Documentação do Trabalho Prático 1 da disciplina de Estrutura de Dados

Marco Tulio Tristão

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brasil

marcotufmg@gmail.com

1. Introdução

Este documento explica a resolução do problema proposto para o Trabalho Prático 1. O problema em questão é implementar um programa que simula partidas de poker. Dado os jogadores, suas apostas e suas cartas, o programa deve computar os ganhadores de cada rodada e o dinheiro que eles acumularam das apostas. Além disso, ao final da partida, os jogadores devem ser imprimidos em ordem decrescente do dinheiro que eles acumularam. Como todos os jogadores começam com a mesma quantia monetária, o seu montante no fim da partida será o indicativo de quem foi mais vitorioso.

Para resolver essa questão, foi necessário exercitar os conhecimentos de estruturas de dados, programação orientada a objetos, análise de complexidade, entre outros na linguagem C++.

2. Método

O trabalho em questão foi projetado na linguagem C++ e compilado pelo GNU C++ Compiler (g++). O Sistema Operacional utilizado foi o Ubuntu-20.04 por meio do Windows Subsystem for Linux 2 (WSL 2), funcionando em uma máquina com o processador Intel Core i5-7300 e 8 GB de memória RAM.

2.1. Estrutura de Dados e Tipos Abstratos de Dados (TADs)

O programa conta com dois TADs: *player* e *Hand*. O TAD *player* representa os jogadores da partida de poker e contém o seu nome, o dinheiro, a aposta da rodada e a mão do jogador, que é uma instância do TAD *Hand*. Ele está localizado em *player.cpp* e possui sua declaração em *player.hpp*.

O TAD *Hand*, por sua vez, conta com três atributos: um vetor de inteiros que representa as cartas e seus respectivos números e naipes, o valor da mão do jogador (qual a sua jogada) e um vetor de três inteiros, que representam os critérios de desempate. Esse TAD é implementado no arquivo *Hand.cpp* e declarado em *Hand.hpp*.

2.2. Funções Relevantes

Exceto a função *main*, que está localizada no arquivo *main.cpp*, e é responsável por abrir os arquivos de entrada e saída e por iniciar cada rodada do jogo, as funções que auxiliam o desenvolvimento das rodadas estão localizadas em *functions.cpp*, que possui seu cabeçalho em *functions.hpp*. Além disso, foram criadas diversas funções para tratamento de erro, que podem ser encontradas em *erro.hpp* (implementação em *erro.cpp*).

As principais funções no andamento da rodada são: *rodada()*, parse_input e *play()*. A função *rodada()* é responsável por fazer a chamada tanto para *parse_input()* (que lerá os dados de cada jogador), quanto para *play()*. Esta última, junto com suas funções auxiliares, irá encontrar a maior jogada da rodada, os jogadores que a jogaram e fará os devidos desempates e distribuições do montante de apostas. Após isso, é impresso os vencedores da rodada.

3. Análise de Complexidade

4.1. Tempo

Todas os métodos tanto da classe *player*, quanto da classe *Hand*, possuem tempo de execução constante. Apesar de haver loops nos métodos de *Hand*, a duração desses depende somente do tamanho da mão do jogador, que sempre contém 5 cartas. Logo, a complexidade de tempo desses métodos é O(1).

As funções de execução da partida possuem diferentes complexidades. Para isso considere n o número de jogadores. A função *play* possui chamada para três funções: *better_hand* (O(n)), *create_round_bet* (O(n)) e *desempata* (O(1)). Nessa função, também existem dois laços aninhados, cada um com limite no número de jogadores na rodada e com um tempo de execução constante. No pior caso, esses laços terão tempo de execução quadrático, fazendo, então *play* ser uma função de complexidade O(n^2).

As demais funções também possuem sua complexidade ditada pelo comprimento dos seus laços. Assim, *print_winner* e parse_input são lineares (O(n)), pois possuem um laço até n, e *print_result* e *rodada* são quadráticas (O(n^2)), a primeira porque possui dois laços aninhados e a segunda porque possuem um laço de tamanho n e tempo de execução linear devido a uma chamada para *parse_input*.

Portanto, o programa como um todo possui uma complexidade de O(m * n^2), sendo m o número de rodadas e n o número de jogadores, pois a cada rodada se gasta O(n^2) para executá-la.

4.1. Espaço

O programa possui uma complexidade linear no espaço, ou seja, O(n), sendo n o espaço gasto por n jogadores (n representando também o número de jogadores). Isso, porque no começo da execução do programa é alocada memória para n jogadores. Após isso, o uso da memória ocorre de maneira constante ou linear, como na alocação de memória para o vetor *participantes* na função *rodada* (O(n)).

4. Análise Experimental

Foram usados *msgassert.h*, *memlog.c* e *analisamem.c*, disponibilizadas e desenvolvidas pelo professor Wagner Meira, para o propósito da análise experimental do programa. Os testes usados para a geração dos gráficos da análise estão localizados na parte 6 da documentação.

4.1. Análise Computacional

Infelizmente, a ferramenta usada gprof não conseguiu contabilizar o tempo de execução do programa. No entanto, ainda é possível notar o número de chamadas feitas em cada função. A primeira análise foi feita com uma pequena entrada de somente uma rodada. Percebe-se que há um grande número de chamada de funções do tipo get e do tipo set. No geral, essas funções tem uma complexidade O(1) de execução, portanto não devem causar grande impacto no tempo total do programa.

% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
0.00	0.00	0.00	30	0.00	0.00	player::get_name()
0.00	0.00	0.00	27	0.00	0.00	player::get_money()
0.00	0.00	0.00	15	0.00	0.00	player::set_money(int)
0.00	0.00	0.00	6	0.00	0.00	Hand::get_tb(int*)
0.00	0.00	0.00	6	0.00	0.00	Hand::set_tb()
0.00	0.00	0.00	6	0.00	0.00	Hand::get_value()
0.00	0.00	0.00	6	0.00	0.00	player::get_bet()
0.00	0.00	0.00	3	0.00	0.00	parse_input(int, player*)
0.00	0.00	0.00	3	0.00	0.00	erro_bet()

0.00	0.00	0.00	3	0.00	0.00	desempata(int const&, int const&, player*)
0.00	0.00	0.00	3	0.00	0.00	Hand::check_suits()
0.00	0.00	0.00	3	0.00	0.00	Hand::check_numbers()
0.00	0.00	0.00	3	0.00	0.00	Hand::check_sequence()
0.00	0.00	0.00	3	0.00	0.00	Hand::different_numbers()
0.00	0.00	0.00	3	0.00	0.00	Hand::sort()
0.00	0.00	0.00	3	0.00	0.00	Hand::set_card()
0.00	0.00	0.00	3	0.00	0.00	Hand::set_value()
0.00	0.00	0.00	3	0.00	0.00	Hand::Hand()
0.00	0.00	0.00	3	0.00	0.00	player::set_bet(int)
0.00	0.00	0.00	3	0.00	0.00	player::set_hand()
0.00	0.00	0.00	3	0.00	0.00	player::set_name(std::string)
0.00	0.00	0.00	1	0.00	0.00	ativaMemLog()
0.00	0.00	0.00	1	0.00	0.00	better_hand(bool*, player*)
0.00	0.00	0.00	1	0.00	0.00	iniciaMemLog(char*)
0.00	0.00	0.00	1	0.00	0.00	print_result(player*)
0.00	0.00	0.00	1	0.00	0.00	print_winner(int, int*, player*)
0.00	0.00	0.00	1	0.00	0.00	finalizaMemLog()
0.00	0.00	0.00	1	0.00	0.00	create_round_bet(bool*, int const&, player*)
0.00	0.00	0.00	1	0.00	0.00	play(bool*, int const&, int const&, player*)
0.00	0.00	0.00	1	0.00	0.00	rodada(int const&, int const&, player*)
0.00	0.00	0.00	1	0.00	0.00	erro_arg(char const*, int)

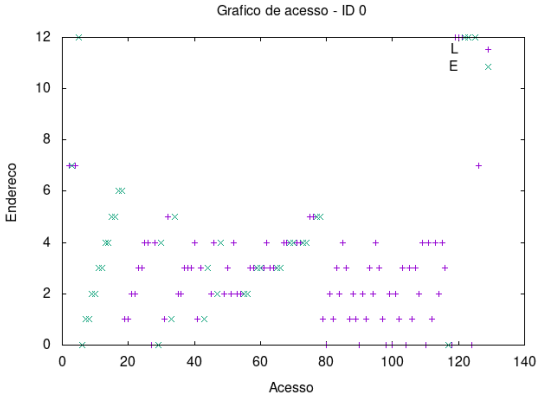
Essa segunda análise mostra que, com aumento da entrada, aumenta também o numero de chamada das funções. Ainda existe o predomínio de funções O(1), mas o número de chamadas a funções mais complexas começa a aumentar linearmente com o número de rodadas.

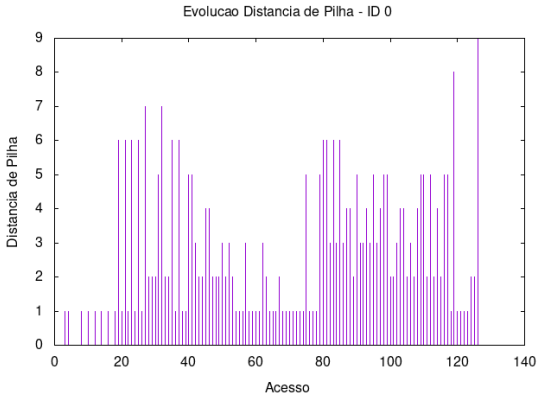
% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
0.00	0.00	0.00	963	0.00	0.00	player::get_name()
0.00	0.00	0.00	454	0.00	0.00	player::set_money(int)
0.00	0.00	0.00	384	0.00	0.00	Hand::get_value()
0.00	0.00	0.00	384	0.00	0.00	player::get_bet()
0.00	0.00	0.00	216	0.00	0.00	player::get_money()
0.00	0.00	0.00	192	0.00	0.00	parse_input(int, player*)
0.00	0.00	0.00	192	0.00	0.00	erro_bet()
0.00	0.00	0.00	192	0.00	0.00	Hand::check_suits()
0.00	0.00	0.00	192	0.00	0.00	Hand::check_numbers()
0.00	0.00	0.00	192	0.00	0.00	Hand::check_sequence()
0.00	0.00	0.00	192	0.00	0.00	Hand::different_numbers()
0.00	0.00	0.00	192	0.00	0.00	Hand::sort()
0.00	0.00	0.00	192	0.00	0.00	Hand::set_card()
0.00	0.00	0.00	192	0.00	0.00	Hand::set_value()
0.00	0.00	0.00	192	0.00	0.00	player::set_bet(int)
0.00	0.00	0.00	192	0.00	0.00	player::set_hand()
0.00	0.00	0.00	128	0.00	0.00	Hand::get_tb(int*)
0.00	0.00	0.00	128	0.00	0.00	Hand::set_tb()
0.00	0.00	0.00	64	0.00	0.00	better_hand(bool*, player*)
0.00	0.00	0.00	64	0.00	0.00	print_winner(int, int*, player*)
0.00	0.00	0.00	64	0.00	0.00	create_round_bet(bool*, int const&, player*)
0.00	0.00	0.00	64	0.00	0.00	play(bool*, int const&, int const&, player*)
0.00	0.00	0.00	64	0.00	0.00	rodada(int const&, int const&, player*)
0.00	0.00	0.00	64	0.00	0.00	desempata(int const&, int const&, player*)
0.00	0.00	0.00	3	0.00	0.00	Hand::Hand()

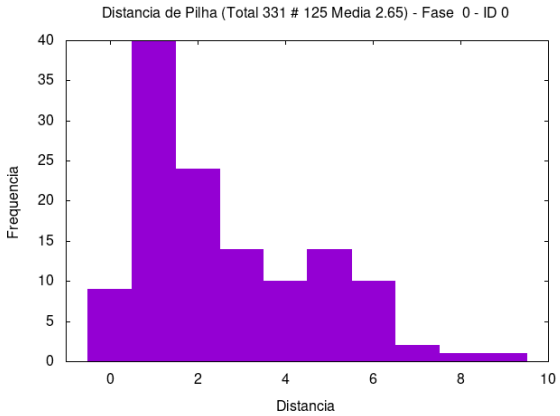
0.00	0.00	0.00	3	0.00	0.00	player::set_name()
0.00	0.00	0.00	3	0.00	0.00	player::player()
0.00	0.00	0.00	3	0.00	0.00	player::~player()
0.00	0.00	0.00	1	0.00	0.00	print_result(player*)
0.00	0.00	0.00	1	0.00	0.00	erro_arg(char const*, int)

4.2.Gráficos de acesso à memória

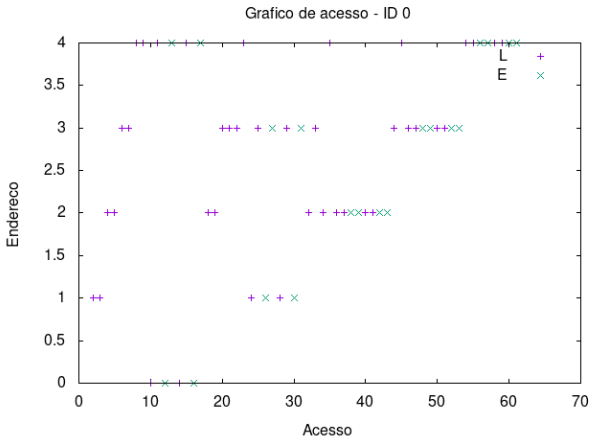
- Leitura de dados do primeiro jogador do teste 2.

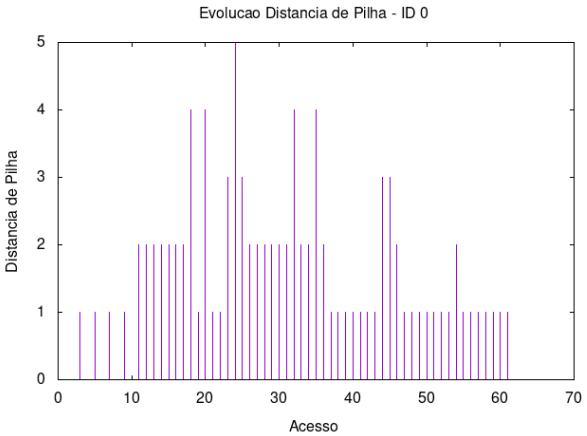


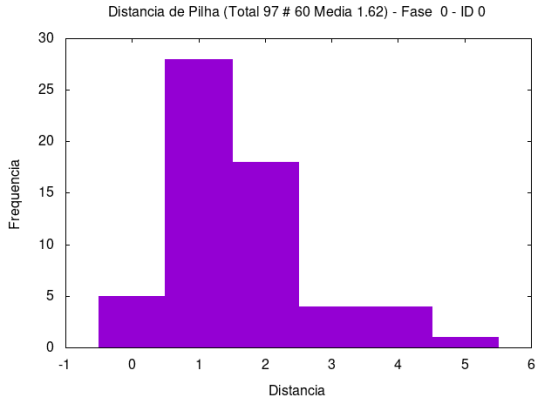




- Ordenação de uma mão inicialmente em ordem decrescente.



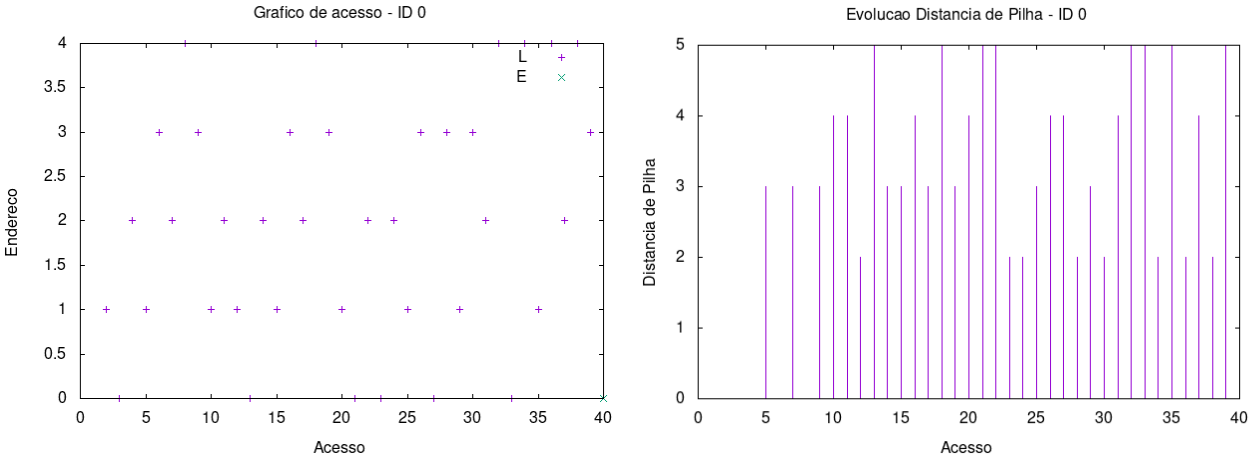


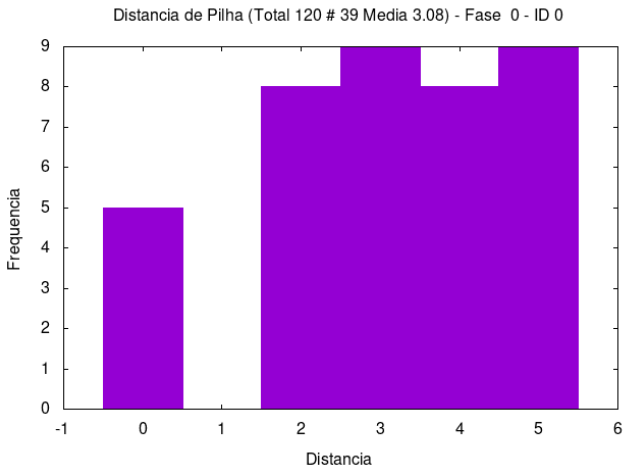


Pode-se ver que o algoritmo de ordenação usado não é eficiente no acesso a endereços na memória, já que os acessos não são usualmente em endereços contínuos. Esse algoritmo só pode ser usado devido ao tamanho do vetor a ser ordenado ser muito pequeno.

Por esses gráficos, principalmente pelo mapa de acesso, também é possível perceber claramente as mudanças entre as fases do algoritmo (a passagem dos loops).

- Definição do valor de uma mão.





- Leitura dos dados de todos os jogadores do teste 2 na primeira rodada.

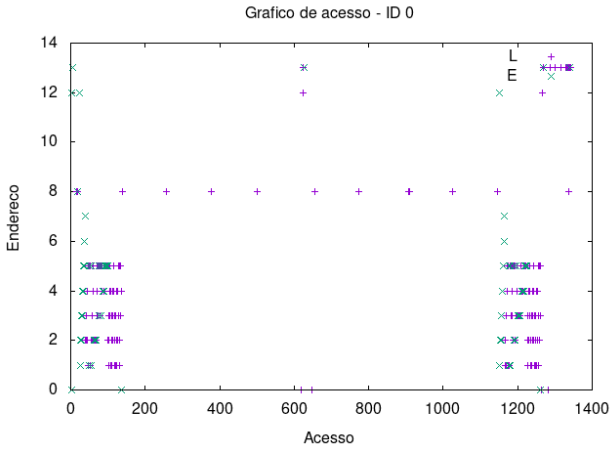
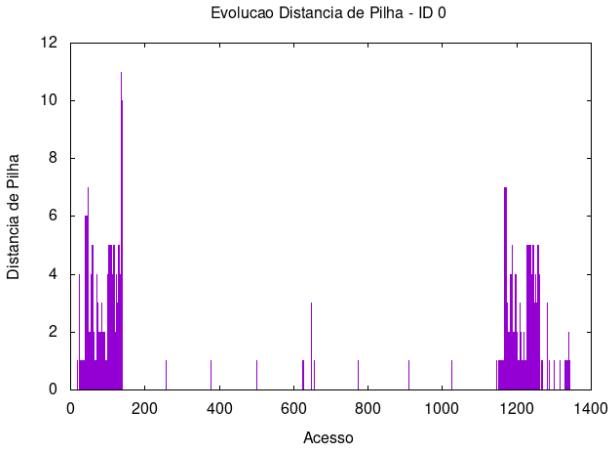
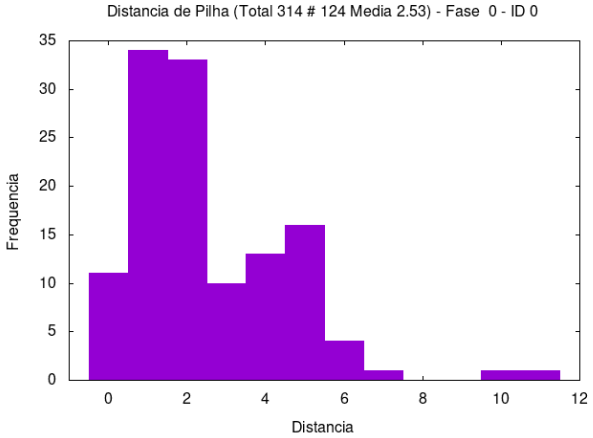
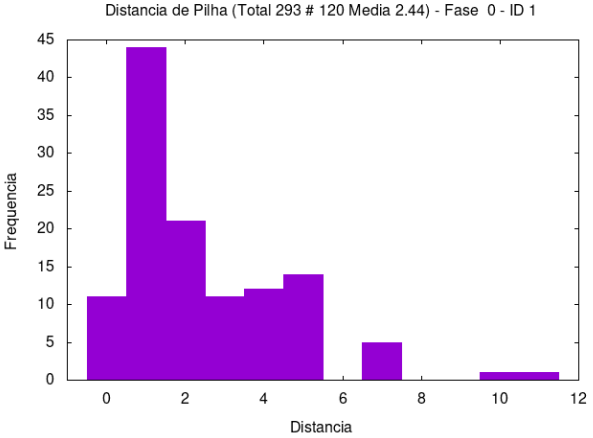
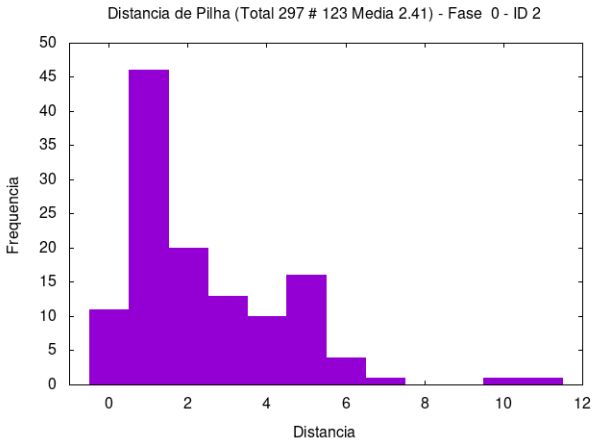


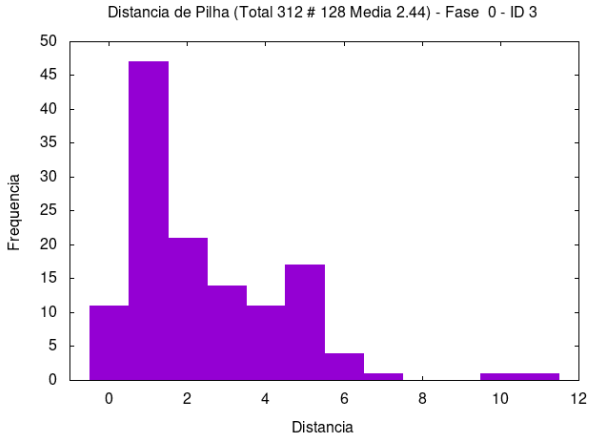
Grafico de acesso - ID 0
Endereco
Acesso
L
E

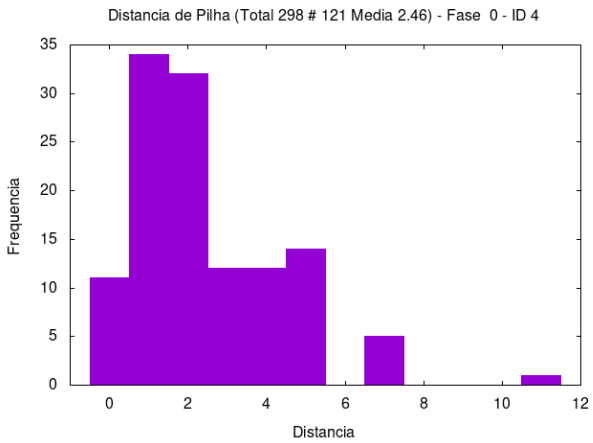

Evolucao Distancia de Pilha - ID 0
Distancia de Pilha
Acesso


Distancia de Pilha (Total 314 # 124 Media 2.53) - Fase 0 - ID 0
Frequencia
Distancia


Distancia de Pilha (Total 293 # 120 Media 2.44) - Fase 0 - ID 1
Frequencia
Distancia

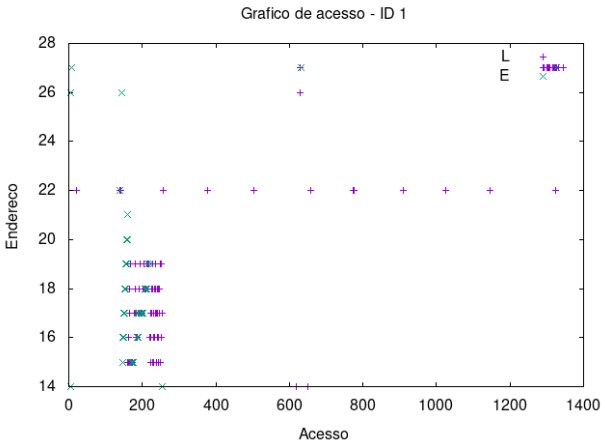

Distancia de Pilha (Total 297 # 123 Media 2.41) - Fase 0 - ID 2
Frequencia
Distancia

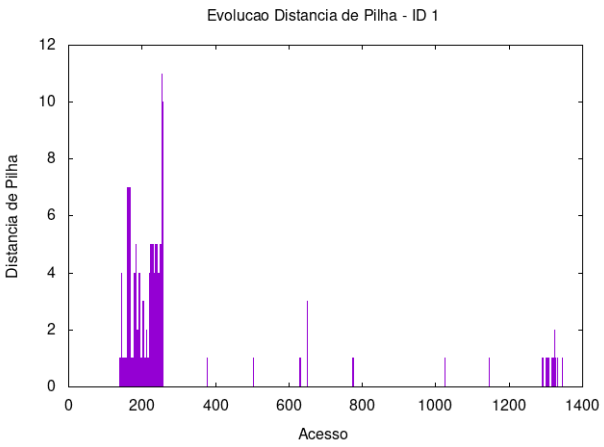

Distancia de Pilha (Total 312 # 128 Media 2.44) - Fase 0 - ID 3
Frequencia
Distancia

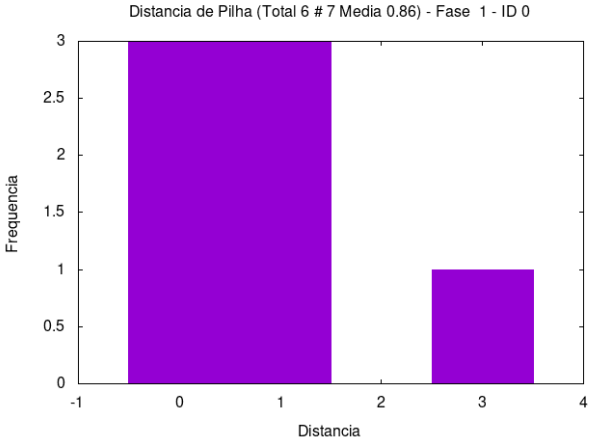

Distancia de Pilha (Total 298 # 121 Media 2.46) - Fase 0 - ID 4
Frequencia
Distancia

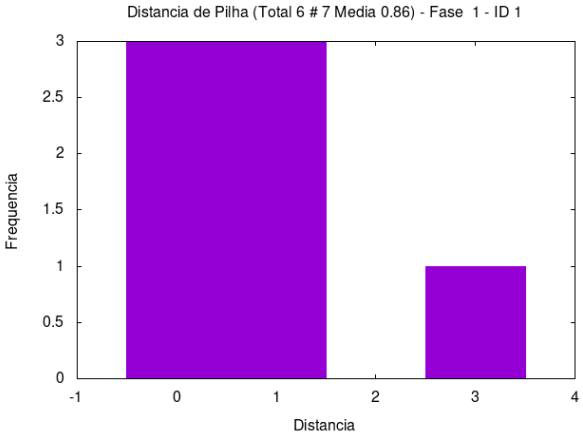
Com o aumento da escala dos gráficos, é perceptível que há uma concentração dos endereços acessados, como se pode perceber pelos gráficos de distância de pilha, que se concentra na média de 2.5.

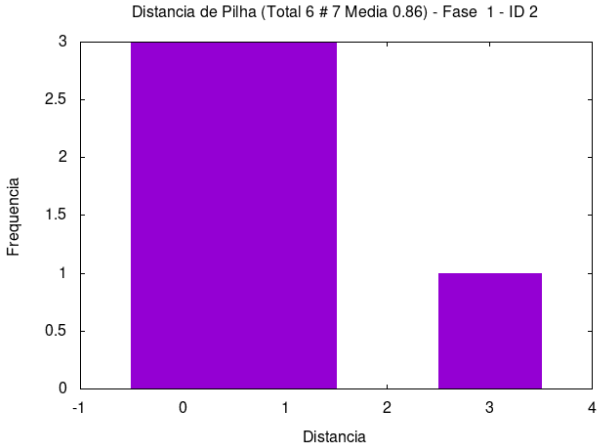
- <u>Andamento da primeira rodada do teste 2 após a leitura dos dados.</u>

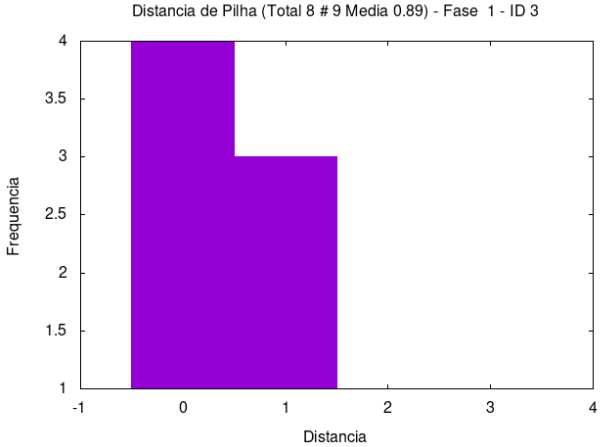


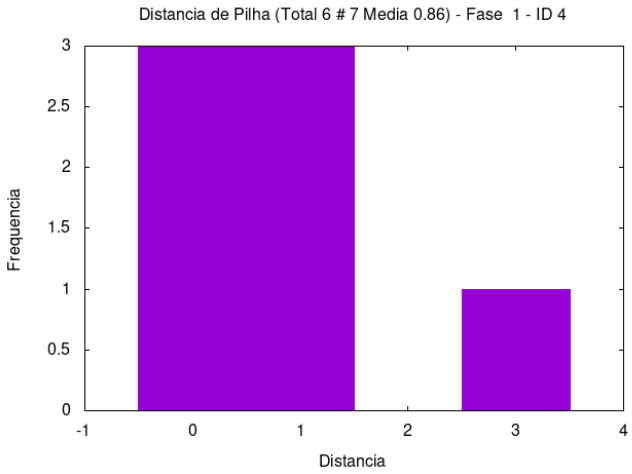












Percebe uma grande consistência e concentração do acesso aos endereços de memória em cada jogador, como mostra os gráficos de distância de pilha, que revelam uma distância média de 0.85. Além disso, também é possível notar no mapa de acesso a concentração dos acessos no começo do eixo de acesso. Essa concentração é resultado da primeira fase do jogo, onde se acessa o valor da mão e das apostas dos jogadores para descobrir os vencedores.

6. Exemplos de entrada e saída

O programa foi testado tanto para entrada mais simples, quanto para outras maiores e mais complexas. Os testes menores foram feitos para mostrar com mais clareza o correto funcionamento do programa para casos ordinários, para o tratamento de empates e rodadas inválidas e para casos com jogadores que possuem nomes compostos.

6.1. Teste 1

Entradas	Saídas
3 1000 3 50 Afonso 100 8C 8P 7E 3O 1E Ednilson 150 8C 3C 1P 9E 13E Betina 50 3C 1P 7C 7E 9P 2 50 Ednilson 150 9E 10E 11E 12E 13E Betina 100 9C 10C 11C 12C 13C 1 50 Afonso 10 9E 8E 7C 13P 4C	1 450 OP Afonso 2 200 S Betina Ednilson 0 0 I #### Afonso 1300 Betina 950 Ednilson 800

6.2. Teste 2

Entradas	Saídas
3 1000 5 50 Giovanni 100 6O 3P 10E 11O 1O John 200 3P 4E 3E 13C 13O Thiago 100 12O 7P 12C 1O 13C Gisele 300 12E 10C 11C 9C 13E Wagner 50 5P 12P 5E 2E 1P 2 50 Wagner 200 2P 13E 9E 12C 2O	1 1000 S Gisele 1 800 OP Wagner 1 1000 F Gisele #### Gisele 2050

Gisele 350 11P 9P 2E 6E 4P 3 100 Thiago 250 1O 4P 1E 3O 8O Gisele 100 9C 8C 8P 2C 6C Giovanni 150 4P 12P 8E 12E 2P	Wagner 1450 John 750 Giovanni 600 Thiago 500

6.3. Teste 3

Entradas	Saídas
1 1000 2 50 Adamastor Jacaré 100 8E 8C 8P 8O 7C Ednilson Pereira Marcos 150 9C 10E 3E 9P	1 350 FK Adamastor Jacaré #### Adamastor Jacaré 1200 Ednilson Pereira Marcos 800

6.4. Teste 4

Entradas	Saídas
1 1000 2 50 Lionel Messi 900 8C 8E 10C 10E 3P Chaves 950 10P 10O 8P 8O 2E 2 50 Lionel Messi 900 8C 8E 10C 10E 3P Chaves 950 10P 10O 8P 8O 2E	1 1950 TP Lionel Messi 0 0 I #### Lionel Messi 2000 Chaves 0

7. Estratégias de Robustez

Como pode-se ver no arquivo *erro.hpp*, foram criadas uma série de funções que objetivam lidar com diversos erros que podem vir a acontecer no decorrer do programa. Alguns desses erros são: a chamada do programa ser feita de maneira incorreta no terminal, não ser possível abrir o arquivo de entrada corretamente, a aposta feita por um jogador ser inválida e o número de jogadores em uma determinada rodada ser inválido. Além de enviar uma mensagem de erro, alguns desses erros podem acarretar o término do programa, enquanto outros, como o erro de aposta inválida, somente causam o lançamento de um alerta e a invalidez de um loop no código.

8. Conclusões

O trabalho em questão consistia em captar entradas de dados e por meio destas descobrir o vencedor de uma partida de poker online. Os dados da entrada se resumem a várias rodadas do jogo, em que cada uma possuía informações de cada jogador que participa da rodada.

Nesse contexto, foram bastante trabalhados os conhecimentos de Programação Orientada a Objetos, tratamento de erros, captação e interpretação de dados e organização de programas com múltiplos arquivos, além de depuração de programas desse perfil. Por fim, os principais desafios/dificuldades encontradas nesse trabalho foram: desenvolver uma estrutura coerente e organizada de classes e funções, avançar um programa sem o uso de nenhuma estrutura de dados trivial da linguagem, gerar gráficos de acesso à memória e depurar um programa com múltiplos arquivos e classes.

9. Referências

- Meira, W. & Pappa, P.L. (2022). *Slides virtuais e aulas da disciplina de estruturas de dados. Disponibilizado via Moodle.* Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte. Acesso em: 23 mai 2022.
- Colin et al. (2021). *Storage-Class Specifiers for External-Level Declarations*. Microsoft Docs. United States of America. Acesso em: 4 jun 2022.
- Lee, K. J. (2022). *Reading Data From Files Using C++*. Bowling Green State University. Bowling Green (OH). Acesso em: 4 jun 2022.
- Meira, W. (2022). *Analisamem*. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais, Belo Horizonte. Acesso em: 29 mai 2022.

10. Instruções para compilação e execução

Na raiz do projeto existe um *Makefile*. Para usar a função, basta se assegurar que existe um arquivo de entrada chamado “entrada.txt” na raiz do projeto e usar o comando “make all” no terminal. Com isso, os arquivos serão compilados e a devida saída da função será impressa em um arquivo chamado “saida.txt”. Caso se deseje somente compilar o arquivo, mas não executá-lo, deve-se chamar “make bin” no terminal. O arquivo executável gerado se chama “main” e está localizado na pasta bin.