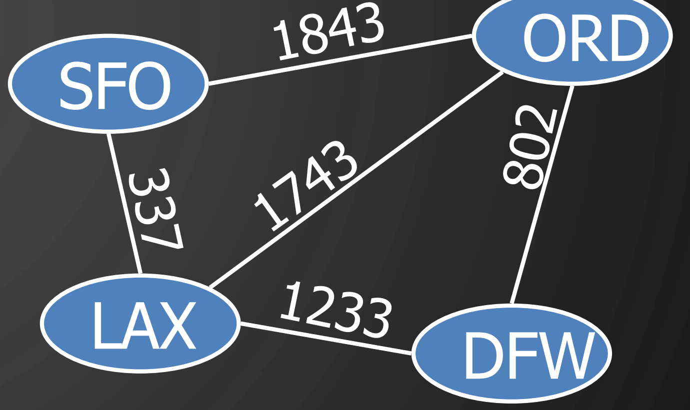
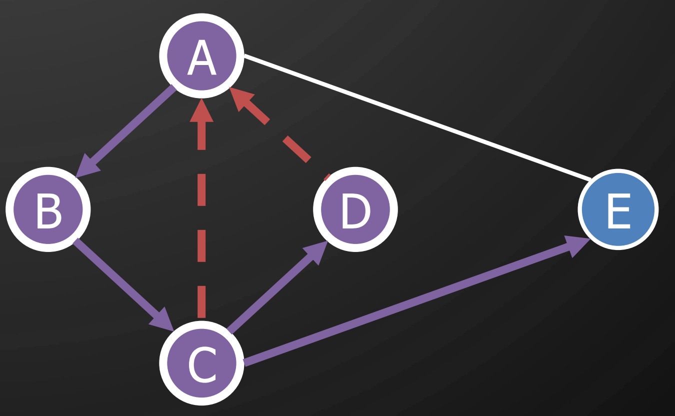
CHAPTER 14
GRAPH ALGORITHMS



ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH DATA STRUCTURES AND ALGORITHMS IN JAVA, GOODRICH, TAMASSIA AND GOLDWASSER (WILEY 2016)

DEPTH-FIRST SEARCH

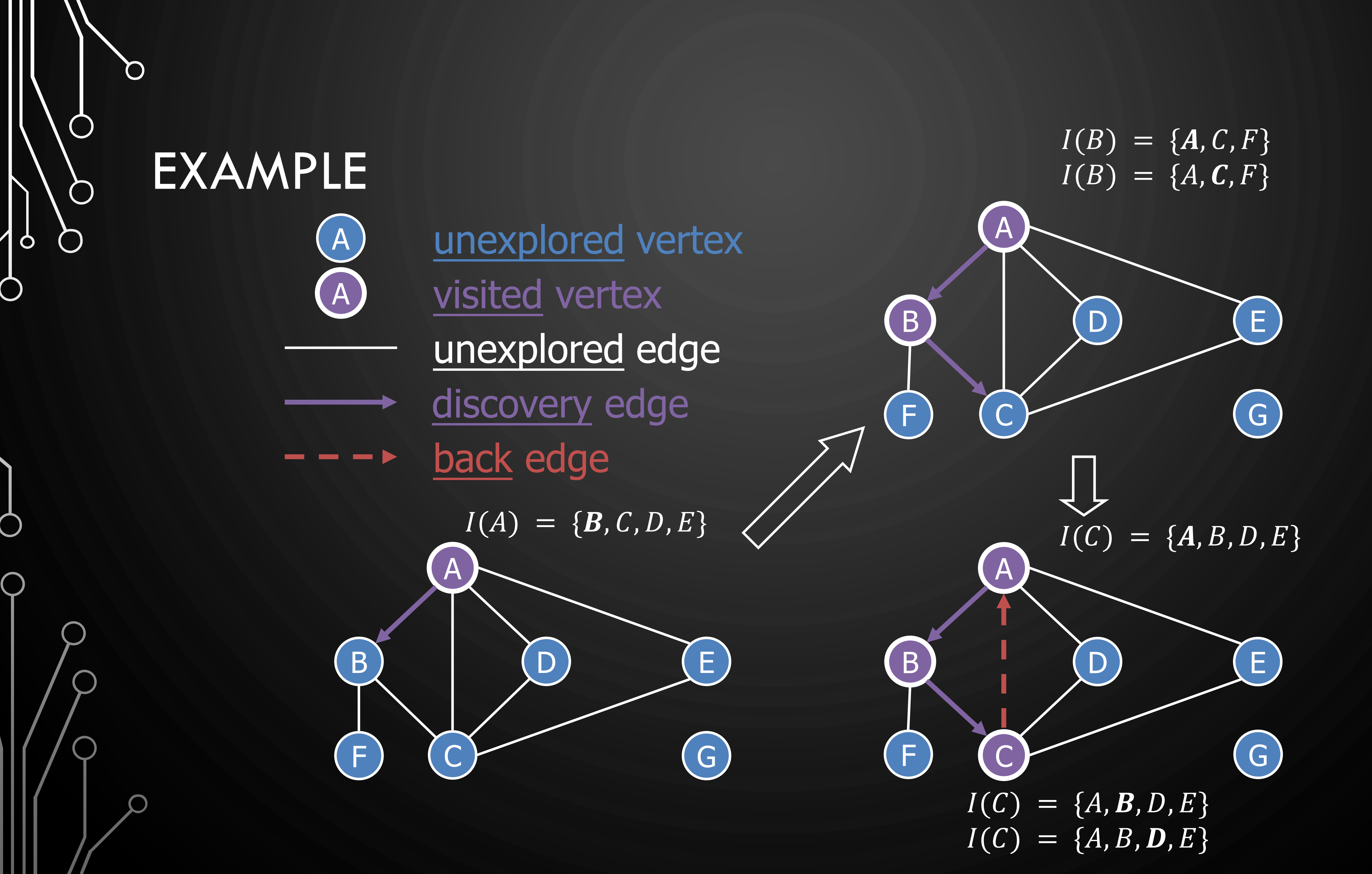


DEPTH-FIRST SEARCH

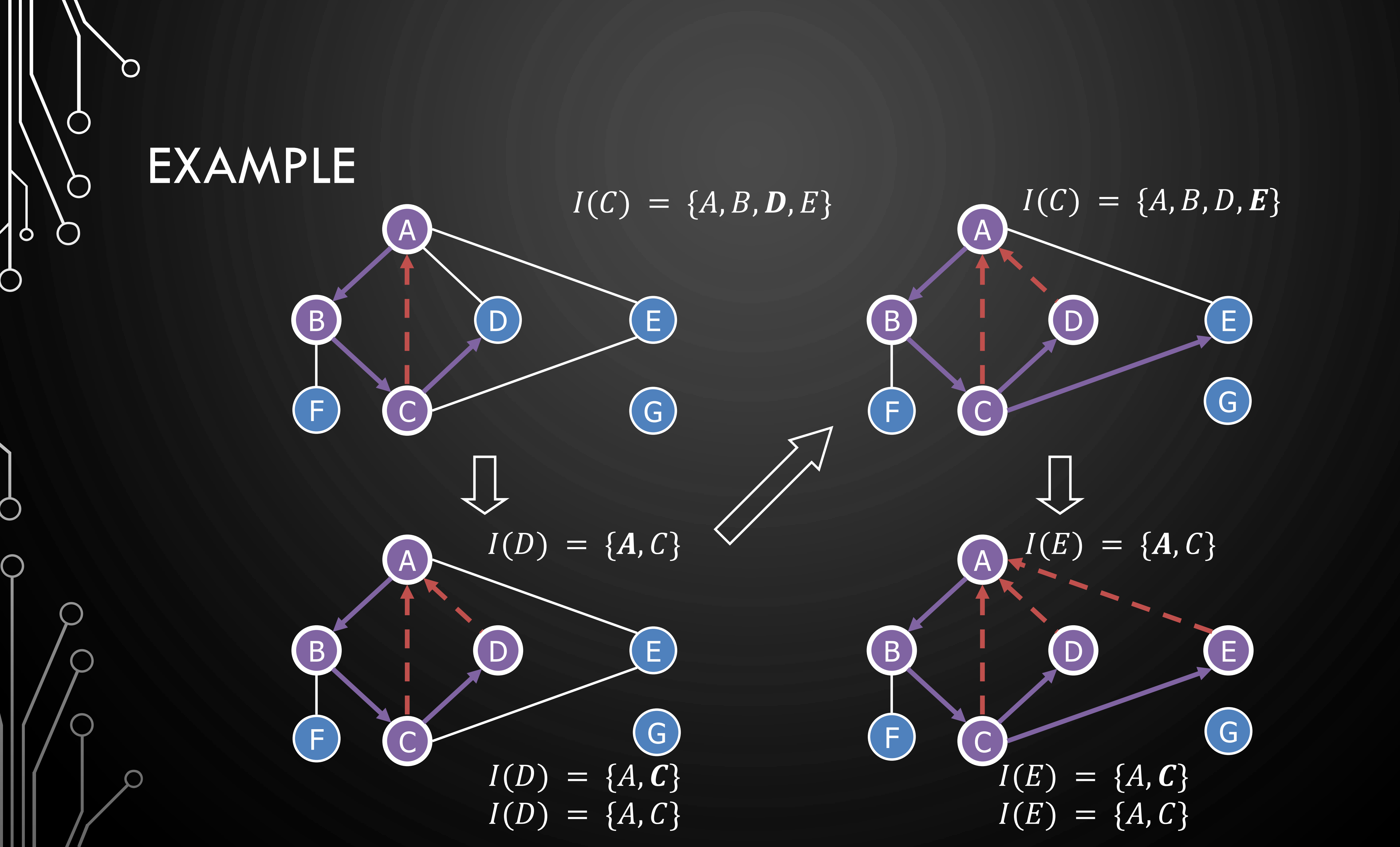
- **Depth-first search (DFS)** is a general technique for traversing a graph
- A DFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
- DFS on a graph with n vertices and m edges takes $O(n + m)$ time
- DFS can be further extended to solve other graph problems
  - Find and report a path between two given vertices
  - Find a cycle in the graph
- Depth-first search is to graphs as what Euler tour is to binary trees

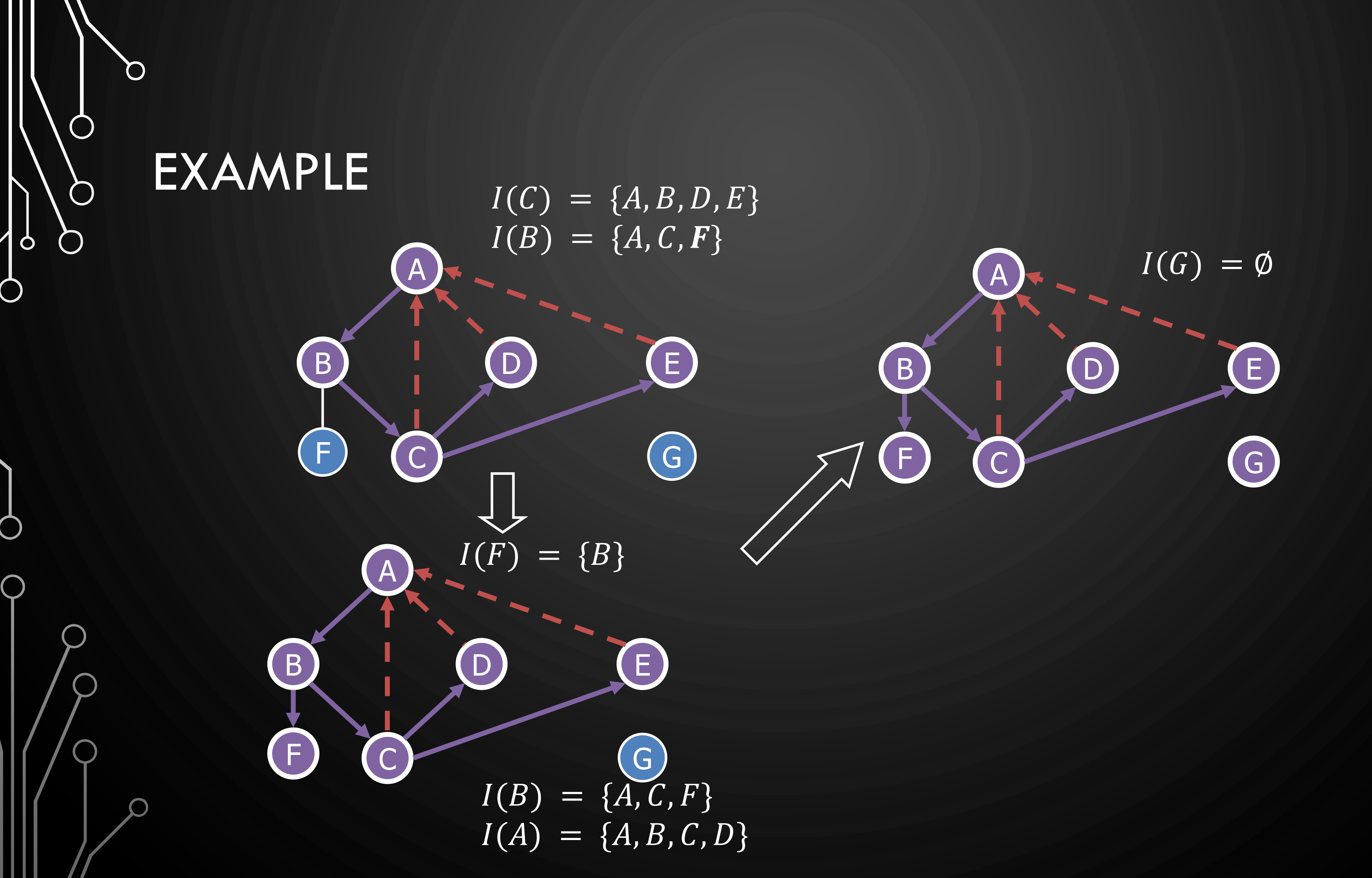
DFS ALGORITHM FROM A VERTEX

```
Algorithm DFS(G, u)
Input: A graph G and a vertex u of G
Output: A collection of vertices reachable from u,
        with their discovery edges
1.  Mark u as visited
2.  for each edge e = (u,v) ∈ G.outgoingEdges(u) do
3.    if v has not been visited then
4.      Record e as a discovery edge for v
5.      DFS(G, v)
```


EXAMPLE
A
unexplored vertex
A
visited vertex
unexplored edge
discovery edge
back edge
I(A) = {B, C, D, E}
A
B
D
E
F
C
G
I(B) = {A, C, F}
I(B) = {A, C, F}
A
B
D
E
F
C
G
I(C) = {A, B, D, E}
A
B
D
E
F
C
G
I(C) = {A, B, D, E}
I(C) = {A, B, D, E}

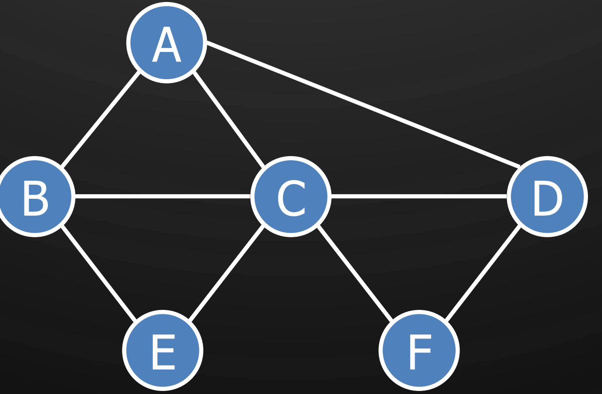
EXAMPLE



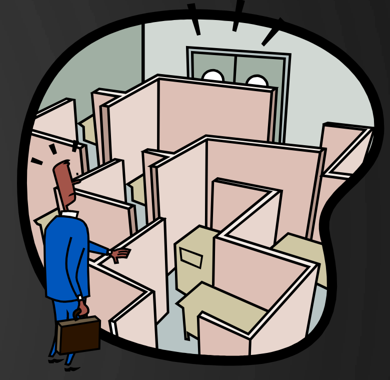

EXAMPLE
I(C) = {A, B, D, E}
I(B) = {A, C, F}
A
B
D
E
F
C
G
I(F) = {B}
A
B
D
E
F
C
G
I(B) = {A, C, F}
I(A) = {A, B, C, D}
I(G) = ∅
A
B
D
E
F
C
G

EXERCISE
DFS ALGORITHM

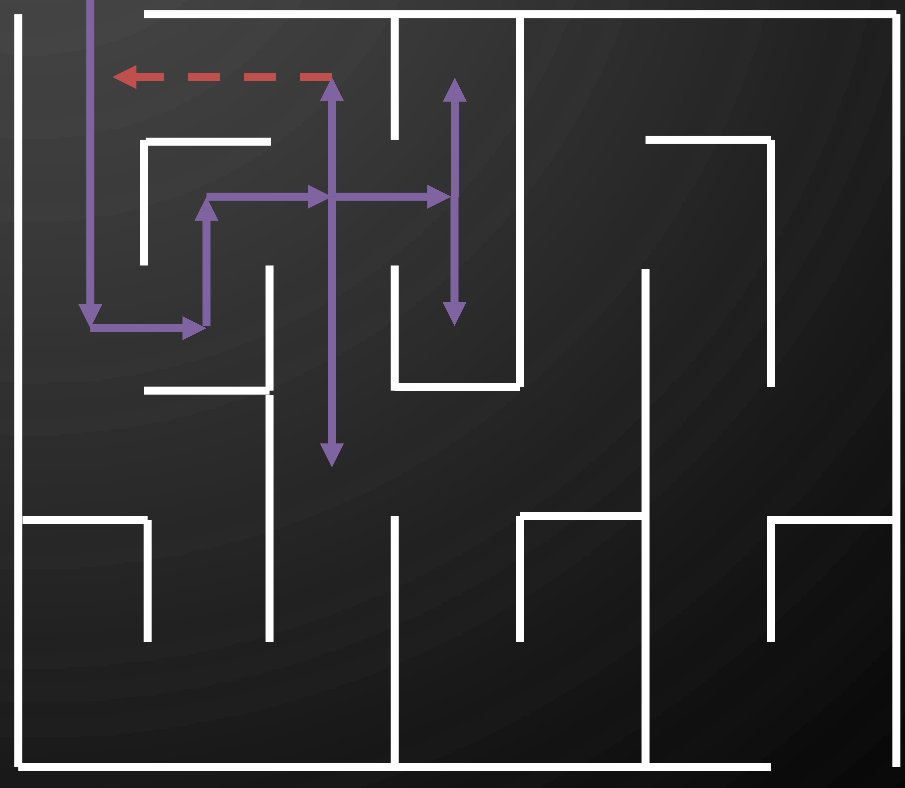
- Perform DFS of the following graph, start from vertex A
  - Assume adjacent edges are processed in alphabetical order
  - Number vertices in the order they are visited
  - Label edges as discovery or back edges



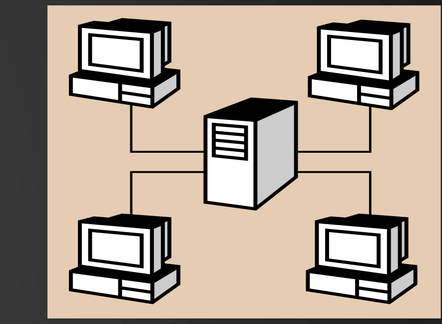
DFS AND MAZE TRAVERSAL



- The DFS algorithm is similar to a classic strategy for exploring a maze
  - We mark each intersection, corner and dead end (vertex) visited
  - We mark each corridor (edge) traversed
  - We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)



DFS ALGORITHM



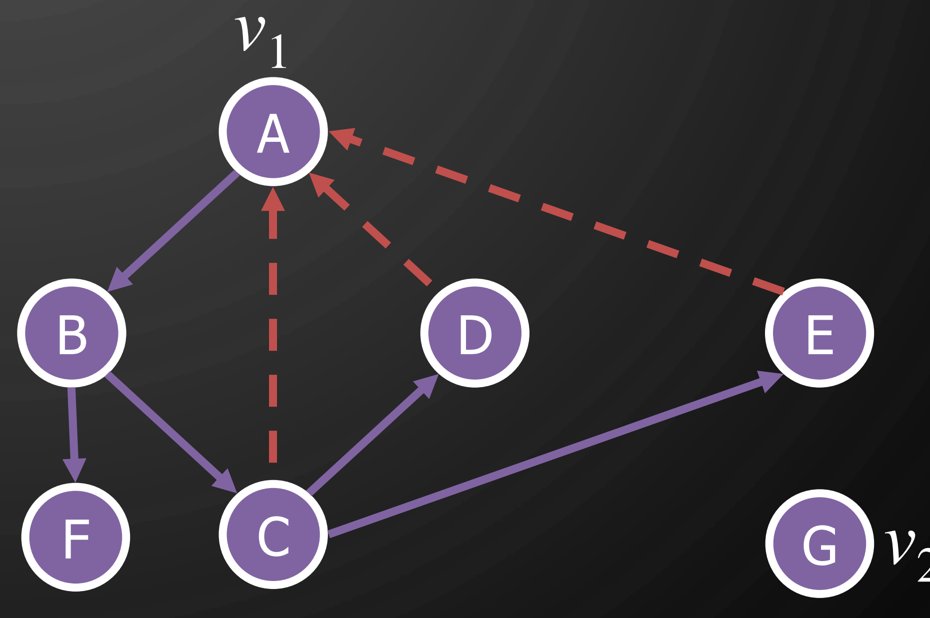
- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

```
Algorithm DFS(G)
Input: Graph G
Output: Labeling of the edges of G
  as discovery edges and back edges
1.  for each v ∈ G.vertices() do
2.    setLabel(v, UNEXPLORED)
3.  for each e ∈ G.edges() do
4.    setLabel(e, UNEXPLORED)
5.  for each v ∈ G.vertices() do
6.    if getLabel(v) = UNEXPLORED then
7.      DFS(G, v)
```

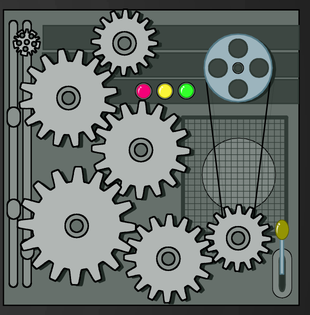
```
Algorithm DFS(G, v)
Input: Graph G and a start vertex v
Output: Labeling of the edges of G in
   the connected component of v as
   discovery edges and back edges
1.  setLabel(v, VISITED)
2.  for each e ∈ G.outgoingEdges(v) do
3.    if getLabel(e) = UNEXPLORED)
4.      w ← G.opposite(v, e)
5.      if getLabel(w) = UNEXPLORED then
6.        setLabel(e, DISCOVERY)
7.        DFS(G, w)
8.      else
9.        setLabel(e, BACK)
```

PROPERTIES OF DFS

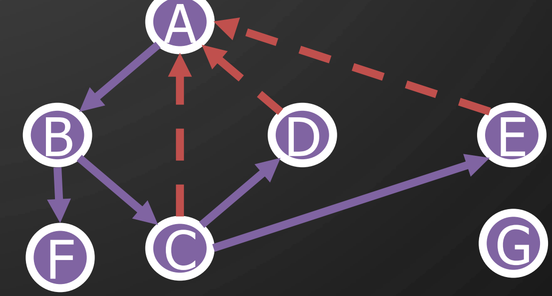
- Property 1
  - $\text{DFS}(G, v)$ visits all the vertices and edges in the connected component of v
- Property 2
  - The discovery edges labeled by $\text{DFS}(G, v)$ form a spanning tree of the connected component of v



ANALYSIS OF DFS



- 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 $BACK$



- Function $\mathrm{DFS}(G, v)$ and the method $\mathrm{outgoingEdges}(\,)$ are called once for each vertex
- DFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\Sigma_v \deg(v) = 2m$

APPLICATION

PATH FINDING

- We can specialize the DFS algorithm to find a path between two given vertices u and z using the template method pattern
- We call $\text{DFS}(G, u)$ with u as the start vertex
- We use a stack S to keep track of the path between the start vertex and the current vertex
- As soon as destination vertex z is encountered, we return the path as the contents of the stack

```
Algorithm pathDFS(G,v,z)
Input: Graph G, a start vertex v,
       a goal vertex z
Output: Path between v and z
1.  setLabel(v, VISITED)
2.  S.push(v)
3.  if v = z then
4.    return S.elements()
5.  for each e ∈ G.outgoingEdges(v) do
6.    if getLabel(e) = UNEXPLORED) then
7.      w ← G.opposite(v,e)
8.      if getLabel(w) = UNEXPLORED then
9.        setLabel(e, DISCOVERY)
10.       S.push(e)
11.       pathDFS(G,w)
12.       S.pop()
13.     else
14.       setLabel(e, BACK)
15. S.pop()
```

APPLICATION

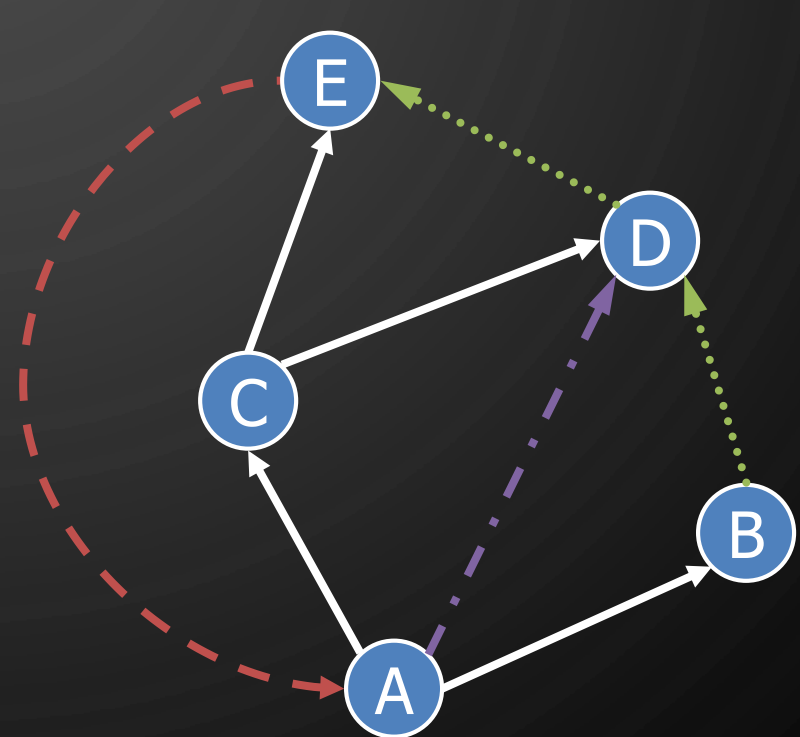
CYCLE FINDING

- We can specialize the DFS algorithm to find a simple cycle using the template method pattern
- We use a stack S to keep track of the path between the start vertex and the current vertex
- As soon as a back edge (v, w) is encountered, we return the cycle as the portion of the stack from the top to vertex w

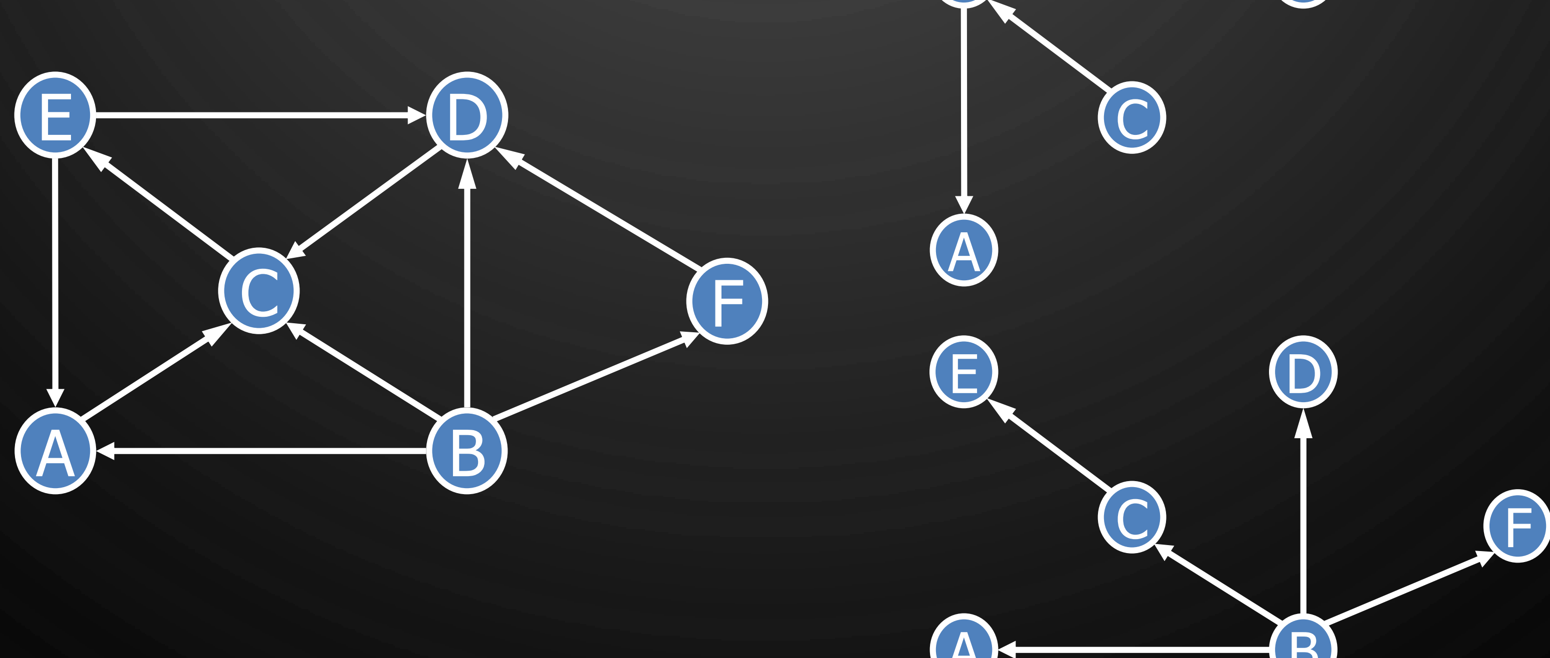
```
Algorithm cycleDFS(G, v)
Input: Graph G, a start vertex v
Output: Cycle containing v
1.  setLabel(v, VISITED)
2.  S.push(v)
3.  for each e ∈ G.outgoingEdges(v) do
4.    if getLabel(e) = UNEXPLORED) then
5.      w ← G.opposite(v, e)
6.      S.push(e)
7.      if getLabel(w) = UNEXPLORED then
8.        setLabel(e, DISCOVERY)
9.        cycleDFS(G, w)
10.       S.pop()
11.     else
12.       Stack T ← ∅
13.       repeat
14.         T.push(S.pop())
15.       until T.top() = w
16.       return T.elements()
17. S.pop()
```

DIRECTED DFS

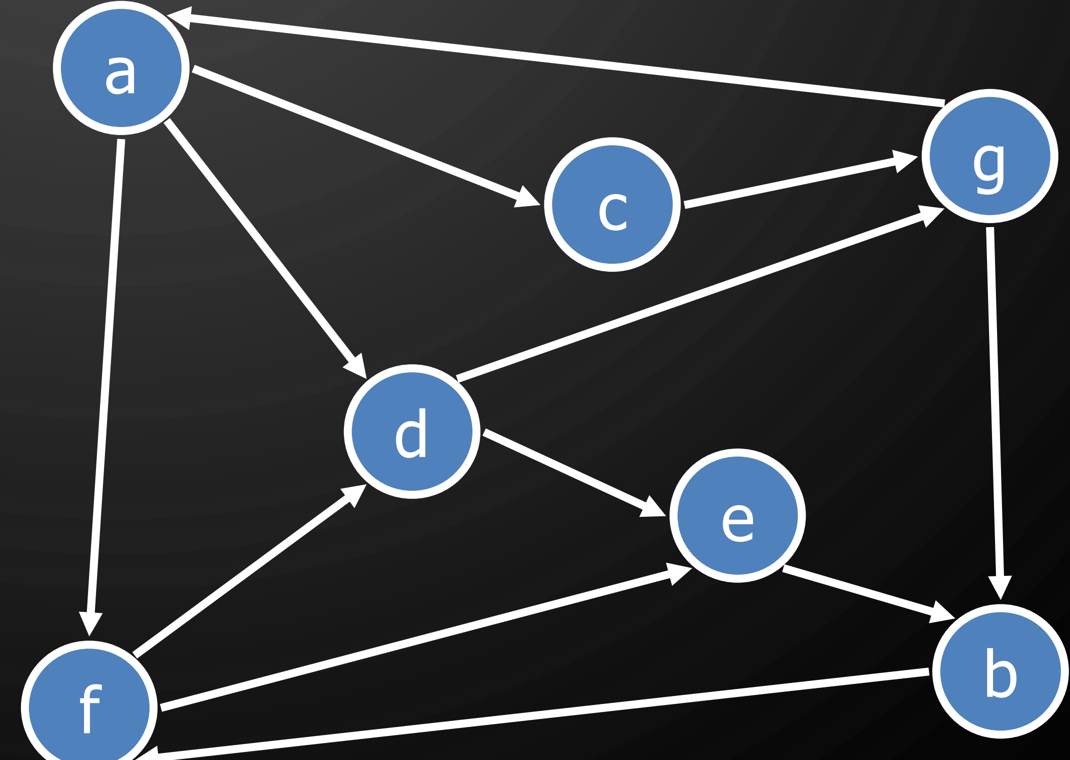
- We can specialize the traversal algorithms (DFS and BFS) to digraphs by traversing edges only along their direction
- In the directed DFS algorithm, we have four types of edges
  - discovery edges
  - back edges
  - forward edges
  - cross edges
- A directed DFS starting at a vertex s determines the vertices reachable from s



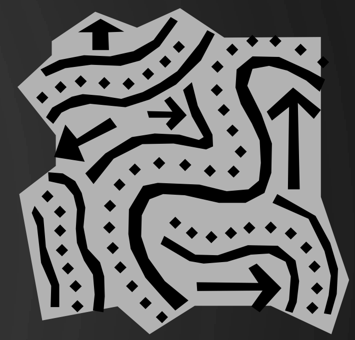
- DFS tree rooted at v: vertices reachable from v via directed paths



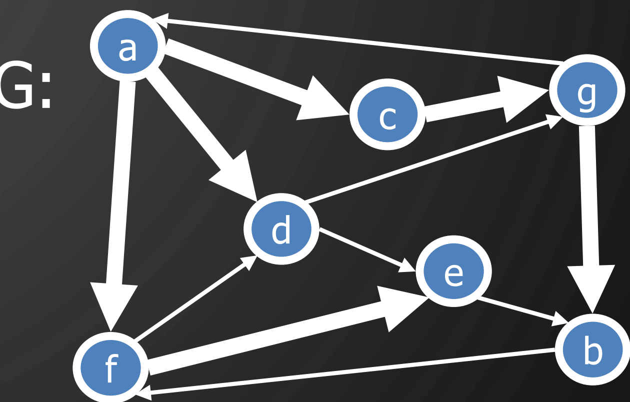
- Each vertex can reach all other vertices

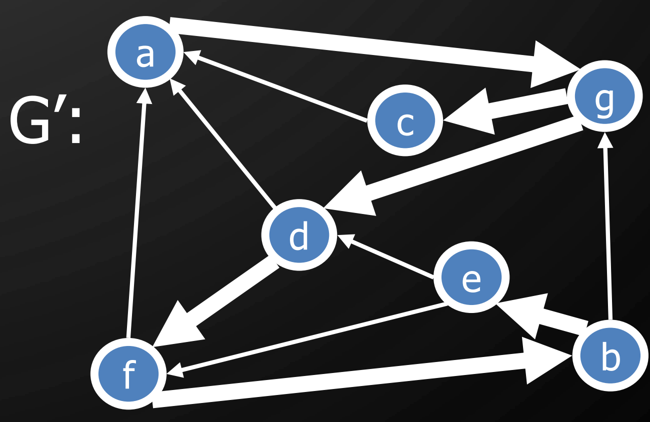


STRONG CONNECTIVITY ALGORITHM



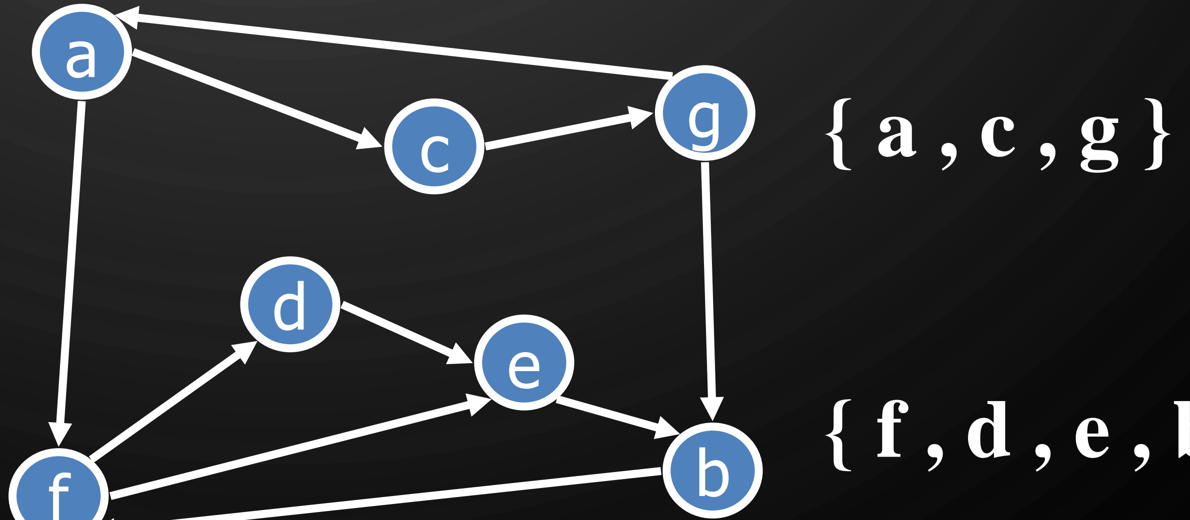
- Pick a vertex v in G
- Perform a DFS from v in G
  - If there's a w not visited, print "no"
- Let G' be G with edges reversed
- Perform a DFS from v in G'
  - If there's a w not visited, print "no"
  - Else, print "yes"
- Running time: $O(n+m)$



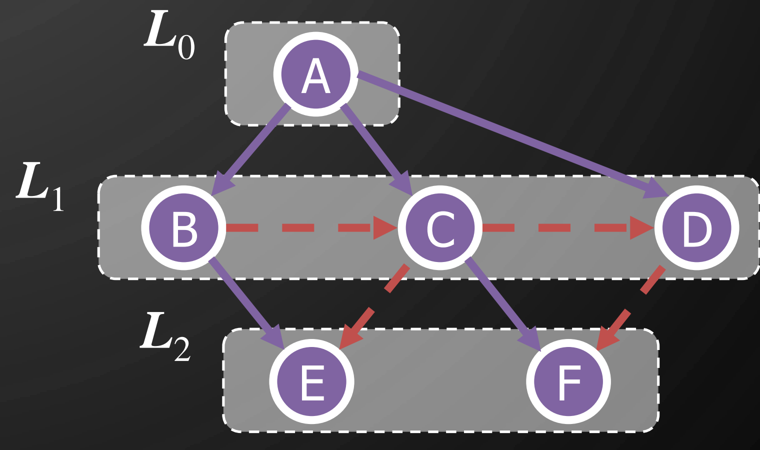


STRONGLY CONNECTED COMPONENTS

- Maximal subgraphs such that each vertex can reach all other vertices in the subgraph
- Can also be done in $O(n + m)$ time using DFS, but is more complicated (similar to biconnectivity).



BREADTH-FIRST SEARCH



BREADTH-FIRST SEARCH

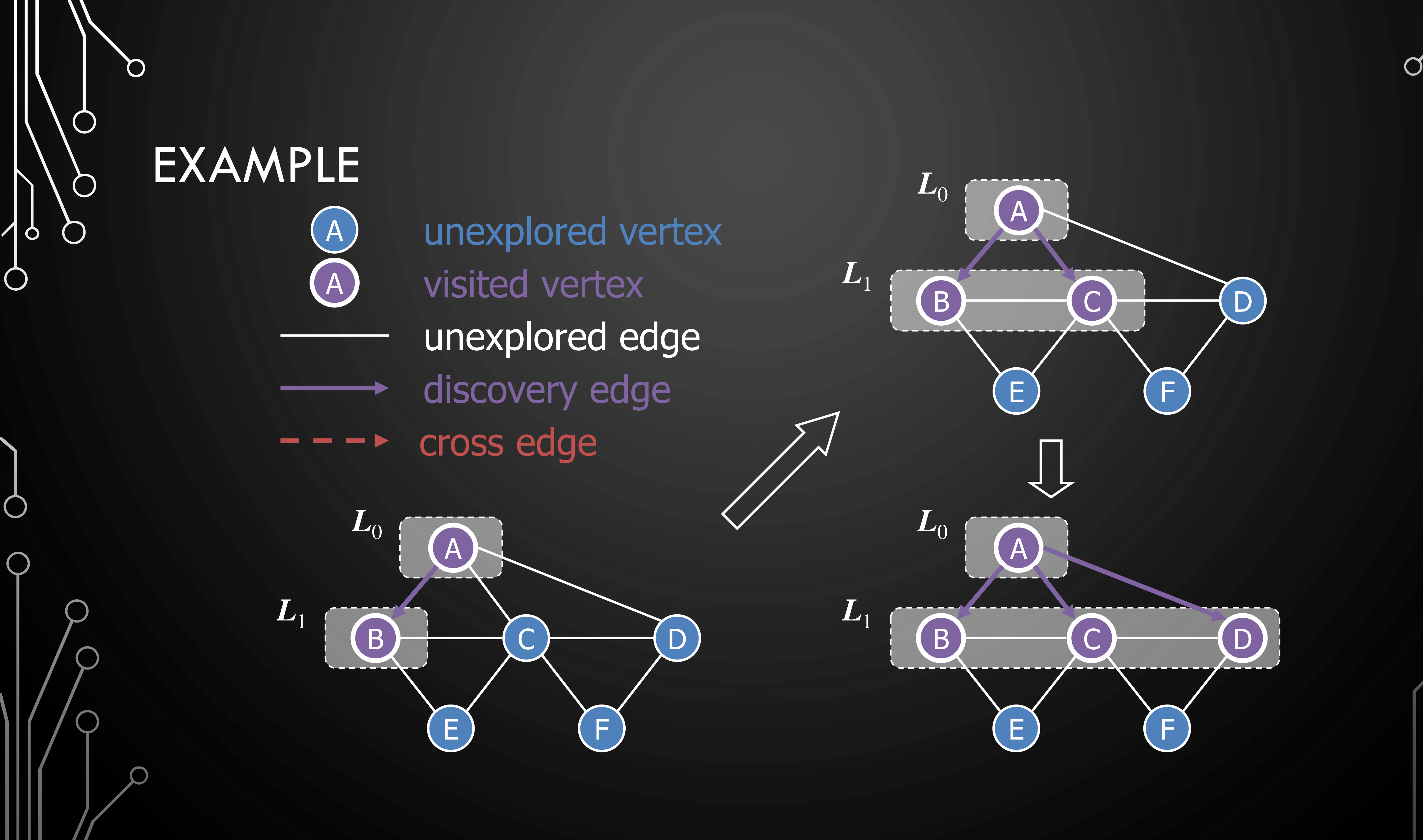
- **Breadth-first search (BFS)** is a general technique for traversing a graph
- A BFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
- BFS on a graph with n vertices and m edges takes $O(n + m)$ time
- BFS can be further extended to solve other graph problems
  - Find and report a path with the minimum number of edges between two given vertices
  - Find a simple cycle, if there is one

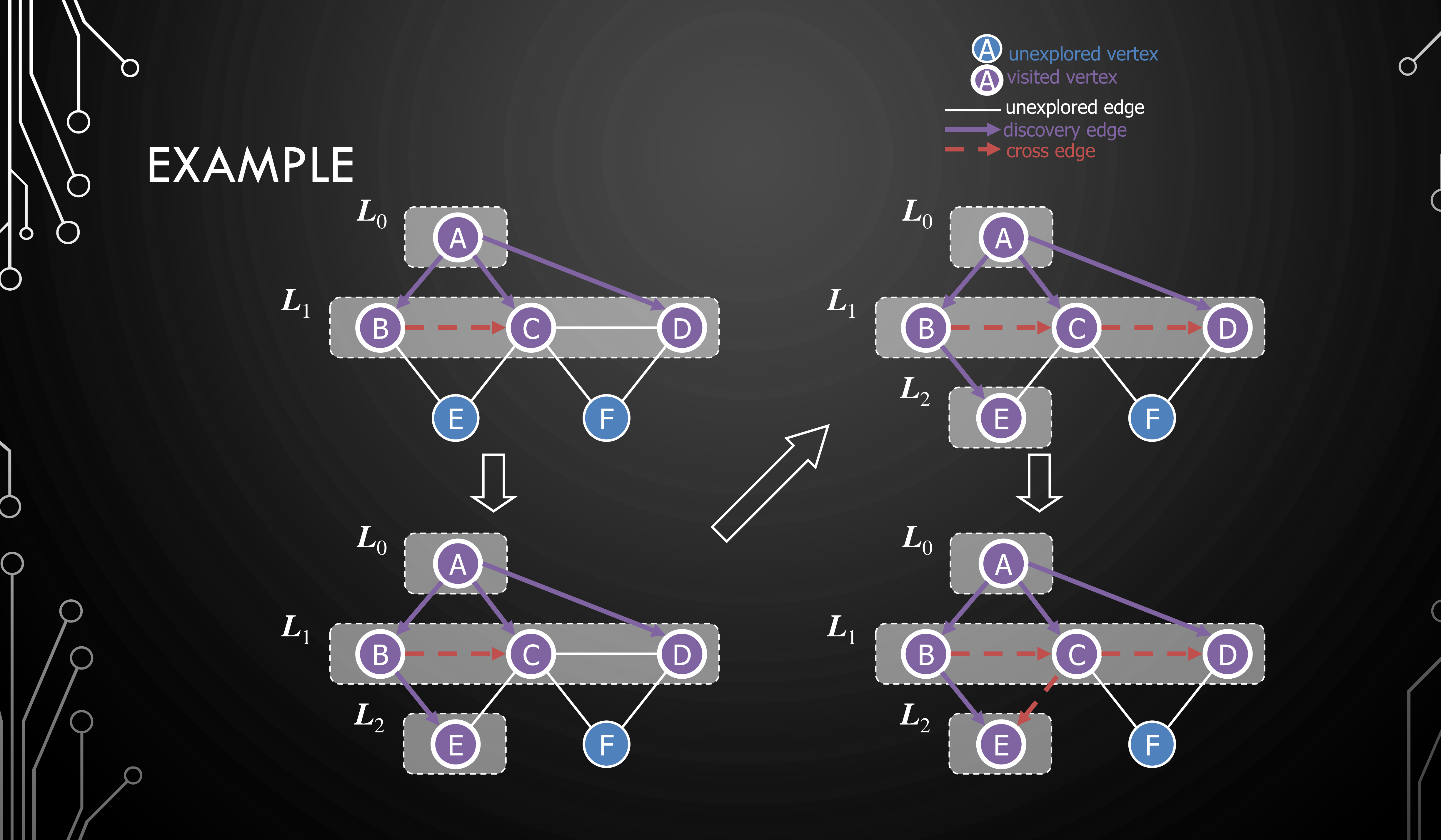
BFS ALGORITHM

- The algorithm uses a mechanism for setting and getting “labels” of vertices and edges

```
Algorithm BFS(G)
Input: Graph G
Output: Labeling of the edges and
   partition of the vertices of G
1. for each v ∈ G.vertices() do
2.   setLabel(v, UNEXPLORED)
3. for each e ∈ G.edges() do
4.   setLabel(e, UNEXPLORED)
5. for each v ∈ G.vertices() do
6.   if getLabel(v) = UNEXPLORED then
7.     BFS(G, v)
```

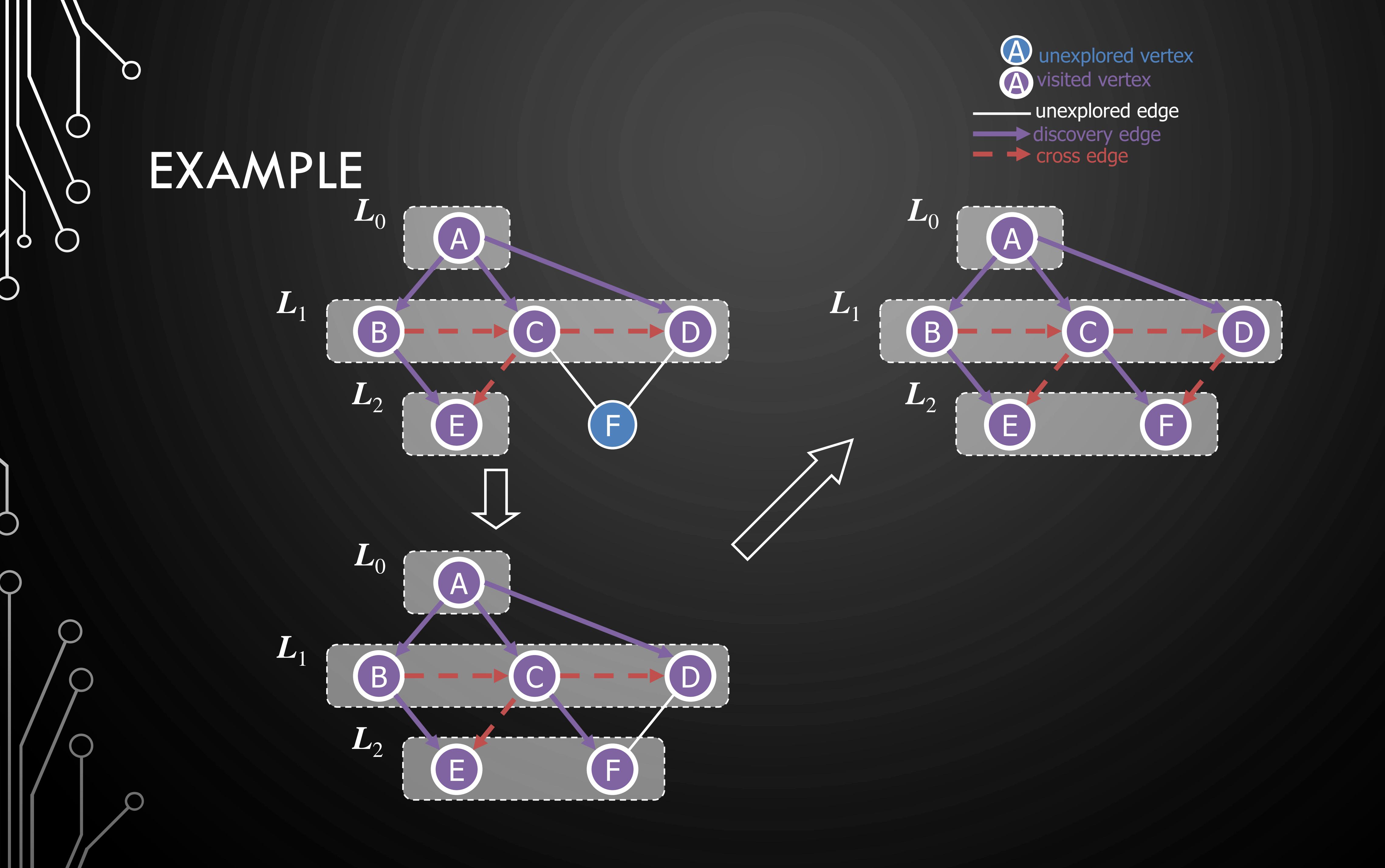
```
Algorithm BFS(G, s)
Input: Graph G, a start vertex s
1.  List L_0 ← {s}
2.  setLabel(s, VISITED)
3.  i ← 0
4.  while ¬L_i.isEmpty() do
5.    List L_{i+1} ← ∅
6.    for each v ∈ L_i do
7.      for each e ∈ G.outgoingEdges(v) do
8.        if getLabel(e) = UNEXPLORED then
9.          w ← G.opposite(v, e)
10.         if getLabel(w) = UNEXPLORED then
11.           setLabel(e, DISCOVERY)
12.           setLabel(w, VISITED)
13.           L_{i+1} ← L_{i+1} ∪ {w}
14.         else
15.           setLabel(e, CROSS)
16.   i ← i + 1
```


EXAMPLE
A
unexplored vertex
A
visited vertex
unexplored edge
discovery edge
cross edge
L_0
L_1
A
B
C
D
E
F
L_0
L_1
A
B
C
D
E
F
L_0
L_1
A
B
C
D
E
F



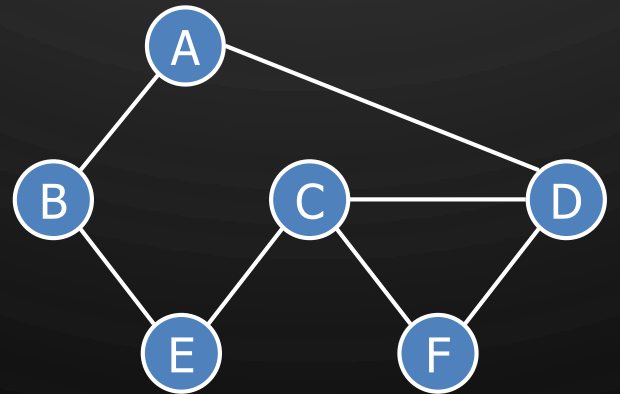
EXAMPLE
A unexplored vertex
A visited vertex
unexplored edge
discovery edge
cross edge
L_0
L_1
L_2
A
B
C
D
E
F

EXAMPLE



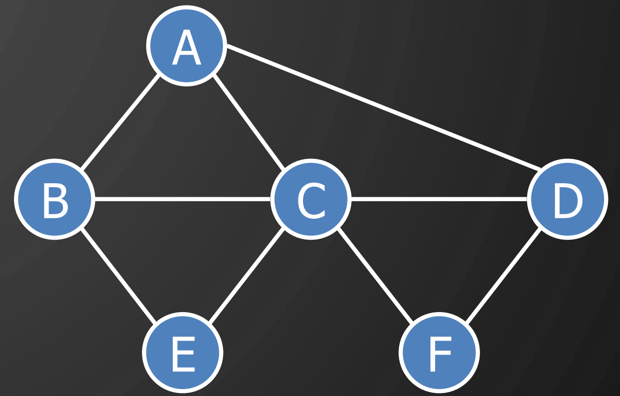
EXERCISE
BFS ALGORITHM

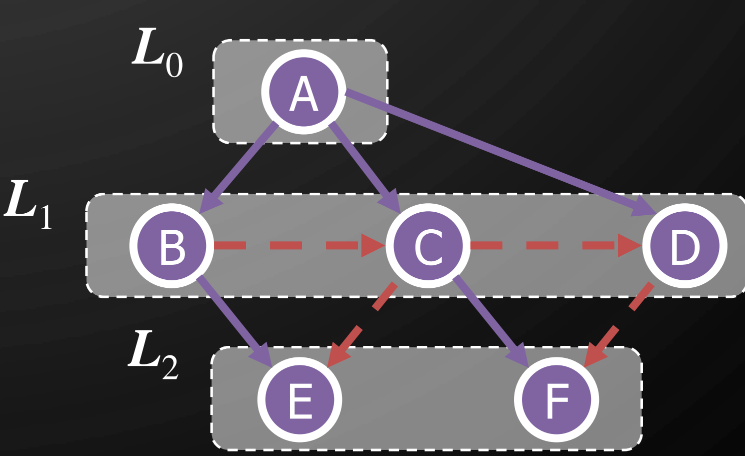
- Perform BFS of the following graph, start from vertex F
  - Assume adjacent edges are processed in alphabetical order
  - Number vertices in the order they are visited and note the level they are in
  - Label edges as discovery or cross edges



PROPERTIES

- Notation
  - G_s: connected component of s
- Property 1
  - $\mathrm{BFS}(G, s)$ visits all the vertices and edges of G_s
- Property 2
  - The discovery edges labeled by $\mathrm{BFS}(G, s)$ form a spanning tree T_s of G_s
- Property 3
  - For each vertex $v \in L_i$
    - The path of T_s from s to v has i edges
    - Every path from s to v in G_s has at least i edges





ANALYSIS

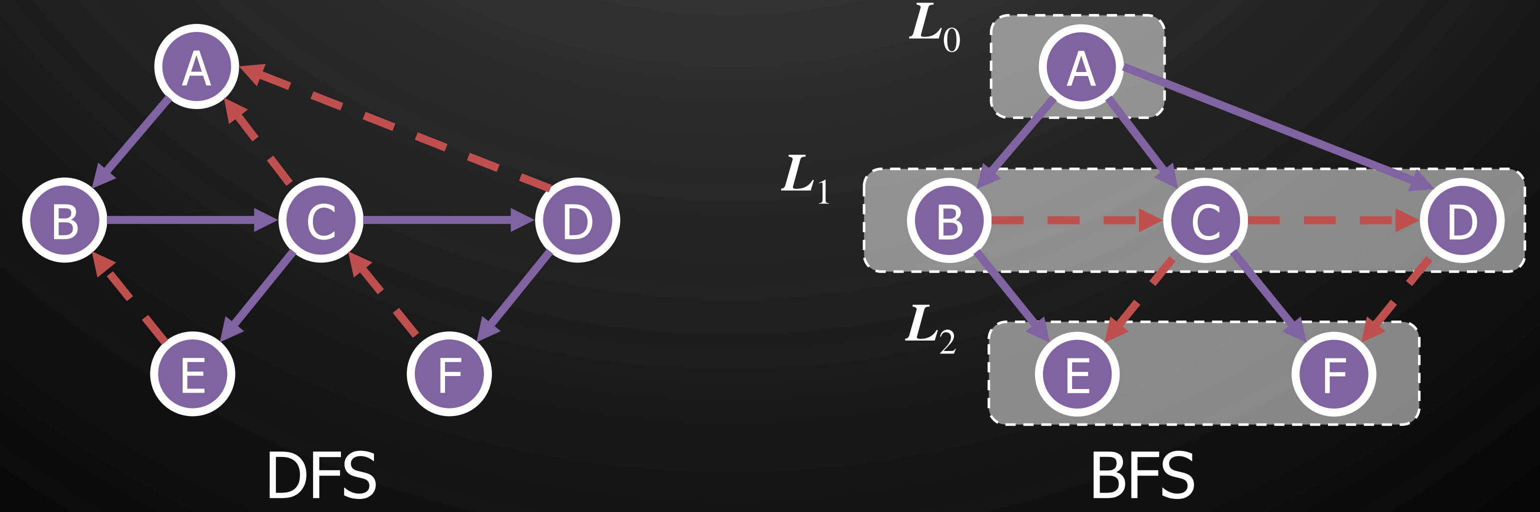
- Setting/getting a vertex/edge label takes O(1) time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or CROSS
- Each vertex is inserted once into a sequence L_i
- Method $\mathrm{outgoingEdges}(\,)$ is called once for each vertex
- BFS runs in $O(n+m)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\Sigma_v \deg(v) = 2m$

APPLICATIONS

- Using the template method pattern, we can specialize the BFS traversal of a graph G to solve the following problems in $O(n + m)$ time
  - Compute the connected components of G
  - Compute a spanning forest of G
  - Find a simple cycle in G, or report that G is a forest
  - Given two vertices of G, find a path in G between them with the minimum number of edges, or report that no such path exists

DFS VS. BFS

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	√	√
Shortest paths		√
Biconnected components	√	



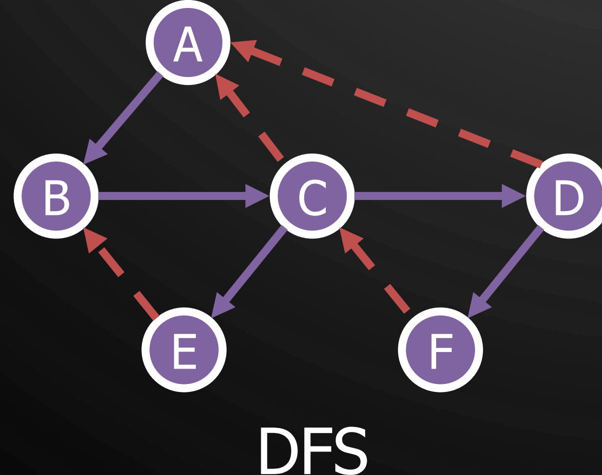
DFS VS. BFS

Back edge (v, w)

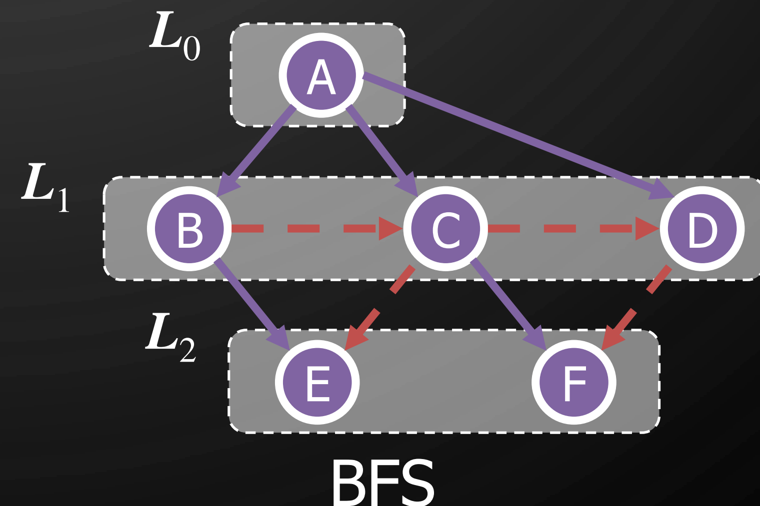
- w is an ancestor of v in the tree of discovery edges

Cross edge (v, w)

- w is in the same level as v or in the next level in the tree of discovery edges

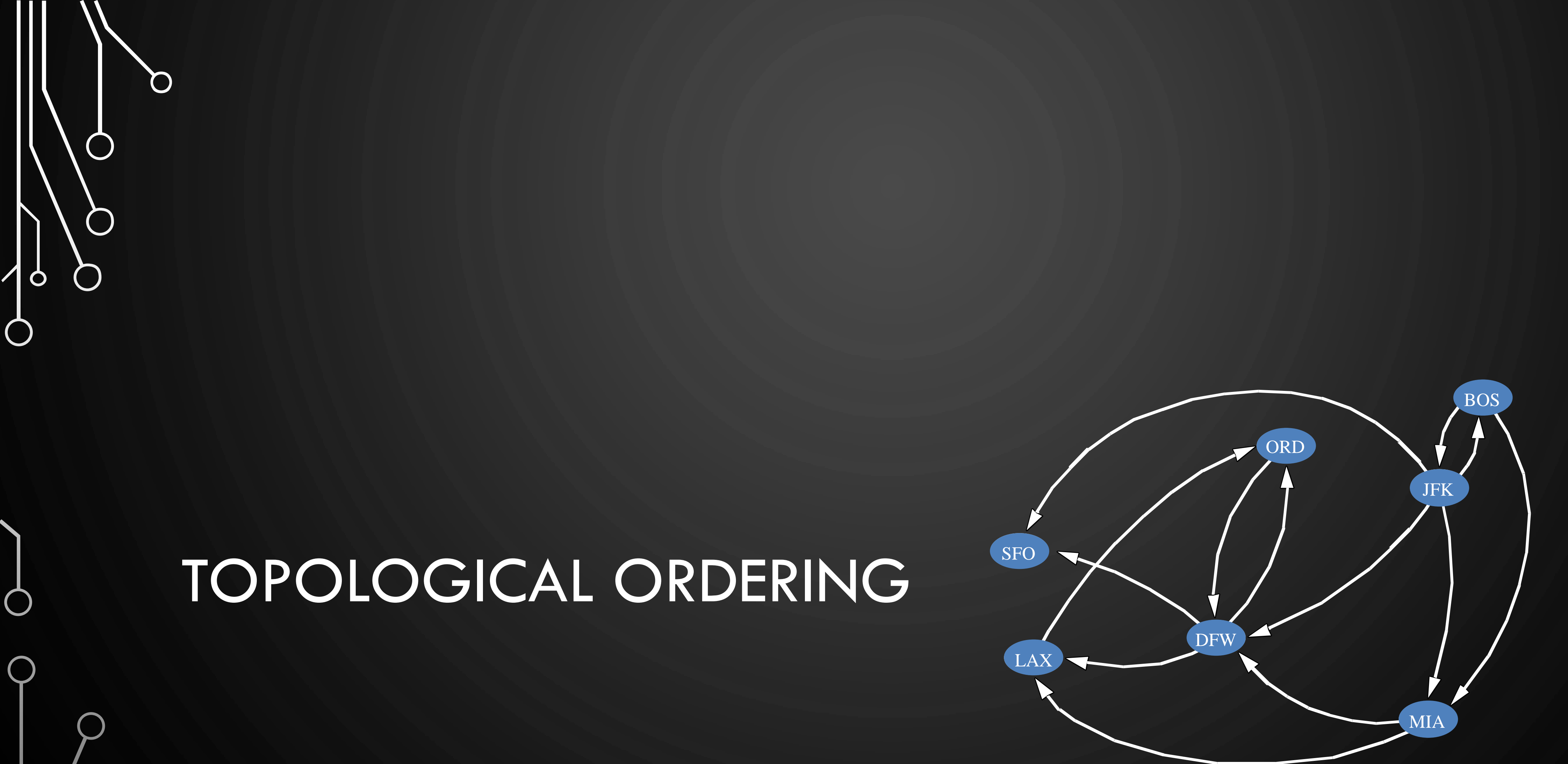


DFS



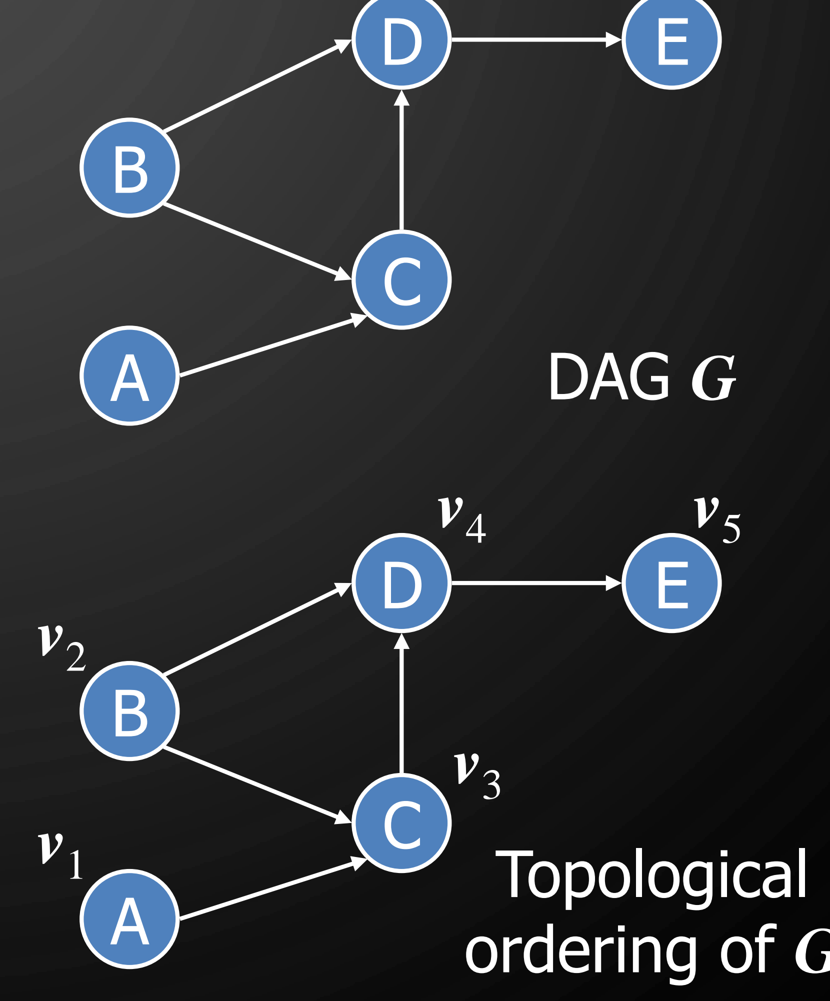
BFS

TOPOLOGICAL ORDERING



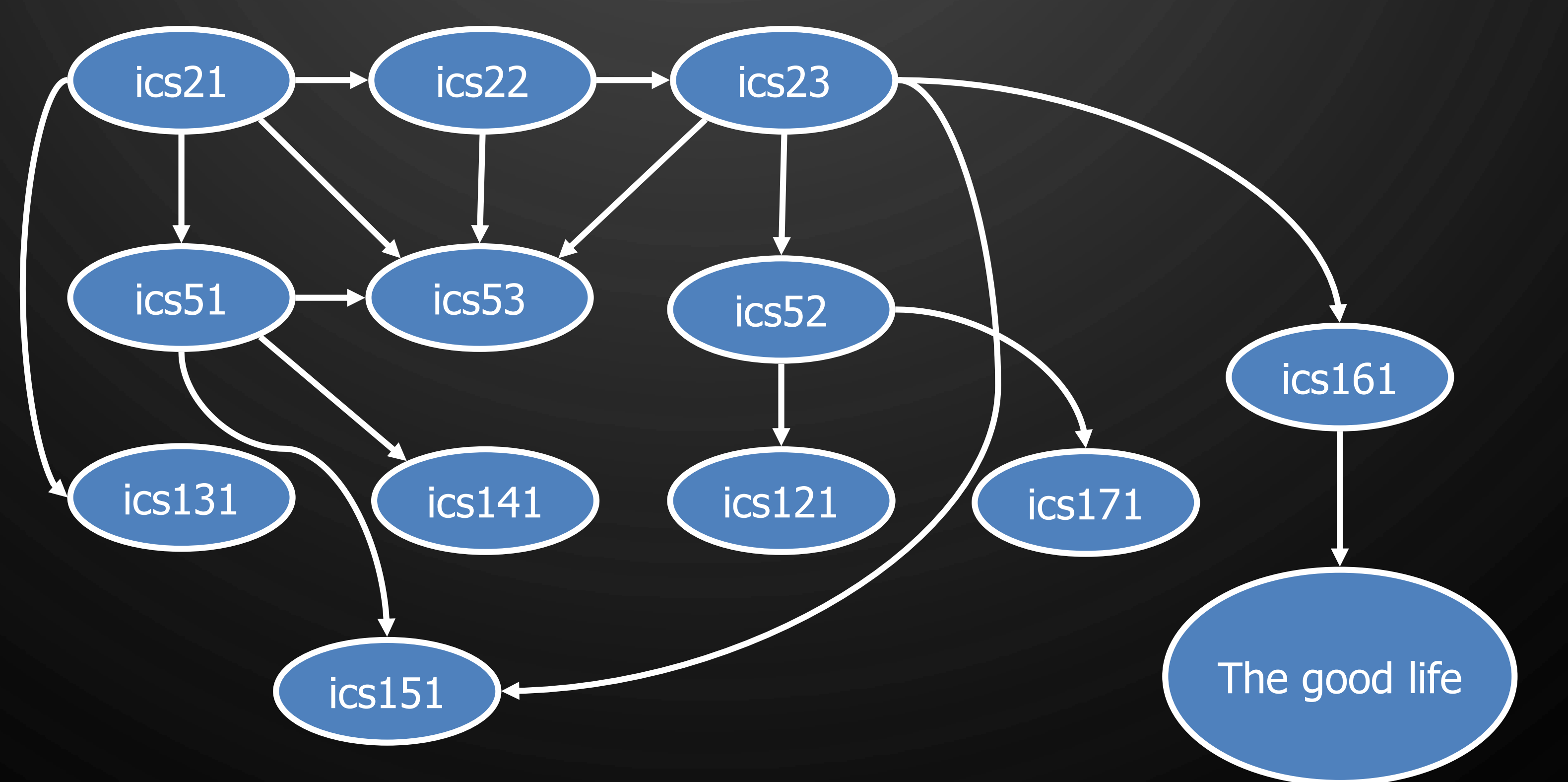
DAGS AND TOPOLOGICAL ORDERING

- A **directed acyclic graph (DAG)** is a digraph that has no directed cycles
- A topological ordering of a digraph is a numbering
    - $v_1, \dots, v_n$
    - Of the vertices such that for every edge (v_i, v_j), we have $i < j$
- Example: in a task scheduling digraph, a topological ordering a task sequence that satisfies the precedence constraints
- Theorem - A digraph admits a topological ordering if and only if it is a DAG

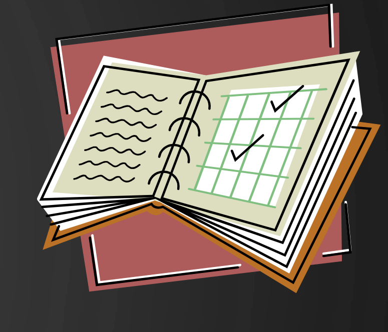


APPLICATION

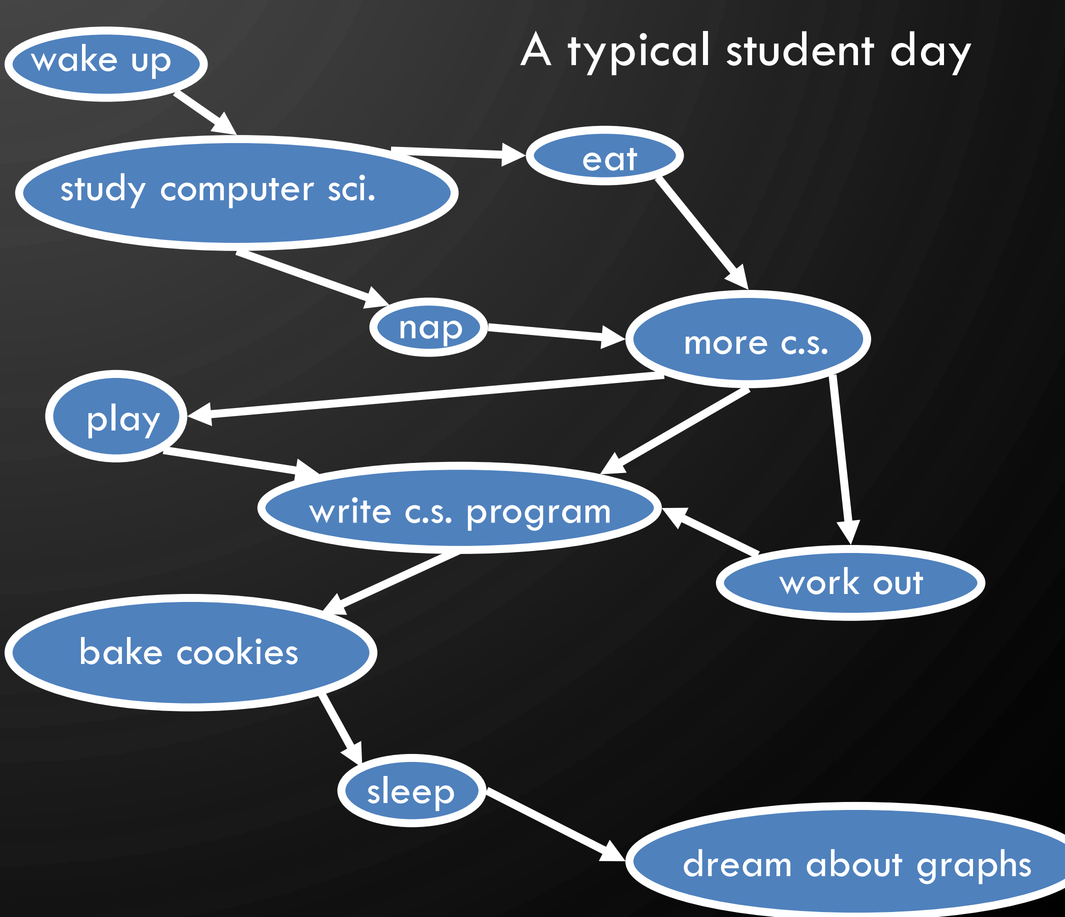
- Scheduling: edge (a, b) means task a must be completed before b can be started



EXERCISE
TOPOLOGICAL SORTING

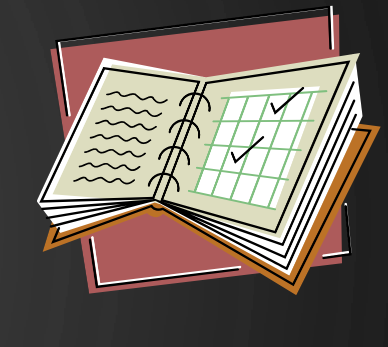


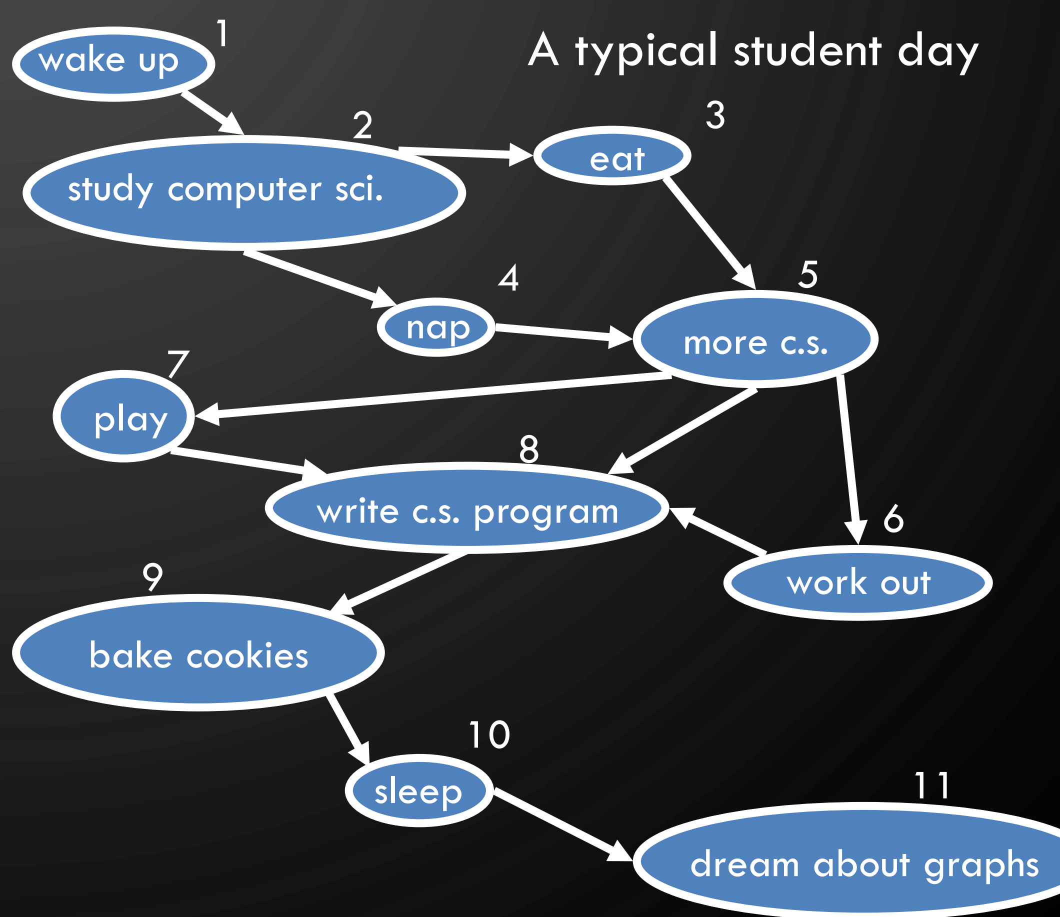
- Number vertices, so that (u, v) in E implies $u < v$



EXERCISE
TOPOLOGICAL SORTING

- Number vertices, so that (u, v) in E implies $u < v$





ALGORITHM FOR TOPOLOGICAL SORTING

```
Algorithm TopologicalSort(G)
Input: Directed Acyclic Graph (DAG) G
Output: Topological ordering of G
1.   H ← G
2.   n ← G.numVertices()
3.   while ¬H.isEmpty() do
4.      Let v be a vertex with no outgoing edges
5.      Label v ← n
6.      n ← n − 1
7.      H.removeVertex(v)
```

IMPLEMENTATION WITH DFS

- Simulate the algorithm by using depth-first search
- $O(n+m)$ time.

```
Algorithm topologicalDFS(G)
Input: DAG G
Output: Topological ordering of G
1.  n ← G.numVertices()
2.  Initialize all vertices as
    UNEXPLORED
3.  for each vertex v ∈ G.vertices() do
4.    if getLabel(v) = UNEXPLORED then
5.      topologicalDFS(G,v)
```

```
Algorithm topologicalDFS(G,v)
Input: DAG G, start vertex v
Output: Labeling of the vertices of G
        in the connected component of v
1.  setLabel(v,VISITED)
2.  for each e ∈ G.outgoingEdges(v) do
3.    w ← G.opposite(v,e)
4.    if getLabel(w) = UNEXPLORED then
5.      //e is a discovery edge
6.      topologicalDFS(G,w)
7.    else
8.      //e is a forward, cross, or back edge
9.  Label v with topological number n
10. n ← n − 1
```

TOPOLOGICAL SORTING EXAMPLE

TOPOLOGICAL SORTING EXAMPLE

9

TOPOLOGICAL SORTING EXAMPLE

8

9

TOPOLOGICAL SORTING EXAMPLE

7

8

9

TOPOLOGICAL SORTING EXAMPLE

6
7
8
9

TOPOLOGICAL SORTING EXAMPLE

6

5

7

8

9

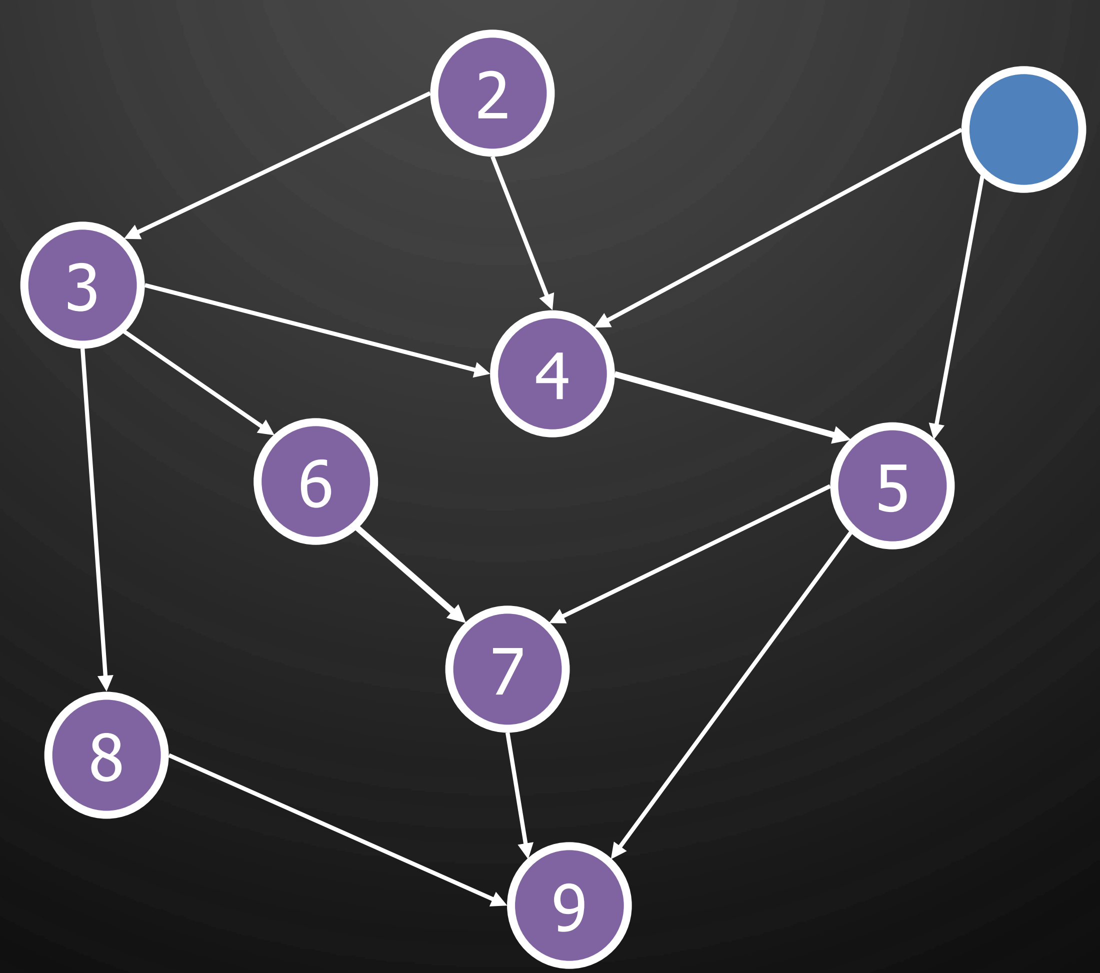
TOPOLOGICAL SORTING EXAMPLE

4
6
5
7
8
9

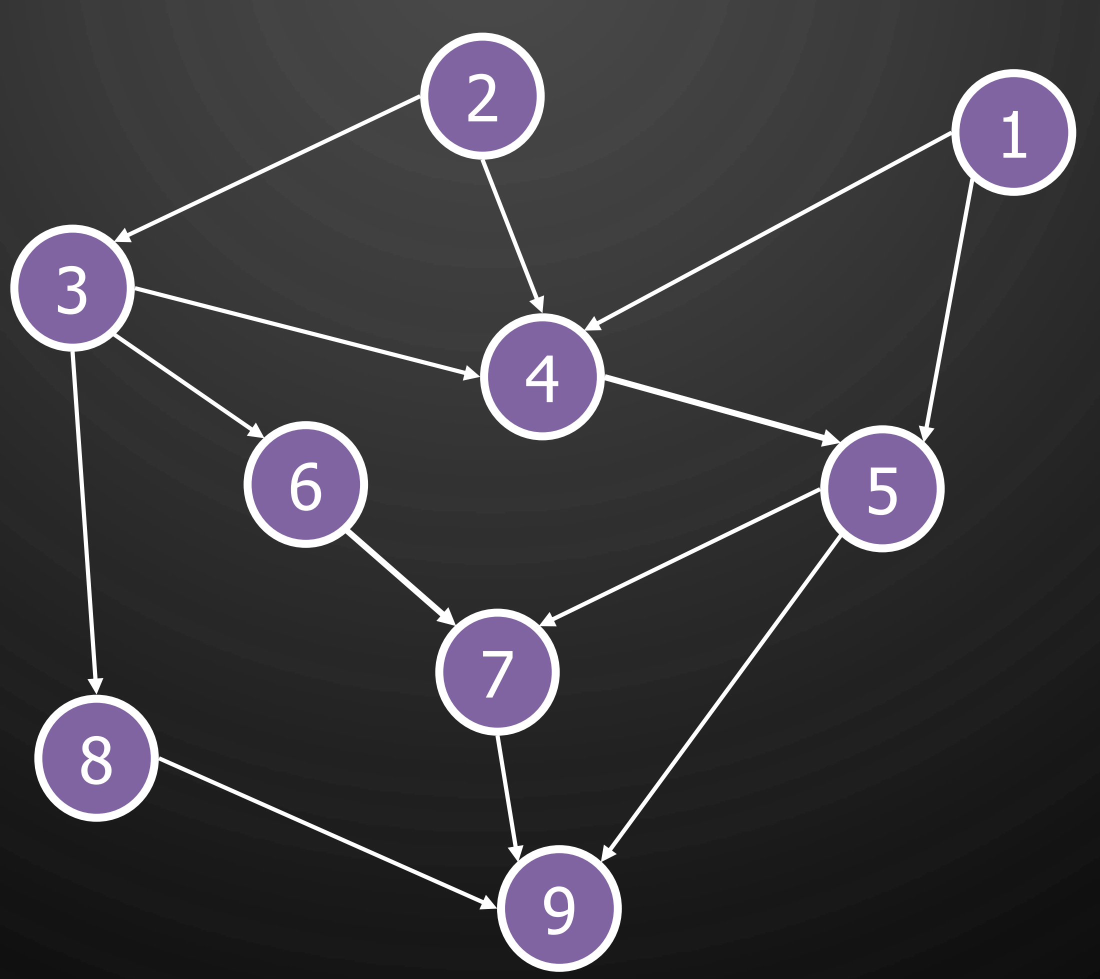
TOPOLOGICAL SORTING EXAMPLE

3
4
6
5
7
8
9

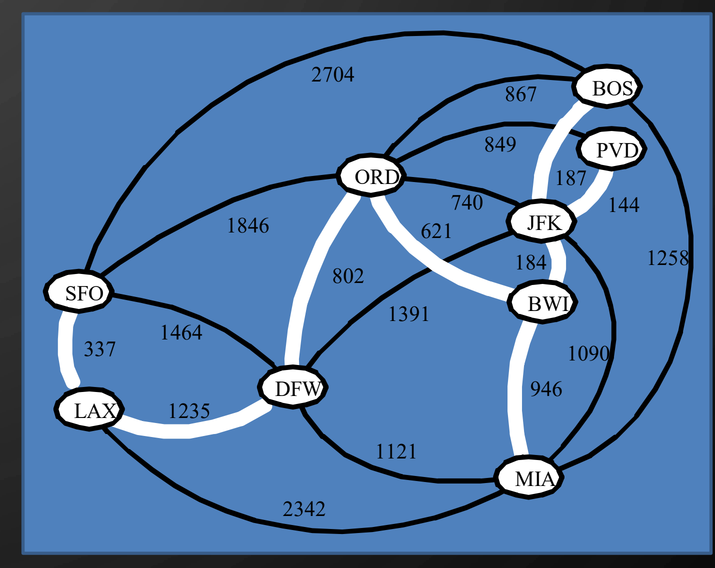
TOPOLOGICAL SORTING EXAMPLE



TOPOLOGICAL SORTING EXAMPLE

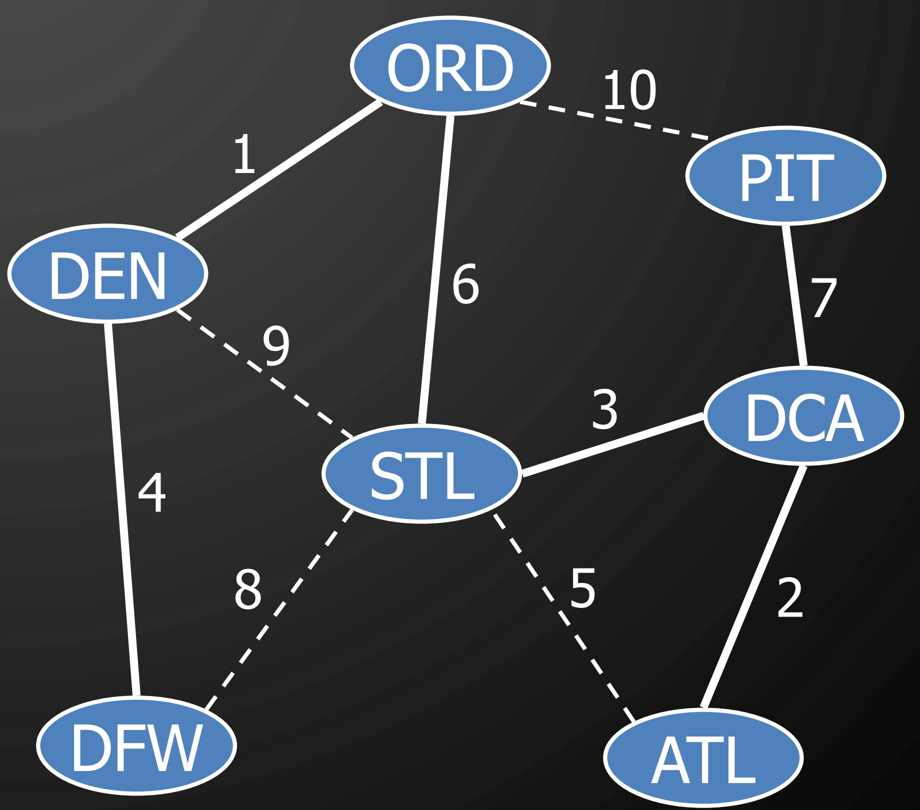


MINIMUM SPANNING TREES



MINIMUM SPANNING TREE

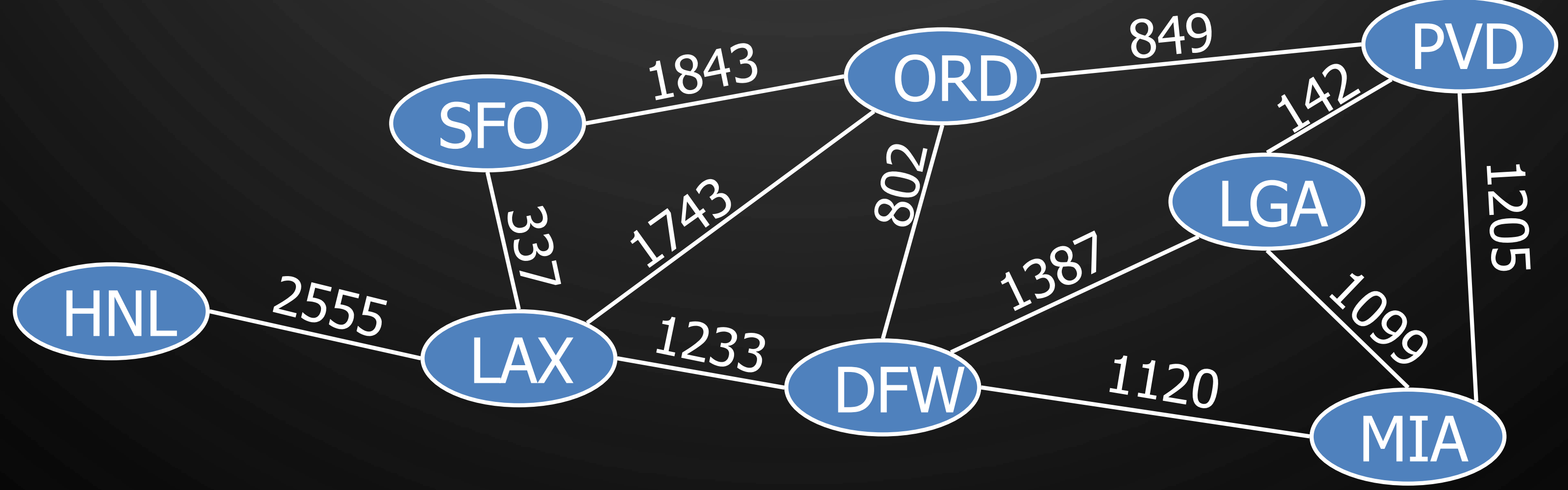
- Minimum spanning tree (MST)
    - Spanning tree of a weighted graph with minimum total edge weight
- Applications
    - Communications networks
    - Transportation networks



EXERCISE

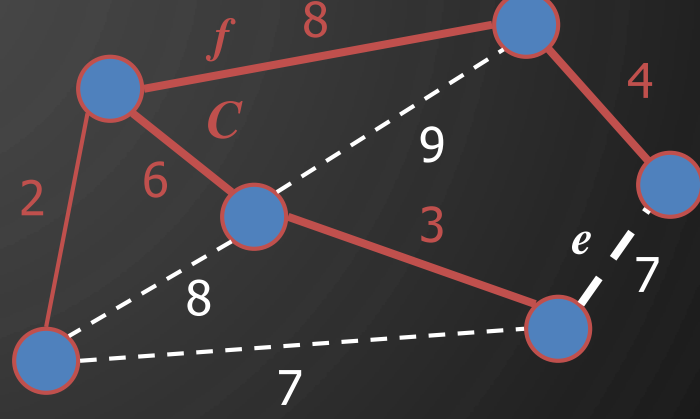
MST

- Show an MST of the following graph.

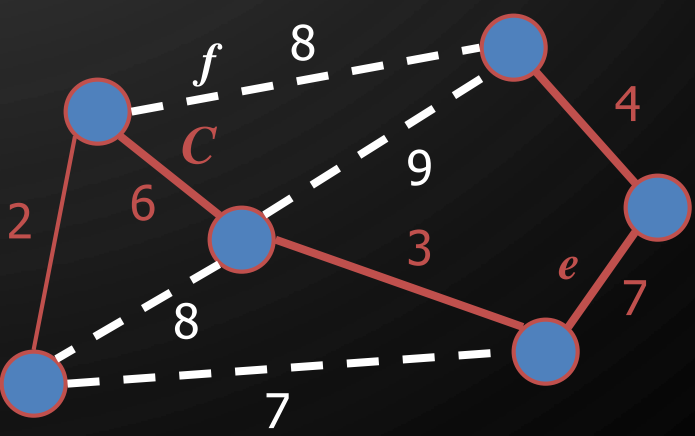


CYCLE PROPERTY

- **Cycle Property:**
  - Let T be a minimum spanning tree of a weighted graph G
  - Let e be an edge of G that is not in T and C let be the cycle formed by e with T
  - For every edge f of C,
    $weight(f) \leq weight(e)$
- Proof by contradiction:
  - If $weight(f) > weight(e)$ we can get a spanning tree of smaller weight by replacing e with f

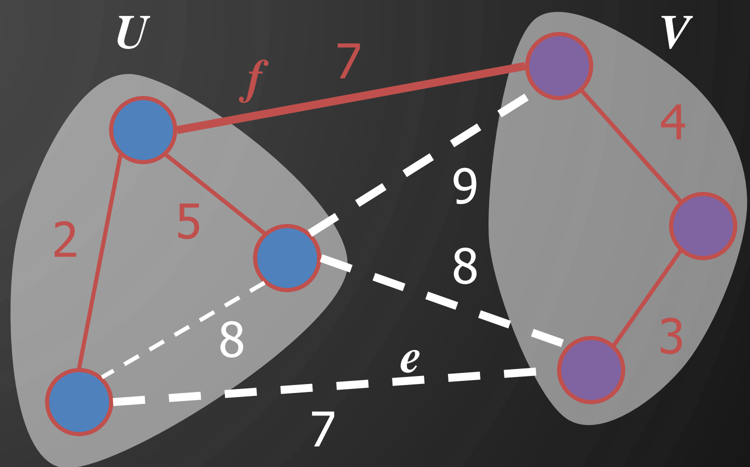


Replacing f with e yields a better spanning tree

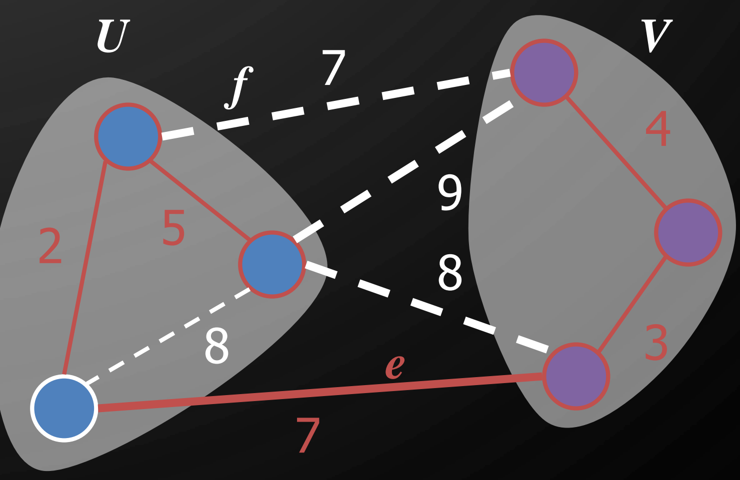


PARTITION PROPERTY

- **Partition Property:**
  - Consider a partition of the vertices of G into subsets U and V
  - Let e be an edge of minimum weight across the partition
  - There is a minimum spanning tree of G containing edge e
- Proof by contradition:
  - Let T be an MST of G
  - If T does not contain e, consider the cycle C formed by e with T and let f be an edge of C across the partition
  - By the cycle property,
    $$weight(f) \leq weight(e)$$
  - Thus, $weight(f) = weight(e)$
  - We obtain another MST by replacing f with e

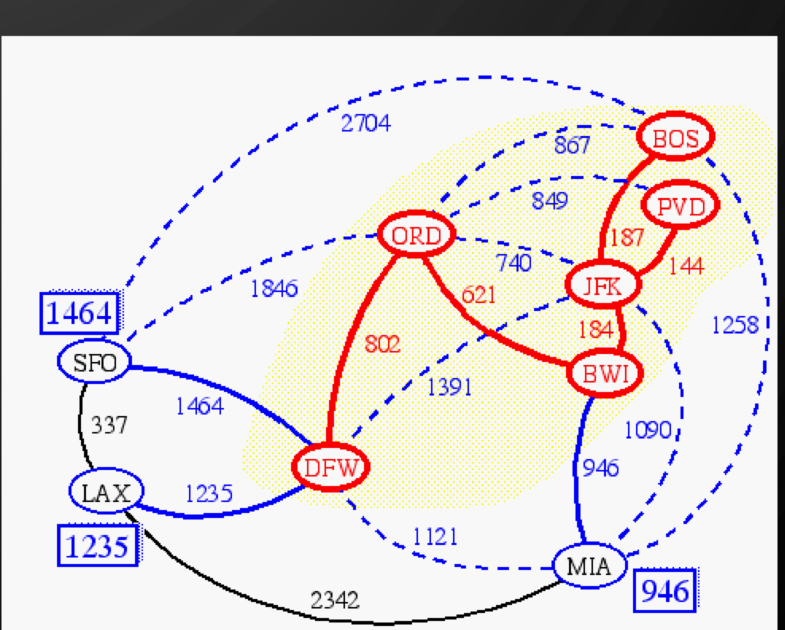


Replacing f with e yields another MST



PRIM-JARNIK'S ALGORITHM

- We pick an arbitrary vertex s and we grow the MST as a cloud of vertices, starting from s
- We store with each vertex v a label $d(v)$ representing the smallest weight of an edge connecting v to a vertex in the cloud
- At each step:
  - We add to the cloud the vertex u outside the cloud with the smallest distance label
  - We update the labels of the vertices adjacent to u

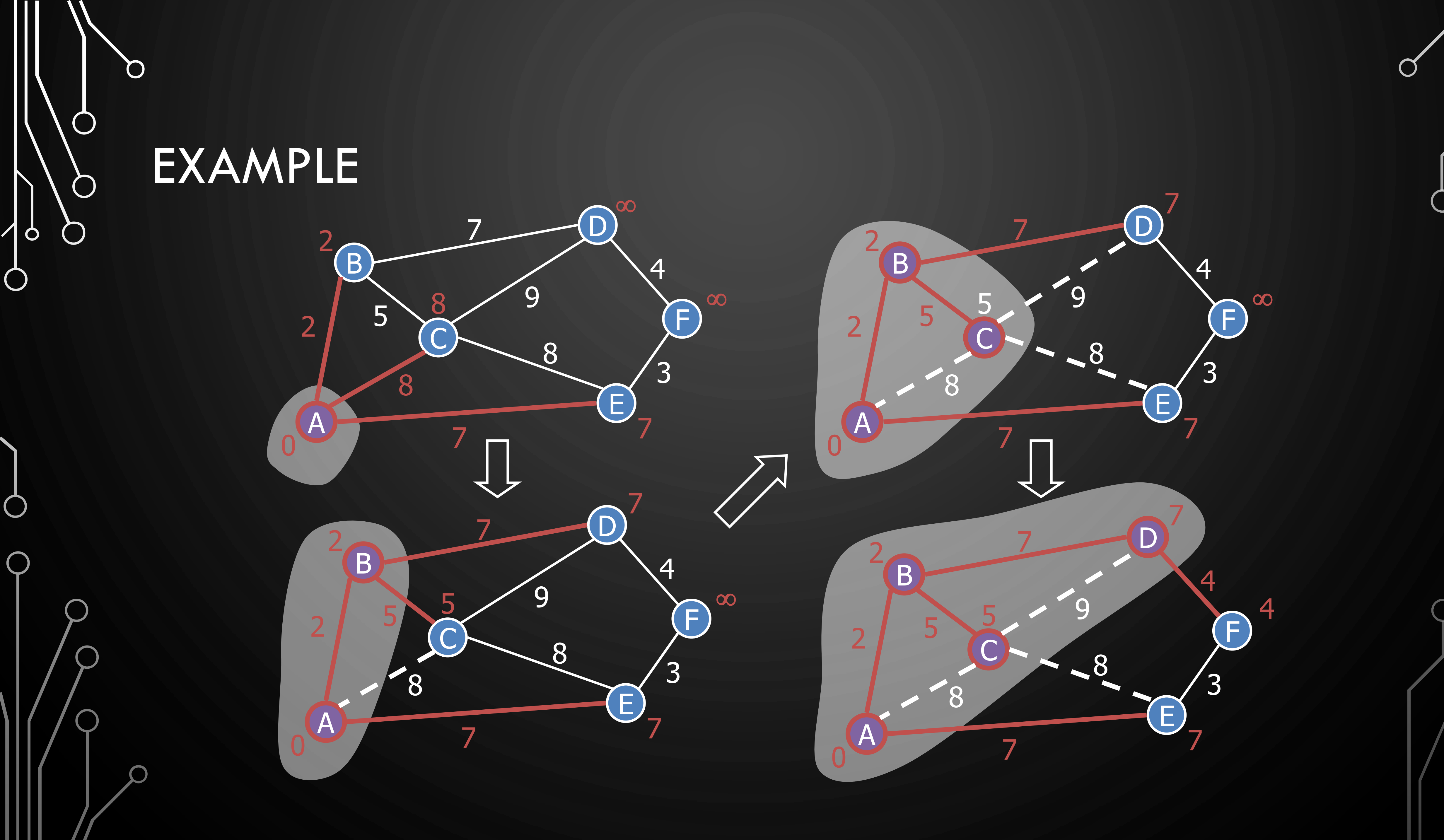


PRIM-JARNIK'S ALGORITHM

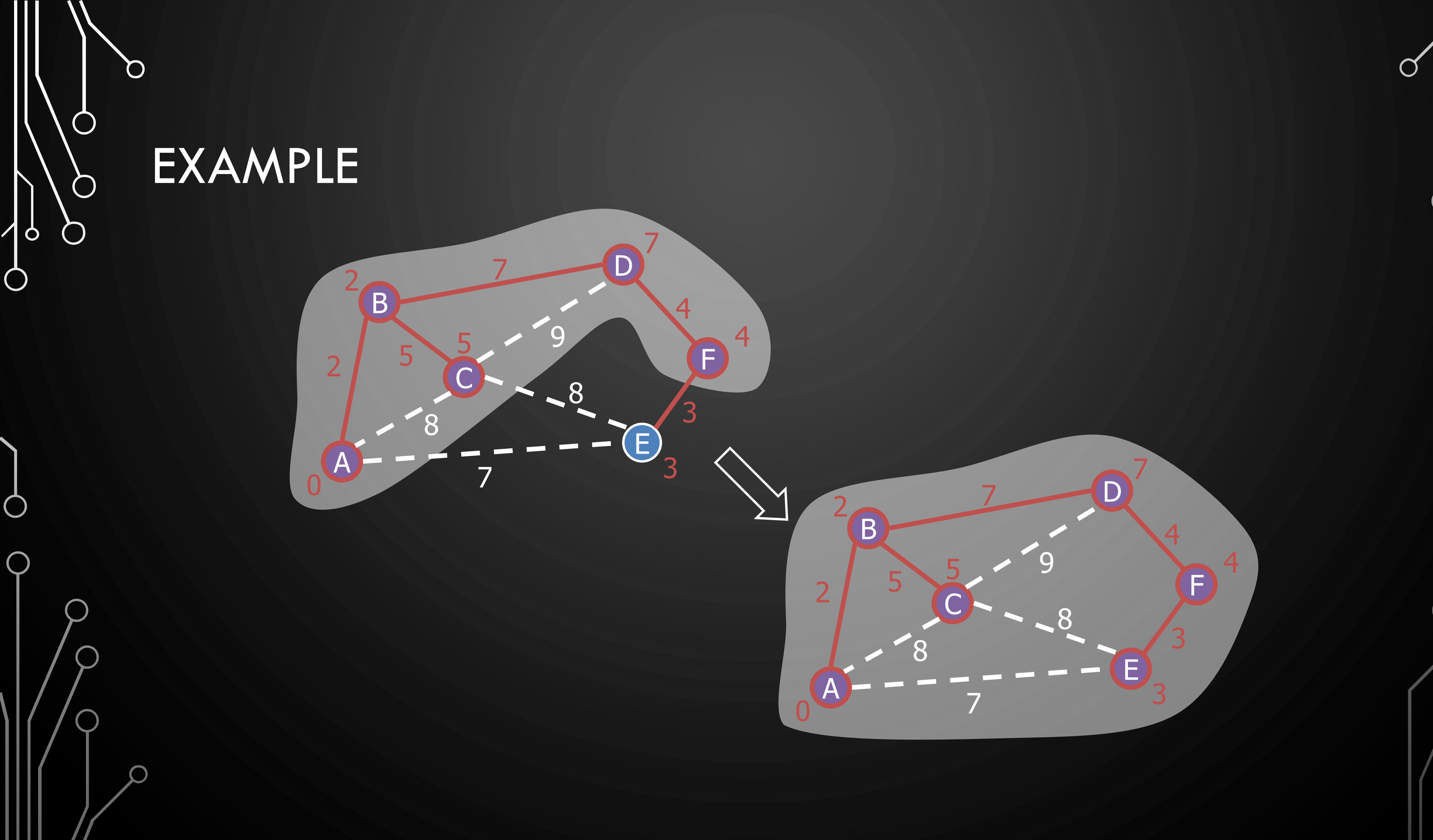
- An adaptable priority queue stores the vertices outside the cloud
  - Key: distance, $D[v]$
  - Element: vertex v
  - $Q.replace(i, k)$ changes the key of an item
- We store three labels with each vertex v:
  - Distance $D[v]$
  - Parent edge in MST $P[v]$
  - Locator in priority queue

```
Algorithm PrimJarnikMST(G)
Input: A weighted connected graph G
Output: A minimum spanning tree T of G
1.  Pick any vertex s of G
2.  D[s] ← 0;  P[s] ← ∅
3.  for each vertex v ≠ s do
4.    D[v] ← ∞;  P[v] ← ∅
5.  T ← ∅
6.  Priority queue Q of vertices with
    D[v] as the key
7.  while ¬Q.isEmpty() do
8.    u ← Q.removeMin()
9.    Add vertex u and edge P[u] to T
10.   for each e ∈ u.outgoingEdges() do
11.     v ← G.opposite(u, e)
12.     if e.weight() < D[v] then
13.       D[v] ← e.weight();  P[v] ← e
14.       Q.replace(v, D[v])
15. return T
```

EXAMPLE

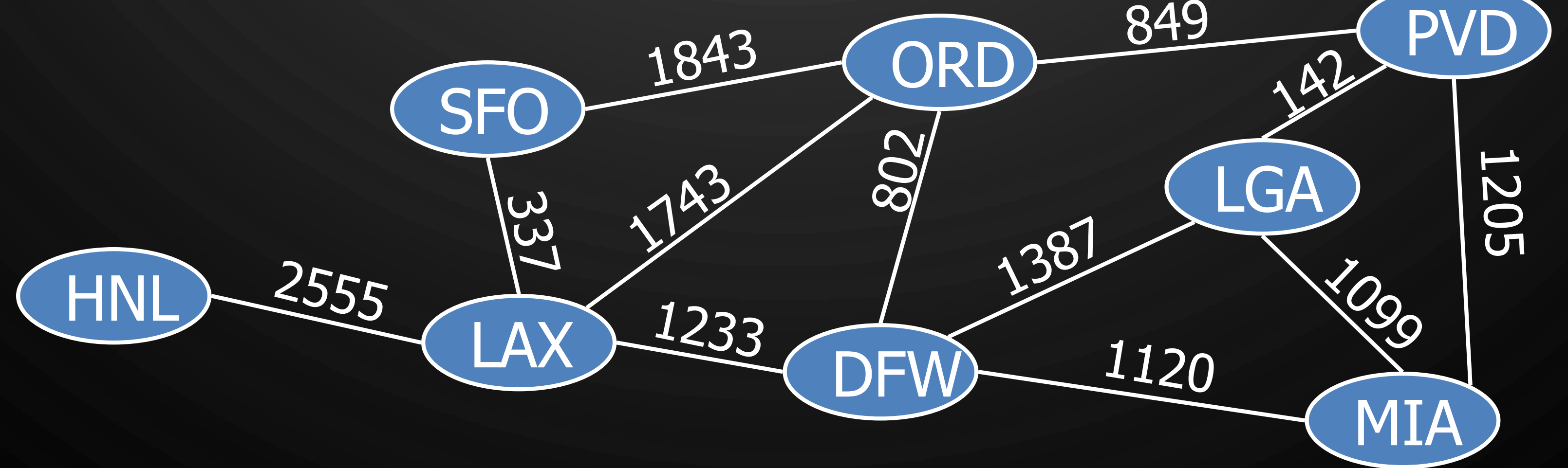


EXAMPLE



EXERCISE
PRIM'S MST ALGORITHM

- Show how Prim's MST algorithm works on the following graph, assuming you start with SFO
  - Show how the MST evolves in each iteration.



ANALYSIS

- Graph operations
  - Method incidentEdges is called once for each vertex
- Label operations
  - We set/get the distance, parent and locator labels of vertex z $O(\deg(z))$ times
  - Setting/getting a label takes $O(1)$ time
- Priority queue operations
  - Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes $O(\log n)$ time
  - The key of a vertex w in the priority queue is modified at most $\deg(w)$ times, where each key change takes $O(\log n)$ time
- Prim-Jarnik's algorithm runs in $O\big((n+m)\log n\big)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\Sigma_v \deg(v) = 2m$
- If the graph is connected the running time is $O(m \log n)$

EXAM 3

- Hack sheet – you can have a single 8 ½" x 11" paper with handwritten notes on both sides with you during the exam. You can put anything on it, but summary slides and the generic tree traversal algorithms make great candidates.
- No Java language/programming questions. Lecture material only.
- Format – 5 questions and a bonus
  - Q1 – Fill-in-the-blank questions (similar to quizzes)
  - Q2 – Short answer with cumulative material
  - Q3 – Sorting and divide-and-conquer (similar to homework)
  - Q4 – Write and/or analyze algorithm using Graph ADT (similar to homework)
  - Q5 – Write and/or analyze algorithm using Graph ADT (similar to homework)
  - Bonus – ?