INE5430 - Inteligência Artificial

Trabalho T1

Caique Rodrigues Marques
c.r.marques@grad.ufsc.br

Fernando Jorge Mota
contato@fjorgemota.com

20 de setembro de 2016

Especificações Técnicas

- Escrito na linguagem de programação Python
- Testado o funcionamento em Python 2.7 e em Python 3.5
- Para melhor execução, recomenda-se o uso do PyPy
- Para executar o programa, vá para a pasta `gomoku/` e rode o comando: `python program.py`

Estrutura de Dados

A estrutura de dados usada no programa, definida na classe `state.py`, é imutável. Bastante similar com um grafo - mais especificadamente, uma árvore - cada objeto dessa estrutura possui uma série de atributos que são usados durante a execução do programa:

- Uma matriz representando o tabuleiro vazio - preenchido com "+"internamente, para fins de representação;
- Uma string representando o jogador que está jogando naquele estado - inicialmente, pelas regras do jogo, será o jogador "O", que representa a peça vermelha;
- Uma string preparada para armazenar uma mensagem a ser exibida para o usuário. Usada principalmente em casos de erro ou avisos;
- Um objeto da classe `Sequences` representando as sequências formadas;
- Cada movimento, que é um objeto da classe `Move`, é armazenado em `Sequences`;
- Um ponteiro para o estado que gerou originalmente o estado que se está usando.

A partir de um determinado estado, então, essa estrutura cria cópias de si mesma com apenas alguns atributos modificados e um ponteiro para a versão que a criou. Quando é feita uma jogada, por exemplo, um novo estado é criado modificando apenas a matriz e o ponteiro "pai"apontando para o estado no qual foi feito aquela jogada. Da mesma forma, quando é imprimida uma mensagem, é criado um novo estado apenas configurando a nova mensagem e o ponteiro "pai"correspondente.

No final das contas, temos, nessa estrutura, algo bem parecido com uma árvore/grafo: o estado inicial seria a raiz e nenhum nodo filho seria, a principio, gerado sem que houvesse extrema necessidade. Dessa forma, é possível evitar uso desnecessário de memória e, pelo fato de cada objeto ser imutável, temos aqui uma vantagem: cada novo objeto só modifica os atributos modificados naquela operação. Todos os atributos que não são efetivamente modificados são referenciados por ponteiros no novo objeto criado, ajudando a economizar memória visto que a estrutura realiza cópias de todo atributo a cada operação feita.

Para fins de economia de processamento com o algoritmo de busca, o minimax, o estado inicial gera uma jogada mais ao centro e em seus filhos as jogadas continuam mais concentradas no centro do tabuleiro, que é onde acontece grande parte do jogo. Isto serve como uma otimização da busca evitando que os estados desnecessários (jogadas iniciais nas bordas, por exemplo) não sejam geradas de princípio.

Função Heurística

Considerações

Para uma máquina que queira jogar o jogo Gomoku é necessário que ela tenha uma boa estratégia a fim de conseguir garantir a vitória. A principal estratégia a se usar é considerar as inúmeras possibilidades de jogadas em uma determinada partida, no entanto, é inviável para uma máquina conseguir computar as várias jogadas e suas consequências possíveis, considerando o tamanho do tabuleiro e a quantidade de jogadas possíveis.

Uma heurística é determinada para que a máquina consiga melhores estratégias a partir de uma estimativa, que diz ao computador, a partir de um determinado estado, se aquele estado está mais próximo de uma vitória ou de uma derrota para o sistema.

A máquina pode, portanto, trabalhar mais na ofensiva ou mais na defensiva, dependendo de como estiver a sua situação no jogo. Os seguintes aspectos do jogo serão considerados pelo nosso algoritmo para uma boa estratégia de jogada ofensiva:

- O número de peças usadas e, portanto, o quanto o jogo está próximo do fim;
- A quantidade de duplas, triplas, quádruplas ou quíntuplas formadas pelo computador em uma jogada;
- Uma jogada que forma mais sequências contíguas também favorecem a vitória. Por exemplo, quando as peças estão dispostas onde forma um "L", tem duas sequências que é uma na horizontal e uma na vertical, aumentando portanto as chances de vitória.

As seguintes situações são consideradas para uma jogada mais defensiva:

- Impedir o adversário de formar quíntuplas, quádruplas ou triplas;
- Considerar uma jogada em que evite que o adversário consiga formar sequências contíguas de peças, diminuindo as chances dele de conseguir, por exemplo, de formar duas triplas em forma de "L".

Ainda tem as ações que dê vantagens à máquina:

- Colocar as peças onde há mais possibilidades de formar sequências, por exemplo, colocar um peça mais próxima ao centro dá mais chances de sequência do que colocar nas bordas;
- Verificar quantas sequências já foram montadas (duplas, triplas e quádruplas) para uma possível quíntupla;
- Avaliar as sequências já formadas e ver se é possível na jogada em questão fechar uma sequência maior a partir de duas menores, por exemplo, duas duplas separadas apenas por uma casa, ao colocar uma peça no meio da dupla, forma uma quíntupla;
- A máquina também deve avaliar em quantas casas estão separadas duas fileiras de peças de mesma cor;
- O número de jogadas realizadas até o término no jogo.

Dadas todas as situações listadas, é possível atribuir uma determinada prioridade a cada uma, de forma que a máquina saiba que o adversário formando uma tripla é razoavelmente mais prejudicial à sua jogada do que a máquina formar uma dupla. A saída de uma função heurística considerando os aspectos acima com um peso pré-determinado para cada uma define o quanto a máquina está ganhando ou perdendo no jogo.

Implementação

A análise da heurística é feita considerando diversos aspectos para uma boa jogada. Em `heuristic.py` é definido uma série de funções que avaliam possíveis formações de sequências e atribui pesos a cada uma e a função geral `evaluate()`, que recebe um estado e o jogador atual, retorna as conclusões tiradas das jogadas realizadas.

A seguinte função $t(s)$, onde s corresponde a uma sequência, define como é a análise do tabuleiro pela máquina para tirar uma conclusão:

$$t(s) = length(s) \times block(s) \times merge(s)$$

Onde há três funções: `length()` avalia o tamanho da sequência (quanto maior, melhor a pontuação final), `block()` avalia a abertura de uma sequência (mais pontos se estiver aberta em ambas as direções) e `merge()` avalia a junção de sequências (aumenta a pontuação se é possível juntar sequências em ambos os lados de uma sequência).

Depois de conseguir a saída da função $t(s)$ referente a todas as sequências do tabuleiro, tanto para o jogador (sp), quanto para o adversário (sa), a seguinte operação é feita:

$$result = \frac{sp}{nsp} - \frac{sa}{nsa}$$

Onde nsp e nsa correspondem ao número de sequências do jogador e do adversário, respectivamente. A conclusão é definida em $result$.

Função Utilidade

A função de utilidade define uma certeza, que está próximo do final do jogo e ela define qual a melhor estratégia tomar para terminar o jogo da melhor maneira possível. É necessário que uma função de utilidade faça um balanço de tudo o que foi feito para, então, atribuir um peso final para a melhor decisão.

Portanto, a seguinte função descreve a utilidade:

$$utlty = \frac{score}{jgds} \times x$$

Onde $score$ corresponde a uma pontuação dada ao jogador que formou mais sequências que o adversário (maior se fez mais que o adversário) e $jgds$ corresponde ao número de peças que o jogador usou. Tudo isso é multiplicado pelo valor de x que define se aconteceu uma vitória ou uma derrota ou um empate. No caso de vitórias, o valor é positivo ($x = +1$), derrota é negativo ($x = -1$) e em empate é neutro ($x = 0$).

Detecção de sequências

Em cada jogada no Gomoku, representando um estado, uma jogada é um objeto da classe `Move`, nela está a posição x e y da peça no tabuleiro e se foi o jogador `X` ou `O` que a jogou. A cada nova jogada, o objeto `Move` é armazenado no objeto da classe `Sequences`, portanto, ela conterá todas as jogadas realizadas e dela é possível achar as sequências sem a necessidade de percorrer o tabuleiro todo. Ambas as estruturas, `Sequences` e `Move` são imutáveis.

Contendo um objeto imutável com todas as jogadas realizadas, a classe `Sequences.py` apresenta que é possível coletar as sequências de acordo com uma especificação: coletar sequências de um jogador, coletar sequências pelo tamanho, coletar sequências que estão fechadas, etc..

Referente a uma sequência em específico, a classe `Sequence` torna a análise possível. Assim, é viável avaliar as extremidades de uma sequência (se ela possui as extremidades inferior e superior abertas ou fechadas) e a menor distância de uma extremidade da sequência a uma peça adversária.

Baseando-se na distância de uma peça a outra, é possível realizar junções (`merges`) de sequências de peças de mesma cor, por exemplo: dada uma disposição (horizontal, vertical ou diagonal), as duas extremidades da sequência são marcados, ao encontrar uma outra sequência na mesma disposição, suas extremidades também são marcadas. Se a posição de uma extremidade de uma sequência for igual à posição de uma extremidade oposta de uma outra sequência, então existe a possibilidade de junção de sequências; considerando que não haja peças adversárias no meio do caminho entre as duas sequências.

Detecção de vitória

Conforme especificado na seção anterior, é realizado uma busca no objeto `Sequences` para avaliar as sequências de peças e quem as jogou. Assim, bastando procurar por sequências onde o tamanho é maior que cinco, já se detecta que o jogo terminou e um vencedor foi declarado.

Problemas encontrados

Apesar das modificações realizadas, o algoritmo não ficou livre de problemas. O mais notável é a questão da performance, mesmo usando o PyPy, não chega a ter o tempo de resposta desejável para uma inteligência artificial. Relacionado a isso, também se notou ser difícil fazer com que a máquina avance mais níveis da árvore de jogadas, sendo mais viável ficar com uma profundidade relativamente baixa de busca, de apenas 2 níveis.