MAC0425/5739 - Inteligência Artificial

Exercício-Programa 1 - Agente de Busca para o Pac-Man

Prazo limite de entrega no e-Disciplinas: 20/05/2022 às 23:55

1 Introdução

Neste exercício-programa estudaremos a abordagem de contrução de agentes inteligentes capazes de resolver problemas através de uma busca no espaço de estados do jogo Pac-Man. Utilizaremos parte do material/código livremente disponível do curso UC Berkeley CS188 [1].

Os objetivos deste exercício-programa são:

(i) compreender a abordagem de resolução de problemas baseada em busca no espaço de estados, através da construção de um jogador autônomo de Pac-Man;

(ii) implementar algoritmos de busca informada e não-informada, bem como um algoritmo de busca online, e comparar seus desempenhos.

Figura 1: Cenário 1 - Ponto fixo de comida

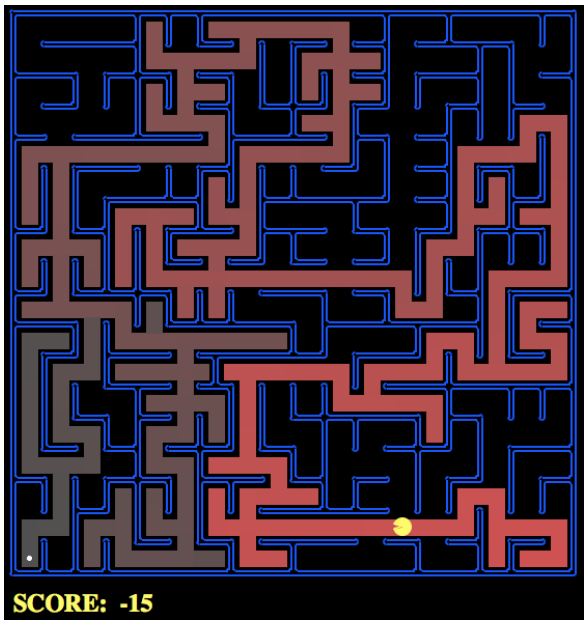
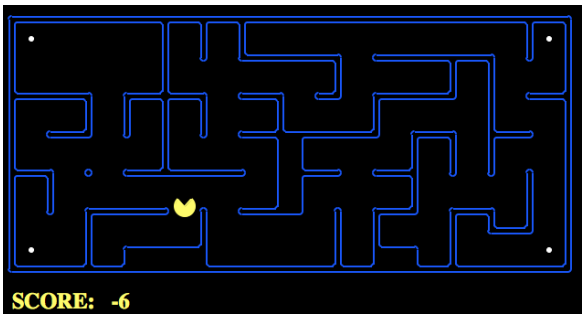


Figura 2: Cenário 2 - Cantos do labirinto



O jogo Pac-Man é um jogo eletrônico em que um jogador, representado por um boca que se abre e fecha, deve se mover 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 **esse exercício-programa consideraremos alguns**

[1] `http://ai.berkeley.edu/project_overview.html`

cenários nos quais utilizaremos algoritmos de busca para guiar o Pac-Man no labirinto a fim de atingir determinadas posições e coletar comida eficientemente.

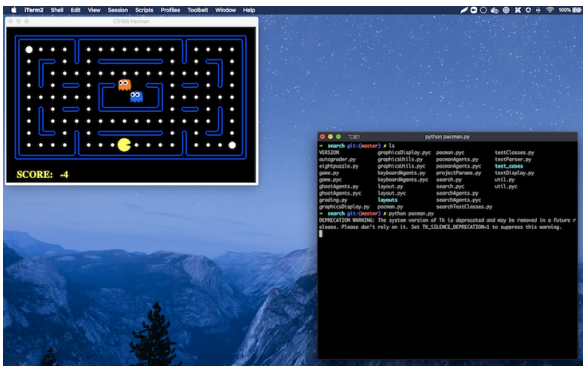
1.1 Instalação

Para a realização deste EP será necessário ter instalado em sua máquina a versão 2.7 do Python [2]. Faça o download dos **arquivo ep1.zip** disponível na página da disciplina no e-Disciplinas. Descompacte o arquivo ep1.zip e rode na raiz do diretório o seguinte comando para testar a instalação:

```
$ cd search/
$ python pacman.py
```

Após executar esta linha de comando uma nova janela deve ter sido aberta em seu desktop com o simulador sendo executado e o agente do Pac-Man parado (pois chamamos a linha de comando sem nenhum algoritmo especificado). Em exemplo da execução pode ser visto abaixo:

Figura 3: Exemplo de execução do simulador



[2] https://www.python.org/downloads/

2 Pac-Man como problema de busca

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

```
# arquivo search.py

class SearchProblem:

    def getStartState(self):
        """ Returns the start state for the search problem. """
        # ...

    def isGoalState(self, state):
        """ Returns True if and only if the state is a valid goal state. """
        # ...

    def getSuccessors(self, state):
        """ For a given state, this should return a list of triples
        (successor, action, stepCost), where 'successor' is a successor to
        the current state, 'action' is the action required to get there, and
        'stepCost' is the incremental cost of expanding to that successor. """
        # ...

    def getCostOfActions(self, actions):
        """ This method returns the total cost of a particular sequence
        of actions. The sequence must be composed of legal moves. """
        # ...
```

Note que embora a formulação de busca seja sempre a mesma para cada cenário de busca, a representação de estados varia de problema a problema. Por exemplo, veja a classe `PositionSearchProblem` no arquivo searchAgents.py e entenda a representação de estados utilizada para esse problema. Isso será importante pois você terá que implementar outra representação de estados para outro cenário de busca em um dos exercícios.

3 Implementação

Arquivos que você precisará editar:

- **search/search.py** onde os algoritmos de busca serão implementados;
- **search/searchAgents.py** onde os agentes baseados em busca serão implementados, isto é, onde formulamos os problemas que resolveremos nesse EP.

Arquivos que você precisará ler e entender (em qualquer pasta):

- **pacman.py** arquivo principal para executar o jogo. Descreve a representação de estado accessível para acesso de informações dos agentes de busca.
- **util.py** estruturas de dados para auxiliar a codificação dos 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. Essas implementações são necessárias para manter a compatibilidade com o aquivo de testes (**autograder.py**). Listas, tuplas, tuplas nomeadas, conjuntos e dicionários do Python podem ser utilizados sem problemas caso necessário.

4 Parte prática

Neste EP, você deverá resolver diferentes problemas de busca para dois ambientes diferentes do jogo Pac-Man, que chamaremos de Cenário 1 e Cenário 2. No Cenário 1 o Pac-Man deve procurar a comida que está localizada em uma única posição do labirinto. No Cenário 2, o Pac-Man deve procurar a comida que está distribuida nos quatro cantos do labirinto. Você deverá implementar os seguintes algoritmos de busca: Busca em Largura, Busca em profundidade, Busca de Custo Uniforme, Busca Heurística e um algoritmo de Busca Online chamado de LRTA*. Para isso, você deverá implementar algumas funções nos arquivos **search.py** e **searchAgents.py** (sinalizadas pelo comentário "*** ADD YOUR CODE HERE ***", contendo um exemplo de código que deverá ser eliminado, chamado de o código `util.raiseNotDefined()`.

Observação: Não esqueça de remover o código `util.raiseNotDefined()` ao final de cada função.

4.1 Cenário 1 - Encontrando a comida numa única localização do labirinto

Para o Cenário 1 a formulação do agente de busta já está disponível no arquivo **searchAgents.py** na classe `PositionSearchProblem`, isto é, a definição da tupla $\langle s0, S, A, T, g, M\rangle$ define o agente para esse ambiente.

Código 1 - Busca em Profundidade (DFS)

Implemente a busca em profundidade (DFS) no arquivo search.py na função `depthFirstSearch`. Para que a busca seja completa, implemente busca em grafo, que evita a expansão de estados previamente visitados. Para testar seu algoritmo rode os comandos:

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

Observação: note que o tabuleiro do jogo mostra os estados visitados pela busca por cor, isto é, quanto mais vermelho escuro mais cedo na busca o estado foi visitado (o que serve para ilustrar e visualizar a árvore de busca). Além disso, para todas essas buscas seu algoritmo DFS deve encontrar uma solução rapidamente, isto é, se demorar mais que alguns milisegundos algo provavelmente está errado no seu código!

Código 2 - Busca em Largura (BFS)

Implemente a busca em largura (BFS) no arquivo search.py na função `breadthFirstSearch`. Novamente, implemente busca em grafo. Para testar seu algoritmo rode os comandos:

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

Observação: se o Pac-Man se mover muito devagar tente usar a opção `--frameTime 0` na linha de comando.

Código 3 - Busca de Custo Uniforme

Implemente a busca de custo uniforme (UCS) no arquivo search.py na função `uniformCostSearch`. Para isso, você precisa adicionar custos variados às ações do agente UCS. No arquivo **searchAgents.py** estão disponíveis 2 implementações de agentes com custo variado: `StayEastSearchAgent` e `StayWestSearchAgent`. Teste esse algoritmo para os 2 agentes de custo variado. Novamente, implemente o UCS como busca em grafo. Para testar seu algoritmo rode o comando:

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

Código 4 - Busca heurística (A^*)

Implemente busca em grafo A^* no arquivo search.py na função `aStarSearch`. Para testar seu algoritmo rode o comando:

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

Código 4 - Busca heurística com Aprendizado em Tempo Real (LRTA*)

Esse exercício programa é obrigatório para os alunos de pós-graduação e optativo para os alunos da graduação, valendo um bonus de 2 pontos na média final. Nesse item você deverá implementar o algoritmo LRTA* (Learning Real-Time A*) (visto na Aula 6). Você deverá se basear no algoritmo da Figura 2 na Seção 6.1. do artigo **Learning in Real-Time Search: A Unifying Framework**, disponível em `http://www.aaai.org/Papers/JAIR/Vol25/JAIR-2504.pdf`. Implemente a busca (LRTA*) no arquivo search.py na função `learningRealTimeAStar`. Teste o algoritmo para cada tamanho de *layout* do Pac-Man com números de *trials* 10, 20 e 100. Use *lookahead* de 1.

Reporte em uma tabela para cada configuração (*layout*, número de *trials*) os seguintes resultados:

1. custo total do caminho
2. número de nós expandidos; e
3. estimativa final do custo do estado inicial no último *trial*.

Para testar seu algoritmo rode o comando:

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

Observação: Você pode passar a opção `-q` (ou `--quietTextGraphics`) para não iniciar a simulação gráfica do algoritmo.

4.2 Cenário 2 - Encontrando os cantos com comida do labirinto

Para este cenário, vamos executar e analisar os algoritmos implementados anteriormente, sem a necessidade de modificá-los. A única modificação que você deverá fazer é definir uma nova função heurística para o A*, como mencionado a seguir.

Código 5 - Formulação de problema de busca dos 4 cantos

Nesse exercício vamos implementar um novo problema de busca. Você deverá completar a implementação dos métodos da classe `CornersProblem` no arquivo searchAgents.py. Você deverá escolher uma representação de estados que codifique **somente a informação necessária** para detectar se todos os 4 cantos do labirinto foram visitados. Para testar a formulação do problema rode o comando:

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

Observação: Não utilize um objeto `GameState` como estado de busca! Se utilizar, seu código provavelmente ficará errado e muito lento.

Código 6 - Heurística para o problema dos 4 cantos

Além da nova formulação de problema, você deverá implementar uma heurística não trivial e consistente na função `cornersHeuristic`. Para testar sua heurística rode o comando:

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

Nesse exercício consideramos heurísticas triviais aquelas que devolvem 0 (zero). A heurística trivial não ajuda muito do ponto de vista da eficiência do algoritmo. Você pode usar a heurística trivial para compará-la com a sua proposta de heurística não-trivial.

O algoritmo LRTA* pode ser testado para esse problema rodando o comando:

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

5 Relatório

Após o desenvolvimento da parte prática, você deverá testar seus algoritmos e redigir um relatório, claro e sucinto (máximo de 4 páginas), fazendo uma análise comparativa de desempenho dos algoritmos implementados nos dois ambientes de teste. Assim, você deverá:

(a) compilar em tabelas as estatísticas das buscas implementadas em termos de nós expandidos e tamanho de plano;

(b) discutir as vantagens e desvantagens de cada método, no contexto dos dados obtidos;

(c) responder as questões teóricas;

(d) sugerir possíveis melhorias ao sistema e relatar dificuldades.

5.1 Questões

Questão 1 - Busca em Profundidade A ordem de exploração do espaço de estados seguiu conforme esperado? O Pac-Man de fato se move para todos os estados explorados em sua deliberação para encontrar uma solução para a meta?

Questão 2 - Busca em Largura A sua implementação de busca em largura encontra uma solução de custo mínimo? Por quê?

Questão 3 - Busca de Custo Uniforme Execute os comandos abaixo e explique sucintamente o comportamento os agentes `StayEastSearchAgent` e `StayWestSearchAgent` em termos da função custo utilizada por cada agente.

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

Questão 4 - Busca Heurística 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*?

Questão 5 - Busca Heurística LRTA* Compare os resultados obtidos para os testes do LRTA*. Quais características diferenciam esse algoritmo dos demais algoritmos implementados? O que você pode dizer sobre o valor estimado do estado inicial a medida que o número de *trials* aumenta? Com base somente nos resultados obtidos como você pode verificar se o o algoritmo convergiu?

6 Entrega

Você deve entregar um arquivo `EP1-NUSP-NOME-COMPLETO.zip` contendo **APENAS** os arquivos:

(1) **search.py** e **searchAgents.py** com as implementações da parte prática;

(2) **relatório** em formato PDF com as questões e comparação e discussão dos resultados (máximo de 4 páginas).

Não esqueça de identificar cada arquivo com seu nome e número USP! No código coloque um cabeçalho em forma de comentário.

7 Critério de avaliação

O critério de avaliação dependerá parcialmente dos resultados dos testes automatizados do autograder.py. Dessa forma você terá como avaliar por si só parte da nota que receberá para a parte prática. **Note que para o algoritmo LRTA* não há testes automatizados no autograder**. Avaliaremos esse algoritmo separadamente. Para rodar os testes automatizados, execute o seguinte comando:

```
$ python autograder.py
```

Com relação ao relatório, 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 dado as características teóricas de cada busca.

Parte prática (30 pontos)

- Cenário 1: autograder (12 pontos)
- Cenário 2: autograder (6 pontos)
- Código 4* (LRTA*) (12 pontos)

Relatório (30 pontos)

- Questões: (10 pontos)
- Comparação e Discussão: (20 pontos)