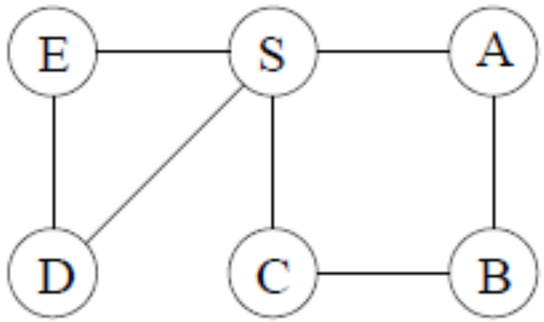
Graph
Path

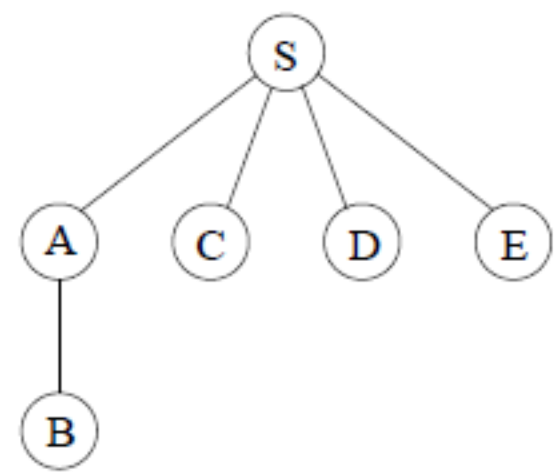
Gou Guanglei(苟光磊)
ggl@cqut.edu.cn

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
```



Order of visitation	Queue contents after processing node
	$[S]$
S	$[A\ C\ D\ E]$
A	$[C\ D\ E\ B]$
C	$[D\ E\ B]$
D	$[E\ B]$
E	$[B]$
B	$[\,]$

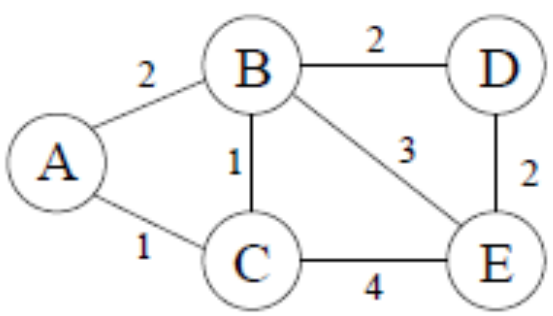


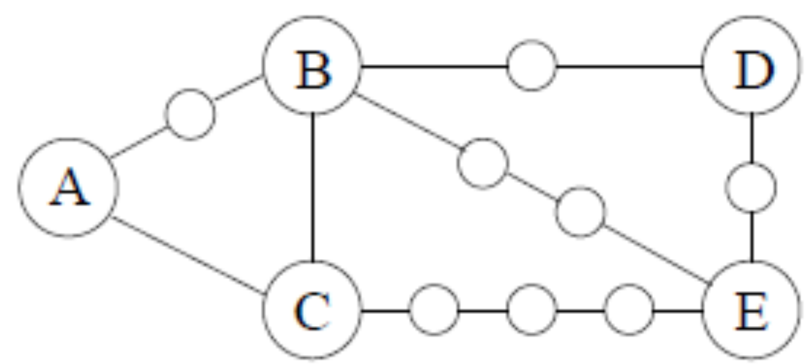
Analysis

- Each vertex is put on the queue exactly once -> 2|V| queue operations
- for loop looks at each edge once (in directed graphs) or twice (in undirected graphs) -> O(|E|) time
- **O(|V | + |E|)**

<u>Weighted graphs</u>

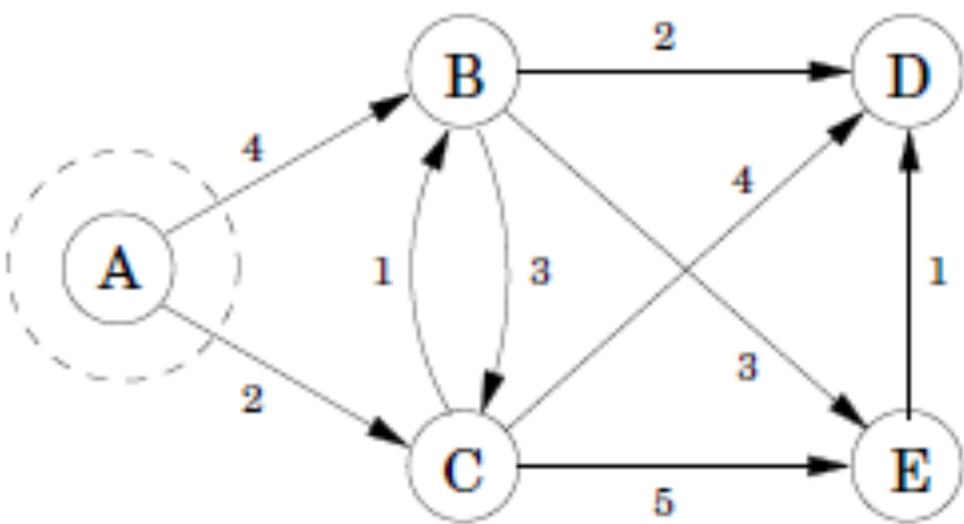
- Breadth-first search finds shortest paths in any graph whose edges have unit length.
- Can we adapt it to a more general graph G = (V, E) whose edge lengths are positive integers?



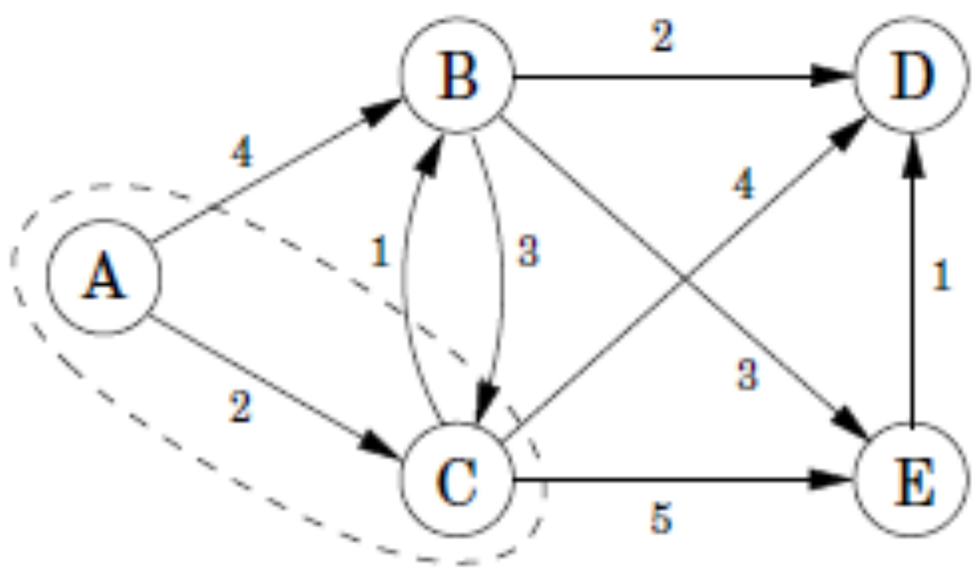


- Dijkstra's algorithm

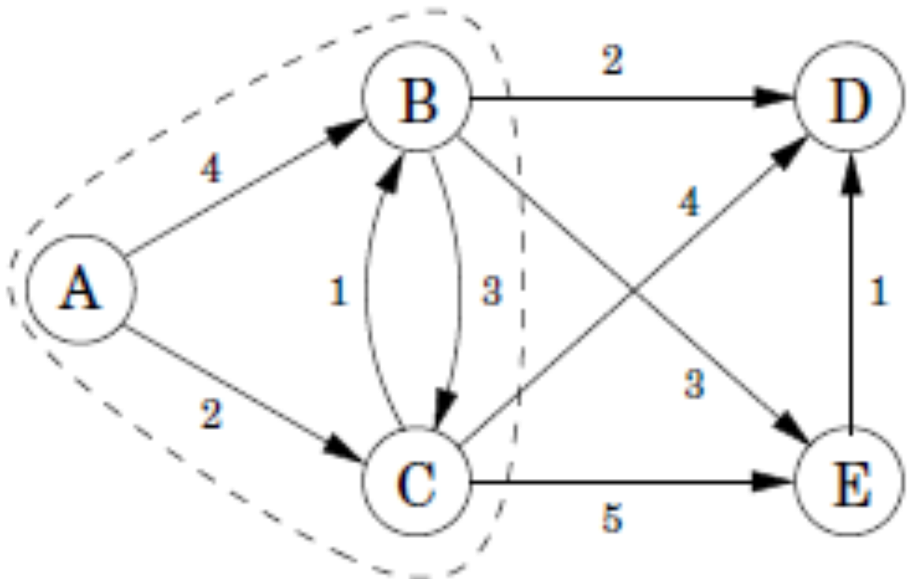
```
procedure dijkstra(G, l, s)
Input:    Graph G = (V, E), directed or undirected;
          positive edge lengths {l_e : e ∈ E}; vertex s ∈ V
Output:   For all vertices u reachable from s, dist(u) is set
          to the distance from s to u.

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)
```



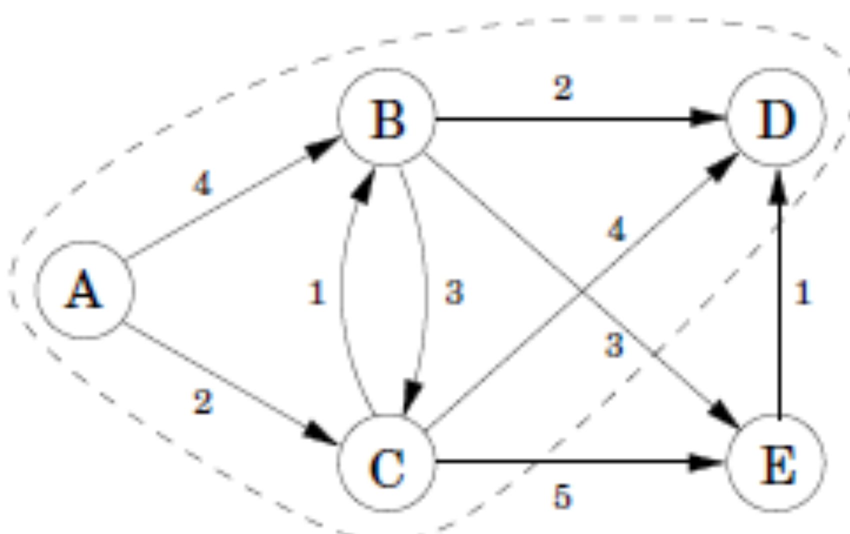
A: 0	D: ∞
B: 4	E: ∞
C: 2	



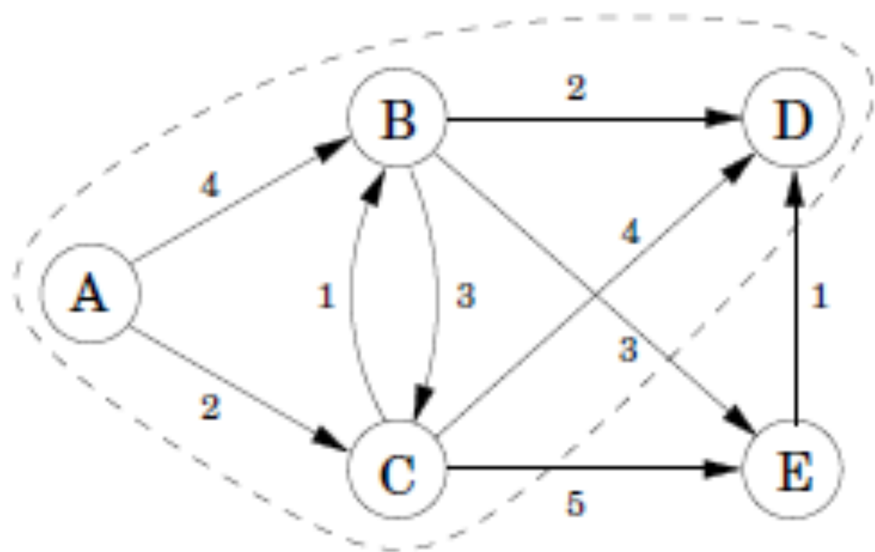
A: 0	D: 6
B: 3	E: 7
C: 2	



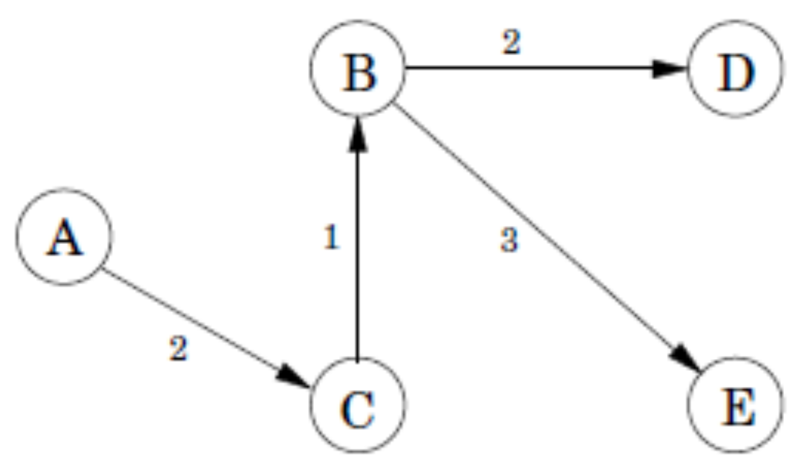
A: 0	D: 5
B: 3	E: 6
C: 2	

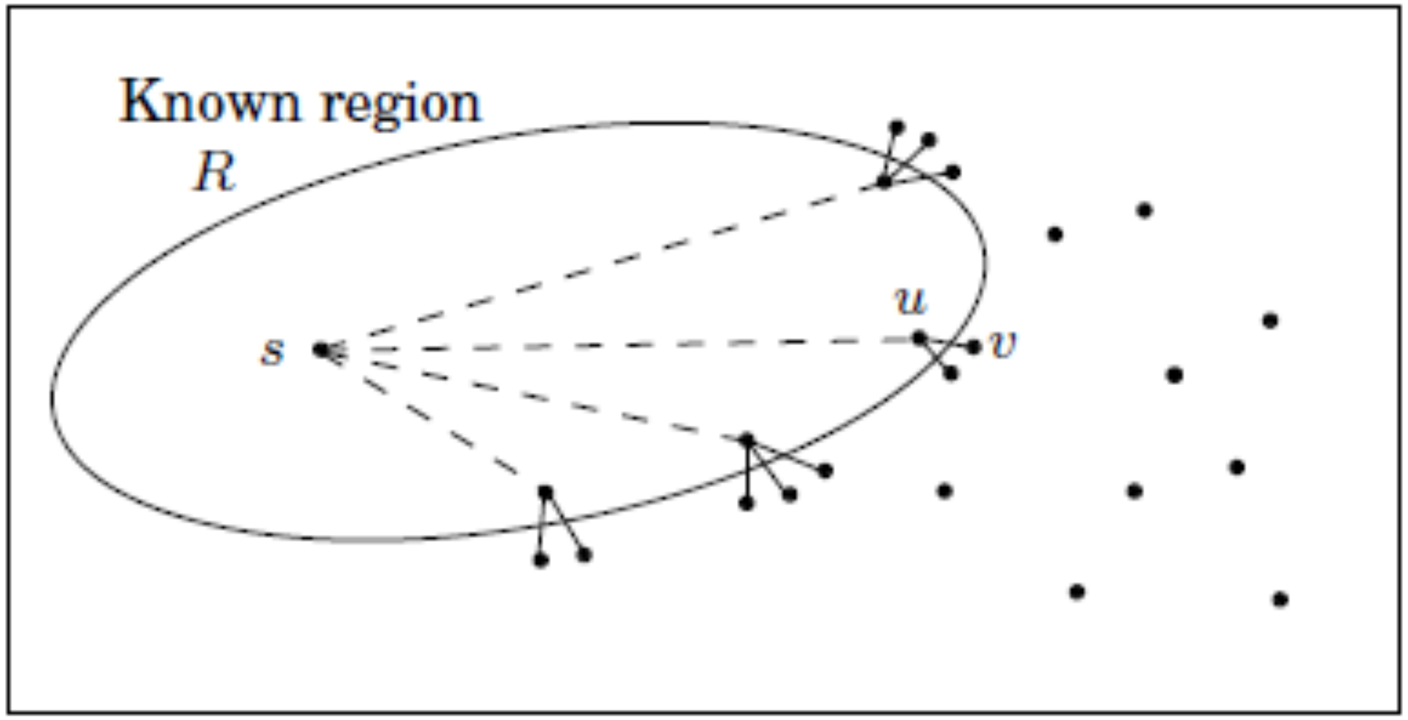


A: 0	D: 5
B: 3	E: 6
C: 2	



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






```
Initialize dist(s) to 0, other dist(·) values to ∞
R = { } (the ``known region'')
while R ≠ V:
   Pick the node v ∉ R with smallest dist(·)
   Add v to R
   for all edges (v,z) ∈ E:
      if dist(z) > dist(v) + l(v,z):
         dist(z) = dist(v) + l(v,z)
```

Correctness

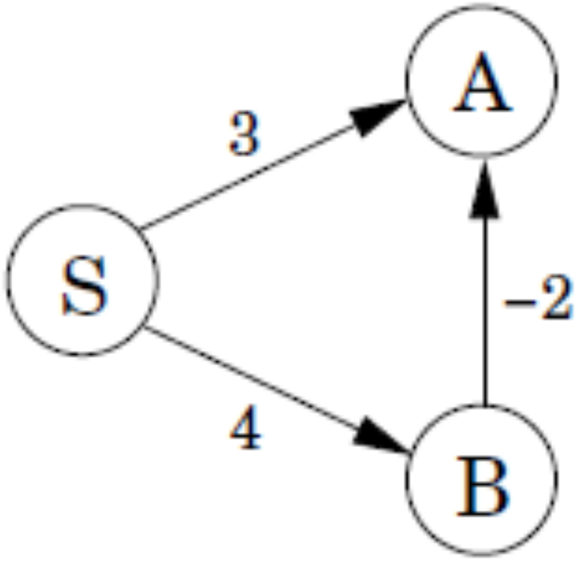
- Use induction.
- At the end of each iteration of the while loop, the following conditions hold:

  (1) there is a value d such that all nodes in R are at distance $\leq$ d from s and all nodes outside R are at distance $\geq$ d from s

  (2) for every node u, the value dist(u) is the length of the shortest path from s to u whose intermediate nodes are constrained to be in R (if no such path exists, the value is ∞).

Negative edges

- Dijkstra's algorithm works in part because the shortest path from the starting point s to any node v must pass exclusively through nodes that are closer than v.
- This **no longer holds** when edge lengths can be **negative**.



Update

- We can consider Dijkstra's algorithm as performing a sequence of the following update procedure.

$$\underline{\texttt{procedure update}}\,((u,v) \in E)$$
$$\texttt{dist}(v) = \min\{\texttt{dist}(v), \texttt{dist}(u) + l(u,v)\}$$

- This *update operation* uses the fact that the distance to v cannot be more than the dist(u) +l(u, v).

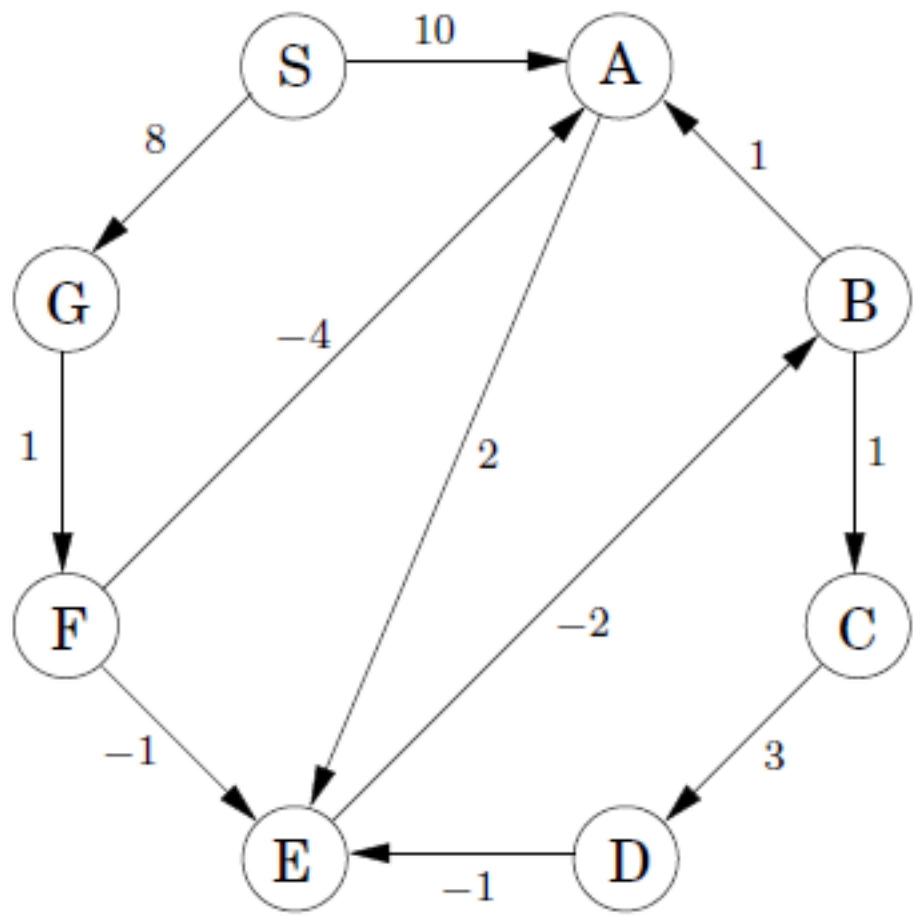
Shortest paths in dags

- we need to perform a sequence of updates that includes every shortest path as a subsequence.
- In any path of a dag, the vertices appear in increasing linearized order.

```
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)
```

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)
```



	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