The Barnes Hut Algorithm: Implementation and Simulation in Julia

20602 - Computer Science (Algorithms)

Pietro Dominietto

Bocconi University

20th October 2021

Table of Contents

The N-body problem

The setting:

- Delimited region of space in $\mathbb{R}^2$

The N-body problem

The setting:

- Delimited region of space in $\mathbb{R}^2$
- N points defined by:
  - A vector for the position in space

$$(x, y)$$

  - A vector for the velocity of the point

$$(v_x, v_y)$$

  - a value for the mass

$$m$$

The N-body problem

The setting:

- Delimited region of space in $\mathbb{R}^2$
- N points defined by:
  - A vector for the position in space

$$(x, y)$$

  - A vector for the velocity of the point

$$(v_x, v_y)$$

  - a value for the mass

$$m$$

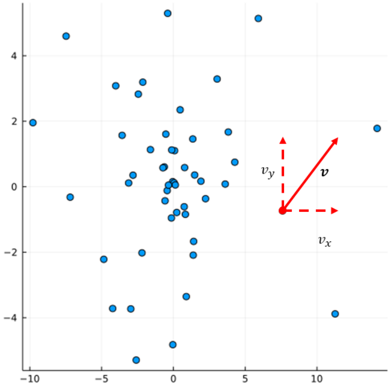
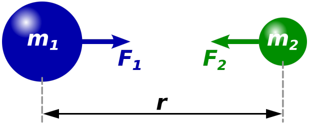


Figure: A cloud of 50 bodies

The N-body problem

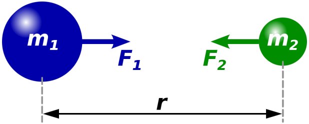
The points in space are subject to a force of attraction, which in this case is the **gravitational pull**, and it behaves according to Newton's law:



$$F_1 = F_2 = G\,\frac{m_1 \times m_2}{r^2}$$

The N-body problem

The points in space are subject to a force of attraction, which in this case is the **gravitational pull**, and it behaves according to Newton's law:



$$F_1 = F_2 = G \frac{m_1 \text{ x } m_2}{r^2}$$

Then, the point is moved accrding to Newton's second law of mechanics:

$$\vec{a}_i = \frac{\vec{F}_i^{net}}{m_i}$$

Brute Force Approach

Therefore, to compute the net force acting on a body one needs to compute all the pairwise gravitational interactions between points.

$$\vec{F}_i^{net} = G \sum_{j \neq i} \frac{m_i m_j}{d_{i,j}^2}$$

Brute Force Approach

Therefore, to compute the net force acting on a body one needs to compute all the pairwise gravitational interactions between points.

$$\vec{F}_i^{net} = G \sum_{j \neq i} \frac{m_i m_j}{d_{i,j}^2}$$

Since forces are symmetrical between two bodies, the total number of computations required would be:

$$\frac{N(N-1)}{2} = O(N^2)$$

where N is the number of points in the space.

Brute Force Approach

Therefore, to compute the net force acting on a body one needs to compute all the pairwise gravitational interactions between points.

$$\vec{F}_i^{net} = G \sum_{j \neq i} \frac{m_i m_j}{d_{i,j}^2}$$

Since forces are symmetrical between two bodies, the total number of computations required would be:

$$\frac{N(N-1)}{2} = O(N^2)$$

where N is the number of points in the space.

- Such time complexity is feasible only for small N
- For example: Earth–Sun simulation, Solar System ($N \sim 20$)

```
function onestepBrute(time::Float64,stars::Array{Star,1},spaceScale::Int64)
    new_stars = copy(stars)
    # prepare a NxNx2 tensor for the interactions
    F_mat = zeros(length(stars),length(stars),2)
    for i in 1:length(stars)                                  O(N)
        # compute the net force
        for j in i:length(stars)                              O(N)
            # compute force for all pairs not already computed
            if i != j
                F = newton(stars[j],stars[i])
                d_j = stars[j]-stars[i]
                cos_0j, sin_0j = get_cos_sin(d_j)             O(1)
                f_j = [F * cos_0j, F * sin_0j]                         O(N^2)
                F_mat[i,j,:] = f_j # * 10^9
                F_mat[j,i,:] = -f_j # * 10^9
            end
        end
        # move the body according to the net force
        net_force = [sum(F_mat[i,:,1]),sum(F_mat[i,:,2])]    O(N)
        new_stars[i] = moveStar(net_force,stars[i],time,spaceScale)
    end
    # return a list of all stars with updated pos & vel
    return new_stars
end
```

Figure: Naive brute force approach

Earth-Sun Simulation

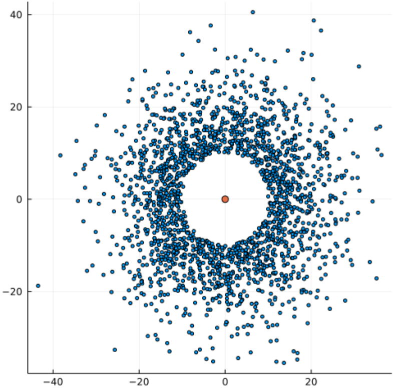
Why is there a `spaceScale`?

$$m_{sun} \sim \texttt{1e30}$$
$$m_{earth} \sim \texttt{1e24}$$
$$d_{e,s}^2 \sim (\texttt{1e11})^2$$
$$G \sim \texttt{1e-11}$$

$$\Rightarrow F_{e,s} \sim \texttt{1e21}$$

Divide the masses by `1e20` and the distance by `1e10`, so that

$$F_{e,s}^{scaled} \sim 10^1$$



Earth Sun Simulation

MAIN IDEA:

Approximate the net force on a body in a *clever* way.

When a point is **"far enough"** from a cluster, the gravitational field generated by the cloud of points is *roughly* the same as if it were generated by a **single point** having as mass the sum of the masses and located in the **center of mass**[1].

[1]The center of mass is the average of the coordinates weighted by the mass of the points, $(x_{CM}, y_{CM}) = (\sum m_i)^{-1}(\sum m_i x_i; \sum m_i y_i)$

The Barnes Hut Algorithm

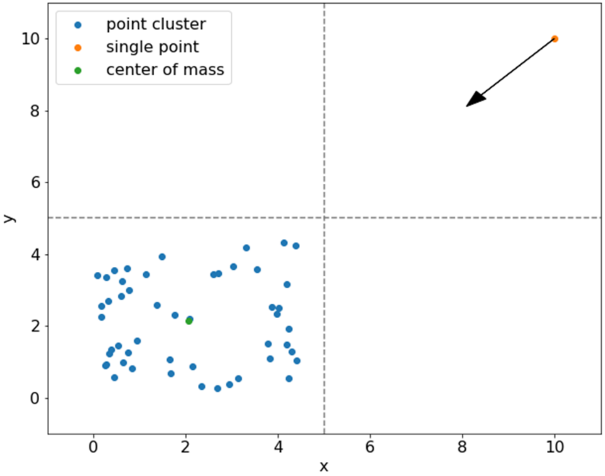


Figure: Approximating the net force with Barnes Hut

The Barnes Hut Algorithm

Problems:

1. How do I define a cluster of points?
2. What classifies as **"far enough"**?

Problem 1: *how to define a cluster of points*

- find a way to characterize the space of particles and organize their positions;
- optimize the necessary retrieval operations and computations.

The Barnes Hut Algorithm

Problem 1: *how to define a cluster of points*

- find a way to characterize the space of particles and organize their positions;
- optimize the necessary retrieval operations and computations.

IDEA: divide the space using a tree structure called **Quadtree** (or Octree in 3D)

The Quadtree

A **Quadtree** is a "map" of space that helps us model groups of points as a single center of mass. Formally, it is a tree structure with the following properties:

- Every node has exactly 4 children

The Quadtree

A **Quadtree** is a “map” of space that helps us model groups of points as a single center of mass. Formally, it is a tree structure with the following properties:

- Every node has exactly 4 children
- Each node of the tree corresponds to a region of space

The Quadtree

A **Quadtree** is a “map” of space that helps us model groups of points as a single center of mass. Formally, it is a tree structure with the following properties:

- Every node has exactly 4 children
- Each node of the tree corresponds to a region of space
- Its children correspond to the 4 quadrants that are obtained by dividing that region with the orthogonal axes

The Quadtree

A **Quadtree** is a "map" of space that helps us model groups of points as a single center of mass. Formally, it is a tree structure with the following properties:

- Every node has exactly 4 children
- Each node of the tree corresponds to a region of space
- Its children correspond to the 4 quadrants that are obtained by dividing that region with the orthogonal axes
- The root node contains all the points and represents all the space.

The Quadtree

A **Quadtree** is a “map” of space that helps us model groups of points as a single center of mass. Formally, it is a tree structure with the following properties:

- Every node has exactly 4 children
- Each node of the tree corresponds to a region of space
- Its children correspond to the 4 quadrants that are obtained by dividing that region with the orthogonal axes
- The root node contains all the points and represents all the space.
- Each child node is recursively defined in the same way.

The Quadtree

An example:[2]

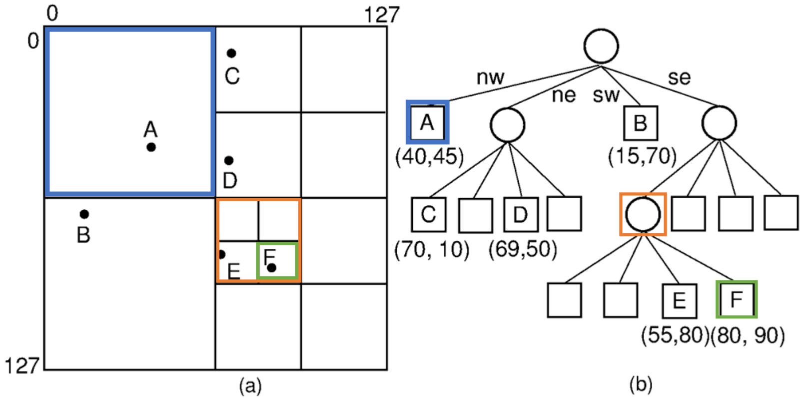


Figure: To the left the space carved by the tree, and to the right the tree structure of the points in the space

[2]source: website

The Quadtree

How to construct the tree:

1. We start at the root of the tree, which represents the entire space, but it is empty at the beginning.

The Quadtree

How to construct the tree:

1. We start at the root of the tree, which represents the entire space, but it is empty at the beginning.
2. For every point in the space, we do the following:
    1. Get the quadrant where the point is, and check if the corresponding child is empty.

The Quadtree

How to construct the tree:

1. We start at the root of the tree, which represents the entire space, but it is empty at the beginning.
2. For every point in the space, we do the following:
    1. Get the quadrant where the point is, and check if the corresponding child is empty.
    2. If yes, add the point and continue.

The Quadtree

How to construct the tree:

1. We start at the root of the tree, which represents the entire space, but it is empty at the beginning.
2. For every point in the space, we do the following:
    1. Get the quadrant where the point is, and check if the corresponding child is empty.
    2. If yes, add the point and continue.
    3. If no, add the point to the child node. Then repeat the same process (from (2)), considering that child as root and as points the ones added to the quadrant.

The Quadtree

How to construct the tree:

1. We start at the root of the tree, which represents the entire space, but it is empty at the beginning.
2. For every point in the space, we do the following:
    1. Get the quadrant where the point is, and check if the corresponding child is empty.
    2. If yes, add the point and continue.
    3. If no, add the point to the child node. Then repeat the same process (from (2)), considering that child as root and as points the ones added to the quadrant.
3. At the end, return the root.

Quadtree Data Structure

Once we constructed the tree, we will only need to traverse it from the root to the leaves, in order to compute the force acting on a point.

Quadtree Data Structure

Once we constructed the tree, we will only need to traverse it from the root to the leaves, in order to compute the force acting on a point.
Hence, we can store the quadtree through Node2D objects which contain four pointers to the children nodes and also information about the quadrant.

```
mutable struct Node2D
    parent::Union{Nothing,Node2D}
    children::Dict{String,Union{Nothing,Node2D}}
    stars::Array{Star,1}
    regCenter::Array{Float64,1}
    x_lim::Array{Float64,1}
    y_lim::Array{Float64,1}
    n::Int64
    centerOfMass::Array{Float64,1}
    totalMass::Float64
    Node2D(parent,children,stars,regCenter,x_lim,
            y_lim,n,centerOfMass,totalMass) =
        new(parent,children,stars,regCenter,x_lim,
            y_lim,n,centerOfMass,totalMass)
end
```

Quadtree Data Structure

Therefore, it is enough to keep track of the root of the tree, and from there we can explore all the other nodes.

This data structure has the following operation times:

	Insert	Get
Average	$O(logN)$	$O(logN)$
Worst	$O(N)$	$O(N)$

Table: Time complexity of operations

As for space complexity, the number of nodes in the tree is

$$\frac{4N-1}{3} = O(N)$$

Build Quadtree in Julia

```julia
function buildQTree(root::Union{Nothing,Node2D},stars::Union{Nothing,Array{Star,1}},
                    x_lim::Array{Float64,1},y_lim::Array{Float64,1})
    if isnothing(root)
        root = createRoot(stars,x_lim,y_lim)
    end
    # Loop on every star
    for s in stars
        Q = getQuadrant(root.regCenter,s.s)
        if isnothing(root.children[Q])
            # If the quadrant is still empty, create the node
            center, x_lim, y_lim = getCenterLimits(root,Q)
            root.children[Q] = Node2D(root,center,x_lim,y_lim)
            root.children[Q] = updateNode!(root.children[Q],s)
        else
            # Otherwise add the star to the node
            root.children[Q] = updateNode!(root.children[Q],s)
            if root.children[Q].n == 2
                # If there was already a star, then we need to go down a level and create other nodes
                stars_ = root.children[Q].stars
            else
                # otherwise we just need to sort the new star
                stars_ = [s]
            end
            # Then recursively call the constructor for that node and its corresponding quadrant
            root.children[Q] = buildQTree(root.children[Q], stars_,
                                          root.children[Q].x_lim, root.children[Q].y_lim)
        end
    end
    # At the end it is enough to return the root, as it contains pointers to all its children
    return root
end
```

O(1)

O(N)

O(1)

O(1)

O(1)

O(logN)

$O(N \log N)$

The Multipole Acceptance Criterion

Problem 2: *What classifies as "far enough"?*

Once we have a way to divide the space into regions and the points into clusters, we need a rule to define what can be considered **"far enough"**.

The Multipole Acceptance Criterion

Problem 2: *What classifies as "far enough"?*

Once we have a way to divide the space into regions and the points into clusters, we need a rule to define what can be considered **"far enough"**.

The Multipole Acceptance Criterion (MAC)

If the size of the cluster s divided by the distance of the cluster's CoM d from the point is lower than a parameter $\theta > 0$, then consider the cluster as a whole, i.e. approximate if:

$$\theta > \frac{s}{d}$$

The Multipole Acceptance Criterion

MAC provides a parameter θ for the precision of the BH algorithm:

- If $\theta = 0$, then BH will never approximate, and it will produce a result equivalent to brute force.

The Multipole Acceptance Criterion

MAC provides a parameter θ for the precision of the BH algorithm:

- If $\theta = 0$, then BH will never approximate, and it will produce a result equivalent to brute force.
- For lower values of θ the algorithm will be more precise but take longer to compute.

The Multipole Acceptance Criterion

MAC provides a parameter θ for the precision of the BH algorithm:

- If $\theta = 0$, then BH will never approximate, and it will produce a result equivalent to brute force.
- For lower values of θ the algorithm will be more precise but take longer to compute.
- If θ is large instead, we gradually give up precision in the result to gain speed in the computation.

The Multipole Acceptance Criterion

MAC provides a parameter θ for the precision of the BH algorithm:

- If $\theta = 0$, then BH will never approximate, and it will produce a result equivalent to brute force.
- For lower values of θ the algorithm will be more precise but take longer to compute.
- If θ is large instead, we gradually give up precision in the result to gain speed in the computation.

A usual good starting point is $\theta = 1$, which can be then adjusted according to the interest of the simulation.

Computing the forces with the Quadtree

Once the quadtree is constructed, we use it to compute the forces as follows.
For each point i (starting at the root node) do:

Computing the forces with the Quadtree

Once the quadtree is constructed, we use it to compute the forces as follows.
For each point i (starting at the root node) do:

1. Compute the node's size s and the distance $d(i, CoM)$ of i from the CoM;

Computing the forces with the Quadtree

Once the quadtree is constructed, we use it to compute the forces as follows.

For each point i (starting at the root node) do:

1. Compute the node's size s and the distance $d(i, CoM)$ of i from the CoM;
2. If the quadrant has only one point, compute the force between it and point i;

Computing the forces with the Quadtree

Once the quadtree is constructed, we use it to compute the forces as follows.
For each point i (starting at the root node) do:

1. Compute the node's size s and the distance $d(i, CoM)$ of i from the CoM;
2. If the quadrant has only one point, compute the force between it and point i;
3. (MAC) If $\theta > s/d$ then compute the force between the point i and the CoM (having as mass the sum of all points in the quadrant);

Computing the forces with the Quadtree

Once the quadtree is constructed, we use it to compute the forces as follows.
For each point i (starting at the root node) do:

1. Compute the node's size s and the distance $d(i, CoM)$ of i from the CoM;
2. If the quadrant has only one point, compute the force between it and point i;
3. (MAC) If $\theta > s/d$ then compute the force between the point i and the CoM (having as mass the sum of all points in the quadrant);
4. If not, repeat the same procedure for each non-empty children of the node considered.

Computing the forces with the Quadtree

Once the quadtree is constructed, we use it to compute the forces as follows.
For each point i (starting at the root node) do:

1. Compute the node's size s and the distance $d(i, CoM)$ of i from the CoM;
2. If the quadrant has only one point, compute the force between it and point i;
3. (MAC) If $\theta > s/d$ then compute the force between the point i and the CoM (having as mass the sum of all points in the quadrant);
4. If not, repeat the same procedure for each non-empty children of the node considered.

At the end of each iteration, we'll have $\vec{F}_i^{net}$ and we are able to move the point accordingly.

Computing the force with the Quadtree

```julia
function computeForceTree(node::Node2D,star::Star,theta::Float64)
    F_s = [0.,0.]
    s = abs(node.x_lim[2]-node.x_lim[1])
    d = distance(star.s,node.centerOfMass)
    if node.n == 1
        if star ∉ node.stars
            f = newton(star,node.stars[1])
            dir = node.stars[1]-star
            cosθ, sinθ = get_cos_sin(dir)
            F_s += [f * cosθ, f * sinθ] # * 10^9
        else
            F_s += [0.,0.]
        end
    elseif s/d < theta
        cm = Star(node.centerOfMass,[0.,0.],node.totalMass)
        f = newton(star,cm)
        dir = cm-star
        cosθ, sinθ = get_cos_sin(dir)
        F_s += [f * cosθ, f * sinθ] # * 10^9
    else
        for k in keys(node.children)
            if ! isnothing(node.children[k])
                F_s += computeForceTree(node.children[k],star,theta)
            end
        end
    end
    return F_s
end
```

O(1)

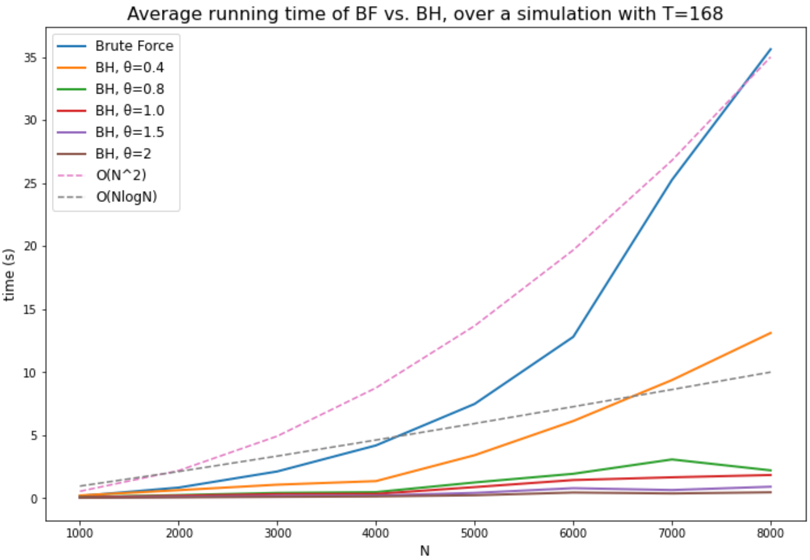
O(1)

O(logN)

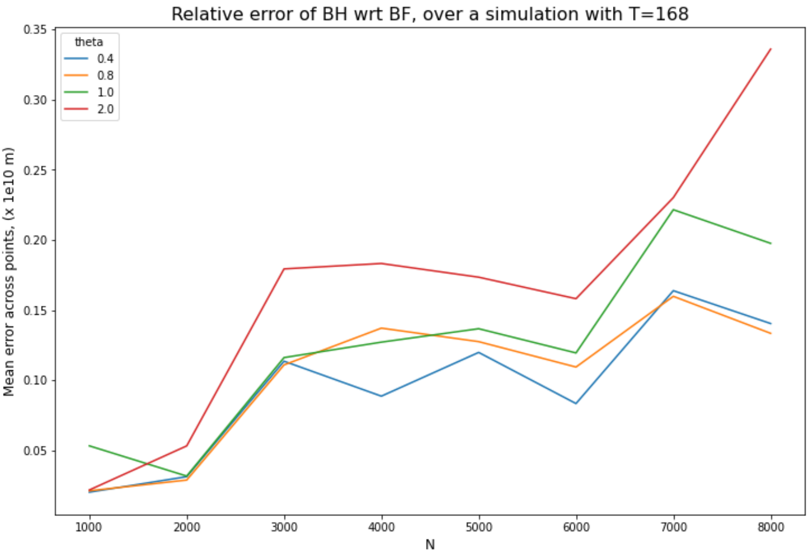
$O(\log N)$

[Galaxy clash simulation]

Simulations and Benchmarking



Simulations and Benchmarking



References

- Barnes, J., Hut, P. A hierarchical O(N log N) force-calculation algorithm. Nature 324, 446–449 (1986). doi.org/10.1038/324446a0
- en.wikipedia.org/wiki/Barnes–Hut-simulation
- jheer.github.io/barnes-hut
- arborjs.org/docs/barnes-hut
- beltoforion.de/en/barnes-hut-galaxy-simulator