najkratšie cesty z daného vrchola do všetkých vrcholov orientovaného hranovo ohodnoteného grafu

- **Bellmanov – Fordov algoritmus**
- **Dijkstrov algoritmus**

INITIALIZE-SINGLE-SOURCE(G, s)

for each vertex $v \in V[G]$
    do $d[v] \leftarrow \infty$
        $\pi[v] \leftarrow$ NIL
$d[s] \leftarrow 0$

RELAX(u, v, w)

if $d[v] > d[u] + w(u, v)$
    then $d[v] \leftarrow d[u] + w(u, v)$
        $\pi[v] \leftarrow u$

BELLMAN-FORD(G, w, s)

INITIALIZE-SINGLE-SOURCE(G, s)
for $i \leftarrow 1$ **to** $|V[G]| - 1$
    do for each edge $(u, v) \in E[G]$
        do RELAX(u, v, w)
for each edge $(u, v) \in E[G]$
    do if $d[v] > d[u] + w(u, v)$
        then return FALSE
return TRUE

A Simple Recurrence

- The input digraph can have negative costs, but nonegative cycles
  - Still positive cycles possibly!
- Recurrence like the one used in critical path
  - $d(a, a) = 0$
  - $d(a, v) = \min\{d(a, u) + c(u,v)\}, u \in Pred(v)$
    - Pred(v) denotes the set of direct predecessors of v
    - i.e. <u, v> is an edge
- The recurrence cannot be used due to possible cycles!
  - For d(a, v), v can be involved in a cycle, recursion on v can be forever
- If G is a DAG, then this recurrence works as the basis for the topological sorting based algorithm of *O(n+m)*

Bellmanov-Mooreov algoritmus (varianta Bellmanovho – Fordovho algoritmu)

- $d_k(a, v)$ = the cost of a shortest path from a to v among all paths of length at most k
  - Number of edges involved ≤k
- If no such a path from a to v of length at most k, $d_k(a, v)=\infty$
- New recurrence
  - $d_0(a, a) = 0$
  - $d_0(a, v) = \infty$
  - $d_k(a, v) = \min\{d_{k-1}(a, v), \min_{u \in Pred(v)}\{d_{k-1}(a, u) + c(u, v)\}\}$
- Bellman-Moore algorithm
  - Repeatedly compute $d_k(a, v)$ for k = 0, 1, 2, …
  - For a digraph with n nodes, the largest k to consider is n-1

Complexity of Bellman-Moore Alg

-
- Cost to compute one $d_k(a, v)$ is *O(|Pred(v)|)*
- Costto computeonecolumn is *O(n+m)*
  - We have *O(n)*columns
- Total cost is *O(n·(n+m))*in the worst case

Negative Cycle Detection

- How to decide whether a digraph has a negative cycle or not?
- Use Bellman-Moore algorithm and calculate also the column k=n
- The graph has a negative cycle if and only if
  - $d_n(a, v) < d_{n-1}(a, v)$ for some vertex v

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S ← Ø
Q ← V[G]
while Q ≠ Ø
     do u ← EXTRACT-MIN(Q)
         S ← S ∪ {u}
         for each vertex v ∈ Adj[u]
             do RELAX(u, v, w)
```

Dijkstra’sAlgorithm Preliminaries

- Assumption
  - No edge has negative cost
- Priority queue
  - Like a symbol table, a set of entries
  - Each entry has form of <key, value>
  - Special operations
  - find_min:ENTRY_TYPE
  - Returns the entry in the priority queue whose key is minimum
  - delete_min:ENTRY_TYPE
  - Deletes and returns the entry that would be returned by find_min
  - We use a priority queue when we are more concerned about items with extreme keys than the total order among all keys
  - In shortest path finding, we care about the shortest path so far, not the order of all possible paths w.r.t. their lengths

Dijkstra’sAlgorithm

- Each vertex has fields
  - v.visited, v.parent, v.distance
- A priorityqueueq
  - When one vertex v is inserted into q, we regard v.distance as the key and v itself as the value, i.e., <v.distance, v>
- dijkstra(g:DIGRAPH, a:VERTEX_TYPE)

1. for each vertex v, set v.visited:=FALSE, v.distance:=∞
2. a.distance:=0; a.visited:=TRUE; a.parent:=NIL; q.insert(a)
3. **while** NOT q.empty**do**

v:=q.delete_min

for each edge (v, w) **do**

ifNOT w.visited **then**

–w.visited:=TRUE; w.parent:=v; w.distance:=v.distance+c(v, w); q.insert(w)

else if v.distance+c(v, w) < w.distance **then**

–w.distance:=v.distance+c(v, w); w.parent:=v

Correctness of Dijkstra’sAlgorithm

- Loop Invariant
  - □z□L: z.distance=d(a, z)
  - □z□L: each successor of z is either in L or in Q
  - □z□Q: z.distanceis the length of one shortest path from node a to node z via vertices in L
- Prove it by induction on k, loop iteration

Complexity of Dijkstra’sAlgorithm

- Cost of step 1 is *O(n)*
- Cost of step 2 is *O(1)*
- Cost of step 3 is *O(n+m)*+ time cost on the priority queue
  - delete_min: *n*times
  - insert: *n*times
  - distance update: *m*times
- Total cost
  - O(n+ m + n·(T(delete_min)) + n·(T(insert)) + m·(T(dist_update)))