CSE 548: Analysis of Algorithms

Lecture 4
(Divide-and-Conquer Algorithms: Polynomial Multiplication)

Rezaul A. Chowdhury
Department of Computer Science
SUNY Stony Brook
Spring 2019

Coefficient Representation of Polynomials

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

$$= a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

$A(x)$ is a polynomial of degree bound n represented as a vector $a = (a_0, a_1, \cdots, a_{n-1})$ of coefficients.

The *degree* of $A(x)$ is k provided it is the largest integer such that a_k is nonzero. Clearly, $0 \le k \le n - 1$.

Evaluating $A(x)$ at a given point:

Takes $\Theta(n)$ time using Horner's rule:

$$A(x_0) = a_0 + a_1 x_0 + a_2 (x_0)^2 + \cdots + a_{n-1} (x_0)^{n-1}$$

$$= a_0 + x_0 \Big(a_1 + x_0\big(a_2 + \cdots + x_0\big(a_{n-2} + x_0(a_{n-1})\big) \cdots \big)\Big)$$

Coefficient Representation of Polynomials

Adding Two Polynomials:

Adding two polynomials of degree bound n takes $\Theta(n)$ time.

$$C(x) = A(x) + B(x)$$

where, $A(x) = \sum_{j=0}^{n-1} a_j x^j$ and $B(x) = \sum_{j=0}^{n-1} b_j x^j$.

Then $C(x) = \sum_{j=0}^{n-1} c_j x^j$, where, $c_j = a_j + b_j$ for $0 \leq j \leq n-1$.

Coefficient Representation of Polynomials

Multiplying Two Polynomials:

The product of two polynomials of degree bound n is another polynomial of degree bound $2n - 1$.

$$C(x) = A(x)B(x)$$

where, $A(x) = \sum_{j=0}^{n-1} a_j x^j$ and $B(x) = \sum_{j=0}^{n-1} b_j x^j$.

Then $C(x) = \sum_{j=0}^{2n-2} c_j x^j$, where, $c_j = \sum_{k=0}^{j} a_k b_{j-k}$ for $0 \le j \le 2n-2$.

The coefficient vector $c = (c_0, c_1, \cdots, c_{2n-2})$, denoted by $c = a \otimes b$, is also called the *convolution* of vectors $a = (a_0, a_1, \cdots, a_{n-1})$ and $b = (b_0, b_1, \cdots, b_{n-1})$.

Clearly, straightforward evaluation of c takes $\Theta(n^2)$ time.

Convolution

$$a_0 + a_1 x + a_2 x^2 + a_3 x^3$$

$$b_3 x^3 + b_2 x^2 + b_1 x + b_0$$

$$a_0 b_0$$

Convolution

				a_0	+	$a_1 x$	+	$a_2 x^2$	+	$a_3 x^3$
$b_3 x^3$	+	$b_2 x^2$	+	$b_1 x$	+	b_0				
				$a_0 b_1 x$	+	$a_1 b_0 x$				

Convolution

$$a_0 + a_1x + a_2x^2 + a_3x^3$$

$$b_3x^3 + b_2x^2 + b_1x + b_0$$

$$a_0b_2x^2 + a_1b_1x^2 + a_2b_0x^2$$

Convolution

a_0	+	$a_1 x$	+	$a_2 x^2$	+	$a_3 x^3$
$b_3 x^3$	+	$b_2 x^2$	+	$b_1 x$	+	b_0
$a_0 b_3 x^3$	+	$a_1 b_2 x^3$	+	$a_2 b_1 x^3$	+	$a_3 b_0 x^3$

Convolution

a_0	+	$a_1 x$	+	$a_2 x^2$	+	$a_3 x^3$		
		$b_3 x^3$	+	$b_2 x^2$	+	$b_1 x$	+	b_0
		$a_1 b_3 x^4$	+	$a_2 b_2 x^4$	+	$a_3 b_1 x^4$		

Convolution

a_0 + $a_1 x$ + $a_2 x^2$ + $a_3 x^3$

$b_3 x^3$ + $b_2 x^2$ + $b_1 x$ + b_0

$a_2 b_3 x^5$ + $a_3 b_2 x^5$

Convolution

a_0 + a_1x + a_2x^2 + a_3x^3

b_3x^3 + b_2x^2 + b_1x + b_0

$a_3b_3x^6$

Coefficient Representation of Polynomials

Multiplying Two Polynomials:

We can use Karatsuba's algorithm (assume n to be a power of 2):

$$A(x) = \sum_{j=0}^{n-1} a_j x^j = \sum_{j=0}^{\frac{n}{2}-1} a_j x^j + x^{\frac{n}{2}} \sum_{j=0}^{\frac{n}{2}-1} a_{\frac{n}{2}+j} x^j = A_1(x) + x^{\frac{n}{2}} A_2(x)$$

$$B(x) = \sum_{j=0}^{n-1} b_j x^j = \sum_{j=0}^{\frac{n}{2}-1} b_j x^j + x^{\frac{n}{2}} \sum_{j=0}^{\frac{n}{2}-1} b_{\frac{n}{2}+j} x^j = B_1(x) + x^{\frac{n}{2}} B_2(x)$$

Then $C(x) = A(x)B(x)$

$$= A_1(x)B_1(x) + x^{\frac{n}{2}}[A_1(x)B_2(x) + A_2(x)B_1(x)] + x^n\, A_2(x)B_2(x)$$

But $A_1(x)B_2(x) + A_2(x)B_1(x)$

$$= [A_1(x) + A_2\,(x)][B_1(x) + B_2\,(x)] - A_1(x)B_1(x) - A_2(x)B_2(x)$$

3 recursive multiplications of polynomials of degree bound $\frac{n}{2}$.

Similar recurrence as in Karatsuba's integer multiplication algorithm leading to a complexity of $\mathrm{O}\left(n^{\log_2 3}\right) = \mathrm{O}\left(n^{1.59}\right)$.

Point-Value Representation of Polynomials

A point-value representation of a polynomial $A(x)$ is a set of n point-value pairs $\{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$ such that all x_k are distinct and $y_k = A(x_k)$ for $0 \le k \le n-1$.

A polynomial has many point-value representations.

Adding Two Polynomials:

Suppose we have point-value representations of two polynomials of degree bound n using the same set of n points.

$$A: \{(x_0, y_0^a), (x_1, y_1^a), \dots, (x_{n-1}, y_{n-1}^a)\}$$

$$B: \{(x_0, y_0^b), (x_1, y_1^b), \dots, (x_{n-1}, y_{n-1}^b)\}$$

If $C(x) = A(x) + B(x)$ then

$$C: \{(x_0, y_0^a + y_0^b), (x_1, y_1^a + y_1^b), \dots, (x_{n-1}, y_{n-1}^a + y_{n-1}^b)\}$$

Thus polynomial addition takes $\Theta(n)$ time.

Point-Value Representation of Polynomials

Multiplying Two Polynomials:

Suppose we have *extended* (why?) point-value representations of two polynomials of degree bound n using the same set of $2n$ points.

$$A: \{(x_0, y_0^a), (x_1, y_1^a), \dots, (x_{2n-1}, y_{2n-1}^a)\}$$

$$B: \{(x_0, y_0^b), (x_1, y_1^b), \dots, (x_{2n-1}, y_{2n-1}^b)\}$$

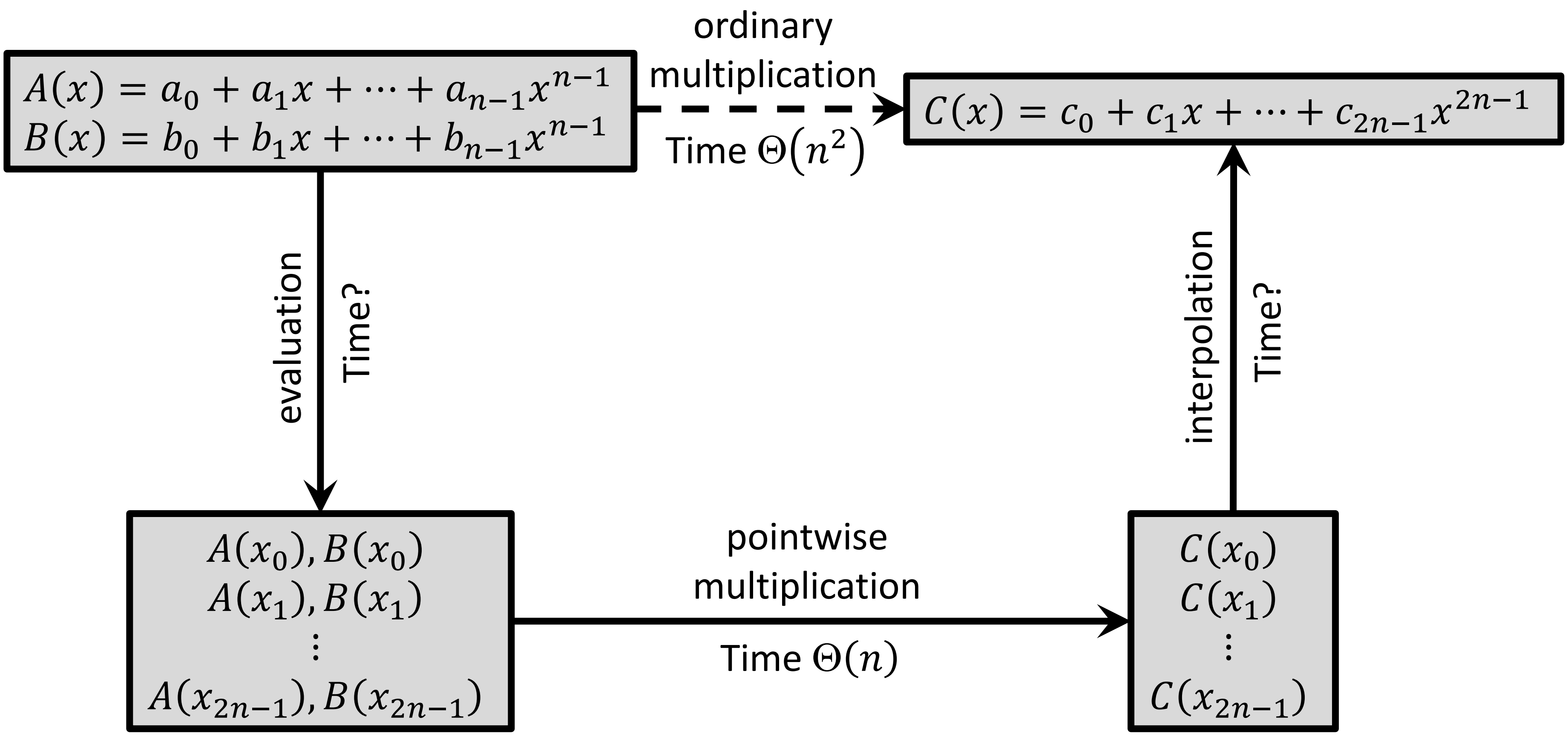
If $C(x) = A(x)B(x)$ then

$$C: \{(x_0, y_0^a y_0^b), (x_1, y_1^a y_1^b), \dots, (x_{2n-1}, y_{2n-1}^a y_{2n-1}^b)\}$$

Thus polynomial multiplication also takes only $\Theta(n)$ time!

(compare this with the $\Theta(n^2)$ time needed in the coefficient form)

Faster Polynomial Multiplication?
(in Coefficient Form)



Faster Polynomial Multiplication? (in Coefficient Form)

Coefficient Representation $\Rightarrow$ Point-Value Representation:

We select any set of n distinct points $\{x_0, x_1, \dots, x_{n-1}\}$, and evaluate $A(x_k)$ for $0 \le k \le n-1$.

Using Horner's rule this approach takes $\Theta(n^2)$ time.

Point-Value Representation $\Rightarrow$ Coefficient Representation:

We can interpolate using Lagrange's formula:

$$A(x) = \sum_{k=0}^{n-1} \frac{\prod_{j \ne k}(x - x_j)}{\prod_{j \ne k}(x_k - x_j)} y_k$$

This again takes $\Theta(n^2)$ time.

In both cases we need to do much better!

Coefficient Form ⇒ Point-Value Form

A polynomial of degree bound n: $A(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$

A set of n distinct points: $\{x_0, x_1, \ldots, x_{n-1}\}$

Compute point-value form: $\{(x_0, A(x_0)), (x_1, A(x_1)), \ldots, (x_{n-1}, A(x_{n-1}))\}$

Using matrix notation:

$$\begin{bmatrix} A(x_0) \\ A(x_1) \\ \cdot \\ \cdot \\ \cdot \\ A(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & (x_0)^2 & \cdots & (x_0)^{n-1} \\ 1 & x_1 & (x_1)^2 & \cdots & (x_1)^{n-1} \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ 1 & x_{n-1} & (x_{n-1})^2 & \cdots & (x_{n-1})^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \cdot \\ \cdot \\ \cdot \\ a_{n-1} \end{bmatrix}$$

We want to choose the set of points in a way that simplifies the multiplication.

In the rest of the lecture on this topic we will assume:

n is a power of 2.

Coefficient Form ⇒ Point-Value Form

Let's choose $x_{n/2+j} = -x_j$ for $0 \le j \le n/2 - 1$. Then

$$\begin{bmatrix} A(x_0) \\ A(x_1) \\ \cdot \\ A(x_{n/2-1}) \\ \hline A(x_{n/2+0}) \\ A(x_{n/2+1}) \\ \cdot \\ A(x_{n/2+(n/2-1)}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & (x_0)^2 & \cdots & (x_0)^{n-1} \\ 1 & x_1 & (x_1)^2 & \cdots & (x_1)^{n-1} \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ 1 & x_{n/2-1} & (x_{n/2-1})^2 & \cdots & (x_{n/2-1})^{n-1} \\ \hline 1 & -x_0 & (-x_0)^2 & \cdots & (-x_0)^{n-1} \\ 1 & -x_1 & (-x_1)^2 & \cdots & (-x_1)^{n-1} \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ 1 & -x_{n/2-1} & (-x_{n/2-1})^2 & \cdots & (-x_{n/2-1})^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ a_{n-1} \end{bmatrix}$$

Observe that for $0 \le j \le n/2 - 1$: $(x_{n/2+j})^k = \begin{cases} (x_j)^k, & if\ k = even, \\ -(x_j)^k, & if\ k = odd. \end{cases}$

Thus we have just split the original $n \times n$ matrix into two almost similar $\frac{n}{2} \times n$ matrices!

Coefficient Form ⇒ Point-Value Form

How and how much do we save?

$$A(x) = \sum_{l=0}^{n-1} a_l x^l = \sum_{l=0}^{n/2-1} a_{2l} x^{2l} + \sum_{l=0}^{n/2-1} a_{2l+1} x^{2l+1}$$

$$= \sum_{l=0}^{n/2-1} a_{2l} \left(x^2\right)^l + x \sum_{l=0}^{n/2-1} a_{2l+1} \left(x^2\right)^l = A_{even}\left(x^2\right) + x A_{odd}\left(x^2\right),$$

where, $A_{even}(x) = \sum_{l=0}^{n/2-1} a_{2l} x^l$ and $A_{odd}(x) = \sum_{l=0}^{n/2-1} a_{2l+1} x^l$.

Observe that for $0 \leq j \leq n/2 - 1$:

$$A\left(x_j\right) = A_{even}\left(x_j^2\right) + x_j A_{odd}\left(x_j^2\right)$$

$$A\left(x_{n/2+j}\right) = A\left(-x_j\right) = A_{even}\left(x_j^2\right) - x_j A_{odd}\left(x_j^2\right)$$

So in order to evaluate $A\left(x_j\right)$ for all $0 \leq j \leq n - 1$, we need:

- $n/2$ evaluations of A_{even} and $n/2$ evaluations of A_{odd}
- n multiplications
- $n/2$ additions and $n/2$ subtractions

Thus we save about half the computation!

Coefficient Form ⇒ Point-Value Form

If we can recursively evaluate A_{even} and A_{odd} using the same approach, we get the following recurrence relation for the running time of the algorithm:

$$T(n) = \begin{cases} \Theta(1), & if\ n = 1, \\ 2T\left(\frac{n}{2}\right) + \Theta(n), & otherwise. \end{cases}$$

$$= \Theta(n \log n)$$

Our trick was to evaluate A at x (positive) and $-x$ (negative).

But inputs to A_{even} and A_{odd} are always of the form x^2 (positive)!

How can we apply the same trick?

Coefficient Form ⇒ Point-Value Form

Let us consider the evaluation of $A_{even}(x_j)$ for $0 \le j \le n/2 - 1$:

$$\begin{bmatrix} A_{even}(x_0) \\ A_{even}(x_1) \\ \cdot \\ \cdot \\ \cdot \\ A_{even}(x_{n/2-1}) \end{bmatrix} = \begin{bmatrix} 1 & (x_0)^2 & (x_0)^4 & \cdots & (x_0)^{n-2} \\ 1 & (x_1)^2 & (x_1)^4 & \cdots & (x_1)^{n-2} \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ 1 & (x_{n/2-1})^2 & (x_{n/2-1})^4 & \cdots & (x_{n/2-1})^{n-2} \end{bmatrix} \begin{bmatrix} a_0 \\ a_2 \\ a_4 \\ \cdot \\ \cdot \\ a_{n-2} \end{bmatrix}$$

In order to apply the same trick on A_{even} we must set:

$$(x_{n/4+j})^2 = -(x_j)^2 \text{ for } 0 \le j \le n/4 - 1$$

Coefficient Form ⇒ Point-Value Form

In A_{even} we set: $x_{n/4+j}^2 = -x_j^2$ for $0 \leq j \leq n/4-1$. Then

$$\begin{bmatrix} A_{even}(x_0) \\ A_{even}(x_1) \\ \cdot \\ A_{even}(x_{n/4-1}) \\ A_{even}(x_{n/4+0}) \\ A_{even}(x_{n/4+1}) \\ \cdot \\ A_{even}(x_{n/4+(n/4-1)}) \end{bmatrix} = \begin{bmatrix} 1 & x_0^2 & (x_0^2)^2 & \cdots & (x_0^2)^{\frac{n}{2}-1} \\ 1 & x_1^2 & (x_1^2)^2 & \cdots & (x_1^2)^{\frac{n}{2}-1} \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ 1 & x_{n/4-1}^2 & (x_{n/4-1}^2)^2 & \cdots & (x_{n/4-1}^2)^{\frac{n}{2}-1} \\ 1 & -x_0^2 & (-x_0^2)^2 & \cdots & (-x_0^2)^{\frac{n}{2}-1} \\ 1 & -x_1^2 & (-x_1^2)^2 & \cdots & (-x_1^2)^{\frac{n}{2}-1} \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ 1 & -x_{n/4-1}^2 & (-x_{n/2-1}^2)^2 & \cdots & (-x_{n/4-1}^2)^{\frac{n}{2}-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_2 \\ a_4 \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ a_{n-2} \end{bmatrix}$$

This means setting $x_{n/4+j} = ix_j$, where $i = \sqrt{-1}$ (imaginary)!

This also allows us to apply the same trick on A_{odd}.

Coefficient Form ⇒ Point-Value Form

We can apply the trick once if we set:

$$x_{n/2+j} = -x_j \text{ for } 0 \le j \le n/2 - 1$$

We can apply the trick (recursively) 2 times if we also set:

$$\left(x_{n/2^2+j}\right)^2 = -\left(x_j\right)^2 \text{ for } 0 \le j \le n/2^2 - 1$$

We can apply the trick (recursively) 3 times if we also set:

$$\left(x_{n/2^3+j}\right)^{2^2} = -\left(x_j\right)^{2^2} \text{ for } 0 \le j \le n/2^3 - 1$$

⋮

We can apply the trick (recursively) k times if we also set:

$$\left(x_{n/2^k+j}\right)^{2^{k-1}} = -\left(x_j\right)^{2^{k-1}} \text{ for } 0 \le j \le n/2^k - 1$$

Coefficient Form $\Rightarrow$ Point-Value Form

Consider the t^{th} *primitive root of unity*:

$$\omega_t = e^{\frac{2\pi i}{t}} = \cos\frac{2\pi}{t} + i \cdot \sin\frac{2\pi}{t} \quad \left(i = \sqrt{-1}\right)$$

Then

$$x_{n/2+j} = -x_j \Rightarrow x_{n/2^1+j} = \omega_{2^1} \cdot x_j$$

$$\left(x_{n/2^2+j}\right)^2 = -\left(x_j\right)^2 \Rightarrow x_{n/2^2+j} = \omega_{2^2} \cdot x_j$$

$$\left(x_{n/2^3+j}\right)^{2^2} = -\left(x_j\right)^{2^2} \Rightarrow x_{n/2^3+j} = \omega_{2^3} \cdot x_j$$

$$\vdots$$

$$\left(x_{n/2^k+j}\right)^{2^{k-1}} = -\left(x_j\right)^{2^{k-1}} \Rightarrow x_{n/2^k+j} = \omega_{2^k} \cdot x_j$$

Coefficient Form $\Rightarrow$ Point-Value Form

If $n = 2^k$ we would like to apply the trick k times recursively.

What values should we choose for $\{x_0, x_1, \dots, x_{n-1}\}$?

Example: For $n = 2^3$ we need to choose $\{x_0, x_1, \dots, x_7\}$.

Choose: $x_0 = 1 \qquad = \omega_8^0$

$k = 3$: $x_1 = \omega_{2^3} \cdot x_0 \qquad = \omega_8^1$

$k = 2$: $x_2 = \omega_{2^2} \cdot x_0 \qquad = \omega_8^2$

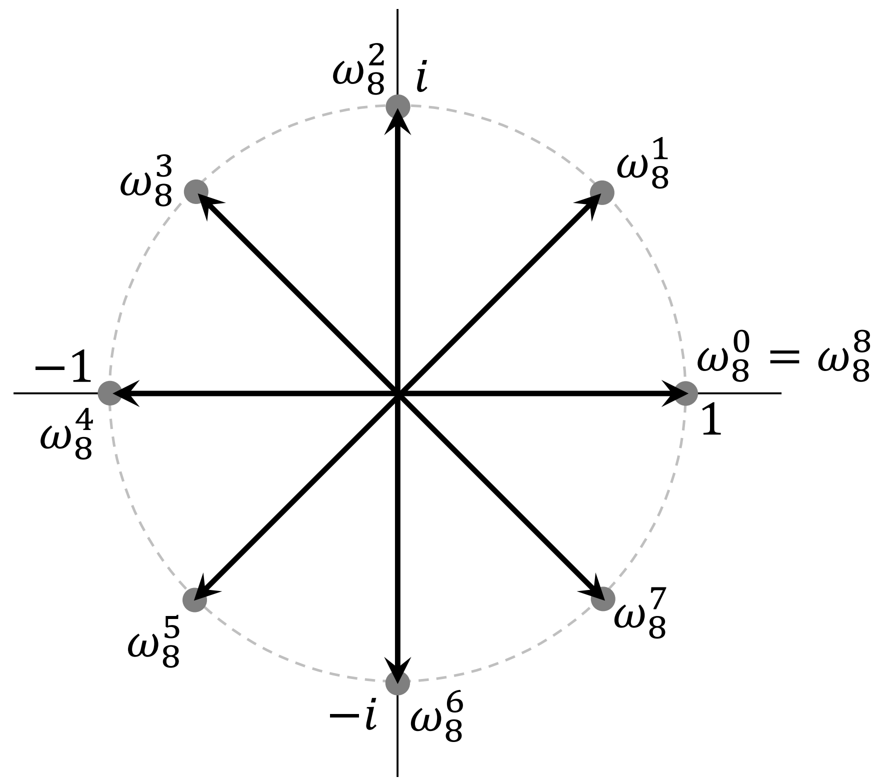
$x_3 = \omega_{2^2} \cdot x_1 \qquad = \omega_8^3$

$k = 1$: $x_4 = \omega_{2^1} \cdot x_0 \qquad = \omega_8^4$

$x_5 = \omega_{2^1} \cdot x_1 \qquad = \omega_8^5$

$x_6 = \omega_{2^1} \cdot x_2 \qquad = \omega_8^6$

$x_7 = \omega_{2^1} \cdot x_3 \qquad = \omega_8^7$



complex 8^{th} roots of unity

Coefficient Form ⇒ Point-Value Form

For a polynomial of degree bound $n = 2^k$, we need to apply the trick recursively at most $\log n = k$ times.

We choose $x_0 = 1 = \omega_n^0$ and set $x_j = \omega_n^j$ for $1 \le j \le n-1$.

Then we compute the following product:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \cdot \\ \cdot \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} A(1) \\ A(\omega_n) \\ A(\omega_n^2) \\ \cdot \\ \cdot \\ A(\omega_n^{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & (\omega_n)^2 & \cdots & (\omega_n)^{n-1} \\ 1 & \omega_n^2 & (\omega_n^2)^2 & \cdots & (\omega_n^2)^{n-1} \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ 1 & \omega_n^{n-1} & (\omega_n^{n-1})^2 & \cdots & (\omega_n^{n-1})^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \cdot \\ \cdot \\ a_{n-1} \end{bmatrix}$$

The vector $y = (y_0, y_1, \cdots, y_{n-1})$ is called the *discrete Fourier transform* (DFT) of $(a_0, a_1, \cdots, a_{n-1})$.

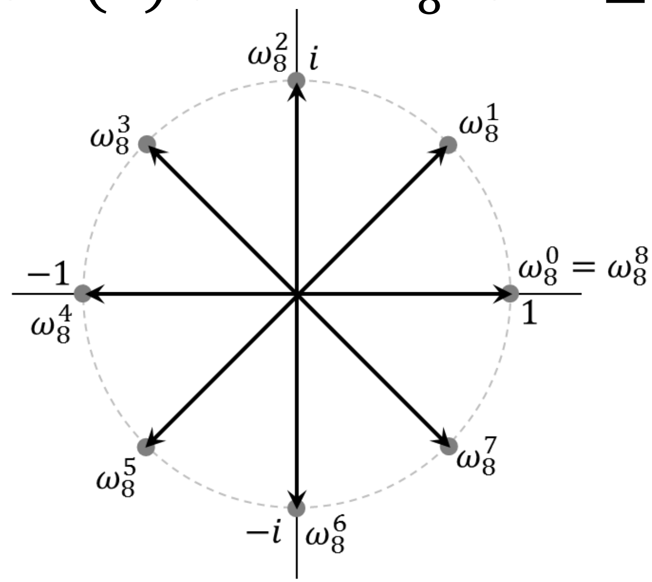
This method of computing DFT is called the *fast Fourier transform* (FFT) method.

Coefficient Form ⇒ Point-Value Form

Example: For $n = 2^3 = 8$:

$$A(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7$$

We need to evaluate $A(x)$ at $x = \omega_8^i$ for $0 \le i < 8$.



complex 8^{th} roots of unity

Now $A(x) = A_{even}(x^2) + x \cdot A_{odd}(x^2)$,

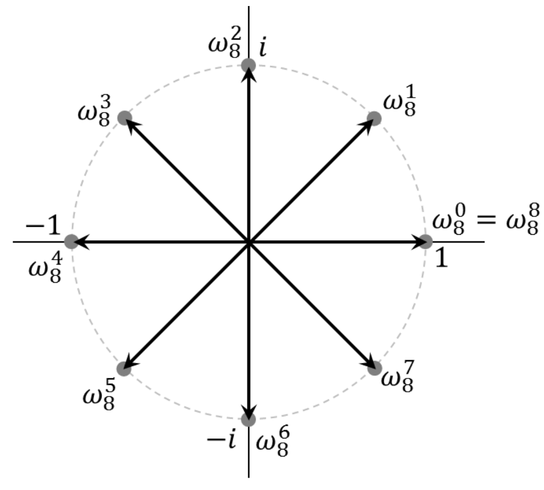
where $A_{even}(y) = a_0 + a_2y + a_4y^2 + a_6y^3$

and $A_{odd}(y) = a_1 + a_3y + a_5y^2 + a_7y^3$

Coefficient Form ⇒ Point-Value Form

Observe that:

$$\omega_8^0 = \omega_8^8 \quad = \omega_4^0$$
$$\omega_8^2 = \omega_8^{10} \quad = \omega_4^1$$
$$\omega_8^4 = \omega_8^{12} \quad = \omega_4^2$$
$$\omega_8^6 = \omega_8^{14} \quad = \omega_4^3$$



Also:

$$\omega_8^4 = -\omega_8^0$$
$$\omega_8^5 = -\omega_8^1$$
$$\omega_8^6 = -\omega_8^2$$
$$\omega_8^7 = -\omega_8^3$$

$$A(\omega_8^0) = A_{even}(\omega_8^0) + \omega_8^0 \cdot A_{odd}(\omega_8^0) = A_{even}(\omega_4^0) + \omega_8^0 \cdot A_{odd}(\omega_4^0),$$
$$A(\omega_8^1) = A_{even}(\omega_8^2) + \omega_8^1 \cdot A_{odd}(\omega_8^2) = A_{even}(\omega_4^1) + \omega_8^1 \cdot A_{odd}(\omega_4^1),$$
$$A(\omega_8^2) = A_{even}(\omega_8^4) + \omega_8^2 \cdot A_{odd}(\omega_8^4) = A_{even}(\omega_4^2) + \omega_8^2 \cdot A_{odd}(\omega_4^2),$$
$$A(\omega_8^3) = A_{even}(\omega_8^6) + \omega_8^3 \cdot A_{odd}(\omega_8^6) = A_{even}(\omega_4^3) + \omega_8^3 \cdot A_{odd}(\omega_4^3),$$

$$A(\omega_8^4) = A_{even}(\omega_8^8) + \omega_8^4 \cdot A_{odd}(\omega_8^8) = A_{even}(\omega_4^0) - \omega_8^0 \cdot A_{odd}(\omega_4^0),$$
$$A(\omega_8^5) = A_{even}(\omega_8^{10}) + \omega_8^5 \cdot A_{odd}(\omega_8^{10}) = A_{even}(\omega_4^1) - \omega_8^1 \cdot A_{odd}(\omega_4^1),$$
$$A(\omega_8^6) = A_{even}(\omega_8^{12}) + \omega_8^6 \cdot A_{odd}(\omega_8^{12}) = A_{even}(\omega_4^2) - \omega_8^2 \cdot A_{odd}(\omega_4^2),$$
$$A(\omega_8^7) = A_{even}(\omega_8^{14}) + \omega_8^7 \cdot A_{odd}(\omega_8^{14}) = A_{even}(\omega_4^3) - \omega_8^3 \cdot A_{odd}(\omega_4^3),$$

Coefficient Form ⇒ Point-Value Form

Rec-FFT ($(a_0, a_1, \ldots, a_{n-1})$)   { $n = 2^k$ *for integer* $k \geq 0$ }

if $n = 1$ *then*

  return (a_0)

$\omega_n \leftarrow e^{2\pi i/n}$

$\omega \leftarrow 1$

$y^{even} \leftarrow$ *Rec-FFT* ($(a_0, a_2, \ldots, a_{n-2})$)

$y^{odd} \leftarrow$ *Rec-FFT* ($(a_1, a_3, \ldots, a_{n-1})$)

for $j \leftarrow 0$ *to* $n/2 - 1$ *do*

  $y_j \leftarrow y_j^{even} + \omega\, y_j^{odd}$

  $y_{n/2+j} \leftarrow y_j^{even} - \omega\, y_j^{odd}$

  $\omega \leftarrow \omega\, \omega_n$

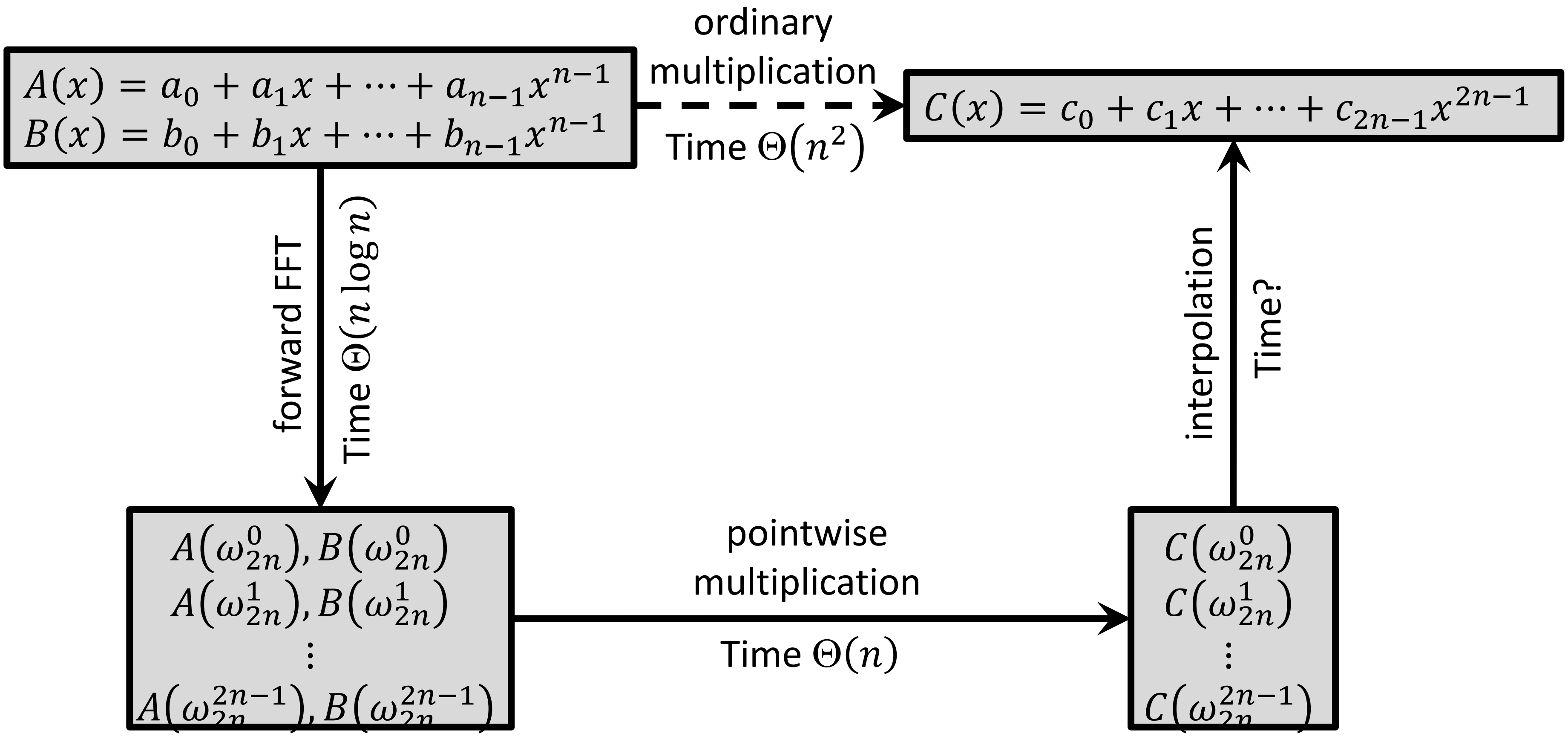
return y

Running time:

$$T(n) = \begin{cases} \Theta(1), & if\ n = 1, \\ 2T\left(\frac{n}{2}\right) + \Theta(n), & otherwise. \end{cases}$$

$$= \Theta(n \log n)$$

Faster Polynomial Multiplication? (in Coefficient Form)



Point-Value Form ⇒ Coefficient Form

Given:

$$\underbrace{\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & (\omega_n)^2 & \cdots & (\omega_n)^{n-1} \\ 1 & \omega_n^2 & \left(\omega_n^2\right)^2 & \cdots & \left(\omega_n^2\right)^{n-1} \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ 1 & \omega_n^{n-1} & \left(\omega_n^{n-1}\right)^2 & \cdots & \left(\omega_n^{n-1}\right)^{n-1} \end{bmatrix}}_{\substack{V(\omega_n) \\ Vandermonde\ Matrix}} \underbrace{\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \cdot \\ \cdot \\ a_{n-1} \end{bmatrix}}_{\bar{a}} = \underbrace{\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \cdot \\ \cdot \\ y_{n-1} \end{bmatrix}}_{\bar{y}}$$

$$\Rightarrow V(\omega_n) \cdot \bar{a} = \bar{y}$$

We want to solve: $\bar{a} = [V(\omega_n)]^{-1} \cdot \bar{y}$

It turns out that: $[V(\omega_n)]^{-1} = \frac{1}{n} V\left(\frac{1}{\omega_n}\right)$

That means $[V(\omega_n)]^{-1}$ looks almost similar to $V(\omega_n)$!

Point-Value Form ⇒ Coefficient Form

Show that: $[V(\omega_n)]^{-1} = \frac{1}{n} V\left(\frac{1}{\omega_n}\right)$

Let $U(\omega_n) = \frac{1}{n} V\left(\frac{1}{\omega_n}\right)$

We want to show that $U(\omega_n)V(\omega_n) = I_n$,
where I_n is the $n \times n$ identity matrix.

Observe that for $0 \le j, k \le n-1$, the $(j,k)^{th}$ entries are:

$$[V(\omega_n)]_{jk} = \omega_n^{jk} \quad \text{and} \quad [U(\omega_n)]_{jk} = \frac{1}{n}\omega_n^{-jk}$$

Then entry (p,q) of $U(\omega_n)V(\omega_n)$,

$$[U(\omega_n)V(\omega_n)]_{pq} = \sum_{k=0}^{n-1} [U(\omega_n)]_{pk} [V(\omega_n)]_{kq} = \frac{1}{n}\sum_{k=0}^{n-1} \omega_n^{k(q-p)}$$

Point-Value Form ⇒ Coefficient Form

$$[U(\omega_n)V(\omega_n)]_{pq} = \frac{1}{n}\sum_{k=0}^{n-1} \omega_n^{k(q-p)}$$

CASE $p = q$:

$$[U(\omega_n)V(\omega_n)]_{pq} = \frac{1}{n}\sum_{k=0}^{n-1} \omega_n^{0} = \frac{1}{n}\sum_{k=0}^{n-1} 1 = \frac{1}{n} \times n = 1$$

CASE $p \neq q$:

$$[U(\omega_n)V(\omega_n)]_{pq} = \frac{1}{n}\sum_{k=0}^{n-1} \left(\omega_n^{q-p}\right)^k = \frac{1}{n} \times \frac{\left(\omega_n^{q-p}\right)^n - 1}{\omega_n^{q-p} - 1}$$

$$= \frac{1}{n} \times \frac{(\omega_n^{n})^{q-p} - 1}{\omega_n^{q-p} - 1} = \frac{1}{n} \times \frac{(1)^{q-p} - 1}{\omega_n^{q-p} - 1} = 0$$

Hence $U(\omega_n)V(\omega_n) = I_n$

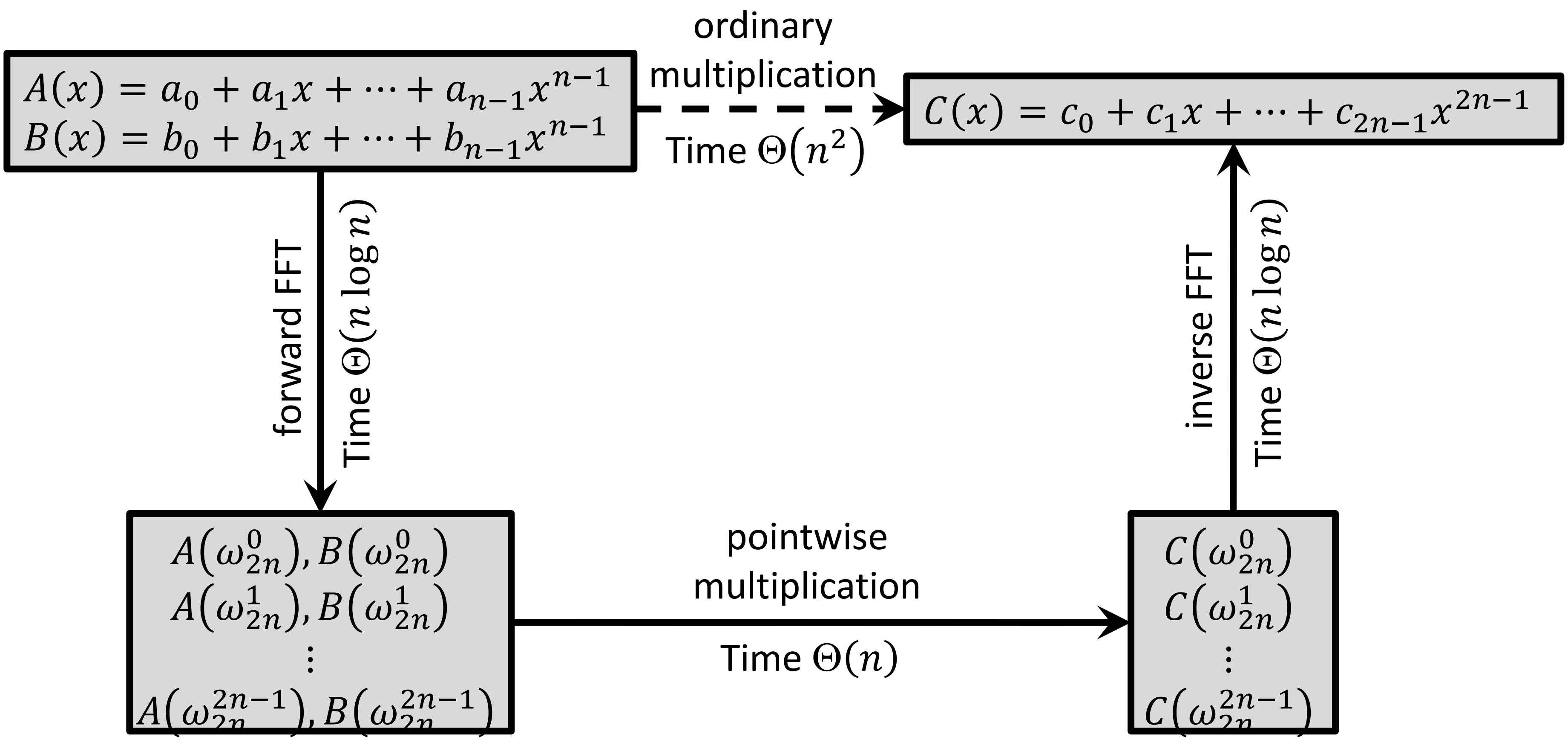
Point-Value Form ⇒ Coefficient Form

We need to compute the following matrix-vector product:

$$\underbrace{\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \cdot \\ \cdot \\ a_{n-1} \end{bmatrix}}_{\bar{a}} = \frac{1}{n} \times \underbrace{\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \frac{1}{\omega_n} & \left(\frac{1}{\omega_n}\right)^2 & \cdots & \left(\frac{1}{\omega_n}\right)^{n-1} \\ 1 & \frac{1}{\omega_n^2} & \left(\frac{1}{\omega_n^2}\right)^2 & \cdots & \left(\frac{1}{\omega_n^2}\right)^{n-1} \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ 1 & \frac{1}{\omega_n^{n-1}} & \left(\frac{1}{\omega_n^{n-1}}\right)^2 & \cdots & \left(\frac{1}{\omega_n^{n-1}}\right)^{n-1} \end{bmatrix}}_{[V(\omega_n)]^{-1}} \underbrace{\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \cdot \\ \cdot \\ y_{n-1} \end{bmatrix}}_{\bar{y}}$$

This inverse problem is almost similar to the forward problem, and can be solved in $\Theta(n \log n)$ time using the same algorithm as the forward FFT with only minor modifications!

Faster Polynomial Multiplication? (in Coefficient Form)



Two polynomials of degree bound n given in the coefficient form can be multiplied in $\Theta(n \log n)$ time!

Some Applications of Fourier Transform and FFT

- Signal processing
- Image processing
- Noise reduction
- Data compression
- Solving partial differential equation
- Multiplication of large integers
- Polynomial multiplication
- Molecular docking

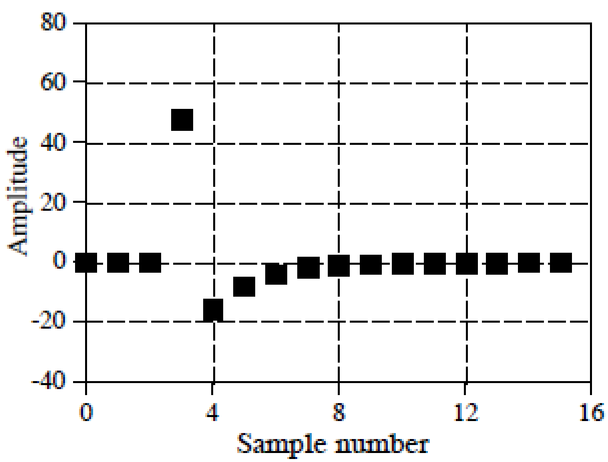
Some Applications of Fourier Transform and FFT

Jean Baptiste Joseph Fourier

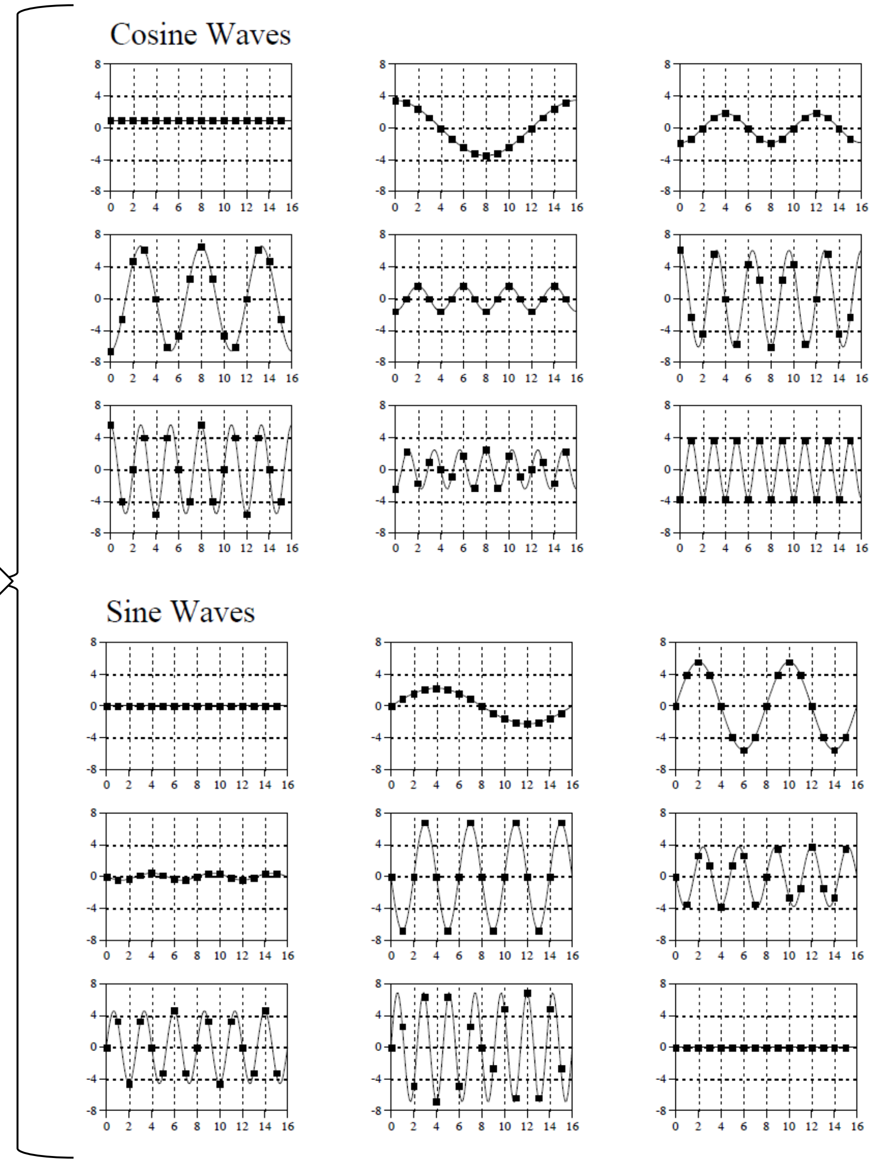
Any periodic signal can be represented as a sum of a series of sinusoidal (sine & cosine) waves. [1807]

Spatial (Time) Domain ⇔ Frequency Domain

Spatial (Time) Domain



Frequency Domain



Source: The Scientist and Engineer's Guide to Digital Signal Processing by Steven W. Smith

Spatial (Time) Domain ⇔ Frequency Domain

Function $s(x)$ (in red) is a sum of six sine functions of different amplitudes and harmonically related frequencies. The Fourier transform, $S(f)$ (in blue), which depicts amplitude vs frequency, reveals the 6 frequencies and their amplitudes.

Source: http://en.wikipedia.org/wiki/Fourier_series#mediaviewer/File:Fourier_series_and_transform.gif (uploaded by Bob K.)

Spatial (Time) Domain ⇔ Frequency Domain

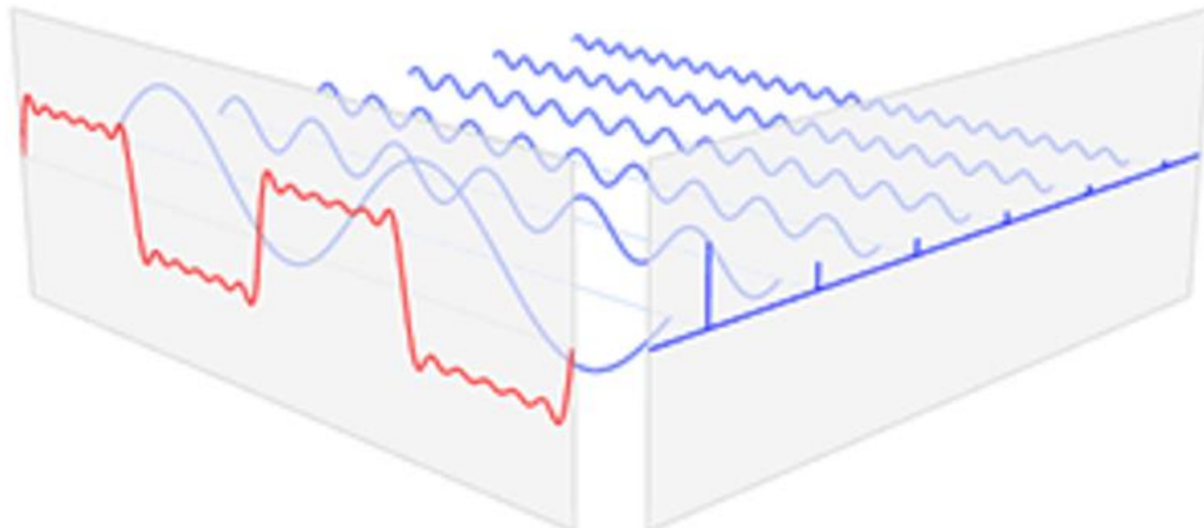
$$s_6(x)$$

$$a_n \cos(nx) + b_n \sin(nx)$$

Function *s*(*x*) (in red) is a sum of six sine functions of different amplitudes and harmonically related frequencies. The Fourier transform, *S*(*f*) (in blue), which depicts amplitude vs frequency, reveals the 6 frequencies and their amplitudes.

Source: http://en.wikipedia.org/wiki/Fourier_series#mediaviewer/File:Fourier_series_and_transform.gif (uploaded by Bob K.)

Spatial (Time) Domain ⇔ Frequency Domain



Function *s*(*x*) (in red) is a sum of six sine functions of different amplitudes and harmonically related frequencies. The Fourier transform, *S*(*f*) (in blue), which depicts amplitude vs frequency, reveals the 6 frequencies and their amplitudes.

Source: http://en.wikipedia.org/wiki/Fourier_series#mediaviewer/File:Fourier_series_and_transform.gif (uploaded by Bob K.)

Spatial (Time) Domain ⇔ Frequency Domain

$S(f)$

Function $s(x)$ (in red) is a sum of six sine functions of different amplitudes and harmonically related frequencies. The Fourier transform, $S(f)$ (in blue), which depicts amplitude vs frequency, reveals the 6 frequencies and their amplitudes.

Source: http://en.wikipedia.org/wiki/Fourier_series#mediaviewer/File:Fourier_series_and_transform.gif (uploaded by Bob K.)

Spatial (Time) Domain ⇔ Frequency Domain

Function *s*(*x*) (in red) is a sum of six sine functions of different amplitudes and harmonically related frequencies. The Fourier transform, *S*(*f*) (in blue), which depicts amplitude vs frequency, reveals the 6 frequencies and their amplitudes.

Source: http://en.wikipedia.org/wiki/Fourier_series#mediaviewer/File:Fourier_series_and_transform.gif (uploaded by Bob K.)

Spatial (Time) Domain ⇔ Frequency Domain (Fourier Transforms)

Let $s(t)$ be a signal specified in the time domain.

The strength of $s(t)$ at frequency f is given by:

$$S(f) = \int_{-\infty}^{\infty} s(t) \cdot e^{-2\pi i f t}\, dt$$

Evaluating this integral for all values of f gives the frequency domain function.

Now $s(t)$ can be retrieved by summing up the signal strengths at all possible frequencies:

$$s(t) = \int_{-\infty}^{\infty} S(f) \cdot e^{2\pi i f t}\, df$$

Why do the Transforms Work?

Let's try to get a little intuition behind why the transforms work.

We will look at a very simple example.

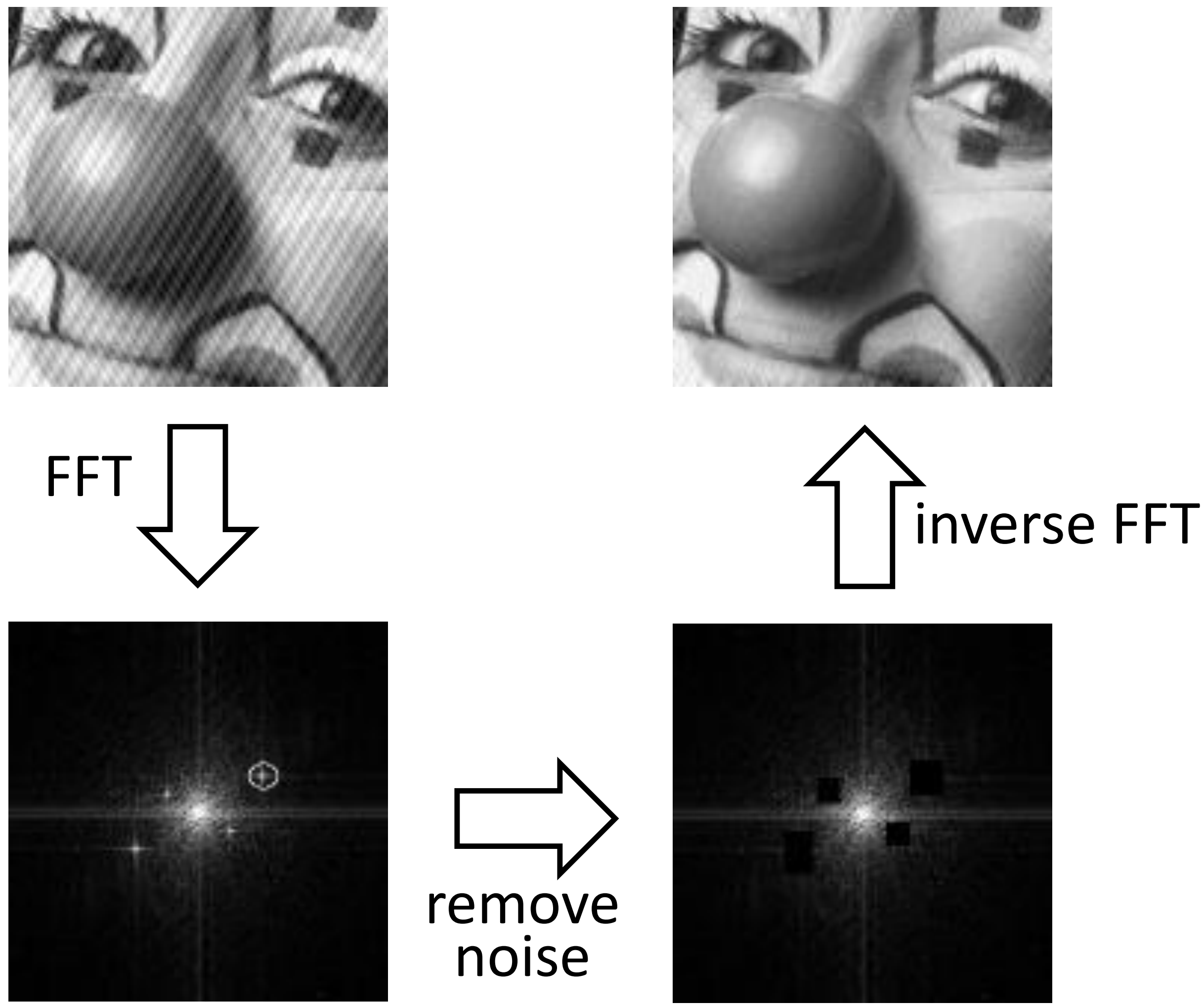
Suppose: $s(t) = \cos(2\pi h \cdot t)$

$$\frac{1}{T}\int_{-T}^{T} s(t)\cdot e^{-2\pi i f t}\,dt = \begin{cases} 1 + \dfrac{\sin(4\pi f T)}{4\pi f T}, & \text{if } f = h, \\ \dfrac{\sin(2\pi(h-f)T)}{2\pi(h-f)T} + \dfrac{\sin(2\pi(h+f)T)}{2\pi(h+f)T}, & \text{otherwise.} \end{cases}$$

$$\Rightarrow \lim_{T\to\infty}\left(\frac{1}{T}\int_{-T}^{T} s(t)\cdot e^{-2\pi i f t}\,dt\right) = \begin{cases} 1, & \text{if } f = h, \\ 0, & \text{otherwise.} \end{cases}$$

So, the transform can detect if $f = h$!

Noise Reduction



Source: http://www.mediacy.com/index.aspx?page=AH_FFTExample

Data Compression

- Discrete Cosine Transforms (DCT) are used for lossy data compression (e.g., MP3, JPEG, MPEG)
- DCT is a Fourier-related transform similar to DFT (Discrete Fourier Transform) but uses only real data (uses cosine waves only instead of both cosine and sine waves)
- Forward DCT transforms data from spatial to frequency domain
- Each frequency component is represented using a fewer number of bits (i.e., truncated / quantized)
- Low amplitude high frequency components are also removed
- Inverse DCT then transforms the data back to spatial domain
- The resulting image compresses better

Data Compression

Transformation to frequency domain using cosine transforms work in the same way as the Fourier transform.

Suppose: $s(t) = \cos(2\pi h \cdot t)$

$$\frac{1}{T}\int_{-T}^{T} s(t)\cdot\cos(2\pi f t)\,dt = \begin{cases} 1 + \dfrac{\sin(4\pi f T)}{4\pi f T}, & \text{if } f = h, \\ \dfrac{\sin(2\pi(h-f)T)}{2\pi(h-f)T} + \dfrac{\sin(2\pi(h+f)T)}{2\pi(h+f)T}, & \text{otherwise.} \end{cases}$$

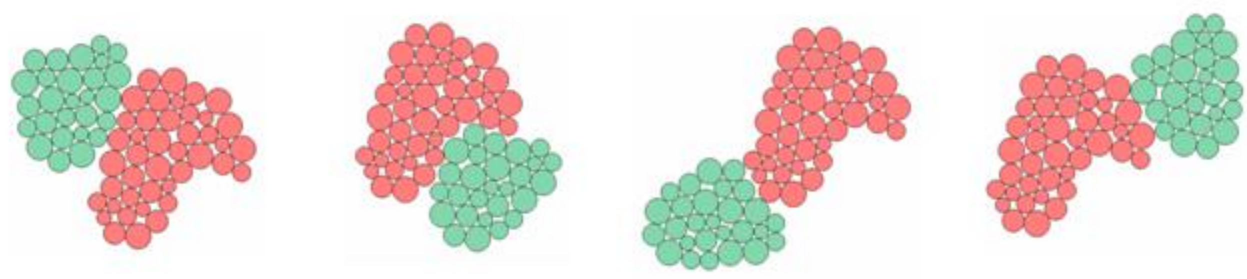
$$\Rightarrow \lim_{T\to\infty}\left(\frac{1}{T}\int_{-T}^{T} s(t)\cdot\cos(2\pi f t)\,dt\right) = \begin{cases} 1, & \text{if } f = h, \\ 0, & \text{otherwise.} \end{cases}$$

So, this transform can also detect if $f = h$.

Protein-Protein Docking

- Knowledge of complexes is used in
  - Drug design
  - Studying molecular assemblies
  - Structure function analysis
  - Protein interactions

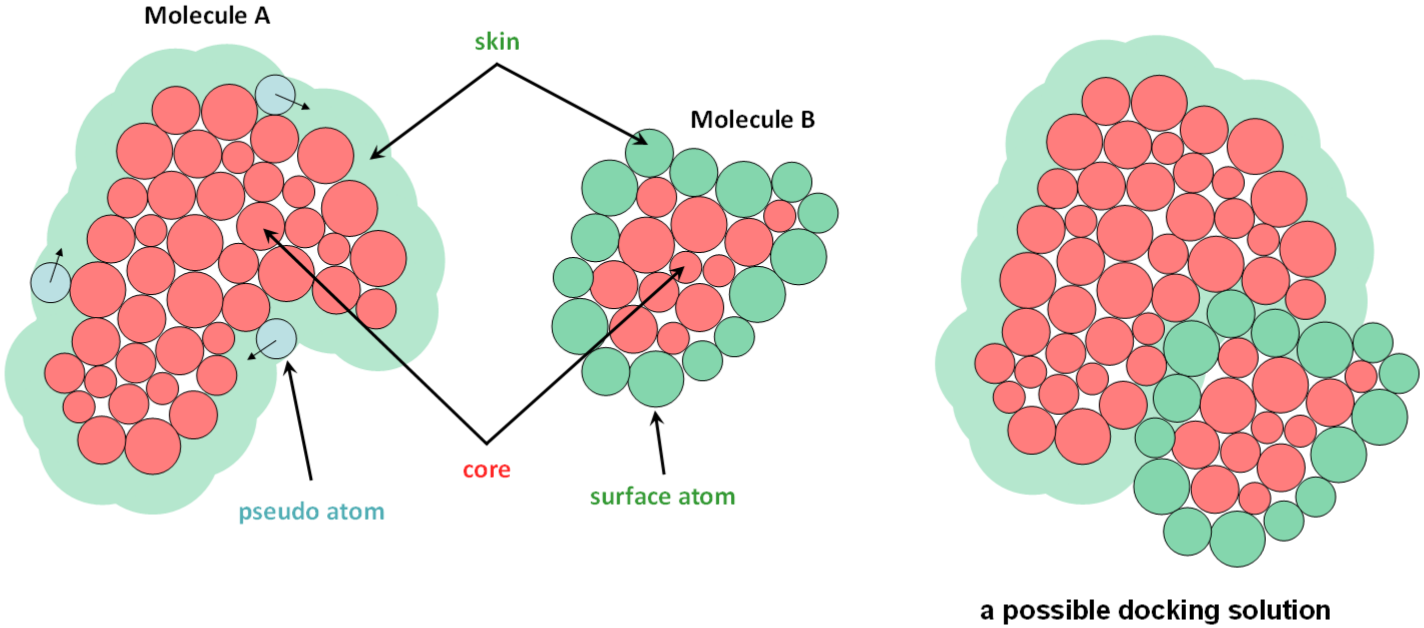
- Protein-Protein Docking: Given two proteins, find the best relative transformation and conformations to obtain a stable complex.



- Docking is a hard problem
  - Search space is huge (6D for rigid proteins)
  - Protein flexibility adds to the difficulty

Shape Complementarity

[Wang'91, Katchalski-Katzir et al.'92, Chen et al.'03]



To maximize skin-skin overlaps and minimize core-core overlaps

- assign positive real weights to skin atoms
- assign positive imaginary weights to core atoms

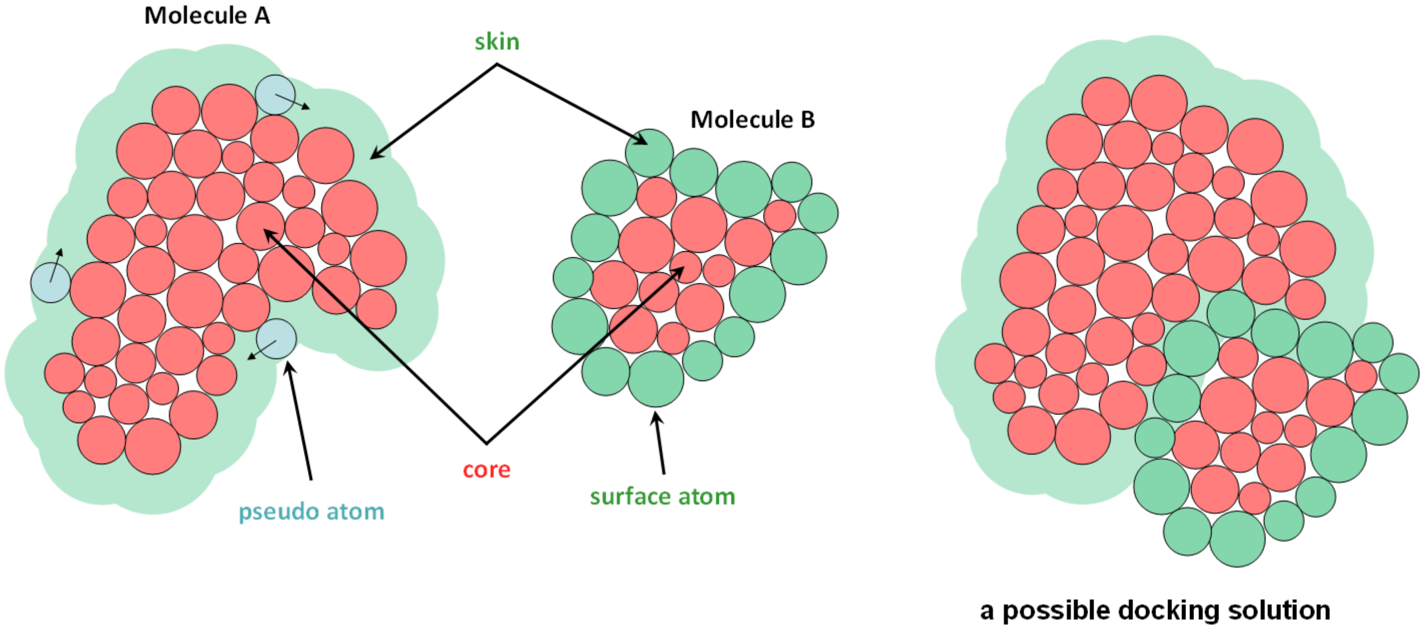
Let A' denote molecule A with the pseudo skin atoms.

For $P \in \{A', B\}$ with M_P atoms, affinity function: $f_P(x) = \sum_{k=1}^{M_P} w_k \cdot g_k(x)$

Here $g_k(x)$ is a Gaussian representation of atom k, and w_k its weight.

Shape Complementarity

[Wang'91, Katchalski-Katzir et al.'92, Chen et al.'03]



Let A' denote molecule A with the pseudo skin atoms.

For $P \in \{A', B\}$ with M_P atoms, affinity function:

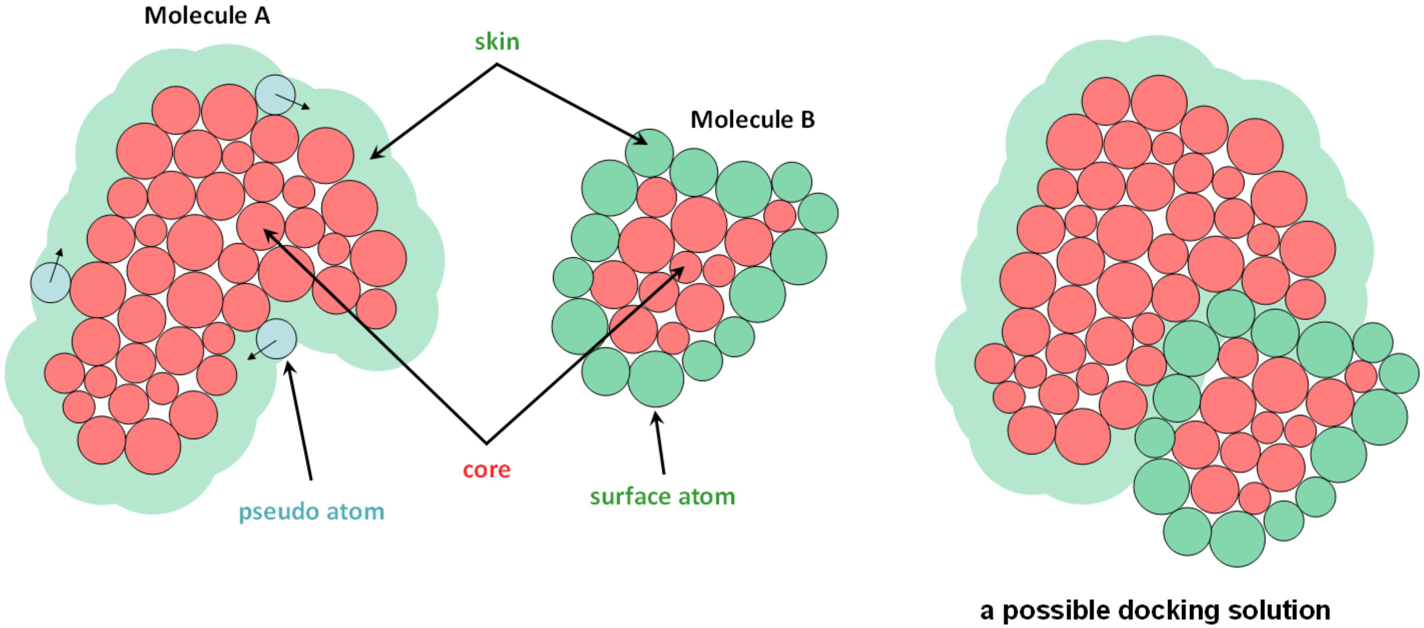
$$f_P(x) = \sum_{k=1}^{M_P} w_k \cdot g_k(x)$$

For rotation r and translation t of molecule B (i.e., $B_{t,r}$),

the interaction score, $F_{A,B}(t,r) = \int_x \; f_{A'}(x) f_{B_{t,r}}(x)\, dx$

Shape Complementarity

[Wang'91, Katchalski-Katzir et al.'92, Chen et al.'03]



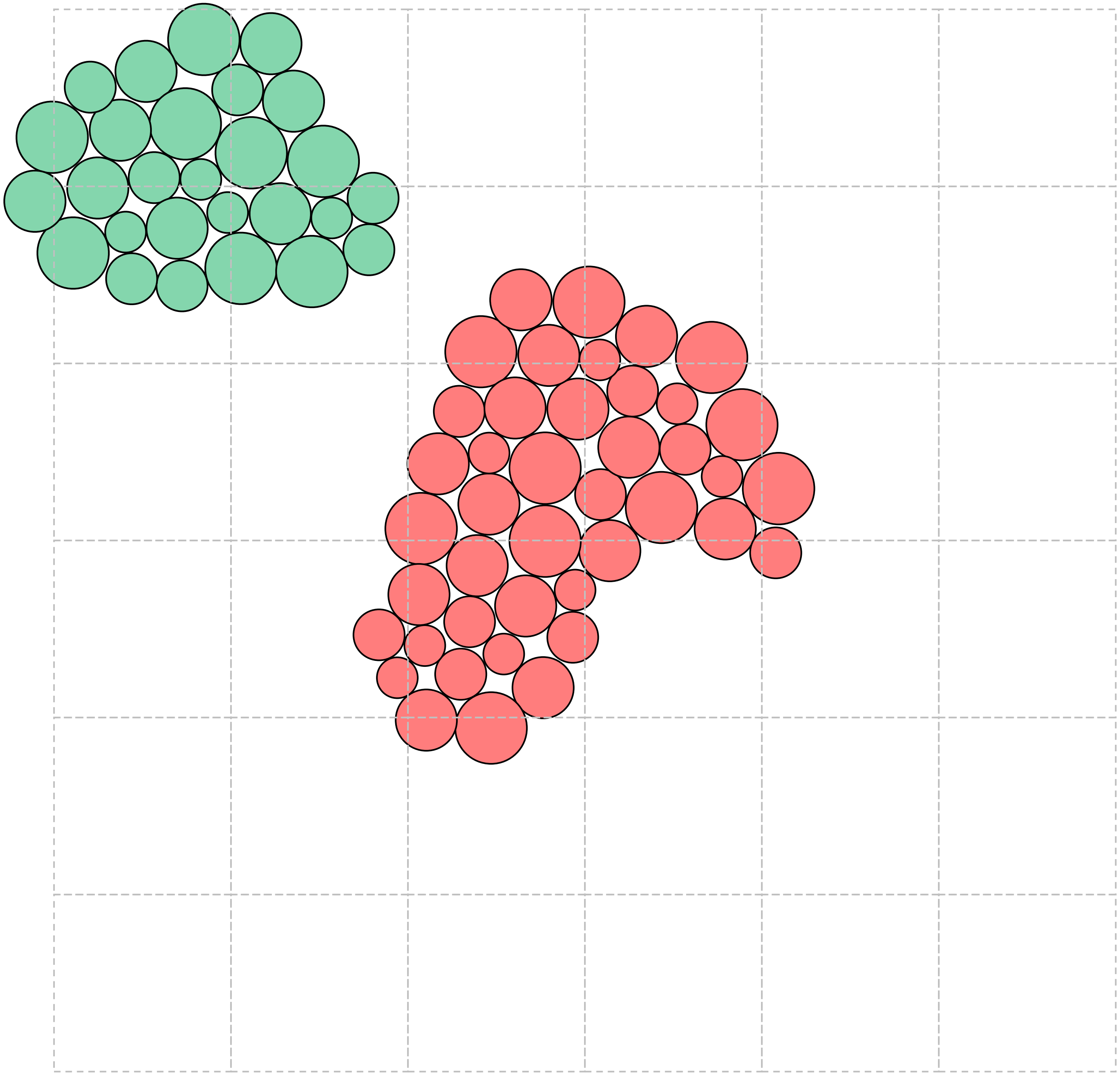
For rotation r and translation t of molecule B (i.e., $B_{t,r}$),

the interaction score, $F_{A,B}(t,r) = \int_x f_{A'}(x) f_{B_{t,r}}(x)\, dx$

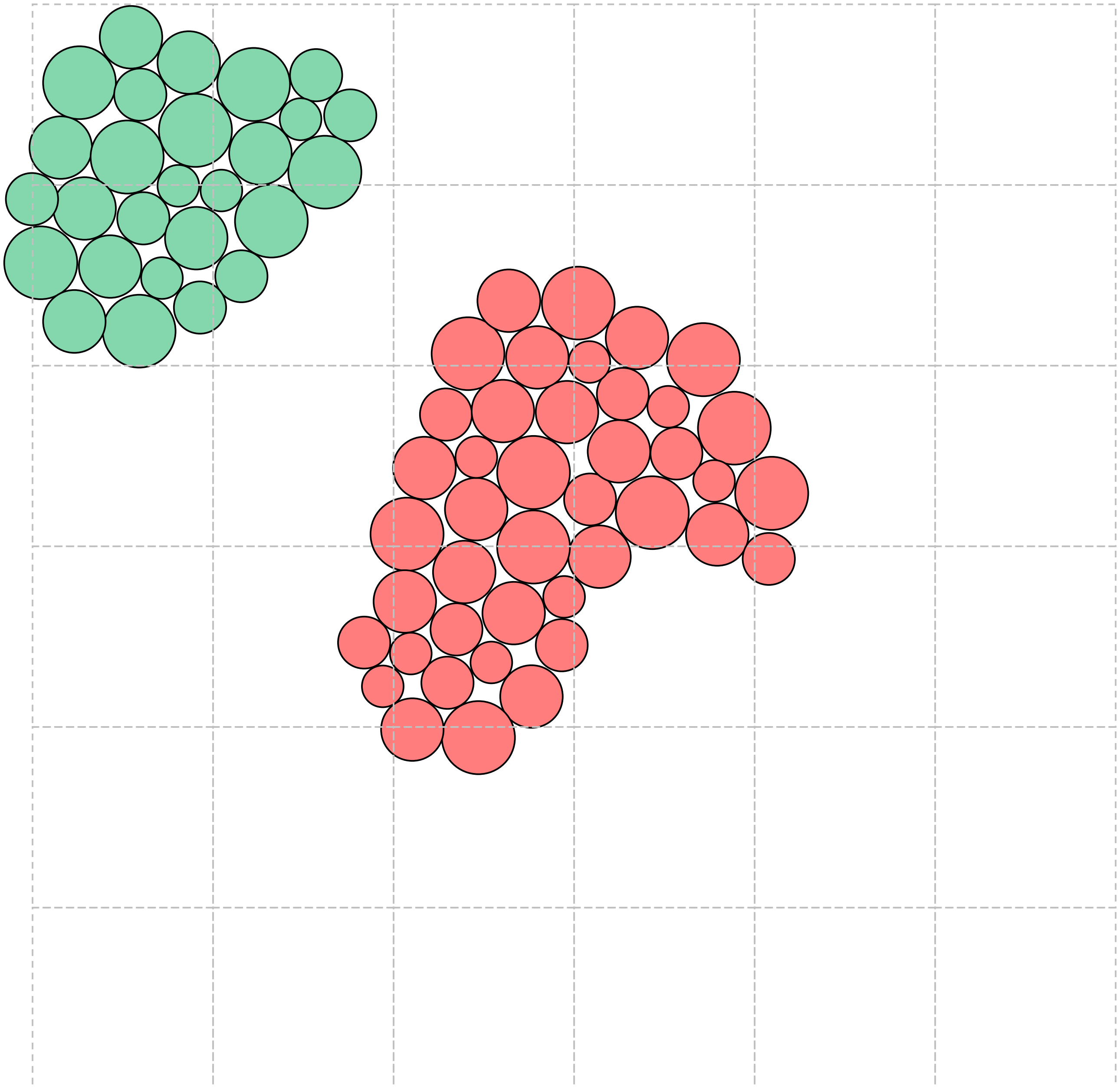
$$Re\left(F_{A,B}(t,r)\right) = \text{skin-skin overlap score} - \text{core-core overlap score}$$

$$Im\left(F_{A,B}(t,r)\right) = \text{skin-core overlap score}$$

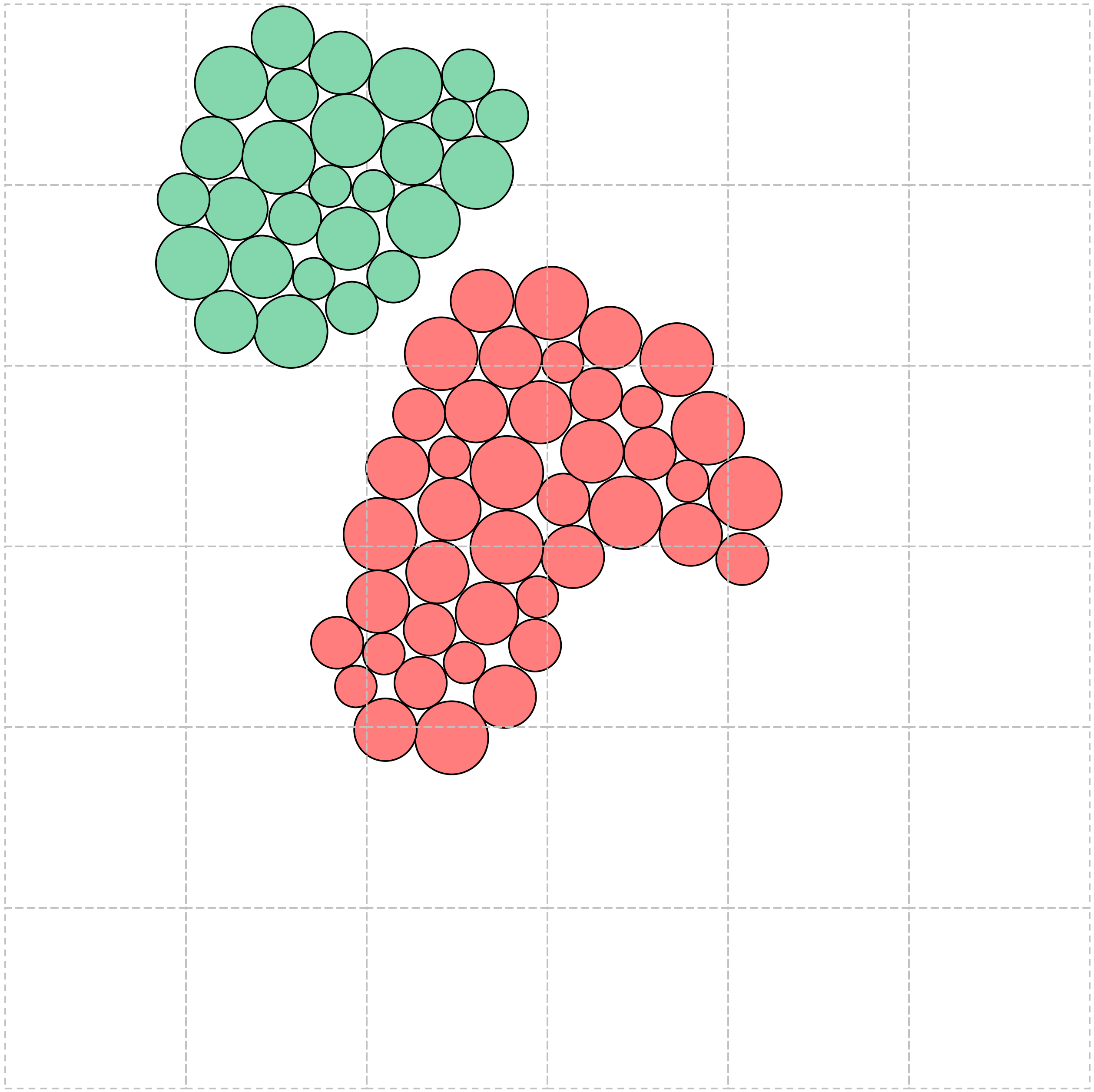
Docking: Rotational & Translational Search



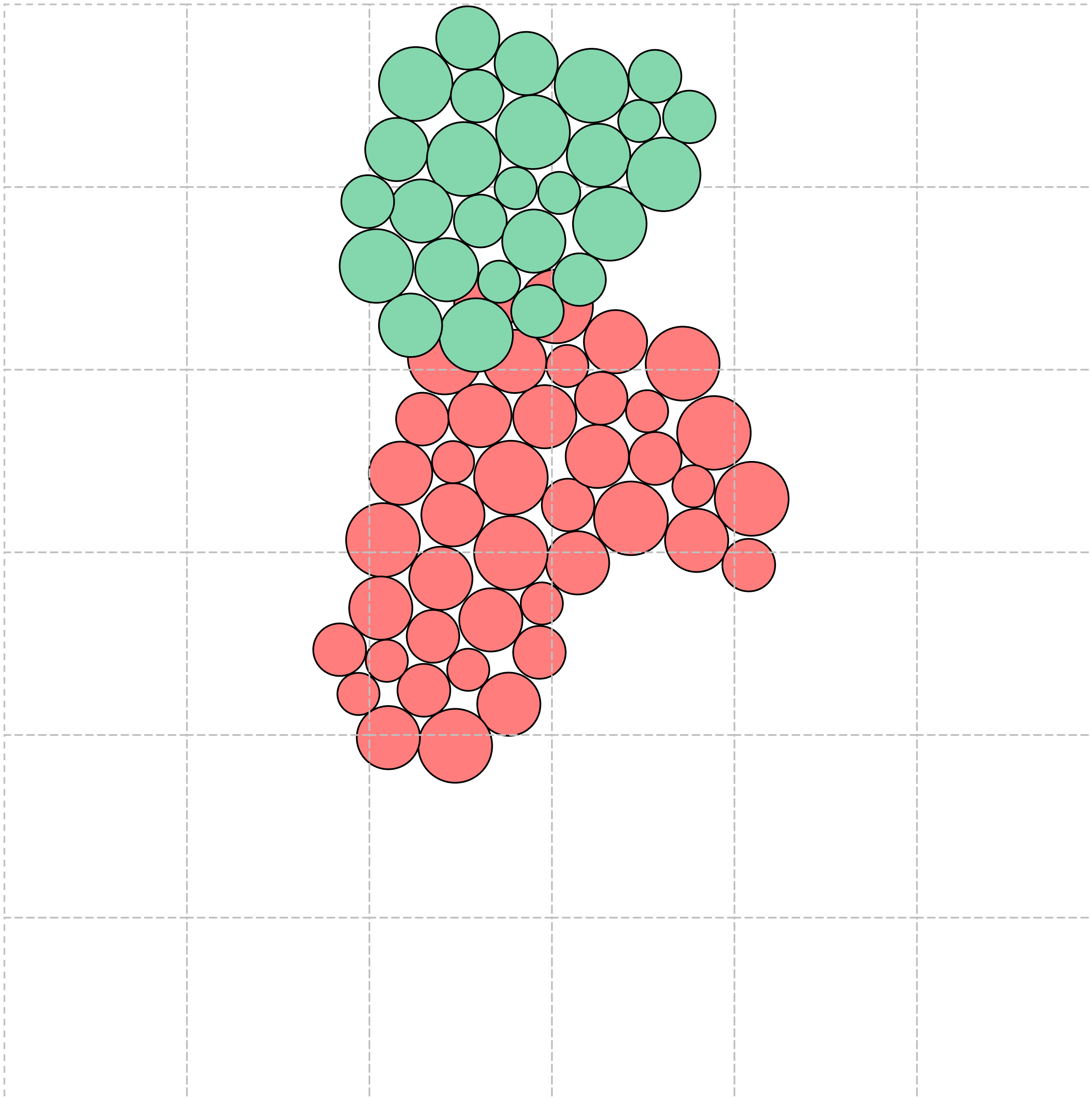
Docking: Rotational & Translational Search



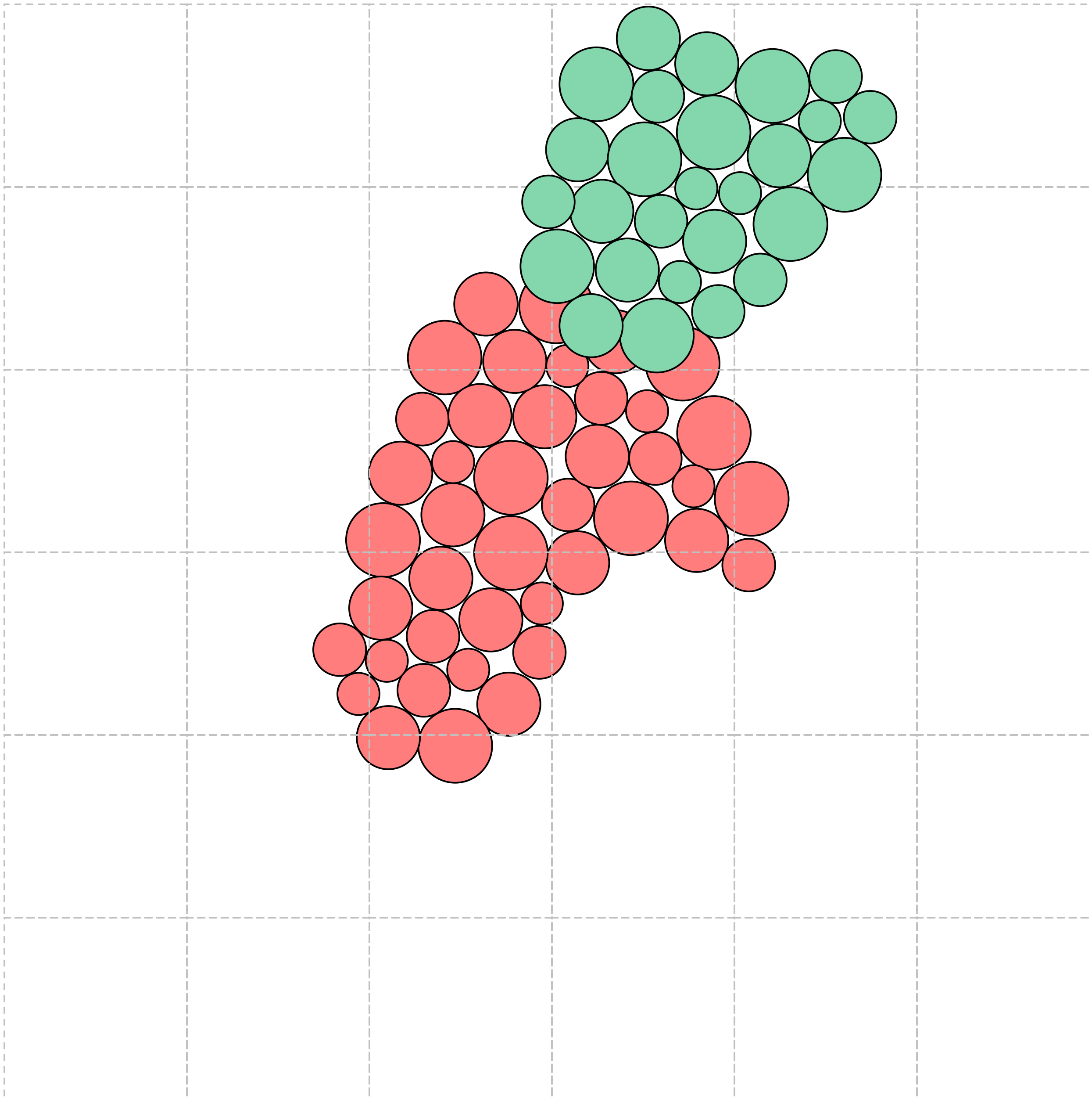
Docking: Rotational & Translational Search



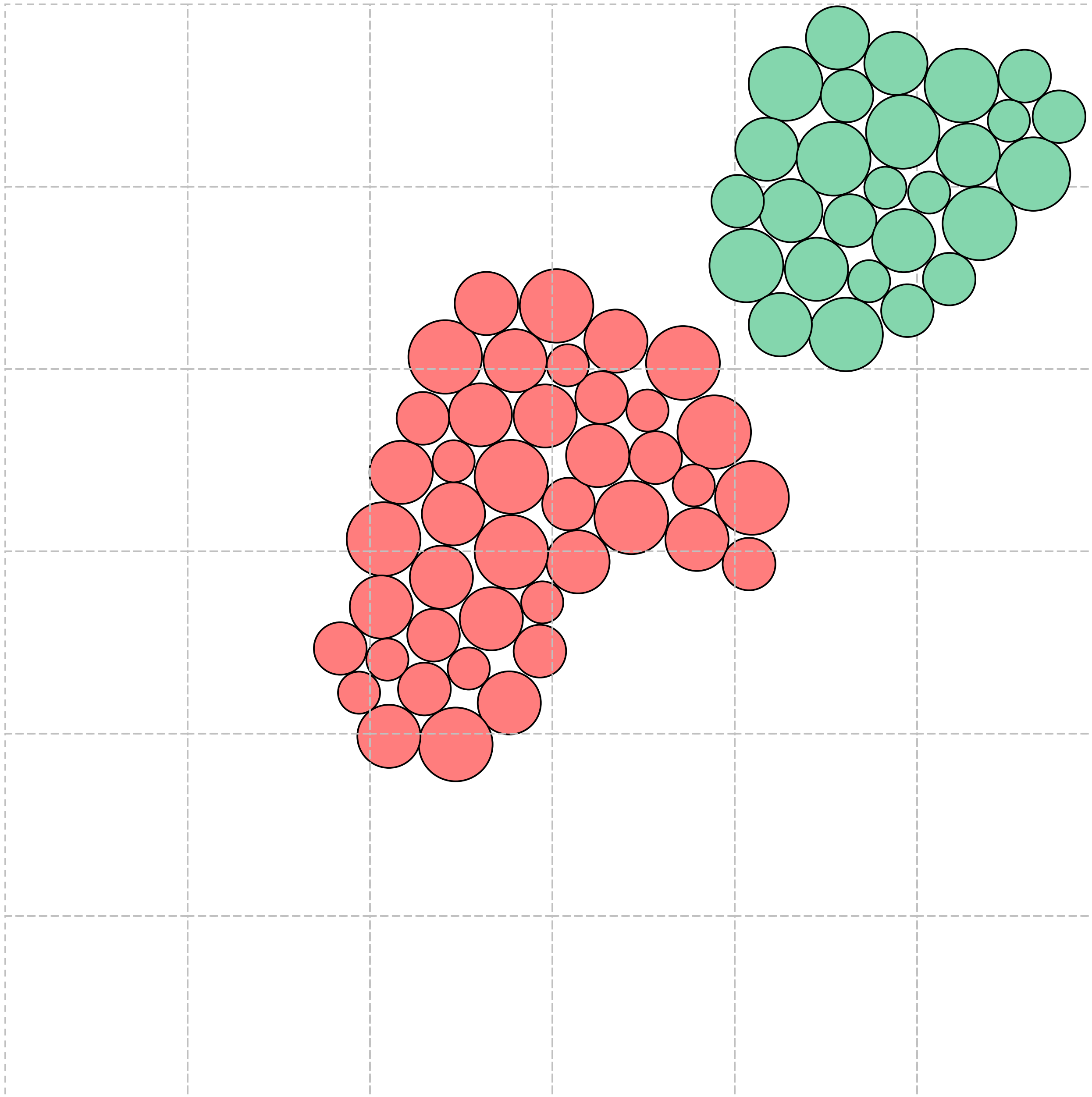
Docking: Rotational & Translational Search



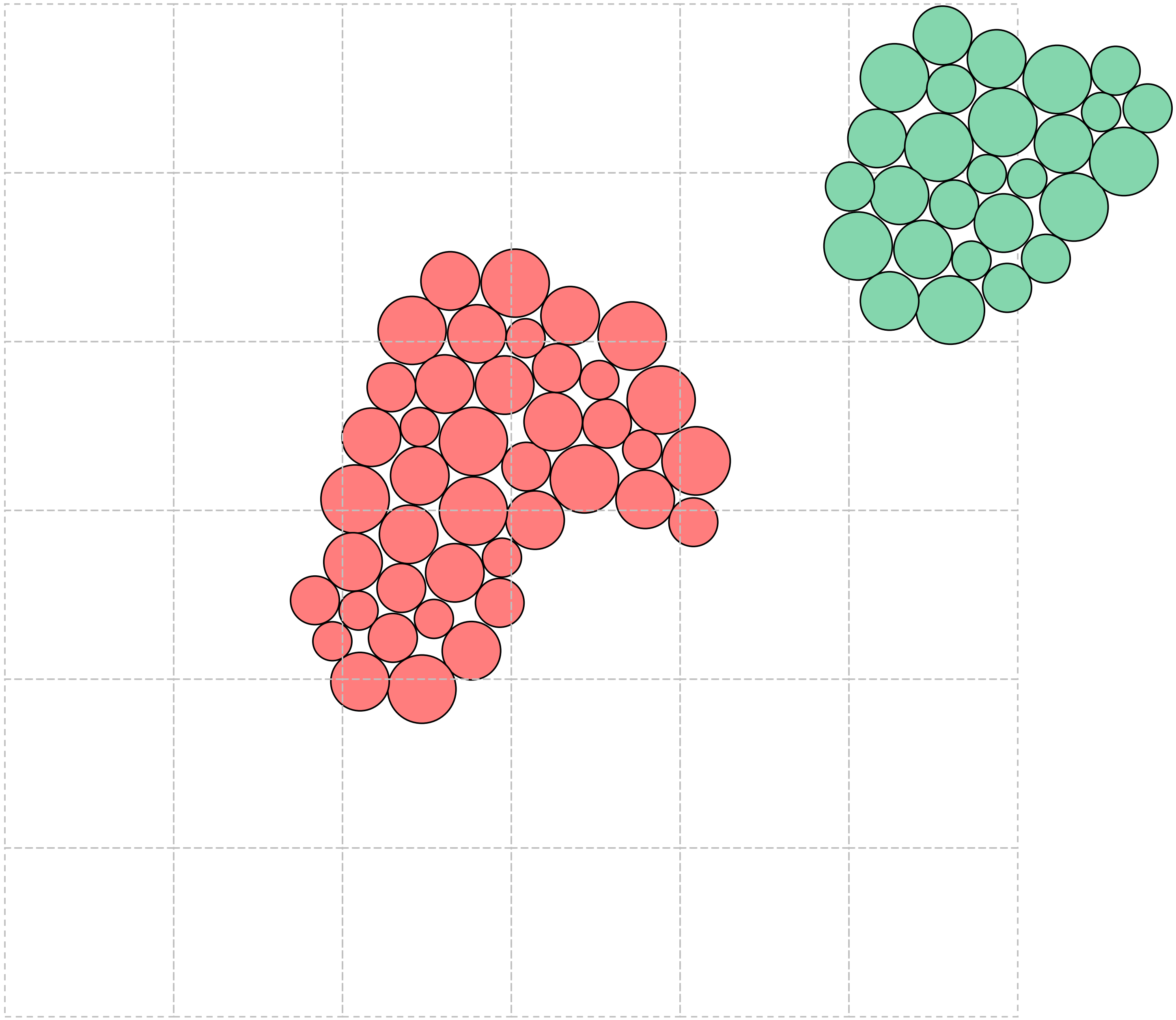
Docking: Rotational & Translational Search



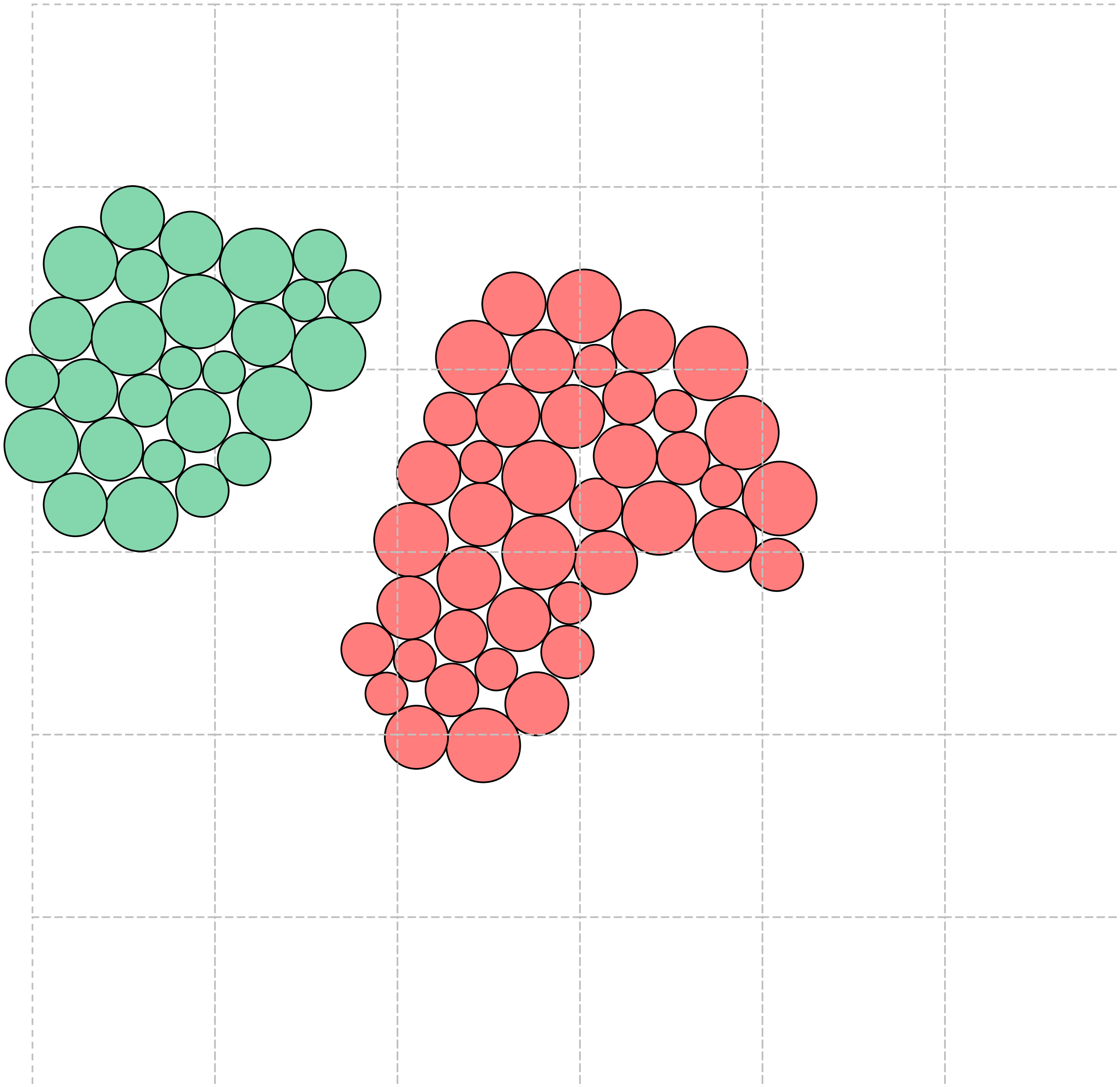
Docking: Rotational & Translational Search



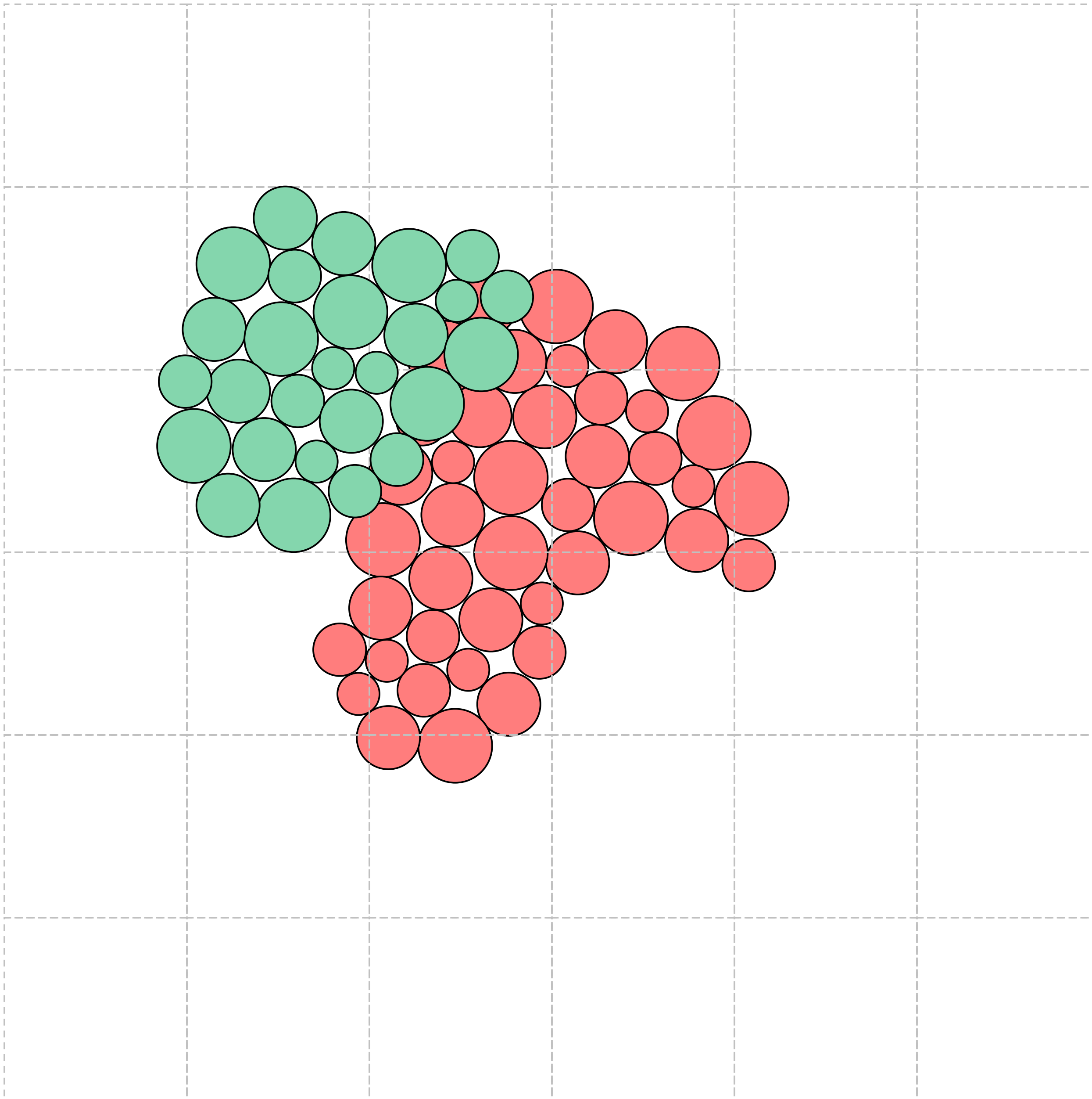
Docking: Rotational & Translational Search



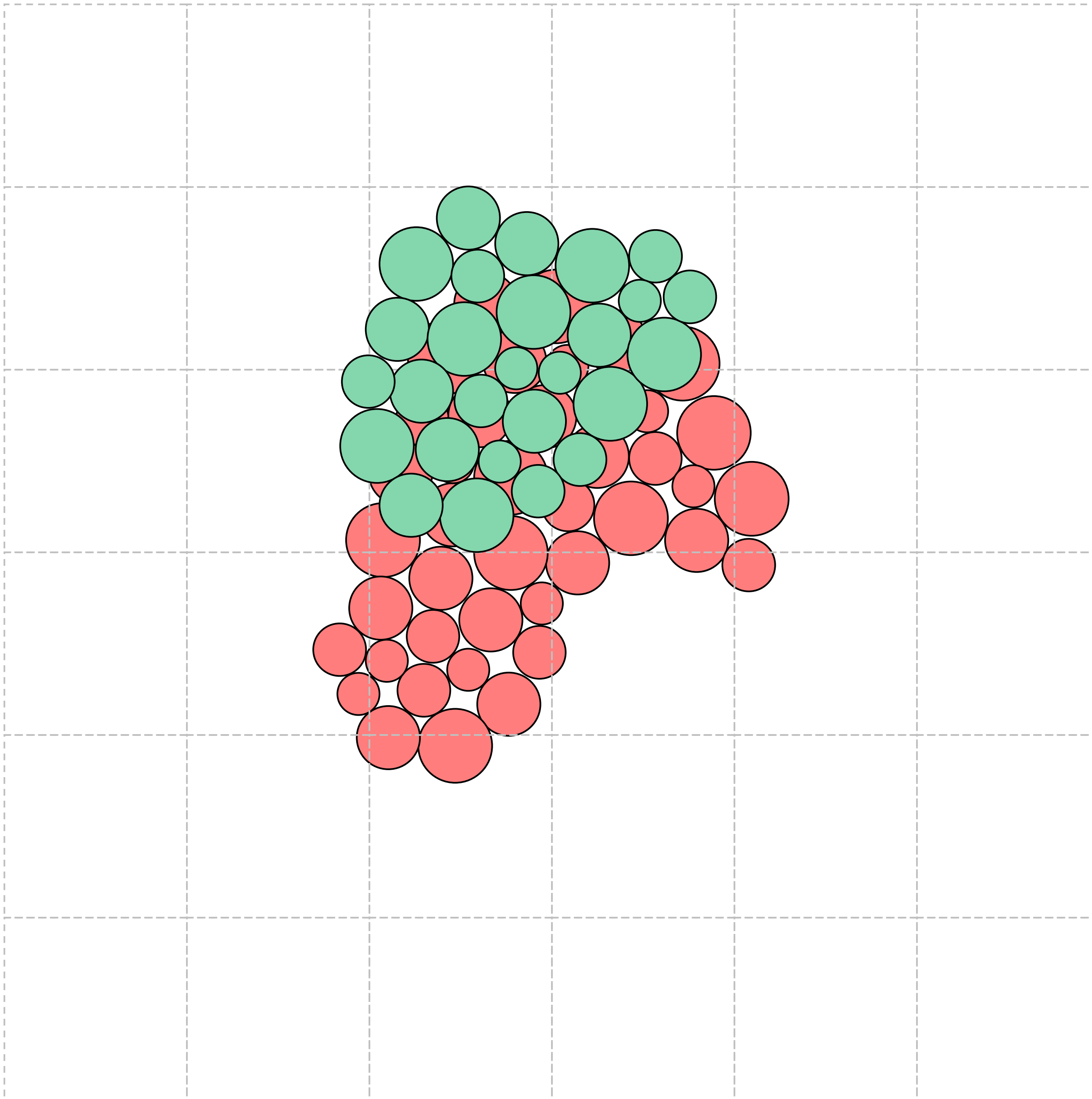
Docking: Rotational & Translational Search



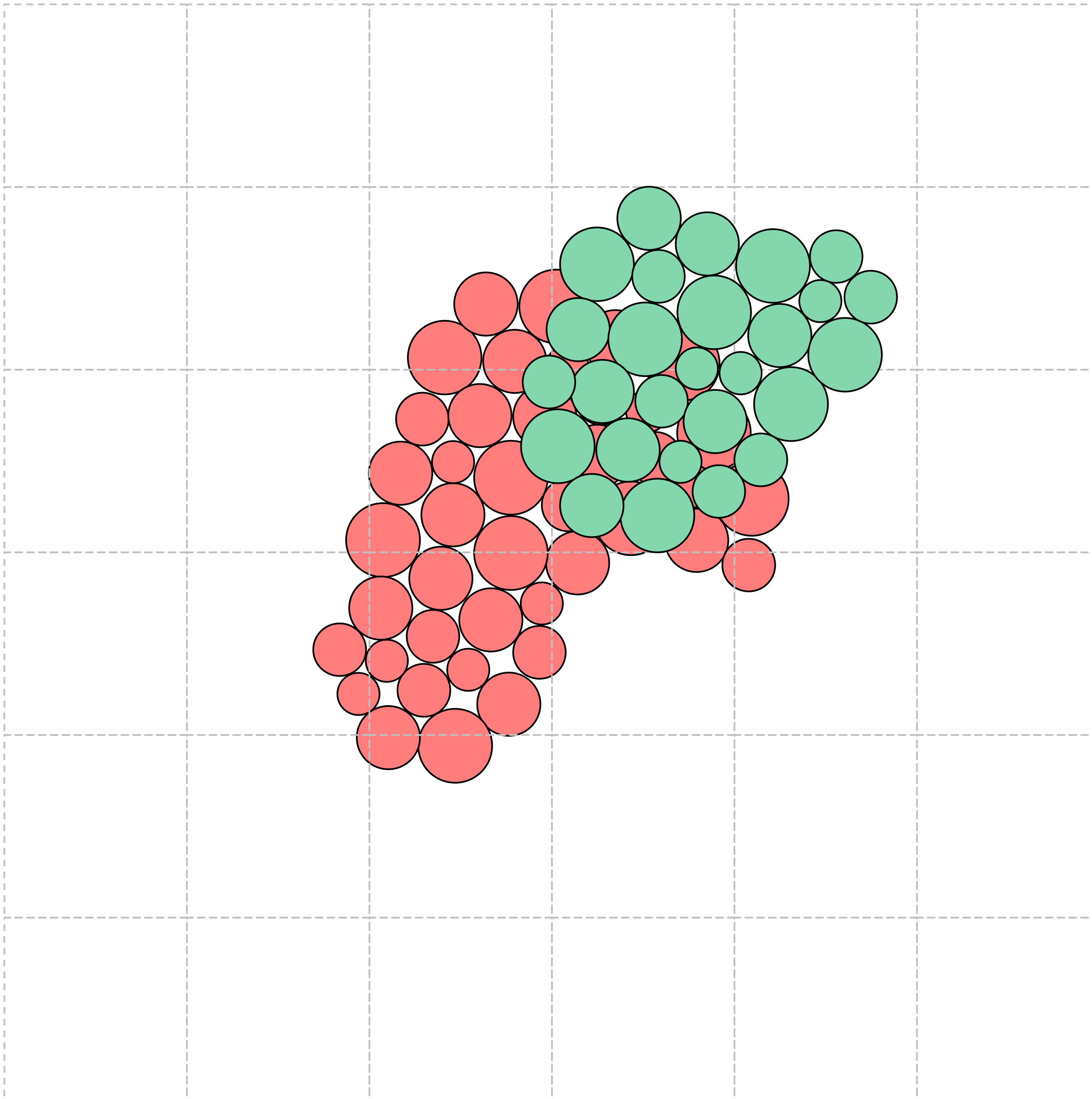
Docking: Rotational & Translational Search



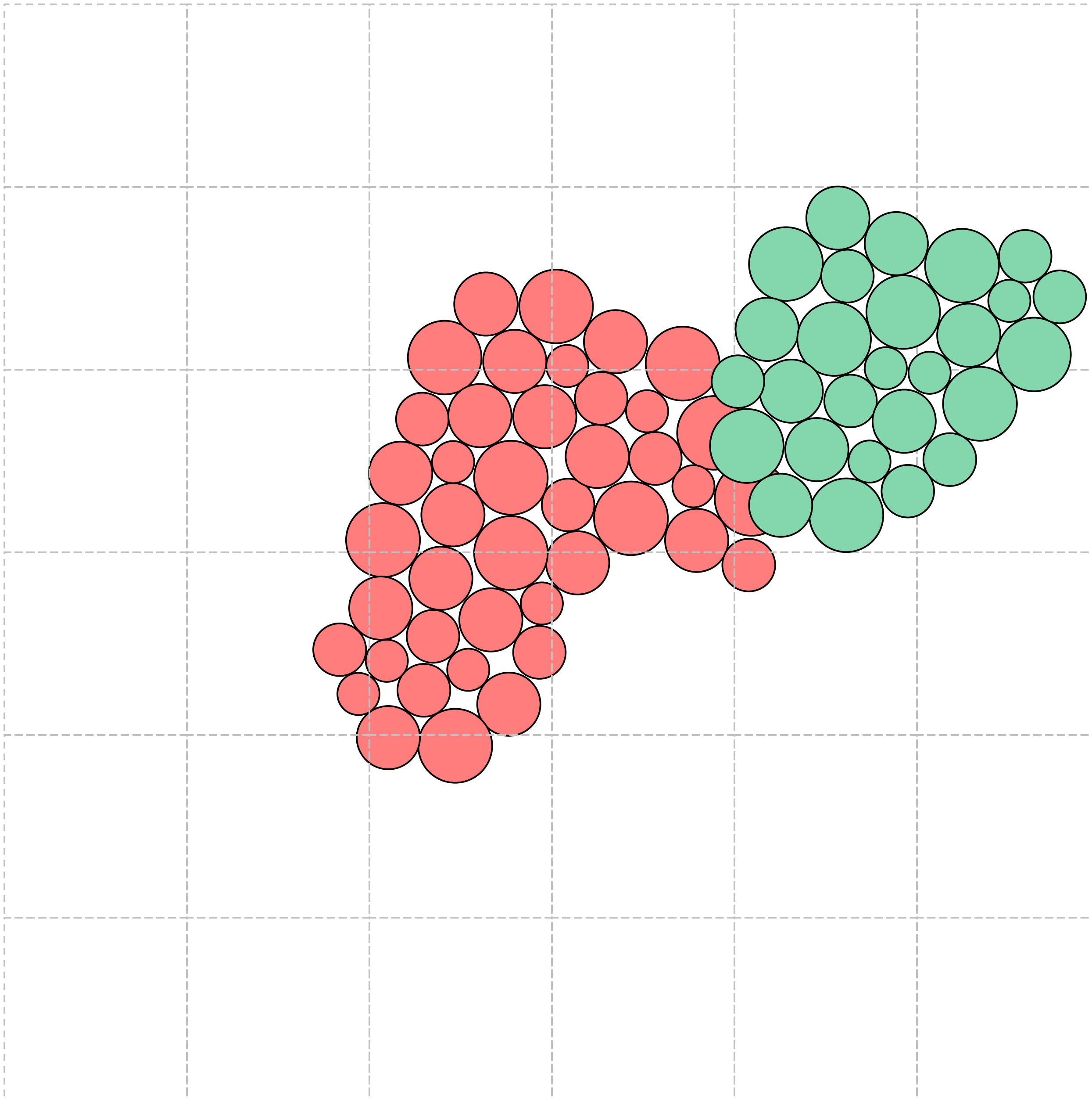
Docking: Rotational & Translational Search



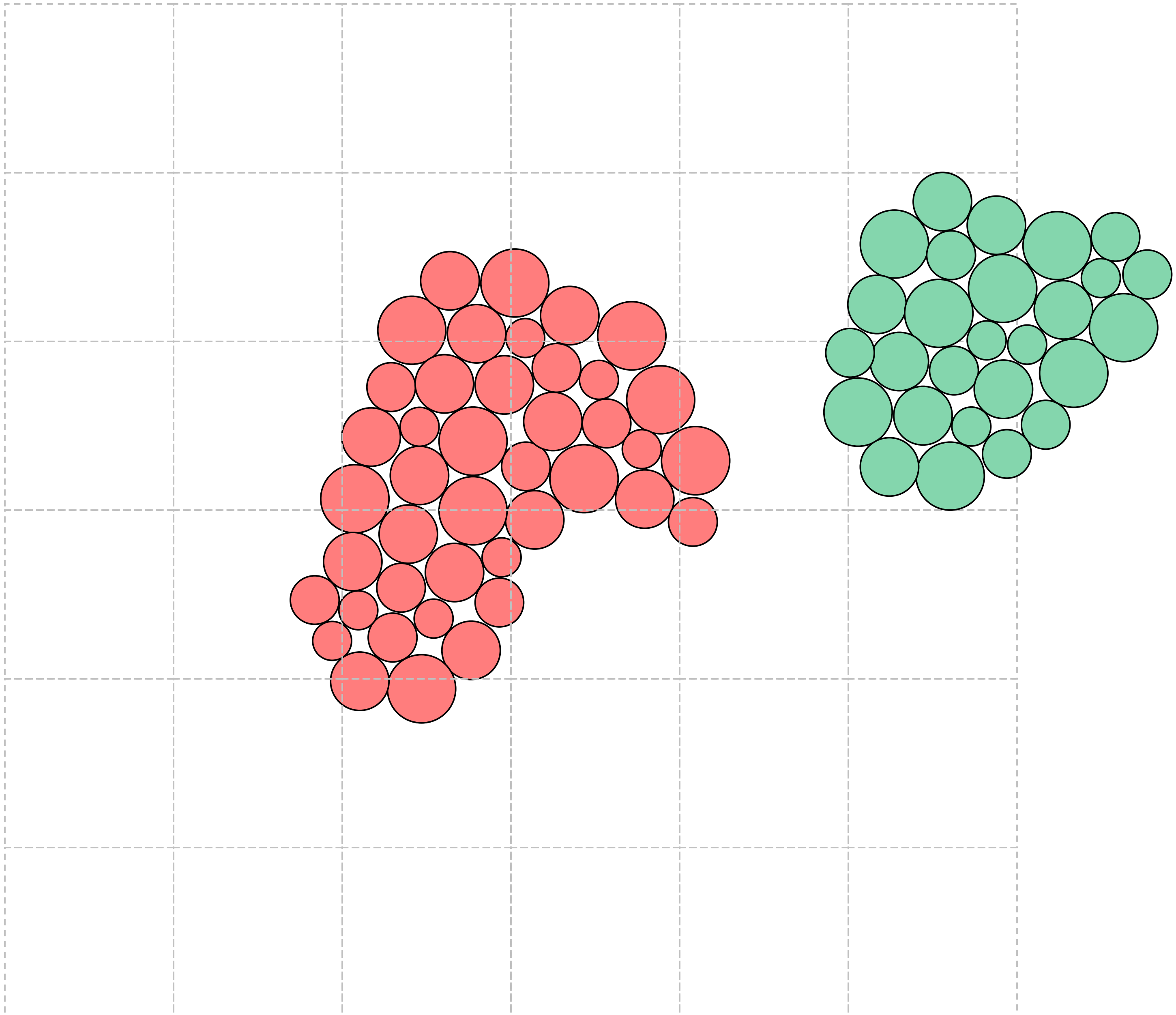
Docking: Rotational & Translational Search



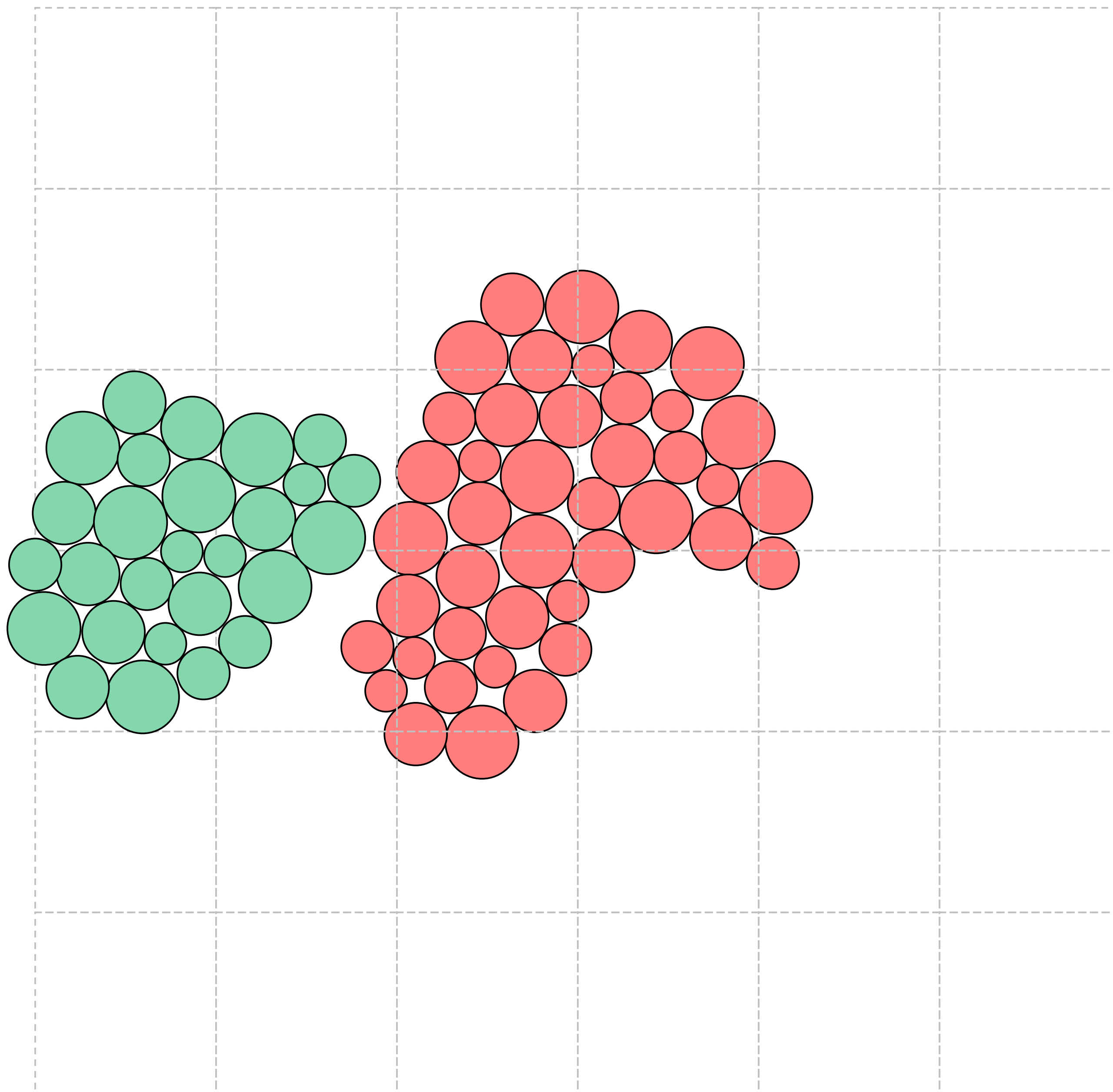
Docking: Rotational & Translational Search



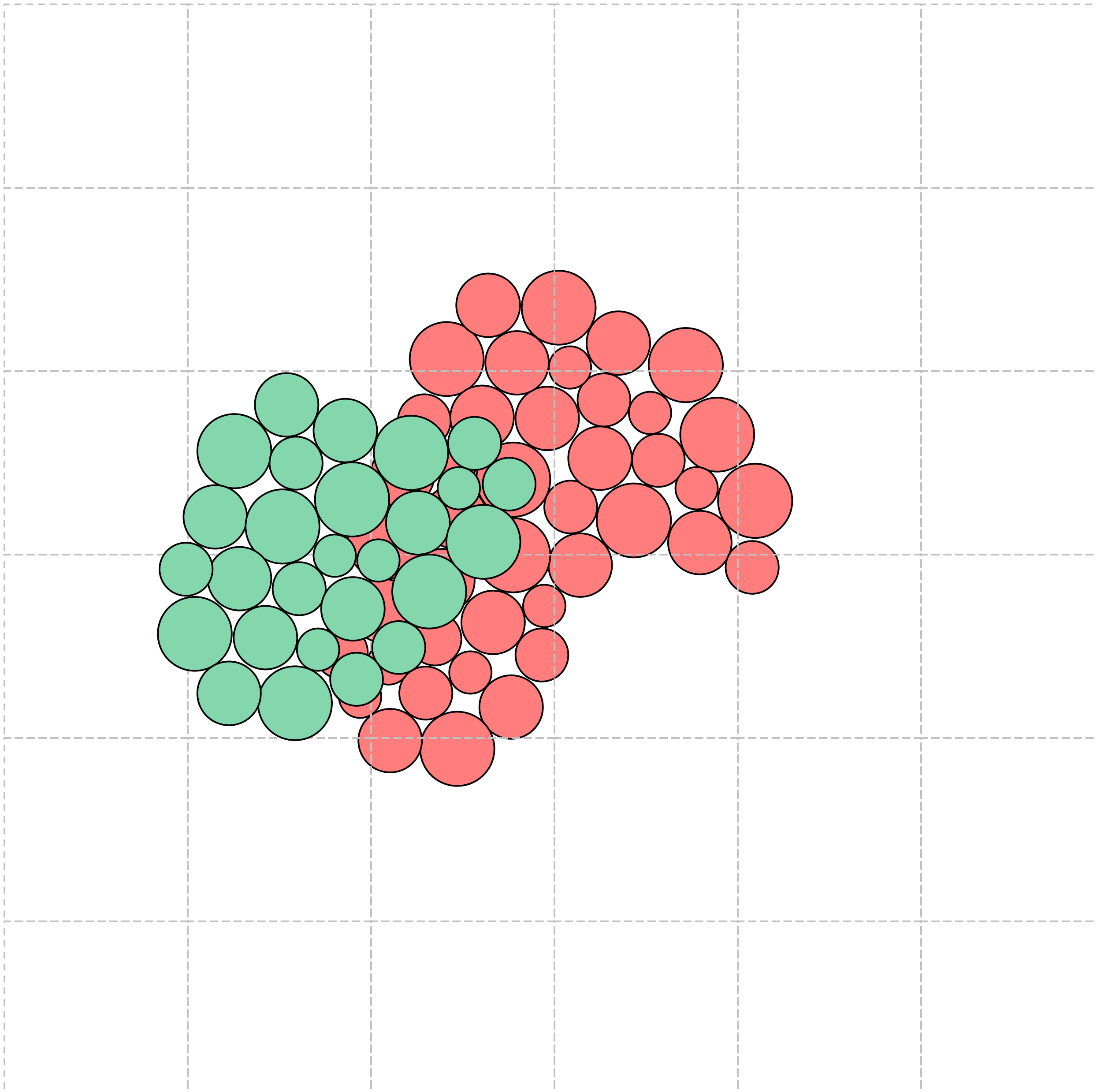
Docking: Rotational & Translational Search



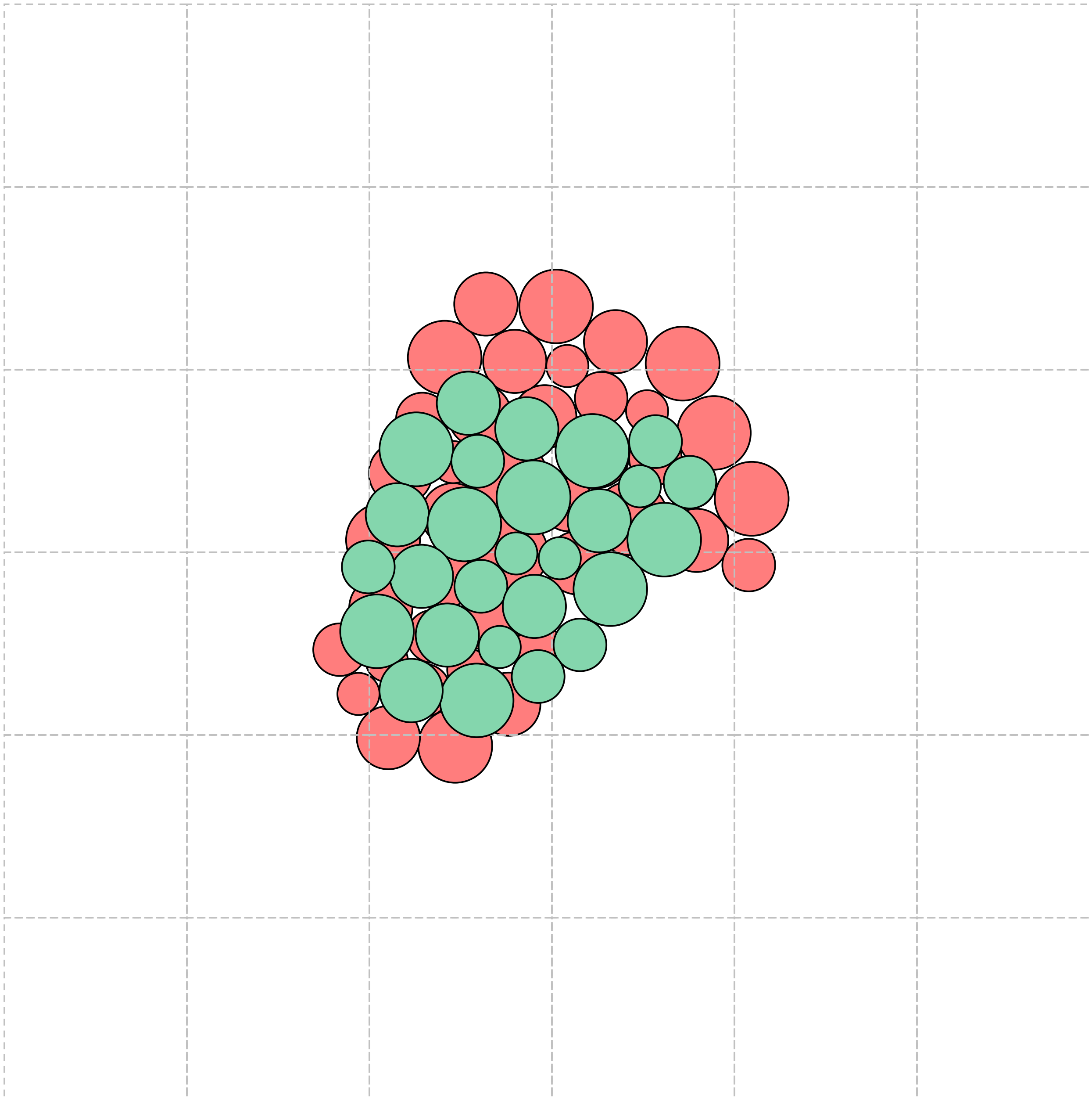
Docking: Rotational & Translational Search



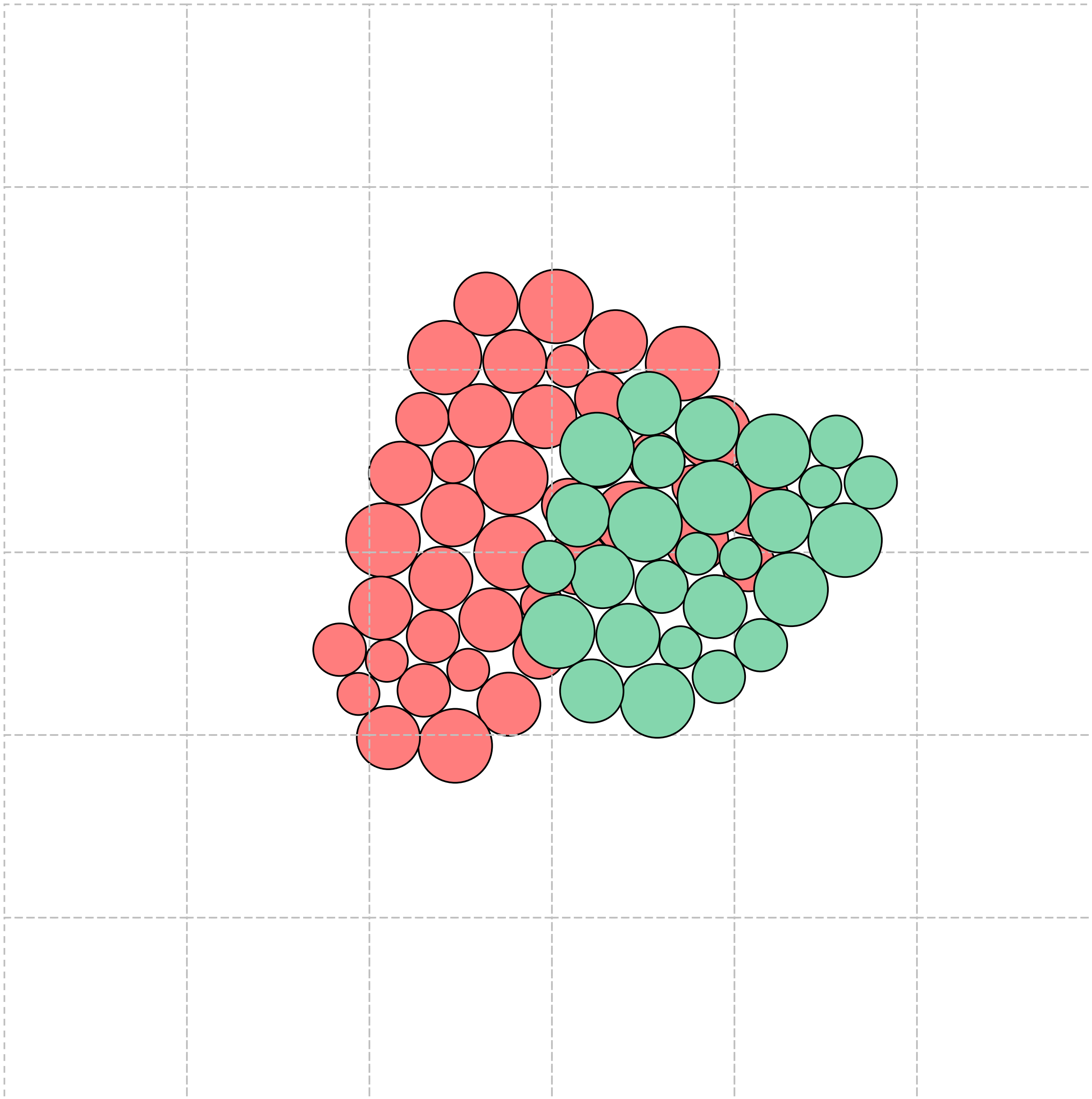
Docking: Rotational & Translational Search



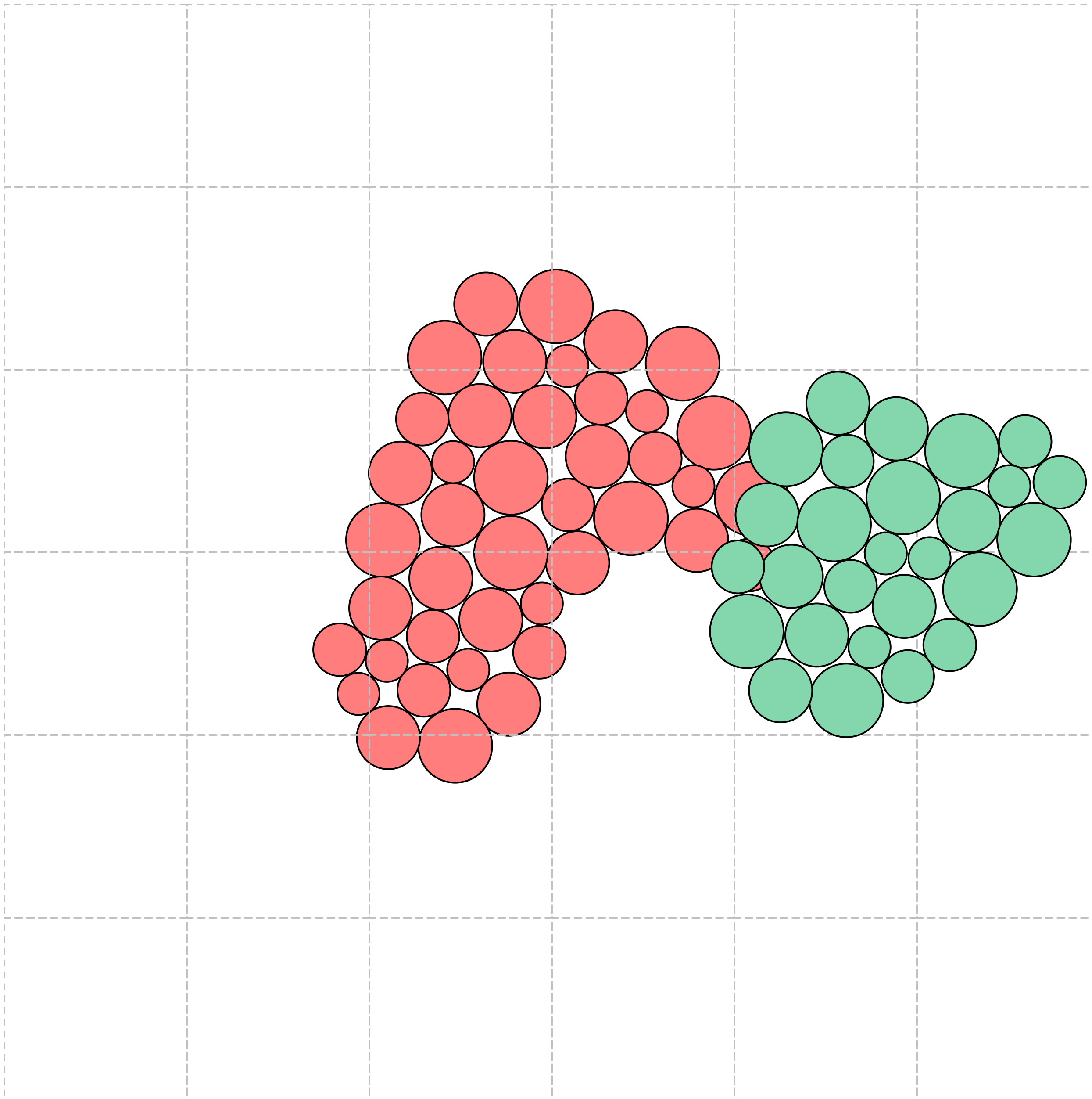
Docking: Rotational & Translational Search



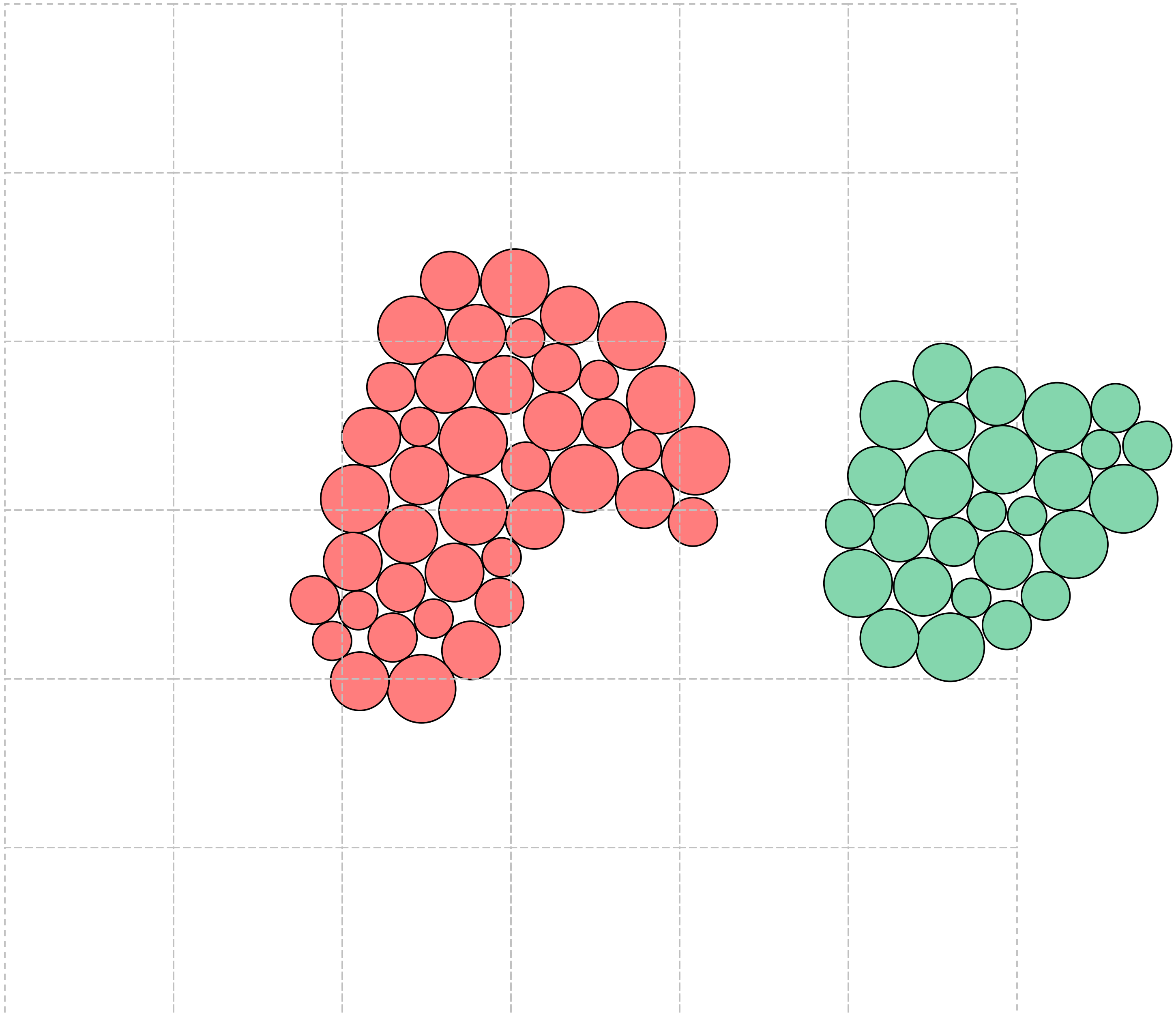
Docking: Rotational & Translational Search



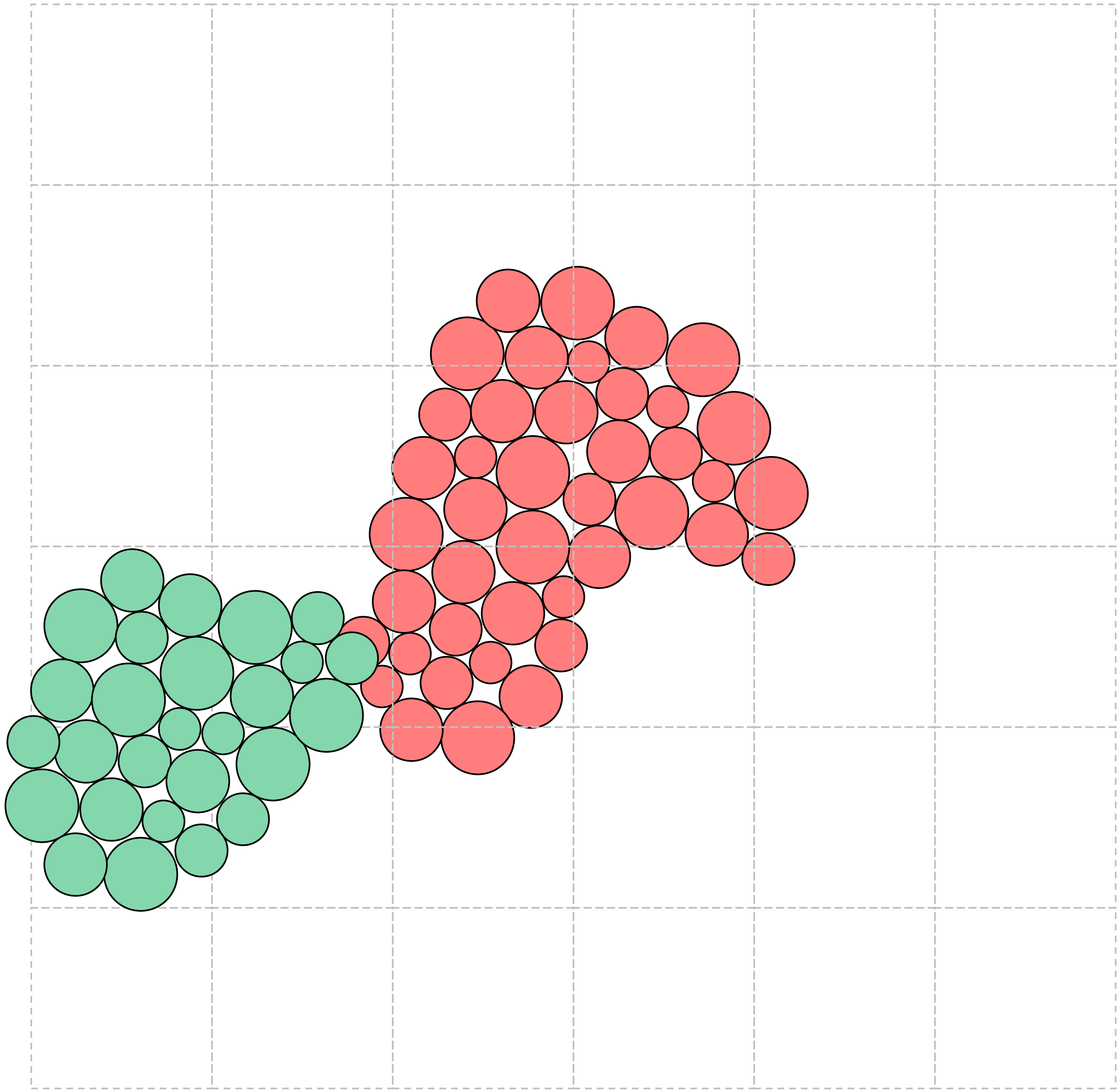
Docking: Rotational & Translational Search



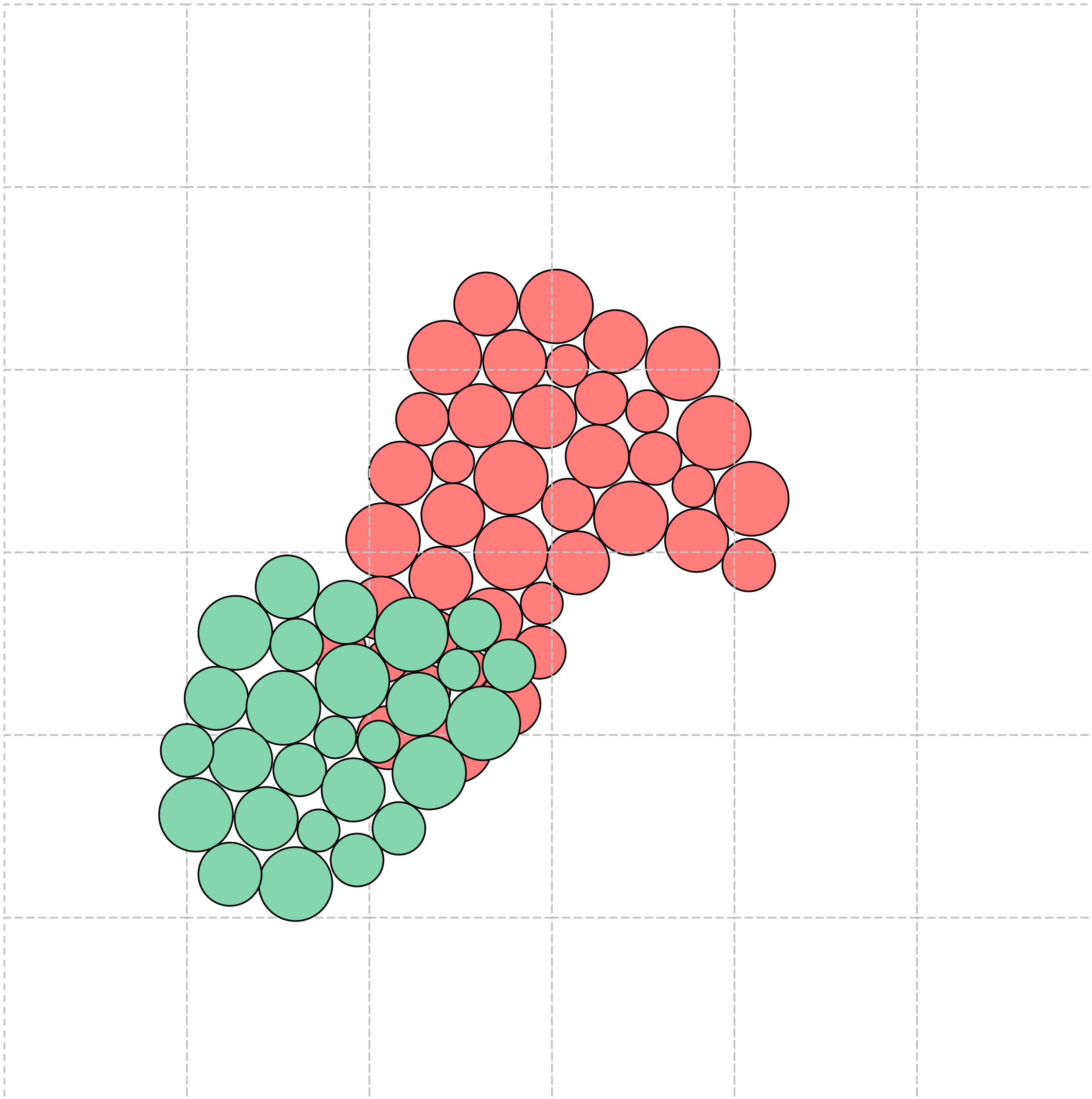
Docking: Rotational & Translational Search



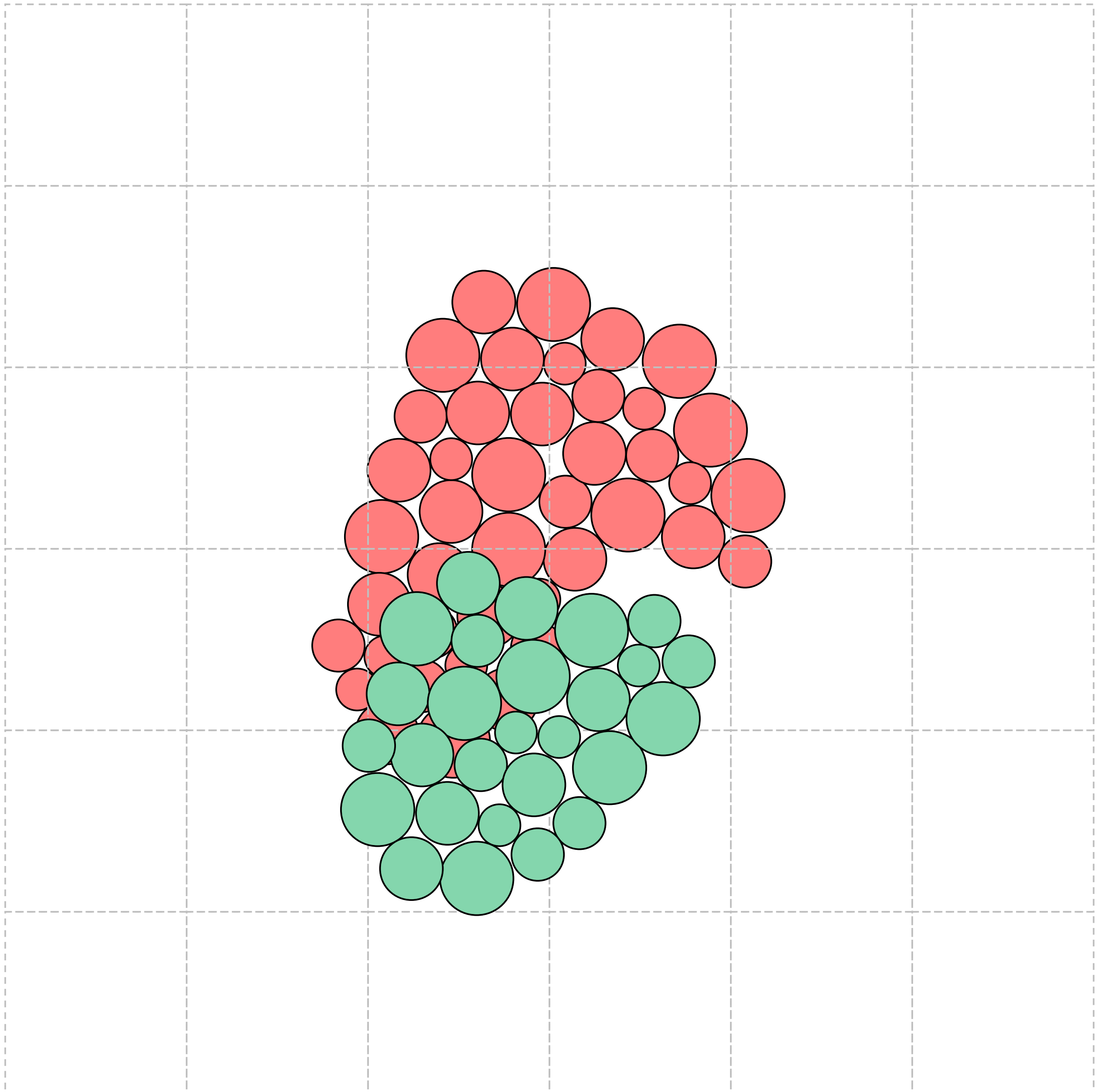
Docking: Rotational & Translational Search



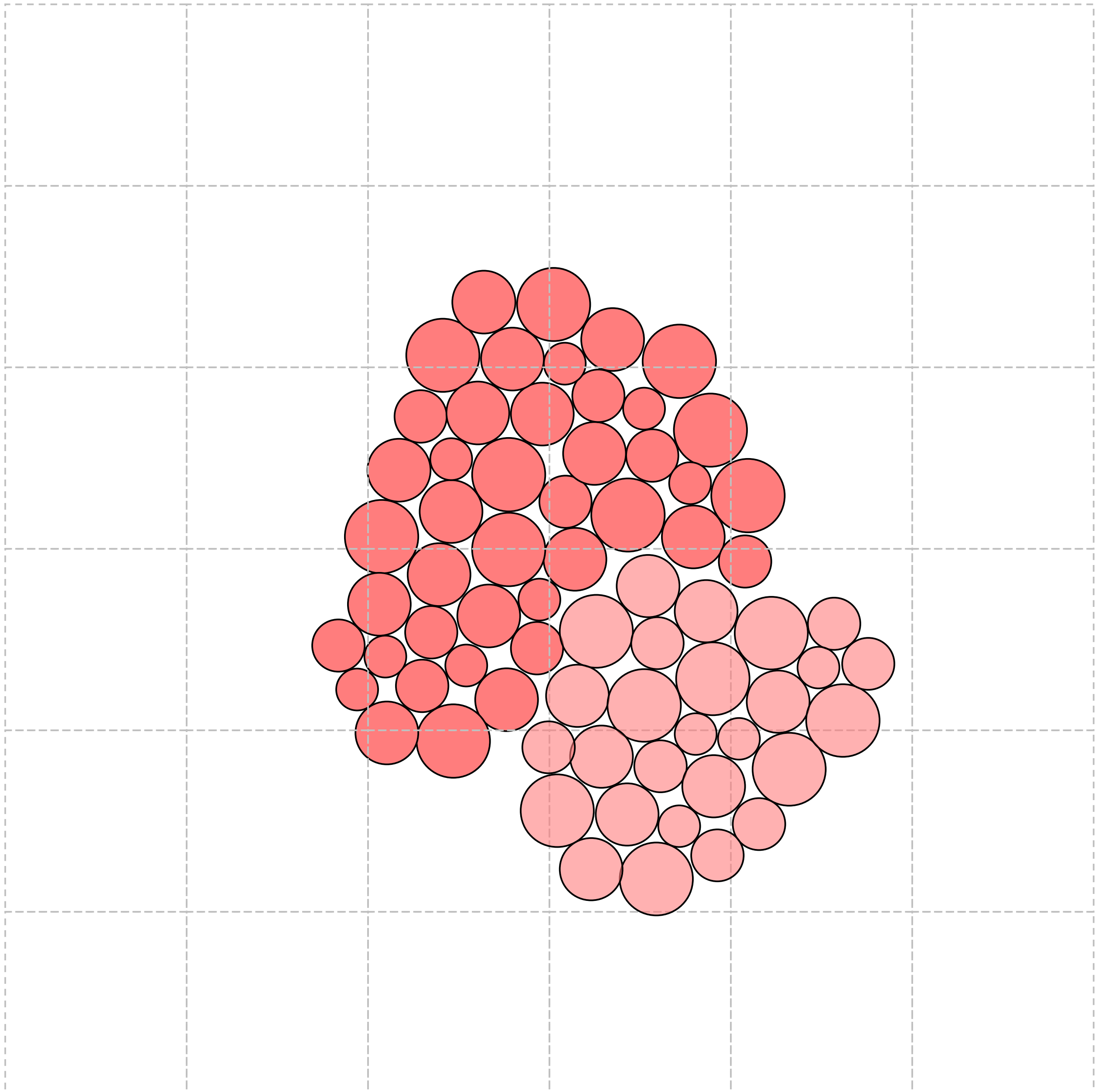
Docking: Rotational & Translational Search



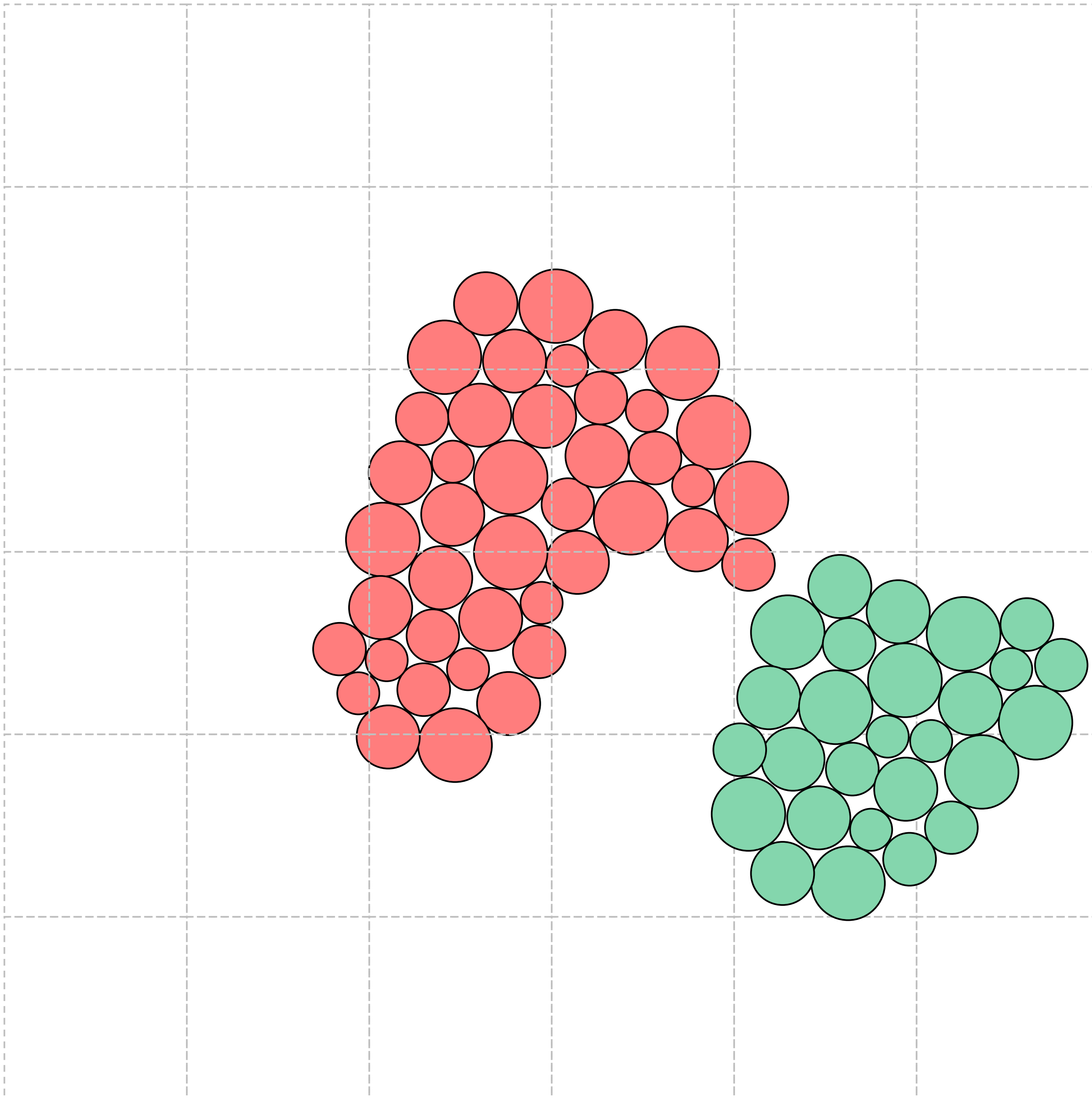
Docking: Rotational & Translational Search



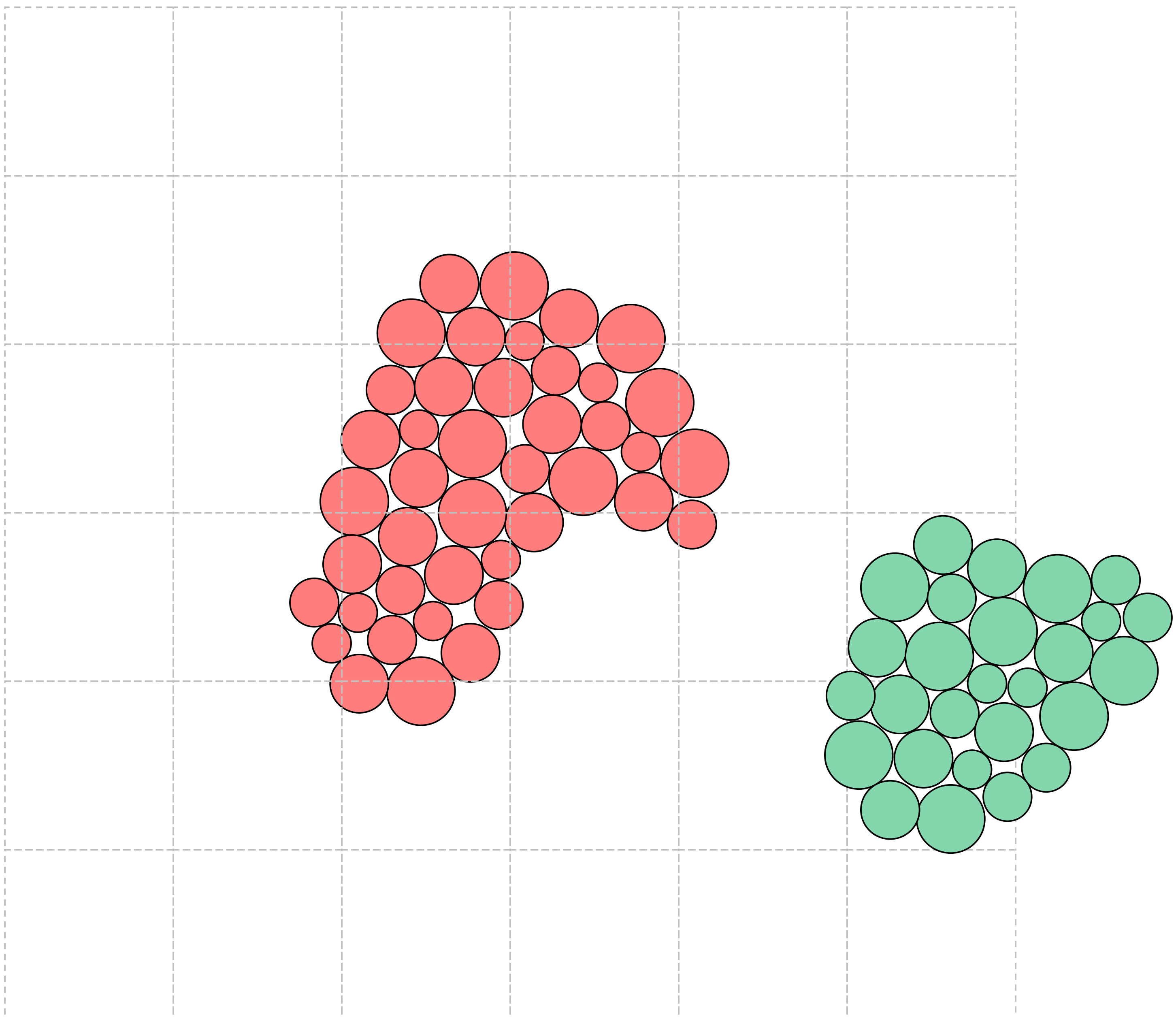
Docking: Rotational & Translational Search



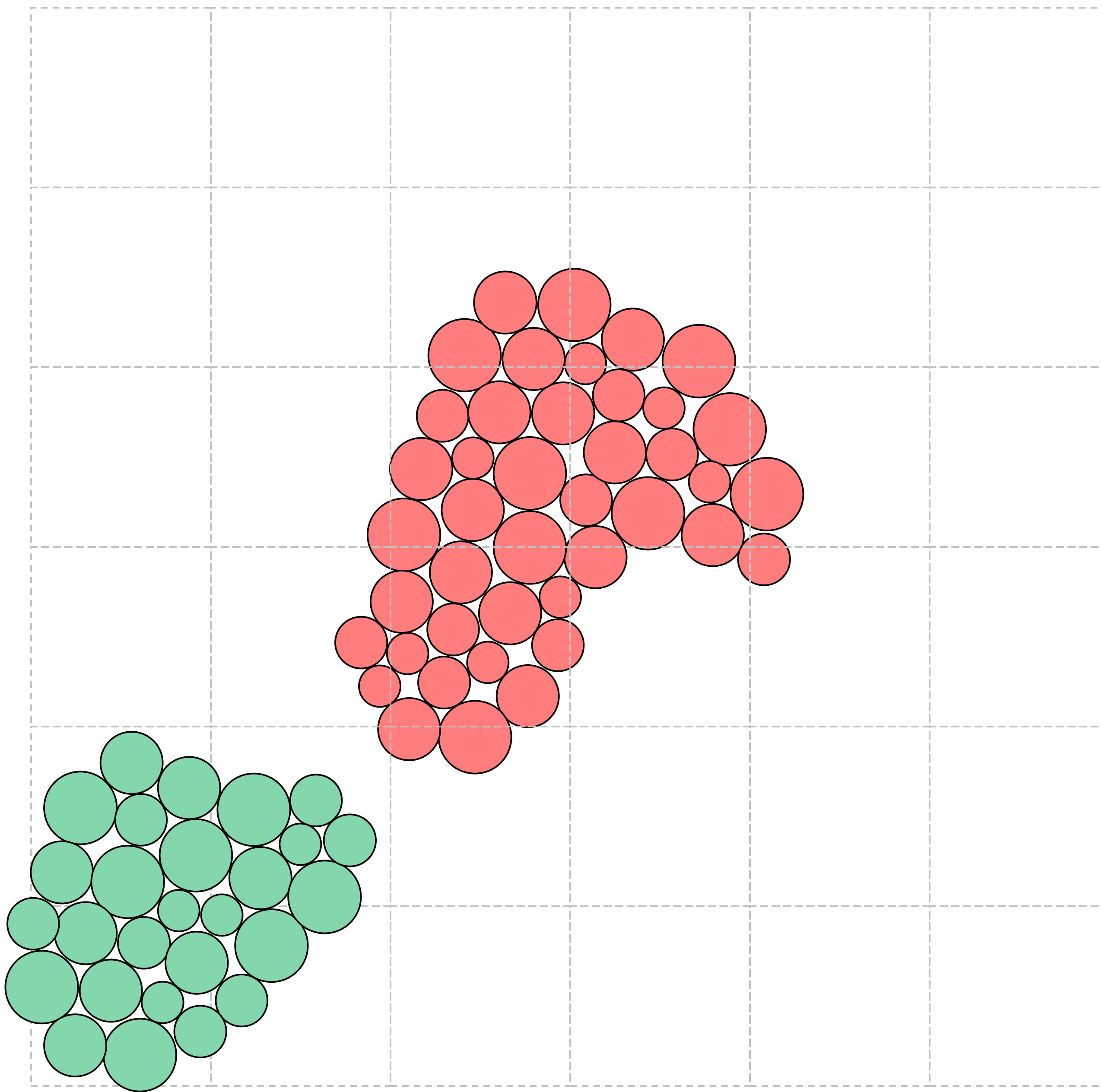
Docking: Rotational & Translational Search



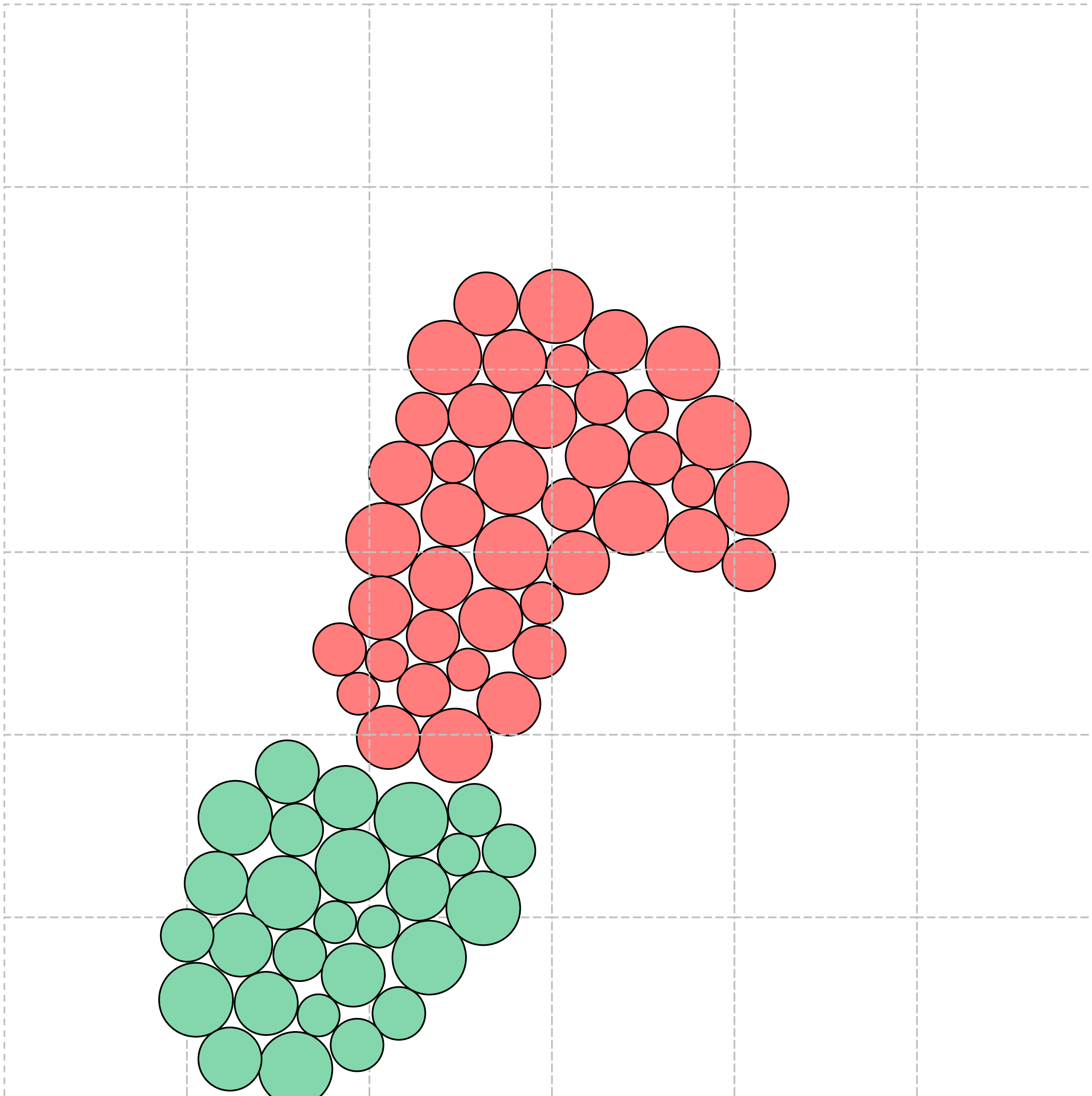
Docking: Rotational & Translational Search



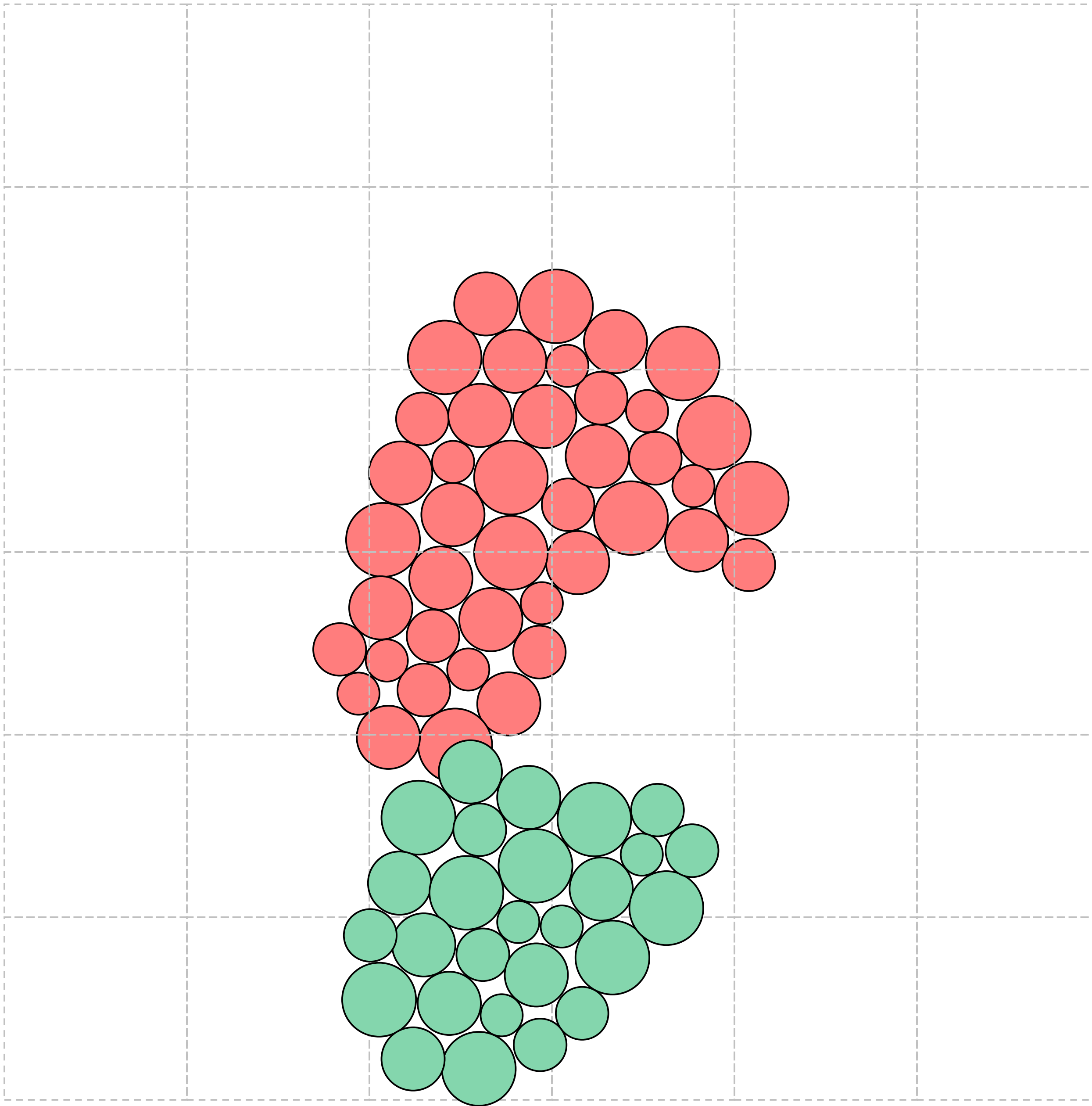
Docking: Rotational & Translational Search



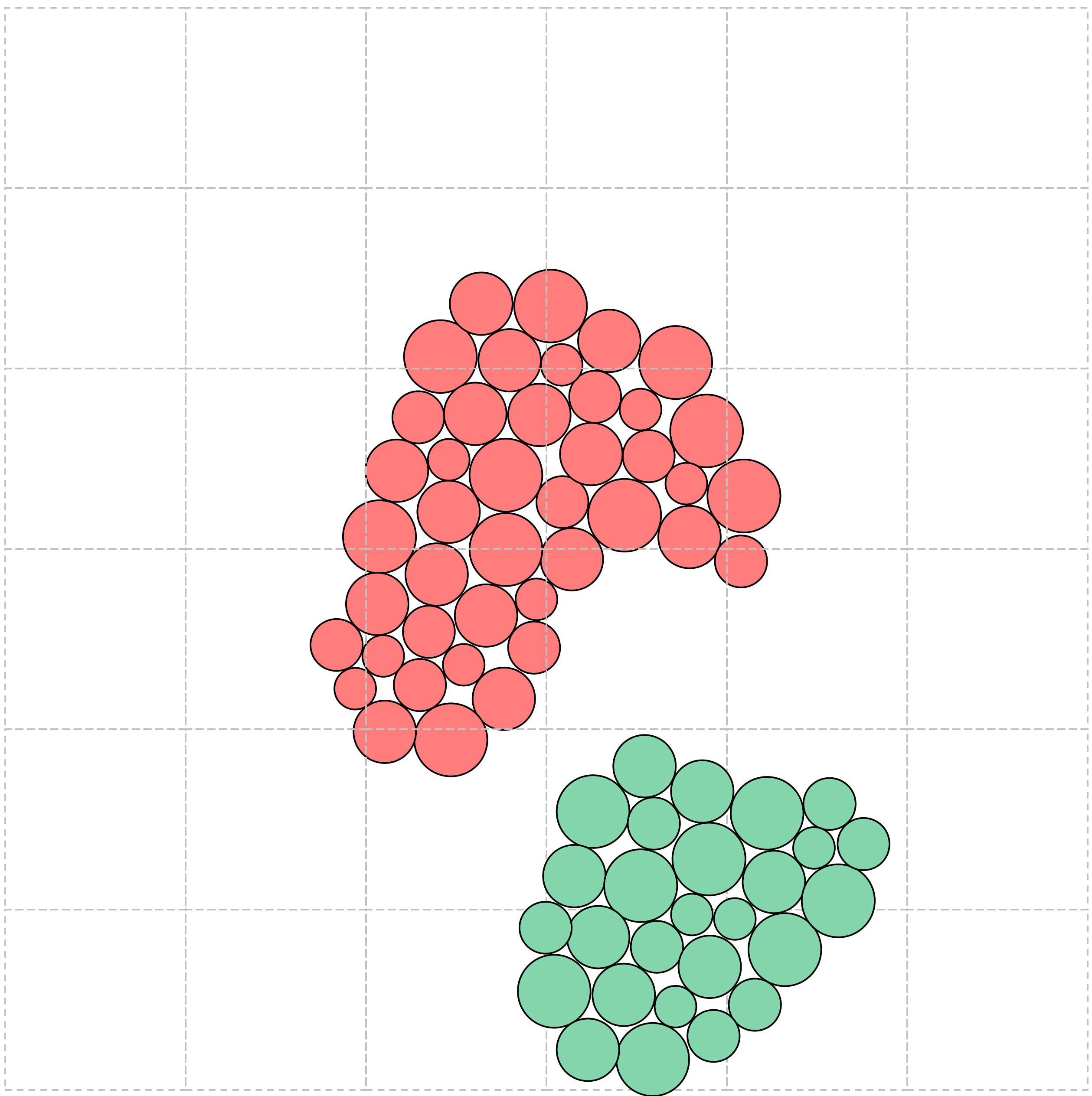
Docking: Rotational & Translational Search



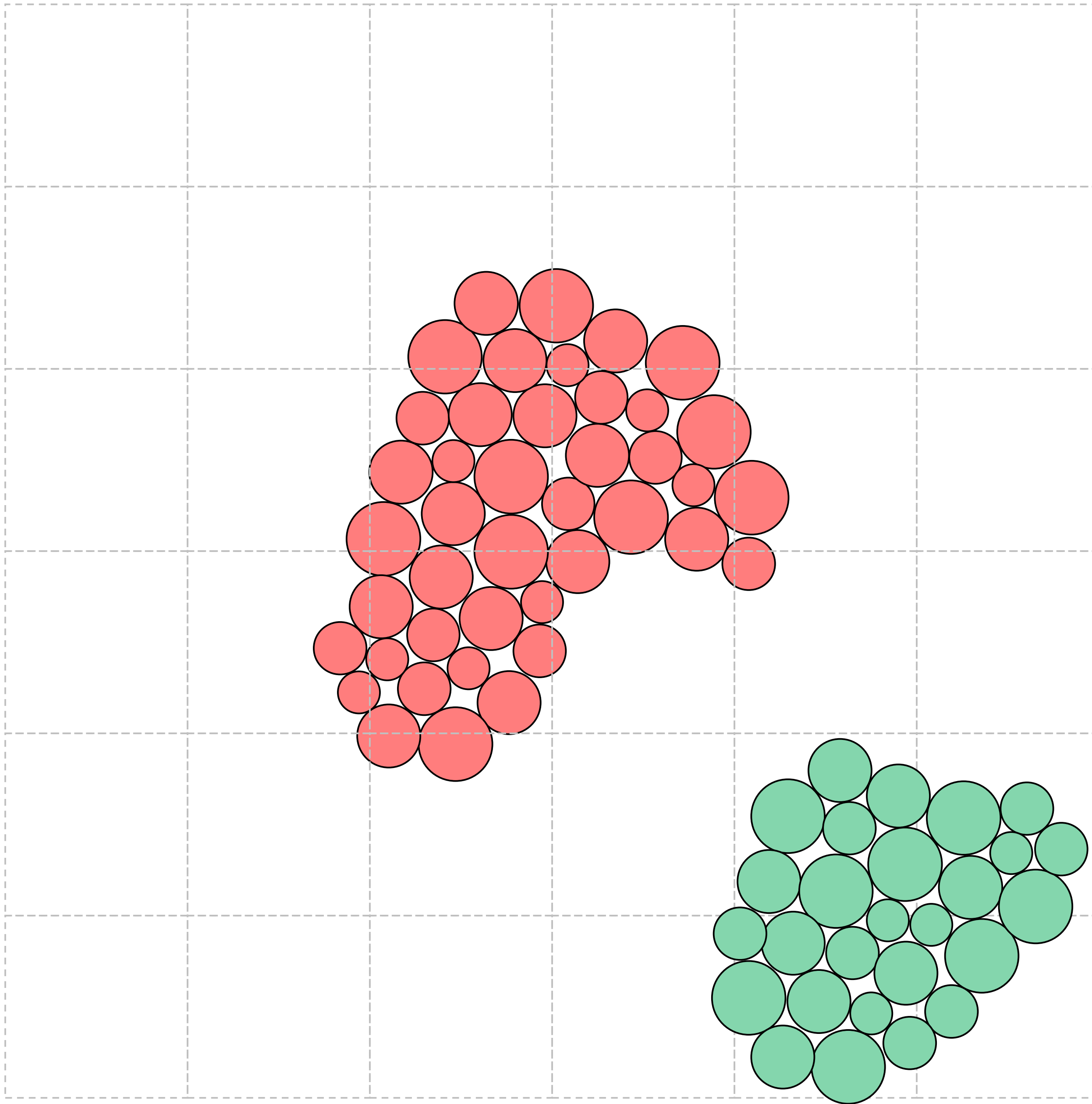
Docking: Rotational & Translational Search



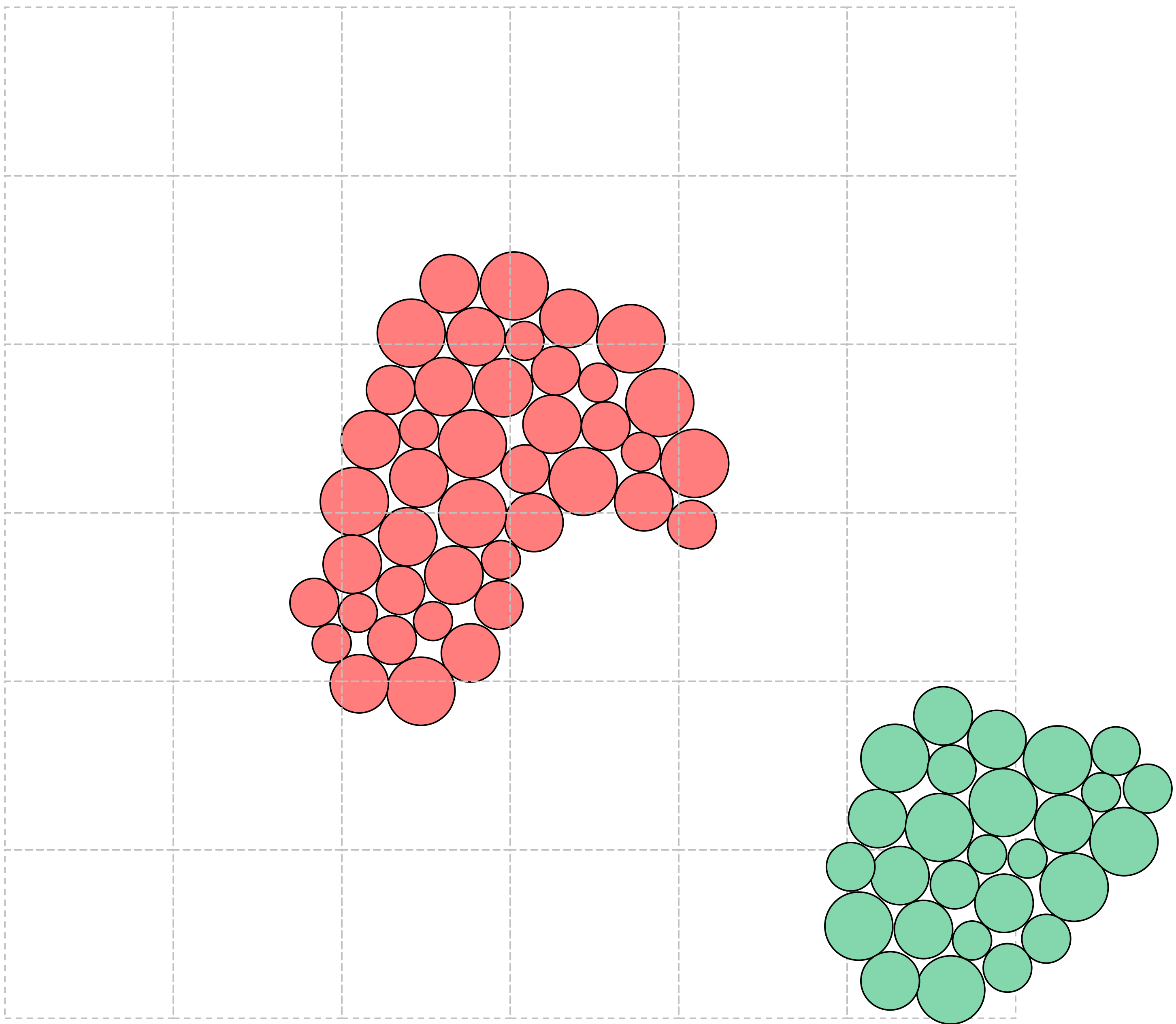
Docking: Rotational & Translational Search



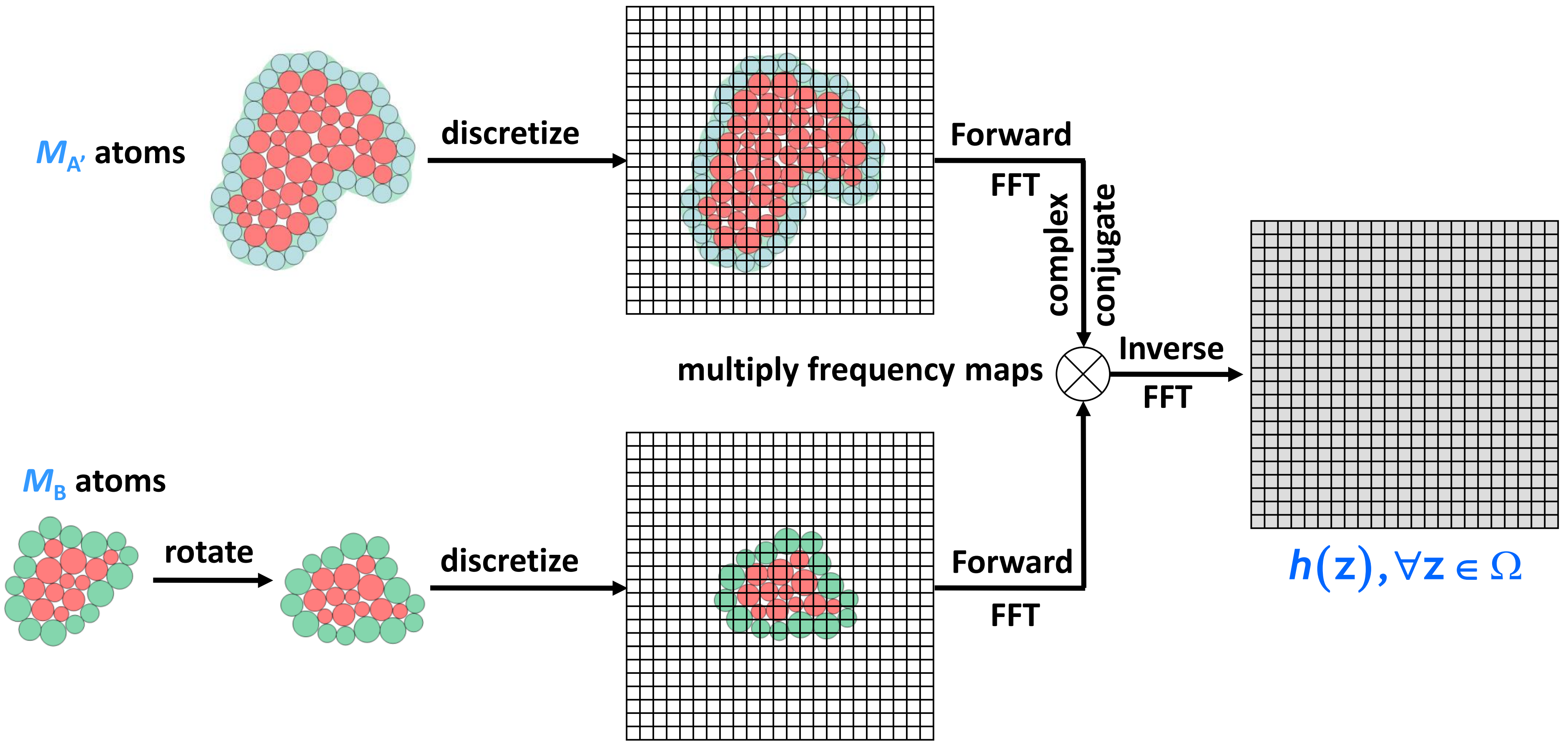
Docking: Rotational & Translational Search



Docking: Rotational & Translational Search



Translational Search using FFT



$$\forall z \in \Omega = [-n, n]^3, \qquad h(z) = \int_{x \in \Omega} f_{A'}(x) f_{B_r}(z - x)\, dx$$