Aula 0

Inteligência artificial

A Inteligência Artificial (IA) abrange uma gama de técnicas que aparecem como comportamento senciente do computador. Por exemplo, a IA é usada para reconhecer rostos em fotografias nas redes sociais, vencer o campeão mundial de xadrez e processar sua fala quando você fala com Siri ou Alexa em seu telefone.

Neste curso, exploraremos algumas das ideias que tornam a IA possível:

0. **Procurar**

Encontrar uma solução para um problema, como um aplicativo de navegação que encontra a melhor rota da origem ao destino, ou como jogar um jogo e descobrir o próximo passo.

1. **Conhecimento**

Representar informações e tirar inferências delas.

2. **Incerteza**

Lidar com eventos incertos usando probabilidade.

3. **Otimização**

Encontrar não apenas uma maneira correta de resolver um problema, mas uma maneira melhor – ou a melhor – de resolvê-lo.

4. **Aprendizado**

Melhorar o desempenho com base no acesso a dados e experiência. Por exemplo, seu e-mail é capaz de distinguir spam de e-mails não-spam com base em experiências anteriores.

5. **Redes neurais**

Uma estrutura de programa inspirada no cérebro humano que é capaz de executar tarefas de forma eficaz.

6. **Linguagem**

Processar a linguagem natural, que é produzida e compreendida pelos humanos.

Procurar

Os problemas de busca envolvem um agente que recebe um estado inicial e um estado objetivo, e retorna uma solução de como passar do primeiro para o último. Um aplicativo navegador utiliza um processo típico de busca, onde o agente (a parte pensante do programa) recebe como entrada sua localização atual e seu destino desejado e, com base em um algoritmo de busca, retorna um caminho sugerido. No entanto, existem muitas outras formas de problemas de pesquisa, como quebra-cabeças ou labirintos.

Encontrar uma solução para um quebra-cabeça de 15 exigiria o uso de um algoritmo de busca.

- **Agente**
  Uma entidade que percebe seu ambiente e age sobre esse ambiente. Em um aplicativo navegador, por exemplo, o agente seria a representação de um carro que precisa decidir quais ações tomar para chegar ao destino.
- **Estado**
  Uma configuração de um agente em seu ambiente. Por exemplo, em um quebra-cabeça de 15 , um estado é qualquer maneira em que todos os números estão organizados no tabuleiro.
  - **Estado inicial**
    O estado a partir do qual o algoritmo de pesquisa é iniciado. Em um aplicativo de navegação, essa seria a localização atual.
- **Ações**
  Escolhas que podem ser feitas em um estado. Mais precisamente, as ações podem ser definidas como uma função. Ao receber o estado scomo entrada, Actions(s)retorna como saída o conjunto de ações que podem ser executadas no estado s. Por exemplo, em um quebra-cabeça 15 , as ações de um determinado estado são as maneiras pelas quais você pode

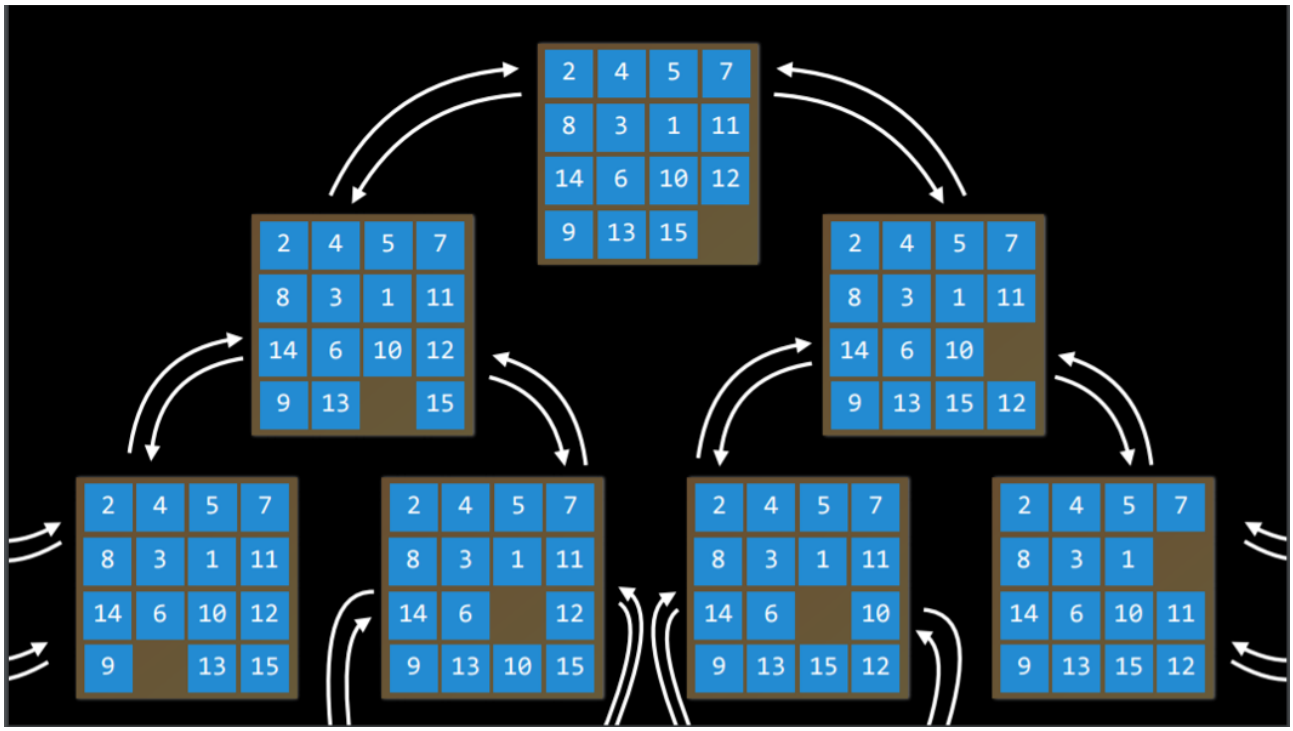
deslizar os quadrados na configuração atual (4 se o quadrado vazio estiver no meio, 3 se estiver próximo a um lado, 2 se estiver no canto).

- **Modelo de Transição**

Uma descrição do estado resultante da execução de qualquer ação aplicável em qualquer estado. Mais precisamente, o modelo de transição pode ser definido como uma função. Ao receber o estado se a ação acomo entrada, Results(s, a)retorna o estado resultante da execução da ação ano estado s. Por exemplo, dada uma certa configuração de um quebra-cabeça de 15 (estado s), mover um quadrado em qualquer direção (ação a) levará a uma nova configuração do quebra-cabeça (o novo estado).

- **Espaço de Estado**

O conjunto de todos os estados alcançáveis a partir do estado inicial por qualquer sequência de ações. Por exemplo, em um quebra-cabeça 15, o espaço de estados consiste em todas as configurações 16!/2 no tabuleiro que podem ser alcançadas a partir de qualquer estado inicial. O espaço de estados pode ser visualizado como um gráfico direcionado com estados, representados como nós, e ações, representadas como setas entre os nós.



- **Teste de meta**

A condição que determina se um determinado estado é um estado objetivo. Por exemplo, em um aplicativo de navegação, o teste de objetivo seria se a localização atual do agente (a representação do carro) está no destino. Se for – problema resolvido. Se não estiver, continuamos pesquisando.

- **Custo do caminho**

Um custo numérico associado a um determinado caminho. Por exemplo, um aplicativo de navegação não leva você simplesmente ao seu objetivo; isso é feito ao mesmo tempo em que minimiza o custo do caminho, encontrando o caminho mais rápido possível para você chegar ao seu estado objetivo.

Resolvendo problemas de pesquisa

- **Solução**

  Uma sequência de ações que leva do estado inicial ao estado objetivo.

  - **Solução ideal**

    Uma solução que possui o menor custo de caminho entre todas as soluções.

Num processo de pesquisa, os dados são frequentemente armazenados num **nó** , uma estrutura de dados que contém os seguintes dados:

- Um estado
- Seu nó pai , por meio do qual o nó atual foi gerado
- A ação que foi aplicada ao estado do pai para chegar ao nó atual
- O custo do caminho do estado inicial até este nó

Os nós contêm informações que os tornam muito úteis para fins de algoritmos de busca. Eles contêm um estado , que pode ser verificado usando o teste de meta para ver se é o estado final. Se for, o custo do caminho do nó pode ser comparado aos custos do caminho de outros nós , o que permite escolher a solução ideal . Uma vez escolhido o nó, em virtude de armazenar o nó pai e a ação que levou do nó pai ao nó atual, é possível rastrear cada passo do caminho desde o estado inicial até este nó, e esta sequência de ações é a solução .

No entanto, os nós são simplesmente uma estrutura de dados – eles não pesquisam, eles armazenam informações. Para realmente pesquisar, usamos a **fronteira** , o mecanismo que “gerencia” os nós . A fronteira começa contendo um estado inicial e um conjunto vazio de itens explorados, e então repete as seguintes ações até que uma solução seja alcançada:

Repita:

1. Se a fronteira estiver vazia,
   - Parar. Não há solução para o problema.
2. Remova um nó da fronteira. Este é o nó que será considerado.
3. Se o nó contiver o estado objetivo,
   - Devolva a solução. Pare .

Outro,

```
* Expand the node (find all the new nodes that could be reached from this node), and add resulting nodes to the frontier.
* Add the current node to the explored set.
```

Pesquisa em profundidade

Na descrição anterior da fronteira , uma coisa não foi mencionada. No estágio 1 do pseudocódigo acima, qual nó deve ser removido? Esta escolha tem implicações na qualidade da solução e na rapidez com que ela é

alcançada. Existem várias maneiras de resolver a questão de quais nós devem ser considerados primeiro, dois dos quais podem ser representados pelas estruturas de dados de **pilha** (na pesquisa em profundidade)
e **fila** (na pesquisa em largura ; e aqui está um exemplo bonito demonstração em desenho animado da diferença entre os dois).
Começamos com a abordagem de pesquisa em profundidade (DFS).
Um algoritmo de busca em profundidade esgota cada direção antes de tentar outra direção. Nestes casos, a fronteira é gerenciada como uma estrutura de dados em pilha . A frase de efeito que você precisa lembrar aqui é “ último a entrar, primeiro a sair ”. Depois que os nós são adicionados à fronteira, o primeiro nó a ser removido e considerado é o último a ser adicionado. Isso resulta em um algoritmo de busca que vai o mais fundo possível na primeira direção que atrapalha, deixando todas as outras direções para depois.
(Um exemplo de uma palestra externa: imagine uma situação em que você está procurando suas chaves. Em uma abordagem de busca em profundidade , se você decidir começar procurando nas calças, primeiro examinaria cada bolso, esvaziando cada bolso e examinando o conteúdo com cuidado. Você deixará de procurar em suas calças e começará a procurar em outro lugar apenas quando tiver esgotado completamente a busca em cada bolso de suas calças.)

- Prós:
  - Na melhor das hipóteses, esse algoritmo é o mais rápido. Se tiver “sorte” e sempre escolher o caminho certo para a solução (por acaso), então a pesquisa em profundidade leva o menor tempo possível para chegar a uma solução.
- Contras:
  - É possível que a solução encontrada não seja a ideal.
  - Na pior das hipóteses, este algoritmo explorará todos os caminhos possíveis antes de encontrar a solução, demorando assim o maior tempo possível antes de chegar à solução.

Exemplo de código:

```
    # Define the function that removes a node from the frontier and returns it.
    def remove(self):
            # Terminate the search if the frontier is empty, because this means that there is no solution.
        if self.empty():
            raise Exception("empty frontier")
        else:
                    # Save the last item in the list (which is the newest node added)
            node = self.frontier[-1]
            # Save all the items on the list besides the last node (i.e. removing the last node)
            self.frontier = self.frontier[:-1]
            return node
```

Pesquisa ampla

O oposto da pesquisa em profundidade seria a pesquisa em largura (BFS). Um algoritmo de busca em largura seguirá múltiplas direções ao mesmo tempo, dando um passo em cada direção possível antes de dar o segundo passo em cada direção. Neste caso, a fronteira é gerenciada como uma estrutura de dados de fila . A frase de efeito que você precisa lembrar aqui é " primeiro a entrar, primeiro a sair ". Nesse caso, todos os novos nós são somados em linha e os nós são considerados com base em qual deles foi adicionado primeiro (primeiro a chegar, primeiro a ser servido!). Isso resulta em um algoritmo de busca que dá um passo em cada direção possível antes de dar um segundo passo em qualquer direção.

(Um exemplo de palestra externa: suponha que você esteja em uma situação em que está procurando suas chaves. Nesse caso, se você começar pelas calças, você olhará no bolso direito. Depois disso, em vez de olhar para o bolso esquerdo , você vai dar uma olhada em uma gaveta. Depois na mesa. E assim por diante, em todos os locais que você imaginar. Somente depois de ter esgotado todos os locais você voltará às calças e procurará no próximo bolso.)

- Prós:
  - Este algoritmo tem garantia de encontrar a solução ótima.
- Contras:
  - É quase certo que esse algoritmo levará mais tempo do que o tempo mínimo para ser executado.
  - Na pior das hipóteses, esse algoritmo leva o maior tempo possível para ser executado.

Exemplo de código:

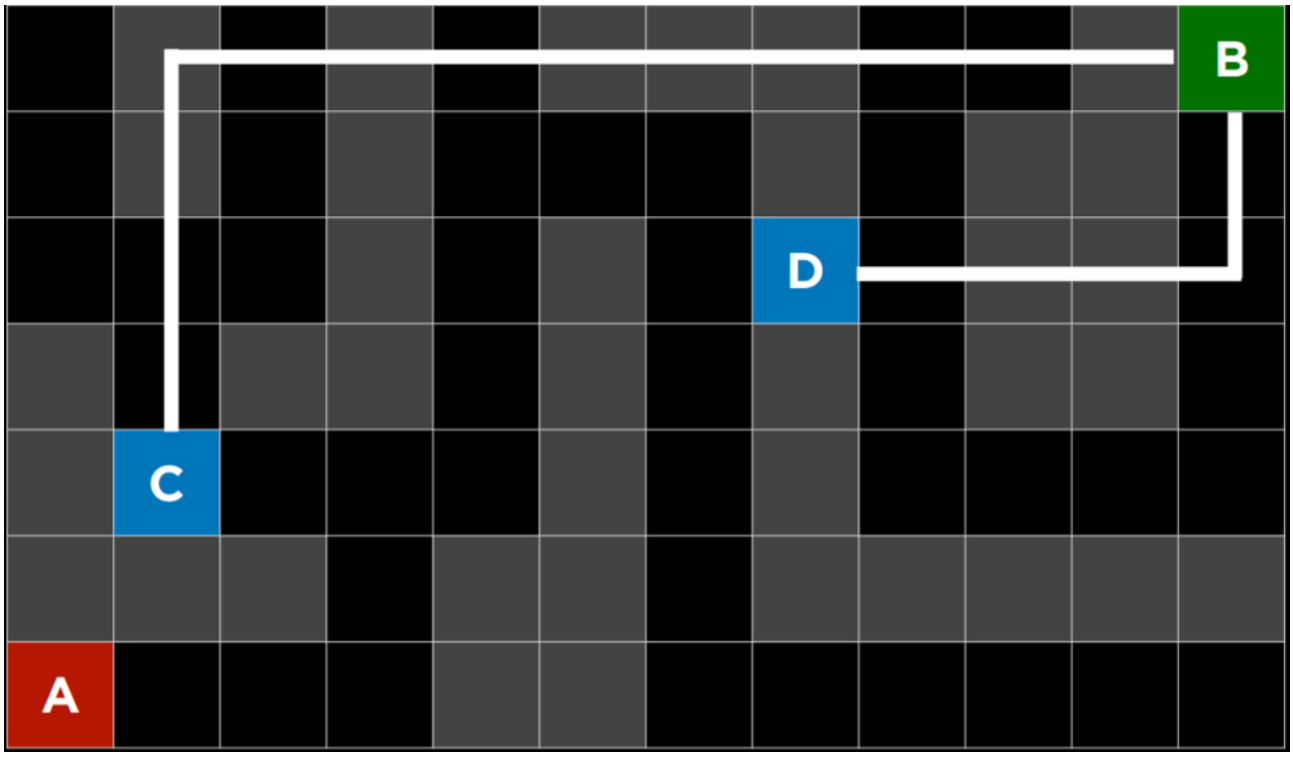
```
    # Define the function that removes a node from the frontier and returns it.
    def remove(self):
            # Terminate the search if the frontier is empty, because this means that there is no solution.
        if self.empty():
            raise Exception("empty frontier")
        else:
            # Save the oldest item on the list (which was the first one to be added)
            node = self.frontier[0]
            # Save all the items on the list besides the first one (i.e. removing the first node)
            self.frontier = self.frontier[1:]
            return node
```

Melhor primeira pesquisa gananciosa

Em largura e em profundidade são ambos algoritmos de pesquisa **desinformados .** Ou seja, esses algoritmos não utilizam nenhum conhecimento sobre o problema que não tenham adquirido por meio de sua própria exploração. No entanto, na maioria das vezes acontece que algum conhecimento sobre o problema está, de facto, disponível. Por exemplo, quando um solucionador de labirinto humano entra em um cruzamento, o

humano pode ver qual caminho segue na direção geral da solução e qual caminho não segue. A IA pode fazer o mesmo. Um tipo de algoritmo que considera conhecimento adicional para tentar melhorar seu desempenho é chamado de algoritmo de busca **informada** .

A busca gananciosa do melhor primeiro expande o nó que está mais próximo do objetivo, conforme determinado por uma **função heurística** h(n) . Como o próprio nome sugere, a função estima o quão próximo do objetivo o próximo nó está, mas pode estar errada. A eficiência do algoritmo ganancioso do melhor primeiro depende de quão boa é a função heurística. Por exemplo, em um labirinto, um algoritmo pode usar uma função heurística que se baseia na **distância de Manhattan** entre os possíveis nós e o final do labirinto. A distância de Manhattanignora paredes e conta quantos passos para cima, para baixo ou para os lados seriam necessários para ir de um local até o local do objetivo. Esta é uma estimativa fácil que pode ser derivada com base nas coordenadas (x, y) da localização atual e da localização do objetivo.



Distância Manhattan

No entanto, é importante enfatizar que, como acontece com qualquer heurística, ela pode dar errado e levar o algoritmo por um caminho mais lento do que teria seguido de outra forma. É possível que um algoritmo de busca desinformado forneça uma solução melhor e mais rapidamente, mas é menos provável que o faça do que um algoritmo informado .

Uma pesquisa

Um desenvolvimento do algoritmo ganancioso do melhor primeiro , a pesquisa A* considera não apenas h(n) , o custo estimado da localização atual até a meta, mas também g(n) , o custo que foi acumulado até a localização atual. Ao

combinar esses dois valores, o algoritmo tem uma maneira mais precisa de determinar o custo da solução e otimizar suas escolhas em movimento. O algoritmo acompanha (custo do caminho até agora + custo estimado para a meta) e, uma vez que exceda o custo estimado de alguma opção anterior, o algoritmo abandonará o caminho atual e voltará para a opção anterior, impedindo-se assim de percorrendo um caminho longo e ineficiente que h(n)erroneamente marcado como melhor.
Mais uma vez, uma vez que este algoritmo também se baseia numa heurística, é tão bom quanto a heurística que emprega. É possível que em algumas situações seja menos eficiente do que a busca gananciosa do tipo melhor primeiro ou mesmo os algoritmos desinformados . Para que a pesquisa A* seja ótima, a função heurística, h(n) , deve ser:

1. Admissível , ou nunca superestimar o verdadeiro custo, e
2. Consistente , o que significa que o custo estimado do caminho até a meta de um novo nó, além do custo de transição do nó anterior para ele, é maior ou igual ao custo estimado do caminho até a meta do nó anterior. Para colocá-lo em forma de equação, h(n) é consistente se para cada nó n e nó sucessor n' com custo de etapa c , $h(n) \leq h(n') + c$.

Pesquisa adversária

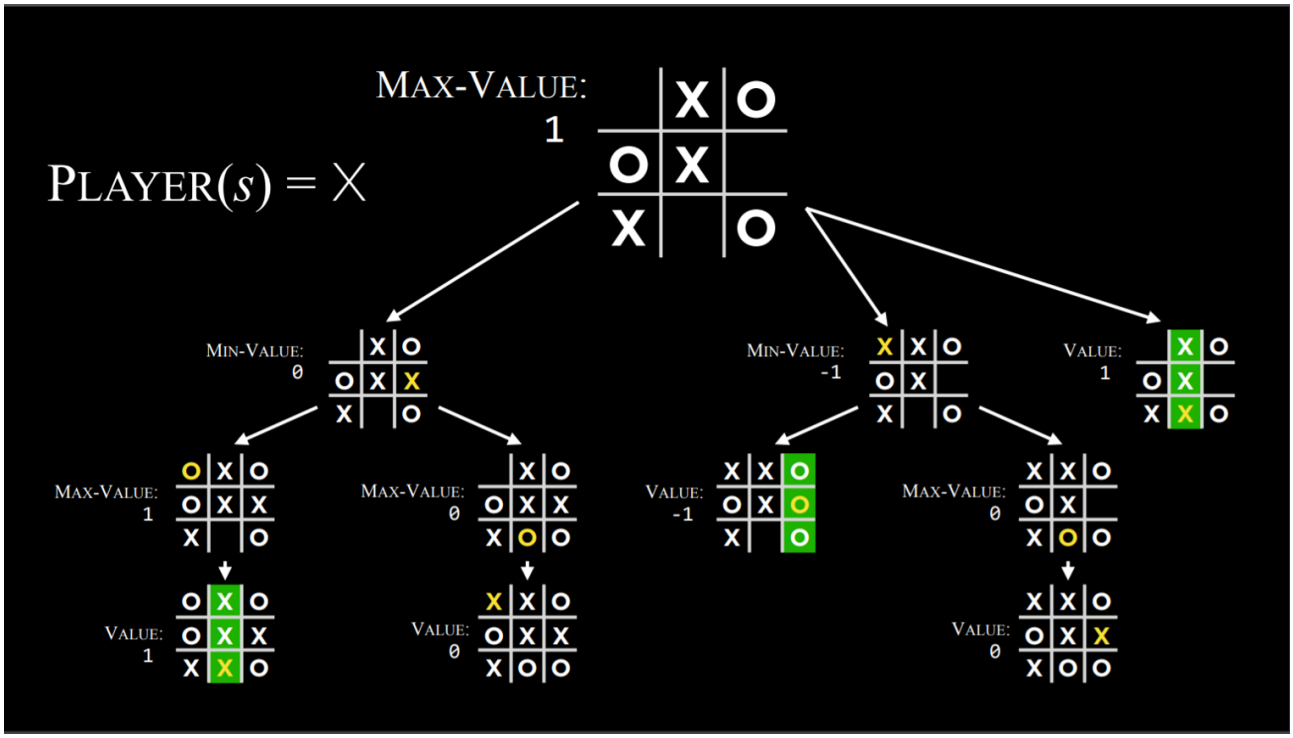
Enquanto, anteriormente, discutimos algoritmos que precisam encontrar uma resposta para uma pergunta, na **busca adversária** o algoritmo enfrenta um oponente que tenta atingir o objetivo oposto. Freqüentemente, a IA que usa busca adversária é encontrada em jogos, como o jogo da velha.

Minimáximo

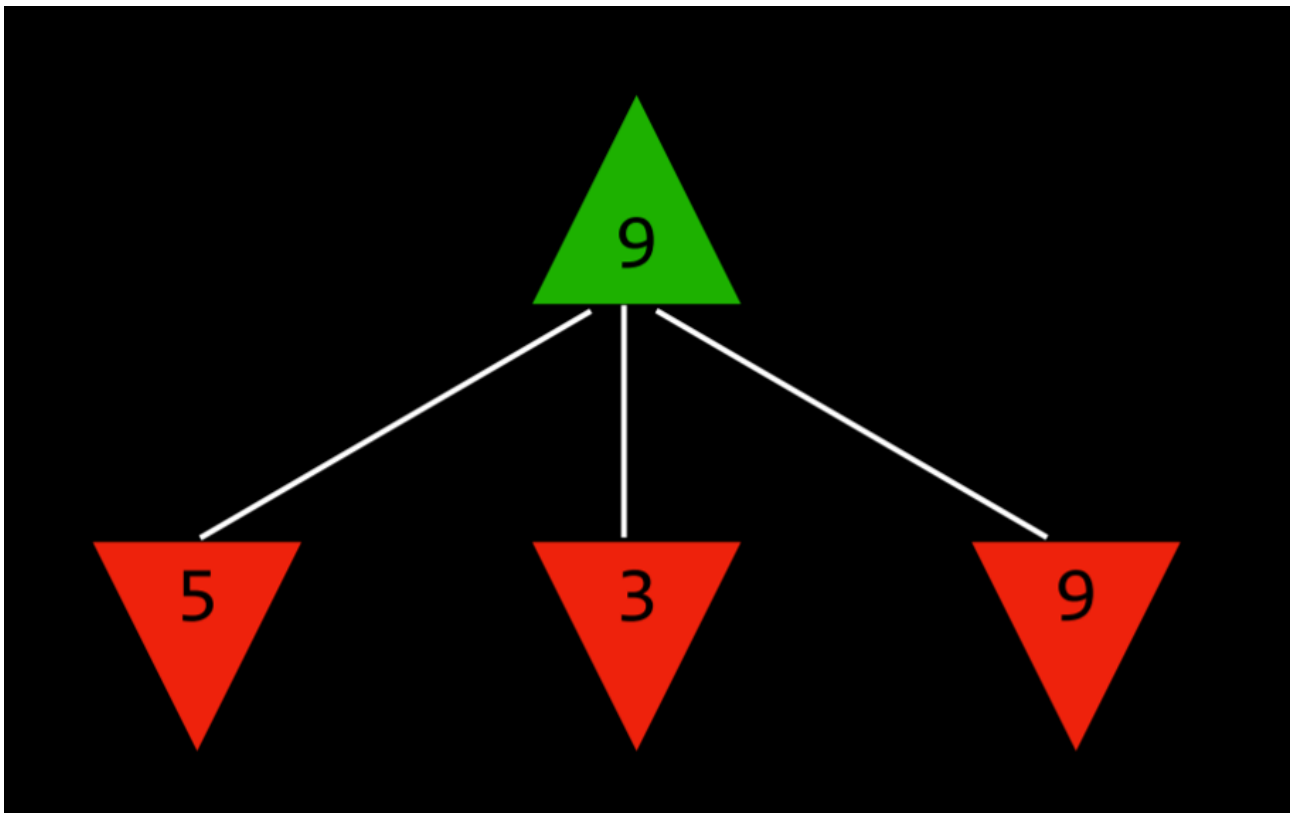
Um tipo de algoritmo de busca adversária, **o Minimax** representa as condições de vitória como (-1) para um lado e (+1) para o outro lado. Outras ações serão impulsionadas por estas condições, com o lado minimizador tentando obter a pontuação mais baixa e o lado maximizador tentando obter a pontuação mais alta.
Representando uma IA Tic-Tac-Toe :

- S_0 : Estado inicial (no nosso caso, um tabuleiro 3X3 vazio)
- Jogador(es) : função que, dado um estado s , retorna qual é a vez do jogador (X ou O).
- Action(s) : uma função que, dado um estado s , retorna todos os movimentos legais neste estado (quais posições estão livres no tabuleiro).
- Resultado(s, a) : uma função que, dado um estado s e uma ação a , retorna um novo estado. Este é o tabuleiro que resultou da realização da ação a no estado s (fazer uma jogada no jogo).
- Terminal(s) : função que, dado um estado s , verifica se esta é a última etapa do jogo, ou seja, se alguém ganhou ou se há empate. Retorna True se o jogo terminou, False caso contrário.
- Utilidade(s) : uma função que, dado um estado terminal s , retorna o valor de utilidade do estado: -1, 0 ou 1.

Como funciona o algoritmo :
Recursivamente, o algoritmo simula todos os jogos possíveis que podem ocorrer, começando no estado atual e até atingir um estado terminal. Cada estado terminal é avaliado como (-1), 0 ou (+1).



Algoritmo Minimax no Tic Tac Toe
Sabendo com base no estado de quem é a vez, o algoritmo pode saber se o jogador atual, ao jogar de forma otimizada, escolherá a ação que leva a um estado com valor inferior ou superior. Dessa forma, alternando entre minimizar e maximizar, o algoritmo cria valores para o estado que resultaria de cada ação possível. Para dar um exemplo mais concreto, podemos imaginar que o jogador maximizador pergunta a cada passo: “se eu realizar esta ação, resultará um novo estado. Se o jogador minimizador joga de forma otimizada, que ação esse jogador pode tomar para chegar ao valor mais baixo?” Porém, para responder a esta questão, o jogador maximizador tem que perguntar: “Para saber o que o jogador minimizador fará, preciso simular o mesmo processo na mente do minimizador: o jogador minimizador tentará perguntar: 'se eu realizar esta ação , que ação o jogador maximizador pode realizar para trazer o valor mais alto?'” Este é um processo recursivo e pode ser difícil entendê-lo; olhar o pseudocódigo abaixo pode ajudar. Eventualmente, através deste processo de raciocínio recursivo, o jogador maximizador gera valores para cada estado que poderiam resultar de todas as ações possíveis no estado atual. Após ter esses valores, o jogador maximizador escolhe o maior.



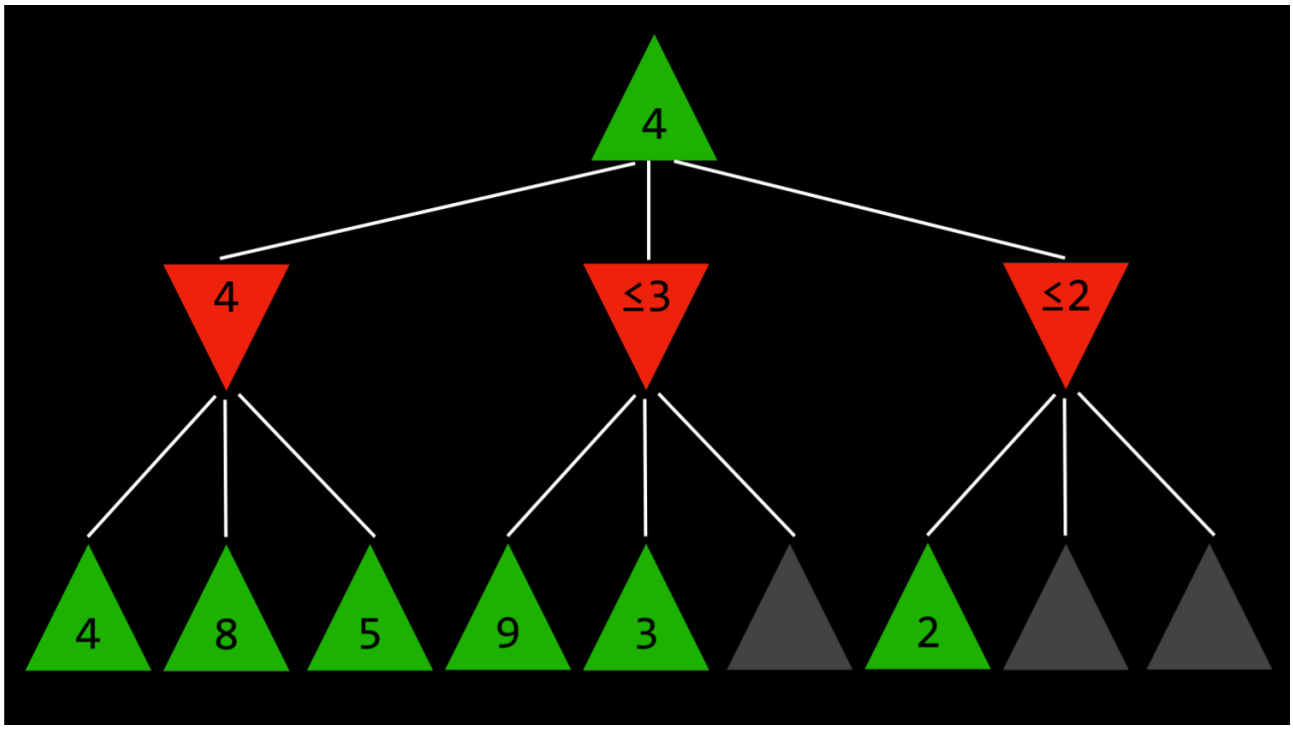
O Maximizador considera os valores possíveis dos estados futuros.
Para colocar em pseudocódigo, o algoritmo Minimax funciona da seguinte maneira:

- Dado um estado
  - O jogador maximizador escolhe a ação a em Actions(s) que produz o valor mais alto de Min-Value(Result(s, a)) .
  - O jogador minimizador escolhe a ação a em Actions(s) que produz o menor valor de Max-Value(Result(s, a)) .
- Valor máximo da função (estado)
  - v = -∞
  - if Terminal(estado) :
    retornar Utilitário(estado)
  - para ação em Actions(state) :
    v = Máx(v, Valor Mínimo(Resultado(estado, ação)))
    retornar v
- Valor mínimo da função (estado) :
  - v = ∞
  - if Terminal(estado) :
    retornar Utilitário(estado)
  - para ação em Actions(state) :
    v = Min(v, Max-Value(Resultado(estado, ação)))
    retornar v

Poda Alfa-Beta

Uma forma de otimizar o Minimax , **a poda alfa-beta** ignora alguns dos cálculos recursivos que são decididamente desfavoráveis. Após estabelecer o valor de uma ação, se houver evidência inicial de que a ação seguinte pode levar o adversário a obter uma pontuação melhor do que a ação já estabelecida, não há necessidade de investigar mais profundamente esta ação porque ela será decididamente menos favorável do que a ação já estabelecida. o previamente estabelecido.

Isto é mais facilmente demonstrado com um exemplo: um jogador maximizador sabe que, na próxima etapa, o jogador minimizador tentará alcançar a pontuação mais baixa. Suponha que o jogador maximizador tenha três ações possíveis, e a primeira tenha valor 4. Então o jogador começa a gerar o valor para a próxima ação. Para isso, o jogador gera os valores das ações do minimizador caso o jogador atual realize esta ação, sabendo que o minimizador escolherá a menor. Porém, antes de terminar o cálculo de todas as ações possíveis do minimizador, o jogador vê que uma das opções tem valor três. Isto significa que não há razão para continuar explorando outras ações possíveis para o jogador minimizador. O valor da ação ainda não avaliada não importa, seja 10 ou (-10). Se o valor for 10, o minimizador escolherá a opção mais baixa, 3, que já é pior que o 4 preestabelecido. Se a ação ainda não avaliada for (-10), o minimizador escolherá esta opção, (-10), que é ainda mais desfavorável ao maximizador. Portanto, calcular ações possíveis adicionais para o minimizador neste ponto é irrelevante para o maximizador, porque o jogador que maximiza já tem uma escolha inequivocamente melhor cujo valor é 4.



Minimax com limitação de profundidade

Há um total de 255.168 jogos Tic Tac Toe possíveis e 10^{29000} jogos possíveis no xadrez. O algoritmo minimax, conforme apresentado até agora, requer a geração de todos os jogos hipotéticos desde um determinado ponto até a condição terminal. Embora computar todos os jogos Tic-Tac-Toe não represente um desafio para um computador moderno, fazê-lo com o xadrez é atualmente impossível.

O Minimax com profundidade limitada considera apenas um número predefinido de movimentos antes de parar, sem nunca chegar a um estado terminal. Contudo, isto não permite obter um valor preciso para cada ação, uma vez que o final dos jogos hipotéticos não foi alcançado. Para lidar com este problema, o Minimax com limitação de profundidade depende de uma **função de avaliação**que estima a utilidade esperada do jogo a partir de um determinado estado, ou, em outras palavras, atribui valores aos estados. Por exemplo, em um jogo de xadrez, uma função de utilidade tomaria como entrada uma configuração atual do tabuleiro, tentaria avaliar sua utilidade esperada (com base nas peças que cada jogador possui e suas localizações no tabuleiro) e então retornaria um resultado positivo ou positivo. um valor negativo que representa o quão favorável o tabuleiro é para um jogador em relação ao outro. Esses valores podem ser usados para decidir sobre a ação correta e, quanto melhor for a função de avaliação, melhor será o algoritmo Minimax que depende dela.