Shortest Paths in Graphs and Digraphs: Bellman-Ford Algorithm

Textbook Reading from *Algorithms: Foundations and Design Strategies*

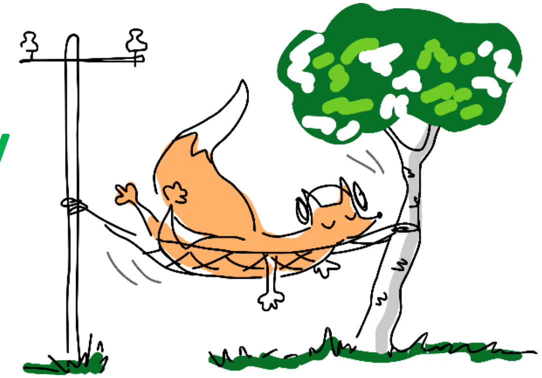
- Chapter 8, Section 8.6, pp. 360-366.

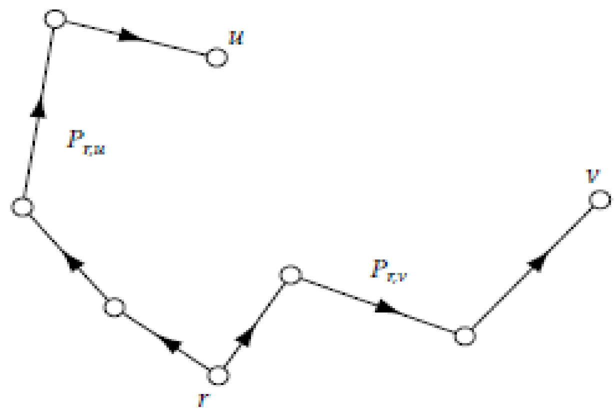
Arbitrary Real Weights

- Dijkstra’s algorithm may fail to output shortest paths when negative weights are allowed.
- The Bellman-Ford algorithm works for negative edge weights, i.e., the weight $c(e)$ of edge e can be any real number.
- If there is no **negative cycle**, i.e., a cycle such that the sum of the weights on the edges is negative, the Bellman-Ford algorithm will output a shortest path from the root vertex r to all the other vertices.
- Otherwise, it will determines that there is a negative cycle. Note that if there is a negative cycle, by repeatedly traversing this cycle we can obtain a walk whose weight is arbitrarily small.

- Like Dijkstra's algorithm the Bellman-Ford shortest-path algorithm maintains a tree directed out of a given root vertex *r*, which we implement using its parent array.
- It performs *n* – 1 passes, each pass scanning all the edges and **relaxing** an edge when encountered.

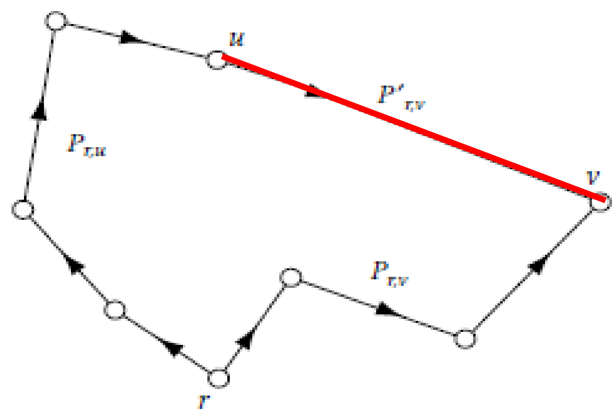
Relax Operation for Edge uv





In Bellman–Ford Algorithm just before considering edge *uv*

(a)



In Bellman–Ford Algorithm after considering edge *uv*.
We compare weight of $P_{r,v}$ with weight of $P'_{r,v}$
and update *Dist*[*v*] with smaller of the two

(b)

Relax Operation for Edge *uv*

if $Dist[u] + c(uv) < Dist[v]$ **then**

$\quad Parent[v] \leftarrow u$

$\quad Dist[v] \leftarrow Dist[u] + c(uv)$

endif

Virtual Edges

As with Dijkstra's algorithm, it is convenient to add the virtual edge (r, v) giving it the weight ∞ in the case when there is no directed path from r to v.

Psuedocode for BellmanFord

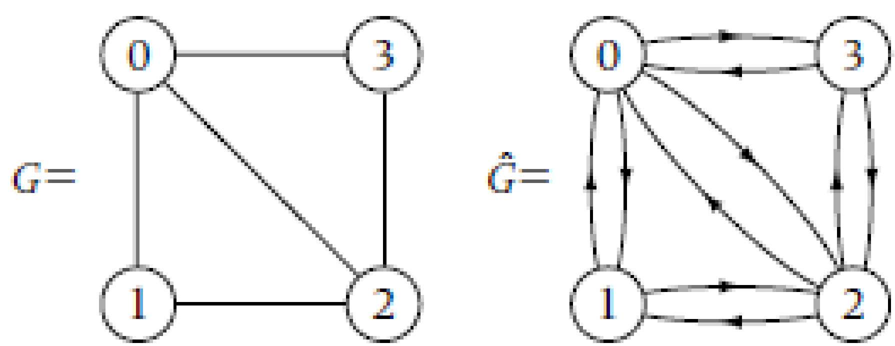
```
procedure BellmanFord(D, r, c, Dist[0:n – 1], Parent[0:n – 1], NegativeCycle)
Input: D (a digraph with vertex set V = {0,...,n – 1} and edge set E)
       c (a weighting of the edges with real numbers)
       r (a vertex of D)
Output:  Parent[0:n – 1] (an array implementing a shortest path tree rooted at r)
         Dist[0:n – 1] (an array of distances from r)
         NegativeCycle      (a Boolean variable having the value .true. if, and
                                 only if, there exists a negative cycle)
       for i ← 0 to n – 1 do          {initialize Dist[0:n – 1] and Parent[0:n – 1]}
            Dist[i] ← ∞
            Parent[i] ← ∞
       endfor
       Dist[r] ← 0
       Parent[r] ← -1
       for Pass ← 1 to n – 1 do      // update Dist[0:n – 1] and  Parent[0:n – 1]
            for each edge uv ∈ E do          // by scanning all the edges
                 if Dist[u] + c(uv) < Dist[v] then
                      Parent[v] ← u
                      Dist[v] ← Dist[u] + c(uv)
                 endif

            endfor
       endfor
       NegativeCycle ← .false. {check for negative cycles}
       for each edge uv ∈ E do
            if Dist[v] > Dist[u] + c(uv) then
                 NegativeCycle ← .true.
            endif
       endfor
end BellmanFord
```

Scan Order of Edges

The order in which the edges are scanned at each iteration doesn’t matter in the sense that a shortest path tree is obtained in the end.

Undirected Graphs

- As we've discussed before the problem of finding a shortest path in a digraph D generalizes the problem of finding a shortest path in an undirected graph G.
- Given a any graph G we can associated the equivalent symmetric digraph $\hat{G}$, where each undirected edge $\{u, v\}$ is replace with the two directed edges (u, v) and (*v*,*u*). Both (u, v) and (v, u) are given the same weight at $\{u, v\}$.



Undirected Graphs cont'd

- Note that for an undirected graph all weights must be nonnegative.
- This is because an edge of negative weight corresponds to a negative cycle of length 2 in the associated symmetric digraph.
- So the algorithm would waste a lot of time just to output that there is a negative cycle or equivalently that one of the edges has negative weight.

Key Observations for Correctness

- After completing k scans through the edges, i.e., k iterations of *BellmanFord*, $Dist[v]$ is no larger than the minimum length $L_{v,k}$ of a path from r to v having at most k edges (where $L_{v,k} = \infty$ if no such path exists), $k = 0, \ldots, n - 1$.

- Therefore, after $n - 1$ scans $Dist[v]$ is the length of a shortest path from r to v.

Complexity of Bellman-Ford

Since Bellman-Ford performs $n - 1$ iterations each iteration scanning all the edges it has worst-case complexity

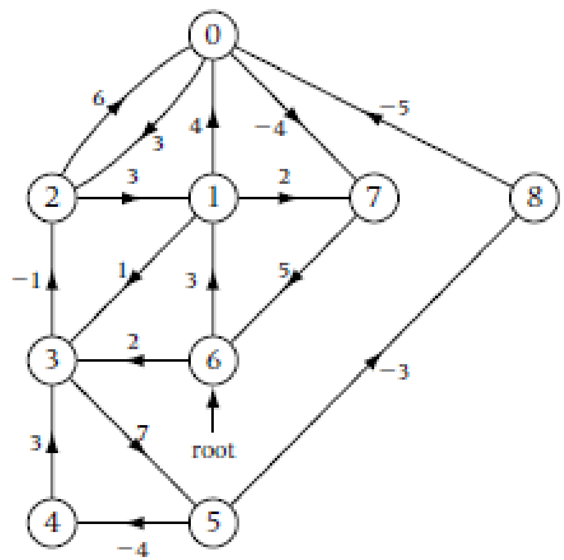
$$W(m,n) \in \Theta(mn).$$

SHORT STOP

- A flag can be added to check whether any edges are relaxed, i.e., distance on head is reduced, during a scan of the edges.
- If no distance has changed during a scan then the algorithm can be terminated.
- This requires a small amount of extra computing time but can be a big savings in the best case, i.e., reduce the computing time to linear in the number of edges.

Action of procedure *BellmanFord* for a sample weighted digraph and root vertex $r = 6$, where edges (u, v) are scanned in lexicographical order

Pass		0	1	2	3	4	5	6	7	8		0	1	2	3	4	5	6	7	8
0	*Dist*:	∞	∞	∞	∞	∞	∞	0	∞	∞	*Parent*:	∞	∞	∞	∞	∞	∞	-1	∞	∞
1	*Dist*:	∞	3	∞	2	∞	∞	0	∞	∞	*Parent*:	∞	6	∞	6	∞	∞	-1	∞	∞
2	*Dist*:	1	3	1	2	5	9	0	5	6	*Parent*:	8	6	3	6	5	3	-1	1	5
3	*Dist*:	1	3	1	2	5	9	0	-3	6	*Parent*:	8	6	3	6	5	3	-1	0	5
4	*Dist*:	1	3	1	2	5	9	0	-3	6	*Parent*:	8	6	3	6	5	3	-1	0	5
5	*Dist*:	1	3	1	2	5	9	0	-3	6	*Parent*:	8	6	3	6	5	3	-1	0	5
6	*Dist*:	1	3	1	2	5	9	0	-3	6	*Parent*:	8	6	3	6	5	3	-1	0	5
7	*Dist*:	1	3	1	2	5	9	0	-3	6	*Parent*:	8	6	3	6	5	3	-1	0	5
8	*Dist*:	1	3	1	2	5	9	0	-3	6	*Parent*:	8	6	3	6	5	3	-1	0	5



What is a computer's favorite snack?

Computer chips.