Representing Dependence
Iteration space graphs

• Represent each *dynamic* instance of a loop as a point in a graph

• Draw arrows from one point to another to represent dependences

    for (i = 0; i < N; i++) {
        a[i + 2] = a[i]
    }
Iteration space graphs

• Represent each \textit{dynamic} instance of a loop as a point in a graph

• Draw arrows from one point to another to represent dependences

\begin{verbatim}
for (i = 0; i < N; i++) {
    a[i + 2] = a[i]
}
\end{verbatim}

• Step 1: Create nodes, 1 for each iteration

• Note: not 1 for each array location!
Iteration space graphs

• Represent each *dynamic* instance of a loop as a point in a graph

• Draw arrows from one point to another to represent dependences

```plaintext
for (i = 0; i < N; i++) {
    a[i + 2] = a[i]
}
```

• Step 2: Determine which array elements are read and written in each iteration

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>W: a[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Iteration space graphs

• Represent each *dynamic* instance of a loop as a point in a graph

• Draw arrows from one point to another to represent dependences

```c
for (i = 0; i < N; i++) {
    a[i + 2] = a[i]
}
```

• Step 3: Draw arrows to represent dependences

```
```
2-D iteration space graphs

- Can do the same thing for doubly-nested loops

- 2 loop counters

```c
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    ...
    a[i+1][j] = a[i][j+2] + 1
    ...
```
Iteration space graphs

• Can also represent output and anti dependences

• Use different kinds of arrows for clarity. E.g.

• for output

• for anti

• Crucial problem: Iteration space graphs are potentially infinite representations!

• Can we represent dependences in a more compact way?
Distance and direction vectors

• Compiler researchers have devised compressed representations of dependences
  • Capture the same dependences as an iteration space graph
  • May lose precision (show more dependences than the loop actually has)
• Two types
  • **Distance vectors:** captures the “shape” of dependences, but not the particular source and sink
  • **Direction vectors:** captures the “direction” of dependences, but not the particular shape
Distance vector

- Represent each dependence arrow in an iteration space graph as a vector
- Captures the “shape” of the dependence, but loses where the dependence originates

Distance vector for this iteration space: (2)

- Each dependence is 2 iterations forward
2-D distance vectors

• Distance vector for this graph:
  • (1, -2)
  • +1 in the i direction, -2 in the j direction

• Crucial point about distance vectors: they are always “positive”
  • First non-zero entry has to be positive
  • Dependences can’t go backwards in time
More complex example

- Can have multiple distance vectors

```
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        a[i+2][j] = a[i+1][j+2] + a[i][j]
```
More complex example

• Can have multiple distance vectors
  
  ```
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      a[i+2][j] = a[i+1][j+2] + a[i][j]
  ```

• Distance vectors
  • (1, -2)
  • (2, 0)

• Important point: order of vectors depends on order of loops, not use in arrays
Problems with distance vectors

• The preceding examples show how distance vectors can precisely summarize all the dependences in a loop nest using just a small number of distance vectors.

• Can’t always summarize as easily!

• Running example:

```c
for (i = 0; i < N; i++)
    a[2*i] = a[i];
```
Loss of precision

- What are the distance vectors for this code?
  - (1), (2), (3), (4) ...

- Note: we have information about the length of each vector, but not about the source of each vector

- What happens if we try to reconstruct the iteration space graph?
Loss of precision

• What are the distance vectors for this code?
  
  • (1), (2), (3), (4) ...

• Note: we have information about the length of each vector, but not about the source of each vector

• What happens if we try to reconstruct the iteration space graph?
Direction vectors

• The whole point of distance vectors is that we want to be able to succinctly capture the dependences in a loop nest

• But in the previous example, not only did we add a lot of extra information, we still had an infinite number of distance vectors

• Idea: summarize distance vectors, and save only the direction the dependence was in

• (2, -1) → (+, −)

• (0, 1) → (0, +)

• (0, -2) → (0, −)  *(can’t happen; dependences have to be positive)*

• Notation: sometimes use ‘<‘ and ‘>’ instead of ‘+’ and ‘−’
Why use direction vectors?

- Direction vectors lose a lot of information, but do capture some useful information
  - Whether there is a dependence (anything other than a ‘0’ means there is a dependence)
  - Which dimension and direction the dependence is in
- Many times, the only information we need to determine if an optimization is legal is captured by direction vectors
  - Loop parallelization
  - Loop interchange