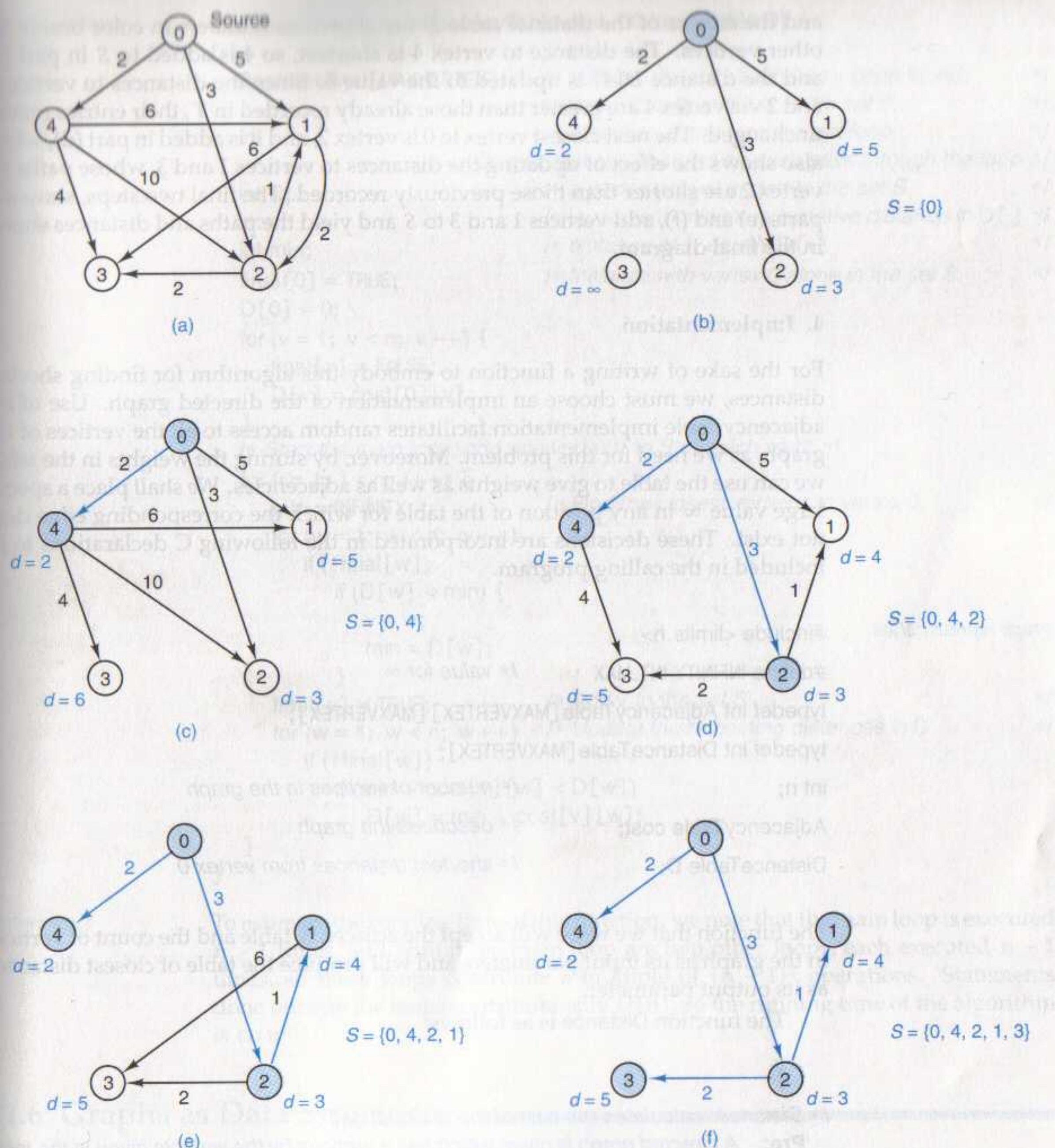

Figure 11.10. Example of shortest paths

and the entries of the distance table D are shown as numbers in color beside the other vertices. The distance to vertex 4 is shortest, so 4 is added to S in part (c), and the distance $D[4]$ is updated to the value 6. Since the distances to vertices 1 and 2 via vertex 4 are greater than those already recorded in T, their entries remain unchanged. The next closest vertex to 0 is vertex 2, and it is added in part (d), which also shows the effect of updating the distances to vertices 1 and 3, whose paths via vertex 2 are shorter than those previously recorded. The final two steps, shown in parts (e) and (f), add vertices 1 and 3 to S and yield the paths and distances shown in the final diagram.

4. Implementation

For the sake of writing a function to embody this algorithm for finding shortest distances, we must choose an implementation of the directed graph. Use of the adjacency-table implementation facilitates random access to all the vertices of the graph, as we need for this problem. Moreover, by storing the weights in the table we can use the table to give weights as well as adjacencies. We shall place a special large value ∞ in any position of the table for which the corresponding edge does not exist. These decisions are incorporated in the following C declarations to be included in the calling program.

graph representation

```
#include <limits.h>
#define INFINITY INT_MAX              /* value for ∞                          */
typedef int AdjacencyTable [MAXVERTEX] [MAXVERTEX];
typedef int DistanceTable [MAXVERTEX];
int n;                                /* number of vertices in the graph      */
AdjacencyTable cost;                  /* describes the graph                  */
DistanceTable D;                      /* shortest distances from vertex 0     */
```

The function that we write will accept the adjacency table and the count of vertices in the graph as its input parameters and will produce the table of closest distances as its output parameter.

The function Distance is as follows:

```
/* Distance: calculates the cost of the shortest path.
   Pre:  A directed graph is given which has n vertices by the weights given in the table
         cost.
   Post: The function finds the shortest path from vertex 0 to each vertex of the graph
         and returns the path that it finds in the array D. */
```

shortest-distance function

```
void Distance(int n, AdjacencyTable cost, DistanceTable D)
{
    Boolean final[MAXVERTEX];  /* Has the distance from 0 to v been found?       */
                               /* final[v] is true iff v is in the set S.        */
    int i;                     /* repetition count for the main loop             */
                       /* One distance is finalized on each pass through the loop. */
    int w;                     /* a vertex not yet added to the set S            */
    int v;                     /* vertex with minimum tentative distance in D[] */
    int min;                   /* distance of v, equals D[v]                     */
    final[0] = TRUE;           /* Initialize with vertex 0 alone in the set S.   */
    D[0] = 0;
    for (v = 1; v < n; v++) {
        final[v] = FALSE;
        D[v] = cost[0][v];
    }
    /* Start the main loop; add one vertex v to S on each pass. */
    for (i = 1; i < n; i++) {
        min = INFINITY;        /* Find the closest vertex v to vertex 0.         */
        for (w = 1; w < n; w++)
            if (!final[w])
                if (D[w] < min) {
                    v = w;
                    min = D[w];
                }
        final[v] = TRUE;       /* Add v to the set S.                            */
        for (w = 1; w < n; w++) /* Update the remaining distances in D.          */
            if (!final[w])
                if (min + cost[v][w] < D[w])
                    D[w] = min + cost[v][w];
    }
}
```

performance

To estimate the running time of this function, we note that the main loop is executed $n - 1$ times, and within the main loop are two other loops, each executed $n - 1$ times, so these loops contribute a multiple of $(n - 1)^2$ operations. Statements done outside the loops contribute only $O(n)$, so the running time of the algorithm is $O(n^2)$.