Trabalho Prático 1: Poker Face

Pedro de Oliveira Guedes
2021040008

DCC/UFMG

Introdução

A documentação aqui presente se refere a um programa produzido na linguagem de programação C + +, com o objetivo de implementar o fluxo básico de execução de um jogo de poker.

O programa é estruturado para funcionar na seguinte ordem: leitura do arquivo com os dados da partida; alocação das rodadas; alocação dos jogadores para as rodadas; reconhecimento das apostas dos jogadores; distribuição das cartas para as mãos de todos os jogadores; verificação do vencedor da rodada; distribuição do prêmio (pote) da rodada e, por fim, classificação final dos jogadores da partida.

Mais detalhes sobre a implementação do jogo serão dados no decorrer deste documento.

Método

Nesta seção, serão dadas informações sobre a organização do programa em pastas, bem como o que cada uma abriga, formatos de entrada e saída, configurações do sistema de testes, etc..

Configurações do sistema de testes

O programa foi implementado na linguagem C + +, compilado pelo G + +, compilador da GNU Compiler Collection. Além disso, foi utilizado o sistema operacional Ubuntu 9.3.0-17 ubuntu1~20.04 em um Windows 10, através do Windows Subsystem for Linux (WSL 2), com um processador AMD Ryzen 3 3200G (3.60GHz, 4 CPUs) e 13 GB RAM.

Formato de entrada e saída

O formato de entrada dos dados é através de um arquivo de extensão “.txt” seguindo o padrão:

```
num_de_rodadas dinheiro_inicial
num_jogadores_rodada_atual pingo_rodada
nome_jogador1 aposta1 carta1 carta2 carta3 carta4 carta5
nome_jogador2 aposta2 carta1 carta2 carta3 carta4 carta5
```

```
.
.
.
```

O formato de saída do programa também é através de um arquivo de extensão “.txt”, que segue o padrão:

```
num_vencedores_rodada1 pote_rodada1 combinação_vencedora_rodada1
nome_vencedor1
nome_vencedor2
.
.
.
num_vencedores_rodada2 pote_rodada2 combinação_vencedora_rodada2
.
.
.
####
jogador_com_mais_fichas quantidade_de_fichas_jogador
segundo_jogador_com_mais_fichas quantidade_de_fichas_jogador
.
.
.
```

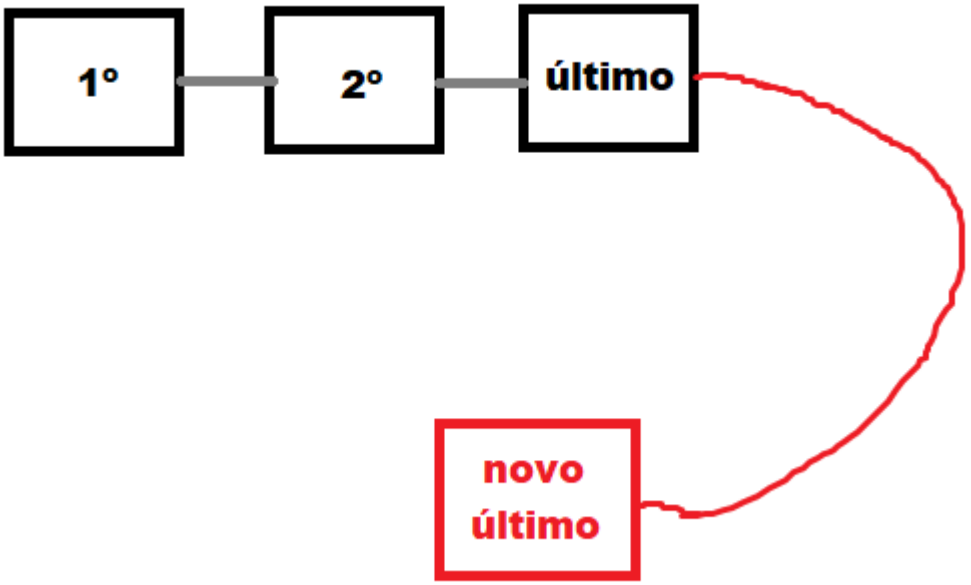
Há algumas entradas consideradas inválidas. Elas geral linhas de saída um pouco diferentes das mostradas acima. Quando uma rodada inválida é detectada pelo programa, ela imprime no arquivo de saída “*0 0 I*”.

Entradas inválidas que invalidam rodadas serão melhor abordadas neste documento na seção de robustez do código.

Estruturas de Dados

Essencialmente, o único Tipo Abstrato de Dados (TAD) implementado foi o de “fila”. Ele segue o princípio *FIFO* (*First In First Out*, que quer dizer “Primeiro a entrar, primeiro a sair” em português) e foi implementada utilizando conceitos de herança em duas classes diferentes: “*BetsChainedQueue*” e “*RoundsChainedQueue*” que enfileiram, respectivamente, as apostas feitas pelos jogadores em cada rodada e as rodadas das partidas.

Foi utilizada a fila em sua implementação encadeada. Dessa forma, novos elementos são simplesmente adicionados ao fim, sem possui conexão alguma com os outros.



Organização de pastas

A pasta raiz do projeto se chama "*01-poker_face*" e abriga, além das subpastas diretamente relacionadas ao programa, a subpasta "analisamem", que não será discutida neste documento, já que foi criada pelos professores da disciplina.

Além desta, a pasta também abriga as subpastas a seguir:

1. **bin:** Abriga todos os arquivos binários gerados pelo programa.
2. **include:** Inclui todos os arquivos de cabeçalho utilizados na implementação. Entre eles: *bets_queue.hpp*, *card.hpp*, *deck.hpp*, *hand.hpp*, *match.hpp*, *msgassert.hpp*, *player.hpp*, *round.hpp* e *round_queue.hpp*.
3. **obj:** Arquivos de objeto gerados para a compilação do programa principal.
4. **src:** Todos os arquivos de código fonte para a implementação do programa. Sendo eles: *bets_queue.cpp*, *card.cpp*, *deck.cpp*, *hand.cpp*, *match.cpp*, *msgassert.cpp*, *player.cpp*, *round.cpp*, *round_queue.cpp* e *poker.cpp*.

Análise de complexidade

A análise de complexidade contemplará as principais funções implementadas nos arquivos de código fonte do programa. Para todas as funções, serão consideradas operações relevantes:

- Complexidade de tempo: leitura e atribuição de valores.
- Complexidade de espaço: alocação de memória dinâmica, recebimento de estruturas dinâmicas e chamadas de funções.

A presente análise de complexidade leva em consideração a notação de complexidade "*grande O*". A título de referência, a quantidade de rodadas na partida será denotada por "*n*" e a quantidade de jogadores em cada rodada, ou mesmo na partida, será denotada por "*m*".

Card(const std::string &cardCode):

Esta função constrói um novo objeto do tipo carta. Faz uma série de atribuições que não dependem do tamanho da entrada no geral, portanto, temos complexidade de tempo: $O(1)$ e complexidade de espaço: $O(1)$.

sortHand():

Esta função implementa o algoritmo "*selection sort*" para ordenar as cartas presentes no vetor estático da classe mão. Apesar de este algoritmo ser conhecidamente $O(n^2)$, sabemos que "*n*" é sempre igual a 5. Portanto, temos complexidade de tempo: $O(1)$ e complexidade de espaço: $O(1)$.

Hand::()*:

Todas as funções da classe "*Hand*" não dependem muito da entrada, já que, assim como foi dito na função anterior, todas são relativas ao vetor estático de cartas que possui sempre 5 posições. Portanto, temos complexidade de tempo: $O(1)$ e complexidade de espaço: $O(1)$ para todos os métodos dessa classe.

initializeDeck():

Esta função itera pelo vetor estático de cartas da classe inicializando o baralho do jogo. Um baralho completo possui, conhecidamente, 52 cartas. Portanto, já que a quantidade de cartas não é alterada de acordo com parâmetros externos, temos complexidade de tempo: $O(1)$ e complexidade de espaço: $O(1)$.

Deck::()*:

Todas as funções da classe "*Deck*" não dependem muito da entrada, já que, assim como foi dito na função anterior, todas são relativas ao vetor estático de cartas que possui sempre 52 posições. Portanto, temos complexidade de tempo: $O(1)$ e complexidade de espaço: $O(1)$ para todos os métodos dessa classe.

Player::()*:

Nenhum dos métodos da classe "*Player*" realiza iterações. Além disso, todas as chamadas de outras funções são para métodos já documentados acima que, como foi mostrado, são sempre $O(1)$ tanto para complexidade de espaço quanto para de tempo. Portanto, temos complexidade de tempo: $O(1)$ e complexidade de espaço: $O(1)$ para todos os métodos dessa classe.

BetsChainedQueue::()*:

Os métodos dessa classe são de inserção ou remoção na fila, operações não muito custosas em termos de tempo, já que inserções são feitas sempre no final, independentemente da quantidade de itens e remoções no início, da mesma maneira. Portanto, temos complexidade de tempo: $O(1)$ para todos os métodos dessa classe. Já a complexidade de espaço varia de acordo com a quantidade de itens enfileirados. Como essa é uma classe para enfileirar as apostas dos jogadores num determinado round, e a quantidade de jogadores no round é denominada por "*m*", temos então a complexidade de espaço: $O(m)$ para todos os métodos dessa classe.

RoundsChainedQueue::()*:

Os métodos dessa classe são de inserção ou remoção na fila, operações não muito custosas em termos de tempo, já que inserções são feitas sempre no final, independentemente da quantidade de itens e remoções no início, da mesma maneira. Portanto, temos complexidade de tempo: $O(1)$ para todos os métodos dessa classe. Já a complexidade de espaço varia de acordo com a quantidade de itens enfileirados. Como essa é uma classe para enfileirar os rounds de uma determinada partida, e a quantidade de jogadores no round é denominada por "*n*", temos então a complexidade de espaço: $O(n)$ para todos os métodos dessa classe.

collectBets():

Este método itera por todos os jogadores envolvidos na rodada, coletando as apostas de cada um deles. Dessa forma, o método possui complexidade de tempo e de espaço $O(m)$, onde "*m*" é a quantidade de jogadores na rodada.

sortEnrolledPlayers():

Este método itera por todos os jogadores envolvidos na rodada, ordenando o array que os contém de forma decrescente, através do algoritmo "*selection sort*" ou "ordenação por seleção". Dessa forma, o método possui complexidade de tempo $O(m^2)$ e de espaço $O(m)$, onde "*m*" é a quantidade de jogadores na rodada.

decideWinningPlayers():

Este método itera por todos os jogadores envolvidos na rodada, verificando quais deles possui a combinação vencedora (obrigatoriamente a que o primeiro jogador do array possui). Dessa forma, o método possui complexidade de tempo e de espaço $O(m)$, onde "*m*" é a quantidade de jogadores na rodada.

*Round::*TieBreaker()*:

Todos os métodos que possuem o sufixo "*TieBreaker*" na classe round estão sempre realizando operações, sejam elas de iteração ou não, sobre a quantidade de jogadores vencedores. Sabemos que a quantidade de vencedores "*w*" é um número $0 \leq w \leq m$, dependendo se a rodada é inválida (*0*) ou se todos os jogadores empataram (*m*). Dessa forma, o método possui complexidade de tempo e de espaço $O(m)$, onde "*m*" é a quantidade de jogadores na rodada.

transferPotCoinsToWinners():

Este método itera sobre a quantidade de jogadores vencedores. Foi explicado no método acima o intervalo para o número de vencedores. Porém, antes disso, chama o método "*tieBreaker()*", que por sua vez chama vários outros, incluindo o "*sortEnroledPlayers*", cuja complexidade já foi estudada. Dessa forma, o método possui complexidade de tempo $O(m^2)$, e de espaço $O(m)$, onde "*m*" é a quantidade de jogadores na rodada.

Match::Match(std::string matchDataPath):

O método construtor da classe partida (*Match*) é responsável por ler o arquivo com as entradas do jogo, que é recebido como parâmetro do método através da variável "*matchDataInput*", e orquestrar todos os elementos da partida, como a construção das rodadas por exemplo. Ele lê a quantidade de rodadas presente, denotada por "*n*" e itera de acordo com o número informado pelas linhas seguintes, lendo, para cada rodada, a quantidade de participantes, denotada por "*m*" e adicionando estes jogadores às rodadas.

Da mesma forma que o método anterior, ele faz uma série de chamadas de outros métodos, um deles é "*transferPotCoins*", cuja complexidade foi estudada acima. Sendo assim, temos que a complexidade de tempo é $O(n * m^2)$, e a complexidade de espaço é $O(n * m)$.

sortPlayersByCoins():

Este método ordena os jogadores da partida decrescentemente, através do algoritmo "*selection sort*", de acordo com a quantidade de fichas que cada um possui. Sendo assim, temos que a complexidade de tempo é $O(m^2)$, e a complexidade de espaço é $O(m)$, onde "*m*" é a quantidade de jogadores da partida.

play(std::string matchResultsPath):

Este método desempilha as partidas empilhadas pelo construtor da classe "*Match*", exibindo os resultados de cada uma delas no arquivo de texto recebido por "*matchResultsPath*". Logo após exibir os resultados das partidas, exibe também o ranking da partida, que consiste em iterar pelos jogadores, mostrando seus nomes e moedas. O método "*sortPlayersByCoins*", que foi estudado no tópico anterior, é chamado.

Sendo assim, temos que a complexidade de tempo é $O(n + m^2)$, e a complexidade de espaço é $O(n + m)$, onde "*m*" é a quantidade de jogadores da partida e "*n*" é a quantidade de rodadas da partida.

chargeOpeningBet(int openingBet):

Este método itera por todos os jogadores da rodada, desempilhando as apostas e cobrando-as de cada um deles. Sendo assim, temos que a complexidade de tempo é $O(m)$, e a complexidade de espaço é $O(m)$, onde "*m*" é a quantidade de jogadores da partida.

findPlayer(std::string playerName):

Este método itera pelos jogadores da rodada, verificando qual deles tem nome igual ao recebido no parâmetro "*playerName*". Sendo assim, temos que a complexidade de tempo é $O(m)$, e a complexidade de espaço é $O(m)$, onde "*m*" é a quantidade de jogadores da partida.

Obs:

As outras funções não serão analisadas quanto à complexidade porque se tratam de métodos que não são particularmente relevantes para o programa.

Estratégias de Robustez

A robustez em um programa é de crucial importância para o bom funcionamento do mesmo. Nesse sentido, uma série de verificações de entrada do usuário se fazem necessárias, para que não haja execuções errôneas ou estouros grosseiros de erro no programa. Para isso, utilizei a biblioteca *msgassert.h*, também presente nos arquivos deste trabalho.

As estratégias serão listadas e explicadas com relação à relevância de utilização.

- **Testes de sanidade:** Sempre que um novo jogador é adicionado à rodada, é realizado um teste de sanidade, para verificar se a aposta e o pingo da rodada não excedem a quantidade de fichas que o jogador possui. Caso exceda, a rodada é invalidada e o fluxo do programa segue normalmente.
- **Inclusão de novos jogadores:** Pelas especificações do trabalho, TODOS os jogadores precisam apostar na primeira rodada. Dessa forma, os jogadores são "cadastrados" no sistema da partida sempre nesta rodada. As rodadas subsequentes apenas pegam os jogadores previamente cadastrados na partida. Caso alguma rodada tente acessar um jogador que não foi cadastrado, a rodada será invalidada e o programa segue normalmente.

- **Consistência das apostas:** Caso uma aposta não seja um número ou não seja múltipla de 50, a rodada é invalidada e o programa segue normalmente.
- **Cartas repetidas:** Como foi explicitado nas especificações do trabalho, cada rodada utilizará apenas um baralho, completamente embaralhado. Sendo assim, não é possível que uma mesma carta esteja nas mãos de mais de um jogador simultaneamente. Por isso foi criada a classe "*Deck*". Ela armazena as 52 cartas do baralho, distribuindo as cartas solicitadas por código. Caso uma carta já tenha sido distribuída para outro jogador, é retornado um ponteiro nulo ao invés de um ponteiro para uma das cartas, o que faz com que a mão do jogador seja inválida (menos que 5 cartas) e a rodada seja invalidada, com o programa seguindo a execução normalmente.

Testes

Os testes para o programa implementado serão divididos entre testes de acesso à memória e testes de desempenho computacional para as diferentes operações. Os resultados das partidas serão armazenados em arquivos ".txt", os tempos de execução em arquivos ".out", mapas de acesso à memória em arquivos ".png" e o teste *gprof* também em um arquivo ".txt".

O teste de memória verifica a performance do programa quanto o acesso à memória alocada pelo mesmo. Será utilizada para metrificar, a ferramenta da "distância de pilha". Basicamente, quando um endereço de memória é acessado, ele é inserido na pilha, quando ele for acessado novamente, ele é retirado da posição da pilha em que volta para a base da mesma. A distância da posição em que o elemento estava ao ser chamado para a primeira posição da fila será a distância de pilha daquela chamada em questão.

A distância de pilha total será calculada pela média ponderada da frequência em que um elemento é desempilhado com aquela distância, sendo que a distância em si é o peso da média. Quanto menor for a distância de pilha, melhor projetado é o código, indicando menor custo de memória.

Os testes de desempenho verificarão o tempo que leva para que uma função seja executada para um determinado tamanho de matriz de entrada. Para isso, será marcado o tempo de início da execução do programa e o tempo de finalização do mesmo. A diferença entre estes tempos é então calculada e será o tempo de execução do programa.

Será analisado não somente o tempo individual de execução para diferentes entradas, mas a evolução deste tempo e se ele comprova ou não a análise de complexidade feita em um dos tópicos anteriores.

Análise Experimental

A análise experimental foi dividida entre testes de acesso à memória e testes de desempenho computacional, que serão abordados em tópicos abaixo. Para mais informações sobre os métodos de teste utilizados para esta análise experimental, consulte o tópico “Testes” deste documento.

Desempenho Computacional

Os testes de desempenho computacional foram realizados 5 vezes com crescimentos lineares nas entradas do jogo. Todas as rodadas terão exatamente 10 jogadores, de forma a não violar as restrições de integridade do programa (cartas repetidas). Inicialmente, serão computadas 1000 rodadas, após isso, 5000 rodadas, crescendo em fatores de 5000 novas rodadas a cada teste.

A seguir, será apresentado o resultado obtido para este teste.

Saída gprof:

Os resultados gerados pelo *gprof* se referem aos custos individuais de cada função na operação do programa como um todo:

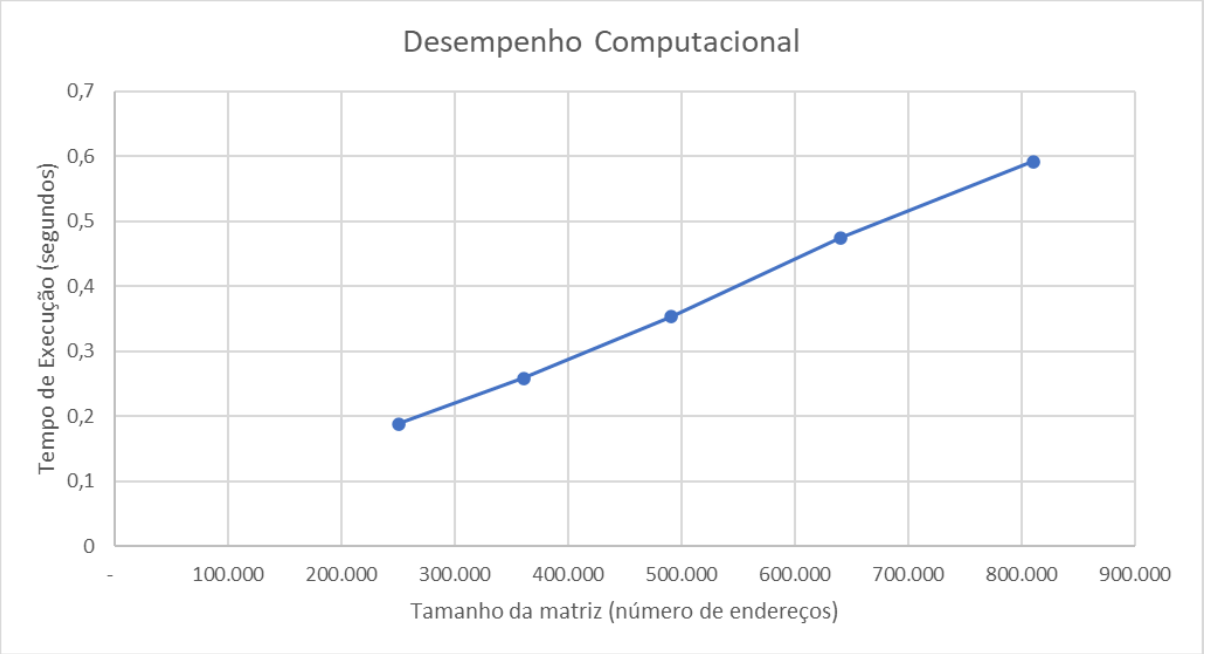
```
Flat profile:


```

Percebemos que a função que mais gasta tempo é ...

Evolução do Desempenho Computacional

Veremos uma tabela que mostra a evolução do desempenho com o crescimento da quantidade de rodadas a serem computadas pelo programa. Será exibido também um gráfico que mostra esta evolução de forma mais visual.

Número de linhas e colunas e nome do arquivo	Tempo inicial	Tempo final	Duração da execução



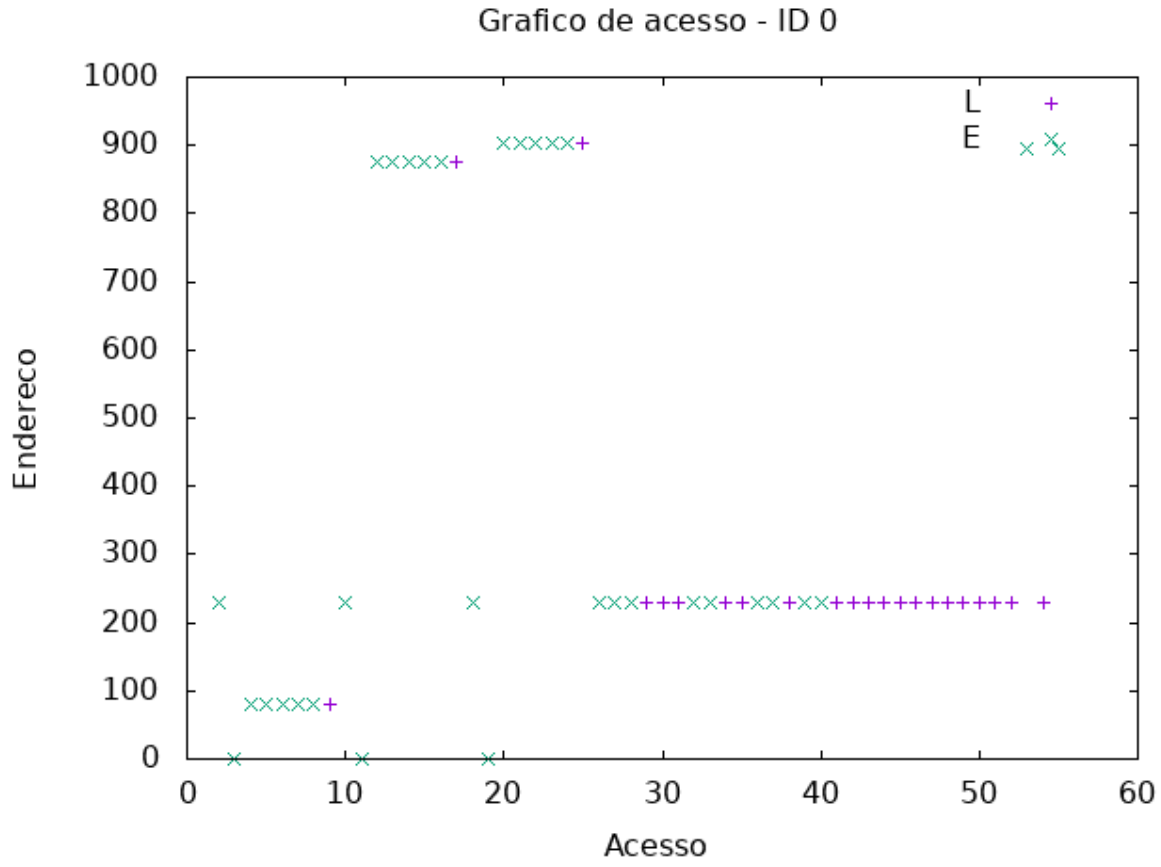
Observamos que o crescimento do tempo de execução, apesar de ter sido estimado como …

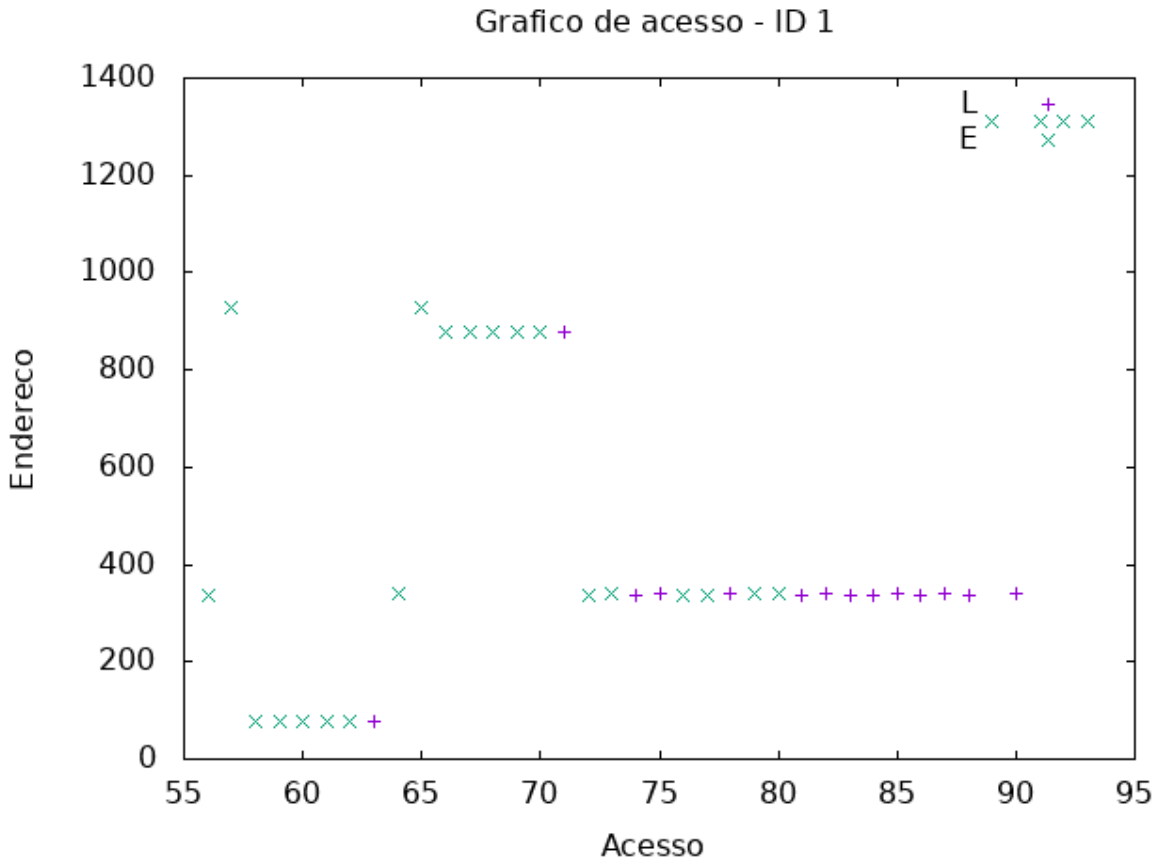
Análise de Padrão de Acesso à Memória e Localidade de Referência

Os testes de memória serão realizados com uma rodada de dois jogadores, para melhor visualização dos acessos. Não serão repetidos estes testes, já que eles são apenas para geração dos gráficos.

Mapa de acesso à memória

Este gráfico mostra como os endereços de memória do programa são acessados a cada chamada de função. Pontos em formato de “X” na cor verde representam acesso à memória para escrita, já os em formato de “+” na cor rosa, representam acesso para leitura.



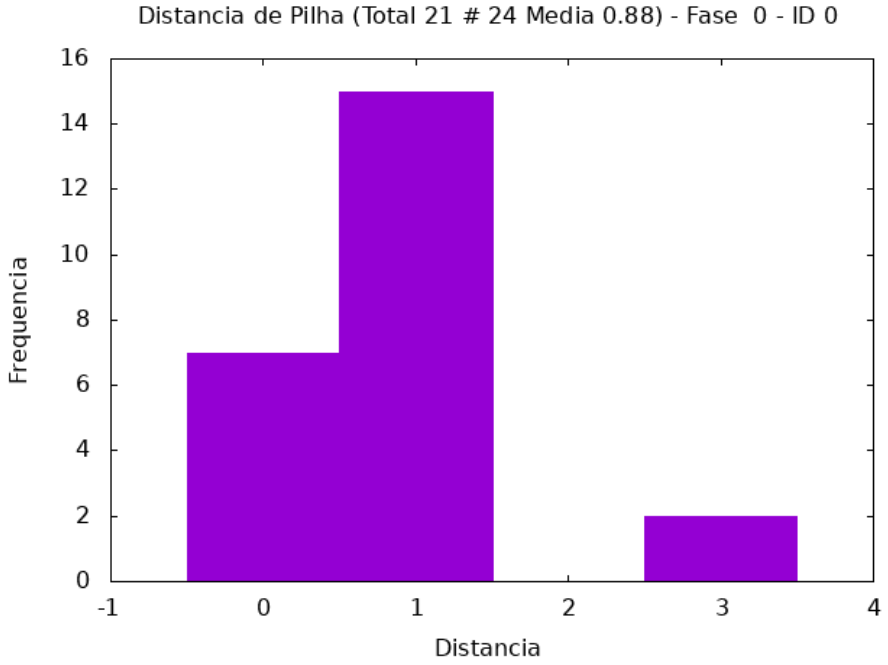


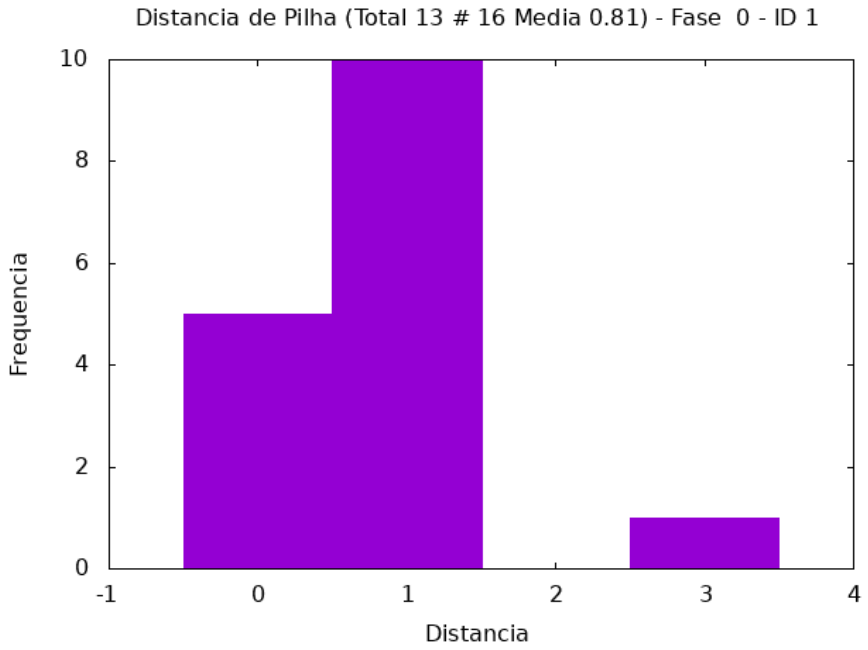
É possível observar que os gráficos de ID 0 e 1 são bem semelhantes. Isso se deve ao fato de que, apesar de serem rodadas diferentes, ambas passam por processos idênticos. A única diferença é a quantidade de jogadores.

Para explicar sucintamente tomaremos como base o gráfico para a rodada de ID 1. Os primeiros acessos (55 a 72) são provenientes da leitura do arquivo de input da partida, se referindo à adição de novos jogadores à rodada e a verificação da mão do jogador. Os próximos acessos (72 a 74) são para cobrança das apostas do jogador. Os acessos seguintes (74 a 81) são para ordenação do array de jogadores do round de acordo com a combinação da mão dos mesmos. Os seguintes (até 92) são para verificação dos ganhadores. Os últimos são o registro da premiação do pote da rodada.

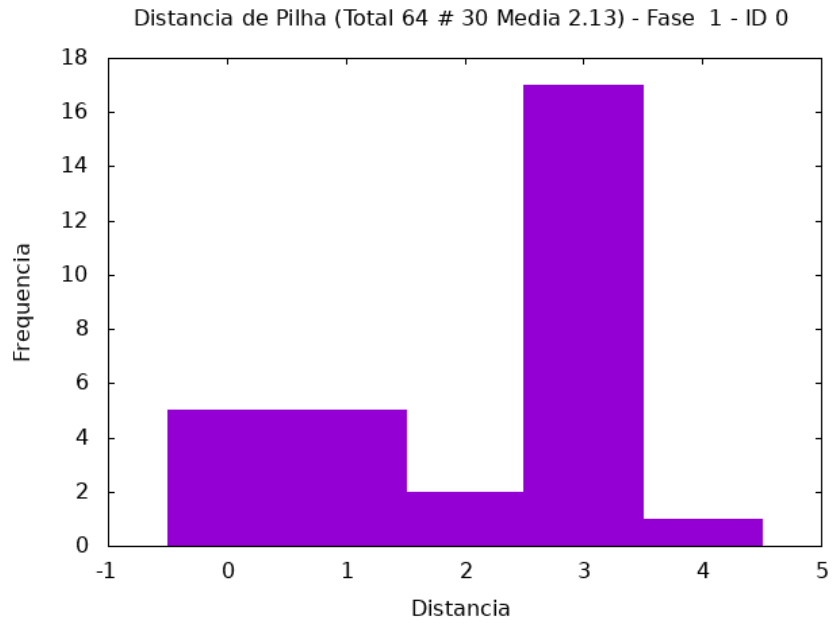
Distância de pilha

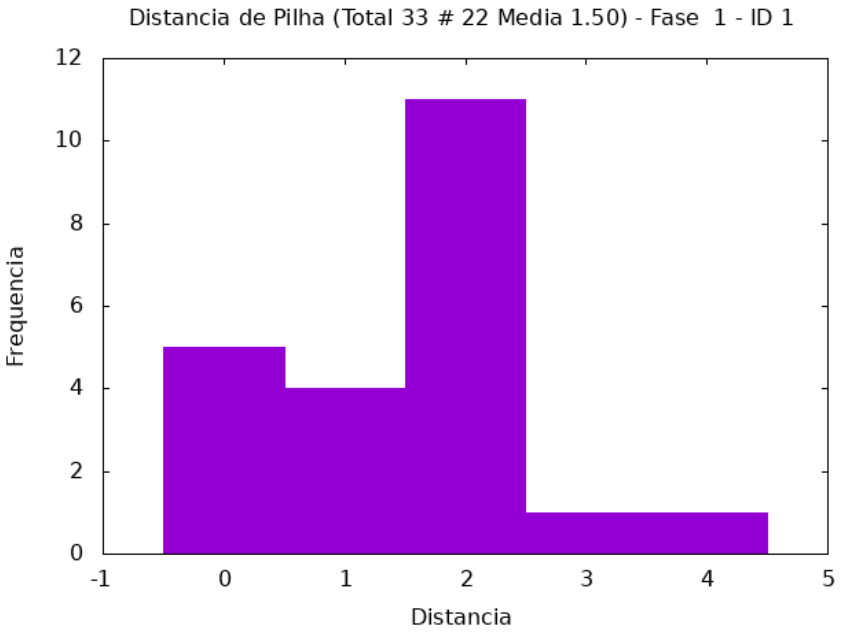
Estes gráficos são chamados de histogramas e mostram a distância de pilha dos acessos aos endereços de memória alocados para as estruturas de dados implementadas.





É notável que ambos os histogramas são bastante similares, variando somente na frequência dos acessos, cujas alterações são fruto da variação da quantidade de jogadores.



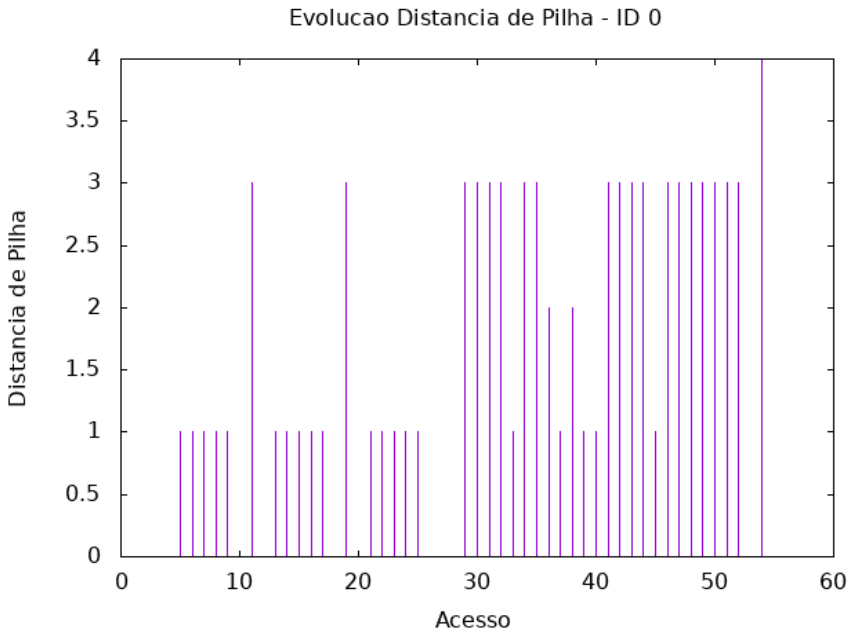


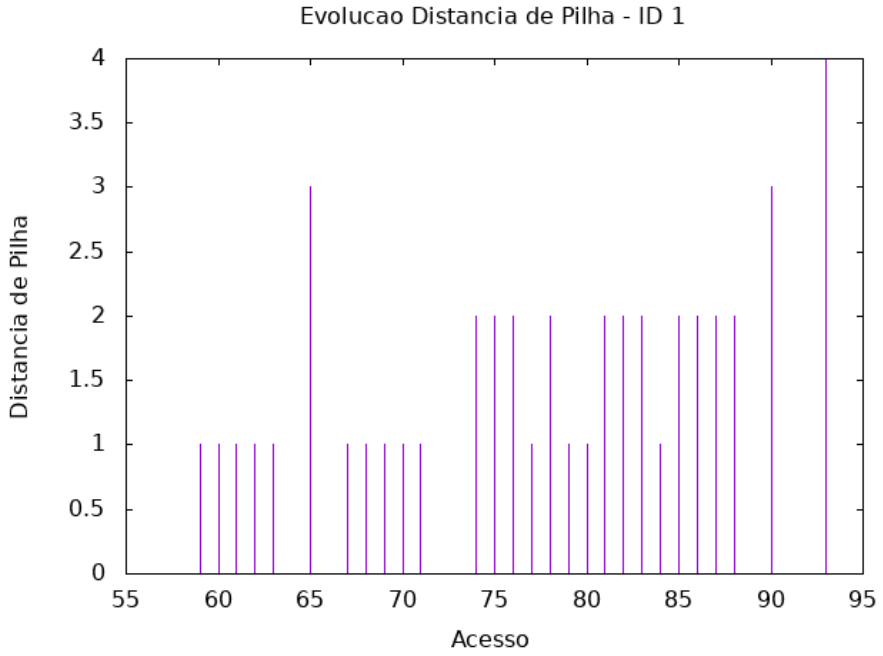
Na fase 1, as operações que mais exigem poder computacional são executadas. Portanto, as distâncias de pilhas são maiores e mais espaçadas, já que há uma série de comparações feitas entre diferentes endereços de memória. É possível identificar o pico dos histogramas dos IDs 0 e 1, respectivamente, em 3 e 2, que são justamente a quantidade de jogadores em cada uma das rodadas que esses IDs representam.

Evolução da Distância de Pilha

O acesso deve ser mais fraco e repetitivo nos primeiros acessos, já que eles se referem à instanciação da rodada e preenchimento de seus atributos. Já o final deve possuir acessos

mais frequentes e intensos, com distância de pilha geralmente igual à quantidade de jogadores envolvidos na respectiva rodada.





Conclusão

Neste trabalho, foram implementados uma série de tipos abstratos de dados para simular um jogo de Pôquer. Para isso, foram criados métodos que permitiram determinadas operações como: adição de jogadores à partida e a rodada, cobrança de apostas, testes de sanidade, entre outros. Toda a implementação foi feita na linguagem C + +, utilizando o compilador G + +, sendo também desenvolvido em sistema Linux.

Foi possível executar os conceitos vistos em sala de aula sobre estruturas de dados e algoritmos de ordenação, bem como realizar a análise de complexidade de cada um destes e exercitar as boas práticas de programação em C + +.

O trabalho contribuiu bastante para a consolidação dos conhecimentos teóricos de estruturas de dados apresentados na disciplina, sendo especialmente importante para a visualização de aplicações práticas dos conhecimentos adquiridos.

Bibliografia

Meira, W. and Pappa, G. (2021). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

Instruções para compilação e execução

1. Extraia o arquivo .zip presente;
2. Abra um terminal na pasta raiz do mesmo;
3. Digite “make”. Isso compilará o programa;
4. Digite “./bin/tp1.exe <argumentos>”.

   Os argumentos podem ser:

   a. -i <inputs_da_partida.txt> (arquivos de entrada para a partida)
   b. -o <outputs_da_partida.txt> (arquivo para registrar os resultados da partida)
   c. -p <arquivo_de_logs.txt> (arquivo para registrar os resultados computacionais de performance e/ou memória da partida)
   d. -l Caso esta flag esteja presente, o acesso à memória é registrado no arquivo de logs de “-p”

OBS: O programa também pode ser executado sem parâmetro algum. Ele está configurado para, por padrão, acessar como input o arquivo "entrada.txt" e como output o arquivo "saida.txt”, desde que ambos estejam na mesma pasta em que o terminal foi aberto.

É possível também executar o comando “*./bin/tp1.exe -h*”. Com isso, as instruções de utilização serão exibidas na tela.