Shortest Paths

- Distance is the length of the shortest path between two nodes
- Breadth-first search (BFS) creates one BFS tree rooted at the initial node
  - Just one tree since we only care about nodes connected to the initial node. The rest have infinite distance.
  - BFS tree is a shortest path tree
  - Distance is the tree depth where each node resides
- Complexity? – Examine algorithm

Figure 4.3 Breadth-first search.

```
procedure bfs(G, s)
Input:    Graph G = (V, E), directed or undirected; vertex s ∈ V
Output:   For all vertices u reachable from s, dist(u) is set
          to the distance from s to u.

for all u ∈ V:
    dist(u) = ∞

dist(s) = 0
Q = [s]  (queue containing just s)
while Q is not empty:
    u = eject(Q)
    for all edges (u, v) ∈ E:
        if dist(v) = ∞:
            inject(Q, v)
            dist(v) = dist(u) + 1
```

Careful look at BFS Complexity

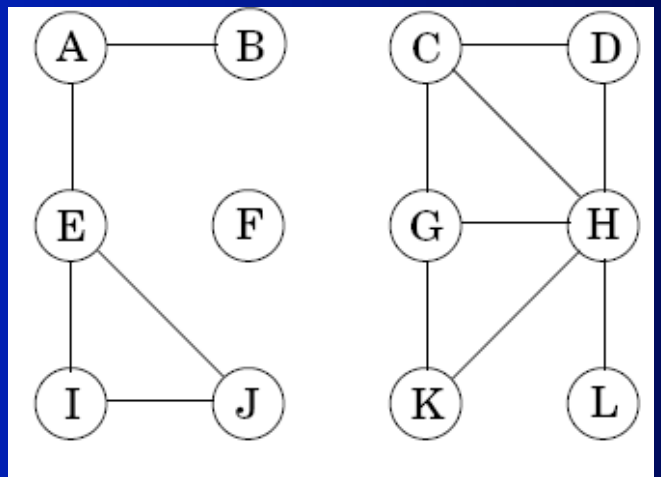
- Initialization step is O(|*V*|)
- Complexity of main section?
  - Each *v* considered exactly once (one eject and one inject)
    - Must consider what the complexity is of eject and inject
  - For each *v*, each edge is considered exactly once for directed graphs and twice for undirected graphs
  - O(|*V*|*complexity(inject) + |*V*|*complexity(eject) + (1 or 2)*|*E*|) = ?

Careful look at BFS Complexity

- Complexity of main section?
  - Each *v* considered exactly once (one eject and one inject)
    - Must consider what the complexity is of eject and inject
  - For each *v*, each edge is considered exactly once for directed graphs and twice for undirected graphs
  - O(|*V*|*complexity(inject) + |*V*|*complexity(eject) + (1 or 2)*|*E*|) = ?
  - For a simple queue the complexity of inject and eject are O(1)
  - Note that since we start at just one node and consider all reachable nodes V_r, the number of unique reachable edges E_r lies between V_r-1 and $2V_r^2$ and thus $E_r = \Omega(V_r)$ (i.e. in complexity sense $E_r \geq V_r$)
  - O(|E_r|) = O(|*E*|)
  - Thus total main section complexity is O(|*E*|)
  - Total overall is O(|*V*|+|*E*|)

Figure 4.3 Breadth-first search.

```
procedure bfs(G, s)
Input:    Graph G = (V, E), directed or undirected; vertex s ∈ V
Output:   For all vertices u reachable from s, dist(u) is set
          to the distance from s to u.

for all u ∈ V:
    dist(u) = ∞

dist(s) = 0
Q = [s]  (queue containing just s)
while Q is not empty:
    u = eject(Q)
    for all edges (u, v) ∈ E:
        if dist(v) = ∞:
            inject(Q, v)
            dist(v) = dist(u) + 1
```

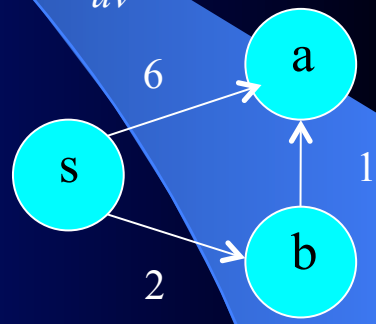


DFS and BFS Comparisons

- DFS (e.g. Explore(G,v)) goes as deep as it can before backtracking for other options
  - Typically will not find shortest path
  - Could find a goal deep in the search tree relatively fast
  - Only need memory equal to stack depth
  - Infinitely deep paths can get you stuck
- BFS
  - Guaranteed to find shortest path
    - Is that always important?
  - Memory will be as wide as the search tree (grows exponentially)
  - Long search if goal is deep in the search tree
- "Best" First Search - Later

Lengths on Edges

- Often edges will correspond to different lengths
  - Map
  - Path cost, time, distance, difficulty, negative values, etc.
- Each $e \in E$ has l_e, also written for u,v as $l(u,v)$ or l_{uv}
- How do we find shortest path in this case
  - Assume positive edge lengths for the moment
- Dijkstra's algorithm
  - Replace BFS queue with a Priority Queue
  - Priority queue keeps the smallest current distance at front
  - Update appropriate distances after a new node is opened
    - Arriving at a visited node may actually shorten the path
  - You will use Dijkstra's in your "Network Routing" project



Priority Queue Operations

- Maintains a set of elements each with an associated numeric key (e.g. distance)
- We will review complexity details in a moment
- Following are four operations which we will consider

Insert	Add a new element to the set (also: “Enqueue”)
Decrease-key	Accommodate the decrease in key value of a particular element
Delete-min	Return the element with the smallest key, and remove it from the set (also: “Dequeue”)
Make-queue	Build a priority queue out of the given elements, with the given key values • In many implementations, this is significantly faster than inserting the elements one by one

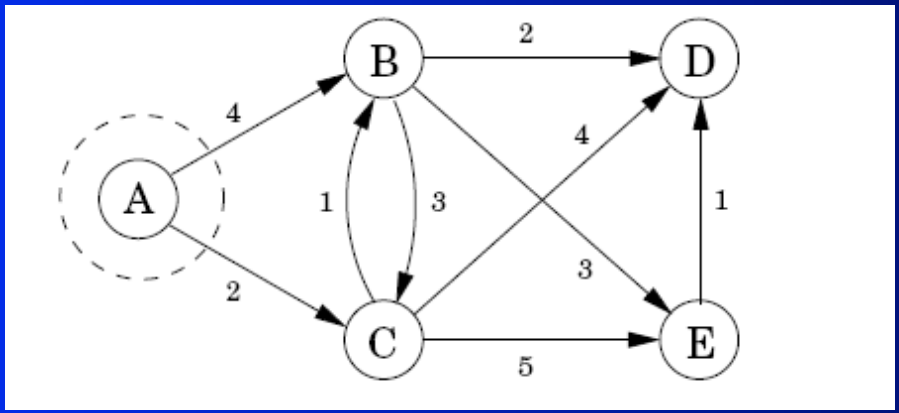
Dijkstra's Algorithm

Figure 4.8 Dijkstra's shortest-path algorithm.

```
procedure dijkstra(G, l, s)
Input:    Graph G = (V, E), directed or undirected;
          positive edge lengths {l_e : e ∈ E}; vertex s ∈ V
Output:   For all vertices u reachable from s, dist(u) is set
          to the distance from s to u.

for all u ∈ V:
    dist(u) = ∞
    prev(u) = nil
dist(s) = 0

H = makequeue(V)   (using dist-values as keys)
while H is not empty:
    u = deletemin(H)
    for all edges (u, v) ∈ E:
        if dist(v) > dist(u) + l(u, v):
            dist(v) = dist(u) + l(u, v)
            prev(v) = u
            decreasekey(H, v)
```

Dijkstra Example

```
H = makequeue(V)   (using dist-values as keys)
while H is not empty:
    u = deletemin(H)
    for all edges (u,v) ∈ E:
        if dist(v) > dist(u) + l(u,v):
            dist(v) = dist(u) + l(u,v)
            prev(v) = u
            decreasekey(H,v)
```



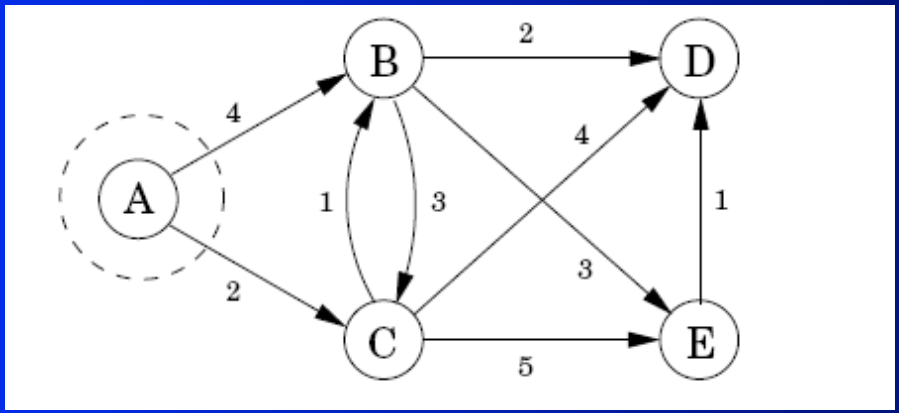
Length of shortest paths:

A: 0
B: ∞
C: ∞
D: ∞
E: ∞

H = makequeue(*V*)
Priority Queue *H*: A:0, B:∞, C:∞, D:∞, E:∞

Dijkstra Example

```
H = makequeue(V)   (using dist-values as keys)
while H is not empty:
    u = deletemin(H)
    for all edges (u, v) ∈ E:
        if dist(v) > dist(u) + l(u, v):
            dist(v) = dist(u) + l(u, v)
            prev(v) = u
            decreasekey(H, v)
```



Length of shortest paths:

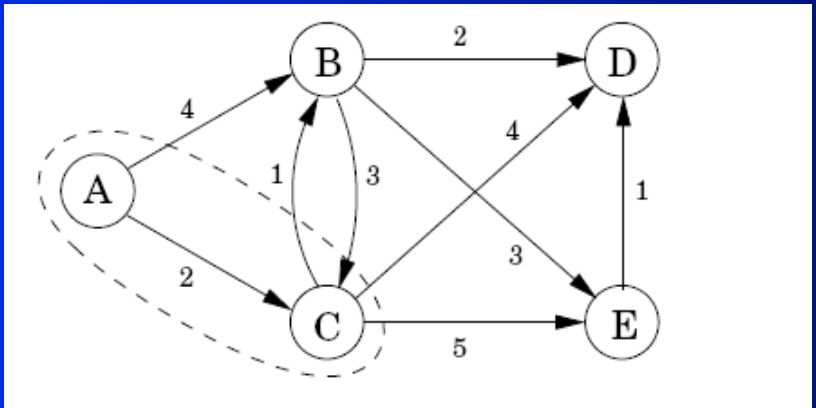
A: 0
B: 4
C: 2
D: ∞
E: ∞

Pull A off of *H*

Priority Queue *H*: C:2, B:4, D:∞, E:∞

Dijkstra Example

```
H = makequeue(V)   (using dist-values as keys)
while H is not empty:
    u = deletemin(H)
    for all edges (u, v) ∈ E:
        if dist(v) > dist(u) + l(u, v):
            dist(v) = dist(u) + l(u, v)
            prev(v) = u
            decreasekey(H, v)
```



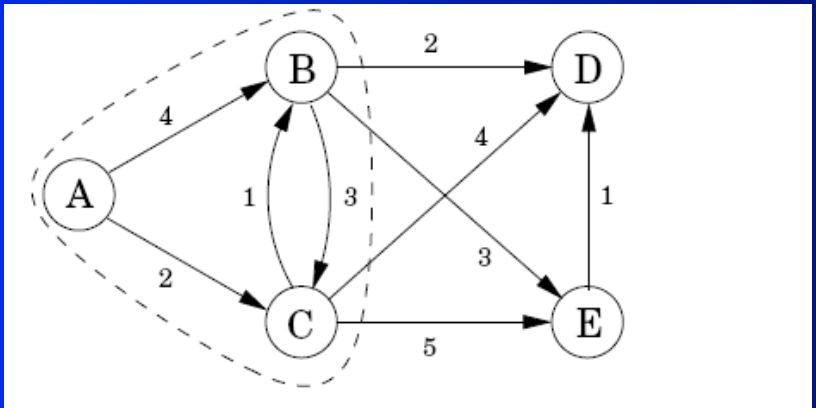
Length of shortest paths:

A: 0
B: 3
C: 2
D: 6
E: 7

Pull C off of *H* and update
Priority Queue *H*: B:3, D:6, E:7
B changes it prev pointer from A to C
Note that once you dequeue a node (deletemin) you are guaranteed that that there is no shorter path to that node. Why?

Dijkstra Example

```
H = makequeue(V)   (using dist-values as keys)
while H is not empty:
    u = deletemin(H)
    for all edges (u, v) ∈ E:
        if dist(v) > dist(u) + l(u, v):
            dist(v) = dist(u) + l(u, v)
            prev(v) = u
            decreasekey(H, v)
```



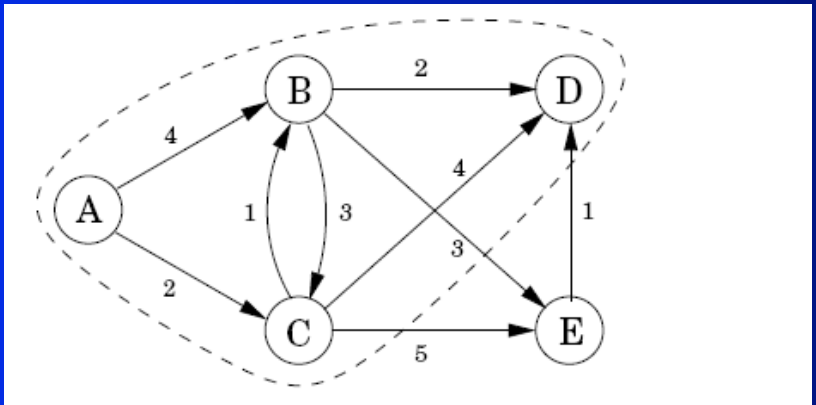
Length of shortest paths:

A: 0
B: 3
C: 2
D: 5
E: 6

Pull B off of *H* and update
Priority Queue *H*: D:5, E:6
Note that decreasekey just works on the frontier, ordering which are the best next nodes to expand, but never undermining an already expanded path

Dijkstra Example

```
H = makequeue(V)   (using dist-values as keys)
while H is not empty:
    u = deletemin(H)
    for all edges (u, v) ∈ E:
        if dist(v) > dist(u) + l(u, v):
            dist(v) = dist(u) + l(u, v)
            prev(v) = u
            decreasekey(H, v)
```



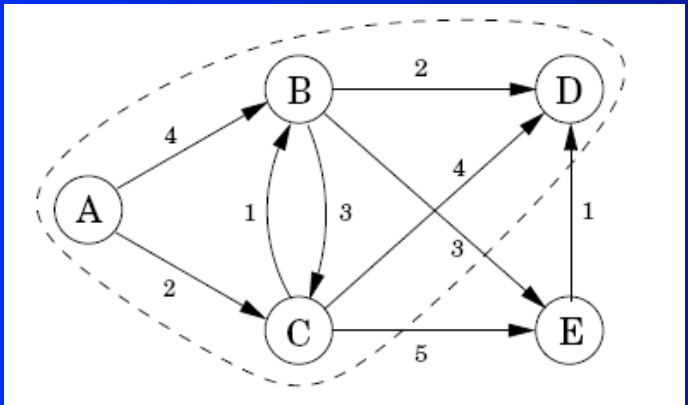
Length of shortest paths:

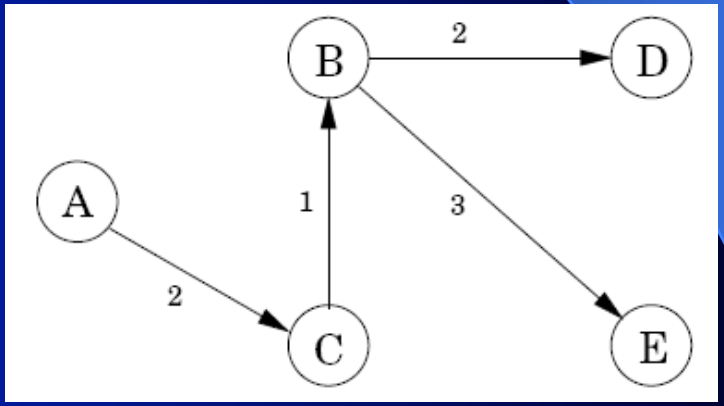
A: 0
B: 3
C: 2
D: 5
E: 6

Pull D off of *H* and update and then E off of *H*
Priority Queue *H*: E:6

Dijkstra Example

```
H = makequeue(V)   (using dist-values as keys)
while H is not empty:
    u = deletemin(H)
    for all edges (u, v) ∈ E:
        if dist(v) > dist(u) + l(u, v):
            dist(v) = dist(u) + l(u, v)
            prev(v) = u
            decreasekey(H, v)
```





A: 0
B: 3
C: 2
D: 5
E: 6

Final Shortest Path Tree
Constructed using prev(u)

Dijkstra's Algorithm Complexity

Figure 4.8 Dijkstra's shortest-path algorithm.

```
procedure dijkstra(G, l, s)
Input:    Graph G = (V, E), directed or undirected;
          positive edge lengths {l_e : e ∈ E}; vertex s ∈ V
Output:   For all vertices u reachable from s, dist(u) is set
          to the distance from s to u.

for all u ∈ V:
    dist(u) = ∞
    prev(u) = nil
dist(s) = 0

H = makequeue(V)   (using dist-values as keys)
while H is not empty:
    u = deletemin(H)
    for all edges (u, v) ∈ E:
        if dist(v) > dist(u) + l(u, v):
            dist(v) = dist(u) + l(u, v)
            prev(v) = u
            decreasekey(H, v)
```

Dijkstra Algorithm Complexity

- Similar to BFS ($O(|V| + |E|)$), but now we have a priority queue instead of a normal queue
- Thus complexity will be $O(|V| \times$(complexity of priority queue operations)$+|E| \times$(complexity of priority queue operations))
- Different priority queue implementations have different complexities
  - Basic operations are insert, make-queue, decrease-key, and delete-min
  - For Dijkstra's algorithm make-queue is done just once and is $O(|V|)$. Worst case is $O(|V| \times$(complexity of insert))
  - delete-min is called once for every node $|V|$
  - decrease-key is called *up to* once for every edge $|E| = |V| \times E_{avg}$
  - the overall complexity is:

$$O(|V| \times (\text{insert}) + |V| \times (\text{delete-min}) + |E| \times (\text{decrease-key}))$$

Priority Queue Implementations

- Array
  - Just ordered by node number, not by key
  - Just an array of size $|V|$
  - Decrease-key/Insert is O(1)
  - Delete-min is O($|V|$) since need to search through entire array each time for smallest key
  - Can keep count of queue items to know when queue is empty, etc.
  - Complexity?
    - O($|V|$×(insert) + $|V|$×(delete-min) + $|E|$×(decrease-key))
- A optimal implementation if graph is dense
  - Dense graphs are less common for large applications

Priority Queue Implementations

- *D*-ary Heap
  - Complete *D*-ary tree
  - Special Constraint: The key value of any node of the tree is less than or equal to that of its children
  - Thus the minimum node is always at the top of the tree
  - Insert: Put new key at next available tree spot (bottom rightmost). Let it "bubble up" tree path, swapping with the node above, until it gets to its proper spot. $\log_d n$ time to "bubble up"
  - Delete-min: Pulls off top and replaces it with last (rightmost) node of tree and then lets it "sift down" to its proper spot – $d\log_d n$ time to "sift down" since have to sort on d sub-nodes at each level to find min to bring up, note that total is O(log$n$) once we set d
  - Decrease-key: same "bubble-up" as Insert, $\log_d(n)$ if maintain separate index into the heap to get to the right element, else O(n)
    - More on this in a second

Priority Queue Implementations

- Binary Heap Implementation notes (can use for project)
  - Can implement the binary heap with an array
    - Note this is *not* an array implementation of a priority queue
  - A balanced *d*/binary tree has a simple fast array indexing scheme

Priority Queue Implementations

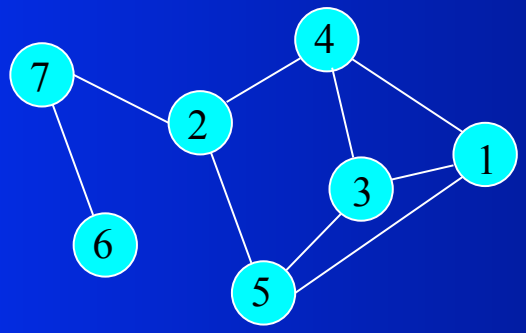
- Binary Heap Implementation notes (can use for project)
  - Can implement the binary heap with an array
    - Note this is *not* an array implementation of a priority queue
  - A balanced *d*/binary tree has a simple fast array indexing scheme
    - node *j* has parent floor(*j*/2)
    - node *j* has children 2*j* and 2*j*+1
  - How do you find an arbitrary node to do decrease_key? – As is, it would be O(|*V*|) to find the node, thus messing up overall complexity
  - Can keep a separate array of size |*V*| which points into the binary heap for each node, thus allowing O(1) lookup of an arbitrary node
    - This is a possible place to keep other info, such as a node flag (not-visited, queued, dequeued) as we will need for One-path
  - Each time you do a binary heap operation with a change in the heap, you need to adjust the pointers in the pointer array.

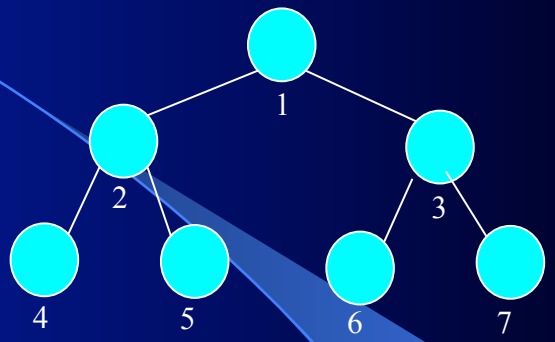
Dijkstra Algorithm Complexity

Implementation	`deletemin`	`insert/ decreasekey`	$\|V\| \times$ `deletemin` + $(\|V\| + \|E\|) \times$ `insert`
Array	$O(\|V\|)$	$O(1)$	$O(\|V\|^2)$
Binary heap	$O(\log \|V\|)$	$O(\log \|V\|)$	$O((\|V\| + \|E\|) \log \|V\|)$
d-ary heap	$O(\frac{d \log \|V\|}{\log d})$	$O(\frac{\log \|V\|}{\log d})$	$O((\|V\| \cdot d + \|E\|) \frac{\log \|V\|}{\log d})$
Fibonacci heap	$O(\log \|V\|)$	$O(1)$ (amortized)	$O(\|V\| \log \|V\| + \|E\|)$

- If dense then unsorted array implementation is best – why?
- Binary heap better as soon as $|E| < |V|^2/\log|V|$
- D-ary heap – depth of heap is $\log_d|V|$
  - constant factor improvement ($\log_2 d$ on depth), though deletemin is bit longer, since have to sort on d sub-nodes at each level to find min to bring up
  - Optimal balance is $d \approx |E|/|V|$ the average degree of the graph
- Fibonacci heap – complex data structure makes this less appealing
  - O(1) (amortized) means over course of the algorithm average is O(1)
- Data Structures Matter!

PQ Implementation Example

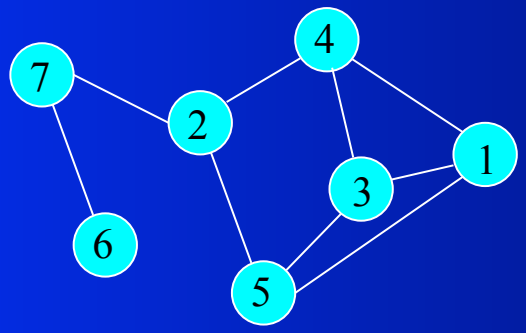


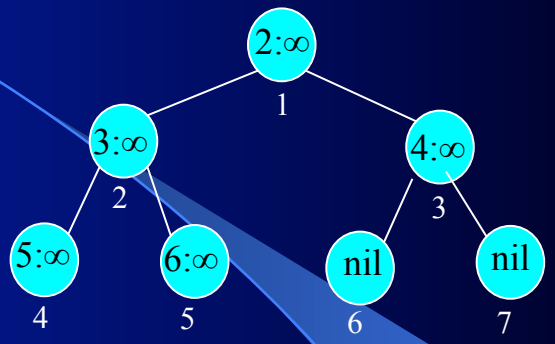


Pointer Array Binary Heap Array

- Note the indexes in the graph and tree are different
- Assume we insert nodes 2-6 into the PQ – what will it look like?

PQ Implementation Example





Pointer Array

1	-1
2	1
3	2
4	3
5	4
6	5
7	-1

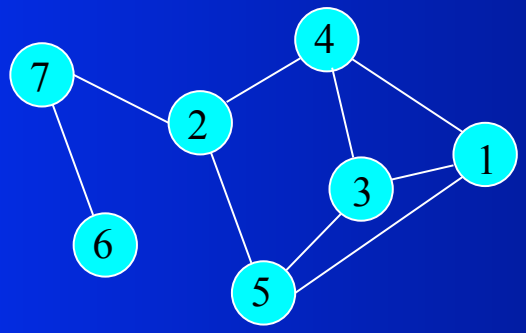
Binary Heap Array

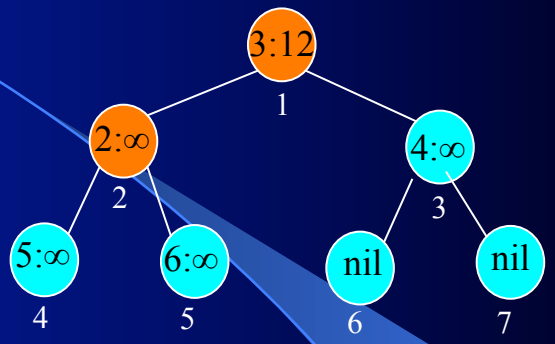
1	2:∞
2	3:∞
3	4:∞
4	5:∞
5	6:∞
6	nil
7	nil

- Note the indexes in the graph and tree are different
- Assume we insert nodes 2-6 into the PQ – what will it look like?
- What happens if we call DecreaseKey(3,12)?

HC: 5

PQ Implementation Example





Pointer Array

1	-1
2	1
3	2
4	3
5	4
6	5
7	-1

1	-1
2	2
3	1
4	3
5	4
6	5
7	-1

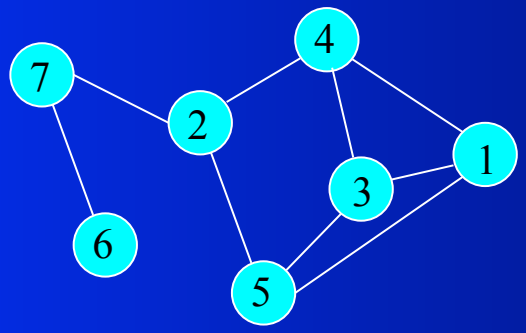
Binary Heap Array

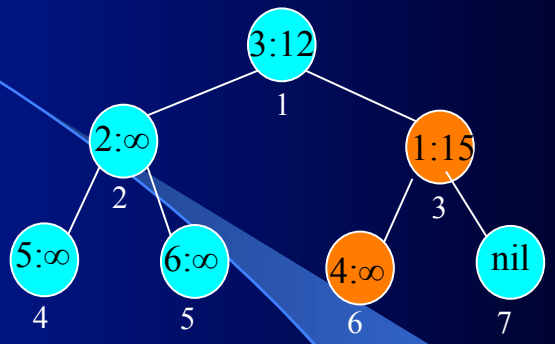
1	2:∞
2	3:∞
3	4:∞
4	5:∞
5	6:∞
6	nil
7	nil

1	3:12
2	2:∞
3	4:∞
4	5:∞
5	6:∞
6	nil
7	nil

- Note the indexes in the graph and tree are different
- Assume we insert nodes 2-6 into the PQ – what will it look like?
- What happens if we call DecreaseKey(3,12)?
- What about Insert(1,15)?

PQ Implementation Example





Pointer Array

1	-1
2	2
3	1
4	3
5	4
6	5
7	-1

1	3
2	2
3	1
4	6
5	4
6	5
7	-1

Binary Heap Array

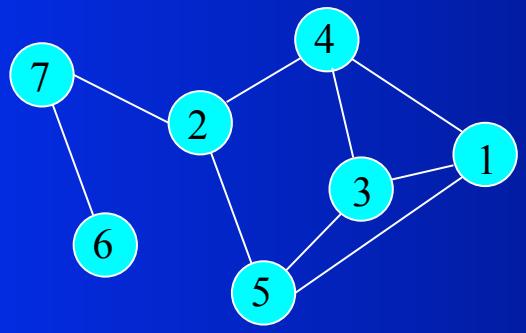
1	3:12
2	2:∞
3	4:∞
4	5:∞
5	6:∞
6	nil
7	nil

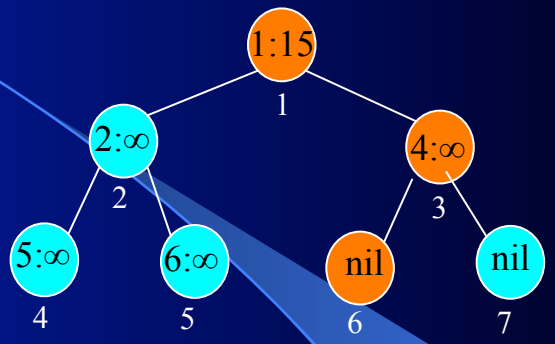
1	3:12
2	2:∞
3	1:15
4	5:∞
5	6:∞
6	4:∞
7	nil

- Note the indexes in the graph and tree are different
- Assume we insert nodes 2-6 into the PQ – what will it look like?
- What happens if we call DecreaseKey(3,12)?
- What about Insert(1,15)?
- What about DM()?

HC: 6

PQ Implementation Example





Pointer Array

1	3
2	2
3	1
4	6
5	4
6	5
7	-1

1	1
2	2
3	-1
4	3
5	4
6	5
7	-1

Binary Heap Array

1	3:12
2	2:∞
3	1:15
4	5:∞
5	6:∞
6	4:∞
7	nil

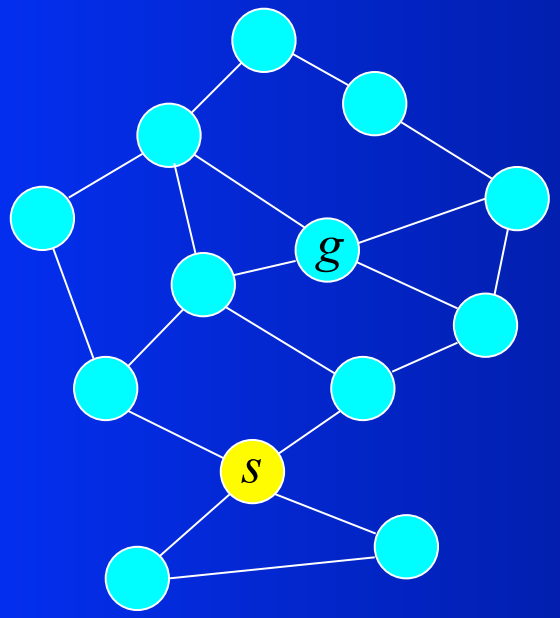
1	1:15
2	2:∞
3	4:∞
4	5:∞
5	6:∞
6	nil
7	nil

- Note the indexes in the graph and tree are different
- Assume we insert nodes 2-6 into the PQ – what will it look like?
- What happens if we call DecreaseKey(3,12)?
- What about Insert(1,15)?
- What about DM()?

HC: 5

Dijkstra Flow with Insert and Goal

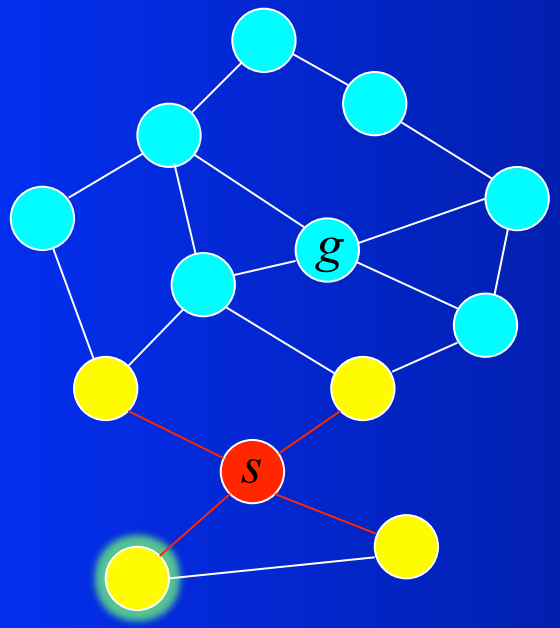
- If graph is large we do not want to start with all nodes in the queue
- This is just a variant (subset) of the algorithm in the text
  - All-Paths vs One-path (my names)
- All distances initially set to ∞ (which is also a flag for not yet queued)
- Yellow: in the Queue
- Red: removed from the Queue



While Queue not empty
    u = deletemin()
    if $u = g$ then break()
    for each edge (u,v)
      if dist(v) = ∞ (i.e. not visited)
        insert(v, dist(u)+len(u,v))
        prev(v) = u
      else if dist(v) > dist(u) + len(u,v)
        dist(v) = dist(u) + len(u,v)
        decreasekey(v, dist(u) + len(u,v))
        prev(v) = u

Dijkstra Flow with Insert and Goal

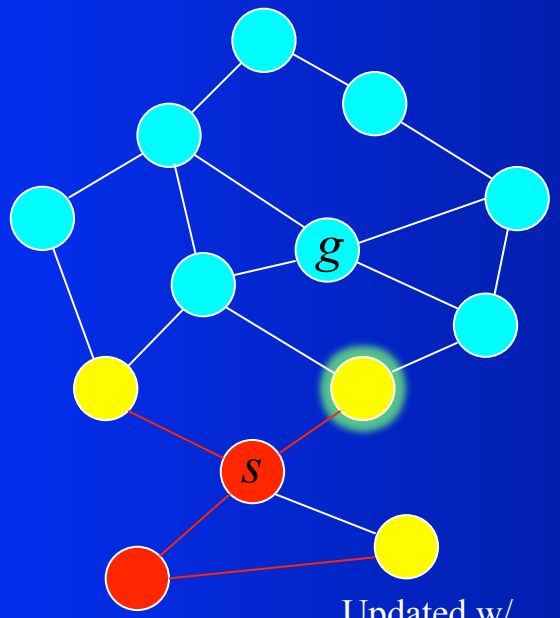
- Yellow: in the Queue
- Red node: removed from Queue
- Red lines: current backpointers which create the cheapest path tree, they can only be created/updated with a just removed node



While Queue not empty
  u = deletemin()
  if $u = g$ then break()
  for each edge (u,v)
    if dist(v) = ∞ (i.e. not visited)
      insert(v, dist(u)+len(u,v))
      prev(v) = u
    else if dist(v) > dist(u) + len(u,v)
      dist(v) = dist(u) + len(u,v)
      decreasekey(v, dist(u) + len(u,v))
      prev(v) = u

Dijkstra Flow with Insert and Goal

- Next deletemin is minimal yellow node (those queued)
- No sense of direction towards *g*, just pushes out the shortest/cheapest frontier until *g* is finally reached
- decreasekey only an option when a removed node has an edge to a node already on the queue

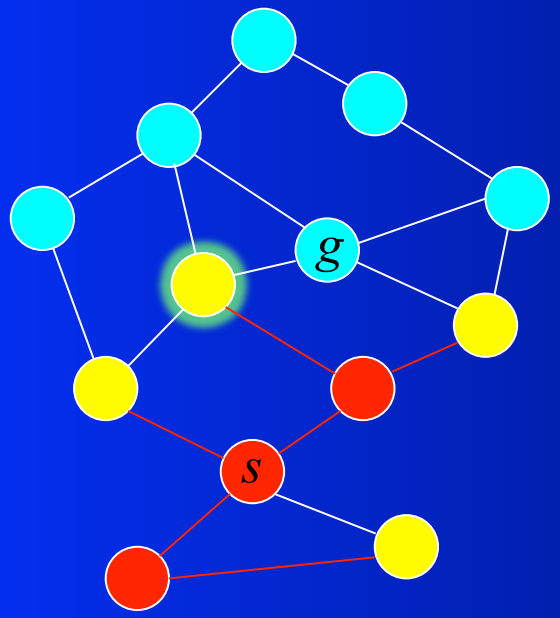


Updated w/ decreasekey

```
While Queue not empty
    u = deletemin()
    if u = g then break()
    for each edge (u,v)
      if dist(v) = ∞ (i.e. not visited )
          insert(v, dist(u)+len(u,v))
          prev(v) = u
      else if dist(v) > dist(u) + len(u,v)
          dist(v) = dist(u) + len(u,v)
          decreasekey(v, dist(u) + len(u,v))
          prev(v) = u
```

Dijkstra Flow with Insert and Goal

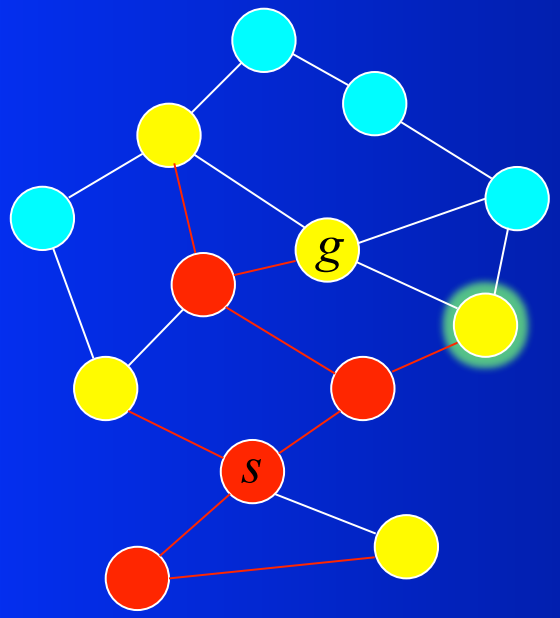
- Why don't we need to revisit a note if it has been removed?
- If it is removed it is already on the least cost path and can never be modified. A later node could have an edge to a removed node but it would have to be a longer path (unless there were negative paths)




```
While Queue not empty
    u = deletemin()
    if u = g then break()
    for each edge (u,v)
      if dist(v) = ∞ (i.e. not visited )
          insert(v, dist(u)+len(u,v))
          prev(v) = u
      else if dist(v) > dist(u) + len(u,v)
          dist(v) = dist(u) + len(u,v)
          decreasekey(v, dist(u) + len(u,v))
          prev(v) = u
```

Dijkstra Flow with Insert and Goal

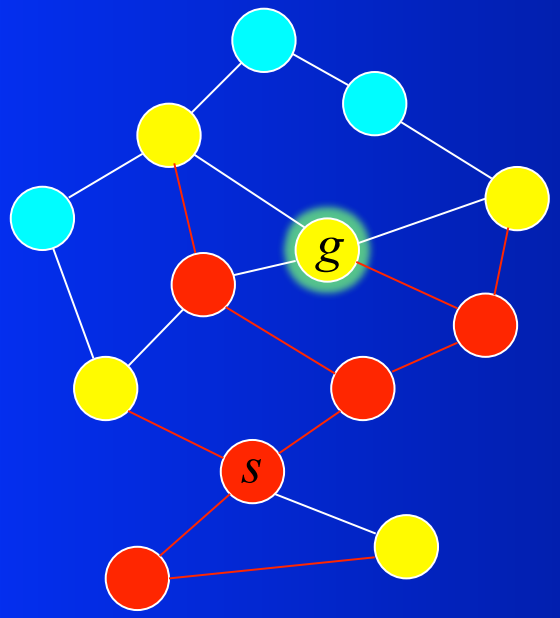
- We can't terminate just when *g* gets on the queue, but as soon as *g* is removed, then we know we have the shortest path




```
While Queue not empty
    u = deletemin()
    if u = g then break()
    for each edge (u,v)
      if dist(v) = ∞ (i.e. not visited )
          insert(v, dist(u)+len(u,v))
          prev(v) = u
      else if dist(v) > dist(u) + len(u,v)
          dist(v) = dist(u) + len(u,v)
          decreasekey(v, dist(u) + len(u,v))
          prev(v) = u
```

Dijkstra Flow with Insert and Goal

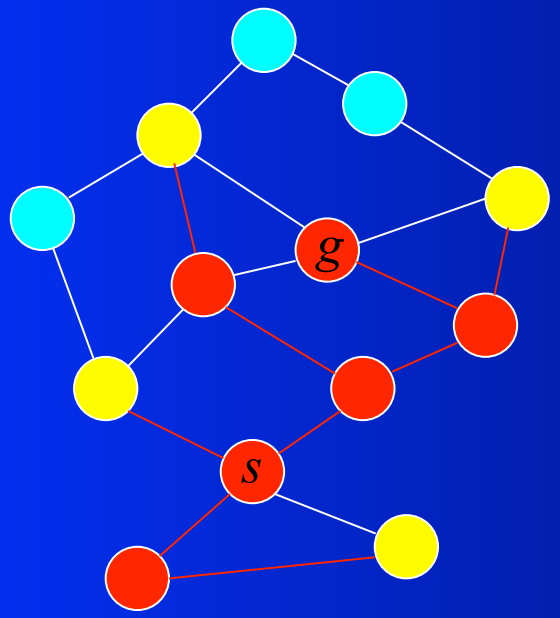
- Another decreasekey example



While Queue not empty
  u = deletemin()
  if $u = g$ then break()
  for each edge (u,v)
    if dist(v) = ∞ (i.e. not visited)
      insert(v, dist(u)+len(u,v))
      prev(v) = u
    else if dist(v) > dist(u) + len(u,v)
      dist(v) = dist(u) + len(u,v)
      decreasekey(v, dist(u) + len(u,v))
      prev(v) = u

Dijkstra Flow with Insert and Goal

- Since *g* had the minimal key it is removed and we can terminate
- Note the red edges are the cheapest path tree from *s*
- Could continue and fill out the entire reachable graph, but not necessary since we reached the goal node




```
While Queue not empty
    u = deletemin()
    if u = g then break()
    for each edge (u,v)
      if dist(v) = ∞ (i.e. not visited )
          insert(v, dist(u)+len(u,v))
          prev(v) = u
      else if dist(v) > dist(u) + len(u,v)
          dist(v) = dist(u) + len(u,v)
          decreasekey(v, dist(u) + len(u,v))
          prev(v) = u
```

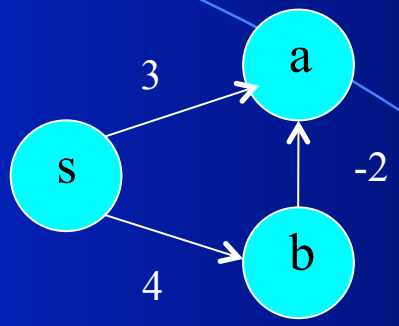
Complexity for One-path Dykstra

- Same as standard Dykstra complexity? For typical cases
- $O(|V|\times(\text{insert}) + |V|\times(\text{delete-min}) + |E|\times(\text{decrease-key})) = O((|V|+|E|)\log|V|)$
- Which should have better constants?
- However,
  - In the main body of One-Path $O(|V|) \leq O(|E|)$ since it only considers reachable nodes, thus the main body is $O(|E|\log|V|)$
    - Which is also the case for All-paths
  - One-path does not need to insert all nodes like All-paths, thus not needing the separate $O(|V|\log|V|)$ complexity (which can dominate in the rare cases where there are many nodes with 0 degree – no edges). Note we could just do all-paths and break when g is dequeued, which is an improvement, but still requires full queue creation. Note that if the break statement were dropped from One-path, the result would be the same as All-paths.
  - One-path still needs to initiate all node distances to ∞ but that is $O(1)$ for each v leading to a full complexity of $O(|V|+|E|\log|V|)$
  - All-paths also has to initiate node distances to ∞, but that $O|V|)$ complexity is subsumed by the $O(|V|\log|V|)$ insert complexity.

Network Routing Project

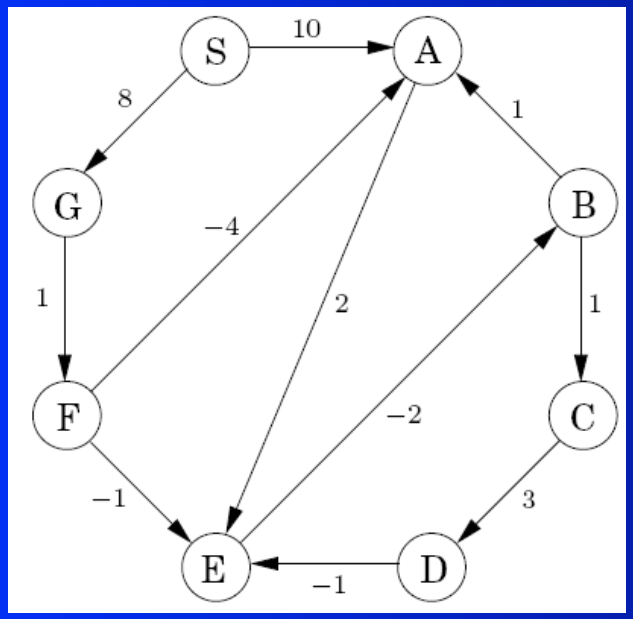
- Interface, etc. is supplied. You will implement Dijkstra's (All-paths and One-path)
  - You will also implement a priority queue where each basic operation (insert, delete-min, and decreaskey) is worse case log*n* (e.g. binary heap*)*
- Network routing uses Dijkstra's algorithm to find the optimal (least-cost) path between source and destination nodes
- You select number of nodes and seed, and a random cost matrix is generated.
- Do you want to find distance from source to every node in the graph?
  - Often the case, since once you have generated the full routing table at a node you do not need to re-calculate paths each time
  - However, networks are constantly changing (new nodes, outages, etc.) so just finding the best path from the source to the destination can in some cases be more efficient than finding them all
  - You will compare the two variations: All-paths and One-path. Both have same Big-O complexity, but…

Negative Edges



- With negative edges we can't keep a guaranteed cheapest frontier, since each time we encounter a negative edge we would have to propagate back decreased costs
- We won't use a priority queue since we can't guarantee the true shortest path is at the front anyway
- Note that the maximum edges in a cheapest path is $|V|$-1, assuming there are no negative cycles
- Just update all edges $|V|$-1 times (max possible path length)
  - Or until no changes are made
  - Gives opportunity for all negative edges in path to properly sum
  - All possible shorter path updates are given a chance (i.e. nodes are not pulled off the queue when they would have been if no negative paths)

```
procedure shortest-paths(G, l, s)
Input:    Directed graph G = (V, E);
          edge lengths {l_e : e ∈ E} with no negative cycles;
          vertex s ∈ V
Output:   For all vertices u reachable from s, dist(u) is set
          to the distance from s to u.

for all u ∈ V:
   dist(u) = ∞
   prev(u) = nil

dist(s) = 0
repeat |V| − 1 times:
   for all e ∈ E:
      update(e)

update(e)
   if dist(v) > dist(u) + l(u, v):
      dist(v) = dist(u) + l(u, v)
      prev(v) = u
```



	Iteration							
Node	0	1	2	3	4	5	6	7
S	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5
C	∞	∞	∞	∞	11	7	6	6
D	∞	∞	∞	∞	∞	14	10	9
E	∞	∞	12	8	7	7	7	7
F	∞	∞	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8

Column n contains the shortest path lengths from S to node x comprising $\leq n$ edges

Bellman-Ford Algorithm

```
procedure shortest-paths(G, l, s)
Input:    Directed graph G = (V, E);
          edge lengths {l_e : e ∈ E} with no negative cycles;
          vertex s ∈ V
Output:   For all vertices u reachable from s, dist(u) is set
          to the distance from s to u.

for all u ∈ V:
    dist(u) = ∞
    prev(u) = nil

dist(s) = 0
repeat |V| − 1 times:
    for all e ∈ E:
        update(e)
```

```
update(e)
    if dist(v) > dist(u) + l(u, v):
        dist(v) = dist(u) + l(u, v)
        prev(v) = u
```

Complexity is ?

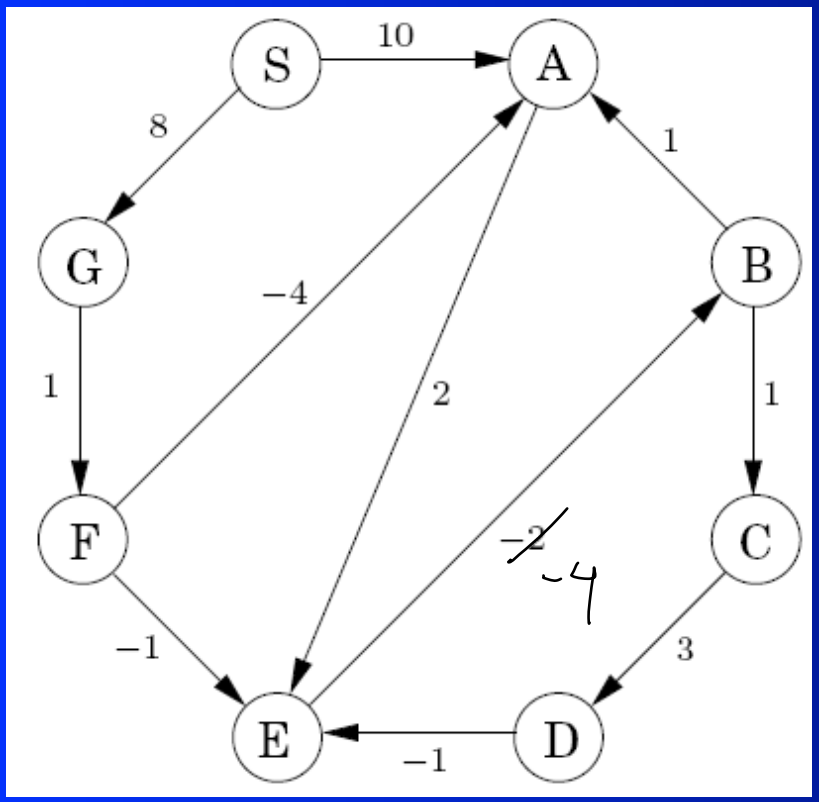
Bellman-Ford Algorithm

```
procedure shortest-paths(G, l, s)
Input:    Directed graph G = (V, E);
          edge lengths {l_e : e ∈ E} with no negative cycles;
          vertex s ∈ V
Output:   For all vertices u reachable from s, dist(u) is set
          to the distance from s to u.

for all u ∈ V:
    dist(u) = ∞
    prev(u) = nil

dist(s) = 0
repeat |V| − 1 times:
    for all e ∈ E:
        update(e)
```

```
update(e)
    if dist(v) > dist(u) + l(u, v):
        dist(v) = dist(u) + l(u, v)
        prev(v) = u
```

Complexity is $|V| \cdot |E|$

Negative Cycles



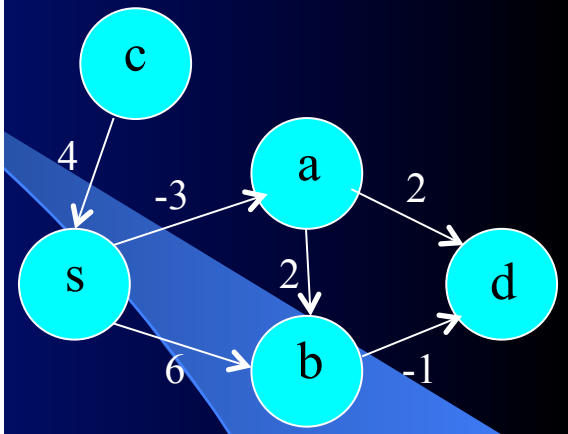
Run Bellman-Ford one more iteration ($|V|$ times). There is a negative cycle iff any cheapest path (distance value) decreases again

When there are negative cycles then by definition cheapest paths do not exist.

Shortest Path with DAGs

```
procedure dag-shortest-paths(G, l, s)
Input:    Dag G = (V, E);
          edge lengths {l_e : e ∈ E}; vertex s ∈ V
Output:   For all vertices u reachable from s, dist(u) is set
          to the distance from s to u.

for all u ∈ V:
   dist(u) = ∞
   prev(u) = nil

dist(s) = 0
Linearize G
for each u ∈ V, in linearized order:
   for all edges (u, v) ∈ E:
      update(u, v)
```

```
update(e)
if dist(v) > dist(u) + l(u, v):
   dist(v) = dist(u) + l(u, v)
   prev(v) = u
```



- Negative paths are all right since there are no cycles and the linearization will cause appropriate ordering