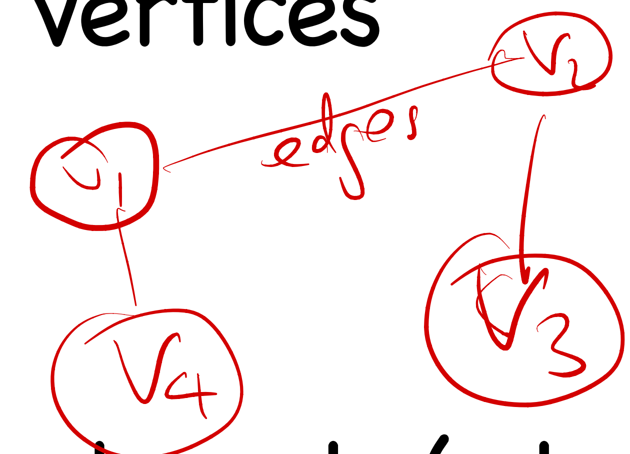
assuming UG knowledge of Graphs

Intro to graphs

Minimum Spanning Trees

Graphs

- nodes/vertices and edges between vertices
  - set V for vertices, set E for edges
  - we write graph G = (V,E)

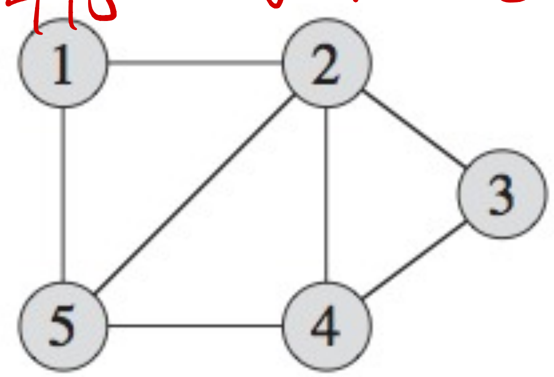


- example : cities on a map (nodes) and roads (edges)

Adjacency matrix

- $a_{ij} = 1$ if there is an edge from vertex i to vertex j
- if graph is undirected, edges go both ways, and the adj. matrix is symmetric

1,2,3,4,5 = vertices ids



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

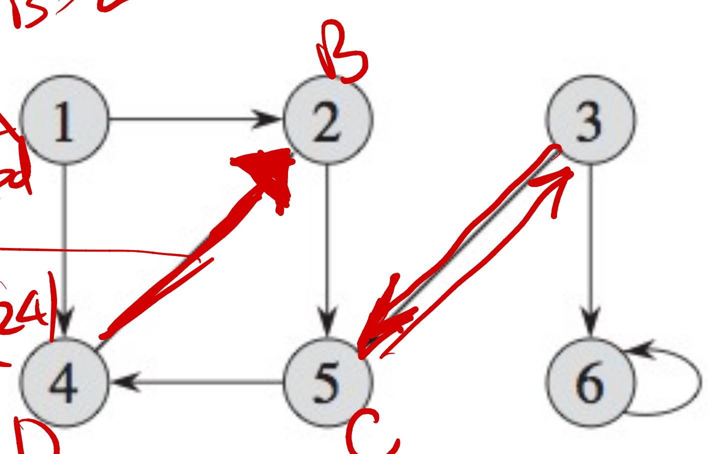
edge (2,3)
= edge (3,2)

- if the graph is directed, the adj. matrix is not necessarily symmetric

A=1 B=2

directed

(42) ≠ (24)



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	1	1	0	0
6	0	0	0	0	0	1

missing edge

edge (3,5)

edge (5,3)
added

edge
link
connect
} ⇒ undirected G

Adjacency lists

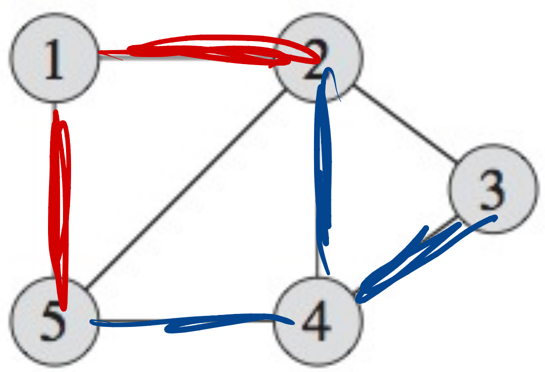
path, direction

edge

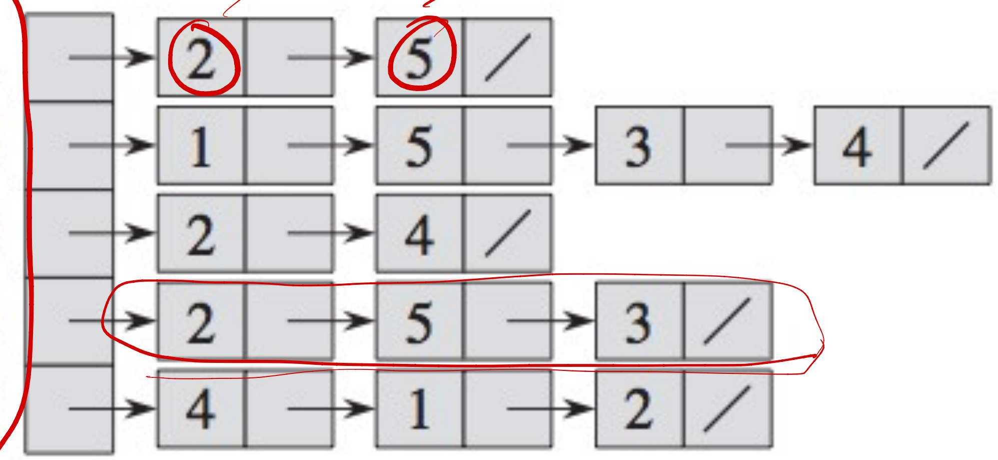
vertices

"linked" to 1

"reachable" from 1

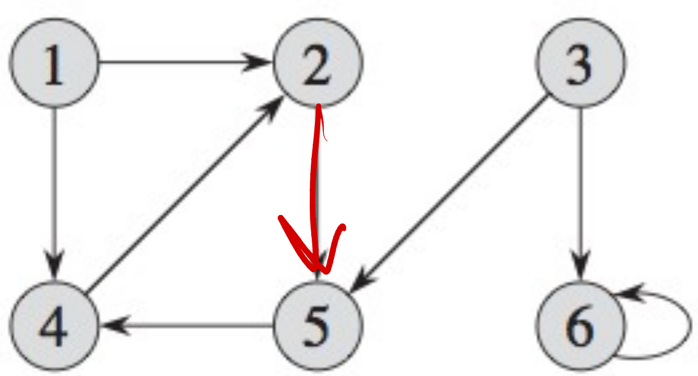


	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0



- linked list marks all edges starting off a given vertex

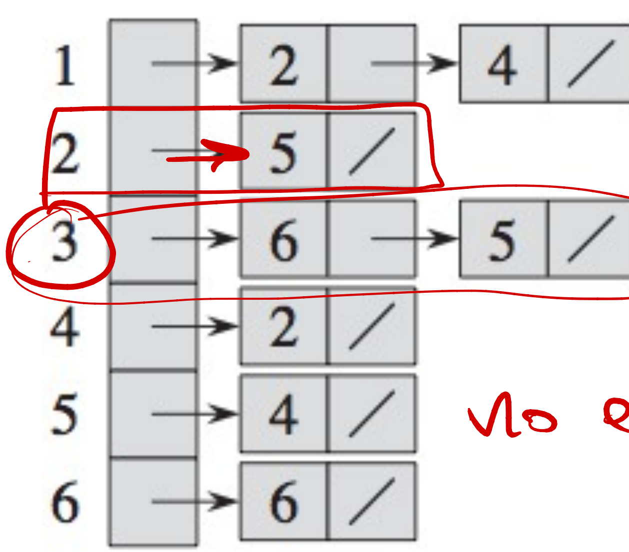
2 → 5



reachable = path
1 → 2 → 5; 1 ⇝ 5

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

edge (5,2) missing



3 → 6
3 → 5

no edge 5 → 2

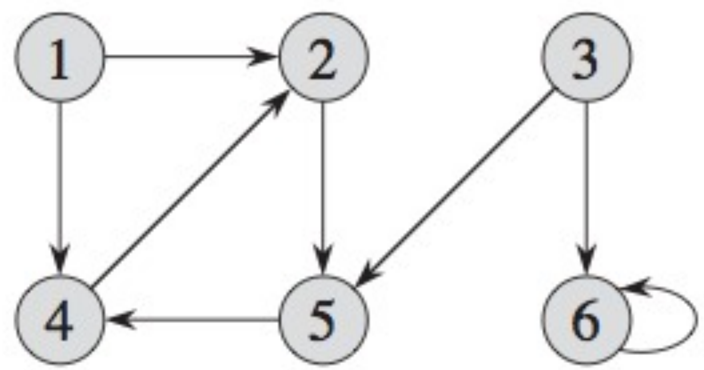
where can I go 1 step

backwards adj list : where can I come from 1 step.

1	null
2	1 4 null
3	null
4	5 1 null
5	2 3 null
6	3 6 null

paths and cycles

- path: a sequence of vertices $(v_1,v_2,v_3,\ldots,v_k)$ such that all (v_i,v_{i+1}) are edges in the graph

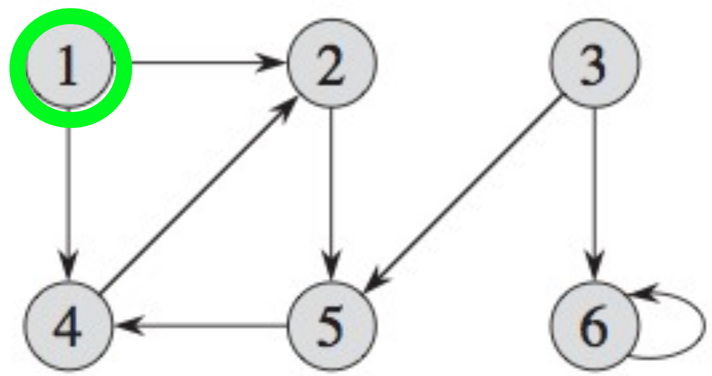


- edges can form a cycle = a path that ends in the same vertex it started

- paths and cycles are defined for both directed and undirected graphs

paths and cycles

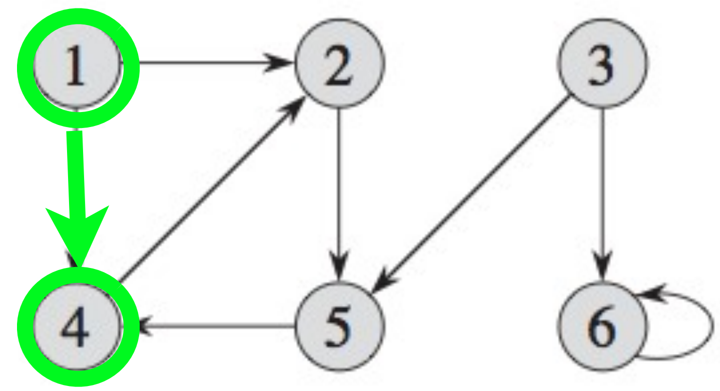
- path: a sequence of vertices $(v_1,v_2,v_3,\ldots,v_k)$ such that all (v_i,v_{i+1}) are edges in the graph



- edges can form a cycle = a path that ends in the same vertex it started
- paths and cycles are defined for both directed and undirected graphs

paths and cycles

- path: a sequence of vertices $(v_1,v_2,v_3,\ldots,v_k)$ such that all (v_i,v_{i+1}) are edges in the graph



- edges can form a cycle = a path that ends in the same vertex it started

- paths and cycles are defined for both directed and undirected graphs

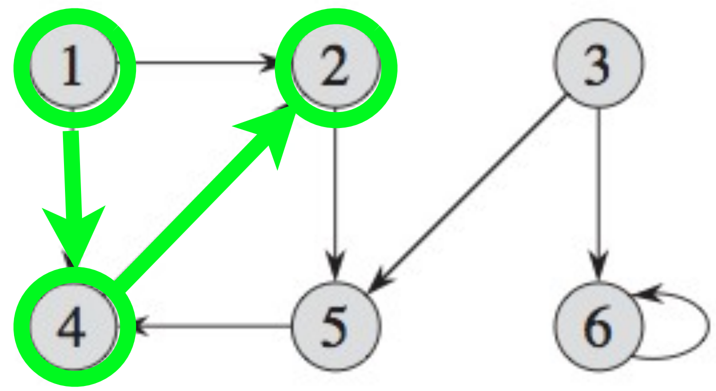
paths and cycles

- path: a sequence of vertices $(v_1,v_2,v_3,\ldots,v_k)$ such that all (v_i,v_{i+1}) are edges in the graph

$1 \rightsquigarrow 2$ two options

$1 \to 2$

$1 \to 4 \to 2$

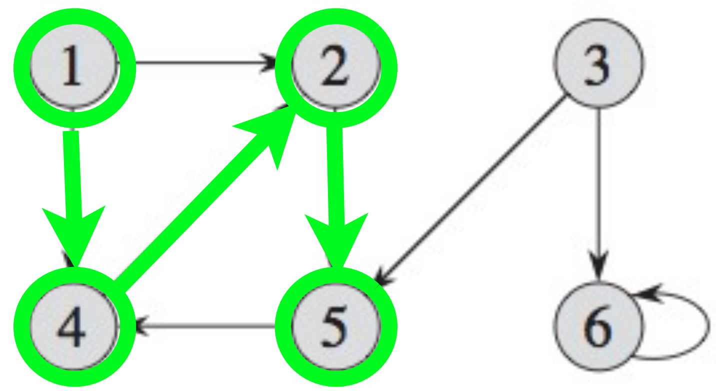


- edges can form a cycle = a path that ends in the same vertex it started

- paths and cycles are defined for both directed and undirected graphs

paths and cycles

- path: a sequence of vertices $(v_1,v_2,v_3,\ldots,v_k)$ such that all (v_i,v_{i+1}) are edges in the graph

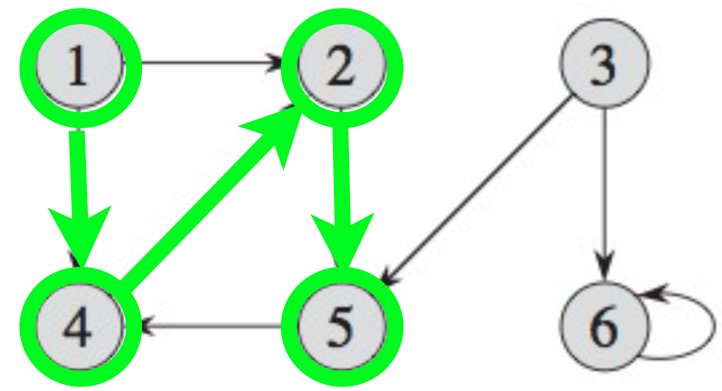


- edges can form a cycle = a path that ends in the same vertex it started

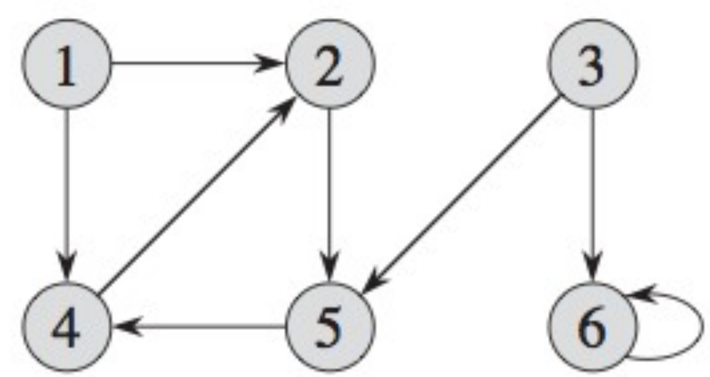
- paths and cycles are defined for both directed and undirected graphs

paths and cycles

- path: a sequence of vertices $(v_1,v_2,v_3,\ldots,v_k)$ such that all (v_i,v_{i+1}) are edges in the graph



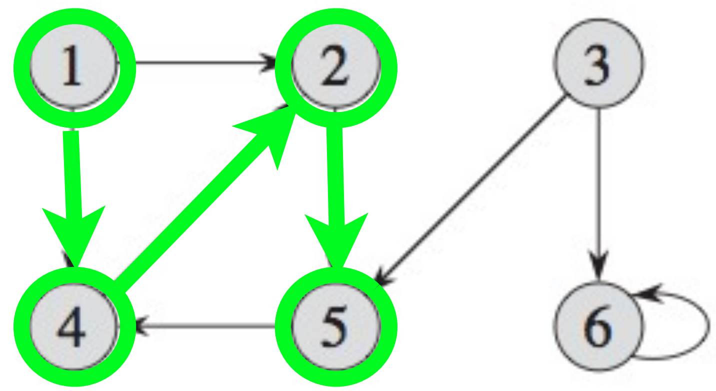
- edges can form a cycle = a path that ends in the same vertex it started



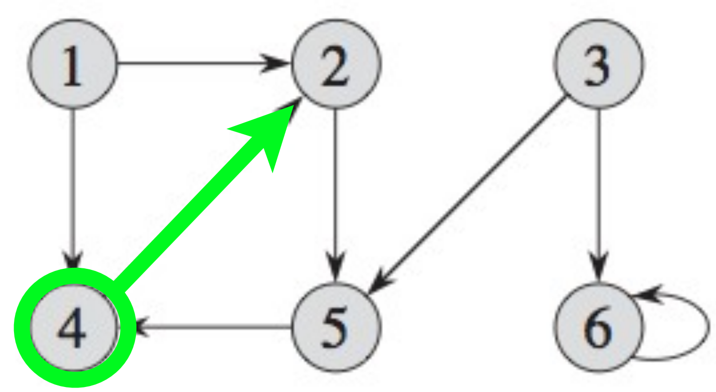
- paths and cycles are defined for both directed and undirected graphs

paths and cycles

- path: a sequence of vertices $(v_1,v_2,v_3,\ldots,v_k)$ such that all (v_i,v_{i+1}) are edges in the graph



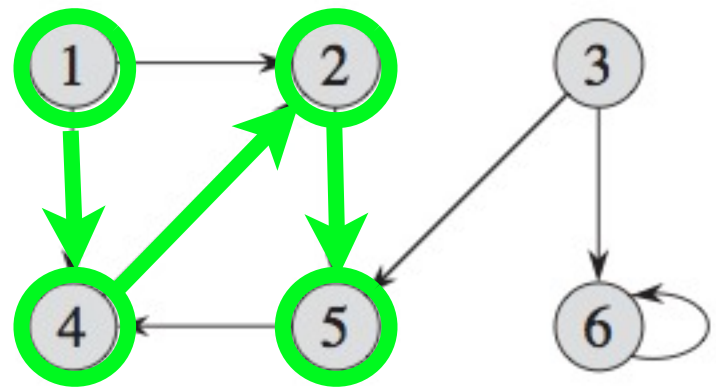
- edges can form a cycle = a path that ends in the same vertex it started



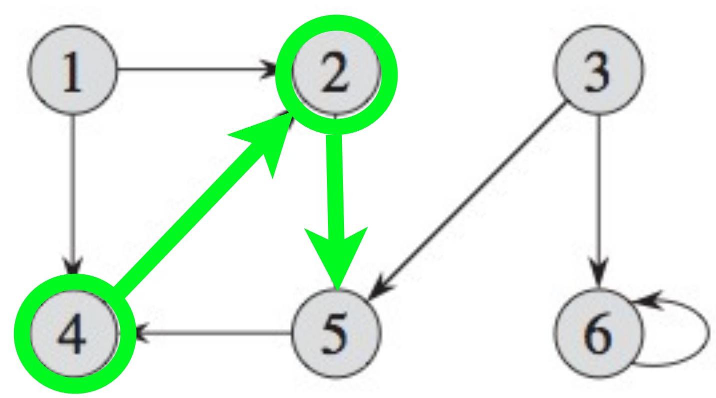
- paths and cycles are defined for both directed and undirected graphs

paths and cycles

- path: a sequence of vertices $(v_1,v_2,v_3,\ldots,v_k)$ such that all (v_i,v_{i+1}) are edges in the graph



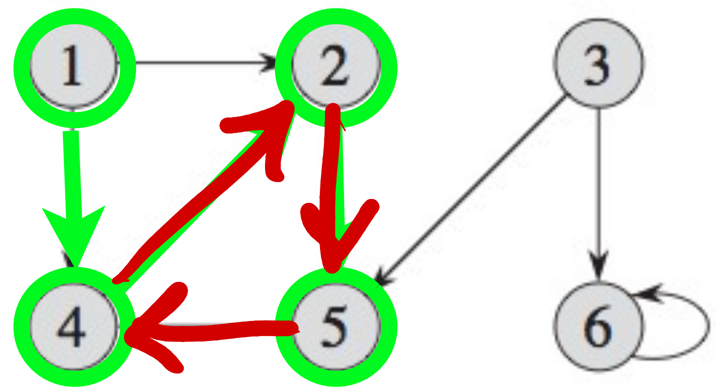
- edges can form a cycle = a path that ends in the same vertex it started



- paths and cycles are defined for both directed and undirected graphs

paths and cycles

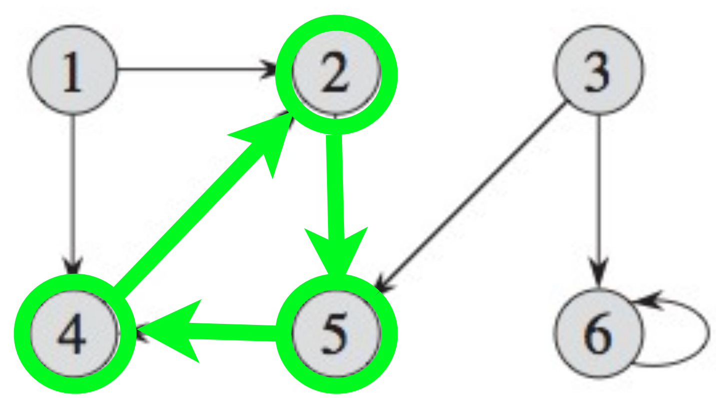
- path: a sequence of vertices ($v_1,v_2,v_3,\ldots,v_k$) such that all (v_i,v_{i+1}) are edges in the graph

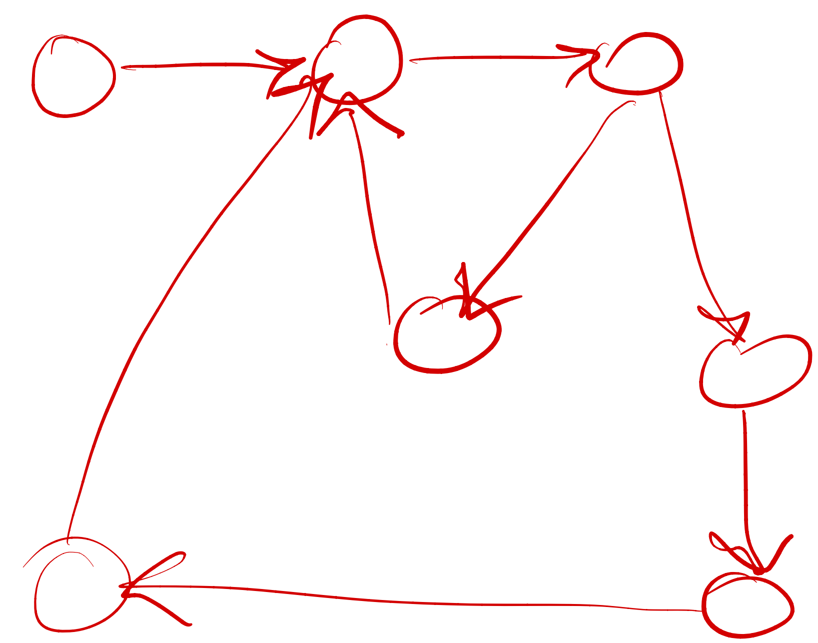


cycle path that ends where it started

simple cycle: does not repeat vertices

- edges can form a cycle = a path that ends in the same vertex it started

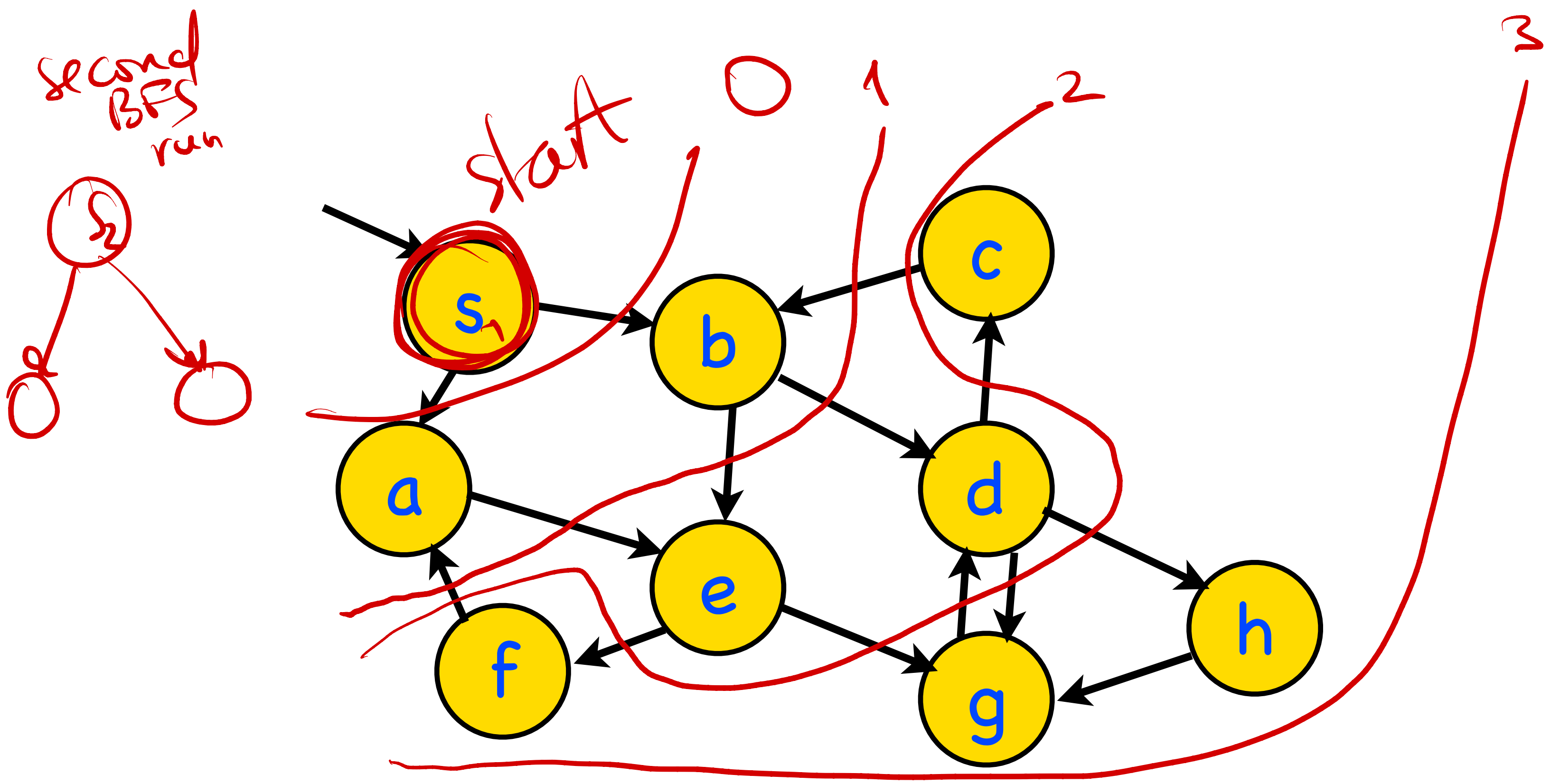




- paths and cycles are defined for both directed and undirected graphs

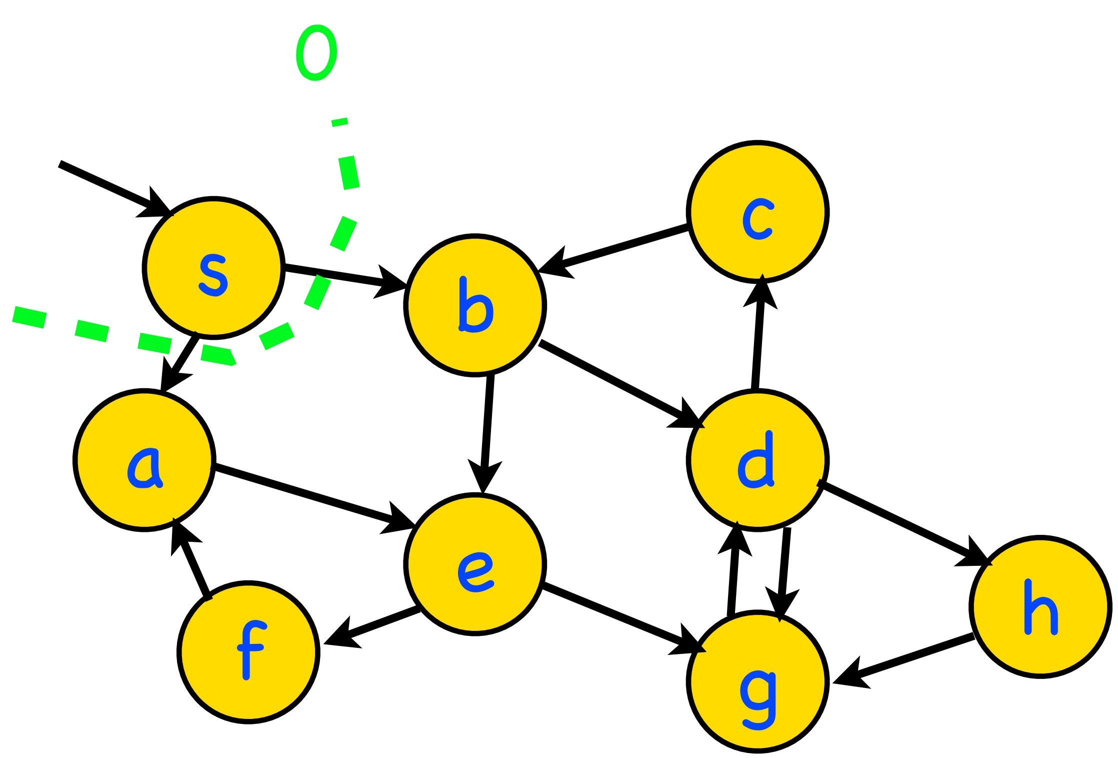
Traverse/search graphs : BFS

- BFS = breadth-first search.
- Start in a given vertex s, find all reachable vertices from s
  - proceed in waves partition of nodes reachable from s descendants(s)
  - computes d[v] = number of edges from s to v. If v not reachable from s, we have d[v] = ∞.



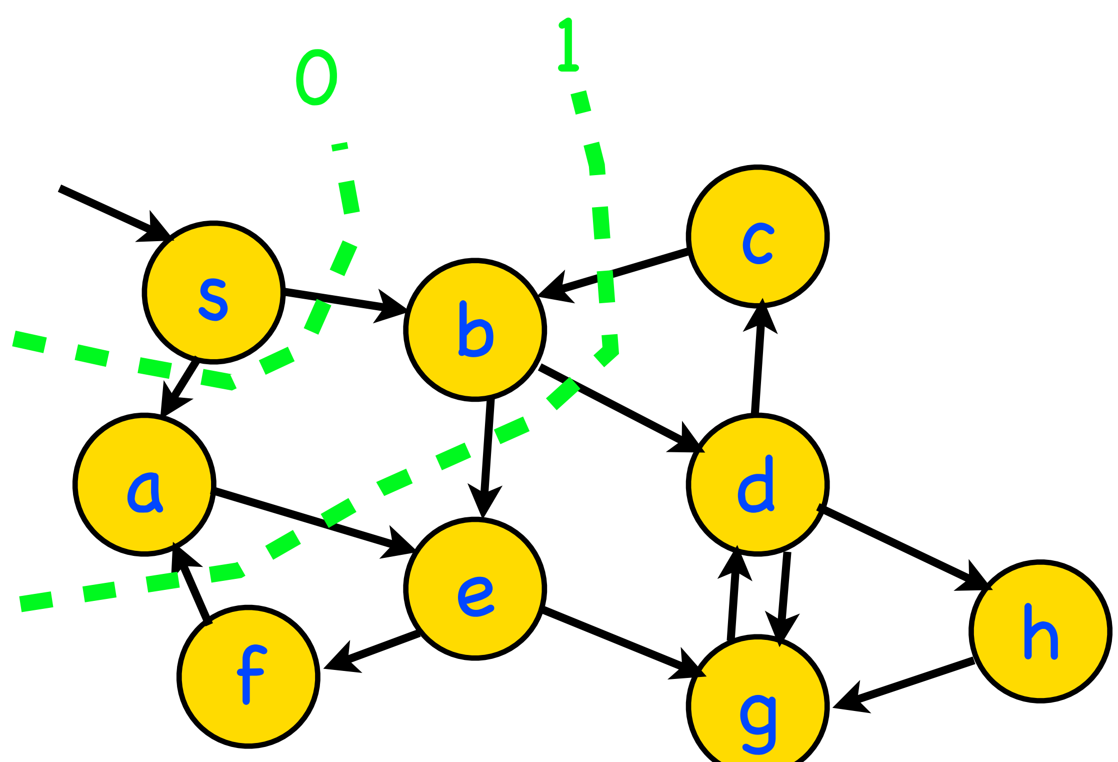
Traverse/search graphs : BFS

- BFS = breadth-first search.
- Start in a given vertex s, find all reachable vertices from s
  - proceed in waves
  - computes d[v] = number of edges from s to v. If v not reachable from s, we have d[v] = ∞.



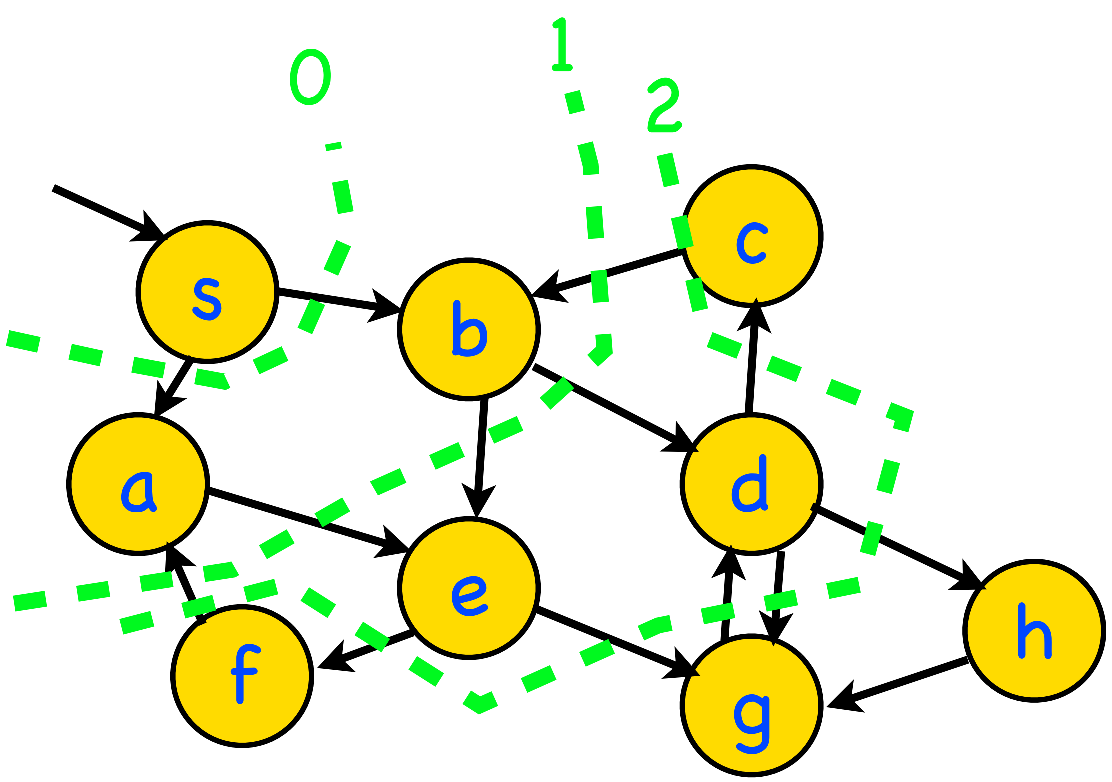
Traverse/search graphs : BFS

- BFS = breadth-first search.
- Start in a given vertex s, find all reachable vertices from s
  - proceed in waves
  - computes d[v] = number of edges from s to v. If v not reachable from s, we have d[v] = ∞.



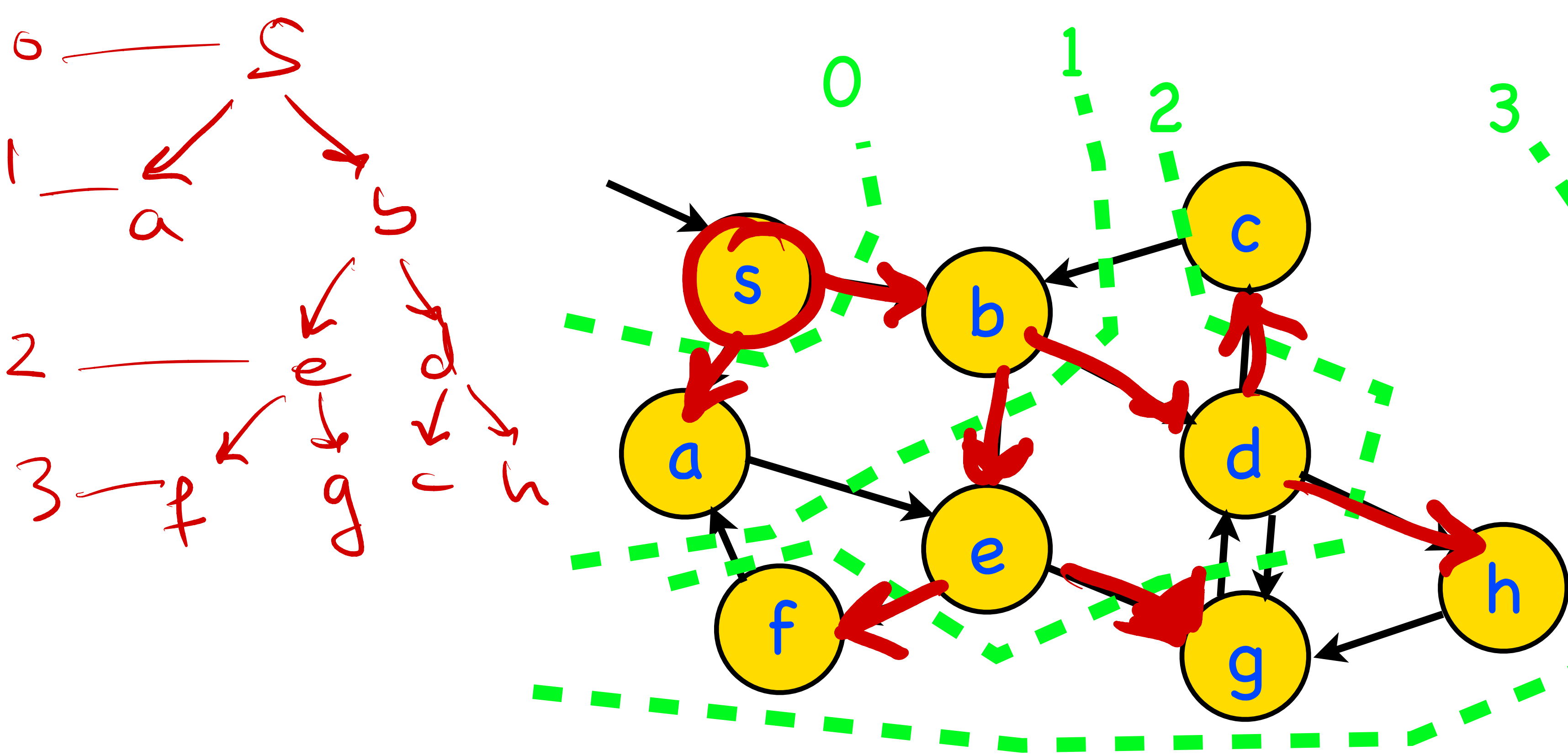
Traverse/search graphs : BFS

- BFS = breadth-first search.
- Start in a given vertex s, find all reachable vertices from s
  - proceed in waves
  - computes d[v] = number of edges from s to v. If v not reachable from s, we have d[v] = ∞.



Traverse/search graphs : BFS

- BFS = breadth-first search.
- Start in a given vertex s, find all reachable vertices from s
  - proceed in waves
  - computes d[v] = number of edges from s to v. If v not reachable from s, we have d[v] = ∞.



adv. edge =

= BFS edge

= edge (u.parent, u)

k k+1

BFS

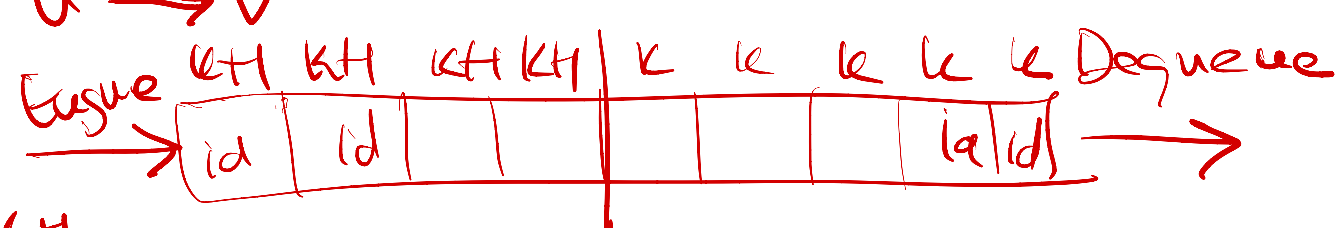
- use a queue to store processed vertices
  - for each vertex in the queue, follow adj matrix to get vertices of the next wave

```
BFS(V,E,s)
for each vertex v≠s, set d[v]=∞
init queue Q; enqueue(Q,s) //puts s in the queue
while Q not empty
    u = dequeue(S) // takes the first elem available from the queue
    for each vertex v ∈ Adj[u]
        if (d[v]==∞) then
            d[v]=d[u]+1
            Enqueue(Q,v)
        end if
    end for
end while
```

BFS

$d[v] = wave(v)$

$u \to v$



BFS queue
waves k, k+1

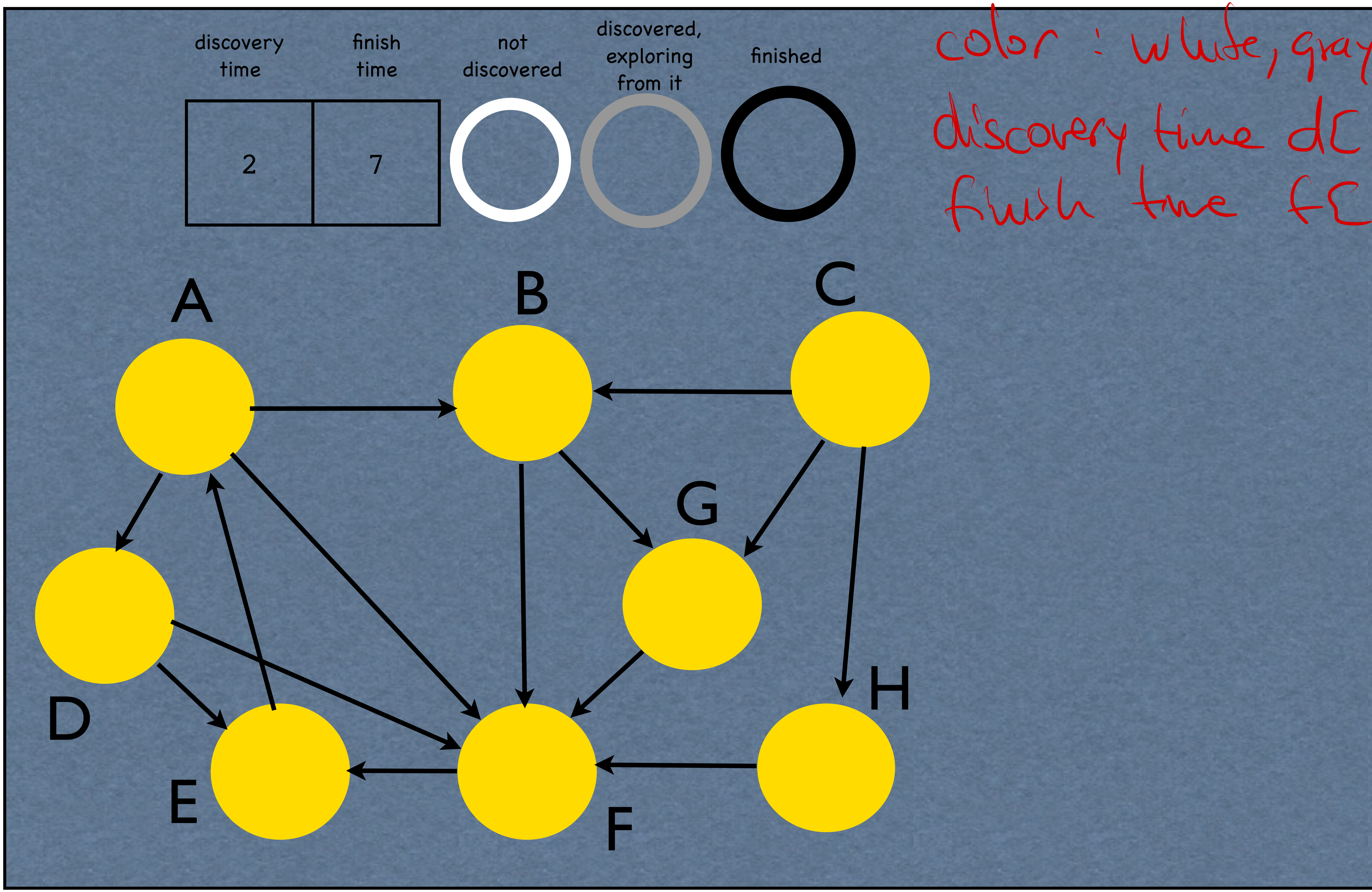
- Running time O(V+E), since each edge and vertex is considered once.

Traverse/search graphs : DFS

- DFS = depth-first search
  - once a vertex is discovered, proceed to its adj vertices, or "children"(depth) rather than to its "brothers" (breadth)

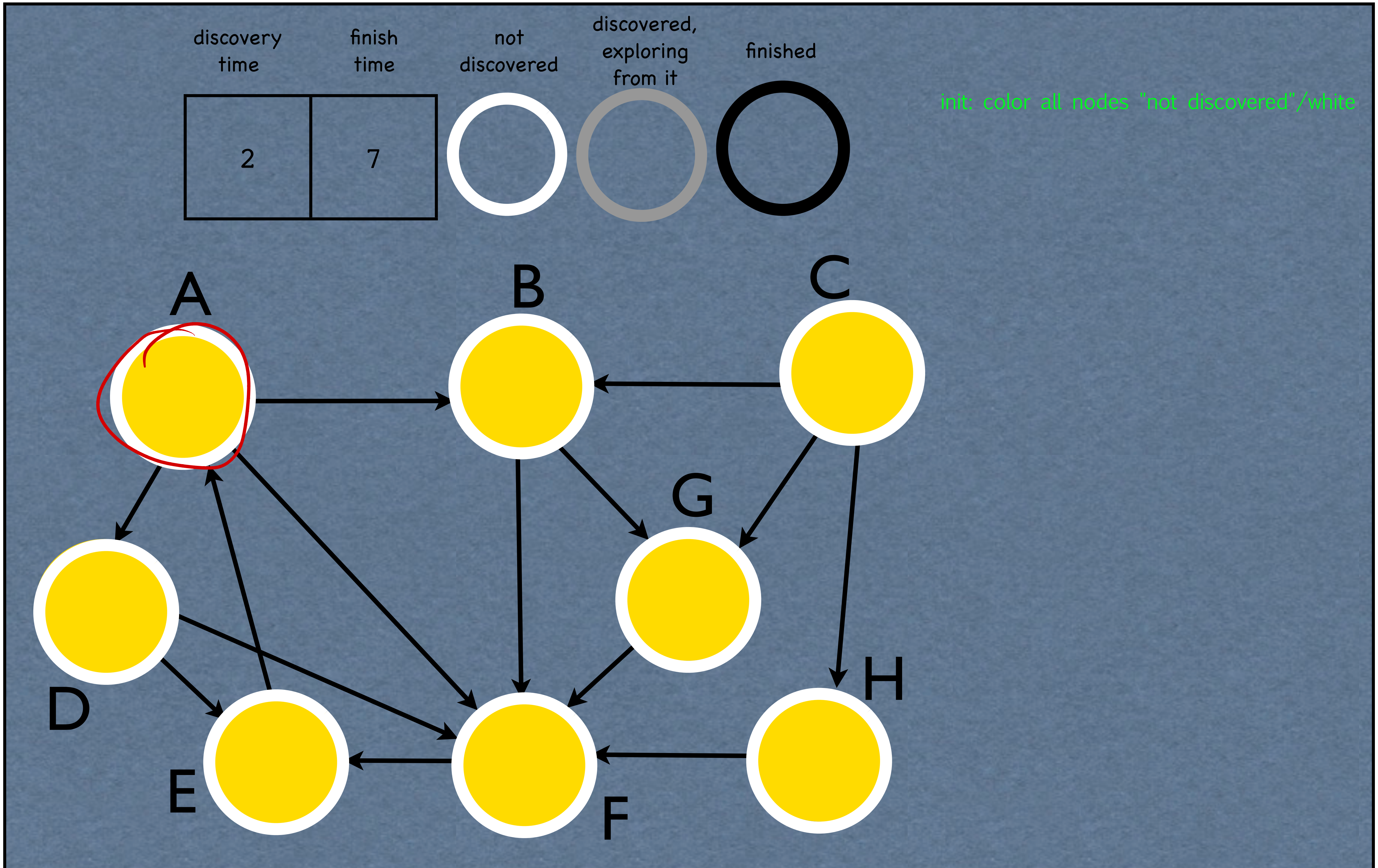
```
DFS-wrapper(V,E)
    foreach vertex u∈V {color[u] = white} end for //color all nodes white
    foreach vertex u∈V
        if (color[u]==white) then DFS-Visit(u)
    end for

DFS-Visit(u) //recursive function
    color[u] = gray; //gray means "exploring from this node"
    time++; discover_time[u] = time;//discover time
    for each v ∈ Adj[u]
        if (color[v]==white) then DFS-Visit(v)//explore from u
    end for
    color [u] = black; finish_time[u]=time; //finish time
```

DFS



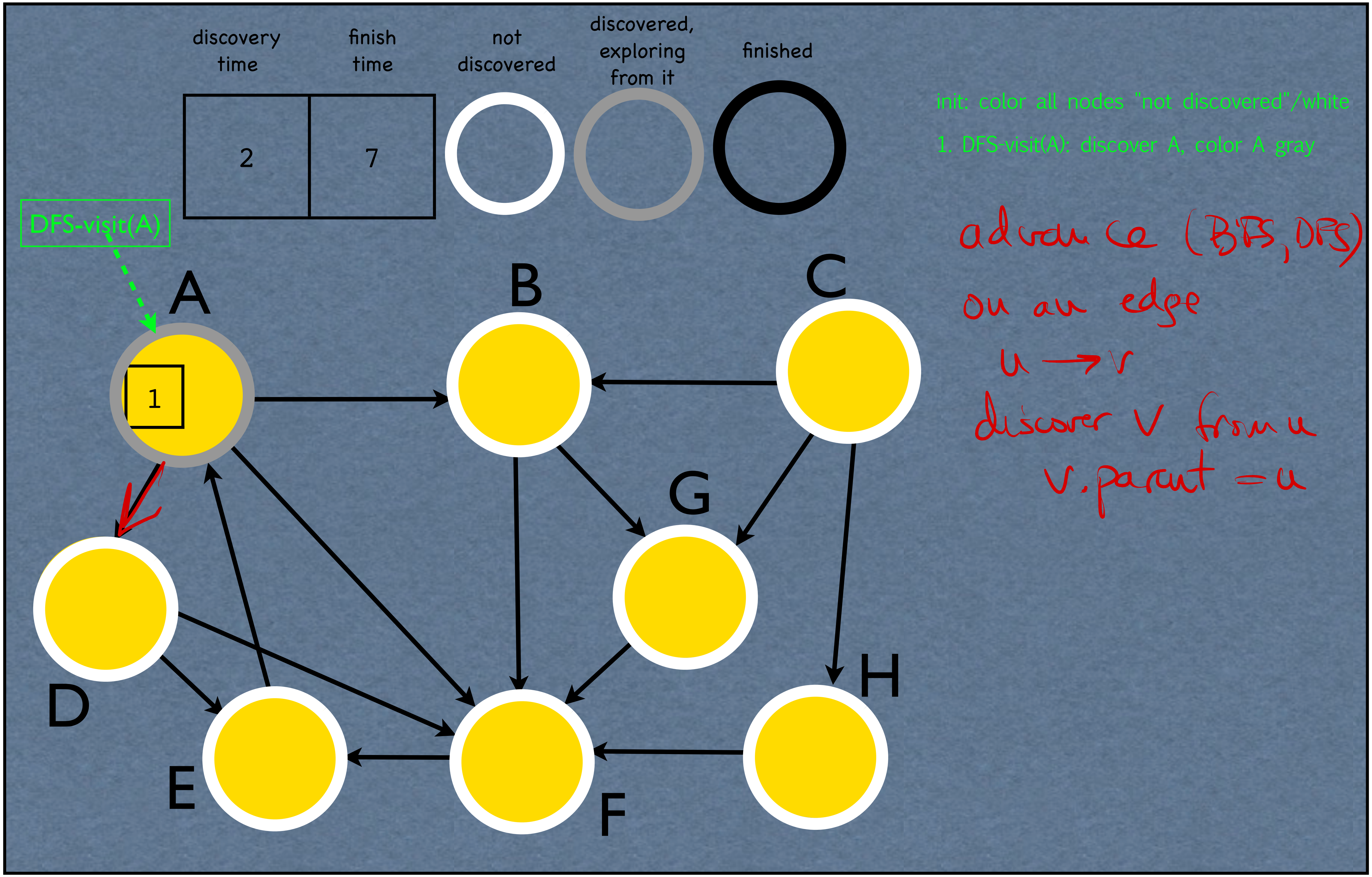
color : white, gray, black
discovery time d[ ]
finish time f[ ]

DFS

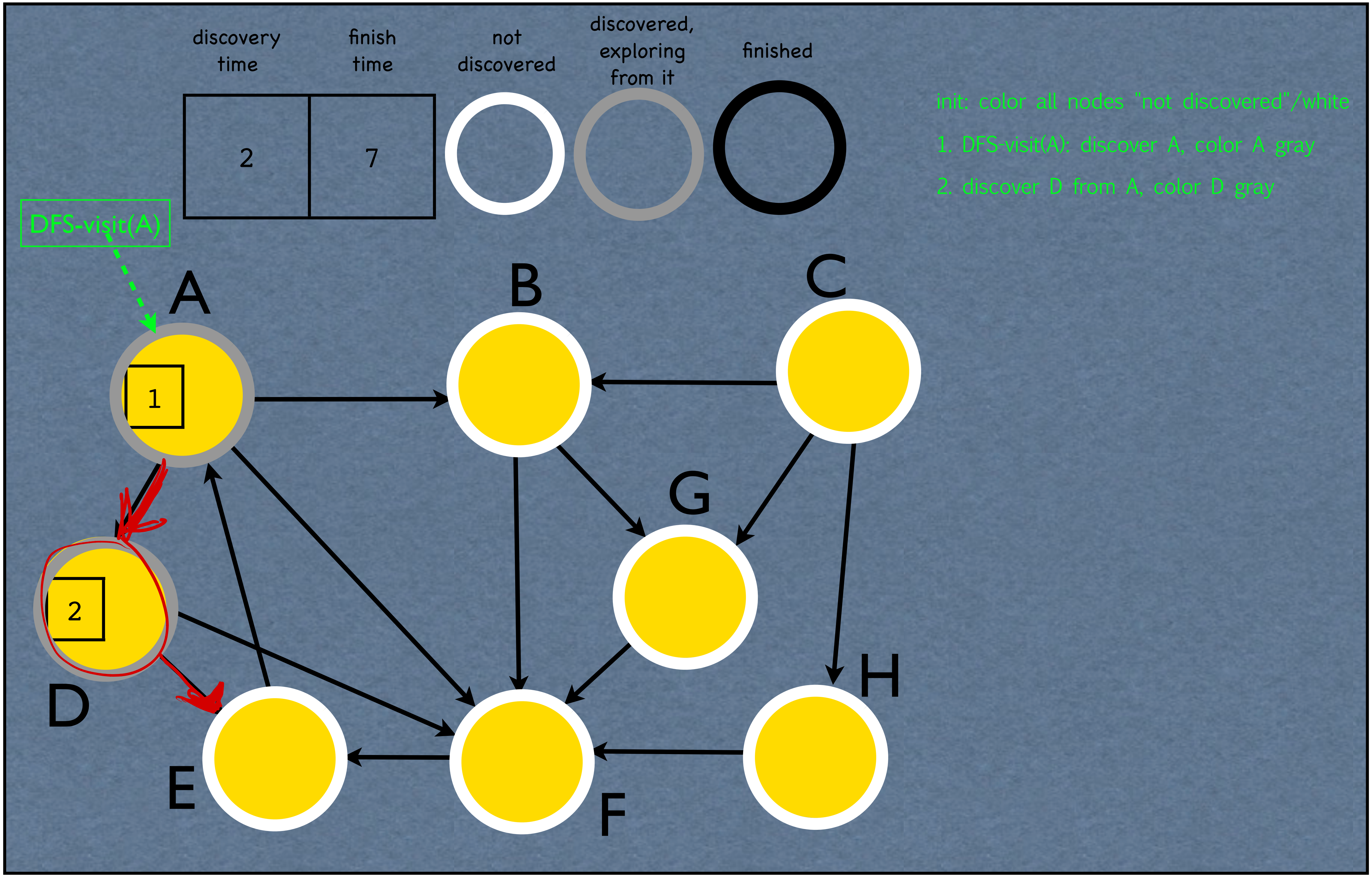


discovery time
finish time
2
7
not discovered
discovered, exploring from it
finished
init: color all nodes "not discovered"/white
A
B
C
D
E
F
G
H

DFS


discovery time
finish time
not discovered
discovered, exploring from it
finished
2
7
DFS-visit(A)
A
B
C
D
E
F
G
H
1
init: color all nodes "not discovered"/white
1. DFS-visit(A): discover A, color A gray
advance (BFS, DFS)
on an edge
u → v
discover v from u
v.parent = u

DFS

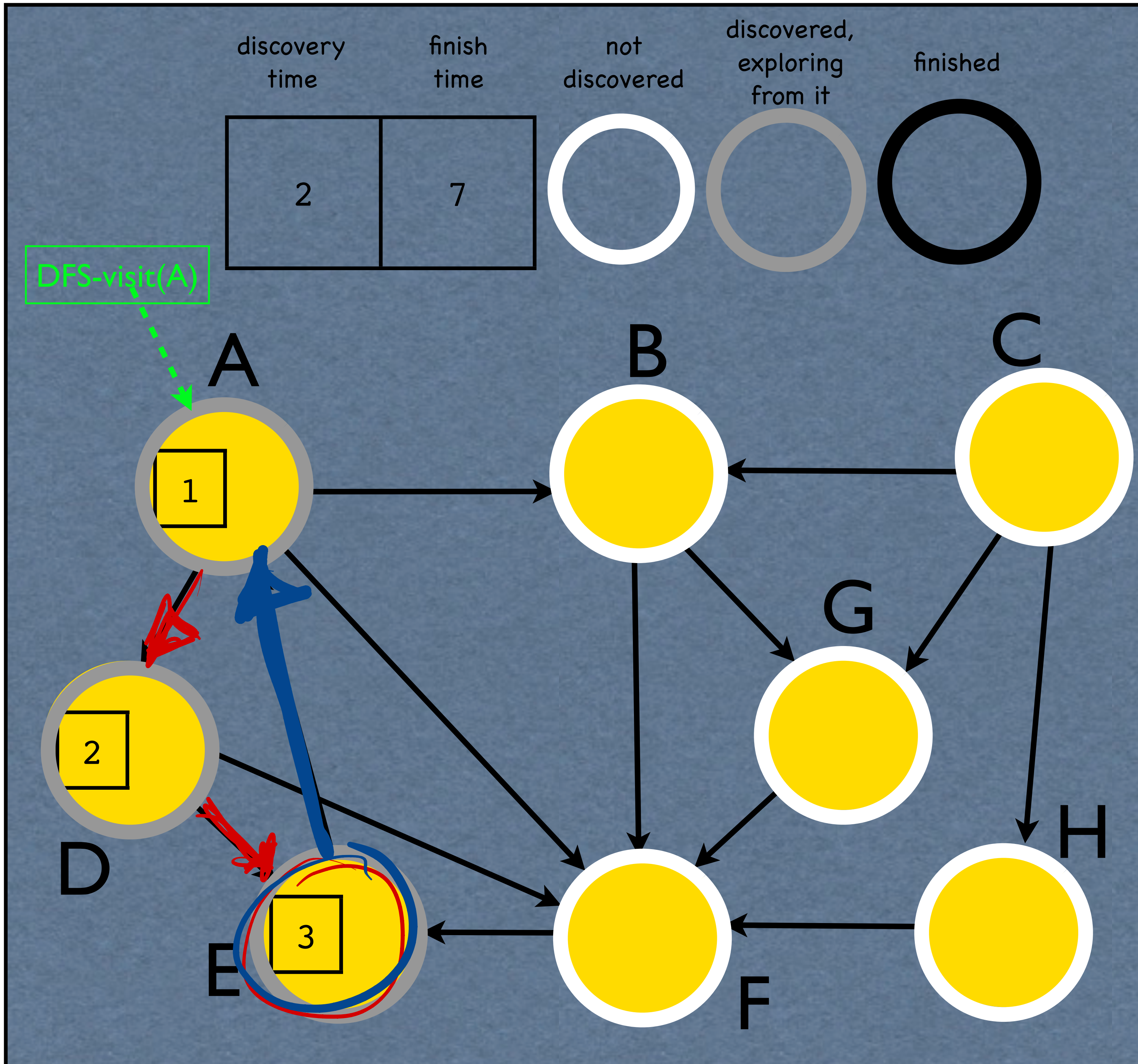


init: color all nodes "not discovered"/white

1. DFS-visit(A): discover A, color A gray

2. discover D from A, color D gray

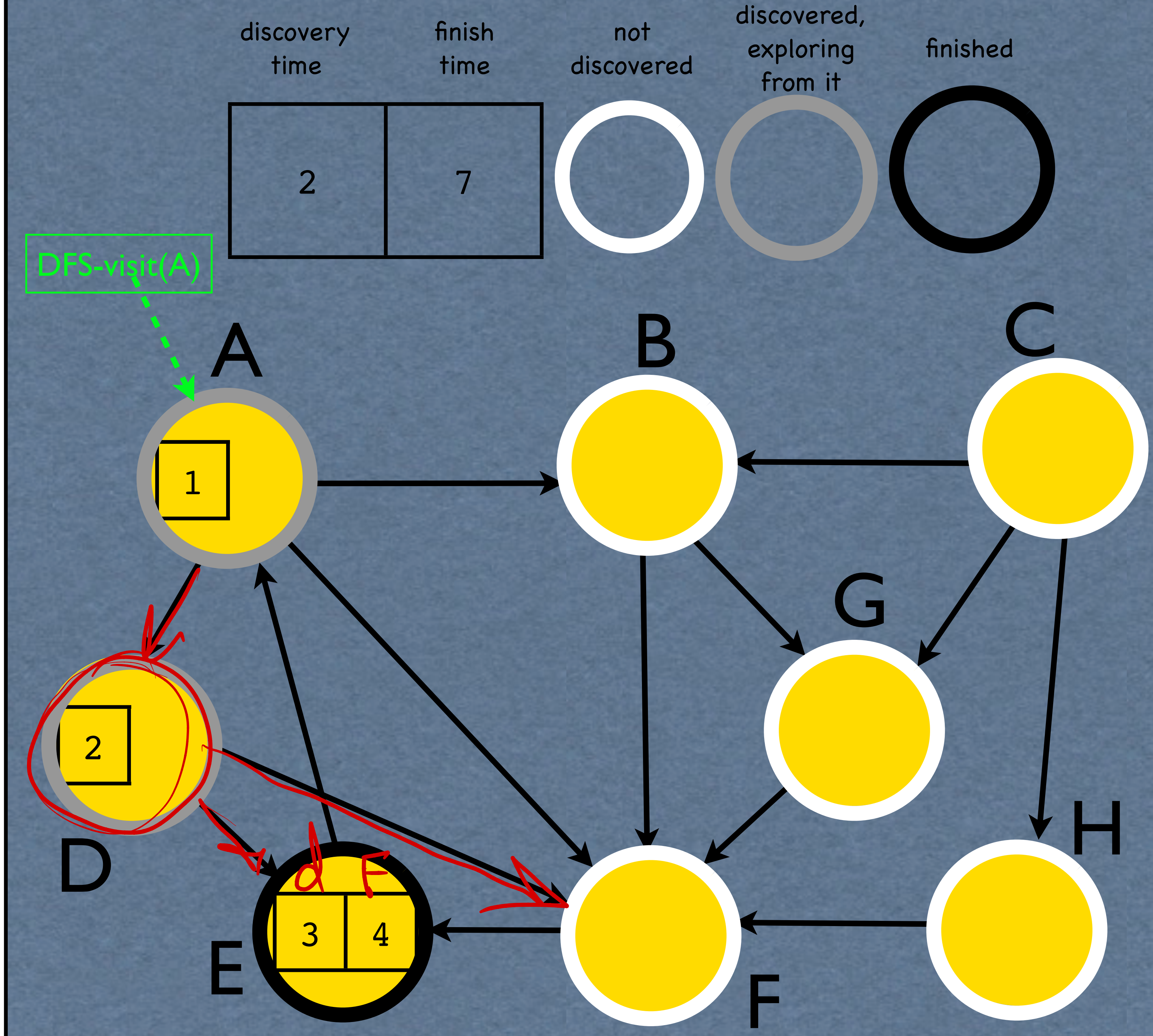
DFS



init: color all nodes "not discovered"/white

1. DFS-visit(A): discover A, color A gray
2. discover D from A, color D gray
3. discover E from D, color E gray

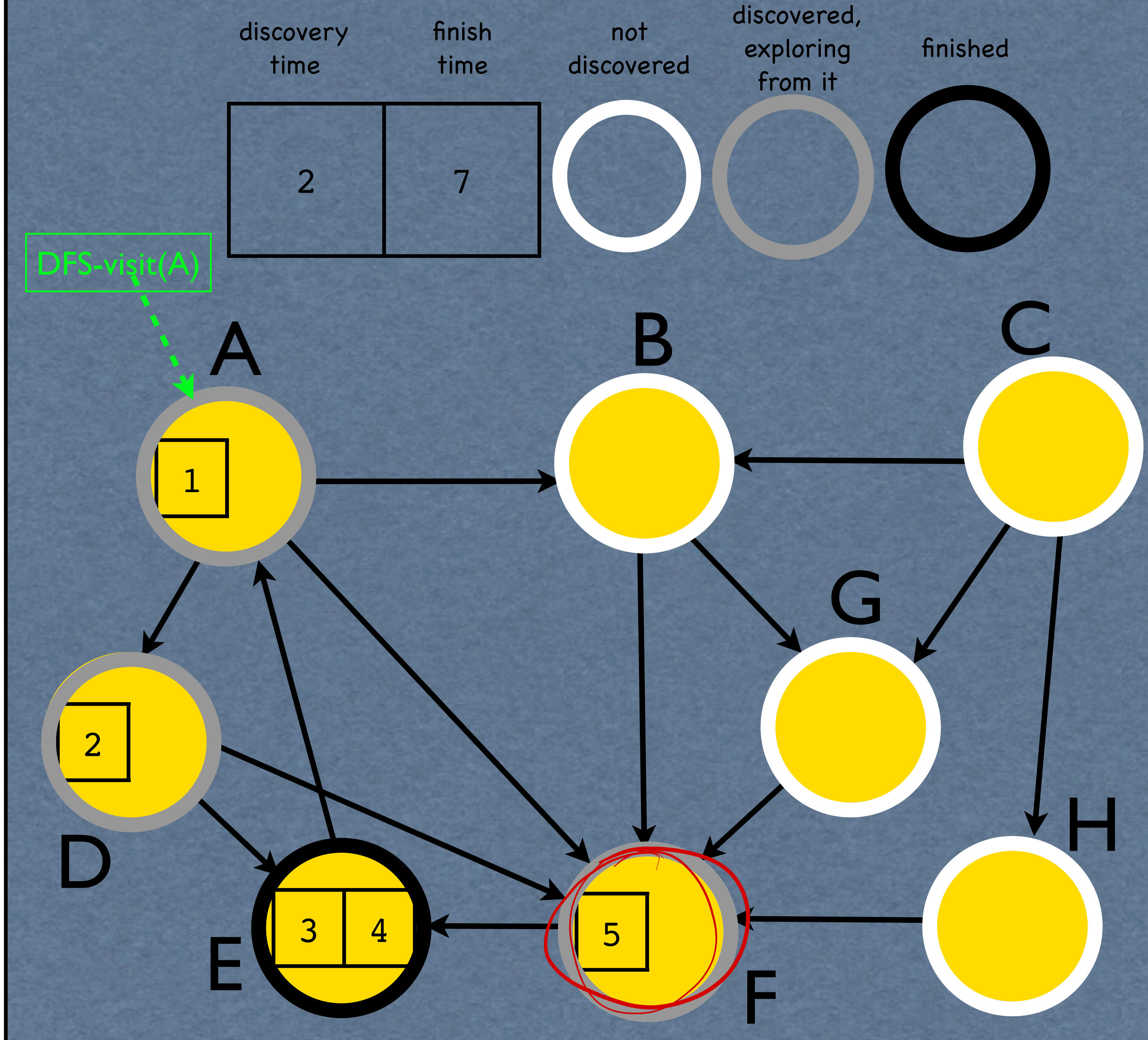
DFS



init: color all nodes "not discovered"/white

1. DFS-visit(A): discover A, color A gray
2. discover D from A, color D gray
3. discover E from D, color E gray
4. finish E, color E black, return to D

DFS



init: color all nodes "not discovered"/white

1. DFS-visit(A): discover A, color A gray

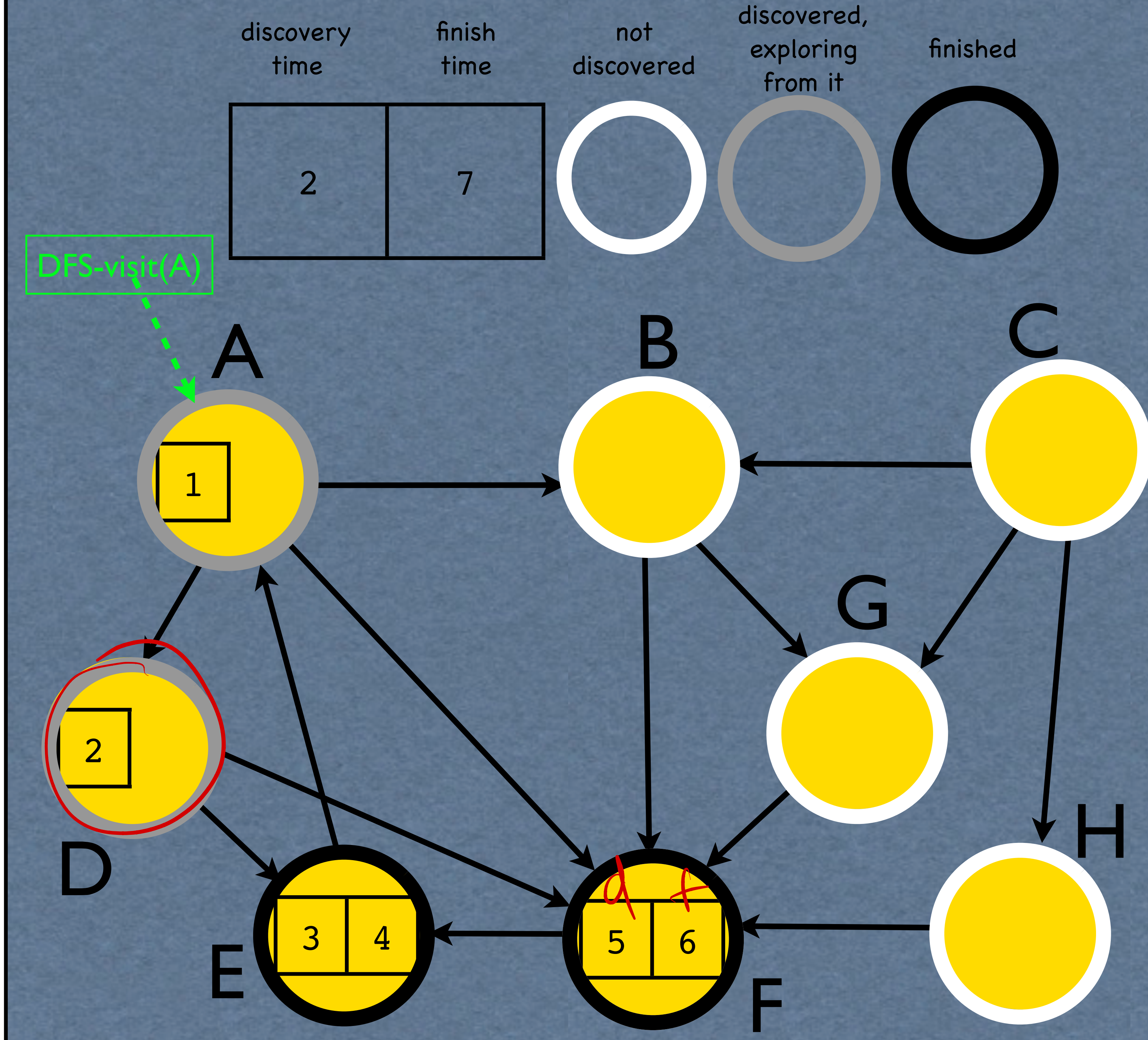
2. discover D from A, color D gray

3. discover E from D, color E gray

4. finish E, color E black, return to D

5 discover F from D, color F gray

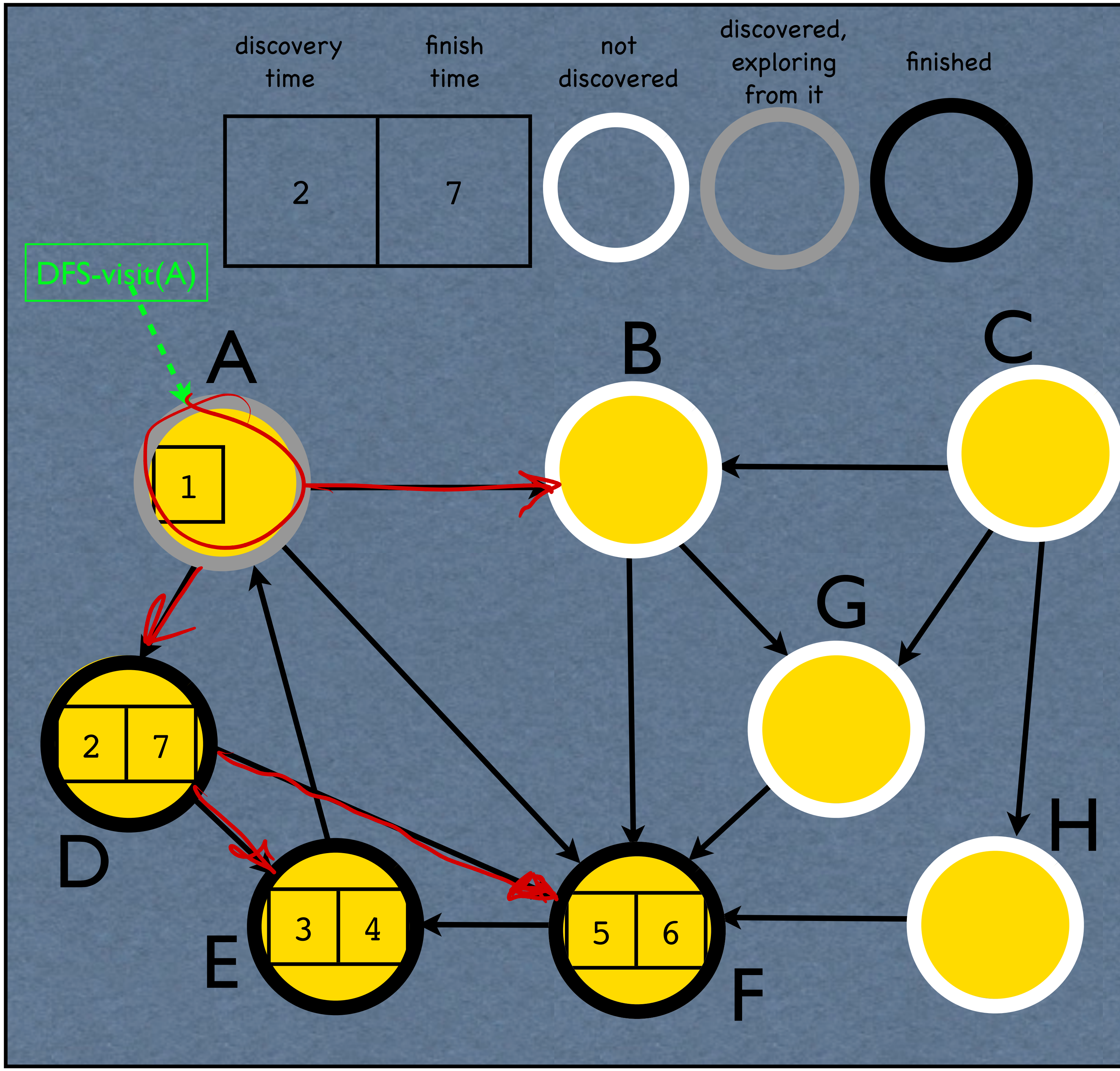
DFS



init: color all nodes "not discovered"/white

1. DFS-visit(A): discover A, color A gray
2. discover D from A, color D gray
3. discover E from D, color E gray
4. finish E, color E black, return to D
5. discover F from D, color F gray
6. finish F, color F black, return to D

DFS



init: color all nodes "not discovered"/white

1. DFS-visit(A): discover A, color A gray

2. discover D from A, color D gray

3. discover E from D, color E gray

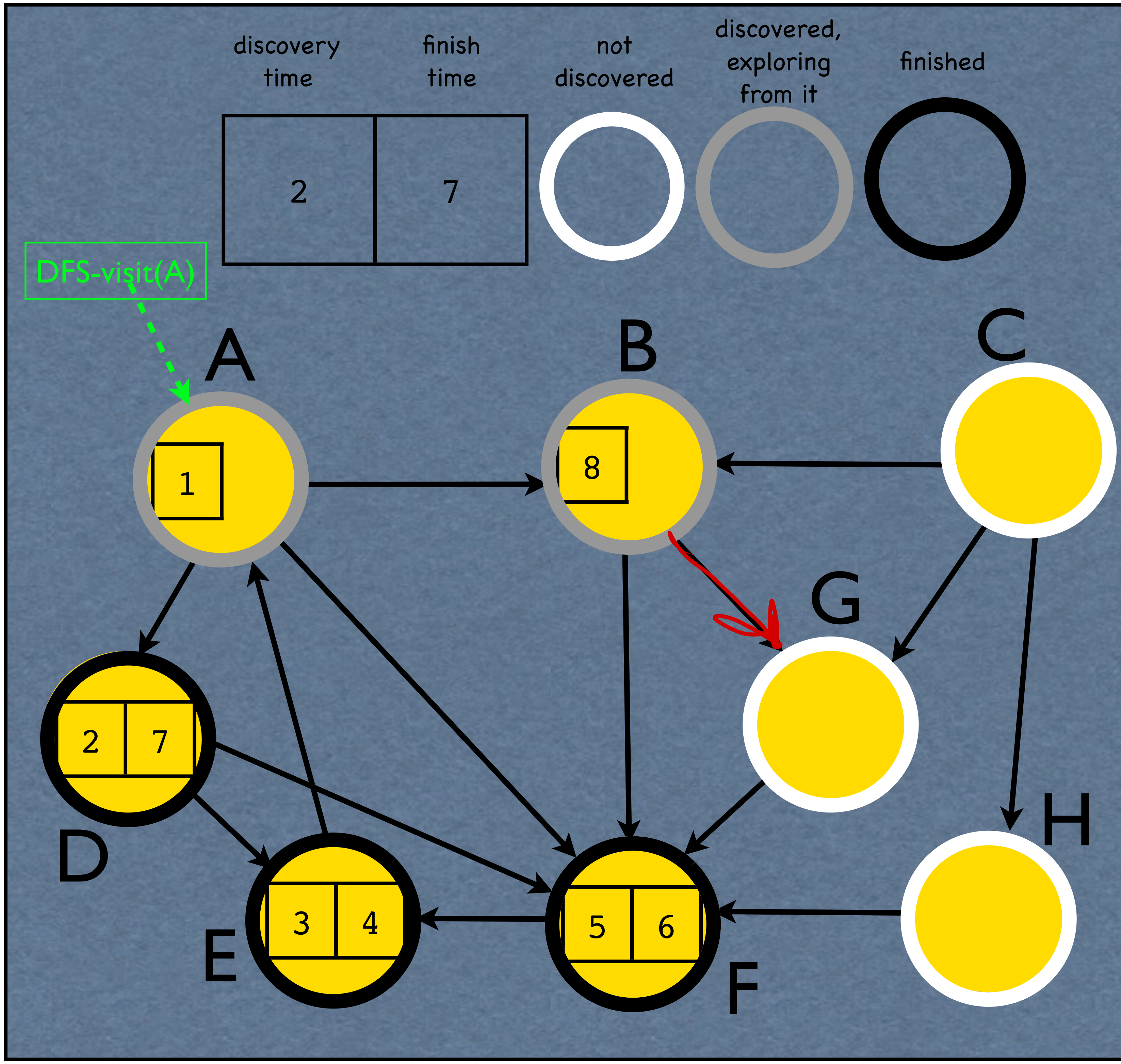
4. finish E, color E black, return to D

5 discover F from D, color F gray

6 finish F, color F black, return to D

7 finish D, color D black, return to A

DFS



init: color all nodes "not discovered"/white

1. DFS-visit(A): discover A, color A gray

2. discover D from A, color D gray

3. discover E from D, color E gray

4. finish E, color E black, return to D

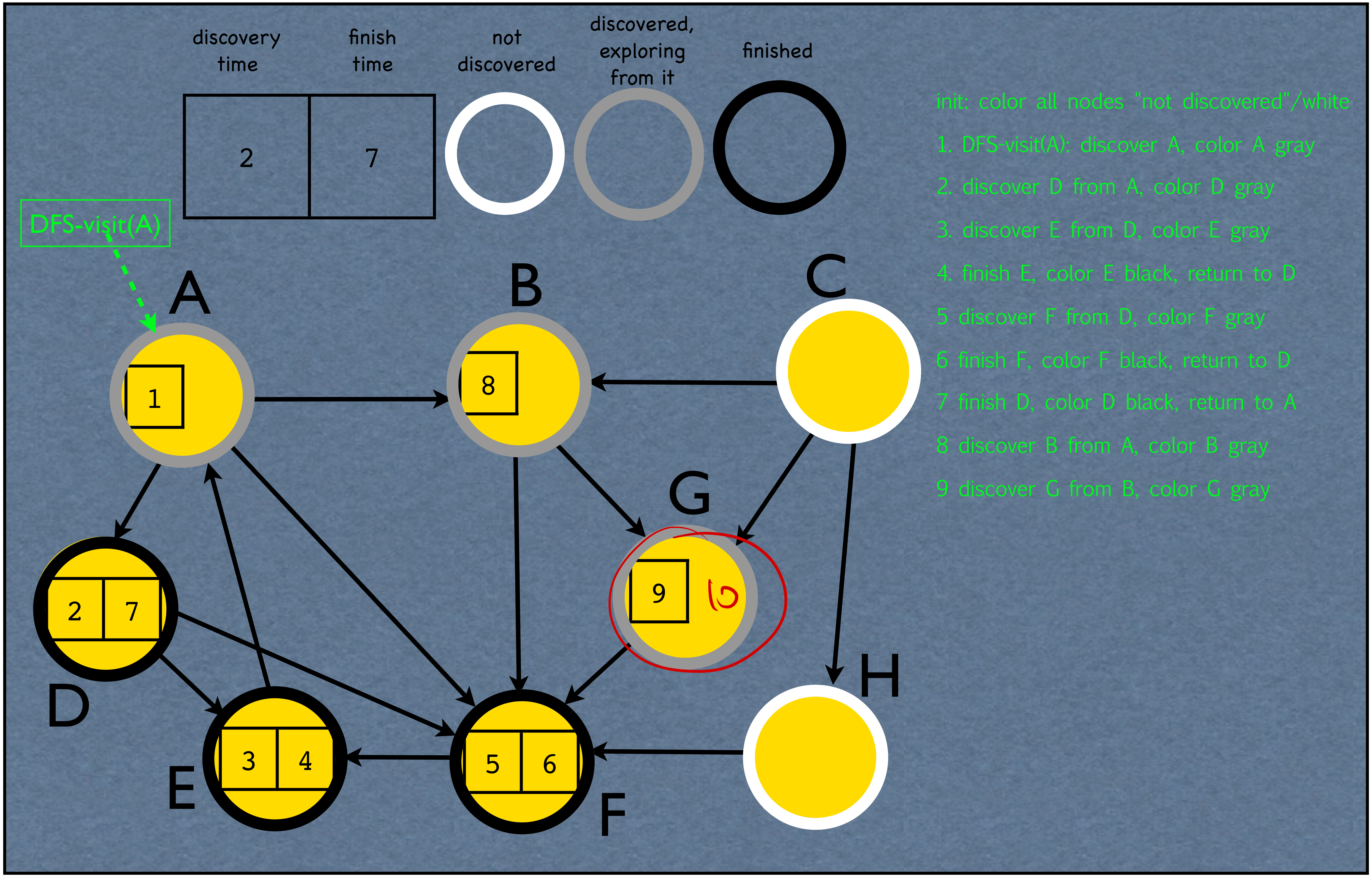
5 discover F from D, color F gray

6 finish F, color F black, return to D

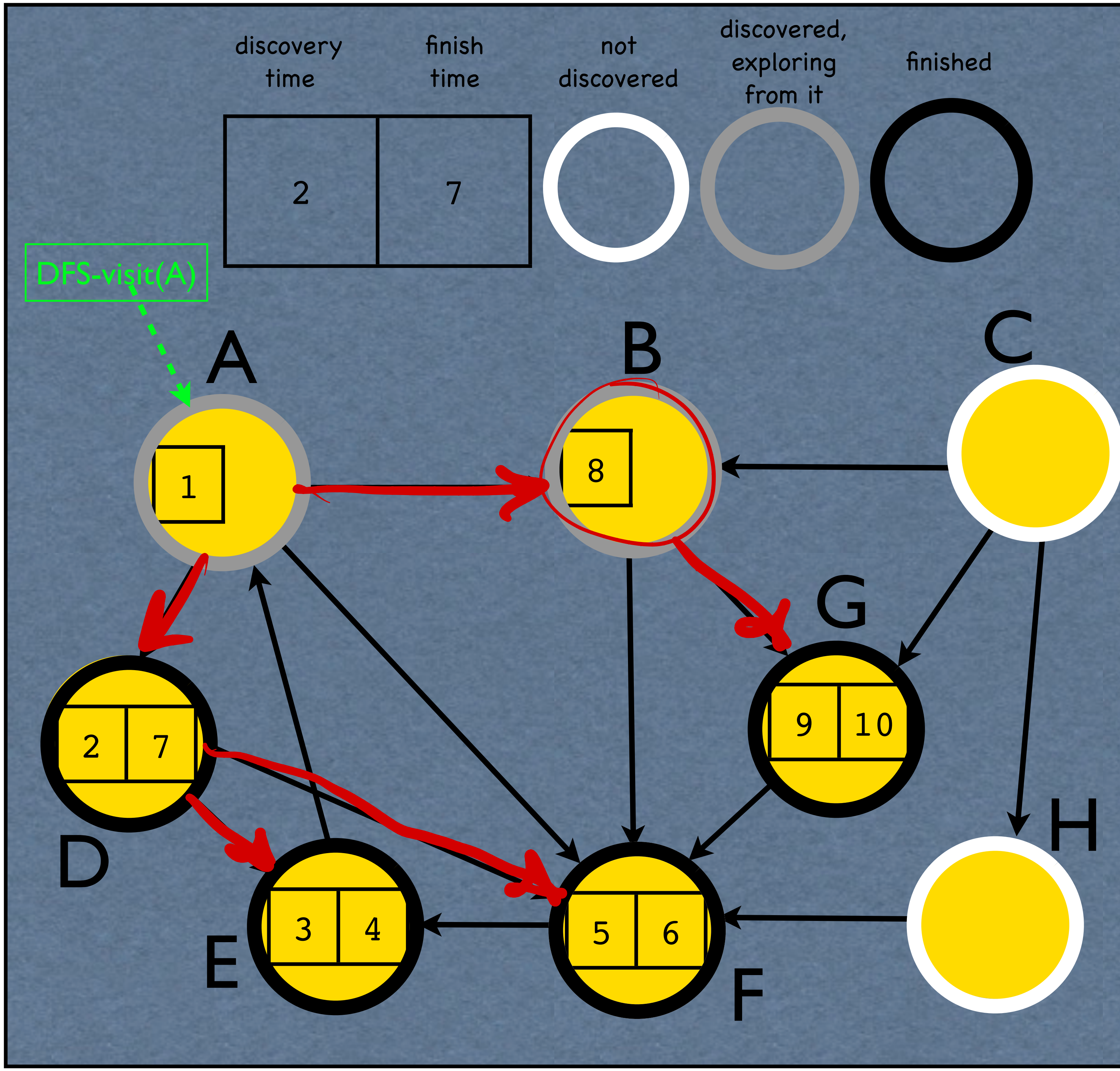
7 finish D, color D black, return to A

8 discover B from A, color B gray

DFS


discovery time
finish time
not discovered
discovered, exploring from it
finished
2
7
DFS-visit(A)
A
B
C
D
E
F
G
H
1
8
9
10
2
7
3
4
5
6
init: color all nodes "not discovered"/white
1. DFS-visit(A): discover A, color A gray
2. discover D from A, color D gray
3. discover E from D, color E gray
4. finish E, color E black, return to D
5 discover F from D, color F gray
6 finish F, color F black, return to D
7 finish D, color D black, return to A
8 discover B from A, color B gray
9 discover G from B, color G gray

DFS



init: color all nodes "not discovered"/white

1. DFS-visit(A): discover A, color A gray

2. discover D from A, color D gray

3. discover E from D, color E gray

4. finish E, color E black, return to D

5 discover F from D, color F gray

6 finish F, color F black, return to D

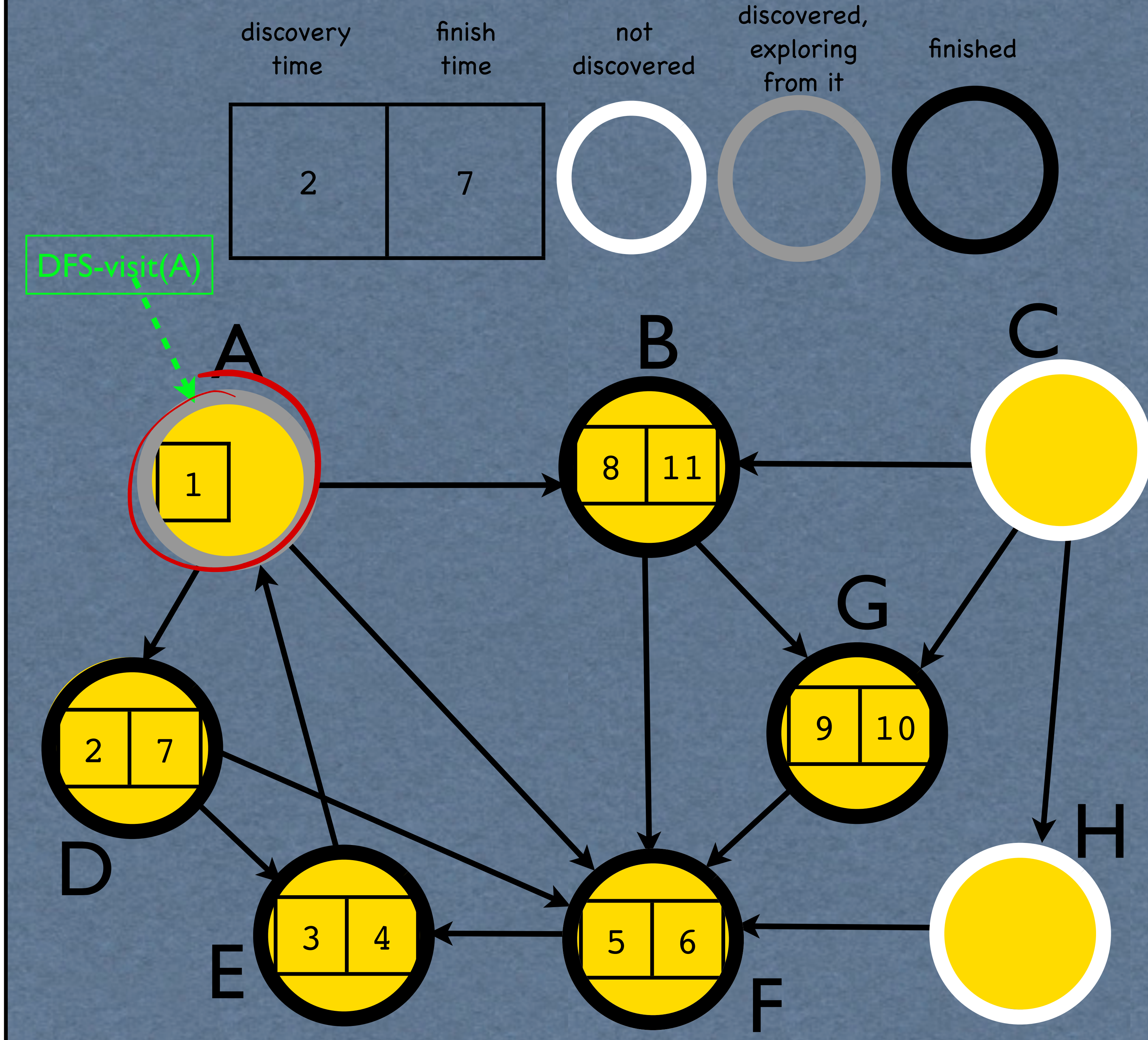
7 finish D, color D black, return to A

8 discover B from A, color B gray

9 discover G from B, color G gray

10 finish G, color G black, return to B

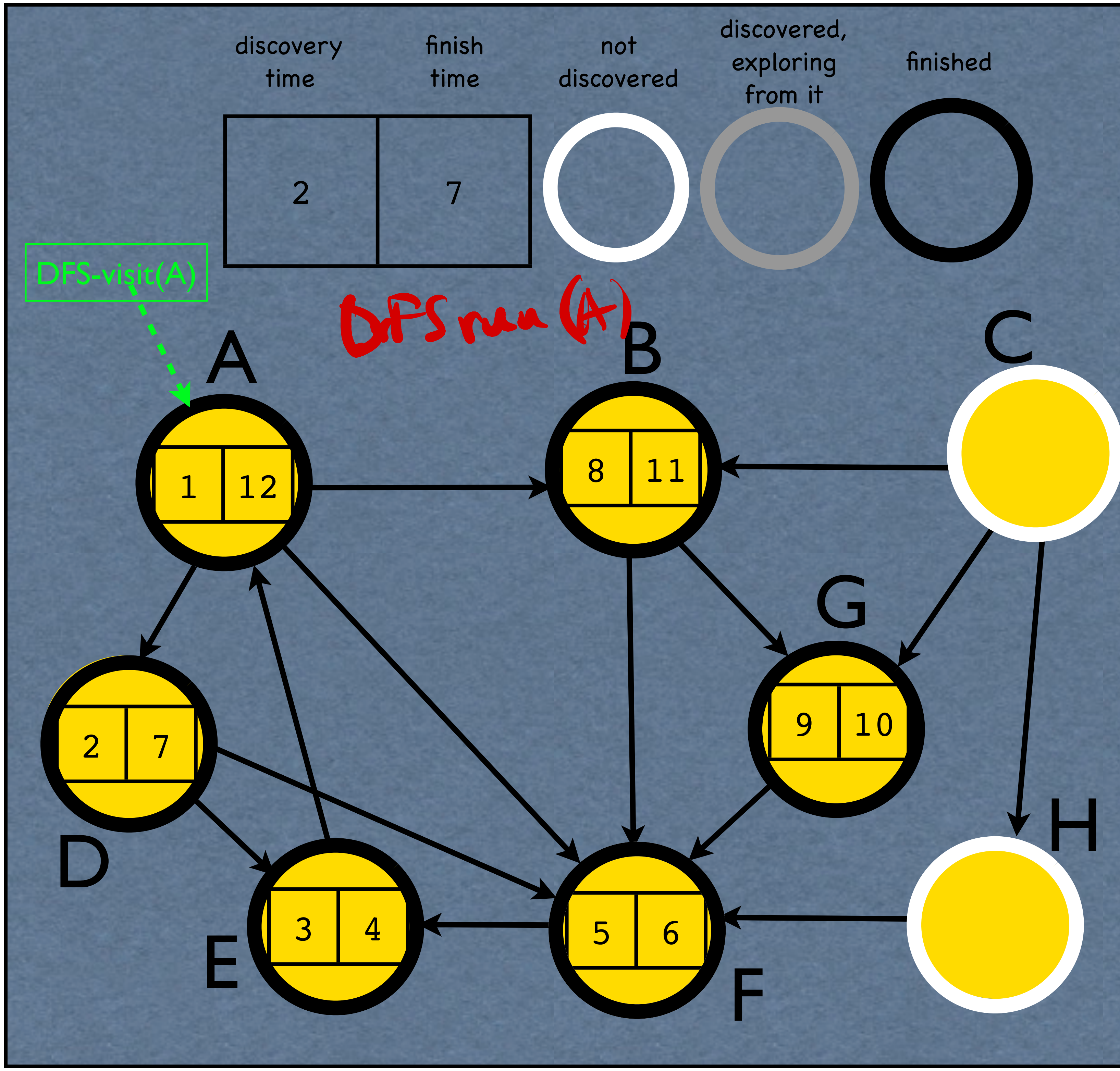
DFS



init: color all nodes "not discovered"/white

1. DFS-visit(A): discover A, color A gray
2. discover D from A, color D gray
3. discover E from D, color E gray
4. finish E, color E black, return to D
5 discover F from D, color F gray
6 finish F, color F black, return to D
7 finish D, color D black, return to A
8 discover B from A, color B gray
9 discover G from B, color G gray
10 finish G, color G black, return to B
11 finish B, color B black, return to A

DFS



init: color all nodes "not discovered"/white

1. DFS-visit(A): discover A, color A gray

2. discover D from A, color D gray

3. discover E from D, color E gray

4. finish E, color E black, return to D

5 discover F from D, color F gray

6 finish F, color F black, return to D

7 finish D, color D black, return to A

8 discover B from A, color B gray

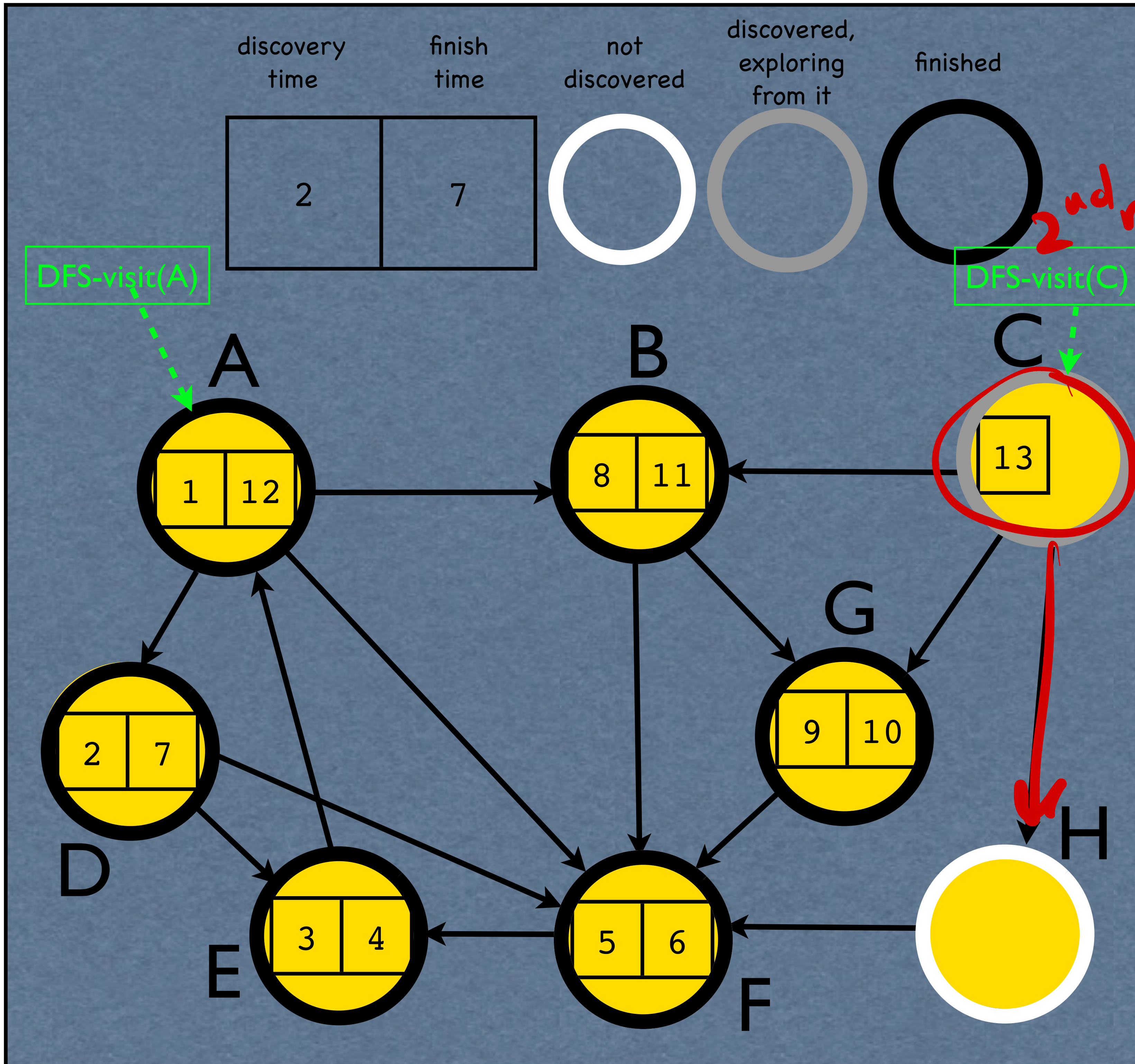
9 discover G from B, color G gray

10 finish G, color G black, return to B

11 finish B, color B black, return to A

12 finish A, color A black, done DFS-visit(A)

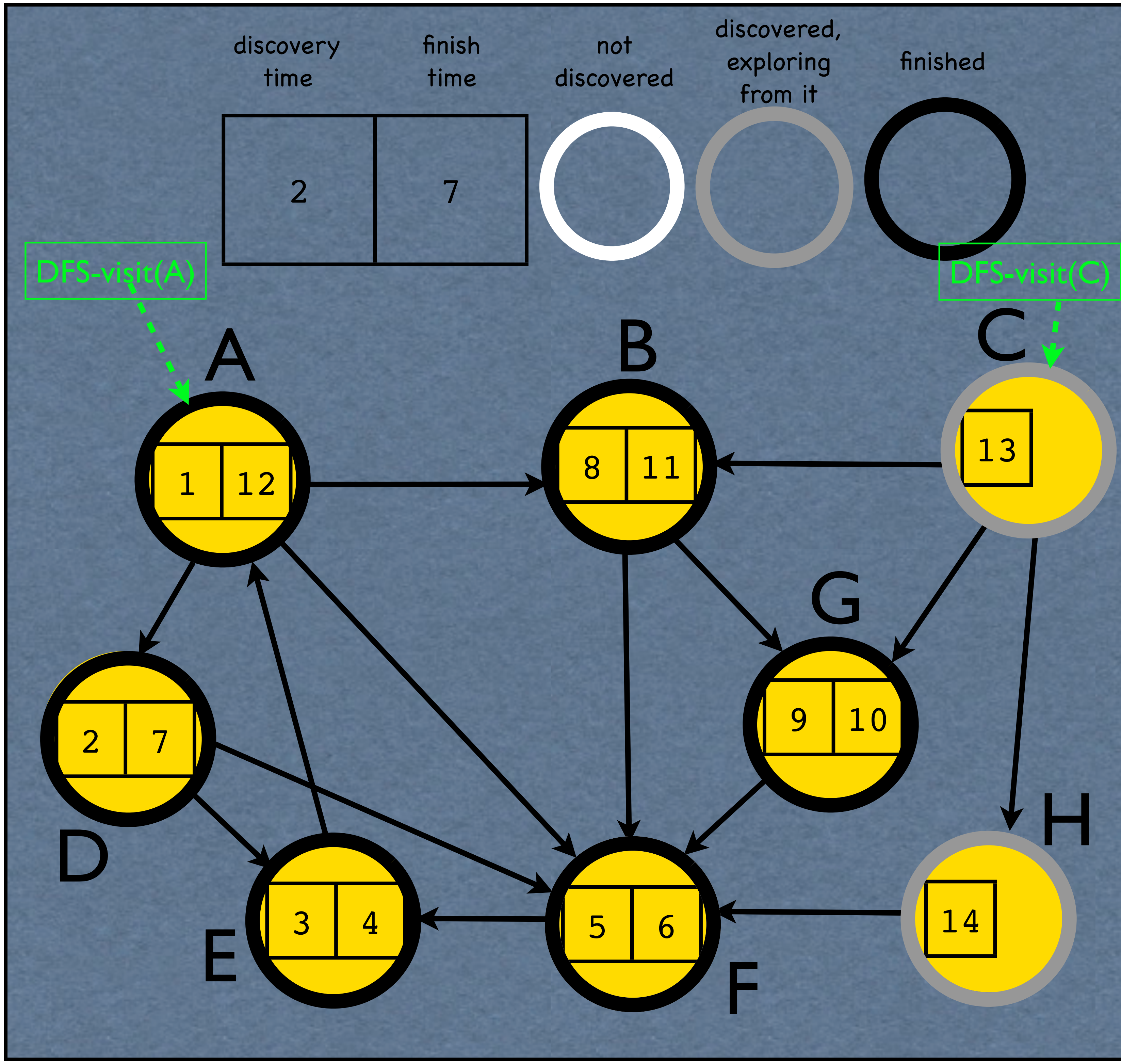
DFS



init: color all nodes "not discovered"/white

1. DFS-visit(A): discover A, color A gray
2. discover D from A, color D gray
3. discover E from D, color E gray
4. finish E, color E black, return to D
5 discover F from D, color F gray
6 finish F, color F black, return to D
7 finish D, color D black, return to A
8 discover B from A, color B gray
9 discover G from B, color G gray
10 finish G, color G black, return to B
11 finish B, color B black, return to A
12 finish A, color A black, done DFS-visit(A)
13 DFS-visit (C), discover C, color C gray

DFS



init: color all nodes "not discovered"/white

1. DFS-visit(A): discover A, color A gray

2. discover D from A, color D gray

3. discover E from D, color E gray

4. finish E, color E black, return to D

5 discover F from D, color F gray

6 finish F, color F black, return to D

7 finish D, color D black, return to A

8 discover B from A, color B gray

9 discover G from B, color G gray

10 finish G, color G black, return to B

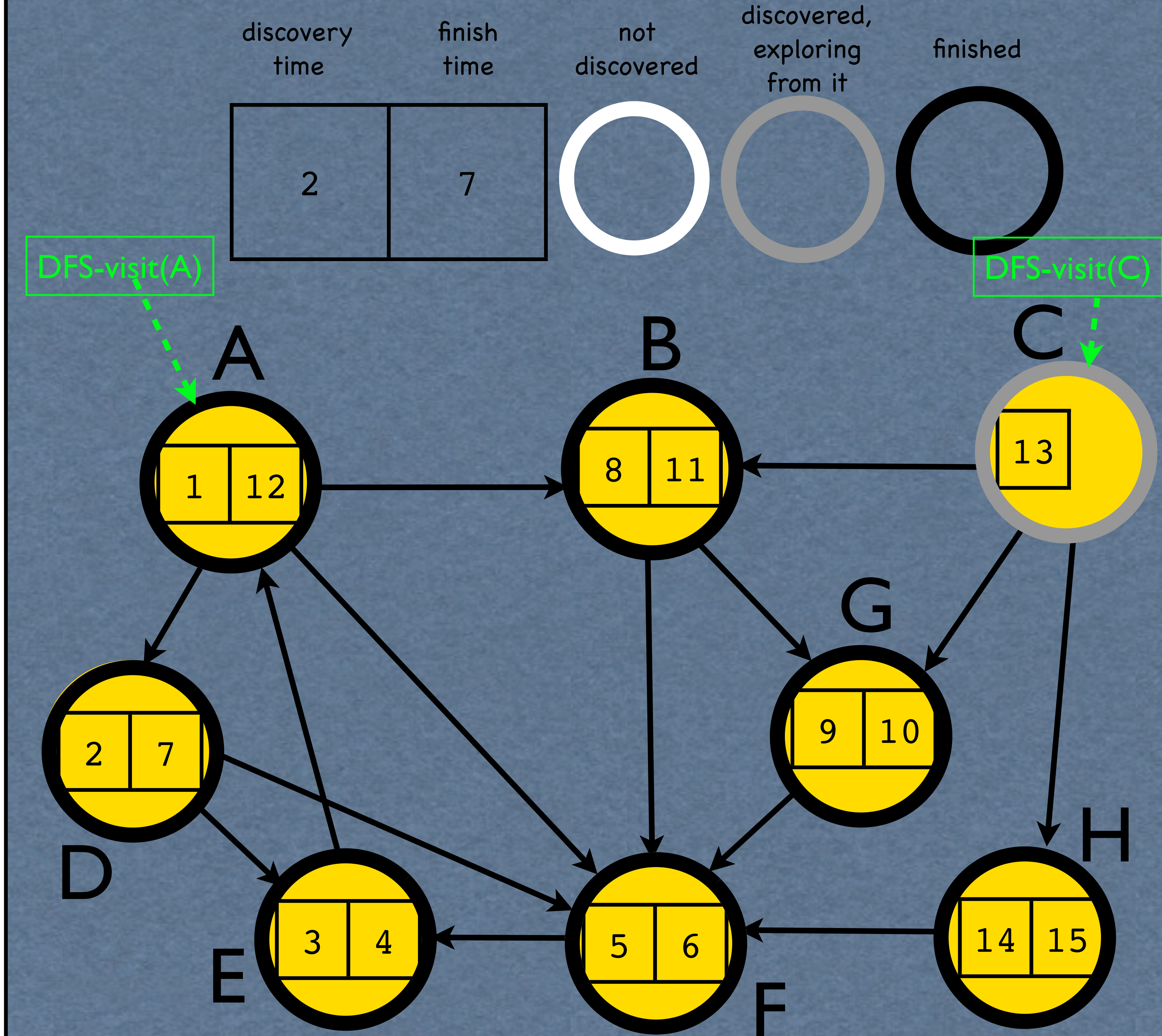
11 finish B, color B black, return to A

12 finish A, color A black, done DFS-visit(A)

13 DFS-visit (C), discover C, color C gray

14 discover H from C, color H gray

DFS



init: color all nodes "not discovered"/white

1. DFS-visit(A): discover A, color A gray
2. discover D from A, color D gray
3. discover E from D, color E gray
4. finish E, color E black, return to D

5 discover F from D, color F gray

6 finish F, color F black, return to D

7 finish D, color D black, return to A

8 discover B from A, color B gray

9 discover G from B, color G gray

10 finish G, color G black, return to B

11 finish B, color B black, return to A

12 finish A, color A black, done DFS-visit(A)

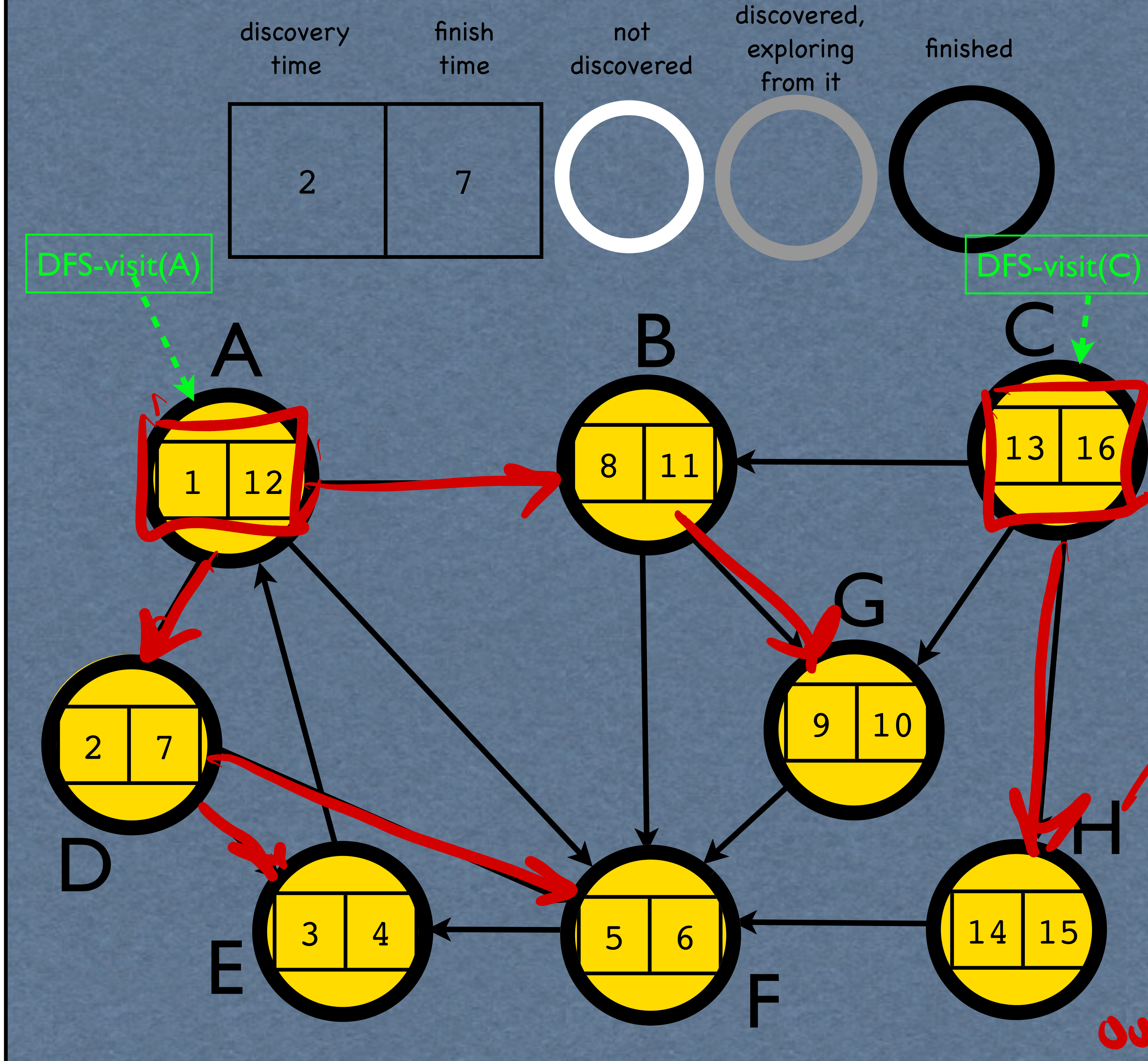
13 DFS-visit (C), discover C, color C gray

14 discover H from C, color H gray

15 finish H, color H black, return to C

DFS

$O(E)$
$O(E+V)$



init: color all nodes "not discovered"/white

1. DFS-visit(A): discover A, color A gray
2. discover D from A, color D gray
3. discover E from D, color E gray
4. finish E, color E black, return to D
5. discover F from D, color F gray
6. finish F, color F black, return to D
7. finish D, color D black, return to A
8. discover B from A, color B gray
9. discover G from B, color G gray
10. finish G, color G black, return to B
11. finish B, color B black, return to A
12. finish A, color A black, done DFS-visit(A)
13. DFS-visit (C), discover C, color C gray
14. discover H from C, color H gray
15. finish H, color H black, return to C
16. finish C, color C black, finish DFS-visit(C)

$x < t < v < y$

d / f

DFS tree

x | y

t, v

t, v

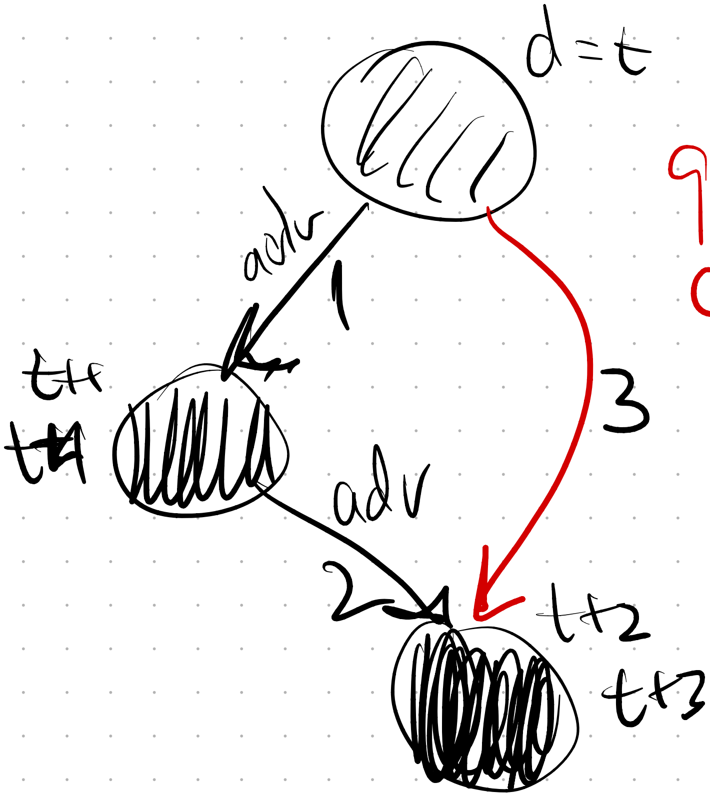
output: d, f arrays
DFS tree(s)

DFS edge classification

DFS, advance

- "tree" edge : from vertices gray to white
  - a tree edge advances the graph exploration/traversal
- "back" edge : from vertices gray to gray
  - a back edge points to a cycle within the current exploration nodes
- "forward" edge : from vertices a(gray) to b(black), if a discovered first
  - discovery_time[a] < discovery_time[b]
  - points to a different part of the tree, already explored from a
- "cross" edge : from vertices a(gray) to b(black), if b discovered first
  - discovery_time[a] > discovery_time[b]
  - points to a different part of the tree, explored before discovering a

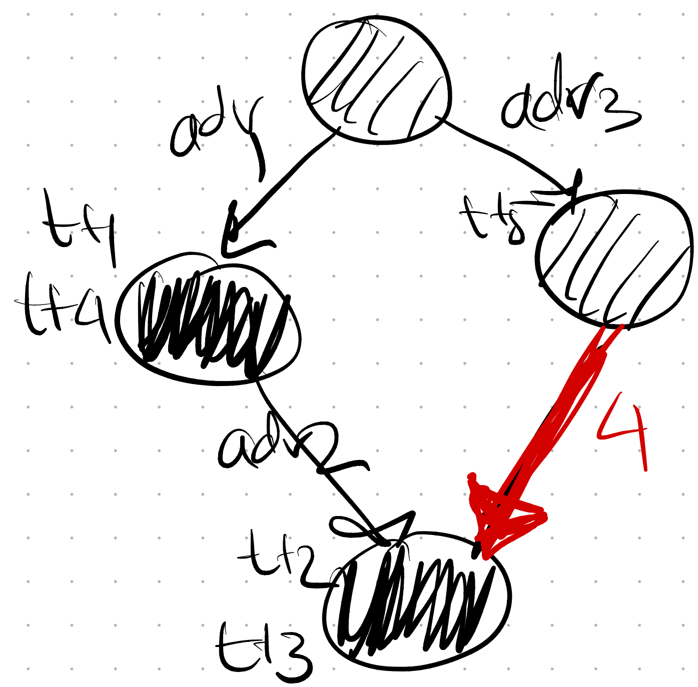
Forward edge



gray → black

d[gray] < d[black]

Cross edge

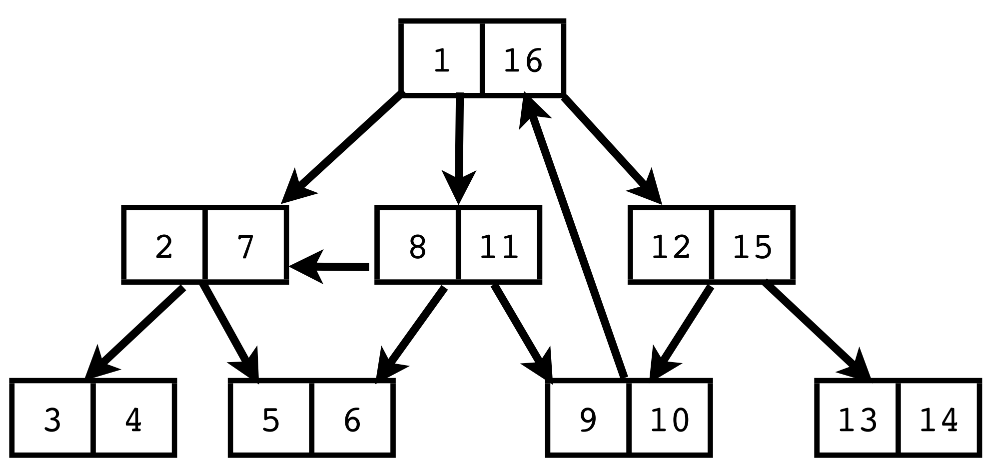


4: gray → black

d[gray] > d[black]

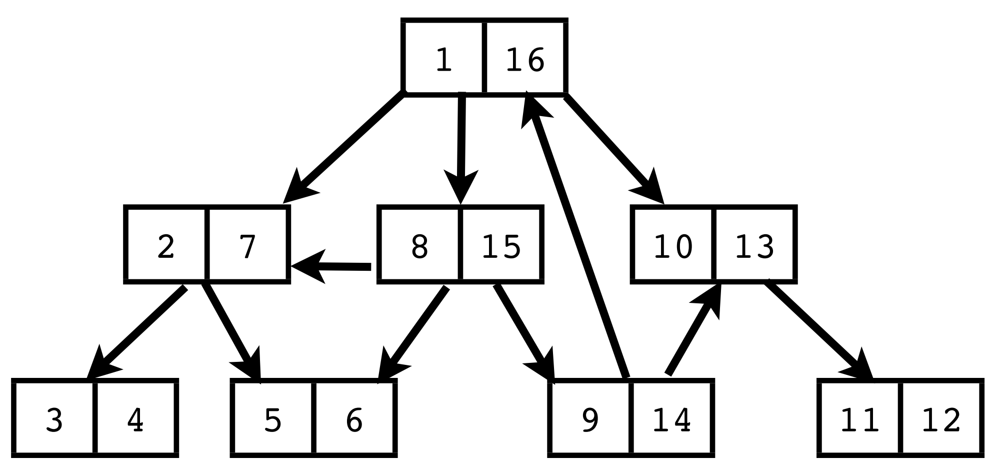
Checkpoint

- on the animated example, label each edge as "tree","back", "cross", or "forward"
- do the same on the following example (DFS discovery and finish times marked for each node)



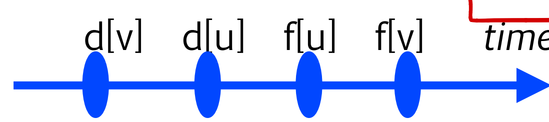
Checkpoint

- almost same example, with a small modification: one edge was reversed



DFS observations

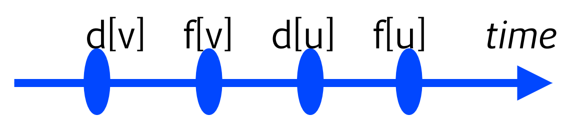
- Running time O(V+E), same as BFS
- vertex v is gray between times discover[v] and finish[v]
- gray time intervals (discover[v], finish[v]) are inclusive of each other
  - (d[v], f[v]) can include (d[u], f[u]) : d[v] < d[u] < f[u] <f[v]

    discover u from ancestor v

    v u

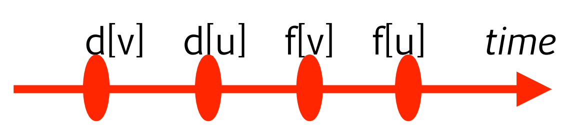
  - (d[v], f[v]) can separate from (d[u], f[u]) : d[v] < f[v] < d[u] <f[u]

    v u

    u not reach from v, discovered later

  - (d[v], f[v]) cannot intersect (d[u], f[u]) : d(v) < d(u) < f[v] <f[u]

    discover u while v=gray (from v)

- graph G=(V,E) is acyclic (does not have cycles) if DFS does not find any "back" edge

Theorem $\left.\begin{array}{l} f[u] > f[v] \\ \boxed{\text{no path } u \rightsquigarrow v} \end{array}\right\} \Rightarrow$ no path $v \rightsquigarrow u$

proof: $f[u] > f[v]$ from previous slide 2 options

- $d[v] < f[v] < d[u] < f[u] \Rightarrow$ $u \rightsquigarrow$ path

OR

- $d[u] < d[v] < f[v] < f[u] \Rightarrow$ path $u \rightsquigarrow v$

$\Rightarrow$ contradicts the hypothesis, not possible

not

$\Rightarrow d[v] < f[v] < d[u] < f[u] \Rightarrow$

no path $v \rightsquigarrow u$ (u discovered after v close)

Undirected graphs cycles

- graph $G=(V,E)$ is acyclic (does not have cycles) if DFS does not find any "back" edge
- since G is undirected, no cycles implies $|E| \leqslant |V|-1$
- running DFS, if we find more than $|V|-1$ edges, there must be a cycle
- Undirected graphs: find-cycles algorithm takes $O(V)$

Directed graphs cycles

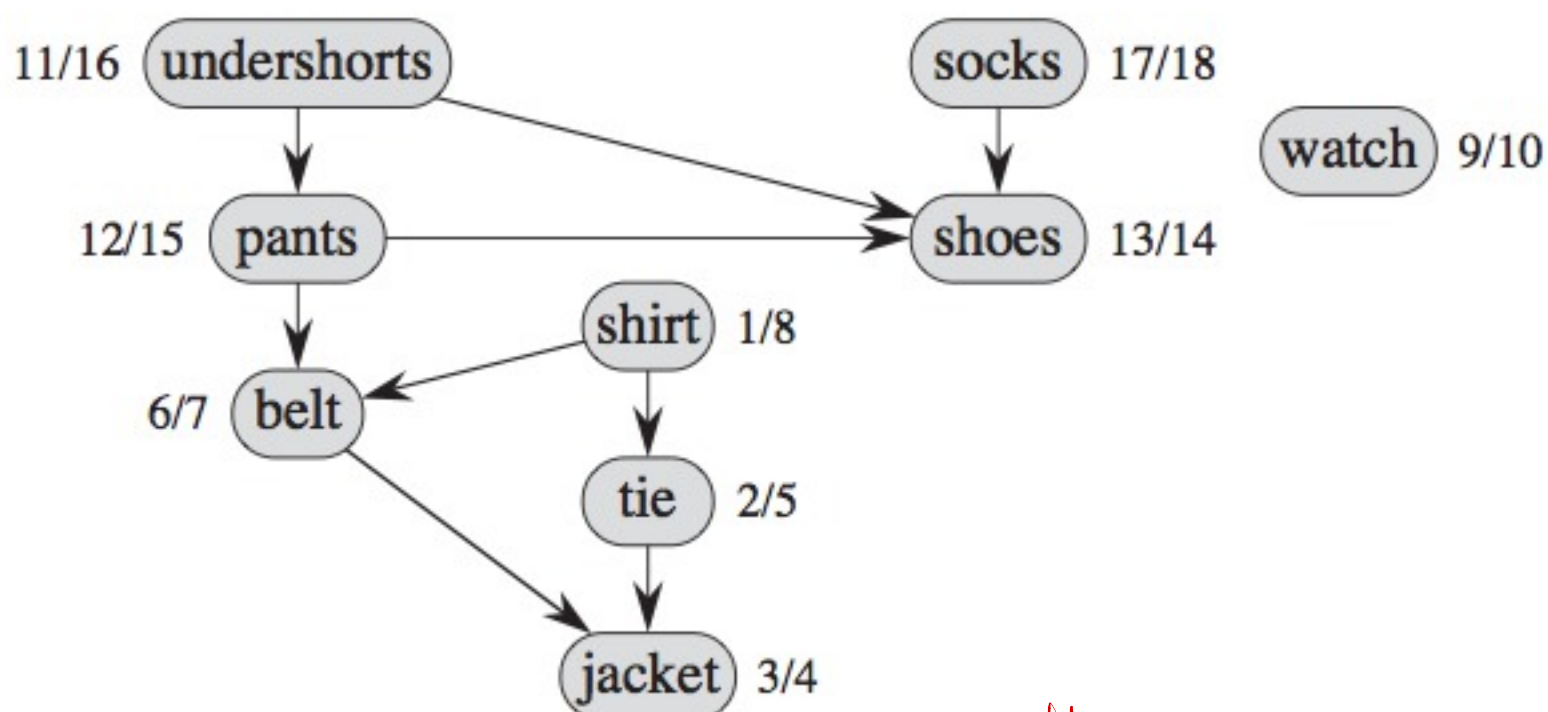
- graph G=(V,E) is acyclic (does not have cycles) if DFS does not find any "back" edge
- for directed graphs, even without cycles they can have more edges, $|E| > |V|-1$
- algorithm to determine cycles: run DFS, look for back edges - $O(V+E)$ time
- DAG = directed acyclic graph

~~Topological~~ LINE sort (DAG)

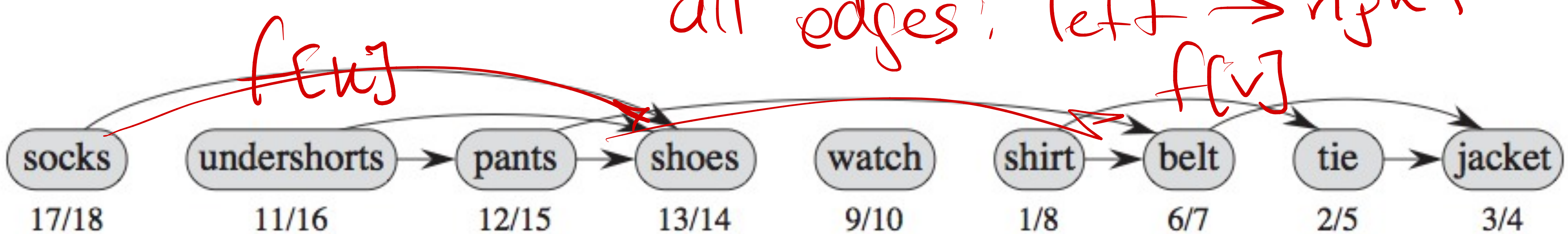
no cycles

- DAG admits topological sort: all vertices "sorted" on a line, such that all edges point from left to right-no cycles - 2 graphs below are the same-
- to do this: algorithm: run DFS, time O(V+E). Output vertices in reverse order given by finishing time

f[ ]



all edges: left → right



Check Point

want: no edge $v \to u$

f[u] bad f[v]

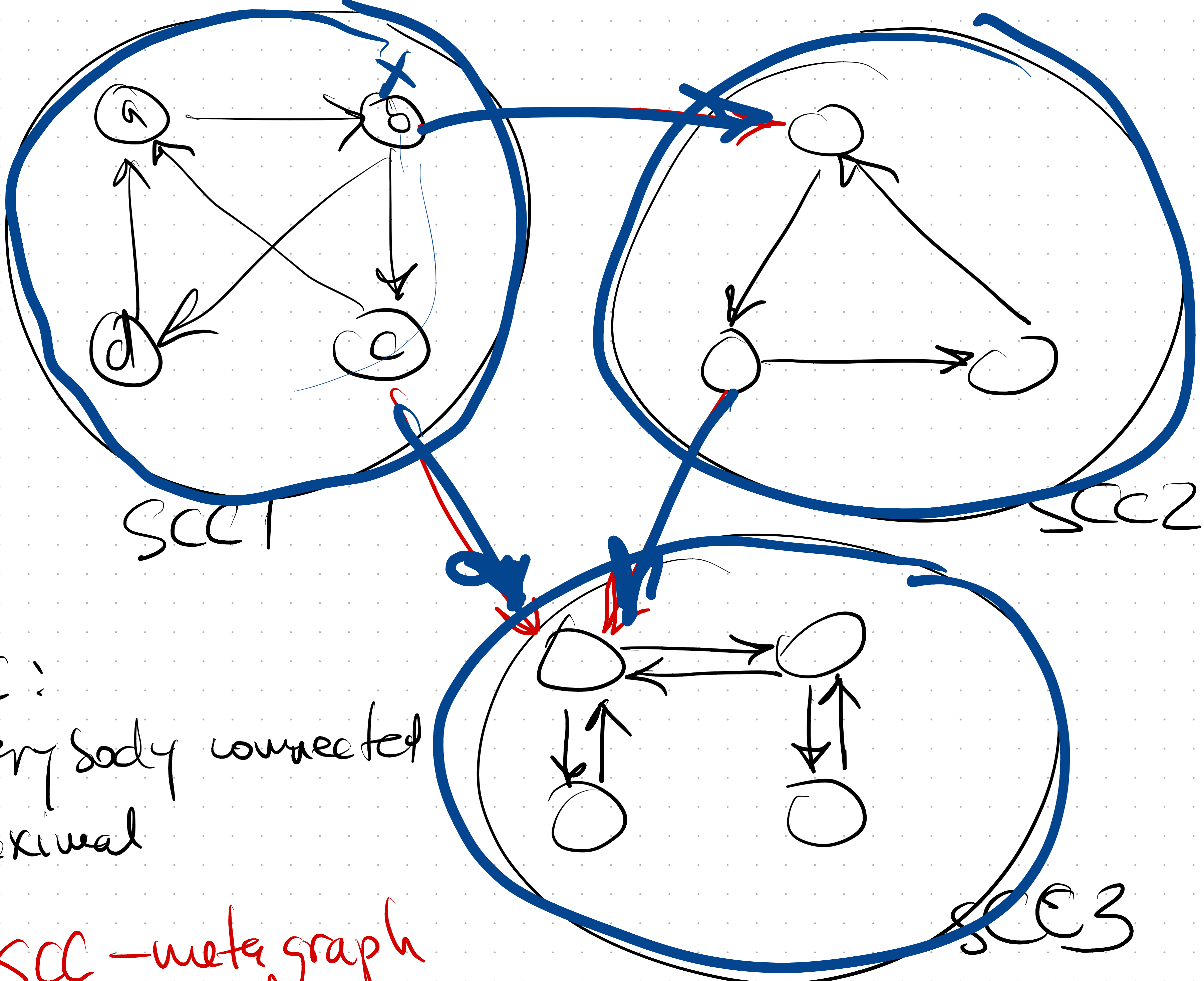
$f[u] > f[v]$

case 1
$\exists$ path $u \rightsquigarrow v$ $\Rightarrow$ no edge $v \to u$ (otherwise cycle)

case 2
no path $u \rightsquigarrow v$ Th) no path $v \rightsquigarrow u$ $\Rightarrow$ no edge $v \to u$

- how can we use DFS to determine if there is a path from u to v ?

- prove that by sorting vertices in the reverse order of finishing times, we obtained a topological sort
  - assuming no cycles
  - in other words, all edges point in the same direction

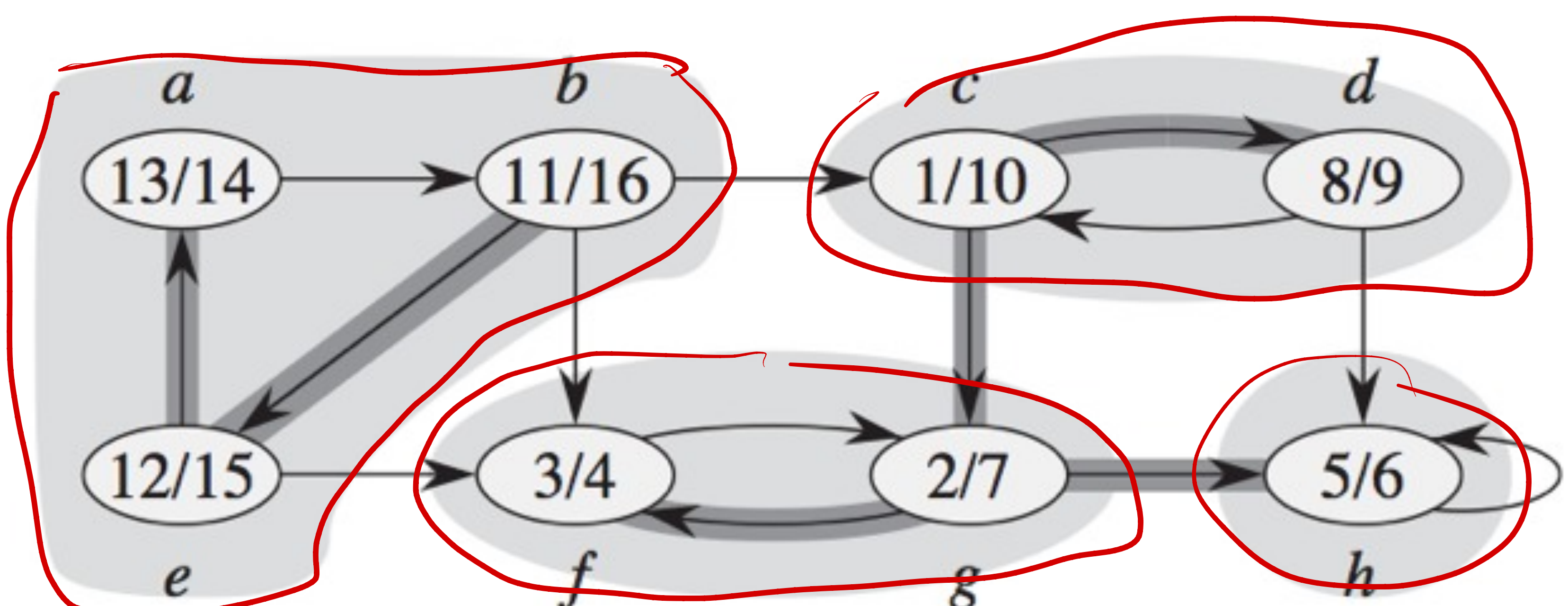


SCC:
- everybody connected
- maximal

Th SCC – meta graph
nodes = SCC
edges = between SCC $\Rightarrow$ DAG

Strongly connected components

- SCC = a set of vertices S⊂V, such that for any two (u,v)∈S, graph G contains a path u⇝v and a path v⇝u
- trivial for undirected graphs
  - all connected vertices are in fact strongly connected
- tricky for directed graphs
- graph below has the DFS discover/finish times and marked 4 strongly connected components; "tree" edges highlighted
- between two SCC, A and B, there cannot exists paths both ways (A∋u⇝v∈B and B∋v′⇝u′∈A)
  - paths both ways would make A and B a single SCC

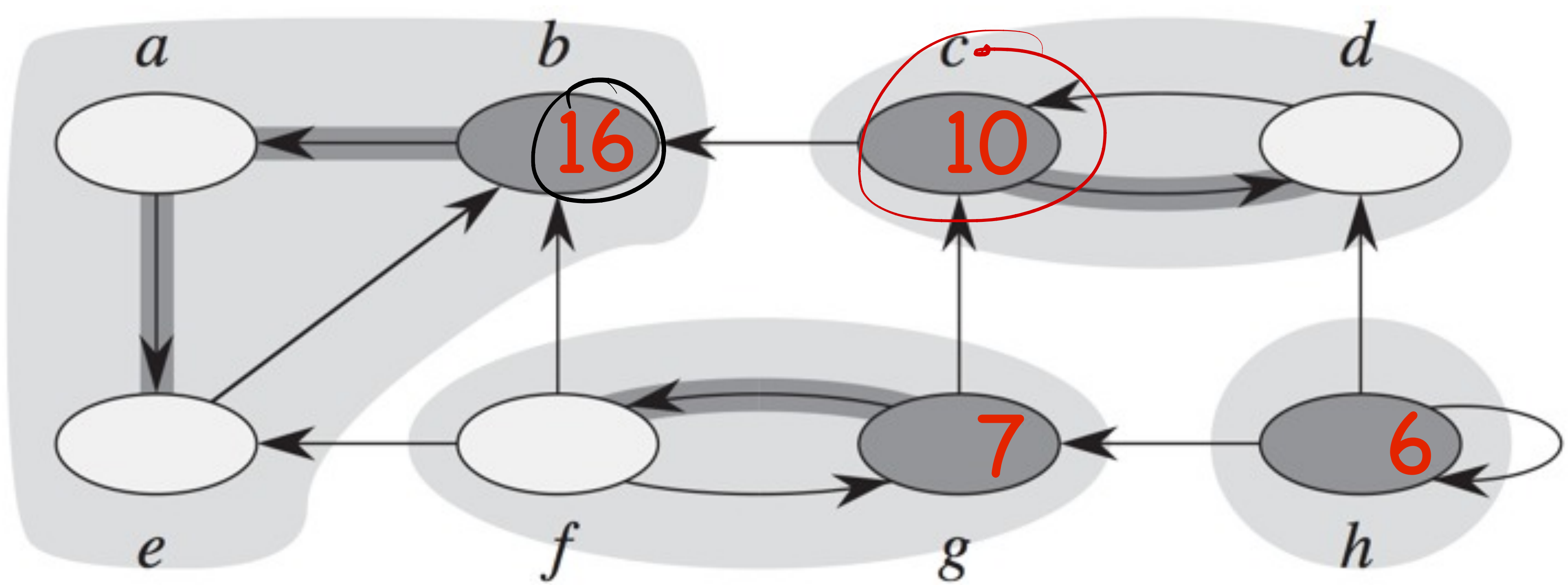


Strongly connected components

- run 1st DFS on G to get finishing times f[u]
- run 2nd DFS on G-reversed (all edges reversed -see picture), each DFS-visit in reverse order of f[u]

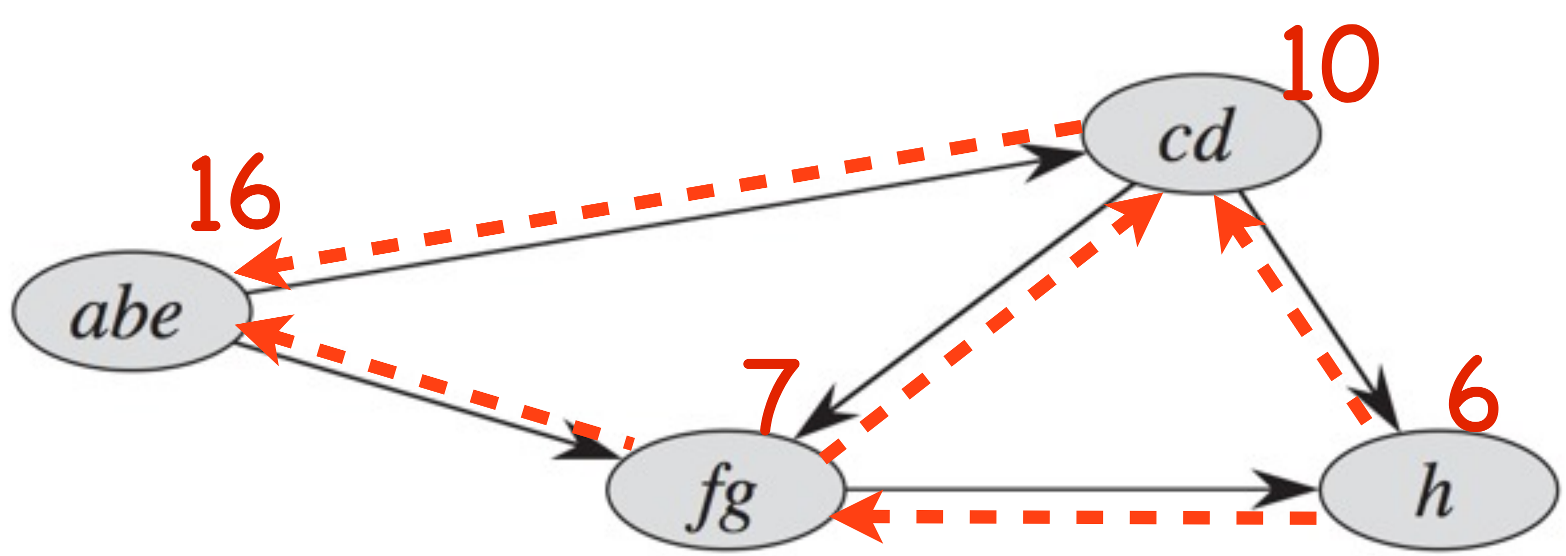
  DFS-run = SCC(root)

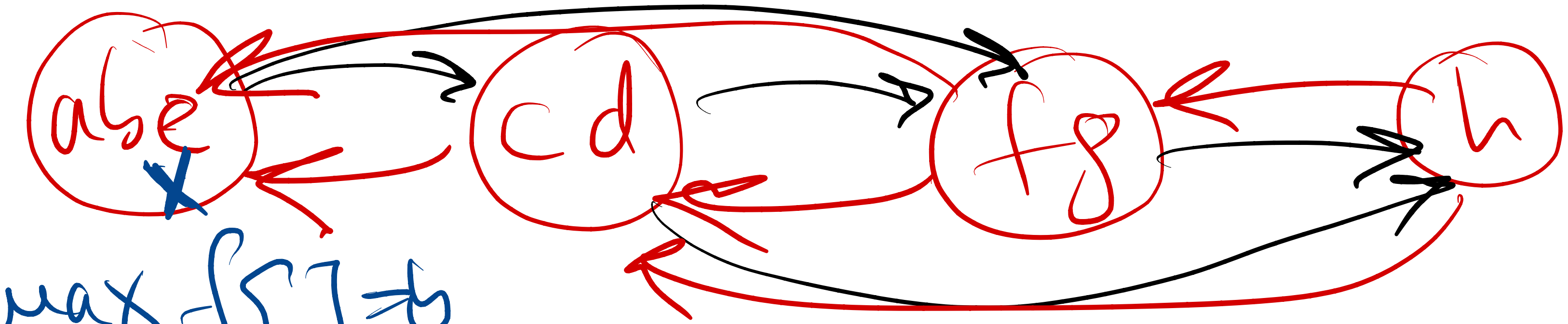
  - finishing times marked in red for the DFS-visit root vertices
- output each tree (vertices reached) obtained by 2nd DFS as an SCC



Strongly connected components

- why 2nd DFS produces precisely the SCC -s?
- SCC-graph of G: collapse all SCC into one SCC-vertex, keep edges between the SCC-vertices
- - SCC graph is a DAG;
  - contradiction argument: a cycle on the SCC-graph would immediately collapse the cycle's SCC-s into one SCC
- reversed edges (shown in red); reversed-SCC-graph also a DAG
- second DFS runs on reversed-edges (red); once it starts at a high-finish-time (like 16) it can only go through vertices in the same SCC (like abe)





$x = \max f[\,] \to b$

first DFS

Th version 2

$f[u] > f[v]$ (first DFS run)

$u, v \notin$ same SCC

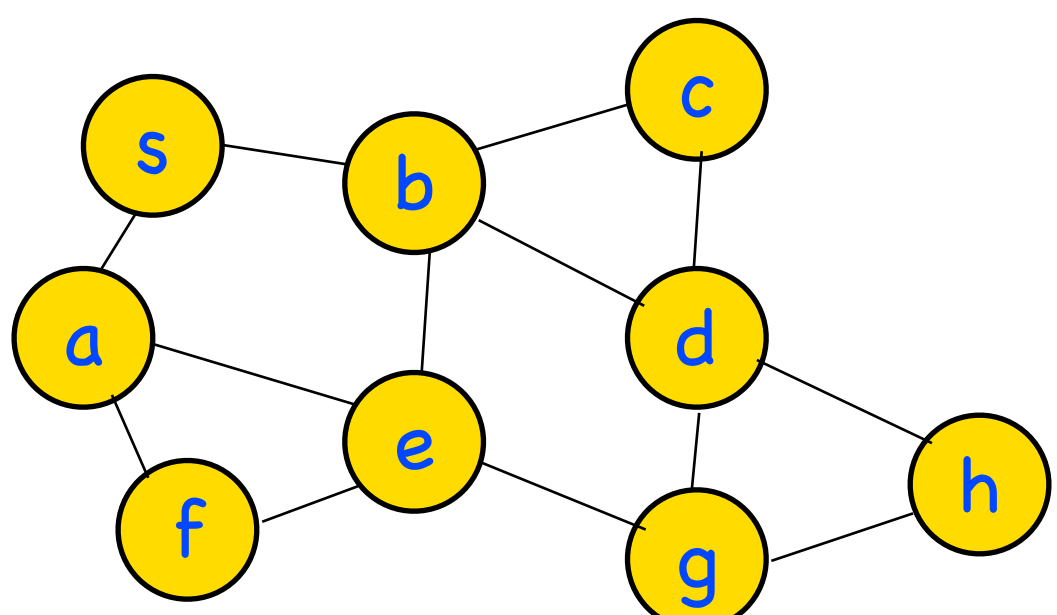
$\Rightarrow$ no path $v \rightsquigarrow u$

Minimum Spanning Trees

Lesson 2

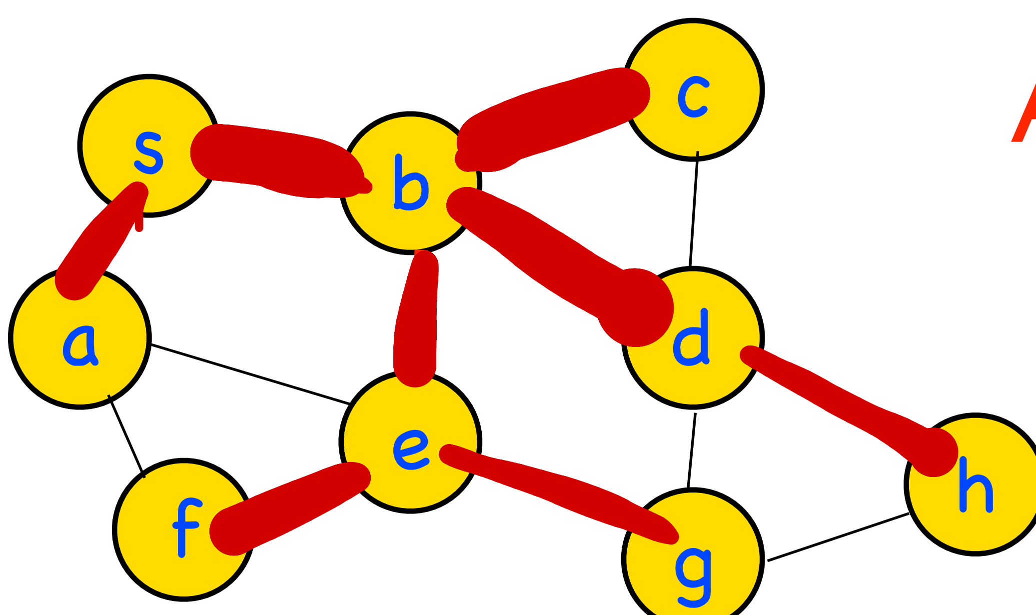
Spanning Trees

- context : undirected graphs
- a set of edges A that "span" or "touch" all vertices, and forms no cycles
  - necessary this set of edges A has size = |V|-1
- spanning tree: the tree formed by the set of spanning edges together with vertex set T = (V,F)



Spanning Trees

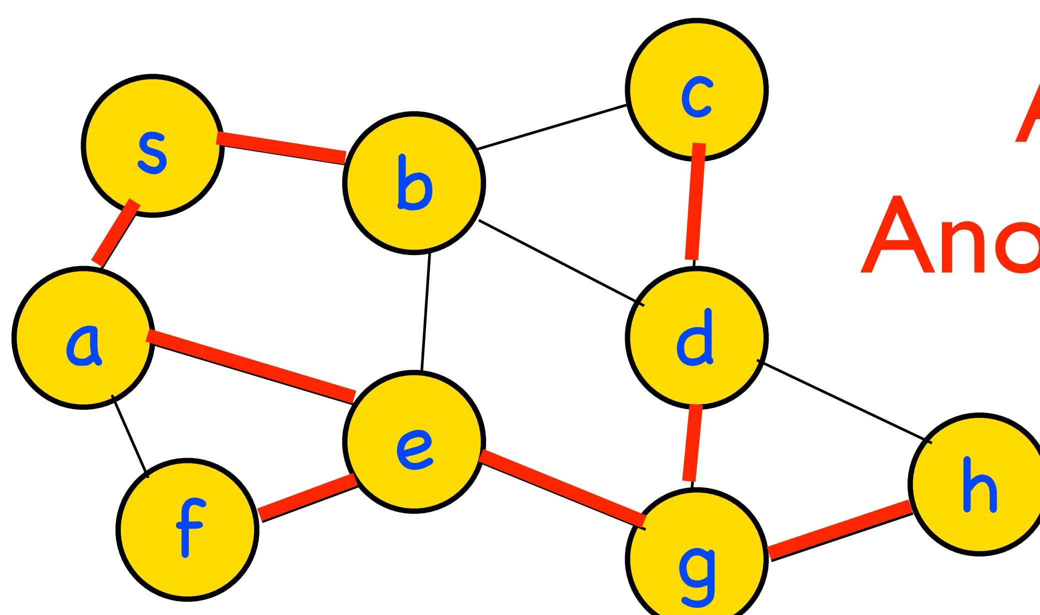
- context : undirected graphs
- a set of edges A that "span" or "touch" all vertices, and forms no cycles
  - necessary this set of edges A has size = |V|-1
- spanning tree: the tree formed by the set of spanning edges together with vertex set T = (V,F)



A spanning tree

Spanning Trees

- context : undirected graphs
- a set of edges A that "span" or "touch" all vertices, and forms no cycles
  - necessary this set of edges A has size = |V|-1
- spanning tree: the tree formed by the set of spanning edges together with vertex set T = (V,F)



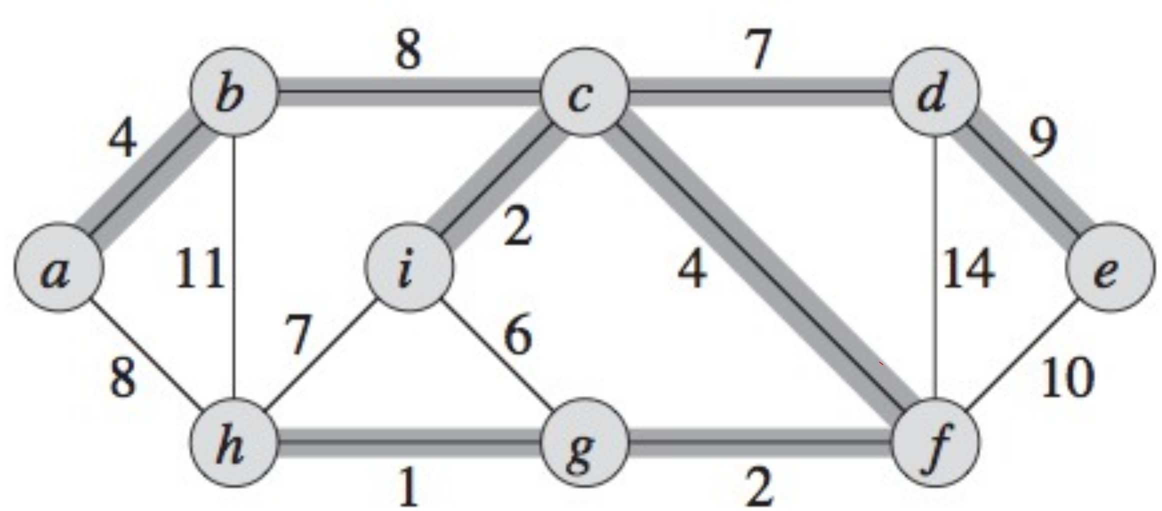
A spanning tree

Another spanning tree

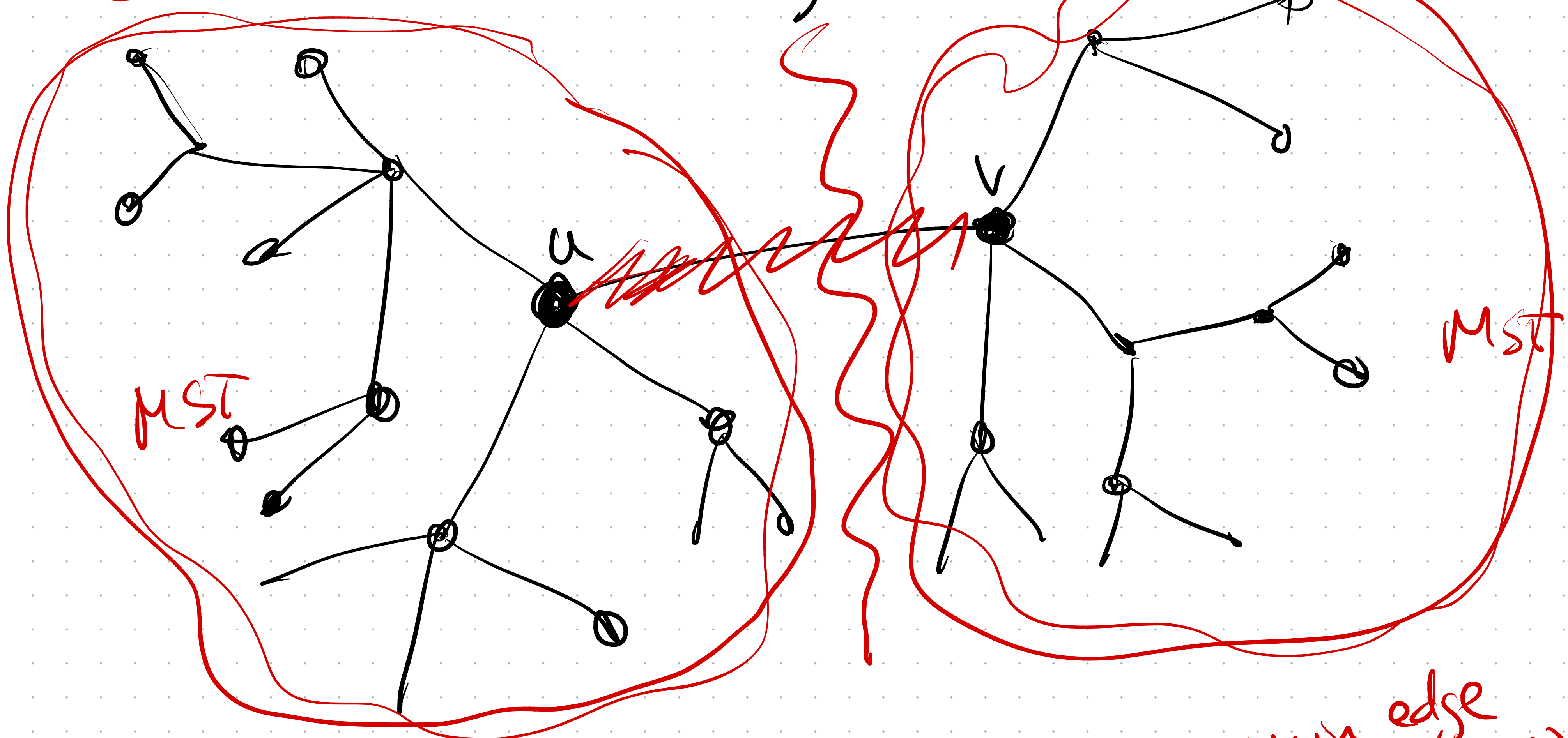
Minimum Spanning Tree (MST)

- context : undirected graph, edges have weights
  - edge $(u,v) \in E$ has weight $w(u,v)$
- MST is a spanning tree of minimum total weight (of its edges)
  - must span all vertices
  - exactly $|V|-1$ edges
  - sum of edges weight be minimum among spanning trees

edge weights ≥ 0?



1. OPTSOL charact. given MST



(u,v) any edge in MST

Th partition G into A and B

existing MST edges → in A
→ or B
(I know ∈ MST)

not across A → B

min edge across (A,B) in part MST

Growing Minimum Spanning Trees

- "safe edge" (u,v) for a given set of edges A: there is a MST that uses A and (u,v)
  - that MST may not be unique

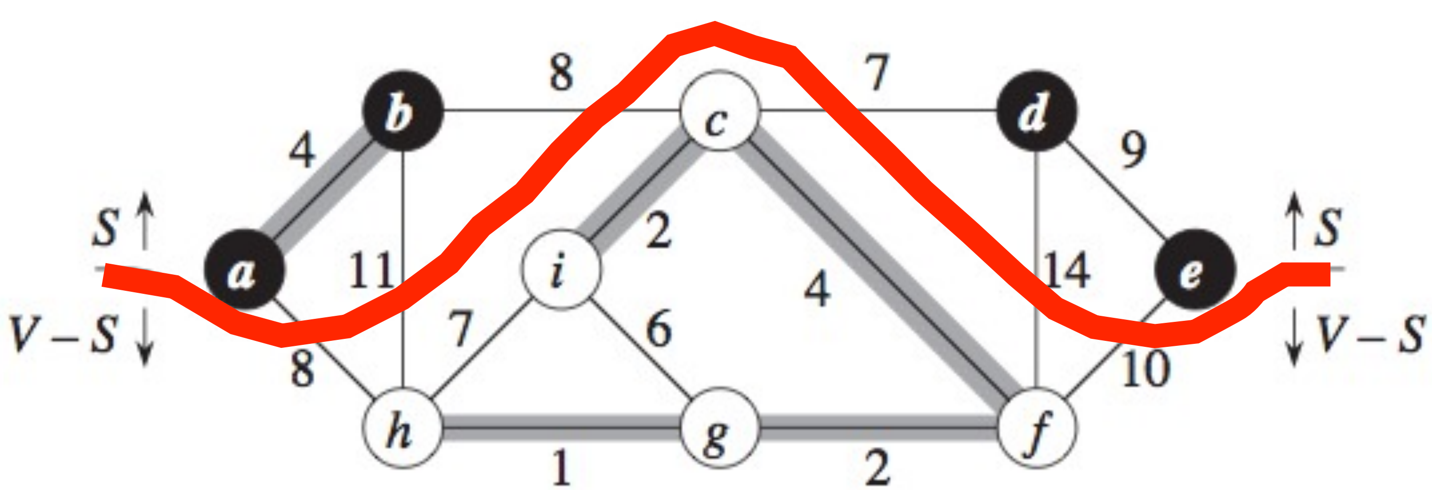
- GENERIC-MST (G)
- A = set of tree edges, initially empty
- while A does not form a spanning tree // *meaning while* |A| < |V|-1
  - find edge (u,v) that is safe for A
  - add (u,v) to A
- end while

- how to find a safe edge to a given set of edges A?
  - Prim algorithm
  - Kruskal algorithm

Th Partition G Into 2 sides

Cuts in the graph

- "cut" is a partition of vertices in two sets : V=S ∪ V-S
- an edge (u,v) crosses the cut (S,V-S) if u and v are on different partitions (one in S the other in V-S)
- cut (S, V-S) respects set of edges A if A has no cross edge
- "min weight cross edge" is a cross edge for the cut, having minimum weight across all cross edges
- Cut Theorem : if A is a set of edges part of some MST, and (S,V-S)a cut respecting A , then a min-weight cross edge is "safe" for A (can be added to A towards an MST)

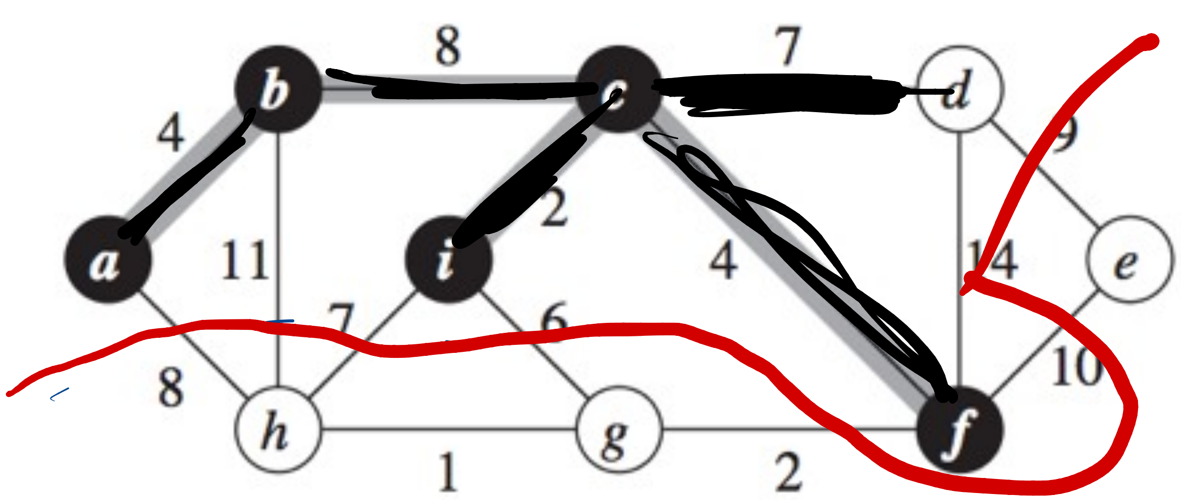


- A={ab, ic, cf, hg, fg}
- cut : S={a,b,d,e} V-S={h,i,c,g,f} respects A
- safe crossing edge : cd, weight(cd)=7

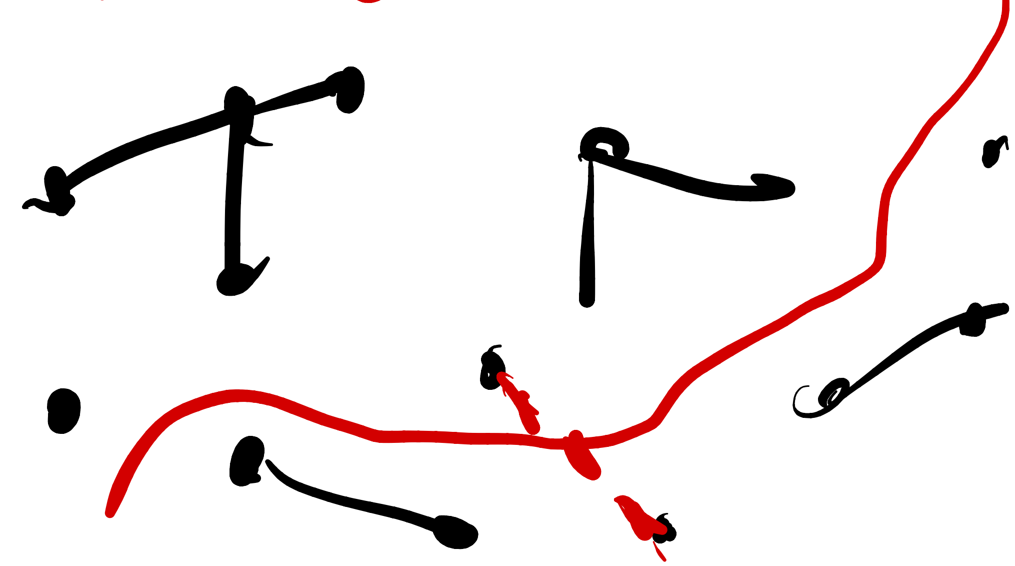
Prim algorithm

- grows a single tree A, S = set of vertices in the tree
  - as opposed to a forest of smaller disconnected trees
- add a safe edge at a time
  - connecting one more node to the current tree

Prim: grows a tree

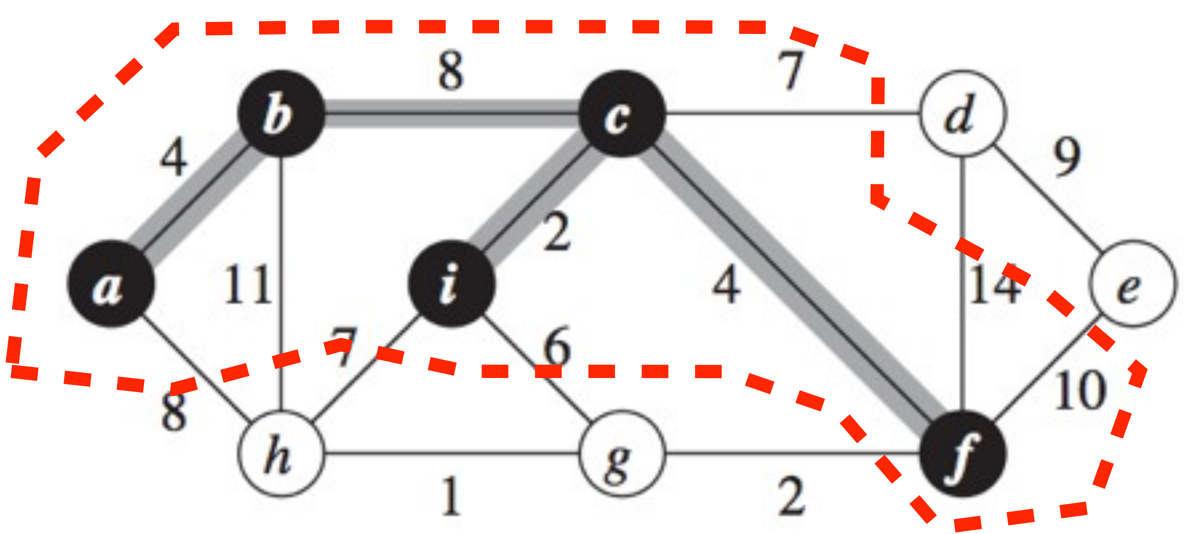


Kruskal: forest



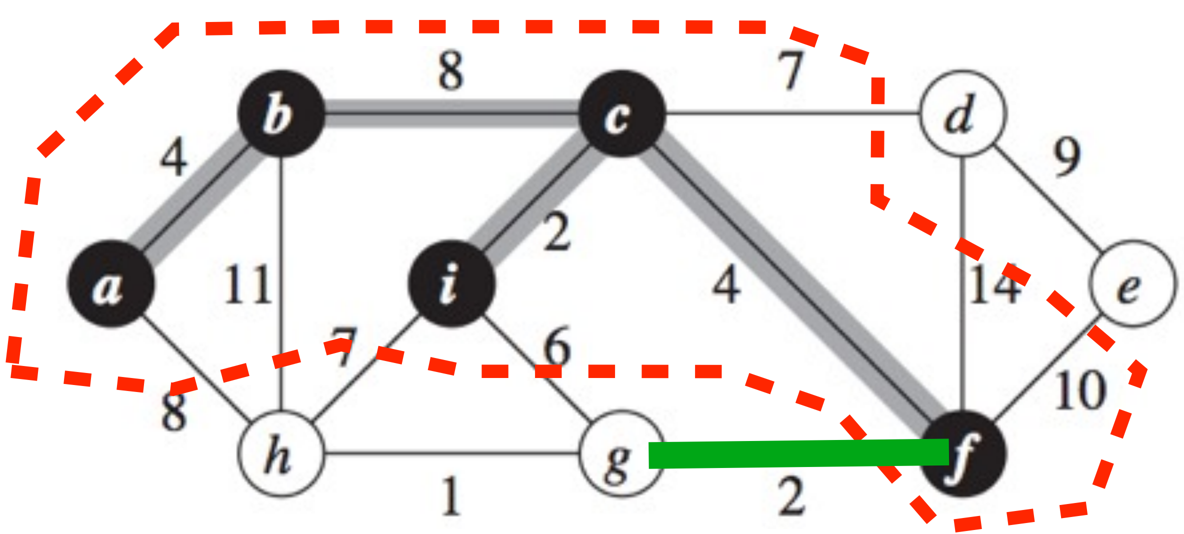
Prim algorithm

- grows a single tree A, S = set of vertices in the tree
  - as opposed to a forest of smaller disconnected trees
- add a safe edge at a time
  - connecting one more node to the current tree
- define cut (S,V-S), which respects A. Using the cut theorem, the min-weight edge across the cut is the next edge added to A



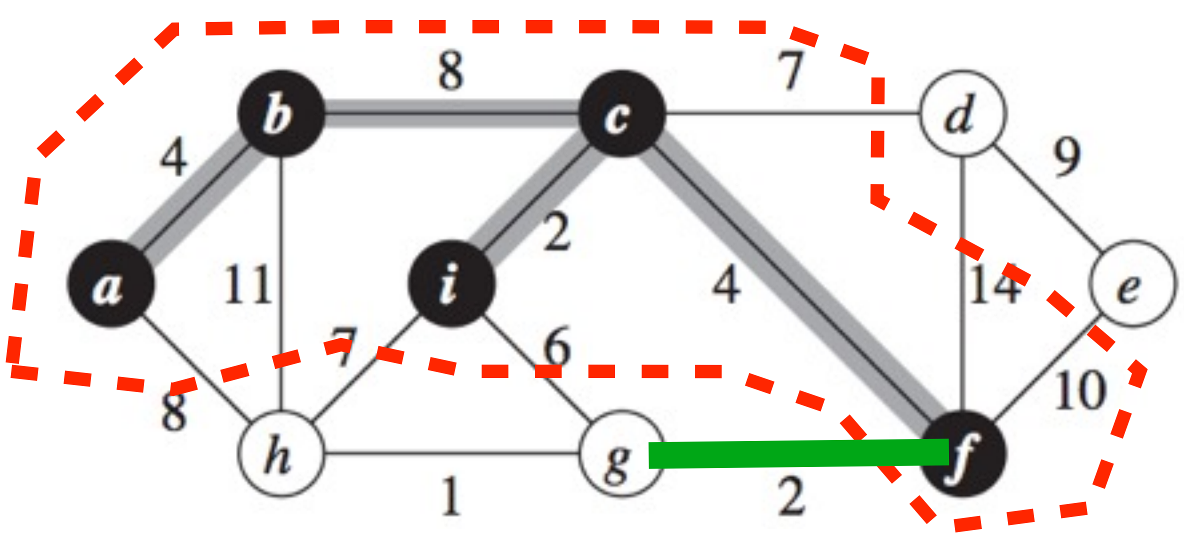
Prim algorithm

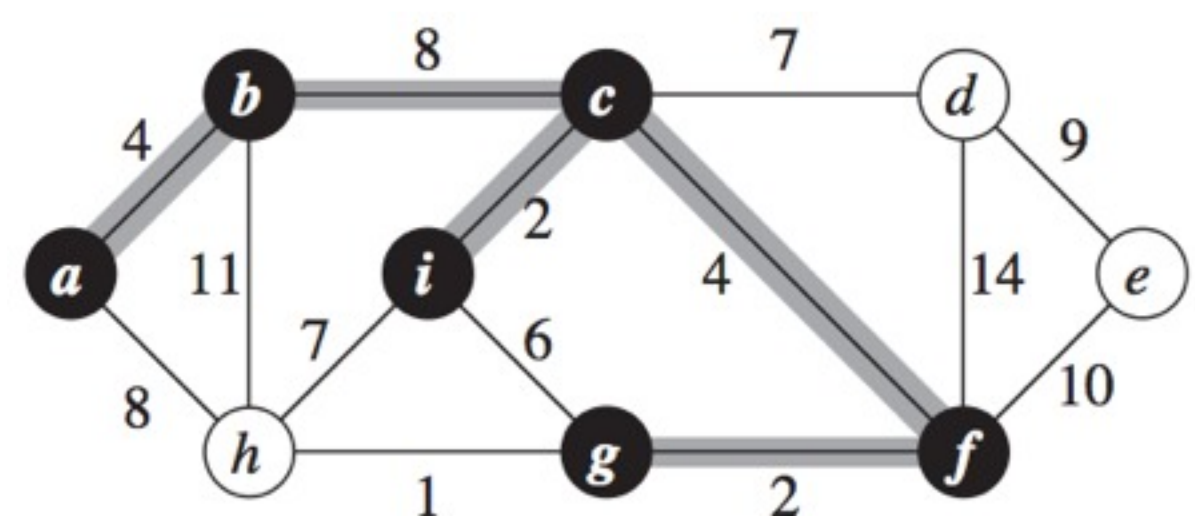
- grows a single tree A, S = set of vertices in the tree
  - as opposed to a forest of smaller disconnected trees
- add a safe edge at a time
  - connecting one more node to the current tree
- define cut (S,V-S), which respects A. Using the cut theorem, the min-weight edge across the cut is the next edge added to A
  - edge gf in the picture is added to A, vertex g added to the tree



Prim algorithm

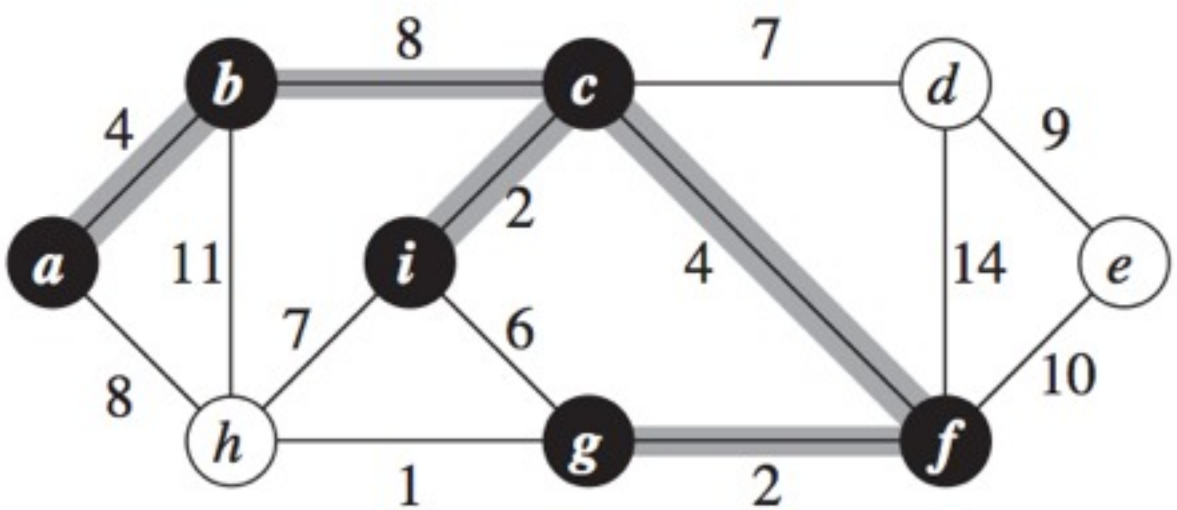
- grows a single tree A, S = set of vertices in the tree
  - as opposed to a forest of smaller disconnected trees
- add a safe edge at a time
  - connecting one more node to the current tree
- define cut (S,V-S), which respects A. Using the cut theorem, the min-weight edge across the cut is the next edge added to A
  - edge gf in the picture is added to A, vertex g added to the tree





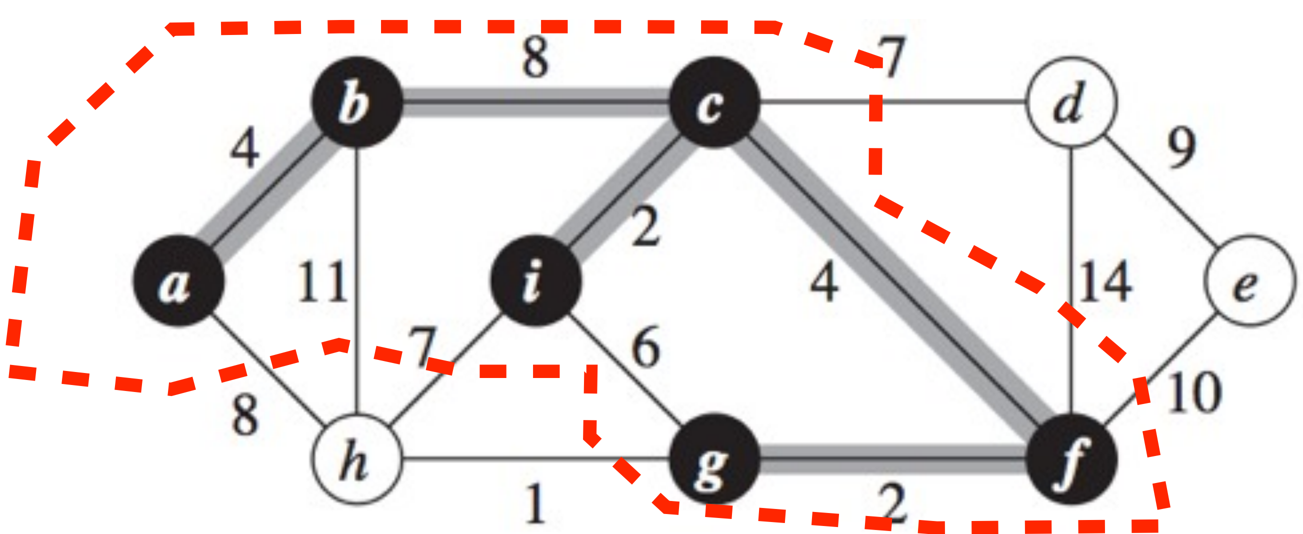
Prim algorithm

- add another(next) safe edge
  - connecting one more node to the current tree



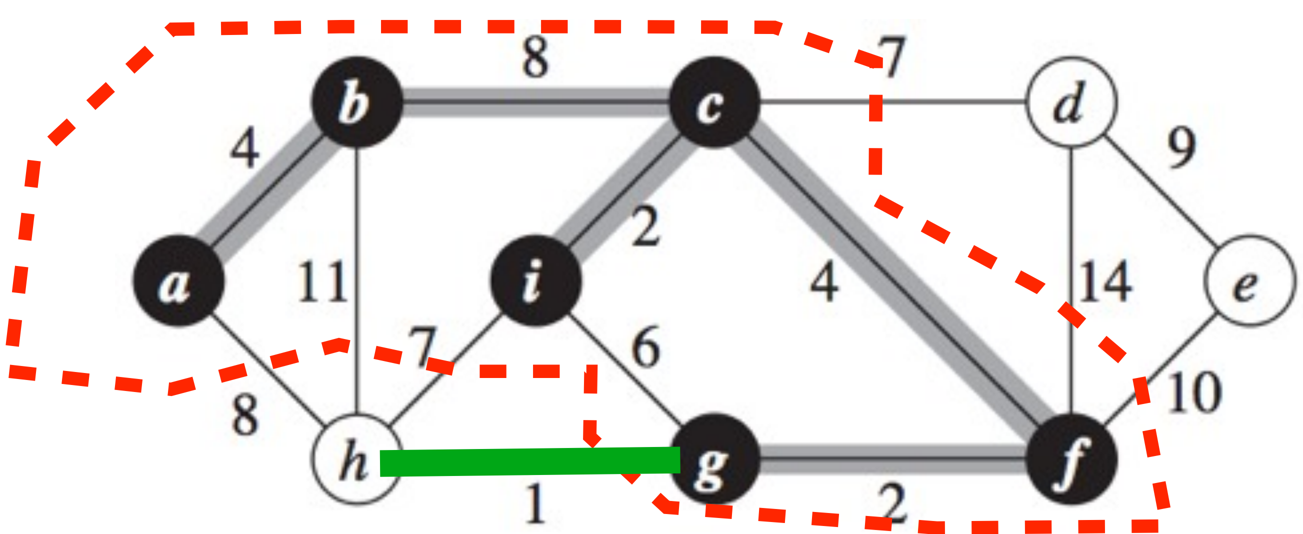
Prim algorithm

- add another(next) safe edge
  - connecting one more node to the current tree
- define cut (S,V-S), which respects A. Using the cut theorem, the min-weight edge across the cut is the next edge added to A



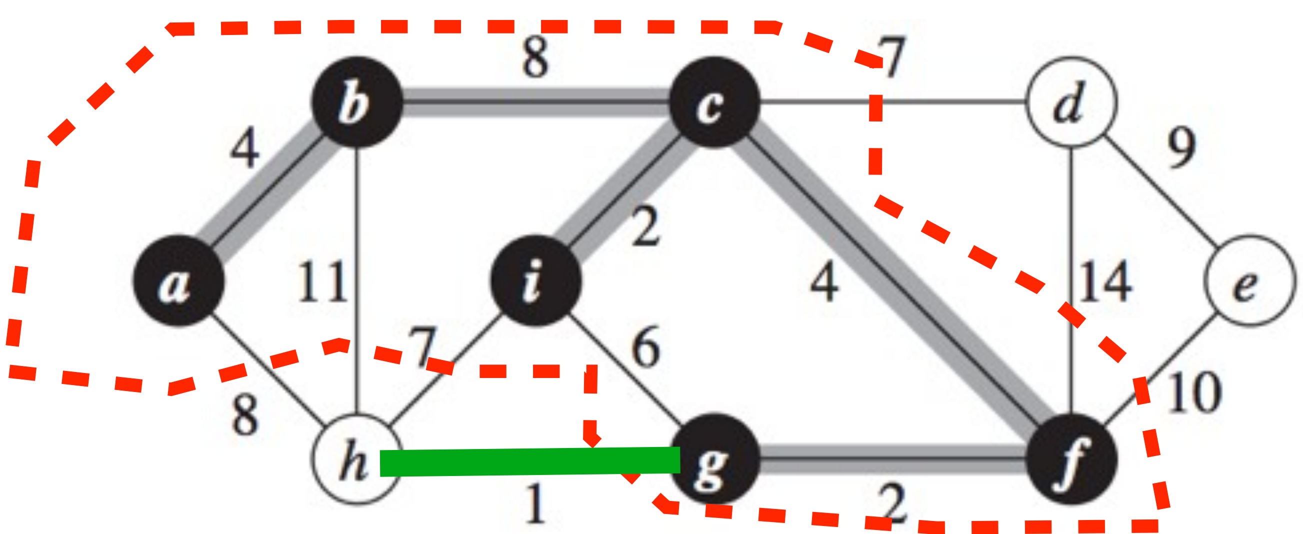
Prim algorithm

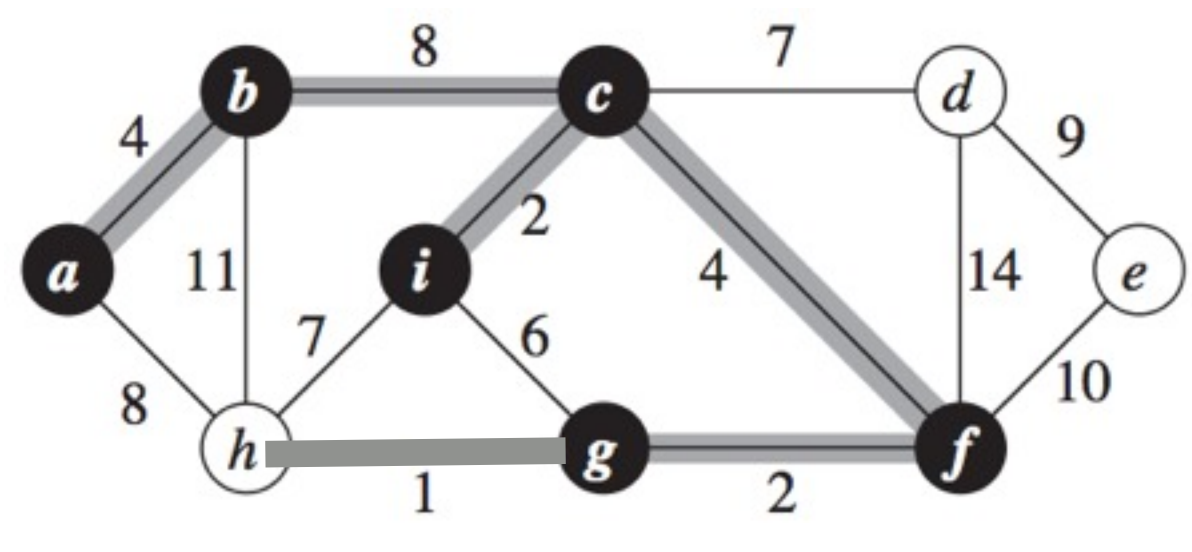
- add another(next) safe edge
  - connecting one more node to the current tree
- define cut (S,V-S), which respects A. Using the cut theorem, the min-weight edge across the cut is the next edge added to A
  - edge hg in the picture is added to A, vertex h added to the tree



Prim algorithm

- add another(next) safe edge
  - connecting one more node to the current tree
- define cut (S,V-S), which respects A. Using the cut theorem, the min-weight edge across the cut is the next edge added to A
  - edge hg in the picture is added to A, vertex h added to the tree





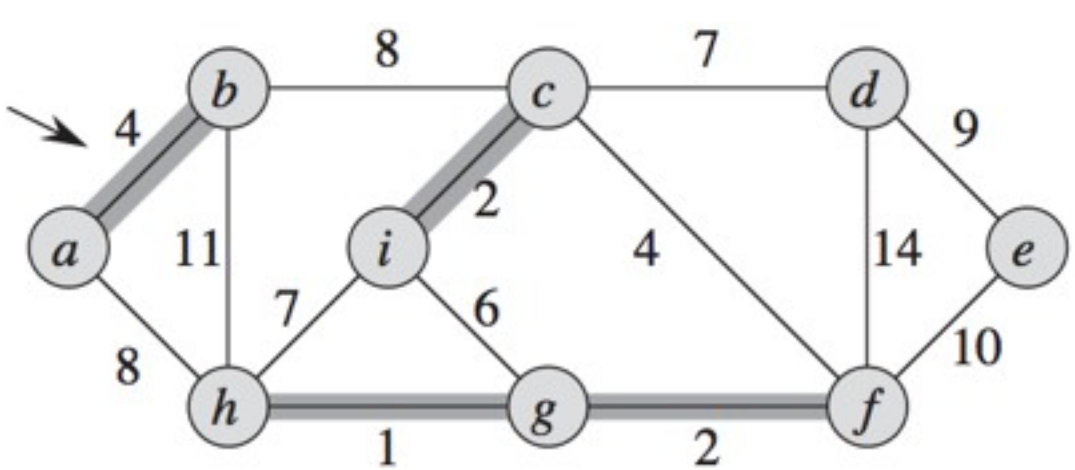
Prim MST algorithm

- Prim simple
  - but implementation a bit tricky
- Running Time depends on implementation of Extract-Min from the Queue
  - best theoretical implementation uses Fibonacci Heaps
  - also the most complicated
  - only makes a practical difference for very large graphs

```
MST-PRIM(G, w, r)
for each u ∈ G.V
    u.key = ∞
    u.π = NIL
r.key = 0
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```

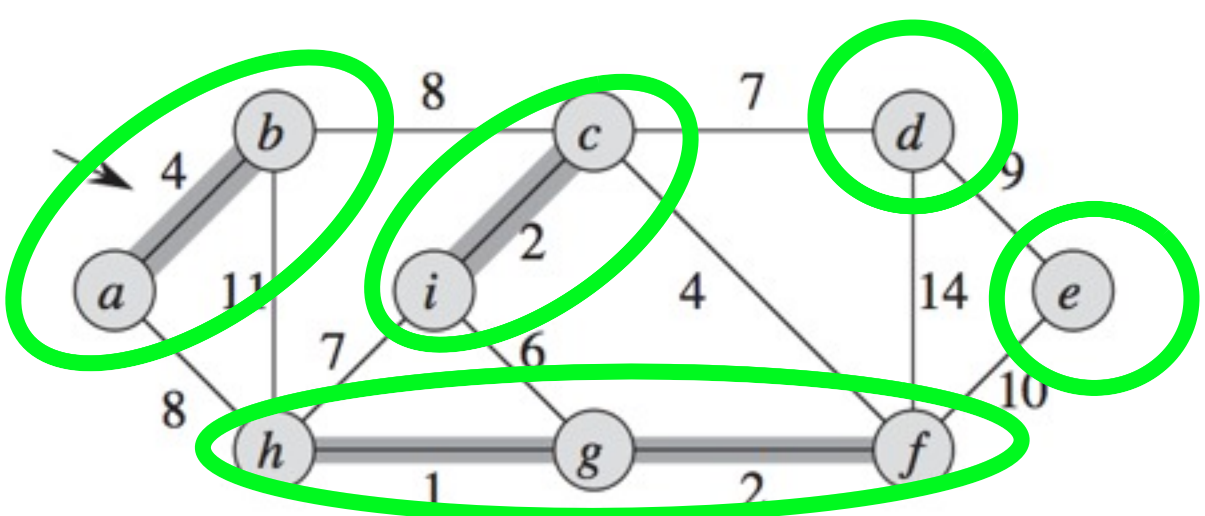
Kruskal MST algorithm

- Grows a forest of trees Forrest = (V,A)
  - eventually all connected into a MST
  - initially each vertex is a tree with no edges, and A is empty



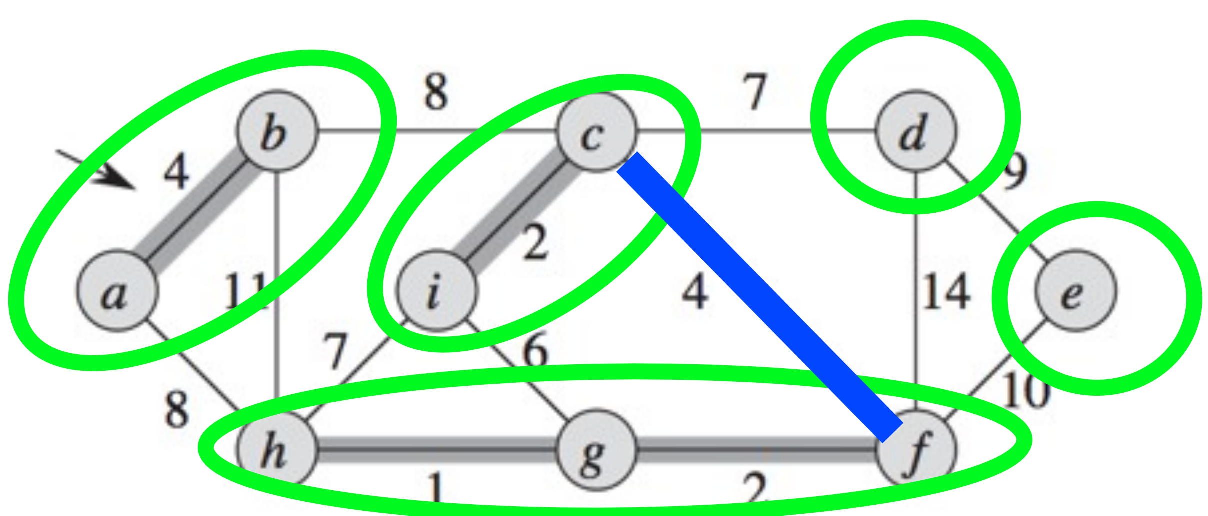
Kruskal MST algorithm

- Grows a forest of trees Forrest = (V,A)
  - eventually all connected into a MST
  - initially each vertex is a tree with no edges, and A is empty
- each edge added connects two trees (or components)



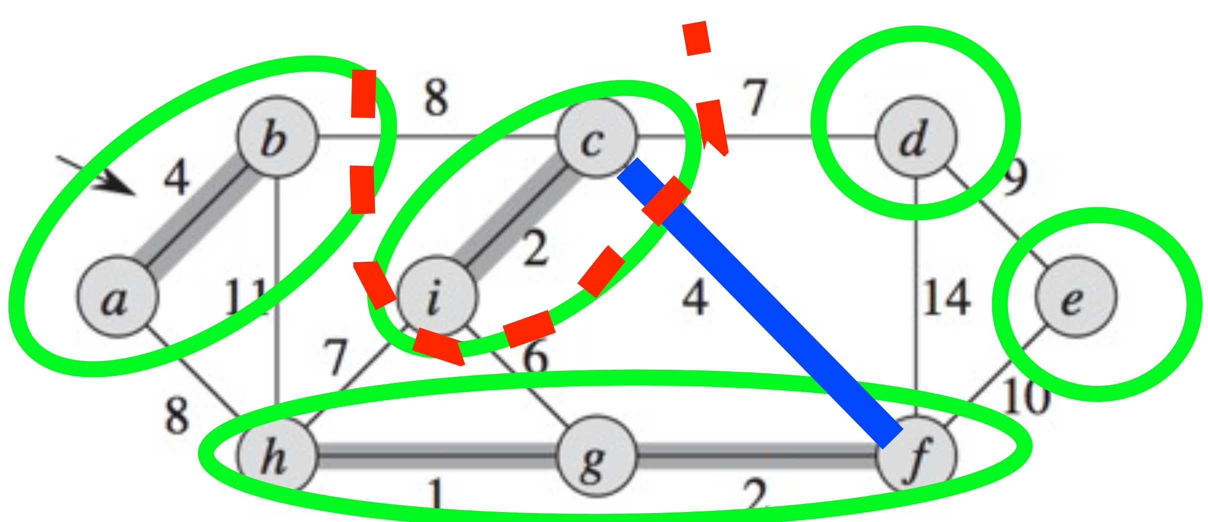
Kruskal MST algorithm

- Grows a forest of trees Forrest = (V,A)
  - eventually all connected into a MST
  - initially each vertex is a tree with no edges, and A is empty
- each edge added connects two trees (or components)
  - find the minimum weight edge (u,v) across two components, say connecting trees T1∋v and T2∋u (edges between nodes of the same trees are no good because they form cycles) (blue in the picture)



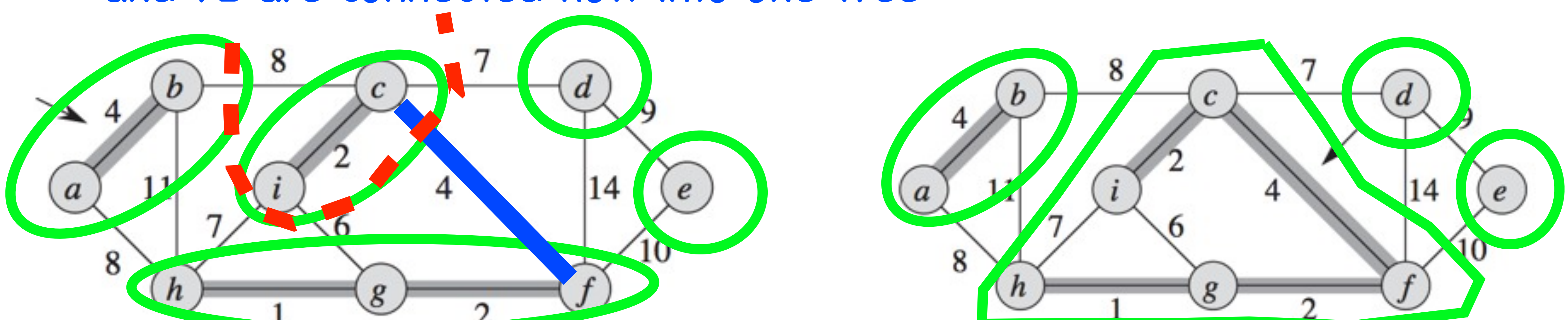
Kruskal MST algorithm

- Grows a forest of trees Forrest = (V,A)
  - eventually all connected into a MST
  - initially each vertex is a tree with no edges, and A is empty
- each edge added connects two trees (or components)
  - find the minimum weight edge (u,v) across two components, say connecting trees T1∋v and T2∋u (edges between nodes of the same trees are no good because they form cycles) (blue in the picture)
  - define cut (S,V-S); S = vertices of T1 (in red). This cut respects set A



Kruskal MST algorithm

- Grows a forest of trees Forrest = (V,A)
  - eventually all connected into a MST
  - initially each vertex is a tree with no edges, and A is empty
- each edge added connects two trees (or components)
  - find the minimum weight edge (u,v) across two components, say connecting trees T1∋v and T2∋u (edges between nodes of the same trees are no good because they form cycles) (blue in the picture)
  - define cut (S,V-S); S = vertices of T1 (in red). This cut respects set A
  - edge (u,v) is the minimum cross edge, thus a safe edge to add to A. T1 and T2 are connected now into one tree



Kruskal algorithm

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ G.V
    MAKE-SET(v)
sort the edges of G.E into nondecreasing order by weight w
for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
return A
```

- Kruskal is simple
- implementation and running time depend on FIND-SET and UNION operations on the disjoint-set forest.
  - chapter 21 in the book, optional material for this course
- running time O(E logV)

MST algorithm comparison

- if you know graph density (edges to vertices)

	Kruskal	Prim with array implement.	Prim w/ binomial heap	Prim w/ Fibonacci heap	in practice
sparse graph $E=O(V)$	$O(V\log V)$	$O(V^2)$	$O(V\log V)$	$O(V\log V)$	Kruskal, or Prim+binom heap
dense graph $E=\Theta(V^2)$	$O(V^2\log V)$	$O(V^2)$	$O(V^2\log V)$	$O(V^2)$	Prim with array
avg density $E=\Theta(V\log V)$	$O(V\log^2 V)$	$O(V^2)$	$O(V\log^2 V)$	$O(V\log V)$	Prim with Fib heap, if graph is large