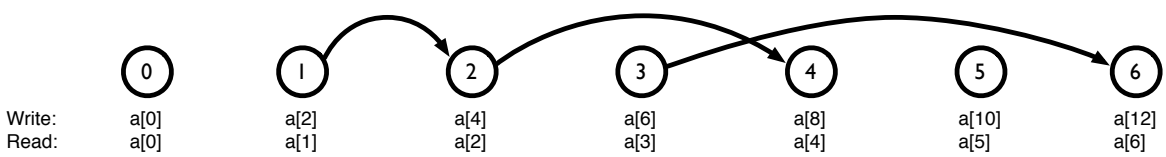
Let us work through a simple example to demonstrate iteration space graphs, distance vectors and dependence vectors. We will do this with a singly-nested loop for simplicity, but hopefully it will be apparent how this applies to doubly nested loops. Consider the following piece of code:

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

First, we construct the iteration space graph. Because this is a singly nested loop, the iteration space is one-dimensional (the `i` dimension). The space looks as follows, with each node labeled by the value of `i` it represents (I've only shown 6 of the nodes for simplicity):

Next, we determine which array elements are read and written in each iteration:

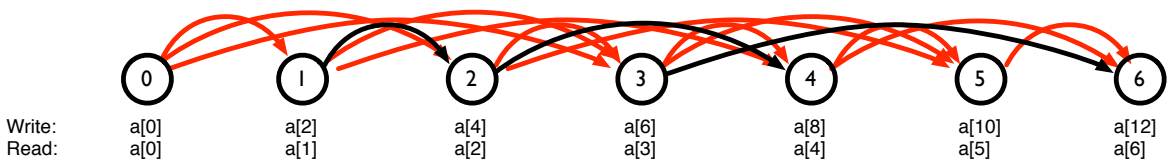
This lets us insert the dependence arrows. Iteration 1 writes to location `a[2]`, and iteration 2 reads from that location, etc. Note that there are other dependence arrows, e.g., from the `i = 5` iteration to the `i = 10` iteration, that are not being shown.



Because we cannot represent this graph completely (since we do not know N), we can consider representing the dependences using distance vectors, instead. For *each* dependence arrow, we determine its corresponding distance vector (how the iteration indices of the source and sink of the arrow are related). For example, the dependence between iteration 1 and iteration 2 is represented by the distance vector (1); the dependence between iteration 2 and iteration 4 is represented by the distance vector (2) (because the iterations are 2 steps apart in the `i` direction).

Here we run into a problem. Consider the set of distance vectors that represents all the dependences in the program: $\{(1), (2), (3)...\}$. Even though we know the length and direction of each dependence vector, we lose information about the actual source and sink for each dependence. We do not know if the distance vector (2) represents a dependence

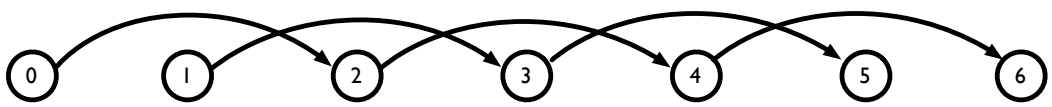
between iteration 2 and iteration 4 or between iteration 4 and iteration 6, and so on. Thus, if we were to try and reconstruct the iteration space graph from just the distance vectors, we would get this (the red arrows represent the dependences that don't actually exist in the program:



Let us consider a simpler example, where distance vectors are useful:

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

Thus, iteration 0 writes to `a[2]`, which is read in iteration 2, iteration 1 writes to `a[3]`, which is read in iteration 3, and so on. The iteration space graph we get is as follows



Now if we try to summarize this information using distance vectors, we see that every dependence has distance 2, so the vector (2) captures every dependence (and doesn't introduce any new ones!).

If we try to summarize this even further by using direction vectors, we see that every dependence is forward in the `i` direction: $(+)$ or $(<)$. However, by going from distance to direction vectors, we lose information again: reconstructing the distance vectors from just the direction vector leads to *all* distance vectors that go forward in the `i` direction: $\{(1), (2), (3)...\}$ — clearly more dependences than actually exist in the program!