Universidade de São Paulo – USP
Instituto de Ciências Matemáticas e de Computação – ICMC
Departamento de Ciências de Computação – SCC

SCC-5774 – Inteligência Artificial I

Professor Gustavo Batista
gbatista@icmc.usp.br

Aluno: Igor Bueno Corrêa
igorbc@usp.br

Projeto – Four in a Row

O objetivo do projeto é criar um programa capaz de jogar o jogo For in a Row (também conhecido como Connect 4, entre outros) usando busca adversária com poda alpha-beta.

Este relatório tem o objetivo de demonstrar a estratégia utilizada na realização do projeto.

Modelagem da grade do jogo

O jogo consiste em uma grade de 6x7 onde as fichas de cada jogador são colocadas alternadamente. A modelagem inicial desta grade no programa é uma matriz 6x7 onde cada posição pode ser um de 3 valores representados pelo tipo `t_player`. `t_player` é um tipo personalizado onde os valores inteiros -1, 0 e 1 correspondem ao jogador min, espaço em branco e jogador *max*, respectivamente. Portanto esta matriz contém os valores onde as posições vazias são 0, as posições do jogador *max* são 1, e do *min* são -1.

Utilizando esta representação como base, foi criada outra variável para representar a grade do jogo. Uma que pudesse ser mais útil no momento de avaliar os estados e verificar se o jogo acabou.

Esta nova representação do estado foi chamada de `four_array` e se trata de um *array* 69x4 de inteiros, que é preenchido de forma similar a matriz 6x7, ou seja, com valores do tipo `t_player`.

Cada uma das 69 posições deste *array* representam os 69 possíveis grupos de "*four in a row*" (quatro fichas seguidas) que estão contidos na grade 6x7. A Tabela 1 mostra o cálculo deste número. O cálculo da função que verifica se o estado é terminal, e da função que avalia o estado, foram feitos usando esta nova representação, o `four_array`, sem lidar com a matriz 6x7.

Heurística de avaliação e verificação de estado terminal

A ideia inicial foi usar o `four_array` para criar uma função de avaliação do estado de forma bem simples: o valor desta função é simplesmente a soma dos 4 valores para cada posição. Ou seja, cada posição do *array* gera uma soma que pode variar de -4 até 4 e a utilidade final do estado é a soma destas 69 somas. Após verificar um péssimo desempenho, a primeira alteração realizada foi elevar ao cubo cada uma destas somas, pois assim o sinal é preservado e acontece uma boa amplificação no impacto de se ter uma soma com módulo maior (se a soma é 1, utilidade é 1, se for 4, utilidade é 64).

Tabela 1 – possíveis grupos de 4 fichas

Localização	Quantidade de grupos de 4	Ocorrências	Total
Linha	4	6	24
Coluna	3	7	21
Diagonal menor	1	4	4
Diagonal média	2	4	8
Diagonal maior	3	4	12
		Total geral:	**69**

Com um desempenho melhor, porém ainda insatisfatório, a próxima ideia foi considerar positivamente para um determinado jogador apenas os grupos de 4 onde este jogador ainda poderia ganhar. Ou seja, se um grupo de 4 contém pelo menos uma ficha de cada jogador, sua utilidade é 0, pois nenhum dos dois jogadores pode vir a vencer a partida através desse grupo. O valor de cada configuração de um grupo de 4 foi alterado da forma mostrada na Tabela 2. Nela é mostrado apenas para o jogador *max*, mas para o *min* o cálculo é o mesmo, apenas com o sinal invertido.

Tabela 2 - heurística de avaliação

Fichas do jogador *max*	Espaços em branco	Valor do grupo de 4
1	3	1
2	2	10
3	1	100
4	0	6900

O último valor, 6900, que consiste em uma vitória, é um valor conceitualmente insuperável (69 vezes o valor de um grupo de 4 com 3 fichas do jogador).

Com esta heurística de cálculo da utilidade de um estado, o desempenho do programa ficou satisfatório.

Para calcular se alguém venceu o jogo, bastou verificar se a quantidade de fichas do jogador *max* ou *min* em um determinado grupo de 4 é igual à 4 ou -4, respectivamente. O caso especial de empate foi verificado através da iteração pela linha mais superior da grade (se tiver algum espaço livre, o jogo não está empatado).

Outras alterações

Diversas alterações foram realizadas no modo com que o código modelo funcionava.

- A função `max_value` e `min_value` foram substituídas por uma única função `minimax`, que recebe também o jogador (tipo `t_player`);
- Como era fundamental para o funcionamento do programa, foi implementada uma função que retorna a coluna onde a ficha deve ser colocada, e não a melhor utilidade encontrada;
- Foi implementado um modo onde, se a avaliação de uma jogada for a mesma avaliação de outra, a jogada escolhida será aquela em que esta avaliação se encontra menos profundamente na arvore de busca de estados. Isso faz com que o programa tente jogar de modo a vencer o quanto antes, mesmo que já esteja confiante que irá vencer mais cedo ou mais tarde. Porém, o efeito adverso disso é que, quando a melhor avaliação é uma derrota, o programa não parece se “esforçar” para prolongar esta derrota, resultando num comportamento aparentemente pouco inteligente;

- A interface com o programa foi estendida de diversas maneiras para facilitar testes. Entre os comandos novos estão:
    - `move max` ou `move min`: calcula a próxima jogada, a efetua e mostra na tela uma representação da grade atual juntamente com informações como utilidade encontrada, número do turno, e se o jogo empatou, ainda continua ou alguém ganhou;
    - `max [coluna]` ou `min [coluna]`: coloca uma ficha na coluna indicada e mostra o estado atual do jogo, possibilitando uma partida contra o programa;
    - `toggle ab`: desabilita ou habilita a poda alpha-beta;

Uma das principais alterações, no entanto, se trata da criação de uma estrutura que representa diversos aspectos de um determinado estado, e não somente o estado da grade.

s_state, representando mais aspectos de um estado

Esta estrutura foi criada para organizar praticamente todas as informações julgadas relevantes sobre um determinado estado. Entre as variáveis que a compõem estão:

- `t_state s`: a matriz 6x7 de t_player representando a grade;
- `int four_utility[69][4]`: o array já discutido, que representa todas as maneiras possíveis de se ganhar o jogo;
- `int utility`: o valor final da avaliação do estado;
- `is_terminal`: pode ser usado como booleano para determinar verificar se o estado é terminal ou não. Porém na verdade os valores usados são:
    - `P1_WINS`: 1
    - `P2_WINS`: -1
    - `ACTIVE_GAME`: 0, ou seja, neste caso `is_terminal` é falso
    - `DRAW`: 2

Com a criação desta estrutura, basta chamar a função `update_state(&my_state)` para depois ter acesso à informações como `my_state.utility` e `my_state.is_terminal`, não havendo assim a necessidade de se implementar funções como `avalia(estado)` e `is_terminal(estado)`. Os sucessores não foram englobados nesta estrutura, portanto a função sucessor foi implementada normalmente.

Resultado

Sendo possível avaliar os próximos 8 estados até o turno 20, e os próximos 11 posteriormente, a atual versão do programa se encontra ranqueada na posição 43 no site theaigames.com. A versão anterior programa estava relativamente estável entre as posições 47 e 42 (Figura 1).

Figura 1 - ranqueamento da versão final

Diversas melhorias ainda poderiam ser realizadas, como a implementação de *busca com profundidade limitada iterativa*, juntamente com uma boa estratégia de aproveitamento do tempo disponível. Também é possível a diminuição de *overhead* no cálculo da função de avaliação dos estados, possibilitando assim uma avaliação mais rápida. Isso, juntamente com a *busca com profundidade limitada iterativa* resultaria em uma busca mais profunda, e consequentemente em melhores resultados. Experimentos com diferentes pesos na avaliação dos estados, em vez do fator de 10 utilizado, talvez também trouxessem melhorias.