da busca em largura, utiliza uma pilha para armazenar os nós visitados. Portanto, as busca em profundidade, como o próprio nome sugere, se preocupa em pesquisar o nós mais profundo o possível.

Quando se chega em um nó que já é conhecido, basta desempilhar (voltar um nível de profundidade) e escolher outro caminho para expandir.

Esta estratégia economiza memória pois não precisamos nos preocupar em guardar estados. Entretanto esta busca apresenta problemas ao escolher ramificações muito longas que não levam a solução.

Um pseudo-código para representar a busca em profundidade seria da seguinte maneira:

```
Busca Profundidade
pilha P;
enquanto(!P.vazia())
{
    s = P.top();
    se há algum vizinho v de s não visitado
    {
        P.push(v);
    }
    se não
    {
        P.pop();
    }
```

C. *Busca Gulosa*

A busca gulosa abre um novo gênero de buscas: busca com informação. Este conjunto de buscas utiliza um dado que vai além da definição do próprio problema para definir a estratégia de busca.

A busca com informação utiliza uma função heurística para estimar qual o melhor caminho a se seguir. No caso de uma busca gulosa, o próximo caminho a ser escolhido sempre será aquele que aparenta levar a melhor solução.

Logo, a busca gulosa sempre busca os nós que localmente são a melhor escolha. Isto é uma vantagem em relação às buscas apresentadas anteriormentes pois não precisa utilizar tanta memória e descarta caminhos que parecem infrutíferos.

Entretanto, utilizar esta estratégia de busca pode causar falsos inícios: um caminho que parecia bom pode ficar ruim ao passar das iterações.

III. Quebra Cabeça de Oito Peças

O problema que os algoritmos deveriam resolver é o quebra cabeça de oito peças. Este clássico problema tráz um tabuleiro 3x3 com oito peças e um espaço em branco. As peças só podem ser movidas para o espaço em branco variando as possibilidades de movimento de duas a quatro peças.

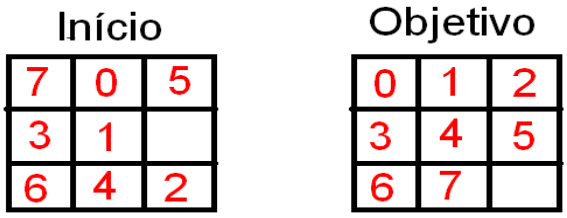

Fig. 1. Exemplo de um estado inicial e o estado objetivo do quebra cabeça.

Este problema foi representado por um grafo de 9 vértices a partir de uma matriz de pesos onde o peso de cada vértice representa a peça do tabuleiro. O valor 8 representa o espaço em branco.

Portanto um movimento no tabuleiro era resultante da troca de valores do vértice que continha a peça em branco (valor 8) com qualquer outro vértice adjacente.

Cada estado do tabuleiro também poderia ser representado por um ID único que era obtido a partir da distribuição das peças no tabuleiro. Por exemplo, o estado objetivo teria seu ID representado pela *string* “012345678” pois esta é a ordem da distribuição de peças no tabuleiro.

IV. Análise Dos Algoritmos De Busca

Esta seção analisa o desempenho de cinco algoritmos de busca para achar a solução do quebra-cabeça de oito peças: busca em largura, busca em profundidade, busca gulosa com: f(h1) = número de peças nas posições erradas; f(h2) = distância das peças das posições corretas; f(h3) = f(h2) + número de conflitos lineares horinzontas. (h3 será explicado posteriormente). Os algoritmos rodaram 100 vezes com um tempo limite de 90 segundos por rodada.

A. *Completeza*

A completeza define se o algoritmo é capaz de sempre achar uma solução para o problema ou não. Neste aspecto, nenhum dos algoritmos de busca se mostrou completo. Ou seja, nenhum obteve a taxa de sucesso de 100%.

Tabela I. Taxa de Sucesso Dos Algoritmos

Algoritmo Utilizado	***Porcentagem de Sucesso***
Busca em Profundidade	3%
Busca em Largura	6%
Busca Gulosa	50%

Como pode ser observado no gráfico, a busca gulosa foi a que se mostrou melhor na resolução do problema do quebra-cabeça de oito peças. Como a busca em profundidade e a busca em largura não utilizam nenhuma informação, é muito fácil seguir por caminhos infrutíferos na árvore de busca. No caso da busca em profundidade, este problema é maior pois a busca pode seguir pelo nível mais fundo de um caminho que não leve a solução. Apesar disso, os espaços de busca no quebra cabeça de oito peças são finitos e como o algoritmo básico foi adaptado para não visitor estados repetidos, a busca em profundidade poderia se mostrar completa.

A busca em largura tem uma estratégia mais inteligente para percorrer os caminhos pois analisa camadas. Entretanto, isso faz com que a busca em largura expanda mais caminhos que a busca em profundidade ficando mais lenta. Neste problema a busca em largura também seria um algoritmo completo pois o fator de ramificação do problema do quebra

cabeça de oito peças se limita ao máximo 4 estados (quando o espaço em branco está no meio).

Porém, como o algoritmo foi limitado em 90 segundos, é difíicil para tanto para a busca em largura quanto a busca em profundidade encontrar uma solução em pouco tempo e desta forma, não foi possível constatar sua completeza.

A busca gulosa se mostrou bem melhor que as outras duas estratégias. Entretanto, diversos *workarounds* foram necessários para otimizar a busca gulosa.

O primeiro problema foi em relação a peça em branco. Considerando ela na contagem da heurística, o algoritmo travava pois se o tabuleiro começasse com a peça em branco no lugar correto, não teria meios de melhorar a heurística. Portanto, a peça em branco foi desconsiderada.

Porém, isto não mudou o problema de chegar em pontos que a heurística piora. As bordas do tabuleiro sempre se tornavam um problema quando a peça em branco estava em um canto e as peças adjacentes estavam na posição correta. Para as heurísticas *h1* e *h2* o único modo era piorá-las. O Segundo *workaround* proposto foi ignorar estes casos e escolher um caminho que piora a heurística.

O terceiro problema foi com o vício em *loops*. Muitas vezes a gulosa caia em ciclos, travando o algoritmo. A solução encontrada foi bloquear os últimos estados visitados. Assim a busca era obrigada a seguir um caminho diferente, saindo do ciclo. Isto combinado a uma escolha aleatoria quando os caminhos tinham o mesmo valor de heurística possibilitou a execução das buscas gulosas.

B. Otimalidade

A otimalidade para este problema, se trata de achar a melhor solução no menor número de movimentos. A tabela a seguir mostra a melhor solução encontrada para os algoritmos, dividindo a busca gulosa por heurística utilizada.

TABELA II. TAXA DE OTIMALIDADE

Algoritmo Utilizado	***Número de movimentos necessários***
Busca em Profundidade	2.545
Busca em Largura	20.970
Busca Gulosa h1	1.105
Busca Gulosa h2	71
Busca Gulosa h3	46

A busca gulosa h3 se mostrou superior aos outros tipos de busca. Isto mostra que quanto mais informações são agregadas ao problema, melhor o algoritmo escolhe qual peça escolher. As buscas cegas não tem qualquer tipo de informação do problema, portanto, chegar em um resultado depende exclusivamente da composição do tabuleiro. Se a Busca em Profundidade escolher descer por um nível em que se encontra a solução, esta terminará sem utilizar muitos movimentos. Semelhante é para as camadas da Busca em Largura. Entretanto como a Busca em Largura visita níveis, o algoritmo adiciona de dois a quatro movimentos na fila de execução sendo que grande parte deles resultará em caminhos infrutíferos.

C. Tempo

O tempo foi calculado durante a execução de cada algoritmo e limitado a 90 segundos para encontrar uma solução. A tabela a seguir mostra o tempo de execução dos algoritmos:

TABELA III. TEMPO DE EXECUÇÃO

Algoritmo Utilizado	***Tempo de Execução (segundos)***
Busca em Profundidade	10,275
Busca em Largua	23,602
Busca Gulosa h1	7,274
Busca Gulosa h2	0,051
Busca Gulosa h3	0,025

O tempo tem grande relação com o número de movimentos feitos durante a execução do algoritmo. Por isso, algoritmos que executam mais passos tendem a demorar mais para encontrar uma solução. Desta maneira, o tempo se mostra proporcional a quantidade de movimentos realizados.

D. Memória

A memória somente se torna um problema para a busca em largura pois a mesma precisa armazenar todos os estados a medida que se expande a árvore.

Para os outros algoritmos, somente interessa o estado atual do tabuleiro, sendo necessário apenas armazenar uma lista dos IDs já visitados a fim de evitar ciclos e caminhos viciosos.

V. ANÁLISE DAS HEURÍSTICAS

A. Domínio de H2 sobre H1

O domínio de uma heurística tem relação com a quantidade de estados expandidos por cada heurística. Uma heurística domina outra quando hi(n) >= hj(n). Ou seja, o *score* sempre é maior ou igual. Isso implica que para uma heurística ser melhor que outra esta deve englobar mais informações acerca do problema. No caso, h2 domina h1 porque a distância das peças já engloba o número de peças de erradas e acrescenta o quão longe uma peça está do seu objetivo. Isso significa que o algoritmo que usa h2 expande um subconjunto promissor de h1, não precisando expandir caminhos que não sejam bons.

B. Criação de H3

A heurística h3 proposta para este trabalho utiliza um conceito já conhecido. O **Conflito Linear** pode ser definido formalmente da seguinte maneira: *Dado duas peças Pi e Pj, estas peças formam um conflito linear se ambas estão na*

mesma linha de sua posição correta, Pj está a esquerda de Pi e a posição correta de Pi está à sua esquerda e a posição correta de Pj está a sua direita. Existe conflito linear vertical e horizontal mas apenas foi usado o conflito horizontal. Esta informação extra diz que duas peças precisam "*dar a volta*" para alcançar seu objetivo. Portanto, embora uma distância menor pareça bom em primeiro momento, colocar duas peças em conflito linear gera mais movimentos futuramente.

Portanto, h3 seria h2 + o número de pares em conflito linear.

$$f(h3) = f(h2) + CLH$$

Esta heurística é admissível pois peças em conflito linear obrigatoriamente tem que se contornar e não superestima o valor real. A seguir, um gráfico compara o desempenho das três heurísticas em rodadas que obtiveram sucesso (escala em log base 10):

No gráfico fica claro ver a dominância entre as heurísticas. A heurística h2 domina h1 e ambas são dominadas por h3 que obteve melhores soluções durante suas execuções de sucesso.

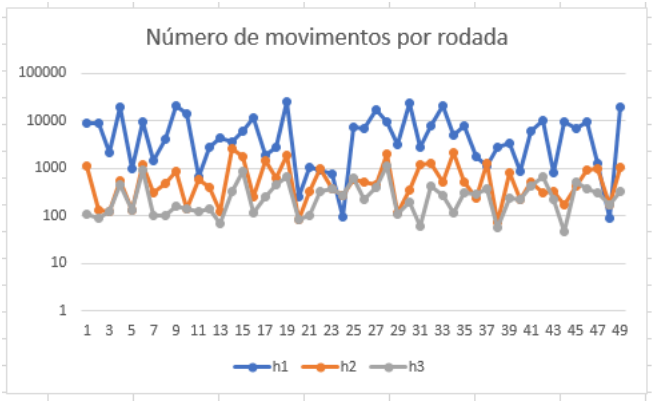

Fig. 2. Gráfico comparative entre o desemepnho das três heurísticas.

VI. Conclusão

Busca cegas podem ser muito úteis pela sua simplicidade na implementação. Entretanto não são eficientes para resolver problemas mais complexos como o quebra-cabeça de oito peças. O grande número de ramificações da árvore faz com que as buscas cegas percam muito tempo processando caminhos infrutíferos.

A busca com informação pode ser uma boa alternativa nesse caso. Entretanto buscas gulosas nem sempre são eficientes. O principal ponto é escolher uma boa função heurística para evitar travamentos e cair em ciclos. O grande problema dos algoritmos gulosos foi criar *workarounds* para evitar que o algoritmo caísse em ciclos e caminhos viciosos.

Portanto, este trabalho analisou que quanto mais se enriquece as informações para uma busca, melhores são os resultados, tanto em completeza quanto otimalidade e tempo de execução.

Referências

[1] RUSSELL, Stuart; NORVIG, Peter. **Inteligência artificial**. Elsevier, 2004.J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892.

[2] CULBERSON, Joseph; SCHAEFFER, Jonathan. **Efficiently searching the 15-puzzle.** 1994.