IA - Relatório Final do Trabalho sobre Métodos de Busca - Gomoku

- Aluno: Quenio Cesar Machado dos Santos
- Matrícula: 14100868
- Semestre: 2015-2

Objetivo

Implementação do jogo Gomoku usando:

- o algoritmo de busca MinMax com podas;
- uma estrutura de dados e de busca adequada;
- uma heurística/utilidade compatível com o MinMax (implementada matematicamente);
- algumas otimizações (as mesmas não podem substituir o MinMax).

Papel das Classes

O jogo foi implementado em C++ usando classes. A classe principal, `Game`, controla a execução do jogo alternando as jogadas entre os jogadores. Ela também define o nível de dificuldade do jogo, ou seja, a profundidade da busca MinMax.

O estado do jogo é mantido pela classe `GameBoard`. Cada momento do jogo é representado por sua própria instância de `GameBoard`. Esta classe provê uma série de métodos que permitem aos jogadores marcar uma posição no tabuleiro e também consultar qual é o estado atual do tabuleiro, tais como, quais posições ainda estão disponíveis para serem jogadas. A classe `GameSlot` mantém o estado de cada posição no tabuleiro. Juntamente com as classes `GamePosition` - que representa as coordenadas de uma posição no tabuleiro - e `GameArea` - que delimita uma área do tabuleiro, `GameSlot` auxilia GameBoard nas sua funções.

Os jogadores são representados pela classe abstrata `Player`, que possui duas sub-

classes concretas: `AIPlayer` - que representa o computador jogando - e `HumanPlayer` - que presenta o usuário do jogo. `HumanPlayer` apenas provê um "prompt" para que o usuário possa informar a posição da sua próxima jogada. `AIPlayer` depende da classe `GameTree` para determinar a melhor jogada do computador.

A classe `GameTree` implementa o algoritmo MinMax, determinando assim a melhor jogada para o computador. Ela depende da classe `GameNode` para descobrir quais são as próximas jogadas possíveis e também para saber a pontuação de um determinado estado do tabuleiro. `GameNode` implementa as funções de utilidade e de heurística, utilizando-se da classes `GameBoard` e `GamePosition` para avaliar o estado do tabuleiro.

Além das classes acima, as seguintes tipos de enumeração tem papéis importantes na implementação:

- `PlayerMaker` tem dois valores possíveis: `X` - que é o marcador de posição do tabuleiro usado pelo computador; e `O` - que é o marcador usado pelo usuário do jogo. Estes marcadores são usados para determinar se uma posição marcada ou jogada é do computador ou do usuário.
- `Direction` tem como valores os pontos cardeiais e permite definir a direção em que se percorre as posições do tabuleiro. É usada na função de heurística e para determinar uma sequência de vitória.
- `PlayerSkill` guarda implicitamente a profundidade da busca MinMax, que corresponde à habilidade do computador.

O tipo `Score` - que é simplemente um apelido de `long` na imlementação atual - guarda as pontuações calculadas pelas funções de utilidade e heurística. Também é utilizado na implementação do MinMax.

Métodos Principais

As classes descritas acima possuem alguns métodos chave que orientam a execução do jogo:

- `void Game::start()` inicia o jogo e executa o "loop" que alterna entre as jogadas entre os jogadores.
- `GameBoard Player::play(gameBoard)` executa uma jogada de um dos jogadores. É um método polimórfico com implementação nas sub-classes `AIPlayer` e

`HumanPlayer`, de acordo com sua funções. Ele recebe o estado atual do tabuleiro em uma instância de `GameBoard` e retorna o novo estado do jogo em uma nova instância de `GameBoard` que já contém a nova jogada.

- `GamePosition GameTree::bestPositionFor(playerMarker)` é executada por `AIPLayer` para descobrir qual é a melhor jogada baseada no estado atual do tabuleiro. Este método implementa o primeiro nível "max" do algoritmo MinMax, guardando a posição a ser jogada cada vez que encontra um valor máximo.
- `Score GameTree::minMax(gameNode, playerMarker, alpha, beta)` implementa o algoritmo MinMax com podas alpha-beta. Também retorna o `Score` de um nó quando atinge uma folha da árvore do jogo. É chamado pelo método `bestPositionFor()` descrito acima.
- `vector<GameNode> GameNode::childrenFor(playerMarker, focus)` retorna os nós-filho de um nó. Cada instância de `GameNode` contém a posição da jogada feita pelo nó e a instância de `GameBoard` com estado gerado por aquela jogada. É este método que determina a amplitude da árvore de busca.
- `Score GameNode::scoreFor(playerMarker)` é a função que retornará o valor de utilidade do tabuleiro associado ao nó, caso o tabuleiro seja terminal (tenha uma vitória ou um empate); ou o valor da função de heurística, se o tabuleiro associado ao nó é não-terminal.
- `Score GameNode::utilityScore()` retornará o `Score` máximo se o computador for o ganhador e o `Score` mínimo se o ganhador for o usuário; e zero, se for um empate. Também leva em conta o nível do nó, para que vitórias do computador nos níveis superiores sejam mais valiosas que aquelas num nível inferior (para ganhar mais rápido); e derrotas ou empates valham mais nos níveis inferiores (perder ou empatar mais tarde).
- `Score GameNode::heuristicScore(playerMarker)` é a função de heurística. O algortimo de heurística será descrito numa seção abaixo.

Estruturas de Dados

As estruturas de dados usadas na implementação são:

- *Array bi-dimensinonal*: a classe `GameBoard` armazena o estado de um tabuleiro num array bi-dimensional, onde a primeira dimensão representa uma linha do tabuleiro e a segunda dimensão representa uma coluna. Na interseção de uma linha e uma coluna

se encontra uma instância de `GameSlot`, que contém o estado de uma única célula do tabuleiro.
- `vector<GameNode>`: a classe `vector`, da biblioteca padrão do C++, é usada para percorrer a árvore de busca do MinMax. Ela contém uma lista de instâncias de `GameNode`, que representam as jogadas a serem avaliadas a partir do configuração atual do tabuleiro.

Além das estruturas mais complexas descritas acima, existem algumas classes agregam informações importantes do jogo:

- `GameSlot` armazena o estado de cada célula do tabuleiro: marcada com `X`, `O`, ou vazia.
- `GamePosition` armazena a posição de uma célula no tabuleiro, mantendo dois índices, linha e coluna, que tem os valores válidos no intervalo [0, 15].
- `GameArea` armazena as coordenadas que delimitam uma área do tabuleiro. É usada para restringir a amplitude de busca do MinMax.

Função de Heurística

A função de heurística implementada em `GameNode::heuristicScore()` analisa o estado de todas as células do tabuleiro para determinar uma pontuação, ou `Score`. Um valor positivo significa que a configuração apresentada do tabuleiro beneficia o computador, enquanto um valor negativo beneficia o usuário do jogo. O valor zero demonstra que não há vantagem para qualquer jogador naquela configuração. Desta forma, o algoritmo do MinMax poderá aplicar as funções de maximização ou minimização em cada nível da busca para determinar a melhor jogada.

A análise é feita em todas as direções do tabuleiro - usando os pontos cardeiais como referência, procurando por sequências relevantes dos marcadores do computador e do usuário. Para cada sequência relevante é dada uma pontuação e, ao final, o somatório de todas as pontuações é retornado como a pontuação do tabuleiro como um todo.

A pontuação de cada sequência relevante é baseada nas táticas do jogo. Em valores absolutos, quanto maior a pontuação de uma sequência, mais valiosa esta será do ponto de vista tático. As sequências táticas são descritas abaixo da mais valiosa para a menos valiosa:

- *Quatro abertos*, ou seja, quatro marcadores do mesmo tipo em sequência, com as duas pontas da sequência abertas, garante a vitória para o jogador daquele marcador em sua próxima jogada, pois o adversário não poderá bloquear ambas as pontas da sequência, assim permitindo ao jogador completar uma sequência de cinco marcadores e ganhar o jogo.
- *Três abertos* é similar a sequência de *quatro abertos*, porém tem somente três marcadores em sequência. Ela é menos valiosa por isto, mas assim é bem valiosa pois o jogador deste marcador está potencialmente prestes a formar uma sequência de *quatro abertos*. Outra forma de *três abertos* acontece quando existe uma posição vazia entre uma sequência de dois marcadores e outra posição com o mesmo marcador. Esta situação também dá a possibilidade ao jogador de completar um *quatro aberto*.
- Sequências de *quatro* ou *três* semi-fechadas, ou seja, com apenas uma ponta aberta, são menos valiosas quanto as sequências descritas acima, visto que o adversário pode neutralizá-las. Porém, elas ainda podem resultar em vitórias e tem o poder de forçar o adversário a defender-se, ao invés de atacar.
- Sequências de *dois* e marcadores "solitários" são significativamente menos valiosos que as sequências acima.

Para se evitar que o somátorio de um grande número de sequências menos valiosas resulte em uma pontuação maior que um pequeno somatório de sequências mais valiosas, utilizou-se uma função exponencial para definir a pontuação de cada sequência tática. O expoente representa o comprimento da sequência encontrada e a base representa o estado da célula, sendo uma célula marcada mais valiosa que uma célula vazia.

Otimizações

Visto que a busca do MinMax é cara num tabuleiro 15x15, devido ao tamanho da árvore gerada para todos os estados do jogo, foi necessário implementar algumas otimizações que permitiram diminuir a árvore de busca, tornando o tempo de execução com quatro níveis mais razoável.

Podas com Alpha & Beta

O algoritmo MinMax que mantém valores de alpha e beta para cada nó permite que

ramos inteiros da árvore de busca possam ser ignorados, pois estes não poderiam trazer resultados melhores do que já se alcançou com os ramos já pesquisados.

A implementação melhorou substancialmente o desempenho do MinMax com até três níveis de busca, mas não foi suficiente para melhorar o desempenho da busca de quatro níveis.

Foco em Áreas do Tabuleiro

Outra otimização que foi necessária foi diminuir a amplitude da árvore de busca. Para que o número de nós-filho gerados para cada nó-pai fosse significativamente menor, e assim diminuir o tamanho da árvoce de busca, usou-se a classe `GameArea` para definir uma área menor do tabuleiro onde o computador deveria focar em cada jogada.

Ao iniciar o jogo, a área de foco é a área central do tabuleiro, pois ainda não exitem células marcadas. Porém, a classe `AIPlayer` vai continuamente verificando o local da última jogada do adversário e vai ajustando a área de foco para que esteja em volta da última jogada.

Observou-se que, com uma área de foco 6x6, consegue-se um desempenho razoável do MinMax e ainda assim o computador é capaz de "perceber" as jogadas mais relevantes na maior parte do tempo. Porém, com uma área de foco menor, apesar de um desempenho ainda melhor do MinMax, o computador começa a ter um desempenho tático pior, pois deixa de ver muitas jogadas.

Com a implementação desta otimização se conseguiu um desempenho bem melhor com quatro níveis de busca.

Ordenação dos Nós de Busca

Ainda outra otimização realizada foi a ordenação dos nós-filho de tal forma que as jogadas que estão mais perto da última jogada do adversário sejam avaliadas primeiro.

Esta otimização permitiu ao MinMax avaliar as podas a partir das jogadas mais relevantes e assim acelerou o processo de poda.

Problemas Encontrados

Abaixo estão alguns das dificuldades encontradas durante a implementação do Gomoku:

- *Táticas e estratégias*: Antes de implementar a função de heurística foi necessário entender bem como o jogo funciona e pesquisar sobre as táticas e estratégias do jogo. Assim mesmo, foi necessário ainda jogar o jogo inúmeras vezes para entender as diferentes situações, o valor relativo de cada uma, e as combinações possíveis de jogada.
- *Implementação da função de heurística*: Mesmo depois da pesquisa inicial e da melhor compressão das táticas e estratégias do jogo, ainda foi necessário muito tempo elaborando o esquema de pontuação. Na implementação inicial, o algoritmo estava avaliando somente as células em torno da jogada a ser realizada. Esta abordagem funcionou bem apenas com um nível de busca, mas simplesmente não funcionou com dois ou mais níveis. Isto se deu porque a base de referência do tabuleiro variava muito entre as avaliações de cada nó da árvore de busca, dado que cada avaliação se dava num conjunto de células diferentes. Após o algoritmo de heurística ser modificado para avaliar todas as células do tabuleiro (e depois de várias iterações de ajuste do algoritmo), o desempenho começou a melhorar na medida em que o número de níveis da busca foi aumentando.
- *Verificação de casos de teste*: Outra dificuldade da implementação foi a necessidade de repetir várias vezes os casos de teste toda vez que uma alteração era feita na função de heurística. Algumas alterações eram boas para um caso, mas "quebravam" outros casos. Portanto, este trabalho de ajuste exigiu muito tempo. A automação da execução dos casos de teste teria acelerado este processo.
- *Implementação das otimizações*: A implementação das otimizações também exigiu tempo devido a necessidade de comparar o desempenho antes e depois da otimização. Foi necessário também refatorar o código do MinMax para implementar podas alpha-beta, o que exigiu mais tempo para testar as mudanças e certificar-se que o MinMax estava funcionando adequedamente.

Limitações

A grande limitação da implementação é o baixo desempenho na busca de 5 níveis ou mais. Para funcionar adequedamente com 5 níveis é necessário uma área de foco de tamanho mínimo, o que compremete enormemente a habilidade do computador.

Outra limitação é que, devido ao tamanho do foco, às vezes, o computador não enxerga

uma boa jogada fora do foco. Para se resolver este problema, seria preciso melhorar o algoritmo que determina a área de foco, de tal forma que tais jogadas relevantes nas bordas da área de foco aparecam na busca do MinMax.

Conclusões

O algoritmo MinMax funciona adequedamente para o jogo Gomoku, porém seu desempenho deixou a desejar. Mesmo com o uso de podas e outras otimizações, não foi possível atingir um desempenho adequado acima de 4 níveis de procura. Seria interessante pesquisar sobre outras formas de poda - e outras otimizações possíveis - para melhorar o desempenho do MinMax e permitir mais níveis de procura. Talvez, otimizações nos métodos auxiliares e nas estruturas de dados pudessem trazer alguma melhoria no desempenho.

Também vale salientar que a implementação adequedada do algoritmo da heurística é fundamental para o sucesso da implementação. É necessário avaliar todo o estado do tabuleiro para que o MinMax funcione corretamente. Na área da função de heurística seria interessante também pesquisar mais sobre possíveis técnicas para a elaboração de um algortimo adequado. Ao menos, seria útil estabelecer boas práticas para que se possa ser mais eficiente na implementação da função de heurística.

Apesar das dificuldades encontradas, foi muito gratificamente ver o computador jogando bem; até melhor que o usuário em alguns casos; já não era possível ganhar do computador quando este iniciava um jogo. Foi muito interessante observar na prática que o programador pode "ensinar" o computador a jogar e ainda se surpreender com a habilidade superior do computador numa partida contra o programador. Esta experiência gera questões interessantes sobre o impacto da inteligência artificial nos próximos anos à medida em que as habilidades dos programas avançam e abrangem cada vez mais as diferentes áreas de habilidade humana.