Documentação Trabalho Prático 1

Lucas Roberto Santos Avelar

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG) - Belo Horizonte – MG – Brasil

lucasrsavelar@ufmg.br

1. Introdução

Esta documentação lida com o problema da simulação de um jogo de Poker. O objetivo é implementar um programa que receba como entrada um arquivo de texto contendo as informações necessárias para o desenvolvimento de diversas rodadas do jogo (como o valor do pingo, as cartas de cada jogador, entre outros), analise essas entradas e retorne o vencedor de cada rodada, bem como o resultado final após todas as rodadas. Para resolver o problema, foi escrito um código que realizava as operações e análises necessárias a fim de determinar qual jogador ganhou e quanto ele ganhou a cada rodada, imprimia seu nome, valor ganho e qual foi sua jogada no arquivo de saída e ao final imprimia o resultado ordenado de cada jogador.

2. Implementação

O código organiza-se principalmente em 2 blocos. Dentro dos arquivos, encontram-se alguns comentários sobre as operações de cada função. A seguir, há uma análise mais detalhada.

O primeiro bloco consiste nos arquivos “funcoes.hpp” e “funcoes.cpp” que reúnem, respectivamente, a declaração e a implementação das diversas funções fundamentais para a execução do programa, bem como da declaração e implementação de uma estrutura “Jogador” a partir da qual é criada um array que reúne as informações dos jogadores das partidas. A estrutura Jogador possui seu nome, mão, combo (jogada), dinheiro e aposta atual, além de variáveis que armazenam inteiros relacionados às suas cartas. A função maiorCartaCombo recebe a mão e o combo de um jogador e retorna a maior carta desse combo. Para isso, ela cria um vetor zerado de 14 posições e coloca uma unidade em cada posição desse vetor

correspondente ao valor da carta (se a carta é um 10, coloca uma unidade na décima posição do vetor). De acordo com o combo passado como parâmetro, a função percorre esse vetor procurando qual o maior índice preenchido, ou seja, qual a maior carta, e a retorna. A função maiorCartaForaCombo funciona de modo semelhante, recebendo os mesmos parâmetros e criando o mesmo vetor, com a diferença que ela retorna a maior carta que não participa da jogada. A função segundoPar é exclusiva para auxiliar no desempate de combos TP (“Two Pairs”), e também funciona de forma semelhante às últimas duas funções. A função analisaMao é a mais importante do programa, pois é a que verifica qual a maior jogada de uma mão passada como parâmetro. Primeiro, ela cria o mesmo vetor das funções anteriores, mas também um array de strings que recebe os naipes das cartas, e percorre o vetor de cartas contando o número de pares, trincas e quadras. Depois, verifica se os naipes são todos iguais ou não. Caso sejam e caso as cartas sejam uma sequência do 10 ao Às, ela retorna um RSF. Depois, ela verifica se as cartas formam uma sequência, e com essas informações, determina qual a maior jogada da mão. Se, por exemplo, as cartas formam uma sequência e os naipes são todos iguais, mas não é um RSF, então ela retorna um SF. Se houver uma quadra, ela retorna um FK, e assim por diante. A função atribuiID recebe um combo como parâmetro e atribui a ele um valor inteiro, que será usado na comparação entre jogadas. A função determinarVencedor recebe um vetor de jogadores e a quantidade de participantes do jogo e retorna um inteiro, que pode ser o índice do jogador vencedor ou o ID do combo vencedor. Primeiro, ela percorre o vetor de jogadores buscando o combo com maior ID, e armazena esse ID. Depois, ela percorre o vetor novamente procurando jogadores que possam ter o mesmo combo do vencedor. Caso haja apenas um vencedor, ela já retorna o índice do vencedor. Se não, inicia-se um bloco de comparações. Se o combo vencedor não for um Two Pairs, compara-se o valor da maior carta do combo do vencedor com o valor da maior carta de outro jogador com o mesmo combo. Permanecendo o empate, as maiores cartas fora da jogada são comparadas. Se o empate persistir, retorna-se o ID do combo vencedor; caso contrário retorna-se o índice do vencedor. Para o caso de um Two Pairs, o mesmo acontece, com a diferença de que o segundo maior par também é comparado. A função resetaJogadores recebe como parâmetros um array de jogadores e o número de participantes, e percorre esse array colocando cada jogador de volta em seu estado inicial (zera a aposta atual, remove a mão e o combo, entre outros). Finalmente, a função ordenaJogadores recebe os mesmos

parâmetros da função anterior e reordena o array de jogadores em ordem decrescente de dinheiro.

Já o segundo bloco corresponde ao arquivo “main.cpp”, onde está localizada a função main do programa. Primeiro, ela declara e inicializa as variáveis necessárias para o funcionamento do programa, bem como declara e abre os arquivos de entrada e saída. Do arquivo de entrada, lê-se o número de rodadas e o dinheiro inicial de cada jogador, assim como o número de participantes e o valor do pingo da primeira rodada. A função main divide a primeira rodada da demais pois é necessário criar e inicializar o array de jogadores na primeira rodada. Depois, ela percorre o vetor de jogadores atribuindo a cada um seu nome, dinheiro inicial, aposta e cartas, e percorre novamente o array deduzindo o valor da aposta e do pingo do dinheiro de cada jogador e adicionando esse montante ao pote (caso não haja falha em teste de sanidade. Se algum jogador não tiver dinheiro para o pingo/aposta ou se a aposta não for múltipla de 50, a rodada é invalidada). Cada jogador, então, recebe seu combo (pela função analisaMao), o ID do combo (pela função atribuiID), as maiores cartas dentro e fora do combo (pelas funções maiorCartaCombo e maiorCartaForaCombo), e a maior carta do segundo combo, caso necessário. A função determinaVencedor, então, analisa qual jogador foi vitorioso. Caso haja apenas um vencedor, seu índice é retornado e ele é impresso no arquivo de saída em conjunto com as informações requeridas, além de receber o valor do pote. Se há mais de um vencedor, o ID do combo vencedor é retornado e os jogadores com esse combo são impressos na saída, enquanto o pote é distribuído entre eles. O programa então entra em um laço que realiza as mesmas operações com as rodadas restantes, porém lembrando de verificar qual jogador corresponde ao nome da entrada. Finalmente, a função ordenaJogadores realiza a reorganização do array de jogadores em ordem decrescente de dinheiro, os resultados finais são impressos na saída e os arquivos de entrada e saída são fechados.

O programa foi testado em um computador com 4GB de memória RAM e processador intel Core i3, sistema operacional Windows 10 (mas a partir de um subsistema WSL Ubuntu), utilizando da linguagem C++ e com o compilador g++.

3. Análise de Complexidade

Complexidade de Tempo: dentro do arquivo “funcoes.cpp”, temos que as funções maiorCartaCombo, maiorCartaForaCombo, segundoPar e analisaMao apresentam laços com um número fixo de iterações (do 0 ao 13), logo possuem uma complexidade constante O(14). A função atribuiID apresenta apenas operações condicionais e de retorno, possuindo complexidade O(1). As funções determinarVencedor e resetaJogadores apresentam laços que iteram por todo o array de jogadores, ou seja, possuem complexidade que depende do tamanho da entrada, logo O(n). Já a função ordenaJogadores apresenta dois laços aninhados que iteram pelo tamanho do array de jogadores, possuindo então complexidade $O(n^2)$.

Dentro do arquivo “main.cpp”, a maior complexidade de tempo reside no laço que trabalha com as rodadas seguintes à primeira, visto que dentro dele estão as diversas chamadas para as funções analisadas acima. Dado que esse laço itera por n – 1 rodadas (= O(n)) e que a chamada com maior complexidade é dada por $O(n^2)$, a função main do programa possui complexidade $O(n^3)$.

Complexidade de Espaço: as funções realizam as operações levando em conta um array de jogadores alocado de forma estática e que possui tamanho n determinado pelo usuário através do arquivo de entrada. Além disso, cada jogador exige uma quantidade fixa de espaço dada por c para armazenar os atributos da estrutura Jogador. Sendo assim, o programa apresenta uma complexidade de espaço dada por n * O(c) = O(c).

4. Estratégias de Robustez

Assim como recomendado na discussão do fórum de dúvidas do Trabalho Prático 1, as estratégias de robustez para esse programa envolvem tratar os três seguintes casos, que são os teste de sanidade: jogador com dinheiro insuficiente para o pingo, jogador com dinheiro insuficiente para realizar sua aposta, ou jogador realizando aposta com um valor que não é múltiplo de 50. Para todos os casos, o tratamento recomendado no mesmo fórum foi considerar a rodada como inválida, o que foi implementado no programa e descrito na seção de Implementação.

5. Testes

Os testes foram realizados com auxílio do gerador de carga disponibilizado no fórum do Trabalho Prático 1, além da entrada dada como exemplo no enunciado do TP1. Foram criados arquivos de texto que seguem o modelo da entrada, com número de jogadores e rodadas variados, e esses arquivos foram utilizados como entrada para a execução do programa.

6. Análise Experimental

A análise de Desempenho Computacional foi efetuada após submeter o programa a entradas com variados números de jogadores e de rodadas. O resultado obtido foi que, mesmo alterando esses valores para quantidades relativamente elevadas, o programa executava em um espaço de tempo extremamente curto. Mesmo em situações com 10 jogadores e mais de 60 rodadas, em que algumas funções eram chamadas mais de 200, 400 e até mesmo 8000 vezes, o gprof acusou que cada função utilizou uma quantidade mínima de tempo. A ausência de qualquer procedimento dinâmico, assim como a quase inexistência de ponteiros e ferramentas que utilizem de espaços extras da memória é a principal causa desse baixo tempo de execução.

7. Conclusões

O trabalho consistiu na implementação de um programa que simula um jogo de Poker, que dado um arquivo de entrada com informações sobre os participantes, rodadas e outros elementos essenciais para o jogo, cria um arquivo de saída contendo os vencedores de cada rodada e o resultado final do jogo. O Trabalho Prático 1 pode ser considerado como extremamente desafiador e como a atividade mais difícil do semestre até agora, principalmente pelo fato de que tivemos que implementar todo o programa praticamente do zero, com pouquíssimo ou nenhum material que pudesse ser utilizado como base ou suporte, um salto bem grande com relação ao Trabalho Prático 0 (que consistia basicamente em reaproveitamento de código) e até mesmo com relação aos TPs de outras disciplinas, tanto desse quanto de outros semestres. Foi necessário pensar bastante sobre qual abordagem seria tomada para resolver os diversos problemas (como ler as entradas, como analisar as mãos dos jogadores, como realizar o desempate, entre outros), com a parte criativa sendo bastante exercitada para pensar nas soluções. Não acredito que exista alguma parte mais ou menos complicada que outra;

todas as etapas da implementação do programa possuíram algum grau de dificuldade e todas exigiram bastante raciocínio. Buscar ideias e inspiração sobre modos de resolver cada dificuldade foi o maior desafio do trabalho.

8. Bibliografia

- Material disponibilizado no minhaUFMG
- https://gamedev.stackexchange.com/questions/49302/determining-poker-hands
- https://cplusplus.com/doc/tutorial/files/
- https://www.guru99.com/cpp-file-read-write-open.html

9. Instruções para compilar e executar

- Antes de tudo, acesse o diretório “bin”, presente dentro do diretório TP, e coloque nele o arquivo de texto “entrada.txt”
- Retorne ao diretório raiz TP
- Escreva no terminal o comando “make”
- Acesse novamente o diretório “bin”
- Escreva no terminal o comando “./tp1.exe”
- O arquivo “saida.txt” será gerado dentro do diretório “bin”
- Para novas entradas, acesse o diretório “bin”, edite o arquivo “entrada.txt” e execute novamente no terminal o comando “./tp1.exe”