Graphs Dijkstra's Shortest Path algorithm   Carol Zander

Dijkstra's shortest path algorithm

The optimal algorithm for finding the shortest path in a weighted graph is named after Edsger Dijkstra, who invented it. This algorithm requires that all of the path weights are positive (which simplifies the problem somewhat over general graphs).

The basic idea is keep track of the best-known path to each vertex. We have to keep updating the paths as we visit new vertices. We also have to visit the vertices in a particular order. The unvisited vertex with the lowest weight path is always the one visited next. This is why the algorithm is a *greedy* algorithm.

After initializing, the second pass considers the neighbors of the source. This gives the initial best paths to these vertices. We also know that, for the neighbor with the smallest edge weight from the source, this is the best possible path. For this particular neighbor Vs, we consider whether we can construct any paths that are better than ones currently known by, first, traveling to Vs and, then, to neighbors of Vs. After this step, it turns out that we will have found the best possible path to whichever vertex has the second smallest weight path (after the smallest weight path, which goes to Vs). This vertex with the second smallest weight path is visited next and then the vertex with the next smaller weight path, and, so on, until all of the vertices have been visited.

Variables

Cost array, C, is a 2-dimensional array; `C[i][j]` is the cost on the edge `i`→`j` (if no edge exists, the `C[i][j] = INFINITY`)

T is a table which records the shortest path found so far. It is a one-dimensional array of structs:

```
   struct Table {
      bool visited;         // whether it has been used as a “v” node
      int dist;             // current best distance to vertex
      int path;             // previous vertex in the path to this vertex
   };
```

Algorithm

Initialization:

```
  T[k].dist set to INFINITY except the source is set to zero (all k)
  T[k].visited set to false (all k)
  T[k].path set to zero (all k)

for i = 1 to (#vertices-1) {
   v = not yet visited node with the minimum distance (haven't gone thru yet)
   mark v as visited
   for each w adjacent to v {
      if (w is not visited)
         // adjust the shortest distance from source to w
         // if going through v is shorter, set .path to v
         T[w].dist = min(T[w].dist, T[v].dist + C[v][w])
   }
}
```

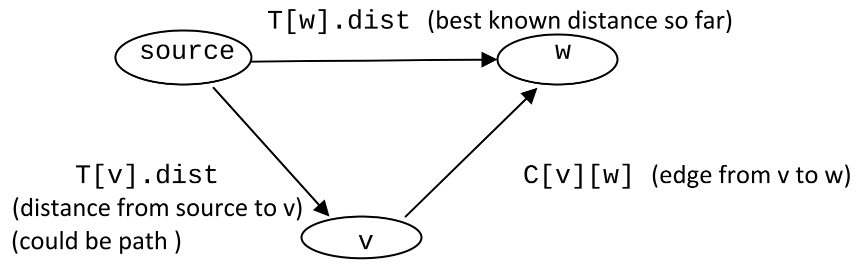
Setting path in the above algorithm: When it is better to go through `v`, i.e., when `T[v].dist + C[v][w])` replaces `T[w].dist` as the shortest distance, set `T[w].path` to v.

The critical line of code is

```
     T[w].dist = min(T[w].dist, T[v].dist + C[v][w])
```

This is asking: should you go through v? In other words, is it better to stick with the shortest distance known so far from the source to w (which is T[w].dist), or is it shorter to go through v (which is (T[v].dist + C[v][w], the distance from the source to v plus the edge cost from v to w))?

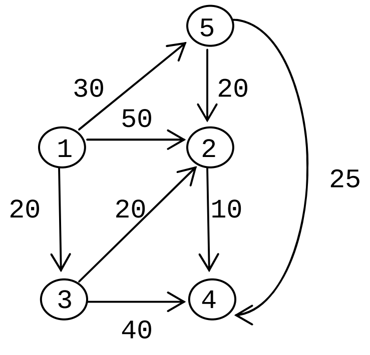
Visually, you're always trying to get from the "source" node to the "w" node.
Should you go through the "v" node? Or not?
Is going through v shorter than the previously known shortest distance?



T[w].dist (best known distance so far)
source
w
T[v].dist
(distance from source to v)
(could be path)
v
C[v][w] (edge from v to w)

Dijkstra’s shortest path algorithm demonstration

Find the shortest distance from the source, node 1, to all other nodes.



```
C:  1   2   3   4   5  ("--" is used for infinity.)
 1 --  50  20  --  30
 2 --  --  --  10  --
 3 --  20  --  40  --
 4 --  --  --  --  --
 3 --  20  --  25  --
```

Consider only row one of the 2-D T array (While the `T[k].dist` is shown as a table, it is only one row, each successive value shown replaces the previous value):

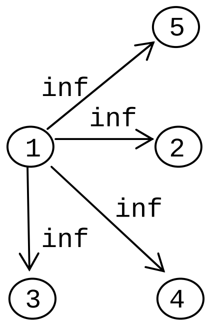
- A line, a bar |, means a node is marked as visited below.
- A value is replaced if `T[v].dist+C[v][w] < T[w].dist.` Comments refer to the computation of the `"for each w adjacent to v"` loop. Detailed arithmetic is shown after final answer.

```
        k: 1   2    3    4    5
T[k].dist: 0  inf  inf  inf  inf  // all are infinity except source to zero
    v = 1  |  50   20   inf  30   // choose v=1: of unmarked, 0 is low
           |
    v = 3  |  40    |   60   30   // choose v=3: of unmarked, 20 is low
           |        |
    v = 5  |  40    |   55    |   // choose v=5: of unmarked, 30 is low
           |        |         |
    v = 2  |   |    |   50    |   // choose v=2: of unmarked, 40 is low
           |   |    |         |
         done done done done done
Final:     0  40   20   50   30
```

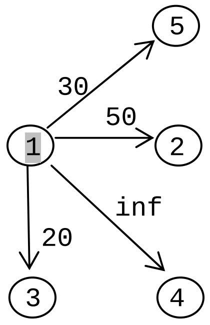
"for each unvisited w adjacent to v" loop details:

```
        T[w].dist   T[v].dist+C[v][w]   Action
v=1                                     mark v=1 visited, never go there again
   w=2     inf          0    +   50     replace inf with 50
   w=3     inf          0    +   20     replace inf with 20
   w=5     inf          0    +   30     replace inf with 30

v=3                                     mark v=3 visited, never go there again
   w=2      50         20    +   20     replace 50 with 40
   w=4     inf         20    +   40     replace inf with 60

v=5                                     mark v=5 visited, never go there again
   w=2      40         30    +   20     leave alone, not shorter through node 5
   w=4      60         30    +   25     replace 60 with 55

v=2                                     mark v=2 visited, never go there again
   w=4      55         40    +   10     replace 55 with 50
```

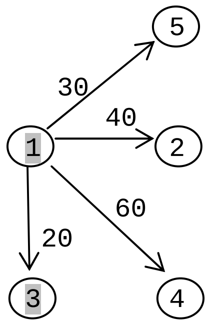
Here's another way to visualize the change. The edges here represent `T[w].dist` as the algorithm is performed. Initially the shortest distance from the source, node 1, to all the other nodes is infinity. As each v is used, the shortest known distances get updated:



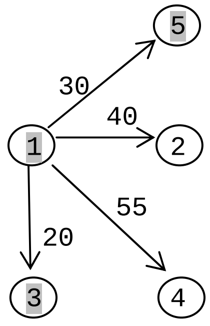
When v is 1, nodes 2, 3, and 5 get their best known shortest distance updated:



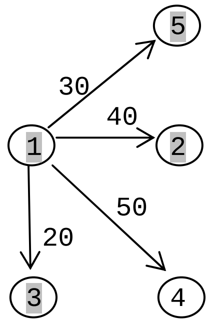
When v is 3, nodes 2 and 4 get their best known shortest distance updated:



When v is 5, node 4 gets its best known shortest distance updated:



When v is 2, node 4 gets its best known shortest distance updated:



Now view the table looking at the .path value. Every time v is used, it is saved in the .path of the node. Here is how the .path changes:

```
        k: 1   2    3    4    5
T[k].path: 0   0    0    0    0
    v = 1  |   1    1    0    1   // when v is chosen to be 1
           |
    v = 3  |   3    |    3    1   // when v is chosen to be 3
           |        |
    v = 5  |   3    |    5    |   // when v is chosen to be 5
           |        |         |
    v = 2  |   |    |    2    |   // when v is chosen to be 2
           |   |    |         |
         done done done done done

Final:     0   3    1    2    1
```

To extract the path from this information, use a simple recursive function, printing on the way back. Here's the idea behind it -- suppose you want the path from the source 1 to 4. Consider 4:

```
   To get to 4, look at the contents of T[4].path, which is 2.
   This says you get to 4 through 2. So, how do you get to 2?

   To get to 2, look at the contents of T[2].path, which is 3.
   This says you get to 2 through 3. So, how do you get to 3?

   To get to 3, look at the contents of T[3].path, which is 1.
   This says you get to 3 through 1. So, how do you get to 1?

   To get to 1, look at the contents of T[1].path, which is 0.
   Base case. Stop at zero.

All the numbers are  4  2  3  1  which is the path backwards.
```