

Relatório de Introdução à Ciência de Computação

Trabalho 5 - Qwirkle

Link para o repositório do GitHub:

https://github.com/lucasfmartins16/Trab_5_ICC

Link para o vídeo com a execução do programa:

https://drive.google.com/file/d/1osUXyd43r_kVBpjKz6-QGiTXQuZdzop/view?usp=sharing

Link para o vídeo com a discussão do programa:

<https://drive.google.com/file/d/1Ho2H8vzArgCD0sQ3OB2soiPXah6fmDZQ/view?usp=sharing>

Ordem dos integrantes: 1º - Lucas Fernandes Martins - nº USP: 11800389

2º - Gabriel Vinícius dos Santos - nº USP: 11819424

Instruções de compilação/execução: Foi comentado no link pertinente do Facebook um link para um repositório do gitHub. Em tal link, é possível baixar todos os arquivos necessários ao programa em formato .zip. Então, deve-se descompactar os arquivos e então executar, respectivamente, os comandos *make all* e *make run*.

Como jogar: o usuário deve escolher o número de jogadores. Caso o número seja menor que 1 ou maior que 18, o usuário deverá inserir o valor. Então, devem ser inseridos os nomes de cada jogador. A partir daí, deve digitar 1 para jogar em modo trapaça e 0 para jogar em modo normal. Caso o usuário digite outra tecla, o usuário deverá digitar novamente.

Discussão da solução: Para implementar as mecânicas do jogo Qwirkle, foi essencial dividir e modular o programa, separando-o nos seguintes arquivos .c de funções:

- Main.c: Contém apenas o menu para selecionar as opções iniciais do jogo e posteriormente entrar no loop do menu de jogo.
- Interface.c: Contém o menu de jogo (que recebe as jogadas), a função para executar uma jogada, uma função para separar palavras de uma string (essencial para o formato de entrada escolhido).
- Dinamica.c: Contém as principais funções que controlam a mecânica do jogo. Por exemplo, devem ser citadas as funções para checar se uma jogada é válida, para calcular os pontos efetuados em uma jogada e para determinar o fim de uma partida.
- Pecas.c: Lá constam as funções para lidar com as peças do jogo, seja para gerar um vetor com todas as peças, seja para escolher randomicamente peças ou até mesmo completar, a cada rodada, o conjunto de peças de cada jogador.

- Tabuleiro.c: Funções referentes ao tabuleiro. Lá estão as funções para redimensionar o tabuleiro depois de uma jogada, para imprimir o tabuleiro, para inicializar o tabuleiro com espaços, etc.

Principais funções e pontos-chave do programa:

- **Struct Peca:** como cada peça é composta por dois chars (uma forma e uma cor), criou-se uma struct Peca que agrupa as informações que definem uma peça. Tal escolha facilitou armazenar as informações e lidar com o tabuleiro, já que, ao invés de criar uma matriz de strings (`***char`), pode-se simplificar o problema criando uma matriz de Peca (`**Peca`). No momento de gerar as peças, a abordagem utilizada também provou seu valor: bastava gerar as combinações de cores e formas e armazená-las num vetor de Peca.

```
typedef struct Peca{
    char forma;
    char cor;
} Peca;
```

- **Redimensionar o tabuleiro:** Para redimensionar o tabuleiro a cada rodada, foi utilizada a função `realloc()`. Caso as linhas ou colunas a serem adicionadas estejam nas bordas inferior ou direita, basta retornar uma matriz com o número atualizado de linhas e colunas. Caso deva-se expandir a matriz nas linha ou coluna 0, é preciso fazer um 'shift' de todos os elementos da matriz. Por exemplo, caso a linha 0 deva ser 'desocupada', em um laço de repetição, os elementos da última linha, recebem os elementos da linha anterior, até deixar a linha 0 vazia.

```
//Realoca linhas
if(l_expand){
    matriz = (Peca **) realloc(matriz, sizeof(Peca*)*(m+l_expand));
    for(i = 0; i < l_expand; i++){
        matriz[m+i] = (Peca*) malloc(sizeof(Peca)*n);
    }
    for(i=0;i<l_expand;i++){
        int j;
        for(j=0;j<n;j++){
            matriz[m+i][j].forma = ' ';
            matriz[m+i][j].cor = ' ';
        }
    }
}
m += l_expand;
//Realoca colunas
if(c_expand){
    for(i = 0; i < m; i++){
```

```

matriz[i] = (Peca *) realloc(matriz[i], sizeof(Peca)*(n+c_expand));
int j;
for(j = 0; j < c_expand; j++){
    matriz[i][n+j].forma = ' ';
    matriz[i][n+j].cor = ' ';
}
}
}
n += c_expand;

```

O código acima exemplifica o processo de realocação com a função `realloc`. Caso as variáveis `l_expand` (linhas a expandir) e `c_expand` (colunas a expandir) não sejam nulas, isso significa que deve ser feito o redimensionamento. Então realoca-se separadamente a nova linha, alocando cada uma de suas colunas, e depois são realocadas as colunas. Essa solução é mais vantajosa do que criar uma nova matriz, uma vez que, em nenhum momento, é preciso ter duas matrizes simultaneamente criadas ocupando memória heap.

- **Entrada:** criou-se uma função que recebe uma string e retorna uma matriz com todas as palavras contidas na string. Isso facilitou o tratamento da entrada, bem como resultou em uma interface gráfica mais amigável: caso o usuário digite um espaço adicional entre uma palavra e outra, o programa ainda segregará corretamente as palavras.

```

char** get_palavras(char *string, int *num){
    //Quantidade de palavras;
    int count_palavra = 0;
    int a = 0;
    int i = 0;
    char** palavra;
    //Aloca dinamicamente uma matriz de chars.
    palavra = (char **) malloc(sizeof(char*)*10);
    if(!palavra){
        printf("Problema de alocacao de memoria!");
        exit(1);
    }
    for(i = 0; i < 10; i++){
        palavra[i] = (char *) malloc(sizeof(char)*20);
        if(!palavra[i]){
            printf("Problema de alocacao de memoria!");
            exit(1);
        }
    }
    //Caso comece com espacos, avanca-se ate a primeira palavra
    while(string[a] == ' '){
        a++;
    }
    //Enquanto a string nao acabar, retiram-se as palavras
    while(string[a] != '\0'){
        i = 0;

```

```

    /*Se o char da posicao a nao for um espaco, coloca-se o caracter em uma
    string da matriz palavra*/
    while(string[a] != ' ' && string[a] != '\0'){
        palavra[count_palavra][i] = string[a];
        i++;
        a++;
    }
    /*
    Se o char da posicao a for um espaco, avanca-se
    */
    palavra[count_palavra][i] = '\0';
    count_palavra++;
    while(string[a] == ' ' && string[a] != '\0'){
        a++;
    }
}
*num = count_palavra;
return palavra;
}

```

- **Repor/Randomizar/Trocar peças:** Para gerar todas as peças, chama-se a função `todas_peças`, que realiza todas as combinações de seis letras e dígitos e retorna um vetor com todos os tipos de peça. Todas as peças tem um número correspondente no vetor de `int usadas`. Quando a peça é utilizada, sua posição em `usadas` recebe 1. Se a peça estiver disponível, o valor correspondente no vetor `usadas` é 0. Assim, Para randomizar uma peça, basta randomizar um número `x` de 0 a 107, ver se a posição `x` do vetor `usadas` é 0, e então retornar a posição `x` do vetor retornado na função `todas_peças()`. O funcionamento da função `todas_peças` pode ser observado abaixo:

```

Peca* todas_pecas(){
    Peca *matriz = (Peca *) malloc(sizeof(Peca)*108);

    int i;

    int letras[6] = {'A', 'B', 'C', 'D', 'E', 'F'};

    int cores[6] = {'1', '2', '3', '4', '5', '6'};

    int j = 0;
    int k = 0;
    int a = 0;

    for(k = 0; k < 3; k++){
        for(i = 0; i < 6; i++){
            for(j = 0; j < 6; j++){
                matriz[a].forma = letras[i];
                matriz[a].cor = cores[j];
                a++;
            }
        }
    }
}

```

```
    }  
  }  
  return matriz;  
}
```

- **Cores:** Para adicionar cores às peças, foi utilizada uma função denominada *cores*, a qual, a partir do “char forma” do typedef struct “Peca” recebido por esta função, utiliza de uma série de prints estratégicos que usam defines para imprimir as cores

2-FUNÇÕES:

2.1 main.c

- **main:** Função primária para receber as entradas de quantidades de jogadores, nome dos jogadores e perguntar se o jogo vai estar no “cheat mode” ou não. Após as entradas serem feitas, chama as funções que vão criar o tabuleiro, enchê-lo com espaços e imprimi-lo. Por fim, entra na função *menu* que iniciará os loops das jogadas.

2.2 dinamica.c

- **check_move:** Segundo as regras do Qwirkle, através de várias *flags*, checa se a jogada é possível, visualizando assim se nenhum erro foi cometido pelo jogador ao jogar na posição indicada. Observa se o espaço é vazio e se na mesma linha ou coluna indicada não há diferença de letra ou número, além de outras checagens mais particulares.
- **points:** analisa as linhas e colunas jogadas pelo jogador e soma o total de pontos feitos numa rodada. Além disso, caso o jogador tenha terminado uma fileira de 6, soma mais 6 pontos indicativos que foi feito um Qwirkle.
- **ver:** analisa se o jogador continua na mesma linha ou coluna em que ele iniciou na primeira jogada.
- **fim_de_jogo:** caso todas as peças tenham acabado, imprime as pontuações finais de cada jogador e finaliza o jogo.

2.3 interface.c

- **get_palavras:** como existem várias opções do que fazer na rodada (trocar, jogar, passar), analisa a string recebida como entrada pelo jogador e a separa em um vetor de strings (matriz de chars) contendo as diversas palavras da frase de entrada, além de indicar o número de palavras que foram lidas da entrada.

- **???jogada:**
- **menu:** interface principal entre os jogadores e o programa. Representado por um loop de *while*. Ao ser iniciada, cria diversas flags que serão essenciais para o funcionamento do programa, além de um vetor *usadas[108]* de inteiros iniciado somente com “zeros” que indica quais peças já foram retiradas e quando uma for retirada é colado o número “um”; este vetor funciona em conjunto com outras funções explicadas posteriormente. Também é criada a matriz *pecas* de dimensões número de jogadores x 6 que guarda as peças que cada jogador tem consigo, semelhante a um jogo de verdade.
- **jogada_invalida:** quando for chamada, imprime uma mensagem de erro.

2.4 pecas.c

- **todas_pecas:** quando chamada, cria um vetor de *Peca* (explicado na parte typedef struct) chamada *matriz[108]* contendo todas as combinações possíveis de peças. Começa em A1,A2,A3,A4,A5,A6,B1,B2... até F6 e repete este ciclo mais duas vezes somando 108 peças. Retorna *matriz[108]* para ser usado em outras funções.
- **retornar_peca:** quando chamada, executa o papel de trocar peças, as colocando de volta no monte com todas peças, sendo assim, usando o vetor *matriz[108]* criado na função *todas_pecas*, analisa onde a peça em questão está posicionada achando a posição correta, e usando essa posição, altera o valor da posição achada no vetor *usadas[108]* criado na função *menu* deixando seu valor como “zero”, indicando que aquela peça agora pode ser retirada.
- **rand_peca:** através da função *srand(time(NULL))* e *a=rand()%108*, sorteia uma posição randomicamente entre “zero” e “cento e sete” e compara com o vetor *usadas[108]* criado na função *menu* e caso a posição esteja disponível, retorna essa posição, e caso não esteja, repete o processo de sorteio até achar uma posição disponível.
- **print_pecas:** em cada rodada, imprime as peças de cada jogador usando a matriz *pecas* criada na função *menu*.
- **puxar_pecas:** no começo do jogo e após cada rodada, reenche a mão do jogador da rodada (ou de todos os jogadores se for no começo do jogo) analisando se a matriz *pecas* criada na função *menu* contém alguma posição vazia e caso tiver, usa a função *rand_peca* colocando assim uma peça aleatória.

2.5 tabuleiro.c

- **aloca_tabuleiro:** no começo do jogo, usa alocação dinâmica para criar uma matriz que será a responsável por armazenar as peças do tabuleiro. No começo do jogo,

ela é criada de dimensões 1 x 1, mas conforme o jogo for avançando, as dimensões são aumentadas através de *reallocs* encontrados na função *update_tabuleiro* (explicada posteriormente).

- **print_tabuleiro**: recebe as dimensões do tabuleiro e a matriz criada no *aloca_tabuleiro* contendo as peças e imprime o tabuleiro seguindo o espaçamento correto, com os devidos divisores e chamando a função *cores* em cada peça impressa.
- **cores**: chamada dentro da função *print_tabuleiro*, é utilizada para mudar as cores das peças para facilitar a identificação pelo jogador. Cada cor é referente ao número de cada peça.
- **fill_tabuleiros**: chamada após a criação da matriz que guarda as peças do tabuleiro, enche essa matriz com o caractere arbitrário "c".
- **update_tabuleiro**: Após uma jogada, utiliza condições para analisar onde a jogada foi feita, e caso esta foi feita numa das bordas do tabuleiro, redimensiona o tabuleiro (representado pela matriz criada na função *aloca_tabuleiro*) usando *reallocs*, aumentando assim o número de linhas e colunas e fazendo os devidos ajustes das posições das peças que já foram jogadas.