Trabalho Prático 1
Pokerface

Guilherme Mota Bromonschenkel Lima - 2019027571

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

guilhermebkel@ufmg.br

1. Introdução

O problema proposto foi implementar um programa que cuidasse da etapa de decisão do jogo de Pôquer, onde é possível saber quem ganhou a partida, além do montante de dinheiro virtual que cada jogador terá ao final das rodadas.

Ao executar o programa, ele faz a leitura de um arquivo composto por representações de uma partida, ou seja, esse arquivo contém uma lista de rodadas, que são um conjunto de jogadas e, cada jogada traz informações do jogador, sua aposta e mão.

Ao fim da execução do programa, é criado um arquivo de saída que contém o resultado da rodada, além do montante final de cada um dos jogadores, ordenado por ordem decrescente.

Durante a execução do programa, fizemos uso de um módulo chamado **memlog** para fazer diversos registros de acesso de memória. Além disso, para realizar a análise experimental foi necessário utilizar a biblioteca **analisamem**, que é responsável por pegar os dados gerados pela biblioteca **memlog** e tratá-los para uma melhor visualização.

Por fim, fazemos uso também de uma ferramenta chamada **gprof** para verificar métricas de desempenho computacional, como a duração de chamadas de funções do programa.

2. Implementação

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

Os testes do programa foram realizados no sistema operacional Ubuntu (versão 18.04).

2.1. Estrutura de Dados

Para que o programa pudesse funcionar da melhor maneira possível, foram criados os seguintes modelos: Balance (montante do jogador), Round (informações da rodada), Play (informações da jogada), Hand (cartas na mão do jogador para determinada jogada).

Com exceção do modelo Hand, todos os outros modelos foram criados através do uso de structs, adicionando informações específicas de seu contexto.

Sendo assim, como a partida é composta por uma combinação entre os vários modelos criados, foi necessário fazer o uso de uma estrutura de dados de lista para armazenar todas as

informações. Vale ressaltar que essa estrutura de dados foi adaptada para tornar mais fácil seu uso, além do entendimento dos métodos presentes nela.

2.2. Formato dos dados

Durante a execução do programa é feita a leitura de um arquivo de entrada para obter os dados que serão utilizados mais tarde.

Nesse sentido, vale ressaltar que durante a leitura do arquivo de entrada nós buscamos os dados de acordo com a formatação esperada para eles. Por fim, durante essa busca, nós continuamente salvamos em memória os dados encontrados utilizando a nossa classe principal PokerFace - que é responsável por efetuar as regras de negócio.

Ao fim do programa, usamos novamente essa classe principal para efetuar todas as regras em cima dos dados obtidos e gerar um arquivo de saída com o resultado final.

2.3. Modularização

Para garantir uma melhor qualidade de código e facilidade de manutenção, temos o programa como um todo dividido em 7 módulos com obrigações diferentes:

O módulo **poker-face** (é a classe responsável por realizar as regras de negócio para gerar os resultados da partida através dos dados de entrada).

O módulo **poker-face-util** (é o contexto responsável por fornecer métodos utilitários para ajudar no processamento das regras de negócio do **poker-face**).

O módulo **poker-face-validation** (é o contexto responsável por fornecer métodos que validem os dados que são utilizados durante a execução do **poker-face**).

O módulo **shared-util** (é o contexto responsável por fornecer métodos utilitários compartilhados entre todos os módulos existentes).

O módulo **arrangement-list** (é a estrutura de dados de lista utilizada pelo **poker-face** para gerenciar todos os modelos durante a execução do programa).

O módulo **memlog** (realiza a gravação de acessos de memória durante a execução do programa)

O módulo **app** (funciona como um orquestrador do programa, fazendo a leitura dos dados de entrada e utilizando o **poker-face** para processar esses dados e gerar um arquivo de saída).

3. Análise de Complexidade

A análise de complexidade do programa foi realizada utilizando como base o módulo **app**, que é o ponto de entrada de execução de todas as regras de negócio. Nesse caso, como não temos nenhuma chamada recursiva, não é necessário realizar uma análise da complexidade de espaço do programa.

3.1. Tempo

Dentro do módulo do **app**, ocorre a execução de alguns loops e dos métodos **PokerFace.startRound**, **PokerFace.readPlay** e **PokerFace.finish**, ou seja, vamos iniciar analisando a complexidade de cada um desses métodos para depois analisar a complexidade do módulo **app** como um todo.

No método **PokerFace.startRound**, todo o conjunto de operações executados nos dão uma complexidade $O(1)$. Ou seja, nesse método temos a seguinte complexidade:

Melhor caso = Pior Caso = $O(1)$

No método **PokerFace.readPlay**, as ações executadas de maior custo são: buscamos na lista de rodadas pela chave $O(1)$ / $O(n)$, preenchemos uma lista de cartas $O(n)$, ordenamos a lista de cartas $O(n^2)$, atualizamos a lista de cartas $O(n)$, verificamos se o modelo de montante do usuário já existe para criarmos caso for necessário $O(n)$, atualizamos a rodada usando sua chave $O(n)$.

Ou seja, teremos a seguinte complexidade para este método:

Melhor caso = Pior caso = $O(n^2)$

No método **PokerFace.finish**, executamos dentro de um loop de n-repetições com as seguintes ações: buscar uma rodada pela chave $O(1)$ / $O(n)$, consolidar o resultado de uma rodada $O(n^2)$ / $O(n^3)$. Fora do loop, nós ordenamos a lista de montante dos usuários $O(n^2)$.

Ou seja, nesse método temos a seguinte complexidade:

Melhor caso = $O(n^3)$

Pior caso = $O(n^4)$

Agora que analisamos cada um dos métodos, podemos voltar para a análise do módulo **app**.

Temos um loop inicial de n-repetições com as seguintes ações: PokerFace.startRound $O(1)$. Dentro desse loop, temos outro loop aninhado de n-repetições com as seguintes ações: salvar as cartas de uma jogada $O(n)$, PokerFace.readPlay $O(n^2)$.

Ou seja, nessa primeira parte do módulo **app** temos a seguinte complexidade:

Melhor caso = Pior caso = $O(n^4)$

Fora do loop, nós executamos as seguintes ações: PokerFace.finish $O(n^3)$ / $O(n^4)$, salvamos os resultados de cada rodada $O(n^2)$, salvamos o montante resultante de cada jogador $O(n)$.

Ou seja, na segunda parte do módulo **app** temos a seguinte complexidade:

Melhor caso = $O(n^3)$

Pior caso = $O(n^4)$

Ou seja, podemos concluir que a nossa complexidade temporal para o módulo do **app** como um todo pode ser dada por:

Melhor caso = Pior caso = $O(n^4)$

4. Estratégia de Robustez

Jogos no geral tendem a ter diversas regras de negócio para trazerem os resultados com base em algum dado de entrada que fornecemos. Portanto, houve uma preocupação maior na garantia de que todos os métodos da classe principal do PokerFace fossem executados conforme o esperado.

Nesse sentido, foram adicionadas algumas validações para garantir a correta execução dos métodos da classe (por exemplo, quando executamos o método finish, não é possível ler uma nova jogada ou rodada, visto que a partida como um todo já finalizou).

Por fim, foram realizados alguns testes de sanidade dentro do método que consolida o resultado da rodada. Por exemplo, quando um jogador sem dinheiro tenta contribuir com o pingo ou aposta, ocorre uma invalidação da rodada como um todo.

5. Testes

Foi realizada uma série de testes para garantir que todas as operações estão sendo executadas corretamente. Logo abaixo é possível ver uma entrada que foi utilizada em um dos testes:

- Entrada:
3 1000
5 50
Giovanni 100 6O 3P 10E 11O 1O
John 200 3P 4E 3E 13C 13O
Thiago 100 12O 7P 12C 1O 13C
Gisele 300 12E 10C 11C 9C 13E
Wagner 50 5P 12P 5E 2E 1P
2 50
Wagner 200 2P 13E 9E 12C 2O
Gisele 350 11P 9P 2E 6E 4P
3 100
Thiago 250 1O 4P 1E 3O 8O
Gisele 100 9C 8C 8C 2C 6C
Giovanni 150 4P 12P 8E 12E 2P

Com a entrada acima, fomos capazes de obter a saída esperada.

- Saída
1 1000 S
Gisele
1 800 OP
Wagner
1 1000 F
Gisele
####
Gisele 2050
Wagner 1350
John 600
Giovanni 550
Thiago 450

6. Análise Experimental

Agora, podemos efetuar uma análise de desempenho computacional e eficiência de acesso à memória.

6.1. Desempenho Computacional

Para esse experimento, foram utilizadas partidas de 1000, 2000, 3000, 4000 e 5000 rodadas, com 5 jogadores em cada.

Vamos usar a métrica de tempo de execução de cada operação como métrica relevante para a avaliação do nosso programa, visto que é uma métrica que nos possibilita entender o custo de cada execução na prática e fazer uma comparação com a análise de complexidade que fizemos anteriormente. Sendo assim, o tempo de execução (em segundos) para cada quantidade de rodadas pode ser visto na tabela abaixo:

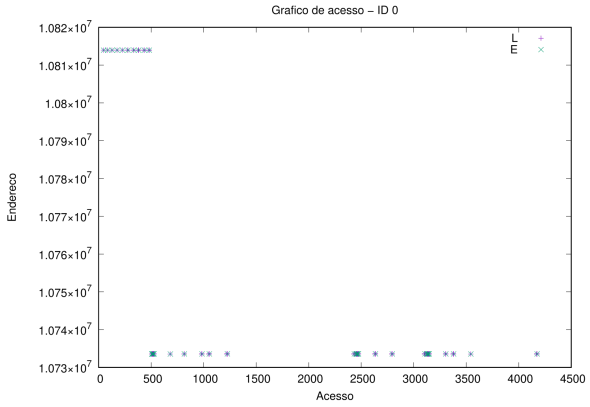
Quantidade de Rodadas	**Tempo de execução** *(segundos)*
1000	0.150690784
2000	0.479448848
3000	1.658090399
4000	2.434718022
5000	3.071523652

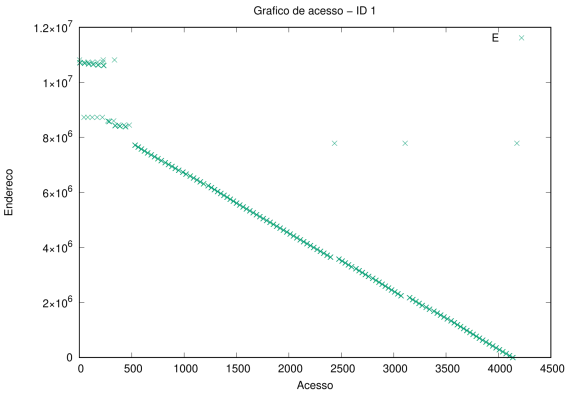
Conforme esperado na análise de complexidade, ocorre um aumento não linear no tempo de execução do programa conforme aumentamos a quantidade de rodadas.

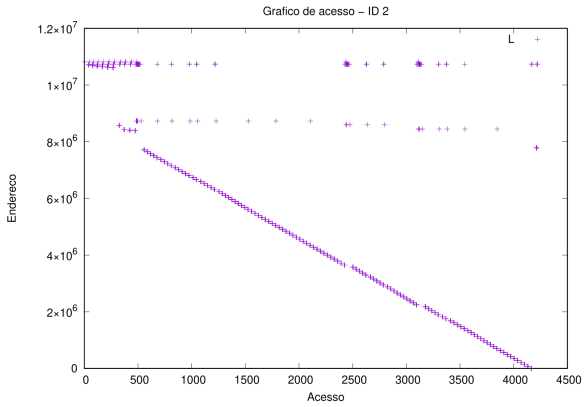
Por fim, ao utilizar o **gprof** para analisar cada operação efetuada em na partida de 5000 rodadas, é possível entender que o maior tempo de execução vem do método **PokerFace.finish**, mesmo que ele seja executado apenas uma vez. Isso era esperado pois, conforme vimos na análise de complexidade do programa, esse método é o que possui a maior ordem de complexidade.

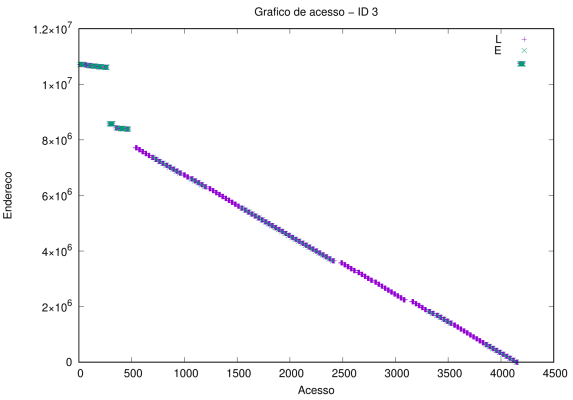
Nesse sentido, pode ser interessante encontrar outra forma de otimizar as regras de negócio que ocorrem no método **PokerFace.finish**, com o intuito de diminuir a complexidade temporal desse método tanto no melhor caso $O(n^3)$ quanto no pior caso $O(n^4)$.

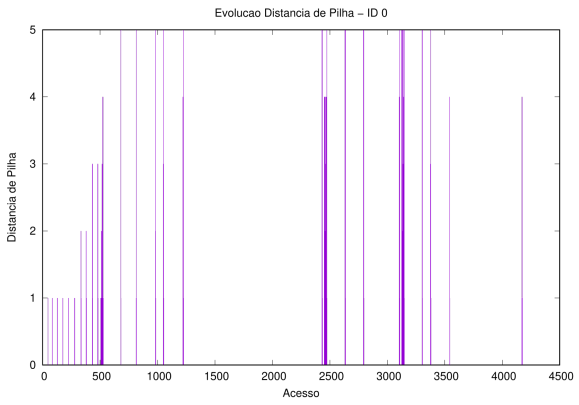
6.2. Análise de padrão de Acesso à Memória e Localidade de Referência

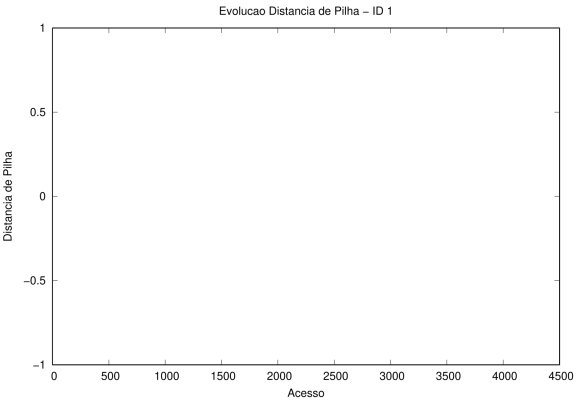


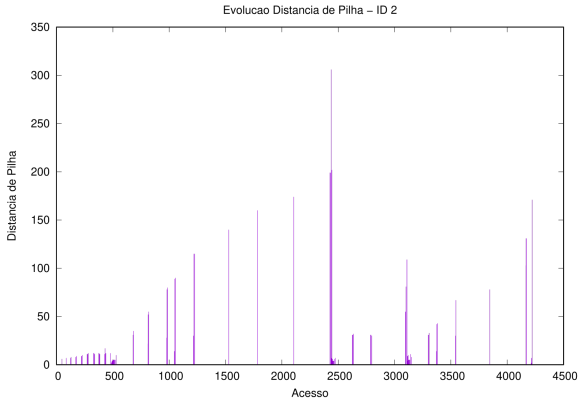


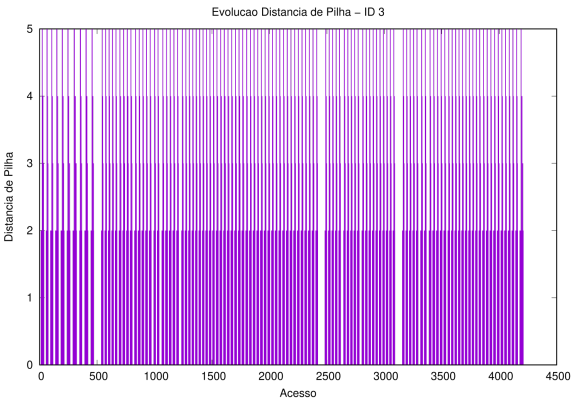


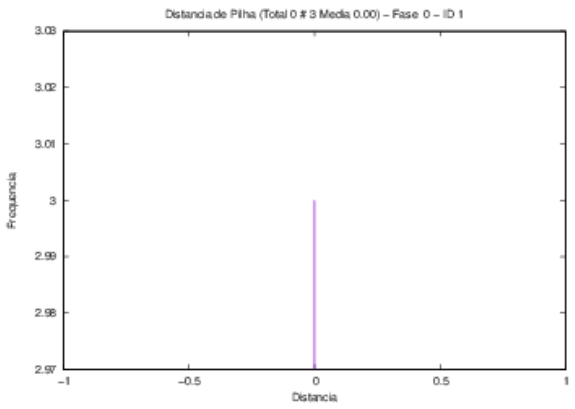


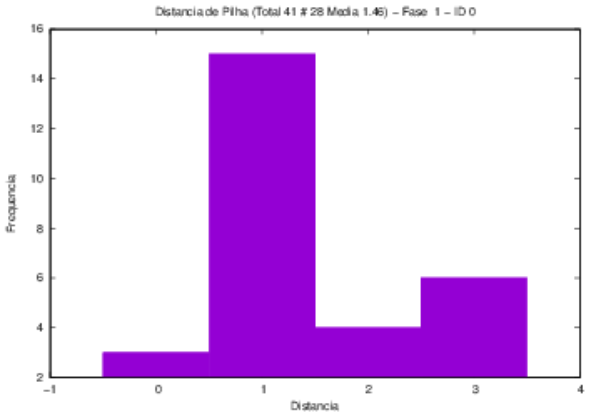


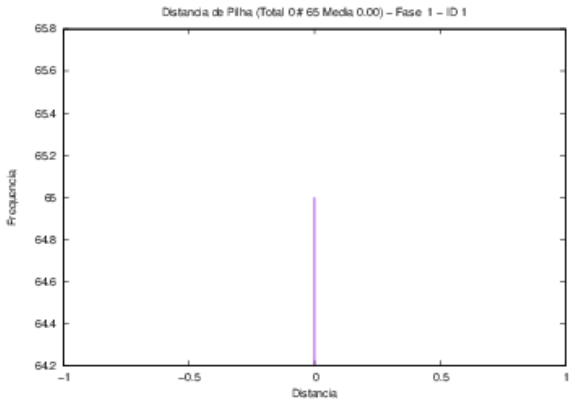


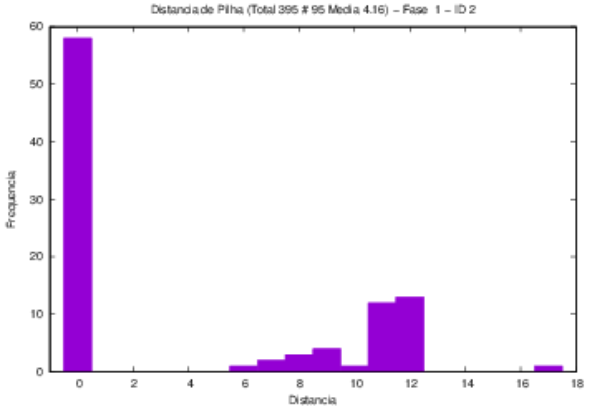


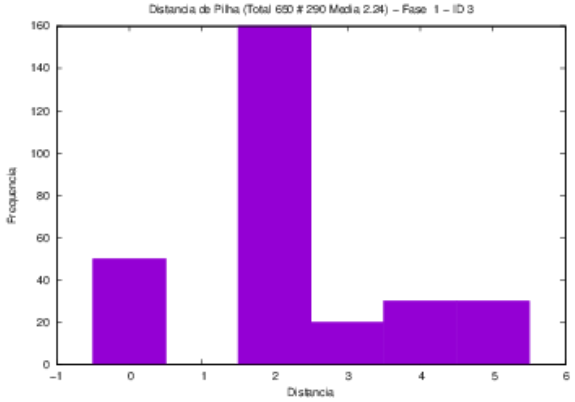


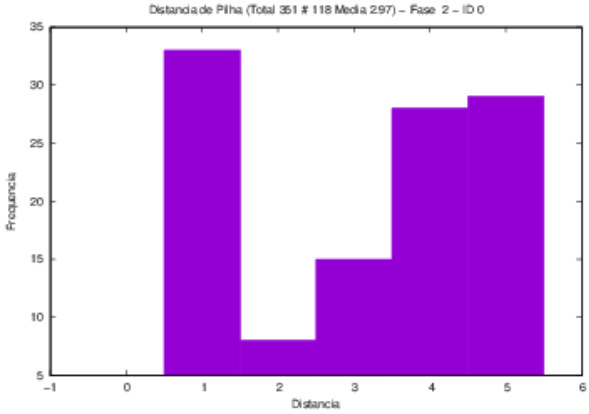

Distancia de Pilha (Total 0 # 3 Media 0.00) – Fase 0 – ID 1
Frequencia
Distancia

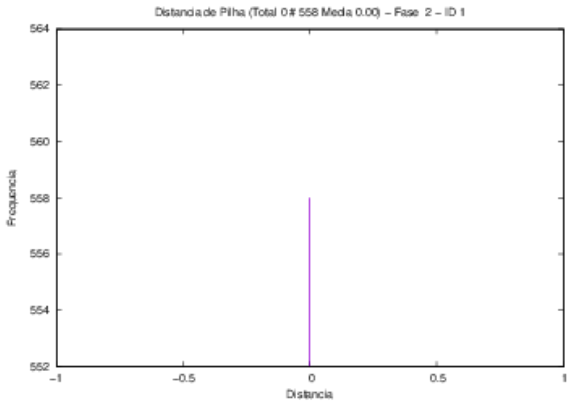

Distancia de Pilha (Total 41 # 28 Media 1.46) – Fase 1 – ID 0
Frequencia
Distancia

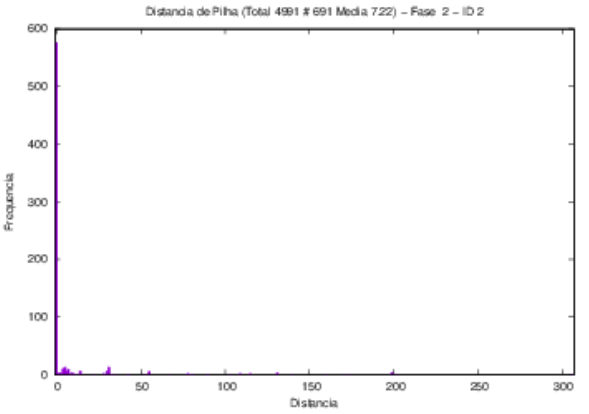

Distancia de Pilha (Total 0 # 65 Media 0.00) – Fase 1 – ID 1
Frequencia
Distancia


Distancia de Pilha (Total 395 # 95 Media 4.16) – Fase 1 – ID 2
Frequencia
Distancia


Distancia de Pilha (Total 650 # 290 Media 2.24) – Fase 1 – ID 3
Frequencia
Distancia


Distancia de Pilha (Total 351 # 118 Media 2.97) – Fase 2 – ID 0
Frequencia
Distancia


Distancia de Pilha (Total 0 # 558 Media 0.00) – Fase 2 – ID 1
Frequencia
Distancia


Distancia de Pilha (Total 4991 # 691 Media 7.22) – Fase 2 – ID 2
Frequencia
Distancia

Nos gráficos acima é possível ver os dados de acesso a memória que ocorrem utilizando a estrutura de dados de lista, que é responsável por guardar todos os modelos necessários para que a aplicação funcione.

Ao observar os gráficos de acesso de memória do método de criação de um item na lista, é possível perceber que os endereços de acesso estão muito próximos.

Isso ocorre pois como estamos utilizando um vetor estático para alocar os dados da nossa lista, o vetor já é criado com posições de dados que estão próximas, ou seja, ao adicionar novos elementos nesse vetor, apenas adicionamos ele do lado do último e, dessa forma, garantimos que os endereços mantenham-se próximos.

Além disso, é possível perceber nos gráficos de acesso de memória de quando estamos realizando uma pesquisa na estrutura de dados, que o comportamento de endereçamento de memória é bem próximo do que ocorre na criação do item no vetor.

Isso também ocorre pelo mesmo motivo de que estamos usando uma lista estática e, portanto, como durante uma pesquisa percorremos um item de cada vez da esquerda para a direita, estamos acessando endereços próximos.

Durante as operações de atualização dos dados que estão no vetor, nem sempre é possível manter uma proximidade dos endereços de acesso, visto que nem sempre um elemento que foi encontrado anteriormente, estará próximo de outro elemento que será encontrado em seguida.

Podemos ver que, no geral, é possível manter uma distância de pilha bem pequena e, isso ocorre devido ao fato de estarmos usando um vetor estático para a nossa estrutura de lista. Portanto, como esse vetor já tem seu espaço alocado no tamanho correto desde sua inicialização, a posição de seus itens fica em localidades próximas.

7. Conclusões

Nesse trabalho prático, fizemos um programa que recebe os dados de uma partida de Pôquer e devolve qual o resultado de cada rodada, além do montante final de cada jogador. Inicialmente fizemos uma análise de complexidade do programa para entender se o comportamento estaria acontecendo dentro do esperado. Além disso, executamos alguns testes para garantir que o programa estava funcionando de forma correta. Em seguida, através da execução desse programa, conseguimos gerar dados de desempenho computacional Por fim, fomos capazes de tratar os dados gerados e analisá-los para entender o comportamento de forma mais aprofundada.

Diante disso, fomos capazes de entender como a escolha da estrutura de dados mais adequada pode facilitar o processo de desenvolvimento do programa. Além disso, foi possível entender na prática como o tipo de algoritmo de ordenação utilizado tem um impacto direto na estabilidade e desempenho do programa.

Por fim, vale ressaltar que, durante a execução do trabalho, a principal dificuldade surgiu mais no sentido de conseguir analisar os dados encontrados e entender o porquê deles se comportarem de tal maneira.

Apêndice

Instruções para compilação e execução

Antes de efetuar a compilação do programa principal, é necessário abrir a pasta **analisamem** e rodar o comando **make**.

Agora, para efetuar a compilação do programa, basta entrar na pasta **TP** e rodar o comando **make** em seu terminal.

Esse comando irá executar as tarefas presentes no arquivo **Makefile** e efetuar tanto a compilação quanto a execução do programa.