Trabalho Prático 1: Poker Face

Gabriel Franco Jallais

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

Belo Horizonte – MG – Brasil

gfjallais@ufmg.br

Entrega: 07/06/2022

1. Introdução

O problema proposto foi simular uma partida virtual de pôquer, onde o sistema seria responsável por determinar os vencedores das rodadas e o montante distribuído, sem que o usuário conheça necessariamente as regras do jogo de cartas, bastando apenas passar como entrada, o número de jogadores, o montante inicial de cada um, o número de rodadas que se deseja jogar, os jogadores que participarão e suas respectivas apostas e mãos.

Para tal, foram criadas diferentes classes, para representar a partida, as rodadas, os jogadores e as cartas. Além disso, foram criadas funções para realizar as ações necessárias para a realização do jogo.

Importante ressaltar que a entrada é fornecida por meio do arquivo ***entrada.txt***, em que, na primeira linha, há o número de rodadas (***n***) e o dinheiro inicial dos participantes, após isso, há ***n*** blocos, para representar cada rodada, onde na sua primeira linha, há o número de jogadores da rodada (***m***) e o pingo, em seguida, as próximas ***m*** linhas possuem os dados de cada jogador, isto é, seu nome, o valor de sua aposta e 5 cartas.

Nas próximas seções serão dados mais detalhes quanto a implementação, a análise de complexidade das operações, além de quais foram as estratégias de robustez adotadas, quais testes foram feitos, junto a análise experimental, por fim, é apresenta a conclusão, a bibliografia utilizada e instruções para compilação e execução.

2. Implementação

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

Foi compilado e executado no sistema operacional Ubuntu 20.04 dentro do WSL2 no Windows 11, com um processador i7-1165G7 com 16 GB de memória RAM.

2.1 Classes

2.1.1 Carta

Classe criada para representar as cartas do jogo.

Possui como atributos protegidos, seu naipe, armazenado como inteiro, da forma: {“P” = 0, “E” = 1, “C” = 2. “D” = 3} e seu número, também inteiro, de 1 a 13.

Atributos: num, naipe.

A classe possui como métodos:
carta(): Seu inicializador padrão, que inicializa seus atributos com -1.

carta(std::string *carta_str****):*** Seu inicializador que recebe uma std::string, de formato “1P” por exemplo, como parâmetro, e extrai da última posição o seu naipe e da substring, de 0 à penúltima posição, o seu número.

int get_num(): Retorna o número da carta.

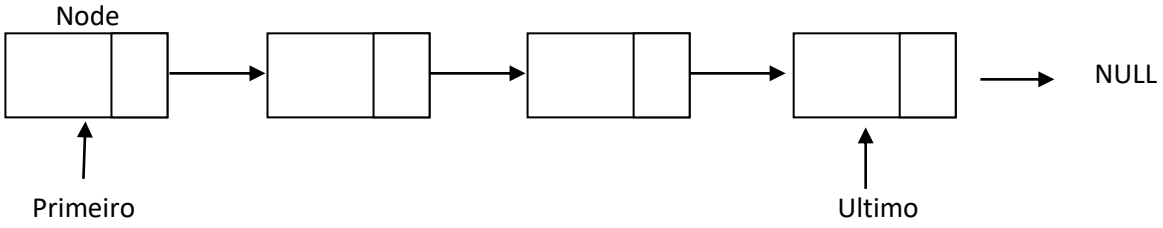
int get_naipe(): Retorna o naipe da carta.

bool operator==(carta *carta****):*** Retorna ***true*** se o número das cartas for igual e ***false*** caso contrário.

void set_carta(int *num****):*** Atribui ao numero da carta o valor *num* passado como parâmetro

2.1.2 Lista (Lista Encadeada – Estrutura de Dados) e Node

Classe criada para auxiliar algumas operações na classe jogador.



Node: Possui como atributos: um ponteiro para o próximo node, um item da classe carta, um inteiro quantidade, que guarda a quantidade de cartas com o mesmo número da presente no node.

Atributos: prox, item, quantidade.

A classe node possui como métodos apenas seu inicializador, que atribui NULL a prox e 1 a quantidade, e q_increment() que incrementa o atributo quantidade.

Lista: Possui como atributos: um ponteiro para o primeiro e para o último node da lista, e um inteiro tam que armazena seu tamanho.

Atributos: *primeiro, *último, tam.

A classe lista possui como métodos:

lista(): Inicializa primeiro com um new node(), atribui a ultimo o atributo primeiro e inicializa tam com 0

~lista(): Chama o método limpa() para desalocar a lista.

node* get_primeiro(): Retorna o ponteiro para a cabeça da lista, no caso, retorna o atributo primeiro.

carta get_item(int** pos**): Retorna o item/carta na posição *pos* da lista.

void set_item(carta** item**, int** pos**): Atribui item, passado como parâmetro, ao item na posição *pos* da lista.

void insere(carta** item**): Insere a carta passada como parâmetro na lista caso não hajam cartas com o mesmo número na lista, se houver, incrementa o atributo quantidade no node que possui a carta com o mesmo número.

node* pesquisa(carta ***item**): Pesquisa pela carta na lista, com o mesmo número que a passada.

void limpa(): Desaloca cada node da lista.

node* posiciona(int** pos**, bool** antes**): Retorna um ponteiro para o node na posição *pos*, caso *antes* seja *false*, e para o node anterior a *pos*, caso *antes* seja *true*.

int get_tam(): Retorna o tamanho da lista.

2.1.3 Jogador

Classe criada para representar os jogadores da partida.

Possui como atributos protegidos: seu nome em formato de string; um vetor de cartas de tamanho 5, para representar a sua mão; sua mão em formato de lista encadeada para facilitar algumas operações; um booleano para indicar se todas as cartas na sua mão são do mesmo naipe e um outro booleano para indicar se o jogador é vencedor ou não; seu montante, seu número de vitórias e a sua aposta, todos os três, inteiros; e por último, um vetor de inteiros com os dados da sua mão, sendo a primeira posição para a classificação da sua jogada, a segunda para o número da carta de maior número na sua mão, e a terceira e a quarta posições variam para cada jogada, mas de forma geral, guardam os valores de desempate para cada caso, a maior trinca, o maior par menor, etc.

Atributos: _nome, montante, v_mao[5], l_mao, msm_naipe, jogada[4], vencedor, num_vitorias, aposta.

A classe possui como métodos:
jogador(): O método apenas instância o objeto, não fazendo nenhum outra operação

jogador(std::string** nome**, int** dinheiro_inicial**, carta ***mao**): Inicializa o nome do jogador com a string passada como parâmetro, inicializa o montante do jogador com o *dinheiro_inicial* passado, inicializa a mão do jogador com os valores em *mao* e junto a isso determina se todas as cartas são do mesmo naipe e atribui o resultado ao atributo msm_naipe, do jogador, inicializa vencedor, num_vitorias e aposta com 0.

int get_montante(): Retorna o montante atual do jogador.

void alt_montante(int *n***):** Soma n ao valor do montante do jogador

std::*string* **get_nome():** Retorna o nome do jogador.

void set_vencedor(bool** b**): Atribui b ao atributo *vencedor*.

bool eh_vencedor(): Retorna o valor de *vencedor.*

void define_mao(): Define em qual categoria se enquadra a mão do jogador, atribui às posições do vetor atributo *jogada* a categoria da jogada, de 1 a 10, a maior carta da mão e outros critérios de desempate.

void set_aposta(int aposta): Atribui ao atributo *aposta* o valor passado como parâmetro.

int get_aposta(): Retorna o valor da aposta do jogador.

bool operator==(jogador** j**): Retorna *true* caso o nome do jogador seja igual ao comparado, e *false*, caso contrário.

int get_categoria_jogada(): Retorna a primeira posição do atributo *jogada*.

void set_mao(carta ***moa**): Atribui ao atributo *mao* a mao passada como parâmetro.

Além disso foram criadas funções auxiliares para a definição da mão, são elas:

bool seguido(lista ***p**): Apenas determina se as cartas na mão do jogador possuem números consecutivos.

int max_element(lista ***p**): Retorna o número da carta de maior número na lista passada.

2.1.4 Rodada

Classe criada para representar cada rodada da partida.

Possui como atributos protegidos: cinco inteiros, seu pote, o número de jogadores da rodada, o número de jogadores na partida, o número de vencedores, e o pingo da rodada, e um booleano para dizer se a rodada é válida ou não. Como atributo público, a classe possui um vetor de jogadores de tamanho 10.

Atributos: pote, num_jogadores_r, num_jogadores_p, num_vencedores, pingo, valid, jogadores_rodada[10].

rodada(): Inicializa todos os seus atributos inteiros com 0, e *valid* com *true*.

void arrecadar(): Arrecada dinheiro de todos os jogadores ao final da rodada, no caso, tira o valor referente ao pingo e a aposta do montante de cada jogador, e caso o jogador não tenha dinheiro suficiente para o arrecadamento a rodada é invalidada.

void def_vencedores(): De acordo com as regras do jogo, o método determina os vencedores, e utiliza dos métodos desempate_maior_carta() e *desempate_maior_conjunto(),* para tentam desempatar a rodada, caso haja múltiplos vencedores.

void desempate_maior_carta(): Itera sobre os atuais vencedores empatados e determina os vencedores com base na maior carta da mão de cada um.

void desempate_maior_conjunto(): Itera sobre os atuais vencedores empatados e determina os vencedores com base no maior conjunto da mão de cada um, isto é, por exemplo, maior tripla, maior dupla, maior quadra.

void distribuir_premiacao(): De acordo com o número de vencedores distribui o valor do pote da rodada igualmente para todos os vencedores.

int get_pote(): Retorna o valor do *pote.*

int get_num_j_r(): Retorna o número de jogadores da rodada em questão.

int get_num_j_p(): Retorna o número de jogadores da partida em questão.

int get_num_v(): Retorna o número de vencedores da rodada em questão.

void set_num_j_r(int** nj**): Atribui ao atributo *num_jogadores_r* o valor *nj* passado como parâmetro.

void set_num_j_p(int** nj**): Atribui ao atributo *num_jogadores_p* o valor *nj* passado como parâmetro.

void set_pingo(int** p**): Atribui ao atributo pingo o valor *p* passado.

bool is_valid(): Retorna o valor do atributo *valid*.

void set_valid(bool** b**): Atribui ao atributo *valid* o valor *b* passado.

void set_num_v(int** nv**): Atribui ao atributo *num_vencedores* o valor *nv* passado.

2.2 Funções do jogo

Essas funções estão presents no arquivo jogo.cpp, e são responsáveis pela execução do jogo, assim como a validação das rodadas e ordenação dos jogadores e do montante.

void selectionSort_m (jogador arr[], int n)
void selectionSort_n (jogador arr[], int n)
void swap(jogador *xp, jogador *yp)

A função *swap(jogador *xp, jogador *yp)* funciona em conjunto com as as outras duas, ambas ordenam o vetor de tamanho *n* passado utilizando o algoritmo selection sort, porém a primeira ordena os jogadores pelo montante, enquanto a segunda ordena pelos nomes dos jogadores, para que assim a saída seja conforme desejado e especificado no enunciado.

bool valida_entrada(std::string entrada)

Apenas valida se o arquivo de nome *entrada*, segue as especificações descritas no enunciado e, para isso, são usadas expressões regulares da biblioteca regex.

3. Análise de Complexidade

Para a análise de complexidade do programa, serão levadas em conta apenas as funções referentes às operações de ordenação, validação do documento de entrada. Será feita apenas a análise de complexidade em relação ao tempo de execução, visto que até então, a análise de complexidade de espaço não foi tão bem abordada em sala.

- void selectionSort_m (jogador arr[], int n)
void selectionSort_n (jogador arr[], int n)

Como visto em sala, e como é facilmente provável, o algoritmo de ordenação selection sort, possui complexidade assintótica $O(n^2)$, visto que os vetores ordenados são de tamanho 10, utilizei tal algoritmo visto a sua fácil implementação.

- bool valida_entrada(std::string entrada)

A função valida_entrada(std::string entrada) possui um loop duplo de for para verificar as linhas do arquivo, se estão de acordo com as especificações, para efetuar a análise, a operação relevante será a regex_match(), isto é, a comparação da linha do arquivo com a expressão regular que ela deveria seguir. Uma comparação é feita fora do loop, dentro do loop mais externo são feitas *n* regex_match(), onde *n* é o número de rodadas da partida, e dentro do loop mais interno são feitas no máximo 10 regex_match(), visto que o limite de jogadores por rodada é 10. Portanto a complexidade da função em questão é $O(10n + 1) = O(n)$.

- A execução do jogo

Como há apenas um loop que depende do número de rodadas, única entrada que pode escalar, a complexidade da execução do jogo é $O(n)$.

Como as demais função não dependem do número de rodadas, que seria a única entrada em que n poderia ficar grande, todas as outras funções possuem complexidade constante, isto é, $O(1)$.

4. Estratégias de Robustez

Para garantir a boa operação do programa, isto é, que ele seja robusto para tolerar erros do usuário e se defender de tentativas de "ataque" que visam achar falhas no programa para explorá-las, foram tomadas as seguintes estratégias.

Primeiramente, é feita a checagem dos arquivos de entrada, se este abriu corretamente, e se este segue as especificações definidas, para isso, são usadas as funções valida_entrada(std::string entrada) e erroAssert(e,m) da biblioteca msgassert.h. Também é checado se o arquivo de saída abre corretamente.

- ***Testes de Sanidade***

Para os testes de sanidade, é checado, primeiramente, se o pingo da rodada é maior ou igual a 50, e em seguida, checa-se se a aposta dos jogadores são não nulas e múltiplas de 50, após isso, dentro do método arrecadar() da classe rodada, verifica-se se os jogadores são capazes de pagar o pingo e as suas apostas. Caso as condições acima não sejam cumpridas, a rodada é invalidada.

Ademais, para cada função/operação descrita na seção 2, são feitas checagens para que as entradas cumpram os requisitos para as operações requisitadas, isto é, no caso dos métodos posiciona, get_item, set_item da classe lista, faz-se a checagem do valor do parâmetro pos passado, para que não haja a tentativa de acesso a posições inexistentes na lista, e, consequentemente, um erro de segmentation fault seja gerado.

5. Testes

Para analisar desempenho computacional do programa, além de fazer a análise do padrão de acesso à memória e localidade de referência, foram feitos os seguintes testes:

Para a análise de desempenho, a análise de memória não foi ativada, para melhor fidelidade e representação do desempenho do programa, e foram dadas entradas de 100 a 750 rodadas indo de 50 em 50.

Já para a análise de acesso e localidade de memória, foi utilizada uma entrada pequena, com apenas 3 rodadas.

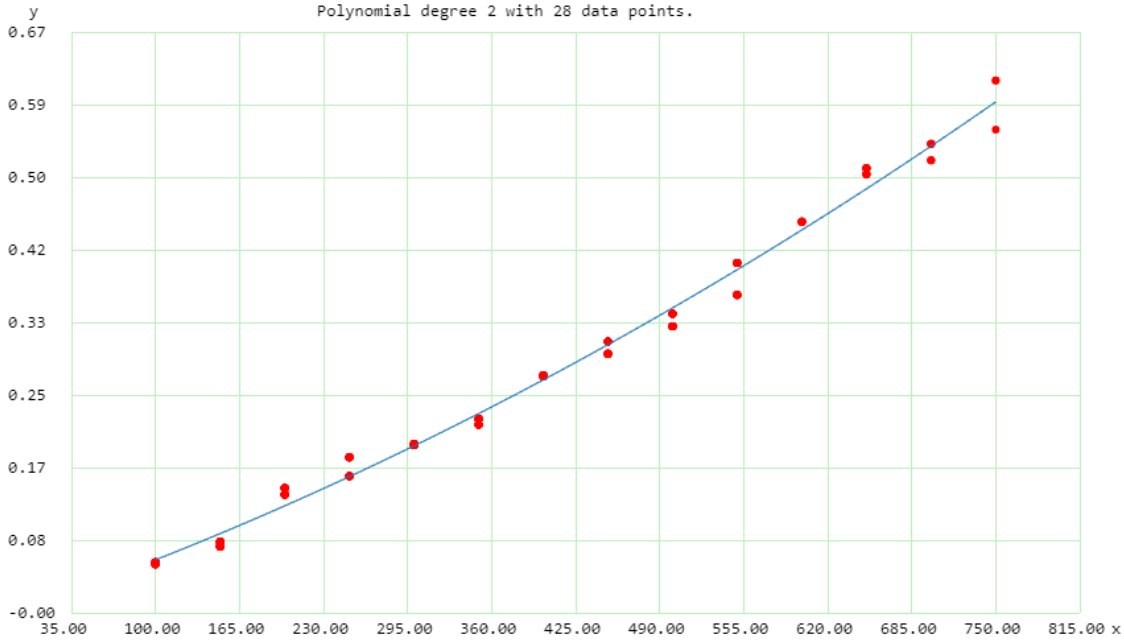
Os casos de teste foram gerados por meio do programa script.py, presente na pasta TP1.zip.

6. Análise Experimental

Para efetuar a análise do programa em questão foram utilizados os softwares memlog e analisamem, que foram responsáveis por mapear o tempo de execução e o acesso à memória. As operações consideradas relevantes para a avaliação foram as mesmas descritas na seção 3.

6.1 Desempenho Computacional

Para verificar a análise de complexidade feita previamente, foi feita uma sequência de testes e traçada uma curva para capturar o comportamento do programa em função do aumento da entrada, segue o gráfico:



Os resultados para cada entrada são oscilantes devido ao hardware e às outras atividades que estavam sendo executadas em paralelo no computador, levando isso em consideração foram feitos alguns testes para tentar melhorar a precisão da análise.

6.1.1 Resultados do gprof

Para analisar os resultados do gprof, utilizaremos uma entrada com 10000 rodadas, para haver uma visualização mais direta do funcionamento do programa e para que o tempo de execução seja grande o suficiente para gerar resultados visíveis.

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
2.27	0.32	0.03				std::function<bool (char)>::operator()(char) const
1.52	0.52	0.02	250000	0.00	0.00	int __gnu_cxx::__stoa<long, int, char, int>(long (*)(char const*, char**, int), char const*, char const*, unsigned long*, int)
1.52	0.54	0.02				lista::limpa()
1.52	0.90	0.02				main
0.76	0.91	0.01	250000	0.00	0.00	Save_errno::~_Save_errno()
0.76	0.92	0.01	50000	0.00	0.00	jogador::define_mao()

```
  0.76     0.93     0.01       1    10.00    10.00  rodada::rodada()
  0.76     0.94     0.01                     lista::insere(carta)
  0.76     0.95     0.01                     lista::get_tam()
  0.76     0.96     0.01                     lista::lista()
  0.38     1.31     0.01                     bool __gnu_cxx::__is_null_pointer<char>(char*)
  0.00     1.32     0.00  2050000     0.00     0.00  carta::get_num()
  0.00     1.32     0.00   999910     0.00     0.00  jogador::get_nome[abi:cxx11]()
  0.00     1.32     0.00   750000     0.00     0.00  carta::operator==(carta)
  0.00     1.32     0.00   669902     0.00     0.00  rodada::get_num_j_p()
  0.00     1.32     0.00   380125     0.00     0.00  carta::carta()
  0.00     1.32     0.00   300000     0.00     0.00  carta::get_naipe()
  0.00     1.32     0.00   250000     0.00     0.00
carta::carta(std::__cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> >)
  0.00 1.32   0.00   250000     0.00     0.00  std::__cxx11::stoi
  0.00     1.32     0.00   250000     0.00     0.00   _gnu_cxx::__stoa
  0.00     1.32     0.00   250000     0.00
0.00__gnu_cxx::__stoa_Save_errno::_Save_errno()
  0.00     1.32     0.00   205000     0.00     0.00  jogador::eh_vencedor()
  0.00     1.32     0.00   150000     0.00     0.00  jogador::get_aposta()
  0.00     1.32     0.00   100000     0.00     0.00  jogador::alt_montante(int)
  0.00     1.32     0.00   100000     0.00     0.00  jogador::set_vencedor(bool)
  0.00     1.32     0.00    50100     0.00     0.00  jogador::get_montante()
  0.00     1.32     0.00    50000     0.00     0.00  max_element(lista*)
  0.00     1.32     0.00    50000     0.00     0.00  seguido(lista*)
  0.00     1.32     0.00    50000     0.00     0.00  jogador::set_aposta(int)
  0.00     1.32     0.00    49990     0.00     0.00  jogador::set_mao(carta*)
  0.00     1.32     0.00    20000     0.00     0.00  rodada::is_valid()
  0.00     1.32     0.00    10000     0.00     0.00  rodada::get_num_v()
  0.00     1.32     0.00     5010     0.00     0.00  jogador::jogador()
  0.00     1.32     0.00     5000     0.00     0.00  rodada::set_num_j_r(int)
  0.00     1.32     0.00     5000     0.00     0.00  rodada::def_vencedores()
  0.00     1.32     0.00     5000     0.00     0.00  rodada::distribuir_premiacao()
  0.00     1.32     0.00     5000     0.00     0.00  rodada::desempate_maior_carta()
  0.00     1.32     0.00     5000     0.00     0.00  rodada::get_pote()
  0.00     1.32     0.00     5000     0.00     0.00  rodada::arrecadar()
  0.00     1.32     0.00     5000     0.00     0.00  rodada::set_num_v(int)
  0.00     1.32     0.00     5000     0.00     0.00  rodada::set_pingo(int)
  0.00     1.32     0.00     5000     0.00     0.00  rodada::set_valid(bool)
  0.00     1.32     0.00     5000     0.00     0.00  jogador::get_categoria_jogada()
  0.00    1.32     0.00       10     0.00     0.00
jogador::jogador(std::__cxx11::basic_string<char, std::char_traits<char>,
std::allocator<char> >, int, carta*)
  0.00     1.32     0.00        1     0.00     0.00  _GLOBAL__sub_I__ZN6rodadaC2Ev
  0.00     1.32     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int,
int)
  0.00     1.32     0.00        1     0.00     0.00  rodada::set_num_j_p(int)
```

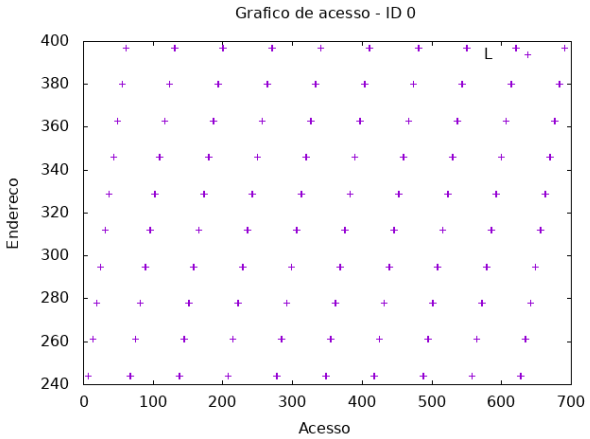
*Alguns dados foram retirados ou tiveram seus nomes reduzidos do resultado para fins de apresentação, como é o caso das operações que envolvem

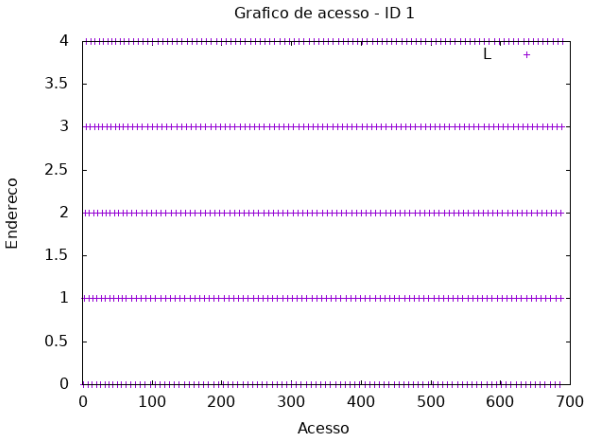
regex, com elas no resultado seria impossível conte-las no espaço requisitado para a documentação, além da interpretação extremamente dificultada. Mantiveram-se todas as funções referentes ao programa em si, citadas em seções anteriores.

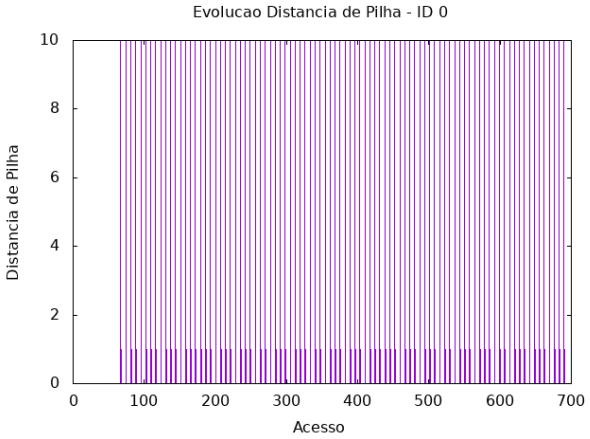
6.2 Análise de Padrão de Acesso à Memória e Localidade de Referência

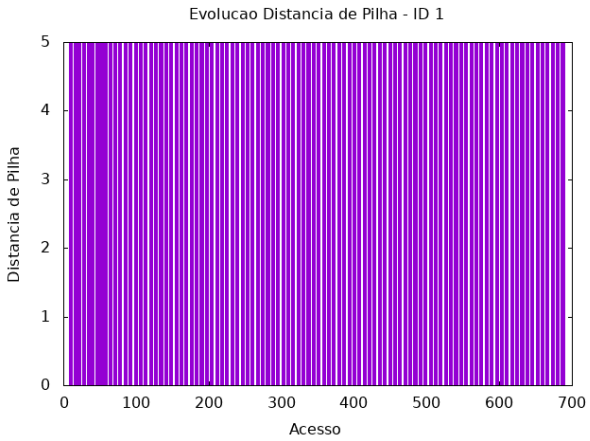
Agora, para entender melhor como o programa acessa memória e a sua localidade de referência, observaremos os gráficos gerados por meio do software gnuplot, e cujos dados foram fornecidos pelo programa analisamem, utilizando uma partida de 10 rodadas, para uma visualização menos poluída.

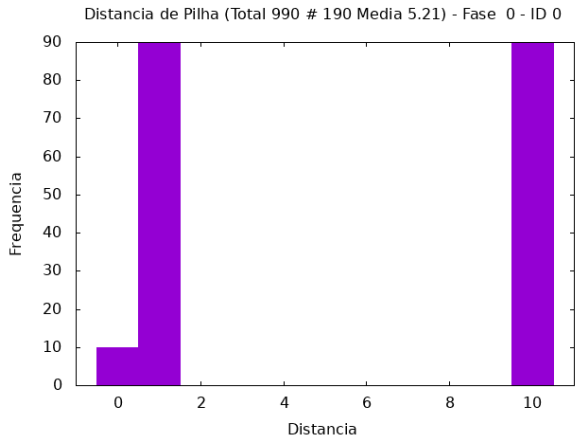
Primeiramente, cabe dizer que, o id 0 é referente ao vetor de jogadores de cada rodada, e o id 1 o vetor de cartas de cada jogador,. Os gráficos mostram como foi feito o acesso à memória ao decorrer do tempo, pontos (+) próximos no eixo x indicam endereços que foram acessados em intervalos próximos de tempo.

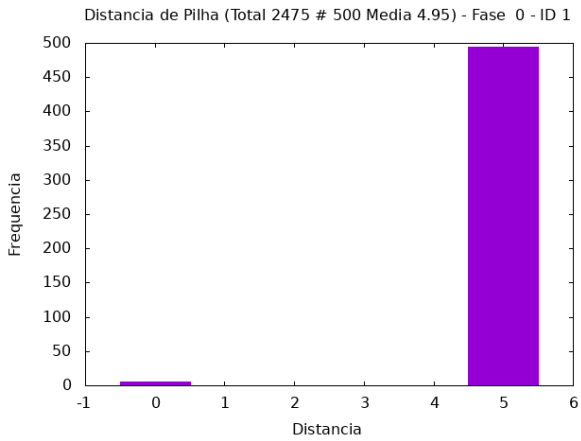












Como as únicas estruturas de dados com as quais foi possível efetuar uma análise de localidade de memória foram vetores estáticos, é possível observar um padrão muito claro e bem linear no acesso à memória, assim como a distância de pilha, que possui pequenas variâncias no ID 0, visto que, em alguns momentos, somente as posições cujo jogador é vencedor é acessada, mas de forma geral, os vetores são acessados até o seu tamanho máximo, no caso das cartas será sempre 5, já no caso dos jogadores, será variável, conforme a entrada, no caso na entrada, 10 jogadores participam da partida.

7. Conclusões

O trabalho consistiu em simular uma partida de pôquer, utilizando os conhecimentos obtidos em sala, além realizar a análise do desempenho em relação ao tempo de execução e o acesso à memória conforme o tamanho da entrada cresce, nesse caso, o número de rodadas da partida, visto as restrições da especificação.

Por se tratar de uma tarefa relativamente simples, não houve grandes dificuldades para a realização do tal, sendo a tarefa de análise a mais laboriosa. Pôde-se observar como as funções seguem a análise assintótica feita na seção 2, e como o acesso a memória e a distância de pilha foram coerentes com a implementação feita para as operações e coerentes com as entradas fornecidas.

8. Bibliografia

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Algoritmos: Teoria e Prática. 3a edição. Elsevier, 2012. ISBN 9788535236996

Nivio Ziviani. Projeto de Algoritmos com implementações em Pascal e C. 3ª edição. Cengage Learning, 2011. ISBN 9788522110506

Apêndice

Instruções para compilação e execução

Para facilitar a compilação e execução do programa jogo.cpp, foi disponibilizado um arquivo Makefile com comandos mais simples.

Para compilar o programa, basta possuir o arquivo Makefile no mesmo diretório que os outros programas, estruturados de forma:

|- src
|- bin
|- obj
|- include
Makefile
entrada.txt

Onde src deve possuir os arquivos .cpp e include deve possuir os arquivos .h e executar o comando “make comp”.

Para executar o programa basta executar o comando na linha de comando no seguinte formato:

(EXE) (COMANDO OPICIONAL 1) (SAÍDA DO COMANDO OPCIONAL) (COMANDO OPICIONAL 2) -1 (ARQUIVO DE ENTRADA)
EXE = tp1.exe

COMANDO OPICIONAL 1 = -p, para a geração de arquivo .out com dados de desempenho computacional
COMANDO OPICIONAL 2 = -l, para a geração de mapas de acesso a memória
ARQUIVOS DE ENTREDA = entrada.txt

Para realizar as experimentações vistas na documentação basta executar o comando “make mem”, para gerar os mapas de acesso e histogramas, o comando “make perf”, para gerar arquivos .out para a análise de desempenho, e o comando “make gprof”, para gerar arquivos .gprof para a análise de desempenho. ***É necessário possuir o programa analisamen.c**

Por fim, o comando “make clean”, pode ser utilizado para apagar os arquivos .o e gmon.out gerados