Relatório do Projeto JogadorTux

Diego Ferreira Ucha
diegoucha@gmail.com

Fábio Montefuscolo
fabio.montefuscolo@gmail.com

Projeto envolvendo a disciplina

Laboratório de Programação IV

Fabrício Barth

Dezembro/2007

1. Introdução e método

Este projeto teve como objetivo criar um jogador inteligente para o jogo *Connect 4*, que será utilizado na competição final contra o jogador criado por cada grupo, e para isso, foi proposta em sala de aula a utilização do algoritmo *Min-Max*, que decidirá em que posição a próxima jogada deveria ser executada, levando em consideração a função objetivo adotada.

Por se tratar de uma competição, o nosso objetivo principal é vencer do maior número de jogadores, que provavelmente teriam adotado o algoritmo *Min-Max*, explicado em sala de aula. Sabendo que dois algoritmos que implementaram *Min-Max* provavelmente resultariam em empate, iniciamos uma pesquisa para encontrar outras formas mais eficazes para resolver o jogo.

A pesquisa final foi feita na tese de mestrado de Victor Allis[1], no livro Inteligência Artificial de Russel e Norvig, na Wikipedia, na página pessoal de John Tromp[2] (onde, inspirado na tese de Allis, fez uma base de dados dos estados do tabuleiro onde já foram efetuadas 8 jogadas, indicando se há pelo menos uma situação de vitória, empate ou derrota, levando em consideração que o oponente fará jogadas "inteligentes"), e no relatório[3] de estudantes da universidade Northeastern do EUA.

A solução resultante foi uma junção da tese de mestrado de Victor Allis (primeira etapa), do proposto na página pessoal de John Tromp (segunda etapa) e do livro de Russel e Norvig e do relatório dos estudantes da universidade de Northeastern (terceira etapa). É importante mencionar que todas as técnicas passaram por mudanças, que o grupo achou importante para tornar o processo mais eficaz.

2. Solução

[1] http://www.farfarfar.com/games/connect_four/connect4.pdf, último acesso em 08/12/2007 às 19h.
[2] http://homepages.cwi.nl/~tromp/c4/c4.html, último acesso em 08/12/2007 às 19h10m.
[3] www.ccs.neu.edu/home/eclip5e/classes/csu520/Smart**ConnectFour**-Final_Paper-v1.0.doc, último acesso em 08/12/2007 às 19h20m.

A solução adotada pode ser dividida em três etapas, que ocorrem seqüencialmente ao longo do jogo, a primeira envolve as três primeiras jogadas executadas pelo JogadorTux, a segunda etapa corresponde à quarta jogada, e a última da quinta jogada em diante, porém cada etapa depende se o JogadorTux é a peça branca ou preta, como será detalhado abaixo.

2.1 **Primeira etapa – Jogada decidida no código**

Se o JogadorTux é o branco, ou seja, é quem começa a partida, as primeiras 3 jogadas serão no meio, dessa forma conseguimos criar uma barreira no meio do tabuleiro, onde evitamos que o adversário possa ligar 4 peças na horizontal e na diagonal, como mostrado abaixo:

```
7 . . . . . . .
6 . . . . . . .
5 . . . . . . .
4 . . . . . . .
3 . . . x . . .
2 . . . x . . .
1 o o o x . . .
  a b c d e f g
```
[4]

Considerando que as jogadas *x* foram realizadas pelo JogadorTux e *o* pelo oponente, podemos identificar que b1 e c1 só servirão para ligar 4 na vertical, e a1 na diagonal e na vertical, ou seja, conseguimos reduzir o potencial de ameaça das peças do oponente de 9 para 4 (considerando que, teoricamente, cada peça pode ter um potencial individual de 3, um para fazer parte de uma ligação de 4 na vertical, outro para horizontal e o último para diagonal).

Pode ocorrer do oponente também só jogar no meio:

```
7 . . . . . . .
6 . . . o . . .
5 . . . x . . .
4 . . . o . . .
3 . . . x . . .
2 . . . o . . .
1 . . . x . . .
  a b c d e f g
```

Desta forma, ignorando o fato de que a base do tabuleiro está livre e que o oponente potencialmente poderá sofrer uma ameaça dupla, ou seja se c1 e e1 fossem tomados pelo

[4] Representação retirada de ftp://ftp.ics.uci.edu/pub/machine-learning-databases/connect-4/connect-4.names, último acesso em 08/12/2007 às 20h.

JogadorTux, o nível de chance de vitória será quase o mesmo, se não igual para ambos os jogadores, já que o potencial de cada peça está sendo aproveitado para uma ligação na horizontal e na diagonal.

Caso o JogadorTux seja o preto, e considerando que o oponente fez a melhor jogada possível inicial, ou seja jogar no meio, o JogadorTux executa a jogada no e1, para evitar que a base seja uma possível ameaça dupla no futuro, depois as outras duas jogadas são executadas no meio para garantir um maior aproveitamento das próprias peças e, possivelmente, reduzir o das peças do jogador oponente.

2.2 **Segunda etapa – Jogada decidida na base de dados**

Para a quarta jogada, foi utilizada uma base de dados que John Tromp criou com todos os estados possíveis (exceto onde um dos jogadores está por ganhar, com 3 peças conectadas, ou já ganhou) para o tabuleiro, depois de executadas 8 jogadas.

A base de dados é um arquivo texto com 67557 estados do tabuleiro, no seguinte formato:

```
b,b,b,b,b,b,b,b,b,b,b,b,x,o,b,b,b,b,x,o,x,o,x,o,b,b,b,b,b,b,b,b,b,b,b,b
,b,b,b,b,b,win
```

O exemplo anterior representa uma linha do arquivo, que é um estado possível do tabuleiro, onde *b* significa um espaço sem peça, *x* a peça do jogador branco e *o* a peça do jogador preto, e no final é indicado se há pelo menos uma condição de vitória, empate ou derrota para o jogador *x*, neste estado.

A partir da representação:

```
6 . . . . . . .
5 . . . . . . .
4 . . . . . . .
3 . . . . . . .
2 o . . . . . .
1 x . . . . . .
  a b c d e f g
```

A ordem das peças seria representada da seguinte forma na base de dados:

```
Primeiro valor= a1: x
Segundo valor= a2: o
```

```
Terceiro valor= a3: b
Quarto valor= a4: b
Quinto valor= a5: b
Sexto valor= a6: b
Sétimo valor= b1: b
Oitavo valor= b2: b
...
```

Com essa base de dados, tivemos um desafio que foi de carregá-la rapidamente para a memória para não sermos penalizados na competição, a primeira solução mais simples e intuitiva seria de ler o arquivo da forma que foi fornecido por John Tromp, porém este processo provou ser demorado depois de testado. Desta forma, criamos um outro arquivo com a base de dados serializada pela classe Serialize[5], que mostrou ser muito eficaz já que o carregamento deste arquivo ocorre em menos de 1 segundo.

Quando o gerenciador do jogo pede ao JogadorTux a sua quarta jogada, o jogador transforma a representação do tabuleiro, definida pelo gerenciador, no formato da base de dados, executa um algoritmo do Java de match de String e obtém o último parâmetro da linha para saber se a próxima jogada hipotética da iteração será de vitória, derrota ou empate, caso o JogadorTux seja a peça preta.

Se o JogadorTux for a branca, na sua quarta jogada ele deve fazer uma iteração nas suas 7 possíveis jogadas, e para cada jogada hipotética, deve ser feita outra iteração de 7 possíveis jogadas do oponente, assim somando 8 jogadas executadas, que agora para as 49 possibilidades, serão procuradas na base de dados, atribuindo peso de +2 se a base de dados indica que Tux vence, +1 se empata e -2 se perde (os mesmos pesos são usados caso JogadorTux seja a peça preta).

É importante ressaltar que, se o JogadorTux começa o jogo, o último parâmetro que é buscado como positivo são “win” e “draw”, com diferentes pesos, enquanto os parâmetros positivos caso o oponente comece são “loses” e “draw”, lembrando que este parâmetro indica se há pelo menos a condição especificada (vitória, derrota ou empate) para o jogador que inicia.

2.3 **Terceira etapa – Jogada decidida pelo Min-Max**

A partir da quinta jogada, foi utilizado o algoritmo Min-Max para decidir qual será o próximo movimento.

[5] http://en.wikipedia.org/wiki/Serialization#Java, último acesso em 08/12/2007 às 21h.

A profundidade adotada foi 7 com uma média de processamento de 2~3 segundos, a pesar de no código estar representado como 5, o 0 também entra na contagem, sendo 6 até agora, porém a cada recursão da função do Min-Max, o algoritmo sempre prevê as próximas jogadas, ao invés de fazer a contagem de pontuação do estado no qual está situado, desta forma a profundidade se torna 7.

A função de utilidade adotada foi considerar +1 caso o Tux ganhe, -1 se perdeu e 0 se ninguém vence. Porém, foi dado um peso maior para estes valores caso o estado hipotético não esteja muito distante do estado real (até o terceiro nível de profundidade), pois em um teste manual, ou seja, humano contra o JogadorTux, detectamos que o JogadorTux executou uma jogada que o prejudicou, comprometendo sua vitória.

É importante lembrar que, desta forma, apenas as folhas da árvore de jogadas hipotéticas serão pontuadas, os seus pais serão o somatório de seus filhos, onde no final existirão 7 jogadas apenas, que o algoritmo escolherá a de maior pontuação para executar a jogada.

Inicialmente, cada estado do tabuleiro seria avaliado para todas as peças, se tivessem 2 juntas seria um peso, com 3 seria outro, porém não é uma função objetivo eficaz pois não avalia o que realmente pode ocorrer, se uma vitória, uma derrota ou empate, por isso decidimos apenas considerar os casos de estado final, pois são os estados objetivo do jogo.

3. **Testes**

Os testes executados (N vezes) para garantir que o JogadorTux funcione, foram:

a) JogadorTux x Aleatório
b) Aleatório x JogadorTux
c) JogadorTux x AleatórioFocado
d) AleatórioFocado x JogadorTux
e) JogadorTux x Humano
f) Humano x JogadorTux
g) JogadorTux x JogadorTux (versão anterior)
h) JogadorTux (versão anterior) x JogadorTux

i) JogadorTux x JogadorTux

Onde, os itens *a*, *b*, *c*, *d*, *g* e *h*, o JogadorTux deve vencer, por se tratar de oponentes com uma heurística inferior, ou sem nenhuma. Em *e* e *f*, foram testados anormalidades nas decisões do JogadorTux. Em *i* o resultado deve ser empate, indicando que o JogadorTux se comporta adequadamente independente se começa ou não a partida.

4. **Considerações finais**

Como melhoria do JogadorTux, na terceira etapa (Min-Max), se existisse um estado hipotético criado pelo algoritmo, onde existe uma ameaça dupla do oponente, ou do JogadorTux, um peso maior deveria ser atribuído a este estado, já que, se o jogo conduzir a ele, a vitória ou derrota é imediata.

Outra melhoria, mais drástica, seria aplicar o algoritmo e a função objetivo de corte Alfa Beta presente no programa Fhourstones, como mencionado por John Tromp ao fazermos as seguintes perguntas:

Diego e Fábio: ...when my program do the 8º move, and this is a "win" move, than is it guaranteed that my program will win?

John Tromp: My database classifies positions as wins, not moves. If the position is a win, then there's at least one winning move. To find it, you'll still have to do an exhaustive alpha beta search. But with a program like Fhourstones, that will only take a second or so for positions of >= 8 plies.

Diego e Fábio: Shouldn't exist a 10-ply, 12-ply, 14-ply, etc... database?

John Tromp: No, that would be a waste of space, since there are increasingly many positions while they become increasingly easier to solve.

John Tromp é PhD e pesquisou sobre o tema "Aspects of Algorithms and Complexity".

Pelo fato do JogadorTux ter sido construído com um bom embasamento teórico e fazer uso de uma base de dados que indica a melhor jogada possível em uma determinada etapa, além do fato de ter passado em todos os testes, onde isso indica que este é um jogador eficiente, é por isso que acreditamos que ele terá um bom desempenho na competição entre

os grupos, já que consideramos que a maioria dos grupos deve ter implementado apenas o Min-Max.

Apêndice A

Abaixo a bibliografia utilizada como apoio no desenvolvimento do projeto:

1. http://www.farfarfar.com/games/connect_four/connect4.pdf, último acesso em 08/12/2007 às 19h.
2. http://homepages.cwi.nl/~tromp/c4/c4.html, último acesso em 08/12/2007 às 19h10m.
3. www.ccs.neu.edu/home/eclip5e/classes/csu520/Smart**ConnectFour**-Final_Paper-v1.0.doc, último acesso em 08/12/2007 às 19h20m.
4. Russel, Norvig. Inteligência Artificial.
5. http://en.wikipedia.org/wiki/Serialization#Java, último acesso em 08/12/2007 às 21h.
6. http://en.wikipedia.org/wiki/Connect_four, último acesso em 08/12/2007 às 22h.