Trabalho Prático 2: Competição Pac-Man

Inteligência Artificial

Thiago Silva Vilela

1 Introdução

O trabalho prático consiste em desenvolver agentes inteligentes para uma variação competitiva do jogo Pac-Man. Nesse jogo, controlamos dois agentes em um labirinto com dois territórios. Os dois agentes devem colaborar para defender seu território e atacar o território dos adversários, pegando o maior número possível de *pacdots* do inimigo.

2 Classificação do problema

De acordo com o livro texto, o problema pode ser classificado da seguinte forma:

- **Parcialmente observável**, uma vez que o agente não possui conhecimento do estado completo do ambiente. Mais especificamente, o agente não conhece a posição de seus inimigos, a não ser que eles estejam suficientemente próximos.
- **Multi-agente**, uma vez que certo agente competirá com dois inimigos por uma maior pontuação, e contará com um aliado para ajudá-lo a maximizar sua pontuação. O ambiente é multi-agente competitivo e multi-agente cooperativo.
- **Determinístico**, uma vez que o próximo estado do ambiente pode ser previsto através do estado atual e da ação escolhida pelo agente. Vale lembrar que, na definição de ambientes determiníticos usada pelo livro texto, ignoramos as incertezas que aparecem somente pelas ações dos outros agentes. O ambiente poderia ser considerado estocástico caso fossem consideradas essas incertezas.
- **Sequencial**, uma vez que a escolha de certa ação em um dado momento afeta todas as decisões futuras.
- **Semi-dinâmico**, uma vez que o agente tem um tempo fixo (1 segundo) para tomar uma decisão e, durante esse tempo, o ambiente não se altera. Dessa forma, o agente sabe que o ambiente não muda enquanto ele está deliberando, mas precisa se preocupar com a passagem do tempo.
- **Discreto**, uma vez que o ambiente é discreto e existe um número finito de estados e ações distintas no problema.

3 Modelagem dos agentes

Nessa seção serão descritas as estratégias e algoritmos utilizados na implementação dos dois agentes desenvolvidos. Um dos agentes foi feito para atacar o território inimigo e comer *pacdots*, enquanto o segundo foi feito para defender seus próprios *pacdots*.

3.1 Agente Ofensivo

O agente ofensivo utiliza simulações de Monte Carlo para avaliar cada ação possível em determinado momento. A ação escolhida para execução é aquela que foi mais bem avaliada.

Normalmente, durante uma simulação, o jogo é jogado aleatoriamente por todos os agentes até que ele termine, e o resultado obtido será, por exemplo, o número total de vitórias obtidas após a execução de várias simulações. No nosso Pac-Man competitivo, no entanto, essa abordagem não é viável: o número de ações tomadas até o fim do jogo pode ser muito alto e não temos visibilidade constante do nosso oponente para simular suas ações. A solução para o primeiro desses problemas é bastante utilizada em aplicações de tempo real, e consiste em realizar as simulações somente até uma certa profundidade d, ou seja, serão simuladas somente d ações de cada agente. Após essa simulação parcial, é necessário utilizar uma função de avaliação no último estado obtido. A solução adotada para o segundo problema foi fixar as ações de todos os outros agentes como **STOP**. Dessa forma, na prática, será necessário simular somente as ações de um único agente (aquele que está decidindo que ação tomar). Essa solução, apesar de bastante simples, mostra resultados promissores se combinada a uma boa função de avaliação.

Dois parâmetros importantes utilizados na simulação de Monte Carlo implementada são a profundidade da simulação a ser utilizada e o número de simulações usadas para avaliar certo estado. Geralmente, quanto maior o valor desses parâmetros, melhores serão os resultados obtidos. Dessa forma, é importante encontrar valores para esses parâmetros que permitam bons resultados e deixem o tempo de execução viável.

A simulação aleatória realizada pelo agente precisa de alguns cuidados para que seja mais eficiente. Não é interessante que o agente escolha sempre aleatoriamente entre todas as ações possíveis a cada passo da simulação. Isso pode levar o agente a ficar indo e voltando, ou até mesmo ficar parado em alguns momentos. Obviamente, na maioria das vezes, não é interessante que o agente fique parado ou indo e voltando entre as duas mesmas posições durante uma simulação. Dessa forma, na execução da simulação aleatória, o agente é proibido de ficar parado (executar a ação **STOP**) ou de reverter sua direção (tomar a ação de sentido contrário à direção corrente). Note que o agente pode realizar essas ações durante o jogo. O que é proibido é seu uso durante as simulações aleatórias.

A função de avaliação usada foi baseada naquela presente no *BaselineAgents*. Ela consiste na combinação linear de *features* e de pesos associados às *features*. As *features* consideradas são as seguintes:

- ***Score* do estado**: pega o *score* do estado final da simulação. O objetivo

dessa *feature* é maximizar a pontuação obtida pelo agente.

- **Distância ao *pacdot* mais próximo**: a distância do agente ao *pacdot* mais próximo no estado final da simulação. O objetivo dessa *feature* é, também, maximizar a possível pontuação obtida. Caso, durante as simulações, hajam dois estados finais com mesma pontuação, aquele onde o agente se encontra mais próximo de outro *pacdot* é mais bem avaliado.

- **Distância ao inimigo mais próximo**: a distância do agente ao inimigo mais próximo ao final da simulação. Essa *feature* permite ao agente fugir do inimigo caso esteja sendo perseguido, ao mesmo tempo em que tenta comer mais *pacdots*.

- **Pacman**: essa feature indica se o agente é um fantasma ou um pacman. Ela é utilizada somente em uma situação bastante específica, onde pode acontecer de o agente voltar ao campo de defesa para se defender e não conseguir mais voltar ao ataque pois o inimigo defensor fica na borda vigiando o agente (o agente acaba achando vantajoso ficar vivo em seu território). Nesse caso, o agente passa a priorizar o fato de ser pacman, e para de se defender na forma de fantasma, priorizando o ataque.

Os pesos das *features* são utilizados de forma dinâmica: normalmente, o maior peso vai para a *feature* referente ao *score*, um peso menor é dado à distância ao inimigo, um peso negativo é dado à distância ao *pacdot* mais próximo (quanto mais perto, melhor) e um peso nulo é dado à *feature* **Pacman**. Caso o oponente esteja no estado *scared*, o peso dado à distância ao inimigo é zerado. Caso nosso agente fique preso no campo de defesa, a *feature* **Pacman** ganha um peso alto.

Por fim, uma última característica foi adicionada ao agente ofensivo: a capacidade de evitar alguns becos sem *pacdots*. Antes de avaliar as ações possíveis, é realizado um pré-processamento na ação. Ela é expandida até uma profundidade 5 e, caso todos os caminhos expandidos a partir dessa ação terminarem em um beco sem *pacdots*, então o agente descarta a ação. Essa característica é particularmente importante no fim do jogo, quando existem poucos *pacdots* restantes e entrar em um beco pode ser bastante ruim, encurralando o agente.

3.2 Agente Defensivo

O agente defensivo é bastante simples. Ele funciona definindo posições alvo e se movendo em direção a elas, assim como o agente defensor do *SimpleTeam*. A definição do alvo, no entanto, é um pouco mais elaborada. O agente define alvos a fim de executar estratégias de alto nível. Essas estratégias são: patrulhar pontos centrais do mapa, verificar posição onde *pacdot* sumiu, perseguir oponente e vigiar *pacdots*. Segue uma breve explicação de cada estratégia, que explica quando ela é escolhida:

- **Patrulhar pontos centrais**: essa é a estratégia executada pelo agente defensivo na maior parte do tempo. Ela consiste em escolher um ponto na borda dos territórios dos dois times e se deslocar para tal ponto. Chamamos esses pontos de pontos de patrulha. Os pontos de patrulha para os quais o agente

poderá se deslocar são encontrados durante o tempo de pré-processamento. Para a escolha do ponto a ser visitado, calculamos algumas probabilidades. Associamos, a cada ponto de patrulha, uma probabilidade que corresponde à chande do agente escolher ir para tal posição. A probabilidade é calculada com base no inverso da distância do *pacdot* mais próximo à posição do ponto de patrulha em questão. Esses valores são normalizados de forma que a soma das probabilidades seja 1. Dessa forma, o agente defenser tem mais chance de se deslocar para pontos de patrulha com um *pacdot* próximo, uma vez que o oponente provavelmente tentará pegar esses *pacdots* primeiro. Sempre que o oponente come um *pacdot*, as probabilidades são recalculadas.

- **Verificar posição onde *pacdot* sumiu**: o agente implementado verifica, a cada iteração do jogo, se algum *pacdot* do seu território desapareceu, uma vez que temos informações das posições de todos os nossos *pacdots*. Caso algum tenha desaparecido, o agente irá se mover para a posição do *pacdot* que desapareceu. Dessa forma espera-se que, caso o inimigo passe pela patrulha do agente defensor, seja possível identificar rapidamente que nosso território foi invadido, assim como estimar a posição do atacante.

- **Perseguir oponente**: durante a patrulha e verificação previamente descritas, caso o agente veja um inimigo, ele passa a persegui-lo.

- **Vigiar *pacdots***: ao fim do jogo, quando o número de *pacdots* a ser defendido é pequeno, é mais vatajoso que o defensor se desloque entre esses *pacdots* ao invés de patrulhar a área central do mapa. Dessa forma, quando temos 4 ou menos *pacdots*, o agente passa a andar aleatoriamente de um para outro.

4 Análise de complexidade

Nessa sessão será analisada a complexidade dos dois agentes implementados.

4.1 Agente Ofensivo

Em termos de tempo, a operação mais custosa do agente ofensivo é avaliar as possíveis ações que serão tomadas usando simulações de Monte Carlo. Seja a o número de ações que o agente pode escolher. Para cada ação, serão realizadas n simulações aleatórias de profundidade d. A avaliação de um estado ao fim da simulação é bastante simples, e o maior custo dessa operação é encontrar o *pacdot* com menor distância à posição atual do agente. Considerando que exitem p *pacdots* no campo do adversário, essa avaliação é da ordem de $O(p)$, considerando que o método *getMazeDistance()* fornecido possua custo constante. Dessa forma, o custo de tomar uma decisão é da ordem de $O(adnp)$. No entanto, a é um número fixo (o agente pode escolher, no pior caso, entre 5 ações), e d e n também são fixados antes da execução. Sendo $c = 5 * d * n$, uma constante, a complexidade temporal será da ordem de $O(cp)$, ou $O(p)$.

A complexidade espacial do agente ofensivo é constante. Durante as simulações e avaliações, somente constantes são armazenadas e cada simulação aleatória utiliza somente uma cópia do *gamestate*. Dessa forma, considerando que o *gamestate* ocupe um espaço constante, o agente ofensivo possui ordem de complexidade espacial constante.

4.2 Agente Defensivo

Como pode ser observado na descrição do agente defensivo, a maioria das suas operações consiste em definir alvos. Tais operações são bastante simples e possuem, no geral, custo constante em termos de tempo. A operação mais custosa do defensor é calcular e recalcular as probabilidades de visitar os pontos de patrulha. Para calcular tal probabilidade é necessário encontrar, para cada ponto de patrulha, o *pacdot* mais próximo. Sendo x o número de pontos de patrulha (que varia dependendo do tamanho do mapa ou do número de paredes na área de encontro entre os dois territórios) e p o número de *pacdots* no nosso território, a complexidade temporal do agente defensivo é, no pior caso, $O(xp)$.

A complexidade espacial do agente ofensivo é $O(p + c)$, sendo p o número de *pacdots* que serão defendidos e c o número de pontos de patrulha no centro do mapa. Isso ocorre pois o agente defensivo sempre armazena a última observação dos *pacdots* a serem defendidos (a posição de cada um dos p *pacdots*) e um dicionário com as posições de cada um dos c pontos de patrulha, associados com as probabilidades de escolher cada um deles.

5 Análise de desempenho e discussão dos resultados

O time de agentes desenvolvido, batizado de *MonteCarloTeam*, foi testado contra os dois times fornecidos: o *SimpleTeam* e o *RandomTeam*. Além disso, foram realizados testes do *MonteCarloTeam* contra ele mesmo. Para cada oponente, foram realizados 20 jogos com o *MonteCarloTeam* no território vermelho e 20 jogos com o *MonteCarloTeam* no território azul. Isso foi feito para avaliar o efeito do território no agente implementado. Os mapas foram gerados aleatoriamente para cada jogo, e o território de cada time possuia sempre 30 *pacdots*, ou seja, a pontuação máxima nesses testes é 28 pontos.

As simulações de Monte Carlo utilizadas pelo agente ofensivo do *MonteCarloTeam* utilizaram profundidade 6, e foram feitas 30 simulações para avaliar cada ação. Essa escolha de números deixa o tempo de decisão do agente bastante baixo, oferecendo ainda um bom resultado. Aumentando esses parâmetros é possível melhorar o desempenho do agente, mas eles foram mantidos relativamente baixos para agilizar a execução dos diversos testes.

As duas tabelas abaixo mostram os resultados obtidos nos jogos contra o *SimpleTeam*. As tabelas mostram o número de vitórias, derrotas e o *score* médio para o time vermelho, assim como o número de jogos empatados.

Vitórias	18
Derrotas	2
Empates	0
Score médio	8.55

Tabela 1: *MonteCarloTeam* (vermelho) x *SimpleTeam* (azul)

Vitórias	3
Derrotas	17
Empates	0
Score médio	-6.2

Tabela 2: *SimpleTeam* (vermelho) x *MonteCarloTeam* (azul)

Inicialmente é possível perceber que, como esperado, o time implementado independe do território. Os resultados são bastante parecidos para os territórios azul e vermelho.

O *MonteCarloTeam* mostrou um desempenho bastante satisfatório contra o *SimpleTeam* em mapas aleatórios, ganhando aproximadamente 90% dos jogos realizados com um *score* de, em média, 7 pontos.

As tabelas a seguir mostram os resultados obtidos nos jogos contra o *RandomTeam*. Novamente, as tabelas mostram o número de vitórias, derrotas e o *score* médio para o time vermelho.

Vitórias	20
Derrotas	0
Empates	0
Score médio	28

Tabela 3: *MonteCarloTeam* (vermelho) x *RandomTeam* (azul)

Vitórias	0
Derrotas	20
Empates	0
Score médio	-28

Tabela 4: *RandomTeam* (vermelho) x *MonteCarloTeam* (azul)

Contra o *RandomTeam*, o *MonteCarloTeam* venceu 100% das vezes. Além disso, ele venceu sempre com o *score* máximo. Esse resultado não é tão surpreendente, uma vez que que o *RandomTeam* possui um desempenho bastante ruim.

A próxima tabela mostra os resultados obtidos pelo *MonteCarloTeam* jogando contra ele mesmo.

Pode-se perceber que, nesse caso, os jogos foram bastante equilibrados. Cada time ganhou quase o mesmo número de vezes. Além disso, o *score* médio ficou próximo de 1, ou seja, as vitórias foram por poucos pontos, o que indica que os jogos

Vitórias	11
Derrotas	9
Empates	0
Score médio	1.5

Tabela 5: *MonteCarloTeam* (vermelho) x *MonteCarloTeam* (azul)

foram equilibrados. Esses resultados mostram, novamente, que o time implementado independe do território alocado.

Além dos testes em mapas aleatórios, foram realizados testes contra o *SimpleTeam* em todos os outros mapas fornecidos no diretório *layout*. Nesses testes não foram feitas análises de número de vitórias, derrotas e pontuação média, mas o *MonteCarloTeam* ganhou a grande maioria das partidas. No entanto, durante esses testes, foi possível perceber uma fraqueza do time desenvolvido. Em mapas muito simples e pequenos, como o *tinyCapture*, onde cada time tem apenas um agente em um mapa muito pequeno, o time desenvolvido perdeu quase sempre para o *SimpleTeam*. Isso ocorreu pois, nesse caso, a estratégia mais simples e direta do *SimpleTeam* foi mais eficiente: ele acaba comendo todos os *pacdots* mais rapidamente. O agente do *MonteCarloTeam* perde tempo, por exemplo, se esquivando do agente inimigo, e acaba demorando mais para comer os *pacdots*. Dessa forma, para certos tipos de mapas específicos, uma estratégia mais simples e direta é superior à estratégia do agente desenvolvido.

Podemos concluir que o time desenvolvido apresentou um bom desempenho. Contra agentes muito simples, como o *RandomTeam*, ele domina o jogo por completo. Com agentes um pouco mais complexos, mas ainda bastante simples, como o *SimpleTeam*, os resultados obtidos são ainda bastante satisfatórios, com uma alta taxa de vitórias e um *score* médio consideravelmente alto.