Relatório do EP2 - Inteligência Artificial

Juliano Garcia de Oliveira Nº USP: 9277086

14 de Maio, 2018

OBS: O meu EP usa a *Priority Queue* com a função *update()*, então modifiquei o arquivo *util.py*. Poderia ter criado uma classe que herda de PQ e adiciona esse novo método, mas acho que mudar direto a definição de PQ é melhor.

Questões

1. (reativo) Como é o desempenho do seu agente? É provável que muitas vezes ele morra com 2 fantasmas no tabuleiro default, a não ser que a sua função de avaliação seja muito boa.

   A função é razoávelmente boa, sendo que testando 100 vezes no layout padrão (com 2 fantasmas), ele vence 80% das vezes, e testando 100 vezes no layout 'trickyLayout', com 4 fantasmas, ele vence 42% das vezes. Os comandos que usei para fazer estes são, respectivamente:

   ```
   $ python pacman.py -p ReflexAgent -k 2 -f -q -n 100
   $ python pacman.py -p ReflexAgent -k 4 -l trickyClassic -f -q -n 100
   ```

2. (minimax) Por que o Pac-Man corre para o fantasma mais próximo neste caso?

   Porque neste caso, o Pac-Man começa com a pontuação 0, e a pontuação de quando ele perde é a pontuação atual -500. Portando, ao fazer o Minimax, quanto mais tempo ele ficar jogando porém morrer, menor vai ser a pontuação, mas ele precisa maximizar a pontuação, então correr para o fantasma mais próximo garante que essa pontuação que ele perderia inicialmente por ficar andando no tabuleiro seja a menor possível, para que a pontuação final seja a maior possível.

3. (minimax) Por que o agente reativo tem mais problemas para ganhar que o agente minimax?

   Porque o agente reativo só consegue levar em conta uma única configuração ação-estado, enquanto no Minimax, além de conseguir olhar vários estados e pegar o melhor (dependendo da profundidade), no Minimax também é levado em conta as ações que o jogadores adversários podem fazer (MIN), o que dá um melhor "controle" do jogo, e dá a estratégia ótima caso todos os jogadores estejam jogando otimamente. Mas isso depende bastante da função do agente reativo, no meu caso, várias vezes o agente reativo ganha mais do que o Minimax (com profundidade 2).

4. (reativo) Que mudanças poderiam ser feitas na função de avaliação (evaluationFunction) para melhorar o comportamento do agente reativo?

   Levar em conta os possíveis movimentos dos fantasmas, quantidade de comida, possivelmente caminhos entre os fantasmas de modo a minimizar a chance de ser encurralado, limpar uma parte mais "cheia" do mapa primeiro, etc.

5. (alpha e beta) Faça uma comparação entre os agentes Minimax e AlphaBeta em termos de tempo e número de nós explorados para profundidades 2, 3 e 4.

   Abaixo estão os resultados, podemos ver que o AlphaBeta pode melhorar bastante o tempo conforme aumenta a profundidade. Os testes dos nós foram gerados modificando o código para verificar quantas vezes a função **generateSuccessor()** for chamada.

Profundidade	Tempo (Minimax)	Tempo (AlphaBeta)
2	2.2s	2.09s
3	5.73s	3.74s
4	60.40s	38.55s

Profundidade	Nós explorados (Minimax)	Nós explorados (AlphaBeta)
2	11333	10230
3	29325	20332
4	278350	180445

   Dividindo os resultados do Minimax pelo AlphaBeta, temos, em ambos os casos, uma razão por volta de 1.5 aproximadamente.

6. (expectimax) Por que o comportamento do expectimax é diferente do minimax?

   Porque o Minimax considera que os fantasmas são jogadores ótimos, enquanto no expectimax os fantasmas são considerados como jogadores aleatórios, isto é, não estão tentando minimizar a recompensa do Pac-Man, eles apenas escolhem uma de suas ações válidas aleatoriamente, e o Expectimax usa essa informação para calcular a média ponderada de cada estado, e pegar o estado que dê o maior valor (recursivamente), e é limitado pela profundidade, assim como o Minimax.

7. (função de avaliação) Inclua uma explicação sobre a sua função de avaliação.

   A minha função de avaliação de estados leva em conta basicamente as seguintes variáveis de um estado:

   - a = *score* do estado
   - b = quantidade de cápsulas no estado
   - c = quantidade de *foods*

   - d = distância até a *food* mais próxima
   - e = distância até o *ghost* mais próximo que não está *scared*
   - f = distância até o *ghost* mais próximo no estado *scared*

   A função de avaliação retorna a seguinte combinação linear desses valores:

   $eval(estado) = a - 2 \cdot d - 1.7 \cdot \frac{1}{e} - 5 \cdot f - 20 \cdot b - 5 \cdot c$

   Os valores foram escolhidos com base em alguns testes e na importância de cada um, de acordo com a estratégia que quero tomar. Inicialmente, uso o valor do *score* e subtraio alguns valores dele. Subtraindo a distância até a *food* mais próxima, quanto mais próximo de uma *food* o Pac-Man está, menos ele subtrai, portanto maior é o valor da função. Como estar próximo de um fantasma que não está *scared* é ruim, pego o inverso desse valor e subtraio, portanto quanto mais perto de um fantasma o Pac-Man está naquele estado, maior será $\frac{1}{e}$, portanto será subtraído um numero maior, logo a função de avaliação é menor quanto mais próximo de um fantasma o Pac-Man está

(isto é, estados onde o Pac-Man fica mais próximo de um fantasma não são desejáveis). Da mesma forma que a distância até a *food* mais próxima, se o Pac-Man está perto de um *ghost* que está *scared*, é desejável que ele fique mais próximo dele para comê-lo e obter mais pontos. E como ele ganha mais pontos comendo um fantasma, dei um peso maior para essa variável do que a distância até a comida. Em seguida, para a quantidade de cápsulas dei um peso razoavelmente grande, porque comer cápsulas permite comer fantasmas, o que pode dar bastante pontos. Mas o peso dessa variável não é tão grande a ponto de fazer o Pac-Man sempre preferir comer uma cápsula do que uma *food*. E finalmente, dou um peso razoável para a quantidade de comidas, assim quanto menos comidas tiver no estado, melhor será o estado segundo essa função de avaliação, levando o Pac-Man a comer sempre que puder.