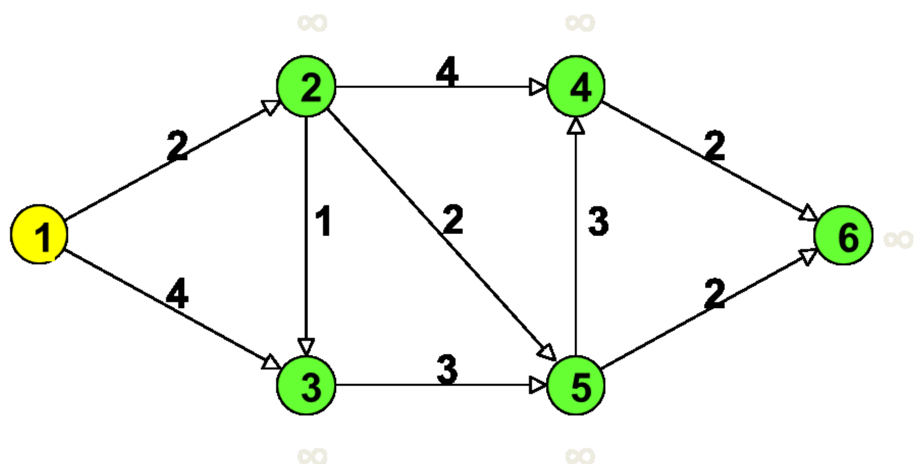
第二章 网络中的若干优化问题

第一节 最短路径问题

最短路径问题的定义

- Given a directed network G=(N,A), a source node *s*, an arc length c_{ij} associated with each arc (*i,j*). The shortest path problem is to find the shortest length directed path (shortest path for short) from s to every other node in the network.



$$\text{Min} \sum_{ij\in A} c_{ij}x_{ij}$$

s.t.

$$\sum_{ij\in A} x_{ij} - \sum_{ji\in A} x_{ji} = \begin{cases} n-1, for \\ -1, for\ all\ i \in N-\{s\} \end{cases}$$

$$x_{ij} \geq 0, ij \in A$$

最短路径问题的意义

- 最短路径问题在网络流问题中具有极其重要的意义：
  - 它们本身有着极为广泛的应用场景
  - 它们是诸多的组合优化和网络优化问题的子问题，因而是学习网络流问题的起点
- 解决最短路径问题是解决很多其他复杂问题的基础

本节目录

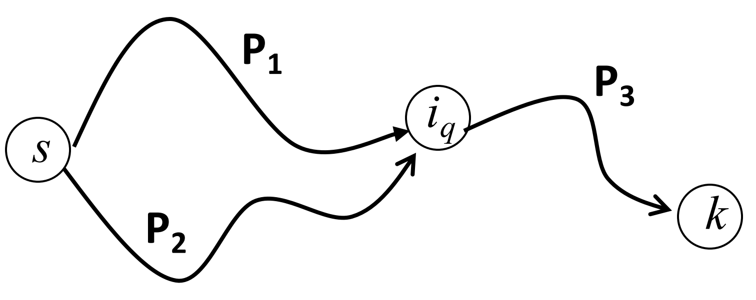
1. Shortest path property
2. Shortest path problems in acyclic networks
   - Reaching algorithm
3. Shortest path problems in cyclic networks
   - ☐ with non-negative arc costs
   - Label setting algorithm (Dijkstra's algorithm)
   - ☐ with arbitrary arc cost but no negative cycle
   - Label correcting algorithms
4. All pairs shortest path problem
   - Floyd-Warshall Algorithm

Assumptions

- The network is directed
- There is a directed path from s to every other node
- All arc lengths are integers
  - Algorithms whose complexity bound depends on C ($C = \max c_{ij}$) rely on this assumption.
  - rational number->integer?
  - irrational number->integer?
- The network does not have a directed cycle of negative cost

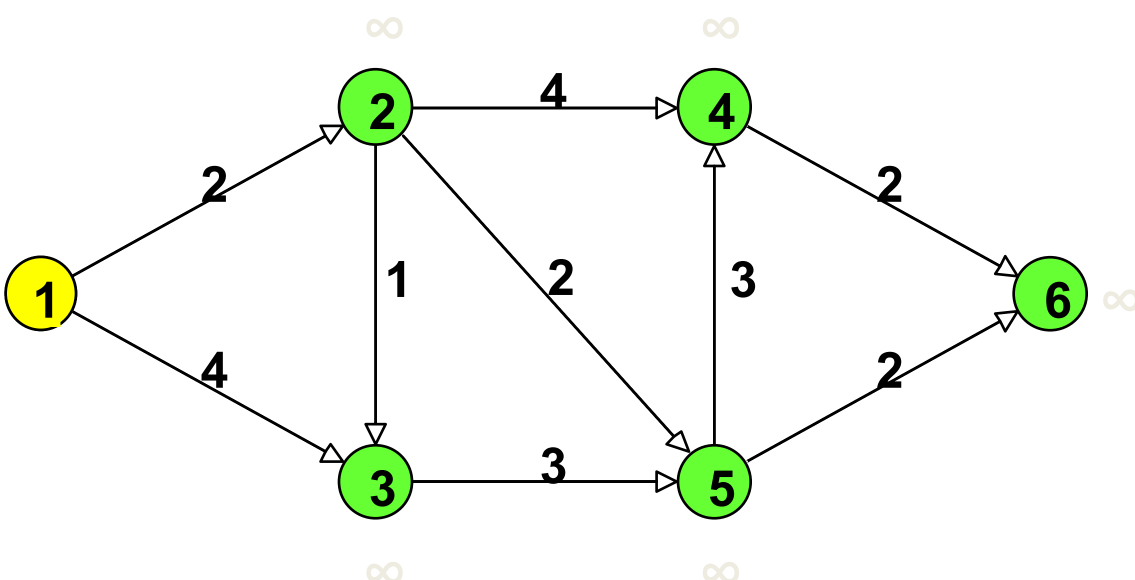
Shortest Path Property 1

- If the path $s=i_1-i_2-\ldots-i_h=k$ is a shortest path from node s to node k, then for every $q=2,3,\ldots,h-1$, the subpaths $s=i_1-i_2-\ldots-i_q$ is a shortest path from node s to node i_q.
  - A subpath of a shortest path is also a shortest path
  - In the figure, if P_1-P_3 is a shortest path to node k, then P_1 must be a shortest path to node i_q.
  - If it is not true, for example, P_2 is a shorter path to node i_q, then P_1-P_3 cannot be a shortest path to node k.



Shortest Path Property 2

- Let the vector ***d*** represent the shortest path distances. Then a directed path P from the source node to node k is a shortest path if and only if $d(j) = d(i) + c_{ij}$ for every arc $(i, j) \in P$.



Outline

1. Shortest path property
2. Shortest path problems in acyclic networks
   – Reaching algorithm
3. Shortest path problems in cyclic networks
   - with non-negative arc costs
   – Label setting algorithm (Dijkstra's algorithm)
   - with arbitrary arc cost but no negative cycle
   – Label correcting algorithms
4. All pairs shortest path problem
   – Floyd-Warshall Algorithm

Shortest path problems in **acyclic networks**

- *For acyclic networks, we can always determine their topological ordering.*
- Once we have determined the topological ordering, the shortest path problem becomes quite easy.

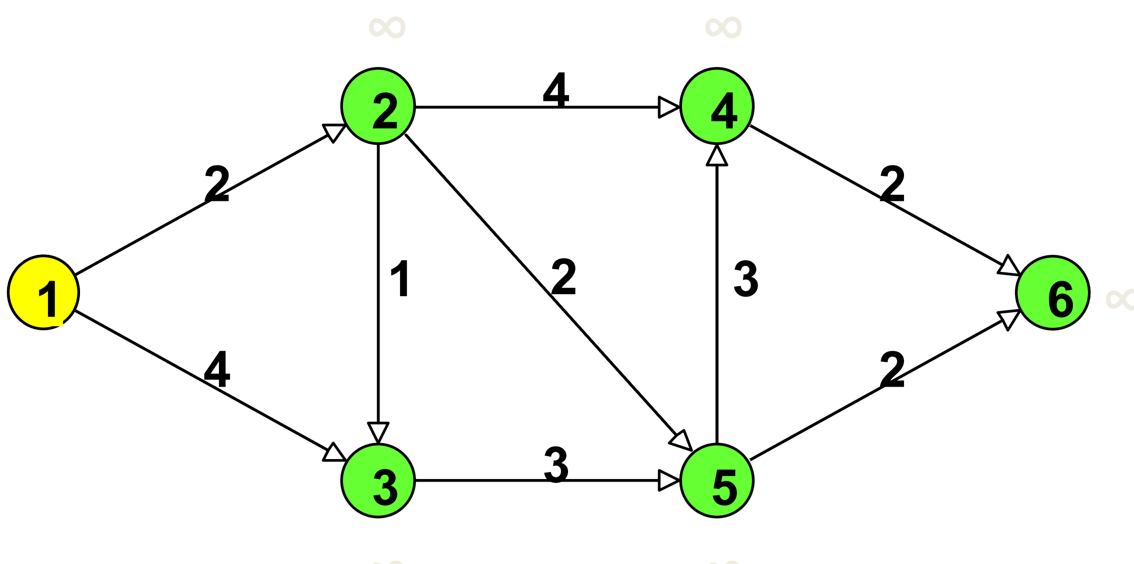
- Suppose we have determined the shortest path distance d(i) from the source node to node i=1,2,…,k. Then we can easily calculate d(k+1)by

$$d(k+1)=\min(\mathbf{d(i)}+c\_ik,\ (i,k)\in A)$$

Shortest path problems in **acyclic networks**

Reaching algorithm: O(m)

Set d(s)=0 and the remaining distance labels to a very large number. Examine nodes in the topological order, and for each node i being examined, we scan arcs in A(i). If for any arc $(i,j) \in A(i)$, we find d(j)>d(i)+c_ij, then we set d(j)=d(i)+c_ij. When the algorithm has examined all the nodes once in this order, the distance labels are optimal.

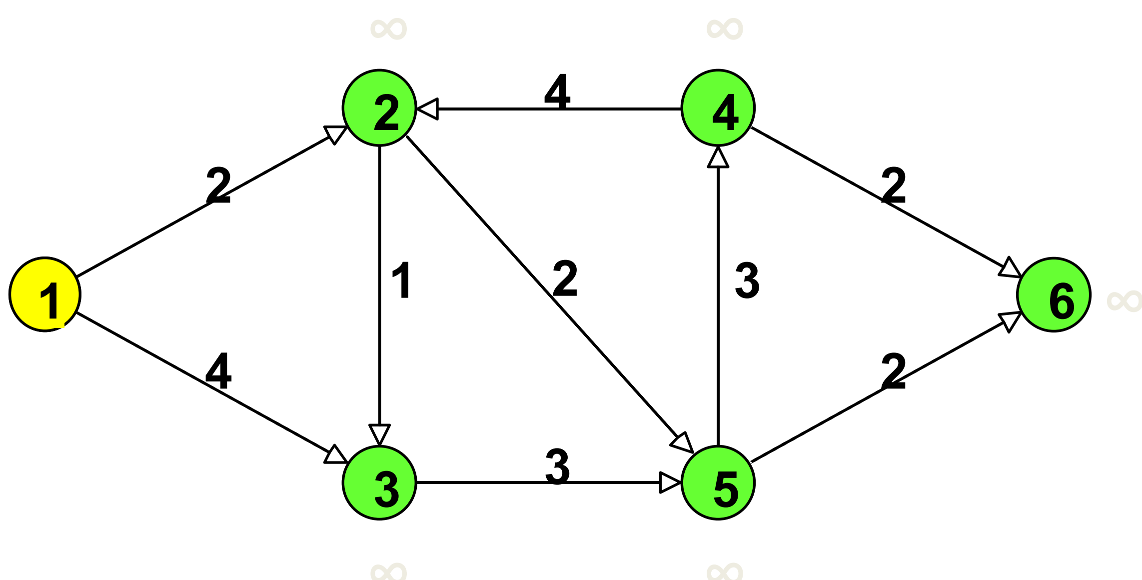


Outline

1. Problem statement
2. Shortest path property
3. Shortest path problems in acyclic networks
   - Reaching algorithm
4. Shortest path problems in cyclic networks
   - with non-negative arc costs
     - Label setting algorithm (Dijkstra's algorithm)
   - with arbitrary arc cost but no negative cycle
     - Label correcting algorithms
5. All pairs shortest path problem
   - Floyd-Warshall Algorithm

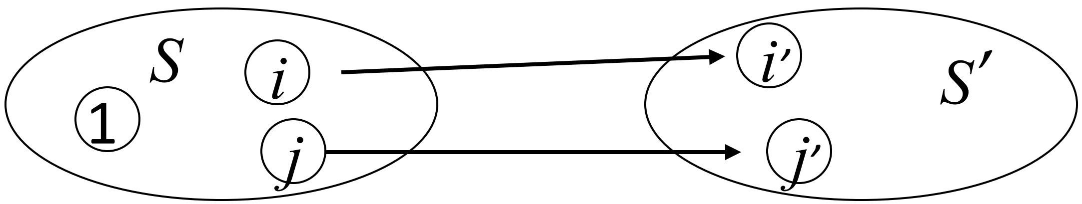
Shortest path problems in cyclic networks

Can we still calculate the shortest distance by d(k+1)=min(**d(i)**+c_ik, (i,k)∈ A)?



Dijkstra’s Algorithm

- Algorithm description
  - Maintain a distance label d(i) with each node i which is an upper bound on the shortest path length;
  - An iterative algorithm, where in each iteration the shortest path to one node is obtained.
- Definition and notation
  - Set *S*: set of nodes permanently labeled
    - Shortest path from source to the node is found
  - Set *S’*: set of nodes temporarily labeled
    - Shortest path from source to the node is yet to be found



Steps of Dijkstra's Algorithm

- Initialization

  Let $S=\emptyset$, $S'=N$, $d(1)=0$, $d(j)=\infty$ for all other nodes j

  - $d(i)$, called distance label, will be used to record an upper bound of the shortest path to node i

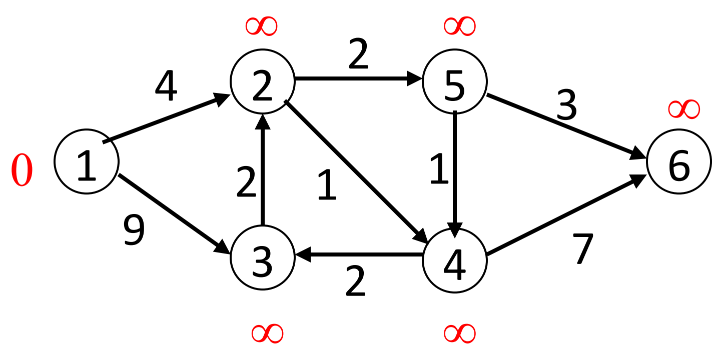
- Repeat until S is the set of all nodes
  - In S', find node i with the smallest distance label

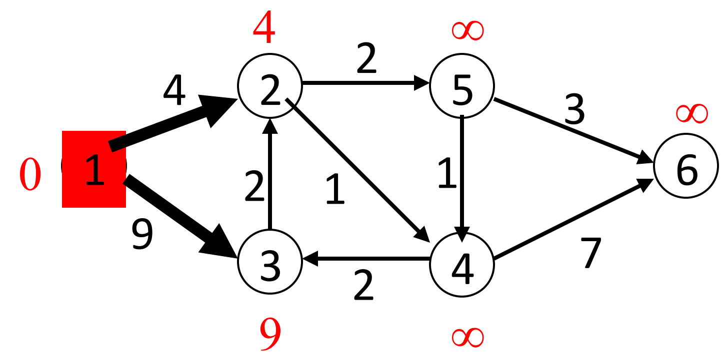
    $d(i)=\min\{\ d(j)\ |\ j \text{ is in } S'\ \}$
  - Let $S = S \cup \{i\}$, and $S' = S' - \{i\}$
  - For each arc (i,j) in $A(i)$
    - If $d(j)>d(i)+c_{ij}$, then $d(j)=d(i)+c_{ij}$ and $pred(j)=i$

Node selection

Distance update

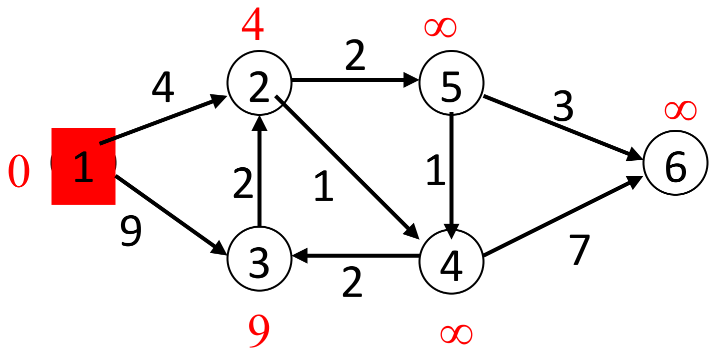
Dijkstra's Algorithm: Examples

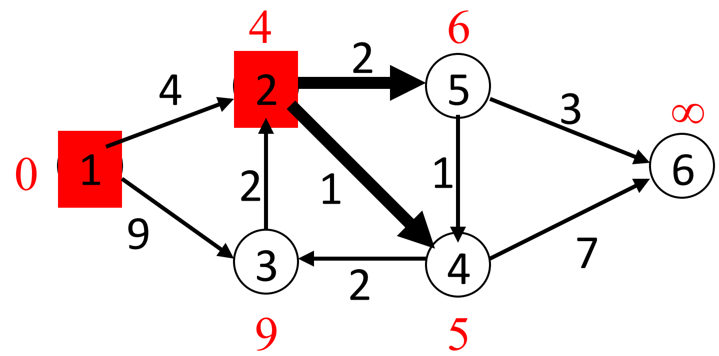




- Iteration 1:
  - Node 1 is selected, S={1}
  - Distance update: $d(2)=4$, $d(3)=9$, pred(2)=1, pred(3)=1

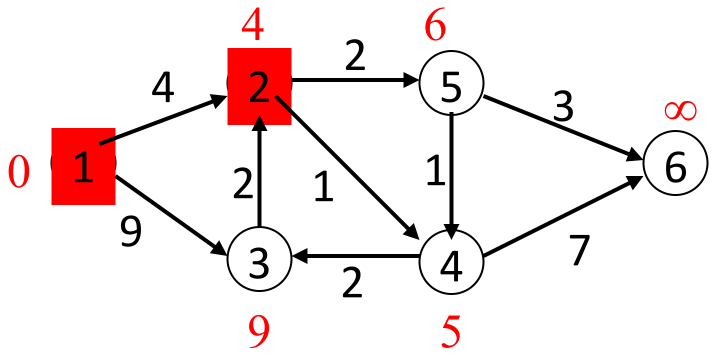
Dijkstra's Algorithm: Examples

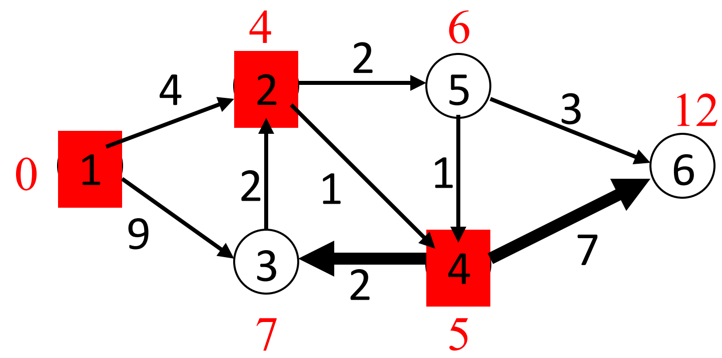




- Iteration 2:
  - Node 2 is selected, S={1,2}
  - Distance update: $d(5)=6$, $d(4)=5$, pred(5)=2, pred(4)=2

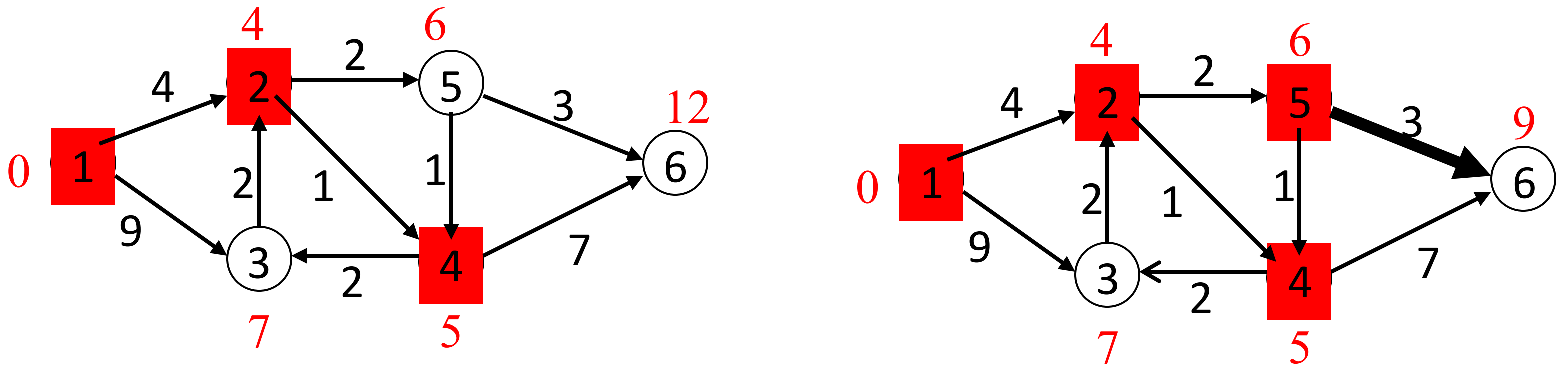
Dijkstra’s Algorithm: Examples





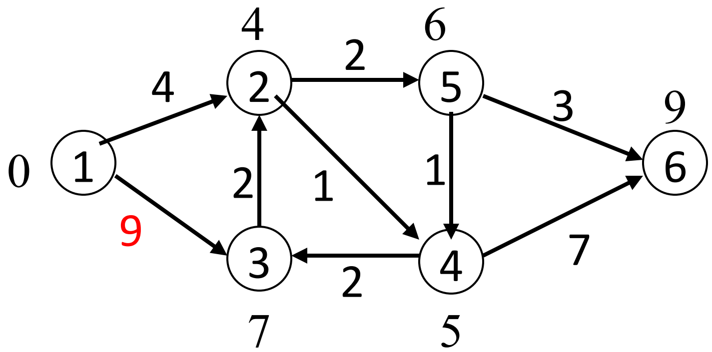
- Iteration 3:
  - Node 4 is selected, S={1,2,4}
  - Distance update: $d(3)=7$, pred(3)=4, $d(6)=12$, pred(6)=4
    - Note that $d(3)$ is updated again

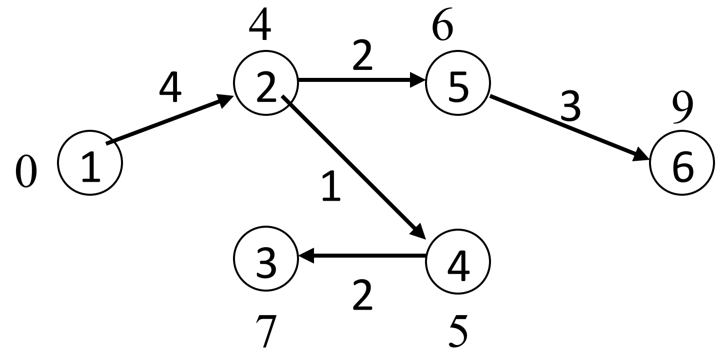
Dijkstra's Algorithm: Examples



- Iteration 4:
  - Node 5 is selected, S={1,2,4,5}
  - Distance update: d(6)=9, pred(6)=5
    - Note that node 6 is updated again

Dijkstra’s Algorithm: Example

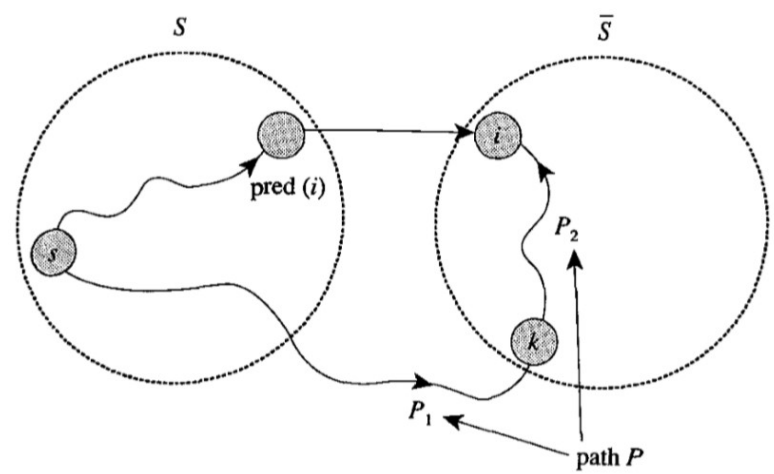




- Dijkstra’s algorithm ends up with a shortest path tree.

Correctness of Dijkstra's algorithm

- When a node *i* is selected, its distance label *d*(*i*) is optimal (i.e., d(i) can't be further reduced).
- When a node *i* is selected d(i), its distance label d(i) is the length of a shortest path to node i among all paths that do not contain any node in $\bar{S}$ as an internal node.
- From the following figure, the length of any path from s to I that contains some nodes in $\bar{S}$ would be at least d(i).
- We therefore can conclude that when a node i is selected, its distance label d(i) is the length of a shortest path from s to i in the network.



Dijkstra's Algorithm: Complexity

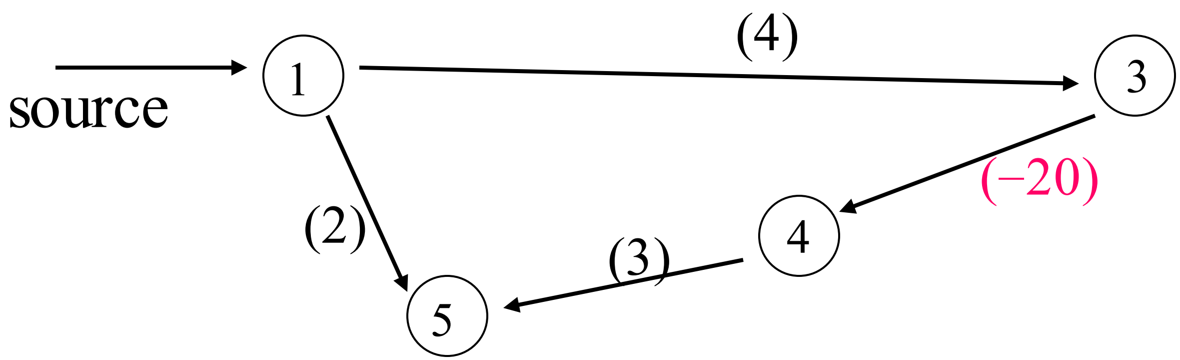
- The algorithm contains n iterations
  - Each iteration determines the shortest path to one specific node
- In each iteration
  - The operation of "node selection" runs in $O(n)$
  - The operation of "distance updates" runs in $|A(i)|$ for the selected node i
- So the overall time complexity is in $O(n^2)$
  - The total number of node selections is in $O(n^2)$
  - The total number of distance updates is in $O(m)$, which is dominated by $O(n^2)$
- The gap between node selection and distance updates motivates people to find algorithms that may be more efficient than Dijkstra's algorithm.
  - Heap implementation $O(m \log_d n)$ where $d=m/n$
  - Fibonacci heap implementation $O(m+n \log n)$

Outline

1. Shortest path property
2. Shortest path problems in acyclic networks
   - Reaching algorithm
3. Shortest path problems in cyclic networks
   - with non-negative arc costs
   - Label setting algorithm (Dijkstra's algorithm)
   - with arbitrary arc cost but no negative cycle
   - Label correcting algorithms
4. All pairs shortest path problem
   - Floyd-Warshall Algorithm

Shortest path problems in cyclic networks with negative costs

- Dijkstra's Algorithm does not work if some arcs have negative costs



In Dijkstra's Algorithm, we first get $d(5)=2$.
However, the real shortest path to node 5 is
$1 \rightarrow 3 \rightarrow 4 \rightarrow 5$, with the cost of -13.

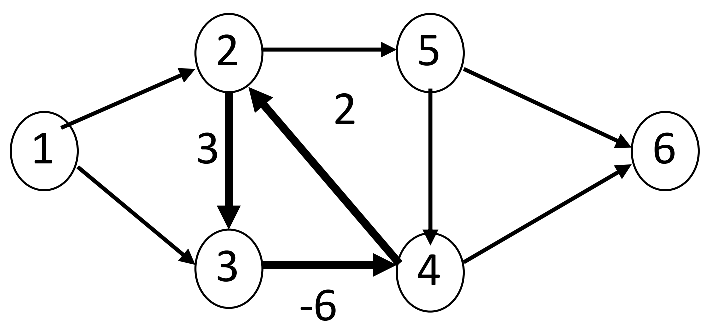
Next

How to find shortest paths from one node to all other nodes for networks with arbitrary arc lengths?

Label-correcting algorithms

Negative Cycles

- When some costs are negative, negative cycles may exist
  - A negative cycle refers to a directed cycle of which the cost is negative
  - The shortest path is not well defined for a network with negative cycles
- There are two problems to solve
  - To detect whether there exist negative cycles
  - To find a shortest path if there are no negative cycles



Optimality Conditions (allowing negative costs)

- Let $\mathbf{d}=(d(1),d(2),\ldots,d(n)\ )$ be a set of distance labels
  - A **necessary** condition for $d(j)$ to be the shortest path length to node j is

    $$d(j)\leq d(i)+c_{ij} \text{ for all arc } (i,j) \text{ in } A$$

    - Reason: If $d(j)>d(i)+c_{ij}$, then $d(j)$ can be reduced to $d(i)+c_{ij}$
  - The condition can be extended to a **sufficient and necessary** condition

Theorem: For any network (N,A), let $d(j)$ denote the length of <u>some directed path</u> from the source node to node j. Then the numbers $d(j)$ represent shortest path distances if and only if they satisfy

$$d(j)\leq d(i)+c_{ij} \qquad \text{for all arc } (i,j) \text{ in } A$$

Optimality Conditions (allowing negative costs)

Theorem: For any network (N,A), let $d(j)$ denote the length of <u>some directed path</u> from the source node to node j. Then the numbers $d(j)$ represent shortest path distances if and only if they satisfy

$$d(j) \le d(i) + c_{ij} \text{ for all arc } (i,j) \text{ in } A \qquad (*)$$

Proof of sufficiency: Consider any solution d(j) satisfying the condition (*). Let $s = i_1 - i_2 - \cdots - i_k = j$ be any directed path P from the source to node j. The condition (*) implies that

$$d(j) = d(i_k) \le d(i_{k-1}) + c_{i_{k-1} i_k},$$

$$d(i_{k-1}) \le d(i_{k-2}) + c_{i_{k-2} i_{k-1}},$$

$$\dots$$

$$d(i_2) \le d(i_1) + c_{i_1 i_2} = c_{i_1 i_2}.$$

Adding these inequalities, we find

$$d(j) = d(i_k) \le c_{i_{k-1} i_k} + c_{i_{k-2} i_{k-1}} \dots + c_{i_1 i_2} = \sum_{ij \in P} c_{ij}$$

So d(j) is a lower bound on the length of any directed path from the source to node j. Since d(j) is the length of some direct path from the source to node j, it is thus the shortest path length.

Reduced arc cost

- Define the reduced arc length c_{ij}^{d} of arc (i, j) with respect to the distance labels $d(.)$ as

$$c_{ij}^{d} = c_{ij} + d(i) - d(j)$$

Property of reduced arc cost:

a) For any directed cycle W, $\sum_{(ij)\in W} c_{ij}^{d} = \sum_{(ij)\in W} c_{ij}$;

b) For any directed path P from node k to node l,
$\sum_{(ij)\in P} c_{ij}^{d} = \sum_{(ij)\in P} c_{ij} + d(k) - d(l)$

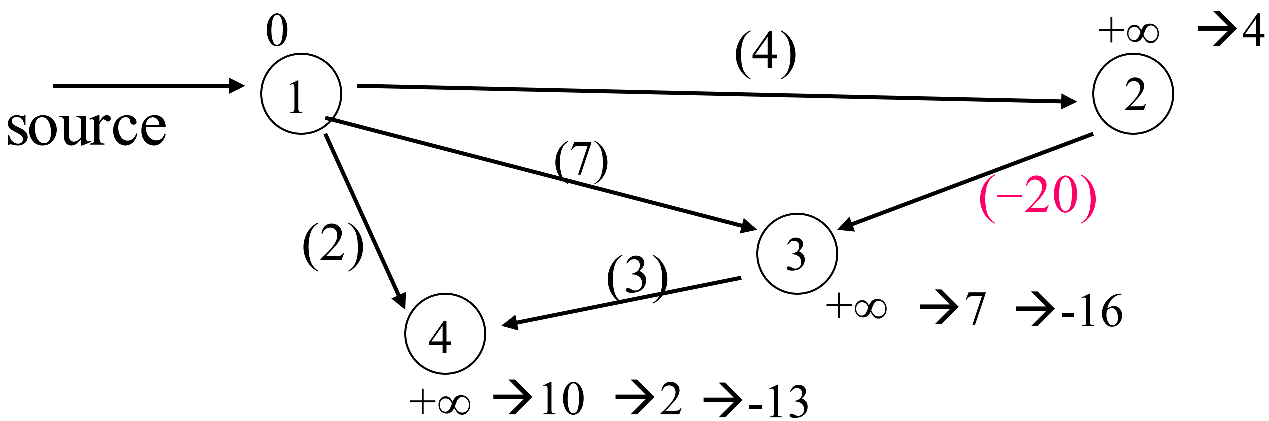
c) If $d(.)$ represent shortest path distances, $c_{ij}^{d} \geq 0$ for every arc $(i, j) \in A$

Generic Label-Correcting Algorithm

- Basic ideas
  - Assume no negative cycles exist
  - Initially set all distance labels to be $+\infty$
    - Except the source node
  - Whenever there is an arc (i,j) such that $d(j) \leq d(i)+c_{ij}$ is violated,
    Update $d(j)$ to be $d(j) = d(i)+c_{ij}$
- Pseudo code for label correcting algorithm

```
Begin
    d(s) = 0 and pred(s) = 0; d(j)= +∞ for other node j in N
    while some arc (i,j) satisfies d(j)>d(i)+c_ij do
    begin
        d(j)=d(i)+c_ij; pred(j)=i;
    end
end
```

Example for Label Correcting



- d(1)=0, d(2)=d(3)=d(4)= +∞
- (1,3) is found, leading to d(3)=7， pred(3)=1
- (3,4) is found, leading to d(4)=10, pred(4)=3
- (1,4) is found, leading to d(4)=2, pred(4)=1
- (1,2) is found, leading to d(2)=4, pred(2)=1
- (2,3) is found, leading to d(3)=-16, pred(3)=2
- (3,4) is found, leading to d(4)=-13, pred(4)=3
- ……

Algorithm Complexity Analysis

- Assuming all costs c_{ij} are integers, then the algorithm will terminate in a finite number of iterations, because
  - Any $d(j)$ can be reduced by at most $2nC$ times

    where ***C=max*{ |*c*$_{ij}$ |, for all (*i,j*) in *A* }**
    - At initialization, we can set $d(j) = nC$
    - In each iteration, one $d(j)$ is reduced by at least 1
    - $d(j)$ cannot be smaller than $-nC$
- Complexity analysis
  - The algorithm needs at most $O(n^2C)$ iterations
  - In each iteration, it takes $O(m)$ time to find a $d(j)$ to update
  - Overall time complexity is in $O(n^2mC)$
    - pseudo-polynomial time

Improvement

- The generic label correcting algorithm is in pseudo-polynomial time
  - The inefficiency come from the fact that it does not specify how to choose an arc (i,j) to update $d(j)$
- Improvement
  - Try to systematically check the arcs

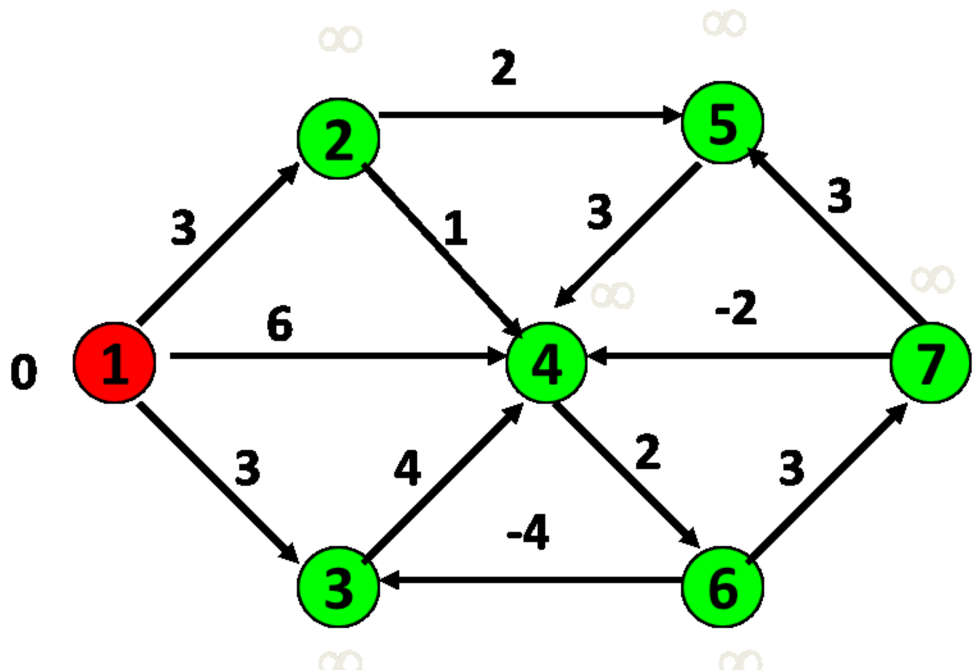
```
Begin
    d(s) = 0 and pred(s) = 0; d(j)= +∞ for other node j in N
    while some arc (i,j) satisfies d(j)>d(i)+c_ij do
    begin
        d(j)=d(i)+c_ij; pred(j)=i;
    end
end
```

Improvement I

$d(s) = 0$ and pred$(s) = 0$; $d(j) = +\infty$ for other node j in N
while some arc (i,j) satisfies $d(j) > d(i) + c_{ij}$ do
begin
    $d(j) = d(i) + c_{ij}$; pred$(j) = i$;
end

- Put all arcs in a list: $a_1, a_2, \ldots, a_m$
  - Check the arcs according to the sequence on the list
    - "Going through all arcs on the list one time" is called a "pass"
  - After one pass is done, we do another pass for all arcs
    - Until all arcs satisfy the optimality condition
- The algorithm needs at most n-1 passes
  - Reason: at the end of the kth pass, the algorithm finds the "shortest paths" for all nodes under the condition that "the shortest paths use k or fewer arcs"
- Each pass we check m arcs
  - Each check takes constant time
- Overall complexity will be $O(nm)$

(1,2)
(1,3)
(1,4)
(2,4)
(2,5)
(3,4)
(4,6)
(5,4)
(6,7)
(7,4)
(7,5)



Improvement II

- Observation: Suppose that during one pass $d(i)$ is not updated. Then in the next pass, we always have $d(j) \leq d(i)+c_{ij}$ for all arcs (i,j) in $A(i)$
  - In the next pass, we only need to check arcs (i,j) of which $d(i)$ is updated in the current pass
  - We do not need to check all arcs in each pass
- Implementation
  - In each pass, we store nodes whose d(i) are updated in a list, and check this list in a first-in-first-out (FIFO) order in the next pass
- Time complexity: $O(nm)$

The FIFO Label-Correcting Algorithm

Begin

  $d(s)=0$, pred(s)=0, $d(j)=\infty$ for all other node j, List={s}

  while (**List** is not empty) do

  begin

    remove the first node i in List

    for each arc (i,j) in $A(i)$ do

      if $d(j)>d(i)+c_{ij}$ then

      begin

       $d(j)=d(i)+c_{ij}$, pred(j) = i

       if j is not in List, then add j to the end of List

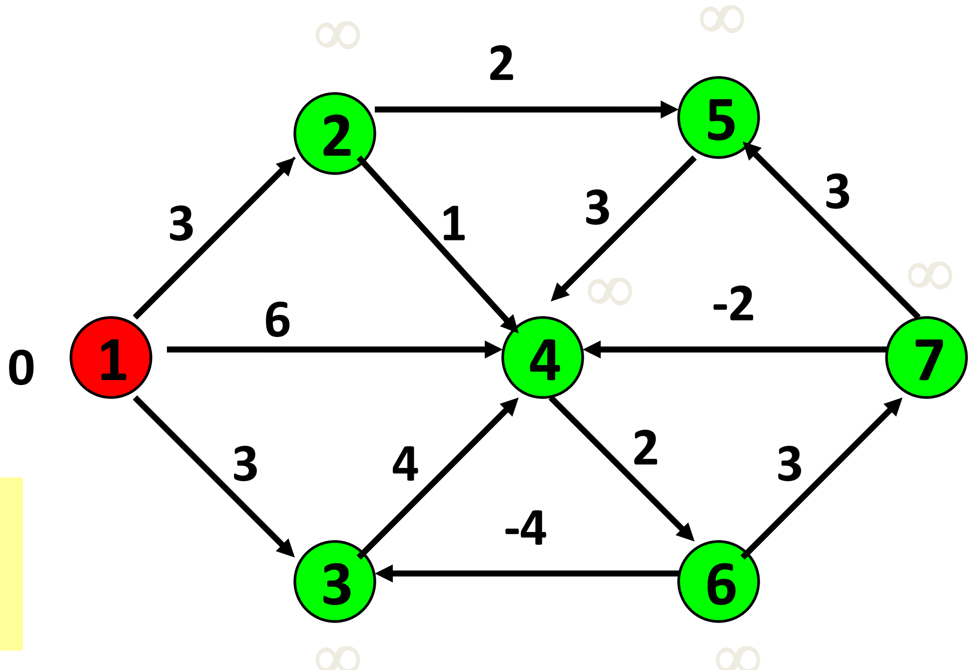
      end

  end

end

We do not need to explicitly maintain the information of "Pass"

FIFO Algorithm: Example



Initialize

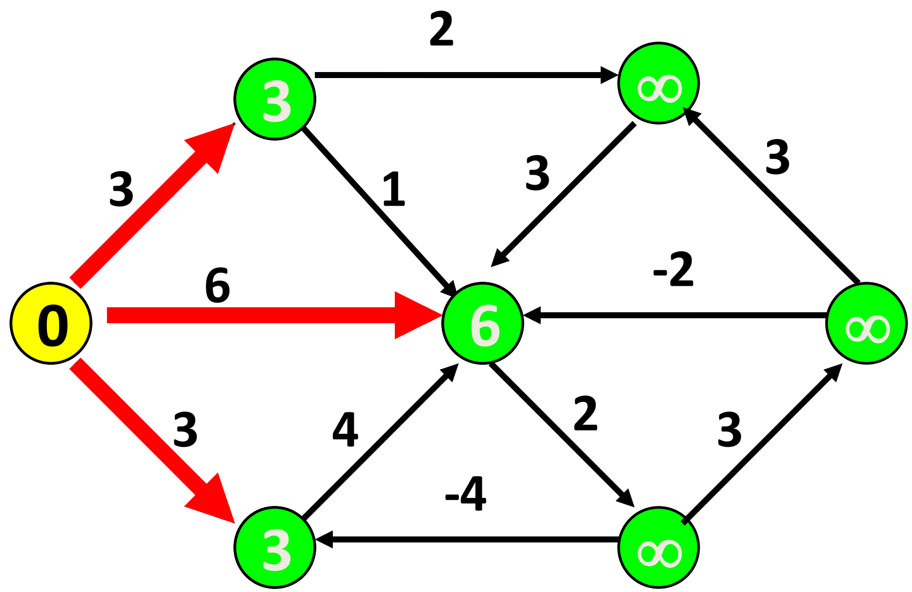
d(1) := 0;
d(j) := ∞ for j ≠ 1

LIST := {1}

In the following slides: the number inside each node *j* is *d*(*j*).

FIFO Algorithm: Example

LIST := { 2, 3, 4 }



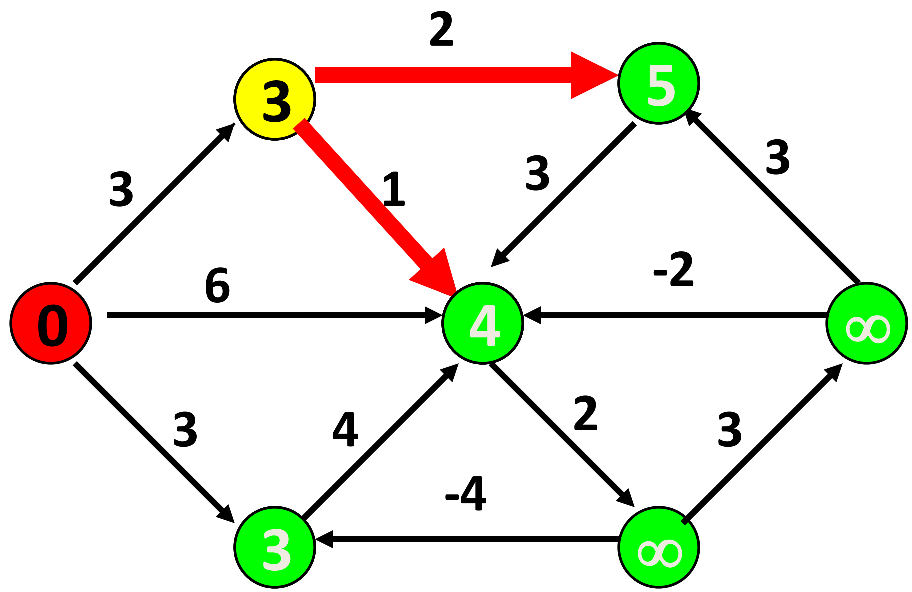
Steps

Take the first node i from LIST: i=1

Update(i): for each arc (i,j) with $d(j) > d(i) + c_{ij}$ replace $d(j)$ by $d(i) + c_{ij}$, and place j at the end of LIST

FIFO Algorithm: Example

LIST := { 3, 4, 5 }

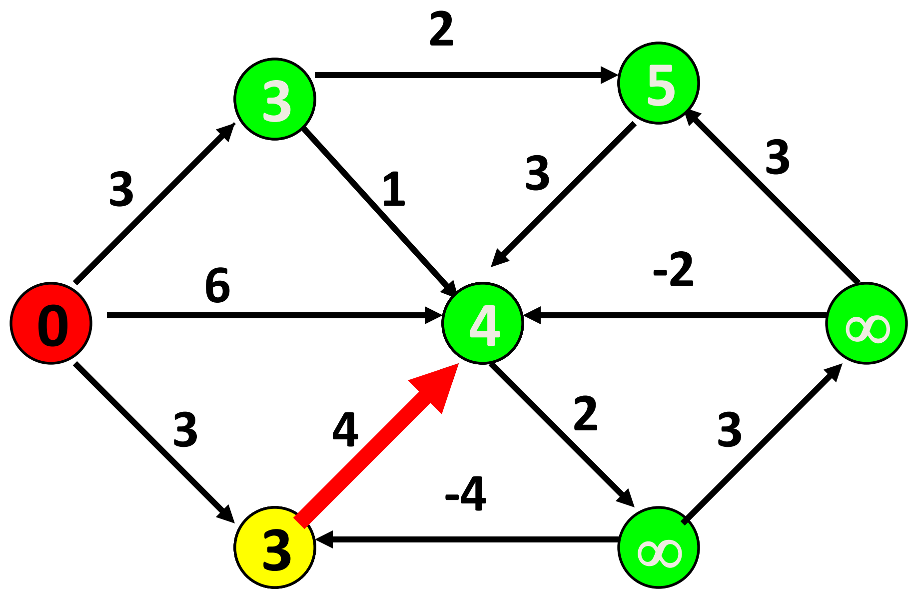


Take the first node i from LIST: i=2

Update(i): for each arc (i,j) with $d(j) > d(i) + c_{ij}$ replace $d(j)$ by $d(i) + c_{ij}$, and place j at the end of LIST

FIFO Algorithm: Example

LIST := { 4, 5 }

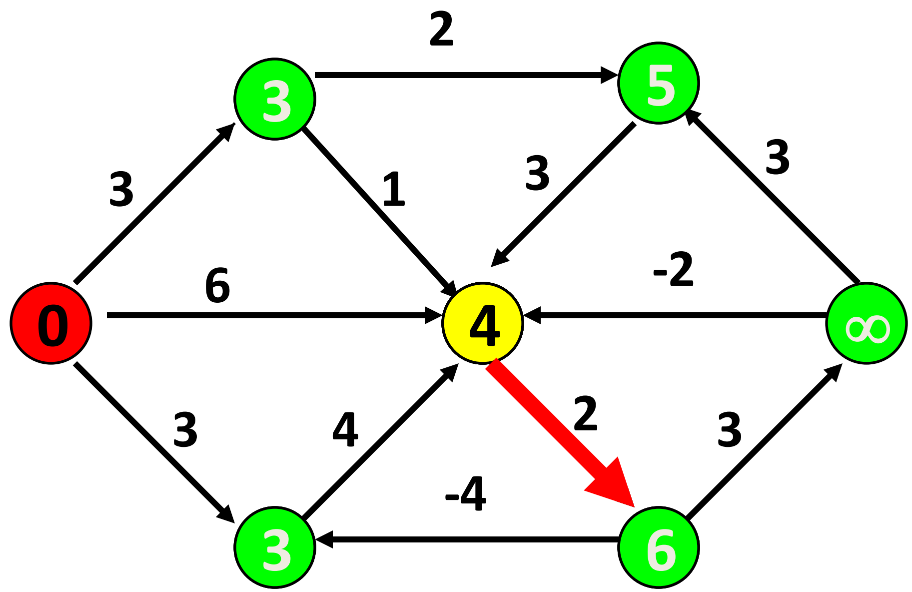


Take the first node i from LIST, i=3

Update(i): for each arc (i,j) with $d(j) > d(i) + c_{ij}$ replace $d(j)$ by $d(i) + c_{ij}$, and place j at the end of LIST

FIFO Algorithm: Example

LIST := { 5, 6 }

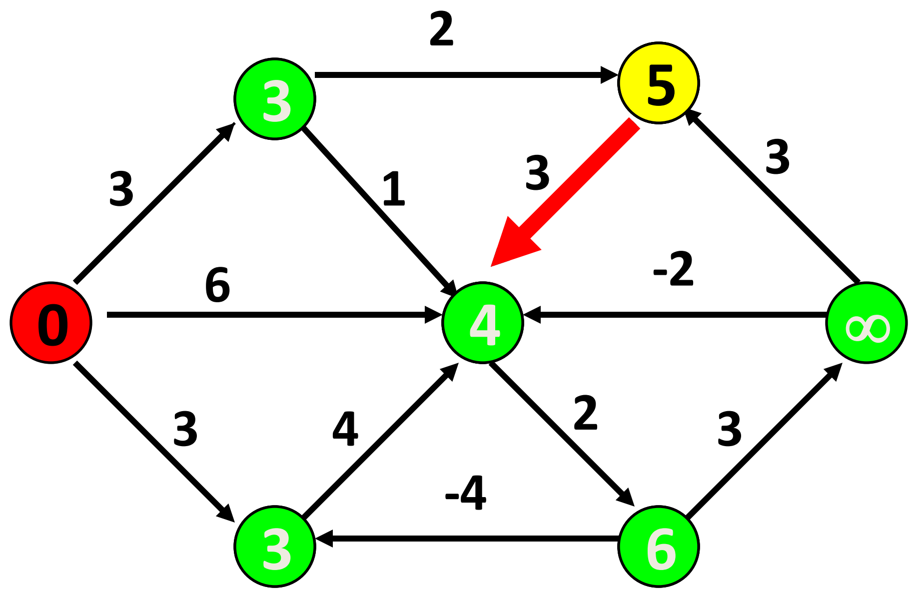


Take the first node i from LIST. i=4

Update(i): for each arc (i,j) with $d(j) > d(i) + c_{ij}$ replace $d(j)$ by $d(i) + c_{ij}$, and place j at the end of LIST

FIFO Algorithm: Example

LIST := { 6 }

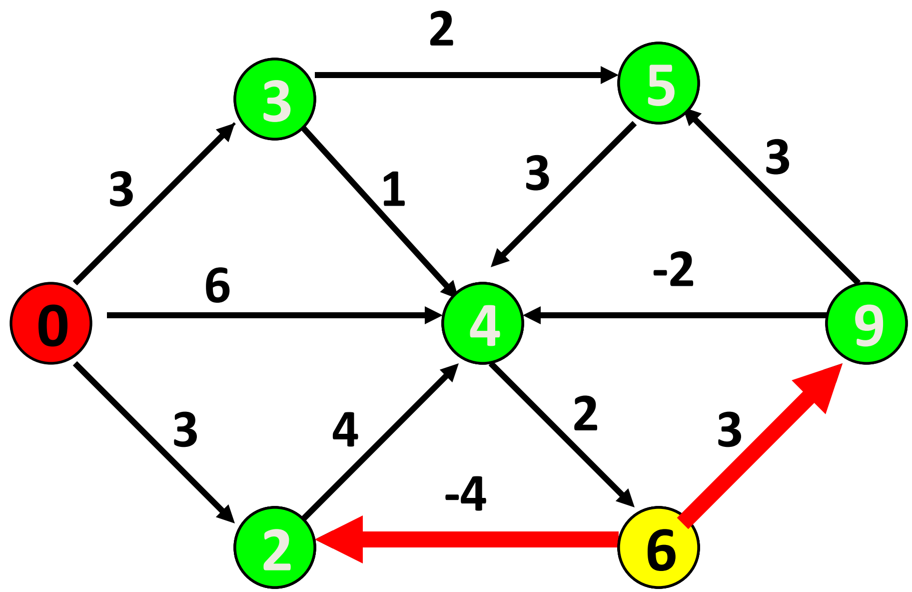


Take the first node i from LIST. i=5

Update(i): for each arc (i,j) with $d(j) > d(i) + c_{ij}$ replace $d(j)$ by $d(i) + c_{ij}$, and place j at the end of LIST

FIFO Algorithm: Example

LIST := { 3, 7 }

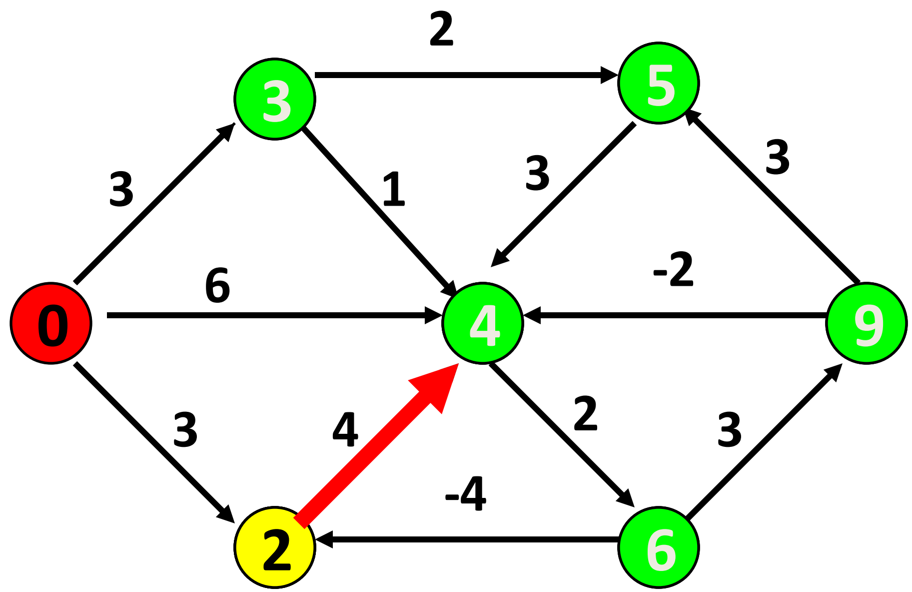


Take the first node i from LIST. i=6

Update(i): for each arc (i,j) with $d(j) > d(i) + c_{ij}$ replace $d(j)$ by $d(i) + c_{ij}$, and place j at the end of LIST

FIFO Algorithm: Example

LIST := { 7 }

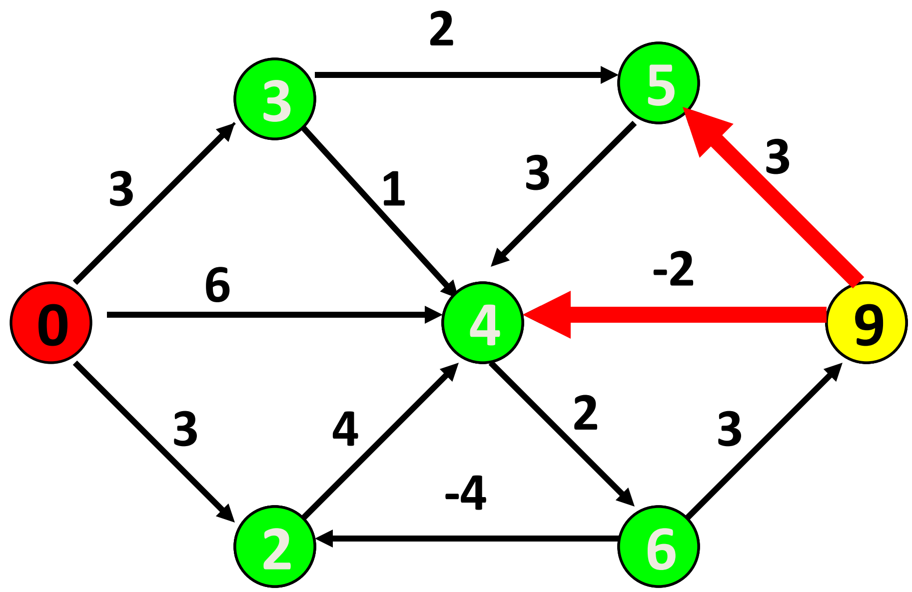


Take the first node i from LIST. i=3

Update(i): for each arc (i,j) with $d(j) > d(i) + c_{ij}$ replace $d(j)$ by $d(i) + c_{ij}$, and place j at the end of LIST

FIFO Algorithm: Example

LIST := { }

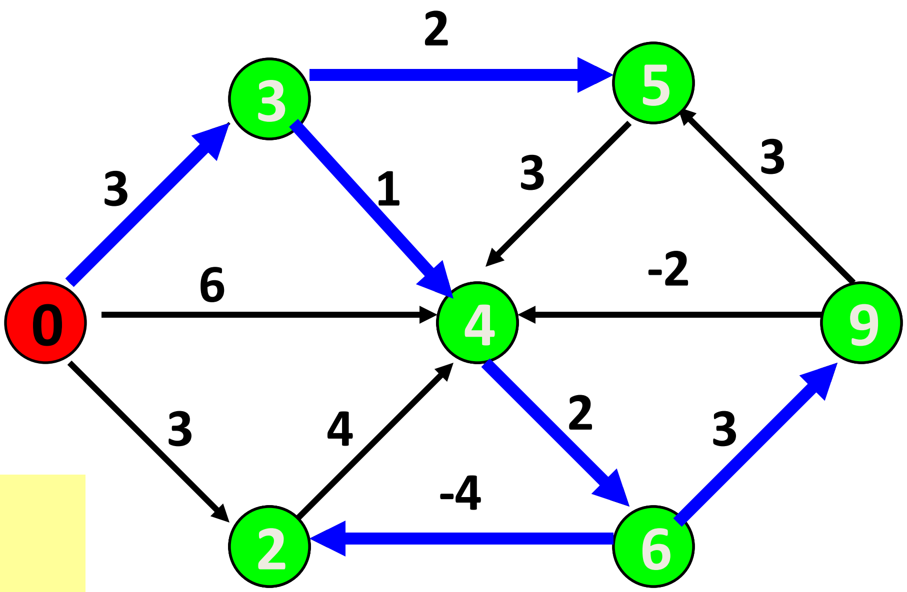


Take the first node i from LIST. i=7

Update(i): for each arc (i,j) with $d(j) > d(i) + c_{ij}$ replace $d(j)$ by $d(i) + c_{ij}$, and place j at the end of LIST

FIFO Algorithm: Example

LIST := { }



LIST is empty.

The distance labels are optimal

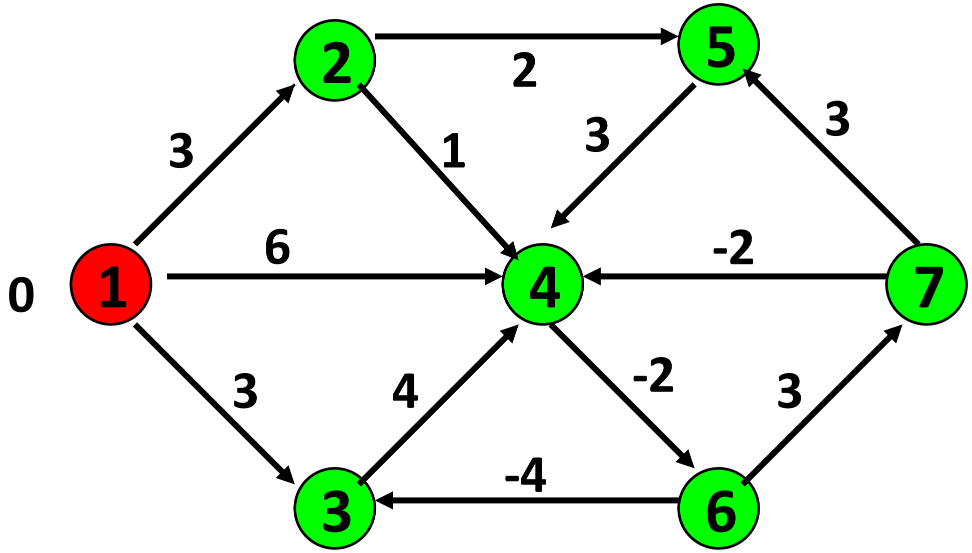
Here is the shortest path tree indicated by the pred(j)

Detecting Negative Cycles

- In the generic label correcting algorithm
  - If we find a distance label $d(j)$ becomes smaller than $-nC$, then a negative cycle is detected
- In FIFO algorithm
  - For each node, we record the number of times it is put in the **list**
  - If there is one node which is put more than n times in the **list**, then a negative cycle is detected
- The time complexity does not increased for the two label correcting algorithms when being used to detect negative cycles
  - The algorithm either reports a negative cycle and stops, or finds a shortest path tree and stops

Detecting Negative Cycles

- Try the label correcting algorithm to examine if there are negative cycles



Outline

1. Shortest path property
2. Shortest path problems in acyclic networks
   – Reaching algorithm
3. Shortest path problems in cyclic networks
   - with non-negative arc costs
   – Label setting algorithm (Dijkstra’s algorithm)
   - with arbitrary arc cost but no negative cycle
   – Label correcting algorithms
4. All pairs shortest path problem
   – Floyd-Warshall Algorithm

All-pair Shortest Path Problem

- To find a shortest path for each pair of the nodes
- Repeated shortest path algorithm
  - Starting from each node, run a single-source shortest path algorithm to find the shortest path tree to all other nodes
- Time complexity
  - $O(n\,S(n,m,C))$ when there is no negative costs
    - We use $S(n,m,C)$ to denote the time for different implementations of shortest path algorithm
  - $O(n^2m)$ when there are negative costs
    - This can be improved

Optimality Condition

- Define $d[i,j]$ to be the distance label from node i to node j
  - A generalization of the single-source distance label $d(j)$
- The optimality condition for $d[i,j]$ to be the shortest path labels is that

$$d[i,j] \leq d[i,k] + d[k,j] \text{ for all nodes } i,\, j \text{ and } k$$

Generic Label Correcting Algorithm

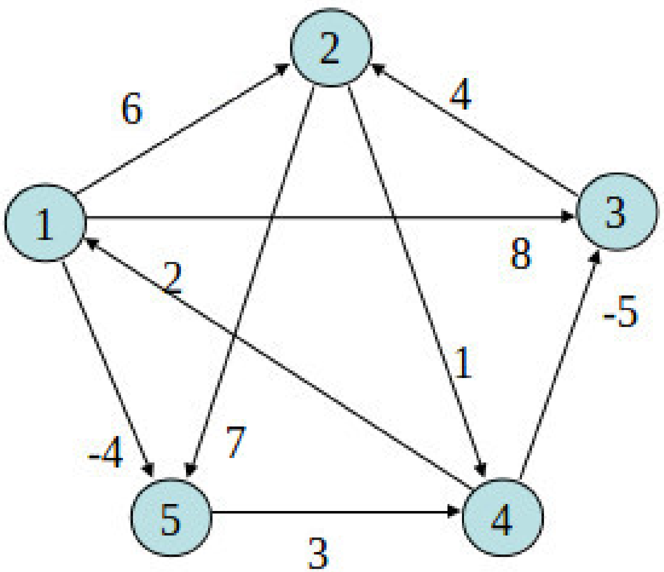
- Whenever these are three nodes *i,j* and *k* such that $d[i,j] \le d[i,k] + d[k,j]$ is violated,

    Update $d[i,j]$ to be $d[i,j] = d[i,k] + d[k,j]$

- Pseudo code for generic label correcting algorithm

```
Begin
    d[i,j] = ∞ for all i and j
    d[i,i] = 0 for all i
    d[i,j] = c_ij for all (i,j) in A
    while there are nodes i,j and k satisfying d[i,j] > d[i,k] + d[k,j] do
    begin
        d[i,j] = d[i,k] + d[k,j]
    end
end
```

example



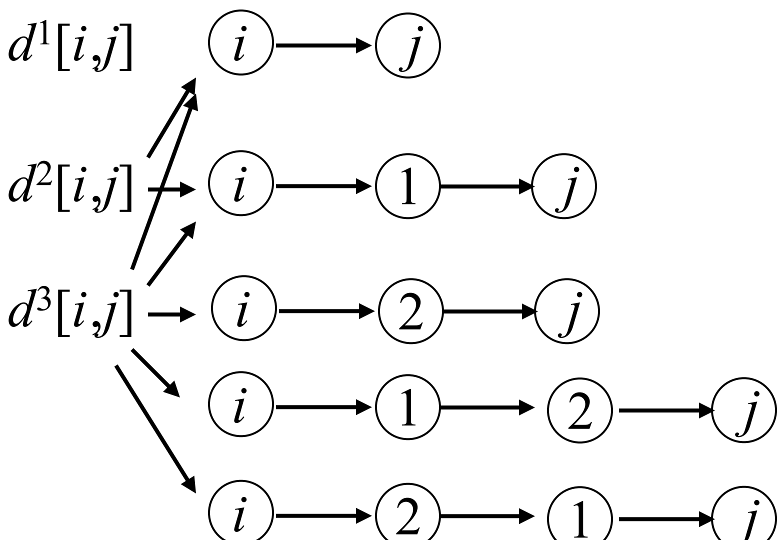
	1	2	3	4	5
1	0	6	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	3	0

Complexity Concerns

- The generic label correcting algorithm is in pesudo-polynomial time
  - Each label can be reduced by up to $O(nC)$ times
  - There are $O(n^2)$ labels
  - Total iterations needed are in $O(n^3C)$
  - In each iterations, it takes $O(n^3)$ time to test the optimality condition
- Floyd-Warshall algorithm can implement the algorithm in $O(n^3)$ time
  - Surprising result because it takes $O(n^3)$ time to test the optimality condition
  - Using the ideas of Dynamic Programming

Floyd-Warshall Algorithm

- Let $d^k[i,j]$ be the shortest path cost from node i to node j where only nodes in $\{1,2,\ldots,k\text{-}1\}$ may be used as internal nodes on the path
  - We will first calculate $d^1[i,j]$, then $d^2[i,j]$, … , until $d^{n+1}[i,j]$
  - The shortest path from i to j is exactly $d^{n+1}[i,j]$



The General Case

- By definition, $d^{k+1}[i,j]$ may or may not contain node k
  - Case 1: If not containing node k, then $d^{k+1}[i,j] = d^k[i,j]$
  - Case 2: If containing node k, then $d^{k+1}[i,j] = d^k[i,k] + d^k[k,j]$
  - Thus $d^{k+1}[i,j]$ should be the smaller of the two cases

Case 1: $i \xrightarrow{d^k[i,j]} j$

Case 2: $i \xrightarrow{d^k[i,k]} k \xrightarrow{d^k[k,j]} j$

Dynamic Program

- Dynamic programming (DP) recursion

$$d^{k+1}[i,j] = \min\{d^k[i,j],\ d^k[i,k] + d^k[k,j]\ \}$$

Case 1: Node k is not used

Case 2: Node k is used

$d^{k+1}[i,j]$ can be calculated if all values of $d^k[i,j]$ are known

Initial conditions for k=1

By definition, $d^1[i,i] = 0$

By definition, $d^1[i,j] = c_{ij}$ if arc (i,j) exists

By definition, $d^1[i,j] = +\infty$ if arc (i,j) does not exist

Floyd-Warshall Algorithm: Pseudo Code

Begin

for all node pairs $[i,j]$ do $d^1[i,j] = \infty$ and pred$[i,j] = 0$;

for all nodes i, do $d^1[i,i] = 0$;

for each arc (i,j) in A, do $d^1[i,j] = c_{ij}$ and pred$[i,j] = i$;

for k=1 to n do

for i=1 to n do

for j=1 to n do

if $d^k[i,j] > d^k[i,k] + d^k[k,j]$ then

$d^{k+1}[i,j] = d^k[i,k] + d^k[k,j]$ and pred$[i,j]$ = pred$[k,j]$;

else

$d^{k+1}[i,j] = d^k[i,j]$;

End

Floyd-Warshall Algorithm: Analysis

- Time complexity
  - The algorithm has three nested loops, each in the order of $O(n)$
  - The computation for the most internal loop is constant
  - So the time complexity is in $O(n^3)$
    - This achieves the efficiency of the repeated shortest path algorithm when the basic Dijkstra's algorithm is used

Detecting Negative Cycles

- In the Floyd-Marshall algorithm, a negative cycle will be detected when we find
  - (1) $d[i,i]<0$, or
  - (2) $d[i,j]< -nC$
- Reason:
  - If $d[i,i]<0$, we must have a k such that $d[i,k]+d[k,i]<0$, which implies a negative cycle $i \rightarrow \ldots \rightarrow k \rightarrow \ldots \rightarrow i$
  - If $d[i,j]< -nC$, from the single-source shortest path problem, we know a negative cycle is found