Nota: Este material complementar, disoinível em https://www.rettore.com.br/public_data/lectures/ representa uma cópia resumida de conteúdos bibliográficos disponíveis gratuitamente na Internet.
Conteúdo extraído e disponível em [1] - Seção 2.2; [2] Capítulo 3 pág. 175-188, e https://www.youtube.com/watch?v=z7XwVVYQRAA

Estratégias de Busca Sem Informação e de Busca Informada

Introdução

Os agentes inteligentes organizam dados de forma que possam localizar eficientemente as informações necessárias para tomar decisões em qualquer momento. À medida que essa estrutura cresce, é necessário realizar buscas eficientes para garantir que o agente seja eficaz em suas ações. De acordo com Russell e Norvig (2014), os algoritmos de busca são essenciais para agentes inteligentes e constituem a base de muitas soluções de inteligência artificial. Sem esses mecanismos, os agentes não conseguiram avaliar suas estruturas de dados e tomar decisões adequadas em um problema de IA.

Exemplo

Imagine que você consegue criar uma árvore com todas as possibilidades de jogadas para o jogo de xadrez. Começando com as 20 possibilidades iniciais para a jogada das brancas, todas as possíveis respostas das pretas, e assim por diante. Depois de várias dezenas de jogadas, você terá uma gigantesca árvore de possibilidades, contendo todas as combinações de jogadas possíveis. A pergunta, nesse caso, é: como visitar os nós dessa árvore hipotética para sempre escolher a melhor jogada, desenhando o melhor caminho até a vitória? A resposta é: precisamos de uma maneira eficiente para atravessar toda a árvore, analisando os resultados obtidos nos nós. É sobre isso que falaremos nesta seção: os mecanismos

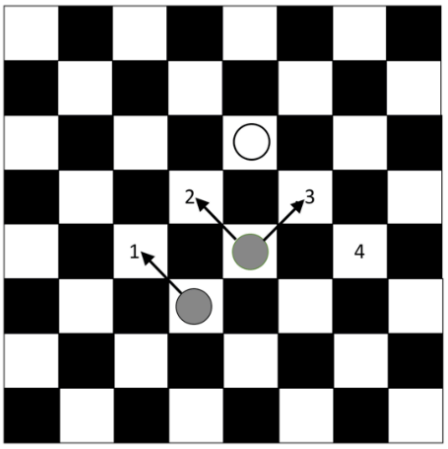
de busca em estruturas de dados complexas. Esses mecanismos são fundamentais aos agentes inteligentes e, portanto, são parte das soluções de inteligência artificial.

Busca Sem Informação (Busca Cega)

Inclui estratégias como busca em largura (BFS), busca em profundidade (DFS), busca de custo uniforme.Conteúdo extraído e disponível em [1] - Seção 2.2; [2] Capítulo 3 pág. 175-188, e https://www.youtube.com/watch?v=z7XwVVYQRAA

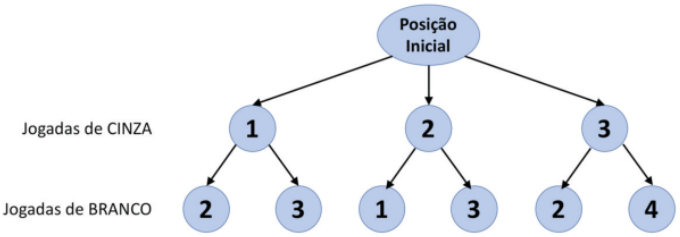
Alguns jogos e programas utilizam grandes estruturas de dados chamadas árvores de decisão, que registram o estado do jogo e as possíveis jogadas futuras. Um exemplo é um jogo de damas, onde, em uma determinada jogada, existem três movimentos possíveis, representados em uma árvore de decisão nas casas numeradas 1, 2 e 3. Para encontrar o dado necessário, é crucial realizar uma busca.[1]

Figura 2.6 | Uma situação em um jogo de damas



Fonte: elaborada pelo autor.

Figura 2.7 | Árvore de Decisão para o jogo da Figura 2.6



Fonte: elaborada pelo autor.

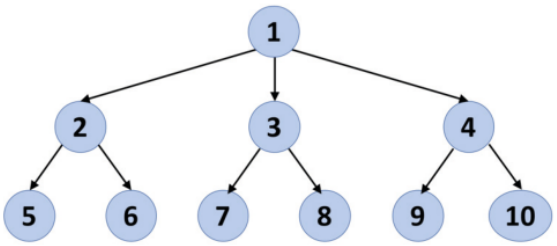
Para cada uma dessas jogadas possíveis, as peças brancas podem se mover, em resposta, para as casas 1, 2, 3 e 4 (dependendo da jogada das peças de cor cinza). Podemos, então, compor uma árvore de decisão para as jogadas possíveis, representada na Figura 2.7 a seguir:

Para entender a árvore de decisão descrita, o primeiro nível representa todas as jogadas possíveis das pedras cinzas, e o segundo nível mostra as jogadas possíveis das pedras brancas em resposta. Em cada nível, existem jogadas melhores e piores, e programas de jogos de damas atribuem valores a essas jogadas com base nos resultados favoráveis ou desfavoráveis.

Busca em Largura

A construção de uma árvore de decisão é essencial em jogos como damas e xadrez, mas, após montar parte da árvore, é necessário um método eficiente para analisar os nós e encontrar a melhor jogada. Para isso, entram em ação algoritmos de busca, como o de busca em largura (breadth-first search - BFS), que explora por completo os elementos que se encontram em um nível (largura horizontal) antes de se lançar ao nível posterior.

Figura 2.8 | Ordem de visita dos nós da Busca em Extensão

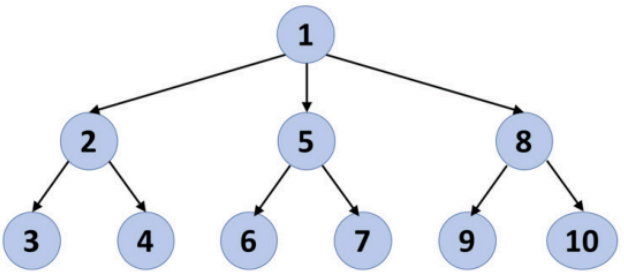


Fonte: elaborada pelo autor.

Busca em Profundidade

Cormen, Leierson, Rivest e Stein (2011) explicam que o método de busca em profundidade explora o máximo possível a profundidade de um ramo de uma árvore, indo até uma folha, antes de voltar à bifurcação mais próxima ainda não explorada para repetir o processo de aprofundamento. No exemplo de uma árvore, ao utilizar esse método, as visitas aos nós podem ocorrer em uma ordem específica, como ilustrado em um exemplo de figura posterior (Figura 2.9).

Figura 2.9 | Exemplo de ordem de visita dos nós da busca em profundidade

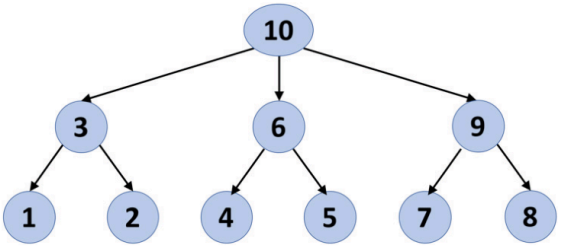


Fonte: elaborada pelo autor.

A busca em profundidade, segundo Cormen, Leierson, Rivest e Stein (2011), é um processo recursivo que envolve três tarefas principais: atravessar o ramo esquerdo (E), atravessar o ramo direito (D), e visitar o nó atual (V). Dependendo da ordem em que essas tarefas são executadas, resultam três tipos de ordenação diferentes dos nós:

- Pré-Ordem (VED): Primeiro o nó atual é visitado (V), seguido pela travessia recursiva do ramo esquerdo (E) e, depois, do ramo direito (D).
- Pós-Ordem (EDV): Primeiro atravessa-se recursivamente o ramo esquerdo (E), depois o direito (D) e, por último, o nó atual é visitado (V).

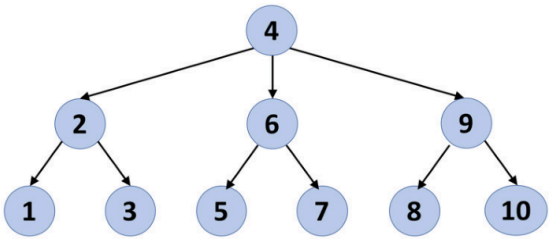
Figura 2.10 | Ordem de visita dos nós em pós-ordem



Fonte: elaborada pelo autor.

- Em-Ordem (EVD): Primeiro atravessa-se o ramo esquerdo (E), depois o nó atual é visitado (V), e por fim atravessa-se o ramo direito (D).

Figura 2.11 | Ordem de visita dos nós em em-ordem



Fonte: elaborada pelo autor.

Cada um desses métodos de busca em profundidade resulta em diferentes sequências de visitação dos nós.

Busca de Custo Uniforme

Quando ações têm custos diferentes, uma escolha óbvia é usar a busca de melhor-caminho, onde a função de avaliação é o custo do caminho desde a raiz até o nó atual. Isso é chamado de algoritmo de Dijkstra pela comunidade de ciência da computação teórica, e de busca de custo uniforme pela comunidade de IA. A ideia é que, enquanto a busca em largura se expande em ondas de profundidade uniforme—primeiro na profundidade 1, depois na profundidade 2, e assim por diante—a busca de custo uniforme se expande em ondas de custo de caminho uniforme. O algoritmo pode ser implementado como uma chamada para a BUSCA DE MELHOR-CAMINHO com o CUSTO DO CAMINHO como função de avaliação.

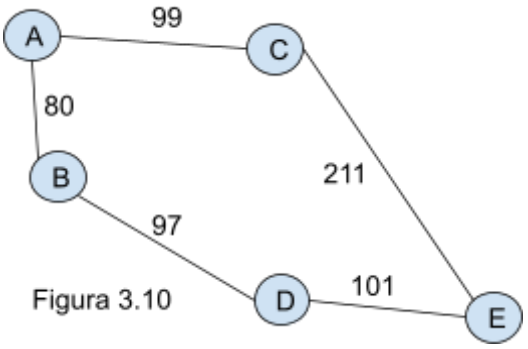


Figura 3.10

O problema apresentado é encontrar o caminho de A a E, Figura 3.10. Os sucessores de A são B e C, com custos de 80 e 99, respectivamente. O nó de menor custo, B, é expandido primeiro, adicionando D com custo total de 80 + 97 = 177. Em seguida, o nó de menor custo passa a ser C, que é expandido, adicionando E com custo total de 99 + 211 = 310.

E é o objetivo, mas o algoritmo só verifica se o nó é o objetivo ao expandir, não ao gerar o nó. Por isso, ainda não detectou que esse é um caminho para o objetivo. O algoritmo continua, escolhendo D para expandir e gerando um segundo caminho para E com um custo de 80 + 97 + 101 = 278, que é menor que o anterior, substituindo o caminho anterior na fronteira. Como este novo caminho tem o menor custo, ele é considerado a seguir, identificado como objetivo e retornado.

A busca de custo uniforme é completa e encontra soluções ótimas em termos de custo, pois a primeira solução encontrada terá um custo tão baixo quanto qualquer outro nó na fronteira. Esse método considera todos os caminhos em ordem crescente de custo, evitando cair em ciclos infinitos, desde que os custos das ações sejam maiores que um valor mínimo positivo (ϵ).

Busca Informada

Utiliza heurísticas para guiar a busca, como a busca A* e busca gulosa.Conteúdo extraído e disponível em [1] - Seção 2.2; [2] Capítulo 3 pág. 175-188

Problemas de natureza heurística são aqueles em que não é possível usar métodos determinísticos, sendo necessário resolver através de tentativa e erro, buscando sempre a melhor aproximação da solução ideal. Russell e Norvig (2013) descrevem que esse processo envolve a análise dos nós e suas disposições, ajustando uma estrutura interna à medida que as visitas ocorrem. Esse mecanismo, que refina a solução a cada nova interação, caracteriza o aprendizado de máquina.

Na busca heurística, os nós não são visitados em uma ordem fixa, como em buscas em profundidade ou em largura, mas a escolha de qual nó visitar ocorre de acordo com o aprendizado adquirido em visitas anteriores. Isso permite "atalhos" que aumentam a eficiência na busca por soluções, especialmente em árvores ou grafos grandes, onde a busca determinística pode ser demorada.

O tempo e a complexidade de uma busca heurística dependem de três fatores: a complexidade da árvore ou grafo, as regras que formam os nós e a eficiência do algoritmo de busca. Por exemplo, árvores geradas por jogos de damas são mais simples do que as geradas por jogos de xadrez devido às regras mais complexas do segundo.

Um exemplo comum de busca heurística é o método Minimax, utilizado em jogos de xadrez. No Minimax, a decisão é baseada na pior jogada possível para o oponente, enquanto o Maxmin considera a melhor jogada garantida para o jogador, sem prever a jogada do adversário.

Greedy Best-First Search

O **Greedy Best-First Search** (Busca Gulosa pelo Melhor Caminho Primeiro) é um algoritmo heurístico de busca que toma decisões com base na proximidade da solução, ou seja, tenta sempre expandir o nó que parece estar mais perto da meta. A ideia central é "gulosamente" escolher o próximo nó de acordo com a função heurística que estima o quão próximo aquele nó está do objetivo, sem considerar o custo do caminho percorrido até ali.

Como funciona?

1. **Função Heurística (h(n))**: O algoritmo usa uma função heurística que atribui um valor a cada nó, estimando quão longe ele está da solução. Quanto menor esse valor, mais promissor é o nó.
2. **Escolha do Próximo Nó**: O algoritmo mantém uma fila de nós, conhecida como **fronteira**. A cada etapa, ele seleciona o nó com o menor valor de h(n), acreditando que ele esteja mais próximo da solução.
3. **Expansão dos Nós**: O nó selecionado é expandido, e seus vizinhos (filhos) são adicionados à fronteira. O processo se repete até encontrar a solução ou a fronteira esvaziar (indicando que não há solução).

Características

- **Eficiente em buscas rápidas**: Como tenta sempre seguir o caminho aparentemente mais curto, pode ser eficiente em muitos casos.
- **Não garante a solução ótima**: Por não considerar o custo total do caminho, pode acabar em uma solução que não é a melhor possível.
- **Pode ser enganado**: Se a função heurística não for precisa, o algoritmo pode explorar caminhos que parecem promissores, mas levam a becos sem saída.

Exemplo Didático

Imagine que você está perdido em uma floresta e seu objetivo é chegar a uma torre que você consegue ver de onde está. O algoritmo Greedy Best-First Search seria como caminhar sempre na direção que te parece mais próxima da torre (usando apenas a visão da torre como sua "heurística"). Você não se preocupa em seguir trilhas que podem ser mais longas, só se preocupa em ir na direção que parece mais curta visualmente, o que nem sempre te levará pelo caminho mais eficiente.

A* search

O algoritmo **A*** é um dos métodos mais eficientes e populares de busca heurística utilizado para encontrar o caminho mais curto em grafos ou árvores. Ele combina elementos da busca em profundidade (que vai longe em um caminho antes de voltar) e da busca em largura (que explora todos os caminhos de forma equilibrada) para fazer escolhas informadas com base em uma função de custo.

Como funciona o A*?

O A* utiliza dois valores principais para cada nó no grafo ou árvore:

1. **g(n):** o custo do caminho percorrido do ponto inicial até o nó atual **n**.
2. **h(n):** uma estimativa do custo do caminho do nó **n** até o objetivo. Este valor é chamado de **heurística**.

O A* busca minimizar a soma desses dois valores:

- **f(n) = g(n) + h(n)**

Aqui, **g(n)** nos diz o quanto já caminhamos (o custo até o momento) e **h(n)** nos dá uma estimativa de quanto ainda resta (uma previsão do custo até o objetivo).

O A* escolhe, a cada passo, o nó que tem o menor valor de **f(n)**, o que significa que ele tenta equilibrar entre o caminho já percorrido e o caminho ainda restante. Isso garante que a busca seja eficiente, evitando nós que estão longe da solução.

Etapas do A*

1. **Começo:** Coloca-se o nó inicial em uma lista chamada **"aberta"** (open list).
2. **Exploração:** A cada passo, o A* remove o nó com o menor valor de **f(n)** da lista aberta, avalia seus vizinhos (nós conectados) e os adiciona à lista aberta.

3. **Atualização:** Para cada vizinho, recalcula-se o valor de **g(n)** (custo até ele) e **f(n)**. Se um vizinho já está na lista aberta com um valor maior de **f(n)**, ele é atualizado.
4. **Objetivo:** O processo continua até que o objetivo seja encontrado, ou a lista aberta esteja vazia (o que indica que não há solução).

Exemplo prático

Imagine que você está em um labirinto e quer encontrar a saída. O algoritmo A* vai explorar primeiro os caminhos que parecem mais promissores (aqueles que têm a menor combinação entre o que já andou e o que ainda falta), evitando caminhos que parecem longos ou sem saída. Ele usa a heurística para prever o quanto falta, o que faz com que economize tempo ao não explorar rotas que não levam a lugar algum.

Vantagens

- O A* é **completo**, ou seja, se houver uma solução, ele vai encontrá-la.
- Ele é **ótimo**, o que significa que encontrará o caminho de menor custo, desde que a heurística usada seja admissível (nunca superestime o custo real).
- Com uma boa função heurística, o A* pode ser muito eficiente.

Referências

1. Inteligência artificial, Ruy Flávio de Oliveira - http://cm-kls-content.s3.amazonaws.com/201802/INTERATIVAS_2_0/INTELIGENCIA_ARTIFICIAL/U1/LIVRO_UNICO.pdf
2. Inteligência Artificial Uma Abordagem Moderna - 4ª Edição (Versão Inglês)
3. Inteligência artificial: Tradução da 3a Edição, Peter Norvig, Stuart Russell, Elsevier Brasil, 2014
4. Conteúdo gratuito disponível na Internet