Trabalho Prático 1

DCC214 - Estrutura de Dados

Igor Joaquim da Silva Costa

1. Introdução

O problema proposto foi simular um jogo de poker virtual entre os alunos de uma sala. Para um conjunto de jogadores e suas respectivas cartas em uma rodada, deveria-se computar quais seriam os possíveis ganhadores e a quantia de dinheiro que eles receberiam, a partir das regras do jogo de Poker. Ao final de todas as rodadas, um placar ordenado pela quantia de dinheiro deveria ser montado.

Para resolver o problema citado, foi seguida uma abordagem voltada à programação orientada a objetos na linguagem c++, onde cada objeto representa parte da modelagem do mini-mundo de interesse. A partir disso, a simulação do jogo acontece pela mútua interação entre objetos por meio de métodos próprios, onde cada TAD contém as responsabilidades de suas respectivas entidades, garantindo a consistência do sistema.

Diante do exposto, a documentação presente possui como objetivo detalhar como o sistema foi implementado (Seção 2), o quão eficiente ele pode ser em termos teóricos(Seção 3) e aplicados (Seção 5), além de explicar como o programa lida com possíveis inconsistências durante a execução do jogo (Seção 4). Por fim, o projeto é sumarizado junto com os aprendizados gerados durante a produção do trabalho(Seção 6).

2. Método

Como abordado na seção 1, o código gerado usa extensivamente os conceitos de Programação Orientada à Objetos como uma maneira de organizar e implementar as responsabilidades do sistema. Desse modo, foram retiradas três entidades fundamentais do mini-mundo poker: carta, jogador e “rodada”, onde a última modela os estados dos jogadores em uma rodada específica.

2.1 Fluxo principal

O tópico atual tem como objetivo esclarecer a nível conceitual como as diferentes classes se relacionam, explicando o fluxo principal do programa. As partes em negrito representam os nomes das classes e métodos utilizados.

O primeiro passo na execução do código se trata da leitura da quantidade de rodadas presentes no jogo e da quantia de dinheiro inicial dos jogadores. Com essas informações em mãos é possível iniciar a primeira **rodada**. Cada **rodada** é composta por **jogadores**, apostas e **cartas.** Caso algum jogador não possa contribuir com a aposta estipulada, a rodada é invalidada e o "**dinheiro resetado**", ou seja, aqueles jogadores que já contribuíram com o pote nessa jogada ganham seu dinheiro de volta. Nas rodadas válidas, o jogo **calcula a rodada** e determina os ganhadores, repartindo o pote entre eles. Para se **calcular a rodada**, basta encontrar quais **jogadores** podem performar as **jogadas** de nível mais alto. Por fim, as informações da **rodada** e o **estado final** do jogo são armazenadas em um arquivo de saída.

Adiante, estão as implementações dessas classes.

2.2 Estrutura de dados: Lista_ordenada

Como diversas partes do programa necessitam de algum tipo de ordenação, a necessidade de armazenar dados com alguma ordem embutida tornou válida a existência de uma classe com tal propriedade. Nesse sentido, o TAD lista ordenada é uma classe template, ou seja, uma classe capaz de trabalhar com qualquer tipo de variável que cumpra certas especificações mínimas de funcionalidade. No caso presente, qualquer elemento com a operação "menor que" pode ser usada para a construção de uma lista_ordenada, que ordena elementos durante a inserção. As únicas atribuições de uma lista ordenada são a busca e a inserção de elementos. A busca foi implementada usando um algoritmo de busca sequencial e a inserção procura o primeiro elemento menor do que aquele que está para ser inserido, se esse elemento não existir, o novo elemento é inserido ao final da lista.

2.2.1 Node_t

Para a implementação da lista encadeada ordenada, foi criada uma outra classe template chamada node_t, que funciona como a célula de uma lista duplamente encadeada. Os únicos atributos dessa classe são ponteiros, um para a próxima célula, um para a célula anterior e outro para o conteúdo armazenado.

2.3 Classes

2.3.1 Carta

TAD responsável por representar uma carta de poker, onde cada carta é composta por uma letra que representa um naipe e um número entre 1 e 13.

Implementação:

-Atributos: a classe carta possui 2 atributos privados, naipe e número.

-Operadores:

Igualdade: Duas cartas são iguais se tiverem o mesmo NÚMERO.

Menor que: Uma carta é menor do que a outra se o NÚMERO da primeira for menor.

2.3.2 Jogador

TAD responsável por gerir as responsabilidades de um jogador na partida. O jogador é entendido como a entidade que possui uma quantia inteira de dinheiro, um nome e uma mão, ou seja, um conjunto de cinco cartas. No quesito responsabilidade, o jogador deve ser capaz de interpretar qual jogada corresponde a sua mão atual, aumentar e diminuir o seu dinheiro e, ao final de uma jogada, devolver suas cartas ao baralho, além de verificar se as apostas feitas por ele seguem as regras do jogo.

Nesse sentido, a classe jogador possui atributos e métodos para modelar cada um desses comportamentos, como, por exemplo, a função set_aposta(), que verifica se a aposta apresentada pelo jogador é múltipla de 50 e menor do que o dinheiro que ele possui atualmente, como é definido nas especificações do jogo.

No que tange a dinâmica do jogo de poker, um detalhe importante da classe jogador é seu atributo “mao”, que é do tipo lista_ordenada<carta>. Armazenando as cartas de maneira ordenada, a verificação de quais jogadas possíveis um conjunto de cartas pode performar se torna mais fácil, já que muitas jogadas dependem de existências de números consecutivos entre as cartas ou determinação da carta de maior número. Além do atributo mão, o atributo valor_jogada também é crucial para o fluxo de jogo. Nele são armazenados o valor da maior jogada possível de se realizar com a mão atual do jogador e o critério de desempate para aquela jogada, facilitando a interação entre vários jogadores, já que esse atributo pode ser usado para determinar os ganhadores de uma rodada.

Por fim, a classe usa de sobrecarga de operadores para facilitar a busca de um jogador em específico e a ordenação dos mesmos. Um jogador é igual ao outro se os nomes forem iguais e um jogador é menor que outro se o valor_jogada do primeiro for menor do que o do segundo. Caso ambos jogadores tenham o mesmo valor de jogada (variável x do par valor_jogada), o critério de desempate (variável y) se torna decisivo. Se ainda assim não for possível decidir qual dos dois é o menor, a operação é decidida pelo jogador que possui maior o nome em ordem lexicográfica.

2.3.3 Rodada

A fim de gerenciar o estado de diversos jogadores durante a partida, foi criado o TAD rodada, a classe que decide quais jogadores ganham determinada jogada e atribui o valor do pote a eles.

O principal método da lista de jogadores é o “calcula_rodada()”, onde, dada uma rodada em específico, é determinada qual a jogada com o maior valor possível de ser executada, além de determinar quais jogadores podem performá-la. O método alcança esse objetivo usando duas estruturas auxiliares, a primeira se tratando de um array de 10 funções que representa as 10 jogadas possíveis, ordenadas da menor para a maior, e a segunda do método calcula_ganhadores(), onde, dada uma das 10 jogadas, são armazenados todos os jogadores que podem realizar aquela jogada em um array auxiliar. Dessa forma, o método calcula_rodada() percorre as 10 possíveis jogadas armazenadas no array de forma decrescente até encontrar a primeira que algum jogador possa fazer, após esse passo, a função percorre o array composto apenas pelos possíveis ganhadores para determinar quais de fato vão ganhar a rodada, com base no parâmetro valor_jogada da classe jogador. Sabendo os índices dos jogadores que passam do critério de desempate, a função divide o pote entre eles e armazena essas informações no arquivo de saída.

Além disso, a classe também garante a integridade do jogo ao evaluar se todos os jogadores podem contribuir para o pingo pela função set_pingo() , reiniciando a rodada em caso de erro ou ao final de todas as jogadas com a função reseta_jogadas() e devolvendo o dinheiro ao jogadores, caso necessário, na função reseta_dinheiro().

2.4 Jogadas

As funções jogadas são representações das 10 possíveis configurações de mãos presentes no jogo de poker. Cada jogada possui um valor único e precisa cumprir seus critérios próprios para acontecer, são elas que decidem quais são os possíveis ganhadores de uma rodada. Além disso, cada jogada segue seu próprio critério de desempate, se existir.

Dito isso, todas as funções jogadas do programa são reflexos das jogadas reais, salvo algumas mudanças para se adequar a especificação do trabalho e para lidar com partes não especificadas, como no caso dos critérios de desempate de cada função. A ideia principal das funções jogada é determinar se a mão de um dado jogador configura aquela jogada em específico, se sim, o atributo valor_jogada do jogador é atualizado para o valor da jogada analisado e a função retorna true, se não, ela retorna zero. Dessa forma, se torna conveniente a comparação de jogadas entre diversos jogadores, visto que jogadores com maior atributo valor_jogada possuem mãos mais fortes do que as dos seus oponentes.

Além disso, a segunda variável do atributo valor_jogada armazena o critério de desempate do jogador, sendo possível desempatar o jogo caso necessário. As funções jogadas fazem esse cálculo de desempate usando as regras do jogo de poker, como, por exemplo, a função flush, que armazena o valor da carta mais alta do jogador como critério de desempate, fazendo com que dois jogadores capazes de executar um flush tenham valores de jogada diferente, baseado nesse segundo parâmetro. Para jogadas com mais de um ganhador, o segundo parâmetro é constante no valor 0. Ademais, jogadas com mais de um critério de desempate também são calculadas da mesma forma, como, por exemplo, a jogada Three of a kind, que em caso de empate é considerada a maior carta da tripla e, em caso de um segundo empate, a maior carta entre os jogadores, seu critério de desempate é da formula 100 * (maior número da tripla) + (carta de maior número), garantindo que o critério de desempate continue válido.

Por fim, existem as jogadas com critérios de desempate não especificados para todo tipo de situação, como a jogada ONE PAIR, onde o critério de desempate primário é o valor da maior carta no par e o critério secundário sendo a maior carta na mão do jogador. Definições dessa forma causam problemas em alguns cenários, como nos casos onde dois jogadores possuem o mesmo par e a mesma carta mais alta, ficando incerto como decidir quem são os ganhadores. Em todos os casos possíveis dessa incerteza acontecer, ambos jogadores são considerados ganhadores, mesmo em jogadas onde apenas um jogador poderia ganhar.

3. Análise de complexidade

3.1 Espaço

Como estamos limitados a um baralho de 52 cartas, o número máximo de jogadores é 10, já que 11*5 = 55, que ultrapassa o número de cartas do baralho. Ainda assim, considerando o número de jogadores como a variável N e o espaço para armazenar uma carta sendo 1, a quantidade máxima de espaço que o programa ocupa é $\Theta(2 * 5N)$, já que, para cada jogador, são armazenadas 5 cartas, além de serem criadas duas listas de jogadores adicionais, uma para armazenar os ganhadores e outra para armazenar o placar, vale dizer que ambas não existem ao mesmo tempo. Dito isso, o programa é O(N) no espaço, ou seja, ele independe da quantidade de rodadas efetuadas.

3.2 Tempo

Para análise de tempo, considere N como o número de jogadores e R como o total de rodadas.

3.2 Inserção/Busca

Considerando a comparação como operação mais significativa, como tanto os jogadores quanto as cartas são ordenados no momento da inserção, o melhor caso é O(1), que equivale a inserir o menor elemento na lista. O pior caso, inserir o maior elemento, é O(N), visto que são necessárias ao menos N comparações para inserir um elemento ao final da lista. De maneira análoga, a busca também é O(1) no melhor caso, procurar o menor elemento, e O(N) no pior caso, procurar o último elemento, já que o algoritmo de busca utilizado simula uma busca sequencial. Dessa forma, tanto a Inserção quanto a Busca são O(N) no número de elementos no pior caso. No contexto de uma rodada, onde cada jogador sempre é buscado ou inserido, a complexidade se torna $O(N^2)$, visto que, para cada um dos N jogadores presentes, acontece uma operação que é O(N) no pior caso. No contexto de múltiplas rodadas, a complexidade cresce para $O(R * N^2)$.

3.3 Executar jogadas

Considerando a comparação como a operação mais significativa, no contexto de determinar a jogada de apenas um jogador, são feitas O(10*5) comparações no pior caso, já

que, para cada jogada, é necessário analisar as 5 cartas na mão do jogador. No contexto de N jogadores, somente executar as jogadas se torna O(10*5*N). Entretanto, como as jogadas precisam ser desempatadas e o pote repartido, são feitas mais O(N + N) operações no caso que todos os jogadores ganham na mesma rodada. Dessa forma, a complexidade se torna O(10*5*(3N)). No contexto de múltiplas rodadas a complexidade se torna O(10*5*(3N) * R) = O(R*N), ou seja cresce linearmente no número de rodadas e no número de jogadores.

3.4 Testes de sanidade

Considerando a comparação e o módulo como as operações mais significativas, no contexto de uma rodada com N jogadores, a complexidade de executar um teste de sanidade é O(N), já que, para cada jogador, a operação correspondente a esse teste de sanidade é feita. Como são feitos 3 testes de sanidade em uma rodada, a complexidade se torna O(3*N). No contexto de múltiplas rodadas, a complexidade cresce para O(R*N*3) = O(R*N), ou seja, cresce linearmente com o número de rodadas e com o número de jogadores.

3.5 Complexidade final

Considerando que a cada rodada, são feitas inserções, buscas, execuções de rodadas e testes de sanidade, a complexidade total do programa fica $O(R * (N^2 + N + N)) = O(R * N^2)$, ou seja, linear no número de rodadas e quadrático no número de jogadores.

4. Estratégias de Robustez

4.1 Testes de sanidade

Como especificado na descrição do jogo, toda aposta deve ser múltipla de 50 e nenhum jogador pode apostar mais do que ele possui, dessa forma, a função set_aposta() retorna falso para todo caso que não segue essas regras. Assim, toda rodada que possui alguma inconsistência não é considerada no placar final do jogo.

5. Análise Experimental

A análise experimental a seguir tem como objetivo medir o quão eficiente é o sistema implementado usando duas métricas, o desempenho computacional - quão rápido o programa é executado com entradas grandes - e análises de acesso em memória.

5.1 Desempenho computacional

5.1.1 Perfil de execução

Para testar o desempenho computacional, primeiramente, o programa foi compilado em estado de "profiling", a fim de analisar quais funções consomem relativamente mais tempo durante a execução do programa. Diante disso, foram feitas três execuções, cada uma composta por 100.000 rodadas aleatórias, geradas por um código python externo feito pelo aluno. Com o profile das execuções pronto, cada uma foi dado como entrada no programa “gprof”, uma ferramenta que auxilia na análise do desempenho computacional. Dito isso, segue a análise das chamadas de funções.

Teste 1:

Tempo de execução (%)	Calls	Função
17.07	2998855	lista_ordenada<carta>::insert(carta*)
9.76		int __gnu_cxx::__stoa<long, int, char, int>(long (*)(char const*, char**, int))
9.76		std::__cxx11::stoi(std::__cxx11::basic_string<char)
7.32		le_partida(std::basic_ifstream<char, std::char_traits<char> >&, rodada&, int)
7.32		carta::carta(std::string)
7.32		carta::operator<(carta const&) const
7.32		_init
6.10	2998855	jogador::insert(std::string)
4.88		__gnu_cxx::__stoa<long, int, char, int>(long (*)(char const*, char**, int), char const*, char const*, unsigned long*, int)::_Save_errno::_Save_errno()
3.66	2197492	jogador::limpa_mao()
2.44	3176935	__gnu_cxx::__enable_if<std::__is_char<char>::__value, bool>::__type std::operator==<char>(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&,
2.44	3176935	operator==(jogador const&, jogador const&)
2.44	998220	jogador::reseta_dinheiro()

2.44	602760	jogador::set_aposta(int)
2.44	599781	jogador::jogador(std::string)
2.44		rodada::find(std::string)
2.44		__gnu_cxx::__stoa<long, int, char, int>(long (*)(char const*, char**, int)
1.22	2998855	node_t<carta>::node_t(carta*&)
1.22	1532	int& std::forward<int&>(std::remove_reference<int&>::type&)

Para evitar redundância, os outros perfis de execução apresentam o mesmo comportamento do teste apresentado acima. No que tange a análise dos experimentos, fica evidente que a função que mais demanda tempo, em porcentagem, é a função para inserir ordenadamente as cartas na mão de um jogador. No caso demonstrado são feitas 5 inserções por jogador por jogada, um número que pode chegar a 5000000 se todos os jogadores participarem de todas as rodadas. Além disso, a função apresenta complexidade quadrática, contribuindo para uma grande parcela de tempo.

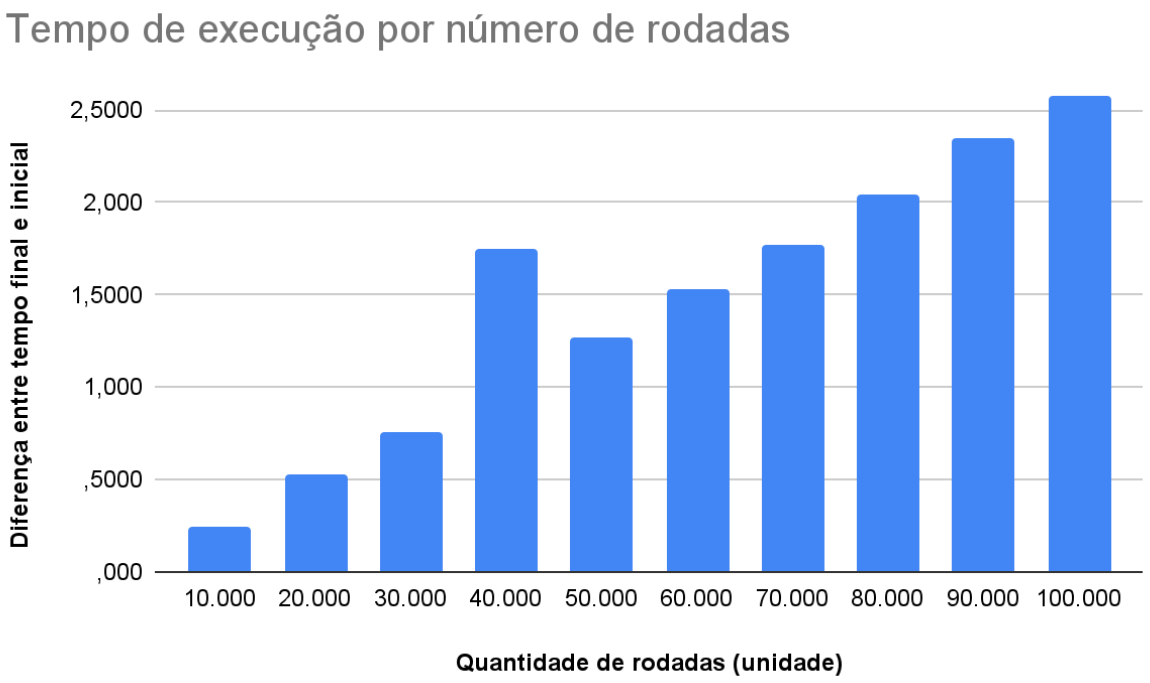
As funções que ocupam a segunda e terceira posição são funções que transformam strings em inteiros, outra operação que acontece durante a leitura dos dados da entrada, especificamente para ler o valor da aposta e para ler o número das cartas. Implementar uma leitura diretamente com números inteiros resolveria esse problema.

O quarto lugar corresponde a função de leitura do arquivo, o que faz sentido, já que a leitura de algo que está em memória secundária não é tão eficaz quanto a leitura em memória primária. Desse ponto em diante, as funções que gastam mais tempo são aquelas ligadas a simulação do jogo de poker e que ocorrem várias vezes na mesma rodada. É válido comentar sobre a performance da função de comparação entre cartas, já que ela é executada toda vez que uma carta é inserida, ocasionando em um grande gasto de tempo, mesmo sendo uma função relativamente barata de executar. Por fim, a função com maior número de chamadas corresponde a comparação entre jogadores, que acontece 2 vezes a cada operação de procura na lista ordenada.

Nesse sentido, conclui-se que ainda existem margens para a melhora no desempenho do programa, a nível de chamadas de funções, visto que muitas delas poderiam ser feitas de outras formas, a fim de melhor utilizar tempo de execução do programa.

5.1.2 Desempenho em termos da entrada

Além do tempo gasto pelas chamadas de funções, deve-se medir a relação entre o aumento do volume da entrada do programa com o seu tempo de execução. Para realizar tal objetivo, foi utilizada a ferramenta Memlog do professor Wagner Meira e ferramentas para a geração de gráficos. A seguir, o gráfico com os resultados.



Como visto acima, o tempo de execução cresce linearmente com o aumento da rodada, assim como foi discutido na seção de análise de complexidade.

5.2 Análise de localidade de referência e acesso de memória

Como explicado em aula, programas eficientes em uso de memória são aqueles que tendem a acessar regiões de memória contíguas, já que essa prática é otimizada pelos sistemas operacionais onde o programa é executado. Tal propriedade é chamada de localidade de referência e é uma ótima métrica de como o programa acessa a memória por ele utilizada.

Nesse sentido, munido das ferramentas Analisamem e Memlog, disponibilizadas pelo professor Wagner Meira, é possível gerar gráficos que demonstram o acesso de memória do programa em suas diversas fases, e, assim, analisar se o acesso de memória do programa segue as boas práticas desejadas. Dito isso, segue a análise de acesso de memória e localidade de referência, onde o id 0 representa os nodes da lista ordenada, o id 1 os jogadores e o id 2 as cartas.

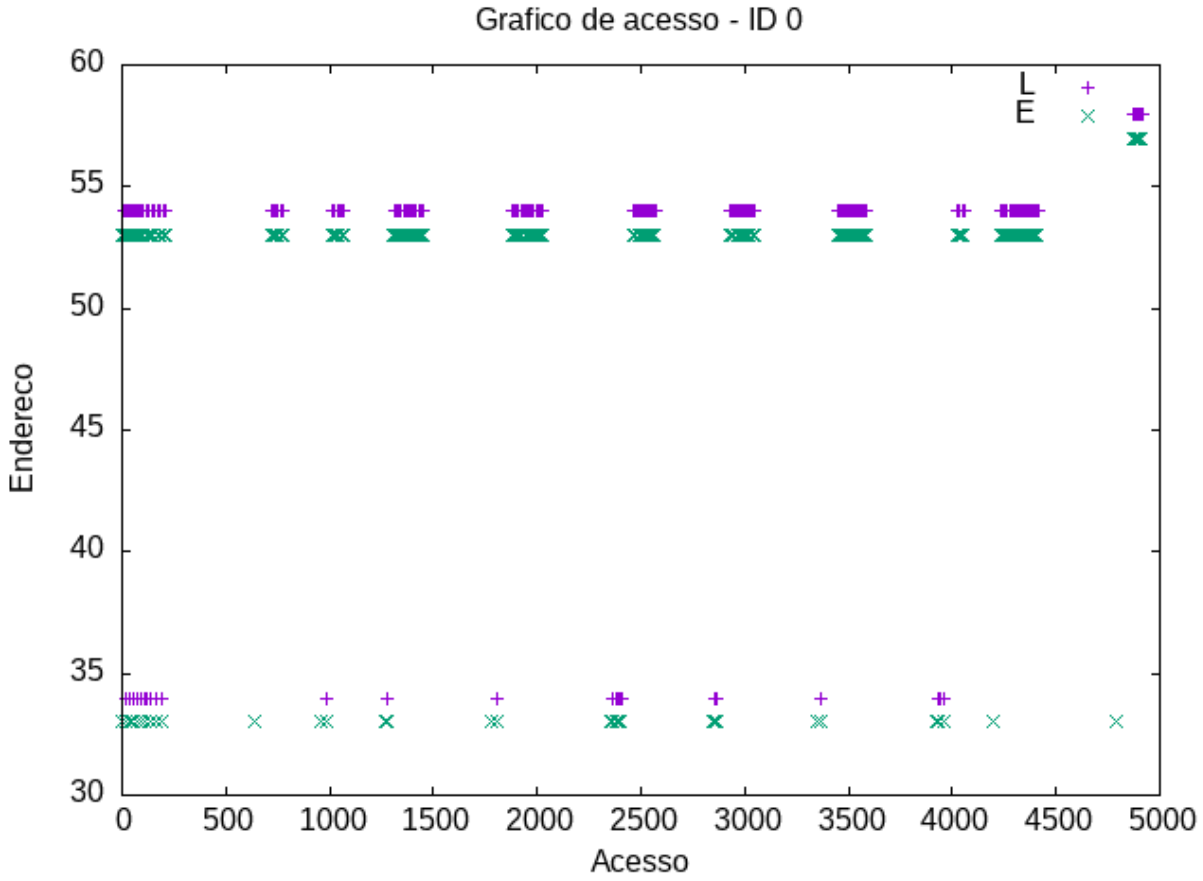


Grafico de acesso - ID 0
L
E
60
55
50
45
40
35
30
Endereco
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
Acesso

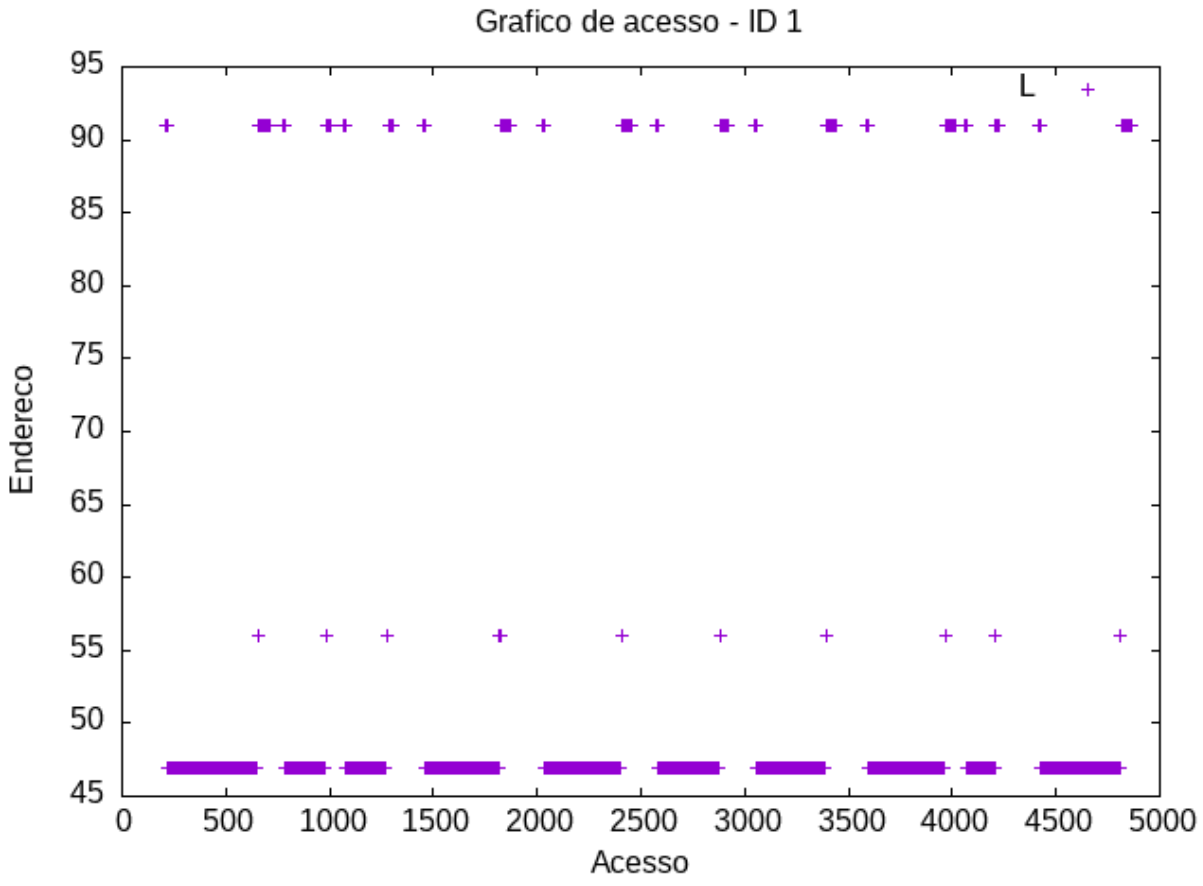
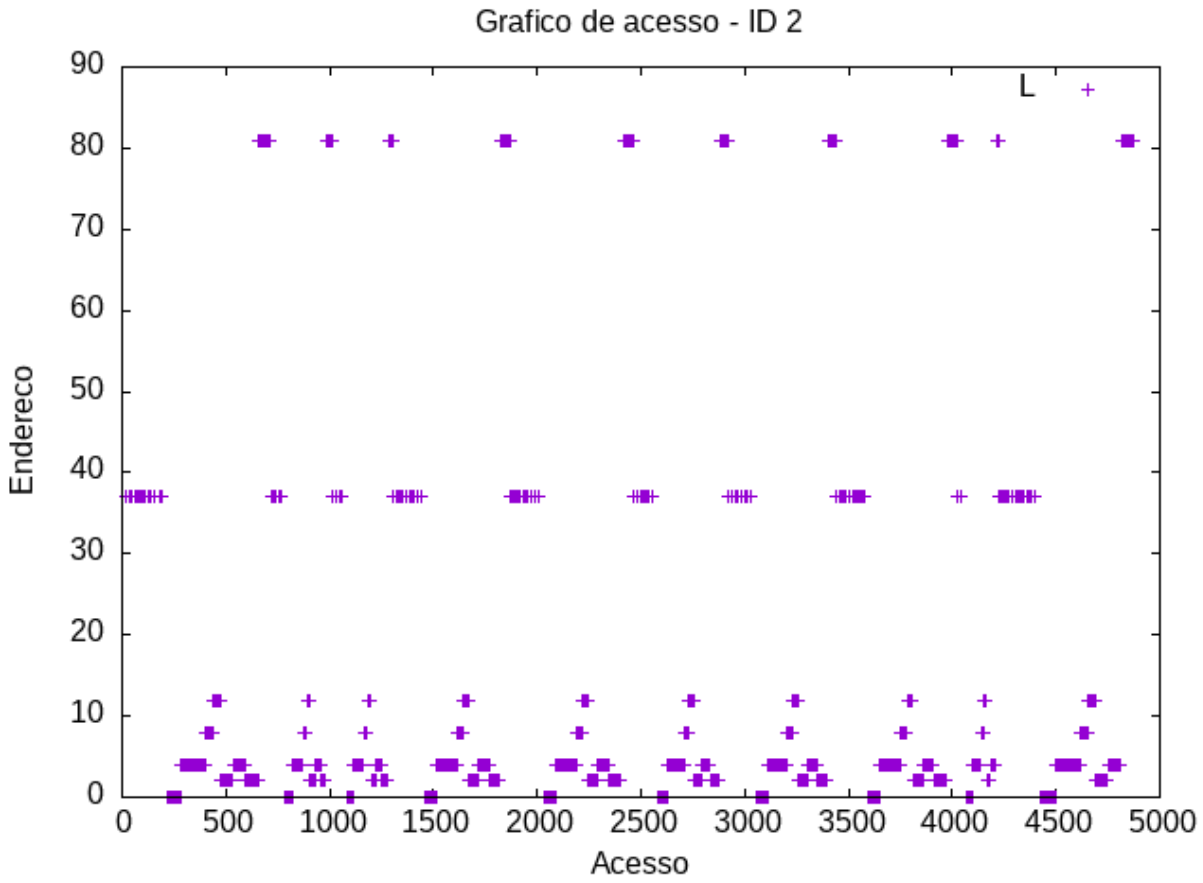
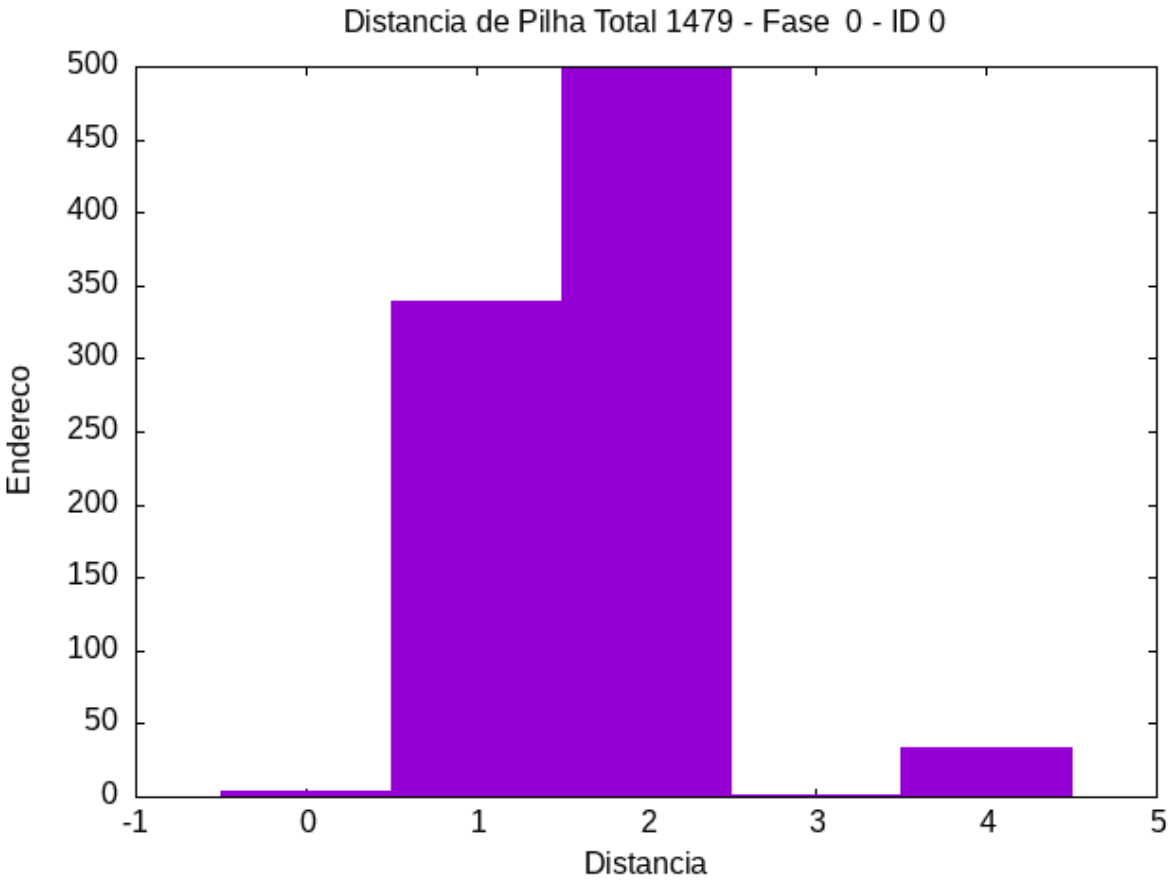


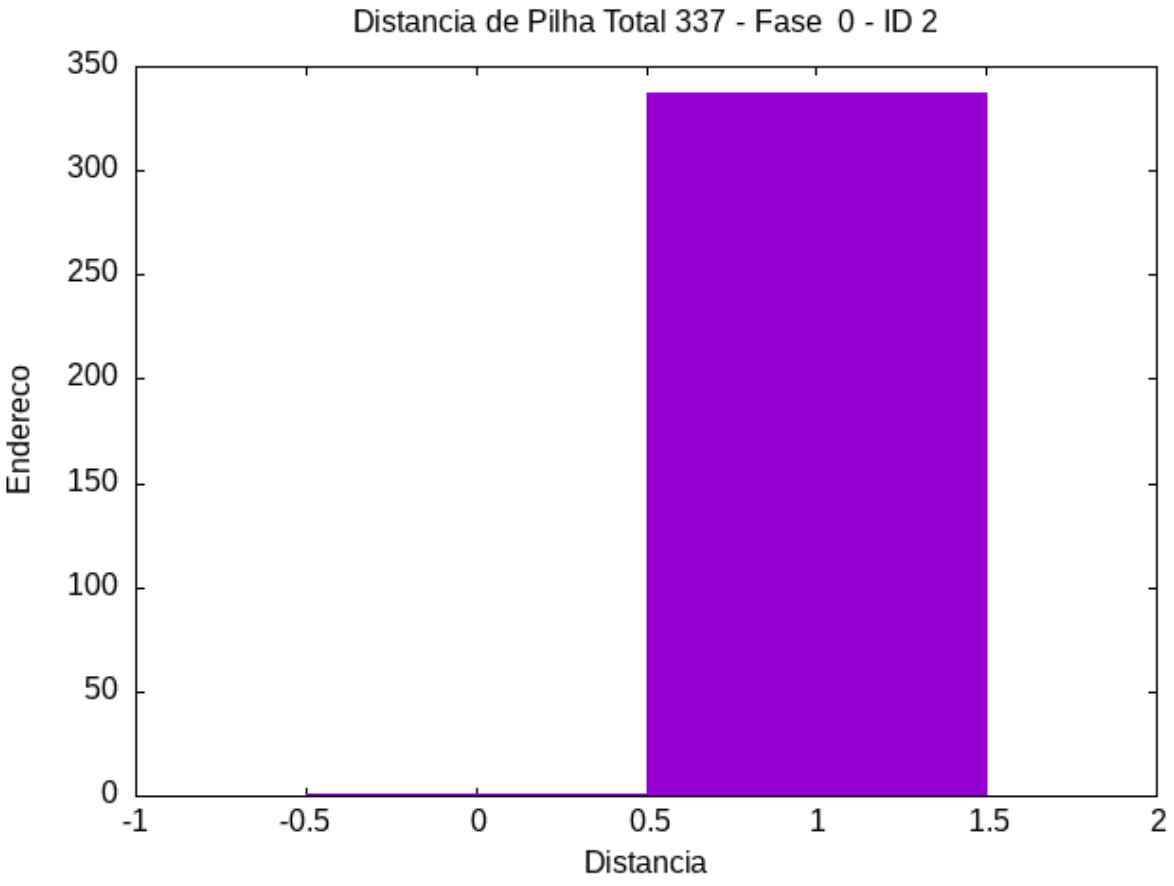
Grafico de acesso - ID 1
L
95
90
85
80
75
70
65
60
55
50
45
Endereco
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
Acesso

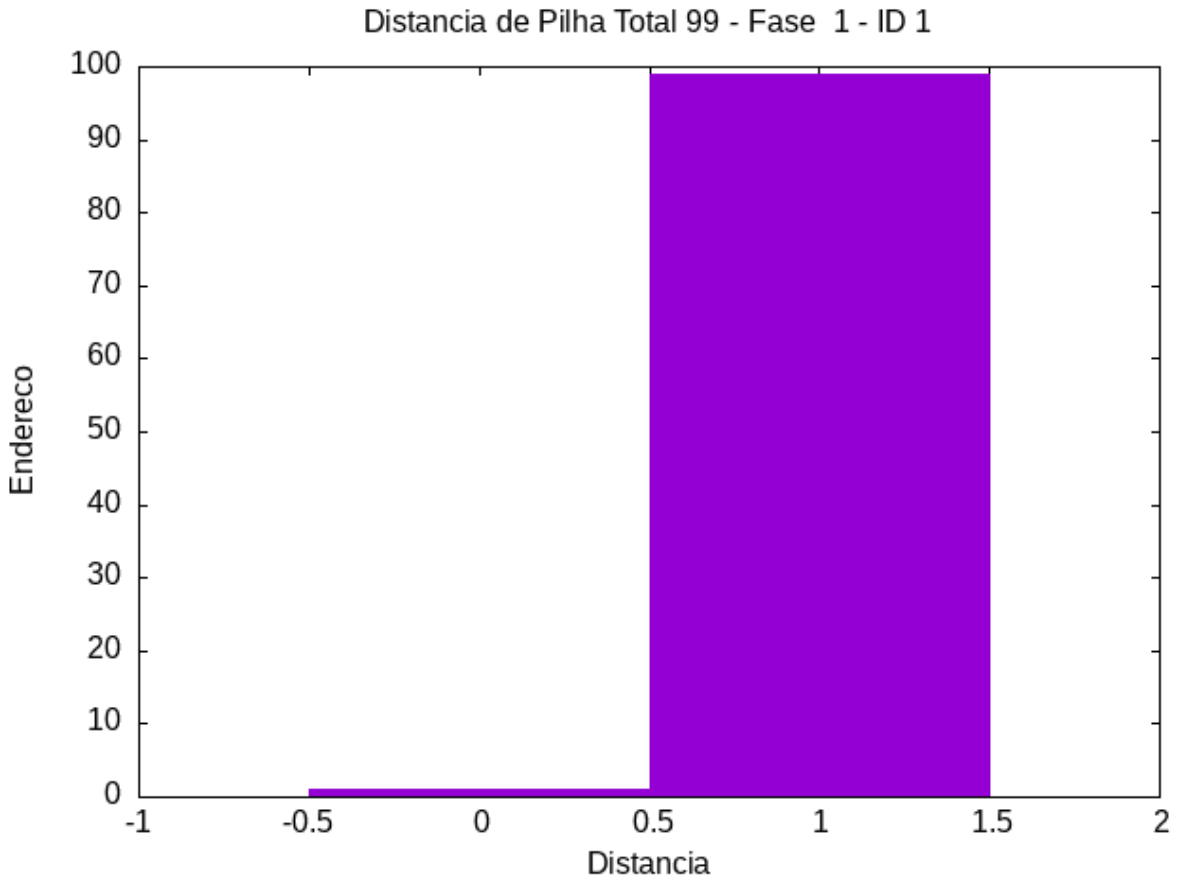


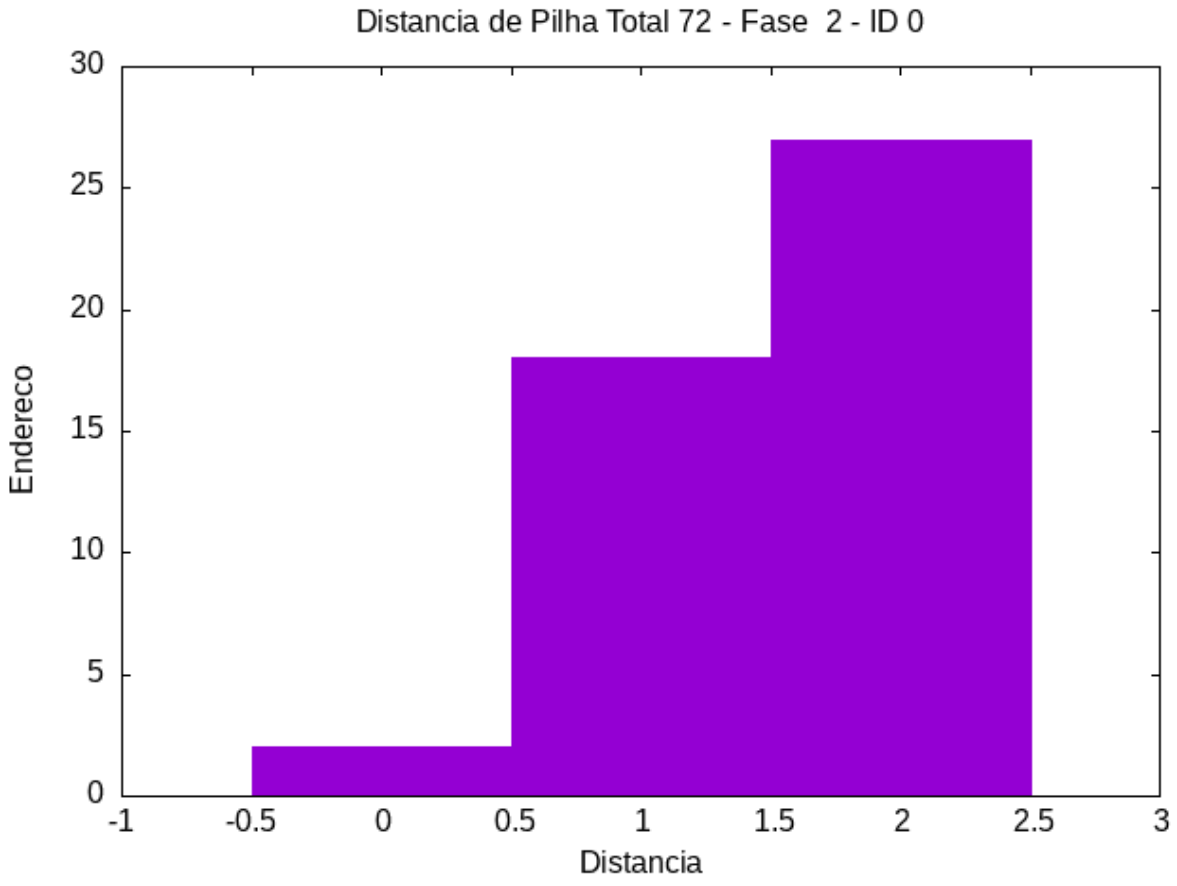
Como visto acima, o programa acessa diversas vezes posições de memória adjacentes, isso acontece porque sempre que a rodada é resetada, a memória utilizada pela rodada anterior é liberada e é reutilizada pela próxima. Dessa forma, o programa trabalha em sintonia com os conceitos essenciais do sistema operacional, sempre demandando e acessando posições de memória já utilizadas antes, dando margem para que esse processo seja otimizado pelo SO.

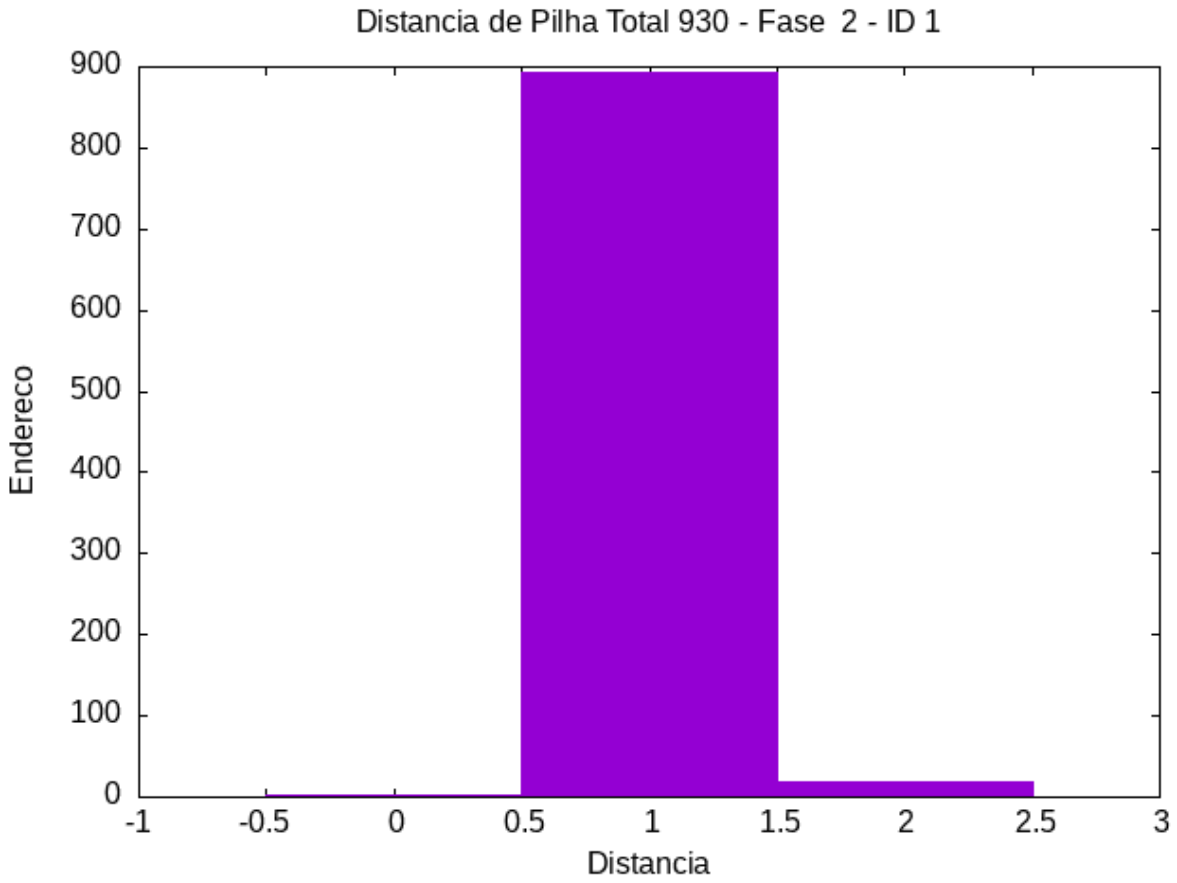
Entretanto, existem outras formas de medir diretamente a localidade de referência do sistema, analisando a distância de pilha nas diversas fases do programa. Nesse sentido, seguem os gráficos de distância de pilha em fases, onde a fase 0 engloba a leitura do arquivo, a fase 1, os testes de sanidade, a fase 2, o cálculo dos ganhadores e a fase 3, o recomeço da jogada.

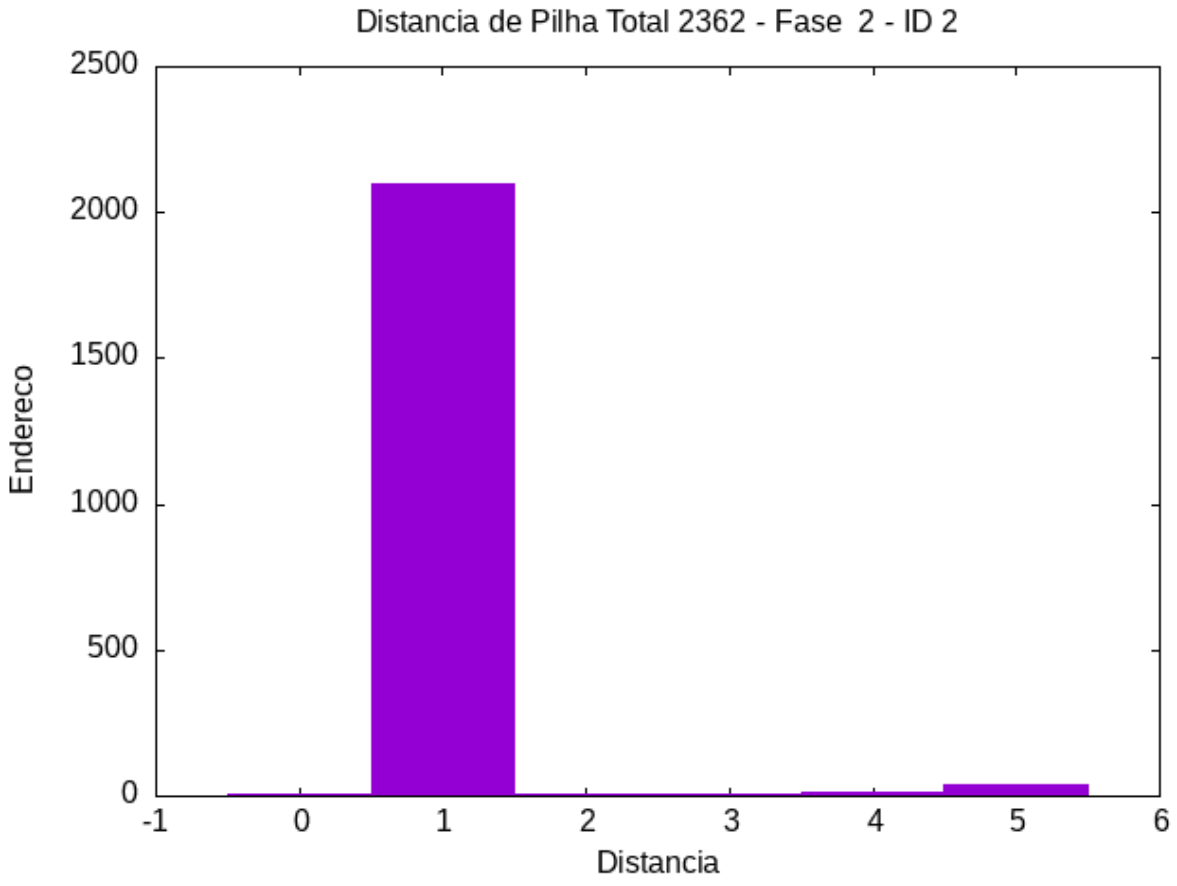

Distancia de Pilha Total 1479 - Fase 0 - ID 0
Endereco
Distancia
500
450
400
350
300
250
200
150
100
50
0
-1
0
1
2
3
4
5

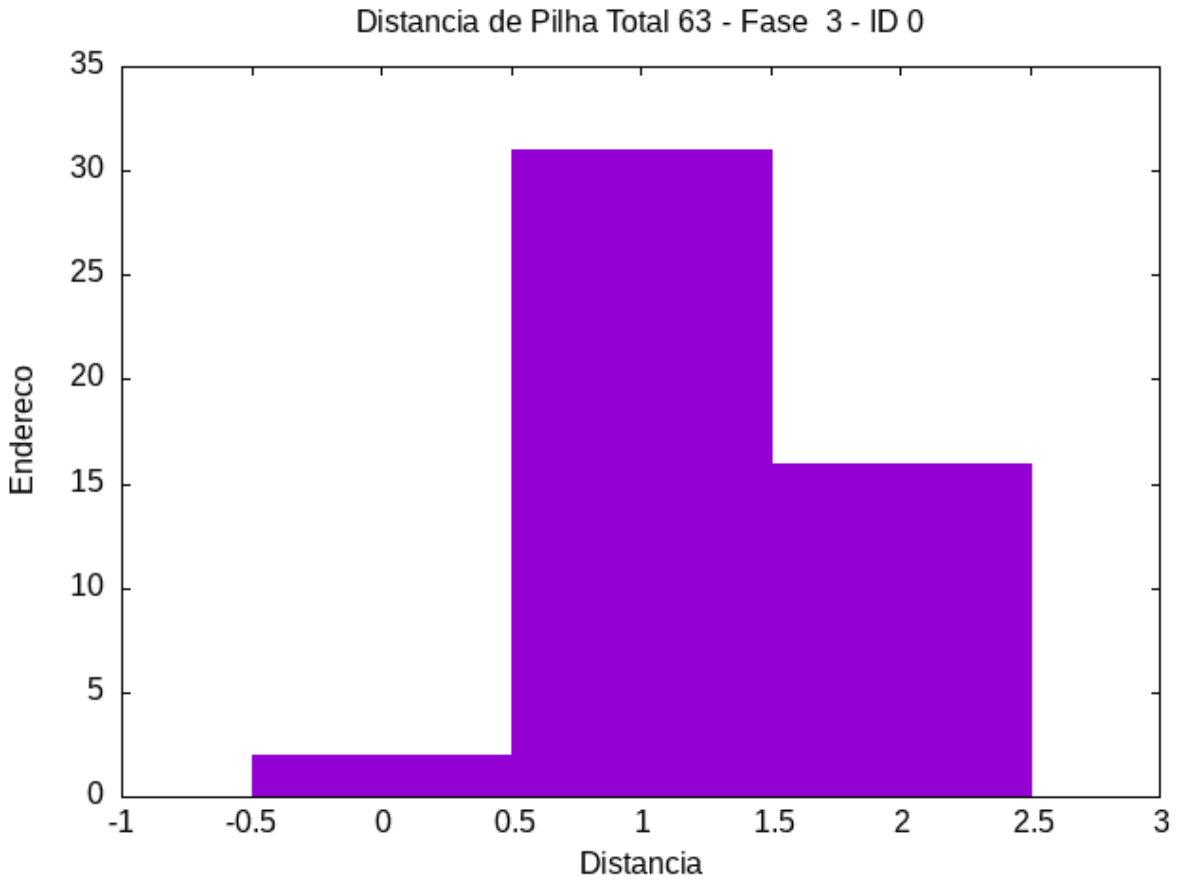

Distancia de Pilha Total 337 - Fase 0 - ID 2
Endereco
Distancia
350
300
250
200
150
100
50
0
-1
-0.5
0
0.5
1
1.5
2

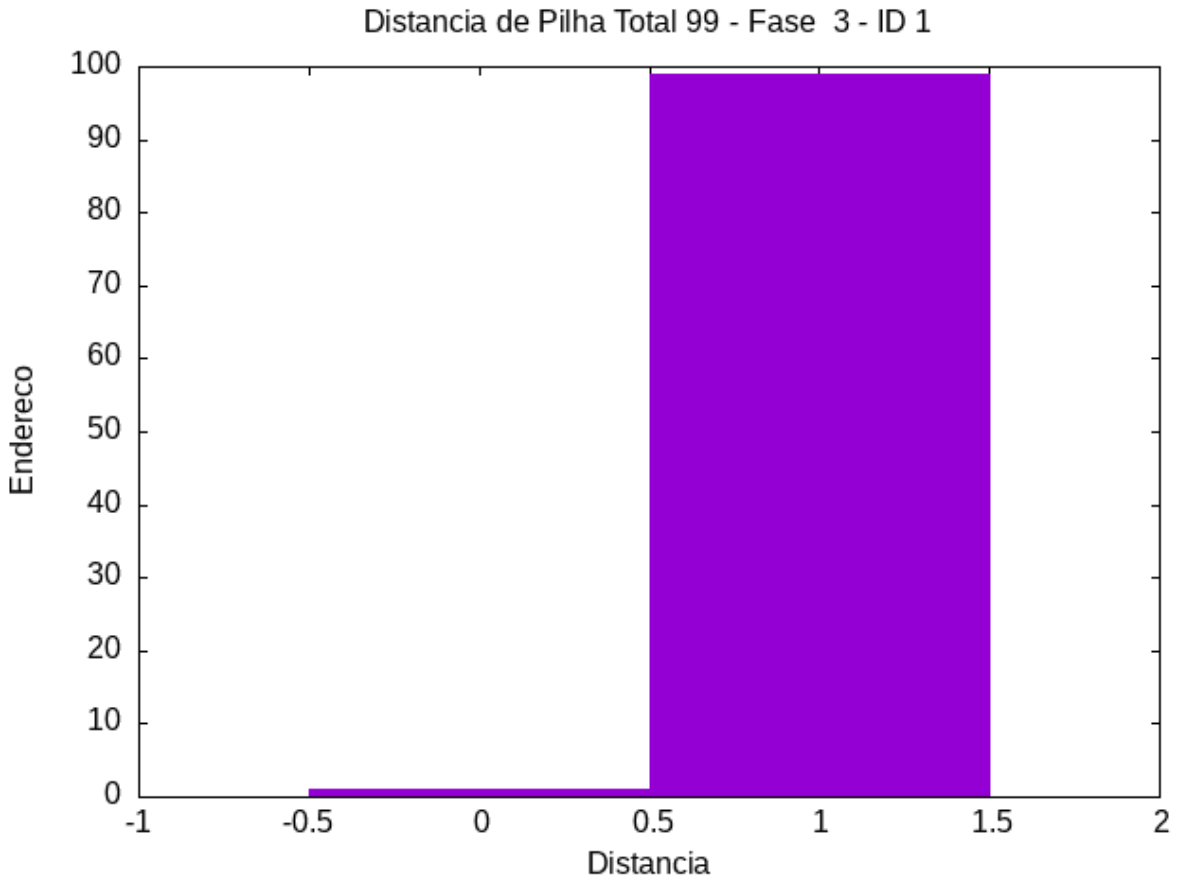

Distancia de Pilha Total 99 - Fase 1 - ID 1
Endereco
Distancia
100
90
80
70
60
50
40
30
20
10
0
-1
-0.5
0
0.5
1
1.5
2

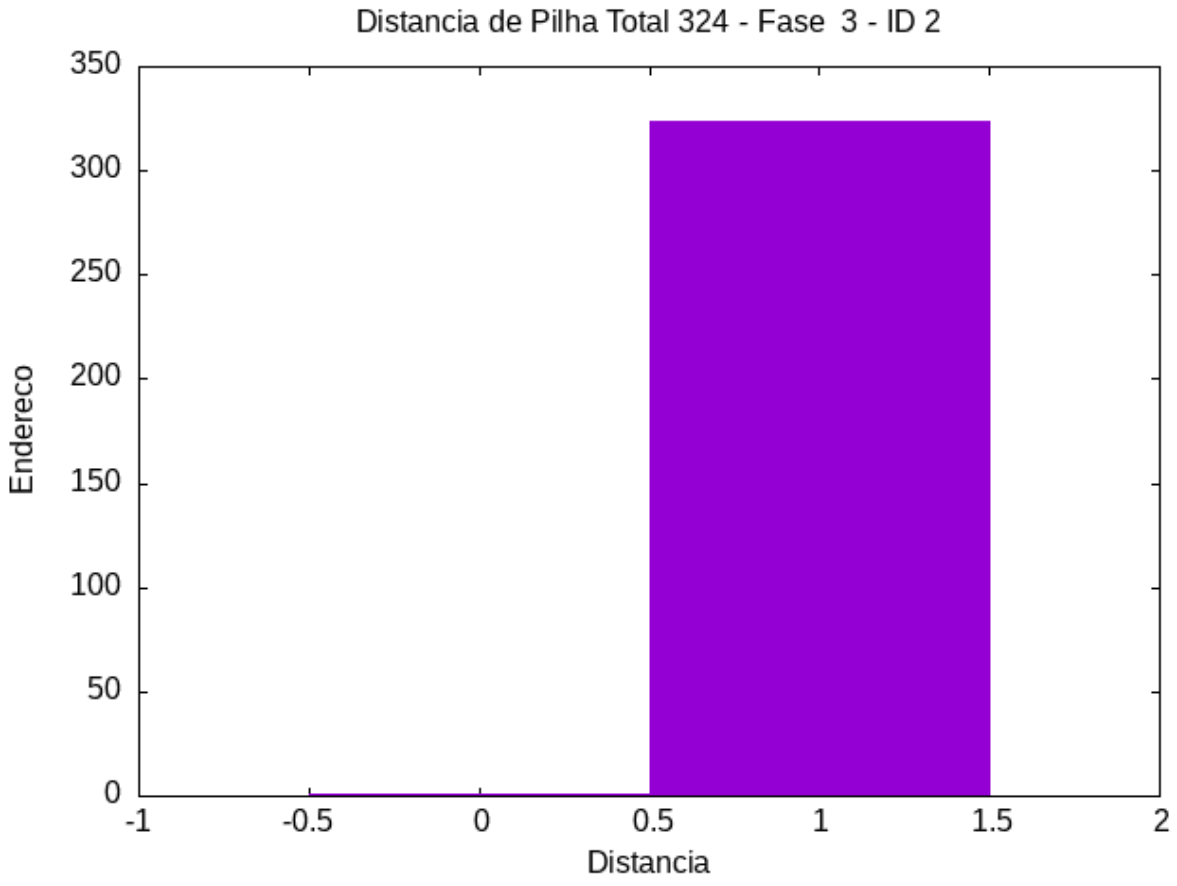

Distancia de Pilha Total 72 - Fase 2 - ID 0
Endereco
Distancia
30
25
20
15
10
5
0
-1
-0.5
0
0.5
1
1.5
2
2.5
3


Distancia de Pilha Total 930 - Fase 2 - ID 1
Endereco
Distancia
900
800
700
600
500
400
300
200
100
0
-1
-0.5
0
0.5
1
1.5
2
2.5
3


Distancia de Pilha Total 2362 - Fase 2 - ID 2
Endereco
Distancia
2500
2000
1500
1000
500
0
-1
0
1
2
3
4
5
6


Distancia de Pilha Total 63 - Fase 3 - ID 0
Endereco
Distancia
35
30
25
20
15
10
5
0
-1
-0.5
0
0.5
1
1.5
2
2.5
3


Distancia de Pilha Total 99 - Fase 3 - ID 1
Endereco
Distancia
100
90
80
70
60
50
40
30
20
10
0
-1
-0.5
0
0.5
1
1.5
2


Distancia de Pilha Total 324 - Fase 3 - ID 2
Endereco
Distancia
350
300
250
200
150
100
50
0
-1
-0.5
0
0.5
1
1.5
2

Dessa forma, conclui-se que o sistema manteve uma baixa distância de pilha durante toda a execução, favorecendo a hipótese do acesso em posições contíguas de memória e sendo uma forte prova de que o programa preza por uma boa localidade de referência.

6. Conclusões

Com o intuito de simular um jogo de poker entre alunos de uma sala, foi projetado, implementado, pensado e testado um sistema de computação que, dado um conjunto de rodadas, jogadores, cartas e apostas, computa quem são os ganhadores de cada rodada e quem terminou com mais dinheiro.

Durante o projeto do sistema foram levadas em consideração não só aspectos lúdicos do jogo de poker, mas também como a linguagem de programação escolhida poderia ser uma ferramenta útil para chegar no objetivo esperado. Toda a questão de mapear um mini-mundo de interesse em um modelo computacional robusto se mostrou bastante produtiva, levando o aluno a pensar em formas criativas de se resolver o problema, como, por exemplo, armazenar o critério de desempate junto ao jogador, possibilitando os desempates de maneira mais simples. Por fim, o tempo extra usado para projetar o sistema trouxe várias recompensas no sentido da implementação, sendo um aspecto a ser levado para trabalhos futuros.

Além disso, os testes e a análise de complexidade se mostraram muito úteis para entender como as entradas influenciam a execução do programa e como alguns detalhes de implementação podem ser significativos em termos de desempenho computacional e localidade de referência.

Nesse sentido, todo o fluxo de trabalho foi essencial para a consolidação de conteúdos aprendidos em sala, além de apresentar, de forma prática, como softwares maiores, mais consistentes e robustos são projetados e implementados.

7. Bibliografia:

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

IBM, Documentação da Linguagem C. Class Templates
https://www.ibm.com/docs/en/zos/2.2.0?topic=only-class-templates-c

8. Instruções para compilação e execução:

8.1 Compilação

Existem partes do programa que são compatíveis apenas às versões mais recentes da linguagem c++, dito isso, deve-se seguir as seguintes configurações para a compilação:

Linguagem: C++
Compilador: Gnu g++
Flags de compilação: -std=c++11 -g
Versão da linguagem: standard C++11
Sistema operacional (preferência): distribuições baseadas no kernel Linux 5.15.

O comando para compilar o programa automaticamente está presente no arquivo **"Makefile"** e sua execução é chamada pelo comando **"make all"**. Ainda assim, seguem as instruções para compilar manualmente:

Para gerar o executável do programa, é necessário, primeiro, gerar o objeto para cada arquivo presente na pasta "/src". Tal objetivo pode ser alcançado seguindo os seguintes comandos em ordem:

```
g++ -std=c++11 -g -Wall -c src/jogadas.cpp -o obj/jogadas.o -I./include/
g++ -std=c++11 -g -Wall -c src/memlog.cpp -o obj/memlog.o -I./include/
g++ -std=c++11 -g -Wall -c src/jogador.cpp -o obj/jogador.o -I./include/
g++ -std=c++11 -g -Wall -c src/rodada.cpp -o obj/rodada.o -I./include/
g++ -std=c++11 -g -Wall -c src/main.cpp -o obj/main.o -I./include/
g++ -std=c++11 -g -Wall -c src/carta.cpp -o obj/carta.o -I./include/
```

Após esse passo, deve-se juntar todos os objetos em um único arquivo executável, seguindo o comando:

```
g++ -std=c++11 -g -Wall -o ./bin/tp1.out ./obj/jogadas.o ./obj/memlog.o ./obj/jogador.o ./obj/rodada.o ./obj/main.o ./obj/carta.o
```

Deste modo, o executável "/bin/tp1.out" estará compilado e pronto para ser utilizado.

8.2 Execução

De maneira análoga, existe um comando no “Makefile” que executa o compilável, chamado **“make exec”**, que se certifica que os objetos foram gerados e que o compilável existe. Ainda assim, seguem as instruções para a execução manual:

1. Certifique-se que o compilável foi gerado de maneira correta, se algum problema ocorrer, execute o comando “make all” presente no “Makefile”.
2. Dado que o compilável foi gerado de maneira correta, certifique-se que existe um arquivo “entrada.txt” na raiz do projeto, é esse arquivo que “alimenta” o programa. Se ele não existir, crie-o.
3. Uma vez que os passos anteriores foram cumpridos, execute o programa com o comando: ./bin/tp1.out
4. A saída estará guardada no arquivo “saida.txt”, na raiz do projeto.