UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA - CIÊNCIA DA COMPUTAÇÃO
INF01048 - INTELIGÊNCIA ARTIFICIAL – TURMA U – 2012/1

RELATÓRIO TRABALHO PRÁTICO (Jogo Tetralath)

Prof. Paulo Martins Engel

Jefferson Rodrigo Stoffel - 180685
João Luiz Grave Gross - 180171

Porto alegre, 18 de junho de 2012.

Sobre o Tetralath

Tetralath é um jogo de competição para dois jogadores. As partidas são realizadas sobre um tabuleiro hexagonal de 61 posições. O jogo começa com o tabuleiro vazio e os jogadores depositam alternadamente peças nas posições vazias até que um jogador vença ou não haja mais posições livres. Cada jogador possui peças de uma cor, podendo estas ser brancas ou pretas. O jogador que possuir as peças brancas começa. Um jogador vence quando formar uma linha de quatro peças consecutivas. Um jogador perde se formar uma linha de três peças consecutivas. Um jogador vence se formar uma linha de quatro peças consecutivas em uma direção e uma linha de três peças consecutivas em outra direção. O jogo empata se o tabuleiro não possuir mais posições livres e não há um vencedor.

1 Características gerais do programa

1.1 Interface

A interface é intuitiva e de fácil utilização. Para executar o jogo, basta abrir uma tela do terminal e acessar a pasta onde está o executável e rodar o jogo através do comando ‘./tetralath’. Logo que o jogo começa é perguntado ao usuário quem começa a jogar, se a IA ou se o jogador humano. A imagem a seguir mostra isso:

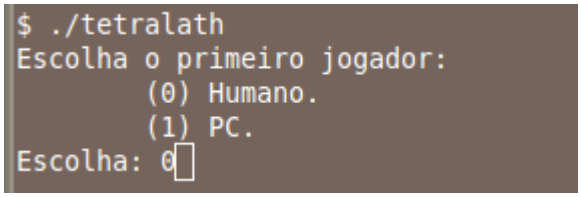

Figura 1: execução do jogo e escolha do jogador inicial

Logo após a escolha do jogador inicial o tabuleiro é carregado. A IA faz as jogadas automaticamente, e o jogador humano deve marcar a posição desejada que esteja livre.

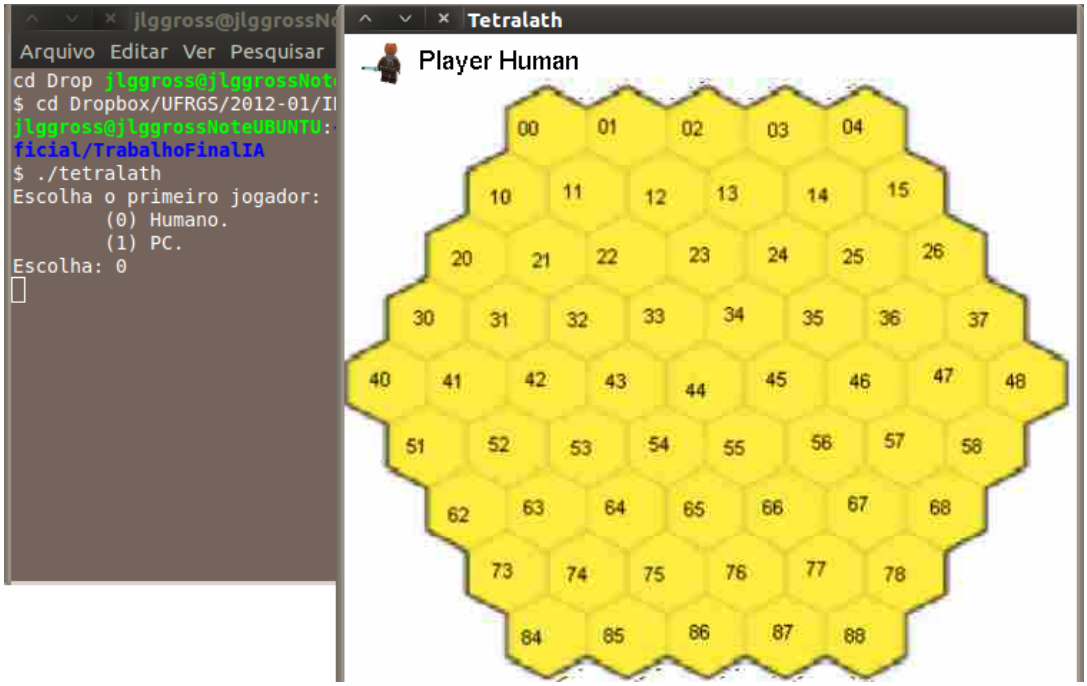

Figura 2: interface carregada após a escolha do jogador

2 Estrutura de dados do tabuleiro e sua manipulação

Este capítulo se destina a detalhar as estruturas de dados utilizadas para a representação do tabuleiro, bem como as funções de manipulação dessas estruturas.

2.1 Tabuleiro

O tabuleiro é composto de uma matriz 9 x 9 de estruturas do tipo:

```
typedef struct {
        int color;
        int neighbors[NEIGHBORS];
} Position;
```

Quando o programa é inicializado o campo color recebe NO_COLOR para as posições válidas e INVALID para as posições inválidas (só são usadas 61 das 81 posições da matriz). Para cada posição válida é armazenado o número correspondente à posição de todos os seus vizinhos no vetor neighbors. Na figura abaixo vemos a representação desta matriz: em amarelo as posições válidas e em cinza as posições inválidas. Vemos ainda o númeração de cada posição, que é obtida pela composição do número da linha (dezena) com a coluna (unidade).

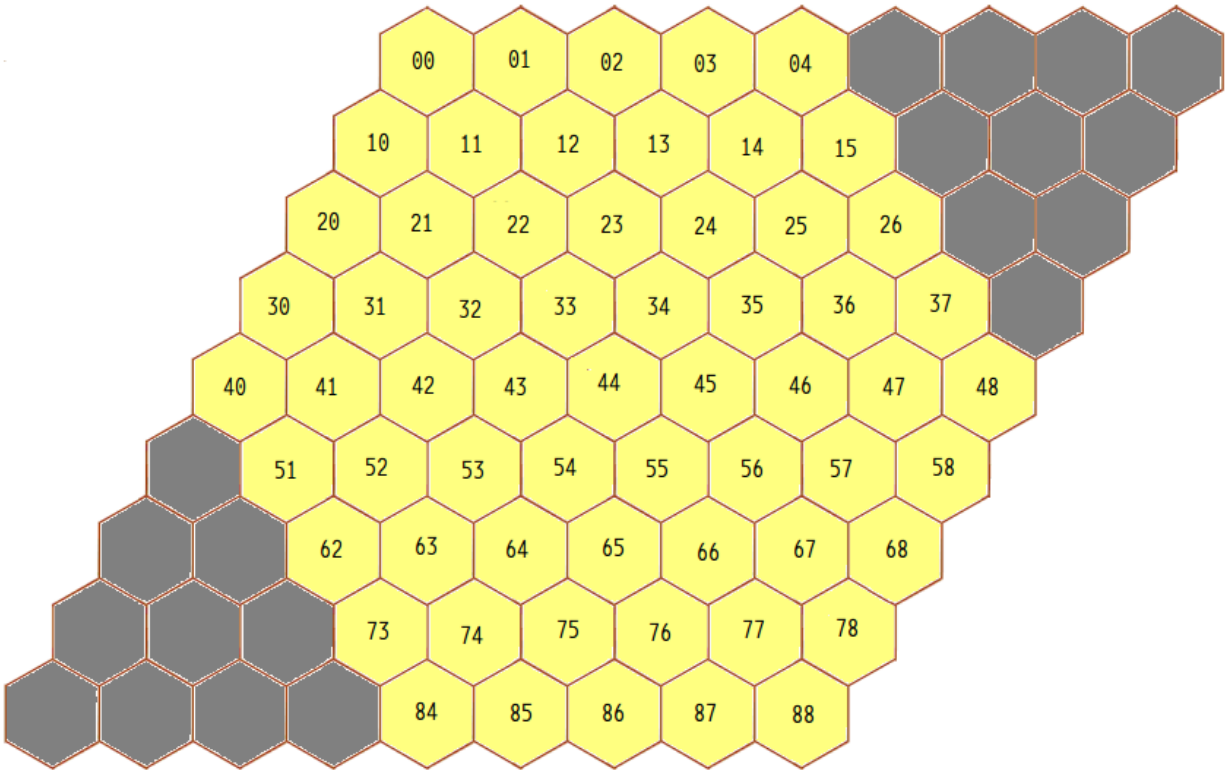

Figura 3: representação do tabuleiro

Como podemos notar, deslocamentos horizontais são simples operações de adição e subtração no número das colunas. Para os deslocamentos diagonais são usadas duas lógicas diferentes. Para todos os deslocamentos na direção canto direito superior até o canto esquerdo inferior, temos operações de adição e subtração no número de linhas. Por exemplo, para ir da posição 00 à posição 40 podemos notar que apenas há incrementos nas linhas. Já para ir da posição 00 à posição 88 um outro comportamento ocorre, onde a linha e a coluna são incrementadas gradativamente até chegar à posição de destino 88.

Logo, dada uma posição, ela possui no máximo 6 caminhos de movimento, e também no máximo 6 peças vizinhas. Para cada um destes 6 movimentos há um comportamento incremental de linhas e colunas diferente. Nas figuras abaixo vemos esses deslocamentos para uma posição e de que maneira os vizinhos são armazenados no vetor de cada posição.

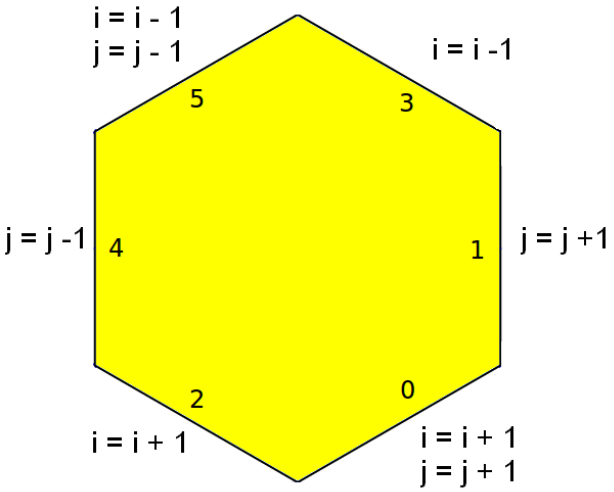

Figura 4: dada uma posição ij, sabemos qual é a posição i'j' dos seus vizinhos

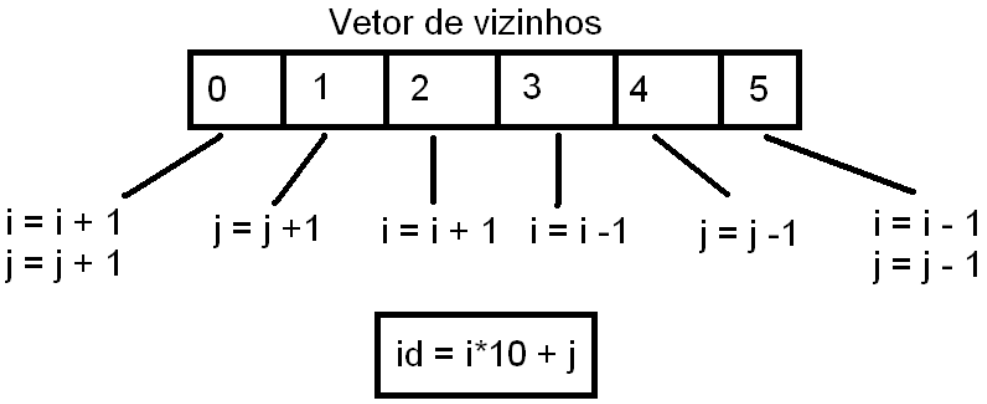

Figura 5: representação do vetor 'neighbors'. Cada posição corresponde a um vizinho específico

2.2 Funções de manipulação

Para a manipulação desta estrutura criamos duas funções para criação (alocação) e inicialização dos valores. Também criamos uma função auxiliar que nos ajuda a identificar o valor da posição que identifica cada vizinho.

2.2.1 Função de criação

Nesta função é alocada memória para a representação da estrutura do tabuleiro. São alocadas 81 estruturas do tipo Position, em uma matriz 9 x 9. Aqui não é feita nenhum tipo de inicialização.

2.2.2 Função de inicialização

Nesta função para cada um dos elementos válidos demonstrados na figura 3, o vetor de vizinhos é inicializado com o id de todos os vizinhos válidos e a variável cor é inicializada com o valor NO_COLOR, indicando que esta posição do tabuleiro não tem cor e está disponível para ser marcada por um dos dois jogadores.

Os elementos inválidos da matriz tem a variável cor inicializada com INVALID e o vetor de vizinhos fica com todas as suas posições inicializadas com INVALID.

No caso de elementos válidos com menos de 6 vizinhos, as posições do vetor ‘neighbors’ onde não existe um vizinho válido também são inicializadas com INVALID. As demais seguem inicializadas com o id do vizinho.

2.2.3 Função auxiliar para obtenção do id

Esta função recebe como argumentos dois valores, o id da posição que está buscando por vizinhos e um valor correspondente ao índice do vizinho do vetor de vizinhos, podendo variar de 0 a 5, tal como consta na figura 5. O objetivo desta função é retornar o id do vizinho, dado o id de uma posição e um índice do vetor de vizinhos, ou seja, o índice de qual vizinho queremos obter o id.

O exemplo apresentado a seguir deixa mais claro o funcionamento desta função:

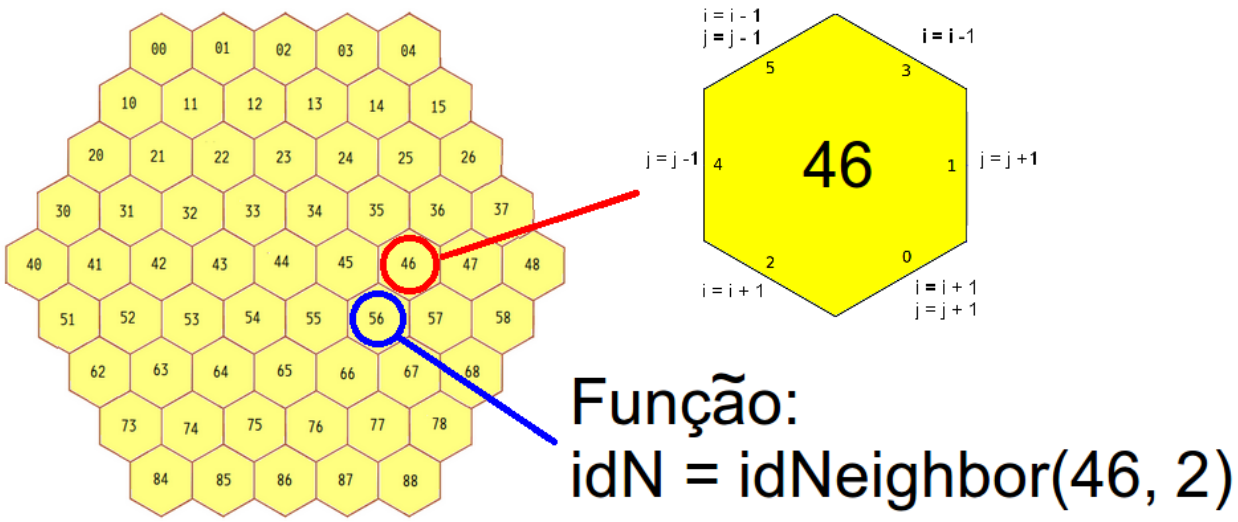

Figura 6: aplicação da função auxiliar idNeighbor(), que retorna o valor 56, correspondente ao vizinho de índice 2

3 Descrição detalhada da função de avaliação

A função de avaliação adota uma estratégia de fácil entendimento. Basicamente, dado o conjunto de peças marcadas de um jogador à função de avaliação, ela pega cada peça deste jogador, marcada no tabuleiro, e varre os seus vizinhos.

Cada vizinho é considerado no algoritmo como uma caminho de varredura. Os caminhos são varridos em linha, retornando um valor associado a jogada mais favorável à IA. O exemplo abaixo mostra como se faz a varredura:

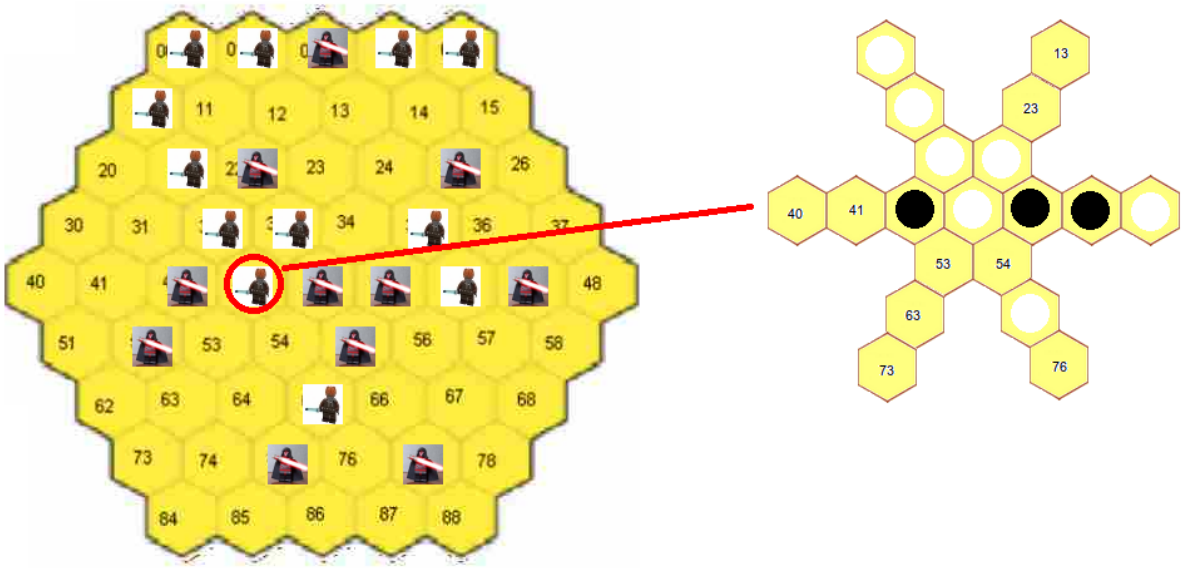

Figura 7: como a função de avaliação ‘vê’ uma posição marcada de um jogador

A figura 7 nos dá uma noção de como a função de avaliação vê cada posição marcada de um jogador. Toda a peça marcada tem seus vizinhos analisados e a partir daí começa a fazer varreduras em linha. Cada caminho é avaliado até fechar 4 peças em linha ou até ser encontrada uma peça do adversário. No exemplo acima a função de avaliação está avaliando a posição 43. As varreduras à esquerda e à direita são consideradas ruins, pois encontram peças do adversário. A varredura na direção das posições 53, 63, 73 é considerada regular, pois tem um caminho livre para jogadas futuras. As varreduras pelas linhas 54, 65, 76 e 33, 23, 13 são caminhos considerados bons, pois já estão melhor preparados para uma jogada de vitória futura. Já o caminho considerado melhor, e de maior valor, 10, 21, 32 forma 4 peças em linha da mesma cor, ou seja um vitória, então o jogo tem que tender a este estado de tabuleiro.

Este processo é refeito para todas as peças do jogador e assim sempre temos como retorno o valor da melhor jogada no tabuleiro. Para cada tabuleiro criado no minimax, simulando jogadas esta análise também é feita. Logo, dentre todos os tabuleiros simulados com jogadas pelo minimax e dentre todas as jogadas de cada tabuleiro, sempre sabemos qual dos tabuleiros é o melhor.

Vale lembrar que a IA tenderá a atingir o estado de tabuleiro que gerou mais vantagem nas análises da função de avaliação.

4 Detalhes sobre a implementação do minimax

A raiz da função pode ser representada pelo seguinte pseudo-código:

```
seedMinimax {
        se testTwoEmptyOne
                retorna jogada
        senão
                α ← -∞
                para cada espaço livre no tabuleiro faça
                        α ← max(α, expandTree(tabuleiro, profundidade, adversário)))
                fim para
                retorna α
        fim senão
}
```

Onde testTwoEmptyOne é uma função que verifica se um dos jogadores está prestes a fechar uma linha de quatro peças consecutivas. Se a IA estiver nesta situção, sua próxima jogada será de vitória. Caso o adversário esteja nesta situação, a próxima jogada da IA será uma ação de bloqueio. A função expandTree é efetivamente o algoritmo de minimax, que pode ser representado pelo seguinte pseudo-código:

```
expandTree(tabuleiro, profundidade, jogador) {
   avalia se alguém ganhou
        se alguém ganhou retorna vitória para aquele jogador
   senão
        se profundidade > 0
                se é a jogada do adversário
                        α ← +∞
                        para cada espaço livre no tabuleiro faça
                                marca tabuleiro
                                α ← min(α, expandTree(tabuleiro, profundidade - 1, ¬jogador)))
                        fim para
                        retorna α
                fim se
                senão
                        α ← -∞
                        para cada espaço livre no tabuleiro faça
                                marca tabuleiro
                                α ← max(α, expandTree(tabuleiro, profundidade - 1, ¬jogador)))
                        fim para
                        retorna α
                fim senão
        senão
                realiza função de avaliação
                retorna resultado da função
```

```
        fim senão
fim senão
fim senão
}
```

4.1 Profundidade alcançada

Para permanecer dentro do limite de 5 segundos, a máxima profundidade alcançada é três. Com o evoluir do jogo, esse número poderia aumentar devido a queda do número de ramificações, porém não é o caso do programa que construímos, que trabalha com um valor constante.

4.2 Tipo de otimização utilizada

Foram utilizadas duas otimizações. A poda alpha beta, que evita que sejam calculados ramos que não trarão nenhum novo resultado para o minimax. A outra função utilizada foi a função “testTwoEmptyOne” que avalia se existe um jogador prestes a fechar um linha de quatro peças consecutivas. Primeiramente a IA verifica se pode vencer em sua próxima jogada. Se puder, retorna essa jogada. Se não, verifica se o adversário poderá fazê-lo. Se sim, retorna essa jogada. Caso contrário, executa o minimax. Portanto, esta otimização evita que o minimax seja aberto para uma jogada óbvia.

5 Análise do desempenho no campeonato

Na primeira rodada do campeonato não obtivemos um bom desempenho, pois nosso algoritmo do minimax não estava funcionando e não conseguimos arrumá-lo a tempo. Nesta fase a IA marcava 3 peças em linha sem ser forçada a isso, ou seja, voluntariamente gerava uma situação de derrota, o que nos gerou a maioria das derrotas.

Já na segunda rodada conseguimos arrumá-lo, mas apenas em parte. A IA não marcava mais voluntariamente situações de derrota, porém a marcação das peças seguia em ordem sequencial no tabuleiro. Conseguimos também implementar bloqueios de vitória do adversário e também a gerar a vitória da IA.

Analisando os códigos após o campeonato descobrimos a causa do problema. No algoritmo de minimax houve uma troca nas funções de chamadas: quando era calculado Max, era retornando Min com seu valor de inicialização e vice-versa. Desta forma, todos os nós folhas tinham o mesmo valor, o que ocasionava a escolha sempre do primeiro valor, e por essa razão o algoritmo marcava o tabuleiro de forma sequencial.

O desempenho na segunda rodada melhorou com relação a primeira, mas poderia ter sido melhor se conseguíssemos evitar a marcação sequencial das peças no tabuleiro, pois desperdiçamos muitas peças em posições do tabuleiro que não contribuíam à vitória.