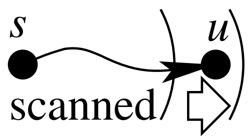
Algorithm and Data Structure Analysis (ADSA)

Shortest Paths2

Pseudocode Dijkstra

```
Function Dijkstra(s : NodeId) : NodeArray×NodeArray              // returns (d, parent)
    d = ⟨∞,...,∞⟩ : NodeArray of ℝ ∪ {∞}                          // tentative distance from root
    parent = ⟨⊥,...,⊥⟩ : NodeArray of NodeId
    parent[s] := s                                                // self-loop signals root
    Q : NodePQ                                                    // unscanned reached nodes
    d[s] := 0;    Q.insert(s)
    while Q ≠ ∅ do
        u := Q.deleteMin                                          // we have d[u] = μ(u)
        foreach edge e = (u,v) ∈ E do
            if d[u] + c(e) < d[v] then                            // relax
                d[v] := d[u] + c(e)
                parent[v] := u                                    // update tree
                if v ∈ Q then Q.decreaseKey(v)
                else Q.insert(v)
    return (d, parent)
```



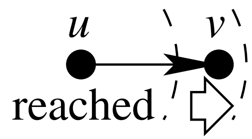


Fig 10.6 Mehlhorn/Sanders

Runtime

- Initialization (arrays, priority queue) takes time O(n).
- Every reachable note is inserted and removed once from Q.
- At most n deleteMin and insert operations.
- Each node is scanned at most once and each edge is relaxed at most once.
- Implies at most m decreaseKey operations.

Total runtime

$$T_{\text{Dijkstra}} = \mathrm{O}\Big(m \cdot T_{decreaseKey}(n) + n \cdot (T_{deleteMin}(n) + T_{insert}(n))\Big)$$

Runtime

Runtime depends on implementation of priority queue.

Original (Dijkstra 1959):

- Maintain the number of reached unscanned nodes.
- An array d storing the distances and an array storing for each node whether it is reached or unscanned.
- Insert and decreaseKey take time $O(1)$
- DeleteMin takes time $O(n)$
- Total Runtime: $O(m+n^2)$

Improvements:

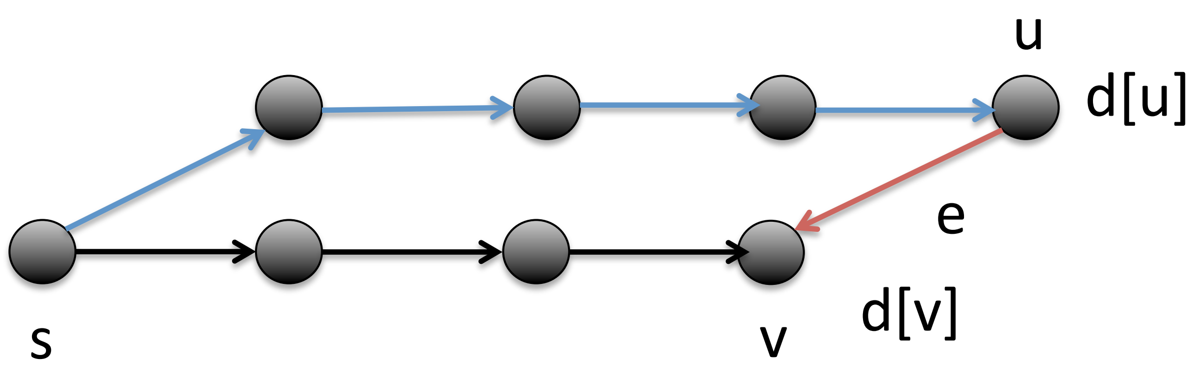
- Binary Heaps: $O((m+n) \log n)$
- Fibonacci Heaps: $O(m + n \log n)$

Bellman-Ford Algorithm

- Dijkstra's algorithm works for acyclic graphs and for non-negative edge costs.
- Bellman-Ford algorithm solves the problem for arbitrary edge costs.
- It uses n-1 rounds and relaxes in each round all edges.
- This works as simple paths have at most n-1 edges.
- After the relaxations are complete, we have all shortest paths to nodes with non-negative cycles.
- We still need to identify the nodes that can be reached by using negative cycles.

Updating

We may to update a previous path from s to v if we find a shorter path

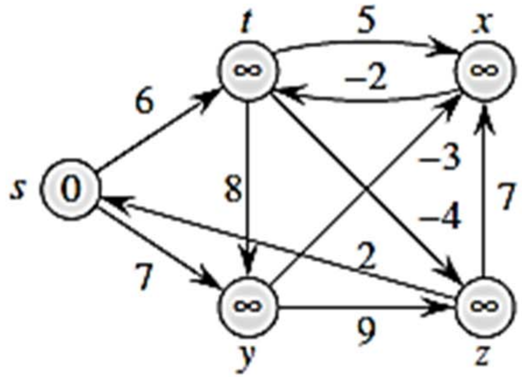


Procedure $relax(e = (u,v) : Edge)$

  if $d[u] + c(e) < d[v]$ **then** $d[v] := d[u] + c(e); \quad parent[v] := u$

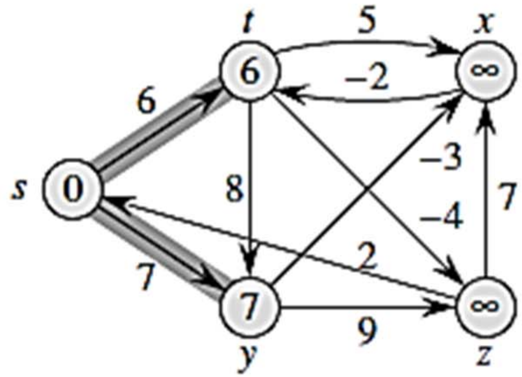
Finding Nodes with Negative Cycles

- Assume that there is an edge e=(u,v) that allows to improve d[v] after the relaxations are complete.
- Then the node v is reachable by using a negative cycle.
- Furthermore, all nodes reachable from v can also be reached by using a negative cycle.
- We set d[v] = -∞ for these nodes v.
- We use postprocessing and the function infect to find nodes reachable by negative cycles.



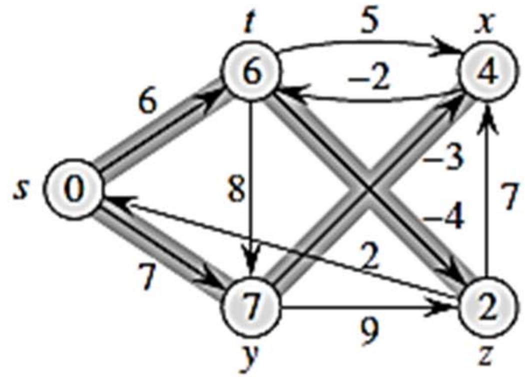
t
x
s
y
z
5
–2
6
8
–3
–4
7
7
2
9
0
∞

(a)



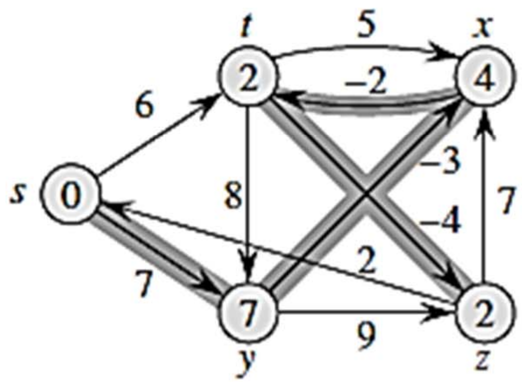
t
x
s
y
z
5
–2
6
8
–3
–4
7
7
2
9
0
6
7
∞

(b)



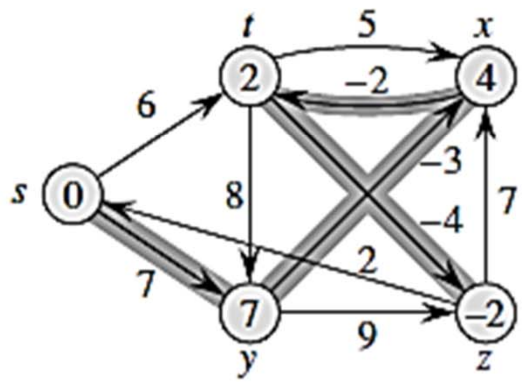
t
x
s
y
z
5
–2
6
8
–3
–4
7
7
2
9
0
6
4
7
2

(c)



t
x
s
y
z
5
–2
6
8
–3
–4
7
7
2
9
0
2
4
7
2

(d)



t
x
s
y
z
5
–2
6
8
–3
–4
7
7
2
9
0
2
4
7
–2

(e)

Bellman-Ford Algorithm

Function $BellmanFord(s : NodeId) : NodeArray \times NodeArray$
$\quad d = \langle\infty,\ldots,\infty\rangle \; : NodeArray$ **of** $\mathbb{R} \cup \{-\infty,\infty\}$ // distance from root
$\quad parent = \langle\bot,\ldots,\bot\rangle \; : NodeArray$ **of** $NodeId$
$\quad d[s] := 0; \quad parent[s] := s$ // self-loop signals root
$\quad$ **for** $i := 1$ **to** $n-1$ **do**
$\quad\quad$ **forall** $e \in E$ **do** $relax(e)$ // round i
$\quad$ **forall** $e = (u,v) \in E$ **do** // postprocessing
$\quad\quad$ **if** $d[u] + c(e) < d[v]$ **then** $infect(v)$
$\quad$ **return** $(d, parent)$

Procedure $infect(v)$
$\quad$ **if** $d[v] > -\infty$ **then**
$\quad\quad d[v] := -\infty$
$\quad\quad$ **foreach** $(v,w) \in E$ **do** $infect(w)$

Fig 10.9 Mehlhorn/Sanders

Runtime O(nm)

All-pairs-shortest-paths (APSP)

Given a directed graph G=(V,E) and a cost function $c: E \rightarrow R$ on the edges.

Compute for each pair of nodes i and j a shortest path.

- Negative-cost edges are allowed, but the graph contains no negative-weight cycles.
- The nodes are labeled 1, …,n.
- We set c(i,j) = ∞ if there is no edge from i to j in G.
- Otherwise, c(i,j) is the cost of the edge from i to j.

Dynamic Programming

Dynamic Programming is powerful approach to solve problems of special structure.

Approach:

- Define and solve subproblems
- Combine solutions
- Subproblems are not independent (they share subsubproblems)
- Solve every subsubproblem just once and store the answer
- Avoid recomputation

Dynamic Programming

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution bottom up.
4. Construct an optimal solution from computed information.

Properties of subpaths

Lemma: Subpaths of a shortest path are also shortest paths.

Proof (by contradiction):

- Assume that the path P is a shortest path from s to v.
- Assume that a subpath from a to b is not a shortest path from a to b

- This implies that there is a shorter path from a to b
- We can use this path to obtain a shorter path from s to v.
- Contradiction to P is shortest path from s to v.

□

Floyd-Warshall Algorithm

Idea: Compute the shortest path from i to j using only the intermediate nodes 1, ..., k.

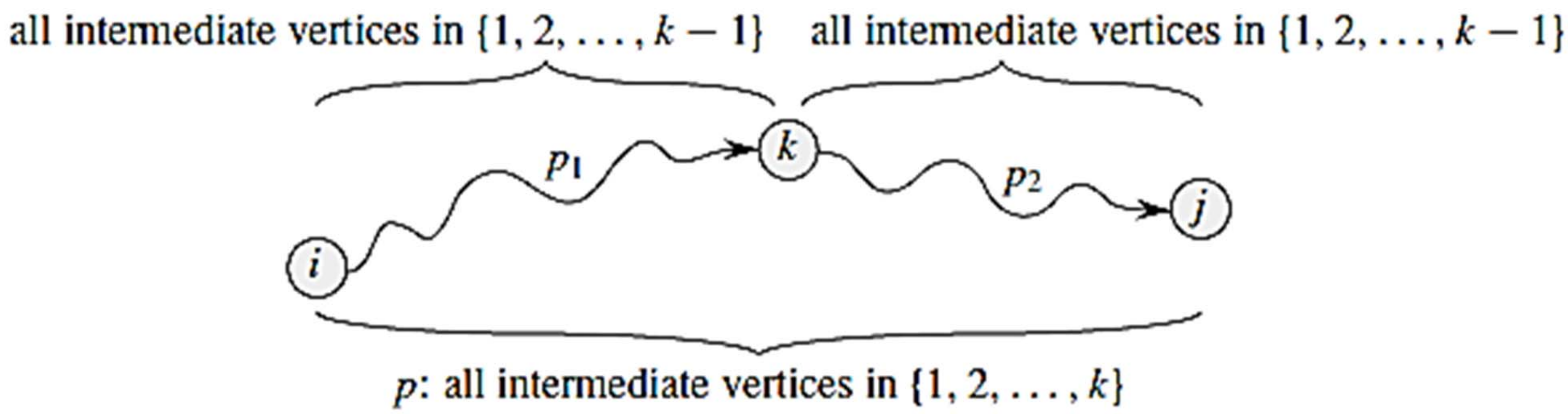
Do this for every k, $0 \leq k \leq n$.

Notation:

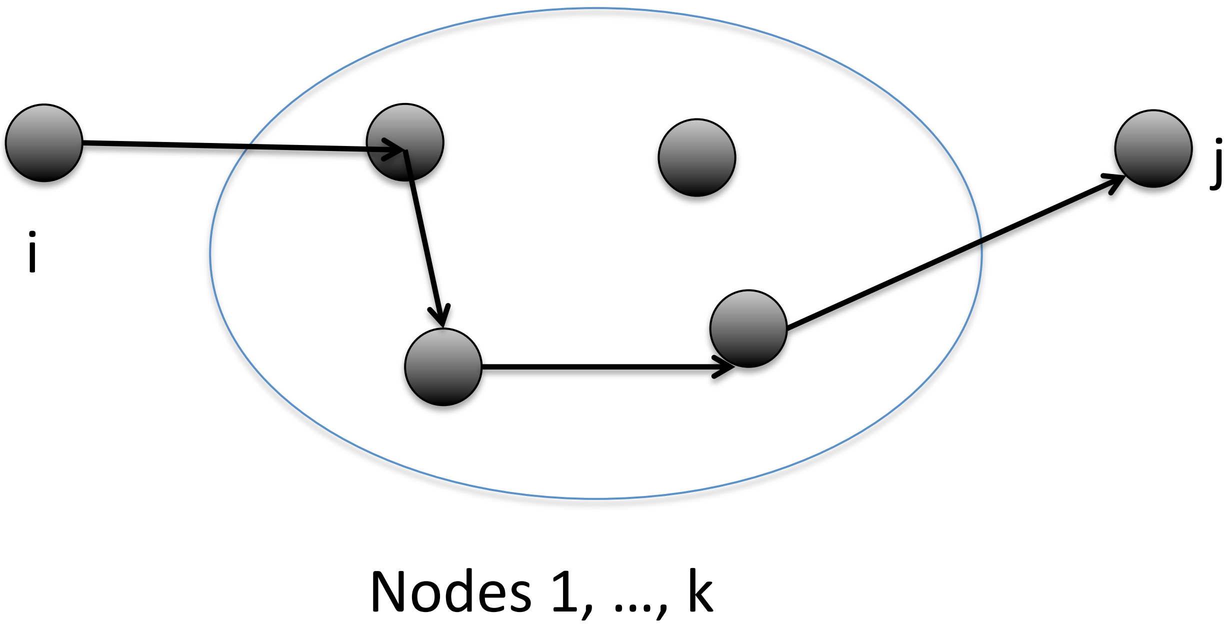
- $d_k(i,j)$ is the cost of a shortest path from i to j if only nodes from the set {1, …., k} are used.
- $N_k(i,j)$: denotes the successor of i in such a path.

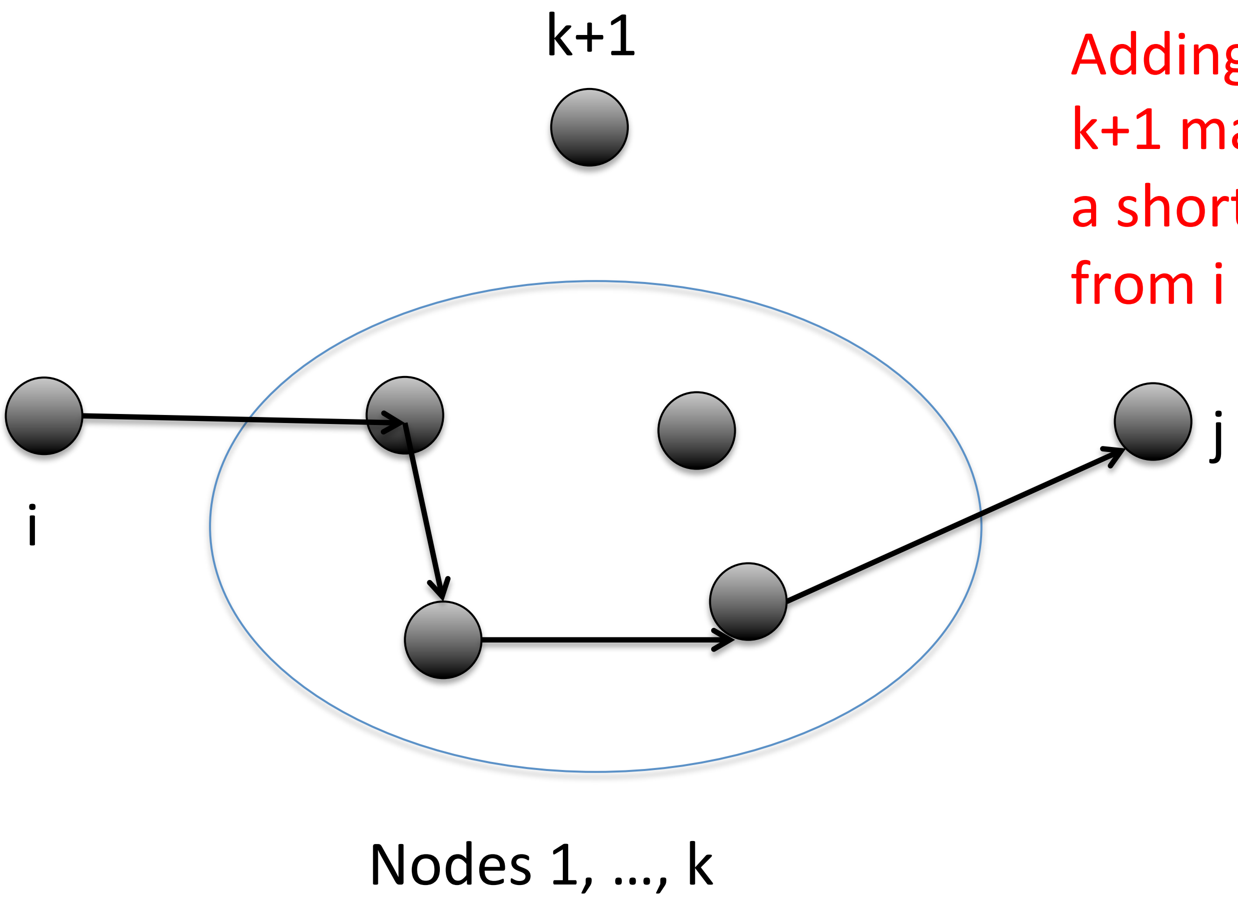
Compute the entries of d and N for the different values of i,j,k

- k=0: no intermediate node is allowed and we have $d_0(i,j) = c(i,j)$ and
  $$N_0(i,j) = j$$
- k+1: we have to check whether node k+1 can be used for a shorter path.

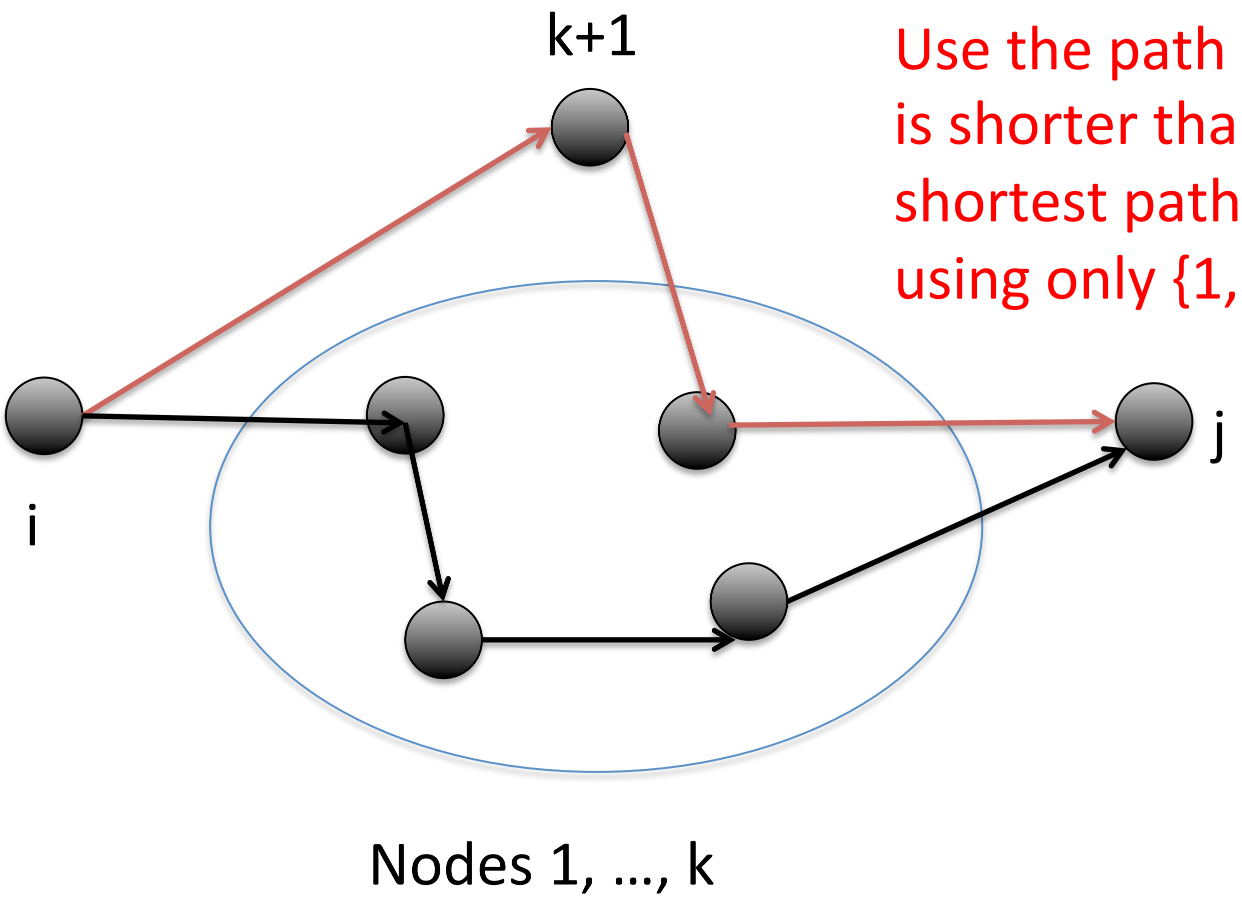


Shortest path from i to j using the nodes 1, …, k





Adding the node k+1 may lead to a shorter path from i to j.



Use the path via k+1 if it is shorter than the shortest path from i to j using only {1, …, k}.

Two possibilities:

- The node k+1 does not improve the shortest path from i to j.
- We get a shorter path by going from i to k+1 and from k+1 to j.

Taking the best option, we get

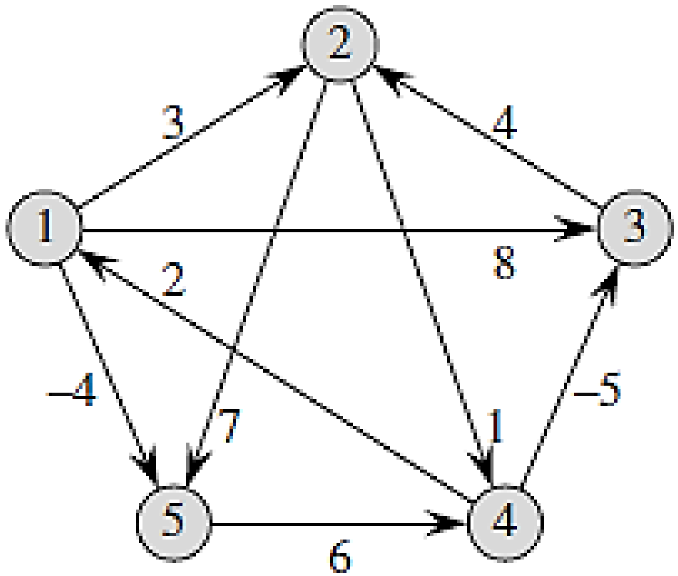
$$d_{k+1}(i,j) = \min\{d_k(i,j), d_k(i,k+1) + d_k(k+1,j)\}$$

- The length of a shortest path from i to j in the given graph G is $d_n(i,j)$

Successors

- If $d_k(i,j) \leq d_k(i,k+1) + d_k(k+1,j)$
  we set $N_{k+1}(i,j) = N_k(i,j)$
- Else we set

$$N_{k+1}(i,j) = N_k(i,k+1)$$



$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

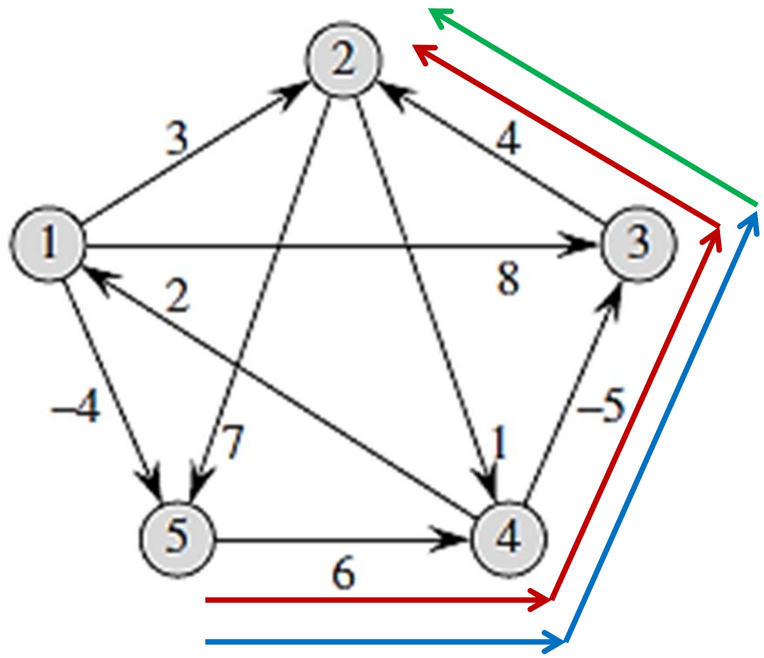
$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$



$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Runtime

- We need to compute all the entries

$$d_k(i,j), 1 \leq i,j,k \leq n$$
$$N_k(i,j), 1 \leq i,j,k \leq n$$

- For k=0, we can set the values directly.
- Each entry for k+1 can be computed in constant time if we have already all entries for k.
- There are $O(n^3)$ entries that have to be computed.
- Total runtime is $O(n^3)$.