Like Dijkstra's Algorithm, Bellman–Ford is based on the principle of relaxation, in which an approximation to the correct distance is gradually replaced by more accurate values until eventually reaching the optimum solution. In both algorithms, the approximate distance to each vertex is always an overestimate of the true distance, and is replaced by the minimum of its old value with the length of a newly found path. However, Dijkstra's algorithm greedily selects the minimum-weight node that has not yet been processed, and performs this relaxation process on all of its outgoing edges; by contrast, the Bellman–Ford algorithm simply relaxes all the edges, and does this times, where is the number of vertices in the graph. In each of these repetitions, the number of vertices with correctly calculated distances grows, from which it follows that eventually all vertices will have their correct distances. This method allows the Bellman–Ford algorithm to be applied to a wider class of inputs than Dijkstra.

Bellman–Ford runs in time, where and are the number of vertices and edges respectively.

```
function BellmanFord(list vertices, list edges, vertex source)
  ::distance[],predecessor[]

  // This implementation takes in a graph, represented as
  // lists of vertices and edges, and fills two arrays
  // (distance and predecessor) with shortest-path
  // (less cost/distance/metric) information

  // Step 1: initialize graph
  for each vertex v in vertices:
     if v is source then distance[v] := 0
     else distance[v] := inf
     predecessor[v] := null

  // Step 2: relax edges repeatedly
  for i from 1 to size(vertices)-1:
     for each edge (u, v) with weight w in edges:
        if distance[u] + w < distance[v]:
           distance[v] := distance[u] + w
           predecessor[v] := u

  // Step 3: check for negative-weight cycles
  for each edge (u, v) with weight w in edges:
     if distance[u] + w < distance[v]:
        error "Graph contains a negative-weight cycle"
  return distance[], predecessor[]
```

from a single source vertex

Simply put, the algorithm initializes the distance to the source to 0 and all other nodes to infinity. Then for all edges, if the distance to the destination can be shortened by taking the edge, the distance is updated to the new lower value. At each iteration that the edges are scanned, the algorithm finds all shortest paths of at most length edges. Since the longest possible path without a cycle can be edges, the edges must be scanned times to ensure the shortest path has been found for all nodes. A final scan of all the edges is performed and if any distance is updated, then a path of length edges has been found which can only occur if at least one negative cycle exists in the graph.

Graph algorithms - Bellman-Ford algorithm

Problem

Given a graph with no negative costs cycles and two vertices *s* and *t*, find a minimum cost walk from *s* to *t*.

- with negative costs.

Idea

The algorithm keeps two mappings:

- *dist[x]* = the cost of the minimum cost walk from *s* to *x* known so far
- *prev[x]* = the vertex just before *x* on the walk above.

Initially, *dist[s]*=0 and *dist[x]*=∞ for $x \neq s$; this reflects the fact that we only know a zero-length walk from *s* to itself.

Then, we repeatedly performs a *relaxation* operation defined as follows: if *(x,y)* is an edge such that *dist[y] > dist[x] + cost(x,y)*, then we set:

- dist[y] = dist[x] + cost(x,y)
- prev[y] = x

The idea of the relaxation operation is that, if we realize that we have a better walk leading to *y* by using *(x,y)* as its last edge, compared to what we know so far, we update our knowledge.

The algorithm

```
Input:
    G : directed graph with costs, V-the set of vertices, E-the set of edges
        cost - the  cost of edges
    s, t : two vertices
Output:
    dist : a map that associates, to each accessible vertex,
           the cost of the minimum  cost walk from s to it
    prev : a map that maps each accessible vertex to its predecessor
            on a path from s to it
Algorithm:
    for x in V do
        dist[x] = ∞
    end for
    dist[s] = 0
    changed = true
    while changed do
        changed = false
        for (x,y) in E do
            if dist[y] > dist[x] + cost(x,y) then
                dist[y] = dist[x] + cost(x, y)
                prev[y] = x
                changed = true
            end if
        end for
    end while
```

and i<n.

→ for k = 1, |V|-1

identify negative cycles:
for (x,y) in E do
if dist [y] > dist[x]+cost(x,y
then print "negative cycl

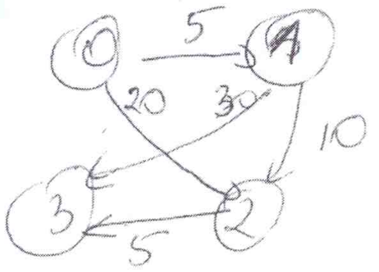
from prev generate the path.

Proof of correctness: it is in three parts:

- at each stage, *dist* and *prev* correspond to existing walks (this comes immediately from how the relaxation operation works;
- the algorithm finishes;
- when the algorithm finishes, *dist[x]* = *d(s,x)* for all vertices *x*.

For the last two parts, we notice that, at iteration *k*, we have that *dist[x]* $\leq w_{k,x}$ (see the Bellman's dynamic programming algorithm). This makes the Bellman-Ford finish in at most *n-1* iterations and end with the correct distances.

's algorithm



s = 0, t = 3

	changed	edge: (x,y)	dist: dictionary	prev: dictionary
initialitation	true		0 1 2 3 0 ∞ ∞ ∞	
iteration 1	false true true true true true	 (0,1) (0,2) (1,2) (1,3) (2,3)	0 1 2 3 0 5 ∞ ∞ 0 5 20 ∞ 0 5 15 ∞ 0 5 15 35 0 5 15 20	0 1 2 3 _ 0 _ _ _ 0 0 _ _ 0 1 _ _ 0 1 1 _ 0 1 2
iteration 2	false	(0,1) (0,2) (1,2) (1,3) (2,3)	0 5 15 20 0 5 15 20 0 5 15 20 0 5 15 20 0 5 15 20	_ 0 1 2 _ 0 1 2 _ 0 1 2 _ 0 1 2 _ 0 1 2

=> stop

The minimum cost walk from s=0 to t = 3 has the cost = dist[3] = 20 and it is built backwards from prev dictionary:

t = 3, prev [3] = 2, prev[2] = 1, prev[1] = 0 = s

walk: $0 \xrightarrow{5} 1 \xrightarrow{10} 2 \xrightarrow{5} 3$