UNIVERSIDADE
DO ESTADO DE MINAS GERAIS | UEMG

Trabalho de Inteligência Artificial.
Data de Entrega 15/09/2015

1. Defina Inteligência. O que é um comportamento inteligente de uma máquina?

2. Caracterize as seguintes categorias “Sistemas que pensam como os humanos”, “Sistemas que agem como os humanos”, “Sistemas que pensam racionalmente” e “Sistemas que agem racionalmente”.

3. Em que consiste um Agente Inteligente? Ilustre duas aplicações para esse tipo de agente.

4. Descreva as características dos seguintes agentes de acordo com a sua estrutura.

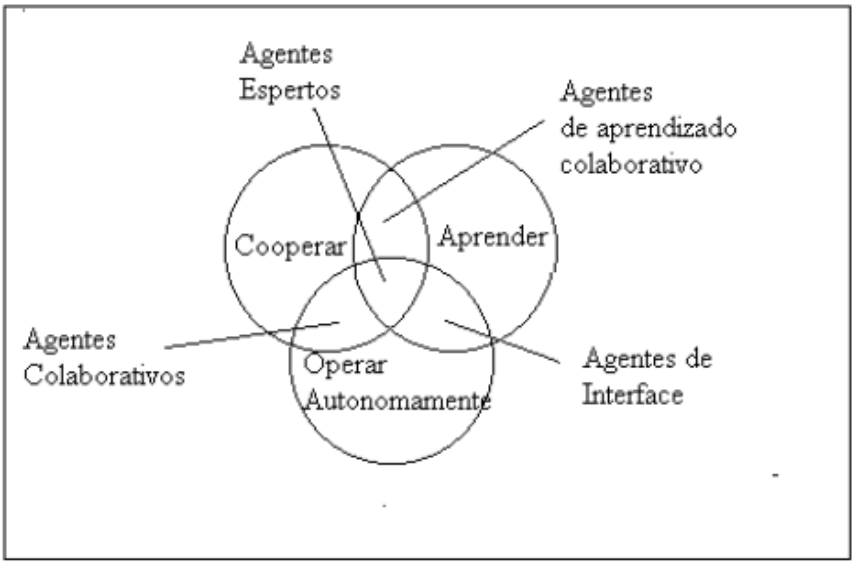
   - Agentes reativos simples
   - Agentes reativos baseados em modelo
   - Agentes baseados em objetivos
   - Agentes baseados na utilidade
   - Agentes com aprendizagem

5. Responda SIM ou NÃO para indicar o que caracteriza cada um dos ambientes apresentados a seguir (justifique as suas respostas).

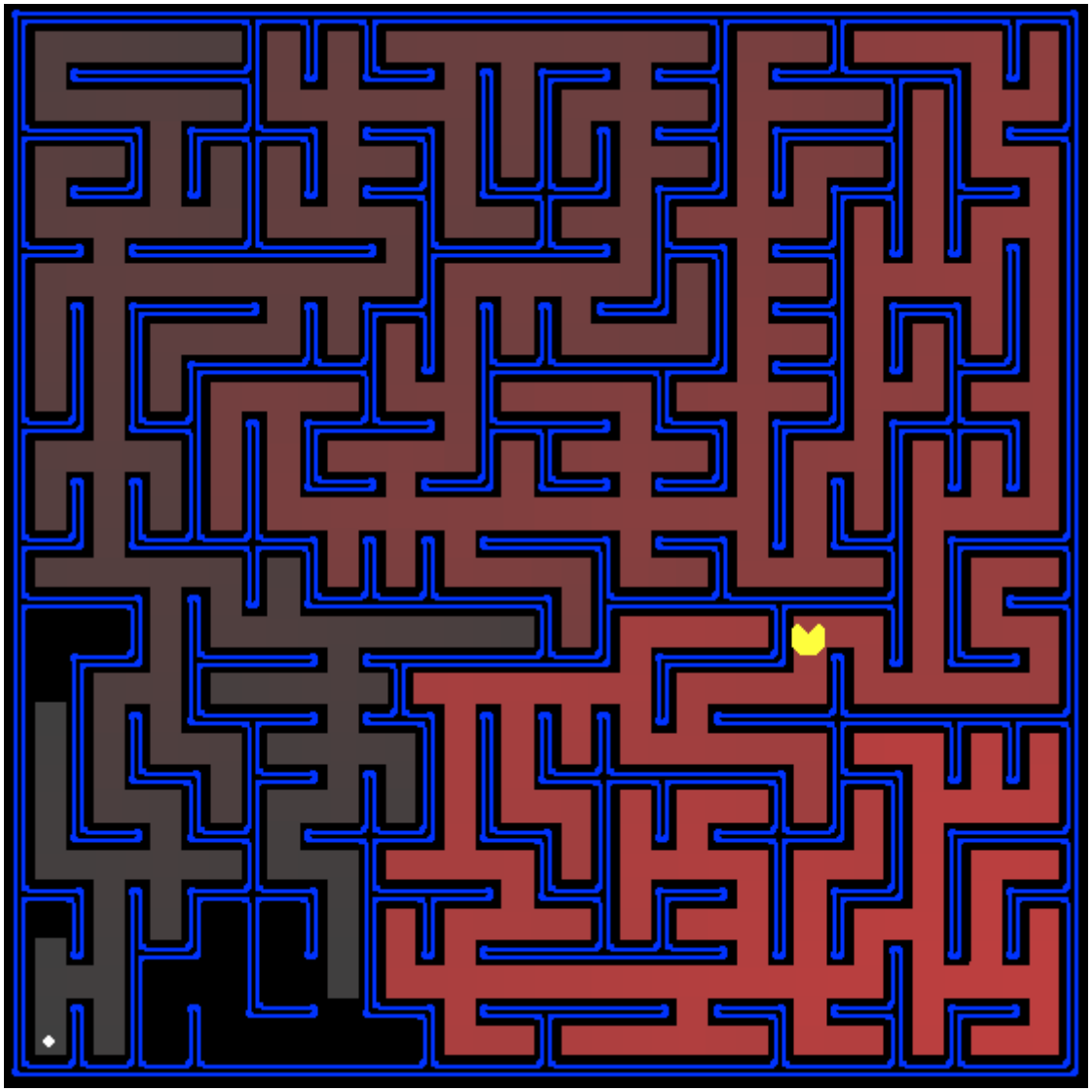
	Catálogo de compras da internet	Assistente matemático para demonstração de teoremas
Completamente Observável		
Determinístico		
Estático		
Episódico		
Discreto		
Agente único		

6. Em que consiste a Inteligência Artificial Distribuída? O que caracteriza um Sistema Multiagente? Ilustre duas aplicações para sistemas multiagentes.

7. Caracterize os agentes, conforme figura a seguir (que apresenta uma tipologia que define quatro tipos de agentes baseados nas suas habilidades de cooperar, aprender e agir autonomamente. Denominados por “agentes espertos, agentes colaborativos, agentes de aprendizado colaborativo e agentes de interface”), e ilustre aplicações para esses agentes.


Agentes
Espertos
Agentes
de aprendizado
colaborativo
Cooperar
Aprender
Agentes
Colaborativos
Operar
Autonomamente
Agentes de
Interface

Introdução



O projeto Pac-Man foi desenvolvido na Universidade de Berkeley para a disciplina de Introdução à Inteligência Artificial. A ideia do projeto é aplicar algumas técnicas clássicas de IA para jogar Pac-Man. O foco do projeto não é desenvolver IA para vídeo games, mas servir como ferramenta dá apoio para o ensino e aprendizagem de conceitos fundamentais de IA. Esses conceitos são amplamente usados para resolver problemas no mundo real, tais como processamento de língua natural, visão computacional, aprendizagem automática e robótica.

Neste trabalho, o agente Pacman tem que encontrar caminhos no labirinto, tanto para chegar a um destino quanto para coletar comida eficientemente. O objetivo do trabalho será programar algoritmos de busca e aplicá-los ao cenário do Pacman.

O código desse trabalho consiste de diversos arquivos Python, alguns dos quais você terá que ler e entender para fazer o trabalho. O código pode ser baixado nesse arquivo zip.

Arquivos que devem ser editados:

search.py	Onde ficam os algoritmos de busca.
searchAgents.py	Onde ficam os agentes baseados em busca.

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 do mundo do Pacman. Este arquivo descreve vários tipos auxiliares como AgentState, Agent, Direction e Grid.
util.py	Estruturas de dados úteis para implementar algoritmos de busca.

Arquivos 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

Nesse exercício do trabalho vamos implementar três algoritmos de busca não informada (busca em profundidade, busca em largura e busca de custo uniforme). Se você nunca programou em python, não se assute. Python é uma linguagem de programação fácil de aprender se você já tem alguma familiaridade com outras linguagens de programação, e um dos métodos de busca será implementado com auxílio do professor. Se precisar de ajuda, tem um tutorial aqui.

Bem-vindo ao Pacman

Depois de baixar o código (search.zip), descompactá-lo e entrar no diretório search, você pode jogar um jogo de Pacman digitando a seguinte linha de comando:

```
python pacman.py
```

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:

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

pacman.py tem opções que podem ser dadas em formato longo (por exemplo, --layout) ou em formato curto (por exemplo, -l). A lista de todas as opções pode ser vista executando:

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

Todos os comandos que aparecem aqui também estão em commands.txt, e podem ser copiados e colados.

Encontrando comida em um ponto fixo usando algoritmos de busca

---

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. Os algoritmos de busca para planejar o caminho não estão implementados -- este será o nosso trabalho. Para entender o que está descrito a seguir, pode ser necessário olhar esse glossário de objetos. Primeiro, verifique que o agente de busca SearchAgent está funcionando corretamente, rodando:

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

O comando acima faz o agente SearchAgent usar o algoritmo de busca tinyMazeSearch, que está implementado em search.py. O Pacman deve navegar o labirinto corretamente.

Agora chegou a hora de implementar os seus algoritmos de busca para o Pacman! Os pseudocódigos dos algoritmos de busca estão no livro-texto, e também nos slides. 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.

Importante: Todas as funções de busca devem retornar uma lista de ações que irão levar o agente do início até o objetivo. Essas ações devem ser legais (direções válidas, sem passar pelas paredes).

Dica: Os algoritmos de busca são muito parecidos. Os algoritmos de busca em profundidade, busca em extensão, busca 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 estratégia para retirar nós da borda. (Porém, implementar dessa forma

não é necessário).

Dica: Dê uma olhada no código dos tipo Stack (pilha), Queue (fila) e PriorityQueue (fila com prioridade) que estão no arquivo util.py!

Etapa 1: Vamos implementar 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 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 pilha Stack 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; pode ter comprimento 246 se forem colocados na ordem reversa).

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

Etapa 2: Implemente o algoritmo de busca em extensão (BFS) na função breadthFirstSearch do arquivo search.py. De novo, implemente a versão que não expande estados que já foram visitados. Teste 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? Senão, verifique a sua implementação. Se o seu código foi escrito de maneira correta, ele deve funcionar também para o quebra-cabeças de 8 peças (seção 3.2 do livro-texto) sem modificações.

```
python eightpuzzle.py
```

Variando a função de custo

A busca BFS vai encontrar o caminho com o menor número de ações até o objetivo. Porém, podemos querer encontrar caminhos que sejam melhores de acordo com outros critérios. Considere o labirinto mediumDottedMaze e o labirinto mediumScaryMaze. Mudando a função de custo, podemos fazer o Pacman encontrar caminhos diferentes. Por exemplo, podemos ter custos maiores para passar por áreas

com fantasmas e custos menores para passar em áreas com comida, e um agente Pacman racional deve poder ajustar o seu comportamento.

Etapa 3: Implemente o algoritmo de busca de custo uniforme (checando estados repetidos) na função uniformCostSearch do arquivo search.py. Teste seu código executando os comandos a seguir, onde os agentes têm diferentes funções de custo (os agentes e as funções são dados):

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
```

(Pergunta 4) O que acontece se você trocar StayWestSearchAgent por StayEastSearchAget nas chamadas acima?

A* search

Implementar busca em um grafo usando o A* na função vazia aStarSearch no arquivo earch.py. O algoritmo A* utiliza uma função heurística como um argumento. Uma função heurística leva dois argumentos: um estado no problema de pesquisa (o argumento principal) e o problema em si (para informações de referência). A função heurística de nullHeuristic em search.py é um exemplo trivial.

Você pode testar a sua implementação do A* no problema original de encontrar um caminho através de um labirinto para uma posição fixa, usando a heurística de distância de Manhattan (implementada já como manhattanHeuristic em searchAgents.py).

python pacman.py -l bigMaze - z p. 5 - SearchAgent - um fn = astar, heurística = manhattanHeuristic

Você verá que A* encontra a solução ideal ligeiramente mais rápida do que busca uniforme de custo (cerca de 549 vs 620 nós procurados expandidos em nossa implementação, mas laços em prioridade pode fazer seus números diferem ligeiramente).

(Pergunta 5) O que acontece na openMaze para as várias estratégias de pesquisa?