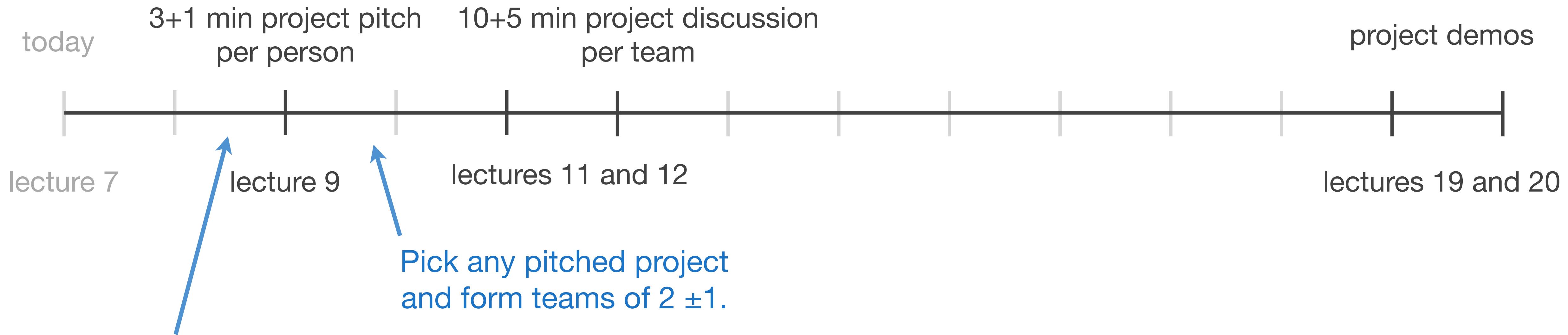


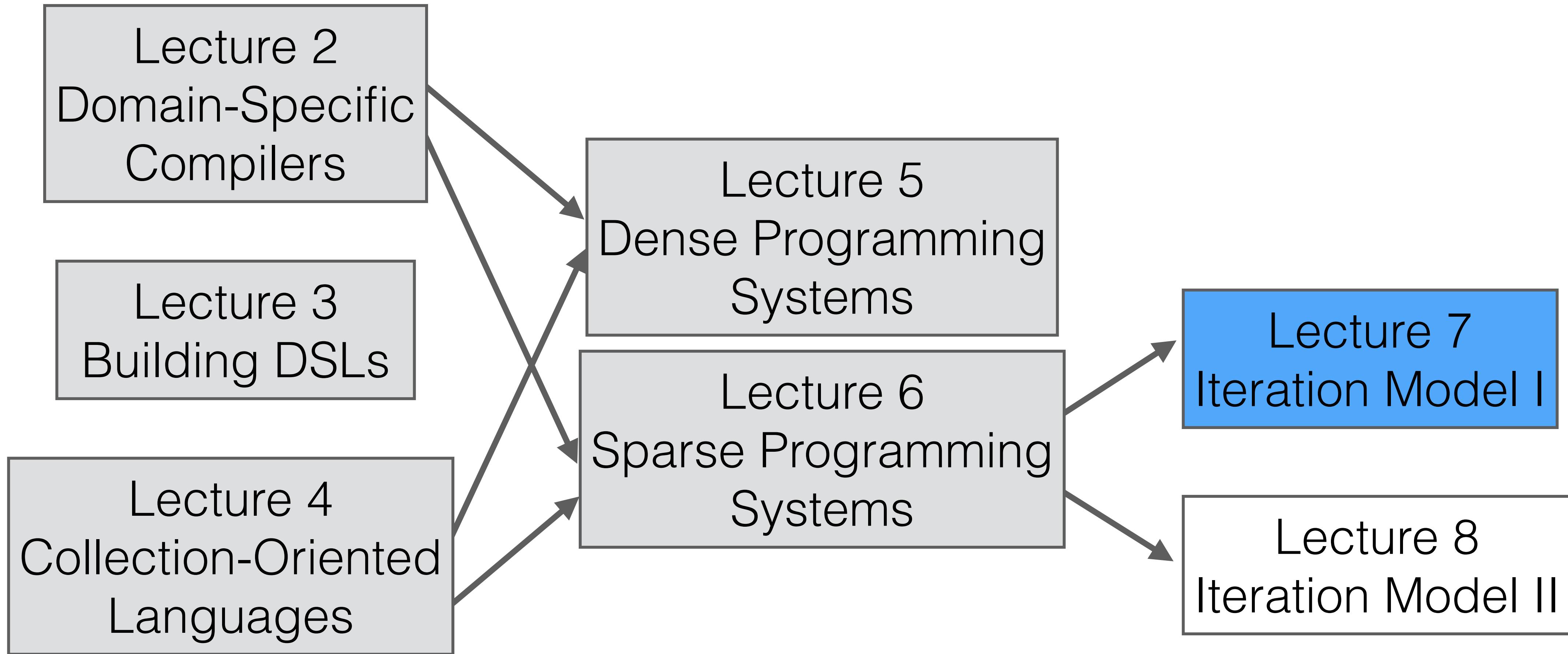
Lecture 7 – Sparse Iteration Model I

Stanford CS343D (Winter 2024)
Fred Kjolstad

Course Project



Each person contributes one
pitch slide to a google slide deck.
These pitches are not binding.



Overview of topics

Lecture 7

- Data representation
- Iteration spaces
- Iteration graph IR
- Iteration lattices to represent coiteration

Lecture 8

- Concrete index notation IR
- Code generation algorithm
- Derived iteration spaces
- Optimizing transformations

Sparse Tensor Algebra Compilation

Tensor Index Notation Expression

$$\begin{array}{lll} A = Bc + a & a = Bc \\ A = B \odot C & A = B + C & a = \alpha Bc + \beta a \\ A = BCd & A = \alpha B & A = 0 \quad A = BC \\ A_{ij} = \sum_{kl} B_{ikl}C_{lj}D_{kj} & A = B^T & a = B^T Bc \\ A_{ijk} = \sum_l B_{ikl}C_{lj} & A_{ik} = \sum_j B_{ijk}c_j & A_{kj} = \sum_{il} B_{ikl}C_{lj}D_{ij} \\ C = \sum_{ijkl} M_{ij}P_{jk}\overline{M_{lk}}\overline{P_{il}} & A_{ij} = (\sum_k B_{ijk}C_{ijk}) + D_{ij} \\ a = \sum_{ijklmnp} M_{ij}P_{jk}M_{kl}P_{lm}\overline{M_{nm}}P_{no}\overline{M_{po}}\overline{P_{ip}} \end{array}$$

Formats

Schedule

Sparse Tensor Algebra
Compiler (taco)

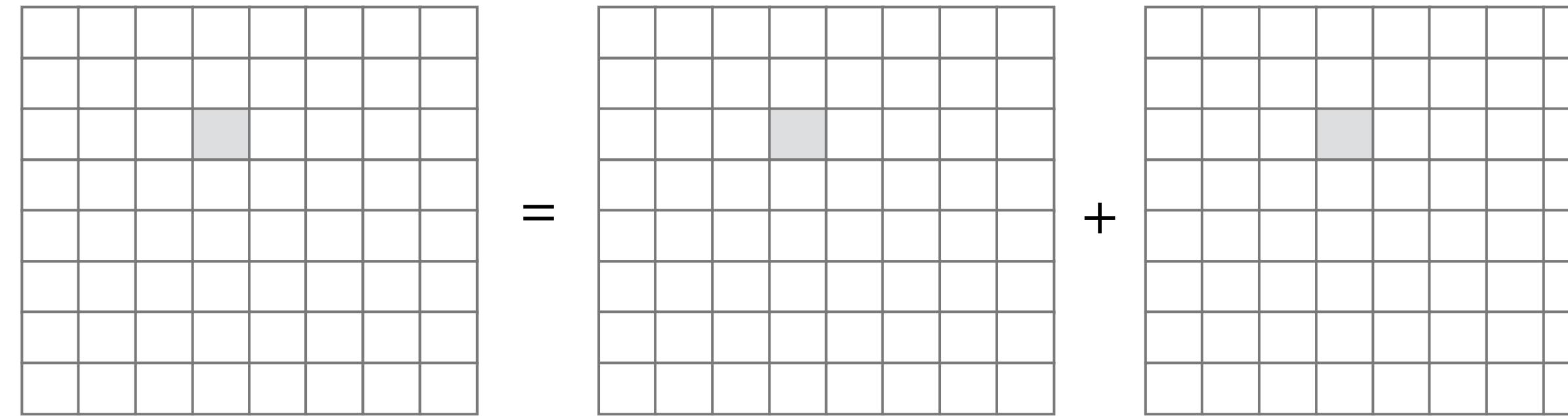


DSAs



Tensor index notation for expressing functionality

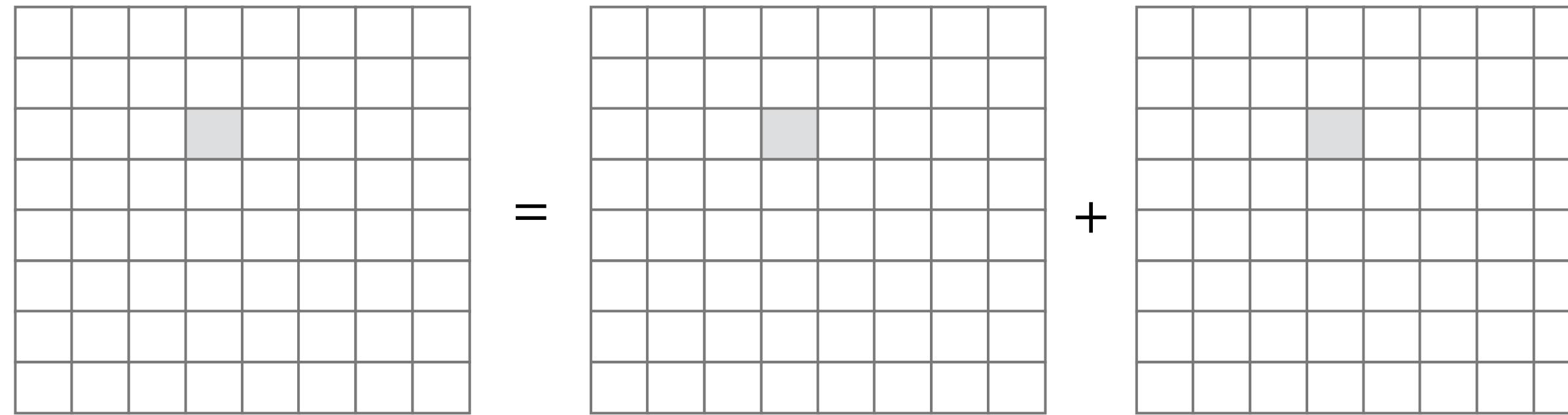
$$A_{ij} = B_{ij} + C_{ij}$$



element-wise

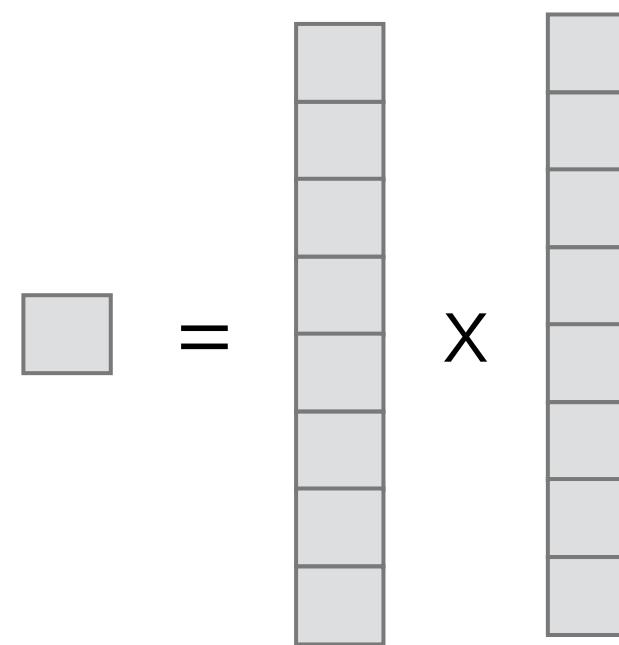
Tensor index notation for expressing functionality

$$A_{ij} = B_{ij} + C_{ij}$$



element-wise

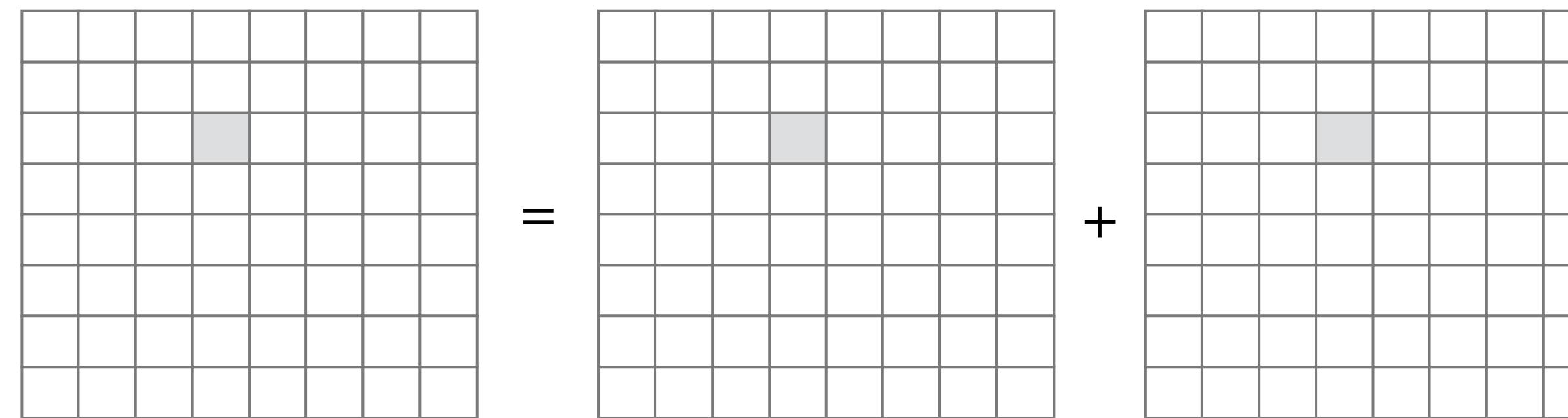
$$\alpha = \sum_i b_i c_i$$



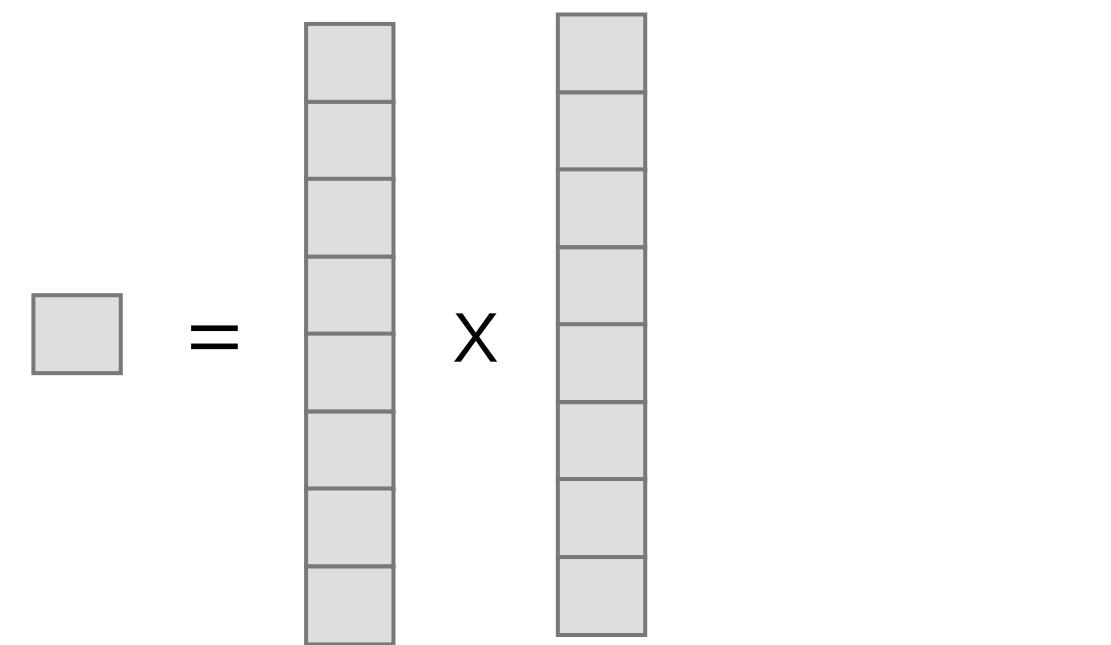
reduction over i

Tensor index notation for expressing functionality

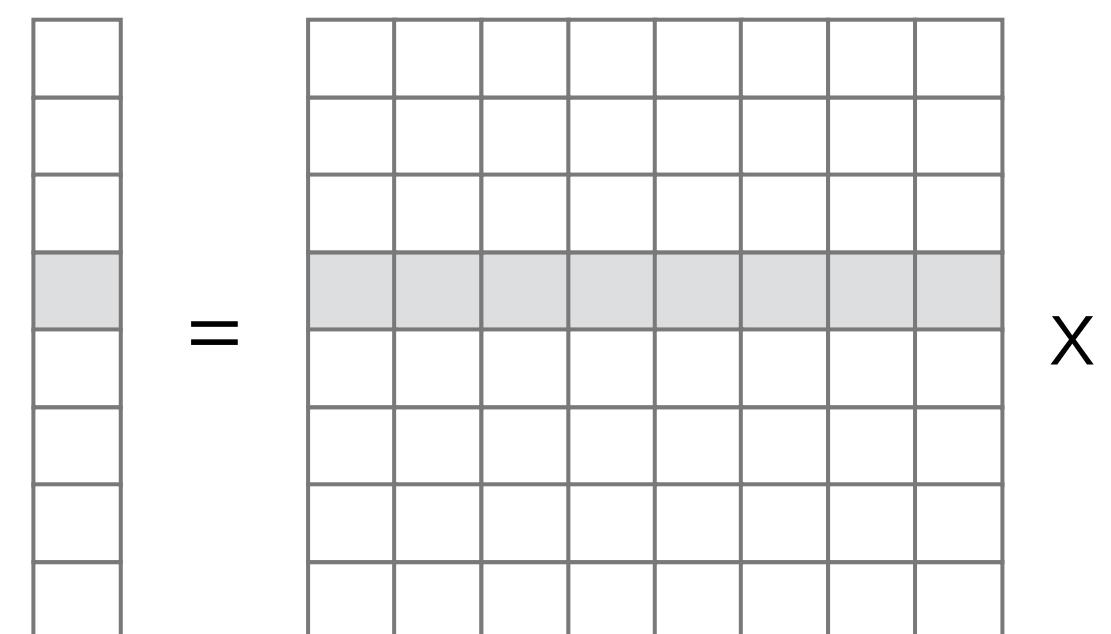
$$A_{ij} = B_{ij} + C_{ij}$$



$$\alpha = \sum_i b_i c_i$$

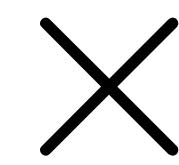


$$a_i = \sum_j B_{ij} c_j$$



Generates fast code for any tensor index notation expression with the given formats and schedule

$$\begin{aligned}
 & a = Bc \\
 & a = Bc + a \\
 & a = Bc + b \quad A = B + C \quad a = \alpha Bc + \beta a \\
 & \quad a = B^T c \quad A = \alpha B \quad a = B(c + d) \\
 & a = B^T c + d \quad A = B + C + D \quad A = BC \\
 & A = B \odot C \quad a = b \odot c \quad A = 0 \quad A = B \odot (CD) \\
 & \quad A = BCd \quad A = B^T \quad a = B^T Bc \\
 & a = b + c \quad A = B \quad K = A^T CA \\
 & A_{ij} = \sum_{kl} B_{ikl} C_{lj} D_{kj} \quad A_{kj} = \sum_{il} B_{ikl} C_{lj} D_{ij} \\
 & \quad A_{lj} = \sum_{ik} B_{ikl} C_{ij} D_{kj} \quad A_{ij} = \sum_k B_{ijk} c_k \\
 & A_{ijk} = \sum_l B_{ikl} C_{lj} \quad A_{ik} = \sum_j B_{ijk} c_j \\
 & \quad A_{jk} = \sum_i B_{ijk} c_i \quad A_{ijl} = \sum_k B_{ikl} C_{kj} \\
 & C = \sum_{ijkl} M_{ij} P_{jk} \overline{M_{lk}} \overline{P_{il}} \quad \tau = \sum_i z_i (\sum_j z_j \theta_{ij}) (\sum_k z_k \theta_{ik}) \\
 & \quad a = \sum_{ijklmnop} M_{ij} P_{jk} M_{kl} P_{lm} \overline{M_{nm}} P_{no} \overline{M_{po}} \overline{P_{ip}}
 \end{aligned}$$



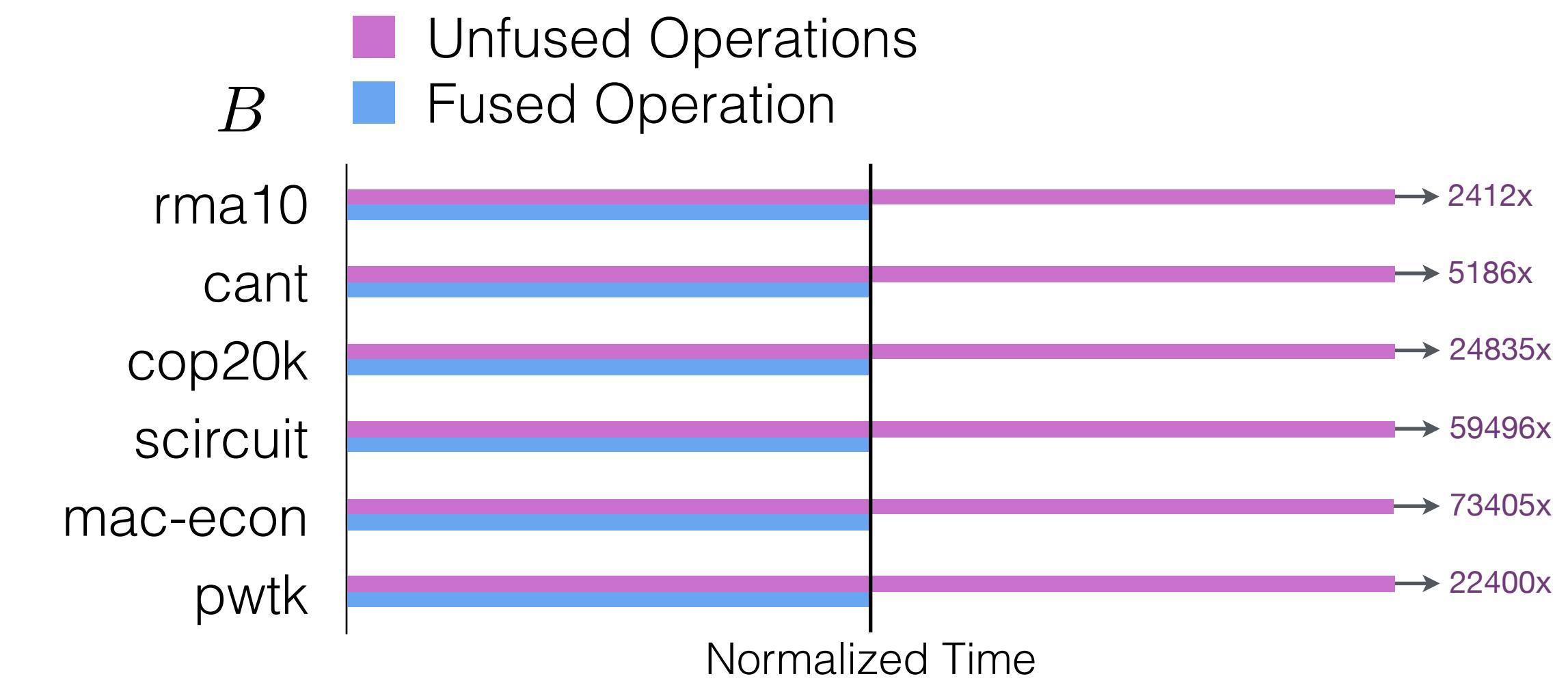
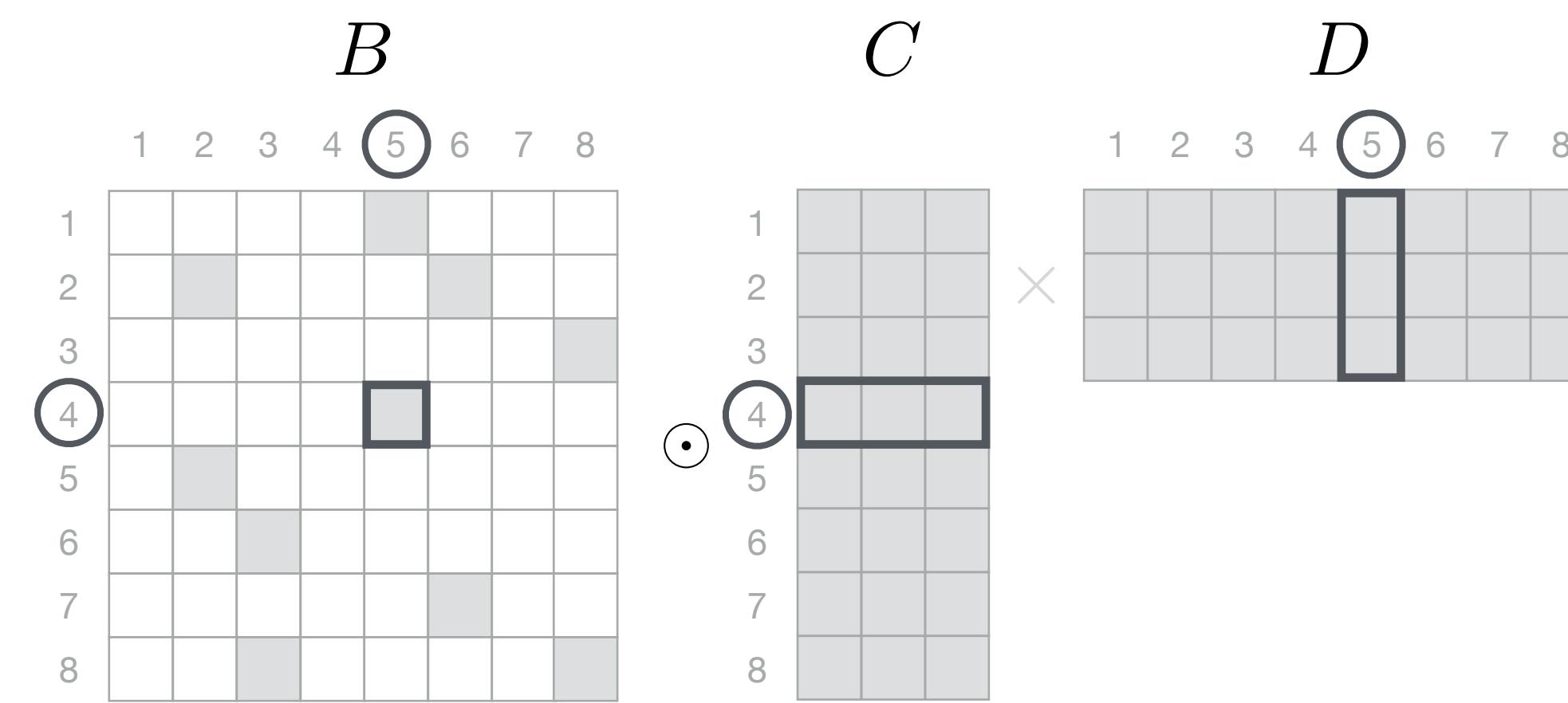
Dense Matrix				
CSR	DCSR	BCSR		
COO	ELLPACK	CSB	CPU	
Blocked COO		CSC	GPUs	TPUs
DIA	Blocked DIA	DCSC		
Sparse vector	Hash Maps			
Coordinates				
CSF	Dense Tensors			
	Blocked Tensors			

×

×

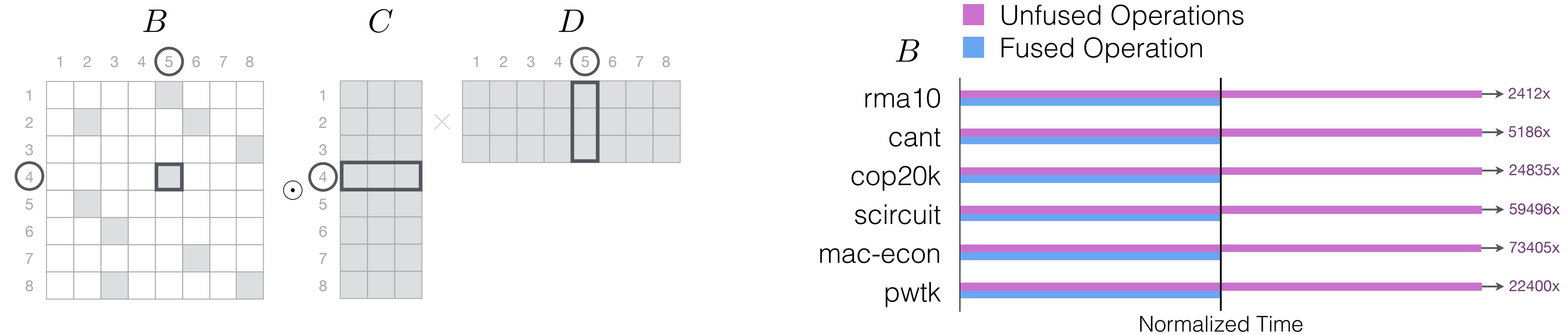
Compound expressions matter for performance

$$A = B \odot (CD)$$



Compound expressions matter for performance

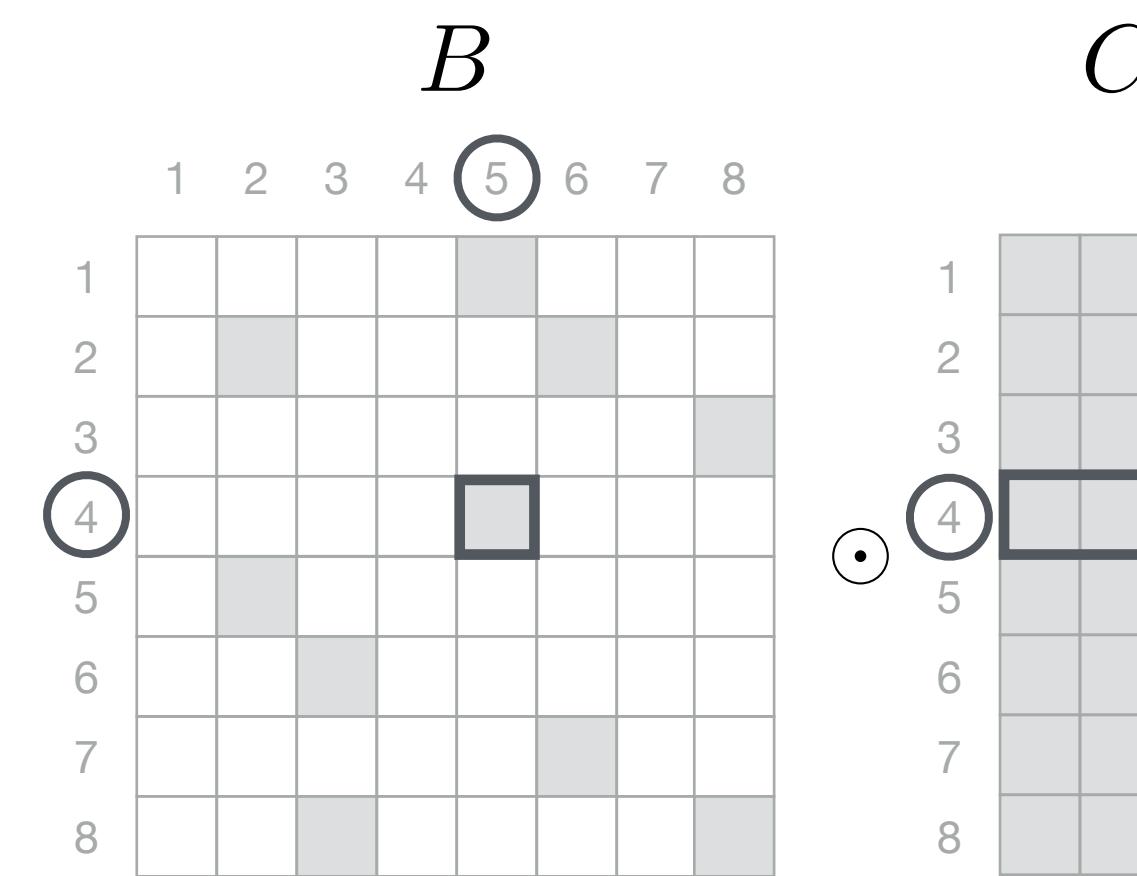
$$A = B \odot (CD)$$



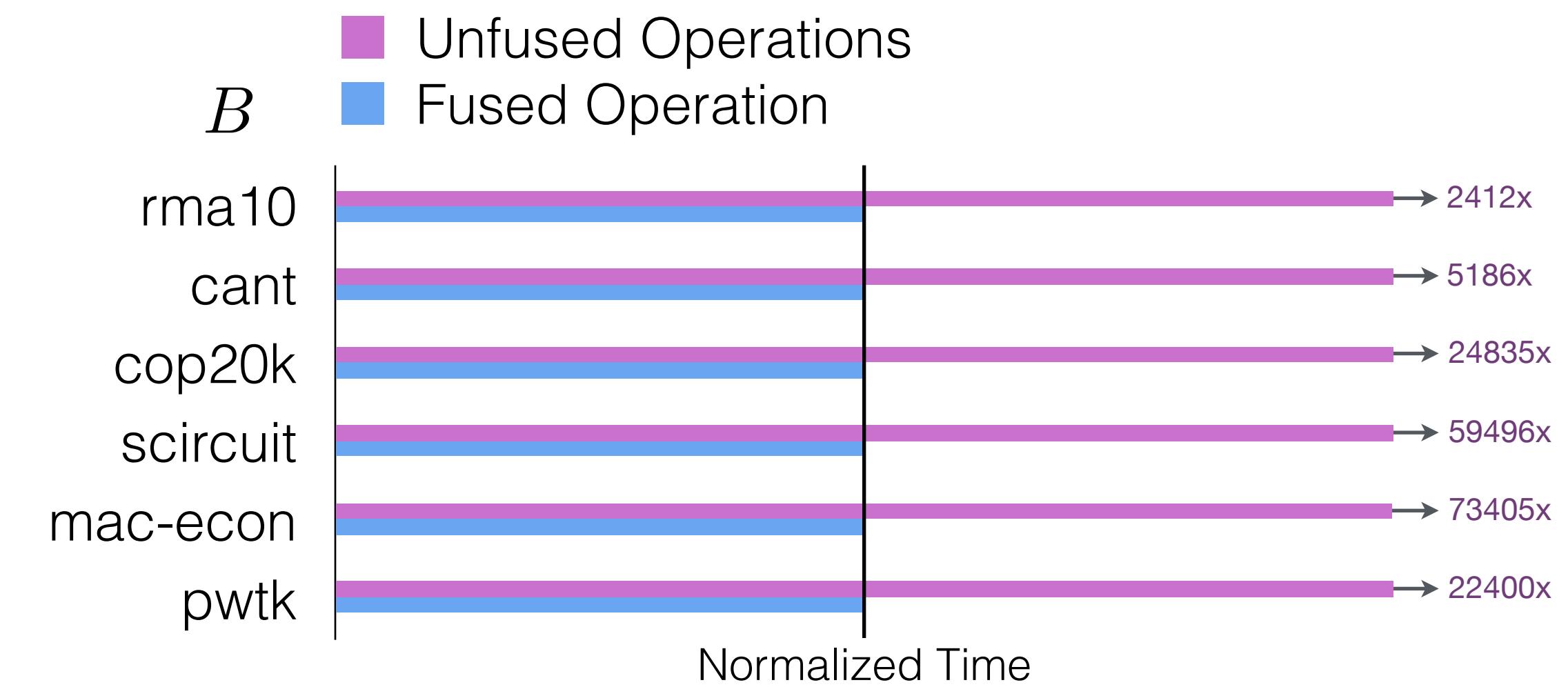
Unfused: $\Theta(n^2k)$

Compound expressions matter for performance

$$A = B \odot (CD)$$



$$\begin{matrix} & \times \\ C & \end{matrix}$$

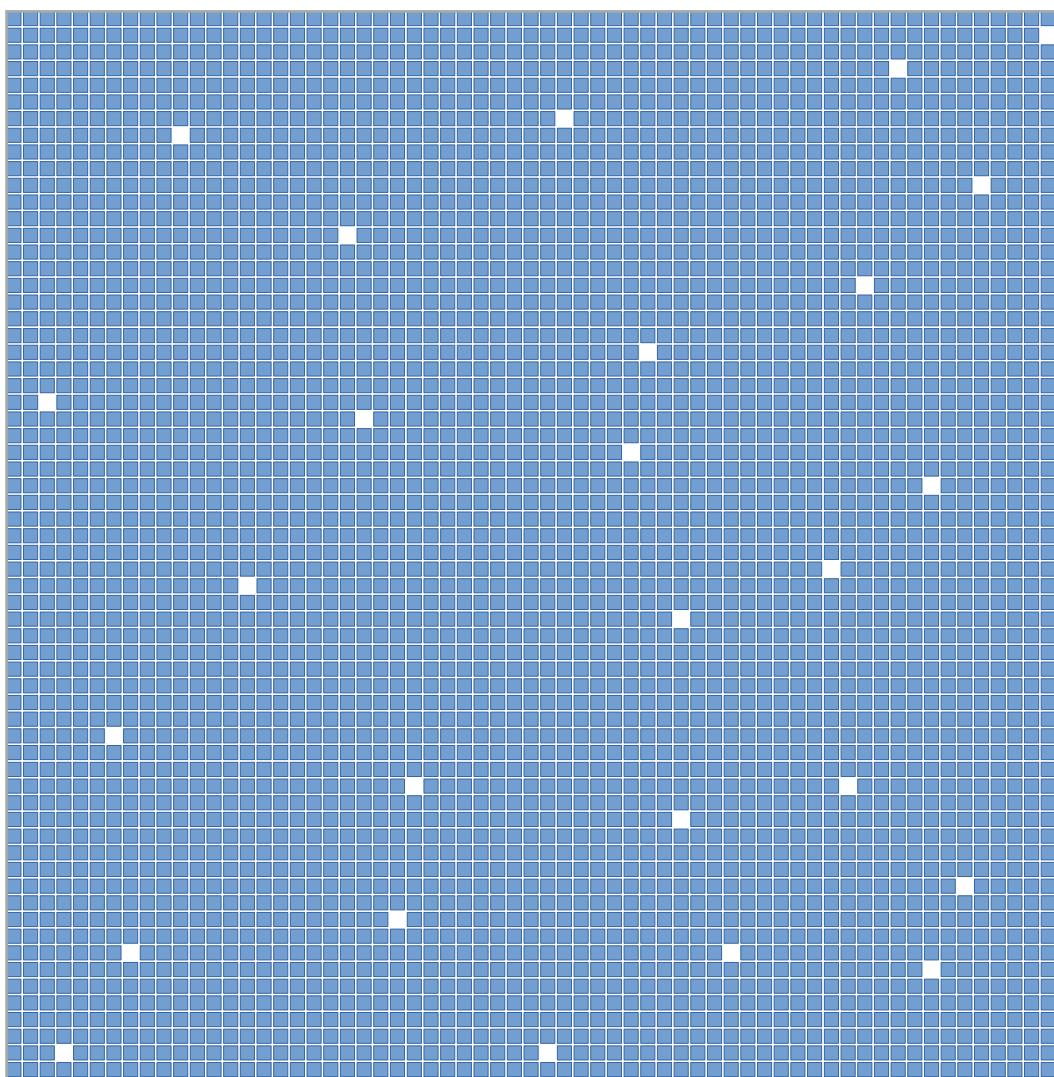


Unfused: $\Theta(n^2k)$

Fused: $\Theta(\text{nnz}_B \cdot k)$

Formats matter for performance

Dense Matrix



Formats

Best performance

Dense

List of Rows

CSR

DCSR

$$y = Ax$$



Normalized time

Formats matter for performance

Thermal Matrix



Formats

Best performance

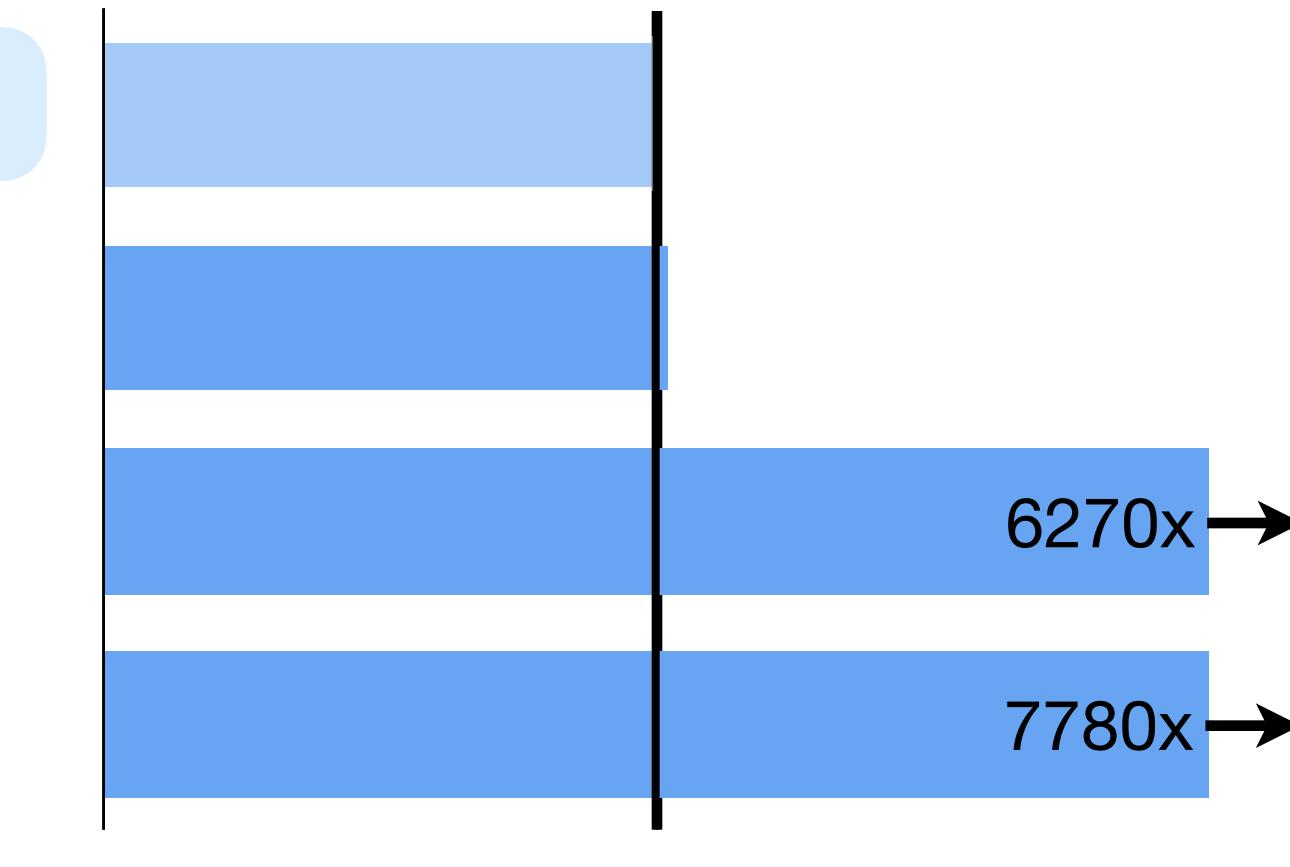
CSR

DCSR

Dense

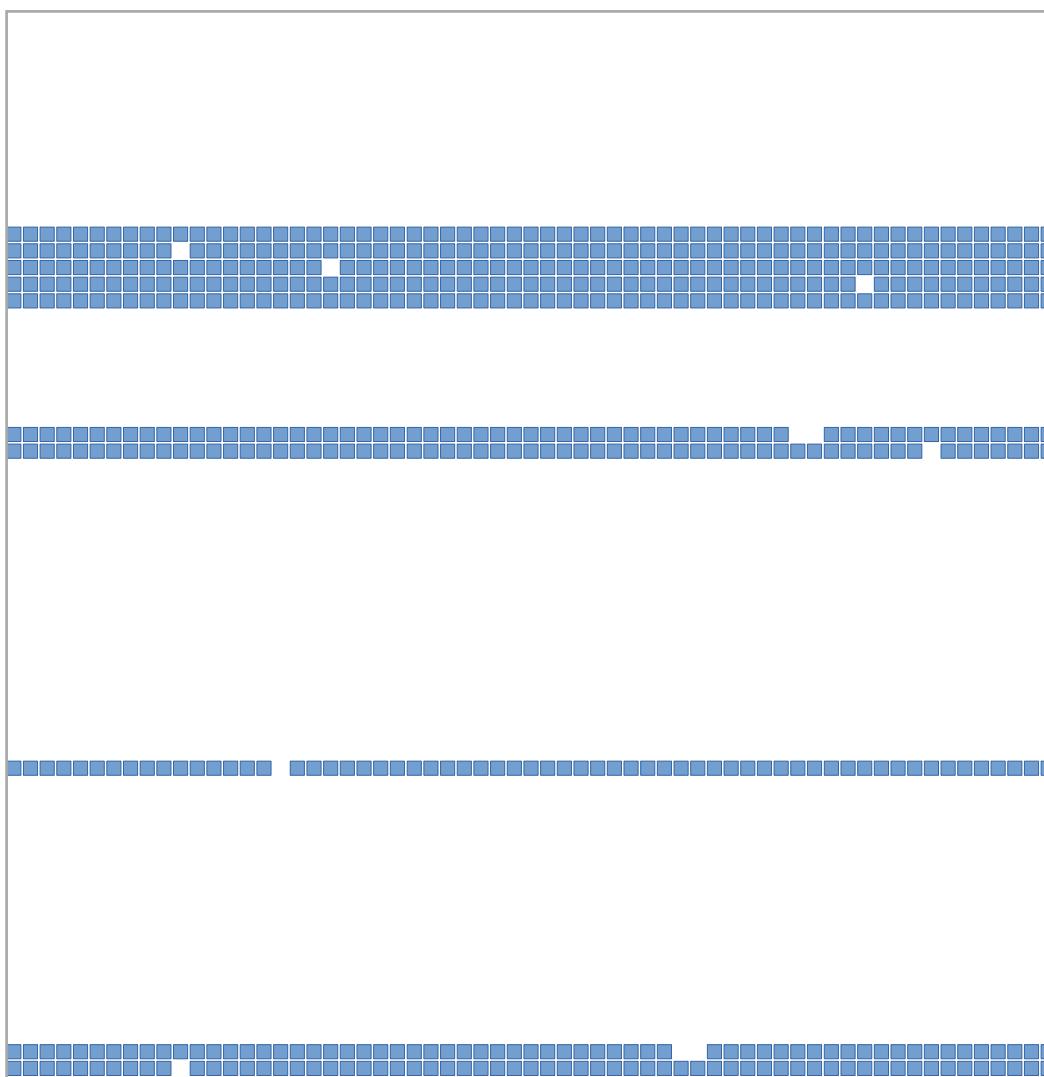
List of Rows

$$y = Ax$$



Formats matter for performance

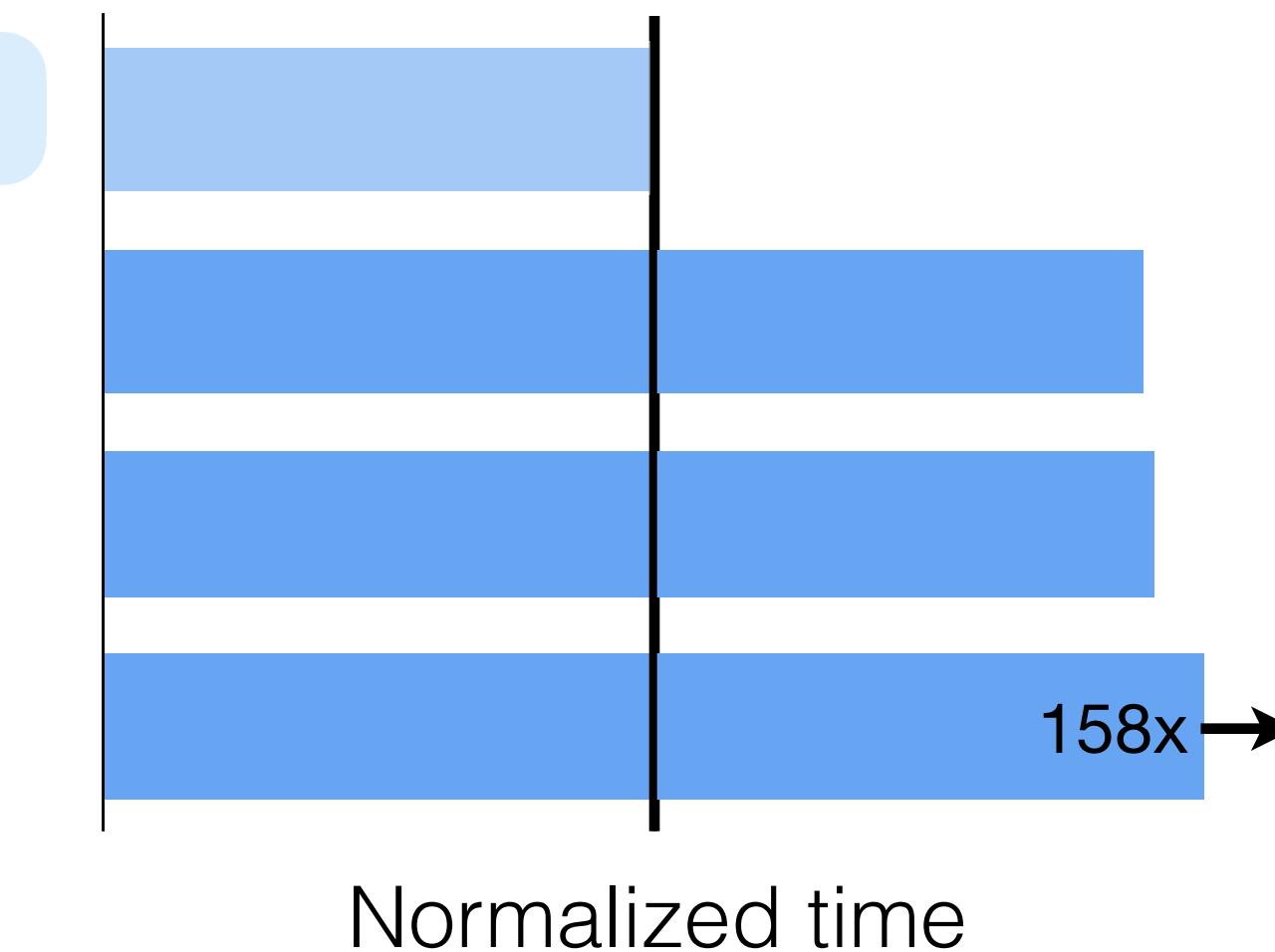
Row-sliced Matrix



Formats

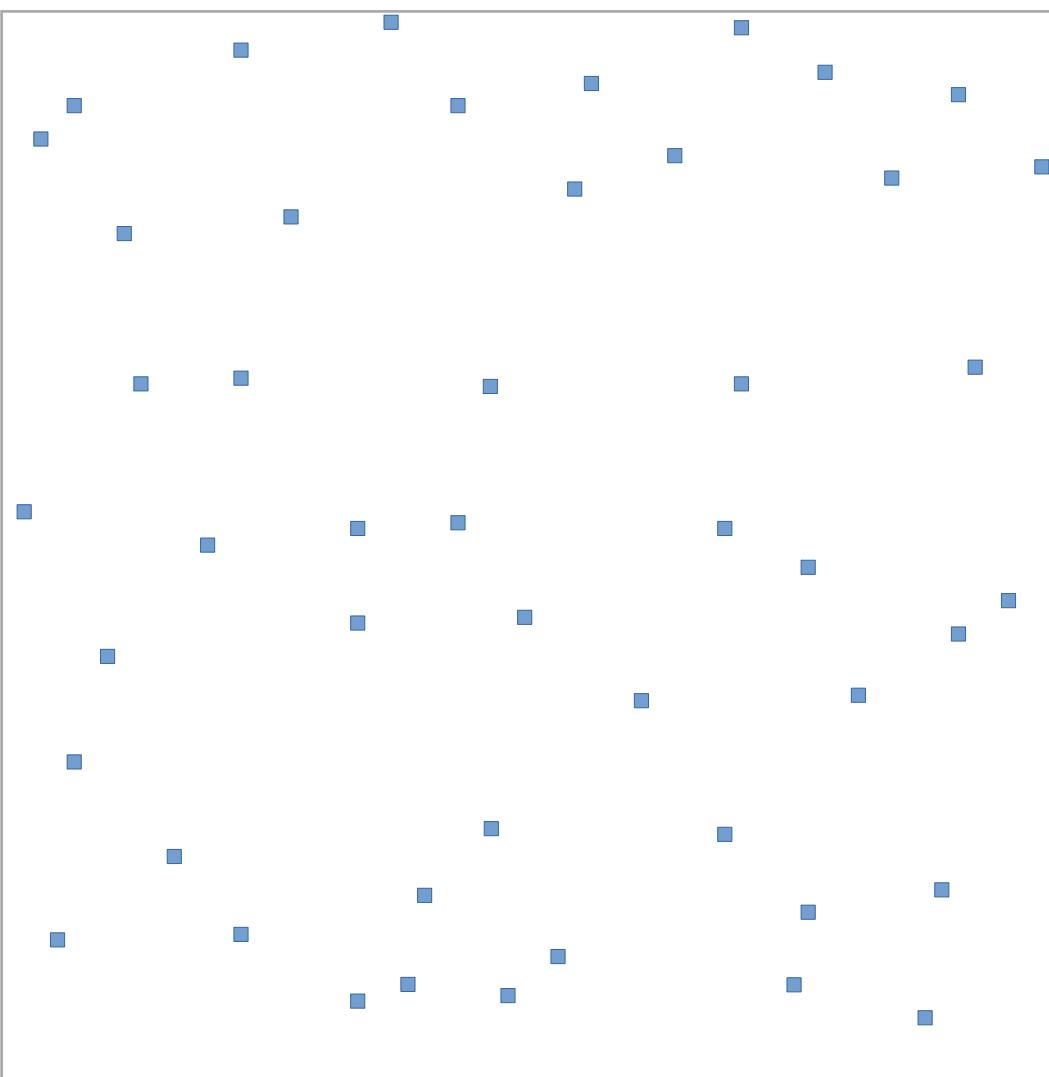
Best performance
List of Rows
DCSR
CSR
Dense

$$y = Ax$$



Formats matter for performance

Hypersparse Matrix



Formats

Best performance

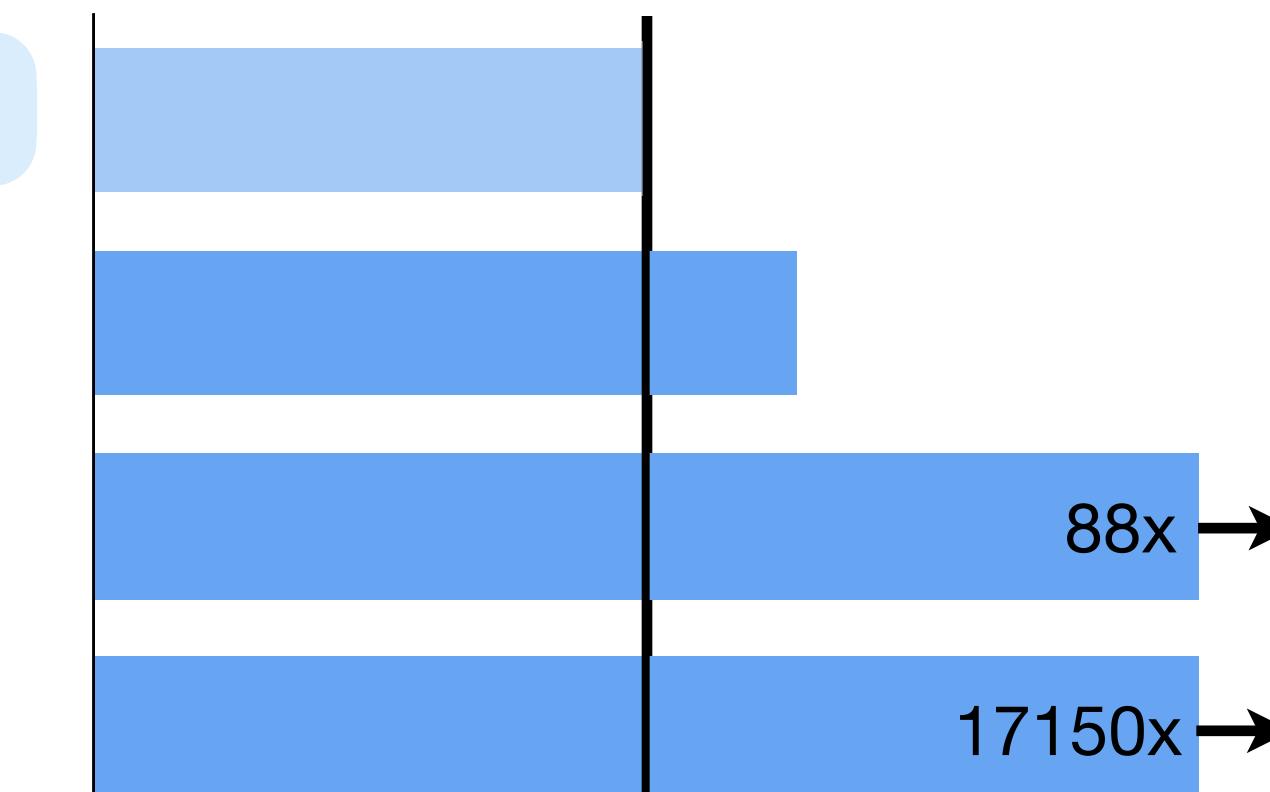
DCSR

CSR

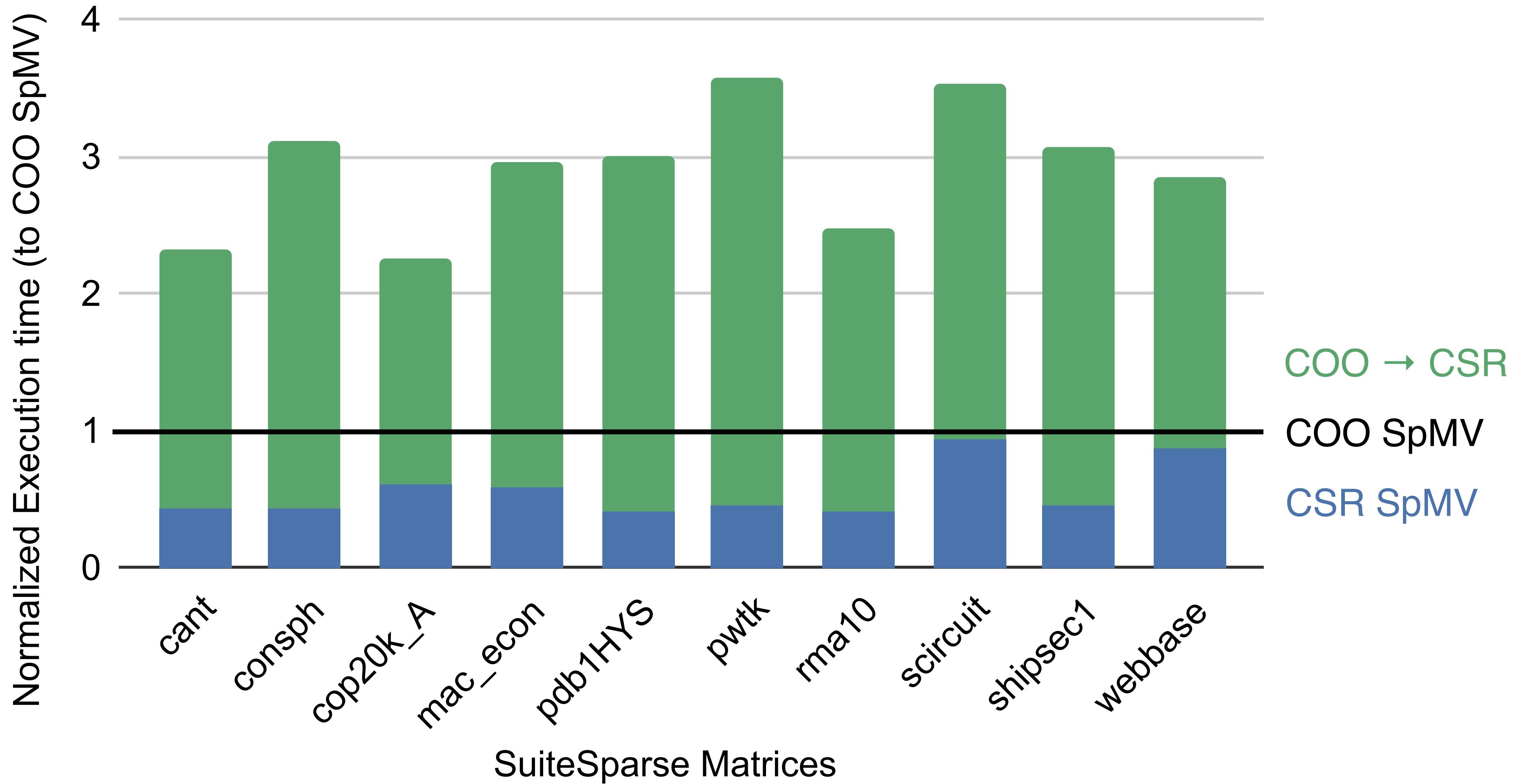
List of Rows

Dense

$$y = Ax$$

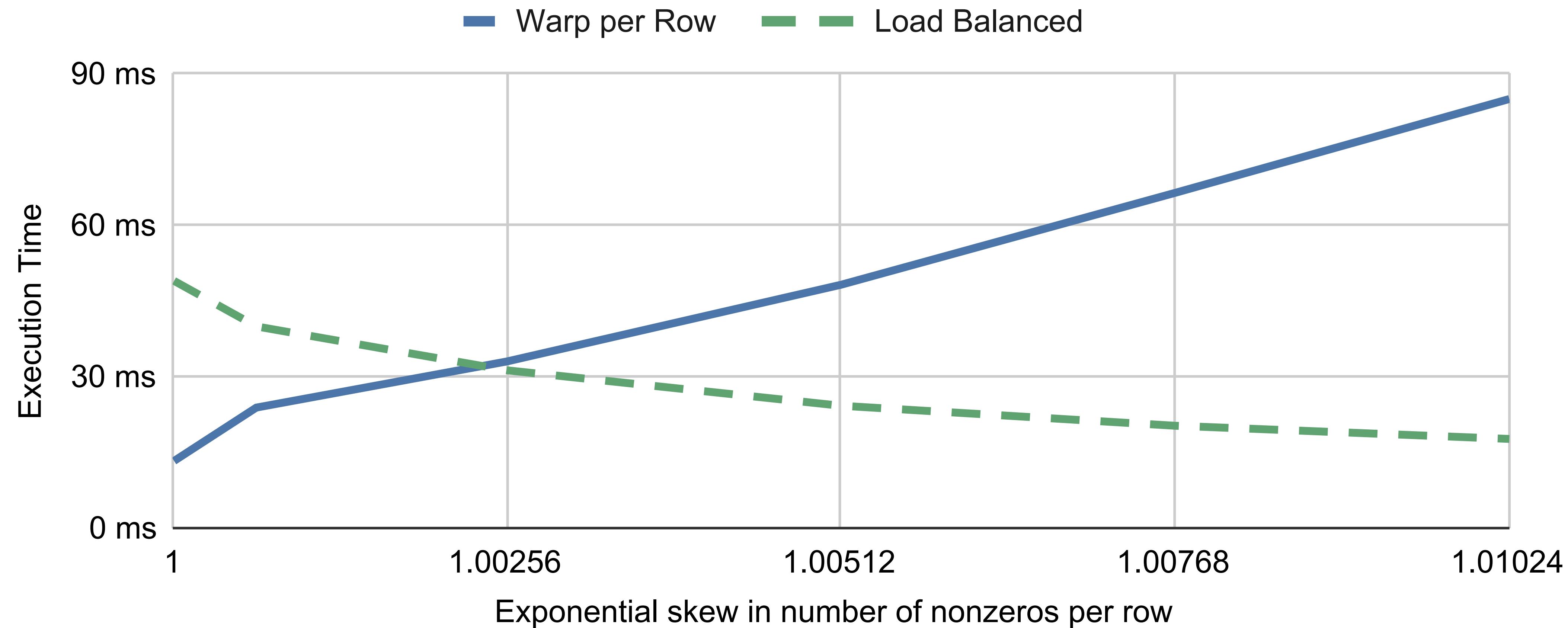


CSR vs COO



Schedules matter for performance

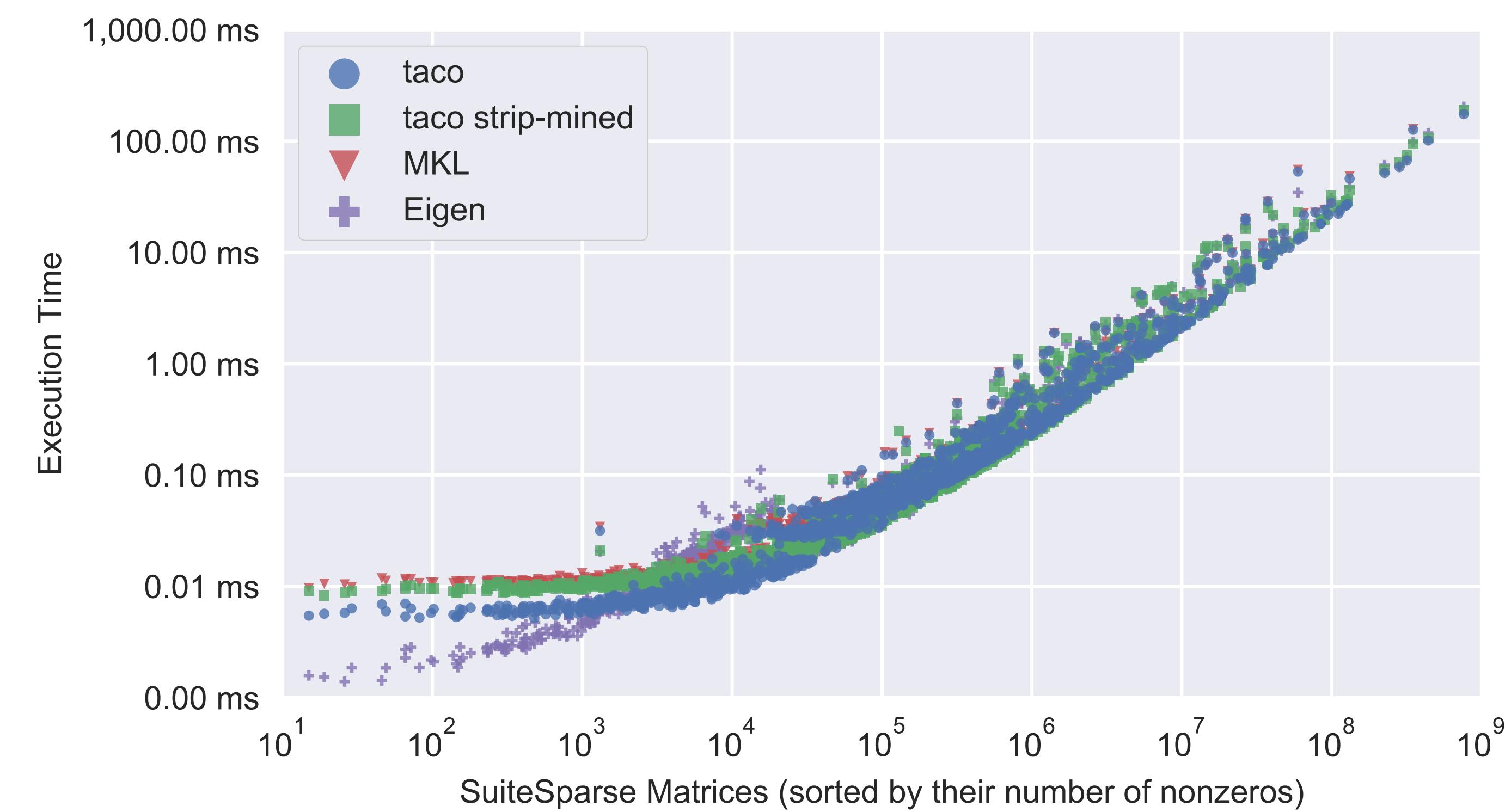
$$y = Ax \text{ (CPU)}$$



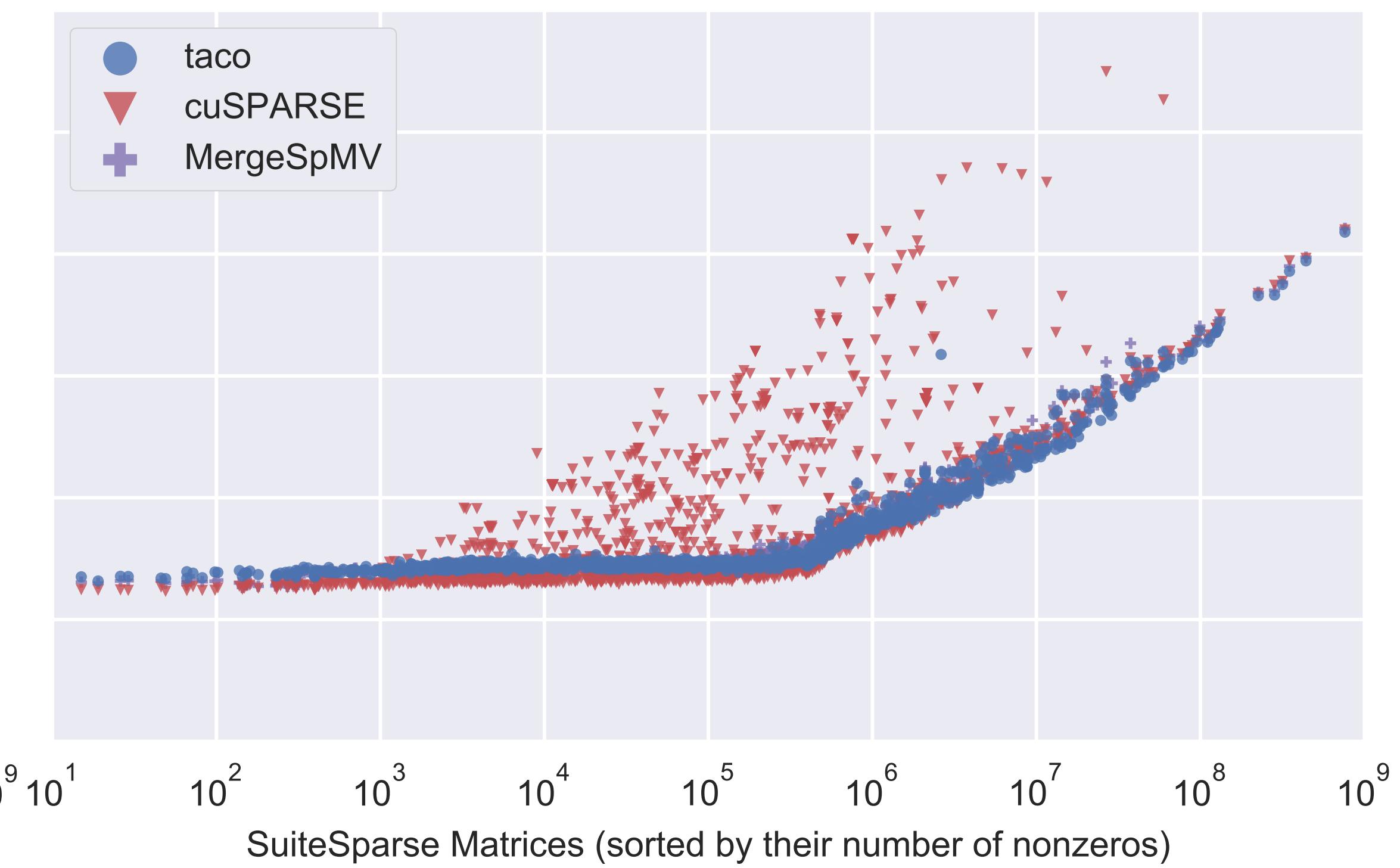
Machines matter for performance

$$a_i = \sum_j B_{ij} c_j \quad (\text{SpMV})$$

CPU



GPU

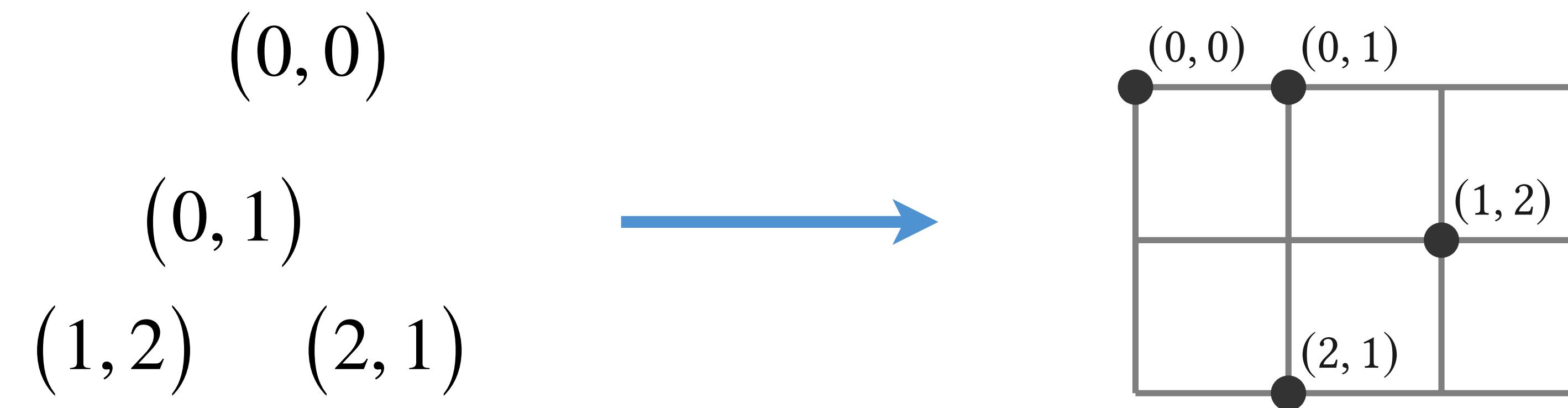


Sparse data structures in graphs, tensors, and relations encode coordinates in a sparse iteration space

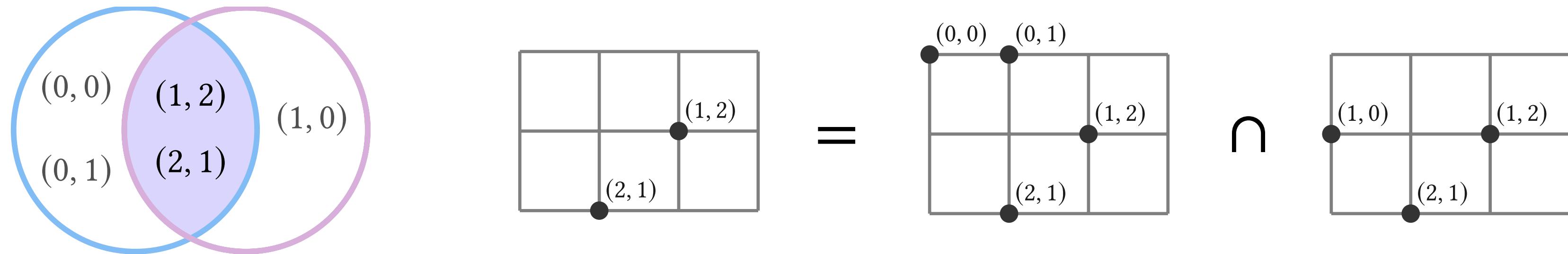
Tensor (nonzeros)		Relation (rows)	Graph (edges)
	(0,1)	(Harry,CS)	(v ₁ ,v ₅)
(2,3)	(0,5)	(Sally,EE)	(v ₄ ,v ₃)
(5,5)	(7,5)	(George,CS)	(v ₅ ,v ₃)
		(Rita,CS)	(v ₃ ,v ₅)
		(Mary,ME)	(v ₃ ,v ₁)

Values may be attached to these coordinates: e.g., nonzero values, edge attributes

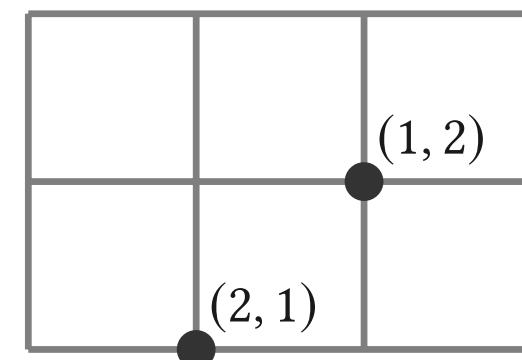
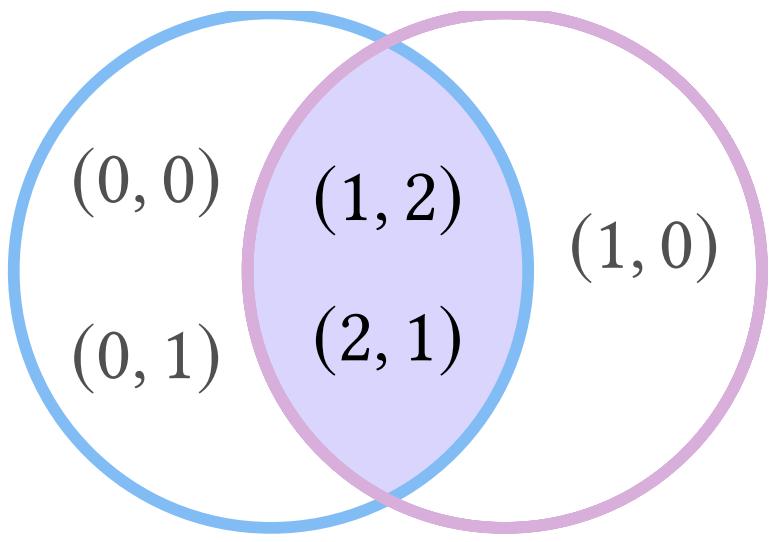
Iteration spaces from coordinate relations



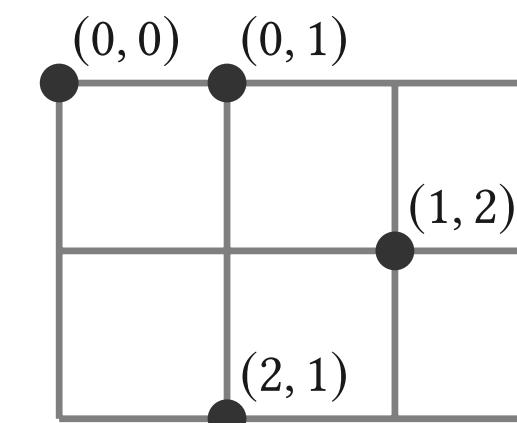
Iteration spaces from set operations



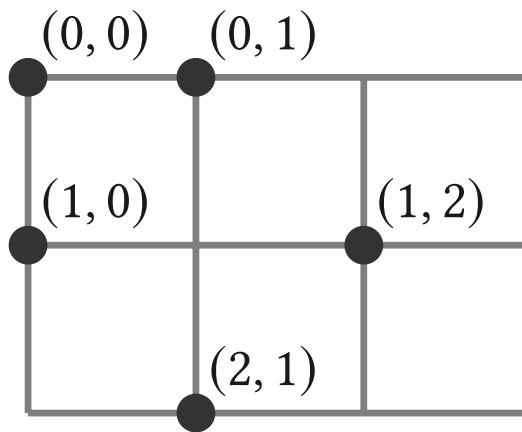
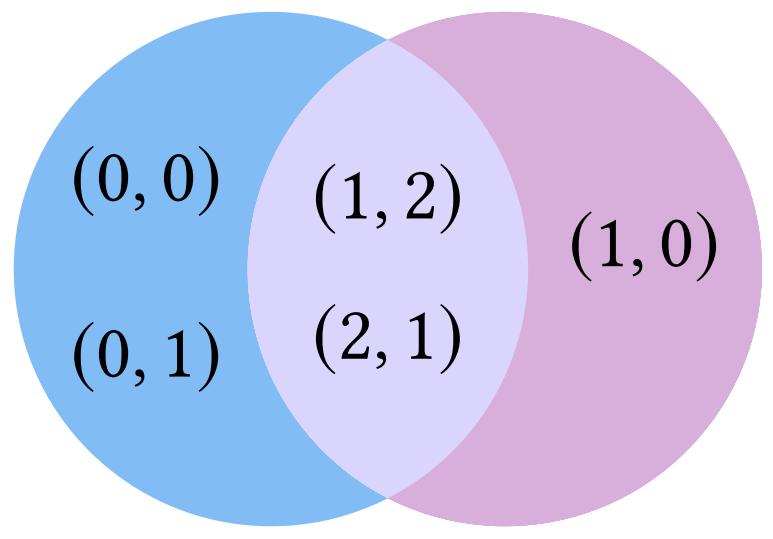
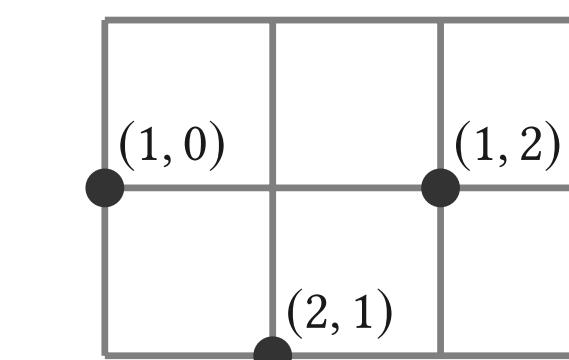
Iteration spaces from set operations



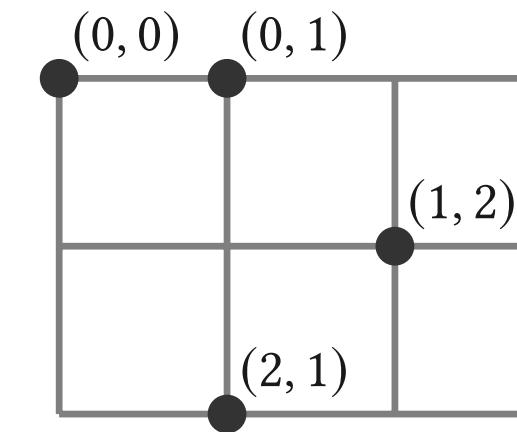
=



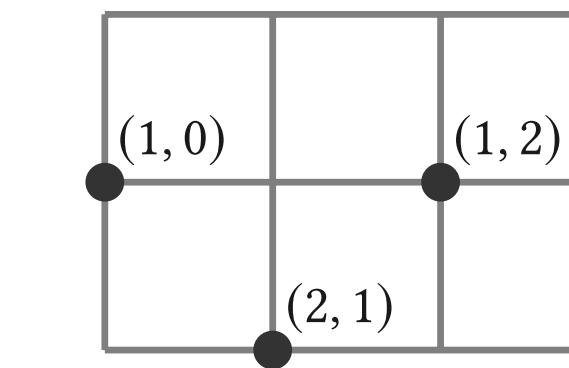
\cap



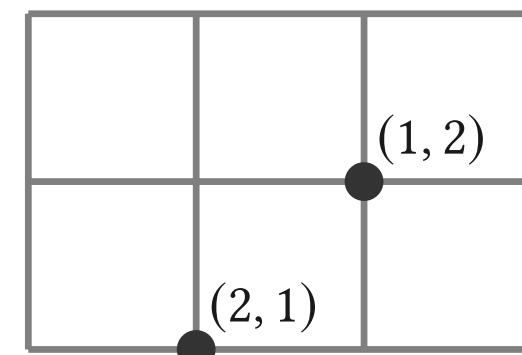
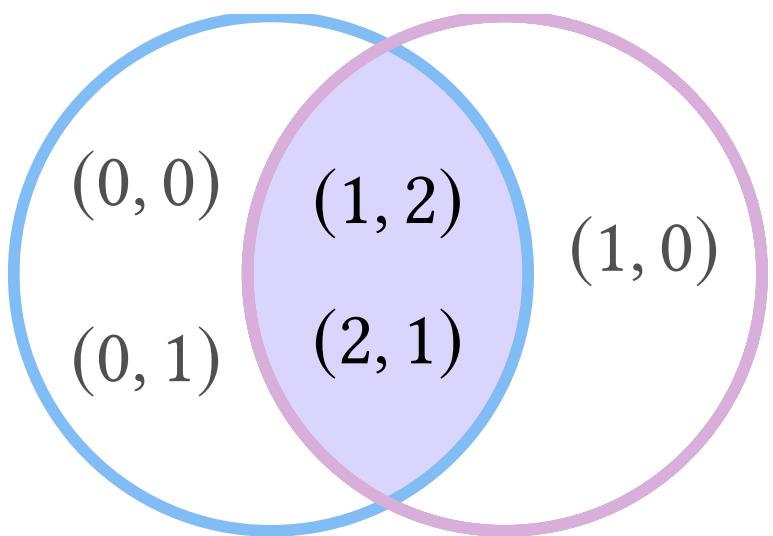
=



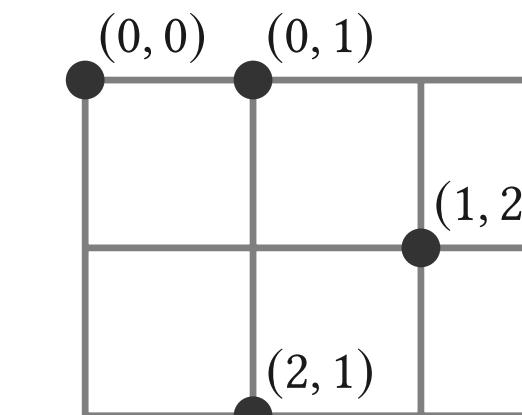
\cup



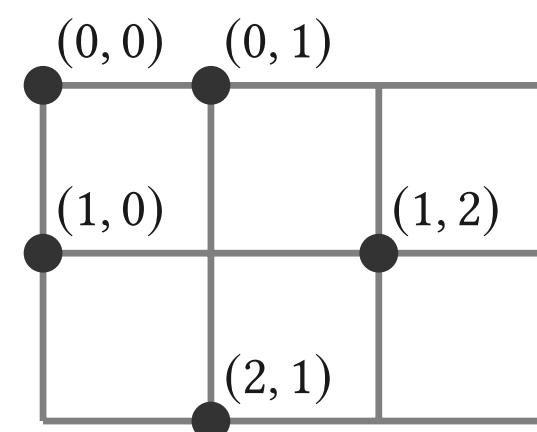
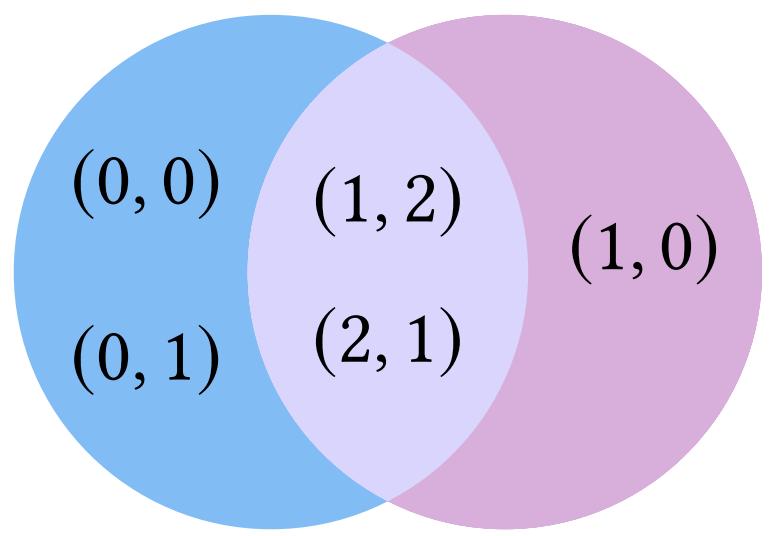
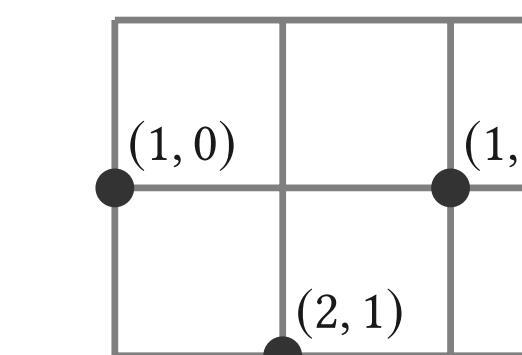
Iteration spaces from set operations



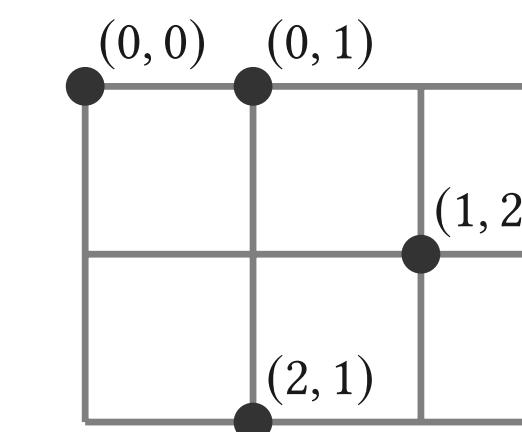
=



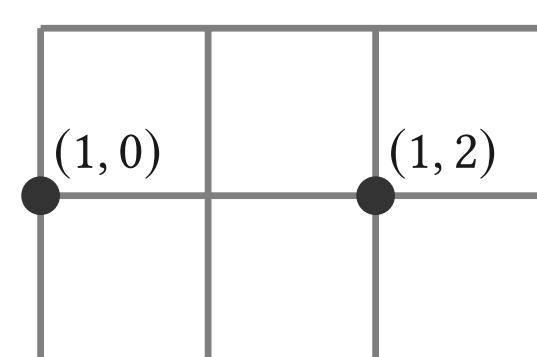
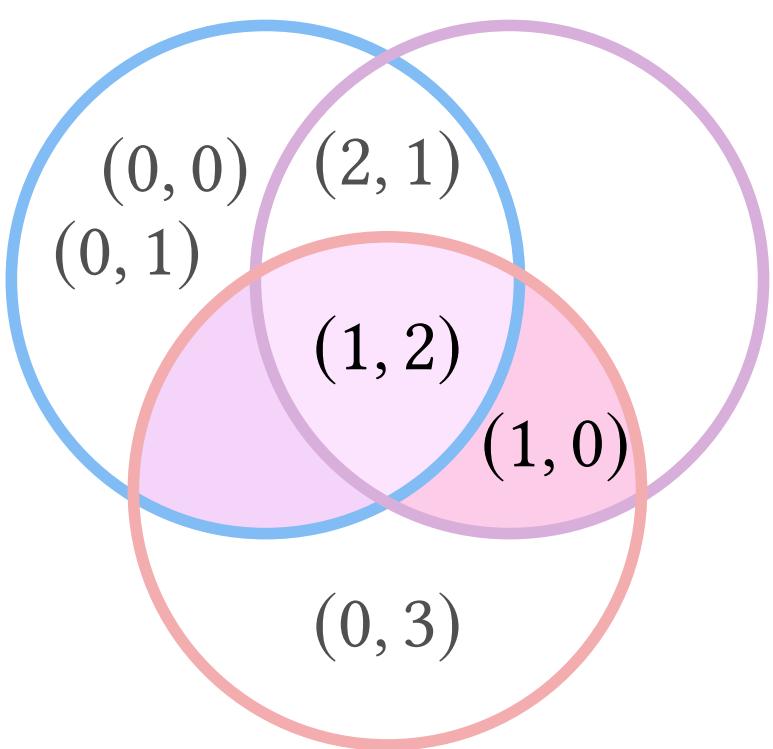
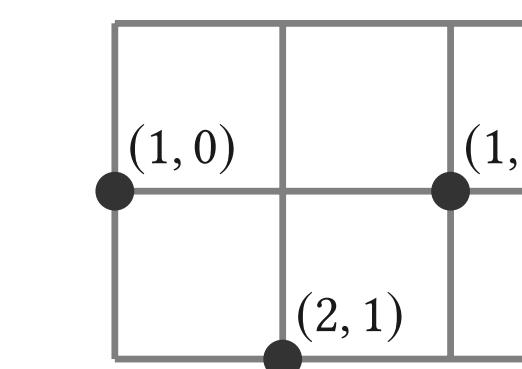
\cap



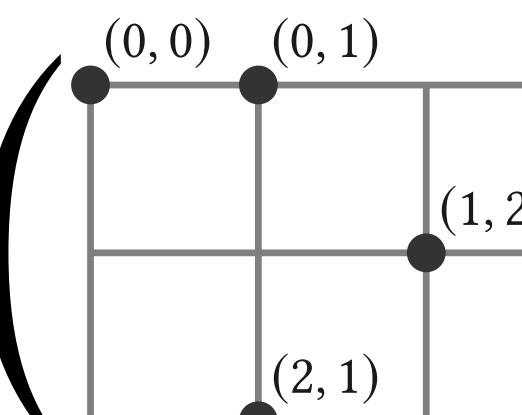
=



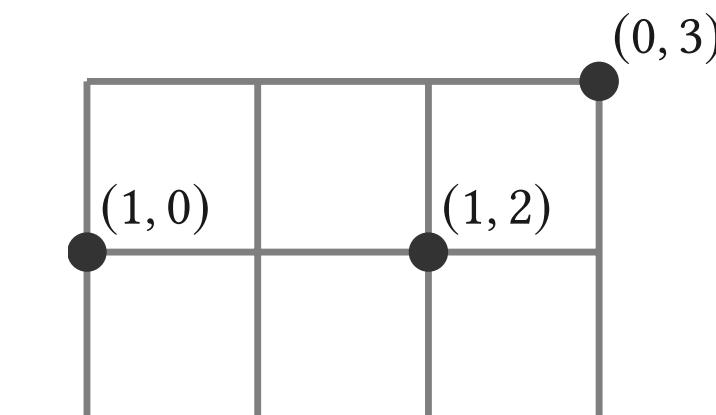
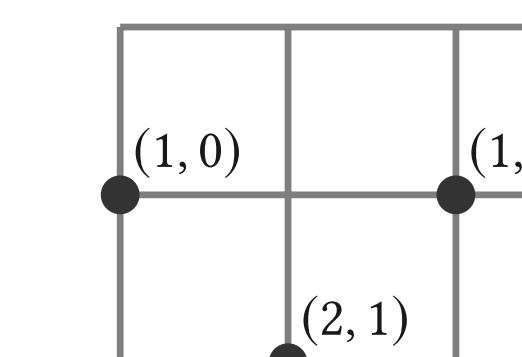
\cup



=

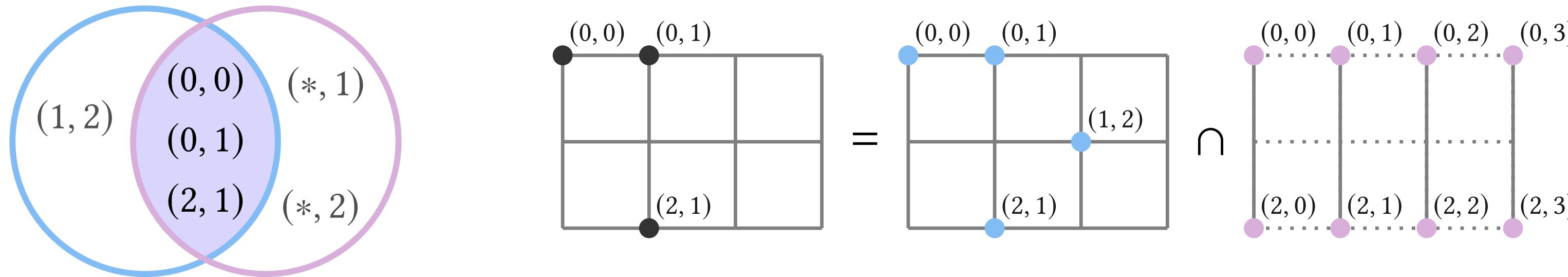


\cup

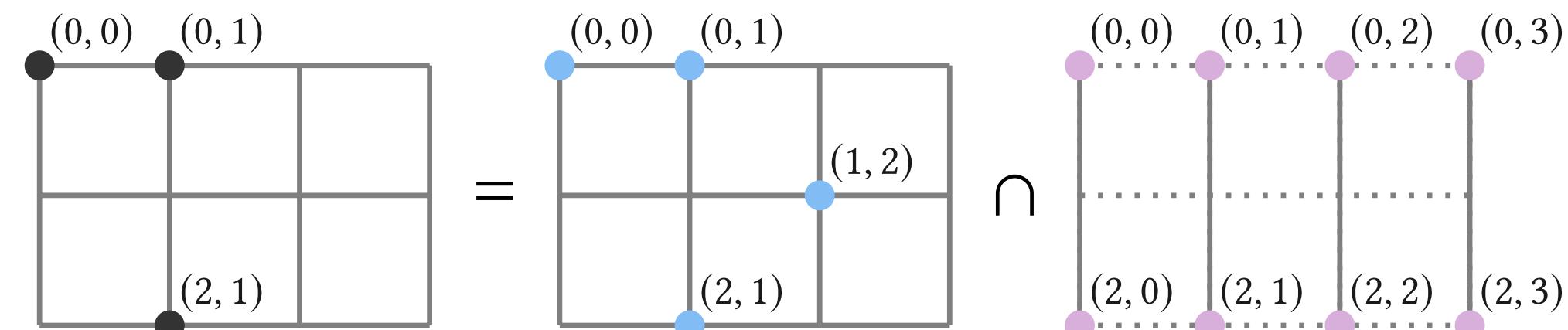
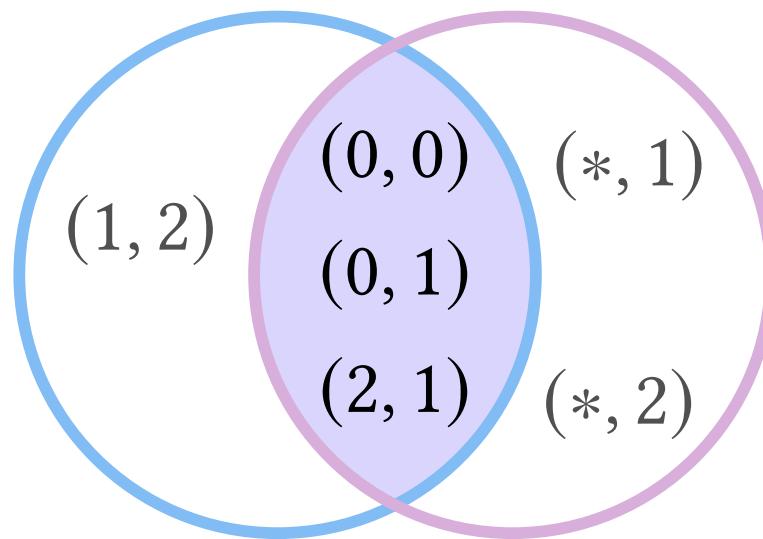


\cap

Iteration spaces from broadcast operations

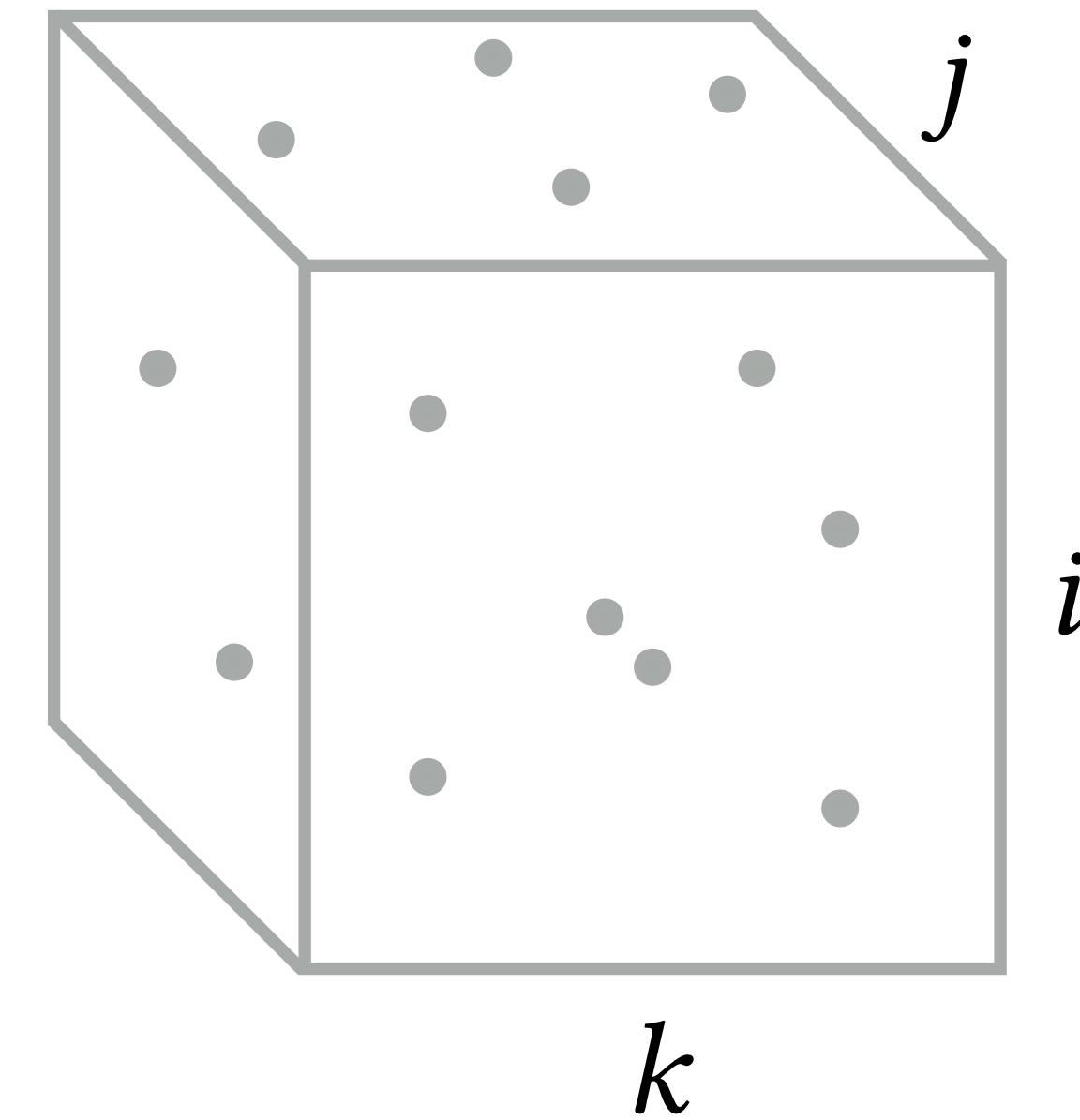


Iteration spaces from broadcast operations

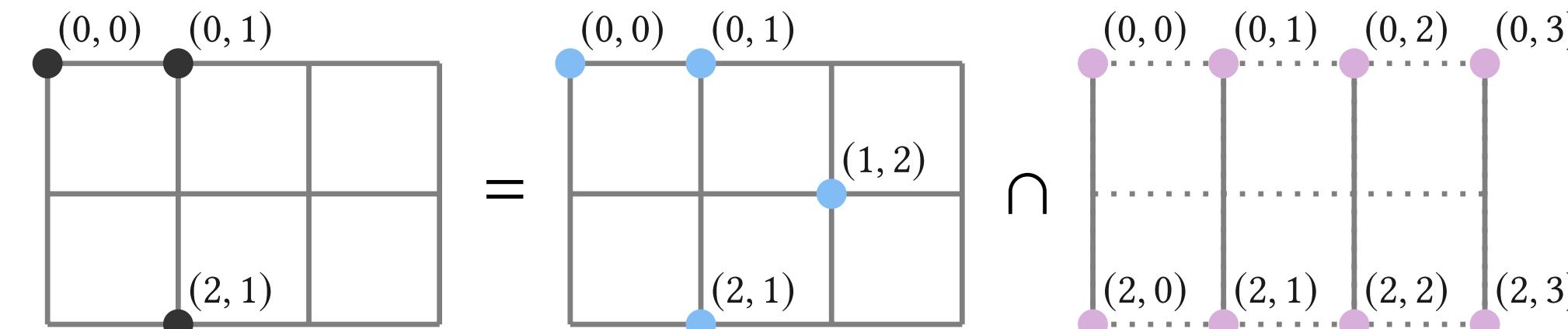
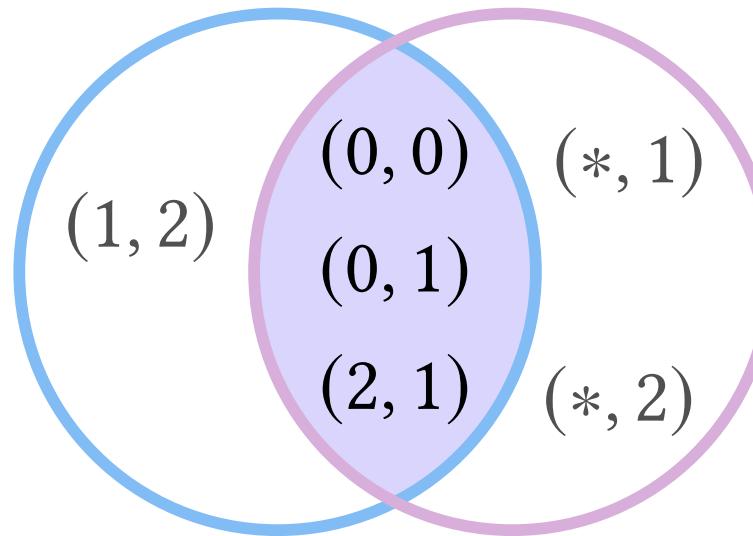


$$A_{ij} = \sum_k B_{ik} C_{kj}$$

$$\textcolor{blue}{B}_{ik} \cap \textcolor{violet}{C}_{kj}$$

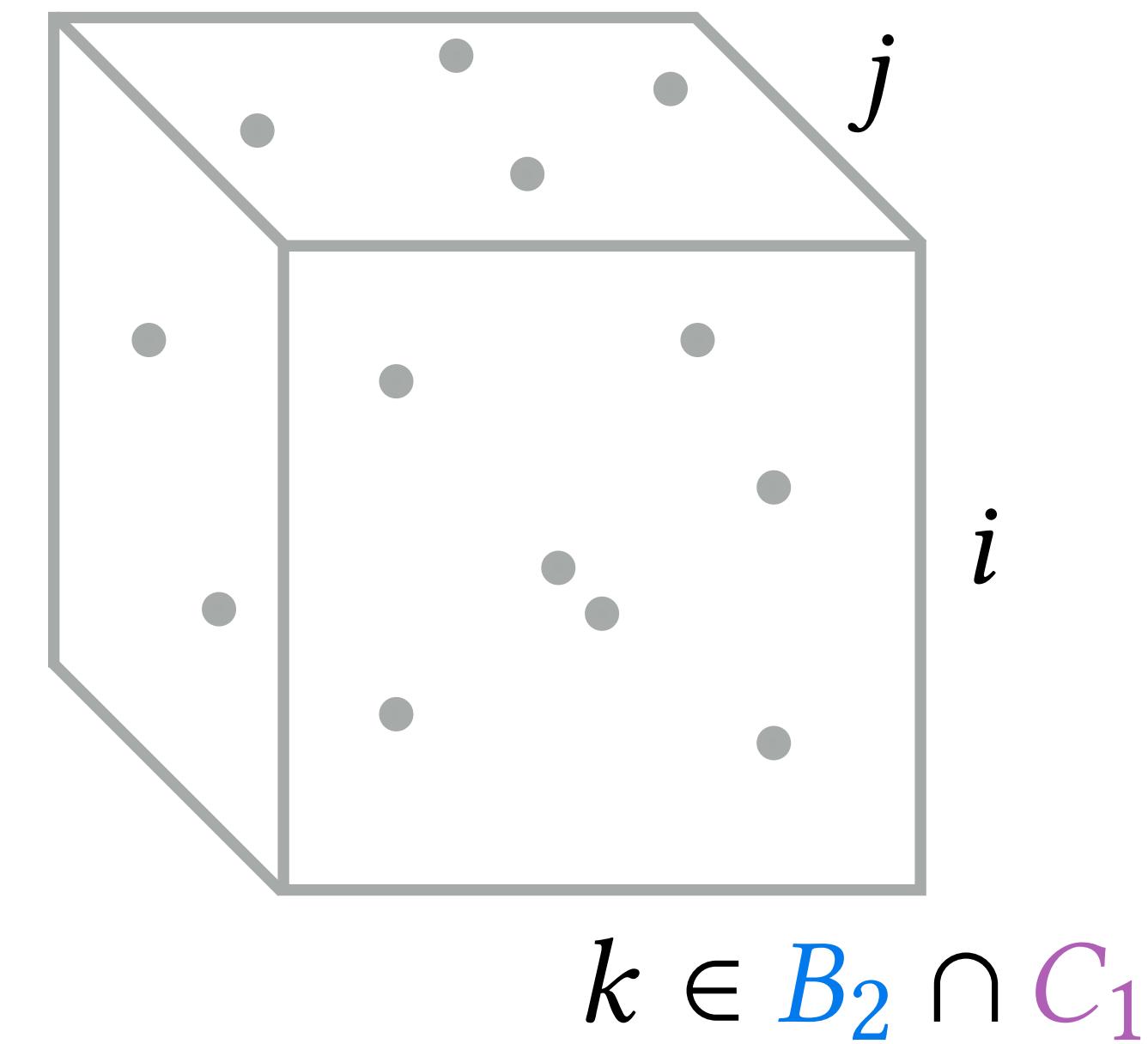


Iteration spaces from broadcast operations

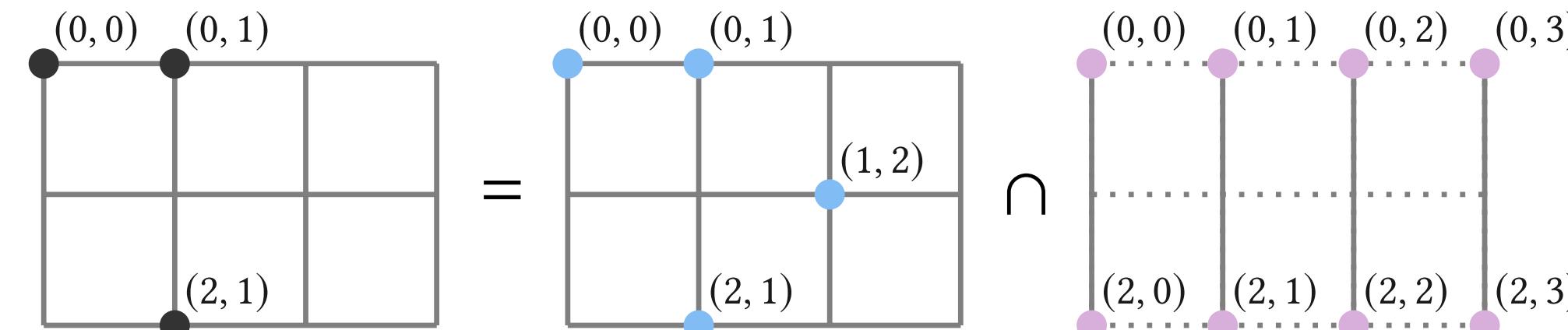
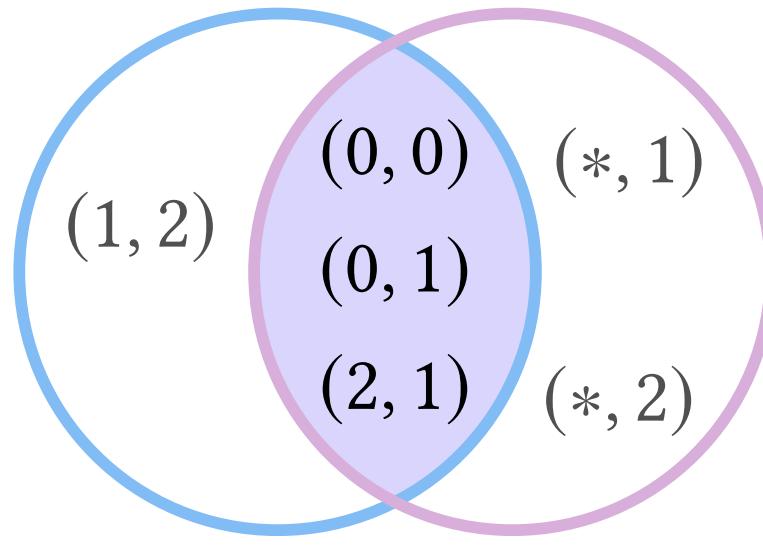


$$A_{ij} = \sum_k B_{ik} C_{kj}$$

$$\textcolor{blue}{B}_{ik} \cap \textcolor{violet}{C}_{kj}$$

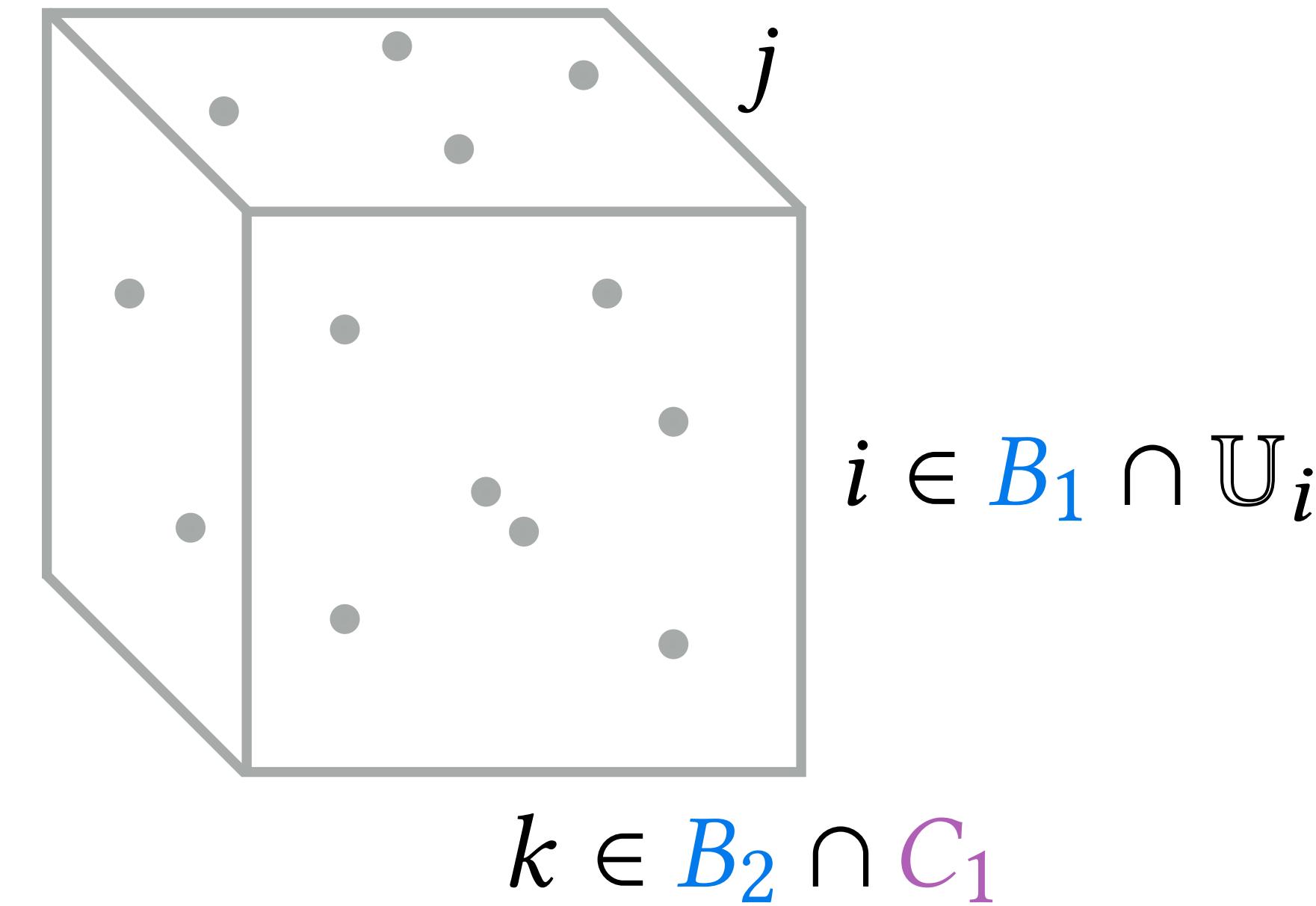


Iteration spaces from broadcast operations

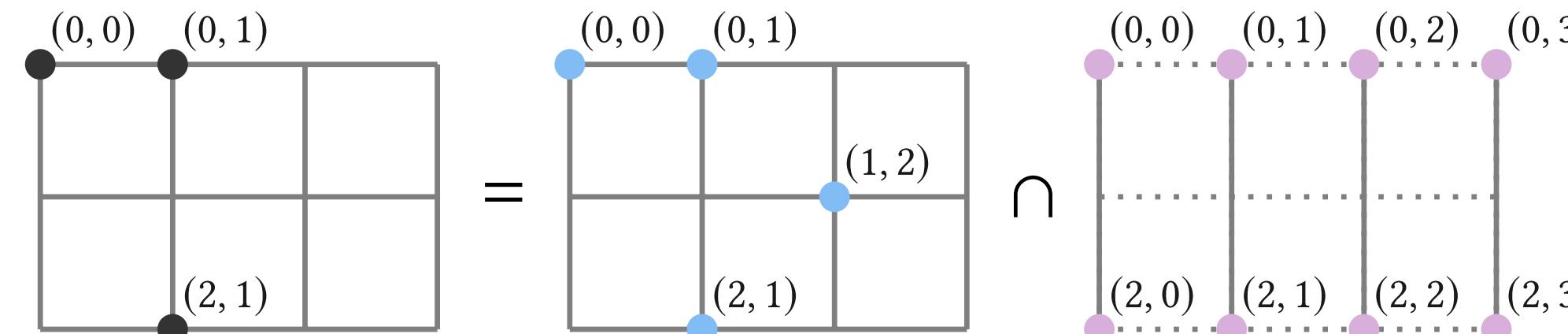
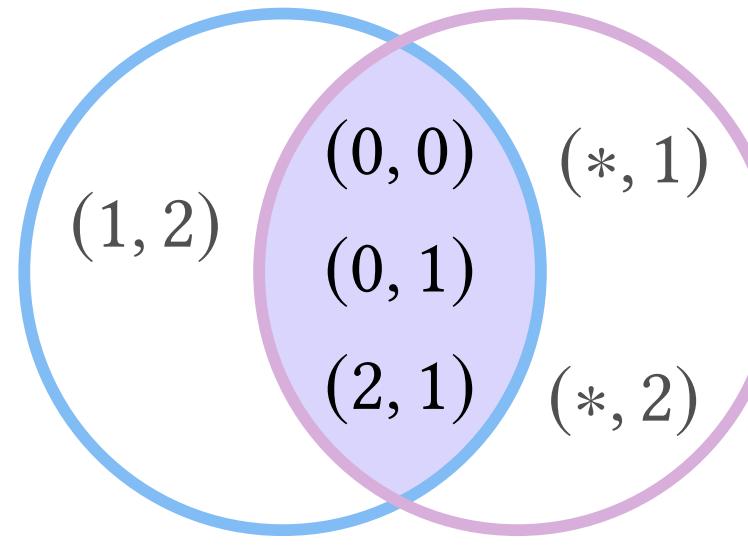


$$A_{ij} = \sum_k B_{ik} C_{kj}$$

$$\textcolor{blue}{B}_{ik} \cap \textcolor{violet}{C}_{kj}$$

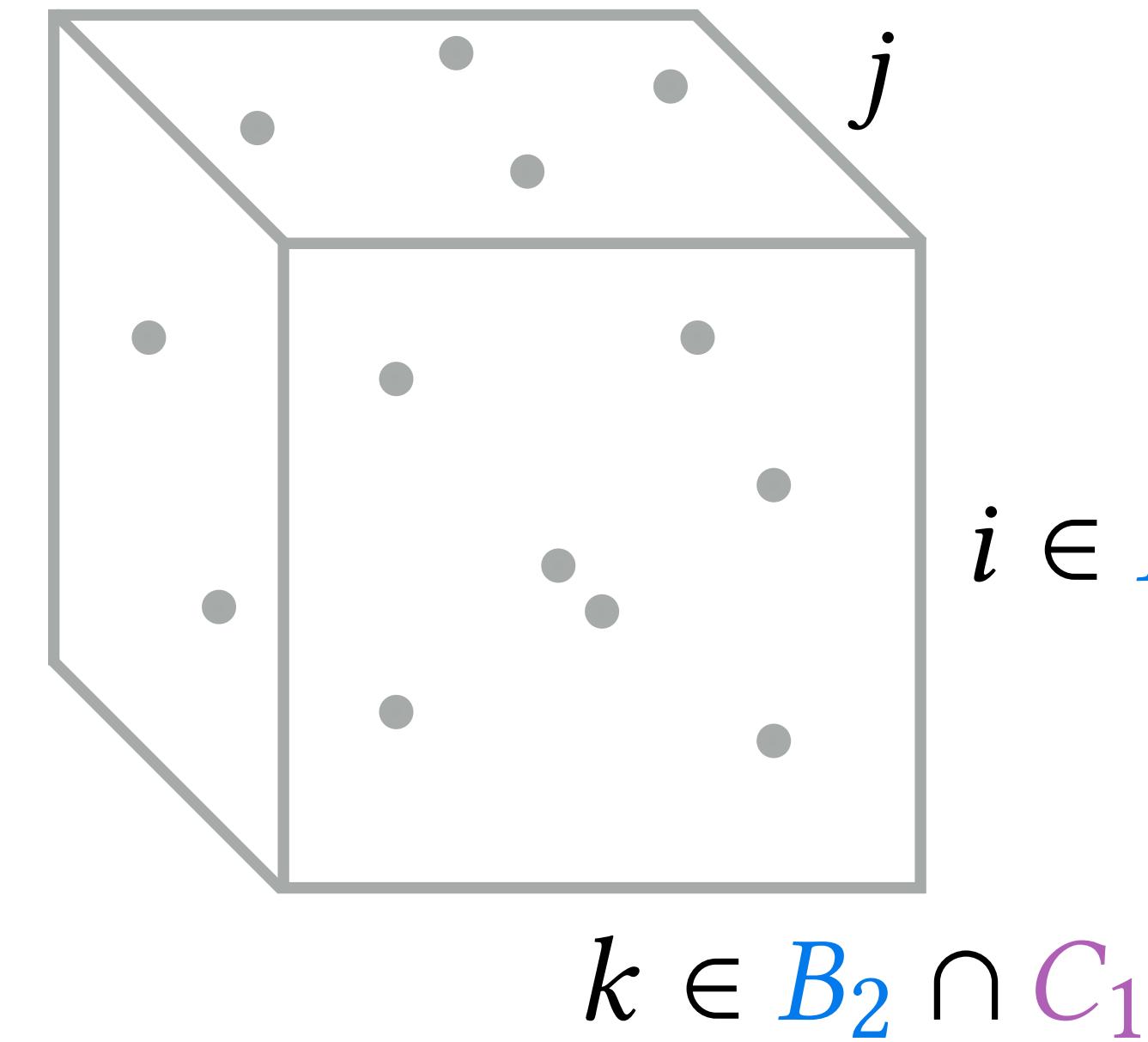


Iteration spaces from broadcast operations



$$A_{ij} = \sum_k B_{ik} C_{kj}$$

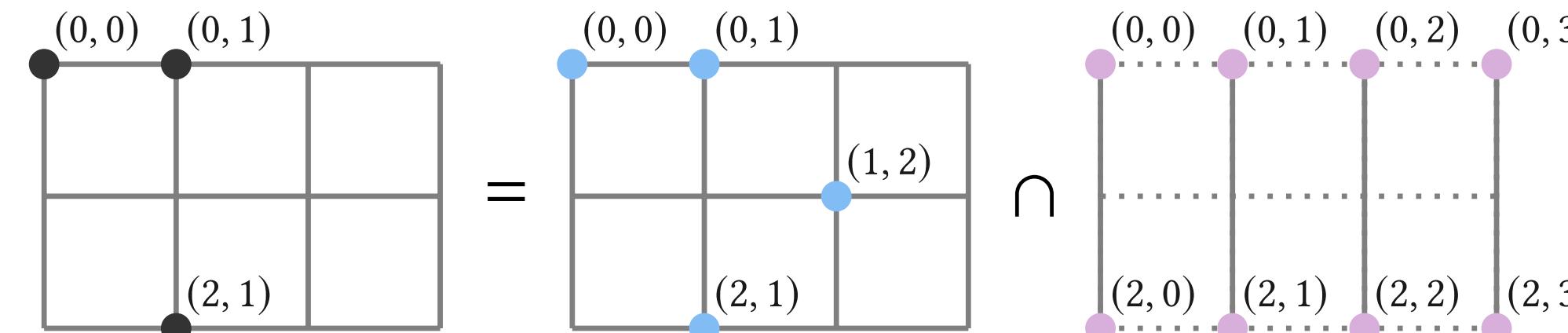
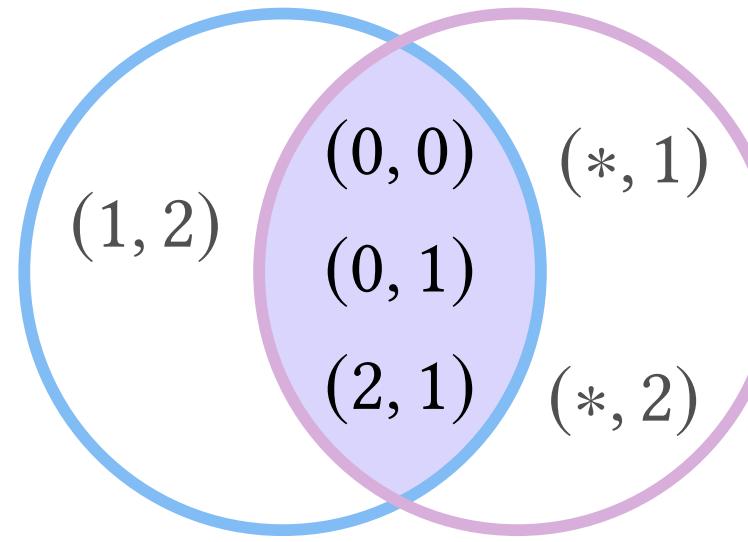
$$B_{ik} \cap C_{kj}$$



$$i \in B_1 \cap \mathbb{U}_i = B_1$$

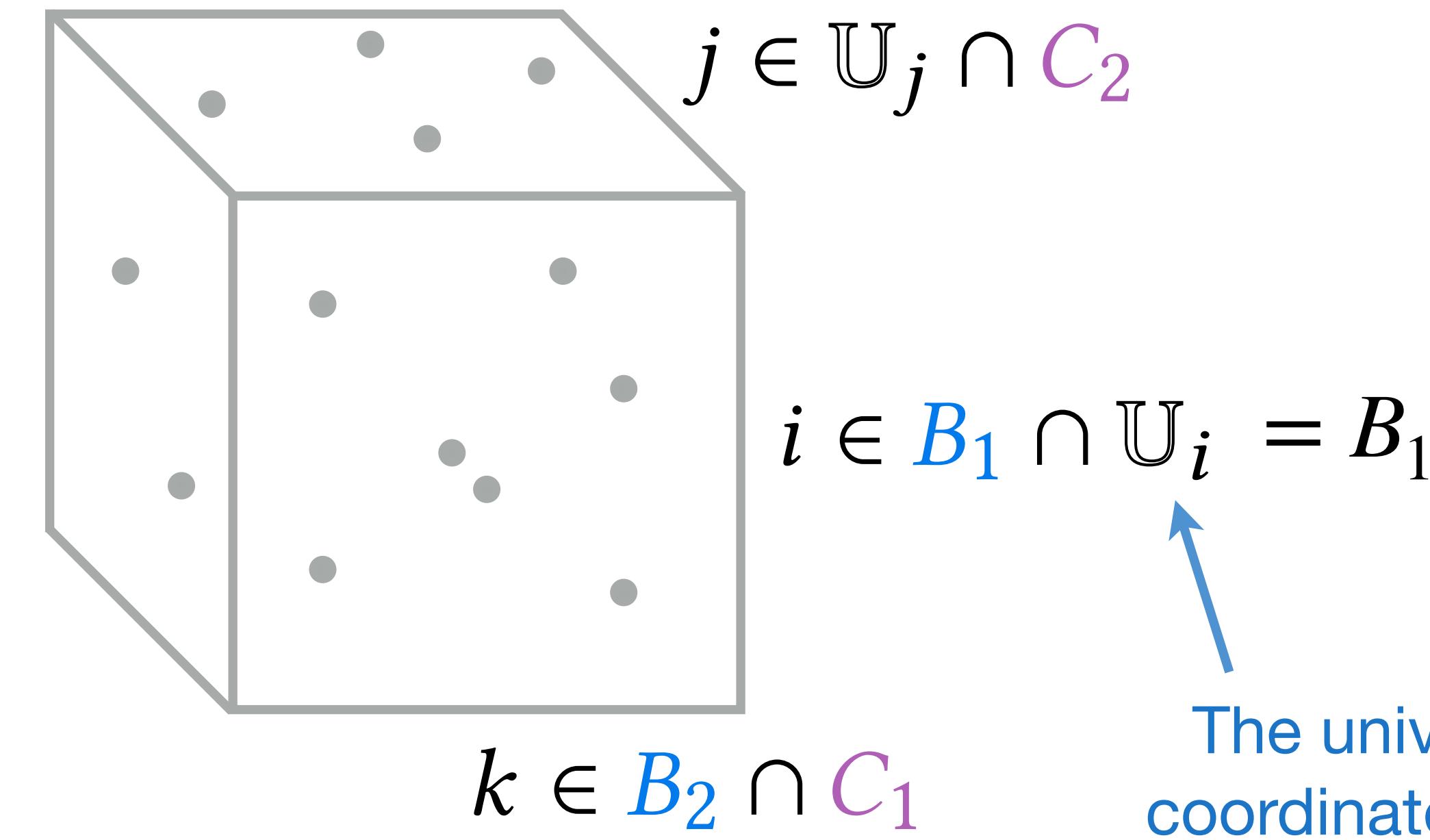
The universe of i consist of all coordinates it may take, of which any data structure stores a subset.

Iteration spaces from broadcast operations



$$A_{ij} = \sum_k B_{ik} C_{kj}$$

$$B_{ik} \cap C_{kj}$$



The universe of i consist of all coordinates it may take, of which any data structure stores a subset.

Coordinate relations → coordinate trees (abstractly)

Matrix

	j_1	j_2	j_3
i_1	a	b	
i_2		e	
i_3	g	h	i

Coordinate relations → coordinate trees (abstractly)

Matrix

	j_1	j_2	j_3
i_1	a	b	
i_2		e	
i_3	g	h	i



Coordinate Relation

$$(i_1, j_1) \rightarrow a \quad (i_1, j_2) \rightarrow b$$

$$(i_3, j_3) \rightarrow i \quad (i_2, j_2) \rightarrow e$$

$$(i_3, j_1) \rightarrow g$$

$$(i_3, j_2) \rightarrow h$$

Coordinate relations → coordinate trees (abstractly)

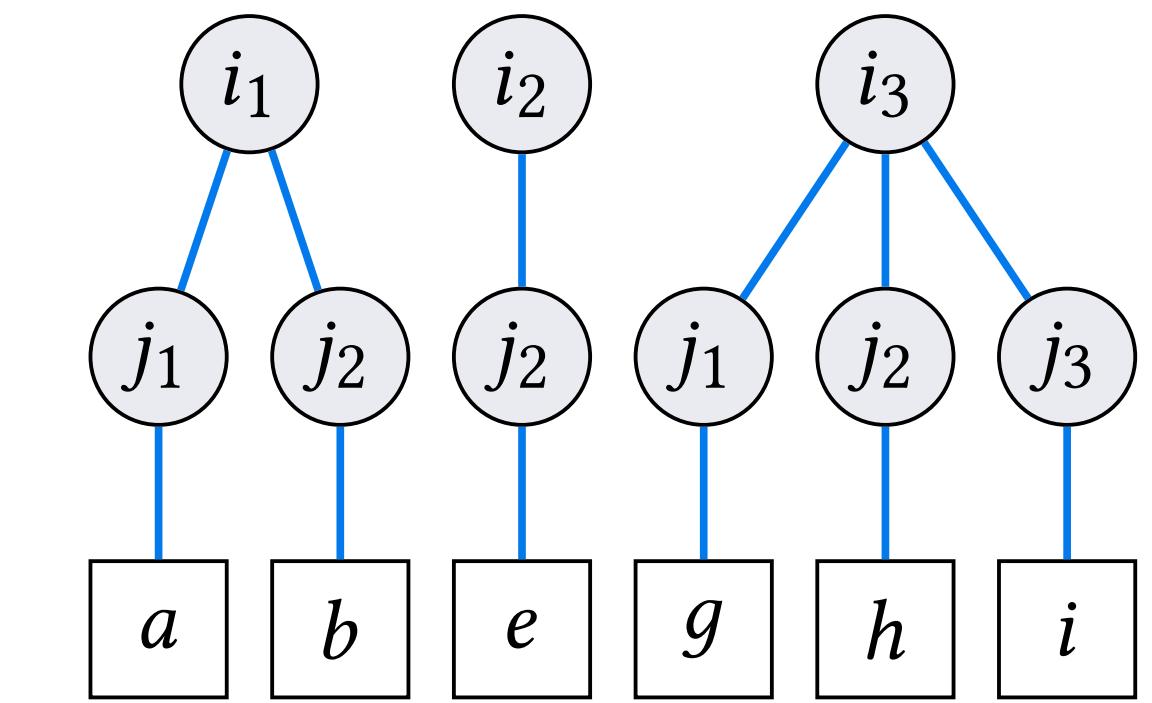
Matrix

	j_1	j_2	j_3
i_1	a	b	
i_2		e	
i_3	g	h	i

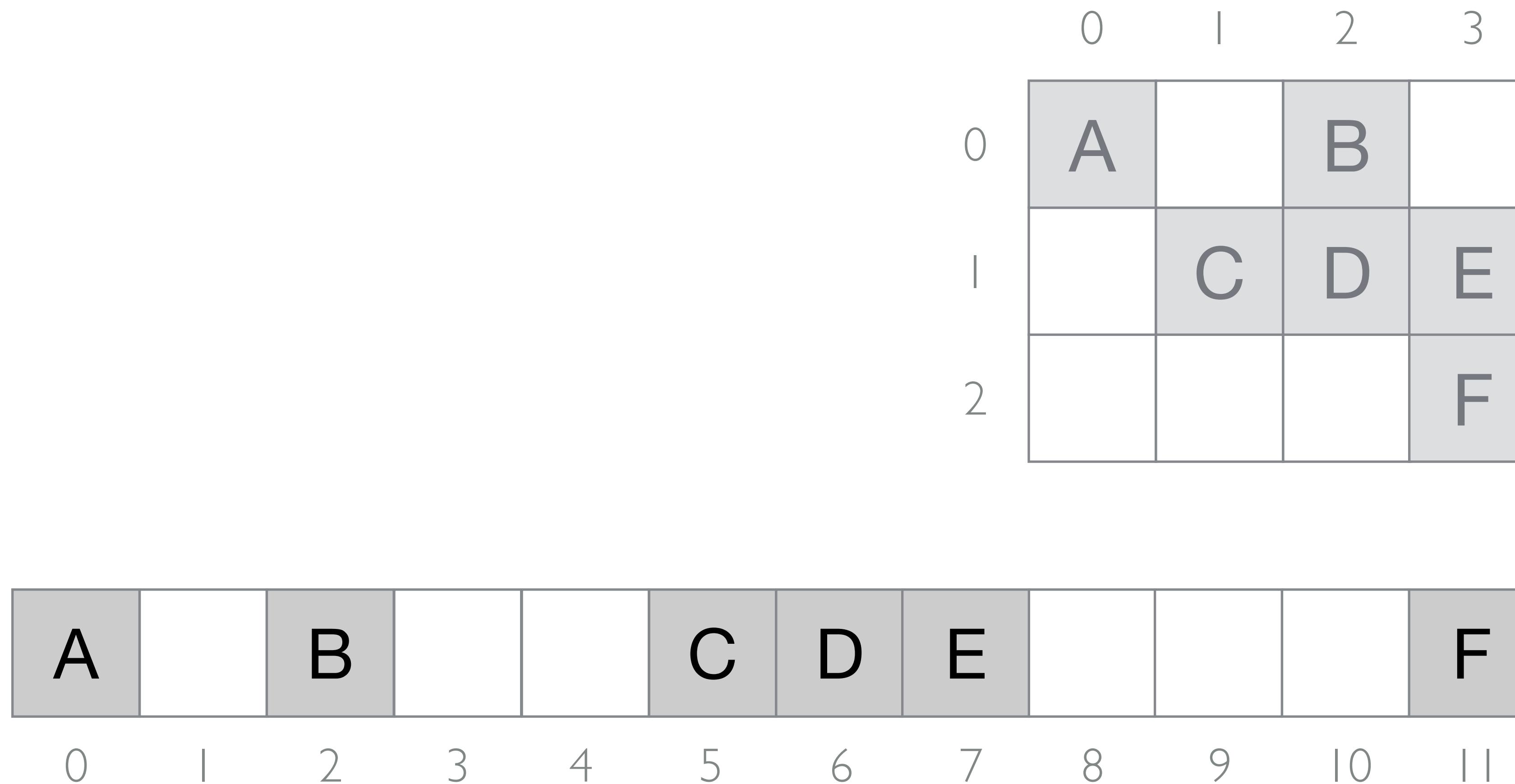
Coordinate Relation

$$\begin{array}{ll} (i_1, j_1) \rightarrow a & (i_1, j_2) \rightarrow b \\ (i_3, j_3) \rightarrow i & (i_2, j_2) \rightarrow e \\ (i_3, j_1) \rightarrow g & \\ (i_3, j_2) \rightarrow h & \end{array}$$

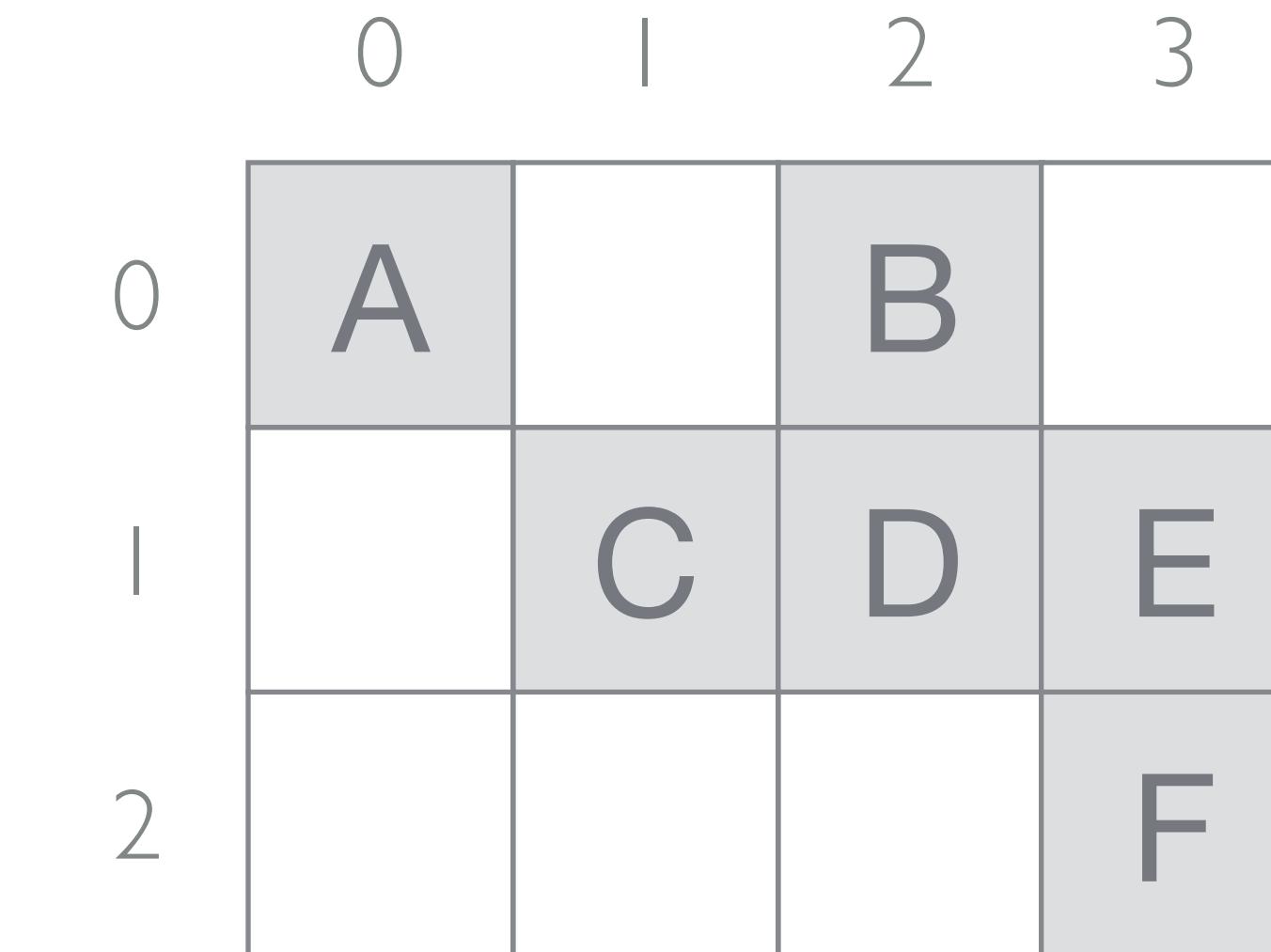
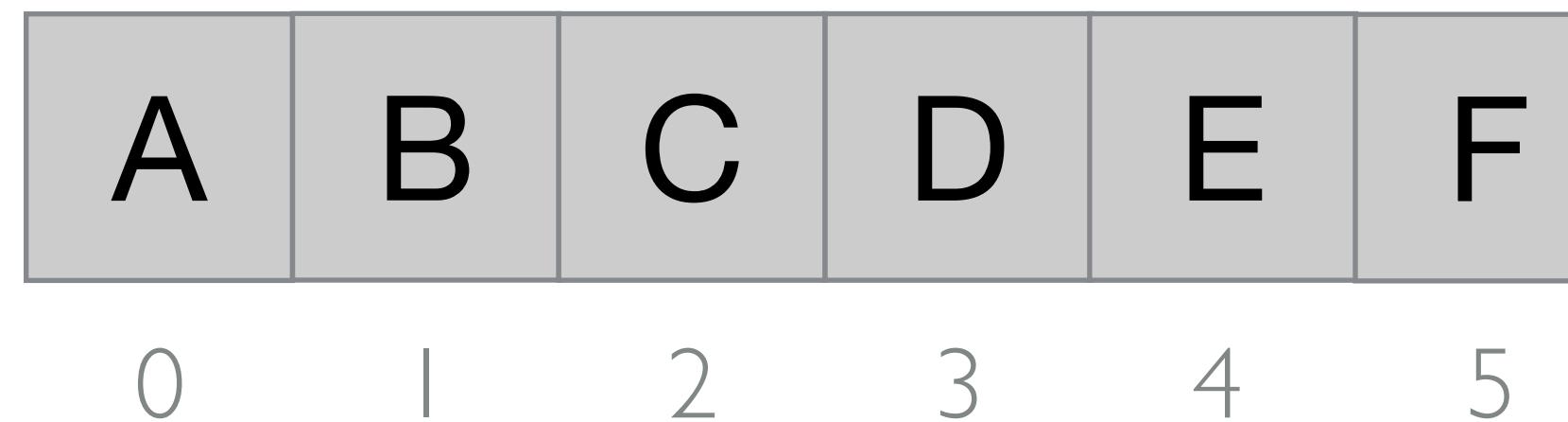
Coordinate Tree



Coordinate relations → coordinate trees (concretely)



Coordinate relations → coordinate trees (concretely)



Coordinate relations → coordinate trees (concretely)

$\text{row}(3) = ???$

$\text{col}(3) = ???$

	0		2	3
0	A		B	
		C	D	E
2				F

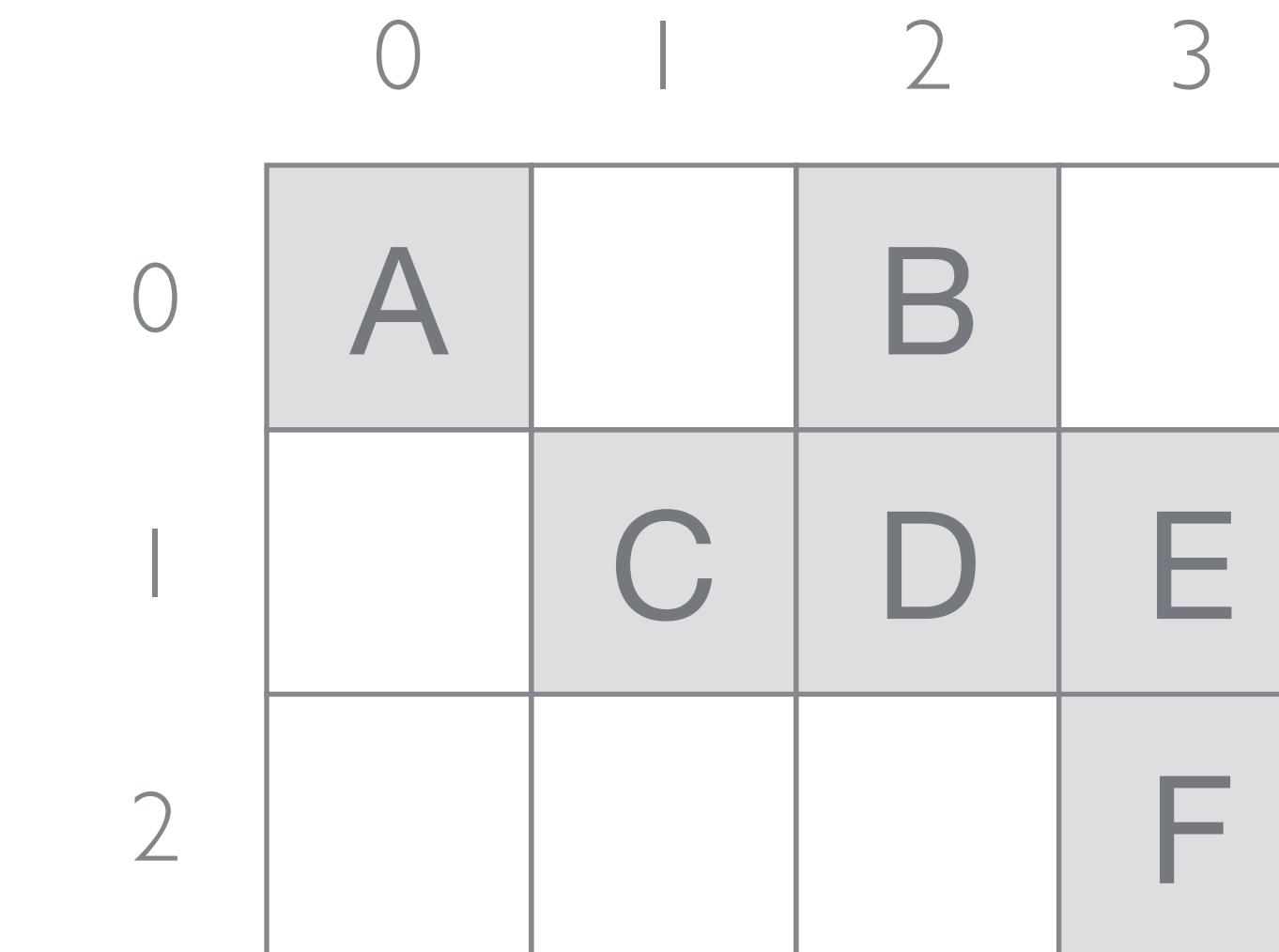


Coordinate relations → coordinate trees (concretely)

Coordinate

	rows					
	0	0	1	1	1	2

cols	0	2	1	2	3	3
------	---	---	---	---	---	---



Coordinate relations → coordinate trees (concretely)

Coordinate

	Coordinate					
rows	0	0	1	1	1	2
cols	0	2	1	2	3	3

0	I	2	3
A		B	
	C	D	E
2			F



Coordinate relations → coordinate trees (concretely)

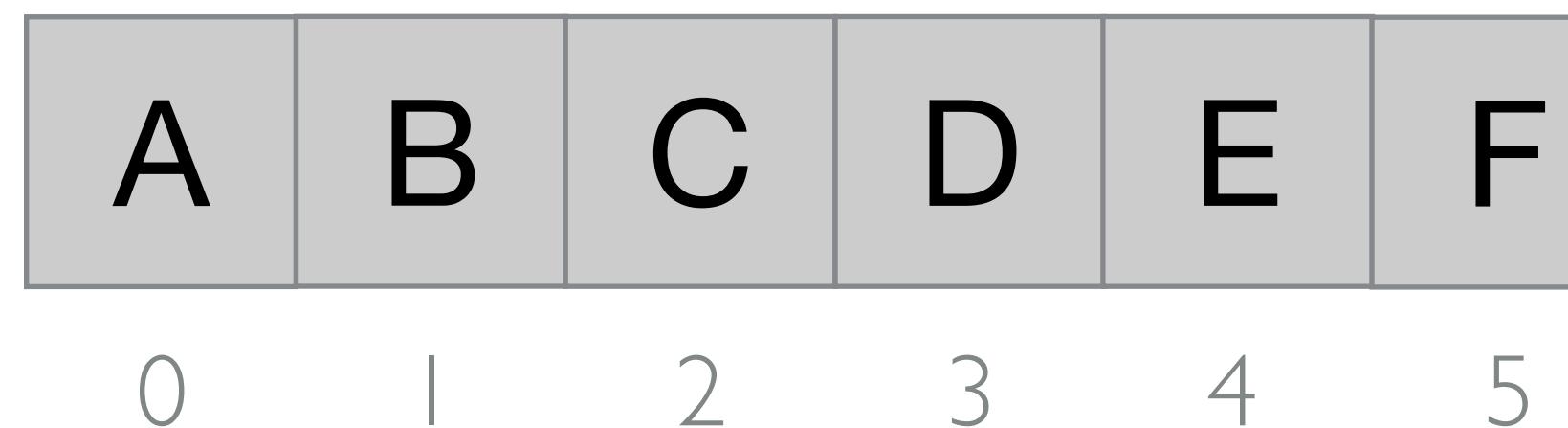
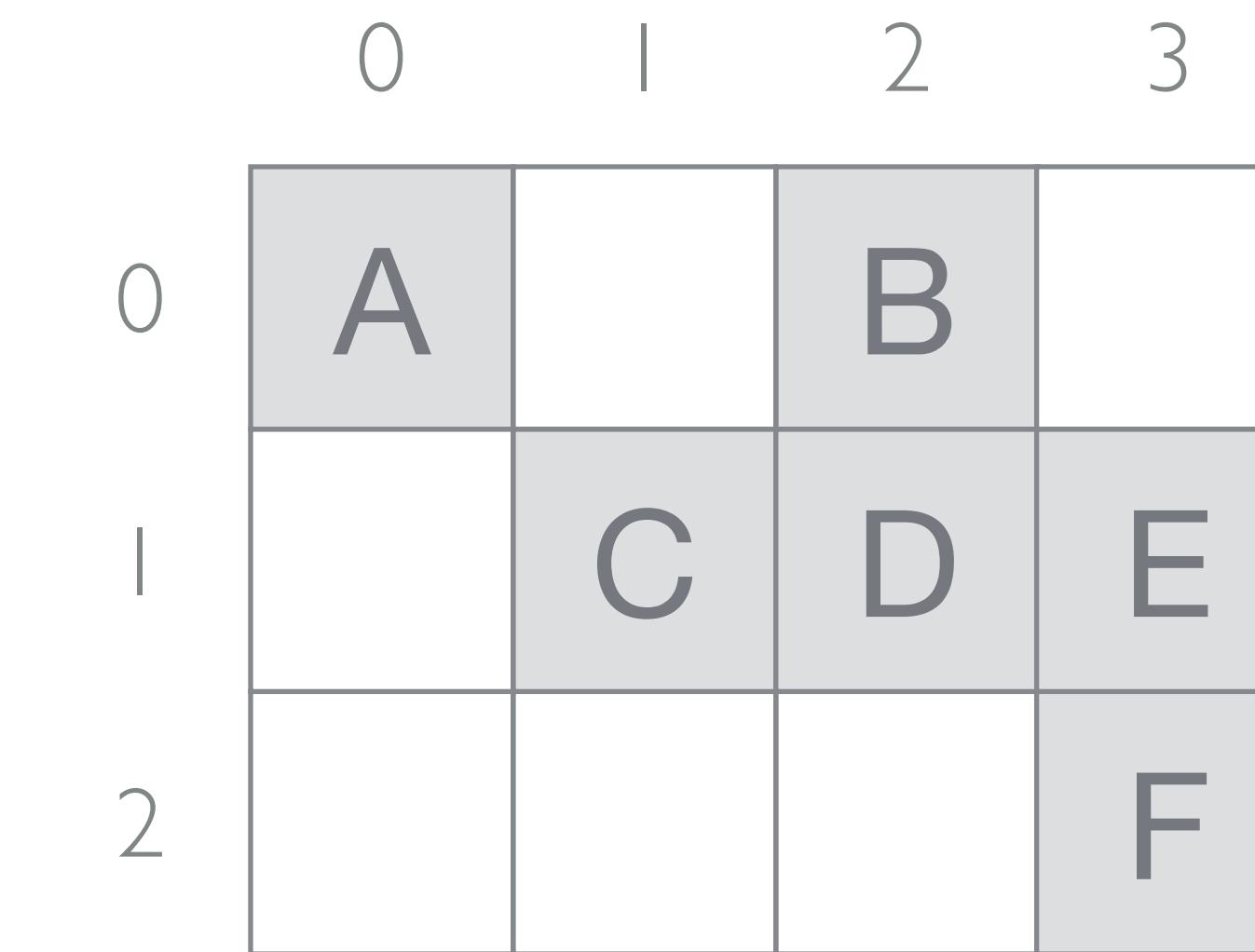
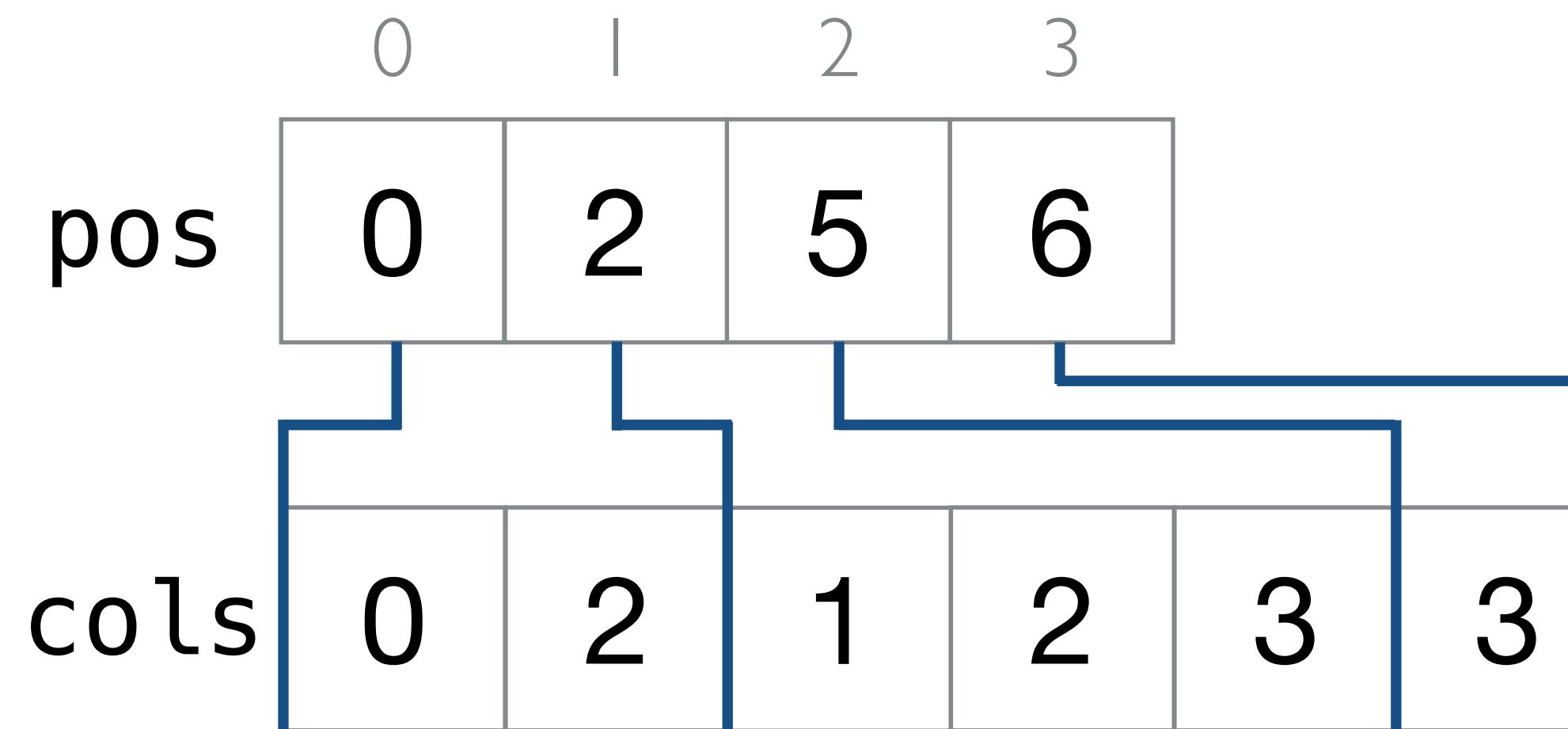
		Coordinate			Duplicates
		rows	0	1	2
cols	0	0	1	1	2

0	1	2	3
A		B	
	C	D	E
			F

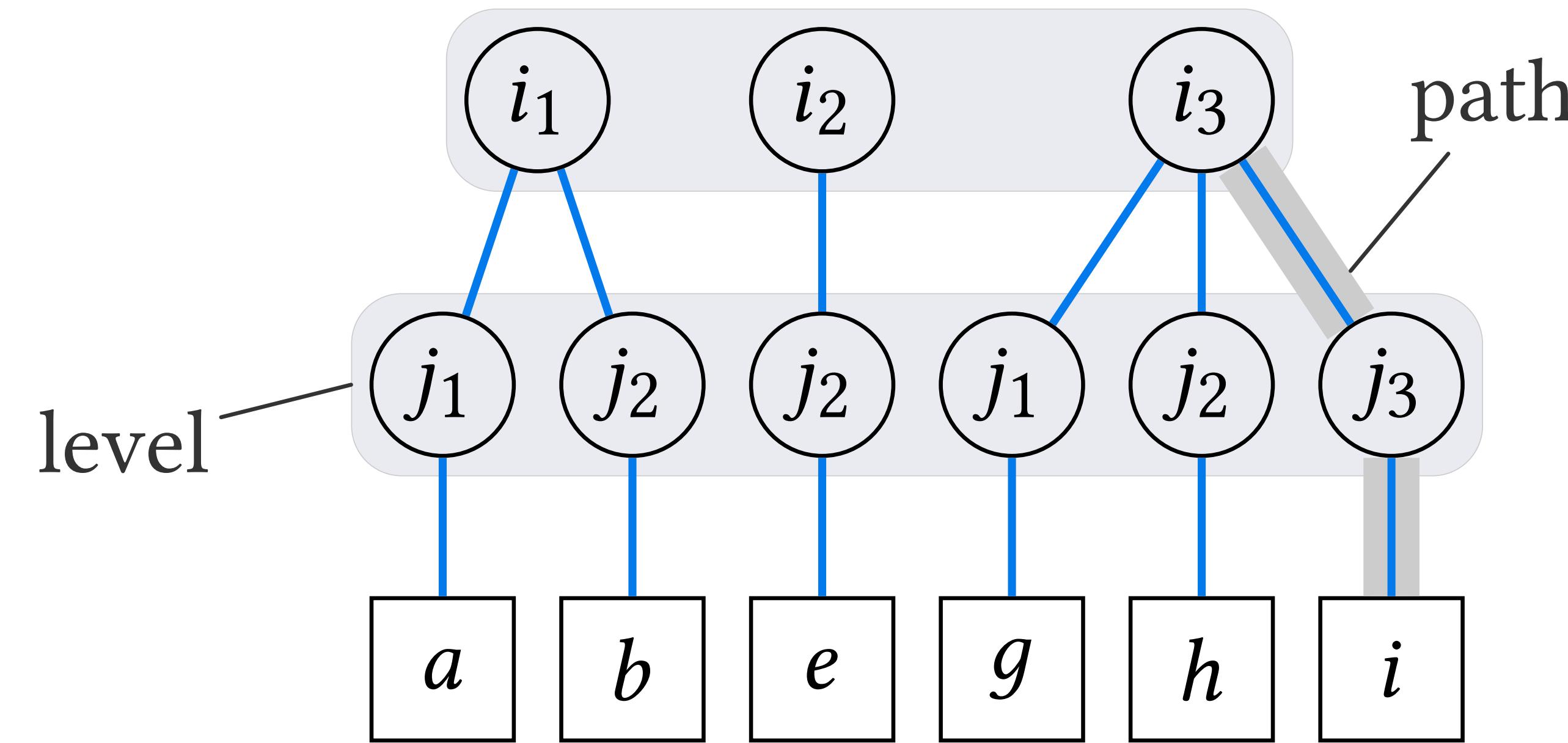


Coordinate relations → coordinate trees (concretely)

Compressed Sparse Rows (CSR)

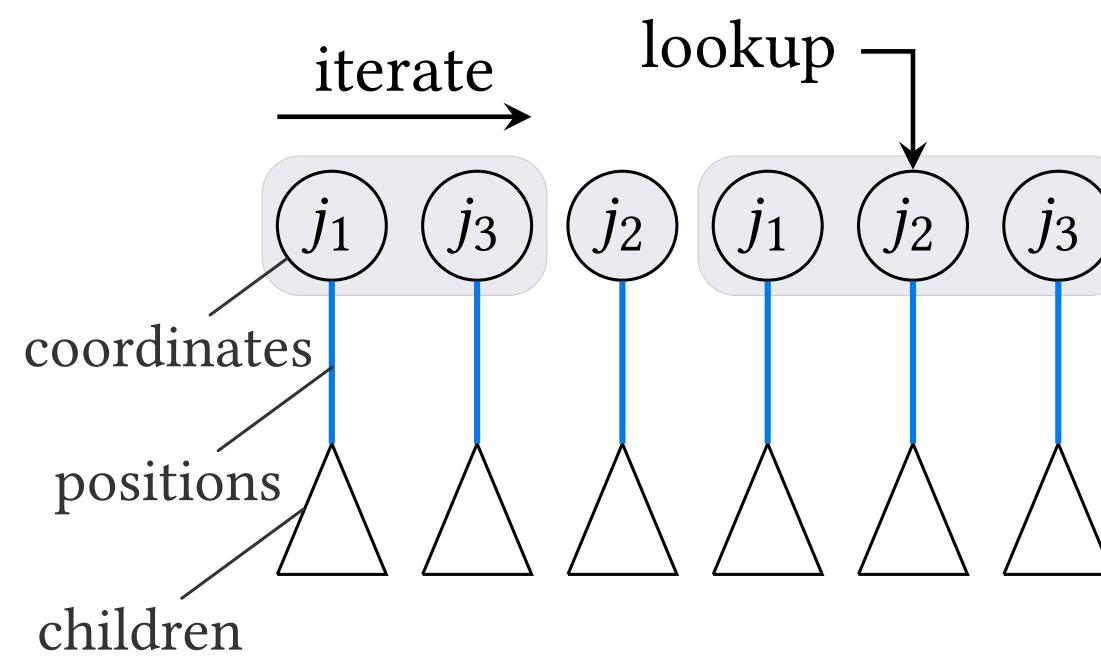


Level-based representation: compiler abstraction

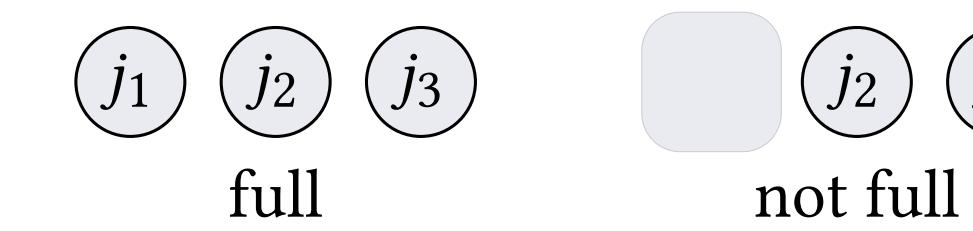


Level abstraction: capabilities and properties

Capabilities

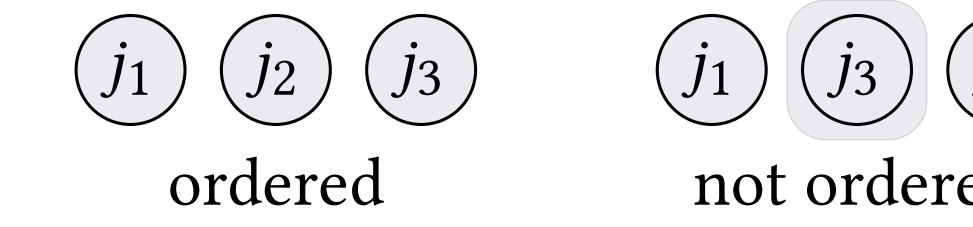


Properties



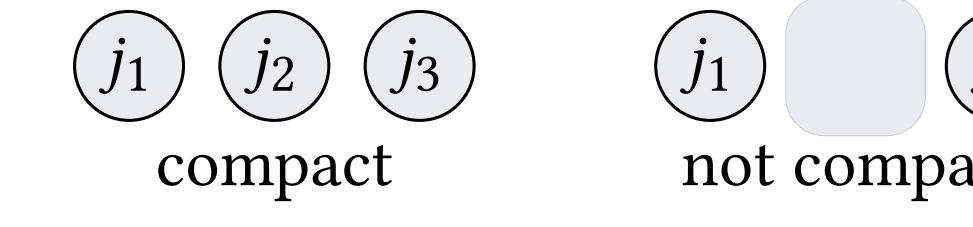
full

not full



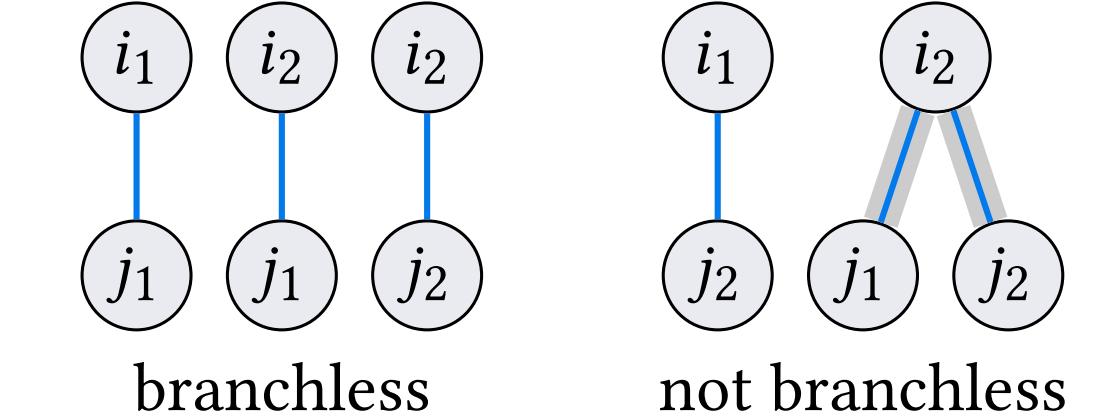
ordered

not ordered

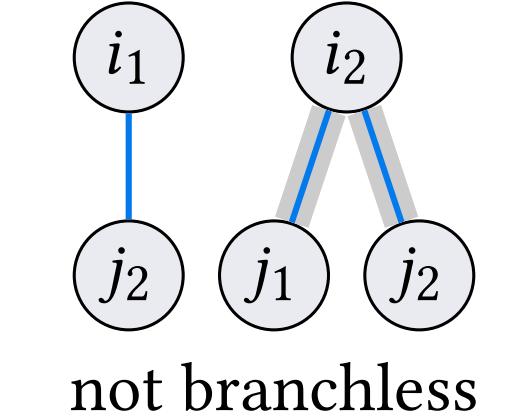


compact

not compact



branchless



not branchless

unique

not unique

The code generator sees only the level abstraction and not specific level types

Level types: dense and compressed

Dense locate capability:

```
locate(pk-1, i1, ..., ik):  
    return <pk-1 * Nk + ik, true>
```

Level types: dense and compressed

Dense locate capability:

```
locate(pk-1, i1, ..., ik):  
    return <pk-1 * Nk + ik, true>
```

Compressed iterate capability

```
pos_bounds(pk-1):  
    return <pos[pk-1], pos[pk-1 + 1]>
```

```
pos_access(pk, i1, ..., ik-1):  
    return <crd[pk], true>
```

Level types: dense and compressed

Dense locate capability:

```
locate(pk-1, i1, ..., ik):  
    return <pk-1 * Nk + ik, true>
```

Compressed iterate capability

```
pos_bounds(pk-1):  
    return <pos[pk-1], pos[pk-1 + 1]>
```

```
pos_access(pk, i1, ..., ik-1):  
    return <crd[pk], true>
```

$$y = Ax$$

```
for (int i = 0; i < m; i++) {  
    for (int pA = A_pos[i]; pA < A_pos[i+1]; pA++) {  
        int j = A_crd[pA];  
        y[i] += A[pA] * x[j];  
    }  
}
```

Level types: dense and compressed

Dense locate capability:

```
locate(pk-1, i1, ..., ik):  
    return <pk-1 * Nk + ik, true>
```

Compressed iterate capability

```
pos_bounds(pk-1):  
    return <pos[pk-1], pos[pk-1 + 1]>
```

```
pos_access(pk, i1, ..., ik-1):  
    return <crd[pk], true>
```

$$y = Ax$$

```
for (int i = 0; i < m; i++) {  
    for (int pA = A_pos[i]; pA < A_pos[i+1]; pA++) {  
        int j = A_crd[pA];  
        y[i] += A[pA] * x[j];  
    }  
}
```

Compressed iterate

Level types: dense and compressed

Dense locate capability:

```
locate(pk-1, i1, ..., ik):  
    return <pk-1 * Nk + ik, true>
```

Compressed iterate capability

$$y = Ax$$

```
for (int i = 0; i < m; i++) {  
    for (int pA = A_pos[i]; pA < A_pos[i+1]; pA++) {  
        int j = A_crd[pA];  
        y[i] += A[pA] * x[j];  
    }  
}
```

```
pos_bounds(pk-1):  
    return <pos[pk-1], pos[pk-1 + 1]>  
  
pos_access(pk, i1, ..., ik-1):  
    return <crd[pk], true>
```

Compressed iterate

Level types: dense and compressed

Dense locate capability:

```
locate(pk-1, i1, ..., ik):  
    return <pk-1 * Nk + ik, true>
```

Compressed iterate capability

$$y = Ax$$

```
for (int i = 0; i < m; i++) {  
    for (int pA = A_pos[i]; pA < A_pos[i+1]; pA++) {  
        int j = A_crd[pA];  
        y[i] += A[pA] * x[j];  
    }  
}
```

```
pos_bounds(pk-1):  
    return <pos[pk-1], pos[pk-1 + 1]>  
  
pos_access(pk, i1, ..., ik-1):  
    return <crd[pk], true>
```

Compressed iterate

Level types: dense and compressed

Dense locate capability:

```
locate(pk-1, i1, ..., ik):  
    return <pk-1 * Nk + ik, true>
```

Compressed iterate capability

```
pos_bounds(pk-1):  
    return <pos[pk-1], pos[pk-1 + 1]>
```

```
pos_access(pk, i1, ..., ik-1):  
    return <crd[pk], true>
```

$$y = Ax$$

```
for (int i = 0; i < m; i++) {  
    for (int pA = A_pos[i]; pA < A_pos[i+1]; pA++) {  
        int j = A_crd[pA];  
        y[i] += A[pA] * x[j];  
    }  
}
```

Dense locate

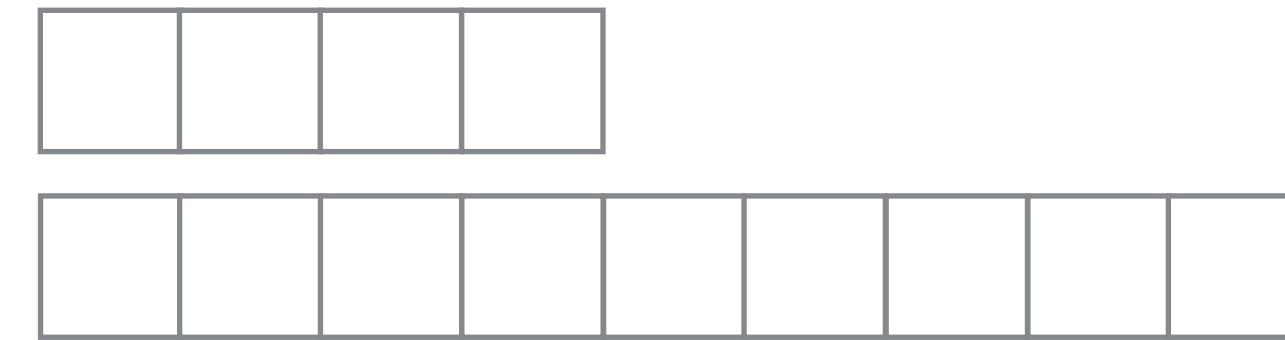
Compressed iterate

Level types can be composed in many ways

Dense



Compressed



Singleton

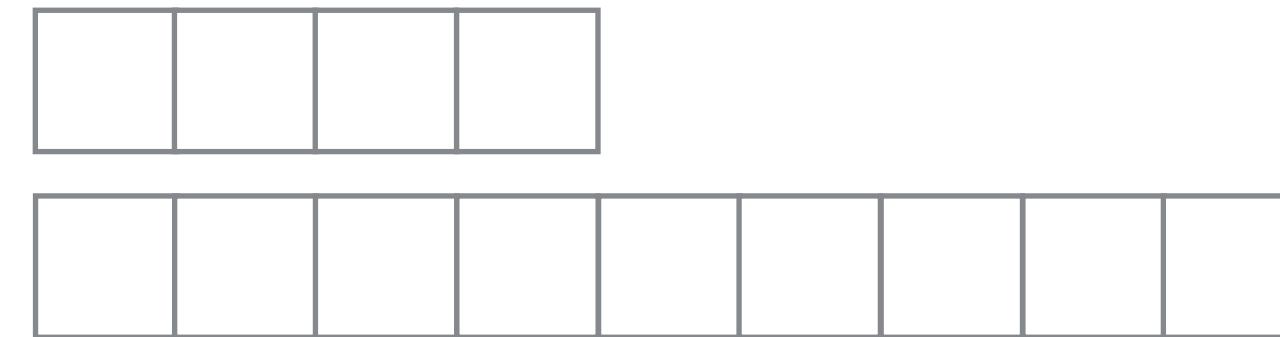


Level types can be composed in many ways

Dense



Compressed



Singleton



0 | 1 | 2 | 3

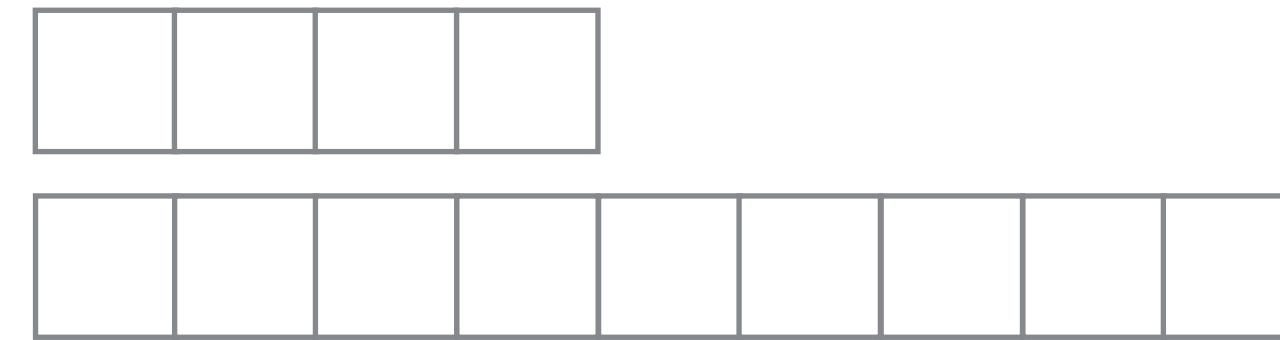
A		B	
C	D	E	
			F

Level types can be composed in many ways

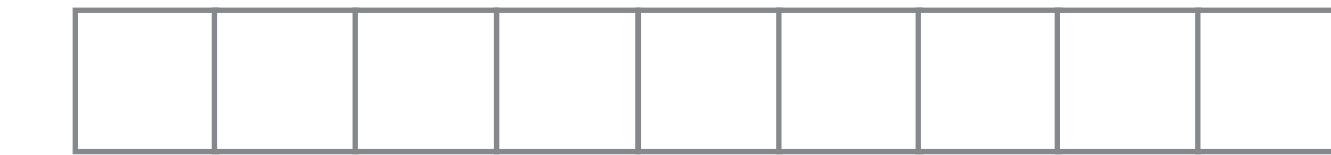
Dense



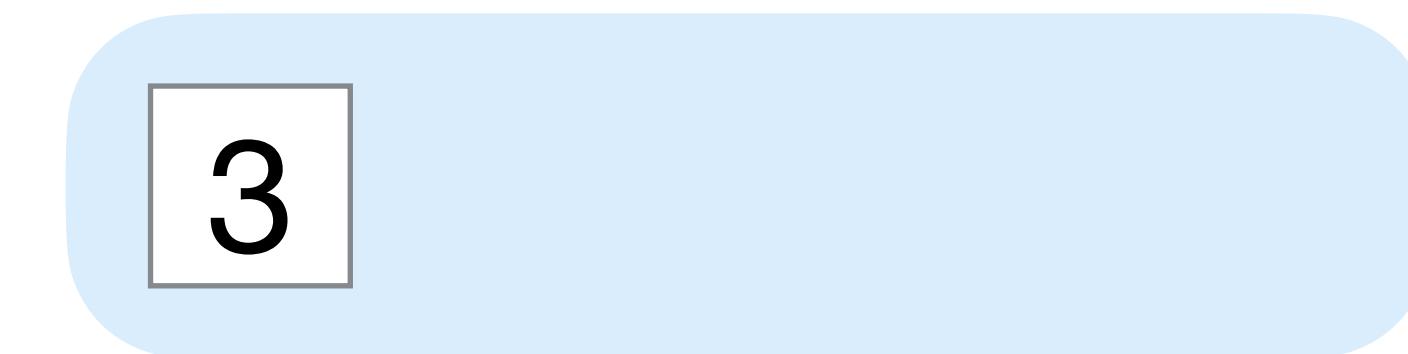
Compressed



Singleton



Dense



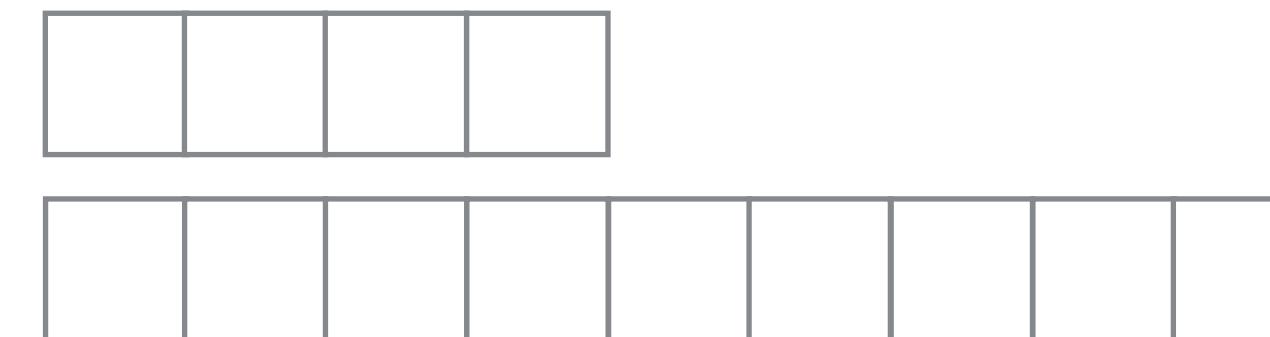
0	A		B	
1		C	D	E
2				F

Level types can be composed in many ways

Dense



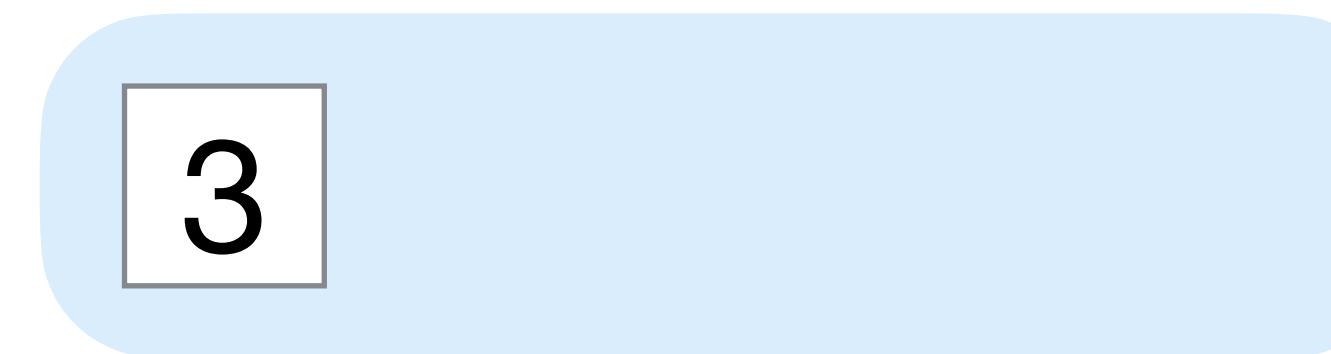
Compressed



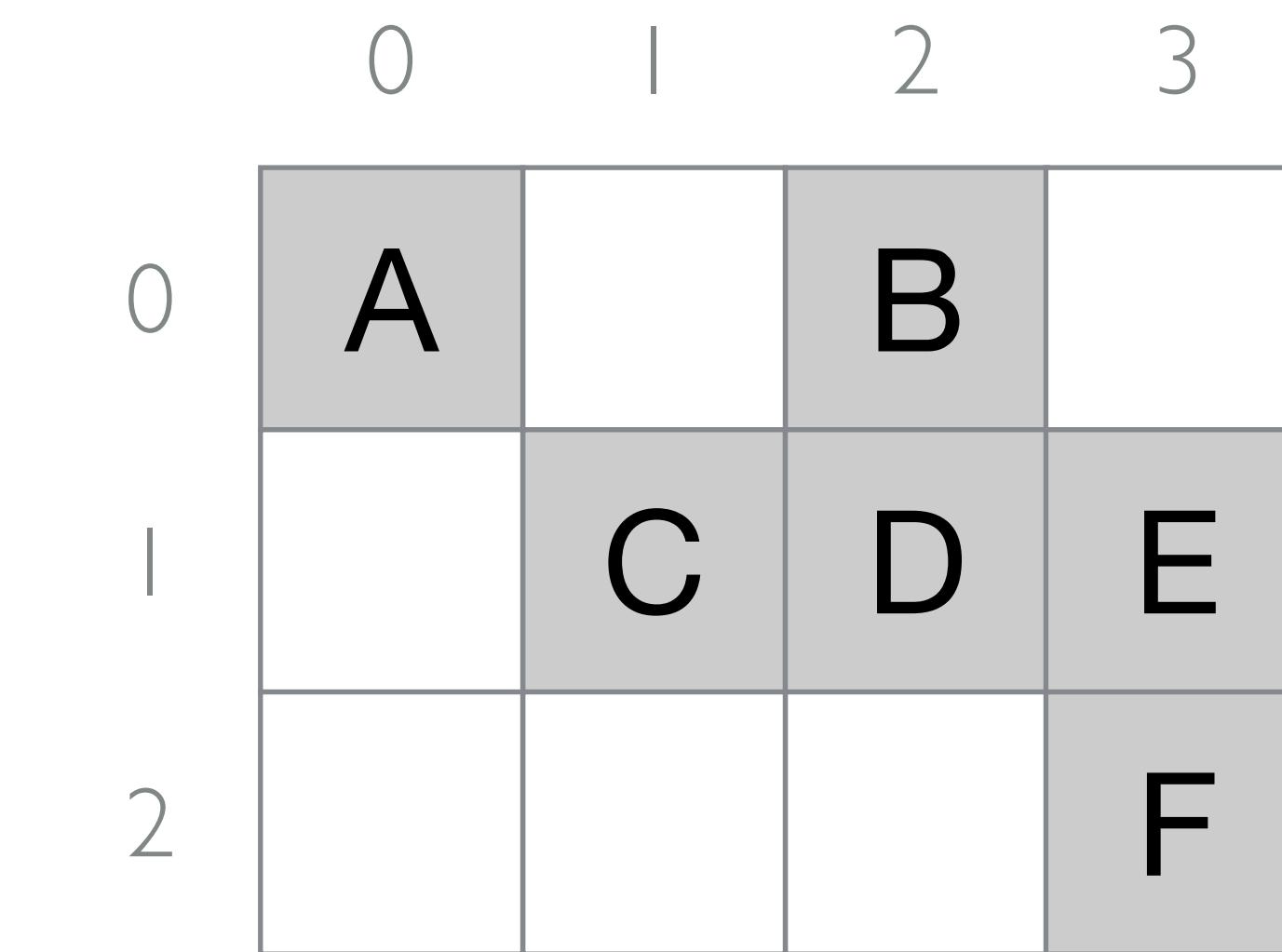
Singleton



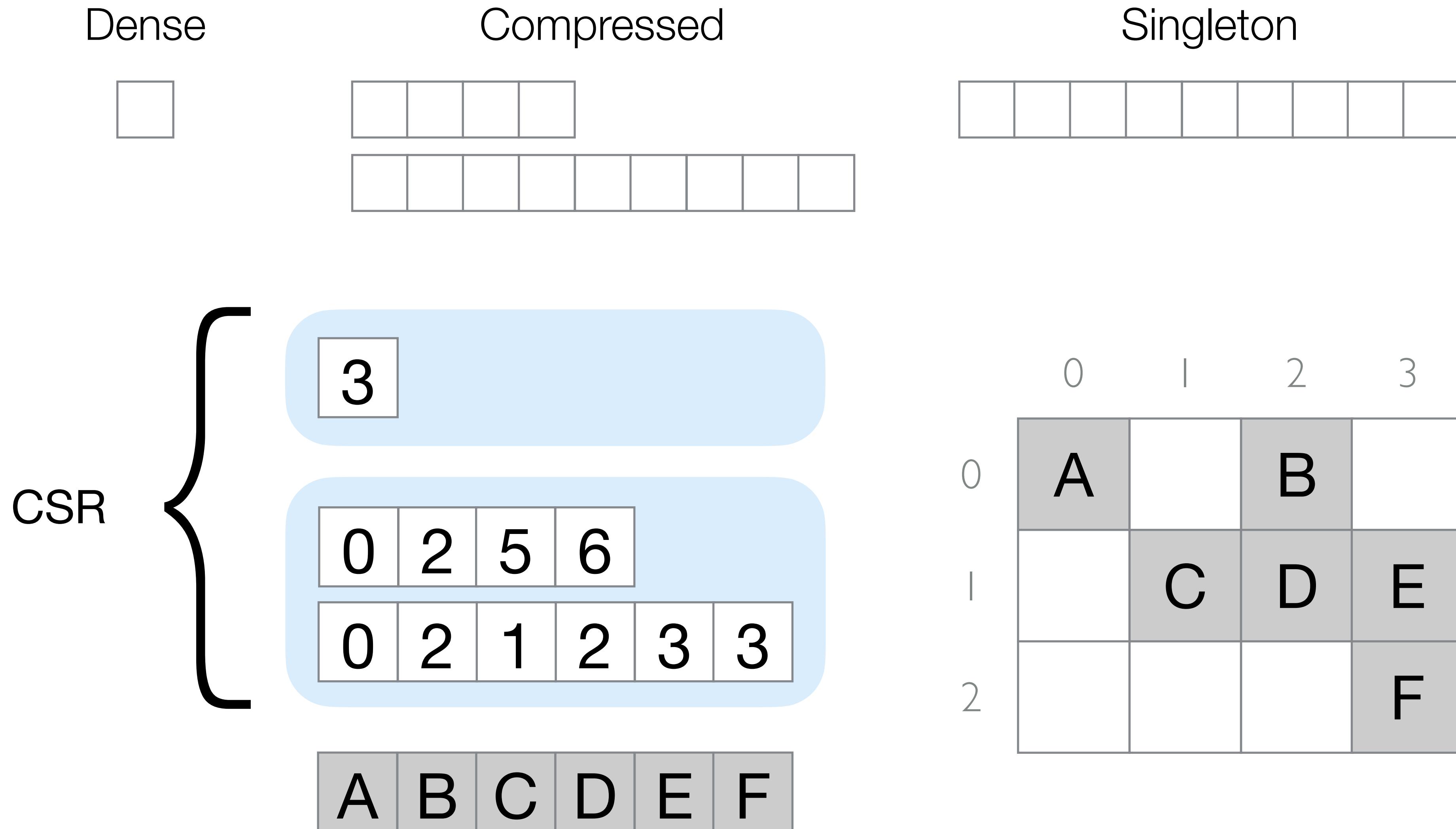
Dense



Compressed



Level types can be composed in many ways

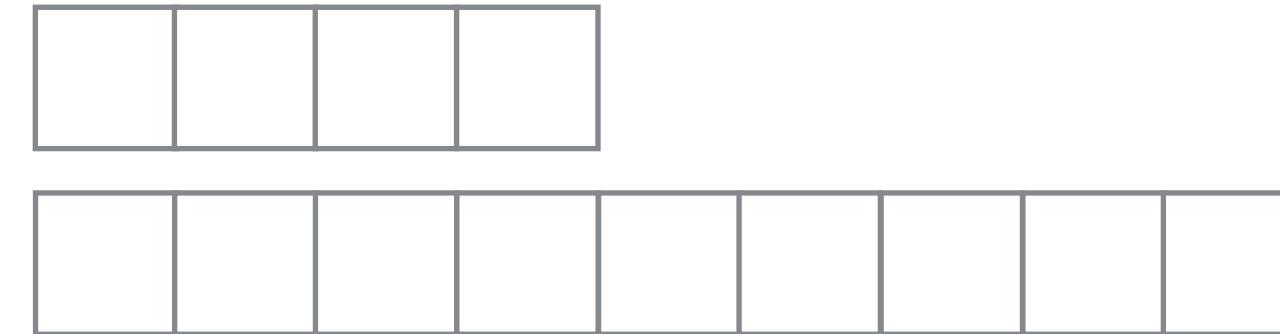


Level types can be composed in many ways

Dense



Compressed



Singleton



0 | 1 | 2 | 3

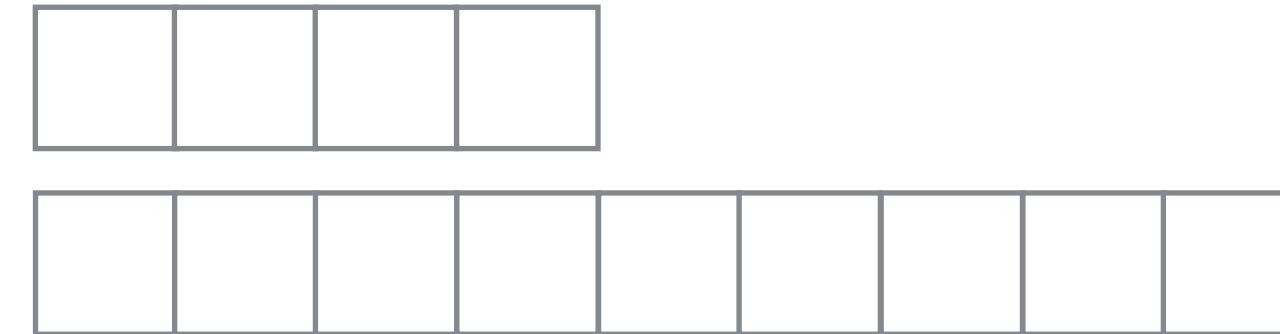
A		B	
	C	D	E
			F

Level types can be composed in many ways

Dense



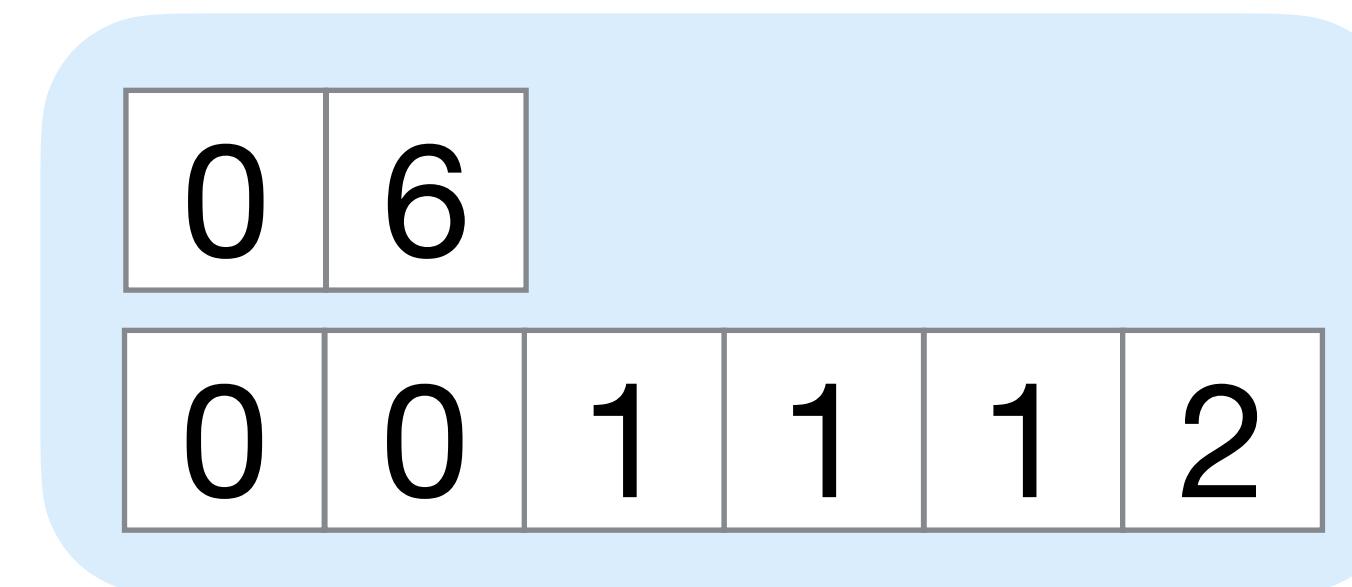
Compressed



Singleton

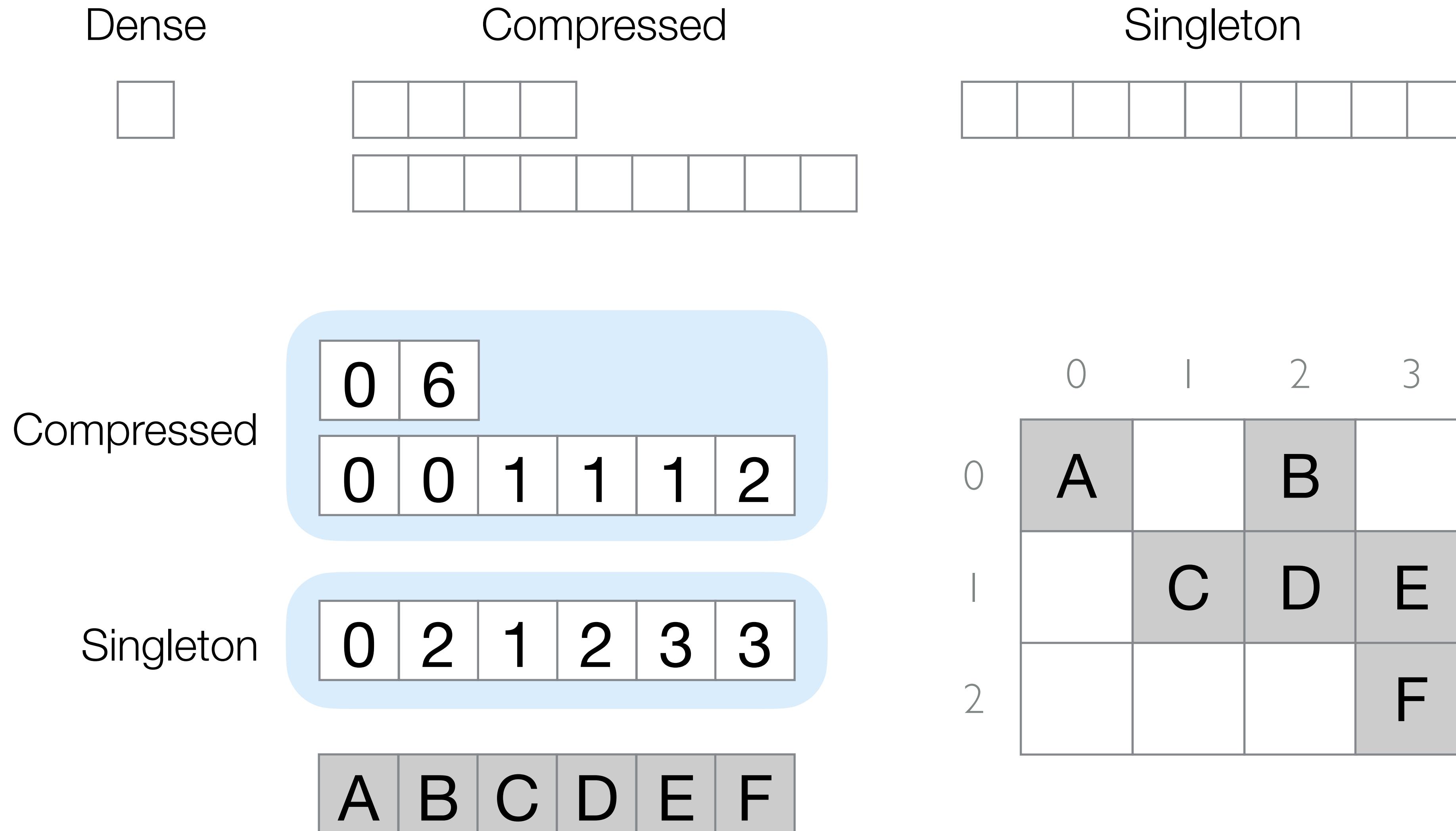


Compressed

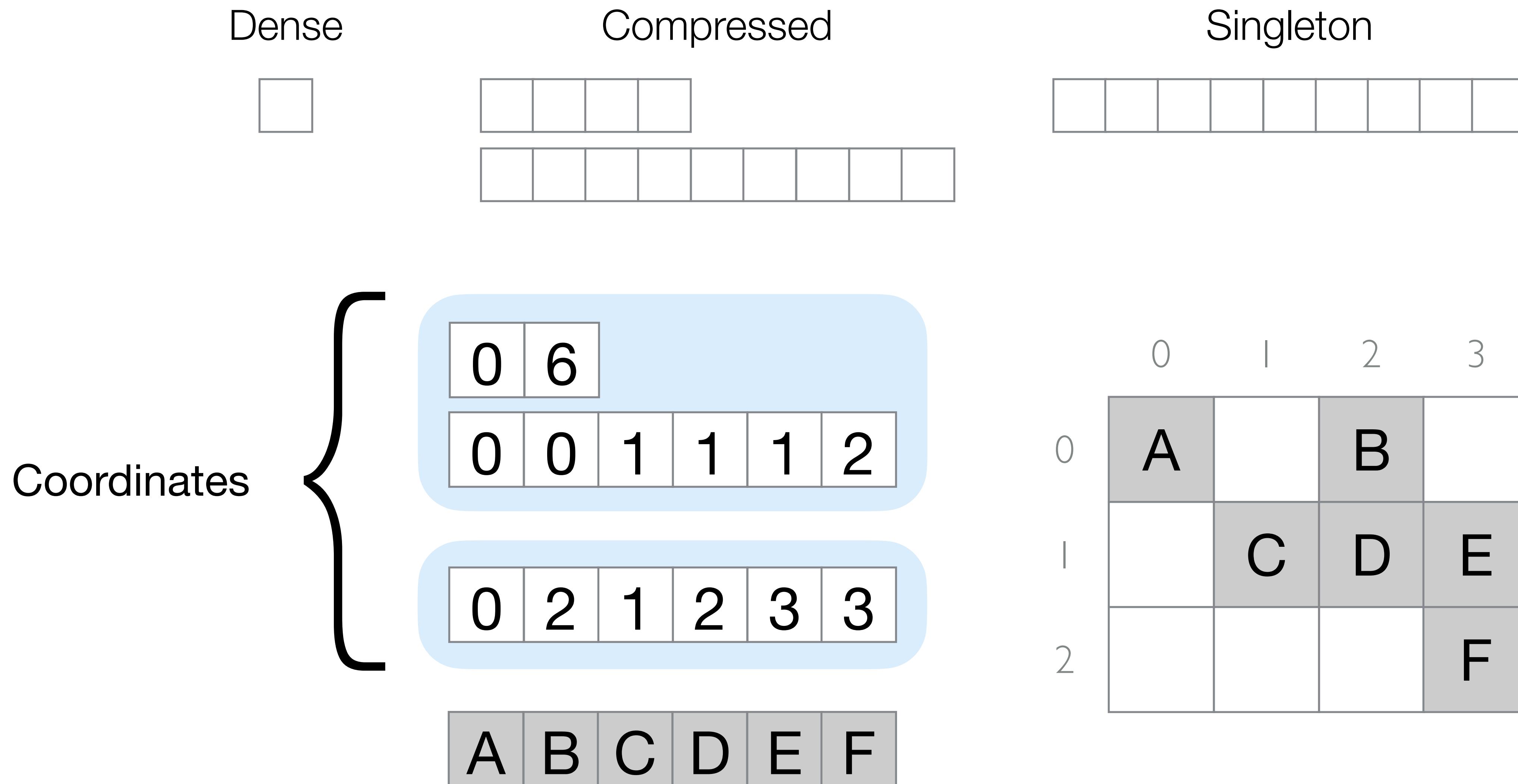


0	A	B	
1	C	D	E
2			F

Level types can be composed in many ways



Level types can be composed in many ways



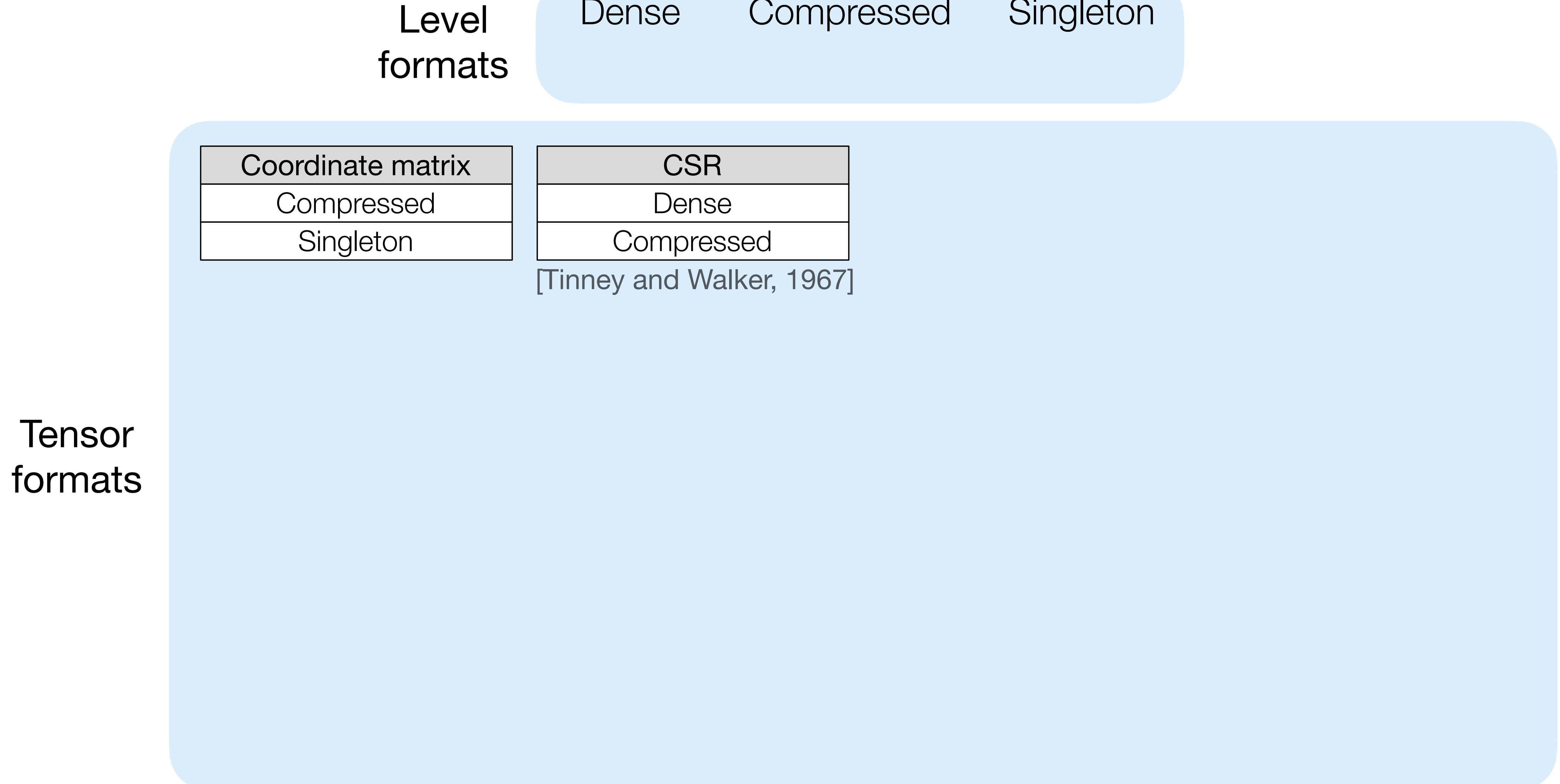
Level types can be composed in many ways

Dense

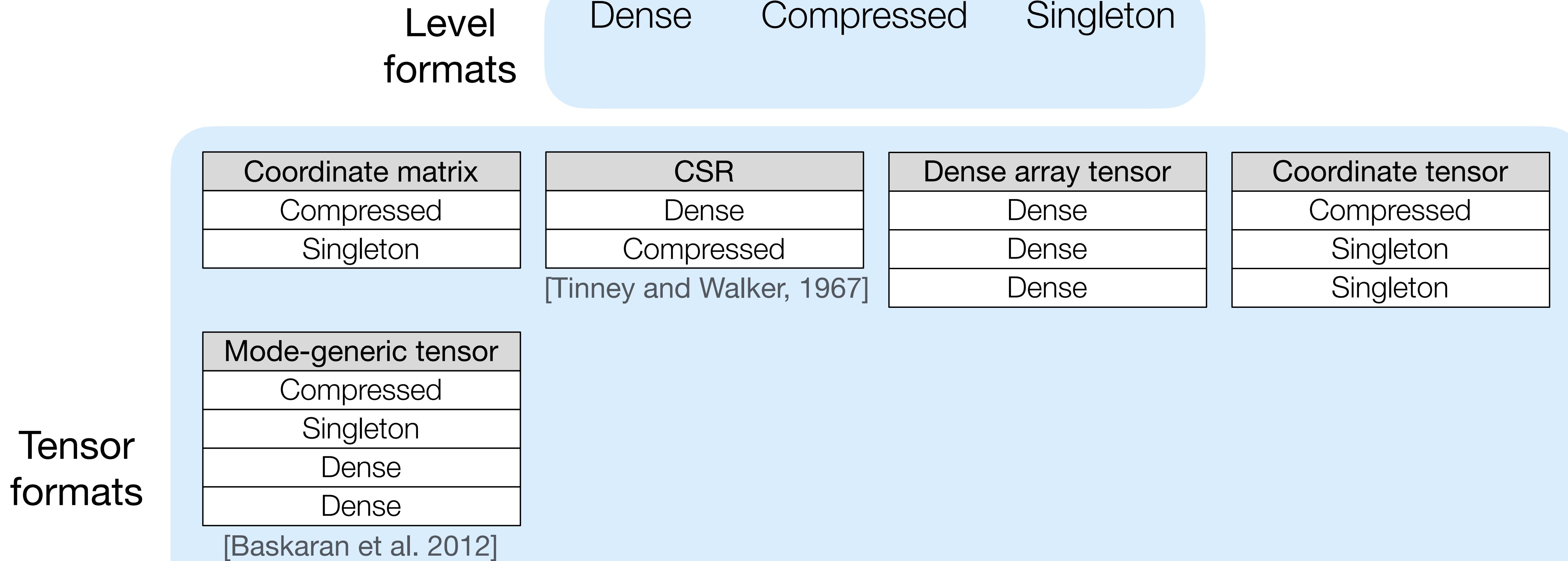
Compressed

Singleton

Level types can be composed in many ways



Level types can be composed in many ways



Level types can be composed in many ways

Tensor formats	Level formats	Dense	Compressed	Singleton	
		Coordinate matrix	CSR	Dense array tensor	Coordinate tensor
	Compressed	Compressed	Dense	Dense	Compressed
	Singleton	Singleton	Compressed	Dense	Singleton
			[Tinney and Walker, 1967]	Dense	Singleton
	Mode-generic tensor	BCSR	CSB	ELLPACK	
	Compressed	Dense	Dense	Dense	
	Singleton	Compressed	Dense	Dense	
	Dense	Dense	Compressed	Singleton	
	Dense	Dense	Singleton		[Kincaid et al. 1989]
			[Buluç et al. 2009]		
			[Im and Yelick 1998]		
			[Baskaran et al. 2012]		

Level types can be composed in many ways

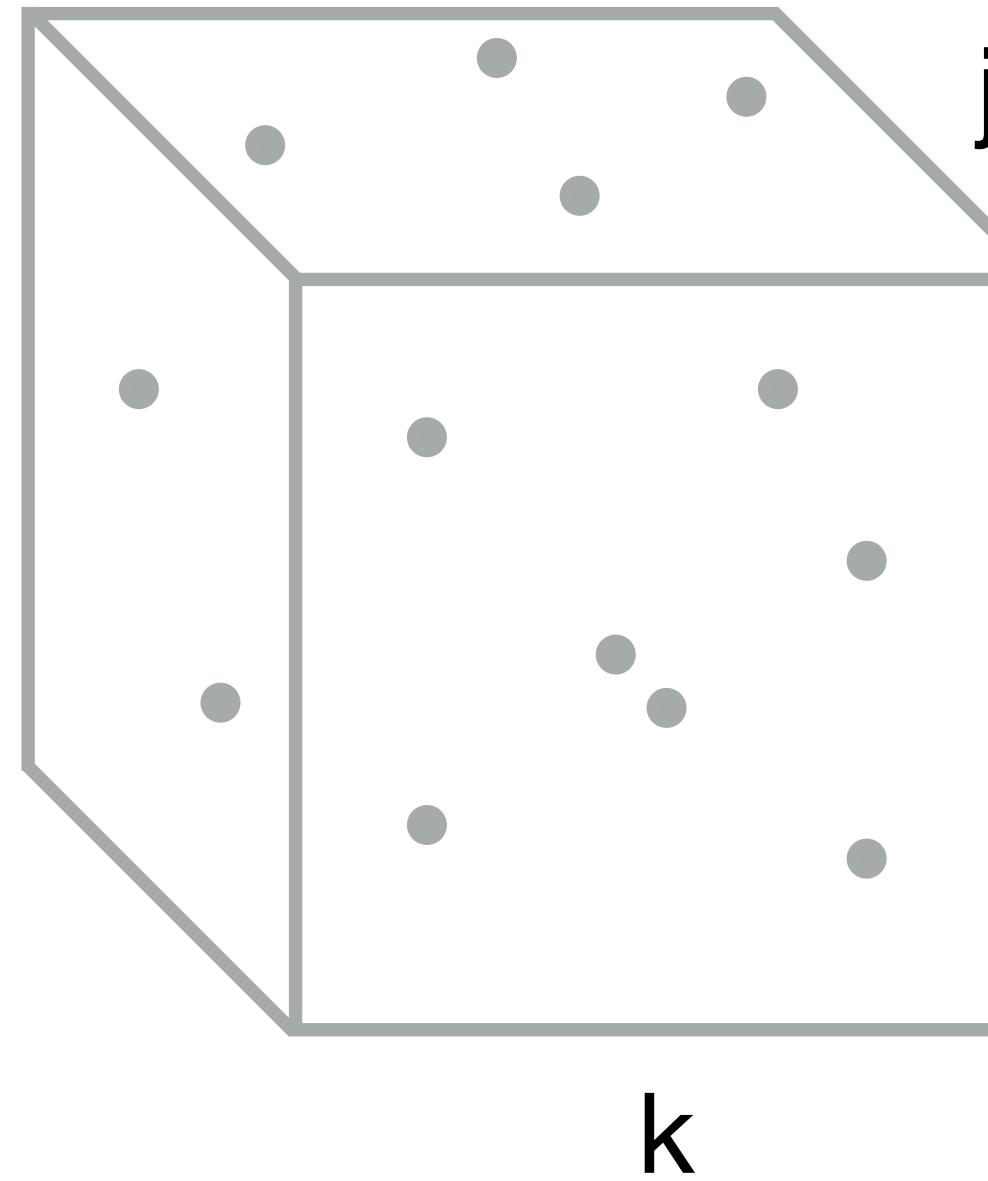
Tensor formats	Level formats	Dense	Compressed	Singleton
		Hashed	Range	Offset
Coordinate matrix	CSR	Dense	Compressed	Singleton
Compressed	Dense	Hashed	Range	Offset
Singleton	Compressed	Range	Offset	
[Tinney and Walker, 1967]				
Mode-generic tensor	BCSR	Dense	CSB	ELLPACK
Compressed	Dense	Hashed	Dense	Dense
Singleton	Compressed	Range	Dense	Dense
Dense	Dense	Offset	Compressed	Singleton
Dense	Dense		Singleton	
[Baskaran et al. 2012]	[Im and Yelick 1998]		[Buluç et al. 2009]	[Kincaid et al. 1989]

Level types can be composed in many ways

Tensor formats	Level formats	Dense	Compressed	Singleton
		Hashed	Range	Offset
Coordinate matrix	CSR	Dense	Dense array tensor	Coordinate tensor
	Compressed	Dense	Dense	Compressed
	Singleton	Compressed	Dense	Singleton
	[Tinney and Walker, 1967]		Dense	Singleton
Mode-generic tensor	BCSR	Dense	CSB	ELLPACK
	Compressed	Dense	Dense	Dense
	Singletton	Compressed	Compressed	Dense
	Dense	Dense	Dense	Singletton
	[Baskaran et al. 2012]	Dense	Singleton	[Kincaid et al. 1989]
Hash map vector	Hash map matrix	DIA	Block DIA	
	Hashed	Dense	Dense	Dense
	[Patwary et al. 2015]	Range	Range	Range
Hash map matrix	Hash map matrix	Offset	Offset	Offset
	Hashed	Dense	Dense	Dense
	Hashed	Range	Range	Range
	[Saad 2003]	Offset	Offset	Offset

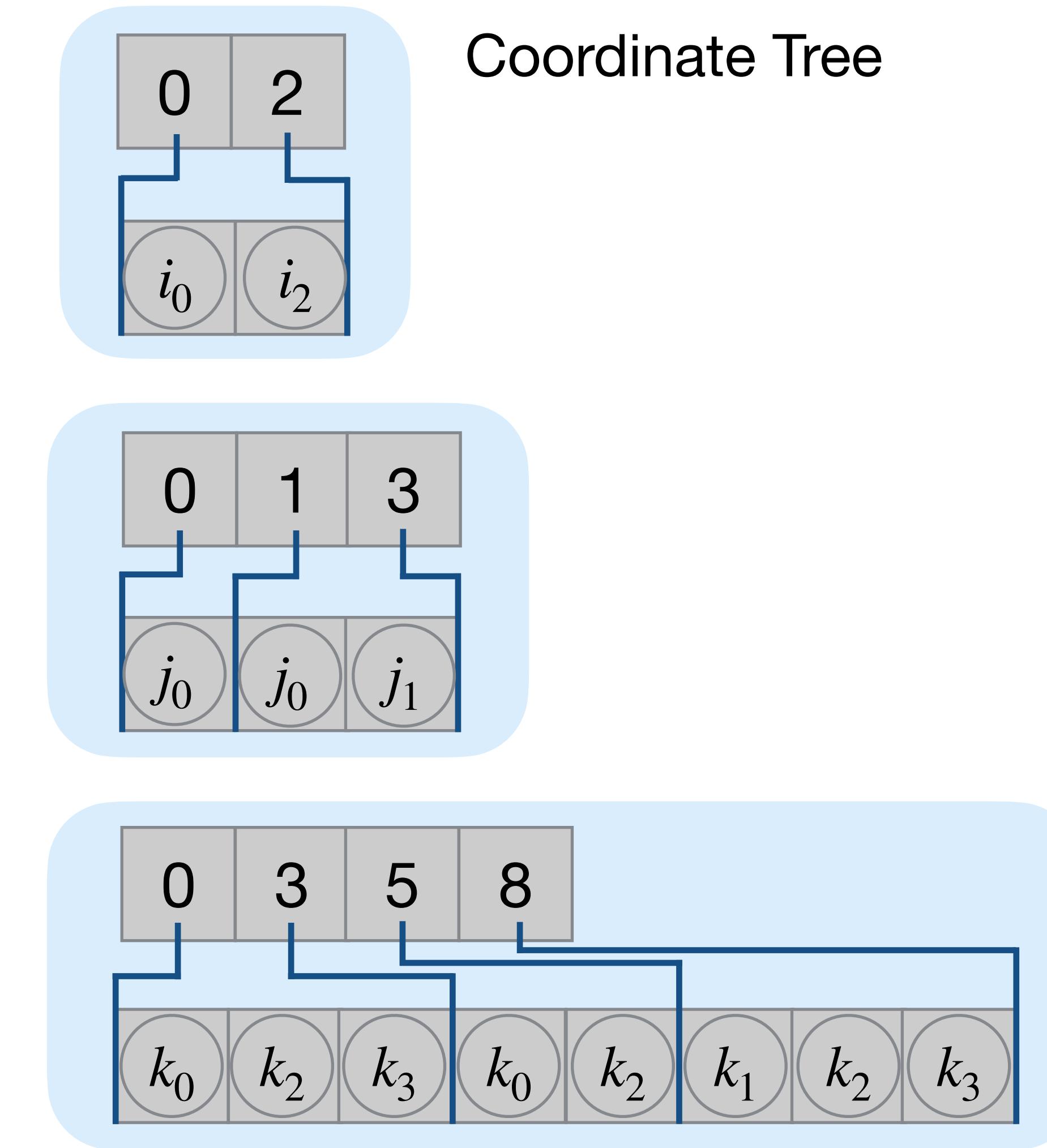
Iteration graphs express iteration spaces and data structure ordering

Iteration Space



B_{ijk}

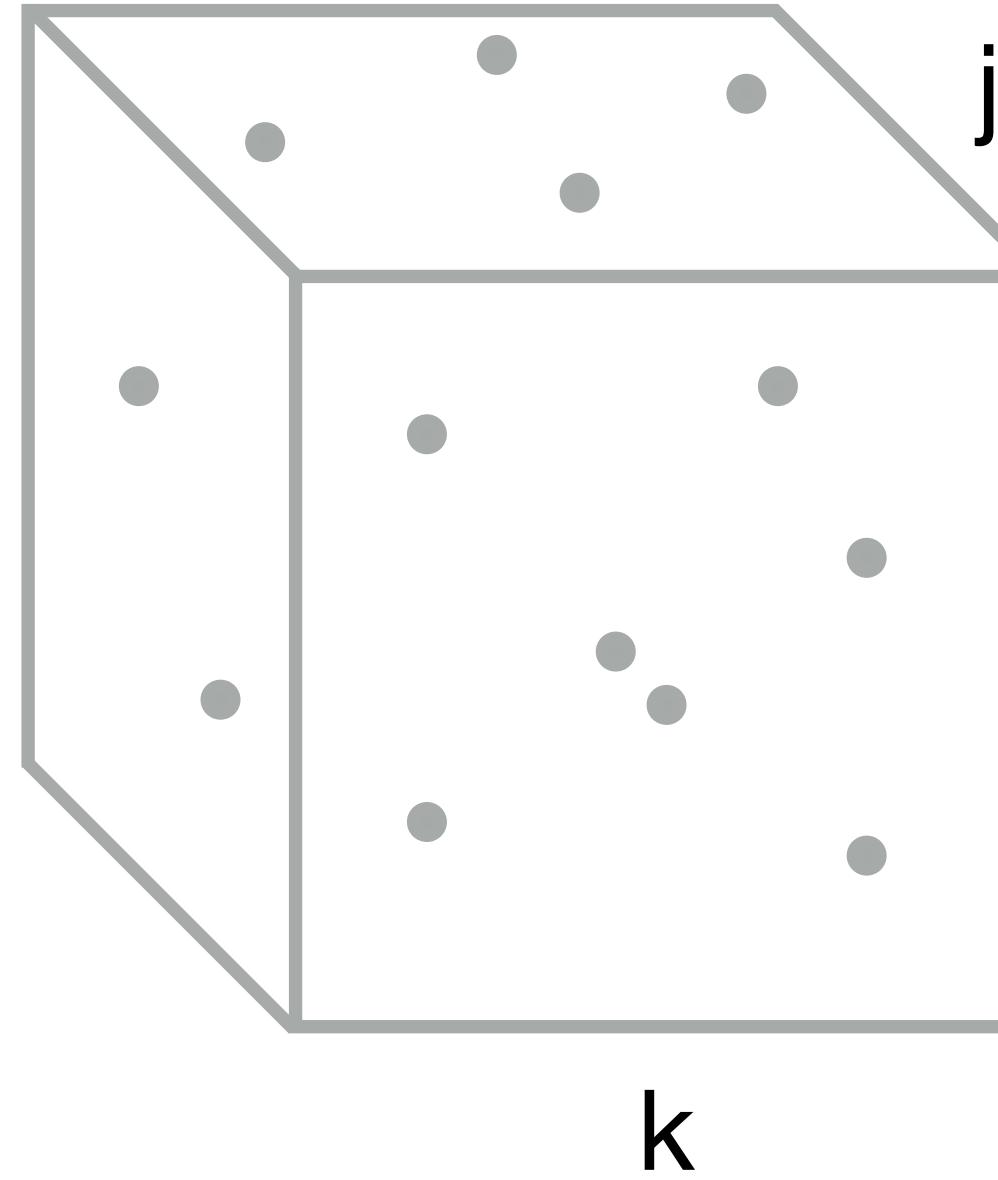
Coordinate Tree



30	40	50	10	70	80	20	60
----	----	----	----	----	----	----	----

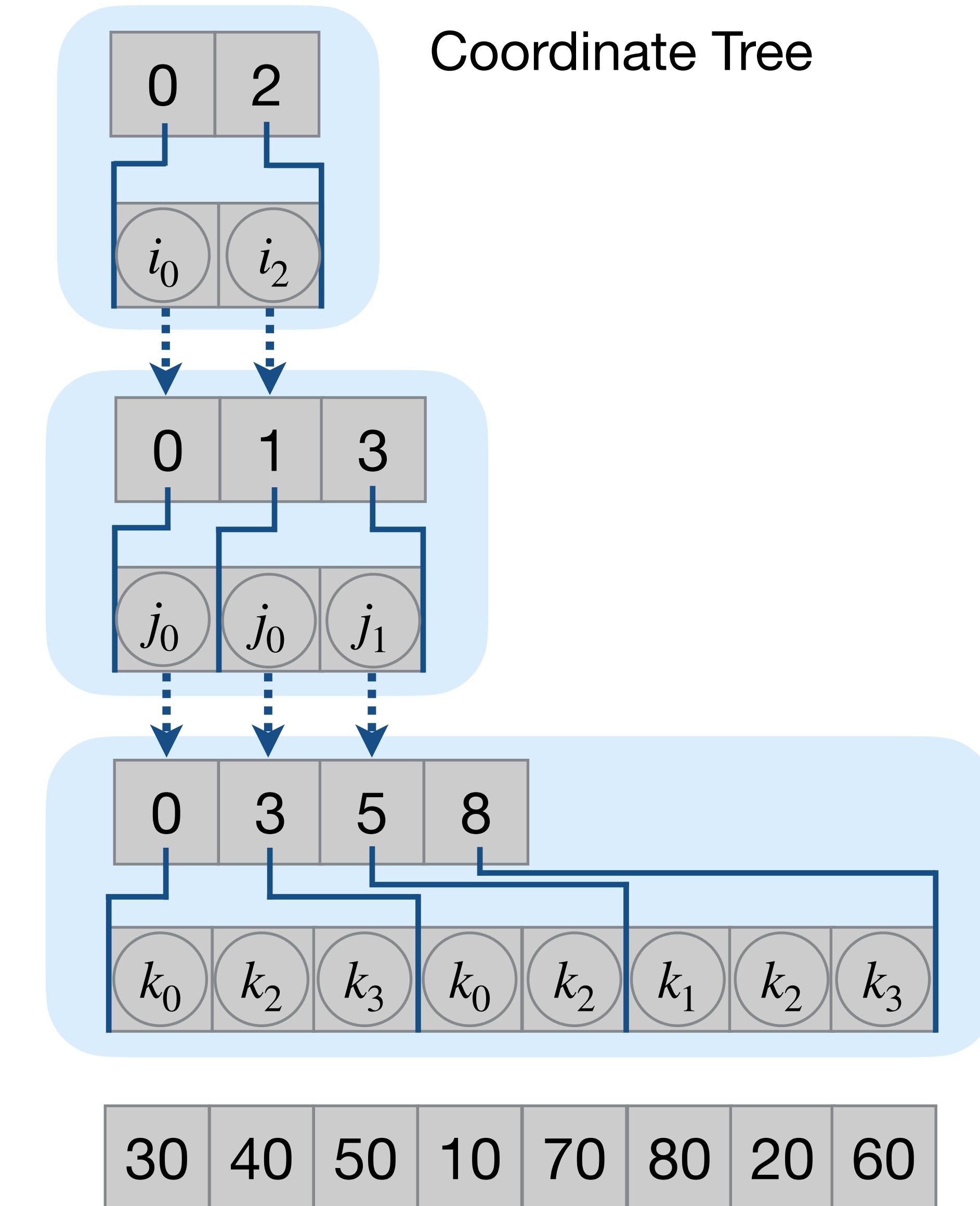
Iteration graphs express iteration spaces and data structure ordering

Iteration Space



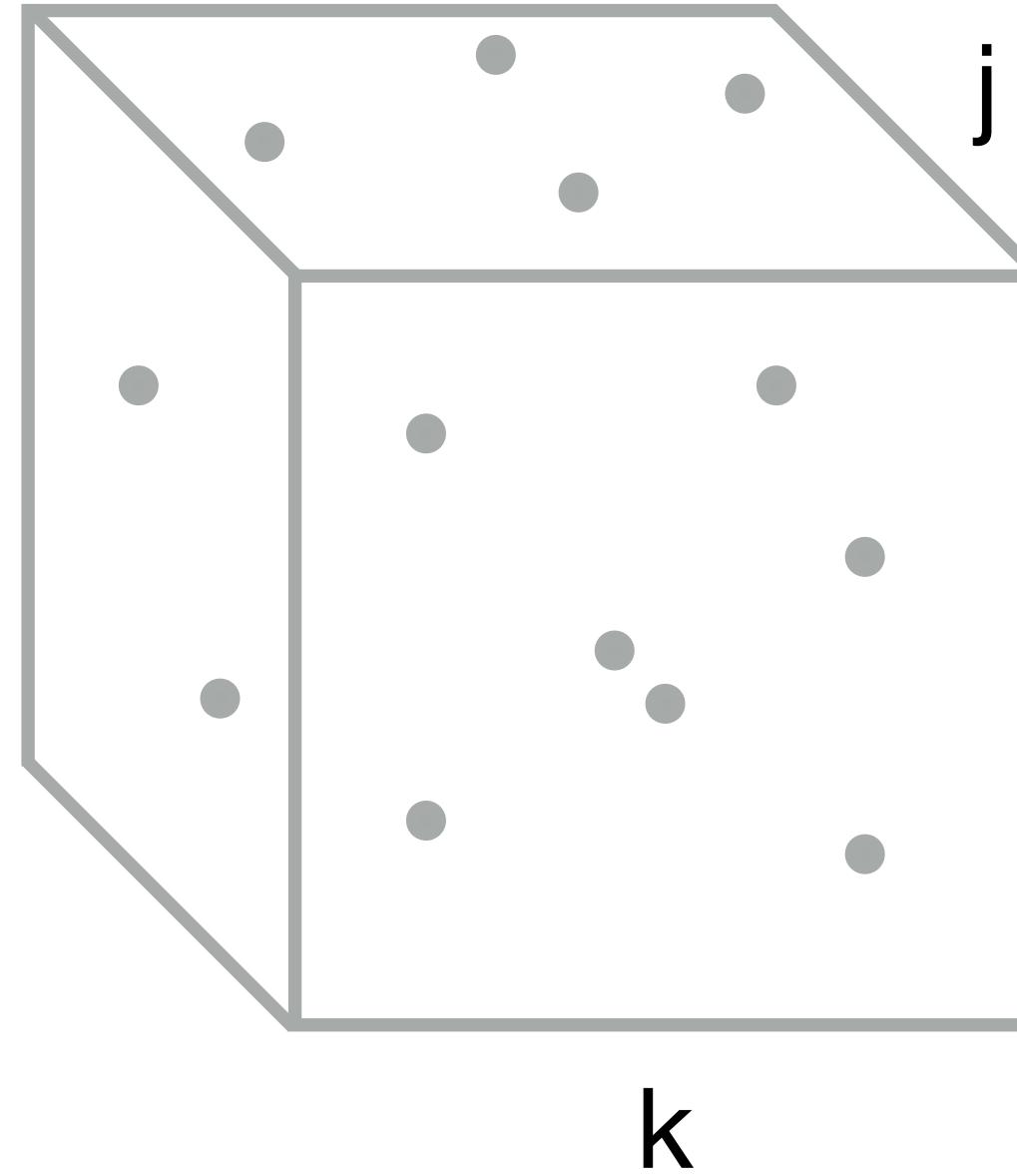
B_{ijk}

Coordinate Tree



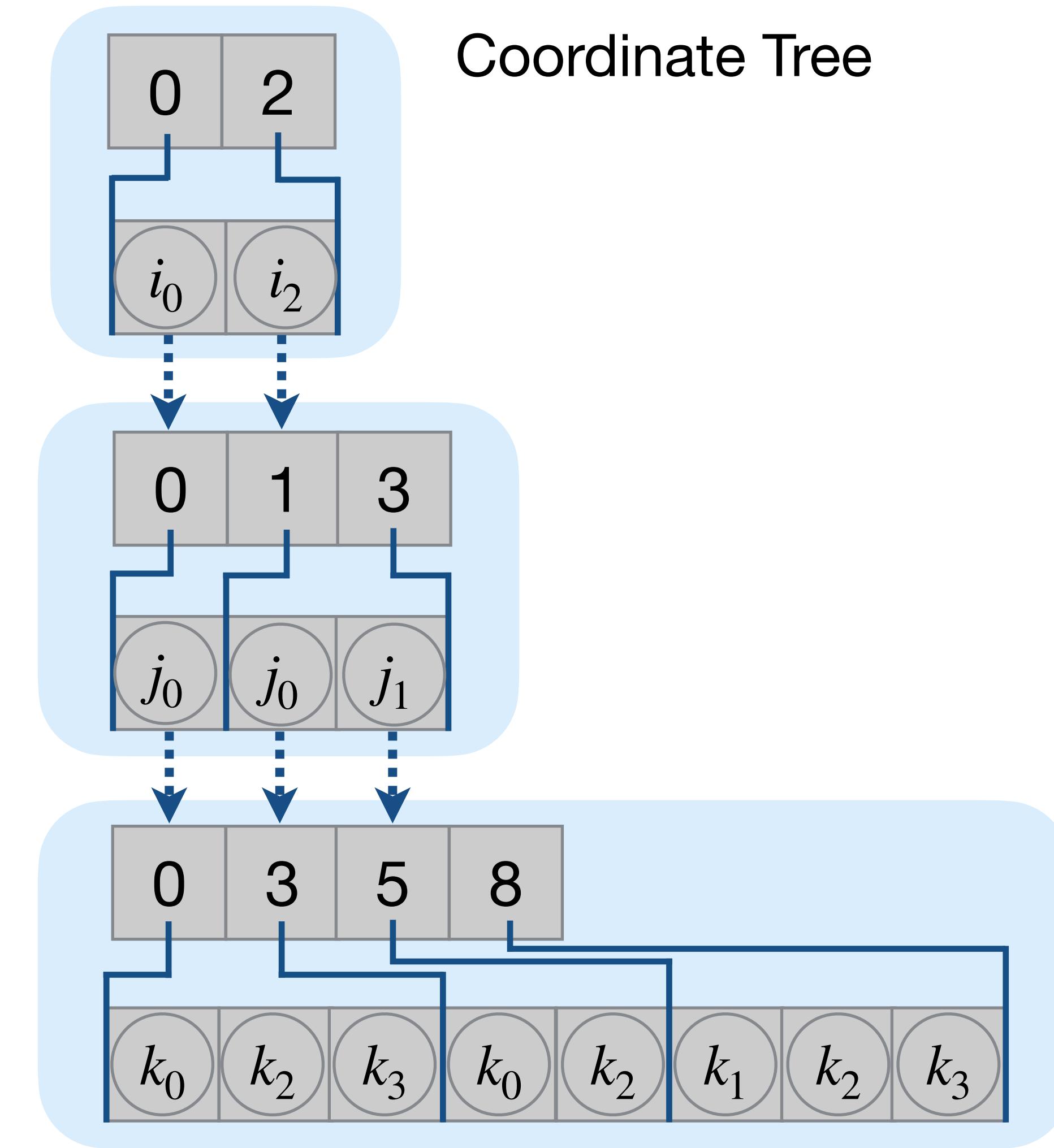
Iteration graphs express iteration spaces and data structure ordering

Iteration Space



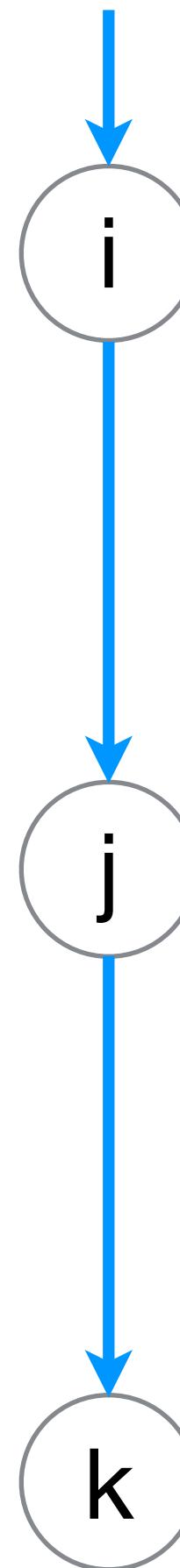
B_{ijk}

Coordinate Tree



30 40 50 10 70 80 20 60

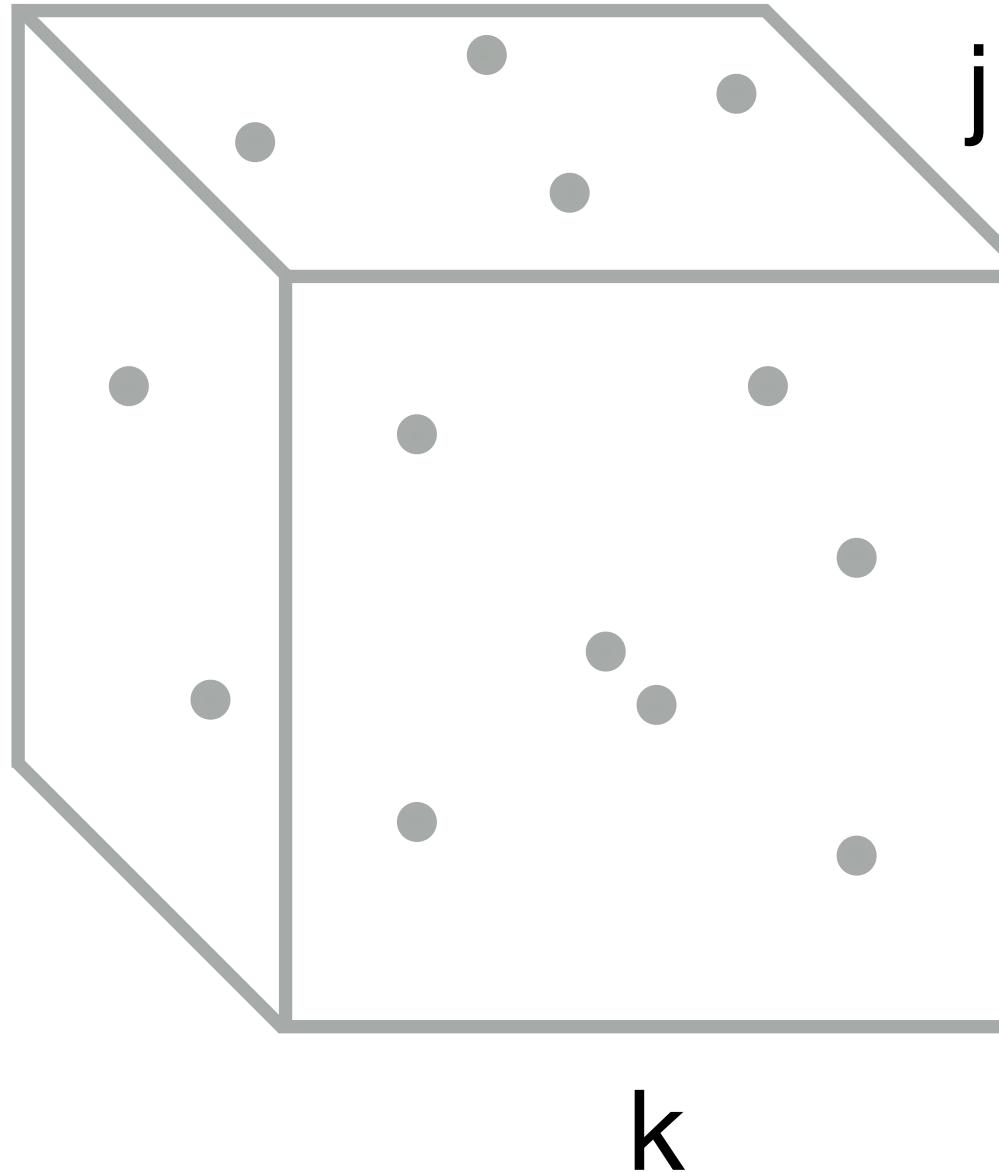
Iteration Graph



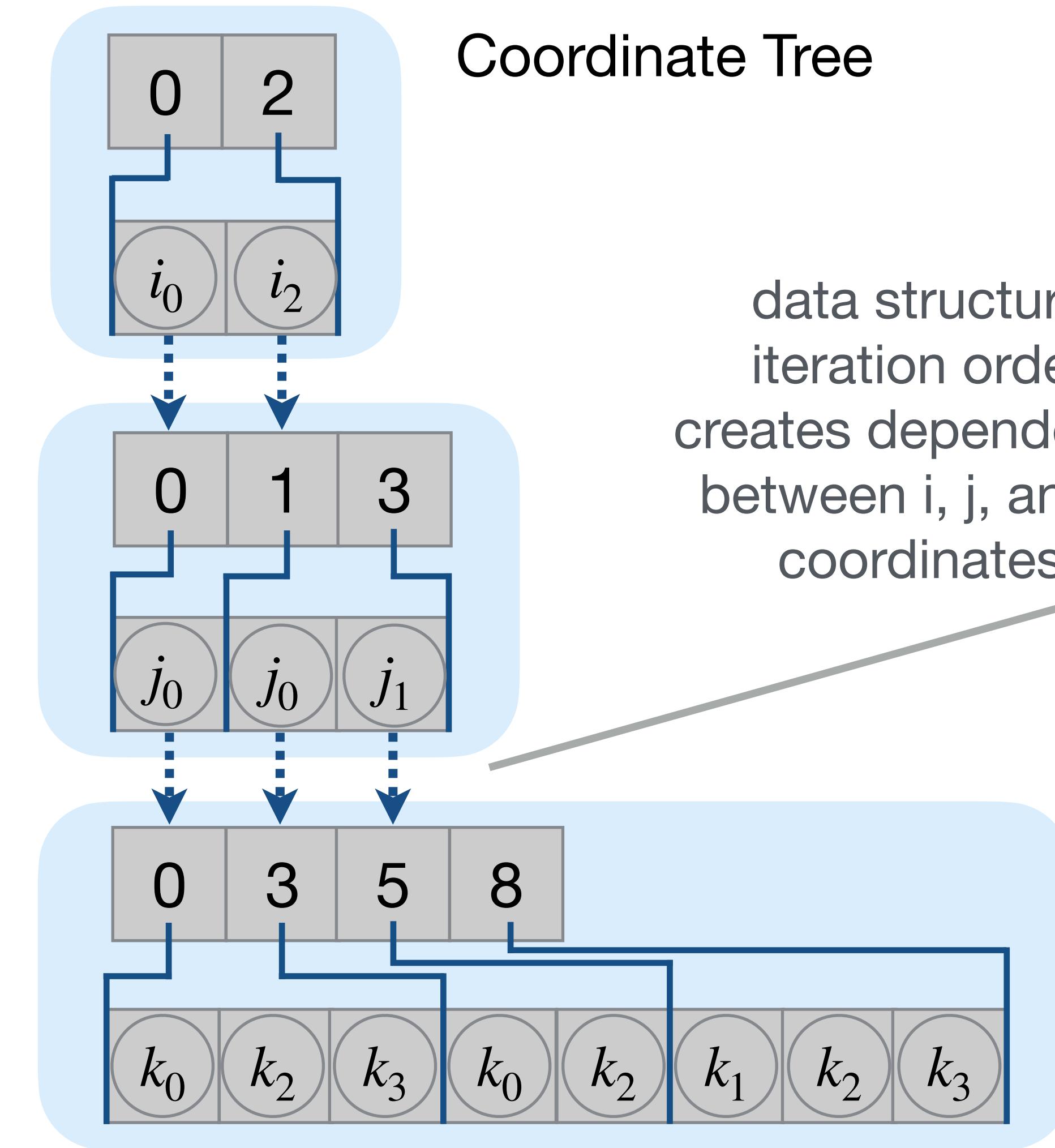
$\forall_i \forall_j \forall_k B_{ijk}$

Iteration graphs express iteration spaces and data structure ordering

Iteration Space



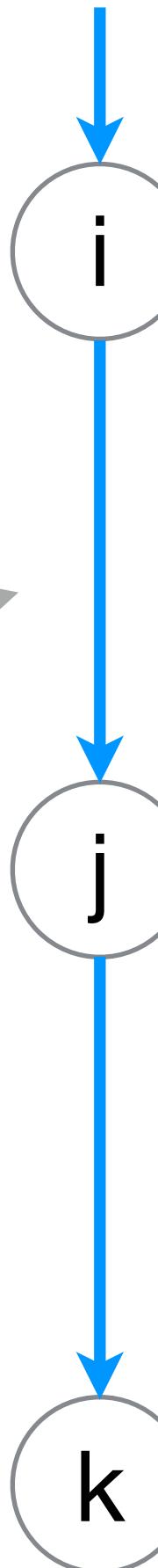
B_{ijk}



Coordinate Tree

data structure
iteration order
creates dependency
between *i*, *j*, and *k*
coordinates

Iteration Graph



30 40 50 10 70 80 20 60

$\forall_i \forall_j \forall_k B_{ijk}$

Sparse iteration spaces and Iteration Graphs

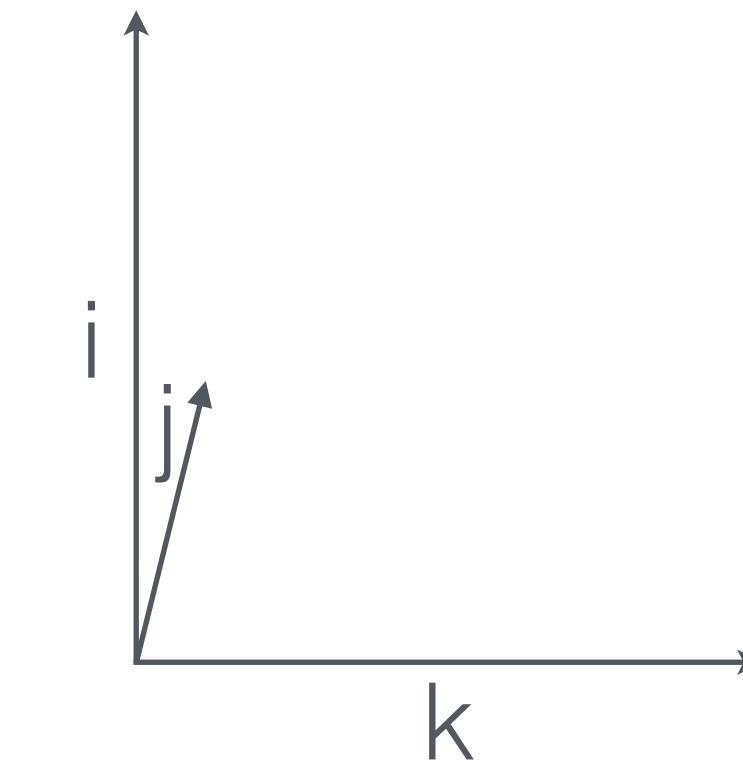
$$A_{ij} = \sum_k B_{ijk} * c_k$$

Sparse iteration spaces and Iteration Graphs

$$A_{ij} = \sum_k B_{ijk} * c_k$$

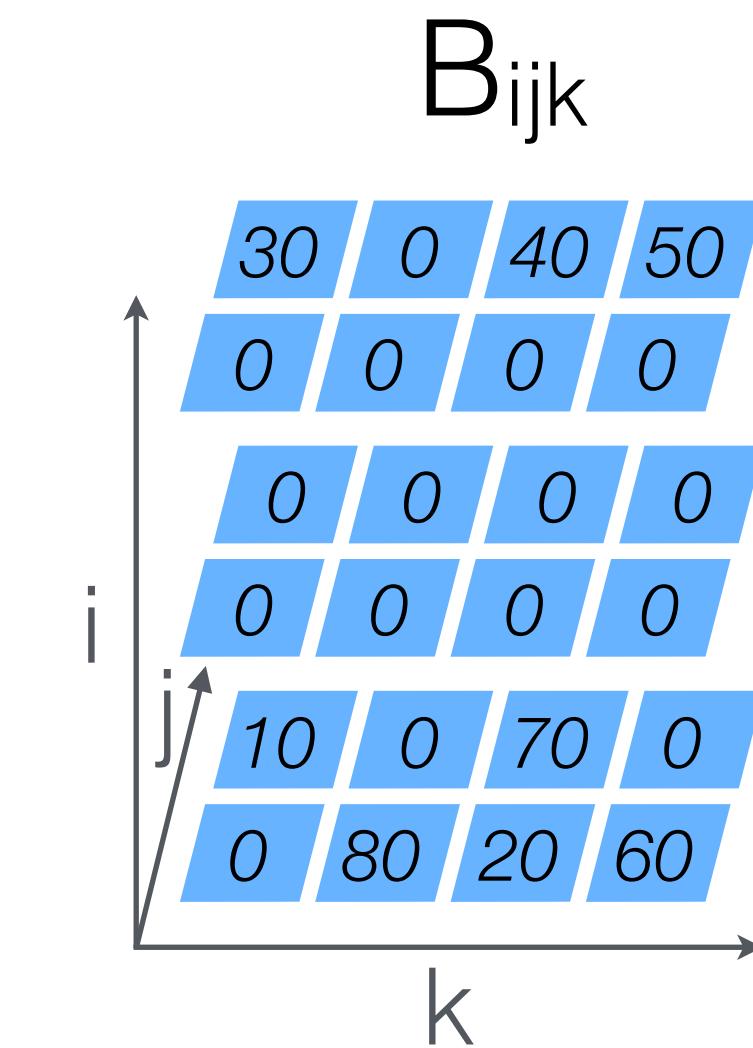

Sparse iteration spaces and Iteration Graphs

$$A_{ij} = \sum_k B_{ijk} * c_k$$



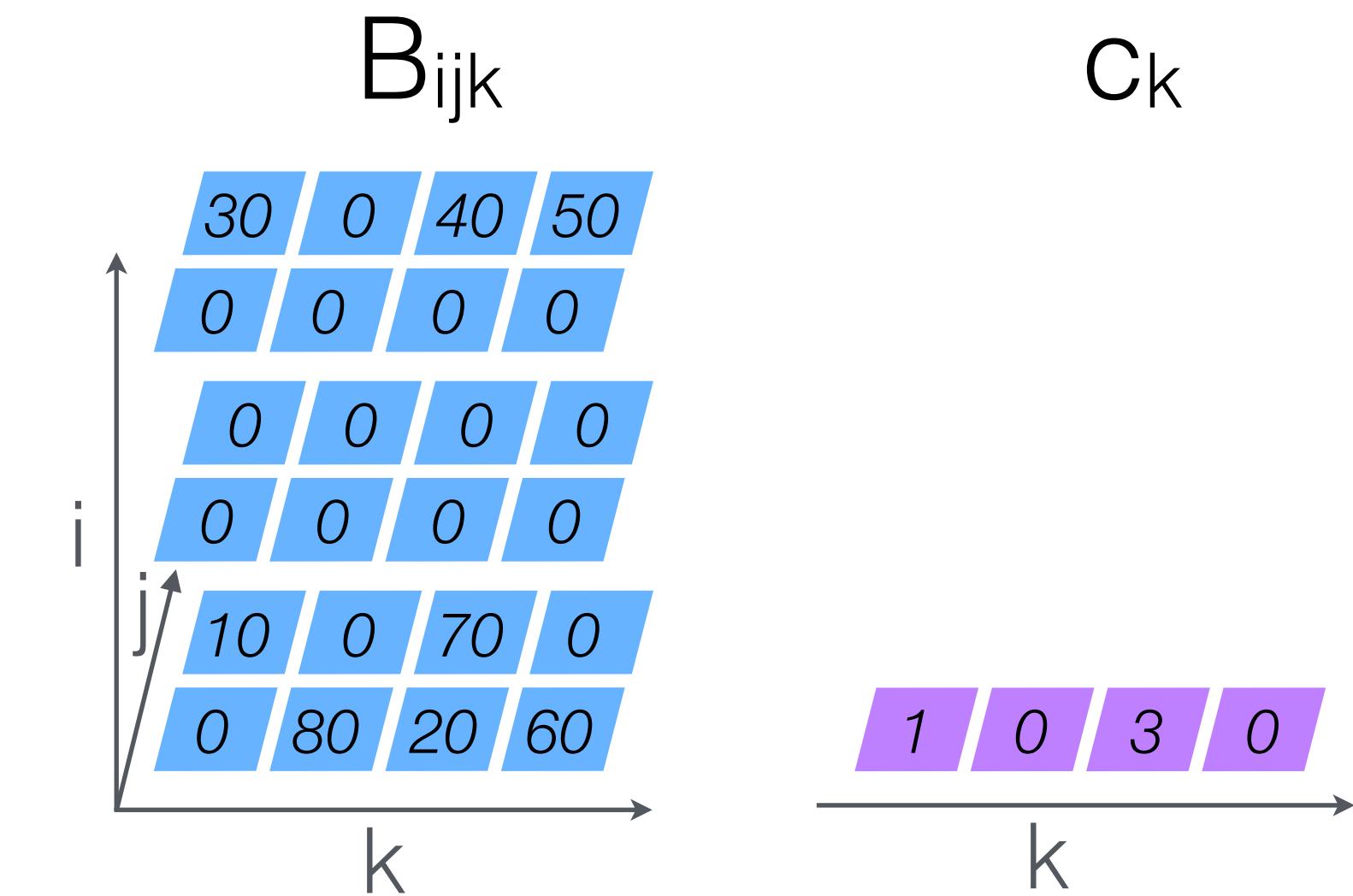
Sparse iteration spaces and Iteration Graphs

$$A_{ij} = \sum_k B_{ijk} * c_k$$



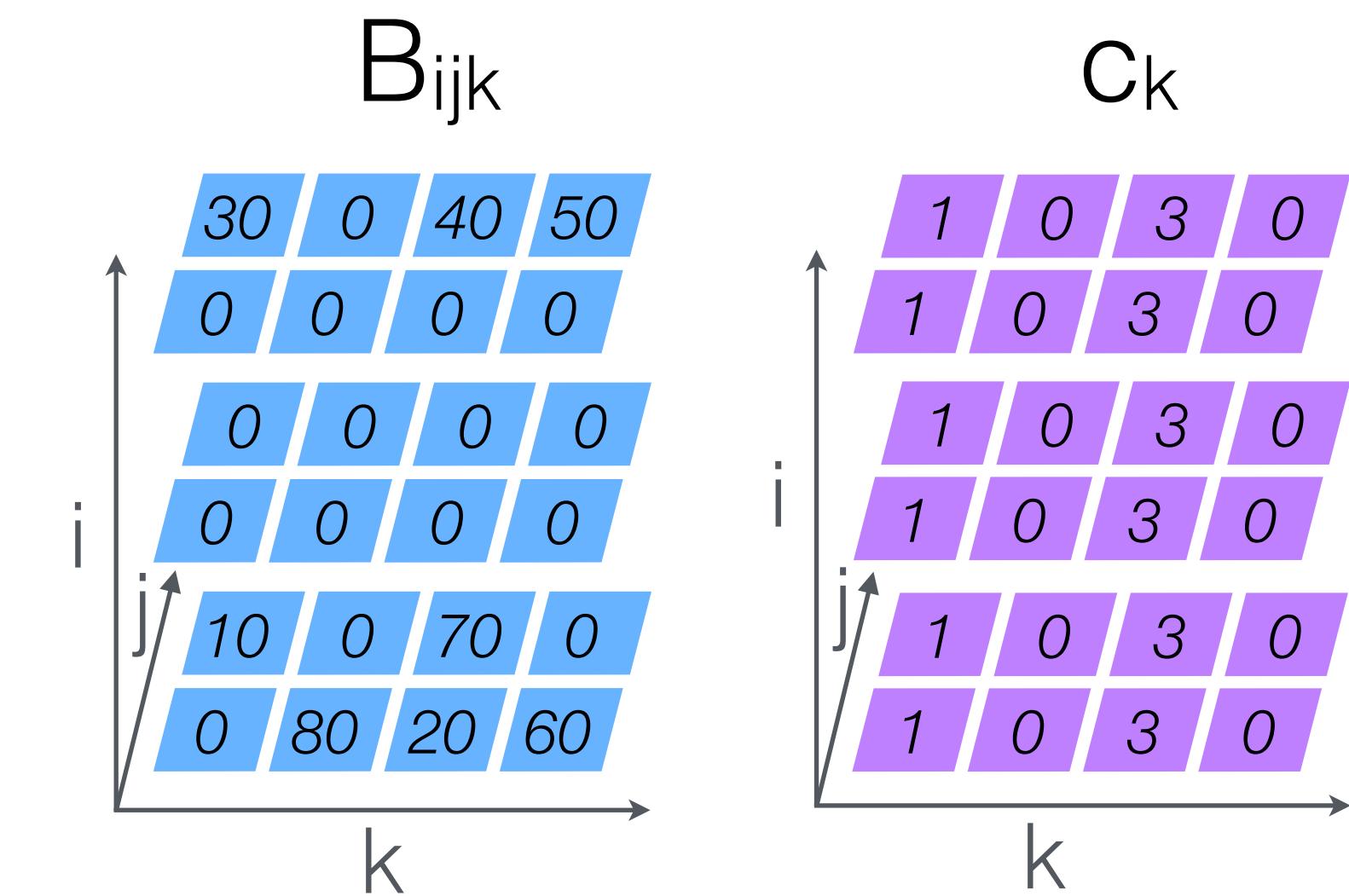
Sparse iteration spaces and Iteration Graphs

$$A_{ij} = \sum_k B_{ijk} * c_k$$



Sparse iteration spaces and Iteration Graphs

$$A_{ij} = \sum_k B_{ijk} * C_k$$



Sparse iteration spaces and Iteration Graphs

$$A_{ij} = \sum_k B_{ijk} * C_k$$

B _{ijk} * C _k			
i	j	k	
		30	0
		0	120
		0	0
		0	0
		0	0
		10	210
		0	60

Dense

Sparse iteration spaces and Iteration Graphs

$$A_{ij} = \sum_k B_{ijk} * C_k$$

B_{ijk}

	30		40	50
i				
j	10		70	
		80	20	60
k				

C_k

1		3	
1		3	
1		3	
1		3	
1		3	

$B_{ijk} * C_k$

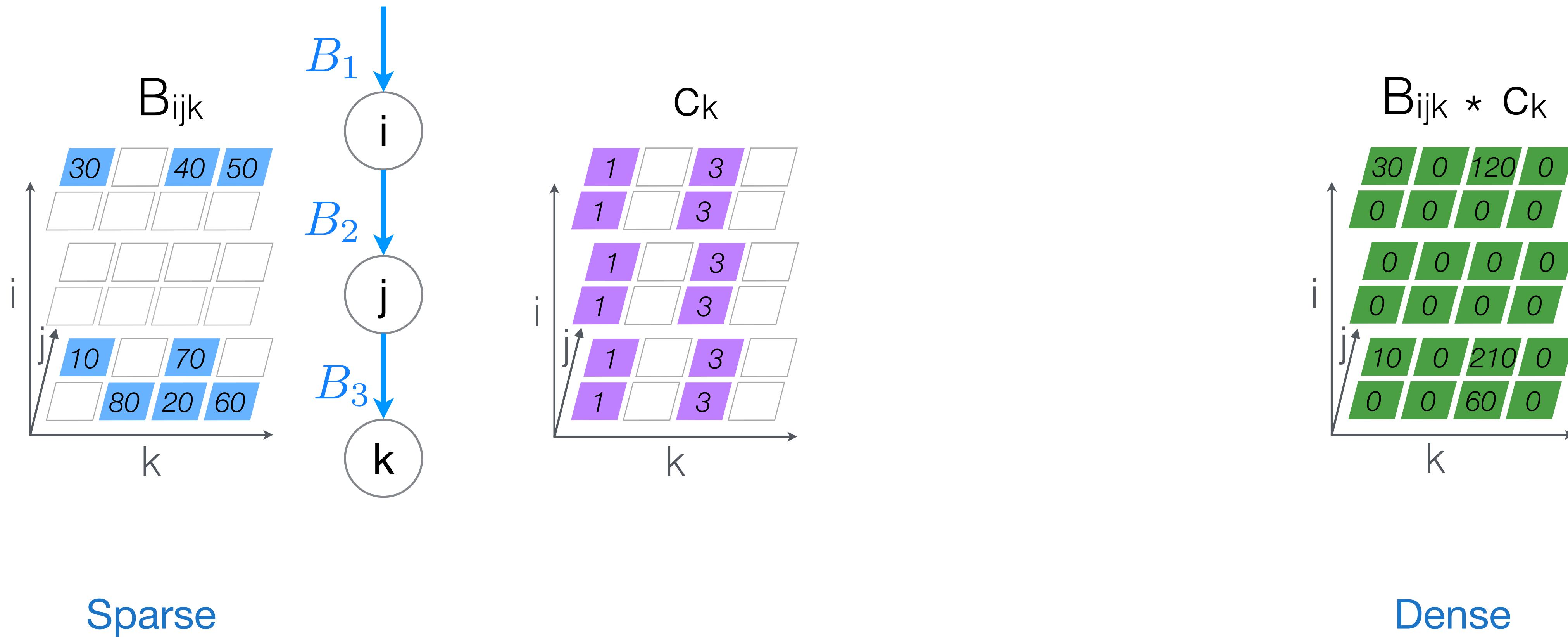
30	0	120	0
0	0	0	0
0	0	0	0
10	0	210	0
0	0	60	0

Sparse

Dense

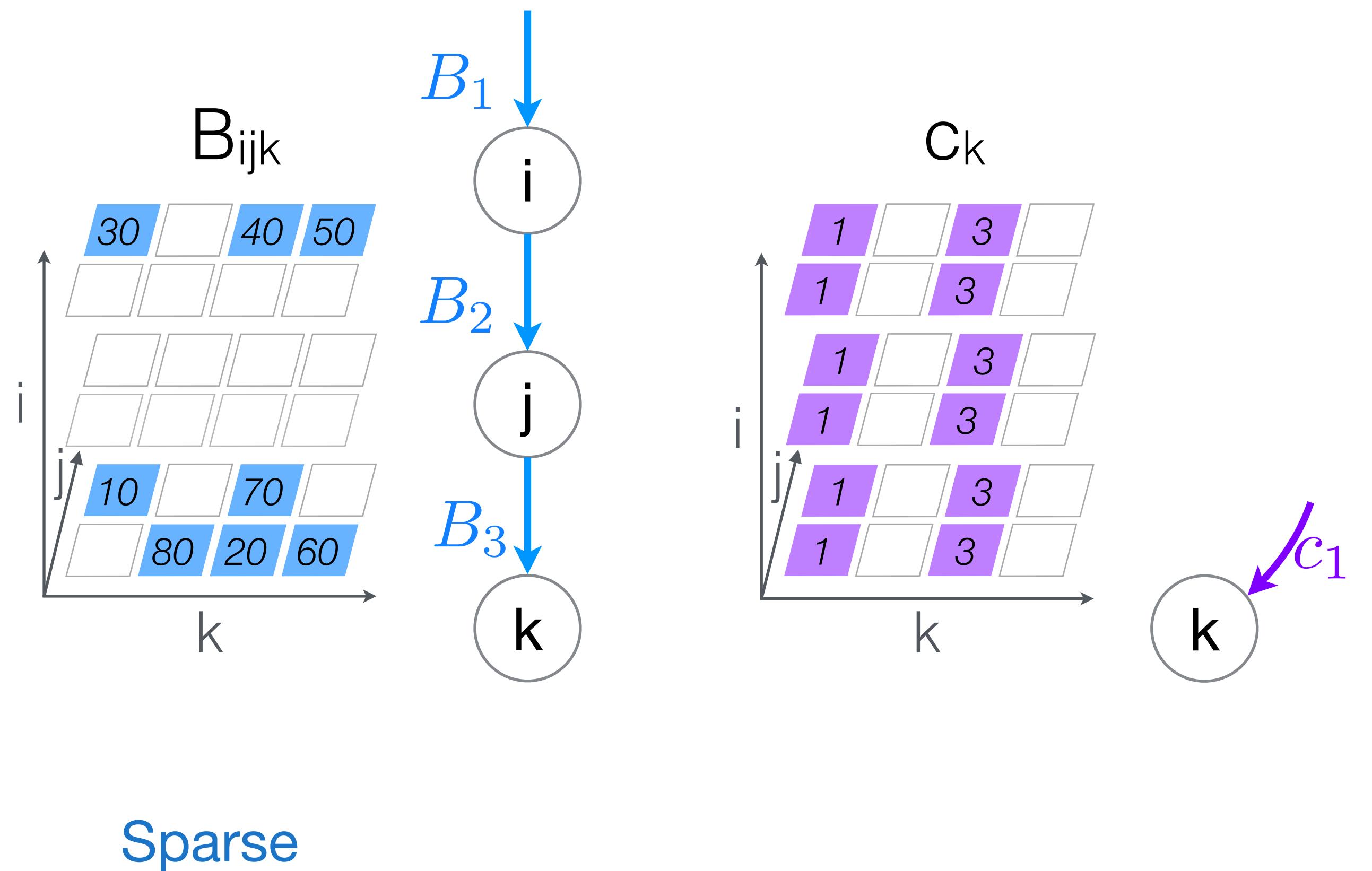
Sparse iteration spaces and Iteration Graphs

$$A_{ij} = \sum_k B_{ijk} * C_k$$



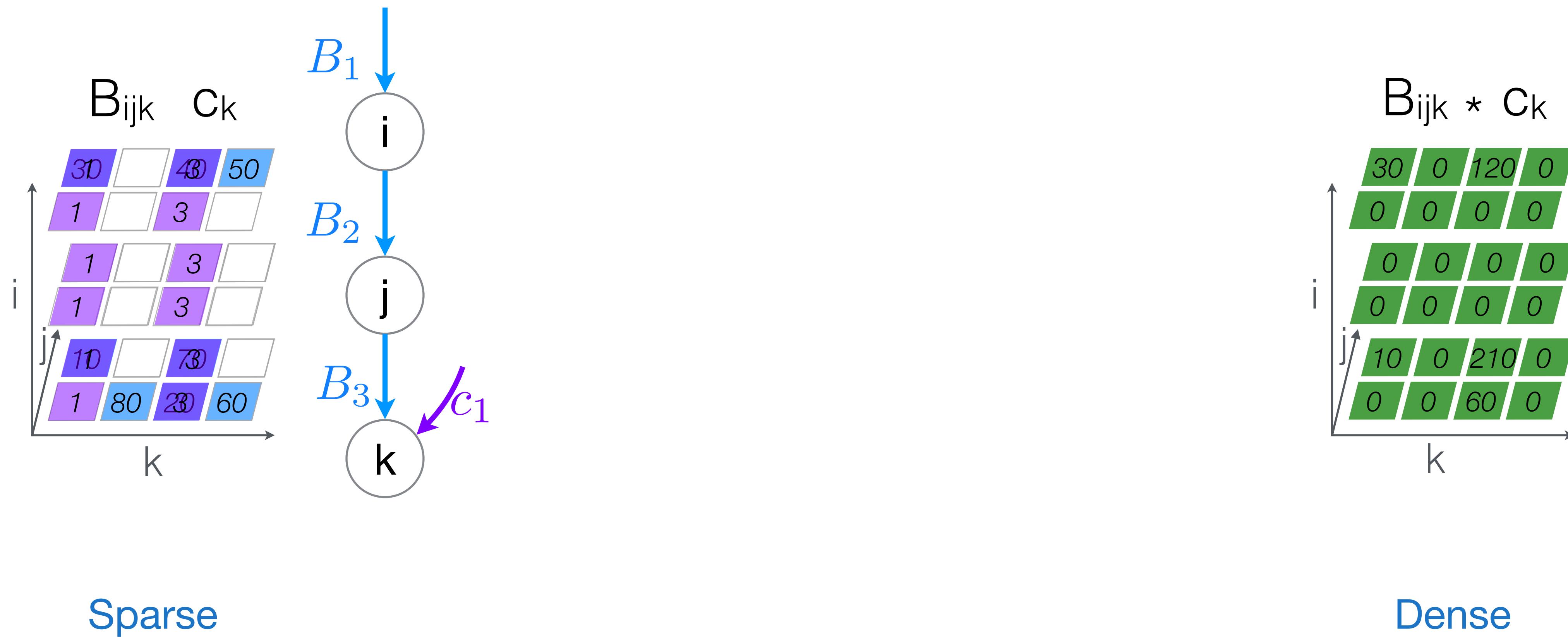
Sparse iteration spaces and Iteration Graphs

$$A_{ij} = \sum_k B_{ijk} * C_k$$



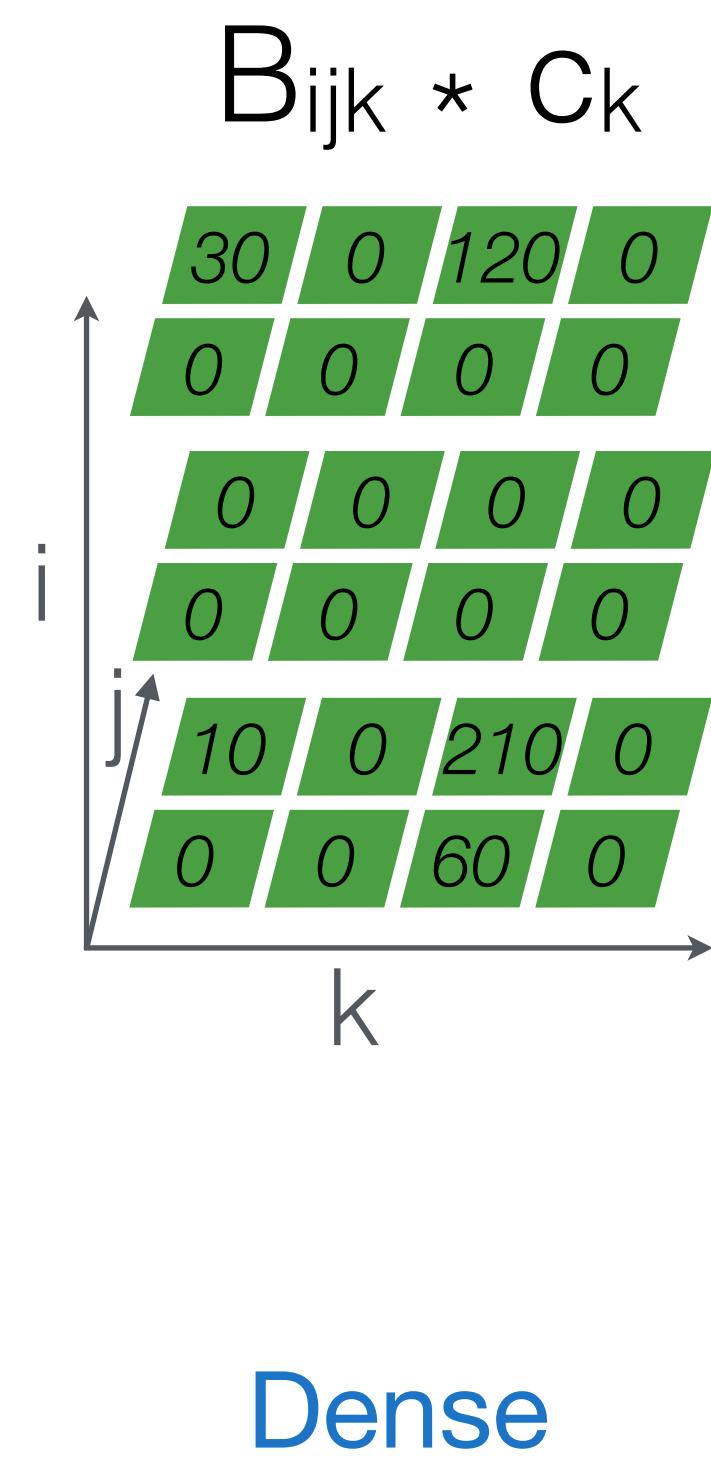
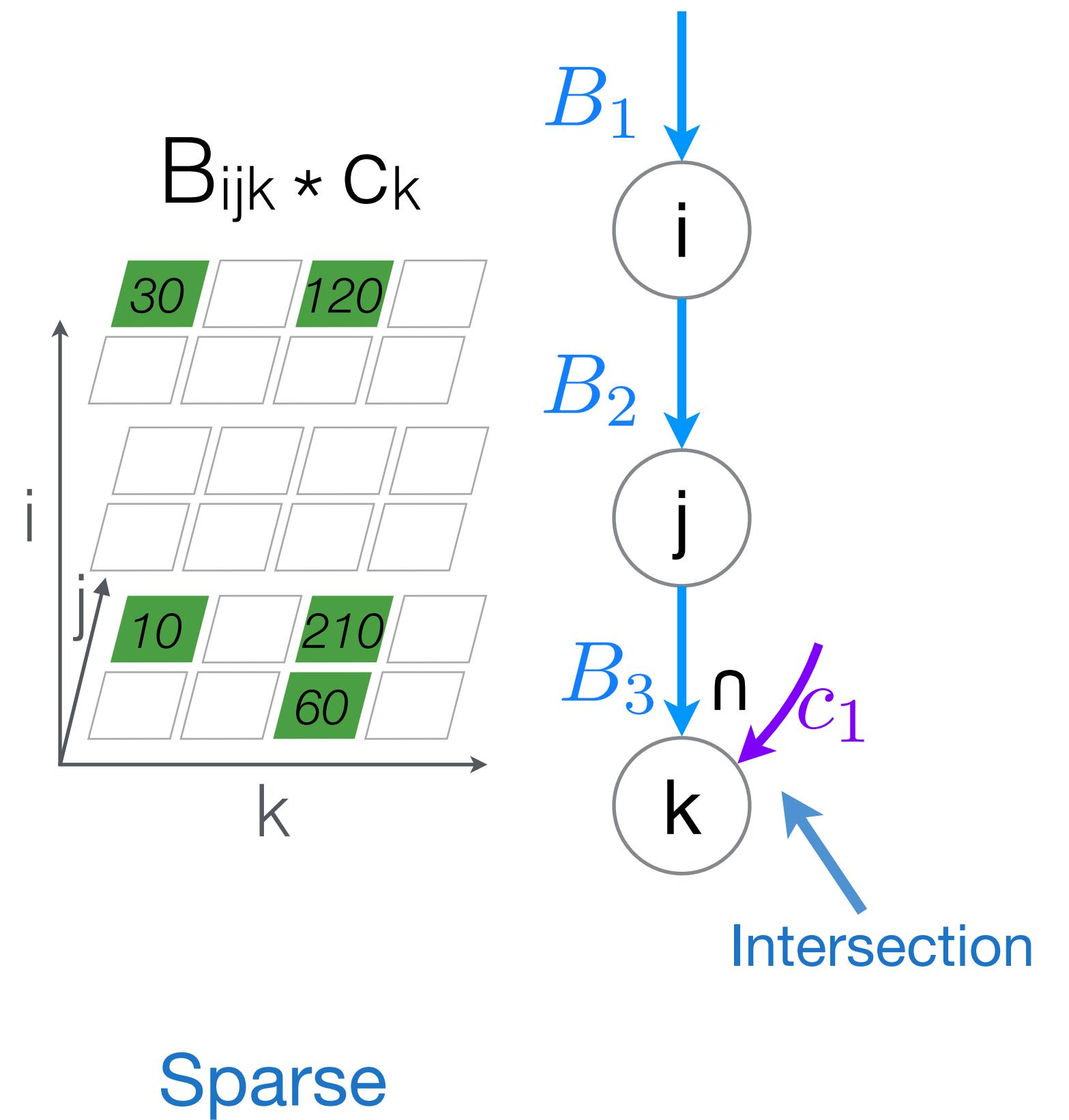
Sparse iteration spaces and Iteration Graphs

$$A_{ij} = \sum_k B_{ijk} * C_k$$

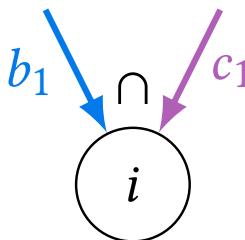


Sparse iteration spaces and Iteration Graphs

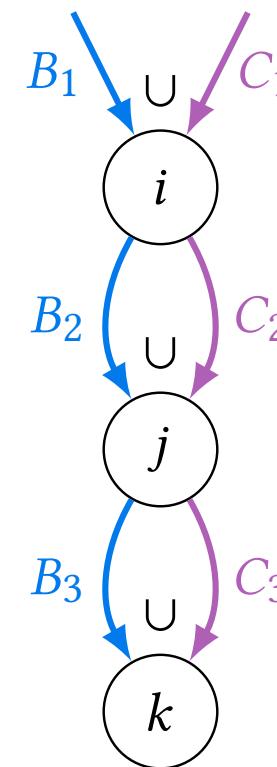
$$A_{ij} = \sum_k B_{ijk} * C_k$$



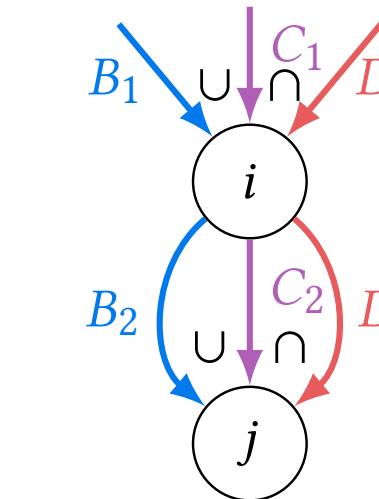
Iteration graph examples



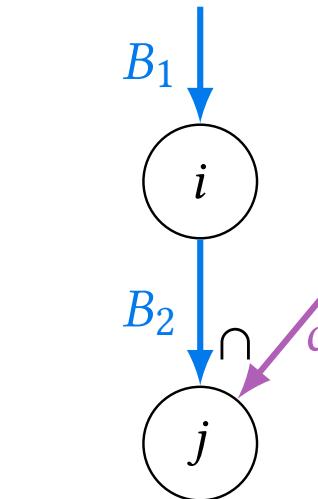
$$\frac{\forall_i \ b_i \cap c_i}{i \in b_1 \cap c_1}$$



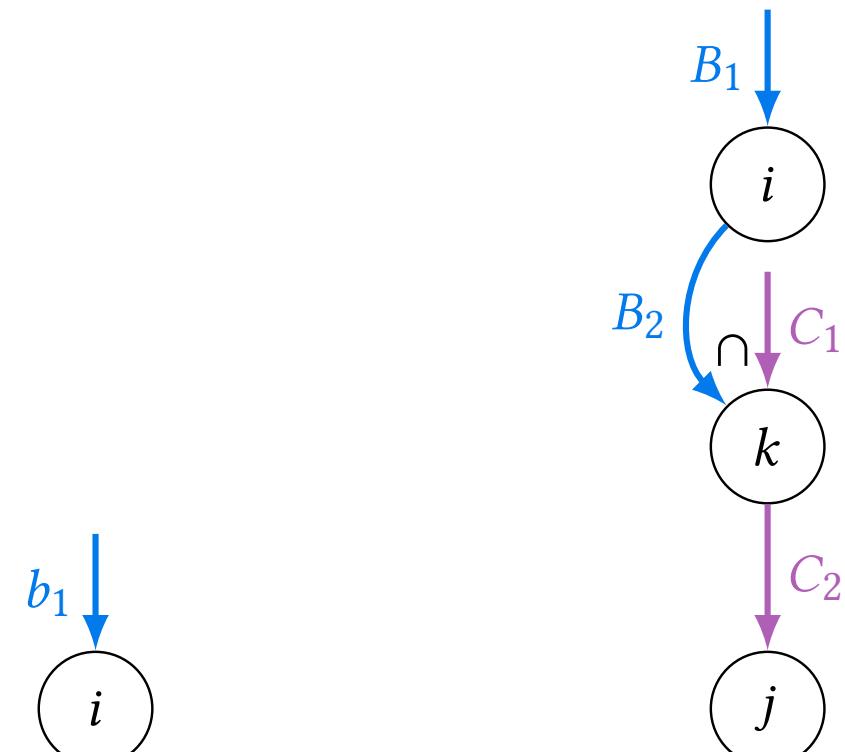
$$\frac{\forall_i \forall_j \forall_k \ B_{ijk} \cup C_{ijk}}{i \in B_1 \cup C_1 \\ j \in B_2 \cup C_2 \\ k \in B_3 \cup C_3}$$



$$\frac{\forall_i \forall_j (B_{ij} \cup C_{ij}) \cap D_{ij}}{i \in (B_1 \cup C_1) \cap D_1 \\ j \in (B_2 \cup C_2) \cap D_2}$$

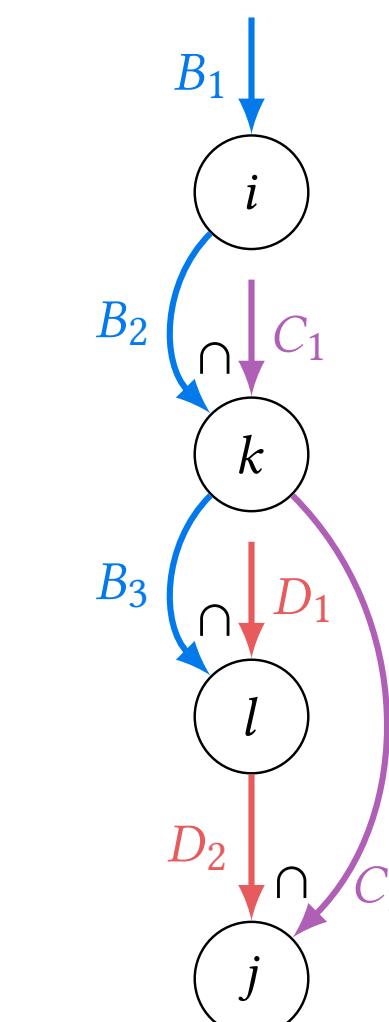


$$\frac{\forall_i \forall_j B_{ij} \cap c_j}{i \in B_1 \cap \mathbb{U}_i \\ j \in B_2 \cap c_1}$$

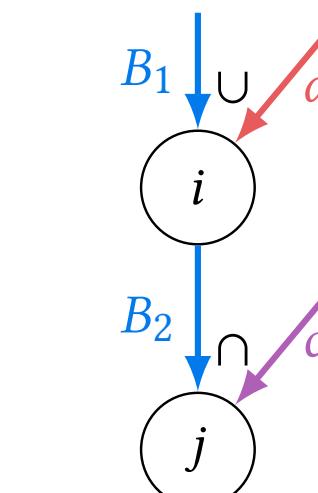


$$\frac{\forall_i \alpha \cup b_i}{i \in \mathbb{U}_i \cap b_1}$$

$$\frac{\forall_i \forall_k \forall_j B_{ik} \cap C_{kj}}{i \in B_1 \cap \mathbb{U}_i \\ k \in B_2 \cap C_1 \\ j \in \mathbb{U}_j \cap C_2}$$



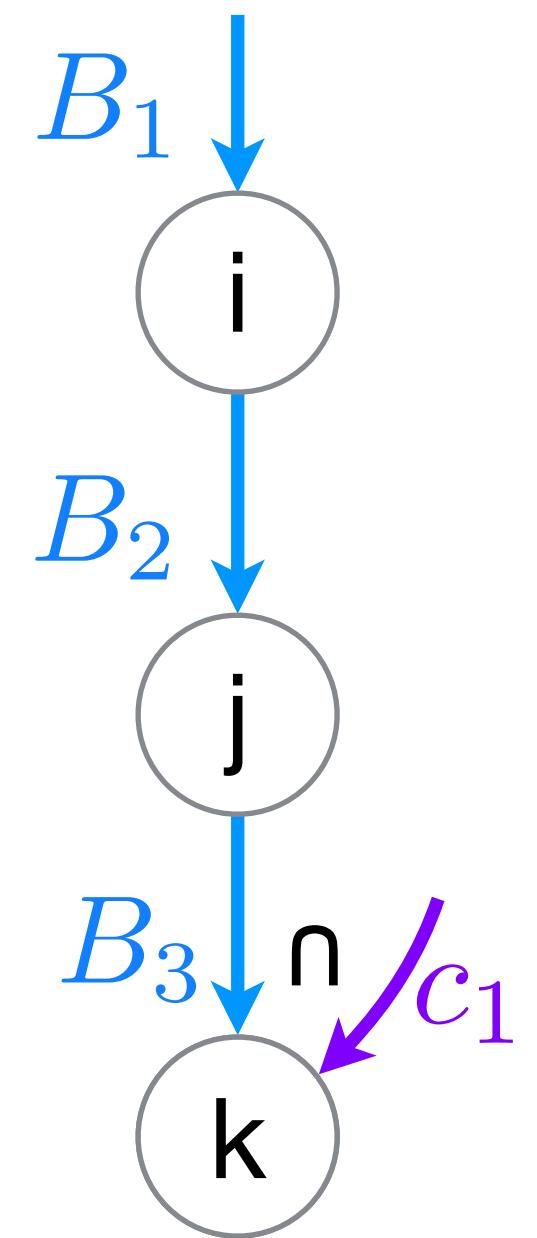
$$\frac{\forall_i \forall_k \forall_l \forall_j B_{ikl} \cap C_{kj} \cap D_{lj}}{i \in B_1 \cap \mathbb{U}_i \cap \mathbb{U}_i \\ k \in B_2 \cap C_1 \cap \mathbb{U}_k \\ l \in B_3 \cap \mathbb{U}_l \cap D_1 \\ j \in \mathbb{U}_j \cap C_2 \cap D_2}$$



$$\frac{\forall_i (\forall_j B_{ij} \cap c_j) \cup d_i}{i \in (B_1 \cap \mathbb{U}_i) \cup d_1 \\ j \in B_2 \cap c_1}$$

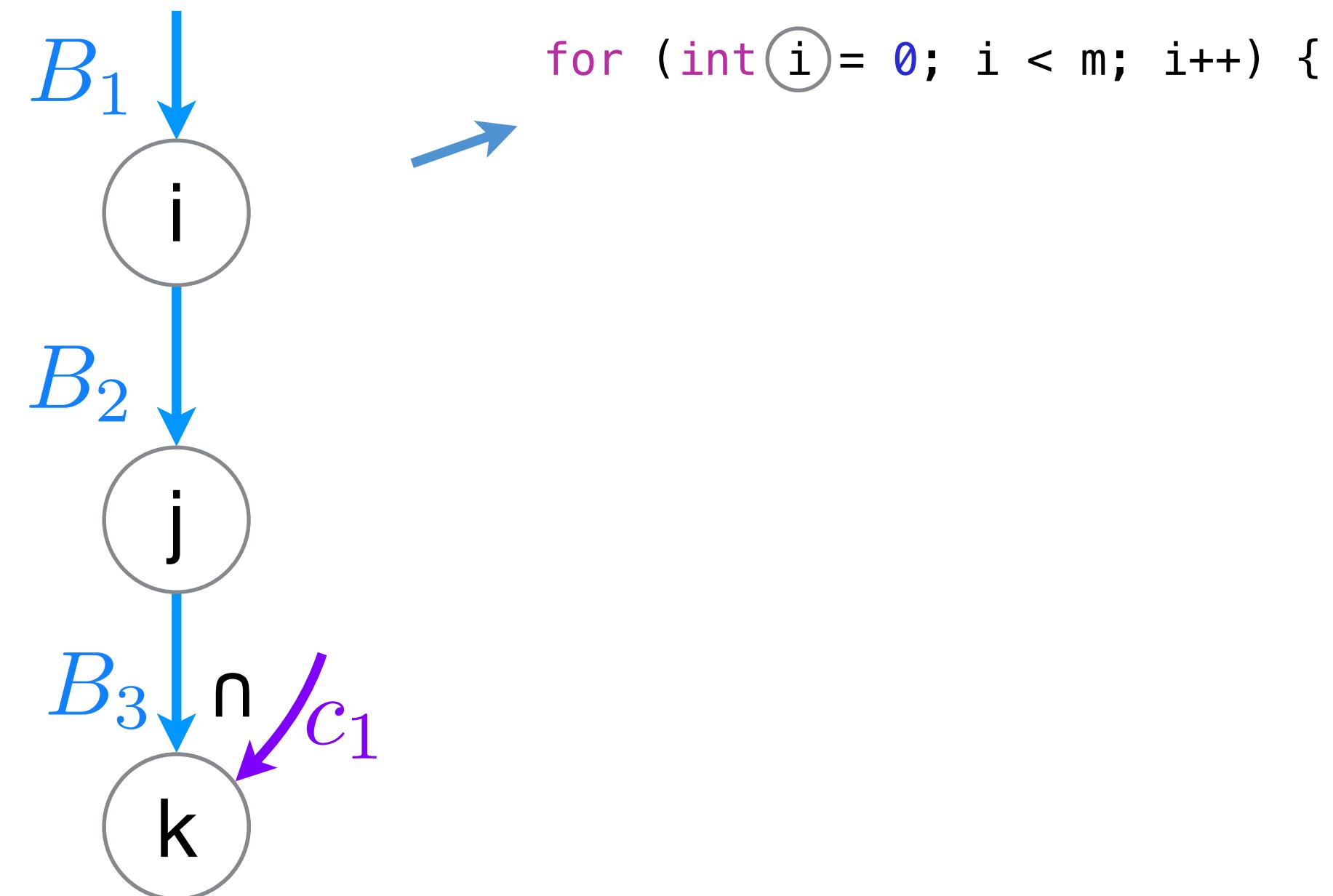
Iteration graphs are lowered to sparse code

$$A_{ij} = \sum_k B_{ijk} c_k$$



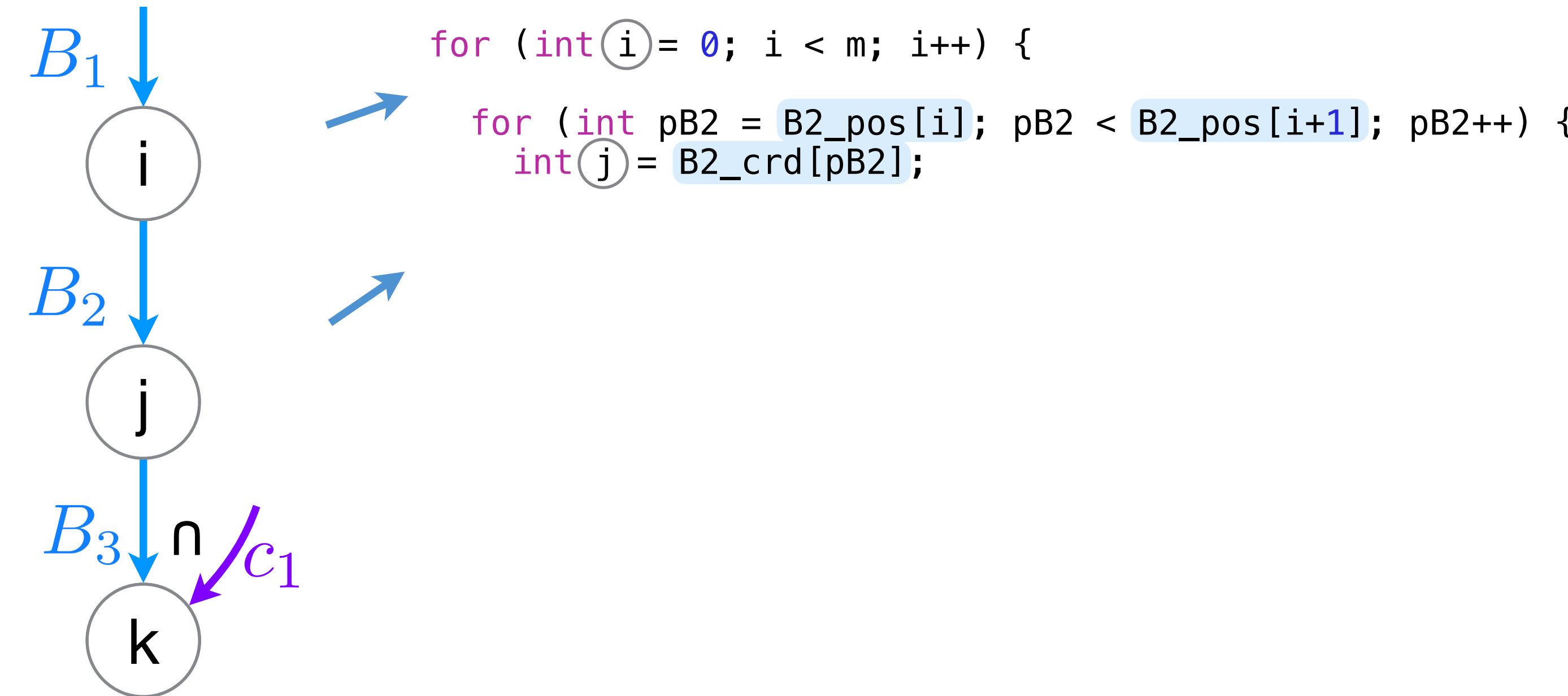
Iteration graphs are lowered to sparse code

$$A_{ij} = \sum_k B_{ijk} c_k$$



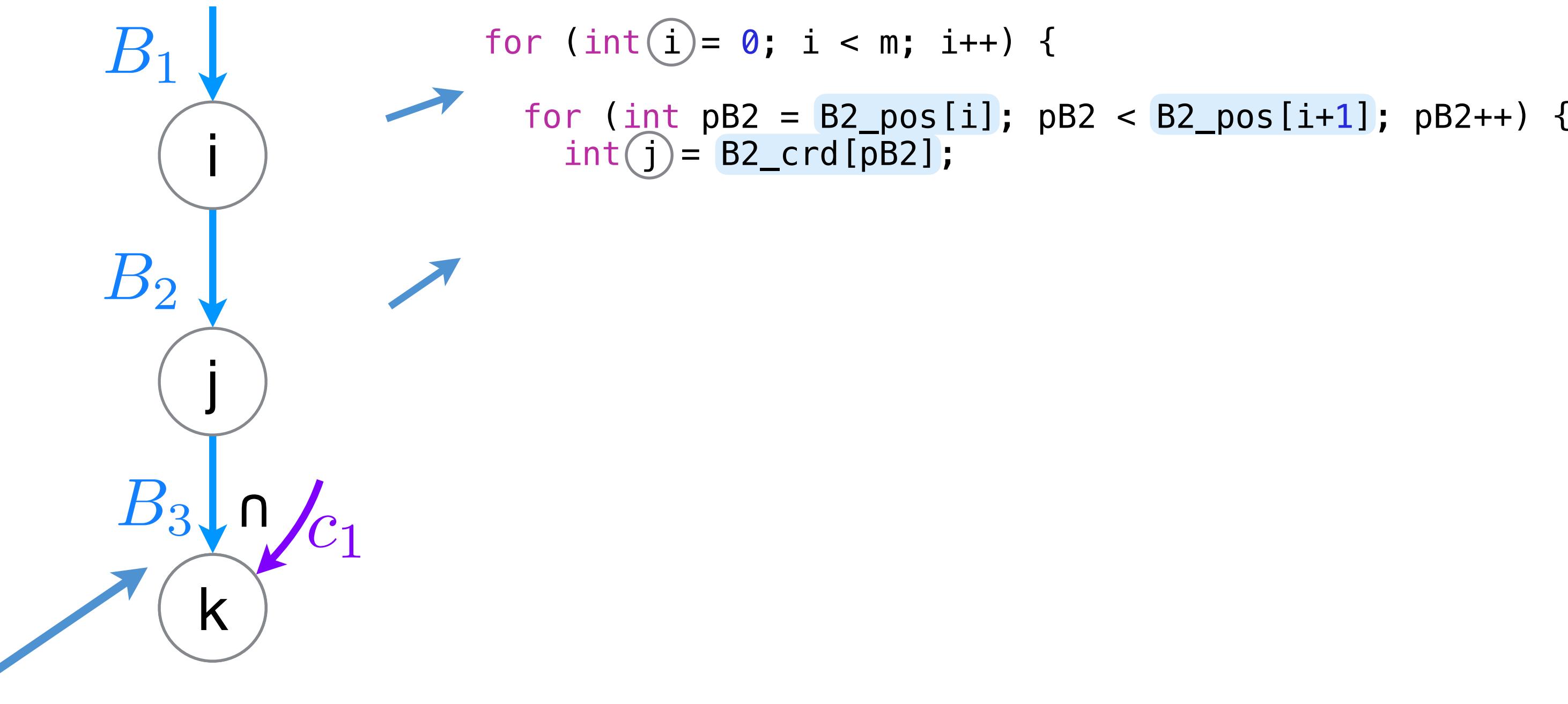
Iteration graphs are lowered to sparse code

$$A_{ij} = \sum_k B_{ijk} c_k$$



Iteration graphs are lowered to sparse code

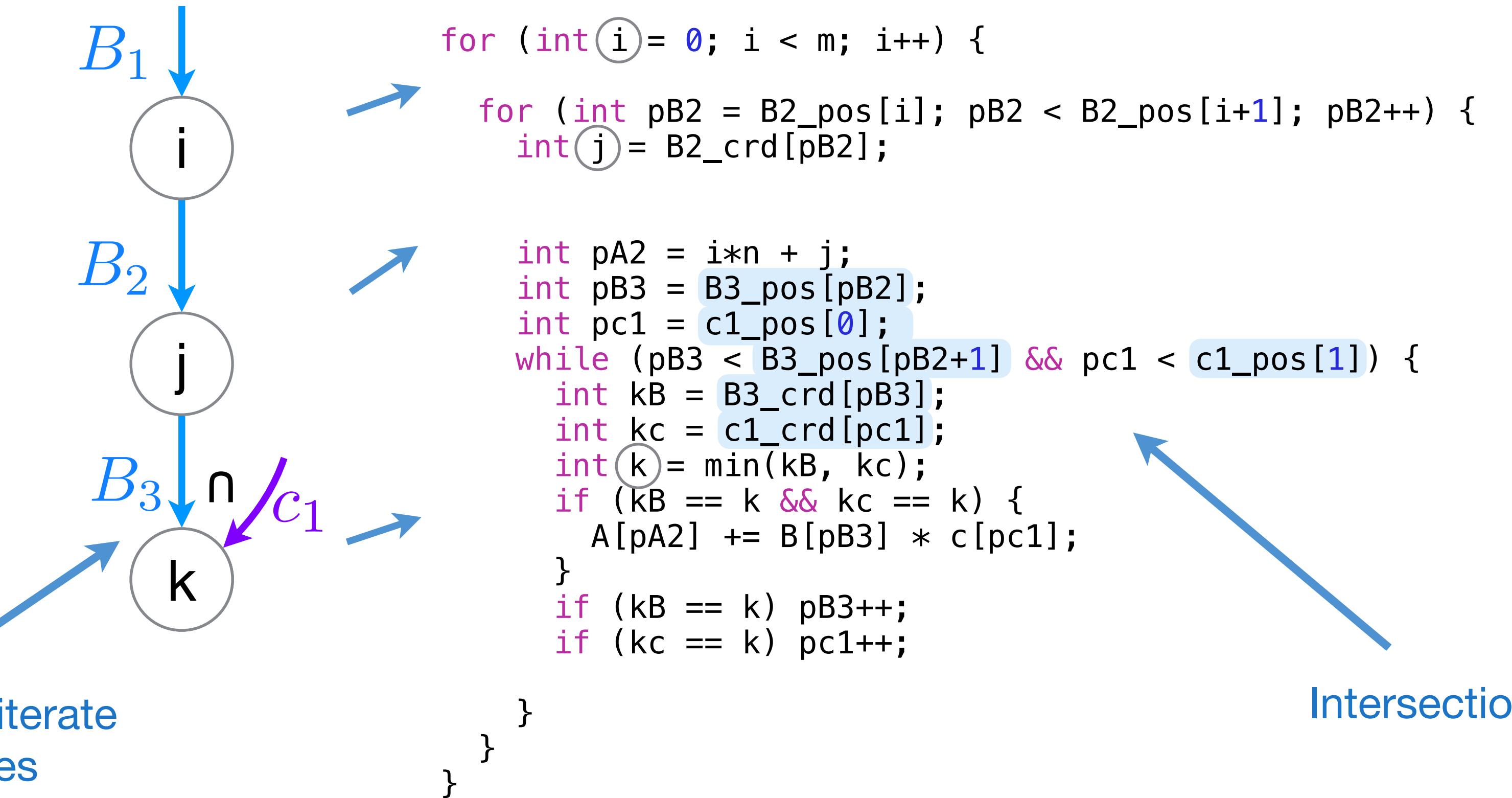
$$A_{ij} = \sum_k B_{ijk} c_k$$



Key operation is to coiterate
over data structures

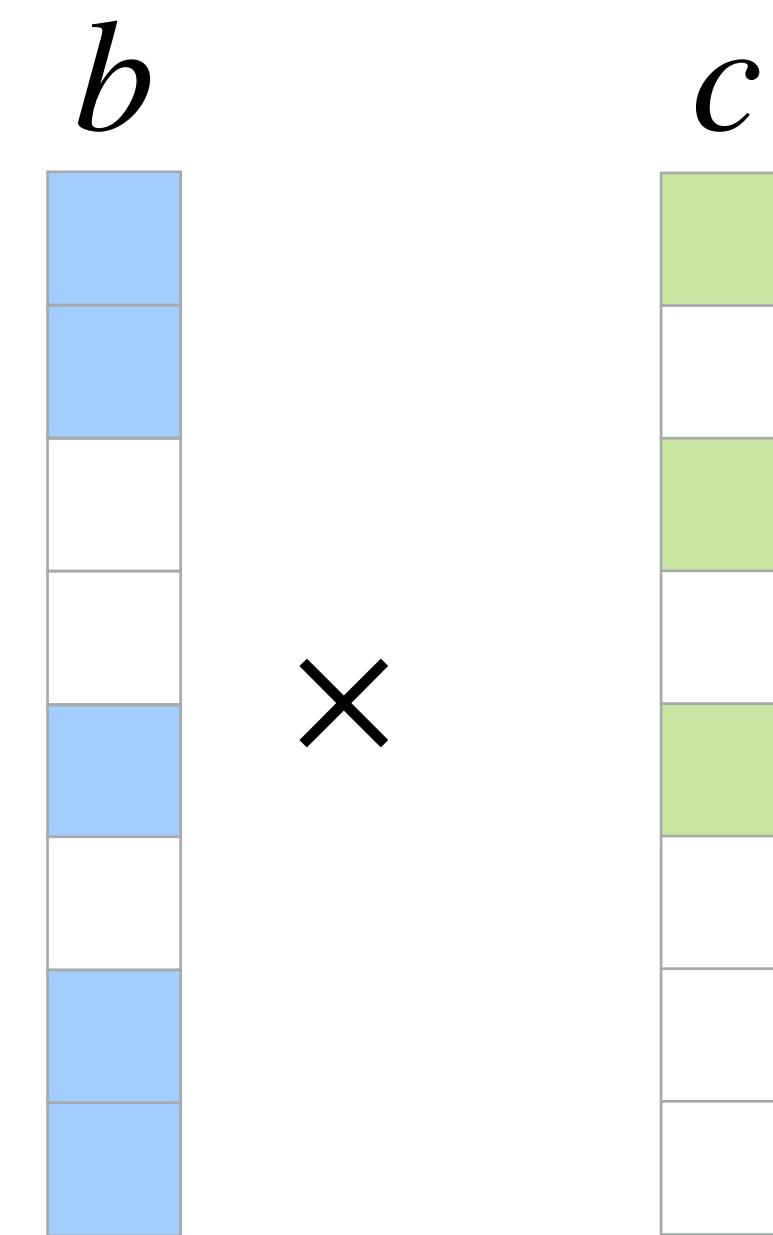
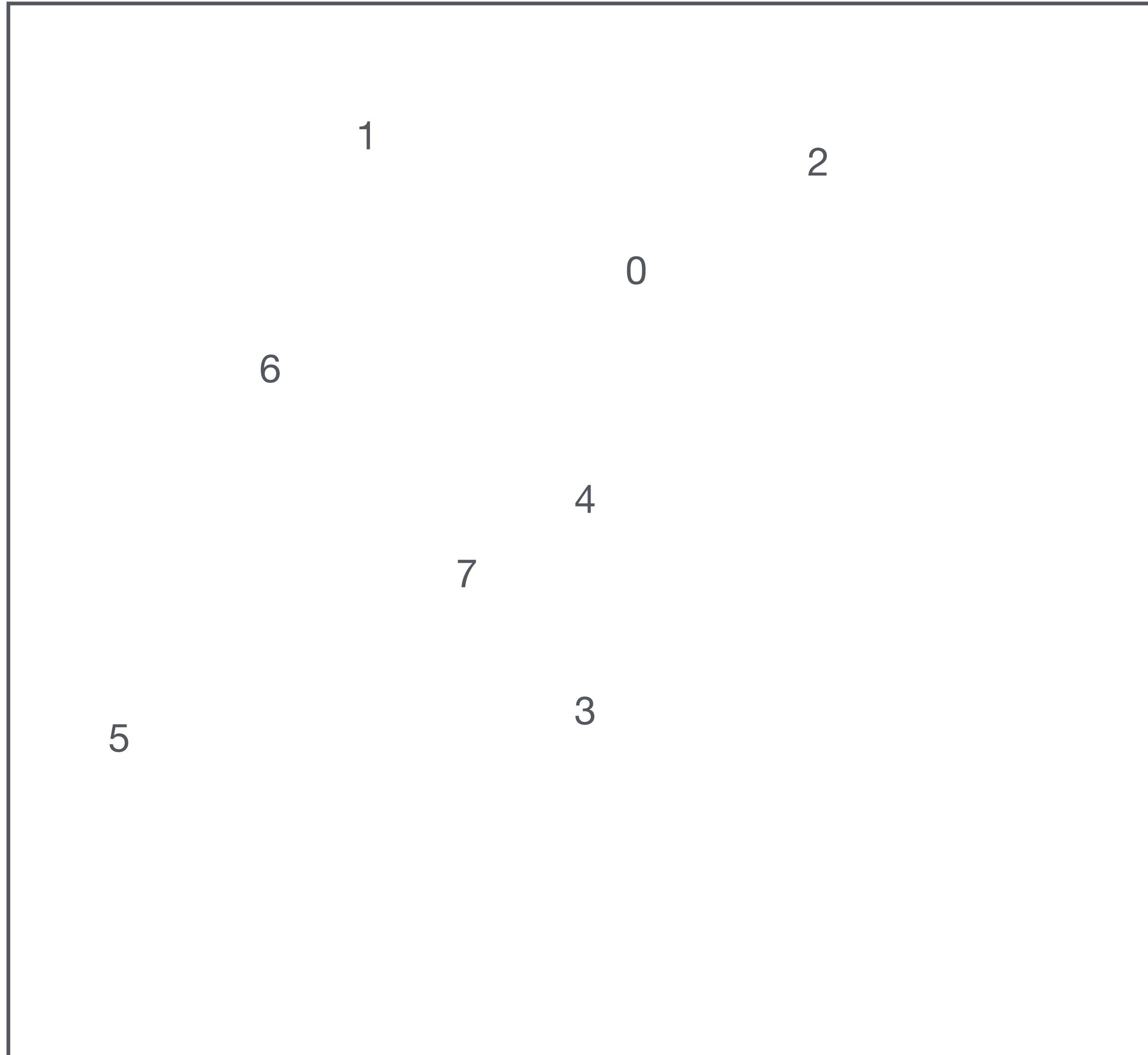
Iteration graphs are lowered to sparse code

$$A_{ij} = \sum_k B_{ijk} c_k$$



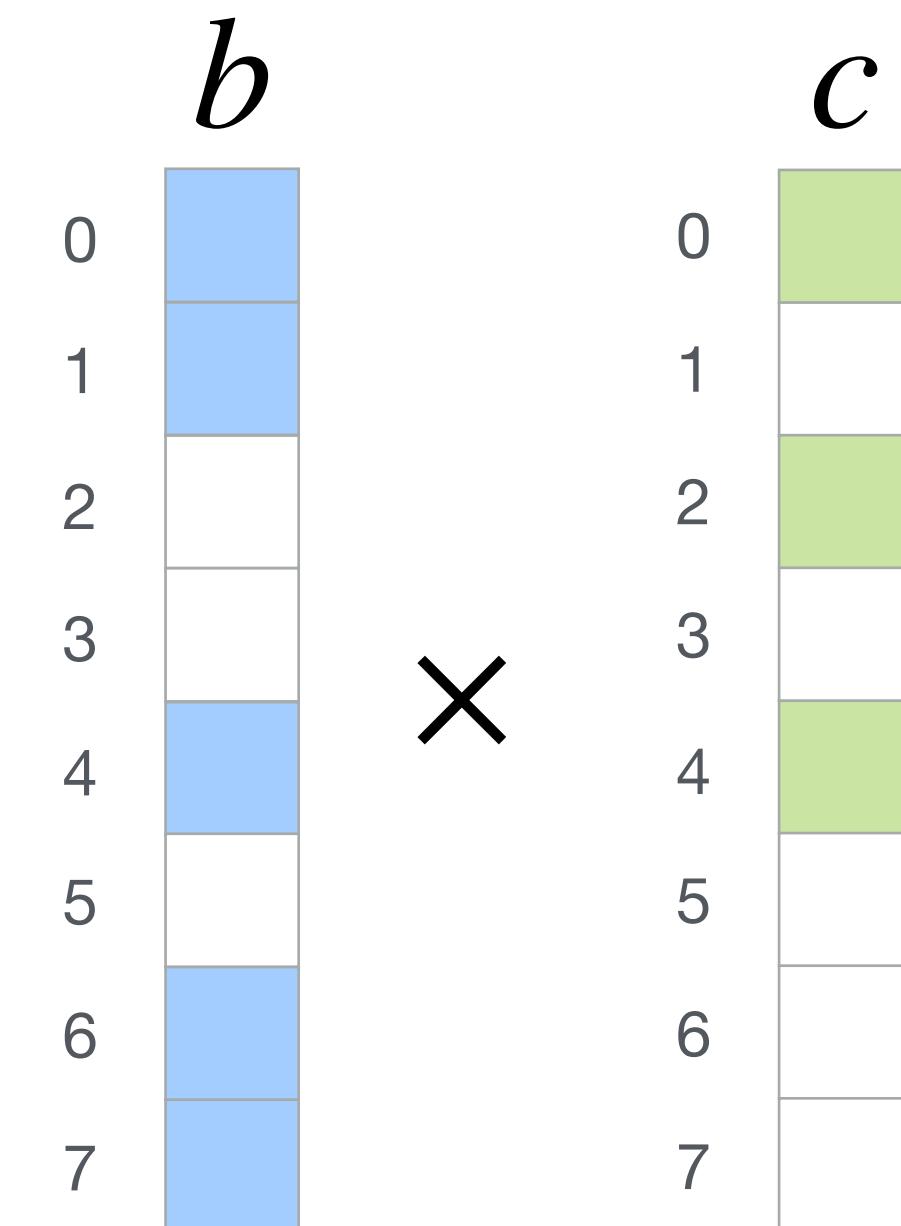
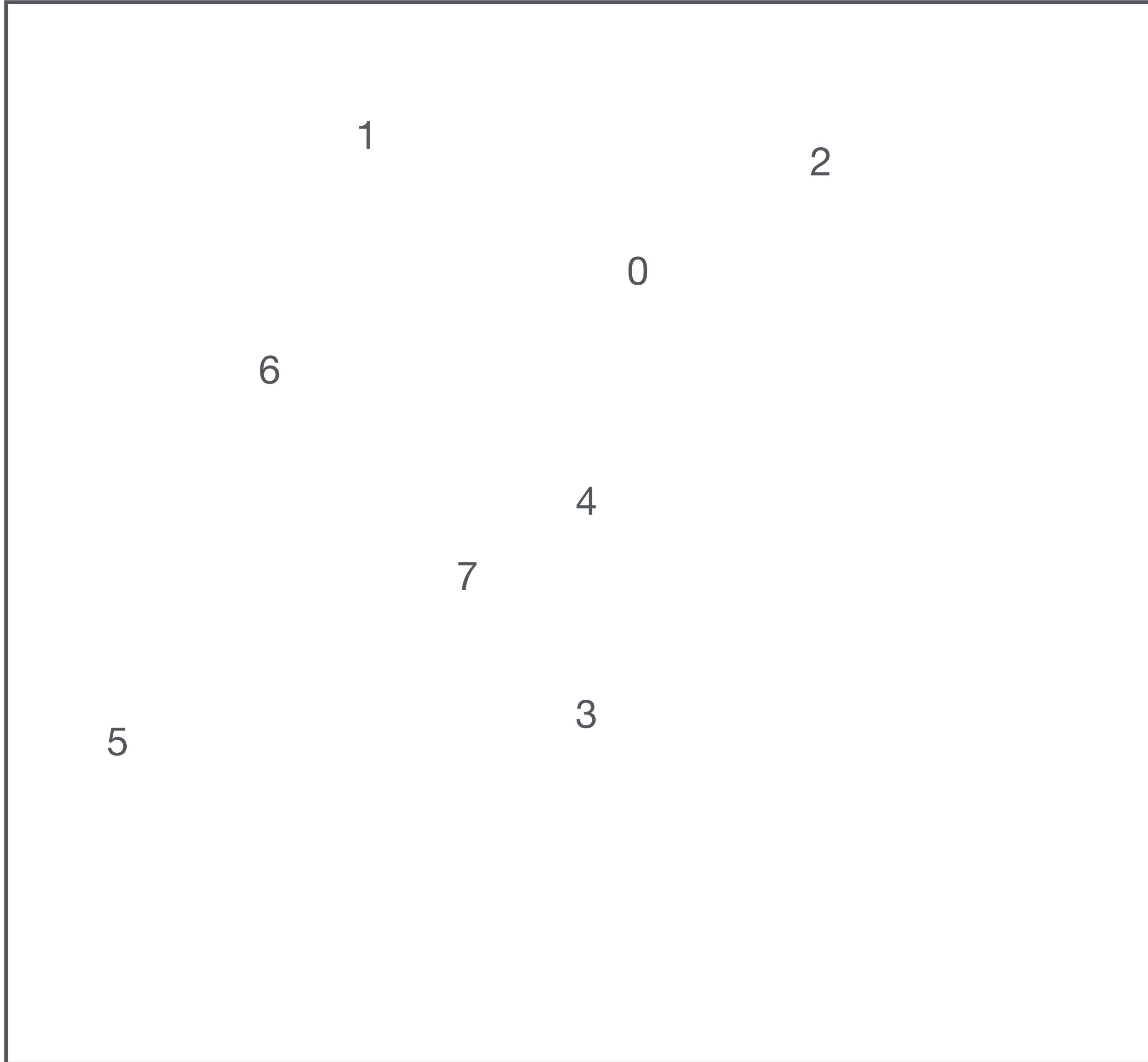
Data structure coiteration

Coordinate Space



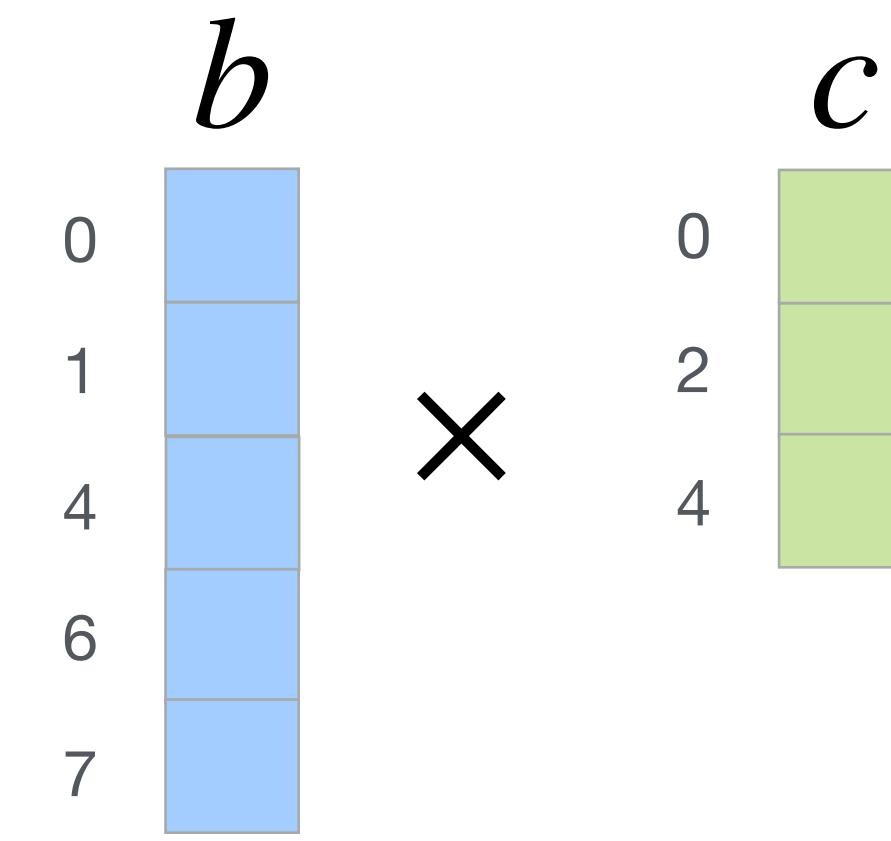
Data structure coiteration

Coordinate Space



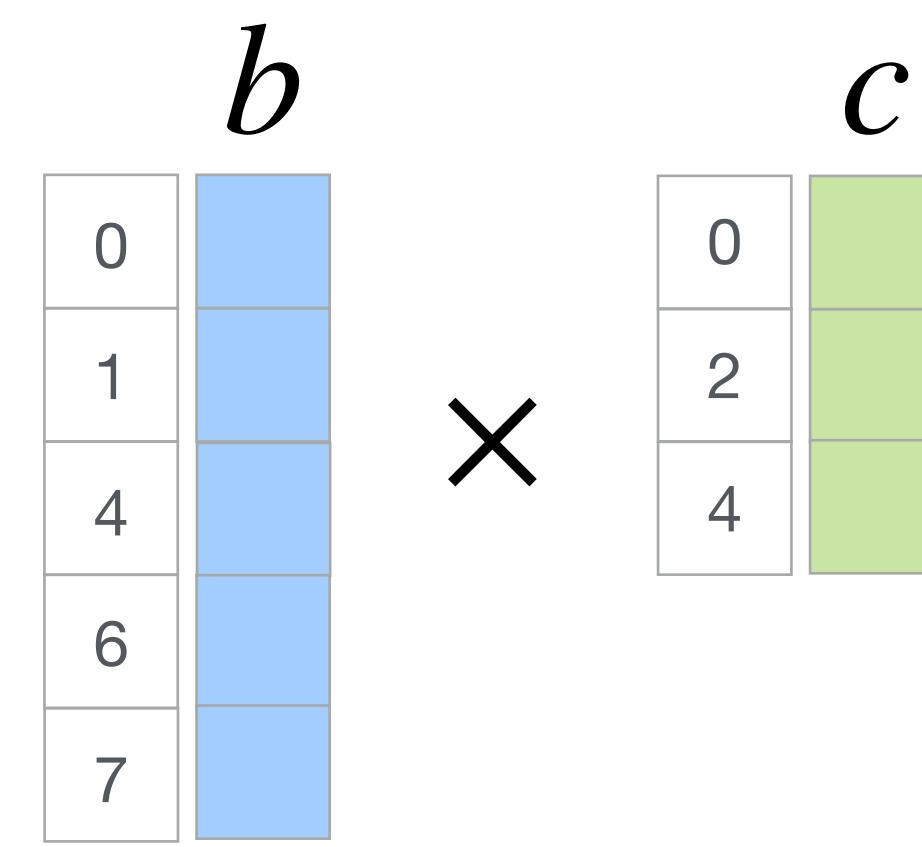
Data structure coiteration

Coordinate Space

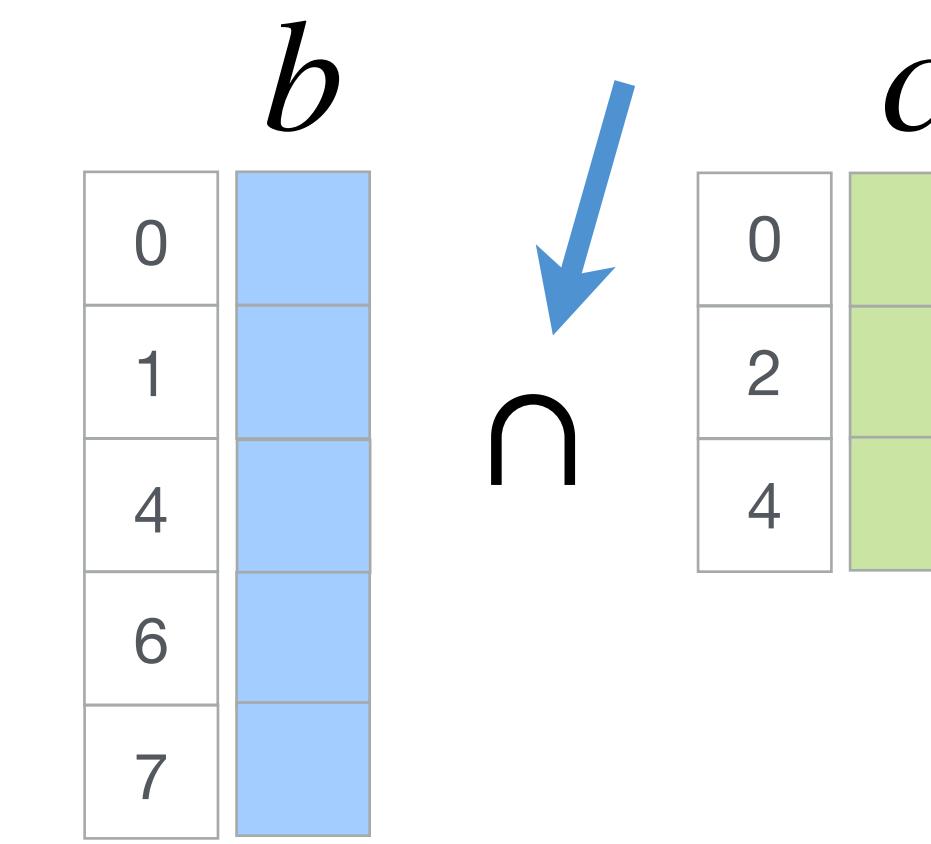
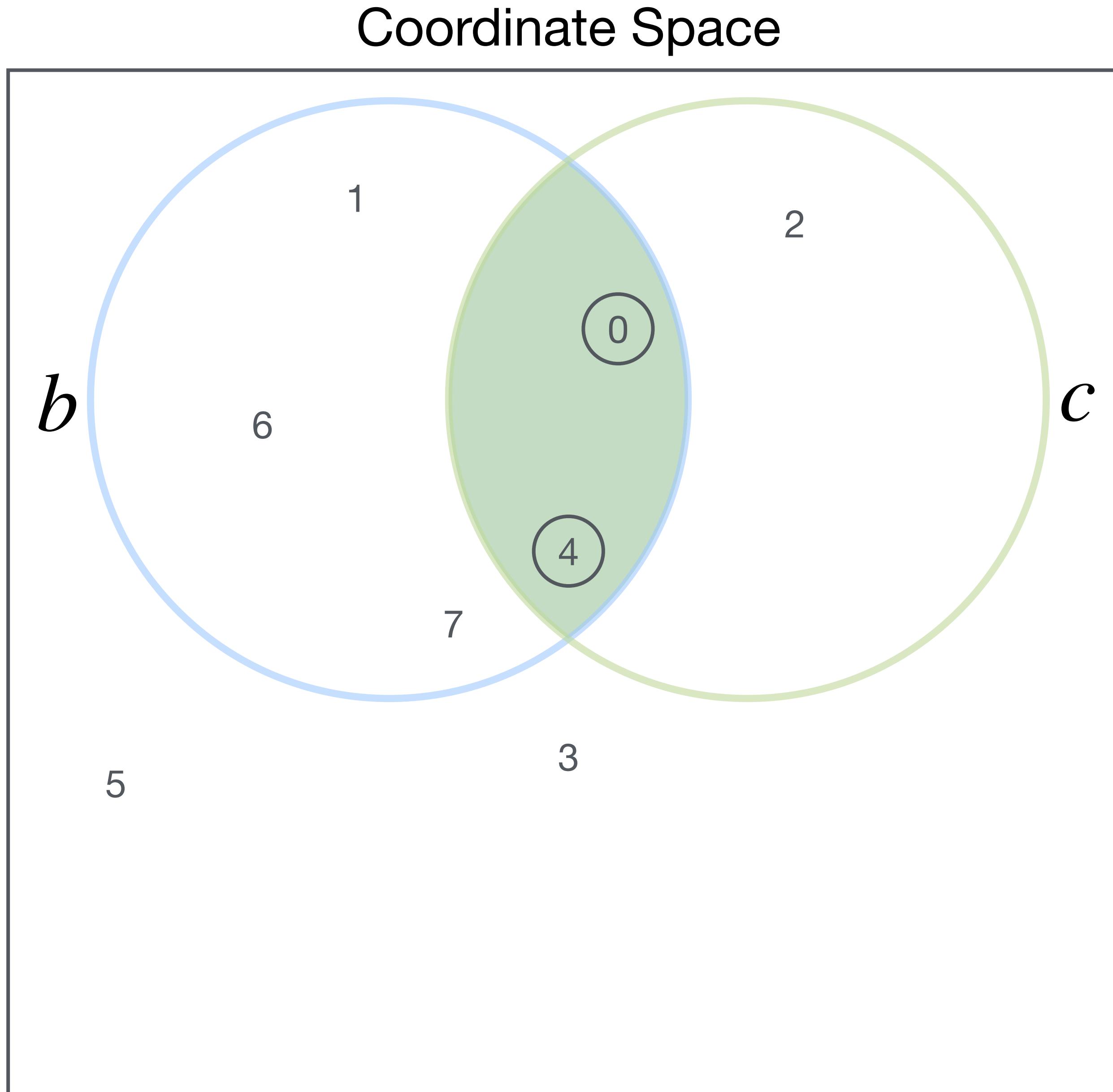


Data structure coiteration

Coordinate Space

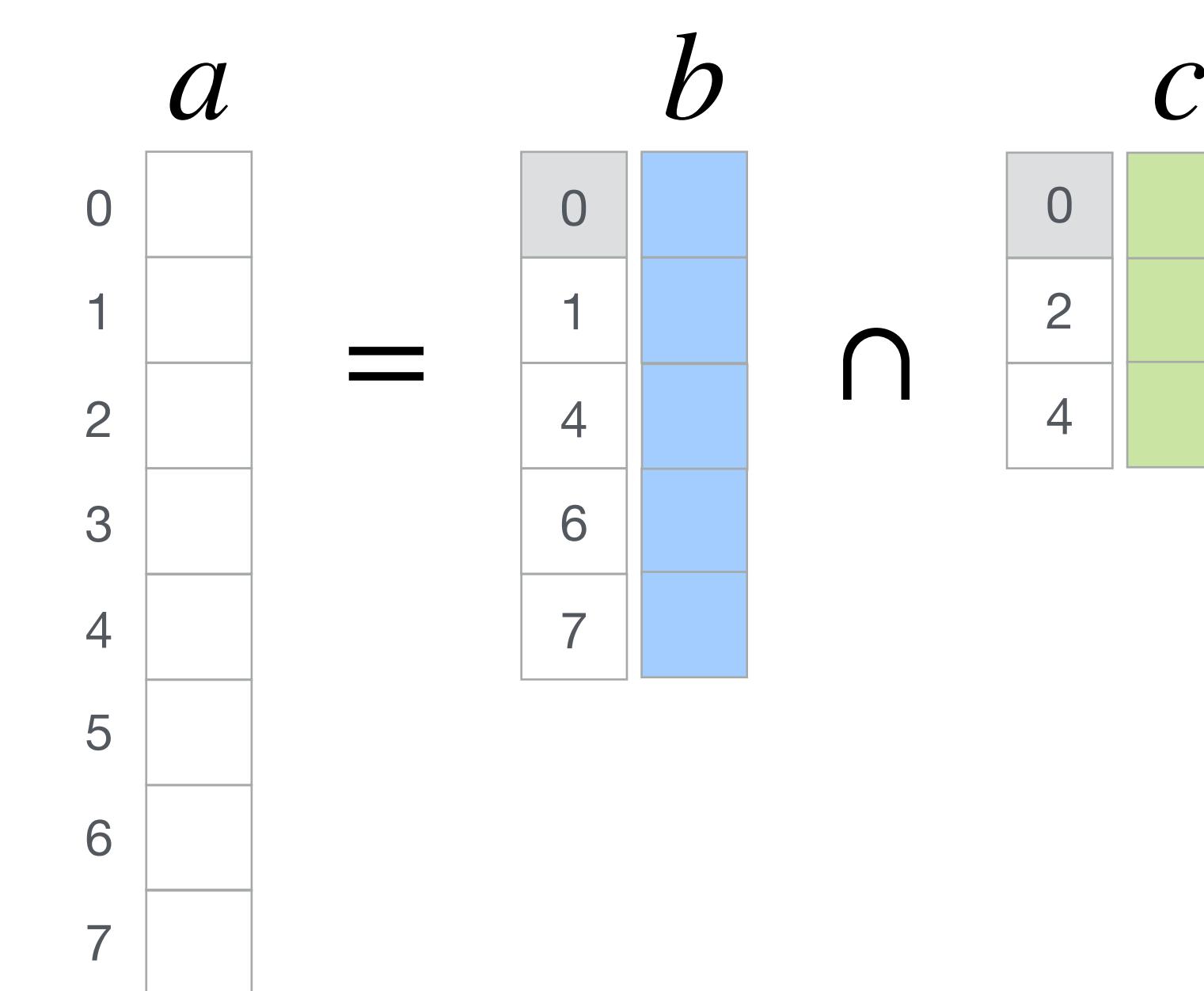
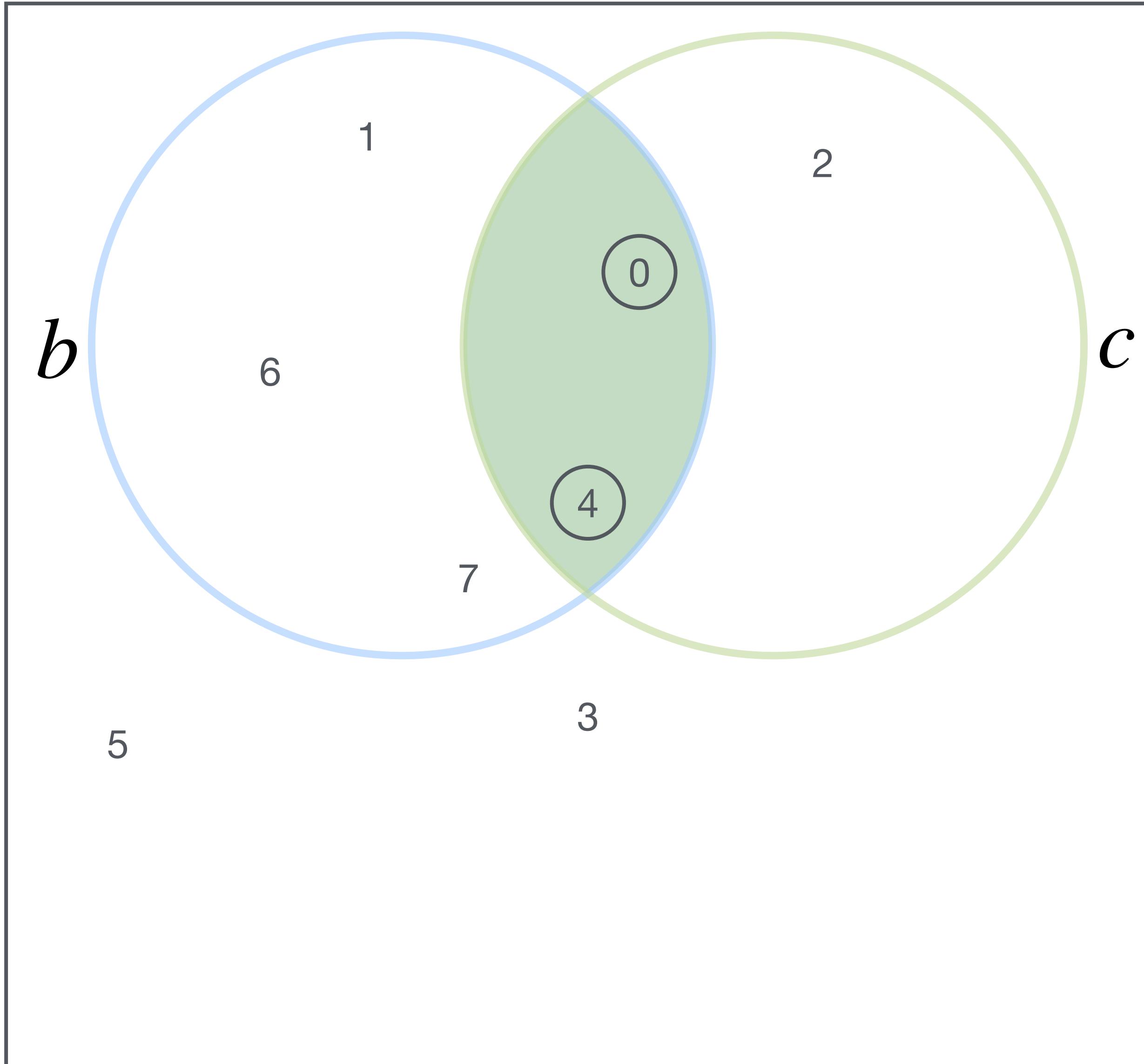


Data structure coiteration



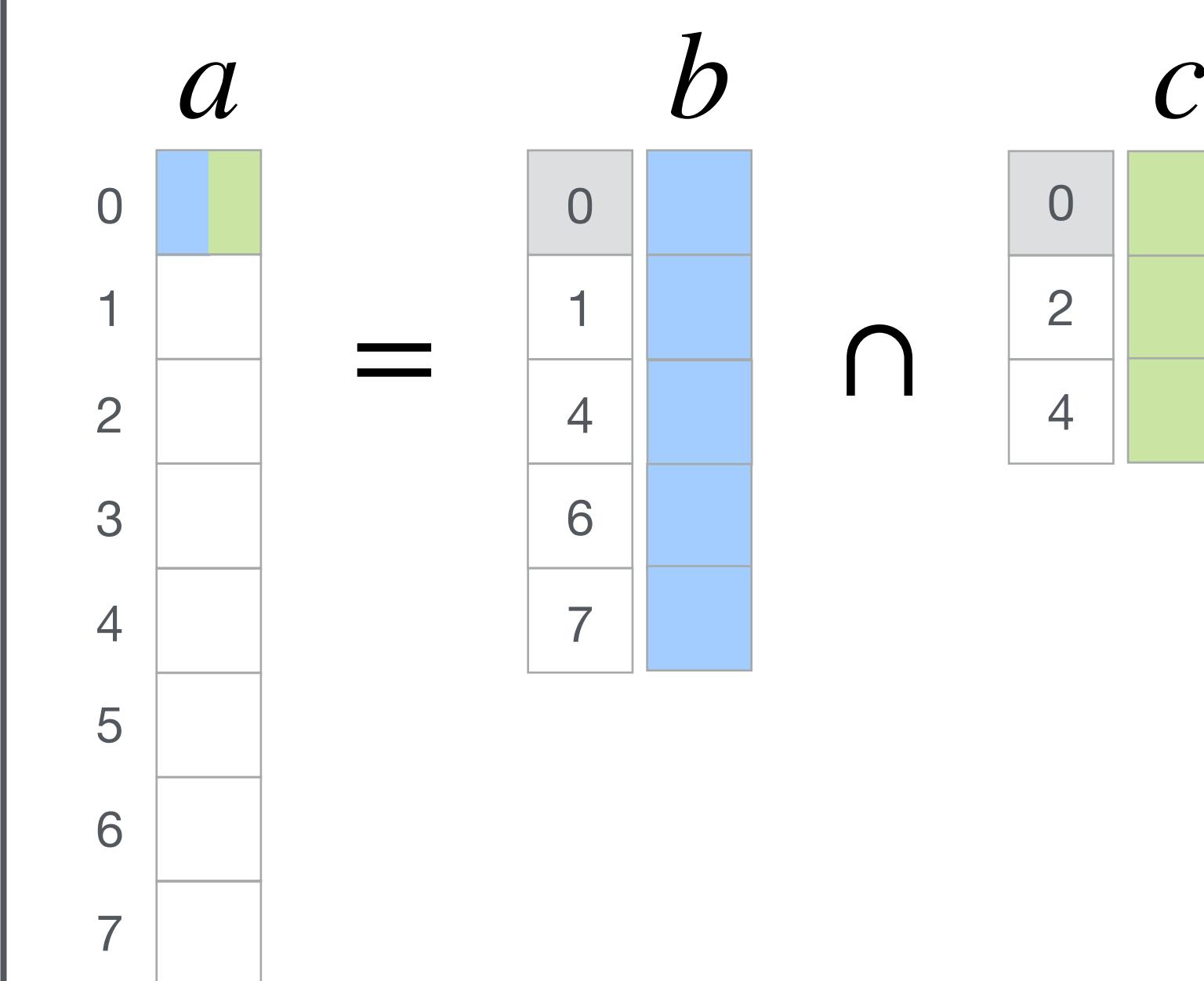
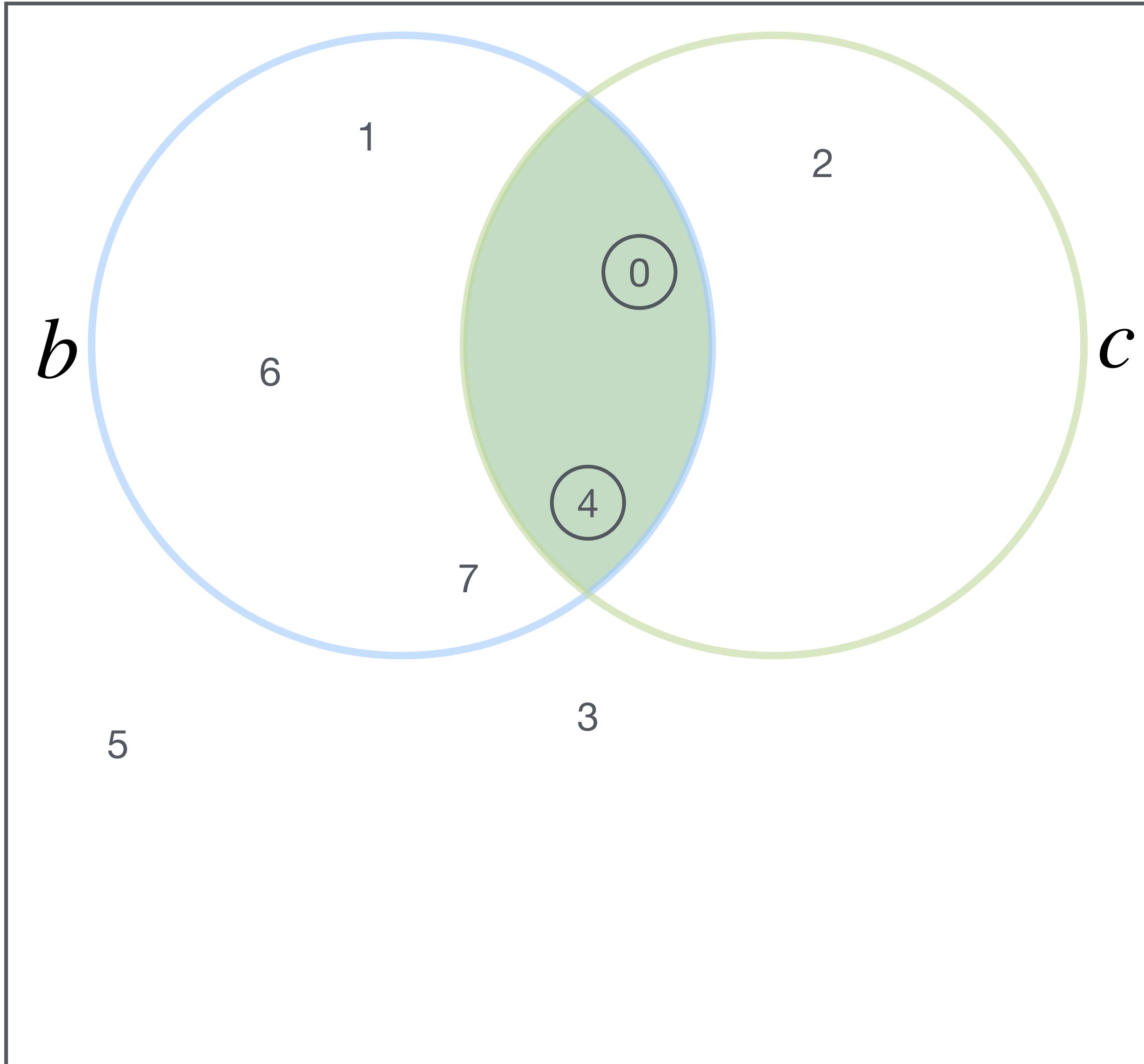
Data structure coiteration

Coordinate Space



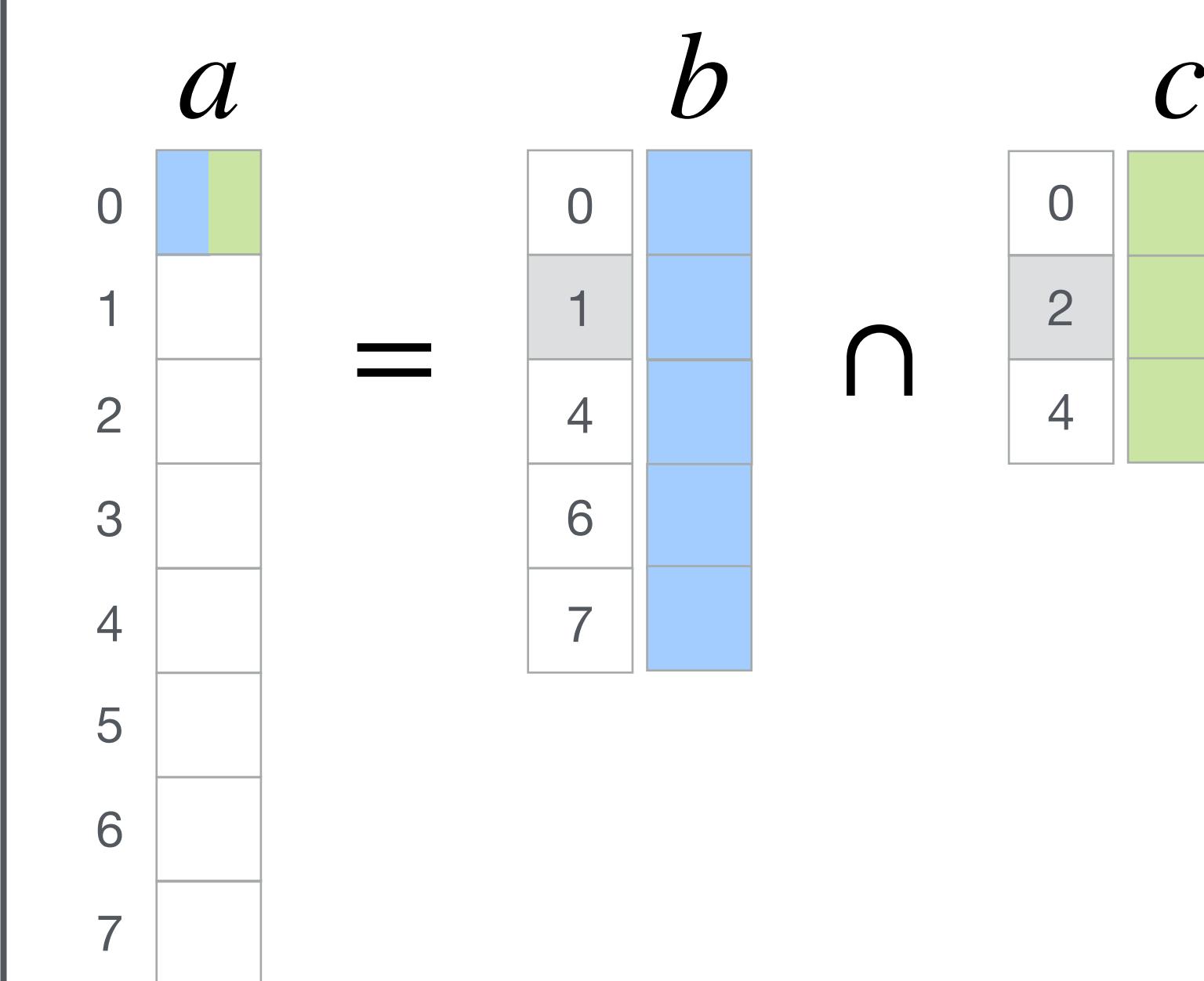
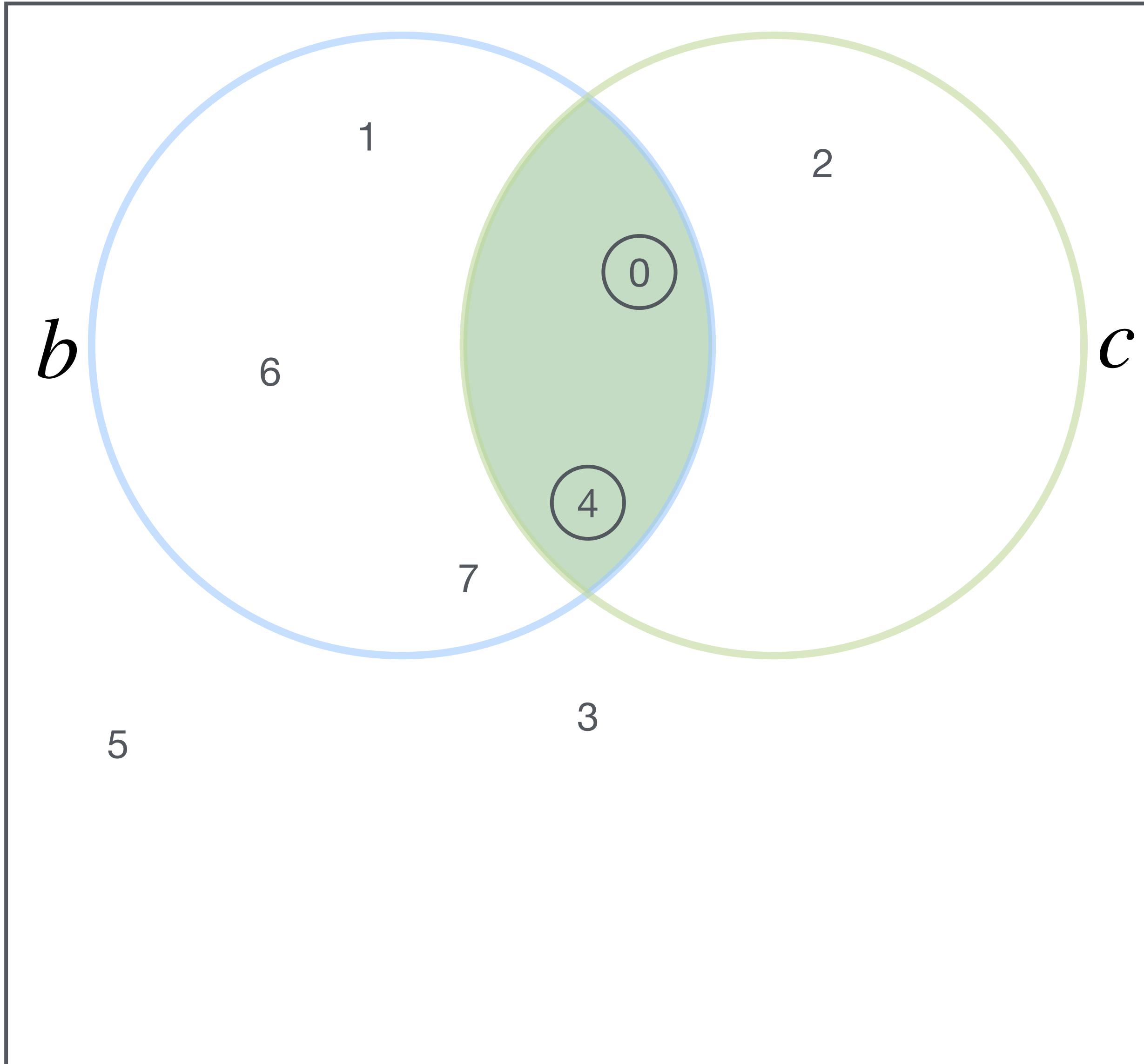
Data structure coiteration

Coordinate Space



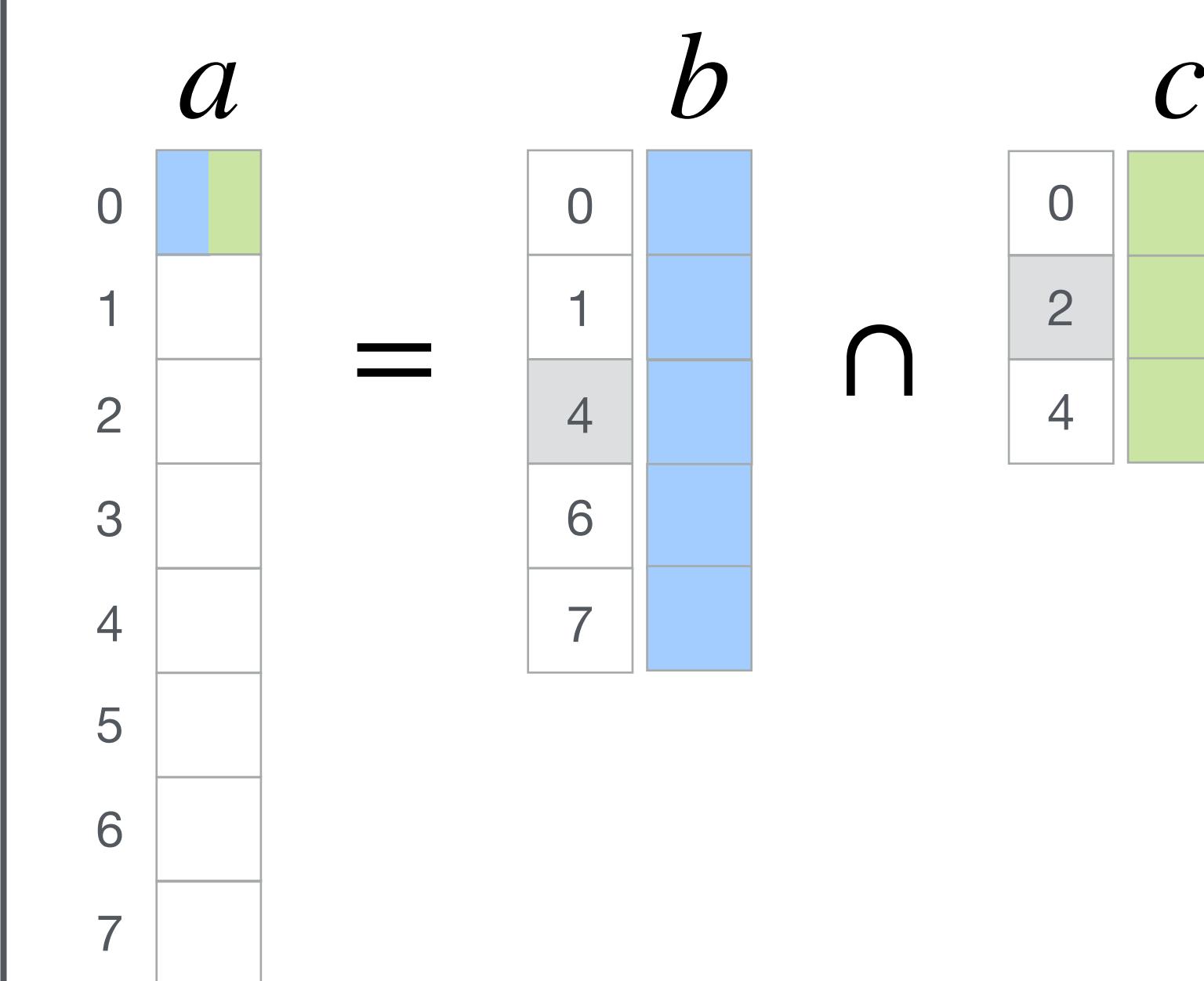
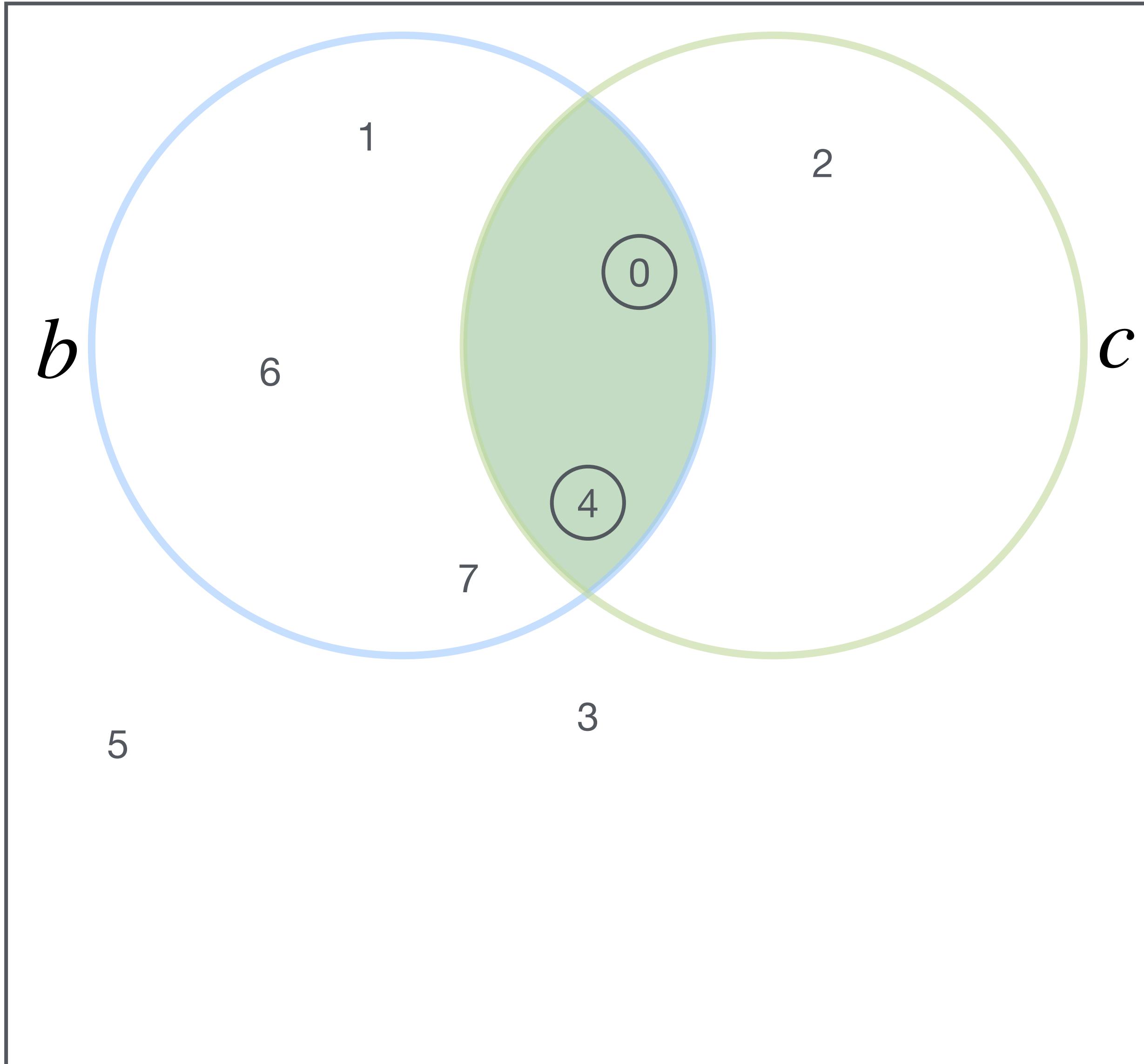
Data structure coiteration

Coordinate Space



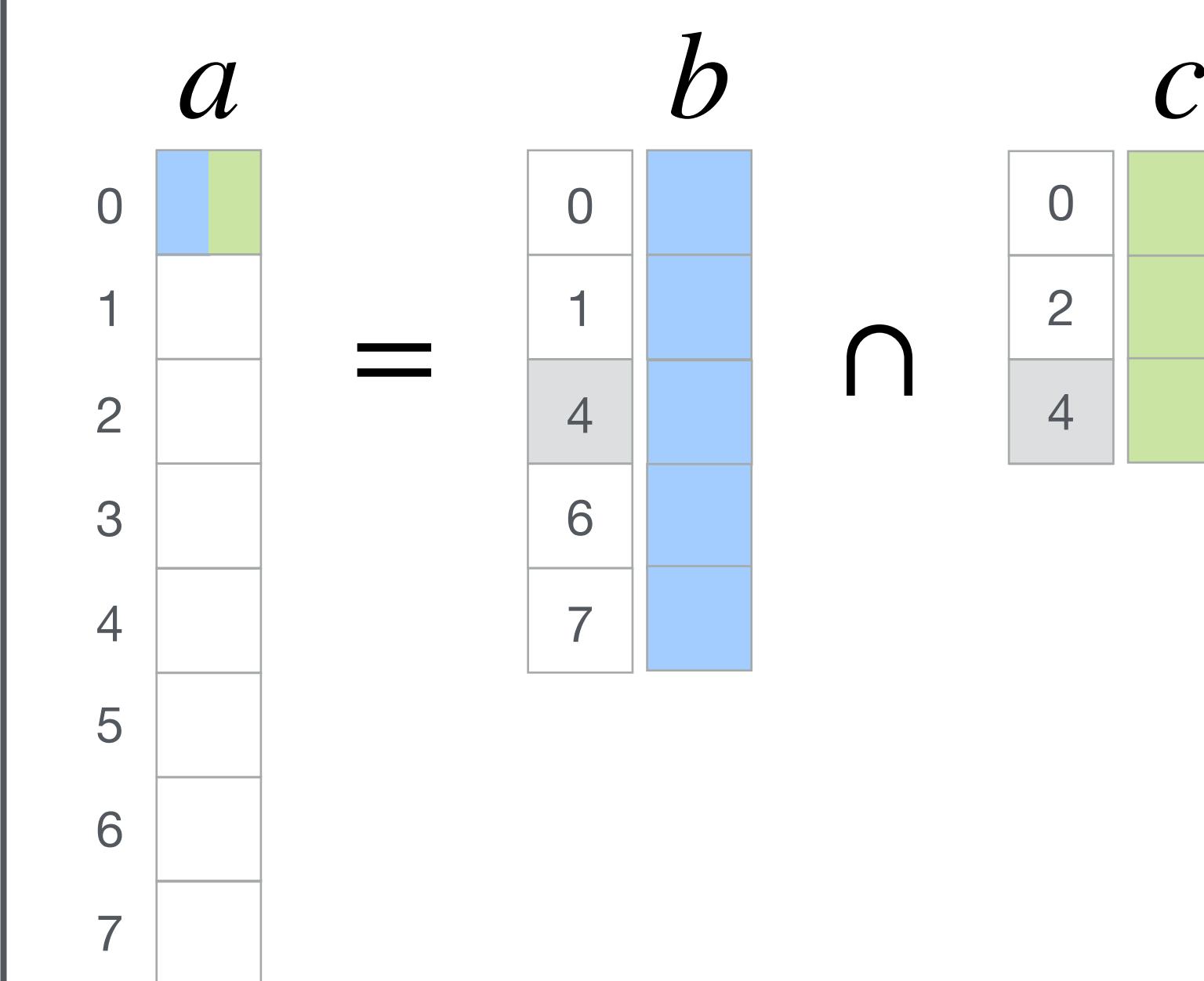
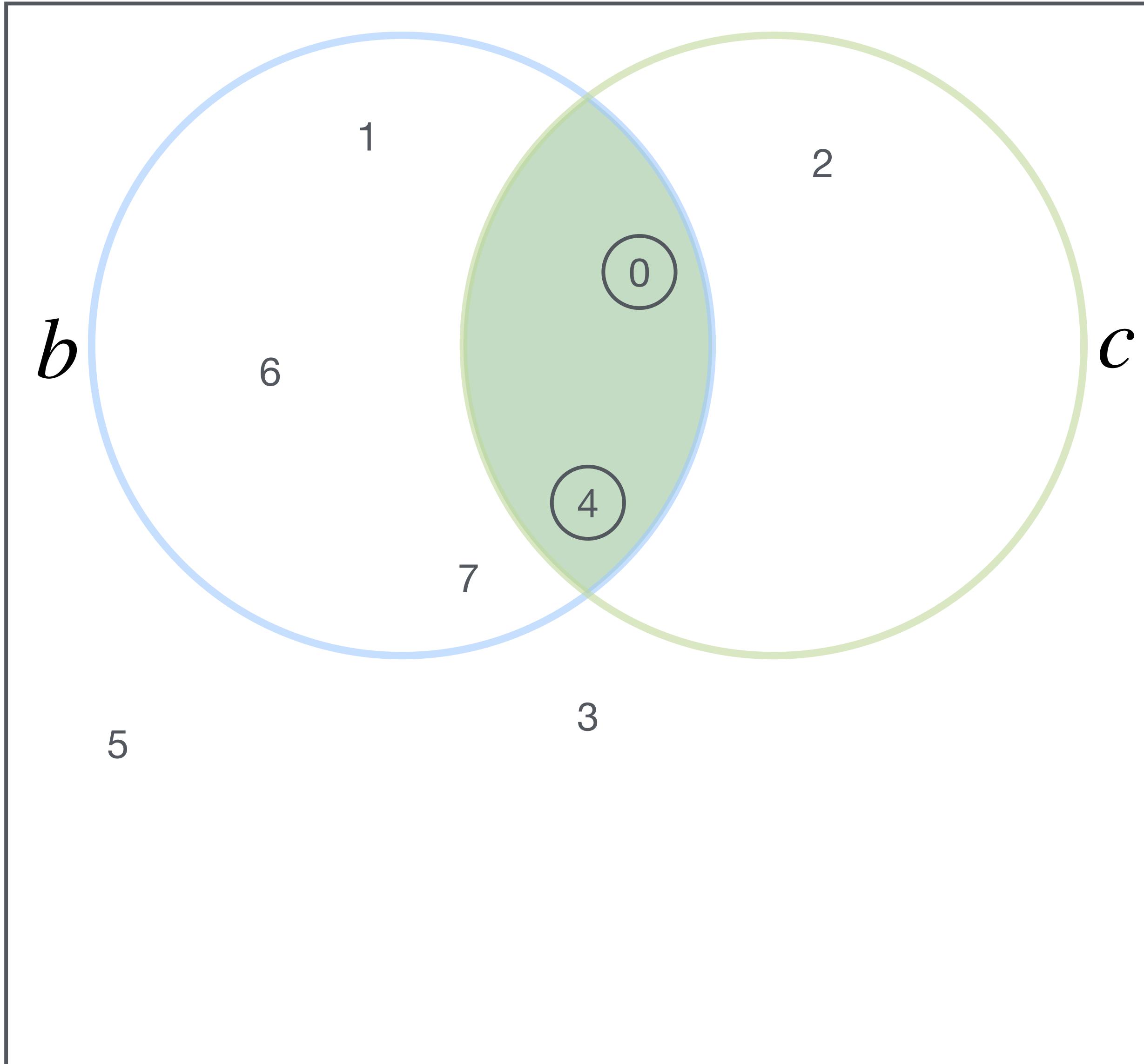
Data structure coiteration

Coordinate Space



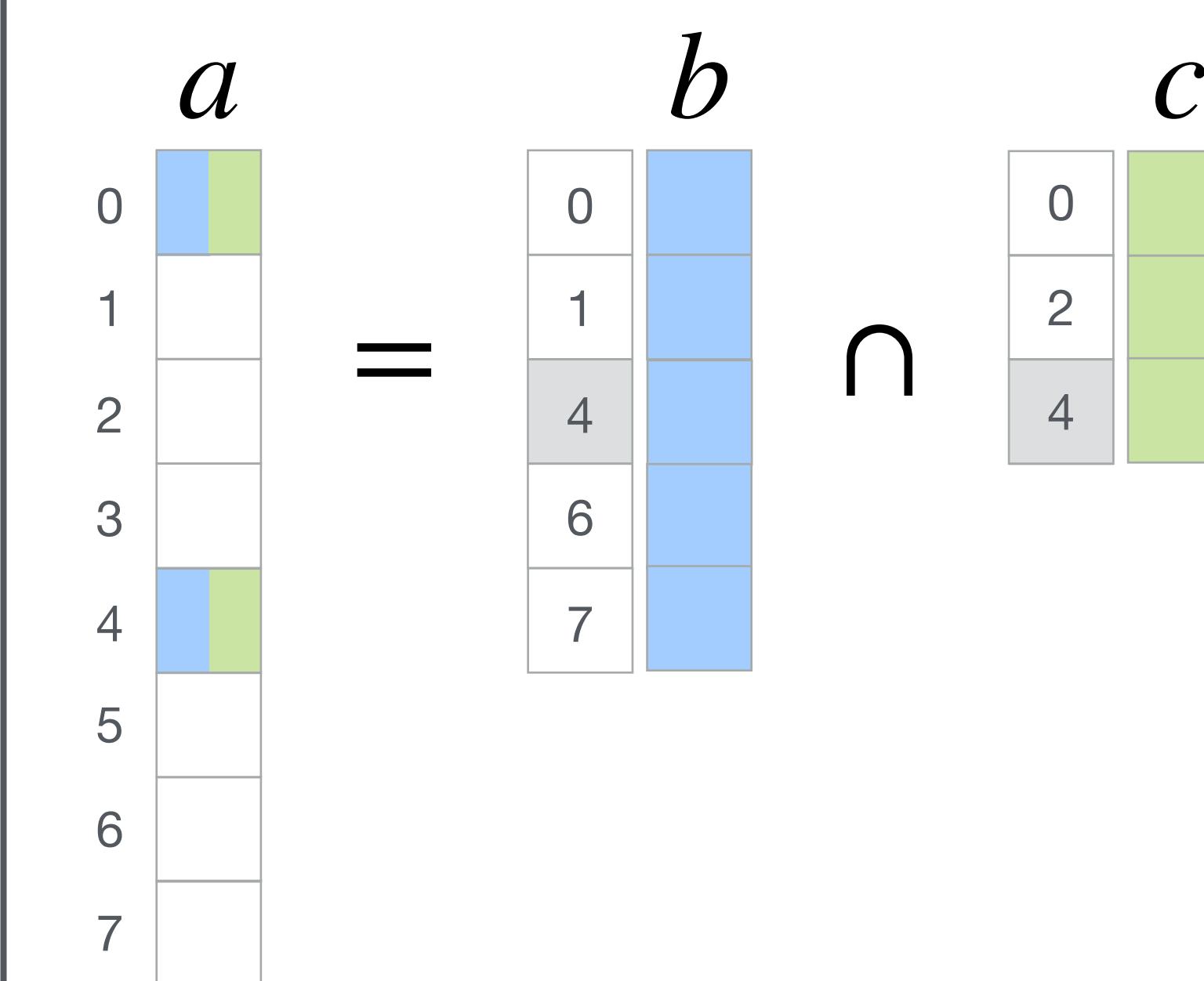
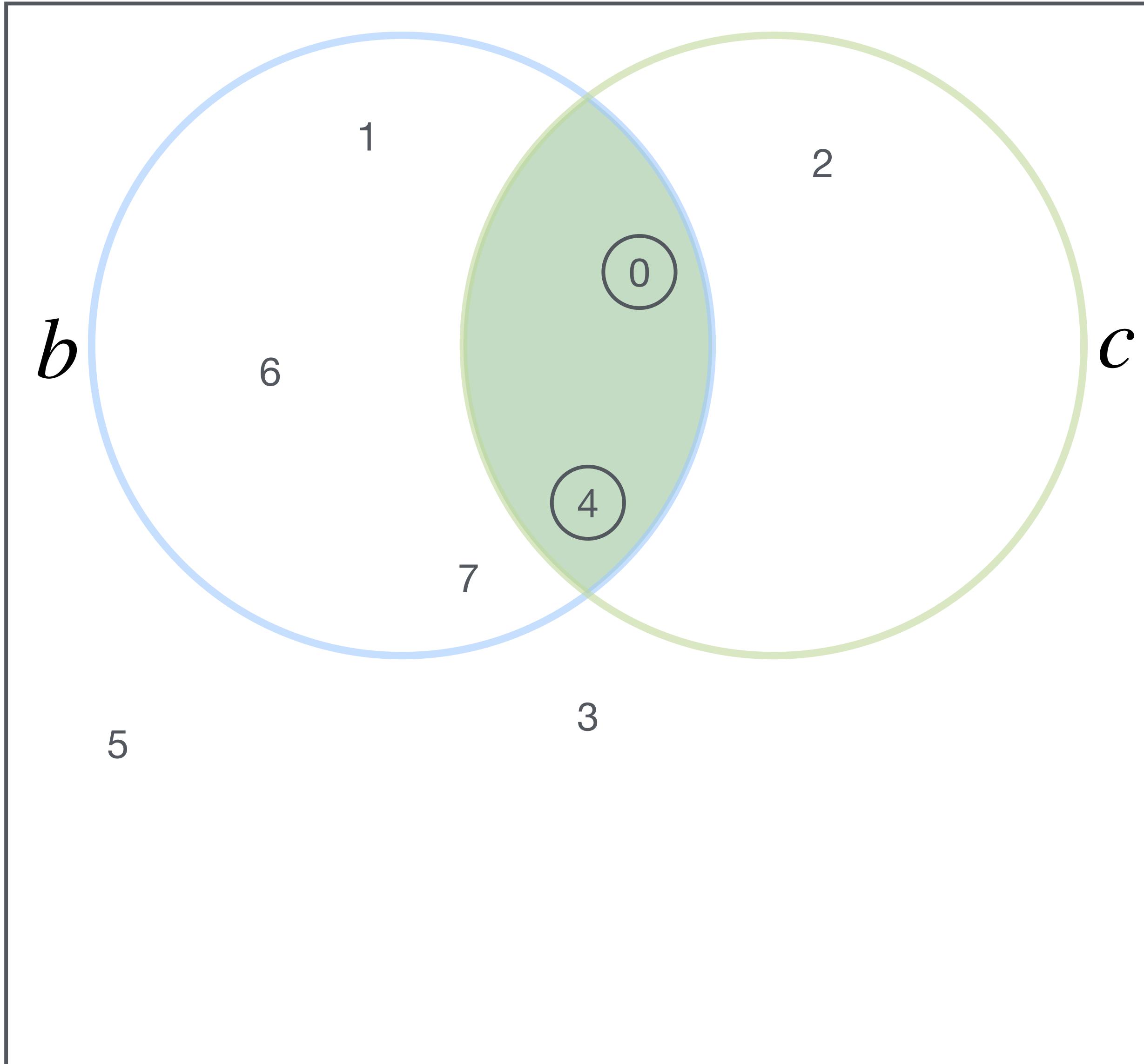
Data structure coiteration

Coordinate Space



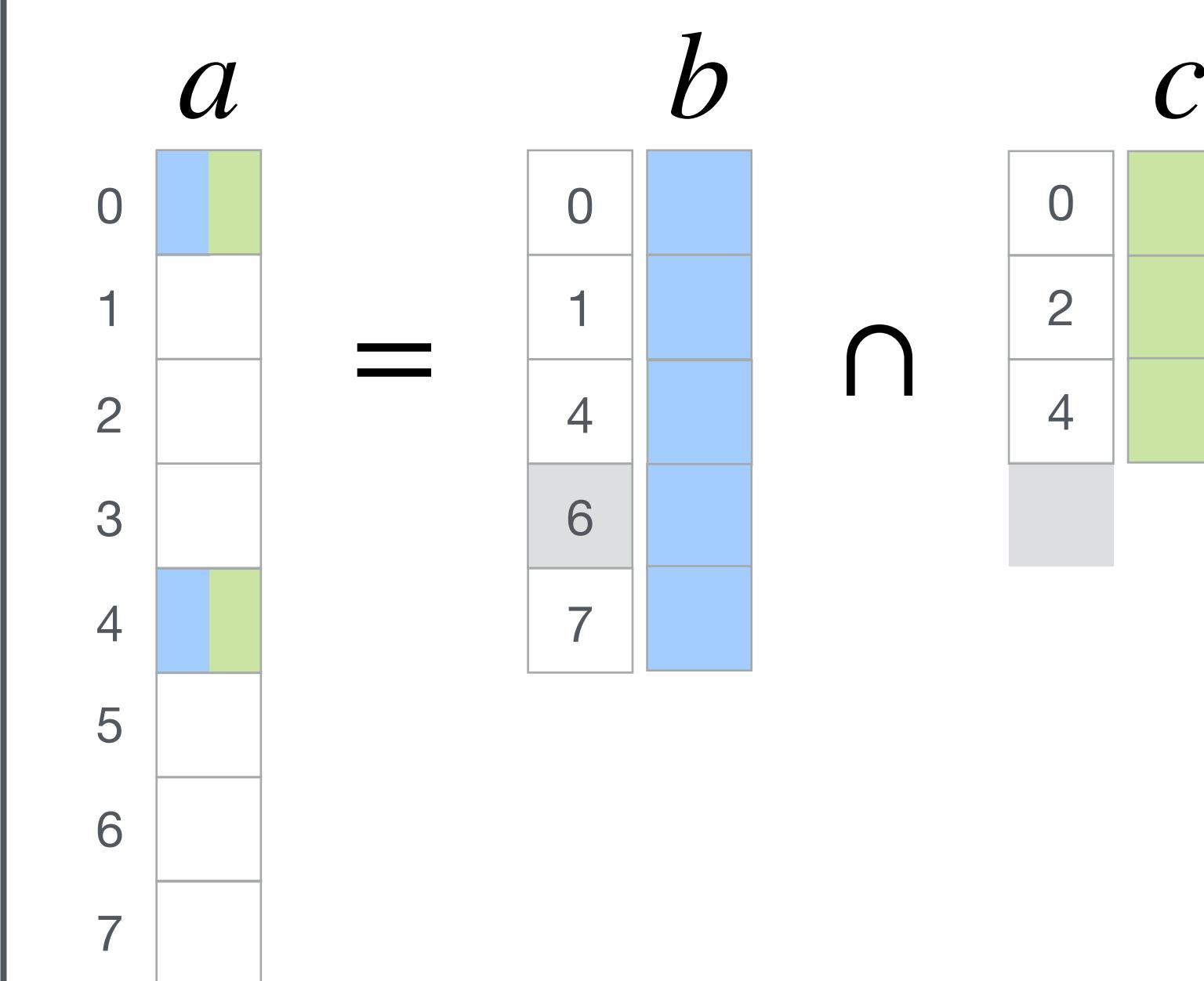
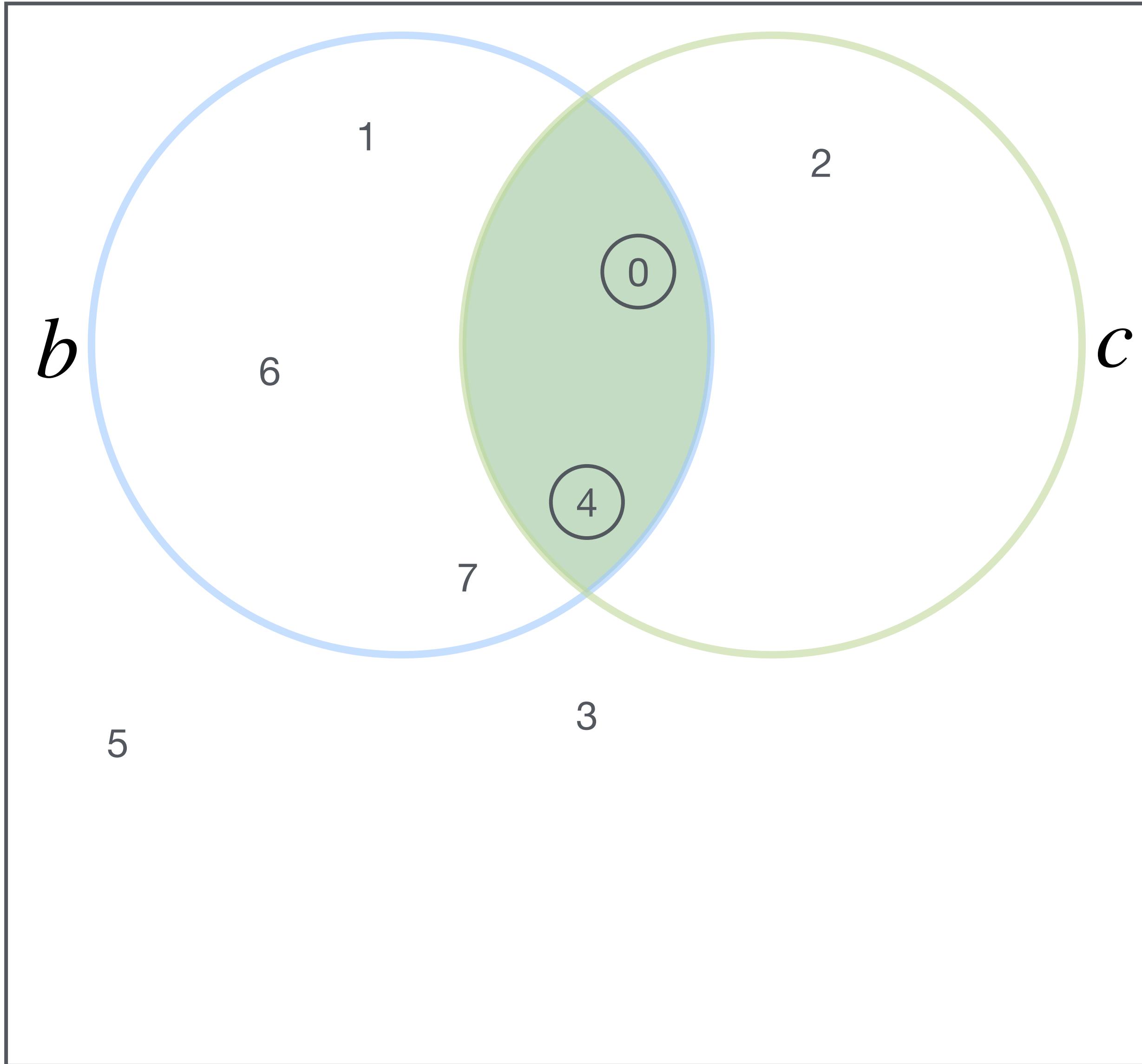
Data structure coiteration

Coordinate Space



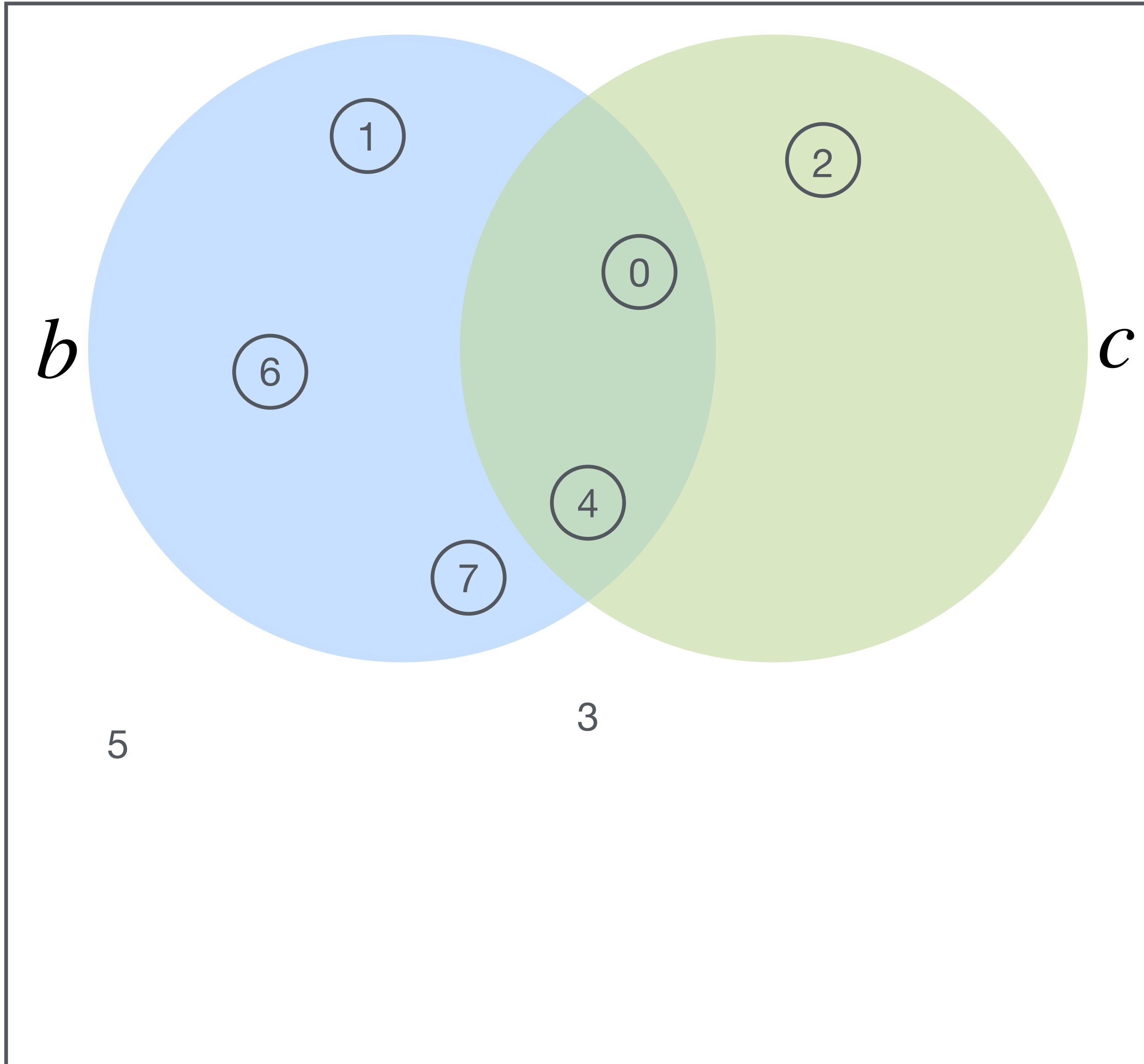
Data structure coiteration

Coordinate Space



Data structure coiteration

Coordinate Space

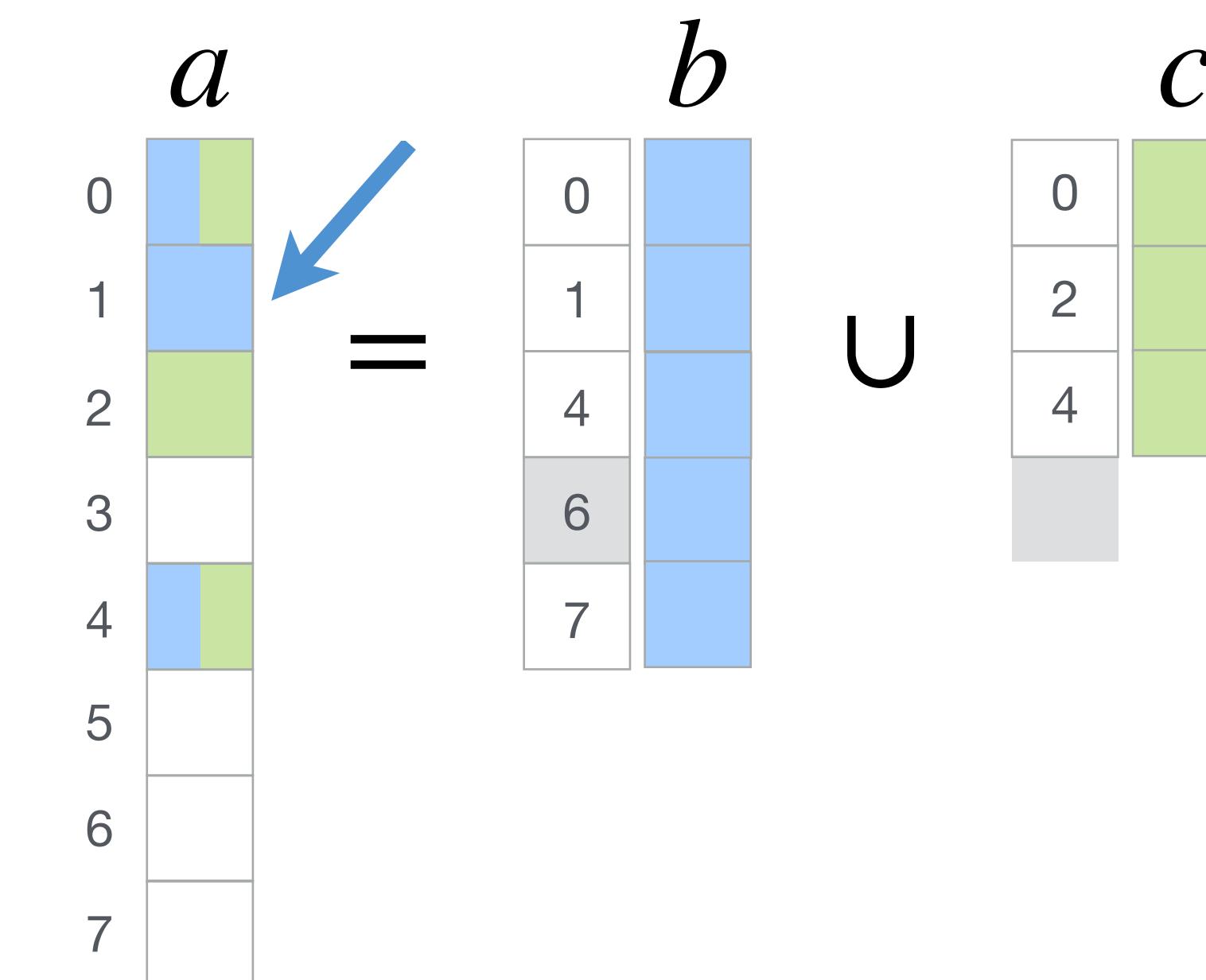
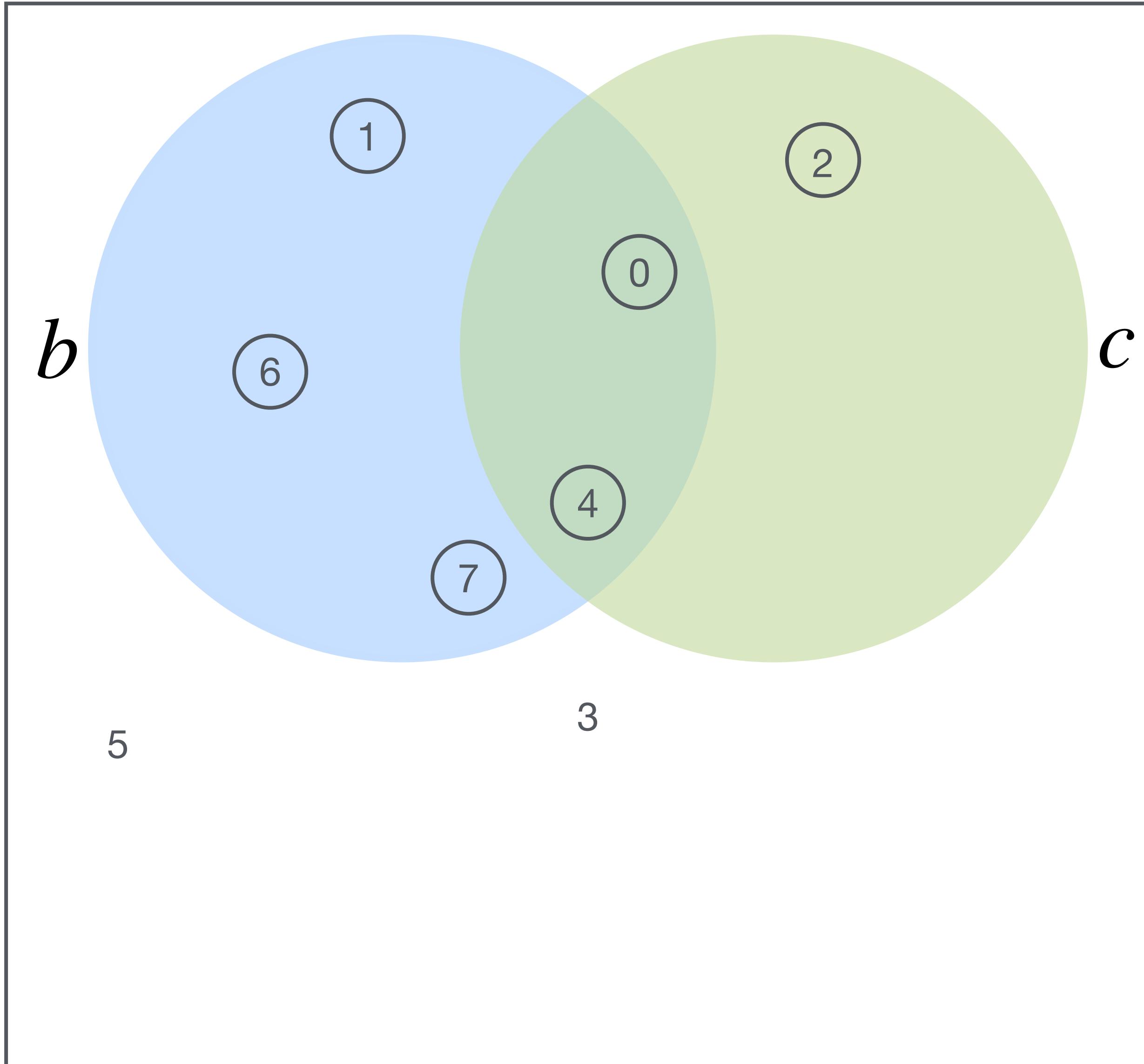


$$a = \begin{array}{c} b \\ \cup \\ c \end{array}$$

The diagram illustrates the union of three sets, a , b , and c . Set a is represented by a vertical stack of 8 boxes, with the first two boxes colored blue and the last one green. Set b is represented by a vertical stack of 8 boxes, with the bottom four boxes colored blue. Set c is represented by a vertical stack of 8 boxes, with the top three boxes colored green. The union of sets b and c is shown as a vertical stack of 8 boxes, with the first two boxes colored blue and the last one green, which corresponds to the intersection of b and c .

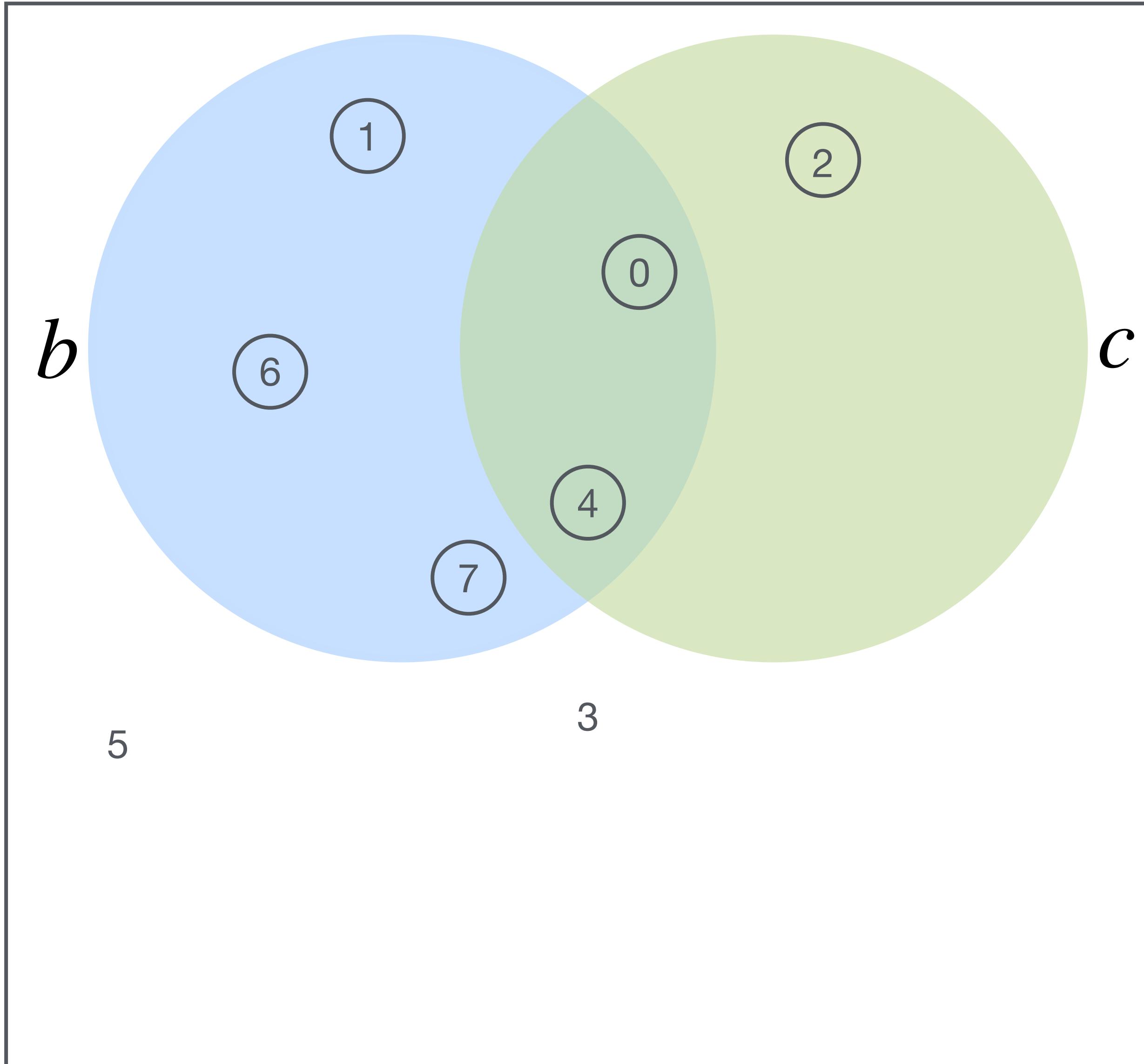
Data structure coiteration

Coordinate Space



Data structure coiteration

Coordinate Space



$$a = \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \quad b = \begin{array}{c} 0 \\ 1 \\ 4 \\ 6 \\ 7 \end{array} \quad c = \begin{array}{c} 0 \\ 2 \\ 4 \end{array}$$

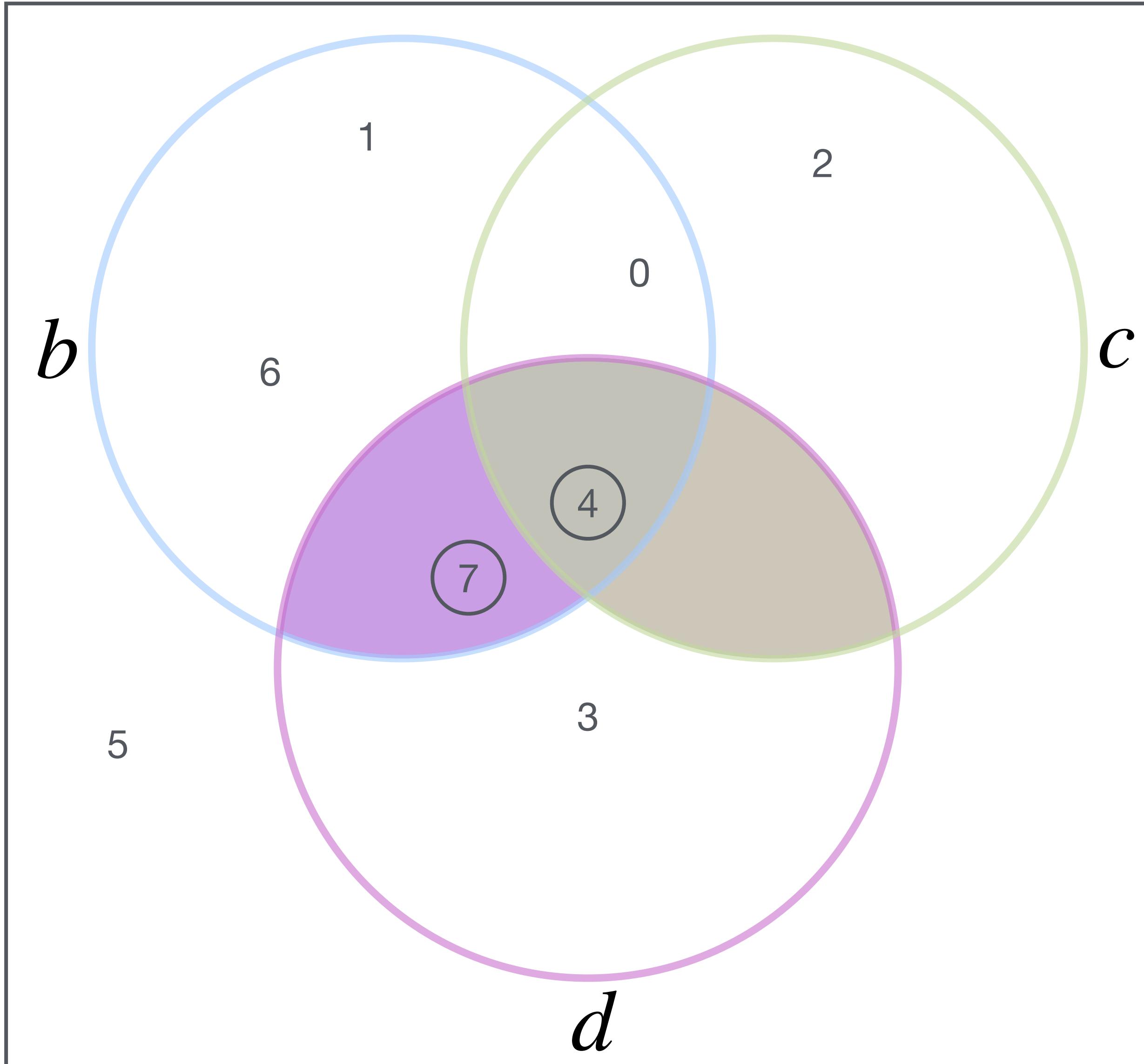
a b c

$=$

\cup

Data structure coiteration

Coordinate Space

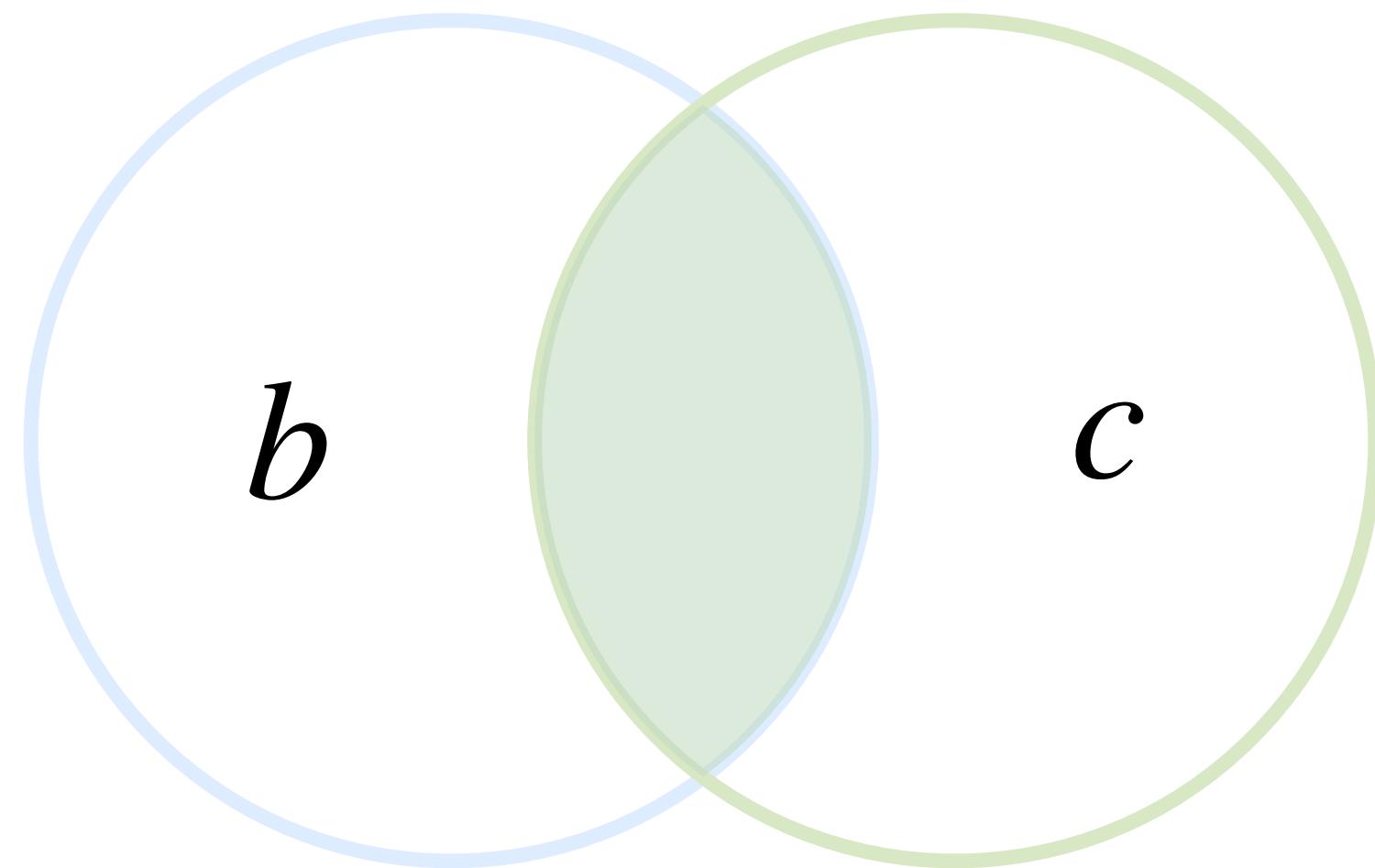


$$a = \left(\begin{array}{c|c} \text{b} & \\ \hline 0 & \\ 1 & \\ 2 & \\ 3 & \\ 4 & \\ 5 & \\ 6 & \\ 7 & \end{array} \right) \cup \left(\begin{array}{c|c} \text{c} & \\ \hline 0 & \\ 2 & \\ 4 & \\ \hline 3 & \\ 4 & \\ 7 & \end{array} \right) \cap \left(\begin{array}{c|c} \text{d} & \\ \hline 3 & \\ 4 & \\ 7 & \end{array} \right)$$

Iteration lattice for multiplications

$$a_i = b_i c_i$$

Iteration lattice for multiplications



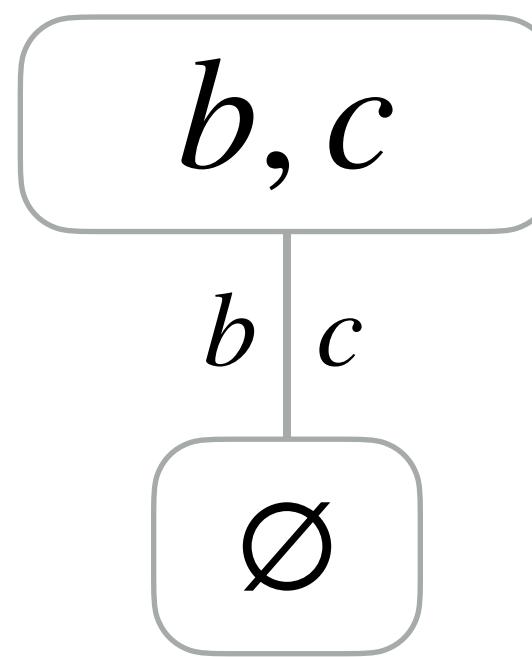
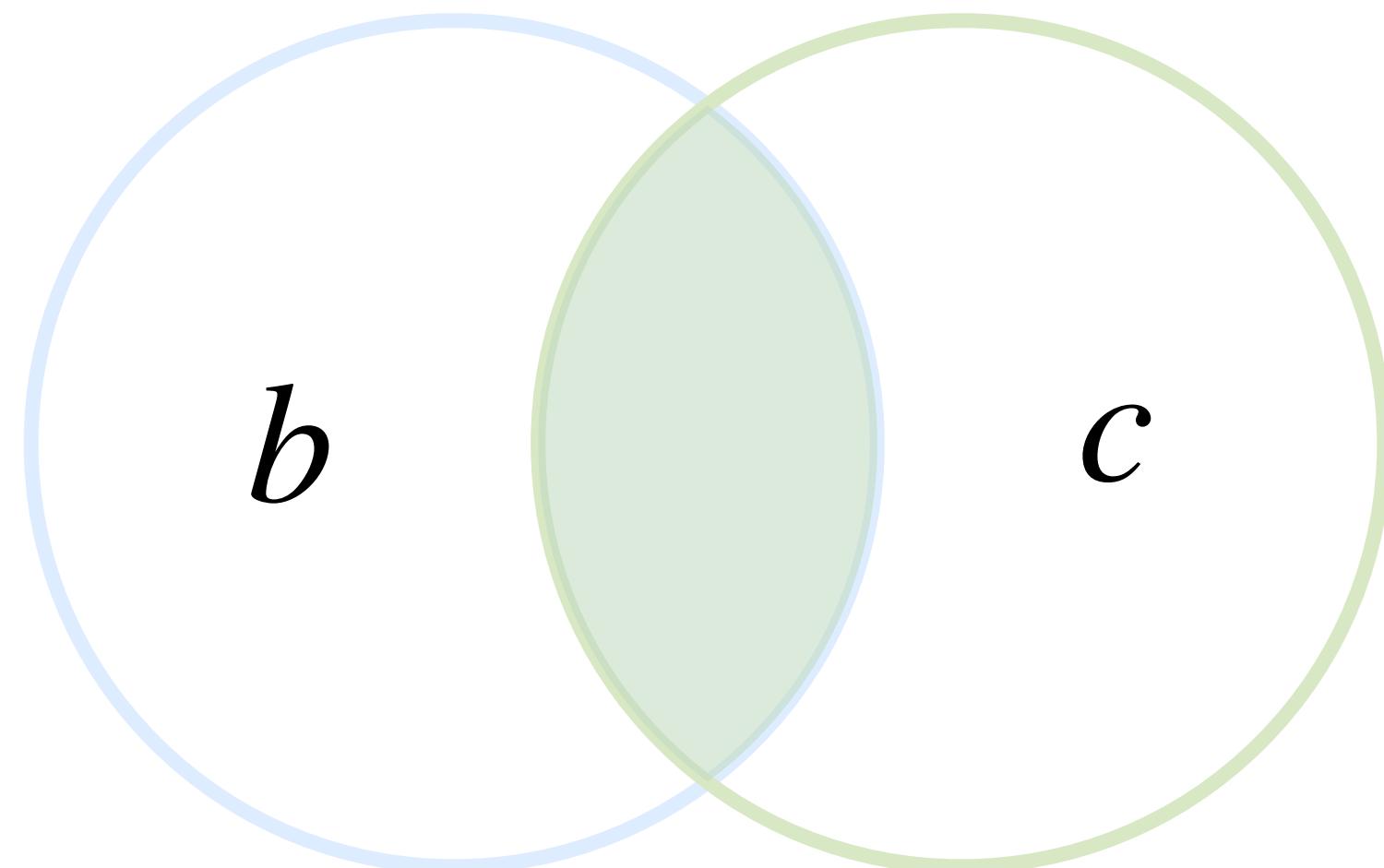
$$a_i = b_i c_i$$

Multiplication requires intersection

$$b \cap c$$

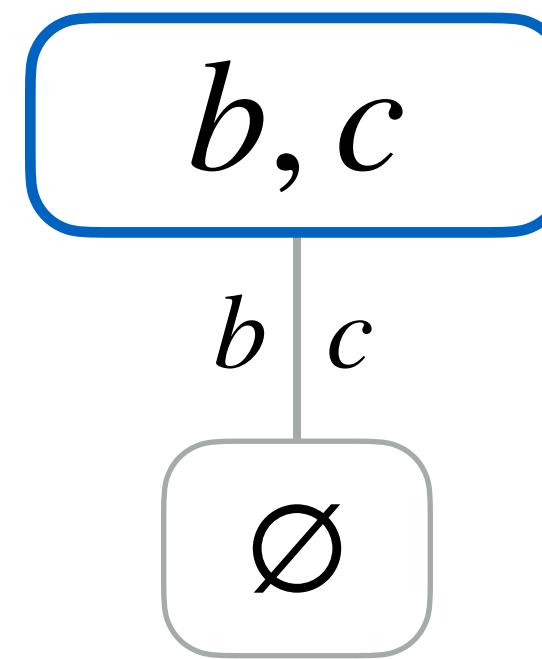
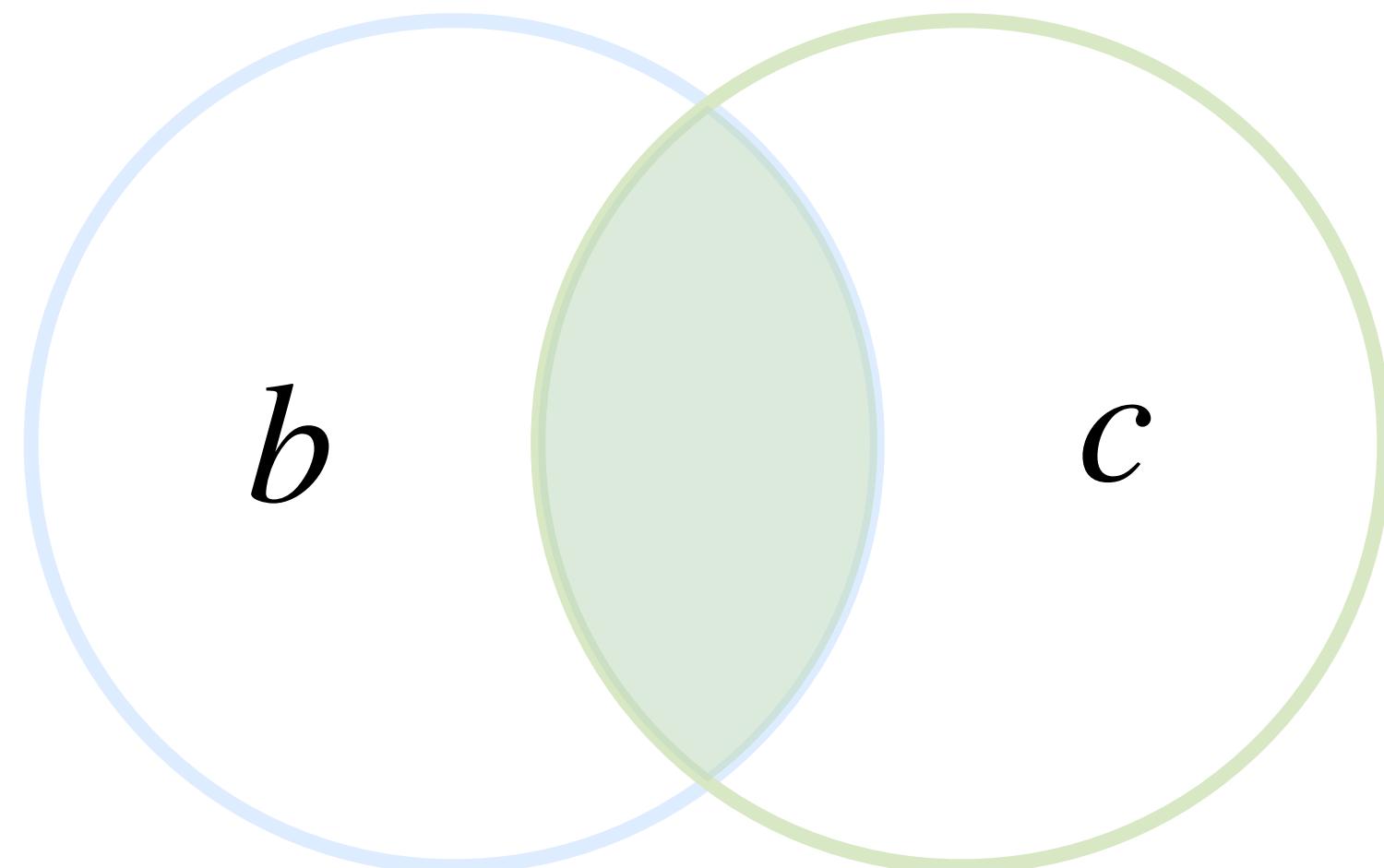
Iteration lattice for multiplications

$$a_i = b_i c_i$$



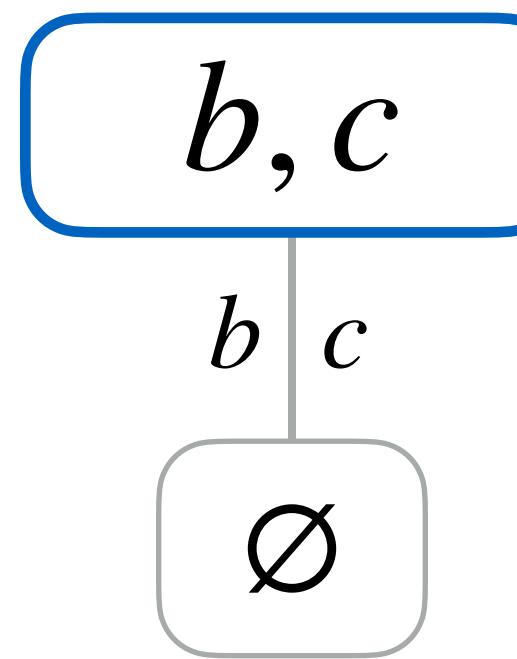
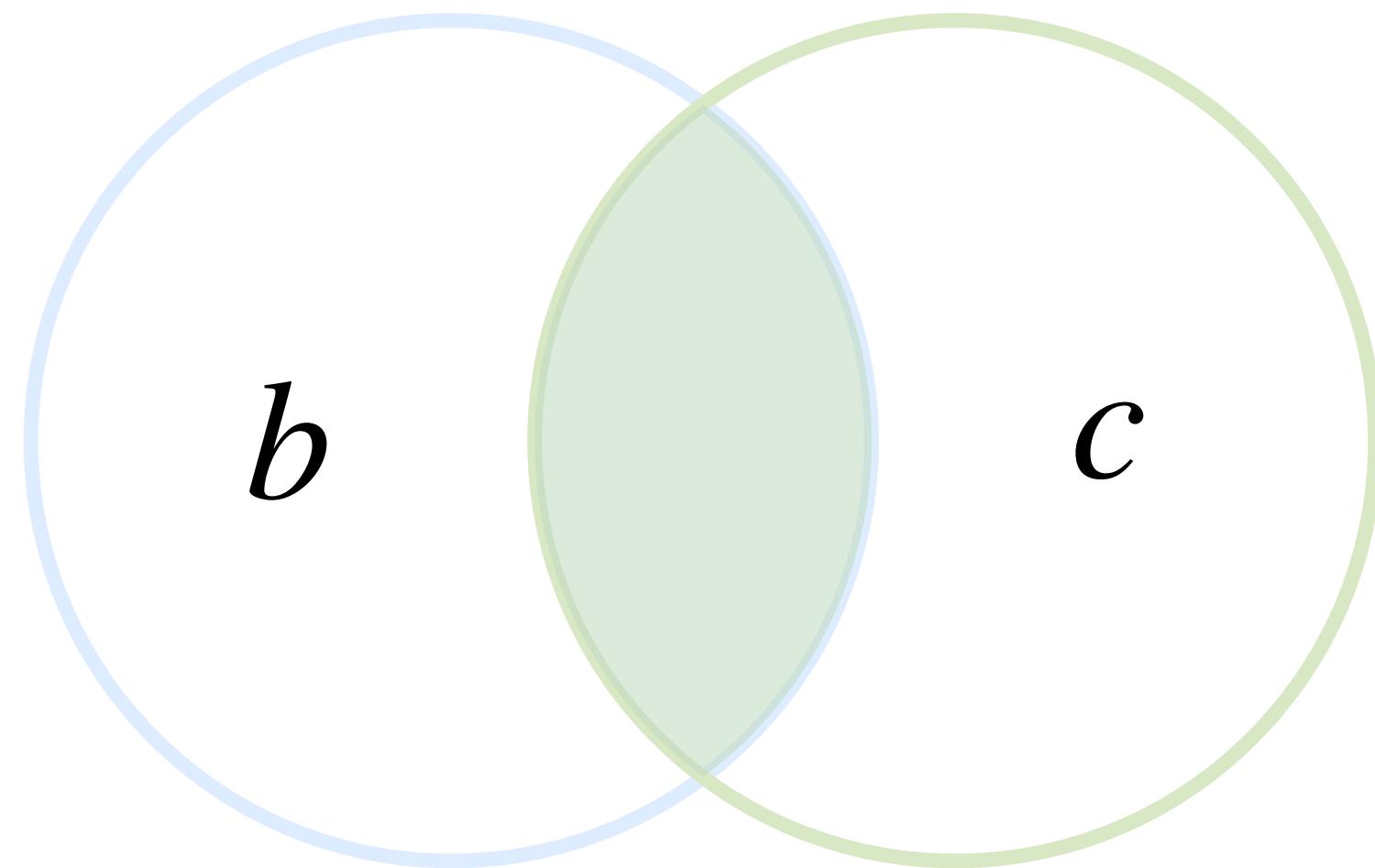
Iteration lattice for multiplications

$$a_i = b_i c_i$$



Iteration lattice for multiplications

$$a_i = b_i c_i$$

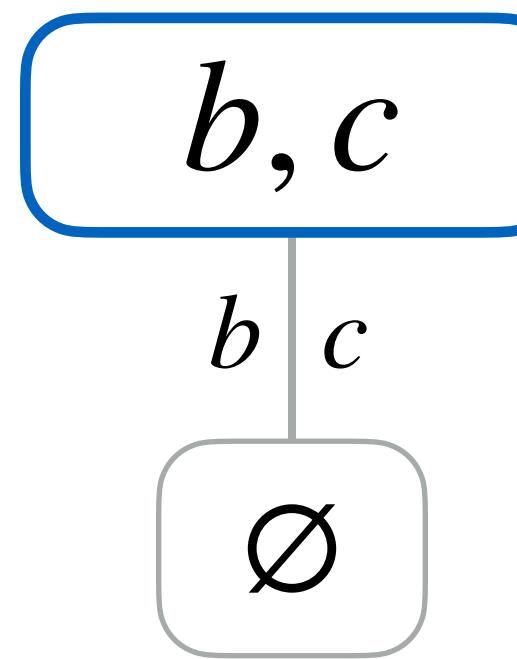
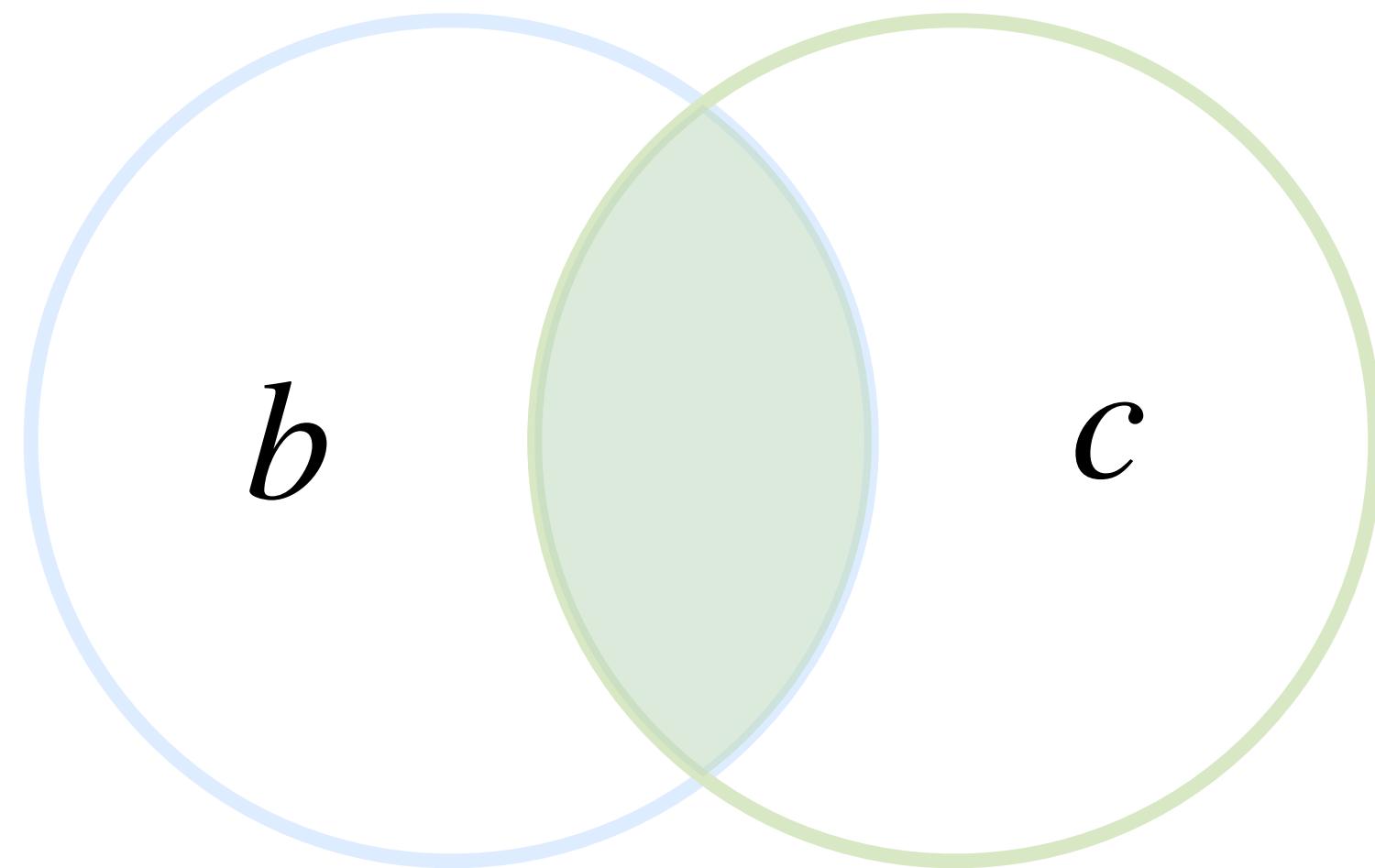


```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int i = min(ib, ic);

    if (ib == i) pb1++;
    if (ic == i) pc1++;
}
```

Iteration lattice for multiplications

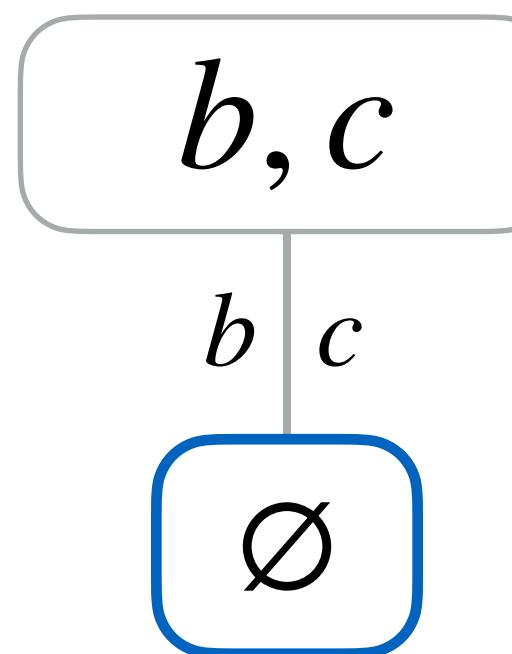
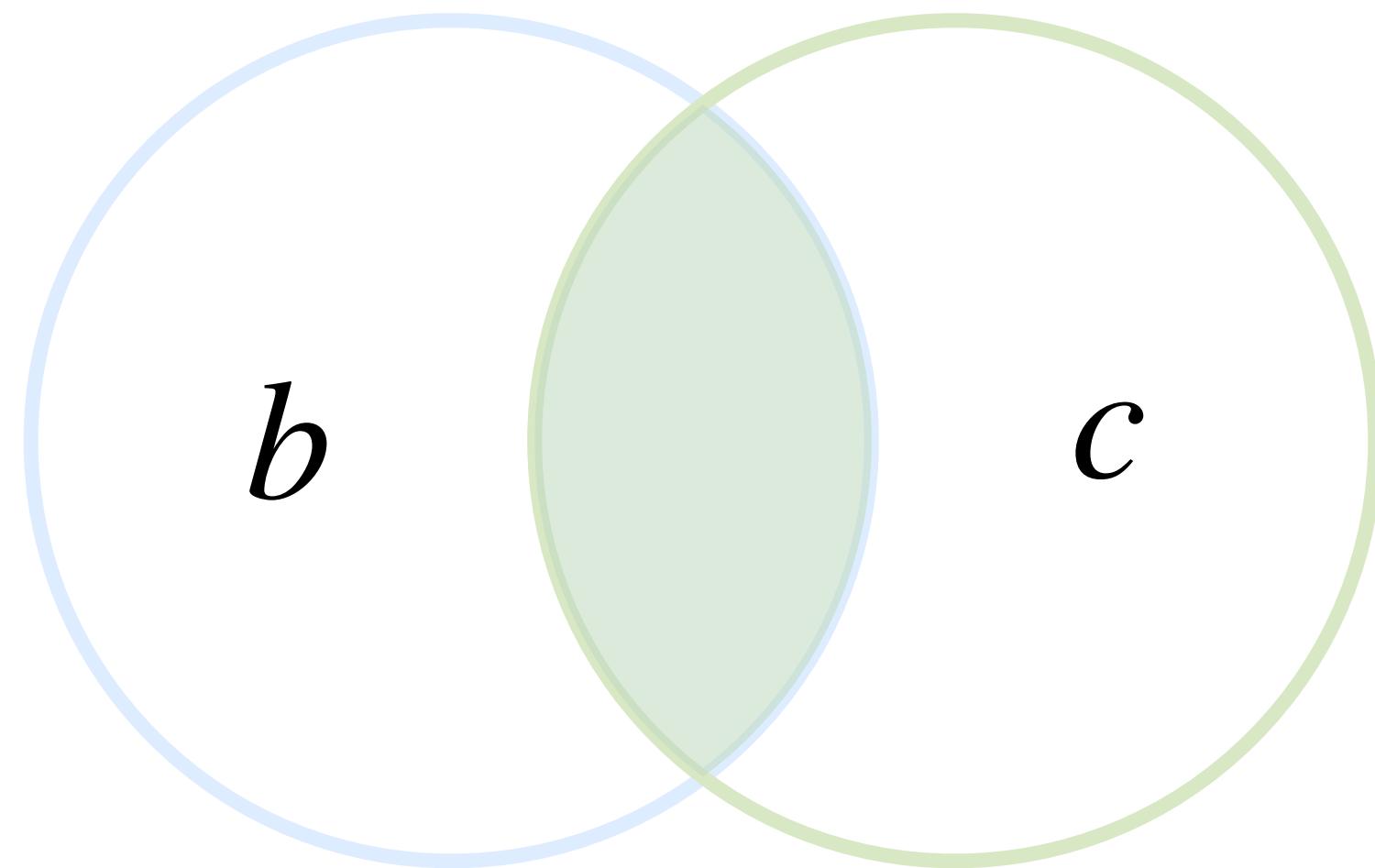
$$a_i = b_i c_i$$



```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int i = min(ib, ic);
    if (ib == i && ic == i) {
        a[i] = b[pb1] * c[pc1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}
```

Iteration lattice for multiplications

$$a_i = b_i c_i$$

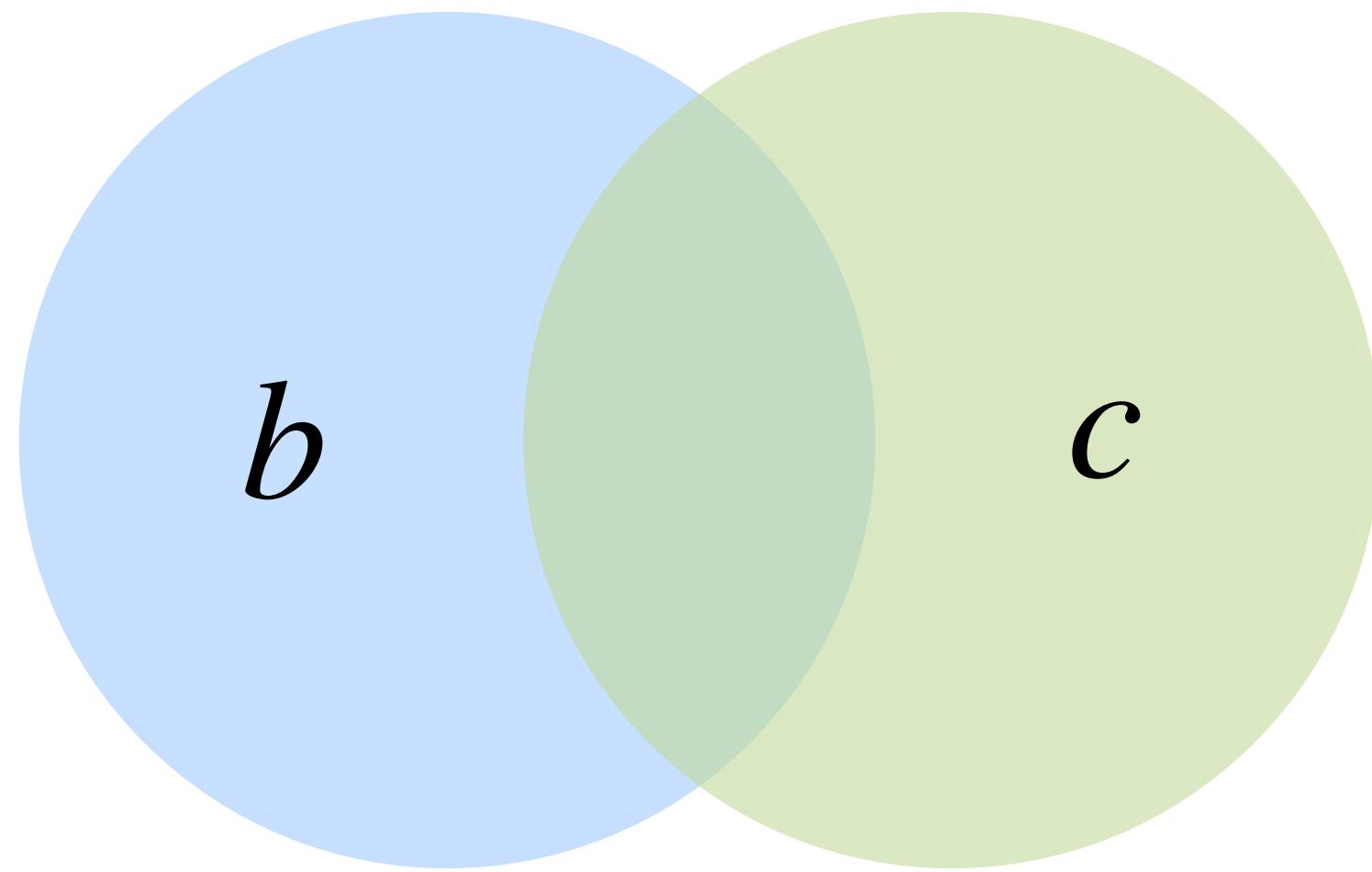


```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int i = min(ib, ic);
    if (ib == i && ic == i) {
        a[i] = b[pb1] * c[pc1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}
```

Iteration lattice for additions

$$a_i = b_i + c_i$$

Iteration lattice for additions

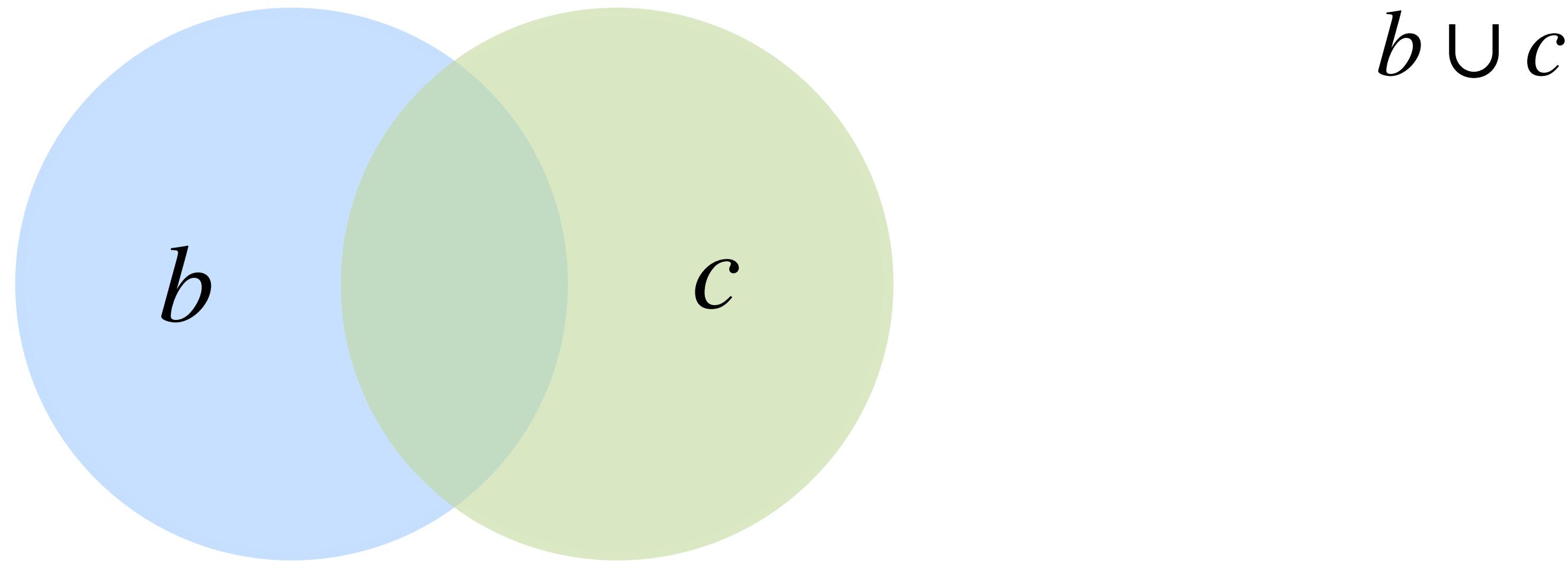


$$a_i = b_i + c_i$$

Addition requires union

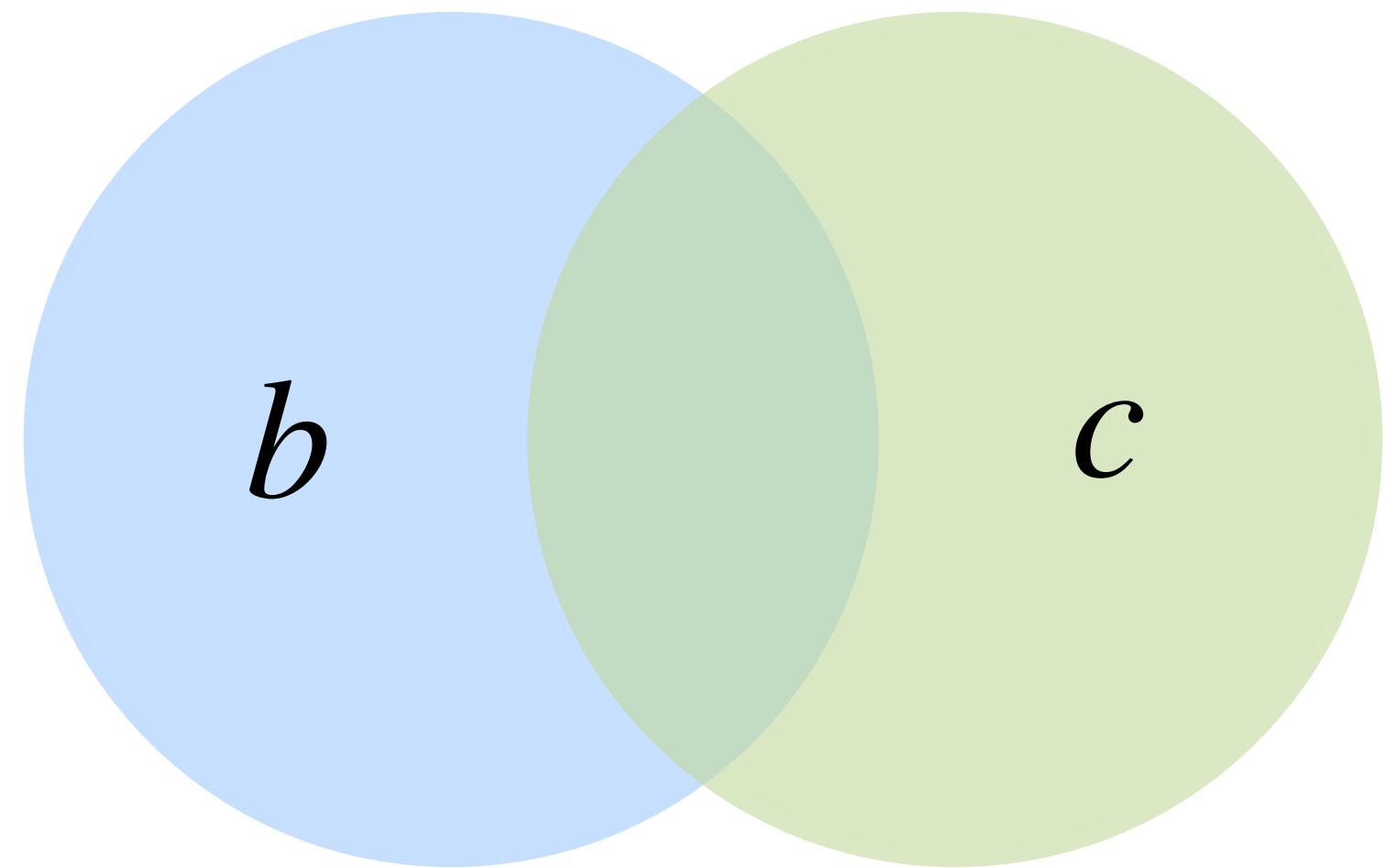
Iteration lattice for additions

$$a_i = b_i + c_i$$



Iteration lattice for additions

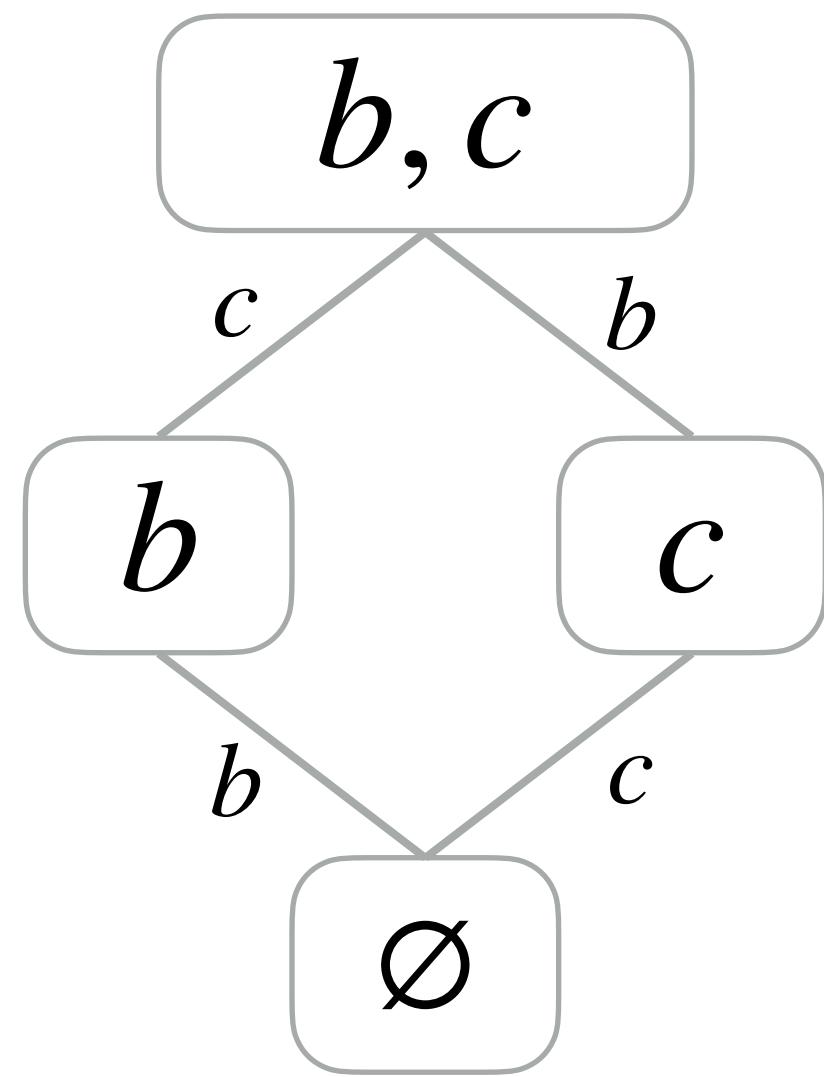
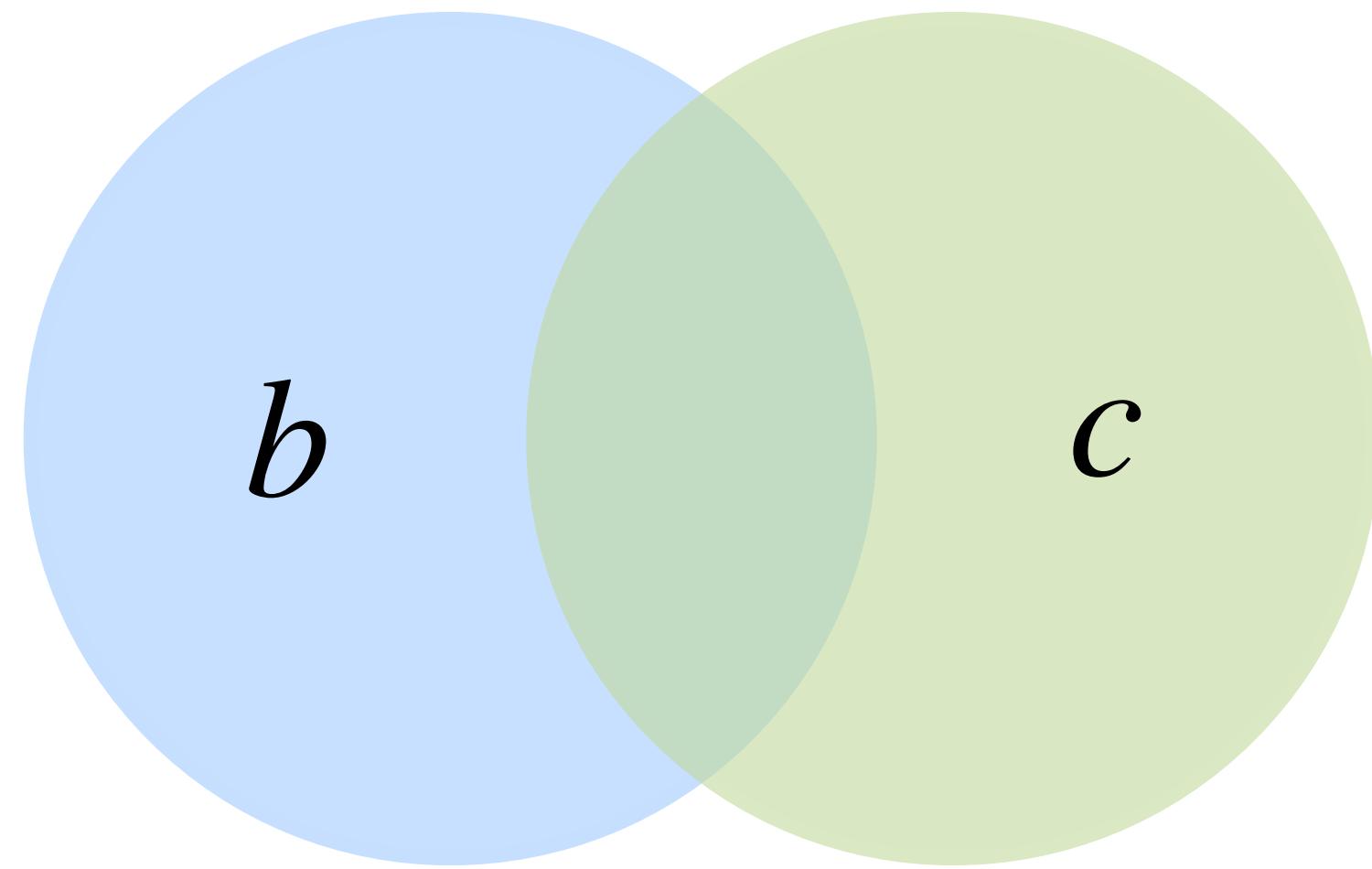
$$a_i = b_i + c_i$$



$$b \cup c = (b \cap c) \cup b \cup c$$

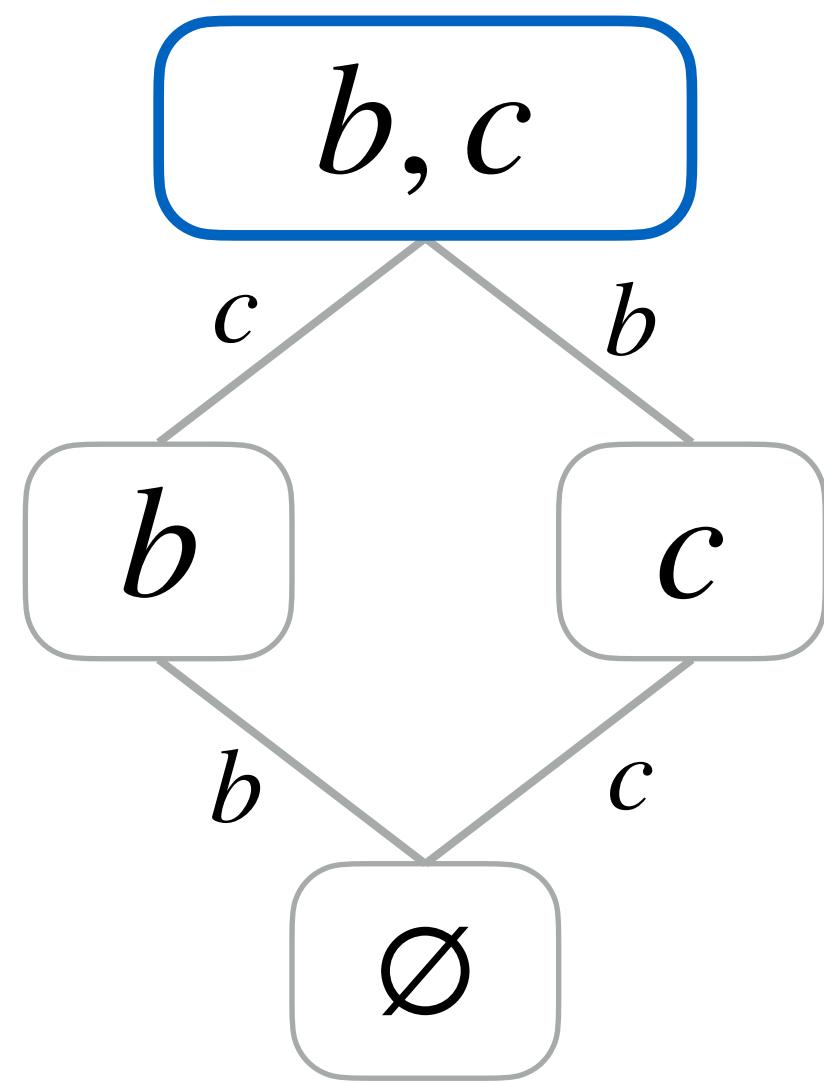
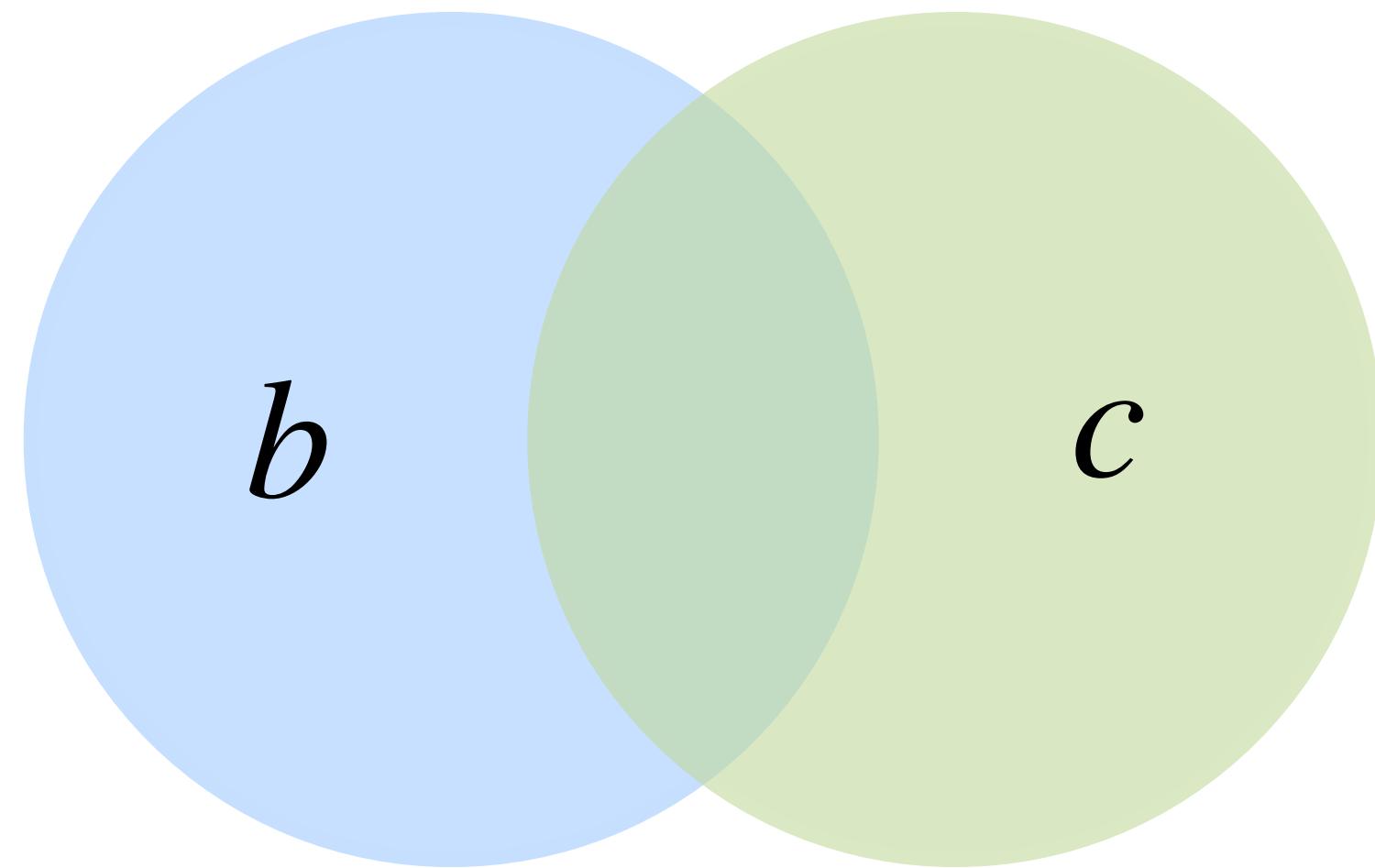
Iteration lattice for additions

$$a_i = b_i + c_i$$



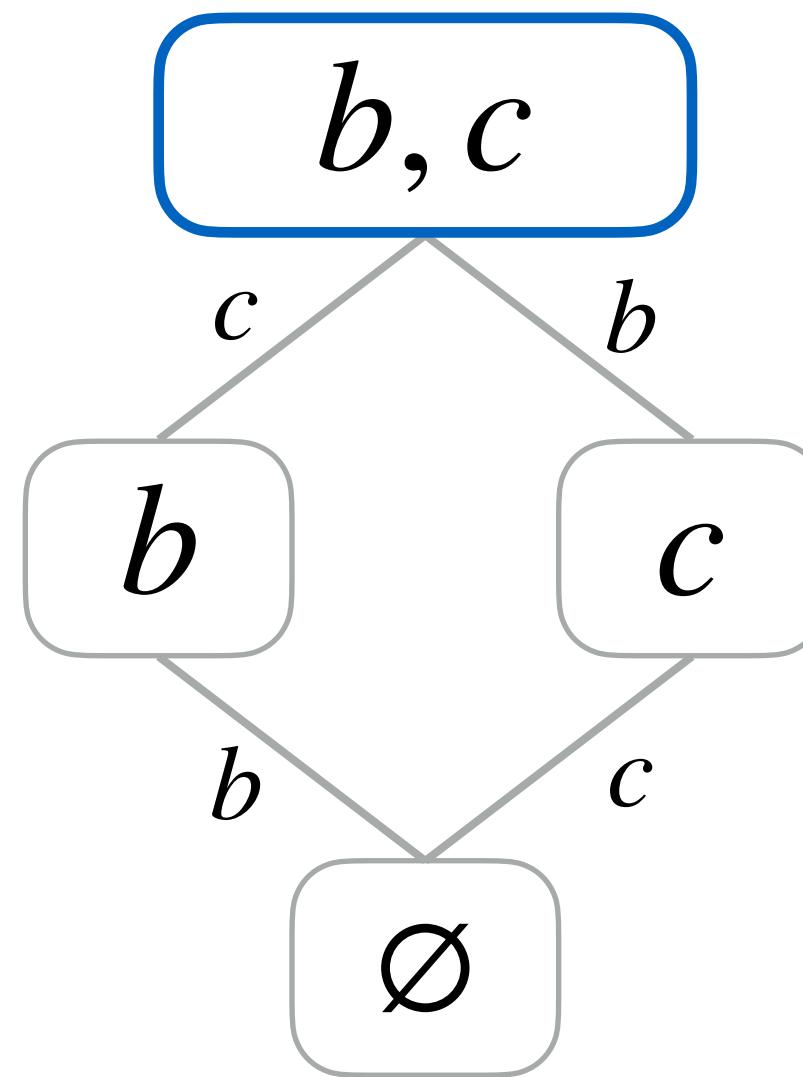
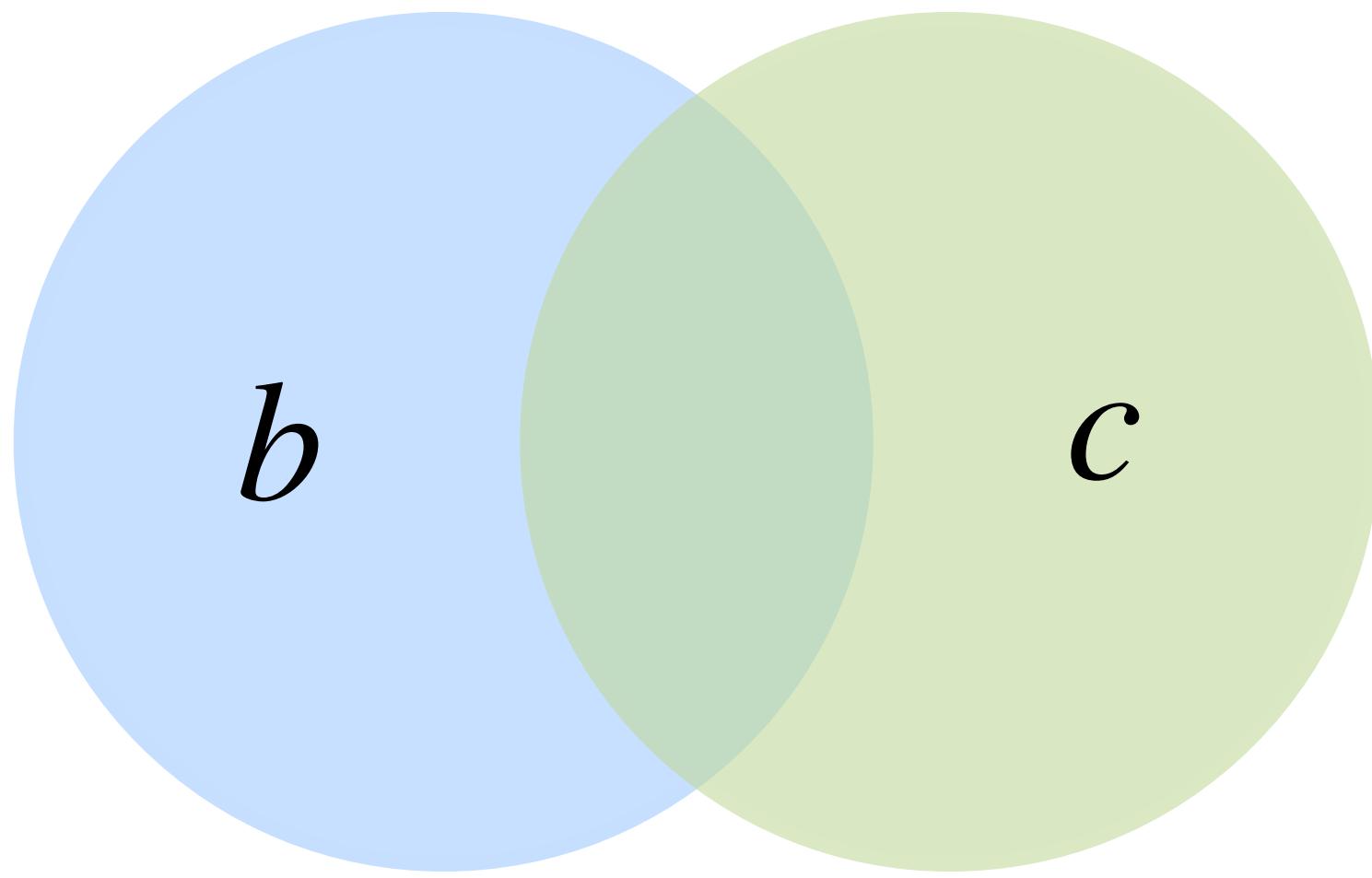
Iteration lattice for additions

$$a_i = b_i + c_i$$



Iteration lattice for additions

$$a_i = b_i + c_i$$

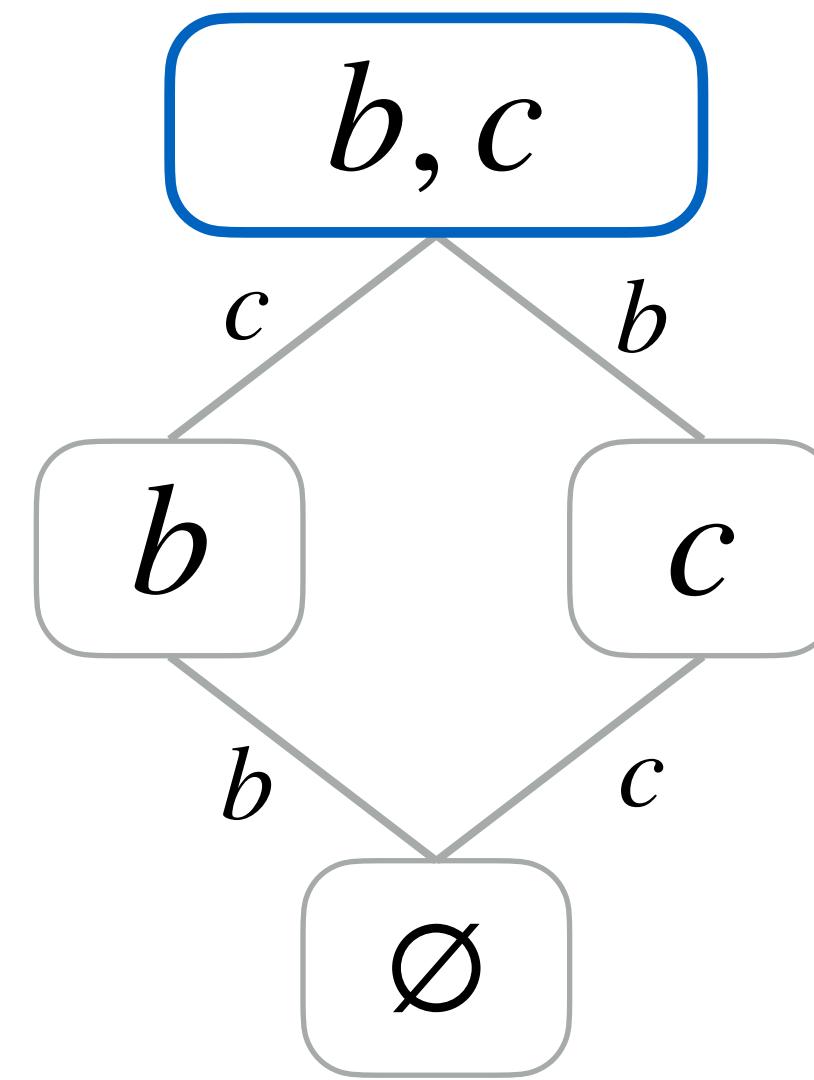
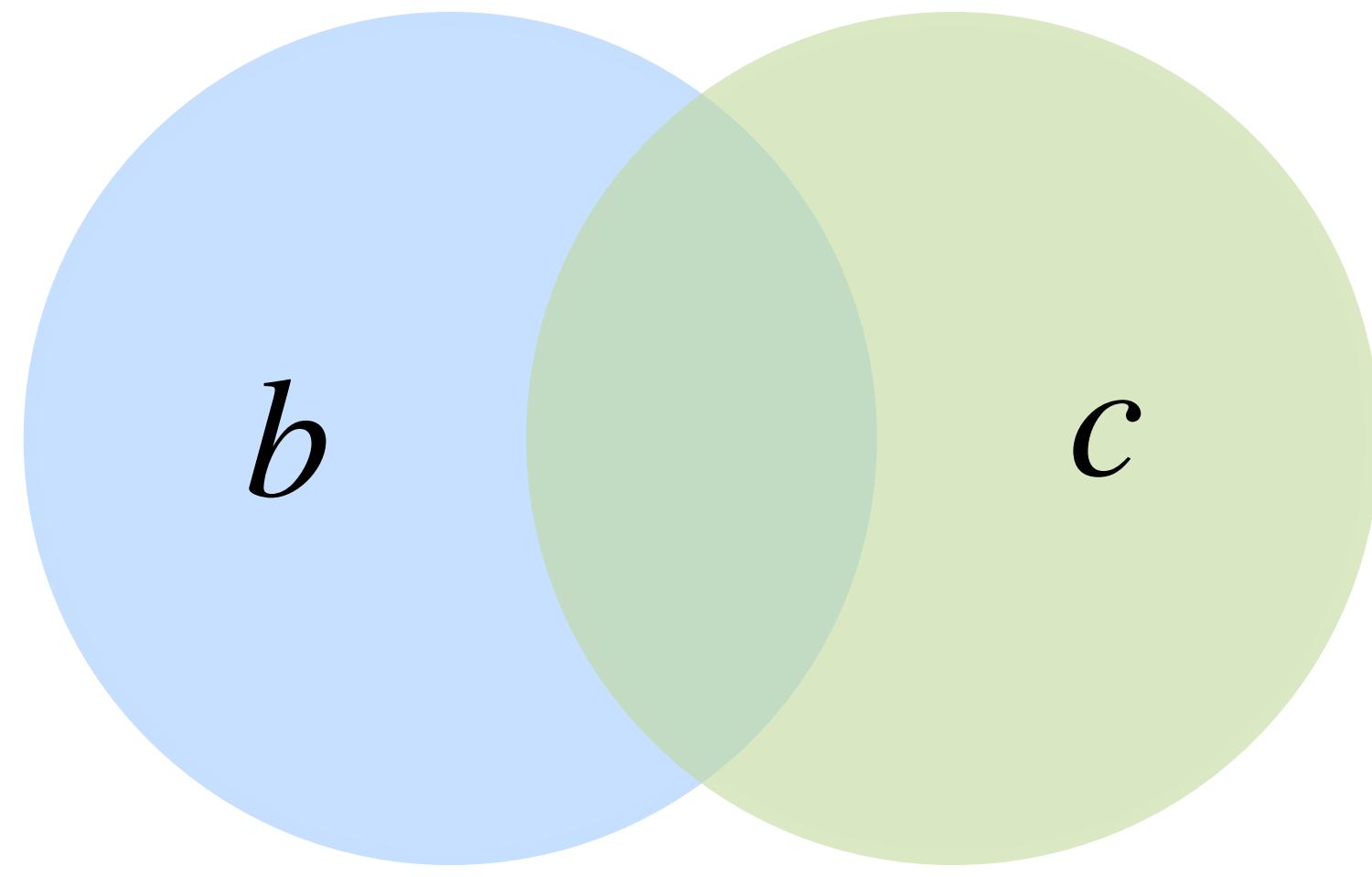


```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int i = min(ib, ic);

    if (ib == i) pb1++;
    if (ic == i) pc1++;
}
```

Iteration lattice for additions

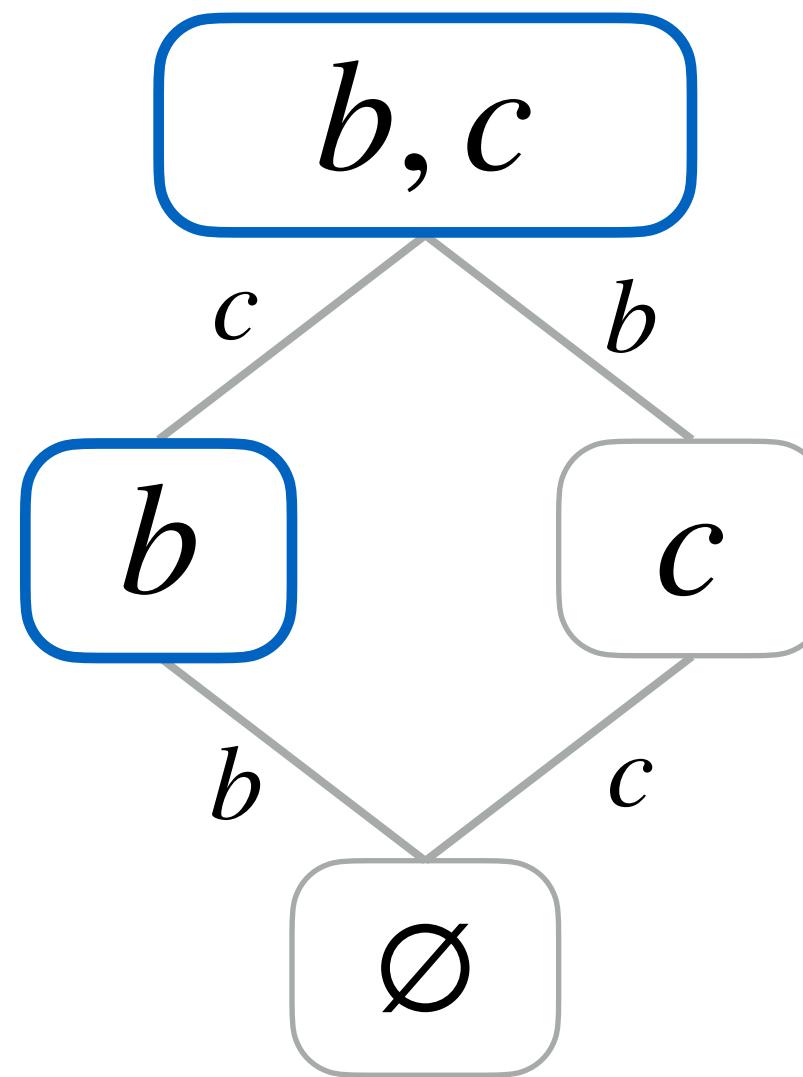
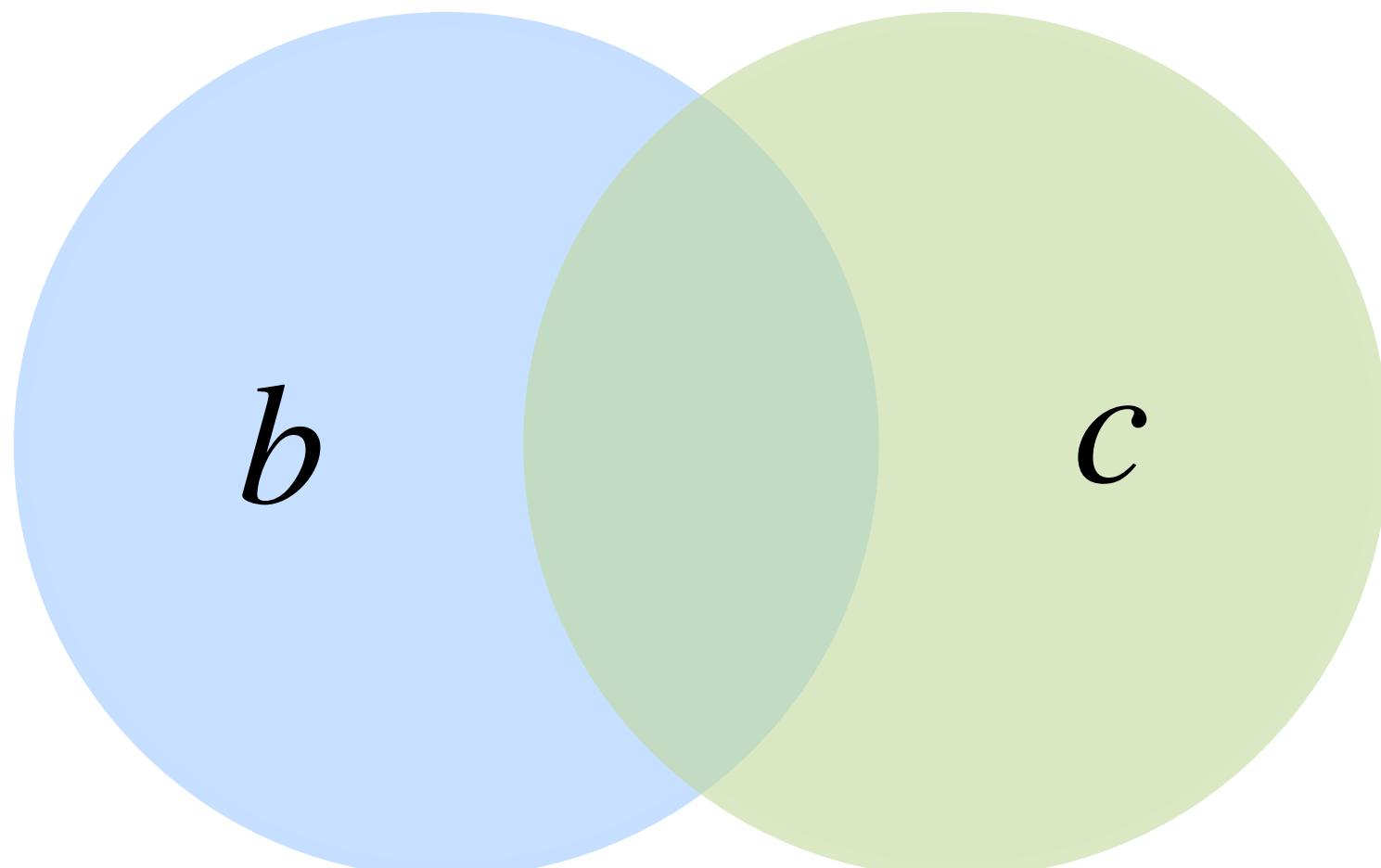
$$a_i = b_i + c_i$$



```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int i = min(ib, ic);
    if (ib == i && ic == i) {
        a[i] = b[pb1] + c[pc1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}
```

Iteration lattice for additions

$$a_i = b_i + c_i$$

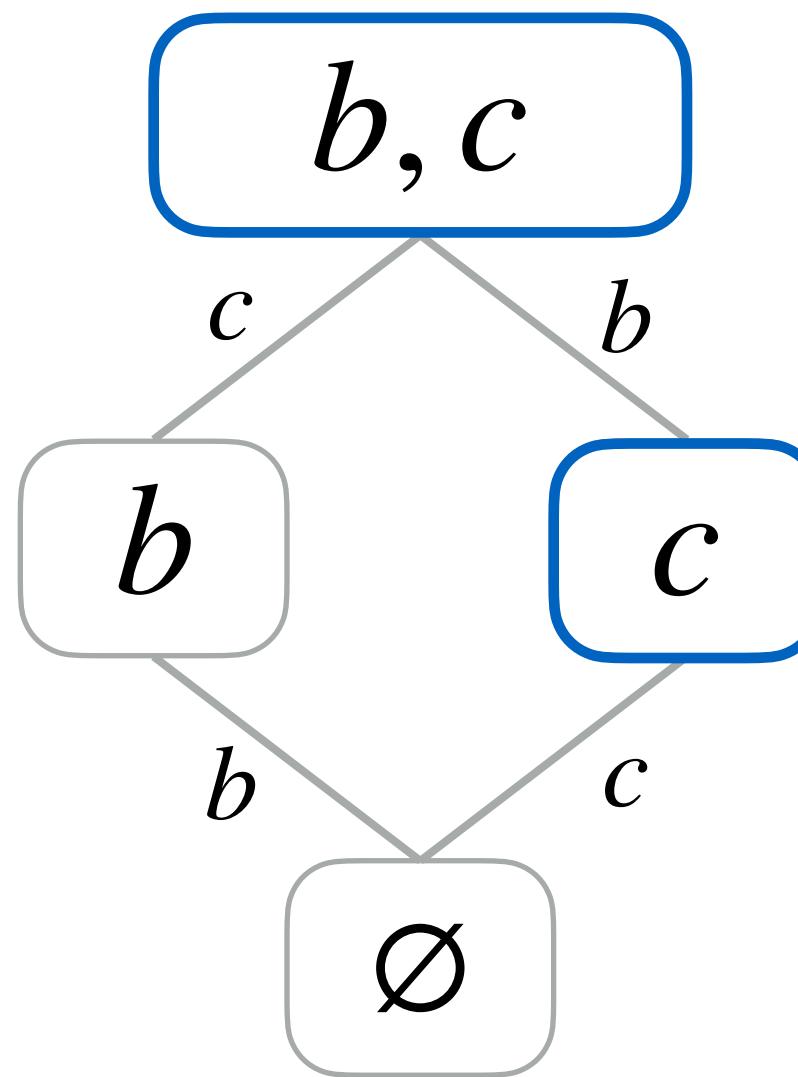
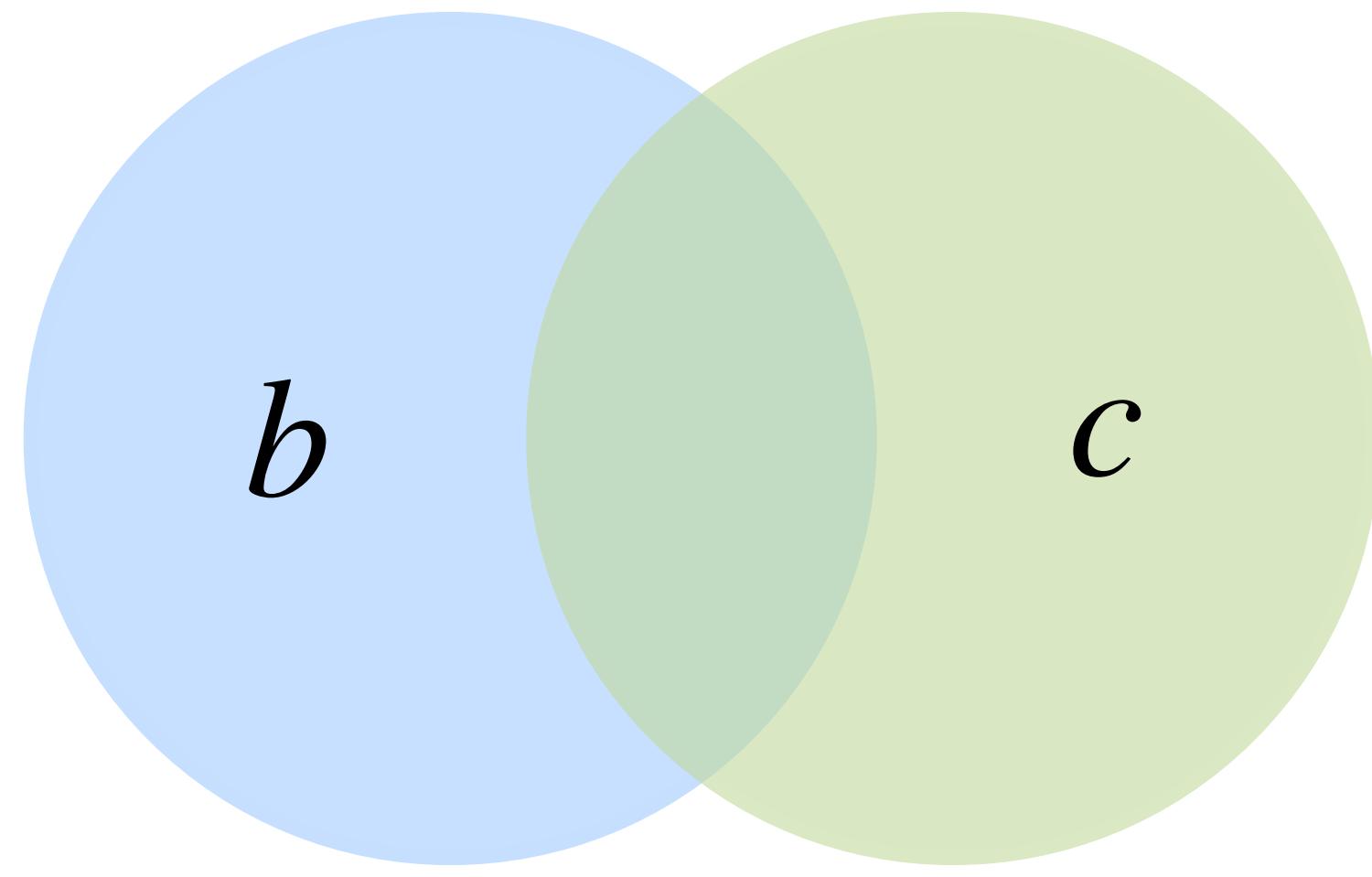


```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int i = min(ib, ic);
    if (ib == i && ic == i) {
        a[i] = b[pb1] + c[pc1];
    } else if (ib == i) {
        a[i] = b[pb1];
    }

    if (ib == i) pb1++;
    if (ic == i) pc1++;
}
```

Iteration lattice for additions

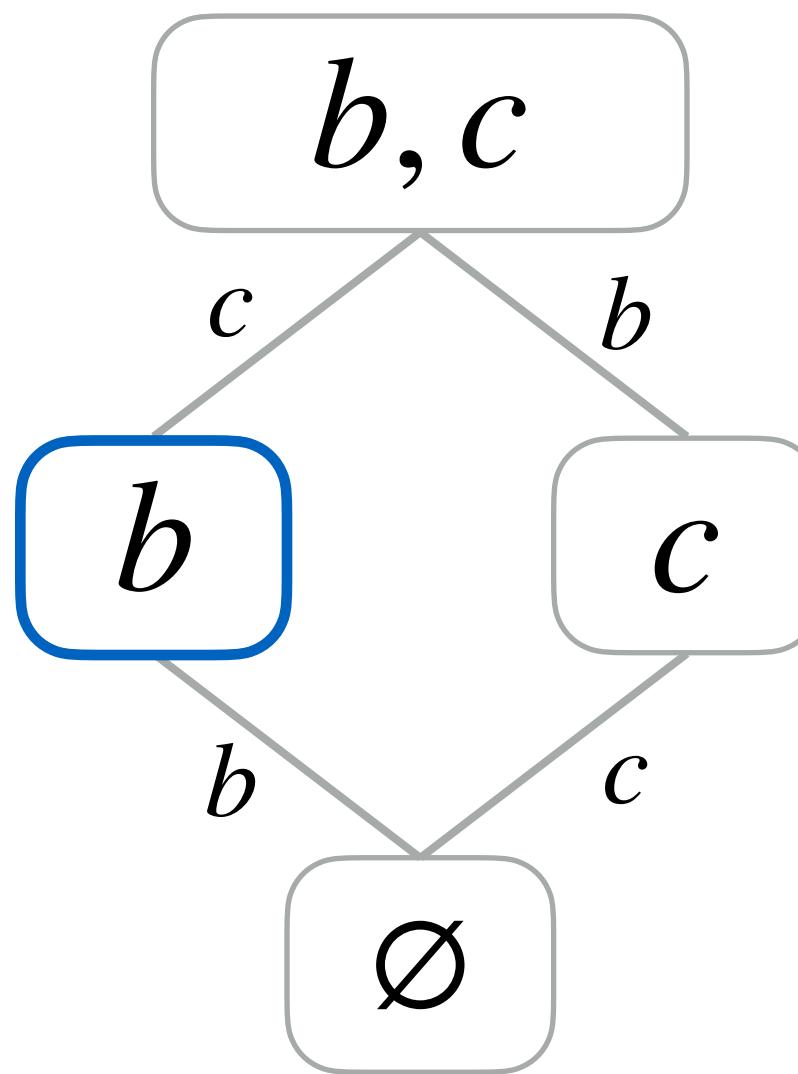
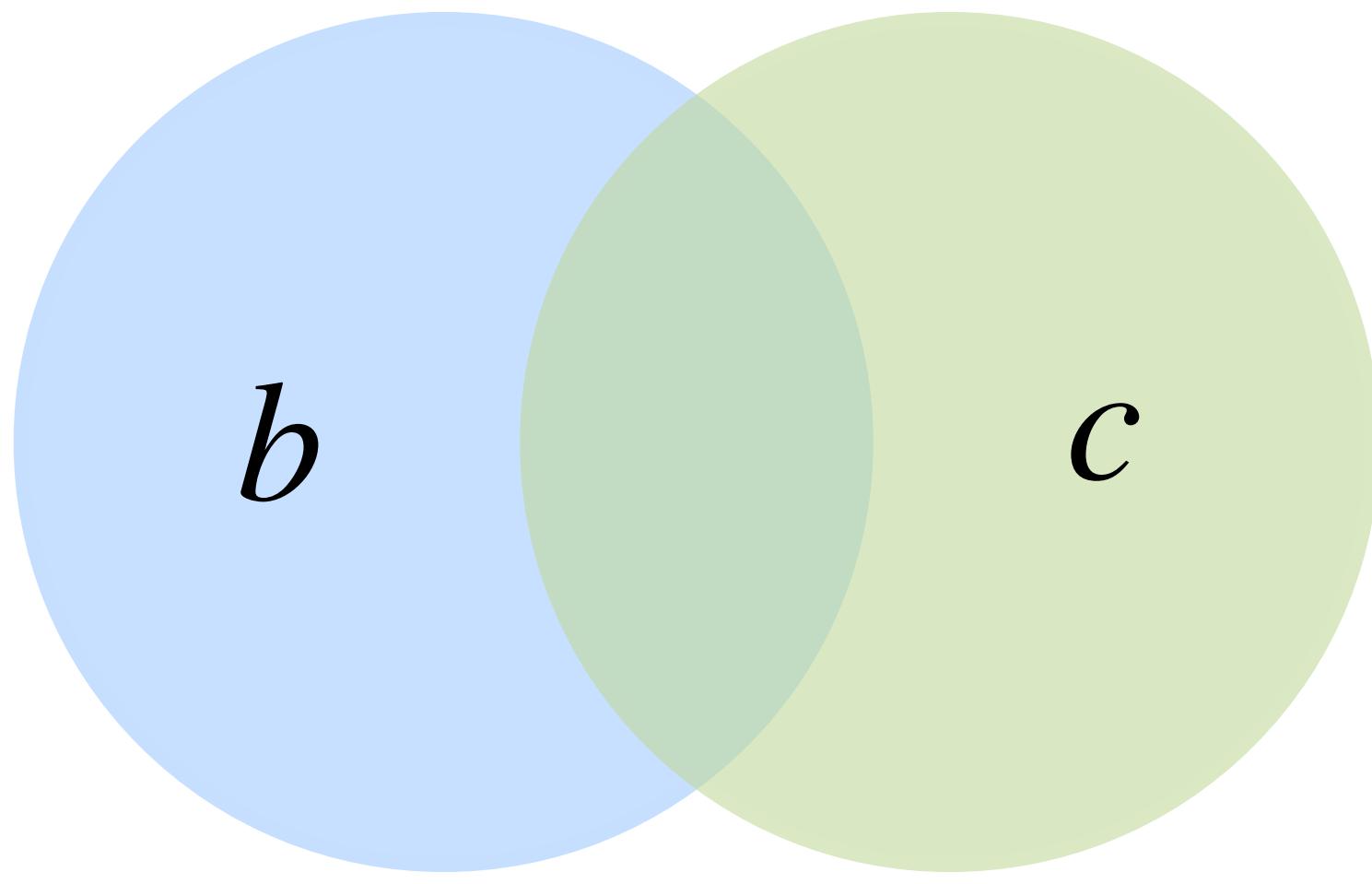
$$a_i = b_i + c_i$$



```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int i = min(ib, ic);
    if (ib == i && ic == i) {
        a[i] = b[pb1] + c[pc1];
    }
    else if (ib == i) {
        a[i] = b[pb1];
    }
    else {
        a[i] = c[pc1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}
```

Iteration lattice for additions

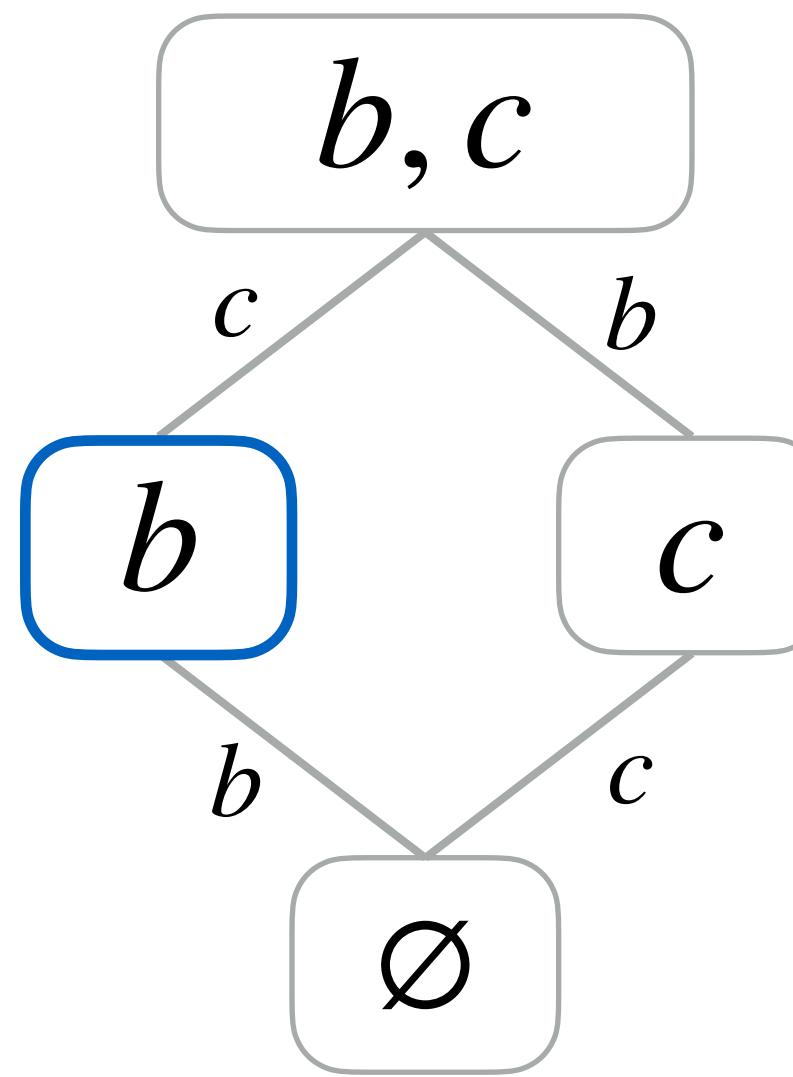
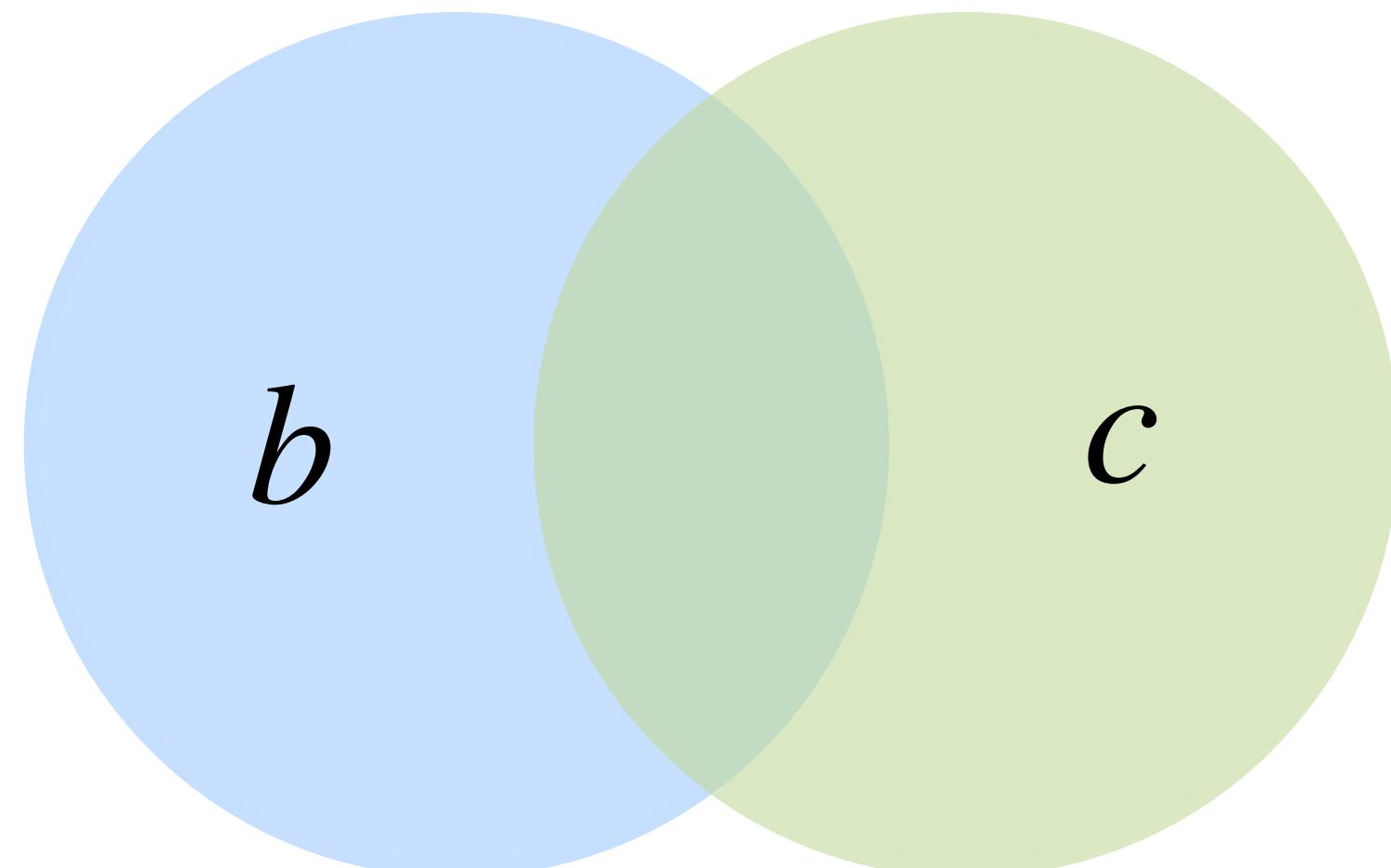
$$a_i = b_i + c_i$$



```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int i = min(ib, ic);
    if (ib == i && ic == i) {
        a[i] = b[pb1] + c[pc1];
    }
    else if (ib == i) {
        a[i] = b[pb1];
    }
    else {
        a[i] = c[pc1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}
```

Iteration lattice for additions

$$a_i = b_i + c_i$$

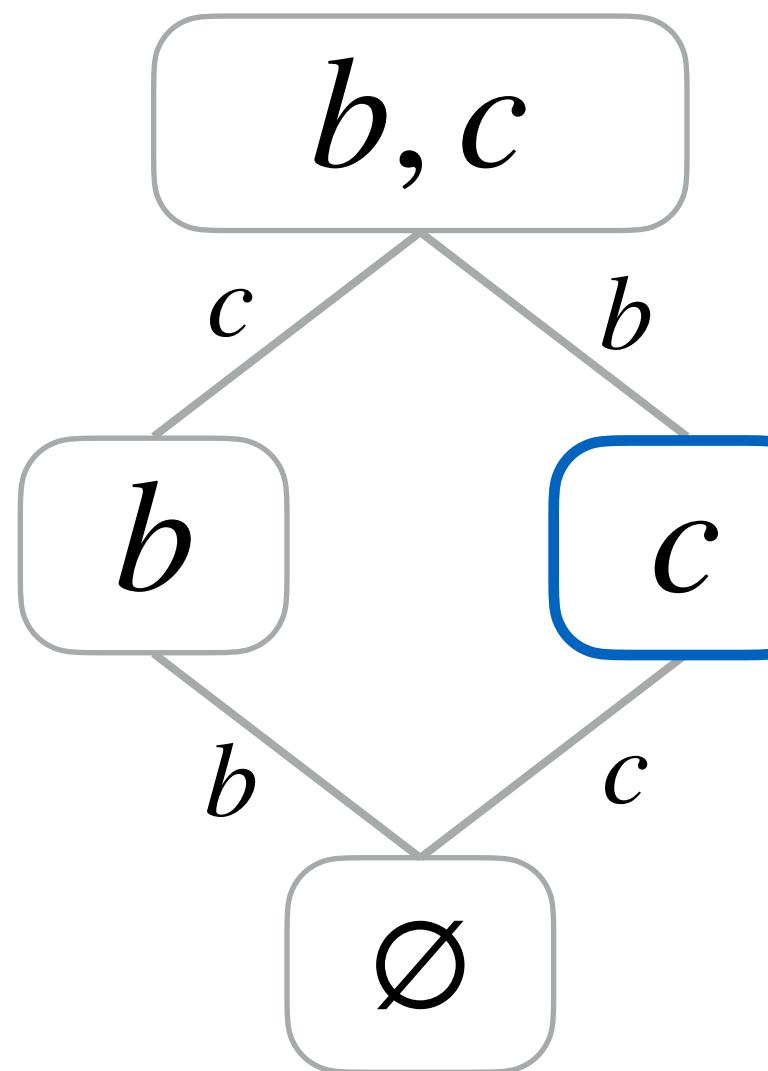
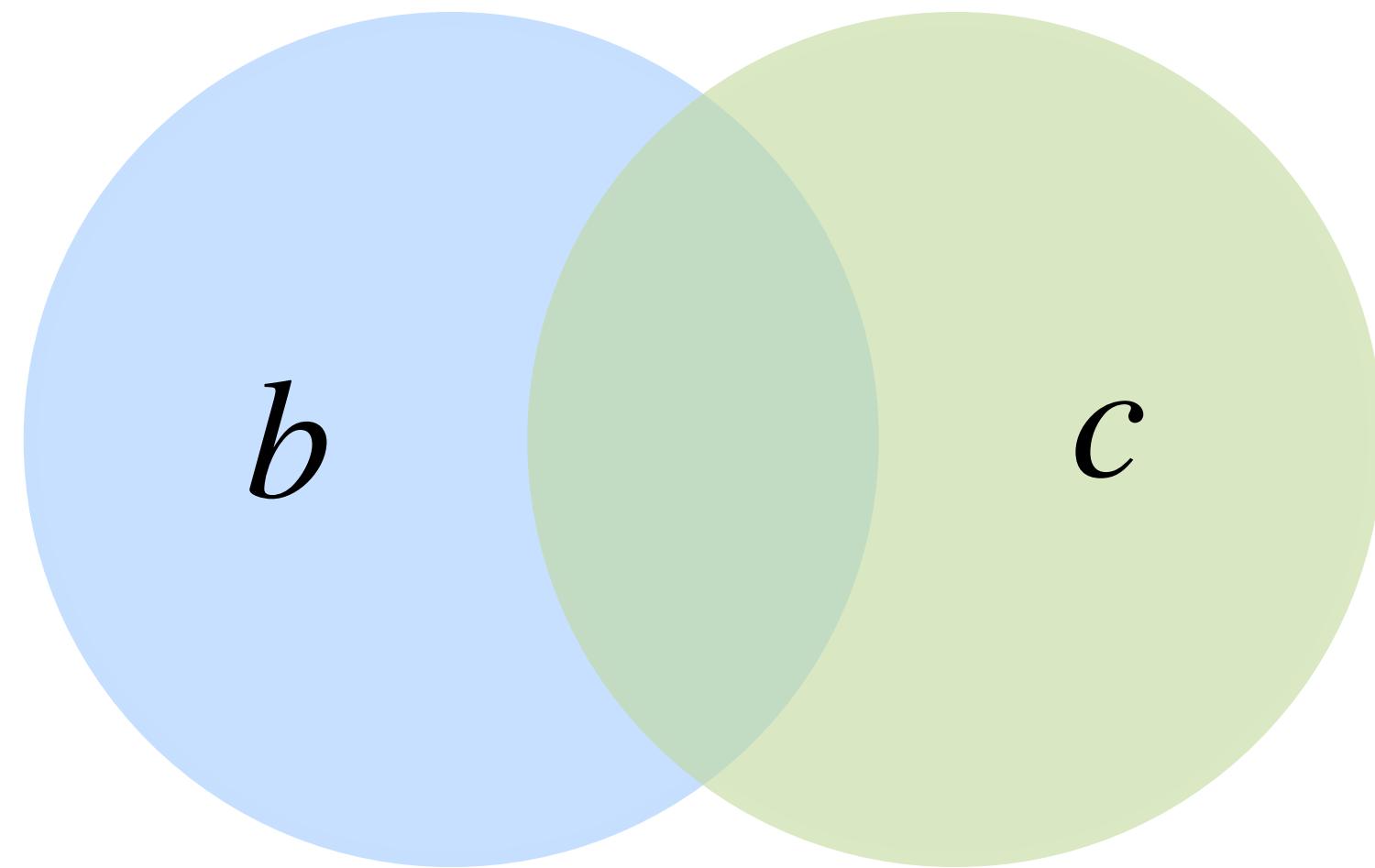


```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int i = min(ib, ic);
    if (ib == i && ic == i) {
        a[i] = b[pb1] + c[pc1];
    }
    else if (ib == i) {
        a[i] = b[pb1];
    }
    else {
        a[i] = c[pc1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}

while (pb1 < b1_pos[1]) {
    int i = b1_crd[pb1];
    a[i] = b[pb1++];
}
```

Iteration lattice for additions

$$a_i = b_i + c_i$$



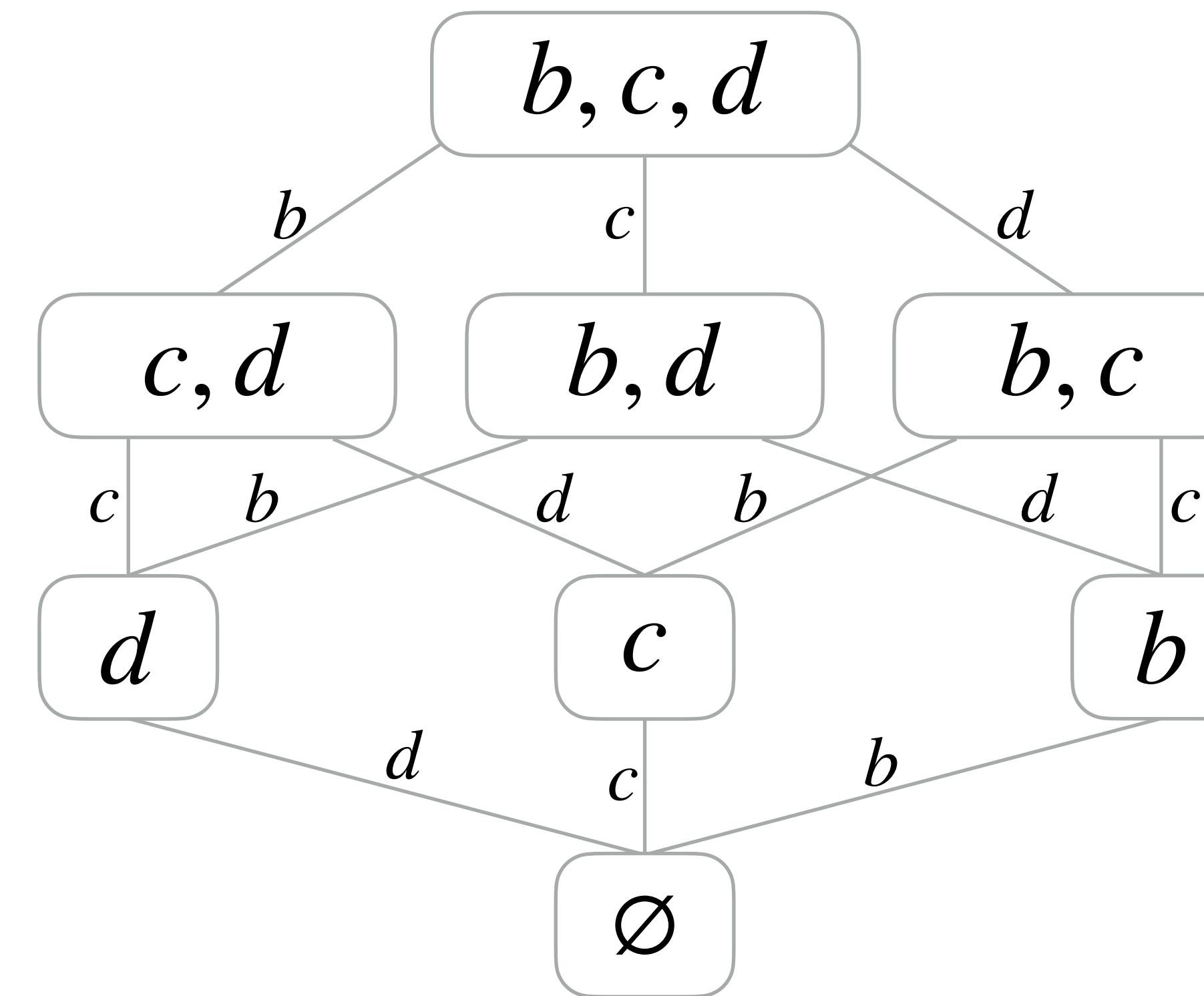
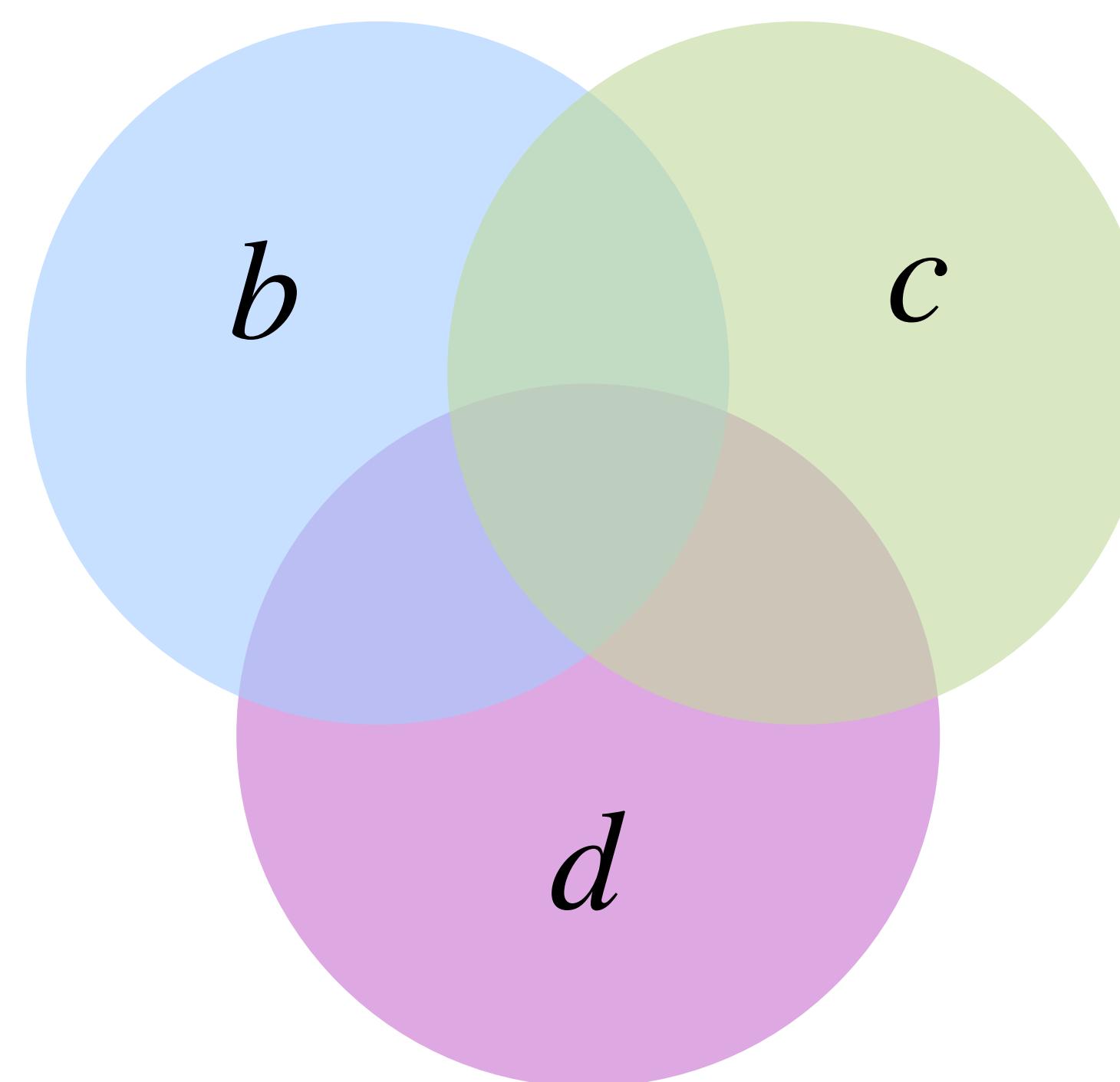
```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int i = min(ib, ic);
    if (ib == i && ic == i) {
        a[i] = b[pb1] + c[pc1];
    }
    else if (ib == i) {
        a[i] = b[pb1];
    }
    else {
        a[i] = c[pc1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}

while (pb1 < b1_pos[1]) {
    int i = b1_crd[pb1];
    a[i] = b[pb1++];
}

while (pc1 < c1_pos[1]) {
    int i = c1_crd[pc1];
    a[i] = c[pc1++];
}
```

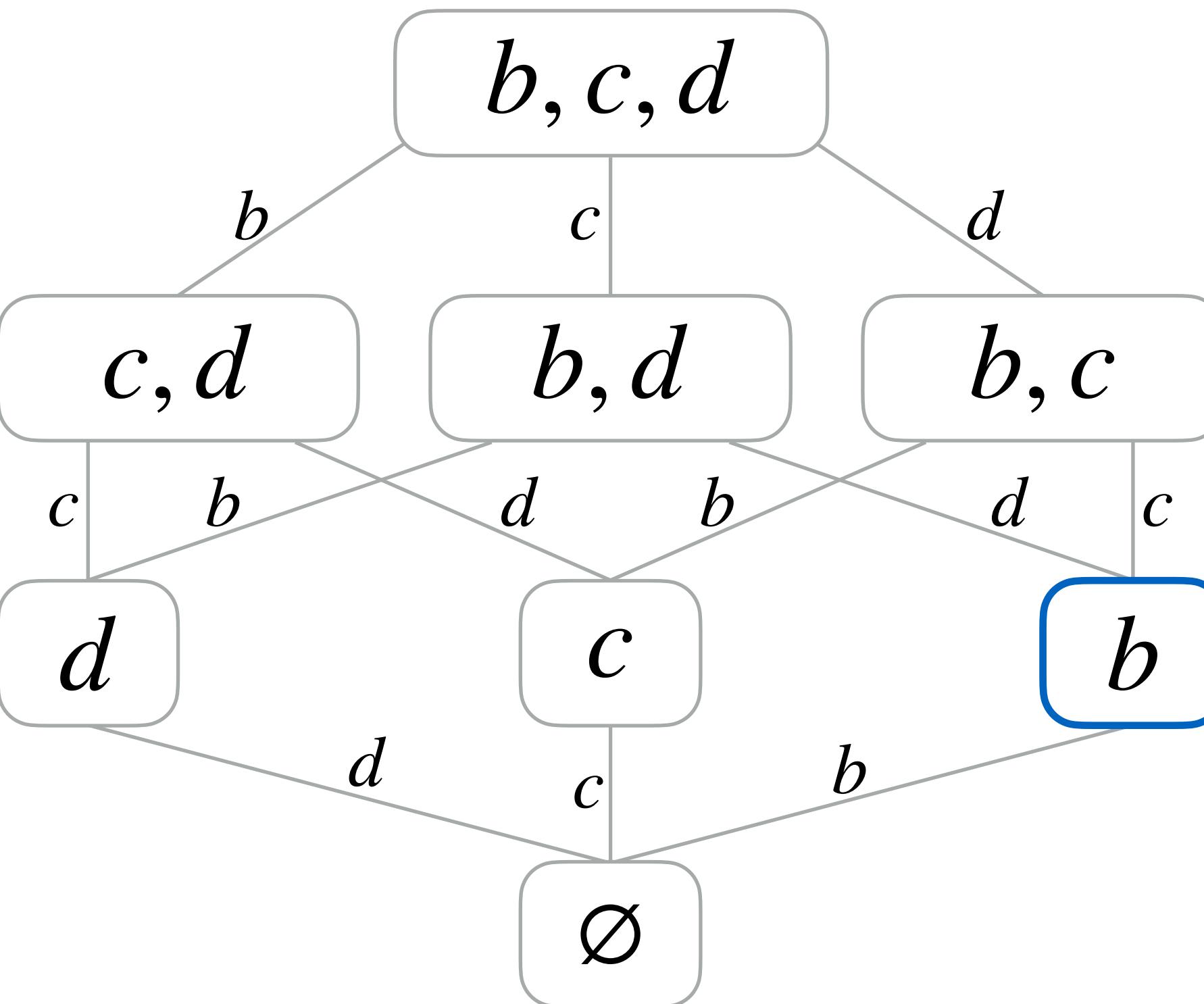
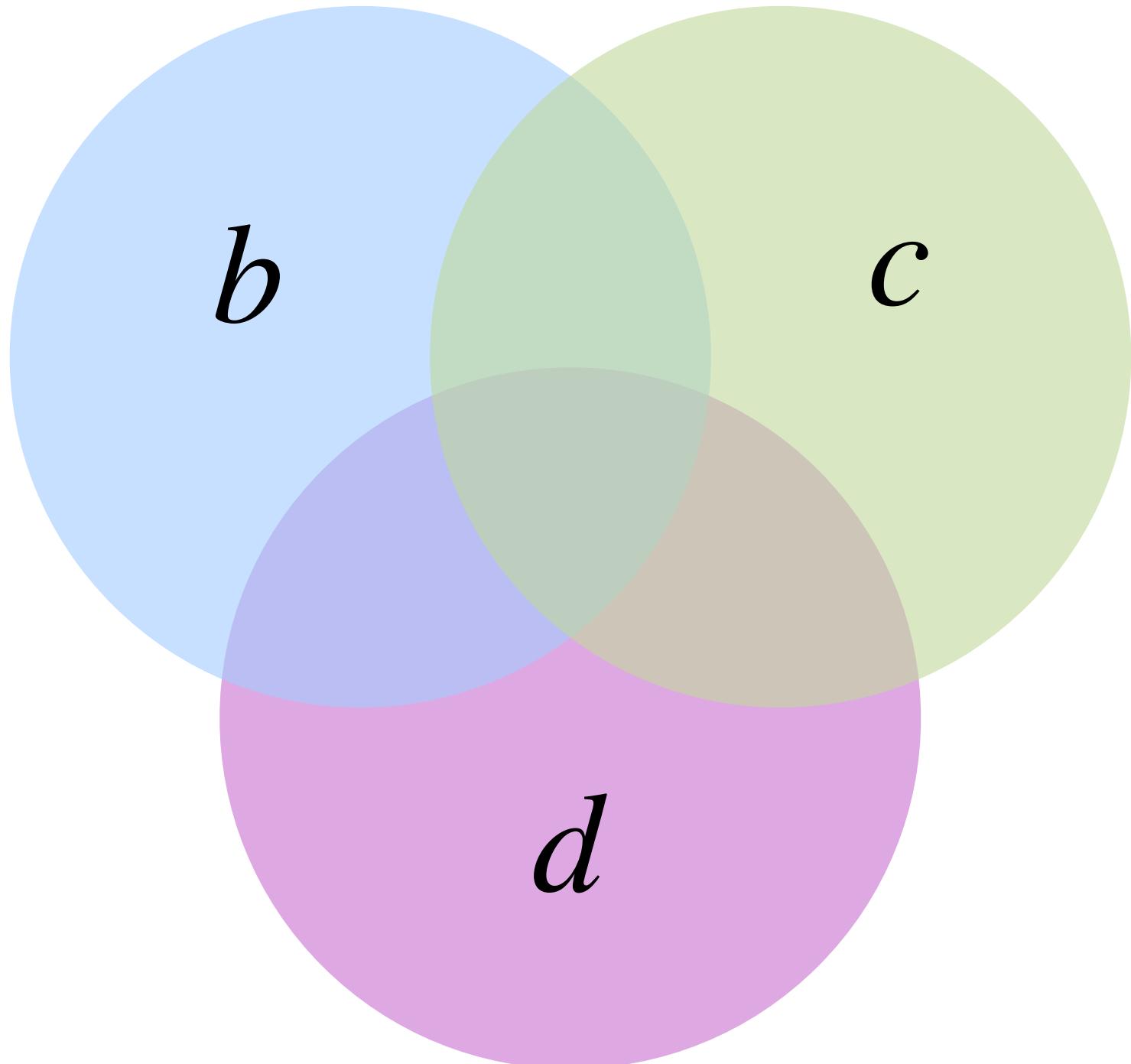
Iteration lattice for a compound expression

$$a_i = b_i + c_i + d_i$$



Iteration lattice for a compound expression

$$a_i = b_i + c_i + d_i$$



```

int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
int pd1 = d1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int id = d1_crd[pd1];
    int i = min(ib, ic, id);
    if (ic == i && id == i) {
        a[i] = c[pc1] + d[pd1];
    } else if (ic == i) {
        a[i] = c[pc1];
    } else {
        a[i] = d[pd1];
    }
    if (ic == i) pc1++;
    if (id == i) pd1++;
}

while (pb1 < b1_pos[1] && pd1 < d1_pos[1]) {
    int ib = b1_crd[pb1];
    int id = d1_crd[pd1];
    int i = min(ib, id);
    if (ib == i && id == i) {
        a[i] = b[pb1] + d[pd1];
    } else if (ib == i) {
        a[i] = b[pb1];
    } else if (id == i) {
        a[i] = d[pd1];
    }
    if (ib == i) pb1++;
    if (id == i) pd1++;
}

while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int i = min(ib, ic);
    if (ib == i && ic == i) {
        a[i] = b[pb1] + c[pc1];
    } else if (ib == i) {
        a[i] = b[pb1];
    } else {
        a[i] = c[pc1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}

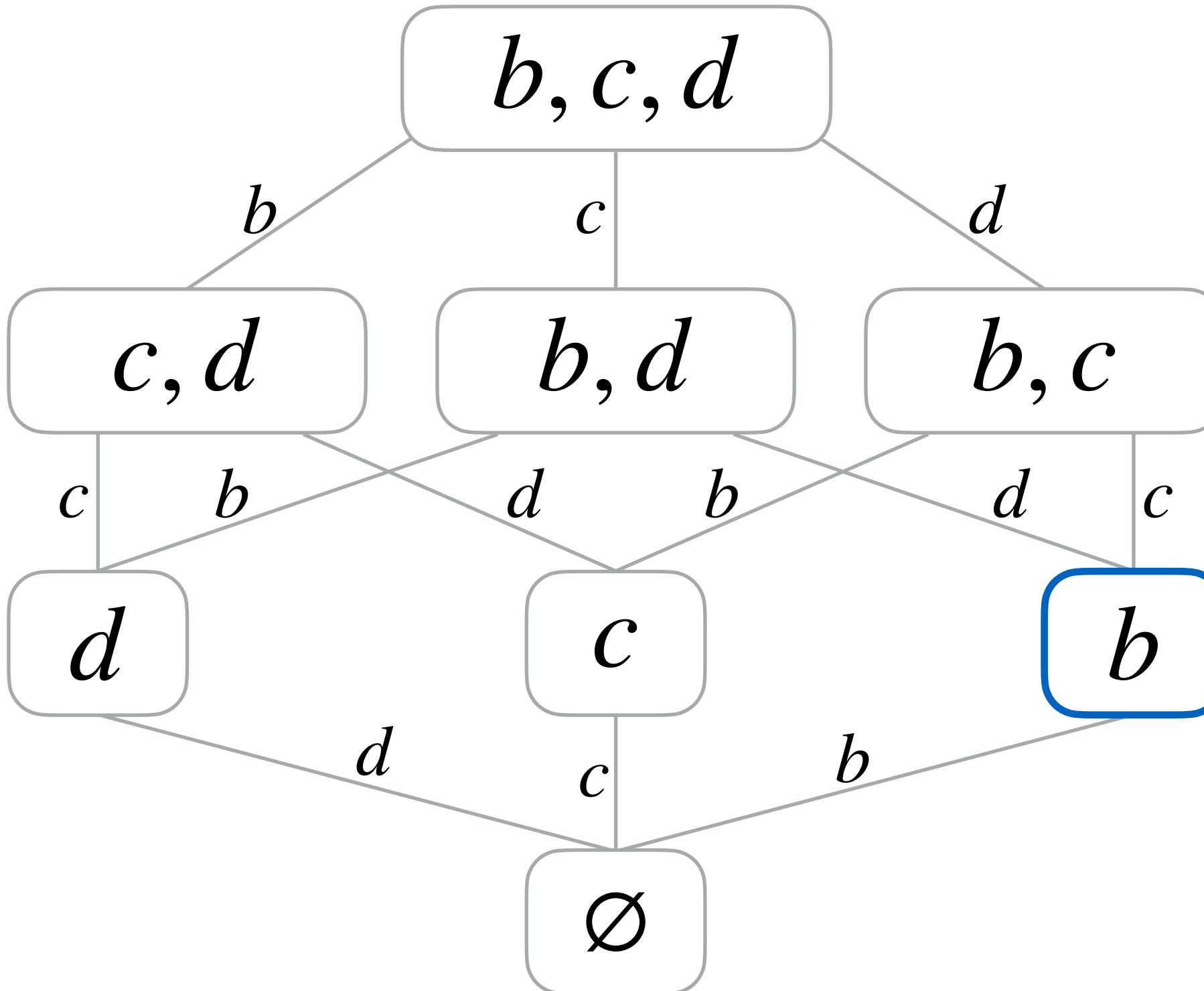
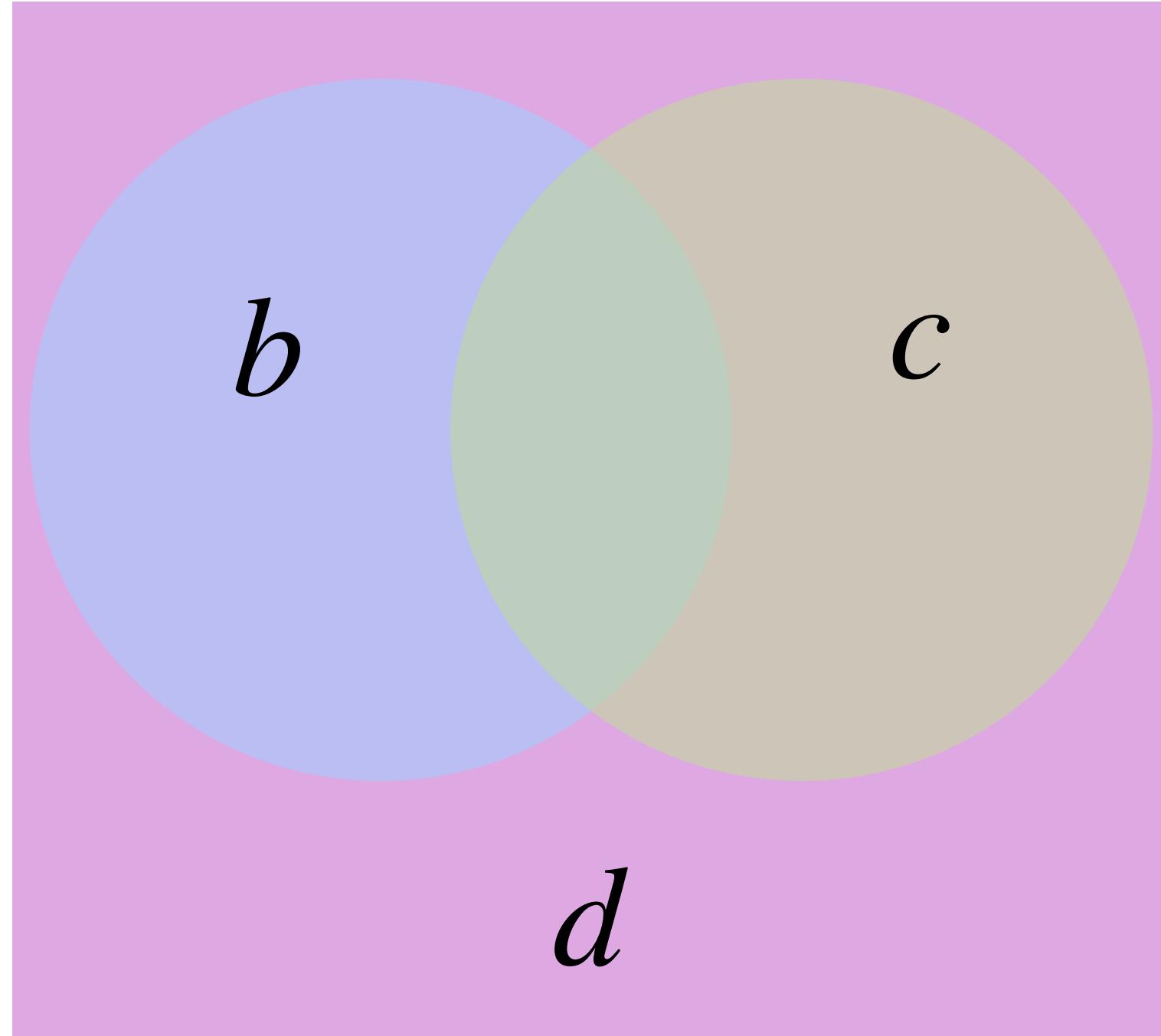
while (pd1 < d1_pos[1]) {
    int id = d1_crd[pd1];
    a[id] = d[pd1];
    pd1++;
}

while (pc1 < c1_pos[1]) {
    int ic = c1_crd[pc1];
    a[ic] = c[pc1];
    pc1++;
}

while (pb1 < b1_pos[1]) {
    int ib = b1_crd[pb1];
    a[ib] = b[pb1];
    pb1++;
}
  
```

Iteration lattice for a compound expression

$$a_i = b_i + c_i + d_i \quad \text{Dense}$$



```

int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
int pd1 = d1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int id = d1_crd[pd1];
    int i = min(ib, ic, id);
    if (ic == i && id == i) {
        a[i] = c[pc1] + d[pd1];
    } else if (ic == i) {
        a[i] = c[pc1];
    } else {
        a[i] = d[pd1];
    }
    if (ic == i) pc1++;
    if (id == i) pd1++;
}

while (pb1 < b1_pos[1] && pd1 < d1_pos[1]) {
    int ib = b1_crd[pb1];
    int id = d1_crd[pd1];
    int i = min(ib, id);
    if (ib == i && id == i) {
        a[i] = b[pb1] + d[pd1];
    } else if (ib == i) {
        a[i] = b[pb1];
    } else {
        a[i] = d[pd1];
    }
    if (ib == i) pb1++;
    if (id == i) pd1++;
}

while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int i = min(ib, ic);
    if (ib == i && ic == i) {
        a[i] = b[pb1] + c[pc1];
    } else if (ib == i) {
        a[i] = b[pb1];
    } else {
        a[i] = c[pc1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}

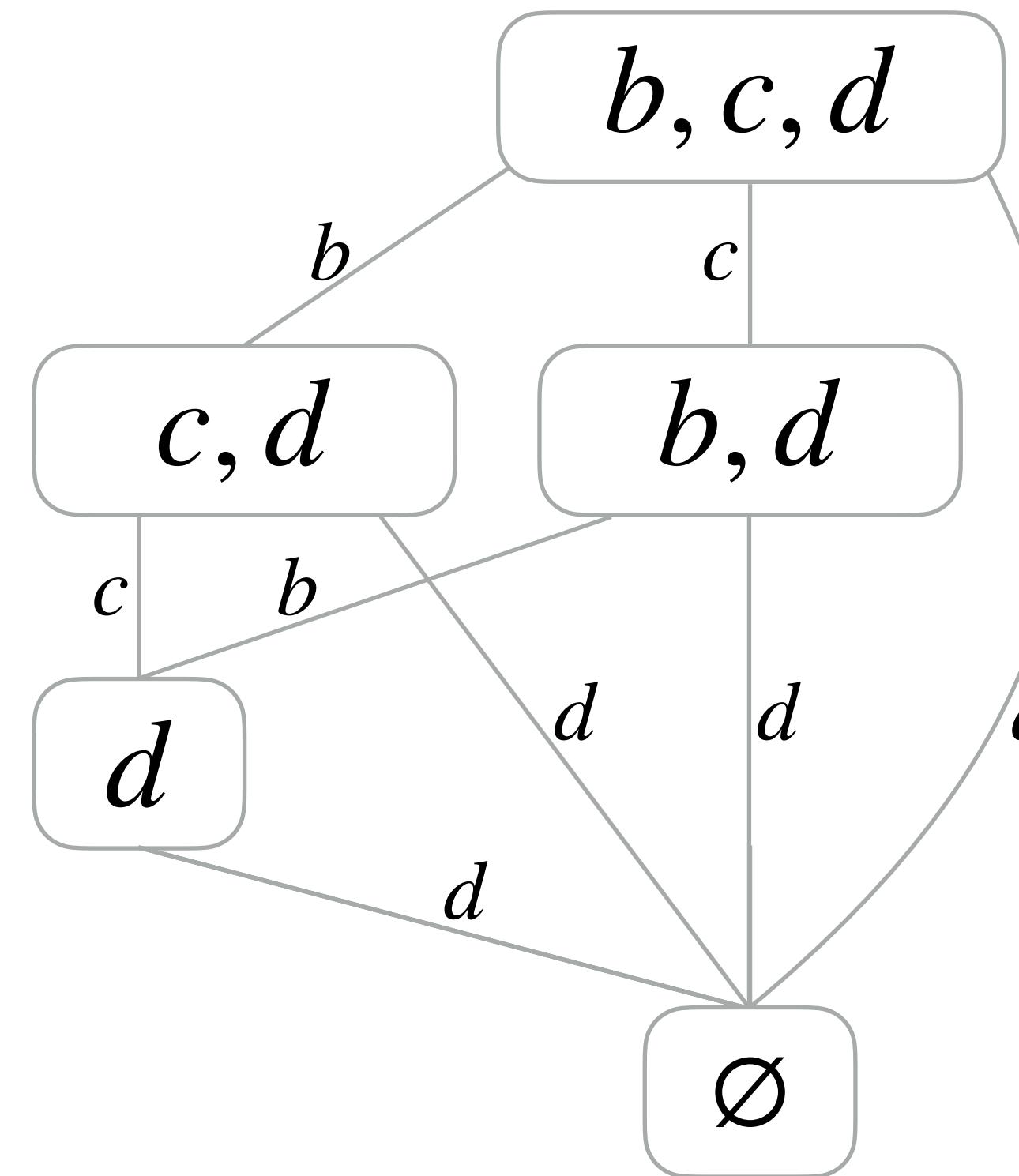
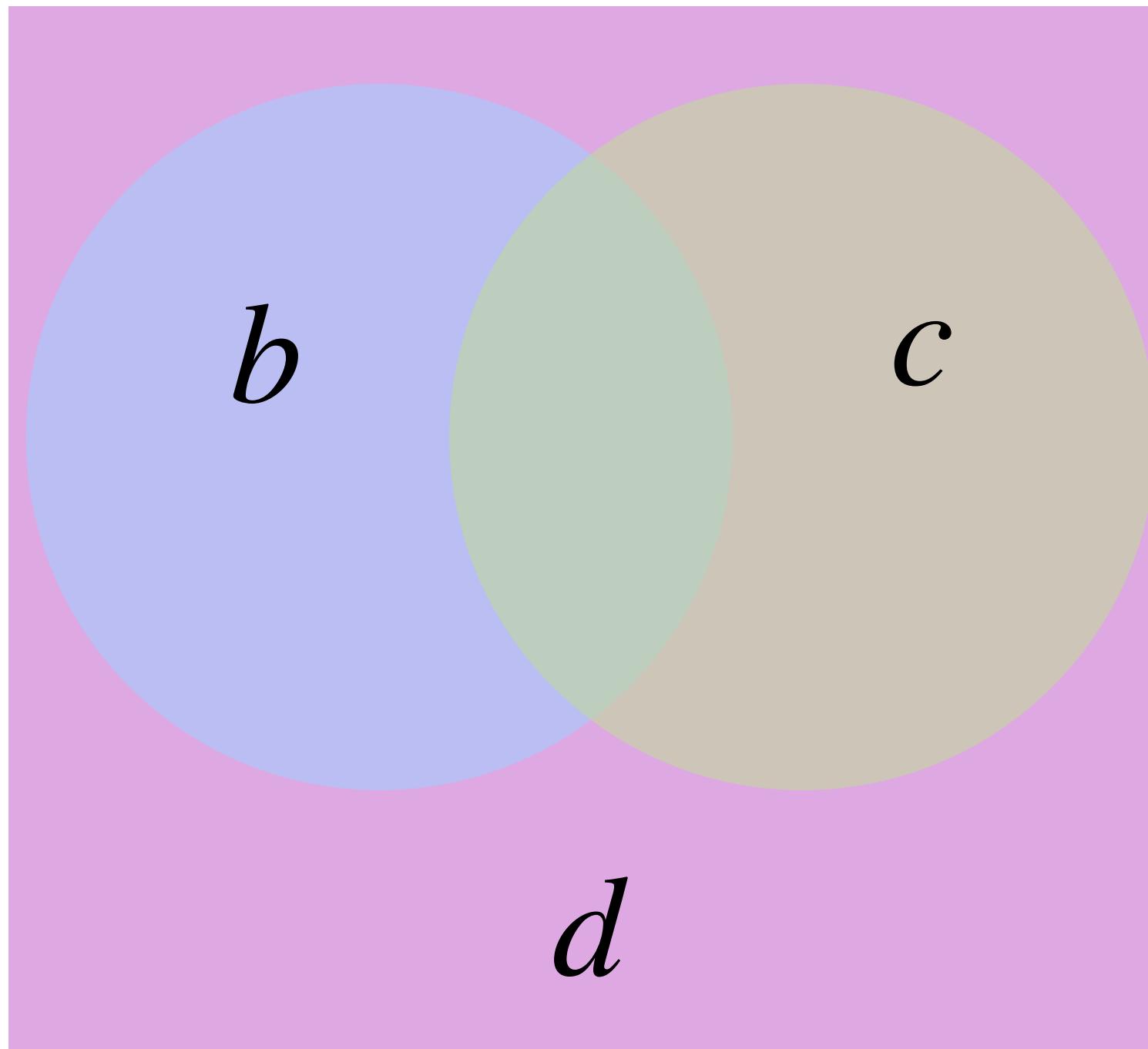
while (pd1 < d1_pos[1]) {
    int id = d1_crd[pd1];
    a[id] = d[pd1];
    pd1++;
}

while (pc1 < c1_pos[1]) {
    int ic = c1_crd[pc1];
    a[ic] = c[pc1];
    pc1++;
}

while (pb1 < b1_pos[1]) {
    int ib = b1_crd[pb1];
    a[ib] = b[pb1];
    pb1++;
}
  
```

Iteration lattice for a compound expression

$$a_i = b_i + c_i + d_i \quad \text{Dense}$$



```

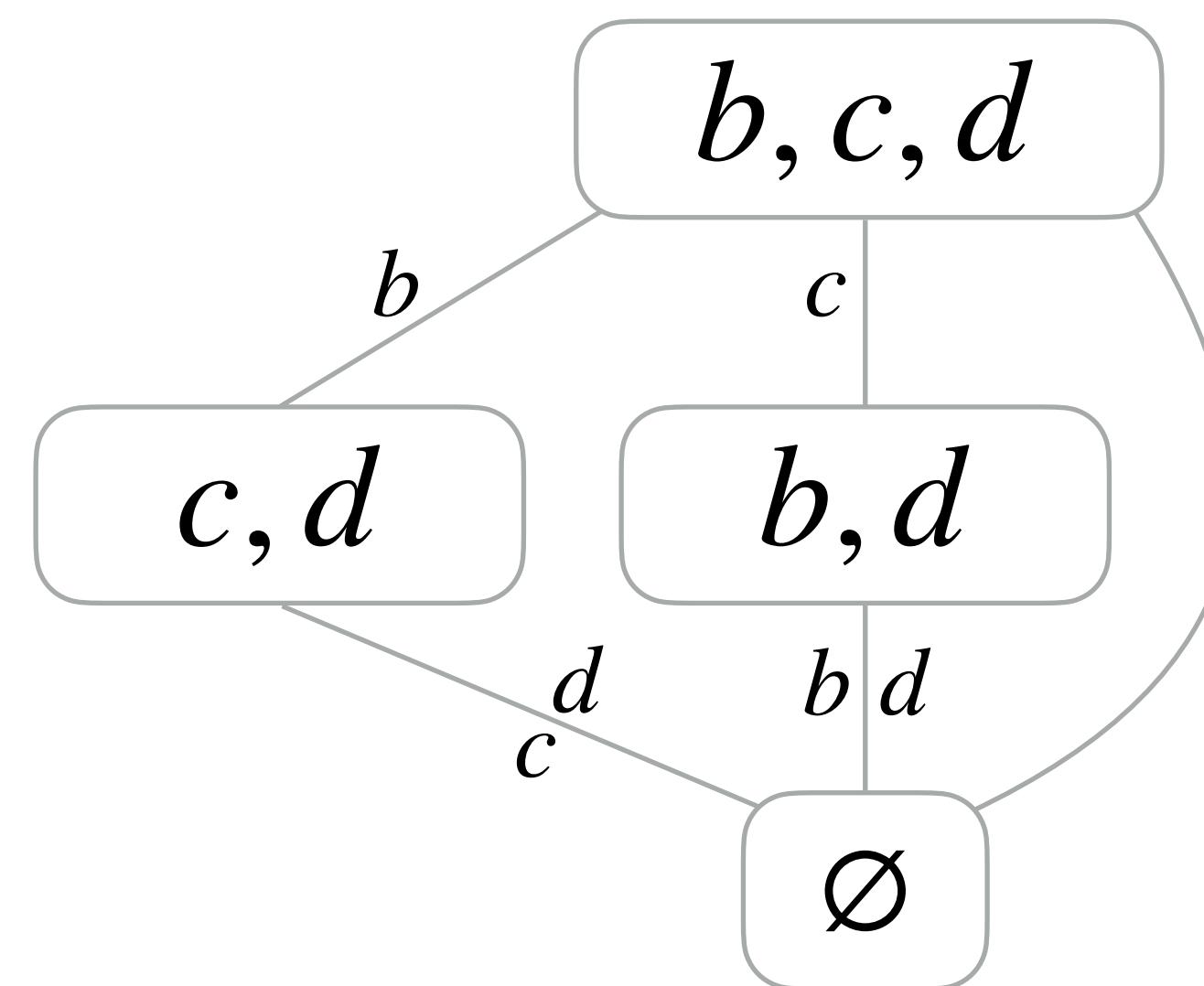
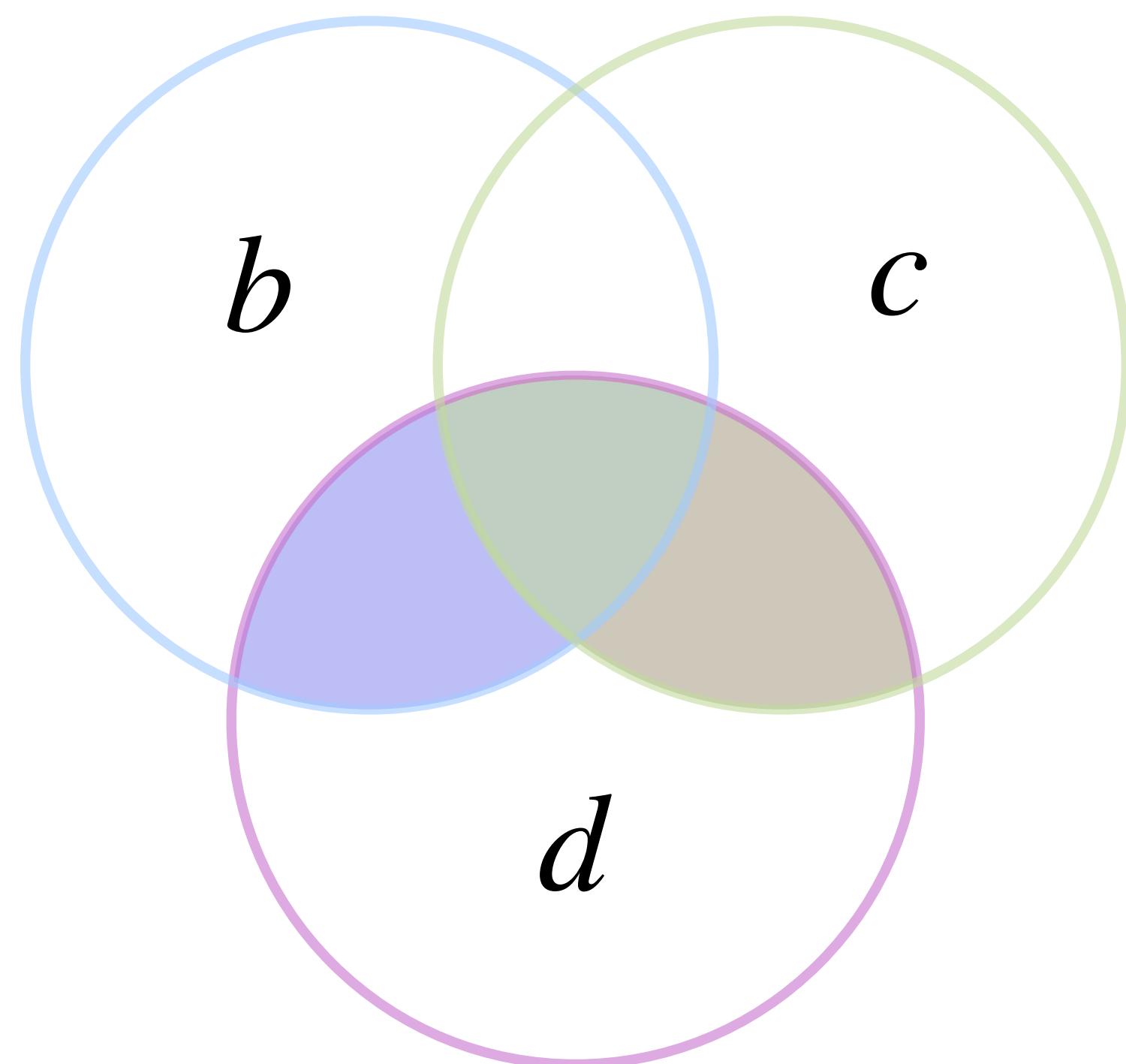
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
int id = 0;
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int pd1 = id;
    int pa1 = id;
    if (ic == id) {
        a[pa1] = c[pc1] + d[pd1];
    } else {
        a[pa1] = d[pd1];
    }
    if (ib == id && ic == id) {
        a[pa1] = b[pb1] + c[pc1] + d[pd1];
    } else if (ib == id) {
        a[pa1] = b[pb1] + d[pd1];
    } else if (ic == id) {
        a[pa1] = c[pc1] + d[pd1];
    } else {
        a[pa1] = d[pd1];
    }
    if (ib == id) pb1++;
    if (ic == id) pc1++;
    id++;
}

while (pb1 < b1_pos[1]) {
    int ib = b1_crd[pb1];
    int pd1 = id;
    int pa1 = id;
    if (ib == id) {
        a[pa1] = b[pb1] + d[pd1];
    } else {
        a[pa1] = d[pd1];
    }
    if (ib == id) pb1++;
    id++;
}

while (id < d1_dimension) {
    int pd1 = id;
    int pa1 = id;
    a[pa1] = d[pd1];
    id++;
}
  
```

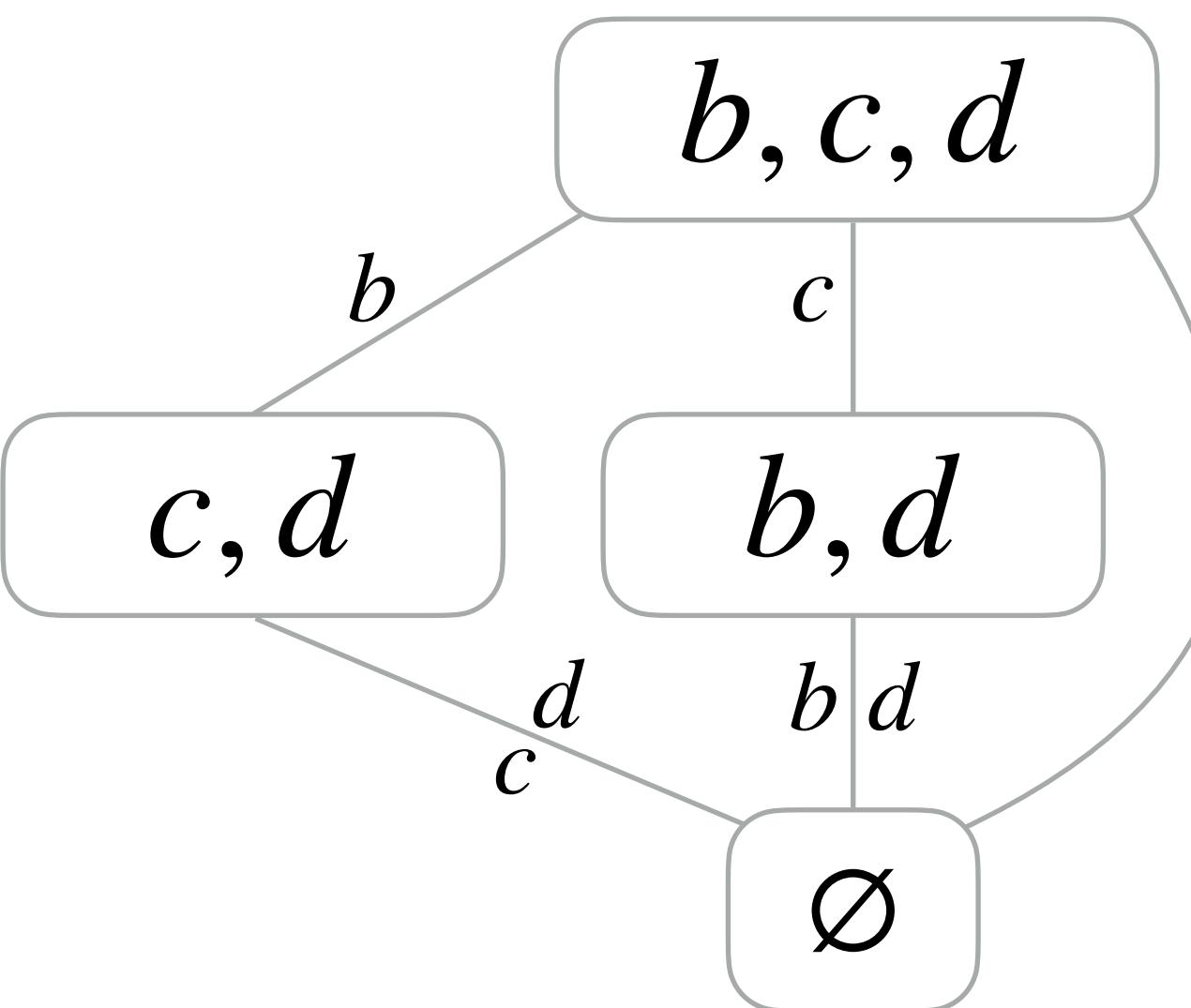
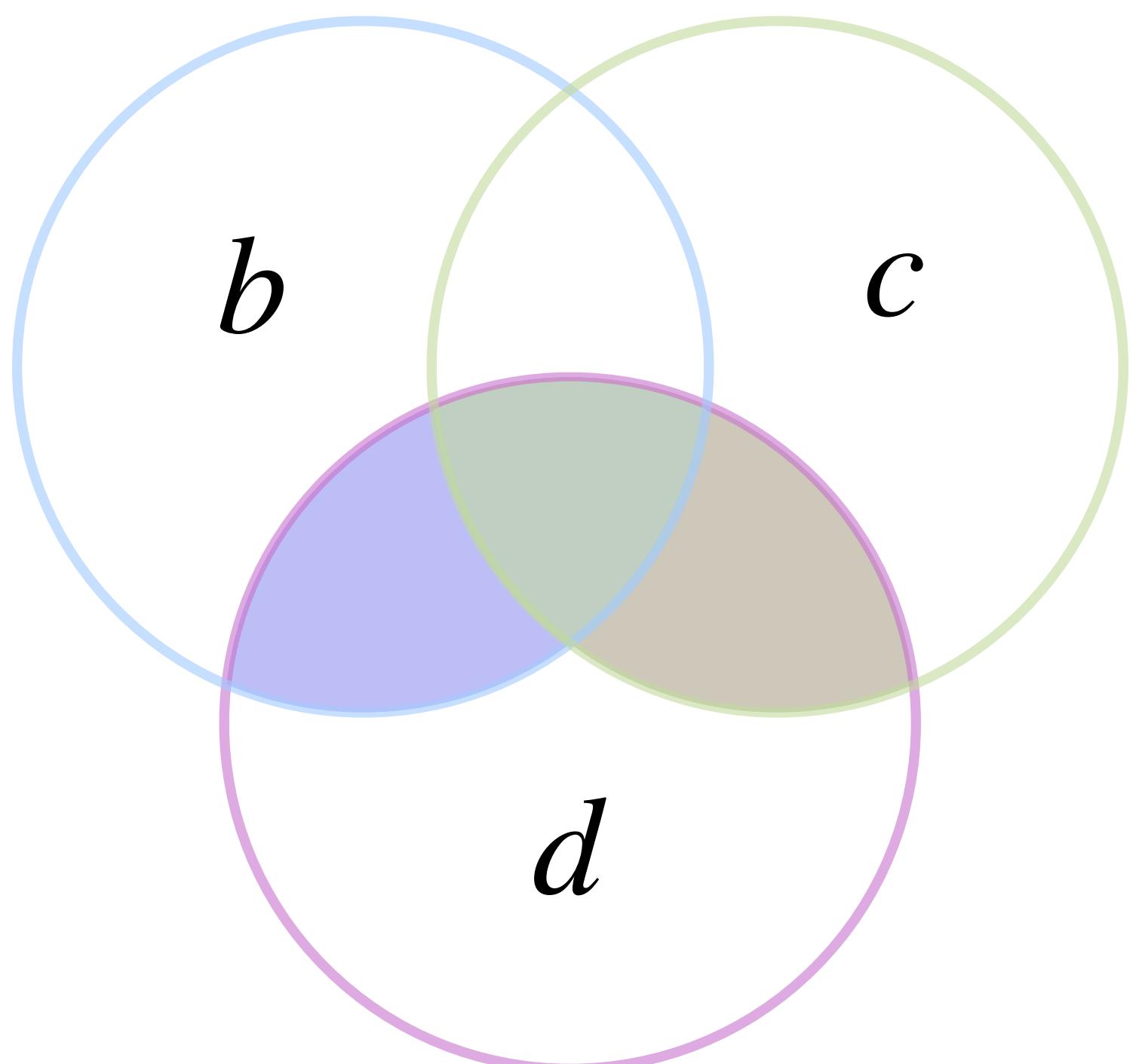
Iteration lattice for a compound expression

$$a_i = (b_i + c_i)d_i$$



Iteration lattice for a compound expression

$$a_i = (b_i + c_i)d_i$$



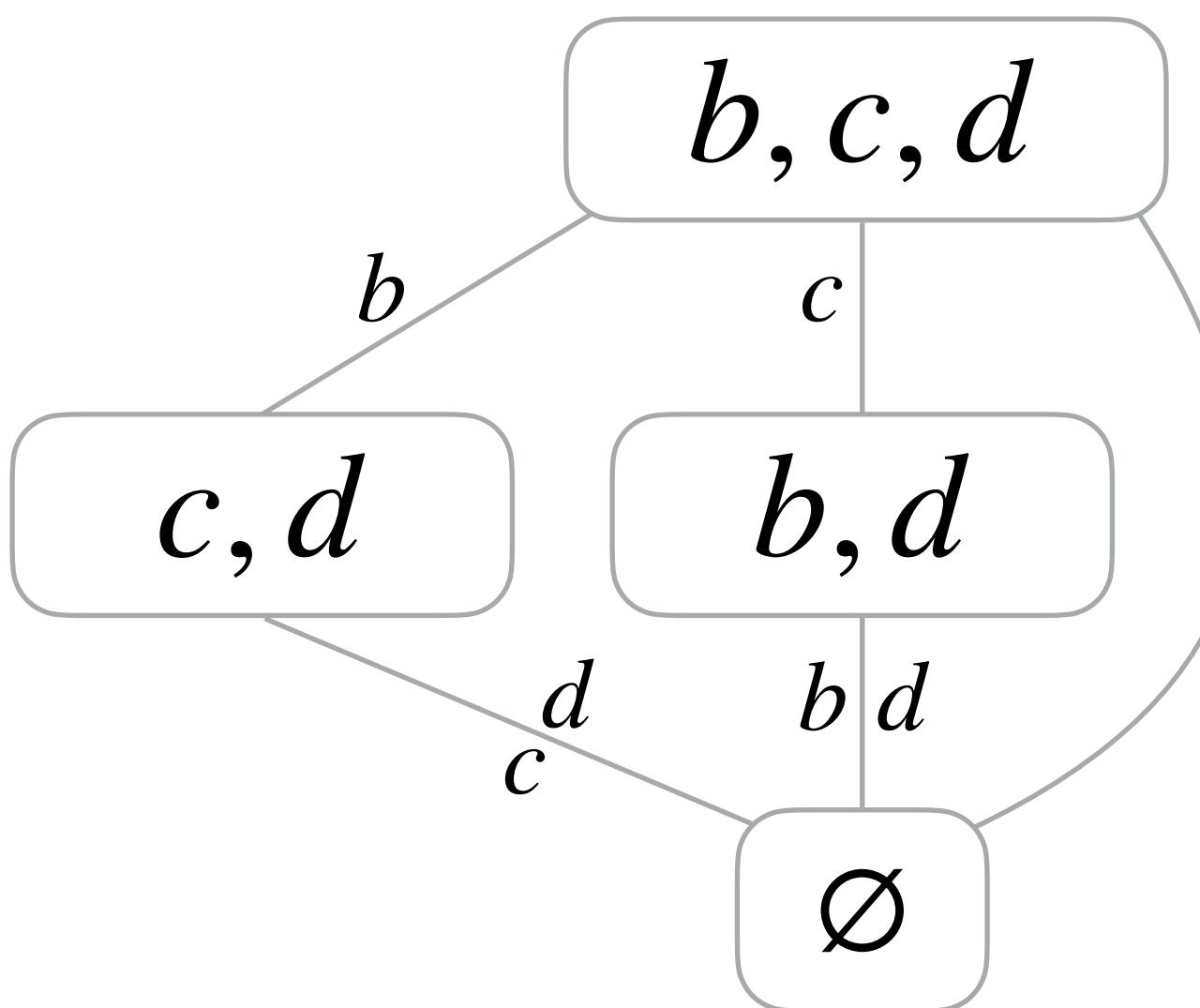
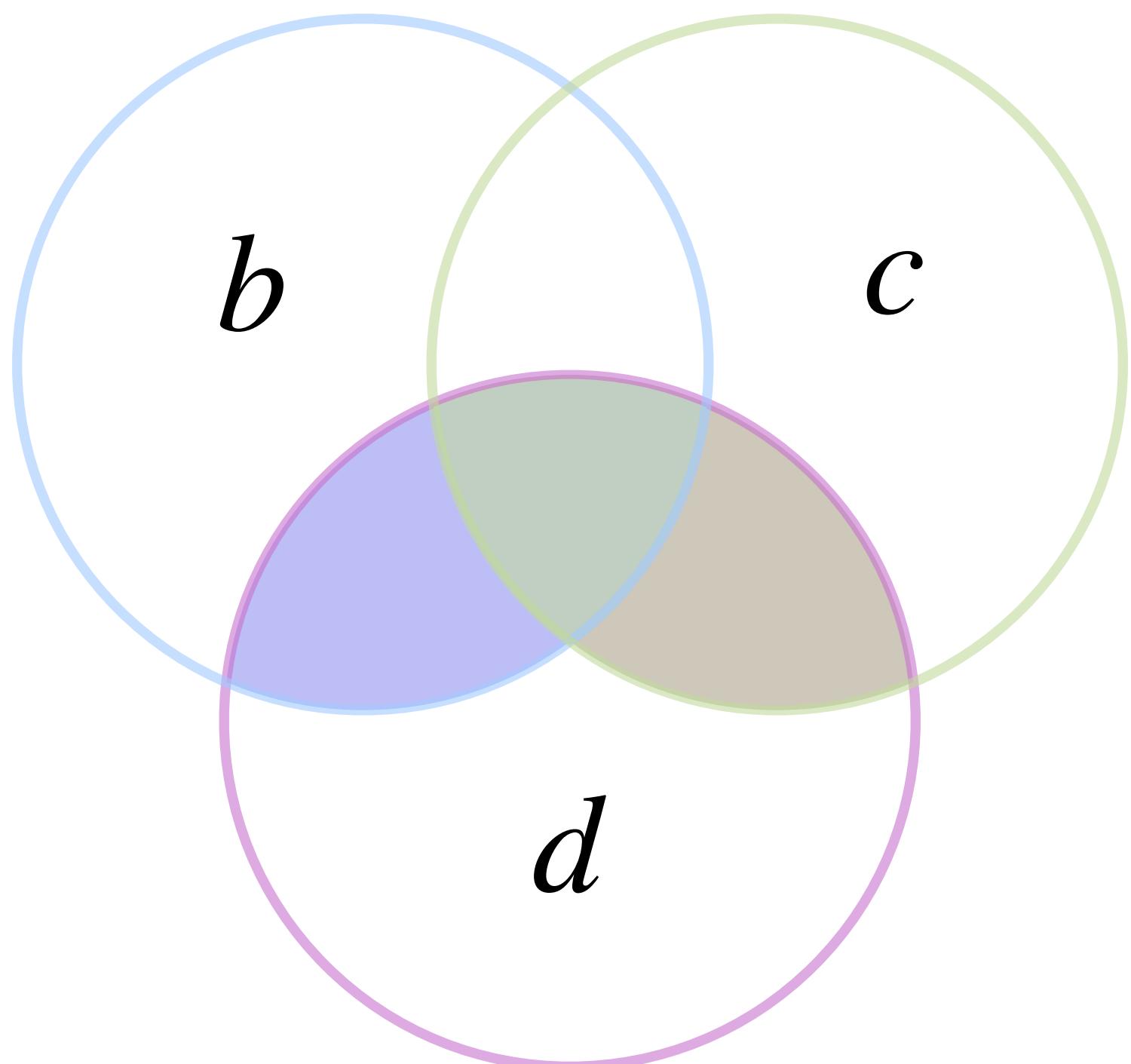
```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
int pd1 = d1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int id = d1_crd[pd1];
    int i = min(ib, ic, id);
    if (ib == i && ic == i && id == i) {
        a[i] = (b[pb1] + c[pc1]) * d[pd1];
    } else if (ib == i && id == i) {
        a[i] = b[pb1] * d[pd1];
    } else if (ic == i && id == i) {
        a[i] = c[pc1] * d[pd1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
    if (id == i) pd1++;
}

while (pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
    int ic = c1_crd[pc1];
    int id = d1_crd[pd1];
    int i = min(ic, id);
    if (ic == i && id == i) {
        a[i] = c[pc1] * d[pd1];
    }
    if (ic == i) pc1++;
    if (id == i) pd1++;
}

while (pb1 < b1_pos[1] && pd1 < d1_pos[1])
{
    int ib = b1_crd[pb1];
    int id = d1_crd[pd1];
    int i = min(ib, id);
    if (ib == i && id == i) {
        a[i] = b[pb1] * d[pd1];
    }
    if (ib == i) pb1++;
    if (id == i) pd1++;
}
```

Iteration lattice for a compound expression

$$a_i = (b_i + c_i)d_i \quad \text{Dense}$$



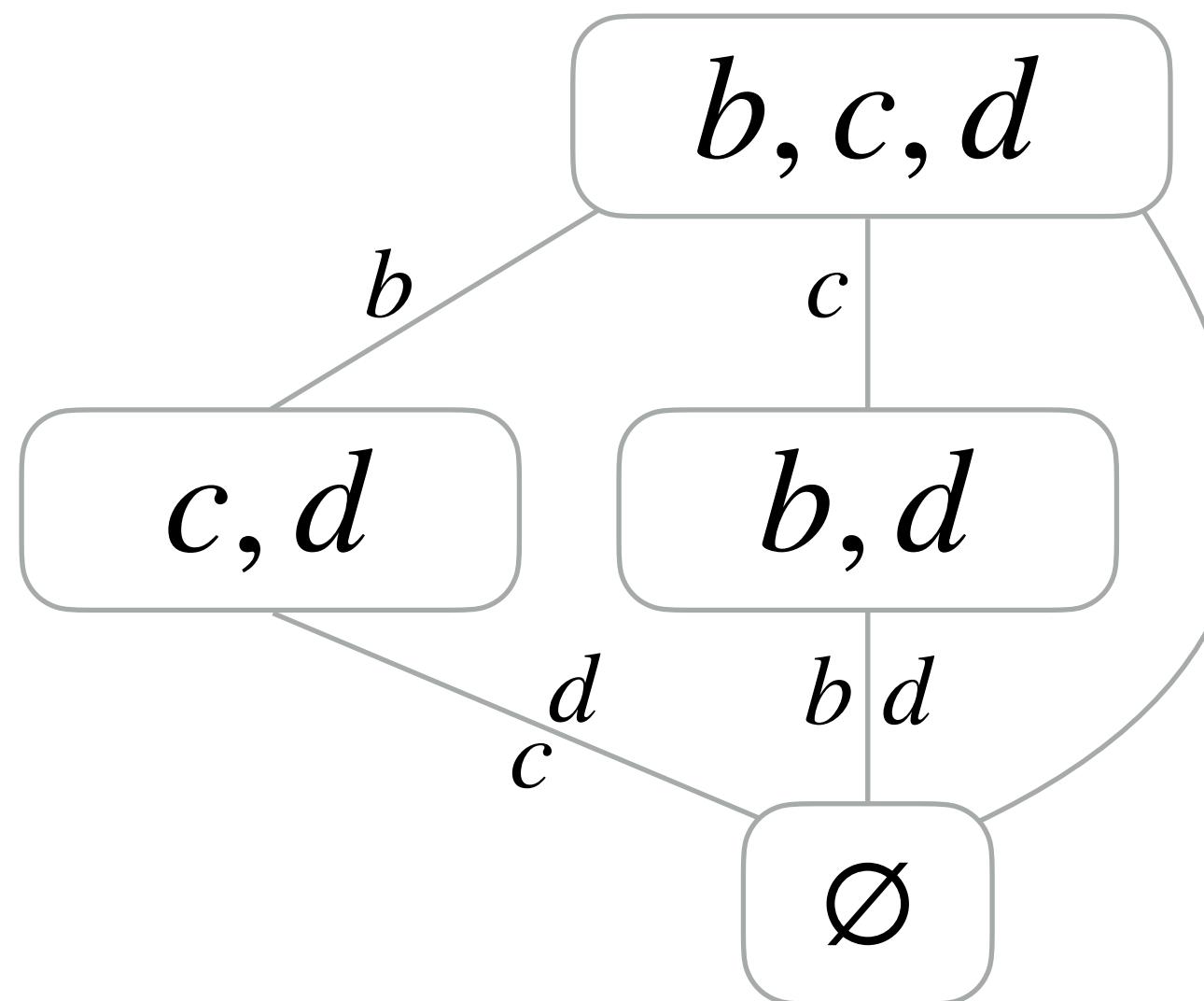
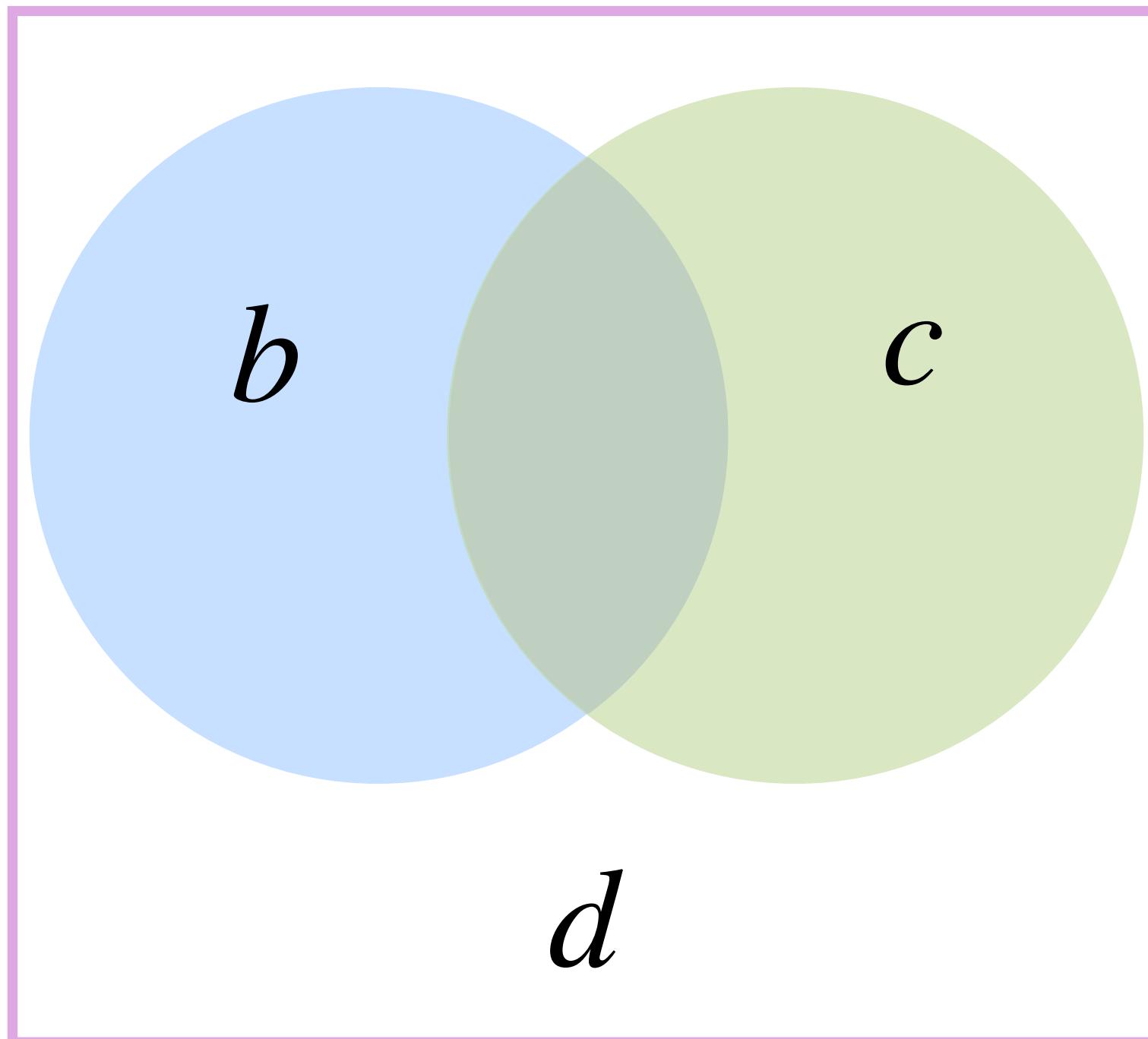
```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
int pd1 = d1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int id = d1_crd[pd1];
    int i = min(ib, ic, id);
    if (ib == i && ic == i && id == i) {
        a[i] = (b[pb1] + c[pc1]) * d[pd1];
    } else if (ib == i && id == i) {
        a[i] = b[pb1] * d[pd1];
    } else if (ic == i && id == i) {
        a[i] = c[pc1] * d[pd1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
    if (id == i) pd1++;
}

while (pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
    int ic = c1_crd[pc1];
    int id = d1_crd[pd1];
    int i = min(ic, id);
    if (ic == i && id == i) {
        a[i] = c[pc1] * d[pd1];
    }
    if (ic == i) pc1++;
    if (id == i) pd1++;
}

while (pb1 < b1_pos[1] && pd1 < d1_pos[1])
{
    int ib = b1_crd[pb1];
    int id = d1_crd[pd1];
    int i = min(ib, id);
    if (ib == i && id == i) {
        a[i] = b[pb1] * d[pd1];
    }
    if (ib == i) pb1++;
    if (id == i) pd1++;
}
```

Iteration lattice for a compound expression

$$a_i = (b_i + c_i)d_i \quad \text{Dense}$$



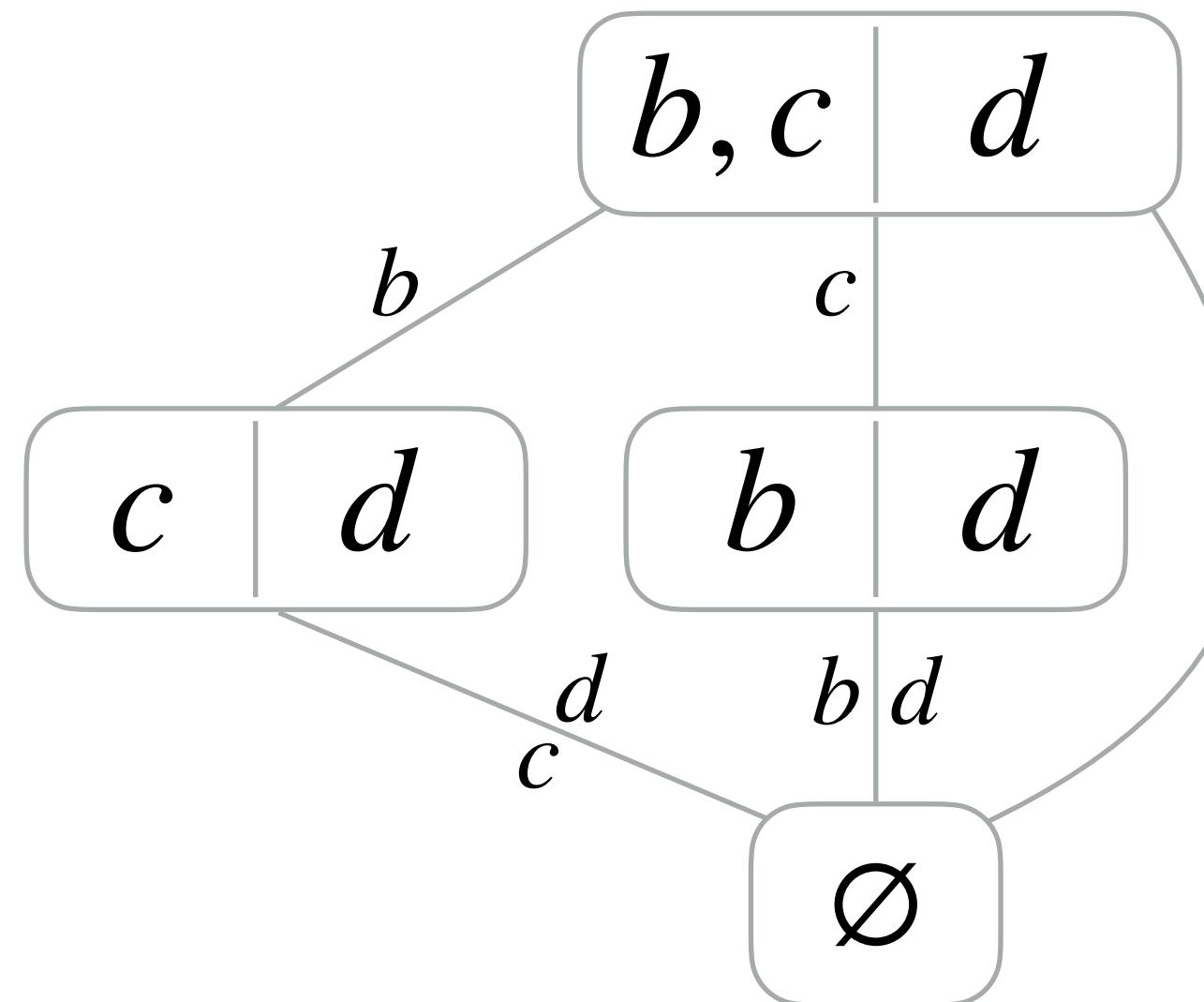
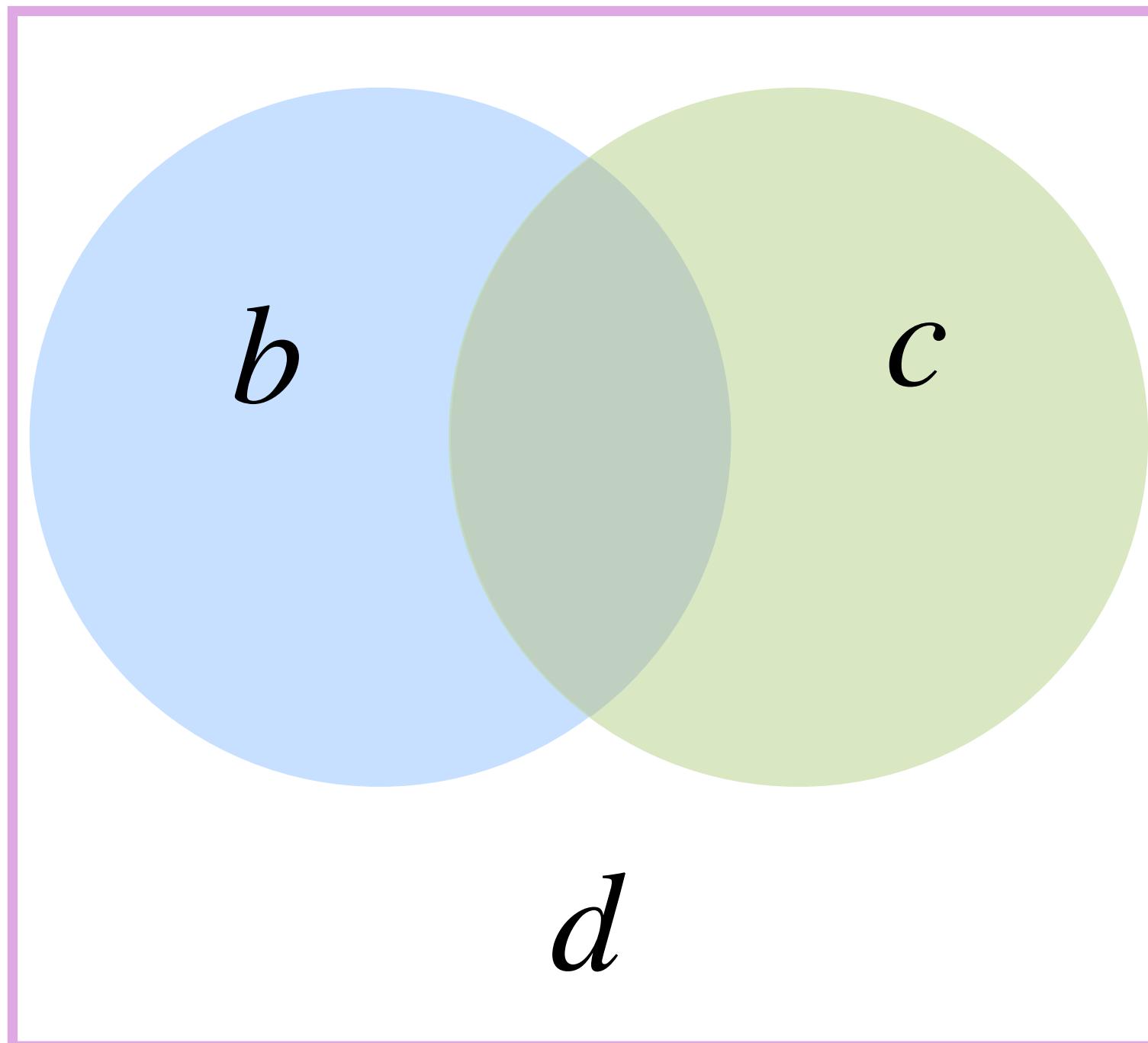
```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
int pd1 = d1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int id = d1_crd[pd1];
    int i = min(ib, ic, id);
    if (ib == i && ic == i && id == i) {
        a[i] = (b[pb1] + c[pc1]) * d[pd1];
    } else if (ib == i && id == i) {
        a[i] = b[pb1] * d[pd1];
    } else if (ic == i && id == i) {
        a[i] = c[pc1] * d[pd1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
    if (id == i) pd1++;
}

while (pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
    int ic = c1_crd[pc1];
    int id = d1_crd[pd1];
    int i = min(ic, id);
    if (ic == i && id == i) {
        a[i] = c[pc1] * d[pd1];
    }
    if (ic == i) pc1++;
    if (id == i) pd1++;
}

while (pb1 < b1_pos[1] && pd1 < d1_pos[1])
{
    int ib = b1_crd[pb1];
    int id = d1_crd[pd1];
    int i = min(ib, id);
    if (ib == i && id == i) {
        a[i] = b[pb1] * d[pd1];
    }
    if (ib == i) pb1++;
    if (id == i) pd1++;
}
```

Iteration lattice for a compound expression

$$a_i = (b_i + c_i)d_i \quad \text{Dense}$$



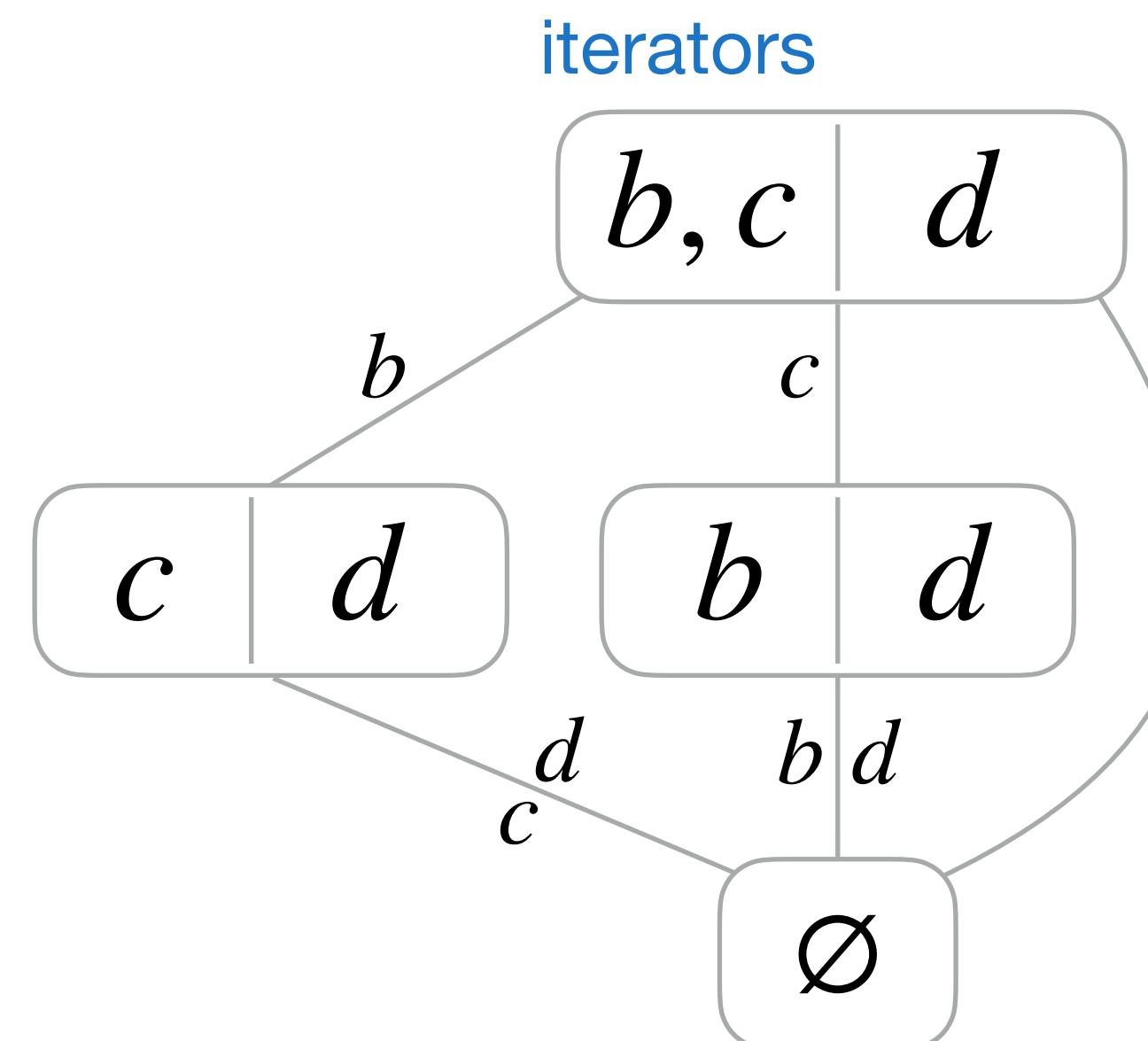
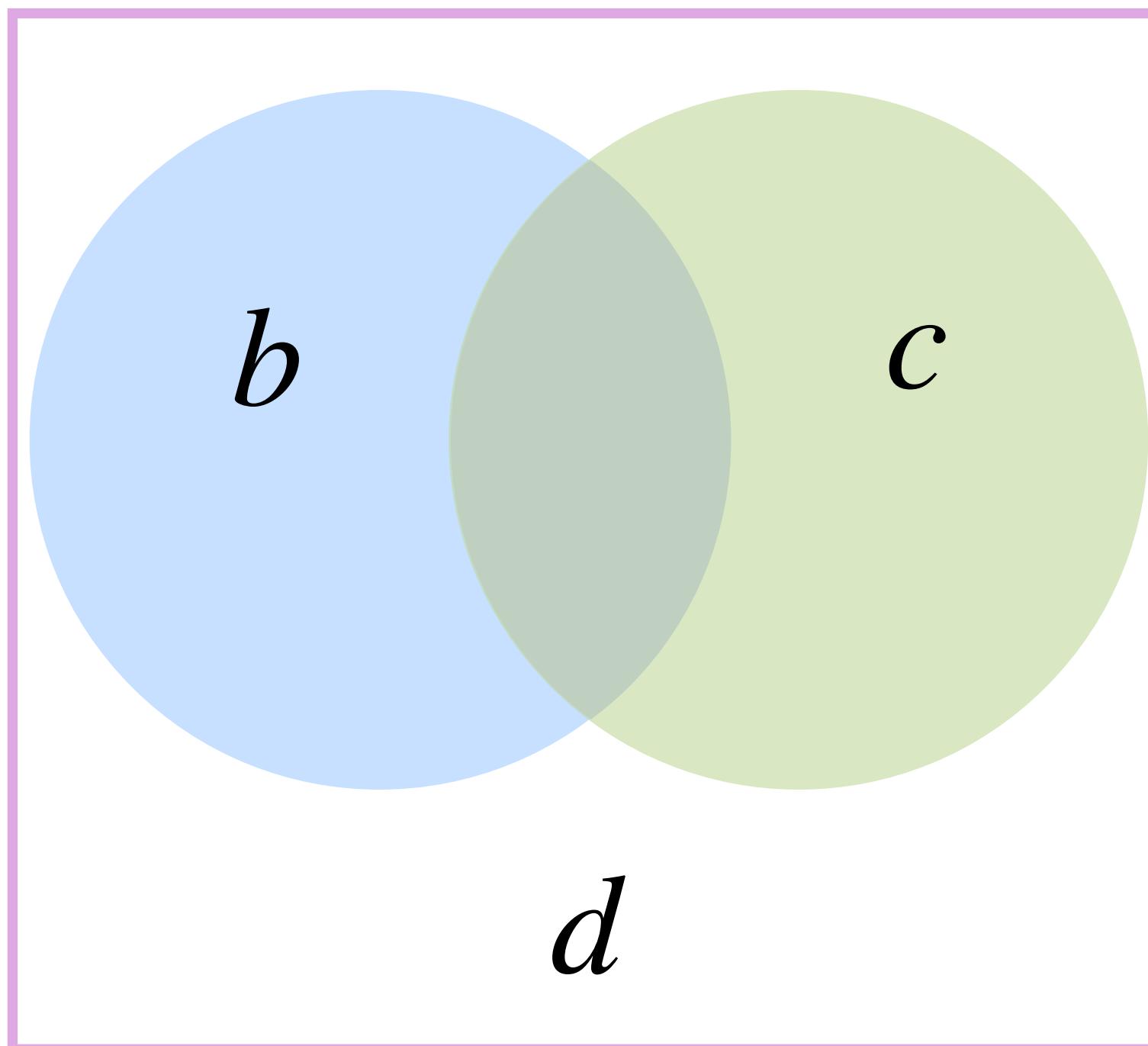
```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
int pd1 = d1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int id = d1_crd[pd1];
    int i = min(ib, ic, id);
    if (ib == i && ic == i && id == i) {
        a[i] = (b[pb1] + c[pc1]) * d[pd1];
    } else if (ib == i && id == i) {
        a[i] = b[pb1] * d[pd1];
    } else if (ic == i && id == i) {
        a[i] = c[pc1] * d[pd1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
    if (id == i) pd1++;
}

while (pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
    int ic = c1_crd[pc1];
    int id = d1_crd[pd1];
    int i = min(ic, id);
    if (ic == i && id == i) {
        a[i] = c[pc1] * d[pd1];
    }
    if (ic == i) pc1++;
    if (id == i) pd1++;
}

while (pb1 < b1_pos[1] && pd1 < d1_pos[1])
{
    int ib = b1_crd[pb1];
    int id = d1_crd[pd1];
    int i = min(ib, id);
    if (ib == i && id == i) {
        a[i] = b[pb1] * d[pd1];
    }
    if (ib == i) pb1++;
    if (id == i) pd1++;
}
```

Iteration lattice for a compound expression

$$a_i = (b_i + c_i)d_i \quad \text{Dense}$$



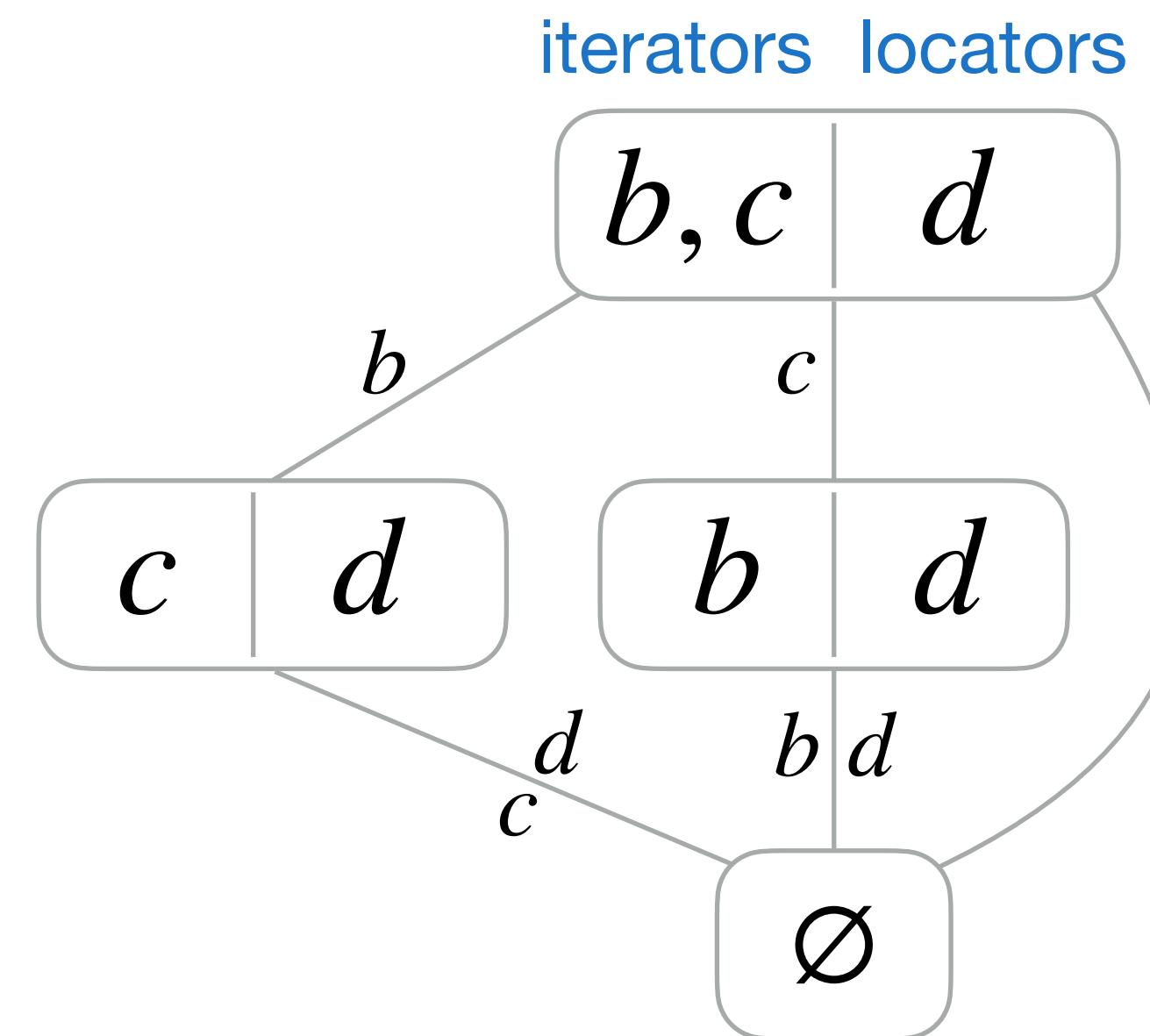
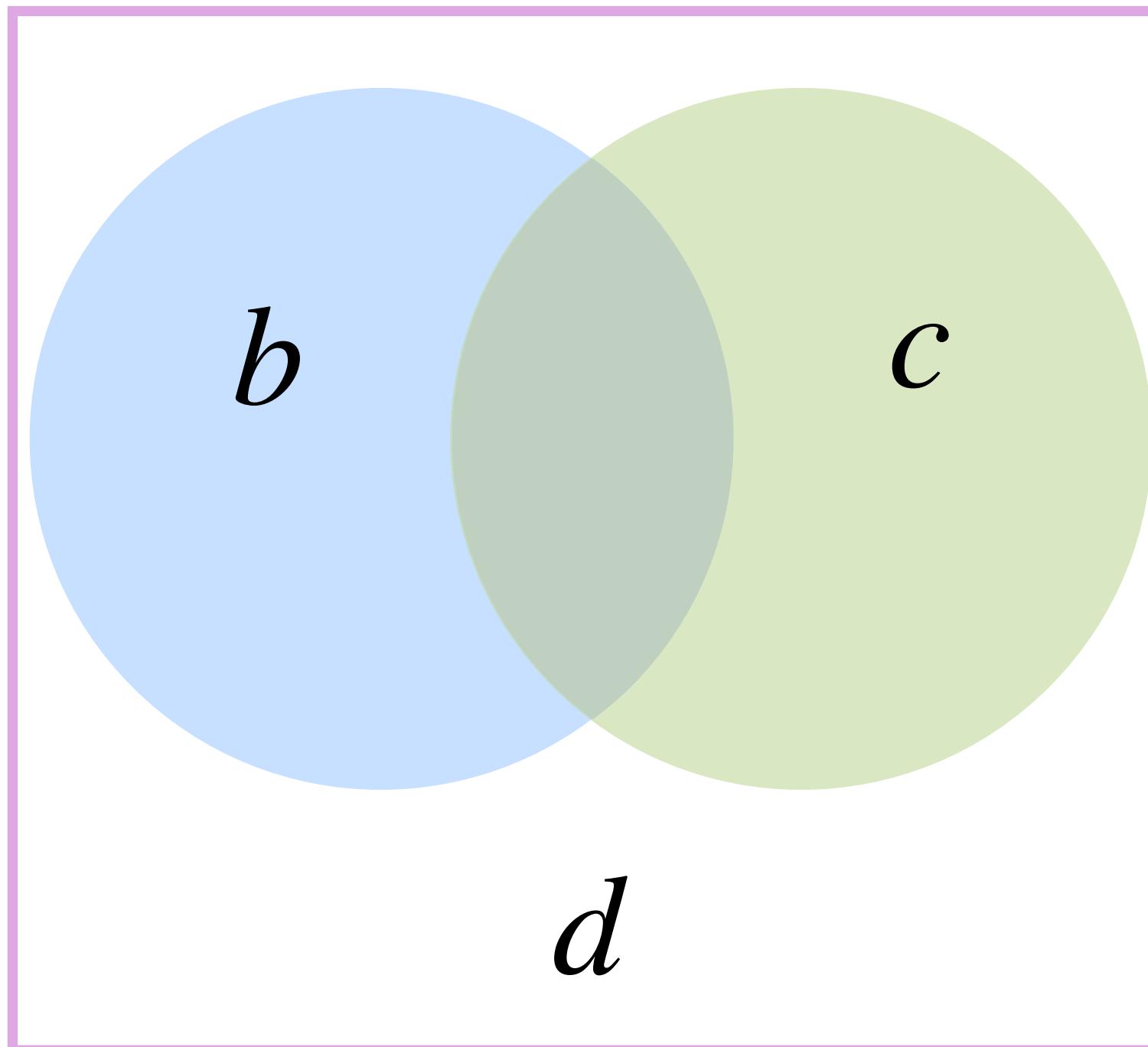
```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
int pd1 = d1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int id = d1_crd[pd1];
    int i = min(ib, ic, id);
    if (ib == i && ic == i && id == i) {
        a[i] = (b[pb1] + c[pc1]) * d[pd1];
    } else if (ib == i && id == i) {
        a[i] = b[pb1] * d[pd1];
    } else if (ic == i && id == i) {
        a[i] = c[pc1] * d[pd1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
    if (id == i) pd1++;
}

while (pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
    int ic = c1_crd[pc1];
    int id = d1_crd[pd1];
    int i = min(ic, id);
    if (ic == i && id == i) {
        a[i] = c[pc1] * d[pd1];
    }
    if (ic == i) pc1++;
    if (id == i) pd1++;
}

while (pb1 < b1_pos[1] && pd1 < d1_pos[1])
{
    int ib = b1_crd[pb1];
    int id = d1_crd[pd1];
    int i = min(ib, id);
    if (ib == i && id == i) {
        a[i] = b[pb1] * d[pd1];
    }
    if (ib == i) pb1++;
    if (id == i) pd1++;
}
```

Iteration lattice for a compound expression

$$a_i = (b_i + c_i)d_i \quad \text{Dense}$$



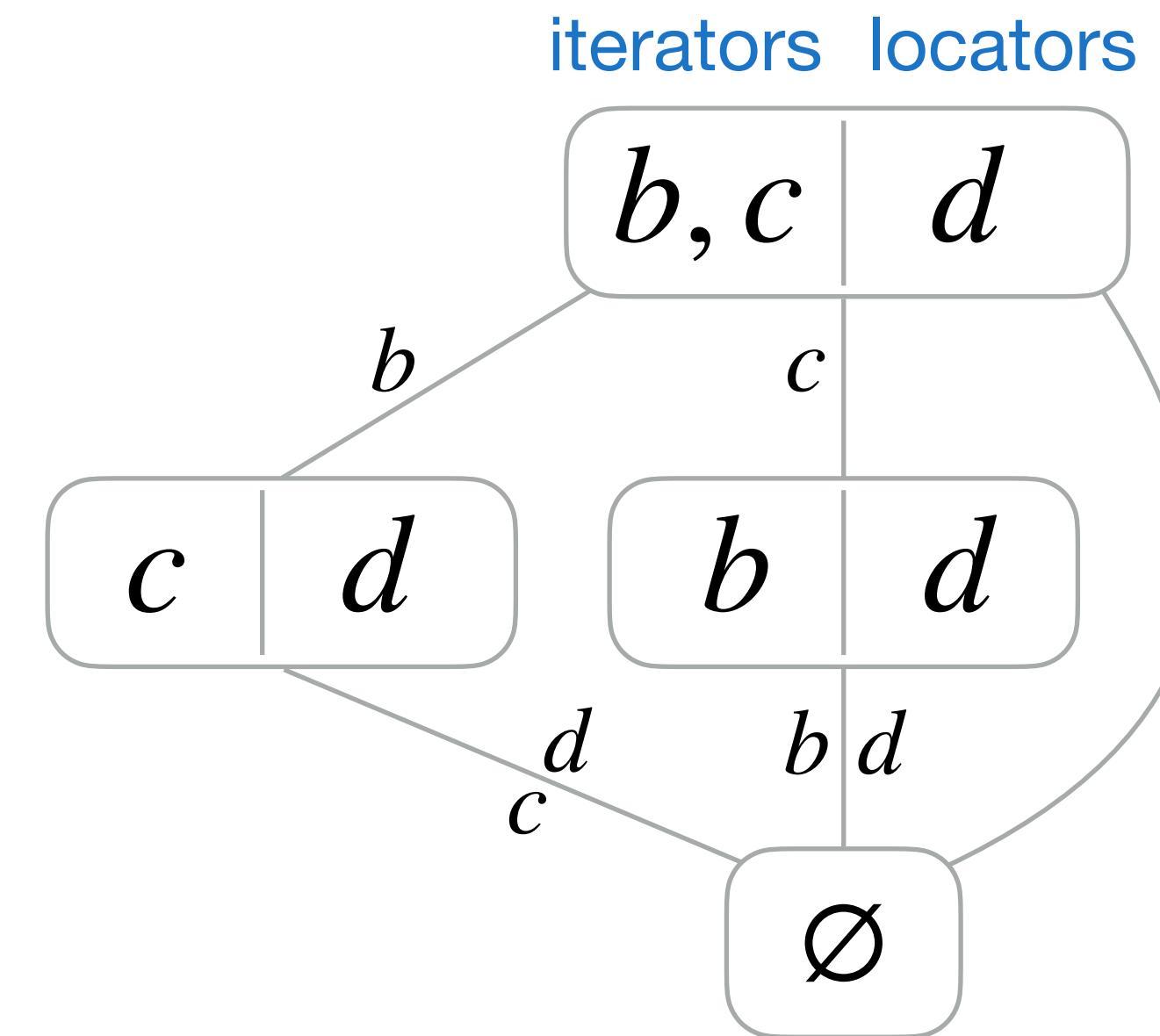
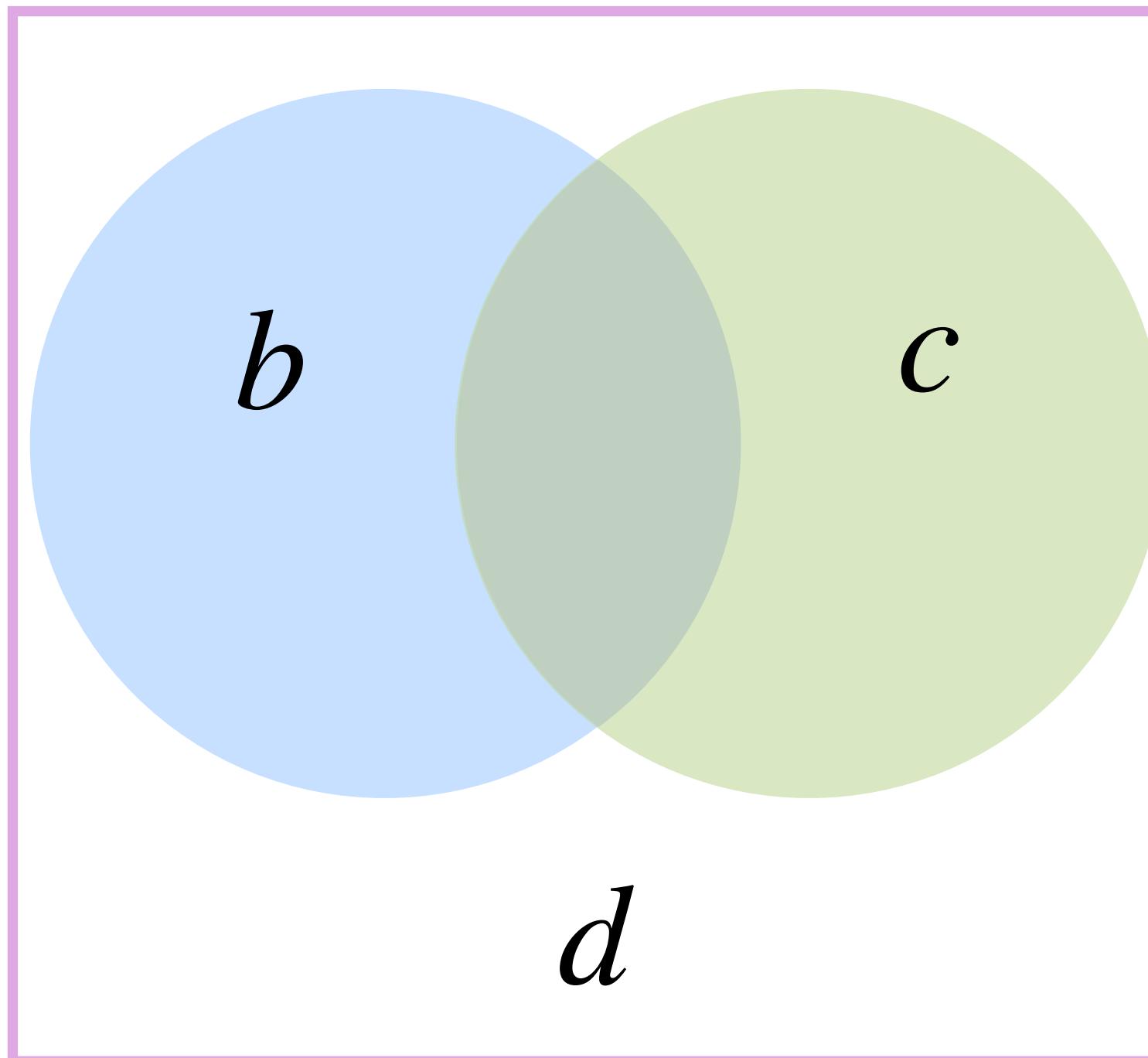
```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
int pd1 = d1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int id = d1_crd[pd1];
    int i = min(ib, ic, id);
    if (ib == i && ic == i && id == i) {
        a[i] = (b[pb1] + c[pc1]) * d[pd1];
    } else if (ib == i && id == i) {
        a[i] = b[pb1] * d[pd1];
    } else if (ic == i && id == i) {
        a[i] = c[pc1] * d[pd1];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
    if (id == i) pd1++;
}

while (pc1 < c1_pos[1] && pd1 < d1_pos[1]) {
    int ic = c1_crd[pc1];
    int id = d1_crd[pd1];
    int i = min(ic, id);
    if (ic == i && id == i) {
        a[i] = c[pc1] * d[pd1];
    }
    if (ic == i) pc1++;
    if (id == i) pd1++;
}

while (pb1 < b1_pos[1] && pd1 < d1_pos[1])
{
    int ib = b1_crd[pb1];
    int id = d1_crd[pd1];
    int i = min(ib, id);
    if (ib == i && id == i) {
        a[i] = b[pb1] * d[pd1];
    }
    if (ib == i) pb1++;
    if (id == i) pd1++;
}
```

Iteration lattice for a compound expression

$$a_i = (b_i + c_i)d_i \quad \text{Dense}$$

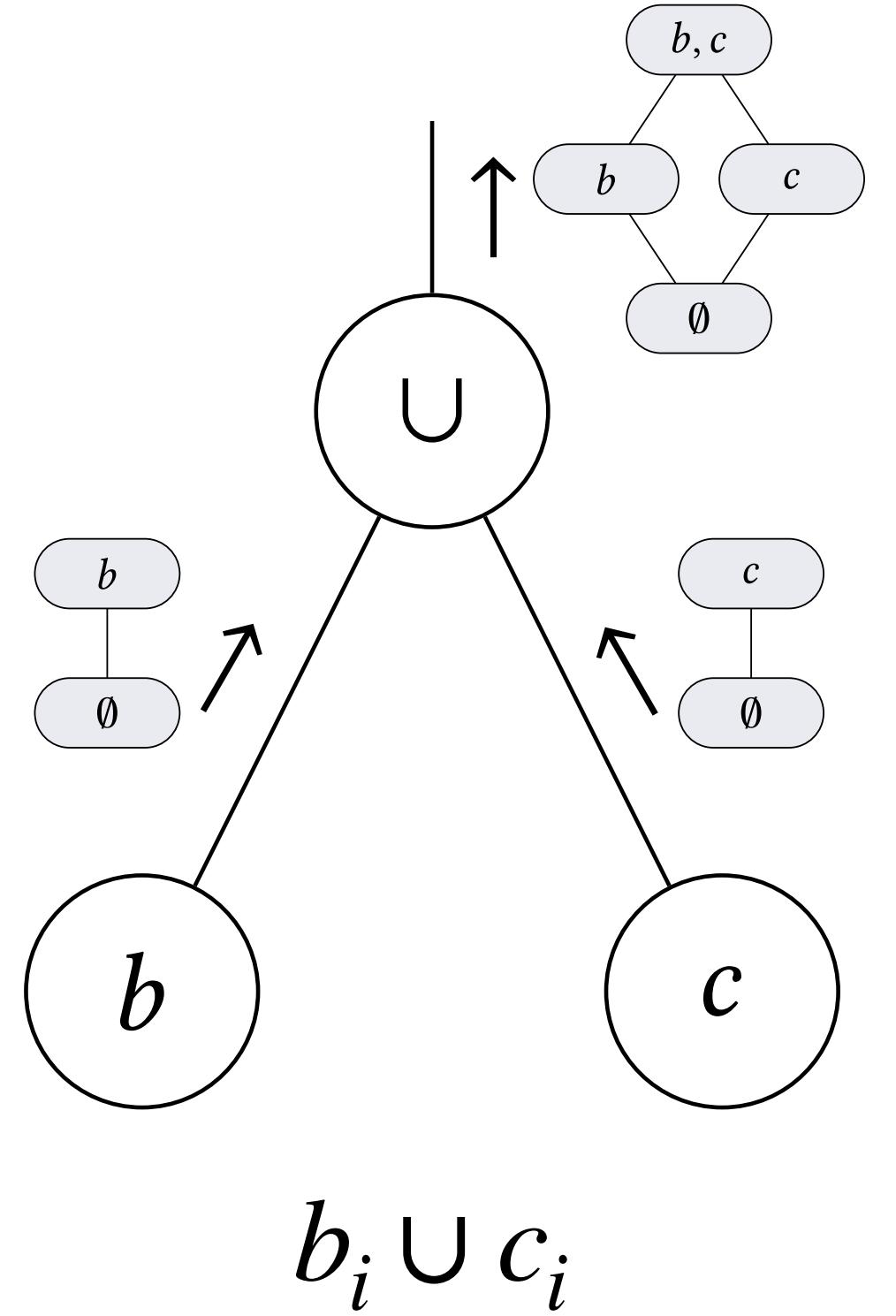


```
int pb1 = b1_pos[0];
int pc1 = c1_pos[0];
while (pb1 < b1_pos[1] && pc1 < c1_pos[1]) {
    int ib = b1_crd[pb1];
    int ic = c1_crd[pc1];
    int i = min(ib, ic);
    if (ib == i && ic == i) {
        a[i] = (b[pb1] + c[pc1]) * d[i];
    } else if (ib == i) {
        a[i] = b[pb1] * d[i];
    } else if (ic == i) {
        a[i] = c[pc1] * d[i];
    }
    if (ib == i) pb1++;
    if (ic == i) pc1++;
}

while (pb1 < b1_pos[1]) {
    int i = b1_crd[pb1];
    a[i] = b[pb1] * d[i];
    pb1++;
}

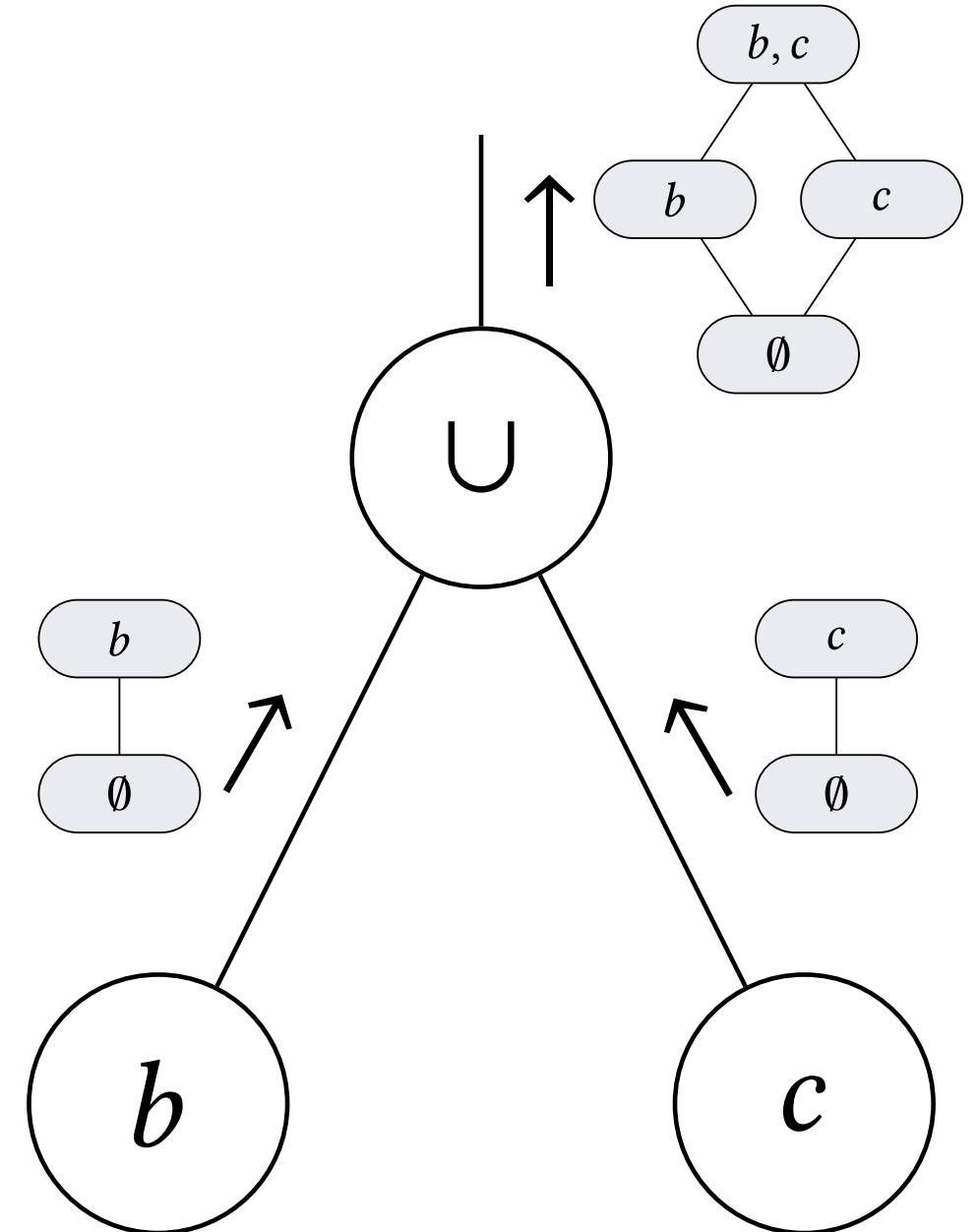
while (pc1 < c1_pos[1]) {
    int i = c1_crd[pc1];
    a[i] = c[pc1] * d[i];
    pc1++;
}
```

Iteration lattice construction



Bottom-up construction from set expression:
create and merge iteration lattices

Iteration lattice construction

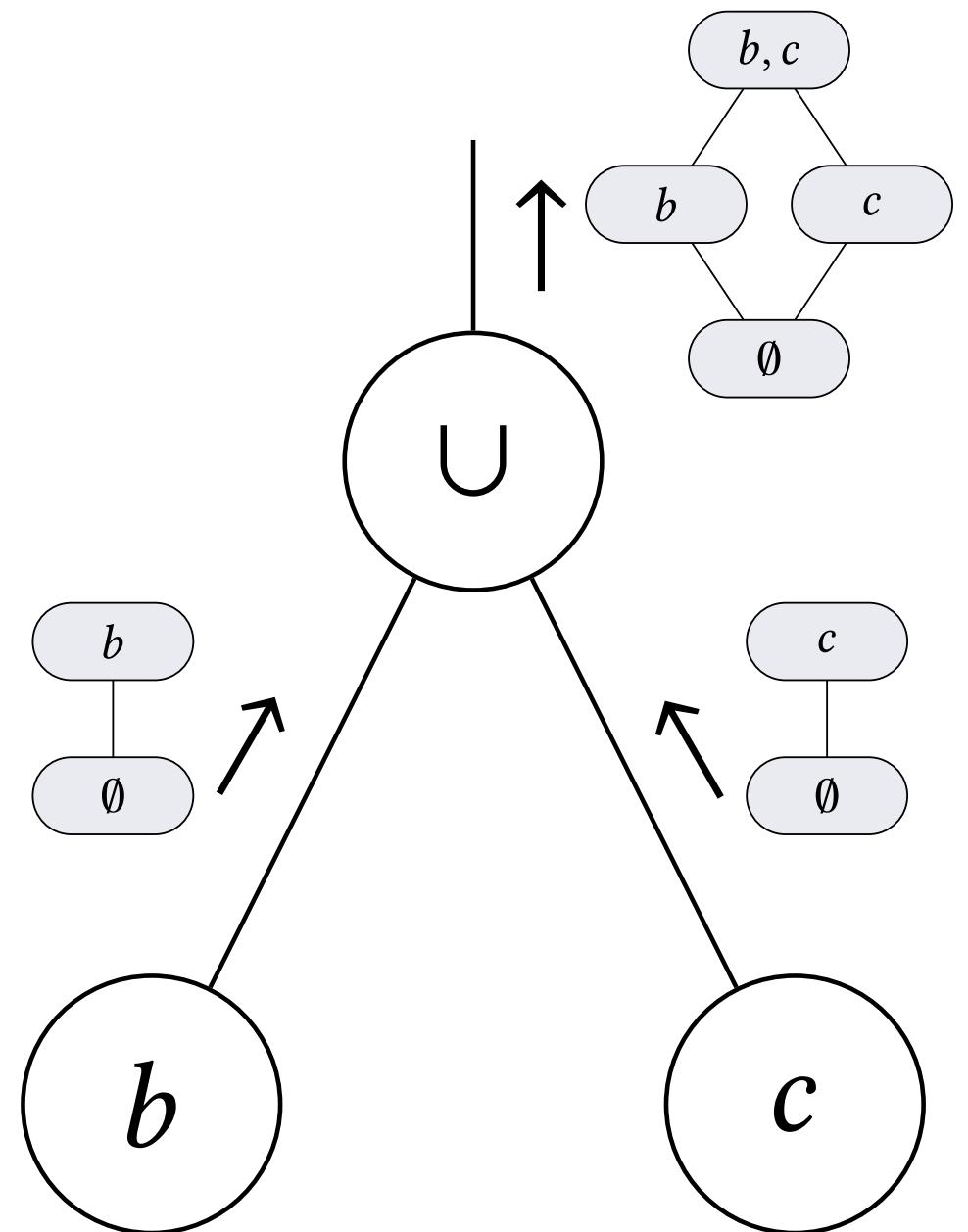


$$b_i \cup c_i$$

Bottom-up construction from set expression:
create and merge iteration lattices

Base cases:

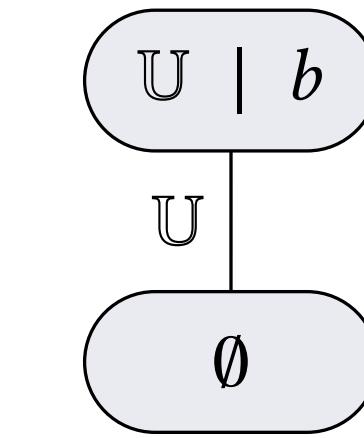
Iteration lattice construction



$$b_i \cup c_i$$

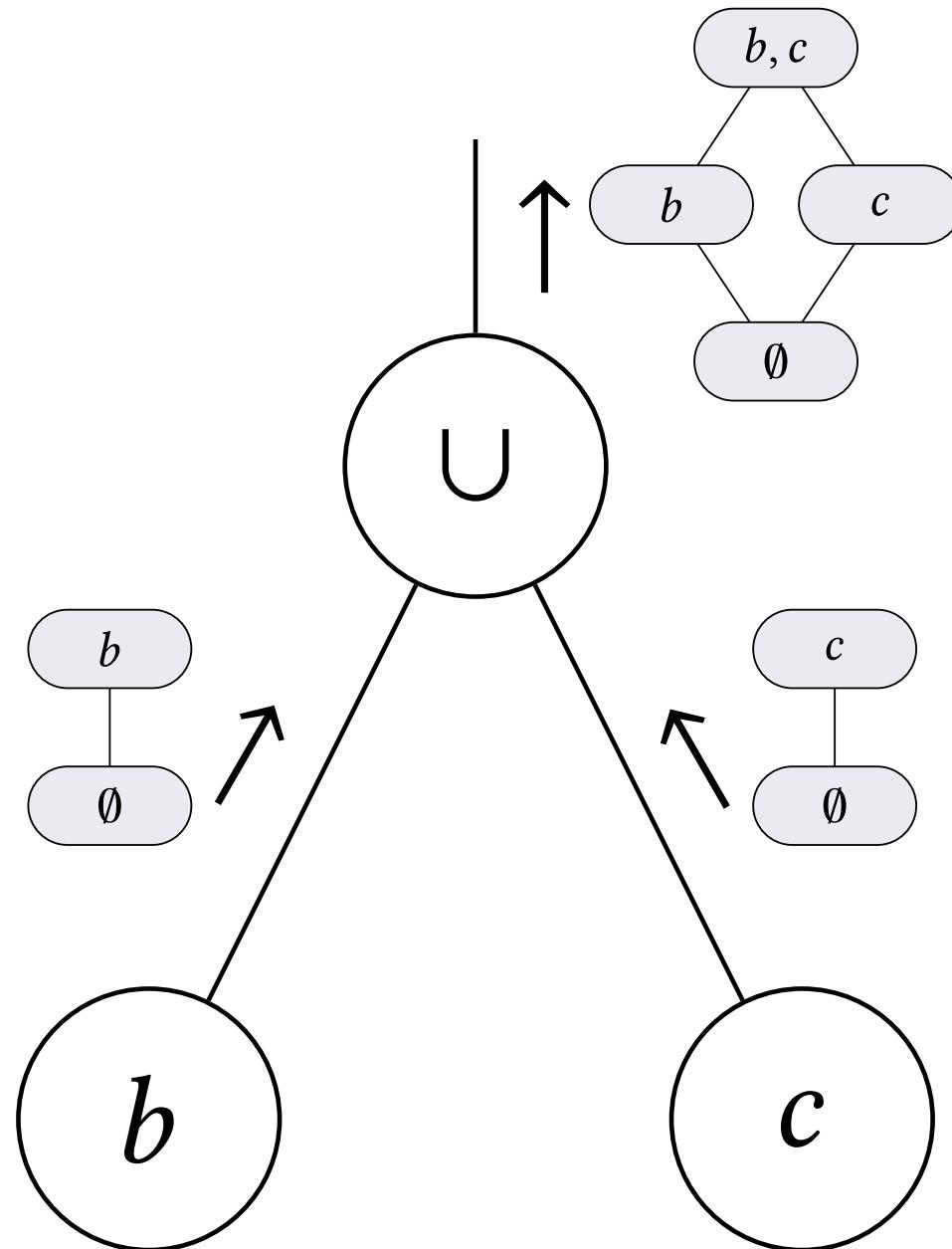
Bottom-up construction from set expression:
create and merge iteration lattices

Base cases:



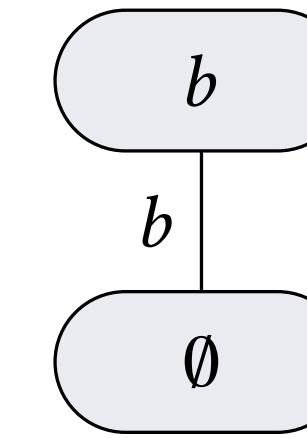
b does not have an iterator,
but supports locate

Iteration lattice construction

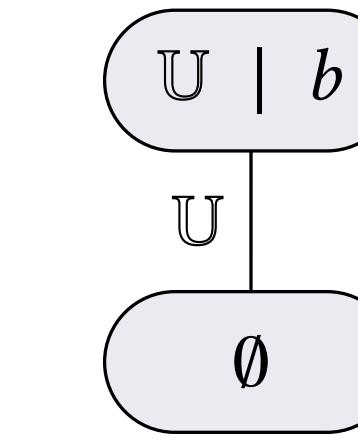


$$b_i \cup c_i$$

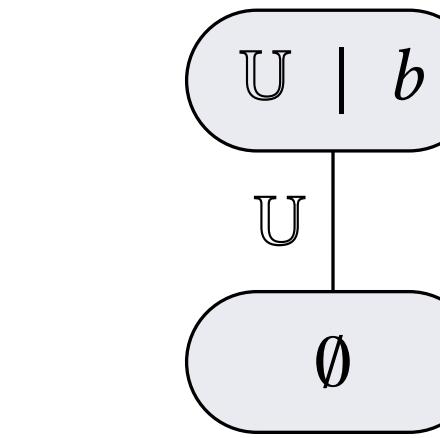
Bottom-up construction from set expression:
create and merge iteration lattices



b has an iterator

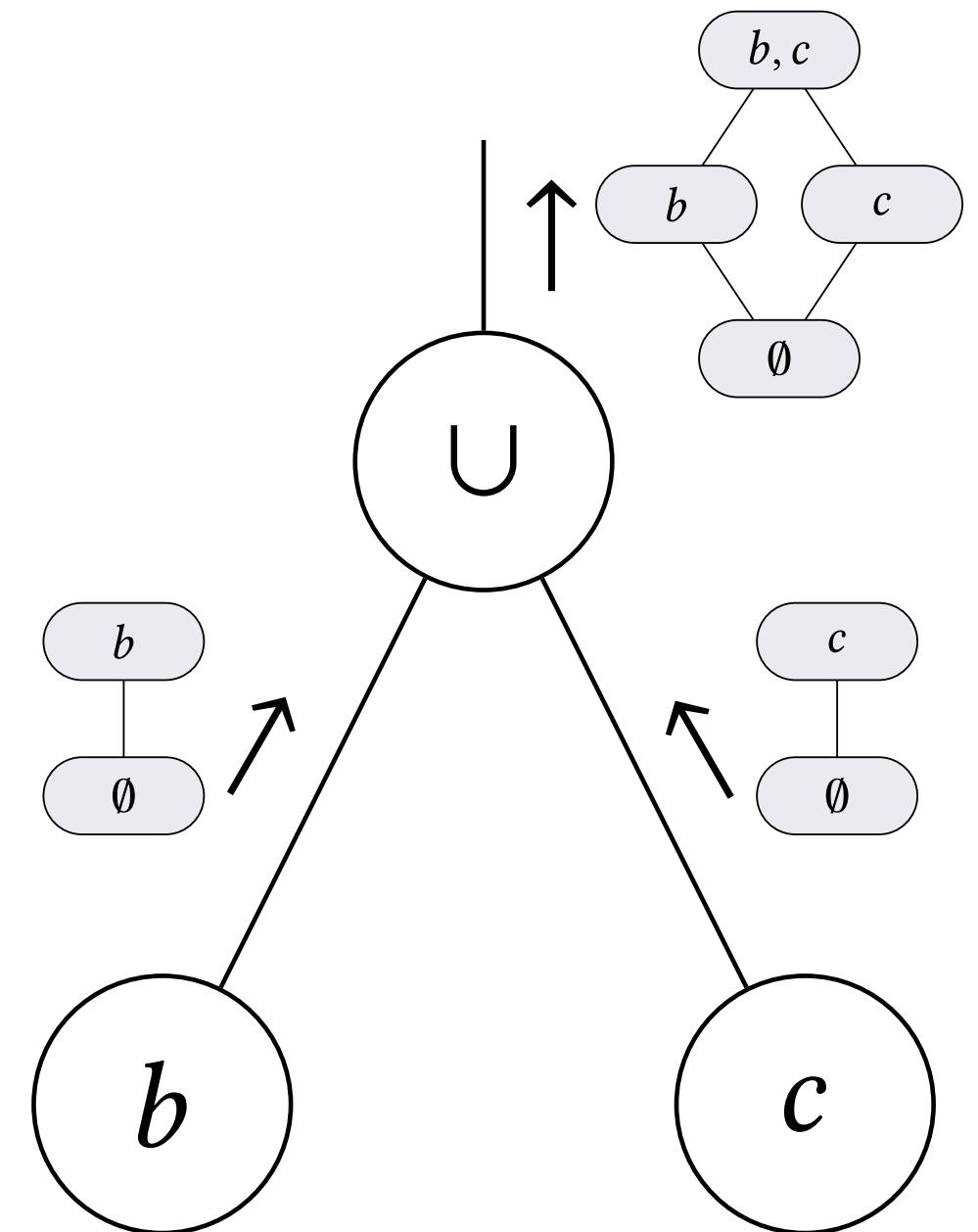


b does not have an iterator,
but supports locate



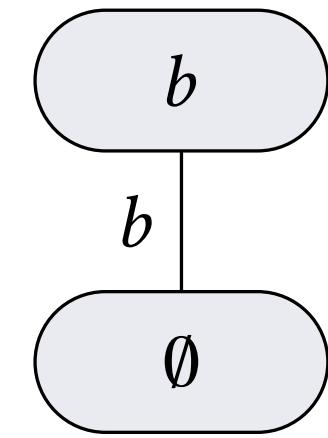
Base cases:

Iteration lattice construction

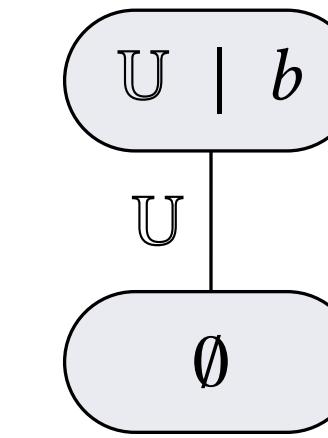


$$b_i \cup c_i$$

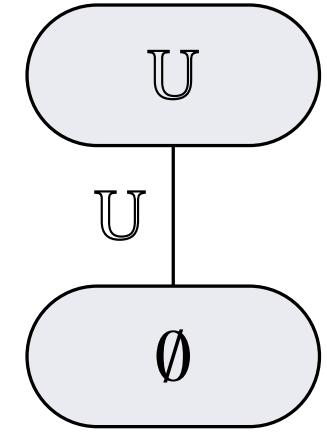
Bottom-up construction from set expression:
create and merge iteration lattices



b has an iterator

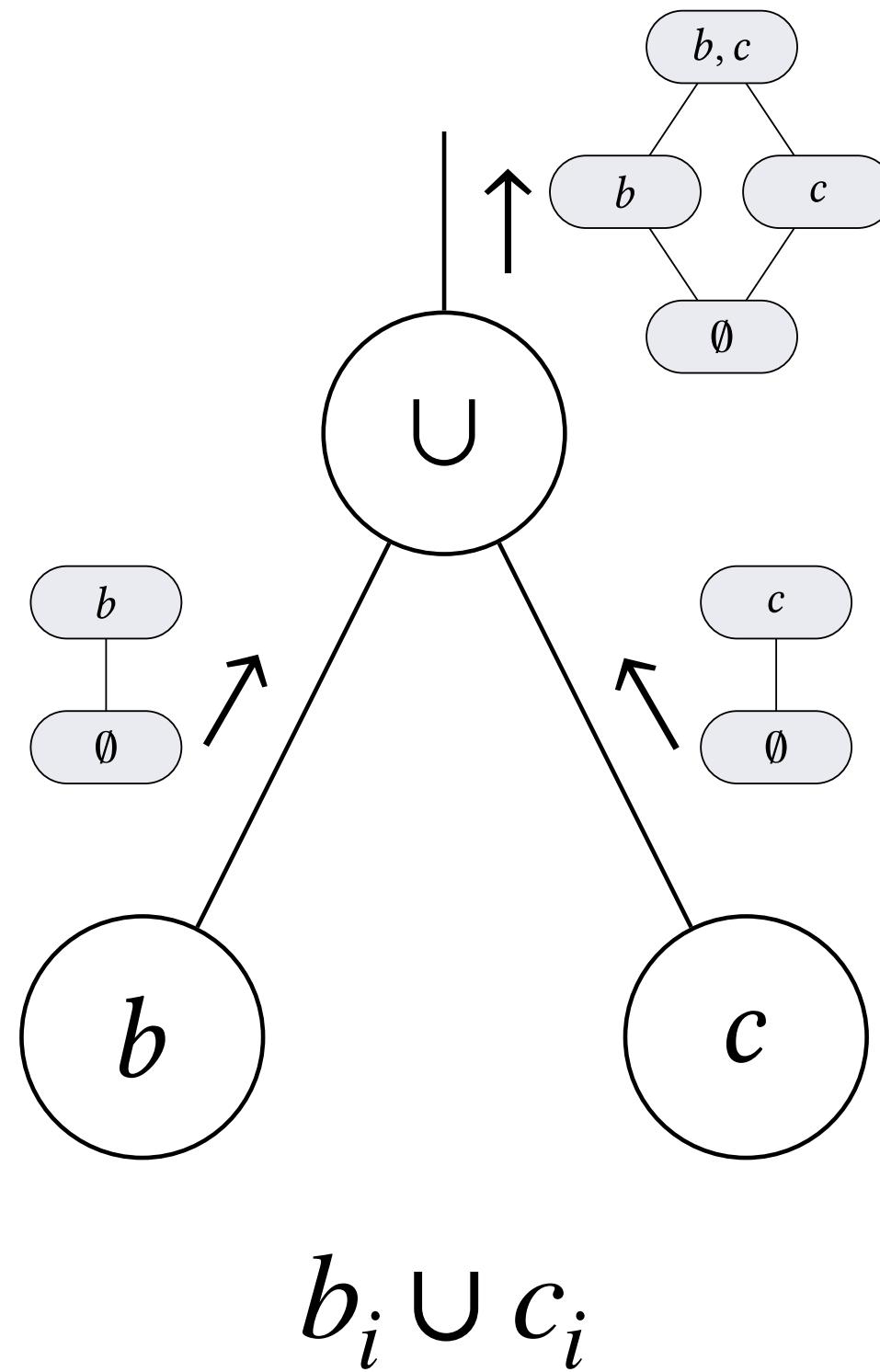


b does not have an iterator,
but supports locate

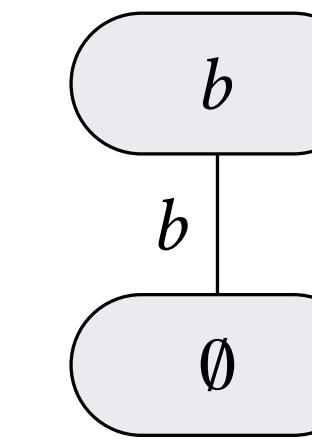


b is the set universe

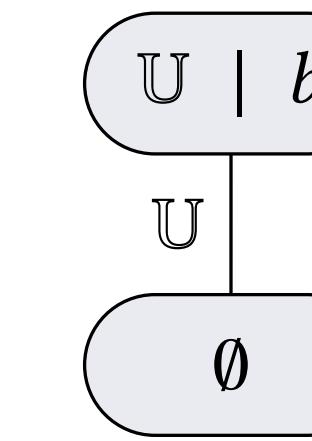
Iteration lattice construction



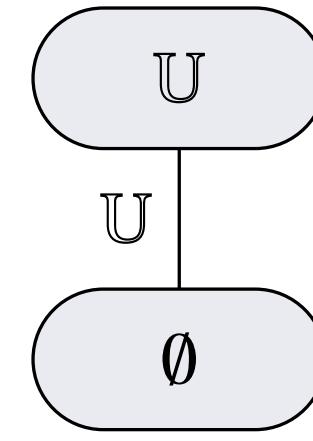
Bottom-up construction from set expression:
create and merge iteration lattices



b has an iterator



b does not have an iterator,
but supports locate



b is the set universe

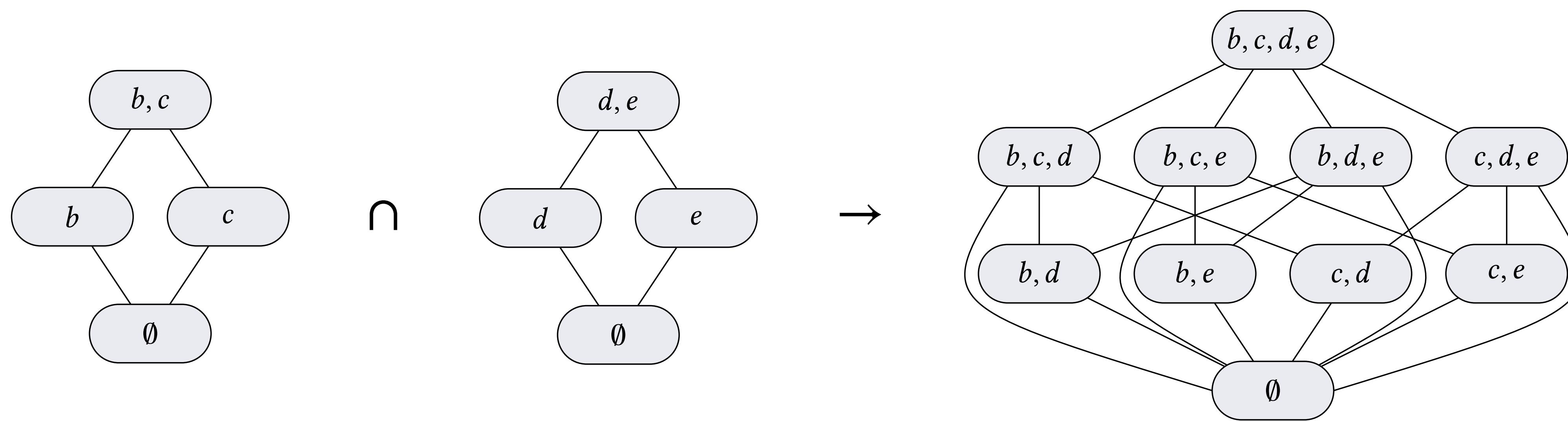
Lattice point merging:

$$((b, c), (d, e)) \rightarrow (b, c, d, e)$$

Lattice points are merged by
taking the union of their iterator
and locator sets respectively

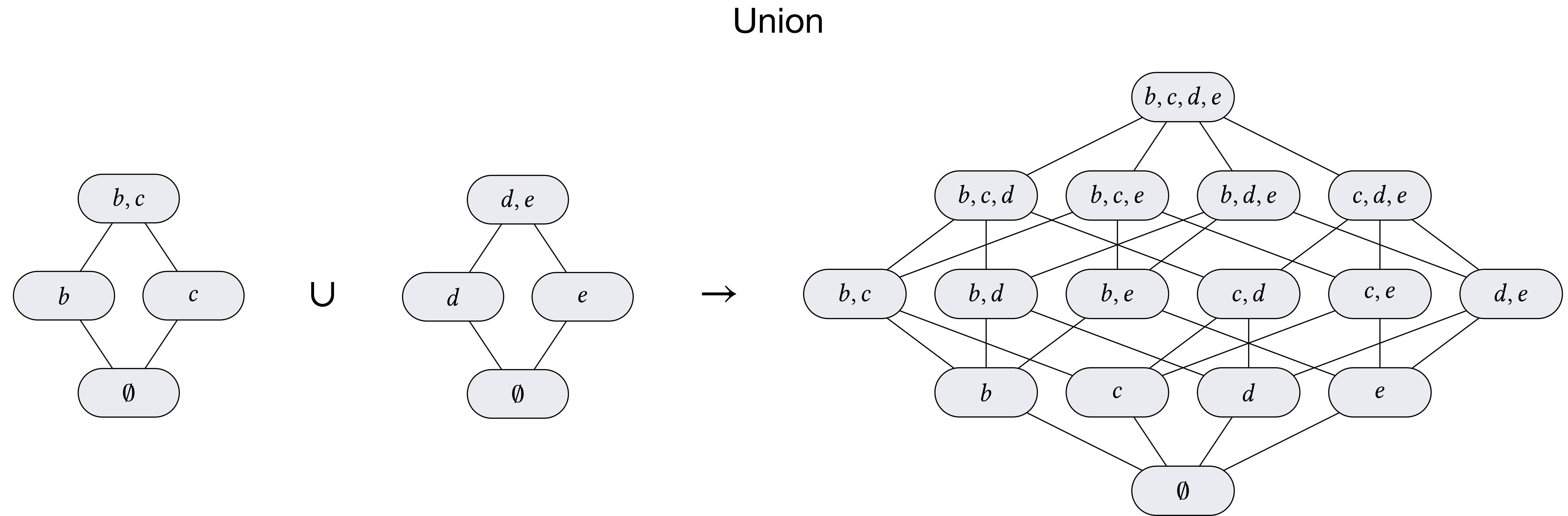
Iteration lattice construction

Intersection



The intersection of two lattices is computed by merging the lattice point pairs in the Cartesian combination of their lattice points.

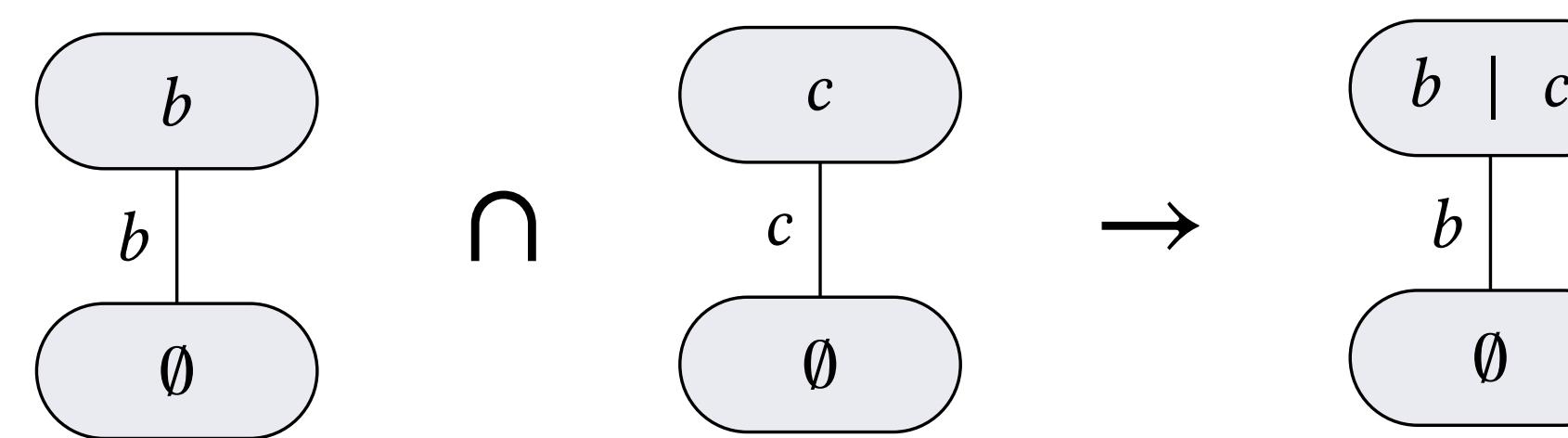
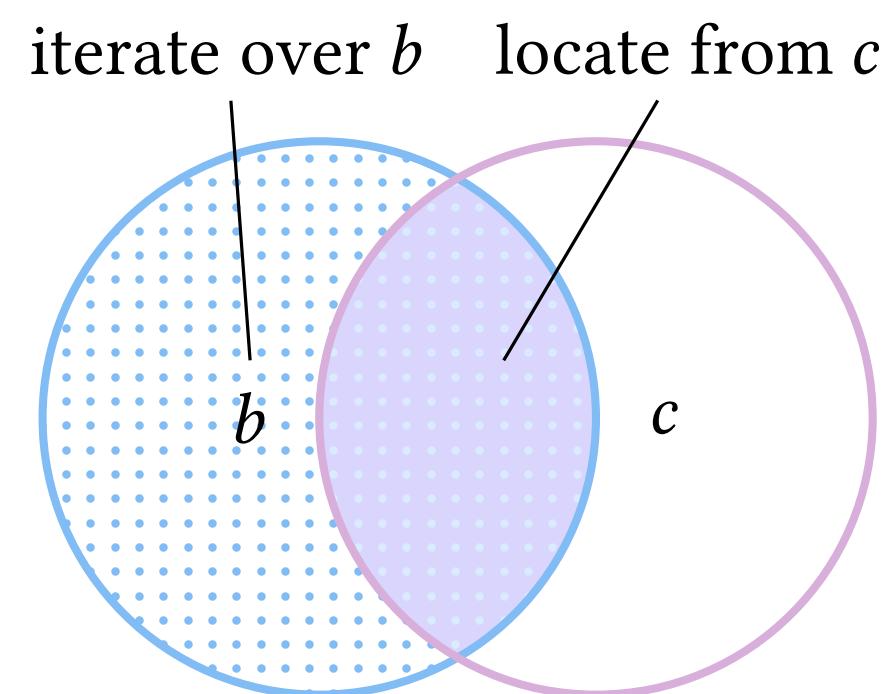
Iteration lattice construction



The union of two lattices is computed by first merging the lattice point pairs in the Cartesian combination of their lattice points. The union of the lattices is then the union of the result and the two initial lattices.

Iteration lattice optimization example

Intersection Optimization



When intersecting two lattices, move the operands with the locate capability from one side of the intersection from the iterators to the locators set.