Grupo Nº 17 (Alameda)

TÉCNICO LISBOA

Inteligência Artificial

1.º Semestre 2013/2014

Moedas e Fios

Relatório de Projecto

Índice

1 Implementação Tipos e Problema

1.1 Tipos Abstractos de Informação

- **Posicao**

Contém uma linha e coluna que são utilizadas para identificar a posição.

- **Fio**

Contém um inteiro que representa o identificador do fio e duas posições: a posição de origem e a posição de destino.

- **Tabuleiro**

Contém a lista de moedas e fios, o número de linhas e colunas do tabuleiro e *valor-total-moedas,* que é o valor total de todas as moedas no tabuleiro. Este último elemento foi adicionado para aumentar a velocidade em chamadas à função *tabuleiro-total-moedas*.

- **Jogo**

Contém um tabuleiro, os pontos do jogador A e B, informação sobre o próximo jogador a jogar e o histórico das jogadas efectuadas.

- **Problema**

Contém um jogo (*estado-inicial*), uma função que devolve qual o próximo jogador a jogar, uma função que devolve uma lista com as acções que se podem efectuar sobre um jogo, uma função que devolve um novo jogo que é o resultado de aplicar uma jogada a um jogo, uma função que verifica, dado um jogo, se este já acabou, uma função de avaliação que dá um valor aproximado ou exacto (caso o jogo esteja terminado) de quanto é que um dado jogo vale para o jogador, e uma função que devolve o histórico de todas as jogadas realizadas.

1.2 Funções específicas do problema

- **Accoes**

A função *accoes* recebe um jogo e retorna uma lista de acções ordenadas pelo identificador (maior para o menor). Esta função é utilizada nas funções minimax para saber que acções têm de ser aplicadas ao nó pai para gerar os novos nós filhos.

- **Resultado**

A função *resultado* recebe um jogo e o id do fio que o jogador pretende remover e devolve o jogo correspondente a remover esse fio, sem alterar o jogo inicialmente recebido (para tal, cria-se uma cópia

desse jogo e aplica-se-lhe a jogada). Esta função é utilizada no minimax para gerar os nós filhos recebendo uma acção da função *accoes* e retornando o nó filho correspondente.

- **Teste-terminal-p**

Esta função recebe um jogo e um inteiro correspondente à profundidade (que é ignorado, conforme o enunciado do projecto) e retorna *T* ("Verdadeiro") se o jogo tiver acabado e *nil* ("Falso") caso contrário. Faz-se uso da função *jogo-terminado-p*. A função *teste-terminal-p* é utilizada no minimax para verificar se o nó representa um jogo terminado.

- **Utilidade**

Esta função recebe um jogo e um inteiro (que representa o jogador) e retorna a utilidade do jogo para o jogador recebido. Esta utilidade é calculada fazendo a diferença entre a pontuação dos dois jogadores. Esta função é utilizada no minimax para saber a utilidade de um dado nó terminal.

- **CalcTempoNos**

Esta função recebe o problema (que será utilizado no minimax) e um jogador e retorna uma estimativa do tempo que demora cada chamada à função do minimax (tempo que leva a processar cada nó). Este valor é depois utilizado para calcular até que profundidade o algoritmo minimax pode correr. Consideramos esta função necessária devido às diferenças de tempo de processamento entre computadores diferentes.

- **CalcProfundidade**

Esta função recebe o tempo máximo para retornar uma jogada, a ramificação da árvore correspondente ao problema e o tempo médio que leva a processar cada nó e retorna o nível de profundidade a que o algoritmo minimax pode ir de modo a não ultrapassar o tempo máximo que lhe é dado. A fórmula matemática que calcula este nível de profundidade é baseada na fórmula para árvores *n-árias* perfeitas[1].

[1] http://en.wikipedia.org/wiki/K-ary_tree#Properties_of_k-ary_trees

2 Algoritmos Minimax, Jogador e Variantes

2.1 Minimax com múltiplas jogadas por jogador

O algoritmo minimax do livro era composto por três funções: uma inicial, que depois chamava a função *max*, e esta por sua vez chamava a função *min*, indo alternando até chegarem a um nó terminal.

Na sua implementação em CLISP, foi efectuada a seguinte alteração para permitir múltiplas jogadas por jogador: antes de chamarmos a função *max* ou *min* verificamos se uma moeda foi retirada do tabuleiro (significando que o jogador tem de voltar a jogar). Se isto se verificar, a função chama-se a ela própria. Caso contrário, chama *min* (no caso da função ser *max*) ou *max* (no caso da função ser *min*).

2.2 Mecanismo de limitação de tempo no jogador automático

Para limitar o tempo do jogador automático calculámos o tempo médio que um nó leva a ser gerado e processado através da função *CalcTempoNos*. Para calcular este tempo a função corre o algoritmo minimax com o nível máximo de profundidade igual a 1 (fazendo assim uma rápida iteração sobre o algoritmo minimax) obtendo como resultado um tempo. Depois, com este tempo, a função *CalcProfundidade* é chamada e retorna a profundidade máxima para a qual o algoritmo minimax tem tempo para retornar uma jogada.[2]

2.3 Variante do algoritmo minimax

A variante do algoritmo minimax que utilizámos foi uma versão do minimax com cortes alfa e beta. Esta técnica utiliza duas variáveis extra ao valor, os nós para tentar calcular que nós pode cortar (para assim diminuir o tempo que leva ao algoritmo acabar, ou no caso de ter limite de tempo poder analisar mais nós para assim retornar uma melhor resposta).

[2] Mais informação sobre as funções *CalcTempoNos* e *CalcProfundidade*, incluindo o que motivou a sua implementação, pode ser encontrada na secção 1.2.

3 Funções Avaliação/Heurísticas

3.1 Heurística 1

3.1.1 Motivação

A primeira heurística que utilizámos foi a função *utilidade* da primeira parte do projecto. Decidimos utilizar esta heurística pois, devido a já estar implementada, possibilitou-nos focar na implementação do algoritmo minimax. Para o seu cálculo basta utilizar os pontos dos jogadores.

3.1.2 Forma de Cálculo

Dependendo do jogador a jogar:

Jogador 1: (*pontos jogador1*) – (*pontos jogador2*)

Jogador 2: (*pontos jogador2*) – (*pontos jogador1*)

3.2 Heurística 2

3.2.1 Motivação

A segunda heurística que utilizámos foi uma variante da primeira. Para a sua criação perguntamo-nos o que pensaríamos se nos questionassem que valor daríamos a um tabuleiro num dado instante. Esta heurística é muito parecida à anterior visto que utiliza os pontos de cada jogador, adicionando o valor de todas as moedas que só tenham um fio ligado a elas, uma vez que estas moedas estão "à mercê" do jogador a jogar naquele momento.

3.2.2 Forma de Cálculo

Dependendo do jogador a jogar:

Jogador 1: (*pontos jogador1*) – (*pontos jogador2*) + (valor das moedas com apenas um fio ligado a elas)

Jogador 2: (*pontos jogador2*) – (*pontos jogador1*) + (valor das moedas com apenas um fio ligado a elas)

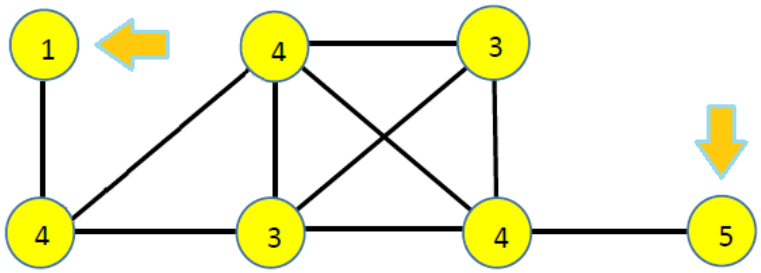

Figura 1 – *neste caso, as moedas de valor '1' e '5' seriam contadas para a avaliação do jogador a jogar.*

4 Estudo Comparativo

4.1 Estudo das variantes do algoritmo minimax/jogador

4.1.1 Critérios a analisar

O critério que utilizámos foi o número de nós visitados. Utilizámos este critério devido à enorme importância de conseguir reduzir este número, tal é o seu impacto na eficiência do algoritmo.

4.1.2 Testes Efectuados

O teste que efectuámos foi correr o algoritmo minimax com e sem cortes em vários tabuleiros para descortinar a redução de visitas a nós obtida.

4.1.3 Resultados Obtidos

A tabela indica o número de nós visitados por teste:

	Sem Cortes	**Com Cortes**
Tabuleiro 1	40 320	1 219
Tabuleiro 2	479 001 600	218 517
Tabuleiro 3	39 916 800	46 307
Tabuleiro 7	39 916 800	34 032

Tabela 1 – *resultados dos testes corridos para quatro tabuleiros diferentes.*

Os testes foram realizados apenas com estes tabuleiros devido ao elevado grau de complexidade dos restantes (ainda assim, o tabuleiro 2 demorou cerca de seis horas a correr).

4.1.4 Comparação dos Resultados Obtidos

Ao comparar os resultados verificámos que obtemos uma redução de mais de 90% nos nós folha visitados. A razão para tal redução é o facto de muitos nós poderem ser cortados de acordo com os seus valores alfa e beta, permitindo assim atingir maiores profundidades e proporcionar um resultado melhor no mesmo tempo útil.

4.2 Estudo das funções de avaliação/heurísticas

4.2.1 Critérios a analisar

Os critérios usados foram, em primeiro lugar, a capacidade das funções levarem à vitória no jogo, e, em segundo, a diferença de pontos entre os jogadores.

4.2.2 Testes Efectuados

Os testes foram efectuados usando sempre a mesma variante do algoritmo minimax/jogador. Os testes consistiram em correr jogos em todos os tabuleiros com jogadores diferentes como argumento e com um jogador humano constituído pelos elementos do grupo (permitindo-nos avaliar passo-a-passo o comportamento do jogador para diferentes situações de jogo). Os testes com o jogador humano acabaram por apelar apenas à nossa curiosidade, uma vez que a escolha do melhor jogador automático tornou-se óbvia após os confrontos entre os dois jogadores implementados.

4.2.3 Resultados Obtidos

Em todos os jogos jogados, o jogador que viria a ser o nosso “vbest” ganhou os jogos (tornando a comparação de pontos redundante).

4.2.4 Comparação dos Resultados Obtidos

O facto de o jogador vencedor usar cortes alfa e beta não deu quaisquer hipóteses ao jogador simples. Os cortes permitem que, no mesmo tempo útil, seja obtida uma jogada melhor que numa solução sem cortes.

4.3 Escolha do jogador-minimax-vbest

Para o jogador-minimax-vbest escolhemos o algoritmo minimax com cortes alfa beta e a segunda heurística (secção 3.2) visto que estas permitem que uma maior profundidade seja atingida e, nas situações em que não é possível atingir um nó folha, que uma acção mais apropriada seja retornada.