UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

JOSÉ PEDRO MARTINEZ
LUCAS ASSIS
RÉGES OBERDERFER

Modelos de Linguagens de Programação - SchnorR Doidão: Galáxias em R

Relatório apresentado como requisito parcial para a obtenção de conceito na Disciplina de Modelos de Linguagens de Programação

Prof. Dr. Lucas Mello Schnorr
Orientador

Porto Alegre
2017

SUMÁRIO

1 INTRODUÇÃO

O presente trabalho consiste na implementação de um problema em dois paradigmas diferentes: funcional e orientado a objeto. A linguagem a ser utilizada deve ser a mesma para ambos os paradigmas.

O objetivo deste trabalho consiste em fornecer aos alunos a oportunidade de estudar uma linguagem de programação moderna com características híbridas (i.e., multiparadigma). O trabalho permitirá aos alunos demonstrarem que aprenderam os princípios de programação relacionados com os diferentes paradigmas estudados ao longo do semestre, demonstrando, ainda, a capacidade de analisar e avaliar linguagens de programação, seguindo os critérios abordados em aula.

As implementações, tanto funcional quanto orientada a objeto, e relatório estão disponíveis em <https://github.com/lbassis/galaxias>.

Linguagem Escolhida: R

R é um projeto GNU voltado a programação estatística e gráfica. Faz parte dele uma linguagem multiparadigma (vetorial, orientada a objetos, imperativa, funcional, procedural e reflexiva) com tipagem dinâmica, fundamentada por S e inspirada por Scheme.

A linguagem fornece uma grande variedade de ferramentas para estatística, como modelagem linear e não-linear, classificação, clustering, testes estatísticos. Um dos seus pontos fortes é a facilidade com que gráficos bem construídos e prontos para uso em publicações científicas podem ser gerados. Esses gráficos incluem formulas e símbolos matemáticos onde necessário.

Resumindo, R é um conjunto integrado de métodos para manipulação de dados, cálculos e display gráfico que inclui (THE R FOUNDATION, 2017):

- Uma série de operações sobre vetores, em particular matrizes;
- Uma extensa coleção de ferramentas intermediárias para análise de dados;
- Ferramentas de plot para análise de dados.
- Uma linguagem de programação bem desenvolvida, simples e eficaz que inclui condicionais, loops, funções recursivas definidas pelo usuário e funções de entrada e saída.

Problema escolhido: Galáxias

O problema consiste em implementar um simulador de partículas, considerando forças físicas de repulsão e atração. Serão utilizadas leis gravitacionais para simular órbitas de estrelas e planetas. Além disso, uma interface será criada para que seja possível acompanhar o movimento simulado.

Para suportar a simulação de muitos corpos, será utilizado o algoritmo de Barnes-Hut (BARNES; HUT, 1986), que consiste em subdividir o espaço em quadrantes (no caso de um espaço bi-dimensional) recursivamente até atingir um dado limite de modo a simplificar o problema sem perdas consideráveis de precisão no modelo.

Problema de N-corpos

$$\theta = \frac{d}{r} \tag{1.1}$$

Uma determinada região do espaço contém N corpos, cada um com suas próprias propriedades como sua massa e com um campo de forças gravitacionais resultante agindo sobre si. A massa de cada corpo é uma característica física, e o campo resultante pode ser calculado a partir das posições dos corpos e de suas massas. Segundo a primeira lei de Newton, os corpos sofrem uma aceleração devido ao campo de forças gravitacionais resultante. Logo, a aceleração de um corpo e a sua posição dependem do campo de força resultante da interação entre todos os corpos.

Numa simulação desse problema, parte-se de um estado s em que as posições e as velocidades dos corpos são conhecidas. A partir de s, calculam-se as forças entre todos os corpos seguindo a equação 1.2, onde G é igual à constante gravitacional universal, m_1 é a massa do corpo 1, m_2 é a massa do corpo 2 e r é a distância entre os dois corpos. Visto que é necessário calcular o campo de forças resultante para cada um dos corpos, $\frac{1}{2}N*(N-1)$ forças são calculadas. Logo, a complexidade de calcular todas as forças é $O(N^2)$.

$$F = G\frac{m_1.m_2}{r^2} \tag{1.2}$$

Uma vez que as forças foram calculadas, as acelerações das partículas são obtidas seguindo a equação 1.3. Para o cálculo das novas posições e velocidades das partículas, assume-se um tempo Δt entre cada etapa da simulação. Também se asssume que Δt

é suficientemente pequeno para que se possa fazer a aproximação que a aceleração é constante durante esse período de tempo. Utilizam-se, pois, as equações 1.4 e 1.5 do Movimento Uniformemente Variado (MUV) para atualizar as posições e as velocidades das partículas.

$$a = \frac{F}{m} \tag{1.3}$$

$$v_{prox} = v_{atual} + a * \Delta t \tag{1.4}$$

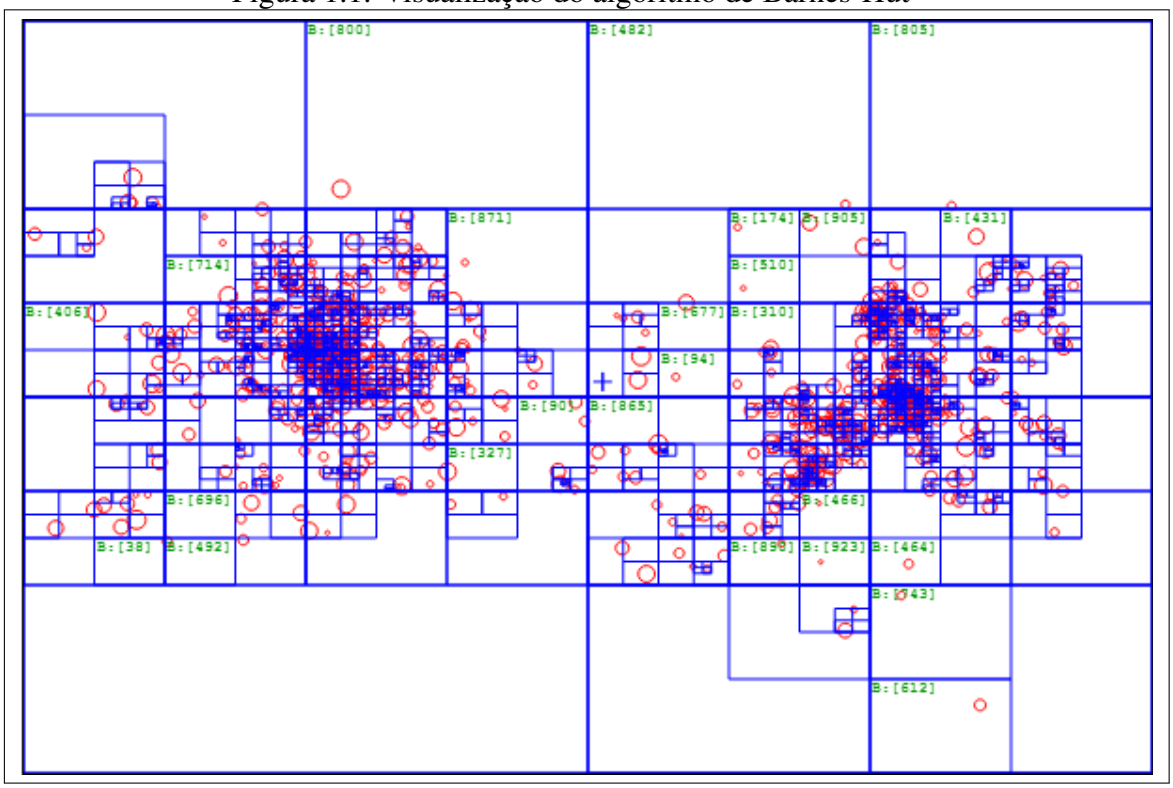
$$s_{prox} = s_{atual} + v_{atual} * \Delta t + \frac{a}{2} * (\Delta t)^2 \tag{1.5}$$

Algoritmo de Barnes-Hut

O algoritmo de Barnes-Hut resolve o problema de N-corpos numa complexidade inferior àquela apresentada na seção anterior. Em vez de calcular todas as forças, uma aproximação baseada em árvores que divide o espaço em quadrantes é utilizada. Esse algoritmo tem complexidade igual a $O(N log(N))$

O algoritmo de Barnes-Hut reduz o número de forças calculadas agrupando partículas. A aproximação utilizada é que um grupo de partículas distantes pode ser reduzido a uma única partícula de centro igual ao centro de massa do grupo. Dessa forma, só é necessário calcular as forças uma a uma para partículas muito próximas, enquanto que as distantes têm seu cálculo simplificado. A simplificação só é válida quando a distância da partícula ao grupo é grande em relação ao tamanho do quadrante do grupo. Para determinar se a razão entre o tamanho d do quadrante do grupo e a distância r é suficiente, utiliza-se o critério da equação (1.1). O algoritmo de Barnes-Hut aplica $\theta = 1$ subdividindo recursivamente o espaço em quadrantes até que apenas uma partícula por quadrante exista, como visto na Figura 1.1.

Figura 1.1: Visualização do algoritmo de Barnes-Hut



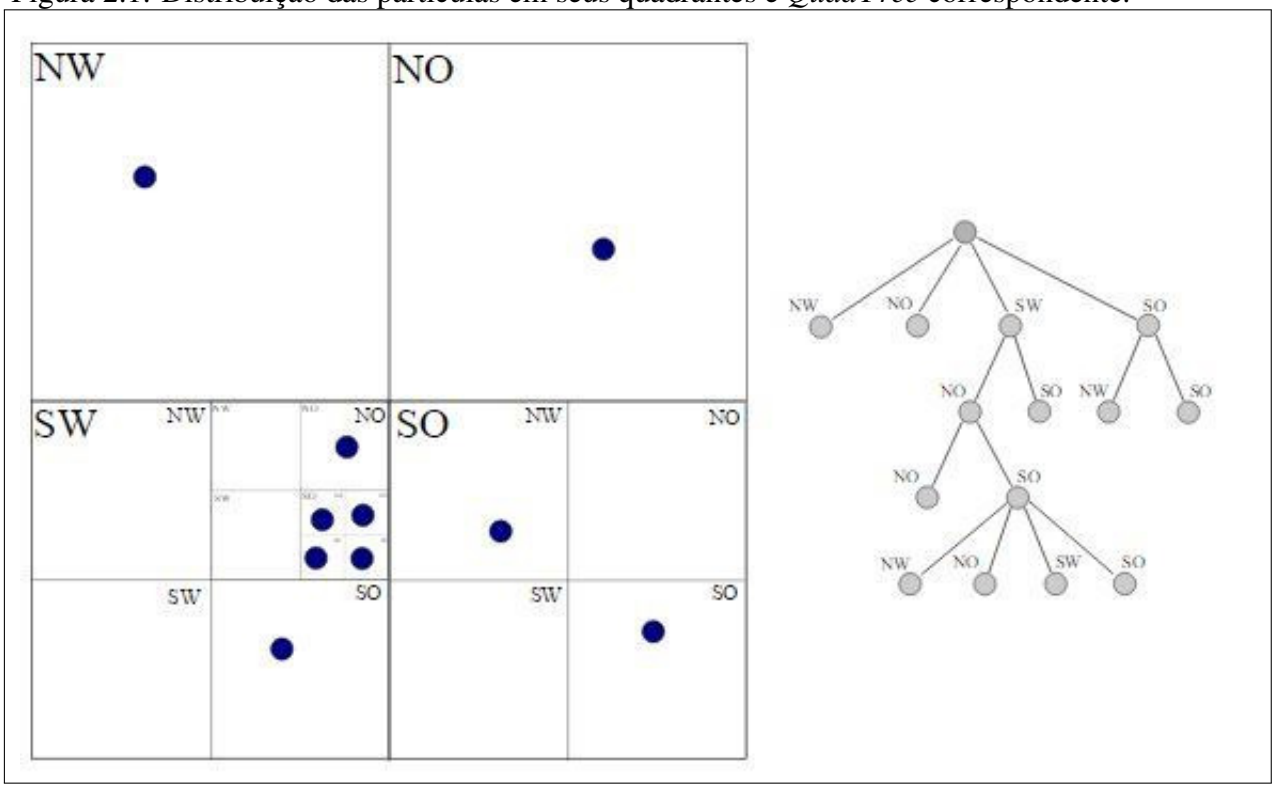
Fonte: (ANSARI, 2016)

2 IMPLEMENTAÇÃO

Implementação Física

O algoritmo recebe como entrada uma lista de partículas com suas posições e suas velocidades. Começa-se subdivindindo o espaço recursivamente em quadrantes até que haja no máximo uma partícula por quadrante. Ao mesmo tempo que os quadrantes são divididos, popula-se uma *QuadTree* hierarquicamente de modo que cada nodo folha seja uma partícula do espaço, como pode ser visto na Figura 2.1. Em um segundo momento, os centros de massa para cada quadrante são calculados. Após isso, calculam-se as forças resultantes para todas as partículas, reduzindo o número de forças calculadas quando um quadrante passa na regra da equação (1.1). Finalmente, as posições e velocidades das partículas são atualizadas considerando um passo de atualização que corresponde ao tempo percorrido entre uma etapa da simulação e a seguinte. A implementação em pseudocódigo pode ser vista no algoritmo 1.

Figura 2.1: Distribuição das partículas em seus quadrantes e *QuadTree* correspondente.



Fonte: (BERG, 2017)

Algorithm 1: Algoritmo de Barnes-Hut para simulação de N-corpos

```
1 function barnesHut (particles, positions, velocities, masses);
  Input  : Uma lista de partículas, suas posições, velocidades e massas
  Output: As posições e velocidades das partículas atualizadas
2 quadTree = createQuadTree(particles, positions, velocities, masses);
3 quadTree = computeMassDistribution(quadTree);
4 quadTree = computeResultantForces(quadTree);
5 newVelocities = updateVelocities(quadTree);
6 newPositions = updatePositions(quadTree);
7 return [newVelocities, newPositions];
```

Implementação no Paradigma Funcional

Visualização Gráfica

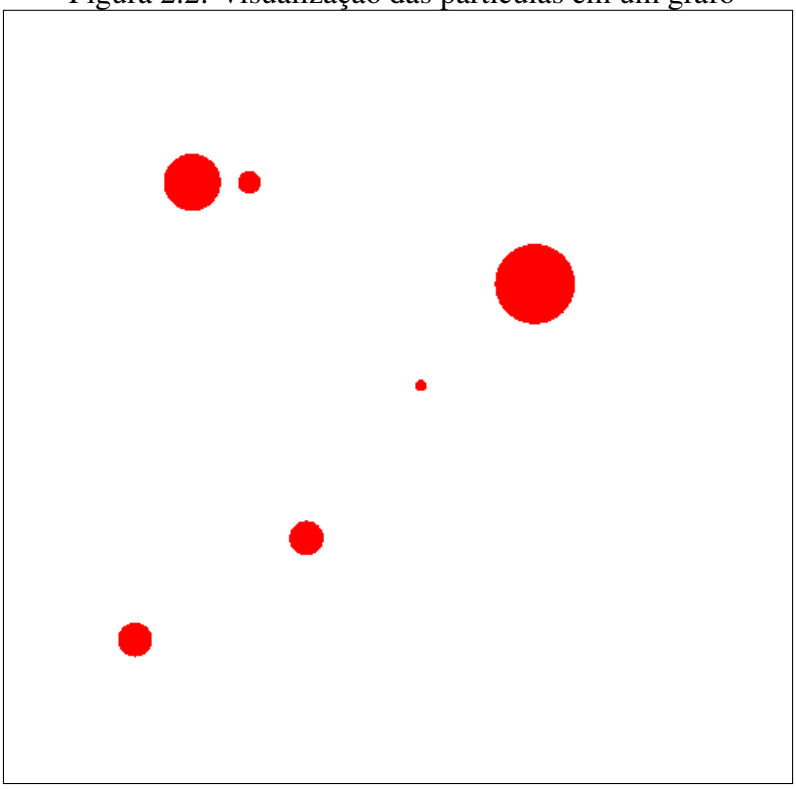
A visualização gráfica foi implementada utilizando o pacote *plotrix* e a biblioteca *ImageMagick*. Em cada iteração do algoritmo, isto é, em cada conjunto de posições calculado, gera-se um gráfico com círculos representando as partículas em um espaço bidimensional, conforme é mostrado na figura 2.2. Em uma abordagem inicial, a ideia foi utilizar diversos gráficos carregados em sequência no ambiente *Rstudio*, com um pequeno intervalo de espera entre cada geração. No entanto, a performance não foi satisfatória; apesar de, inicialmente e com um número pequeno de partículas, a animação ficasse fluída, com o passar do tempo o tempo entre as gerações é cada vez maior, tornando o algoritmo muito lento em pouco tempo.

A partir daí, a ideia escolhida foi a de selecionar inicialmente um número de iterações; depois disso, são geradas todas as imagens necessárias para representar esse número e, por fim, as imagens são transformadas em um arquivo *.gif*. Dessa maneira, o algoritmo perde a capacidade de ser demonstrado em tempo real, já que a duração da simulação é definida antes da execução da mesma. No entanto, dessa maneira foi possível obter uma animação adequada ao esperado e, portanto, essa foi a solução escolhida.

A geração das imagens implementa um laço utilizando recursão. Enquanto um número de iterações (passado por parâmetro para a função) não for nulo, a recursão é chamada, reduzindo em uma unidade o número de iterações. Dentro do laço, uma partícula é desenhada utilizando a função *draw_circle*, implementadas conforme mostrado no algoritmo 2. Depois disso, uma função *lapply* mapeia esse algoritmo para que ele seja aplicado a todas as partículas de uma lista, desenhando todos os elementos. Ao final da

execução do laço, uma funçao da biblioteca *ImageMagick* transforma as imagens geradas em um arquivo *.gif* animado.

Figura 2.2: Visualização das partículas em um grafo



Algorithm 2: Algoritmo que desenha uma partícula

```
1 function draw_particle (particle);
  Input: Uma partícula encapsulada em uma estrutura de nodo
2 draw_circle(qnode_x(node), qnode_y(node), qnode_mass(node), col =
    "red", nv = 1000, border = NA, lty = 1, lwd = 1);
```

Aspectos funcionais

Segue abaixo a descrição e demonstração de algumas técnicas tipicamente funcionais utilizadas na elaboração deste trabalho.

Elementos imutáveis

A utilização de elementos imutáveis é recorrente na solução desenvolvida. Como exemplo, pode-se citar a função de atualização da posição de uma partícula. A cada itera-

ção do laço, uma nova partícula é criada (e, consequentemente, uma nova lista de partículas) para que esta ocupe a nova posição calculada com a velocidade também atualizada. A função que cria a nova partícula é mostrada no algoritmo 3.

Algorithm 3: Algoritmo de atualização de posição

```
1 particle_setPositionAndVelocity (particle, point, velocity);
  Input  : Uma partícula e dois novos pontos representando sua nova posição e
           nova velocidade
  Output: Uma nova partícula criada com os parâmetros atualizados
2 particle_setPositionAndVelocity <- function(particle, point, velocity);
3 new_particle(point, qnode_mass(particle), velocity,;
4 new_point(qnode_fx(particle), qnode_fy(particle)), qnode_size(particle));
```

Funções não nomeadas

Funções não nomeadas são utilizadas frequentemente para que elas possam ser passadas como se fossem um elemento de primeira ordem da implementação. Neste trabalho, por exemplo, foram utilizadas para definir a função da velocidade em função do tempo e da distância, demonstrada no algoritmo 4 que é aplicada posteriormente nas componentes horizontal e vertical da velocidade de uma partícula.

Algorithm 4: Trecho do algoritmo do cálculo da nova posição

```
1 updateVelocityAndPosition (node, deltaT);
  Input  : Uma partícula encapsulada em um nodo e um intervalo de tempo a
           ser considerado
  Output: Uma nova partícula criada com os parâmetros atualizados
2 calcVelocityComponent <- function(v,a,t)(v + a * t);
3 newVx <- calcVelocityComponent(vx, ax, deltaT);
4 newVy <- calcVelocityComponent(vy, ay, deltaT);
```

Currying

O *Currying* é uma técnica que transforma uma função de duas entradas em uma função com uma entrada que retorna outra função de uma entrada. Nesse trabalho, essa implementação foi utilizada para acessar as colunas dos *data.frames* criados - os *qnodes*. O algoritmo 5 mostra a implementação da função "original".

Algorithm 5: Algoritmo seletor de colunas do dataframe node

```
1 qnode_data (data, node);
  Input  : A descrição de uma coluna do dataframe e uma partícula
           encapsulada em um nodo
  Output: O valor associado à coluna selecionada no nodo fornecido
2 if(is.data.frame(node));
3 node[data][[1]];
4 elseif(!qnode_empty(node));
5 node[[1]][data][[1]];
6 else0;
```

A partir daí, torna-se fácil construir as funções específicas de cada seletor, que utilizam como argumento apenas o *qnode* a ser acessado. Utilizando a função *Curry* do pacote *functional*, temos a seguinte implementação para obter o parâmetro *point_x*, por exemplo.

Algorithm 6: Algoritmo seletor da coluna x do dataframe node

```
1 qnode_x (node);
  Input  : Uma partícula encapsulada em um nodo
  Output: O valor da coluna x no nodo fornecido
2 Curry(qnode_data, data = "point_x");
```

Funções de ordem superior próprias

Funções de ordem superior aceitam que outras funções sejam passadas como parâmetro. Como exemplo, temos a implementação da funçao *best_point*, conforme visto na função 7. Para determinar os limites do gráfico e o centro geométrico da simulação, são computados o ponto mais à esquerda/mais acima e a maior distãncia em uma das dimensões até outro ponto pertencente à simulação. Tendo a função de alta ordem *best_point* e usando as funções primitivas min e max, é possível atualizar os limites *top_left* e *bottom_right* (este usado para atualizar a maior distância entre os pontos) recursivamente.

Algorithm 7: Algoritmo que encontra os extremos dos quadrantes

```
1 best_point (p, q, fun);
  Input  : Um ponto, um quadrante e uma função para determinar o ponto
           mais apropriado
  Output: O ponto mais extremo do quadrante
2 new_point(fun(q["x"][[1]], p["x"][[1]]), fun(q["y"][[1]], p["y"][[1]]));;
```

Funções de ordem superior prontas

Um dos exemplos a ser citado de funções de ordem superior já implementadas pela linguagem é a função *lapply*. Ela funciona como um funtor, mapeando uma função que recebe um elemento e generaliza-na para que ela receba uma lista desses elementos, aplicando a função inicial a todos eles. No algoritmo 2, por exemplo, desenha-se uma partícula. A partir dele, pode-se construir o algoritmo 8, generalizando sua implementação para trabalhar com uma lista de partículas em vez de um único elemento.

Algorithm 8: Algoritmo que desenha todas as partículas do instante atual

```
1 draw_particles (particles);
  Input: Uma lista de partículas encapsulada em nodos
2 invisible(lapply(particles, draw_particle));
```

Funções como elemento de primeira ordem

Em linguagens funcionais, é possível que funções sejam passadas como parâmetros de outras funções, funcionando como elementos de primeira ordem. Essa técnica foi utilizada no *loop* que desenha as partículas para indicar qual a função que atualizaria suas posições, conforme mostrado no algoritmo 9.

Algorithm 9: Laço que desenha os quadros da animação

```
1  drawing_ loop (particles, iteractions, name, updatePosAndVel);
   Input: Uma lista de partículas encapsulada em nodos, o número de iterações
          do laço, o nome dos arquivos temporários e uma função que devolve
          uma lista de nodes
2  if(iteractions == 0);
3  system("convert -delay 10 *.jpg result.gif");
4  file.remove(list.files(pattern = ".jpg"));
5  else;
6  name <- paste(name,"a");
7  filename <- paste(name,".jpg");
8  png(filename = filename);
9  plot.new();
10 frame();
11 draw_particles(normalize_masses(particles));
12 dev.off();
13 particles <- updatePosAndVel(qList_toParticles(particles));
14 drawing_loop(particles, iteractions - 1, name, updatePosAndVel);
```

Funções com pattern matching

Pattern matching é uma técnica que procura padrões dentro de sequências de valores, muito utilizada, por exemplo, em processamento de linguagens. Como esse trabalho é uma implementação que simula fenômenos físicos, o grupo não encontrou uma aplicação interessante para esse tipo de algoritmo, portanto não há nenhum exemplo a ser mostrado.

Recursão

A recursão foi utilizada frequentemente como iterador para que todos os elementos de uma lista fossem devidamente processados. Um possível exemplo é o algoritmo 9 mostrado na sessão anterior. Ao final de cada iteração, diminui-se em uma unidade o número de iterações restantes e a função *drawing_loop* é chamada novamente.

Análise do algoritmo

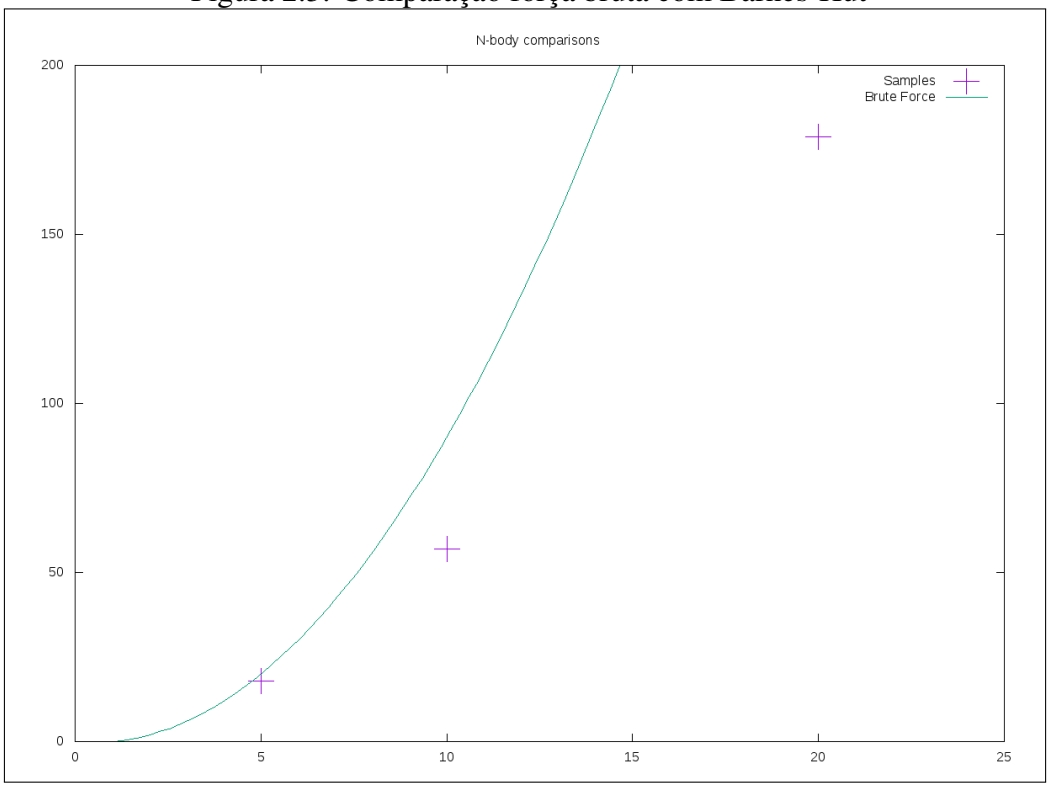
O grupo teve dificuldade em encontrar valores de variáveis adequados para realizar as simulações. Portanto, não foi gerado nenhum caso de teste grande o suficiente para comprovar a eficiência do algoritmo em relação à uma solução em força bruta. Por isso, foi elaborado o gráfico 2.3 a partir do caso de teste elaborado, comparando a quantidade de comparações do algoritmo de força bruta com a solução implementada. O eixo X representa o número de partículas, enquanto o eixo Y representa o número de comparações.

Análise da linguagem

A seguir são atribuídas notas às diferentes características da linguagem R quando considerando uma implementação usando o paradigma funcional. As notas foram dadas em um intervalo [0,5].

A seguir, temos as análises que levaram a cada uma das notas mostradas na tabela.

Figura 2.3: Comparação força bruta com Barnes-Hut



Simplicidade

O operador <- vai contra a simplicidade da linguagem, já que ele implementa a atribuição, assim como o operador =.

Ortogonalidade

Diversas funções do programa alternam entre data.frames e listas, e a linguagem oferece suporte a isso naturalmente.

Expressividade

A expressividade é adequada no geral, porém no acesso à listas, a necessidade de usar dois colchetes para recuperar um elemento torna o código menos expressivo do que ele poderia ser. Como alternativa, temos as funções *head* e *tail*, porém as duas também contam com um fator incomum que é a necessidade de especificar quantos elementos serão recuperados da lista.

Tabela 2.1: Características da linguagem R

Característica	*Nota [0, 5]*
Simplicidade	3.5
Ortogonalidade	4.5
Expressividade	3.0
Adequabilidade e variedade de estruturas de controle	4.0
Mecanismos de definição de tipos	4.0
Suporte a abstração de dados	3.0
Suporte a abstração de processos	4.5
Modelo de tipos	3.0
Portabilidade	3.0
Reusabilidade	4.5
Suporte e documentação	3.0
Tamanho de código	4.0
Generalidade	3.0
Eficiência	4.0
Custo	2.5

Fonte: Autor

Adequabilidade e variedade de estruturas de controle

As estruturas de controle existem, mas a implementação funcional utiliza recursão em seu lugar. Apesar disso, alguns laços iterativos foram úteis para depuração do código.

Mecanismos de definição de tipos

O tipo data.frame, utilizado para representar tabelas, funciona como uma definição de estrutura e atendeu bem ao desejado no trabalho.

Suporte a abstração de dados

A linguagem fornece a estrutura data.frame, que equivale a uma tabela em que cada coluna pode ter um valor de diferentes tipos. No entanto, não é criada uma estrutura propriamente dita, com sua própria semântica.

Suporte a abstração de processos

Implementa funções recursivas e inclusive funções como elementos de primeira ordem, tornando o suporte a abstração de processos excelente.

Modelo de tipos

Sendo uma linguagem dinamicamente tipada, o modelo de tipos deixa a desejar. Em alguns casos, houveram problemas de difícil identificação que, com um sistema de tipos mais robusto, seriam facilmente detectáveis.

Portabilidade

A linguagem oferece diversos pacotes, mas todos os utilizados precisaram ser instalados posteriormente.

Reusabilidade

A utilização de funções como elementos de primeira ordem faz com que a linguagem possua uma ótima reusabilidade.

Suporte e documentação

Especialmente para este trabalho, que não utiliza a linguagem em um contexto em que ela se destaca, foi difícil encontrar referências para várias funcionalidades. No entanto, isso talvez esteja mais relacionado com a natureza da aplicação desenvolvida do que com a linguagem em si.

Tamanho de código

Nada especialmente diferente do comum. A linguagem não tem o código muito longo nem notoriamente curto.

Generalidade

Apesar de ter sido usada satisfatoriamente nesse trabalho, a linguagem fornece muito mais recursos para algoritmos estatísticos e similares do que qualquer outra implementação.

Eficiência

Para que essa característica seja analisada apropriadamente, seria necessário implementar o mesmo algoritmo em outro paradigma/linguagem, portanto fica difícil avaliar

sem ao menos concluir a implementação orientada a objeto. No entanto, pode-se observar que a maior parte do tempo que o algoritmo leva é na criação dos arquivos de imagem, indicando que as funções próprias são eficientes.

Custo

A curva de aprendizado é relativamente íngreme; além disso, o Rstudio eventualmente apresenta problemas em sua instalação e na instalação de seus pacotes. Por fim, para uma simulação física como a aqui apresentada, as estruturas nativas da linguagem e do ambiente não facilitam, portanto conclui-se que nesse caso o custo é alto.

3 CONCLUSÃO

Ao final deste trabalho, foi possível formar uma opinião sobre várias das construções da programação funcional que o grupo nunca havia implementado. No entanto, se tratando de uma simulação física, também foi confirmado que R não é a linguagem mais adequada para construir esse tipo de programa. O algoritmo implementado demonstra uma possibilidade de simular diversas partículas com uma complexidade reduzida. Contudo, devido à dificuldade para gerar casos de teste, só foram criadas simulações com um número pequeno de partículas, não aproveitando o potencial total do algoritmo. Por esse motivo, o grupo optou por elaborar o gráfico que demonstra a eficiência do algoritmo em relação à uma implementação em força bruta.

Houve algumas dificuldades iniciais para definir as estruturas da implementação, já que foi necessário encapsular as partículas juntamente com os subquadrantes.

Por fim, um problema que foi detectado no final da implementação foi que o grupo não implementou colisões. Dessa maneira, assim que duas partículas ficam muito próximas uma da outra, as suas acelerações crescem até que elas sejam "arremessadas"com uma velocidade irreal. No entanto, até a conclusão da implementação orientada a objeto, acredita-se que esse problema terá sido tratado.

REFERÊNCIAS

ANSARI, S. **Barnes-Hut N-body Simulation in HTML/Javascript**. 2016. Available from Internet: <https://github.com/Elucidation/Barnes-Hut-Tree-N-body-Implementation-in-HTML-Js>.

BARNES, J.; HUT, P. A hierarchical o(n log n) force-calculation algorithm. **Nature**, Nature Publishing Group SN -, v. 324, p. 446 EP –, Dec 1986. Available from Internet: <http://dx.doi.org/10.1038/324446a0>.

BERG, I. **The Barnes-Hut Galaxy Simulator**. 2017. Available from Internet: <http://beltoforion.de/article.php?a=barnes-hut-galaxy-simulator>.

THE R FOUNDATION. **R: What is R?** 2017. Available from Internet: <https://www.r-project.org/about.html>.