**Problema das mãos de poker da disciplina de paradigmas da programação.**

Ana Beatriz

Anderson Mateus

Marcos Eduardo

UFRPE – Universidade Federal Rural de Pernambuco

Departamento de Estatística e Informática

Campus Dois Irmãos (Sede) – Recife – PE - Brasil

**Abstract.** *This report involves the resolution of the "poker hand problem" and the various aspects related to its development.*

**Resumo.** *Esse relatório envolve a resolução do “problema das mãos de poker” e os vários aspectos relacionados ao seu desenvolvimento.*

1. **Informações preliminares**

Nesta etapa, iremos implementar o problema anterior utilizando o paradigma **Funcional**. Tendo como objetivo base, comparar o desempenho desta implementação, a anterior, discernindo os pontos em que ambas apresentam variação nos dados colhidos para o desempenho do código.

1. **Linguagem escolhida**

Visando uma proximidade as linguagens que dominamos, uma curva menor de aprendizagem e a aplicação possível ao paradigma Funcional, selecionamos a linguagem **Scala**, que é uma linguagem **multiparadigma**. Iremos a seguir apresentar algumas informações sobre a mesma.

1. **História da Linguagem**

Scala surgiu como objetivo para validar duas hipóteses, sendo a primeira: As linguagens de programação para componentes precisam ser escaláveis, onde os mesmos conceitos devem descrever partes pequenas e grandes, e a segunda hipótese: A escalabilidade pode ser conseguida unificando e generalizando conceitos de programação funcional e orientada a objetos.

Scala foi desenvolvida em 2001 por Martin Odersky e pelo grupo dele na École Polytechnique Fédérale de Lausanne (EPFL), Lausana na Suíça. A meta era combinar programação orientada a objetos e programação funcional, mas sem as restrições impostas pela linguagem Java. O primeiro passo foi o Funnel, uma linguagem minimalista baseada em redes funcionais.

Surgiu assim Scala, que trouxe algumas das ideias de Funnel, colocando dentro de uma linguagem mais pragmática com foco especial no funcionamento com plataformas padrões. Scala não é uma extensão de Java, mas é completamente inter operável com ele. O design do Scala começou em 2001. Um primeiro lançamento ao público foi em 2003. Em 2006, uma segunda versão remodelada foi lançada como Scala v 2.0. Desde então a linguagem tem ganhado popularidade. Atualmente a linguagem está na versão 2.12.3.

1. **Paradigmas Envolvidos**

A linguagem Scala é multiparadigma, podendo implementar os paradigmas orientação a objetos e funcional. Porém Scala é puramente orientada a objetos, no sentido em que todos os valores são objetos, e permite paradigma funcional no sentido de que cada função é um valor.

1. **Vantagens e Desvantagens**

Assim como toda linguagem possui suas vantagens e desvantagens, Scala não é diferente, as suas principais vantagens são: **Objetos imutáveis**, sendo um trunfo para reduzir a preocupação nas implementações thread-safe; **Velocidade na implementação**, pois Scala permite que o programador digite menos código para implementar o desejado; **Facilidade em resolver problemas de recorrência**, pois Scala possui uma biblioteca de “Ator”, específica para resolver problemas de concorrência mais rapidamente.

Já suas principais desvantagens são: **Ilegibilidade do código**, onde seu código em casos se trata apenas de poucas linhas que executam na verdade muitas coisas aninhadas; **Os programas podem ser menos eficiente**, por se tratar de um paradigma de abstração alta, a quantidade de coisas que rodam por trás do que foi implementado é imensa; **Difícil aprendizado, Se você já programa em Java, poderá ter dificuldades na adoção**, em função da abordagem diferenciada, tanto na sintaxe quanto no paradigma de programação que o Scala oferece.

1. **Possíveis Aplicações**

Scala é ideal para projetos funcionais, sistemas complexos de engenharia ou ambientes onde você precisa:

* Resolver problemas de concorrência;
* Executar tarefas DSL (Linguagem de Domínio-específico);
* Implementar soluções funcionais;
* Integrar com recursos compartilhados, arquivos, bancos de dados ou processos de back-end e serviços múltiplos;
* Aprender um novo paradigma de desenvolvimento;

1. **Relacionando o uso dos Aspectos Estudados**

**VARIÁVEIS E TIPOS**

**ESTRUTURAS DE CONTROLE E OPERADORES LÓGICOS**

1. **Referências**

http://ccsl.ime.usp.br/pt-br/uma-introducao-a-linguagem-scala

http://docs.scala-lang.org/?pageId=294934

http://www.scala-lang.org/

https://run.unl.pt/bitstream/10362/14782/1/Rodrigues\_2014.pdf - tcc

http://lampwww.epfl.ch/~odersky/talks/popl06.pdf

https://trilhandoscala.wordpress.com/2012/10/24/scala-os-pros-e-os-contras/

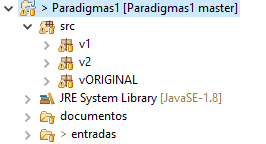
http://informacaocomdiversao.blogspot.com.br/2009/02/paradigma-funcional.html

1. **Informações preliminares**

Devido às habilidades disponíveis, foi escolhida a linguagem Java e o paradigma **OO** para a resolução do problema, porém buscando a melhor otimização da solução, a mesma apresenta uma mesclagem de abordagens em paradigma de **Orientação a Objetos** e **Descritivo**, no caso, estamos utilizando o mínimo de **abstração** possível, quanto a **encapsulamento**, sua utilização se resume aos métodos das classes, não utilizamos **herança** e **polimorfismo**, pois nesta solução não condiz o uso destas características para a otimização proposta par ao algoritmo. Esta decisão de não utilizar estas características do paradigma de orientação a objetos foi tomada visando um maior desempenho para o programa, dentro das classes, em alguns pontos, utilizamos objetos definidos pela biblioteca java para auxiliar o desenvolvimento da solução. O arquivo contendo o código fonte pode ser baixado através do link:

<https://github.com/MattLack/pokerprevision.git>

Para executar o código, basta importar o projeto para o Eclipse (Qualquer versão), e executar a classe PokerPrevision, que contém o método main(), classe presente em qualquer um dos pacotes de versão disponíveis no projeto. Não esquecer de apontar na linha 49, o endereço do arquivo de testes no qual deseja-se utilizar.



A imagem acime descreve os diretórios do projeto. Dentro do diretório **src**, o pacote **vORIGINAL**, contém a versão original do algoritmo, os demais **v1** e **v2** contém respectivamente versões diferentes do projeto, com alterações de tipo e estruturas de controle. O diretório **entradas** contém as entradas teste disponibilizadas pelo professor, sendo estas, de mil, um milhão de entradas e cem milhões de entradas (pokerK, pokerM, poker100M respectivamente). E o diretório **documentos** contém alguns documentos referentes ao projeto.

1. **Como o problema foi resolvido**

Ao tratar o problema com java e OO, foi criada uma classe ***Testes*** onde os métodos de reconhecimento das mãos foram aplicados, exceto a lógica de jogo e a divisão de mãos, que por sua vez, são feitas na classe ***Logic***. Deste modo, foram criados os métodos em ordem decrescente em nível de força, **Royal Flash, Straight Flush, Quadra, Full House, Flush, Sequencia, Trinca, Dois pares, Um par e Carta alta.**

Os métodos em sua maioria recebem o valor e o naipe com exceções do **Flush, Sequência** que só precisa receber o naipe. O Array “*valor*“refere-se aos 5 valores das cartas que a mão contém, estes valores são antes ordenados para que possam ser transmitidos aos métodos de validação, com a finalidade de facilitar na decisão da jogada. O Array “*naipe*” contém os 5 valores referentes respectivamente aos naipes, que podem ser quatro e com os seguintes valores: **H, C, S e D**.

Na classe ***Logic*** é feita toda parte de divisão das cartas nas mãos dos jogadores, e implementado que as cartas **T, J, Q , K e A** são substituídas respectivamente para 10, 11, 12, 13 e 14 para conseguirmos estabelecer a força de uma sobre a outra, além de receber os arrays de valores e naipes como entrada nos métodos de teste, que verificarão os mesmos até achar um caso em que todas as opções listadas nas condições sejam verdade, caso não, irá prosseguir até que o fluxo seja totalmente concluído ou não haja mais opções em que os testes se enquadrem.

Adicionalmente, alguns dos métodos possuem uma opção de desempata, que podem receber valores de ambas as mãos, com a finalidade de decidir em qual opção o jogador 1 vence.

Basicamente, as classes são instanciadas utilizando a abordagem de singleton, para otimizar a alocação de memória para as classes e, é através da leitura do arquivo, na classe ***PokerPrevision***, que o uso do singleton transmitindo a leitura para a classe Logic, que divide a mão e sequencia a jogada, que o programa de fato cumpre a sua proposta.

A classe Testes possui um main próprio, você pode utilizá-lo para testar os métodos implementados.

# 3. Relacionando o uso dos aspectos estudados

**Variáveis e Tipos**

Java é uma linguagem fortemente tipada, somos então obrigados a declarar cada variável e seu tipo. Nem sempre precisamos inicializar (atribuir valores) estas variáveis, mas é mandatório em java, declarar nome e tipo, como no trecho de código a seguir, presente no início da classe Logic.

String hand1;

String hand2;

**int**[] valorHAND1 = **new** **int**[5];

Outro exemplo, referente a um dos métodos da classe Testes:

**byte** cont = 0;

**int** value = 0;

Estes são exemplos de variáveis declaradas e inicializadas. Em várias ocasiões, as variáveis podem ser apenas declaradas, e não inicializadas. Porém, antes de serem “usadas” para algo, devem ter um valor associado a elas, ou atribuídos logo ao início da execução de um procedimento que depende o valor a ser transmitido a ele, ou seja, devem receber um valor. Não devemos confundir, porém, variáveis com tipos primitivos.

Veja a seguir um trecho de código da classe PokerPrevision:

fr = **new** FileReader(FILENAME);

br = **new** BufferedReader(fr);

Como podemos ver, uma variável pode ser associada tanto a primitivos quanto a tipos abstratos de dados. Assim sendo, uma **variável** é uma abstração a uma posição de memória e o **tipo** é o tipo de dado contido em tal posição de memória. Existem linguagens em que a declaração de tipo não é necessária (python por exemplo). Não é o nosso caso.

Vamos agora falar das características de uma variável usando um trecho do código da classe Testes:

**int** pares = 0;

**for** (**int** i = 1; i < 5; i++) {

**if** ((aux[i] == aux[i - 1]) && (aux[i] != pares)) {

**if** (pares > 0)

pares = aux[i];

**else** **if** (pares == 0) {

pares = aux[i];

}

}

}

Analisando esse pequeno trecho de código, podemos observar detalhes interessantes. A primeira é que cada variável tem um **nome** e “pares”, por exemplo, é o nome de uma variável.

Cada variável tem um **tipo**. Como podemos observar, a variável “i”, é do tipo **int**.

Cada variável tem também um **endereço** na memória para o qual seu nome aponta, como exemplo, no caso, i = “1” , o endereço é a posição de memória onde o “1” está gravado. Adicionalmente, o “1” em questão é o **valor** da variável.

Certas variáveis tem um **escopo**, que nada mais é do que onde esta variável pode ser vista. A variável “i” tem escopo no bloco de execução “for”. Fora desse laço elas não existem. A variável “i” também possui um **tempo de vida**, ou seja, elas só existem enquanto o laço “for” está sendo executado.

**ESTRUTURAS DE CONTROLE E OPERADORES LÓGICOS**

Foram utilizadas algumas estruturas de controle no código. Exemplificaremos algumas:

**if** (v == **true** && valort == 5) {

**return** **true**;

} **else** {

**return** **false**;

}

Vemos a estrutura “if (testa condição)”. Ela executa o código que está contido nela apenas se a condição for verdadeira. A estrutura “else”, tem uso opcional, e irá tratar do que acontece caso o “if” não aconteça. Reparar que podem ser utilizados vários “if” dentro do código, de forma encadeada.

Note o operador *&&* no trecho do código acima. Esse operador é o chamado **curto circuito**, caso a primeira condição retorne “false”, ele interrompe imediatamente a cadeia de testes, e passa para o próximo trecho de código. Numa iteração muito grande, isso pode significar uma redução considerável de custo de processamento.

**while**(0<=i){

**if** (valor1[i] > valor2[i]) {

**return** 1;

} **else** **if** (valor2[i] > valor1[i]) {

**return** 2;

}

i--;

}

**for** (**int** i = 1; i < aux.length; i++) {

**if** (aux[i] == aux[i - 1]) {

**return** aux[i];

}

}

Agora vemos o uso de duas outras estruturas de controle bastante usadas, sobretudo para iterar sobre algo. A estrutura “while(condição)” executa um bloco de código enquanto uma condição de teste é verdadeira. A estrutura “for”, na realidade, é uma forma de “while” muito usada para iterar sobre um determinado dado (tipicamente strings, arrays, sets, etc...), cujos elementos contidos no dado podem ser acessados individualmente.

Nos dois exemplos acima vimos comparações e atribuições, descritos pelos operadores “=” e “==”, respectivamente.

**PASSAGEM POR VALOR E POR REFERÊNCIA, MODULARIZAÇÃO**

Certas funções e atribuições no código permitem que os dados sejam passados tanto por valor quanto por referência. Vejamos como exemplo, uma das funções da classe TESTES:

**public** **boolean** testFLUSH(**char**[] naipe) {

**int** v = 0;

**char** naipev = naipe[0];

**for** (**int** i = 0; i < 5; i++) {

**if** (naipe[i] == naipev) {

v++;

}

}

**if** (v == 5) {

**return** **true**;

} **else** {

**return** **false**;

}

}

No exemplo acima, a função recebe a referência a um array do tipo **char**.

O código foi modularizado, e alguns módulos utilizam o retorno de outros, com a finalidade de agilizar o desenvolvimento e o desempenho do código, buscando implementar o mínimo de repetições de código possíveis.

**TESTES COMPARATIVOS DE PERFORMARCE:**

Agora testaremos o código implementado e o desempenho do mesmo em relação.

Para definir o arquivo de entrada, você deve seguir para a classe **POKERPREVISION** e dirigir-se ao método **MAIN**, neste por sua vez, você identificará uma chamada ao método ***PokerHands***, através dele, você deve dizer o diretório e o arquivo a ser lido.

Atualmente dentro do diretório entradas, que está dentro do projeto, temos os arquivos de mil valores e um milhão de valores, para usá-los em seus testes, basta apenas trocar o último caractere referente ao nome do arquivo, sendo K para o de mil e M para o de um milhão de entradas.

A máquina de testes é um notebook Asus, processador core i7 – 4720HQ 2.6GHz – com turbo bust para 3.6GHz, contendo 4 núcleos físicos e mais 4 virtuais, com 24Gb de memória RAM, rodando Windows 10 e Java 8.

As tabelas a seguir apresentam os resultados colhidos com mil, um milhão e cem milhões de entradas respectivamente, para as três versões do código. São colhidos três momentos (tempos) em escala de segundos, para cada versão e arquivo de número de entradas e, ao final, é apresentado uma média destes tempos, para servir melhor como medida de comparação.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Entrada 1K | Tempo1 /s | Tempo2 /s | Tempo3 /s | Média Final /s |
| vORIGINAL | 0.031 | 0.016 | 0.015 | 0.0206 |
| V1 | 0.032 | 0.015 | 0.015 | 0.0206 |
| V2 | 0.016 | 0.015 | 0.015 | 0.0153 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Entrada 1M | Tempo1 /s | Tempo2 /s | Tempo3 /s | Média Final /s |
| vORIGINAL | 0.573 | 0.579 | 0.641 | 0.598 |
| V1 | 0.484 | 0.484 | 0.516 | 0.495 |
| V2 | 0.703 | 0.766 | 0.688 | 0.719 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Entrada 100M | Tempo1 /s | Tempo2 /s | Tempo3 /s | Média Final /s |
| vORIGINAL | 50.014 | 49.969 | 48.485 | 49.49 |
| V1 | 40.057 | 40.263 | 40.902 | 40.407 |
| V2 | 57.488 | 56.965 | 57.585 | 57.346 |

Como podemos notar, para o arquivo de entrada 1K, a vORIGINAL e a v1 apresentaram o mesmo desempenho. Porém a diferença destas para a v2 é de 0.053 segundos, aproximadamente 20% de diferença.

Para a entrada de 1M podemos ter uma melhor observação sobre o desempenho de cada versão. A v1 apresentou o melhor desempenho, dentre as demais, com diferença de 0.103 para vORIGINAL e 0.224 para v2, já para a entrada de 100M esta diferença de fato se torna visível, sendo de cerca de 9 segundos para a vORIGINAL e 17 segundos para a v2.

Fica claro assim, que a alteração feita na v1 fez uma diferença significativa na performance do algoritmo. Esta alteração foi a substituição das estruturas “*for*” por estruturas “*while*”. E fica claro também que a linguagem Java não lida muito bem com um dado muito grande, a string de entrada conteria 20 milhões de caracteres, além de ficar claro que a utilização de uma mesclagem de orientação a objetos e descritivo, possibilitou a obtenção de um bom desempenho no tempo de execução do programa, isto unido a utilização de tipos primitivos para quase toda a gama lógica do programa.