MAC0425/5739 - Inteligência Artificial

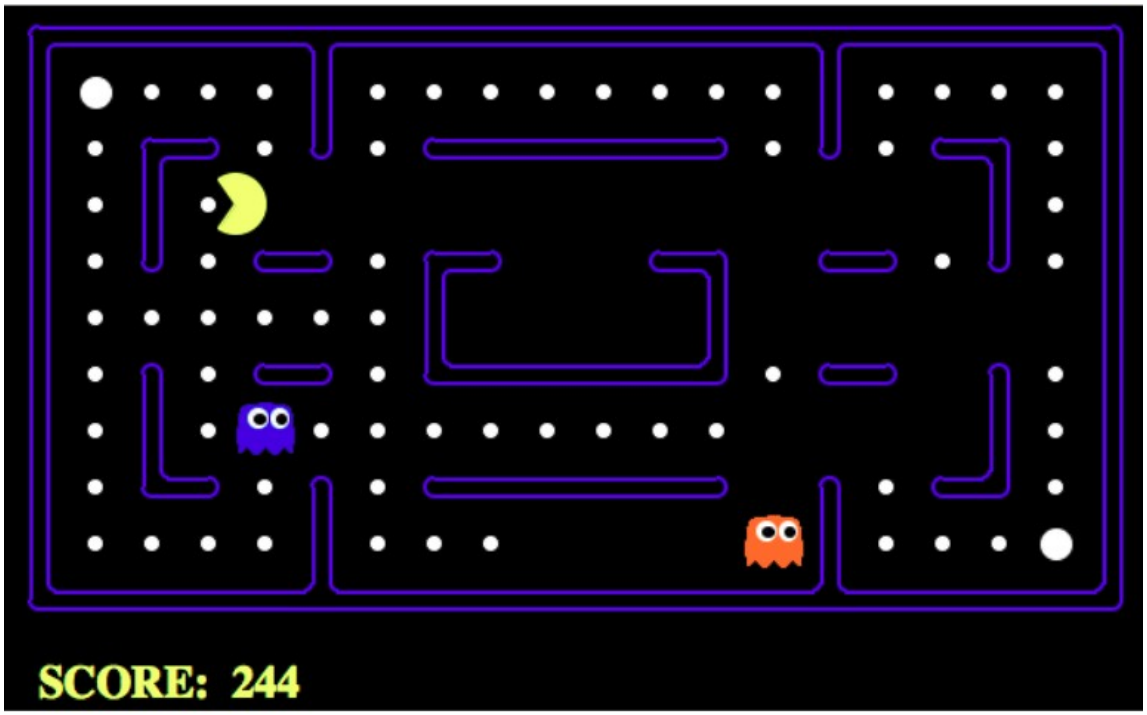
Exercício-Programa 1 - Busca

Prazo limite de entrega: 13/04/18 às 23h55

1) Introdução

Neste exercício-programa estudaremos a abordagem de resolução de problemas através de busca no espaço de estados, implementando um jogador inteligente de Pac-Man. Utilizaremos parte do material/código livremente disponível do curso UC Berkeley CS188.

Pac-Man é um jogo eletrônico em que um jogador, representado por uma boca que se abre e fecha, deve mover-se em um labirinto repleto de comida e fantasmas que o perseguem. O objetivo é comer o máximo possível de comida sem ser alcançado pelos fantasmas, em ritmo progressivo de dificuldade.



Existem muitas variações do jogo Pac-Man. Para este exercício-programa, consideraremos um cenário no qual utilizaremos algoritmos de busca para guiar nosso Pac-Man no labirinto a fim de atingir determinadas posições e coletar comida eficientemente. Técnicas de busca adversarial, porém, serão abordadas posteriormente.

Os objetivos deste exercício-programa são:

- compreender a abordagem de resolução de problemas baseada em busca no espaço de estados;
- estudar uma formulação de problema de busca para criar um jogador autônomo de Pac-Man;
- implementar algoritmos de busca informada e não-informada e comparar seus desempenhos.

Instalação:

Para a realização deste EP será necessário ter instalado em sua máquina a versão 3.6 ou posterior do Python. Feito o download do arquivo ep1.zip disponível no e-Disciplinas, descompacte o arquivo e rode no terminal, após abrir o respectivo diretório *search*, o seguinte comando para testar a instalação jogando um jogo do Pacman:

```
$ python pacman.py
```

Este projeto inclui um auto-avaliador (autograder) para que você possa avaliar suas próprias respostas. Ele pode ser executado com o comando:

```
$ python autograder.py
```

Para mais detalhes para instalação do Python e recursos necessários e do uso do autograder, consultar https://inst.eecs.berkeley.edu/~cs188/sp23/projects/proj0/. Para mais detalhes do projeto, consultar https://inst.eecs.berkeley.edu/~cs188/sp23/projects/proj1/#q4-3-pts-a-search.

2) Códigos

O código para este projeto consiste em vários arquivos Python, alguns dos quais você precisará ler e entender para concluir a tarefa, e alguns dos quais você pode ignorar:

Arquivos que devem ser editados:	
`busca/search.py`	Onde ficam os algoritmos de busca.
`searchAgents.py`	Onde ficam os agentes baseados em busca. (Será editado apenas na sessão bônus).

Arquivos que devem ser lidos:	
`pacman.py`	O arquivo principal que roda jogos de Pacman. Esse arquivo também descreve o tipo *GameState*, que será amplamente usado nesse trabalho.
`game.py`	A lógica por trás do mundo do Pacman. Esse arquivo descreve vários tipos auxiliares como *AgentState*, *Agent*, *Direction* e *Grid*.
`util.py`	Estruturas de dados úteis para implementar algoritmos de busca.

Observação: utilize as estruturas de dados *Stack*, *Queue*, *PriorityQueue* e *PriorityQueueWithFunction* disponíveis no arquivo *util.py* para implementação da fronteira de busca. Listas, tuplas, tuplas nomeadas, conjuntos e dicionários do Python podem ser utilizados sem problemas caso necessário.

Arquivos auxiliares que podem ser ignorados:	
graphicsDisplay.py	Visualização gráfica do Pacman
graphicsUtils.py	Funções auxiliares para visualização gráfica do Pacman.
textDisplay.py	Visualização gráfica em ASCII para o Pacman.
ghostAgents.py	Agentes para controlar fantasmas.
keyboardAgents.py	Interfaces de controle do Pacman a partir do teclado.
layout.py	Código para ler arquivos de layout e guardar seu conteúdo.
autograder.py	Autoavaliador de projeto
testParser.py	Analisa arquivos de teste e solução do autograder
testClasses.py	Classes gerais de teste de autoavaliação
test_cases/	Diretório contendo os casos de teste para cada pergunta
searchTestClasses.py	Classes de teste específicas do Projeto 1

O que deve ser entregue: Cada dupla deve entregar os arquivos `search.py` e `searchAgents.py`, que serão modificados no trabalho, conforme especificado anteriormente; bem como um relatório respondendo as perguntas listadas abaixo.

Importante: Por favor, não altere os nomes de nenhuma das funções ou classes fornecidas dentro do código, ou você causará problemas no autograder. – No entanto, a correção da sua implementação, e não apenas o julgamento do autograder, é que determinará sua nota final. Se necessário, seu código será revisado e as tarefas avaliadas individualmente para garantir que você receba o devido crédito. Seu trabalho também passará por verificação de plágio. Se você copiar o código de outra dupla e enviá-lo com pequenas alterações, será detectado, e haverá consequências.

3) Parte prática

Neste exercício-programa você resolverá diversos problemas de busca. Independente da busca, a interface que implementa a formulação do problema é definida pela classe abstrata (disponível no arquivo `search.py` na pasta `search/`) e seu conjunto de métodos.

Você deverá implementar algumas funções no arquivo *`search.py`* e, posteriormente, novos agentes no arquivo `searchAgents.py`. (Não esqueça de remover o código *`util.raiseNotDefined()`* ao final de cada função na hora da implementação.)

Vamos começar com alguns exemplos já implementados no código que você baixou. O agente mais simples em *searchAgents.py* é o agente *`GoWestAgent`*, que sempre vai para oeste (um agente reflexivo trivial). Este agente pode ganhar às vezes.:

```
$ python pacman.py --layout testMaze --pacman GoWestAgent
```

Mas, as coisas se tornam mais difíceis quando virar é necessário, como ocorre neste labirinto:

```
$ python pacman.py --layout tinyMaze --pacman GoWestAgent
```

(Quando o Pacman ficar preso, você pode sair do jogo digitando Ctrl+c em seu terminal, ou fechando a janela do jogo.)

No arquivo *searchAgents.py*, você irá encontrar o programa de um agente de busca (*SearchAgent*), que planeja um caminho no mundo do Pacman e executa o caminho passo-a-passo. Mas, a princípio, os algoritmos de busca para planejar o caminho não estão implementados -- este será o seu principal trabalho. (Para entender o que está descrito a seguir, pode ser necessário olhar o glossário de objetos ao final deste enunciado.)

Primeiro, verifique que o agente de busca *SearchAgent* está funcionando corretamente:

```
$ python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch
```

O comando acima faz o agente *SearchAgent* usar o método *tinyMazeSearch*, que já está implementado em *search.py* com um plano de ação feito especialmente para o agente vencer no `tinyMaze`. (Nesse exemplo, porém, não é realizada uma busca, como irá ocorrer nos métodos que você irá implementar. É apenas uma demonstração da navegação do agente, e da saída dos métodos de busca)

Note que *pacman.py* tem opções que podem ser usadas na chamada em formato longo (por exemplo, `--layout`) ou em formato curto (por exemplo, `-l`). A lista de todas as opções e seus valores default pode ser vista executando:

```
$ python pacman.py -h
```

Agora, chegou a hora de implementar os seus algoritmos de busca para o Pacman! Os pseudocódigos dos algoritmos de busca estão no livro-texto. Lembre-se que um nó da busca deve conter não só o estado mas, também, toda a informação necessária para reconstruir o caminho (sequência de ações) até aquele estado.

Dicas:

1. Todas as funções de busca devem retornar uma lista de *ações* que irão levar o agente do início até o objetivo, assim como se observa no exemplo *tinyMazeSearch*. Essas ações devem ser legais (direções válidas, sem passar pelas paredes).

   Veja com atenção os comentários e recomendações nos métodos presentes no arquivo *search.py*, bem como o retorno do exemplo mostrado acima. Esse entendimento deverá esclarecer como utilizar a estrutura dos códigos deste EP. E não se esqueça das estruturas de dados disponíveis em *util.py*.

2. Os algoritmos de busca são muito parecidos. Os algoritmos de busca em profundidade, em extensão, de custo uniforme e A* diferem somente na ordem em que os nós são retirados da borda. Então o ideal é tentar implementar a busca em profundidade corretamente e depois será mais fácil implementar as outras. Uma possível implementação é criar um algoritmo de busca genérico que possa ser configurado com uma opção de estratégia para retirar nós da borda. (Porém, implementar dessa forma ou não, fica a seu critério).

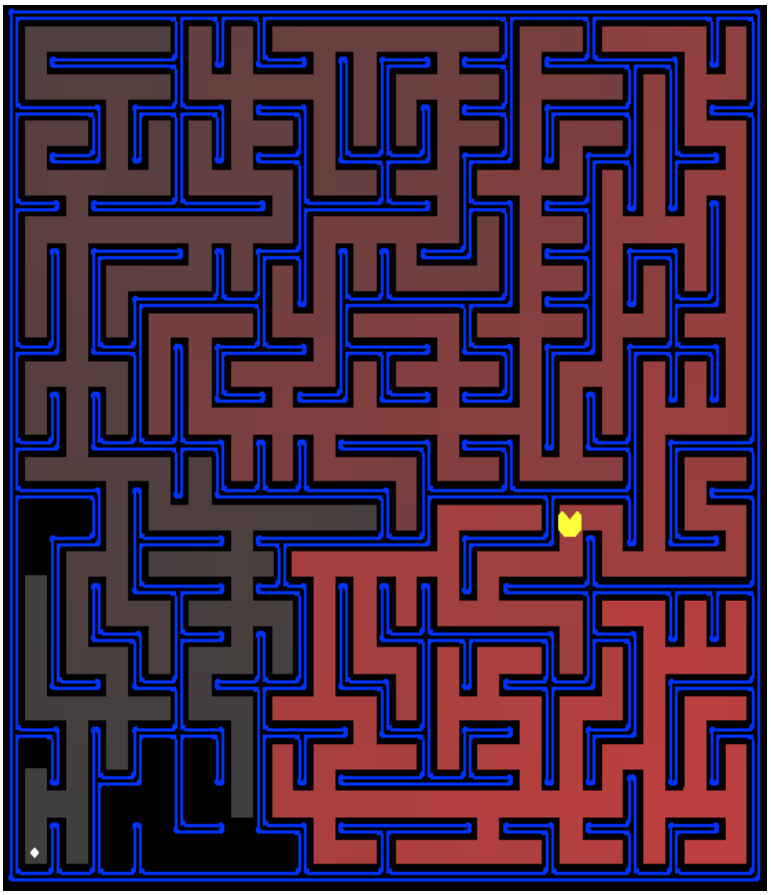
➔ **Passo 1 (2 pontos):** implemente o algoritmo de busca em profundidade (DFS) na função *depthFirstSearch* do arquivo *search.py*. Para que a busca seja completa, implemente a versão de DFS de busca em grafo – que não expande estados repetidos (seção 3.5 do livro). Teste seu código executando:

```
$ python pacman.py -l tinyMaze -p SearchAgent

$ python pacman.py -l mediumMaze -p SearchAgent

$ python pacman.py -l bigMaze -z .5 -p SearchAgent
```

A saída do Pacman irá mostrar os estados explorados e a ordem em que eles foram explorados (vermelho mais forte significa que o estado foi explorado antes).



➔ **(Pergunta 1):** a ordem de exploração foi de acordo com o esperado? O Pacman realmente passa por todos os estados explorados no seu caminho para o objetivo?

Dica: Se você usar a implementação *Stack* (*em util.py*) como estrutura de dados, a solução encontrada pelo algoritmo DFS para o *mediumMaze* deve ter comprimento 130 (se os sucessores forem colocados na pilha na ordem dada por *getSuccessors* (função de *search.py*); pode ter comprimento 246 se forem colocados na ordem reversa).

Você pode usar o autograder para testar essa implementação rodando:

```
$ python autograder.py -q q1
```

➔ **(Pergunta 2):** essa é uma solução ótima? Senão, o que a busca em profundidade está fazendo de errado?

➔ **Passo 2 (2 pontos):** implemente o algoritmo de busca em extensão (BFS) na função `breadthFirstSearch` do arquivo *search.py*. De novo, implemente a versão com busca em grafo, que não expande estados que já foram visitados. Pode testar seu código executando:

```
$ python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
```

```
$ python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
```

➔ **(Pergunta 3):** a busca BFS encontra a solução ótima? Por quê? Se não, verifique a sua implementação.

Dica: Se achar que o Pac-Man está se movendo muito devagar, tente a opção `-frameTime 0`.

Para teste específico deste *Passo* com o autograder, rode (e assim poderá ser feito para cada implementação, com o `x` em `-q qx` seguindo o número do respectivo *Passo* deste enunciado):

```
$ python autograder.py -q q2
```

➔ **Passo 3 (2 pontos):** implemente o algoritmo de busca em grafo de custo uniforme na função `uniformCostSearch` (UCS) em `search.py`.

Enquanto BFS encontra um caminho de menor número de ações para o objetivo, podemos querer encontrar caminhos que sejam "melhores" em outros sentidos.

Considere `mediumDottedMaze` e `mediumScaryMaze`. Ao mudar a função de custo, podemos incentivar Pacman a encontrar caminhos diferentes. Por exemplo, podemos deixar mais caro passar em áreas perigosas com fantasmas, oucobrar menos em áreas com bastante comida, e um agente racional do Pacman deveria ajustar seu comportamento em resposta.

Após a correta implementação, você deve observar um comportamento bem-sucedido em todos os três dos seguintes layouts, onde os agentes abaixo são todos agentes UCS que diferem apenas na função de custo que usam (Os agentes e funções de custo já estão escritos para você. Basta implementar a busca.):

```
$ python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
```

```
$ python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
```

```
$ python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
```

` *Nota*: Você deve obter custos de caminho muito baixos e muito altos para o `StayEastSearchAgent` e `StayWestSearchAgent`, respectivamente, devido às suas funções de custo exponenciais – aconsulte `searchAgents.py` para obter detalhes).

➔ **(Pergunta 4):** O caminho obtido com o agente usado nas questões anteriores (`SearchAgent`) é igual ou diferente ao obtido com as outras buscas para o mesmo labirinto ? Por quê?

➔ **Passo 4 (2 pontos):** implemente a busca A* (com checagem de estados repetidos) na função *`aStarSearch`* do arquivo *`search.py`*.

A busca A* recebe uma função heurística como parâmetro. Heurísticas têm dois parâmetros: o parâmetro principal é um estado do problema de busca, e o outro é próprio problema (que chega como parâmetro de referência para os métodos de busca). Ou seja, o parâmetro deve ser usado na forma: *`heuristic(<um_estado>, problem)`*

A heurística implementada na função *`nullHeuristic`* do arquivo *`search.py`* é um exemplo trivial. Teste sua implementação de A* no problema original de encontrar um caminho para uma posição fixa do labrinto usando a heurística de distância Manhattan (já implementada na função *`manhattanHeuristic`* do arquivo *`searchAgents.py`*).

```
$ python pacman.py -l bigMaze -z .5 -p SearchAgent -a
fn=astar,heuristic=manhattanHeuristic
```

➔ **(Pergunta 5):** você deve ter percebido que o algoritmo A* encontra uma solução mais rapidamente que outras buscas. Por quê? Qual a razão para se implementar uma heurística consistente para sua implementação da busca A*?

➔ **(Pergunta 6):** o que acontece em *`openMaze`* para as várias estratégias de busca?

Sessão Bônus!

➔ **Passo 5 (1 ponto extra):** implemente o problema de busca *`CornersProblem`* em *`searchAgents.py`* para encontrar o caminho mais curto através do labirinto que passa por todos os seus quatro cantos (ou quinas) – independentemente de ter comida lá ou não.

Para isso, você irá formular um novo problema. Você precisará escolher uma representação de estados que codifique todas as informações necessárias para detectar se todos os quatro cantos foram alcançados. Agora, seu agente de busca deve resolver:

```
$ python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,
prob=CornersProblem
```

```
$ python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,
prob=CornersProblem
```

Uma instância da classe *`CornersProblem`* representa um problema de busca completo, não um estado específico. Estados específicos serão retornados pelas suas funções, que retornarão uma estrutura de dados de sua escolha (por exemplo, tupla, conjunto etc.) que representa um estado.

Para receber crédito total, você precisa construir uma representação de estado abstrata que não codifique informações irrelevantes (como a posição dos fantasmas, onde tem mais comida etc.). Em particular, não use um `GameState` como estado de busca (ou seu código ficará muito lento e, também, estará errado).

Dicas:

- Certifique-se de completar o Passo 2 antes de tentar este, pois o passo 5 baseia-se em sua implementação do Passo 2.
- As únicas partes do estado do jogo que você precisa referenciar em sua implementação são a posição inicial do PacMan e a localização dos quatro cantos.
- Ao codificar getSuccessors, certifique-se de adicionar filhos à sua lista de sucessores com um custo de 1.
- O caminho mais curto através de `tinyCorners` tem 28 passos.

➔ **(Pergunta 7):** O caminho mais curto sempre passa pela comida mais próxima primeiro? Por quê?

➔ **Passo 6 (1 ponto extra):** implemente uma heurística consistente[1] não trivial[2] para o `CornersProblem` em `cornersHeuristic`. Lembre-se, as heurísticas são apenas funções que recebem um estado de busca e retornam uma estimativa de custo para um objetivo mais próximo. Heurísticas mais eficientes retornarão valores mais próximos dos custos reais para o objetivo. Rode:

```
$ python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
```

Dicas:

➔ `AStarCornersAgent` é um atalho para: `-p SearchAgent -a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic`

- Certifique-se de completar o Passo 4 antes de trabalhar nessa questão, pois ela se baseia na implementação do A*.

Classificação: Dependendo de quantos nós sua heurística expande, você será classificado da seguinte forma:

Número de nós expandidos	Nota
mais de 2000	0
no máximo 2000	0.3
no máximo 1600	0.7
no máximo 1200	1

Certifique-se de que sua heurística retorne 0 em todos os estados de objetivo, seja consistente, não trivial e nunca retorne um valor negativo, ou a nota também será zerada!

➔ **(Pergunta 8):** Compare a execução no *`mediumCorners`* com a do Passo anterior nesse labirinto. A utilização da sua heurística se provou vantajosa?

[1] Para ser admissível, os valores da heurística devem ser limites inferiores para o custo real do caminho mais curto para o objetivo mais próximo (e não negativo). Porém, a admissibilidade não é suficiente para garantir a corretude na busca em grafos - você precisa da condição mais forte de consistência. Para ser consistente, deve também ser atendido que, se uma ação tiver custo c, então executar essa ação só pode causar uma queda na heurística de no máximo c.

No entanto, as heurísticas admissíveis geralmente também são consistentes, especialmente se forem derivadas de relaxamentos de problemas. Portanto, geralmente é mais fácil começar pensando em heurísticas admissíveis. Depois de chegar em uma heurística admissível que funcione bem, você pode verificar se ela é realmente consistente. A única maneira de garantir a consistência é com uma prova. No entanto, a inconsistência muitas vezes pode ser detectada verificando que, para cada nó expandido, o valor f de seus sucessores são maiores ou iguais. Além disso, se execuções do mesmo problema utilizando UCS e A* retornarem caminhos de comprimentos diferentes, sua heurística é inconsistente.

[2] As heurísticas triviais são aquelas que retornam zero em todos os lugares (o que equivale a utilizar UCS) e a heurística que calcula o verdadeiro custo de conclusão. A primeira não economiza tempo, enquanto a última ultrapassará o limite de tempo do autograder. Você deseja uma heurística que reduza o tempo total de execução – embora, para esta tarefa, o autograder verifique apenas a contagem de nós (além de impor um limite de tempo razoável).

4) Entrega

Você deve entregar um arquivo `ep1-SeuNomeVaiAqui.zip` contendo **APENAS** os arquivos:

- `search.py` e `searchAgents.py` com as implementações da parte prática;
- relatório em formato PDF com as questões apresentadas para discussão dos resultados (máximo de 3 páginas + 1 página para a parte bônus).

Não esqueça de identificar cada arquivo com seu nome e número USP! Para os códigos, coloque um cabeçalho em forma de comentário.

5) Avaliação

O critério de avaliação dependerá parcialmente dos resultados dos testes automatizados do `autograder.py`. Desta forma, você terá como avaliar por si só parte da nota que receberá para a parte prática:

```
$ python autograder.py
```

Em relação ao relatório, que vale 2 pontos, avaliaremos principalmente sua forma de interpretar comparativamente os desempenhos de cada busca. Não é necessário detalhar a estratégia

de cada busca ou qual estrutura de dados você utilizou para cada busca, mas deve ficar claro que você compreendeu os resultados obtidos conforme esperado, dadas as características de cada busca.

6) Glossário de Objetos

Este é um glossário dos objetos principais na base de código relacionada a problemas de busca:

- `SearchProblem (search.py)`
  Um *SearchProblem* é um objeto abstrato que representa o espaço de estados, função sucessora, custos, e estado objetivo de um problema.
- `PositionSearchProblem (searchAgents.py)`
  Um tipo específico de *SearchProblem* --- corresponde a procurar por uma única comida no labirinto.
- Função de Busca
  Uma função de busca é uma função que recebe como entrada uma instância de *SearchProblem*, roda algum algoritmo, e retorna a sequência de ações que levam ao objetivo. Exemplos de função de busca são *`depthFirstSearch`* e *`breadthFirstSearch`*, que deverão ser implementadas. A função de busca dada *`tinyMazeSearch`* é uma função muito ruim que só funciona para o labirinto *`tinyMaze`*.
- `SearchAgent`
  `SearchAgent` é uma classe que implementa um agente (um objeto que interage com o mundo) e faz seu planejamento de acordo com uma função de busca. *`SearchAgent`* primeiro usa uma função de busca para encontrar uma sequência de ações que levem ao estado objetivo, e depois executa as ações uma por vez.