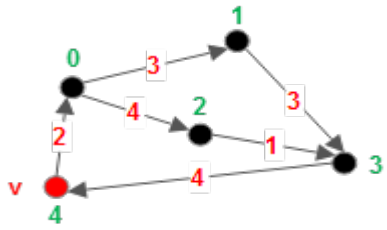
Extending the shortest-path algorithm to calculate shortest paths
David Gries

The shortest-path algorithm calculates the distance of the shortest path from start node v to every node in a graph. We now extend the algorithm to calculate the shortest paths themselves.

This practice often works well: Start with a fairly simple basic algorithm and then extend it to calculate more information.

Consider the graph to the right. It has n = 5 nodes, with node numbers in 0..4, given in green. Red node v = 4 is the start node. The edge weights are given in red. Here are the shortest paths from v to all nodes:



(v, 0)	distance 2
(v, 0, 1)	distance 5
(v, 0, 2)	distance 6
(v, 0, 2, 3)	distance 7 —path (v, 0, 1, 3) has distance 8
(v)	distance 0

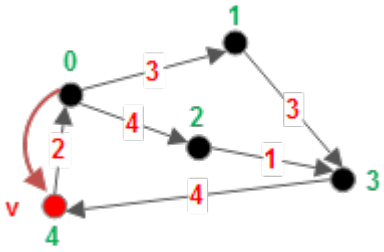
The first problem is to decide how to save the shortest paths. What data structure should be used? The obvious approach is to store information in start node v. For example, use an array c such that for each node w, c[w] contains the neighbor of v on the shortest path to w. Array c is shown to the right. It doesn't matter what is in c[4] since that is the start node and the shortest path contains exactly 1 node, v. All the other elements of c are 0 because all shortest paths from v to other nodes go from v to 0.

```
c[0] = 0
c[1] = 0
c[2] = 0
c[3] = 0
c[4] = ?
```

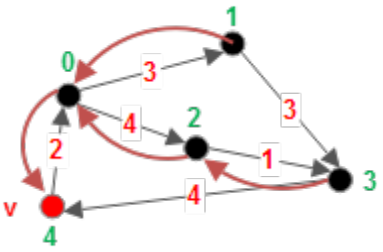
If we continue with this approach, we probably need an array for each node. This approach requires a *lot* of space, and it will probably require a lot of code to create these arrays. There *must* be a better approach. One that requires a lot less space, hopefully one value per node.

Use backpointers!

Instead, we do the following. Consider the shortest path from v to 0: (v, 0). The node *preceding* 0 on this path is node v. We therefore draw a *backpointer* from 0 to v, shown in the diagram to the right as a curved arrow.



We do this for all nodes except the start node. For each node w except v, draw a curved arrow from w to the previous node on the shortest path from v to w. This is shown in the second diagram to the right.



Thus, for each node w, the backpointers give the reverse of the shortest path from v to w. That's neat!

We can maintain these backpointers in an array bp. Thus, we have two arrays:

- d[w] contains the distance of the shortest path from v to w.
- bp[w] contains the previous node on the shortest path from v to w.

For this graph, here are arrays d and bp:

d[0]: 2	bp[0]: 4
d[1]: 5	bp[1]: 0
d[2]: 6	bp[2]: 0
d[3]: 7	bp[3]: 2
d[4]: 0	bp[4]: ?

We can construct the shortest path from v to any node w using the backpointers in time proportional to the distance of the shortest path. That's pretty good. And, for a graph with n nodes, only O(n) space is needed for the backpointers.

Our next task is to modify the shortest path algorithm to create backpointer array bp.

To the right, we give the invariant and theorem of the shortest-path algorithm. Below is the algorithm. It sets d[w] for each node w reachable from start node v to the distance of the shortest path from v to w.

```
F= {v}; d[v]= 0; S= {};
// invariant: P1, P2, and P3
while (F != {}) {
   f= a node in F with minimum d value;
   Remove f from F and add it to S;
   for each w with (f, w) an edge {
      if (w not in S or F) {
           d[w]= d[f] + wgt(f, w);
           add w to F;
      }
      else if d[f] + wgt(f, w) < d[w] {
            d[w]= d[f] + wgt(f, w);
          }
   }
}
```

Invariant
P1. For node s in settled set S, at least one shortest v-s path contains only settled nodes and d[s] is its distance.

P2. For node f in *frontier set* F, at least one v-f path contains only *settled* nodes, except for f, and d[f] is the minimum distance from v to f over all paths from v to f that contain only settled nodes except for the last one, f.

P3. Edges leaving S end in the frontier.

Theorem. For f in *frontier* set F with minimum d value (over nodes in F), d[f] is the shortest-path distance from v to f.

We modify the algorithm to fill in elements of array bp. There are two situations in which a new shortest-path is formed. We investigate them.

1. Case w is not in S or F. Here, w is in the far-out set and is placed in the frontier set. The new shortest path (so far) from v to w is (v, ..., f, w). Therefore, f is the backpointer for w, and the statement bp[w]= f; is needed.
2. Case w is in S or F and d[f] + wgt(f, w) < d[w]. Here, the new shortest path (so far) from v to w is (v, ..., f, w). Therefore, f is the backpointer for w, and the statement bp[w]= f; is needed.

The modified algorithm is given below, with the additional assignments shown in red. Wow! Isn't that simple? Isn't that neat?

```
F= {v}; d[v]= 0; S= {};
// invariant: P1, P2, and P3
while (F != {}) {
   f= a node in F with minimum d value;
   Remove f from F and add it to S;
   for each w with (f, w) an edge {
      if (w not in S or F) {
           d[w]= d[f] + wgt(f, w); bp[w]= f;
           add w to F;
      }
      else if (d[f] + wgt(f, w) < d[w]) {
            d[w]= d[f] + wgt(f, w); bp[w]= f;
          }
   }
}
```

Invariant
P1. For node s in settled set S, at least one shortest v-s path contains only settled nodes and d[s] is its distance.

P2. For f in *frontier set* F, at least one v-f path contains only *settled* nodes, except for f, and d[f] is the minimum distance from v to f over all such paths from v to f.

P3. Edges leaving S end in the frontier.

P4. For w in S or F, bp[w] is the backpointer on the shortest known path from v to w.

Theorem. For f in *frontier* set F with minimum d value (over nodes in F), d[f] is the shortest-path distance from v to f.

This version of the shortest path algorithm uses three sets: settled set S, frontier set F, and the far-off set, which contains all nodes that are not in S or F. It uses two arrays: d and b.

Implementing this algorithm in Java would require finding a good implementation of F (use a heap) and a new data structure to efficiently maintain the information d[w] and bp[w] for each node in S or F. Since this new data structure will contain information about all nodes in S or F, it turns out set S is not needed and need not be implemented.