# Relatório EP5 - MAC0121

João Gabriel Basi - N° USP: 9793801

## 1 O programa

O programa implementa uma IA capaz de jogar o jogo Hex, um jogo de tabuleiro onde os jogadores se alternam colocando uma peça de sua cor no tabuleiro, com o objetivo de fazer um caminho entre as bordas opostas do tabuleiro de sua cor. O primeiro a conseguir fazer um caminho vence.

## 2 As funções

### 2.1 No arquivo da main (hex.c)

- *show_usage*: Mostra uma mensagem de erro na saída de erro de acordo com o número fornecido;
- *play*: Faz uma jogada;
- *beginGameLoop*: Começa o loop do jogo;

### 2.2 BoardFuncs.h

Biblioteca de funções que implementam o tabuleiro e ajudam a manuseá-lo.

- *BTile*: Struct que implementa uma casa do tebueiro;
- *BoardCreate*: Cria um tabuleiro;
- *freeBoard*: Desaloca um tabuleiro;
- *printGame*: Imprime o tabuleiro na saída de erro;
- *isValid*: Verifica se a posição do tabuleiro é válida;
- *getWinner*: Verifica se alguém venceu;
- *updateWeigths*: Atualiza os pesos das peças do tabuleiro.

## 2.3 PathFinding.h

Biblioteca de funções que ajudam o programa na criação e no manuseio de caminhos pelo tabuleiro.

- *macro e macroAux*: Vetores que auxiliam o programa à achar caminhos pelo tabuleiro;
- *PNode*: Struct que implementa um nó de lista ligada Path;
- *Path*: Struct que implemneta uma cabeça de lista ligada para um caminho de posições do tabuleiro;
- *PathDestroy*: Destroi uma lista ligada do tipo Path;
- *Intersec*: Acha a primeira posição comum entre dois caminhos;
- *findPath*: Acha um caminho entre as duas laterais opostas de uma cor;
- *pathcmp*: Vê se dois caminhos são iguais.

## 2.4 auxfuncs.h

Biblioteca de funções que ajudam o programa a manusear a memória e resumir operações.

- *N*: Define o tamanho do tabuleiro;
- *INF*: Define infinito como sendo $2^{30}$;
- *Pos*: Struct que armazena uma posição do tabuleiro;
- *emalloc*: Aloca uma posição de memória e imprime uma mensagem de erro se der errado;
- *criaMatriz*: Aloca uma matriz;
- *freeMatriz*: Desaloca uma matriz;
- *criaVetor*: Aloca um vetor;
- *coltoi*: Transforma uma cor em um inteiro;
- *troca*: Troca o conteúdo de duas variáveis.

# 3 Implementação da IA

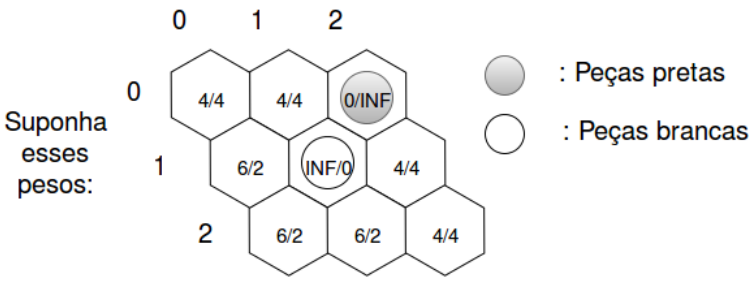
Esse programa não foi baseado em nenhum outro algoritmo existente (ao menos eu não achei nenhum parecido). A ideia dele é achar o suposto caminho que o outro jogador está tentando construir e tentar barrá-lo, para isso só foi preciso implementar uma função que acha um caminho vantajoso (nem sempre é o melhor possível) para cada jogador e outra que acha uma posição em comum entre eles (no caso, como nunca há empates, pois um jogador precisa bloquear a passagem do outro para vencer, podemos garantir que sempre há uma posição em comum entre os dois caminhos).

## 3.1 Pesos

Primeiro damos dois pesos para cada célula do tabuleiro, um em relação às peças brancas e outro em relação às peças pretas. Se uma célula está vazia e tem vizinhos vazios, seu peso é 4 para as duas cores. Para cada peça branca colocada em sua vizinhança, seu peso para as brancas diminui em 2 e seu peso para as pretas aumenta em 2, e o peso para as pretas funciona analogamente. Cada casa vazia pode ter no máximo peso 8 e no mínimo peso 0. Para as casas ocupadas, o peso é definido como sendo 0, para a cor da peça que está lá, e INF para a cor oposta. Quanto menor o peso de uma casa em relção à uma cor, melhor ela é para essa cor.

## 3.2 Achando o caminho

Para achar um caminho, a função findPath chama a função partialFindPath em todas as posições de uma das bordas de um jogador, e a partialFindPath executa um backtracking recursivo a partir daquela posição. O backtracking tem uma prioridade de verificação para cada cor de peça e lado do tabuleiro, essa prioridade é calculada a parir dos vetores macro e macroAux, como mostram as figuras:




Os pesos são dados por: <peso correspondente às pretas>/<peso correspondente às brancas>

(Obs.: Se não houver vizinho o peso passa a ser INF)

A função findPath chama a função partialFindPath em cada célula da borda branca para tentar achar um caminho. Vamos supor que a partilaFindPath foi chamada na célula (0,1) do tabuleiro:

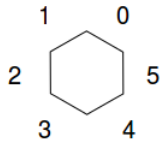
A função partialFindPath primeiro chama a função getPriority para saber qual vizinho deve ser checado primeiro (o campo de paridade serve para que a função faça um caminho perpendicular à borda analizada, nesse caso vamos começar com 1)

getPriority(board, 0, 1, 'b', 1, 1) :

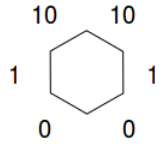
decode('b' ,1) => [10, 10, 1, 0, 0, 1]
decodeAux('b', 0) => [1, 0, 5, 4, 3, 2]

Cada célula do tabuleiro tem um vetor de ponteiros para os seus vizinhos. Eles estão organizados dessa maneira:

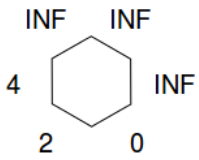
Cada célula do tabuleiro tem um vetor de ponteiros para os seus vizinhos. Eles estão organizados dessa maneira:




O vetor decode contém pesos que são colocados nos índices indicados pelo vetor do decodeAux. Então a célula fica assim:




Por fim é somado o peso de cada vizinho na ordem do decodeAux ao vetor do decode:



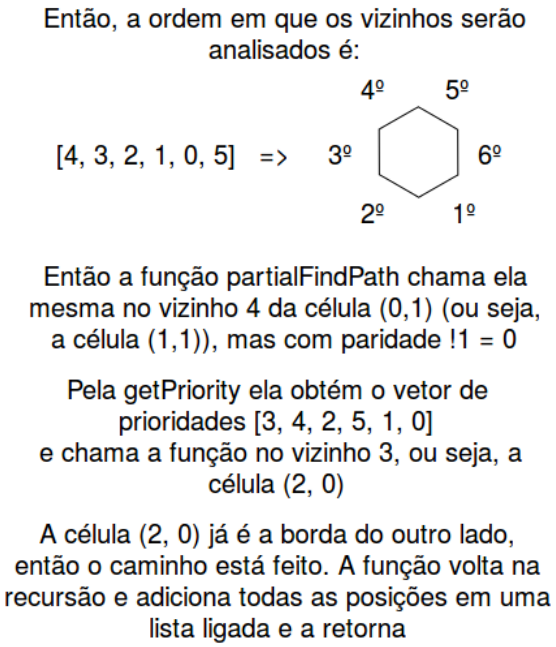

fazendo pesosVizinhos[i] = pesosVizinhos[decodeAux[i]]

==========> [INF, INF, INF, 0, 2, 4]

[INF, INF, INF, 0, 2, 4]
\+
[10, 10, 1, 0, 0, 1]
= [INF, INF, INF, 0, 2, 5]

O vetor de pesos obtido é pareado com o vetor do decodeAux e é feito um sort simultâneo de acordo com o vetor de pesos

[INF, INF, INF, 0, 2, 5]
[1, 0, 5, 4, 3, 2]
=>
[0, 2, 5, INF, INF, INF]
[4, 3, 2, 1, 0, 5]




A interseção dos dois caminhos será onde o programa irá fazer sua jogada.

# 4 Decisões de implementação

A minha intenção em criar o vetor macroAux é de alternar a prioridade de vizinhos simétricos e com o mesmo peso, achando caminhos tanto por um lado quanto por outro. Utilizei da estabilidade do bubble sort (realizado no getPriority) para obter prioridades diferentes para a mesma célula utilizando o vetor macroAux.

O algoritmo, foi construído com partes de outros algoritmos. Por exemplo, o algoritmo que acha o caminho foi baseado no algoritmo A* (não implementei com fila pois não sabia como armazenar o caminho) e os pesos baseados em uma implementação do jogo pensando nas peças como resistores [1].

# 5 Prós e contras

## 5.1 Prós

- Joga bem no começo, bloqueando os possíveis caminhos que o outro jogador possa fazer;
- Não achei nenhum algoritmo símples (como fazer um caminho em linha reta ou jogar aleatóriamente) que possa vencê-lo.

### 5.2 Contras

- Conforme as rodadas passam, ele fica mais confuso, pois não consegue achar direito os caminhos em meio à tantas peças, então as vezes faz jogadas que não adiantam em nada;

## Referências

[1] Hexy `http://home.earthlink.net/~vanshel/VAnshelevich-01.pdf`