Documentação do Trabalho Prático 0 – Caça aos Pokémons

1. **Introdução**

A manipulação de Tipos Abstratos de Dados (TAD) é extremamente importante para projetos maiores de algoritmos e estruturas de dados. O objetivo deste trabalho é criar, no mínimo, um TAD armazenando os detalhes da implementação do mecanismo do jogo. Esse TAD deve conter obrigatoriamente as funções explorar, andar e caminho_percorrido.

2.**Implementação**

Estrutura de Dados:

Para a implementação do trabalho foi criado um Tipo Abstrato de Dados Jogador com a seguinte composição:

```
4  typedef struct
5  {
6      char nome[30] ;
7      int *vet_caminho;
8      int p_inicial[2];
9      int p_atual[2];
10     char char_init[4];
11     int score;
12     int **vizinhos;
13     int pokebolas;
14     int jogadas;
15 } Jogador;
```

Nome representa o nome do jogador, *vet_caminho armazena o caminho percorrido de cada jogador, p_inicial armazena a posição inicial do jogador, p_atual representa a posição atual do Jogador, char_init foi utilizado para facilitar a conversão da posição inicial ao ler o arquivo, score armazena a pontuação do jogador, **vizinhos armazena a vizinhança do jogador em determinada rodada, pokebolas representa o numero de pokebolas restantes e jogadas representa o numero de jogadas feitas por um jogador.
Note que vet_caminho precisa ser alocado dinamicamente para 3x(tamanho lateral do campo).
Além disso, ****vizinhos** é alocado como uma matriz 8x3, onde cada um dos 8 vetores contem o valor do vizinho e sua localização:

0	1	6
-7	6	2
2	0	-3

```
8   jogador[i].vizinhos[0] = -3; //valor
9   jogador[i].vizinhos[1] =  2; //posição
10  jogador[i].vizinhos[2] =  2; //posição
```

Funções e procedimentos:

O TAD possui as seguintes funções

int andar(int vet[], int player, Jogador *play, int **validos) : recebe como parâmetro um vetor com os valores das casas vizinhas, o numero do jogador, o jogador e uma matriz contendo os vizinhos imediatos. Essa função é responsável por mover o jogador quando ele tem pokebolas

restantes e não excedeu o numero de rodadas.

A função entra em um loop procurando o maior valor em **vet[]**, diferente de zero e ainda válido. Caso não encontre, o jogo acaba. Caso encontre, ele retorna o "**i**" do loop em que o maior termo foi encontrado, para que outras informações, como posição, possam ser encontradas no vetor da matriz **vizinhos** na mesma posição:

```
 6  if(jogador[i].pokebolas > 0){
 7  posição = andar(vet[], i, jogador, validos);
 8
 9  jogador[i].score =  jogador[i].score + jogador[i].vizinhos[posição][0]; //soma score
10  jogador[i].pokebolas--;                                                 //subtrai pokebolas
11  jogador[i].p_atual[0] = jogador[i].vizinhos[posição][1];                //ajusta posição
12  jogador[i].p_atual[1] = jogador[i].vizinhos[posição][2];                //ajusta posição
13  validos[jogador[i].p_atual[0]][jogador[i].p_atual[1]] = 0;              //impede de ser usado novamente
14  jogador[i].jogadas++;                                                   //soma nuemro de jogadas
15  }
```

Quando não é possível andar, a função retorna 404.

int semPokestop(int vet[], int player, Jogador *play, int **validos) : recebe os mesmos parâmetros da função **andar** , porém é responsável por mover o jogador quando ele está sem pokebolas e não existe um pokestop imediatamente próximo.

int pokestop(int vet[], int player, Jogador *play, int **validos) : recebe também os mesmos parâmetros de **semPokestop** e **andar**, porém só é utilizada quando o jogador não possui mais pokebolas e não excedeu o numero de rodadas. Caso essa função determine que não existe um pokestop próximo, é chamada a função **semPokestop**.

int **explorar(Jogador *play, int i,int camp,int **mapa) : esta função recebe como parâmetro um Jogador, o número do jogador, o tamanho do mapa e uma matriz contendo o mapa. Ela é responsável por determinar os vizinhos existentes ao redor da posição atual do jogador.
Essencialmente, a função lê o que existe nas posições ao redor do jogador e armazena valores e posições na matriz vizinhos[8][3].

int caminho_percorrido(Jogador *play, int player, int i) : recebe como parâmetro um Jogador, o numero do jogador e posição do vetor que armazena o caminho percorrido desejada. Ela retorna um inteiro correspondente a uma posição na matriz.
Basicamente, retorna sequencialmente o conteúdo de ***vet_caminho**, facilitando a impressão no arquivo de saída.

Programa principal:

O programa principal, main.c, abrimos os arquivos de entrada e saída e alocamos algumas variáveis, entre elas:
mapa: matriz contendo as informações do mapa;
validos: matriz contendo as posições do mapa que ainda não foram acessadas;
play: vetor de struct Jogador
play[i].vizinhos: matriz com informações dos vizinhos.

Para realizar as jogadas, temos um **while** dentro de um **for**. O **for** é responsável por mudar o jogador, e o **while** pelo loop de cada jogada. Antes do **while** setamos algumas variáveis :

```
5  rodada = 0;              //rodada atual
6  playing = true;          //condição de loop
7  play[i].pokebolas = 3;   //pokebolas
8  play[i].jogadas = 0;     //numero de jogadas
```

Como primeira ação do loop, temos a condição de parada

```
if(rodada > r_max){playing = false;}
```

em que r_max é o numero máximo de rodadas.

Para iniciar a rodada, usamos a função **explorar,** assim podemos tomar decisões. Então, se o jogador possui pokebolas, chamamos a função **andar**, se ela retorna **404**, finalizamos a rodada. Se o jogador não possui mais pokebolas, chamamos a função **Pokestop**, caso não exista nenhum chamamos a função **semPokestop,** caso não exista movimentos válidos, encerramos a rodada. Após cada uma da funções, atualizamos as variáveis do jogador, como Score, numero de pokebolas, posição atual, posições válidas e numero de jogadas.

Para determinar o ganhador, primeiro verificamos se existem jogadores com pontuações iguais, caso não existam, o maior pontuador vence. Caso existam posições iguais, verificamos se o numero de jogadas de cada um é a mesma, caso seja, todos os empatados ganham, caso não, os/o com menor número de jogadas ganha.

<u>Organização do código e detalhes técnicos:</u>

O código está dividido em 3 arquivos: main.c, jogador.c e jogador.h.
Todo o código foi feito em um editor de texto simples e compilado pelo terminal com um Makefile (disponível na pasta do tp) . Portanto pode ser compilado com GCC usando o comando “make” no diretório dos arquivos.

Análise de complexidade

int andar(int vet[], int player, Jogador *play, int **validos) :
A função se inicia com algumas atribuições e entra em um loop de **i = 0** até **i = 7,** assim sendo uma função de ordem **O(1).**

int semPokestop(int vet[], int player, Jogador *play, int **validos) :
A função tem o mesmo funcionamento de **andar()**, mudando apenas o valor das variáveis, portante também tem ordem de complexidade **O(1).**

int pokestop(int vet[], int player, Jogador *play, int **validos) :
Assim como a função **semPokestop(),** esta função tem apenas algumas atribuições e um loop de tamanho pré-definido, portanto possui ondem de complexidade **O(1).**

int **explorar(Jogador *play, int i,int camp,int **mapa) :
A função possui 8 grupos de 3 atribuições, por não possuir loop ela tem a ordem de complexidade **O(1).**

int caminho_percorrido(Jogador *play, int player, int i):
Sendo uma função bem simples, ela apenas retorna a posição desejada do vetor inserido de forma a facilitar a impressão do caminho percorrido no arquivo de saída, portanto possui a ordem de complexidade **O(1).**

Programa Principal:
O programa principal escolhe entre chamar a função **andar()** e a função **pokestop()**, e caso **pokestop()** não ache um movimento provável, ela chama **semPokestop()**, por essa razão cada loop do programa chama no máximo duas funções. Pelo fato de que o loop é definido pelo numero de jogadores, sua ordem de complexidade seria **O(n)**, entretanto existe um **while** dentro do loop, que

depende principalmente do número máximo de jogadas. Como o número máximo de jogadas depende do tamanho do mapa, a ordem do **while** seria **O(n)**, no pior caso, em que o jogador nunca fica preso por células já visitadas. No melhor caso, o jogador jogaria 9 vezes e se cercaria, encerrando o jogo. Portanto, pelo **while** dentro do **for**, o programa principal teria a ordem de complexidade de **$O(n^2)$.**

Testes:
Para realizar os testes, foram usados os arquivos disponibilizados pelos professores.
(apenas o primeiro será apresentado neste documento, uma vez que os demais são demasiadamente grandes)

entrada.txt

```
8
4 0 0 0 4 6 4 0
6 0 4 -7 0 3 5 -2
6 6 0 0 6 0 0 6
5 0 0 4 0 6 0 0
0 0 0 0 0 5 0 6
0 1 3 2 -8 0 6 5
6 0 1 2 5 0 0 6
4 1 2 6 0 6 6 0
4
J1: 3,6
J2: 6,7
J3: 3,3
J4: 3,6
```

saida.txt

```
J1: 74 7,5 6,6 7,7 7,7 6,7 5,6 4,6 5,6 6,5 7,4 6,3 5,2 4,2 5,1 6,2 7,3 7,2 6,1 5,1 4,0 5,0 6,0 7,1 7,0
J2: 84 6,7 5,6 6,5 6,4 7,5 7,4 6,3 5,2 4,3 3,4 4,4 5,3 6,2 7,1 6,0 7,0 6,0 5,0 4,1 4,1 5,2 6,3 7,0
J3: 56 4,3 5,4 6,4 7,5 6,6 7,7 7,7 6,7 5,6 4,6 5,6 6,5 7,0
J4: 74 7,5 6,6 7,7 7,7 6,7 5,6 4,6 5,6 6,5 7,4 6,3 5,2 4,2 5,1 6,2 7,3 7,2 6,1 5,1 4,0 5,0 6,0 7,1 7,0
VENCEDORES J2:
```

No caso acima o jogador J2 vence por possuir o maior score.

<u>Conclusão:</u>

Depois de implementar e alterar diversas vezes, percebe-se que, mesmo que todo o conteúdo e técnicas a serem utilizadas estejam claras no campo teórico, é essencial colocá-los a pratica, uma vez que só assim podemos aprender e entender verdadeiramente o que está sendo feito.