Trabalho Pratico 1

Poker Face

João Vitor Ferreira

Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brasil

joaovitorferreira@ufmg.br

1. Introdução

O problema proposto foi de implementar um código no qual realiza a análise e saída de uma partida inteira de Poker. Tendo armazenados os resultados de todas as rodadas. Lendo os dados de um arquivo e salvando os resultados das rodadas e do jogo num arquivo denominado saída

2. Método

O programa foi desenvolvido na linguagem C++/C, compilada pelo compilador G++ da GNU Compiler Collection.

2.1. Estrutura de Dados

A implementação teve como base o uso de 4 struct que armazenam as variáveis relevantes ao código.

Baralho armazena os dados das cartas, contendo 3 variáveis para isto, uma para a carta completa no formato string nomeada _carta, o valor da carta no formato int

_valor_carta e o naipe da carta como um char de nome _naipe_carta. Esta classe não contém nenhuma função agregada a ela.

Jogador armazena e lida com as operações relevante aos jogadores. Possui 10 variáveis e 5 funções. As variáveis guardam o nome do jogador como uma string sendo ela chamada de _nome, o dinheiro deste jogador como um int de nome _dinheiro, uma classificação numérico int para o tipo de mão que ele possui _tipo_da_mao, uma string _tipo_jogada que contém a sigla do conjunto de cartas que ele possui está associada a _tipo_jogada, um in _ganhou_partida que armazena se o jogador venceu a partida, e _valor_quarterto, _valor_trinca,_valor_par_1, _valor_par_2, armazenam os valores das cartas que compõem o quarteto, trinca e pares casos eles estejam presentes na mão do jogador na rodada, e _id que é uma identificação para o jogador e por fim há a inicialização de um Baralho que é um vetor contendo 5 posições que armazena as informações das cartas contidas na mão do jogador na rodada. As funções são:

- string converterJogadorString(char *jog), recebe um char como parâmetro e realiza a conversão deste char para um string e retorna esta string.
- string converterCartaString(int val_carta, char naipe), recebe um int contendo o valor da carta e um char contendo o naipe. Converte o valor de int para string e o char para string e retornas as duas concatenadas nessa respectiva ordem.
- int EncontraJogador(Jogador jogador[], string nome_buscado, int qtde_jogadores), recebe como parâmetro um vetor de jogadores, o nome do jogador procurado e a quantidade de jogadores totais na partida. Através de um loop realiza a procura deste jogador na partida e caso ela seja encontrada a posição em que ele se encontra é retornada.
- void LimpaDadosJogador(Jogador jogador[], int qtd), recebe o vetor de jogadores como parâmetro e realiza um loop que seta todos os parâmetros do jogador como inválidos, exceto o nome e dinheiro.
- void AtribuiGanhosVencedor(Jogador jogador[], int qtd, int ganhos), recebe o jogador, a quantidade de jogadores e o valor do montante do pote. Realiza um loop que encontra a quantidade de jogadores que venceram a partida e cria uma variável que armazena essa quantidade, depois realiza outro laco que realiza o

incremento do dinheiro dos jogadores vencedores pela razão dos ganhos pela quantidade que ganharam a partida.

Pote armazena os dados do pote da rodada. Contém 2 variáveis e 4 funções. As variáveis que ele possui são denominadas montante, armazena os valores de apostas e pingos das rodadas no formato int e _id uma identificação. As funções são:

- void CriaPote(Pote *pote_rodada), recebe como parâmetro um pote e inicializa o montante do pote com o valor nulo 0. Por ser um void não possui retorno.
- void adicionaPingo(Jogador jogador[], Pote *pote_rodada, int valor_pingo, int num_jogadores), recebe um vetor de jogadores, um pote, o valor do pingo e um número total de jogadores. Ela percorre o vetor de jogadores fazendo a subtração no valor de dinheiro que eles possuem e a inserção desse valor no pote, o valor que é retirado de 1 e passado para o outro é o que está contido na variável valor_pingo. Por ser um void não possui retorno.
- void adicionaAposta(Pote *pote_rodada, int valor_aposta), recebe o pote e o valor da aposta como parâmetros e realiza a inserção desse valor no Pote. Por ser um void não possui retorno.
- int GetMontante(Pote *pote_rodada), recebe um Pote como referência e retorna o valor de apostas e pingos adicionados a ele.

Rodada é somente o header que contém as funções que serão usadas para realizar as manipulações nas cartas. As funções são:

- void Ordena(Jogador *jogador, int posicao), recebe o jogador e sua posição no vetor. Realiza um Bubble Sort para organizar as cartas pôr em valor crescente. Por ser um void não possui retorno.
- void OrdenaJogadores(Jogador *jogador, int qtde_jog) , recebe o jogador a quantidade total de jogadores. Realiza um Bubble Sort para organizar os jogadores por seu valor de dinheiro de forma decrescente. Por ser um void não possui retorno.
- int BuscaRepeticaoNaipe(Jogador *jogador, int num_jogador), recebe o jogador e sua posição no vetor. Faz a análise e comparação de todos os naipes

das cartas na mão do jogador e retorna se todas as cartas possuem o mesmo naipe.

- int BuscaRepeticaoValor(Jogador *jogador, int num_jogador), recebe o jogador e sua posição no vetor. Faz a análise e comparação de todos os valores das cartas na mão do jogador e retorna se nas cartas possui algum valor repetido, caso haja retorna se é uma dupla, duas duplas, uma trinca, uma trinca e dupla ou um quarteto.
- int AnalisaOrdemFlush(Jogador *jogador, int num_jogador), recebe o jogador e sua posição no vetor. Realiza a análise e comparação de todos os valores das cartas, após a função BuscaRepeticaoNaipe ter retornado que o naipe é o mesmo, ela compara todos os valores se são maiores ou iguais a 10, caso seja haverá uma adição na variável, caso contrário haverá um subtração. No fim ela é retornada e terá 3 analises de valor diferentes, caso seja 4 todas as cartas serão maiores de 10, em caso de ser -4 todas as cartas são menores que 10.
- int AnalisaOrdemCrescente(Jogador *jogador, int num_jogador), recebe o jogador e sua posição no vetor. Realiza comparações de uma carta do jogador com a próxima e define se a carta atual é uma unidade maior que as próximas formando assim uma sequência crescente. E retorna se ela é crescente ou não.
- void AnalisaJogada(Jogador *jogador, int posicao), recebe o jogador e sua posição no vetor. Realiza a chamada das funções acima e comparar os resultados dela e atribui o _tipo_da_mao e _tipo_rodada de cada jogador.
- void BuscaRepeticaoDeCarta(Jogador *jogador, int num_jogador), recebe o jogador e sua posição no vetor. Chamado pela função AnalisaJogada, realiza um laço no vetor de carta do jogador e atribui as variáveis de carta repetida o valor da carta que é repetida, o que servira para definir o tipo de mão.
- int BuscaRepeticaoDeMao(Jogador *jogador, int tipo_retorno, int qtde_jog), recebe o jogador, o número correspondente ao tipo que ela irá retornar e a quantidade de jogadores no vetor. Analisa dentre todos os jogadores que possuem a mesma mão, se houver algum caso de empate, no caso de houver 2 grupos de jogadores com a mãos de tipo iguais a mão que tiver o maior peso, ou seja, o menor tipo de mão sera a considerada. Logo após isso retorna o que é especificado na função seja o tipo da mão, a quantidade de jogadores com essa mão ou a posicao dos jogadores

- int ResolveEmpate(Jogador *jogador, int jog_a, int jog_b), recebe o jogador e duas posições a e b. Analisa a mão dos jogadores e retorna o resultado da análise do empate e qual jogador venceu a partida se foi possível realiza o desempate dos jogadores.
- String VenceuPartida(Jogador *jogador, int num_jogadores, int pote) recebe o jogador, a quantidade total de jogadores e o montante do pote. Compara o tipo de mão dos jogadores, caso encontre um empate chama a função resolve empate. Após decidir qual jogador venceu a partida inicializa a variável _ganhou_partida do(s) jogador(es) vencedor(es) e retorna o resultado que sera salvo no arquivo determinado saída, na ordem a quantidade de jogadores vencedores, o valor do pote, o tipo de mão e o nome do(s) jogador(es).

Partida contém somente os comandos que são necessários para que o jogo seja executado e é feito numa função denominada void ExecutaPartida().

Main somente há o main do programa nele é iniciado e finalizado a Analise de Complexidade e chamado a função ExecutaPartida.

2.2. Classes

A modularização foi implementada ao longo do código, por tanto foi necessário a criação de 3 struct, uma para o Pote no qual armazenaria o valor do montante e o id do pote ambas são do tipo int. Uma para o Baralho que tem as variáveis, _carta (string), _naipe_carta (char), _valor_carta (int)). Jogador, _ganhou_partida (int), _valor_quarterto (int), _valor_trinca (int),_valor_par_1 (int), _valor_par_2 (int) _nome (string), _tipo_jogada (string), _dinheiro (int), _id (int), Baralho Cartas_jogador[5] (Um array de 5 posições, a quantidade de cartas que cada jogador possui na mão na rodada), _tipo_de_mao (int) e _tipo_jogada (string).

2.3. Formato de Entrada e Saída

O formato de entrada é a inserção do arquivo entrada preenchido conforme especificado na seção instruções para compilação e execução deste documento. A saída é padronizada e está contida no arquivo ‘saida.txt’ que é gerado na execução do programa

3. Analise de Complexidade

3.1. Tempo

Inicialmente iremos seguir algumas preposições as estruturas auxiliares mais básicas consideraremos com o $O(1)$, e o custo da funções são:

- Pote

void criaPote(Pote *pote_rodada) = Nesta função há somente uma atribuição, portanto

$$= O(1)$$

void adicionaPingo(Jogador jogador[], Pote *pote_rodada, int valor_pingo, int num_jogadores) = Nesta função há duas atribuições e um loop. Logo

$$2 \cdot O(1) + O(n)$$

$$= \max\left(O(1), O(n)\right)$$

$$= O(n)$$

void adicionaAposta(Pote *pote_rodada, int valor_aposta) = Nesta função há somente uma atribuição, portanto:

$$O(1)$$

int GetMontante(Pote *pote_rodada) = Nesta função há somente uma atribuição, dessa forma temos:

$$O(1)$$

- Jogador

string converterJogadorString(char *jog) = Aqui há somente 1 atribuição e o retorno dessa variável logo temos:

$$O(1)$$

string converterCartaString(int val_carta, char naipe) = Aqui há 2 atribuição e o retorno da junção dessa variável:

$$2 \cdot O(1)$$

$$= O(1)$$

int EncontraJogador(Jogador jogador[], string nome_buscado, int qtde_jogadores) = Nesta função temo uma atribuição de variável, 1 comparação e um loop. Como somente os laços estão sendo considerados como custo temos que:

$$O(n) \cdot 2 \cdot O(1)$$

$$\max\left(O(n), O(1)\right)$$

$$= O(n)$$

- Rodada

void Ordena(Jogador *jogador, int posicao) = Implementa dois laços alinhados e realiza 1 comparação e 3 atribuições, logo temos que $O(n^2)$

void OrdenaJogadores(Jogador *jogador, int qtde_jog) Implementa dois laços alinhados e realiza 1 comparação e 3 atribuições, logo temos que $O(n^2)$

int BuscaRepeticaoNaipe(Jogador *jogador, int num_jogador) = Implementa um for, uma comparação e uma atribuição, o custo seria n, porém o laço possui o tamanho fixado em 5 execuções então

$$3 \cdot O(1)$$

$$= O(1)$$

int BuscaRepeticaoValor(Jogador *jogador, int num_jogador) = Implementa um for, 4 comparações e 7 atribuições, o custo seria n, porém o laço possui o tamanho fixado em 5 execuções então

$$11 \cdot O(1)$$

$$= O(1)$$

int AnalisaOrdemFlush(Jogador *jogador, int num_jogador) = Implementa um for, 2 comparações e 4 atribuições, o custo seria n, porém o laço possui o tamanho fixado em 5 execuções então

$$6 \cdot O(1)$$

$$= O(1)$$

int AnalisaOrdemCrescente(Jogador *jogador, int num_jogador) = Implementa um laço, 2 comparações e 4 atribuições, o custo seria n, porém o laço possui o tamanho fixado em 5 execuções então

$$6 \cdot O(1)$$

$$= O(1)$$

void AnalisaJogada(Jogador *jogador, int posicao) = Somente realiza comparações e em caso de alguma ser positiva realiza 2 atribuições como neste caso as comparações e atribuições não estão sendo consideradas temo que $O(1)$

void BuscaRepeticaoDeCarta(Jogador *jogador, int num_jogador) = Implementa um laço, 6 comparações e 7 atribuições, o custo seria n, porém o laço possui o tamanho fixado em 5 execuções então

$$13 \cdot O(1)$$

$$= O(1)$$

int BuscaRepeticaoDeMao(Jogador *jogador, int tipo_retorno, int qtde_jog) = Implementa um laço com dezenas de comparações e atribuições dentro dele, Portanto temos que

$$O(n)$$

int ResolveEmpate(Jogador *jogador, int jog_a, int jog_b) = Somente realiza comparações e atribuições tem-se então $O(1)$

string VenceuPartida(Jogador *jogador, int num_jogadores) = Realiza um laço e 5 comparações fora atribuições resultando em $O(n)$

- Partida

void ExecutaPartida() = Como somente são considerados os laços, eles serão contabilizados somente

$$O(n) \cdot \big(O(n) \cdot O(n)\big) + O(n)$$

$$= \max\left(O(n), 2 \cdot O(n), O(n)\right)$$

$$= O(n)$$

3.2. Espaço

O espaço utilizado é variável e dependera da quantidade de jogadores que serão adicionados na partida, considerando um valor x de jogadores sera alocado um quantidade x de posições de jogadores, no entanto cada jogador possui 5 cartas então sera alocado no total para cada jogador 6x posições de memória especificas somente para guardar o local onde estão os dados do jogador e das cartas, realizando outra analise cada jogador possui 10 atributos e cada carta 3. Considerando que em algumas funções são criados locais para armazenas estes conteúdos, no entanto ao realizar uma analisa mais detalhada notamos que, na grande maioria das funções e manipulações são feitas nos locais já previamente estabelecidos, usando assim de um espaço menor. Desta forma desconsiderarei o espaço usado por manipulações que utilizem de espaço que não foram previamente alocados e que minhas funções somente utilizem do espaço alocado inicialmente, então teríamos $\theta(n)$ para os

jogadores e como cada jogador possui 5 cartas $\theta(5)$ para as cartas. Desta forma temos que o espaço é de $\theta(n)$.

4. Estratégias de Robustez

Para garantir o correto funcionamento do código foram adicionados errosAssert, dos quais param o código toda vez que um erro é localizado. Os erros que foram adicionados estão relacionados com a abertura dos arquivos e parâmetros passados invalidamente. No caso da abertura do arquivo caso ele está vazio um erro retornara avisando que não há nada no arquivo. No caso dos parâmetros temos erros para todos que não são passados corretamente seguindo as instruções, ou seja, o programa sera abortado em caso de valores menores que 0 para o número de rodadas, dinheiro dos jogadores, número de jogadores, pingo da rodada, aposta dos jogadores, o número de jogadores das rodas subsequentes a primeira ser maior do que a primeira rodada, na rodada haver um jogador que não foi inicializado anteriormente, caso o valor da carta seja maior que 13 ou menor que 1.

5. Análise Experimental

Para realizar a análise utilizamos o gerador de carga disponibilizado no Moodle da disciplina, o “geracarga”. Foi gerado uma partida contendo 10 jogadores e 20 rodadas. Os mapas de acesso à memória desse caso, além dos tempos tomados pelo programa e a distância de pilha. Para cada um desses testes foi gerado uma derivação para cada jogador da partida.

6.1 Mapas de Acesso

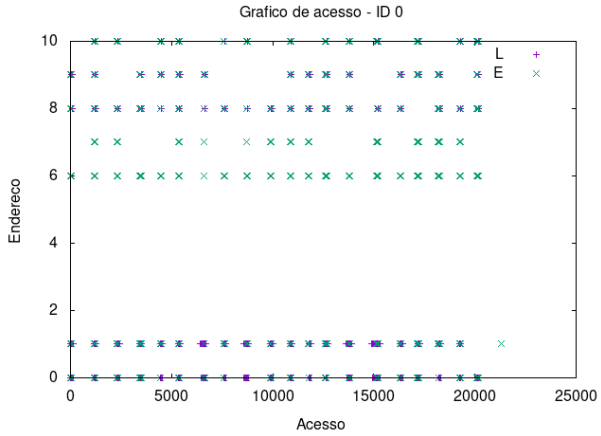


Grafico de acesso - ID 0
Endereco
Acesso
L
E
0
2
4
6
8
10
0
5000
10000
15000
20000
25000

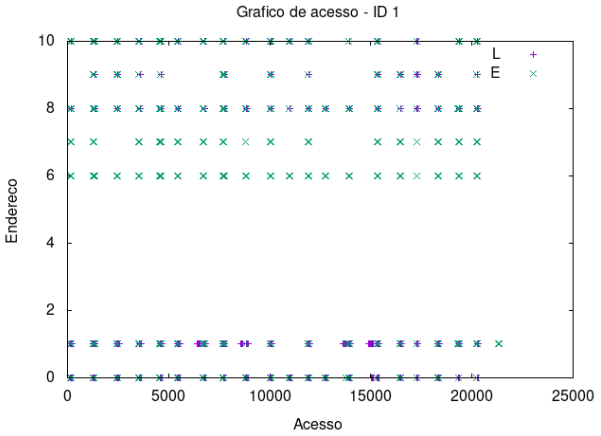


Grafico de acesso - ID 1
Endereco
Acesso
L
E
0
2
4
6
8
10
0
5000
10000
15000
20000
25000

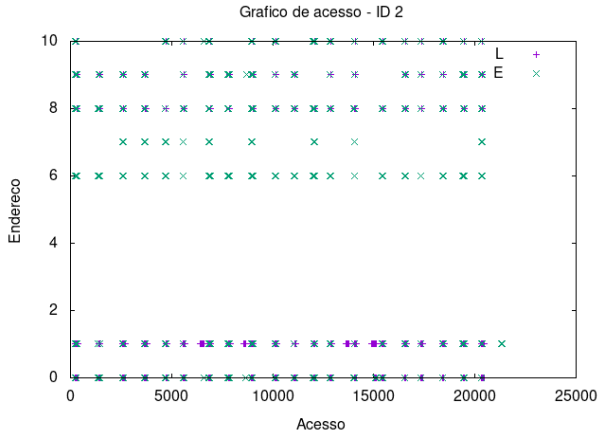


Grafico de acesso - ID 2
Endereco
Acesso
L
E
0
2
4
6
8
10
0
5000
10000
15000
20000
25000

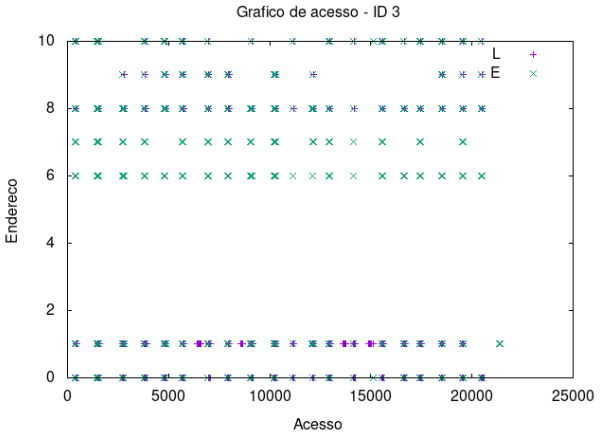


Grafico de acesso - ID 3
Endereco
Acesso
L
E
0
2
4
6
8
10
0
5000
10000
15000
20000
25000

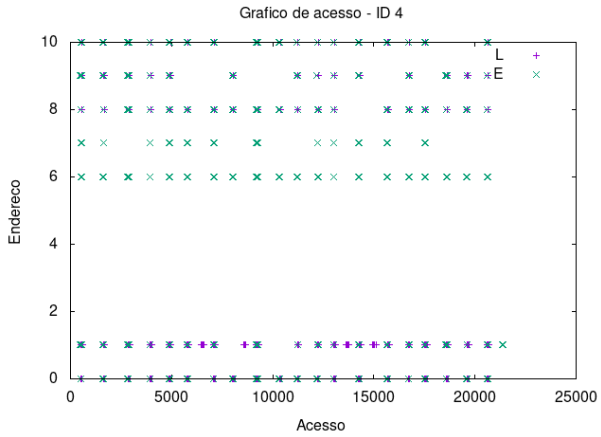


Grafico de acesso - ID 4
Endereco
Acesso
L
E
0
2
4
6
8
10
0
5000
10000
15000
20000
25000

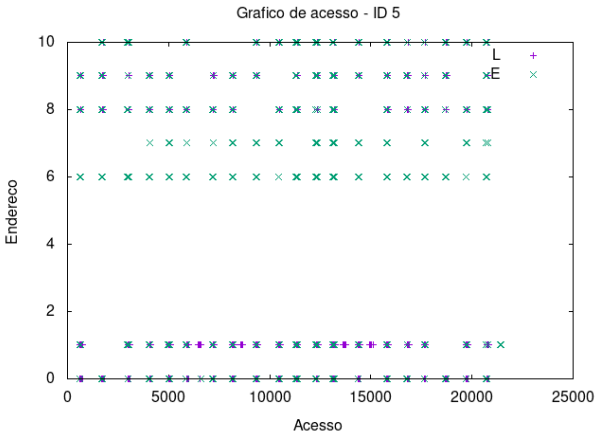
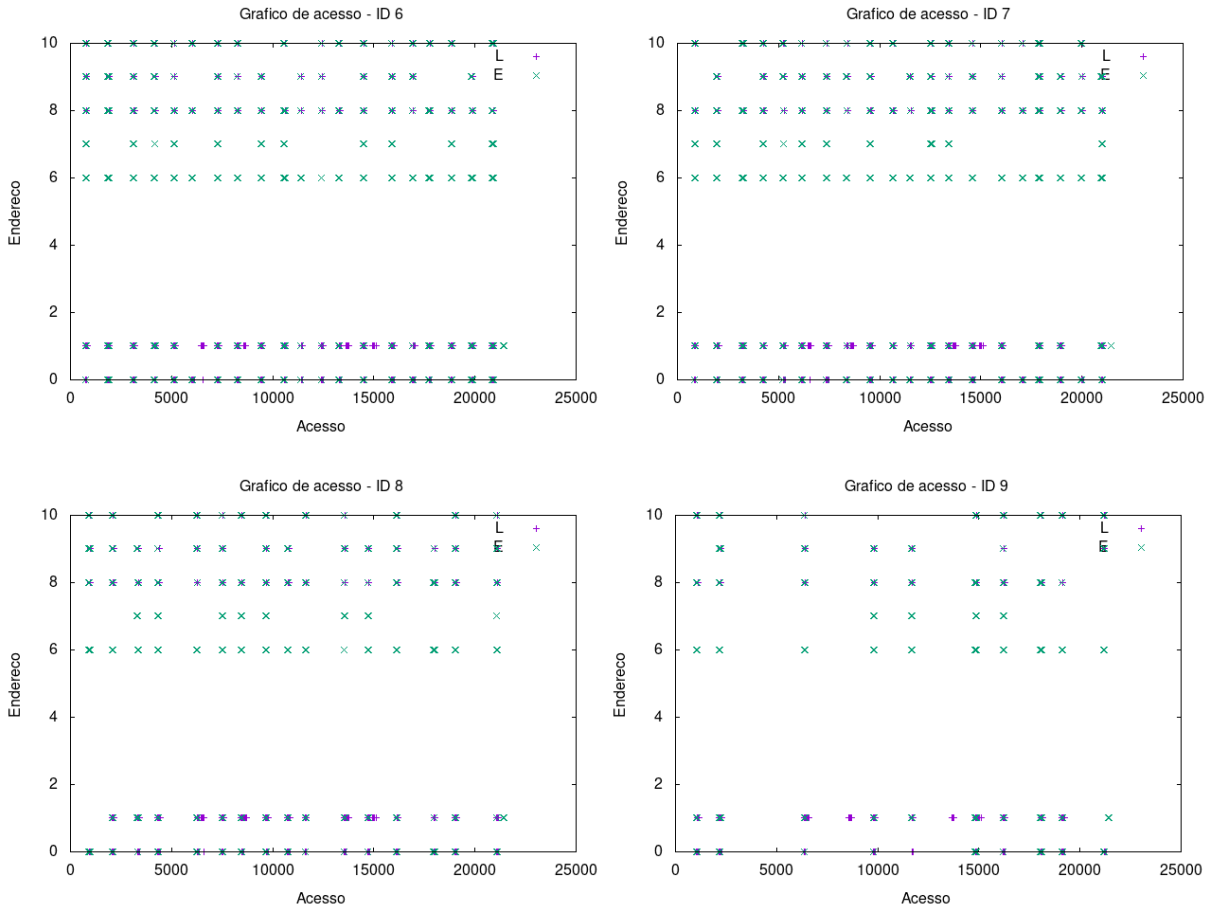
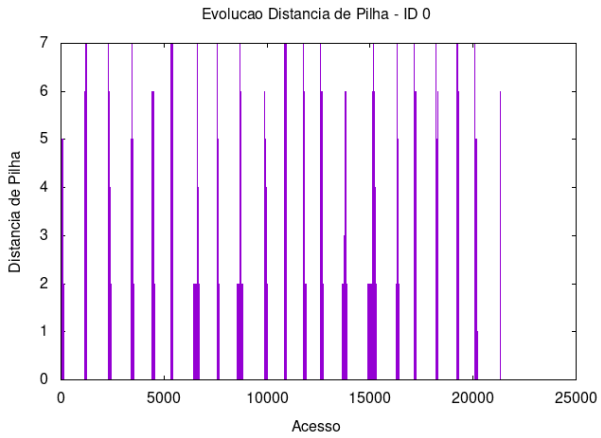


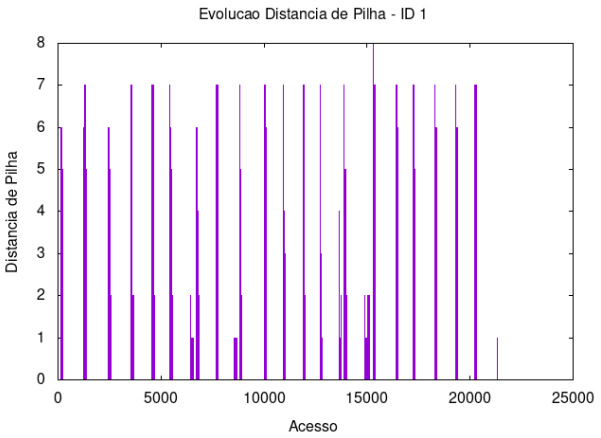
Grafico de acesso - ID 5
Endereco
Acesso
L
E
0
2
4
6
8
10
0
5000
10000
15000
20000
25000

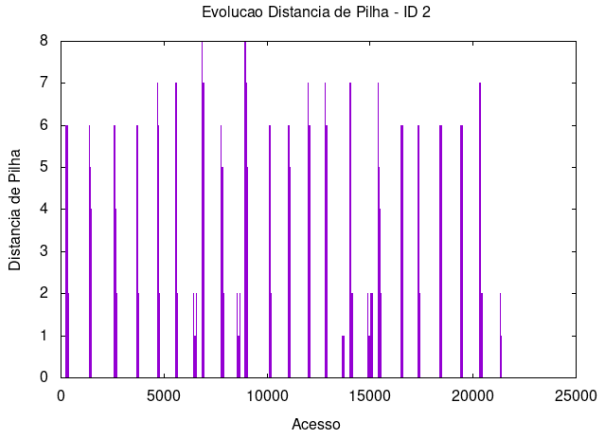


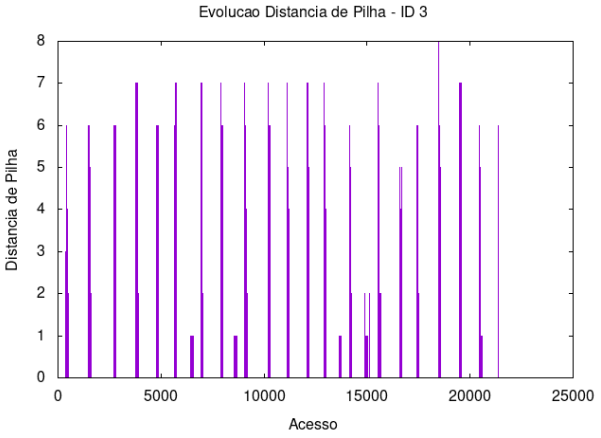
Analisando os gráficos de uma maneira mais geral é notório a forma em que a memória é acessada para leitura e escrita deriva exatamente da forma em que o programa foi desenvolvido. Nota-se que os acesso são sempre consecutivos e seguem certo padrão, ou seja, é possível notar certa continuidade nos gráficos se analisarmos áreas especificas de todos juntamente, vemos que o programa faz a alocação continua. O que resulta num acesso a memoria padronizado. Outra questão notável é a simetria visto que os jogadores são exatamente os mesmos em todas as rodadas então é possível perceber em quais rodadas cada jogar estava presente, já que cada marcação de leitura/escrita representa um momento do jogo em que o jogador correspondente ao id estava sendo manipulado.

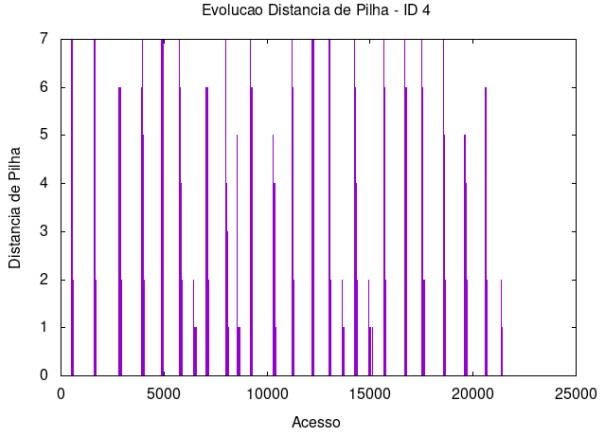
6.1.2 Evolução Distância de pilha

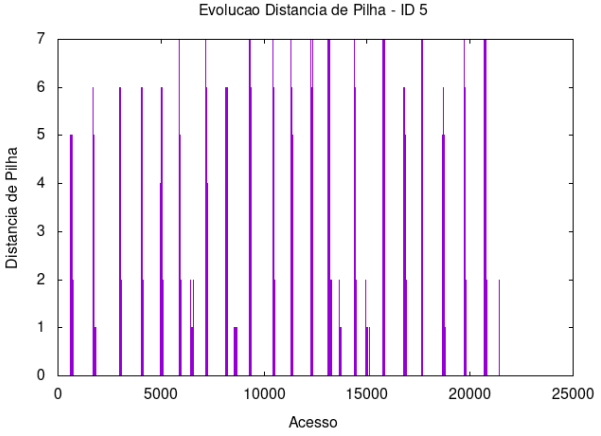

Evolucao Distancia de Pilha - ID 0
Distancia de Pilha
Acesso
0
1
2
3
4
5
6
7
0
5000
10000
15000
20000
25000

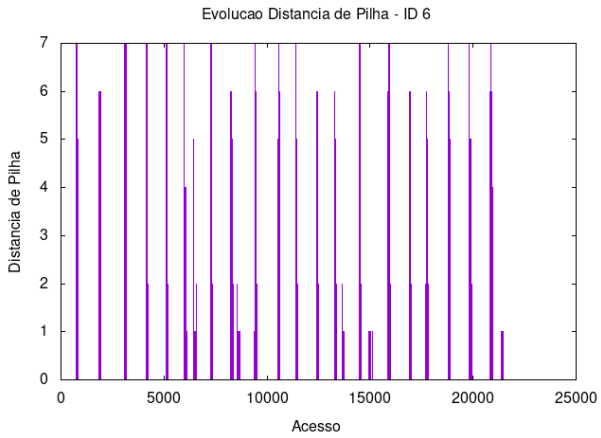

Evolucao Distancia de Pilha - ID 1
Distancia de Pilha
Acesso
0
1
2
3
4
5
6
7
8
0
5000
10000
15000
20000
25000

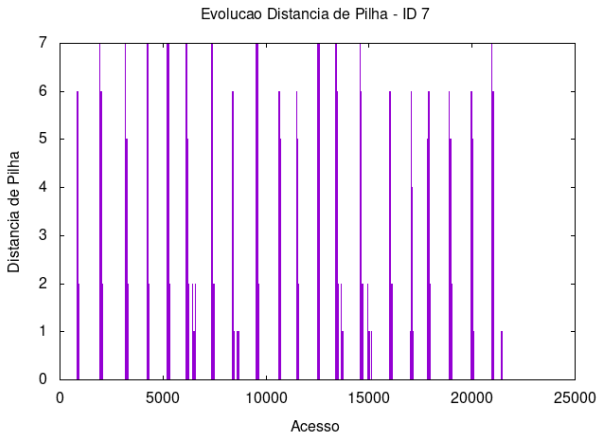

Evolucao Distancia de Pilha - ID 2
Distancia de Pilha
Acesso
0
1
2
3
4
5
6
7
8
0
5000
10000
15000
20000
25000

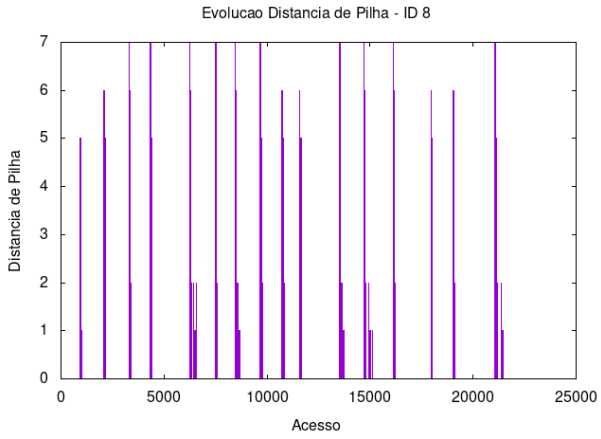

Evolucao Distancia de Pilha - ID 3
Distancia de Pilha
Acesso
0
1
2
3
4
5
6
7
8
0
5000
10000
15000
20000
25000

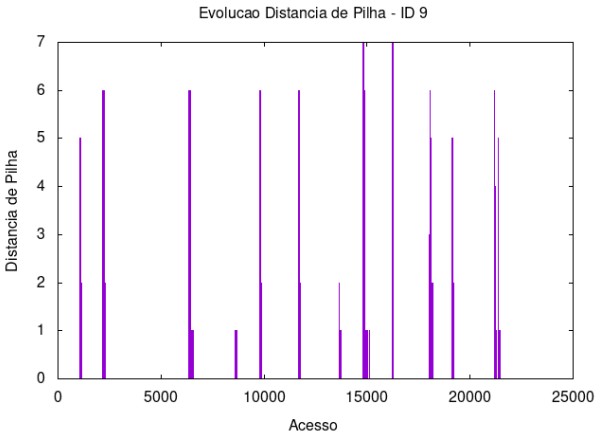

Evolucao Distancia de Pilha - ID 4
Distancia de Pilha
Acesso
0
1
2
3
4
5
6
7
0
5000
10000
15000
20000
25000


Evolucao Distancia de Pilha - ID 5
Distancia de Pilha
Acesso
0
1
2
3
4
5
6
7
0
5000
10000
15000
20000
25000



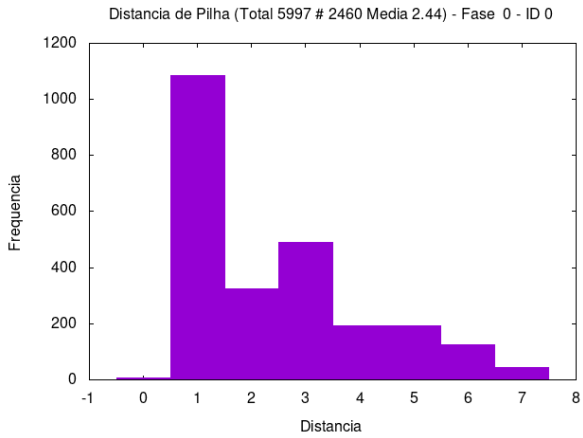


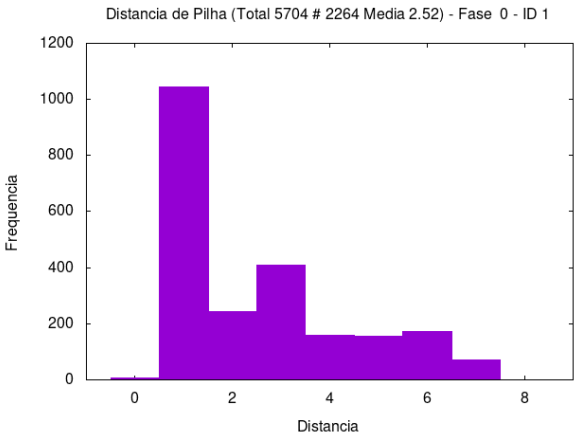


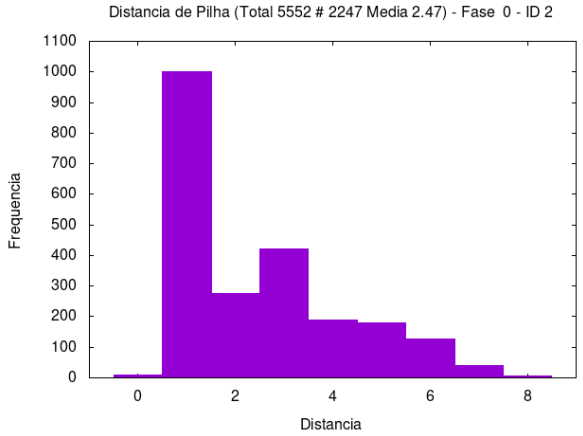


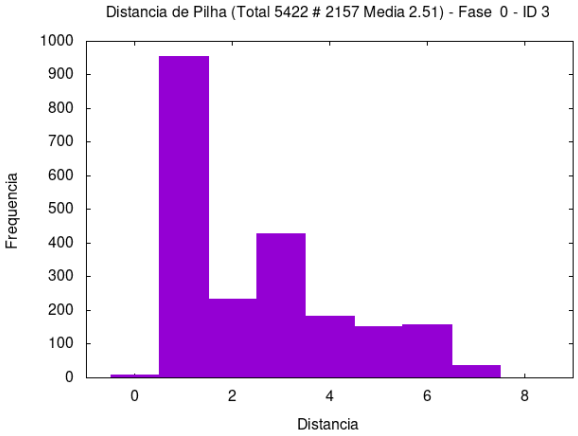
Da mesma forma em que foi possível notar quando que cada jogador participou da rodada, os gráficos de evolução de pilha também mostram esse fato. Outro ponto que é possível notar é o de que em geral acessos tendem a não serem superiores a 7 posições, isso pode se derivar do fato de que a grande maioria das manipulações é realizada sobre as cartas que cada jogador possui, esse número de cartas é fixo em 5, então, a maior parte da distância é provinda das manipulações das cartas. Outra analise cabível de ser realizada é uma junção das conclusões chegas no acesso da memória, podemos ver que como temos os acessos sendo feitos sequencialmente não há grande distancias nas quais é possível acessar conteúdo e o conteúdo em geral rodeia sobre as cartas ele está geralmente em poucas posições atras na pilha.

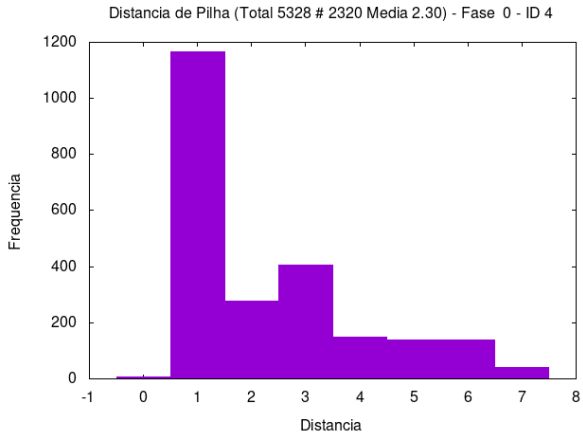
6.1.3 Distância De Pilha

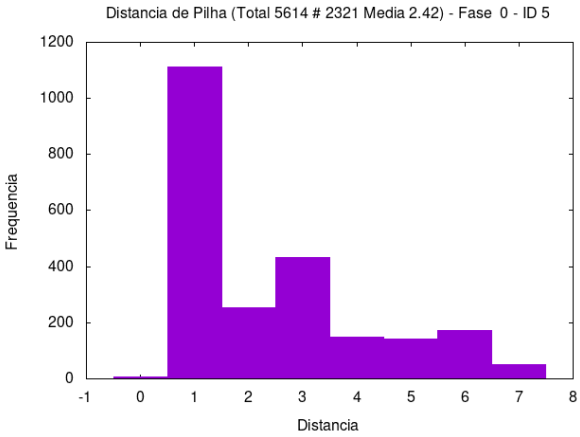

Distancia de Pilha (Total 5997 # 2460 Media 2.44) - Fase 0 - ID 0
Frequencia
Distancia
1200
1000
800
600
400
200
0
-1
0
1
2
3
4
5
6
7
8

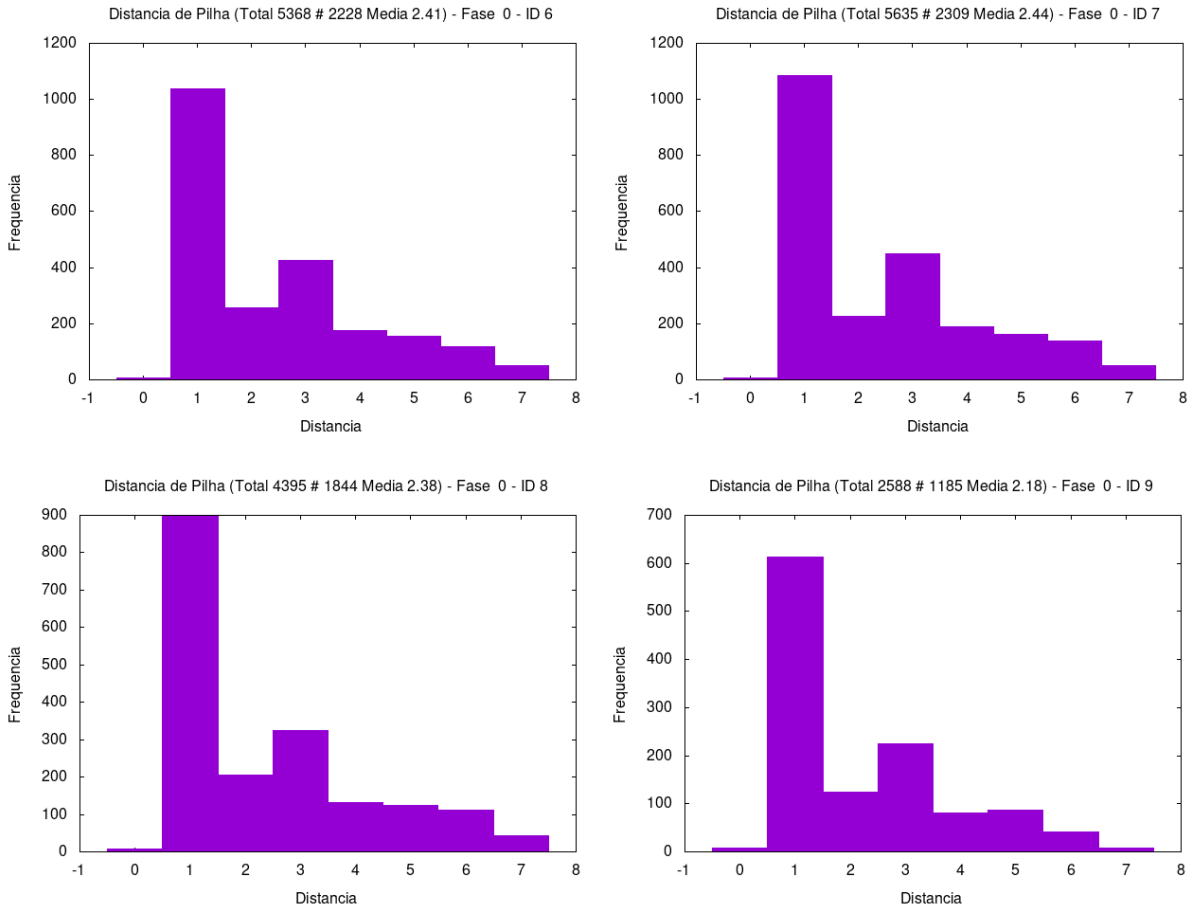

Distancia de Pilha (Total 5704 # 2264 Media 2.52) - Fase 0 - ID 1
Frequencia
Distancia
1200
1000
800
600
400
200
0
0
2
4
6
8


Distancia de Pilha (Total 5552 # 2247 Media 2.47) - Fase 0 - ID 2
Frequencia
Distancia
1100
1000
900
800
700
600
500
400
300
200
100
0
0
2
4
6
8


Distancia de Pilha (Total 5422 # 2157 Media 2.51) - Fase 0 - ID 3
Frequencia
Distancia
1000
900
800
700
600
500
400
300
200
100
0
0
2
4
6
8


Distancia de Pilha (Total 5328 # 2320 Media 2.30) - Fase 0 - ID 4
Frequencia
Distancia
1200
1000
800
600
400
200
0
-1
0
1
2
3
4
5
6
7
8


Distancia de Pilha (Total 5614 # 2321 Media 2.42) - Fase 0 - ID 5
Frequencia
Distancia
1200
1000
800
600
400
200
0
-1
0
1
2
3
4
5
6
7
8



Analisando a distância de pilha as conclusões chegadas na evolução se tornam mais claras. Podemos notas que a frequência é extremamente alta em posições de distância 1, o que pode ser concluído desse fato é que em geral os dados que são necessários serem acessados estão a 1 “acesso” de distância. Desse fato temos um menor consumo de memoria e tempo de execução, já que, os dados foram alocados sequencialmente, basta apenas acessar o conteúdo guardado na posicao seguinte. Outro dado interessante pé de que há uma drástica diminuição de acessos em relação a distância antes e depois de 1, mais acentuada na posicao 7.

6.1.4 Gprof

Ao realizar a analise gprof obtive todos os resultados gerados, uma das possíveis causas para esse resultado pode derivar do tempo de execução do programa que não consta na análise, outro motivo pode ser que a analise foi configurada de forma errada porém, outro caso pode ser o de que não é possível se extrair nenhuma expressão de tempo devido ao baixo tempo de execução. Existem diversos fatores que podem ser resultantes dessa impossibilidade de analise, porém não é possível afirmar com certeza qual deles é o culpado por esse infortuno. Então da análise gprof não é possível inferir nada.

6. Conclusão

Após a implementação do programa foi notório que ele pode ser visto de 4 momentos diferentes. A primeira foi de realizar as leituras e alocações corretamente dos atributos lidos do arquivo e distribui-los pela variáveis corretamente. A segunda é a de ser capaz de analisar a mão de cada jogador e determinar e atribuir uma classificação a cada um dos 10 tipos de mãos vencedoras disponíveis no jogo de poker usado como exemplo, e após definir uma classificação realizar a comparação com cada mão dos outros jogadores da rodada e determinar qual mão tem o maior peso e sera a vencedora. O terceiro é um ponto que não ocorre em todas as rodadas mas é de relevância para o programa que é de fazer o tratamento do empate e determinar quem é(foram) o(s) vencedor(es) da partida e delegar o valor do montante para ele(s). O quarto e final é de pegar o resultado de todas as partidas e imprimi-lo num arquivo além do placar dos jogadores ao final de todas as rodadas.

O grande ponto central deste trabalho era o de ser capaz de implementar funções e realizar estes procedimento de forma simples e objetiva, e que ao final da execução resultasse numa saída correta. O uso de técnicas de estruturas de dados foi

imprescindível para a execução deste trabalho, pois sem ela a implementação seria catastroficamente complexa e resultaria num programa pouco funcional. Pode-se notar que o custo para executar o programa não foi tão alto, parte deste custo computacional não ser elevado se deve ao fato que pouquíssimas funções possuem o custo não linear ou fixo, dado que ocorrem menos de 4 funções com o custo $O(n^2)$, poucas com o custo de $O(n)$ e a grande maioria com custo $O(1)$.

Portando é notório que a boa implementação e domínio das estruturas de dados, e a análise de custo são essenciais para que obtenhamos um programa o mais próximo do ideal, que realize o que é necessário de maneira requisitada, possua um custo computacional o mais baixo possível e seja simples de se entender. Requisitos os quais foram implementados no TP1.

7. Bibliografia

Ziviani, N. (2006). *Projetos de Algoritmos com Implementações em Java e C++:˜ Capítulo 3: Estruturas de Dados Básicas´* . Editora Cengage.

8. Instruções para compilação e execução

A compilação do código é feita através do Makefile, utilizando-se do comando make all, caso seja a primeira vez fazendo a execução ou somente make caso a execução já tenha sido feita anteriormente e os arquivos na pastas OBJ já tenham sido gerados. Para que a execução ocorra corretamente é necessário incluir na pasta diretório raiz do TP um arquivo nomeado entrada no formato txt, contendo os dados da partida, que são o número de rodadas, o valor de dinheiro dos jogadores, seguidos de uma quebra de linha. os dados de cada rodada o número de jogadores, o pingo dos jogadores, após esses dados uma quebra de linha e o nome, aposta e cartas que compõe a mão deste jogador, seguido de uma quebra de linha.