Graph

Graph Visits

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Search Algorithms

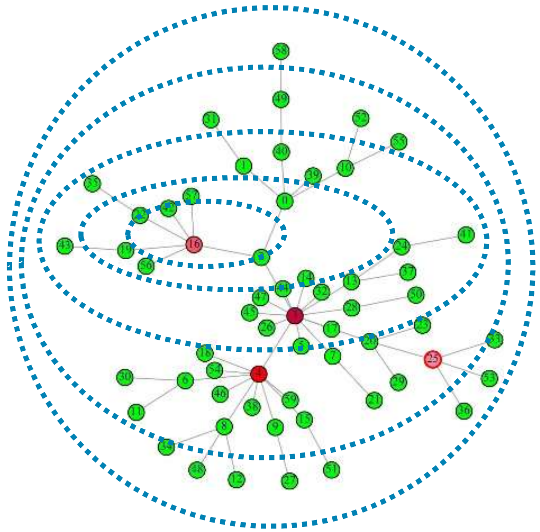
- Searching a graph means systematically following the edgtes of the graph so as to visit the vertices of the graph
  - A graph-searching algorithm can discover much about the structure of a graph
  - Many algorithms
    - Begin by searching their input graph to obtain structural information
    - Are derived from basic searching algorithms

Search Algorithms

- Given a graph $G = (V, E)$ a visit
  - Starts from a given node
  - Follows the edges according to a known strategy
  - Lists the nodes found, possibly adding additional information for each vertex or edge
  - Stops when the entire graph (or the desired part) has been reached

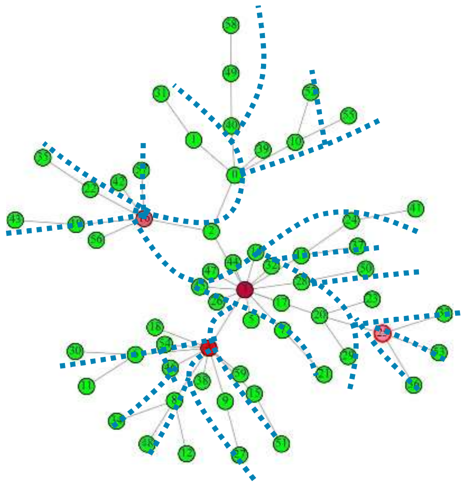
Search Algorithms

- The two most used algorithms to visit a graph are
  - Breadth-First Search (BFS)
    - It visits the graph following its onion-ring shape, i.e., it visits all nodes at a given distance from the source node at the same time before moving the a higher distance
    - Computes the minimum distances
    - Build a BFS tree



Search Algorithms

- Depth-First Search (DFS)
  - It recursively goes in-depth along a given path starting from the source node, before moving to another path
  - Computes the discovery and finishing times
  - Labels all edges
  - Build a DFS tree



Breadth-first search

- Processing the graph in breadth-first means
  - Expanding in parallel the whole border (frontier) between already discovered nodes and not yet discovered nodes
- It starts from a given (source) node s
  - It identifies all nodes reachable from the source node **s**
  - It visits them
  - It moves onto nodes at a higher distance
  - It goes on till it has visited all nodes

Breadth-first search

- Breadth-first search
  - Computes the minimum distance (the shortest path) from **s** to all the nodes reachable from **s**
  - Uses a FIFO queue to store nodes while visiting them

Unreachable nodes from **s** remain unvisited

  - Generates a BFS tree in which all visited (i.e., reached) nodes are finally inserted
    - For each visited node maintain the **parent** (or **predecessor**) using
      - An array of predecessors (one elment for each vertex)
      - A backward reference for each vertex (the **pred** field)

Breadth-first search

- During the visit, breadth-first
  - Generates discovery times for all visited nodes
    - This is the time indicating the first time the node is encountered during the visit
  - Colors nodes depending on their visiting status
    - White nodes
      - Are nodes not yet discovered
    - Gray nodes
      - Are nodes discovered but whose manipulation is not yet complete
    - Black nodes
      - Discovered and completed

Pseudo-code

```
BFS (G, s)
  for each vertex v ∈ V
    v.color = WHITE
    v.dtime = ∞
    v.pred = NULL
  queue_init (Q)
  s.color = GRAY
  s.dtime = 0
  s.pred = NULL
  queue_enqueue (Q, s)
  while (!queue_empty (Q))
     u = queue_dequeue (Q)
     for each v ∈ Adj(u)
       if (v.color == WHITE)
         v.color = GRAY
         v.dtime = u.dtime + 1
         v.pred = u
         queue_enqueue (Q, v)
     u.color = BLACK
```

Init all vertices

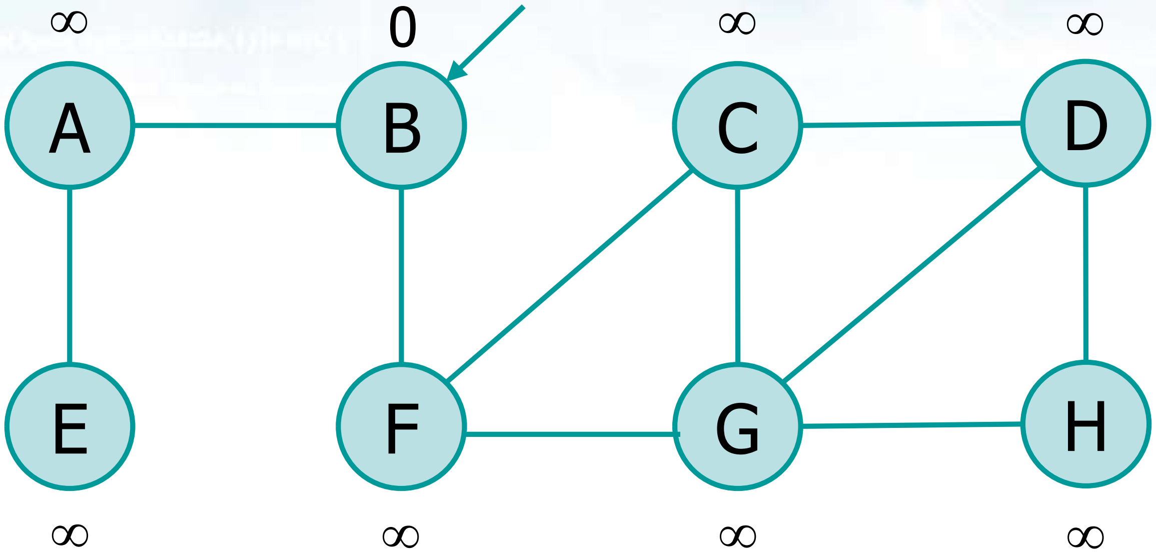
Init source vertex and FIFO queue

While the queue is not empty

Extract next vertex from the queue

For each adjacent vertex

Example



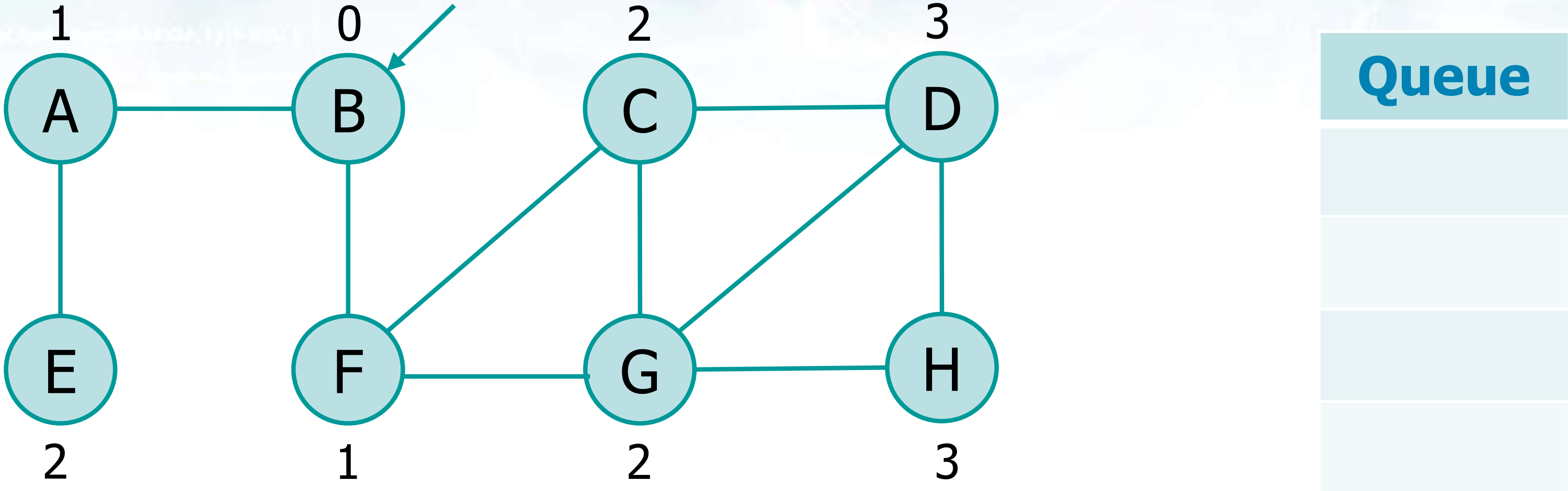
Queue

A	B	C	D	E	F	G	H
0	1	2	3	4	5	6	7
-1	-1	-1	-1	-1	-1	-1	-1

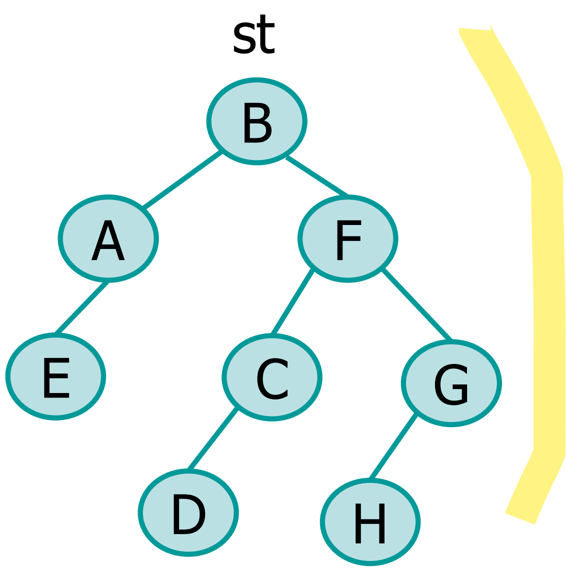
We usually adopt the alphabetic order to generate the same sequence of steps

```
...
while (!queue_empty (Q))
  u = queue_dequeue (Q)
  for each v ∈ Adj(u)
    if (v.color == WHITE)
      v.color = GRAY
      v.dtime = u.dtime + 1
      v.pred = u
      queue_enqueue (Q, v)
  u.color = BLACK
```

Solution



A	B	C	D	E	F	G	H
0	1	2	3	4	5	6	7
1	0	2	3	2	1	2	3



BFS tree
The shortest path from B to H is B, F, G, H, with length = 3

Implementation (with adjacency list)

Client
(code extract)

```
g = graph_load(argv[1]);
printf("Initial vertex? ");
scanf("%d", &i);
src = graph_find(g, i);

graph_attribute_init (g);
graph_bfs (g, src);

n = g->g;
printf ("List of vertices:\n");
while (n != NULL) {
  if (n->color != WHITE) {
    printf("%2d: %d (%d)\n",
      n->id, n->dist, n->pred ? n->pred->id : -1);
  }
  n = n->next;
}

graph_dispose(g);
```

Vertex init: $\forall v \in V$, set color as WHITE discovery time as INT_MAX predecessor as NULL

Print BFS info

Note: Unconnected components remain unvisited

Implementation (with adjacency list)

Function **queue_*** belong to the queue library

```
void graph_bfs (graph_t *g, vertex_t *n) {
  queue_t *qp;
  vertex_t *d;
  edge_t *e;

  qp = queue_init (g->nv);
  n->color = GREY;
  n->dist = 0;
  n->pred = NULL;
  queue_put (qp, (void *)n);
```

Implementation (with adjacency list)

```
  while (!queue_empty_m(qp)) {
    queue_get(qp, (void **)&n);
    e = n->head;
    while (e != NULL) {
      d = e->dst;
      if (d->color == WHITE) {
        d->color = GREY;
        d->dist = n->dist + 1;
        d->pred = n;
        queue_put (qp, (void *)d);
      }
      e = e->next;
    }
    n->color = BLACK;
  }
  queue_dispose (qp, NULL);
}
```

If the queue is not empty

Extract vertex on head and visit its adjacency list

And more specifically all adjancent white nodes

Nodes on the frontier are grey

Nodes managed are back

Complexity

```
BFS (G, s)
  for each vertex v ∈ V
    v.color = WHITE
    v.dtime = ∞
    v.pred = NULL
  queue_init (Q)
  s.color = GRAY
  s.dtime = 0
  s.pred = NULL
  queue_enqueue (Q, s)
  while (!queue_empty (Q))
    u = queue_dequeue (Q)
    for each v ∈ Adj(u)
      if (v.color == WHITE)
        v.color = GRAY
        v.dtime = u.dtime + 1
        v.pred = u
        queue_enqueue (Q, v)
    u.color = BLACK
```

For each vertex O(1)
For all vertices O(|V|)

The cost to enqueue and dequeue a vertex is O(1)
Each vertex is inserted and extract from the queue
For all vertices O(|V|)

The procedure scans all adjacency lists
The sum of the length of all lists is $\Theta(|E|)$
The cost to manage them is O(|E|)
Notice that the cost is O(|E|) not $\Theta(|E|)$ because we visit only the connected component including the starting vertex not the entire graph

Complexity

```
BFS (G, s)
  for each vertex v ∈ V
    v.color = WHITE
    v.dtime = ∞
    v.pred = NULL
  queue_init (Q)
  s.color = GRAY
  s.dtime = 0
  s.pred = NULL
  queue_enqueue (Q, s)
  while (!queue_empty (Q))
     u = queue_dequeue (Q)
     for each v ∈ Adj(u)
       if (v.color == WHITE)
         v.color = GRAY
         v.dtime = u.dtime + 1
         v.pred = u
         queue_enqueue (Q, v)
     u.color = BLACK
```

Globally the cost is given by

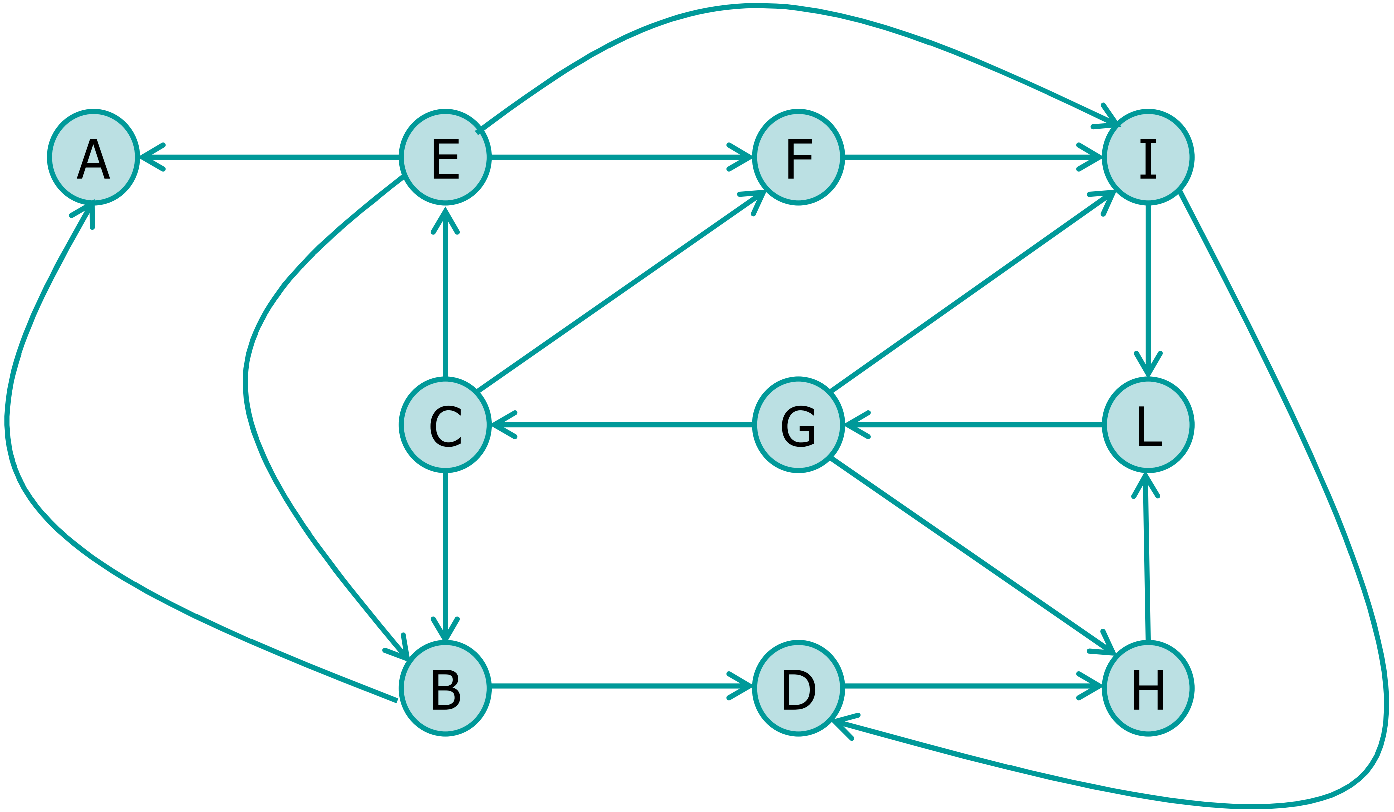
Init and queue → $O(|V|)$
Adjacency lists → $O(|E|)$

Thus → $T(n) = O(|V|+|E|)$

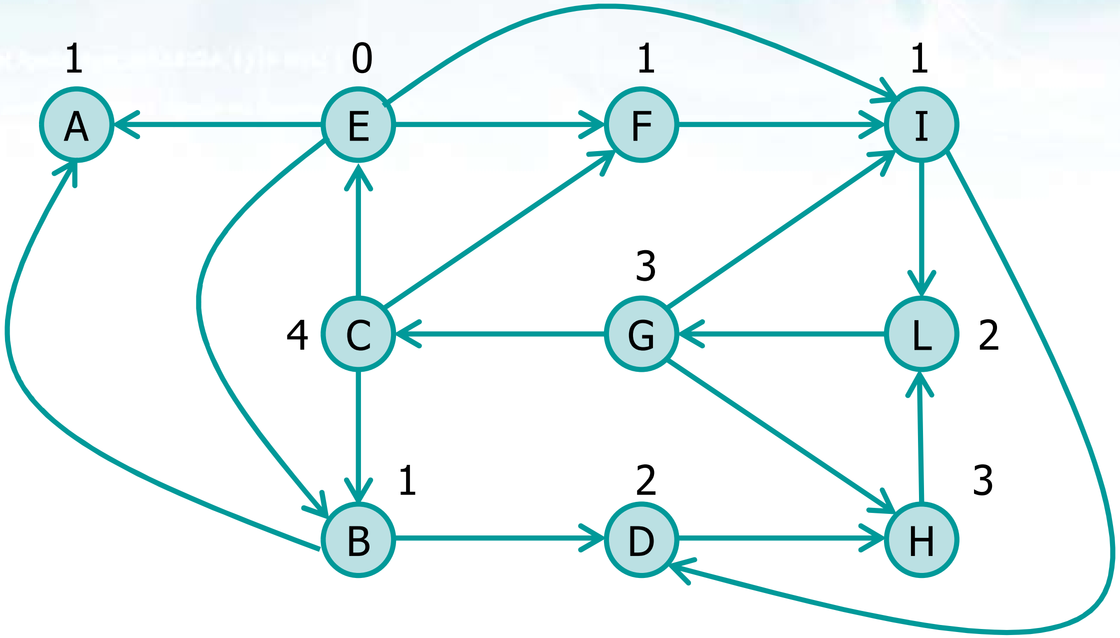
Exercise

- Given the following graph, visit it Breadth-First starting from vertex E
  - Report the resulting BFS tree

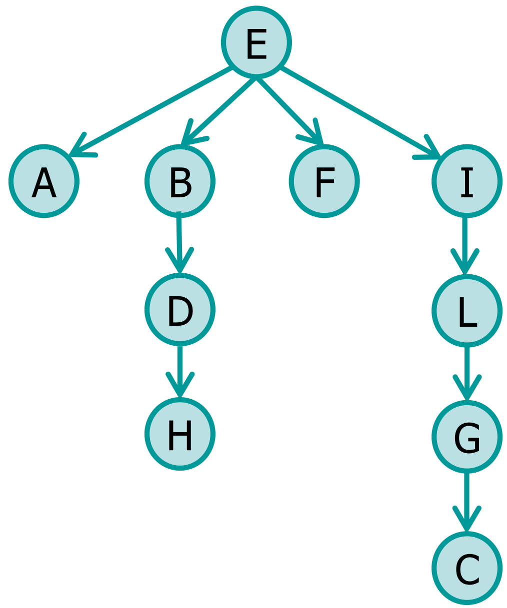
to solve it use the white -> grey technique



Solution

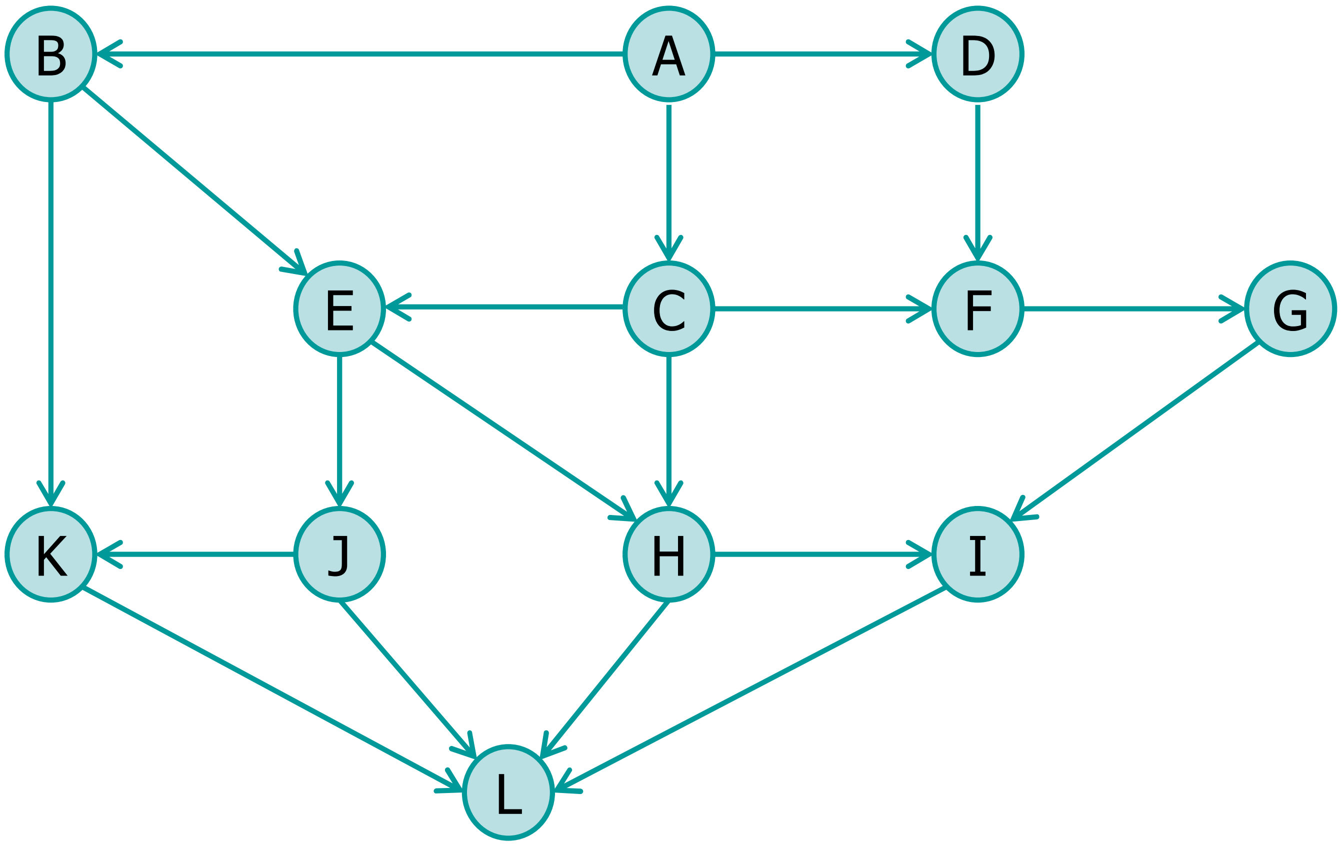



```
Node=[ 0] A Discovery_time= 1 Predecessor=[ 4]
Node=[ 1] B Discovery_time= 1 Predecessor=[ 4]
Node=[ 2] C Discovery_time= 4 Predecessor=[ 6]
Node=[ 3] D Discovery_time= 2 Predecessor=[ 1]
Node=[ 4] E Discovery_time= 0 Predecessor=[-1]
Node=[ 5] F Discovery_time= 1 Predecessor=[ 4]
Node=[ 6] G Discovery_time= 3 Predecessor=[ 9]
Node=[ 7] H Discovery_time= 3 Predecessor=[ 3]
Node=[ 8] I Discovery_time= 1 Predecessor=[ 4]
Node=[ 9] L Discovery_time= 2 Predecessor=[ 8]
```

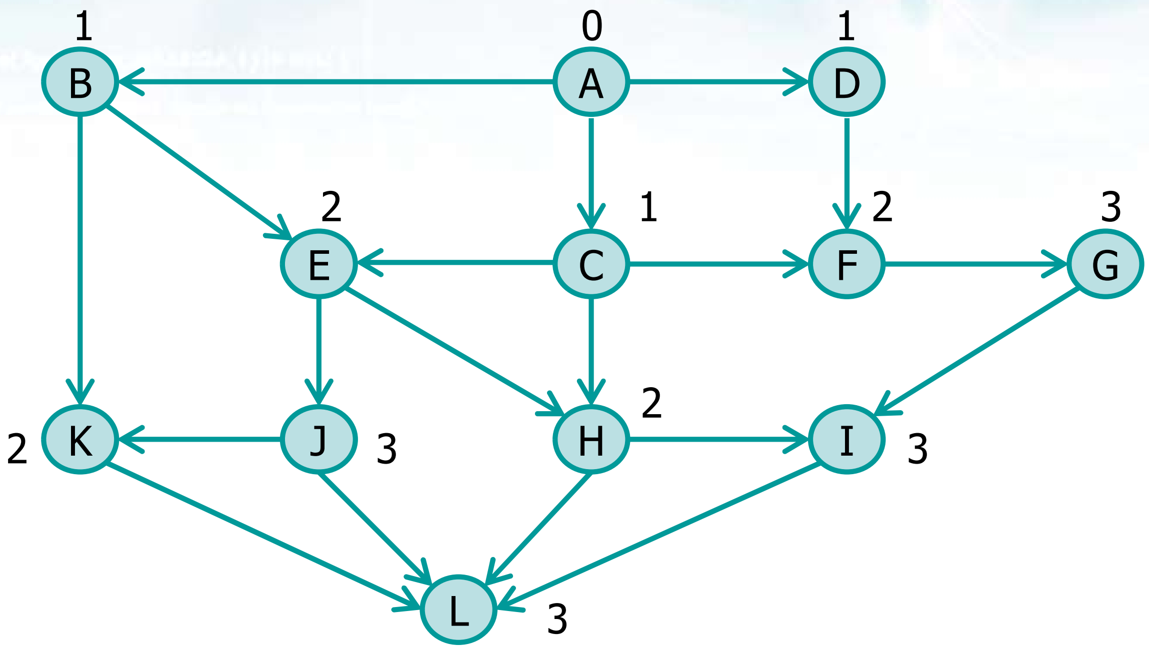


Exercise

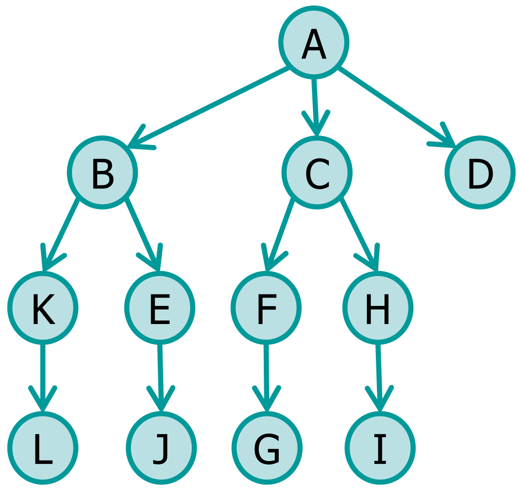
- Given the following graph, visit it Breadth-First starting from vertex A
  - Report the resulting BFS tree



Solution




```
Node=[ 0] A Discovery_time= 0 Predecessor=[-1]
Node=[ 1] B Discovery_time= 1 Predecessor=[ 0]
Node=[ 2] C Discovery_time= 1 Predecessor=[ 0]
Node=[ 3] D Discovery_time= 1 Predecessor=[ 0]
Node=[ 4] E Discovery_time= 2 Predecessor=[ 1]
Node=[ 5] F Discovery_time= 2 Predecessor=[ 2]
Node=[ 6] G Discovery_time= 3 Predecessor=[ 5]
Node=[ 7] H Discovery_time= 2 Predecessor=[ 2]
Node=[ 8] I Discovery_time= 3 Predecessor=[ 7]
Node=[ 9] J Discovery_time= 3 Predecessor=[ 4]
Node=[10] K Discovery_time= 2 Predecessor=[ 1]
Node=[11] L Discovery_time= 3 Predecessor=[10]
```



Depth-first search

- Given a connected (or unconnected) graph, starting from a source node **s**
  - It expands the last discovered node that has still undiscovered adjacent nodes
    - It searches deeper in the graph whenever possible
  - It visits all the nodes of the graph
    - No matter they are reachable from **s** or not
    - It **restarts** (from an unreached nodes) if not all nodes have been reached

DFS differs from BFS (even if BFS can be modified at will)

Depth-first search

- During the visit graph nodes are conceptually classified as
  - White
    - Not yet discovered nodes
  - Gray
    - Already discovered, but not yet completed
  - Black
    - Discovered and completed

Depth-first search

- It labels each node with two timestamps and a flag
  - Timestamps are discrete times with time that evolves according to a counter time
  - Its discovery time
    - The first time the node is encountered in the visit during the recursive descent, in pre-order visit
  - Its endprocessing or finishing or completion or quit time
    - The end of node processing, when the procedure exit from recursion, in post-order visit
  - The flag defines the node's parent in the depth-first visit

Depth-first search

- It labels each edge with an attribute, describing the edge a
  - T(ree), B(ackward), F(orward), C(ross)
    - For directed graphs
  - T(ree), B(ackward)
    - For undirected graphs
      - Forward edges become Backward edges
      - Cross edges become Tree edges
- It generates a forest of DFS trees

Pseudo-code

```
DFS (G)
  for each vertex v ∈ V
    v.color = WHITE
    v.dtime = v.endtime = ∞
    v.pred = NULL
  time = 0
  for each vertex v ∈ V
    if (v.color = WHITE)
      DFS_r (G, v)
DFS_r (G, u)
  time++
  u.dtime = time
  u.color = GRAY
  for each v ∈ Adj(u)
        if (v.color == WHITE)
          v.pred = u
          DFS_r (G, v)
  u.color = BLACK
  time++
  u.endtime = time
```

Init all vertices

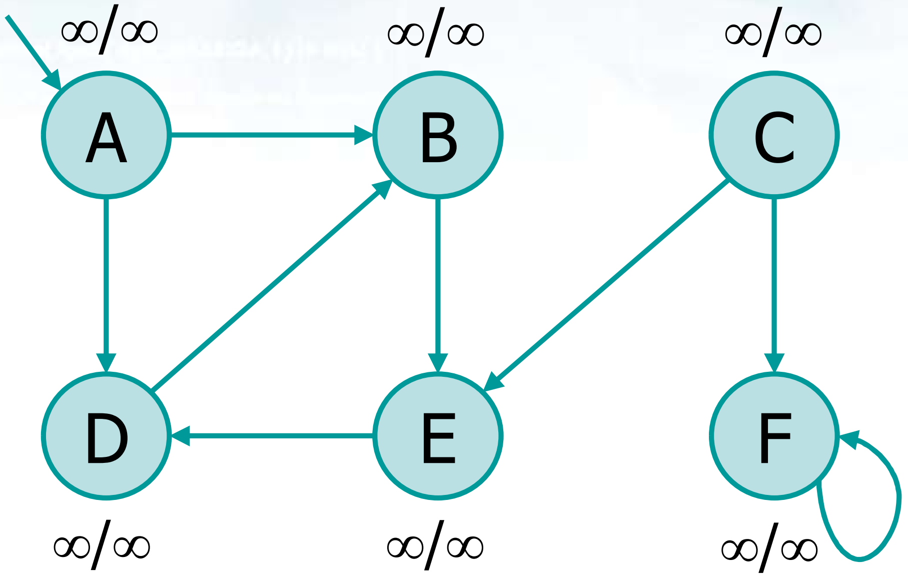
For eacfh possibile source vertex call recursive function

Set node attributes

Recur

Set node attributes

Example



A	B	C	D	E	F
0	1	2	3	4	5
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1

We usually adopt the alphabetic order to generate the same sequence of steps

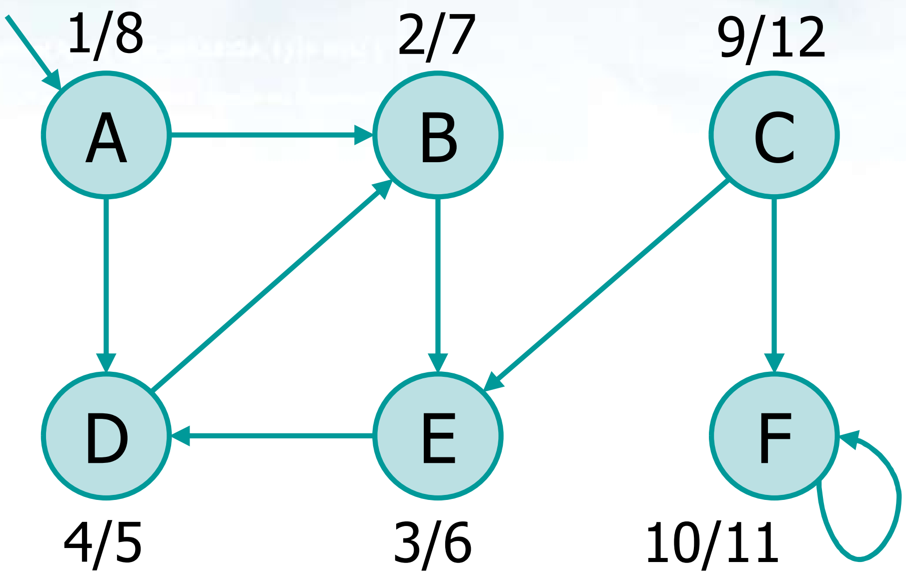
```
DFS_r (G, u)
  time++
  u.dtime = time
  u.color = GRAY
  for each v ∈ Adj(u)
       if (v.color == WHITE)
         v.pred = u
         DFS_r (G, v)
  u.color = BLACK
  time++
  u.endtime = time
```

Solution

move into B because I am using alphabetic order

grey/black
(black obtained by recursively going backwards)

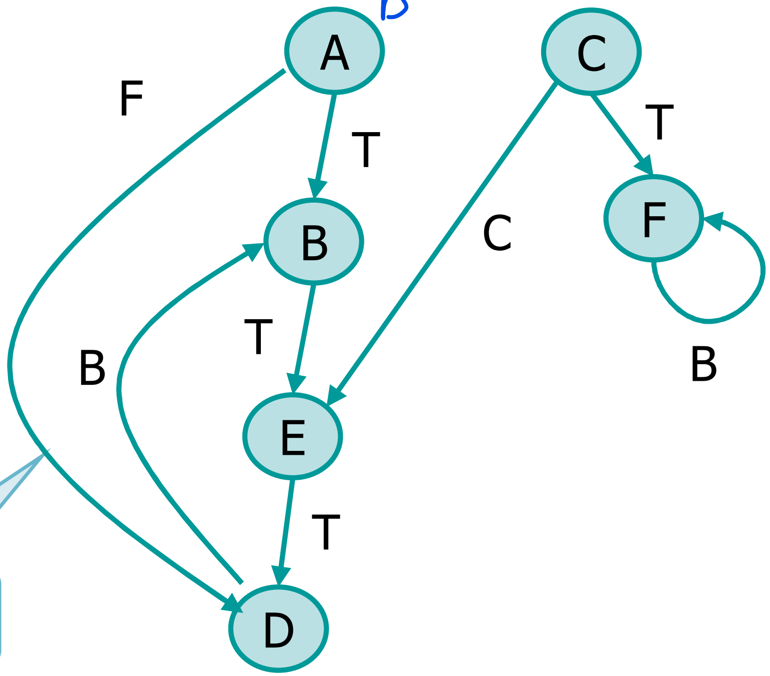
for those disconnected I recall the function



A	B	C	D	E	F
0	1	2	3	4	5
1	2	9	4	3	10
8	7	12	5	6	11

A ↓ B ↓ E ↓ D

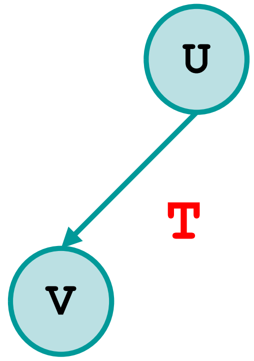
C ↓ F



DFS tree

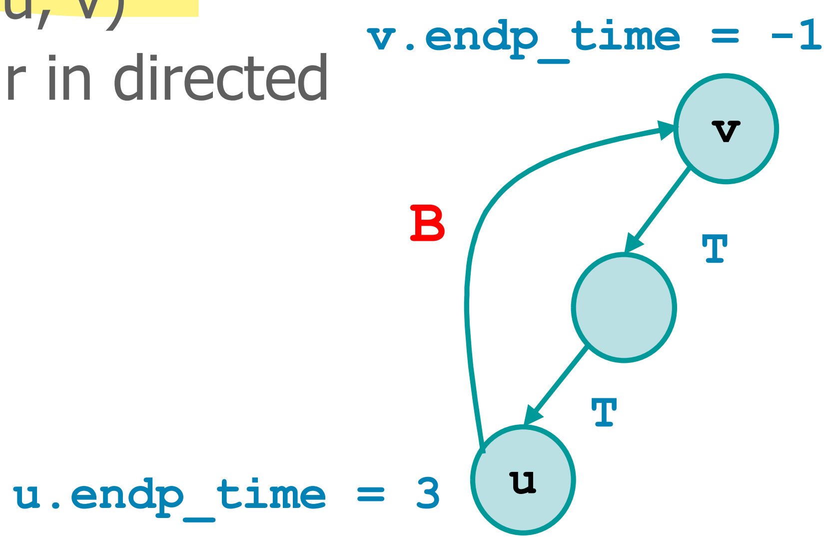
Edge labelling in directed graphs

- Given a directed graph and an edge (u, v)
  - A tree (T) edge is an edge of the DFS forest
    - The edge (u,v) is a T edge if
      - Vertex v is discovered by exploring edge (u,v)
      - Vertex v is **WHITE** when reached with edge (u, v)



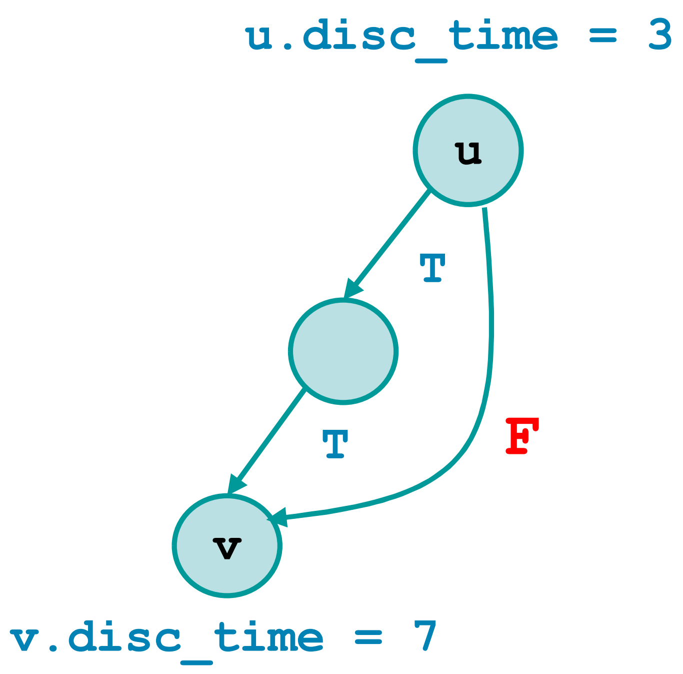
Edge labelling in directed graphs

- Given a directed graph and an edge (u, v)
  - A back (B) edge is an edge connecting a vertex u to an ancestor v in a depth-first tree
    - As (u, v) is reaching an ancestor
      - When visited, v.endp_time is not defined
      - At the end of the visit, it will be
        - v.endp_time > u.endp_time
    - The edge (u,v) is a B edge if the vertex v is **GRAY** when reached with edge (u, v)
    - Self-loop (which may occur in directed graphs) are B edges



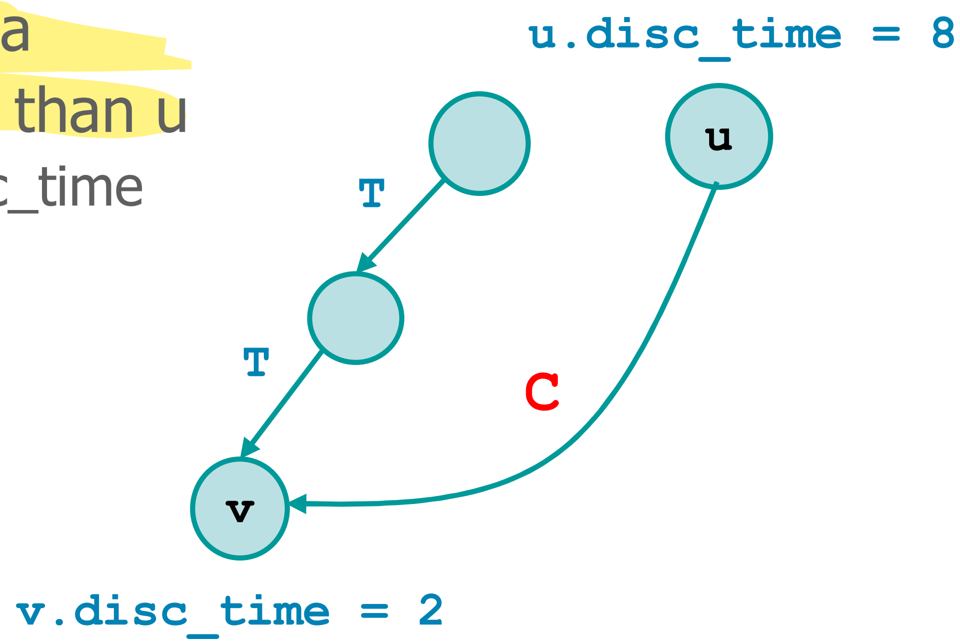
Edge labelling in directed graphs

- Given a directed graph and an edge (u, v)
  - A forward (F) edge is a nontree edge connecting a vertex u to a descendant v in a depth-first tree
    - The edge (u, v) is a F edge if the vertex v is **BLACK and** it has a **higher** discovery time than u
      - v.disc_time > u.disc_time



Edge labelling in directed graphs

- Given a directed graph and an edge (u, v)
  - A cross (C) edge is one of the other edges
    - A cross edge can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees
    - The edge (u,v) is a C edge if the vertex v is **BLACK and** it has a **lower** discovery time than u
      - v.disc_time < u.disc_time

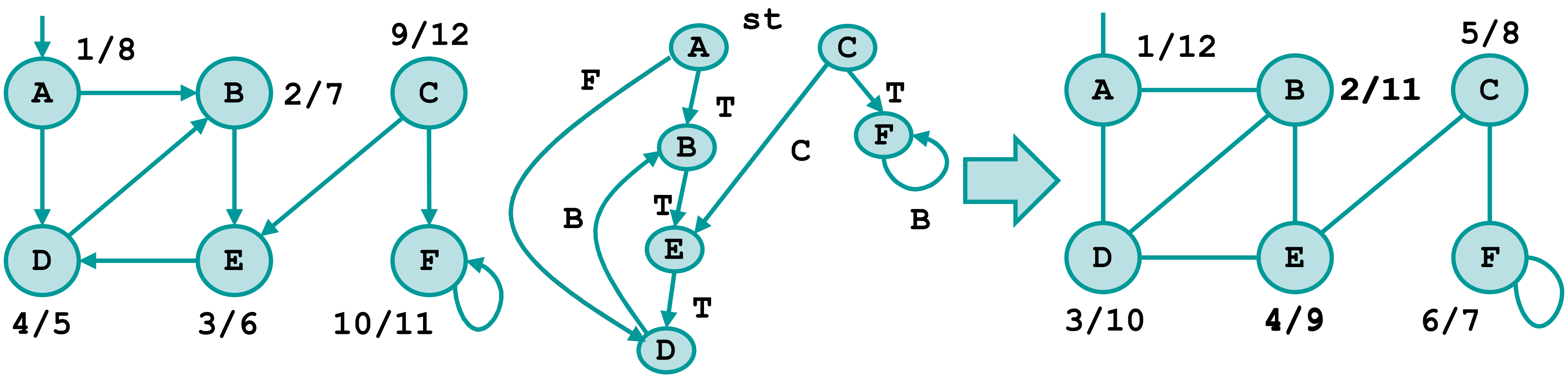


Edge labelling in undirected graphs

- For undirected graphs, since (u, v) and (v, u) are really the same edge, we may have some ambiguity in how edges are classified
- In every undirected graph, every edge is either a tree (T) or a back (B) edge
- The definitions may be derived from the previous ones

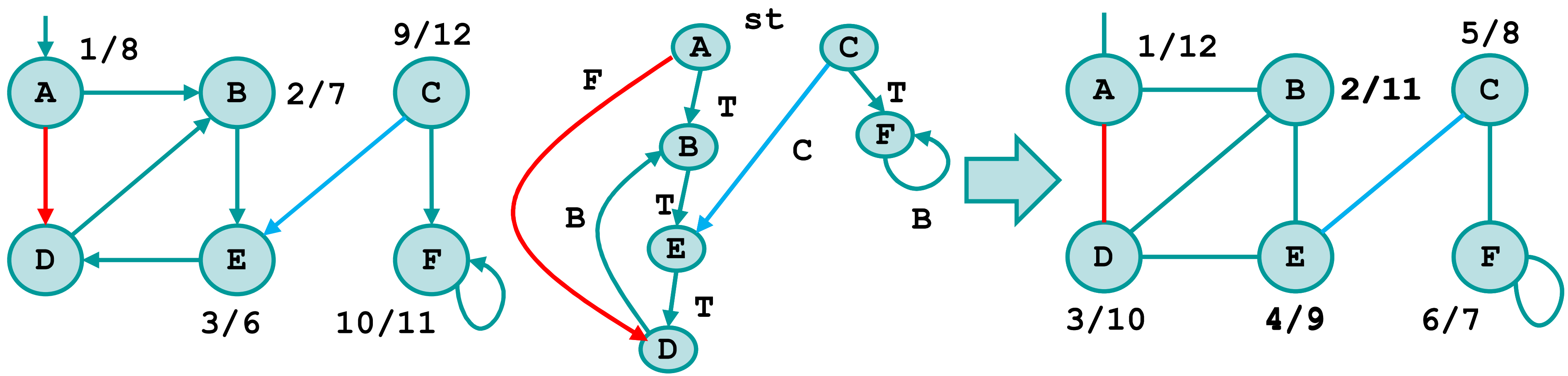
Edge labelling in undirected graphs

- Tree edges are defined as before
  - Towards a WHITE vertex
- Backward edges are defined as before
  - Towards a GRAY vertex



Edge labelling in undirected graphs

- As each edge can be traversed both ways
  - **Forward** edges do not exist, as they are traversed "before" from v to u when they are just **Backward** edges and
    - v.disc_time > u.disc_time
  - **Cross** edges do not exist, as they are traversed "before" from v to u when they are just **Tree** edges and
    - v.disc_time < u.disc_time



Implementation (with adjacency list)

Client (extract)

Vertex init: ∀v∈V, set
color as WHITE
discovery and finisching times as INT_MAX
predecessor as NULL

```
g = graph_load (argv[1]);

printf ("Initial vertex? ");
scanf ("%d", &i);

src = graph_find (g, i);

graph_attribute_init (g);
graph_dfs (g, src);

graph_dispose (g);
```

DFS
(recursive function)

Implementation (with adjacency list)

```
void graph_dfs (graph_t *g, vertex_t *n) {
  int currTime=0;
  vertex_t *tmp, *tmp2;

  printf("List of edges:\n");
  currTime = graph_dfs_r (g, n, currTime);
  for (tmp=g->g; tmp!=NULL; tmp=tmp->next) {
    if (tmp->color == WHITE) {
      currTime = graph_dfs_r (g, tmp, currTime);
    }
  }

  printf("List of vertices:\n");
  for (tmp=g->g; tmp!=NULL; tmp=tmp->next) {
    tmp2 = tmp->pred;
    printf("%2d: %2d/%2d (%d)\n",
      tmp->id, tmp->disc_time, tmp->endp_time,
      (tmp2!=NULL) ? tmp->pred->id : -1);
  }
}
```

Implementation (with adjacency list)

```
int graph_dfs_r(graph_t *g, vertex_t *n, int currTime) {
  edge_t *e;
  vertex_t *t;

  n->color = GREY;
  n->disc_time = ++currTime;
  e = n->head;
  while (e != NULL) {
    t = e->dst;
    switch (tmp->color) {
      case WHITE: printf("%d -> %d : T\n", n->id, t->id);
                  break;
      case GREY : printf("%d -> %d : B\n", n->id, t->id);
                  break;
      case BLACK:
        if (n->disc_time < t->disc_time) {
          printf("%d -> %d : F\n",n->disc_time,t->disc_time);
        } else {
          printf("%d -> %d : C\n", n->id, t->id);
         }
    }
```

Implementation (with adjacency list)

```
      if (tmp->color == WHITE) {
        tmp->pred = n;
        currTime = graph_dfs_r (g, tmp, currTime);
      }
      e = e->next;
    }
    n->color = BLACK;
    n->endp_time = ++currTime;

    return currTime;
  }
```

Complexity

```
DFS (G)
  for each vertex v ∈ V
    v.color = WHITE
    v.dtime = v.endtime = ∞
    v.pred = NULL
  time = 0
  for each vertex v ∈ V
    if (v.color = WHITE)
      DFS_r (G, v)
DFS_r (G, u)
  time++
  u.dtime = time
  u.colro = GRAY
  for each v ∈ Adj(u)
        if (v.color == WHITE)
          v.pred = u
          DFS_r (G, v)
  u.color = BLACK
  time++
  u.endtime = time
```

For each vertex O(1)
For all vertices O(|V|)

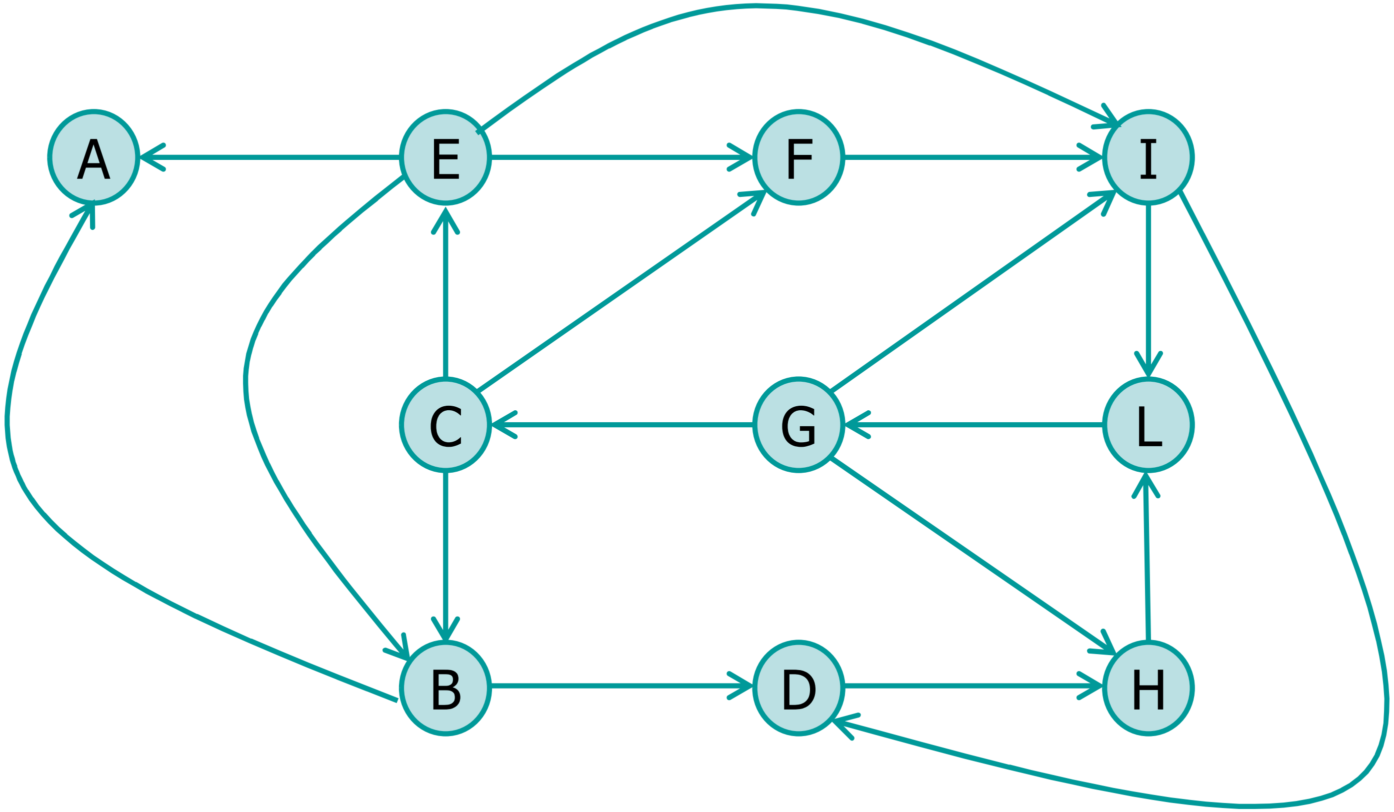
DFS_r is called once for each vertex v → $\Theta(|V|)$

The procedure scans all adjacency lists
Sum of the length of all lists → $\Theta(|E|)$
Cost to manage them → $\Theta(|E|)$

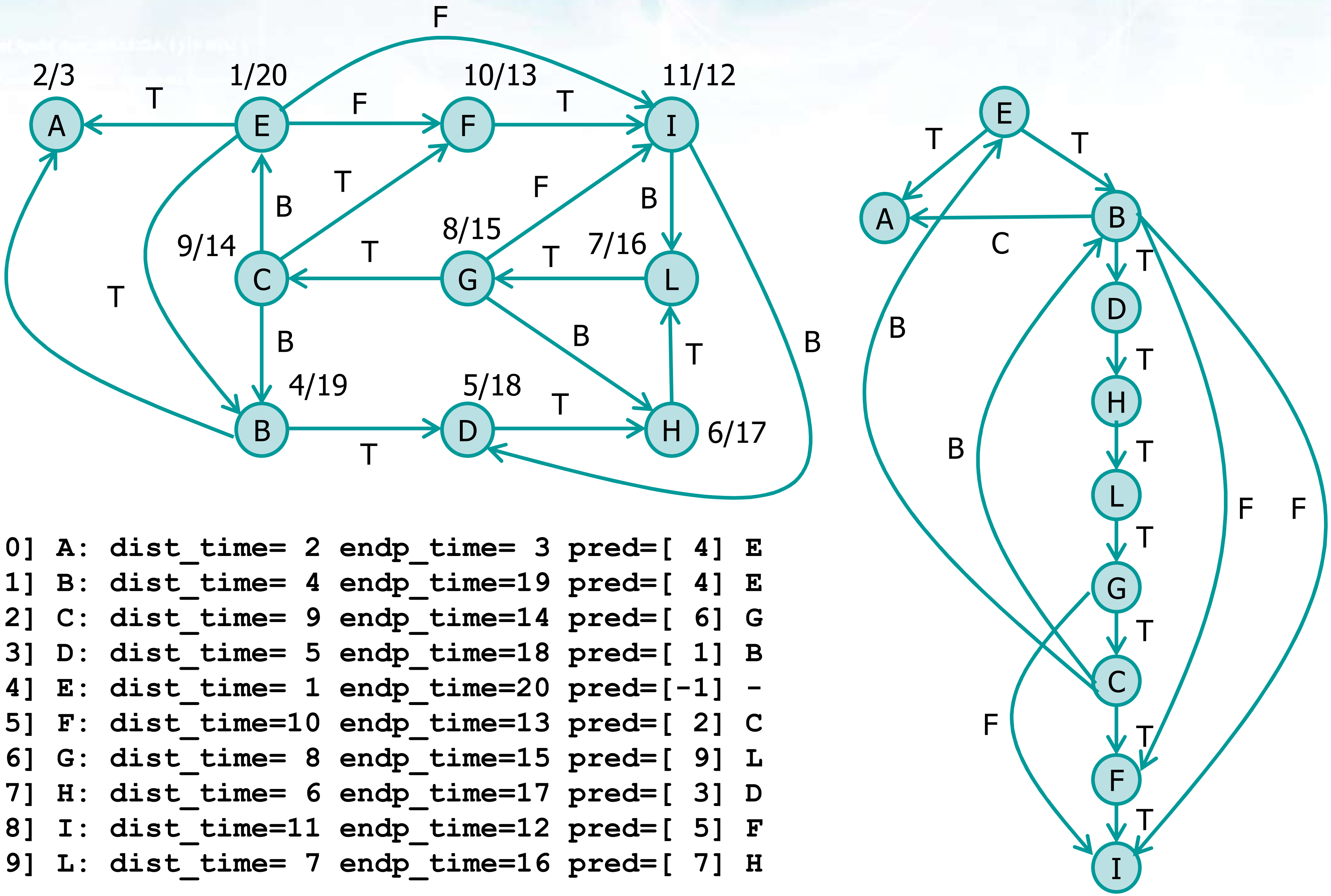
Globally the cost is given by
$T(n) = \Theta(|V|+|E|)$

Exercise

- Given the following graph, visit it Depth-First starting from vertex E
  - Label all edges
  - Report the resulting DFS tree



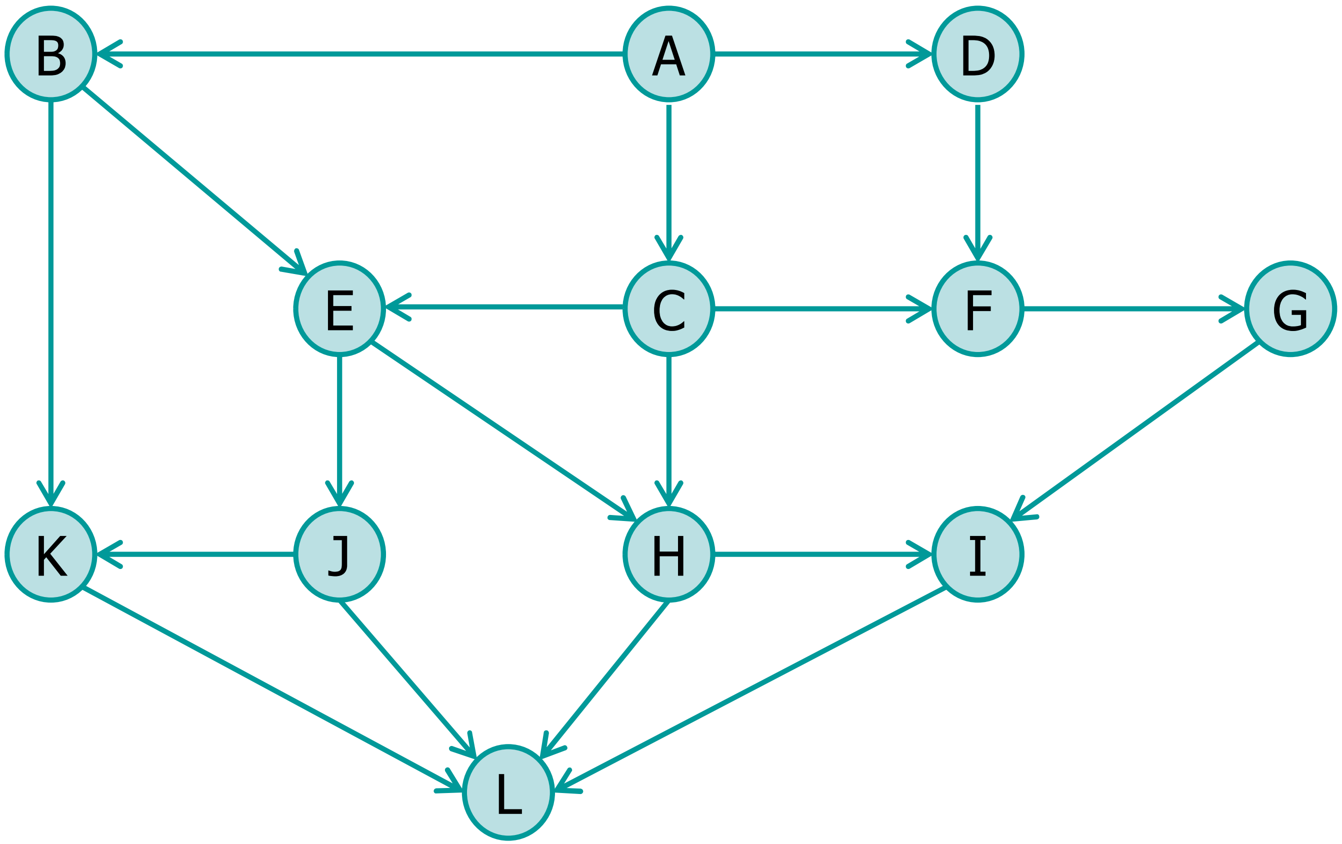
Solution



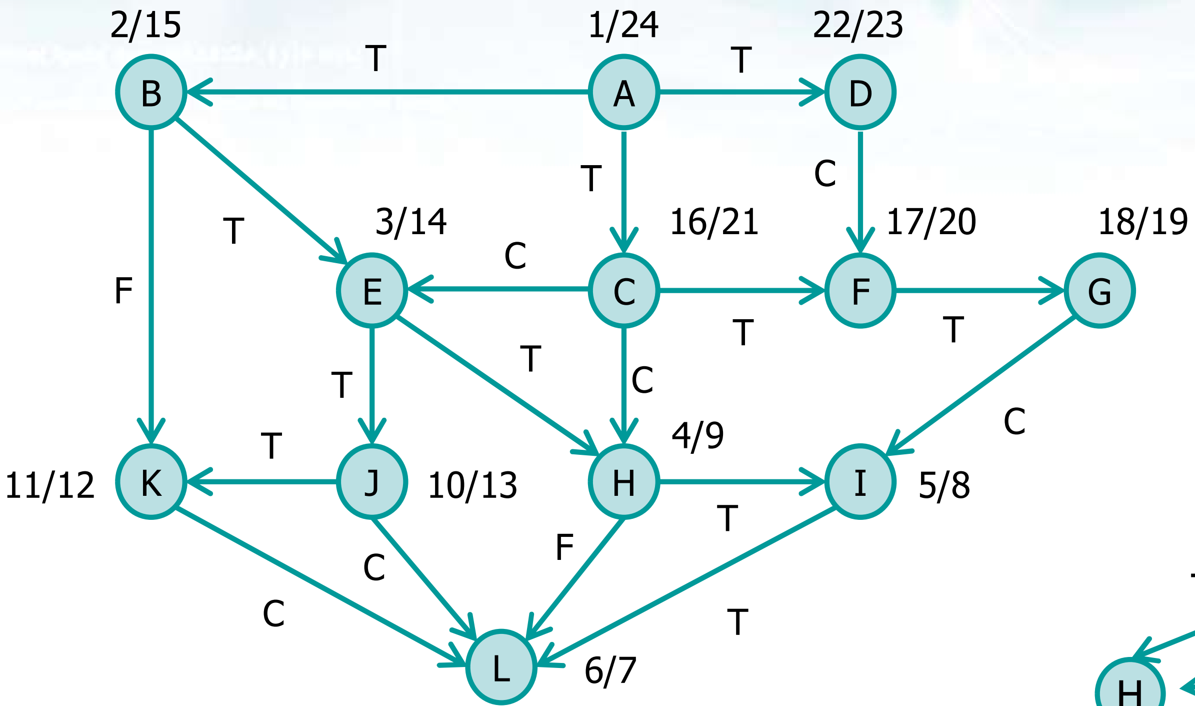

```
[ 0] A: dist_time= 2 endp_time= 3 pred=[ 4] E
[ 1] B: dist_time= 4 endp_time=19 pred=[ 4] E
[ 2] C: dist_time= 9 endp_time=14 pred=[ 6] G
[ 3] D: dist_time= 5 endp_time=18 pred=[ 1] B
[ 4] E: dist_time= 1 endp_time=20 pred=[-1] -
[ 5] F: dist_time=10 endp_time=13 pred=[ 2] C
[ 6] G: dist_time= 8 endp_time=15 pred=[ 9] L
[ 7] H: dist_time= 6 endp_time=17 pred=[ 3] D
[ 8] I: dist_time=11 endp_time=12 pred=[ 5] F
[ 9] L: dist_time= 7 endp_time=16 pred=[ 7] H
```

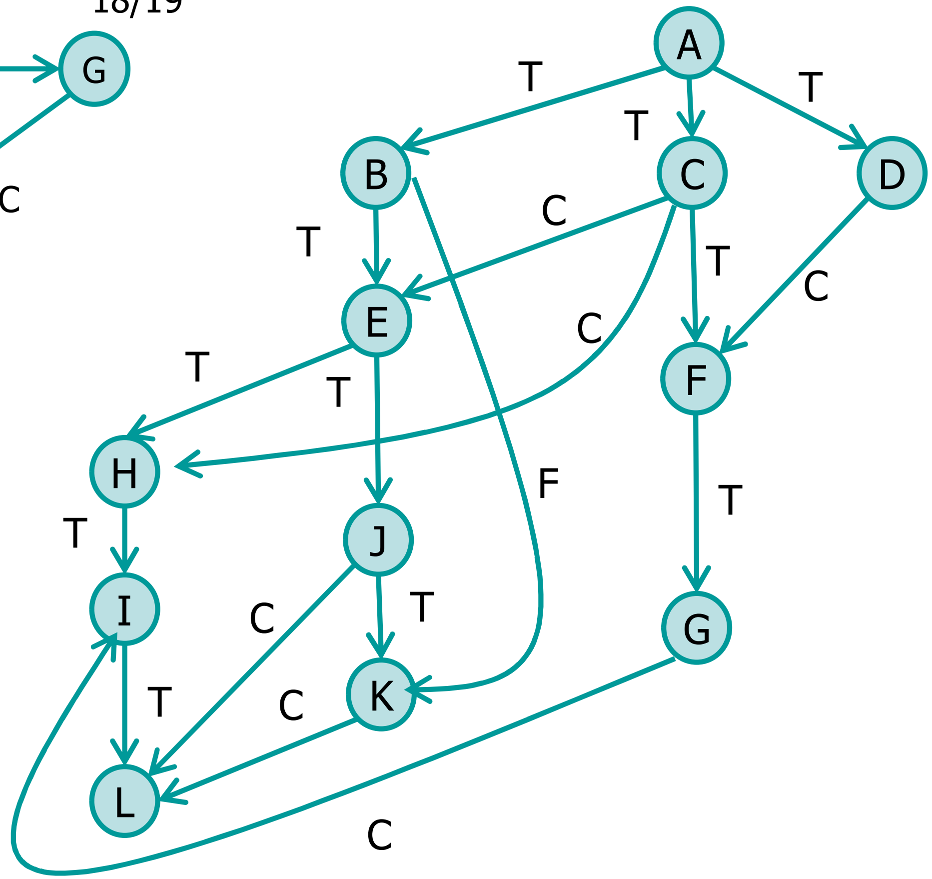
Exercise

- Given the following graph, visit it Depth-First starting from vertex A
  - Label all edges
  - Report the resulting DFS tree



Solution






```
[ 0] A: dist_time= 1 endp_time=24 pred=[-1] -
[ 1] B: dist_time= 2 endp_time=15 pred=[ 0] A
[ 2] C: dist_time=16 endp_time=21 pred=[ 0] A
[ 3] D: dist_time=22 endp_time=23 pred=[ 0] A
[ 4] E: dist_time= 3 endp_time=14 pred=[ 1] B
[ 5] F: dist_time=17 endp_time=20 pred=[ 2] C
[ 6] G: dist_time=18 endp_time=19 pred=[ 5] F
[ 7] H: dist_time= 4 endp_time= 9 pred=[ 4] E
[ 8] I: dist_time= 5 endp_time= 8 pred=[ 7] H
[ 9] J: dist_time=10 endp_time=13 pred=[ 4] E
[10] K: dist_time=11 endp_time=12 pred=[ 9] J
[11] L: dist_time= 6 endp_time= 7 pred=[ 8] I
```