Trabalho Prático 1: Busca em Pac-Man

Luiz Chaimowicz

2016-09-08

Sumário

Esta especificação foi adaptada dos exercícios de Pac-Man de Berkeley, originalmente criados por John DeNero e Dan Klein.

Introdução

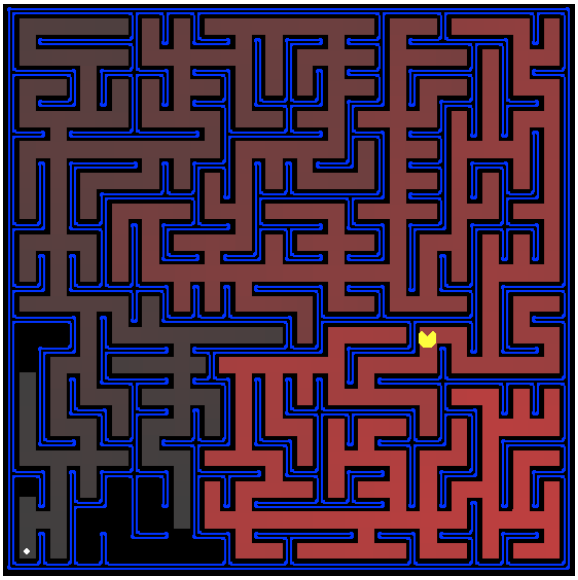

Figura 1: Ensine o Pac-Man a buscar no labirinto.

Neste projeto seu agente de Pac-Man buscará caminhos através de um labirinto, tanto para alcançar uma localização em particular quanto para coletar comida de forma eficiente. O objetivo é que você construa algoritmos de busca gerais e os aplique em cenários relacionados ao Pac-Man.

Este projeto inclui um avaliador automático para que você possa verificar suas respostas em seu computador. Execute-o com o comando:

```
python autograder.py
```

Note que alguns sistemas operacionais mais recentes podem ter apenas o Python 3 instalado. É possível verificar qual versão está instalada com o comando `python --version`. Não é incomum, quando o padrão for o Python 3, existir um binário `python2` disponível. Se esse for seu caso, substitua os comandos `python` deste documento por `python2`. A instalação do interpretador Python está fora do escopo deste documento.

O código deste projeto consiste de vários arquivos Python, alguns dos quais você precisará ler e compreender para completar essa lista de exercícios; outros arquivos você poderá ignorar com segurança. É possível baixar o código do moodle da disciplina.

Arquivos que você editará

- `search.py`: Onde todos seus algoritmos de busca residirão.
- `searchAgents.py`: Onde todos seus agentes baseados em busca residirão.

Arquivos potencialmente interessantes

- `pacman.py`: O ponto de entrada do Pac-Man. Descreve o tipo `GameState`, usado neste projeto.
- `game.py`: A lógica por trás do mundo. Define vários tipos de suporte, como `AgentState`, `Agent`, `Direction` e `Grid`.
- `util.py`: Estruturas de dados úteis para implementação de algoritmos de busca.

Arquivos de suporte que você pode ignorar

- `graphicsDisplay.py`: Gráficos.
- `graphicsUtils.py`: Suporte para gráficos.
- `textDisplay.py`: Gráficos ASCII.
- `ghostAgents.py`: Agentes para controle dos fantasmas.
- `keyboardAgents.py`: Interfaces com teclado para controle do Pac-Man.
- `layout.py`: Código para leitura e salvamento de arquivos de layout.
- `autograder.py`: Avaliador automático do projeto.
- `testParser.py`: Parser dos arquivos de teste e solução do avaliador automático.
- `testClasses.py`: Classes de teste.
- `test_cases/`: Diretório contendo os casos de teste de cada questão.
- `searchTestClasses.py`: Classes de avaliação específicas para este projeto.

Mecânica e submissão

Você irá completar partes de `search.py` e `searchAgents.py` durante a elaboração desta lista de exercícios. Você deve submeter esses arquivos com seu código e comentários. Por favor *não modifique* nem submeta nenhum outro arquivo desta distribuição.

Avaliação

Seu código será avaliado automaticamente para verificação de corretude. Por favor *não modifique* o nome de nenhuma função ou classe dentro do código. Note, no entanto, que sua pontuação será baseada na avaliação do seu código e não no julgamento do avaliador automático. Se necessário, revisaremos sua submissão individualmente para garantir que você receba o devido crédito.

Honestidade acadêmica

Nós verificaremos as submissões par-a-par em busca de redundância. Se você copiar o código de alguém e submetê-lo com pequenas mudanças, nós saberemos. Os detectores de plágio são difíceis de enganar. Portanto, nem tente. Não nos desaponte, pois confiamos que vocês submeterão somente o fruto de seu trabalho.

Obtendo ajuda

Se você tiver alguma dúvida, contate a organização do curso: temos o moodle e possibilidade de agendamento para tirar dúvidas. O objetivo deste projeto é que ele seja compensador e que você aprenda, não frustrante. No entanto, só saberemos como ajudar se você nos perguntar.

Discussão

Ao postar suas dúvidas ou discutir com colegas, tente não dar respostas diretas. O aprendizado é melhor quando entendemos o problema a ponto de elaborar respostas.

Bem-vindo ao Pac-Man

Após baixar, descompactar e mudar para o diretório do código, será possível jogar um jogo de pacman com o comando

```
python pacman.py
```

Pacman vive em um mundo azul de corredores tortuosos e lanches deliciosos. Navegar este mundo com eficiência será o primeiro passo do pacman para dominar este ambiente.

O agente mais simples implementado em `searchAgents.py` é o `GoWestAgent`, que sempre anda para o oeste/esqueda. De vez em quando ele consegue até ganhar:

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

Mas as coisas ficam mais difíceis se ele tiver que virar por algum motivo:

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

Se o Pacman ficar preso, você pode sair do jogo pressionando CTRL-C no terminal.

Em breve, seu agente não só resolverá o `tinyMaze`, mas *qualquer* outro labirinto que você quiser.

Note que `pacman.py` suporta opções que podem ser especificadas de forma longa (`--layout`, por exemplo) ou curta (`-l`). É possível ver a lista de opções com:

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

Todos os comandos que aparecem neste documento estão presentes no arquivo `commands.txt`, para facilitar. Em sistemas UNIX (que inclui Mac OS X), é possível executar os comandos em ordem com `bash commands.txt`.

Nota: Caso você encontre algum erro relacionado a `Tkinter`, visite http://tkinter.unpythonic.net/wiki/How_to_install_Tkinter.

Parte 1: Busca em profundidade (DFS)

No arquivo `searchAgents.py` você encontrará um agente do tipo `SearchAgent` que planeja um caminho através do mundo do Pac-Man e executa esse caminho passo a passo. Os algoritmos que formulam o caminho ainda não foram implementados, pois esse é o seu trabalho. À medida que você resolver as partes abaixo, pode ser interessante consultar o glossário de objetos.

Primeiro teste se o agente de busca funciona corretamente com

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

O comando acima instrui o `SearchAgent` a usar o algoritmo de busca `tinyMazeSearch`, implementado em `search.py`. O Pac-Man deveria navegar com sucesso nesse labirinto.

É chegada a hora de implementar funções de busca genéricas para auxiliar Pac-Man em seu planejamento! O pseudo-código dos algoritmos de busca estão presentes nos slides vistos em aula e no livro-texto. Lembre-se que um nó na busca deve conter não só o estado, mas também toda a informação necessária para reconstruir o caminho que leva àquele estado.

Note: Todas as suas funções de busca devem retornar uma lista de ações que levarão o agente do início ao objetivo. Todas as ações devem ser movimentos legais em direções válidas e sem cruzar paredes.

Note também que será necessário usar as classes `Stack`, `Queue` e `PriorityQueue` providas no módulo `util.py`. A implementação dessas estruturas de dados possui propriedades necessárias para compatibilidade com o avaliador automático.

Dica: Cada algoritmo é muito parecido. Os algoritmos para DFS, BFS, UCS e A* diferem apenas em detalhes de como a borda é gerenciada. Portanto, se concentre em implementar o DFS corretamente e o resto será relativamente simples. Repare que é possível implementar um método de busca genérico configurável para diferentes estratégias de gerenciamento de nós. (Sua implementação não precisa ser assim.)

Implemente o algoritmo de busca em profundidade (DFS) na função `depthFirstSearch` de `search.py`. Para tornar seu algoritmo completo, escreva a versão de busca em grafos do DFS, que evita expandir quaisquer estados já visitados.

Seu código deve ser capaz de encontrar soluções rapidamente para:

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

```
python pacman.py -l mediumMaze -p SearchAgent
```

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

O "tabuleiro" realçará os estados explorados e a ordem na qual eles foram explorados (vermelho mais forte significa exploração mais antiga). A ordem de exploração do seu algoritmo é a mesma

que você esperava? O Pac-Man realmente visita todos os quadrados explorados a caminho de seu objetivo?

Dica: Se você usar a classe `Stack`, a solução encontrada pelo seu DFS deve ter comprimento 130 (supondo que você adiciona sucessores à borda na ordem dada por `getSuccessors`; encontrará 246 caso adicione na ordem reversa). Essa solução é de custo mínimo? Se não, pense no que sua busca poderá estar fazendo errado.

Parte 2: Busca em largura (BFS)

Implemente o algoritmo de busca em largura (BFS) na função `breadthFirstSearch` no arquivo `search.py`. Evite visitar nós já visitados. Teste o código da mesma forma que testou a busca em profundidade.

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

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

O BFS encontra uma solução de custo mínimo? Se não, verifique sua implementação.

Dica: Se a taxa de atualização for muito lenta, tente usar a opção `--frameTime 0`.

Uma forma de validar se seu código de busca está suficientemente genérico é testá-lo com o problema de busca do eight-puzzle. Ele deveria funcionar sem qualquer mudança.

```
python eightpuzzle.py
```

Parte 3: Busca de Custo Uniforme

Apesar do BFS encontrar um caminho com o menor número de ações até o objetivo, podemos estar interessados em outra definição de "melhor". Considere os mapas `mediumDottedMaze` e `mediumScaryMaze`.

Ao mudar a função de custo podemos encorajar o Pac-Man a encontrar caminhos diferentes. Por exemplo, podemos considerar que passar por regiões infestadas de fantasmas é mais caro e podemos considerar mais barato passar por regiões com comida. Nesse caso, um agente racional deveria ajustar seu comportamento de acordo.

Implemente o algoritmo de busca de custo uniforme na função `uniformCostSearch` em `search.py`. Sugerimos que procure no módulo `util.py` por estruturas de dados que sejam úteis em sua implementação. Uma vez implementado, você deveria observar comportamento correto nos três layouts abaixo, onde os agentes abaixo são todos agentes UCS que diferem somente na função de custo que usam (os agentes e funções de custo serão implementados por você):

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

python pacman.py -l mediumDottedMaze -p StayEastSearchAgent

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

Você deveria encontrar caminhos de custo muito baixo e muito alto para `StayEastSearchAgent` e `StayWestSearchAgent` respectivamente, devido a suas funções de custo exponenciais. (Veja `searchAgents.py` para mais detalhes.)

Parte 4: Busca com A*

Implemente o algoritmo A* na função `aStarSearch` no módulo `search.py`. A* recebe uma função heurística como argumento. As heurísticas recebem dois argumentos: um estado no problema de busca (o argumento principal) e o problema de busca. A função heurística `nullHeuristic` é um exemplo trivial e pode ser encontrada no módulo `search.py`.

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

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

Você deveria ver que o A* encontra a solução ligeiramente mais rápido que a função de busca uniforme (cerca de 549 versus 620 nós de busca expandidos na nossa implementação). O que acontece no `openMaze` para as diversas estratégias de busca?

Parte 5: O problema dos cantos: representação

O verdadeiro poder do A* se tornará aparente em um problema de busca mais desafiador. Agora é hora de formular um novo problema e projetar uma heurística para ele.

Em labirintos de cantos há quatro pontos, um em cada canto do labirinto. Nosso novo problema de busca é encontrar o menor caminho pelo labirinto que toca todos os quatro cantos (haja comida naquele ponto ou não). Note que para alguns labirintos, como `tinyCorners`, o menor caminho nem sempre passa pela comida mais próxima primeiro! O menor caminho em `tinyCorners` tem 28 passos.

Antes de trabalhar nesta questão é desejável que você tenha respondido a questão 2, pois esta questão se baseia no conhecimento adquirido naquela.

Implemente o problema de busca `CornersProblem` em `searchAgents.py`. Você precisará encontrar uma representação de dados que codifique toda informação necessária para detectar se todos os quatro cantos foram encontrados. Agora, seu agente deveria resolver os problemas:

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

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

Para receber a pontuação completa você deve definir uma representação de dados que não codifica informação irrelevante (como a posição dos fantasmas, onde está a comida, etc.). Em particular, não use um `GameState` como estado de busca. Seu código ficará **muito** lento se você fizer (e também estará errado).

Dica: As únicas partes do estado que você deveria referenciar em sua implementação são a posição inicial do Pac-Man e a localização dos quatro cantos.

Nossa implementação de `breadthFirstSearch` expande um pouco menos de 2000 nós no mapa `mediumCorners`. No entanto, heurísticas podem reduzir a quantidade de busca necessária.

Parte 6: O problema dos cantos: Heurística

Somente inicie a solução deste problema após resolver a questão 4.

Implemente uma heurística não trivial e consistente para o problema `CornersProblem` no módulo `cornersHeuristic`.

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

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

Admissibilidade *versus* Consistência

Lembre-se que heurísticas são funções que recebem estados e retornam números que estimam o custo do objetivo mais próximo. Heurísticas mais eficientes retornam valores mais próximos ao custo real de alcançar o objetivo. Para ser admissível, os valores da heurística devem ser lower bounds do custo do menor caminho ao objetivo mais próximo (além de serem não negativos). Para ser consistente, também deve ser verdade que se uma ação tem custo c, então tomar aquela ação só pode fazer o valor da heurística diminuir de no máximo c.

Lembre-se que admissibilidade não é suficiente para garantir corretude em busca em grafos – é necessário que haja consistência. No entanto, heurísticas admissíveis são normalmente consistentes, principalmente quando obtidas de relaxações de problemas. Portanto, normalmente é melhor fazer brainstorm de heurísticas admissíveis. Quando você tiver uma heurística admissível que funciona bem, aí você pode testar se ela é também consistente. A única forma de garantir consistência é com uma prova. No entanto, inconsistência pode ser detectada através da verificação de que para cada nó expandido, seus sucessores possuem valor maior ou igual. Ademais, se UCS e A* retornarem caminhos de tamanhos diferentes, sua heurística é inconsistente.

Heurísticas não triviais

As heurísticas triviais são aquelas que sempre retornam zero (UCS) e a heurística que completa o custo real. A primeira não te economizará nada, enquanto a segunda fará um timeout no avaliador automático. Você quer uma heurística que reduza o tempo total de computação. No entanto, neste problema o avaliador automático somente verificará a contagem de nós expandidos (além de forçar um limite de tempo razoável).

Pontuação

Sua heurística deve ser não trivial, não negativa e consistente para ser pontuada. Certifique-se de que ela retorna 0 em todo estado final e que nunca retorna nenhum valor negativo. Dependendo de quantos nós você expandir, sua pontuação para esta parte será:

Número de nós expandidos	Pontuação
mais de 2000	0%
até 2000	33%
até 1600	66%
até 1200	100%

Novamente: se a heurística retornar valores negativos você não receberá quaisquer pontos.

Parte 7: Comendo toda a comida: Heurística

Somente inicie a solução deste problema após resolver a questão 4.

Agora é hora de resolver um problema difícil: comer toda a comida no menor número de passos possível. Para isso, precisaremos de uma nova definição de que formaliza o problema da comida: `FoodSearchProblem` em `searchAgents.py` (implementado para você). Uma solução é definida como um caminho que coleta toda a comida do mundo do Pac-Man. Neste caso em particular as soluções não levam em consideração fantasmas ou a comida "do poder"; soluções só dependem da posição das paredes, comida normal e o Pac-Man. Se você tiver escrito seus códigos de busca corretamente, A* com a heurística vazia (equivalente a busca de custo uniforme) deveria encontrar uma solução ótima para `testSearch` sem quaisquer mudanças de seu código (com custo total de 7).

```
python pacman.py -l testSearch -p AStarFoodSearchAgent
```

Note: `AStarFoodSearchAgent` é um atalho para `-p SearchAgent -a fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic`.

Você deveria perceber que UCS começa a ficar lento até para o simples problema `tinySearch`. Como referência, nossa implementação leva 2.5 segundos para encontrar um caminho de tamanho 27 após expandir 5057 nós de busca.

Complete `foodHeuristic` em `searchAgents.py` com uma heurística consistente para `FoodSearchProblem`. Teste o seu agente no mundo `trickySearch`:

```
python pacman.py -l trickySearch -p AStarFoodSearchAgent
```

Nosso agente UCS encontra a solução ótima em aproximadamente 13 segundos, explorando mais de 16000 nós.

Pontuação

Qualquer heurística consistente não negativa receberá 25%. Certifique-se de que ela retorna 0 em todo estado final e que nunca retorna nenhum valor negativo. Dependendo de quantos nós você expandir, sua pontuação para esta parte será:

Número de nós expandidos	Pontuação
mais de 15000	25%
até 15000	50%
até 12000	75%
até 9000	100%
até 7000	125%

Novamente: se a heurística retornar valores negativos você não receberá quaisquer pontos. Você consegue resolver `mediumSearch` em pouco tempo? Se sim, ou nós estamos muito impressionados, ou sua heurística é inconsistente.

Parte 8: Busca sub-ótima

Às vezes, mesmo com A* e uma boa heurística, encontrar o caminho ótimo por todos os pontos é difícil. Nesses casos, nós ainda gostaríamos de encontrar um caminho razoavelmente bom de forma rápida. Nesta parte você implementará um agente que sempre, de forma gulosa `;)` come a comida mais próxima. `ClosestDotSearchAgent` já foi implementado para você em `searchAgents.py`. No entanto, falta uma função importante, que encontra o caminho para o ponto mais próximo.

Implemente a função `findPathToClosestDot` em `searchAgents.py`. Nosso agente resolve esse labirinto (de forma sub-ótima) em menos de um segundo com um custo de 350:

```
python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5
```

Dica: A forma mais rápida de completar `findPathToClosestDot` é preenchendo `AnyFoodSearchProblem`, que está sem seu teste de estado objetivo. Depois, resolve o problema com uma função de busca apropriada. A solução deveria ser curta!

Seu agente `ClosestDotSearchAgent` nem sempre encontrará o menor caminho possível no labirinto. Certifique-se de entender o motivo e tente bolar um exemplo pequeno onde repetidamente ir para o ponto mais próximo não resulta em encontrar o menor caminho.

Submissão

Você deve submeter sua implementação pelo moodle da disciplina. Para submissão execute o script `submit.py`. Ele coletará os arquivos que devem ser submetidos e gerará um arquivo zip para submissão no moodle. Por favor não tente usar qualquer programa de compressão. Em suma: não tente ser esperto. O script foi escrito para garantir consistência na submissão.

```
python prepare-submission.py
```

O script perguntará por seu número de matrícula e gerará um arquivo cujo nome consistirá de seu número de matrícula com a extensão `.zip`. Dentro desse arquivo serão adicionados os arquivos `search.py` e `searchAgents.py`.

Glossário de objetos

Aqui vai um glossário dos objetos chave usados no código relacionado 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 do problema. Você interagirá com `SearchProblems` apenas através dos métodos definidos em `search.py`.

`PositionSearchProblem` (`searchAgents.py`)

Um tipo específico de `SearchProblem` com o qual você trabalhará – corresponde a buscar por um único ponto de comida em um labirinto.

`CornersProblem` (`searchAgents.py`)

Um tipo específico de `SearchProblem` que você definirá – corresponde a buscar um caminho por todos os cantos de um labirinto.

`FoodSearchProblem` (`searchAgents.py`)

Um tipo específico de `SearchProblem` com o qual você trabalhará – corresponde a buscar uma forma de comer todos os pontos de comida em um labirinto.

Função de busca

Uma função de busca é uma função que recebe uma instância de `SearchProblem` como parâmetro, executa algum algoritmo e retorna uma sequência de ações que levarão a um objetivo. Exemplos de funções de busca são `depthFirstSearch` e `breadthFirstSearch`, que você terá que escrever. Você poderá se basear na função `tinyMazeSearch`, uma função bem ruim que só funciona corretamente em `tinyMaze`.

`SearchAgent`

`SearchAgent` é uma classe que implementa um `Agente` (um objeto que interage com o mundo) e planeja através de uma função de busca. Primeiro o `SearchAgent` usa a função de busca fornecida para montar um plano de ações que o levará ao estado objetivo e depois executa, uma de cada vez, essas ações.