Lecture Notes on
Data Structures

M1522.000900

Seoul National University

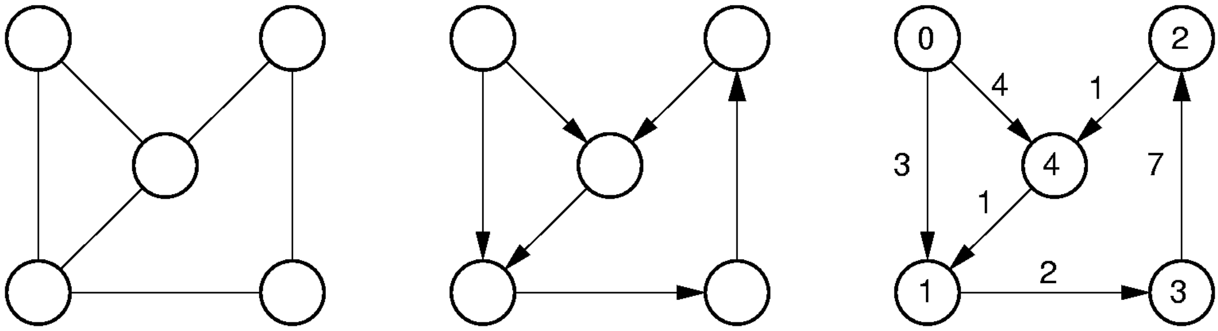
Fall 2022

Part VI

Graph

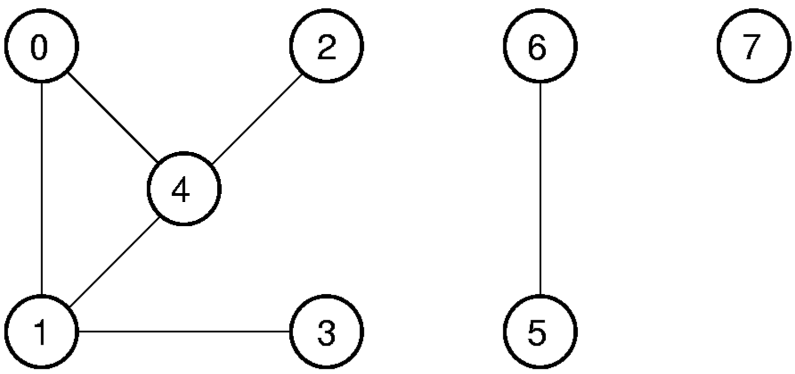
Graph: Definitions

- A graph: $G = (V, E)$, where V is a set of vertices and E is a set of edges. If $e \in E$, then there exist u and v such that $e = \overline{uv}$, and $u, v \in V$.
- A directed graph: $G = (V, E)$, where each edge has a direction; $\vec{uv} \neq \vec{vu}$.
- A weighted graph: $G = (V, E, W)$, where each edge has a weight $w \in W$.



(a) (b) (c)

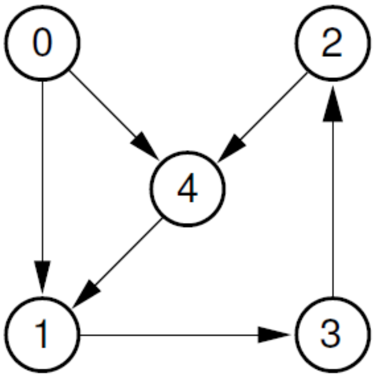
- A graph $G = (V, E)$ is connected iff $\forall u, v \in V, \exists$ a path from u to v.
- A directed graph $G = (V, E)$ is strongly connected iff $\forall u, v \in V, \exists$ a path from u to v, and vice versa (*i.e.*, mutually reachable).



Graph Implementations

- Adjacency Matrix: for a graph $G = (V, E)$, use a $|V| \times |V|$ matrix M such that

$$M[i,j] = \begin{cases} 1 & \text{if } \overline{v_i v_j} \in E \text{ (or } \vec{v_i v_j} \in E) \\ 0 & \text{otherwise.} \end{cases}$$

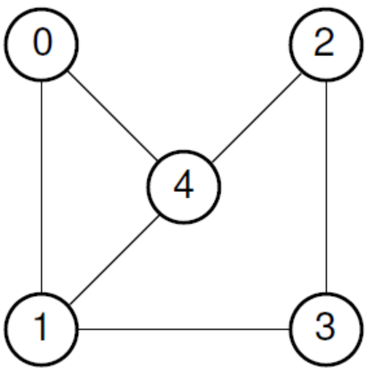


(a)

	0	1	2	3	4
0		1			1
1				1	
2					1
3			1		
4		1			

(b)

- If a graph is undirected, the matrix M is symmetric.

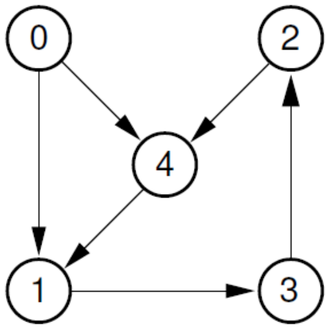


(a)

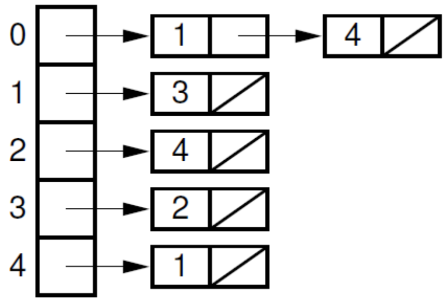
	0	1	2	3	4
0		1			1
1	1			1	1
2				1	1
3		1	1		
4	1	1	1		

(b)

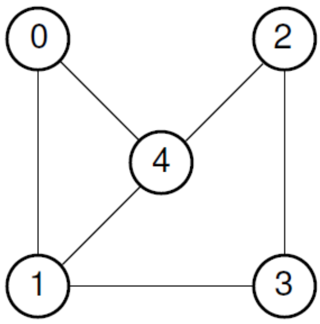
- Adjacency List: for each vertex, create a list of neighbors.
  1. A directory of vertices
  2. A list of adjacent vertices per each vertex



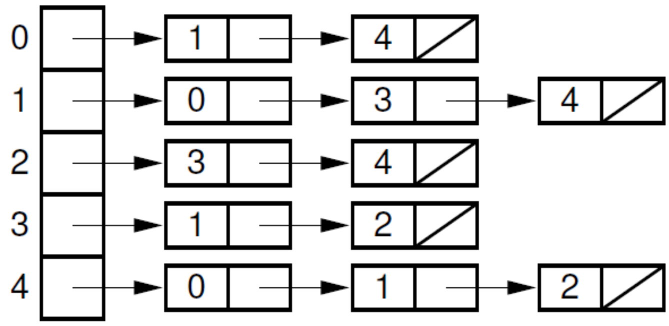
(a)



(c)



(a)



(c)

Adjacency Matrix vs. Adjacency List

	Adjacency Matrix	Adjacency List
Space requirement	$\Theta(\|V\|^2)$	$\Theta(\|V\| + \|E\|)$
Is $\overline{uv} \in E$?	$\mathcal{O}(1)$	$\mathcal{O}(degree(v))$
Print all neighbors of v	$\Theta(\|V\|)$	$\Theta(degree(v))$
Print all edges $\in E$	$\Theta(\|V\|^2)$	$\Theta(\|V\| + \|E\|)$

Graph Traversal

Visit each and every vertex in a graph G exactly once.

- There is no root. Any vertex can be a starting point.
- Order is determined by the topology of a graph, but may not be unique.

There are two difficulties in the graph traversal.

- A graph may be disconnected.
- A graph may contain circles.

Graph Traversal: DFS

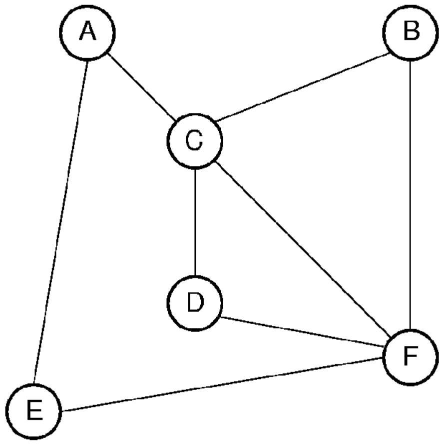
Algorithm 1 (Depth First Search)

```
DFS(G,v)
   print v;
   for each u such that uv ∈ E
      if (u is not visited) DFS(G,u);
```

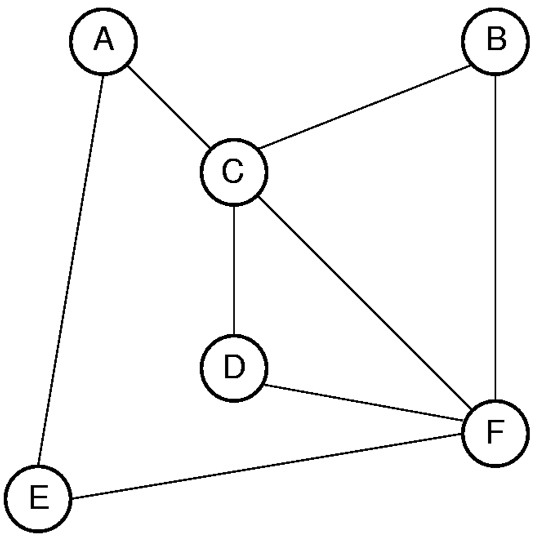
- Whenever a vertex v is visited, visit all of its *unvisited* neighbors *recursively*.
- A stack is used implicitly by recursion.

Example 1

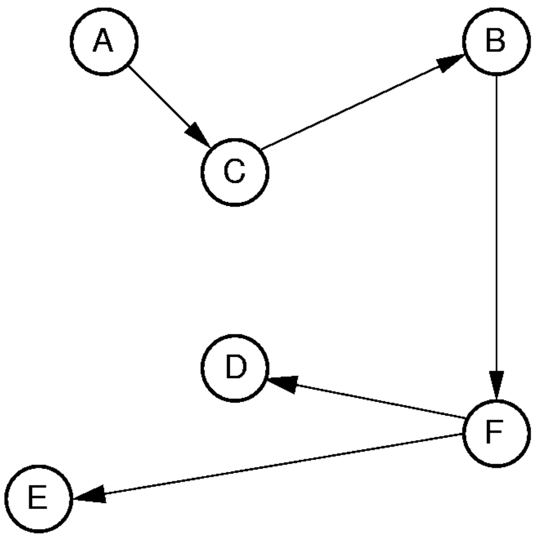
Perform DFS for a graph given below, starting from vertex A.




```
DFS(A)
   print A
   DFS (C)
      print C
      DFS(B)
         print B
         DFS(F)
            print F
            DFS(D)
               print D
            DFS(E)
               print E
      DFS(D)??
      DFS(F)??
   DFS(E)??
```

(a)



(b)

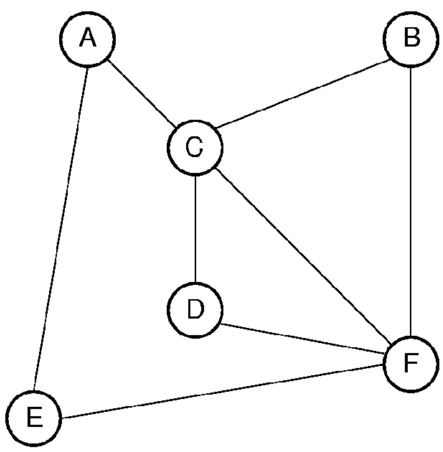
Algorithm 2 (Breadth First Search)

```
BFS(G,s)
   print s;                                   // s is visited
   Enqueue(Q,s);
   while (Q is not empty) {
      v = Dequeue(Q);
      for each u such that uv ∈ E
         if (u is not visited) {
            print u;                          // u is visited
            Enqueue(Q,u);
         }
   }
```

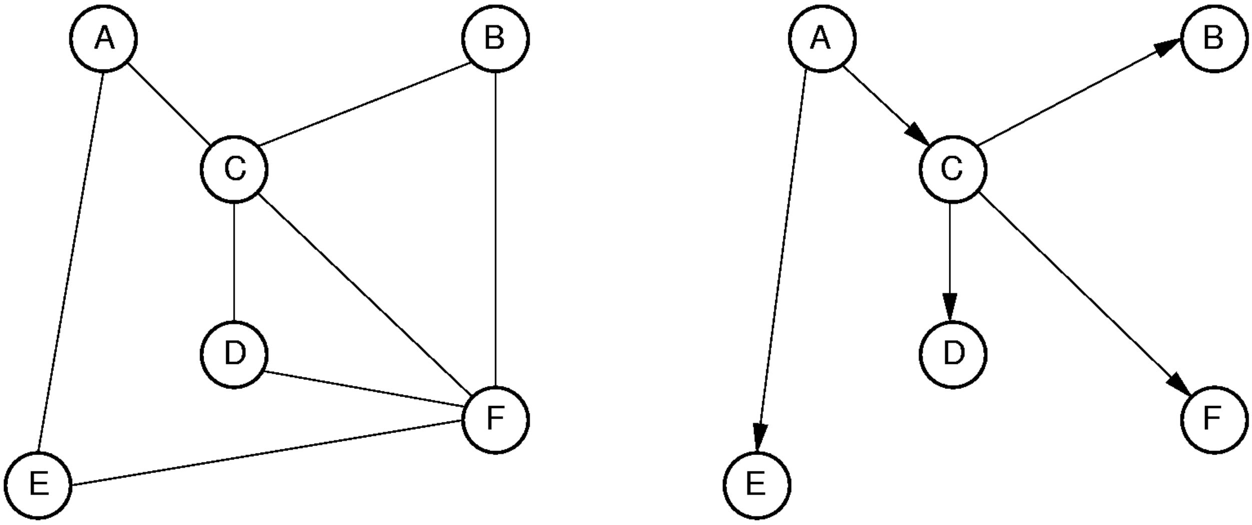
- BFS is an iterative algorithm. A queue is used explicitly.
- Invariant: all vertices in the queue are already visited.

Example 2

Perform BFS for the following graph starting from vertex A.



1. Visit the start vertex s.
2. Then, visit all vertices adjacent to s.
3. Then, visit all vertices two edges away from s.
4. Then, visit all vertices three edges away from s.
5. ...



(a) (b)

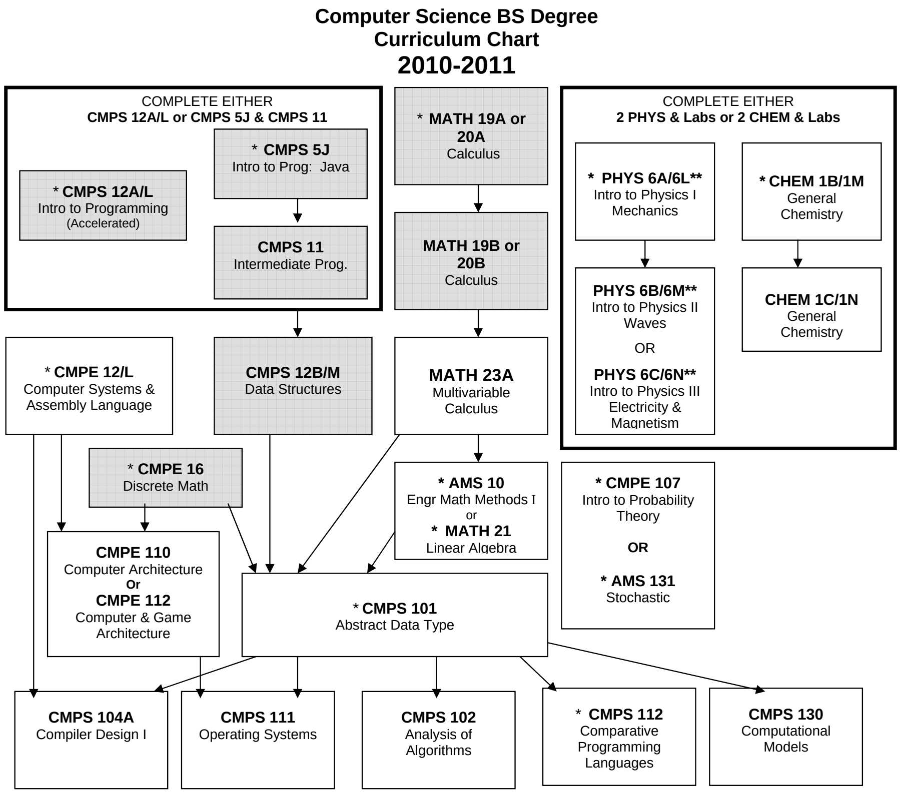
Analysis of Graph Traversals

- DFS visits each vertex once and processes each edge twice (or once if directed).
- BFS enqueues (or dequeues) each vertex once and processes each edge twice (or once if directed).
- The running time is $\mathcal{O}(|V| + |E|)$ if an adjacency list is used, because

$$\sum_{v \in V} degree(v) = 2 \times |E|.$$

- The running time is $\mathcal{O}(|V|^2)$ if an adjacency matrix is used.
- The size of a stack or a queue is $\mathcal{O}(|V|)$.

- Given a *directed acyclic graph (DAG)* G, find a linear ordering of vertices of G such that
  - if $\vec{uv} \in E$, then u appears before v in the ordering.
- Topological order does not exist for a cyclic graph.
- Note that the edges represent precedence or prerequisites. For example, activity-on-vertex (AOV) networks in the project management.



Courtesy of University of California, Santa Cruz

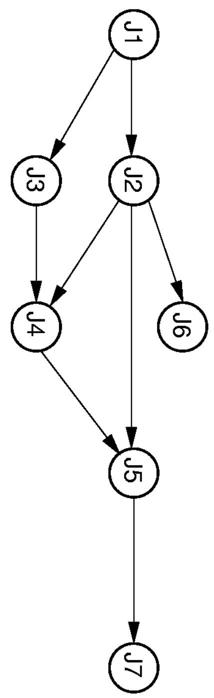
Algorithm 3 (Topological Sort)

```
TopoSort(G)
   for each v ∈ V, compute indegree(v);
   for each v ∈ V
      if (indegree(v)==0) { print v; Enqueue(Q,v); }
   while (Q is not empty) {
      v = Dequeue(Q);
      for each u such that vu⃗ ∈ E {
         indegree(u)--;                    // vu⃗ is removed from G
         if (indegree(u)==0) { print u; Enqueue(Q,u); }
      }
   }
```

- Invariant: all vertices in the queue are already visited.

Example 3

Find a topological ordering for the following DAG.



indegree

J_1	J_2	J_3	J_4	J_5	J_6	J_7	Queue
0	1	1	2	2	1	1	J_1
0	0	0	2	2	1	1	J_2, J_3
0	0	0	1	1	0	1	J_3, J_6
0	0	0	0	1	0	1	J_6, J_4
0	0	0	0	1	0	1	J_4
0	0	0	0	0	0	1	J_5
0	0	0	0	0	0	0	J_7

$$J_1 \to J_2 \to J_3 \to J_6 \to J_4 \to J_5 \to J_7$$

Problem 1

Add an edge $J_7 \to J_4$ to the DAG of Example 3 and run Algorithm 3 (Topological Sort) for the modified graph. Does Algorithm 3 terminates? If it does, does it produce a correct topological order?

Problem 2

Add an edge $J_4 \to J_6$ to the DAG of Example 3 and answer the same questions.

Analysis of Topological Sort

	Adjacency List	Adjacency Matrix
The 1st for-loop	$\mathcal{O}(\lvert E \rvert)$	$\mathcal{O}(\lvert V \rvert^2)$
The 2nd for-loop	$\mathcal{O}(\lvert V \rvert)$	$\mathcal{O}(\lvert V \rvert)$
The while-loop	$\mathcal{O}(\lvert E \rvert)$	$\mathcal{O}(\lvert V \rvert^2)$

- The running time of Topological Sort is $\mathcal{O}(|V| + |E|)$ (or $\mathcal{O}(|V|^2)$).

Shortest Paths: Problems

Single-Pair: Given a pair of vertices s (start) and t (end), find the shortest path from s to t.

Single-Source: Given a start vertex s, find the shortest paths from s to all the other vertices.

All-Pair: For each pair of vertices $u, v \in E$, find the shortest path from u to v.

☞ The three problems will be addressed by two algorithms: Dijkstra's and Floyd's.

Single-Source Shortest Paths

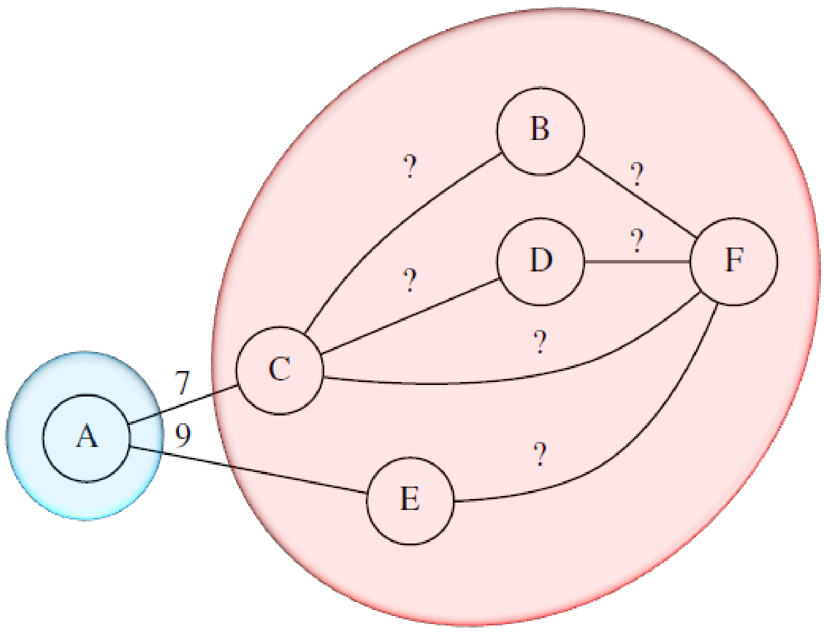
- If all edges have the same weight, BFS can find all S.S.S.P.
  - All vertices $k+1$ edges away from the source are visited after all vertices k edges away from the source are visited.
  - In the tree resulting from the BFS traversal, the source is the root node and the level of a non-root node (or vertex) is the distance from the source vertex.
- What about DFS?
  - A vertex close to the source may be visited later than another vertex farther from the source.
- Dijkstra's algorithm is a *weighted* version of BFS.

Dijkstra's Algorithm for Single-Source [1959]

- All weights are assumed to be non-negative.
- Find the shortest path for vertices in the increasing order of distance (rather than "one edge away, then two edges away, then ...").
  - If the shortest paths are found for $v_1, v_2, \ldots, v_{i-1}$ in the given order, then $d(s, v_1) \leq d(s, v_2) \leq \ldots \leq d(s, v_{i-1})$.
- Suppose the shortest paths have been found for $S = \{s, v_1, v_2, \ldots, v_{i-1}\}$, and v_i is the next one for which SP will be found. Then, v_i is one of the *direct* neighbors of the vertices in S.
  - $d(s, v_i) = min_{v_j \in S}\{d(s, v_j) + w_{ji}\}$
  - See Theorem 5.
  - v_1 is a neighbor of s.
  - v_2 is a neighbor of s or v_1.
  - v_3 is a neighbor of s, v_1 or v_2.
  - ...

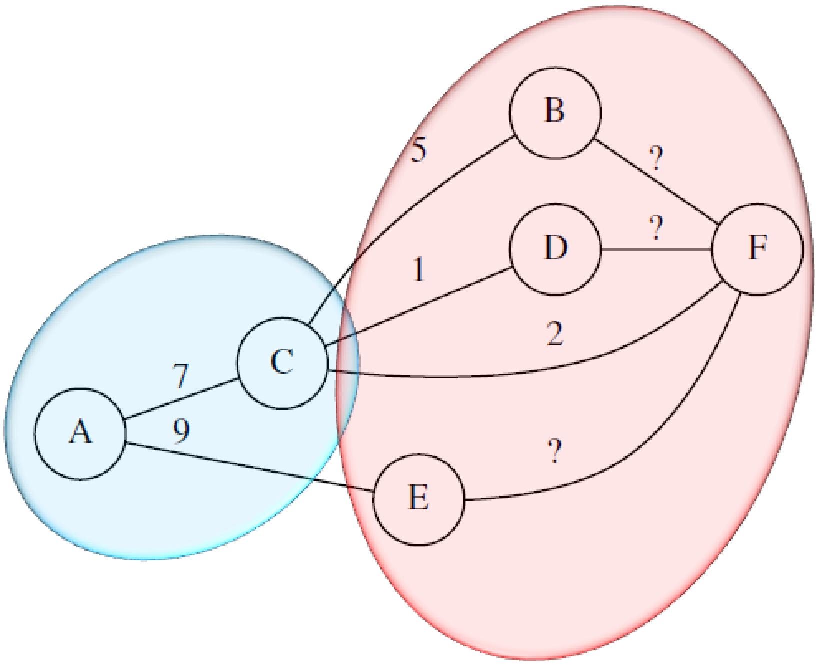
Are the following statements true or false?

- The shortest distance from A to C is 7.
- The shortest distance from A to E is 9.



Are the following statements true or false?

- The shortest distance from A to D is 8.
- The shortest distance from A to E (or F) is 9.



Algorithm 4 (Dijkstra's Single-Source Shortest Paths)

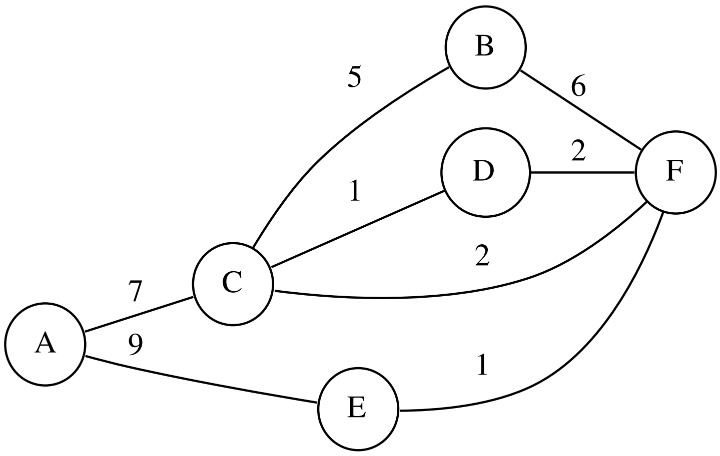
// Assume s is the source vertex.
// Initially, $S = \{s\}$ and d[s] = 0.

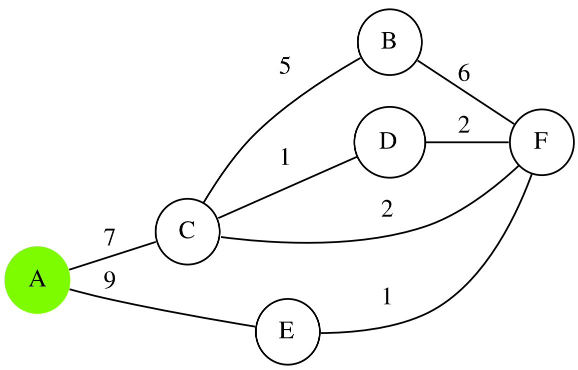
for each $v \in V - S$, d[v] = $\begin{cases} w_{sv} & \text{if } \overline{sv} \in E \\ \infty & \text{otherwise} \end{cases}$

while($V - S \neq \emptyset$) {
  find $v \in V - S$ such that $d[v]$ is minimum;
      // v is among the vertices on the fringe of S.
  print d[v];   // Shortest path to v found.
  $S = S \cup \{v\}$;
  for each fringe $u \in V - S$ such that $\overline{vu} \in E$
    if $(d[v] + w_{vu} < d[u])$ $d[u] = d[v] + w_{vu}$;
}

Example 4

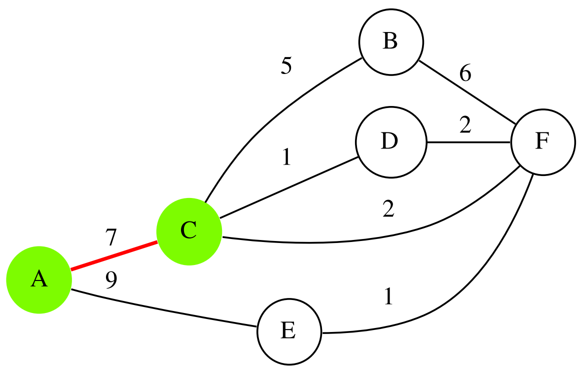
Find the shortest paths from vertex A to all the other vertices.





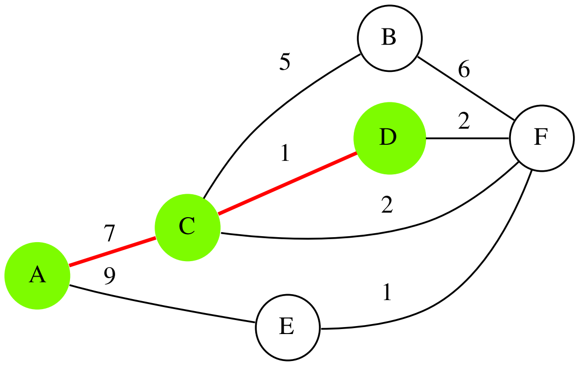
v	A	B	C	D	E	F
d	0	∞	7	∞	9	∞

$$\begin{aligned} S &= \{A\} \\ V - S &= \{B, C, D, E, F\} \end{aligned}$$



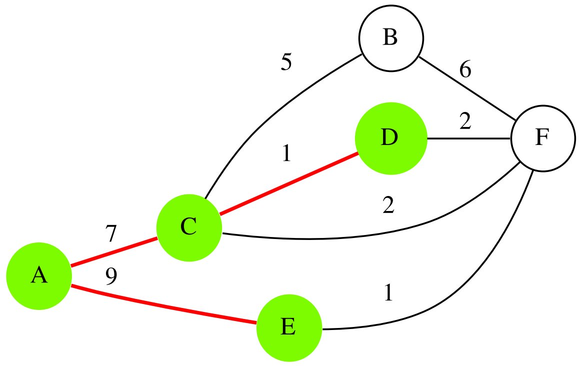
v	A	B	C	D	E	F
d	0	12	7	8	9	9

$$\begin{aligned} S &= \{A, C\} \\ V - S &= \{B, D, E, F\} \end{aligned}$$



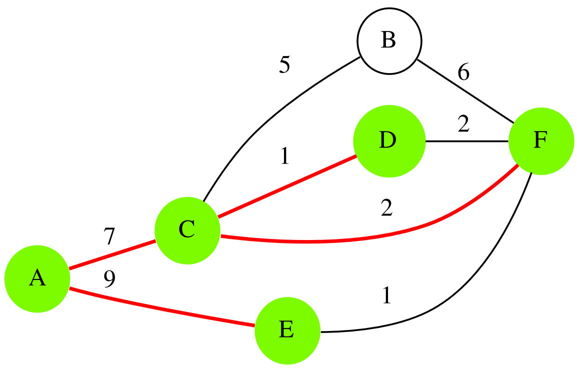
v	A	B	C	D	E	F
d	0	12	7	8	9	9

$$\begin{aligned} S &= \{A, C, D\} \\ V - S &= \{B, E, F\} \end{aligned}$$



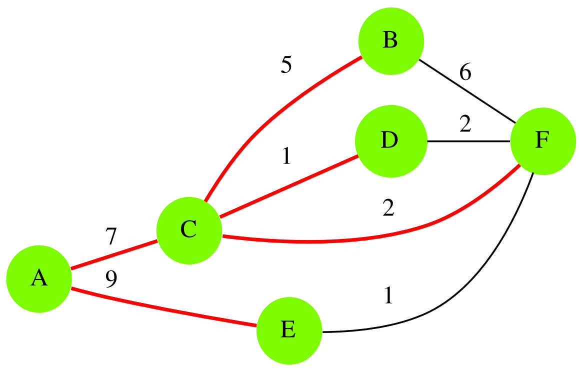
v	A	B	C	D	E	F
d	0	12	7	8	9	9

$$\begin{aligned} S &= \{A, C, D, E\} \\ V - S &= \{B, F\} \end{aligned}$$



v	A	B	C	D	E	F
d	0	12	7	8	9	9

$$\begin{aligned} S &= \{A, C, D, E, F\} \\ V - S &= \{B\} \end{aligned}$$



v	A	B	C	D	E	F
d	0	12	7	8	9	9

$$\begin{aligned} S &= \{A, B, C, D, E, F\} \\ V - S &= \{\} \end{aligned}$$

Analysis of Dijkstra's Algorithm

The running time of Dijkstra's algorithm depends on how `d[]` values are maintained.

If `d[]` values are stored in an (unsorted) array,

- $\mathcal{O}(|V| \times |V|)$ for finding (and removing) the minimum `d[]` value,
- $\mathcal{O}(|E|)$ for updating `d[]` values.

Therefore, the running time is $\mathcal{O}(|V|^2)$ because $|E| \le |V|^2$.

If `d[]` values are stored in a min-heap H,

- Finding and removing the minimum `d[]` value,
  - $\mathcal{O}(\log|H|)$ with a min-heap.
  - Repeated $\mathcal{O}(|V|)$ times.
- Updating `d[u]` by deleting its old value and inserting a new one.
  - $\mathcal{O}(|H|)$ for deleting an old `d[]` value.
  - $\mathcal{O}(\log|H|)$ for inserting a new `d[]` value.
  - Repeated $\mathcal{O}(|E|)$ times.

The size of the min-heap $|H|$ is $\mathcal{O}(|V|)$. Thus, the running time of Dijkstra's algorithm is

$$\mathcal{O}(|V| \times \log|V| + |E| \times |V|) = \mathcal{O}(|E| \times |V|).$$

<u>Worse than $\mathcal{O}(|V|^2)$!!</u>

If *the step for "deleting old* `d[]` *values" is omitted*,

- Finding and removing the minimum `d[]` value,
  - $\mathcal{O}(\log |H|)$ with a min-heap.
  - Repeated $\mathcal{O}(max\{|V|, |H|\})$ (or $\mathcal{O}(|E|)$) times.
- Inserting a new `d[u]` value without deleting its old one.
  - $\mathcal{O}(\log |H|)$ for inserting a new `d[]` value.
  - Repeated $\mathcal{O}(|E|)$ times.

Now, $|H|$ can grow as large as $|E|$ (*i.e.*, $|H| \in \mathcal{O}(|E|)$ instead of $\mathcal{O}(|V|)$). Thus, the running time will be

$$\mathcal{O}(|E| \times \log |E| + |E| \times \log |E|) = \mathcal{O}(|E| \log |E|).$$

This is better than $\mathcal{O}(|V|^2)$ (and $\mathcal{O}(|E| \times |V|)$) if G is sparse ($|E| \approx |V|$).

Other choices for storing the `d[]` values:

- With a heap with locator, $\mathcal{O}(|V| \log |V|) + \mathcal{O}(|E| \log |V|)$.
- With a Fibonacci heap, $\mathcal{O}(|V| \log |V| + |E|)$.

Correctness of Dijkstra's Shortest Path Algorithm

- Once a vertex (say v) is visited or added to the set S, Dijkstra's algorithm declares that a shortest path to v has been found. From this point on, its distance (*i.e.*, `d[v]`) never gets updated any more.
- We need to show that $\mathtt{d}[\mathtt{v}] = d(s, v)$ when v is visited.
- Well, a few lemmas first ...

Lemma 1

A subpath of a shortest path is a shortest path. If $v_1, v_2, v_3, \ldots, v_k$ is a shortest path from v_1 to v_k, then a subpath $v_i, v_{i+1}, \ldots, v_j$ is a shortest path from v_i to v_j for any i and j such that $1 \leq i \leq j \leq k$.

Proof. (By contradiction) Suppose $v_i, v_{i+1}, \ldots, v_j$ is not a shortest path. Then, there exists a shorter path $v_i \rightsquigarrow v_j$. Therefore $v_1, v_2, v_3, \ldots, v_k$ cannot a shortest path from v_1 to v_k, because it will be longer than $v_1, v_2, \ldots v_i \rightsquigarrow v_j, v_{j+1}, \ldots, v_k$. □

Corollary 2

For any vertex v on a shortest path $s \rightsquigarrow u$, $d(s, v) + d(v, u) = d(s, u)$.

Lemma 3

For any edge $\overline{uv} \in E$, $d(s,v) \leq d(s,u) + w_{uv}$.

Proof. (By contradiction) Let $s \rightsquigarrow v$ a shortest path from s to v whose length is $d(s,v)$. Suppose $d(s,v) > d(s,u) + w_{uv}$. Then, $s \rightsquigarrow u \rightarrow v$ is a path from s to v shorter than $s \rightsquigarrow v$. □

Lemma 4

If $v_1 \rightsquigarrow v_{k-1}, v_k$ is a shortest path from v_1 to v_k, then

$$d(v_1, v_k) = d(v_1, v_{k-1}) + w_{v_{k-1}v_k}.$$

Proof. (By contradiction) Since $v_1 \rightsquigarrow v_{k-1}, v_k$ is a shortest path, $d(v_1, v_k) \leq d(v_1, v_{k-1}) + w_{v_{k-1}v_k}$ by Lemma 3. Suppose $d(v_1, v_k) < d(v_1, v_{k-1}) + w_{v_{k-1}v_k}$. Then, there exists a path $v_1 \rightsquigarrow v_k$ shorter than a shortest path $v_1 \rightsquigarrow v_{k-1}, v_k$. □

☞ This Lemma can also be derived from Corollary 2.

Theorem 5

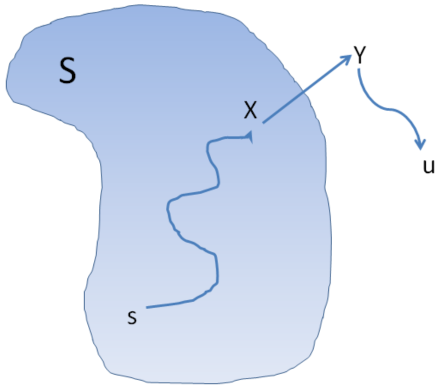
When a vertex v is pulled into the set S by Dijkstra's algorithm,

$$d[v] = d(s, v),$$

where $d(s, v)$ is the length of a shortest path from s to v.

<u>*Proof.*</u> (By contradiction) Suppose u is the **first** vertex pulled to S such that $d[u] > d(s, u)$.

From $d[u] > d(s, u)$, we know that $d(s, u) < \infty$ and there exists a shortest path from s to u. Consider the moment right before u is pulled into S. Let x and y be the vertices on the shortest path $(s \rightsquigarrow u)$ such that $x \in S, y \notin S$, and $\overline{xy} \in E$ is on the shortest path.



$$\begin{aligned} d[x] &= d(s, x) && x \in S \text{ and } u \text{ is the first incorrect one} \\ d[y] &\leq d[x] + w_{xy} && \text{invariant right after } x \text{ is pulled to } S \ (y \notin S) \\ d(s, y) &= d(s, x) + w_{xy} && \text{by Lemma 4} \end{aligned}$$

From the equations above, it follows that $d[y] \leq d(s, x) + w_{xy} = d(s, y)$. Therefore, $d[y] = d(s, y)$.

$$\begin{aligned} d(s, y) &\leq d(s, u) && y \text{ occurs before } u \text{ on the shortest path} \\ d[u] &\leq d[y] && \text{from that } u\text{, not } y\text{, is pulled into } S \end{aligned}$$

Then, from the three equations above,

$$d[u] \leq d[y] = d(s, y) \leq d(s, u).$$

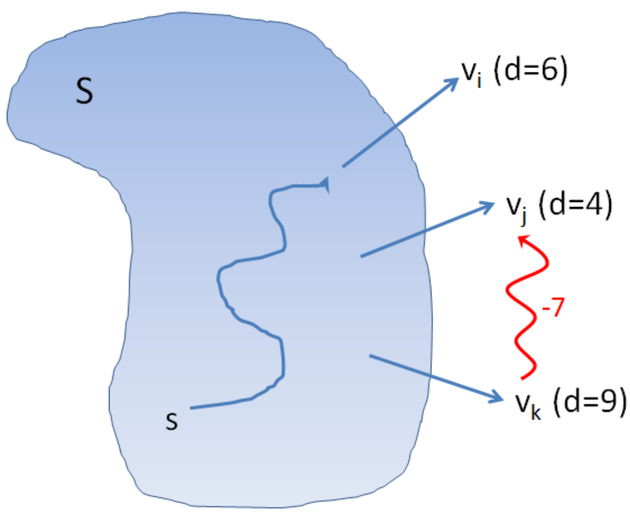
This is a contraction to the assumption that $d[u] > d(s, u)$. □

Problem 3

Algorithm 4 (Dijkstra's shortest path) just computes the length of shortest paths. Modify it to print actual paths.

HINT: When a vertex v is pulled to S, whatever vertex updated d[v] most recently is the previous vertex on the path from the source to v.

Negative weights



- Dijkstra's algorithm determines that a shortest path to v_j has been found and its length is 4.
- There is a shorter path $s \rightsquigarrow v_k \rightsquigarrow v_j$ whose length is $9 + (-7) = 2$.
- This is because Dijkstra's algorithm only considers the fringe edges between S and $V - S$.

Remark 1

In an undirected graph, even a single negative-weight edge constitutes a negative-weight cycle.

Algorithm 5 (Bellman-Ford's Single-Source Shortest Paths)

```
// Assume n vertices are indexed from 0 to n-1; v_0 = s.
for(k=0; k < n ;k++)                       // initialize d_0(s,v).
   d[k] = { 0   if k = 0
          { ∞   otherwise
for(k=1; k < n ;k++) {                     // compute d_k(s,v).
   for each v_i v_j ∈ E
      if (d[i] + w_ij < d[j]) d[j] = d[i] + w_ij;
}
```

When negative weights are allowed, it will not be enough to consider edges between S and $V - S$. All edges must be considered.

- There is no set S which vertices are pulled into.
- Every edge is processed $|V| - 1$ times to update `d[]` values.

Theorem 6

At the end of the execution of the Bellman-Ford algorithm (Algorithm 5),

$$d[v] = d(s, v)$$

for each vertex $v \in V$ if there is no negative-weight cycle in the graph.

Proof. (Sketch) Let $d_k(s, v)$ denote the length of a shortest path from s to v that contains at most k edges. After the k-th iteration of the main for-loop, $d[v] = d_k(s, v)$. After the $(n-1)$-th iteration, $d[v] = d(s, v)$, because $d_{n-1}(s, v) = d(s, v)$. □

Notes on Bellman-Ford algorithm

- The running time of Bellman-Ford algorithm is $\mathcal{O}(|V| \times |E|)$.
- Bellman-Ford algorithm does not work correctly if there exists a *negative-weight cycle* in a graph.
  - In an undirected graph with even a single negative-weight edge, there exists no shortest path for any pair of vertices in the graph.
  - So, Bellman-Ford algorithm will work for
    1. an undirected graph with no negative-weight *edge*, and
    2. a directed graph with no negative-weight *cycle*.

All-Pair Shortest Paths

For each pair of vertices $u, v \in V$, find the shortest distance from u to v.

- We could run Dijkstra's algorithm $|V|$ times.
- The running time is not so bad: $\mathcal{O}(|V|^3)$ or $\mathcal{O}(|V| \times |E| \log |E|)$.

Floyd proposed an algorithm with a dynamic programming flavor.

- Dijkstra's one-dimensional array `d[]` is not enough.
- Use a $|V| \times |V|$ matrix `A[]` instead of the array.

Floyd's Algorithm for All-Pair [1962]

- Assume that vertices are indexed (or numbered) from 0 to $n-1$.
- $A^k[i,j]$ stores the length of a shortest *k-path* from v_i to v_j.
- A *k-path* is a path all intermediate vertices of which are indexed by a number less than k. For example, $v_8 \to v_1 \to v_5 \to v_3$ is a *6-path* (or *7-path*, ...).
- The matrix A is initialized as follows. $A^0[i,j] = \begin{cases} 0 & \text{if } i = j \\ w_{ij} & \text{if } \overline{v_i v_j} \in E \\ \infty & \text{otherwise.} \end{cases}$
- Compute $A^{k+1}[i,j]$ from $A^k[i,j]$.
- $A^n[i,j]$ is the length of a shortest path from v_i to v_j.

- A shortest *(k+1)-path* from v_i to v_j is
  1. **either** a shortest $v_i \rightsquigarrow v_j$ *k-path*,
  2. **or** a shortest $v_i \rightsquigarrow v_k$ *k-path* followed by a shortest $v_k \rightsquigarrow v_j$ *k-path*.
- $A^{k+1}[i,j] = min\{A^k[i,j], A^k[i,k] + A^k[k,j]\}$.

- This algorithm is an example of *Dynamic Programming*.
  - A kind of divide-and-conquer.
  - Break a problem down to simpler subproblems in a recursive manner.
  - Algorithms are iterative rather than recursive.
  - Non-recursive algorithms systematically record the answers to the sub-problems in a table.

Algorithm 6 (Floyd's All-Pair Shortest Paths)

```
// Assume n vertices are indexed from 0 to n-1.
for(i=0; i < n ;i++)                    // initialize A^0 matrix.
   for(j=0; j < n ;j++)
```

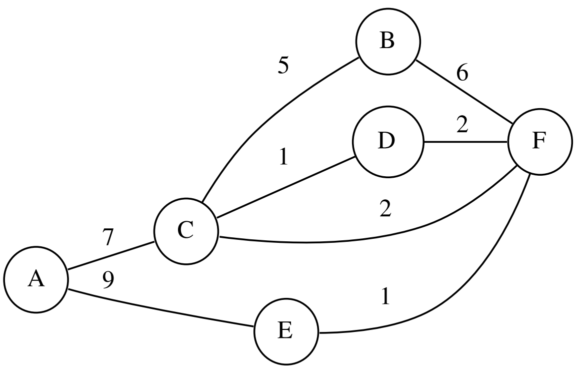
$$\texttt{A[i,j]} = \begin{cases} 0 & \texttt{if } i = j \\ w_{ij} & \texttt{if } \overline{v_i v_j} \in E \\ \infty & \texttt{otherwise} \end{cases}$$

```
for(k=0; k < n ;k++)                    // compute A^{k+1} matrix.
   for(i=0; i < n ;i++)
      for(j=0; j < n ;j++)
         if (A[i,j] > A[i,k]+A[k,j]) A[i,j] = A[i,k]+A[k,j];
```

- Floyd's algorithm works for a graph with negative-weight edges but no negative-weight cycles.

Example 5

Find the shortest path lengths for all pairs of vertices.



$$A^0 = \begin{array}{c} \\ v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array} \begin{array}{c} \begin{array}{cccccc} v_0 & v_1 & v_2 & v_3 & v_4 & v_5 \end{array} \\ \begin{bmatrix} 0 & \infty & 7 & \infty & 9 & \infty \\ & 0 & 5 & \infty & \infty & 6 \\ & & 0 & 1 & \infty & 2 \\ & & & 0 & \infty & 2 \\ & & & & 0 & 1 \\ & & & & & 0 \end{bmatrix} \end{array}$$

Problem 4

Floyd's algorithm takes advantage of the fact that the next matrix (*i.e.*, A^{k+1}) in the sequence can be written over its predecessor (*i.e.*, A^k). Is this safe? How can you be sure that if $A^{k+1}[i,j]$ is updated then it will be updated by A^k values but not A^{k+1} values?

Problem 5

Algorithm 6 (Floyd's shortest path) just computes the length of shortest paths. Modify it to produce actual paths.

HINT: Use another matrix B^k to keep track of preceding vertices on the shortest paths. That is, $B^k[i,j]$ stores the predecessor of v_j on the shortest *k-path* from v_i.

MST: Minimum Spanning Tree

Suppose you need to install a set of secure phone lines that connect all the branch offices with a minimum cost. What should you do?

- All the branch offices stay connected.
- The aggregate length of phone lines is minimal.
- No redundant phone lines are appreciated.

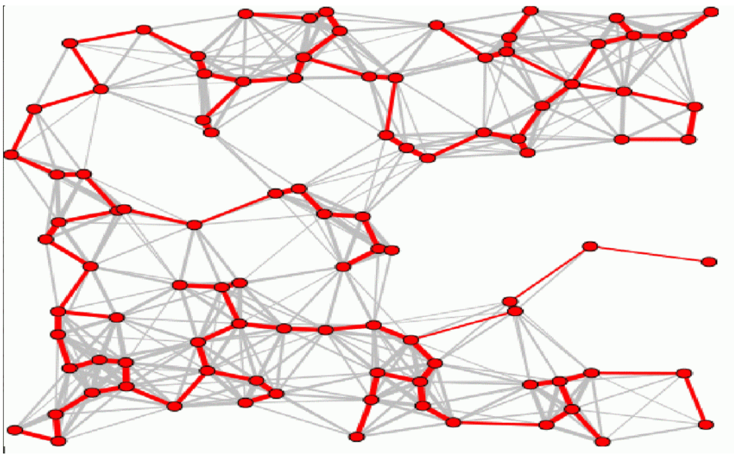
There is a long list of applications such as

- VLSI layout, wireless communication ...
- medical imaging, proteomics, bioterrorism ...

Definition 1

Given an undirected and weighted graph $G = (V, E, W)$, a Minimum Spanning Tree (MST) $T = (V', E', W')$ is a subset of G such that

1. $V' = V, E' \subseteq E$, and $W' \subseteq W$,
2. $\sum_{e \in E'} w(e)$ is minimal,
3. T is connected.



Prim's MST Algorithm

Outline of Prim's algorithm: Let S denote a set of visited vertices.

- Start with $S = \{s\}$, where s is an arbitrary start vertex in V.
- Pick the least-weight edge $\overline{uv} \in E$ such that $u \in S$ and $v \in V - S$.
- Add the vertex v to S; add the edge $\overline{uv}$ to MST.
- Repeat until $V - S$ becomes empty.

Algorithm 7 (Prim's MST (Naive version))

```
S = {s};                          // s is an arbitrary start vertex.
while(V − S ≠ ∅) {
    find the least-weight edge uv ∈ E
              such that u ∈ S and v ∈ V − S;
    print uv;                          // uv becomes part of MST
    S = S ∪ {v};                          // v is pulled to S
}
```


Analysis of the Naive version

What is the running time of the naive version of Prim's MST algorithm?

- Finding the least-weight edge makes $|S| \times |V - S|$ comparisons in the worst case (*i.e.*, fully connected).
- The total number of comparisons is (with k being $|S|$)

$$\leq \sum_{k=1}^{|V|} k \times (|V| - k) = |V| \sum_{k=1}^{|V|} k - \sum_{k=1}^{|V|} k^2$$

$$= \frac{|V|^2(|V|+1)}{2} - \frac{|V|(|V|+1)(2|V|+1)}{6}.$$

- The running time is $\mathcal{O}(|V|^3)$.

Prim's MST: Improved Algorithm

Use an array `d[]` in a <u>similar</u> way Dijkstra's algorithm does. Also, use another array `neighbor[]` to keep track of *candidate* edges.

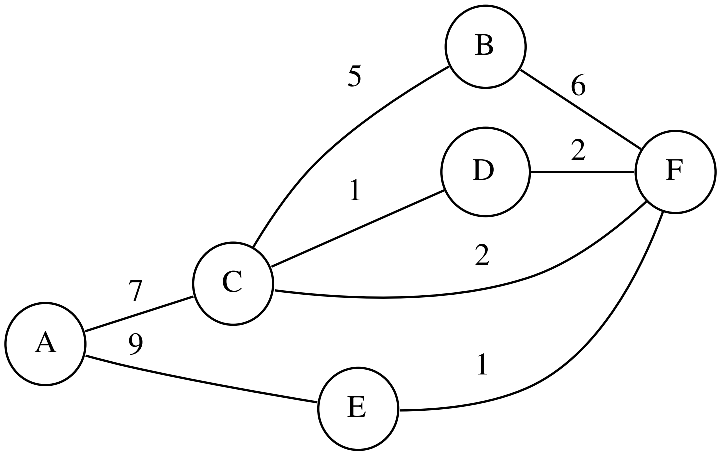
- For each fringe vertex $v \in V - S$,
  - Keep track of the shortest edge from v to ANY vertex in S.
  - For lengths, $d[v] = min\{w(\overline{uv}) \mid u \in S\}$
  - For edges, $neighbor[v] = u$ such that $w_{uv} = d[v]$.
- Whenever a vertex x is added to S,
  - If x reduces $d[v]$, then update $d[v]$ and $neighbor[v]$ for $\forall v \in V - S$.
  - That is, if $w(\overline{xv}) < d[v]$, then $d[v] = w(\overline{xv})$; $neighbor[v] = x$;

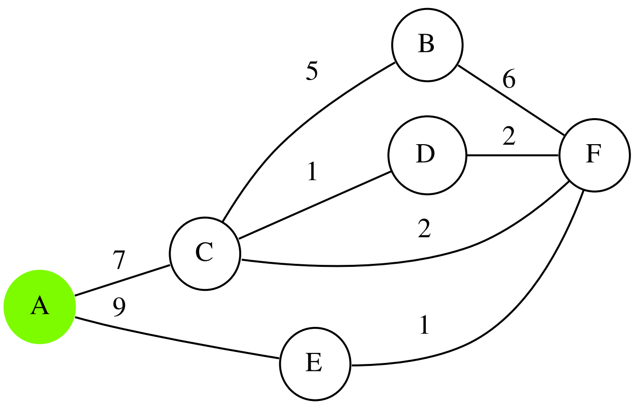
Algorithm 8 (Prim's Minimum Spanning Tree (Improved version))

```
S = {s};                              // s is an arbitrary start vertex.
for each v ∈ V − S {
   d[v], neighbor[v] = { w_sv, s;           if sv ∈ E
                       { ∞, s or null;      otherwise
}
while(V − S ≠ ∅) {
   find v ∈ V − S such that d[v] is minimum;
   print v,neighbor[v];                // This edge becomes part of MST.
   S = S ∪ {v};
   for each u ∈ V − S such that vu ∈ E {
      if (w_uv < d[u]) {                 // uv is a new fringe edge.
         d[u] = w_uv; // the shortest distance to ANY vertex in S.
         neighbor[u] = v;
      }
   }
}
```

Example 6

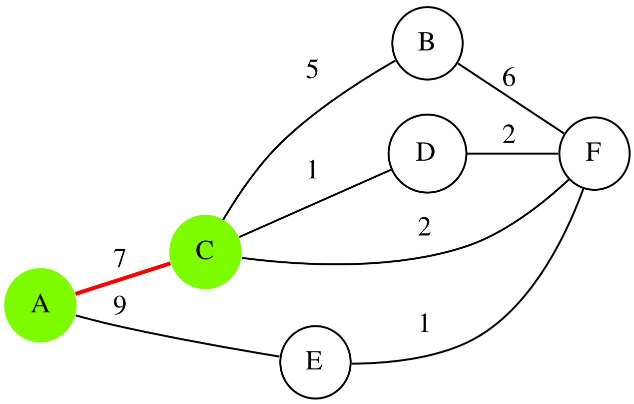
Find a minimum spanning tree using Prim's MST algorithm.





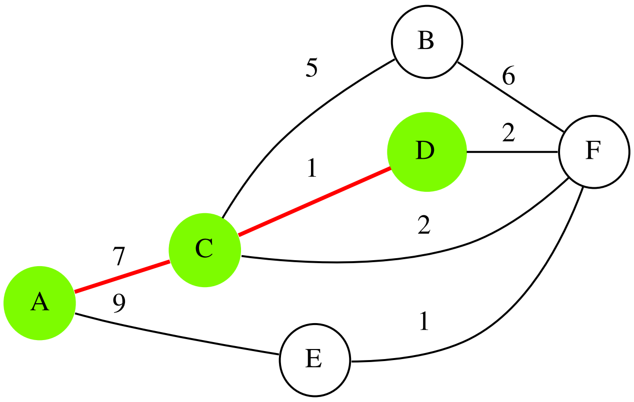
v	A	B	C	D	E	F
d	0	∞	7	∞	9	∞
n	-	A	A	A	A	A

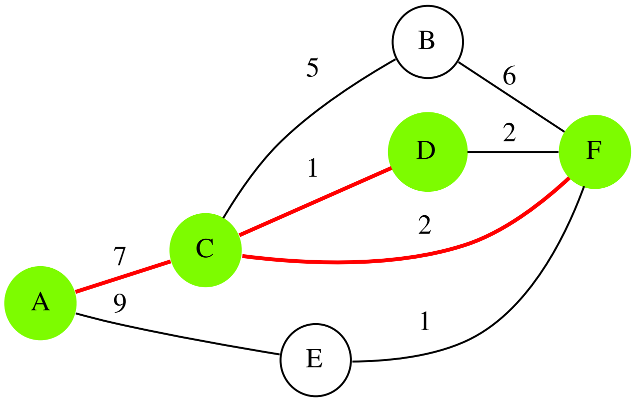
S $= \{A\}$
MST $= \varnothing$



v	A	B	C	D	E	F
d	0	5	7	1	9	2
n	-	C	A	C	A	C

S $= \{A, C\}$
MST $= \{\overline{AC}\}$



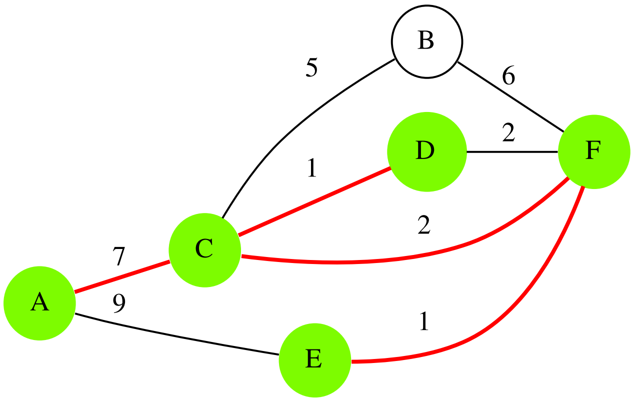


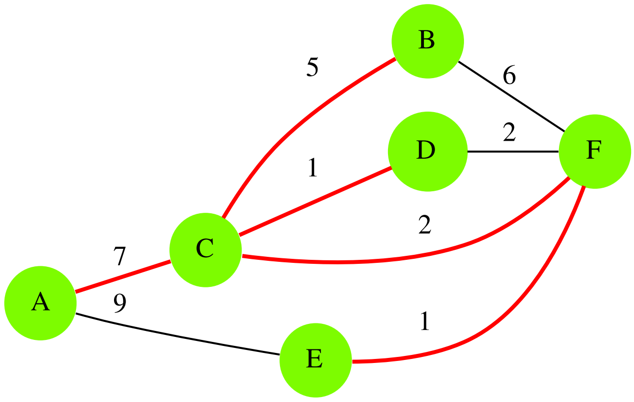
v	A	B	C	D	E	F
d	0	5	7	1	9	2
n	-	C	A	C	A	C

v	A	B	C	D	E	F
d	0	5	7	1	1	2
n	-	C	A	C	F	C

$$S = \{A, C, D\}$$
$$MST = \{\overline{AC}, \overline{CD}\}$$

$$S = \{A, C, D, F\}$$
$$MST = \{\overline{AC}, \overline{CD}, \overline{CF}\}$$





v	A	B	C	D	E	F
d	0	5	7	1	1	2
n	-	C	A	C	F	C

v	A	B	C	D	E	F
d	0	5	7	1	1	2
n	-	C	A	C	F	C

$$S = \{A, C, D, F, E\}$$
$$MST = \{\overline{AC}, \overline{CD}, \overline{CF}, \overline{EF}\}$$

$$S = \{A, C, D, F, E, B\}$$
$$MST = \{\overline{AC}, \overline{CD}, \overline{CF}, \overline{EF}, \overline{BC}\}$$

Analysis of the Improved version

What is the running time of the improved version of Prim's MST algorithm? (Recall that an unsorted list is used to store $d[\,]$.)

- Finding the minimum $d[v]$ makes $|V - S|$ comparisons.
- The total number of comparisons is $\sum_{k=1}^{|V|}(|V| - k) = \frac{|V|(|V|+1)}{2}$.
- For each $v \in V$, $d[v]$ (and $neighbor[v]$) is updated $degree(v)$ times.
- The total number of updates is $\sum_{v \in V} degree(v) = |E|$.

The running time is $\mathcal{O}(|V|^2 + |E|) = \mathcal{O}(|V|^2)$.

With a heap with locator for $d[\,]$, the running time is $\mathcal{O}(|E| \log |V|)$.

- Better than $\mathcal{O}(|V|^2)$ if G is sparse ($|E| \approx |V|$).

Correctness of Prim's MST Algorithm

Theorem 7

Prim's algorithm produces a minimum spanning tree.

Proof. (By contradiction) Suppose $\overline{uv}$ is the first *wrong* edge chosen by Prim's algorithm. Then, there must exist an MST $T(V, E', W')$ of $G(V, E, W)$ such that $\overline{uv} \notin E'$. Consider the moment when $\overline{uv}$ is selected. $\overline{uv}$ is a fringe edge connecting S and $V - S$. Although $\overline{uv} \notin E'$, there exists a path between u and v in T because $u, v \in V$. This implies that there exists another fringe edge $\overline{xy} \in T$ connecting S and $V - S$.

$T \cup \{\overline{uv}\}$ has a cyple, but $T \cup \{\overline{uv}\} - \{\overline{xy}\}$ will be another spanning tree because $\overline{uv}$ and $\overline{xy}$ are part of a cycle. Since Prim's algorithm selects $\overline{uv}$ instead of $\overline{xy}$ when both are fringe edges between S and $V - S$, it must be that $w(\overline{uv}) \leq w(\overline{xy})$. Therefore, $w(T \cup \{\overline{uv}\} - \{\overline{xy}\}) \leq w(T)$. This makes $T \cup \{\overline{uv}\} - \{\overline{xy}\}$ another MST, which is a contradiction to the assumption that $\overline{uv}$ is a wrong edge. □

Kruskal's MST algorithm

Outline of Kruskal's algorithm

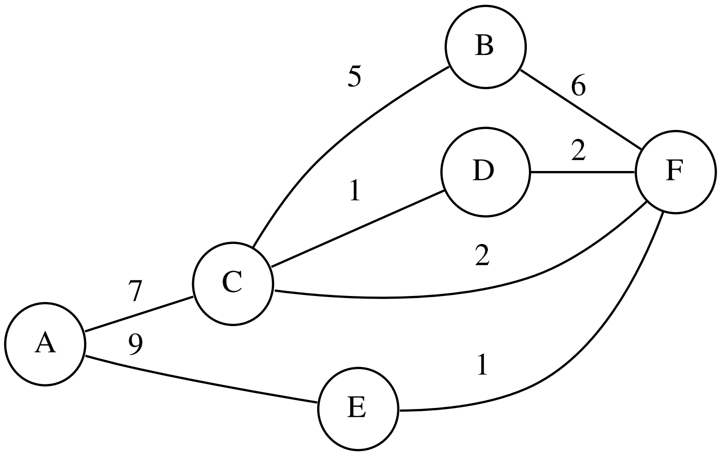
- Start with $|V|$ equivalence classes (one vertex in each class).
- Merge the classes until an MST is found (a single eq. class).

Algorithm 9 (Kruskal's Minimum Spanning Tree)

Assign each vertex to a separate class;
$F = \{e \in E$ | in an increasing order of weights$\}$;
while(number of printed edges < |V|-1) {
  Pick an edge $\overline{uv} \in F$ in the order;
  if (u and v are in different classes) {
    print $\overline{uv}$;
    merge their classes;
  }
}

Example 7

Find a minimum spanning tree using Kruskal's MST algorithm.



edges	$\overline{CD}$	$\overline{EF}$	$\overline{CF}$	$\overline{DF}$	$\overline{BC}$	$\overline{BF}$	$\overline{AC}$	$\overline{AE}$
weights	1	1	2	2	5	6	7	9
MST								

Analysis of Kruskal's MST

- Sort the edges in increasing order of weights: $\mathcal{O}(|E| \log |E|)$.
- Pick edges in the order: $\mathcal{O}(|E|)$.
- Check and merge classes: $\mathcal{O}(|E| \times |V|)$ or $\mathcal{O}(|E| \times \log |V|)$ by UNION/FIND.

If UNION/FIND algorithm is used, the running time is

$$\mathcal{O}(|E| \log |E| + |E| \log |V|) = \mathcal{O}(|E| \log |E|).$$

- Can be better or worse than Prim's $\mathcal{O}(|V|^2)$.
- Comparable to Prim's $\mathcal{O}(|E| \log |V|)$ (with a heap with locator) for both sparse ($|E| \approx |V|$) or dense ($|E| \approx |V|^2$) graphs.