DOCUMENTAÇÃO TP0

Ramon Gonçalves Gonze

INTRODUÇÃO

O objetivo deste trabalho prático é a implementação de conceitos discutidos em sala de aula, como ponteiros, listas encadeadas, análise de complexidade de funções, manipulação de vetores ,entre outros. Um conceito fundamental para a construção do algoritmo deste trabalho, são os Tipos Abstratos de Dados (TADs). Esses são estruturas nas quais organizam dados e categorias de forma objetiva. O trabalho é uma simulação simplificada de um jogo de Pokémon. No algoritmo teremos a manipulação de vetores estáticos e dinâmicos, que farão parte das operações dos TADs implementados. Para gerar o executável do arquivo, deve-se abrir o prompt de comando (Windows) ou o terminal (Linux), acessar a pasta que contém os 3 arquivos do tp, *main.c*, *funcoes.c* e *funcoes.h*, e então executar dois comandos: *gcc -c funcoes.c* e logo em seguida *gcc main.c funcoes.o*.

IMPLEMENTAÇÃO

❑ Estrutura de dados

Para a implementação do trabalho, foi considerada a seguinte tabela para indicar o que cada campo (posição da matriz) do mapa contém:

Descrição	Número
Campo vazio	0
Dragonite	1
Alakazam	2
Magmar	3
Eevee	4
PIkachu	5
Pokéstop	6
Campo inválido	7

Tabela 1: Indica a relação entre números e itens dos campos do mapa.

Foram criados 3 TADs:

- Jogador

  Esse TAD descreve os jogadores, e contém 7 itens:

  - char *nome → Variável que armazenará o nome do jogador.

  - int score → Variável que armazenará os pontos ganhos pelo jogador, baseado na quantidade de pokémons que ele capturar.

  - int passos → Variável que contará quantos passos o jogador já percorreu, e este será usado para limitar a quantidade de rodadas (3 * Tamanho_da_matriz - 1).

  - int coluna → Variável que armazena a coluna da matriz que o jogador se encontra.

  - int linha → Variável que armazena a linha da matriz que o jogador se encontra.

  - int pokebolas → Variável que armazena a quantidade de pokébolas que o jogador possui.

  - int pokemons[QTD_POKEMONS] → Vetor de tamanho 5 (valor definido por QTD_POKEMONS) que armazena a quantidade de cada pokémon que o jogador possui. A tabela 2 indica qual posição do vetor corresponde a cada Pokémon.

Pokémon	CP
Dragonite	0
Alakazam	1
Magmar	2
Eevee	3
Pikachu	4

Tabela 2 - Indica a posição do vetor correspondente ao pokémon.

- t_celula

  Esse TAD é uma célula de uma lista encadeada, e possui 3 itens:

  - int coluna → Variável que armazena a coluna do campo.

  - int linha → Variável que armazena a linha do campo.

  - struct celula *prox → Ponteiro do tipo t_celula, que aponta para a próxima célula da lista, ou aponta para NULL caso a célula em questão seja a última da lista.

- t_lista

  Esse TAD é o que define a lista, e possui 2 itens:

  - t_celula *primeiro → Um ponteiro do tipo t_celula, que é utilizado para apontar para a “cabeça” da lista.

  - t_celula *ultimo → Um ponteiro do tipo t_celula que é utilizado para apontar para o último item da lista.

- **Funções e procedimentos**

O TAD *Jogador* possui as seguintes funções:

void explorar(Jogador *player, int **mapa, int TAM_MATRIZ, int *vizinhanca)
Preenche o vetor de inteiros vizinhanca de tamanho 8 (valor definido por CAMPOS) que contém, de acordo com o esquema 1, os valores do mapa de todos os campos em volta do jogador. Caso o jogador se encontre em uma posição na qual alguma das posições de 0 à 7 se encontra fora do mapa, ela atribui o número 7 (CAMPO_INVALIDO) para o campo.

0	1	2
7	Jogador	3
6	5	4

Esquema 1: Valores dos campos em volta do jogador, que vai de 0 à 7.

void move_jogador(Jogador *player, int prox_posicao)
Essa função recebe o valor do campo - identificado de acordo com o Esquema 1 - no qual o jogador deverá se mover na rodada e armazena na váriavel *prox_posicao*. De acordo com a posição recebida, a função altera as variáveis inteiras *coluna* e *linha* do TAD Jogador.

int andar(Jogador *player, int **mapa, int *vizinhanca, t_lista *lista)
Essa função tem como propósito definir para qual campo do mapa o jogador deverá andar.
A regra do jogo diz que o número de passos que cada jogador pode dar é de (3 * Ordem_da_matriz) -1. A primeira ação desta função é verificar se o jogador ainda possui passos disponíveis para dar. Caso não tenha, a função retorna 0.
Os critérios para a escolha seguem a seguinte ordem:

→ Se jogador possui pokébolas:
- Caso o jogador possua pokébolas, ele procura o campo que possui o maior valor inteiro diferente de 6 (pokéstop) e 7 (campo inválido). Essa verificação é feita no vetor *vizinhanca*. Após retornar o maior valor , é verificado, em ordem crescente, qual posição do vetor vizinhança possui este maior valor. Então as funções *caminhoPercorrido* e *move_jogador* são chamadas.

→ Se o jogador não possui pokébolas:
- Ele procura no vetor *vizinhanca* o valor 6, que corresponde ao pokéstop. Caso encontre um campo com este valor, as funções *caminhoPercorrido* e *move_jogador* são chamadas.
- Se não houver pokéstops por perto, o algoritmo procura por um campo vazio analisando de forma crescente o vetor *vizinhanca*, e ao encontrar, as funções *caminhoPercorrido* e *move_jogador* são chamadas.

- Se não houver campos vazios, o algoritmo procura por um campo com o valor diferente de 7, analisando de forma crescente o vetor *vizinhanca*, e ao encontrar, as funções *caminhoPercorrido* e *move_jogador* são chamadas.

Se algum dos movimentos acima foi feito, a função retornará 1. Caso nenhum dos requisitos acima seja atendido, significa que o jogador se encontra em uma posição em que só há campos inválidos em volta dele, portanto a função retorna 0.

void iniciaJogador(Jogador *player, int **mapa, int TAM_MATRIZ)
Essa função tem o objetivo de atribuir os status iniciais dos jogadores. Estes são:
- Score: Começa com o valor do campo da primeira posição do jogador;
- Passos: Recebe o número de passos máximos que o jogador pode dar, que é (3 * Tamanho_do_mapa) - 1.
- Pokébolas: Se a primeira posição do jogador possui um pokémon, ele é pego, então o jogador começa com 2 pokébolas. Caso a primeira posição não contenha nenhum pokémon, o jogador começa com 3 pokébolas.
- Pokémons: O vetor *pokemons[QTD_POKEMONS]* é inicializado com 0 em todas as posições.

void criaLista(t_lista *lista, Jogador player)
Cria uma lista encadeada, que será usada para armazenar o caminho percorrido de cada jogador durante as rodadas. A função cria uma nova célula (variável do TAD t_celula) e faz os ponteiros *primeiro* e *ultimo* (do TAD t_lista) apontarem para ela. Armazena a posição inicial do jogador nas variáveis *coluna* e *linha* dessa nova célula.

void caminhoPercorrido(t_lista *lista, Jogador player)
Essa função insere um elemento no fim da lista encadeada, que já está criada. Ela é chamada na função *andar* antes do jogador fazer o seu próximo movimento. Assim que ela é chamada, ela cria uma nova célula (variável do TAD t_celula) e armazena a posição atual do jogador (antes do movimento). Após criada, o ponteiro *ultimo* da lista passa a apontar para esta nova célula.

Os TADs *t_celula* e *t_lista* possuem as funções *andar*, *criarLista* e *caminhoPercorrido*.

A função *maior_valor* não é utilizada por nenhum dos TADs apresentados.

int maior_valor(int vetor[], int tam)
Recebe um vetor de inteiros e retorna o maior valor contido neste vetor.

❑ Programa principal

O programa principal pode ser dividido em 6 partes:

1. Faz a leitura do arquivo de entrada, atribui as informações às suas respectivas variáveis: Ordem da matriz do mapa do jogo, o mapa, número de jogadores, nomes e posições iniciais dos jogadores.
2. Inicia todos os jogadores através da função *iniciaJogador*.
3. Executa as jogadas de todos os jogadores. Há um loop no programa principal que, a cada movimento que o jogador faz, ele decresce em 1 o número de passos disponíveis do jogador. Quando a quantidade de passos disponíveis for igual a zero, o jogo termina para aquele jogador.
4. Após cada jogada finalizada, no loop principal, é escrito no arquivo de saida (saida.txt, que foi aberto no início da função main): Nome do jogador, seu score, e o caminho percorrido por ele.
5. Após escrita as informações do primeiro jogador no arquivo de saída, os ponteiros *mapa*, *vizinhanca* e *lista* são liberados, para o uso do próximo jogador. O mapa também é lido novamente no arquivo de entrada, pois ele foi alterado pelos passos do primeiro jogador.
6. Depois de todas as jogadas realizadas, o algoritmo conta quantos jogadores possui o maior score, e, caso haja mais de um, os testes de desempates são feitos. Após definido o(s) vencedor(es), é escrito seus nomes na última linha do arquivo de saída, e o programa é finalizado.

- **Análise de complexidade**

Todas as funções, exceto as funções main e maior_valor, têm ordem de complexidade O(1), pois elas se baseiam em constantes que não se alteram, independente do arquivo de entrada lido.

Função explorar: A função possui um laço for que é executado 8 vezes (constante CAMPOS), e dentro deste laço, há um switch que executa comandos O(1), como comparações e atribuições.

Função move_jogador: A função possui um switch que faz 1 ou 2 atribuições para todos os casos possíveis do valor da variável *prox_posicao*.

Função andar: Esta função possui algumas comparações que, em alguns casos, executa laços for 8 vezes (constante CAMPOS). O restante da função faz somente comparações, atribuições e chamada de funções.

Função inicia jogador: Como anteriormente descrito, a função faz a atribuição de algumas variáveis do TAD *Jogador*, e não possui nenhum laço de repetição.

Função criaLista: Esta função faz alocação de memória para uma variável e algumas outras atribuições.

Função caminhoPercorrido: Esta função executa os mesmos tipos de ações da função criaLista.

Função maior_valor: A análise desta será feita em função da variável *tam*, que é o tamanho do vetor que está sendo recebido como parâmetro. A função possui um laço for que é executado *tam-2* vezes, de 1 até *tam-1*. Dentro deste laço, é feita uma comparação, e caso ela seja positiva, é executada uma atribuição para uma variável. Logo, a ordem de complexidade da função é O(n).

Programa principal: A análise desta será feita em função da ordem da matriz do mapa do jogo, que é uma matriz quadrada. O valor da ordem da matriz está sendo armazenado pela variável TAM_MATRIZ. Vamos considerar *n* = *TAM_MATRIZ*. Dividindo em partes:

1. O primeiro laço de repetição é executado n vezes (0 até n-1), dentro dele é feita uma atribuição. Logo, a ordem é O(n).
2. O segundo laço é executado n vezes (0 até n-1) e possui um outro laço, que também é executado n (0 até n-1) vezes. O objetivo destes dois laços é o preenchimento do mapa do jogo. A ordem de complexidade destes é $O(n^2)$.
3. O laço principal da função main, que executa as jogadas dos jogadores, é executado x vezes, sendo x o número de jogadores. Como a análise está sendo feita para *n*, iremos considerar que este laço é executado um número constante de vezes. Dentro do laço, há 4 laços:
    - **for(i=player[jogada].passos; i>0; i--)**

      A variável *i* está recebendo a quantidade de passos disponíveis para o jogador, que é igual a (3* n) - 1. Logo, ele é executado 3n -1 vezes. Dentro deste são executados comandos O(1). Portanto, sua ordem de complexidade é O(n).
    - **for(i=0; i< (3*TAM_MATRIZ-1) - player[jogada].passos; i++)**

      Semelhante ao laço descrito acima, este também é executado 3n -1 vezes e dentro deste também são executadas funções O(1). Portanto, sua ordem de complexidade é O(n).
    - **for(i=0; i<TAM_MATRIZ; i++)**

      Este laço é executado n vezes (0 até n-1), e faz somente uma atribuição a cada vez que ele é executado. Ordem de complexidade O(n).
    - **for(i=0; i<TAM_MATRIZ; i++)** e **for(j=0; j<TAM_MATRIZ; j++)**

      Estes dois laços funcionam da seguinte forma: O laço que percorre de i = 0 até i = n-1 é o laço externo, e o laço que percorre de j = 0 até j = n-1 é o laço interno, e está inserido no laço externo. No laço interno, há somente a chamada da função *fscanf*. A ordem de complexidade do laço externo é $O(n^2)$.

      Somando as ordens de complexidade encontrados nestes quatro laços: $O(n) + O(n) + O(n) + O(n^2) = O(n^2)$.

Como as 3 partes estão separadas e executadas individualmente dentro da função main, para acharmos a complexidade dessa devemos somar a complexidade das 3 partes:

$$O(n) + O(n^2) + O(n^2) = \mathbf{O(n^2)}$$

Logo, a ordem de complexidade da função main é $O(n^2)$.

CONCLUSÃO

O trabalho foi feito baseado no conteúdo discutido em sala e nos slides fornecidos pelo professor. A aplicação de técnicas e métodos foram utilizadas sem grandes problemas. A utilização de TADs se mostrou importante para a organização do código. Sua implementação, juntamente com a manipulação de vetores e listas encadeadas atingiram o objetivo proposto no enunciado do trabalho prático.

BIBLIOGRAFIA

OLMO, Pedro. **Algoritmos e Estruturas de Dados I**. Disponível em: <http://homepages.dcc.ufmg.br/~olmo/AEDS1.html>. Acesso em: 02 out. 2016.