Domain-Heuristics for Arc-Consistency Algorithms

M.R.C. van Dongen (dongen@cs.ucc.ie)
"homepage:" http://www.cs.ucc.ie/~dongen

6 September, 2001

Outline

- Constraint Networks.
- Arc-Consistency.
- Case Study.
- Time-Complexity Results.
- Discussion and Future Work.

Constraint Networks

Let $D(x)$ denote the domain of the variable x.

C_S is called a *constraint* on S if $C_S \subseteq \times_{x \in S} D(x)$.

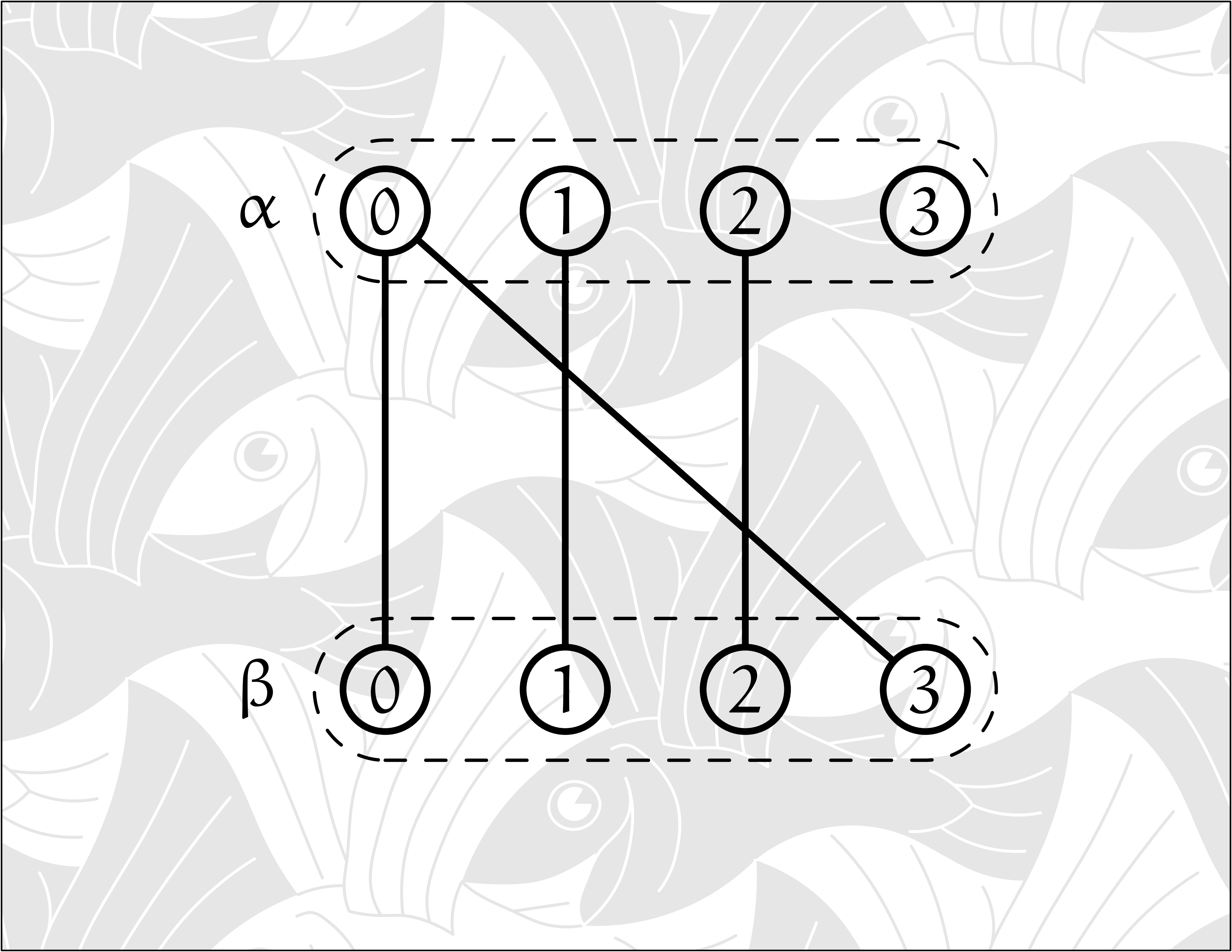
A tuple is said to *satisfy* C_S if it is in C_S.

A *constraint network* is a collection of variables, their domains, and zero or more constraints between subsets of these variables.

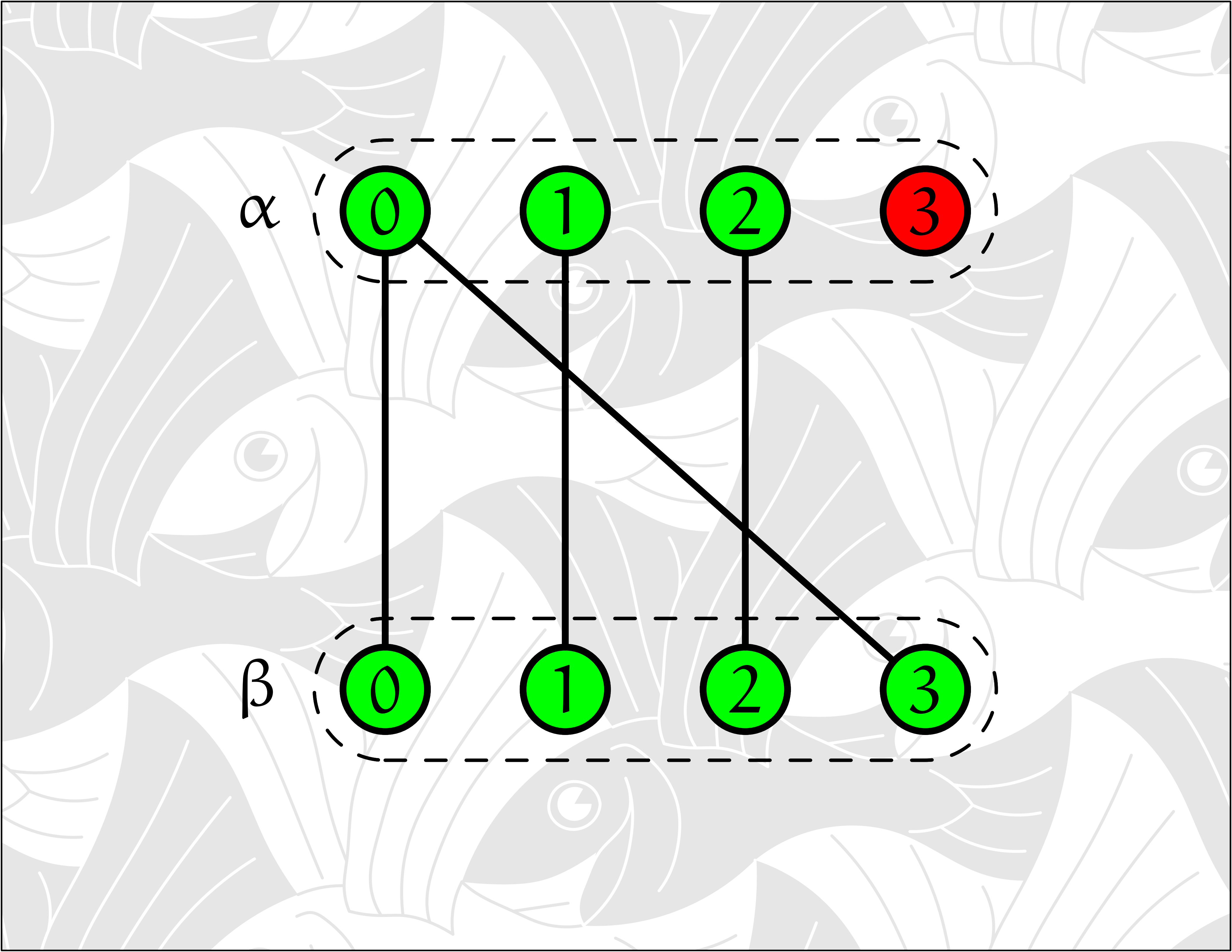
A constraint network is called *binary* each of its constraints is between two variables or less.

Arc-Consistency

A binary constraint network is called *arc-consistent* if none of the domains of its variables is empty and every binary constraint between two variables satisfies the property that each of the values in the domain of each of these variables is *supported* by some value in the domain of the other variable.



α
0
1
2
3
β
0
1
2
3


α
0
1
2
3
β
0
1
2
3

Heuristics

Arc-consistency algorithms carry out *support-checks* to find out about the properties of Constraint Satisfaction Problems.

They use *arc-heuristics* to select the constraint that will be used for the next support-check.

They use *domain-heuristics* to select the values that will be used for the next support-check.

Some Existing Arc-Consistency Algorithms

Two well known arc-consistency algorithms are AC-3 with a $\mathbf{O}(ed^3)$ and AC-7 with a $\mathbf{O}(ed^2)$ worst-case time-complexity.

One of the nice properties of AC-7 is that—as opposed to AC-3—it doesn't repeat support-checks. As a matter of fact, its worst-case time-complexity is optimal and it behaves well in practice.

AC-3 on the other hand has nicer space-complexity characteristics than AC-7 ($\mathbf{O}(e + nd)$ vs. $\mathbf{O}(ed^2)$).

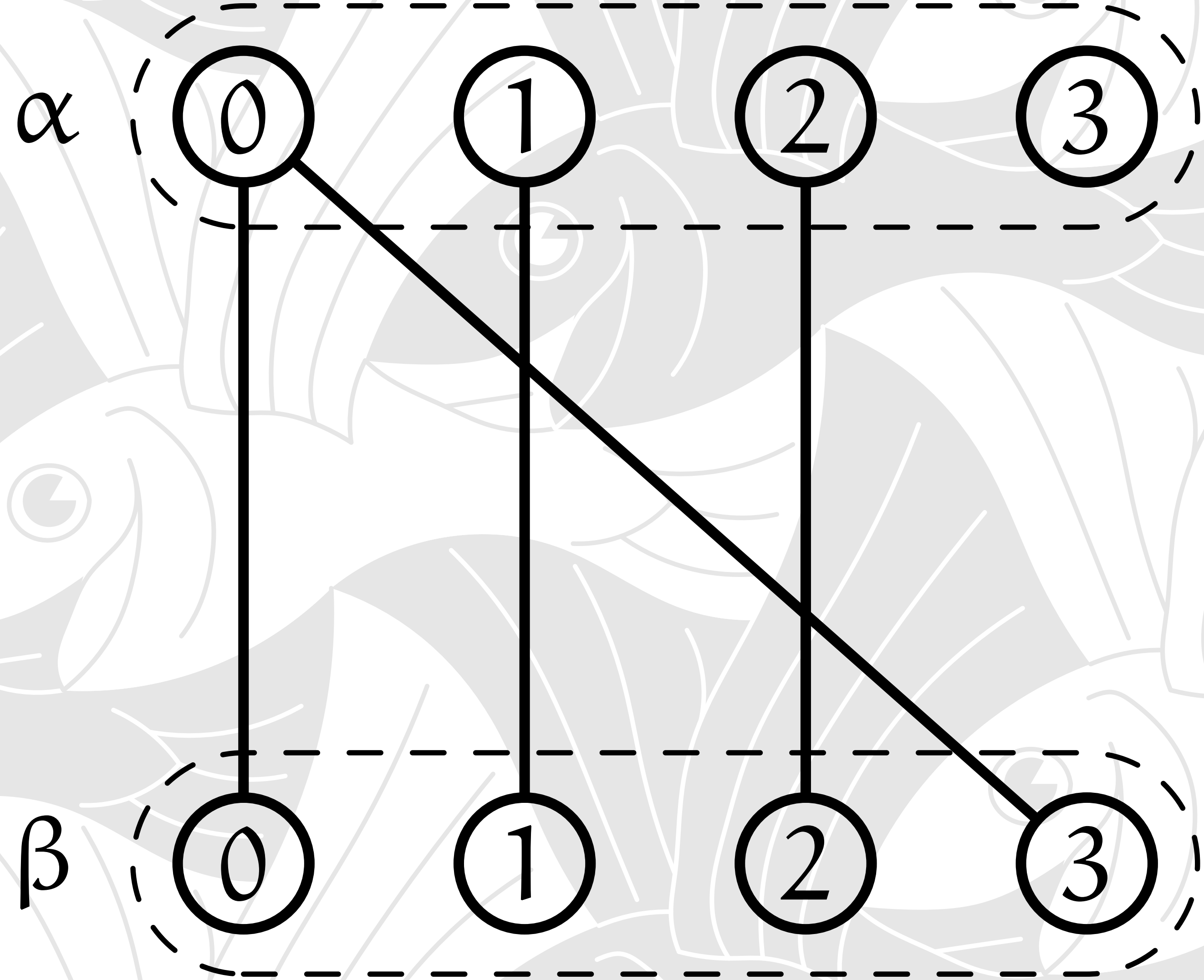
Algorithm $\mathcal{L}$

AC-7 never repeats support-checks. It uses a "seek support" heuristic. It uses directed relations $R_{\alpha\beta}$ and $R_{\beta\alpha}$.

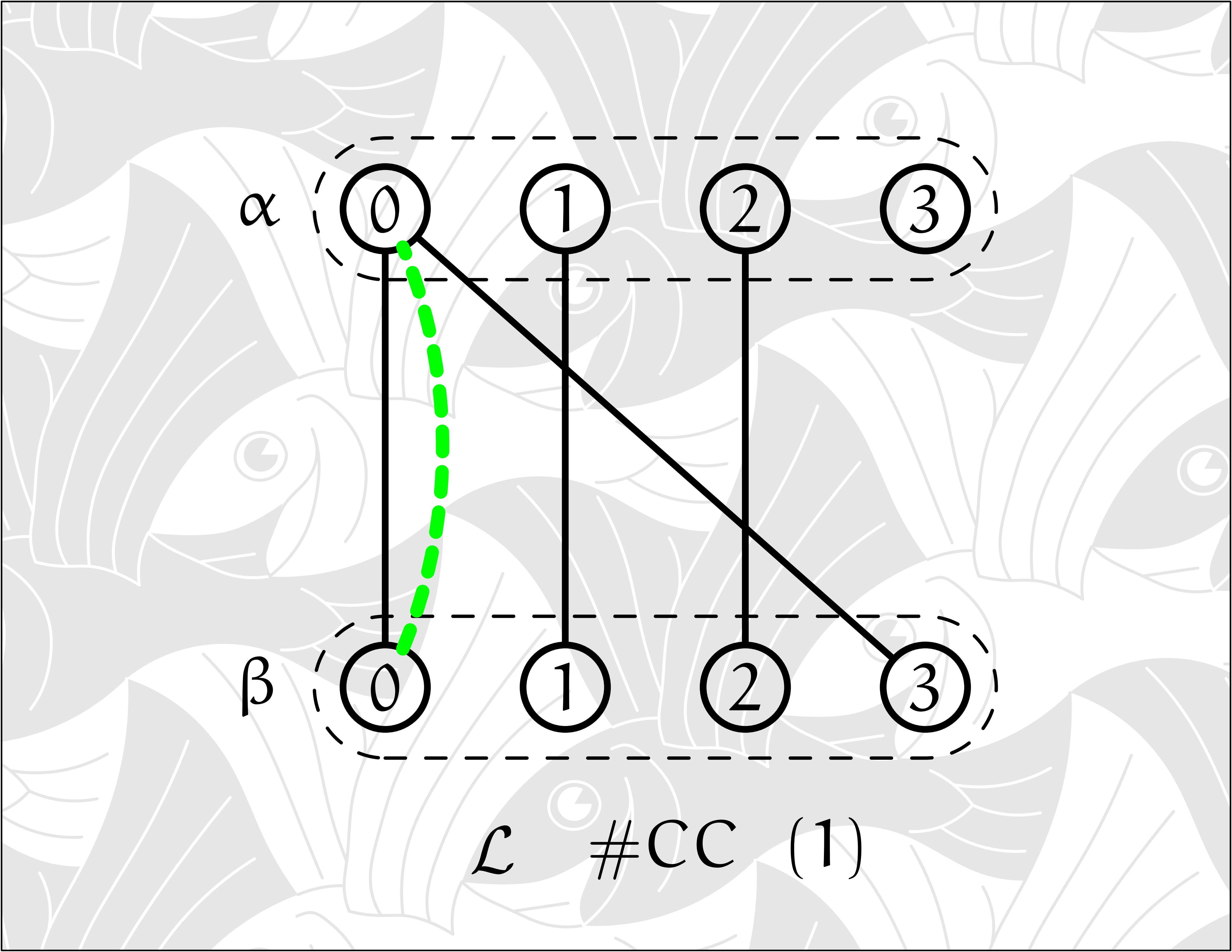
if $a \in D(\alpha)$ **is unsupported** then AC-7 will try to find support for a using a check of the form $(\,a, b\,) \in R_{\alpha\beta}$.

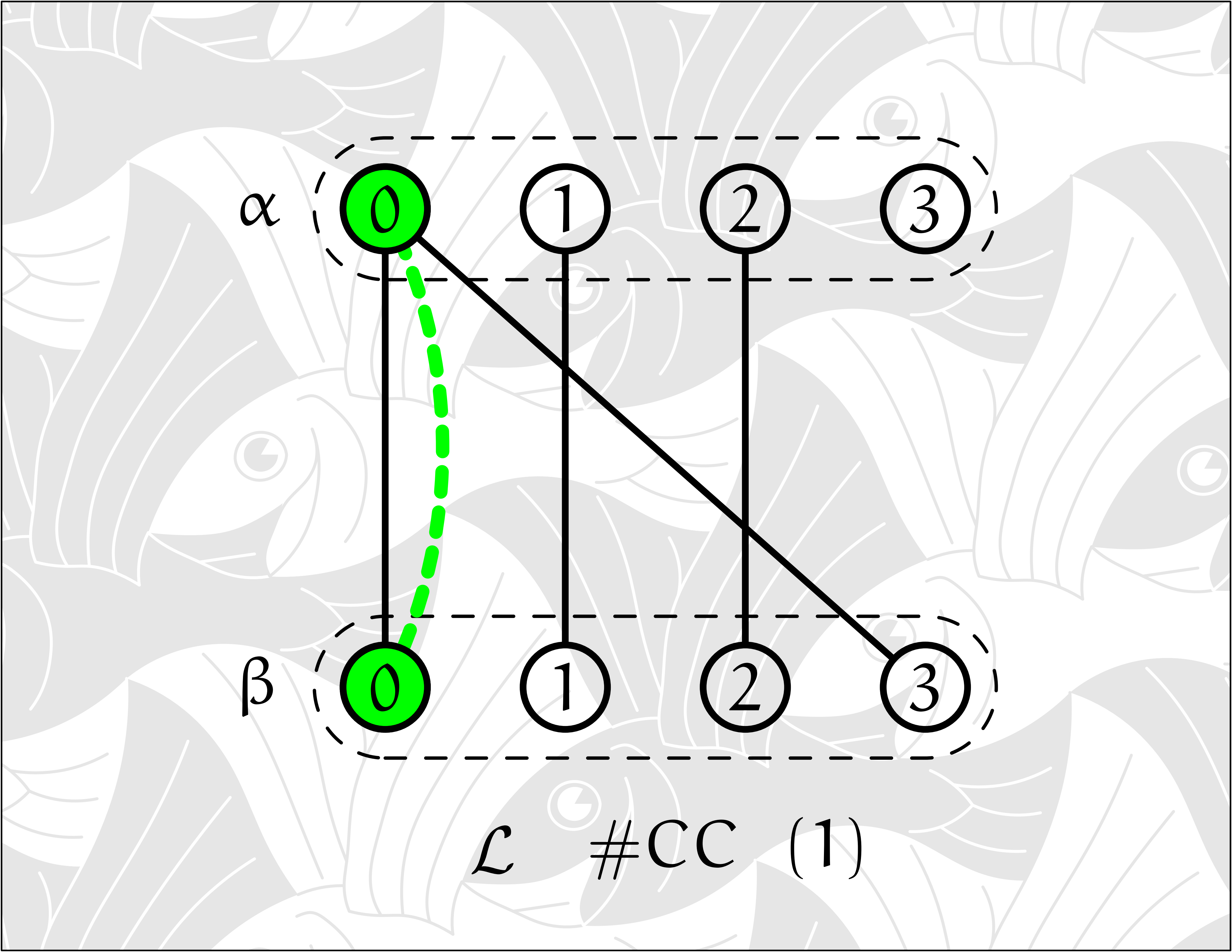
if $b \in D(\beta)$ **is unsupported** then AC-7 will try to find support for b using a check of the form $(\,b, a\,) \in R_{\beta\alpha}$.

AC-7 normally comes equipped with lexicographical arc-heuristics and domain-heuristics. Let $\mathcal{L}$ be that algorithm.

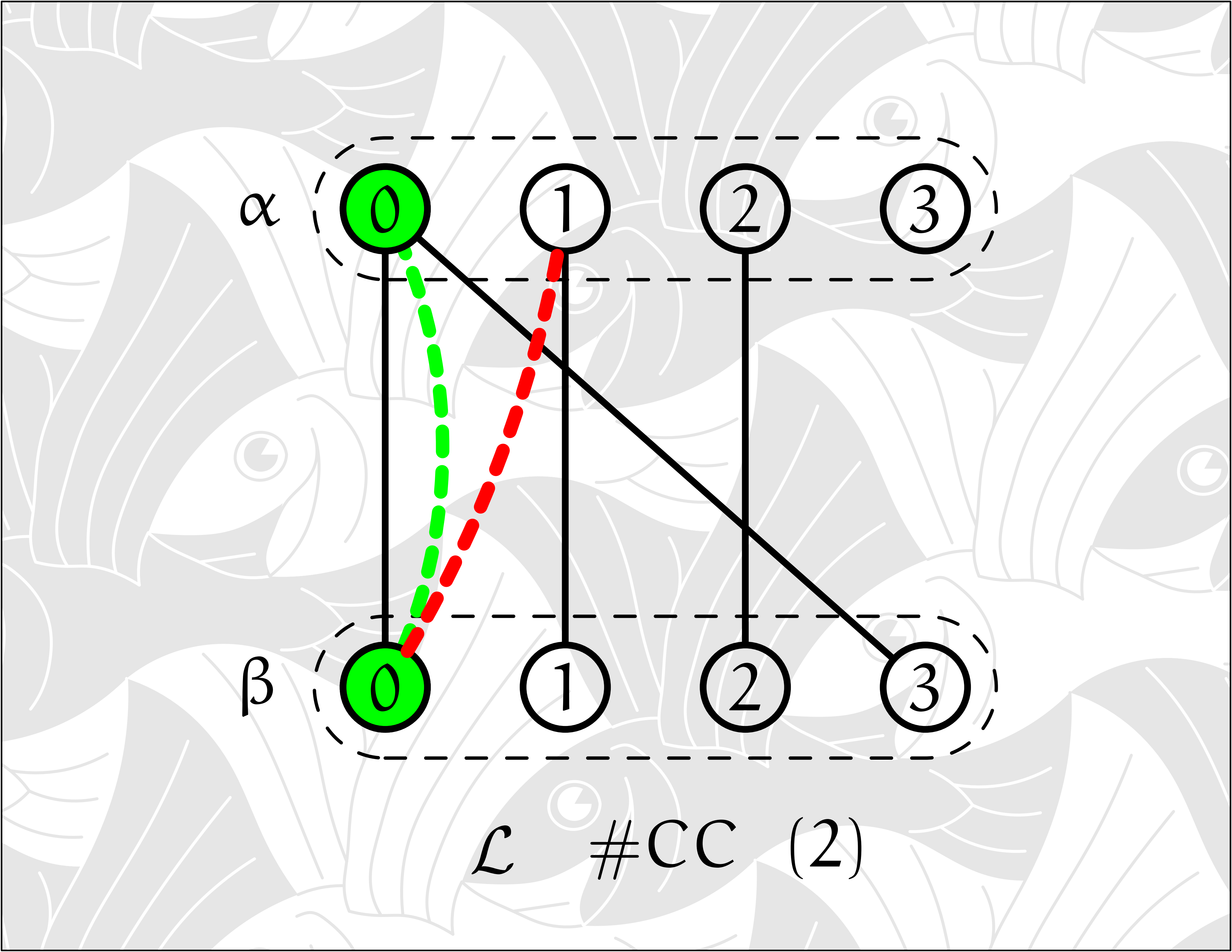


$\mathcal{L}$ #CC (0)

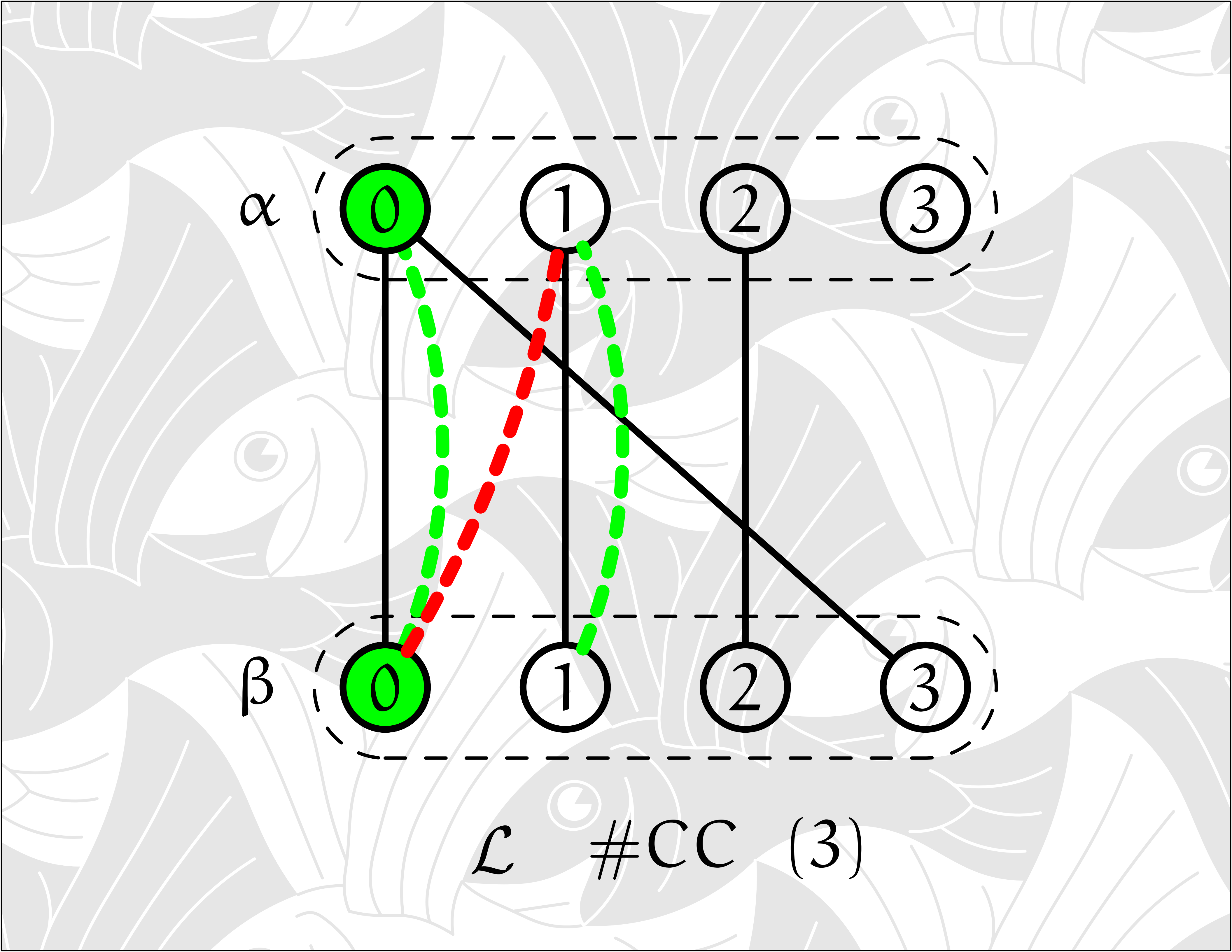

α
0
1
2
3
β
0
1
2
3
ℒ #CC (1)



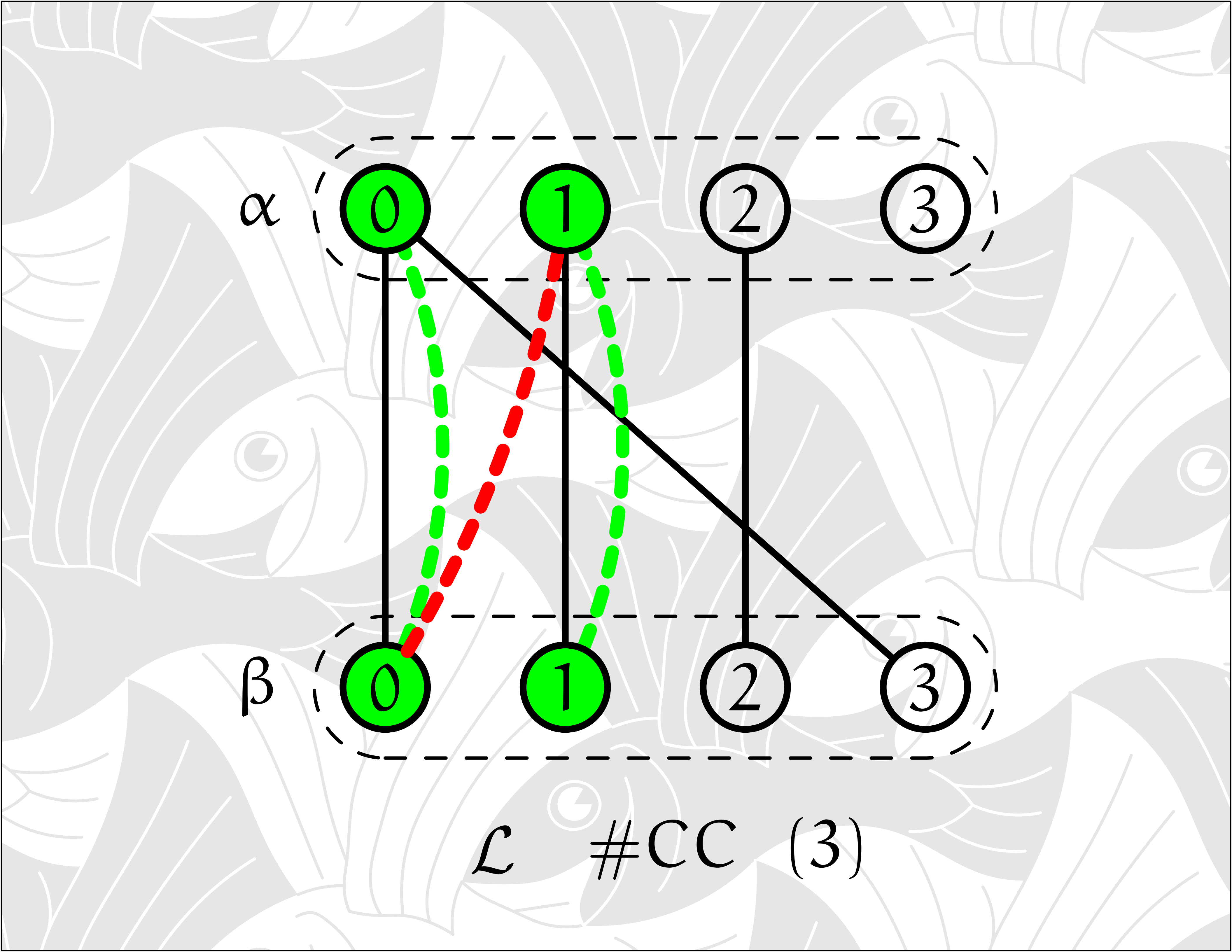
α
0
1
2
3
β
0
1
2
3
ℒ #CC (1)



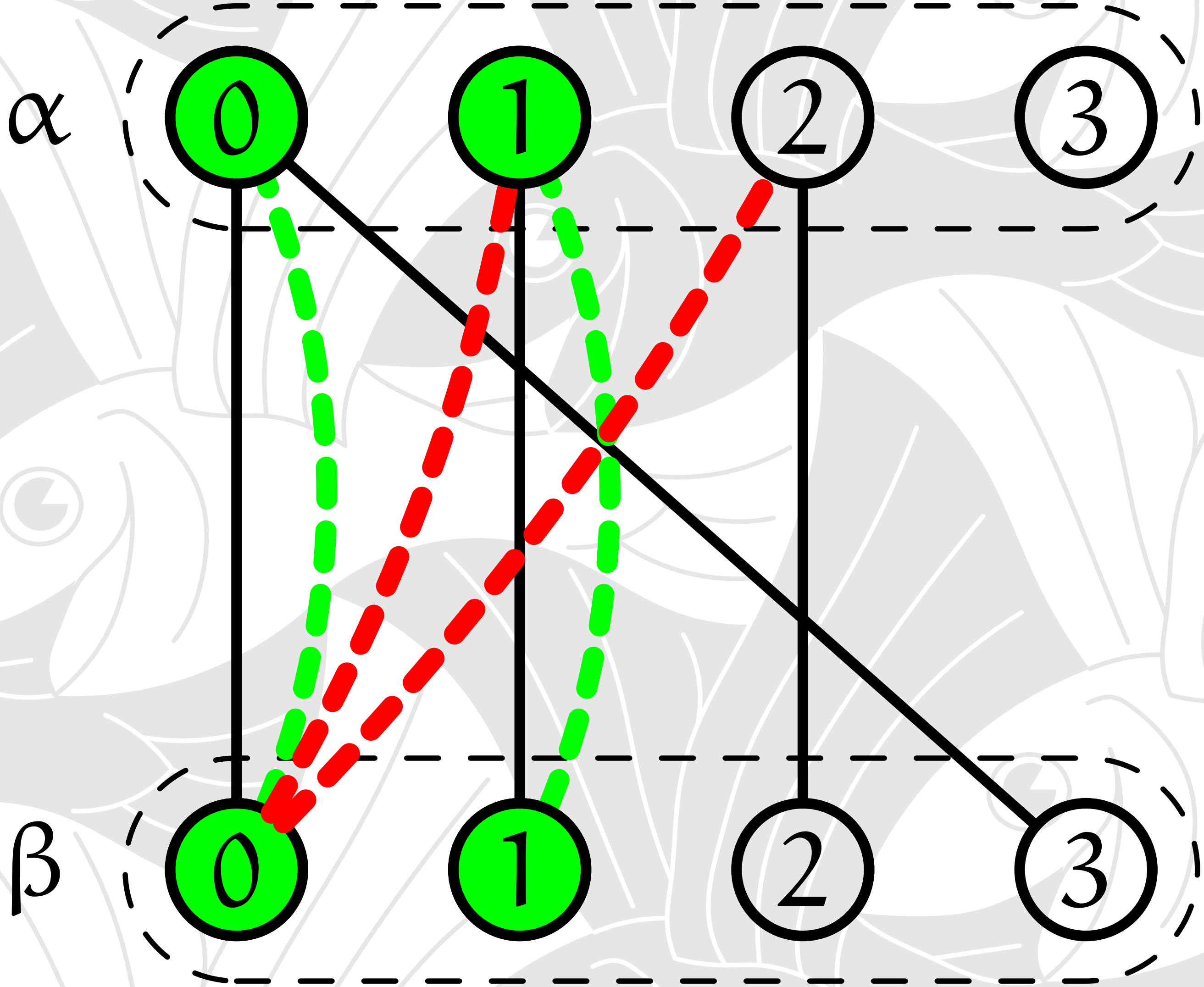
α
0
1
2
3
β
0
1
2
3
ℒ #CC (2)



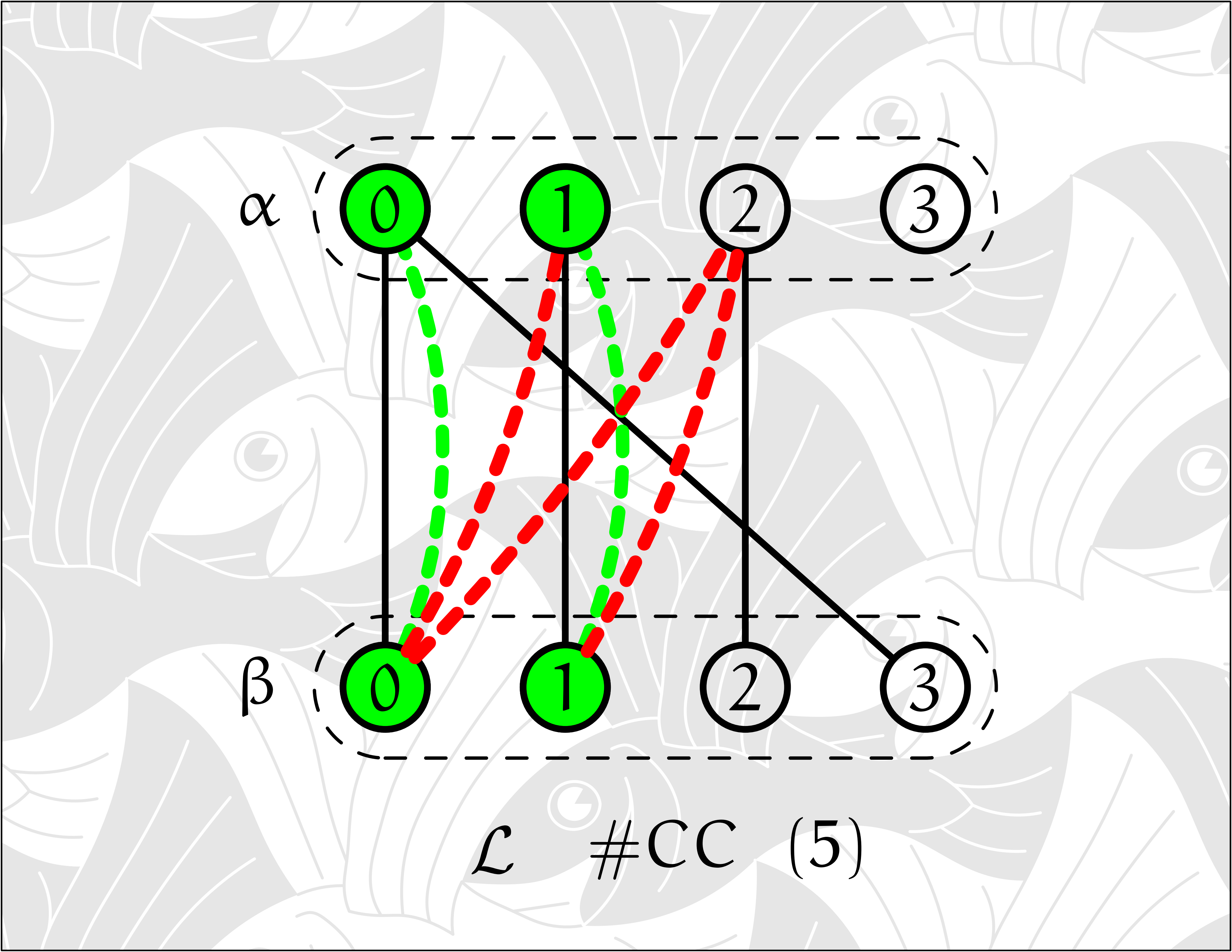
α
β
0
1
2
3
0
1
2
3
$\mathcal{L}$ #CC (3)



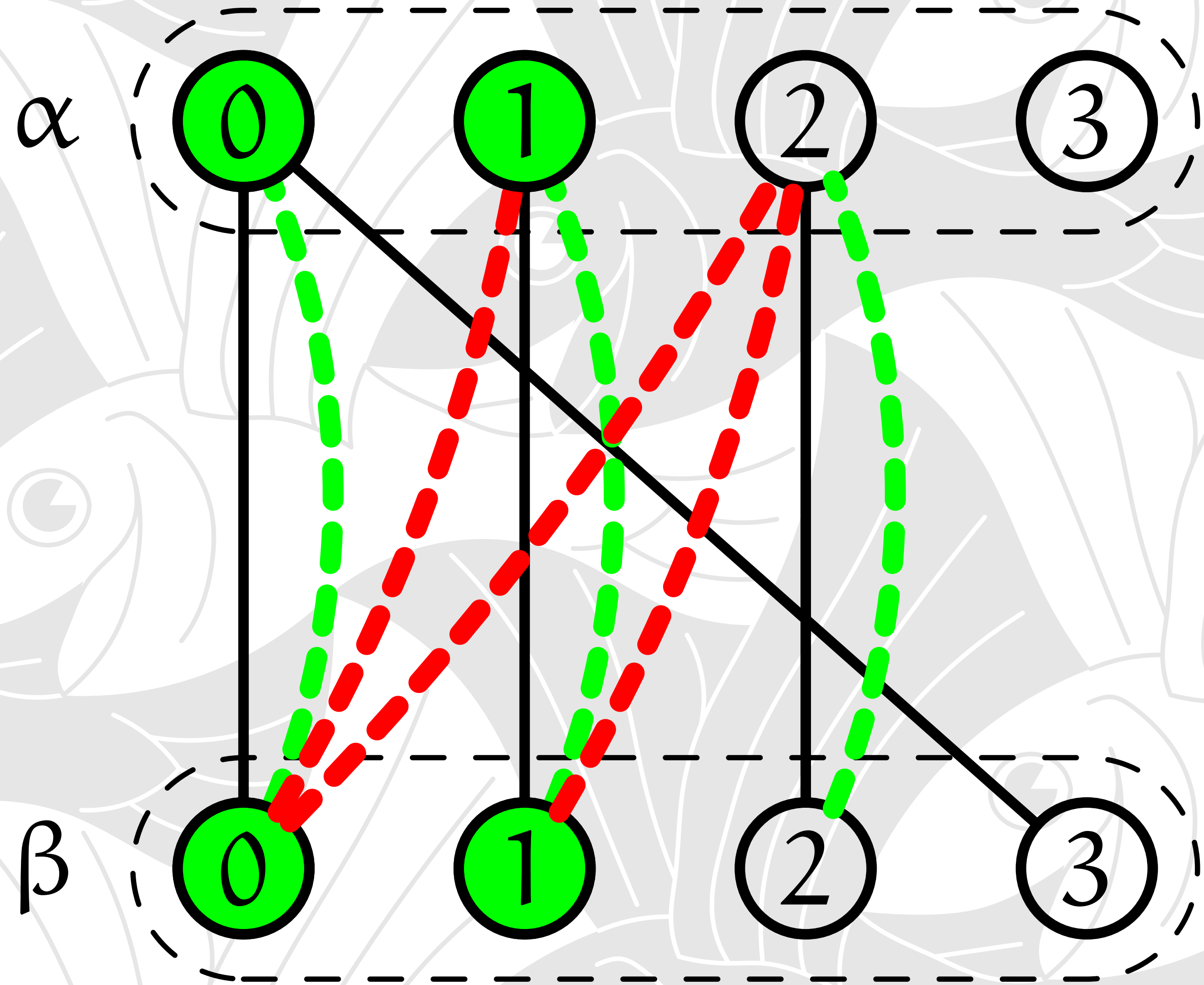
α
β
0
1
2
3
0
1
2
3
ℒ #CC (3)



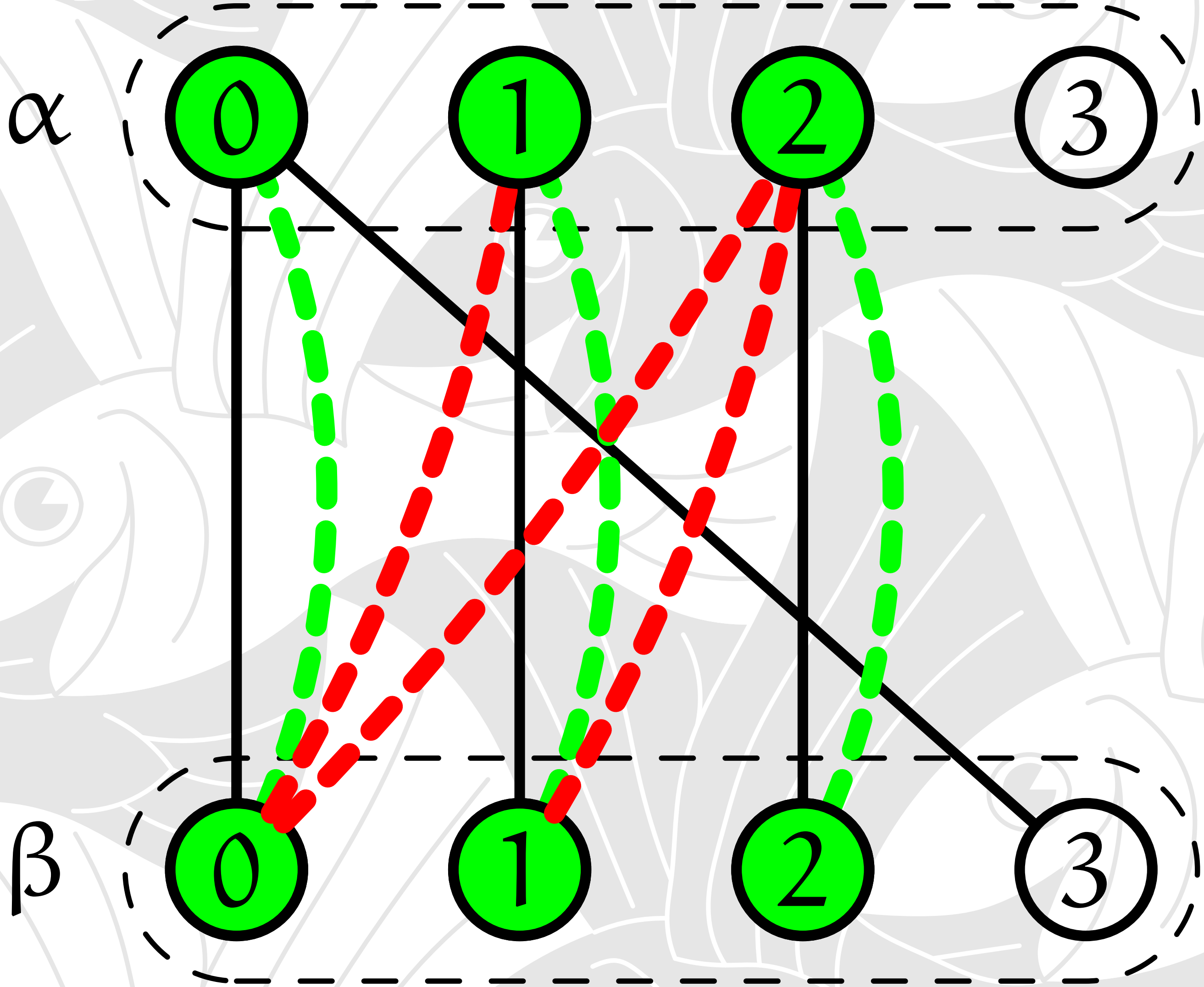
$\mathcal{L}$ #CC (4)



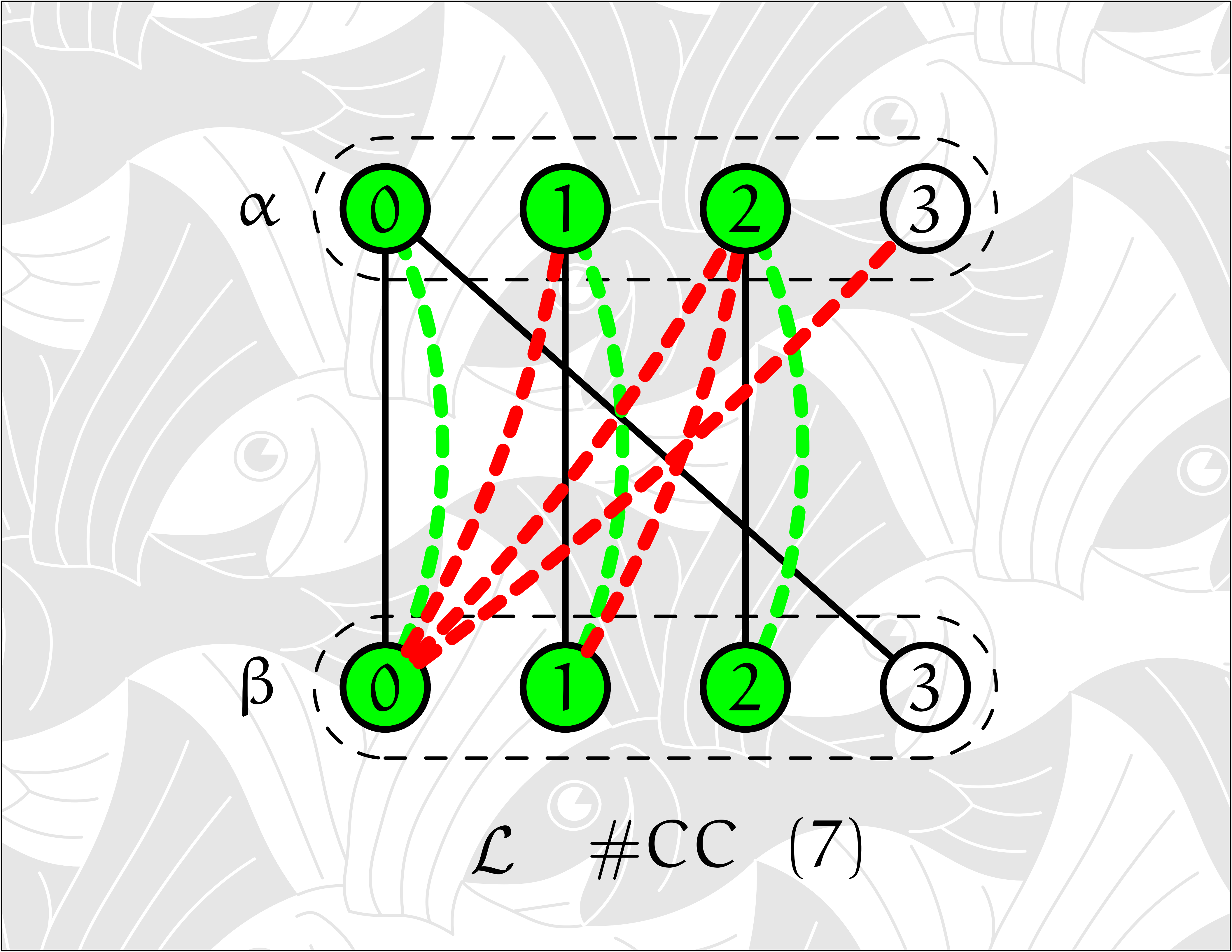
α
β
0
1
2
3
0
1
2
3
ℒ #CC (5)

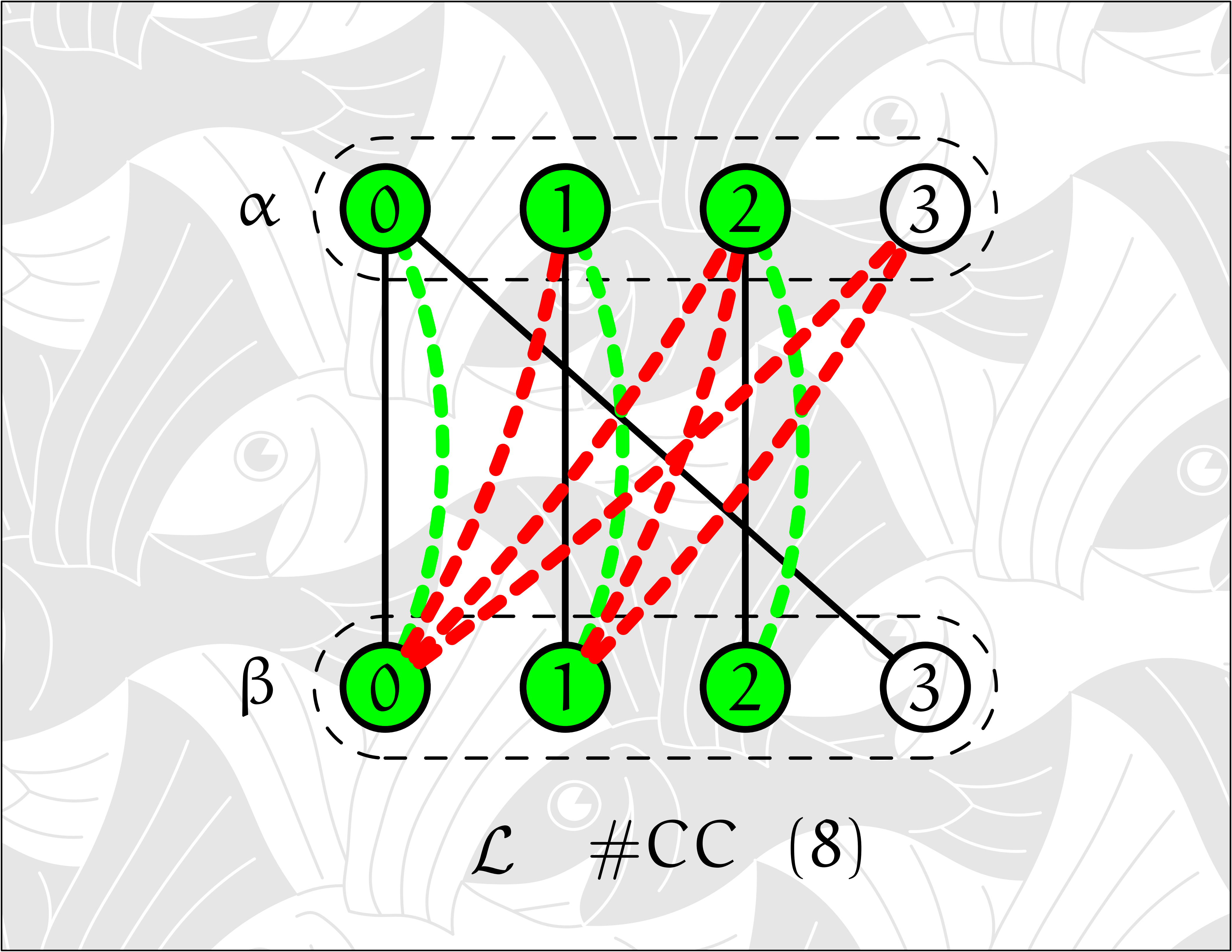


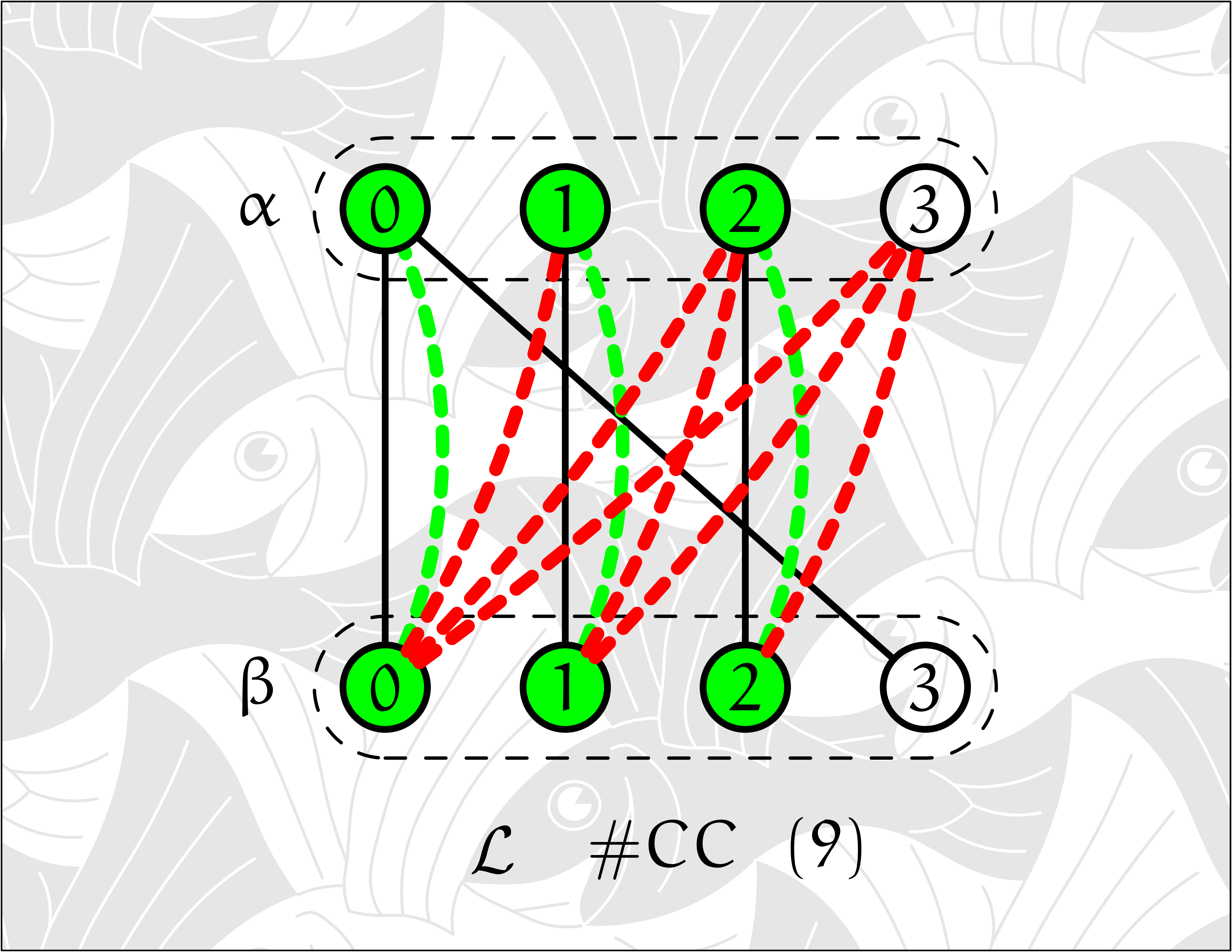
$\mathcal{L}$ #CC (6)



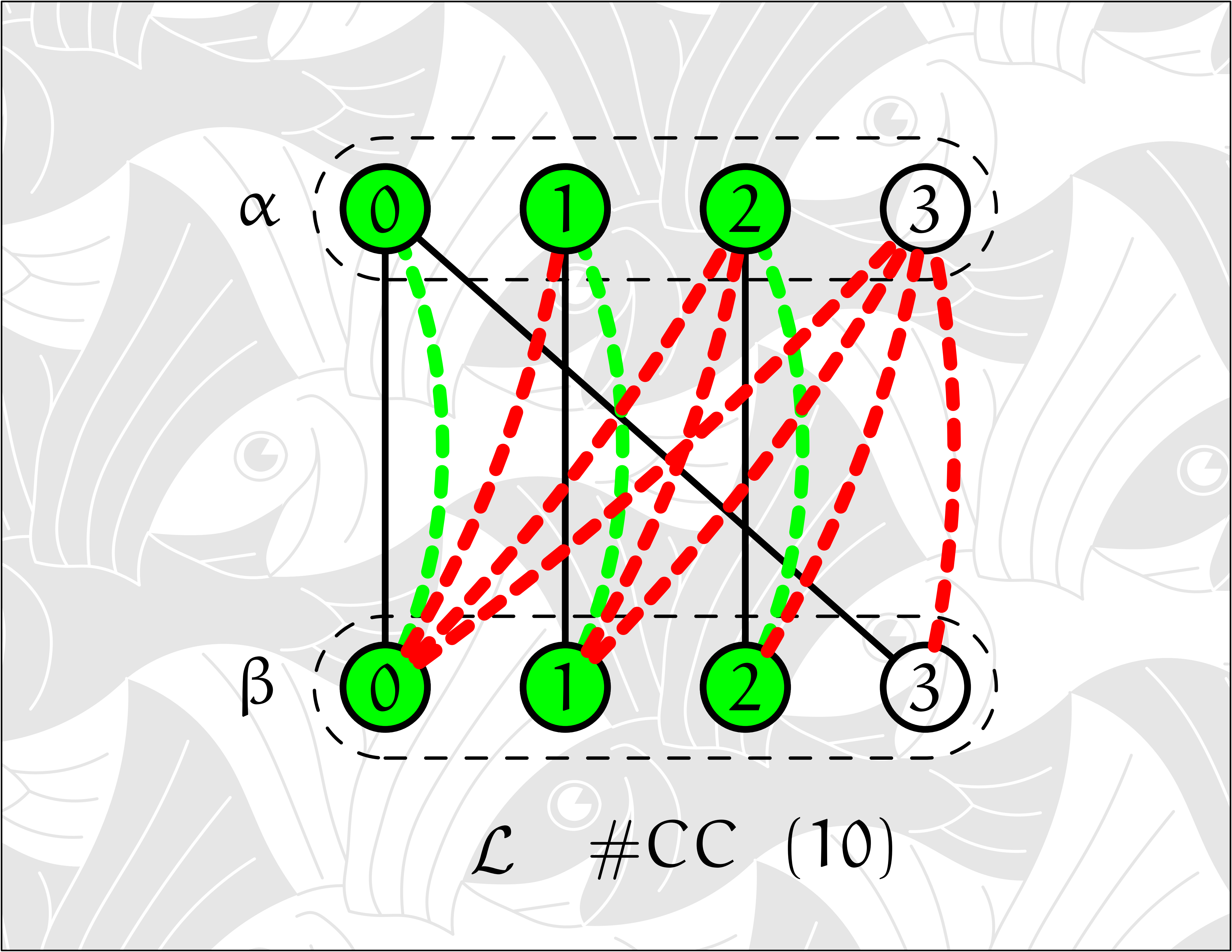
$\mathcal{L}$ #CC (6)


α
0
1
2
3
β
0
1
2
3
$\mathcal{L}$ #CC (7)

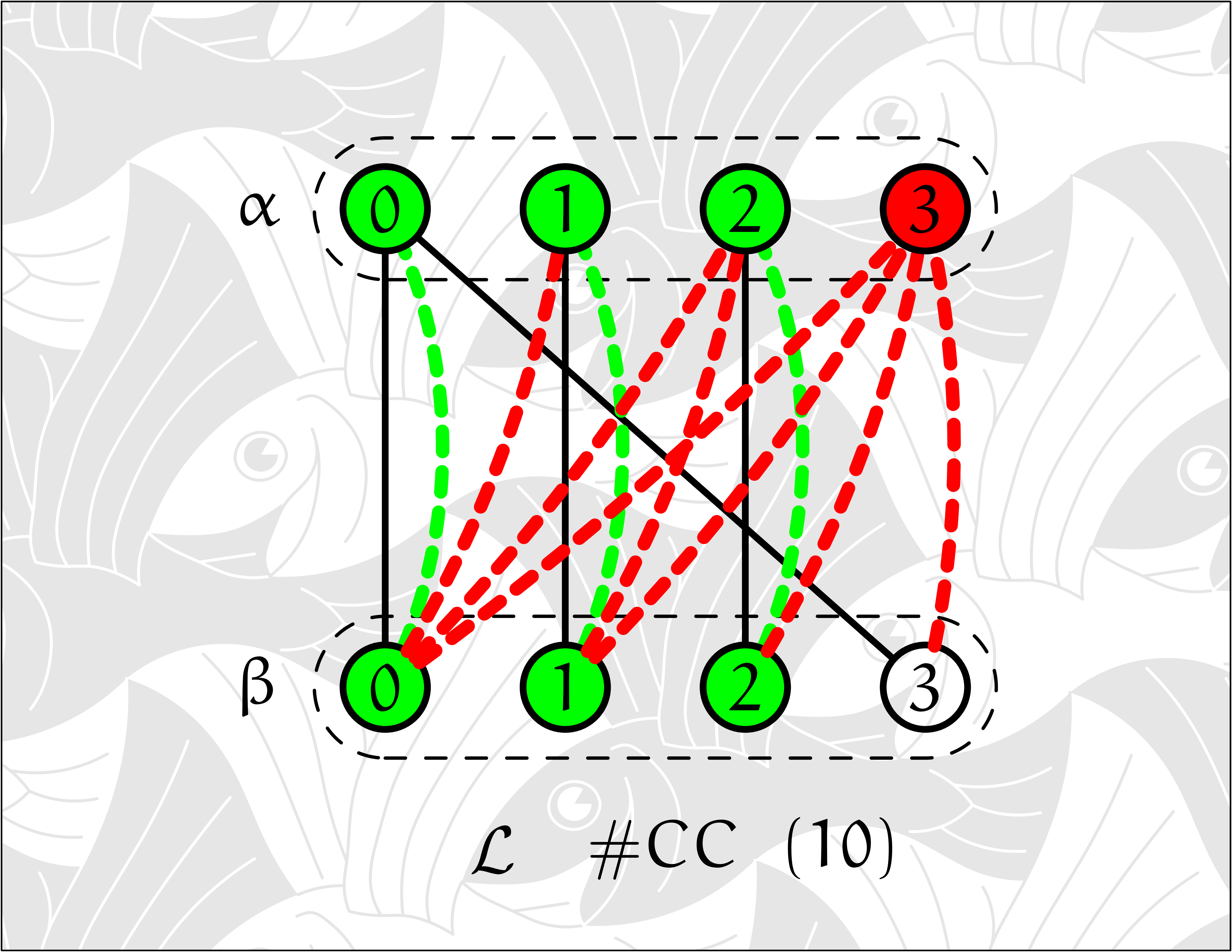

α
β
0
1
2
3
0
1
2
3
$\mathcal{L}$ #CC (8)

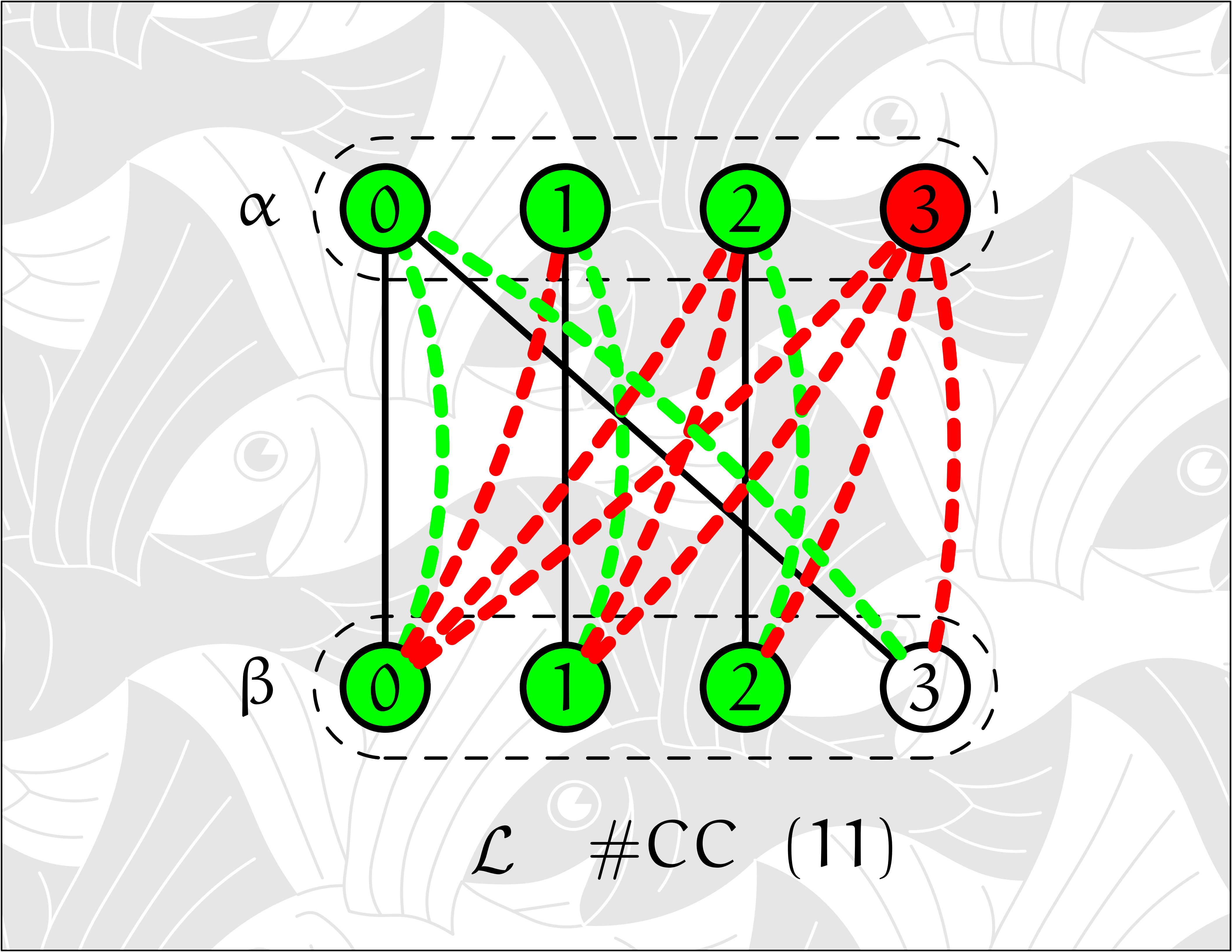


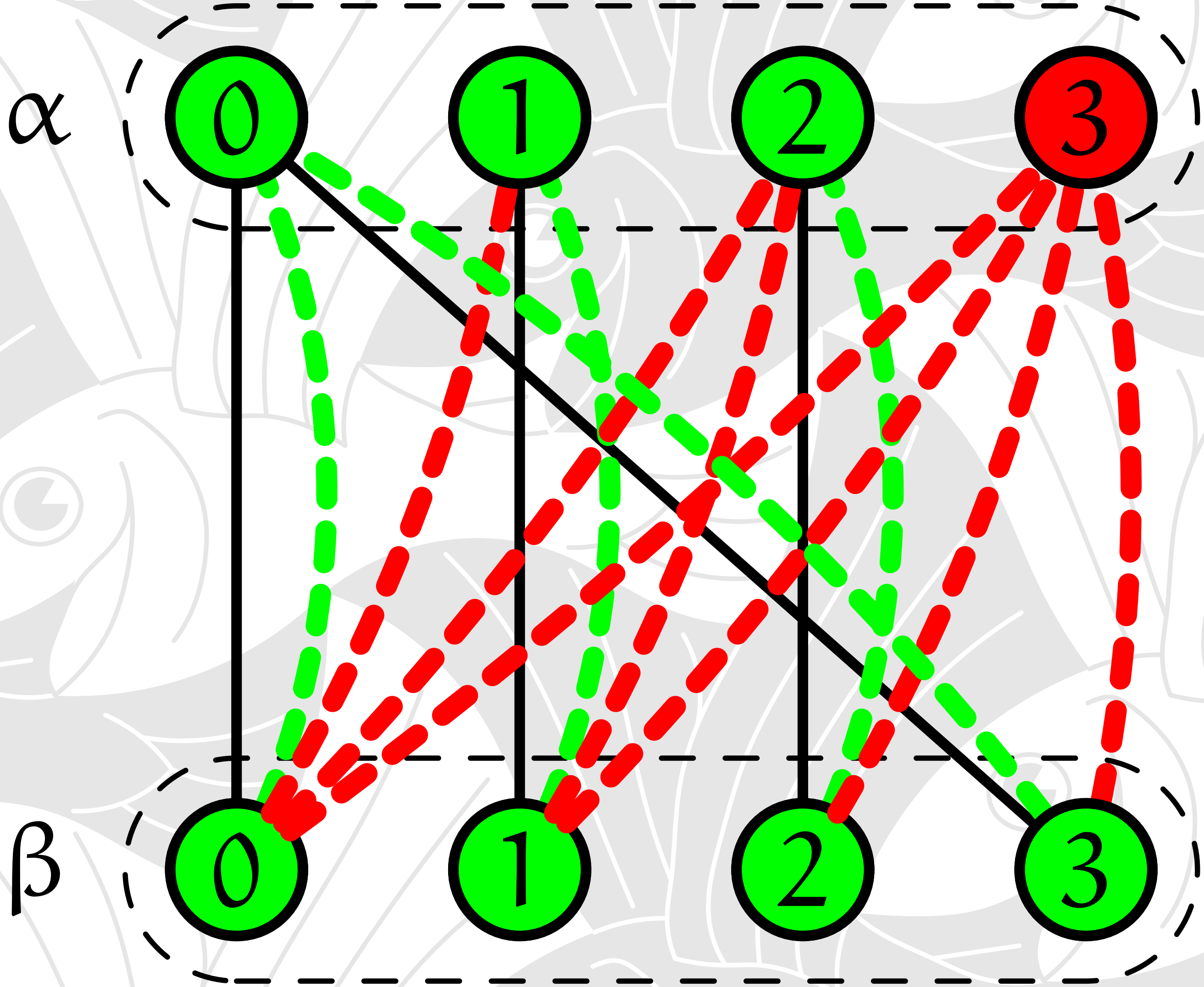
α
0
1
2
3
β
0
1
2
3
ℒ #CC (9)



α
0
1
2
3
β
0
1
2
3
ℒ #CC (10)


α
β
0
1
2
3
0
1
2
3
ℒ #CC (10)


α
0
1
2
3
β
0
1
2
3
$\mathcal{L}$ #CC (11)



$\mathcal{L}$ #CC (11)

Support-Checks

A zero-support check is a check between two values whose supports are non-empty.

Support-Checks

A zero-support check is a check between two values whose supports are non-empty.

A single-support check is a check between a value whose support is non-empty and a value whose support is empty.

Support-Checks

A zero-support check is a check between two values whose supports are non-empty.

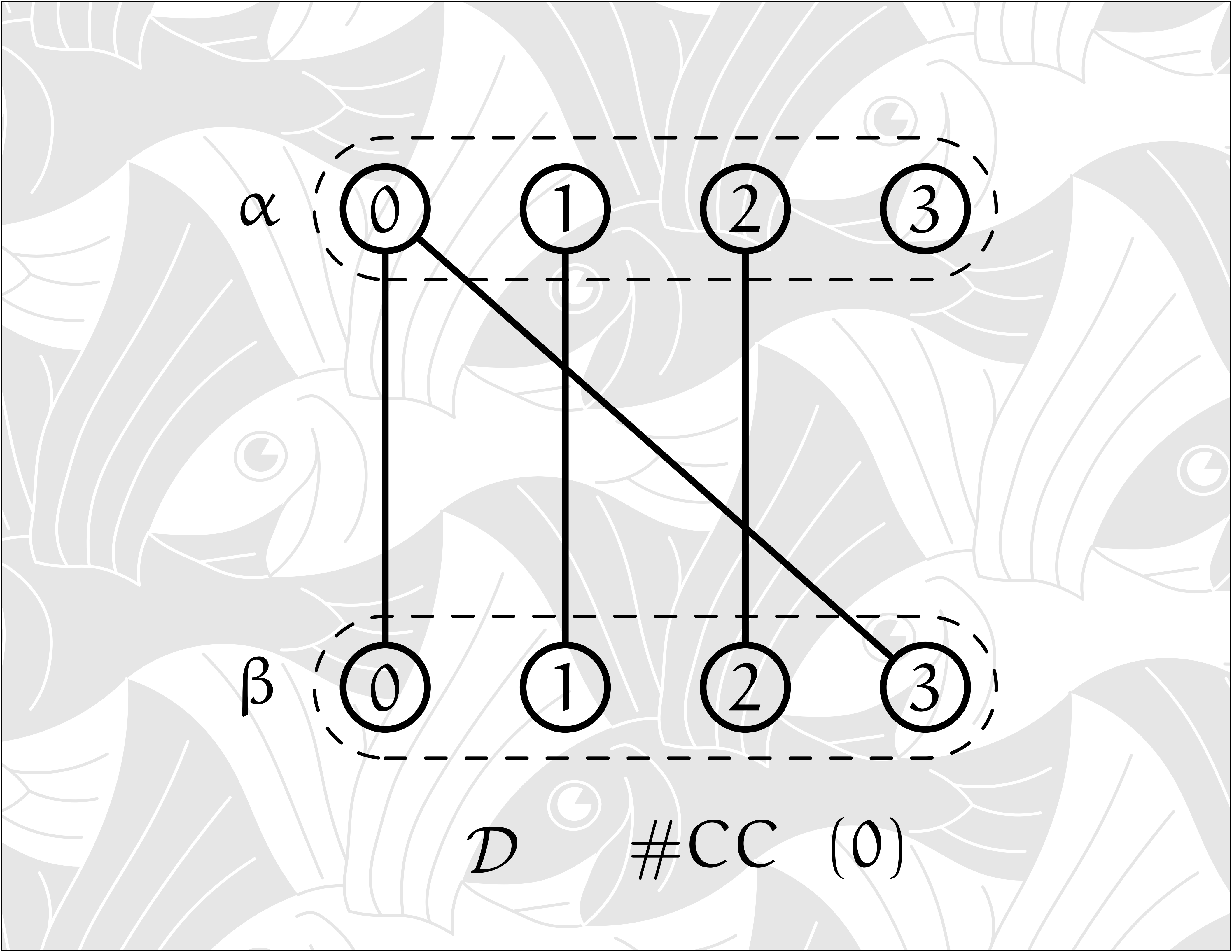
A single-support check is a check between a value whose support is non-empty and a value whose support is empty.

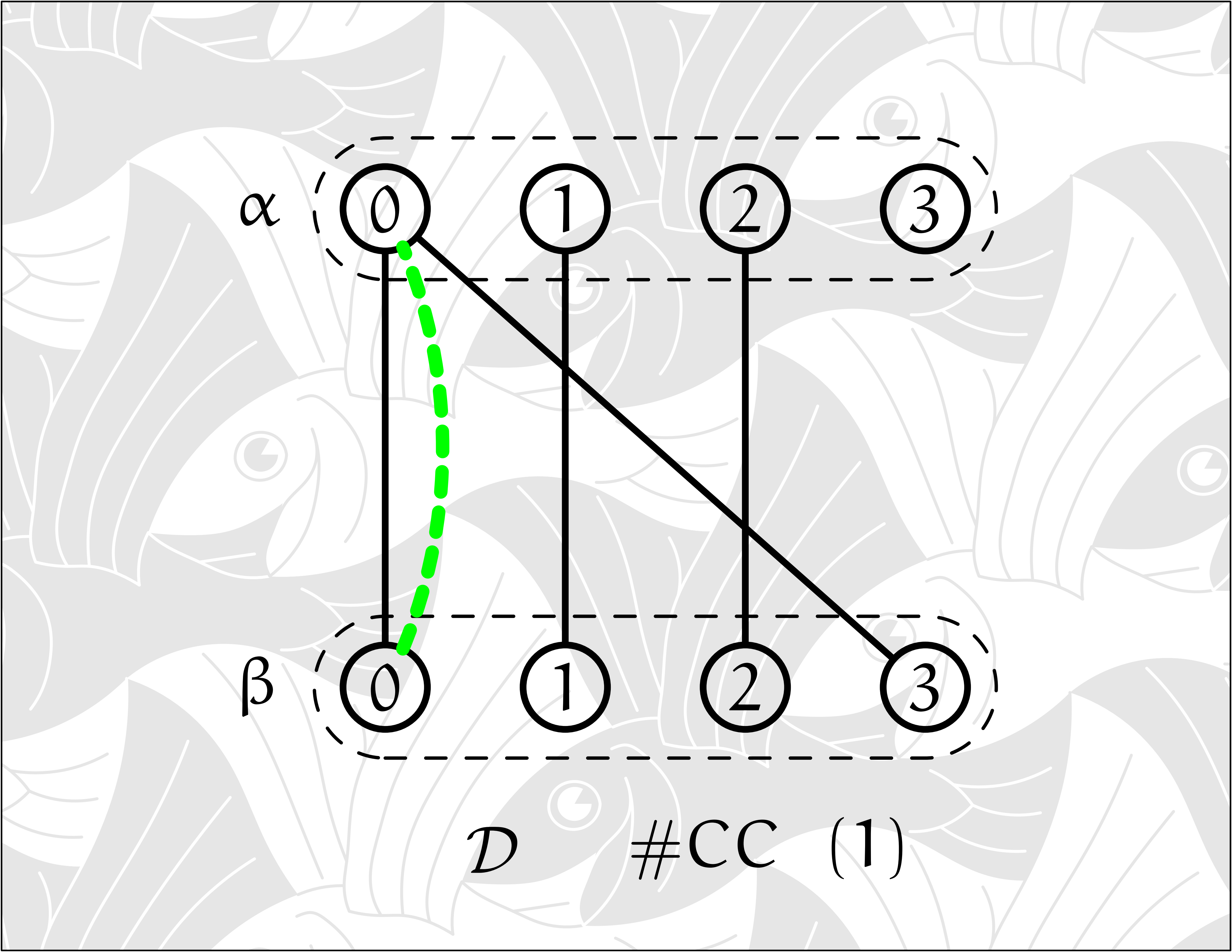
A double-support check is a check between two values whose supports are empty.

Algorithm $\mathcal{D}$

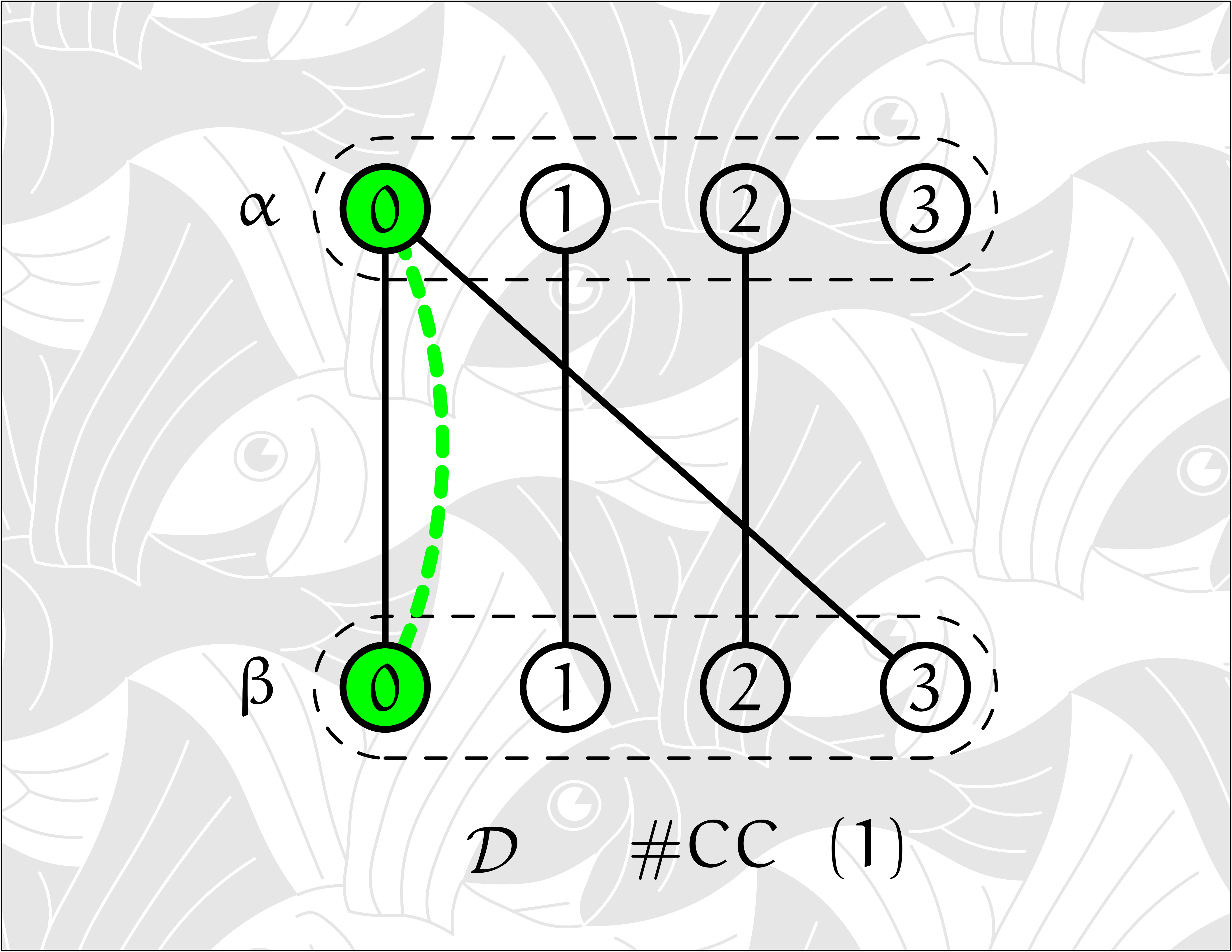
An algorithm which uses a heuristic to *maximise* the number of double-support checks.

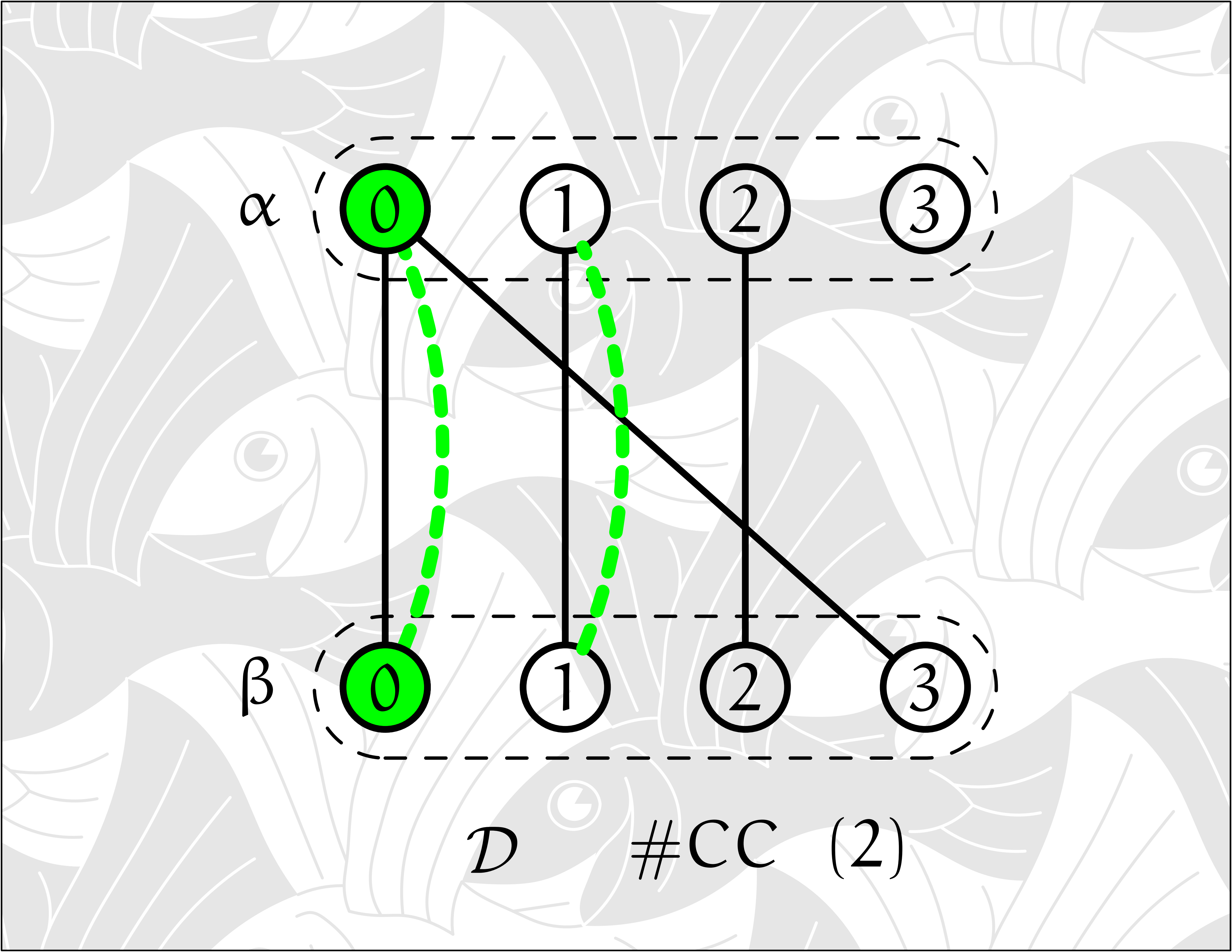
This heuristic can be incorporated into most arc-consistency algorithms.

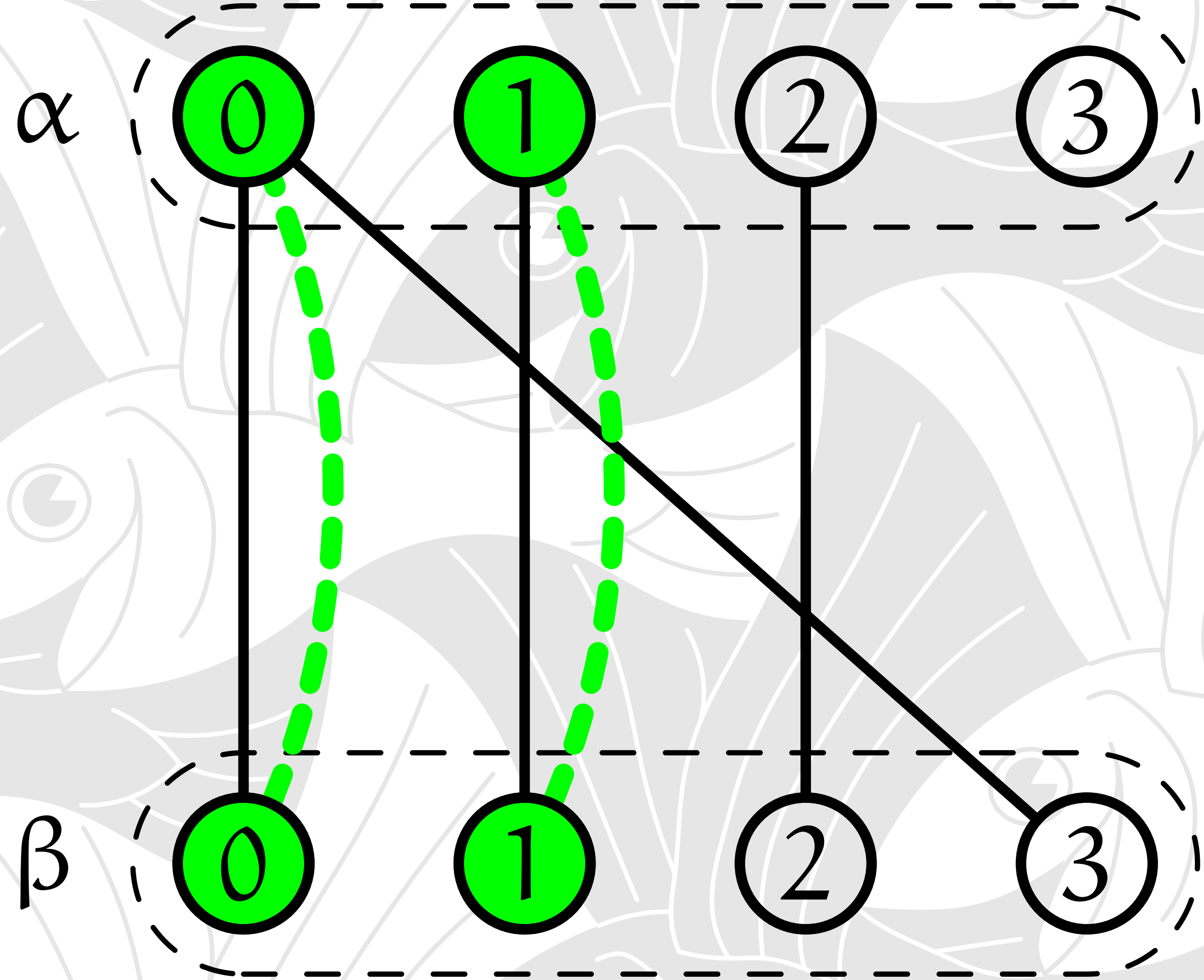

α
0
1
2
3
β
0
1
2
3
𝒟
#CC (0)



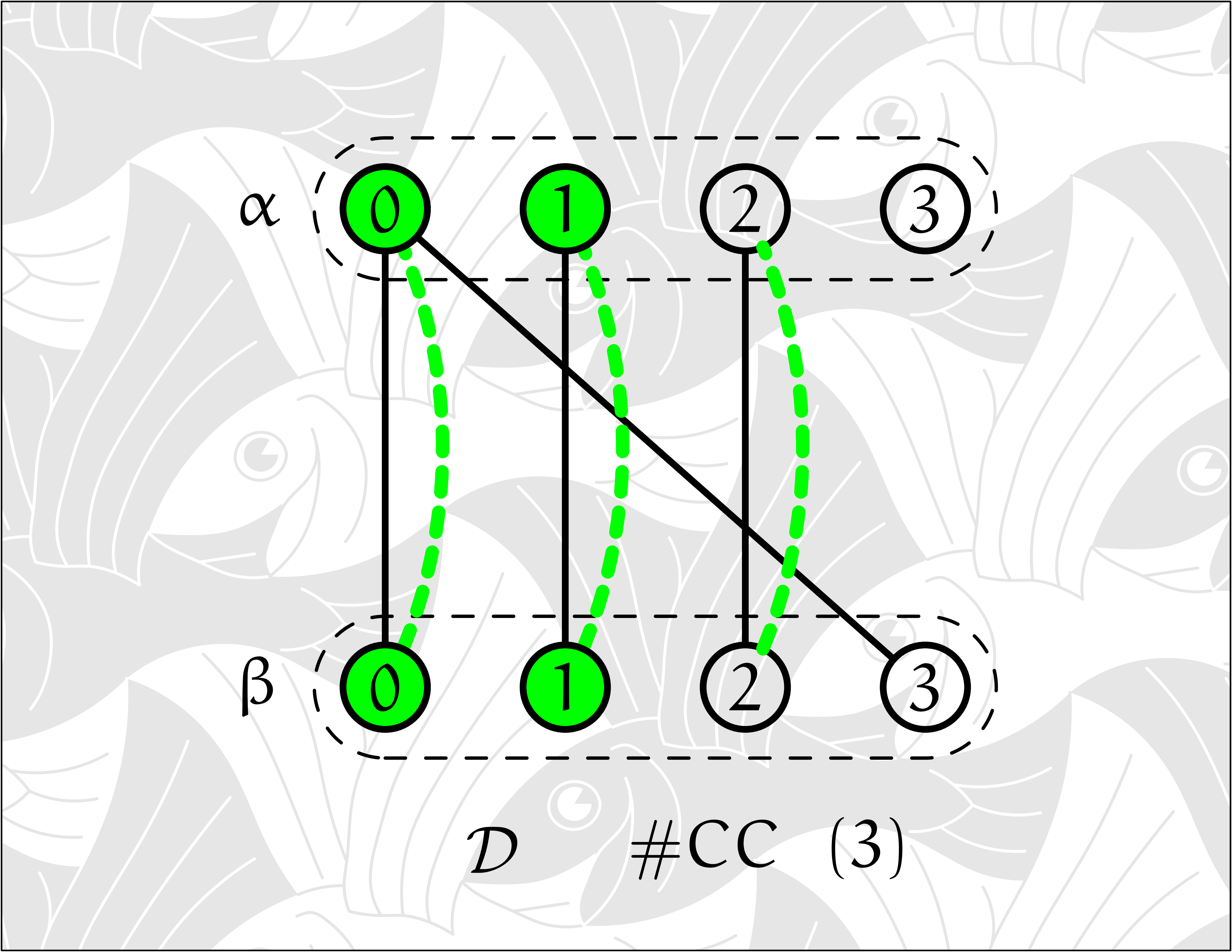
α
0
1
2
3
β
0
1
2
3
𝒟
#CC (1)


α
0
1
2
3
β
0
1
2
3
$\mathcal{D}$ #CC (1)

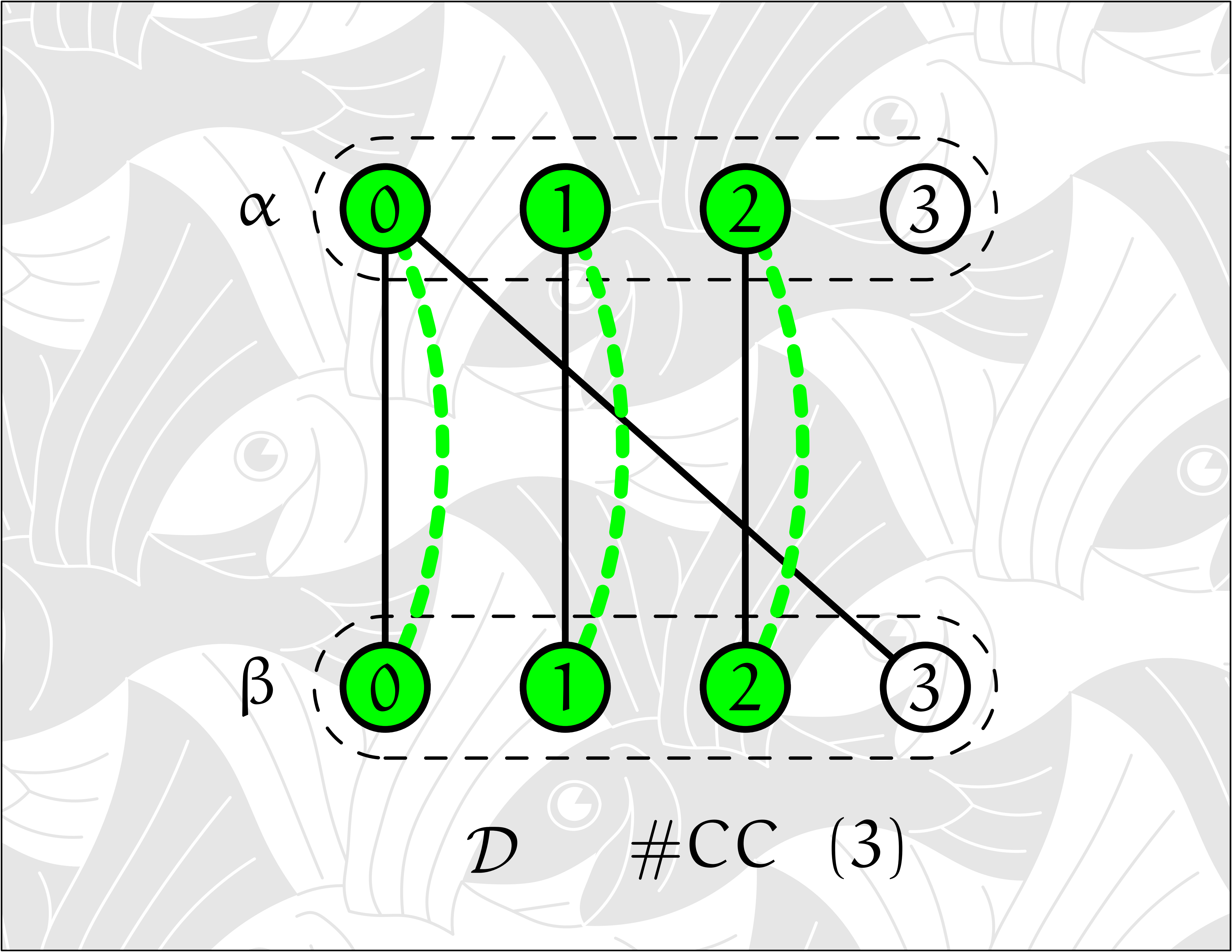

α
β
0
1
2
3
0
1
2
3
𝒟 #CC (2)

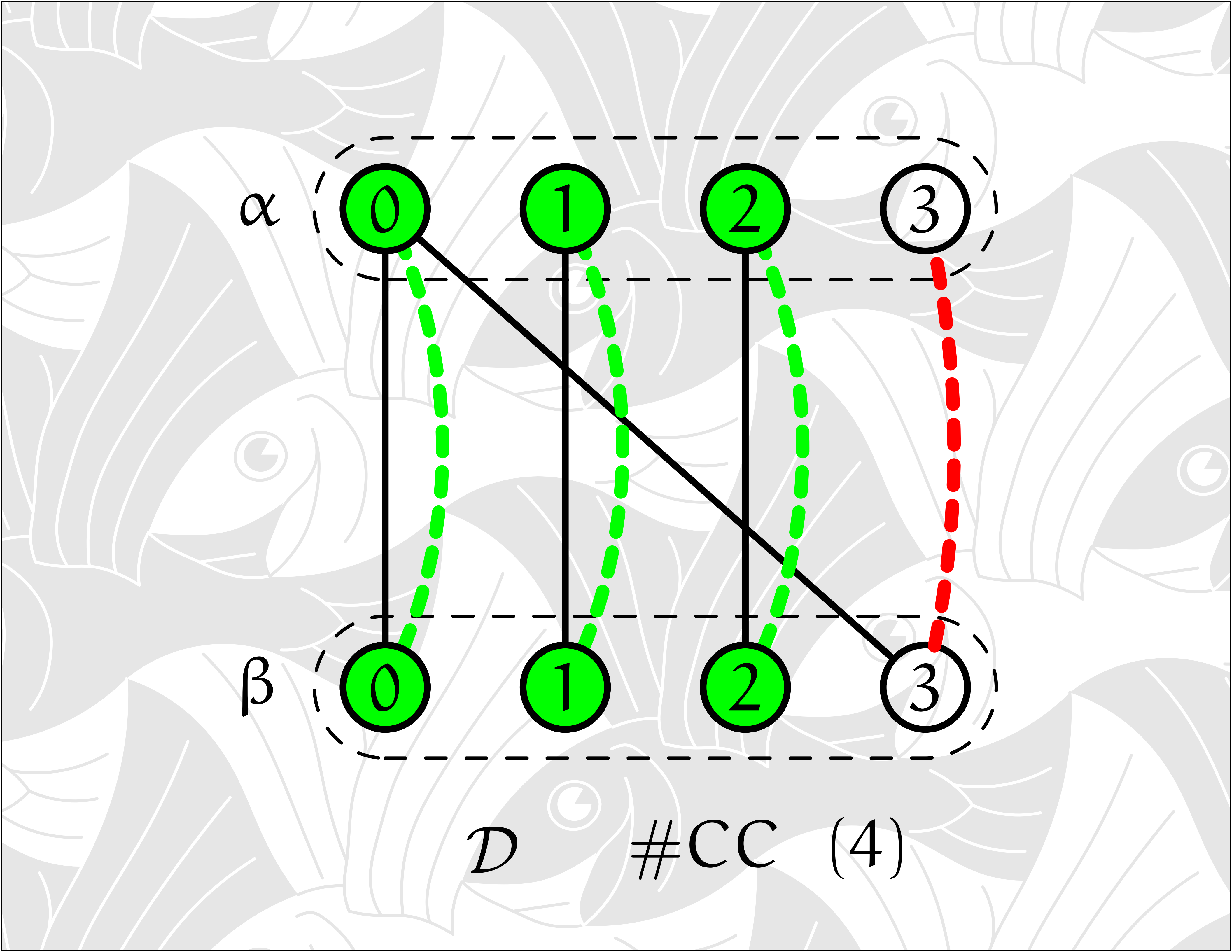


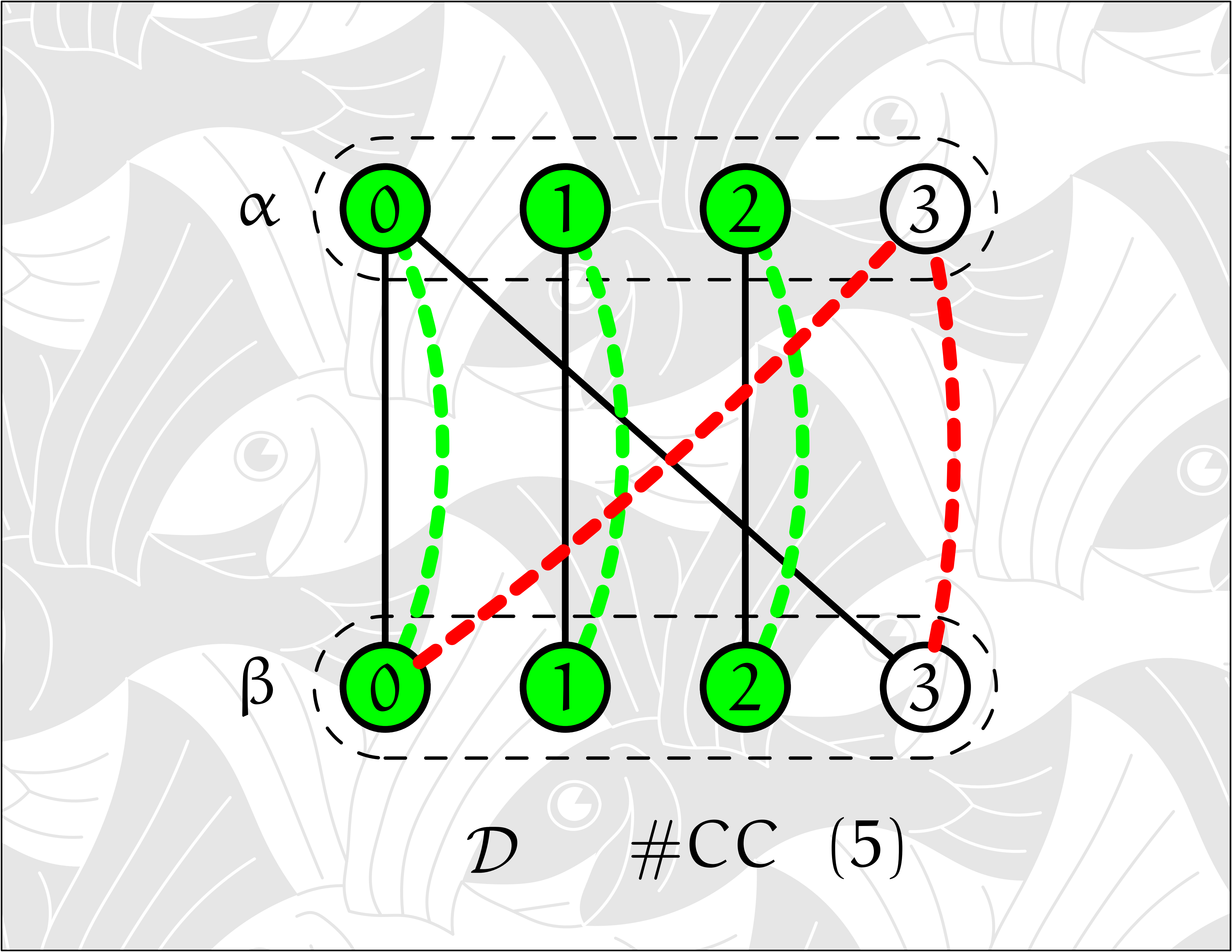
$\mathcal{D}$ #CC (2)

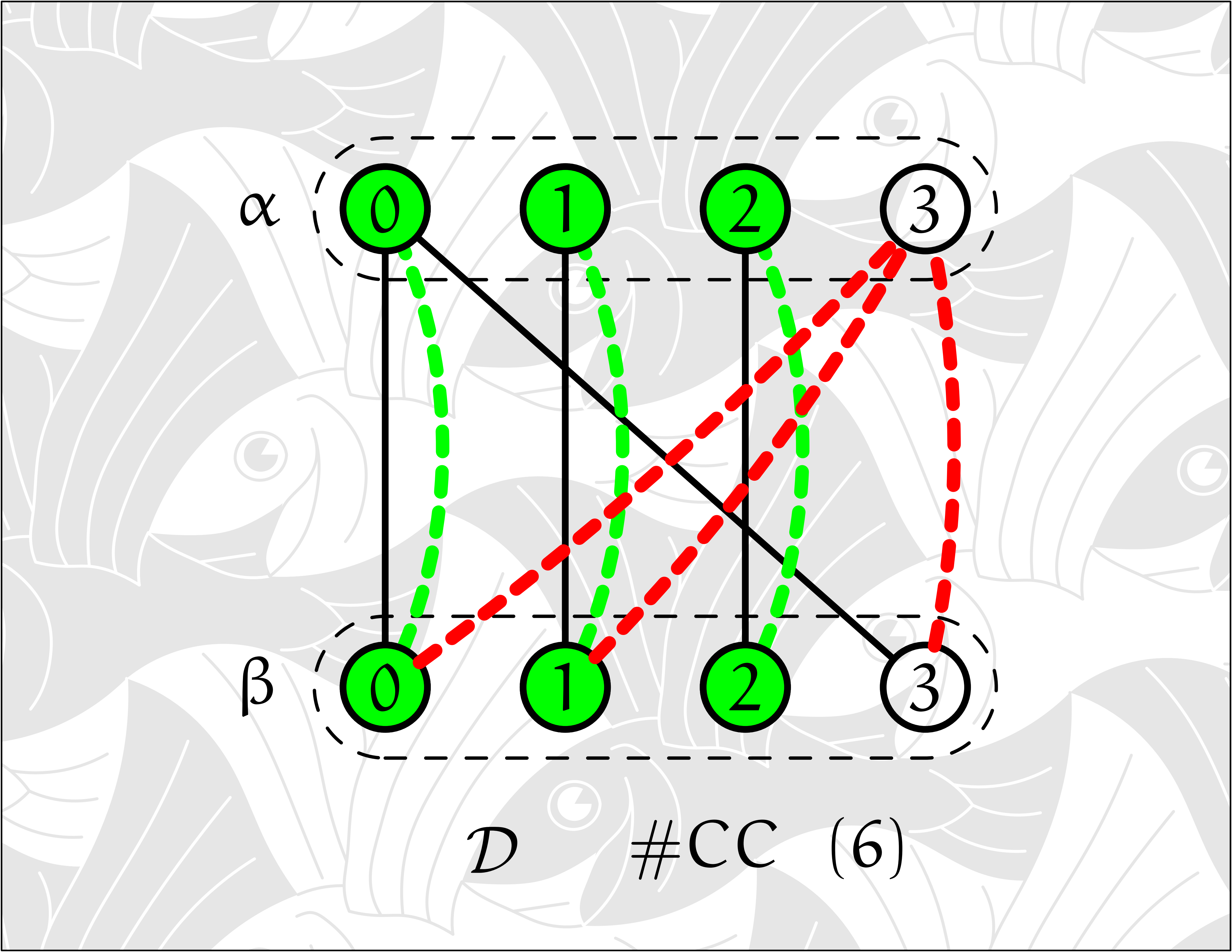


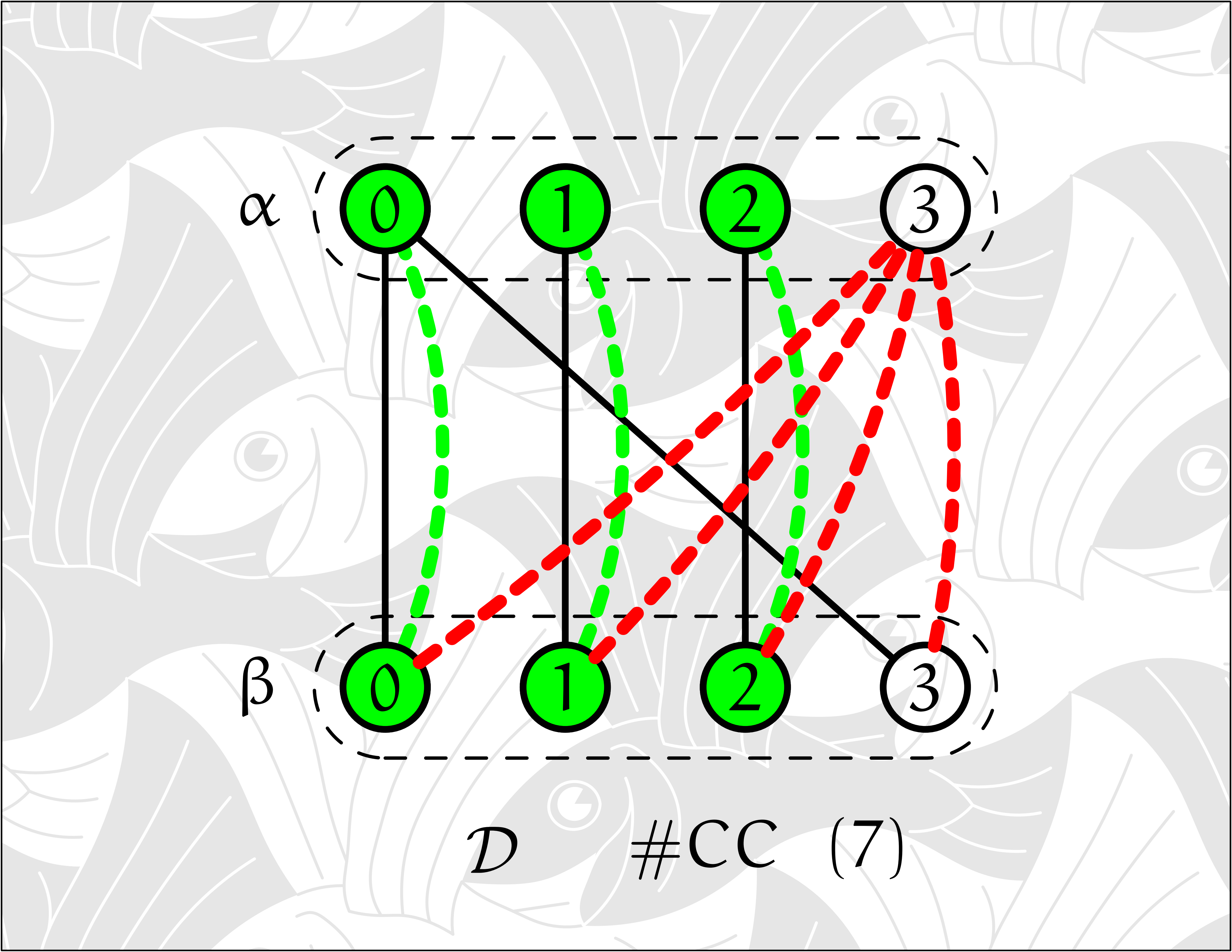
α
0
1
2
3
β
0
1
2
3
$\mathcal{D}$ #CC (3)


α
0
1
2
3
β
0
1
2
3
𝒟
#CC (3)

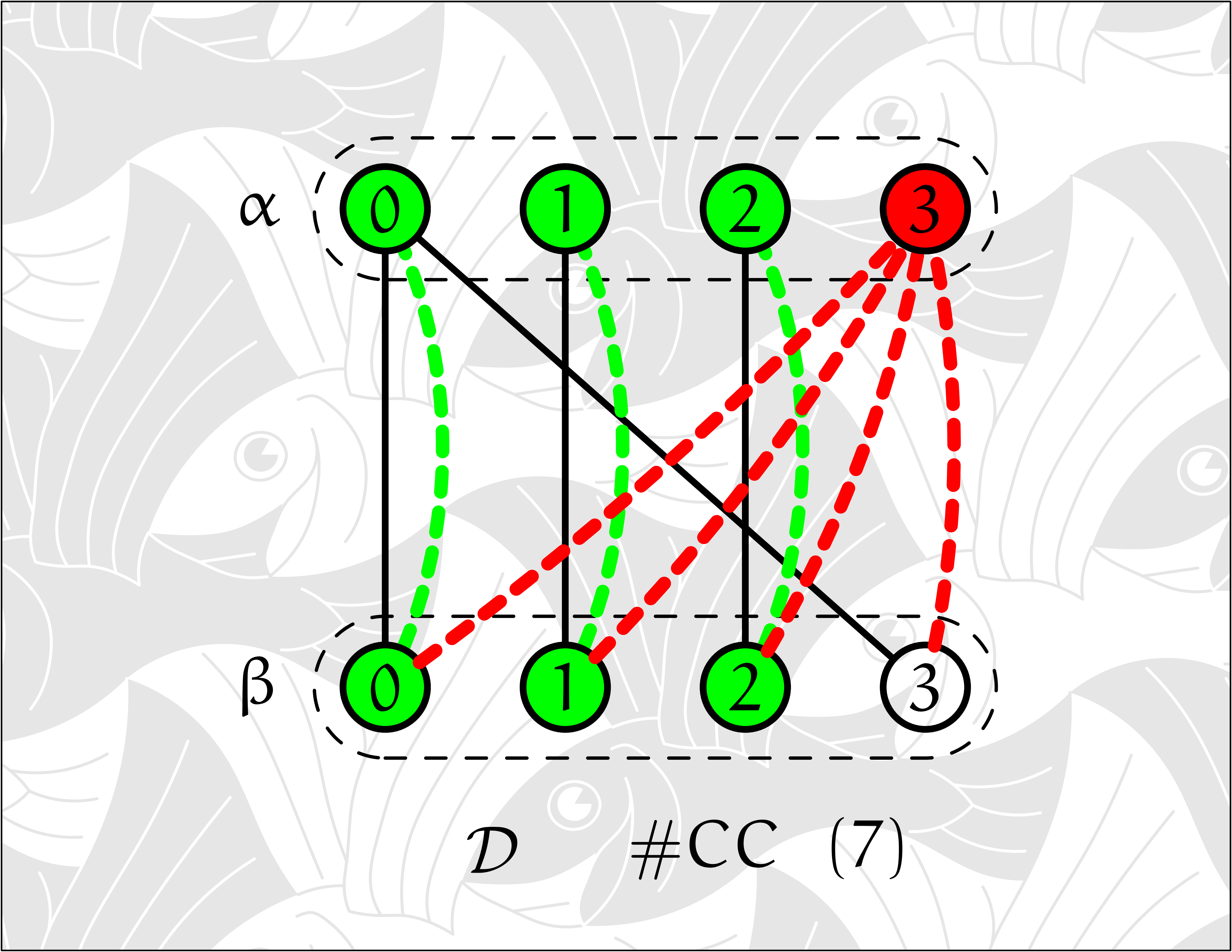

α
β
0
1
2
3
0
1
2
3
𝒟 #CC (4)


α
0
1
2
3
β
0
1
2
3
$\mathcal{D}$
#CC
(5)

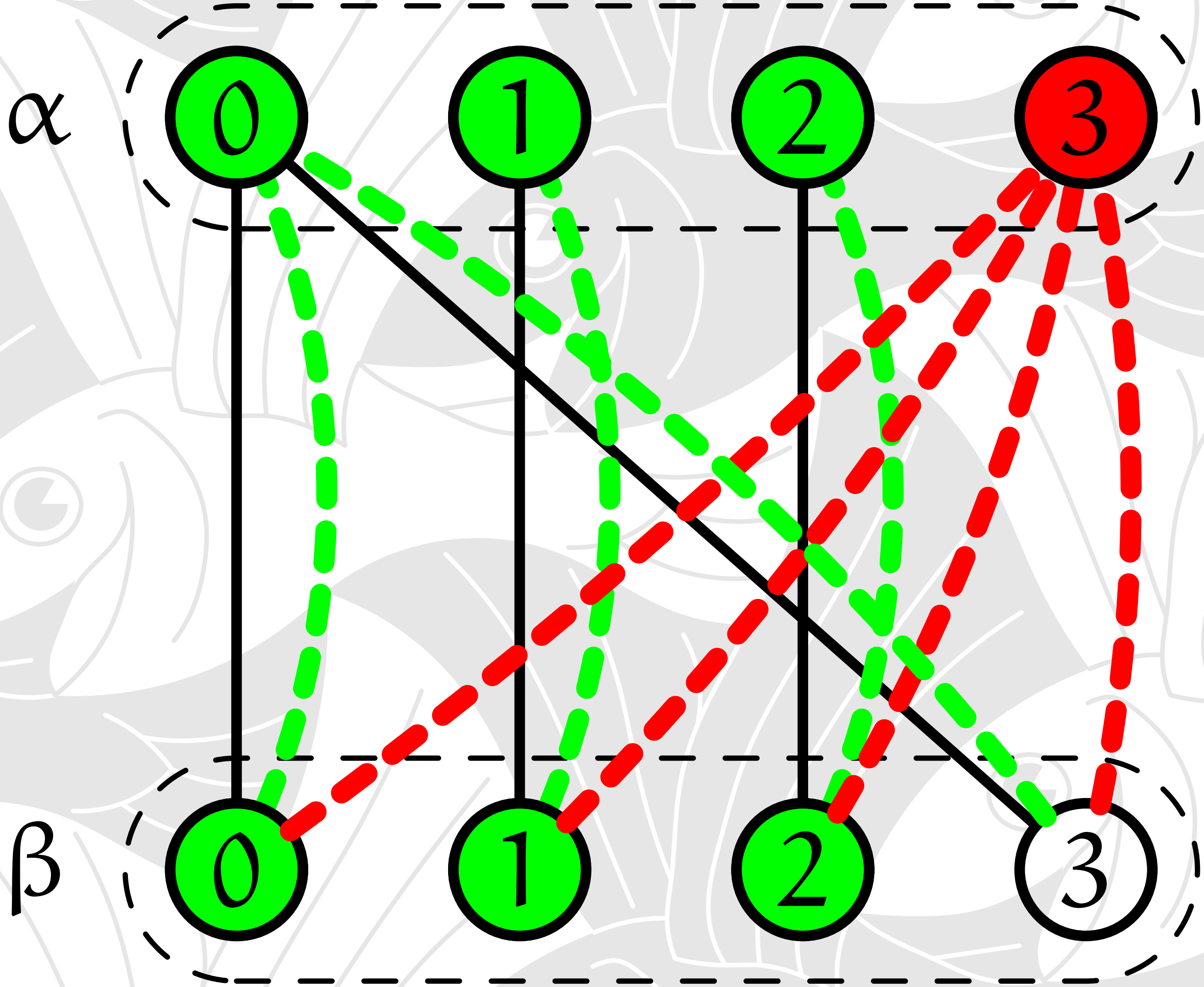

α
0
1
2
3
β
0
1
2
3
𝒟
#CC (6)



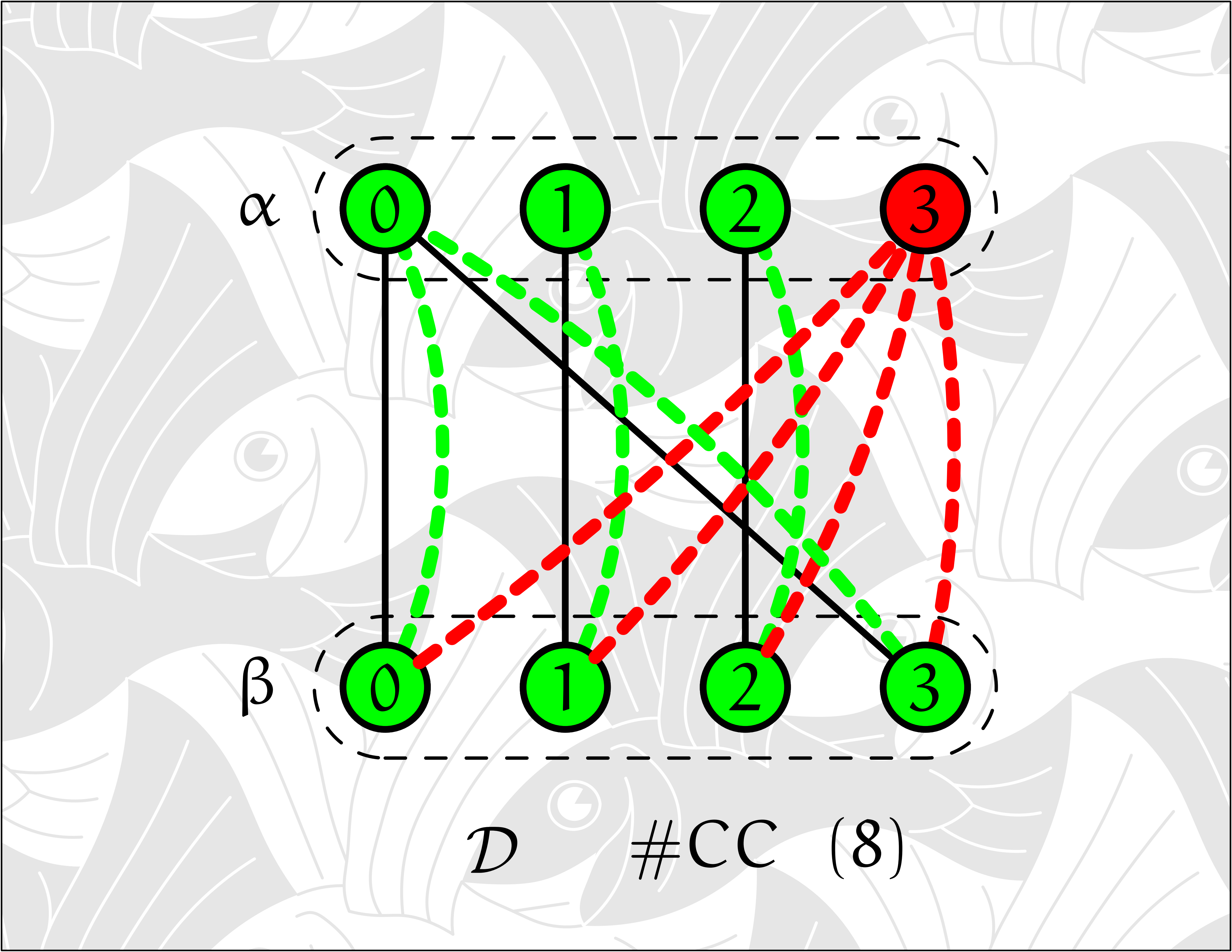
α
β
0
1
2
3
0
1
2
3
$\mathcal{D}$ #CC (7)



α
0
1
2
3
β
0
1
2
3
$\mathcal{D}$
#CC (7)



$\mathcal{D}$ #CC (8)



α
β
0
1
2
3
0
1
2
3
$\mathcal{D}$ #CC (8)

Case Study

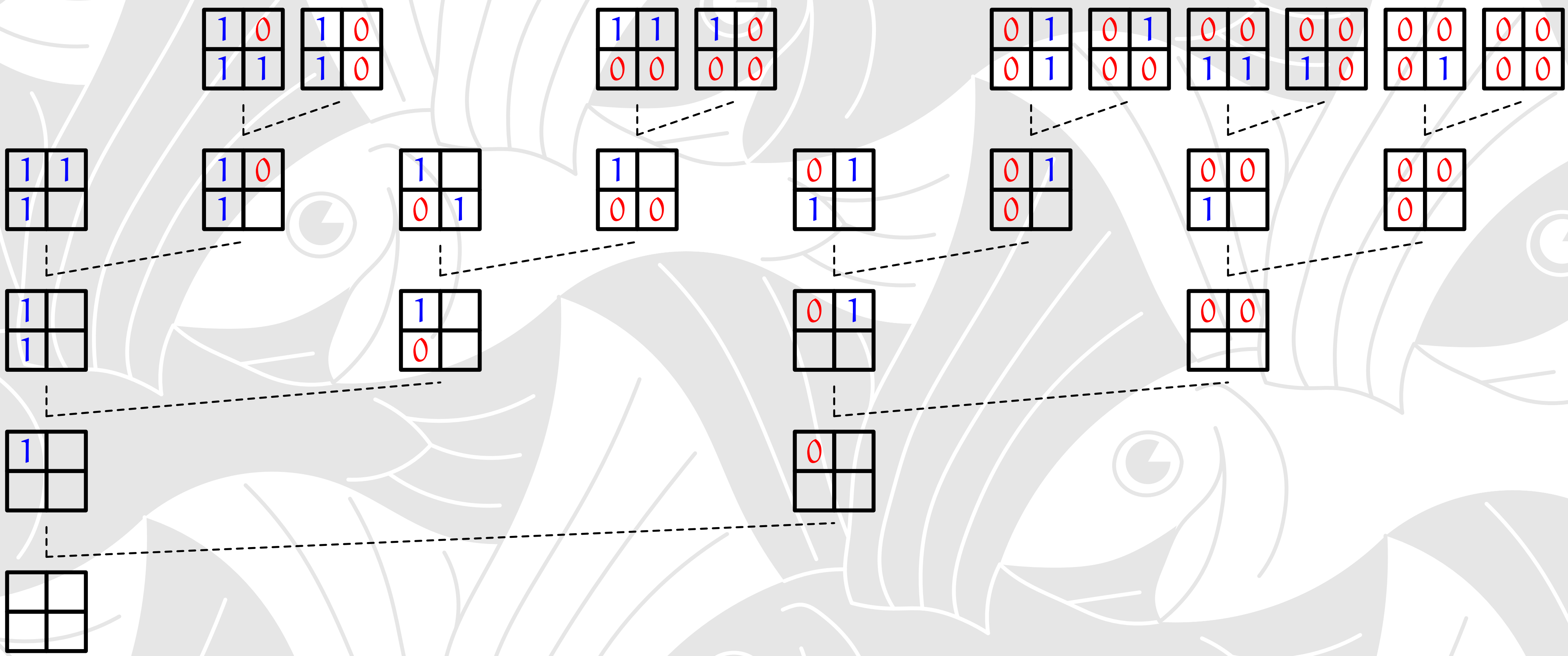
Definition 1. [Trace] *Let $\mathcal{A}$ be an arc-consistency algorithm, let M be an a by b constraint between α and β, and let*

$$M_{i_1 j_1}?, M_{i_2 j_2}?, \ldots, M_{i_l j_l}?$$

be the support-checks required by $\mathcal{A}$ to find the support of α and β. The trace *of M w.r.t. $\mathcal{A}$ is the sequence*

$$(\, i_1, j_1, M_{i_1 j_1} \,), (\, i_2, j_2, M_{i_2 j_2} \,), \ldots, (\, i_l, j_l, M_{i_l j_l} \,).$$

Traces of $\mathcal{L}$ for the Two by Two Case



Properties of Traces

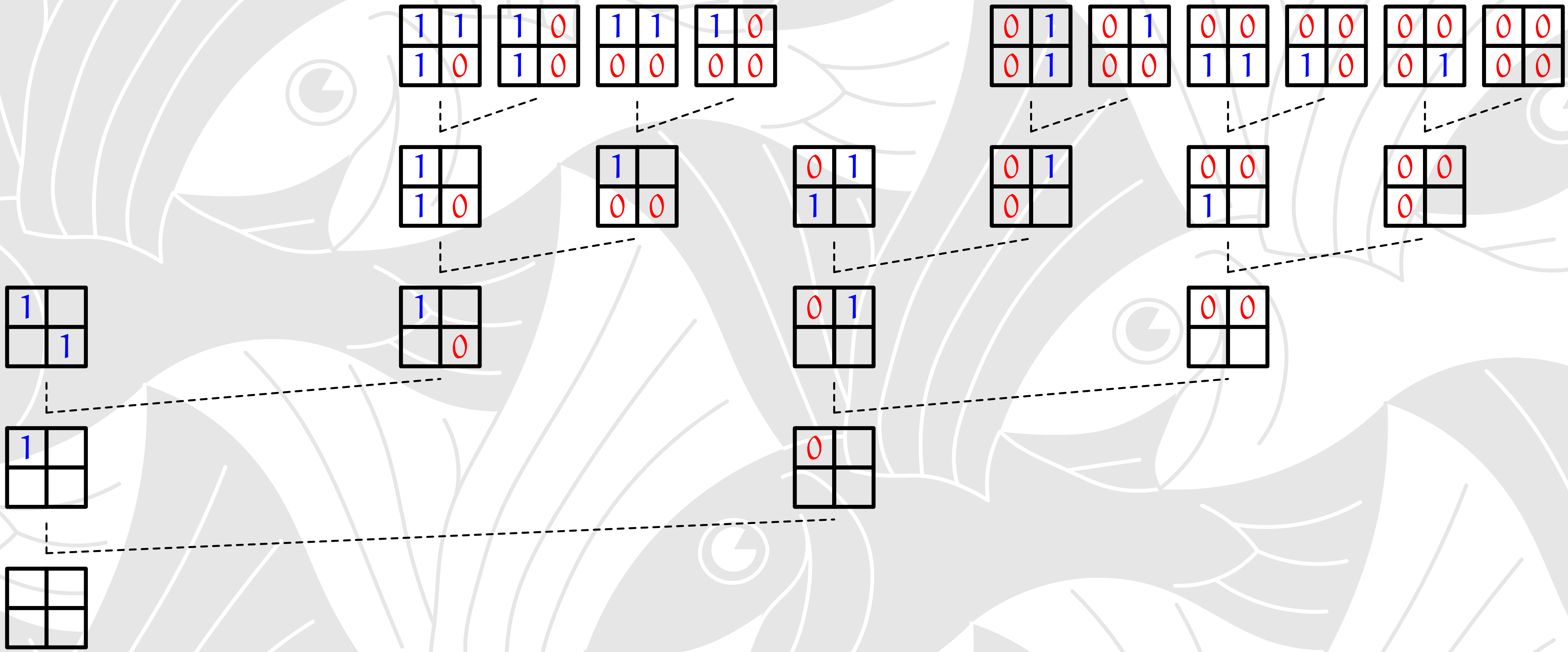
Let $\mathcal{A}$ be an arc-consistency algorithm which does not repeat support-checks, let $\mathfrak{t}$ be a trace of a constraint in $\mathbb{M}^{\mathfrak{ab}}$ w.r.t. $\mathcal{A}$, and let $\mathfrak{l}$ be the length of $\mathfrak{t}$.

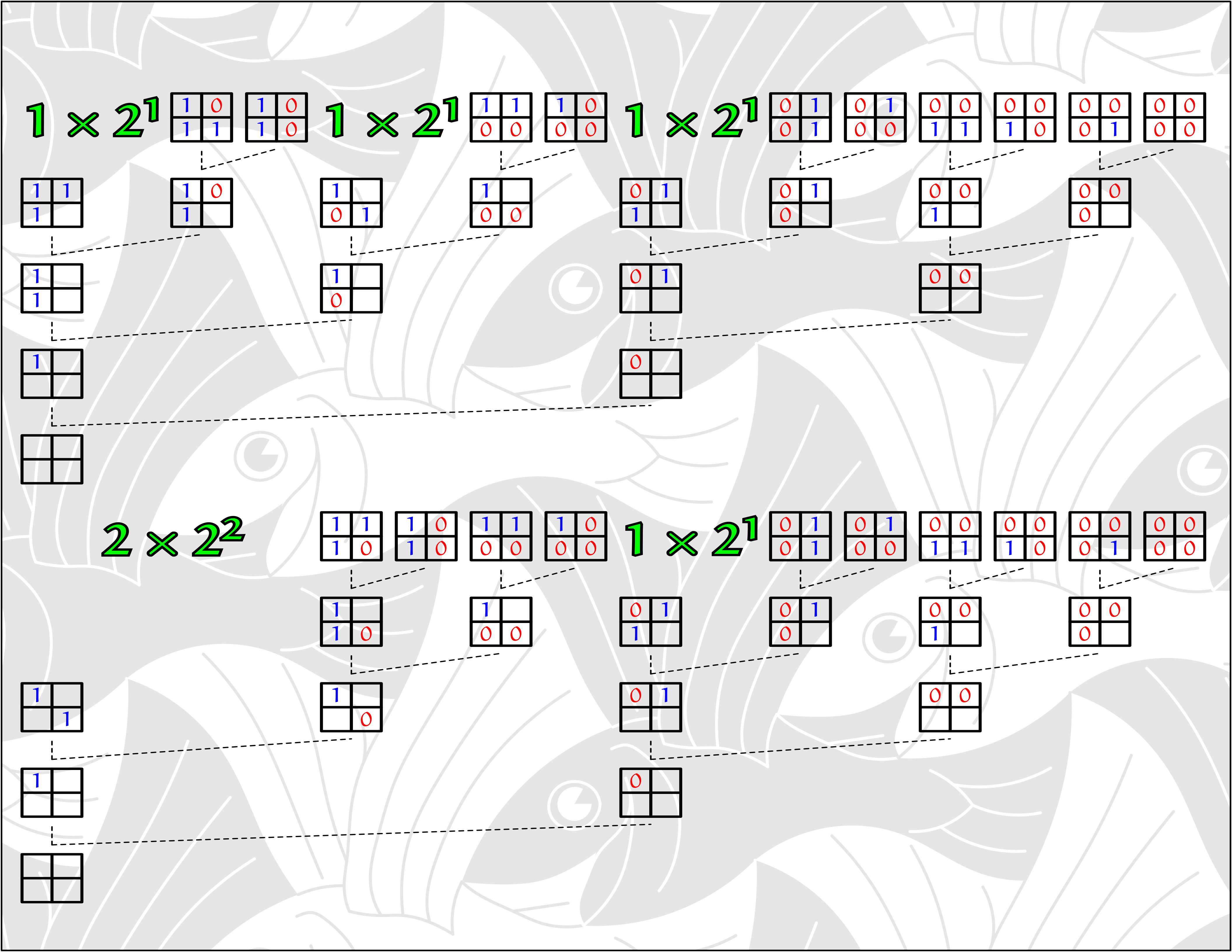
There are exactly $2^{\mathfrak{ab}-\mathfrak{l}}$ constraints in $\mathbb{M}^{\mathfrak{ab}}$ whose traces w.r.t. $\mathcal{A}$ are equal to $\mathfrak{t}$.

Theorem 1. [Trace Property] *Let* $\mathfrak{t}$ *be a trace of a constraint in* $\mathbb{M}^{\mathfrak{ab}}$ *w.r.t. some algorithm* $\mathcal{A}$*, and let* $\mathfrak{l}$ *be the length of* $\mathfrak{t}$*. The* average savings *of the constraints in* $\mathbb{M}^{\mathfrak{ab}}$ *whose trace w.r.t.* $\mathcal{A}$ *is equal to* $\mathfrak{t}$ *are given by:*

$$(\mathfrak{ab} - \mathfrak{l})2^{\mathfrak{ab}-\mathfrak{l}}/2^{\mathfrak{ab}} = (\mathfrak{ab} - \mathfrak{l})2^{-\mathfrak{l}}.$$

Traces of $\mathcal{D}$ for the Two by Two Case




1 × 2^1
1 × 2^1
1 × 2^1
2 × 2^2
1 × 2^1

Comparison for the Two by Two Case

Algorithm	Savings	Checks
$\mathcal{L}$	$3 \times 1 \times 2^1 = 6$	58
$\mathcal{D}$	$1 \times 1 \times 2^1 + 1 \times 2 \times 2^2 = 10$	54

A Lower Bound for $\mathrm{avg}_{\mathcal{L}}(a, b)$

$$(2-\epsilon)a + 2b + \mathbf{O}(1) + \mathbf{O}(a2^{-b}) \leq \mathrm{avg}_{\mathcal{L}}(a, b),$$

where

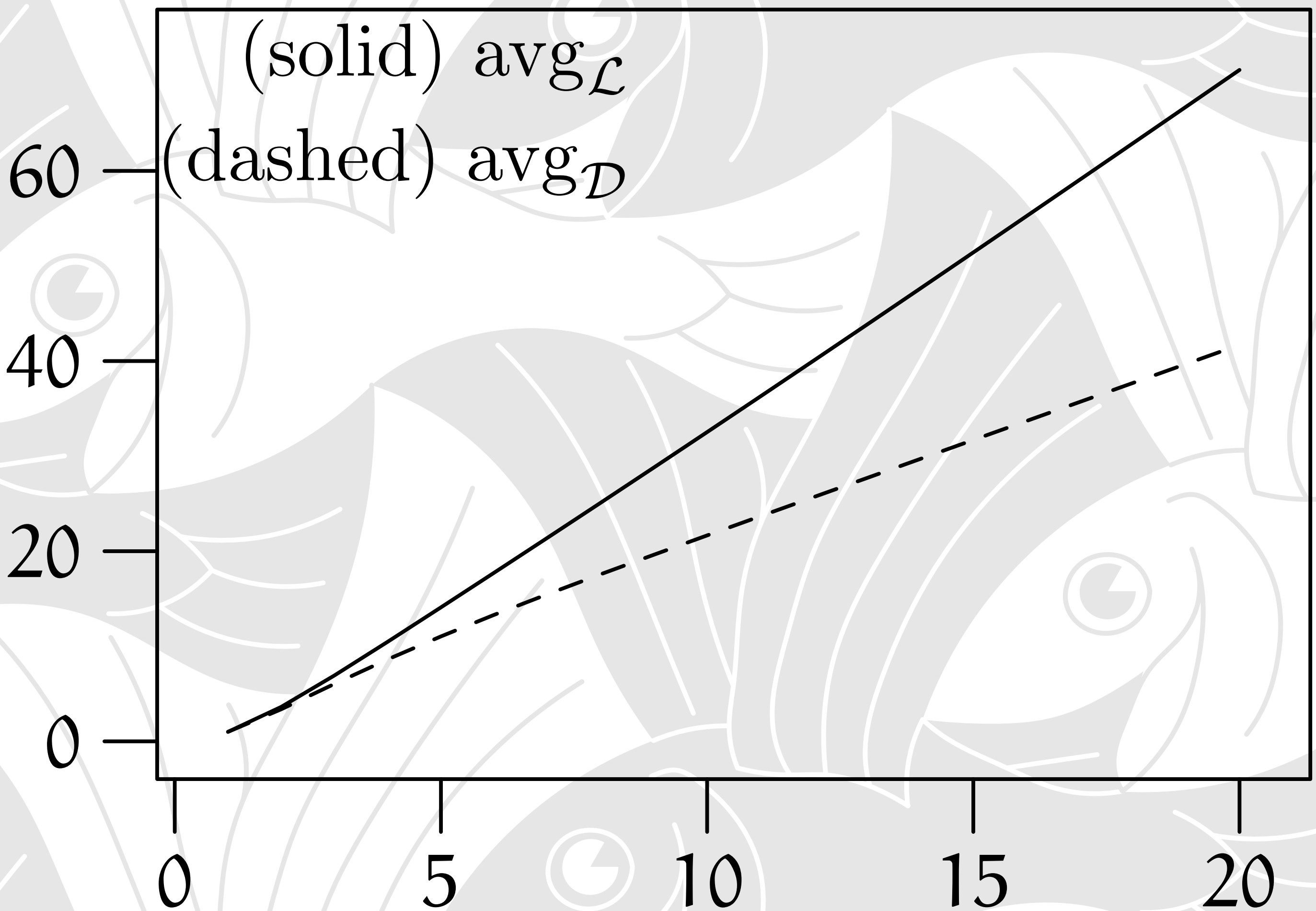
$$\epsilon = 2^{-s} + 2\sum_{k=0}^{s}\binom{s}{k}(-1)^k(2^{k+1}-1)^{-1}.$$

An Upper Bound for $\mathrm{avg}_{\mathcal{D}}(a, b)$

Let $a + b \geq 14$. Then

$$\begin{aligned}\mathrm{avg}_{\mathcal{D}}(a,b) \;\leq\; & 2\max(a,b) + 2 \\ & - (2\max(a,b) + \min(a,b))2^{-\min(a,b)} \\ & - (3\max(a,b) + 2\min(a,b))2^{-\max(a,b)}.\end{aligned}$$

Comparison of $\mathcal{L}$ and $\mathcal{D}$



Discussion

- First attempt to study average time-complexity of arc-consistency algorithms. http://www.ucc.ie/~dongen/papers/pdf/UCC/00/TR0004.pdf.

- Arc-consistency algorithms should prefer double-support checks at domain level.

- $\mathcal{D}$ is better on average than $\mathcal{L}$.

- Evidence has been presented that $\mathcal{D}$ is "good."

Future Work

1. Incorporate the double-support heuristic into an algorithm which does not repeat support-checks.

2. Study the average time-complexity of $\mathcal{L}$ and $\mathcal{D}$ if there are more than two variables.

3. Generalise the notion of double-support check to k-consistency, where $k > 2$.

Questions Anybody?