Objectives

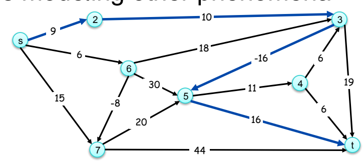
- Dynamic Programming
  - Shortest Path

Shortest Paths

- **Problem**: Given a directed graph G = (V, E), with edge weights c_{vw}, find shortest path from node s to node t

allow negative weights

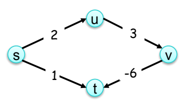
- Allows modeling other phenomena



Shortest Paths: Failed Attempts

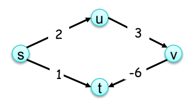
- Dijkstra. Can fail if negative edge costs

Shortest path from s →t?



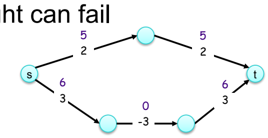
Shortest Paths: Failed Attempts

- Dijkstra. Can fail if negative edge costs

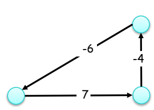


- Re-weighting. Adding a constant to every edge weight can fail

Why?



Shortest Paths: Negative Cost Cycles



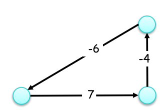
- If some path from s to t contains a negative cost cycle, there does **not** exist a shortest s-t path

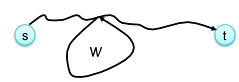
Why?

- Otherwise, there exists one that is *simple* (i.e., does not repeat nodes)

What does this mean about number of edges in path?

Shortest Paths: Negative Cost Cycles



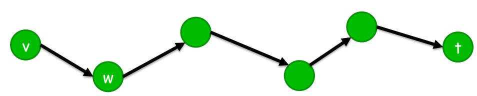


- If some path from s to t contains a negative cost cycle, there does **not** exist a shortest s-t path
- Otherwise, there exists one that is *simple* (i.e., does not repeat nodes)
  - Path has *at most n-1* edges, where n is # of nodes in graph

Towards a Recurrence

- **OPT(*i*,*v*)**: minimum cost of a *v*-*t* path *P* using *at most i* edges
  - This formulation eases later discussion
- Original problem is OPT(n-1, s)

Break down into subproblems based on *i* and *v*



Path P

Shortest Paths: Dynamic Programming

- Def. OPT(i, v) = minimum cost of a *v*-*t* path *P* using at most *i* edges
  - Case 1: *P* uses at most *i-1* edges
    - OPT(i, v) = OPT(i-1, v)
  - Case 2: *P* uses exactly *i* edges
    - if (*v*, *w*) is first edge, then OPT uses (*v*, *w*), and then selects best *w*-*t* path using at most *i-1* edges
    - Cost: cost of chosen edge

$$OPT(i,v) = \begin{cases} 0 & \text{if } i=0 \\ \min\left\{ OPT(i-1,\,v),\ \min_{(v,w)\in E} \{ OPT(i-1,\,w)+c_{vw} \} \right\} & \text{otherwise} \end{cases}$$

Shortest Paths: Implementation

```
Shortest-Path(G, t)
  n = number of nodes in G
  foreach node v ∈ V
     M[0, v] = ∞    # infinite cost to reach all nodes
  M[0, t] = 0    # no cost to reach destination from dest

  for i = 1 to n-1
     foreach node v ∈ V
        M[i, v] = M[i-1, v]  # at most cost of 1 less
        foreach edge (v, w) ∈ E
           M[i, v] = min(M[i, v], M[i-1, w] + c_vw )
```

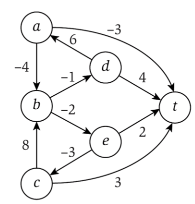
Analysis?

Cost of chosen edge

- Shortest path is M[n-1, s]

Starting node

Example

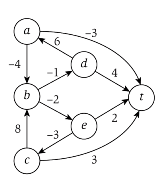


Number of edges in path

	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞					
b	∞					
c	∞					
d	∞					
e	∞					

What edges do we need to look at for each node?

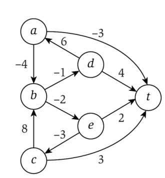
Example



Edges
b , t
d, e
b, t
a, t
c, t

	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞					
b	∞					
c	∞					
d	∞					
e	∞					

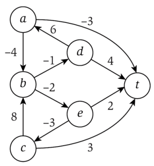
Example



Edges
b , t
d, e
b, t
a, t
c, t

	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3				
b	∞	∞				
c	∞	3				
d	∞	4				
e	∞	2				

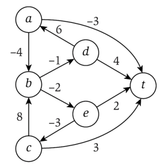
Example



Edges
b , t
d, e
b, t
a, t
c, t

	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3			
b	∞	∞	0			
c	∞	3	3			
d	∞	4	3			
e	∞	2	0			

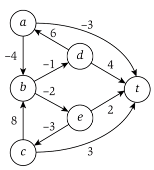
Example



Edges
b , t
d, e
b, t
a, t
c, t

	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4		
b	∞	∞	0	-2		
c	∞	3	3	3		
d	∞	4	3	2		
e	∞	2	0	0		

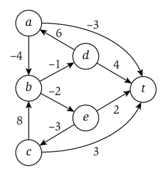
Example



Edges
b , t
d, e
b, t
a, t
c, t

	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	
b	∞	∞	0	-2	-2	
c	∞	3	3	3	3	
d	∞	4	3	2	0	
e	∞	2	0	0	0	

Example



Edges
b , t
d, e
b, t
a, t
c, t

	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	-6
b	∞	∞	0	-2	-2	-2
c	∞	3	3	3	3	3
d	∞	4	3	2	0	0
e	∞	2	0	0	0	0

Shortest Paths: Implementation

```
Shortest-Path(G, t)
   n = number of nodes in G
   foreach node v ∈ V
      M[0, v] = ∞    # infinite cost to reach all nodes
   M[0, t] = 0    # no cost to reach destination from dest

   for i = 1 to n-1
      foreach node v ∈ V
         M[i, v] = M[i-1, v]  # at most cost of 1 less
         foreach edge (v, w) ∈ E
            M[i, v] = min(M[i, v], M[i-1, w] + c_vw )
```

$O(n^3)$

- Shortest path is M[n-1, s]

Based on Example Experience

- What could we do to improve the algorithm’s runtime/space requirements?

Shortest Paths: Practical Improvements

- Practical improvements
  - Maintain only one array M[v] = shortest *v*-t path that we have found so far
  - No need to check edges of the form (*v*, *w*) ***unless*** M[w] changed in previous iteration
- Theorem. Throughout algorithm, M[v] is length of some *v*-*t* path, and after *i* rounds of updates, the value M[v] is no larger than the length of shortest *v*-*t* path using ≤ *i* edges.
- Overall impact
  - Memory: O(m + n)
  - Running time: O(mn) worst case but substantially faster in practice

Bellman-Ford: Efficient Implementation

```
Push-Based-Shortest-Path(G, s, t)
   foreach node v ∈ V
      M[v] = ∞
      successor[v] = ϕ

   M[t] = 0
   for i = 1 to n-1
      foreach node w ∈ V
         if M[w] has been updated in previous iteration
            foreach node v such that (v, w) ∈ E
               if M[v] > M[w] + c_vw
                  M[v] = M[w] + c_vw
                  successor[v] = w

      If no M[w] value changed in iteration i, stop.
```


DISTANCE VECTOR PROTOCOL

Problem Context

- Application of shortest-path problem: *routers in communication network find most efficient path to destination*
- Model of communication network
  - Nodes ≈ routers
  - Edge ≈ direct communication link
  - Cost of edge ≈ delay on link ← *Naturally nonnegative*
- Possible solution: Dijkstra's algorithm

Distance Vector Protocol

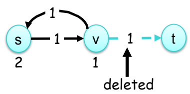
- Model of communication network
  - Nodes ≈ routers
  - Edge ≈ direct communication link
  - Cost of edge ≈ delay on link ← *Naturally nonnegative but Bellman-Ford used anyway!*
- Dijkstra's algorithm. Requires *global* information of network
- Bellman-Ford. Uses only *local* knowledge of neighboring nodes
  - **Distribute** algorithm: each node *v* maintains its value M[v]
    - Updates its value after getting neighbor's values:
      - $\min_{w \in V} (c_{vw} + M[w])$

Distance Vector Protocol

- Each router maintains a vector of shortest path lengths to every other node (distances) and the first hop on each path (directions)
- Algorithm: each router performs *n* separate computations, one for each potential destination node
- Synchronization. We don't expect routers to run in lockstep. The order in which each `foreach` loop executes in not important. Moreover, algorithm still converges even if updates are asynchronous.
- "Routing by rumor."
- Used in many routers, e.g. RIP, Xerox XNS RIP, Novell's IPX RIP, …

Issues with Distance Vector Protocol

- Original algorithm developed for one central machine; costs known in advance, didn't change
- Edge costs may change during algorithm (or fail completely)



"counting to infinity"

Path Vector Protocols

- Link state routing
  - Each router stores the *entire path*
    - Not just the distance and the first hop
  - Based on Dijkstra's algorithm
  - Avoids "counting-to-infinity" problem and related difficulties
  - Requires significantly more storage
- Ex. Border Gateway Protocol (BGP), Open Shortest Path First (OSPF)

This Week

- Keep reading Chapter 6
- Exam 2 due Friday
  - Wednesday: work day
  - No "outside resources"
  - OK: Your notes, my slides, book