Trabalho Prático 1

Luiz Felipe de Sousa Faria – 2020027148

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

lutilipe@ufmg.br

1. Introdução

O problema proposto foi implementar um jogo de Poker. As regras do jogo são semelhantes ao original. No entanto, cada jogador já começa com cinco cartas na mão e todos são obrigados a realizar o Pingo (*Ante*) em todas as rodadas. O objetivo da documentação é analisar o desempenho, a robustez, a localidade de referência e distância de pilha das principais estruturas, funções e métodos utilizados para o desenvolvimento do jogo, bem como refletir sobre algumas decisões e estruturas implementadas no projeto. Para resolver o jogo, foram utilizados o algoritmo de ordenação Bubble Sort e estruturas de dados pilha e vetores.

2. Método

2.1. Descrição

O programa principal se encontra no diretório ***TP***. Esse diretório é dividido em:

- ***include***: contém o arquivo *header* das classes *Player* (jogador), *Game* (jogo), *Hand* (mão) e *Card* (carta), do *memlog,* de funções de assert e de uma implementação de uma *Stack* (pilha);
- ***src***: contém a implementação das funções do include. A função *main* está em *poker.cpp*;
- ***obj***: contém os .o gerados pelo programa (vazio inicialmente)
- ***bin***: contém o binário do programa (vazio inicialmente)
- ***Makefile***: gera os arquivos binário e *.o* do programa e executa os testes de desempenho e análise de localidade.

O programa foi escrito em *C++* e compilado utilizando o *G++*.

O programa tem como entrada o arquivo “entrada.txt”, que contém as informações para o desenvolvimento do jogo, tais como número de rodadas, dinheiro inicial, nome e aposta de cada jogador; e valor do pingo para cada rodada.

Como saída, o programa gera o arquivo “saida.txt”, que contém o vencedor de cada rodada, junto com o dinheiro que recebeu e o rank da rodada (ou a string “0 0 I” caso a rodada for invalidada). No final, escreve de forma decrescente a classificação dos jogadores na partida, ordenados pela quantidade de dinheiro que obtiveram ao final de todas as rodadas.

As principais classes do programa são:

- ***Card***: responsável por guardar as informações de uma carta (naipe e valor) e obter esses dados através de uma *string*;
- ***Hand***: representa a mão do jogador. Possui um vetor de *Card* que representa as cartas do jogador; possui também 4 pilhas que armazenam as cartas do jogador, separadas pela repetição delas (1 a 4 repetições);
- ***Player***: representa um jogador da partida. Cada jogador possui uma *Hand,* além de outros atributos, como nome, aposta na rodada e total de dinheiro;
- ***PlayerRef***: armazena um ponteiro que se refere a um determinado *Player* da partida. Foi utilizado para realizar operações sobre o vetor contendo os jogadores da rodada e evitar vazamento de memória ou perda de referência do jogador.
- ***Game:*** representa o jogo em si. Controla as regras do jogo; lida com cada rodada e decide o(s) vencedor(es).

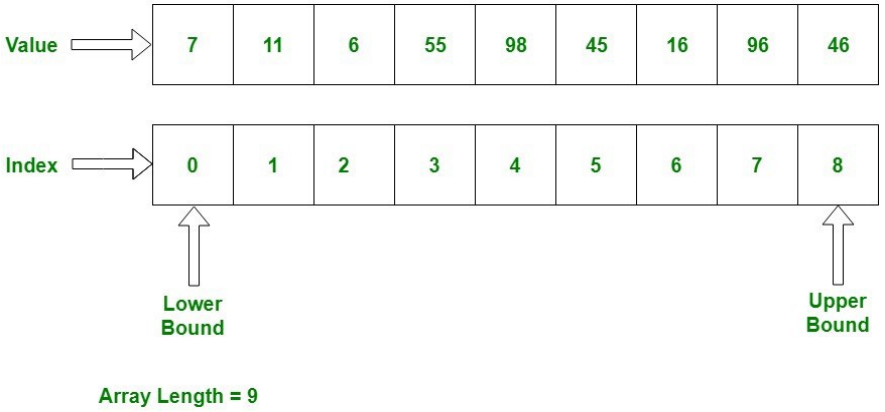
2.2. Estrutura de dados utilizadas

No projeto, foram utilizadas duas estruturas de dados principais: vetores (estáticos e dinâmicos) e pilhas encadeadas.

Os vetores (figura 1) são estruturas que armazenam sequencialmente na memória os dados. Eles podem ser estáticos, na qual seu tamanho é definido na compilação, ou dinâmicos, na qual seu tamanho é definido na execução e o mesmo é armazenado na memória *Heap*. Eles foram utilizados para armazenar os jogadores da rodada e da partida (localizado na classe *Game*); as cartas na mão dos jogadores (classe *Hand*); contar a repetição de cada carta (class *Hand*).

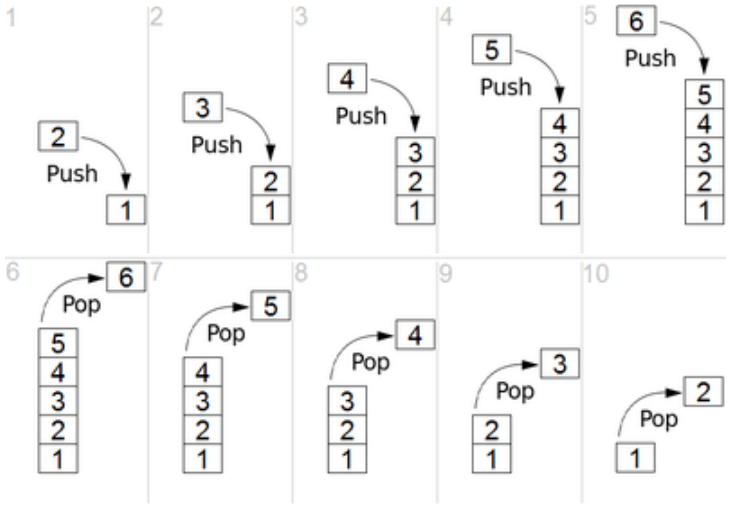
Já as pilhas encadeadas (figura 2) são estruturas de dados semelhantes aos vetores. No entanto, cada elemento dela possui um ponteiro para o próximo (ao invés de ser sequencial) e as operações de adição e remoção são feitas no elemento do topo, o que caracteriza a estrutura como ***LIFO*** (*last in, first out* -> primeiro a entrar, último a sair), No projeto, elas foram utilizadas para armazenar o valor das cartas dos jogadores separadas por quantidade de vezes que aparecem (esse valor está no intervalo fechado 1-4).

Figura 1: vetores



Disponível em: https://medium.com/nossolab/estrutura-de-dados-cf8875b1ed3d

Figura 2: Pilhas



Disponível em: https://osprogramadores.com/blog/2017/09/10/estruturas-dados-pilha/

2.3. Principais métodos e funções

Foram consideradas as principais funções aquelas chamadas na função ***handleRound*** (responsável por fazer as operações de cada rodada), localizada em *Game.cpp,* bem como as funções chamadas internamente por elas. As principais são (na ordem de execução), então:

- ***setRound***: responsável por criar o *setup* inicial do jogo (caso for o primeiro *round*) e criar o *setup* de cada *round* separadamente. Dentre esse setup, está a criação dos jogadores da partida (todos eles estão na primeira rodada da

partida); fazer a contribuição de cada jogador (pingo e aposta); e validar a rodada.

- ***setPlayersRank:*** responsável por classificar a mão de cada jogador da rodada. Para isso, a *Hand* de cada jogador possui o método ***rankHand***. Esse método identifica as repetições na mão e checa se as sequências se classificam como algum dos 10 tipos do jogo Poker: *Royal Straight Flush, Straight Flush, Four of a Kind, Full House, Flush, Straight, Three of a Kind, Two Pairs, One pair e High Card.*
- ***sortPlayersByRank***: como o nome sugere, ordena os jogadores a partir do rank obtido no método anterior. Utiliza o Bubble Sort.
- ***checkForDraws:*** a partir do vetor ordenado obtido do último método, checa-se aqueles jogadores com mesmo rank. Aqueles que tiverem um rank menor do que o maior rank, são excluídos da rodada (e do vetor de jogadores da rodada).
- ***handleDraws***: lida com os jogadores que tiveram o mesmo *rank*. Para isso, segue o critério de desempate para cada um dos *rank* descrito no enunciado. Internamente, chama-se a função ***compareWithSameRankHand***, responsável por comparar duas mãos de mesmo rank. Ela utiliza das pilhas descritas em *Hand* na seção 2.1. Um exemplo é para o tipo Full House (representado por uma tripla e um par). Primeiro compara a tripla de cada mão, desempilhando os valores da pilha ***triplesBundle*** *,* que contém o valor da tripĺa do maior para o menor (no caso só tem uma tripla, então só vai ter um valor), e compara com o da outra mão. Se for igual, compara o ***pairsBundle***, que contém os valores do par, fazendo o mesmo processo.
- ***handleRoundWinners***: Realiza um Bubble Sort sobre um vetor contendo os vencedores - mais precisamente, os jogadores que empataram da rodada - (caso o número de vencedores seja maior do que 1), ordenando eles alfabeticamente, e depois, percorre o mesmo vetor e escreve o quanto cada um recebeu, o rank da rodada e o nome de cada um no arquivo de saída.
- ***handleGameClassification***: realiza um BubbleSort para ordenar os jogadores baseado no quanto de dinheiro eles possuem ao final do jogo e escreve no arquivo de saída a classificação deles em ordem decrescente.

3. Análise de Complexidade e Espaço

A análise de complexidade e espaço foi feita sobre as funções descritas na seção 2.3. Considera-se **N** o número de jogadores da rodada e **M** a quantidade de rodadas. Segue-se então:

- ***setRound***: essa função, por ter a principal função de setar as informações vindas do arquivo de entrada para cada jogador, e lidar com as apostas e pingos de cada um, possui complexidade de tempo dada por ***O(5N) = O(N)*** e complexidade de espaço dada ***O(N),*** dada pela criação do vetor contendo os

jogadores da rodada (No caso da primeira, cria dois vetores de tamanho N). N é a quantidade de jogadores da rodada. Feita para **M** rodadas, a complexidade de tempo dá-se por **O(M.N).** A espacial se mantém.

- ***setPlayersRank:*** essa função percorre a *Hand* de cada jogador para encontrar o rank. A mão é composta por cinco cartas. Logo, a complexidade de tempo se dá por **O(4M.N) = O(M.N).** Para a complexidade de espaço, cria-se para cada jogador 4 pilhas, para armazenar as cartas baseado na quantidade de vezes que aparecem na mão. A quantidade máxima de valores inseridos em todas as pilhas juntas é 5 e a mínima é 2. Logo, O(1).
- ***sortPlayersByRank***: trata-se de um algoritmo de BubbleSort realizado em cada rodada. Logo, complexidade de tempo é **O(M.N^2)** e espaço **O(1)**.
- ***checkForDraws:*** trata-se de uma comparação linear para eliminar aqueles jogadores que possuem um rank menor do que o rank máximo (primeira posição do vetor ordenado). Sendo assim, a complexidade de tempo é **O(M.N)** e espaço **O(1)**.
- ***handleDraws***: essa função tenta desempatar os jogadores de mesmo rank. Com isso, ela percorre o vetor contendo os jogadores com maior rank (esse vetor tem tamanho mínimo de 1 e máximo de 4) e decide o vencedor, utilizando uma abordagem de dois ponteiros (ponteiros nesse caso é um índice). Como essa função não cresce em razão de N jogadores, é plausível considerar ela como sendo **O(M)** em termo da complexidade de tempo e **O(1)** em espaço.
- ***handleRoundWinners***: por fazer Bubble Sort um vetor de tamanho mínimo 1 e tamanho máximo 4, é possível considerar a complexidade de tempo da função como **O(16M) = O(M)**. A complexidade de espaço é **O(1)**.
- ***handleGameClassification***: realiza um BubbleSort sobre o vetor contendo todos os jogadores, e escreve o resultado em um arquivo de saída. Logo, a complexidade de tempo é **O(N) + O(N^2) = O(N^2).** A complexidade de espaço é constante.

Diante do exposto, é possível inferir a complexidade de tempo do programa como um todo como sendo **O(M.N^2)** e complexidade de espaço como sendo **O(N)**.

4. Estratégias de Robustez

Para garantir a robustez do programa, foram implementadas algumas estratégias para impedir valores indevidos de variáveis e acessos incorretos na memória.

A fim de lidar com acessos incorretos na memória, foram utilizadas funções de assert, que impedem o acesso a índices fora do tamanho dos vetores e da pilha. Caso esse acesso ocorra, o programa é abortado.

A fim de garantir a consistência da rodada, bem como validá-la, foi criada uma classe ***RoundException.*** Quando ocorre alguma inconsistência que invalida a

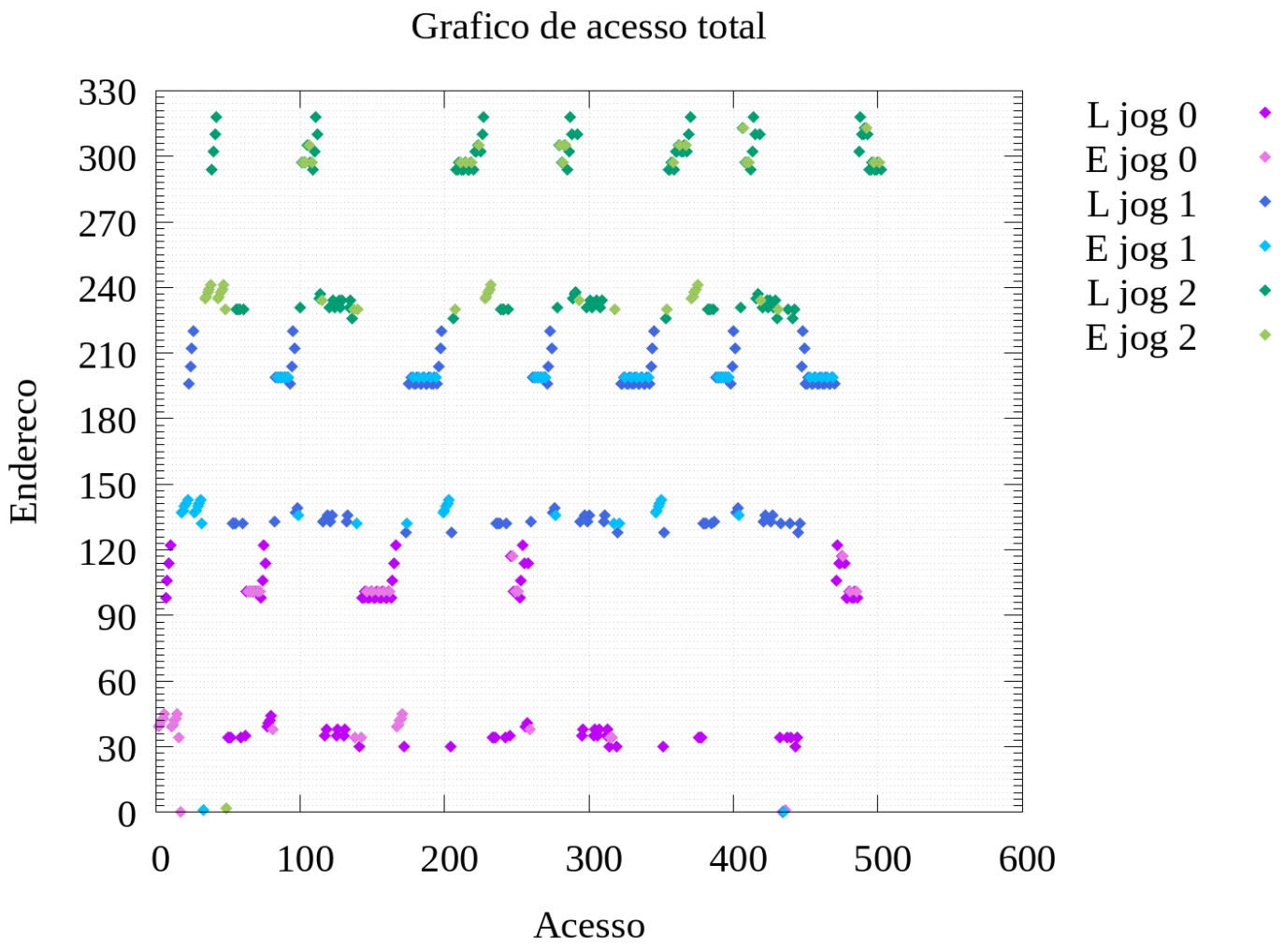
rodada, essa classe é disparada. Ela possui um *handler* que escreve no arquivo de saída que aquela rodada foi invalidada, e continua para a próxima rodada. Dentre as inconsistências que essa classe cobre, estão:

- Jogador apostar ou realizar o pingo sem ter uma quantia suficiente para tal;
- Jogador estar na rodada e não fazer nenhuma aposta;
- Jogador estar em uma rodada depois da primeira, mas não apostou na primeira;
- Valor negativo ou nulo do pingo ou aposta;

5. Análise Experimental

Para a análise experimental, foram realizados alguns experimentos a fim de medir a distância de pilha por rodada do programa, bem como essa distância de pilha varia com o tempo; calcular o tempo de execução das principais funções; comparar o tempo de execução do programa em relação a variância do tamanho da entrada, tanto de jogadores quanto de rodadas; e criar uma mapa de acesso de X jogadores em Y rodadas.

Figura 3: mapa de acesso à memória



5.1. Localidade de referência e acesso à memória

Para a criação do mapa de acesso e cálculo da distância de pilha, foram considerados três jogadores e cinco rodadas. Em cada rodada, os jogadores possuíam as seguintes combinações (dividido por rodada):

1) 1 - OnePair | 2 - OnePair | 3 - HighCard;
2) 1 - HighCard | 2 - Flush | 3 - Straight;
3) 1 - RoyalStraightFlush | 2 - StraightFlush | 3 - FullHouse;

Cada jogador é representado por uma cor no mapa. Os acessos são divididos em leitura e escrita em memória (**L** e **E**, respectivamente). O mapa está representado na **Figura 3**.

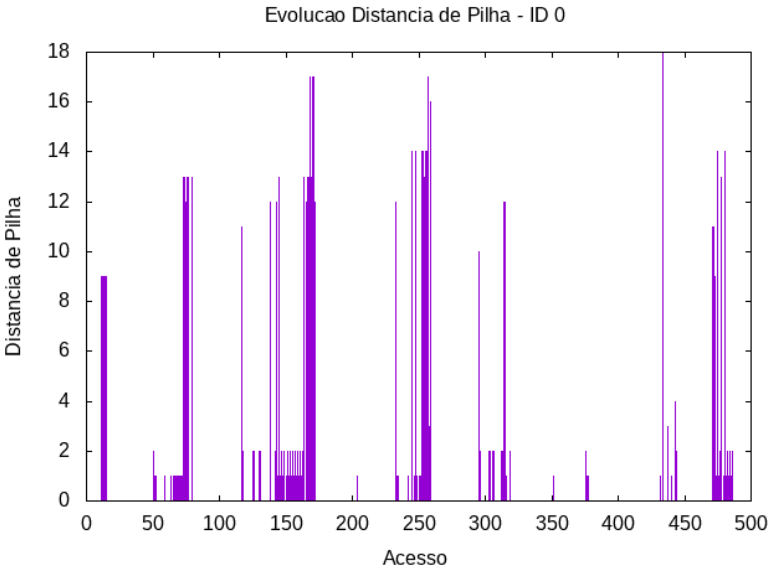
Fazendo uma análise do mapa, é possível perceber o comportamento do programa durante a execução de certas funções. Por exemplo, para a função ***getNumberOfDuplicatesAndBuildBundles*** - responsável por identificar a quantidade de cartas repetidas na mão do jogador e popular as pilhas com o valor dessas cartas, que mais tarde, serão usadas para desempate - é possível perceber que ela é representada no mapa pelos acessos superiores de cada jogador. Isso se deve ao fato dessa função ser linear, percorrer a mão do jogador e ir empilhando no seu respectivo *bundle* de repetição. No início da próxima rodada, essa pilha é limpa, permitindo que uma nova comparação seja feita.

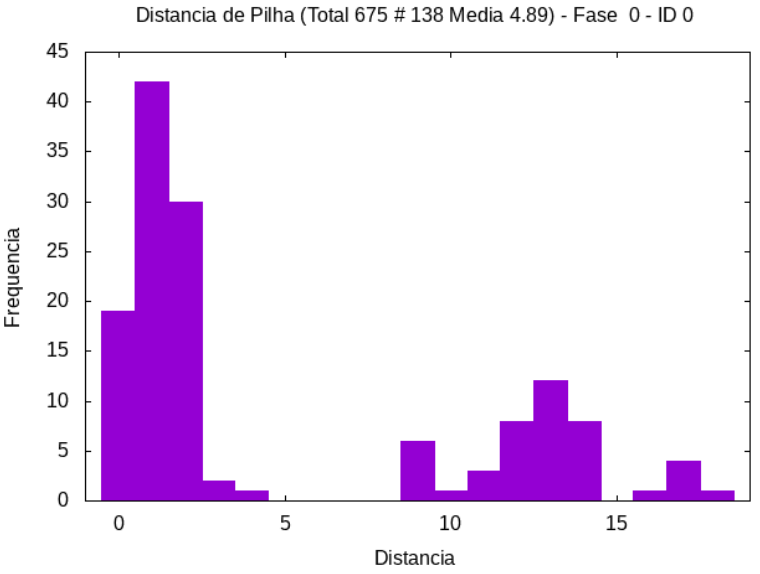
É importante ressaltar que ao fazer a análise desse mapa, é nítido perceber que o programa executa essa função que compara as repetições e monta as pilhas para todos os jogadores, independente da combinação que possui. Ou seja, um jogador que possui um ***Royal Straight Flush,*** que é a combinação mais alta do jogo, também passará por essa função, mesmo que sua mão represente que esse jogador venceu ou empatou a rodada, independente de qualquer coisa. Sendo assim, a execução dessa função seria desnecessária, mas foi mantida para seguir o fluxo de execução do programa.

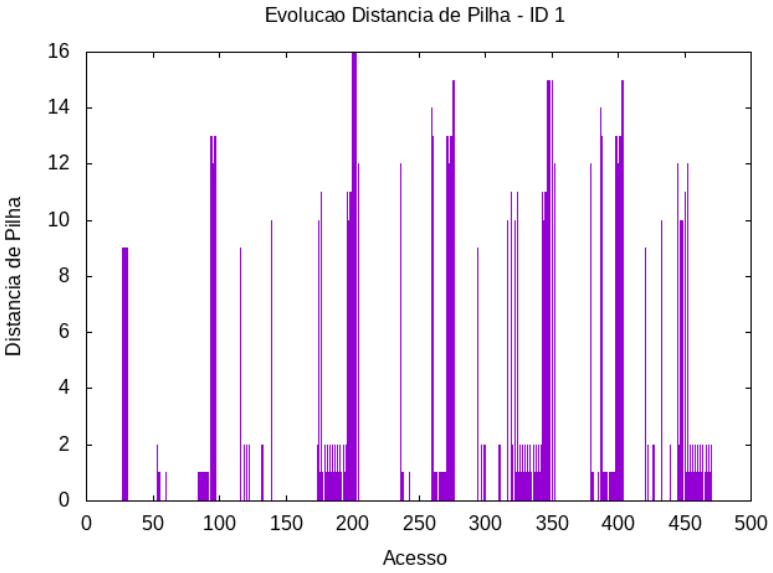
As funções de *sort*, tanto da mão quanto do *ranking* de cada jogador, também é percebida no mapa por um aglomerado de acessos na memória. Isso se deve ao fato dessas funções serem **O(N^2)** e realizarem alguns *swaps* dos valores do vetor.

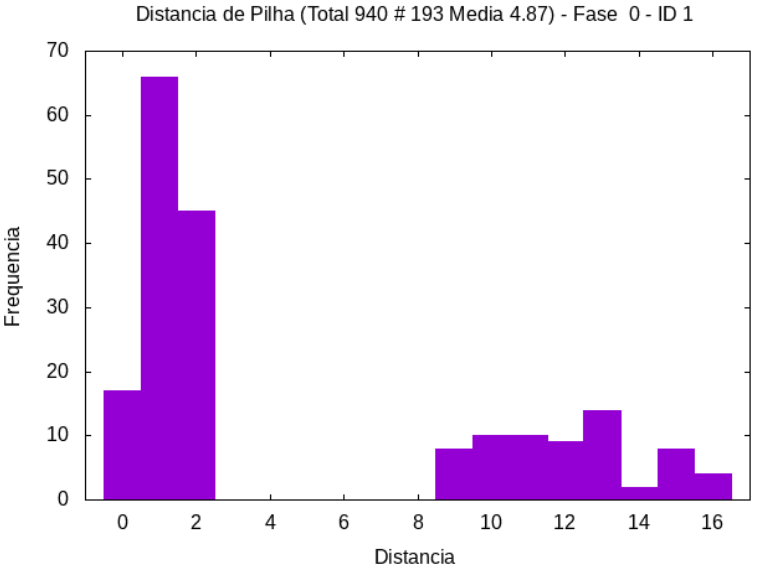
A distância de pilha total, bem como a evolução dela durante o decorrer do programa, é evidenciada pelos histogramas abaixo. Esses gráficos estão divididos para cada jogador.

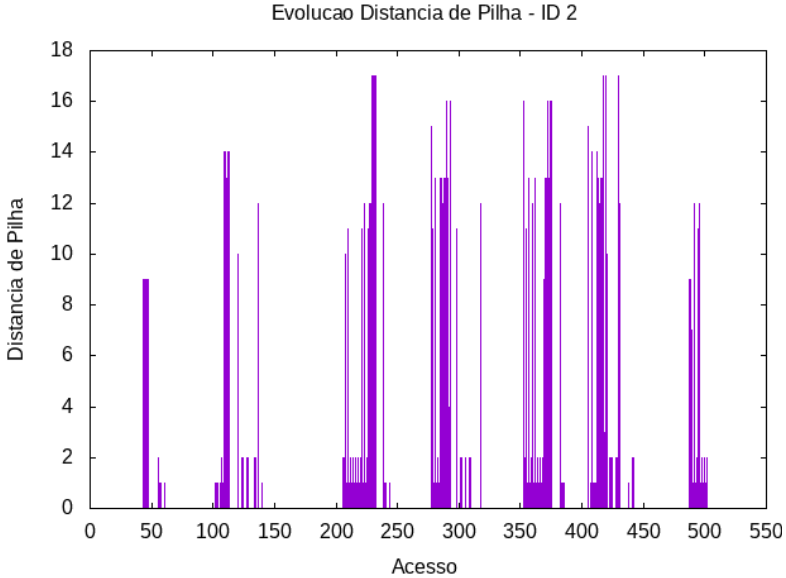
Apesar de muitos acessos terem uma distância de pilha baixa (entre 0-5), eles ocorreram muitas vezes. Isso fez com que a distância de pilha total fosse grande, mesmo para um número de entrada pequeno, o que indica uma necessidade de melhoria no programa.

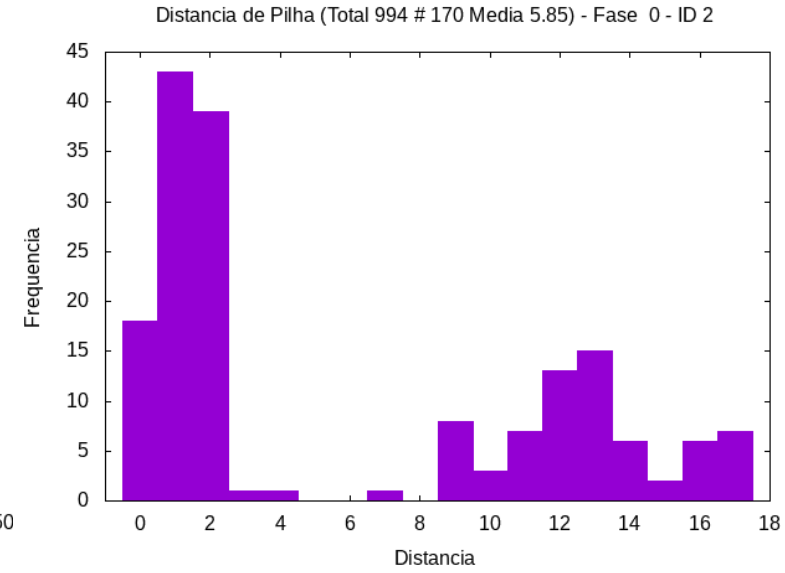

Evolucao Distancia de Pilha - ID 0
Distancia de Pilha
Acesso
0
2
4
6
8
10
12
14
16
18
0
50
100
150
200
250
300
350
400
450
500


Distancia de Pilha (Total 675 # 138 Media 4.89) - Fase 0 - ID 0
Frequencia
Distancia
0
5
10
15
20
25
30
35
40
45
0
5
10
15


Evolucao Distancia de Pilha - ID 1
Distancia de Pilha
Acesso
0
2
4
6
8
10
12
14
16
0
50
100
150
200
250
300
350
400
450
500


Distancia de Pilha (Total 940 # 193 Media 4.87) - Fase 0 - ID 1
Frequencia
Distancia
0
10
20
30
40
50
60
70
0
2
4
6
8
10
12
14
16


Evolucao Distancia de Pilha - ID 2
Distancia de Pilha
Acesso
0
2
4
6
8
10
12
14
16
18
0
50
100
150
200
250
300
350
400
450
500
550


Distancia de Pilha (Total 994 # 170 Media 5.85) - Fase 0 - ID 2
Frequencia
Distancia
0
5
10
15
20
25
30
35
40
45
0
2
4
6
8
10
12
14
16
18

5.2. Desempenho computacional

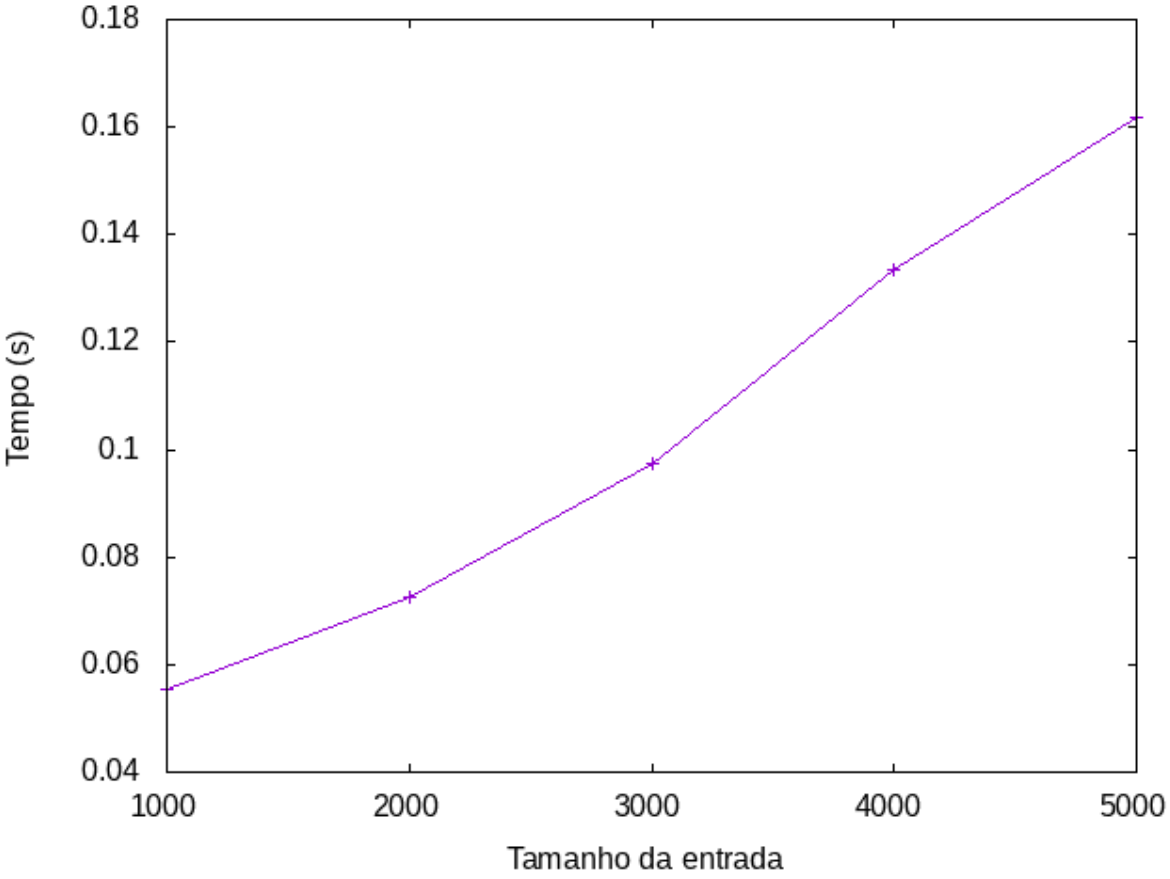
Para o teste de desempenho computacional, foram utilizadas entradas de tamanhos diferentes:

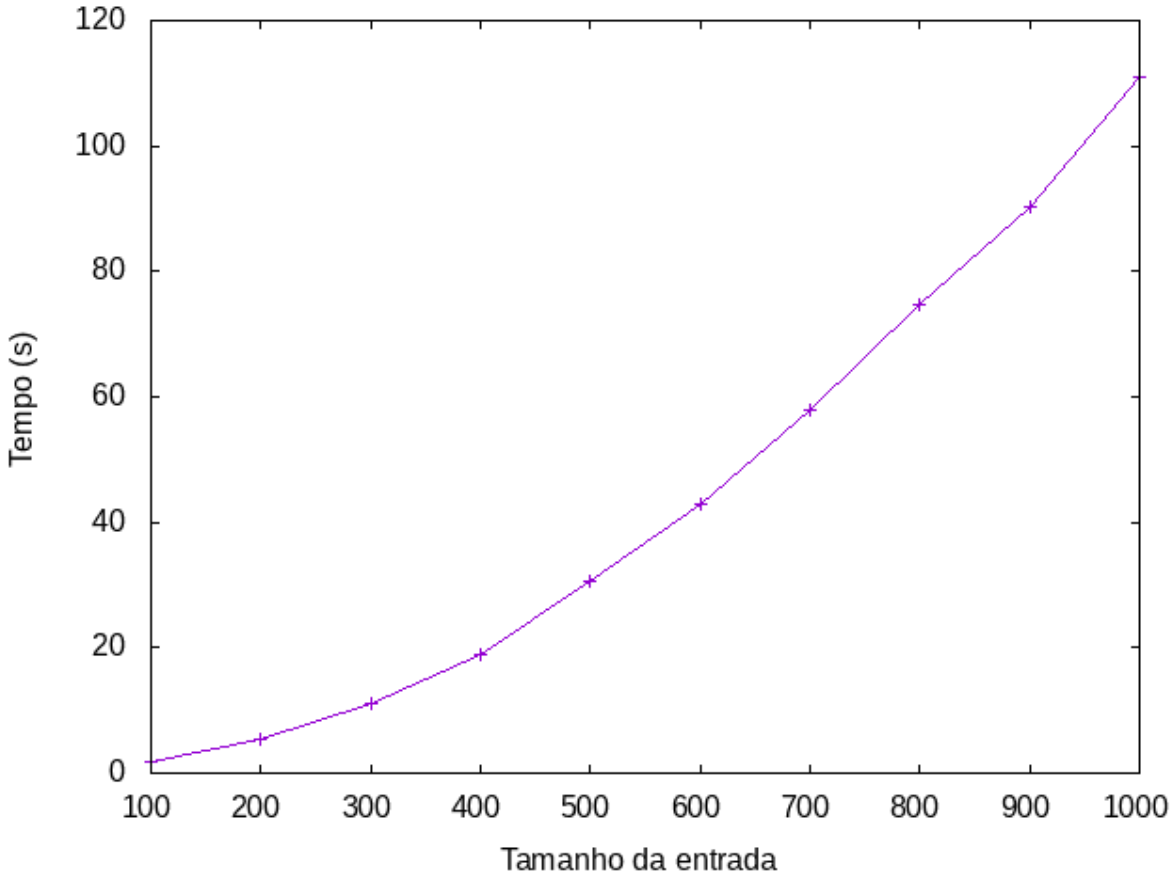
- Para a tabela de tempo de execução, considerou-se uma entrada de 1000 rodadas e 200 jogadores;
- Para o gráfico que compara o tempo de execução e o tamanho da entrada, variou-se primeiro o número de rodadas (1000 até 5000), fixando os jogadores em 10. Depois, fixou-se as rodadas em 1000 e variou-se os jogadores de 100 até 1000, com intervalo de 100.

Os resultados obtidos foram:

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
12.51	0.14	0.14	39921790	0.00	0.00	Player::getHand()
10.72	0.26	0.12	993	0.00	0.00	Game::sortPlayersByRank()
8.93	0.46	0.10	19760700	0.00	0.00	Hand::operator<(Hand*)
8.93	0.56	0.10	52954109	0.00	0.00	PlayerRef::getRef()
8.93	0.65	0.09	1000	0.00	0.00	Game::mountPlayersInRound(bool)
7.15	0.73	0.08	39720993	0.00	0.00	Hand::getRank()
2.68	0.88	0.0	199274	0.00	0.00	Hand::setRank(Hand::Rank)
2.68	0.91	0.03	1003	0.00	0.00	Player::increaseAmount(int)
1.79	0.97	0.02	20081127	0.00	0.00	Player::getName[abi:cxx11]()
1.79	0.99	0.02	12430689	0.00	0.00	PlayerRef::setRef(Player*)
1.79	01.03	0.02	200000	0.00	0.00	Hand::sortCards()
1.79	01.06	0.01	2000000	0.00	0.00	Card::operator>(Card)
0.89	01.08	0.01	854794	0.00	0.00	Stack<Card::Number>::push(Card::Number)
0.89	01.09	0.01	200000	0.00	0.00	Player::doBet()
0.89	1.10	0.01	199800	0.00	0.00	Player::resetBet()
0.89	1.11	0.01	199281	0.00	0.00	Hand::getNumberOfDuplicatesAndBuildBundles()
0.89	1.12	0.01	1000	0.00	0.00	Game::setPlayersRank()
0.89	1.12	0.00	5826953	0.00	0.00	Card::getValue()
0.89	1.12	0.00	2538837	0.00	0.00	Stack<Card::Number>::empty()
0.89	1.12	0.00	857192	0.00	0.00	StackNode<Card::Number>::StackNode()
0.89	1.12	0.00	854794	0.00	0.00	Stack<Card::Number>::pop()
0.89	1.12	0.00	802398	0.00	0.00	Stack<Card::Number>::clean()
0.89	1.12	0.00	797124	0.00	0.00	Stack<Card::Number>::size()

Na tabela, é possível perceber a diferença de tempo de execução da função. A função ***sortPlayersByHand***, apesar de ser chamada um número de vezes pequeno comparada às outras funções, ocupa ~10% do tempo total da execução.


0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
Tempo (s)
1000
2000
3000
4000
5000
Tamanho da entrada


120
100
80
60
40
20
0
Tempo (s)
100
200
300
400
500
600
700
800
900
1000
Tamanho da entrada

O primeiro gráfico representa o experimento na qual o número de jogadores ficou fixo em 10 e variou-se apenas a quantidade de rodadas. Nesse caso, é possível perceber a linearidade dos casos, mesmo tendo um aumento de 1000 em cada intervalo.

O segundo gráfico representa o experimento na qual fixou-se a quantidade de rodadas em 1000 e variou o número de jogadores de 100 até 1000, com intervalo de 100. Nesse caso, ao contrário do anterior, o gráfico cresce com uma função quadrática.

Analisando os dois gráficos, é possível comprovar a complexidade de tempo do programa, proposta na seção 3, como sendo **$O(M.N^2)$,** em que M é a quantidade de rodadas e N a quantidade de jogadores.

6. Conclusão

O presente trabalho propôs-se a descrever um jogo de Pôquer em C++, utilizando estruturas de dados e algoritmos de ordenação, bem como fazer a análise computacional de desempenho e acesso à memória das operações realizadas.

A análise feita sobre o programa acaba se tornando mais complexa, devido às estruturas não sequenciais e a complexidade dos algoritmos implementados. No entanto, com a geração de gráficos e tabelas feitas durante as análises, é possível visualizar o comportamento da estrutura com diferentes tamanhos de entrada, tanto de rodadas quanto jogadores, ao longo da execução do programa, tornando a análise mais simples.

7. Bibliografia

CHORMEN, Thomasb H. *et al*. **Algoritmos - Teoria e Prática**. 3ª edição.
Capítulo 3: Crescimento de funções. Editora Elsevier, 10 de abril de 2012.

APÊNDICE A - INSTRUÇÕES PARA COMPILAÇÃO E EXECUÇÃO

A compilação do programa pode acontecer apenas rodando o comando **make** ou **make run**.

O programa necessita a existência de um arquivo chamado **entrada.txt** na raiz do projeto (mesmo diretório do arquivo MakeFile). Esse arquivo deve seguir o padrão:

Partida = [número de rodadas, dinheiro inicial][$Rodada_1$, $Rodada_2$, ..., $Rodada_n$]
Rodada = [número de jogadores, pingo][$Jogada_1$, $Jogada_2$, ..., $Jogada_j$]
Jogada = [nome do jogador, aposta, mão]

O parâmetro **-l** pode ser passado para o programa para habilitar o log do acesso à memória. Esse recurso é desabilitado por padrão. Ele depende do programa analisam, não contido na *build* do projeto.