Universidade Federal de Santa Catarina

Trabalho 1 - Primeira Entrega

INE5430 - Inteligência Artificial

Alunos
Andrei Donati
Ígor Assis Rocha Yamamoto
Luis Felipe Pelison

Professor
Elder Rizzon Santos

Agosto de 2017

Sumário

1 Problema

Este trabalho tem como objetivo o desenvolvimento de um jogo denominado "5 em Linha", também conhecido como Gomoku. A solução que será desenvolvida deve possuir Inteligência Artificial (I.A.) como competidora com o humano, aplicando o algoritmo MiniMax com podas Alfa-Beta, um estilo de busca adversária. O jogo consiste de um tabuleiro 15x15, onde o objetivo é formar uma sequência de 5 peças, em qualquer direção. A imagem abaixo representa o tabuleiro e um caso de vitória das peças pretas.

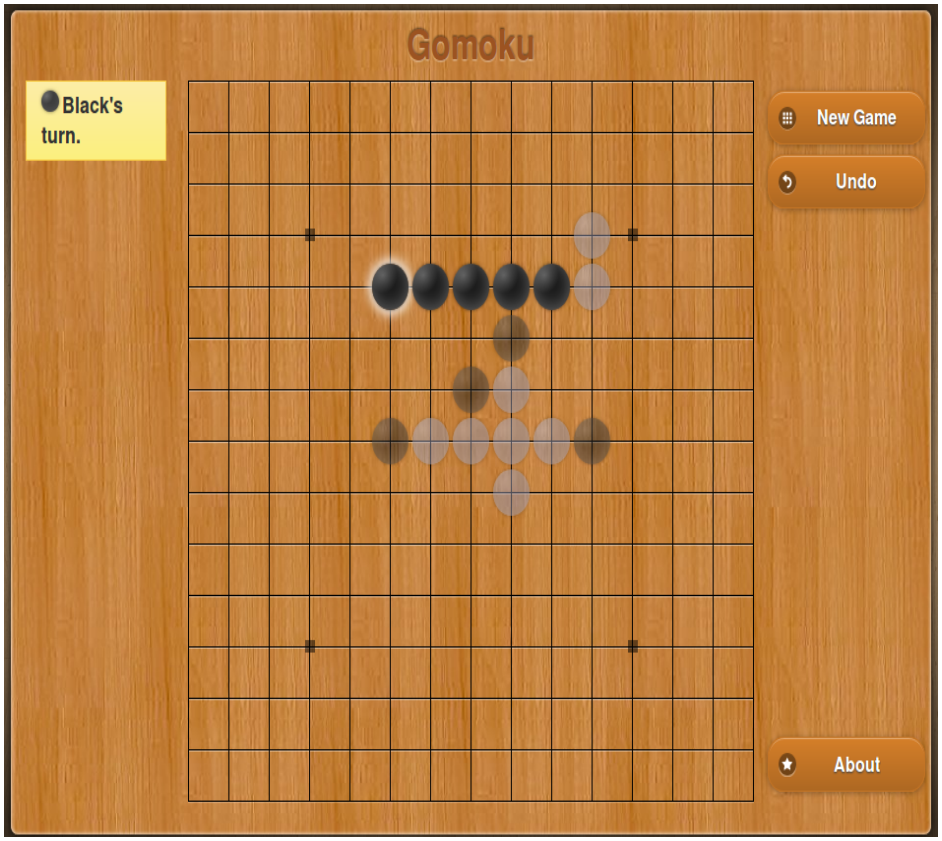

Figura 1: Vitória das pretas. Fonte do jogo: http://gomoku.yjyao.com/

Alguns requisitos do projeto são:

- O desenvolvimento de uma interface gráfica, onde podem jogar humanos e computador (selecionando quem irá iniciar a partida);
- Um estudo sobre possibilidades de cálculo para a utilidade e heurística;

- Explicações sobre a estrutura de dados escolhida, limitações do projeto e os problemas enfrentados;
- Desenvolver e explicar o papel das principais classes e métodos;
- Explicar a técnica utilizada para detecção de fim de jogo e quando ela é aplicada;
- Explicar e desenvolver a detecção de sequencias de 4 peças, pelo menos.

Para essa primeira etapa do projeto, será desenvolvido a matemática e as técnicas de cálculo da utilidade e heurística (Capitulo 2), a estratégia e a estrutura de dados que modela os estados do jogo (Capítulo 3), a detecção de sequencias de peças (Capítulo 4.1), a detecção de fim de jogo (Capítulo 4.2) e o código-fonte utilizado (Capítulo 5).

2 Definição matemática da utilidade e heurística

Na Inteligência Artificial, algumas técnicas são utilizadas em conjunto para a realização da busca de estados. O cálculo de funções como utilidade e heurística são essenciais para o funcionamento do sistema de I.A., já que são elas que guiam o algoritmo em busca da melhor jogada, e consequentemente da vitória. Para a resolução do problema proposto, será usado o algoritmo MiniMax com poda Alfa-Beta. Esse é algoritmo que requer a busca em profundidade. Para tal, inicialmente foi realizado um trabalho experimental a fim de decidir alguns parâmetros gerais para a heurística. Após algumas conclusões, segue as principais que serão aplicadas neste trabalho:

- Priorizar jogadas na região central do tabuleiro;
- Levar em consideração o número de peças que compõem uma sequência;
- Diferenciar sequências que têm aberturas maiores (não estão trancadas por peças adversárias ou fim do tabuleiro);
- Considerar sequências que não podem chegar a 5 peças, pois estão próximos do final do tabuleiro, por exemplo;
- Considerar o aumento do número de profundidade como algo ruim.

Com base nisso, juntamos as ideias inicias e colocamos em um formato matemático, para depois implementar no algoritmo. Para isso, dividimos em conjuntos as sequências possíveis e suas aberturas. Por exemplo, um conjunto é:

- "Quartet_2Opens- Significa que é uma sequência de 4 peças, com 2 aberturas possíveis, ou seja, um vazio de cada lado da sequência.

Após isso, ordenamos todos os conjuntos, de forma que o mais prioritário ficasse em primeiro. A fim de colocarmos valores matemáticos (pesos) que irão multiplicar pela quantidade de vezes que aquela sequência aparece. Assim, poderemos montar uma função matemática que recebe o estado e calcula sua função heurística ou utilidade.

Assim, todos os conjuntos e seus valores numéricos ficaram dessa forma:

- "Quintet": 200000
- "Quartet_2Opens": 120000
- "Quartet_1Open": 50000

- "Triplet_2Opens": 30000
- "Triplet_1Open": 15000
- "ProbQuartet_2Opens": 7000
- "ProbQuartet_1Open": 3000
- "Double_2Opens: 1000
- "Double_1Open": 400
- "ProbTriplet_2Opens": 100
- "ProbTriplet_1Open": 40

Onde 'Quintet' corresponde a uma sequência de 5 peças, 'Quartet': 4 peças, 'Triplet': 3 peças, 'Double': 2 peças e 'Prob' corresponde a uma sequência com um vazio no meio, por exemplo, 'ProbQuartet' é uma sequência de 2 peças + um vazio + 1 peça. Todas essas peças sendo de um jogador só. Os valores foram decididos de forma que as melhores jogadas tenham maior valor, em qualquer circunstância, para que o algoritmo sempre escolha pela melhor/maior.

Com isso realizado, também definimos uma matriz "máscara"que dará maiores pesos para jogadas na região central do tabuleiro. Dessa forma, a heurística tenderá o algoritmo por jogadas nessa região. Assim, a matriz que será multiplicada, elemento a elemento, pela matriz de estado, será a seguinte matriz M:

```
[[0, 0, 0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0, 0, 0],
[0, 0, 0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0, 0, 0],
[0, 0, 0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0, 0, 0],
[0, 0, 0, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0, 0, 0],
[0, 0, 0, 0.2, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.2, 0, 0, 0],
[0, 0, 0, 0.2, 0.4, 0.6, 0.6, 0.6, 0.6, 0.6, 0.4, 0.2, 0, 0, 0],
[0, 0, 0, 0.2, 0.4, 0.6, 0.8, 0.8, 0.8, 0.6, 0.4, 0.2, 0, 0, 0],
[0, 0, 0, 0.2, 0.4, 0.6, 0.8, 1.0, 0.8, 0.6, 0.4, 0.2, 0, 0, 0],
[0, 0, 0, 0.2, 0.4, 0.6, 0.8, 0.8, 0.8, 0.6, 0.4, 0.2, 0, 0, 0],
[0, 0, 0, 0.2, 0.4, 0.6, 0.6, 0.6, 0.6, 0.6, 0.4, 0.2, 0, 0, 0],
[0, 0, 0, 0.2, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.2, 0, 0, 0],
[0, 0, 0, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0, 0, 0],
[0, 0, 0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0, 0, 0],
[0, 0, 0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0, 0, 0],
[0, 0, 0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0, 0, 0]]
```

Juntando tudo isso, a nossa função heurística final ficou da seguinte maneira:

$Heuristic = (NumberOfQuintet*200000+NumberOfQuartet\_2Opens*120000+NumberOfQuartet\_1Open*50000+NumberOfTriplet\_2Opens" *30000+NumberOfTriplet\_1Open*15000+NumberOfProbQuartet\_2Opens*7000+NumberOfProbQuartet\_1Open*3000+NumberOfDouble\_2Opens*1000+NumberOfDouble\_1Open*400+NumberOfProbTriplet\_2Opens*100+NumberOfProbTriplet\_1Open*40)+SumElements(StateMatrix.*M)$

Onde StateMatrix representa a entrada da função, sendo a matriz do estado. O operador ".*"representa a multiplicação elemento a elemento, e não a multiplicação tradicional de matrizes. E, por fim, o "SumElements"representa o sematório de todos os elementos da matriz resultado.

Além da função heurística, também temos a função utilidade. Essa, será aplicada em estados finais, para avaliar a pontuação de um nó folha. A diferença dessa função para a heurística é que ela deve resultar um valor maior e também não terá a matriz de pesos para a região central. Assim, a função utilidade ficará da seguinte forma:

$Utility = 2*(NumberOfQuintet*200000+NumberOfQuartet\_2Opens*120000+NumberOfQuartet\_1Open" *50000+NumberOfTriplet\_2Opens" *30000+NumberOfTriplet\_1Open*15000+NumberOfProbQuartet\_2Opens*7000+NumberOfProbQuartet\_1Open*3000+NumberOfDouble\_2Opens*$

$1000 + NumberOfDouble\_1Open * 400 + NumberOfProbTriplet\_2Opens * 100 + NumberOfProbTriplet\_1Open * 40)$

3 Estratégias e Otimizações

O capítulo anterior trouxe como foi pensado e calculado os valores para as funções heurística e utilidade. Assim, ele não deixa de ser uma parte da nossa estratégia para a resolução do problema.

Seguindo, então, com o desmembramento de nossa estratégia, procuraremos deixar mais claro nessa parte do documento como será realizado a implementação dos estados do tabuleiro na linguagem de programação, o papel das principais classes e métodos que pensamos em utilizar, algumas limitações e problemas da ideia geral e também possíveis otimizações que pensamos em utilizar para a segunda parte, ou não.

Como primeira estratégia, nessa primeira etapa do projeto, tomamos algumas decisões. Dentre elas, partimos para o desenvolvimento do projeto na linguagem de programação mais familiar para os integrantes do grupo, Python. Com isso em mãos, realizamos algumas pesquisas sobre como esse tipo de abordagem de inteligência artificial em jogos é utilizada por outras pessoas no mundo, seja na academia ou como hobbie. Achamos dois bons artigos que falavam sobre ([1] e [2]), além de muitas pessoas interessadas no desenvolvimento do mesmo jogo na internet. Isso nos deu alguma experiência para tomar outras decisões.

Decidimos escolher uma abordagem de orientação a objetos, portanto as principais classes iremos relatar agora. O código-fonte do que foi gerado até aqui pode ser observado em anexo.

Para os estados do jogo/tabuleiro, implementamos um array de 2 dimensões, uma matriz. Essa matriz é 17x17, pois inserimos as bordas sendo o caracter '2' nas linhas 0 e 16 e colunas 0 e 16. Além disso, a parte central, 15x15, começa zerada, ou seja, com todos os elementos iguais a '0'. O '0' nos indica que não há peças ali, é vazio. Depois, será inserida peça por peça, sendo a peça do jogador 1 igual a '1' e a peça do jogador 2 igual a '-1'. O método responsável por essa parte é chamado de player_move().

Para a interface gráfica, optamos por algo básico. Será um 'plot' em um gráfico da biblioteca matplotlib, do Python. Nele, ambas as opções de jogo poderão ser realizadas, humano conta humano e humano contra máquina.

Agora, partindo para as otimizações que poderão ser realizadas, nós pensamos em algumas possibilidades. Porém, elas ficarão para a segunda etapa do projeto, se forem viáveis e úteis.

Dentre elas, podemos citar algumas:

- Fazer a busca por sequências apenas em um raio de 4 peças distantes das peças presentes no tabuleiro, em vez de pesquisar no tabuleiro inteiro.
- Implementar uma fila de filhos prioritários no momento de geração dos filhos, na parte do algoritmo MiniMax.
- Depois de encontrar uma sequência de peças, saltar algumas casas, a fim de procurar por outro padrão não sobreposto ao anterior.
- Começar a analisar em uma profundidade maior de jogadas a partir da oitava peça jogada.

Para essa primeira etapa, as estratégias foram as representadas acima. Além dessas, nós citaremos duas principais no próximo capítulo (4). Outros requisitos do projeto serão deixados para a parte dois do projeto, como o desenvolvimento do algoritmo, melhorias na interface, com a disponibilização das iterações que o algoritmo chegou e os modos de jogo possíveis.

4 Detecção de sequência de peças

Este capítulo será dedicado a tratar de como foi feita a detecção de sequências de peças de um mesmo jogador no tabuleiro. Ele é de extrema importância visto que toda a heurística e função utilidade se baseará nas sequências de peças formadas no tabuleiro. O algoritmo de detecção de sequência é muito similar, independente do número de peças em sequência que busca ser detectado e consiste em:

- Produzir vetores de todas as linhas do tabuleiro
- Produzir vetores de todas as colunas do tabuleiro
- Produzir vetores de todas as diagonais maiores que 4 do tabuleiro
- Substituir os valores dos vetores para que eles tenham "e"em um espaço "empty"(vazio, possível de receber uma peça), "x"no espaço com uma peça do jogador e "n"no espaço "not possible"(não pode receber uma peça, pois é borda ou uma peça do adversário)
- Verificar se, em algum dos vetores produzidos, existe alguma sequência pré-definida de peças

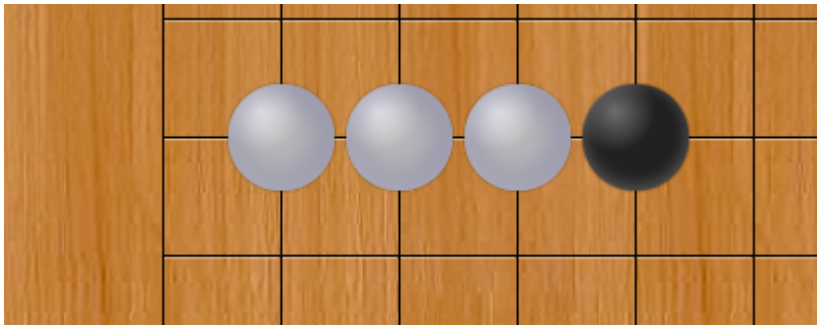

Figura 2: Situação onde a sequência de peças deve ter heurística 0.

As sequências pré-definidas citadas no último passo do algoritmo são sequências de peças que são associadas a valores determinados de heuristica. Um exemplo de sequência é quatro peças de um mesmo jogador com ambas as pontas livres; outro exemplo é uma sequência também de quatro peças de um mesmo jogador mas agora com apenas um dos lados livres.

Mesmo sendo um capítulo voltado para detecção de sequência, é importante destacar que como uma otimização foi adicionado na heurística, toda a sequência de peças deve observar, no mínimo, 5 posições no tabuleiro, mesmo se, por exemplo, a busca é para uma sequência de apenas três peças. Isso deve acontecer pois caso encontremos uma situação como a da imagem abaixo, que apesar da próxima jogada formar uma quádrupla, nunca será possível ganhar o jogo. Assim, a heurística desta sequência deve ser 0:

4.1 Detecção de sequências de quatro, três e duas peças

A detecção de todas essas sequências se dá de uma maneira bastante parecida. Após todos os vetores de linhas, colunas e diagonais serem produzidas, uma string representando o padrão da sequência de peças buscada, que está armazenada em um dicionário de dados, será procurada e contada. Para o caso de duas peças em sequência, temos quatro possibilidades de padrões, considerando a distinção entre peças consecutivas com e sem ambos os lados livres: ['eexxe'], ['exxee'], ['nxxeee'] e ['eeexxn'], lembrando que "e"representa um espaço "empety"(possível de receber uma peça), "x"representa um espaço com uma peça do jogador e "n"um espaço "not possible"(não pode receber uma peça). Assim como na busca por uma sequência de duas peças, a busca por uma sequência de três peças está focada em encontrar o padrão de strings: ['exxxe'], ['nxxxee'] e ['eexxxn'] e para os quartetos os padrões buscados são: ['nxxxxe'], ['exxxxn'] e ['exxxxe'] Há também outras sequências buscadas, como a de um provável trio (duas peças em sequência seguido de uma aber-

tura e mais uma peça) e um provável quarteto (três peças em sequência, seguido de um espaço vazio e mais uma peça).

É importante destacar os motivos da decisão desta implementação de busca por strings. O primeiro motivo é a rapidez de busca, visto que a linguagem oferece bibliotecas que otimizam o trabalho deixando o sistema como um todo mais rápido. Em segundo lugar foi a facilidade de implementação, também por existir bibliotecas que já apresentam funções prontas.

Maiores detalhes da implementação estão dentro do código heuristic.py, em anexo.

4.2 Detecção de sequências de cinco peças e fim de jogo

A detecção de fim de jogo se dá de duas maneiras: detecção de cinco peças ou falta de lugares para jogar. A primeira opção se dá de uma maneira muito similar a detecção de sequência de 4 ou 3 peças, porém neste caso a string a ser buscada é ['xxxxx']. A segunda opção se dá por uma leitura completa no tabuleiro a cada jogada. Como esta não é a maneira mais otimizada, outras opções serão avaliadas para a próxima entrega.

5 Código-fonte

O código fonte está em anexo ao envio do relatório e também pode ser encontrado no repositório https://github.com/igoryamamoto/ine5430-gomoku

Referências

[1] Kulikov J. Self-teaching Gomoku player using composite patterns with adaptive scores and the implemented playing framework. Tallinn University Of Technology.

[2] L.V. Allis, H.J. van den Herik, M.P.H. Huntjens. Go-Moku and Threat-Space Search. University of Limburg and Vrije Universiteit.