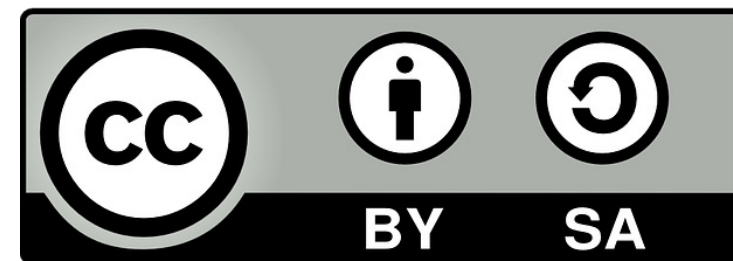


Algorithms (2024 Summer)

#9 : 幅優先探索, 深さ優先探索

矢谷 浩司

東京大学工学部電子情報工学科



2回目の授業アンケート実施中です！

ご協力のほど，なにとぞよろしくお願いいたします。

フォームのURLは#generalに流してあります。

今日からはグラフです！

アルゴリズムの授業で心が折れるポイントの一つ. . .

なんとなく圧倒されてしまう感じがありますが、
1つ1つは難しくない考え方なので、しっかり消化
していきましょう。

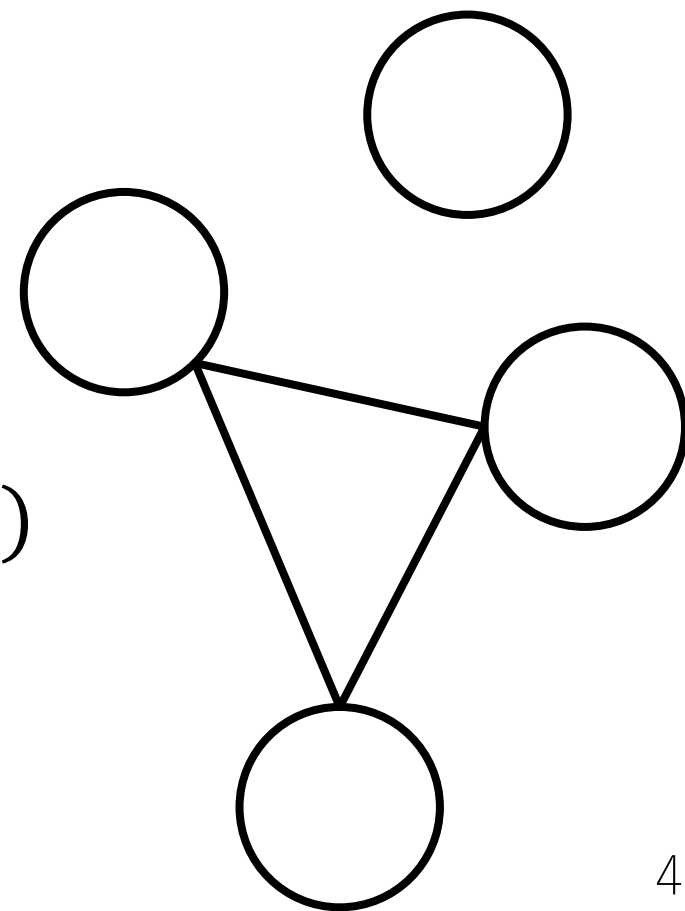
自分でコードを書いて復習するのもオススメです。

グラフ

ノード（頂点, vertices）と、ノード間の連結を表すエッジ（辺, edges）で構成されるデータ構造.

ノードは他のノードと連結されていないこともある.

V をノード, E を辺の集合として, $G = (V, E)$ と表すことが出来る.



グラフ

ツリー：連結で閉路が存在しない

→あるノードから、とある経路を一周して元のノードに戻ってくることができない

グラフ：閉路が存在することもある

→一周して元のノードに戻ってくる経路がある場合がある

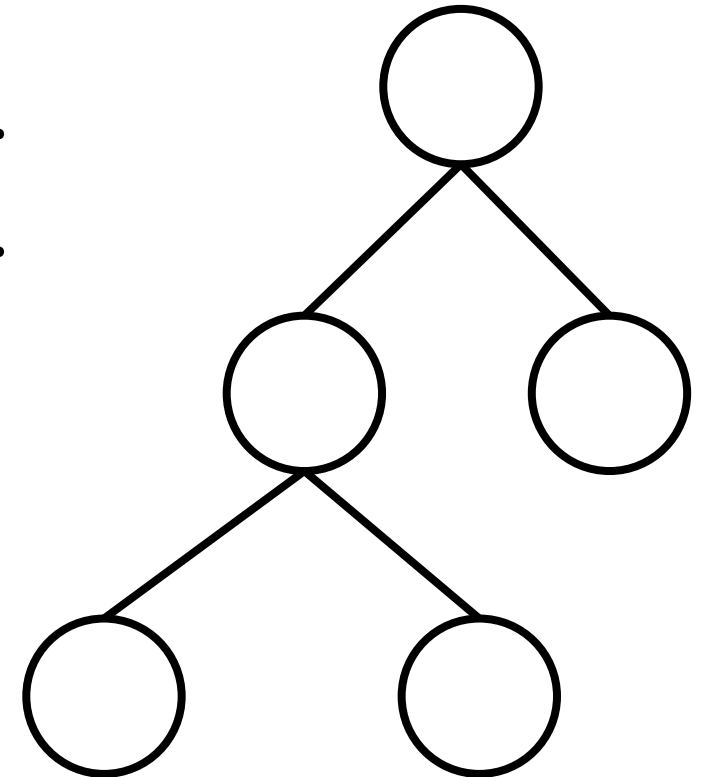
→連結でないこともある

ツリーはグラフの一種.

ツリー (木)

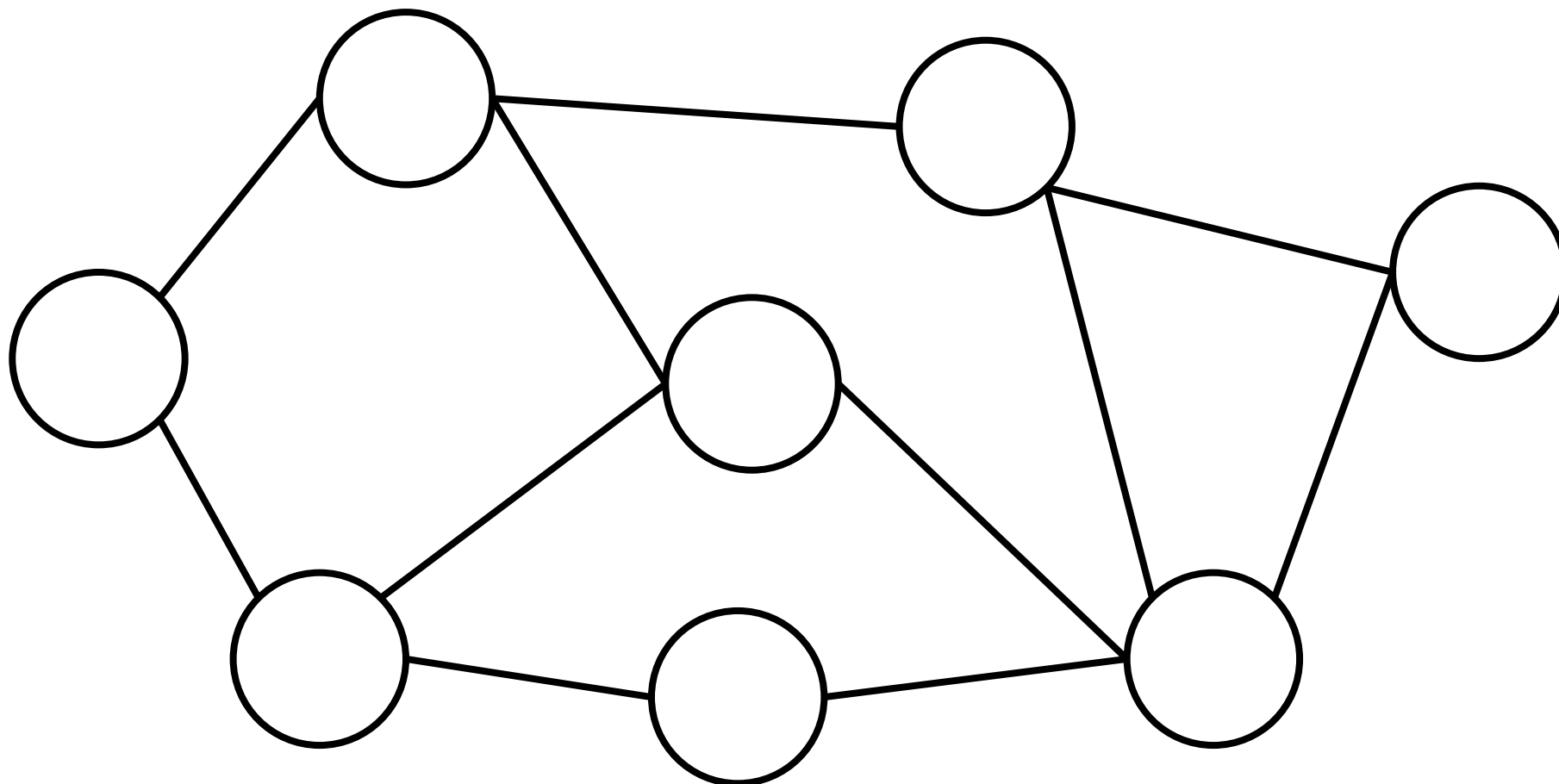
ノードの数が n であるグラフ T が木である, ということは以下のように言うこともできる (同値である).

- T には閉路がなく, 辺の数は $n - 1$ である.
- T は連結グラフで, 辺の数は $n - 1$ である.
- T の任意の2点を結ぶ経路は1つしかない.



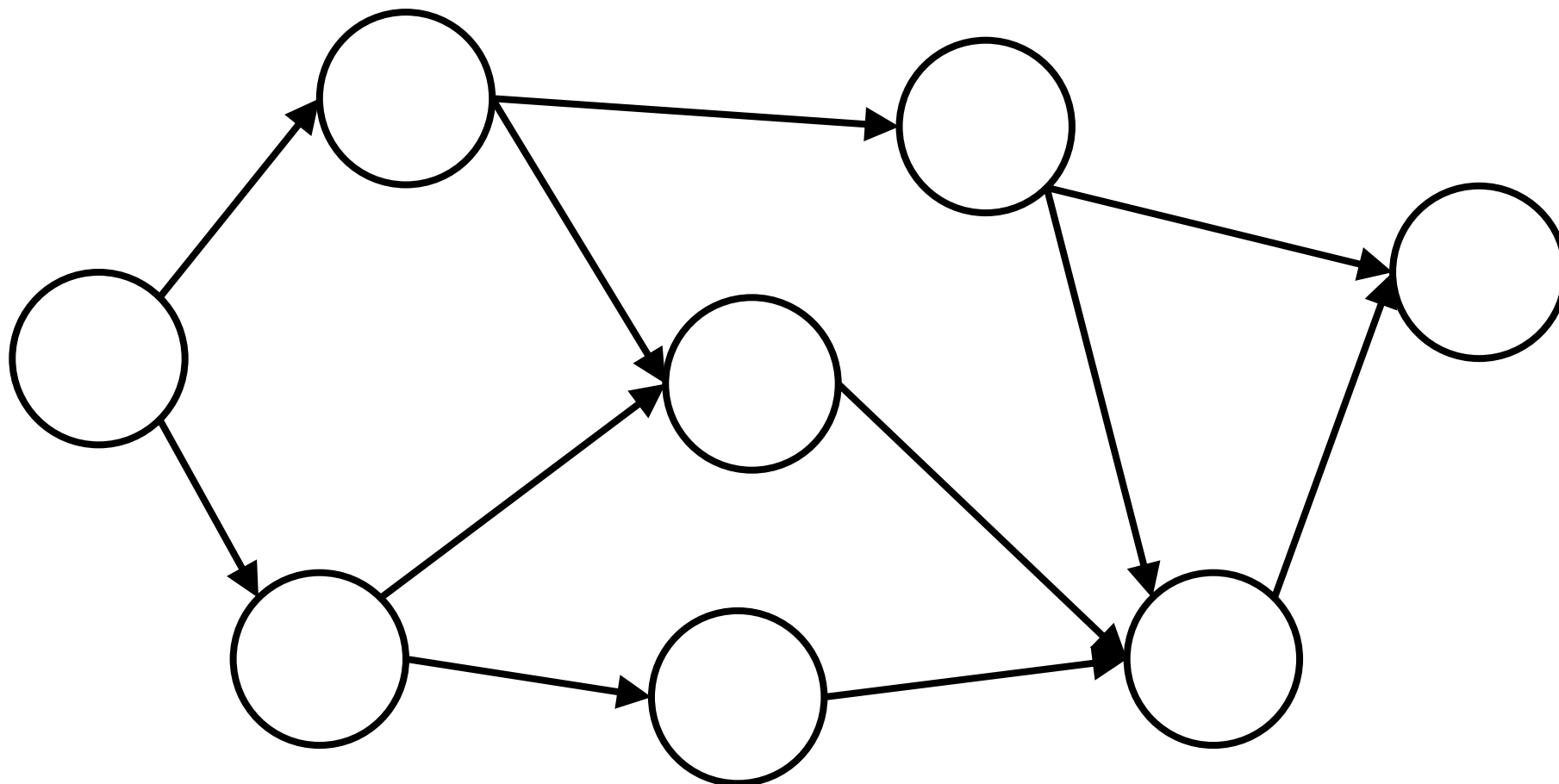
無向グラフ

ノード間が向きのない辺（エッジ）でつながっている。



有向グラフ

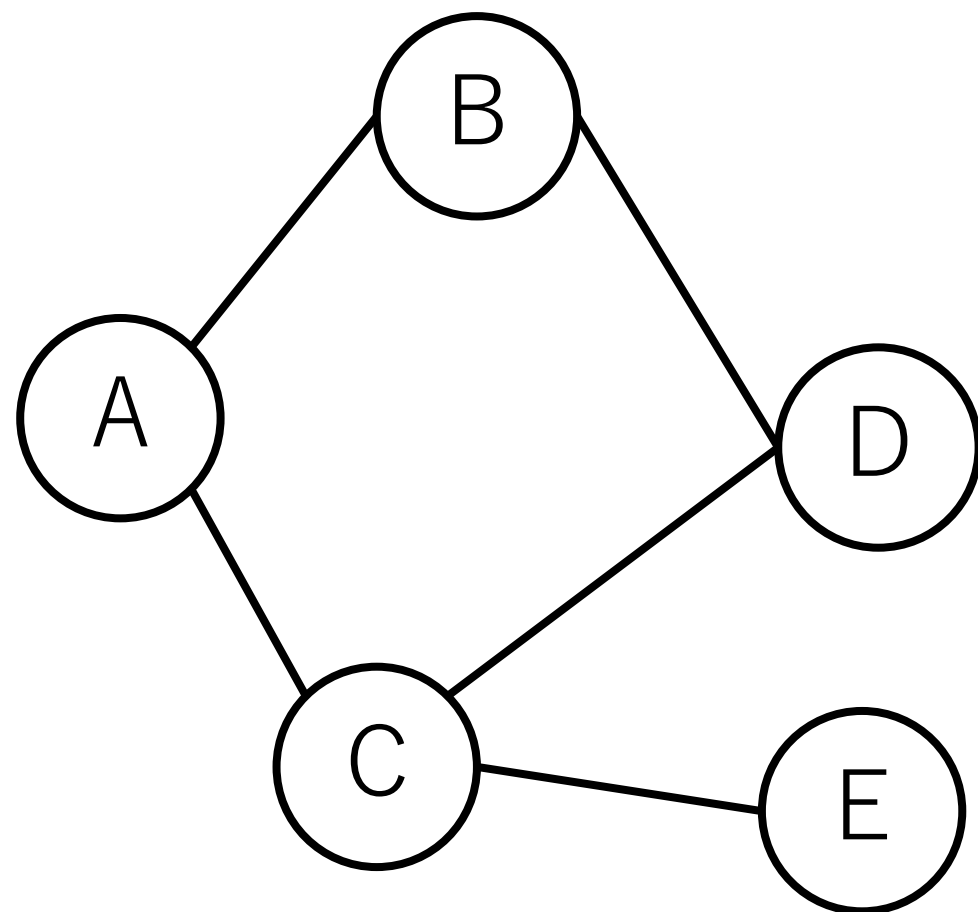
ノード間が向きのある辺（エッジ）でつながっている。



グラフを表すデータ構造

隣接リスト

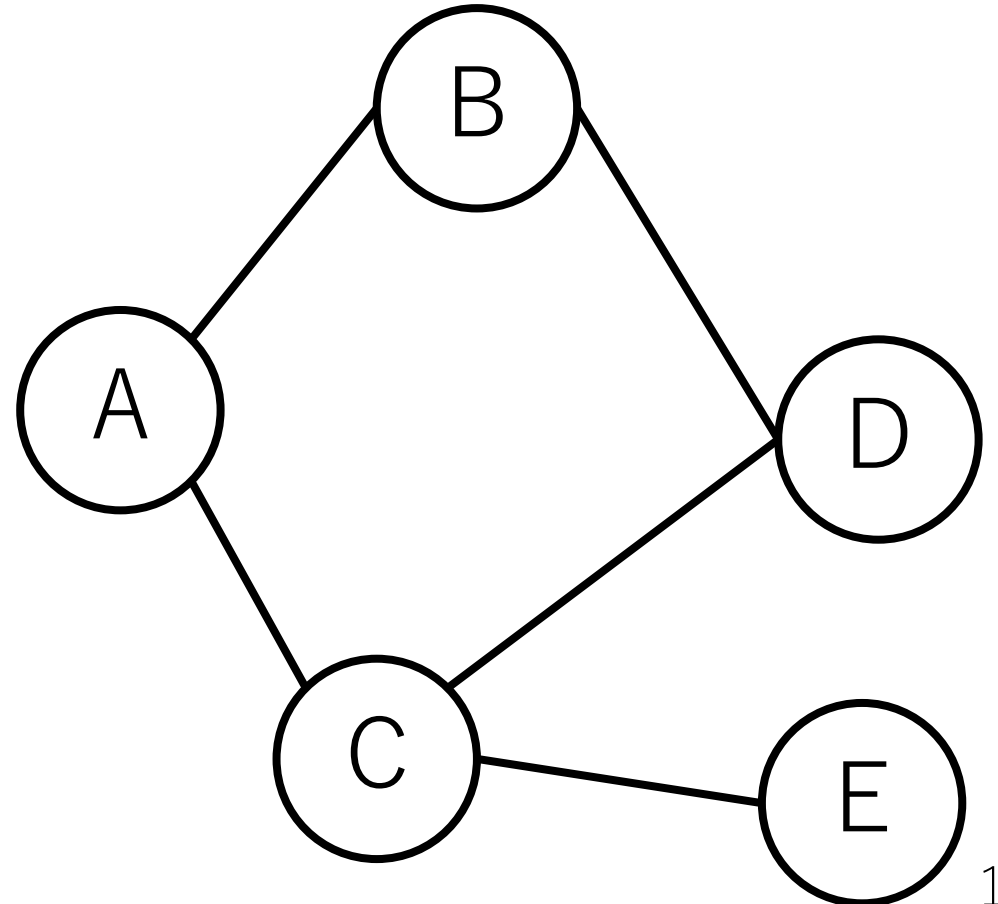
隣接行列



隣接リスト (無向グラフ)

各要素が各頂点の接続先を表す.

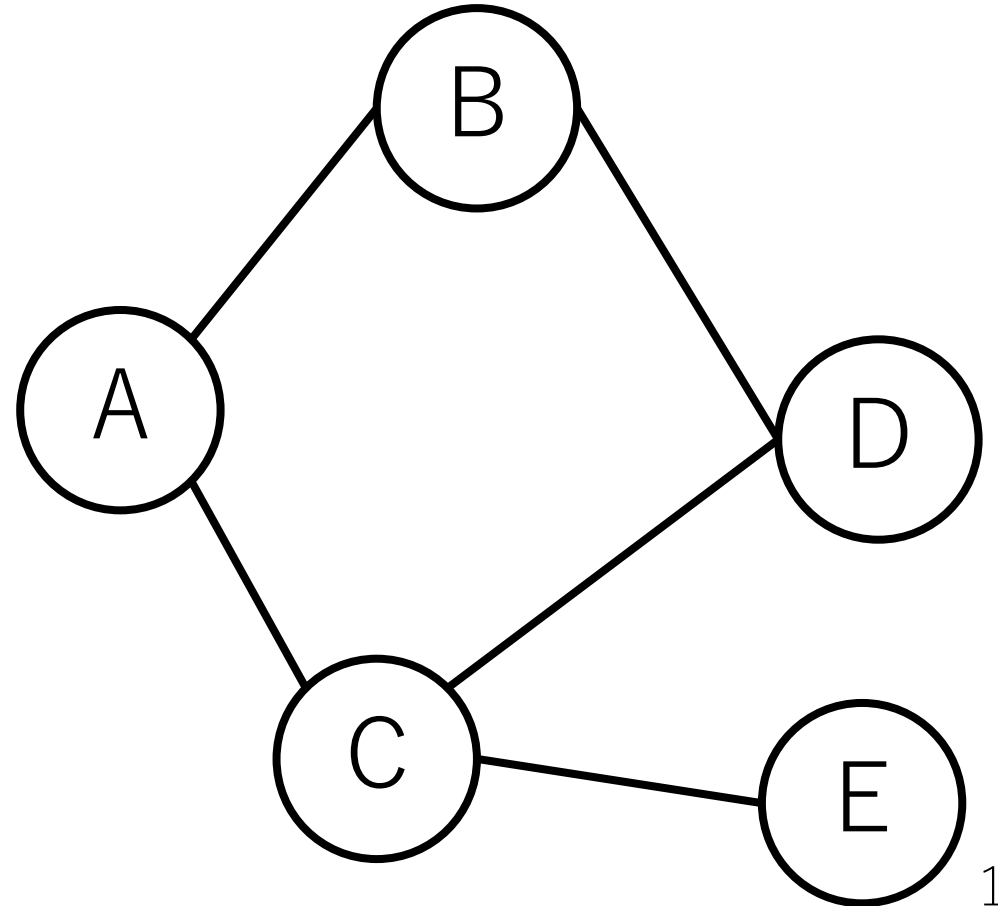
```
edge = [  
  [1, 2],    #ノードA node[0]  
  [0, 3],    #ノードB node[1]  
  [0, 3, 4], #ノードC node[2]  
  [1, 2],    #ノードD node[3]  
  [2]        #ノードE node[4]  
]
```



隣接リスト (無向グラフ)

各要素が[接続元, 接続先]になっている.

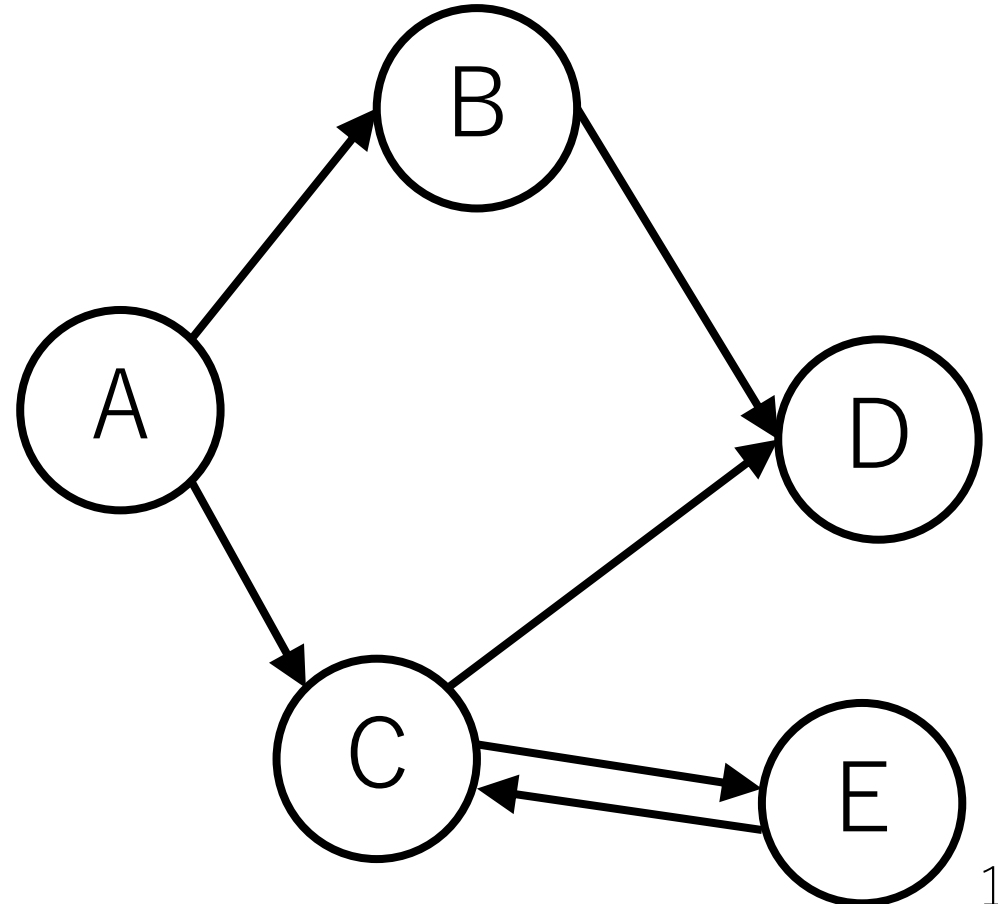
```
edge = [  
[0, 1], [1, 0],    # A-B  
[0, 2], [2, 0],    # A-C  
[1, 3], [3, 1],    # B-D  
[2, 3], [3, 2],    # C-D  
[2, 4], [4, 2]     # C-E  
]
```



隣接リスト (有向グラフ)

各要素が各頂点の接続先を表す.

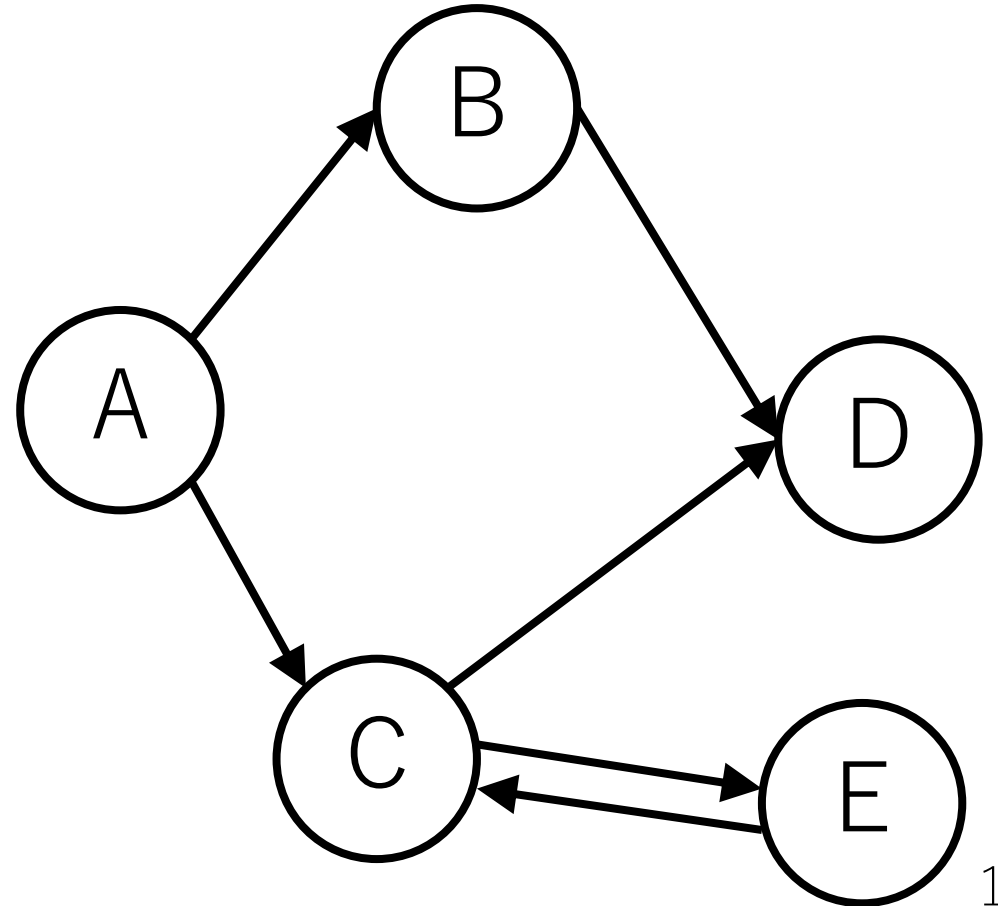
```
edge = [  
  [1, 2],    #ノードA node[0]  
  [3],       #ノードB node[1]  
  [3, 4],    #ノードC node[2]  
  [],        #ノードD node[3]  
  [2]        #ノードE node[4]  
]
```



隣接リスト (有向グラフ)

各要素が[接続元, 接続先]になっている.

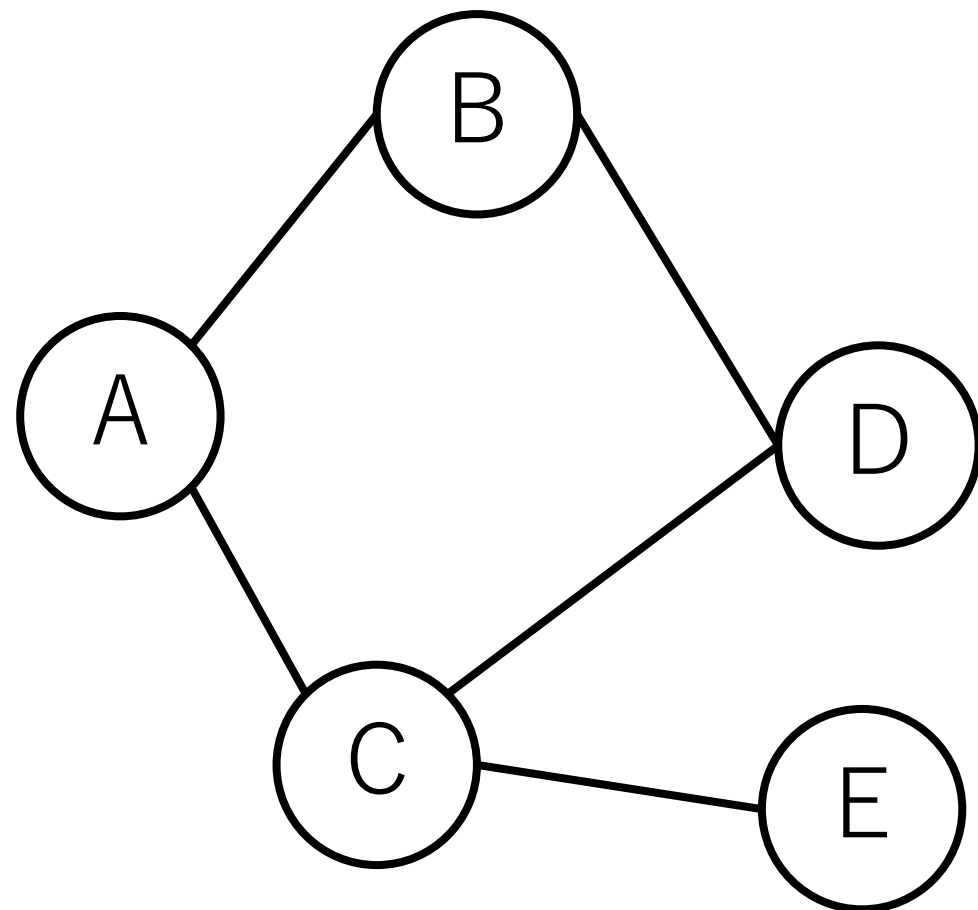
```
edge = [  
[0, 1],    # A->B  
[0, 2],    # A->C  
[1, 3],    # B->D  
[2, 3],    # C->D  
[2, 4],    # C->E  
[4, 2]     # E->C  
]
```



隣接行列 (無向グラフ)

無向グラフの場合は対称行列.

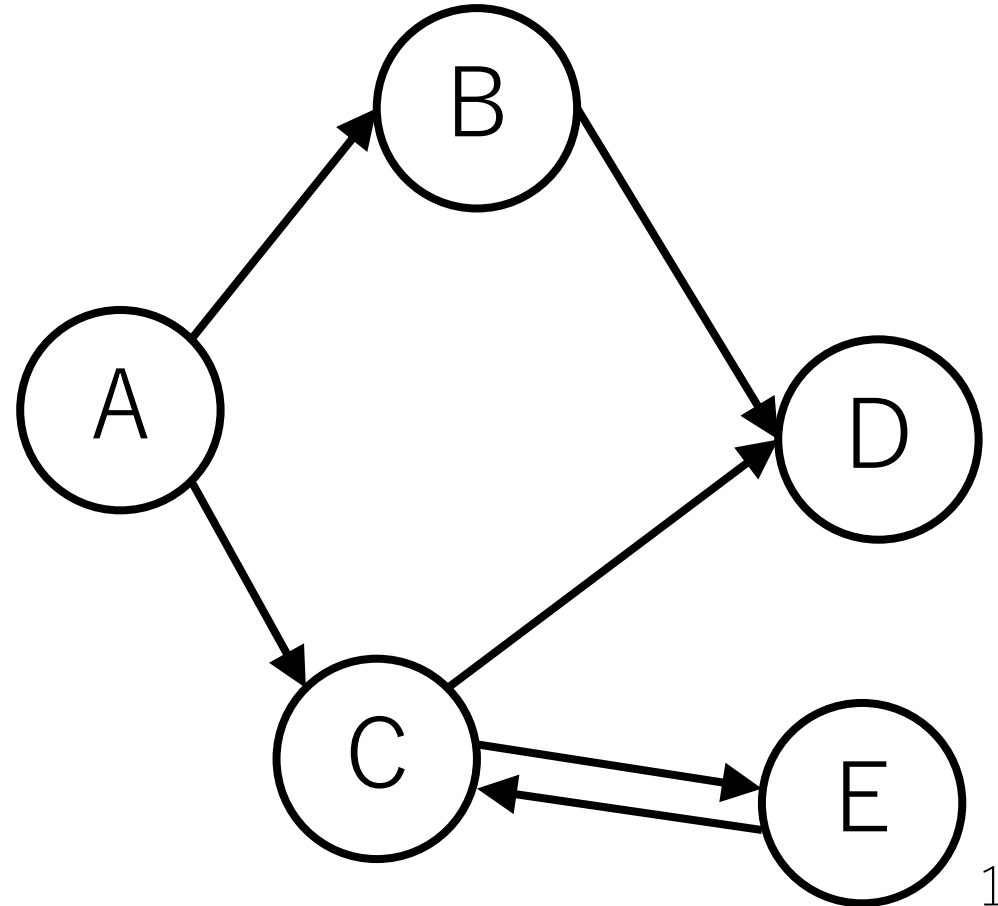
```
edge = [  
    [0, 1, 1, 0, 0], #ノードA  
    [1, 0, 0, 1, 0], #ノードB  
    [1, 0, 0, 1, 1], #ノードC  
    [0, 1, 1, 0, 0], #ノードD  
    [0, 0, 1, 0, 0] #ノードE  
]
```



隣接行列 (有向グラフ)

繋がっている向きにのみ値を持つ。

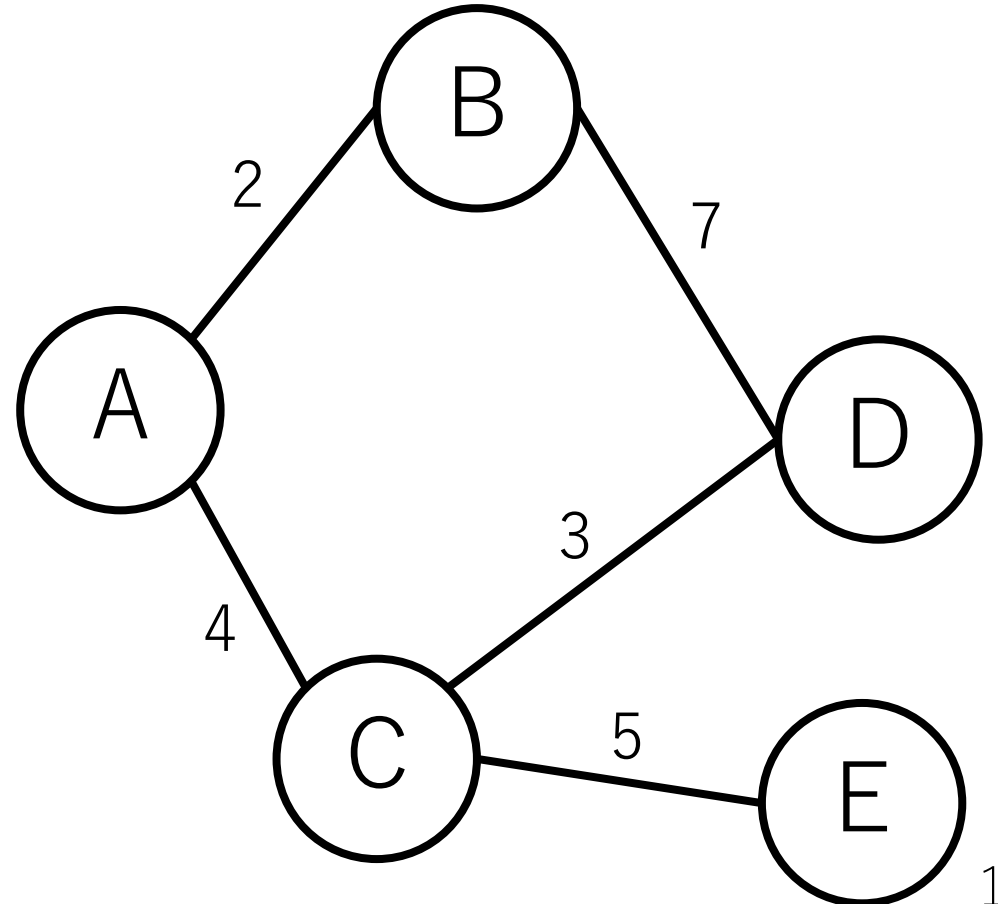
```
edge = [  
  [0, 1, 1, 0, 0], #ノードA  
  [0, 0, 0, 1, 0], #ノードB  
  [0, 0, 0, 1, 1], #ノードC  
  [0, 0, 0, 0, 0], #ノードD  
  [0, 0, 1, 0, 0] #ノードE  
]
```



隣接リスト (無向グラフ, コスト付き)

[接続先, コスト]で並んでいる.

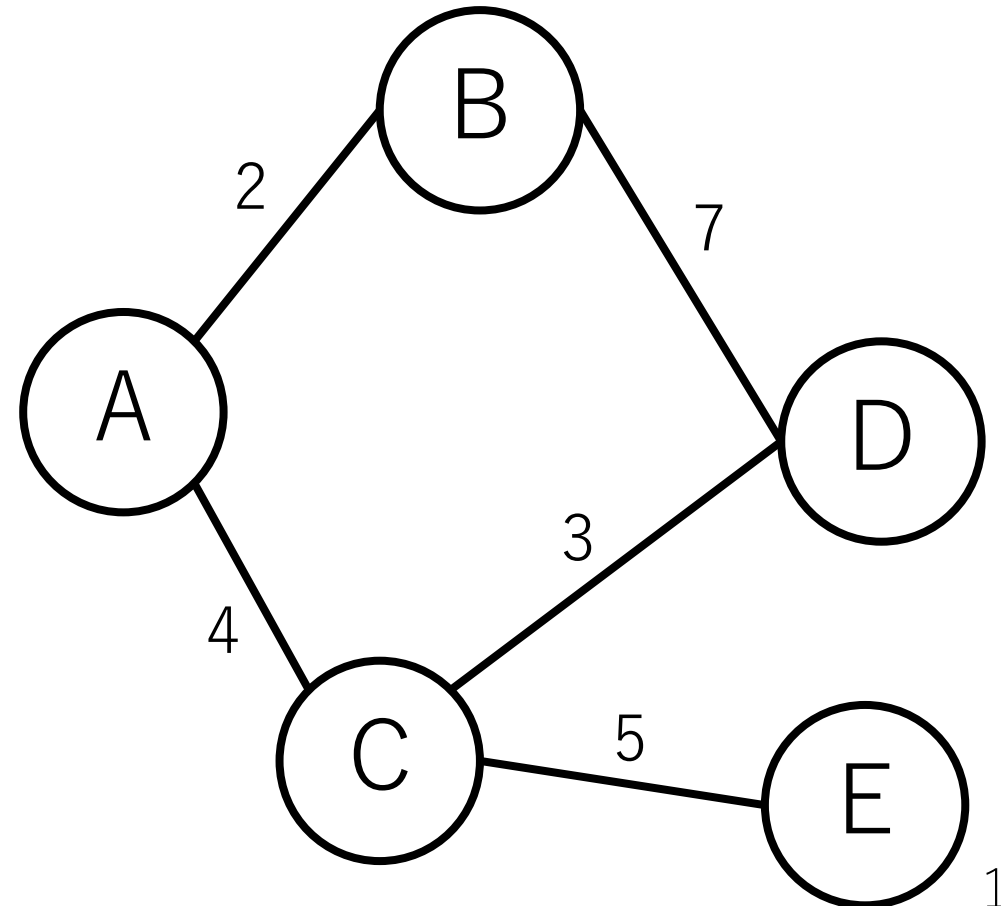
```
edge = [  
  [[1, 2], [2, 4]],      #ノードA  
  [[0, 2], [3, 7]],      #ノードB  
  [[0, 4], [3, 3], [4, 5]], #ノードC  
  [[1, 7], [2, 3]],      #ノードD  
  [[2, 5]]               #ノードE  
]
```



隣接リスト (無向グラフ, コスト付き)

[接続元, 接続先, コスト]で並んでいる.

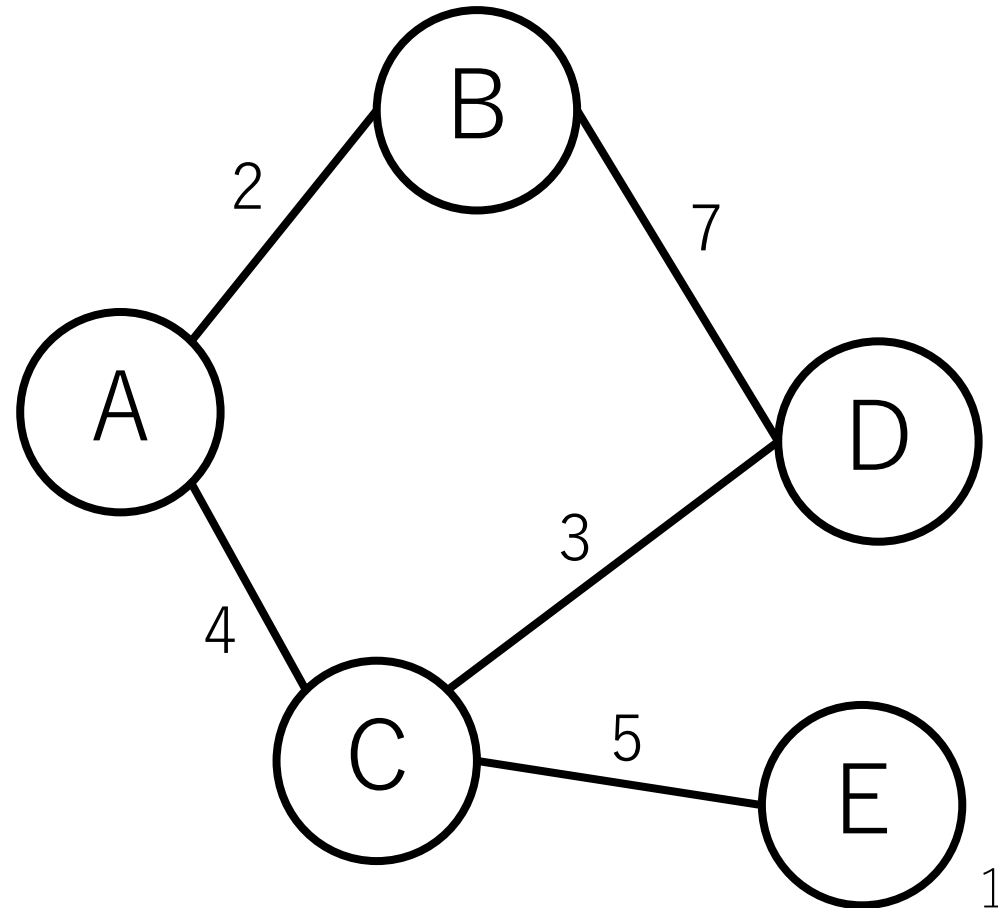
```
edge = [  
  [0, 1, 2], [1, 0, 2],      # A-B  
  [0, 2, 4], [2, 0, 4],      # A-C  
  [1, 3, 7], [3, 1, 7],      # B-D  
  [2, 3, 3], [3, 2, 3],      # C-D  
  [2, 4, 5], [4, 2, 5]      # C-E  
]
```



隣接行列 (無向グラフ)

無向グラフの場合は対称行列。
値は経路のコストを表す。

```
edge = [  
    [0, 2, 4, 0, 0], #ノードA  
    [2, 0, 0, 7, 0], #ノードB  
    [4, 0, 0, 3, 5], #ノードC  
    [0, 7, 3, 0, 0], #ノードD  
    [0, 0, 5, 0, 0] #ノードE  
]
```

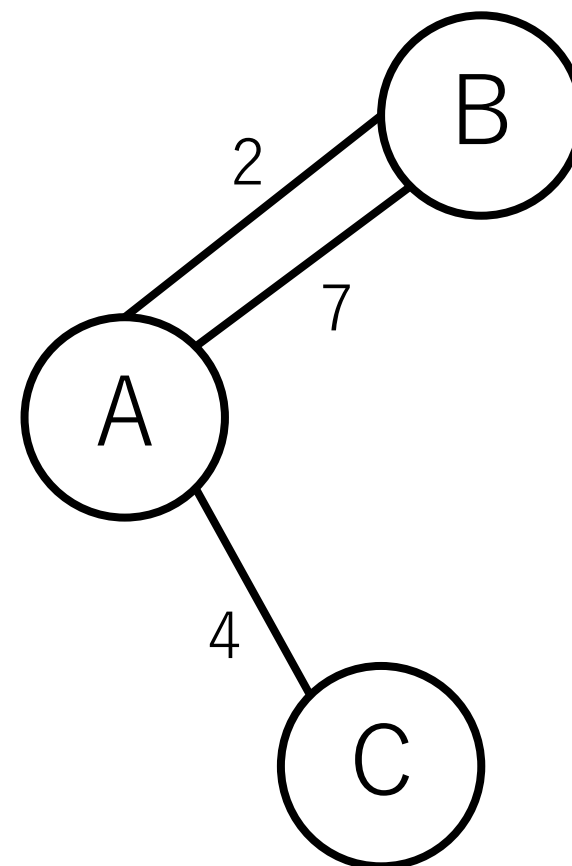


	隣接行列	隣接リスト
空間計算量	常に $O(V ^2)$. 辺が少ないと (疎なグラフ) メモリの無駄遣いになる.	辺の数に応じたメモリ量しか使わず, $O(E)$.
ある1つの辺の有無・コスト	ある辺の有無やコストの参照を $O(1)$ で計算可能.	ある辺の有無やコストを調べる時, 探索が必要.
ある1つの辺の追加・削除	$O(1)$ 回の操作で実現可能.	場合によっては, 辺の追加, 削除がやや面倒.
隣接ノードを全部取得	$O(V)$ 回の操作が必要.	調べているノードに接続している辺の回数回の操作で可能.
グラフ内の辺を全部取得	$O(V ^2)$ 回の操作が必要.	$O(E)$ 回の操作で可能.

多重辺がある場合

隣接リストならば，コストの有無にかかわらず表現可能.

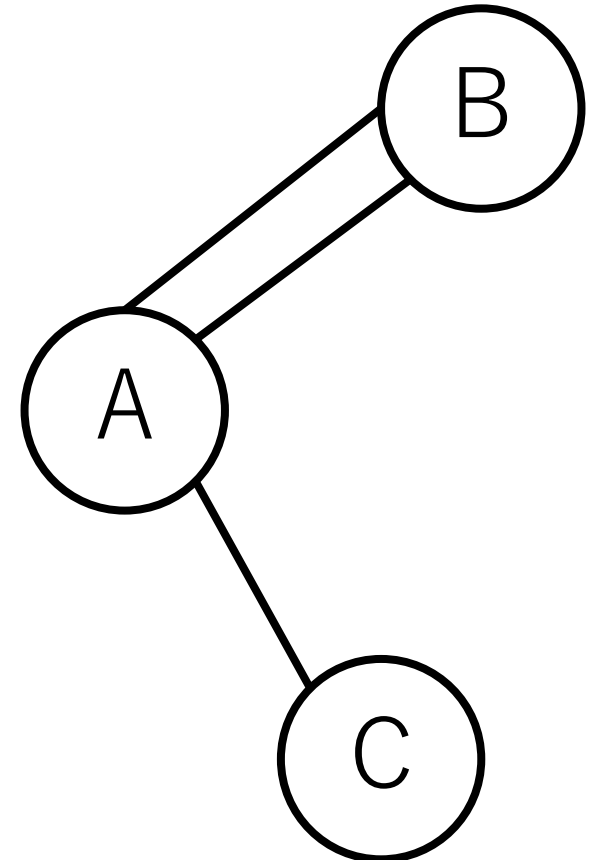
```
edge = [  
  [0, 1, 2], [1, 0, 2],    # A-B  
  [0, 1, 7], [1, 0, 7],    # A-B  
  [0, 2, 4], [2, 0, 4],    # A-C  
]
```



多重辺がある場合

隣接行列ならば，コストが無い場合は表現可能．各要素の値が対応するノード間の辺の数になる．

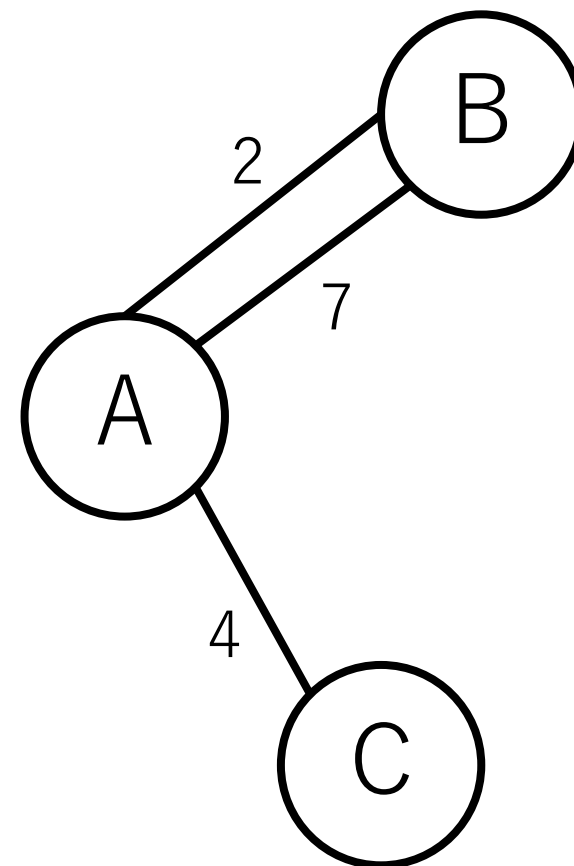
```
edge = [  
    [0, 2, 1], #ノードA  
    [2, 0, 0], #ノードB  
    [1, 0, 0], #ノードC  
]
```



多重辺がある場合

コストがある場合でも，以下のように強引に表せなくはないが，これを「行列」というかは微妙. . . 😅

```
edge = [  
    [0, [2, 7], 4], #ノードA  
    [[2, 7], 0, 0], #ノードB  
    [4, 0, 0], #ノードC  
]
```



隣接リストと隣接行列

少なくとも多重辺がない場合には，どちらを使ってもグラフを正確に表現することが可能。

ただし，どちらを使うかによってアルゴリズムの計算量が変わる可能性がある。

アルゴリズムでどんな情報にアクセスすべきかに注目をして，どちらを選ぶべきか判断できると良いです。

今日の問題：グラフの探索

全てのノードが連結されているかを確認したい。

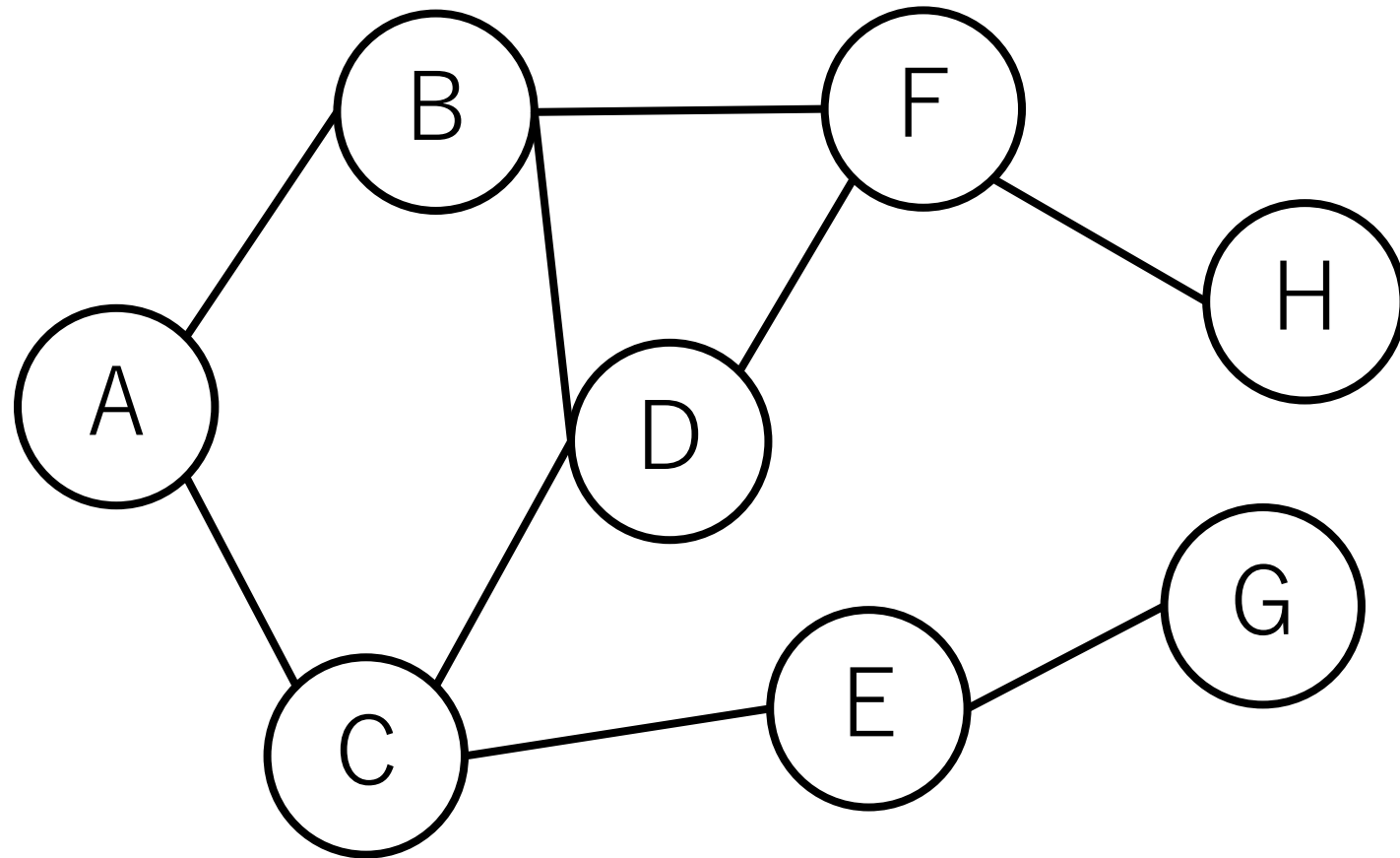
あるノードとあるノードを結ぶ経路が存在するかを確かめたい。

ある特定のノードを探し出したい。

ある条件を満たすノードに操作を行う（色を塗るなど）。

グラフの探索問題

AとGは繋がっているか？



考え方

大きく分けて2つある.

後戻りしないように, 可能性のあるルート全てにおいて1ステップずつ行くパターン.

とりあえず行けるところまで行き, ダメなら後戻りするパターン.

BFSとDFS

幅優先探索 (Breadth first search, BFS)

後戻りしないように、可能性のあるルート全てにおいて1ステップずつ行くパターン。

深さ優先探索 (Depth first search, DFS)

とりあえず行けるところまで行き、ダメなら後戻りするパターン。

BFS

幅優先探索 (Breadth first search, BFS)

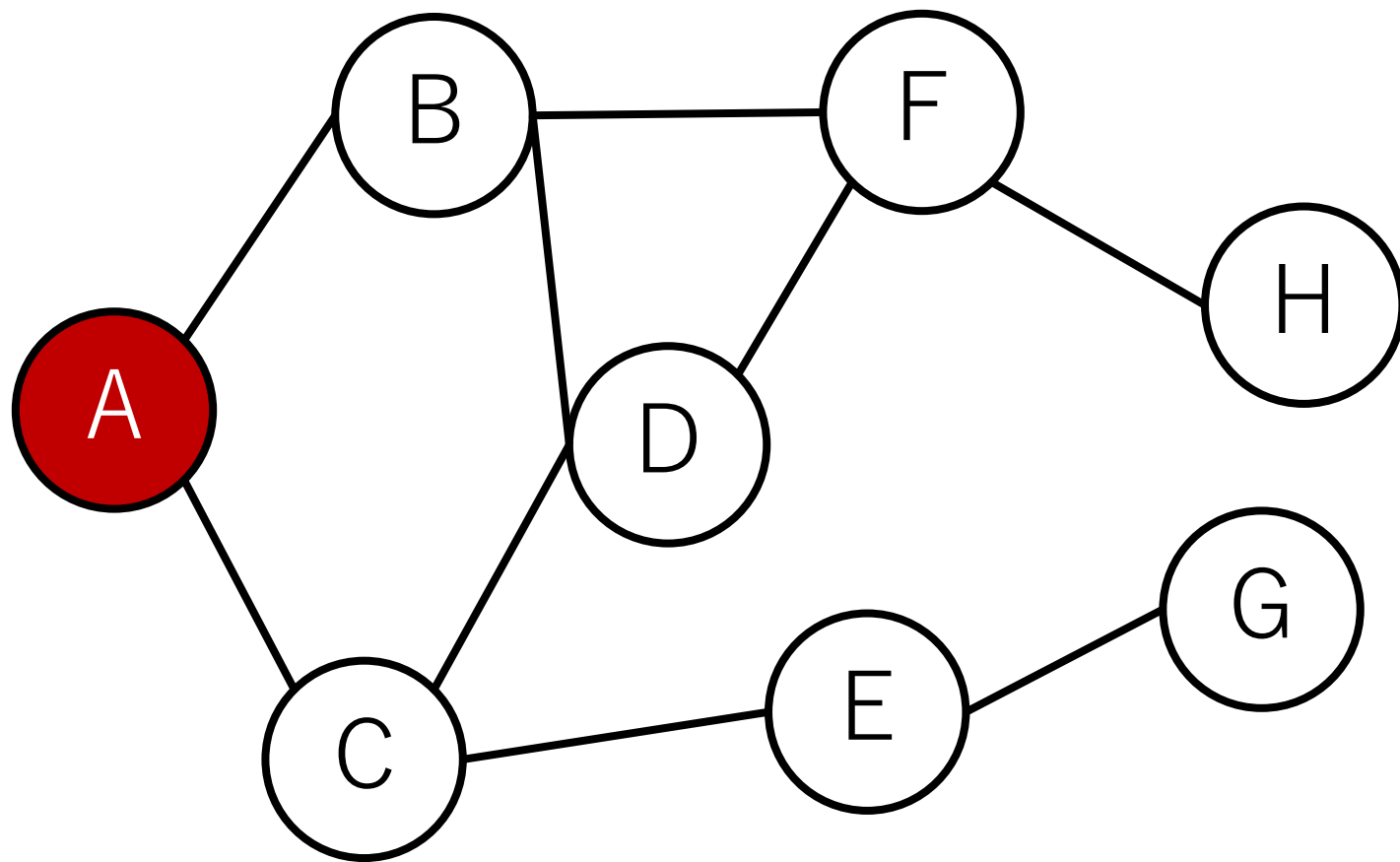
後戻りしないように、可能性のあるルート全てにおいて1ステップずつ行くパターン。

深さ優先探索 (Depth first search, DFS)

とりあえず行けるところまで行き、ダメなら後戻りするパターン。

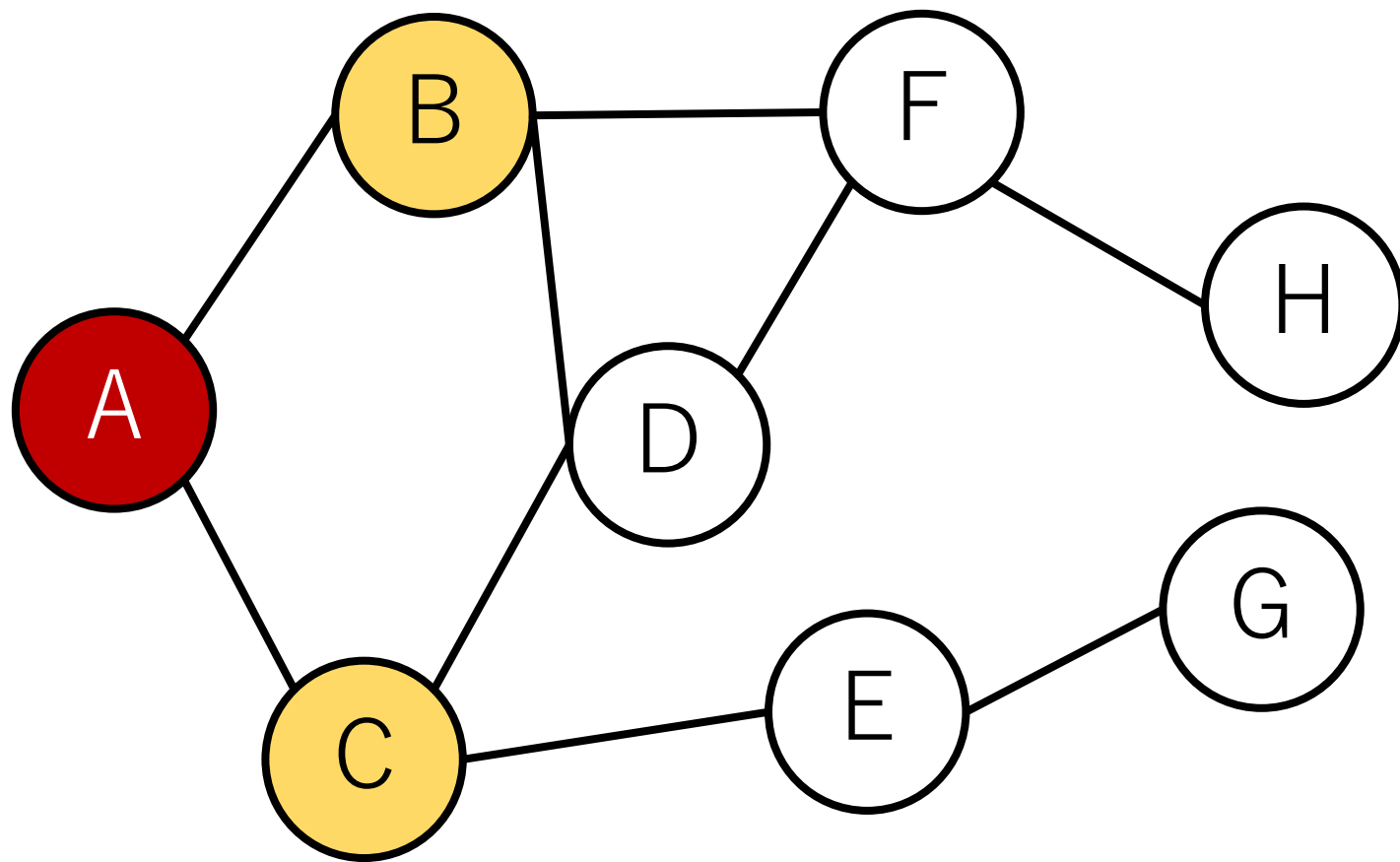
BFSの例

Aからスタート.



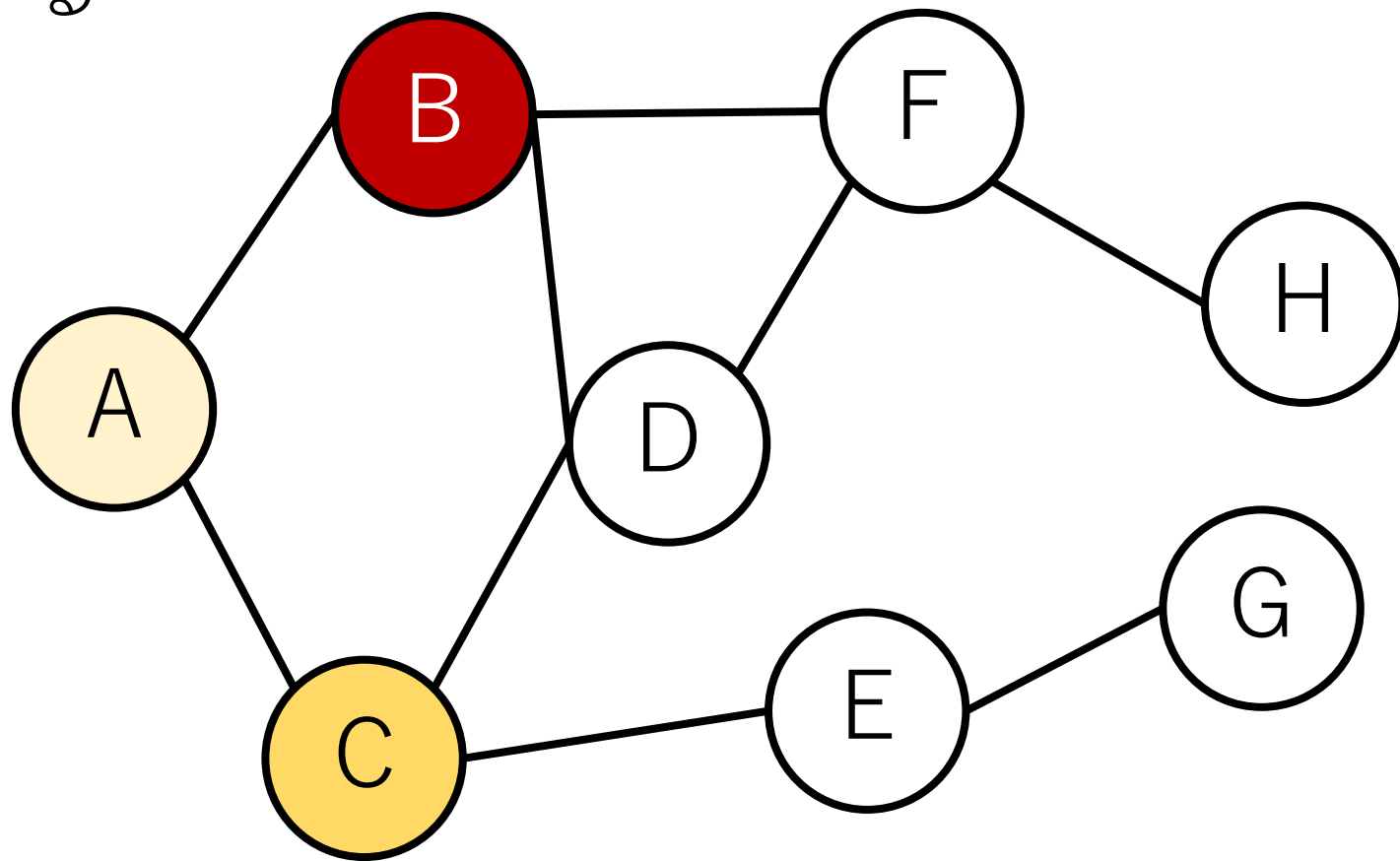
BFSの例

1ステップでつながっているのは、BとC.



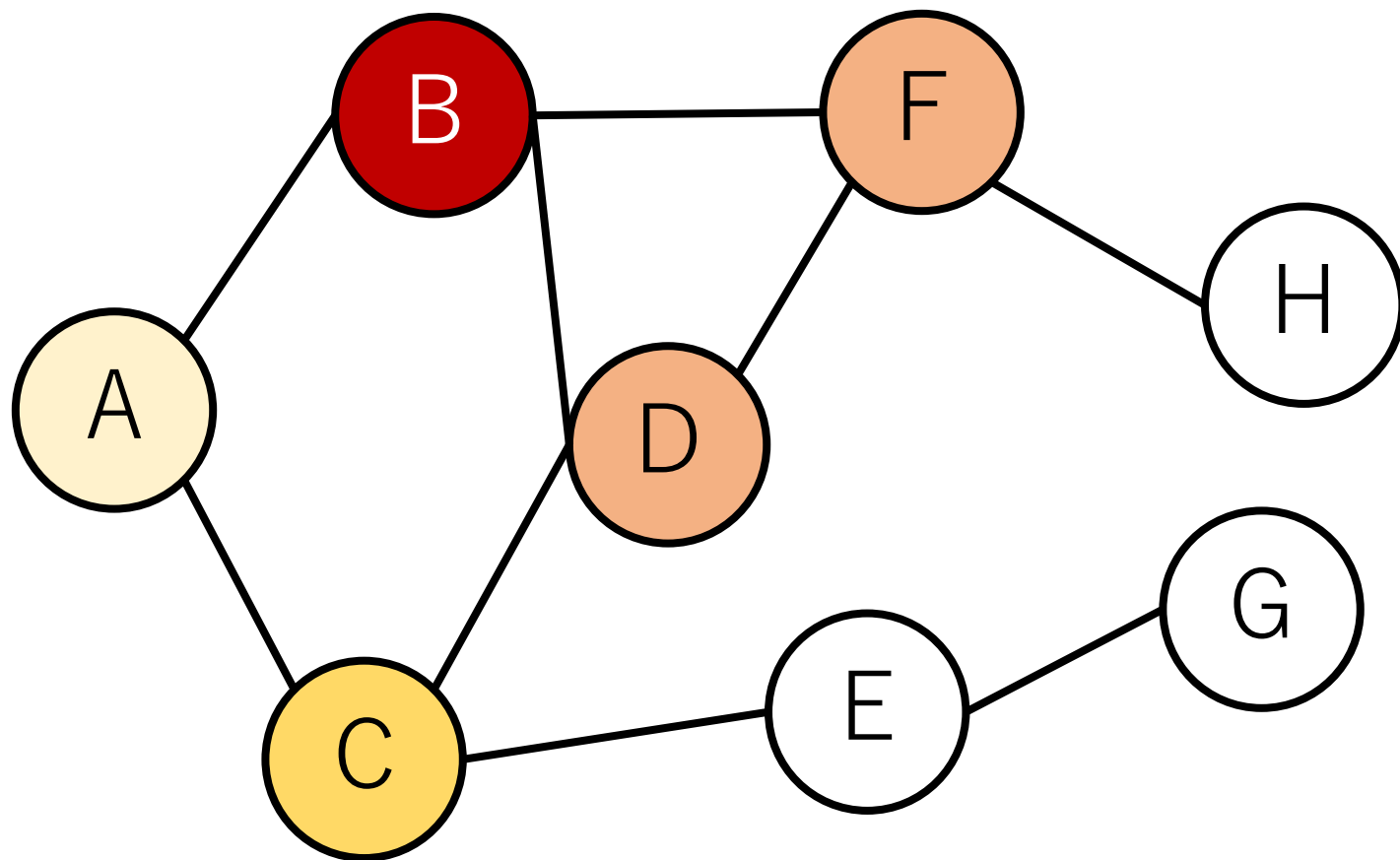
BFSの例

Bから1ステップでつながる
ノードを見る。(Cから
始めてもよい。)



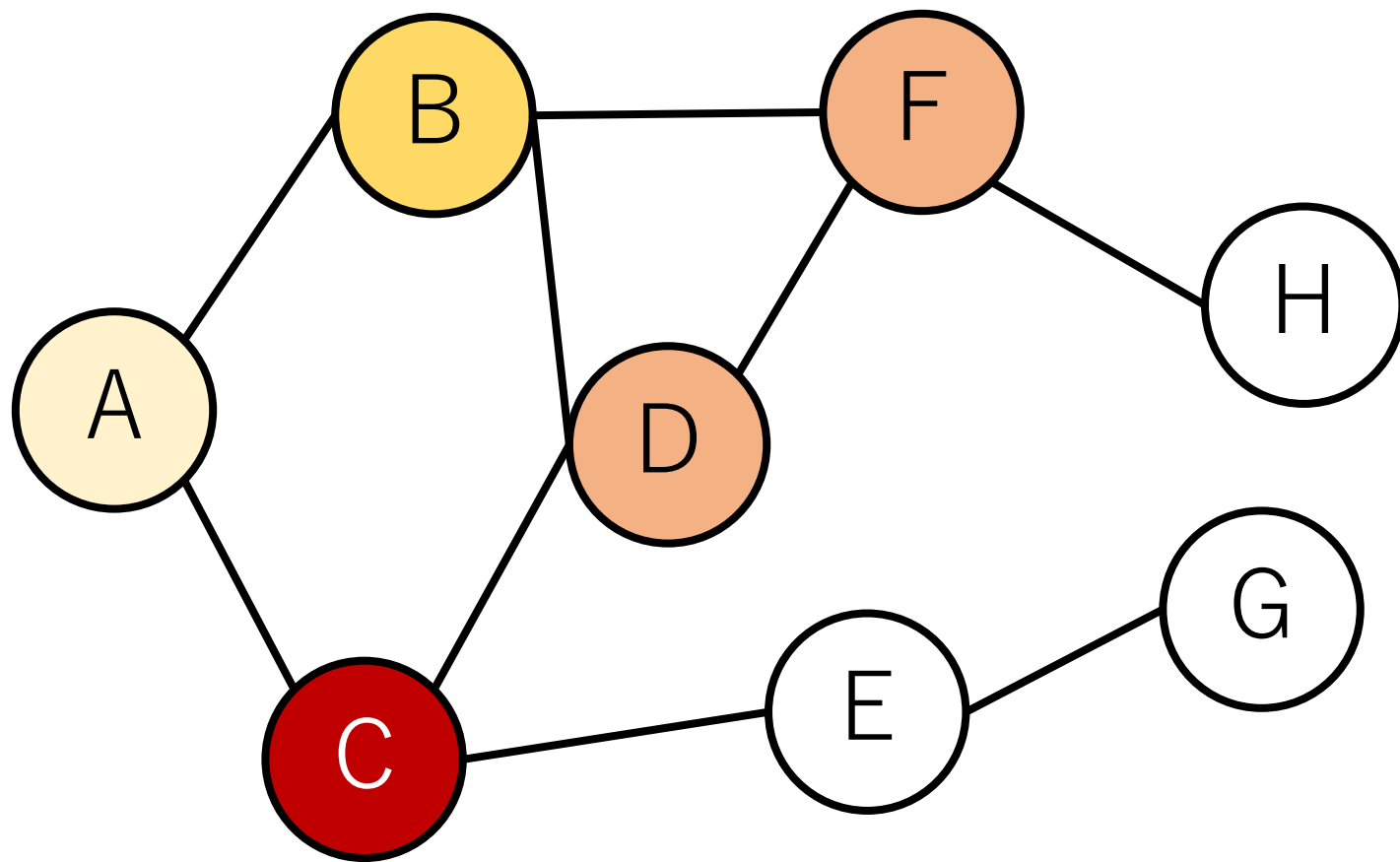
BFSの例

DとFを発見.



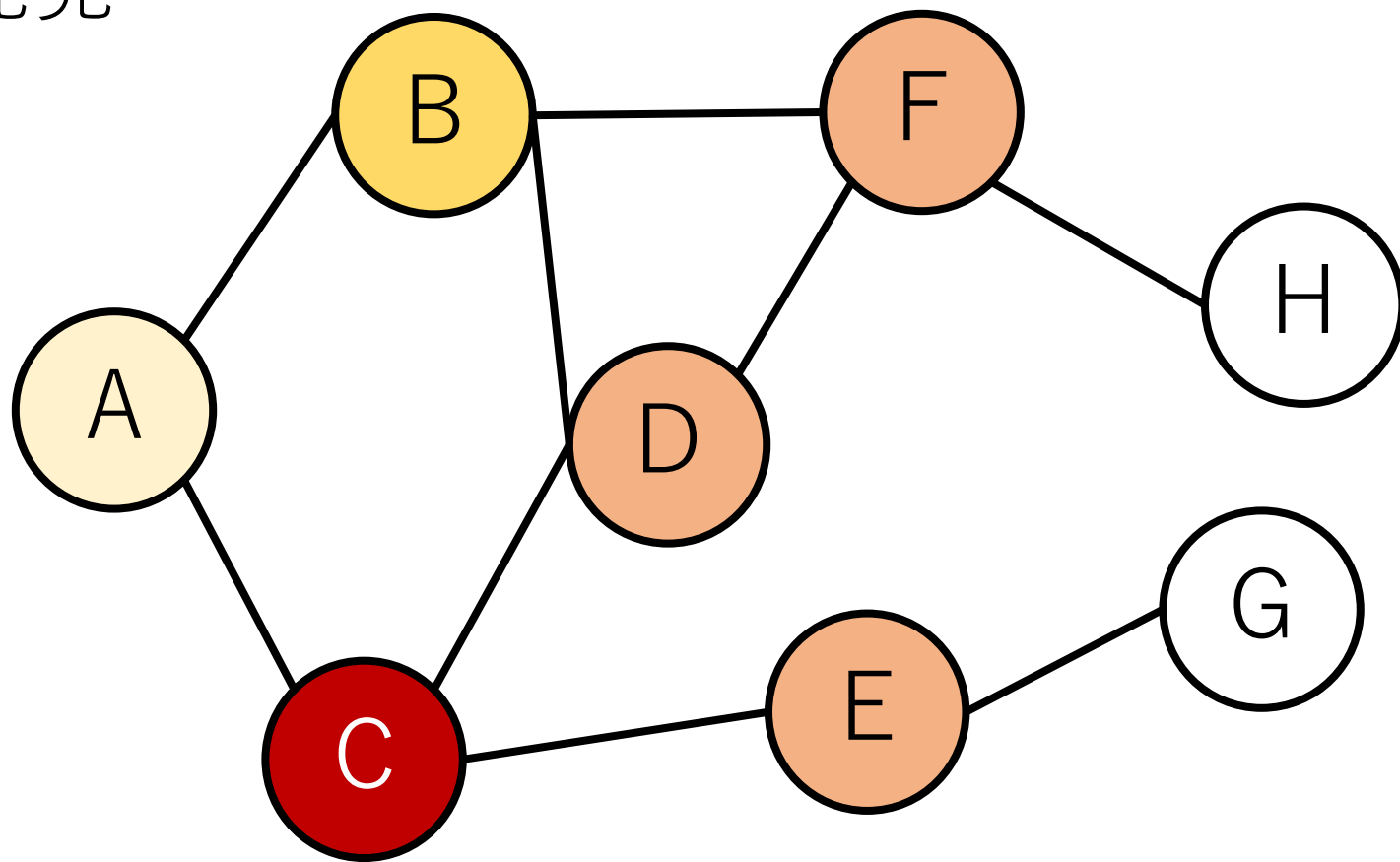
BFSの例

Cに移る.



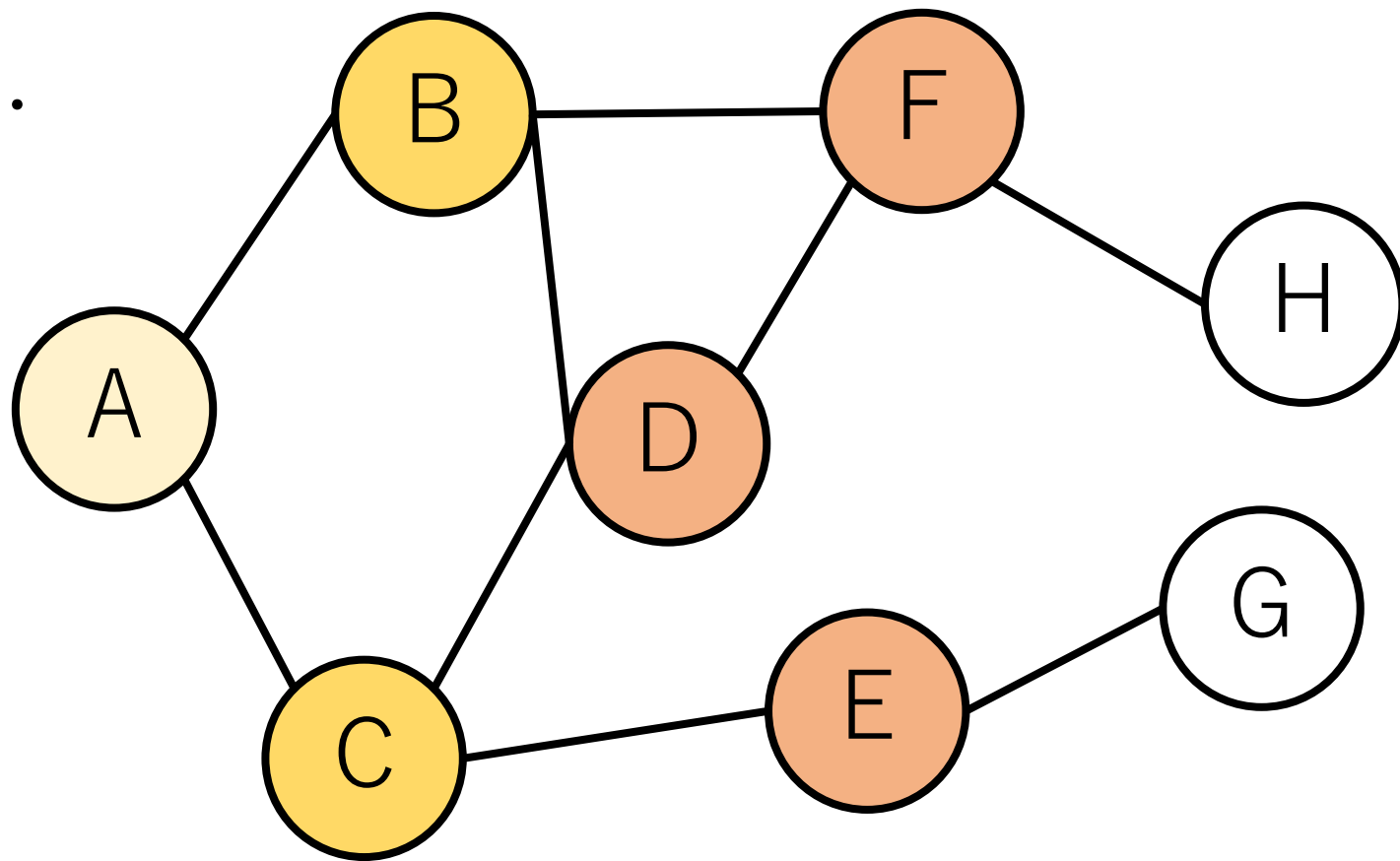
BFSの例

Eを新しく発見.
済み) (Dは発見済み)



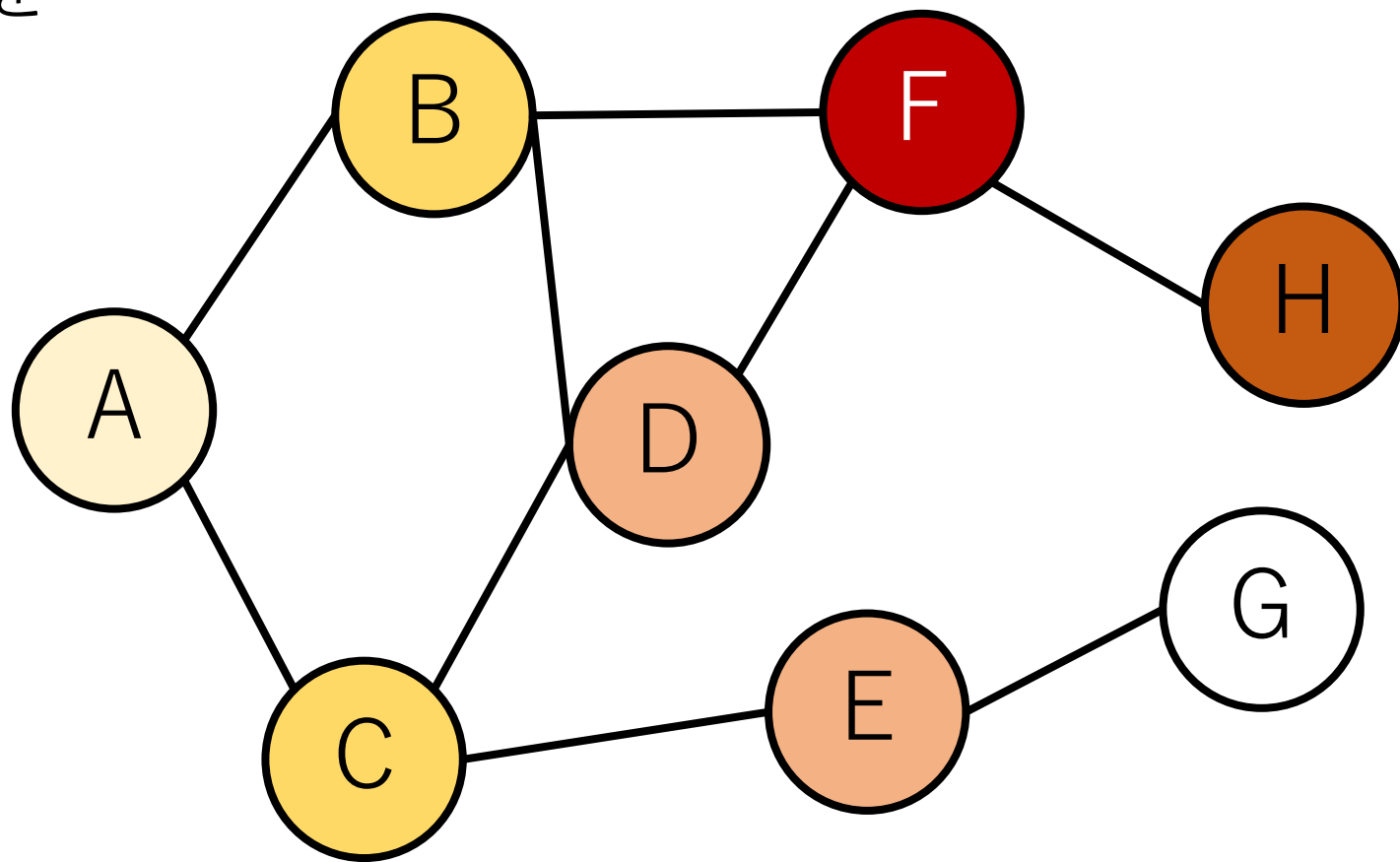
BFSの例

これでB, Cから辿れる
ノードの探索は全部終了.



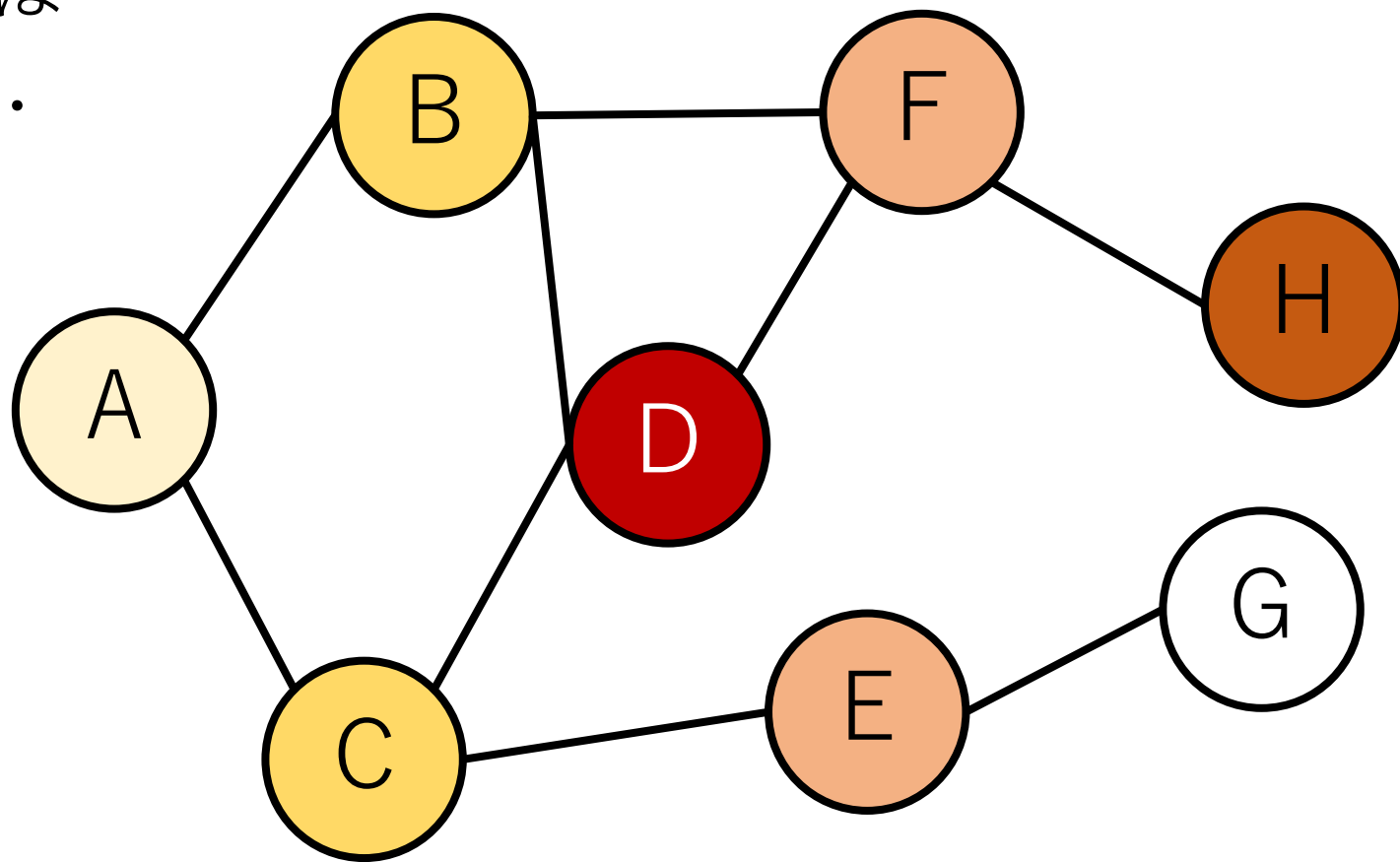
BFSの例

Fに移って，同様の探索を行い，Hを発見．



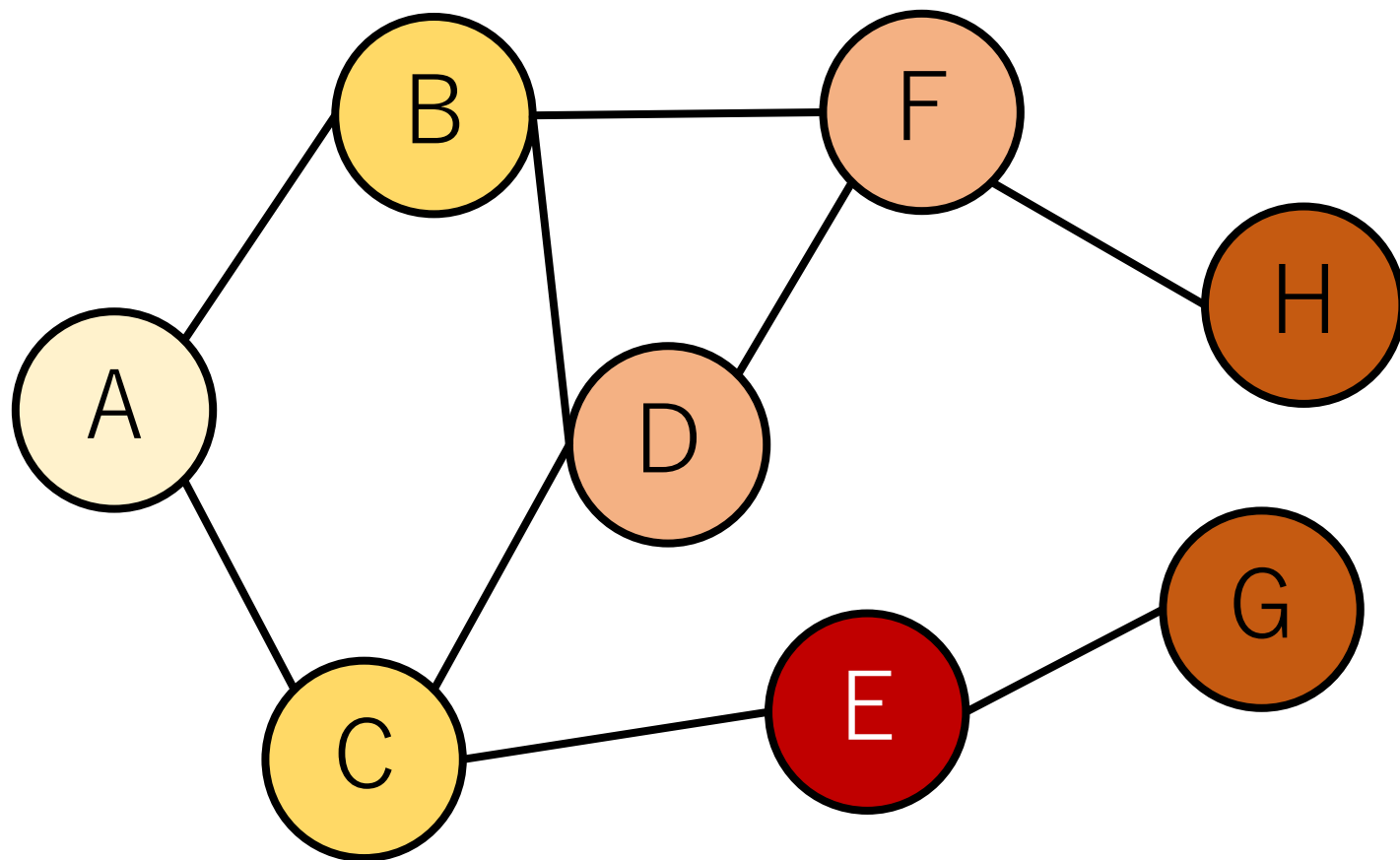
BFSの例

Dからは未発見のノードはないので、何も行わない。



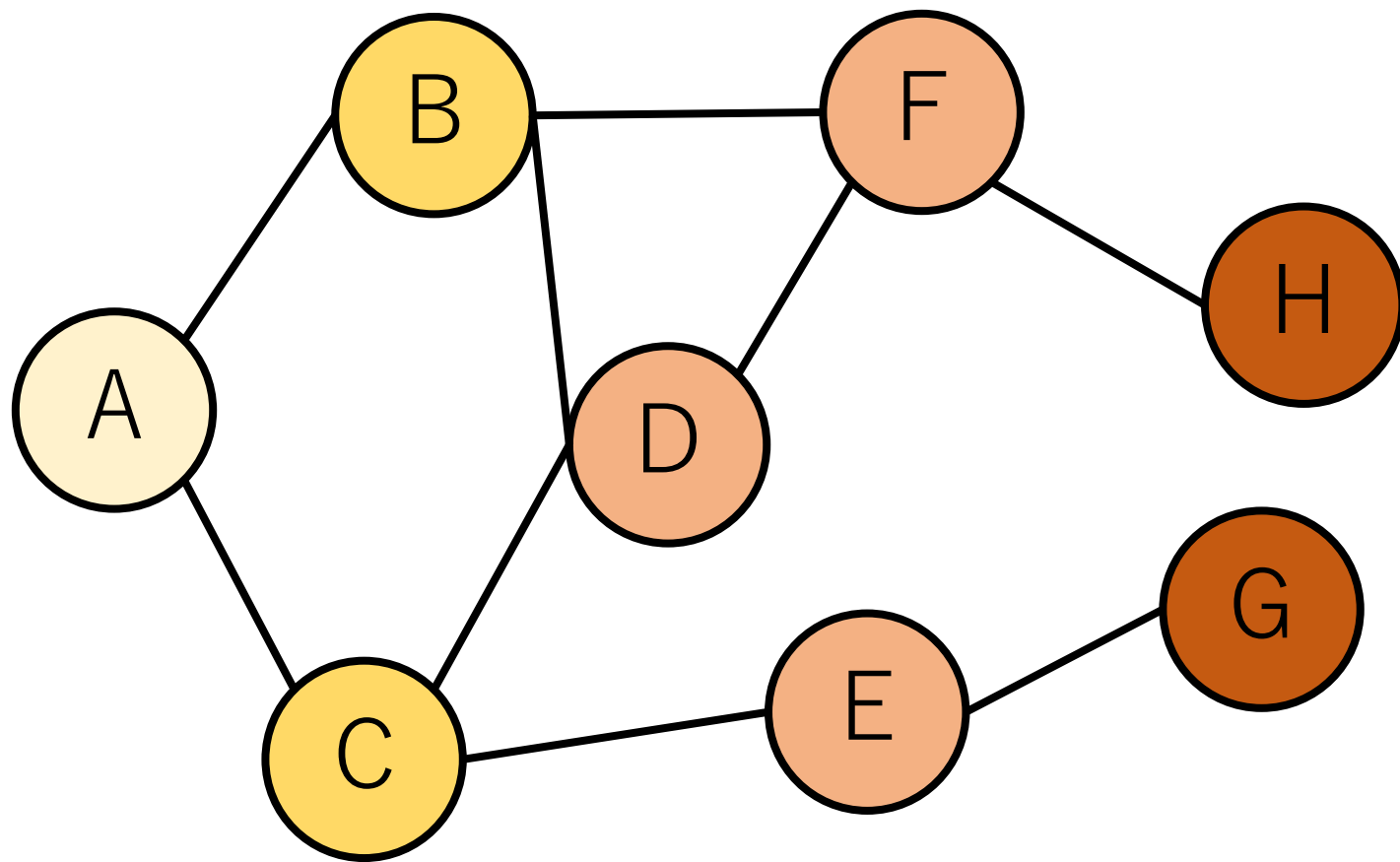
BFSの例

Eに移ってGを発見.



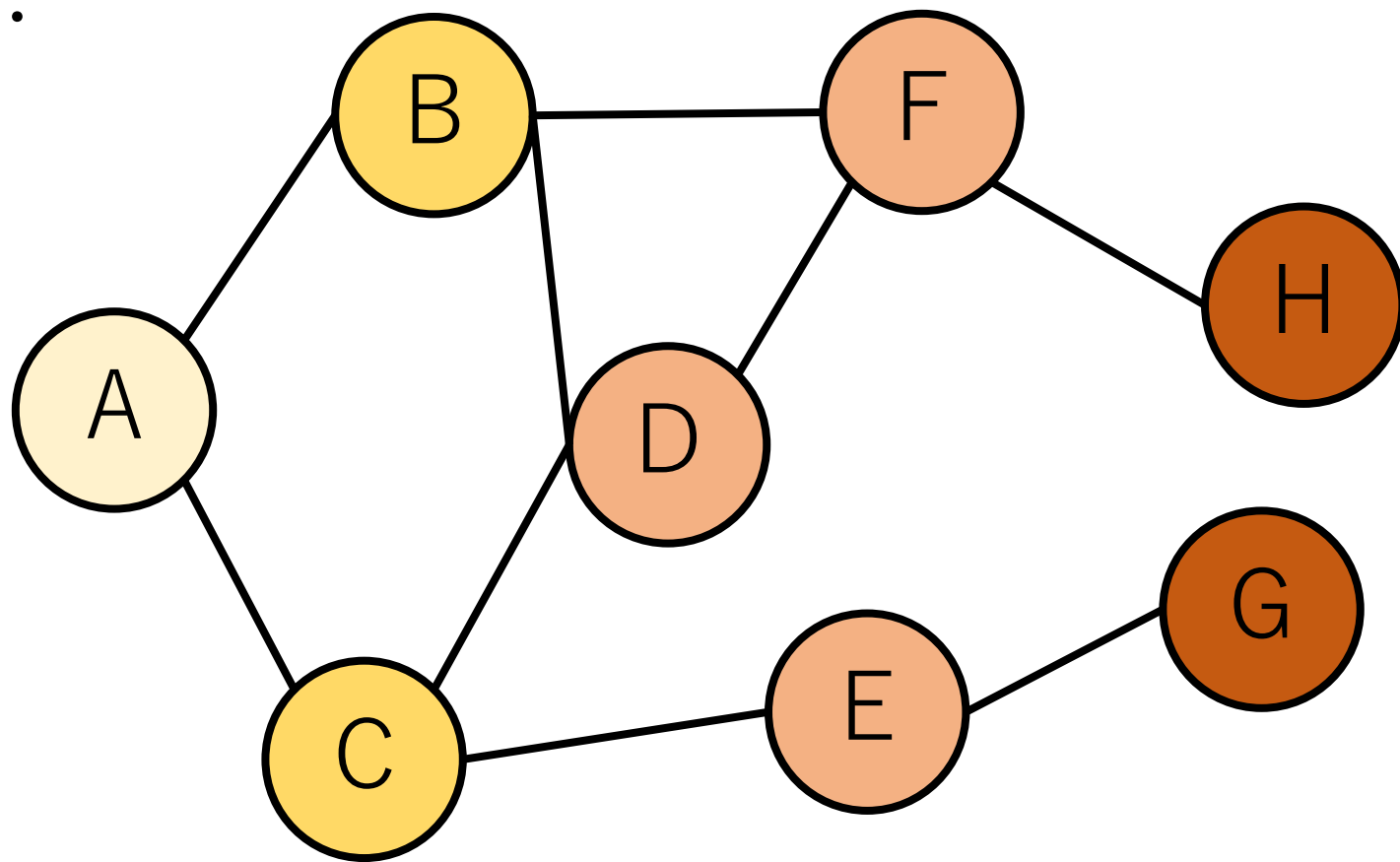
BFSの例

GとHの先には未発見の
ノードはない。



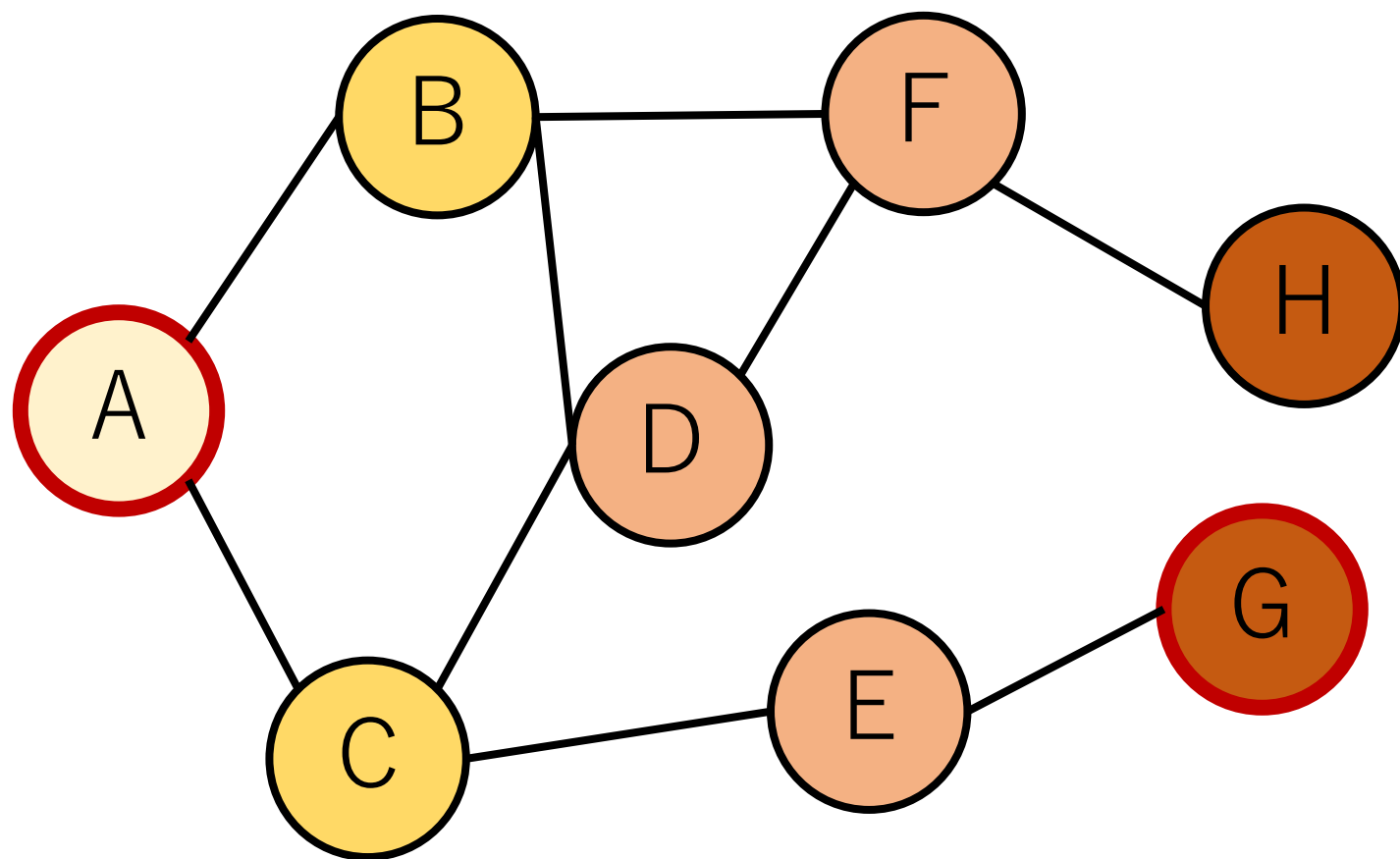
BFSの例

これで全部の探索が完了。



BFSの例

AからGへはつながっていることがわかった！



BFSの実装方針

キューを使って実装することがほとんど.

キューから取り出す:

取り出したノードに移動する.

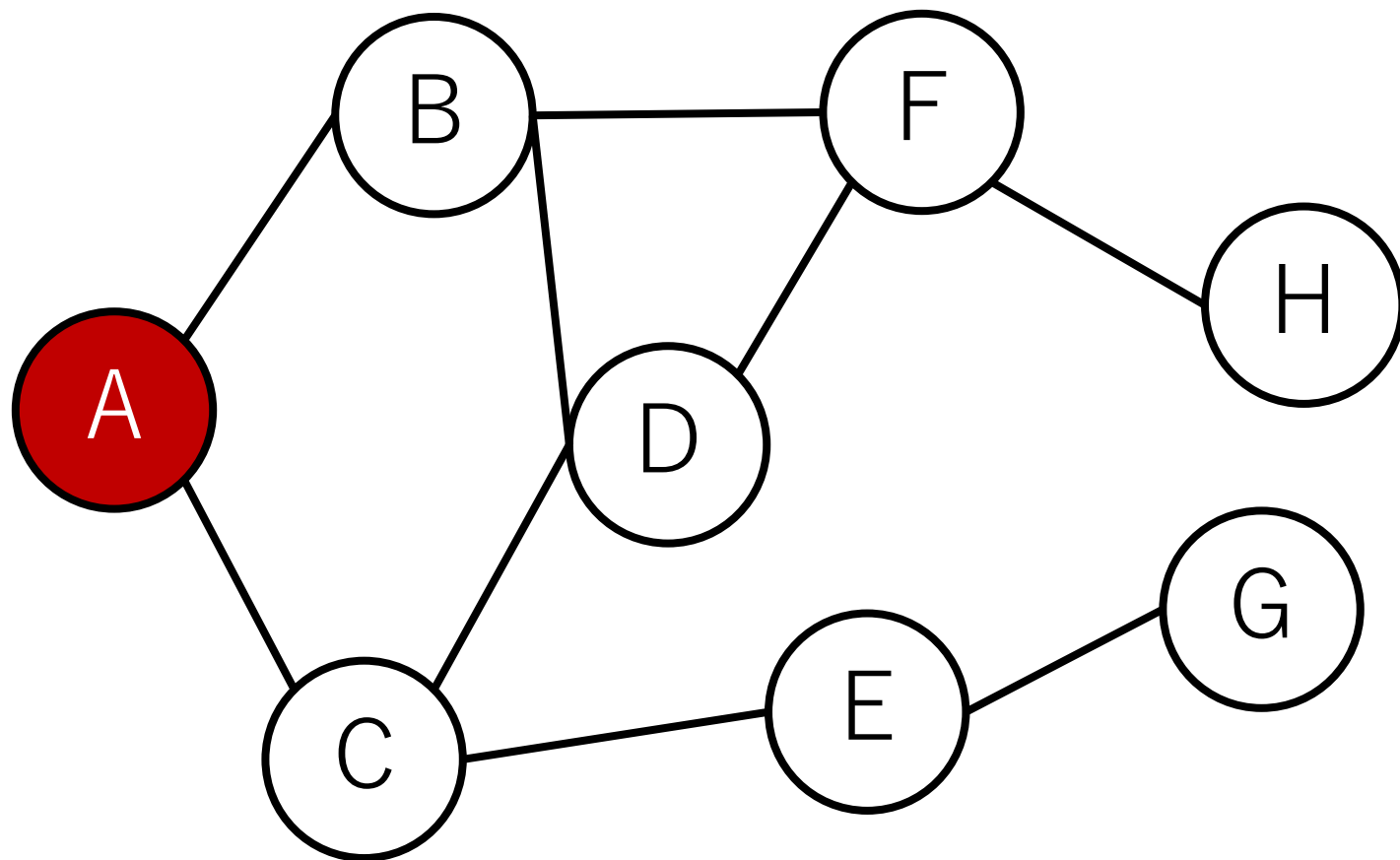
キューに入れる:

新しく見つかったノードを入れる.

BFSの例 & キューのダンプ

Aからスタート.

キューにAをいれて
初期化. キューの
最初からスタート.

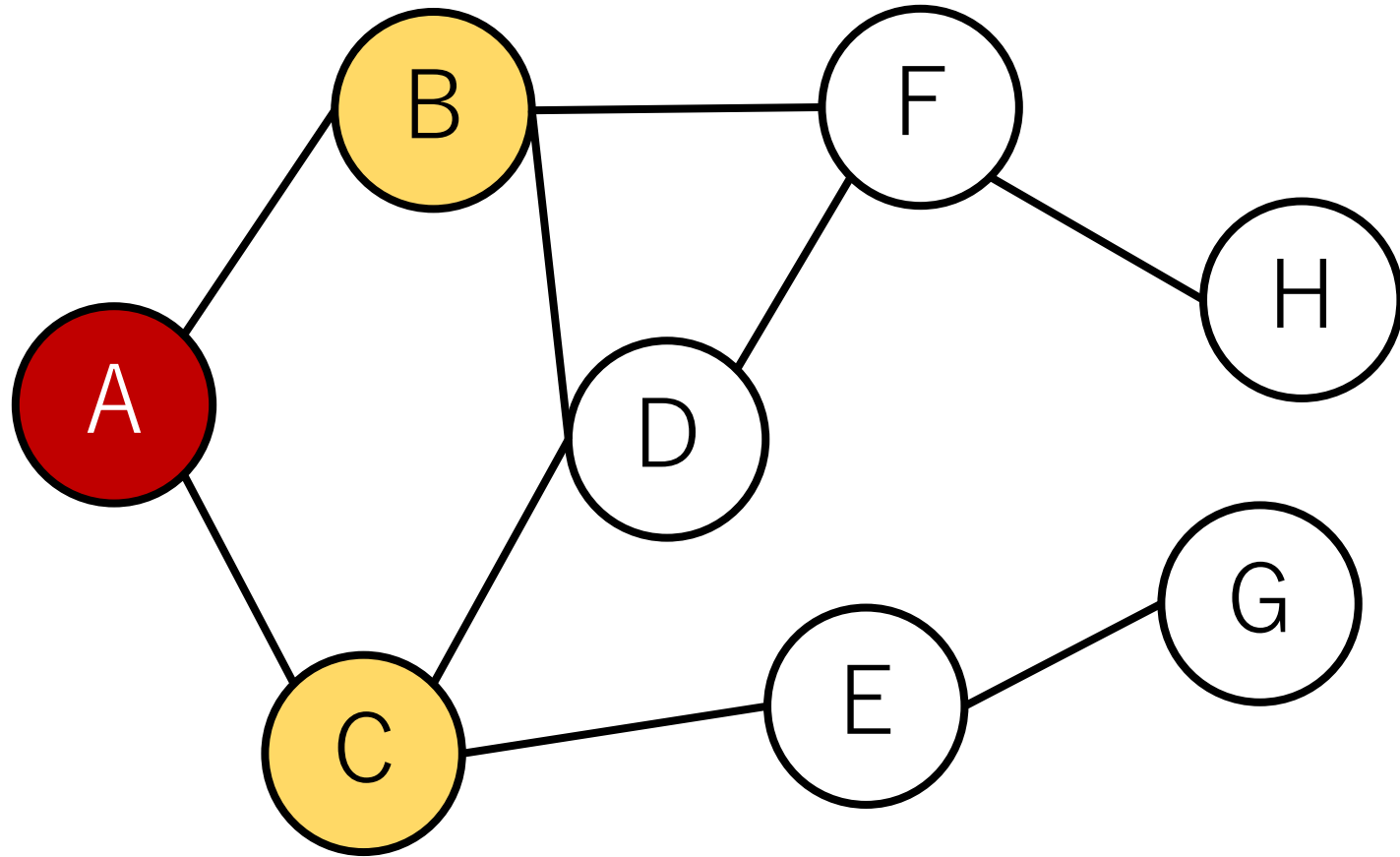


A

BFSの例 & キューのダンプ

1ステップでつながっているのは、BとC.

Aを取り出し、BとCをキューに入れる。

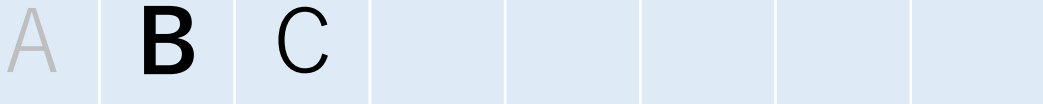
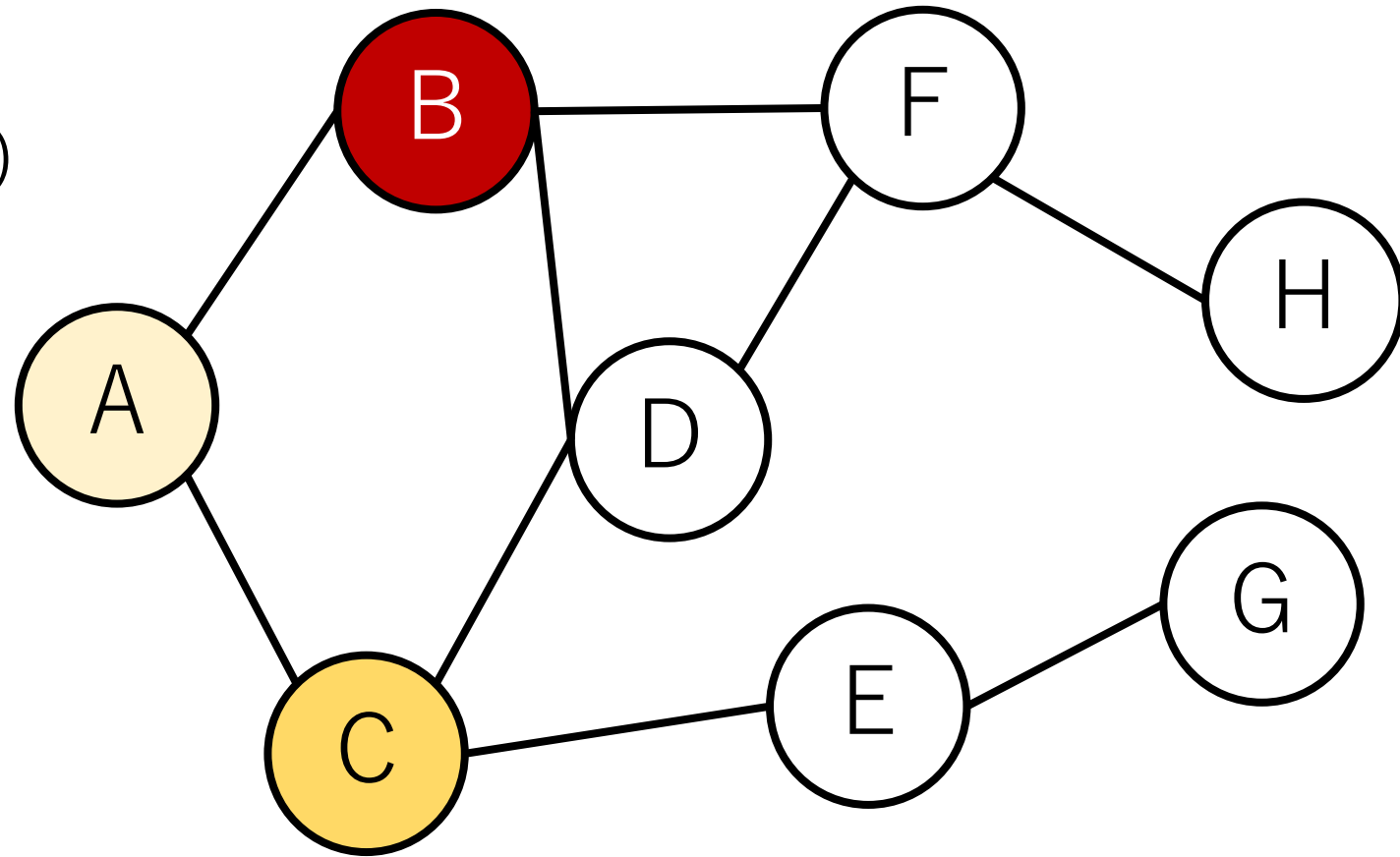


A	B	C						
---	---	---	--	--	--	--	--	--

BFSの例 & キューのダンプ

Bから1ステップでつながるノードを見る。
(Cから始めてもいい。)

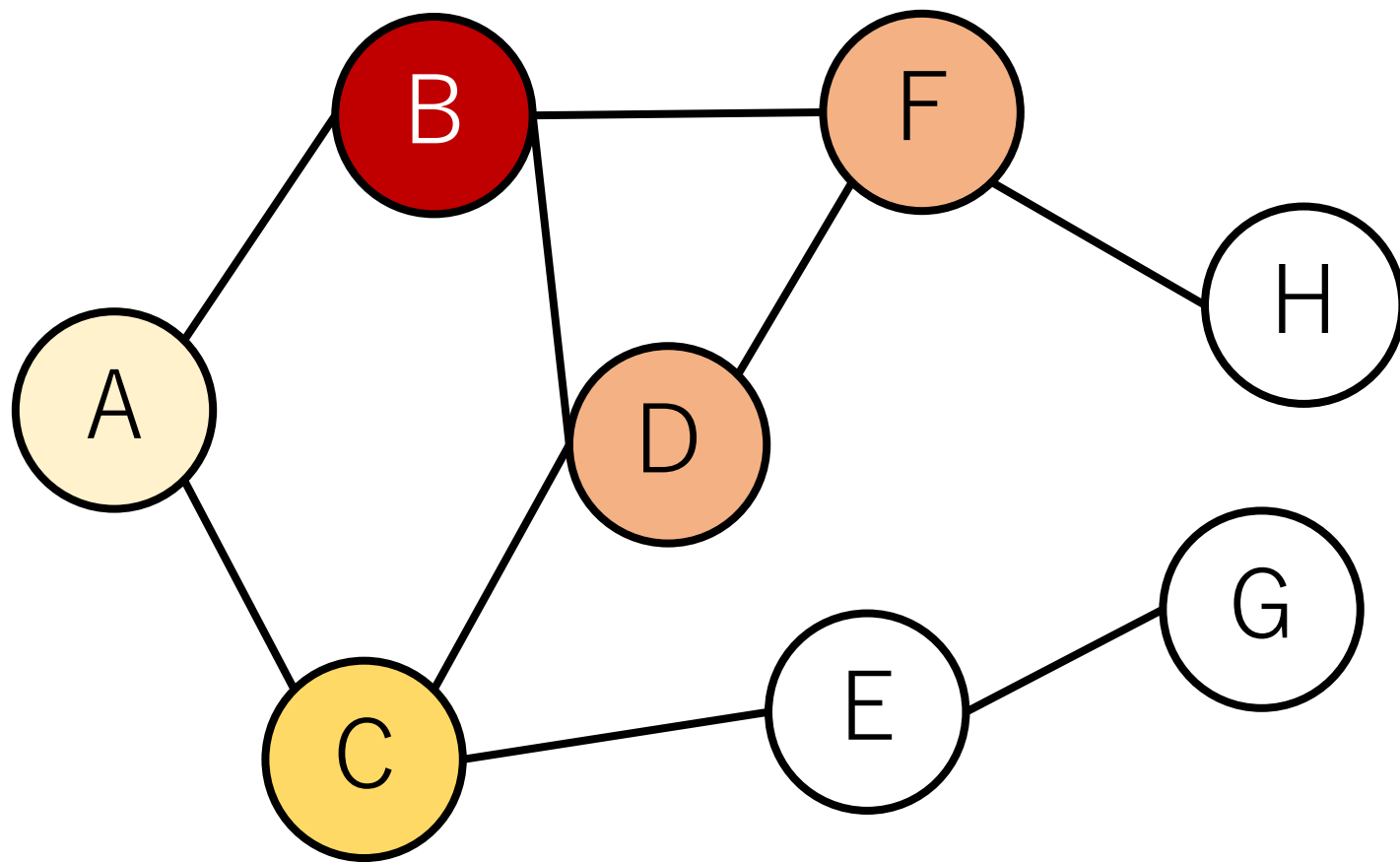
キューから取り出す。



BFSの例 & キューのダンプ

DとFを発見.

DとFをキューに入れる.

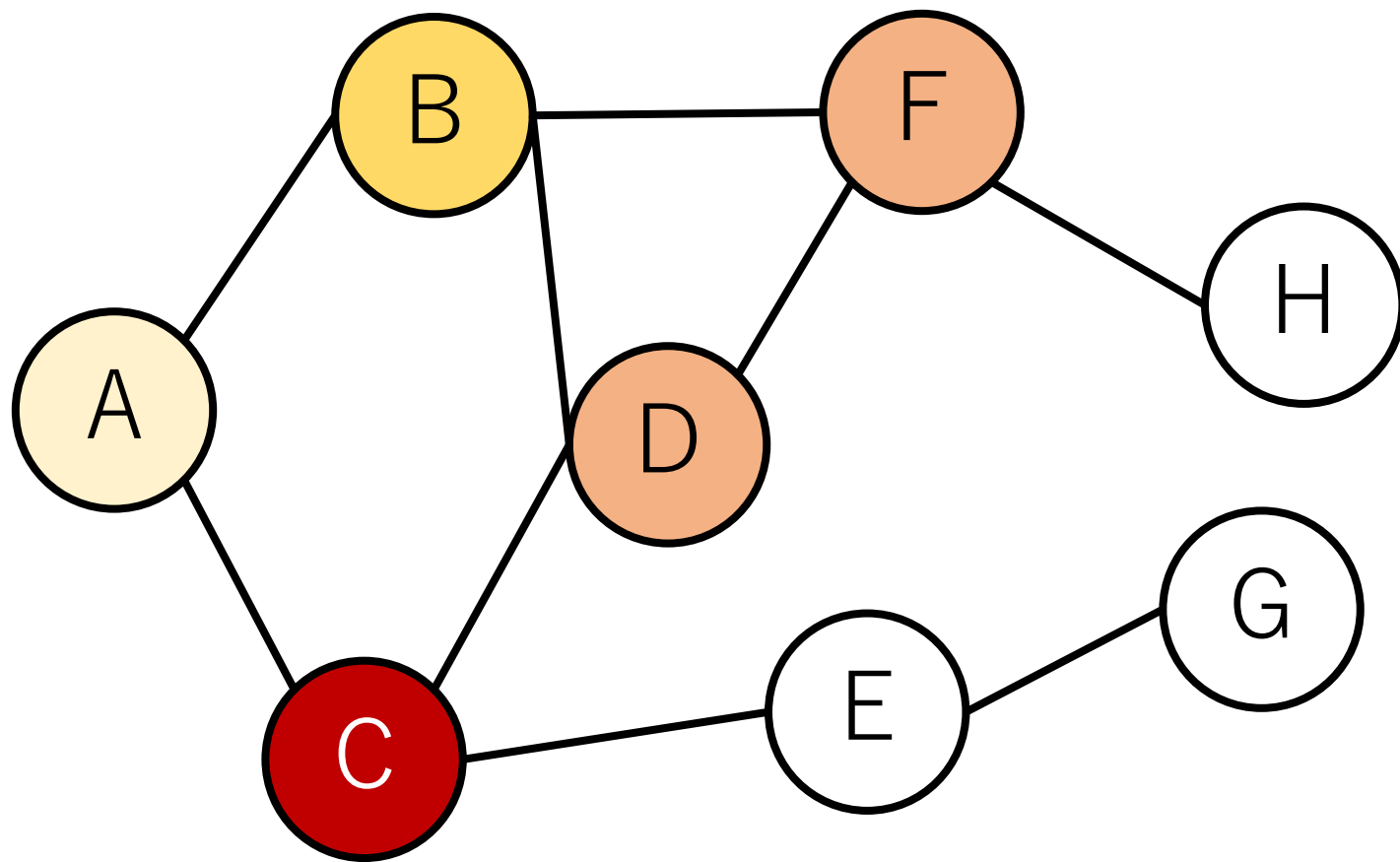


A	B	C	F	D			
---	---	---	---	---	--	--	--

BFSの例 & キューのダンプ

Cに移る.

キューから取り出す.

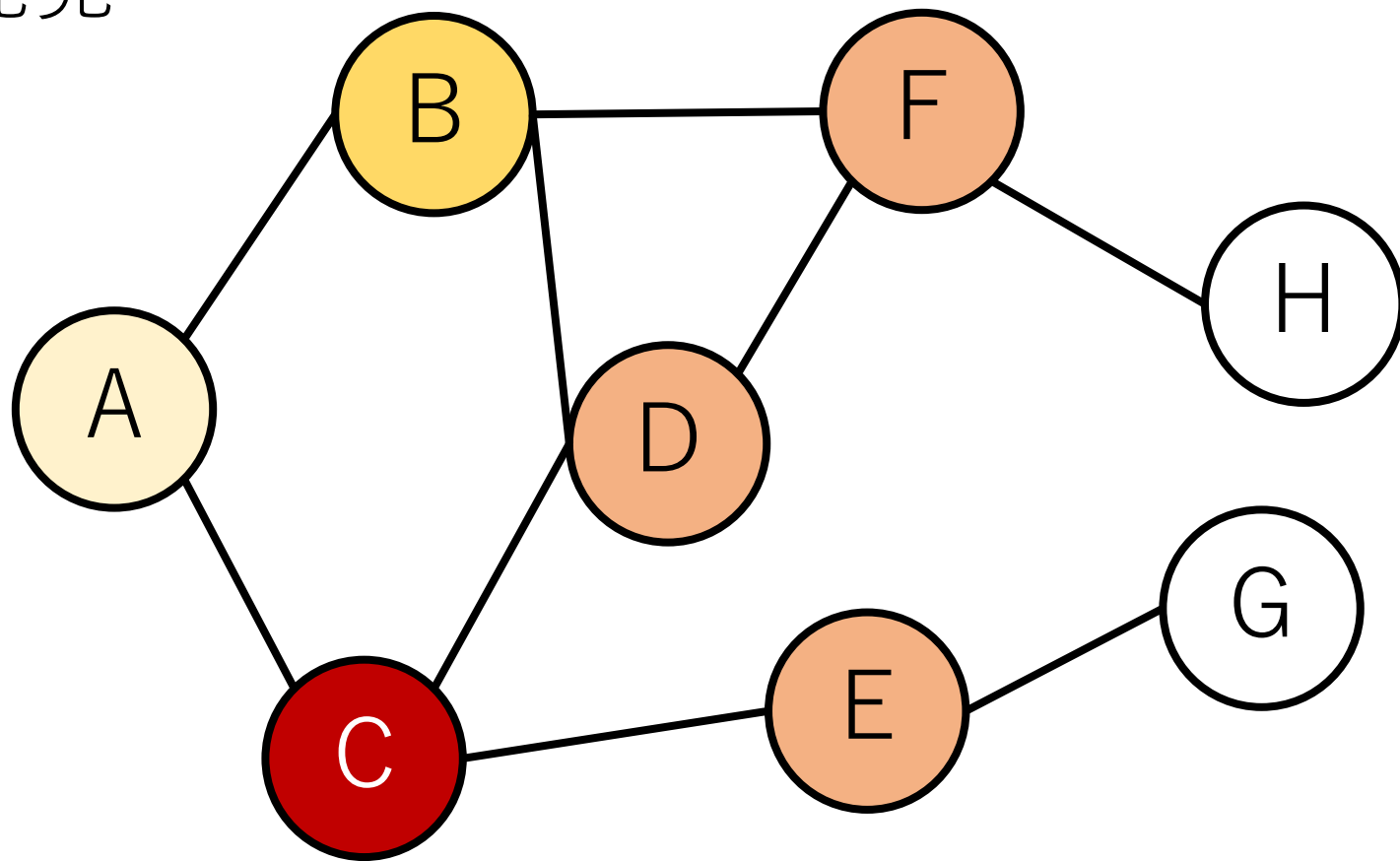


A	B	C	F	D			
---	---	----------	---	---	--	--	--

BFSの例 & キューのダンプ

Eを新しく発見. (Dは発見済み)

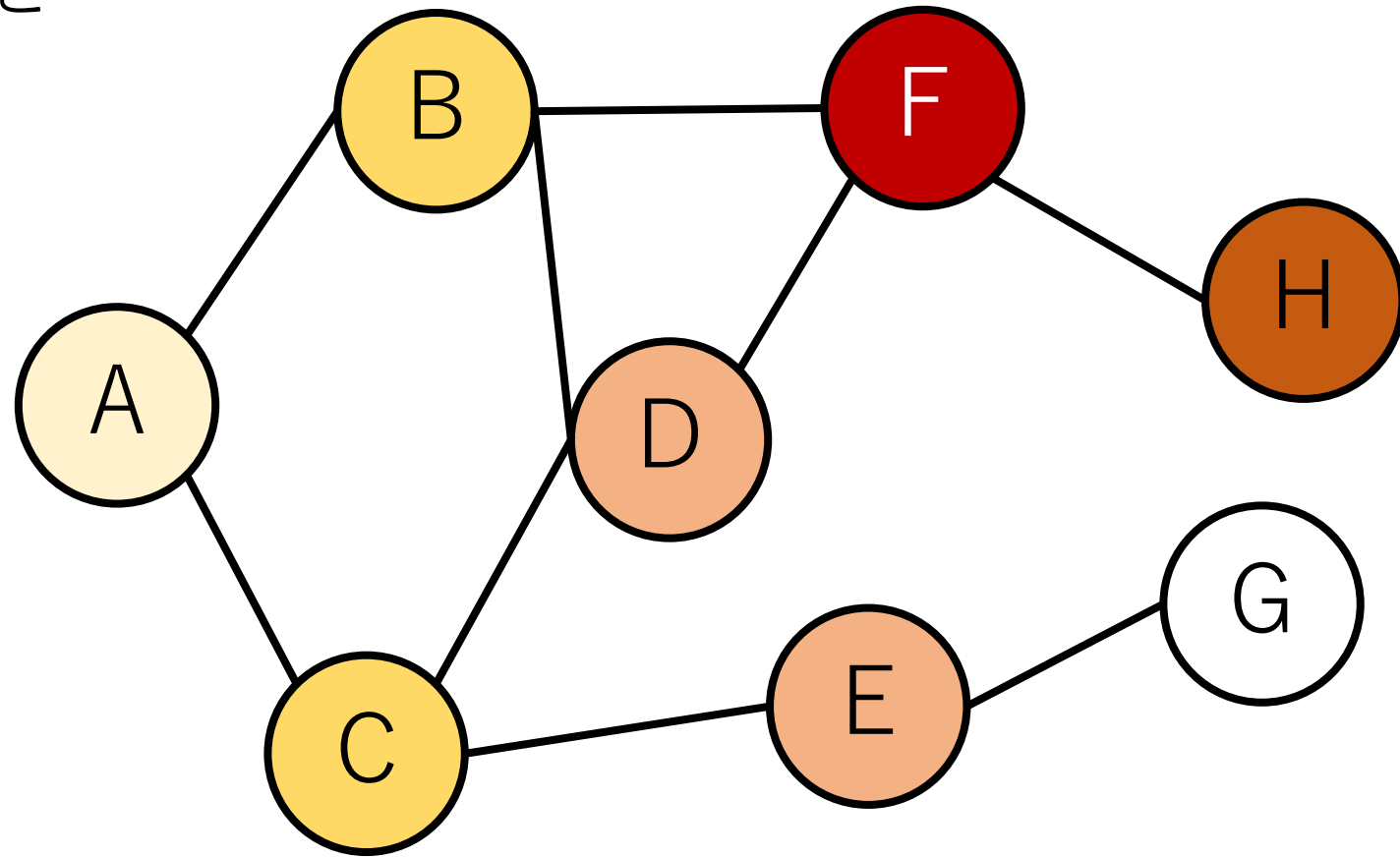
Eをキューに入れる.



BFSの例 & キューのダンプ

Fに移って，同様の探索を行い，Hを発見．

キューから取り出し，Hを入れる．

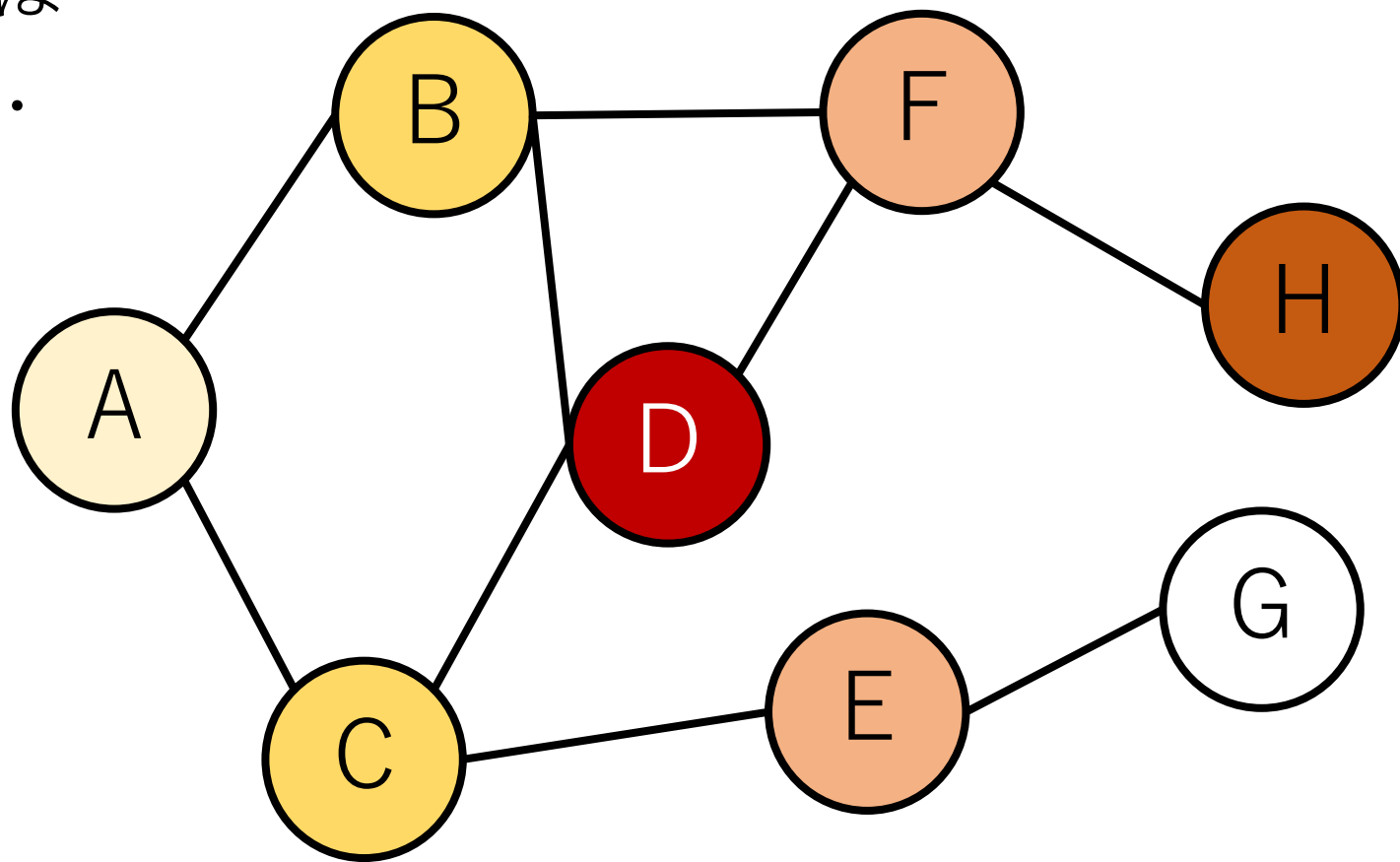


A	B	C	F	D	E	H	
---	---	---	----------	---	---	---	--

BFSの例 & キューのダンプ

Dからは未発見のノードはないので、何も行わない。

キューから取り出す。

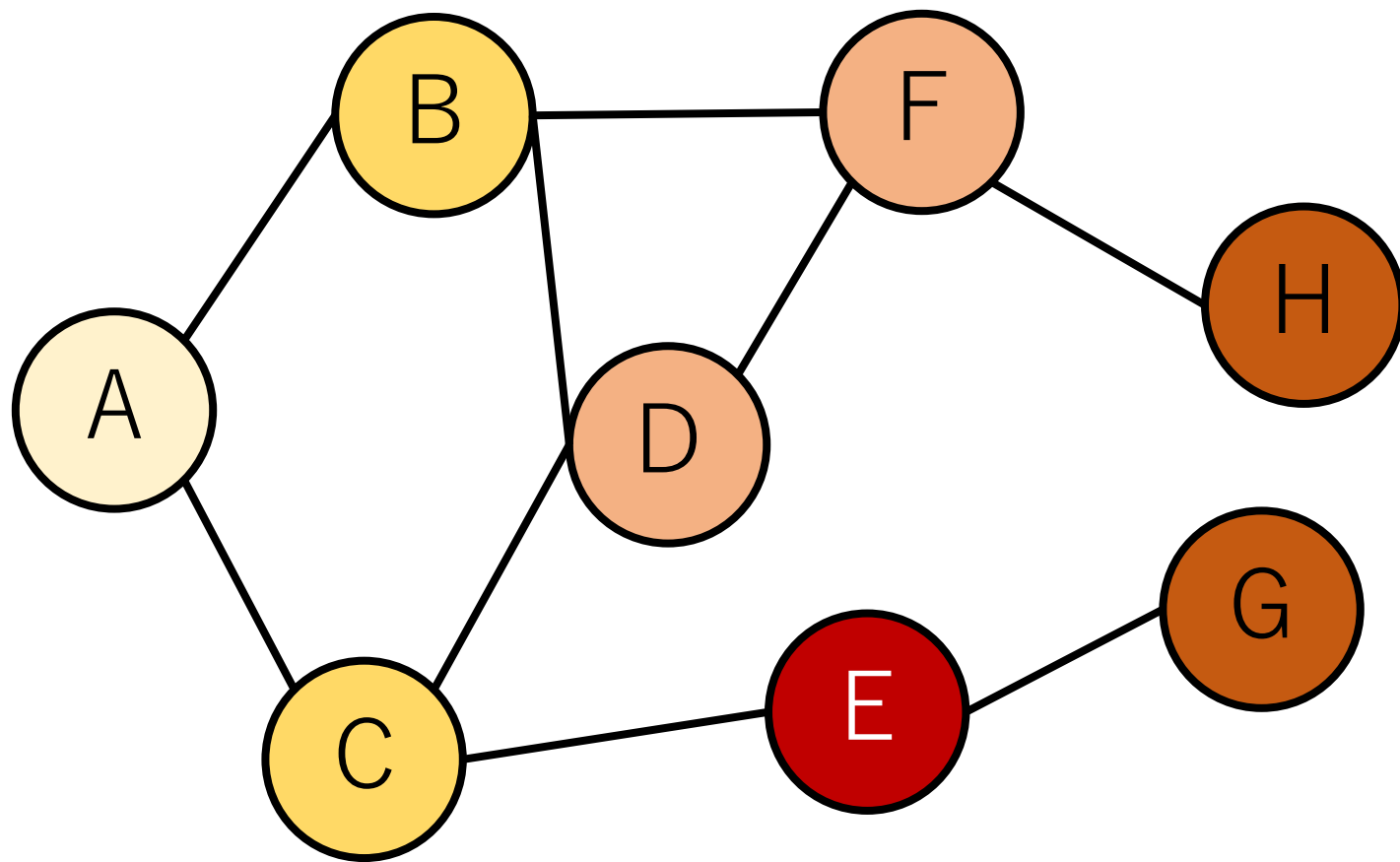


A	B	C	F	D	E	H	
---	---	---	---	----------	---	---	--

BFSの例 & キューのダンプ

Eに移ってGを発見.

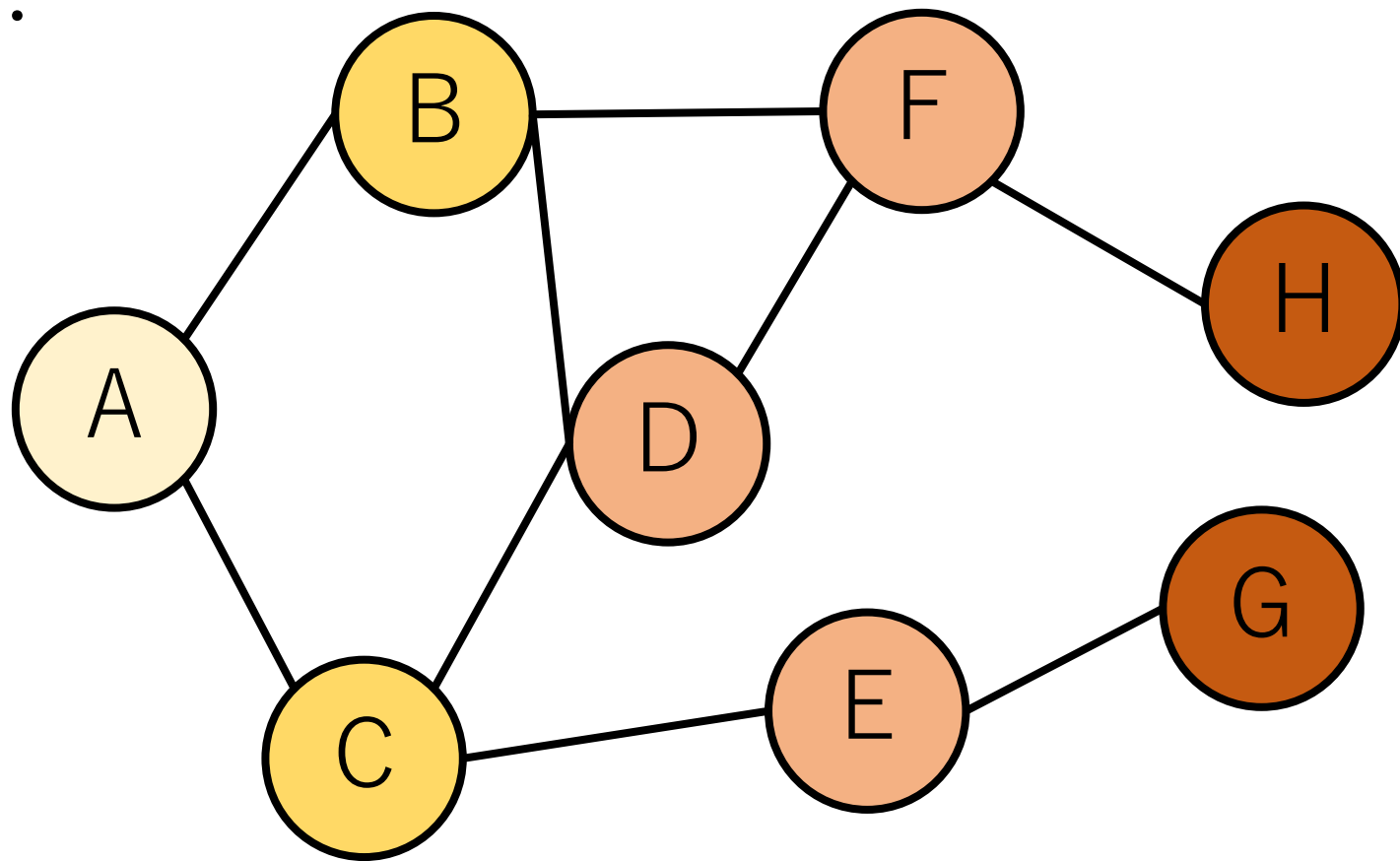
キューから取り出し,
Gを入れる.



A	B	C	F	D	E	H	G
---	---	---	---	---	----------	---	---

BFSの例 & キューのダンプ

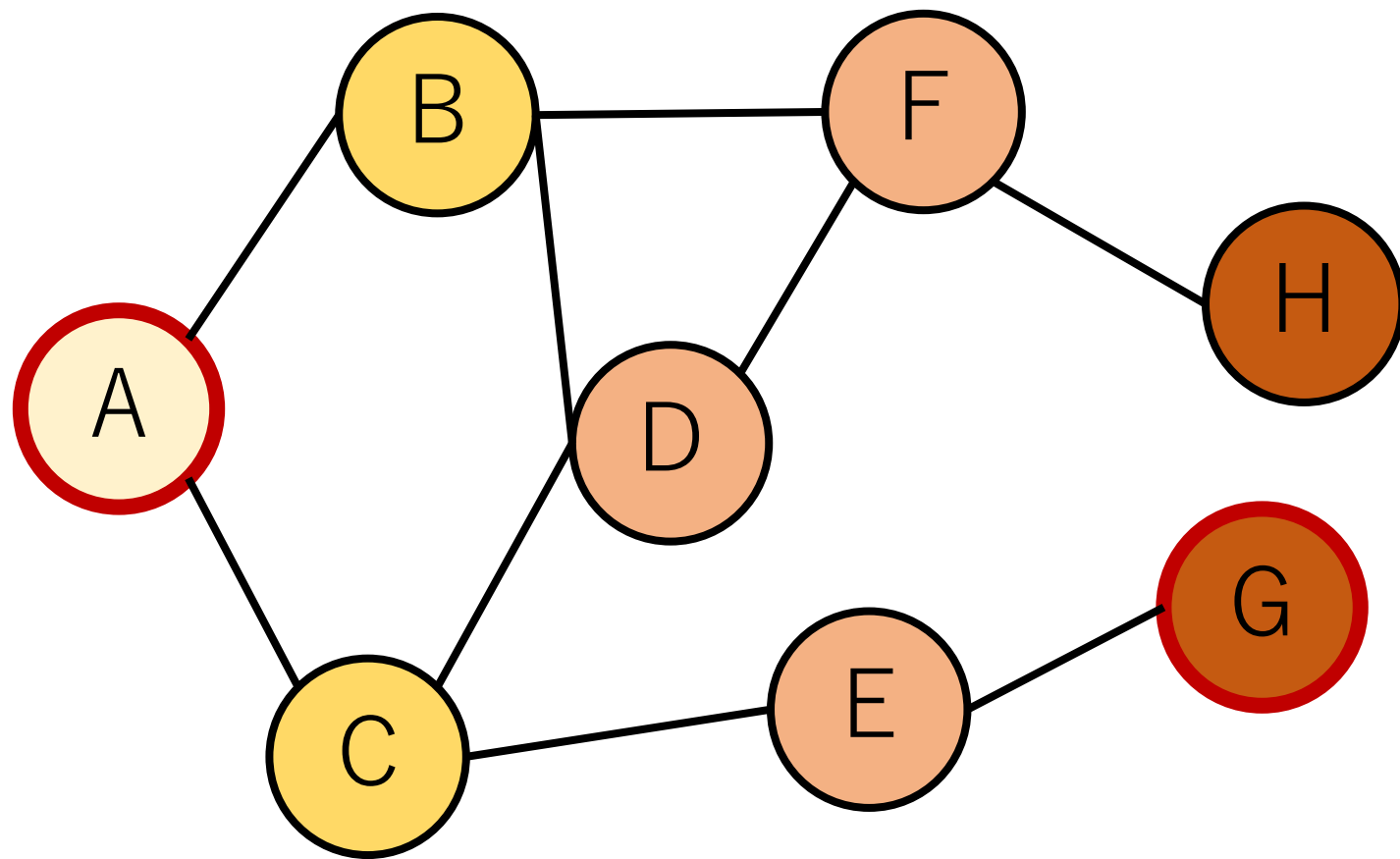
これで全部の探索が完了。



A	B	C	F	D	E	H	G
---	---	---	---	---	---	---	---

BFSの例 & キューのダンプ

AからGへはつながっていることがわかった！



A B C F D E H G

DFS

幅優先探索 (Breadth first search, BFS)

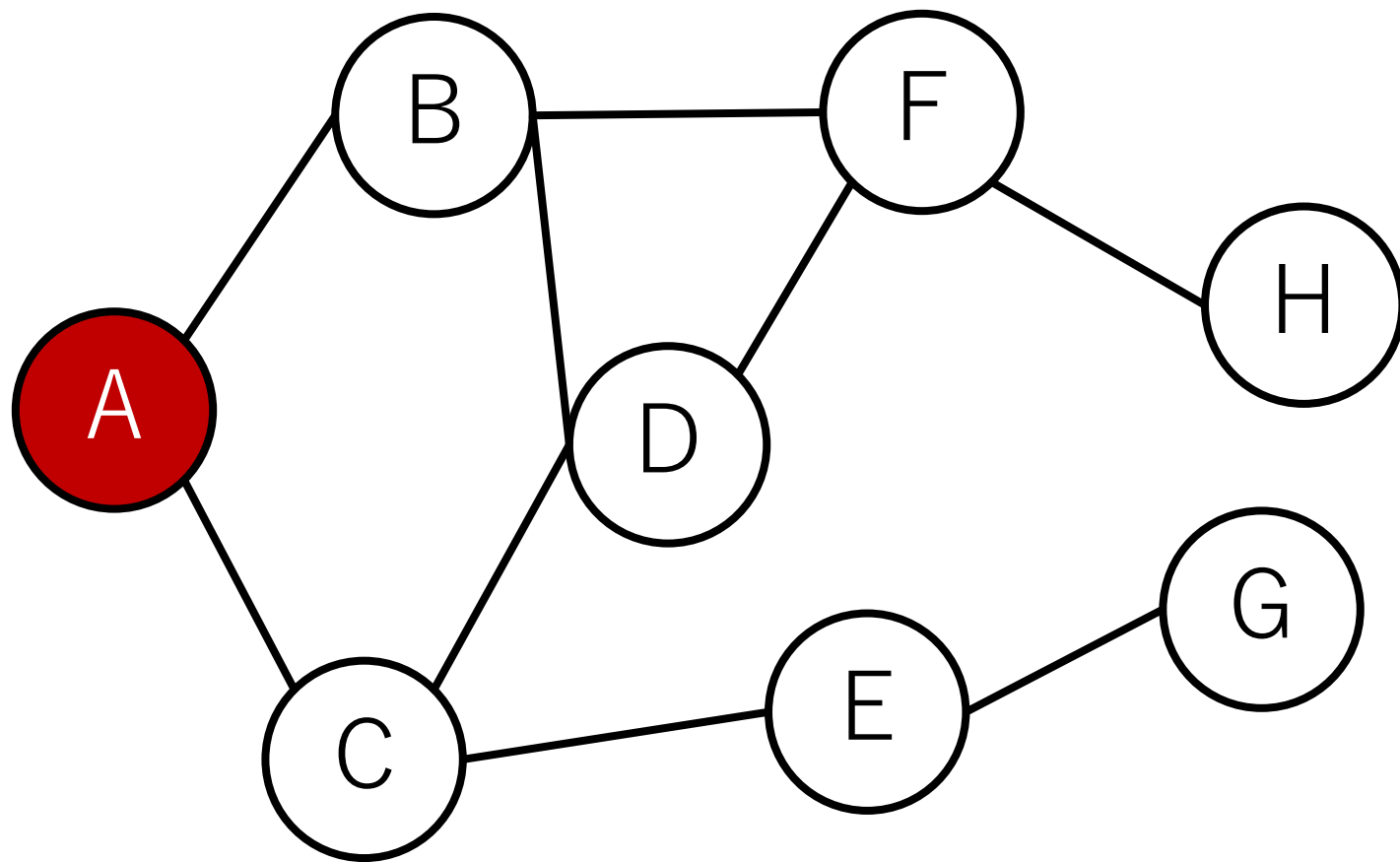
後戻りしないように、可能性のあるルート全てにおいて1ステップずつ行くパターン。

深さ優先探索 (Depth first search, DFS)

とりあえず行けるところまで行き、ダメなら後戻りするパターン。

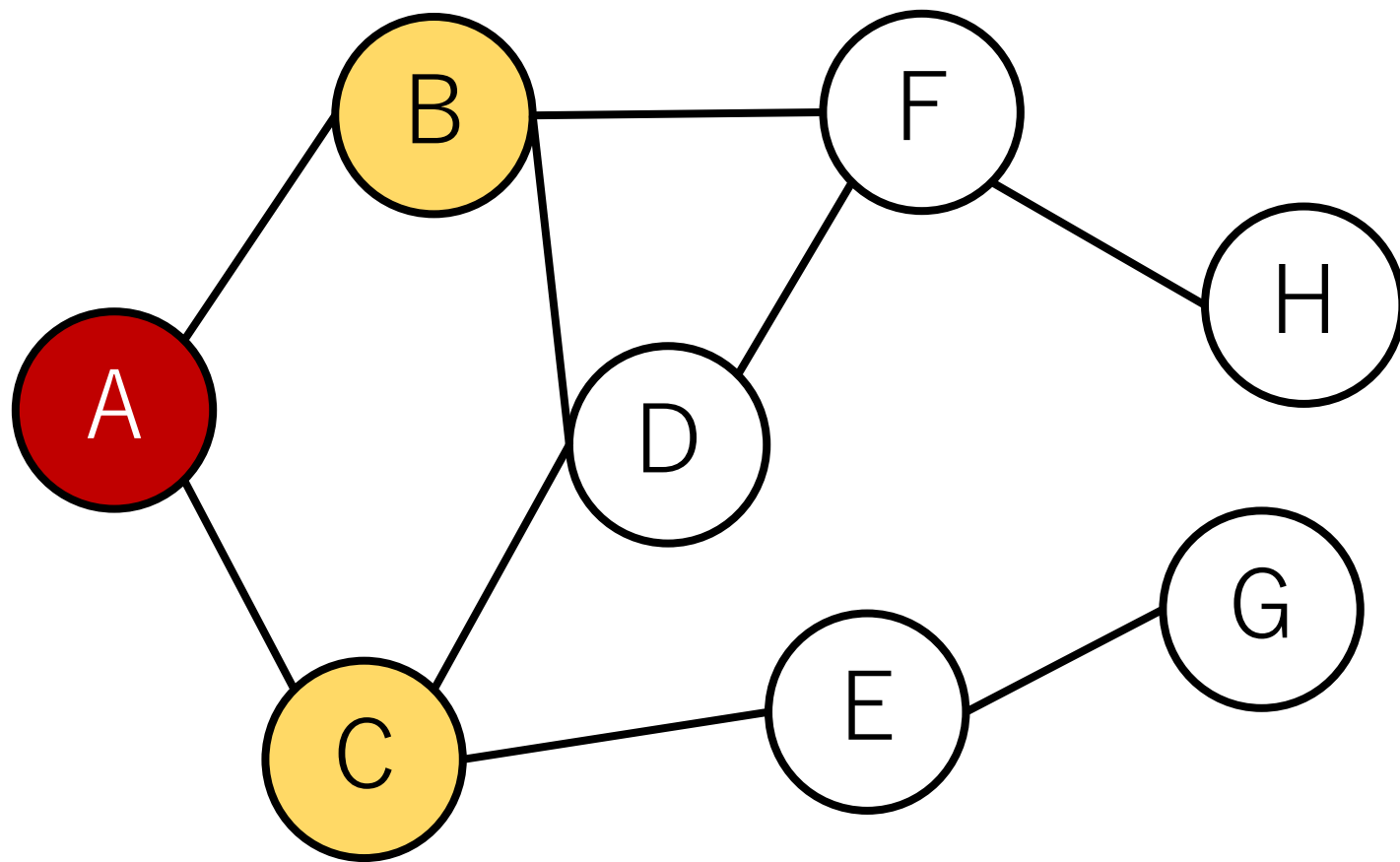
DFSの例

Aからスタート.



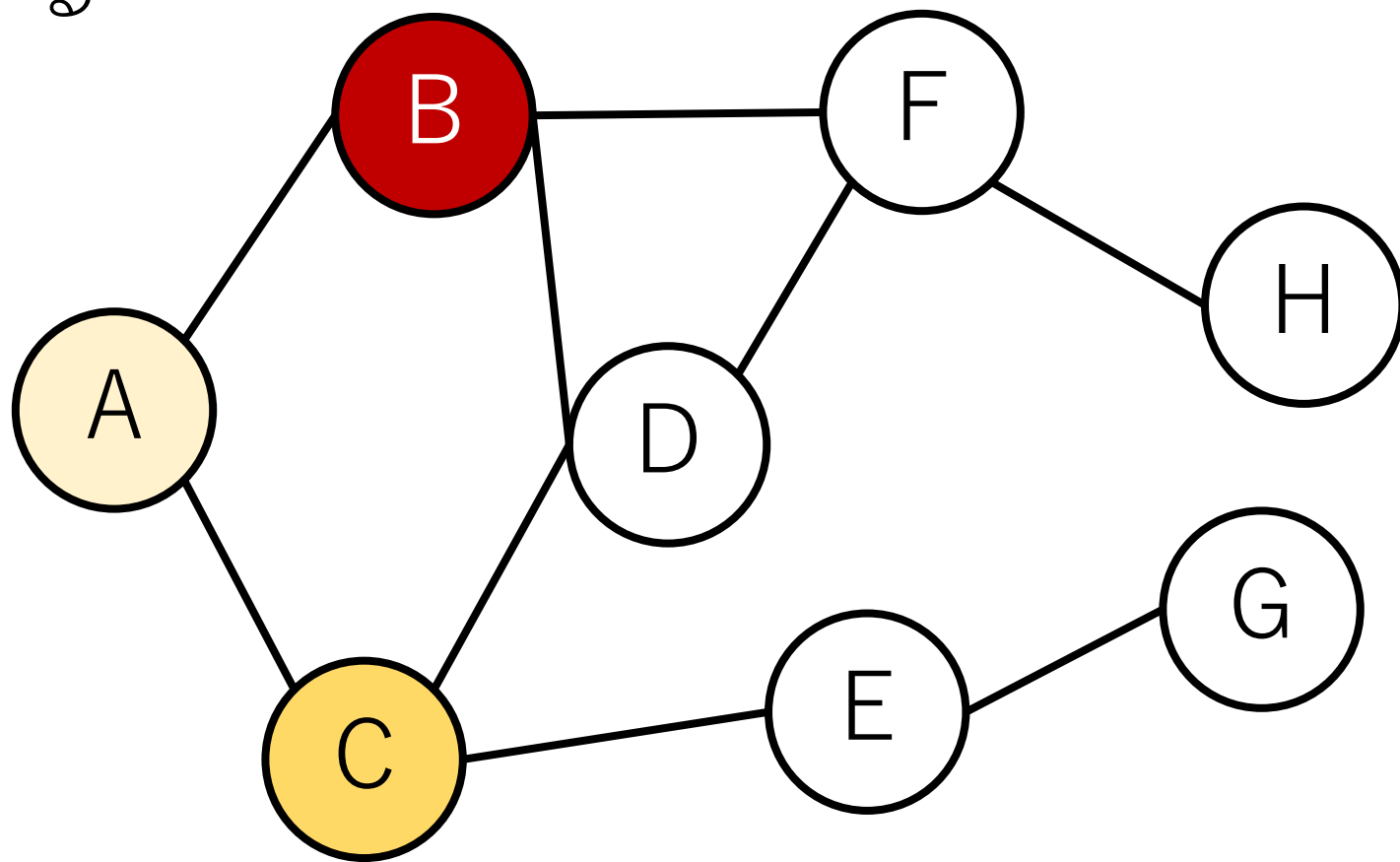
DFSの例

1ステップでつながっているのは, BとC.



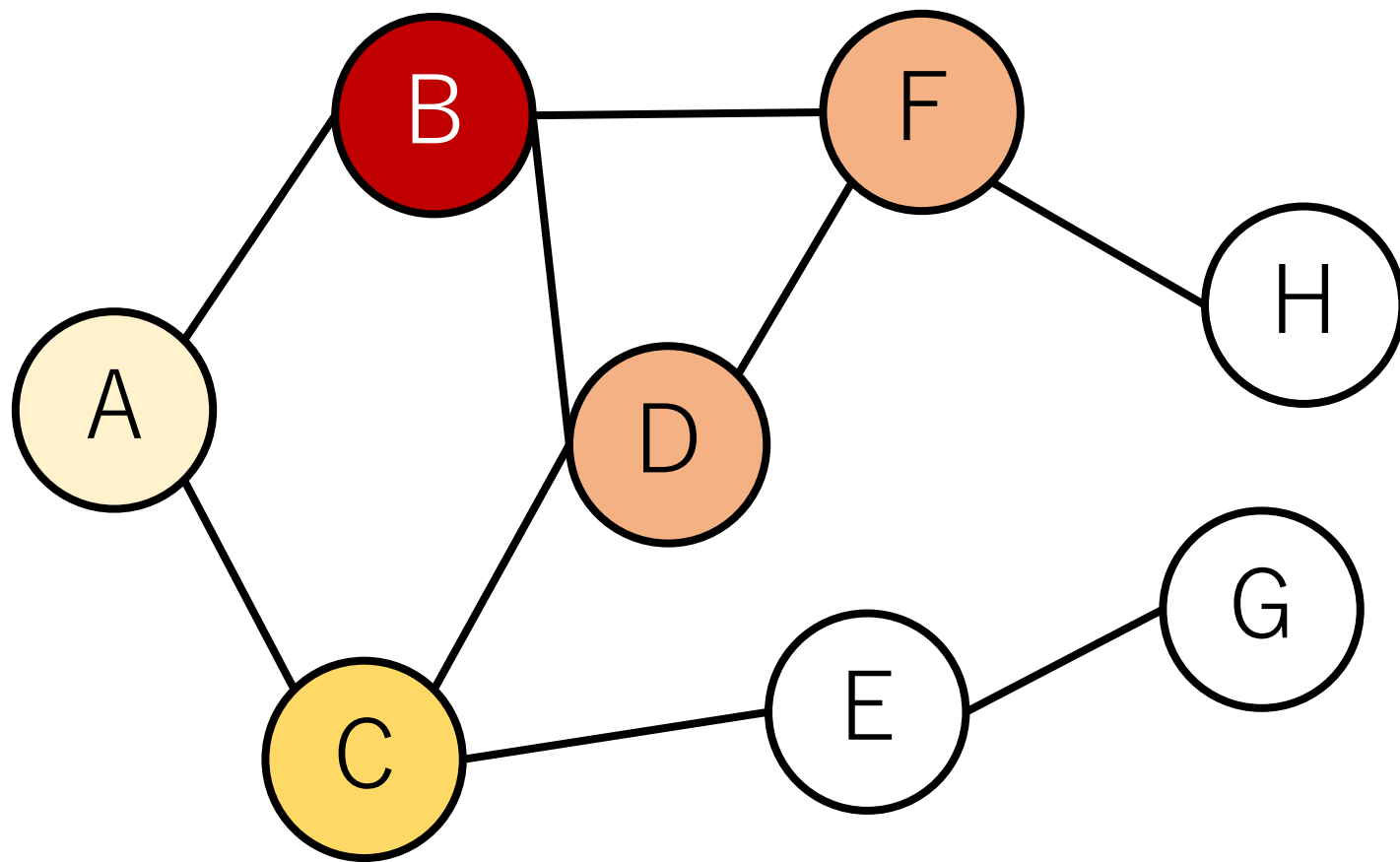
DFSの例

Bから1ステップでつながる
ノードを見る。(Cから
始めてもよい。)



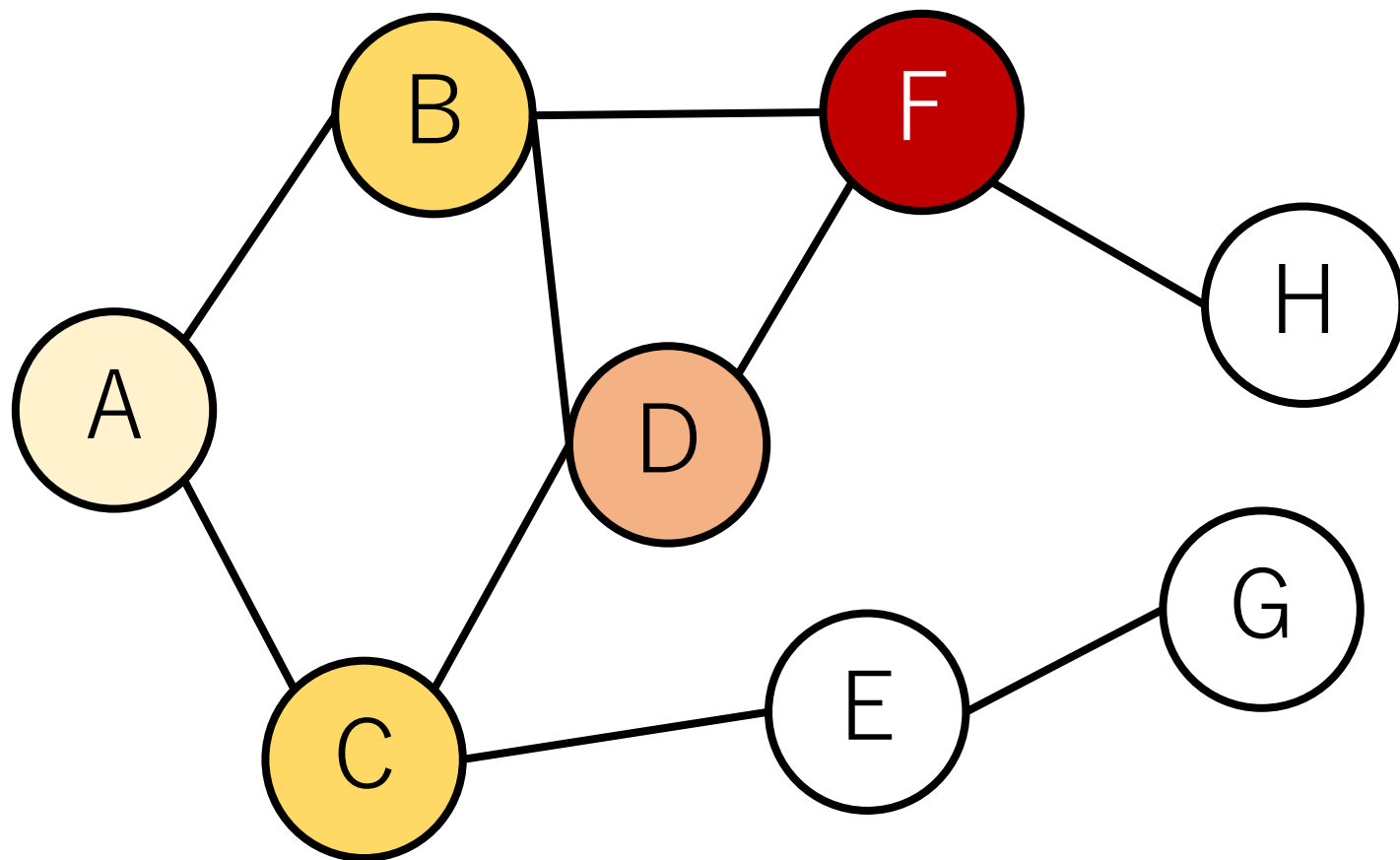
DFSの例

DとFを発見.



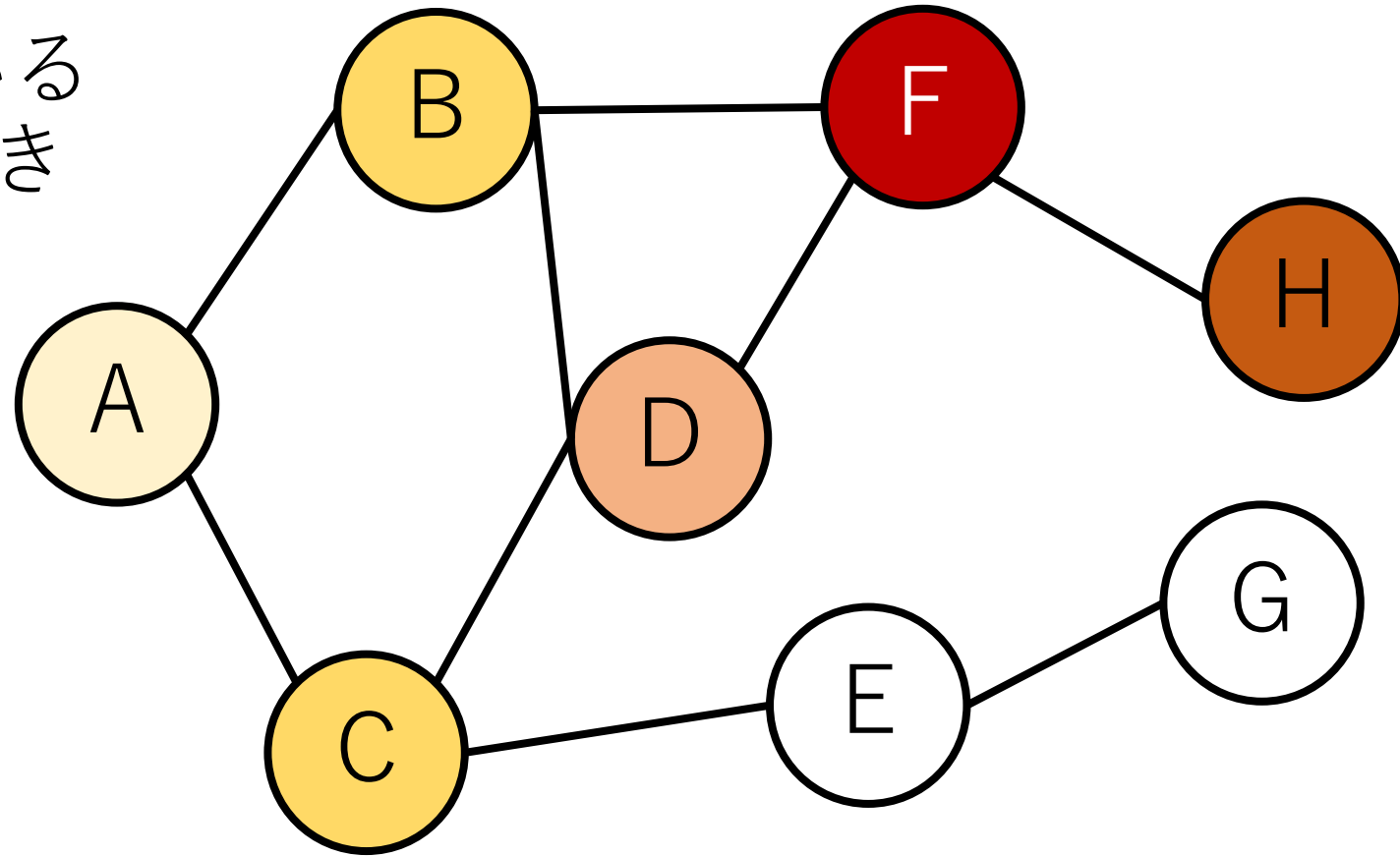
DFSの例

Fに移る.



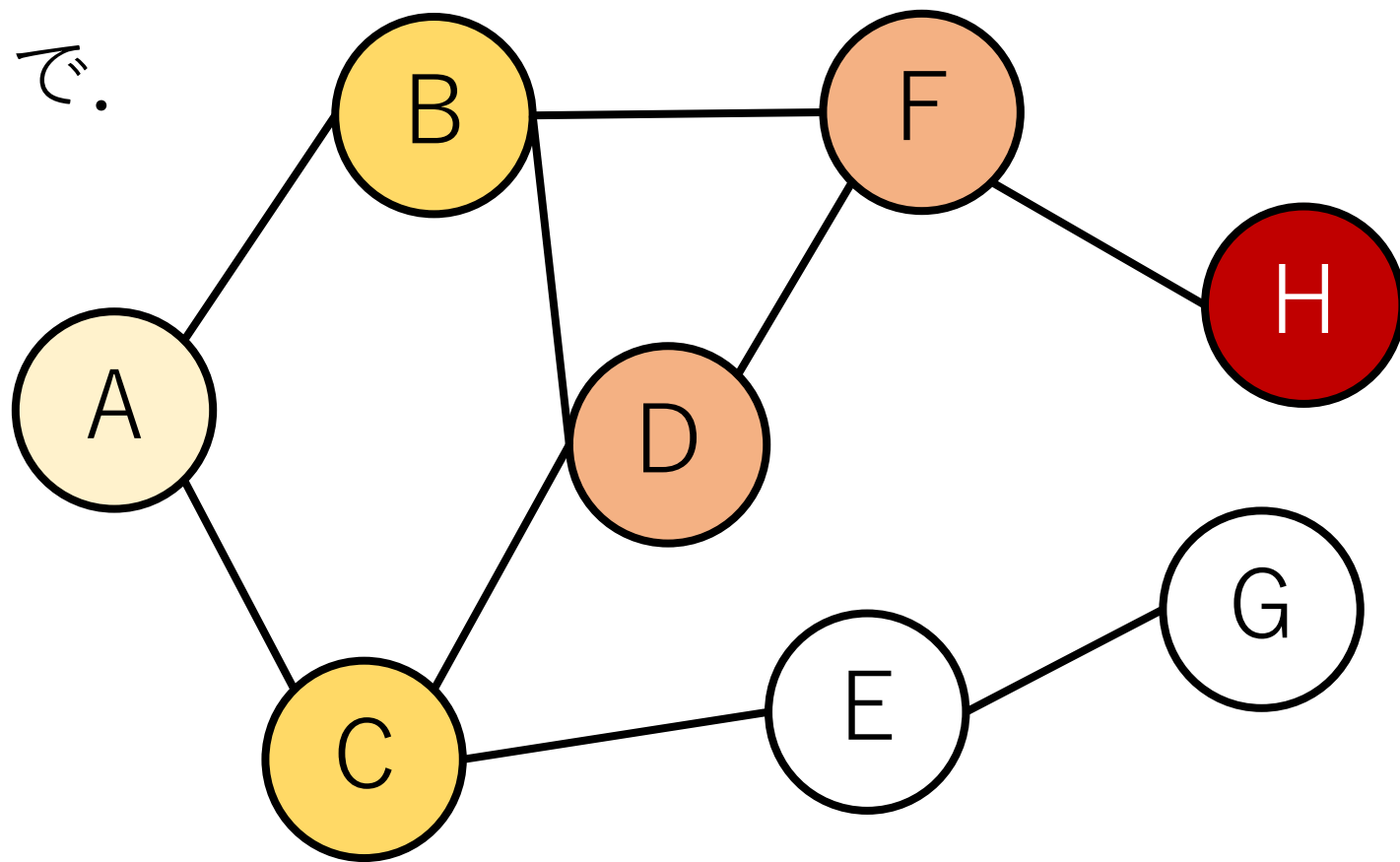
DFSの例

DとHを発見. DはBから見たときに発見はしているので, Hのみを訪れるべき候補として追加する.



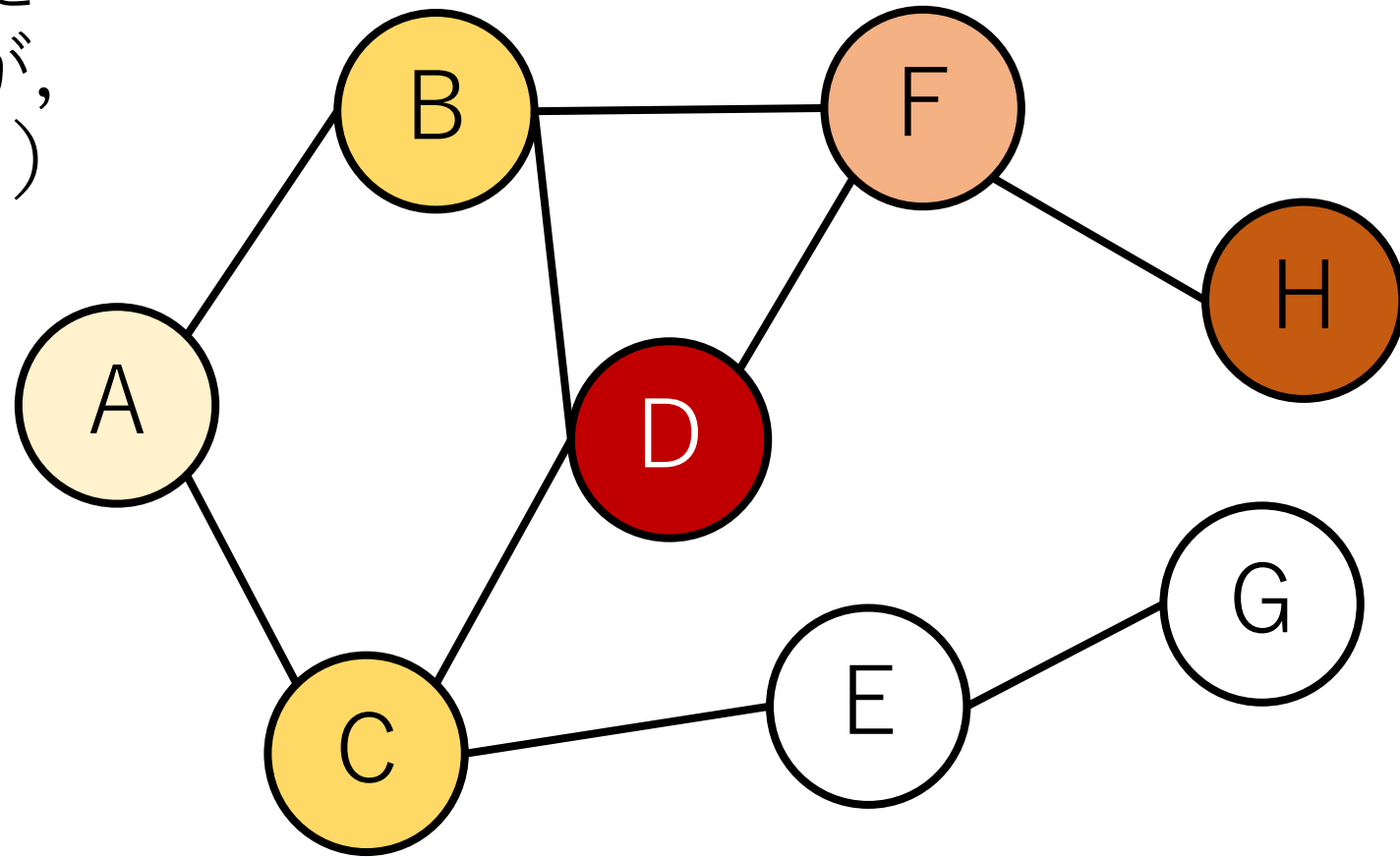
DFSの例

Hに移るが、その先は存在しないので、ここまで。



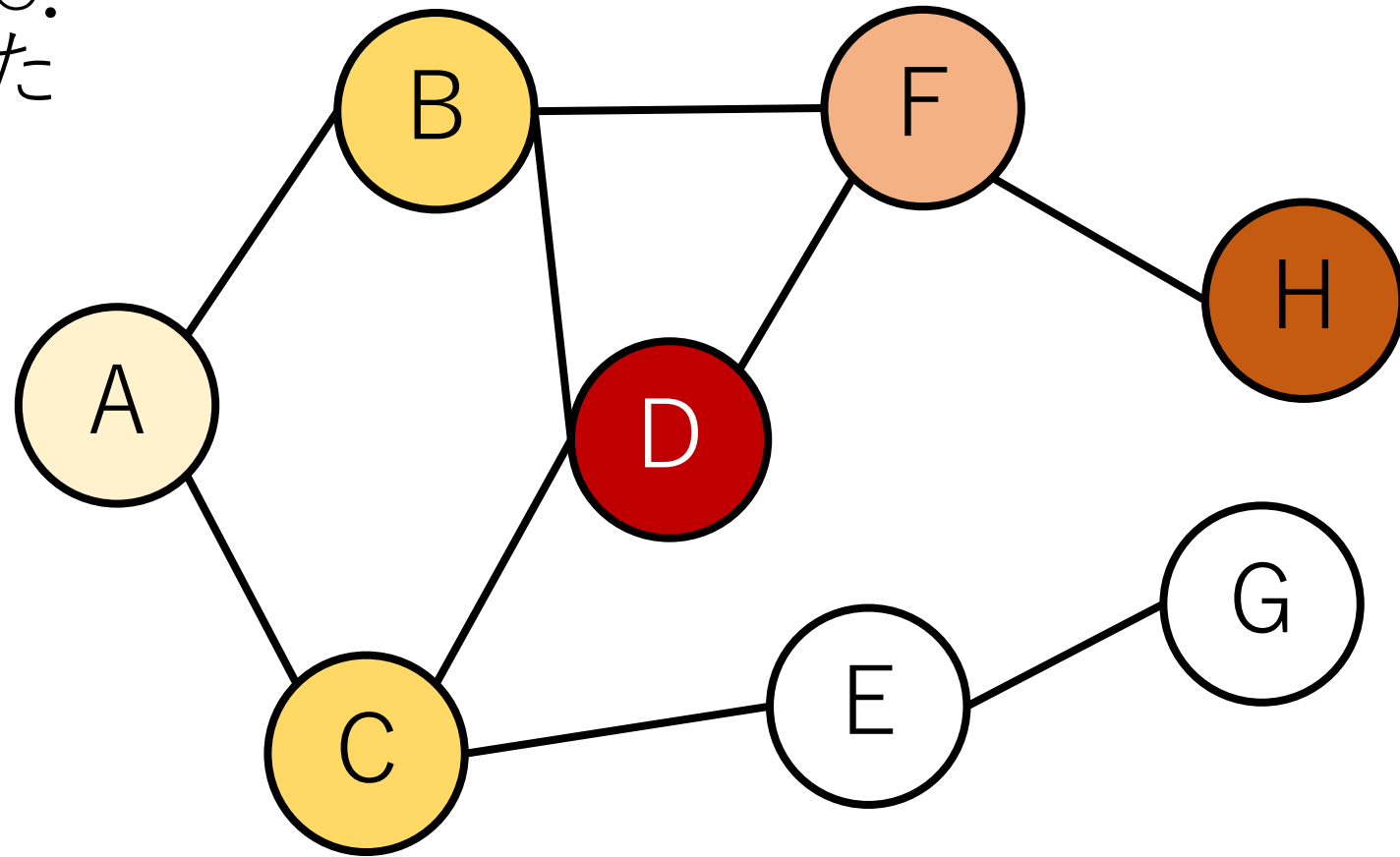
DFSの例

Bまで後戻り，今度はDに移動する．（発見済だが，訪問はまだしていない．）



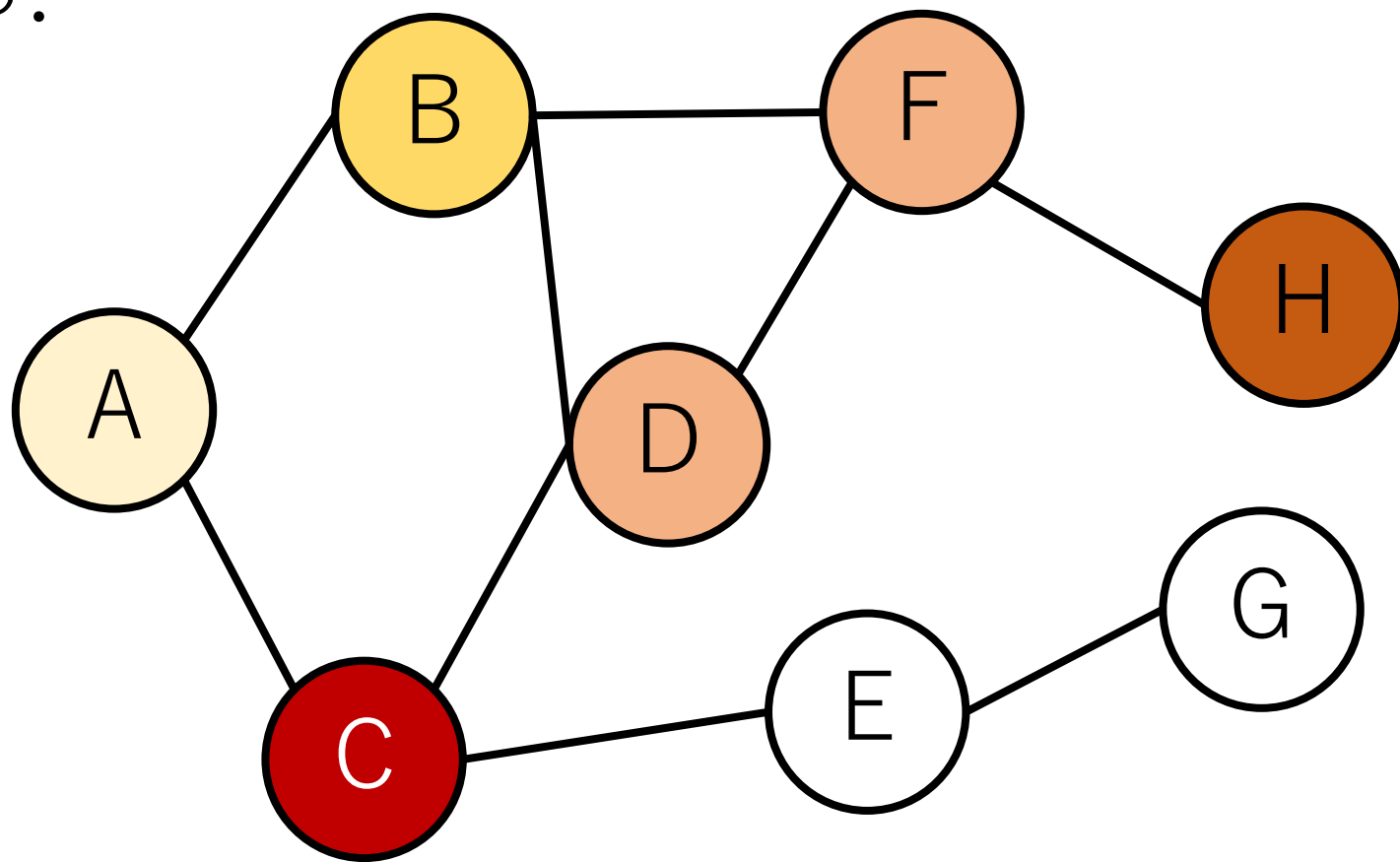
DFSの例

Dから未訪問のノードはC.
ただし、これもAから見た
ときに発見済なので、D
からは移動しない。



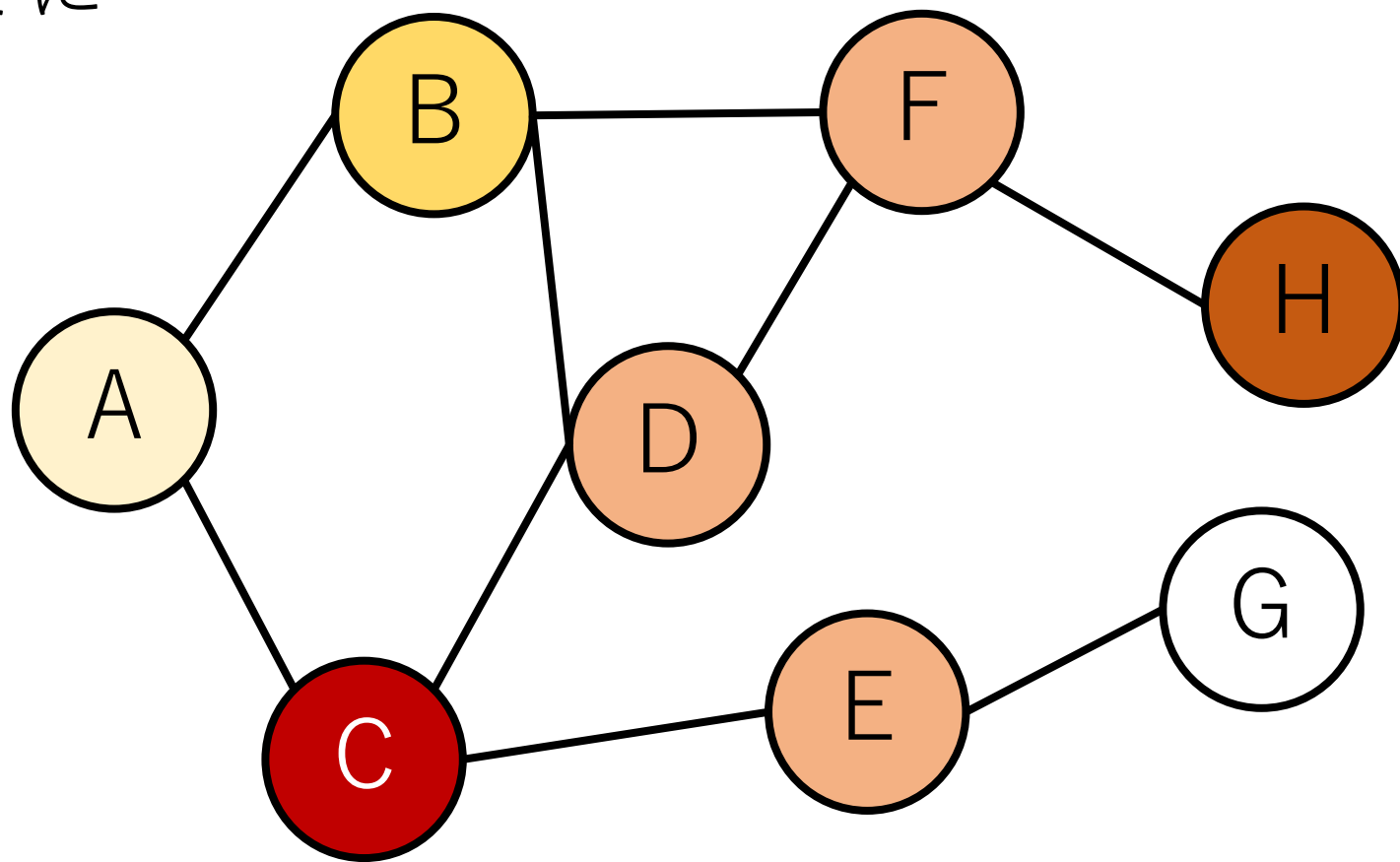
DFSの例

Aに戻って, Cに移動する.



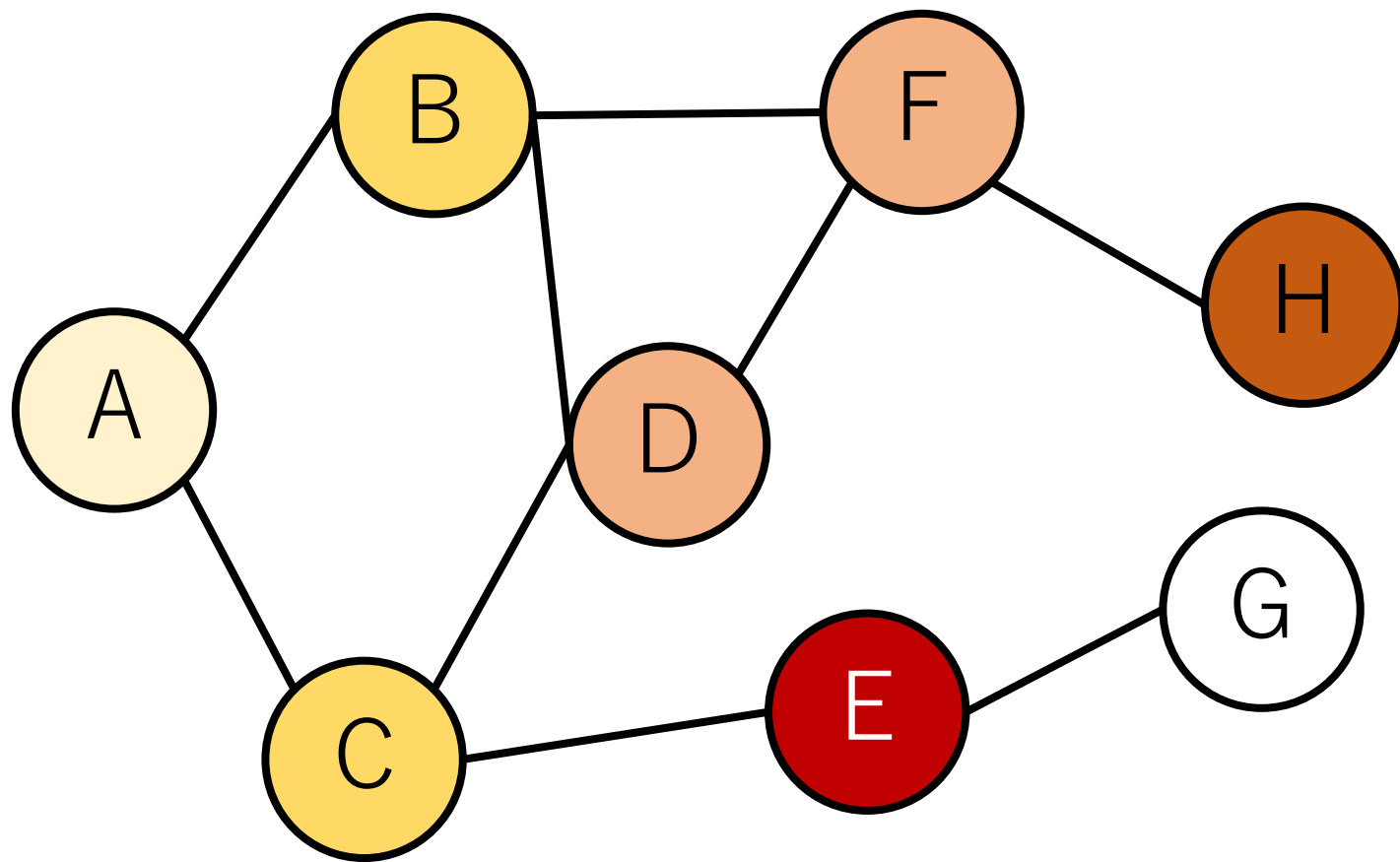
DFSの例

Eを見つける。
(Aはすでに
訪問済)



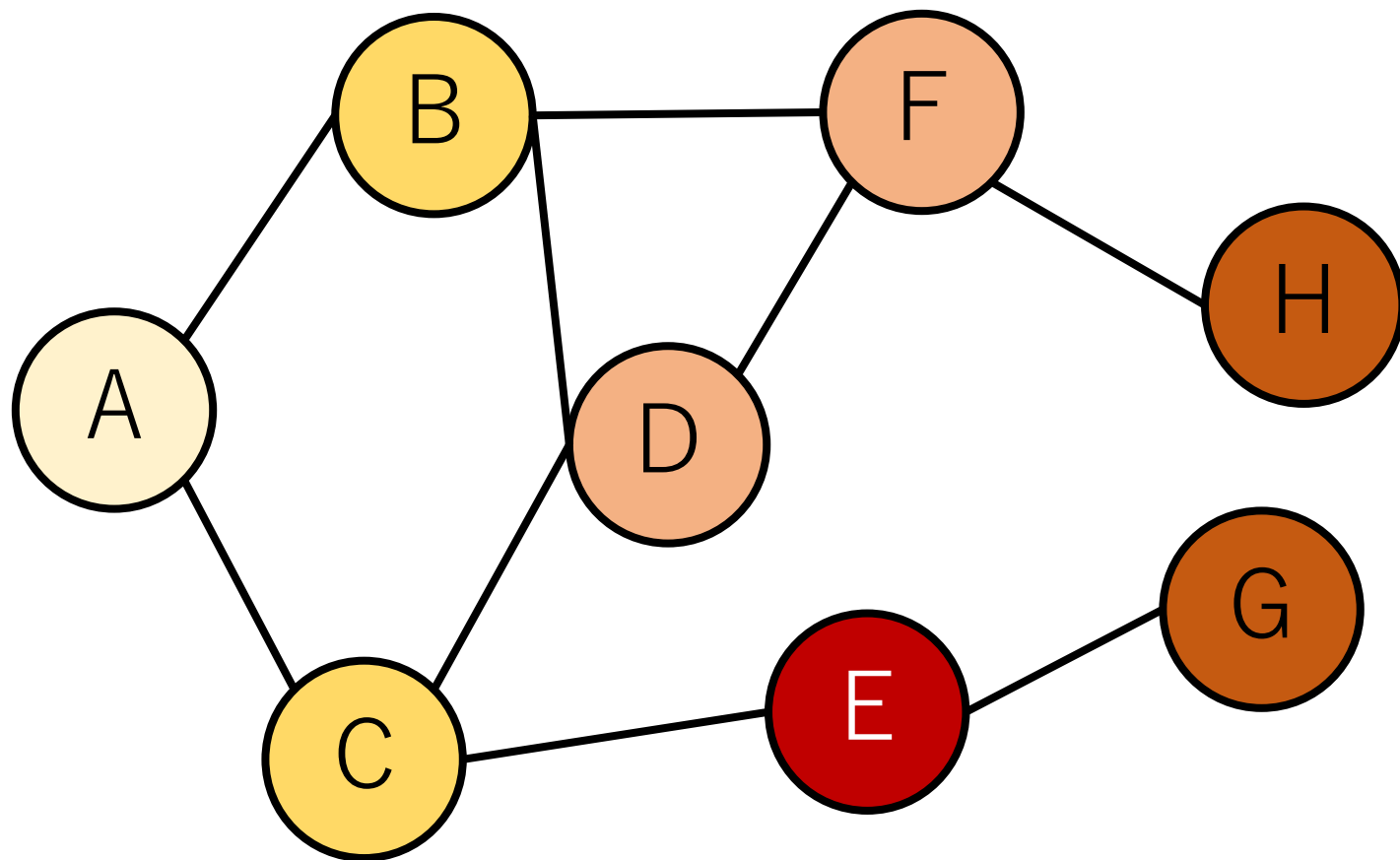
DFSの例

Eに移る.



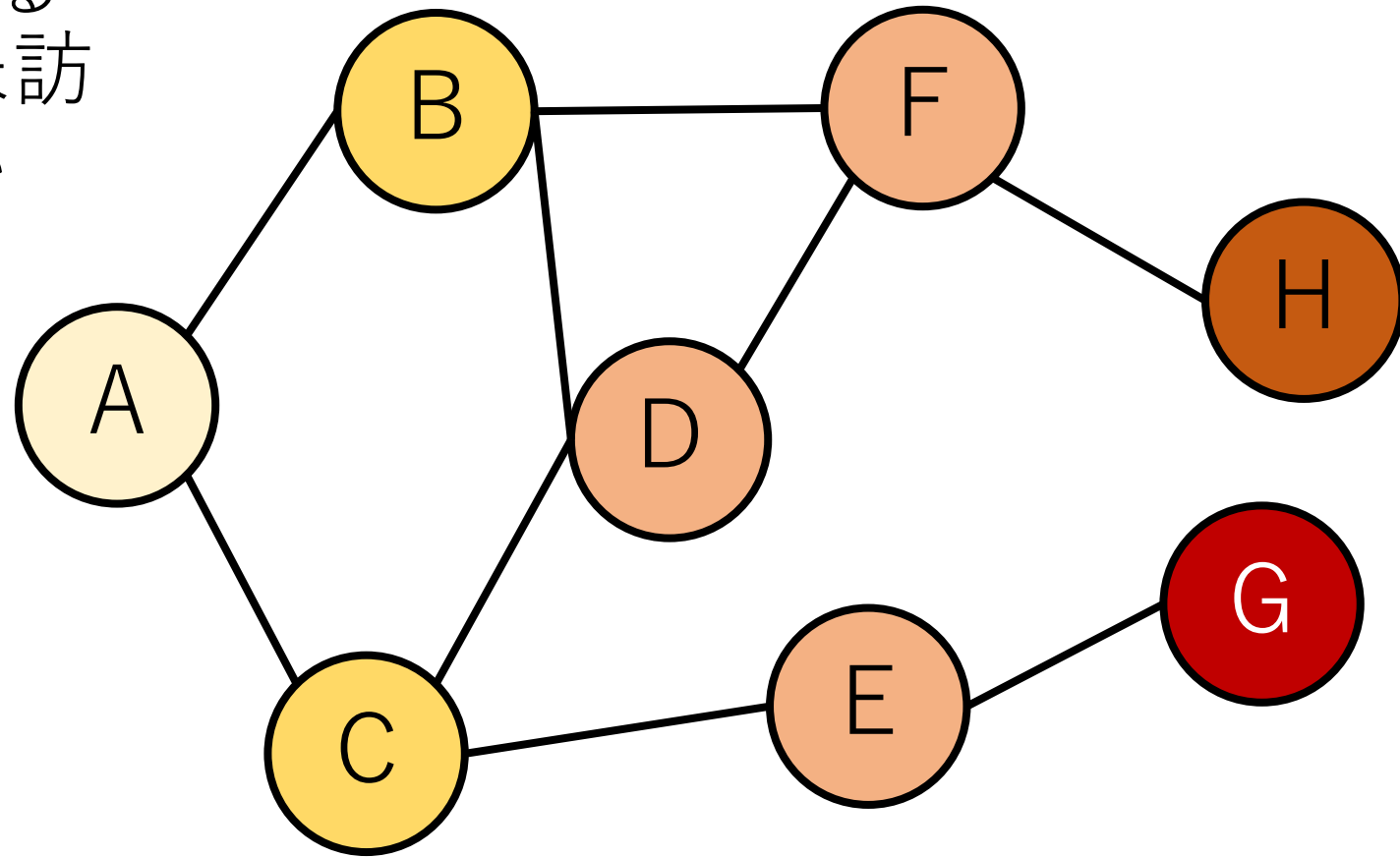
DFSの例

Gを見つける。



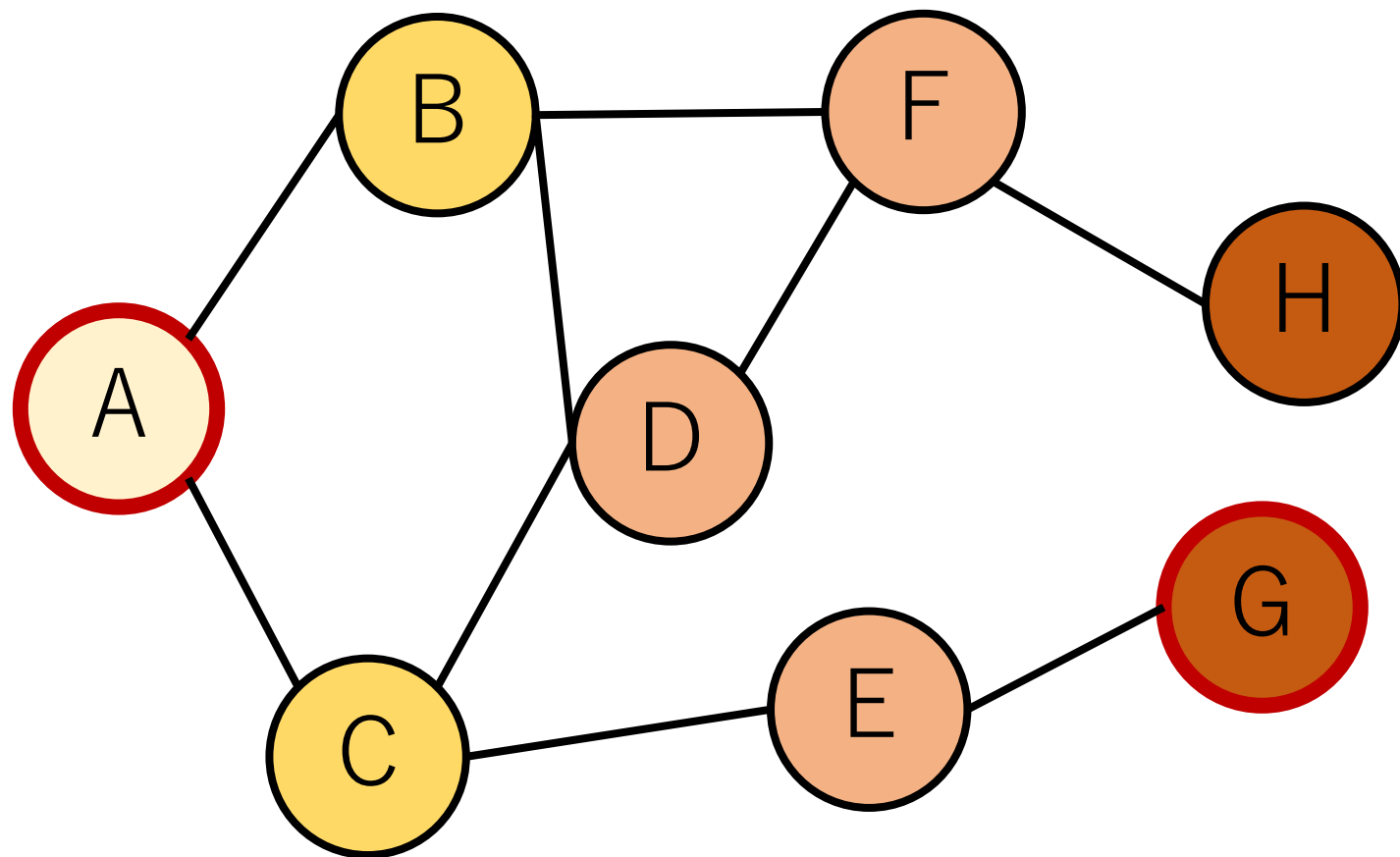
DFSの例

Gからはこれ以上つながる
ノードはない。また、未訪
問なノードも存在しない
ので、探索終了。



DFSの例

AからGへはつながっていることがわかった！



DFSの実装方針

スタックを使って実装。（再帰で実装することも多い）

スタックから取り出す：

取り出したノードに移動する。

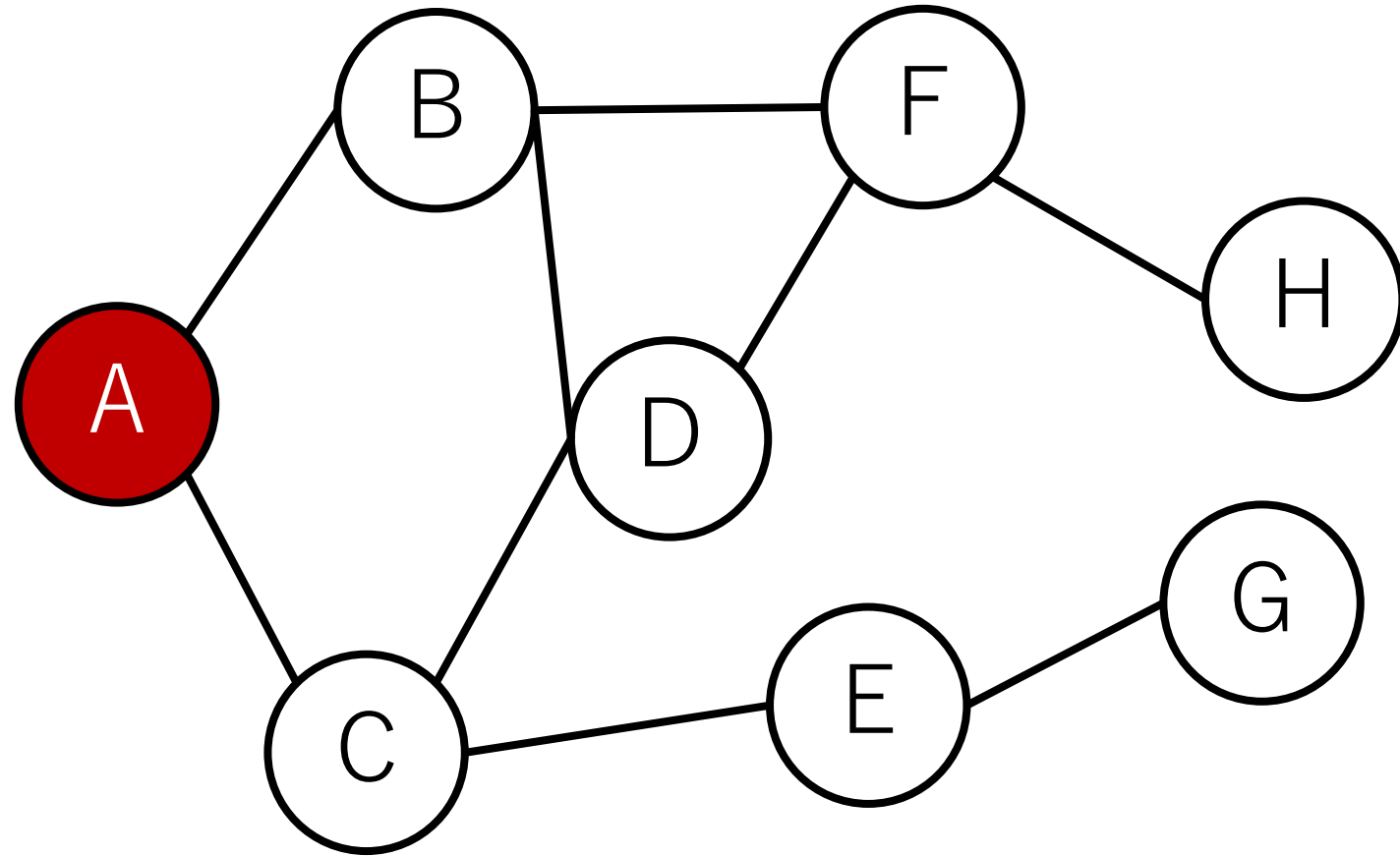
スタックに入れる：

新しく見つかったノードを入れる。

DFSの例 & スタックのダンプ

Aからスタート.

スタックにAをいれて
初期化. スタックの
最初からスタート.

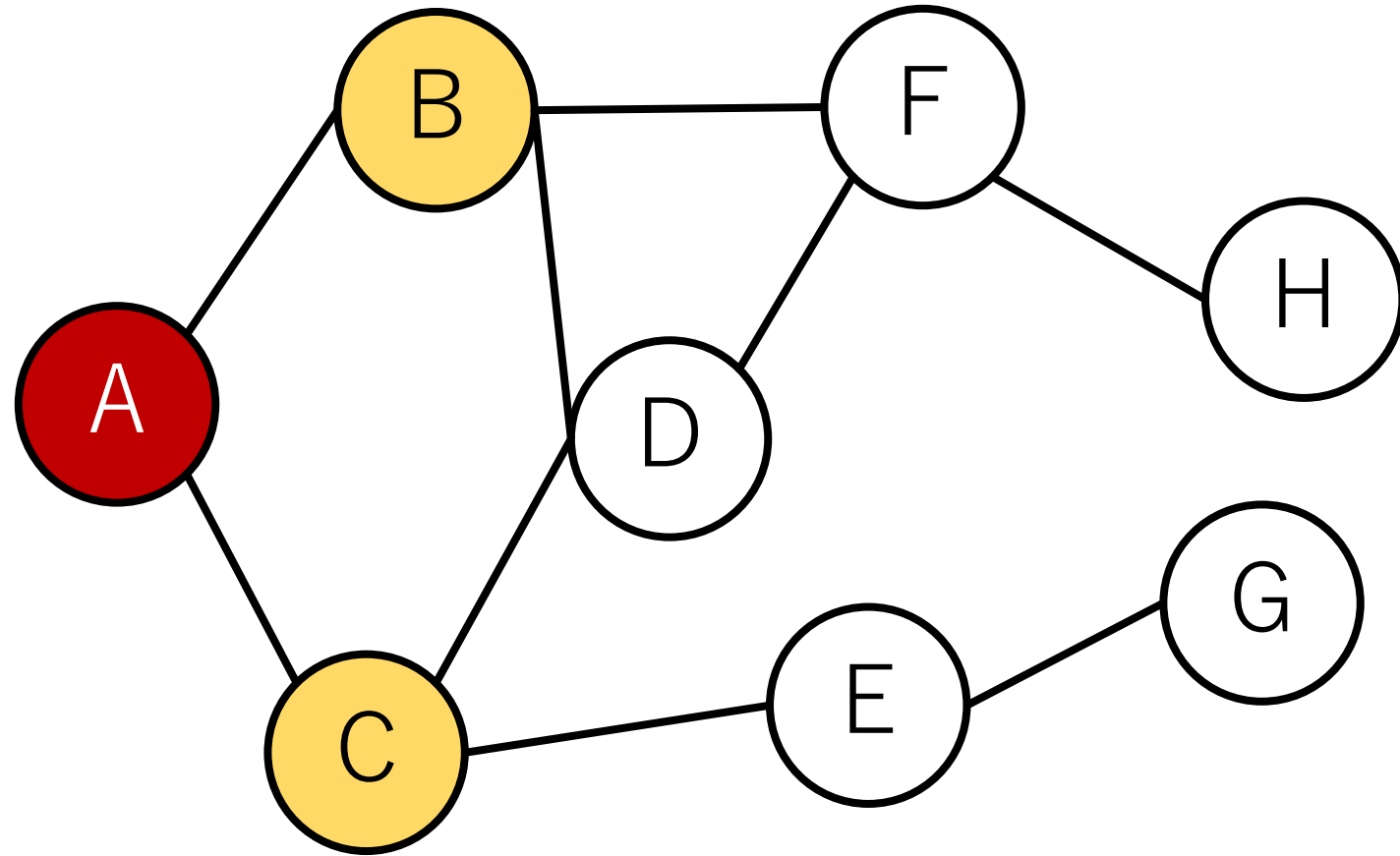


A

DFSの例 & スタックのダンプ

1ステップでつながっているのは、BとC.

BとCをスタックに入れる.



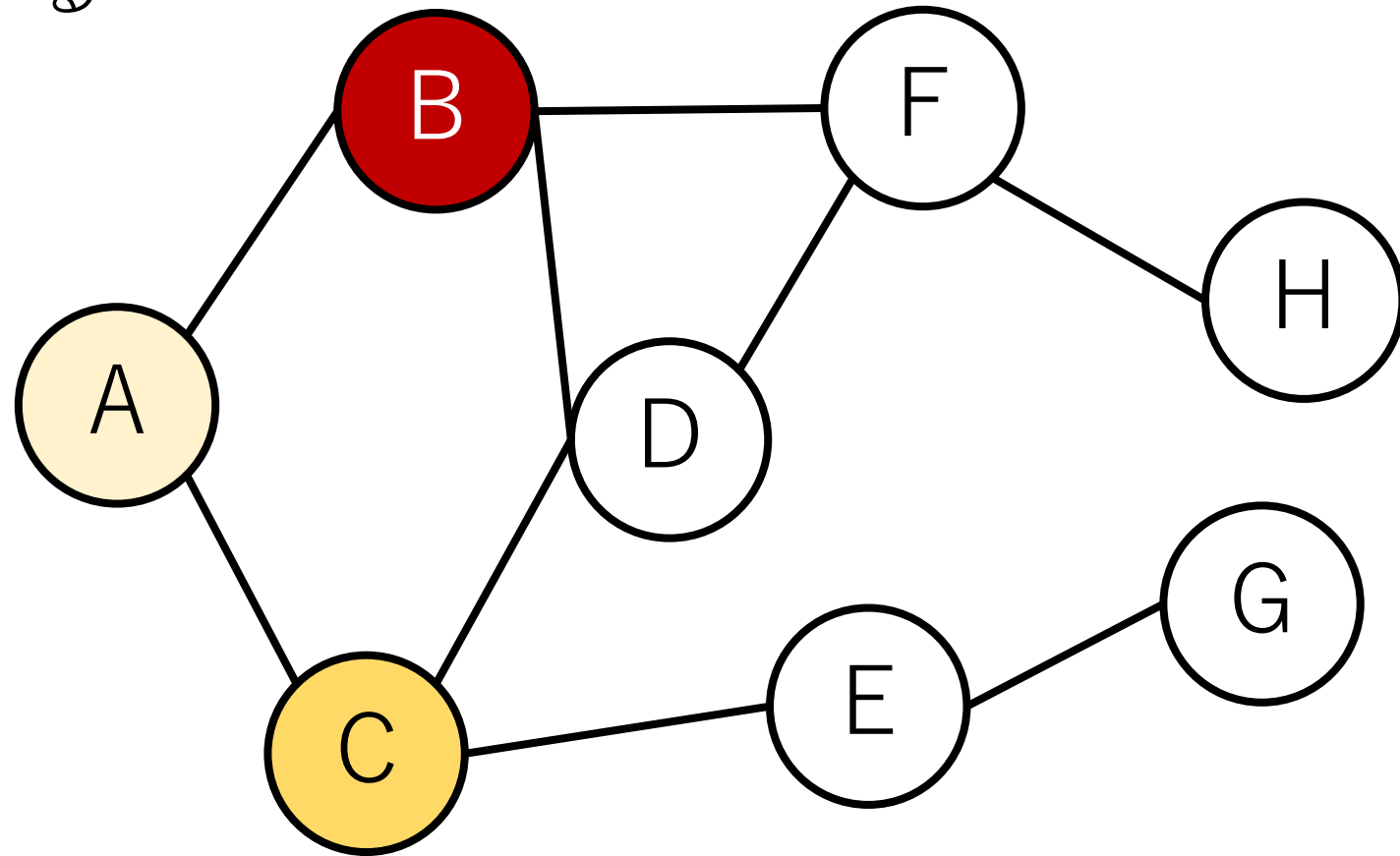
C

B

DFSの例 & スタックのダンプ

Bから1ステップでつながる
ノードを見る。(Cから
始めてもよい。)

スタックから取り出す。



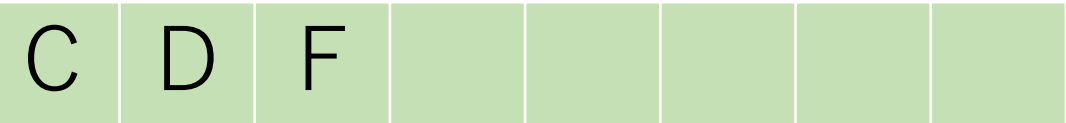
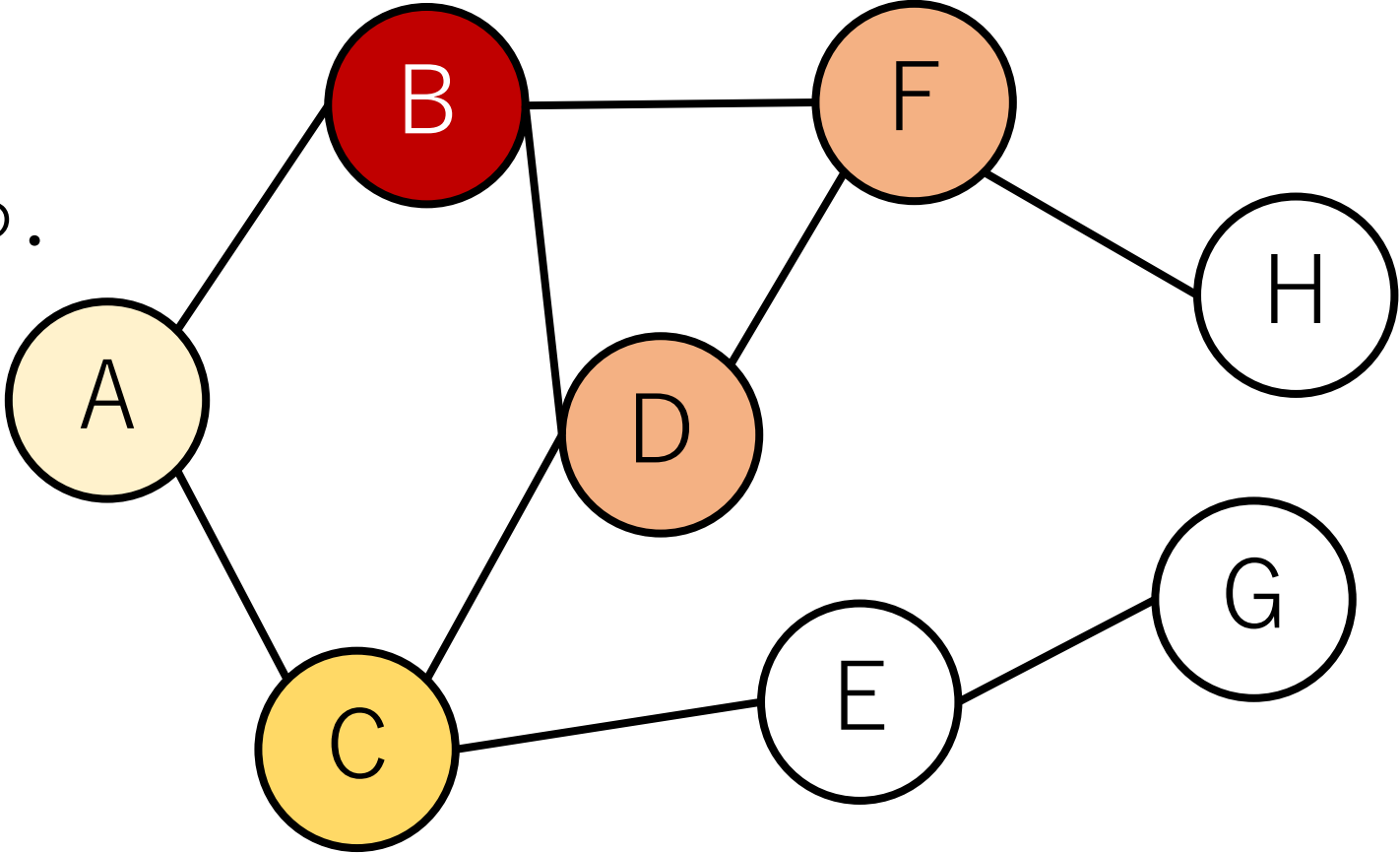
C

B

DFSの例 & スタックのダンプ

DとFを発見.

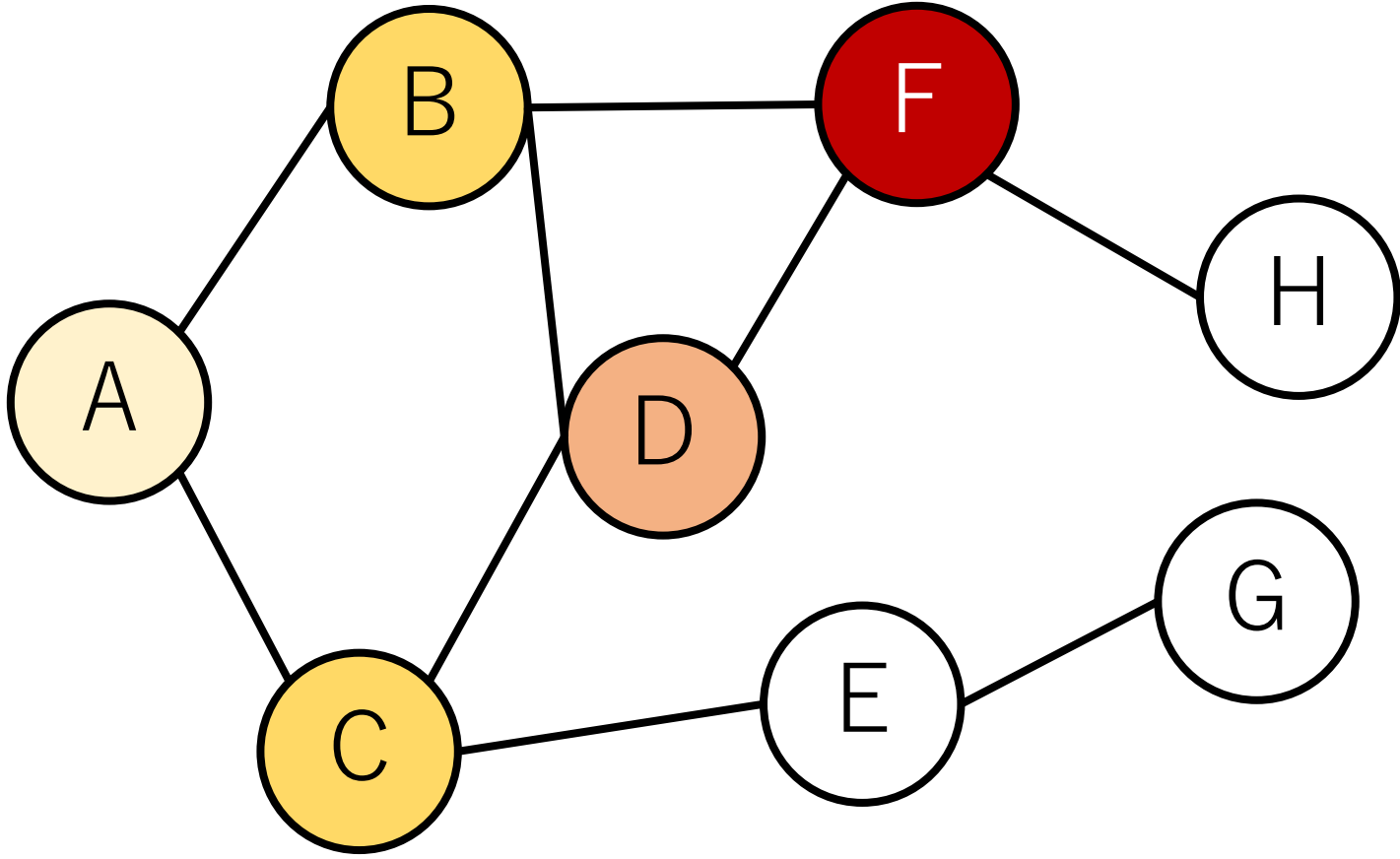
DとFをスタックにいれる.



DFSの例 & スタックのダンプ

Fに移る.

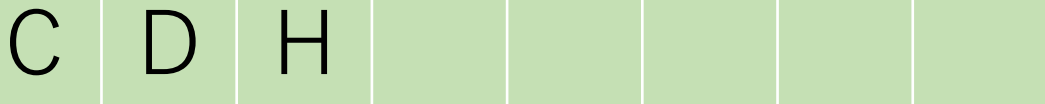
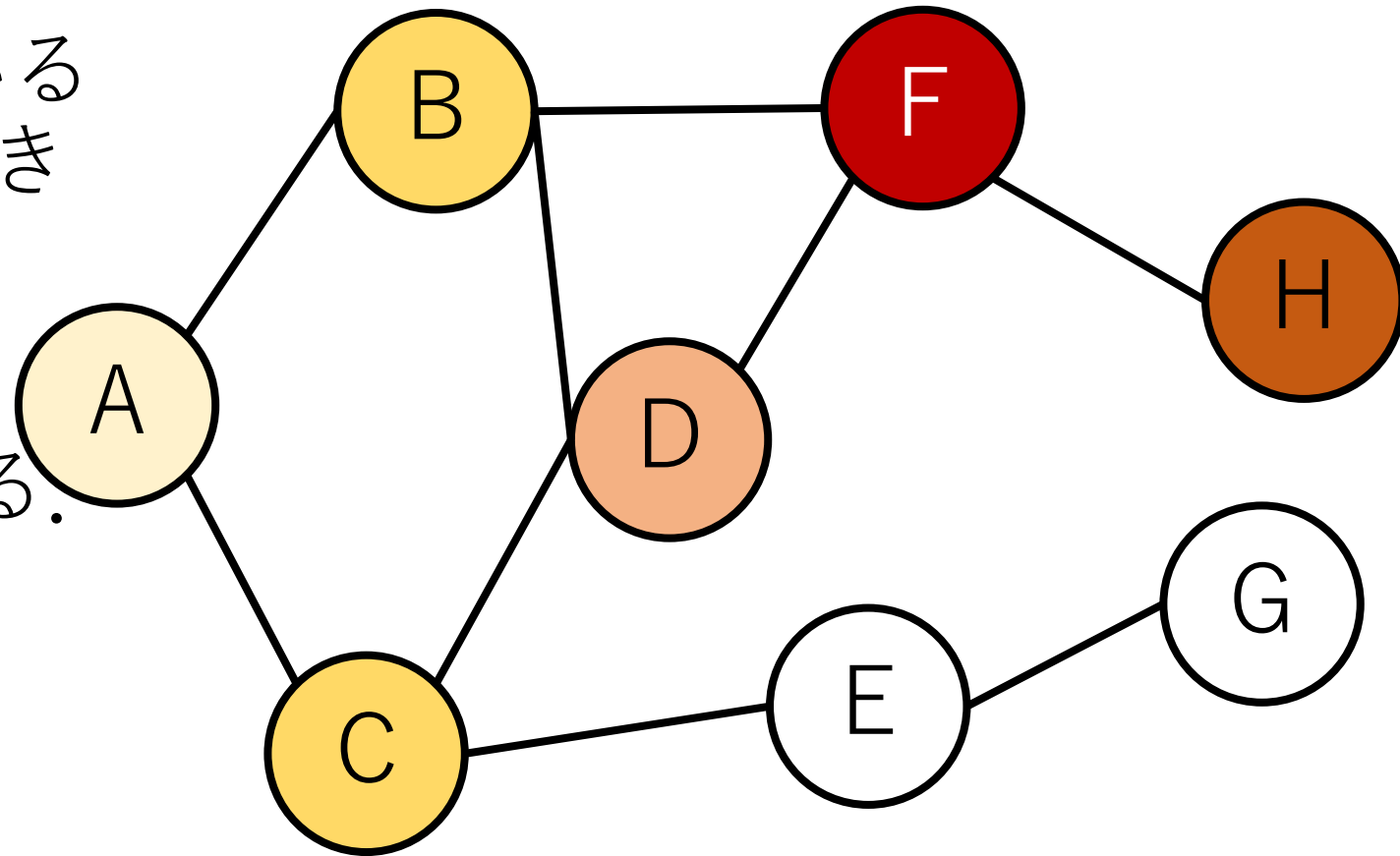
スタックから取り出す.



DFSの例 & スタックのダンプ

DとHを発見. DはBから見たときに発見はしているので, Hのみを訪れるべき候補として追加する.

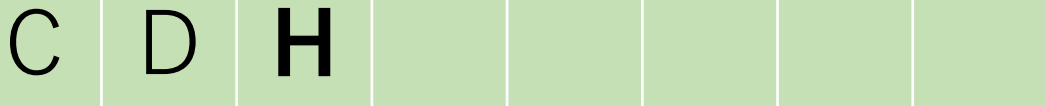
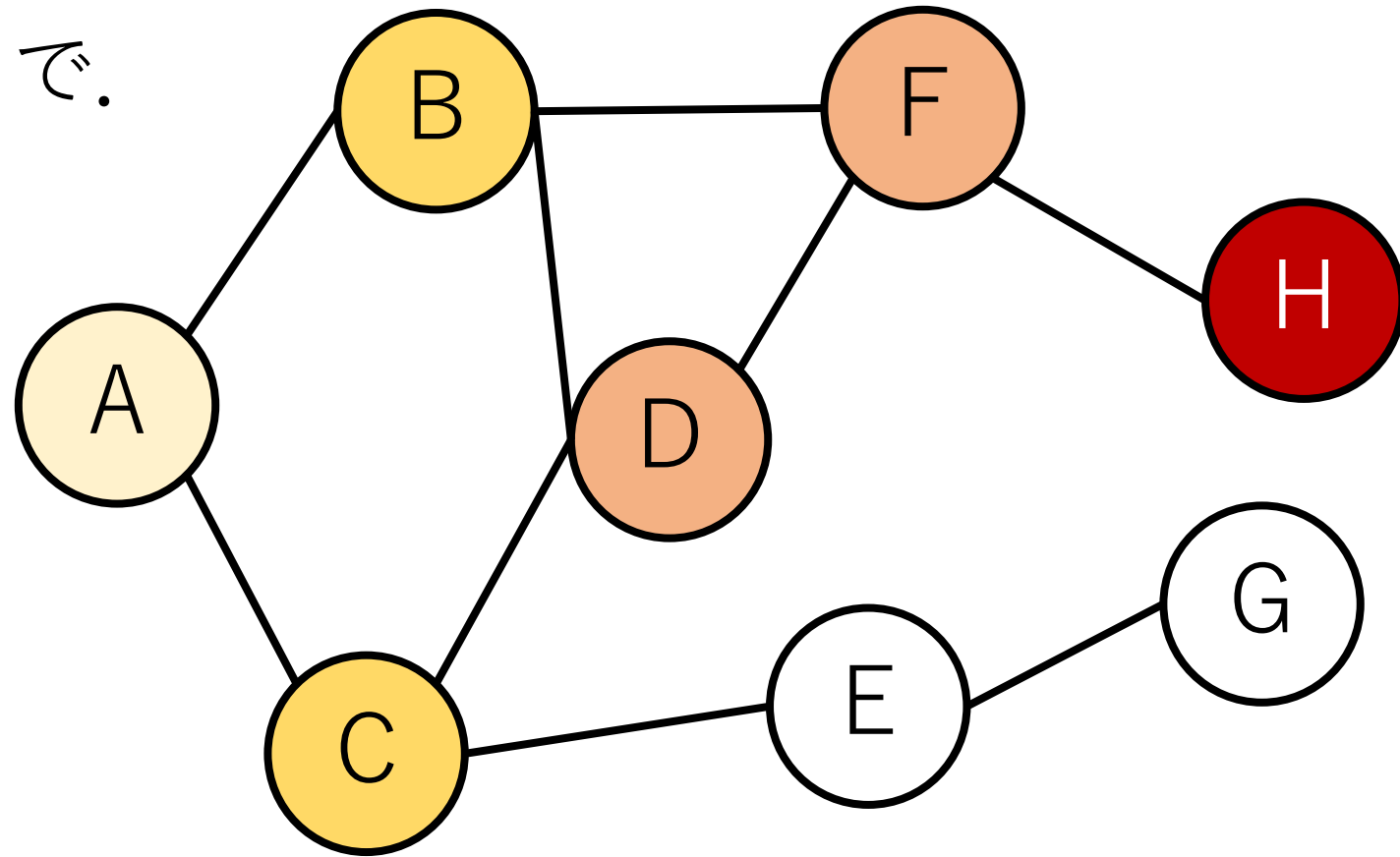
DとHをスタックに入れる.



DFSの例

Hに移るが、その先は存在しないので、ここまで。

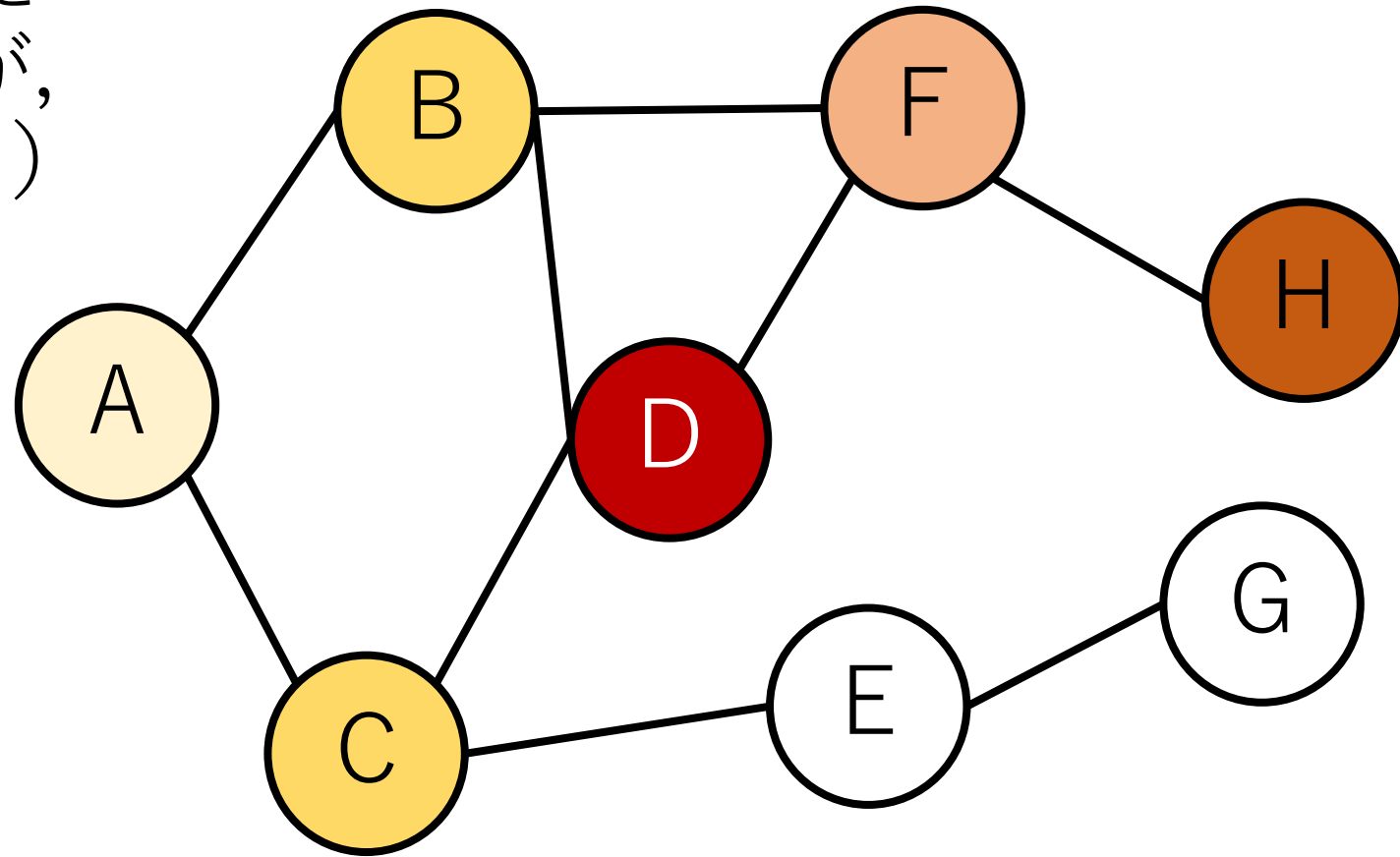
スタックから取り出す。



DFSの例

Bまで後戻り，今度はDに移動する。（発見済だが，訪問はまだしていない。）

スタックから取り出す。

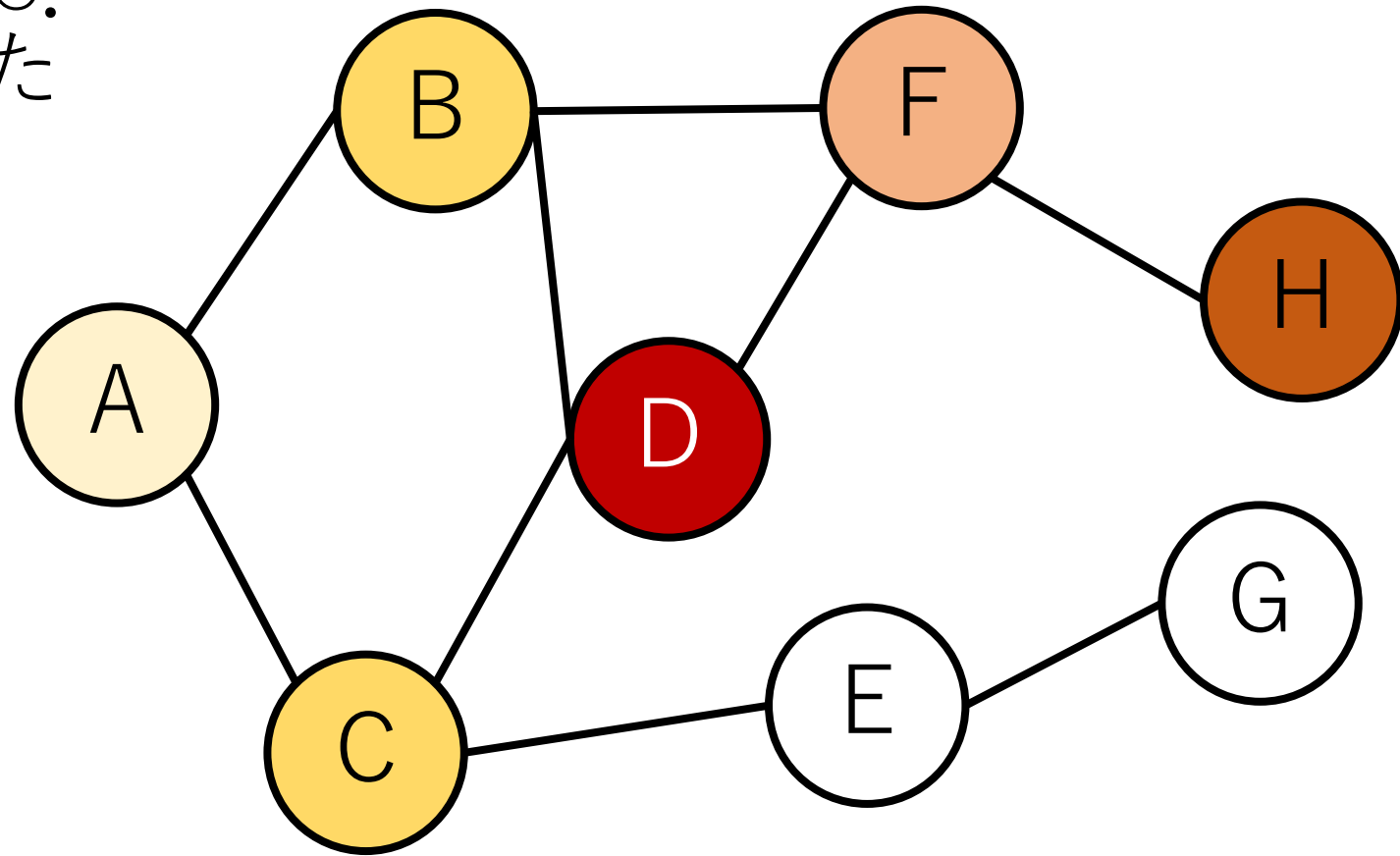


C

D

DFSの例

Dから未訪問のノードはC.
ただし、これもAから見た
ときに発見済なので、D
からは移動しない。



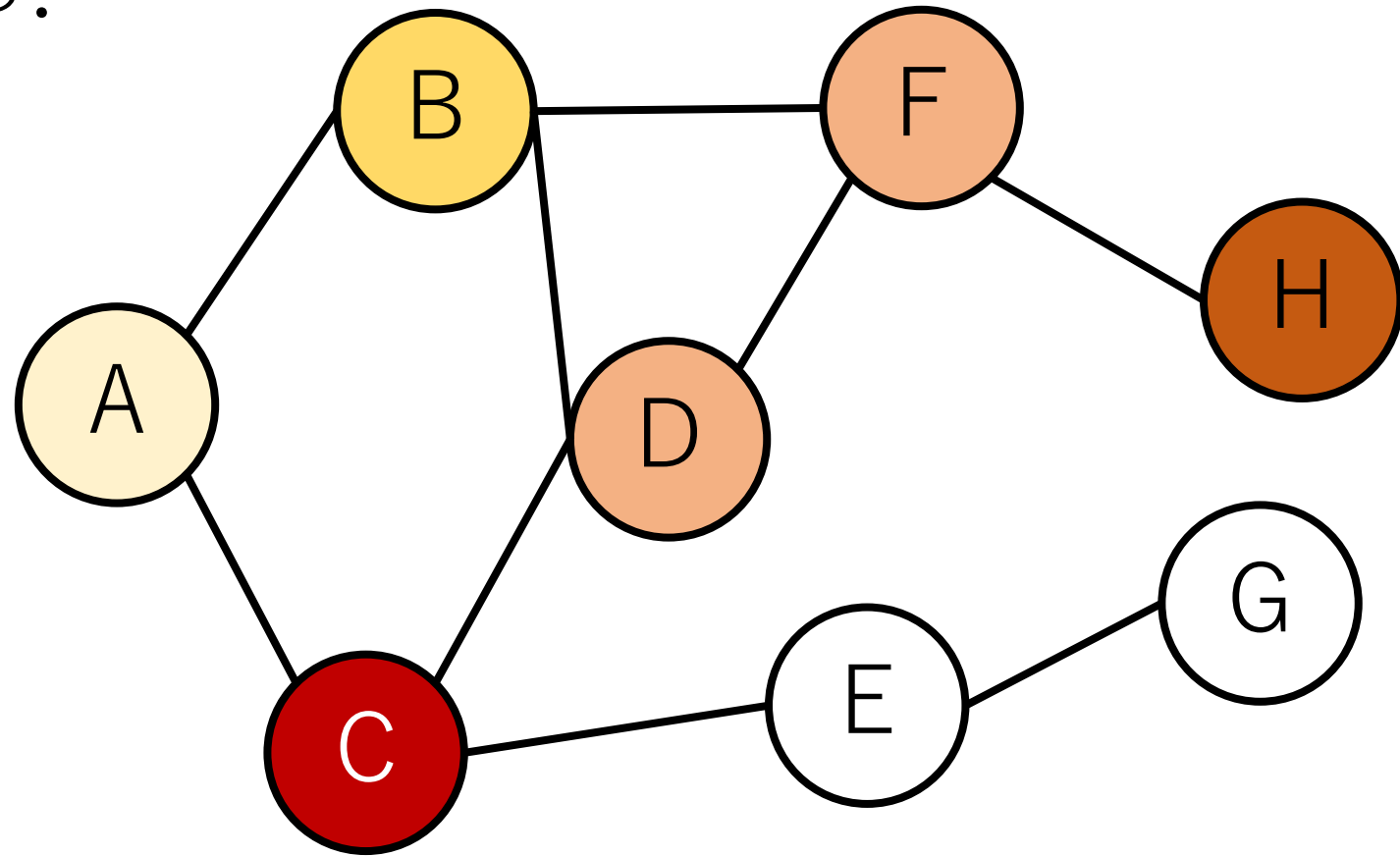
C

D

DFSの例

Aに戻って, Cに移動する.

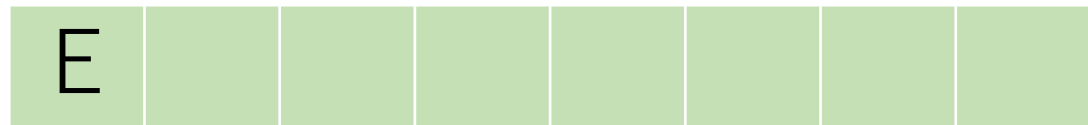
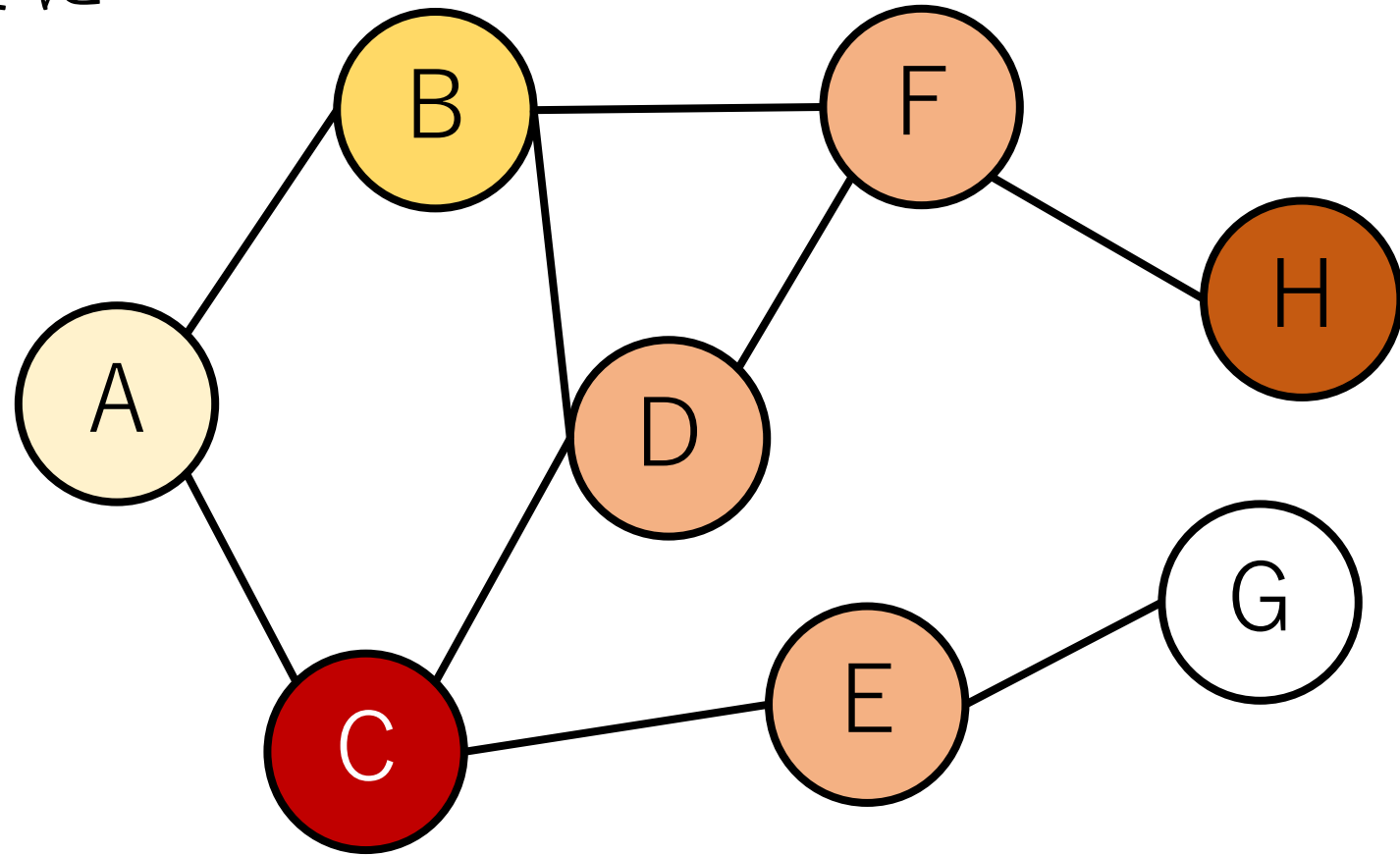
スタックから取り出す.



DFSの例

Eを見つける。(Aはすでに
訪問済)

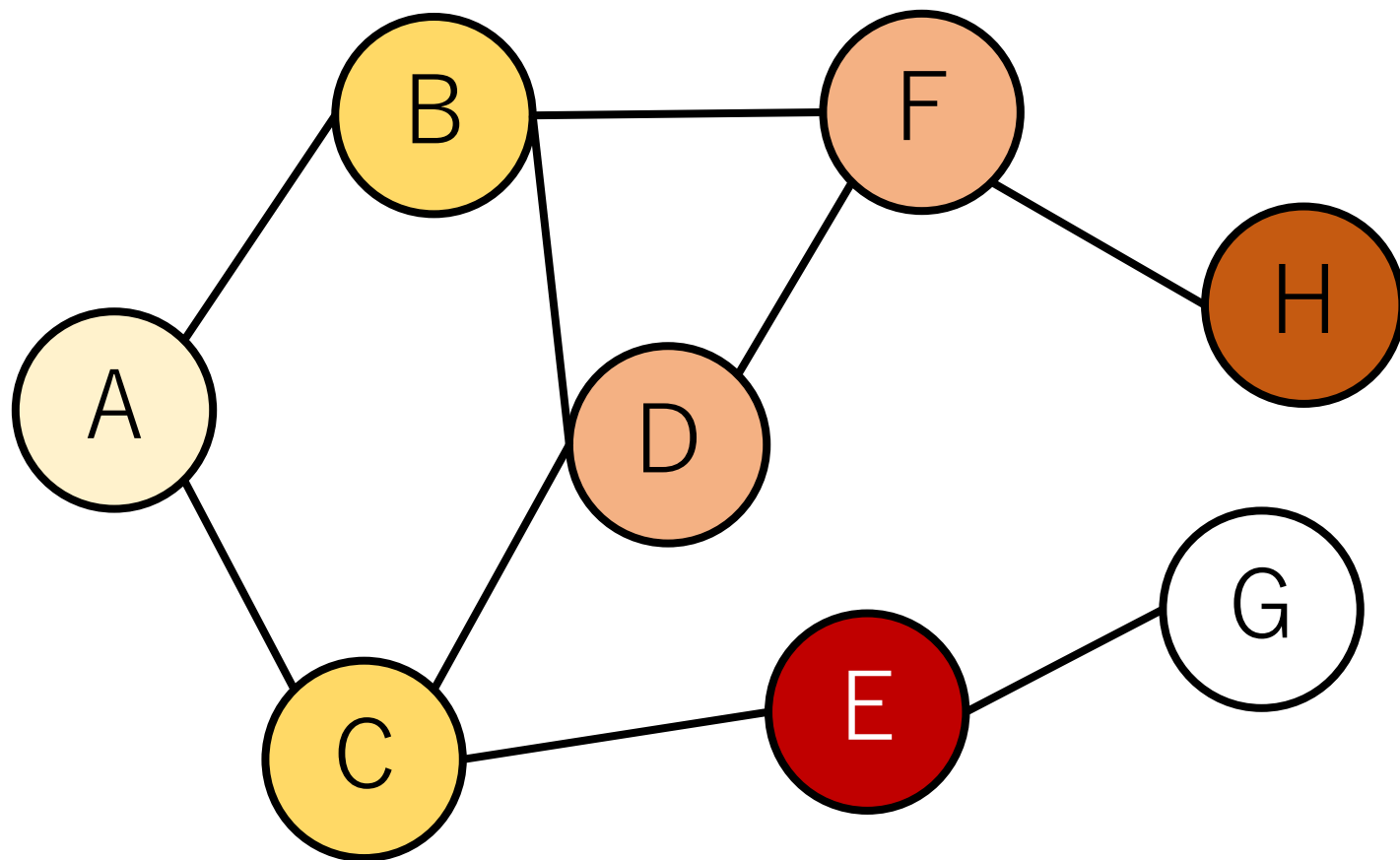
Eをスタックに入れる。



DFSの例

Eに移る.

スタックから取り出す.

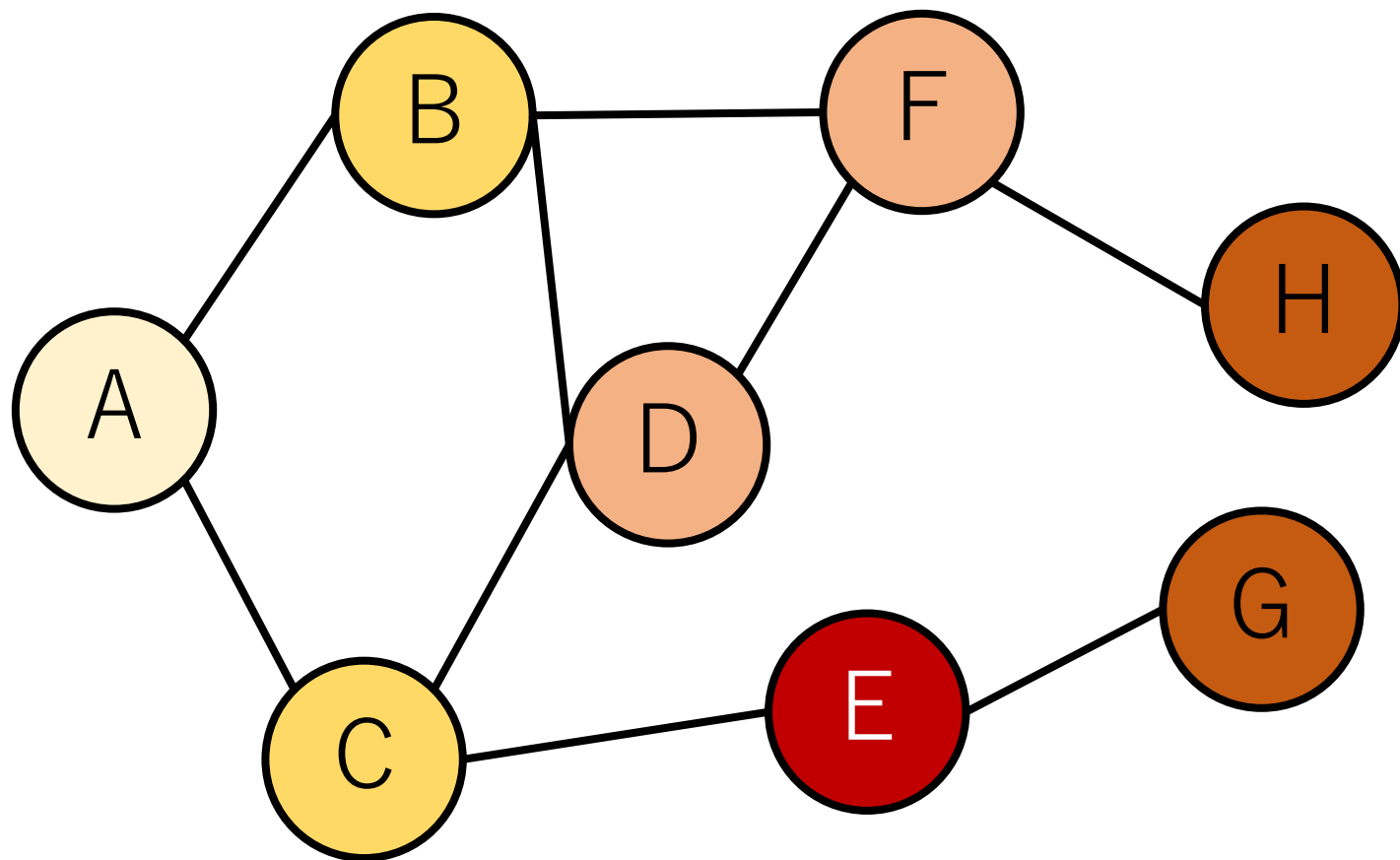


E

DFSの例

Gを見つける。

Gをスタックに入れる。

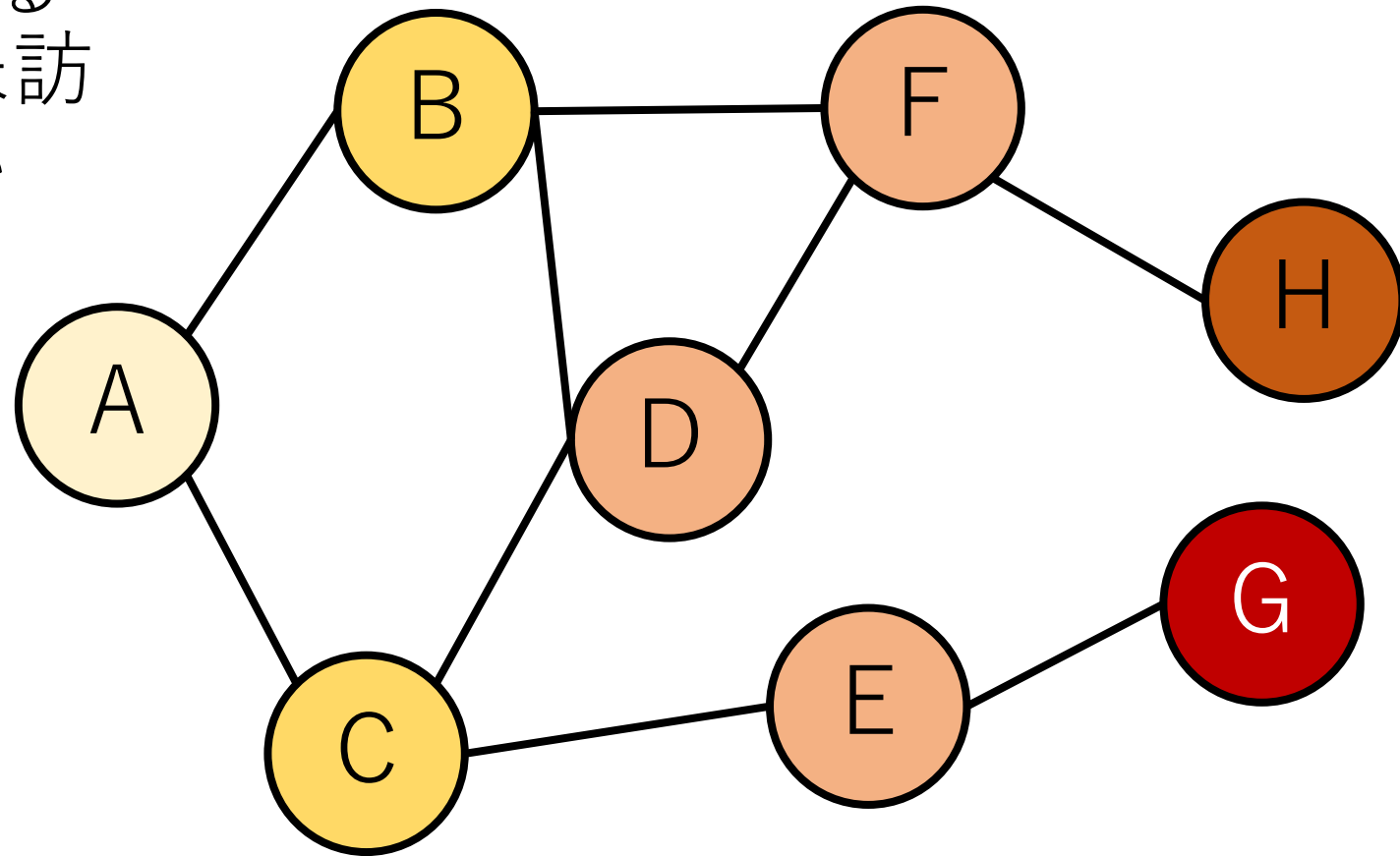


G

DFSの例

Gからはこれ以上つながる
ノードはない。また、未訪
問なノードも存在しない
ので、探索終了。

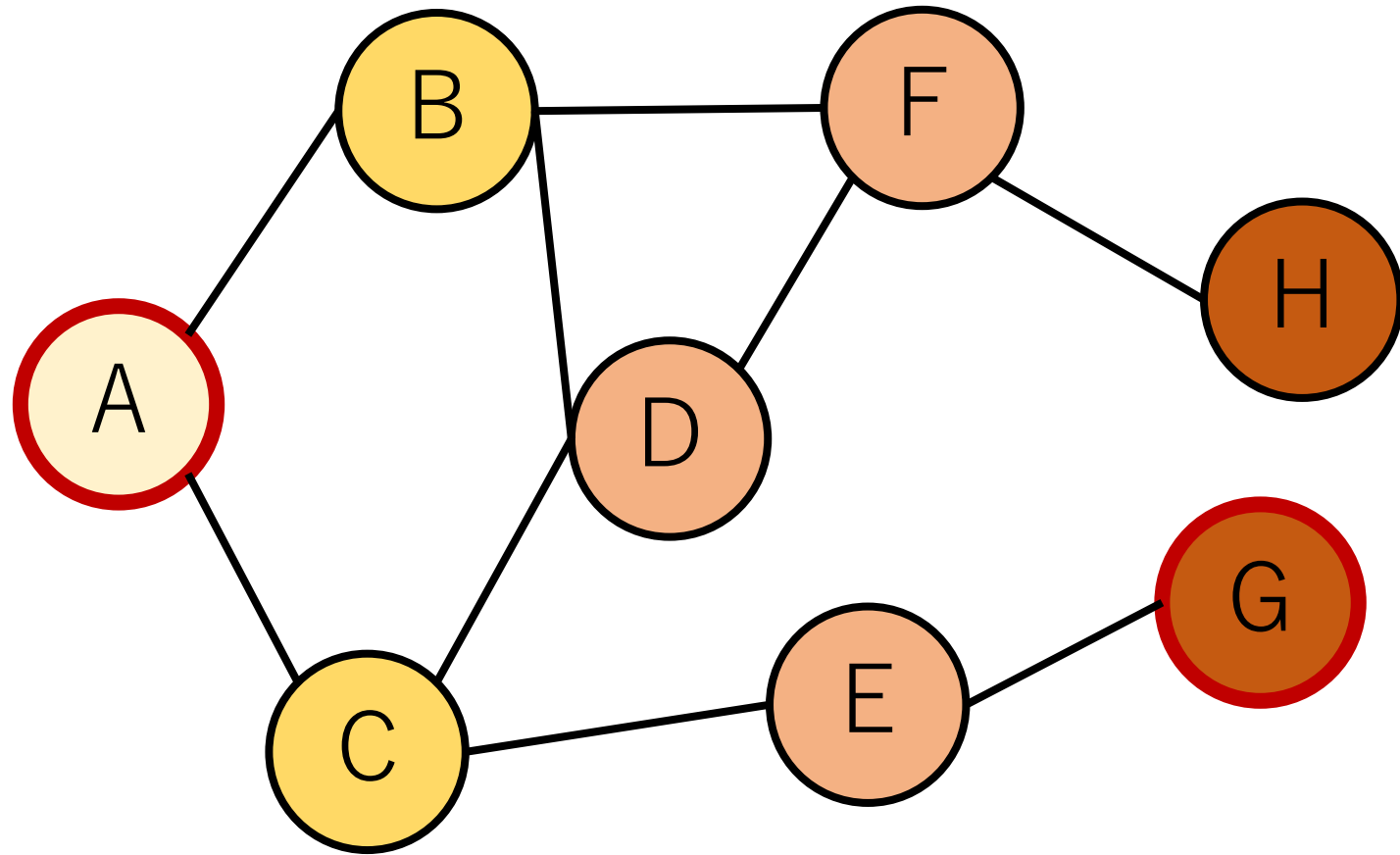
スタックから取り出す。



G

DFSの例

AからGへはつながっていることがわかった！



G

BFSとDFSの比較

取り出したノードに移動する

BFS：キューから取り出す

DFS：スタックから取り出す

新しく見つかったノードを入れる

BFS：キューに入れる

DFS：スタックに入れる

BFSとDFSの比較

取り出したノードに移動する

BFS：キューから取り出す

DFS：スタックから取り出す

新しく見つかったノードを入れる

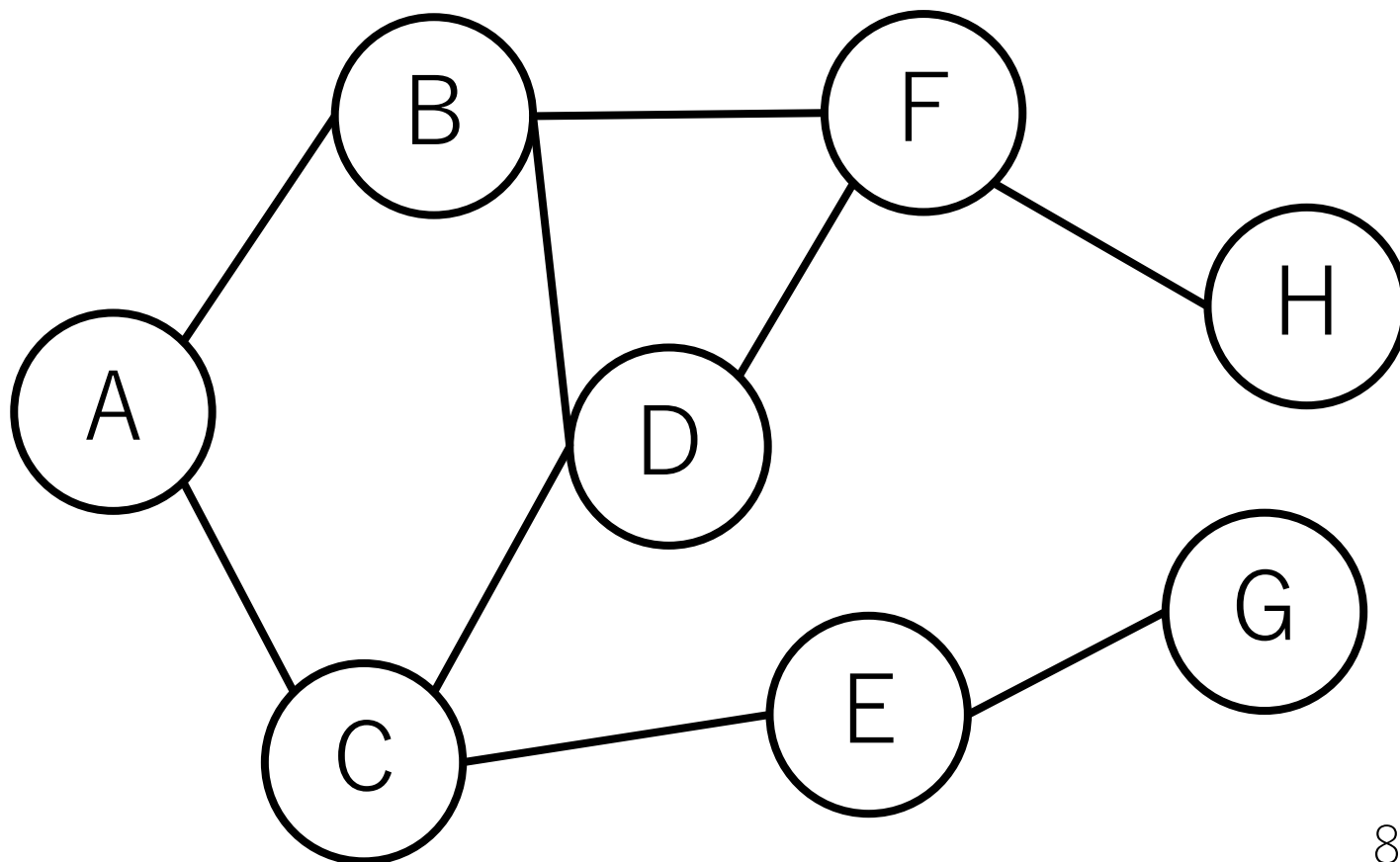
BFS：キューに入れる

DFS：スタックに入れる

データ構造だけが違う！

BFSの実装 (隣接リスト)

```
edges_list = [  
[1, 2],    #ノードA  
[0, 3, 5], #ノードB  
[0, 3, 4], #ノードC  
[1, 2, 5], #ノードD  
[2, 6],    #ノードE  
[1, 3, 7], #ノードF  
[4],       #ノードG  
[5]]       #ノードH
```



BFSの実装（初期化）

```
from collections import deque
# 出力をわかりやすくするためのリストで，なくてもよい
n_name = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']

# 引数：隣接リスト，開始ノード，見つけたいノード
def bfs(edges, start, end):
    # 訪問する候補となるノードを保持． dequeを利用．
    waiting = deque()
```

BFSの実装（初期化）

```
def bfs(edges, start, end):  
    waiting = deque()
```

```
# 発見・訪問済みかどうかのリスト。 Vはノードの数。
```

```
# 0: 未発見, 1: 発見済だが未訪問, 2: 訪問済
```

```
done = [0]*V
```

```
done[start] = 2      # 開始ノードは訪問済
```

BFSの実装（探索）

```
def bfs(edges, start, end):  
    ...  
    # 開始ノードから接続されているノードを追加  
    for n in edges[start]:  
        done[n] = 1    # 発見したが未訪問なので1  
        waiting.append(n)    # キューに追加
```

BFSの実装（探索）

```
def bfs(edges, start, end):
```

```
    ...
```

```
    # waitingの長さが0でない限り，発見したが未訪問の  
    # ノードがあることになる。
```

```
    while len(waiting):
```

```
        # キューから取り出して，ノード移動。
```

```
        cur_node = waiting.popleft()
```

BFSの実装 (探索)

```
def bfs(edges, start, end):  
    ...  
    while len(waiting):  
        ...  
        # cur_nodeを訪問済みにする。  
        if done[cur_node] != 2:  
            done[cur_node] = 2  
            print('Moved to {}'.format(node_name[cur_node]))  
            if(end == cur_node): print('=FOUND!=')
```

BFSの実装（探索）

```
def bfs(edges, start, end):
```

```
    ...
```

```
        if done[cur_node] != 2:
```

```
            # cur_nodeから接続されているノードを追加
```

```
                for n in edges[cur_node]:
```

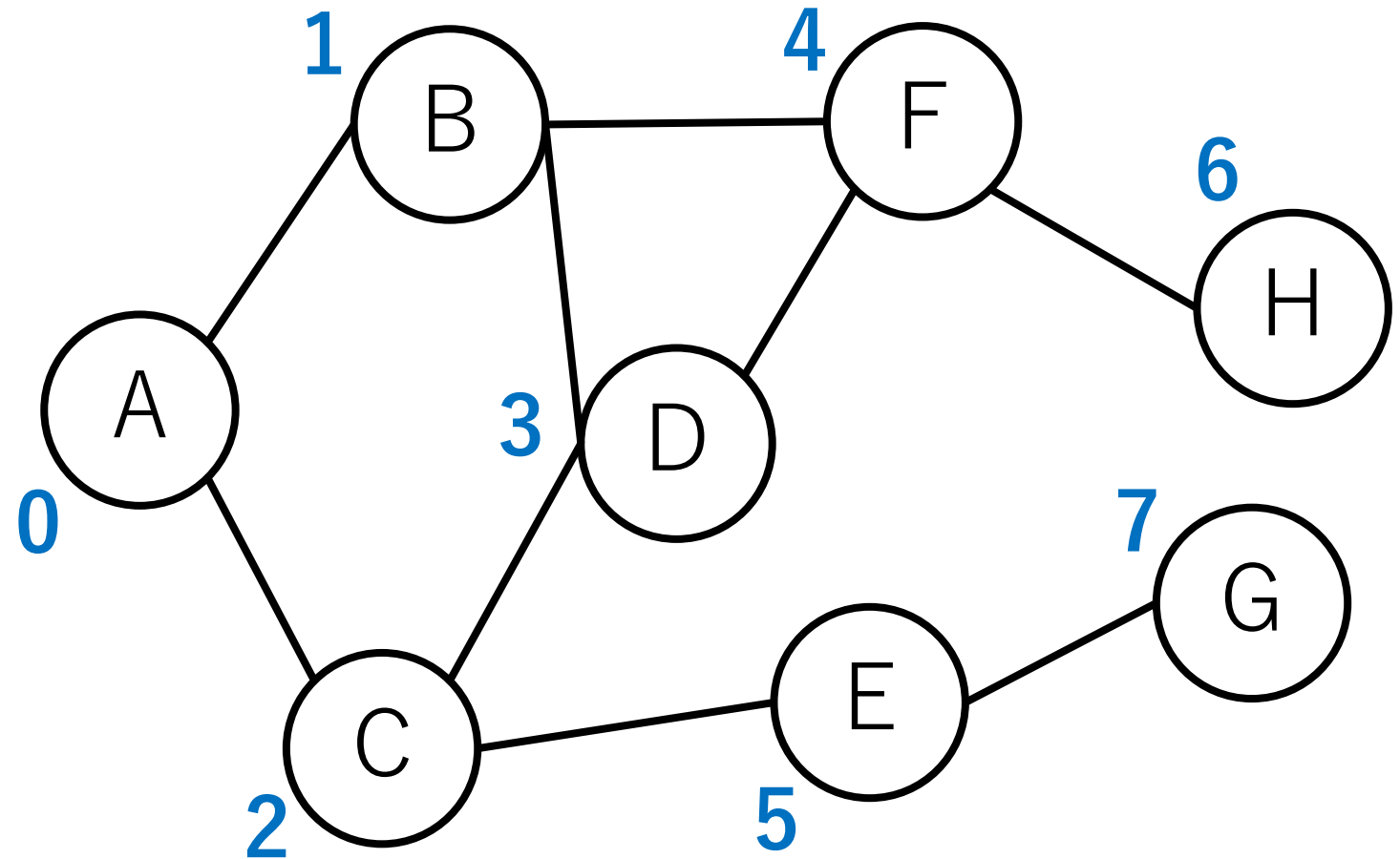
```
                    if done[n] == 0:        # 初めて発見
```

```
                        done[n] = 1        # 未訪問なので1
```

```
                        waiting.append(n)   # キュー追加
```

bfs(edge, 0, 6)の実行結果

Moved to B
Moved to C
Moved to D
Moved to F
Moved to E
Moved to H
Moved to G
=FOUND!=



DFSの実装（探索）

```
def dfs(edges, start, end):
```

```
    ...
```

```
    while len(waiting):
```

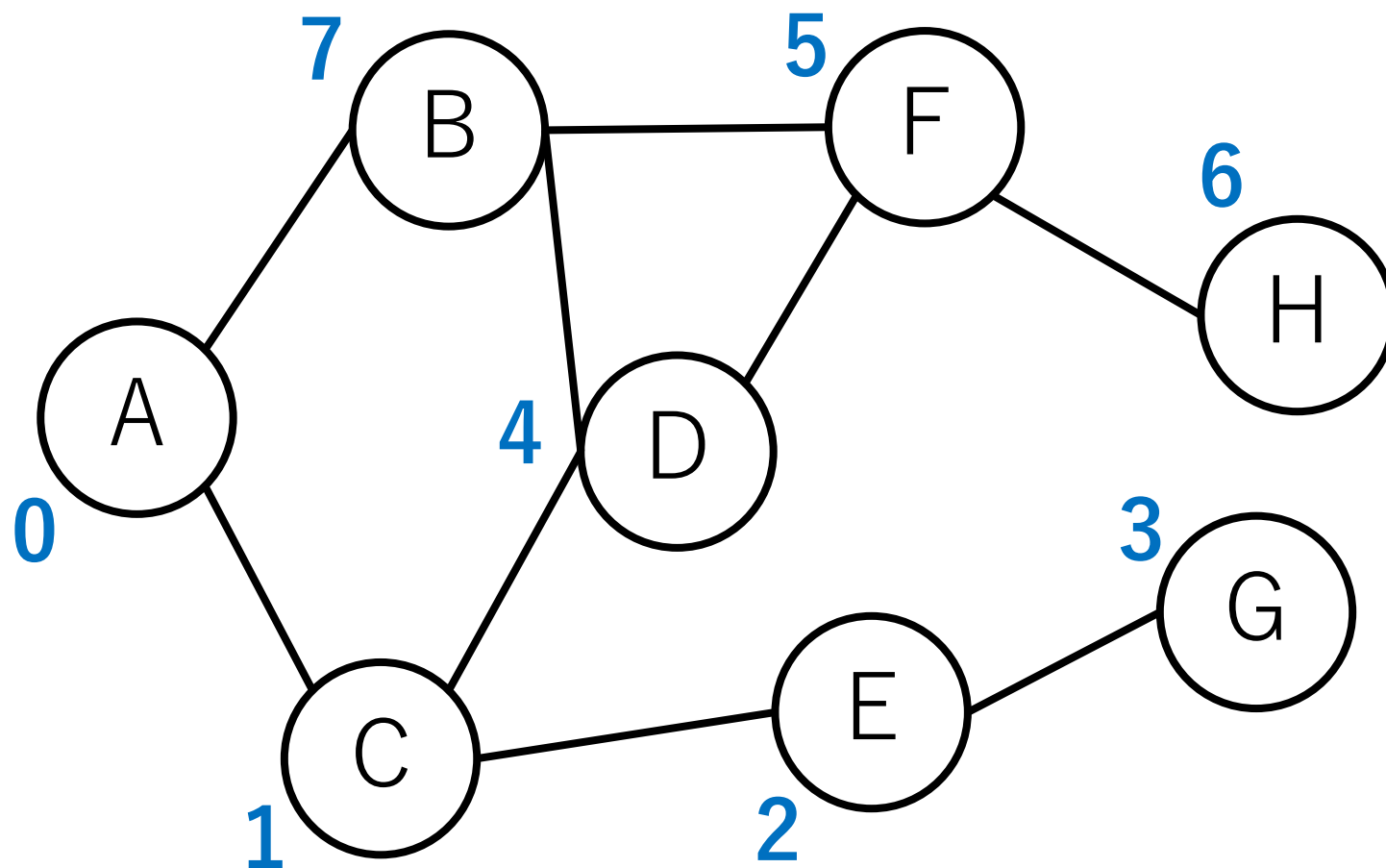
```
        # スタックから取り出して，ノード移動.
```

```
        cur_node = waiting.pop()
```

変更はこれだけ！

dfs(edge, 0, 6)の実行結果

Moved to C
Moved to E
Moved to G
=FOUND!=
Moved to D
Moved to F
Moved to H
Moved to B



visitedを未訪問, 訪問済の2値にする実装

先のコードで言えば, $done[n] = 1$ の行を削除したものに等価.

その場合, キュー, スタックへの値の入り方が変わり, 訪問順序も変わることがあるが, 幅優先, 深さ優先, になることには変わりない.

先の実装例はこの変更もできるように少し冗長な書き方になっています.

この実装への変更をしないのであれば,
 $if\ done[cur_node] \neq 2$ の判定が必要ありません.

重要なポイント！（再掲）

取り出したノードに移動する

BFS：キューから取り出す

DFS：スタックから取り出す

新しく見つかったノードを入れる

BFS：キューに入れる

DFS：スタックに入れる

データ構造だけが違う！

BFS, DFSの計算量

全てのノードを1度訪れ, 全ノードを処理するまでに全ての辺を1通りチェックする. ノードの数を $|V|$, 辺の数を $|E|$ とする.

隣接行列の場合, ある1つのノードを見るときに他の全部のノードとの接続をチェックするので, $O(|V|^2)$.

隣接リストの場合, つながっているノードのみを処理することになるので, $O(|V| + |E|)$.

メモリ消費量

メモリ消費量はグラフ，ツリーの構造による。

BFS：並行して探索途中となっている，同じ深さにあるノード全ての情報を保持する必要がある。

→横に広いようなグラフ，ツリーでは不利。

DFS：現在探索を行っているパスに関係するノードの情報を保持する必要がある。

→縦に長い（深い）グラフ，ツリーでは不利。

DFSの再帰版

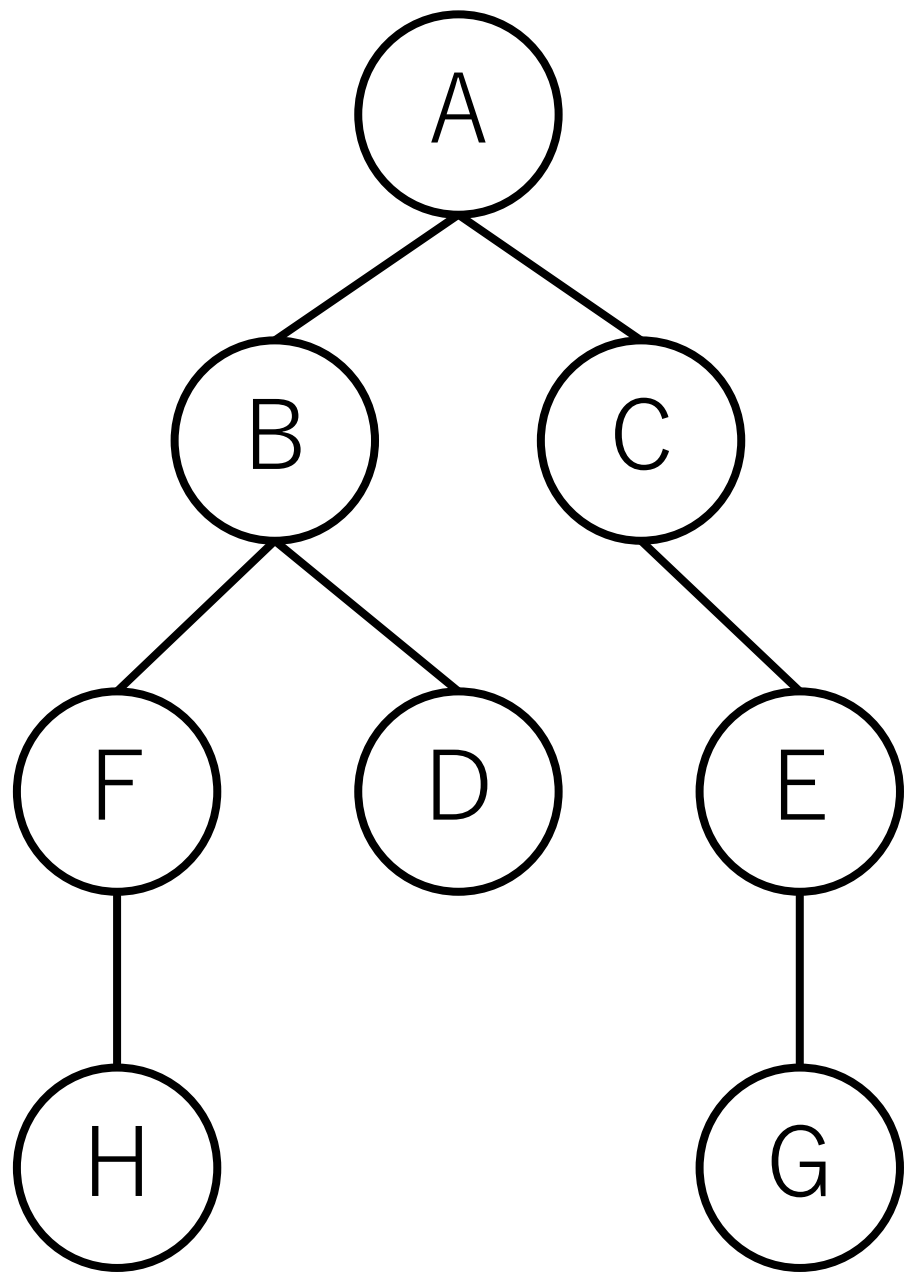
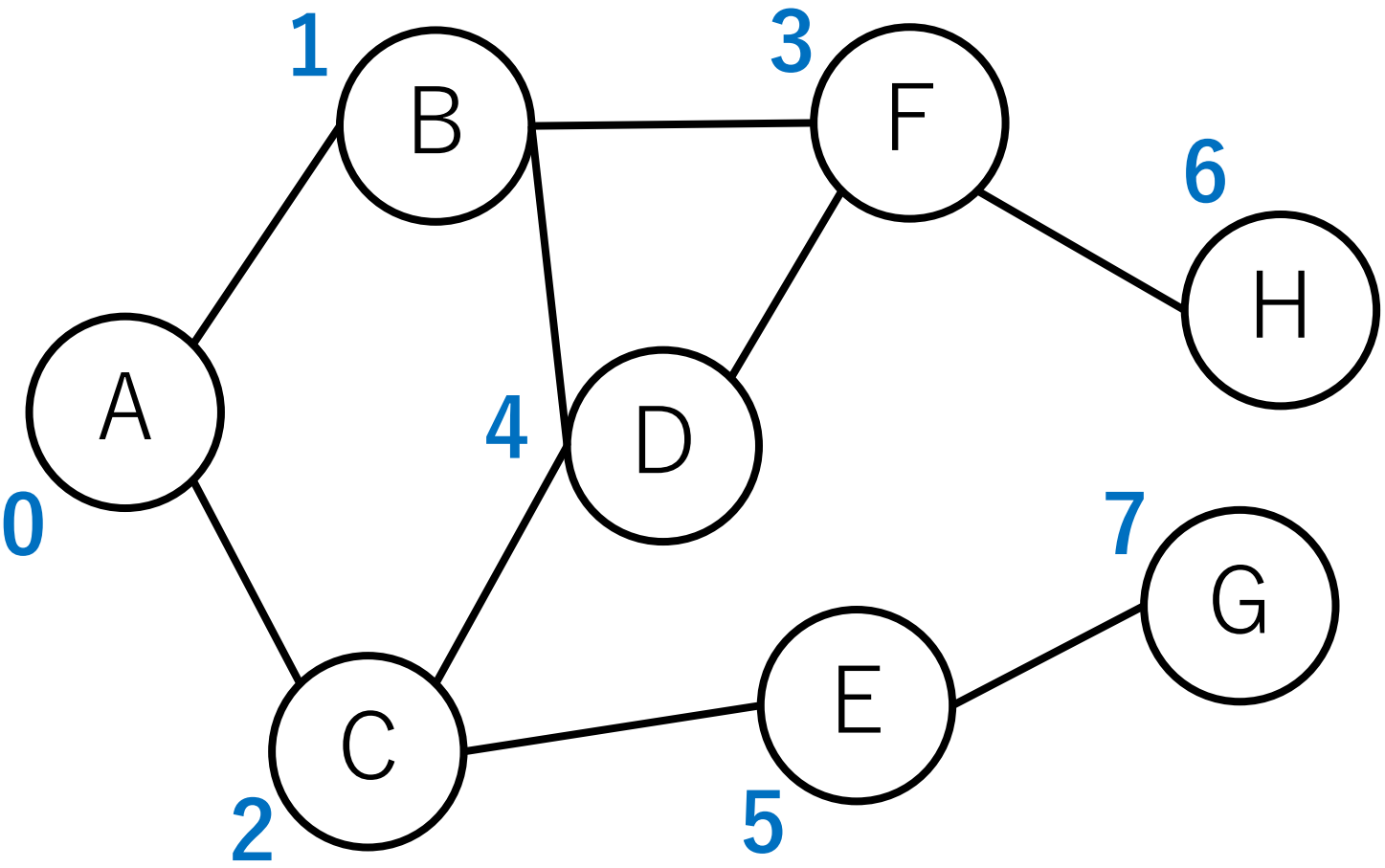
スタックを使うDFSは，再帰を使って書くことも出来る．
接続している未訪問のノードに移動して，DFSをすれば
よい．なお，この場合doneはBooleanで十分．（ただし，
再帰呼び出しの上限回数には注意．）

```
for n in edges[start]:  
    if [nが未訪問ならば]:  
        [nを訪問済にする]  
        [DFSを再帰で呼び出す]
```

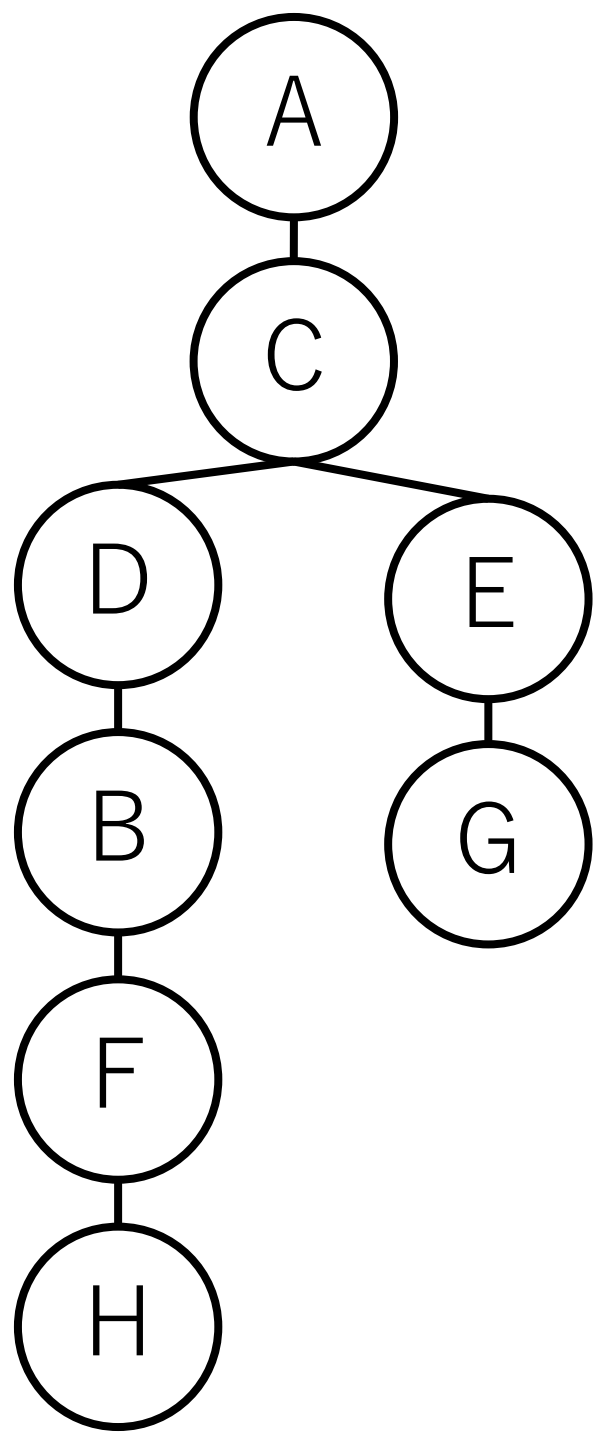
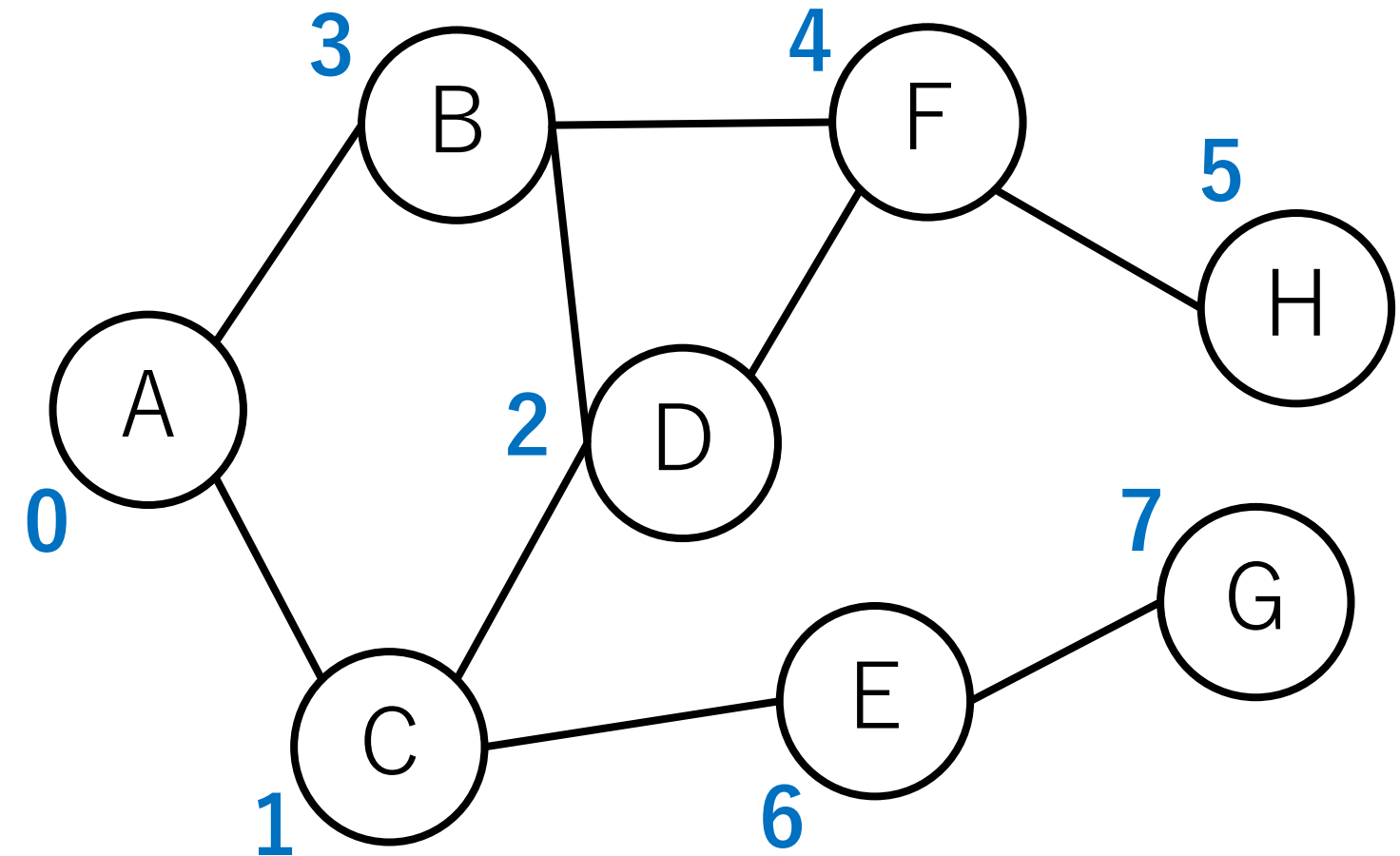
BFS木, DFS木

BFS, DFSを行うと, 通った辺の順番を基に木が構成できる.

BFS木



DFS木



BFS, DFSを応用できる問題

ノードAとノードBとの接続関係に関する問題

AからBにたどり着けるか？

AからBにたどり着くのに何ステップかかるか？

セルを順に塗りつぶしていく問題

Flood fillアルゴリズム

BFSが得意な問題

最小回のステップ，最短距離（ただし，辺のコストが一定のグラフ）などを求めるもの。

BFSは開始ノードから接続しているノードに対して1つずつステップを増やして探索する。よって，目標とするノードが見つかった時点で最短距離がわかる！

一方，DFSでは全ての経路をチェックしないと最短かどうかを確認できない。

BFSが得意な問題

問題文

たかはし君は迷路が好きです。今、上下左右に移動できる二次元盤面上の迷路を解こうとしています。盤面は以下のような形式で与えられます。

- まず、盤面のサイズと、迷路のスタート地点とゴール地点の座標が与えられる。
- 次に、それぞれのマスが通行可能な空きマス(' . ')か通行不可能な壁マス('#')かという情報を持った盤面が与えられる。盤面は壁マスで囲まれている。スタート地点とゴール地点は必ず空きマスであり、スタート地点からゴール地点へは、空きマスを通って必ずたどり着ける。具体的には、入出力例を参考にすると良い。

今、彼は上記の迷路を解くのに必要な最小移動手数を求めたいと思っています。どうやって求めるかを調べていたところ、「幅優先探索」という手法が効率的であることを知りました。幅優先探索というのは以下の手法です。

- スタート地点から近い(たどり着くための最短手数が少ない)マスから順番に、たどり着く手数を以下のように確定していく。説明の例として図1の迷路を利用する。

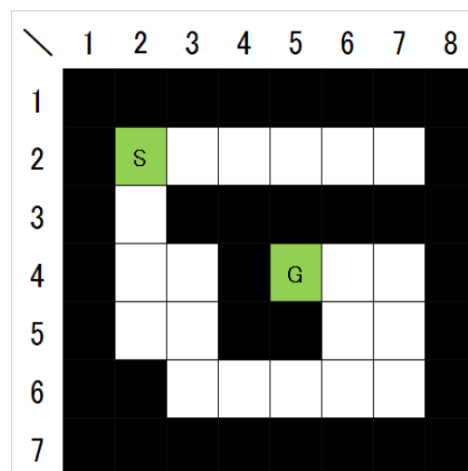


図1. 説明に用いる盤面

DFSが得意な問題

オイラーツアー

LCA

橋の検出（発展的内容）

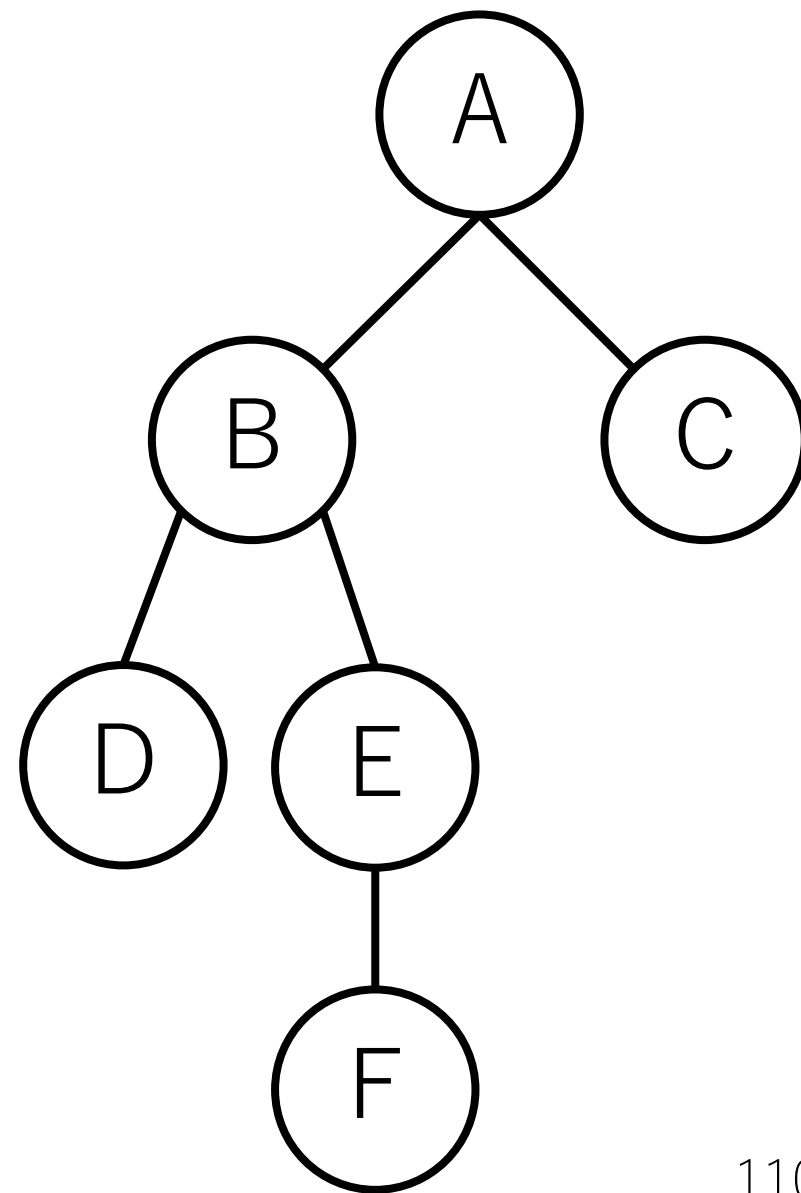
BFS, DFSを初めて勉強する方はこの前のスライドまでの内容とオイラーツアーをしっかりと掴んでもらえばOKです。特に、橋の検出の内容はちょっと腕に覚えのある人向け。



DFSが得意な問題：オイラーツアー

与えられた木において，DFSを行い，
そのたどった順番を出力したもの。

「木」でなければいけない点に注意。
DFS自体は木でなくても使える。



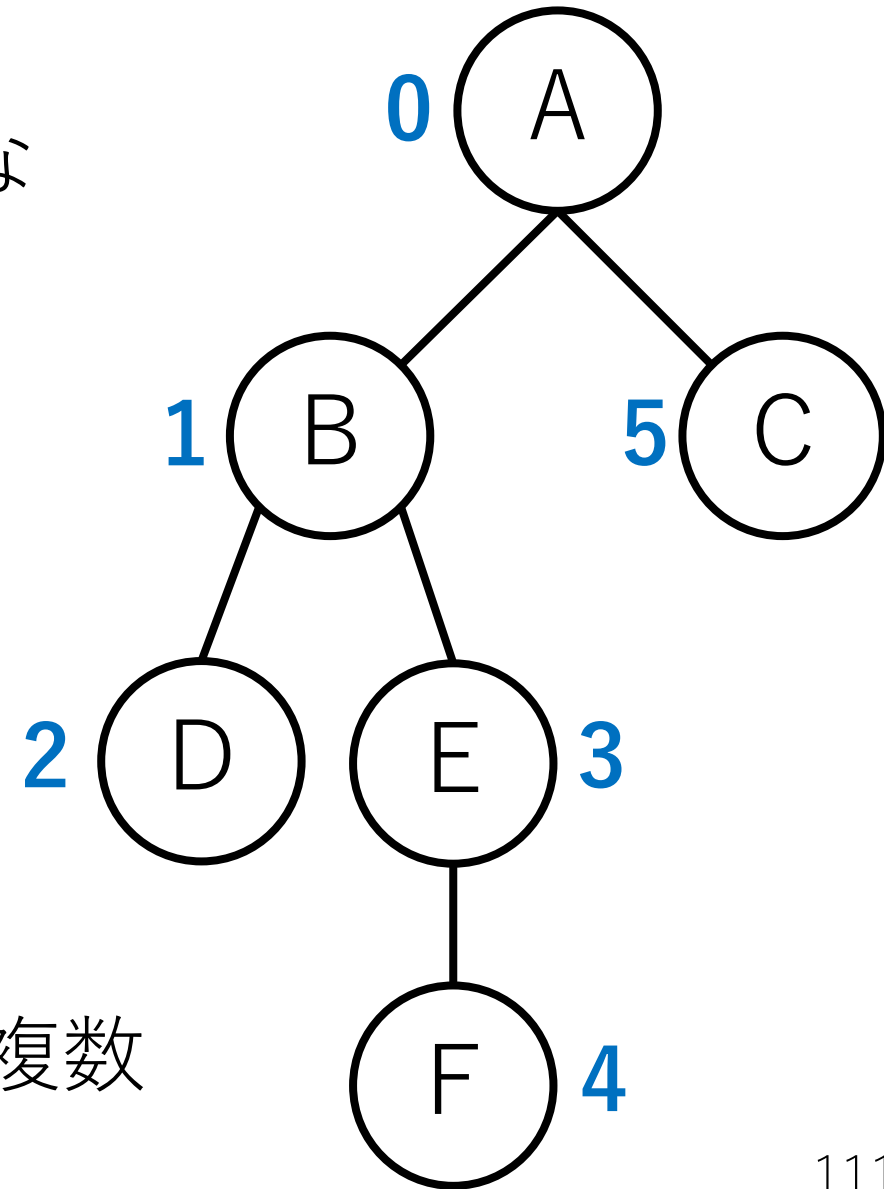
DFSが得意な問題：オイラーツアー

例えば，Aを根ノードとし，右のような順番でDFSで辿ったとする。

その時，

A, B, D, B, E, F, E, B, A, C, A

と辿ったノードを順に出力すると，オイラーツアーの出力となる。



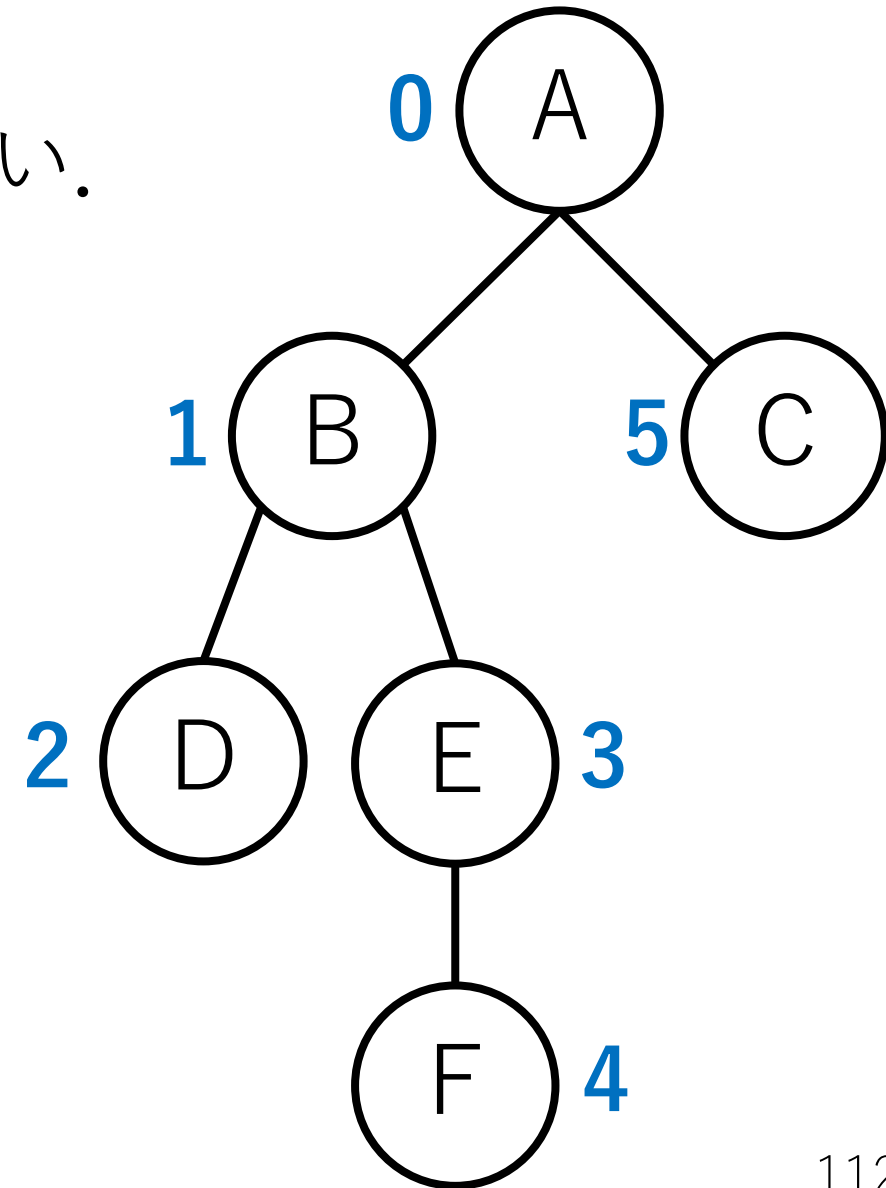
オイラーツアーはDFSの辿り方により複数存在しうる。

オイラーツアー生成手法

通ったノードを逐次出力していけば良い。

再帰を使ったDFSで実装するのが簡単。

上から通ってきた時としたから戻ってきた時両方で出力をしないといけないことに注意。



オイラーツアー生成手法の実装例

```
tree = [ # グラフのときと違い, 子ノードの情報のみ保持.  
[1, 2], #ノードA  
[3, 4], #ノードB  
[], #ノードC  
[], #ノードD  
[5], #ノードE  
[] #ノードF  
]  
node_name = ['A', 'B', 'C', 'D', 'E', 'F'] # 本質的には不要  
tour = []
```

オイラーツアー生成手法の実装例

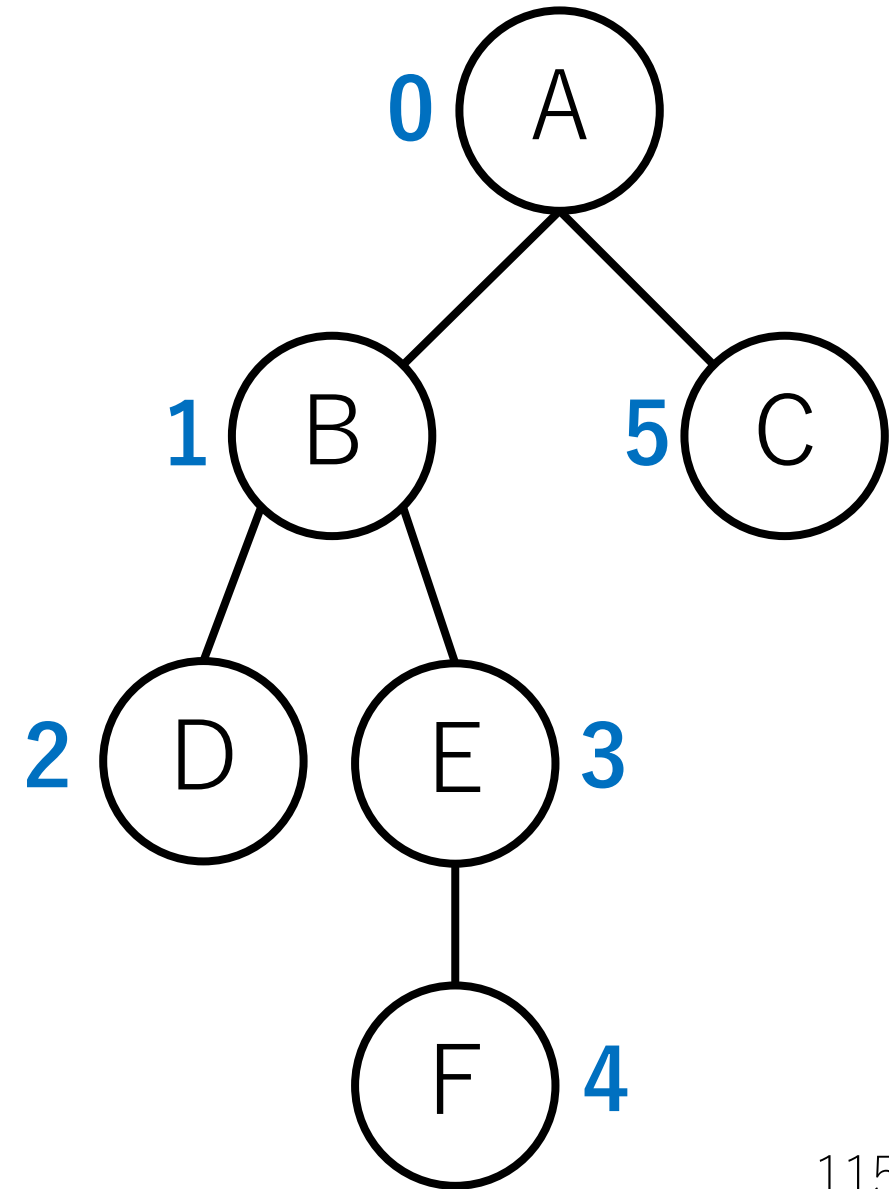
```
def eulertour(cur_node, prev_node):  
    # 初めてノードを通るときに記録  
    tour.append(node_name[cur_node])  
    for next_node in tree[cur_node]:  
        if next_node != prev_node:  
            eulertour(next_node, cur_node)  
    # 戻ってきて通るときに記録  
    tour.append(node_name[cur_node])
```

オイラーツアー生成手法の実行例

```
eulertour(0, -1)  
print(tour)
```

====出力結果====

```
['A', 'B', 'D', 'B', 'E', 'F', 'E', 'B', 'A', 'C', 'A']
```



オイラーツアー生成手法の計算量

木のノードの数を $|V|$ とする.

木なので, 辺の数は $|V| - 1$.

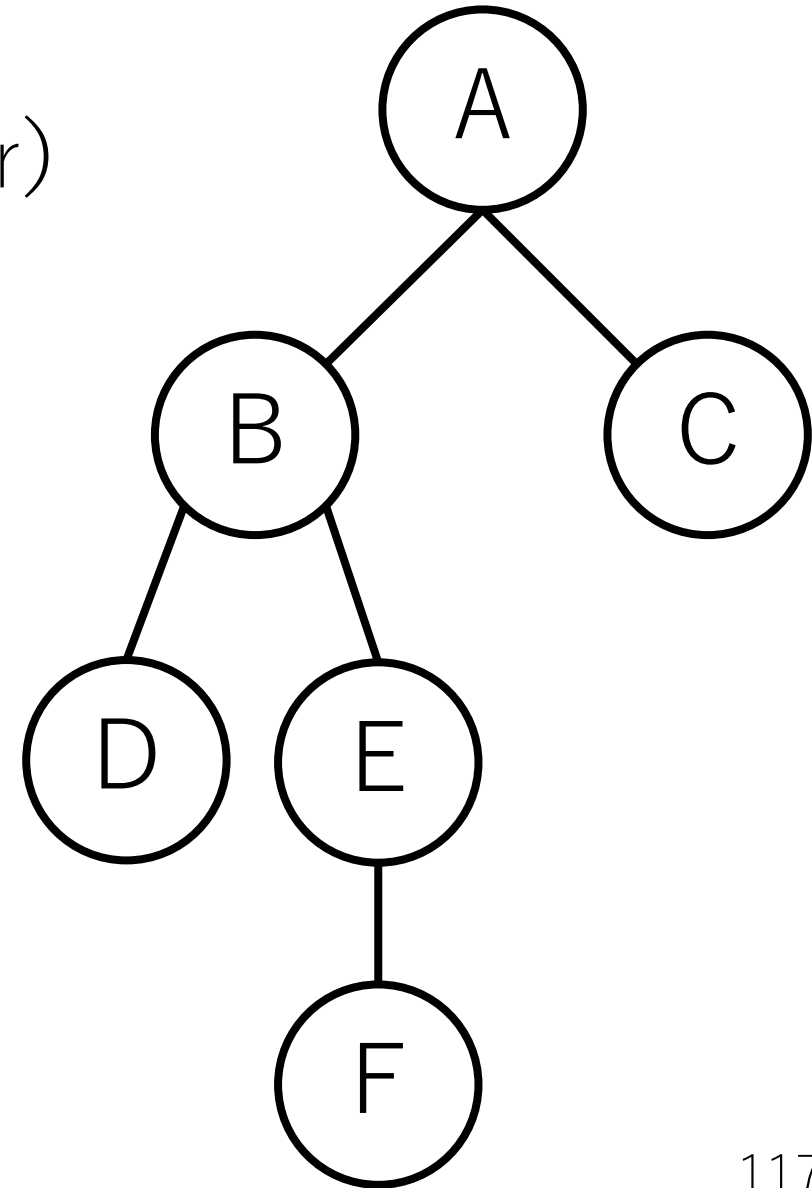
上記実装例では, 処理の内容はDFSと同じであり,
 $O(|V|)$.

オイラーツアアの応用：LCA

最近共通祖先 (Lowest Common Ancestor)
最小共通祖先とも。

木において、ある2つのノードから根ノードに向かって辿っていたとき、最初に合流するノードのこと。

例えば、右の木において、DとFのLCAはBとなる。



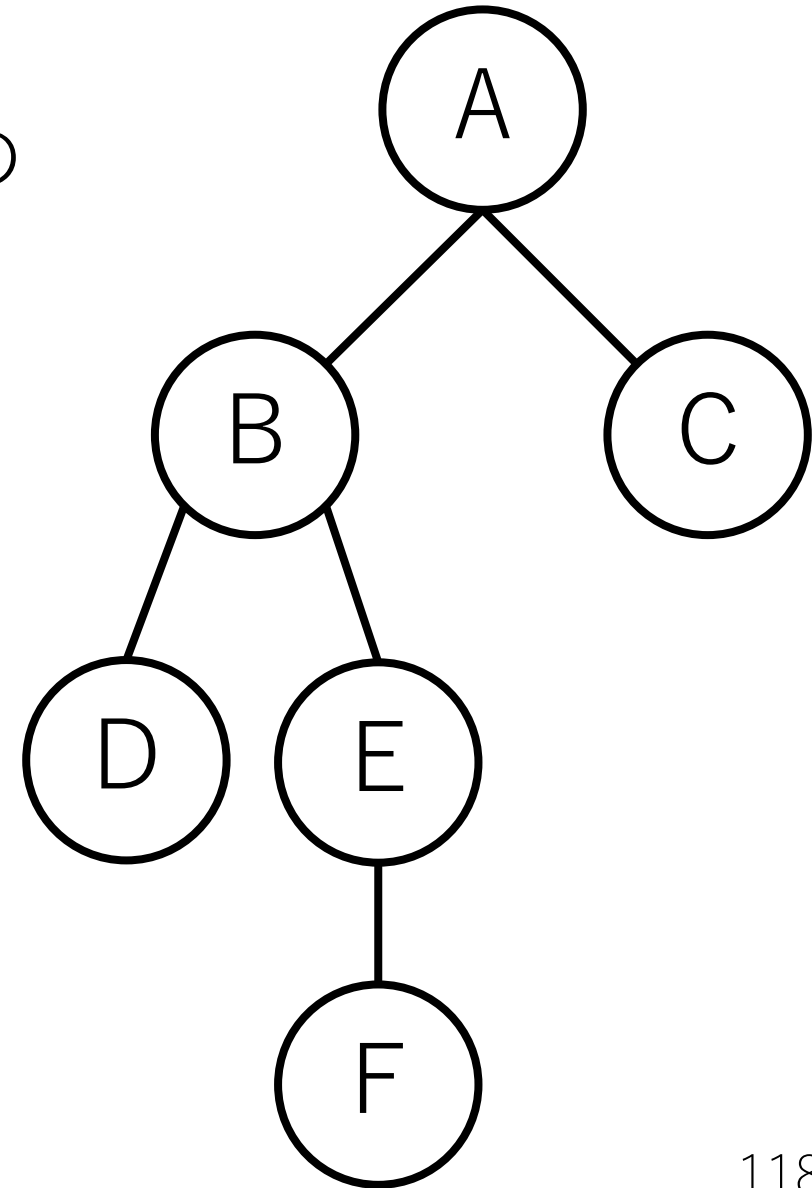
オイラーツアアの応用：LCA

LCAが高速に求めれば，2つのノード間の距離（最短のルートでの辺の数）を以下の式ですぐに求めることができる。

[あるノードからLCAまでの距離] +
[もう1つのノードからLCAまでの距離]

DからFまでの距離は，

$$[D \rightarrow B] + [F \rightarrow B] = 3$$

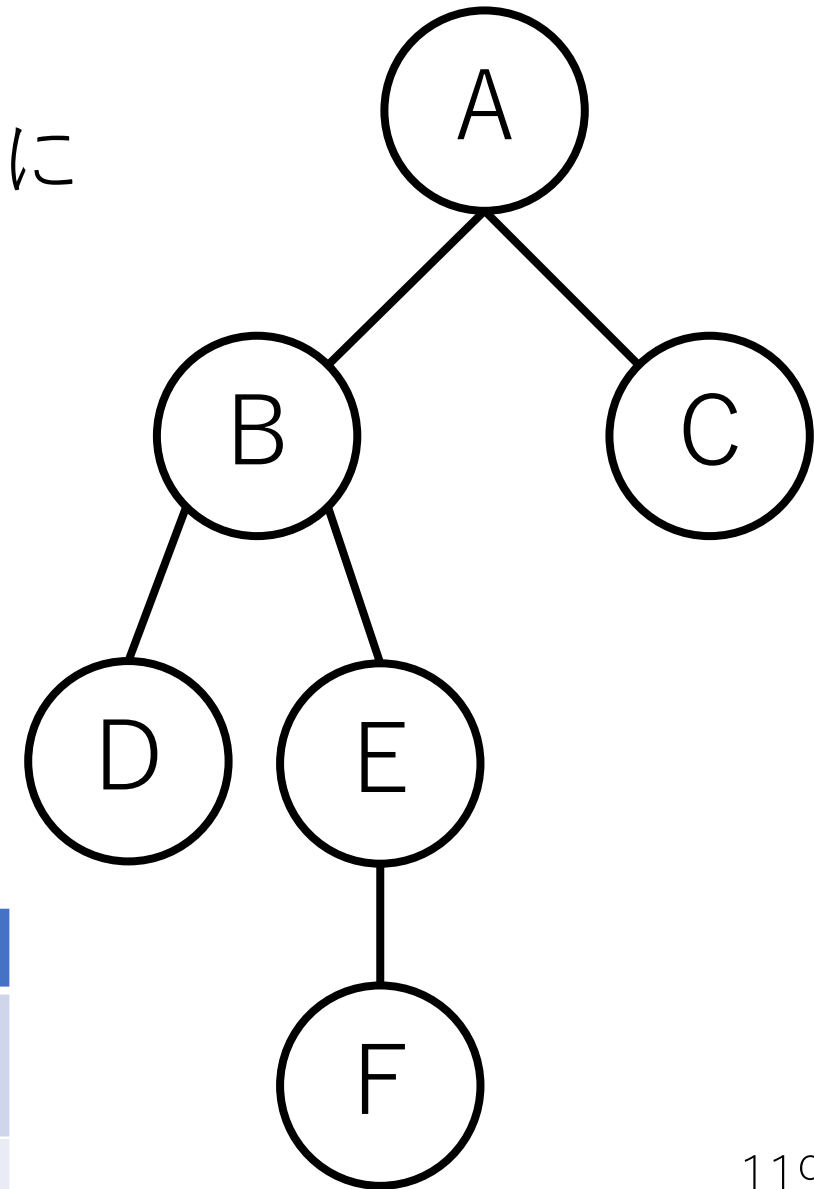


LCAの求め方

木のオイラーツアーを求める。そのときに同時に各ノードの深さも記録する。

これをまとめた配列を作る。

以下のような表ができるイメージ。

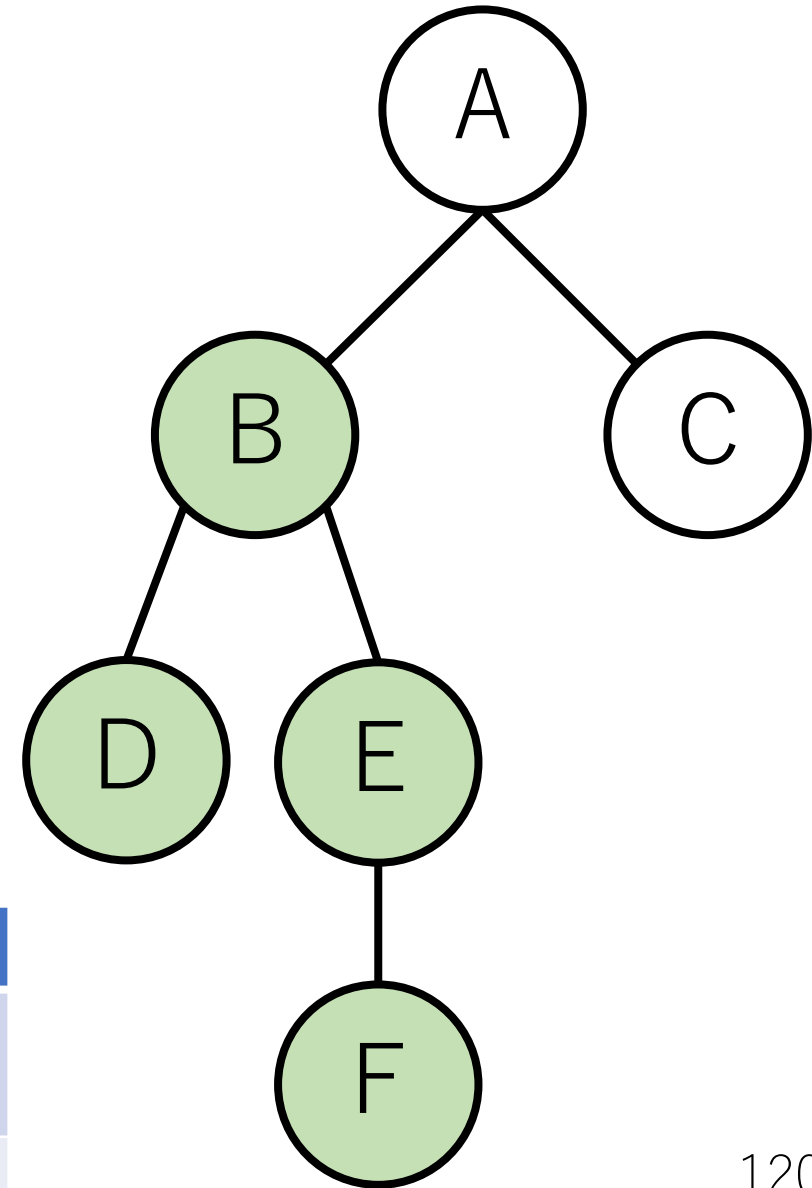


時刻	0	1	2	3	4	5	6	7	8	9	10
オイラーツアー	A	B	D	B	E	F	E	B	A	C	A
深さ	0	1	2	1	2	3	2	1	0	1	0

LCAの求め方

この表において、今対象にしている2つノードが最初に現れる場所を見つける。

下の緑色部分と右図の緑色の部分木がちょうど対応していることがわかる。

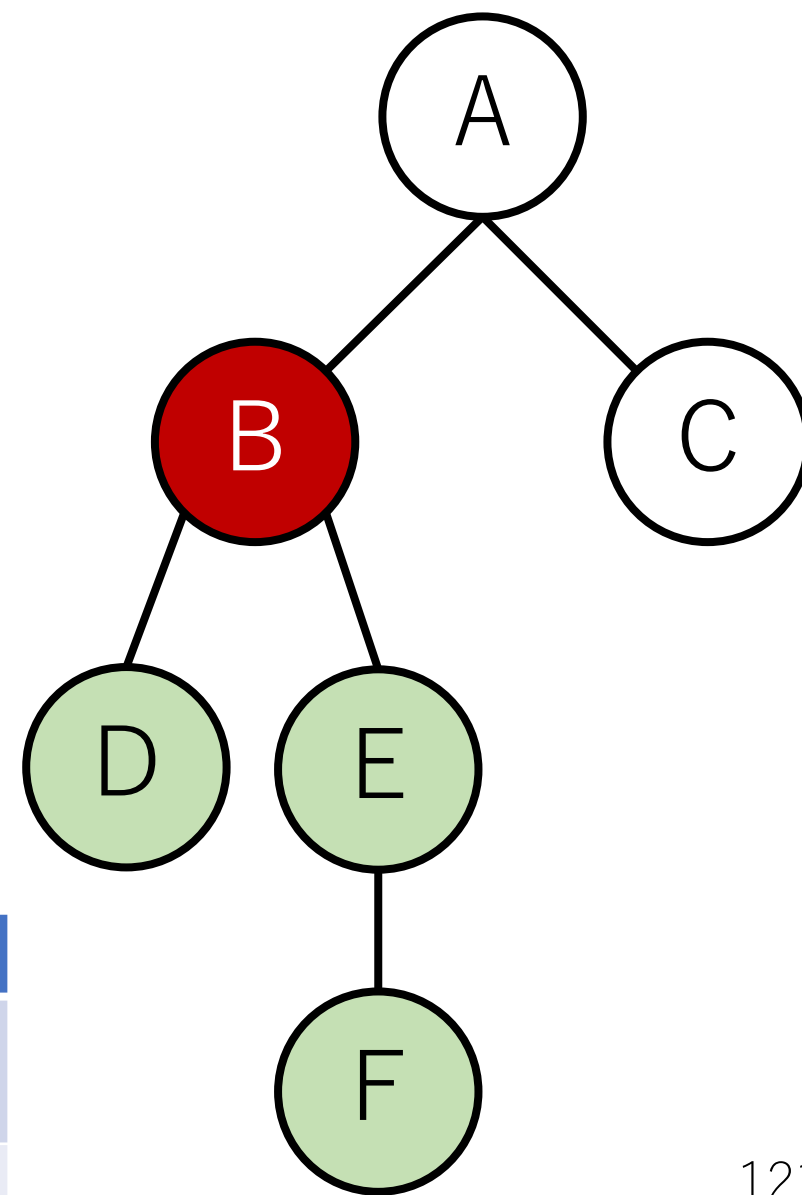


時刻	0	1	2	3	4	5	6	7	8	9	10
オイラー ツアー	A	B	D	B	E	F	E	B	A	C	A
深さ	0	1	2	1	2	3	2	1	0	1	0

LCAの求め方

LCAはこの部分木において、最も深さの浅いものとなる。

つまり、B.

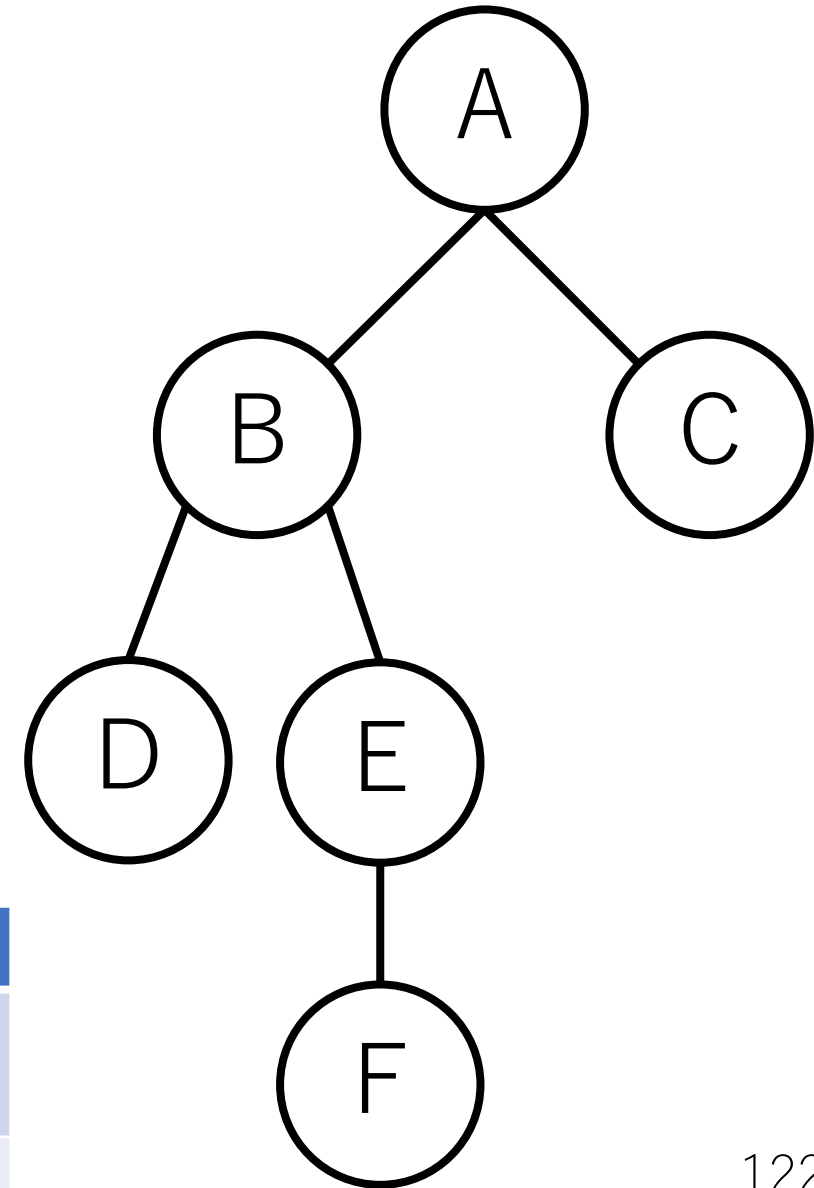


時刻	0	1	2	3	4	5	6	7	8	9	10
オイラー ツアー	A	B	D	B	E	F	E	B	A	C	A
深さ	0	1	2	1	2	3	2	1	0	1	0

LCAの求め方

オイラーツアーにより生成された配列の部分列は、ある部分木に必ず対応する。

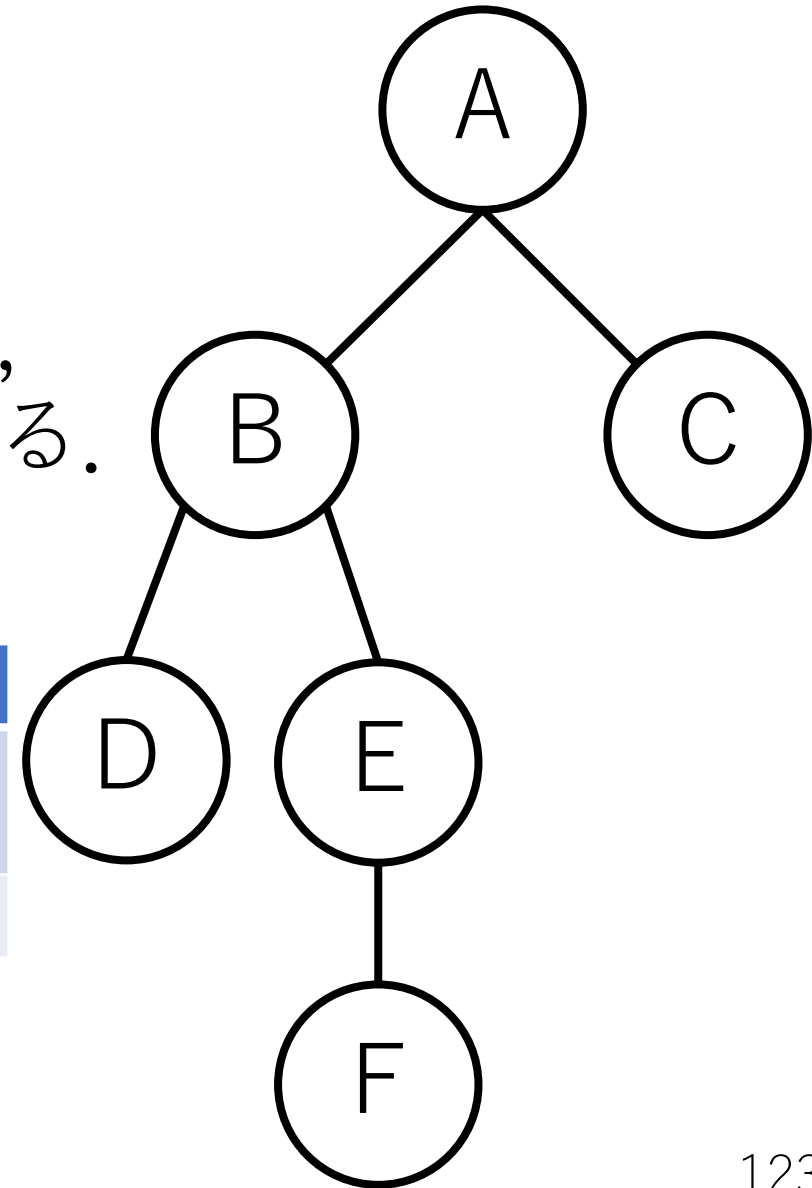
さらに、ある2つのノードを含む最小の部分木を取り出すためには、その2つのノードを初めて訪れた時刻の間を見れば良い。



時刻	0	1	2	3	4	5	6	7	8	9	10
オイラーツアー	A	B	D	B	E	F	E	B	A	C	A
深さ	0	1	2	1	2	3	2	1	0	1	0

LCAの求め方

各ノードを最初に訪れた時刻を保持する配列を別に作っておき，それを参照するように実装すると，木に変更がない限り，オイラーツアーの表を簡単に再利用できる。



時刻	0	1	2	3	4	5	6	7	8	9	10
オイラーツアー	A	B	D	B	E	F	E	B	A	C	A
深さ	0	1	2	1	2	3	2	1	0	1	0

ノード	A	B	C	D	E	F
初訪問時刻	0	1	9	2	4	5

LCAを求める手法の実装例

```
tour = []; depth = []; firstVisit = [-1]*6
```

```
tree = [
```

```
[1, 2], #ノードA
```

```
[3, 4], #ノードB
```

```
[], #ノードC
```

```
[], #ノードD
```

```
[5], #ノードE
```

```
[] #ノードF
```

```
]
```

```
node_name = ['A', 'B', 'C', 'D', 'E', 'F']
```

LCAを求める手法の実装例

```
def eulertour(cur_node, prev_node, d): # 深さも渡す

    # 最初に訪れた時刻を保持する配列を更新
    if firstVisit[cur_node] == -1:
        firstVisit[cur_node] = len(tour)
    tour.append(cur_node)
    depth.append(d) # 深さも記録
```

LCAを求める手法の実装例

```
def eulertour(cur_node, prev_node, d):  
    ...  
    for next_node in tree[cur_node]:  
        if next_node != prev_node:  
            eulertour(next_node, cur_node, d+1)  
            tour.append(cur_node)  
            depth.append(d)      # 深さも記録
```

LCAを求める手法の実装例

```
def lca(n1, n2):
    if firstVisit[n1] > firstVisit[n2]: n1, n2 = n2, n1
    lca_index = -1; lca_depth = 10**6 # 大きな数を準備

    # 最も浅いノードを線形に探索
    for i in range(firstVisit[n1], firstVisit[n2]+1):
        if depth[i] < lca_depth:
            lca_index = tour[i]; lca_depth = depth[i]

    print(node_name[lca_index])
```

LCAを求める手法の実行例

eulertour(0, -1, 0)

lca(3, 5) # DとFのLCA

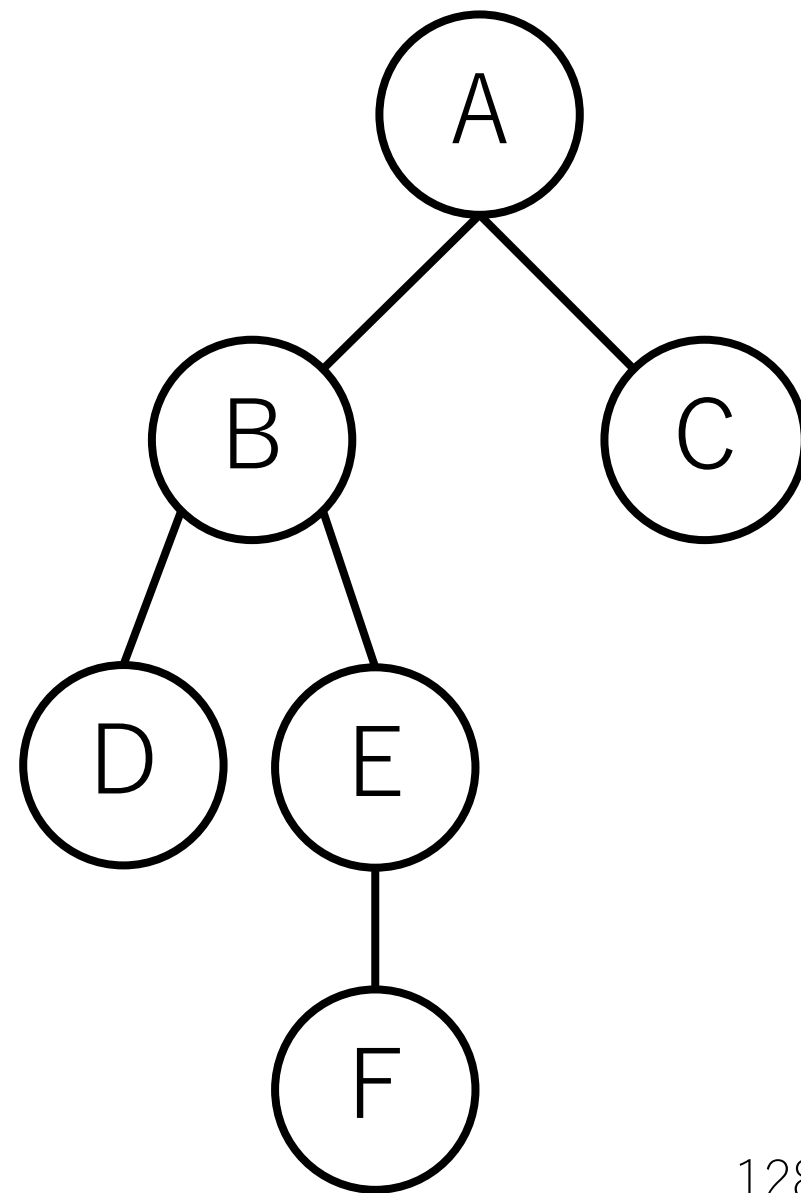
====出力結果====

B

lca(2, 5) # CとFのLCA

====出力結果====

A



LCAの計算量

上記実装の場合,

オイラーツアー： $O(|V|)$

複数回LCAを見つける必要がある場合も、ここは1回行えば良い。

LCAを探し出す： $O(|V|)$

部分列は通常 $O(|V|)$ の長さになると期待でき、それを線形に探索する。

よって、LCAの問い合わせが Q 回来る場合は、全体の計算量は $O(Q|V|)$ 。

LCAの計算量

LCAを探し出す部分でセグメント木を使うことで、計算量を落とすこともできる。

与えられた区間における深さが最も浅いノードを求めればよいので、それを保持するセグメント木を作っておく。

セグメント木の構築には $O(|V|)$ 必要だが、構築後で深さが最も浅いノードを出すのは $O(\log|V|)$ で可能。

こうすると、LCAの問い合わせが Q 回来る場合、全体の計算量を $O(|V| + Q \log|V|)$ に削減できる。

DFSが得意な問題

オイラーツアー

LCA

橋の検出（発展的内容）

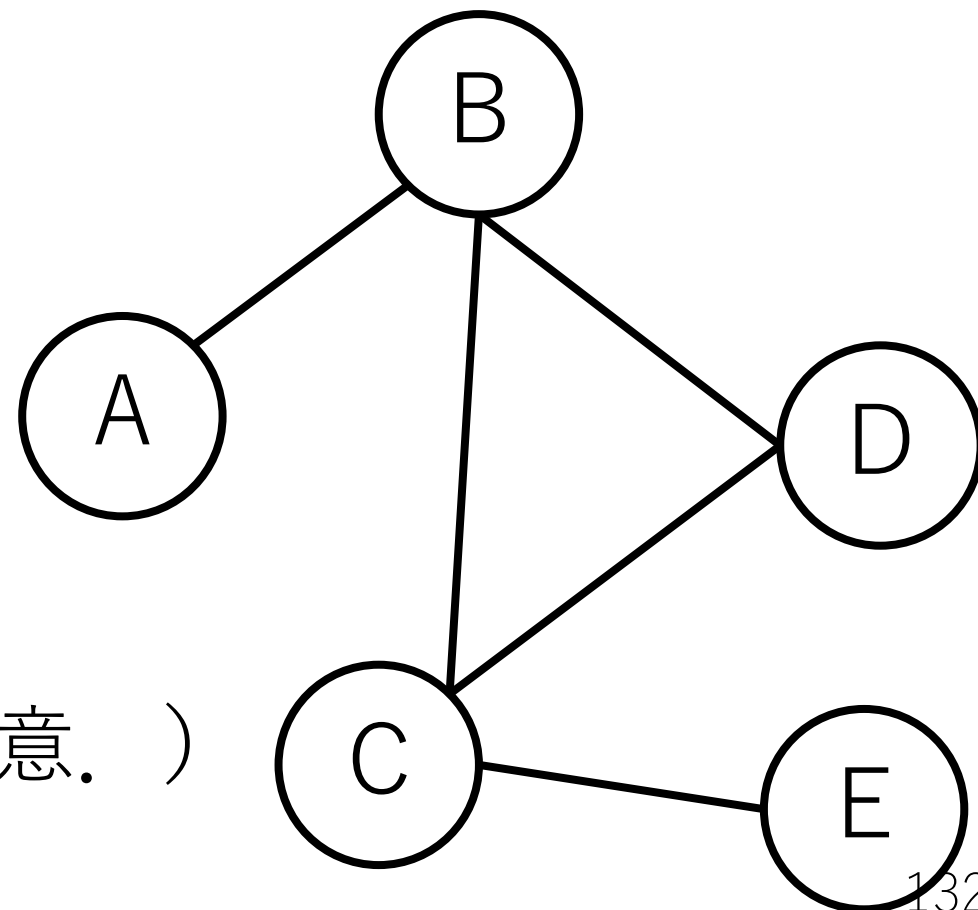
BFS, DFSを初めて勉強する方はこの前のスライドまでの内容とオイラーツアーをしっかりと掴んでもらえばOKです。特に、橋の検出の内容はちょっと腕に覚えのある人向け。



DFSが得意な問題：橋の検出

橋 = 切ってしまうとグラフ全体が非連結になる, あるいは連結成分の数が増える辺.

右のグラフではA-BとC-Eの辺が橋となる.

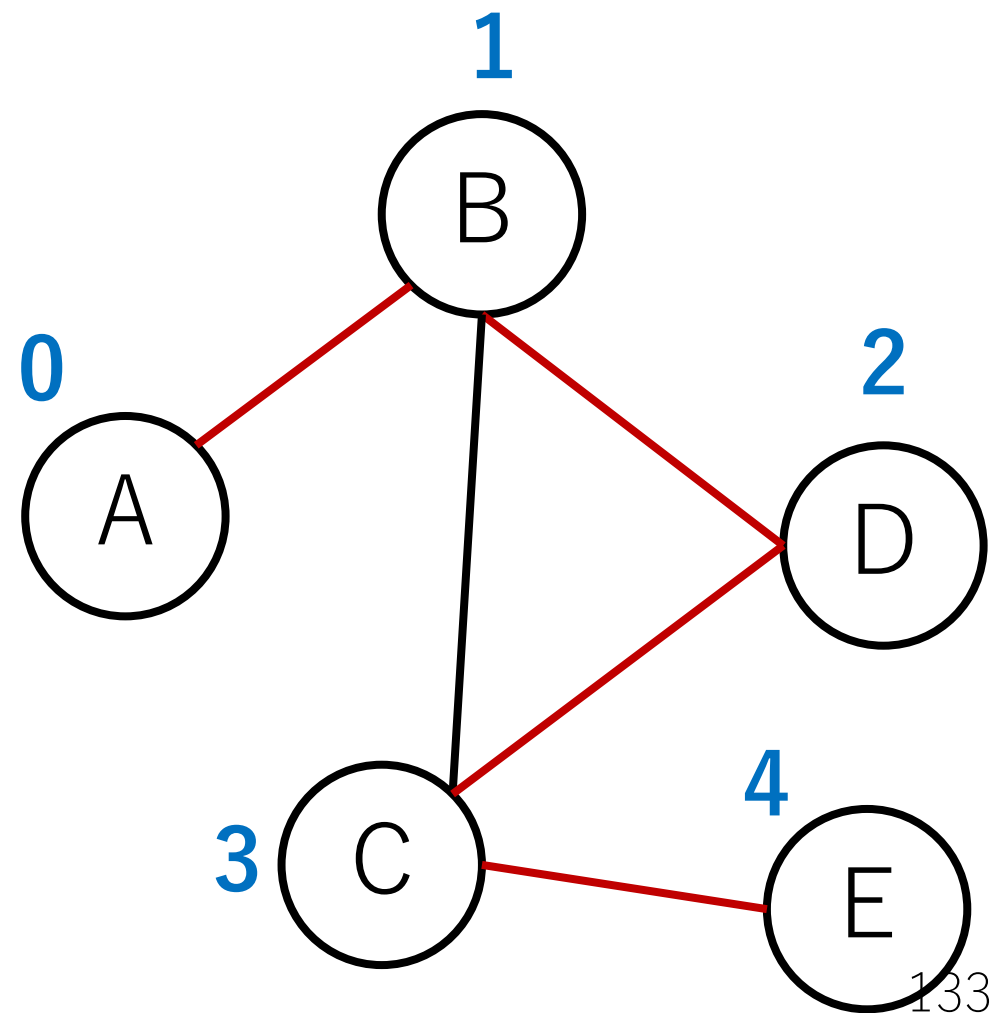


(木からグラフの話に戻ったので注意.)

橋の検出

DFSでグラフを探索.

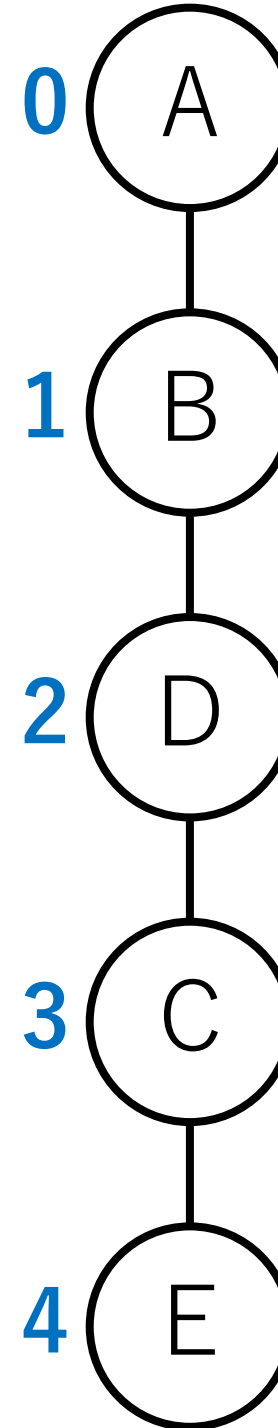
これによって訪問順が出る.



橋の検出

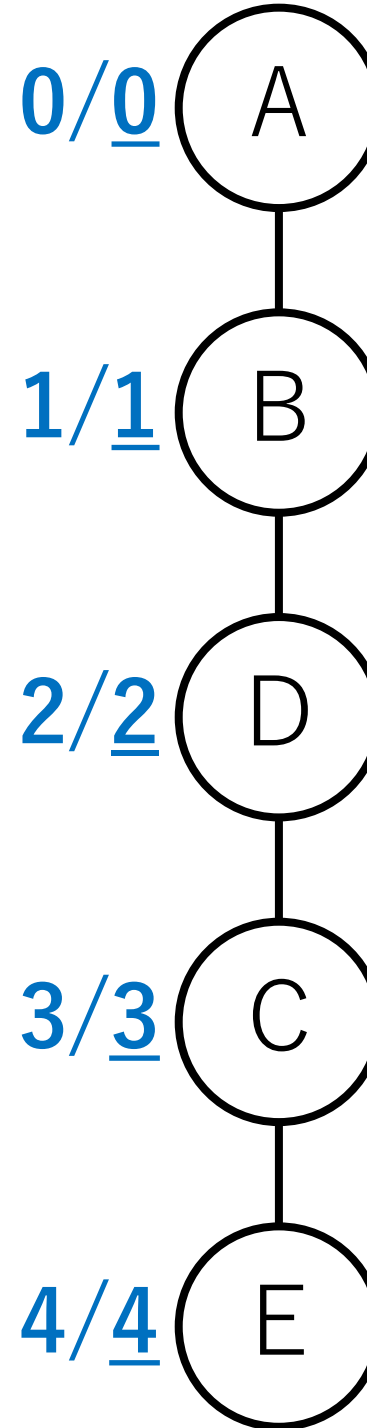
今回のDFSで辿った経路のみを残した木,
DFS木を生成する.

今回の例では右のようにたまたま一直線
になる.



橋の検出

このDFS木において，lowlinkという指標を考える（下線付きの値）．デフォルトは訪問の順番と同じ値．

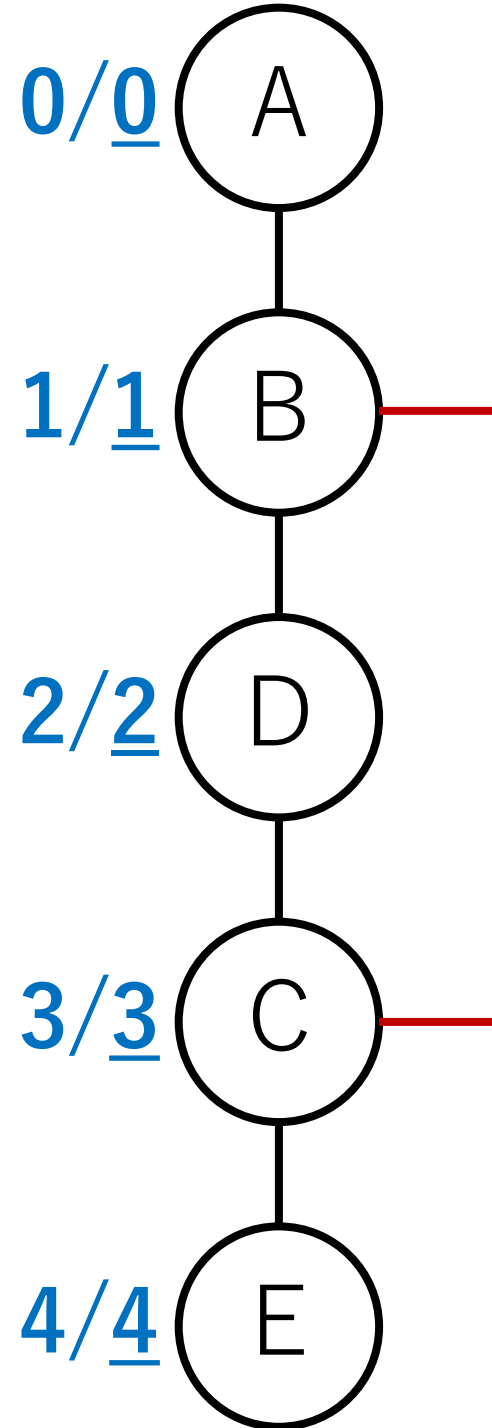


橋の検出

さらにこのDFS木において，先程の探索では通らなかった辺を追加する（赤色）。

これを後退辺と呼ぶ。

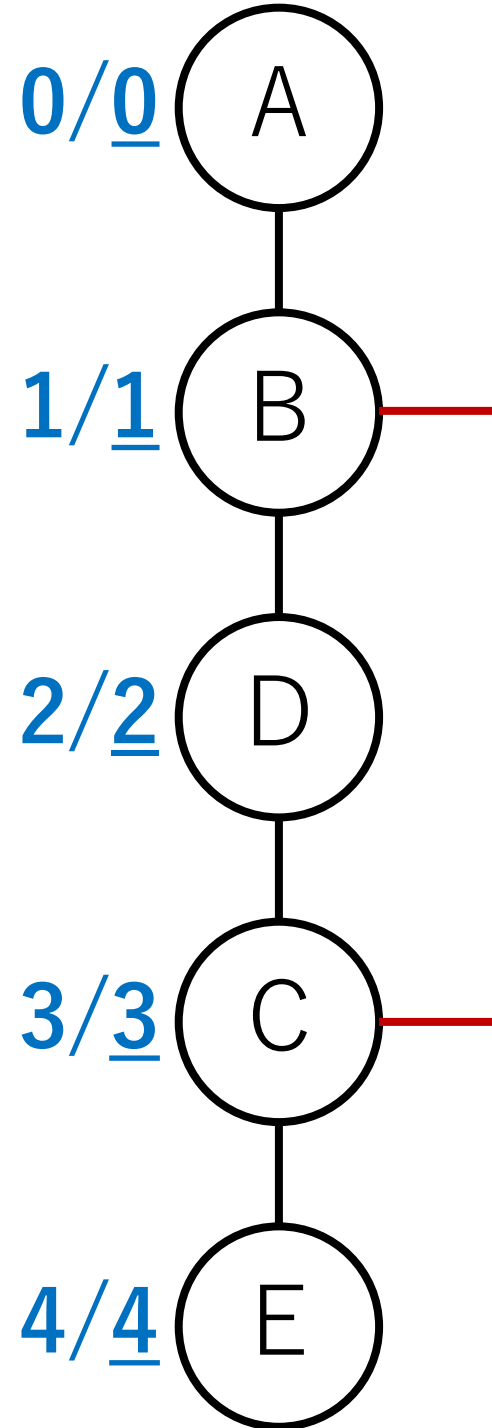
訪問順序がより前のノードに接続する，探索では使わなかった辺。



橋の検出

lowlinkの値をDFS木の葉ノードから更新していく.

自分から見て葉ノード側に移動する(何度も移動できる)か, 後退辺を1度だけ使って到達できる全てのノードの内, 最も小さい訪問順序の値を新しいlowlinkとする.

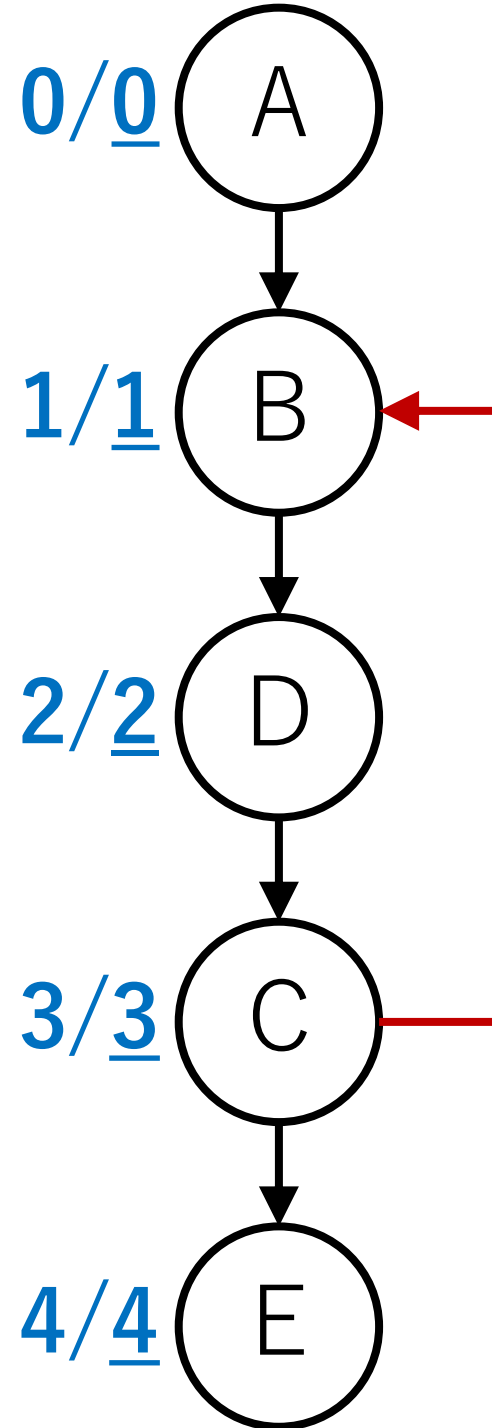


橋の検出

lowlink更新のために辿ることのできるパスのイメージは右のような感じ.

黒色（探索に使った辺）は根から葉ノード方向に，赤色（後退辺）は葉から根ノード方向に結ぶ.

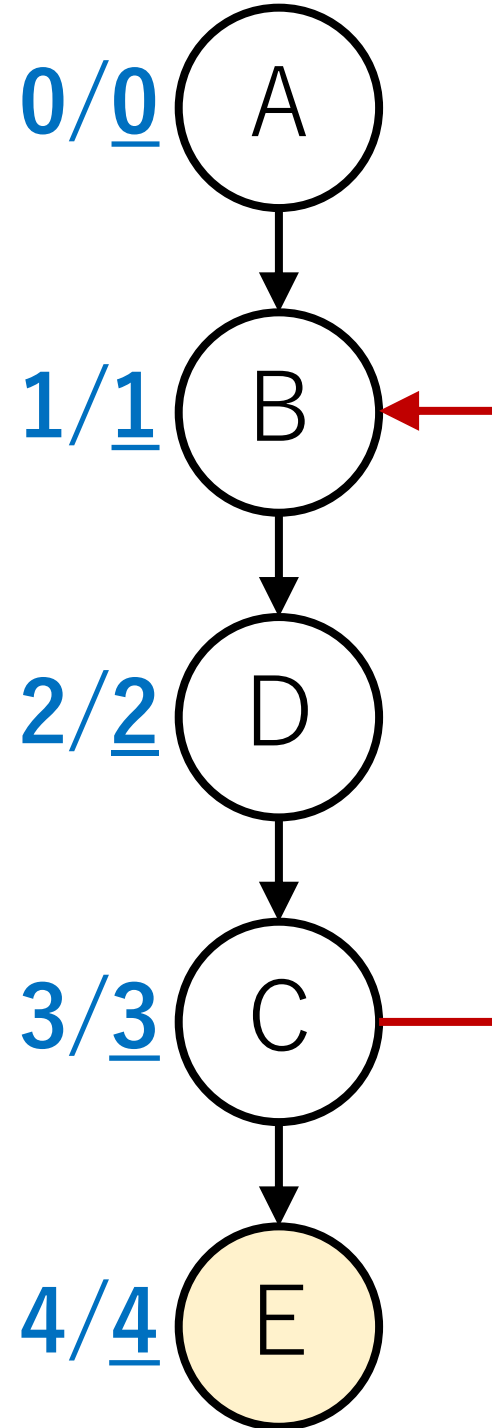
ただし，後退辺は1回しか使えない.



橋の検出

Eの場合，より葉ノードに向かう辺はなく，また，後退辺もない。

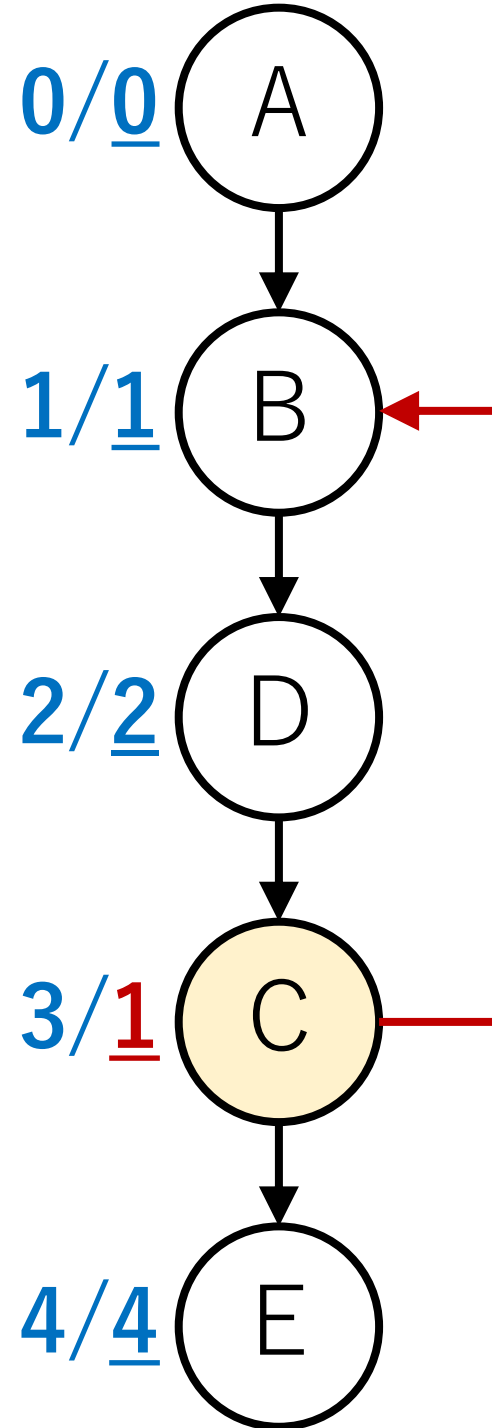
よって，ここは4そのまま。



橋の検出

Cの場合，後退辺を使うと，B（訪問順序1）に到達できる。

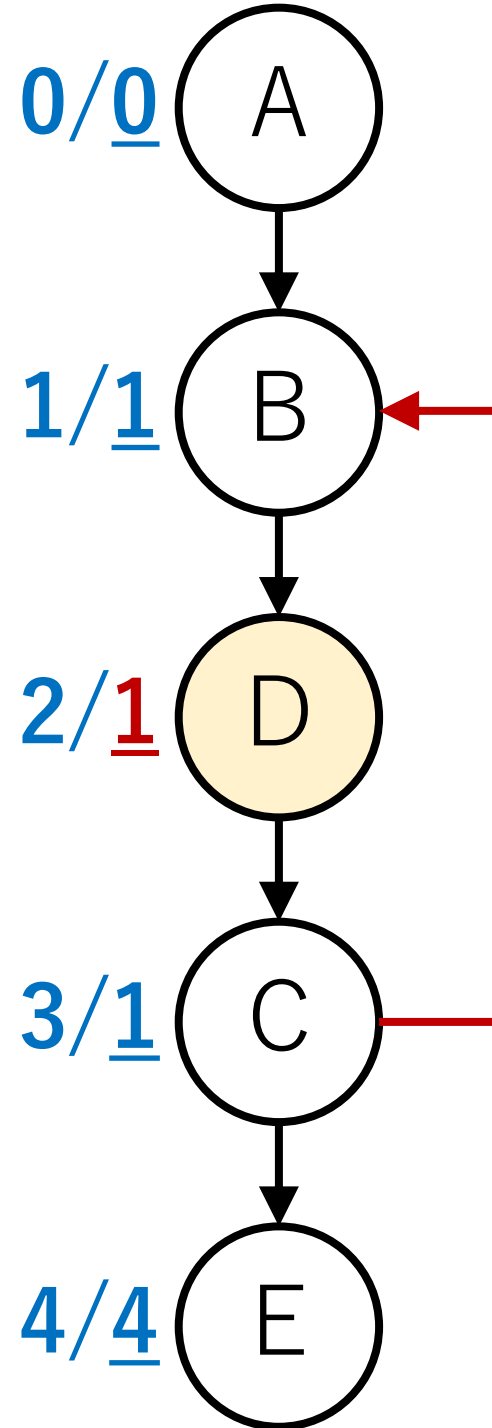
よって，ここは1に更新。



橋の検出

Dの場合, $D \rightarrow C \rightarrow B$ と辿ることが可能.

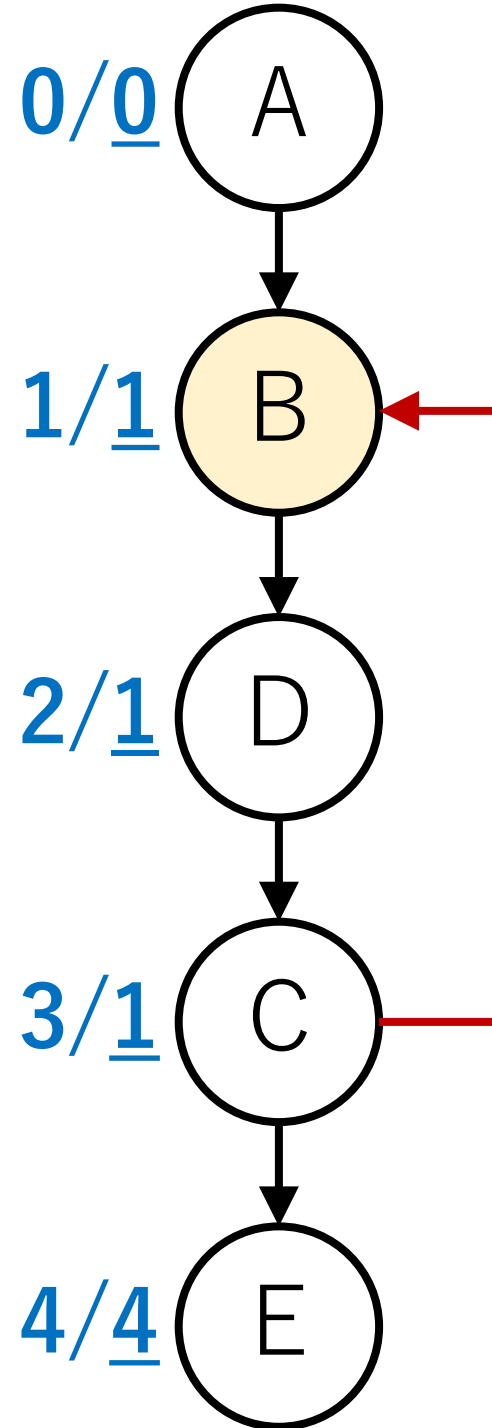
よって, こども1に更新.



橋の検出

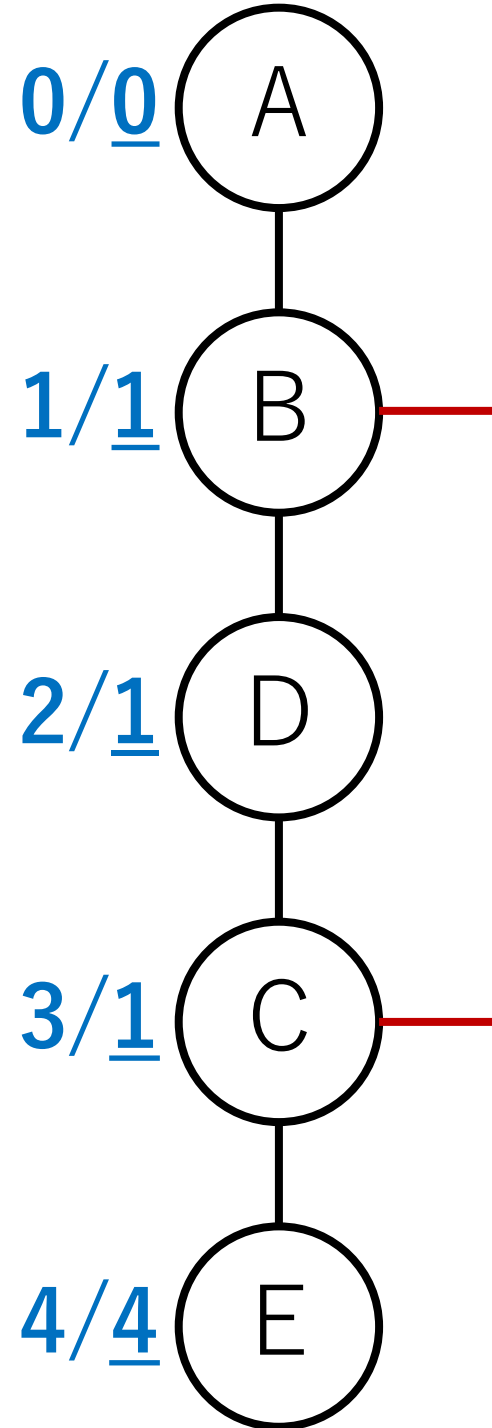
Bの場合，現在のlowlinkよりも小さな訪問順序を持つノードにはたどり着けない。

よって，ここは1のまま。



橋の検出

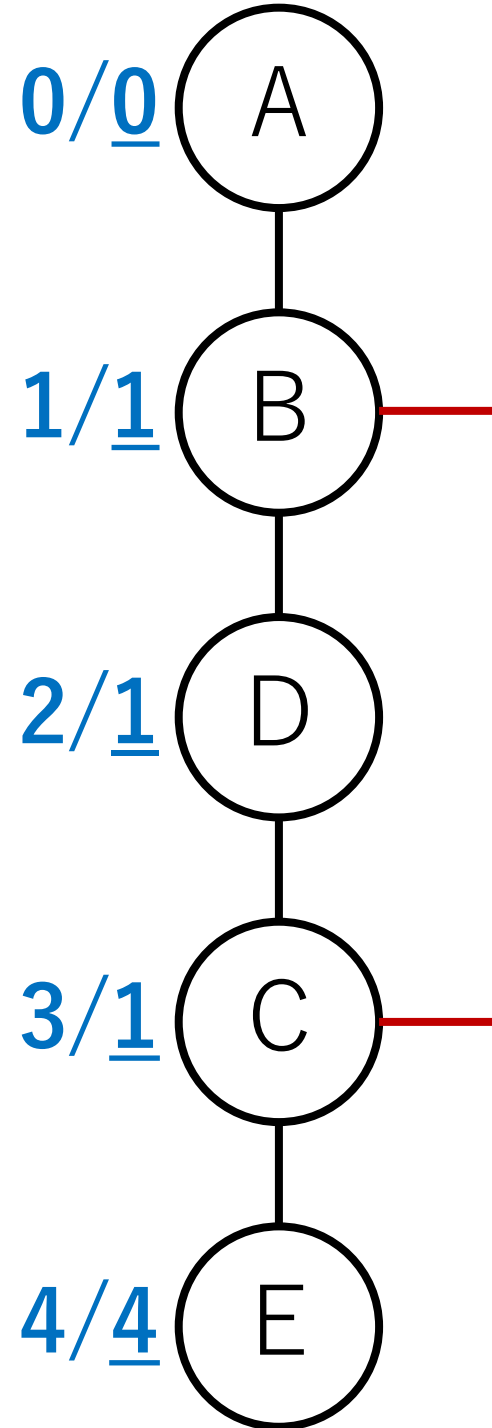
lowlinkは最終的には右のような値になる.



橋の検出

lowlinkは、探索のときに使ったパスとは別の経路で到達できるDFS木の最も根に近い（到達順序の若い）ノードを表している。

これを手がかりに橋かどうかを判断できる。

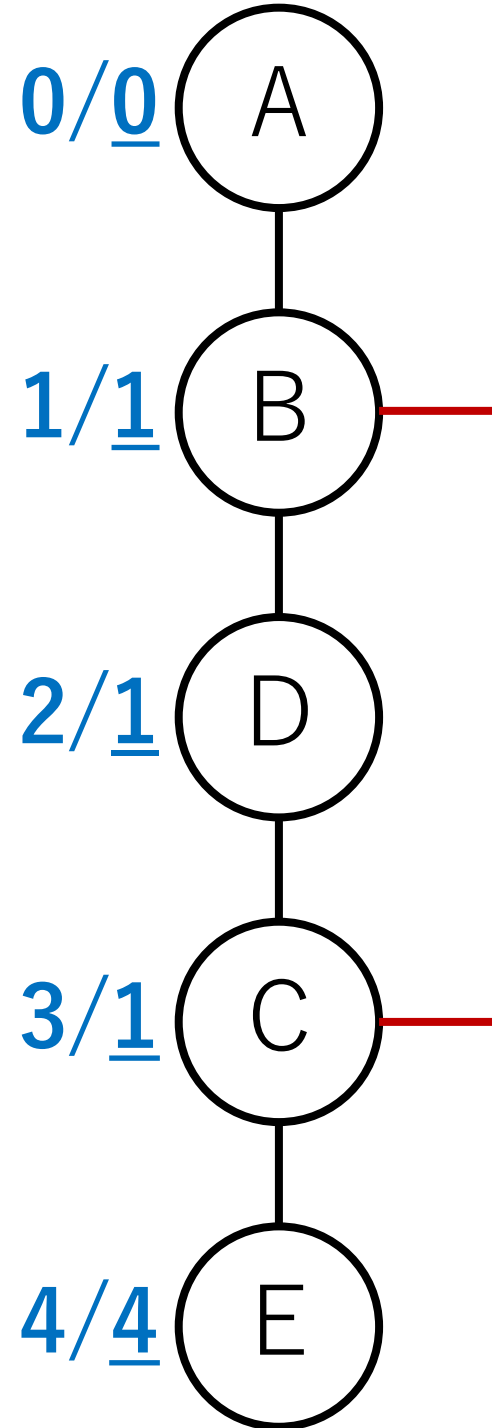


橋の検出

「別経路がない = 橋である」となる。

そして、これは以下の関係性から判断できる。

[あるノードの訪問順序]
< [その子ノードのlowlink]



橋の検出

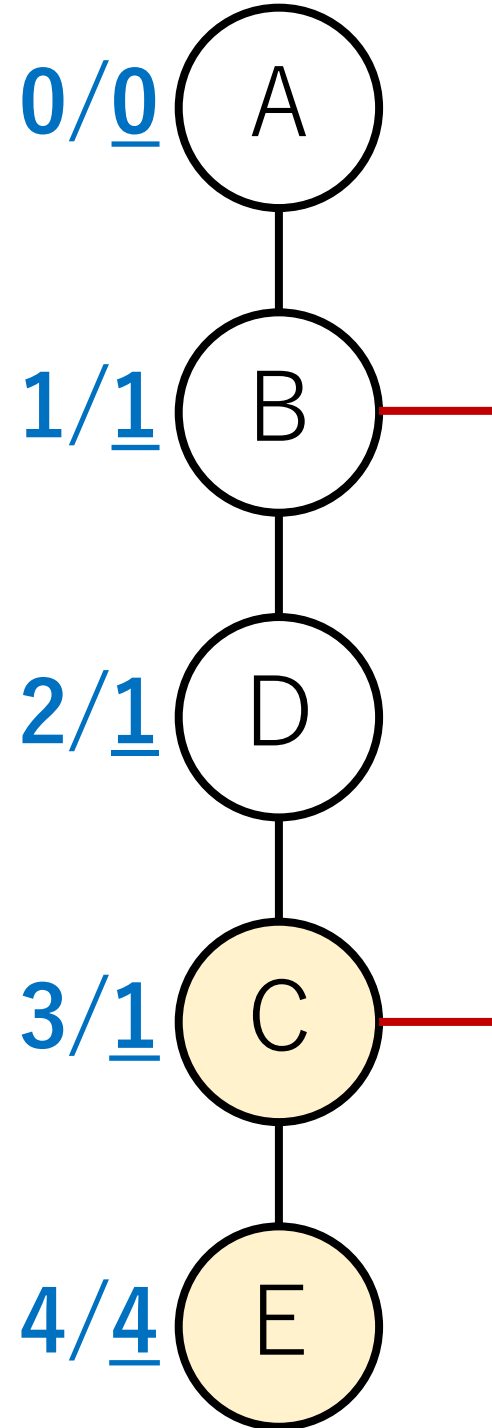
C-E間を見てみると、

[ノードCの訪問順序] : 3

[ノードEのlowlink] : 4

となり関係性が満たされている。

→ よって、ここは橋。



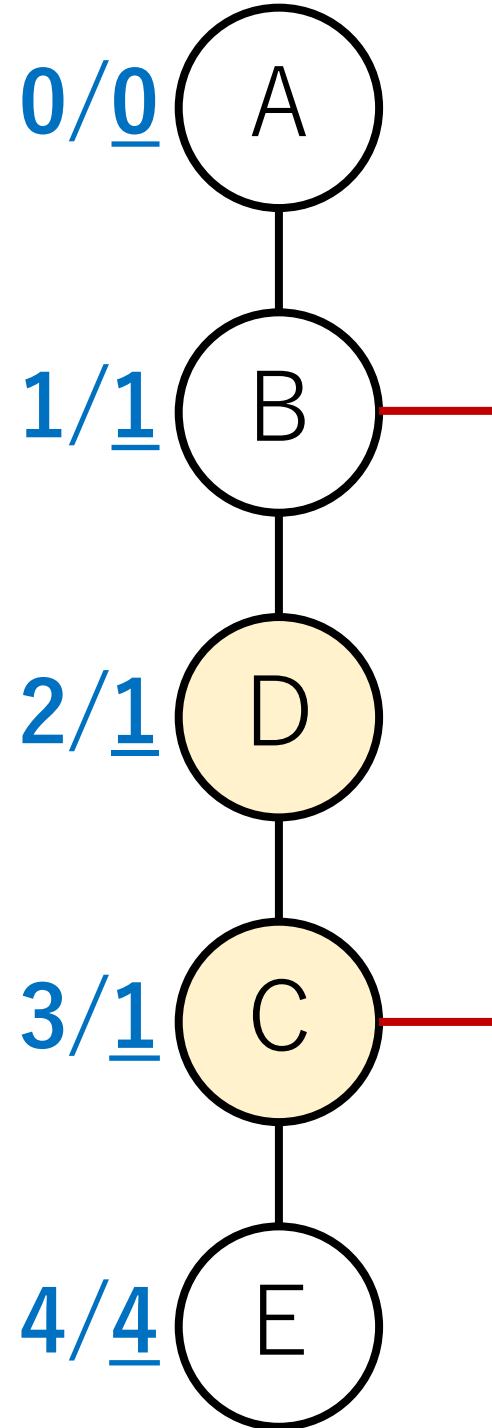
橋の検出

D-C間を見てみると,

[ノードDの訪問順序]: 2

[ノードCのlowlink]: 1

となり関係性が満たされていない。
→ よって, ここは橋ではない。



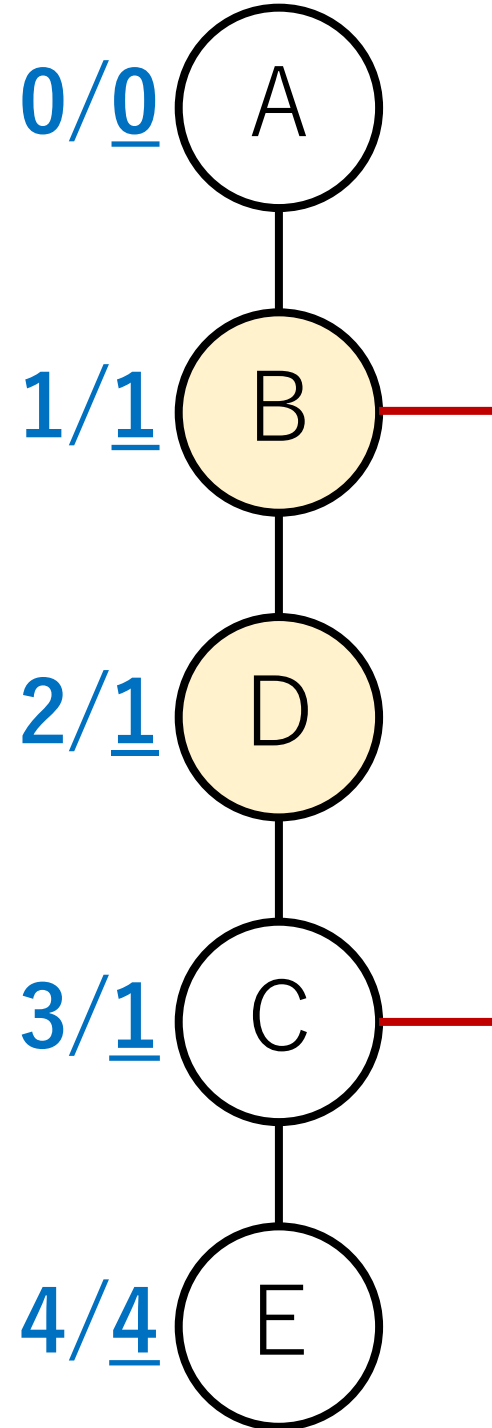
橋の検出

B-D間を見てみると,

[ノードBの訪問順序] : 1

[ノードDのlowlink] : 1

となり関係性が満たされていない。
→ よって, ここは橋ではない。



橋の検出

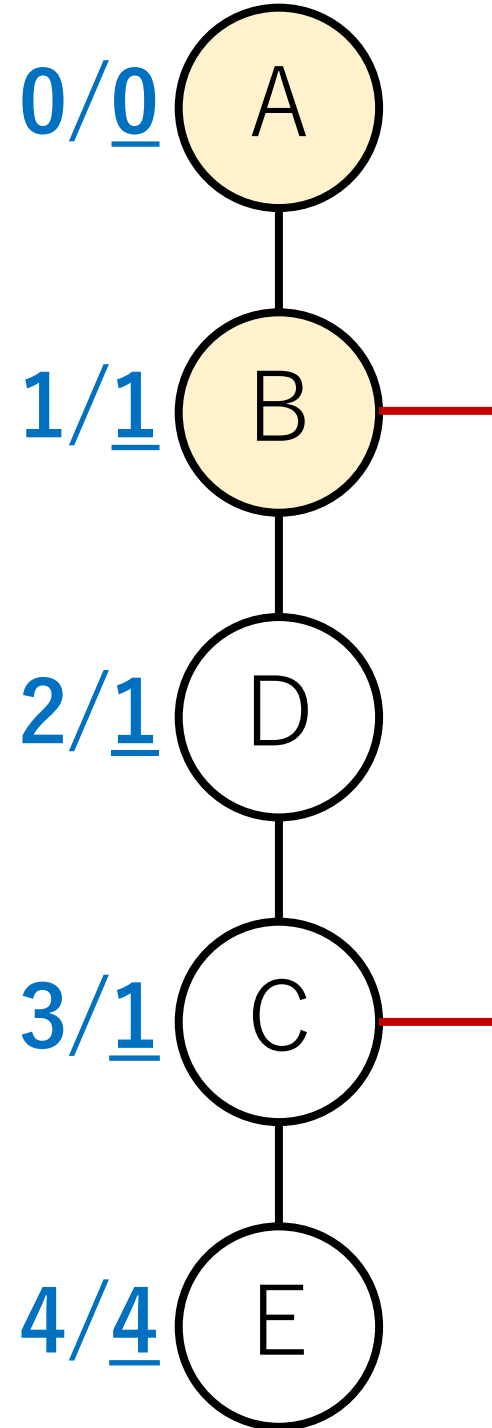
A-B間を見てみると,

[ノードAの訪問順序] : 0

[ノードBのlowlink] : 1

となり関係性が満たされている.

→ よって, ここは橋.



橋の検出

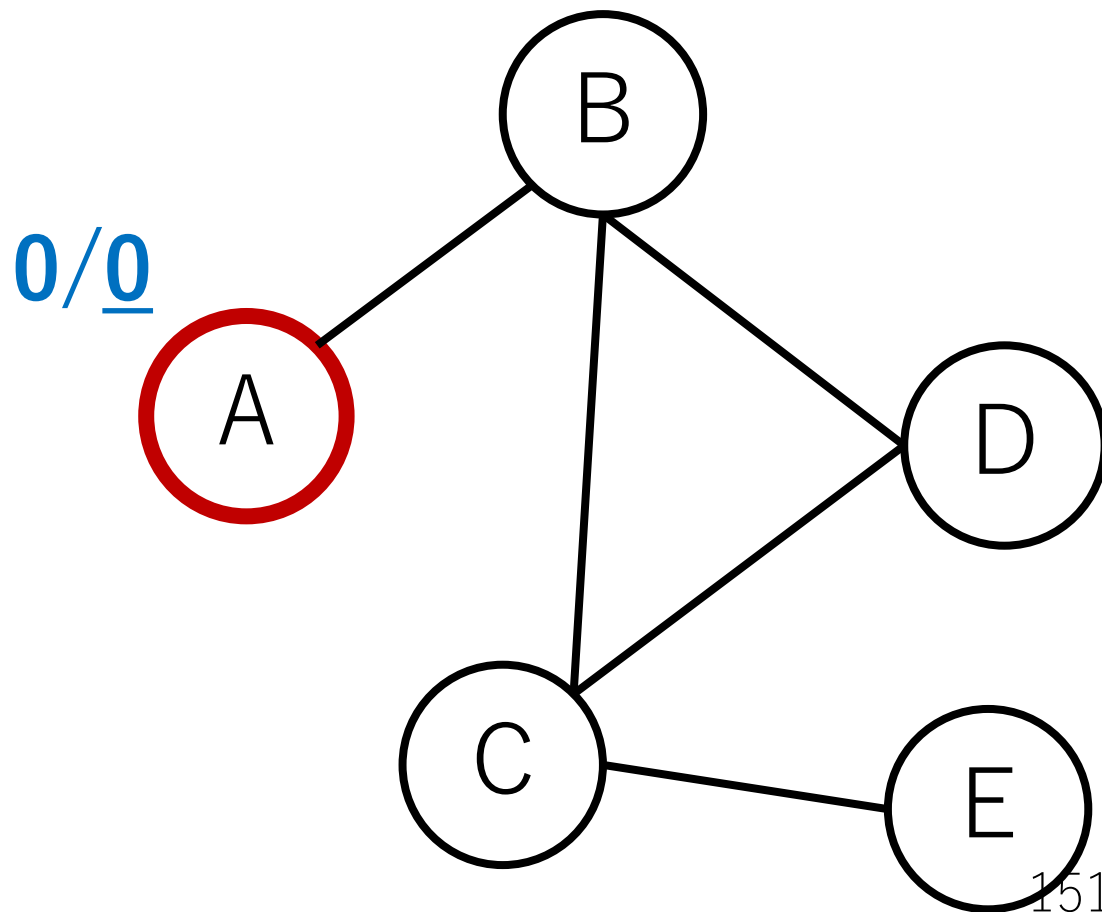
実装上は再帰版DFSで探索すると同時に、lowlinkを定数回で更新でき、同時に橋の検出をすることができる。

このため、計算量はDFSのときと変わらない。

橋の検出のアルゴリズム実行例

DFSでグラフを探索する.

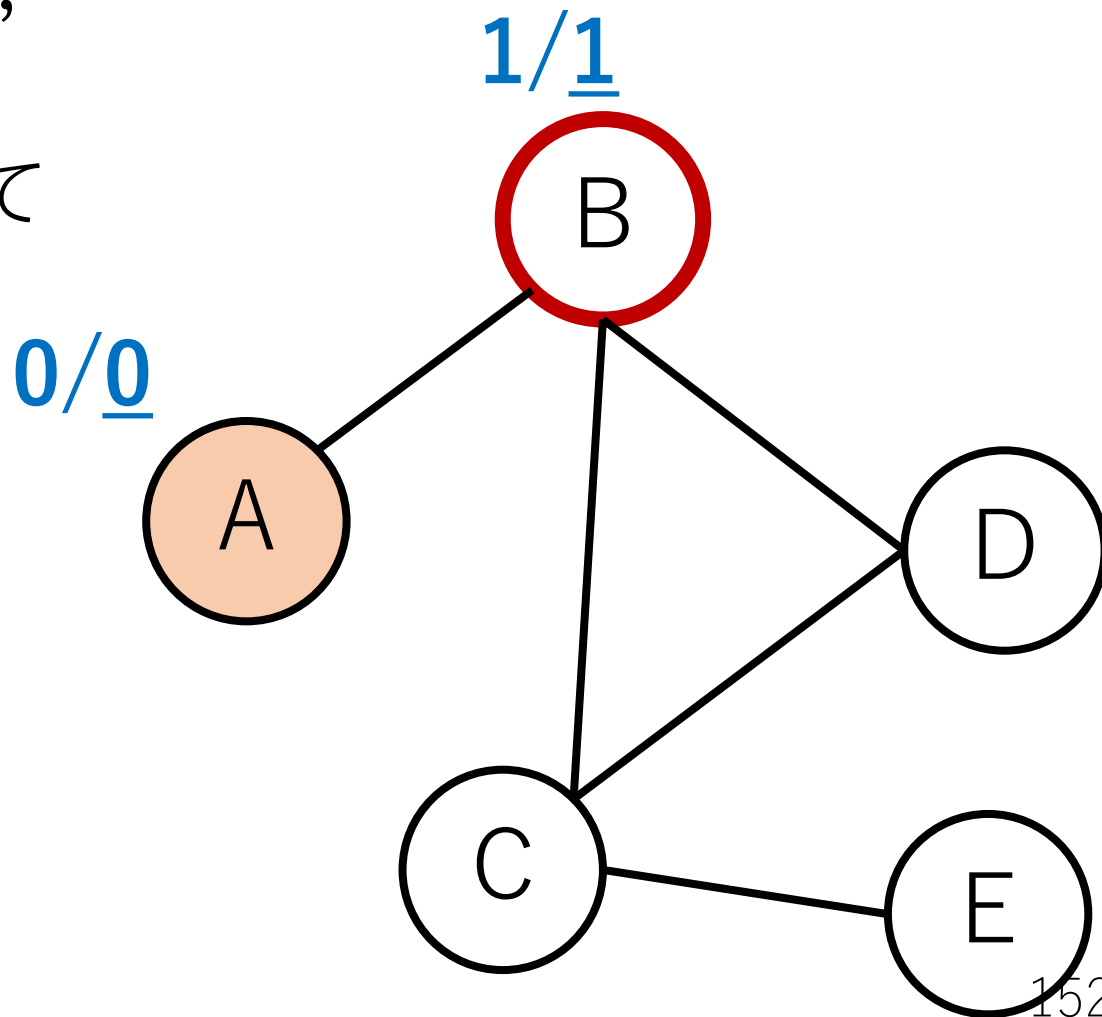
初めて訪問したノードにおいて,
とりあえずはlowlink = 訪問順序
としておく.



橋の検出のアルゴリズム実行例

Aからスタートすると仮定すると、
次のノードはBとなる。

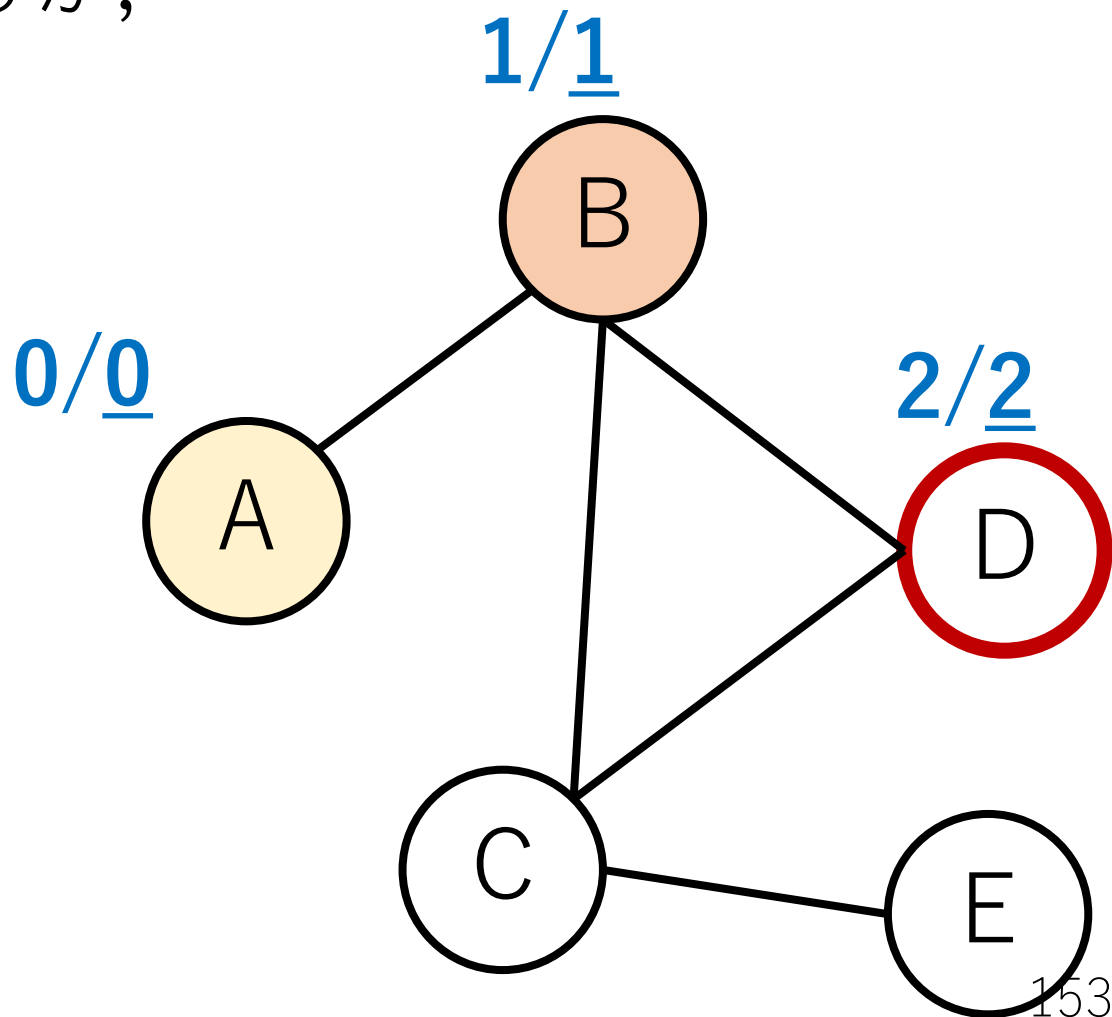
どのノードからスタートして
もよい。



橋の検出のアルゴリズム実行例

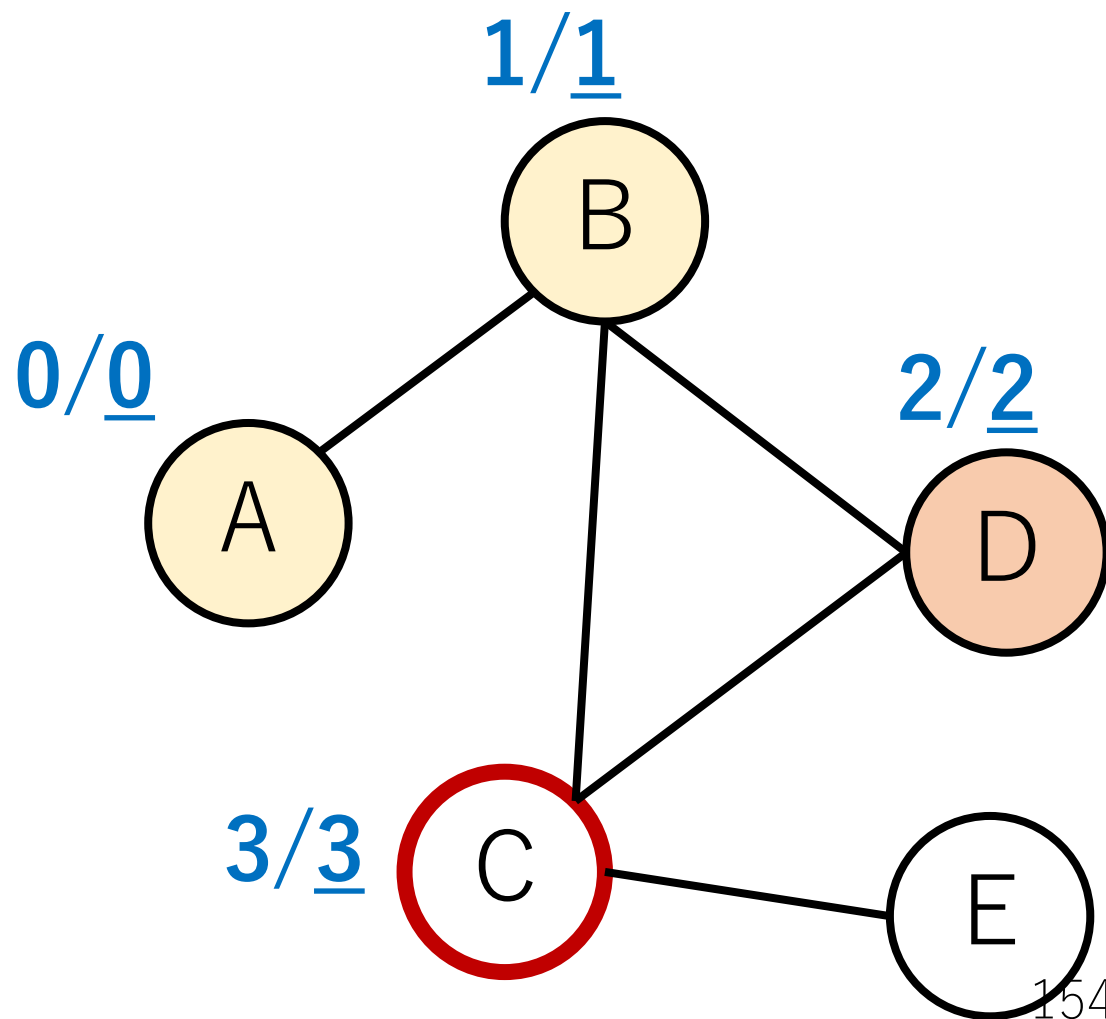
BからはCとDが次のノードとなるが、
Dに進むことにする。

直前にいたノードは今後の
プロセスでも無視する



橋の検出のアルゴリズム実行例

DからはCにしか進めないので、
Cに進む。

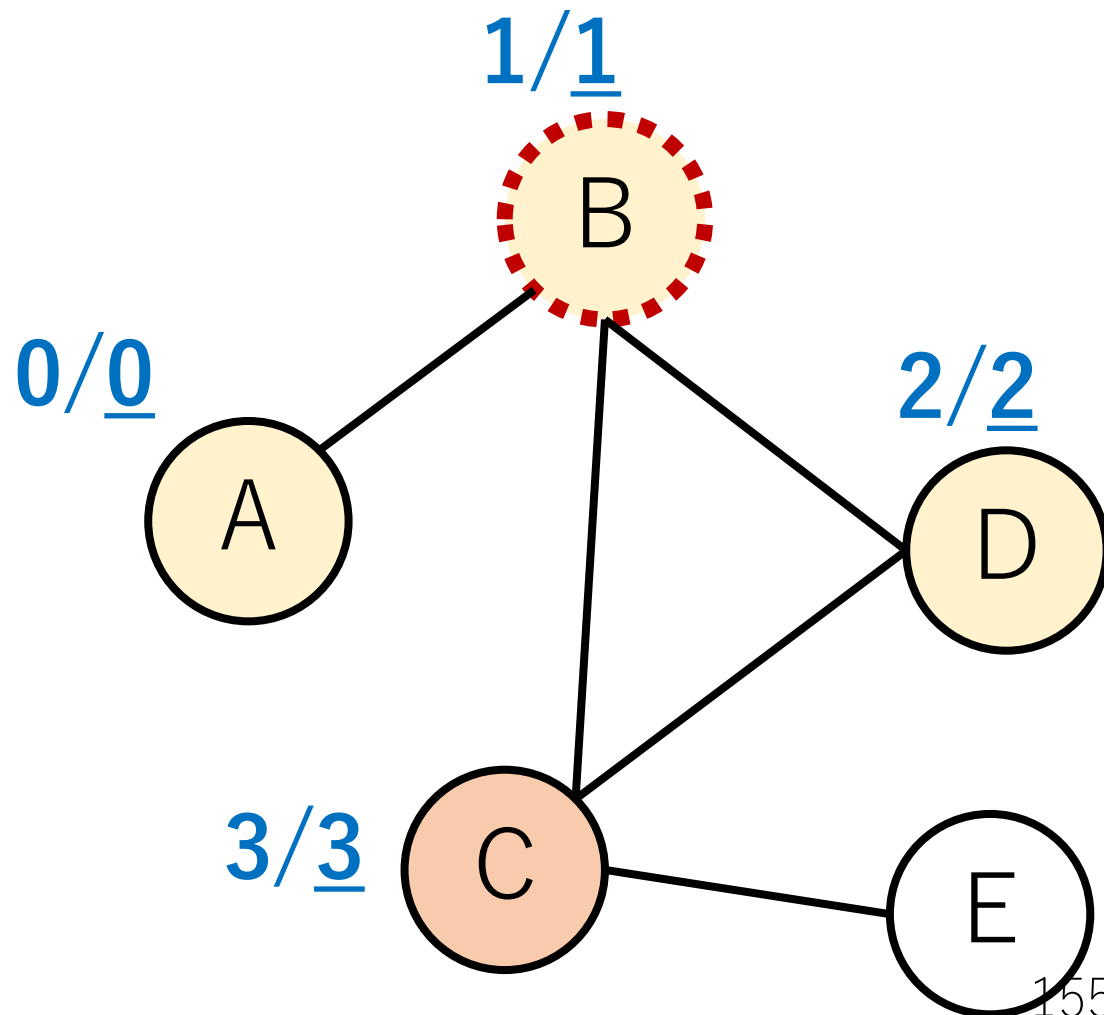


橋の検出のアルゴリズム実行例

CからはBとEが次の候補となる。

Bは訪問済だが、直前のノードではない。

ただし、Bは訪問済なので、Bより先に進むことはない。

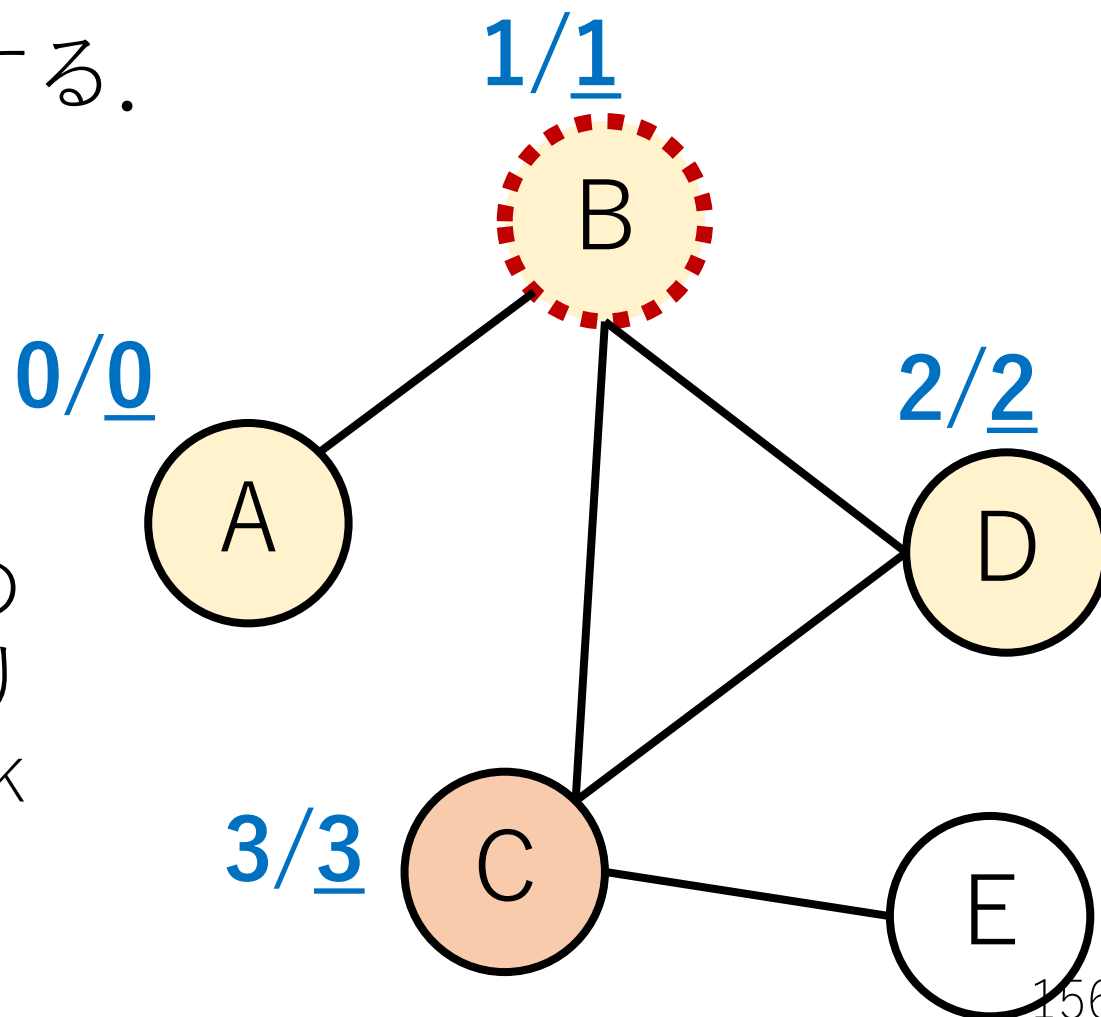


橋の検出のアルゴリズム実行例

このような状況は通ってきた辺が
後退辺になっていることを示唆する。

BからCに至る新たなルート
が見つかったことになる。

このような場合、今いるノード
の現在のlowlinkの値と、到達先の
ノードの訪問順序を比較し、より
小さい方を今いるノードのlowlink
にする。



橋の検出のアルゴリズム実行例

この例では、

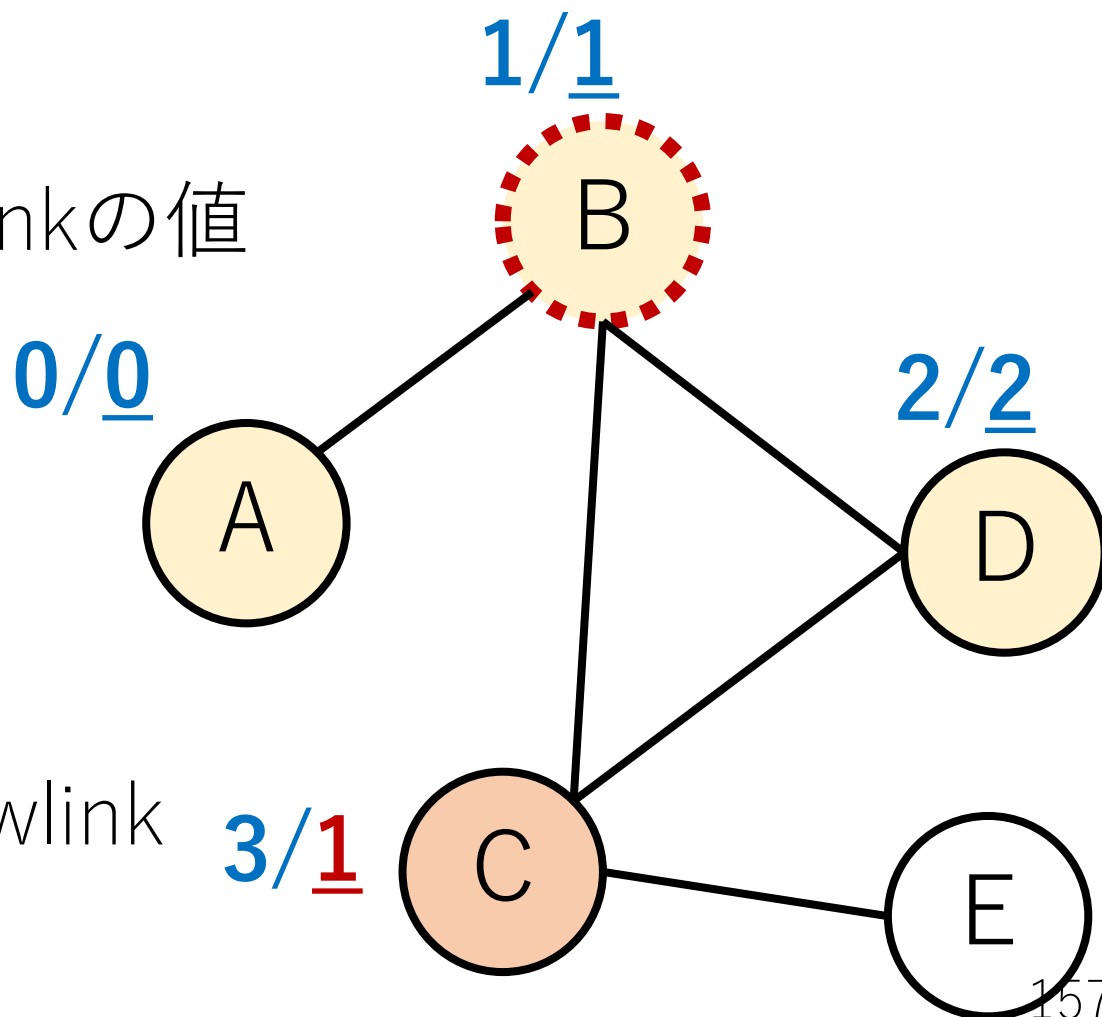
今いるノード (C) の現在のlowlinkの値

= 3

到達先のノード (B) の訪問順序

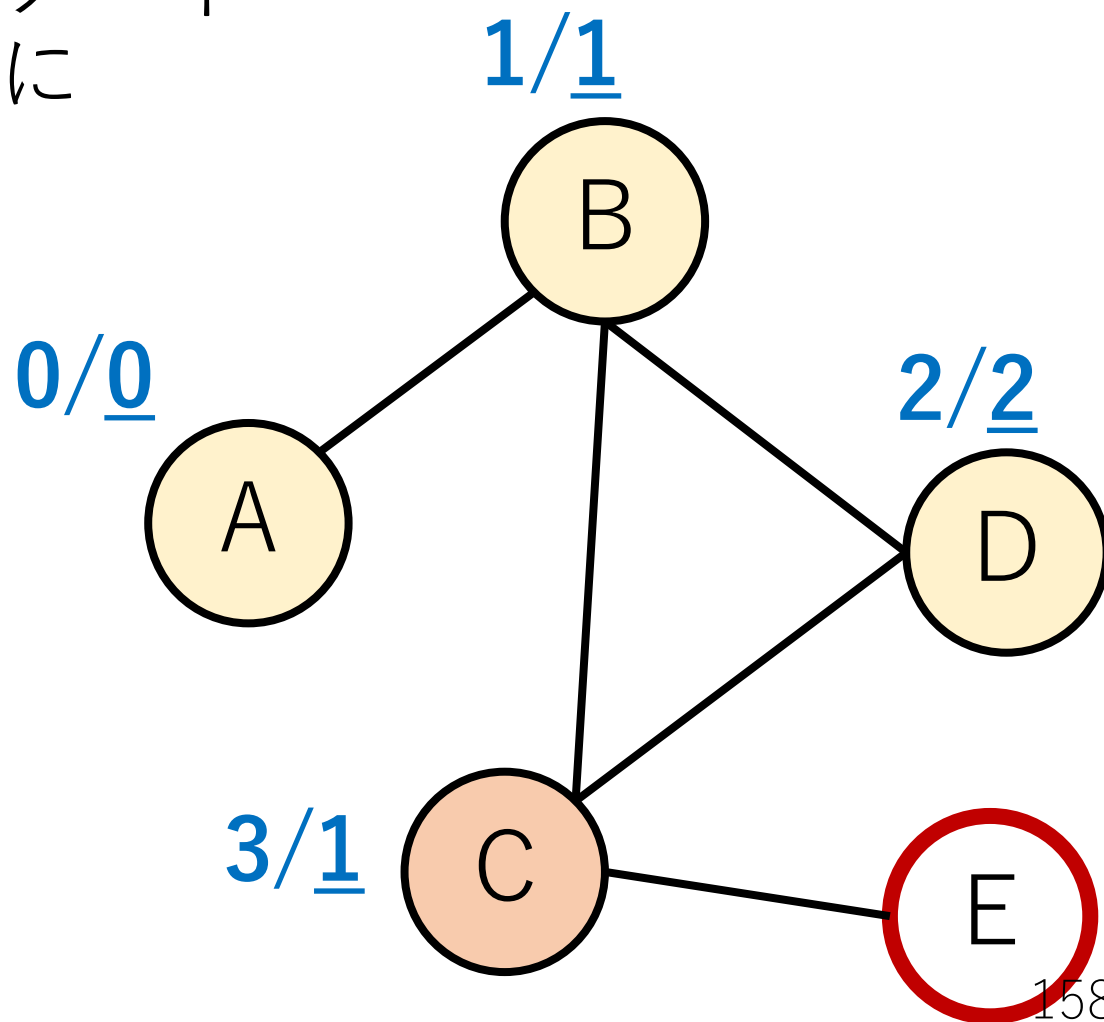
= 1

なので、今いるノード (C) のlowlink
の値を1に更新する。



橋の検出のアルゴリズム実行例

続いて、Cから辿ることのできるノードはE。これは未訪問なので、ここに移動する。

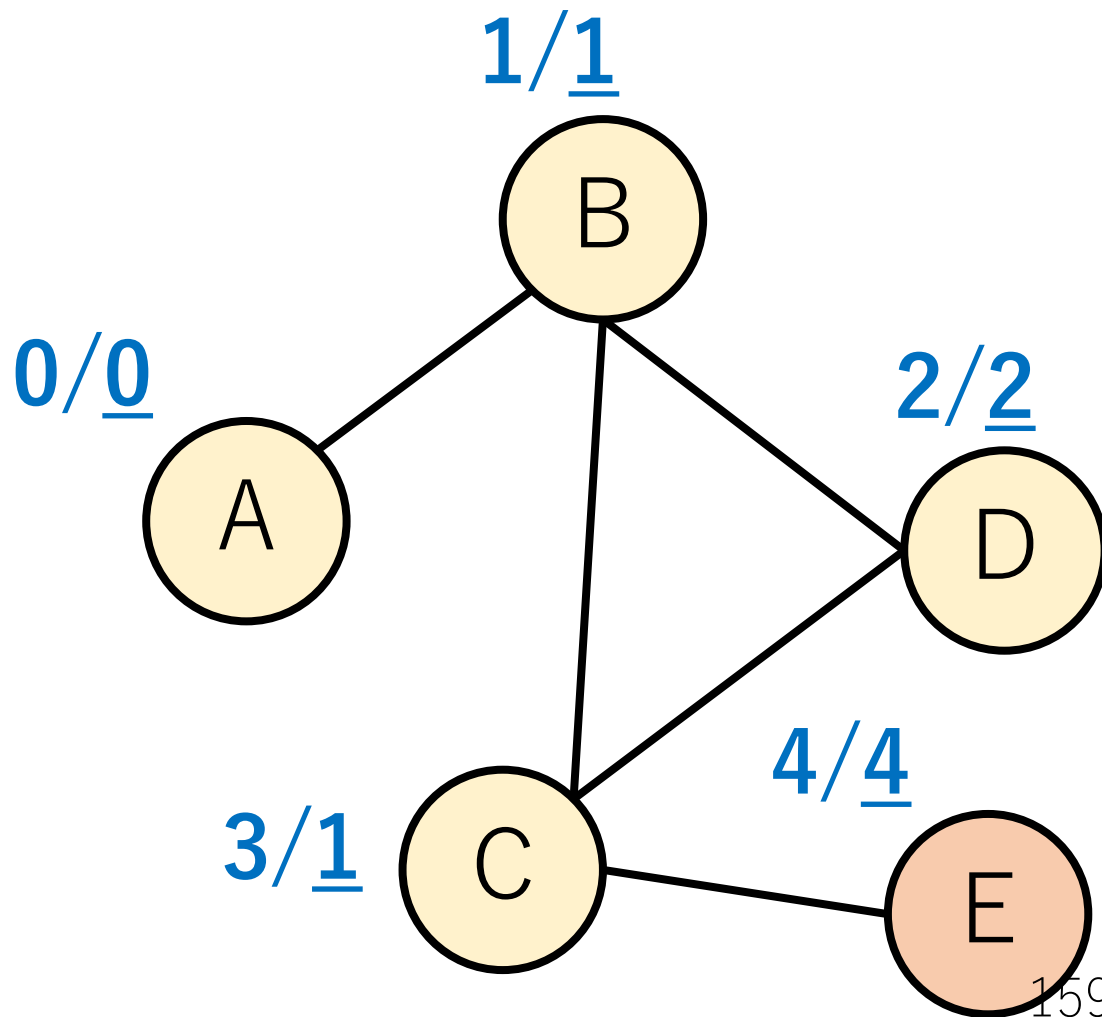


橋の検出のアルゴリズム実行例

Eからは新たに辿ることのできる
ノードはない。

Cは直前のノード。

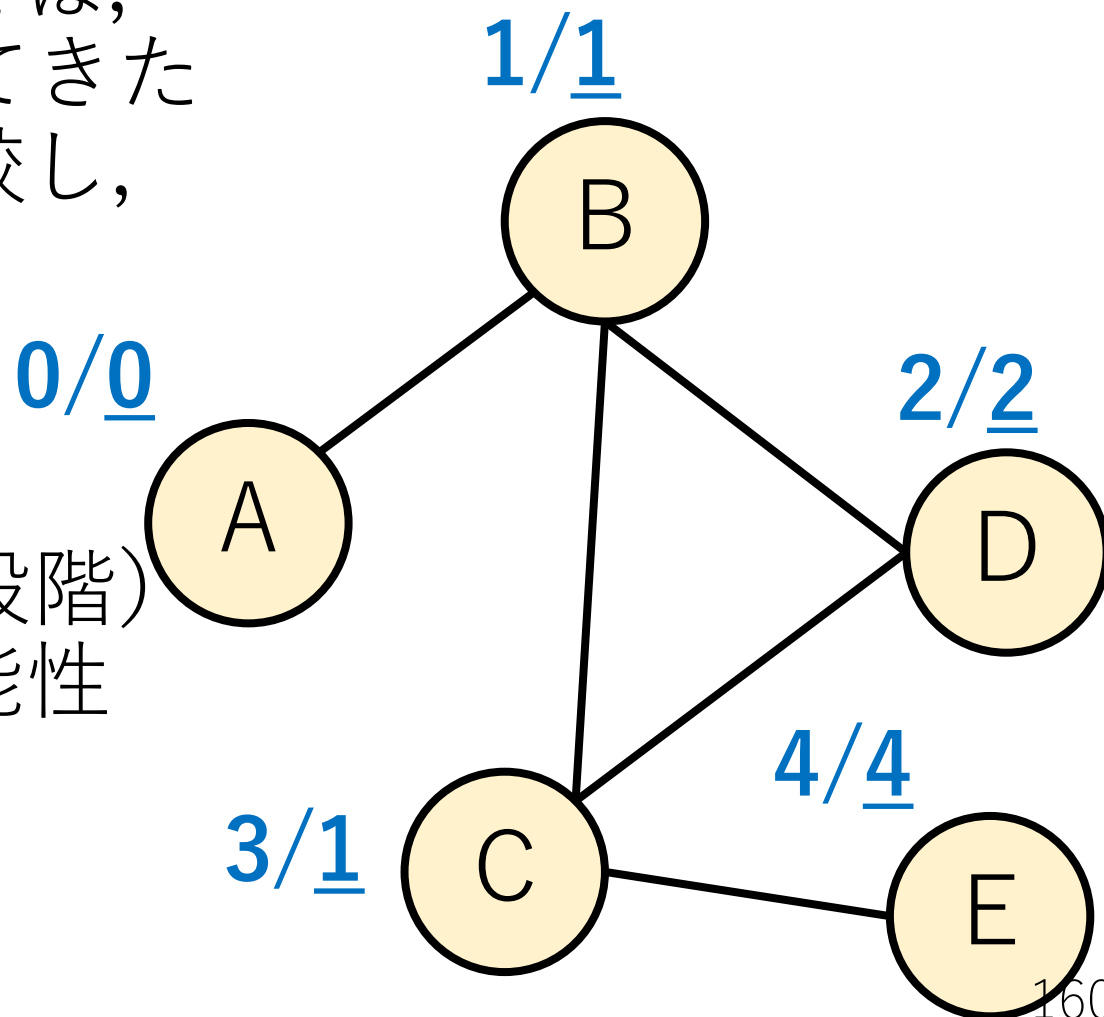
DFSの再帰はここで終了し、
順次戻っていくことになる。



橋の検出のアルゴリズム実行例

DFSの再帰から戻ってきた直後では、戻って来る前のノードと、戻ってきたところのノードとのlowlinkを比較し、より小さい方を新たなlowlinkの値とする。

この段階（再帰から戻ってくる段階）でもlowlinkの更新が行われる可能性がある。



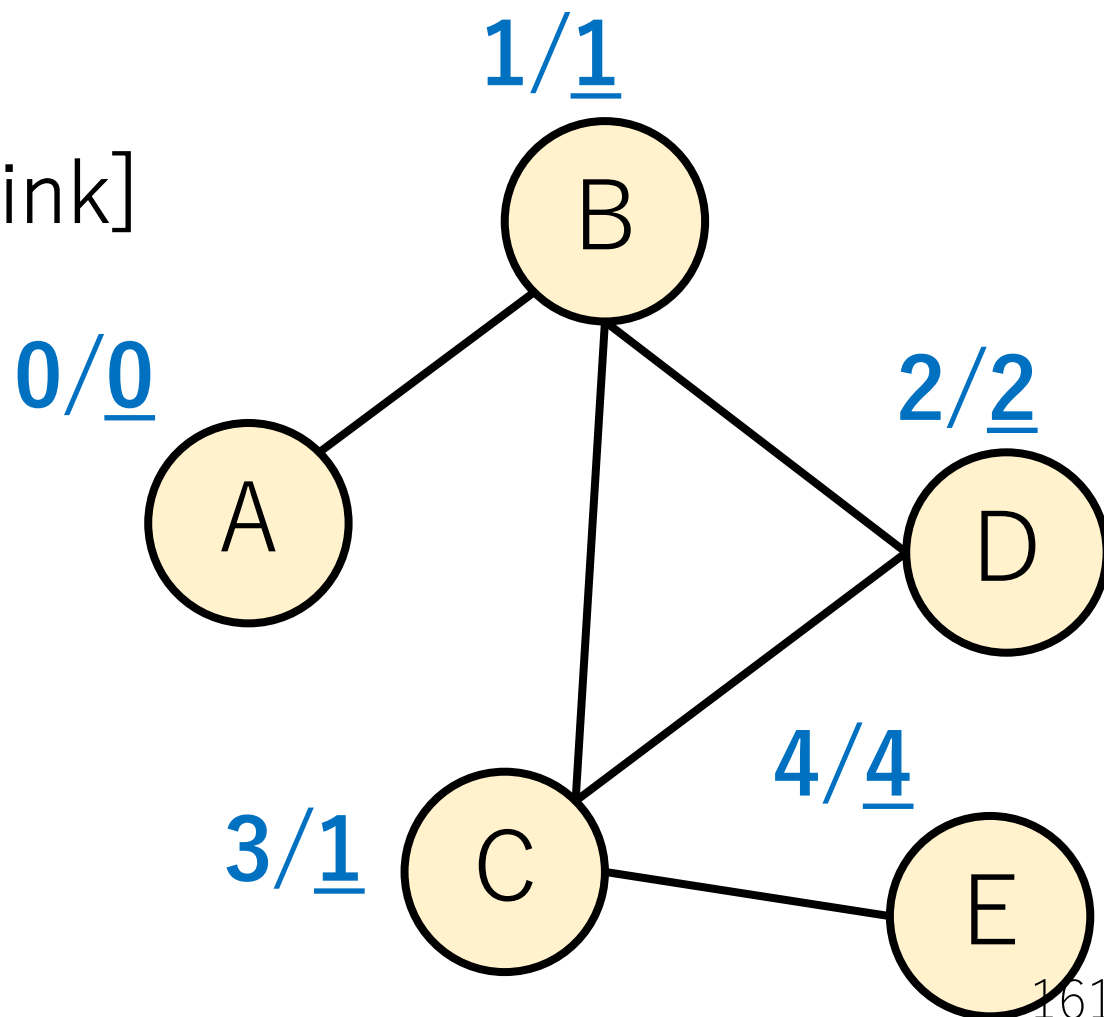
橋の検出のアルゴリズム実行例

さらに,

[あるノードの訪問順序]

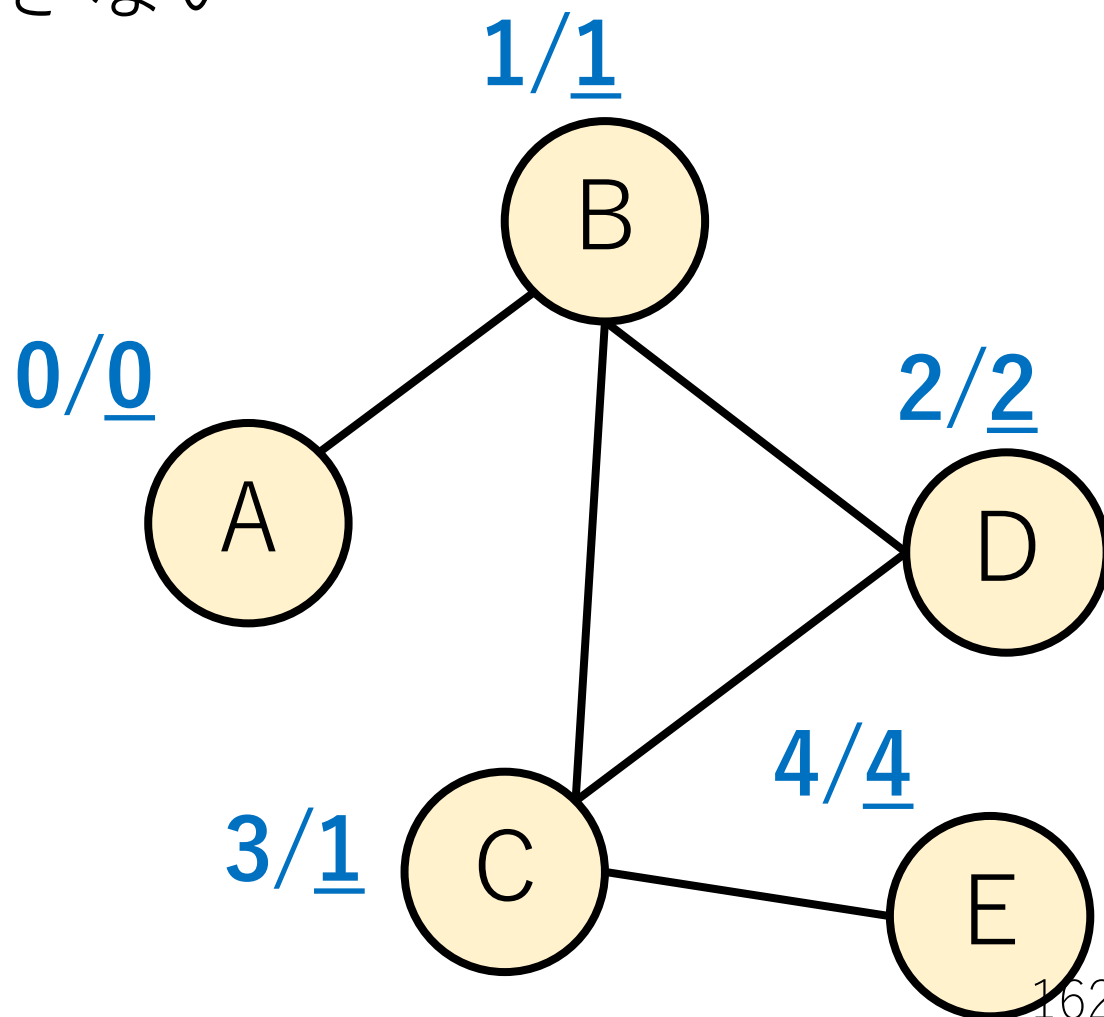
< [その子ノードのlowlink]

であるかをチェックし, 橋の検出を行う.



橋の検出のアルゴリズム実行例

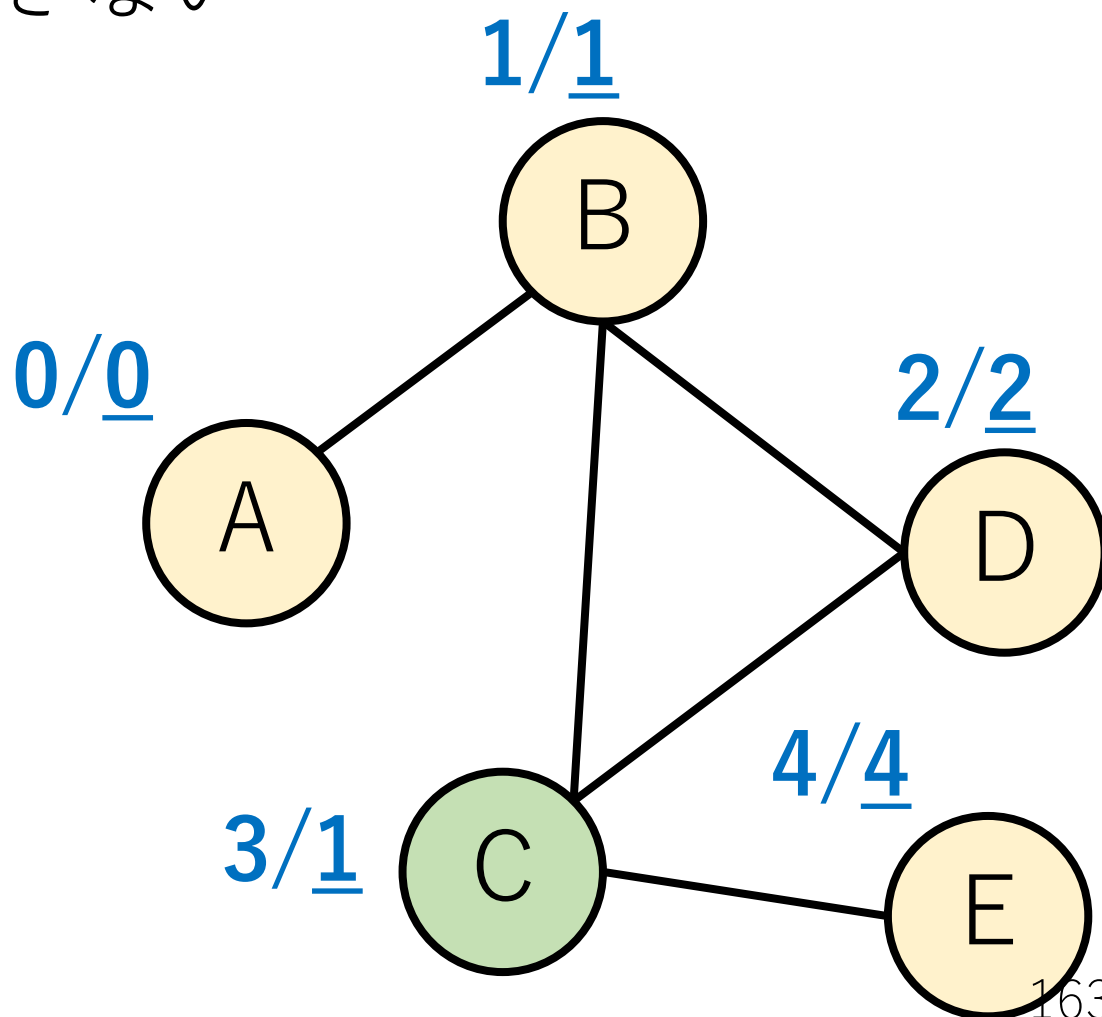
Eは終端なので、これ以上何もできないため、lowlinkの値はそのまま。



橋の検出のアルゴリズム実行例

Eは終端なので、これ以上何もできないため、lowlinkの値はそのまま。

よって、1つ前のノードであるCに立ち戻る。



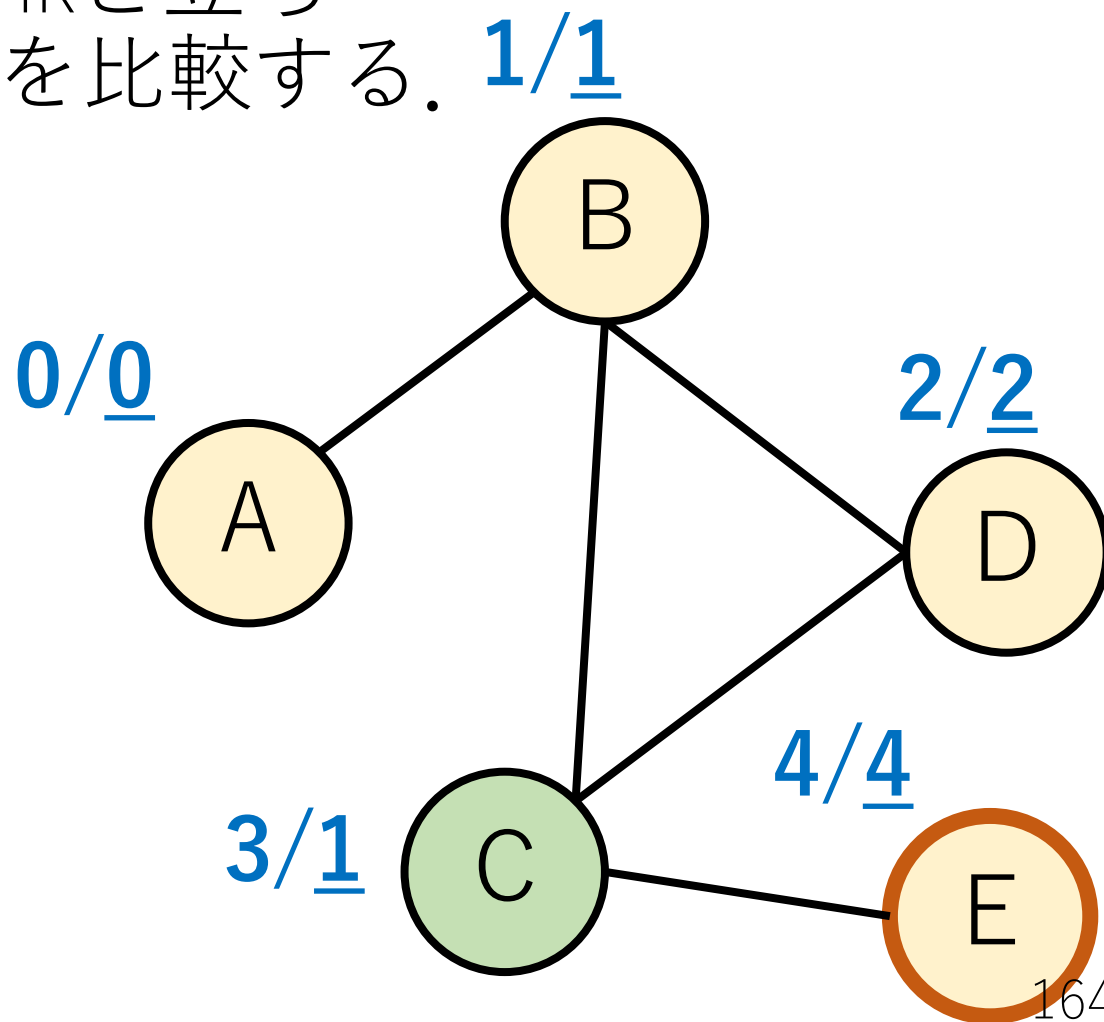
橋の検出のアルゴリズム実行例

立ち戻る前のノード (E) のlowlinkと立ち
戻った後のノード (C) のlowlinkを比較する. $1/\underline{1}$

Eのlowlink = 4

Cのlowlink = 1

よって, Cのlowlinkは1のまま.



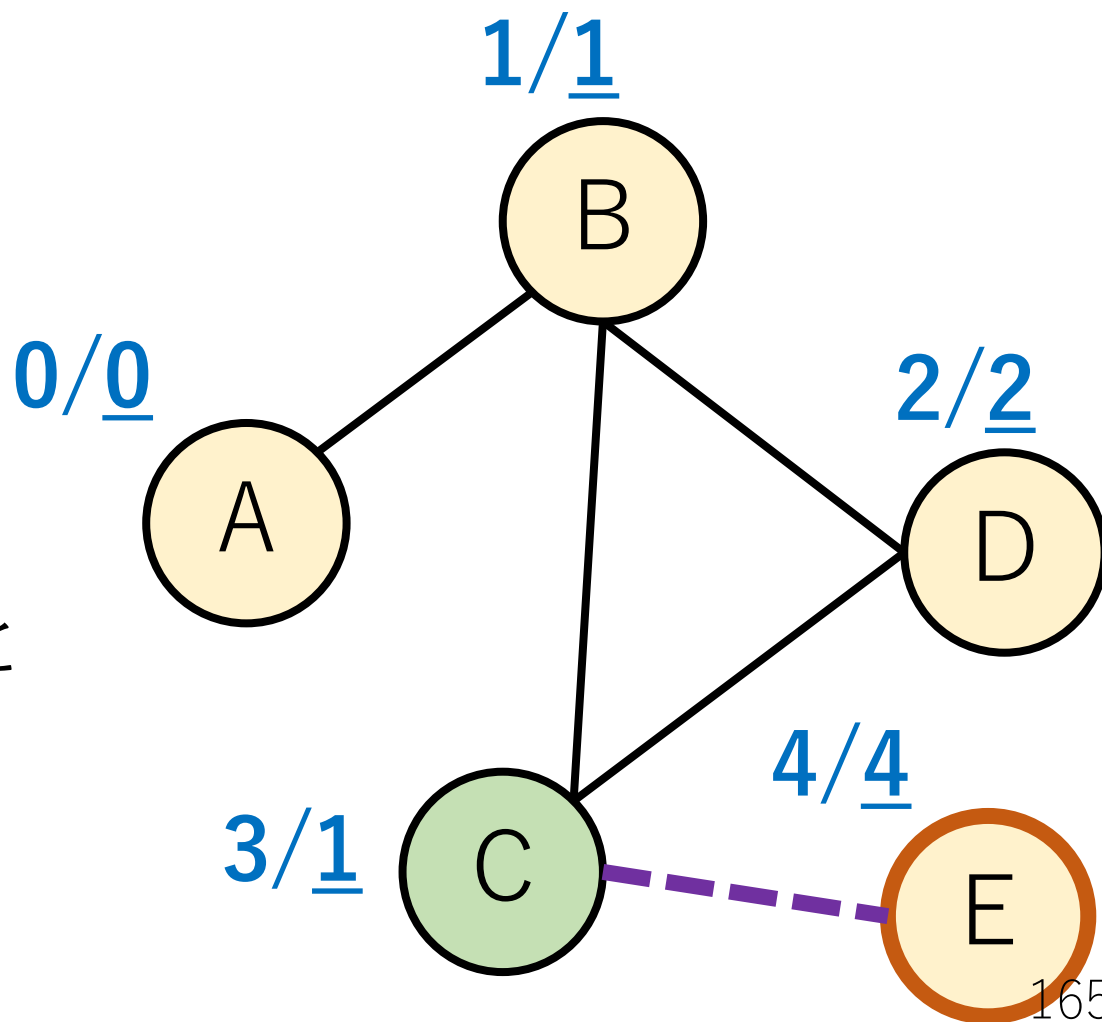
橋の検出のアルゴリズム実行例

さらに,

ノードCの訪問順序 = 3

ノードEのlowlink = 4

であるから, “[あるノードの訪問順序] < [その子ノードのlowlink]” が成立し, この辺が橋であることがわかる.

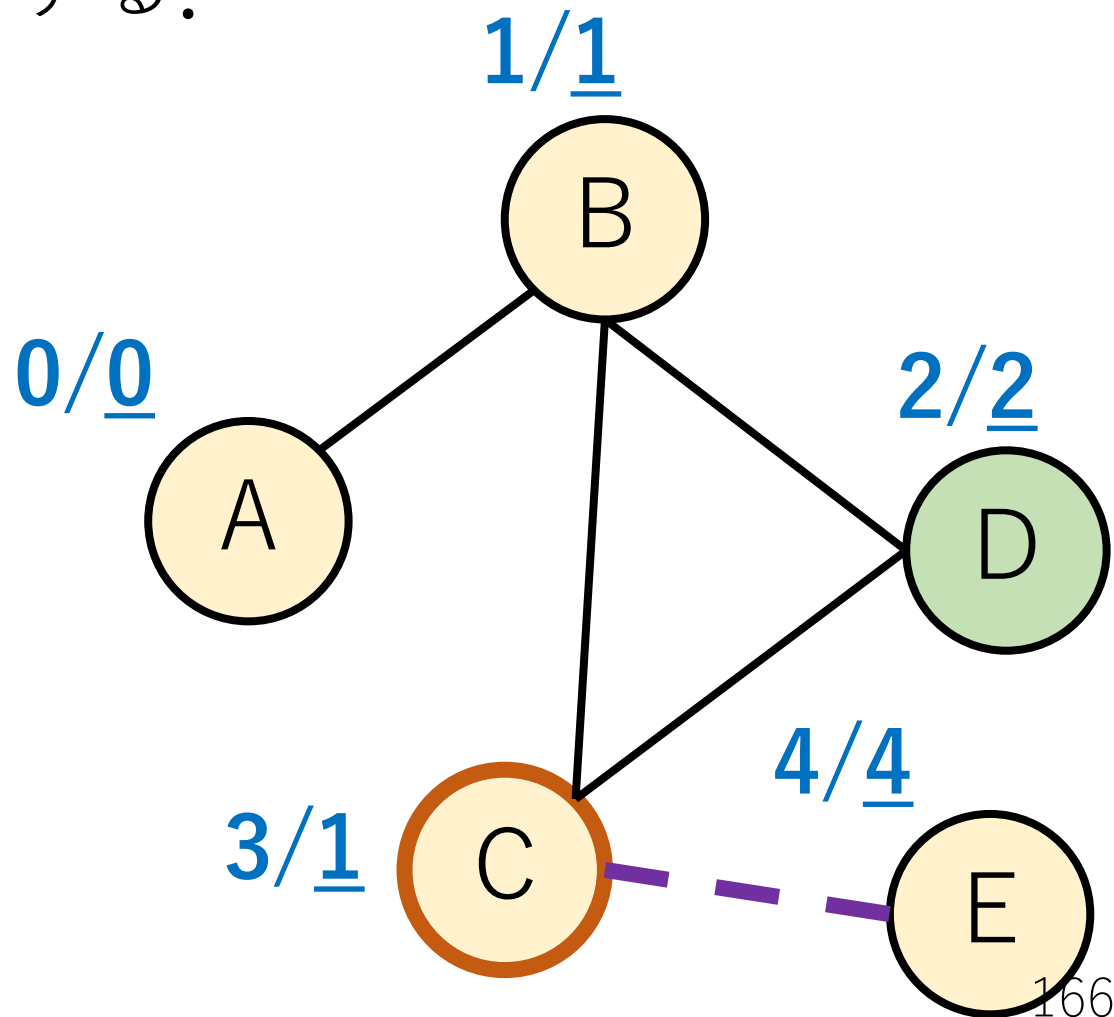


橋の検出のアルゴリズム実行例

次に，Dに立ち戻り，同じ処理をする．

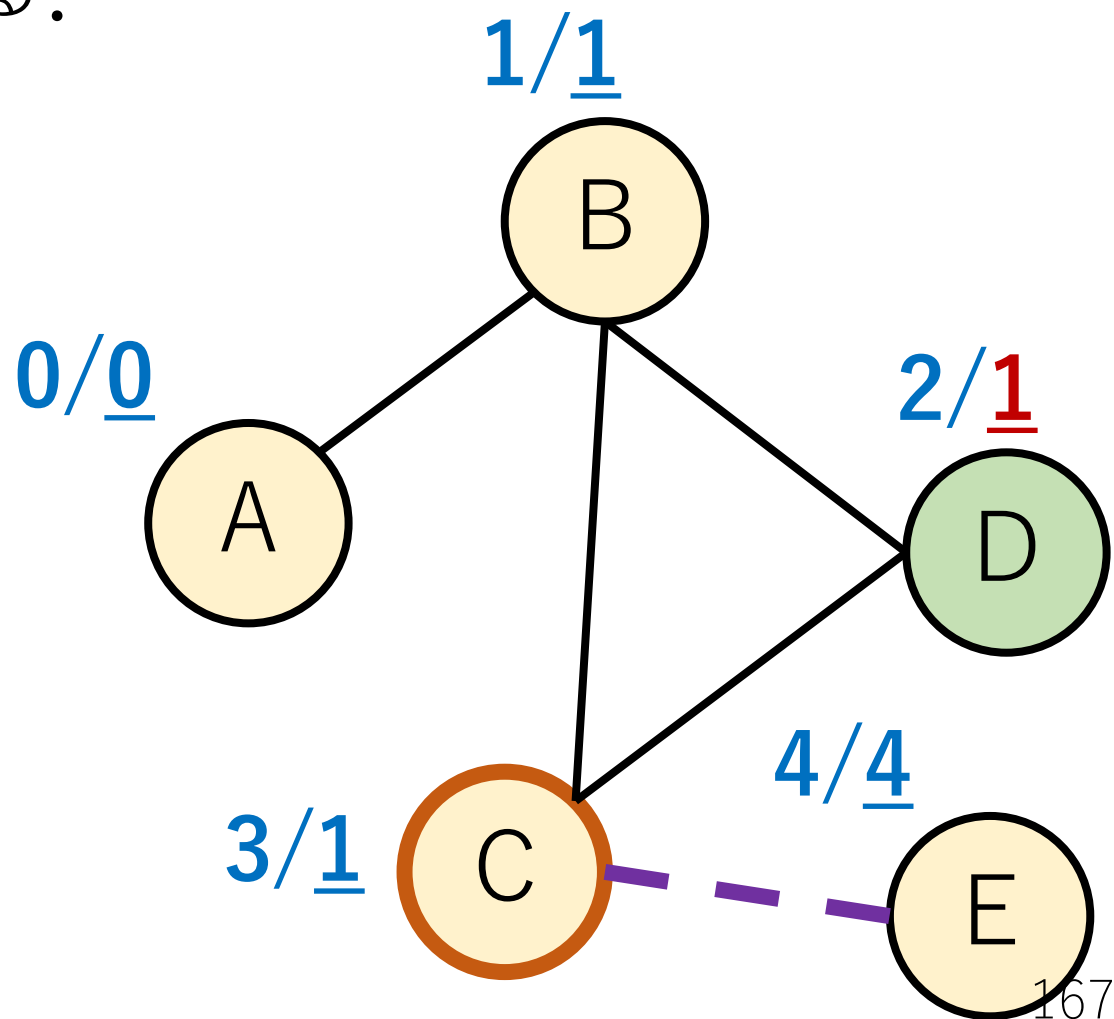
Cのlowlink = 1

Dのlowlink = 2



橋の検出のアルゴリズム実行例

よって、Dのlowlinkを1に更新する。



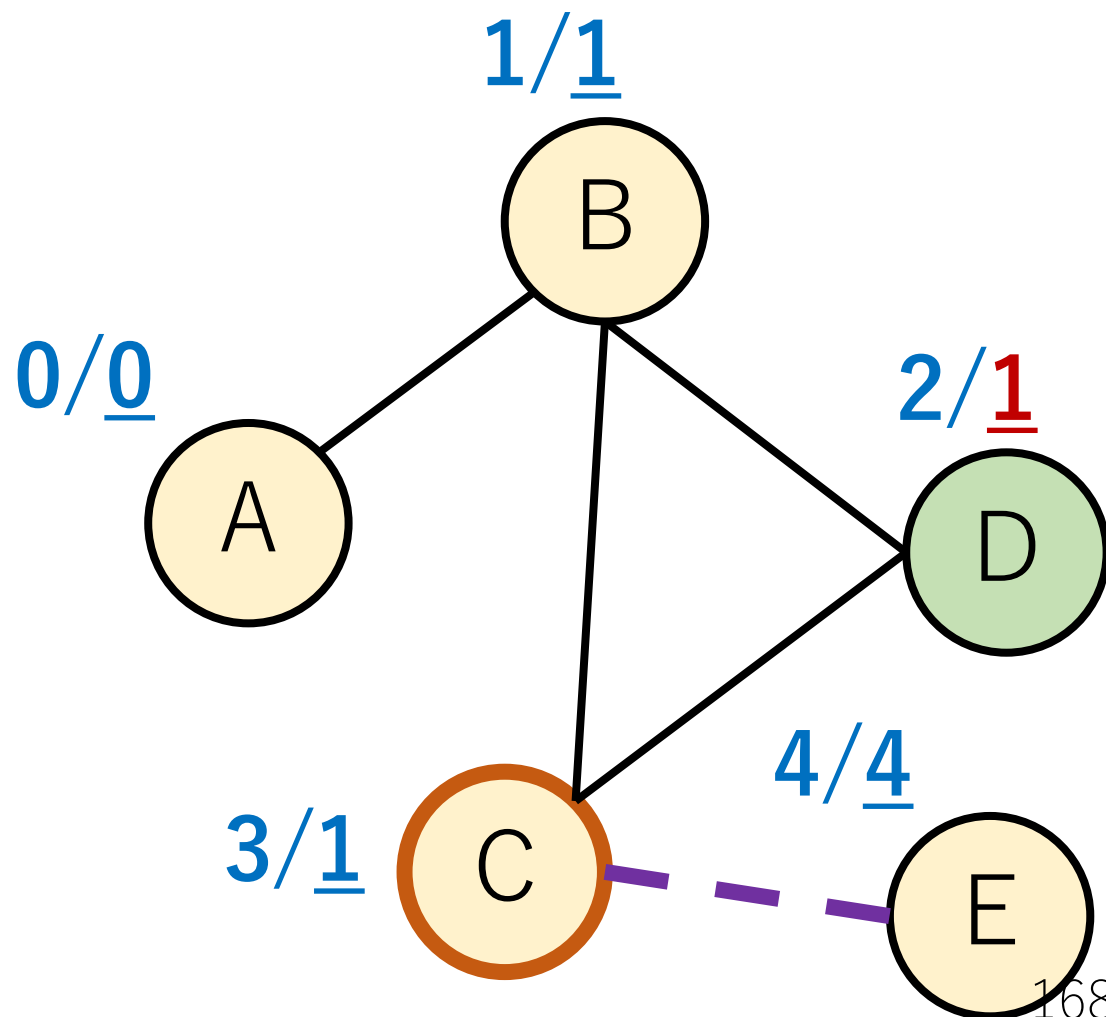
橋の検出のアルゴリズム実行例

さらに,

ノードBの訪問順序 = 2

ノードCのlowlink = 1

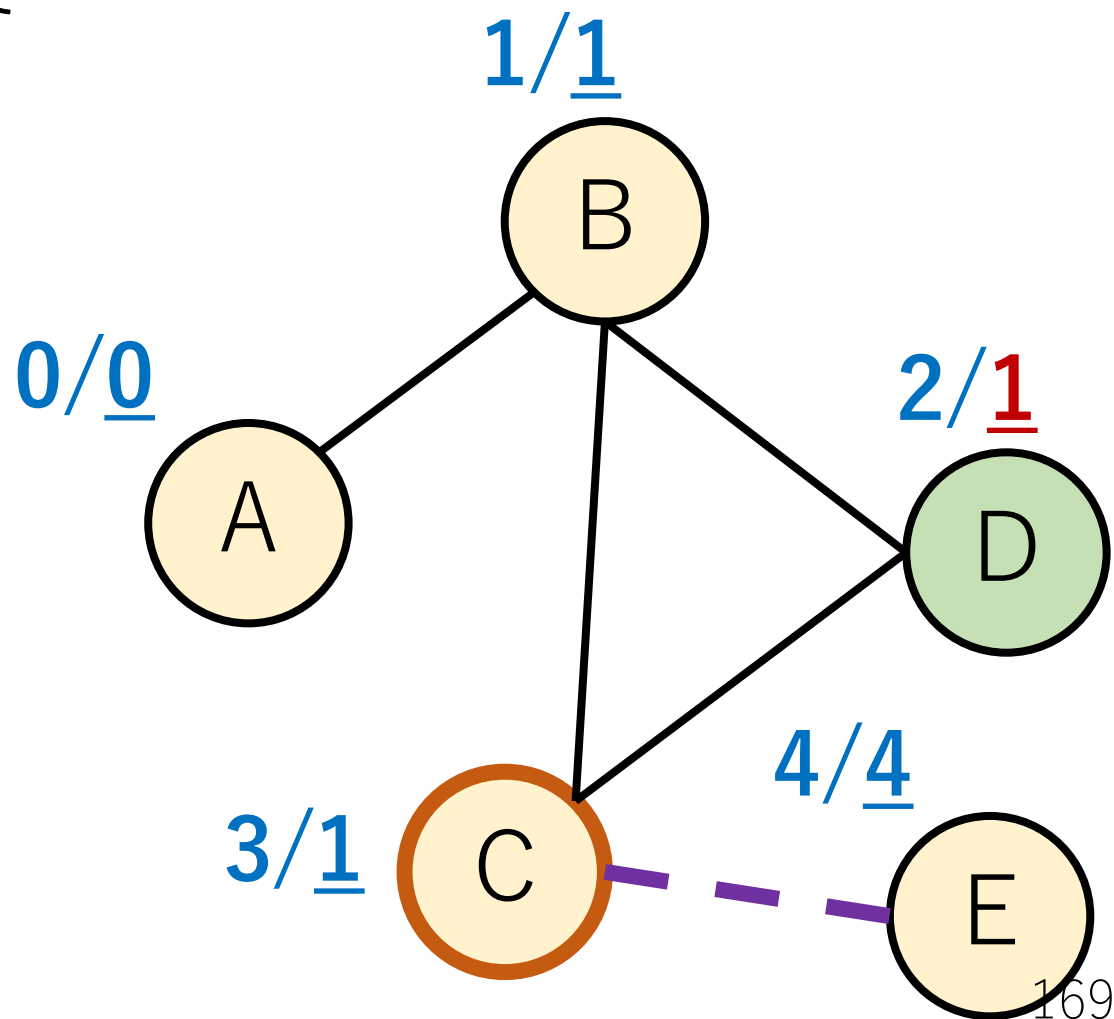
であるから, “[あるノードの訪問順序] < [その子ノードのlowlink]” が成立せず, この辺は橋でないことがわかる.



橋の検出のアルゴリズム実行例

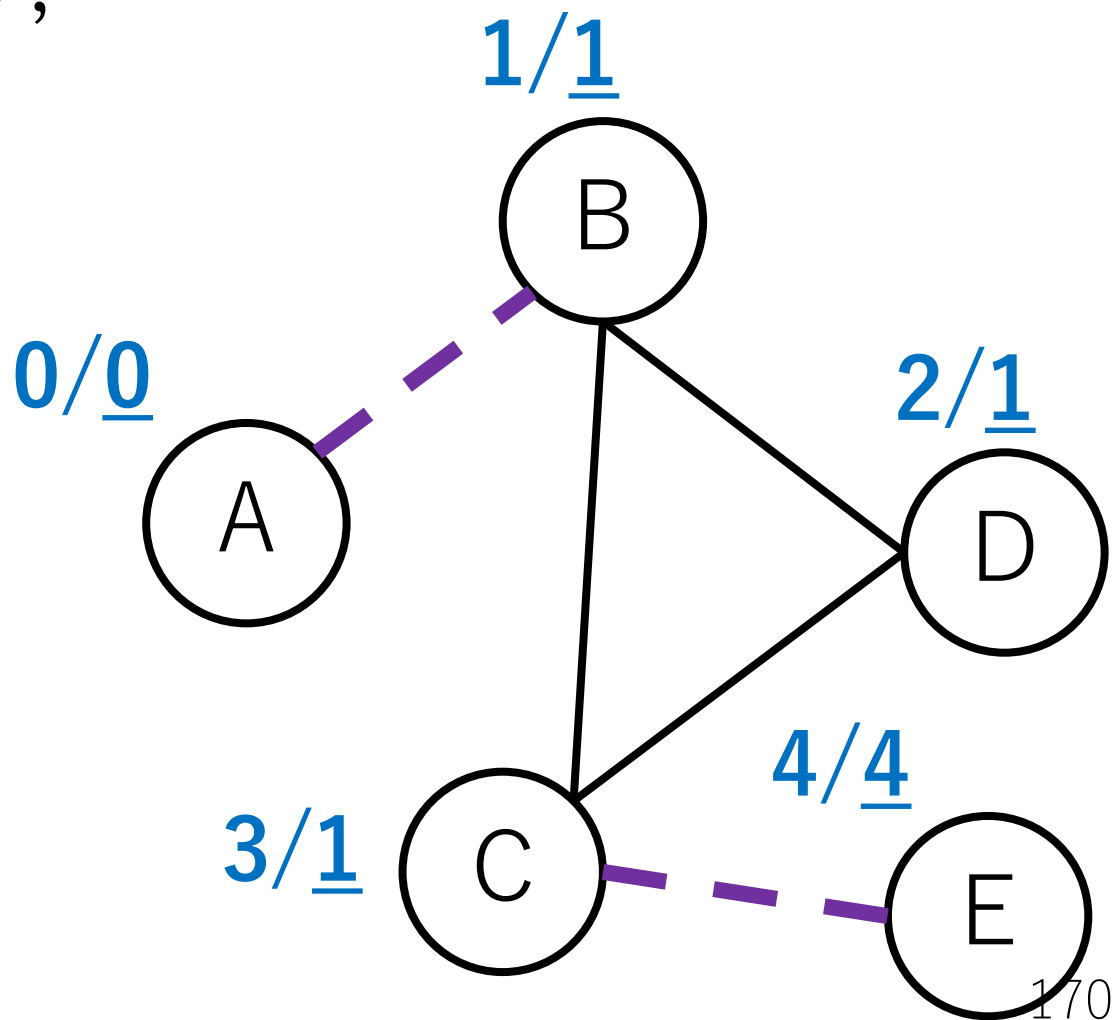
以降，スタート地点のノードまで
これを繰り返す。

この例ではもう lowlink の
更新はない。



橋の検出のアルゴリズム実行例

最終的には，このような形になり，
A-B, C-Eの辺が橋であることが
求められる。



実装例

```
import sys
sys.setrecursionlimit(1000000) # 再帰の上限回数を増やす

# Vはノードの数
edge = [[] for i in range(V)] # 隣接リスト
visited = [0]*V # 訪問済かどうかを保持
lowlink = [10**6]*V # lowlinkを保持
order = [0]*V # 訪問順序を保持
```

実装例

```
# 再帰で実装するバージョン
# 現時点のノード, 1つ前のノード, 訪問順序のカウント
def dfs_bridge(edges, start, prev, count):

    visited[start] = 1
    order[start] = lowlink[start] = count
    count += 1
```

実装例

```
def dfs_bridge(edges, start, prev, count):  
    ...  
    # 再帰でDFS. 戻ってきたときにlowlinkを更新.  
    for n in edges[start]:  
        if not visited[n]:  
            visited[n] = 1  
            count = dfs_bridge(n, start, count)  
            lowlink[start] = min(lowlink[n], lowlink[start])
```

実装例

```
def dfs_bridge(edges, start, prev, count):  
    ...  
    for n in edges[start]:  
        ...  
        # 後退辺がある場合はそれをチェック。  
        elif prev != n:  
            lowlink[start] = min(lowlink[start], order[n])
```

実装例

```
def dfs_bridge(edges, start, prev, count):  
    ...  
    # 橋が見つかったら表示.  
    if order[start] < lowlink[n]:  
        print('{}-{}'.format(start, n))  
  
    return count
```

実行例

```
edge = [  
[1],          #ノードA  
[0, 2, 3],   #ノードB  
[1, 3, 4],   #ノードC  
[1, 2],      #ノードD  
[2]]         #ノードE
```

===実行結果===

2-4

0-1

似たような話として

関節点の検出

そのノード，及び出ている全ての辺を削除すると，
グラフ全体が非連結になるようなノードを見つける。

二重辺連結成分分解

どの辺を1つ取り除いても連結であることが保たれる
部分グラフを取り出してくる。

→橋を全部取り除けば良い。

と、一応紹介しましたが、

橋の検出の話は、今の時点ですぐに理解できなくても大丈夫です。😊

ただしlowlinkのように、ノードや辺に対してそれらの状態を表す変数を定義し、それらの関係性を見ることで、グラフの性質を見極めることは、これからも出てきます。

追加で定義される変数がどういう意味を持っているか、少しずつ理解しながら、グラフのアルゴリズムを理解してもらえればと思います。

まとめ

グラフの表現

隣接リストと隣接行列

BFSとDFS

データ構造の違いが探索方針の違いになることを理解する

BFSとDFSの応用

最短経路, オイラーツアー, LCA, 橋の検出

コードチャレンジ：基本問題#9-a [1点]

スライドで説明したdfsを，再帰を使う方法で再実装してください。

(pythonには再帰で呼び出せる回数の上限がありますので，一般的なグラフではこの方法が使えらるとは限りませんが，この課題では問題ないです。)

コードチャレンジ：基本問題#9-b [1点]

迷路のスタートからゴールに辿り着くための最短距離をBFSもしくはDFSを使って求めてください。

最短距離を求める，というときにはどちらのほうが一般的には有利か，をよく考えて実装をしてください。

DFSを用いる場合，再帰は使用しないでください。

dequeを使用しても構いません。

コードチャレンジ：Extra問題#9 [3点]

BFS, DFSを利用する問題.

2回目の授業アンケート実施中です！

ご協力のほど，なにとぞよろしくお願いいたします。

フォームのURLは#generalに流してあります。