Graph Theory: Search

Anton Gerdelan <gerdela@scss.tcd.ie>

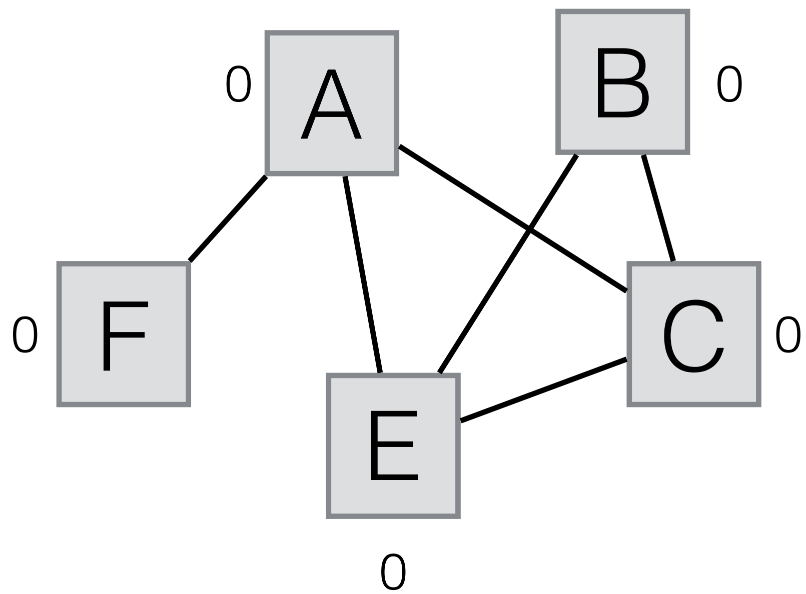
Graph Traversal

- Visiting nodes in a graph (graph traversal)
  - trickier than tree because cycles -> infinite loop
- Traversing a graph similar to finding a spanning tree
  - add flag to each vertex to show if it has been visited yet

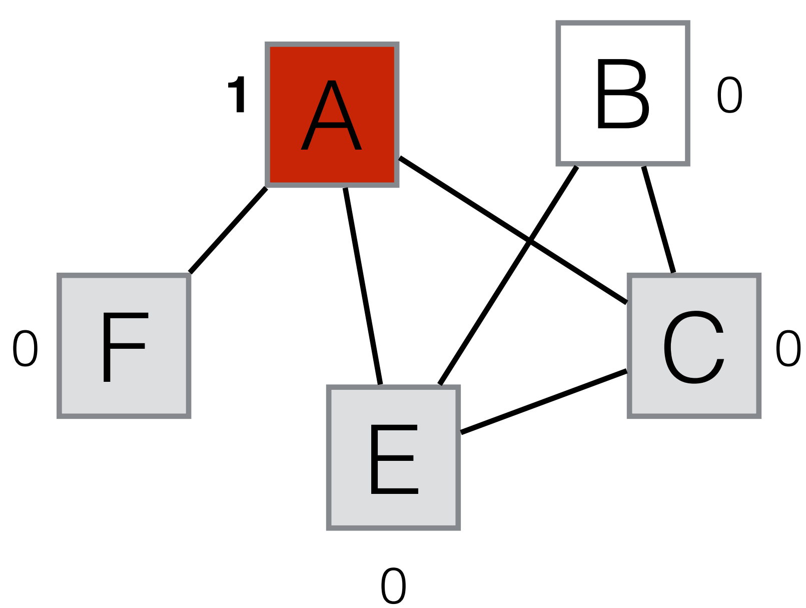
Depth First Search (DFS)

- mark all vertices as **not visited**
- for all vertices in graph, if vertex **v** has not been visited then use **recursive** function `DFS( v )`
- `DFS( v )`
  - Mark vertex **v** as **visited**
  - for all vertices connected to **v**
    - if **v** has not been visited
      - `DFS( x )`

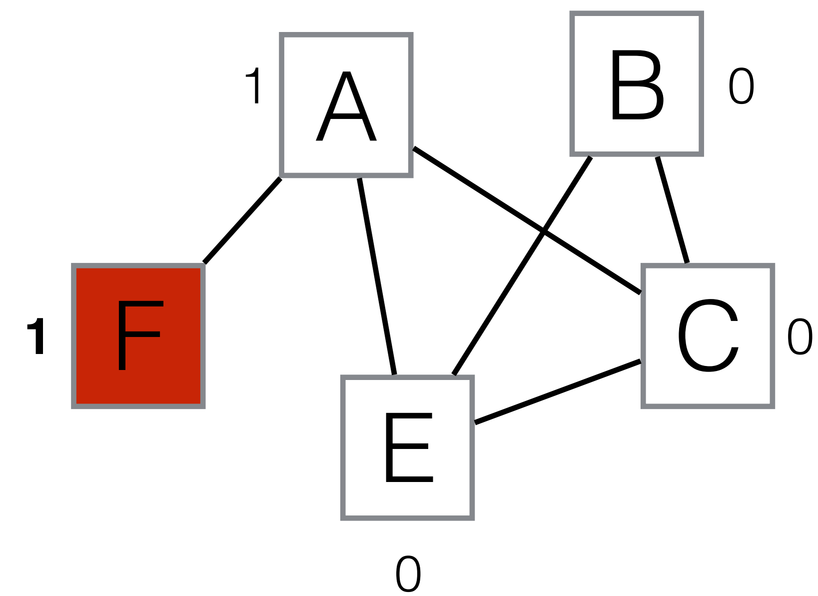
1) mark all vertices as 0,
i choose vertex A



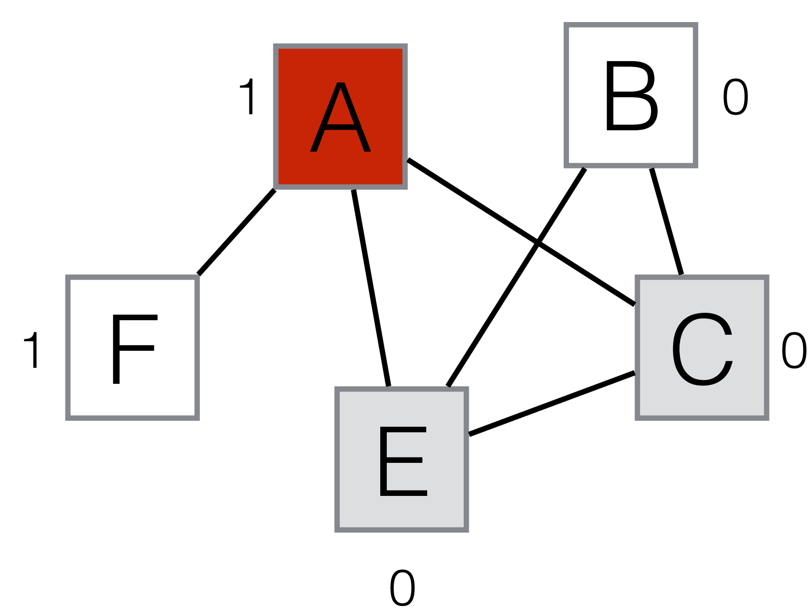
2) `DFS( A )` - mark as 1,
choose from {F, E, C}



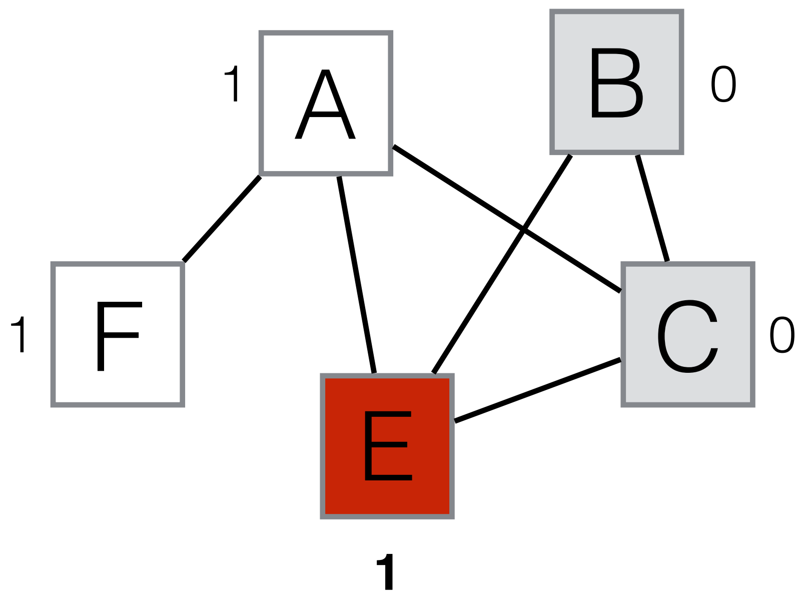
3) F not visited - `DFS( F )` - mark as 1
no unvisited connections - **`return`**



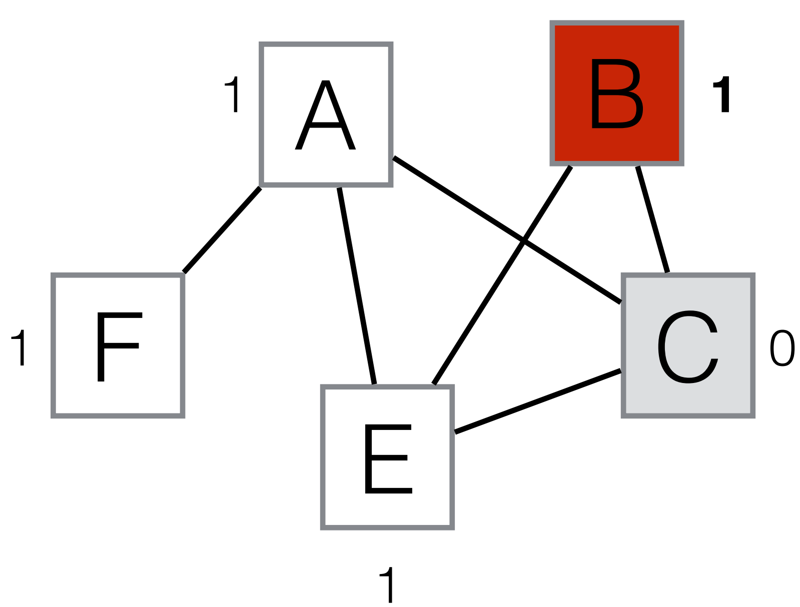
4) back at A
choose from {E, C}



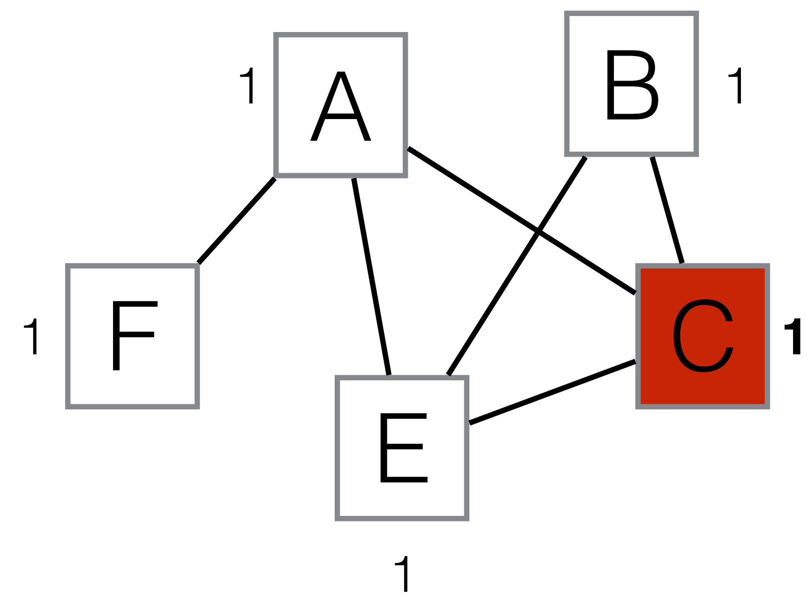
5) `DFS( E )` - mark as 1, choose from {B, C}



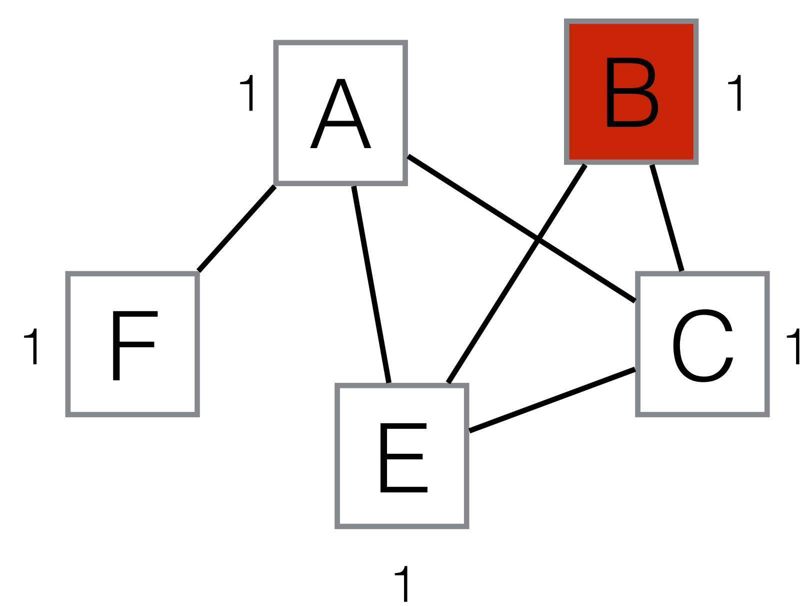
6) `DFS( B )` - mark as 1, choose C



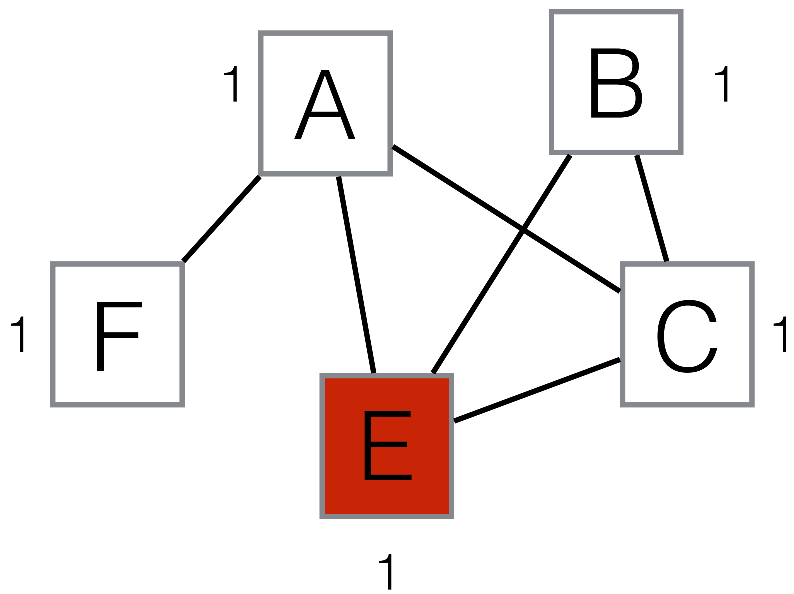
7) `DFS( C )` - mark as 1 **`return`**



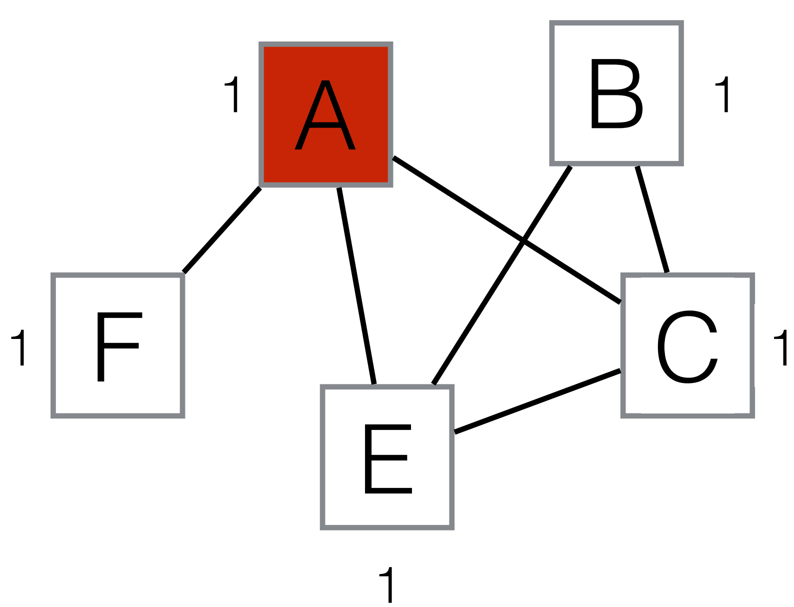
8) back at B - all visited - **`return`**



9) back at E
- all visited - **`return`**



10) back at A
- all visited - **`return`**
recursion done

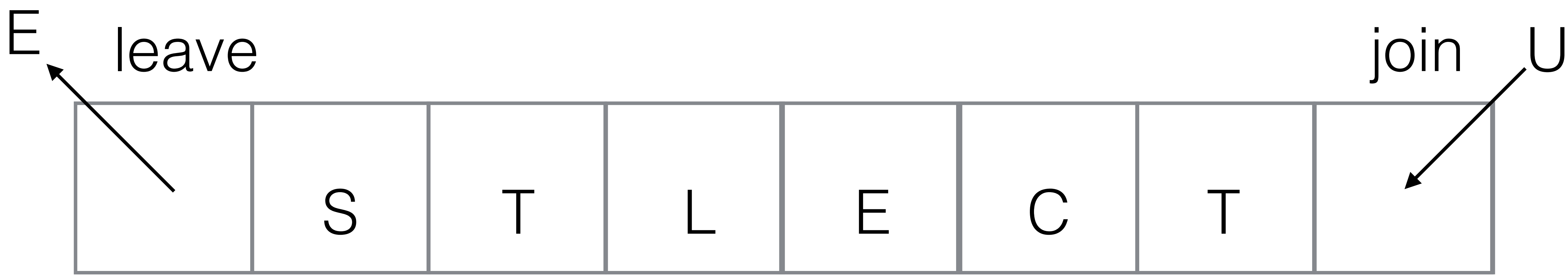


traversal sequence:
A,F,E,B,C

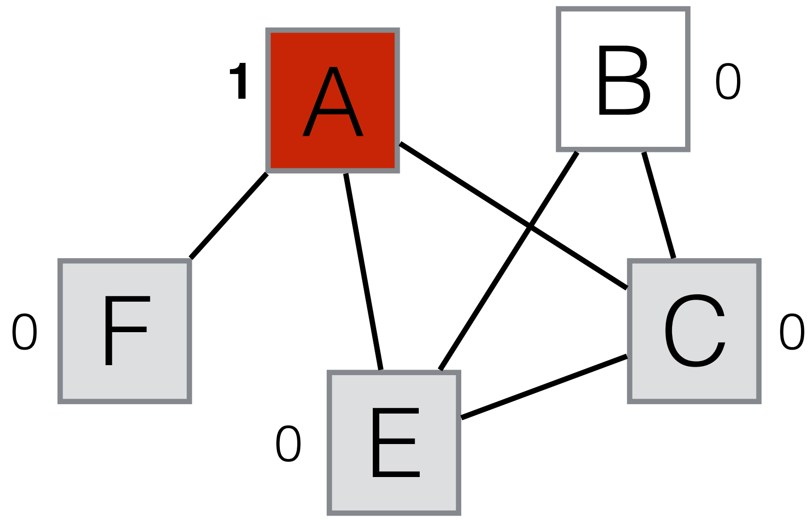
Breadth First Search (BFS)

- mark all vertices as **not visited**
- create an empty **queue** *Q* of vertices
- for all vertices in graph, if vertex **v** has not been visited then use **iterative** function
  `BFS( v ) // "hello! i am not recursive"`
- `BFS( v )`
  - Mark vertex **v** as **visited**
  - add **v** to **Q**
  - **while** Q is not empty
    - remove front vertex, **x**, from Q
    - for all vertices, **i**, adjacent to **x**,
      - if vertex **i** has not been visited then **mark as visited** and add it to **Q**

Queue (ADT)

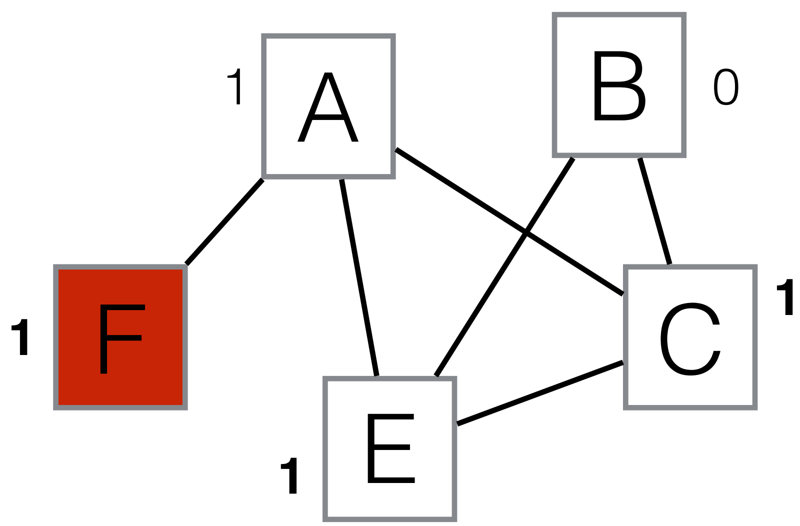


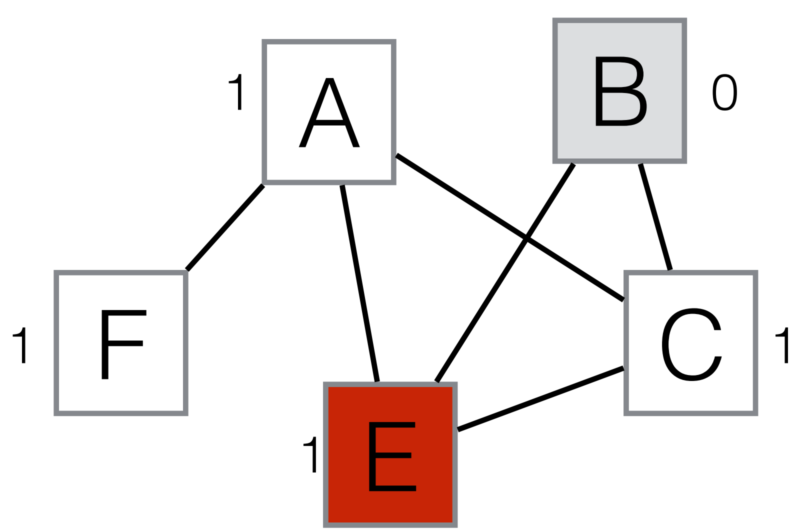
- `join()  leave()  front()  is_empty()`
- queue is first-in first-out (**FIFO**) data type
    - (stack is **LIFO**)
    - *NB* queue analogy is people joining a line. stack is a mechanical dish stacker (like in a buffet) `push()` `pop()` etc.
- using 'circular' array would be okay - keep wrap-around start and end indices - tricky
- a linked list might be easier to manage

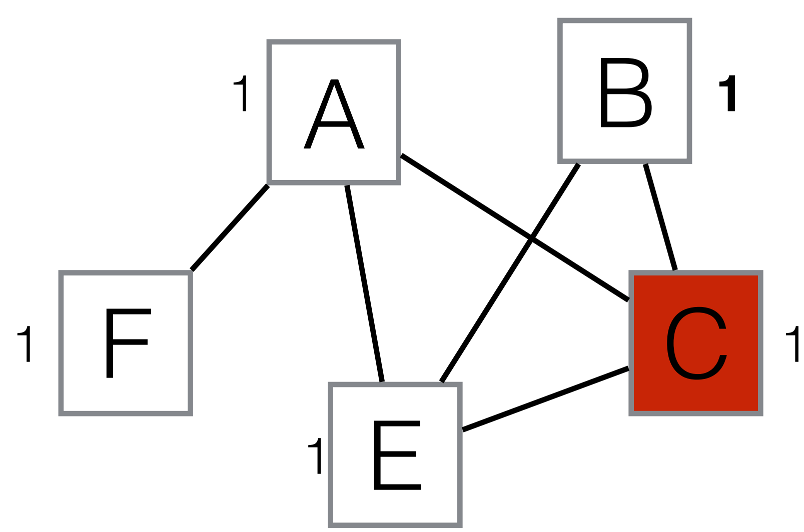


X	Q	
	-	starts empty
	A	**visit** A, add to Q
A	-	leave Q
A	F,E,C	**visit** adjacents, add to Q
F	E,C	leave Q
F	E, C	no unvisited adjacent
E	C	leave Q
E	C,B	**visit** adjacent, add to Q
C	B	leave Q
		all nodes visited - **halt**

(if vertices in graph still unvisited - repeat for each)



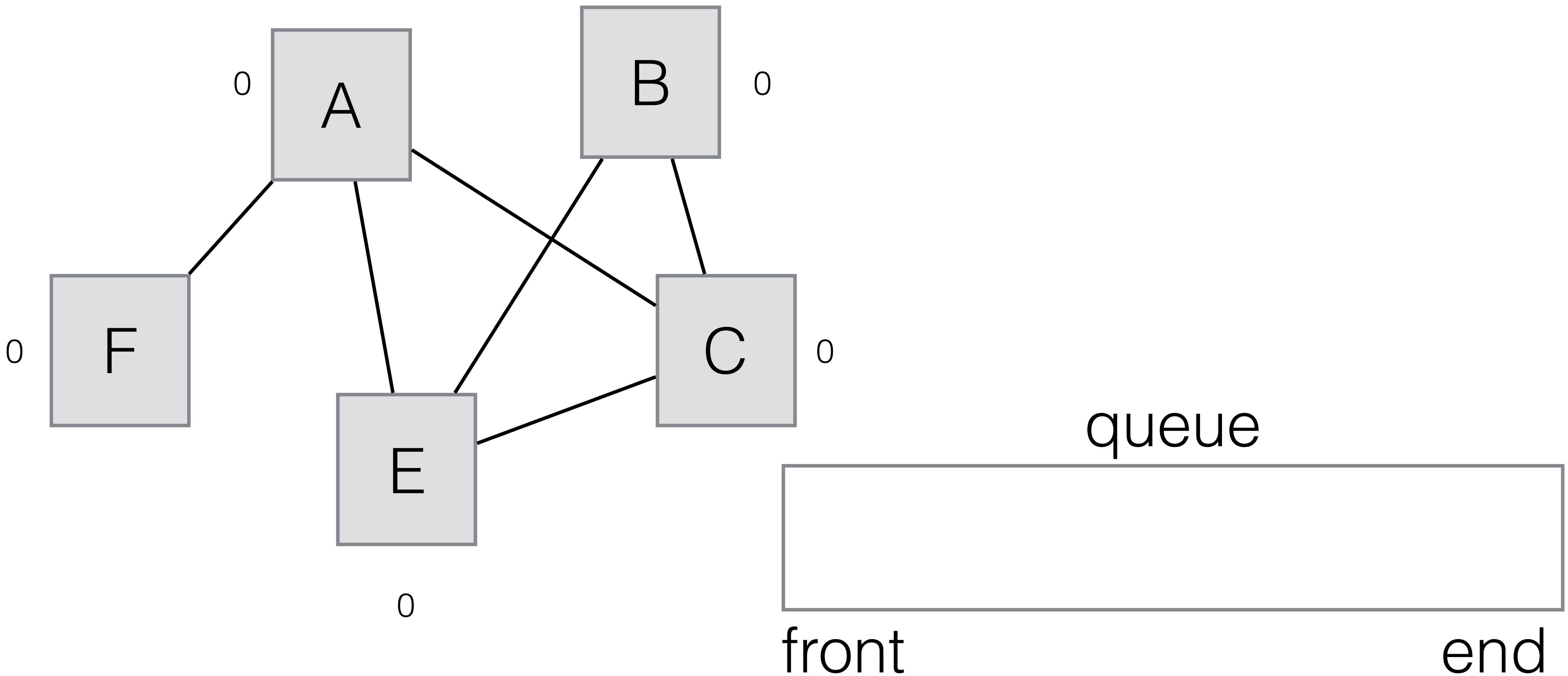




traversal sequence:
A, F, E, C, B

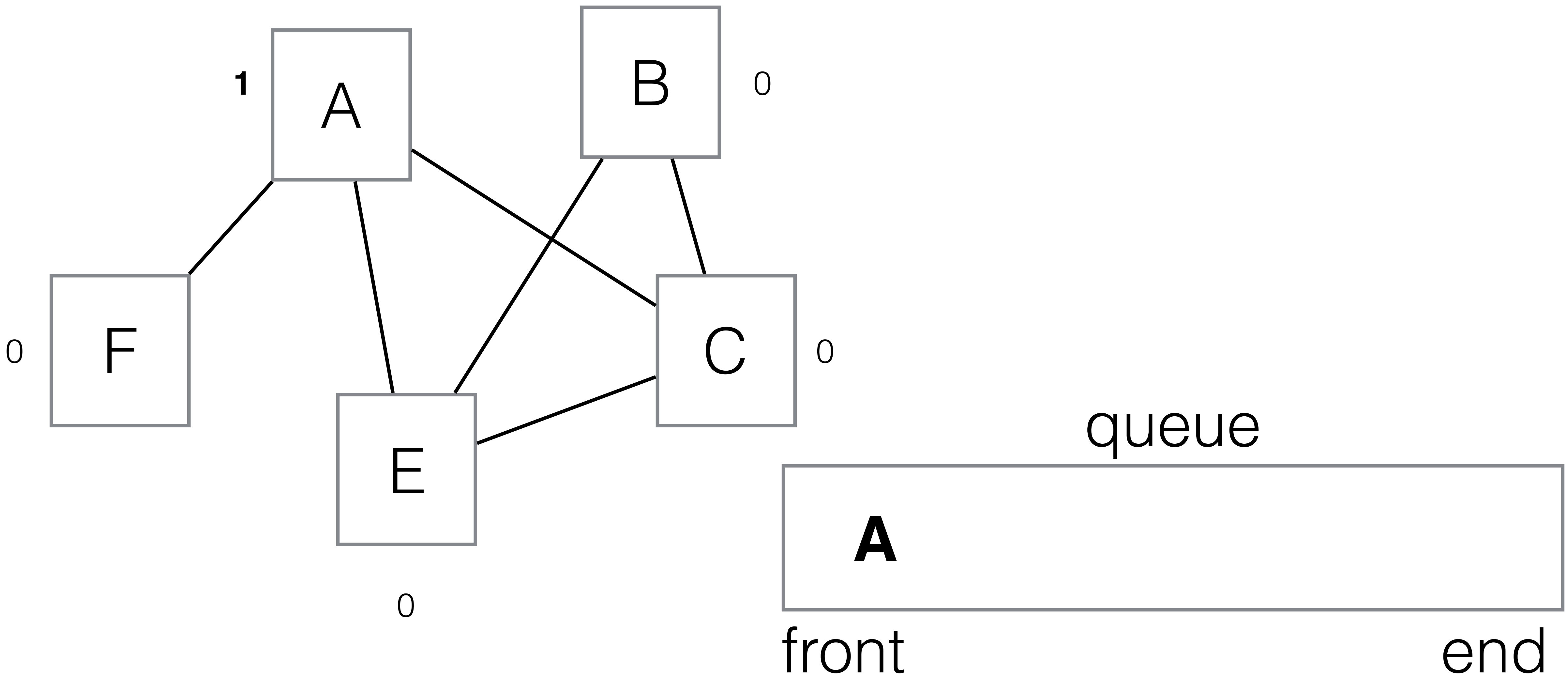
BFS Recap

- BFS is very commonly used to solve lots of problems
  - web-crawling / Internet / Wikipedia
  - social network - contacts: “people you may know”
- BFS works on directed and undirected graphs
- Requires a queue
- Is not recursive



let's start at A

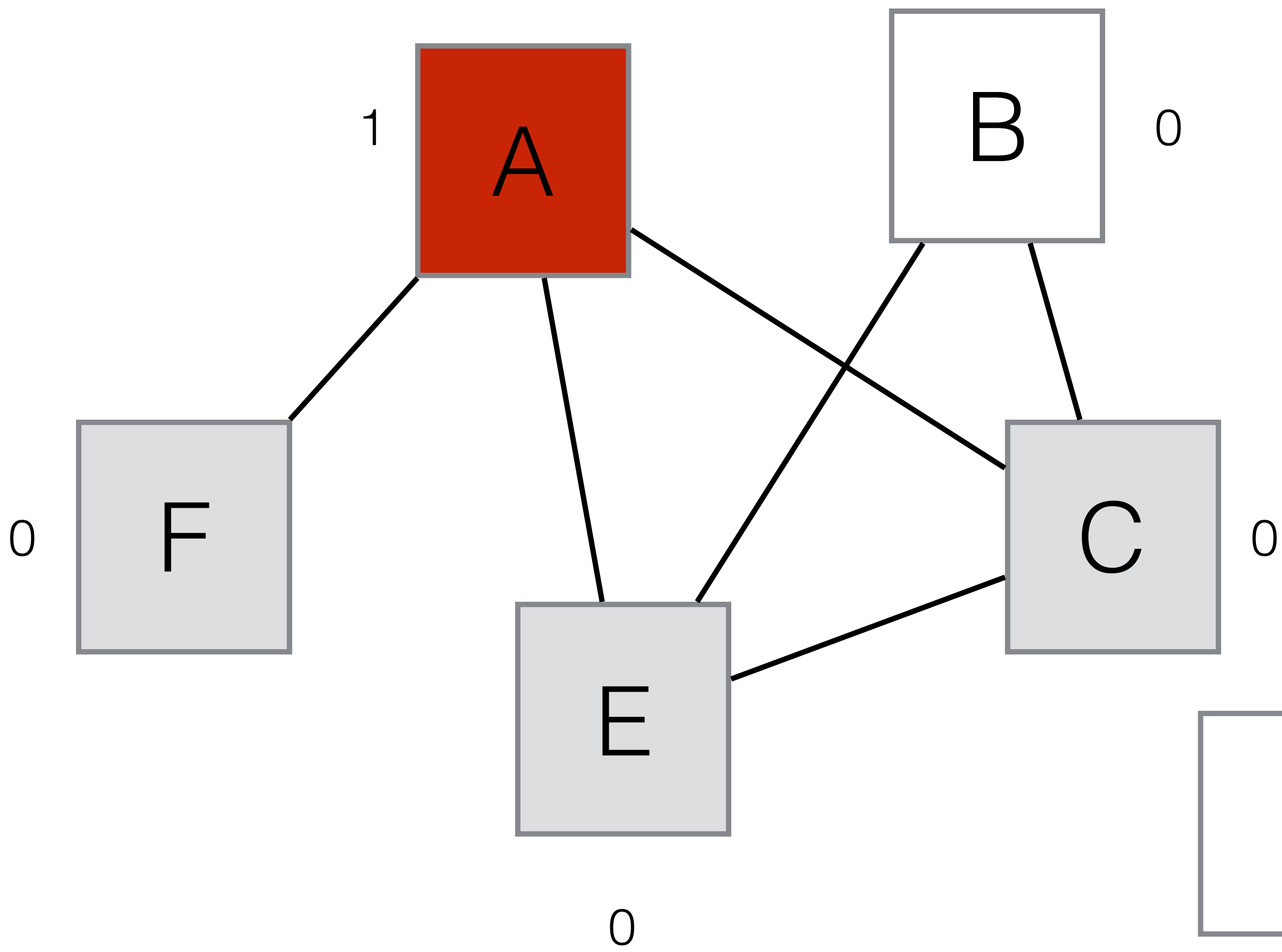
sequence visited

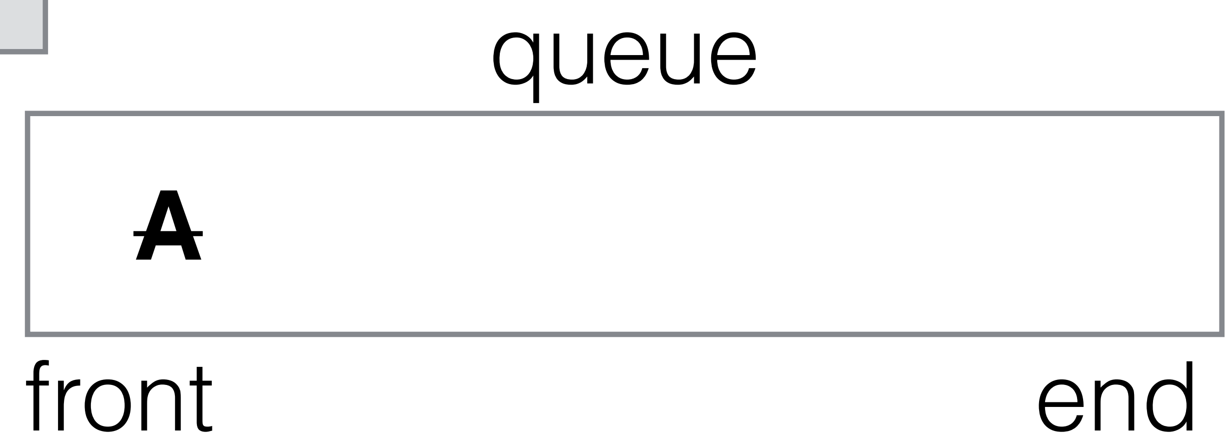


- mark ‘A’ visited
- enqueue ‘A’

sequence visited

A

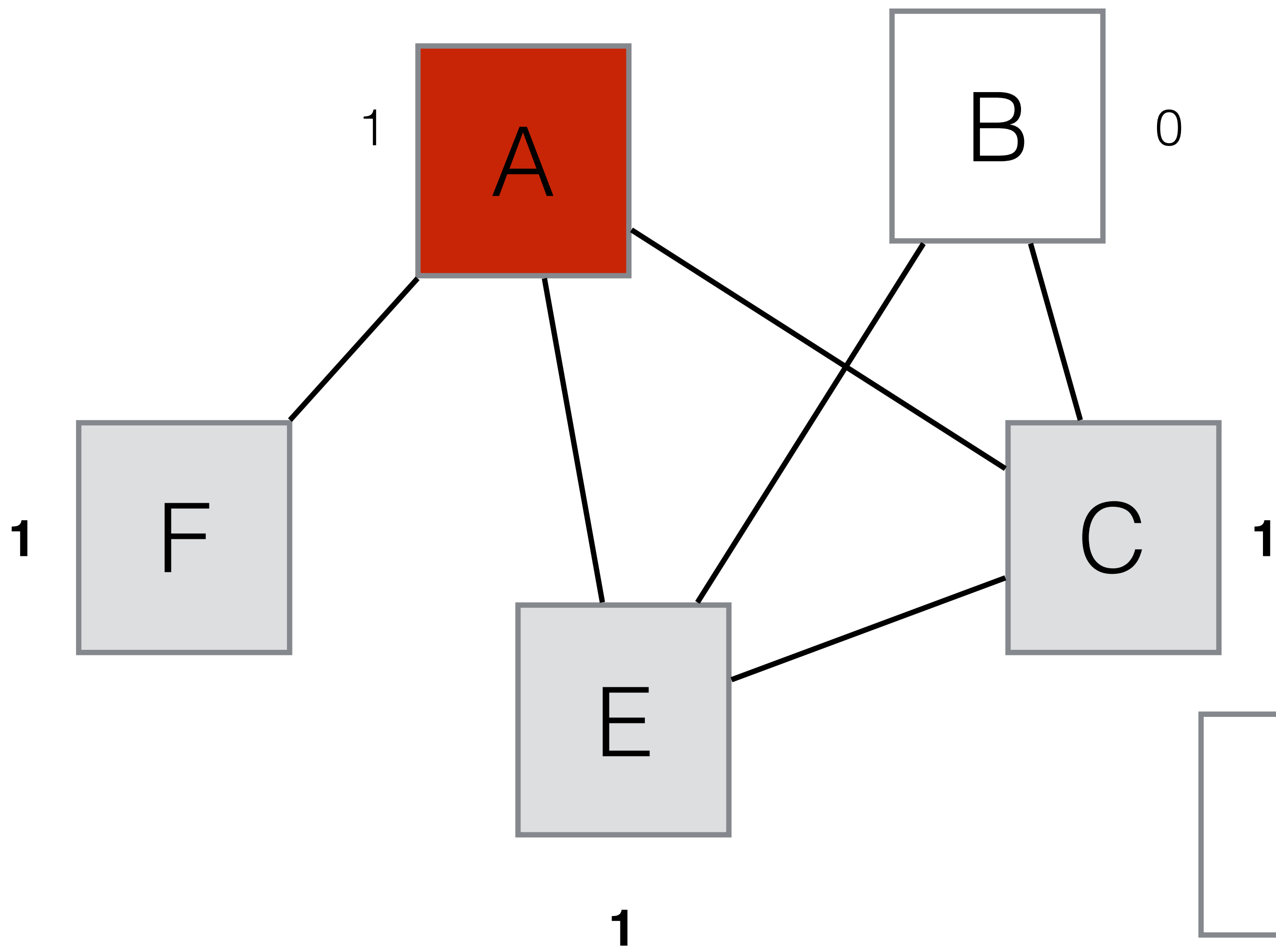


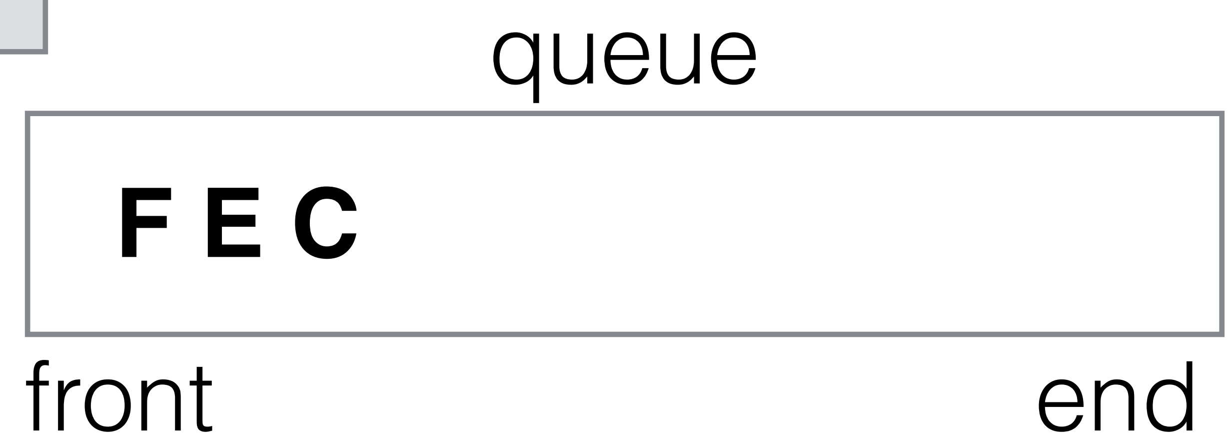


- queue is not empty so:
  - dequeue ‘A’
  - ‘A’ is current vertex
  - unvisited neighbours (the **frontier**) is in grey

sequence visited

A

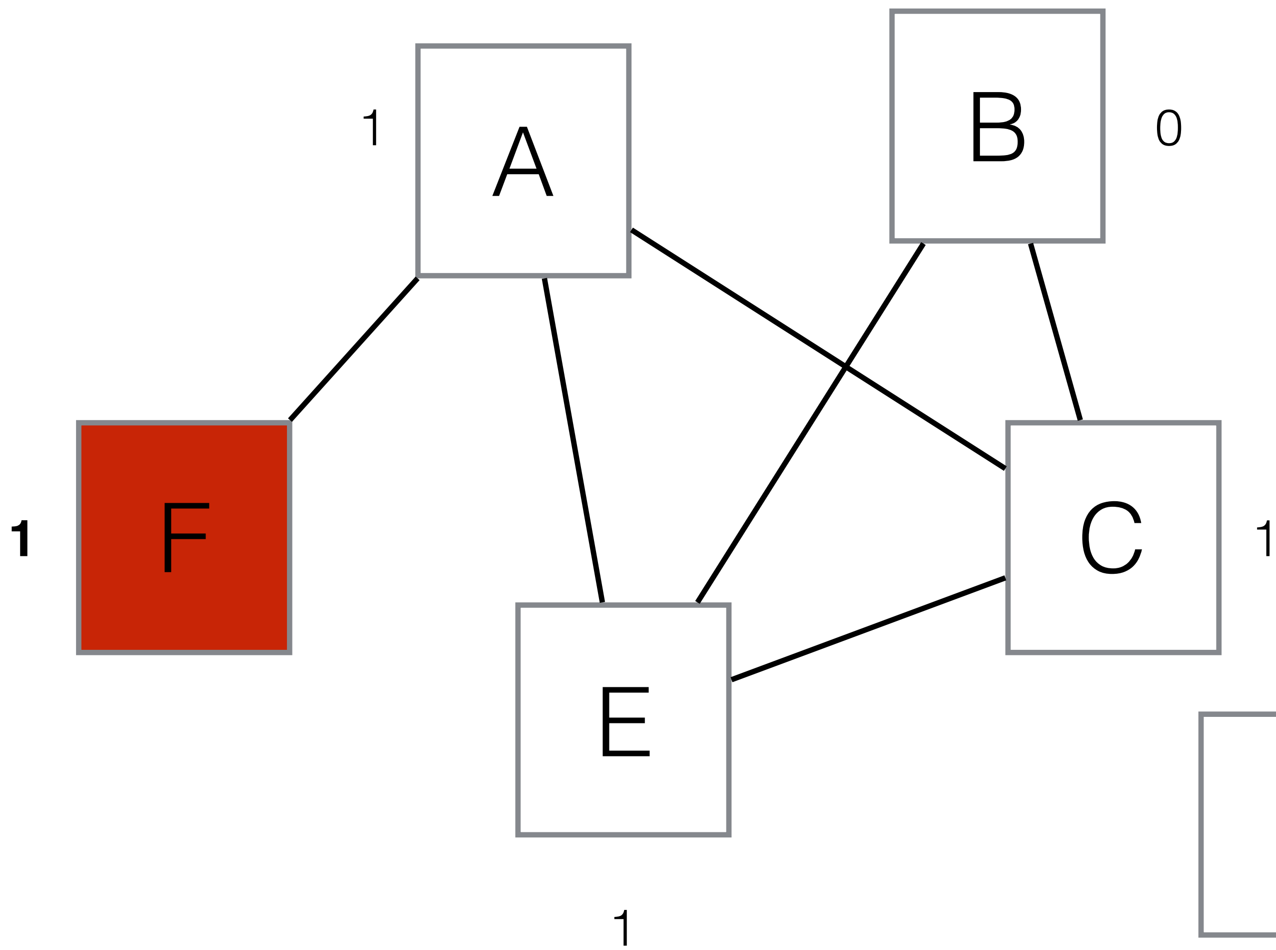




- mark each of these visited
- and add each to queue

sequence visited

A **F E C**



queue

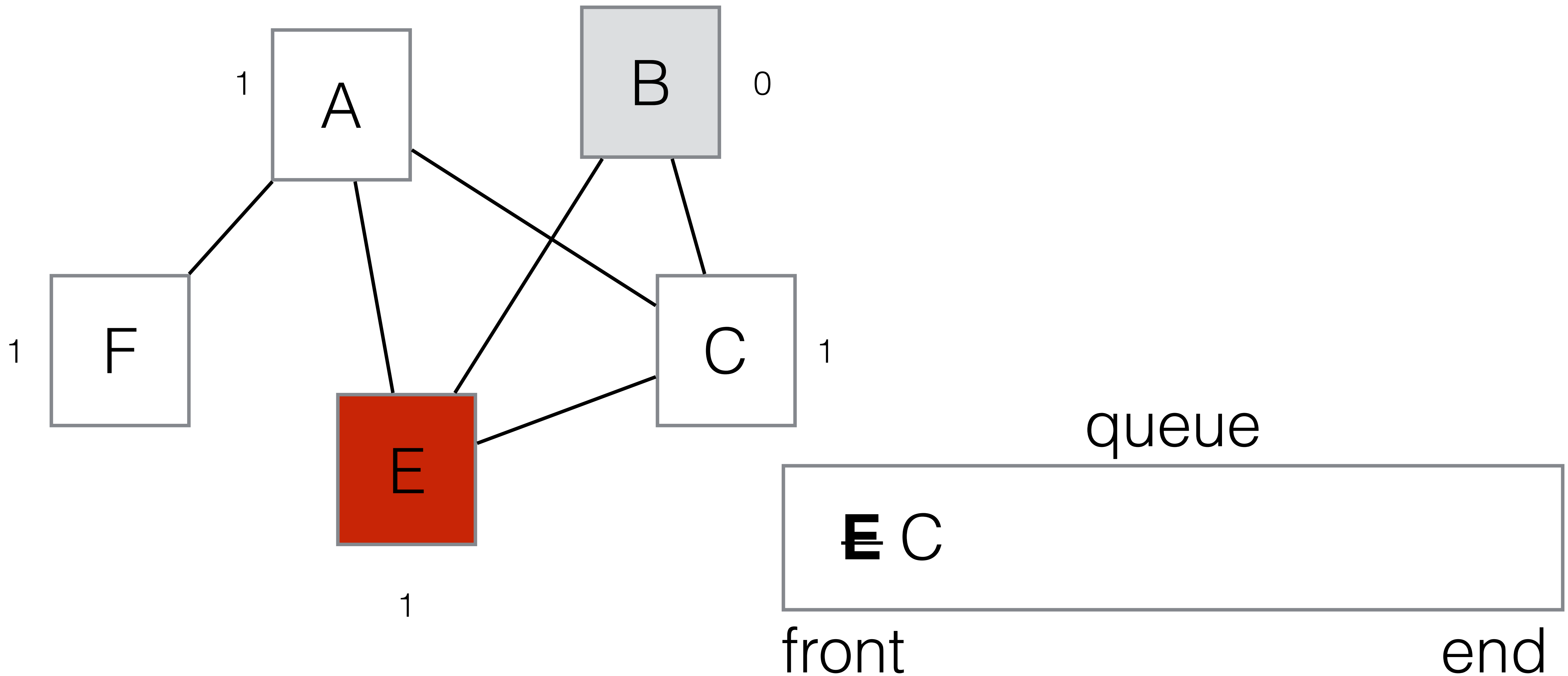
~~F~~ E C

front end

- queue is not empty so:
  - dequeue ‘F’
  - ‘F’ is current vertex
  - no unvisited neighbours

sequence visited

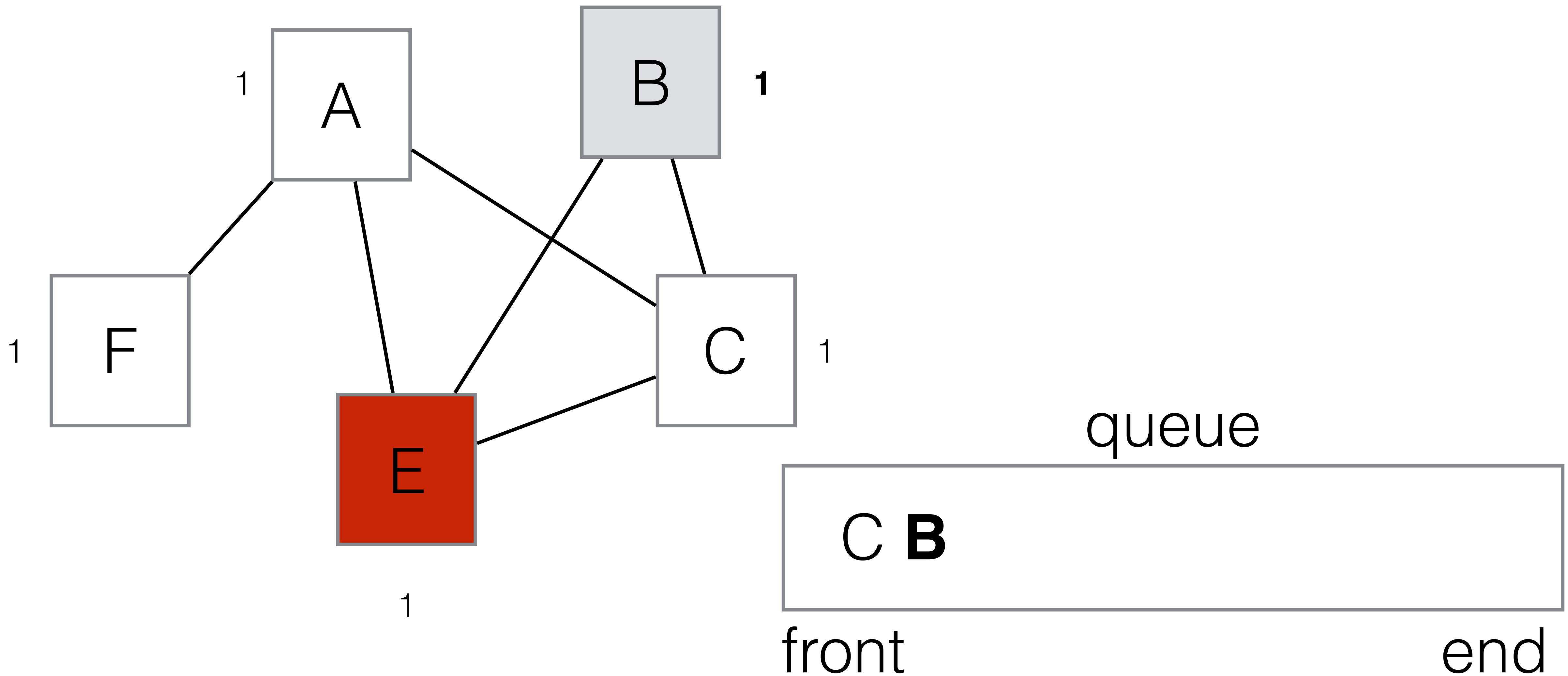
A F E C



- queue is not empty so:
  - dequeue ‘E’
  - ‘E’ is current vertex
  - unvisited neighbour in grey

sequence visited

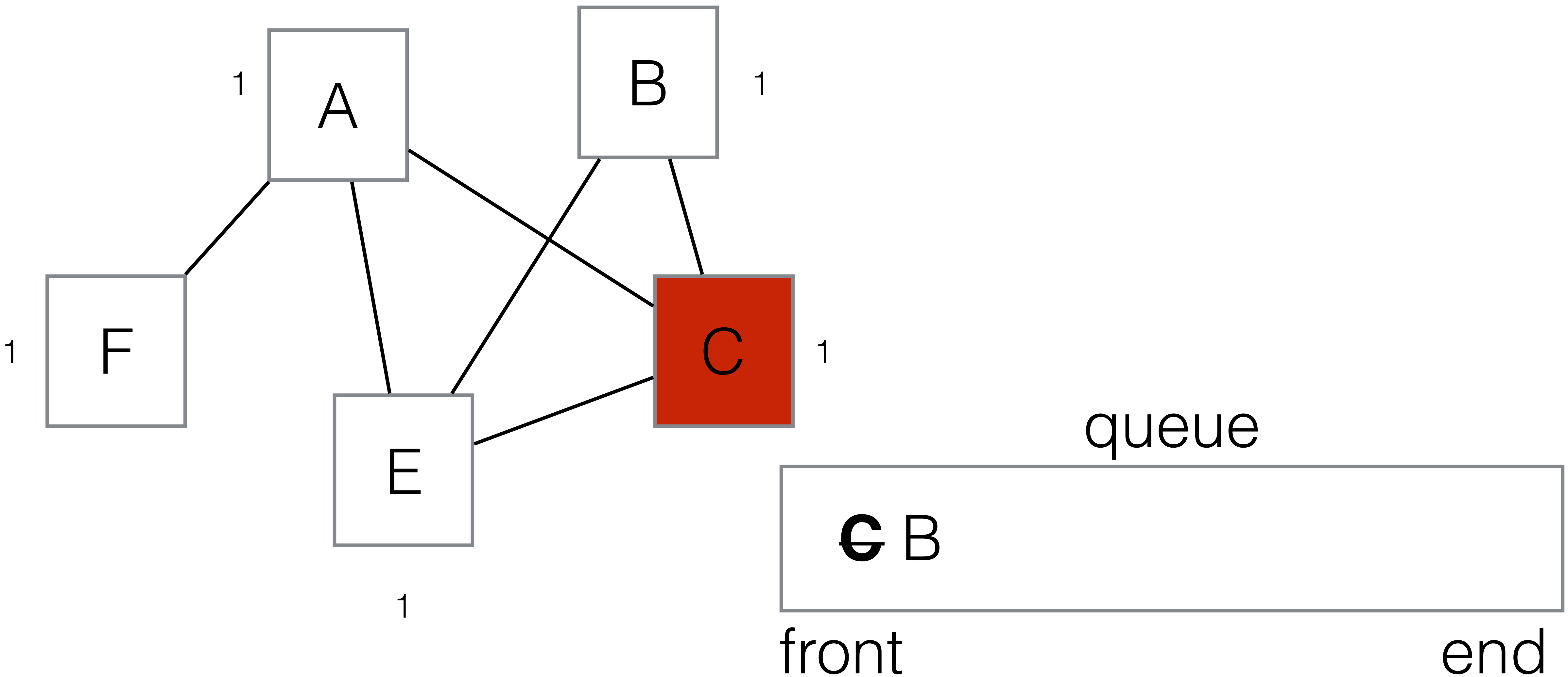
A F E C



- mark ‘B’ visited
- enqueue B

sequence visited

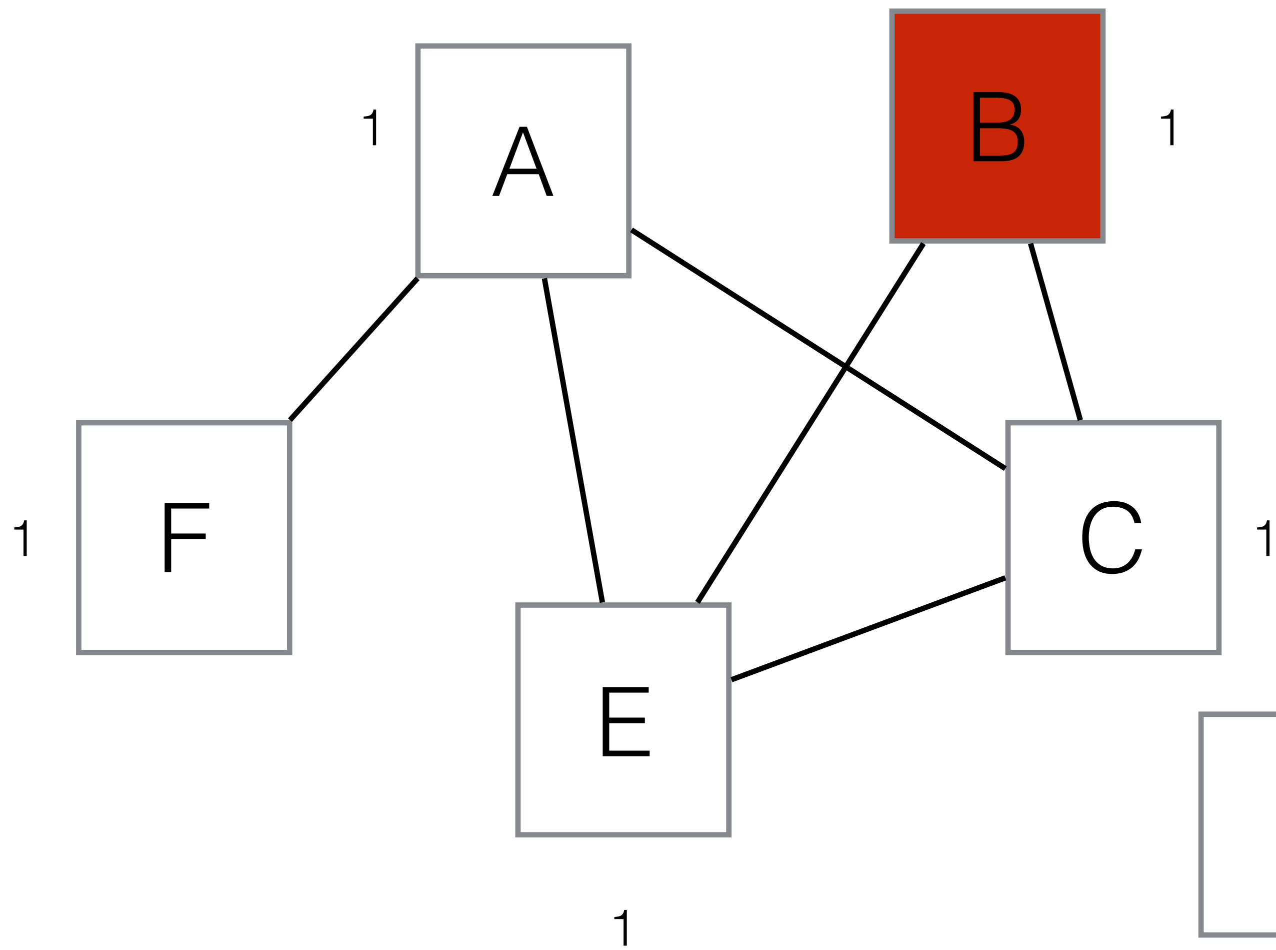
A F E C B

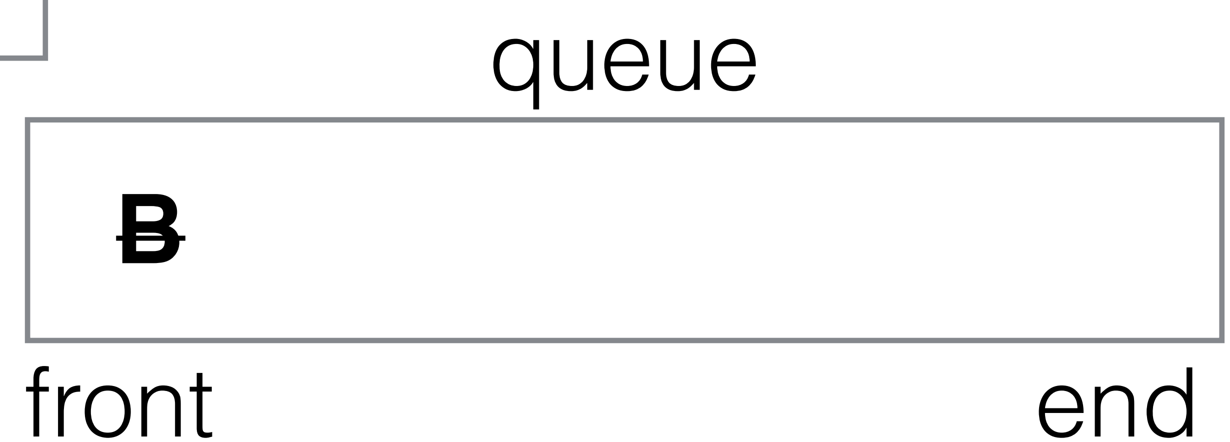


- queue is not empty so:
  - dequeue ‘C’
  - ‘C’ is current vertex
  - no unvisited vertices

sequence visited

A F E C B





- queue is not empty so:
  - dequeue ‘B’
  - ‘B’ is current vertex
  - no unvisited vertices
- queue is empty
  - halt

sequence visited

A F E C B

BFS vs DFS

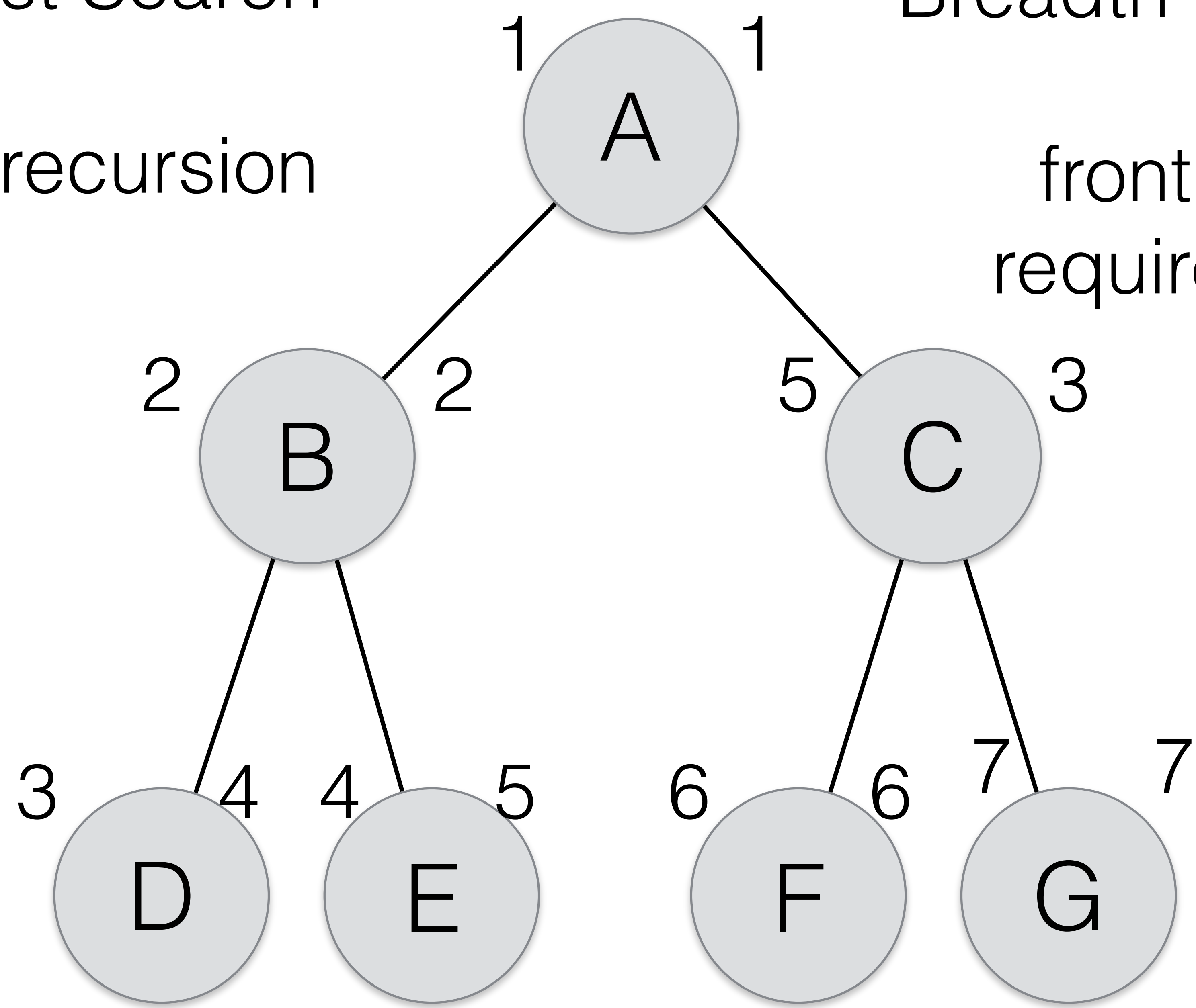
- easiest to compare difference on a tree
  - Anton: draw helpful diagram here to compare them
- visit sequence differs
- are you more likely to find your results earlier in a BFS or DFS sequence?
- implementation may affect sequence - e.g. order that all adjacent nodes are visited in BFS
- recursive function might need rewrite for large graphs
  - **Q. why?**

Depth-First Search

[branch] recursion

Breadth-First Search

frontiers first
requires queue



A B D E C F G

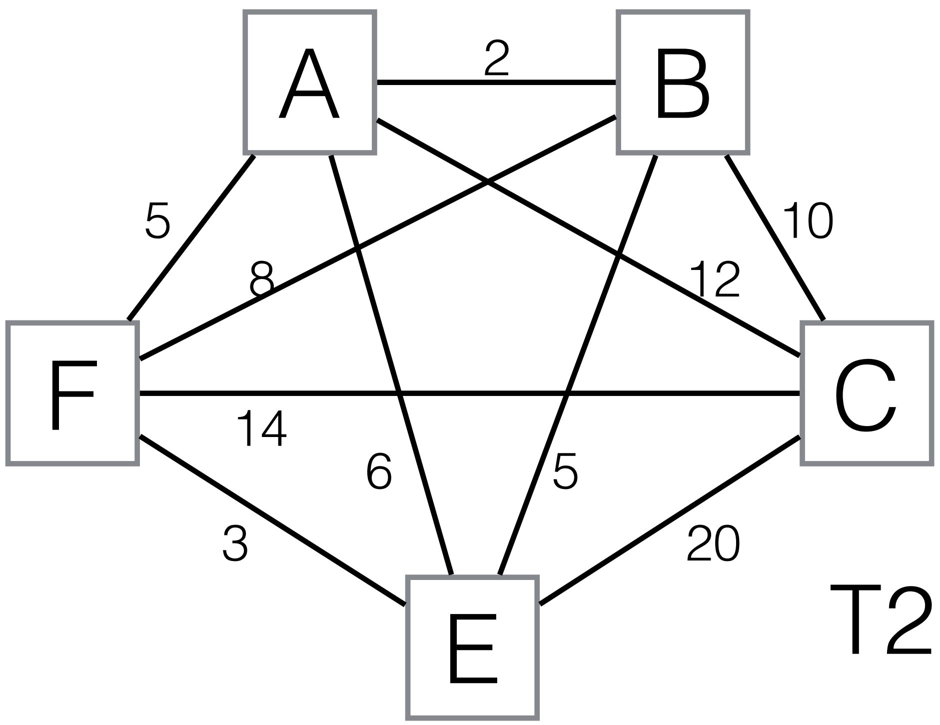
A B C D E F G

Spanning Trees

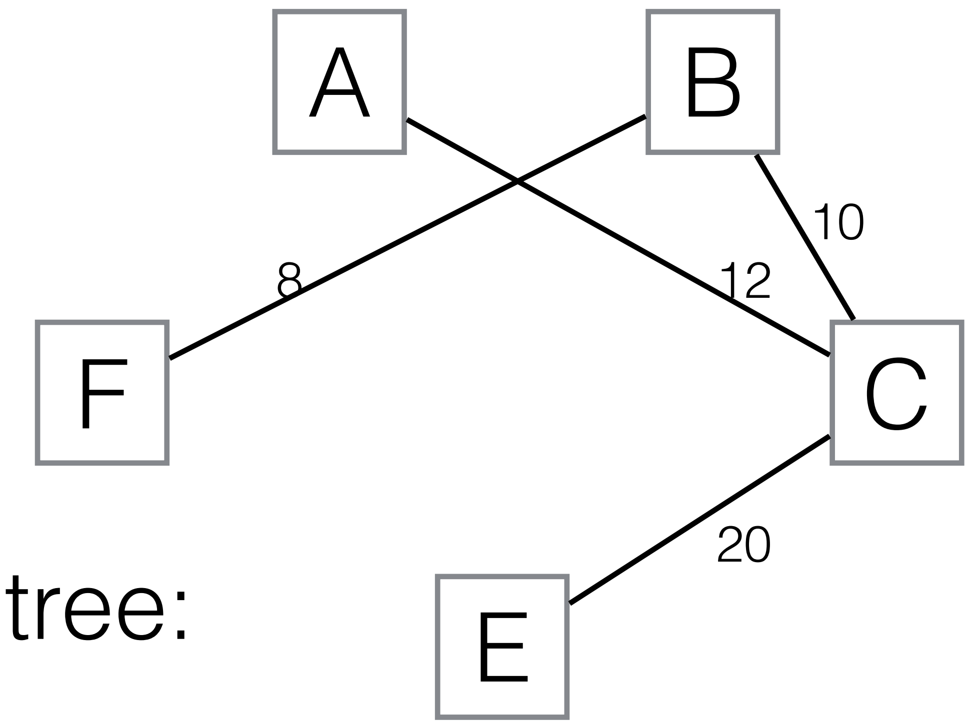
- A spanning tree of graph **G** consists of
  - is a sub-graph - **simplifies** the graph for traversal
  - all vertices in **G**
  - only <u>some</u> of the edges
  - should be representable as a tree
- Edges are chosen so that new graph is still **connected** but is **<u>acyclic</u>**
- A graph can contain many spanning trees
- *Q. can a graph that is not <u>connected</u> contain a ST?*

Spanning Trees

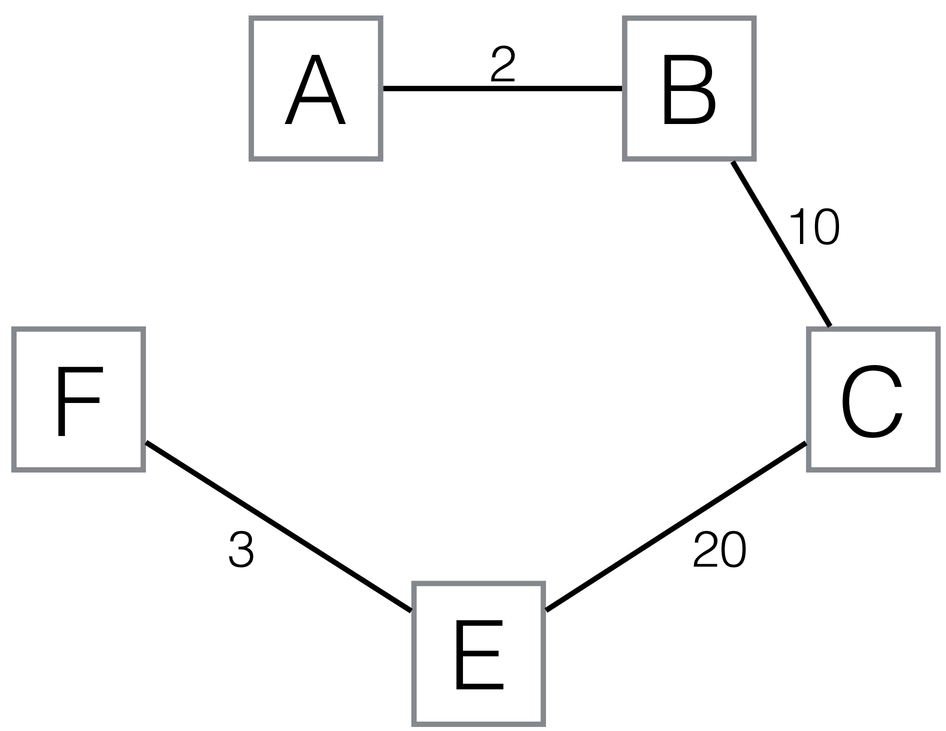
If graph G is



T1 is a spanning tree:



T2 is a spanning tree:



total cost of T1
= 8 + 12 + 10 + 20
= 50

cost of T2
= 35

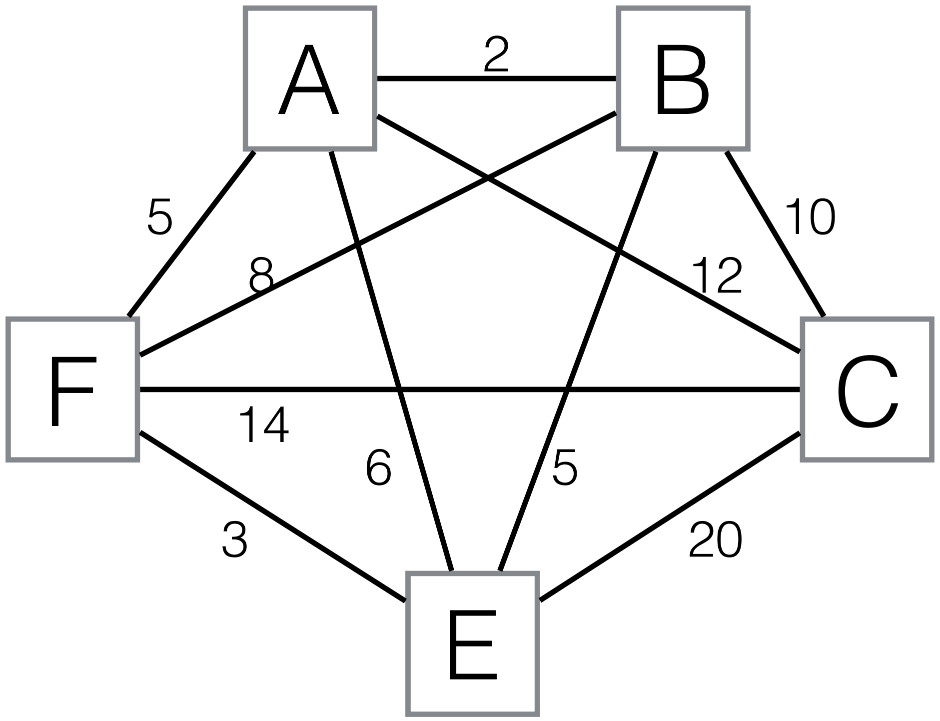
Minimum Cost Spanning Tree (MCST)

- A spanning tree with the lowest **length** is a MCST
- Different algorithms for finding a MCST
  - Kruskal's Algorithm
  - Prim's Algorithm
  - Boruvka's Algorithm
  - mixtures

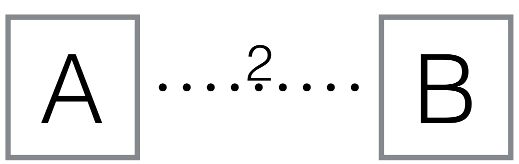
Kruskal's Algorithm

- Joseph Kruskal, 1956
- This is one method for finding the MCST
- **Greedy** algorithm paradigm (short-sighted best choices)
  - solve in stages - make **optimal local choice** for each stage
  - hope this results close to a global optimum
- Start with empty spanning tree
- Add <u>next lowest</u> weighted edge to spanning tree, as long as <u>no cycles</u> are formed
- Repeat previous step until all edges have been considered

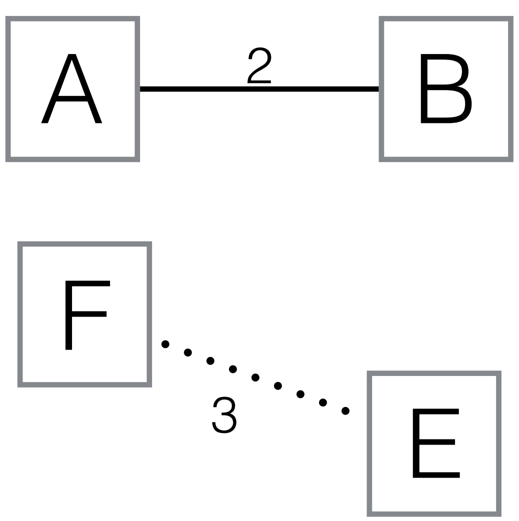
If graph G is



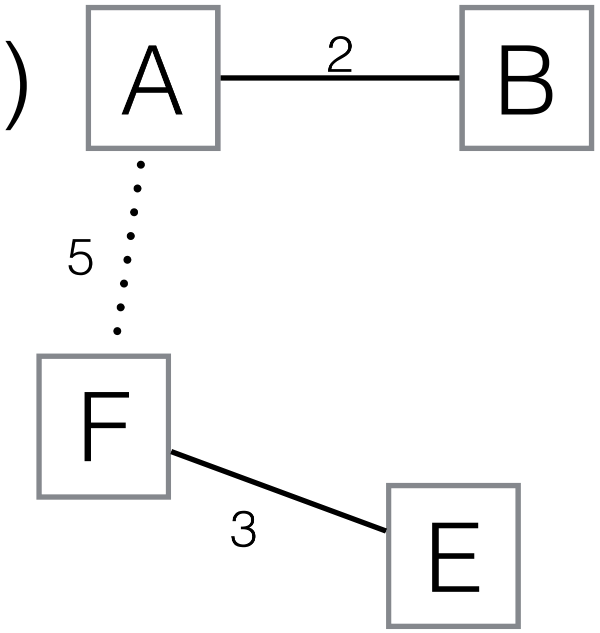
a)



b)

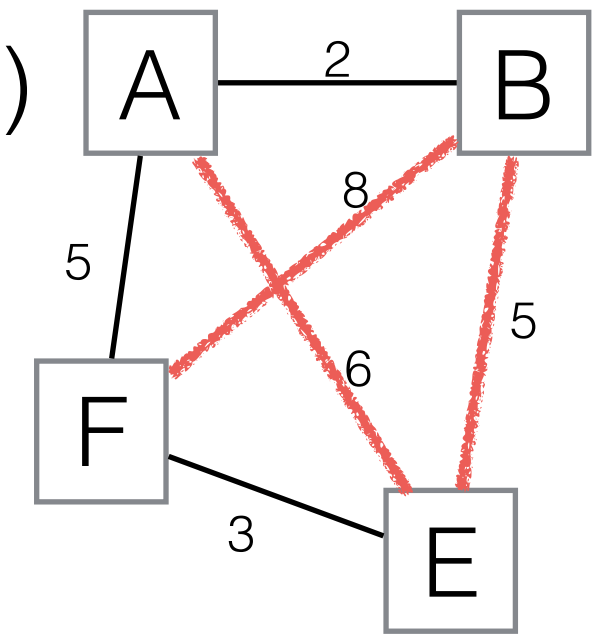


c)



could also have chosen (B,E)

d)

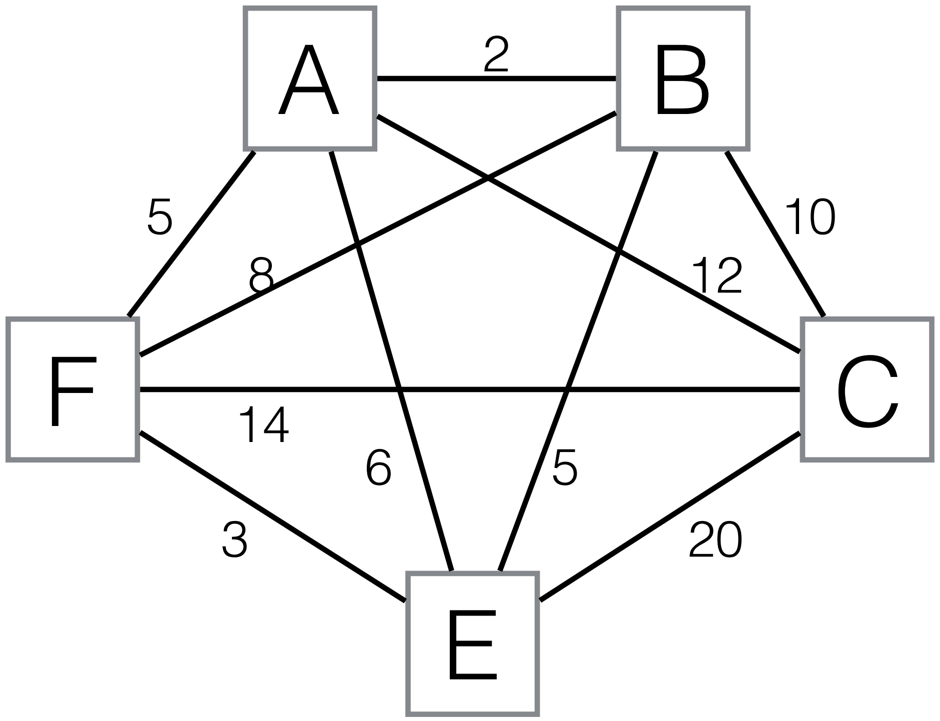


choose next lowest weight

cycle detected!
can not add (B,E)

nor (A,E)!
nor (F,B)!

If graph G is

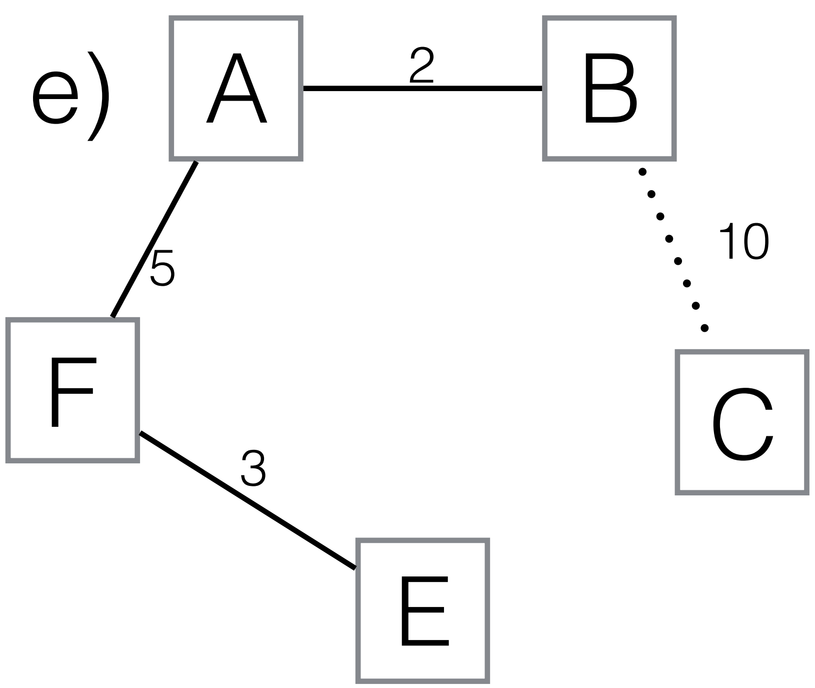


MCST = 2 + 3 + 5 + 10
= 20

MCSTs are not unique

easiest way to check for cycles in tree:

e)



don't add an edge if both of its end points are already in the tree