Métodos de Busca Cega

Os métodos de busca estão baseados na ideia de que se tem um estado inicial e através da aplicação de uma série de ações o estado vai evoluindo até chegar ao estado final que se constitui naquela situação em que o objetivo é atendido; isto pode ser feito de maneira a maximizar uma medida de desempenho.

A resolução de problemas através de busca, também chamada de resolução de problemas baseada na abordagem por espaço de estados, está baseada no conceito de *estado*. O estado corresponde a um momento do problema e é identificado pela especificação do conjunto de valores de cada uma das variáveis relevantes ao problema em questão. O problema passa de um estado a outro (correspondendo à alteração do valor de ao menos uma variável que o caracteriza) através da execução de uma ação. Desta forma, uma sequência de ações faz com que o problema passe de um estado inicial a um estado final.

As diversas fases para a construção de um Sistema Inteligente seguindo a abordagem de espaço de estados são as seguintes:

- *Formulação do objetivo*: Neste momento devem ser especificados os objetivos a serem atingidos para que se possa aceitar a solução, os quais por sua vez determinam qual é a medida de desempenho a ser utilizada;
- *Formulação do problema*: A fim de poder determinar a sequência adequada de ações para se atingir o objetivo, é necessário conhecer todos os elementos que especificam de maneira única o problema e suas características. Uma das formas de controlar o nível de complexidade (e, por conseguinte, da qualidade da solução) é através do processo de decisão em relação a quais ações e estados devem ser considerados, ou seja, através da especificação do nível de abstração;
- *Definição do Algoritmo de Busca*: Em função do problema, dos objetivos, e das características desejadas da solução, define-se qual é o conjunto de algoritmos que podem ser utilizados. Em havendo mais de um algoritmo passível de utilização, busca-se aquele que pode apresentar os menores requisitos de consumo de memória, de menor tempo de processamento, ou de uma combinação de ambas;
- *Obtenção da Solução*: Tendo-se todas as definições, executa-se o algoritmo escolhido sobre o problema para a obtenção da solução, aqui entendida como a sequência de passos (ações) que conduz o problema do seu estado inicial ao estado desejado;
- *Execução*: A partir da definição da sequência de ações que permitem sair do estado inicial e chegar ao estado final, obedecendo aos objetivos estipulados, executam-se estes passos no problema físico para resolvê-lo.

A fim de se ter uma correta especificação do problema segundo a abordagem de espaço de estados, ou métodos de busca, é necessário identificar os seguintes elementos:

- *Estado inicial*: Indica o estado em que se encontra o sistema no momento inicial, isto é, o ponto de partida. A partir deste, aplica-se a sequência de ações para que o estado final desejado seja alcançado;
- *Operadores* (também chamados de *função sucessor S*): Representa o conjunto de ações possíveis de serem executadas de maneira geral. A cada momento um subconjunto destas ações é possível. Se e_0 for o estado inicial e e_f for o estado final almejado, deseja-se obter a sequência de ações $e_0 \xrightarrow{a_1} e_1$, $e_1 \xrightarrow{a_2} e_2$, ..., $e_i \xrightarrow{a_i} e_f$. Apesar de existir um subconjunto de ações possíveis a cada momento, somente uma deve ser escolhida para ser executada a cada instante. A política de escolha da próxima ação a ser executada é definida pela estratégia de busca, a qual caracteriza cada um dos métodos de busca;
- *Espaço de estados*: Dado o estado inicial, o espaço de estados representa o conjunto de todos os estados que podem ser alcançados a partir do mesmo, sendo representado como uma árvore de busca. Apesar disto, os métodos de busca não têm como objetivo construir tal árvore; pelo contrário, um bom método de busca chega ao estado objetivo explorando ao mínimo tal árvore. Define-se um caminho no espaço de estados como sendo uma sequência de ações que conduz de um estado a outro;
- *Teste do objetivo*: É o teste aplicado pelo sistema para determinar se a descrição do estado atual corresponde a um estado objetivo, tendo em vista que de forma genérica pode-se não ser capaz de reconhecer o estado automaticamente. Se o teste aplicado ao estado atual é positivo, considera-se que o estado final desejado foi atingido; caso contrário, não;
- *Custo do caminho g*: Vários algoritmos buscam não apenas um caminho entre os estados inicial e final, como também procuram um caminho que otimize algum critério. Para tanto, faz-se necessário a determinação do custo entre os estados inicial e final e, portanto, leva-se em conta o custo de cada ação a ser tomada. De maneira geral, o custo total é dado por uma função que assinala um custo a um caminho entre dois nós.

Um ponto a ser ressaltado é que para um mesmo problema pode-se escolher diferentes estados e ações para a sua representação. A escolha dos estados e ações mais indicados está ligado à abstração em relação ao problema, ou seja, a que nível de detalhe é necessário trabalhar. Caso haja diferentes representações possíveis em um nível adequado de abstração, normalmente opta-se pela representação que conduz ao menor número de estados e ações, pois isto faz com que o espaço de busca seja menor.

Para a geração da sequência de ações na busca pela solução de problemas, executa-se sequência de passos. A cada momento, deve-se buscar estados e para isto, aplicam-se os operadores ao estado atual, gerando novos estados. Isto é chamado de expansão dos estados. A escolha do próximo estado a ser expandido é realizada pela estratégia de busca. No decorrer do processo de busca constrói-se

Algoritmo Geral de Busca

função BUSCA_GERAL (*problema, estratégia*) **retorna** {uma solução ou falha}

inicializar a árvore de busca usando o estado inicial do *problema*;

iterar

se não há candidatos a expansão

então retorna falha;

escolher um nó folha para expansão segundo a *estratégia*;

se o nó contém um estado objetivo

então retorna a solução correspondente;

senão expandir o nó e adicionar os nós resultantes à árvore de busca;

fim

Algoritmo 1 – Descrição informal do Algoritmo Geral de Busca.

uma árvore de busca. Este processo pode ser informalmente representado de forma genérica pelo Algoritmo Geral de Busca, indicado no algoritmo 1.

Assumindo-se que a implementação ocorra usando uma estrutura do tipo fila, tem-se o Algoritmo Geral de Busca conforme indicado no Algoritmo 2.

Algoritmo Geral de Busca

função BUSCA_GERAL (*problema, ENFILEIRAR_FN*) **retorna** {uma solução ou falha}

nós $\rightarrow$ CRIAR-FILA (CRIAR-NÓ (ESTADO-INICIAL [*problema*]));

iterar

se *nós* está vazio

então retorna falha;

nó $\leftarrow$ REMOVE_PRIMEIRO (*nós*);

se TESTE-OBJETIVO [*problema*] aplicado a ESTADO (*nó*) tem sucesso

então retorna *nó*;

senão ENFILEIRAR-FN (*nós*, EXPANDIR (*nó*, OPERADORES [*problema*]));

fim

Algoritmo 2 – Algoritmo Geral de Busca baseado em estrutura do tipo fila.

Existem basicamente duas grandes estratégias de busca: a *não informada, ou cega*, e a busca *informada, ou heurística*. A busca não-informada ou busca cega não tem informações sobre o número de passos ou custo do caminho do estado atual ao

objetivo, adotando sempre uma forma padrão para a escolha do próximo estado. Já a busca informada ou heurística dispõe de informações que auxiliam a determinar o *provável* melhor caminho entre o estado atual e o estado objetivo, e utilizam esta informação (ou "dica", já que ela não garante a melhor resposta sempre).

As estratégias de busca normalmente devem ser avaliadas em função de algumas propriedades, sendo as principais:

- *Completude*: Indica se a estratégia encontrará a solução sempre que ela existir;
- *Complexidade temporal*: Fornece uma indicação da demora para encontrar a solução;
- *Complexidade de espaço*: Indica a quantidade de memória necessária para a busca;
- *Otimalidade*: Significa que a estratégia encontra a melhor solução quando existem várias.

A seguir temos algumas das estratégias cegas mais comuns. Sua vantagem está na simplicidade de implementação e no fato de que não são necessárias muitas modificações entre a implementação para solução de diferentes problemas. Sua desvantagem em relação às estratégias heurísticas (ou informadas) está em seu menor poder, ou seja, em geral elas necessitam de um número de iterações bem maior para encontrar a solução.

BUSCA PRIMEIRO EM LARGURA

Nesta estratégia, todos os nós na profundidade d da árvore de busca são expandidos antes dos nós em profundidade d+1. Sua implementação, baseada no Algoritmo de Busca Geral, está indicada no Algoritmo 3. A figura 1 dá uma indicação de como ocorre o desenvolvimento da busca.

Algoritmo de Busca em Largura

função BUSCA_PRIMEIRO_EM_LARGURA (*problema*) **retorna** {uma solução ou falha}

    retorna BUSCA_GERAL (*problema*, ENFILEIRAR_NO_FIM);

fim

Algoritmo 2 – Algoritmo de Busca em Largura.

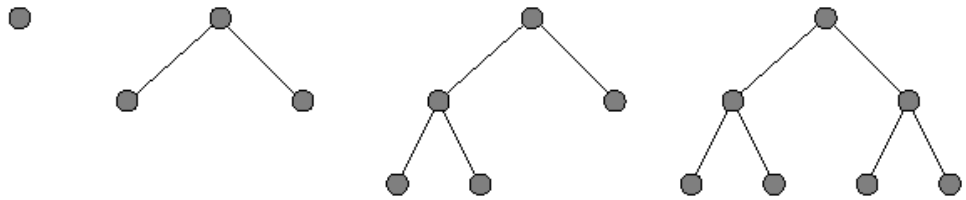

Figura 1 – Desenvolvimento da Busca em Largura.

BUSCA DE CUSTO UNIFORME

Este tipo de busca é uma variante da busca primeiro em largura. Ela tem a característica de modificar a busca primeiro em largura através da expansão do nó de menor custo na maior profundidade, mesmo que já se tenha atingido o estado objetivo. Ela encontra a melhor solução se o custo do caminho nunca decrescer ao longo do caminho:

$$g(\text{SUCESSOR}(n)) \geq g(n) \quad (1)$$

A figura 2 apresenta o comportamento deste tipo de busca.

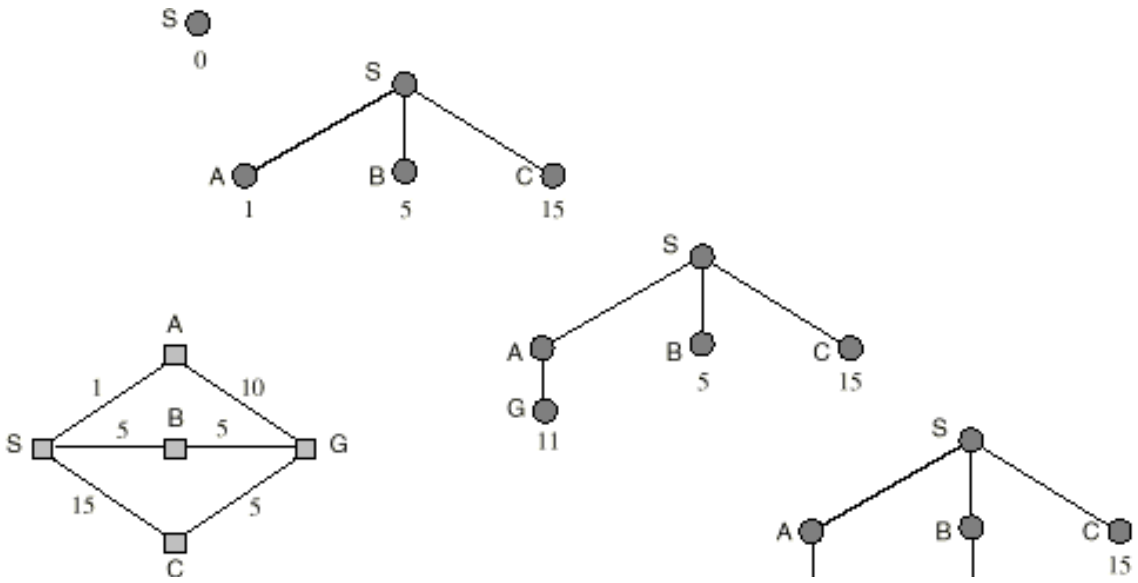

Figura 2 – Desenvolvimento usando a Busca de Custo Uniforme.

BUSCA PRIMEIRO EM PROFUNDIDADE

Este tipo de busca tem como característica expandir sempre o nó no nível mais profundo da árvore de busca, ou seja, um ramo mais à direita somente será explorado se o ramo imediatamente à sua esquerda foi exaustivamente testado e não se encontrou a solução. Desta forma, somente quando a busca atinge um nó não-objetivo sem expansão é que ocorre o retorno para um nível superior. Em função disto, a busca to tipo primeiro em profundidade não é completa (e, portanto, também não é ótima). Um exemplo de situação em que o problema tem solução, mas que ainda assim este tipo de abordagem não é capaz de encontrar a solução consiste em uma árvore na qual o primeiro ramo à esquerda (o primeiro a ser testado) não apresenta entre seus nós o nó objetivo, mas que tem profundidade infinita. Este tipo de problema pode acontecer quando houver uma ação que desfaz o que outra ação realizou, por exemplo. A implementação deste método de busca, baseada no Algoritmo de Busca Geral, está indicada no Algoritmo 4.

A figura 3 apresenta o comportamento do método de busca em profundidade.

Algoritmo de Busca em Profundidade
função BUSCA_PRIMEIRO_EM_PROFUNDIDADE (*problema*) **retorna** {uma solução ou falha} **retorna** BUSCA_GERAL (*problema*, ENFILEIRAR_NA_FRENTE); **fim**

Algoritmo 4 – Algoritmo de Busca em Profundidade.

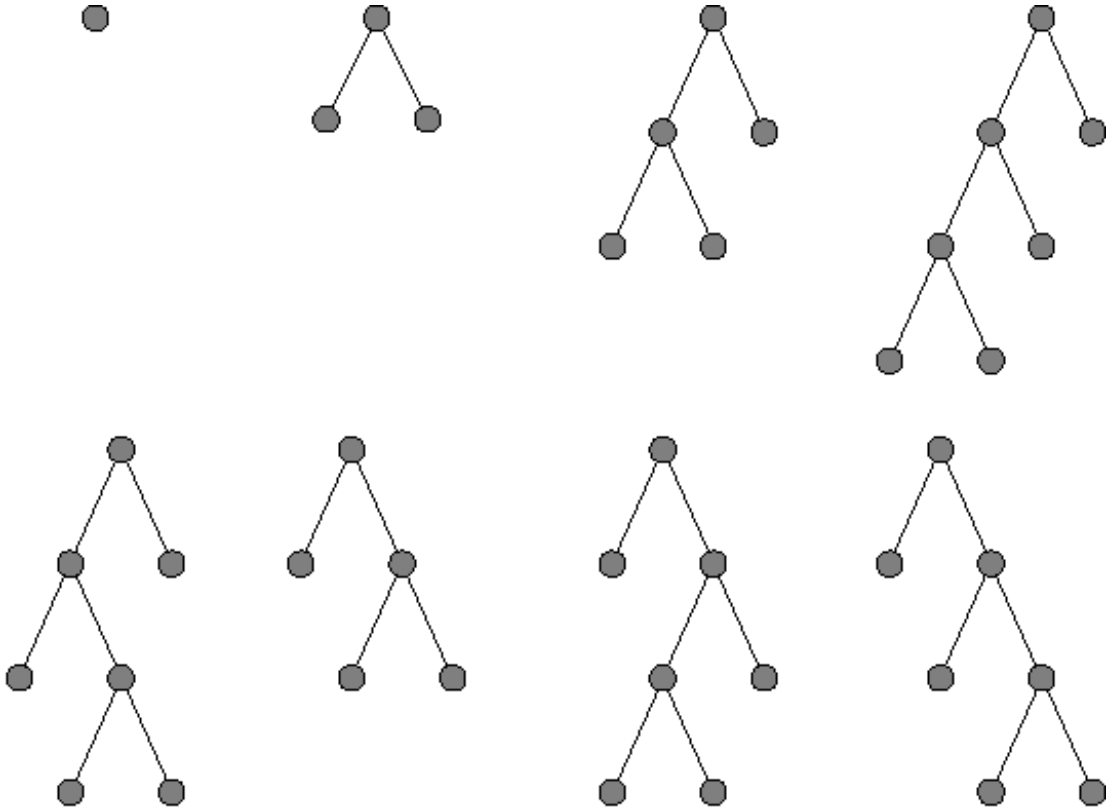

Figura 3 – Desenvolvimento da busca usando a abordagem em Profundidade.

BUSCA COM PROFUNDIDADE LIMITADA

Coloca um limite na máxima profundidade de um caminho. O limite pode ser implementado em um algoritmo especial de busca com profundidade limitada ou usando o algoritmo de busca geral com operadores que mantenham um registro da profundidade. Este algoritmo encontra a solução caso a mesma em um nível de profundidade menor ou igual ao limite estipulado. Quando se conhecer a priori o número aproximado de ações a serem tomadas para se obter a solução do problema, este método pode ser considerado como completo e ótimo. Em situações práticas, entretanto, não se conhece tal nível, o que implica que não se pode garantir que o algoritmo encontrará a solução.

BUSCA COM APROFUNDAMENTO ITERATIVO

Este algoritmo provê uma alteração no algoritmo de busca em profundidade de tal forma que ele encontra a solução (completo) e garante-se a melhor solução em termos do número de passos (ótimo). Para tanto, ele faz uso do método de busca com profundidade limitada, chamando-o com várias profundidades limites: 0, 1, 2, …, n, até encontrar a solução. Sua descrição encontra-se no Algoritmo 05. Na figura 4 tem-se uma visão do seu funcionamento em um problema.

Algoritmo de Busca com Aprofundamento Iterativo

função BUSCA-APROFUNDAMENTO-ITERATIVO (*problema*) **retorna** uma sequência solução

inputs: *problema*, um problema

for *profundidade* 0 **to** ∞ **do**

se BUSCA-PROFUNDIDADE-LIMITADA (*problema, profundidade*) tem sucesso

then return resultado

end

return falha

Algoritmo 5 – Algoritmo de Busca com Aprofundamento Iterativo.

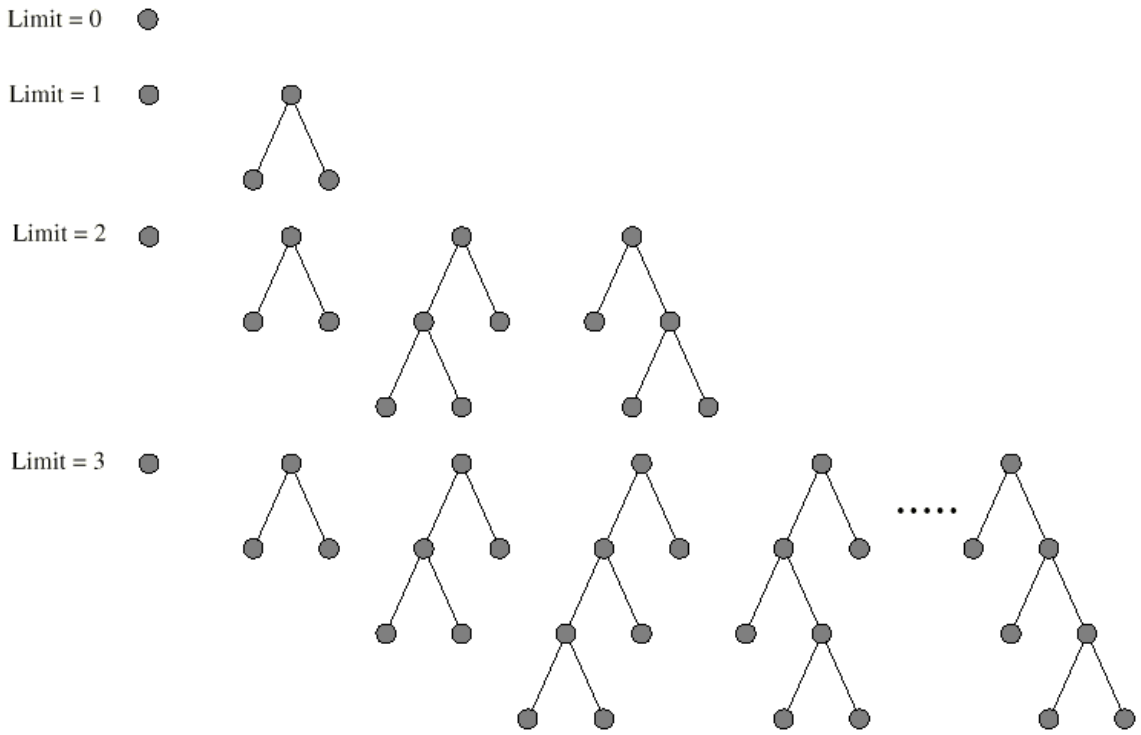

Figura 4 – Desenvolvimento de busca usando a abordagem Aprofundamento Iterativo.

Este método de busca além de ter propriedades interessantes, ou seja, ser completo e ótimo, também tem baixos requisitos de memória, tal qual o algoritmo de busca em profundidade. No momento da implementação deve ser tomado cuidado para que o algoritmo chame o procedimento de busca com profundidade limitada como o novo valor limite, mas sempre partindo de zero, ou seja, a cada novo nível a busca deve ser reiniciada, para evitar sobrecarregar a memória.

MÉTODO DE BUSCA BIDIRECIONAL

Uma ideia interessante para tentar obter a solução de um problema usando o princípio dos métodos de busca consiste no uso da busca bidirecional. Neste caso, parte-se do princípio de que se tem um estado inicial e um estado final, usando-se um algoritmo escolhido para realizar a busca. Simultaneamente, realiza-se a busca com este algoritmo (ou outro algoritmo de busca) do estado final para o estado inicial. Se existir solução do problema, em algum momento os caminhos traçados pelos algoritmos irão se encontrar, como indicado na figura 05.

Apesar de parecer bastante intuitivo, este método não é muito prático em problemas reais, pois para poder determinar que os dois caminhos têm um ponto em comum e tem-se uma solução (ligação entre o estado inicial e o estado inicial) é necessário testar se cada novo estado é igual a algum dos estados pré-existentes. Para isto ser possível é preciso armazenar todos os estados desenvolvidos, o que pode exigir grandes requisitos de memória.

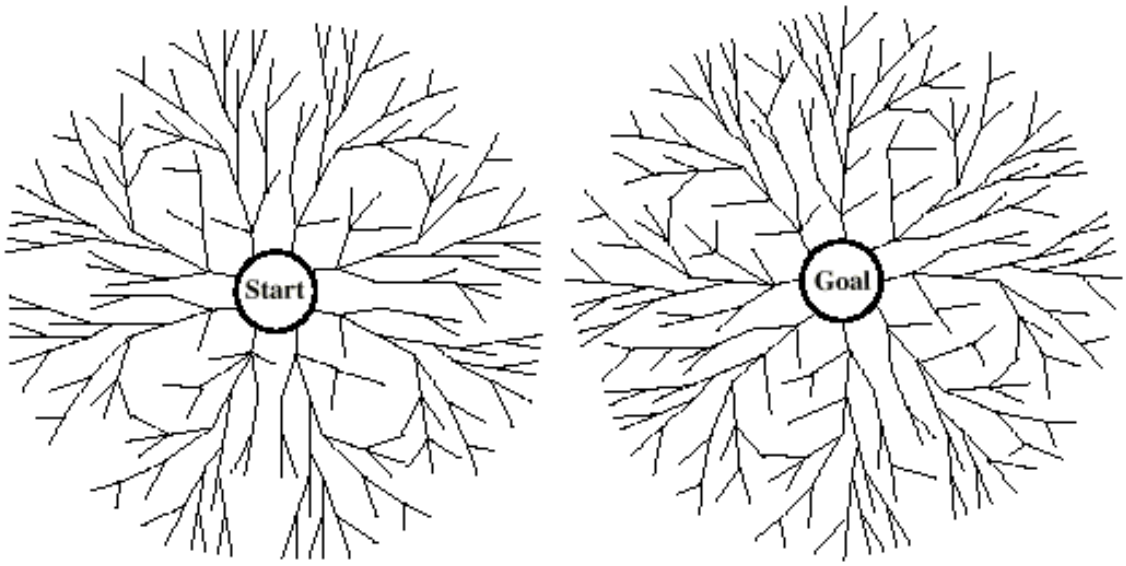

Figura 5 – Desenvolvimento de busca segundo o princípio bidirecional.

COMPARAÇÃO ENTRE OS MÉTODOS DE BUSCA

Na tabela 01 observa-se uma comparação dos métodos de busca apresentados em relação às características de interesse.

Tabela 1 – Comparação entre os Métodos de Busca.

Método de Busca	Característica		
	Completo?	Ótimo?	Uso de memória
Largura	Sim	Sim	Elevado
Custo Uniforme	Sim	Sim	Elevado
Profundidade	Não	Não	Baixo
Prof. Limitada	Não	Não	Baixo
Aprofundamento Iterativo	Sim	Sim	Baixo
Bidirecional	Sim	Sim	Elevado

REPETIÇÃO DE ESTADOS

Um problema comum quando se usam métodos de busca cega é a possibilidade de criação de ciclos. Isto pode acontecer em situações nas quais uma determinada ação (ou sequência de ações) pode desfazer as mudanças causadas por outra ação (ou por outra sequência de ações). Se isto acontece, em lugar de uma árvore, tem-se o surgimento de um grafo. Esta possibilidade é problemática por vários motivos:

1. O algoritmo pode permanecer indefinidamente dentro do ciclo e nunca encontrar a solução, mesmo que ela exista;
2. Em função do ciclo, há um aumento no tempo para se obter a solução;
3. O consumo de memória é maior.

É desejável evitar que os métodos entrem em um dos ciclos (geração de grafos). Para tanto, existem três distintas possibilidades:

1. Não gerar qualquer estado que já tenha sido criado: Neste caso, evita-se de forma absoluta a possibilidade de permanência dentro de um ciclo. Para isto, entretanto, é necessário comparar cada novo estado com todos os estados já criados até o momento, ou seja, é precisar armazenar todos os estados até o final do processo de busca, o que não é viável, em geral, pois há um consumo excessivo de memória;
2. Não gerar um novo estado igual a um dos últimos estados gerados: Para diminuir o uso de memória, armazenam-se os últimos n estados gerados e compara-se o novo estado com estes últimos estados. Em caso de repetição de estados, o novo estado é considerado um ponto final e elimina-se o mesmo da lista de estados candidatos a caminho para a solução. Este procedimento garante que se evitam ciclos caso o tamanho deles seja menor ou igual que o número de últimos estados armazenados. Em situações reais é muito difícil ter uma ideia do tamanho dos ciclos, motivo pelo qual esta estratégia não é muito comum;
3. Não continuar com um filho que seja igual ao pai: Apesar de muito simples, esta abordagem pode ser bastante útil, tendo em vista que a maior incidência de ciclos é de pequena dimensão (ciclos de poucos estados).

Métodos de Busca Heurística

O protótipo dos métodos de busca heurística é chamado de *Busca do Melhor Primeiro*, sendo apresentado no Algoritmo 6. Ele utiliza uma medida estimada do custo da solução e tenta sua minimização. Esta medida deve incorporar uma estimativa do custo do caminho de um determinado estado ao estado objetivo mais próximo. Existem basicamente duas abordagens: a) Expandir o nó mais próximo a um objetivo; ou b) Expandir o nó no caminho da solução de menor custo. Estes dois métodos são chamados de Método de Busca Gulosa e Método de Busca A*, respectivamente.

Algoritmo de Busca do Melhor Primeiro

função BUSCA_MELHOR_PRIMEIRO (*problema, AVAL-N*) **retorna** {uma solução ou falha}

entradas: *problema*, um problema

Aval-Fn, uma função de avaliação

Ordenar-F, uma função que ordena os nós segundo Aval-Fn

retorna BUSCA_GERAL (*problema*, *Ordenar-Fn*);

fim

Algoritmo 6 – Algoritmo de Busca do Melhor Primeiro.

MÉTODO DE BUSCA GULOSA (“GREEDY SEARCH”)

Este método representa a implementação da estratégia mais simples de busca do melhor primeiro, estando descrito no Algoritmo 7. Ela consiste em minimizar o custo estimado do nó considerado até o objetivo, ou seja,

$h(n) \equiv$ custo estimado mais “barato” do caminho entre o estado *n* e o estado objetivo

Algoritmo de Busca Gulosa

função BUSCA_GULOSA (*problema*) **retorna** {uma sequência solução ou falha}

retorna BUSCA_MELHOR_PRIMEIRO (*problema*, *h(n)*);

fim

Algoritmo 7 – Algoritmo Guloso (“gready search”).

MÉTODO DE BUSCA A*

O método de busca gulosa não é ótimo e nem mesmo completo. Relembrando o método de busca de custo uniforme (que usa *g(n)*, ou seja, a estimativa do custo desde o ponto inicial até o estado sendo considerado), ele representa uma alternativa que é completa e ótima, mas que tem desempenho ineficiente.

É possível combinar os métodos de busca gulosa e de custo uniforme, usando uma função de custo *f(n)* como sendo o custo estimado total da solução de menor custo desde o estado inicial até a solução passando pelo nó *n* considerado:

$$f(n) = g(n) + h(n)$$

O Algoritmo 8 descreve o método de busca A*.

Algoritmo de Busca A*
função BUSCA_A* (*problema*) **retorna** {uma sequência solução ou falha} **retorna** BUSCA_MELHOR_PRIMEIRO (*problema*, *f(n)*); **fim**
Algoritmo 8 – Algoritmo de Busca A*.

Considerações:

- Uma heurística é chamada de *admissível* se ela nunca superestima o custo de atingir o objetivo, ou seja, ela é otimista;
- A estratégia A* é completa e ótima se na função *f* de avaliação a função heurística *h* for admissível.

MÉTODOS GERAIS – MÉTODOS DE MELHORA ITERATIVA

Nos algoritmos de melhora iterativa, inicia-se com uma configuração completa (que representa o estado inicial) e realizam-se modificações nesta configuração (estado atual) para melhorar a qualidade até se atingir o estado objetivo, que representa a situação em que o problema esteja resolvido. Nestes métodos não se tem um rastro das ações realizadas para se atingir o estado objetivo; o objetivo é encontrar somente o estado que representa a solução do problema. Há basicamente dois tipos: subida da montanha (ou descida do gradiente), que tenta melhoras graduais no estado atual e o recozimento simulado.

ALGORITMO DE SUBIDA DA MONTANHA (“HILL CLIMBING”)

Este método verifica quais estados podem ser atingidos a partir do atual e escolhe um melhor; caso não haja nenhum melhor ele fornece o atual como solução. O Algoritmo 9 apresenta a especificação deste algoritmo. Embora ele seja de fácil implementação e bastante usado, ele apresenta alguns problemas quando se tem máximos locais, planícies e cadeias de montanhas. Para melhorar o desempenho muitas vezes usa-se o algoritmo de subida da montanha com reinícios em pontos aleatórios.

Algoritmo de Subida da Montanha (“Hill Climbing”)

função SUBIDA_DA_MONTANHA (*problema*) **retorna** {um estado solução}

entradas: *problema*, um problema

estático: *atual*, um nó

próximo, um nó

atual ← CRIAR_NÓ(ESTADO_INICIAL[*problema*])

loop do

próximo ← um sucessor de maior valor de *atual*

if VALOR[*próximo*] < VALOR[*atual*] **then return** *atual*

atual ← *próximo*

fim

Algoritmo 9 – Algoritmo de Subida da Montanha.

ALGORITMO DE RECOZIMENTO SIMULADO (“SIMULATED ANNEALING”)

Seu funcionamento está apresentado no Algoritmo 10. Ao contrário do algoritmo de subida da montanha, o algoritmo de recozimento simulado tenta modificações abruptas, às vezes aceitando que o estado atual seja prejudicado esperando que resultados futuros sejam melhores apesar disto.

BUSCA EM FEIXE (“BEAM SEARCH”)

Este algoritmo trabalha com *k* estados simultaneamente em lugar apenas um só. Ele começa com *k* estados selecionados aleatoriamente e, a cada iteração, todos os sucessores de todos os *k* estados são gerados. Se qualquer um dos estados é o objetivo então o algoritmo para e o apresenta; caso contrário ele seleciona os k melhores da lista completa e repete o procedimento.

Algoritmo de Recozimento Simulado ("Simulated Annealiing")

função RECOZIMENTO_SIMULADO (*problema*, *agenda*) **retorna** {um estado solução }

entradas: *problema*, um problema

agenda, um mapeamento do tempo para "temperatura"

estático: *atual*, um nó

próximo, um nó

T, uma "temperatura" controlando a possibilidade de passos para trás

atual ← CRIAR_NÓ(ESTADO_INICIAL[*problema*])

for *t* = 1 **to** ∞ **do**

T ← *agenda*[*t*]

If *T* = 0 **then return** *atual*

próximo ← um sucessor de *atual* aleatoriamente escolhido

ΔE ← VALOR[*próximo*] – VALOR[*atual*]

if ΔE < 0 **then** *atual* ← *próximo*

else *atual* ← *próximo* (somente com probabilidade $e^{\Delta/T}$

fim

Algoritmo 10 – Algoritmo de Recozimento Simulado.

ALGORITMOS GENÉTICOS

Estes algoritmos combinam as seguintes teorias, seguindo os passos indicados na Figura 6:

- A teoria da evolução das espécies - a sobrevivência das estruturas/soluções mais adaptadas a um ambiente/problema;
- Estruturas genéticas - utiliza a conceitos de hereditariedade e variabilidade genética para troca de informações entre as estruturas, visando a melhoria das mesmas.

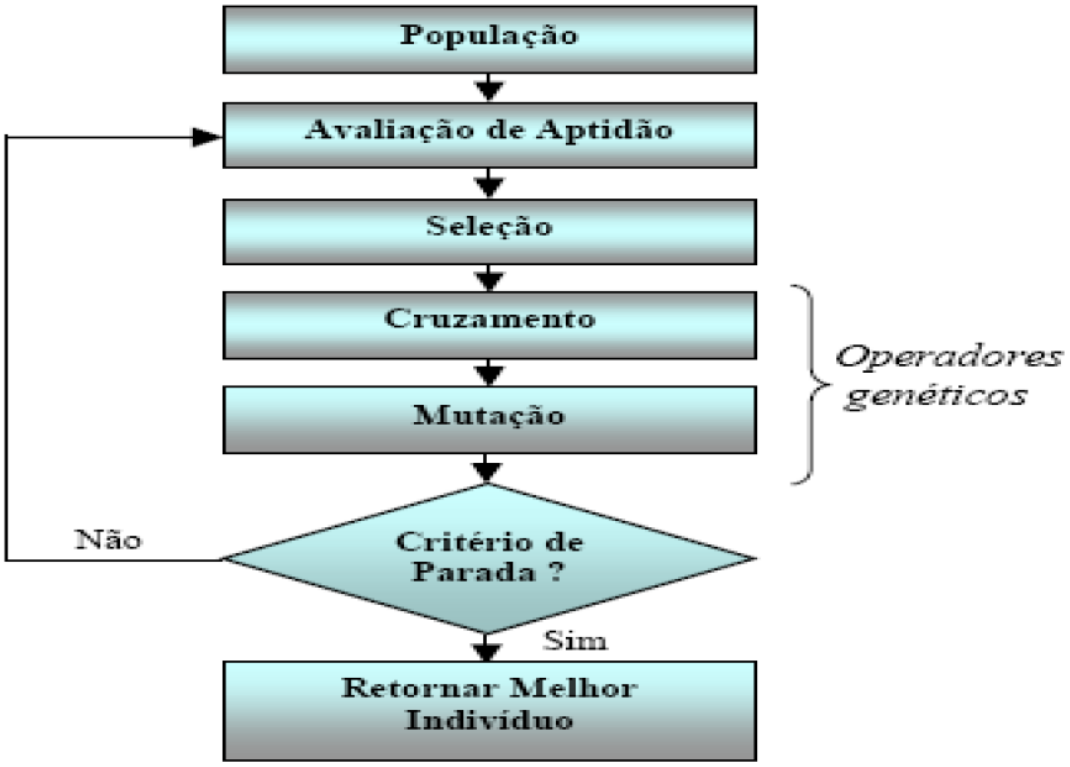

Figura 6 – Passos do Algoritmo Genético Canônico.

TEORIA DE JOGOS

A presença de um oponente torna a Teoria de Jogos mais complexa que os problemas de busca, caracterizando-os como problemas de contingência, ou seja, a cada momento a ação a ser tomada deve ser replanejada. Eles são situações mais próximas da realidade que os problemas de busca e a eficiência é muito importante.

Para a tomada da decisão perfeita em jogos com dois oponentes, consideram-se as seguintes condições: os oponentes jogam alternadamente e no final o vencedor ganha pontos ou o vencido perde pontos.

São elementos da definição formal nos problemas envolvendo jogos:

- Estado inicial
- Conjunto de operadores
- Teste de finalização
- Função de utilidade

O algoritmo que apresenta a forma de decisão mais adequada para problemas de jogos é denominado de algoritmo MINIMAX, sendo descrito no Algoritmo 11.

Algoritmo MINIMAX

```
função DECISÃO_MINIMAX (jogo) retorna {um operador}
        for each op in OPERADORES[jogo]
                VALOR[op] ← VALOR_MINIMAX(APLICAR(op, jogo), jogo)
        end
        returna  op com o maior VALOR[op]

função VALOR_MINIMAX(estado, jogo) retorna um valor de utilidade
        if TESTE_TERMINAL[jogo](estado) then
                retorna UTILIDADE[jogo](estado)
        else if MAX deve movimentar em estado then
                retorna o maior VALOR_MINIMAX de SUCESSORES(estado)
        else
                retorna o menor VALOR_MINIMAX de SUCESSORES(estado)
fim
```

Algoritmo 11 – Algoritmo MINIMAX.

Não se pode, em geral, ir até o final da árvore, como no MINIMAX, por restrições de espaço de memória e de tempo. Para gerar um algoritmo viável, altera-se o algoritmo MINIMAX, criando o algoritmo Alfa-Beta, no qual poda-se a árvore em determinado nível; neste caso, em vez da função de utilidade usa-se uma função de avaliação heurística, que fornece uma estimativa do valor dos estados-folha.