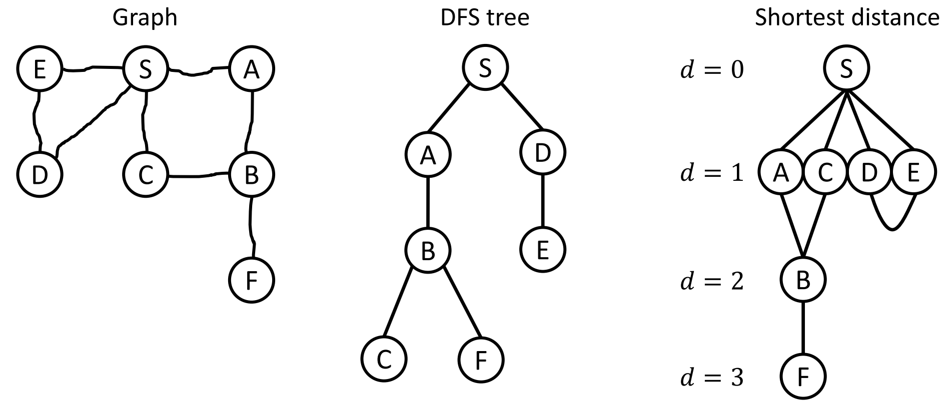
Graphs: Shortest paths

Jordi Cortadella and Jordi Petit
Department of Computer Science

Distance in a graph

Depth-first search finds vertices reachable from another given vertex. The paths are not the shortest ones.

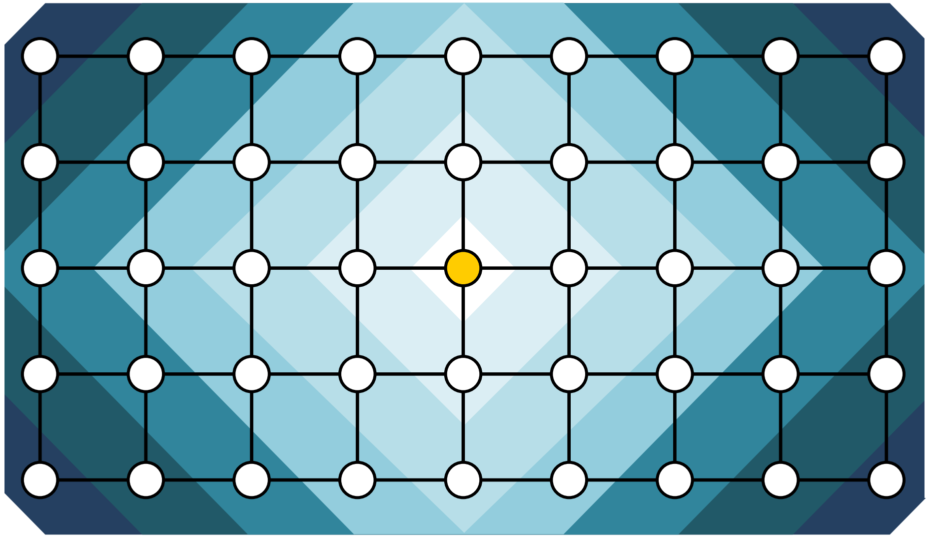


Distance between two nodes: length of the shortest path between them

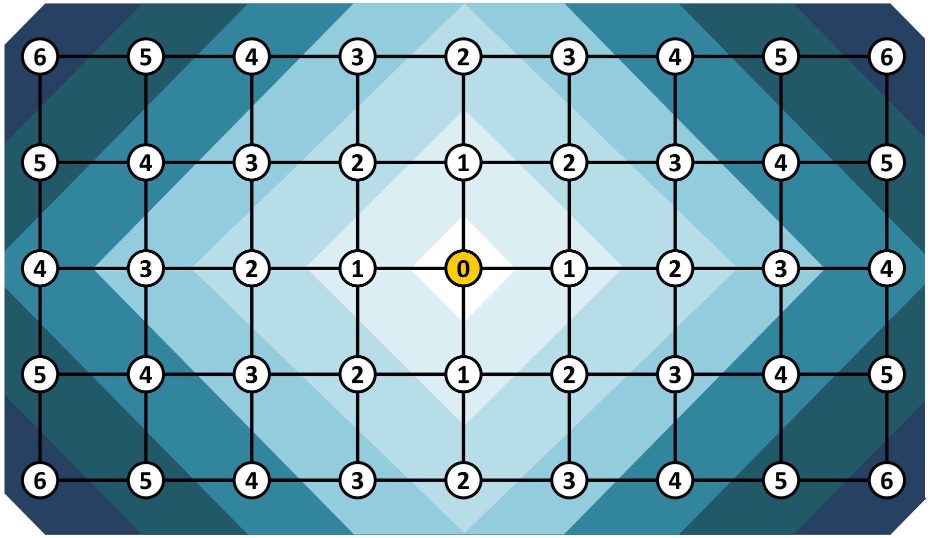
Breadth-first search

Similar to a wave propagation

Breadth-first search



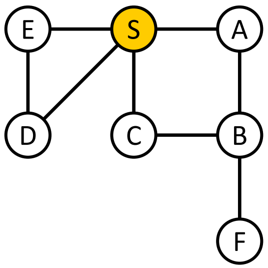
Breadth-first search



BFS algorithm

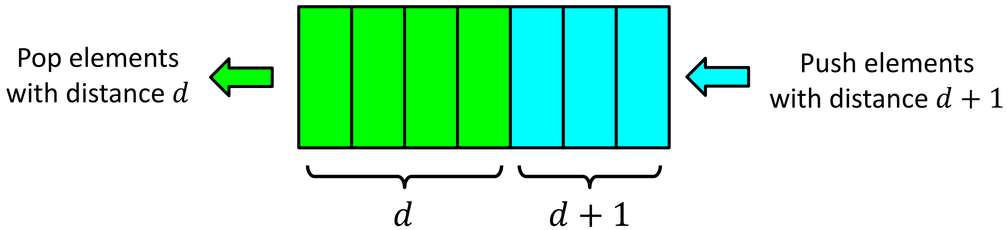
- BFS visits vertices layer by layer: $0,1,2,\ldots,d$.
- Once the vertices at layer d have been visited, start visiting vertices at layer $d+1$.
- Algorithm with two active layers:
  - Vertices at layer d (currently being visited).
  - Vertices at layer $d+1$ (to be visited next).
- Central data structure: a queue.

BFS algorithm: simulation



Visited	Queue	S	A	B	C	D	E	F
	S_0	0	∞	∞	∞	∞	∞	∞
S_0	$A_1\ C_1\ D_1\ E_1$	0	1	∞	1	1	1	∞
A_1	$C_1\ D_1\ E_1\ B_2$	0	1	2	1	1	1	∞
C_1	$D_1\ E_1\ B_2$	0	1	2	1	1	1	∞
D_1	$E_1\ B_2$	0	1	2	1	1	1	∞
E_1	B_2	0	1	2	1	1	1	∞
B_2	F_3	0	1	2	1	1	1	3
F_3		0	1	2	1	1	1	3

BFS queue



BFS algorithm

```
def BFS(G, s) → dist:
  """Input: Graph G(V,E), source vertex s.
     Output: For each vertex u, dist[u] is
             the distance from s to u."""

  for all u ∈ V: dist[u]= ∞

  dist[s] = 0
  Q = {s}  # Queue containing just s
  while not Q.empty():
    u = Q.pop_front()
    for all (u,v) ∈ E:
      if dist[v] == ∞:
        dist[v] = dist[u] + 1
        Q.push_back(v)
```

Runtime $O(|V| + |E|)$: Each vertex is visited once, each edge is visited once (for directed graphs) or twice (for undirected graphs).

Reachability: BFS vs. DFS

Input: A graph G and a source node s.
Output: $\forall u \in V$: visited$[u]$ ⇔ u is reachable from s.
The function processes the nodes in BFS/DFS order

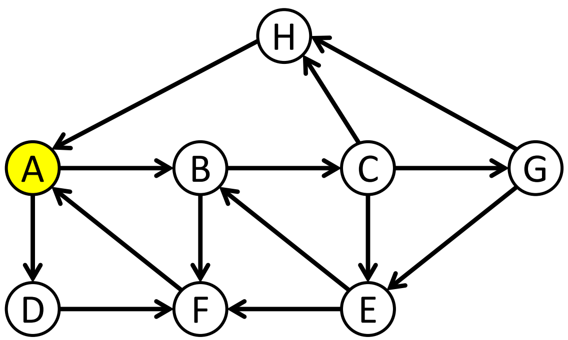
```
def BFS(G, s) → visited:
  for all u ∈ V:
    visited[u] = False

  Q = ⇇      # Empty queue
  Q.push_back(s)
  visited[s] = True
  while not Q.empty():
    u = Q.pop_front()
    process(u)
    for all (u,v) ∈ E:
      if not visited[v]:
        visited[v] = True
        Q.push_back(v)
```

```
def DFS(G, s) → visited:
  for all u ∈ V:
    visited[u] = False

  S = ⊔       # Empty stack
  S.push(s)
  while not S.empty():
    u = S.pop()
    process(u)
    if not visited[u]:
      visited[u] = True
      for all (u,v) ∈ E:
        S.push(v)
```


Reachability: BFS vs. DFS



DFS order:	A	B	C	E	F	G	H	D
BFS order:	A	B	D	C	F	E	G	H
Distance:	0	1	1	2	2	3	3	3

Weights on edges

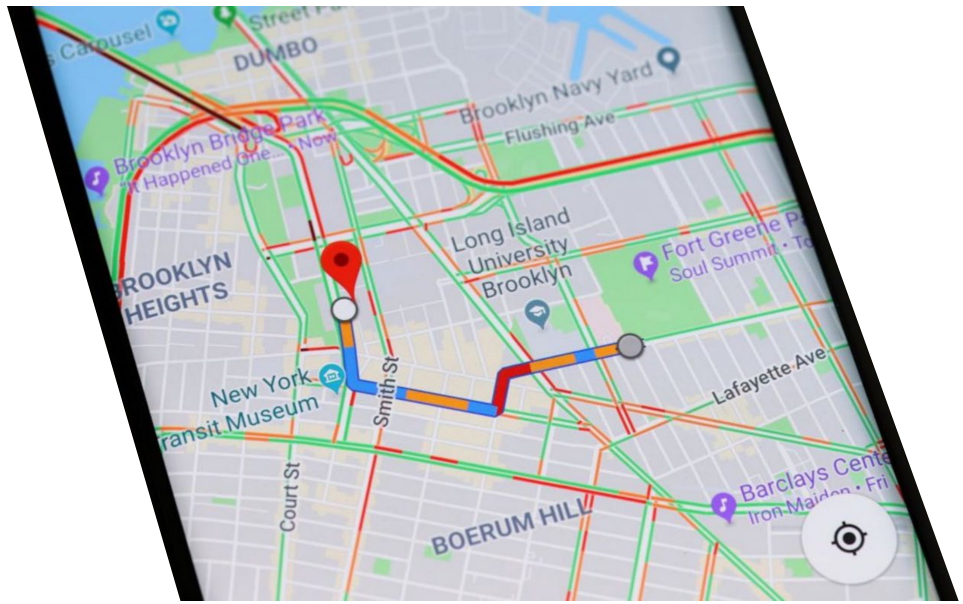
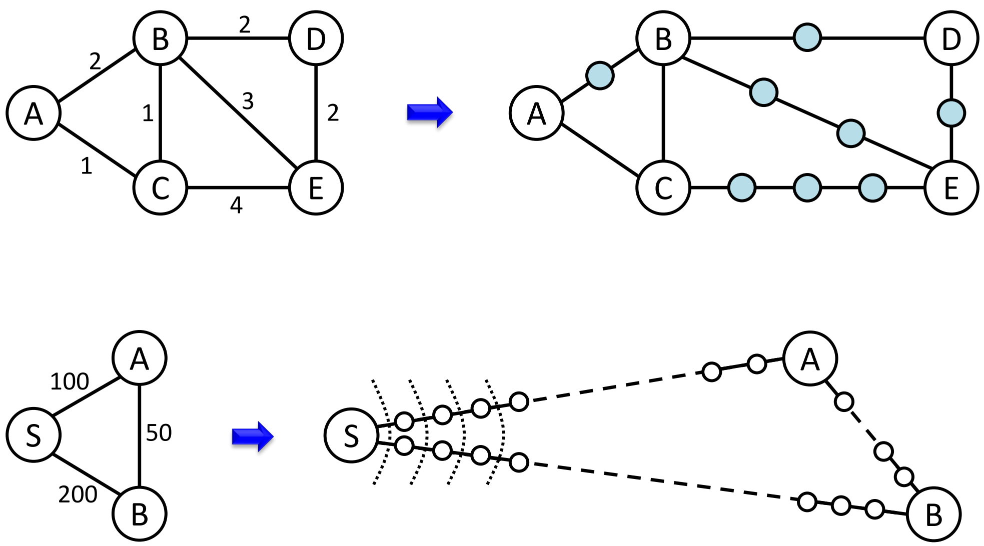


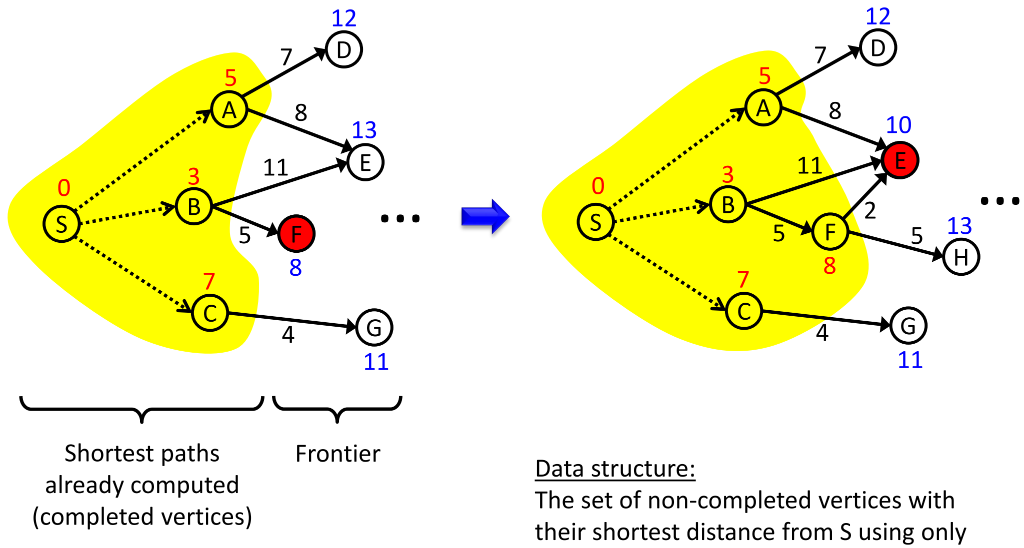
Image credits: https://thegadgetflow.com/blog/google-maps-vs-google-earth/

Reusing BFS



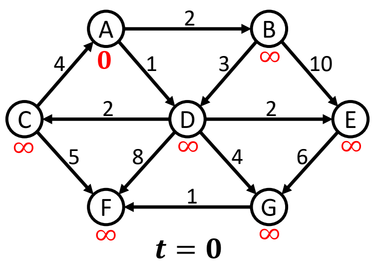
Inefficient: many cycles without any interesting progress. How about real numbers?

Dijkstra's algorithm: invariant

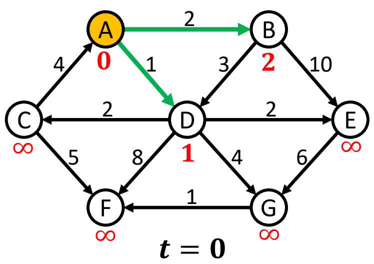


<u>Data structure:</u>
The set of non-completed vertices with their shortest distance from S using only the completed vertices.

Example

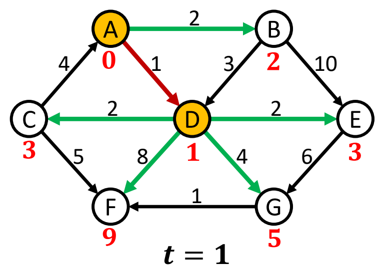


Done	Queue
	A:0
	B:∞
	E:∞
	D:∞
	C:∞
	F:∞
	G:∞

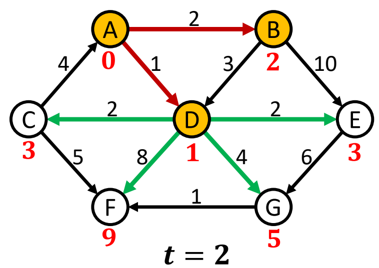


Done	Queue
A:0	D:1
	B:2
	E:∞
	C:∞
	F:∞
	G:∞

Example

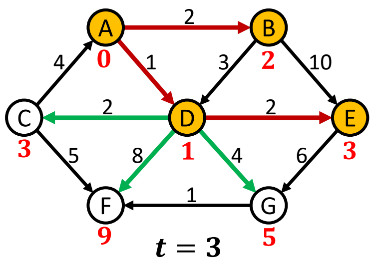


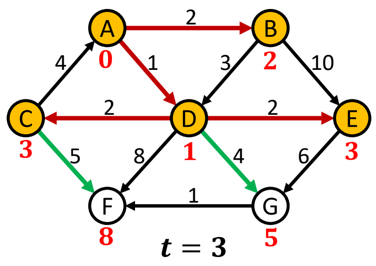
Done	Queue
A:0	B:2
D:1	E:3
	C:3
	G:5
	F:9



Done	Queue
A:0	E:3
D:1	C:3
B:2	G:5
	F:9

Example

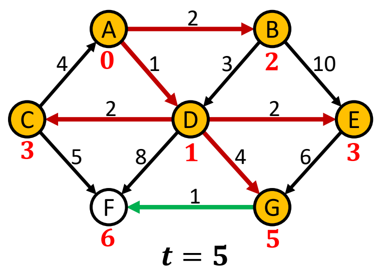


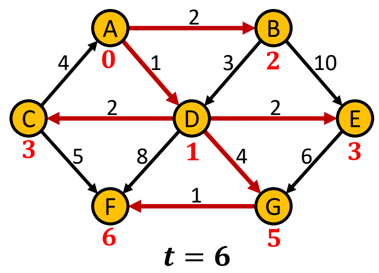


Done	Queue
A:0	C:3
D:1	G:5
B:2	F:9
E:3	

Done	Queue
A:0	G:5
D:1	F:8
B:2	
E:3	
C:3	

Example



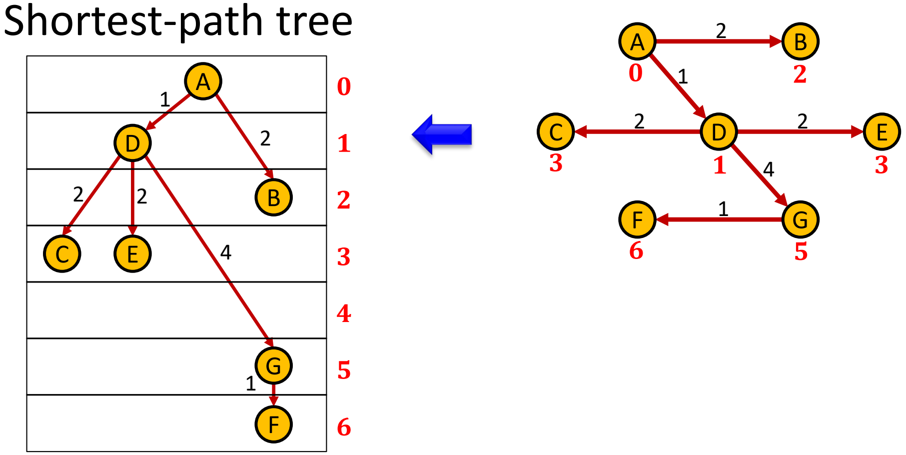


Done	Queue
A:0	F:6
D:1	
B:2	
E:3	
C:3	
G:5	

Done	Queue
A:0	
D:1	
B:2	
E:3	
C:3	
G:5	
F:6	

Example

Shortest-path tree



We need to:

- keep a list non-completed vertices and their expected distances.
- select the non-completed vertex with shortest distance.
- update the distances of the neighbouring vertices.

Dijkstra’s algorithm for shortest paths

```
def ShortestPaths(G, s, len) → dist, prev:
    """Input: Graph G(V,E), source vertex s,
              positive edge lengths {len(e): e ∈ E}
       Output: dist[u] has the distance from s,
               prev[u] has the predecessor in the tree
    """
    for all u ∈ V:
        dist[u] = ∞
        prev[u] = nil

    dist[s] = 0
    Q = makequeue(V)  # priority queue (dist as value)

    while not Q.empty():
        u = Q.deletemin()
        for all (u,v) ∈ E:
            if dist[v] > dist[u] + len(u,v):
                dist[v] = dist[u] + len(u,v)
                prev[v] = u
                Q.decreasekey(v)
```

Dijkstra's algorithm: complexity

```
Q = makequeue(V)
while not Q.empty():
  u = Q.deletemin()                       ← |V| times
  for all (u,v) ∈ E:
    if dist[v] > dist[u] + len(u,v):
      dist[v] = dist[u] + len(u,v)
      prev[v] = u
      Q.decreasekey(v)                    ← |E| times
```

- The skeleton of Dijkstra's algorithm is based on BFS, which is $O(|V| + |E|)$
- We need to account for the cost of:
  - **makequeue**: insert $|V|$ vertices to a list.
  - **deletemin**: find the vertex with min dist in the list ($|V|$ times)
  - **decreasekey**: update dist for a vertex ($|E|$ times)
- Let us consider two implementations for the list: **vector** and **binary heap**

Dijkstra's algorithm: complexity

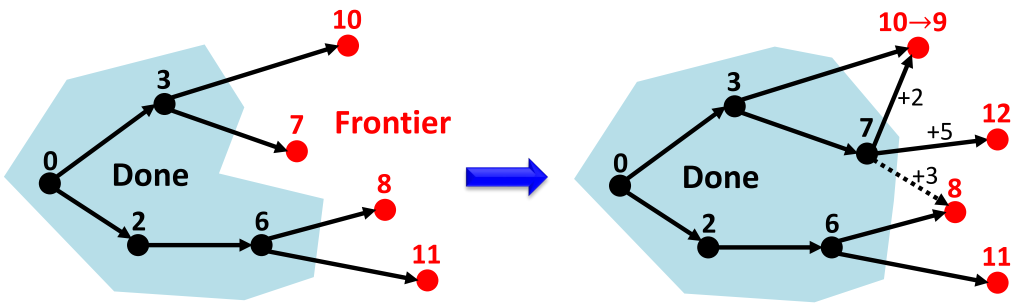
Implementation	deletemin	insert/ decreasekey	Dijkstra's complexity
Vector	$O(\|V\|)$	$O(1)$	$O(\|V\|^2)$
Binary heap	$O(\log \|V\|)$	$O(\log \|V\|)$	$O((\|V\| + \|E\|) \log \|V\|)$

Binary heap:

- The elements are stored in a complete (balanced) binary tree.
- **Insertion:** place element at the bottom and let it *bubble up* swapping the location with the parent (at most $\log_2 |V|$ levels).
- **Deletemin:** Remove element from the root, take the last node in the tree, place it at the root and let it *bubble down* (at most $\log_2 |V|$ levels).
- **Decreasekey:** decrease the key in the tree and let it *bubble up* (same as insertion). A data structure might be required to known the location of each vertex in the heap (table of pointers).

For connected graphs: $O((|V| + |E|) \log |V|) = O(|E| \log |V|)$

Why Dijkstra's works

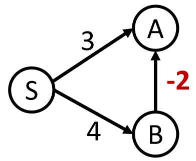


- A tree of open paths with distances is maintained at each iteration.
- The shortest paths for the internal nodes have already been calculated.
- The node in the frontier with shortest distance is "frozen" and expanded. Why? Because no other shorter path can reach the node.

Disclaimer: this is only true if the **distances are non-negative!**

Graphs with negative edges

- Dijkstra's algorithm does not work:



Dijkstra would say that the shortest path S→A has length=3.

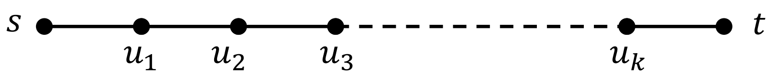
- Dijkstra is based on a safe update each time an edge (u, v) is treated:

$$\text{dist}(v) = \min\{\text{dist}(v), \text{dist}(u) + l(u, v)\}$$

- Problem: shortest paths are consolidated too early.
- Possible solution: add a constant weight to all edges, make them positive, and apply Dijkstra.
  – It does not work, prove it!

Graphs with negative edges

- The shortest path from s to t can have at most $|V| - 1$ edges:



- If the sequence of updates includes

$$(s, u_1), (u_1, u_2), (u_2, u_3), \dots, (u_k, t),$$

in that order, the shortest distance from s to t will be computed correctly (updates are always safe). Note that the sequence of updates does not need to be consecutive.

- Solution: update all edges $|V| - 1$ times !

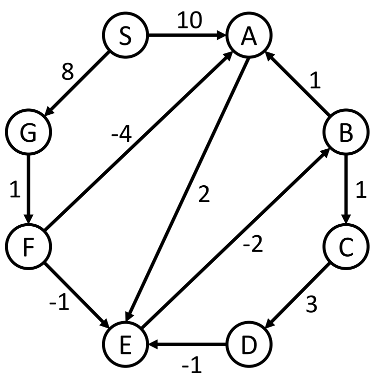
- Complexity: $O(|V| \cdot |E|)$.

Bellman-Ford algorithm

```
def ShortestPaths(G, s, len) → dist, prev:
    """Input: Graph G(V,E), source vertex s,
              edge lengths {len(e): e ∈ E}, no negative cycles
       Output: dist[u] has the distance from s,
               prev[u] has the predecessor in the tree
    """
    for all u ∈ V:
        dist[u] = ∞
        prev[u] = nil

    dist[s] = 0
    repeat |V| − 1 times:
        for all (u,v) ∈ E:
            if dist[v] > dist[u] + len(u,v):
                dist[v] = dist[u] + len(u,v)
                prev[v] = u
```

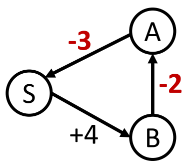

Bellman-Ford: example



	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

Negative cycles

- What is the shortest distance between S and A?



Bellman-Ford does not work as it assumes that the shortest path will not have more than $|V| - 1$ edges.

- A negative cycle produces $-\infty$ distances by endlessly applying rounds to the cycle.

- How to detect negative cycles?
  - Apply Bellman-Ford (update edges $|V| - 1$ times)
  - Perform an extra round and check whether some distance decreases.

Shortest paths in DAGs

- DAG's property:

  In any path of a DAG, the vertices appear in increasing topological order.

- Any sequence of updates that preserves the topological order will compute distances correctly.

- Only one round visiting the edges in topological order is sufficient: $O(|V| + |E|)$.

- How to calculate the longest paths?
  - Negate the edge lengths and compute the shortest paths.
  - Alternative: update with max (instead of min).

DAG shortest paths algorithm

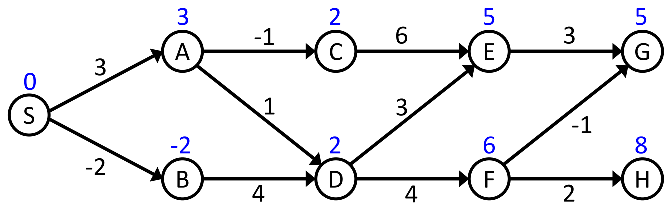
```
def DagShortestPaths(G, s, len) → dist, prev:
    """Input: DAG G(V,E), source vertex s,
              edge lengths {len(e):e∈E}
       Output: dist[u] has the distance from s,
               prev[u] has the predecessor in the tree
    """
    for all u ∈ V:
        dist[u] = ∞
        prev[u] = nil

    dist[s] = 0
    Linearize G
    for all u ∈ V in linearized order:
        for all (u,v) ∈ E:
            if dist[v] > dist[u] + len(u,v):
                dist[v] = dist[u] + len(u,v)
                prev[v] = u
```

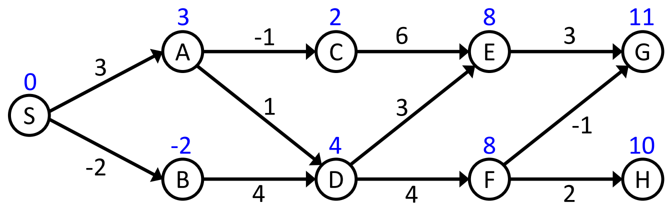

DAG shortest/longest paths: example

Linearization: S A B C D E F G H

Shortest paths



Longest paths



Shortest paths: summary

Single-source shortest paths

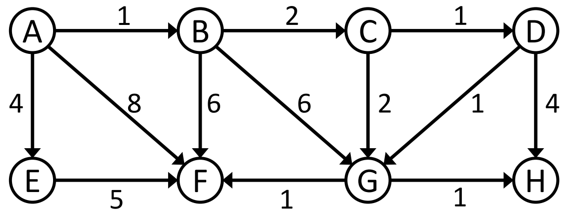
Graph	Algorithm	Complexity
Unit edge-length	BFS	$O(\|V\| + \|E\|)$
Non-negative edges	Dijkstra	$O((\|V\| + \|E\|) \log \|V\|)$
Negative edges	Bellman-Ford	$O(\|V\| \cdot \|E\|)$
DAG	Topological sort	$O(\|V\| + \|E\|)$

A related problem: All-pairs shortest paths

- Floyd-Warshall algorithm ($O(|V|^3)$), based on dynamic programming.
- Other algorithms exist.

EXERCISES

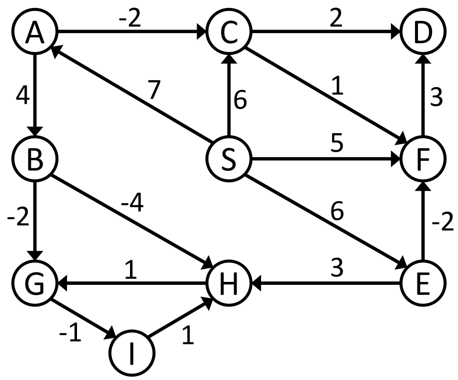
Dijkstra (from [DPV2008])



Run Dijkstra's algorithm starting at node A:

- Draw a table showing the intermediate distance values of all the nodes at each iteration
- Show the final shortest-path tree

Bellman-Ford (from [DPV2008])



Run Bellman-Ford algorithm starting at node S:

- Draw a table showing the intermediate distance values of all the nodes at each iteration
- Show the final shortest-path tree