

A Unified Framework for Schedule and Storage Optimization

William Thies, Frédéric Vivien*,
Jeffrey Sheldon, and Saman Amarasinghe

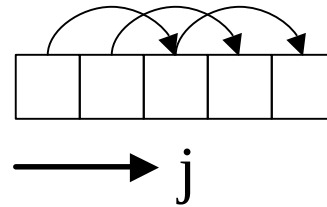
MIT Laboratory for Computer Science

* ICPS/LSIIT, Université Louis Pasteur

<http://compiler.lcs.mit.edu/aov>

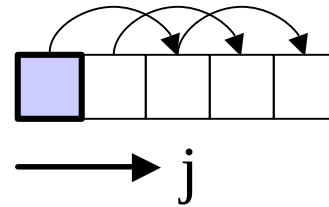
Motivating Example

```
for i = 1 to n  
  for j = 1 to n  
     $A[j] = f(A[j], A[j-2])$ 
```



Motivating Example

```
for i = 1 to n
  for j = 1 to n
    A[j] = f(A[j], A[j-2])
```

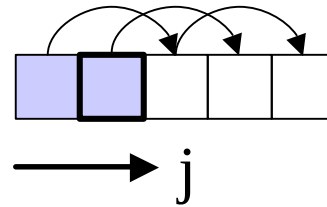


$t = 1$

$(i, j) = (1, 1)$

Motivating Example

```
for i = 1 to n
  for j = 1 to n
    A[j] = f(A[j], A[j-2])
```

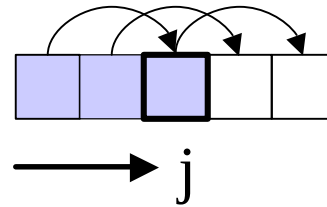


$t = 2$

$(i, j) = (1, 2)$

Motivating Example

```
for i = 1 to n
  for j = 1 to n
    A[j] = f(A[j], A[j-2])
```

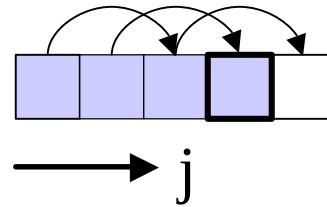


$t = 3$

$(i, j) = (1, 3)$

Motivating Example

```
for i = 1 to n
  for j = 1 to n
    A[j] = f(A[j], A[j-2])
```

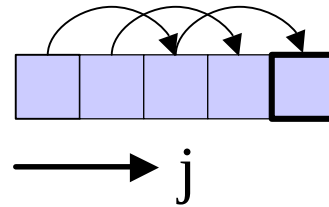


$t = 4$

$(i, j) = (1, 4)$

Motivating Example

```
for i = 1 to n
  for j = 1 to n
    A[j] = f(A[j], A[j-2])
```

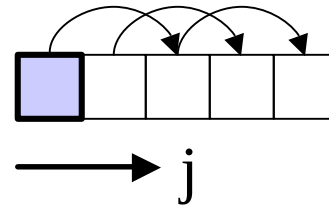


$t = 5$

$(i, j) = (1, 5)$

Motivating Example

```
for i = 1 to n
  for j = 1 to n
    A[j] = f(A[j], A[j-2])
```

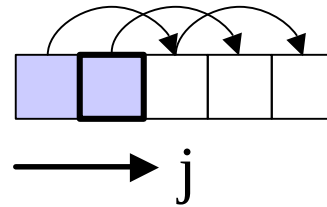


$t = 6$

$(i, j) = (2, 1)$

Motivating Example

```
for i = 1 to n
  for j = 1 to n
    A[j] = f(A[j], A[j-2])
```

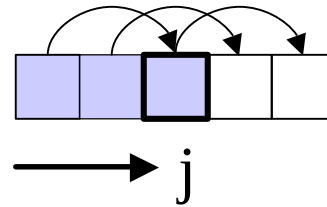


$t = 7$

$(i, j) = (2, 2)$

Motivating Example

```
for i = 1 to n
  for j = 1 to n
    A[j] = f(A[j], A[j-2])
```

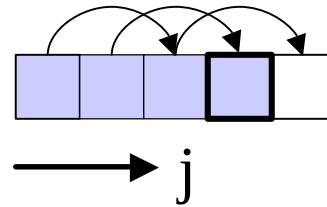


$t = 8$

$(i, j) = (2, 3)$

Motivating Example

```
for i = 1 to n
  for j = 1 to n
    A[j] = f(A[j], A[j-2])
```

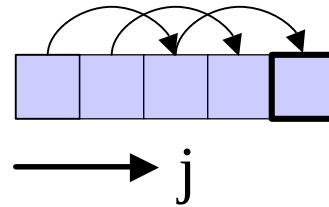


$t = 9$

$(i, j) = (2, 4)$

Motivating Example

```
for i = 1 to n
  for j = 1 to n
    A[j] = f(A[j], A[j-2])
```

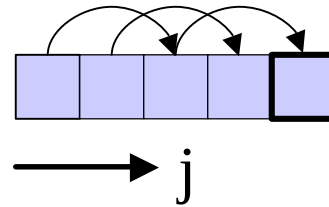


$t = 10$

$(i, j) = (2, 5)$

Motivating Example

```
for i = 1 to n
  for j = 1 to n
    A[j] = f(A[j], A[j-2])
```

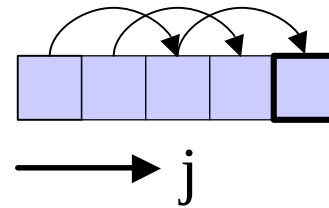


$t = 25$

$(i, j) = (5, 5)$

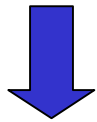
Motivating Example

```
for i = 1 to n
  for j = 1 to n
    A[j] = f(A[j], A[j-2])
```



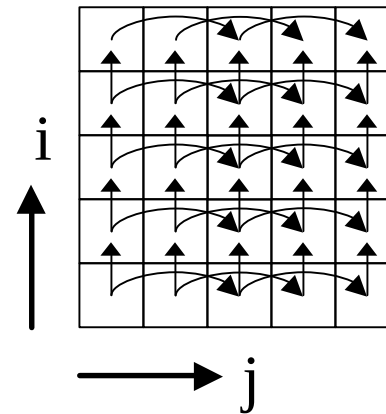
$t = 25$

$(i, j) = (5, 5)$



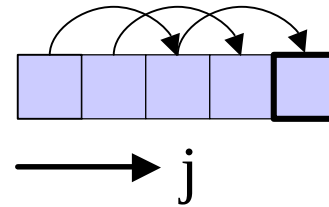
Array Expansion

```
init A[0][j]
for i = 1 to n
  for j = 1 to n
    A[i][j] = f(A[i-1][j],
                A[i][j-2])
```



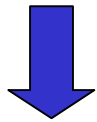
Motivating Example

```
for i = 1 to n
  for j = 1 to n
    A[j] = f(A[j], A[j-2])
```



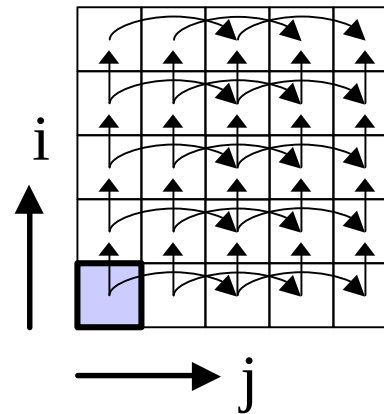
$t = 25$

$(i, j) = (5, 5)$



Array Expansion

```
init A[0][j]
for i = 1 to n
  for j = 1 to n
    A[i][j] = f(A[i-1][j],
                A[i][j-2])
```

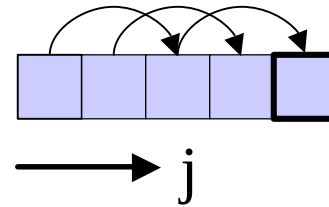


$t = 1$

$(i, j) = (1, 1)$

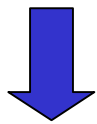
Motivating Example

```
for i = 1 to n
  for j = 1 to n
    A[j] = f(A[j], A[j-2])
```



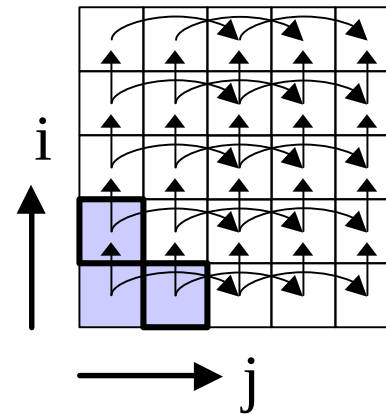
$t = 25$

$(i, j) = (5, 5)$



Array Expansion

```
init A[0][j]
for i = 1 to n
  for j = 1 to n
    A[i][j] = f(A[i-1][j],
                A[i][j-2])
```

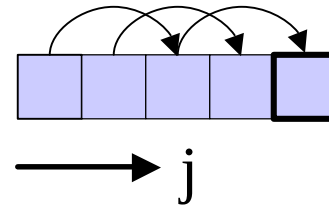


$t = 2$

$(i, j) = \{(1, 2), (2, 1)\}$

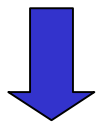
Motivating Example

```
for i = 1 to n
  for j = 1 to n
    A[j] = f(A[j], A[j-2])
```



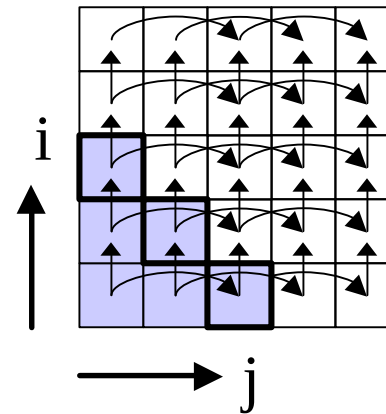
$t = 25$

$(i, j) = (5, 5)$



Array Expansion

```
init A[0][j]
for i = 1 to n
  for j = 1 to n
    A[i][j] = f(A[i-1][j],
                A[i][j-2])
```

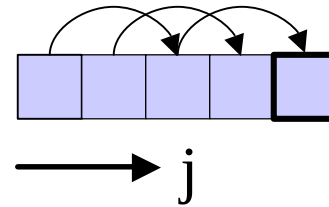


$t = 3$

$(i, j) = \{(1, 3),$
 $(2, 2),$
 $(3, 1)\}$

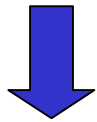
Motivating Example

```
for i = 1 to n
  for j = 1 to n
    A[j] = f(A[j], A[j-2])
```



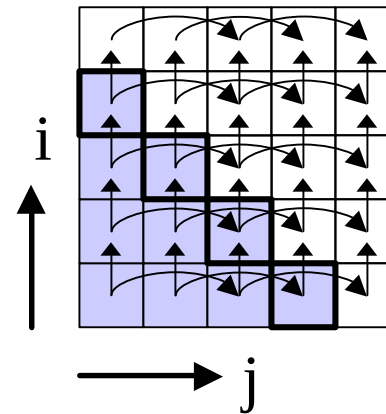
$t = 25$

$(i, j) = (5, 5)$



Array Expansion

```
init A[0][j]
for i = 1 to n
  for j = 1 to n
    A[i][j] = f(A[i-1][j],
                A[i][j-2])
```

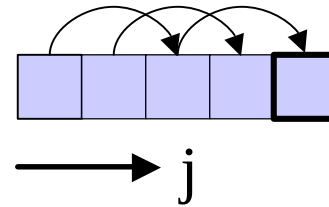


$t = 4$

$(i, j) = \{(1, 4),$
 $(2, 3),$
 $(3, 2),$
 $(4, 1)\}$

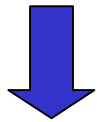
Motivating Example

```
for i = 1 to n
  for j = 1 to n
    A[j] = f(A[j], A[j-2])
```



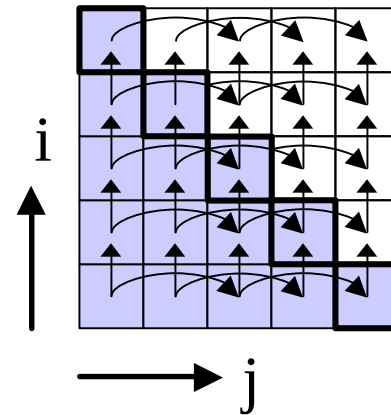
$t = 25$

$(i, j) = (5, 5)$



Array Expansion

```
init A[0][j]
for i = 1 to n
  for j = 1 to n
    A[i][j] = f(A[i-1][j],
                A[i][j-2])
```

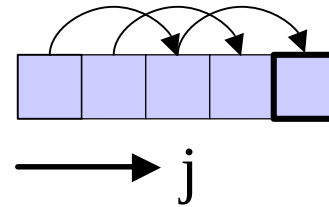


$t = 5$

$(i, j) = \{(1, 5),$
 $(2, 4),$
 $(3, 3),$
 $(4, 2),$
 $(5, 1)\}$

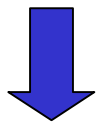
Motivating Example

```
for i = 1 to n
  for j = 1 to n
    A[j] = f(A[j], A[j-2])
```



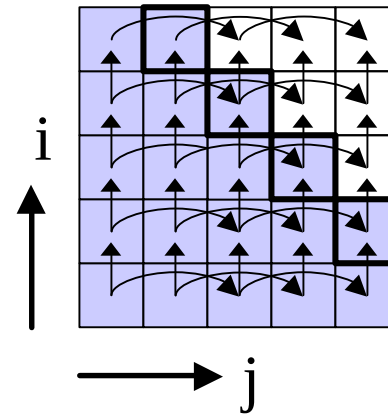
$t = 25$

$(i, j) = (5, 5)$



Array Expansion

```
init A[0][j]
for i = 1 to n
  for j = 1 to n
    A[i][j] = f(A[i-1][j],
               A[i][j-2])
```

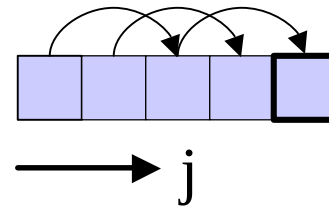


$t = 6$

$(i, j) = \{(2, 5),$
 $(3, 4),$
 $(4, 3),$
 $(5, 2)\}$

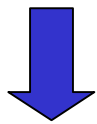
Motivating Example

```
for i = 1 to n
  for j = 1 to n
    A[j] = f(A[j], A[j-2])
```



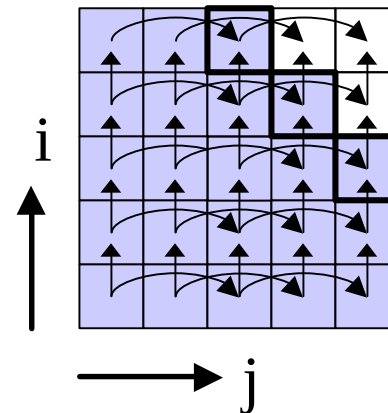
$t = 25$

$(i, j) = (5, 5)$



Array Expansion

```
init A[0][j]
for i = 1 to n
  for j = 1 to n
    A[i][j] = f(A[i-1][j],
                A[i][j-2])
```

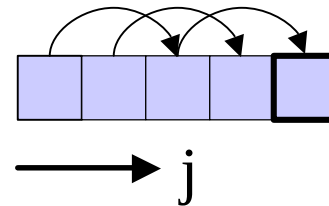


$t = 7$

$(i, j) = \{(3, 5),$
 $(4, 4),$
 $(5, 3)\}$

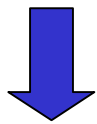
Motivating Example

```
for i = 1 to n
  for j = 1 to n
    A[j] = f(A[j], A[j-2])
```



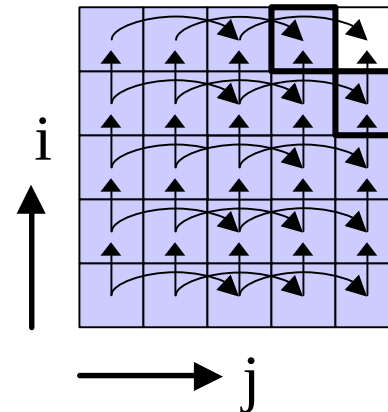
$t = 25$

$(i, j) = (5, 5)$



Array Expansion

```
init A[0][j]
for i = 1 to n
  for j = 1 to n
    A[i][j] = f(A[i-1][j],
                A[i][j-2])
```

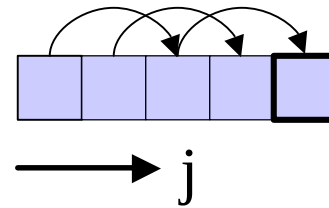


$t = 8$

$(i, j) = \{(4, 5), (5, 4)\}$

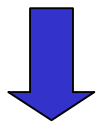
Motivating Example

```
for i = 1 to n
  for j = 1 to n
    A[j] = f(A[j], A[j-2])
```



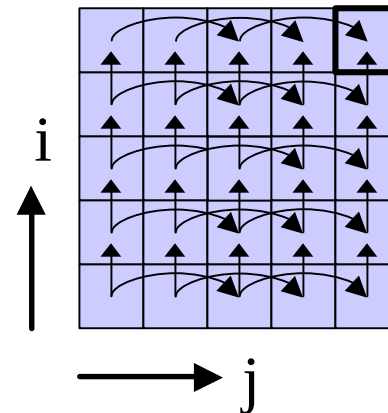
$t = 25$

$(i, j) = (5, 5)$



Array Expansion

```
init A[0][j]
for i = 1 to n
  for j = 1 to n
    A[i][j] = f(A[i-1][j],
                A[i][j-2])
```

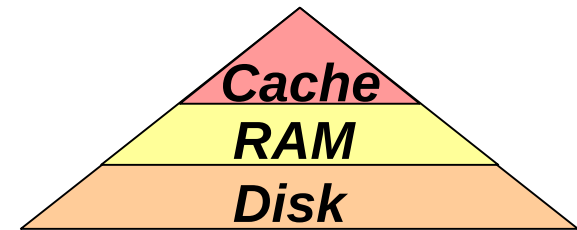


$t = 9$

$(i, j) = (5, 5)$

Parallelism/Storage Tradeoff

- Increasing storage can enable parallelism
 - But storage can be expensive



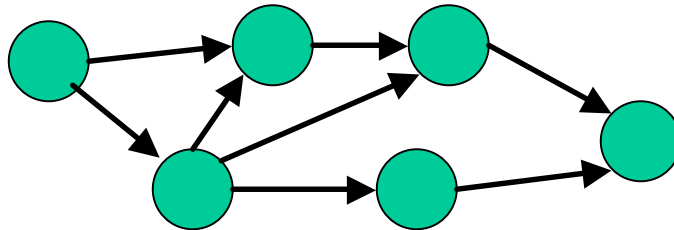
- Phase ordering problem
 - Optimizing for storage restricts parallelism
 - Maximizing parallelism restricts storage options
 - Too complex to consider all combinations
- ➔ Need **efficient framework** to integrate schedule and storage optimization

Outline

- Abstract problem
- Simplifications
- Concrete problem
- Solution Method
- Conclusions

Abstract Problem

- Given DAG of dependent operations



- Must execute producers before consumers
- Must store a value until all consumers execute
- Two parameters control execution:
 1. A scheduling function θ
 - Maps each operation to execution time
 - Parallelism is implicit
 2. A fully associative store of size m

Abstract Problem

- We can ask three questions:
- Two parameters control execution:
 1. A scheduling function θ
 - Maps each operation to execution time
 - Parallelism is implicit
 2. A fully associative store of size m

Abstract Problem

- We can ask three questions:
 1. Given θ , what is the smallest m ?
- Two parameters control execution:
 1. A scheduling function θ
 - Maps each operation to execution time
 - Parallelism is implicit
 2. A fully associative store of size m

Abstract Problem

- We can ask three questions:
 1. Given θ , what is the smallest m ?
 2. Given m , what is the “best” θ ?
- Two parameters control execution:
 1. A scheduling function θ
 - Maps each operation to execution time
 - Parallelism is implicit
 2. A fully associative store of size m

Abstract Problem

- We can ask three questions:
 1. Given θ , what is the smallest m ?
 2. Given m , what is the “best” θ ?
 3. What is the smallest m that is valid for all legal θ ?
- Two parameters control execution:
 1. A scheduling function θ
 - Maps each operation to execution time
 - Parallelism is implicit
 2. A fully associative store of size m

Outline

- Abstract problem
- **Simplifications**
- Concrete problem
- Solution Method
- Conclusions

Simplifying the Schedule

- Real programs aren't DAG's
 - Dependence graph is parameterized by loops
 - Too many nodes to schedule
 - Size could even be unknown (symbolic constants)
- Use classical solution: affine schedules
 - Each statement has a scheduling function θ
 - Each θ is an affine function of the enclosing loop counters and symbolic constants
 - To simplify talk, ignore symbolic constants:

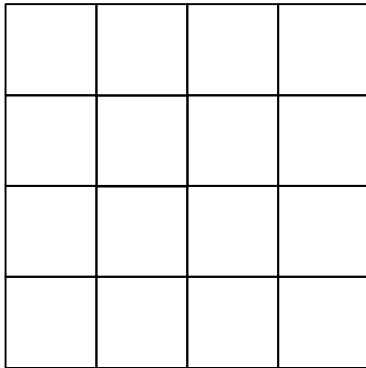
$$\theta(\vec{i}) = \vec{\beta} \cdot \vec{i}$$

Simplifying the Storage Mapping

- Programs use arrays, not associative maps
 - If size decreases, need to specify which elements are mapped to the same location

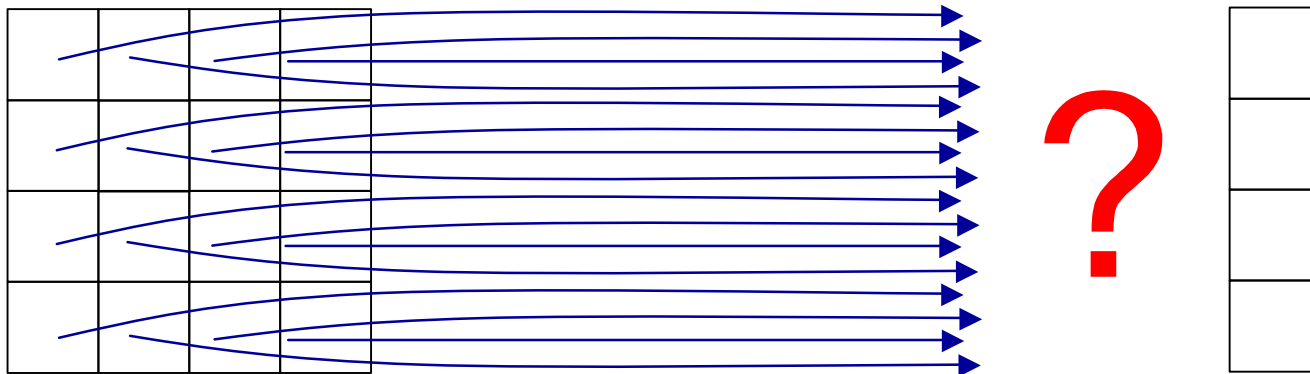
Simplifying the Storage Mapping

- Programs use arrays, not associative maps
 - If size decreases, need to specify which elements are mapped to the same location



Simplifying the Storage Mapping

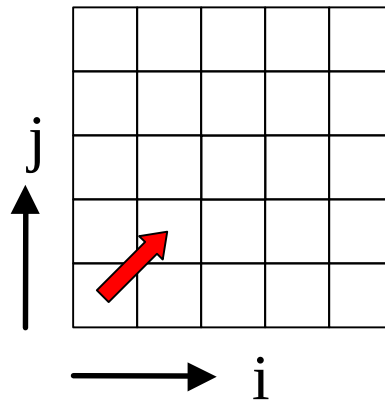
- Programs use arrays, not associative maps
 - If size decreases, need to specify which elements are mapped to the same location



Occupancy Vectors (Strout et al.)

- Specifies unit of overwriting within an array
- Locations collapsed if separated by a multiple of $\vec{v}$

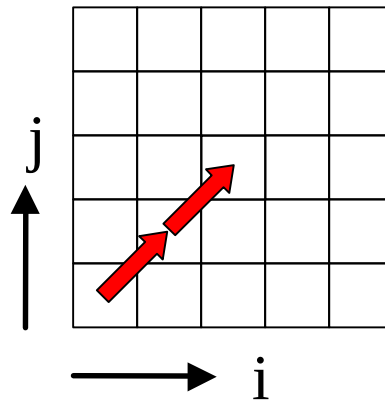
$$\vec{v} = (1, 1)$$



Occupancy Vectors (Strout et al.)

- Specifies unit of overwriting within an array
- Locations collapsed if separated by a multiple of $\vec{v}$

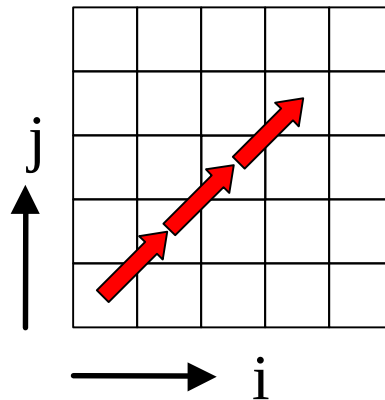
$$\vec{v} = (1, 1)$$



Occupancy Vectors (Strout et al.)

- Specifies unit of overwriting within an array
- Locations collapsed if separated by a multiple of $\vec{v}$

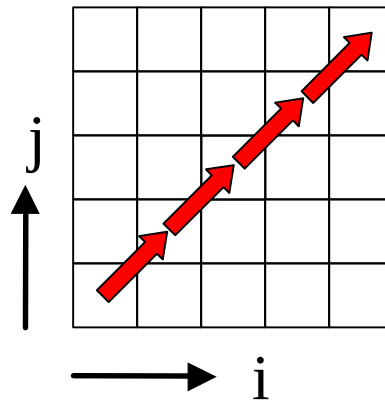
$$\vec{v} = (1, 1)$$



Occupancy Vectors (Strout et al.)

- Specifies unit of overwriting within an array
- Locations collapsed if separated by a multiple of $\vec{v}$

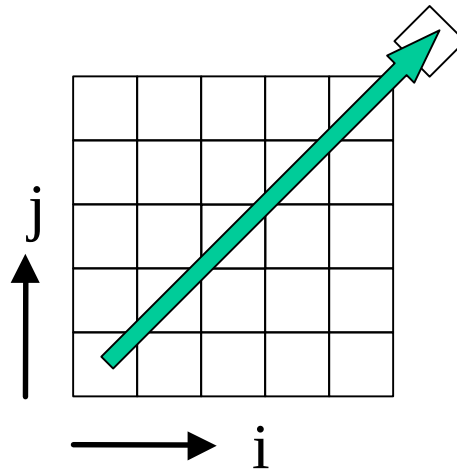
$$\vec{v} = (1, 1)$$



Occupancy Vectors (Strout et al.)

- Specifies unit of overwriting within an array
- Locations collapsed if separated by a multiple of $\vec{v}$

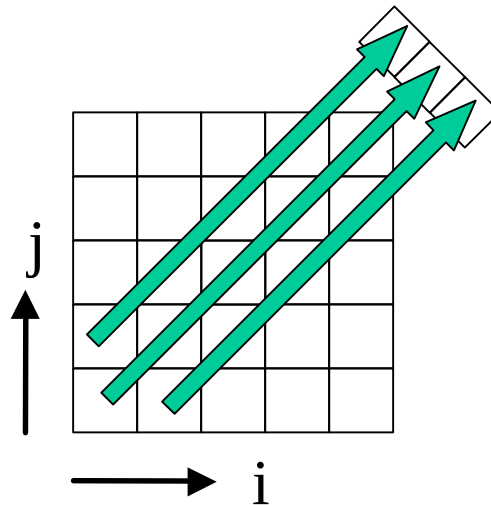
$$\vec{v} = (1, 1)$$



Occupancy Vectors (Strout et al.)

- Specifies unit of overwriting within an array
- Locations collapsed if separated by a multiple of $\vec{v}$

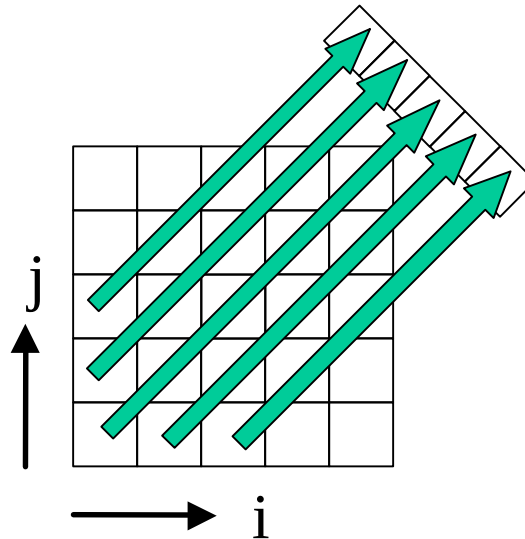
$$\vec{v} = (1, 1)$$



Occupancy Vectors (Strout et al.)

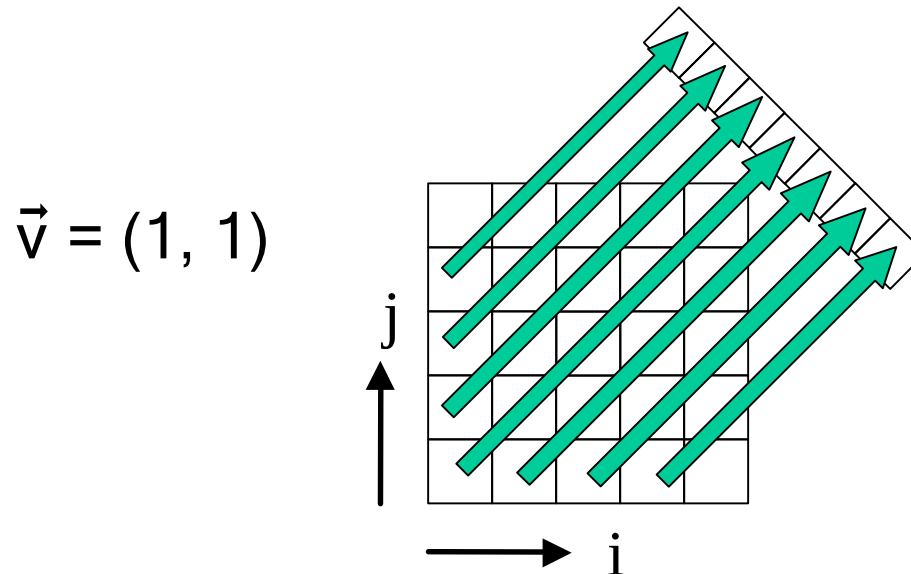
- Specifies unit of overwriting within an array
- Locations collapsed if separated by a multiple of $\vec{v}$

$$\vec{v} = (1, 1)$$



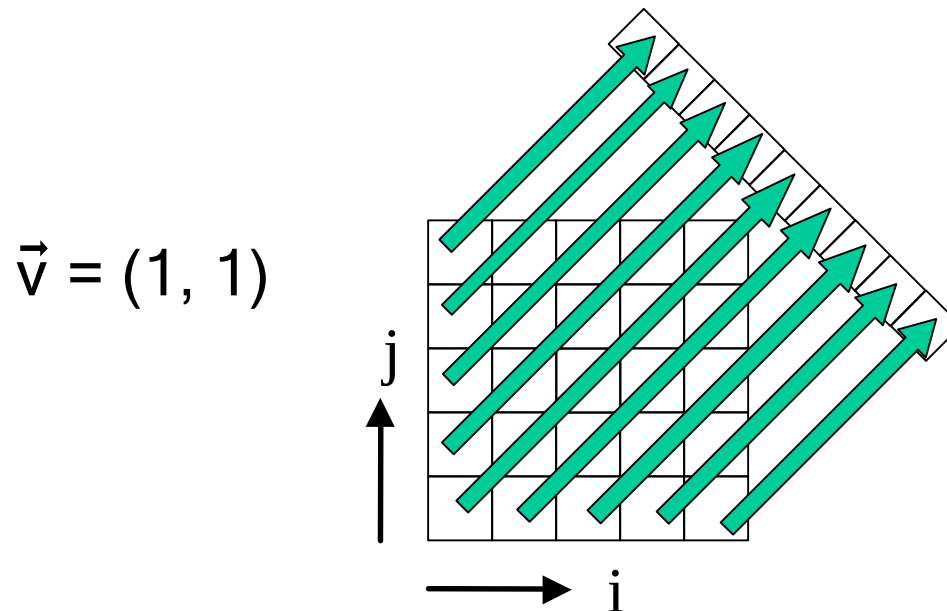
Occupancy Vectors (Strout et al.)

- Specifies unit of overwriting within an array
- Locations collapsed if separated by a multiple of $\vec{v}$



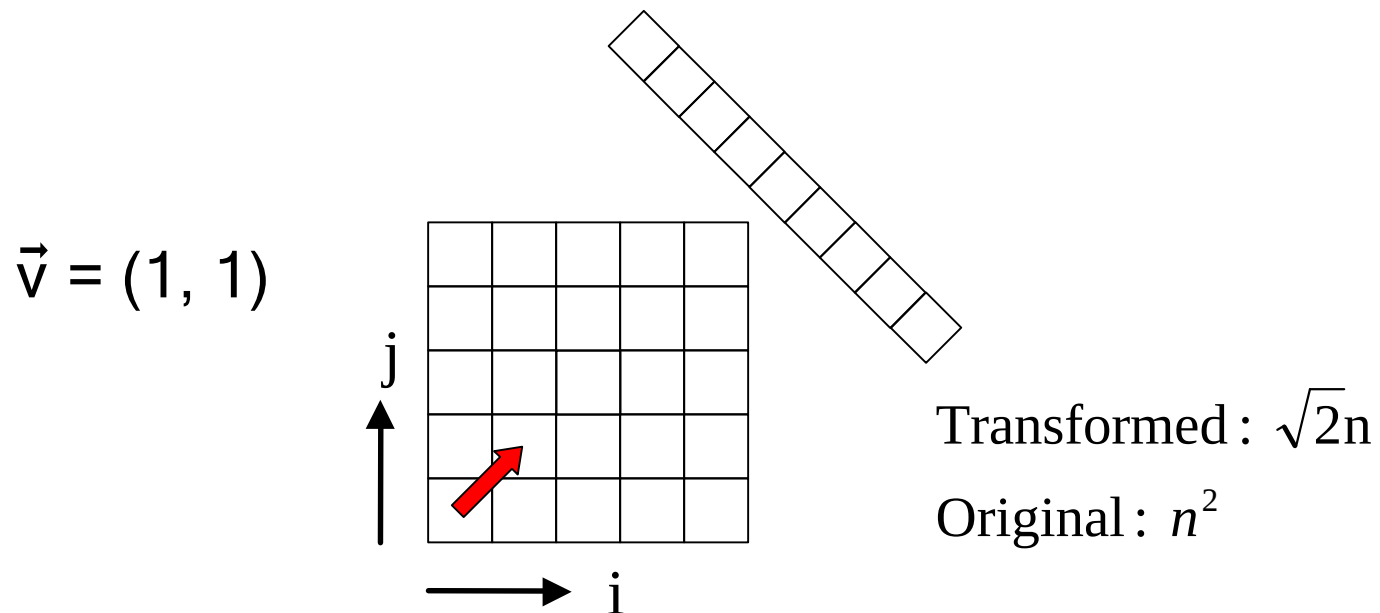
Occupancy Vectors (Strout et al.)

- Specifies unit of overwriting within an array
- Locations collapsed if separated by a multiple of $\vec{v}$



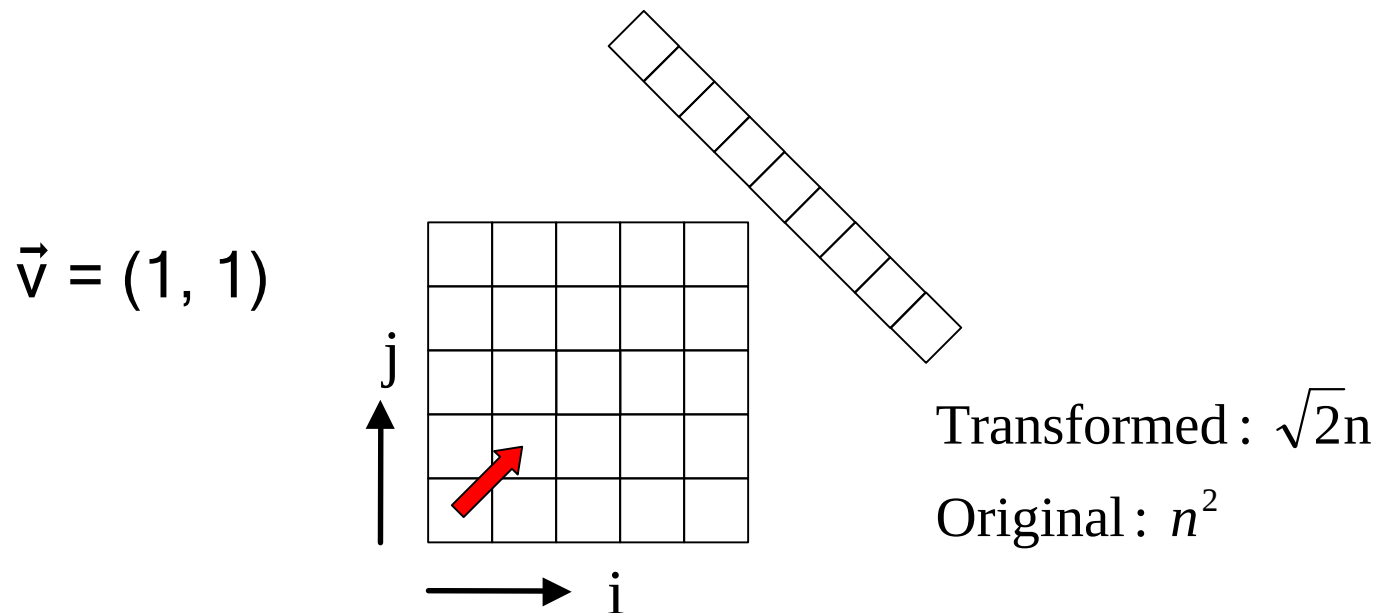
Occupancy Vectors (Strout et al.)

- Specifies unit of overwriting within an array
- Locations collapsed if separated by a multiple of $\vec{v}$



Occupancy Vectors (Strout et al.)

- For a given schedule, $\vec{v}$ is valid if semantics are unchanged using transformed array
- Shorter vectors require less storage

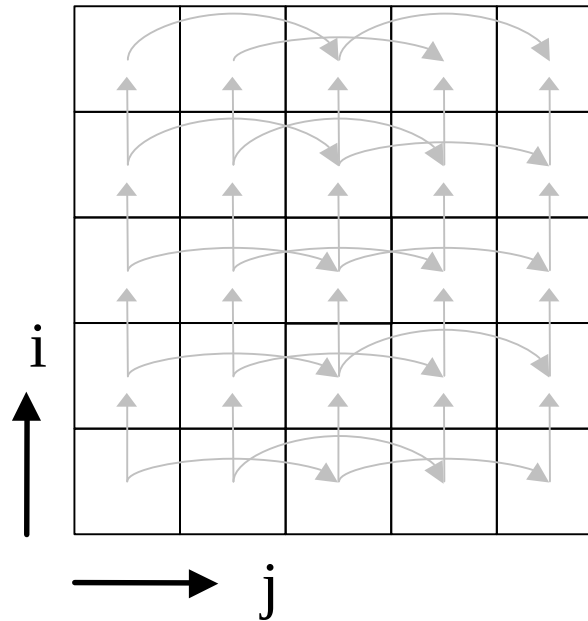


Outline

- Abstract problem
- Simplifications
- Concrete problem
- Solution Method
- Conclusions

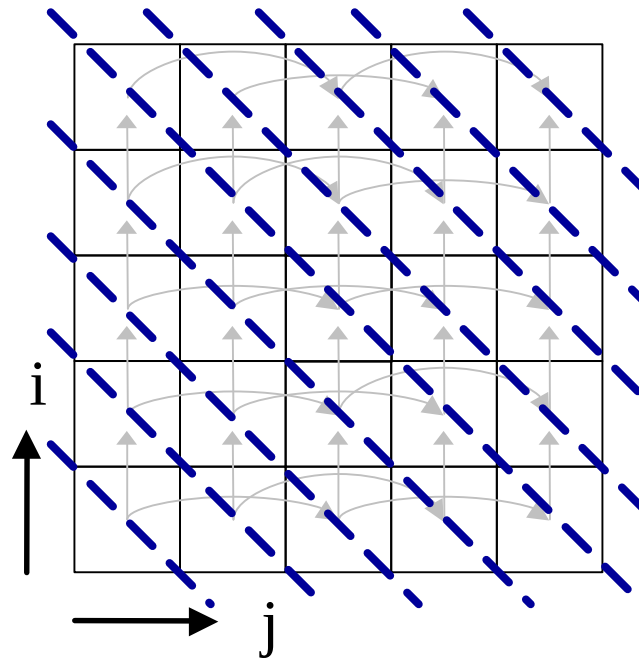
Answering Question #1

- Given $\theta(i, j) = i + j$, what is the shortest valid occupancy vector $\vec{v}$?



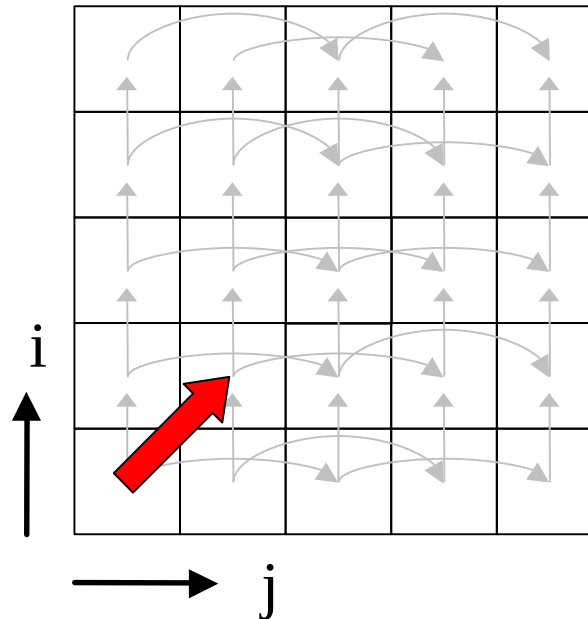
Answering Question #1

- Given $\theta(i, j) = i + j$, what is the shortest valid occupancy vector $\vec{v}$?



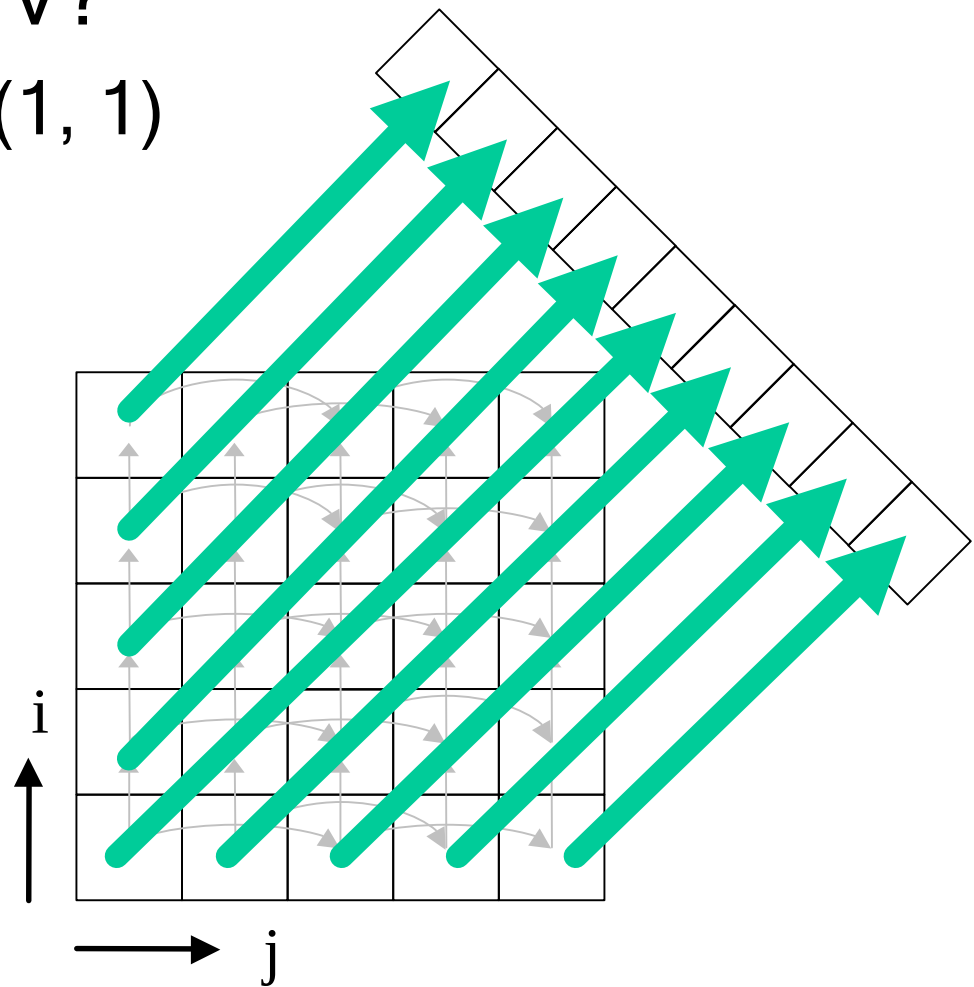
Answering Question #1

- Given $\theta(i, j) = i + j$, what is the shortest valid occupancy vector $\vec{v}$?
 - ➔ Solution: $\vec{v} = (1, 1)$



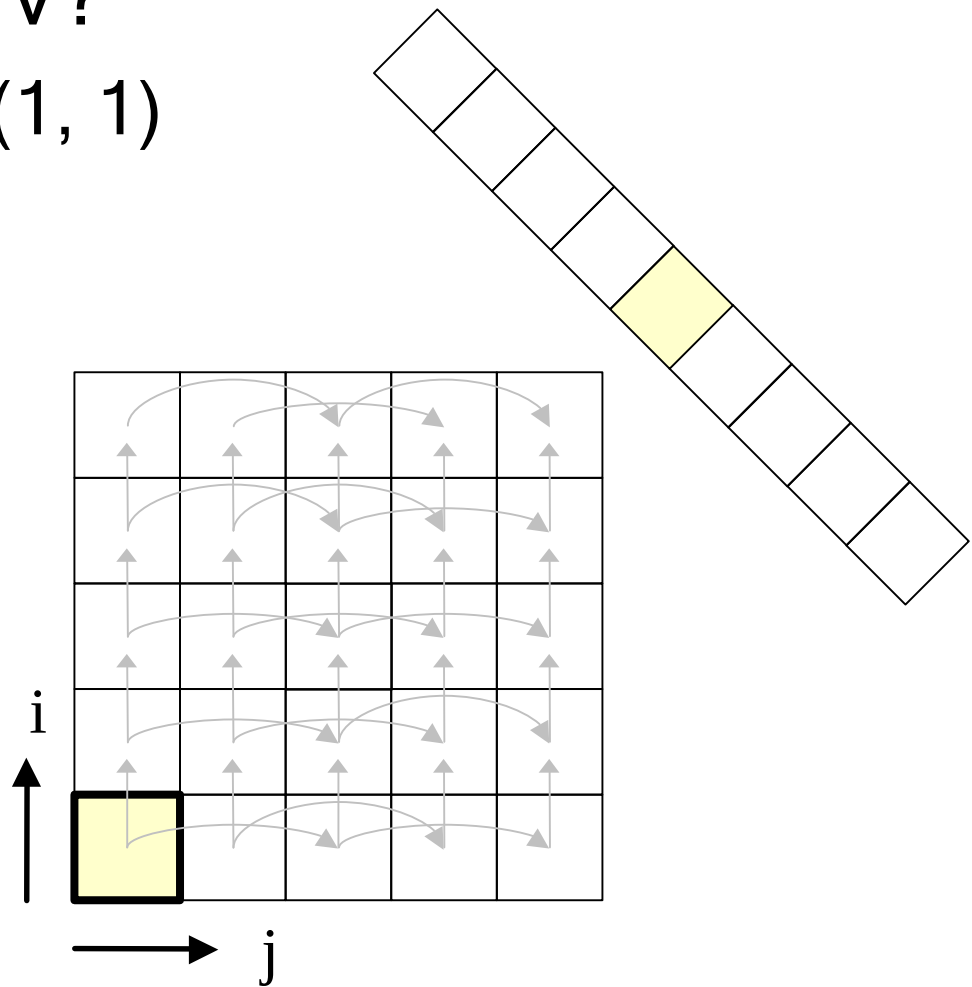
Answering Question #1

- Given $\theta(i, j) = i + j$, what is the shortest valid occupancy vector $\vec{v}$?
 - ➔ Solution: $\vec{v} = (1, 1)$



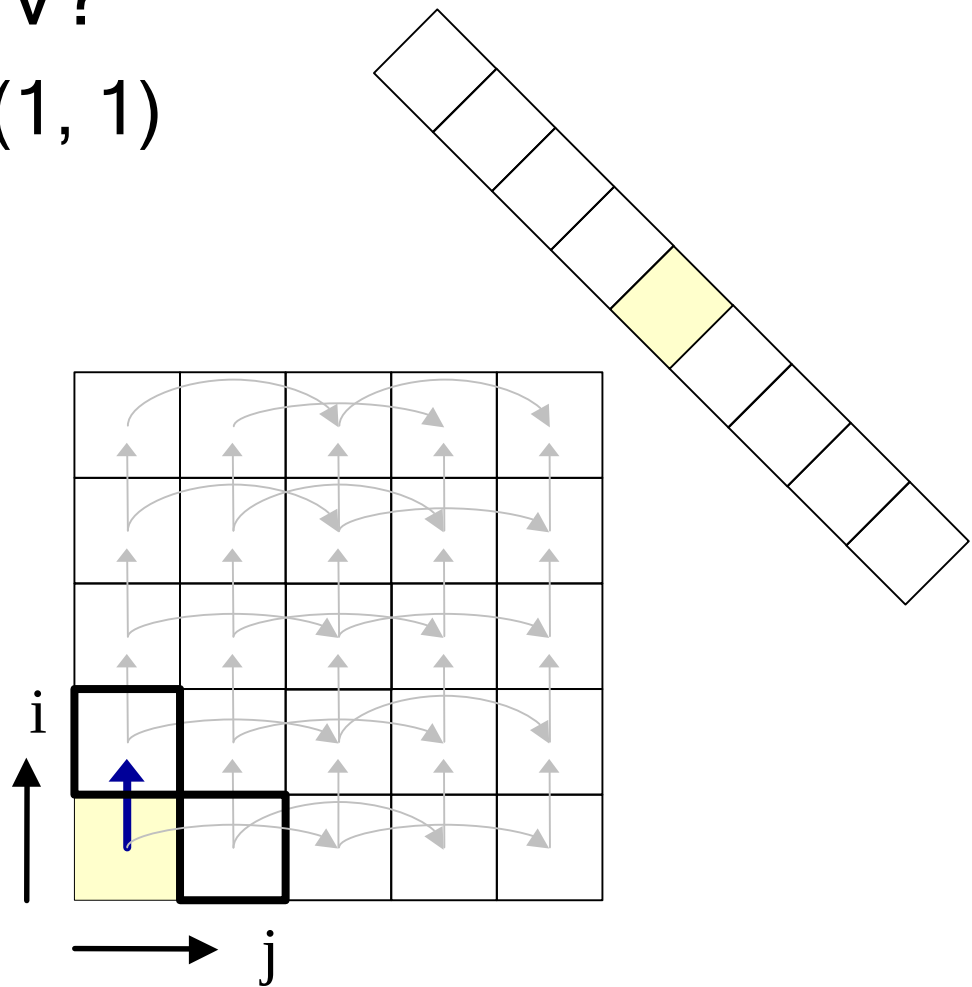
Answering Question #1

- Given $\theta(i, j) = i + j$, what is the shortest valid occupancy vector $\vec{v}$?
 - ➔ Solution: $\vec{v} = (1, 1)$



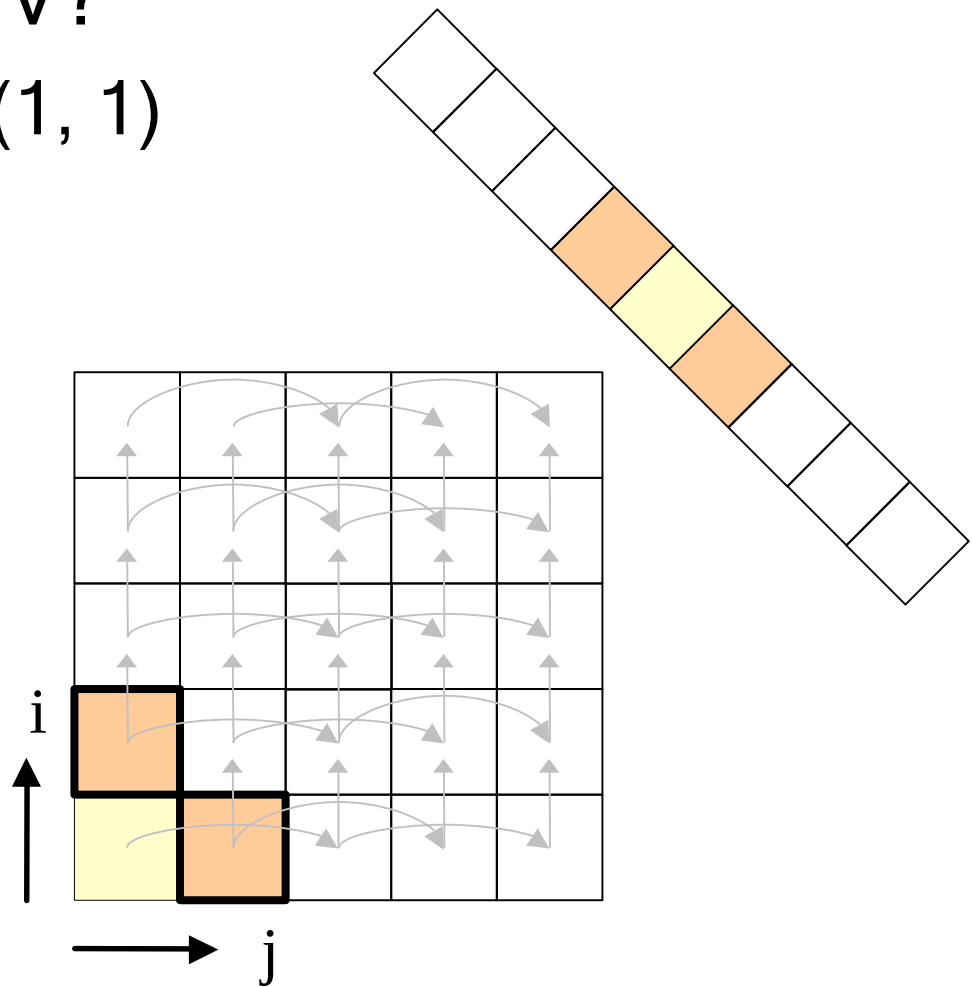
Answering Question #1

- Given $\theta(i, j) = i + j$, what is the shortest valid occupancy vector $\vec{v}$?
 - ➔ Solution: $\vec{v} = (1, 1)$



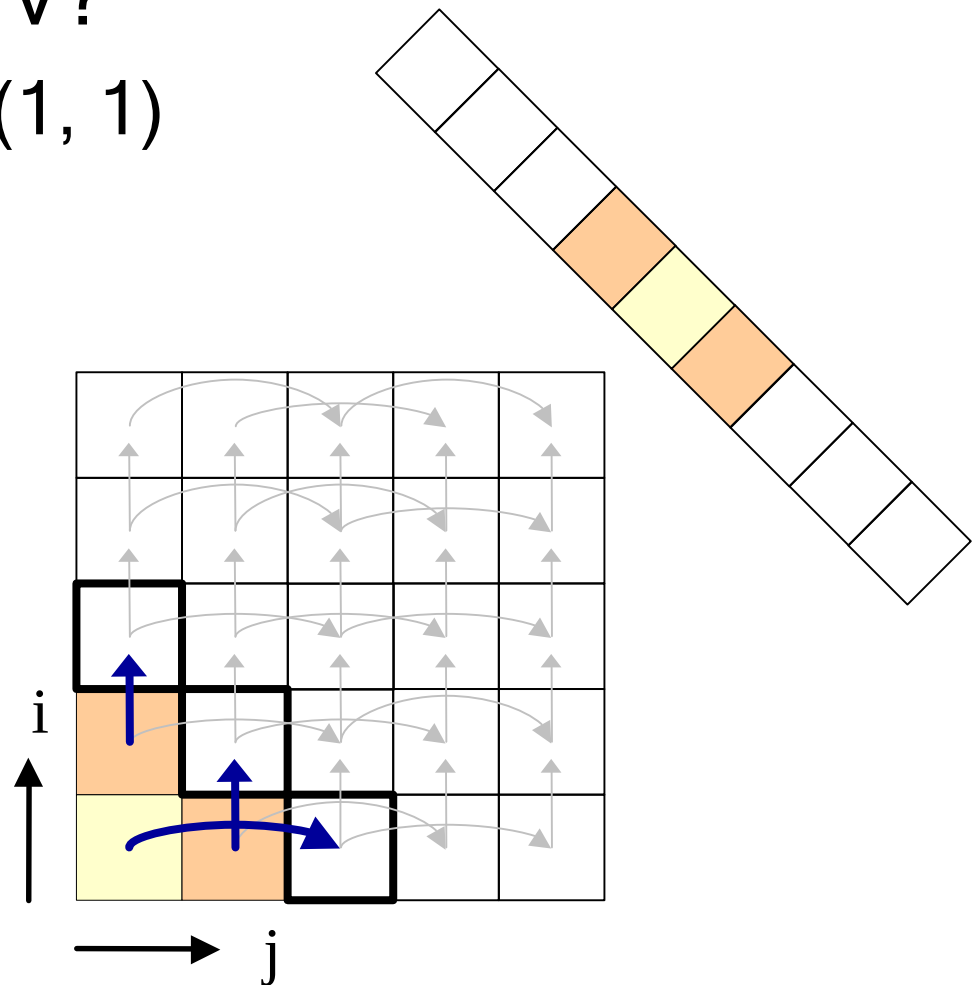
Answering Question #1

- Given $\theta(i, j) = i + j$, what is the shortest valid occupancy vector $\vec{v}$?
 - ➔ Solution: $\vec{v} = (1, 1)$



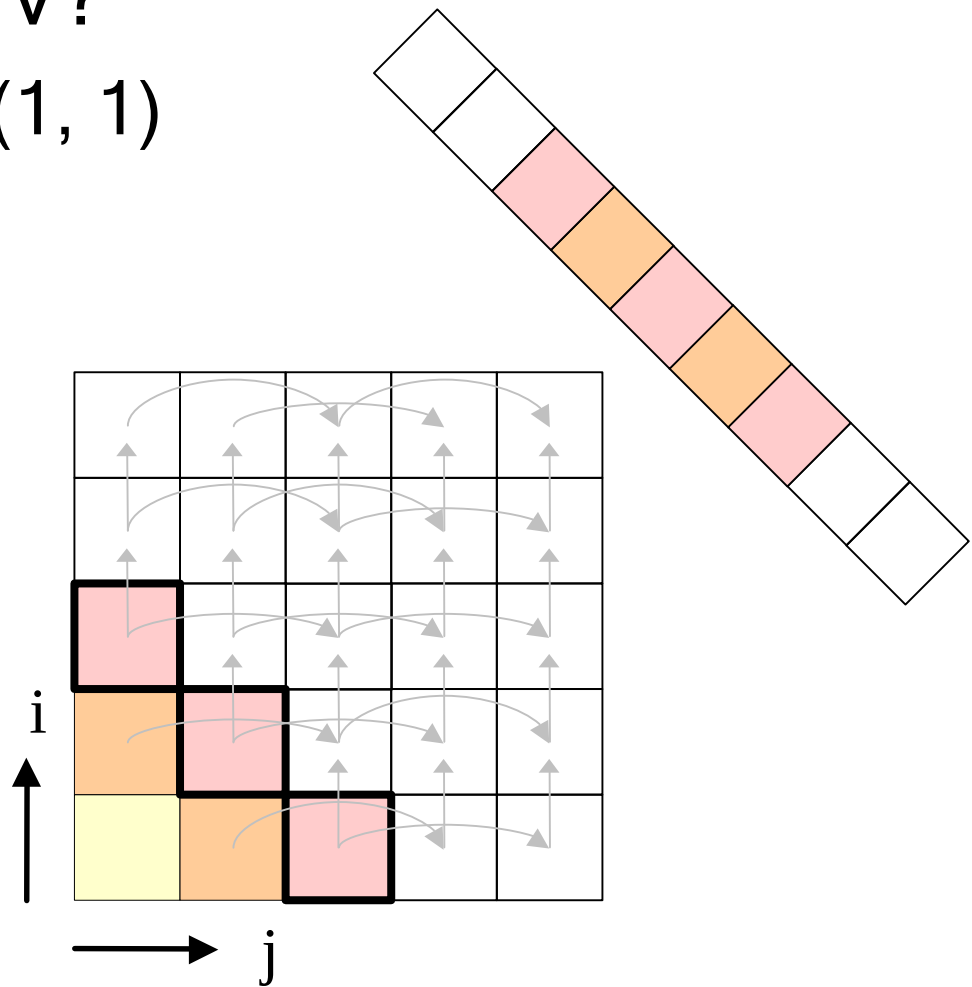
Answering Question #1

- Given $\theta(i, j) = i + j$, what is the shortest valid occupancy vector $\vec{v}$?
 - ➔ Solution: $\vec{v} = (1, 1)$



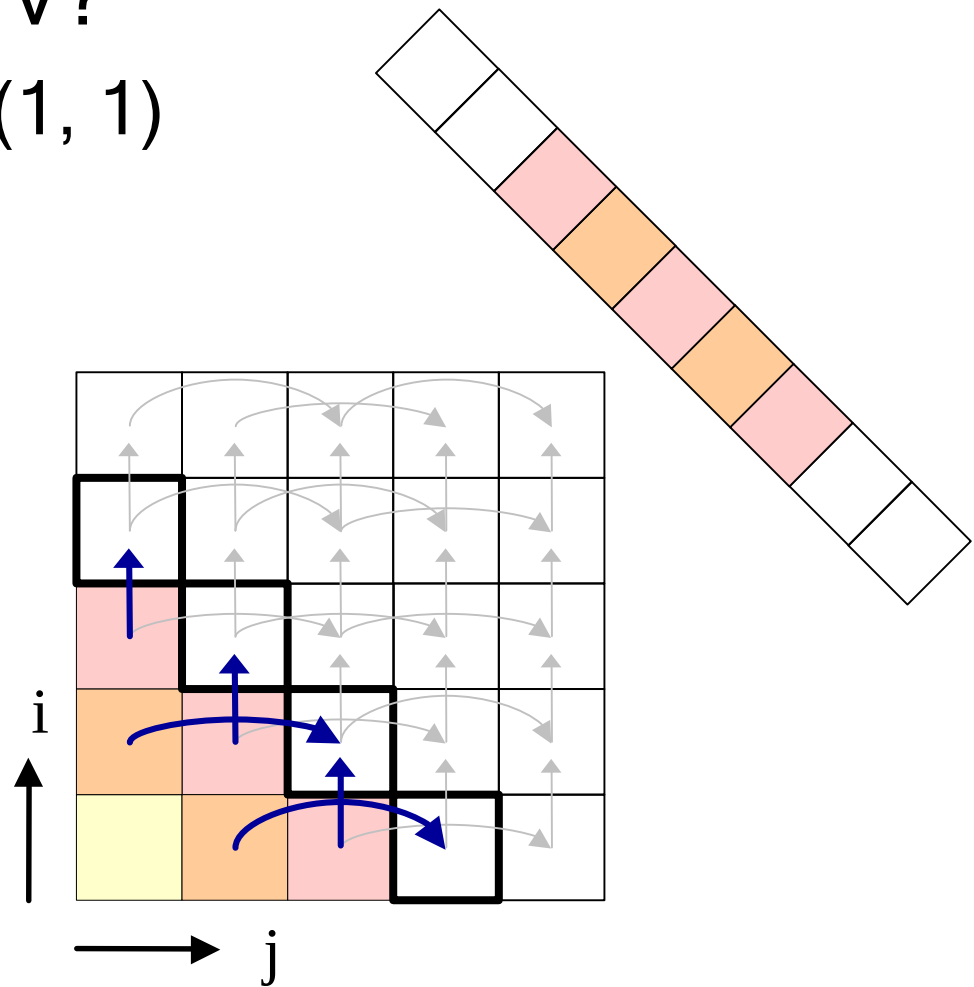
Answering Question #1

- Given $\theta(i, j) = i + j$, what is the shortest valid occupancy vector $\vec{v}$?
 - ➔ Solution: $\vec{v} = (1, 1)$



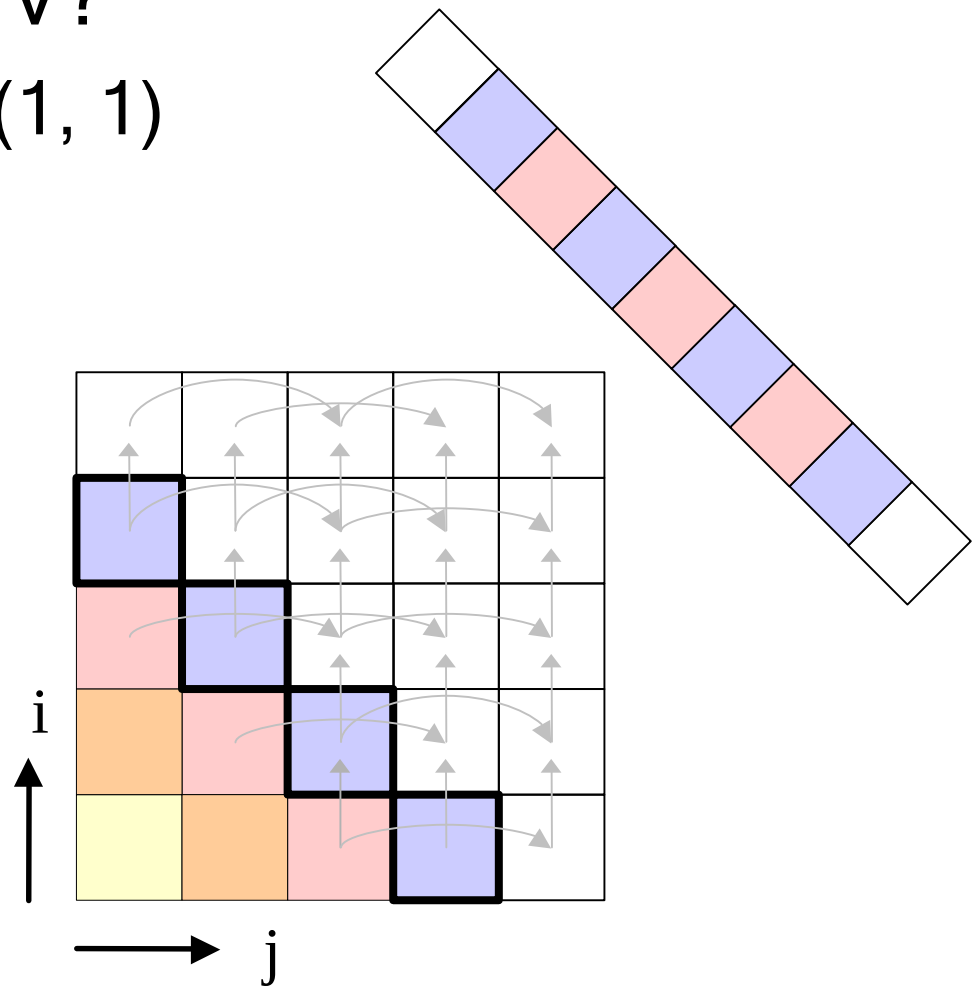
Answering Question #1

- Given $\theta(i, j) = i + j$, what is the shortest valid occupancy vector $\vec{v}$?
 - ➔ Solution: $\vec{v} = (1, 1)$



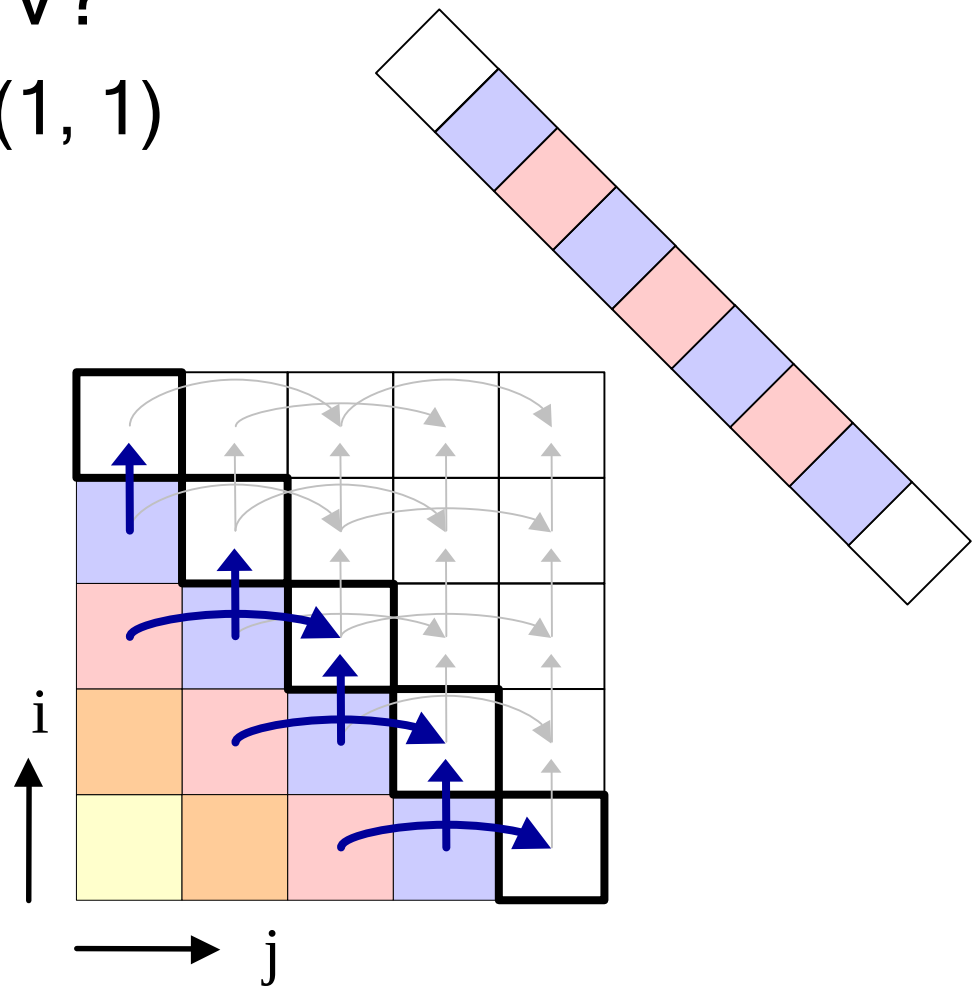
Answering Question #1

- Given $\theta(i, j) = i + j$, what is the shortest valid occupancy vector $\vec{v}$?
 - ➔ Solution: $\vec{v} = (1, 1)$



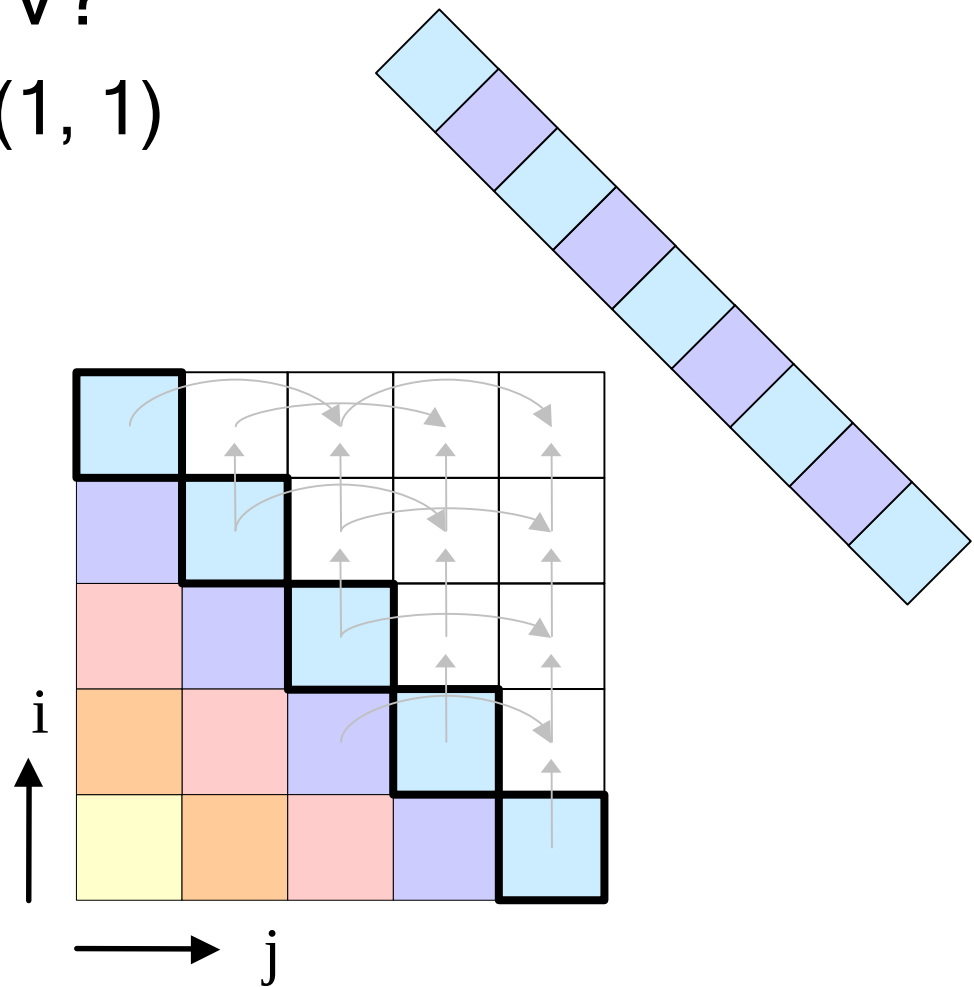
Answering Question #1

- Given $\theta(i, j) = i + j$, what is the shortest valid occupancy vector $\vec{v}$?
 - ➔ Solution: $\vec{v} = (1, 1)$



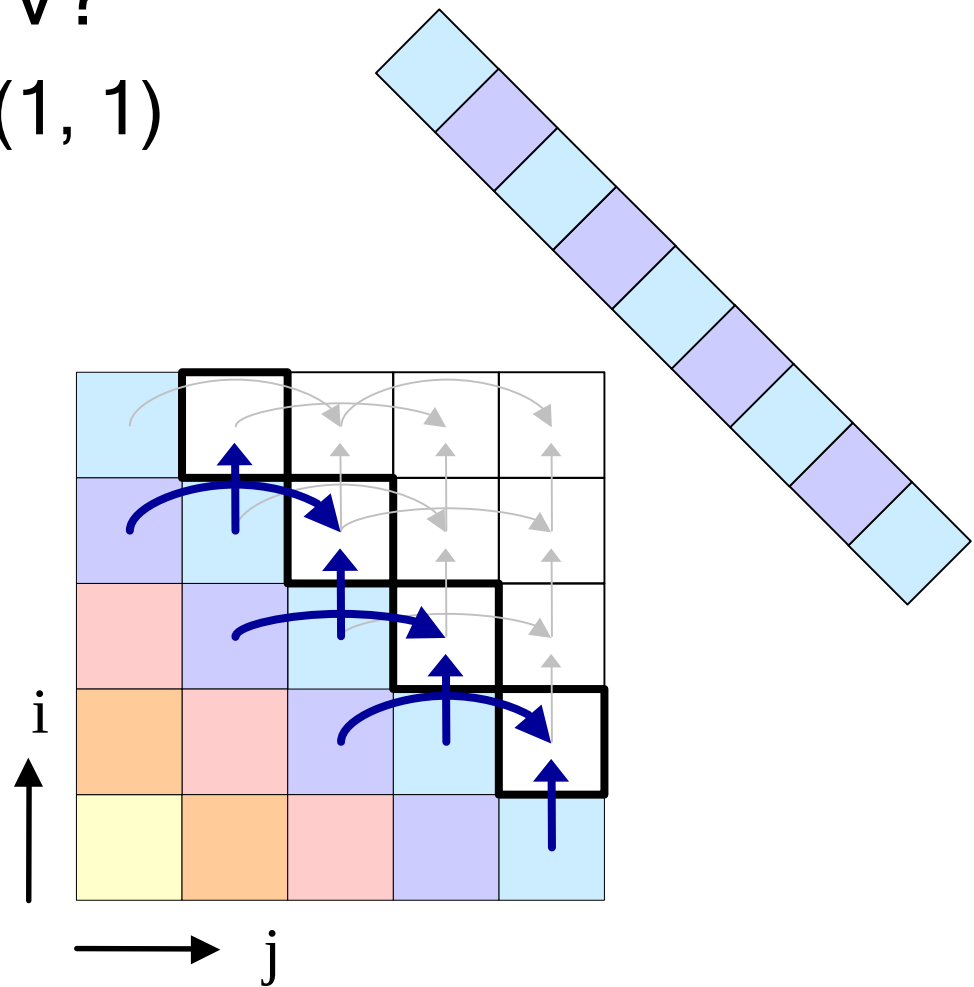
Answering Question #1

- Given $\theta(i, j) = i + j$, what is the shortest valid occupancy vector $\vec{v}$?
→ Solution: $\vec{v} = (1, 1)$



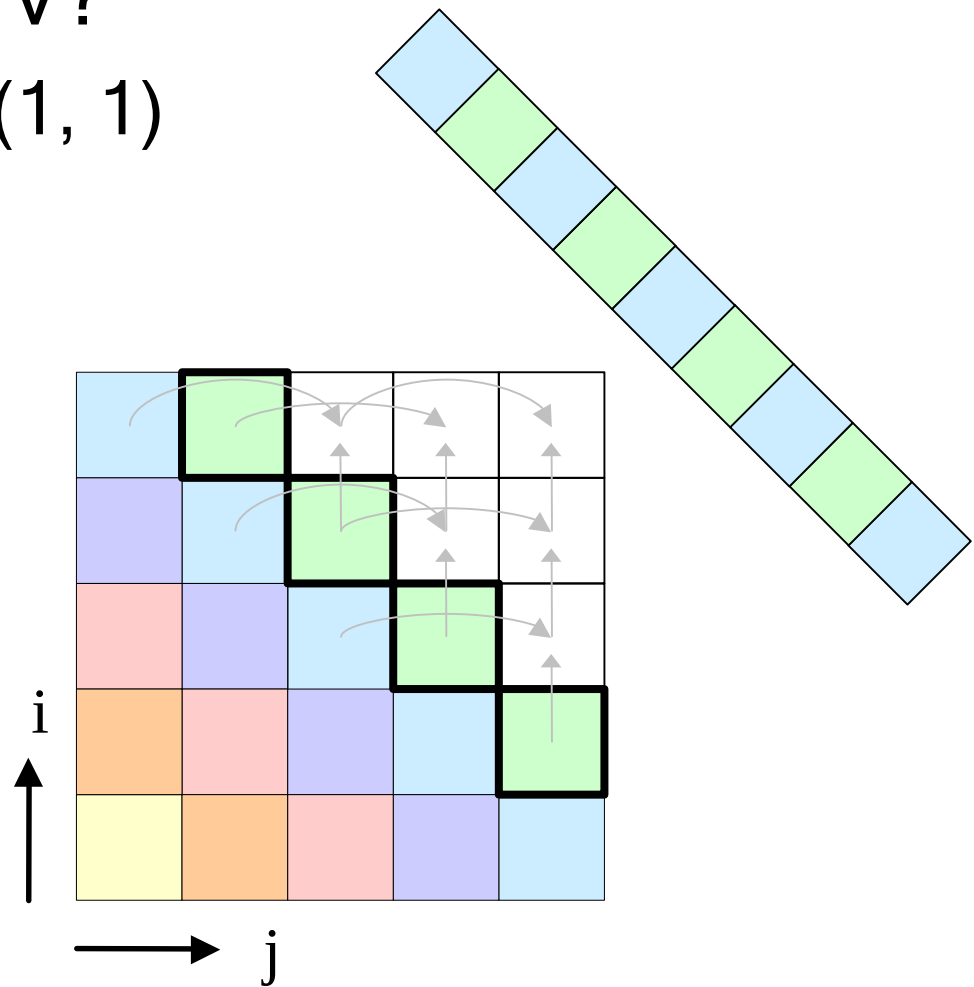
Answering Question #1

- Given $\theta(i, j) = i + j$, what is the shortest valid occupancy vector $\vec{v}$?
 - ➔ Solution: $\vec{v} = (1, 1)$



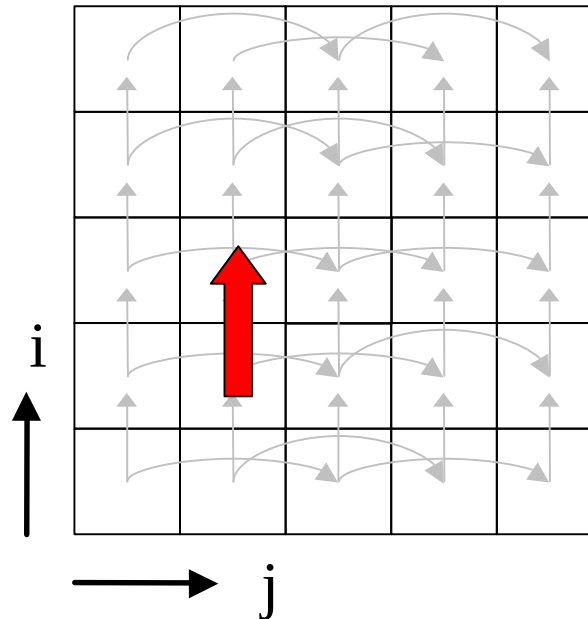
Answering Question #1

- Given $\theta(i, j) = i + j$, what is the shortest valid occupancy vector $\vec{v}$?
 - ➔ Solution: $\vec{v} = (1, 1)$



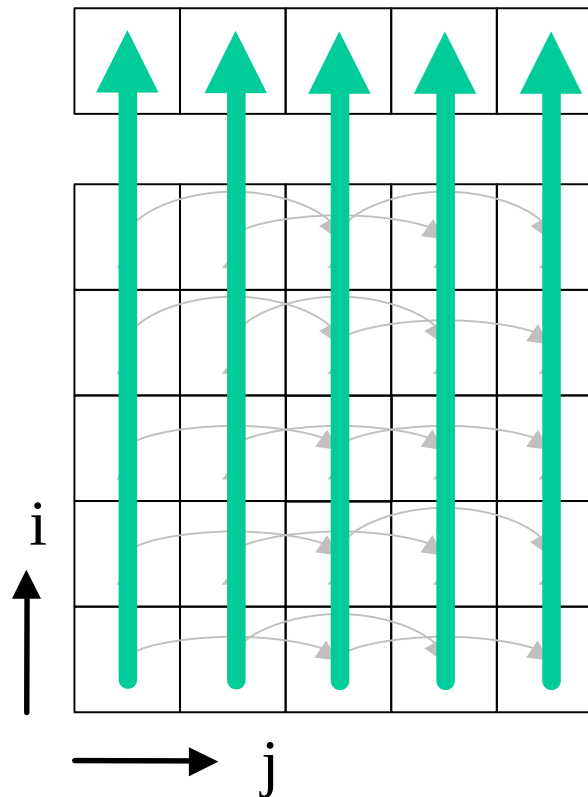
Answering Question #1

- Given $\theta(i, j) = i + j$, what is the shortest valid occupancy vector $\vec{v}$?
 - ➔ Why not $\vec{v} = (0, 1)$?



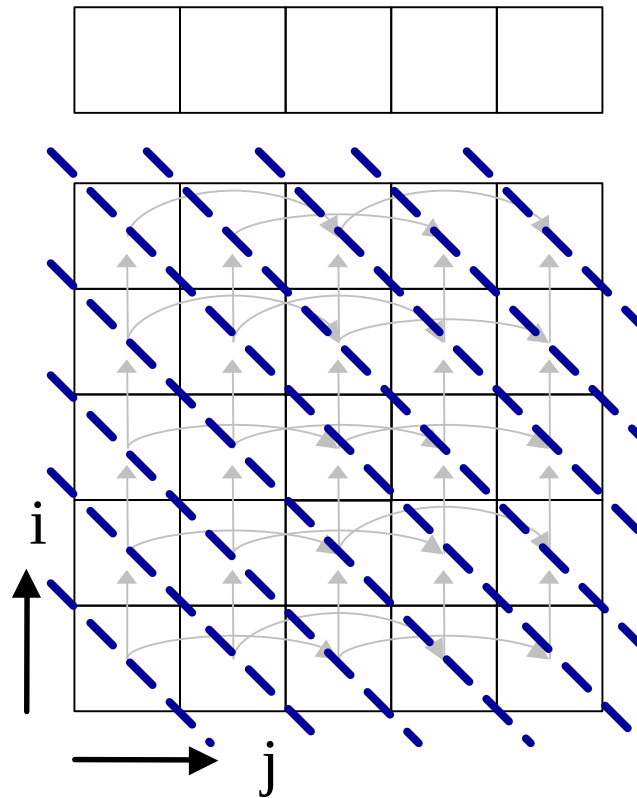
Answering Question #1

- Given $\theta(i, j) = i + j$, what is the shortest valid occupancy vector $\vec{v}$?
 - ➔ Why not $\vec{v} = (0, 1)$?



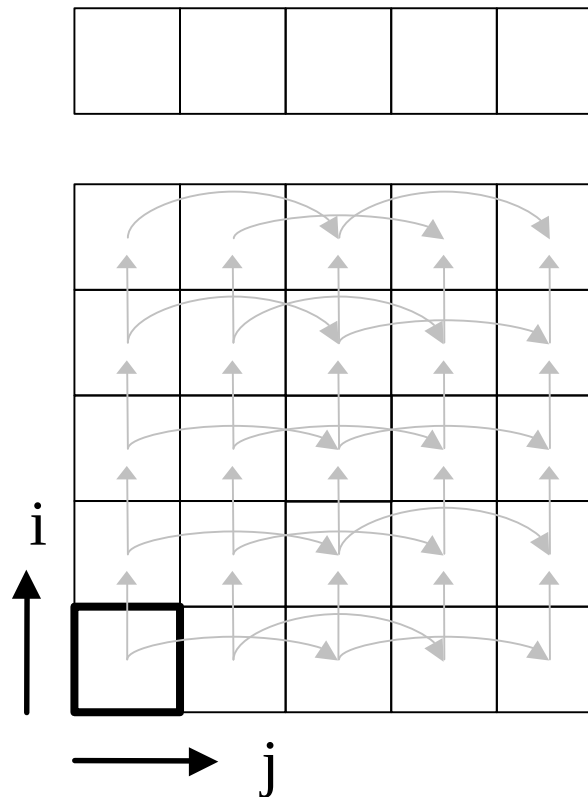
Answering Question #1

- Given $\theta(i, j) = i + j$, what is the shortest valid occupancy vector $\vec{v}$?
 - ➔ Why not $\vec{v} = (0, 1)$?



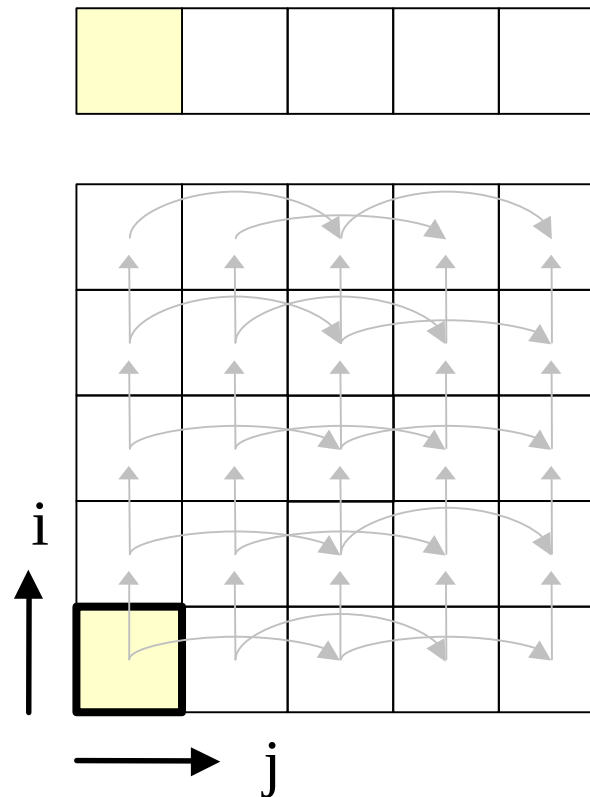
Answering Question #1

- Given $\theta(i, j) = i + j$, what is the shortest valid occupancy vector $\vec{v}$?
 - ➔ Why not $\vec{v} = (0, 1)$?



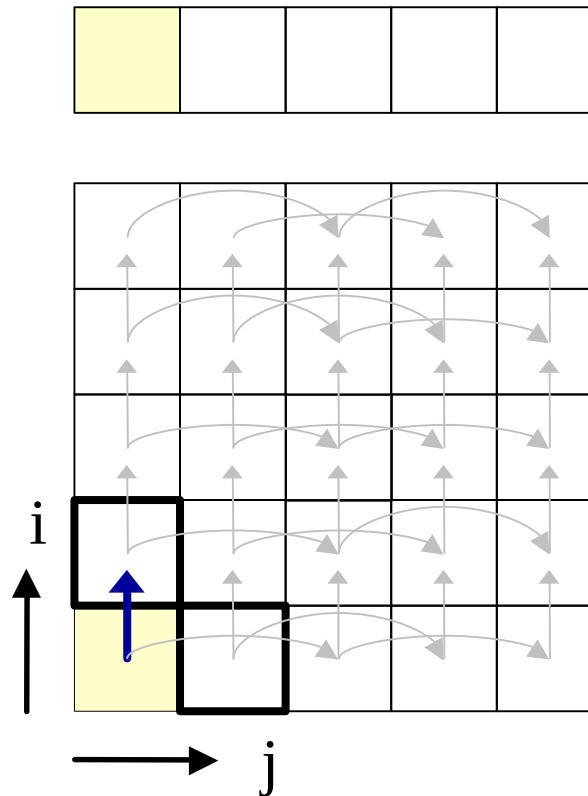
Answering Question #1

- Given $\theta(i, j) = i + j$, what is the shortest valid occupancy vector $\vec{v}$?
 - ➔ Why not $\vec{v} = (0, 1)$?



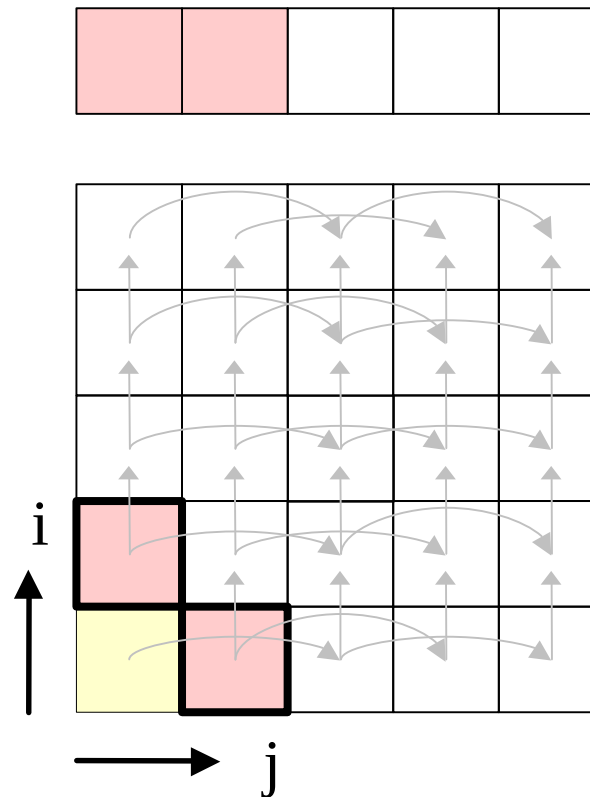
Answering Question #1

- Given $\theta(i, j) = i + j$, what is the shortest valid occupancy vector $\vec{v}$?
 - ➔ Why not $\vec{v} = (0, 1)$?



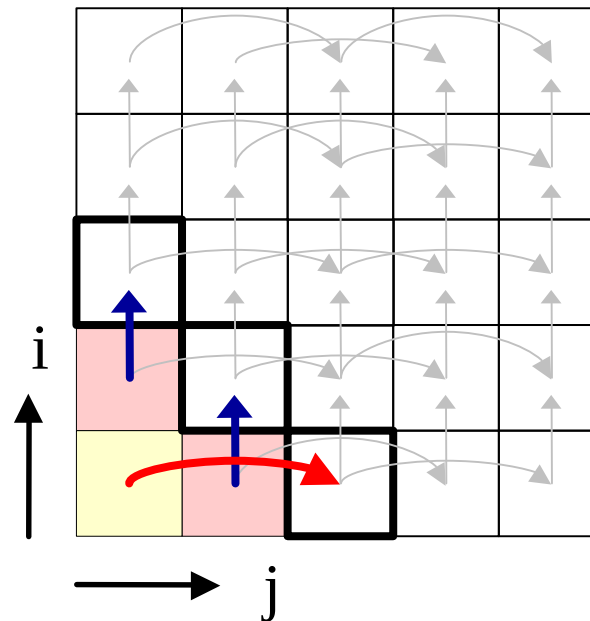
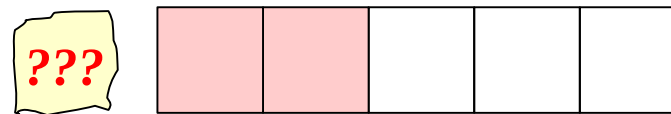
Answering Question #1

- Given $\theta(i, j) = i + j$, what is the shortest valid occupancy vector $\vec{v}$?
 - ➔ Why not $\vec{v} = (0, 1)$?



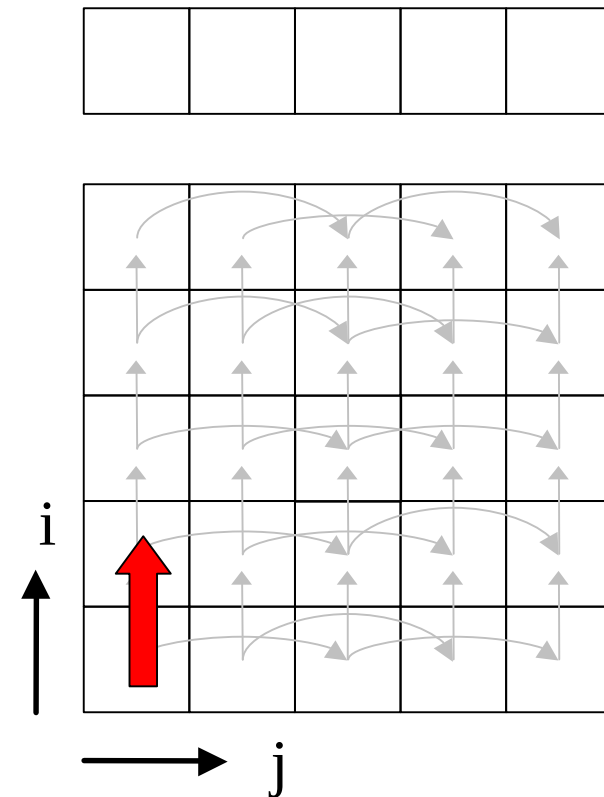
Answering Question #1

- Given $\theta(i, j) = i + j$, what is the shortest valid occupancy vector $\vec{v}$?
 - ➔ Why not $\vec{v} = (0, 1)$?



Answering Question #2

- Given $\vec{v} = (0, 1)$, what is the range of valid schedules θ ?



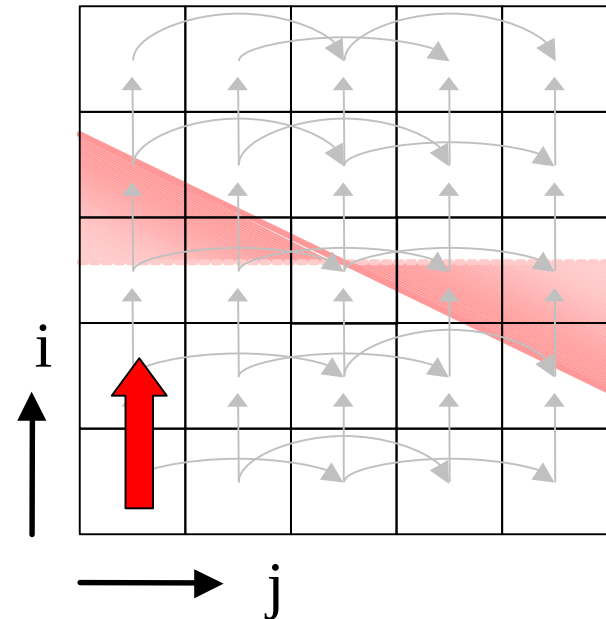
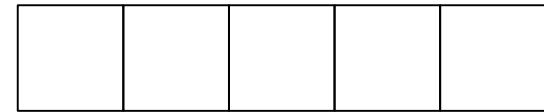
Answering Question #2

- Given $\vec{v} = (0, 1)$, what is the range of valid schedules θ ?

→ $\theta(i, j)$ is between:

$$\theta(i, j) = 2 * i + j \text{ (inclusive)}$$

$$\theta(i, j) = i \text{ (exclusive)}$$



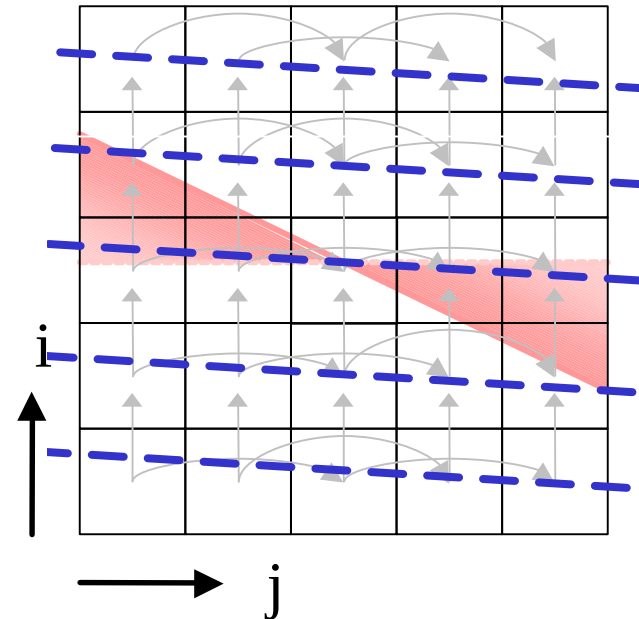
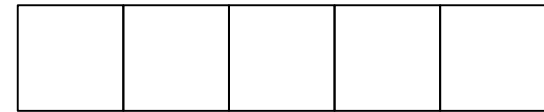
Answering Question #2

- Given $\vec{v} = (0, 1)$, what is the range of valid schedules θ ?

→ $\theta(i, j)$ is between:

$$\theta(i, j) = 2 * i + j \text{ (inclusive)}$$

$$\theta(i, j) = i \text{ (exclusive)}$$



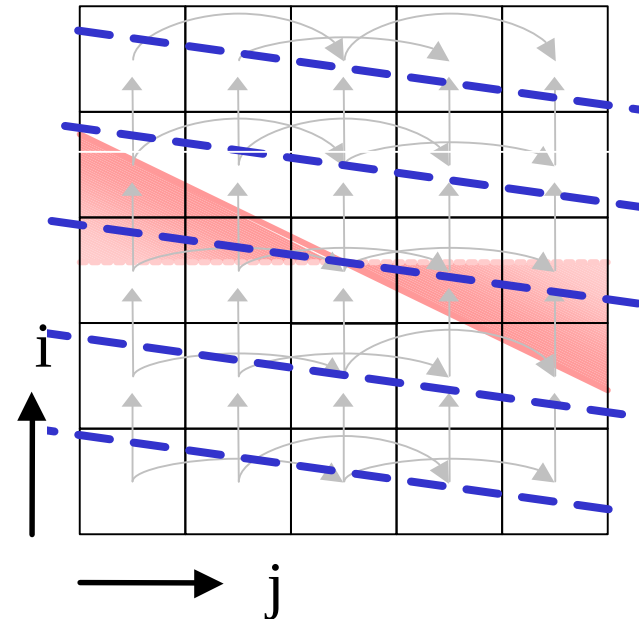
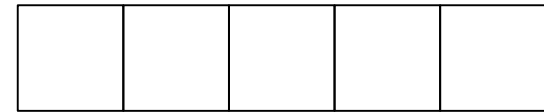
Answering Question #2

- Given $\vec{v} = (0, 1)$, what is the range of valid schedules θ ?

→ $\theta(i, j)$ is between:

$$\theta(i, j) = 2 * i + j \text{ (inclusive)}$$

$$\theta(i, j) = i \text{ (exclusive)}$$



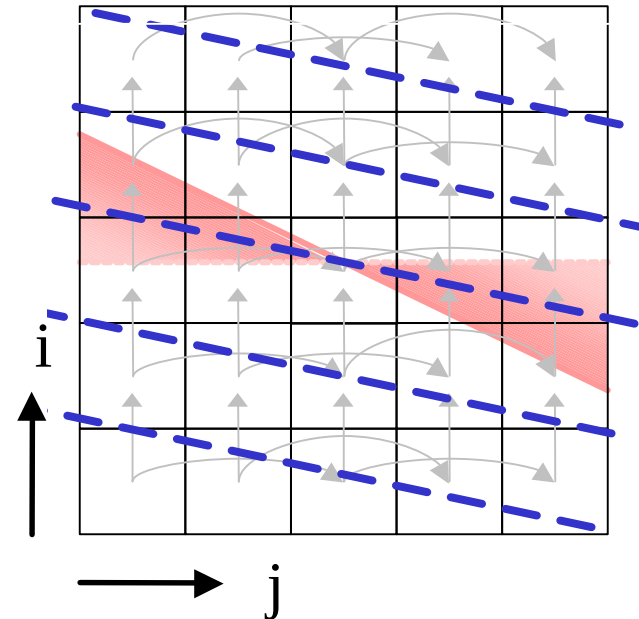
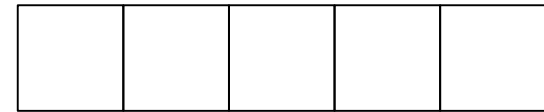
Answering Question #2

- Given $\vec{v} = (0, 1)$, what is the range of valid schedules θ ?

→ $\theta(i, j)$ is between:

$$\theta(i, j) = 2 * i + j \text{ (inclusive)}$$

$$\theta(i, j) = i \text{ (exclusive)}$$



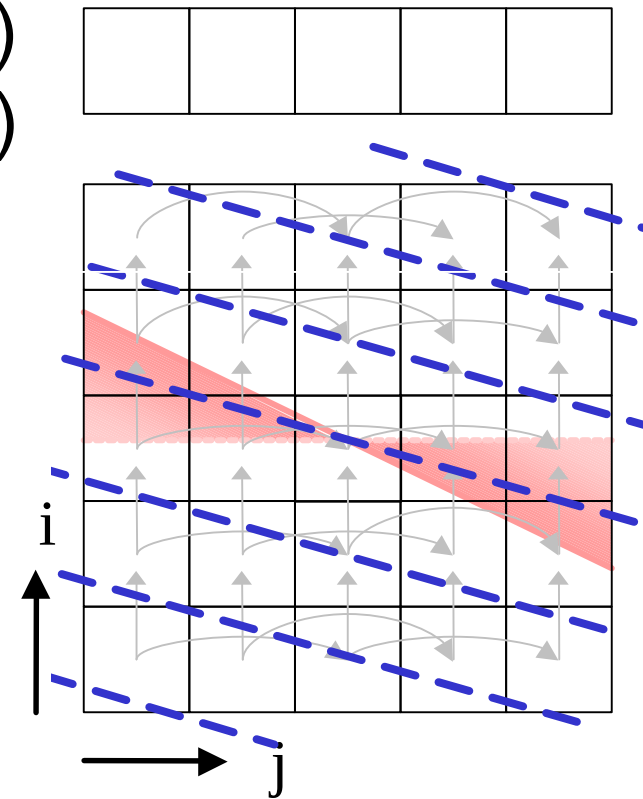
Answering Question #2

- Given $\vec{v} = (0, 1)$, what is the range of valid schedules θ ?

→ $\theta(i, j)$ is between:

$$\theta(i, j) = 2 * i + j \text{ (inclusive)}$$

$$\theta(i, j) = i \text{ (exclusive)}$$



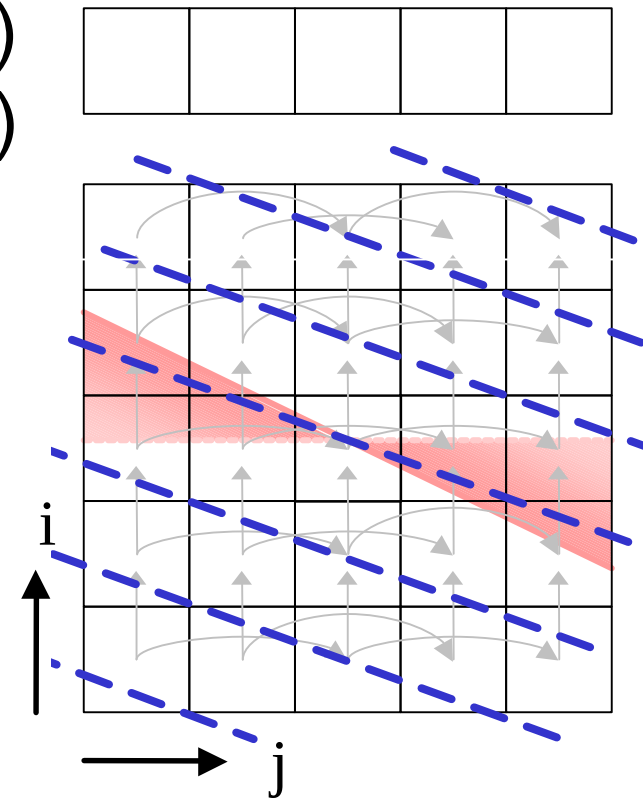
Answering Question #2

- Given $\vec{v} = (0, 1)$, what is the range of valid schedules θ ?

→ $\theta(i, j)$ is between:

$$\theta(i, j) = 2 * i + j \text{ (inclusive)}$$

$$\theta(i, j) = i \text{ (exclusive)}$$



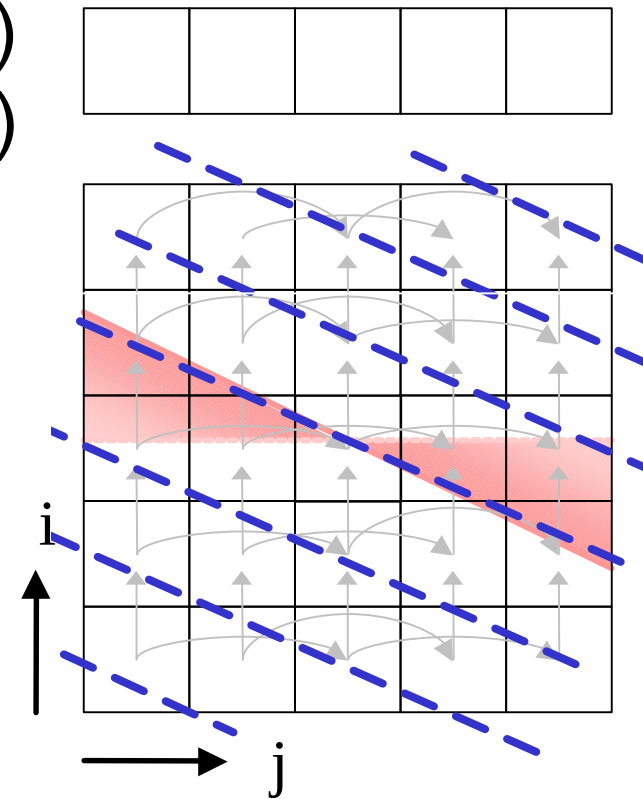
Answering Question #2

- Given $\vec{v} = (0, 1)$, what is the range of valid schedules θ ?

→ $\theta(i, j)$ is between:

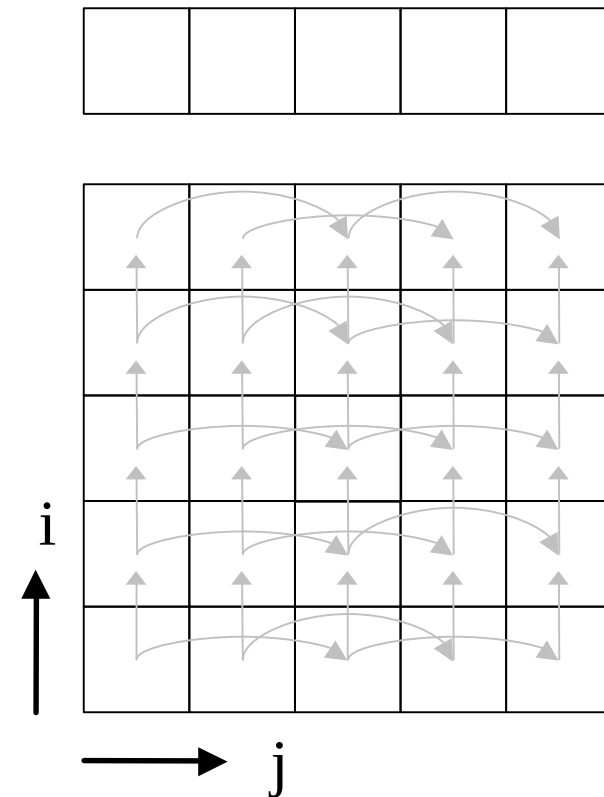
$$\theta(i, j) = 2 * i + j \text{ (inclusive)}$$

$$\theta(i, j) = i \text{ (exclusive)}$$



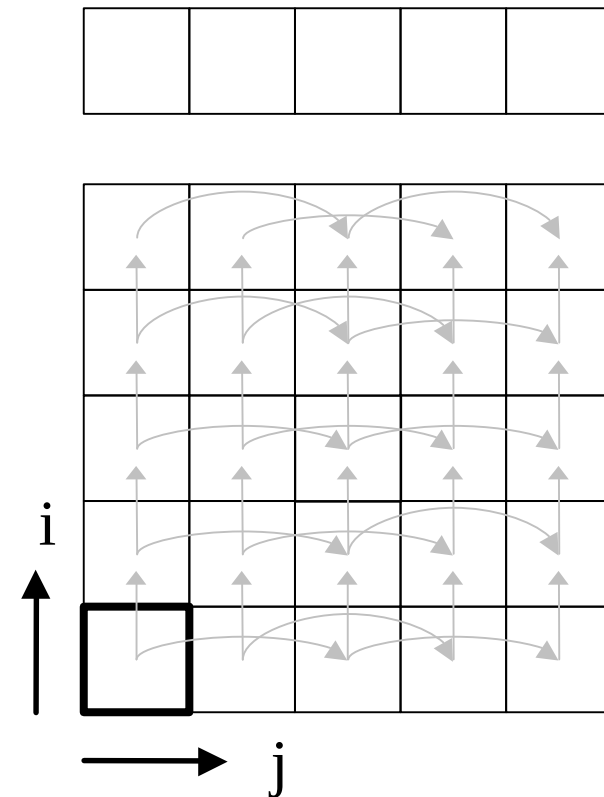
Answering Question #2

- Given $\vec{v} = (0, 1)$, what is the range of valid schedules θ ?
 - ➔ Lets try $\theta(i, j) = 2 * i + j$



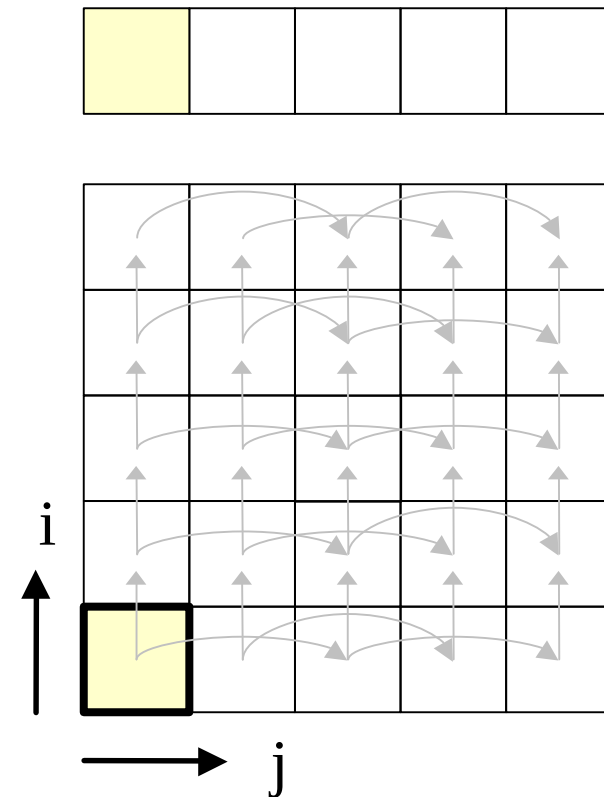
Answering Question #2

- Given $\vec{v} = (0, 1)$, what is the range of valid schedules θ ?
 - ➔ Lets try $\theta(i, j) = 2 * i + j$



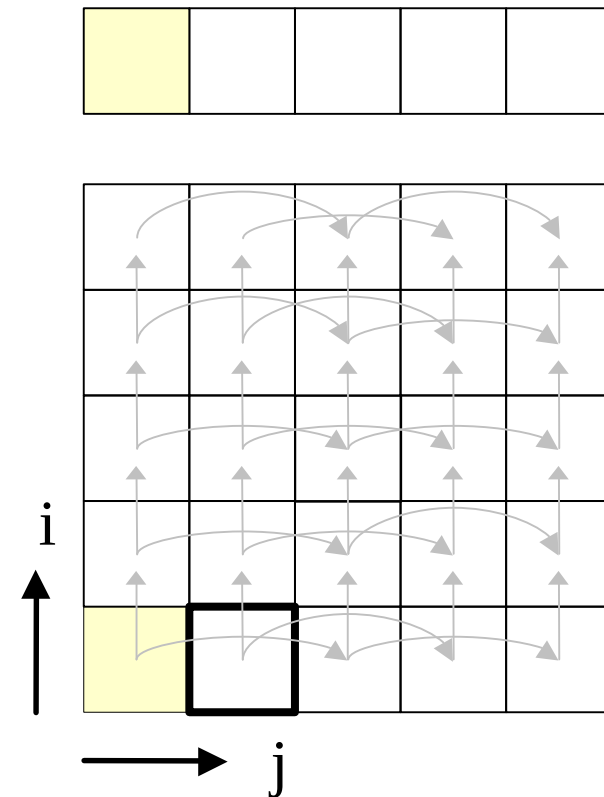
Answering Question #2

- Given $\vec{v} = (0, 1)$, what is the range of valid schedules θ ?
 - ➔ Lets try $\theta(i, j) = 2 * i + j$



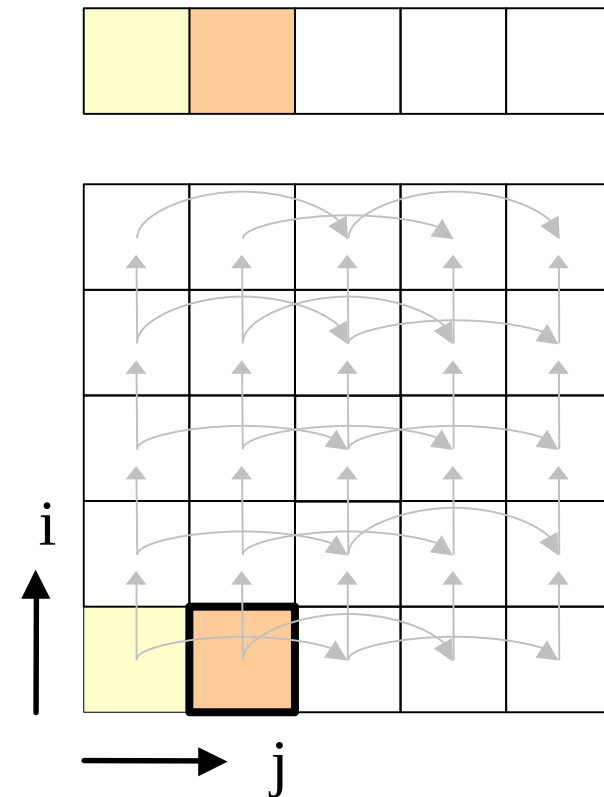
Answering Question #2

- Given $\vec{v} = (0, 1)$, what is the range of valid schedules θ ?
 - ➔ Lets try $\theta(i, j) = 2 * i + j$



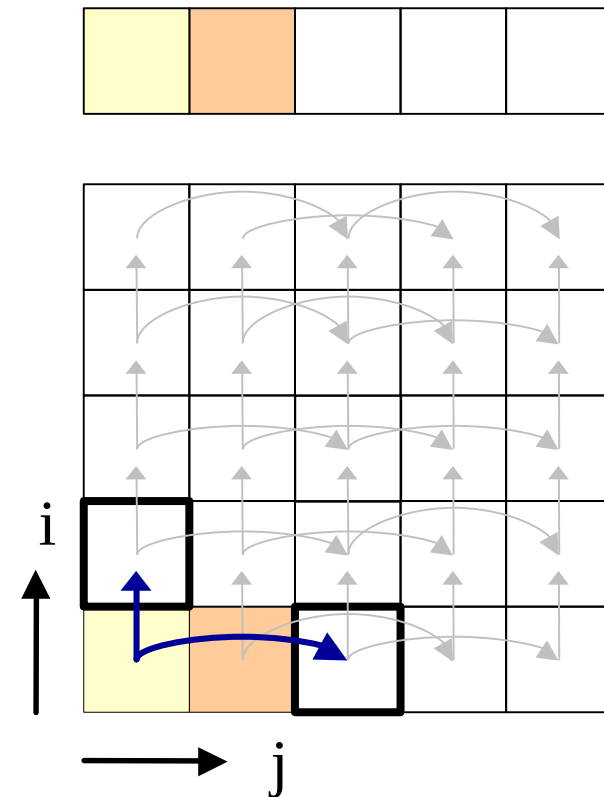
Answering Question #2

- Given $\vec{v} = (0, 1)$, what is the range of valid schedules θ ?
 - ➔ Lets try $\theta(i, j) = 2 * i + j$



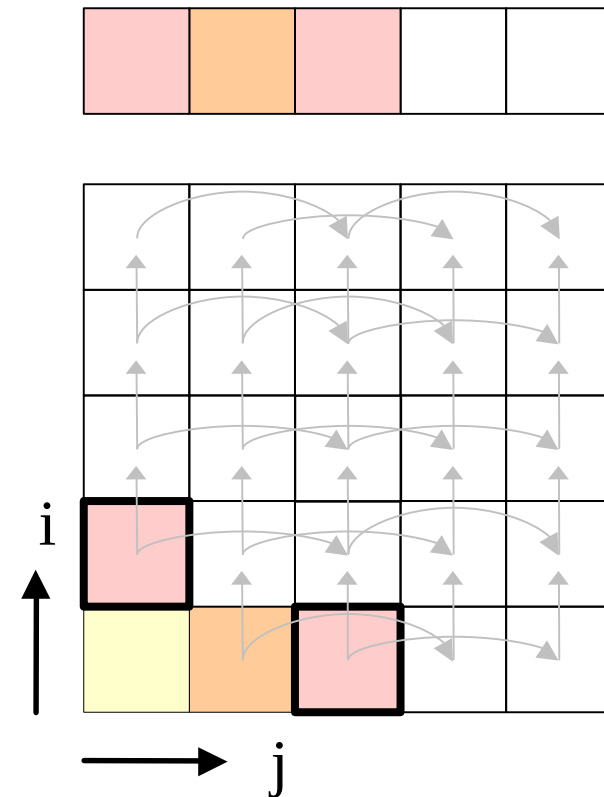
Answering Question #2

- Given $\vec{v} = (0, 1)$, what is the range of valid schedules θ ?
 - ➔ Lets try $\theta(i, j) = 2 * i + j$



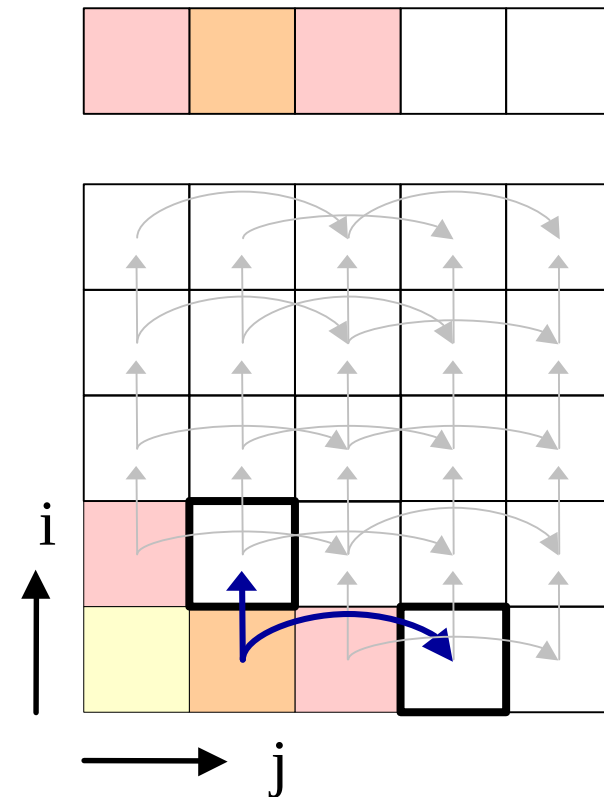
Answering Question #2

- Given $\vec{v} = (0, 1)$, what is the range of valid schedules θ ?
 - ➔ Lets try $\theta(i, j) = 2 * i + j$



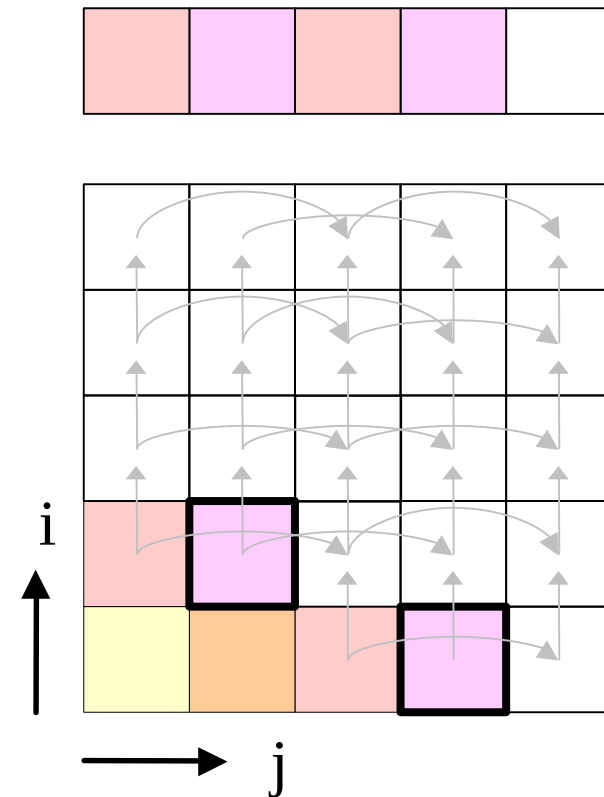
Answering Question #2

- Given $\vec{v} = (0, 1)$, what is the range of valid schedules θ ?
 - ➔ Lets try $\theta(i, j) = 2 * i + j$



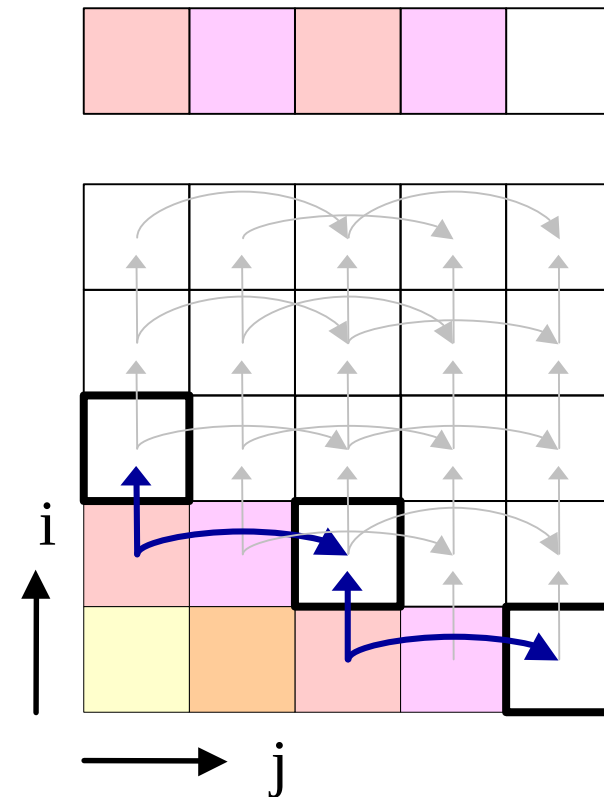
Answering Question #2

- Given $\vec{v} = (0, 1)$, what is the range of valid schedules θ ?
 - Lets try $\theta(i, j) = 2 * i + j$



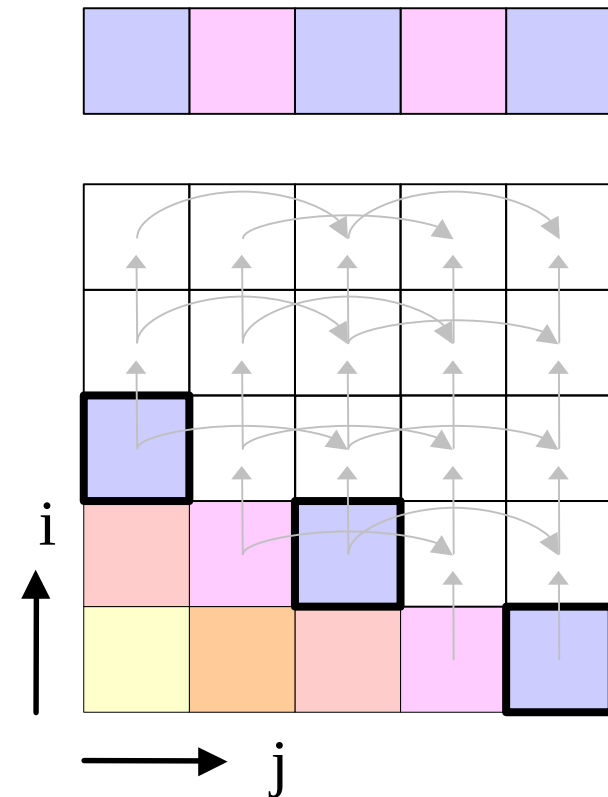
Answering Question #2

- Given $\vec{v} = (0, 1)$, what is the range of valid schedules θ ?
 - ➔ Lets try $\theta(i, j) = 2 * i + j$



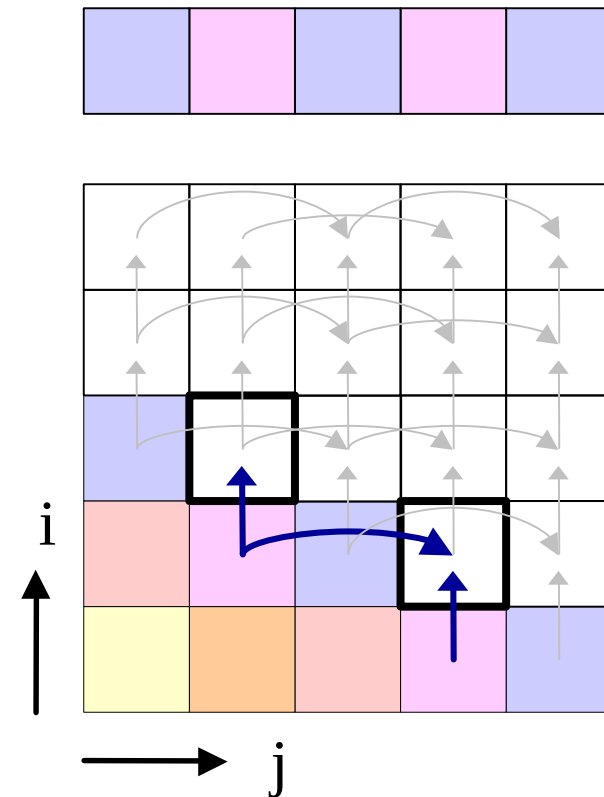
Answering Question #2

- Given $\vec{v} = (0, 1)$, what is the range of valid schedules θ ?
 - ➔ Lets try $\theta(i, j) = 2 * i + j$



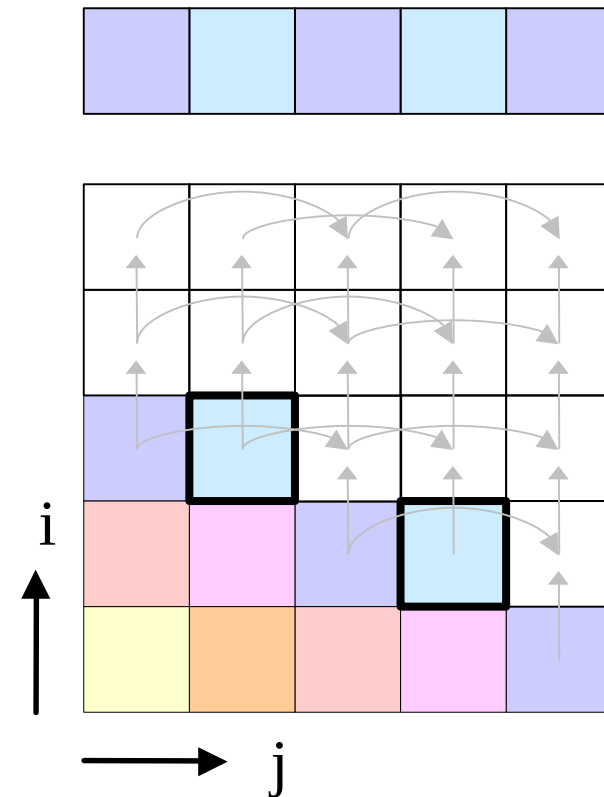
Answering Question #2

- Given $\vec{v} = (0, 1)$, what is the range of valid schedules θ ?
 - ➔ Lets try $\theta(i, j) = 2 * i + j$



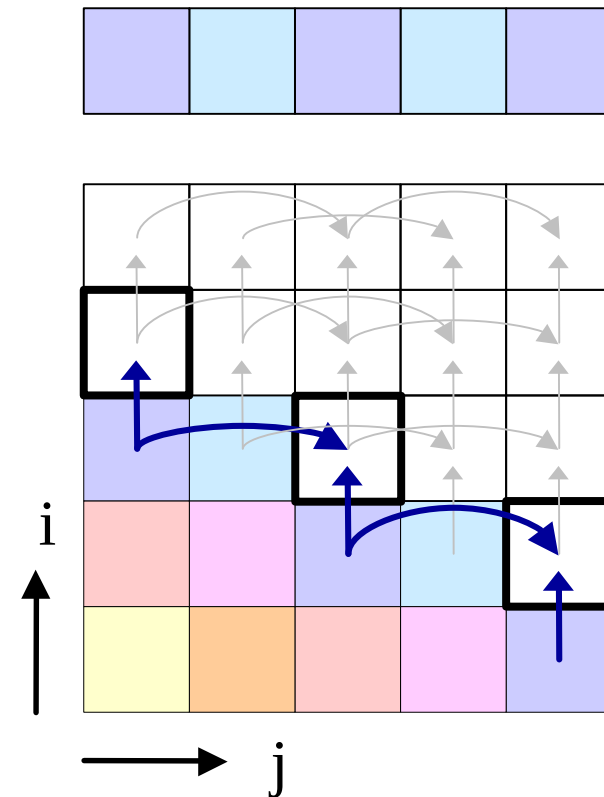
Answering Question #2

- Given $\vec{v} = (0, 1)$, what is the range of valid schedules θ ?
 - ➔ Lets try $\theta(i, j) = 2 * i + j$



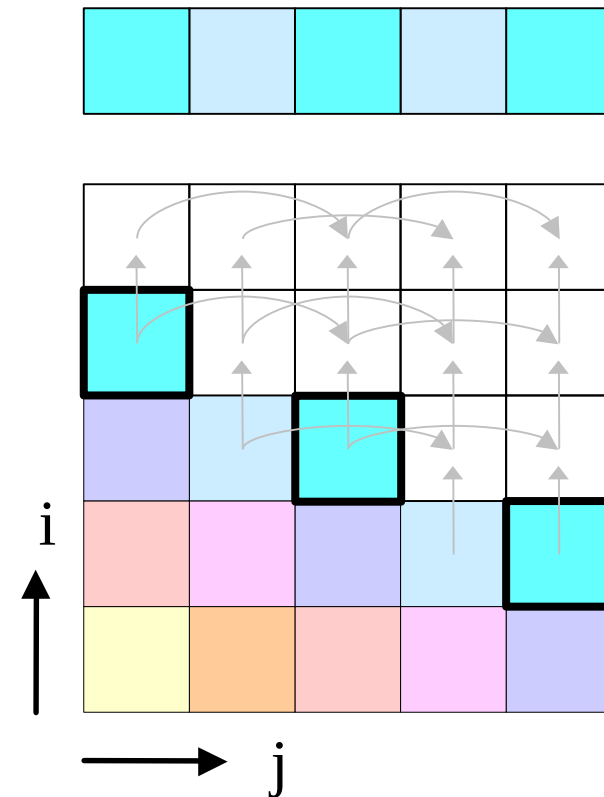
Answering Question #2

- Given $\vec{v} = (0, 1)$, what is the range of valid schedules θ ?
 - ➔ Lets try $\theta(i, j) = 2 * i + j$



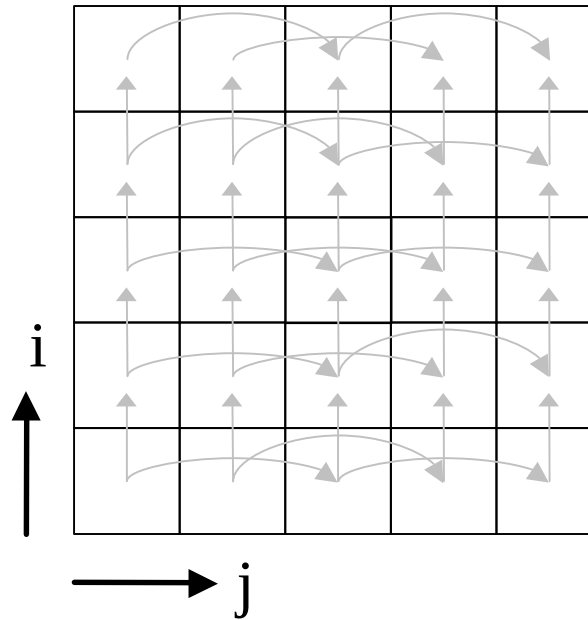
Answering Question #2

- Given $\vec{v} = (0, 1)$, what is the range of valid schedules θ ?
 - ➔ Lets try $\theta(i, j) = 2 * i + j$



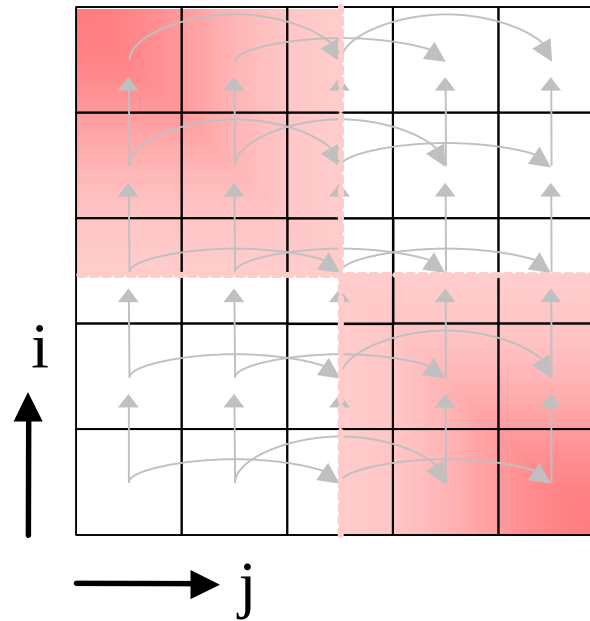
Answering Question #3

- What is the shortest $\vec{v}$ that is valid for all legal affine schedules?



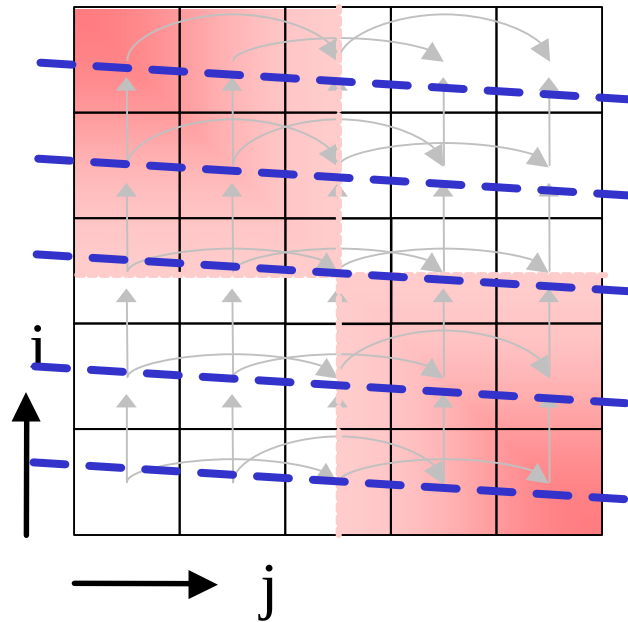
Answering Question #3

- What is the shortest $\vec{v}$ that is valid for all legal affine schedules?
 - ➔ Range of legal θ



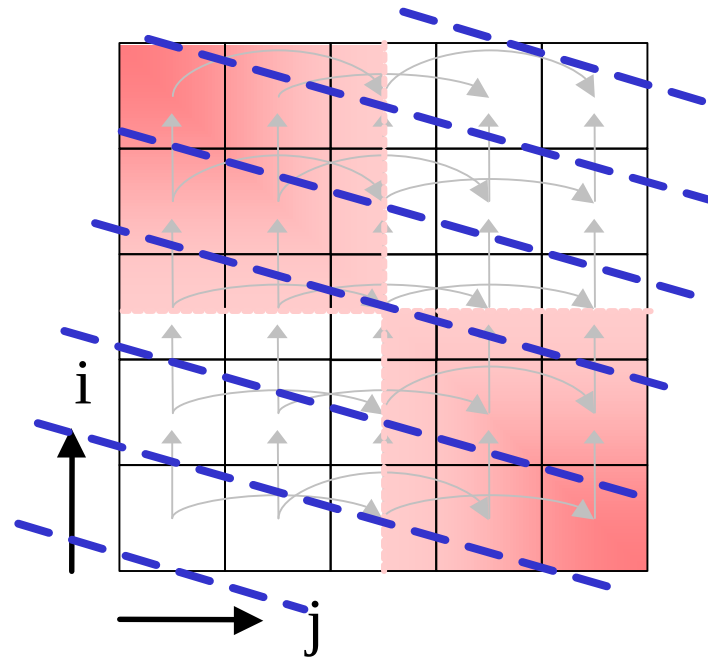
Answering Question #3

- What is the shortest $\vec{v}$ that is valid for all legal affine schedules?
 - ➔ Range of legal θ



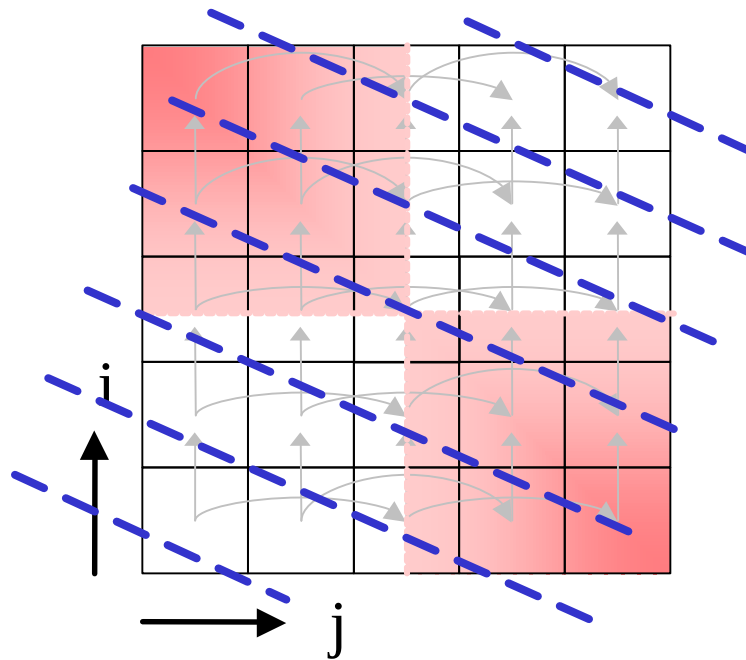
Answering Question #3

- What is the shortest $\vec{v}$ that is valid for all legal affine schedules?
 - ➔ Range of legal θ



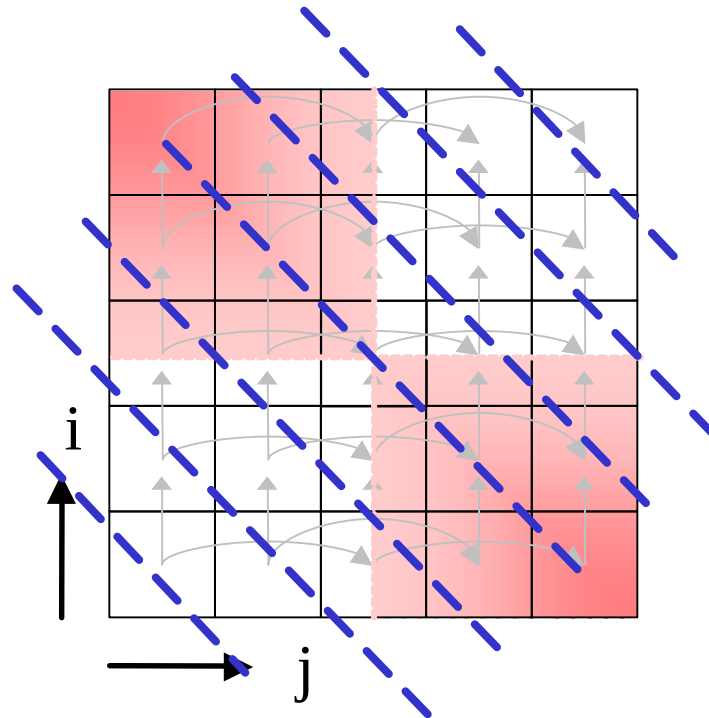
Answering Question #3

- What is the shortest $\vec{v}$ that is valid for all legal affine schedules?
➔ Range of legal θ



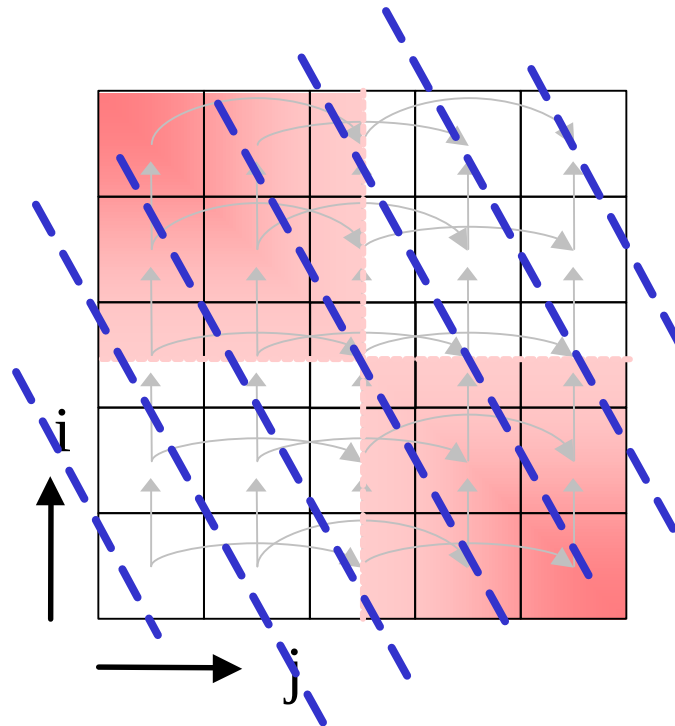
Answering Question #3

- What is the shortest $\vec{v}$ that is valid for all legal affine schedules?
 - ➔ Range of legal θ



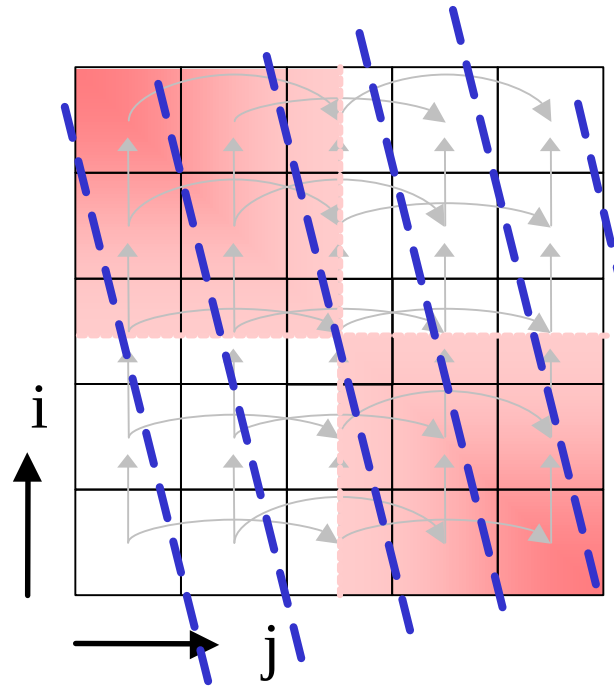
Answering Question #3

- What is the shortest $\vec{v}$ that is valid for all legal affine schedules?
 - ➔ Range of legal θ



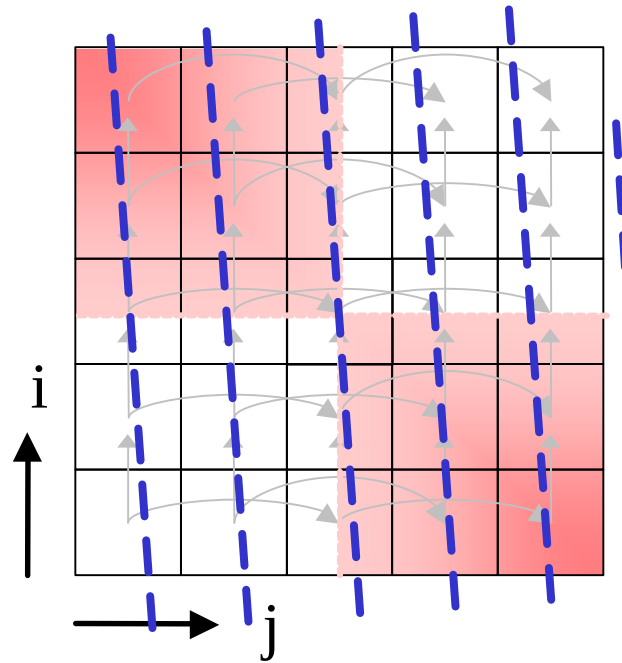
Answering Question #3

- What is the shortest $\vec{v}$ that is valid for all legal affine schedules?
 - ➔ Range of legal θ



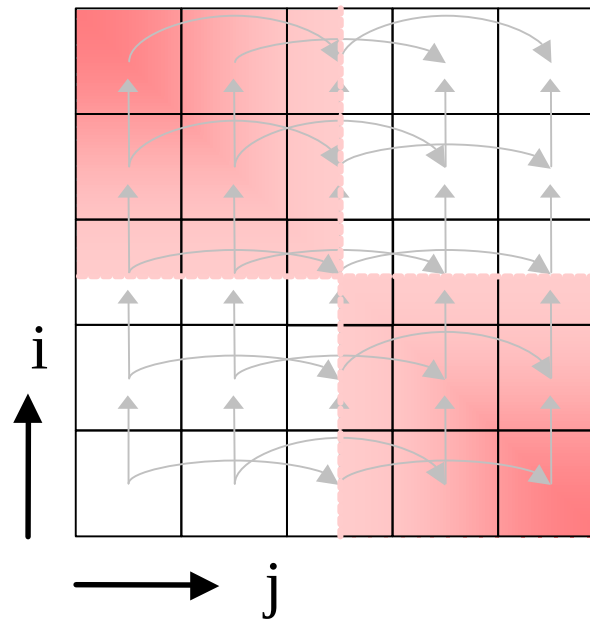
Answering Question #3

- What is the shortest $\vec{v}$ that is valid for all legal affine schedules?
 - ➔ Range of legal θ



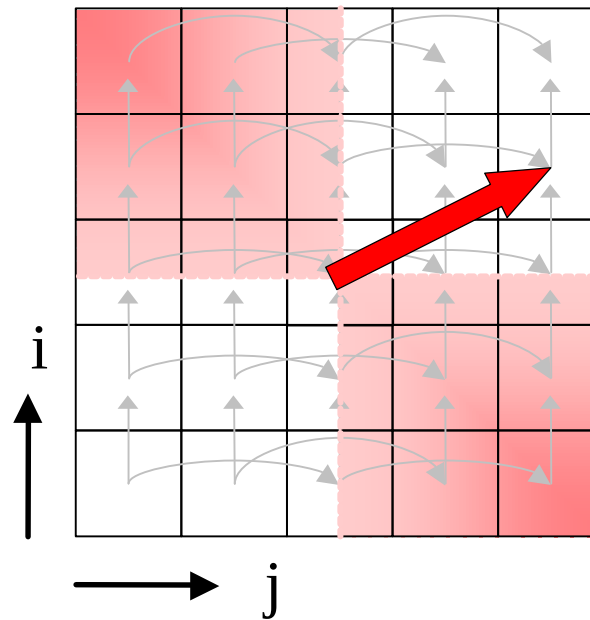
Answering Question #3

- What is the shortest $\vec{v}$ that is valid for all legal affine schedules?
 - ➔ Range of legal θ



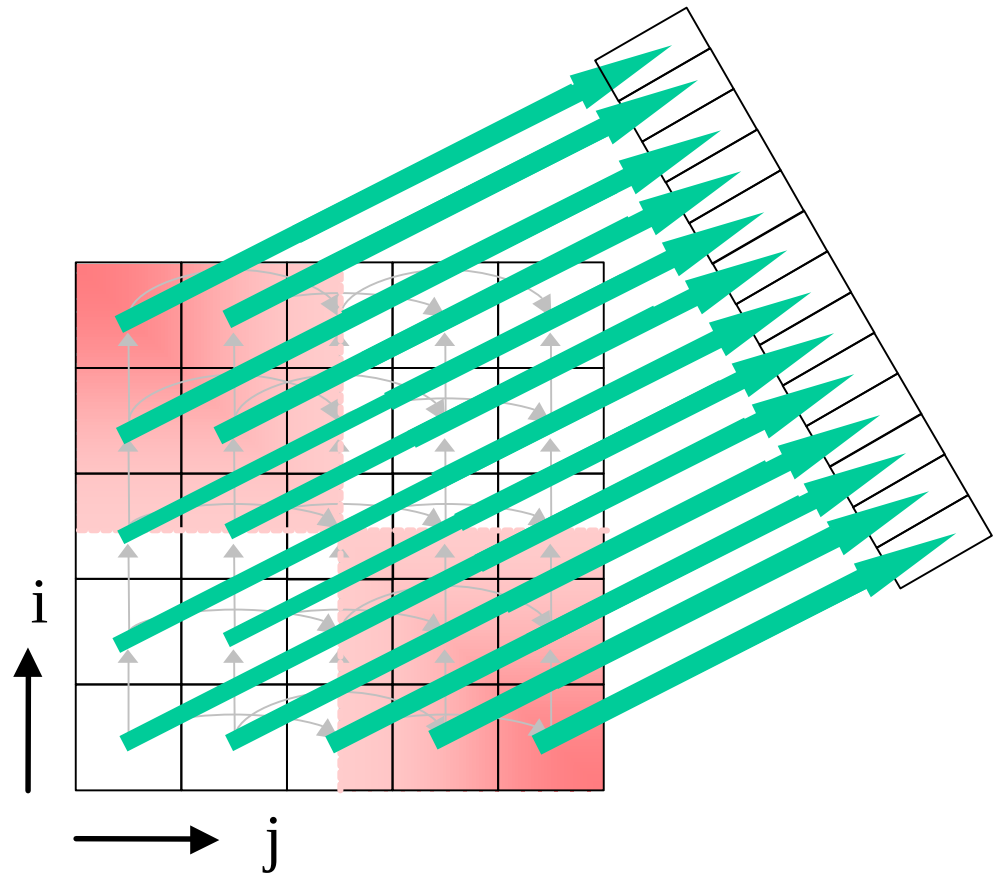
Answering Question #3

- What is the shortest $\vec{v}$ that is valid for all legal affine schedules?
 - ➔ Range of legal θ
 - ➔ $\vec{v} = (2, 1)$



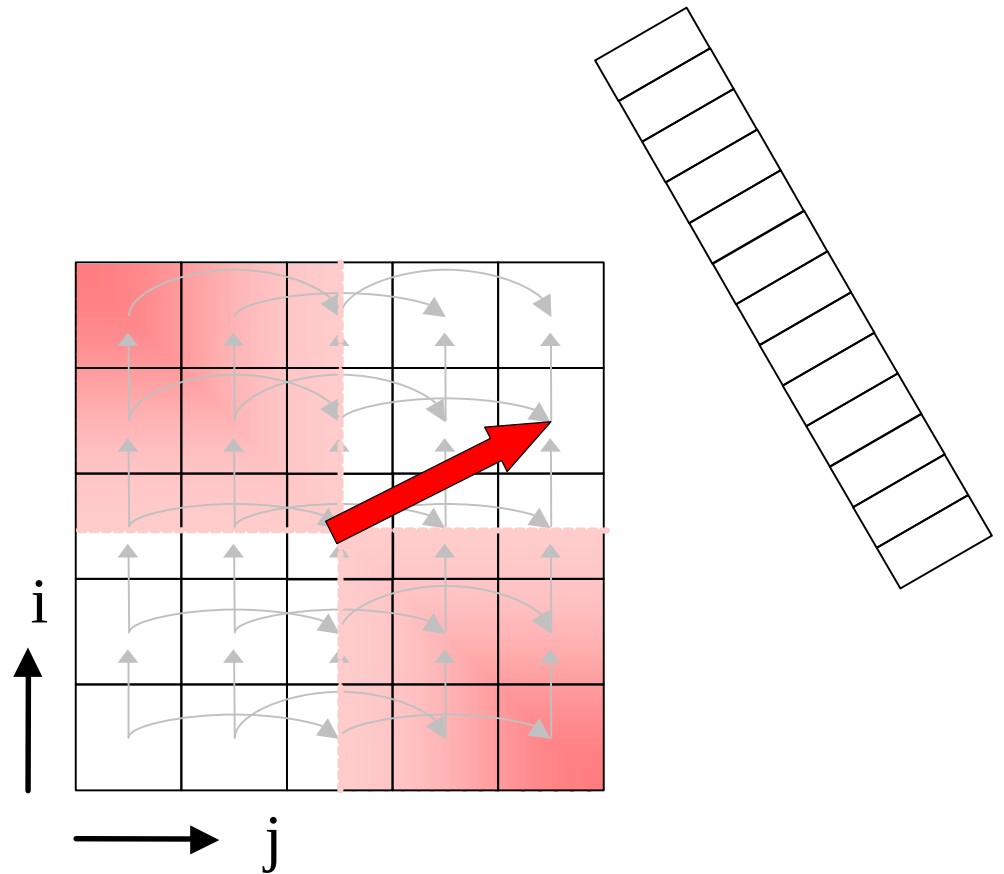
Answering Question #3

- What is the shortest $\vec{v}$ that is valid for all legal affine schedules?
 - ➔ Range of legal θ
 - ➔ $\vec{v} = (2, 1)$



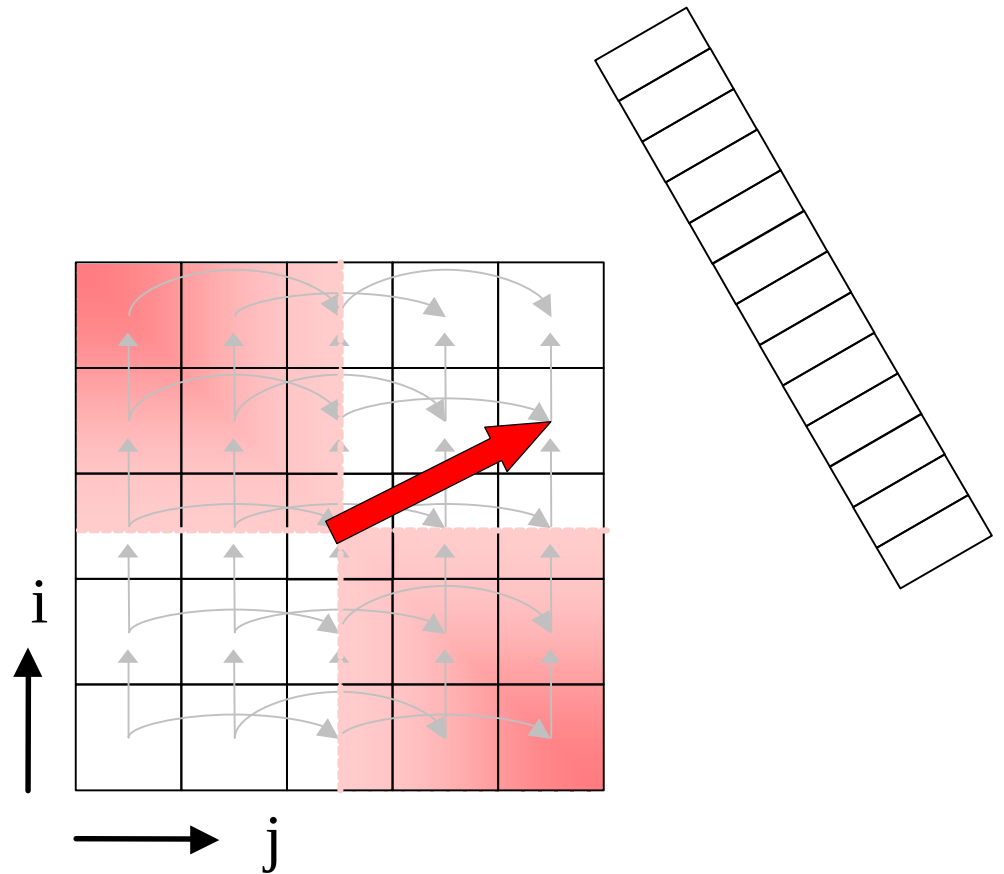
Answering Question #3

- What is the shortest $\vec{v}$ that is valid for all legal affine schedules?
 - ➔ Range of legal θ
 - ➔ $\vec{v} = (2, 1)$



Answering Question #3

- What is the shortest $\vec{v}$ that is valid for all legal affine schedules?
 - ➔ Range of legal θ
 - ➔ $\vec{v} = (2, 1)$



- Def: $\vec{v}$ is an **affine occupancy vector** (AOV)

Outline

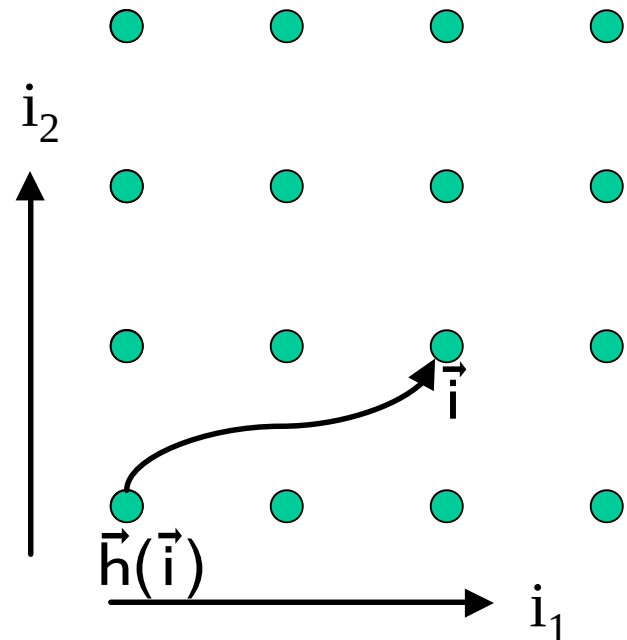
- Abstract problem
- Simplifications
- Concrete problem
- **Solution Method**
- Conclusions

Schedule Constraints

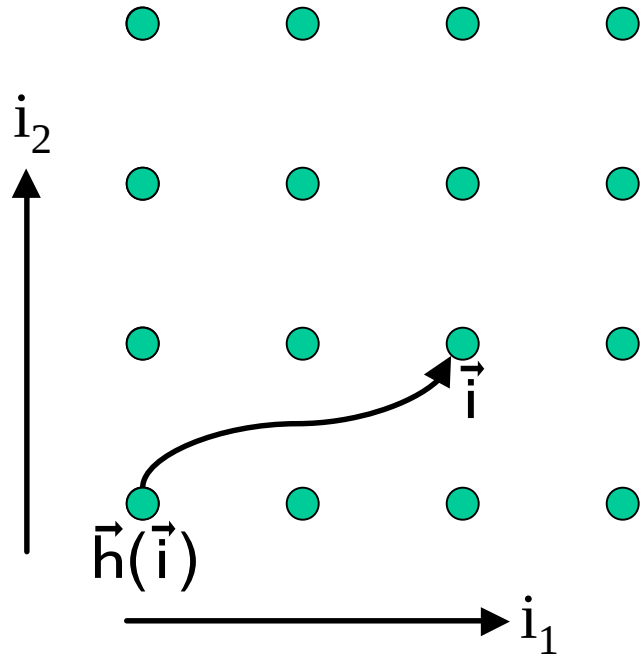
- Dependence analysis yields:
 - iteration $\vec{i}$ depends on iteration $\vec{h}(\vec{i})$
 - $\vec{h}$ is an affine function
- Consumer must execute after producer

Schedule Constraint

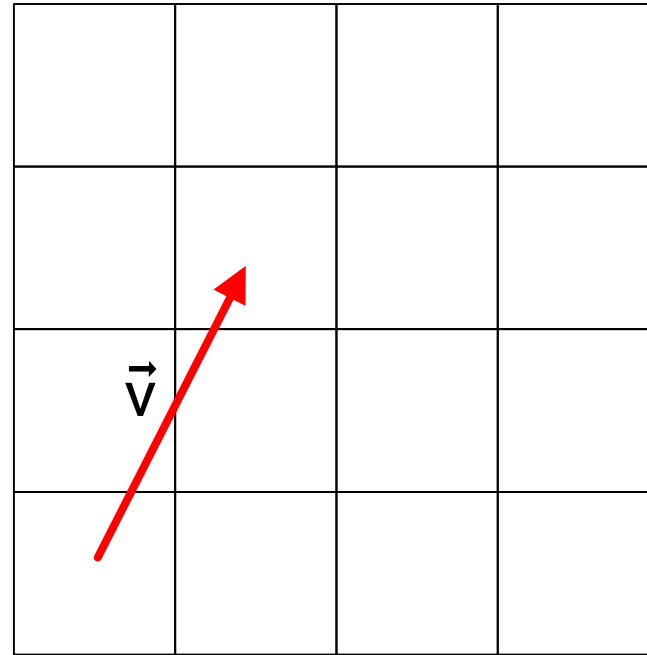
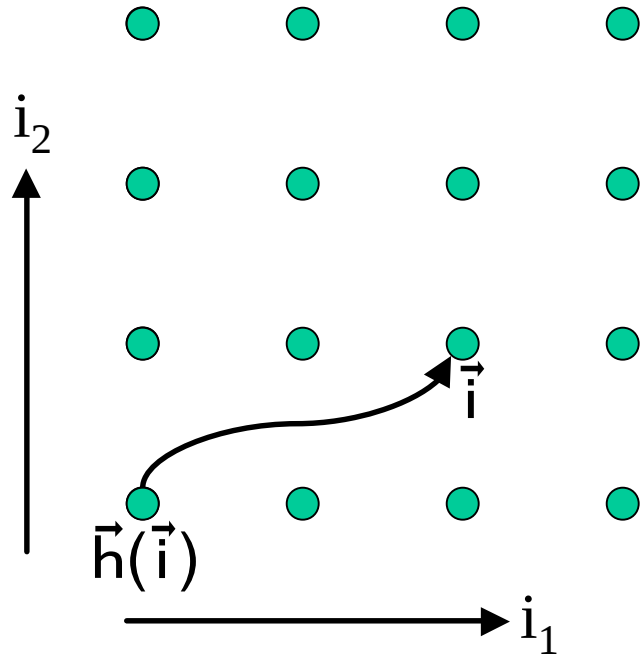
$$\theta(\vec{i}) \geq \theta(\vec{h}(\vec{i})) + 1$$



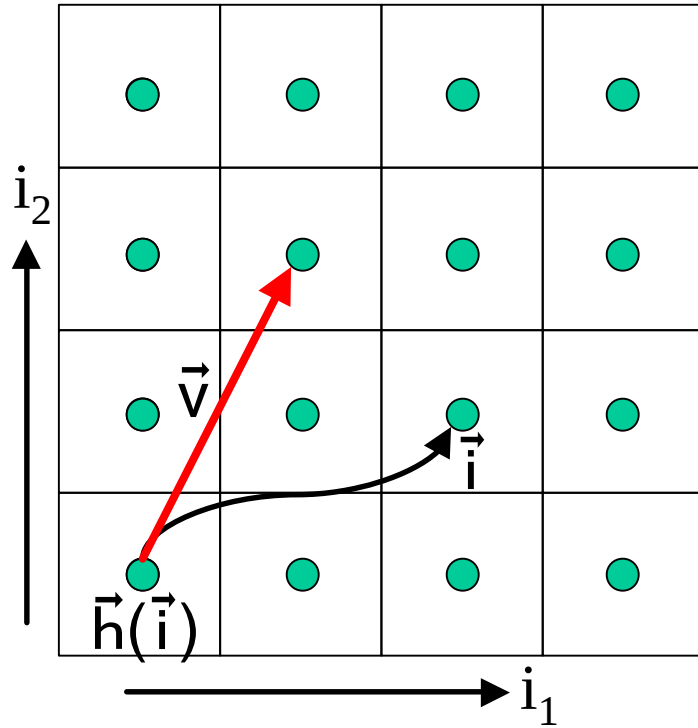
Storage Constraints



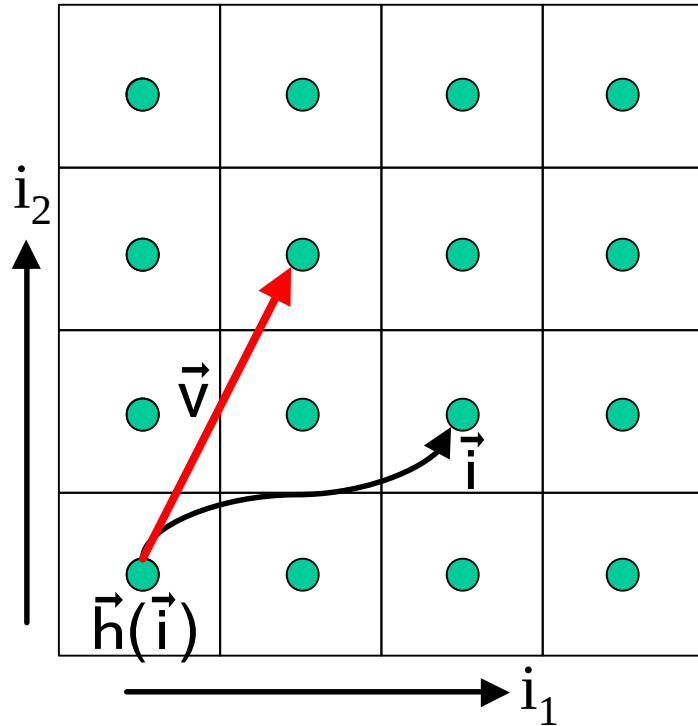
Storage Constraints



Storage Constraints



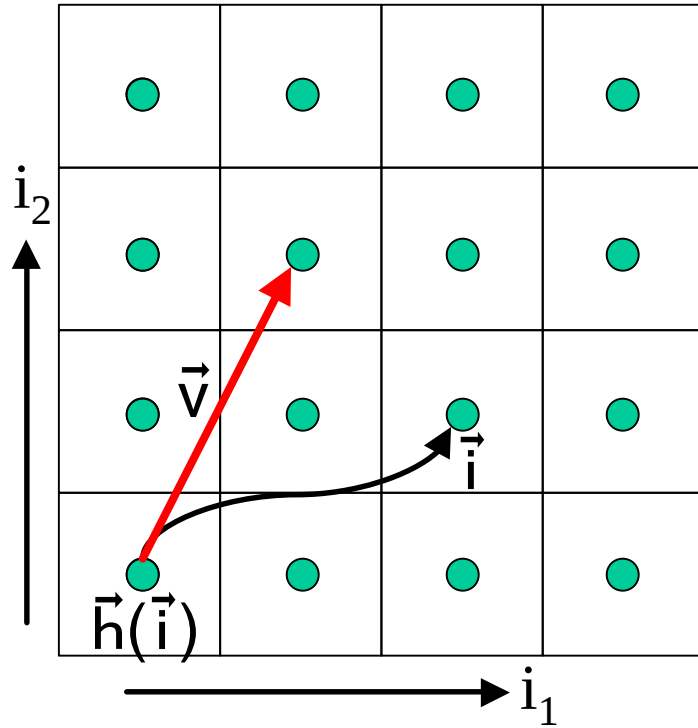
Storage Constraints



dynamic single assignment

```
for i = 1 to n  
  for j = 1 to n  
    A[i][j] = ...  
    B[i][j] = ...
```

Storage Constraints

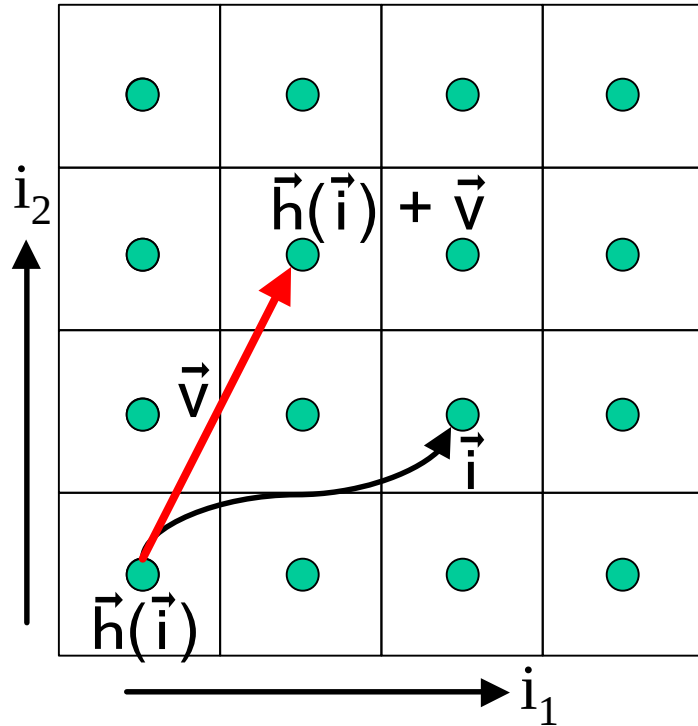


Consumer:

Producer:

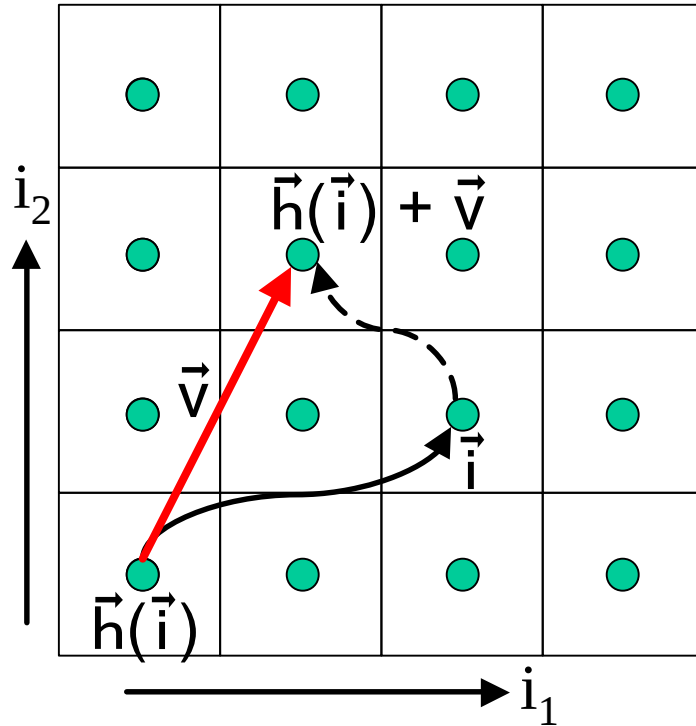
$\vec{i}$
 $\vec{h}(\vec{i})$

Storage Constraints



Consumer: $\vec{i}$
Producer: $\vec{h}(\vec{i})$
Overwriting producer: $\vec{h}(\vec{i}) + \vec{v}$

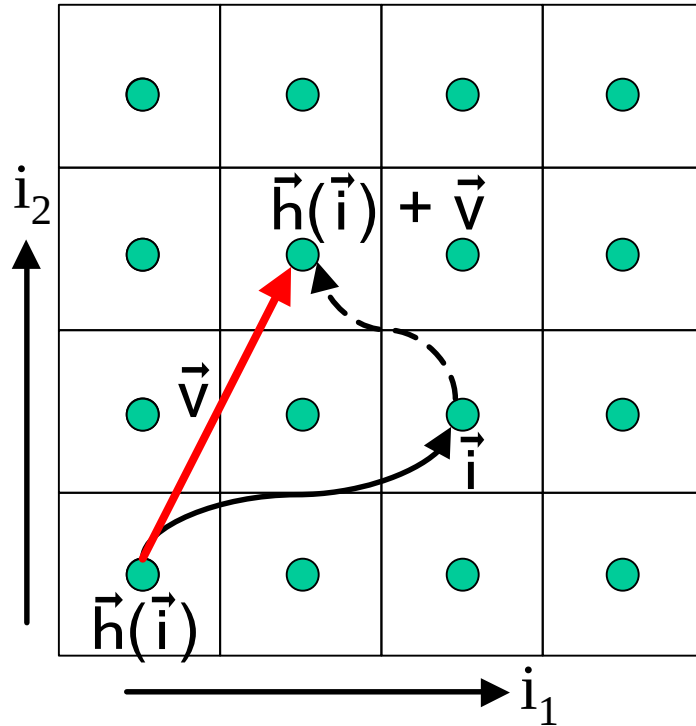
Storage Constraints



Consumer: $\vec{i}$
Producer: $\vec{h}(\vec{i})$
Overwriting producer: $\vec{h}(\vec{i}) + \vec{v}$

➔ Consumer must execute before producer is overwritten

Storage Constraints



Consumer: $\vec{i}$

Producer: $\vec{h}(\vec{i})$

Overwriting producer: $\vec{h}(\vec{i}) + \vec{v}$

→ Consumer must execute before producer is overwritten

Storage Constraint

$$\theta(\vec{i}) \leq \theta(\vec{h}(\vec{i}) + \vec{v})$$

The Constraints

- A given $(\theta, \vec{v})$ combination is valid if
 - For all dependences $\vec{h}$,
 - For all iterations $\vec{i}$ in the program:

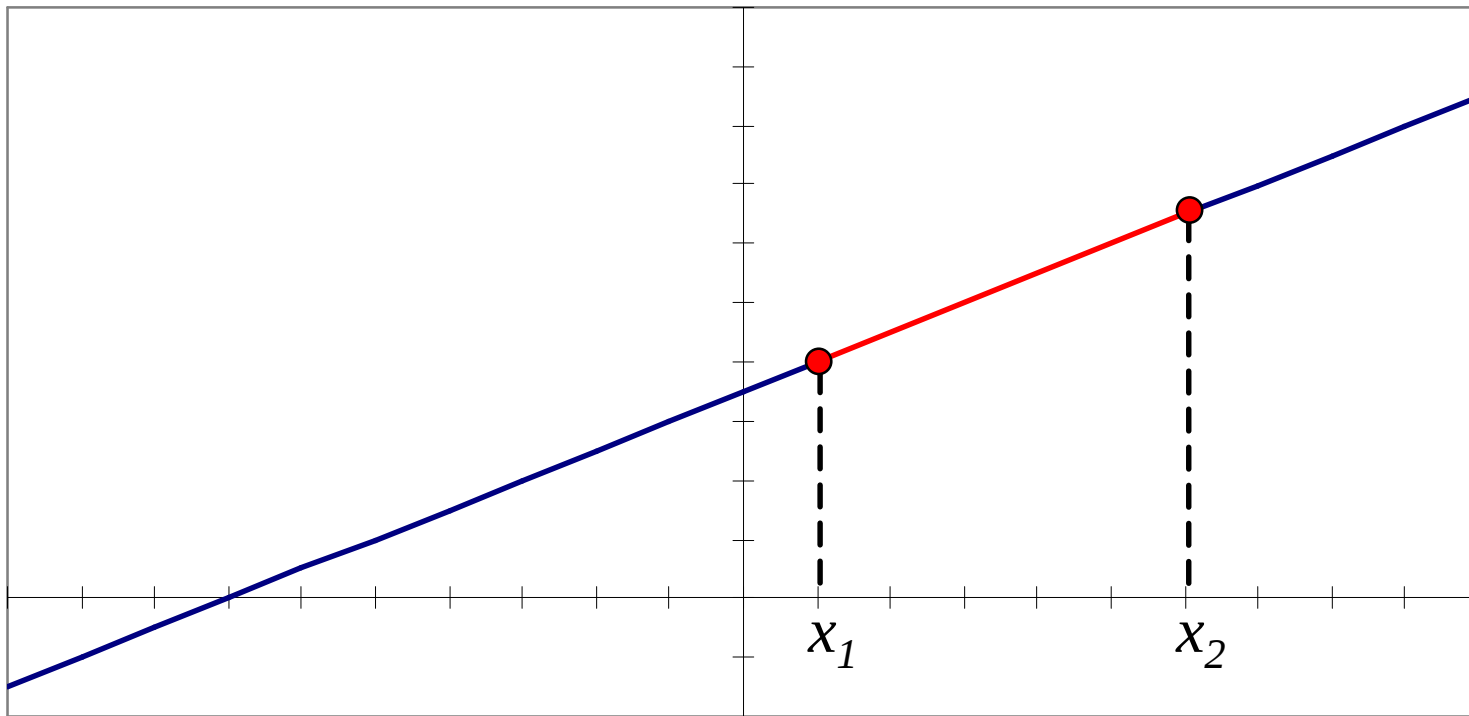
$$\begin{cases} \theta(\vec{i}) \geq \theta(\vec{h}(\vec{i})) + 1 & \text{schedule constraint} \\ \theta(\vec{i}) \leq \theta(\vec{h}(\vec{i})) + \vec{v} & \text{storage constraint} \end{cases}$$

The Constraints

- A given $(\theta, \vec{v})$ combination is valid if
 - For all dependences $\vec{h}$,
 - For all iterations $\vec{i}$ in the program:
$$\begin{cases} \theta(\vec{i}) \geq \theta(\vec{h}(\vec{i})) + 1 & \text{schedule constraint} \\ \theta(\vec{i}) \leq \theta(\vec{h}(\vec{i})) + \vec{v} & \text{storage constraint} \end{cases}$$
- Given θ , want to find $\vec{v}$ satisfying constraints
 - Might look simple, but it is not
 - Too many $\vec{i}$'s to enumerate!
 - Need to reduce the number of constraints

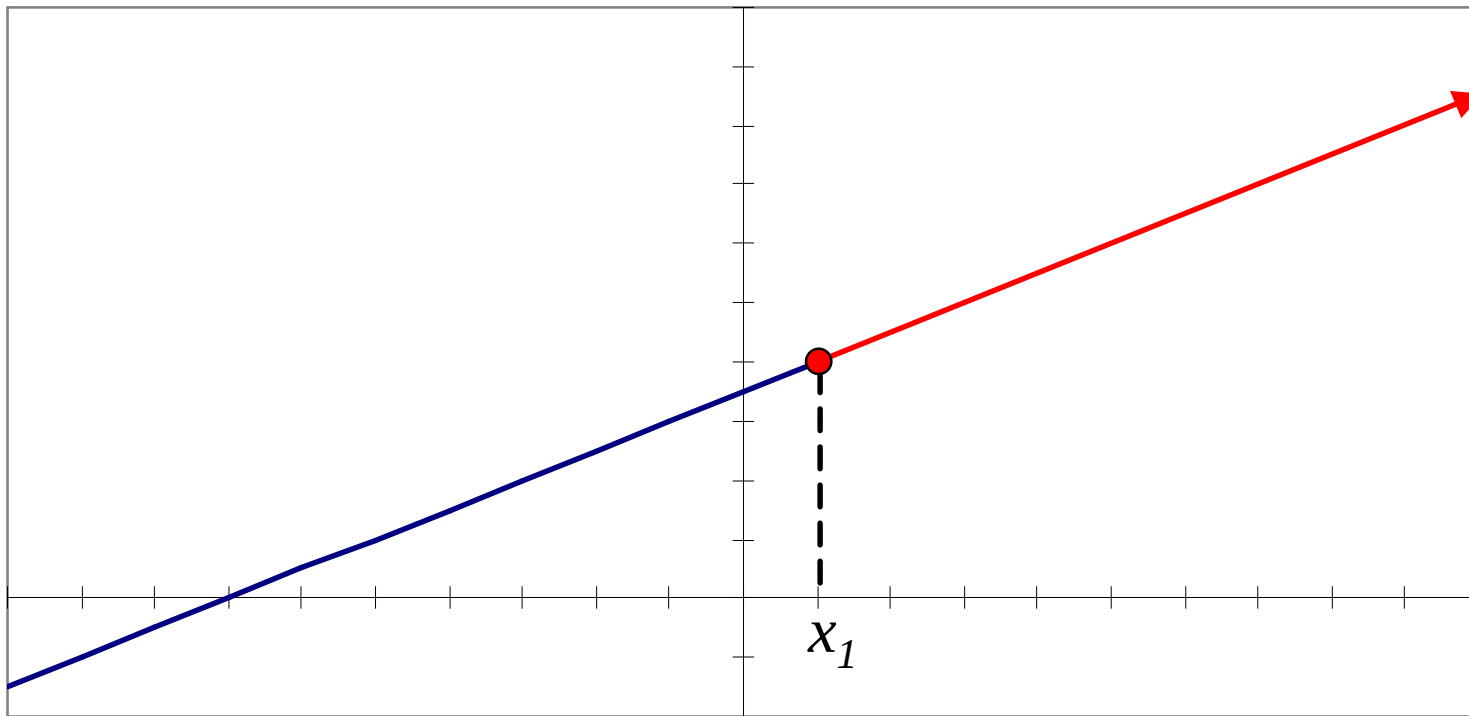
The Vertex Method (1-D)

- An affine function is non-negative within an interval $[x_1, x_2]$ iff it is non-negative at x_1 and x_2



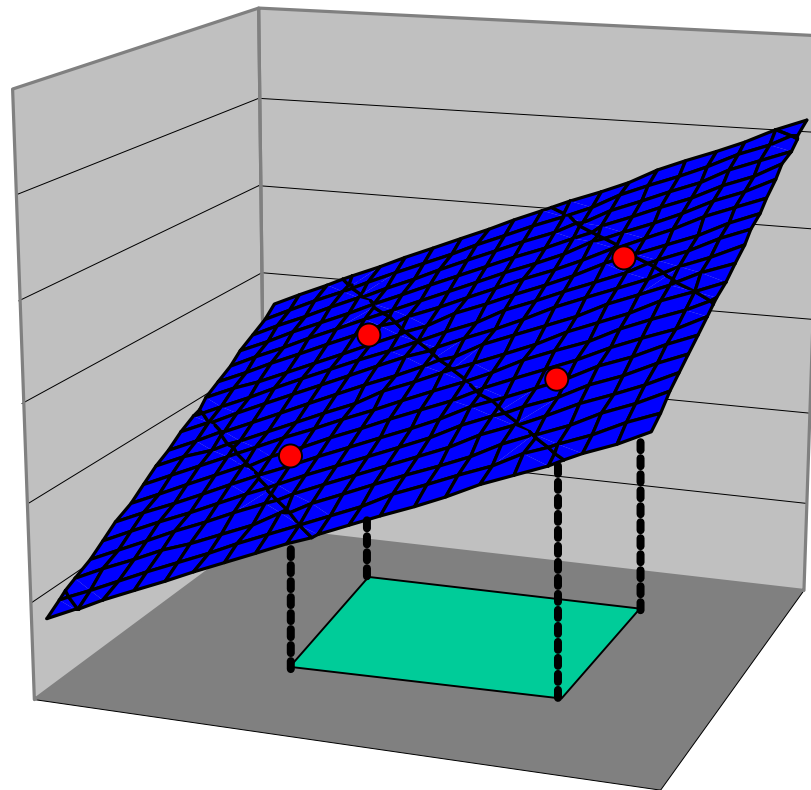
The Vertex Method (1-D)

- An affine function is non-negative over an unbounded interval $[x_1, \infty)$ iff it is non-negative at x_1 and is non-decreasing along the interval



The Vertex Method

- The same result holds in higher dimensions
 - An affine function is nonnegative over a bounded polyhedron D iff it is nonnegative at vertices of D



Applying the Method (Quinton87)

- Recall the storage constraints
 - For all iterations $\vec{i}$ in the program:

$$\theta(\vec{i}) \leq \theta(\vec{h}(\vec{i}) + \vec{v})$$

- Re-arrange:

$$0 \leq \theta(\vec{h}(\vec{i}) + \vec{v}) - \theta(\vec{i})$$

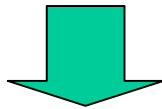
- The right hand side is:
 1. An affine function of $\vec{i}$
 2. Nonnegative over the domain D of iterations

➔ We can apply the vertex method

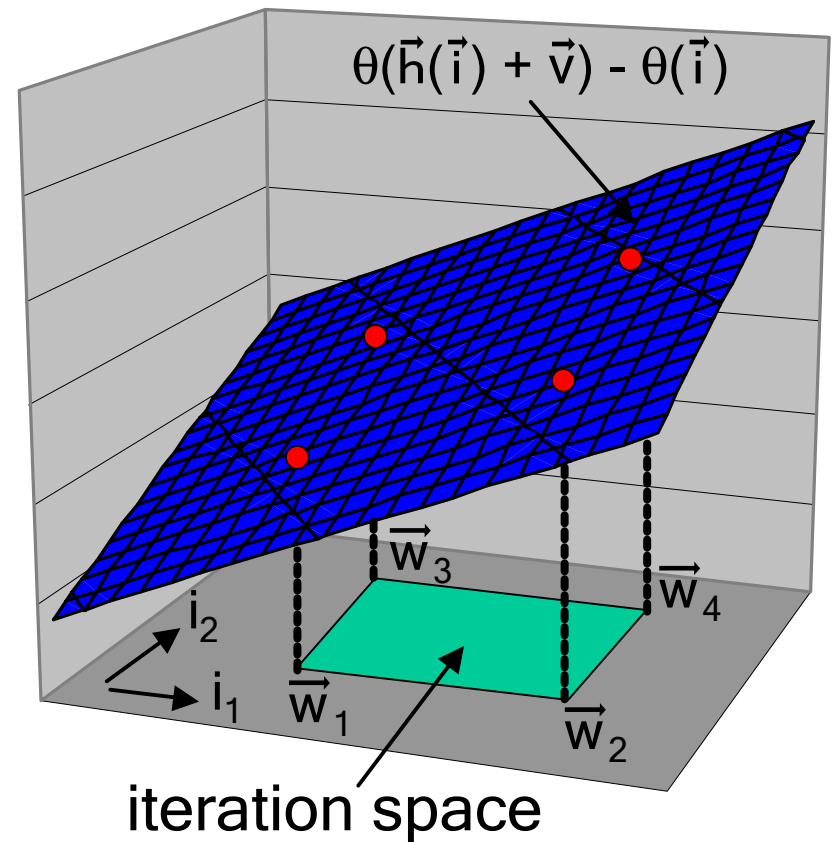
Applying the Method

- Replace $\vec{i}$ with the vertices $\vec{w}$ of its domain:

$$\forall \vec{i} \in D, \theta(\vec{h}(\vec{i}) + \vec{v}) - \theta(\vec{i}) \geq 0$$



$$\begin{cases} \theta(\vec{h}(\vec{w}_1) + \vec{v}) - \theta(\vec{w}_1) \geq 0 \\ \theta(\vec{h}(\vec{w}_2) + \vec{v}) - \theta(\vec{w}_2) \geq 0 \\ \theta(\vec{h}(\vec{w}_3) + \vec{v}) - \theta(\vec{w}_3) \geq 0 \\ \theta(\vec{h}(\vec{w}_4) + \vec{v}) - \theta(\vec{w}_4) \geq 0 \end{cases}$$



The Reduced Constraints

- Apply same method to schedule constraints
- Now a given $(\theta, \vec{v})$ combination is valid if
 - For all dependences $\vec{h}$,
 - For all vertices $\vec{w}$ of the iteration domain:

$\theta(\vec{w}) \geq \theta(\vec{h}(\vec{w})) + 1$	schedule constraint
$\theta(\vec{w}) \leq \theta(\vec{h}(\vec{w})) + \vec{v}$	storage constraint

- These are linear constraints
 - θ and $\vec{v}$ are variables; $\vec{h}$ and $\vec{w}$ are constants
 - Given θ , constraints are linear in $\vec{v}$ (& vice-versa)

Answering the Questions

$\theta(\vec{w}) \geq \theta(\vec{h}(\vec{w})) + 1$	schedule constraint
$\theta(\vec{w}) \leq \theta(\vec{h}(\vec{w}) + \vec{v})$	storage constraint

1. Given θ , we can “minimize” $|\vec{v}|$
 - Linear programming problem
2. Given $\vec{v}$, we can find a “good” θ
 - Feautrier, 1992
3. To find an AOV... still too many constraints!
 - For all θ satisfying the schedule constraints:
 $\vec{v}$ must satisfy the storage constraints

Finding an AOV

$\begin{cases} \theta(\vec{w}) \geq \theta(\vec{h}(\vec{w})) + 1 \\ \theta(\vec{w}) \leq \theta(\vec{h}(\vec{w}) + \vec{v}) \end{cases}$	schedule constraint storage constraint
--	--

- Apply the vertex method again!
 - Schedule constraints define domain of valid θ
 - Storage constraints can be written as a non-negative affine function of components of θ :
 - Expand $\theta(\vec{i}) = \vec{\beta} \cdot \vec{i}$
$$\vec{\beta} \cdot \vec{w} \leq \vec{\beta} \cdot (\vec{h}(\vec{w}) + \vec{v})$$
 - Simplify
$$(\vec{h}(\vec{w}) + \vec{v} - \vec{w}) \cdot \vec{\beta} \geq 0$$

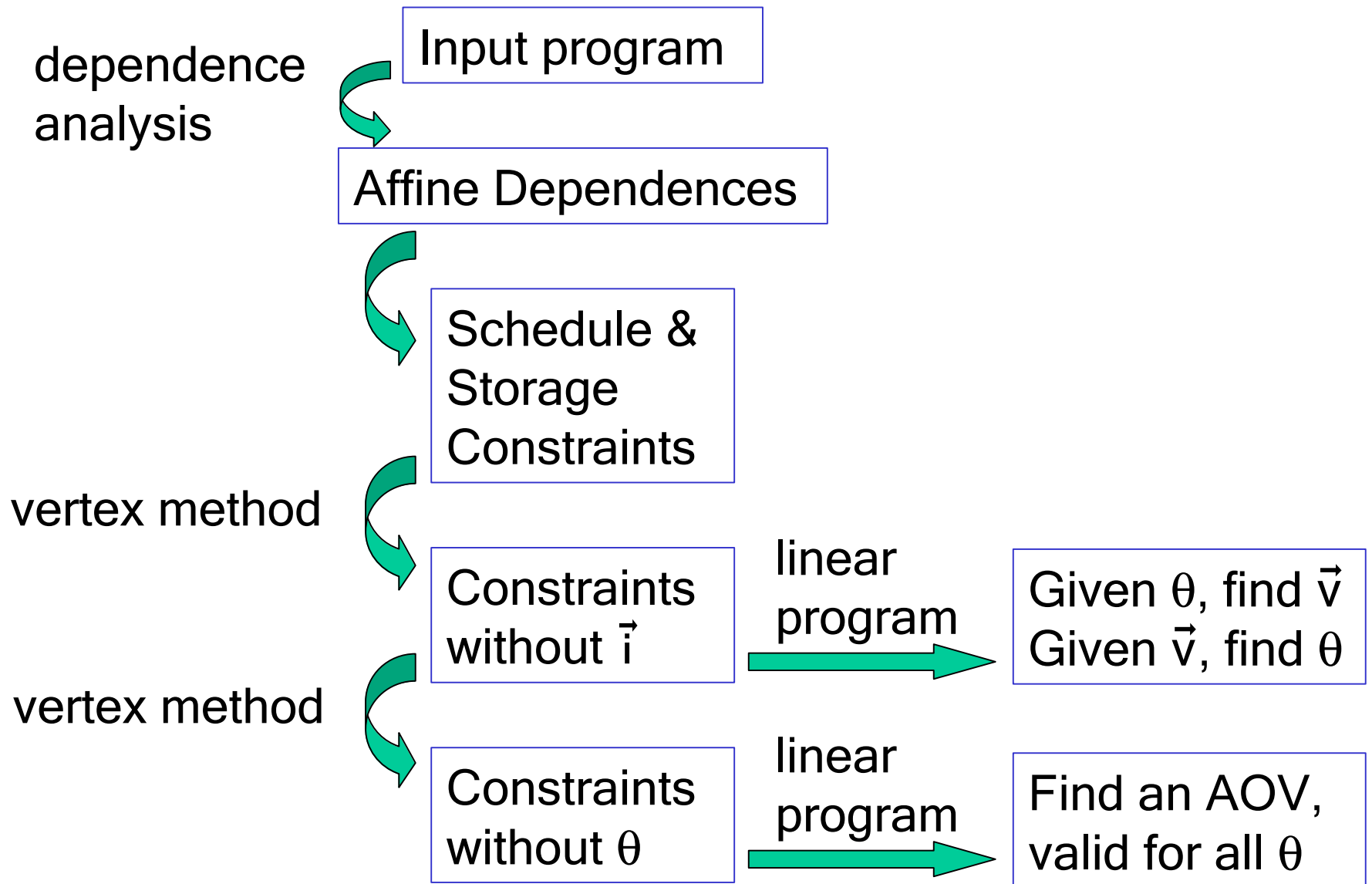
Finding an AOV

- Our constraints are now as follows:
 - For all dependences $\vec{h}$,
 - For all vertices $\vec{w}$ of the iteration domain,
 - For all vertices $\vec{\tau}$ of the space of valid schedules:

$$\vec{\tau} \cdot \vec{w} \leq \vec{\tau} \cdot (\vec{h}(\vec{w}) + \vec{v}) \quad \text{AOV constraint}$$

- Can find “shortest” AOV with linear program
 - Finite number of constraints
 - $\vec{h}$, $\vec{w}$, and $\vec{\tau}$ are known constants

The Big Picture



Details in Paper

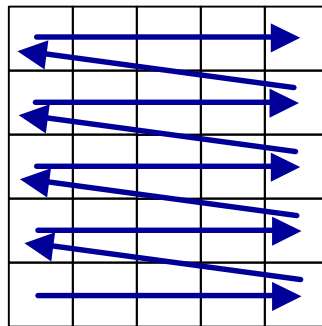
- Symbolic constants
- Inter-statement dependences across loops
- Farkas' Lemma for improved efficiency

Related Work

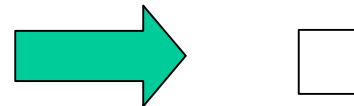
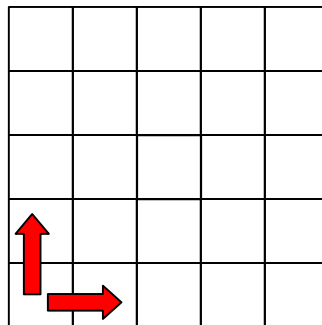
- Universal Occupancy Vector (Strout et al.)
 - Valid for all schedules, not just affine ones
 - Stencil of dependences in single loop nest
- Storage for ALPHA programs (Quilleré, Rajopadhye, Wilde)
 - Polyhedral model, with occupancy vector analog
 - Assume schedule is given
- PAF compiler (Cohen, Lefebvre, Feautrier)
 - Minimal expansion → scheduling → contraction
 - Storage mapping $A[i \bmod x][j \bmod y]$

Future Work

- Allow affine left hand side references
 - $A[2*j][n-i] = \dots$
- Consider multi-dimensional time schedules



- Collapse multiple dimensions of storage



Conclusions

- Unified framework for determining:
 1. A good storage mapping for a given schedule
 2. A good schedule for a given storage mapping
 3. A good storage mapping for all valid schedules
- Take away: representations and techniques
 - Occupancy vectors
 - Affine schedules
 - Vertex method