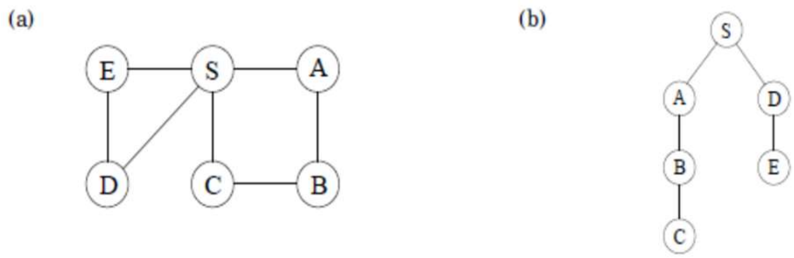
Chapter 4: Paths in Graphs

4.1 Breadth-first search (BFS)

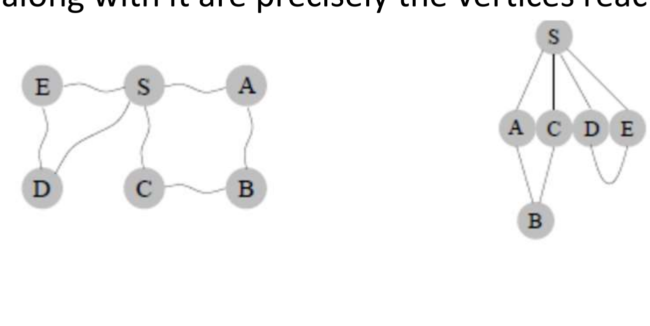
Paths

- A spanning tree provides explicit paths between connected vertices.
- For instance, the (b) below is a DFS tree of graph (a) starting from S.
- From (b), one may identify a path between any two nodes.
- For instance, the path between C and S in the tree has a length of 3.
- But, the *shortest path* has length of 1 from C to S in (a).



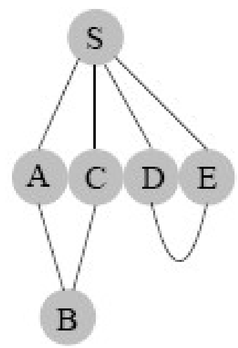
Finding the shortest path physically

- Consider a physical realization of a graph as *a ball for each vertex* and a piece of equal length *string for each edge*.
- If we lift the ball for vertex S high enough, the other balls that get pulled up along with it are precisely the vertices reachable from S.

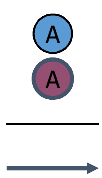


Breadth-first search (BFS)

- Breadth-first search (BFS) starts from a given vertex and visits ***all of the neighbor nodes at the present depth*** (level) prior to moving on to the next depth (level).



BFS example (starting from A)

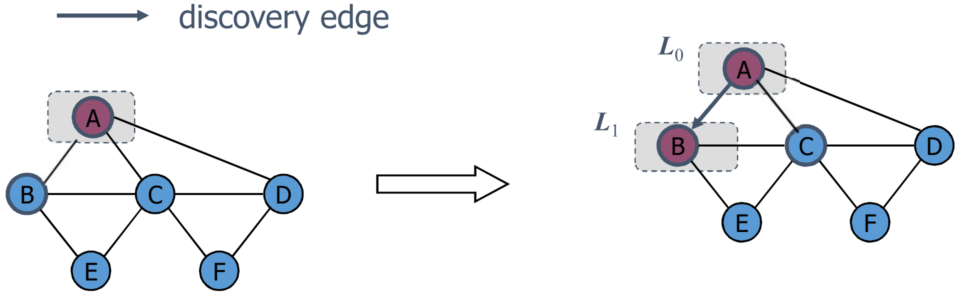


unexplored vertex

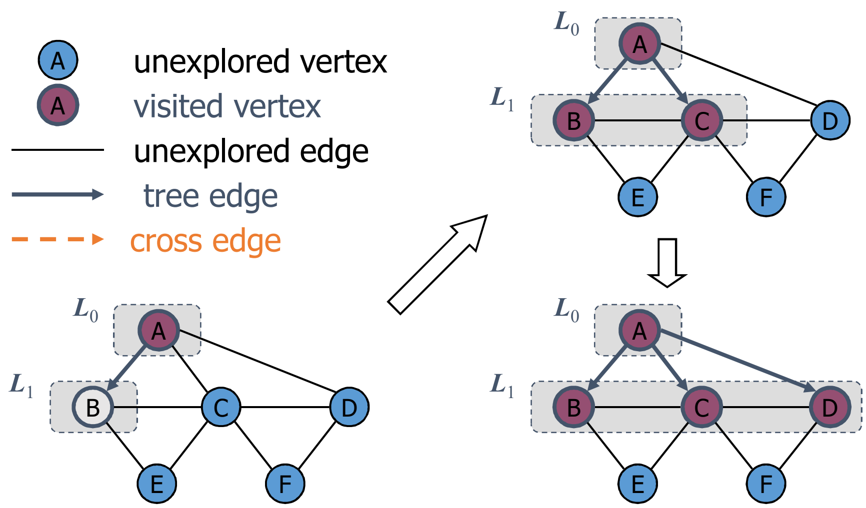
visited vertex

unexplored edge

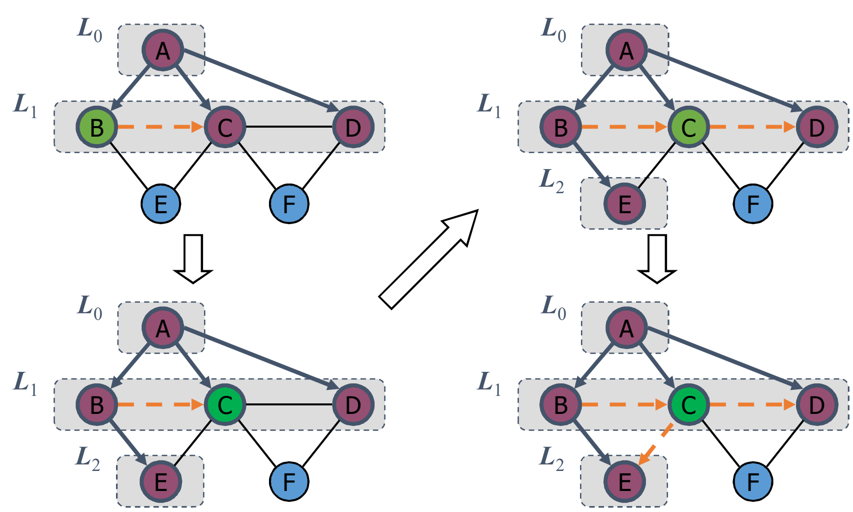
discovery edge



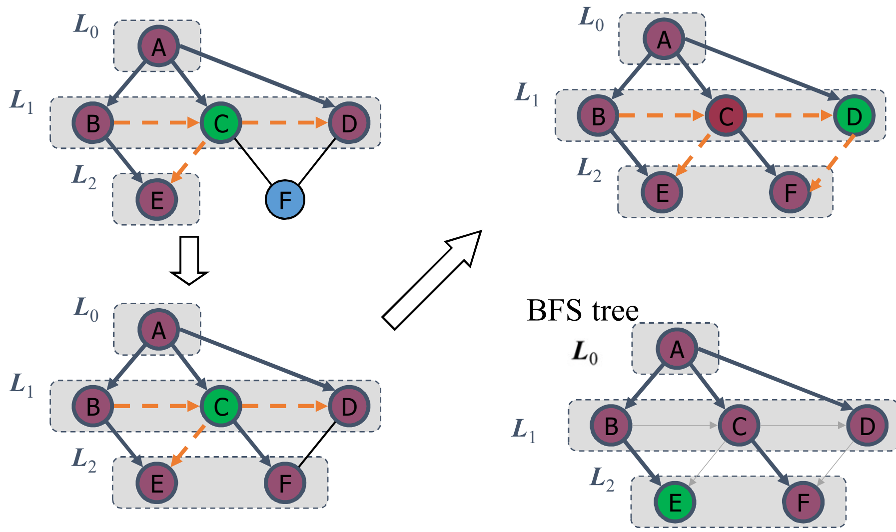
Example



Example (cont.)



Example (cont.)

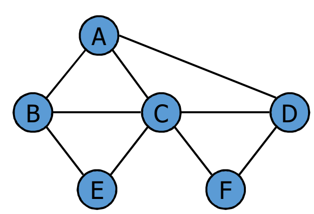


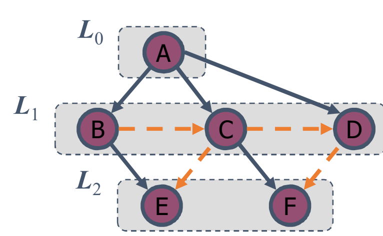
Properties

Property 1: ***BFS*(*G, s*)** visits all the vertices and edges connected to ***s***

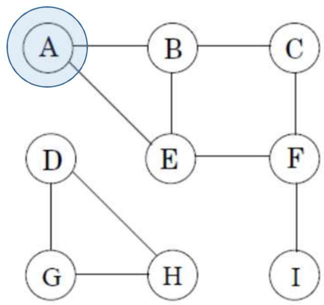
Property 2: The discovery edges labeled by ***BFS*(*G, s*)** form a spanning tree $\boldsymbol{T_s}$

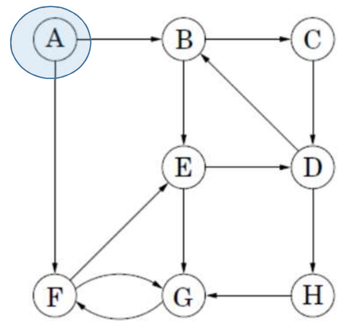
Property 3: For each vertex ***v*** in $\boldsymbol{L_i}$, the path of $\boldsymbol{T_s}$ from ***s*** to ***v*** has ***i*** edges and every path from ***s*** to ***v*** in ***G*** has at least ***i*** edges if connected





In class exercises: compare DFS and BFS trees





BFS

```
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
```

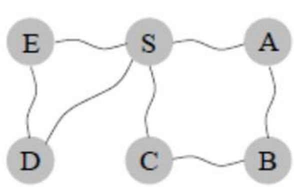
Code: `bfs.py`

```
def bfs(graph, start):
    visited, queue = set(), [start]
    p =[]
    while queue:
      vertex = queue.pop(0)
      if vertex not in visited:
         visited.add(vertex)
         p.append(vertex)
         queue.extend(graph[vertex]\
                 - visited)
    return p
```

```
g = {'A': set(['B', 'C']),
        'B': set(['A', 'D', 'E']),
        'C': set(['A', 'F']),
        'D': set(['B']),
        'E': set(['B', 'F']),
        'F': set(['C', 'E'])
    }

v = bfs(g, 'A')
print(v)
```

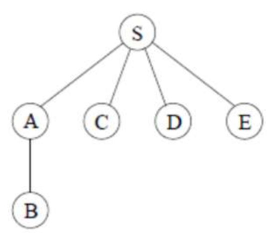
Visualizing the execution https://goo.gl/A8jY3y

Revisit the motivation example




```
G = {'S': ['A','C','D','E'],
     'A': ['S','B'],
     'B': ['A','C'],
     'C': ['S','B'],
     'D': ['S','E'],
     'E': ['S','D']}
```

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	[]

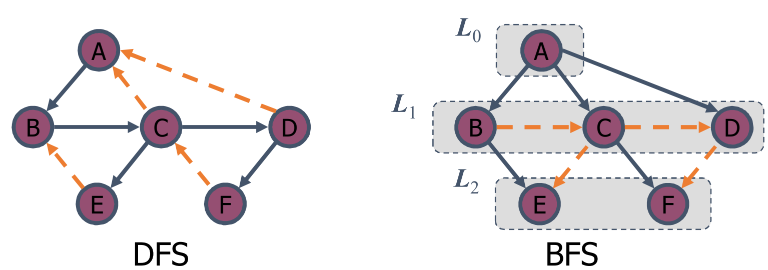


Time complexity of BFS

- Setting/getting a vertex/edge label takes ***O***(1) time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or CROSS
- Each vertex is inserted once into a sequence L_i
- Method *incidentEdges* is called once for each vertex
- Hence, BFS is **O**(**|V|** + **|E|**) time.

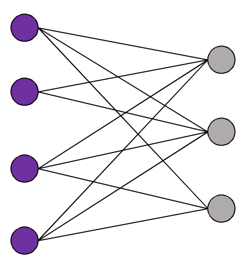
DFS vs. BFS

Applications	DFS	BFS
Spanning tree/forest, path	Yes	Yes
Shortest path	**No**	Yes
Topological order of DAG	Yes	**No**



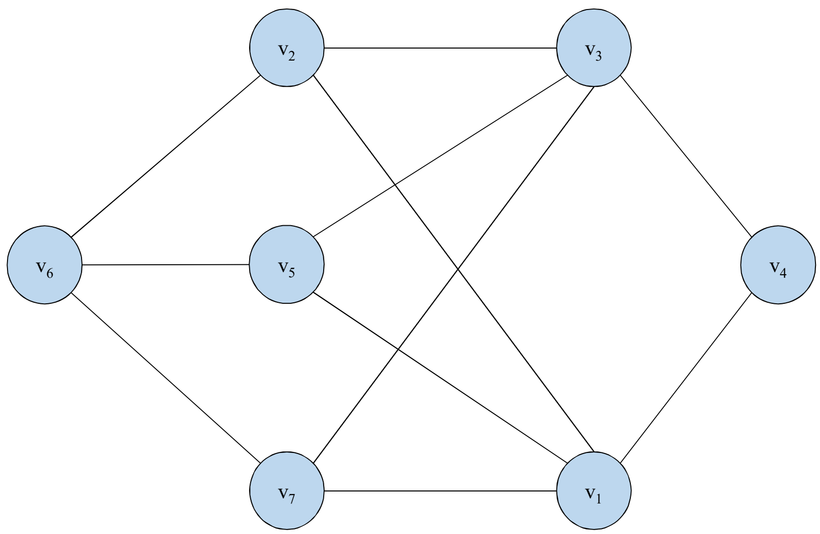
4.2 An application of BFS: Is a graph bipartite or not?

Bipartite graphs



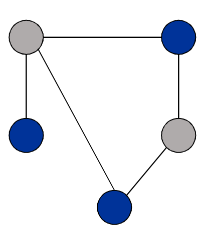
- Def. An undirected graph G = (V, E) is bipartite if the nodes can be colored red or blue such that every edge has one red and one blue end.
- Applications.
  - Employment: applicant (purple), employer (gray).
  - Scheduling: machines (purple), jobs (gray).
  - Dating: female (purple), male (gray).

Is the graph below bipartite?

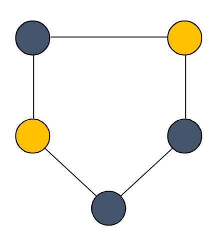


An obstruction to bipartiteness

- Lemma: If a graph G is bipartite, it **<u>cannot</u>** contain an odd length cycle.
- Pf. Not possible to 2-color the odd cycle. (See the illustration below)



Bipartite (2-colorable)



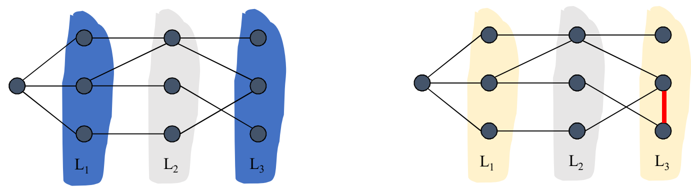
Not bipartite (not 2-colorable)

Bipartite graphs

Lemma: Let G be a connected graph, and L_0, …, L_k be the layers produced by BFS starting at node s. Exactly one of the following holds.

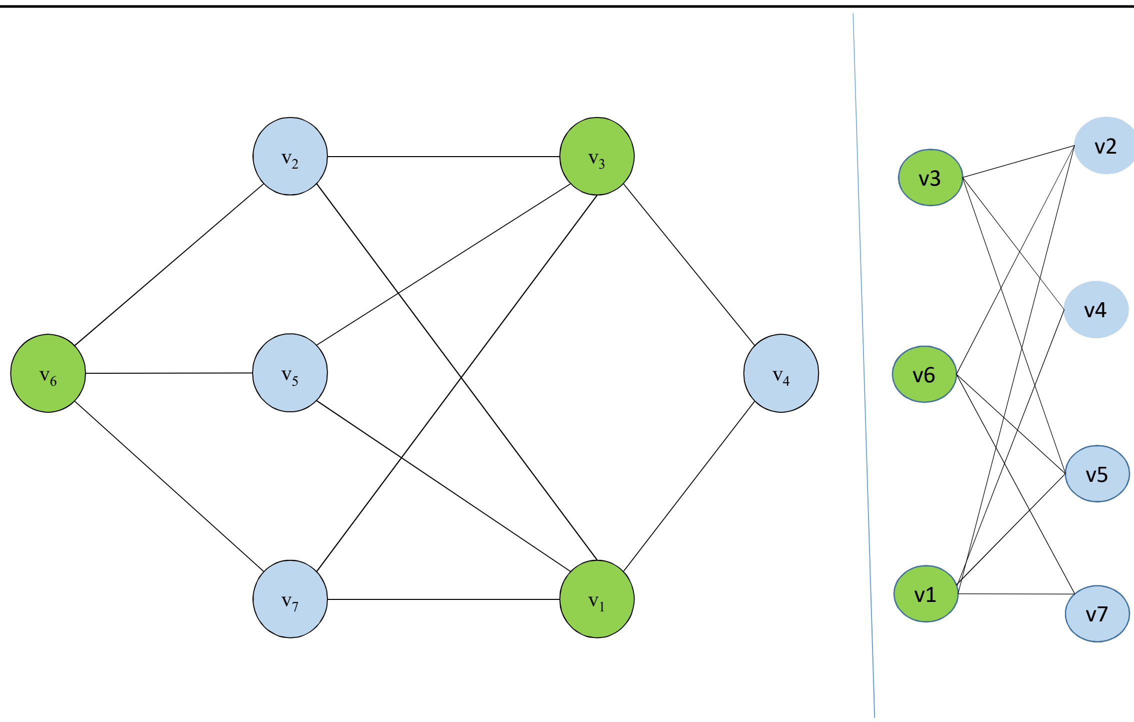
(i) No edge of G joins two nodes of the same layer, and G is bipartite.

(ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).



Case (i) Case (ii)

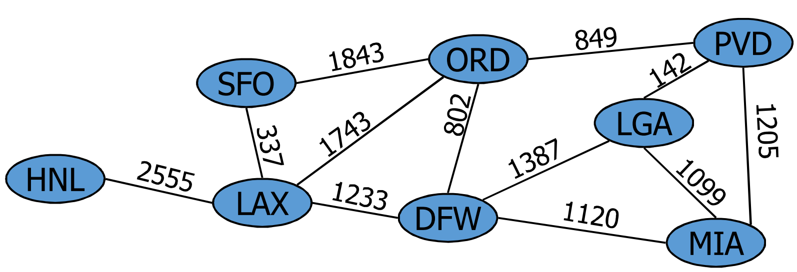
21



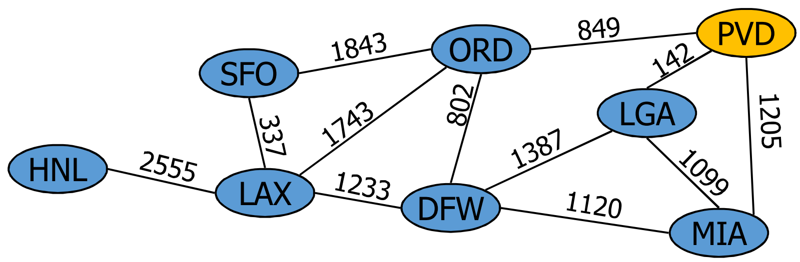
4.3 Dijkstra’s shortest paths algorithm

Weighted graphs

- Weights in a weighted graph have practical meanings such as distances, costs, etc.
- Finding paths with minimum weights between two vertices is called the shortest path problem.

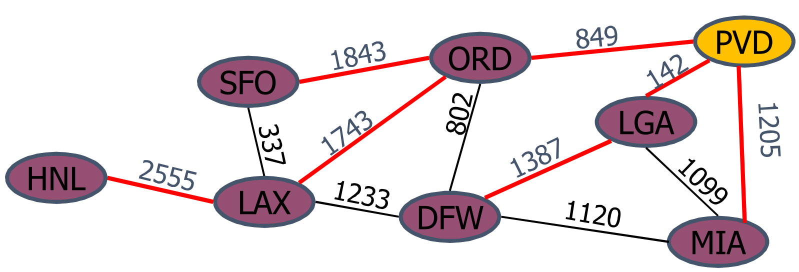


Hands on: find shortest paths from PVD to others in the graph



Work it out in few minutes to find out shortest paths from PVD to ORD, to LGA, to MIA, DFW, SFO, LAX, and HNL.

Here is the result:



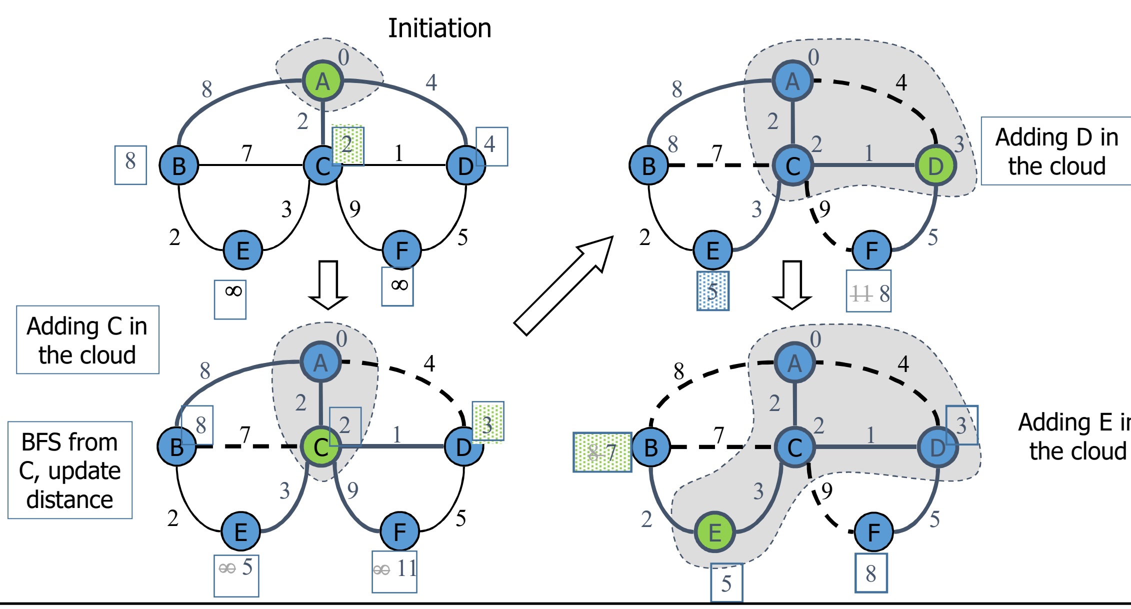
Observed properties:

1. Shortest paths from a starting vertex to all the other vertices form a tree.
2. A sub-path of a shortest path is itself a shortest path.

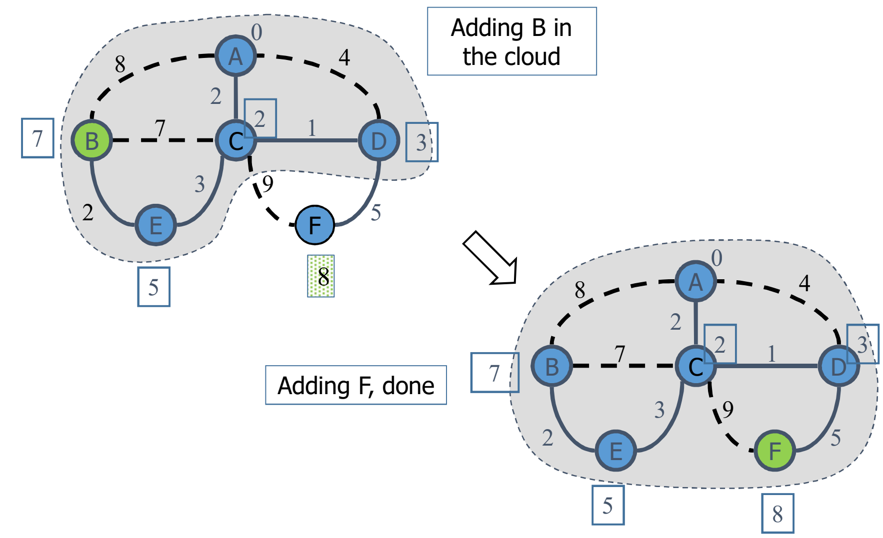
A general approach: Dijkstra's idea

- We grow a "**cloud**" of vertices, beginning with ***s*** and eventually covering all the vertices
- We store with each vertex ***v*** a label ***d***(***v***), the distance from ***s*** to ***v***, in the cloud and its adjacent vertices.
- At each step
  - We add to the cloud the vertex ***u*** outside the cloud with the smallest distance label, ***d***(***u***)
  - We update the labels of the vertices adjacent to ***u***

Dijkstra's algorithm illustrated



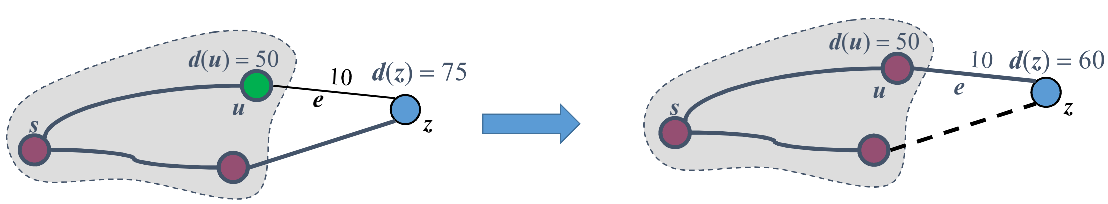
Example (cont.)



Edge relaxation

- Let ***u*** be the vertex most recently added to the cloud.
- For all edge ***e* = (*u, z*)** with ***z*** not in the cloud, updates distance ***d*(*z*)** as follows:

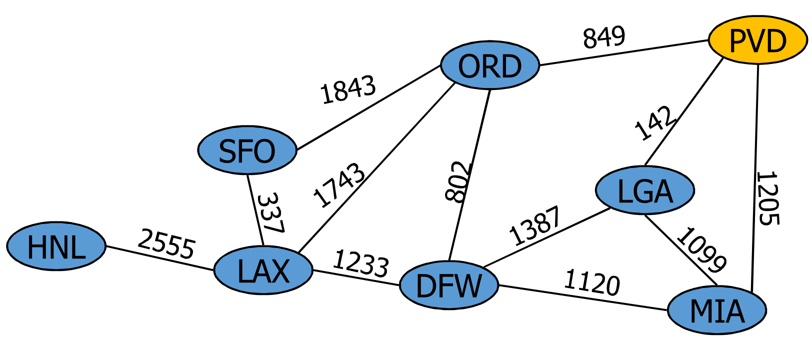
$$\boldsymbol{d(z)} \leftarrow \min\{\boldsymbol{d(z)}, \boldsymbol{d(u)} + \boldsymbol{weight(e)}\}$$



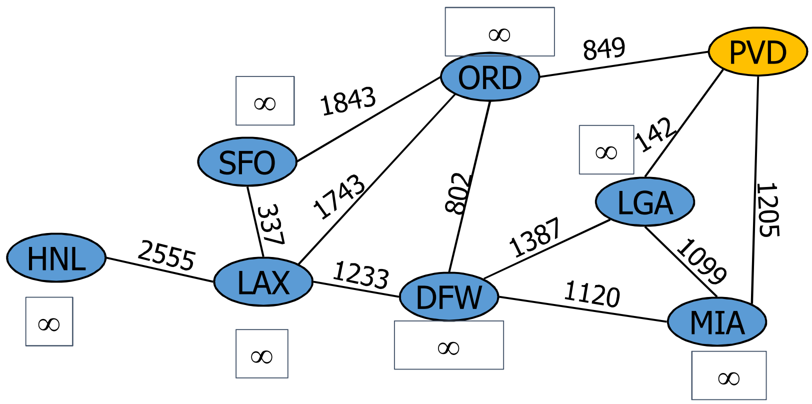
Dijkstra's idea

- Starting the cloud with the root.
- Label the distance of all other vertices as ∞.
- Perform BFS and update the labels in level 1
- While not all vertices in the cloud
  - Bringing in the vertex **u** into the cloud with ***least distance***.
  - For every edge (**u**, **z**) and **z** not in the cloud perform edge relaxation
- Dijkstra's requires that ***no edge with a negative weight***.

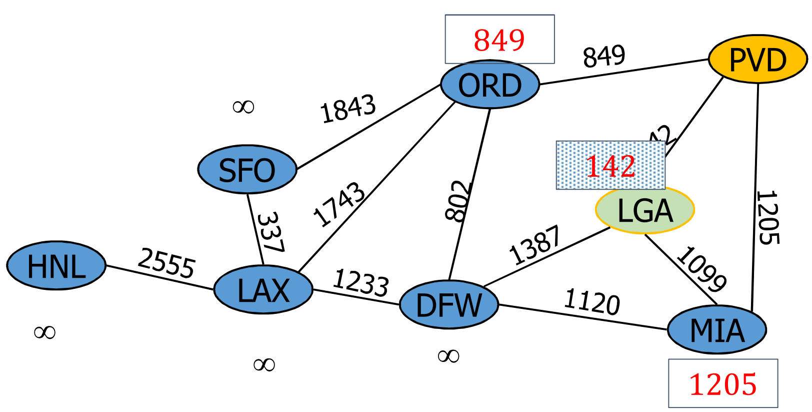
In class exercise: find shortest paths from PVD



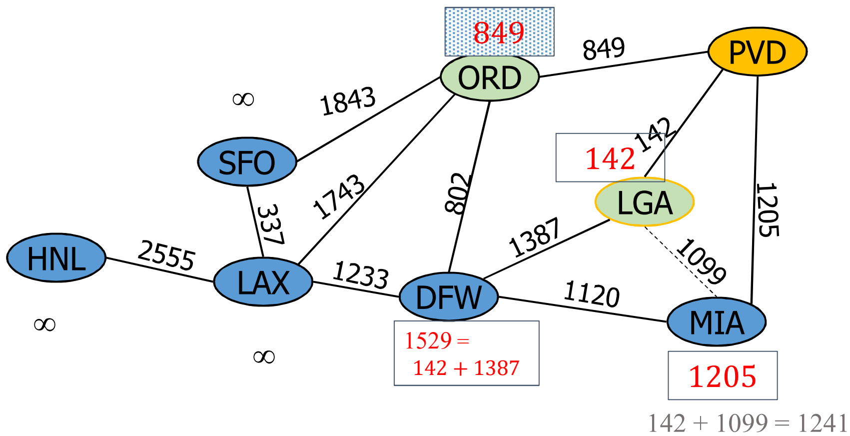
In class exercise:



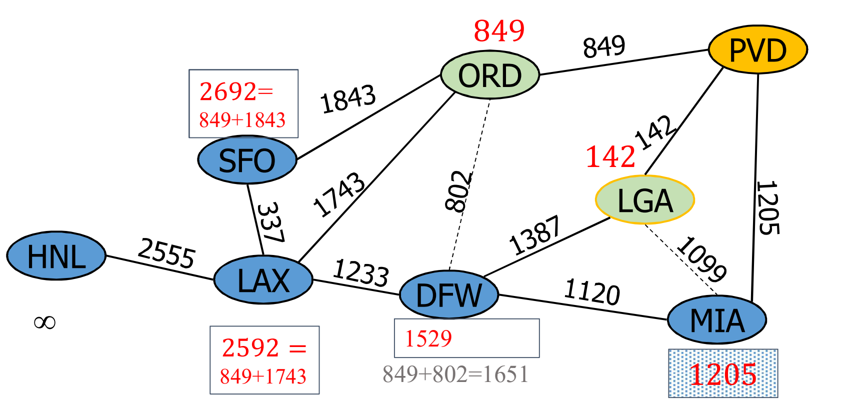
In class exercise: apply Dijkstra's algorithm to find shortest paths from PVD



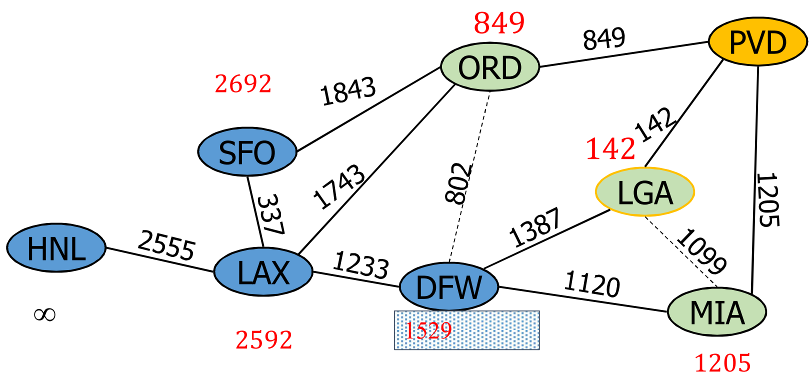
In class exercise: apply Dijkstra’s algorithm to find shortest paths from PVD



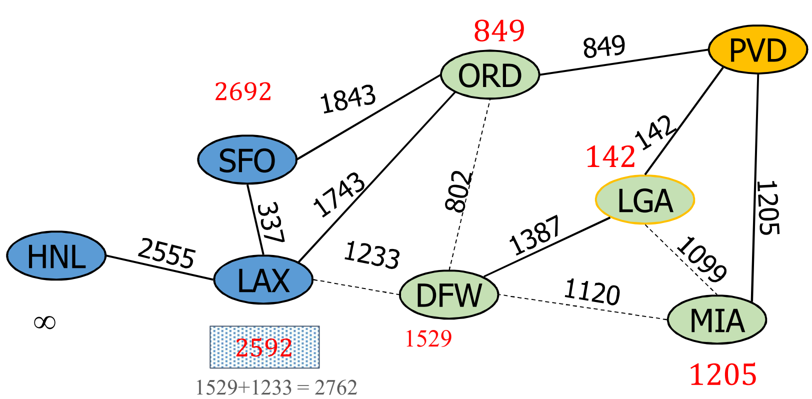
In class exercise: apply Dijkstra’s algorithm to find shortest paths from PVD



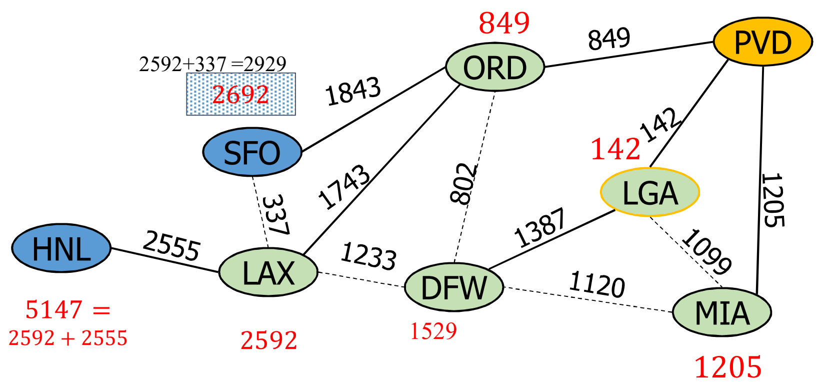
In class exercise:



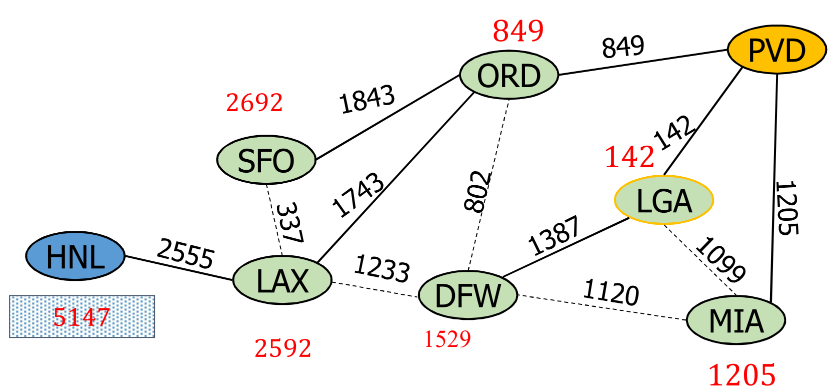
In class exercise



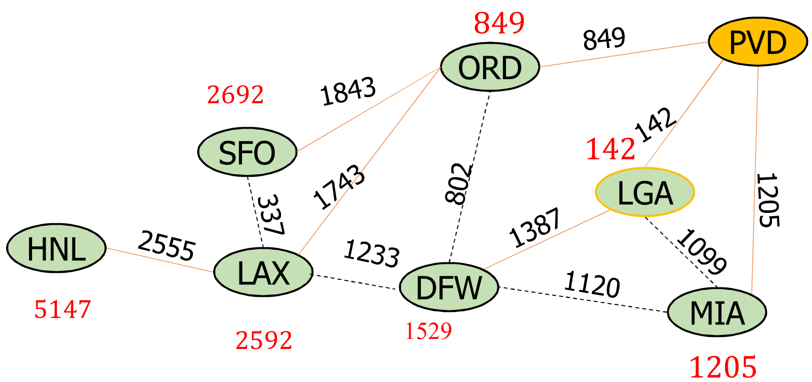
In class exercise



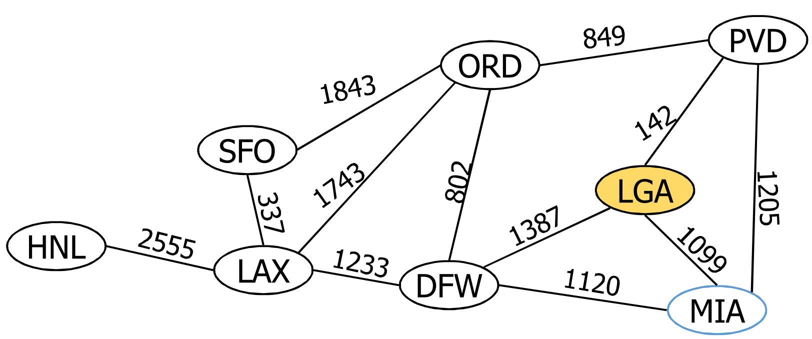
In class exercise



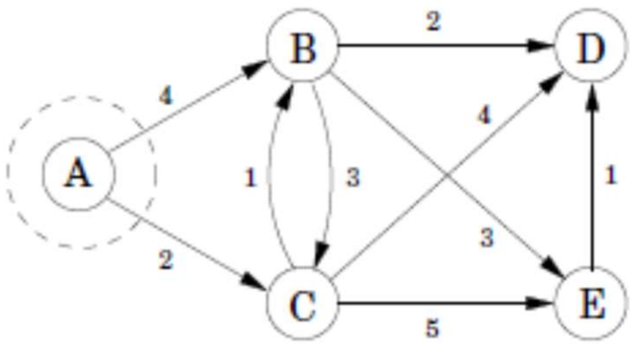
In class exercise



Exercise: find the shortest path tree from LGA



Find a shortest path tree for the digraph



Dijkstra's algorithm

Algorithm ***DijkstraShortestPath***(***G, v***)
input *A simple undirected graph G with nonnegative edge weights, and a vertex v of G*
output *A label D[u] for each vertex u of G, such that D[u] is the distance from v to u in G*
for all ***u*** $\in$ ***G.vertices***( )
  if ***u*** = ***v***
   D[u] $\leftarrow$ ***0***
  else
   D[u] $\leftarrow \infty$

Let a priority queue Q contain all vertices of G using the D labels as keys
while $\neg$***Q.isEmpty***( )
 u $\leftarrow$ ***Q.removeMin***( )

for all ***vertex z adjacent to u and z is in Q***
 do
 { perform relaxation on the edge (u,z) ***e*** }
 if ***D[u] + w(u,z) < D[z]*** **then**
  D[z] $\leftarrow$ D***[u] + w(u,z)***
return *the label D[u] of each vertex of G*

Time complexity

- Initialization for each vertex takes *O* (|V|) operations
- Priority queue operations
  - Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes O(log |V|) time
  - The key of a vertex ***w*** in the priority queue is modified at most deg(***w***) times, where each key change takes O(log |V|) time
- Dijkstra's algorithm runs in O[(|V|+|E|)(log |V|)] time provided the graph is represented by the adjacency list structure.

Code: `dijkstra.py`

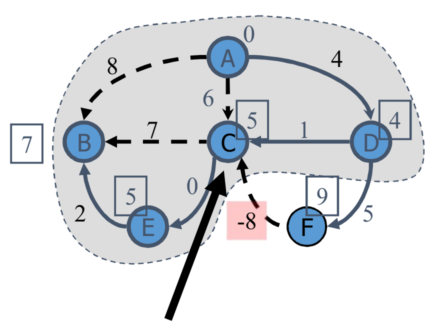
```
def dijkstra(graph, initial):
  visited = {initial: 0}
  path = {}
  nodes = set(graph.nodes)

  while nodes:
    min_node = None
    for node in nodes:
      if node in visited:
        if min_node is None:
          min_node = node
        elif visited[node] <  \
           visited[min_node]:
          min_node = node
```

```
    if min_node is None:
      break
    nodes.remove(min_node)
    current_weight = visited[min_node]

    for edge in graph.edges[min_node]:
      weight = current_weight +  \
       graph.distance[(min_node, edge)]
      if edge not in visited or \
         weight < visited[edge]:
        visited[edge] = weight
        path[edge] = min_node

  return visited, path
```

4.4 Bellman-Ford shortest path algorithm

Does Dijkstra's algorithm work for graph with a negative-weighted edge?

- No, it doesn't!
- When a node brought in the cloud through a negative incident edge, it could mess up distances of all vertices already in the cloud.



C's true distance is 1, but it is already in the cloud with d(C)=5!

Shortest paths in a digraph with negative weight

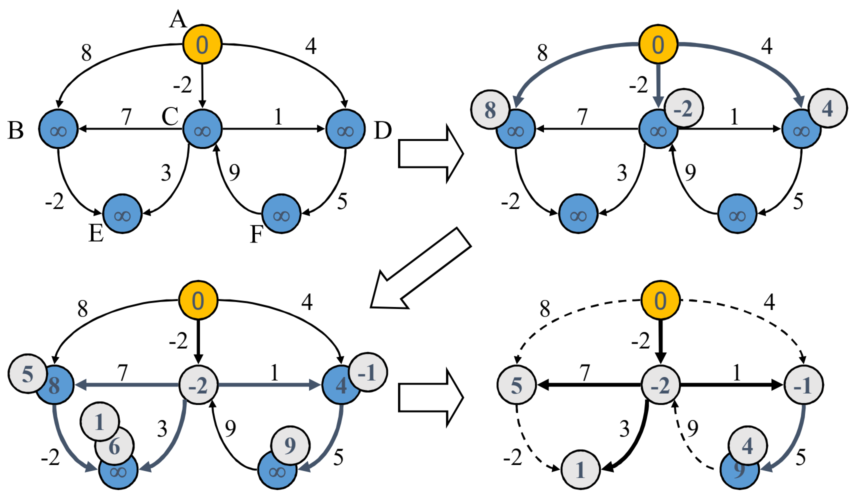
- In studying shortest paths of a graph with negative weighted edges, we need to assume the graph is directed. Otherwise, one may repeatedly use negatively weighted edges to reduce distances without a lower bound.
- Similarly, there should be no negative weighted cycles for finding shortest paths in a weighted digraph.
- **Bellman-Ford algorithm** finds shortest paths in a single source digraph with negative weighted edges but not negative cycle.

The idea of Bellman-Ford algorithm

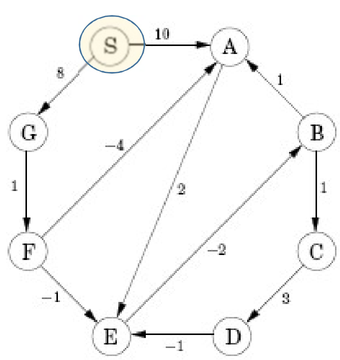
- Initialize the distance to infinity except the root as 0
- For j from 1 to |V| -1, performs relaxation to update the distances. This is because of a path from starting vertex to another has at most |V| - 1 edges
- The k-th iteration finds all shortest paths that use *k* edges.
- Running time: O(|V||E|).

An example

Nodes are labeled with their d(v) values

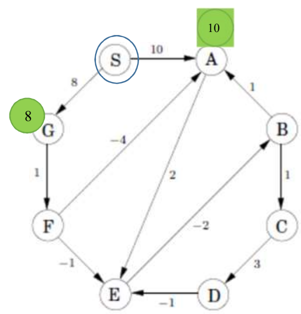


Hands on: the example on text p.124



	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

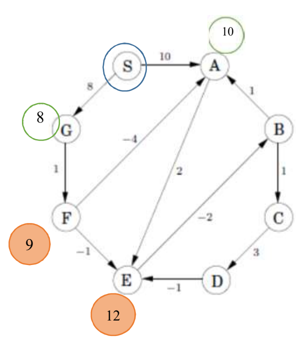
Example: continue



	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

- Length 1 walk (starting from S one step BFS): d(A) = 10, d(G) = 8

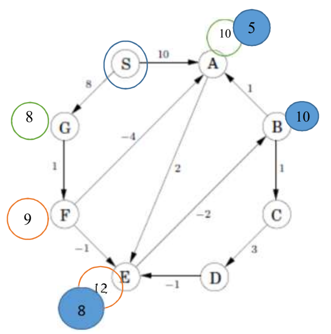
Example: continue



	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

- Length 2 walk (one step BFS starting from A and G): d(E) = 12, d(F) =9

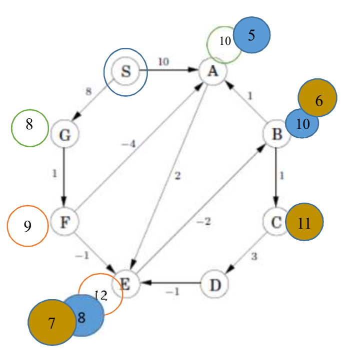
Example: continue



Node	Iteration							
	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

- Length 3 walk (one step BFS staring from E and F): d(B) = 10, update d(A) = 5, and d(E) = 8

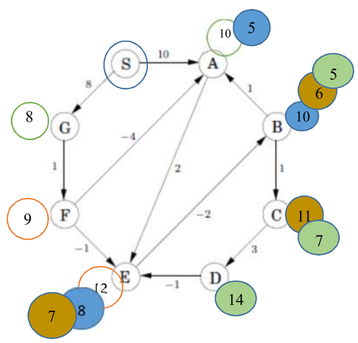
Example: continue



Node	Iteration							
	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

- Length 4 walk (one step BFS starting from A, B, E): update d(B) = 6, d(C) = 11, d(E) = 7.

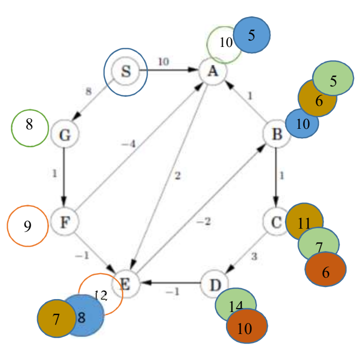
Example: continue



Node	Iteration							
	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

- Length 5 walk (one step BFS starting from B, C, E): update d(C) = 7, d(D) = 14, d(B) = 5

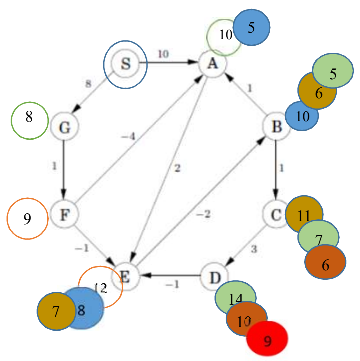
Example: continue



Node	Iteration							
	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

- Length 6 walk (starting from B, C, D): update d(C) = 6, d(D) = 10

Example: continue



	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

- Length 7 walk (starting from C, D BFS): update d(D) = 9
- Done! There are total 8 vertices.

Code: `bellman-ford.py`

```
# Referencve https://gist.github.com/joninvski/701720
def initialize(graph, source):
    d = {} # Stands for destination
    p = {} # Stands for predecessor
    for node in graph:
        d[node] = float('Inf')
        p[node] = None
    d[source] = 0 # For the source we know how to reach
    return d, p

def relax(node, neighbour, graph, d, p):
    if d[neighbour] > d[node] + graph[node][neighbour]:
        # Record this lower distance
        d[neighbour]  = d[node] + graph[node][neighbour]
        p[neighbour] = node
```

```
def bellman_ford(graph, source):
    d, p = initialize(graph, source)
    for i in range(len(graph)-1): #Run this until is converges
        for u in graph:
            for v in graph[u]: #For each neighbour of u
                relax(u, v, graph, d, p) #Lets relax it
    # check for negative-weight cycles
    for u in graph:
        for v in graph[u]:
            assert d[v] <= d[u] + graph[u][v]
    return d, p
graph = {   'a': {'b': -1, 'c':  4}, b': {'c':  3, 'd':  2, 'e':  2},  'c': {},
        'd': {'b':  1, 'c':  5},   'e': {'d': -3}}
d, p = bellman_ford(graph, 'a')
print(d)
print(p)
```

Comments on Bellman-ford algorithm

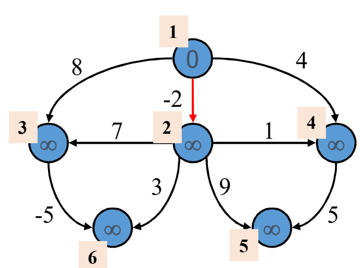
- Although Bellman-Ford algorithm works for non-negative weight graph too, its complexity O(|V||E|) is much higher than that of Dijkstra's algorithm O(|V|+|E|)log(|V|).
- Do not choose Bellman-Ford if a graph does not have a negative-weighted edge.
- Bellman-Ford does not work when a digraph contains a negative cycle.

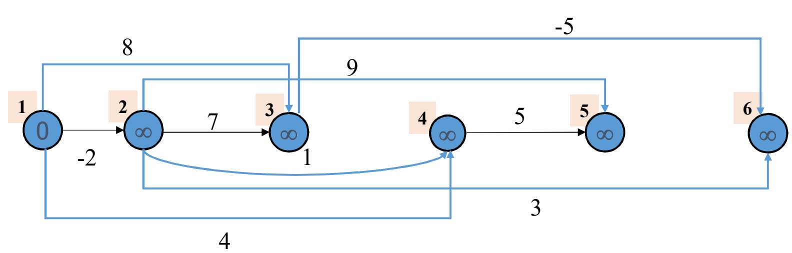
4.5 Shortest paths in DAGs

Properties of a dag

- A dag contains no cycles at all.
- The vertices of a dag can be placed in its topological order.
- So, we can find shortest paths of a dag after linearizing it.

Hands on: Solving it in its linearized setting





Shortest paths in DAGs

- Input: A weighted directed acyclic graph (DAG), *G*, with *n* vertices and *m* edges, and a distinguished vertex *s*
- A label *D*[*u*], for each vertex *u* in *G* such that *D[u]* is the distance from *s* to *u*
- Initialization: $D[s] \leftarrow 0$, $D[u] \leftarrow \infty$ for $u \neq s$
- Linearize G in its topological order
- Updating:

  for each $u \in V$, in its topological order

  for each edge $(u, v) \in E$

  $D[v] = \min\{ D[v], D[u] + w(u, v) \}$
- We are going to study it in detail after chapter 5.

Summary

- BFS vs. DFS
- Determine if a graph is bipartite with BFS
- Dijkstra's shortest path algorithm: its limitation and complexity
- Bellman-Ford algorithm: its limitation and complexity
- Shortest paths for a dag through linearization.