Tudo bem. Bem-vindos a todos à Introdução à Inteligência Artificial com Python. Meu nome é Brian Yu. E nesta classe, exploraremos algumas das ideias, técnicas e algoritmos que estão na base da inteligência artificial. Agora, a inteligência artificial abrange uma ampla variedade de tipos de técnicas. Sempre que você ver um computador fazer algo que pareça inteligente ou racional de alguma forma, como reconhecer o rosto de alguém em uma foto, ou ser capaz de jogar um jogo melhor do que as pessoas, ou ser capaz de entender o idioma humano quando falamos com nossos telefones e eles entendem o que queremos dizer e são capazes de responder a nós, esses são todos exemplos de IA, ou inteligência artificial. E nesta classe, exploraremos algumas das ideias que tornam essa IA possível. Então, começaremos nossas conversas com a pesquisa. O problema de que temos uma IA e gostaríamos que ela fosse capaz de procurar soluções para algum tipo de problema, não importa qual seja esse problema. Seja tentar obter direções de condução de um ponto A a um ponto B, ou tentar descobrir como jogar um jogo, dando um jogo de tic-tac-toe, por exemplo, descobrindo qual movimento deve fazer. Depois disso, vamos olhar para o conhecimento. Idealmente, queremos que nossa IA saiba informações, seja capaz de representar essas informações e, mais importante, seja capaz de tirar inferências dessas informações. Ser capaz de usar as informações que sabe e tirar conclusões adicionais. Então, falaremos sobre como a IA pode ser programada para fazer isso. Em seguida, exploraremos o tópico da incerteza. Falando sobre ideias de, o que acontece se um computador não tiver certeza sobre um fato, mas talvez só tenha certeza com uma certa probabilidade? Então, falaremos sobre algumas das ideias por trás da probabilidade e de como os computadores podem começar a lidar com eventos incertos, a fim de serem um pouco mais inteligentes nesse sentido. Depois disso, voltaremos nossa atenção para a otimização. Problemas de quando o computador está tentando otimizar para algum tipo de objetivo, especialmente em uma situação em que pode haver várias maneiras de um computador resolver um problema, mas estamos procurando por uma maneira melhor ou, potencialmente, a melhor maneira, se isso for possível. Em seguida, olharemos para o aprendizado de máquina ou aprendizado mais geral. Ao olhar para como, quando temos acesso a dados, nossos computadores podem ser programados para serem muito inteligentes, aprendendo a partir de dados e aprendendo com a experiência, sendo capazes de realizar uma tarefa cada vez melhor com maior acesso a dados. Então, o seu email, por exemplo, onde a sua caixa de entrada de email de alguma forma sabe quais dos seus emails são bons e quais são spam. Esses são todos exemplos de computadores sendo capazes de aprender com experiências e dados anteriores. Também olharemos para como os computadores são capazes de se inspirar na inteligência humana, olhando para a estrutura do cérebro humano e como as redes neurais podem ser um analogia computacional para essa ideia. E como, aproveitando certo tipo de estrutura de um programa de computador, podemos escrever redes neurais que são capazes de realizar tarefas muito, muito eficazmente. E, finalmente, voltaremos nossa atenção para o idioma. Não linguagens de programação, mas idiomas humanos que falamos todos os dias. E olhando para os desafios que surgem quando um computador tenta entender o idioma natural e como algumas das técnicas de processamento de linguagem natural que ocorrem na inteligência artificial moderna podem realmente funcionar. Mas hoje começaremos nossa conversa com a pesquisa. Esse problema de tentar descobrir o que fazer quando temos algum tipo de situação em que o computador está, algum tipo de ambiente em que um agente está, para assim dizer. E gostaríamos que esse agente fosse capaz de de alguma forma procurar uma solução para esse problema. Agora, esses problemas podem vir em qualquer número de formatos diferentes.

Um exemplo, por exemplo, poderia ser algo como esse clássico quebra-cabeça de 15 peças com os blocos deslizantes que você pode ter visto, onde você está tentando deslizar as peças para garantir que todos os números estejam em ordem. Este é um exemplo do que você pode chamar de um problema de busca. O quebra-cabeça de 15 começa em um estado inicialmente misturado e precisamos de alguma forma de encontrar movimentos para retornar o quebra-cabeça ao seu estado resolvido. Mas há problemas semelhantes que você pode enquadrar de outras maneiras. Tentar encontrar seu caminho através de um labirinto, por exemplo, é outro exemplo de um problema de busca. Você começa em um lugar, tem algum objetivo para onde está tentando chegar e precisa descobrir a sequência correta de ações que o levará desse estado inicial ao objetivo. E, embora isso seja um pouco abstrato, sempre que falamos sobre resolução de labirintos nesta classe, você pode traduzi-lo para algo um pouco mais real, como direções de condução. Se você já se perguntou como o Google Maps consegue descobrir qual é a melhor maneira de você chegar do ponto A ao ponto B e quais são as curvas a serem feitas, por exemplo, a que horas, dependendo do tráfego, é muitas vezes algum tipo de algoritmo de busca. Você tem uma IA que está tentando chegar de uma posição inicial a algum tipo de objetivo, tomando alguma sequência de ações. Então, começaremos nossas conversas hoje pensando nesses tipos de problemas de busca e o que é necessário para resolver um problema de busca desses para que uma IA possa encontrar uma boa solução. Para isso, no entanto, precisaremos introduzir um pouco de terminologia, algumas das quais já usei. Mas a primeira vez que precisaremos pensar é em um agente. Um agente é apenas alguma entidade que percebe seu ambiente, de alguma forma consegue perceber as coisas ao seu redor e agir sobre esse ambiente de alguma forma. Então, no caso das direções de condução, seu agente pode ser alguma representação de um carro que está tentando descobrir quais ações tomar para chegar a um destino. No caso do quebra-cabeça de 15, o agente pode ser a IA ou a pessoa que está tentando resolver esse quebra-cabeça, tentando descobrir quais peças mover para chegar à solução. Em seguida, introduzimos a ideia de um estado. Um estado é apenas alguma configuração do agente em seu ambiente. Então, no quebra-cabeça de 15, por exemplo, qualquer estado pode ser um desses três, por exemplo. Um estado é apenas alguma configuração das peças. Cada um desses estados é diferente e exigirá uma solução ligeiramente diferente. Uma sequência diferente de ações será necessária em cada um desses para chegar desse estado inicial ao objetivo, que é para onde estamos tentando chegar. O estado inicial, então. O que é isso? O estado inicial é apenas o estado onde o agente começa. É um desses estados onde vamos começar e este será o ponto de partida para nosso algoritmo de busca, digamos assim. Vamos começar com esse estado inicial e, em seguida, começar a raciocinar sobre ele, pensar em quais ações podemos aplicar a esse estado inicial para descobrir como chegar do início ao fim, da posição inicial ao que quer que seja nosso objetivo. E como nos movemos da posição inicial ao objetivo? Bem, no final, é através da tomada de ações. Ações são apenas escolhas que podemos fazer em qualquer estado dado. E na IA, sempre vamos tentar formalizar essas ideias um pouco mais precisamente, de modo que possamos programá-las um pouco mais matematicamente, digamos assim. Então, isso será um tema recorrente e podemos definir mais precisamente as ações como uma função. Vamos efetivamente definir uma função chamada ações que recebe como entrada S, onde S será algum estado que existe dentro de nosso ambiente, e ações de S vai levar o estado como entrada e retornar como saída o conjunto de todas as ações que podem ser executadas nesse estado. E então é possível que algumas ações sejam válidas em certos estados e não em outros. E veremos exemplos disso em breve também.

No caso do quebra-cabeça de 15 peças, por exemplo, geralmente haverá quatro ações possíveis que podemos

fazer a maior parte do tempo. Podemos deslizar uma peça para a direita, deslizar uma peça para a esquerda, deslizar uma peça para cima ou deslizar uma peça para baixo, por exemplo. E essas serão as ações disponíveis para nós. Então, de alguma forma, nosso AI, nosso programa, precisa de alguma codificação do estado, que geralmente será em algum formato numérico, e alguma codificação dessas ações. Mas também precisa de alguma codificação da relação entre essas coisas, como os estados e as ações se relacionam entre si? E para fazer isso, vamos introduzir ao nosso AI um modelo de transição, que será uma descrição do que estado obtemos depois de realizarmos alguma ação disponível em algum outro estado. E novamente, podemos ser um pouco mais precisos sobre isso, definir esse modelo de transição um pouco mais formalmente, novamente, como uma função. A função será uma função chamada resultado, que dessa vez recebe dois inputs. O primeiro input é S, algum estado. E o segundo input é A, alguma ação. E a saída desta função resultado é que ela nos dará o estado que obtemos depois de realizarmos a ação A no estado S. Então, vamos dar uma olhada em um exemplo para ver mais precisamente o que isso significa. Aqui está um exemplo de um estado do quebra-cabeça de 15 peças, por exemplo. E aqui está um exemplo de uma ação, deslizar uma peça para a direita. O que acontece se passarmos esses como inputs para a função resultado? Novamente, a função resultado recebe este tabuleiro, este estado, como seu primeiro input. E leva uma ação como segundo input. E, claro, aqui estou descrevendo as coisas visualmente para que você possa ver visualmente qual é o estado e qual é a ação. Em um computador, você pode representar uma dessas ações apenas como algum número que representa a ação. Ou se você estiver familiarizado com enums que permitem enumerar múltiplas possibilidades, pode ser algo assim. E o estado pode ser representado apenas como um array, ou array bidimensional, de todos esses números que existem. Mas aqui vamos mostrar visualmente para que você possa ver. Quando tomamos este estado e esta ação, passamos para a função resultado, a saída é um novo estado. O estado que obtemos depois de tirarmos uma peça e deslizarmos para a direita, e este é o estado que obtemos como resultado. Se tivéssemos uma ação diferente e um estado diferente, por exemplo, e passássemos isso para a função resultado, teríamos uma resposta completamente diferente. Então, a função resultado precisa cuidar de descobrir como pegar um estado e pegar uma ação e obter o que resulta. E isso vai ser o nosso modelo de transição que descreve como é que os estados e as ações estão relacionados entre si. Se tomarmos este modelo de transição e pensarmos mais geralmente e em todo o problema, podemos formar o que podemos chamar de espaço de estado, o conjunto de todos os estados que podemos obter a partir do estado inicial por meio de qualquer sequência de ações, tomando zero ou uma ou duas ou mais ações além disso, então poderíamos desenhar um diagrama que parece algo assim. Onde cada estado é representado aqui por um tabuleiro de jogo. E há setas que conectam cada estado a todos os outros estados que podemos obter a partir dele. E o espaço de estado é muito maior do que o que você vê aqui. Isso é apenas uma amostra do que o espaço de estado realmente poderia parecer. E, em geral, em muitos problemas de pesquisa, sejam eles esse quebra-cabeça de 15 peças específico ou direções de condução ou algo mais, o espaço de estado vai parecer algo assim. Temos estados individuais e setas que os conectam. E, muitas vezes, apenas para simplificar, vamos simplificar nossa representação de toda essa coisa como um gráfico, alguma sequência de nós e arestas que conectam nós. Mas você pode pensar nessa representação mais abstrata como a mesma ideia. Cada uma dessas pequenas círculos, ou nós, vai representar um dos estados dentro do nosso problema. E as setas aqui representam as ações que podemos tomar em qualquer estado, levando-nos de um estado particular para outro estado, por exemplo.

Tudo bem. Então, agora temos essa ideia de nós que representam esses estados, ações que podem nos levar de

um estado para outro e um modelo de transição que define o que acontece depois que tomamos uma ação específica. Então, o próximo passo que precisamos descobrir é como sabemos quando a IA terminou de resolver o problema. A IA precisa de alguma forma de saber quando chega ao objetivo, que encontrou o objetivo. Então, a próxima coisa que precisamos codificar em nossa inteligência artificial é um teste de objetivo, alguma forma de determinar se um determinado estado é um estado-objetivo. No caso de algo como direções de condução, pode ser bastante fácil. Se você estiver em um estado que corresponde ao que o usuário digitou como seu destino pretendido, bem, então você sabe que está em um estado-objetivo. No quebra-cabeça de 15, pode ser verificar os números para garantir que eles estejam em ordem ascendente. Mas a IA precisa de alguma forma de codificar se qualquer estado em que ela esteja é um objetivo ou não. E alguns problemas podem ter um objetivo, como um labirinto onde você tem uma posição inicial e uma posição final e esse é o objetivo. Em outros problemas mais complexos, você pode imaginar que existem múltiplos objetivos possíveis, que existem múltiplas maneiras de resolver um problema. E talvez não nos importemos com qual o computador encontra, desde que ele encontre um objetivo particular. No entanto, às vezes um computador não se preocupa apenas em encontrar um objetivo, mas em encontrar um objetivo bem, ou um com baixo custo. É por isso que a última peça de terminologia que usamos para definir esses problemas de pesquisa é algo chamado custo de caminho. Você pode imaginar que, no caso de direções de condução, seria bastante irritante se eu dissesse que queria direções de ponto A a ponto B e que a rota que o Google Maps me deu era uma longa rota com muitos desvios desnecessários, que demorou mais do que deveria para chegar ao destino. É por isso que, ao formular problemas de pesquisa, muitas vezes atribuímos a cada caminho algum tipo de custo numérico, algum número nos dizendo quanto custa essa opção em particular, onde alguns dos custos para qualquer ação particular podem ser mais caros do que o custo para alguma outra ação, por exemplo. Embora isso só aconteça em alguns tipos de problemas. Em outros problemas, podemos simplificar o diagrama e apenas assumir que o custo de qualquer ação particular é o mesmo. E isso provavelmente é o caso em algo como o quebra-cabeça de 15, por exemplo, onde não faz diferença se eu estou me movendo para a direita ou para a esquerda. A única coisa que importa é o número total de passos que tenho que dar para ir de ponto A a ponto B. E cada um desses passos tem um custo igual. Podemos apenas assumir que é algum custo constante, como um. E assim agora forma a base para o que podemos considerar um problema de pesquisa. Um problema de pesquisa tem algum tipo de estado inicial, algum lugar onde começamos, algum tipo de ação que podemos tomar ou múltiplas ações que podemos tomar em qualquer estado e tem um modelo de transição, alguma forma de definir o que acontece quando vamos de um estado e tomamos uma ação, em que estado acabamos como resultado. Além disso, precisamos de algum teste de objetivo para saber se chegamos a um objetivo ou não. E então precisamos de uma função de custo de caminho que nos diga para qualquer caminho particular, seguindo alguma sequência de ações, qual é o custo desse caminho. Qual é o seu custo em termos de dinheiro ou tempo ou alguma outra recurso que estamos tentando minimizar o uso.

A meta, no final, é encontrar uma solução, onde uma solução, neste caso, é apenas alguma sequência de ações que nos levará do estado inicial ao estado objetivo. E, idealmente, gostaríamos de encontrar não apenas qualquer solução, mas a solução ótima, que é uma solução que tem o menor custo de caminho entre todas as soluções possíveis. E, em alguns casos, pode haver múltiplas soluções ótimas, mas uma solução ótima significa apenas que não há maneira de termos feito melhor em termos de encontrar essa solução. Então, agora definimos o problema. E agora precisamos começar a descobrir como é que vamos resolver esse tipo de problema de pesquisa. E, para isso, você provavelmente imaginará que nosso computador precisará representar uma grande quantidade de dados

sobre esse problema em particular. Precisamos representar dados sobre onde estamos no problema. E talvez precise considerar múltiplas opções ao mesmo tempo. E, muitas vezes, quando estamos tentando embalar uma grande quantidade de dados relacionados a um estado juntos, faremos isso usando uma estrutura de dados que chamaremos de nó. Um nó é uma estrutura de dados que apenas vai manter o controle de uma variedade de valores diferentes, e especificamente no caso de um problema de pesquisa, ele vai manter o controle desses quatro valores em particular. Cada nó vai manter o controle de um estado, o estado em que estamos. E cada nó também vai manter o controle de um pai. Um pai sendo o estado antes de nós, ou o nó que usamos para chegar a esse estado atual. E isso será relevante porque, eventualmente, uma vez que alcançamos o nó objetivo, uma vez que chegamos ao fim, queremos saber qual sequência de ações usamos para chegar a esse objetivo. E a maneira como saberemos isso é olhando para esses pais para manter o controle do que nos levou ao objetivo, e o que nos levou a esse estado, e o que nos levou ao estado anterior, e assim por diante, retrocedendo até o começo para que saibamos a sequência inteira de ações que precisávamos para chegar do início ao fim. O nó também vai manter o controle da ação que tomamos para chegar do pai ao estado atual. E o nó também vai manter o controle de um custo de caminho. Em outras palavras, ele vai manter o controle do número que representa quanto tempo levou para chegar do estado inicial ao estado em que atualmente acontecemos. E veremos por que isso é relevante à medida que começarmos a falar sobre algumas das otimizações que podemos fazer em termos desses problemas de pesquisa em geral. Então, esta é a estrutura de dados que vamos usar para resolver o problema. E agora vamos falar sobre a abordagem, como podemos realmente começar a resolver o problema? Bem, como você pode imaginar, o que vamos fazer é começar em um estado particular e vamos apenas explorar a partir daí. A intuição é que, a partir de um determinado estado, temos múltiplas opções que poderíamos tomar, e vamos explorar essas opções. E, uma vez que exploramos essas opções, vamos descobrir que mais opções do que isso se tornarão disponíveis. E vamos considerar todas as opções disponíveis para serem armazenadas dentro de uma única estrutura de dados que chamaremos de fronteira. A fronteira vai representar todas as coisas que poderíamos explorar a seguir, que ainda não exploramos ou visitamos. Então, em nossa abordagem, vamos começar este algoritmo de pesquisa começando com uma fronteira que contém apenas um estado. A fronteira vai conter o estado inicial, porque no começo, esse é o único estado que conhecemos. Esse é o único estado que existe. E então nosso algoritmo de pesquisa vai efetivamente seguir um loop. Vamos repetir algum processo de novo e de novo e de novo. A primeira coisa que vamos fazer é se a fronteira estiver vazia, então não há solução. E podemos informar que não há maneira de chegar ao objetivo. E isso certamente é possível. Existem certos tipos de problemas que uma IA pode tentar explorar e perceber que não há maneira de resolver esse problema.

E assim, essa é uma informação útil para os humanos saberem, também. Então, se a fronteira estiver vazia, isso significa que não há mais nada para explorar e ainda não encontramos uma solução, então não há solução. Não há mais nada para explorar. Caso contrário, o que faremos é remover um nó da fronteira. Então, no momento inicial, a fronteira contém apenas um nó que representa o estado inicial. Mas com o tempo, a fronteira pode crescer. Ela pode conter vários estados. Então, aqui, vamos remover apenas um único nó da fronteira. Se esse nó for um objetivo, então encontramos uma solução. Então, removemos um nó da fronteira e nos perguntamos: é isso o objetivo? E fazemos isso aplicando o teste de objetivo que discutimos anteriormente, perguntando se estamos no destino ou se todos os números do quebra-cabeça de 15 estão em ordem. Então, se o nó contiver o objetivo, encontramos uma solução. Ótimo. Está feito. E, caso contrário, o que precisaremos fazer é expandir o nó. Este é um termo na inteligência artificial. Expandir o nó significa apenas olhar para todos os vizinhos desse nó. Em outras

palavras, considere todas as ações possíveis que eu poderia tomar a partir do estado que esse nó representa e quais nós eu posso chegar a partir daí. Vamos pegar todos esses nós, os próximos nós aos quais eu posso chegar a partir desse nó atual que estou olhando, e adicioná-los à fronteira. E então vamos repetir esse processo. Então, em um nível muito alto, a ideia é começar com uma fronteira que contém o estado inicial. E estamos constantemente removendo um nó da fronteira, olhando para onde podemos chegar a seguir e adicionando esses nós à fronteira, repetindo esse processo repetidamente até que ou removemos um nó da fronteira e ele contenha um objetivo, o que significa que resolvemos o problema. Ou encontramos uma situação em que a fronteira está vazia, a qual nos deixa sem solução. Então, vamos realmente tentar levar o pseudocódigo e colocá-lo em prática, dando uma olhada em um exemplo de um problema de busca de amostra. Aqui, tenho um gráfico de amostra. A está conectado a B por essa ação, B está conectado aos nós C e D, C está conectado a D, E está conectado a F. E o que eu gostaria de fazer é ter a minha IA encontrar um caminho de A para E. Queremos chegar desse estado inicial a esse estado de objetivo. Então, como vamos fazer isso? Bem, vamos começar com a fronteira que contém o estado inicial. Isso vai representar nossa fronteira. Então, nossa fronteira, inicialmente, conterá apenas A, esse estado inicial onde vamos começar. E agora vamos repetir esse processo. Se a fronteira estiver vazia, sem solução. Isso não é um problema, pois a fronteira não está vazia. Então, vamos remover um nó da fronteira como o próximo a ser considerado. Há apenas um nó na fronteira. Então, vamos removê-lo da fronteira. Mas agora A, esse nó inicial, esse é o nó que estamos considerando. Seguimos o próximo passo. Perguntamos a nós mesmos, esse nó é o objetivo? Não, não é. A não é o objetivo. E é o objetivo. Então, não retornamos a solução. Então, em vez disso, vamos para esse último passo, expandir o nó e adicionar os nós resultantes à fronteira. O que isso significa? Bem, significa pegar esse estado A e considerar para onde podemos chegar a seguir. E depois de A, o que podemos chegar a seguir é apenas B. Então, é isso que obtemos quando expandimos A. Encontramos B. E adicionamos B à fronteira. E agora B está na fronteira e repetimos o processo novamente. Digamos, tudo bem. A fronteira não está vazia. Então, vamos remover B da fronteira. B é agora o nó que estamos considerando. Perguntamos a nós mesmos, B é o objetivo? Não, não é. Então, vamos adiante e expandimos B e adicionamos seus nós resultantes à fronteira. O que acontece quando expandimos B? Em outras palavras, quais nós podemos chegar a partir de B? Bem, podemos chegar a C e D. Então, vamos adicionar C e D à fronteira. E agora temos dois nós na fronteira, C e D. E repetimos o processo novamente.

Removemos um nó da fronteira, por enquanto faremos isso arbitrariamente, apenas escolhendo C. Veremos mais tarde por que escolher qual nó remover da fronteira é na verdade uma parte importante do algoritmo. Mas por enquanto eu removo arbitrariamente C, digamos que não é o objetivo, então adicionaremos E, o próximo à fronteira. Então, digamos que eu remova E da fronteira. E agora estou olhando para o estado E. É um estado de meta? É porque estou tentando encontrar um caminho de A para E. Então eu retornaria o objetivo. E isso, agora, seria a solução, que agora estou apto a retornar a solução e encontrei um caminho de A para E.

Então, essa é a ideia geral, a abordagem geral deste algoritmo de busca, para seguir esses passos constantemente removendo nós da fronteira até que seja possível encontrar uma solução. Então, a próxima pergunta que você pode razoavelmente fazer é: o que pode dar errado aqui? Quais são os problemas potenciais com uma abordagem como essa? E aqui está um exemplo de um problema que pode surgir com essa abordagem. Imagine esse mesmo gráfico, o mesmo de antes, com uma mudança. A mudança sendo, agora, em vez de apenas uma seta de A para B,

também temos uma seta de B para A, o que significa que podemos ir nos dois sentidos. E isso é verdade em algo como o quebra-cabeça de 15, onde quando deslizo um azulejo para a direita, então posso deslizar um azulejo para a esquerda para voltar à posição original. Eu posso voltar e forth entre A e B. E é isso que essas setas duplas simbolizam, a ideia de que a partir de um estado eu posso chegar a outro e então posso voltar. E isso é verdade em muitos problemas de busca. O que vai acontecer se eu tentar aplicar a mesma abordagem agora? Bem, começarei com A, o mesmo de antes. E removo A da fronteira. E então vou considerar para onde posso chegar a partir de A. E depois de A, a única escolha é B, então B entra na fronteira. Então eu direi, tudo bem. Vamos dar uma olhada em B. É a única coisa que resta na fronteira. Para onde posso chegar de B? Antes era apenas C e D, mas agora, devido a essa seta reversa, posso chegar a A ou C ou D. Então, todos os três A, C e D. Todos eles agora entram na fronteira. São lugares para onde posso chegar a partir de B. E agora removo um da fronteira, e, você sabe, talvez eu seja azarado e talvez eu escolha A. E agora estou olhando para A novamente. E considero para onde posso chegar a partir de A. E de A, bem, posso chegar a B. E agora começamos a ver o problema, que se eu não tomar cuidado, vou de A para B e depois de volta para A e então para B novamente. E eu poderia estar nesse loop infinito onde nunca faço nenhum progresso porque estou constantemente indo para trás e para a frente entre dois estados que já vi. Então, qual é a solução para isso? Precisamos de alguma forma lidar com esse problema. E a forma como podemos lidar com esse problema é de alguma forma manter o controle do que já exploramos. E a lógica será: bem, se já exploramos o estado, não há razão para voltar a ele. Uma vez que exploramos um estado, não volte a ele, não se preocupe em adicioná-lo à fronteira. Não há necessidade. Então, aqui estará nossa abordagem revisada, uma maneira melhor de abordar esse tipo de problema de busca. E vai parecer muito similar, apenas com algumas modificações. Começaremos com uma fronteira que contém o estado inicial. O mesmo de antes. Mas agora começaremos com outra estrutura de dados, que seria apenas um conjunto de nós que já exploramos. Então, quais são os estados que exploramos? Inicialmente, está vazio. Temos um conjunto explorado vazio. E agora repetimos. Se a fronteira estiver vazia, sem solução. O mesmo de antes. Removemos um nó da fronteira, verificamos se é um estado de meta, retornamos a solução. Nada disso é diferente até agora. Mas agora, o que faremos é adicionar o nó ao estado explorado. Então, se acontecer de removermos um nó da fronteira e não for o objetivo, adicionaremos ao conjunto explorado para que saibamos que já exploramos. Não precisamos voltar a ele novamente se acontecer de aparecer mais tarde.

Então, o último passo, expandimos o nó e adicionamos os nós resultantes à fronteira. Mas antes de adicionarmos sempre os nós resultantes à fronteira, vamos ser um pouco mais espertos desta vez. Só vamos adicionar os nós à fronteira se eles ainda não estiverem na fronteira e se eles ainda não estiverem no conjunto explorado. Então, vamos verificar tanto a fronteira quanto o conjunto explorado, certificar-se de que o nó ainda não está em um dos dois e, desde que não esteja, adicionaremos à fronteira, mas não de outra forma. E é essa abordagem revisada que, finalmente, vai ajudar a garantir que não voltemos e forth entre dois nós. Agora, o ponto que eu tenho meio que ignorado aqui até agora é este passo aqui, removendo um nó da fronteira. Antes eu escolhia arbitrariamente, como vamos remover um nó e pronto. Mas, na verdade, é muito importante como decidimos estruturar nossa fronteira, como adicionamos e como removemos nossos nós. A fronteira é uma estrutura de dados. E precisamos fazer uma escolha sobre em que ordem vamos remover os elementos? E uma das estruturas de dados mais simples para adicionar e remover elementos é algo chamado pilha. E uma pilha é um tipo de dados primeiro a entrar, primeiro a sair. O que significa que a última coisa que eu adiciono à fronteira será a primeira coisa que eu removo da fronteira. Então, a coisa mais recente a entrar na pilha, ou na fronteira neste caso, será o nó que eu

explorar. Então, vamos ver o que acontece se eu aplicar essa abordagem baseada em pilha a algo como esse problema, encontrando um caminho de A para E. O que vai acontecer? Bem, novamente, começaremos com A. E diremos, tudo bem. Vamos olhar para A primeiro. E então, note que desta vez adicionamos A ao conjunto explorado. A é algo que agora exploramos, temos essa estrutura de dados que está acompanhando. Então, diremos de A podemos chegar a B. E tudo bem. De B, o que podemos fazer? Bem, de B, podemos explorar B e chegar a C e D. Então, adicionamos C e depois D. Então, agora, quando exploramos um nó, vamos tratar a fronteira como uma pilha, primeiro a entrar, primeiro a sair. D foi o último a entrar, então vamos explorar isso primeiro. E diremos, tudo bem, para onde podemos chegar de D? Bem, podemos chegar a F. Então, tudo bem. Vamos colocar F na fronteira. E agora, porque a fronteira é uma pilha, F é a coisa mais recente que entrou na pilha. Então, F é o que vamos explorar a seguir. Vamos explorar F e dizer, tudo bem. De onde podemos chegar de F? Bem, não podemos chegar a lugar nenhum, então nada é adicionado à fronteira. Então, agora, qual foi a coisa mais recente adicionada à fronteira? Bem, não é C, a única coisa que resta na fronteira. Vamos explorar isso, a partir do qual podemos dizer, tudo bem, de C podemos chegar a E. Então, E vai para a fronteira. E então, diremos, tudo bem. Vamos olhar para E e E agora é a solução. E agora resolvemos o problema. Então, quando tratamos a fronteira como uma pilha, um tipo de dados primeiro a entrar, primeiro a sair, é o resultado que obtemos. Vamos de A para B para D para F e, então, nós nos afastamos e descemos para C e, então, E. E é importante ter um senso visual de como esse algoritmo está funcionando. Fomos muito fundo nessa árvore de pesquisa, digamos assim, até o fundo onde atingimos um beco sem saída. E então, efetivamente, recuamos e exploramos essa outra rota que não tentamos antes. E é essa ideia de ir muito fundo na árvore de pesquisa, essa maneira como o algoritmo acaba funcionando quando usamos uma pilha, que chamamos essa versão do algoritmo de busca em profundidade. A busca em profundidade é o algoritmo de busca onde sempre exploramos o nó mais profundo da fronteira. Mantemos indo mais e mais fundo através da nossa árvore de pesquisa. E então, se atingirmos um beco sem saída, recuamos e tentamos algo diferente. Mas a busca em profundidade é apenas uma das opções de busca possíveis que poderíamos usar. Acaba por haver outro algoritmo chamado busca em largura, que se comporta muito de forma semelhante à busca em profundidade, com uma diferença.

Em vez de sempre explorar o nó mais profundo na árvore de pesquisa, como o Depth First Search (DFS) faz, o Breadth First Search (BFS) sempre vai explorar o nó mais superficial na fronteira. Então, o que isso significa? Bem, significa que, em vez de usar uma pilha, que o DFS usa, onde o item mais recente adicionado à fronteira é o que exploraremos em seguida, o BFS usará uma fila, que é um tipo de dados primeiro a entrar, primeiro a sair, onde a primeira coisa que adicionamos à fronteira é a primeira a ser explorada. Eles formam efetivamente uma linha ou uma fila, onde quanto mais cedo você chegar à fronteira, mais cedo você será explorado. Então, o que isso significaria para o mesmo problema, encontrar um caminho de A para E? Bem, começamos com A, da mesma forma que antes. Então, vamos explorar A e dizer: de onde podemos chegar a partir de A? Bem, de A podemos chegar a B. Da mesma forma que antes. De B, da mesma forma que antes. Podemos chegar a C e D, então C e D são adicionados à fronteira. Dessa vez, no entanto, adicionamos C à fronteira antes de D, então exploraremos C primeiro. Então, C é explorado. E de C, para onde podemos chegar? Bem, podemos chegar a E. Então, E é adicionado à fronteira. Mas porque D foi explorado antes de E, olharemos para D em seguida. Então, vamos explorar D e dizer: de onde podemos chegar a partir de D? Podemos chegar a F. E somente então diremos: tudo bem. Agora podemos chegar a E. E o que o BFS fez foi começar aqui, olhar para C e D e, em seguida, olhar para E. Efetivamente, estamos olhando para coisas a uma distância do estado inicial, depois a duas distâncias do estado

inicial. E somente então, coisas que estão a três distâncias do estado inicial. Ao contrário do DFS, que foi tão fundo quanto possível na árvore de pesquisa até atingir um beco sem saída e, finalmente, ter que recuar. Então, agora temos dois algoritmos de pesquisa diferentes que poderíamos aplicar para tentar resolver um problema. E vamos dar uma olhada em como esses funcionariam na prática com algo como resolução de labirinto, por exemplo. Aqui está um exemplo de um labirinto. Essas células vazias representam lugares onde nosso agente pode se mover. Essas células escurecidas e cinzentas representam paredes que o agente não pode atravessar. E, finalmente, nosso agente, nosso AI, vai tentar encontrar uma maneira de chegar da posição A à posição B por meio de alguma sequência de ações, onde essas ações são esquerda, direita, cima e baixo. O que o DFS faria nesse caso? Bem, o DFS seguirá apenas um caminho. Se ele chegar a um cruzamento, onde tem várias opções diferentes, o DFS, nesse caso, vai escolher uma. Não há uma preferência real. Mas vai continuar seguindo um até chegar a um beco sem saída. E quando ele atingir um beco sem saída, o DFS efetivamente volta ao último ponto de decisão e tenta o outro caminho. Exaurindo completamente todo esse caminho e quando ele perceber que, OK, o objetivo não está aqui, então ele volta sua atenção para esse caminho. Vai tão fundo quanto possível. Quando ele atinge um beco sem saída, ele recua e então tenta essa outra rota, continua indo tão fundo quanto possível em um caminho particular e quando percebe que é um beco sem saída, então ele recuará. E, finalmente, encontrará seu caminho até o objetivo. E talvez você tenha sorte e talvez tenha feito uma escolha diferente mais cedo, mas, no final, é assim que o DFS vai funcionar. Vai continuar seguindo até atingir um beco sem saída. E quando atingir um beco sem saída, recuará e procurará por uma solução diferente. E então, uma coisa que você pode razoavelmente perguntar é: esse algoritmo sempre vai funcionar? Ele sempre vai realmente encontrar uma maneira de chegar do estado inicial ao objetivo? E a resposta é que, desde que nosso labirinto seja finito, desde que haja espaços finitos onde podemos viajar, sim. O DFS vai encontrar uma solução, porque eventualmente ele vai explorar tudo. Se o labirinto acontecer de ser infinito e houver um espaço de estado infinito, o que existe em certos tipos de problemas, então é uma história um pouco diferente.

Mas, desde que nosso labirinto tem um número finito de quadrados, vamos encontrar uma solução. A próxima pergunta, no entanto, que queremos fazer é: será uma boa solução? É a solução ótima que podemos encontrar? E a resposta não é necessariamente. E vamos dar uma olhada em um exemplo disso. Neste labirinto, por exemplo, estamos tentando encontrar o caminho de A para B. E você nota aqui que há múltiplas soluções possíveis. Poderíamos ir desta forma, ou poderíamos subir para chegar de A a B. Agora, se tivermos sorte, a busca em profundidade escolherá essa maneira e chegará a B. Mas não há razão, necessariamente, para que a busca em profundidade escolha entre subir ou ir para a direita. É uma espécie de ponto de decisão arbitrário porque ambos serão adicionados à fronteira. E, eventualmente, se tivermos azar, a busca em profundidade pode escolher explorar primeiro este caminho porque é apenas uma escolha aleatória neste ponto. Ele vai explorar, explorar, explorar e eventualmente encontrará o objetivo, este caminho particular, quando na verdade havia uma melhor solução. Havia uma solução mais ótima que usava menos passos, supondo que estamos medindo o custo de uma solução com base no número de passos que precisamos dar. Então, a busca em profundidade, se tivermos azar, pode acabar não encontrando a melhor solução quando uma solução melhor estiver disponível. Então, se for DFS, busca em profundidade. Como a BFS, ou busca em largura, se compara? Como ela funcionaria nesta situação particular? Bem, o algoritmo vai parecer muito diferente visualmente em termos de como a BFS explora. Porque a BFS olha primeiro os nós mais rasos, a ideia é que a BFS vai olhar primeiro para todos os nós que estão a uma distância do estado inicial. Olhe aqui e olhe aqui, por exemplo. Apenas nos dois nós que estão imediatamente ao

lado deste estado inicial. Então, ele vai explorar nós que estão a duas distâncias, olhando para o estado e para esse estado, por exemplo. Então, ele vai explorar nós que estão a três distâncias, este estado e aquele estado. Enquanto a busca em profundidade escolheu apenas um caminho e continuou seguindo-o, a busca em largura, por outro lado, está tomando a opção de explorar todos os caminhos possíveis de alguma forma ao mesmo tempo, saltando de volta entre eles, olhando mais e mais profundamente em cada um, mas certificando-se de explorar os mais rasos ou os mais próximos do estado inicial mais cedo. Então, vamos continuar seguindo este padrão, olhando coisas que estão a quatro distâncias, olhando coisas que estão a cinco distâncias, olhando coisas que estão a seis distâncias, até finalmente chegarmos ao objetivo. E neste caso, é verdade que tivemos que explorar alguns estados que, no final, não nos levaram a lugar algum. Mas o caminho que encontramos para o objetivo foi o caminho ótimo. Esta é a maneira mais curta que poderíamos chegar ao objetivo. E então, o que pode acontecer então em um labirinto maior? Bem, vamos dar uma olhada em algo assim e como a busca em largura vai se comportar. Bem, a busca em largura, novamente, vai continuar seguindo os estados até que receba um ponto de decisão. Poderia ir para a esquerda ou para a direita. E enquanto a DFS escolheu apenas um e continuou seguindo-o até chegar a um beco sem saída, a BFS, por outro lado, vai explorar ambos. Ele vai dizer, olhe para este nó, então este nó, e vou olhar para este nó, então aquele nó, e assim por diante. E quando ele chegar a este ponto de decisão aqui, em vez de escolher um à esquerda ou dois à direita e explorar aquele caminho, ele vai novamente explorar ambos, alternando entre eles, indo mais e mais fundo. Vamos explorar aqui, e então talvez aqui e aqui, e então continuar. Explore aqui e lentamente faça nosso caminho, você pode ver visualmente cada vez mais longe. Quando chegarmos a este ponto de decisão, vamos explorar tanto para cima quanto para baixo até, finalmente, chegarmos ao objetivo. E o que você notará é que, sim, a busca em largura encontrou o caminho de A para B seguindo este caminho particular. Mas precisou explorar muitos estados para fazê-lo. E então vemos algumas trocas entre DFS e BFS.

Que em DFS pode haver alguns casos em que há algumas economias de memória, em comparação com uma abordagem de primeira largura onde a busca em largura primeiro, neste caso, teve que explorar muitos estados. Mas talvez isso nem sempre seja o caso. Então, agora vamos realmente nos voltar para algum código. E olhe para o código que poderíamos realmente escrever para implementar algo como busca em profundidade ou busca em largura no contexto de resolver um labirinto, por exemplo. Então, vou entrar no meu terminal. E o que eu tenho aqui dentro de maze.pi é uma implementação da mesma ideia de resolver labirintos. Eu defini uma classe chamada nó que, neste caso, está mantendo o estado, o pai, em outras palavras, o estado antes do estado, e a ação. Neste caso, não estamos mantendo o custo do caminho porque podemos calcular o custo do caminho no final depois de termos encontrado o caminho do estado inicial ao objetivo. Além disso, eu defini uma classe chamada fronteira de pilha. E se desconhecido com uma classe, uma classe é uma maneira para mim de definir uma maneira de gerar objetos em Python. Refere-se a uma ideia de programação orientada a objetos onde a ideia aqui é que eu gostaria de criar um objeto que seja capaz de armazenar todos os meus dados de fronteira. E eu gostaria de ter funções, conhecidas como métodos nesse objeto, que eu possa usar para manipular o objeto. E então o que está acontecendo aqui, se desconhecido com a sintaxe, é que eu tenho uma função que inicialmente cria uma fronteira que eu vou representar usando uma lista. E inicialmente minha fronteira é representada pela lista vazia. Não há nada na minha fronteira para começar. Eu tenho uma função que adiciona algo à fronteira, adicionando-o ao final da lista. Eu tenho uma função que verifica se a fronteira contém um estado particular. Eu tenho uma função vazia

que verifica se a fronteira está vazia. Se a fronteira estiver vazia, isso significa que o comprimento da fronteira é zero. E então eu tenho uma função para remover algo da fronteira. Eu não posso remover algo da fronteira se a fronteira estiver vazia. Então, verifico isso primeiro. Mas, caso contrário, se a fronteira não estiver vazia, lembre-se de que estou implementando essa fronteira como uma pilha, uma estrutura de dados de último a entrar, primeiro a sair. O que significa que a última coisa que eu adiciono à fronteira, em outras palavras, a última coisa na lista, é o item que eu deveria remover dessa fronteira. Então, o que você verá aqui é que eu removi o último item de uma lista. E se você indexar em uma lista Python com um negativo, isso lhe dá o último item na lista. Desde que zero é o primeiro item, o negativo um se enrola e lhe dá o último item na lista. Então, damos aquele nó. Chamamos aquele nó, atualizamos a fronteira aqui na linha 28 para dizer, vá em frente e remova aquele nó que você acabou de remover da fronteira. E então retornamos o nó como resultado. Então, essa classe aqui efetivamente implementa a ideia de uma fronteira. Ele me dá uma maneira de adicionar algo à fronteira e uma maneira de remover algo da fronteira como uma pilha. Eu também, só por precaução, implementei uma versão alternativa da mesma coisa chamada fronteira Q. Que, entre parênteses, você verá aqui, herda da fronteira de pilha, o que significa que fará todas as mesmas coisas que a fronteira de pilha fez, exceto pela maneira como removemos um nó da fronteira será ligeiramente diferente. Em vez de remover do final da lista, da maneira que faríamos em uma pilha, vamos remover do início da lista. self.frontierzero vai me dar o primeiro nó na fronteira, o primeiro que foi adicionado. E esse será o que retornamos no caso de um Q. Aqui embaixo eu tenho uma definição de uma classe chamada labirinto. Isso vai lidar com o processo de levar uma sequência, um arquivo de texto como labirinto, e descobrir como resolvê-lo.

Assim, vamos usar como entrada um arquivo de texto que parece algo assim, por exemplo, onde vemos os sinais de cerquilha que aqui representam paredes e eu tenho o personagem A representando a posição inicial e o personagem B representando a posição final. E você pode dar uma olhada no código para analisar esse arquivo de texto agora. Essa é a parte menos interessante. A parte mais interessante é essa função de resolução aqui, onde a função de resolução vai descobrir como realmente chegar do ponto A ao ponto B. E aqui vemos uma implementação da mesma ideia que vimos há um momento. Vamos manter o controle de quantos estados exploramos apenas para podermos relatar esses dados mais tarde. Mas eu começo com um nó que representa apenas o estado inicial. E eu começo com uma fronteira que, neste caso, é uma fronteira de pilha. E dado que estou tratando minha fronteira como uma pilha, você pode imaginar que o algoritmo que estou usando aqui agora é o de busca em profundidade. Porque a busca em profundidade ou DFS usa uma pilha como sua estrutura de dados. E inicialmente, essa fronteira vai conter apenas o estado inicial. Inicializamos um conjunto explorado que inicialmente está vazio. Não há nada que tenhamos explorado até agora. E agora aqui está nosso loop, essa noção de repetir algo de novo e de novo. Primeiro, verificamos se a fronteira está vazia chamando aquela função vazia que vimos a implementação há um momento. E se a fronteira estiver realmente vazia, vamos adiante e levantar uma exceção ou um erro do Python para dizer, desculpe. Não há solução para esse problema. Caso contrário, vamos remover um nó da fronteira, como chamando frontier.remove e atualizar o número de estados que exploramos. Porque agora exploramos um estado adicional, então dizemos self.numexplored mais igual a um, adicionando um ao número de estados explorados. Uma vez que removemos um nó da fronteira, lembre-se de que o próximo passo é ver se é ou não o objetivo, o teste de objetivo. E no caso do labirinto, o objetivo é bem fácil. Verifico para ver se o estado do nó é igual ao objetivo. Inicialmente, quando eu configurei o labirinto, configurei esse valor chamado objetivo que é a propriedade do labirinto, então posso verificar se o nó é realmente o

objetivo. E se for o objetivo, então o que eu quero fazer é retroceder para descobrir quais ações tomei para chegar a esse objetivo. E como faço isso? Lembramos que cada nó armazena seu pai - o nó que veio antes que usamos para chegar a esse nó - e também a ação usada para chegar lá. Então, posso criar esse loop onde estou constantemente olhando o pai de cada nó e mantendo o registro, para todos os pais, qual ação tomei para chegar do pai a este. Então, esse loop vai continuar repetindo esse processo de olhar por todos os nós pais até chegarmos ao estado inicial, que não tem pai, onde node.parent vai ser igual a nenhum. Enquanto faço isso, vou estar construindo a lista de todas as ações que estou seguindo e a lista de todas as células que fazem parte da solução. Mas vou reverter porque quando eu construo indo do objetivo até o estado inicial, estou construindo a sequência de ações do objetivo ao estado inicial, mas quero reverter para obter a sequência de ações do estado inicial ao objetivo. E isso, finalmente, vai ser a solução. Então, tudo isso acontece se o estado atual for igual ao objetivo. E, caso contrário, se não for o objetivo, bem, então vou adicionar esse estado ao conjunto explorado para dizer, já explorei esse estado agora. Não precisa voltar a ele se eu encontrá-lo no futuro. E então, essa lógica aqui implementa a ideia de adicionar vizinhos à fronteira. Estou dizendo, olhe todos os meus vizinhos. E implementei uma função chamada vizinhos que você pode dar uma olhada. E para cada um desses vizinhos, vou verificar, o estado já está na fronteira? O estado já está no conjunto explorado?

Se não estiver em nenhum dos dois, então eu irei adicionar este novo nó filho - este novo nó - à fronteira. Há uma quantidade razoável de sintaxe aqui, mas a chave aqui não é entender todas as nuances da sintaxe, embora sinta-se à vontade para dar uma olhada mais de perto neste arquivo por conta própria para ter uma noção de como está funcionando. Mas a chave é ver como isso é uma implementação do mesmo pseudocódigo, da mesma ideia que estávamos descrevendo há um momento na tela quando estávamos olhando para os passos que poderíamos seguir para resolver esse tipo de problema de busca. Então, agora vamos ver isso em ação. Eu irei executar maze.py em maze1.txt, por exemplo. E o que veremos aqui é uma impressão do que o labirinto inicialmente parecia. E então aqui, embaixo, é depois que resolvemos. Tivemos que explorar 11 estados para fazê-lo, e encontramos um caminho de A para B. E neste programa, eu acabei de gerar uma representação gráfica disso também - então eu posso abrir maze.png, que é gerado por este programa - que mostra onde, na cor mais escura aqui, está o muro. Vermelho é o estado inicial, verde é o objetivo e amarelo é o caminho que foi seguido. Encontramos um caminho do estado inicial ao objetivo. Mas agora vamos dar uma olhada em um labirinto mais sofisticado para ver o que pode acontecer em vez disso. Vamos olhar agora para maze2.txt, onde agora aqui temos um labirinto muito maior. Novamente, estamos tentando encontrar o caminho de A para B, mas agora você imaginará que a busca em profundidade talvez não seja tão sortuda. Pode não conseguir o objetivo na primeira tentativa. Pode ter que seguir um caminho e depois retroceder e explorar algo um pouco mais tarde. Então vamos tentar isso. Executar pythonmaze.py de maze2.txt, desta vez tentando neste outro labirinto. E agora a busca em profundidade consegue encontrar uma solução. Aqui, como indicado pelas estrelas, é uma maneira de chegar de A a B. E podemos representar isso visualmente abrindo este labirinto. Aqui está o que esse labirinto parece. E destacado em amarelo, é o caminho que foi encontrado do estado inicial ao objetivo. Mas quantos estados temos que explorar antes de encontrarmos esse caminho? Bem, lembre-se de que, no meu programa, eu estava acompanhando o número de estados que exploramos até agora. E então eu posso voltar para o terminal e ver que, bem, para resolver este problema, tivemos que explorar 399 estados diferentes. E na verdade, se eu fizer uma pequena modificação no programa e digo ao programa no final quando saímos esta imagem, eu adicionei um argumento chamado "show explored". E se eu definir "show explored" como verdadeiro e reexecutar este

programa pythonmaze.py executando-o em maze2, e então eu abro o labirinto, o que você verá aqui é, destacado em vermelho, todos os estados que tiveram que ser explorados para chegar do estado inicial ao objetivo. A Busca em Profundidade, ou DFS, não encontrou o caminho para o objetivo de imediato. Fez uma escolha de primeiro explorar esta direção. E quando explorou esta direção, teve que seguir todos os caminhos concebíveis, até o fim, mesmo este longo e sinuoso, para perceber que, você sabe o que, isso é um beco sem saída. E em vez disso, o programa precisava retroceder. Depois de ir nesta direção, deve ter ido nesta direção. Teve sorte aqui ao não escolher este caminho. Mas teve azar aqui, explorando esta direção, explorando um monte de estados que não precisava e, da mesma forma, explorando toda essa parte superior do gráfico quando provavelmente não precisava fazer isso também. Então, no geral, a busca em profundidade aqui realmente não está desempenhando de forma ótima, ou provavelmente explorando mais estados do que precisa. Ele encontra uma solução ótima, o melhor caminho para o objetivo, mas o número de estados necessários para explorar para fazê-lo, o número de passos que tive que dar, foi muito maior. Então vamos comparar. Como a Busca em Largura, ou BFS, faria neste mesmo labirinto em vez disso? E para fazer isso, é uma mudança muito fácil. O algoritmo para DFS e BFS é idêntico com a exceção do que estrutura de dados usamos para representar a fronteira.

Que, em DFS, eu usei uma fronteira de pilha - último a entrar, primeiro a sair - enquanto em BFS, vou usar uma fronteira de fila - primeiro a entrar, primeiro a sair, onde a primeira coisa que eu adiciono à fronteira é a primeira coisa que eu removo. Então, volto para o terminal, executo novamente este programa no mesmo labirinto e agora você verá que o número de estados que tivemos que explorar foi apenas 77, em comparação com quase 400 quando usamos a pesquisa em profundidade. E podemos ver exatamente por quê. Podemos ver o que aconteceu se abrirmos o maze.png agora e dar uma olhada. Novamente, o destaque amarelo é a solução que o BFS encontrou, que, incidentalmente, é a mesma solução que a pesquisa em profundidade encontrou. Eles estão encontrando a melhor solução, mas observe todas as células brancas não exploradas. Havia muito menos estados que precisavam ser explorados para chegarmos à meta, pois o BFS opera um pouco mais superficialmente. Está explorando coisas que estão próximas do estado inicial sem explorar coisas que estão mais distantes. Então, se a meta não estiver muito longe, o BFS pode se comportar muito bem em um labirinto que parece um pouco assim. Agora, neste caso, tanto o BFS quanto o DFS acabaram encontrando a mesma solução, mas isso nem sempre será o caso. E, de fato, vamos dar uma olhada em mais um exemplo, por exemplo, maze3.txt. No maze3.txt, observe que aqui há várias maneiras de se chegar de A a B. É um labirinto relativamente pequeno, mas vamos ver o que acontece. Se eu usar - e eu vou desligar o "show explored" para que vejamos apenas a solução. Se eu usar BFS, pesquisa em largura, para resolver o maze3.txt, bem, então encontramos uma solução. E se eu abrir o labirinto, aqui está a solução que encontramos. É a ótima. Com apenas quatro passos, podemos chegar do estado inicial ao que a meta acaba por ser. Mas o que acontece se tentarmos usar, pesquisa em profundidade, ou DFS, em vez disso? Bem, novamente, eu vou voltar para a minha fronteira de fila, onde a fronteira de fila significa que estamos usando pesquisa em largura. E eu vou mudar para uma fronteira de pilha, o que significa que agora estaremos usando pesquisa em profundidade. Eu vou executar novamente o Pythonmaze.py. E agora você verá que encontramos uma solução, mas não é a solução ótima. Esta, em vez disso, é o que nosso algoritmo encontra. E talvez a pesquisa em profundidade tenha encontrado esta solução. É possível, mas não é garantido, que, se acabarmos por ter azar, se escolhermos este estado em vez desse estado, então a pesquisa em profundidade pode encontrar um caminho mais longo para chegar do estado inicial ao estado de destino. Então, vemos alguns compromissos aqui onde a pesquisa em profundidade pode não encontrar a solução ótima. Então, nesse ponto,

parece que a pesquisa em largura é bastante boa. É isso o melhor que podemos fazer, onde vai nos encontrar a solução ótima e não precisamos nos preocupar com situações em que podemos acabar encontrando um caminho mais longo para a solução do que realmente existe? Onde a meta está longe do estado inicial - e poderíamos ter que dar muitos passos para chegar do estado inicial à meta - o que acabou acontecendo foi que este algoritmo, BFS, acabou explorando basicamente todo o gráfico, tendo que passar por todo o labirinto para encontrar o caminho do estado inicial ao estado de destino. O que gostaríamos, no final, é que nosso algoritmo fosse um pouco mais inteligente. E agora o que significaria para nosso algoritmo ser um pouco mais inteligente, neste caso? Bem, vamos olhar de volta para onde a pesquisa em largura poderia ter tomado uma decisão diferente e considerar a intuição humana neste processo, também. Como, o que um humano poderia fazer ao resolver este labirinto que é diferente do que o BFS acabou escolhendo? Bem, o primeiro ponto de decisão que o BFS fez foi aqui mesmo, quando fez cinco passos e acabou em uma posição onde tinha um beco sem saída. Ele poderia ir para a esquerda ou para a direita. Nesses primeiros passos, não havia escolha. Apenas uma ação poderia ser tomada a partir de cada um desses estados.

Assim, o algoritmo de busca fez o único que qualquer algoritmo de busca poderia fazer, ou seja, continuar seguindo aquele estado após o próximo estado. Mas este ponto de decisão é onde as coisas ficam um pouco interessantes. A busca em profundidade, o primeiro algoritmo de busca que olhamos, escolheu dizer, vamos escolher um caminho e esgotar esse caminho, ver se algo desse jeito tem o objetivo, e se não, vamos tentar o outro caminho. A busca em largura tomou a abordagem alternativa de dizer, você sabe o que? Vamos explorar coisas que são rasas, próximas a nós primeiro, olhar para a esquerda e para a direita, depois para a esquerda e para a direita, assim por diante, alternando entre nossas opções na esperança de encontrar algo próximo. Mas, no final, o que um humano faria se confrontado com uma situação como essa de ir para a esquerda ou para a direita? Bem, um humano pode visualmente ver que, tudo bem, estou tentando chegar ao estado B, que está lá em cima, e ir para a direita parece estar mais perto do objetivo. Como, parece que ir para a direita deve ser melhor do que ir para a esquerda porque estou progredindo para chegar ao objetivo. Agora, é claro, existem algumas suposições que estou fazendo aqui. Estou fazendo a suposição de que podemos representar essa grade como, tipo, uma grade bidimensional, onde eu sei as coordenadas de tudo. Sei que A está na coordenada 0,0 e B está em algum outro par de coordenadas. E sei em que par de coordenadas eu estou agora, então posso calcular que, sim, indo nessa direção, isso está mais perto do objetivo. E isso pode ser uma suposição razoável para alguns tipos de problemas de busca, mas talvez não em outros. Mas por enquanto, vamos assumir que - que eu sei qual é o meu par de coordenadas atual e eu sei a coordenada x, y do objetivo que estou tentando alcançar. E nessa situação, gostaria de um algoritmo um pouco mais inteligente e que de alguma forma saiba que eu deveria estar fazendo progresso em direção ao objetivo, e isso provavelmente é a maneira de fazer isso porque, em um labirinto, se mover na direção de coordenadas do objetivo geralmente, embora nem sempre, é uma boa coisa. E então aqui fazemos uma distinção entre dois tipos diferentes de algoritmos de busca - busca não informada e busca informada. Os algoritmos de busca não informados são algoritmos como DFS e BFS, os dois algoritmos que acabamos de olhar, que são estratégias de busca que não usam nenhum conhecimento específico do problema para poder resolver o problema. DFS e BFS realmente não se importavam com a estrutura do labirinto ou qualquer coisa sobre a maneira como um labirinto é para resolver o problema. Eles apenas olham para as ações disponíveis e escolhem entre essas ações, e não importa se é um labirinto ou algum outro problema. A solução, ou a maneira

como ele tenta resolver o problema, realmente será basicamente a mesma. O que vamos olhar agora é uma melhoria na busca não informada. Vamos dar uma olhada na busca informada. As buscas informadas serão estratégias de busca que usam conhecimento específico do problema para poder encontrar melhor uma solução. E no caso de um labirinto, esse conhecimento específico do problema é algo como, se eu for para o quadrado que está geograficamente mais próximo do objetivo, isso é melhor do que estar em um quadrado que está geograficamente mais longe. E isso só podemos saber pensando sobre esse problema e raciocinando sobre qual conhecimento pode ser útil para o nosso agente de IA saber um pouco sobre. Existem vários tipos diferentes de busca informada. Especificamente, primeiro, vamos olhar para um tipo específico de algoritmo de busca chamado busca gulosa de melhor primeiro. A Busca Gulosa de Melhor Primeiro, frequentemente abreviada como GBFS, é um algoritmo de busca que, em vez de expandir o nó mais profundo, como DFS, ou o nó mais superficial, como BFS, este algoritmo sempre vai expandir o nó que ele acha que está mais próximo do objetivo. Agora, o algoritmo de busca não vai saber com certeza se é o mais próximo do objetivo, porque se soubesse o que estava mais próximo do objetivo o tempo todo, já teríamos uma solução.

Como o conhecimento do que está próximo ao objetivo, poderíamos simplesmente seguir esses passos para chegar da posição inicial à solução. Mas se não conhecemos a solução - o que significa que não sabemos exatamente o que está mais próximo do objetivo -, podemos usar uma estimativa do que está mais próximo do objetivo, conhecido como heurística - apenas alguma forma de estimar se estamos próximos ou não do objetivo. E faremos isso usando uma função heurística, convencionalmente chamada h (n), que recebe uma entrada de estado e retorna nossa estimativa de quão próximos estamos do objetivo. Então, como seria essa função heurística na verdade no caso de um algoritmo de resolução de labirinto? Onde estamos tentando resolver um labirinto, qual é a aparência de uma heurística? Bem, a heurística precisa responder a uma pergunta, como entre essas duas células, C e D, qual é melhor? Qual eu prefiro estar se estou tentando encontrar o meu caminho para o objetivo? Bem, qualquer humano provavelmente poderia olhar isso e dizer, você sabe o que? D parece ser melhor. Mesmo se o labirinto for complicado e você não tenha pensado em todas as paredes, D provavelmente é melhor. E por que D é melhor? Bem, porque se você ignorar a parede - vamos apenas fingir que as paredes não existem por um momento e relaxar o problema, digamos assim - D, apenas em termos de pares de coordenadas, está mais próximo deste objetivo. É menos passos que eu precisaria dar para chegar ao objetivo, em comparação com C, mesmo se você ignorar as paredes. Se você souber apenas o x, y da coordenada de C, e o x, y da coordenada do objetivo, e da mesma forma, você sabe o x, y da coordenada de D, você pode calcular que D, apenas geograficamente, ignorando as paredes, parece ser melhor. E então essa é a função heurística que vamos usar, e é algo chamado distância de Manhattan, um tipo específico de heurística, onde a heurística é, quantos quadrados verticalmente e horizontalmente e então da esquerda para a direita - então não me permitindo ir diagonalmente, apenas para cima ou para a direita ou para a esquerda ou para baixo. Quantos passos eu preciso dar para chegar de cada uma dessas células ao objetivo? Bem, como se mostra, D está muito mais próximo. São menos passos. Só precisa levar seis passos para chegar ao objetivo. Novamente, aqui ignorando as paredes. Relaxamos um pouco o problema. Estamos apenas preocupados em, se você fizer as contas, subtrair os valores x um do outro e os valores y um do outro, qual é a nossa estimativa de quão longe estamos? Podemos estimar que D está mais próximo do objetivo do que C. E então agora temos uma abordagem. Temos uma forma de escolher qual nó remover da fronteira. E em cada etapa do nosso algoritmo, vamos remover um nó da fronteira. Vamos explorar o nó, se ele tiver o menor valor para essa função heurística, se ele tiver a menor distância de Manhattan para o objetivo. E

então, o que isso realmente pareceria? Bem, primeiro me permita rotular este gráfico, rotular este labirinto, com um número que representa o valor desta função heurística, o valor da distância de Manhattan de qualquer uma dessas células. Então, desta célula, por exemplo, estávamos a uma distância do objetivo. Desta célula, estávamos a duas distâncias do objetivo. Três distâncias, quatro distâncias. Aqui estamos a cinco distâncias, porque temos que ir para a direita e depois para cima. De algum lugar como aqui, a distância de Manhattan é 2. Estamos apenas a dois quadrados do objetivo, geograficamente, embora na prática teremos que tomar um caminho mais longo, mas não sabemos disso ainda. A heurística é apenas alguma forma fácil de estimar quanto estamos longe do objetivo. E talvez nossa heurística seja excessivamente otimista. Ela pensa que sim, estamos apenas a dois passos de distância, quando na prática, quando você considera as paredes, pode ser mais passos. Então, a coisa importante aqui é que a heurística não é uma garantia de quantos passos serão necessários. É uma estimativa. É uma tentativa de tentar aproximar. E parece geralmente o caso de que os quadrados que parecem mais próximos do objetivo têm valores menores para a função heurística do que os quadrados mais distantes. Então, usando a busca gulosa de melhor primeiro, o que esse algoritmo realmente faria?

Bem, novamente, para essas primeiras cinco etapas, não há muita escolha. Nós começamos esse estado inicial, A. E dizemos, tudo bem. Temos que explorar esses cinco estados. Mas agora temos um ponto de decisão. Agora temos uma escolha entre ir para a esquerda e ir para a direita. E antes, quando o DFS e o BFS escolheriam arbitrariamente porque depende da ordem em que você lança esses dois nós na fronteira - e não especificamos em que ordem você os coloca na fronteira, apenas a ordem em que os tira. Aqui podemos olhar para 13 e 11 e dizer, tudo bem, este quadrado está a uma distância de 11 do objetivo, de acordo com nossa heurística, de acordo com nossa estimativa. E esse aqui estimamos que está a 13 do objetivo. Então, entre essas duas opções, entre essas duas escolhas, eu prefiro o 11. Eu prefiro estar a 11 passos do objetivo, então vou para a direita. Somos capazes de tomar uma decisão informada porque sabemos um pouco mais sobre este problema. Então, então seguimos 10, 9, 8 - entre os dois setes. Na verdade, não temos muito jeito de saber entre eles. Então, então temos que fazer uma escolha arbitrária. E você sabe o que? Talvez a gente escolha errado. Mas tudo bem porque agora ainda podemos dizer, tudo bem, vamos tentar esse sete. Dizemos sete, seis. Temos que fazer essa escolha, mesmo que isso aumente o valor da função heurística. Mas agora temos outro ponto de decisão entre seis e oito. E entre esses dois - e realmente, também estamos considerando o 13, mas esse é muito maior. Entre seis, oito e 13, bem, o seis é o menor valor, então preferimos o seis. Somos capazes de tomar uma decisão informada de que ir por esse caminho à direita é provavelmente melhor do que ir por aquele caminho. Então, vamos por esse caminho. Vamos para cinco. E agora encontramos um ponto de decisão onde realmente vamos tomar uma decisão que talvez não queiramos tomar, mas infelizmente não há muito jeito de contornar isso. Vejamos quatro e seis. Quatro parece mais perto do objetivo, certo? Está subindo, e o objetivo está mais para cima. Então acabamos por seguir essa rota, que nos leva a um beco sem saída. Mas tudo bem porque ainda podemos dizer, tudo bem, agora vamos tentar o seis e agora seguir esta rota que nos levará ao objetivo. E assim é como o greedy best-first search pode tentar abordar este problema, dizendo sempre que temos uma decisão entre vários nós que poderíamos explorar, vamos explorar o nó que tem o menor valor de h (n), esta função heurística que está estimando quanto eu tenho que andar. E acontece que, neste caso, acabamos por fazer melhor, em termos de número de estados que precisamos explorar, do que o BFS precisava. O BFS explorou toda essa seção e toda aquela seção. Mas conseguimos eliminar isso aproveitando essa heurística, essa informação sobre quão perto estamos do objetivo ou alguma estimativa dessa ideia. Então isso parece muito melhor. Então, não preferiríamos sempre um algoritmo

como este a um algoritmo como a busca em largura? Bem, talvez. Uma coisa a considerar é que precisamos criar uma boa heurística. Quão boa é a heurística vai afetar quão bom é este algoritmo. E criar uma boa heurística às vezes pode ser desafiador. Mas a outra coisa a considerar é perguntar, assim como fizemos com os dois algoritmos anteriores, se este algoritmo é ótimo? Ele sempre encontrará o caminho mais curto do estado inicial ao objetivo? E para responder a essa pergunta, vamos olhar para este exemplo por um momento. Dê uma olhada neste exemplo. Novamente, estamos tentando chegar de A a B, e novamente, etiquetei cada uma das células com sua distância de Manhattan do objetivo, o número de quadrados para cima e para a direita que você precisaria percorrer para chegar daquela quadrado ao objetivo. E vamos pensar, o greedy best-first search que sempre escolhe o menor número acabaria encontrando a solução ótima? Qual é o caminho mais curto e esse algoritmo o encontraria? E a coisa importante a perceber é que aqui está o ponto de decisão.

Estimamos estar a 12 passos do objetivo. E temos duas opções. Podemos ir para a esquerda, que estimamos estar a 13 passos do objetivo, ou podemos subir, onde estimamos estar a 11 passos do objetivo. E entre essas duas, a busca gulosa de melhor primeiro diz: o 11 parece melhor do que o 13. E fazendo isso, a busca gulosa de melhor primeiro acabará por encontrar este caminho para o objetivo. Mas acontece que este caminho não é otimizado. Há uma maneira de chegar ao objetivo usando menos passos. E é na verdade desta forma, desta forma que, no final, envolveu menos passos, mesmo que, neste momento, escolher a pior opção entre as duas - ou o que estimamos ser a pior opção, com base nos hereges. E é assim que queremos dizer que este é um algoritmo guloso. Está tomando a melhor decisão, localmente. Neste ponto de decisão, parece ser melhor ir para cá do que ir para o 13. Mas no grande quadro, nem sempre é otimizado, pois pode encontrar uma solução quando, na verdade, havia uma solução melhor disponível. Então, gostaríamos de alguma forma de resolver este problema. Gostamos da ideia desta heurística, de ser capaz de estimar o caminho, a distância entre nós e o objetivo, e isso nos ajuda a tomar melhores decisões e a eliminar a necessidade de pesquisar por todas as partes do espaço de estado. Mas gostaríamos de modificar o algoritmo para que possamos alcançar a otimização, para que possa ser otimizado. E qual é a maneira de fazer isso? Qual é a intuição aqui? Bem, vamos dar uma olhada neste problema. Neste problema inicial, a busca gulosa de melhor primeiro encontrou esta solução aqui, este caminho longo. E a razão pela qual não foi ótimo é porque, sim, os números heurísticos caíram bastante, mas mais tarde, e eles começaram a se recuperar. Eles subiram 8, 9, 10, 11 - até 12, neste caso. Então, como podemos tentar melhorar este algoritmo? Bem, uma coisa que podemos perceber é que, se passarmos por todo este algoritmo, por este caminho, e acabarmos indo para o 12, e tivemos que dar tantos passos - como quem sabe quantos passos isso é - apenas para chegar a este 12, também poderíamos, como alternativa, ter tomado muito menos passos, apenas seis passos, e acabar neste 13 aqui. E sim, 13 é mais do que 12, então parece que não é tão bom, mas exigiu muito menos passos. Certo? Só levou seis passos para chegar a este 13 em vez de muitos mais passos para chegar a este 12. E, enquanto a busca gulosa de melhor primeiro diz: ah, bem, 12 é melhor do que 13, então escolha o 12, nós podemos dizer de forma mais inteligente: prefiro estar em algum lugar que heuristicamente parece levar um pouco mais se eu puder chegar lá muito mais rapidamente. E vamos codificar essa ideia, essa ideia geral, em um algoritmo mais formal conhecido como busca A estrela. A busca A estrela vai resolver este problema, ao invés de considerar apenas a heurística, também considerando quanto tempo levou para chegar a qualquer estado. Então, a distinção é a busca gulosa de melhor primeiro, se eu estiver em um estado agora, a única coisa que me importa é qual é a distância estimada, o valor heurístico, entre mim e o objetivo. Enquanto a busca A estrela levará em consideração duas peças de informação. Vai levar em consideração, quanto eu estimo estar longe do objetivo, mas

também quanto eu tive que viajar para chegar aqui? Porque isso também é relevante. Então, vamos procurar algoritmos expandindo o nó com o menor valor de g (n) mais h (n). h (n) é o mesmo heurístico sobre o qual estávamos falando há alguns momentos que vai variar com base no problema, mas g (n) será o custo para alcançar o nó - quantos passos eu tive que dar, neste caso, para chegar à minha posição atual. Então, como é que esse algoritmo de busca se parece na prática? Bem, vamos dar uma olhada. Novamente, temos o mesmo labirinto. E novamente, eu os rotulei com sua distância de Manhattan. Este valor é o valor h (n), a estimativa heurística de quantos quadrados estão distantes do objetivo.

Agora, à medida que começamos a explorar estados, não nos preocupamos apenas com esse valor heurístico, mas também com g(n), o número de passos que tive que dar para chegar lá. E me preocupo em somar esses dois números. Então, como isso se parece? Nesse primeiro passo, eu já dei um passo. E agora eu estou estimado a estar a 16 passos do objetivo. Então, o valor total aqui é 17. Então, eu dou mais um passo. Agora eu já dei dois passos. E eu me estimo a estar a 15 passos do objetivo - novamente, um valor total de 17. Agora eu já dei três passos. E eu me estimo a estar a 14 passos do objetivo, e assim por diante. Quatro passos, uma estimativa de 13. Cinco passos, estimativa de 12. E agora, aqui está um ponto de decisão. Eu poderia estar a seis passos do objetivo com uma heurística de 13 para um total de 19, ou eu poderia estar a seis passos do objetivo com uma heurística de 11 com uma estimativa de 17 para o total. Então, entre 19 e 17, eu prefiro pegar o 17 - o 6 mais 11. Então, até agora, nada diferente do que vimos antes. Ainda estamos tomando essa opção porque parece ser melhor. E continuo tomando essa opção porque parece ser melhor. Mas é aí que as coisas ficam um pouco diferentes. Agora eu poderia estar a 15 passos do objetivo com uma distância estimada de 6. Então, 15 mais 6, valor total de 21. Alternativamente, eu poderia estar a seis passos do objetivo - porque isso estava a cinco passos de distância, então isso está a seis passos de distância - com um valor total de 13 como minha estimativa. Então, 6 mais 13 - isso é 19. Então, aqui avaliaríamos g (n) mais h (n) para ser 19 - 6 mais 13 - enquanto aqui seríamos 15 mais 6, ou 21. E então a intuição é, 19 menos que 21, escolha aqui. Mas a ideia é, no final, eu prefiro ter tomado menos passos para chegar a 13 do que ter tomado 15 passos e estar em seis, porque isso significa que tive que dar mais passos para chegar lá. Talvez haja um caminho melhor por aqui. Então, em vez disso, vamos explorar este caminho. Agora, se formos mais um - isso é sete passos mais 14, é 21, então entre esses dois é meio que um empate. Talvez acabemos explorando isso de qualquer maneira. Mas depois disso, à medida que esses números começam a ficar maiores nos valores heurísticos e esses valores heurísticos começam a ficar menores, você verá que na verdade vamos continuar explorando por esse caminho. E você pode fazer as contas para ver que, em cada ponto de decisão, a pesquisa A estrela vai tomar uma decisão com base na soma de quantos passos eu tive que dar para chegar à minha posição atual e então quanto eu estimo que estou longe do objetivo. Então, embora tivéssemos que explorar alguns desses estados, a solução final que encontramos foi, de fato, uma solução ótima. Ele realmente nos encontrou o caminho mais rápido possível para chegar do estado inicial ao objetivo. E resulta que o A * é um algoritmo de busca ótimo sob determinadas condições. Então, quais são as condições? h de n, minha heurística, precisa ser admissível. O que significa que uma heurística seja admissível? Bem, uma heurística é admissível se ela nunca subestima o custo real. Cada evento sempre precisa acertar exatamente a distância ou subestimá-la. Vimos um exemplo antes em que o valor heurístico era muito menor do que o custo real que levaria. Isso está totalmente bem. Mas o valor heurístico nunca deve subestimar. Nunca deve pensar que estou mais longe do objetivo do que realmente estou. E, ao mesmo tempo, para fazer uma declaração mais forte, h de n

também precisa ser consistente. E o que significa ser consistente? Matematicamente, significa que, para cada nó, que chamaremos de n, e sucessor, o nó depois de mim, que chamaremos de n primo, onde custa c fazer esse passo, o valor heurístico de n precisa ser menor ou igual ao valor heurístico de n primo mais o custo. Então, é muita matemática, mas em palavras, o que significa, no final, é que se eu estiver aqui neste estado agora, o valor heurístico de mim para o objetivo não deve ser mais do que o valor heurístico do meu sucessor, o próximo lugar aonde eu poderia ir, mais o quanto custaria para apenas fazer esse passo, de um passo para o próximo passo.

Então, isso é apenas para garantir que meu heurístico seja consistente entre todos esses passos que eu possa tomar. Desde que isso seja verdade, então a busca A * vai me encontrar uma solução ótima. E aqui é onde muito do desafio de resolver esses problemas de busca às vezes pode entrar, que a busca A * é um algoritmo conhecido e você poderia escrever o código com relativa facilidade. Mas é escolher o heurístico que pode ser o desafio interessante. Quanto melhor o heurístico for, melhor eu poderei resolver o problema e menos estados que eu terei que explorar. E eu preciso me certificar de que o heurístico satisfaça essas restrições específicas. Então, no geral, esses são alguns dos exemplos de algoritmos de busca que podem funcionar. E certamente, há muito mais do que isso. A *, por exemplo, tende a usar bastante memória, então há abordagens alternativas ao A * que, no final, usam menos memória do que essa versão do A * acaba por usar. E há outros algoritmos de busca que são otimizados para outros casos também. Mas agora, até agora, só estamos olhando para algoritmos de busca onde há um único agente. Estou tentando encontrar uma solução para um problema. Estou tentando navegar pelo meu caminho através de um labirinto. Estou tentando resolver um quebra-cabeça de 15 peças. Estou tentando encontrar direções de condução de um ponto A a um ponto B. Às vezes, em situações de busca, entramos em uma situação adversarial onde eu sou um agente tentando tomar decisões inteligentes e há alguém mais lutando contra mim, para falar, que tem um objetivo oposto, alguém onde eu estou tentando ter sucesso, alguém mais que quer que eu falhe. E isso é mais popular em algo como um jogo, um jogo como jogo da velha, onde temos essa grade 3x3 e X e O tomam turnos escrevendo um X ou um O em qualquer uma dessas quadras. E o objetivo é conseguir três X em uma linha, se você for o jogador X, ou três O em uma linha, se você for o jogador O. E os computadores ficaram muito bons em jogar jogos, jogo da velha muito facilmente, mas até jogos mais complexos. E então você pode imaginar, qual é a decisão inteligente em um jogo? Então, talvez X faça uma jogada inicial no meio e O jogue aqui em cima. Qual é a jogada inteligente para X agora? Onde você deve se mover se você fosse X? E resulta que há algumas possibilidades. Mas se uma IA estiver jogando este jogo de forma ótima, então a IA pode jogar algum lugar como o canto superior direito, onde nesta situação, O tem o objetivo oposto de X. X está tentando ganhar o jogo, para conseguir três em uma linha diagonalmente aqui, e O está tentando impedir esse objetivo, oposto ao objetivo. E então O vai colocar aqui, para tentar bloquear. Mas agora, X tem uma jogada bastante inteligente. X pode fazer uma jogada, assim, onde agora X tem duas maneiras possíveis de ganhar o jogo. X poderia ganhar o jogo conseguindo três em uma linha horizontalmente aqui, ou X poderia ganhar o jogo conseguindo três em uma linha verticalmente desta forma. Então não importa onde O faça sua próxima jogada. O poderia jogar aqui, por exemplo, bloqueando os três em uma linha horizontalmente, mas então X vai ganhar o jogo conseguindo um três em uma linha verticalmente. E então há uma quantidade razoável de raciocínio que está acontecendo aqui para que o computador possa resolver um problema. E é similar em espírito aos problemas que já olhamos até agora. Há ações, há algum tipo de estado do tabuleiro e alguma transição de uma ação para a próxima, mas é diferente no sentido de que isso agora não é apenas um problema de busca clássico, mas um problema de busca adversário, que eu sou o jogador X, tentando encontrar as melhores jogadas para fazer, mas eu sei que há algum adversário

que está tentando me impedir. Então precisamos de algum tipo de algoritmo para lidar com essas situações de busca adversárias. E o algoritmo que vamos olhar é um algoritmo chamado Minimax, que funciona muito bem para esses jogos determinísticos, onde há dois jogadores.

Pode funcionar para outros tipos de jogos também, mas vamos olhar agora para jogos onde eu faço uma jogada, meu oponente faz uma jogada e eu estou tentando ganhar, e meu oponente também está tentando ganhar. Ou, em outras palavras, meu oponente está tentando me fazer perder. Então, o que precisamos para fazer este algoritmo funcionar? Bem, sempre que tentamos traduzir este conceito humano de jogar um jogo, ganhar e perder, para um computador, queremos traduzi-lo em termos que o computador possa entender. E, no final das contas, o computador realmente só entende números. Então, queremos alguma forma de traduzir um jogo de X's e O's em uma grade para algo numérico, algo que o computador possa entender. O computador normalmente não entende noções de ganhar ou perder, mas entende o conceito de maior e menor. Então, o que podemos fazer é, podemos pegar cada uma das possíveis formas como um jogo de velha pode se desenrolar e atribuir um valor, ou uma utilidade, a cada um desses possíveis resultados. E em um jogo de velha, e em muitos tipos de jogos, há três possíveis resultados. Os resultados são, O ganha, X ganha ou ninguém ganha. Então, jogador um ganha, jogador dois ganha ou ninguém ganha. E por enquanto, vamos atribuir cada um desses possíveis resultados a um valor diferente. Vamos dizer que O ganhando - isso terá um valor de -1. Ninguém ganhando - isso terá um valor de 0. E X ganhando - isso terá um valor de 1. Então, acabamos de atribuir números a cada um desses três possíveis resultados. E agora, temos dois jogadores. Temos o jogador X e o jogador O. E vamos chamar o jogador X de jogador max. E vamos chamar o jogador O de jogador min. E a razão é que, no algoritmo Minimax, o jogador max, que neste caso é X, está tentando maximizar a pontuação. Estas são as opções possíveis para a pontuação, -1, 0 e 1. X quer maximizar a pontuação, o que significa que, se possível, X gostaria desta situação em que X ganha o jogo. E damos a ele uma pontuação de 1. Mas se isso não for possível, se X precisar escolher entre essas duas opções, -1 significando O ganhando, ou 0 significando ninguém ganhando, X preferiria que ninguém ganhe, pontuação de 0, do que uma pontuação de -1, O ganhando. Então, esta noção de ganhar e perder no tempo foi reduzida matematicamente a apenas esta ideia de tentar maximizar a pontuação. O jogador X sempre quer que a pontuação seja maior. E, do outro lado, o jogador min, neste caso O, está tentando minimizar a pontuação. O jogador O quer que a pontuação seja o menor possível. Então, agora, transformamos este jogo de X's e O's e ganhar e perder em algo matemático, algo onde X está tentando maximizar a pontuação, O está tentando minimizar a pontuação. Vamos agora olhar todas as partes do jogo que precisamos para codificá-lo em uma IA para que uma IA possa jogar um jogo como o jogo da velha. Então, o jogo vai precisar de algumas coisas. Vamos precisar de algum estado inicial, que vamos chamar aqui de S0, que é como o jogo começa, como uma tabela de jogo da velha vazia, por exemplo. Vamos também precisar de uma função chamada jogador, onde a função jogador vai receber como entrada um estado, aqui representado por S. E a saída da função jogador vai ser, de quem é a vez? Precisamos poder dar uma tabela de jogo da velha para o computador, executá-la por meio de uma função e essa função nos diz de quem é a vez. Precisamos de alguma noção de ações que podemos tomar. Veremos exemplos disso em breve. Precisamos de alguma noção de um modelo de transição - igual ao anterior. Se eu tiver um estado e tomar uma ação, preciso saber qual é o resultado como consequência disso. Preciso de alguma forma de saber quando o jogo acabou. Então, isso é equivalente a algo como um teste de objetivo, mas preciso de algum teste terminal, alguma forma de verificar se um estado é um estado terminal, onde um estado terminal significa que o jogo acabou. No jogo clássico da velha, um estado terminal significa que alguém

conseguiu três em uma linha ou todos os quadrados da tabela da velha estão preenchidos.

Qualquer uma dessas condições torna-o um estado terminal. Em um jogo de xadrez, isso poderia ser algo como, quando há xeque-mate, ou se o xeque-mate não é mais possível, isso se torna um estado terminal. E finalmente precisaremos de uma função de utilidade, uma função que toma um estado e nos dá um valor numérico para esse estado terminal, alguma maneira de dizer, se X ganhar o jogo, isso tem um valor de 1. Se O ganhar o jogo, isso tem o valor de -1. Se ninguém ganhar o jogo, isso tem o valor de 0. Então vamos dar uma olhada em cada um deles. O estado inicial, podemos representar no jogo da velha como o tabuleiro de jogo vazio. É a partir daqui que começamos. É o lugar a partir do qual começamos essa busca. E novamente, eu representarei essas coisas visualmente. Mas você pode imaginar isso realmente sendo apenas uma matriz, ou uma matriz bidimensional, de todos esses quadrados possíveis. Então precisamos da função jogador que, novamente, toma um estado e nos diz de quem é a vez. Supondo que X faça o primeiro movimento, se eu tiver um tabuleiro de jogo vazio, então minha função jogador vai retornar X E se eu tiver um tabuleiro de jogo onde X fez um movimento, minha função jogador vai retornar O. A função jogador toma um tabuleiro de jogo da velha e nos diz de quem é a vez. Em seguida, consideraremos a função ações. A função ações, assim como aconteceu na busca clássica, toma um estado e nos dá o conjunto de todas as ações possíveis que podemos tomar nesse estado. Então vamos imaginar que é a vez de O se mover em um tabuleiro de jogo que parece assim. O que acontece quando passamos isso para a função ações? Então a função ações toma esse estado do jogo como entrada e a saída é um conjunto de ações possíveis é um conjunto de - eu poderia me mover no canto superior esquerdo, ou eu poderia me mover no meio inferior. Essas são as duas escolhas de ação possíveis que eu tenho quando começo nesse estado particular. Agora, assim como antes, quando adicionamos estados e ações, precisamos de algum tipo de modelo de transição para nos dizer, quando tomamos essa ação no estado, qual é o novo estado que obtemos? E aqui, definimos isso usando a função resultado que toma como entrada um estado, bem como uma ação. E quando aplicamos a função resultado a este estado, dizendo que vamos deixar que O se mova neste canto superior esquerdo, o novo estado que obtemos é este estado resultante, onde O está no canto superior esquerdo. E agora, isso parece óbvio para alguém que sabe como jogar jogo da velha. Claro, você joga no canto superior esquerdo - é o tabuleiro que você obtém. Mas todas essas informações precisam ser codificadas na IA. A IA não sabe como jogar jogo da velha até que você diga à IA como as regras da velha funcionam. E essa função, definindo a função aqui, nos permite dizer à IA como esse jogo realmente funciona e como as ações realmente afetam o resultado do jogo. Então a IA precisa saber como o jogo funciona. A IA também precisa saber quando o jogo acabou. Isso é definindo uma função chamada terminal que toma como entrada um estado S, de modo que, se tomarmos um jogo que ainda não acabou, passá-lo para a função terminal, a saída é falsa. O jogo não acabou. Mas se tomarmos um jogo que acabou, porque X conseguiu três em uma fileira ao longo dessa diagonal, passar isso para a função terminal, então a saída será verdadeira, porque o jogo agora, de fato, acabou. E finalmente, dissemos à IA como o jogo funciona em termos de quais movimentos podem ser feitos e o que acontece quando você faz esses movimentos. Dissemos à IA quando o jogo acabou. Agora precisamos dizer à IA qual é o valor de cada um desses estados. E fazemos isso definindo essa função de utilidade, que toma um estado, S, e nos diz a pontuação ou a utilidade desse estado. Então, novamente, dissemos que, se X ganhar o jogo, essa utilidade tem um valor de 1, enquanto que, se O ganhar o jogo, então a utilidade disso é -1. E a IA precisa saber, para cada um desses estados terminais onde o jogo acabou, qual é a utilidade desse estado? Então posso dar a você um tabuleiro de jogo como este, onde o jogo, de fato, acabou, e perguntar à IA qual é o valor desse estado, ela poderia fazê-lo.

O valor do estado é 1. No entanto, onde as coisas ficam interessantes é se o jogo ainda não acabou. Vamos imaginar um tabuleiro de jogo assim. Estamos no meio do jogo. É a vez do O jogar. Então, como sabemos que é a vez do O jogar? Podemos calcular isso, usando a função jogador. Podemos dizer, jogador de S, passe no estado. O é a resposta, então sabemos que é a vez do O jogar. E agora, qual é o valor deste tabuleiro e qual ação o O deve tomar? Bem, isso vai depender. Temos que fazer alguns cálculos aqui. E é aqui que o algoritmo Minimax realmente entra em cena. Lembre-se de que o X está tentando maximizar a pontuação, o que significa que o O está tentando minimizar a pontuação. O gostaria de minimizar o valor total que obtemos no final do jogo. E porque este jogo ainda não acabou, ainda não sabemos exatamente qual é o valor deste tabuleiro. Temos que fazer alguns cálculos para descobrir isso. Então, como fazemos esse tipo de cálculo? Bem, para isso, vamos considerar, assim como faríamos em uma situação de pesquisa clássica, quais ações podem acontecer a seguir e quais estados isso nos levará? E resulta que nesta posição, só há dois quadrados abertos, o que significa que só há duas posições abertas onde o O pode jogar. O pode jogar no canto superior esquerdo ou no meio inferior. E o Minimax não sabe de cara qual desses movimentos será melhor, então ele vai considerar os dois. Mas agora nos deparamos com a mesma situação. Agora eu tenho mais dois tabuleiros de jogo, nenhum dos quais está terminado. O que acontece a seguir? E agora é nesse sentido que o Minimax é o que chamamos de algoritmo recursivo. Ele vai agora repetir o mesmo processo exato, embora agora considerando-o do ponto de vista oposto. É como se eu estivesse agora me colocando - se eu sou o jogador O, vou me colocar nos sapatos do meu oponente, meu oponente como jogador X, e considerar, o que meu oponente faria se estivesse nessa posição? O que meu oponente faria, o jogador X, se estivesse nessa posição? E o que aconteceria então? Bem, o outro jogador, meu oponente, o jogador X, está tentando maximizar a pontuação, enquanto eu, como jogador O, estou tentando minimizar a pontuação. Então, o X está tentando encontrar o valor máximo que eles podem obter. E então o que vai acontecer? Bem, a partir deste tabuleiro, o X só tem uma escolha. O X vai jogar aqui e eles vão conseguir três em linha. E sabemos que esse tabuleiro, o X ganhando - isso tem um valor de 1. Se o X ganhar o jogo, o valor desse tabuleiro de jogo é 1. E então, a partir desta posição, se este estado só pode levar a este estado, é a única opção possível e este estado tem um valor de 1, então o valor máximo que o jogador X pode obter deste tabuleiro de jogo também é 1 daqui. O único lugar aonde podemos chegar é a um jogo com o valor de 1, então este tabuleiro de jogo também tem um valor de 1. Agora, consideramos este aqui. O que vai acontecer agora? Bem, o X precisa fazer um movimento. A única jogada que o X pode fazer é no canto superior esquerdo, então o X vai lá. E neste jogo, ninguém ganha o jogo. Ninguém tem três em linha. Então, o valor desse tabuleiro de jogo é 0. Ninguém ganhou. E então, novamente, pelo mesmo raciocínio, se deste tabuleiro, o único lugar aonde podemos chegar é a um tabuleiro onde o valor é 0, então este estado também deve ter um valor de 0. E agora vem a parte da escolha, a ideia de tentar minimizar. Eu, como jogador O, agora sei que se eu fizer essa escolha, me movendo no canto superior esquerdo, isso vai resultar em um jogo com um valor de 1, assumindo que todos joguem de forma otimizada. E se eu jogar no meio inferior, escolher essa bifurcação na estrada, isso vai resultar em um tabuleiro de jogo com um valor de 0. Eu tenho duas opções. Eu tenho um 1 e um 0 para escolher, e eu preciso escolher. E como jogador min, eu prefiro escolher a opção com o valor mínimo. Então, sempre que um jogador tiver múltiplas escolhas, o jogador min escolherá a opção com o menor valor. O jogador max escolherá a opção com o maior valor.

Entre o 1 e o 0, o 0 é menor, o que significa que eu prefiro empatar o jogo a perder. E então, este tabuleiro de jogo, vamos dizer, também tem um valor de 0, porque se eu estiver jogando de forma ótima, eu escolherei este caminho. Eu colocarei meu O aqui para bloquear o três em linha do X. O X se moverá no canto superior esquerdo e o jogo acabará, e ninguém ganhará o jogo. Então, este é o raciocínio do Minimax, considerar todas as opções possíveis que eu posso tomar, todas as ações que eu posso tomar e, em seguida, colocar-me nos sapatos do meu oponente. Eu decido qual movimento vou fazer agora, considerando qual movimento meu oponente fará na próxima rodada. E para isso, eu considero qual movimento eu faria na rodada seguinte, assim por diante, até chegar ao final do jogo, a um desses chamados estados terminais. Na verdade, este ponto de decisão, onde eu estou tentando decidir como o jogador O o que decidir, pode ter sido apenas uma parte do raciocínio que o jogador X, meu oponente, estava usando na jogada anterior. Isso pode ser parte de uma árvore maior onde o X está tentando fazer um movimento nesta situação e precisa escolher entre três opções diferentes para tomar uma decisão sobre o que acontecer. E quanto mais longe estamos do final do jogo, mais profunda tem que ser essa árvore, porque cada nível nesta árvore corresponderá a um movimento, um movimento ou ação que eu tomo, um movimento ou ação que meu oponente toma, para decidir o que acontece. E na verdade, descobre-se que, se eu for o jogador X nesta posição e eu faço o raciocínio recursivamente e vejo que tenho uma escolha - três escolhas, na verdade, uma das quais leva a um valor de 0, se eu jogar aqui, e se todos jogarem de forma ótima, o jogo será um empate. Se eu jogar aqui, então O vai ganhar e eu vou perder, jogando de forma ótima. Ou aqui, onde eu, o jogador X, posso ganhar - bem, entre uma pontuação de 0 e -1 e 1, eu prefiro escolher o tabuleiro com um valor de 1, porque é o valor máximo que eu posso obter. E então, este tabuleiro também teria um valor máximo de 1. E então, esta árvore pode ficar muito, muito profunda, especialmente à medida que o jogo começa a ter cada vez mais movimentos. E este raciocínio funciona não apenas para o jogo da velha, mas para qualquer um destes tipos de jogos onde eu faço um movimento, meu oponente faz um movimento e, eventualmente, temos esses objetivos adversários. E podemos simplificar o diagrama em um diagrama que parece assim. Esta é uma versão mais abstrata da árvore Minimax, onde esses são cada estado, mas eu não estou mais representando-os exatamente como tabuleiros de jogo da velha. Isso apenas representa algum jogo genérico que pode ser jogo da velha, pode ser algum outro jogo. Qualquer uma dessas setas verdes apontando para cima - isso representa um estado de maximização. Eu gostaria que a pontuação fosse o mais alta possível. E qualquer uma dessas setas vermelhas apontando para baixo - esses são estados de minimização, onde o jogador é o jogador min e eles estão tentando fazer a pontuação o mais baixa possível. Então, se você imaginar nesta situação, eu sou o jogador maximizador, este jogador aqui, e eu tenho três escolhas - uma escolha me dá uma pontuação de 5, uma escolha me dá uma pontuação de 3 e uma escolha me dá uma pontuação de 9. Bem, então, entre essas três escolhas, minha melhor opção é escolher este 9 aqui, a pontuação que maximiza minhas opções de todas as três opções. E então, eu posso dar a este estado um valor de 9, porque entre minhas três opções, essa é a melhor escolha que eu tenho disponível para mim. Então, essa é minha decisão agora. Você imagina que é como um movimento longe do final do jogo. Mas então você também pode fazer uma pergunta razoável. O que meu oponente pode fazer dois movimentos longe do final do jogo? Meu oponente é o jogador minimizador. Eles estão tentando fazer a pontuação o mais baixa possível. Imagine o que teria acontecido se eles tivessem que escolher qual escolha fazer. Uma escolha nos leva a este estado, onde eu, o jogador maximizador, vou optar por 9, a maior pontuação que eu posso obter.

Um leva a este estado, onde eu, o jogador maximizador, escolheria 8, que é então a maior pontuação que eu

posso obter. Agora, o jogador minimizador, forçado a escolher entre um 9 ou um 8, vai escolher a menor pontuação possível, que neste caso é um 8. E isso é, então, como esse processo se desdobraria. Mas o jogador minimizador, neste caso, considera ambas as suas opções e, em seguida, todas as opções que aconteceriam como resultado disso. Então, isso agora é uma imagem geral do que o algoritmo Minimax parece. Vamos agora tentar formalizá-lo usando um pouco de pseudocódigo. Então, o que exatamente está acontecendo no algoritmo Minimax? Bem, dado um estado, S, precisamos decidir o que acontecer. O jogador max - se for a vez do jogador max, então max vai escolher uma ação, A, em ações de S. Lembre-se de que ações é uma função que leva um estado e me dá de volta todas as ações possíveis que eu posso tomar. Ele me diz todos os movimentos que são possíveis. O jogador max vai especificamente escolher uma ação, A, no conjunto de ações que me dá o maior valor de min valor de resultado de S e A. Então, o que isso significa? Bem, significa que eu quero fazer a opção que me dá a maior pontuação de todas as ações, A. Mas qual será a pontuação? Para calcular isso, preciso saber o que meu oponente, o jogador min, vai fazer se tentar minimizar o valor do estado que resulta. Então, dizemos, qual estado resulta depois que eu tomo esta ação e o que acontece quando o jogador min tenta minimizar o valor desse estado? Eu considero isso para todas as minhas opções possíveis. E depois de ter considerado isso para todas as minhas opções possíveis, eu escolho a ação, A, que tem o maior valor. Da mesma forma, o jogador min vai fazer a mesma coisa, mas ao contrário. Eles também vão considerar, quais são todas as ações possíveis que eles podem tomar se for a vez deles? E eles vão escolher a ação, A, que tem o menor valor possível de todas as opções. E a maneira como eles sabem qual é o menor valor possível de todas as opções é considerando o que o jogador max vai fazer, dizendo, qual é o resultado de aplicar esta ação ao estado atual e, em seguida, o que o jogador max tentaria fazer? Qual valor o jogador max calcularia para aquele estado específico? Então, todos fazem suas decisões com base em tentar estimar o que a outra pessoa faria. E agora precisamos nos voltar para essas duas funções, maxValue e minValue. Como você realmente calcula o valor de um estado se estiver tentando maximizar seu valor e como você calcula o valor de um estado se estiver tentando minimizar o valor? Se você puder fazer isso, então temos uma implementação inteira deste algoritmo Minimax. Então, vamos tentar. Vamos tentar implementar esta função maxValue que recebe um estado e retorna como saída o valor desse estado se eu estiver tentando maximizar o valor do estado. Bem, a primeira coisa que posso verificar é ver se o jogo acabou, porque se o jogo acabou - em outras palavras, se o estado é um estado terminal - então isso é fácil. Eu já tenho essa função de utilidade que me diz qual é o valor do tabuleiro. Se o jogo acabou, eu só verifico, X ganhou? O ganhou? É um empate? E a função de utilidade sabe qual é o valor do estado. O que é mais complicado é se o jogo não acabou, porque então preciso fazer essa razão recursiva de pensar, o que meu oponente vai fazer na próxima jogada? Então eu quero calcular o valor deste estado e quero que o valor do estado seja o mais alto possível. E vou manter o controle desse valor em uma variável chamada v.

E se eu quiser que o valor seja o mais alto possível, preciso dar a v um valor inicial. E inicialmente, eu só vou adiante e defini-lo para ser o mais baixo possível, porque eu ainda não sei quais opções estão disponíveis para mim.

Inicialmente, vou definir v como infinito negativo, o que parece um pouco estranho, mas a ideia aqui é que eu quero que o valor inicial seja o mais baixo possível, pois, ao considerar minhas ações, sempre vou tentar fazer melhor do que v. E se eu definir v como infinito negativo, sei que sempre posso fazer melhor do que isso. Então, agora vou considerar minhas ações. E isso vai ser algum tipo de loop, onde para cada ação em ações de estado -

lembre-se, ações é uma função que leva meu estado e me dá todas as ações possíveis que eu posso usar nesse estado. Então, para cada uma dessas ações, quero compará-la a v e dizer: tudo bem, v vai ser igual ao máximo de v e a esta expressão. Então, o que é esta expressão? Bem, primeiro é obter o resultado de tomar a ação e o estado, e depois obter o valor mínimo desse estado. Em outras palavras, digamos que eu quero descobrir, a partir desse estado, qual é o melhor que o jogador mínimo pode fazer, pois eles vão tentar minimizar a pontuação. Então, qualquer que seja a pontuação resultante do valor mínimo desse estado, compare-a com meu melhor valor atual e escolha o máximo desses dois, pois estou tentando maximizar o valor. Em resumo, o que essas três linhas de código estão fazendo é passar por todas as minhas ações possíveis e perguntar: como eu maximizo a pontuação, dado o que meu oponente vai tentar fazer? Depois de todo esse loop, posso simplesmente retornar v, e esse é agora o valor desse estado em particular. E para o jogador mínimo, é exatamente o oposto disso, a mesma lógica, só para trás. Para calcular o valor mínimo de um estado, primeiro verificamos se é um estado terminal. Se for, retornamos sua utilidade. Caso contrário, vamos tentar minimizar o valor do estado, dado todas as minhas ações possíveis. Então, preciso de um valor inicial para v, o valor do estado. E inicialmente, vou defini-lo como infinito, pois sei que sempre posso obter algo menor que o infinito. Então, ao começar com v igual ao infinito, garanto que a primeira ação que encontrar será menor que esse valor de v.

Então, faço a mesma coisa - percorro todas as minhas ações possíveis e, para cada um dos resultados que poderíamos obter quando o jogador máximo toma sua decisão, vamos pegar o mínimo disso e o valor atual de v. Então, depois de tudo isso, obtenho o menor valor possível de v, que então devolvo ao usuário. Então, na prática, esse é o pseudocódigo do Minimax. É assim que tomamos um jogo e descobrimos qual é o melhor movimento a ser feito, recursivamente usando essas funções maxValue e minValue, onde maxValue chama minValue, minValue chama maxValue, de volta e meia, até chegarmos a um estado terminal, a partir do qual nosso algoritmo pode simplesmente retornar a utilidade desse estado em particular. O que você pode imaginar é que isso começará a ser um processo longo, especialmente à medida que os jogos começam a ficar mais complexos, à medida que começamos a adicionar mais movimentos e mais opções possíveis e jogos que podem durar um pouco mais. Então, a próxima pergunta a fazer é: quais são as possíveis otimizações que podemos fazer aqui? Como podemos fazer melhor para usar menos espaço ou levar menos tempo para conseguir resolver esse tipo de problema? E vamos dar uma olhada em algumas possíveis otimizações. Mas, por enquanto, vamos dar uma olhada neste exemplo. Novamente, estamos voltando para essas setas para cima e para baixo. Vamos imaginar que agora sou o jogador máximo, essa seta verde. Estou tentando fazer a pontuação o mais alto possível. E este é um jogo fácil, onde só há dois movimentos. Faço um movimento, uma dessas três opções, e então meu oponente faz um movimento, uma dessas três opções, com base no que eu faço. E como resultado, obtemos algum valor. Vamos olhar a ordem em que faço essas cálculos e ver se há alguma otimização que eu possa fazer nesse processo de cálculo. Vou ter que olhar para esses estados um de cada vez. Então, digamos que eu comece aqui à esquerda e digo: tudo bem, agora vou considerar, o que o jogador mínimo, meu oponente, vai tentar fazer aqui?

Bem, o jogador min vai olhar para todas as três ações possíveis e olhar para seus valores, porque esses são estados terminais. Eles são o fim do jogo. E então eles vão ver, tudo bem, este nó tem um valor de 4, valor de 8, valor de 5. E o jogador min vai dizer, bem, tudo bem. Entre essas três opções, 4, 8 e 5, eu pego a menor. Então, este estado agora tem um valor de 4. Então eu, como jogador max, digo, tudo bem, se eu tomar essa ação, terá um valor de 4. É o melhor que eu posso fazer, porque o jogador min vai tentar minimizar minha pontuação. Então,

agora, o que acontece se eu tomar essa opção? Vamos explorar isso a seguir. E agora eu exploro o que o jogador min faria se eu escolhesse essa ação. E o jogador min vai dizer, tudo bem, quais são as três opções? O jogador min tem opções entre 9, 3 e 7, e então 3 é o menor entre 9, 3 e 7. Então vamos adiante e dizer que este estado tem um valor de 3. Então agora eu, como jogador max - eu agora explorei duas das minhas três opções. Sei que uma das minhas opções me garantirá uma pontuação de 4, pelo menos, e uma das minhas opções me garantirá uma pontuação de 3. E agora considero minha terceira opção e digo, tudo bem, o que acontece aqui? O mesmo raciocínio lógico - o jogador min vai olhar esses três estados, 2, 4 e 6, dizendo que a opção mínima possível é 2, então o min jogador quer o dois. Agora eu, como jogador max, calculei todas as informações olhando duas camadas profundas, olhando para todos esses nós. E agora eu posso dizer, entre os 4, os 3 e os 2, você sabe o que? Eu prefiro pegar o 4, porque se eu escolher essa opção, se meu oponente jogar otimamente, eles vão tentar me levar ao 4, mas é o melhor que eu posso fazer. Eu não posso garantir uma pontuação maior, porque se eu escolher qualquer uma dessas duas opções, eu posso obter um 3, ou eu posso obter um 2. E é verdade que aqui embaixo é um 9, e essa é a maior pontuação de qualquer uma das pontuações. Então eu posso ser tentado a dizer, você sabe o que? Talvez eu devesse pegar essa opção, porque eu posso obter o 9. Mas se o jogador min estiver jogando inteligentemente, se estiverem fazendo os melhores movimentos em cada opção possível quando chegarem a fazer uma escolha, eu ficarei com um 3, enquanto eu poderia melhorar, jogando otimamente, ter garantido que eu teria o 4. Então isso não afeta a lógica que eu usaria como um jogador Minimax tentando maximizar minha pontuação a partir desse nó. Mas acontece que isso levou bastante cálculo para eu descobrir isso. Eu tive que raciocinar por todos esses nós para chegar a essa conclusão. E isso é para um jogo bem simples, onde eu tenho três escolhas, meu oponente tem três escolhas e então o jogo acaba. Então o que eu gostaria de fazer é encontrar alguma maneira de otimizar isso. Talvez eu não precise fazer todos esses cálculos para ainda chegar à conclusão de que, você sabe o que? Esta ação à esquerda - é o melhor que eu poderia fazer. Vamos lá e tentar de novo e tentar ser um pouco mais inteligente sobre como eu vou sobre isso. Então, primeiro, começo da mesma forma. Eu não sei o que fazer inicialmente, então eu só tenho que considerar uma das opções e considerar o que o jogador min poderia fazer. Min tem três opções, 4, 8 e 5. E entre essas três opções, min diz, 4 é o melhor que eles podem fazer, porque eles querem tentar minimizar a pontuação. Agora, eu, como jogador max, considerarei minha segunda opção, fazendo esse movimento aqui e considerando o que meu oponente faria em resposta. O que o jogador min fará? Bem, o jogador min vai, a partir desse estado, olhar para suas opções. E eu diria, tudo bem. 9 é uma opção, 3 é uma opção. E se eu estiver fazendo a matemática a partir deste estado inicial, fazendo todos esses cálculos, quando eu vejo um 3, isso deve ser imediatamente um sinal vermelho para mim, porque quando eu vejo um 3 aqui neste estado, eu sei que o valor desse estado será no máximo 3. Vai ser 3 ou algo menos que 3, mesmo que eu ainda não tenha olhado para essa última ação ou mesmo para ações mais distantes se houvesse mais ações que pudessem ser tomadas aqui. Como eu sei disso?

Bem, eu sei que o jogador min tentará minimizar minha pontuação. E se eles verem um 3, a única maneira de isso ser algo diferente de um 3 é se essa coisa restante que ainda não olhei for menor que 3, o que significa que não há maneira para esse valor ser algo maior que 3, porque o jogador min já pode garantir um 3 e eles estão tentando minimizar minha pontuação. Então, o que isso me diz? Bem, isso me diz que se eu escolher essa ação, minha pontuação será 3, ou talvez até menos que 3, se eu tiver azar. Mas eu já sei que essa ação me garantirá um 4. E então, dado que eu sei que essa ação me garante uma pontuação de 4, e essa ação significa que eu não posso

fazer melhor que 3, se eu estiver tentando maximizar minhas opções, não há necessidade de eu considerar esse triângulo aqui. Não há valor, nenhum número que possa ir aqui, que mudaria minha mente entre essas duas opções. Eu sempre vou optar por esse caminho que me dá um 4, em vez desse caminho, onde o melhor que eu posso fazer é um 3, se meu oponente jogar de forma ótima. E isso vai ser verdade para todos os estados futuros que eu olhar também. Mas se eu olhar para cá, para o que o jogador min pode fazer aqui, se eu ver que esse estado é um 2, eu sei que esse estado é no máximo um 2, porque a única maneira que esse valor poderia ser algo diferente de 2 é se um desses estados restantes for menor que um 2, e então o jogador min optaria por isso. Então, mesmo sem olhar para esses estados restantes, eu, como jogador maximizador, posso saber que escolher esse caminho à esquerda vai ser melhor do que escolher qualquer um desses dois caminhos à direita, porque esse não pode ser melhor que 3, esse não pode ser melhor que 2, e então 4 nesse caso é o melhor que eu posso fazer. E eu posso dizer agora que esse estado tem um valor de 4. Então, para fazer esse tipo de cálculo, eu estava fazendo um pouco mais de controle, mantendo as coisas, mantendo o controle o tempo todo de, qual é o melhor que eu posso fazer, qual é o pior que eu posso fazer, e para cada um desses estados, dizendo, bem, se eu já sei que eu posso conseguir um 4, então, se o melhor que eu posso fazer nesse estado é um 3, não há razão para eu considerar isso. Eu posso efetivamente podar essa folha e qualquer coisa abaixo dela da árvore. E é por isso que essa abordagem, essa otimização para Minimax, é chamada de poda alfa-beta. Alfa e beta representam esses dois valores que você terá que manter em mente, o melhor que você pode fazer até agora e o pior que você pode fazer até agora. E a poda é a ideia de, se eu tiver uma grande árvore de pesquisa, longa e profunda, eu posso pesquisá-la de forma mais eficiente se eu não precisar pesquisar tudo, se eu puder remover alguns dos nós para tentar otimizar a forma como eu olho por todo esse espaço de pesquisa. Então, a poda alfa-beta pode definitivamente nos poupar muito tempo enquanto vamos sobre o processo de pesquisa, tornando nossas pesquisas mais eficientes. Mas mesmo assim, ainda não é ótimo conforme os jogos ficam mais complexos. O jogo da velha, felizmente, é um jogo relativamente simples, e podemos razoavelmente perguntar uma coisa como, quantos jogos de jogo da velha existem no total? Você pode pensar. Você pode tentar estimar, quantos movimentos existem em qualquer ponto? Quão longos podem ser os jogos? Na verdade, há cerca de 255.000 jogos de jogo da velha que podem ser jogados. Mas compare isso com um jogo mais complexo, algo como um jogo de xadrez, por exemplo - muito mais peças, muito mais movimentos, jogos que duram muito mais tempo. Quantos jogos de xadrez totais podem haver? Na verdade, depois de apenas quatro movimentos cada, quatro movimentos do jogador branco, quatro movimentos do jogador preto, existem 288 bilhões de jogos de xadrez que podem resultar dessa situação, depois de apenas quatro movimentos cada. E indo ainda mais longe. Se você olhar para jogos de xadrez inteiros e quantos jogos de xadrez possíveis podem haver como resultado disso, há mais de 10 para os 29.000 jogos de xadrez possíveis, muito mais jogos de xadrez do que jamais poderiam ser considerados.

Esse é um problema bem grande para o algoritmo Minimax, pois o Minimax começa com um estado inicial, considera todas as ações possíveis e todas as ações possíveis depois disso, até chegarmos ao fim do jogo. E isso será um problema se o computador precisar olhar por esses muitos estados, o que é muito mais do que qualquer computador poderia fazer em qualquer quantidade razoável de tempo. Então, o que fazemos para resolver esse problema? Em vez de olhar por todos esses estados, o que é totalmente intractável para um computador, precisamos de alguma abordagem melhor. E acontece que essa abordagem melhor geralmente assume a forma de algo chamado Minimax de profundidade limitada. Onde normalmente o Minimax é de profundidade ilimitada - apenas continuamos, camada após camada, movimento após movimento, até chegarmos ao fim do jogo - o

Minimax de profundidade limitada vai dizer: você sabe o que? Após certo número de movimentos - talvez eu olhe 10 movimentos à frente, talvez eu olhe 12 movimentos à frente, mas após esse ponto, vou parar e não considerar movimentos adicionais que possam vir depois disso, apenas porque seria computacionalmente intractável considerar todas essas opções possíveis. Mas o que fazemos depois de chegar a 10 ou 12 movimentos de profundidade e chegarmos a uma situação em que o jogo não acabou? O Minimax ainda precisa de uma maneira de atribuir uma pontuação a esse tabuleiro de jogo ou estado do jogo para descobrir qual é o seu valor atual, o que é fácil de fazer se o jogo acabou, mas não tão fácil de fazer se o jogo ainda não acabou. Então, para isso, precisamos adicionar um recurso adicional ao Minimax de profundidade limitada chamado função de avaliação, que é apenas alguma função que vai estimar a utilidade esperada de um jogo a partir de um estado dado. Então, em um jogo como xadrez, se você imaginar que um valor de jogo de 1 significa que o branco ganha, o valor negativo 1 significa que o preto ganha, 0 significa que é um empate, então você pode imaginar que uma pontuação de 0,8 significa que o branco tem muitas chances de ganhar, embora certamente não seja garantido. E você teria uma função de avaliação que estima quão bom é o estado do jogo. E dependendo de quão bom for essa função de avaliação, é isso que vai limitar o quão bom é o AI. Quanto melhor o AI for em estimar quão bom ou ruim é qualquer tabuleiro de jogo em particular, melhor o AI vai conseguir jogar aquele jogo. Se a função de avaliação for pior e não tão boa para estimar qual é a utilidade esperada, então vai ser muito mais difícil. E você pode imaginar tentar criar essas funções de avaliação. No xadrez, por exemplo, você pode escrever uma função de avaliação com base em quantas peças você tem, em comparação com quantas peças o seu oponente tem, porque cada uma tem um valor na sua função de avaliação. Provavelmente precisa ser um pouco mais complicado do que isso para considerar outras possíveis situações que possam surgir também. E há muitas outras variantes do Minimax que adicionam recursos adicionais para ajudá-lo a desempenhar melhor em situações maiores e mais computacionalmente intractáveis, onde não poderíamos possivelmente explorar todos os movimentos possíveis, então precisamos descobrir como usar funções de avaliação e outras técnicas para conseguir jogar esses jogos, no final, melhor. Mas agora foi uma olhada nesse tipo de busca adversária, esses problemas de busca onde temos situações onde estou tentando jogar contra algum tipo de oponente. E esses problemas de busca aparecem por toda parte na inteligência artificial. Falamos muito hoje sobre problemas de busca mais clássicos, como tentar encontrar direções de um local para outro. Mas a qualquer momento em que um AI é confrontado com a tarefa de tomar uma decisão, como o que devo fazer agora para fazer algo que seja racional ou algo que seja inteligente ou tentar jogar um jogo, como descobrir qual movimento fazer, esses tipos de algoritmos realmente podem ser úteis. Acaba por ser que para o jogo da velha, a solução é bastante simples, porque é um jogo pequeno.

XKCD criou famosamente um webcomic onde ele lhe dirá exatamente qual a jogada a fazer como a jogada ótima, não importando o que o seu oponente fizer. Esse tipo de coisa não é tão possível para um jogo muito maior como o dama ou xadrez, por exemplo, onde o xadrez é totalmente computacionalmente intratável para a maioria dos computadores serem capazes de explorar todos os estados possíveis. Então, realmente precisamos que nossa IA seja muito mais inteligente sobre como ela vai lidar com esses problemas e como ela vai lidar com esse ambiente em que ela se encontra e, finalmente, procurar por uma dessas soluções. Então, isso foi uma olhada na pesquisa e inteligência artificial. Da próxima vez, vamos dar uma olhada no conhecimento, pensando em como é que nossas IA são capazes de conhecer informações, raciocinar sobre essas informações e tirar conclusões, tudo em nossa análise da IA e dos princípios por trás dela. Até a próxima.