Algorithms (2021 Summer)
#9：幅優先探索，深さ優先探索

矢谷 浩司

前回出た質問

「メモ化再帰のnoteは半分しか埋まっていないので計算量は漸化式方式の半分になっているのでしょうか」

→テーブルの値を埋める場所はメモ化再帰のほうが少なくはなりますが，計算量のオーダーは変わりません．また，メモ化再帰では再帰のオーバヘッドがあるので，計算時間は必ずしも短くなるわけではありません．

前回出た質問

「どうして1マス下に行くのが削除になるのかがよくわからないです。」

→TAの役山さんのコメント.

dp[i][j] = dp[i][j-1] + 1 という遷移式を考えてもらうとよいと思います. 左辺の dp[i][j] は一つ目の文字列のi文字目までと二つ目の文字列のj文字目までの距離を表しますが , 右辺はそれを二つ目の文字列のj文字目を削除した dp[i][j-1]+1 として考えています.

前回出た質問

「行と列の文字が一致しているかしていないかで何が変わるのかよく分からないです。」

→TAの役山さんのコメント.

aa という文字列と aa という文字列の距離は0です.
これを踏まえて aab という文字列と aaa という
文字列の距離を考えようと思うと，先程の二つの
文字列にそれぞれ b，a という異なる文字を付け
加えているので距離は1になります.

前回出た質問

「行と列の文字が一致しているかしていないかで何が変わるのかよく分からないです。」

→TAの役山さんのコメント.

ところが, aaa という文字列と aaa という文字列の距離は，最初の二つの文字列に同じ a を追加しているので，距離は0のままになります.

このように，左上からの遷移を考えるときに，対応させたい文字の一致・不一致によって遷移式が少し変わります.

前回出た質問

「ペア組が1対多の時、その分だけ手の本数が増えて距離が無駄に長くなったりしないのでしょうか？」

→もちろん距離の定義によって変わりますが，このような場合はDPでの遷移の中で不採用になるはずです．

前回出た質問

「距離の定義で一対多を許しているのは、波形の横軸方向長さが異なっても比較できるようにするためですか？」

→TAの役山さんのコメント.

一対多を許すことで，横軸方向のシフトや拡大・縮小についても柔軟に扱うことができます.

前回の課題の修正

Extra課題の修正があり，失礼いたしました．🤦

track上では公開したあとは変更ができませんので，slackでのアナウンスに注意を払っていただけると嬉しいです．

アンケートでのコメント

「Extra課題とレポート課題で得点に差が出るのでは？」

→基本的にはExtra課題の分布，平均点（合計が0点のものを除く）と大きな相違がないように採点を調整する予定です．ただし，2つは本質的に違う課題なので，全く同じような分布や平均点にはならないことはご承知おきください．

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．

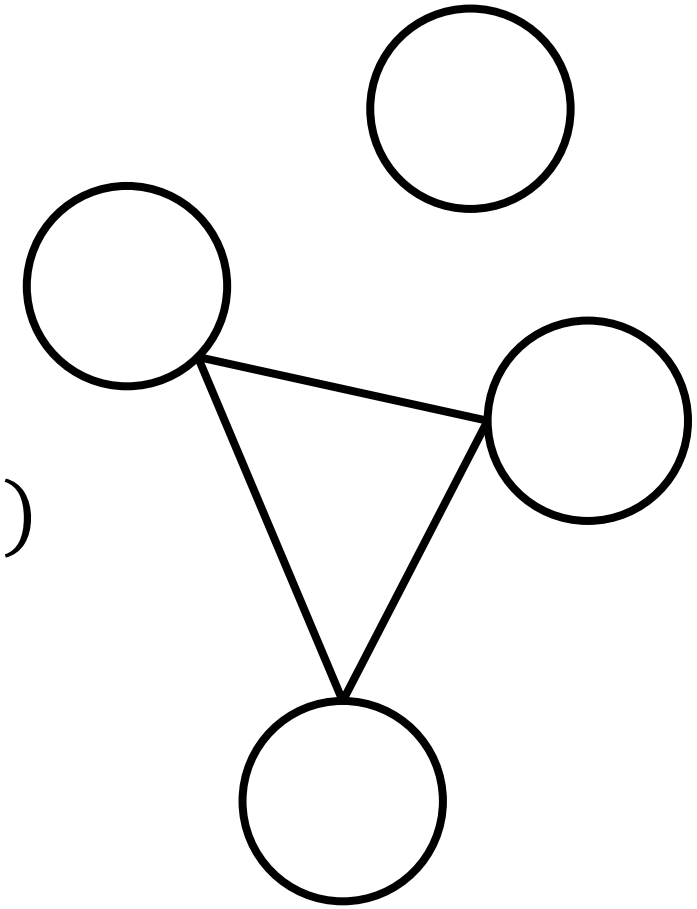
自分でコードを書いて復習するのもオススメです．

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

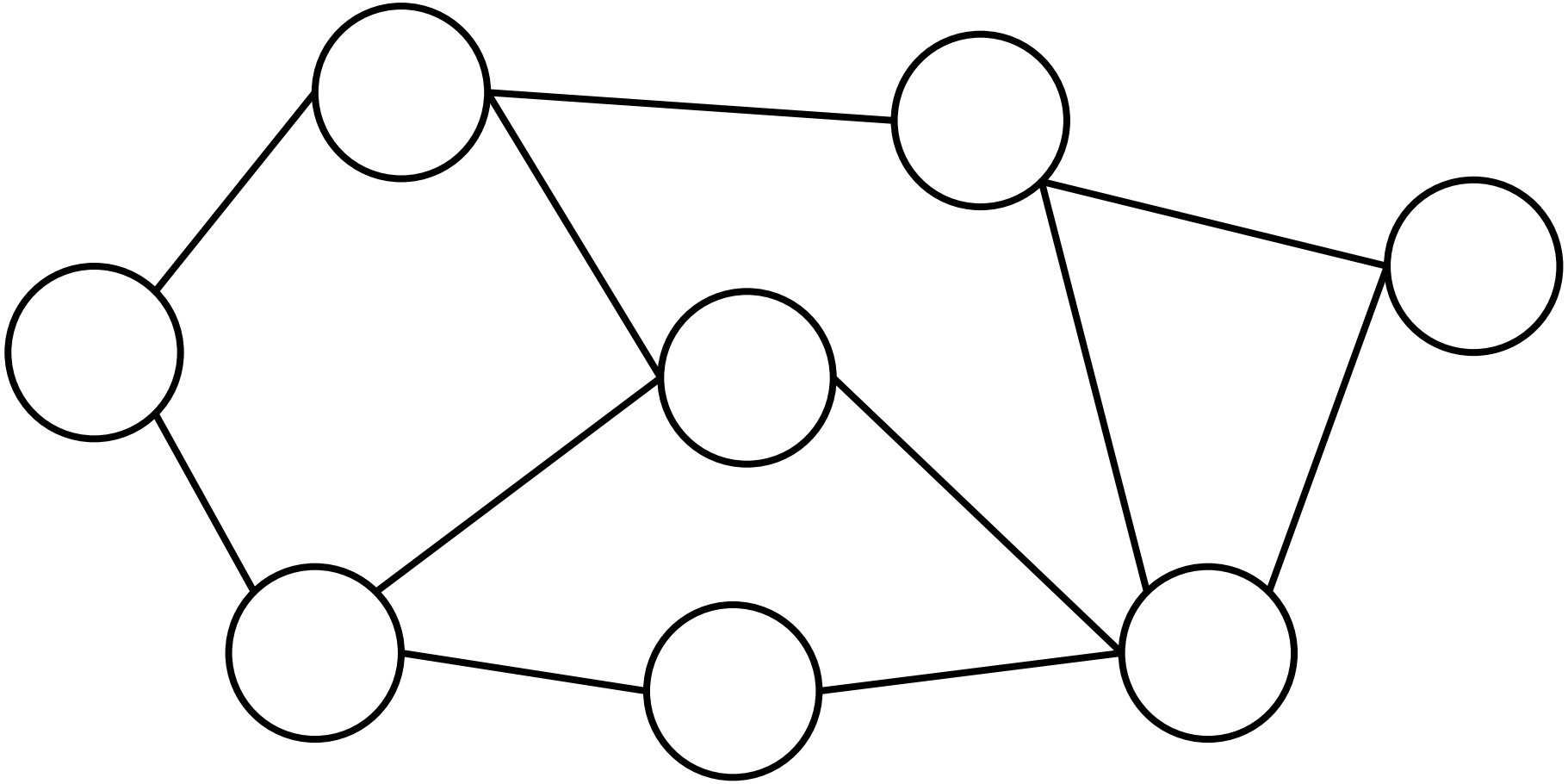
→一周して元のノードに戻ってくる経路がある場合がある

→連結でないこともある

ツリーはグラフの一種.

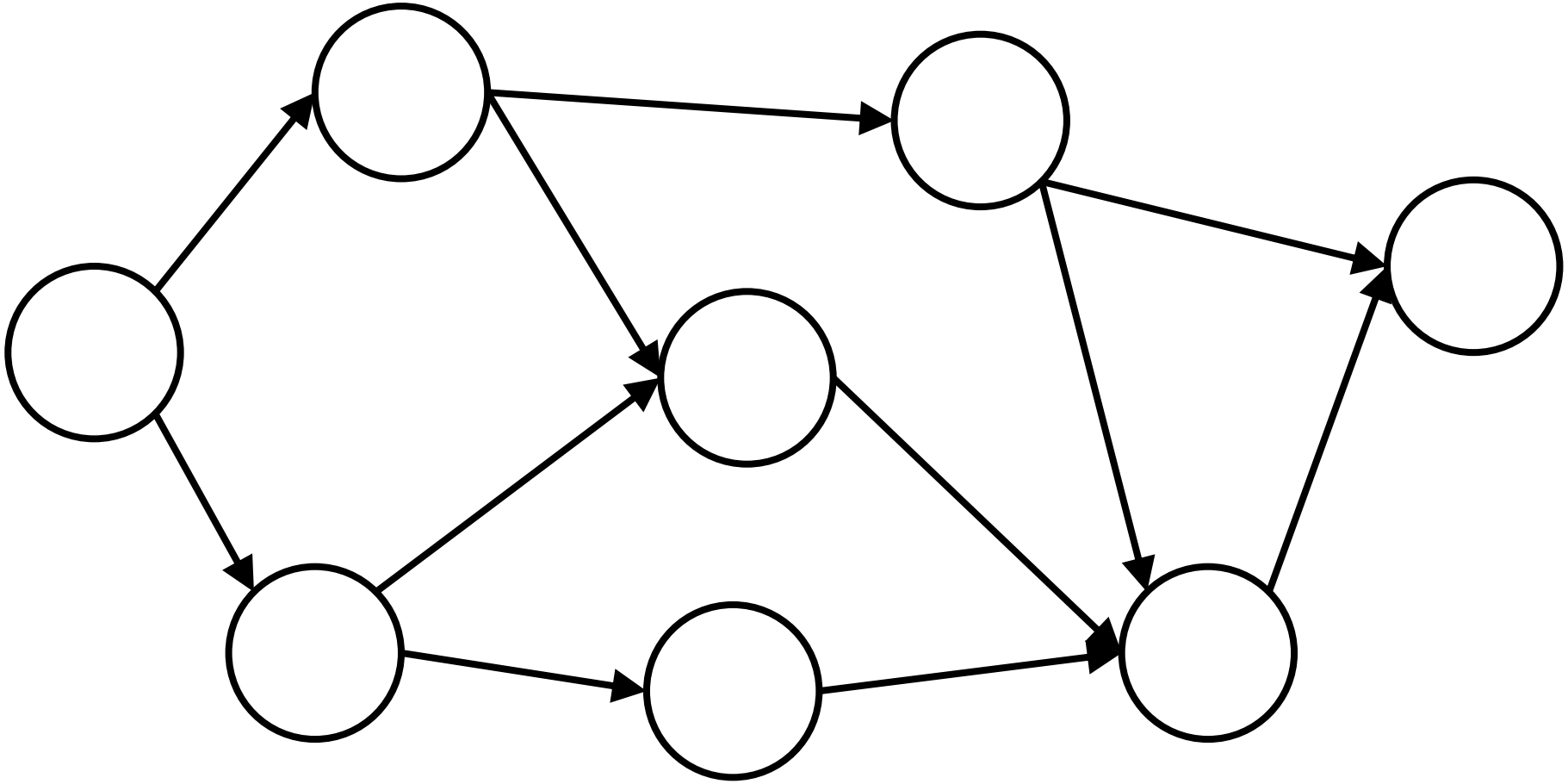
無向グラフ

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

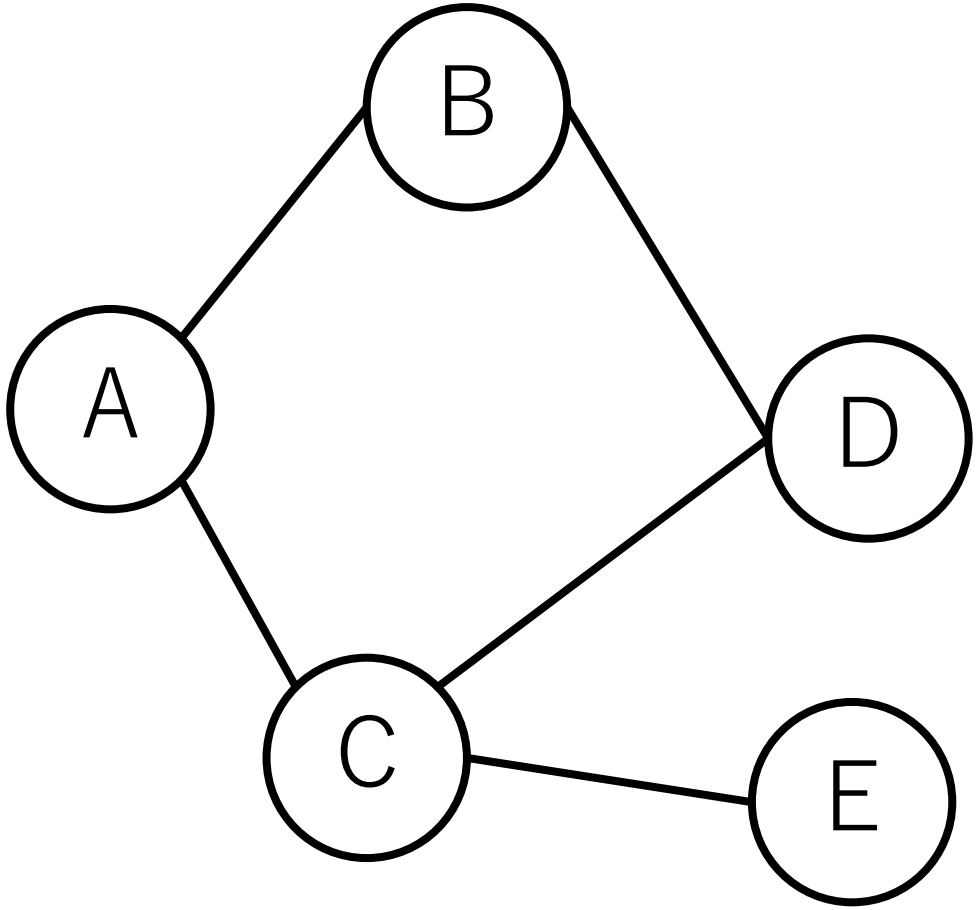
ノード間が向きのある辺（エッジ）でつながっている.



グラフを表すデータ構造

隣接リスト

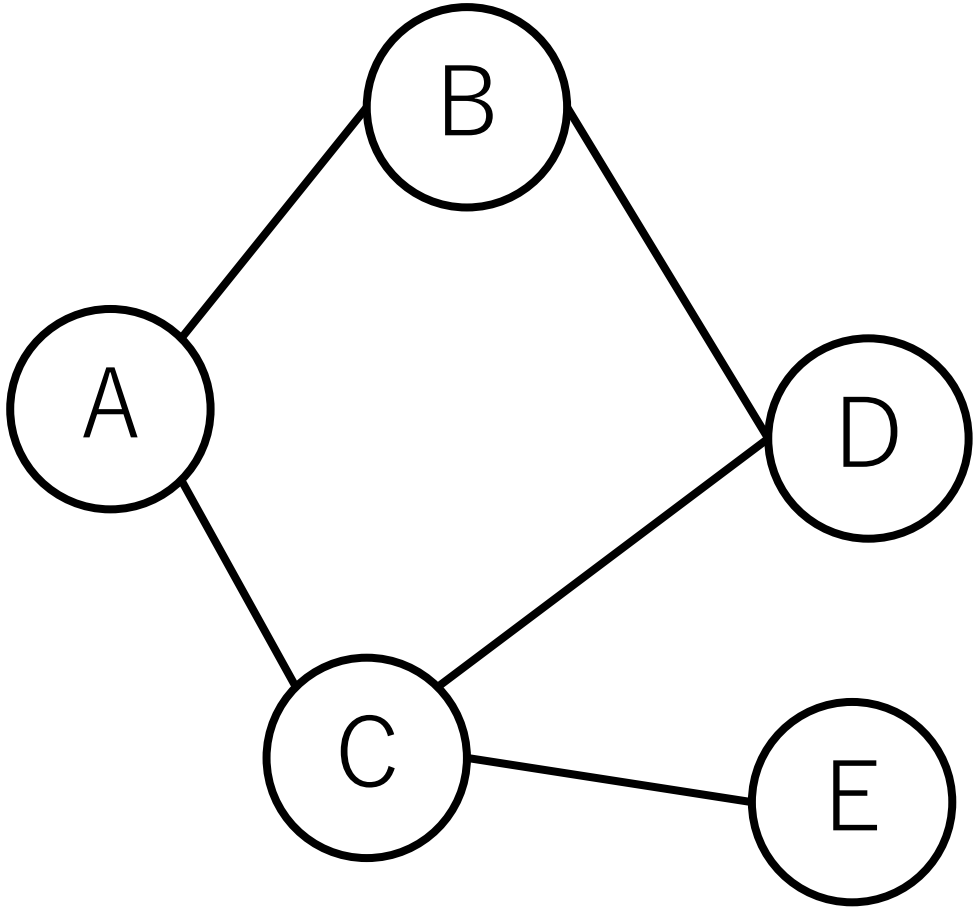
隣接行列



隣接リスト（無向グラフ）

各要素が各頂点の接続先を表す.

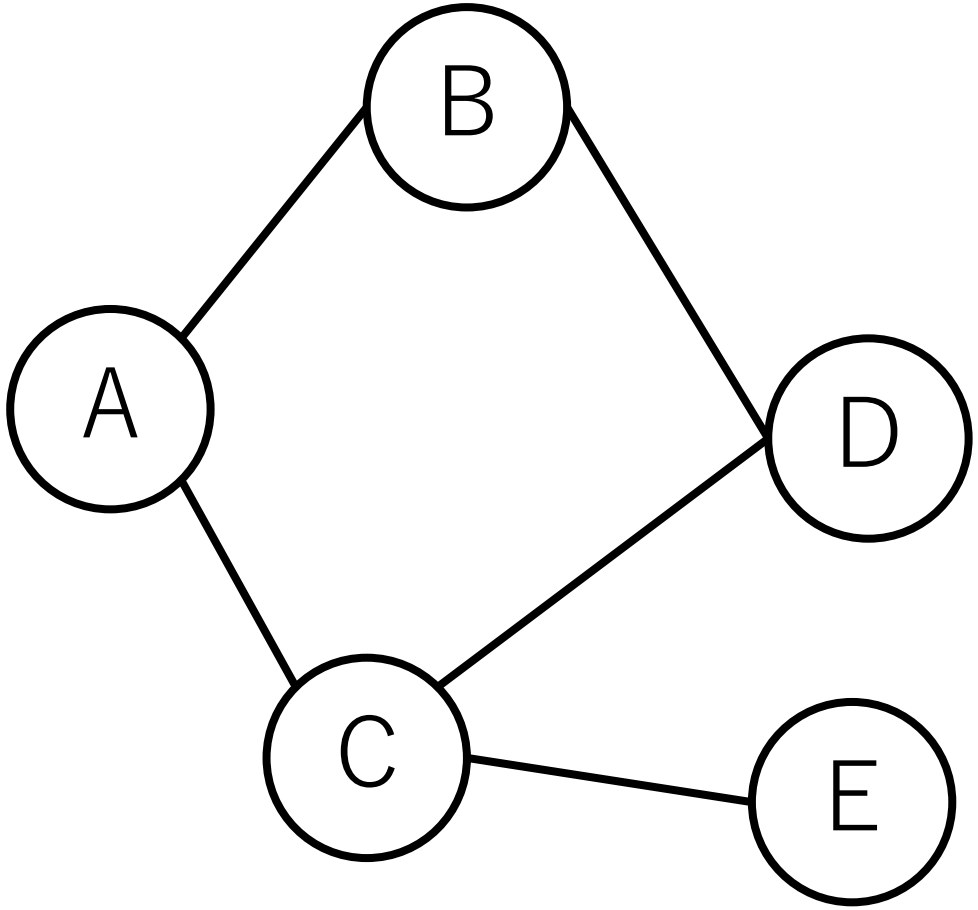
```
edge = [
[1, 2],      #ノードA node[0]
[0, 3],      #ノードB node[1]
[0, 3, 4],   #ノードC node[2]
[1, 2],      #ノードD node[3]
[2]          #ノードE node[4]
]
```



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.

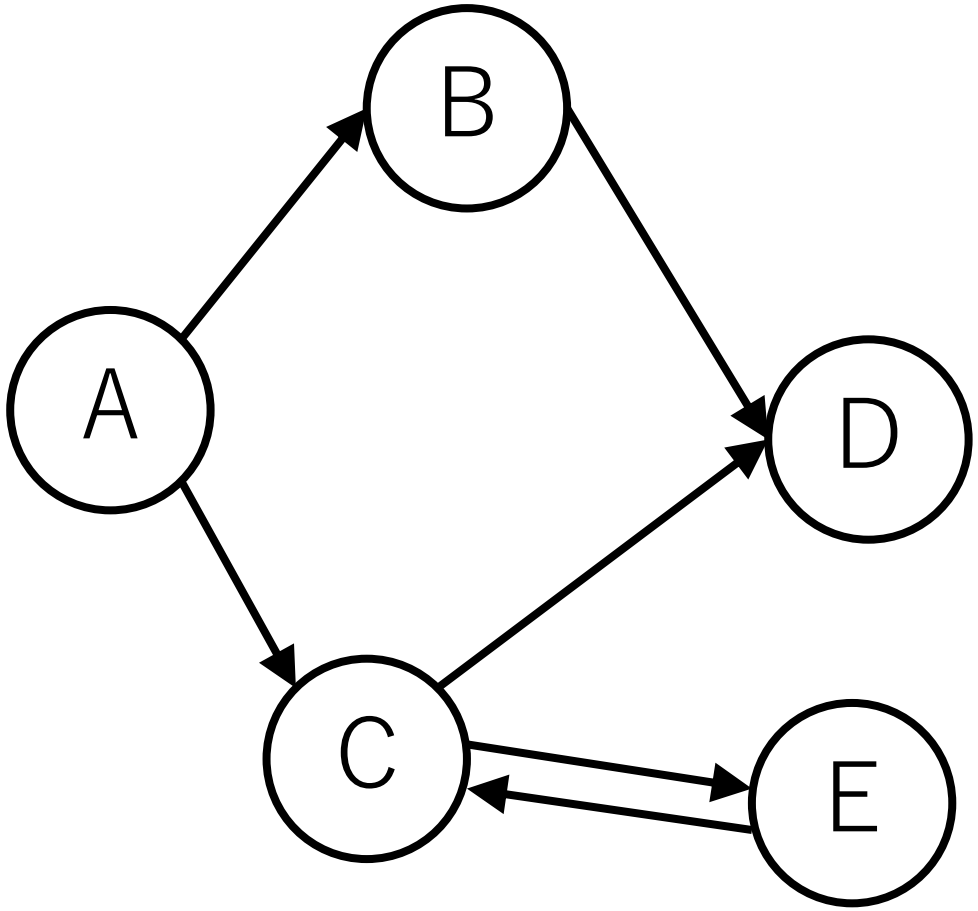
```
edge = [
[0, 1], [1, 0],     # A-B
[0, 2], [2, 0],     # A-C
[1, 3], [3, 1],     # B-D
[2, 3], [3, 2],     # C-D
[2, 4], [4, 2]      # C-E
]
```



隣接リスト（有向グラフ）

各要素が各頂点の接続先を表す.

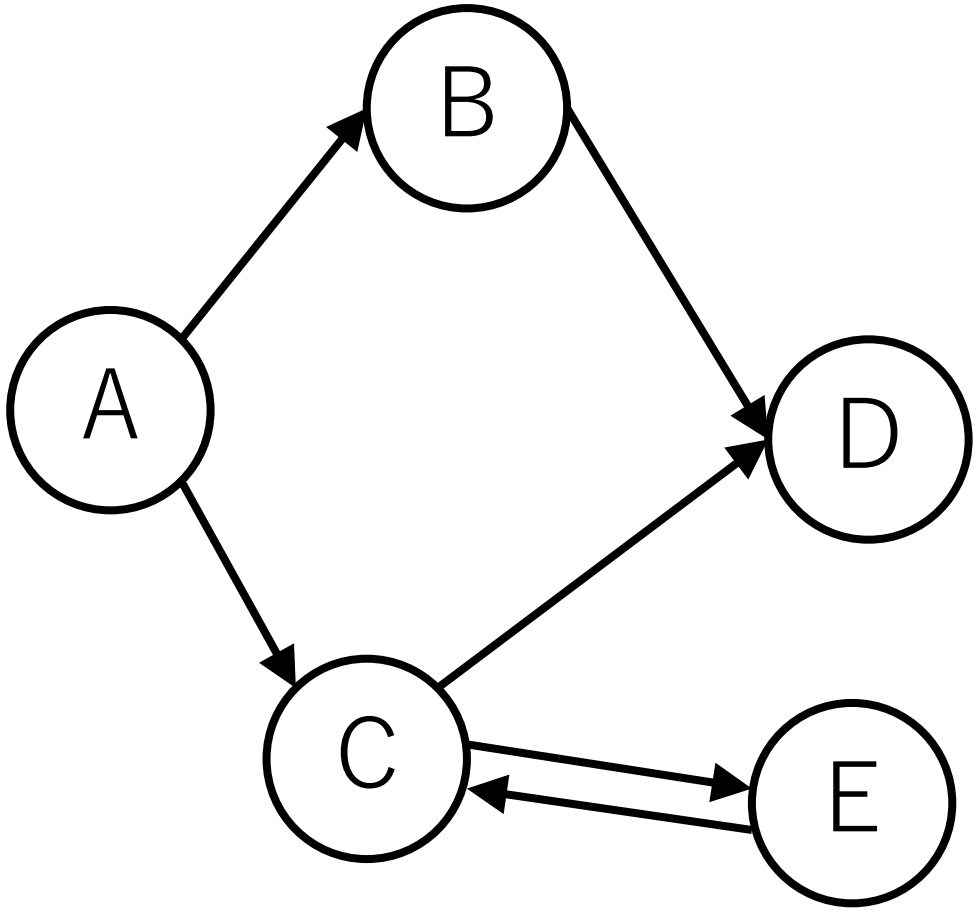
```
edge = [
[1, 2],      #ノードA node[0]
[3],         #ノードB node[1]
[3, 4],      #ノードC node[2]
[],          #ノードD node[3]
[2]          #ノードE node[4]
]
```



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.

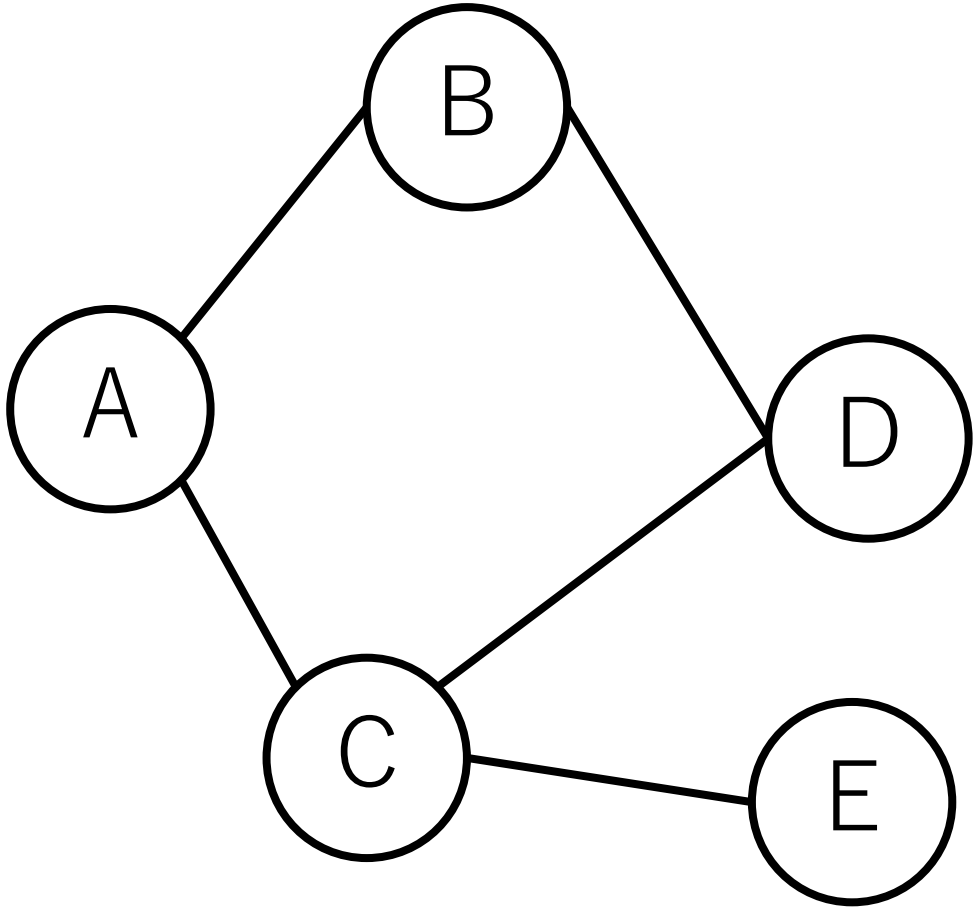
```
edge = [
[0, 1],      # A->B
[0, 2],      # A->C
[1, 3],      # B->D
[2, 3],      # C->D
[2, 4],      # C->E
[4, 2]       # E->C
]
```



隣接行列（無向グラフ）

無向グラフの場合は対称行列.

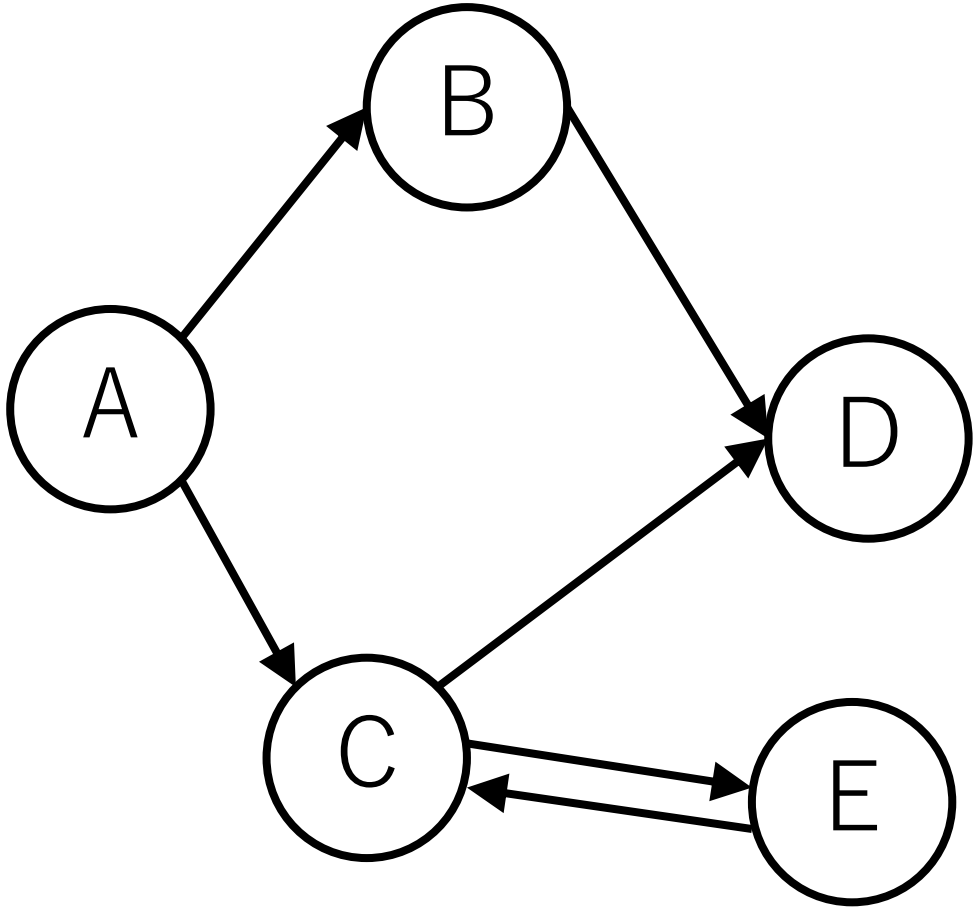
```
edge = [
        [0, 1, 1, 0, 0], #ノードA
        [1, 0, 0, 1, 0], #ノードB
        [1, 0, 0, 1, 1], #ノードC
        [0, 1, 1, 0, 0], #ノードD
        [0, 0, 1, 0, 0]  #ノードE
]
```



隣接行列（有向グラフ）

繋がっている向きにのみ値を持つ.

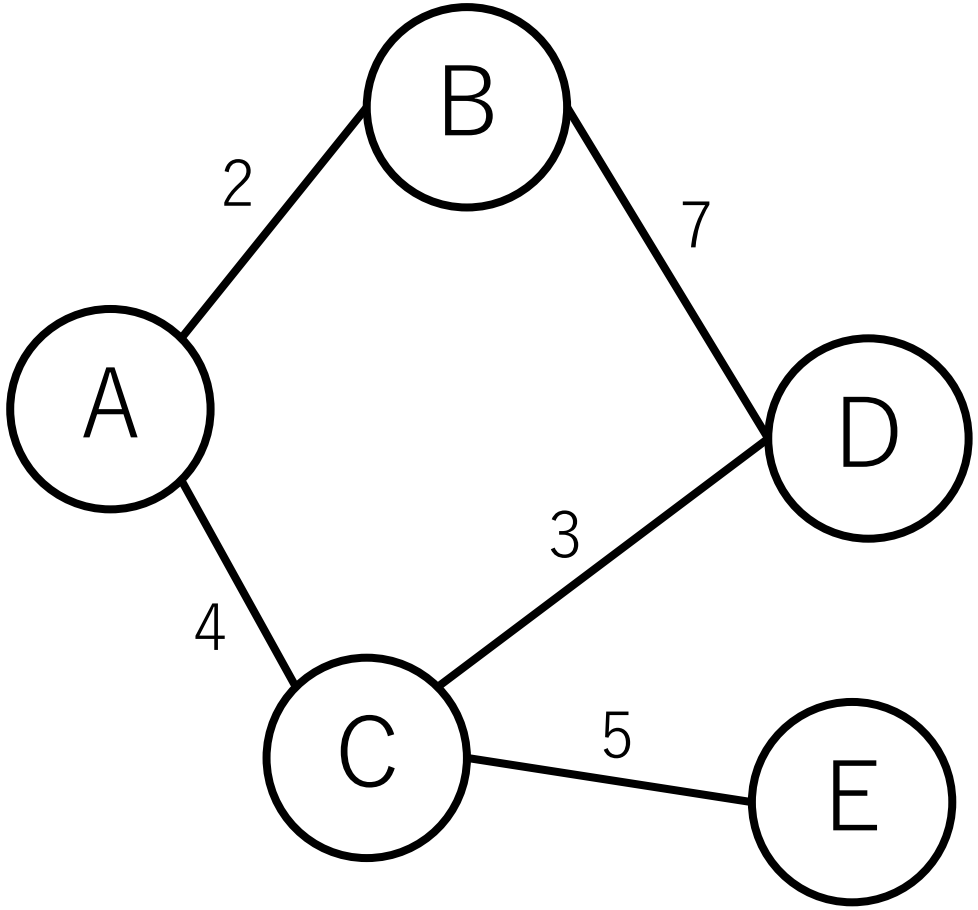
```
edge = [
        [0, 1, 1, 0, 0], #ノードA
        [0, 0, 0, 1, 0], #ノードB
        [0, 0, 0, 1, 1], #ノードC
        [0, 0, 0, 0, 0], #ノードD
        [0, 0, 1, 0, 0]  #ノードE
]
```



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.

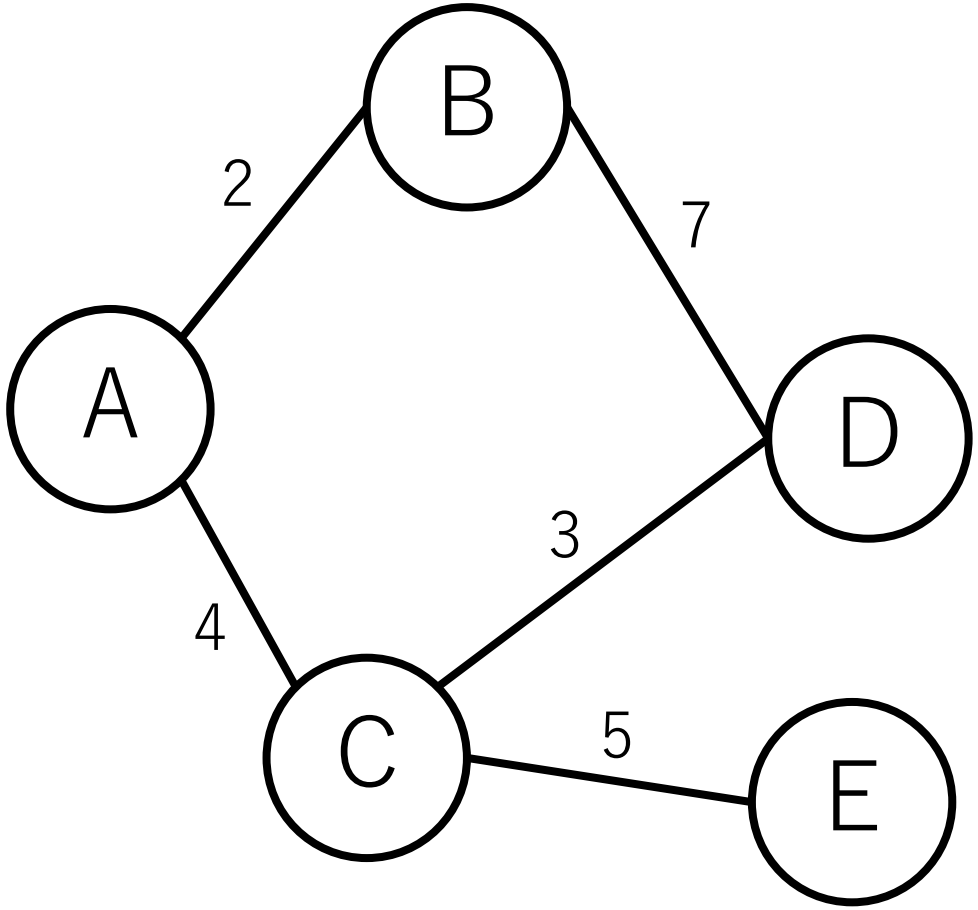
```
edge = [
[[1, 2], [2, 4]],          #ノードA
[[0, 2], [3, 7]],          #ノードB
[[0, 4], [3, 3], [4, 5]], #ノードC
[[1, 7], [2, 3]],          #ノードD
[[2, 5]]                   #ノードE
]
```



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．

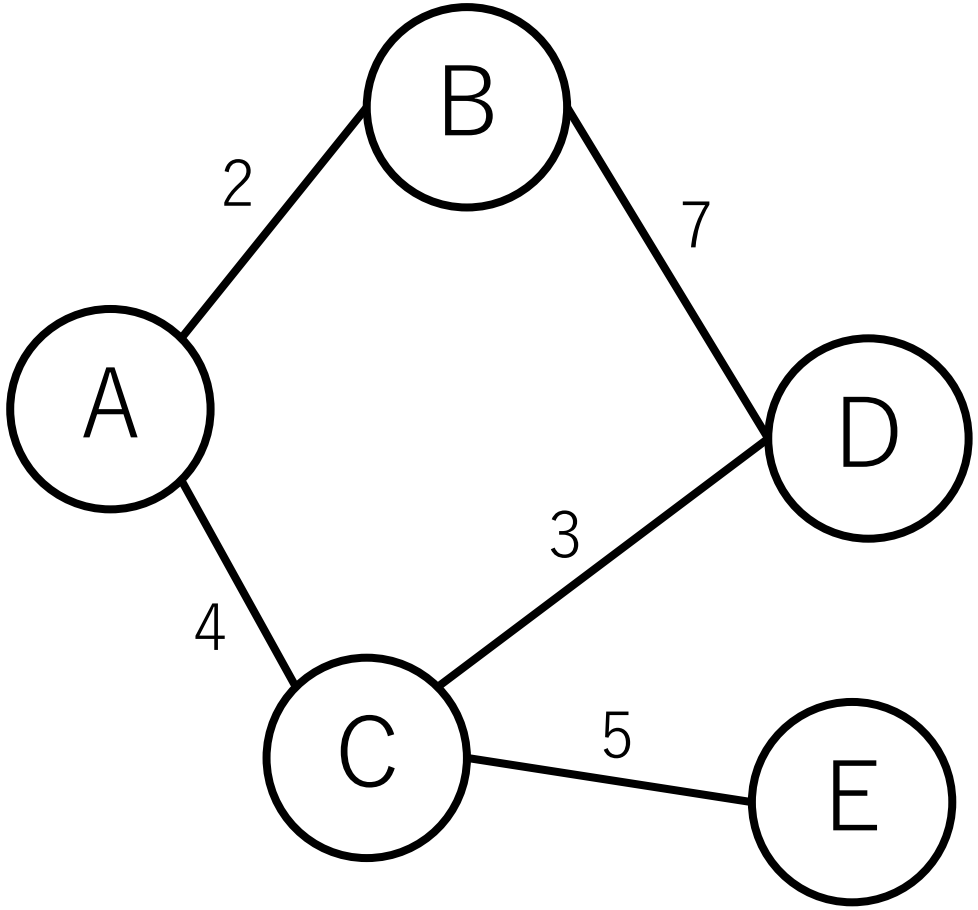
```
edge = [
[0, 1, 2], [1, 0, 2],      # A-B
[0, 2, 4], [2, 0, 4],      # A-C
[1, 3, 7], [3, 1, 7],      # B-D
[2, 3, 3], [3, 2, 3],      # C-D
[2, 4, 5], [4, 2, 5]       # C-E
]
```



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個数回の操作で可能．
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．

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.

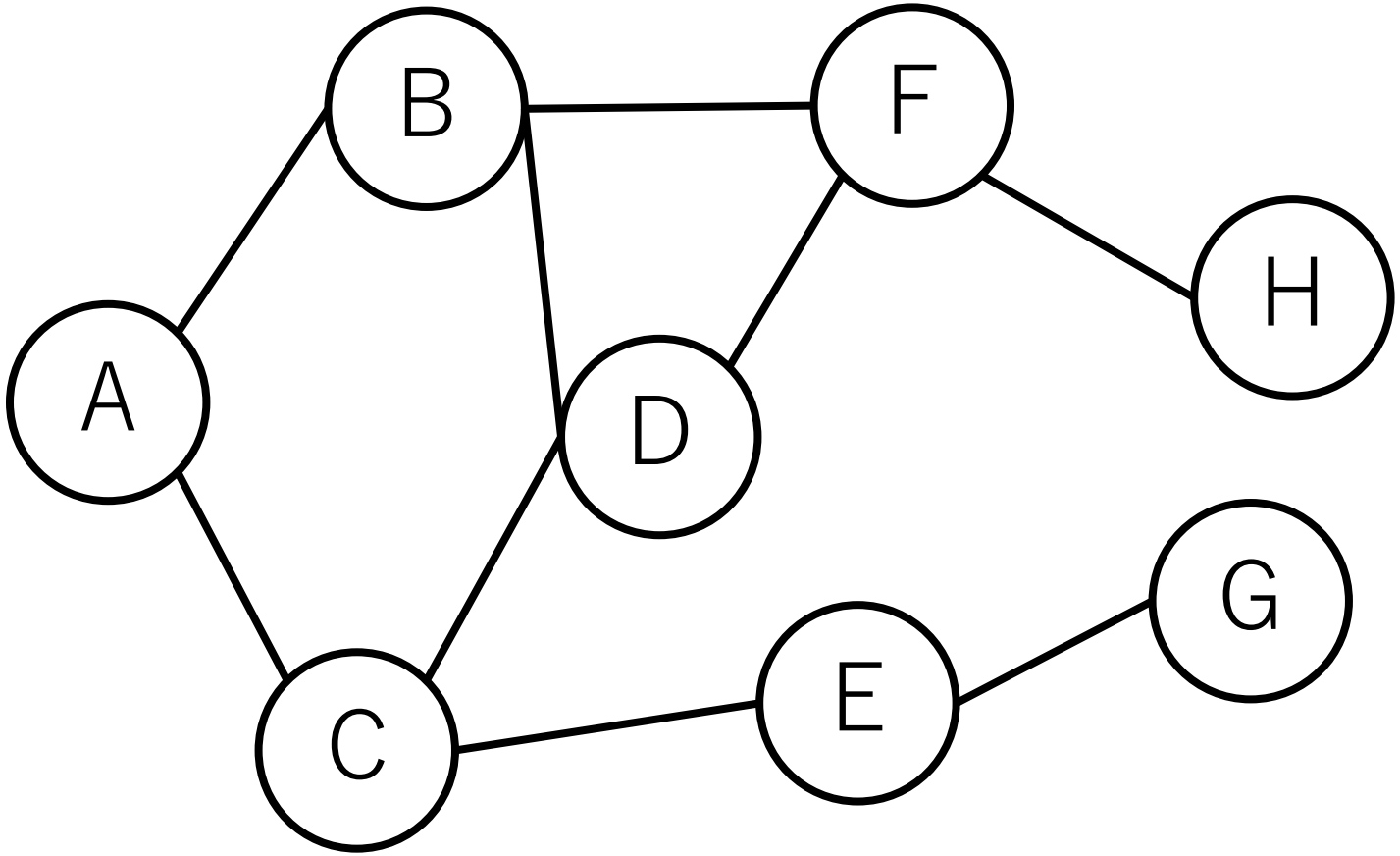
あるノードとあるノードを結ぶ経路が存在するかを確かめたい.

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）

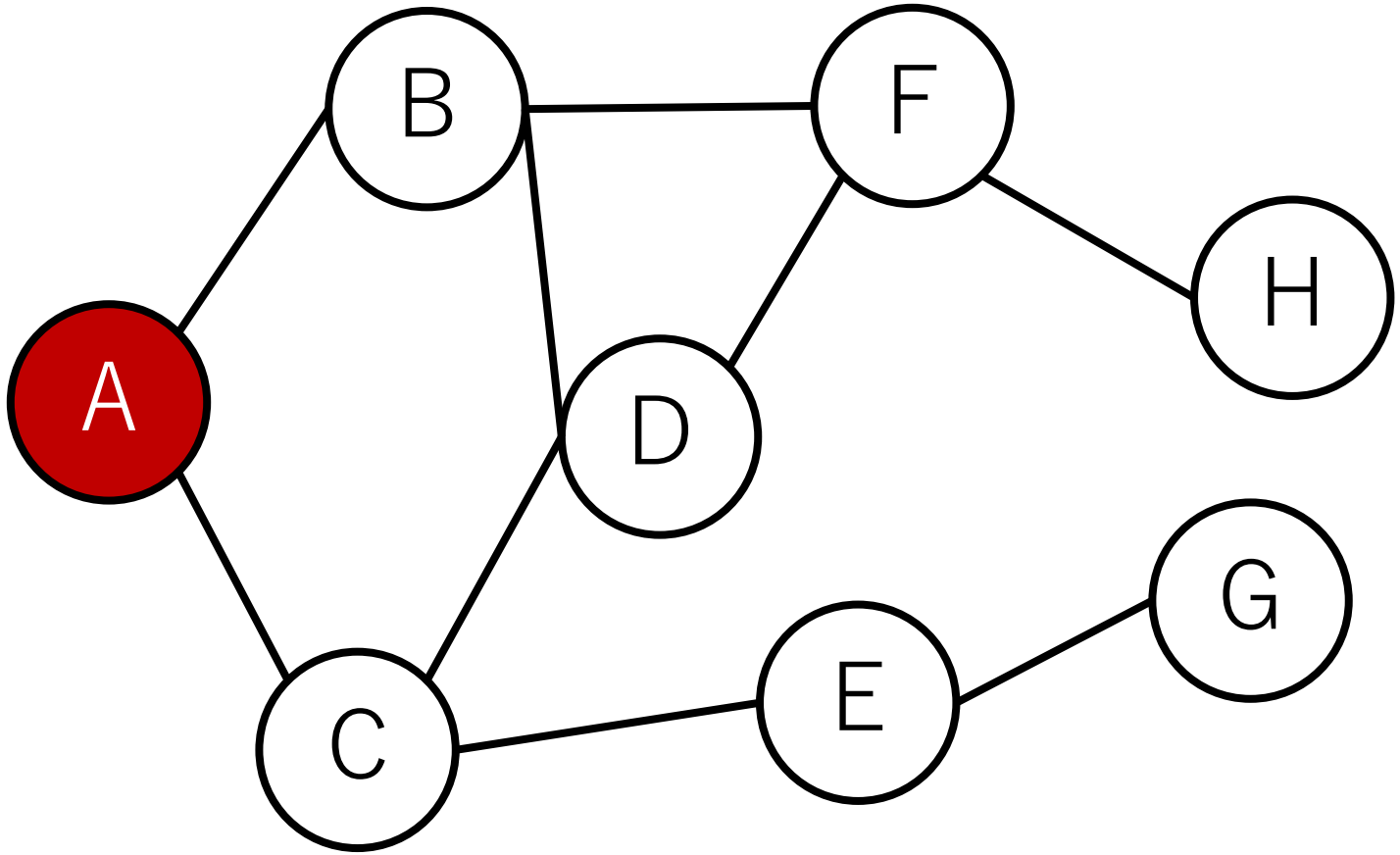
後戻りしないように，可能性のあるルート
全てにおいて1ステップずつ行くパターン．

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．

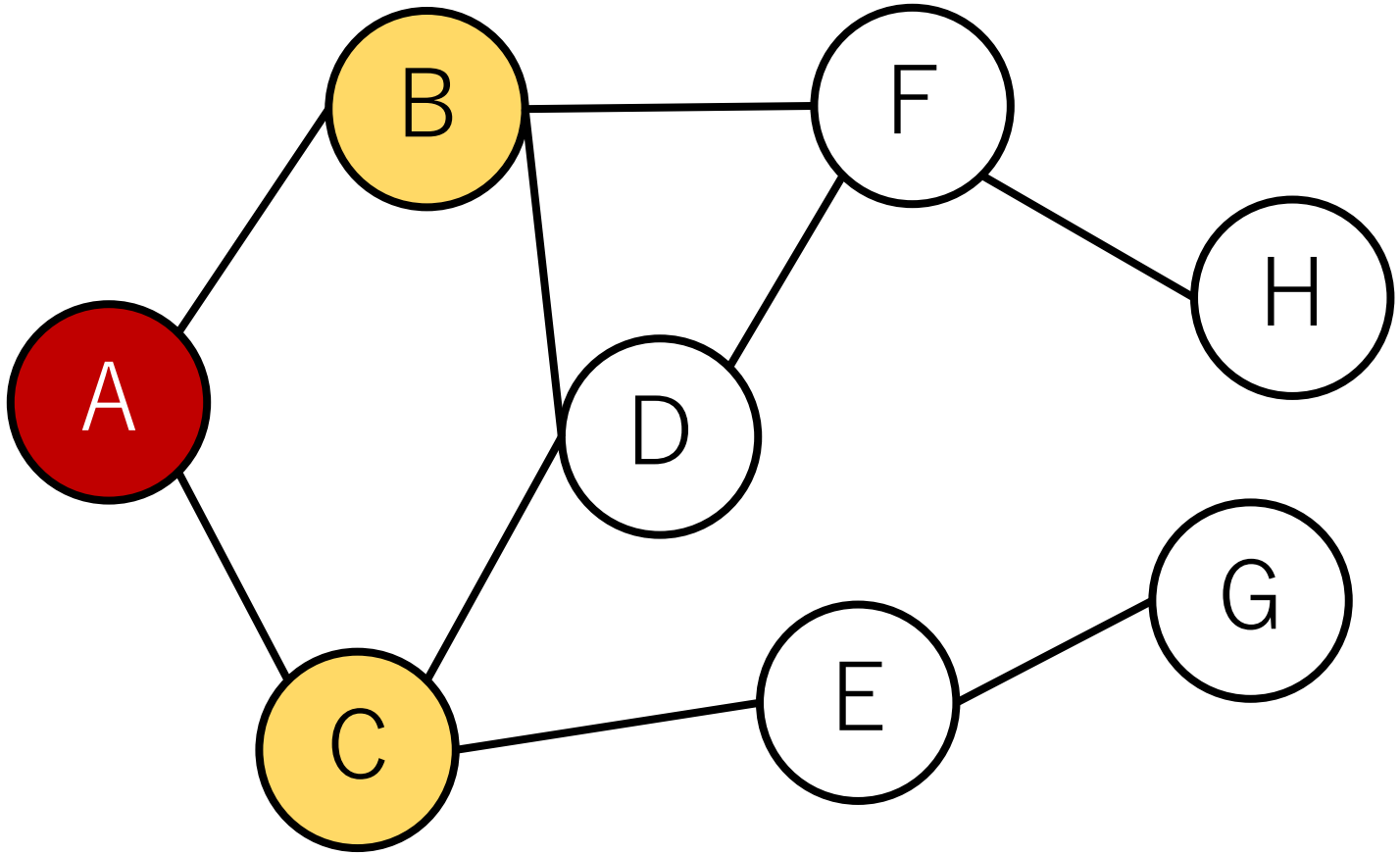
BFSの例

Aからスタート.



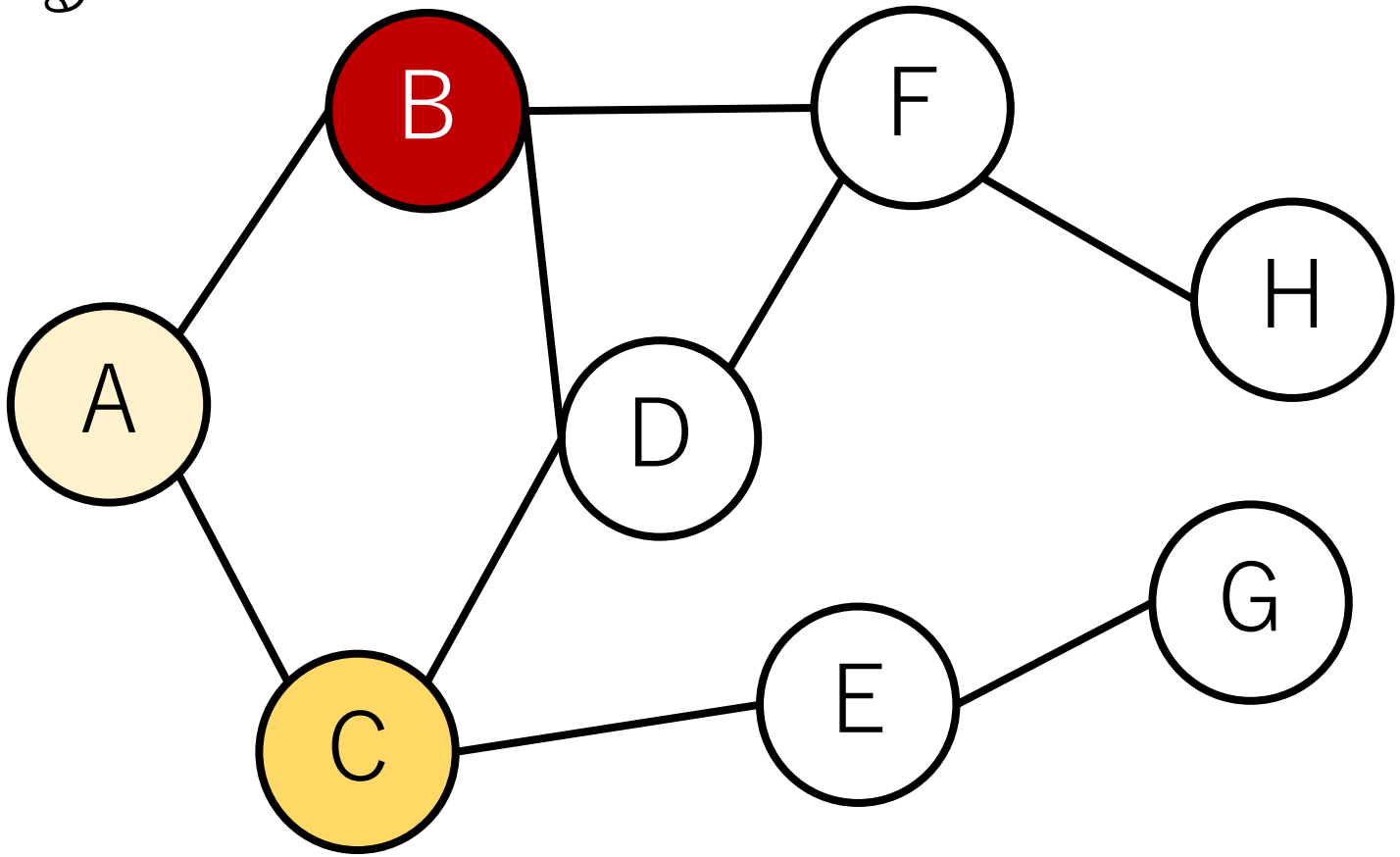
BFSの例

1ステップでつながっているのは，BとC.



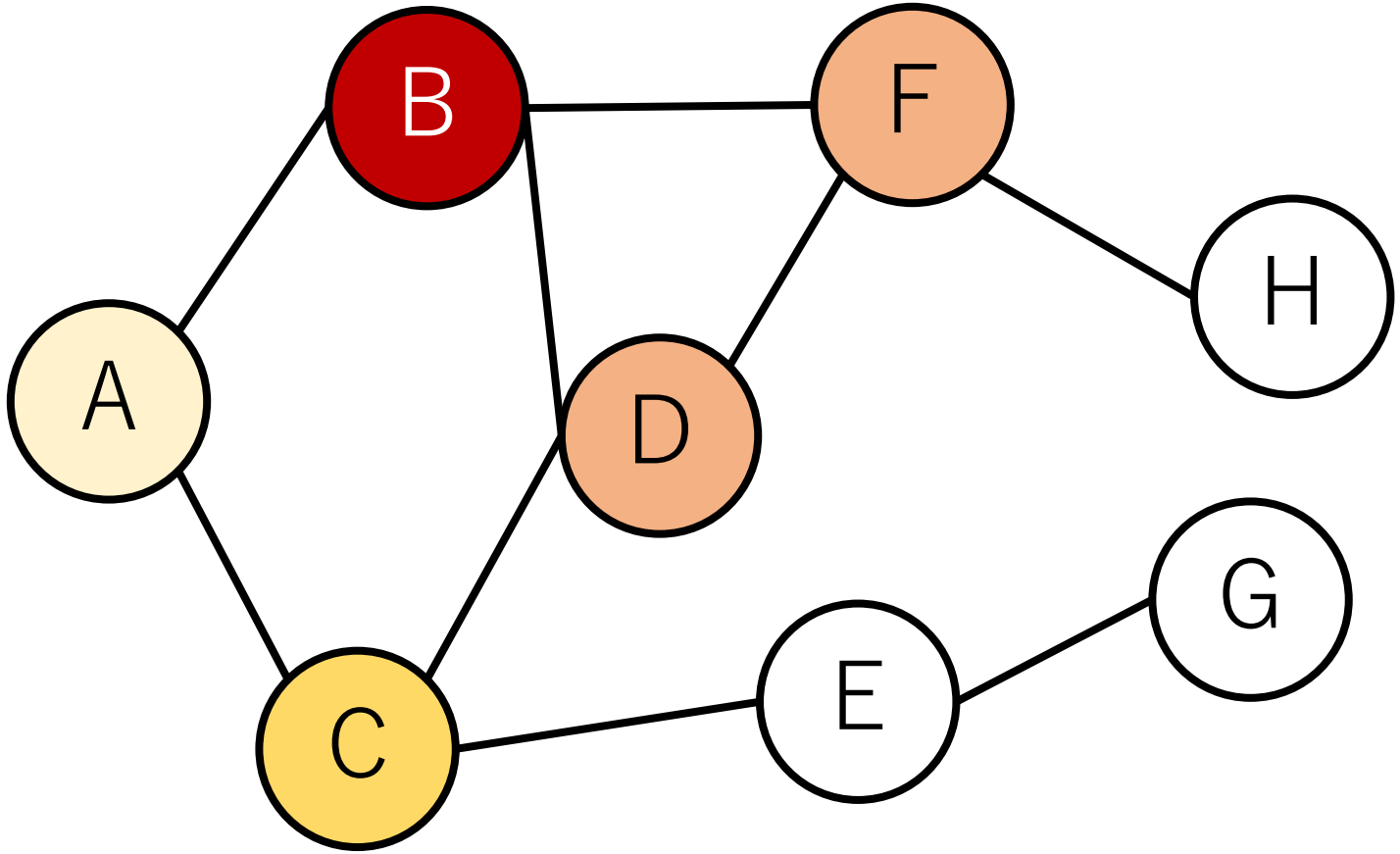
BFSの例

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．）



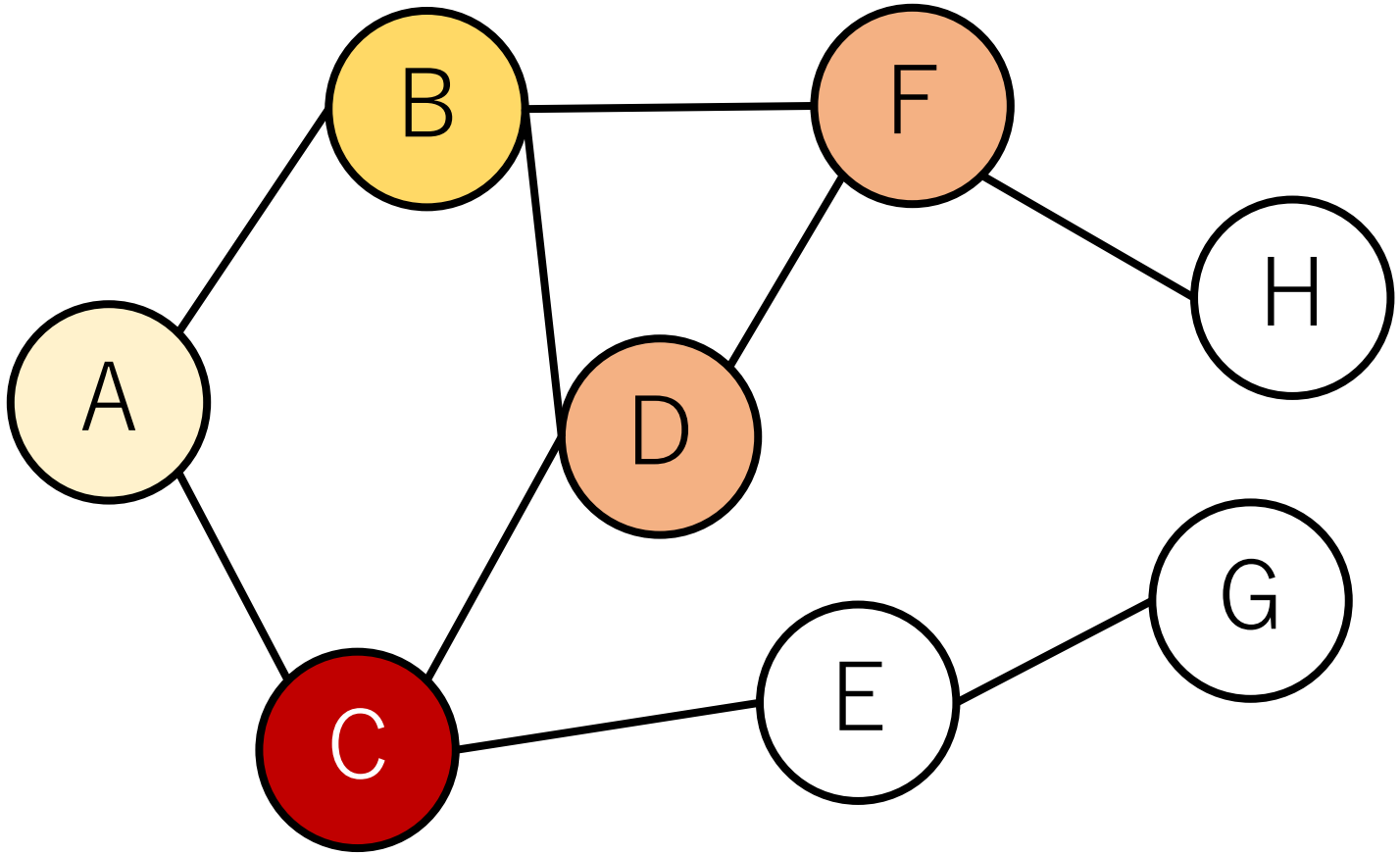
BFSの例

DとFを発見.



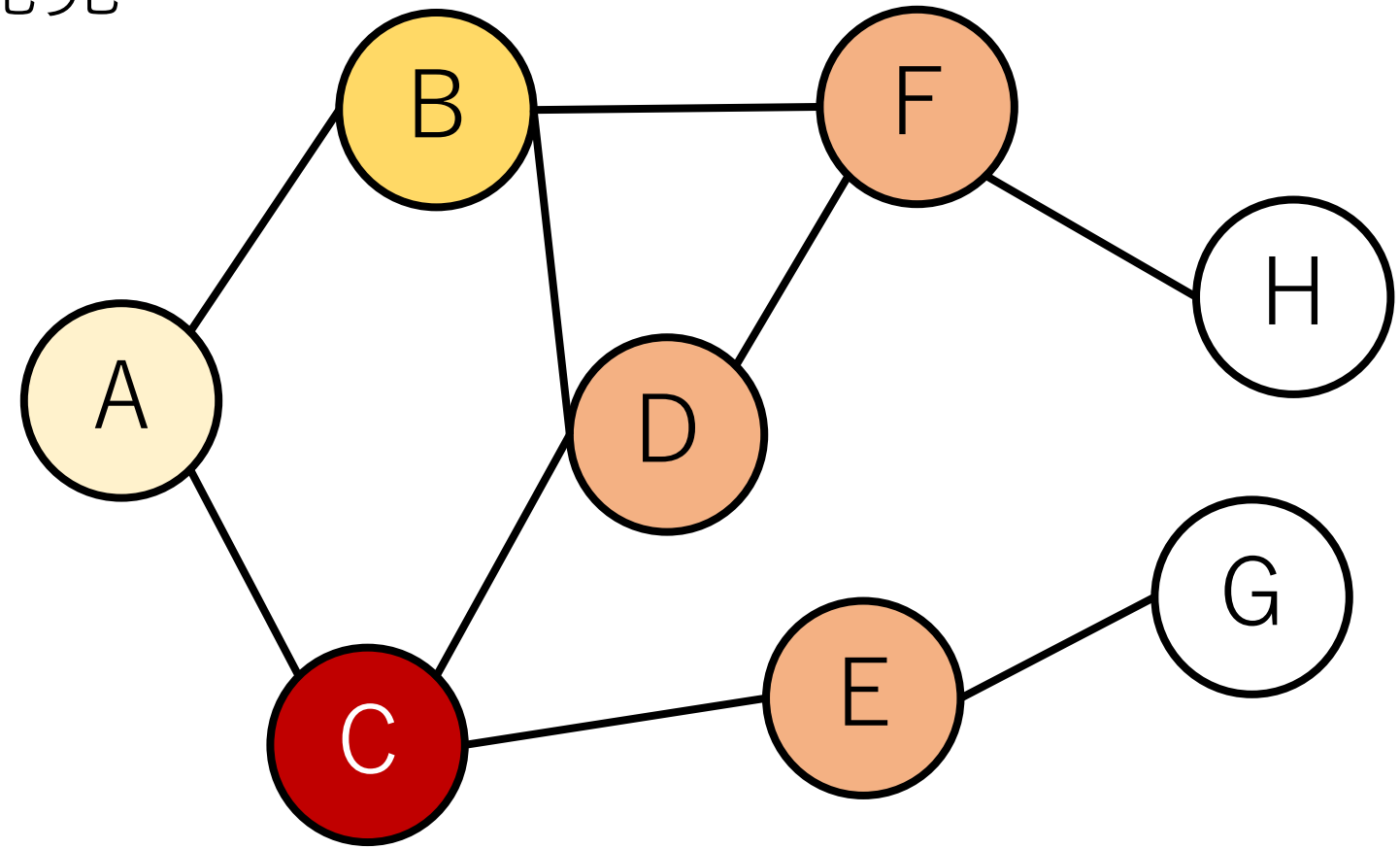
BFSの例

Cに移る.



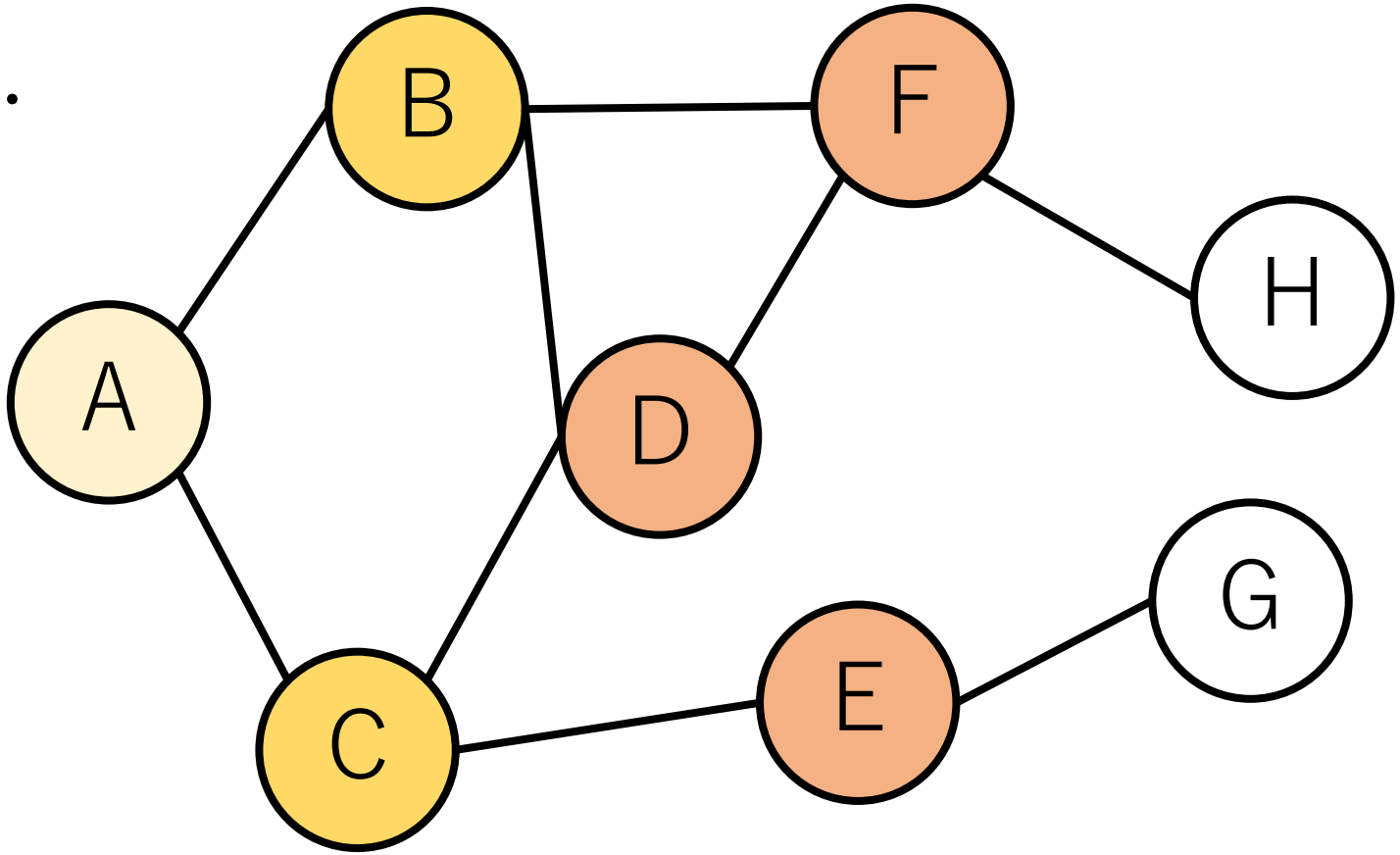
BFSの例

Eを新しく発見.　(Dは発見済み)



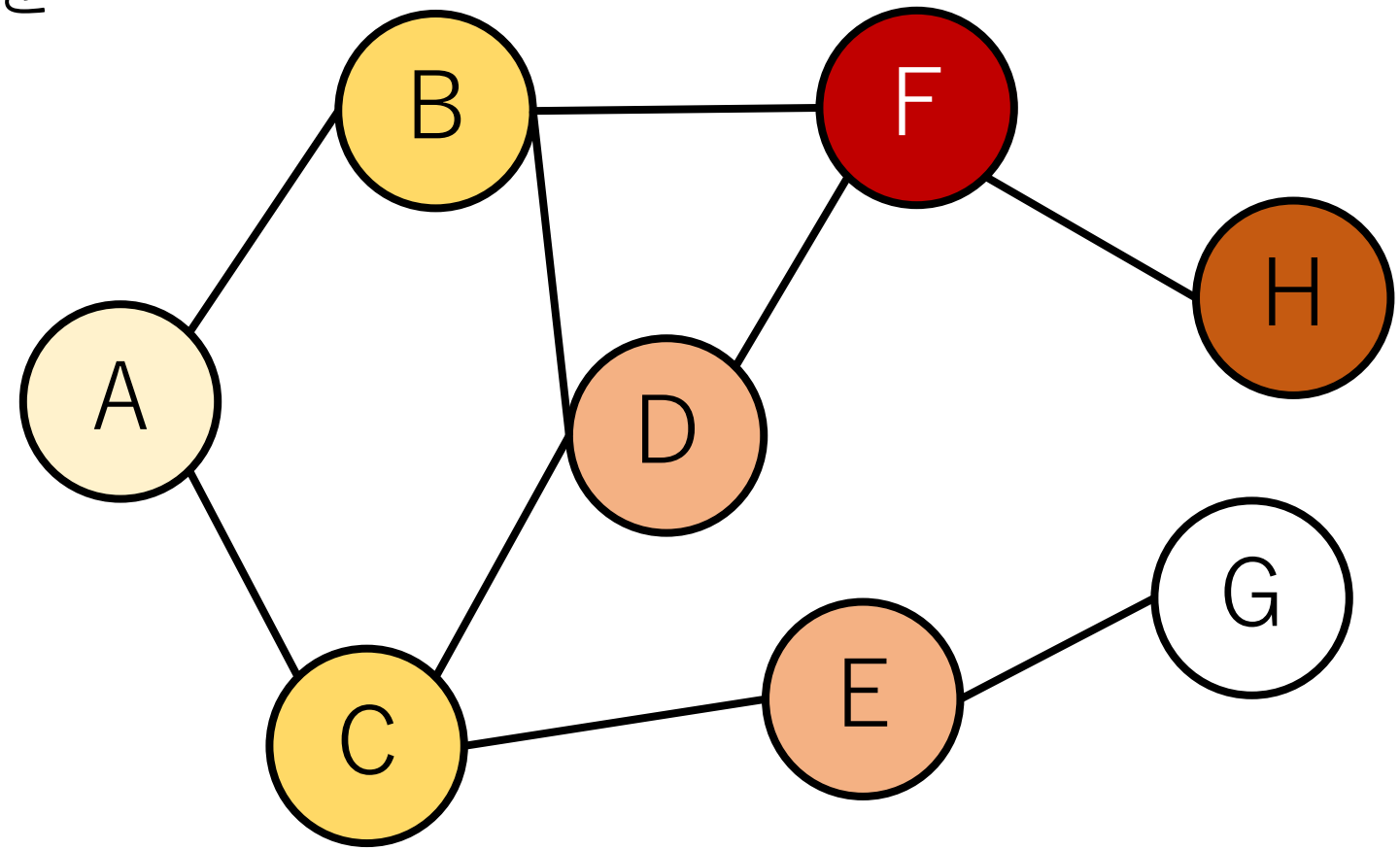
BFSの例

これでB，Cから辿れる
ノードの探索は全部終了.



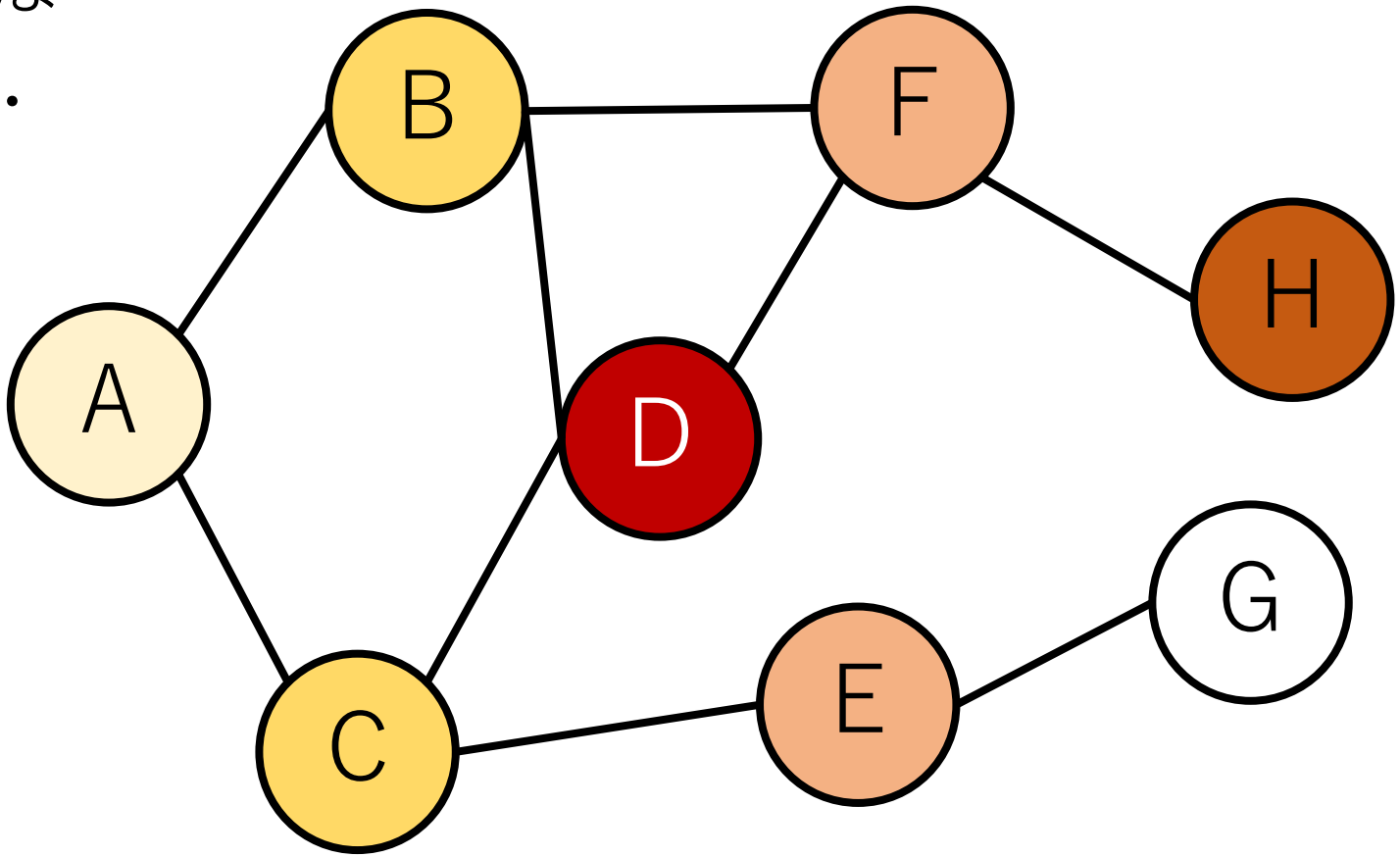
BFSの例

Fに移って，同様の探索を
行い，Hを発見.



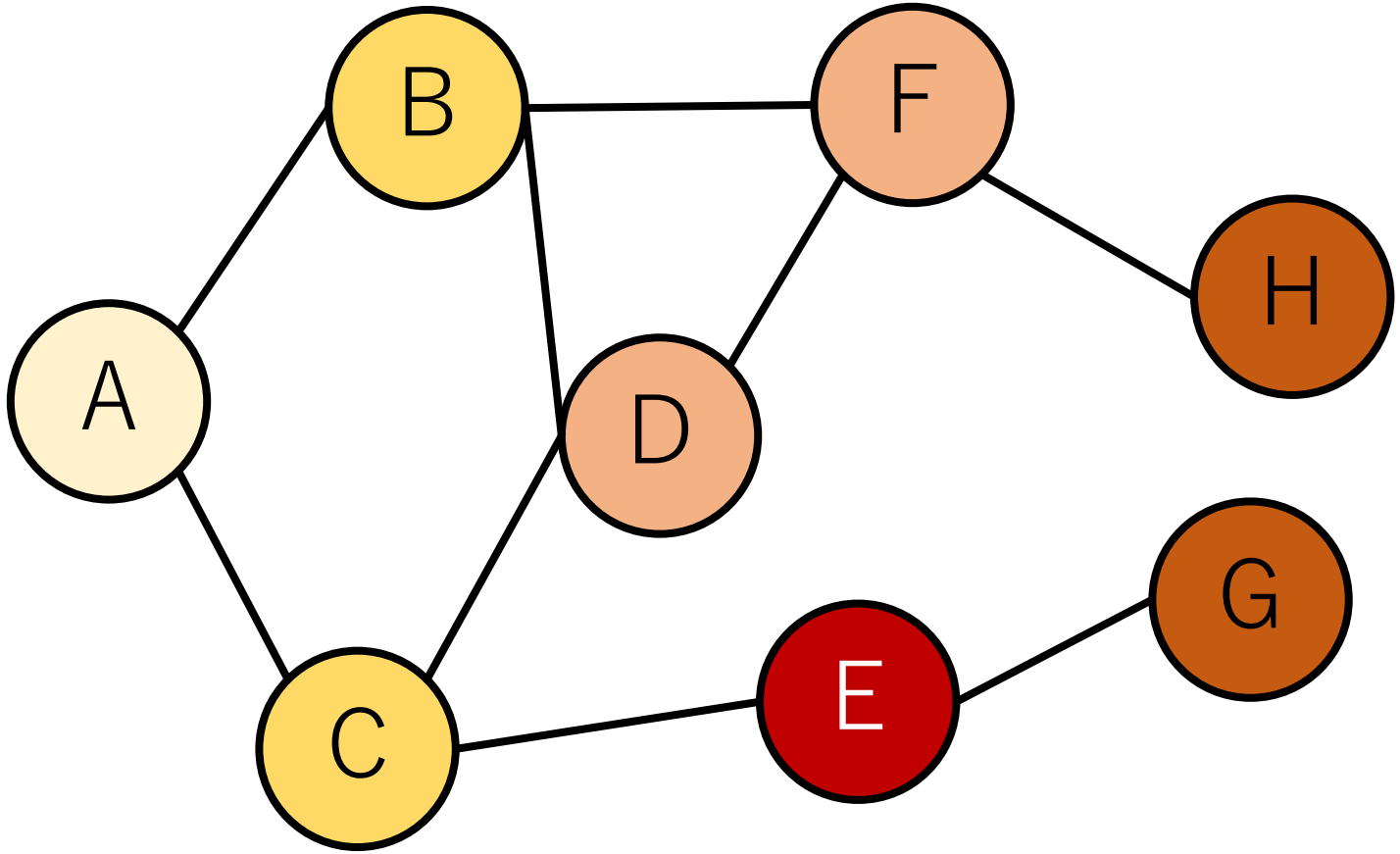
BFSの例

Dからは未発見のノードはないので，何も行わない．



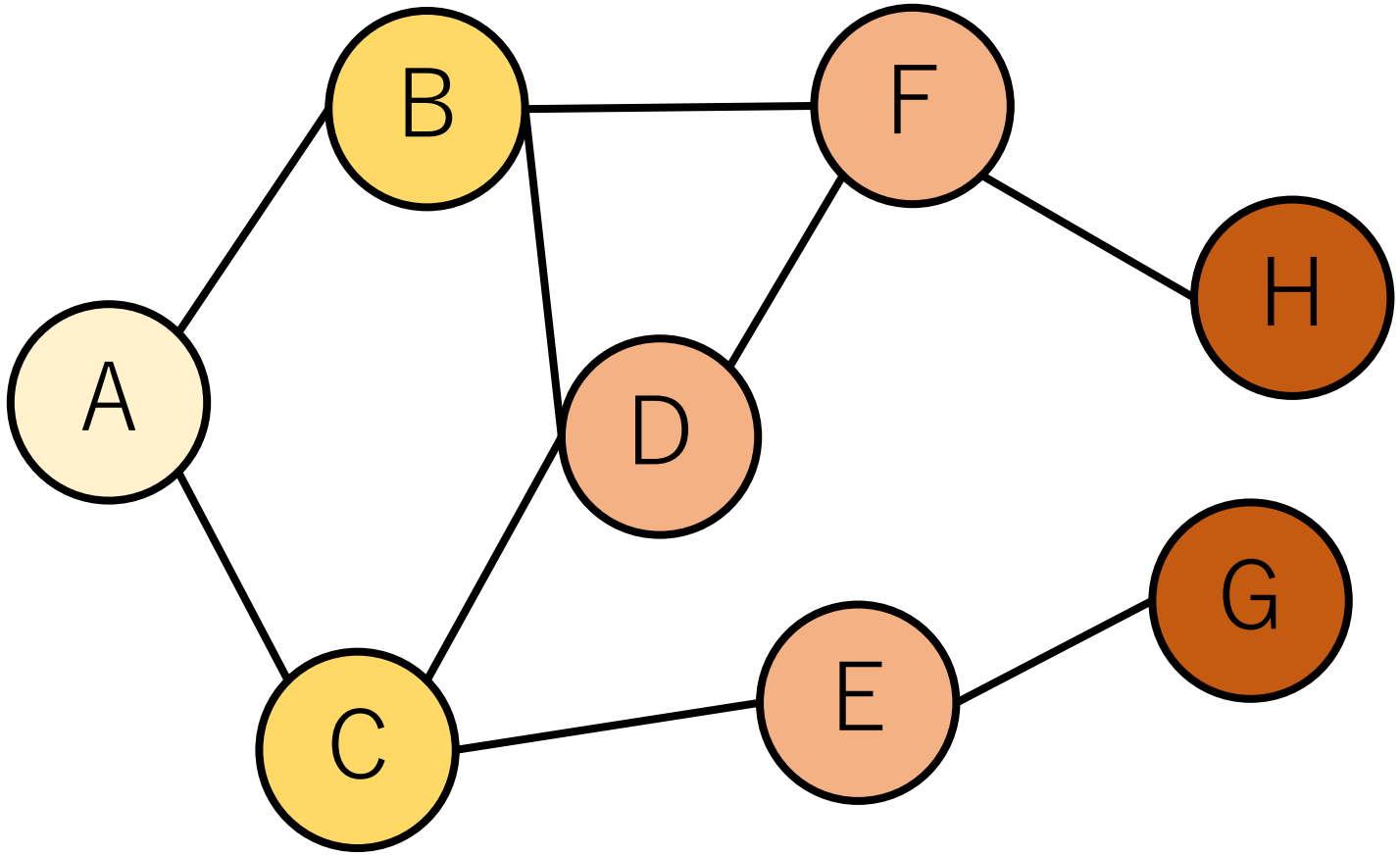
BFSの例

Eに移ってGを発見.



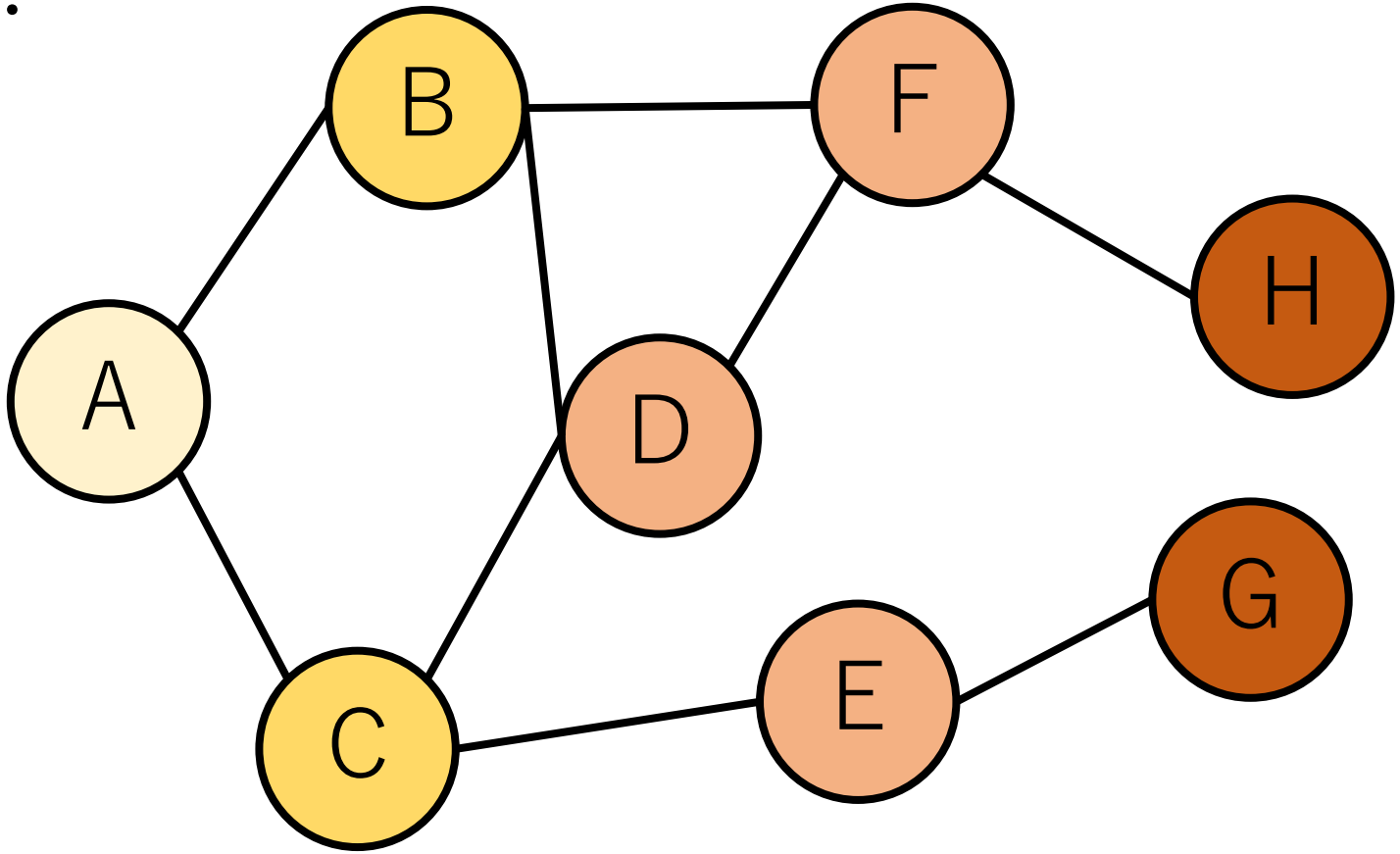
BFSの例

GとHの先には未発見のノードはない.



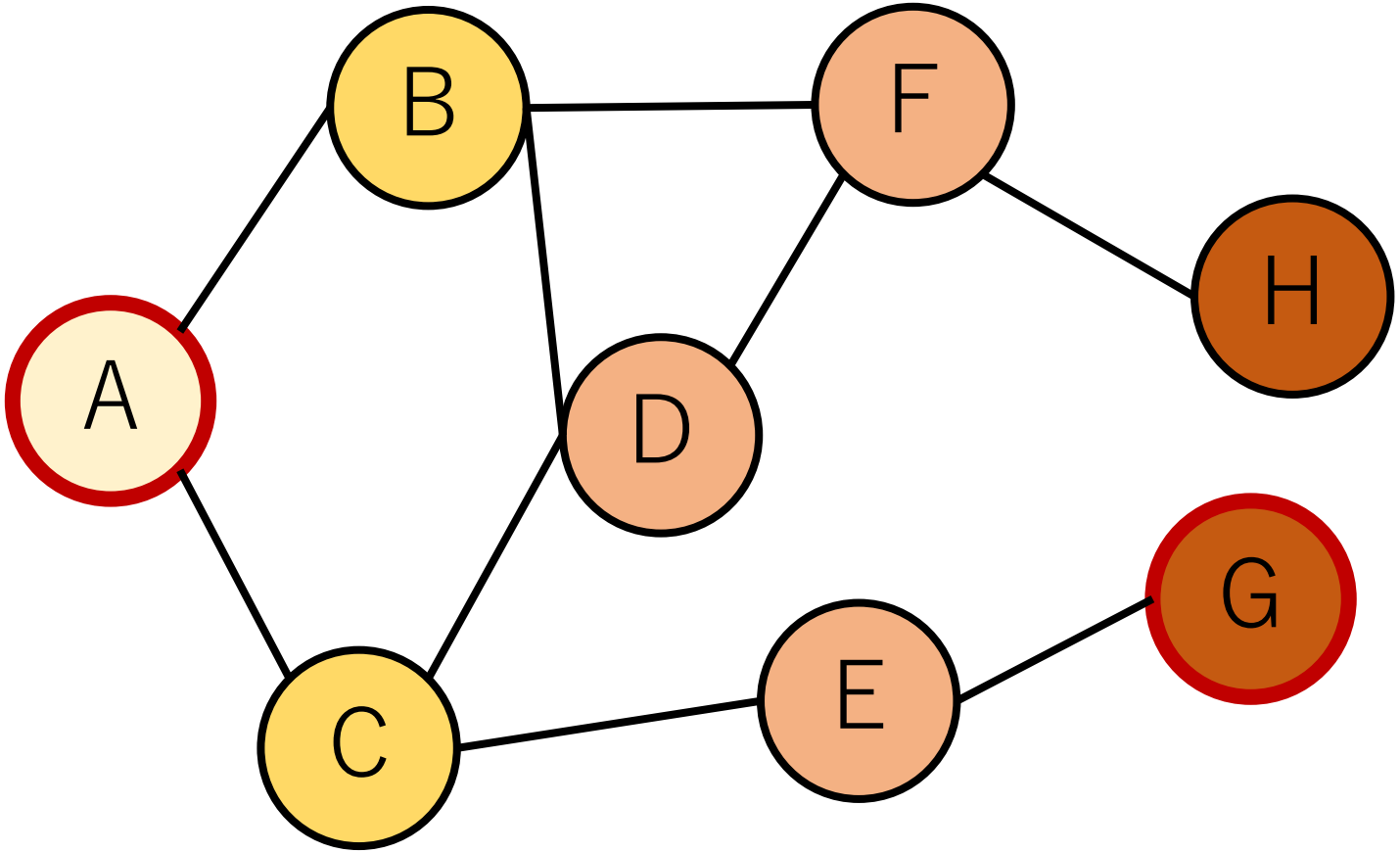
BFSの例

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.

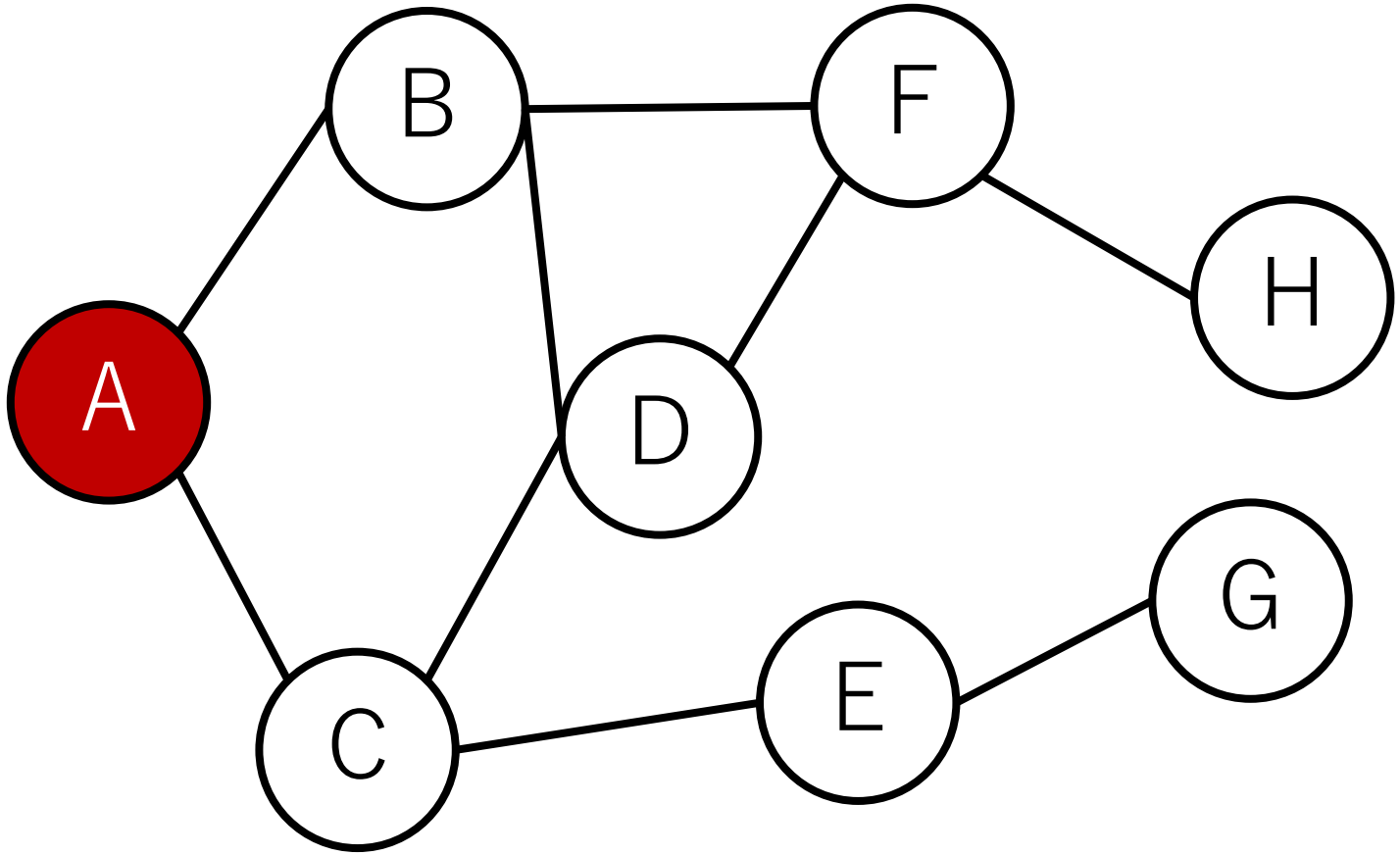
キューから取り出す：
　　取り出したノードに移動する.

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.



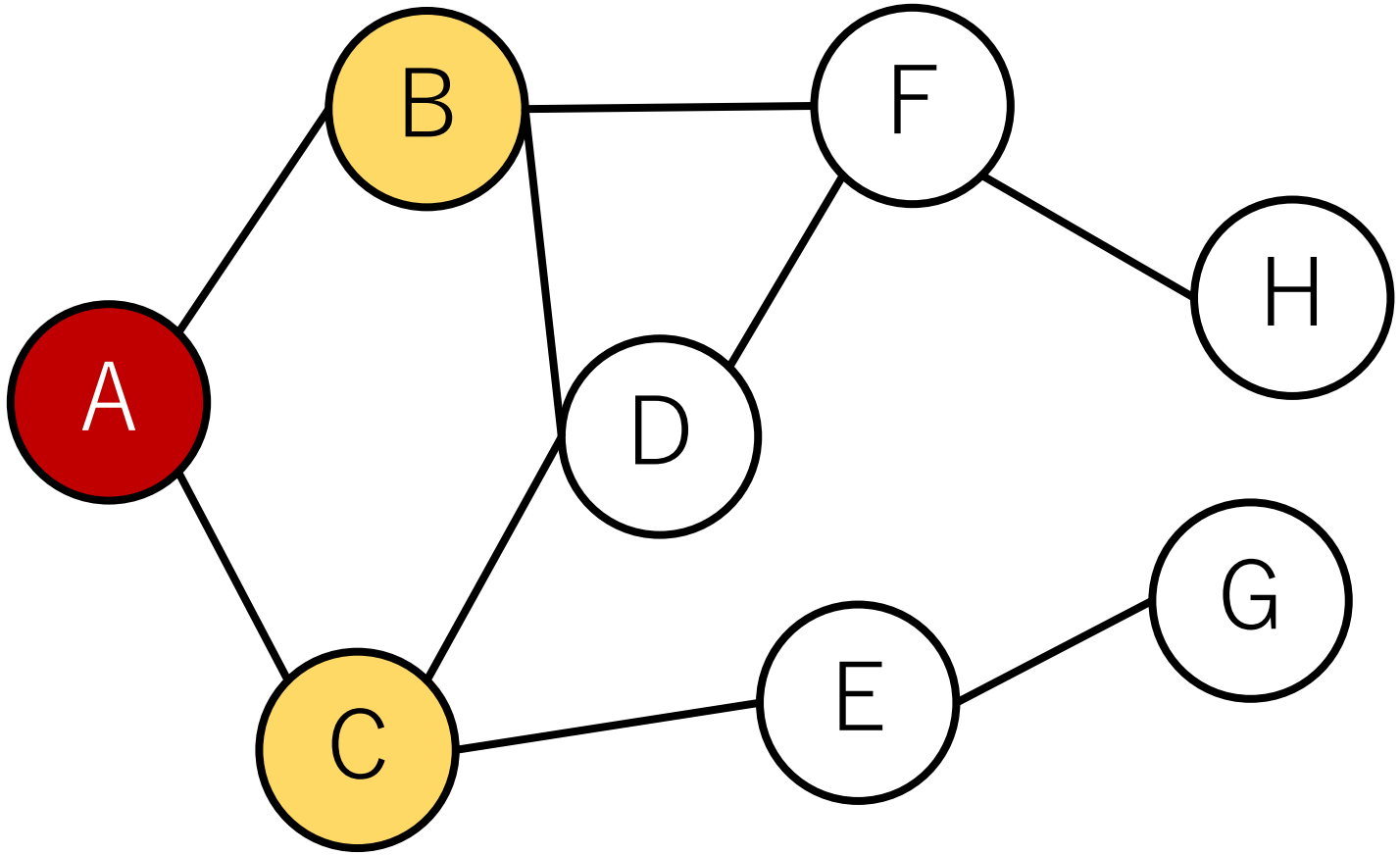
A							

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.

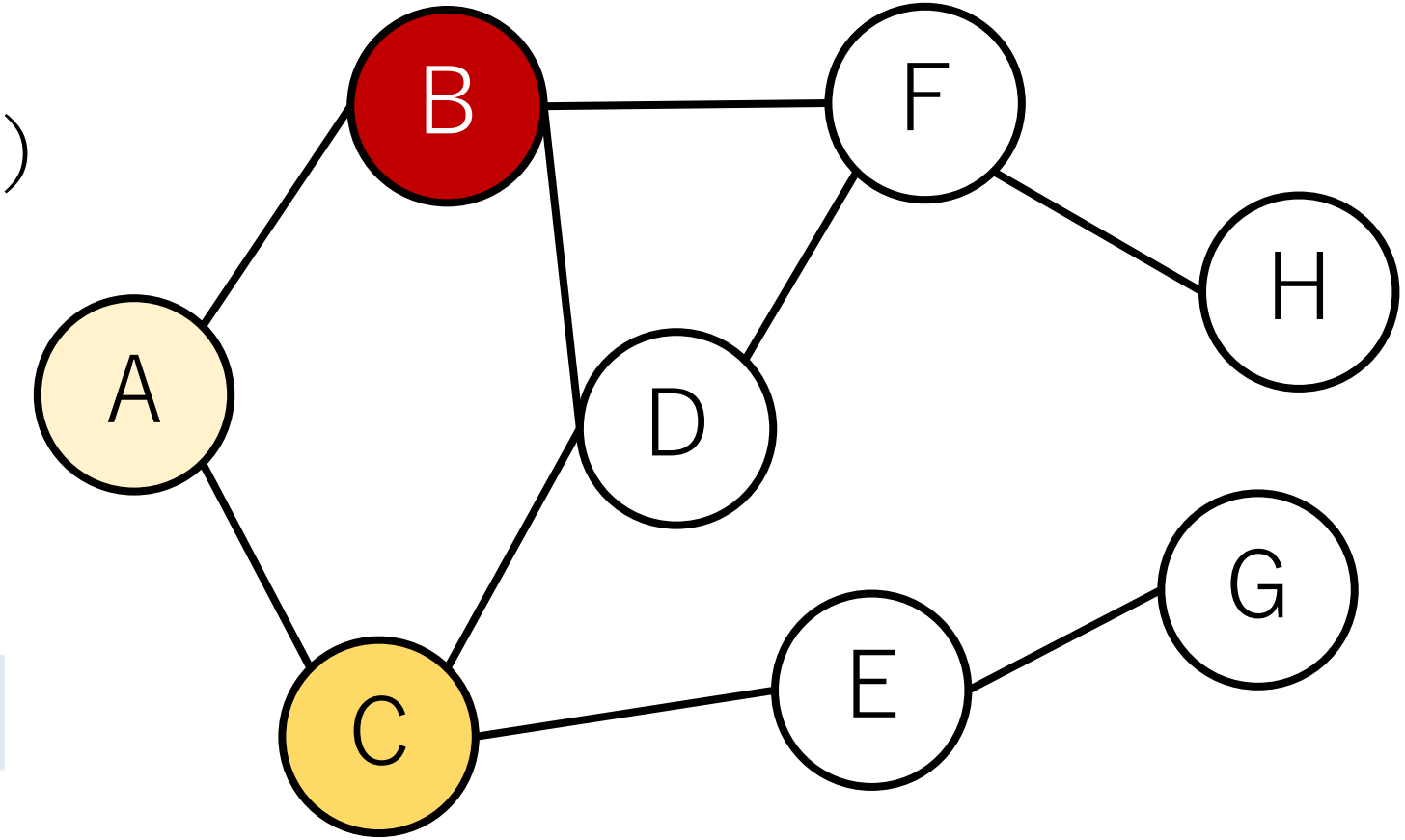
A	**B**	**C**					



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から始めてもよい.）

キューから取り出す.



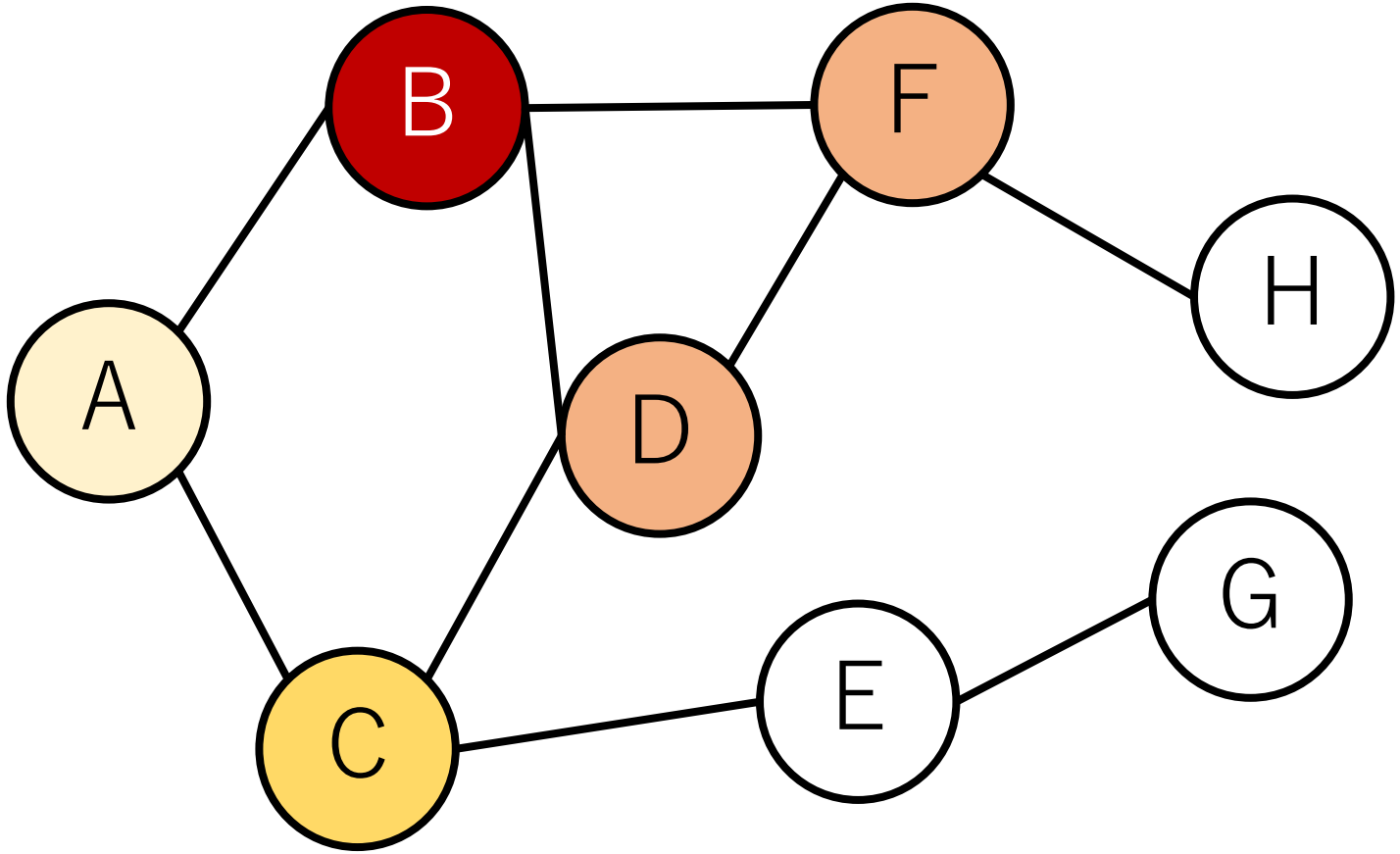
A	**B**	C					

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			

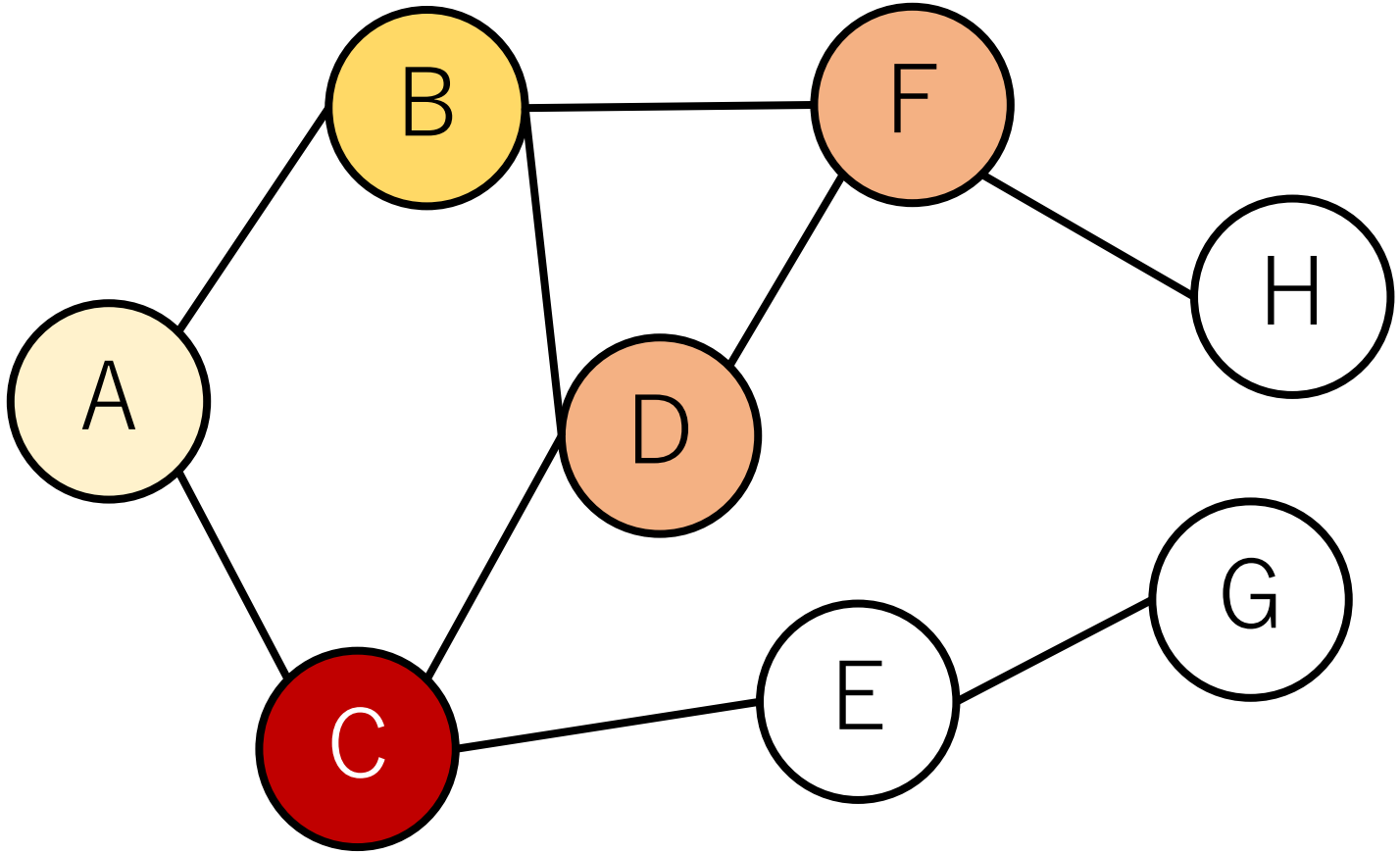


BFSの例＆キューのダンプ

Cに移る.

キューから取り出す.

A	B	**C**	F	D			

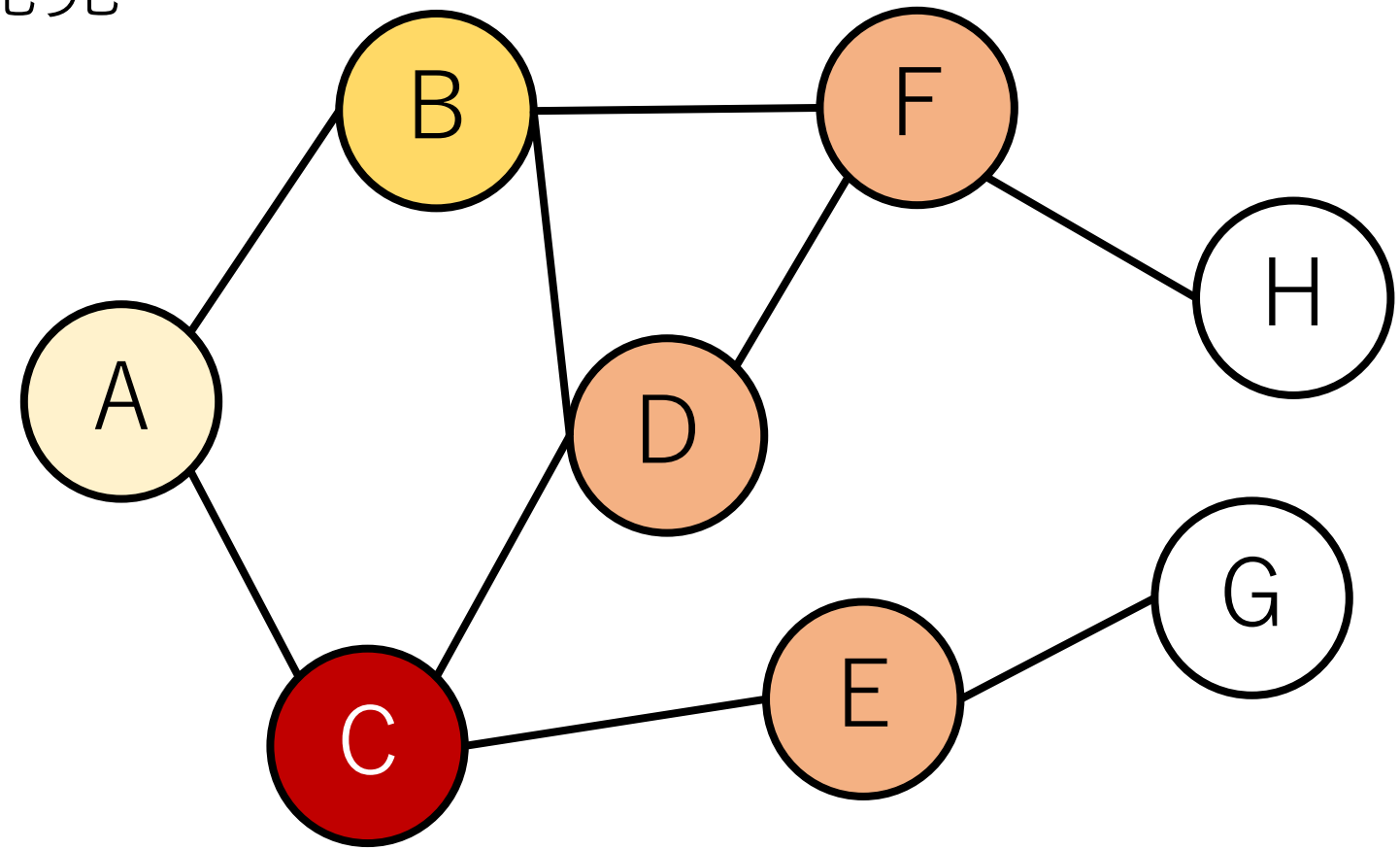


BFSの例＆キューのダンプ

Eを新しく発見．（Dは発見済み）

Eをキューに入れる．

A	B	C	F	D	E		

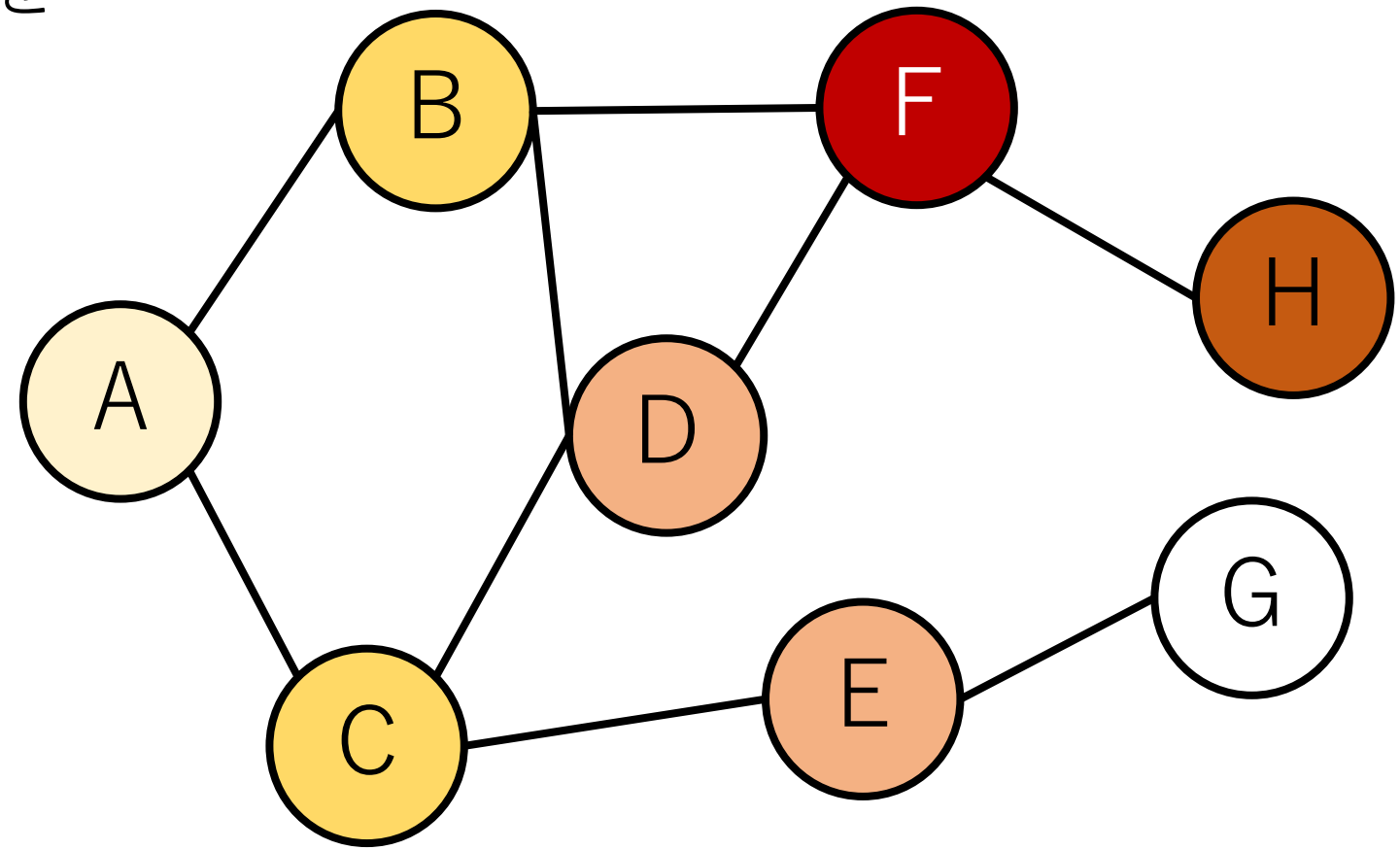


BFSの例＆キューのダンプ

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.

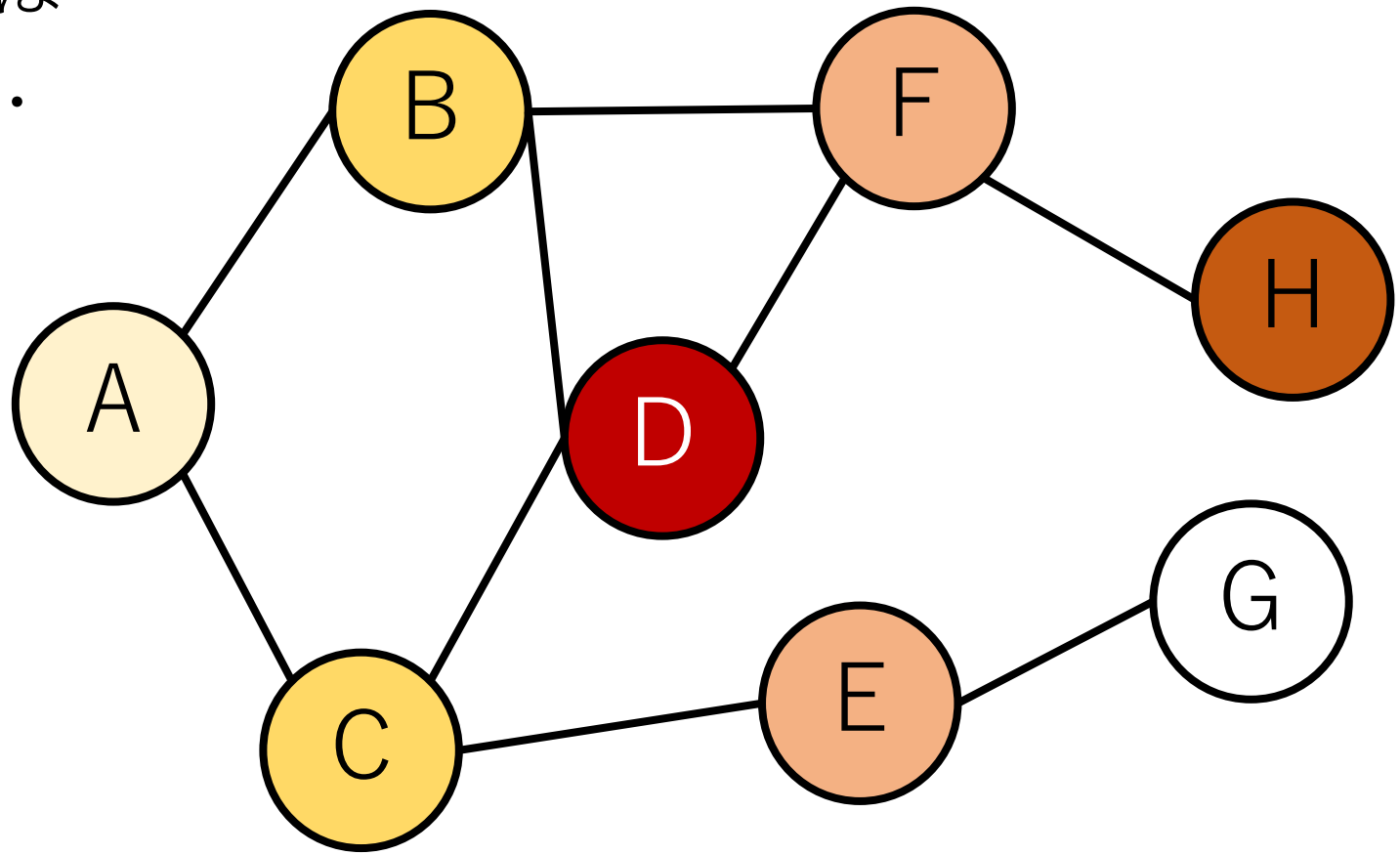
A	B	C	**F**	D	E	H	



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．



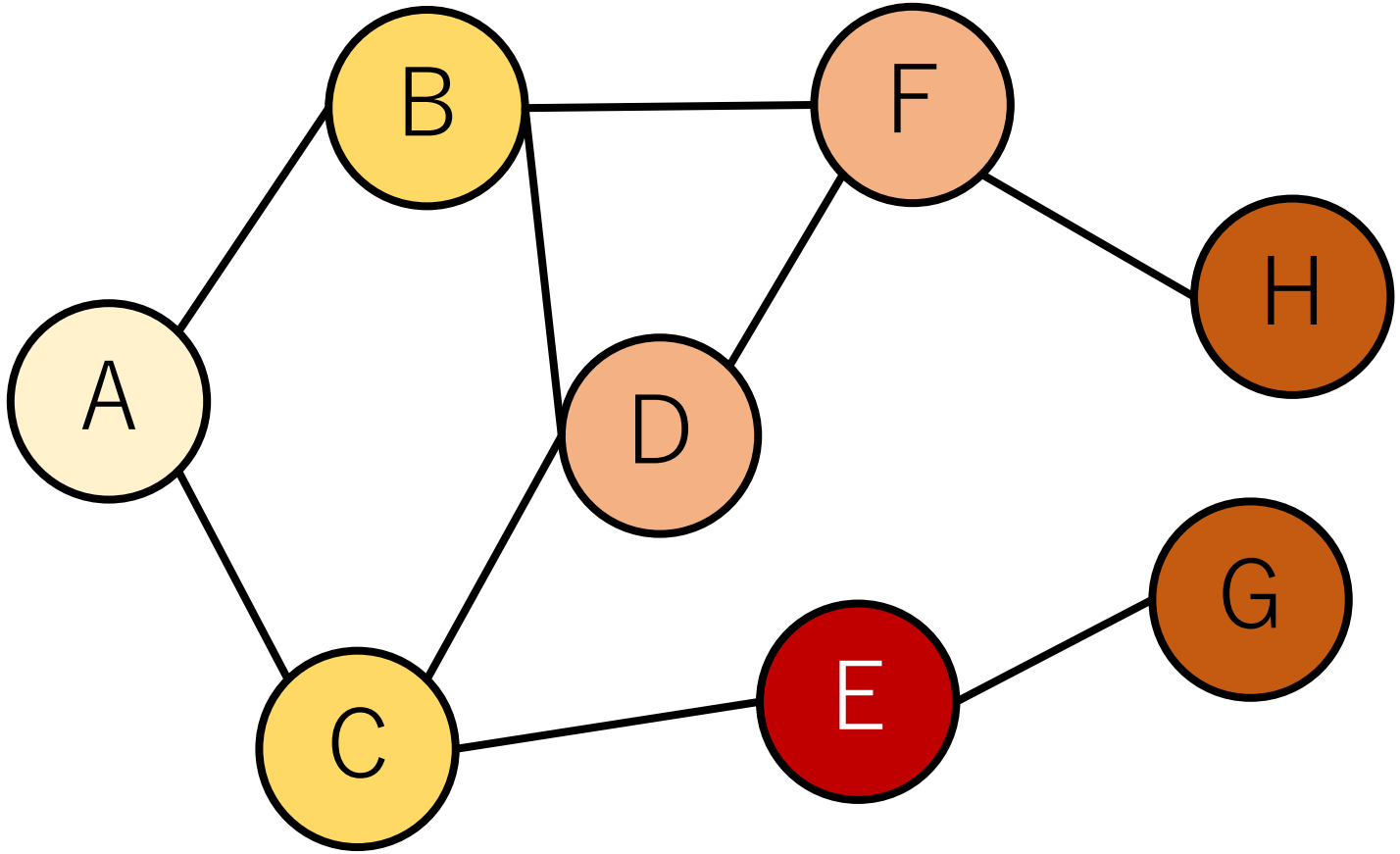
A	B	C	F	**D**	E	H	

BFSの例＆キューのダンプ

Eに移ってGを発見.

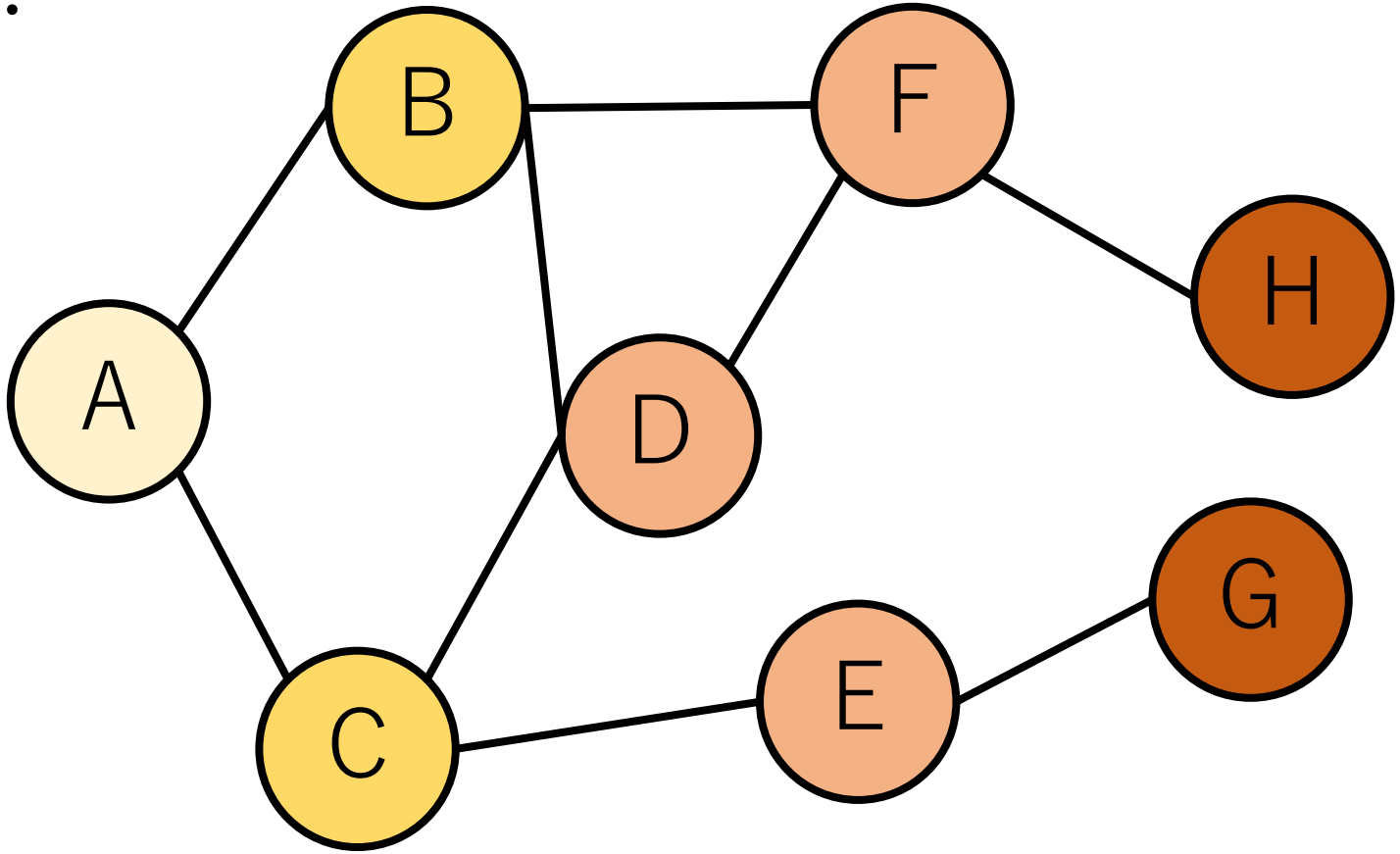
キューから取り出し,
Gを入れる.

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



BFSの例＆キューのダンプ

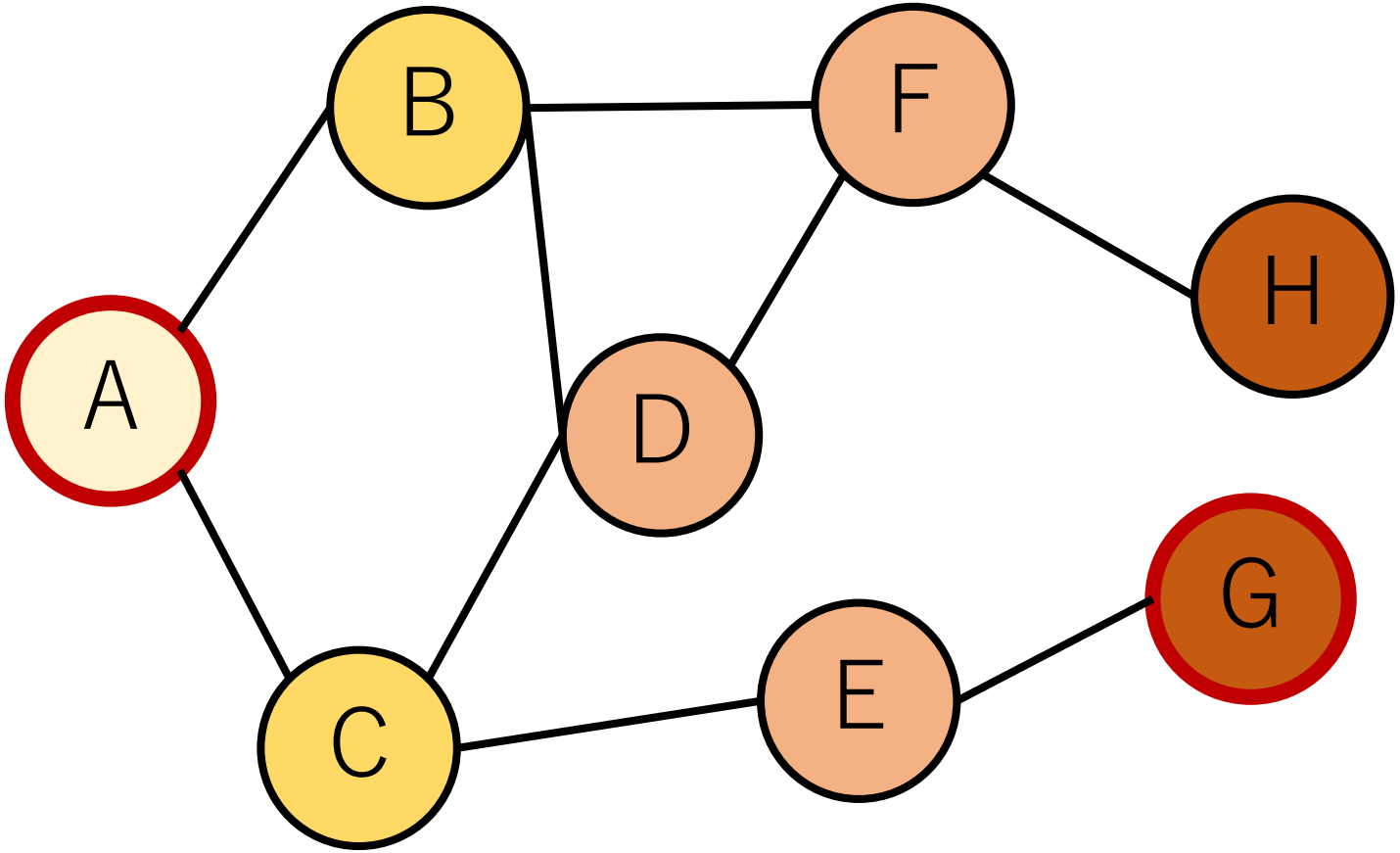
これで全部の探索が完了.



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）

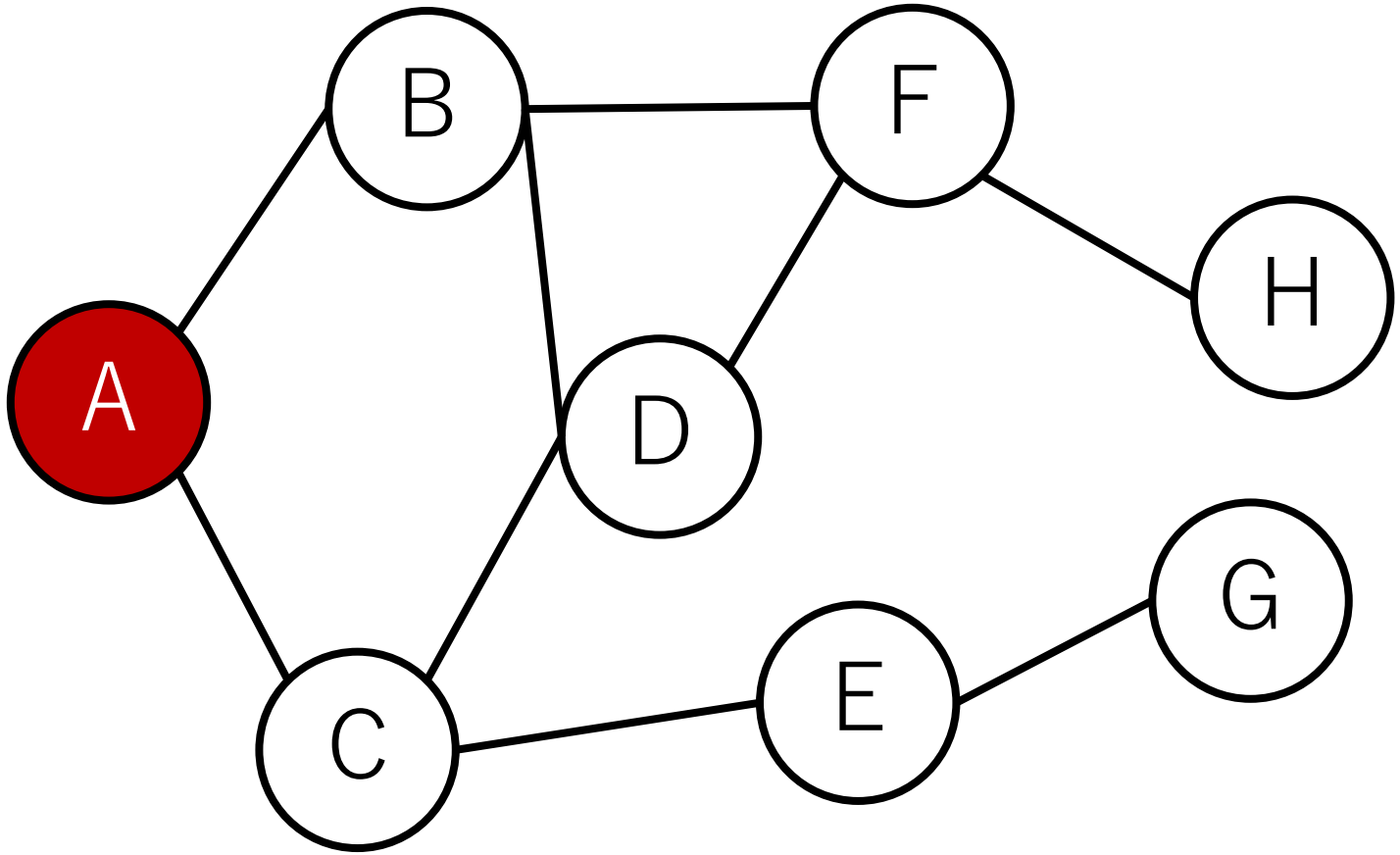
後戻りしないように，可能性のあるルート
全てにおいて1ステップずつ行くパターン.

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.

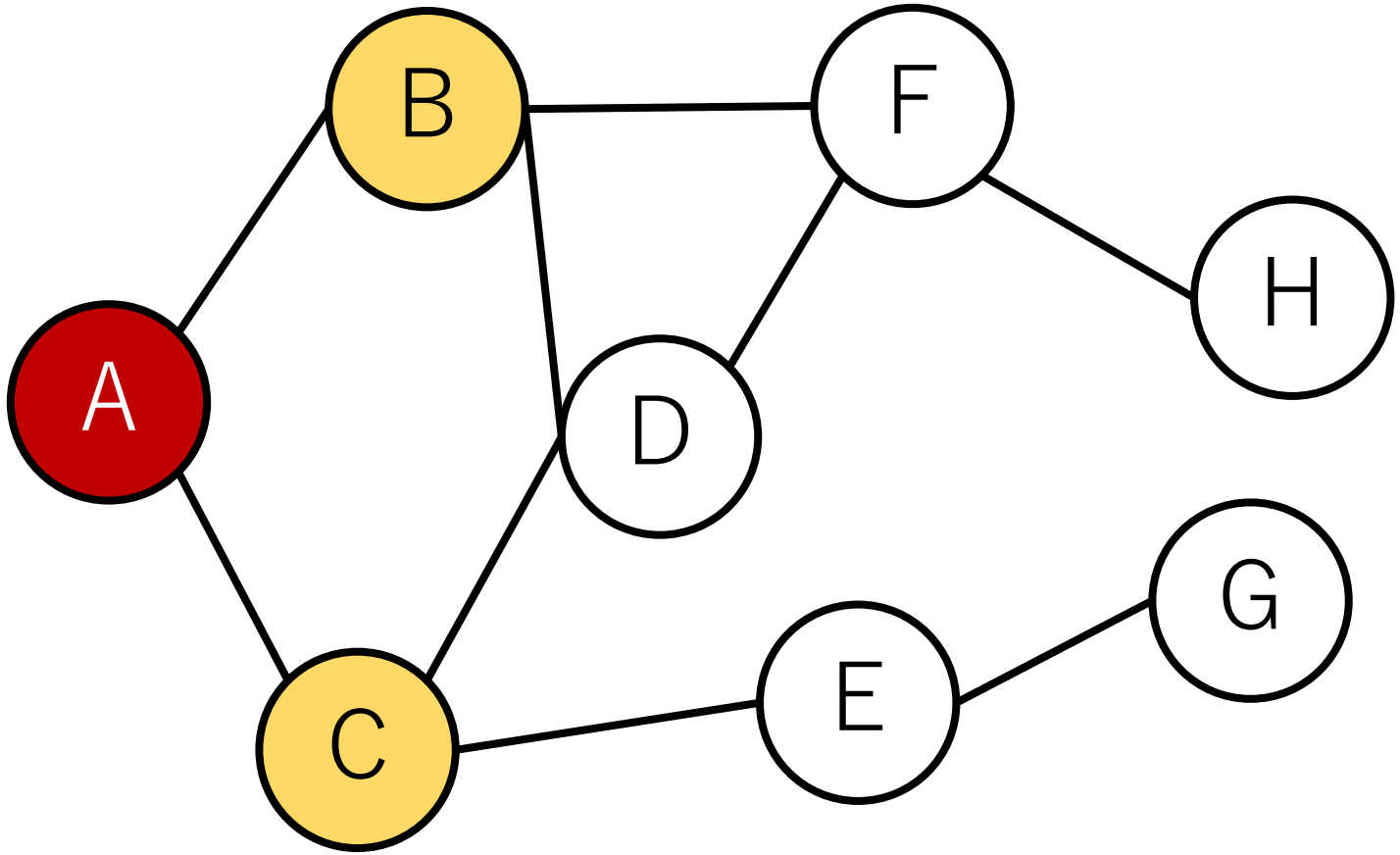
DFSの例

Aからスタート.



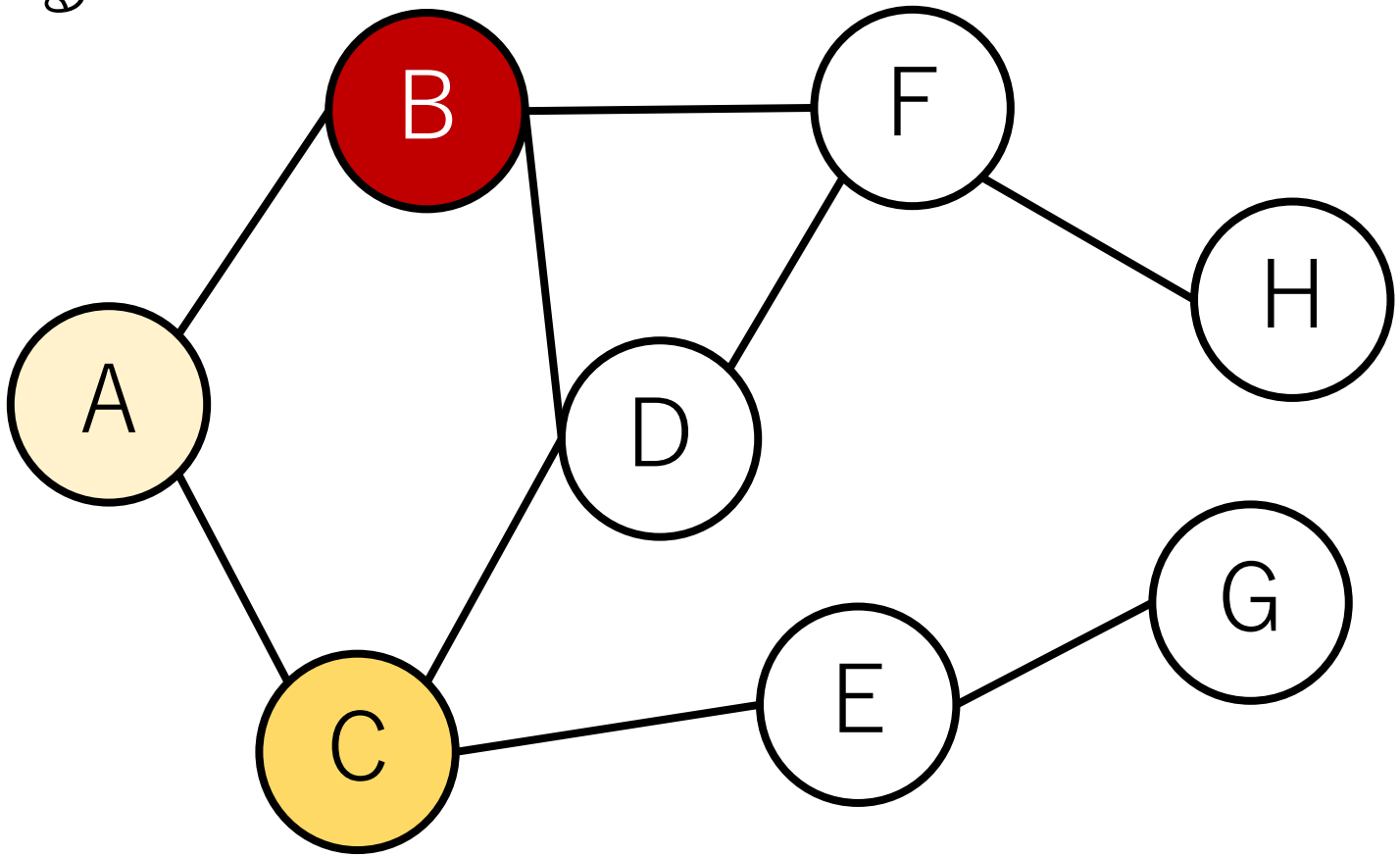
DFSの例

1ステップでつながっているのは，BとC.



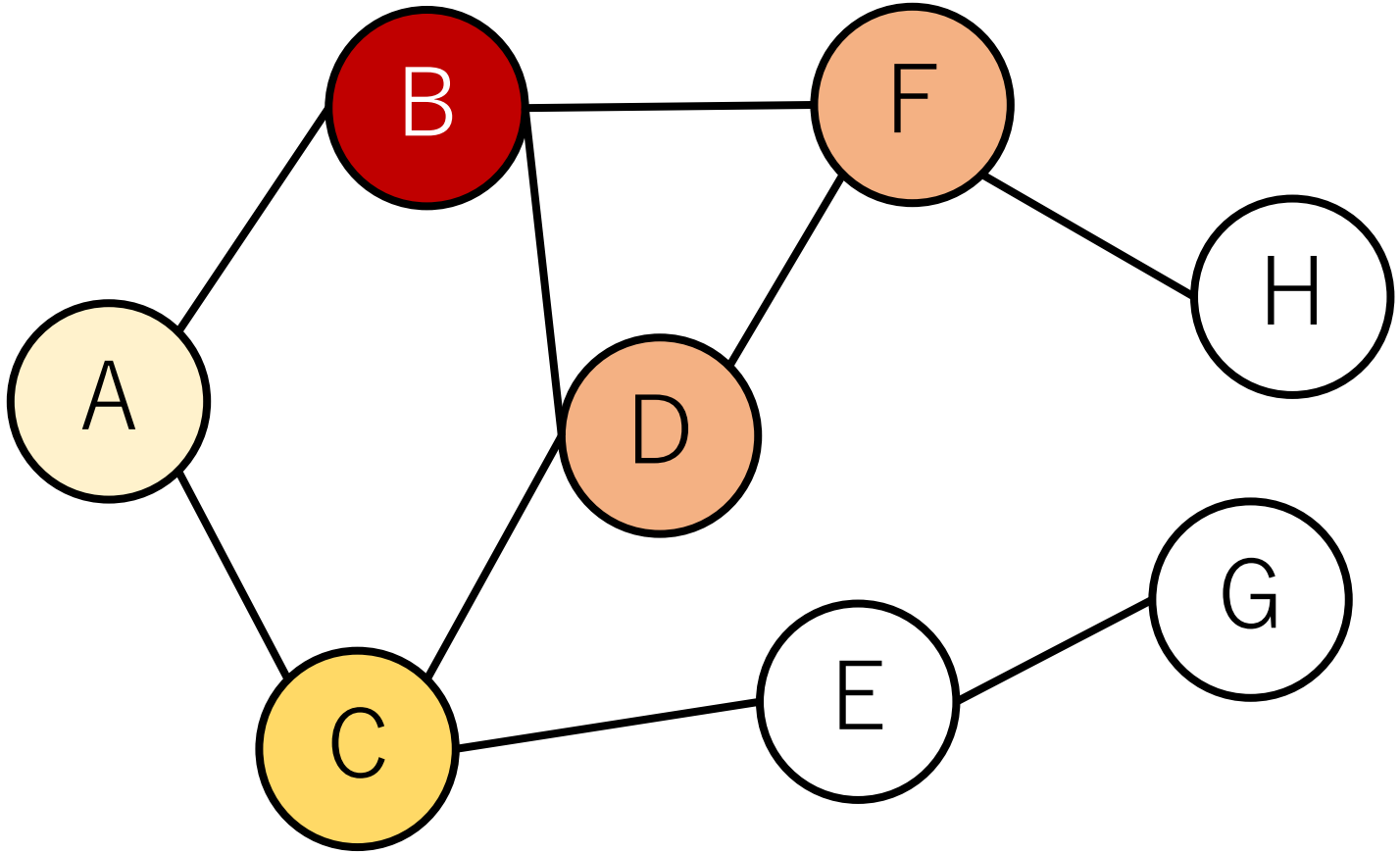
DFSの例

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．）



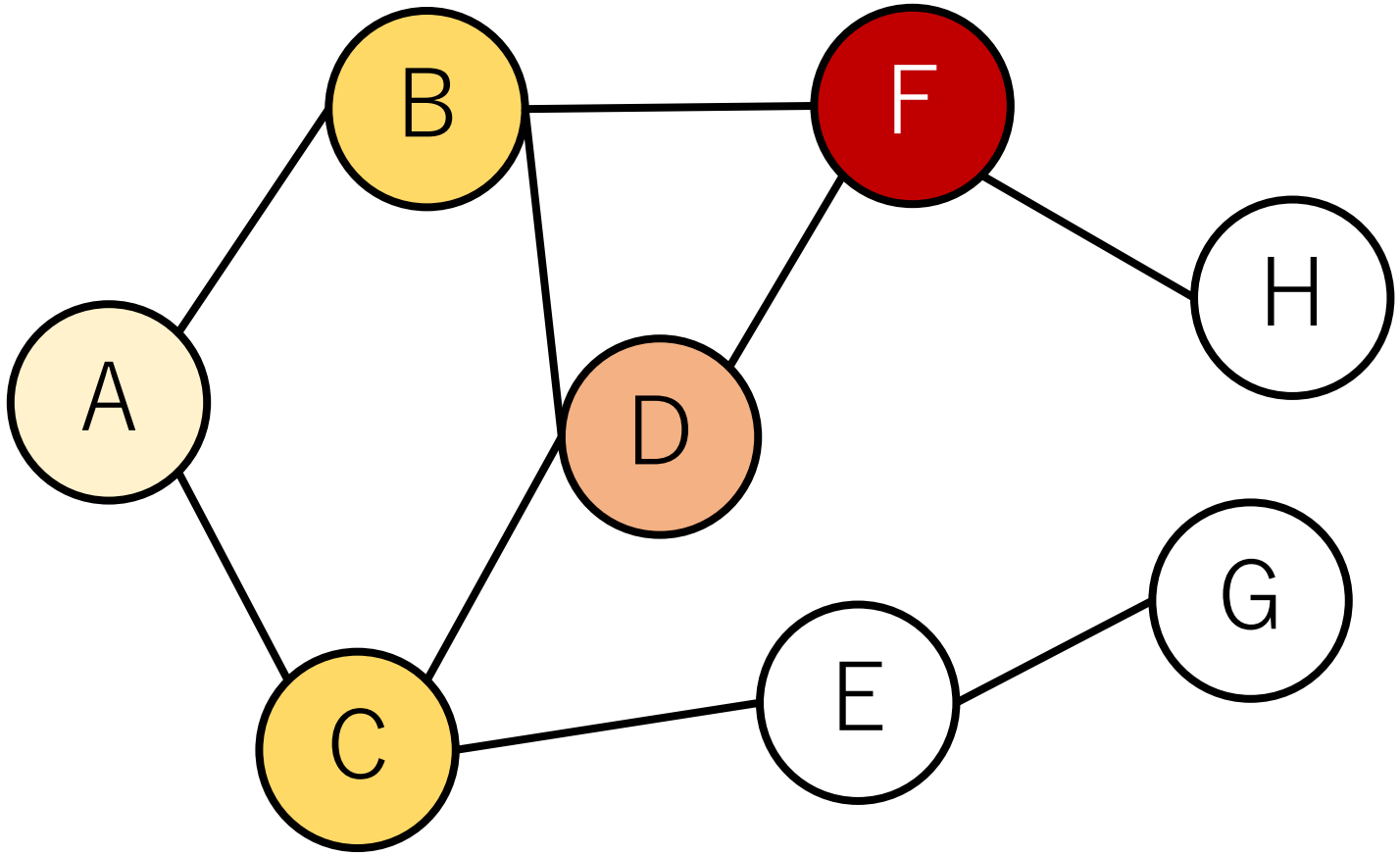
DFSの例

DとFを発見.



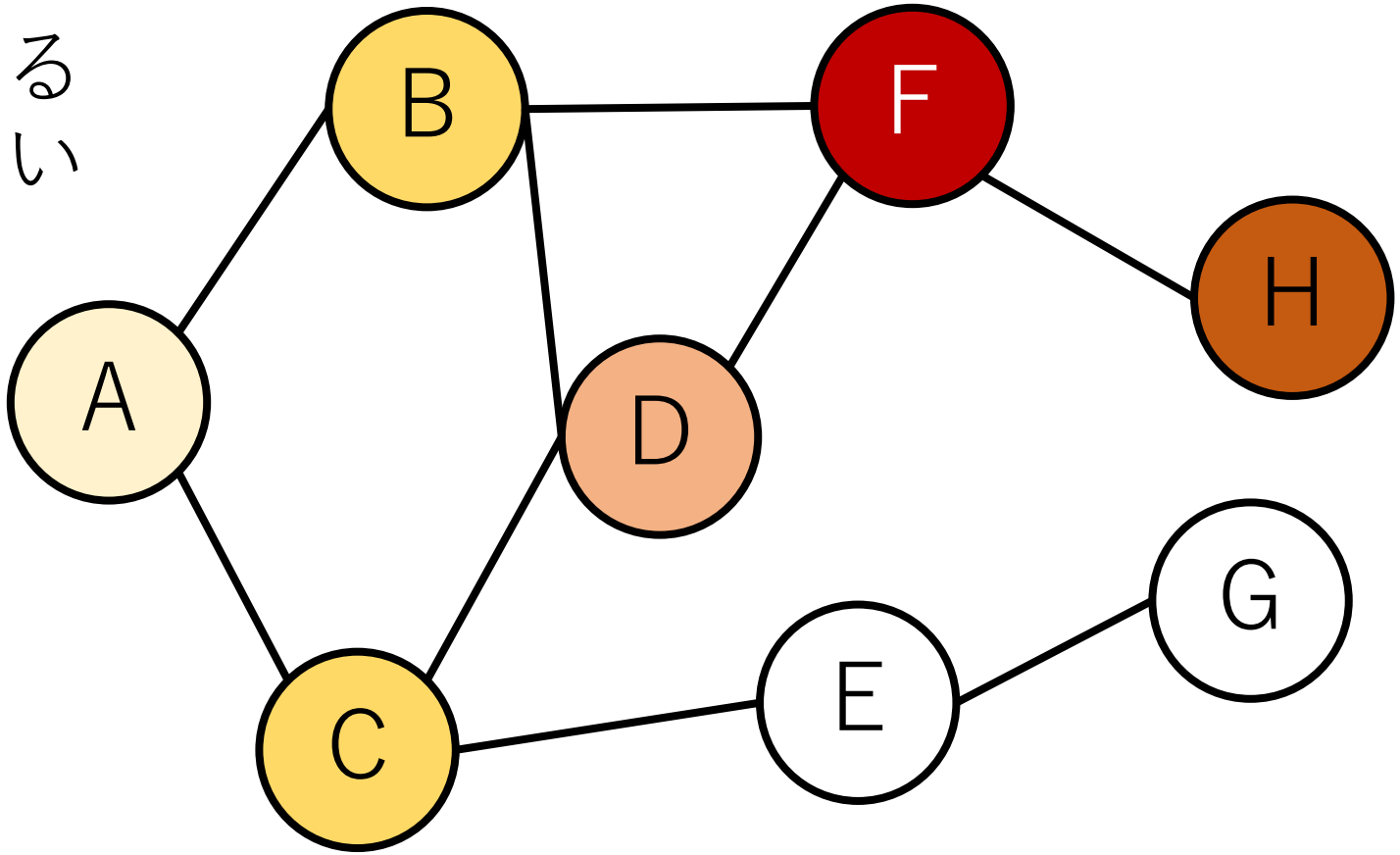
DFSの例

Fに移る.



DFSの例

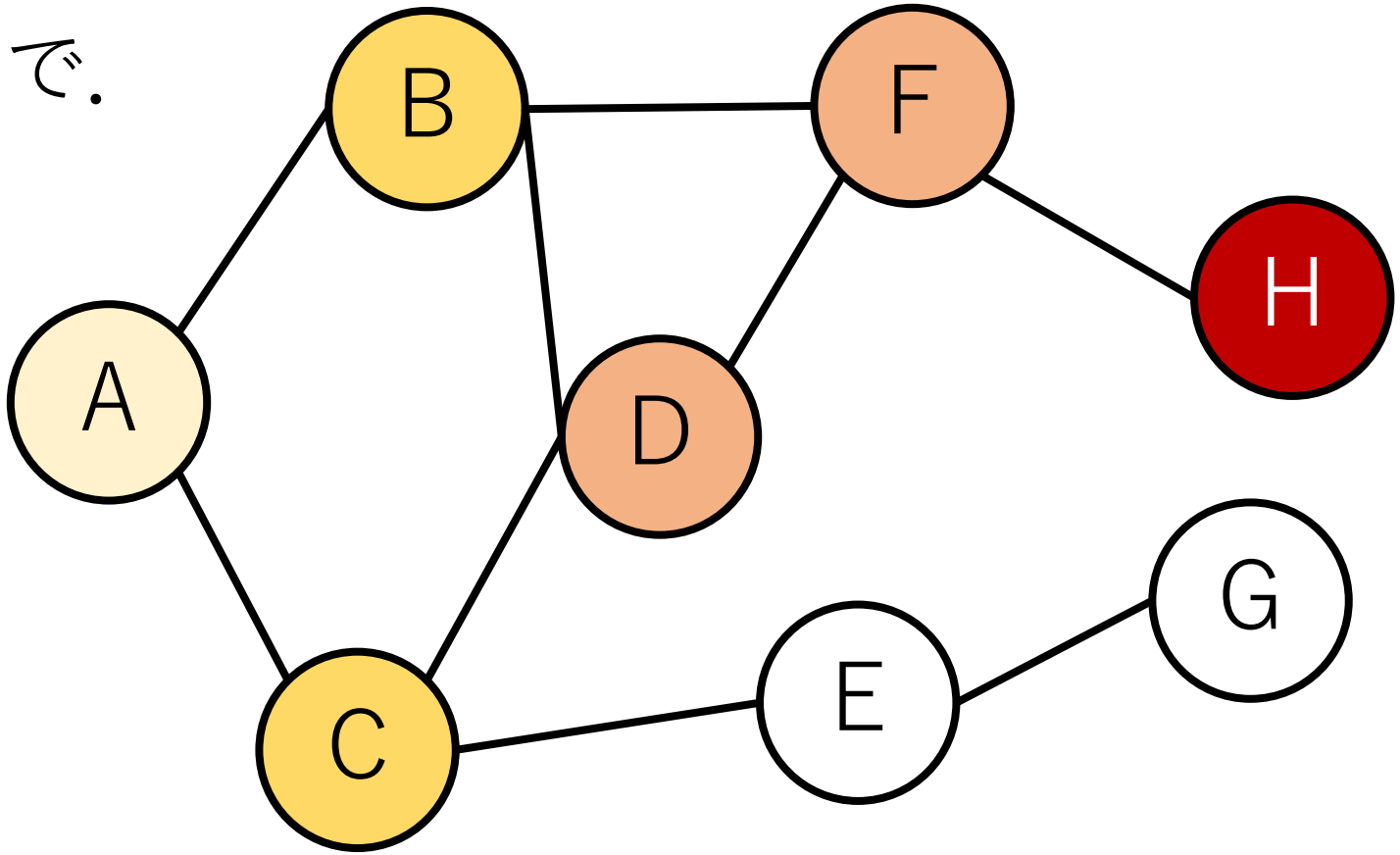
DとHを発見．DはBから見たときに発見はしているが，まだ訪問はしていないので，訪れるべき候補として残す．



（DはBから見たときに発見したので，Fからは移動せず，Bに戻ったときに移動してくる，という実装もある．）

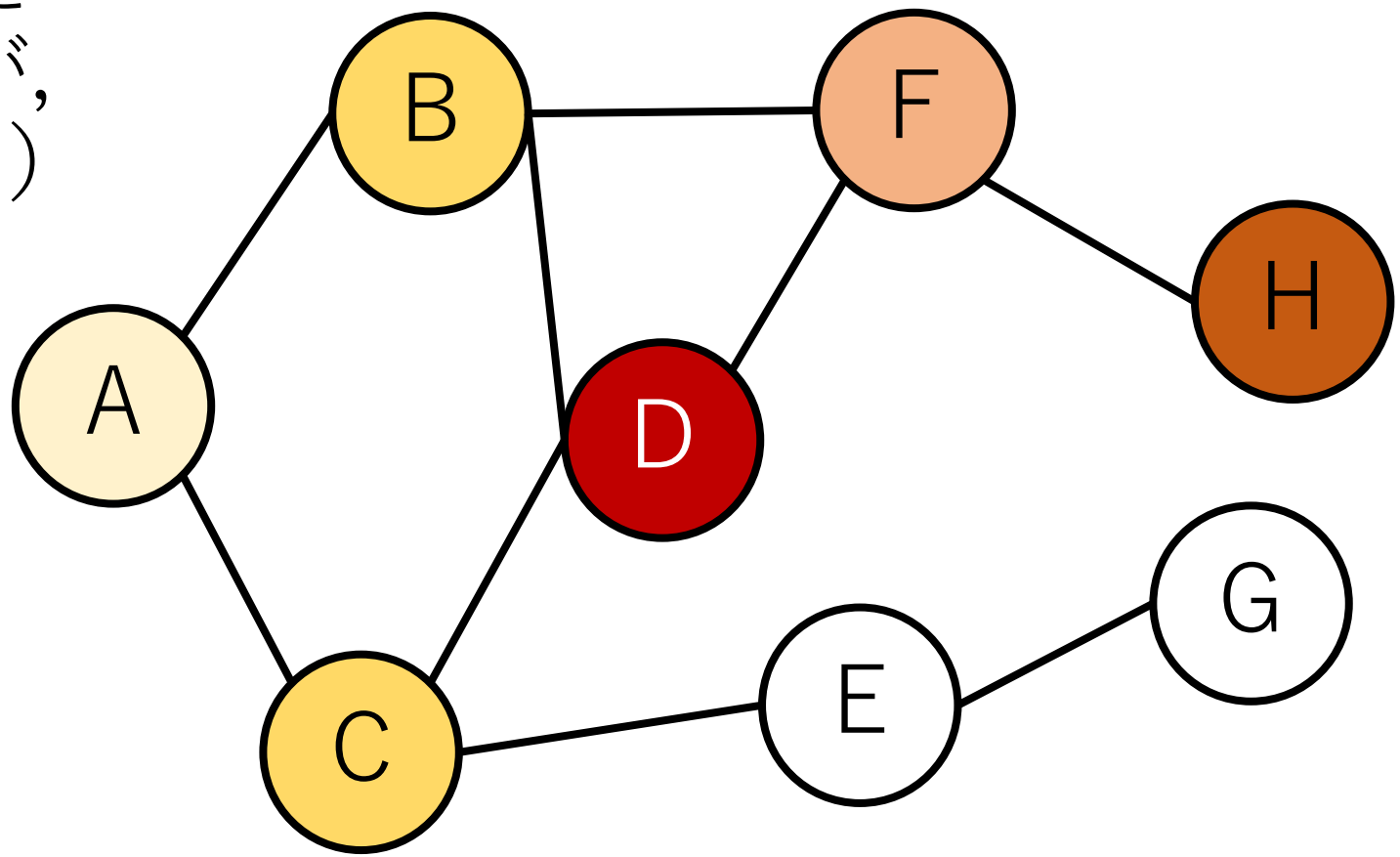
DFSの例

Hに移るが，その先は
存在しないので，ここまで．



DFSの例

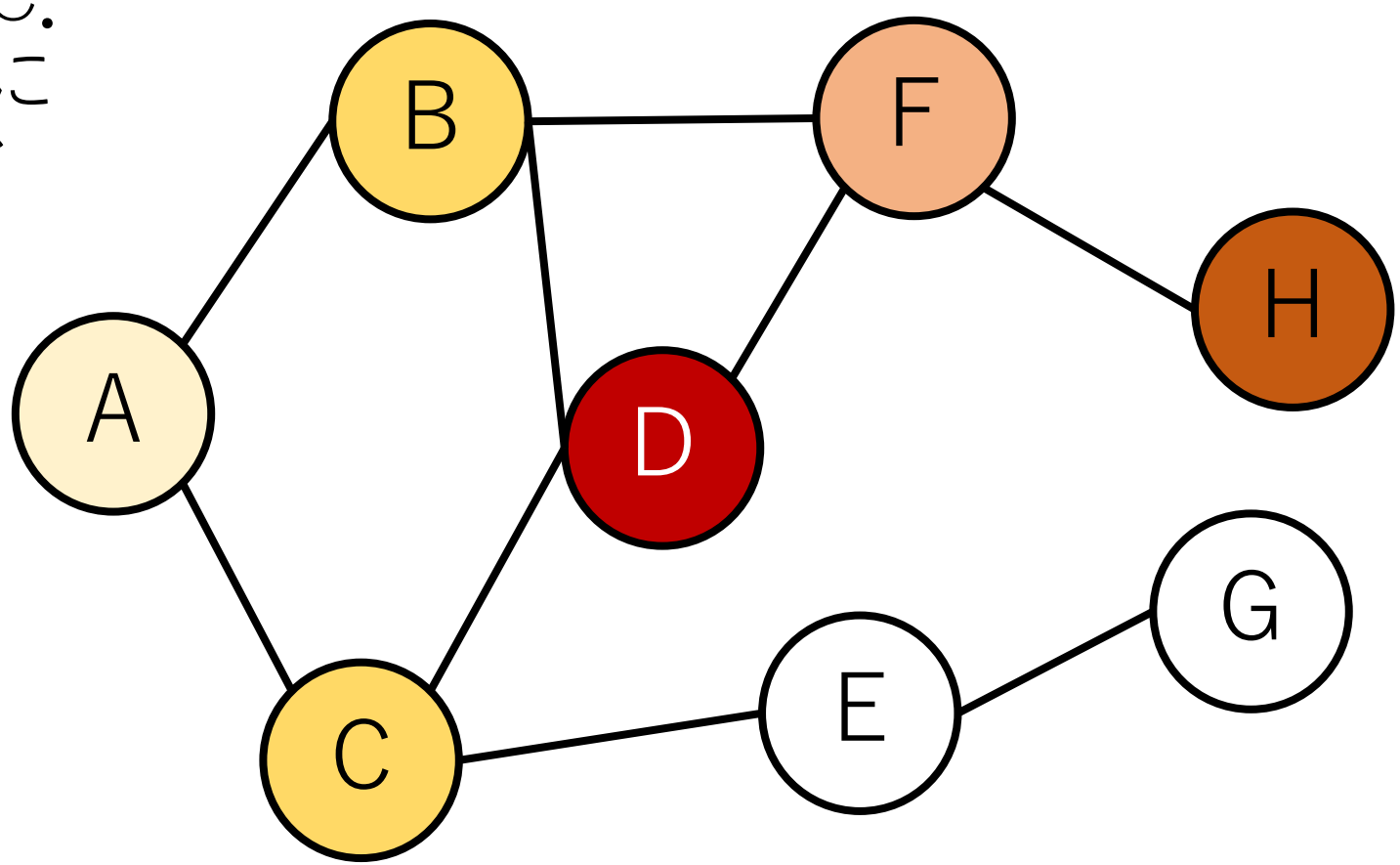
Fまで後戻り，今度はDに移動する．（発見済だが，訪問はまだしていない．）



（DはBから見たときに発見したので，Fからは移動せず，Bに戻ったときに移動してくる，という実装もある．）

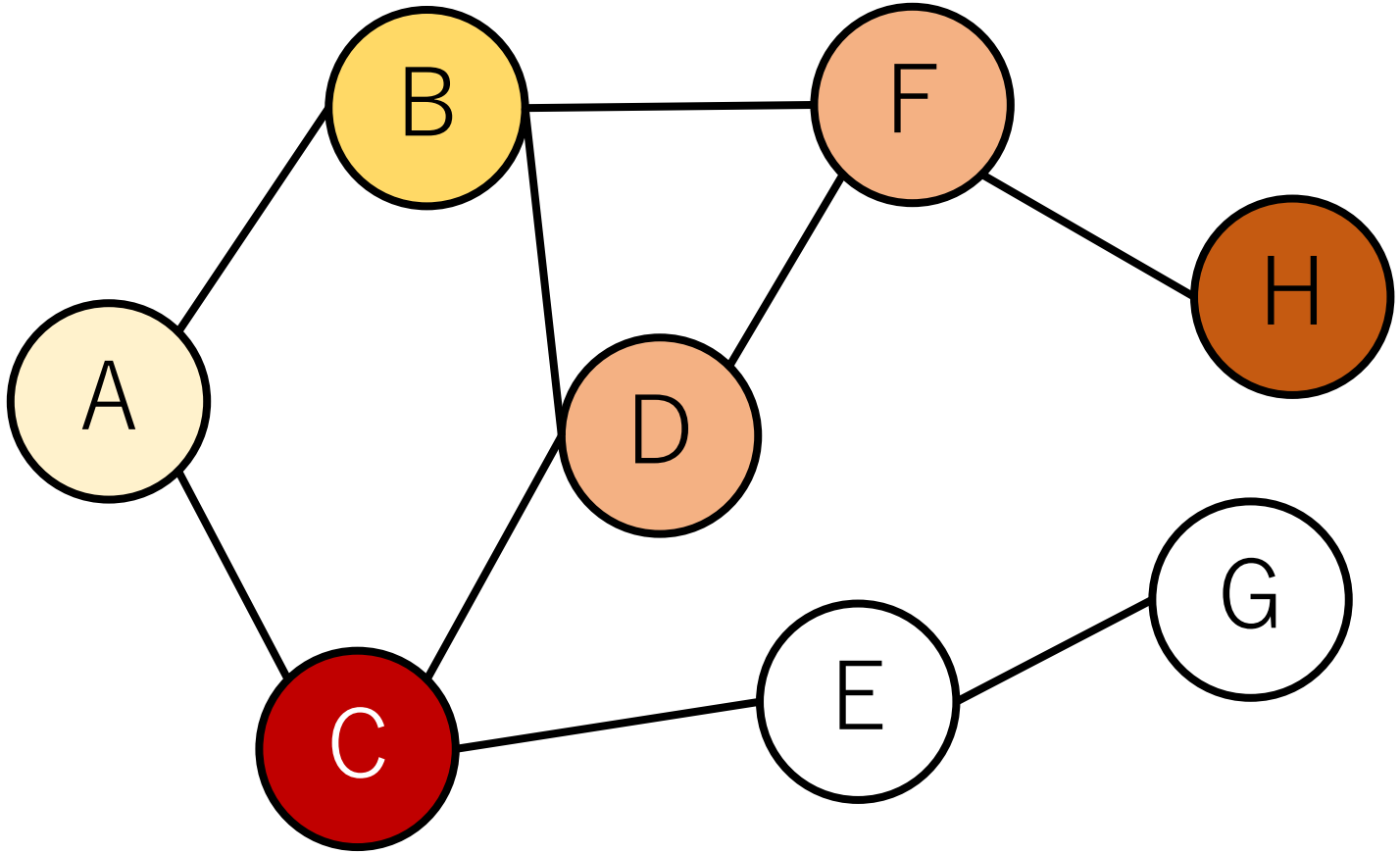
DFSの例

Dから未訪問のノードはC.
（これもAから見たときに発見済だが，訪問はまだしていない．）



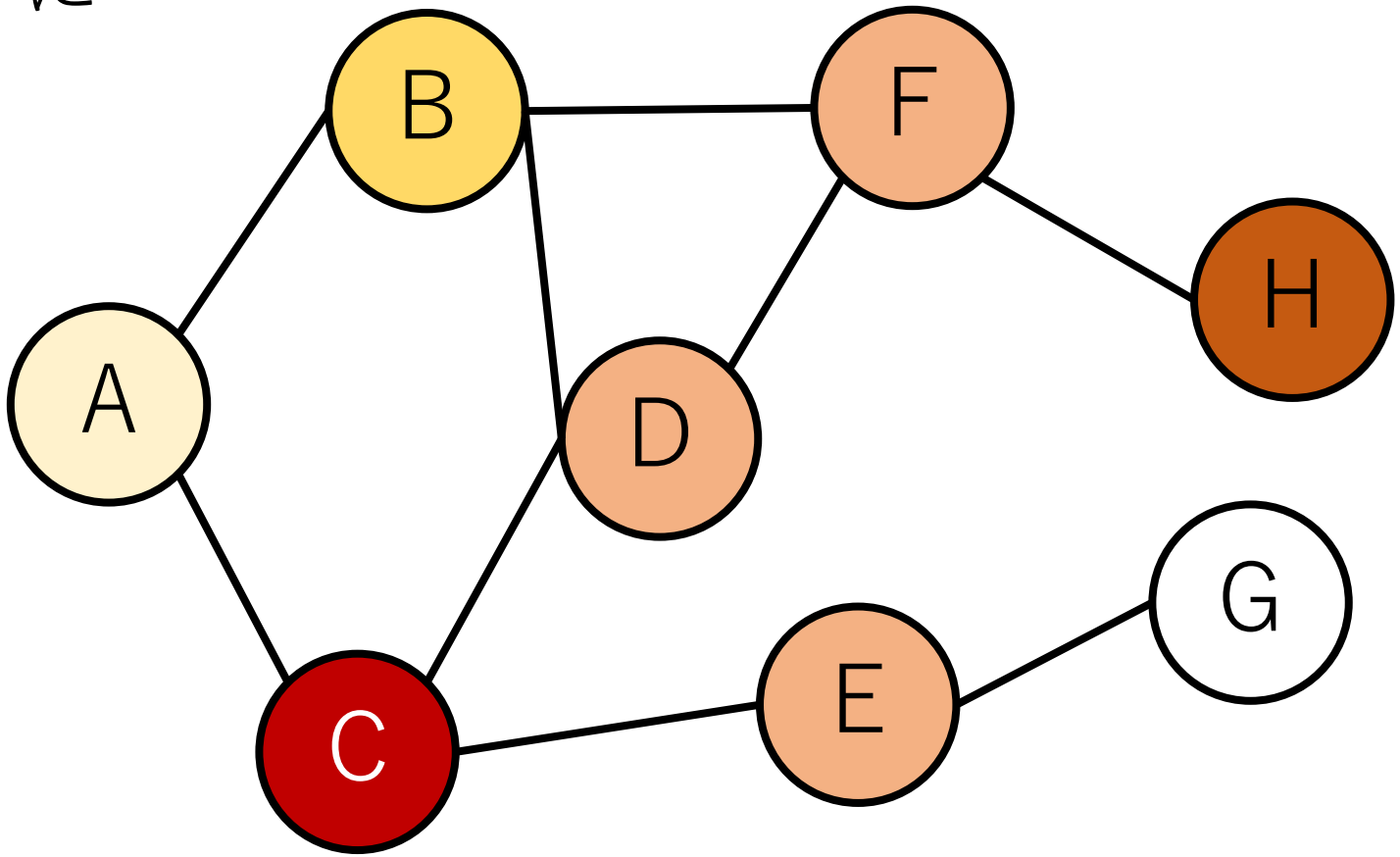
DFSの例

よって，Cに移動する．



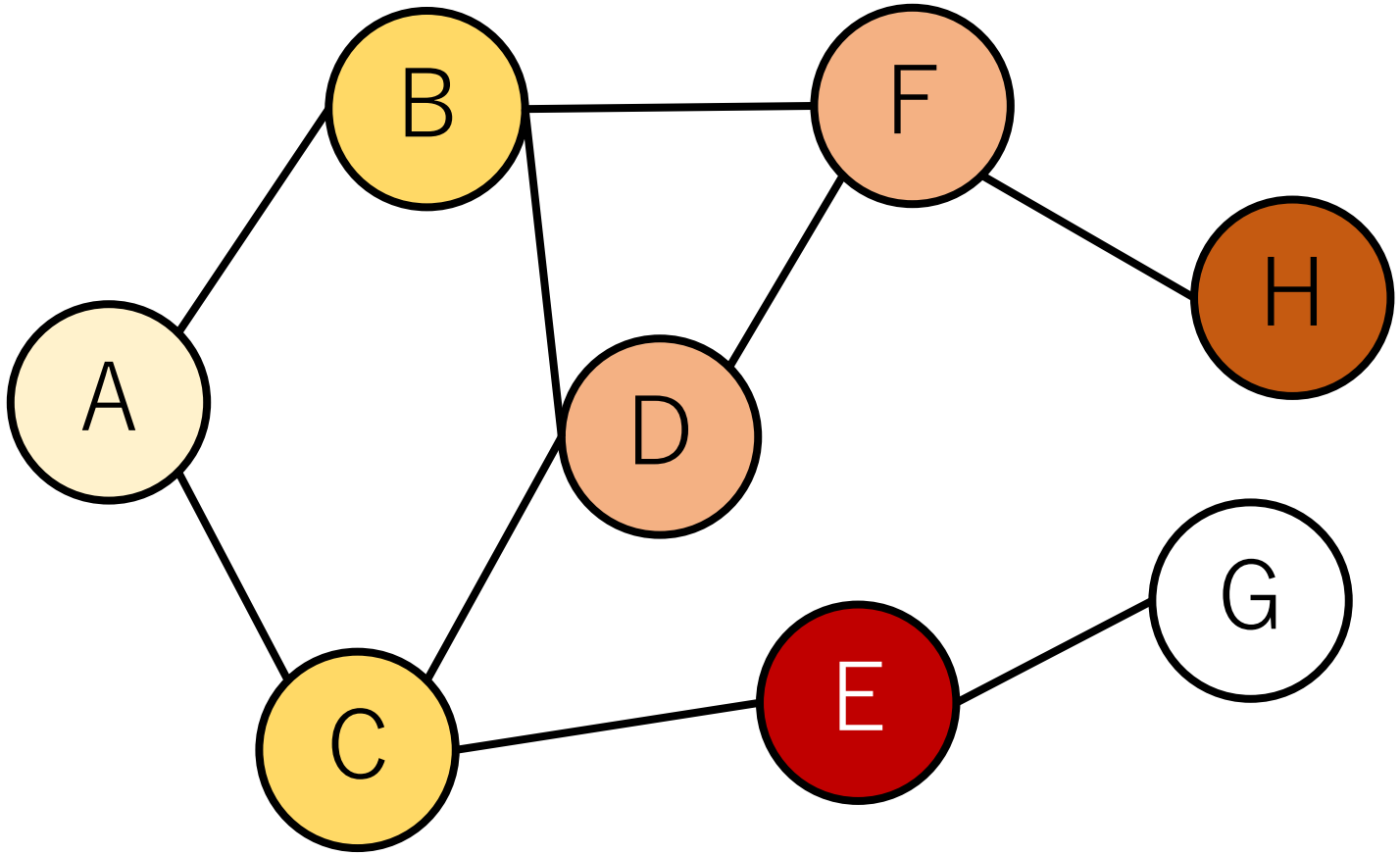
DFSの例

Eを見つける．　(Aはすでに訪問済)



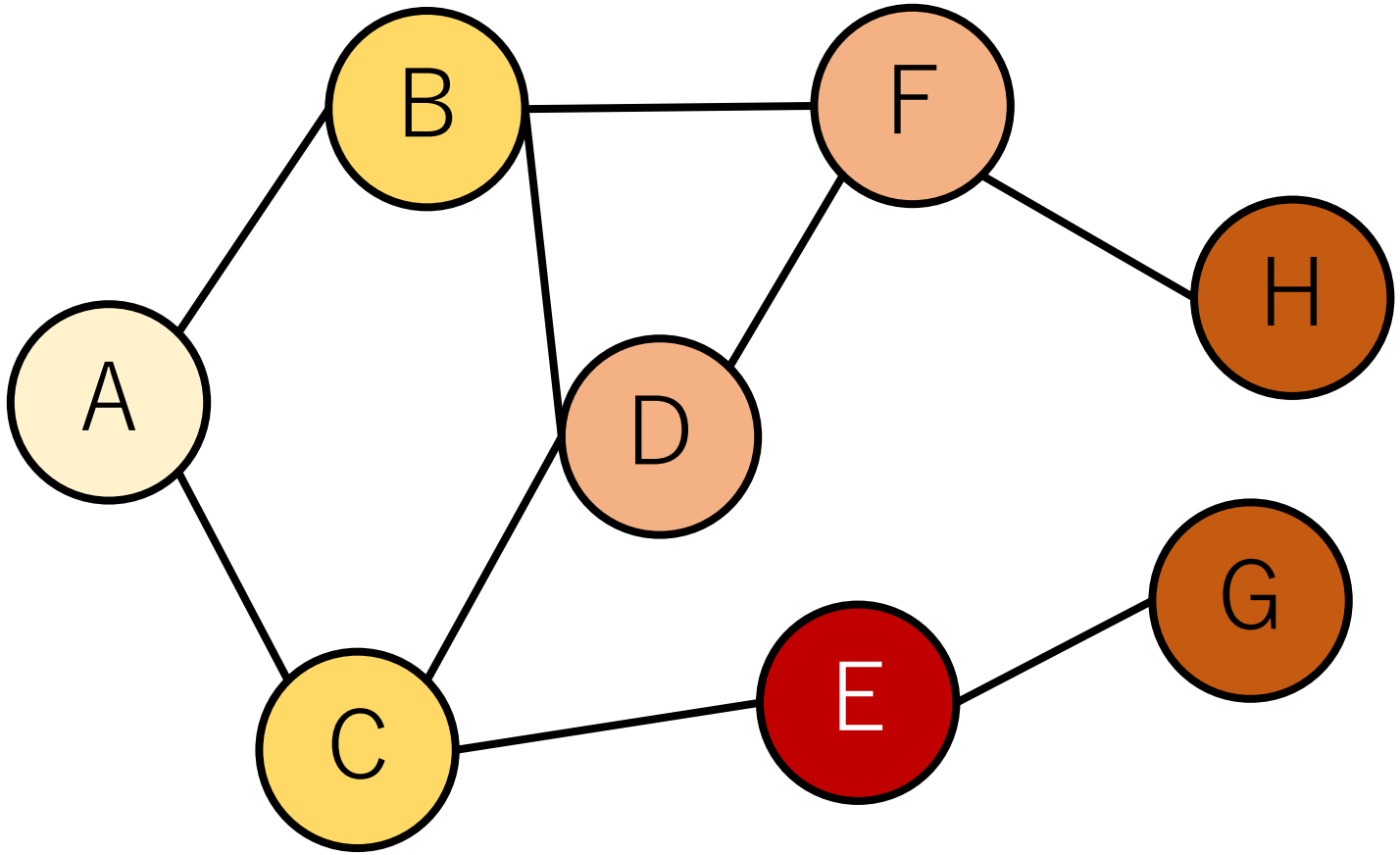
DFSの例

Eに移る.



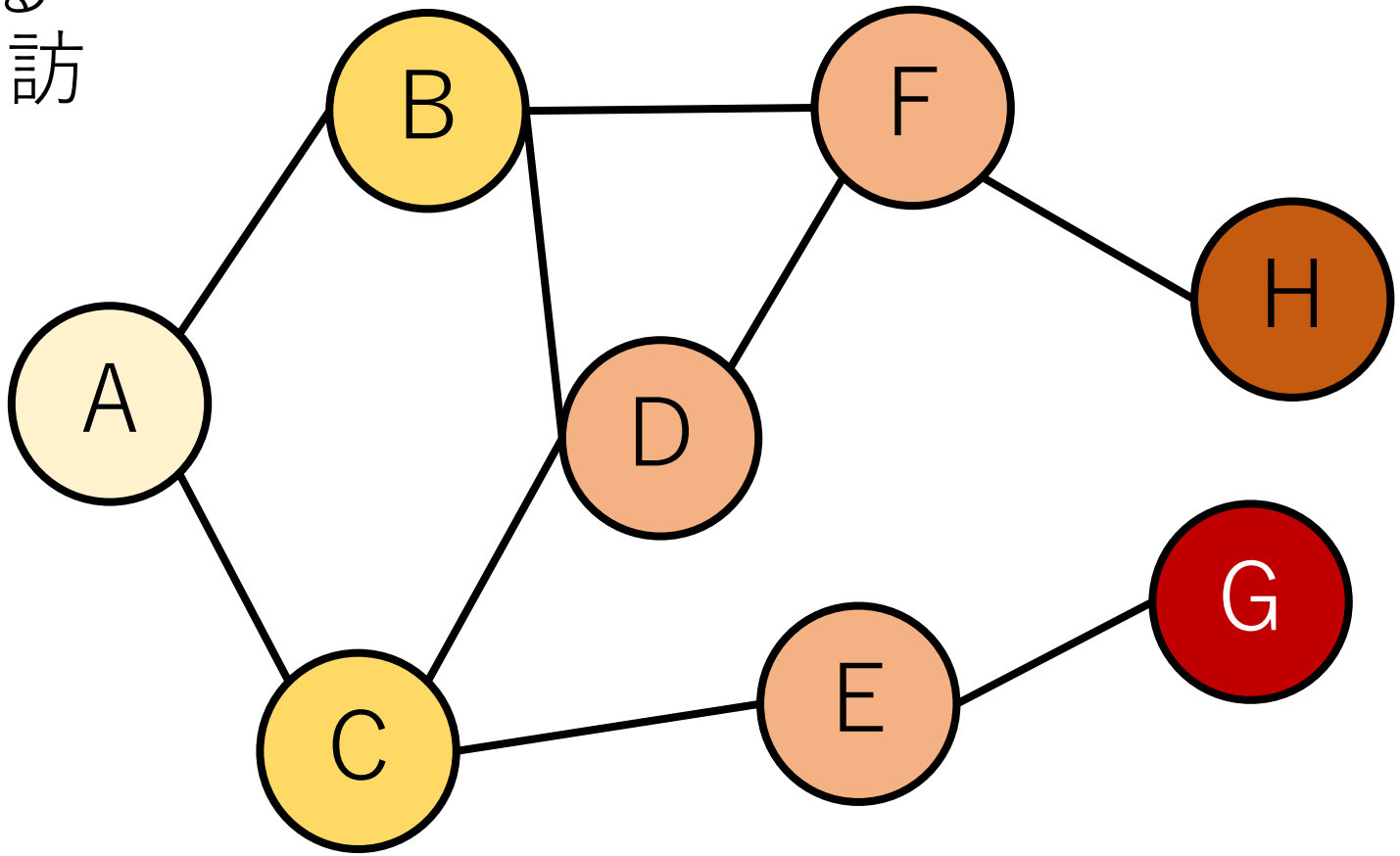
DFSの例

Gをみつける.



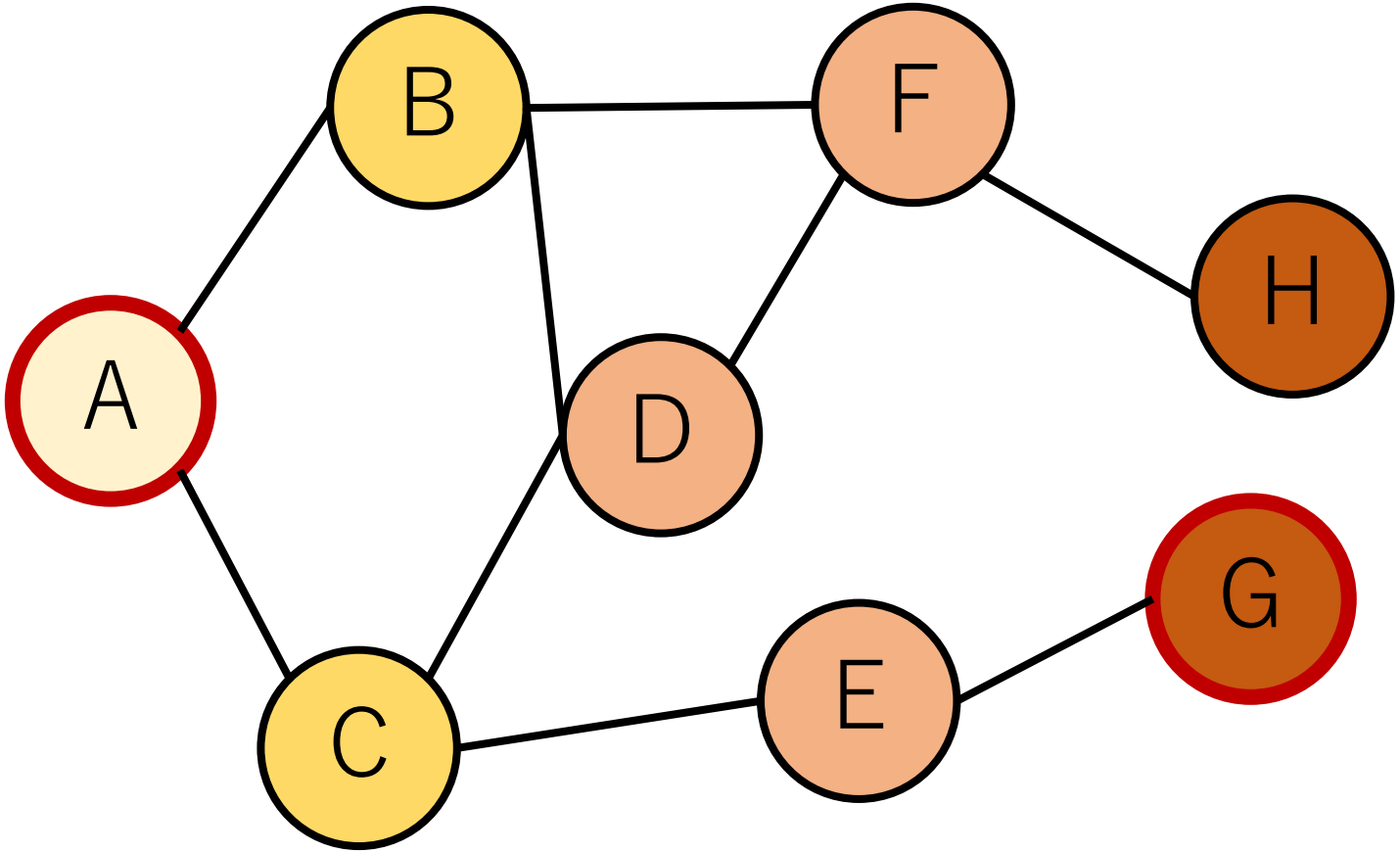
DFSの例

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）

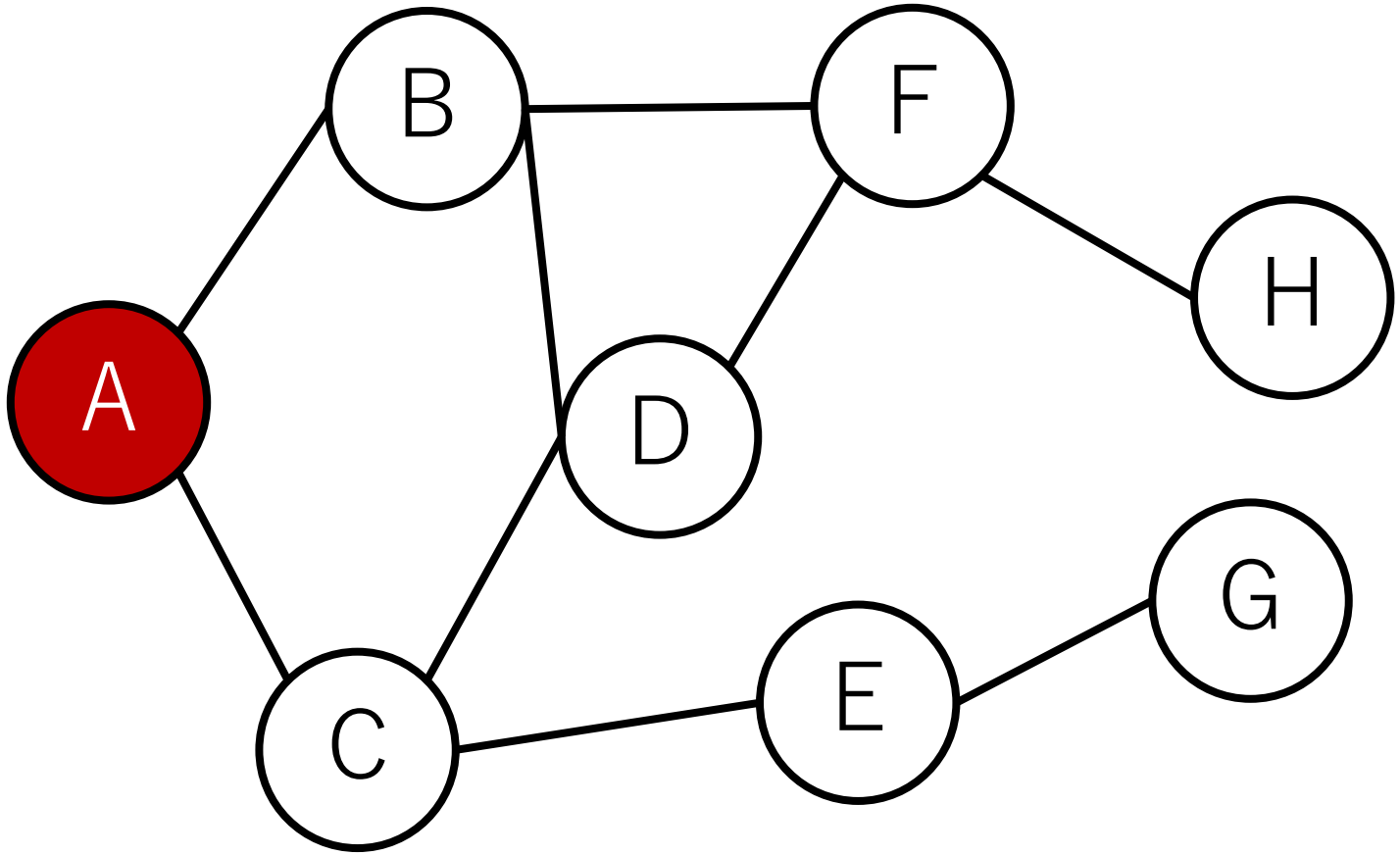
スタックから取り出す：
　　取り出したノードに移動する.

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.

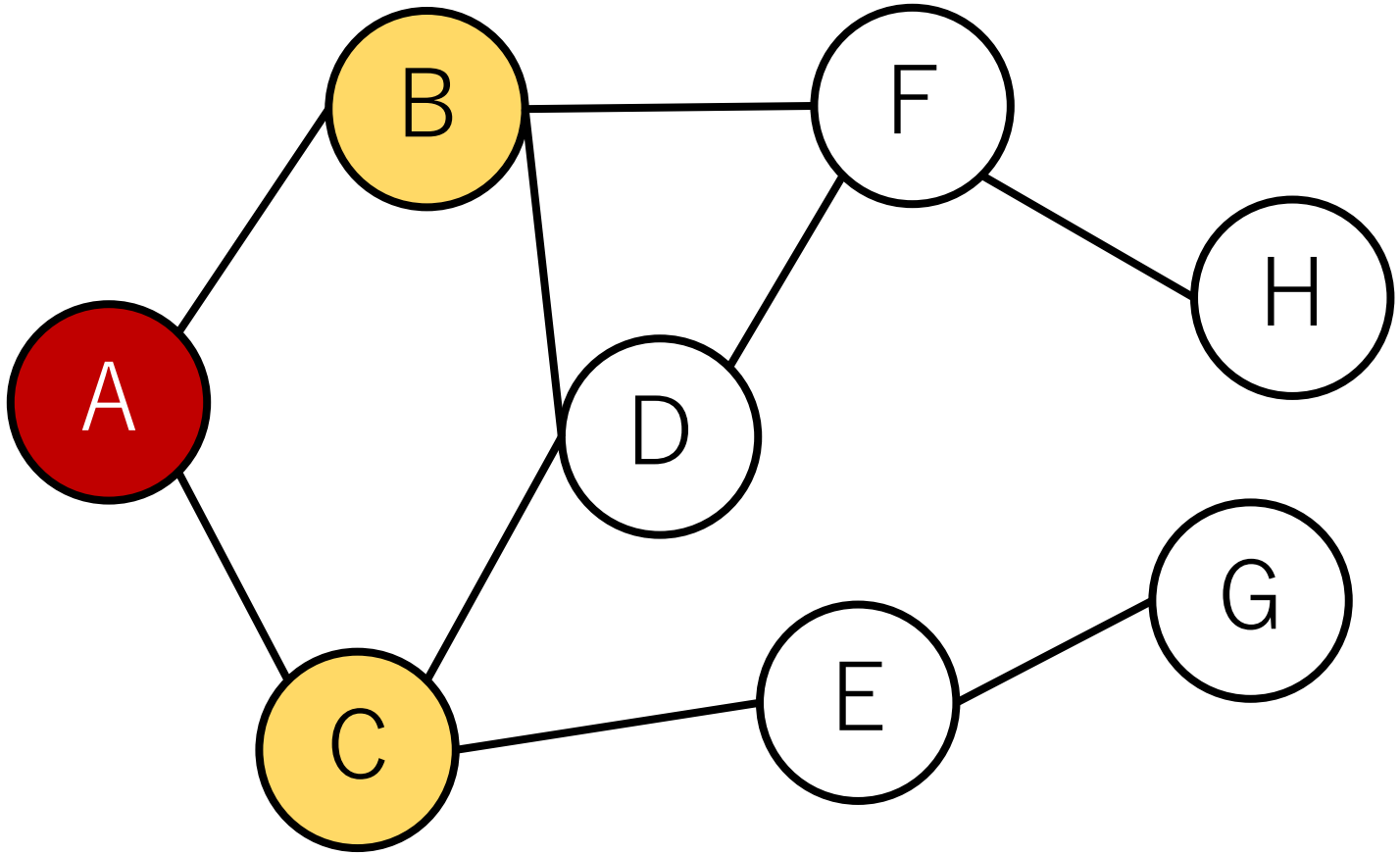


A							

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.

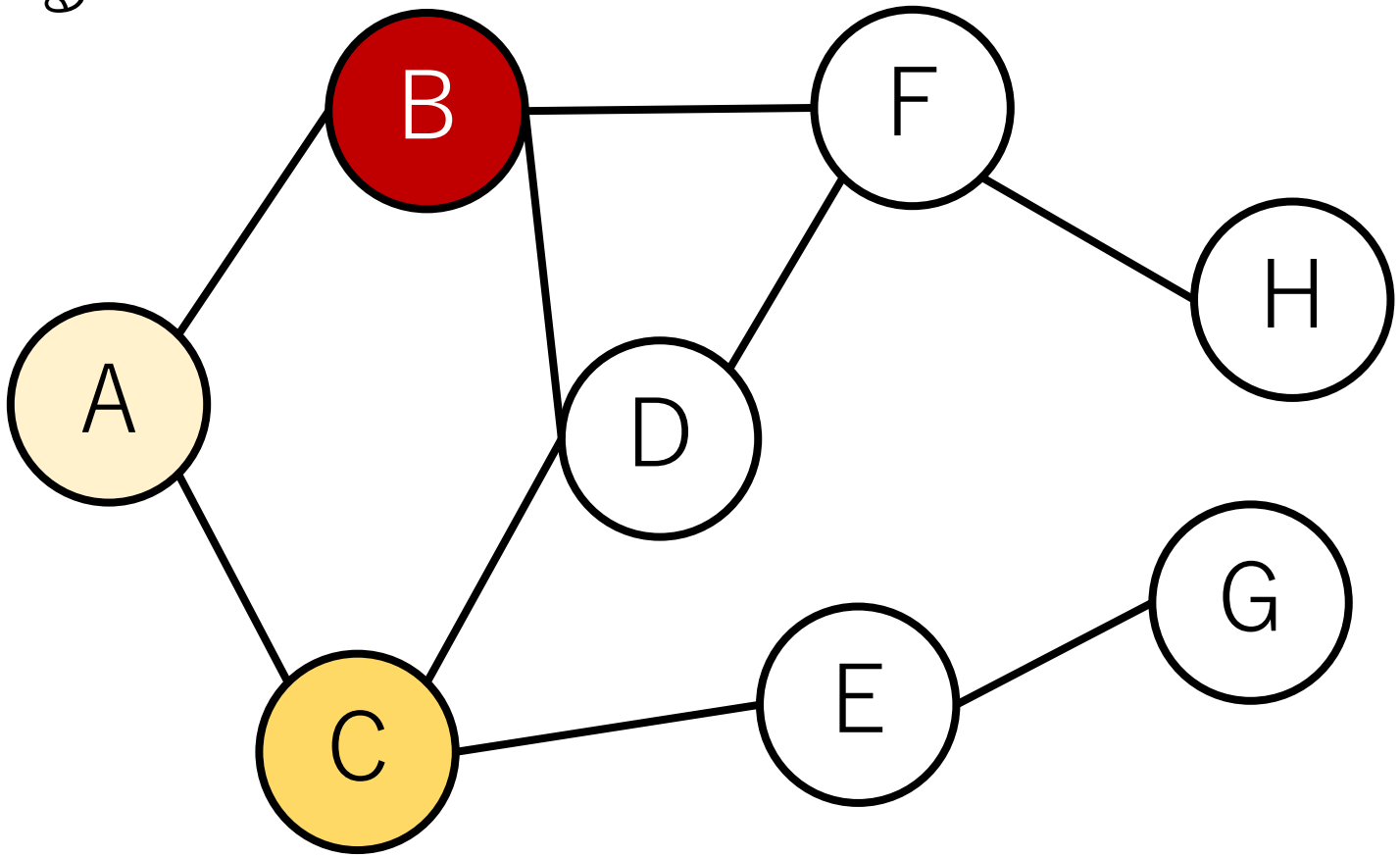
BとCをスタックに入れる.



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．）

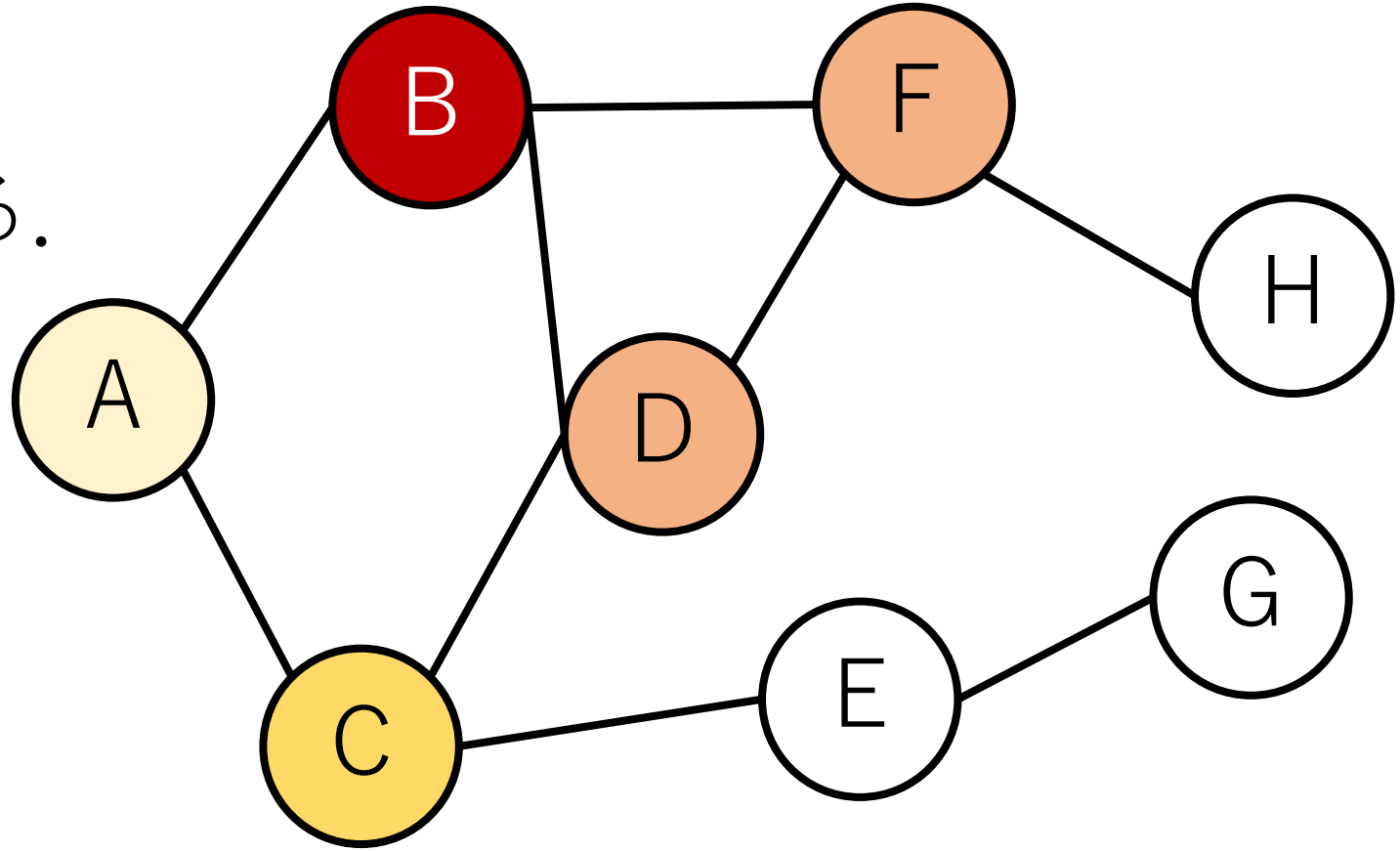
スタックから取り出す．



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.



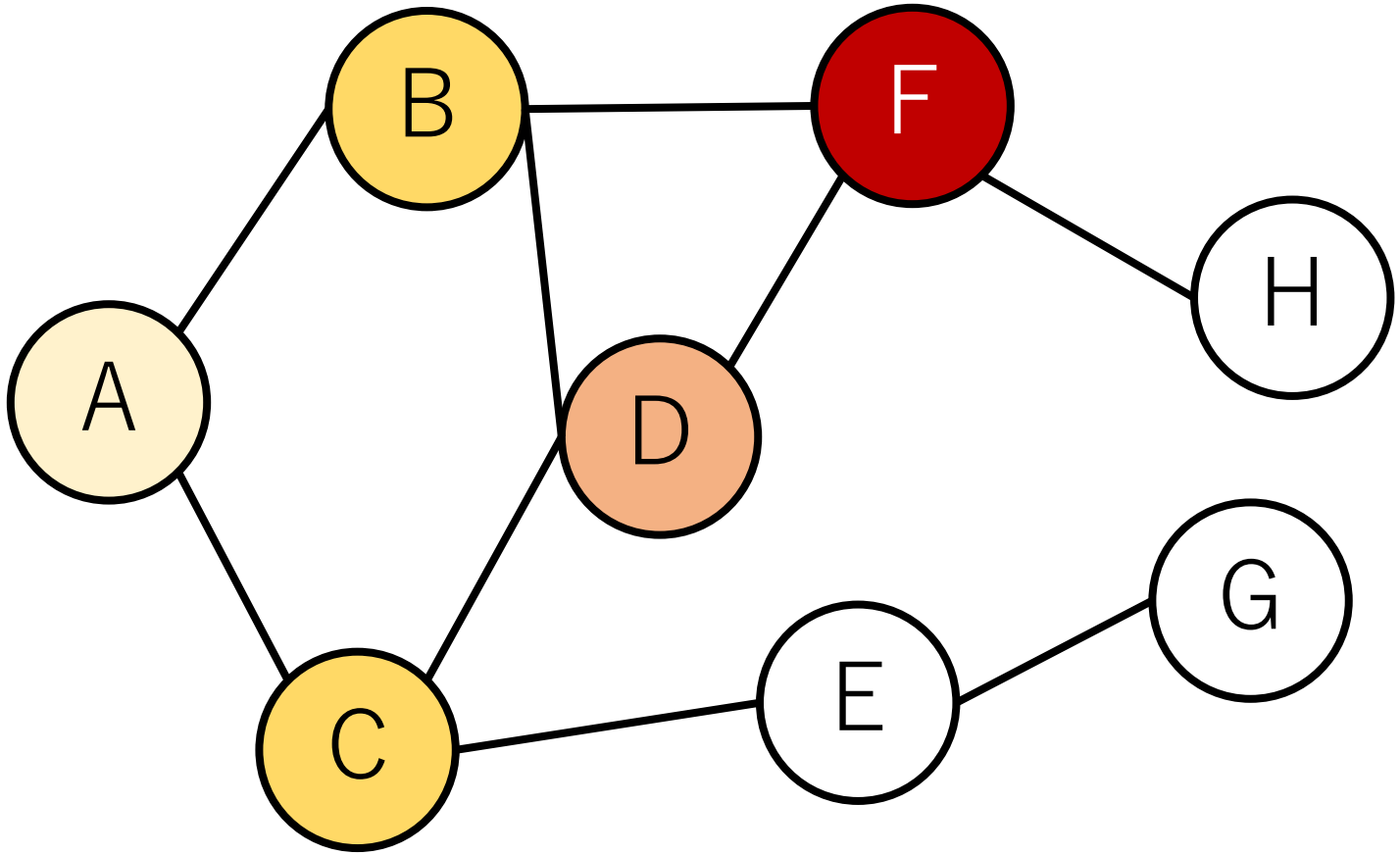
C	D	F					

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.

C	D	**F**					

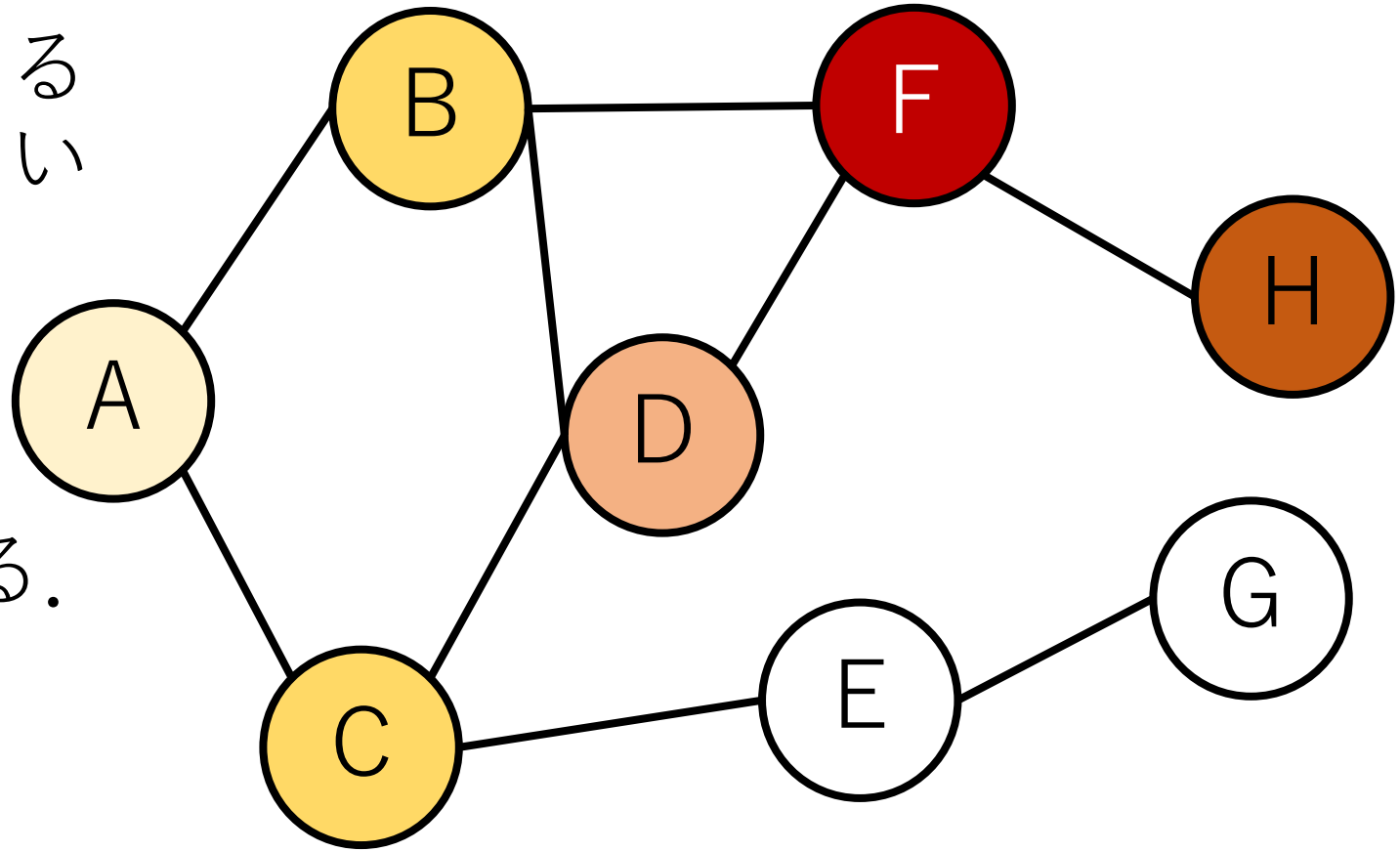


DFSの例＆スタックのダンプ

DとHを発見．DはBから見たときに発見はしているが，まだ訪問はしていないので，訪れるべき候補として残す．

DとHをスタックに入れる．

C	D	D	H				



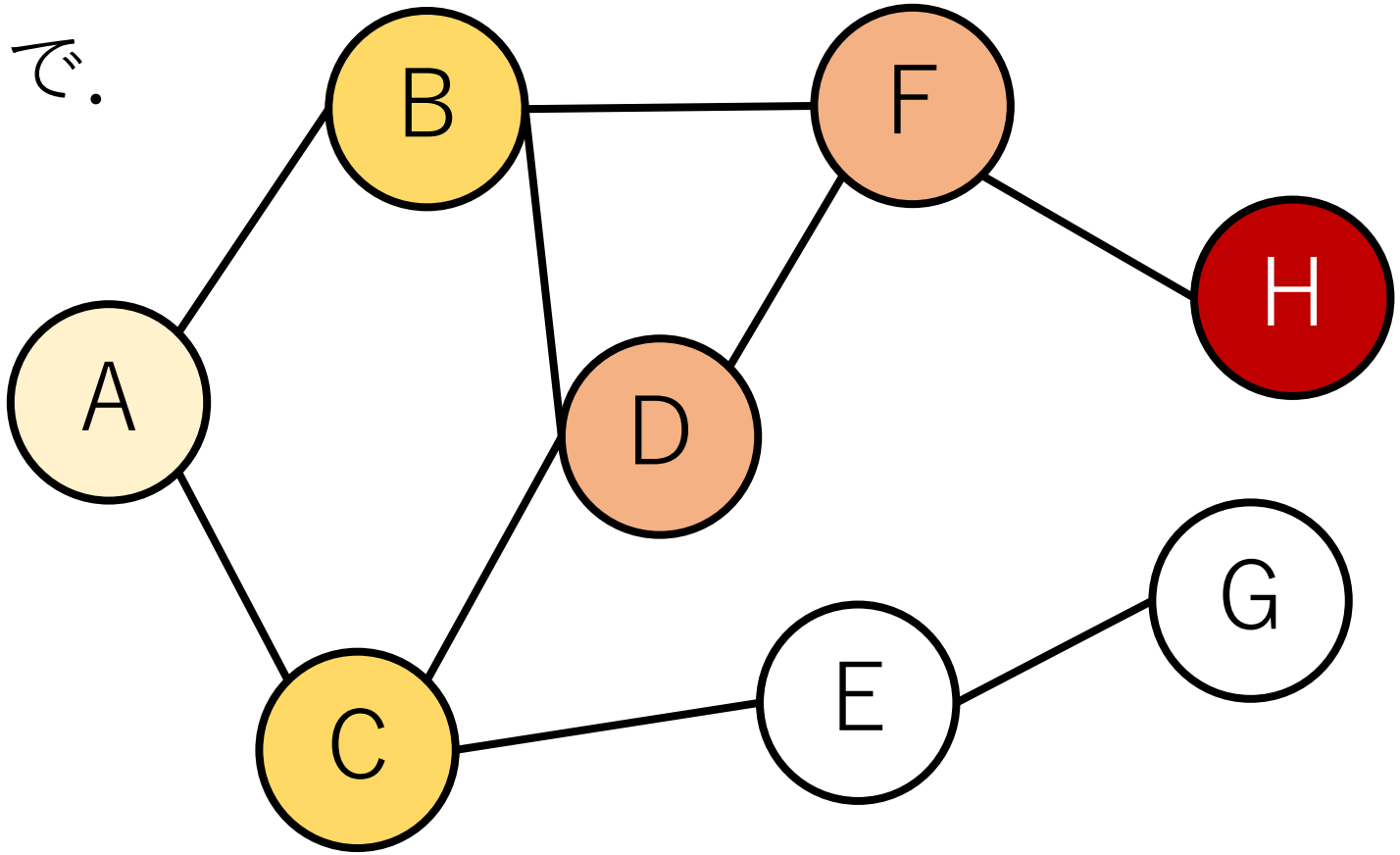
（発見済の場合にはスタックに入れない，という実装もある．）

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.

C	D	D	**H**				

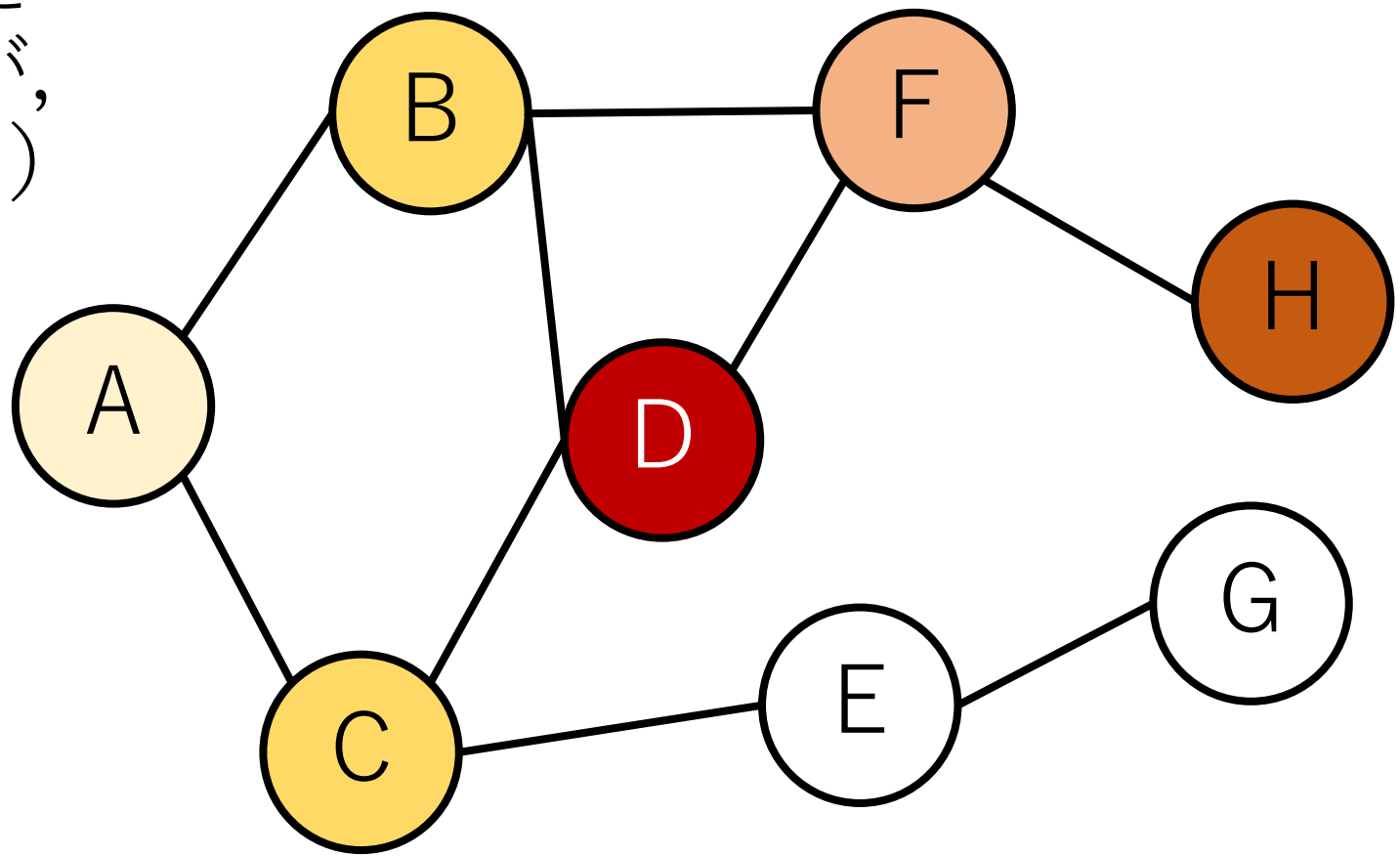


DFSの例

F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	**D**					

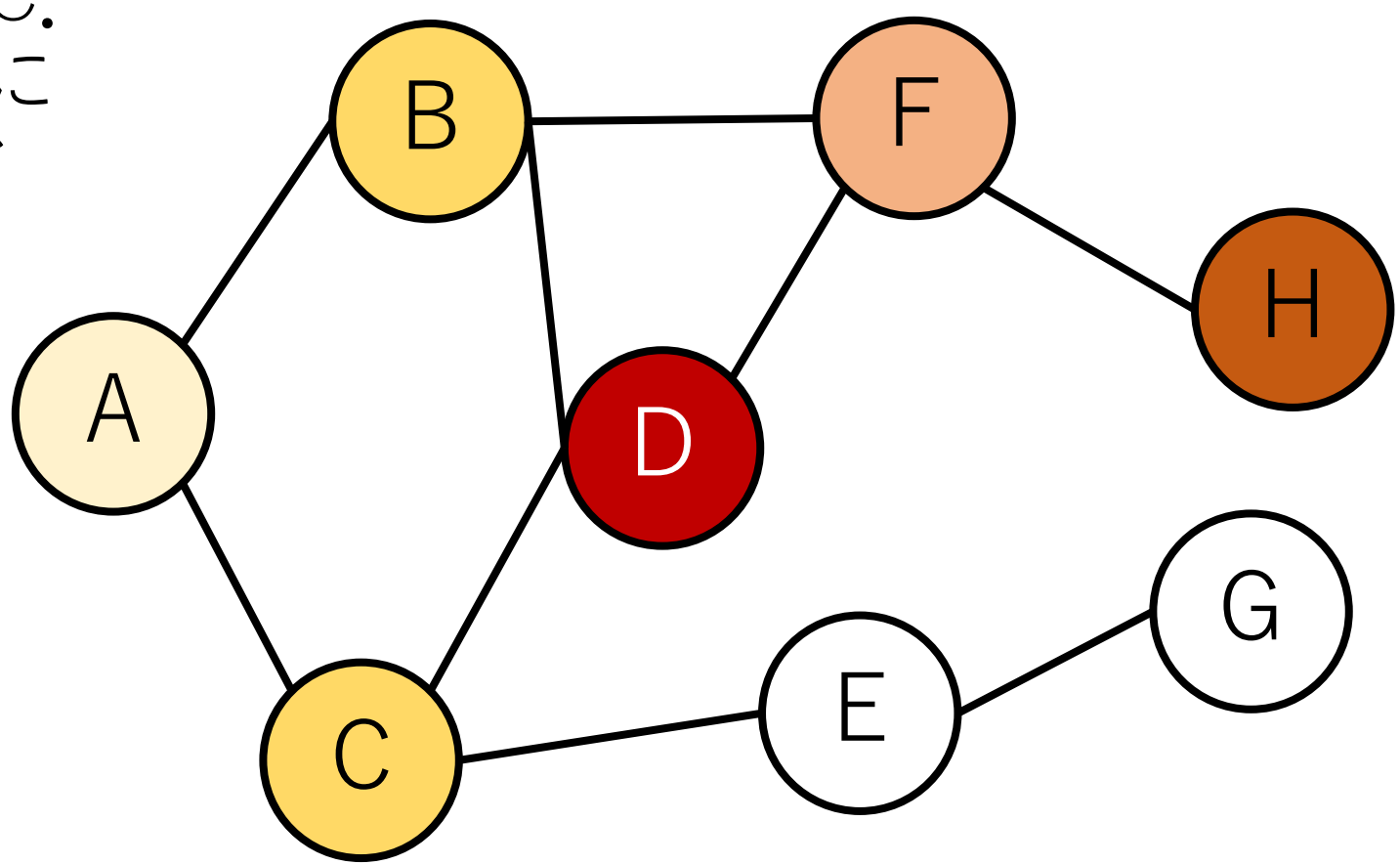


DFSの例

Dから未訪問のノードはC.
（これもAから見たときに発見済だが，訪問はまだしていない．）

Cをスタックに入れる．

C	D	C					

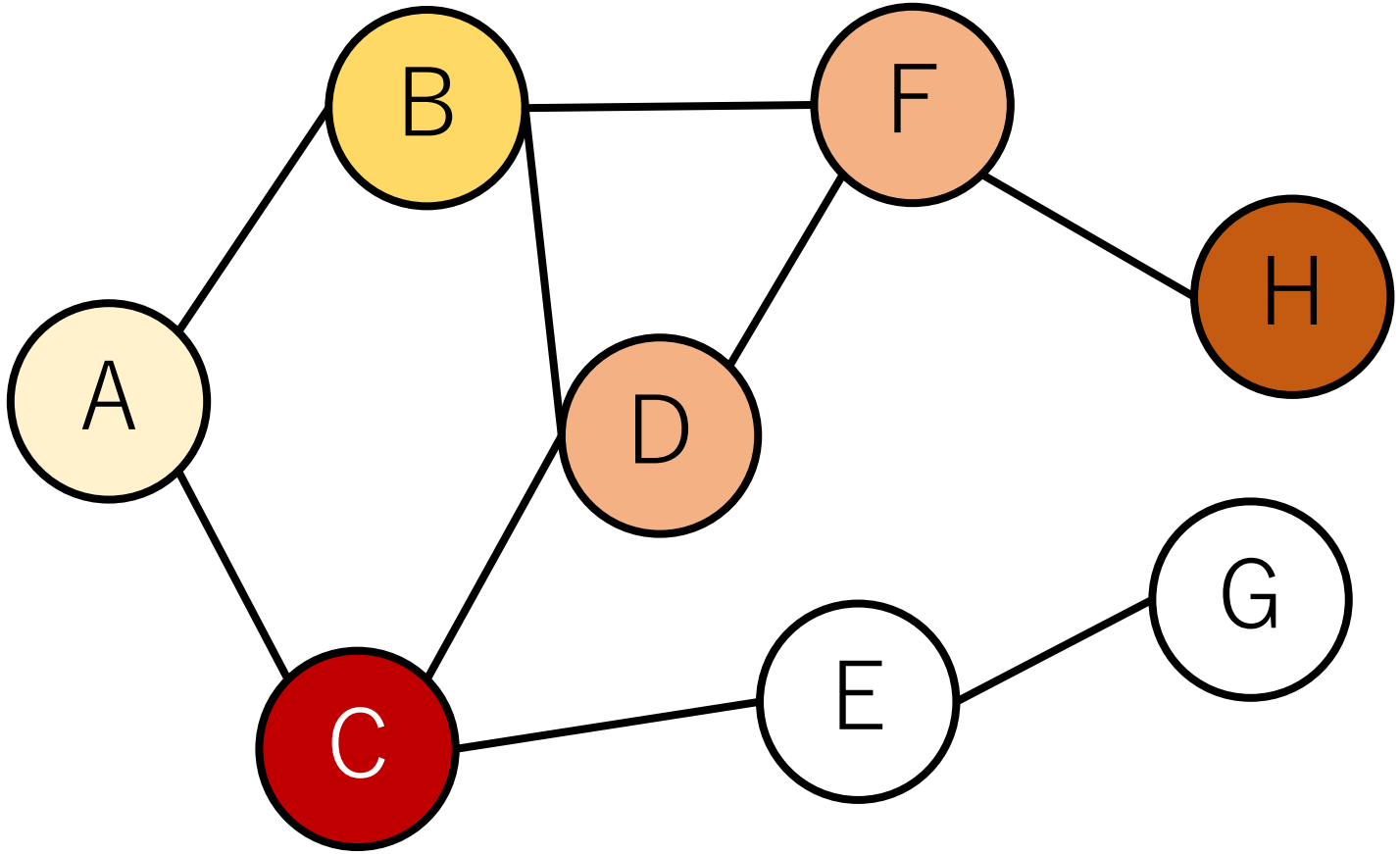


DFSの例

よって，Cに移動する．

スタックから取り出す．

C	D	**C**					

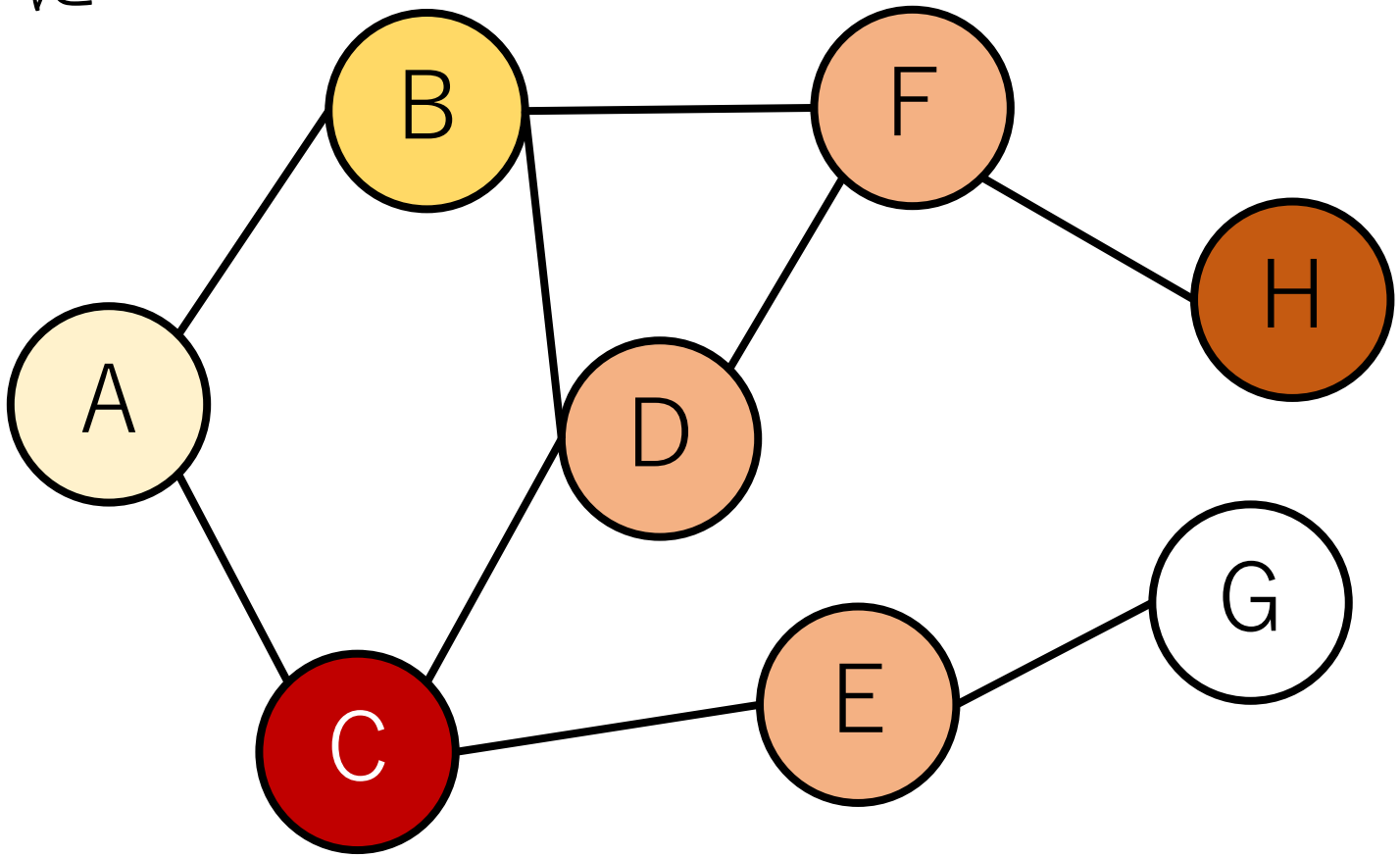


DFSの例

Eを見つける．（Aはすでに訪問済）

Eをスタックに入れる．

C	D	E					

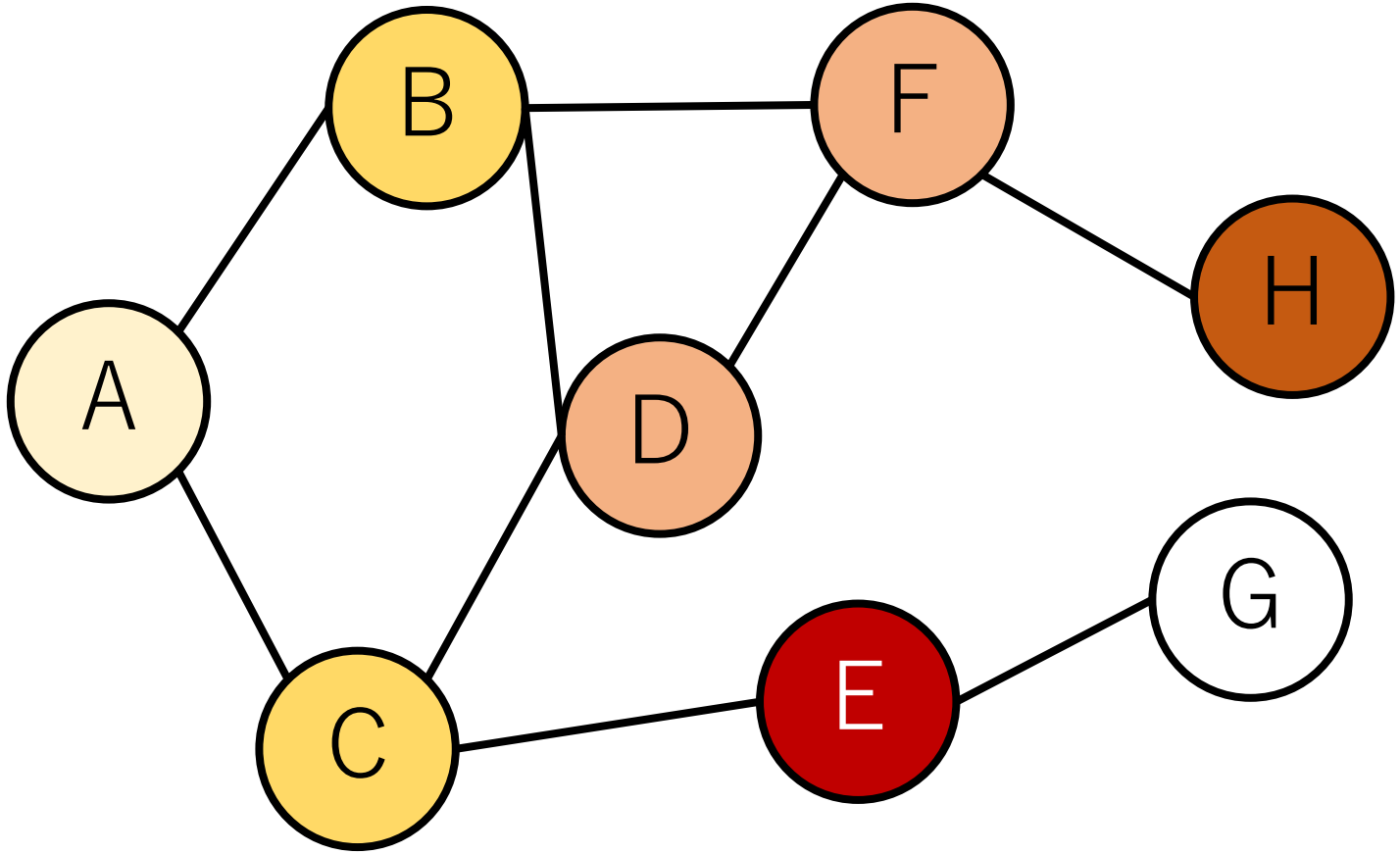


DFSの例

Eに移る.

スタックから取り出す.

C	D	**E**					

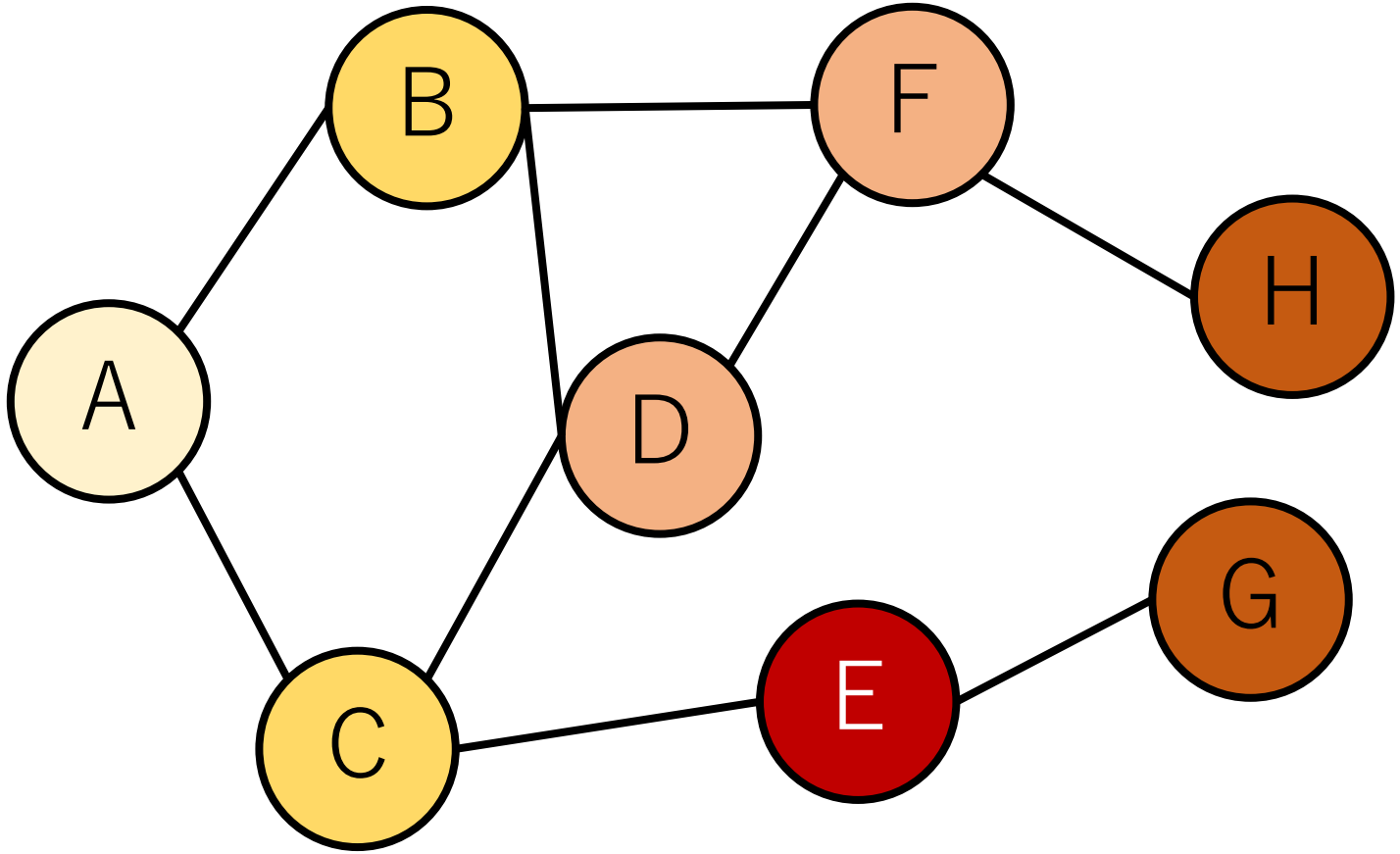


DFSの例

Gをみつける.

Gをスタックに入れる.

C	D	G					

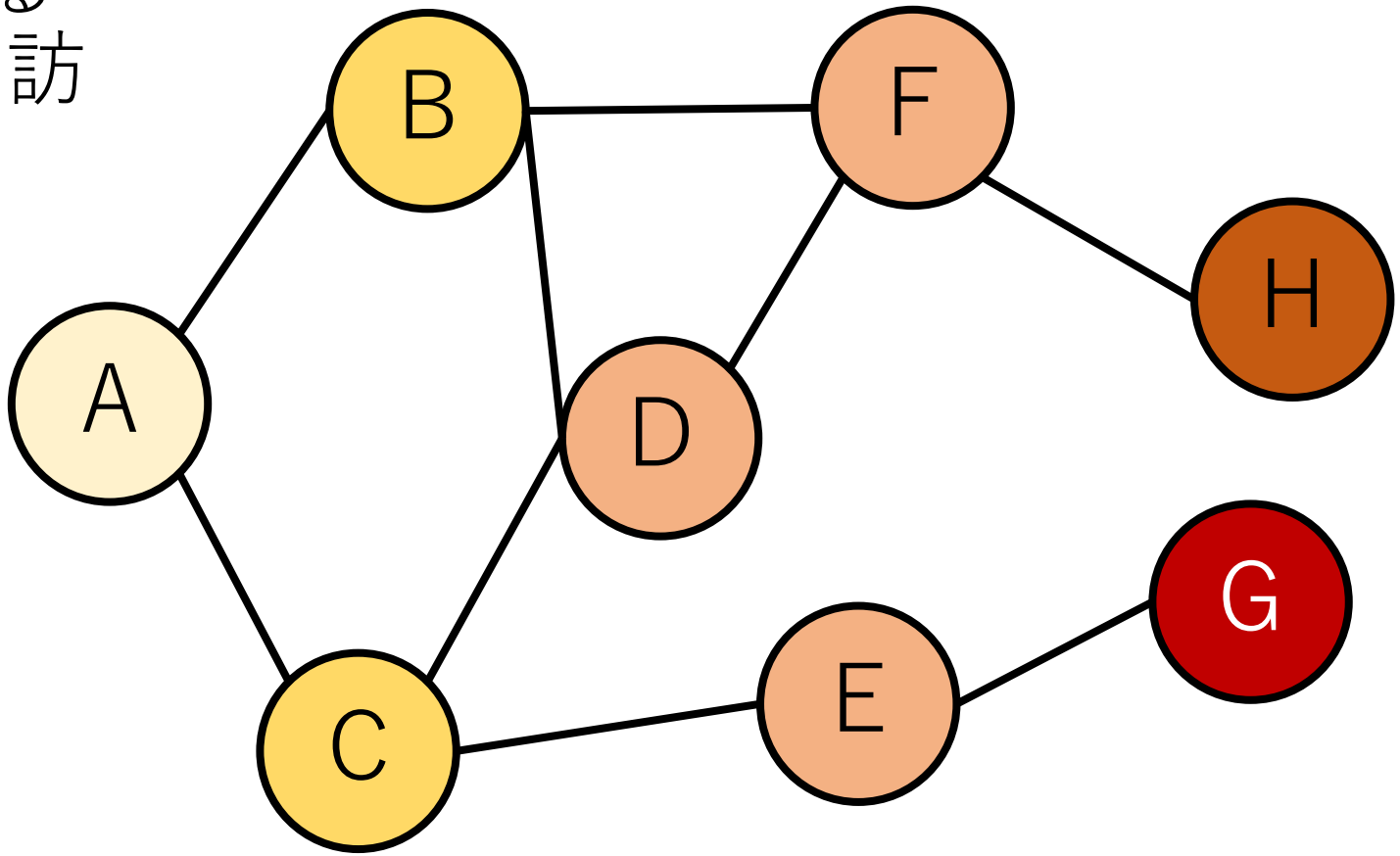


DFSの例

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．

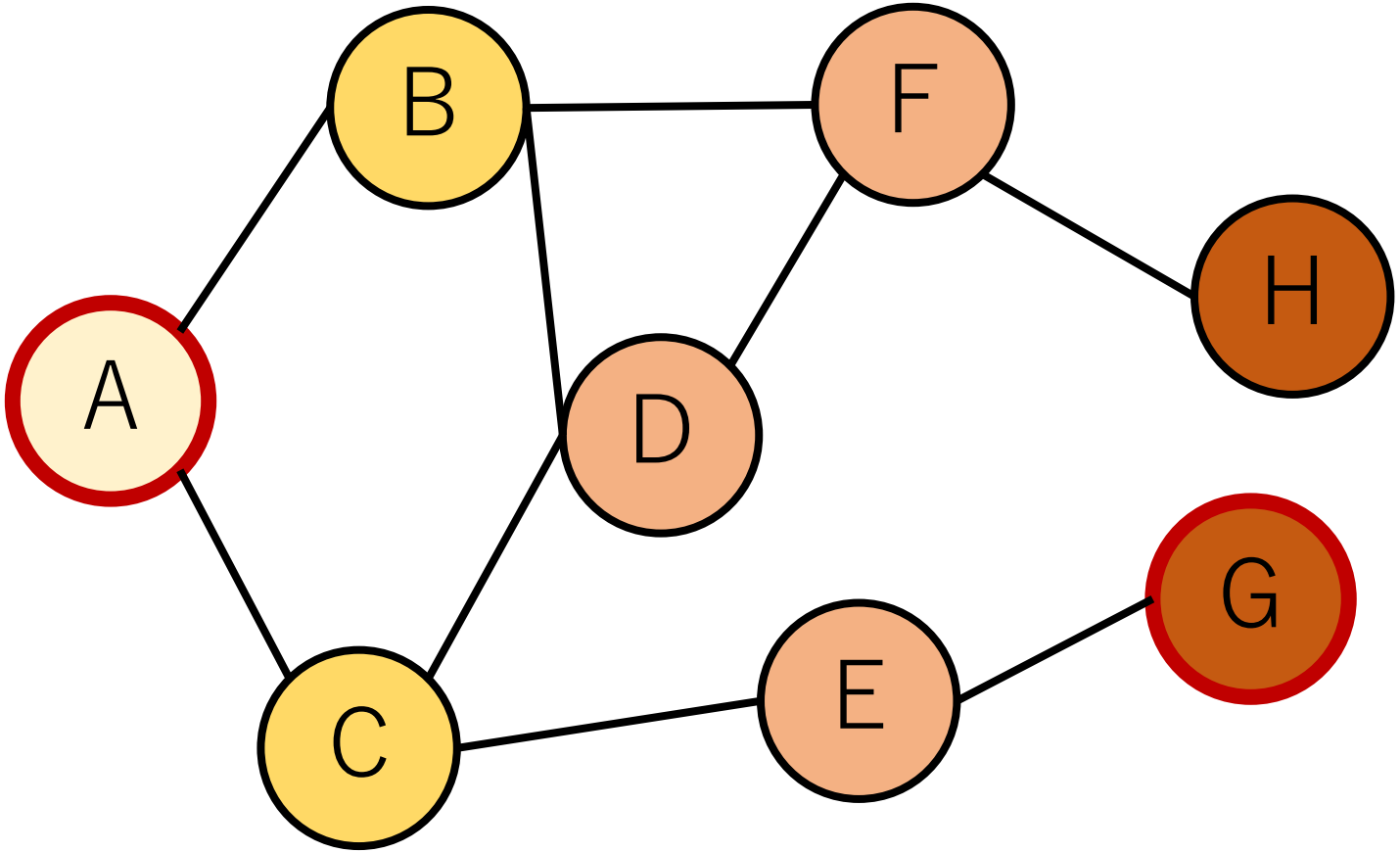
スタックから取り出す．

C	D	**G**					



DFSの例

AからGへはつながっていることがわかった！



C	D	**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

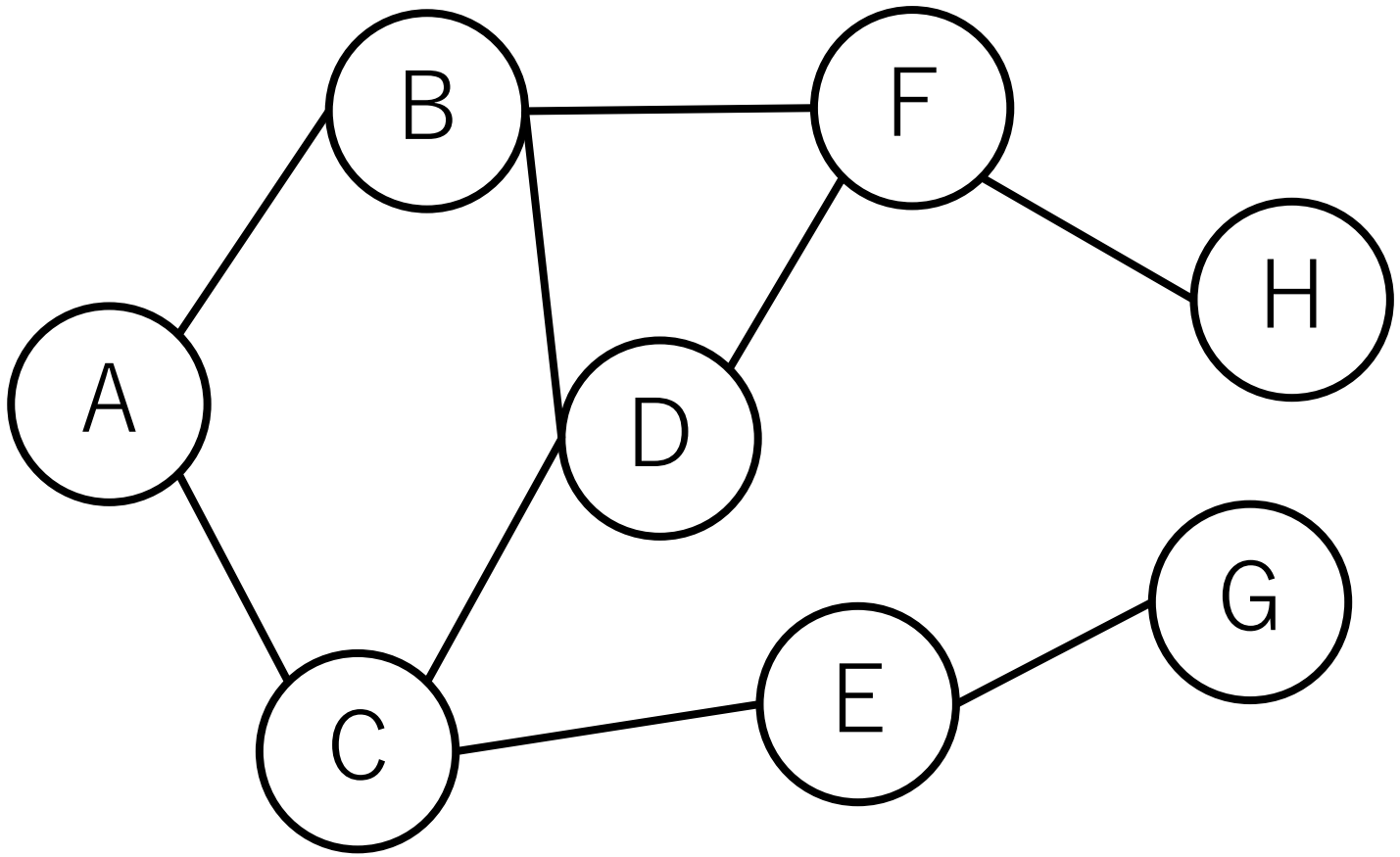
新しく見つかったノードを入れる

- 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）

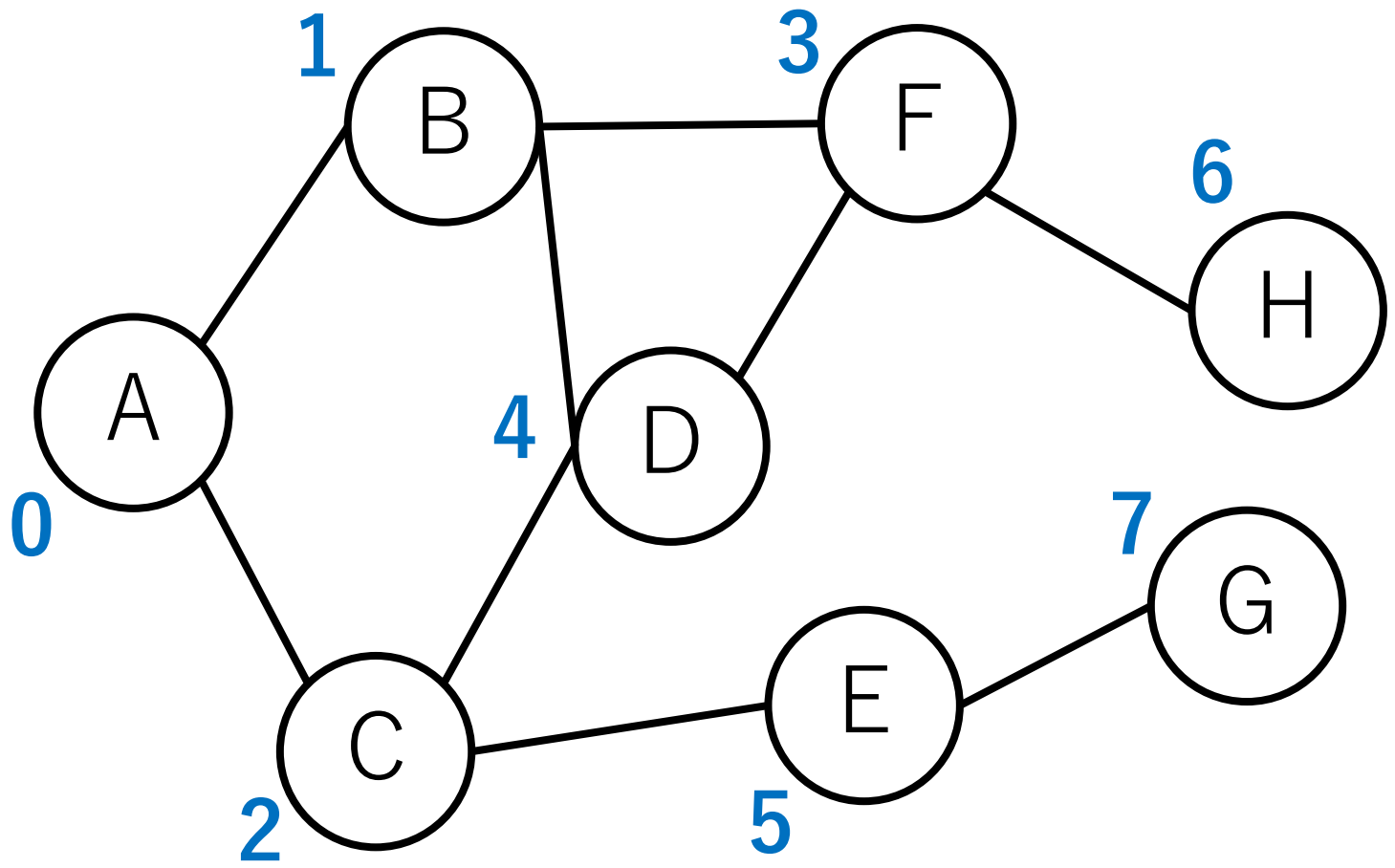
```
def bfs(edges, start, end):
        …
                # cur_nodeが未訪問なら処理する.
                # BFSでは本来必要のない判定だが，DFSでも同じ
                # コードが使えるようにするために入れている.
                if done[cur_node] != 2:
                        done[cur_node] = 2          # 訪問したので2
                        print('Moved to {}'.format(n_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] != 2:
                        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 F
Moved to D
Moved to E
Moved to H
Moved to G
=FOUND!=



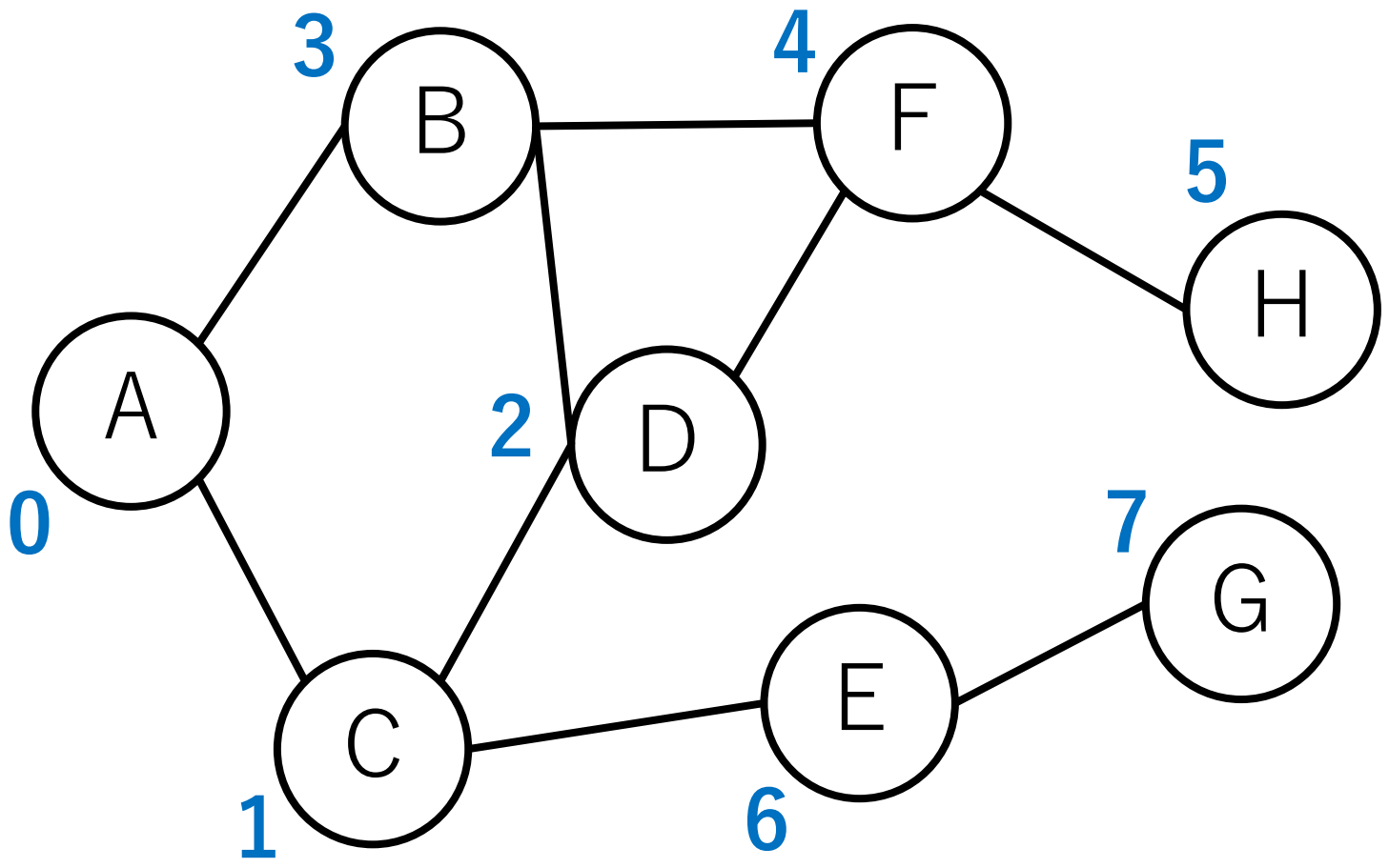
DFSの実装（探索）

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



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

メモリ消費量としてはBFSのほうが大きくなりがち.

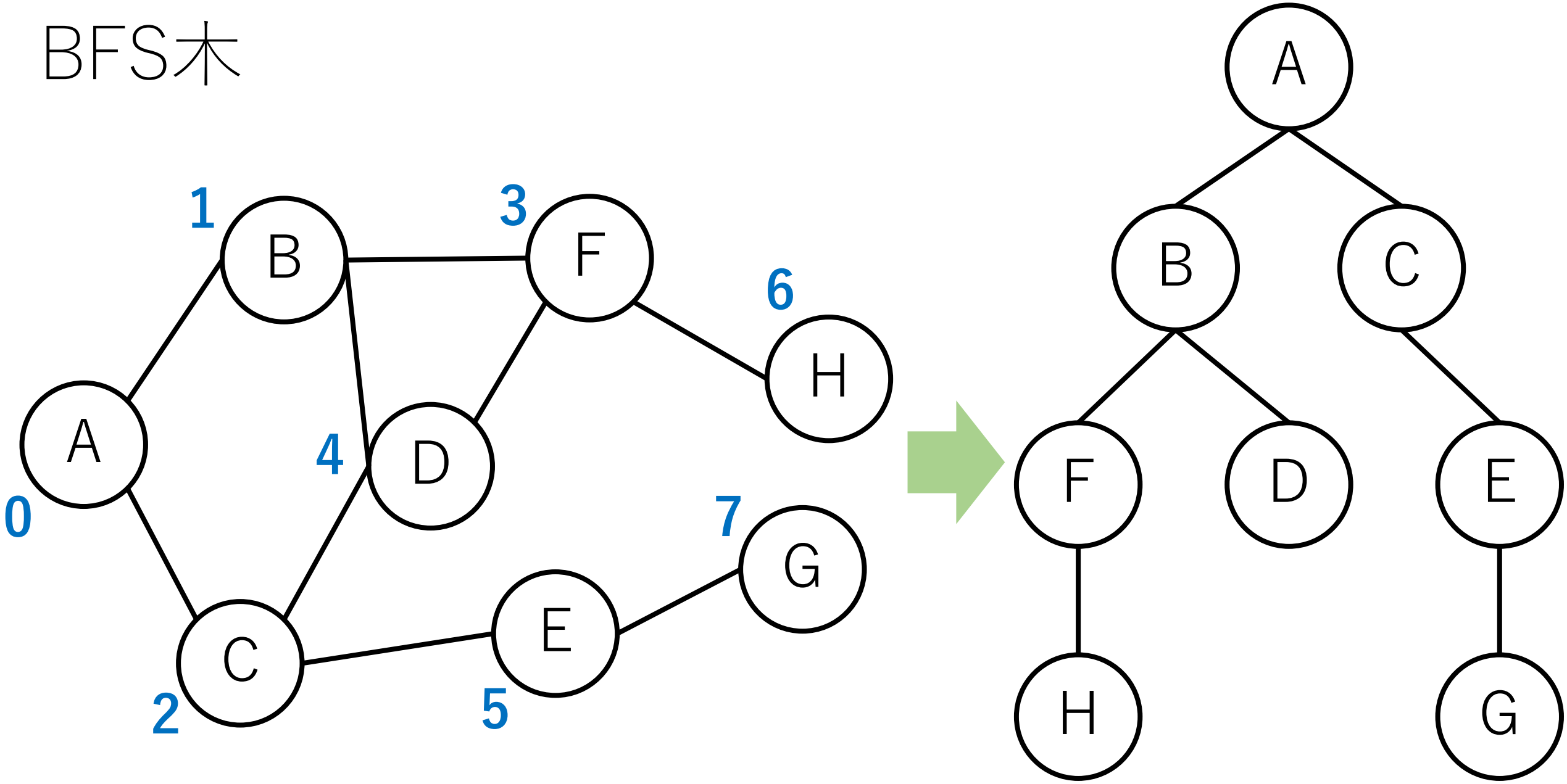
BFS：並行して探索途中となっている全てのノードの情報を保持する必要がある.

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主に保持することになる.

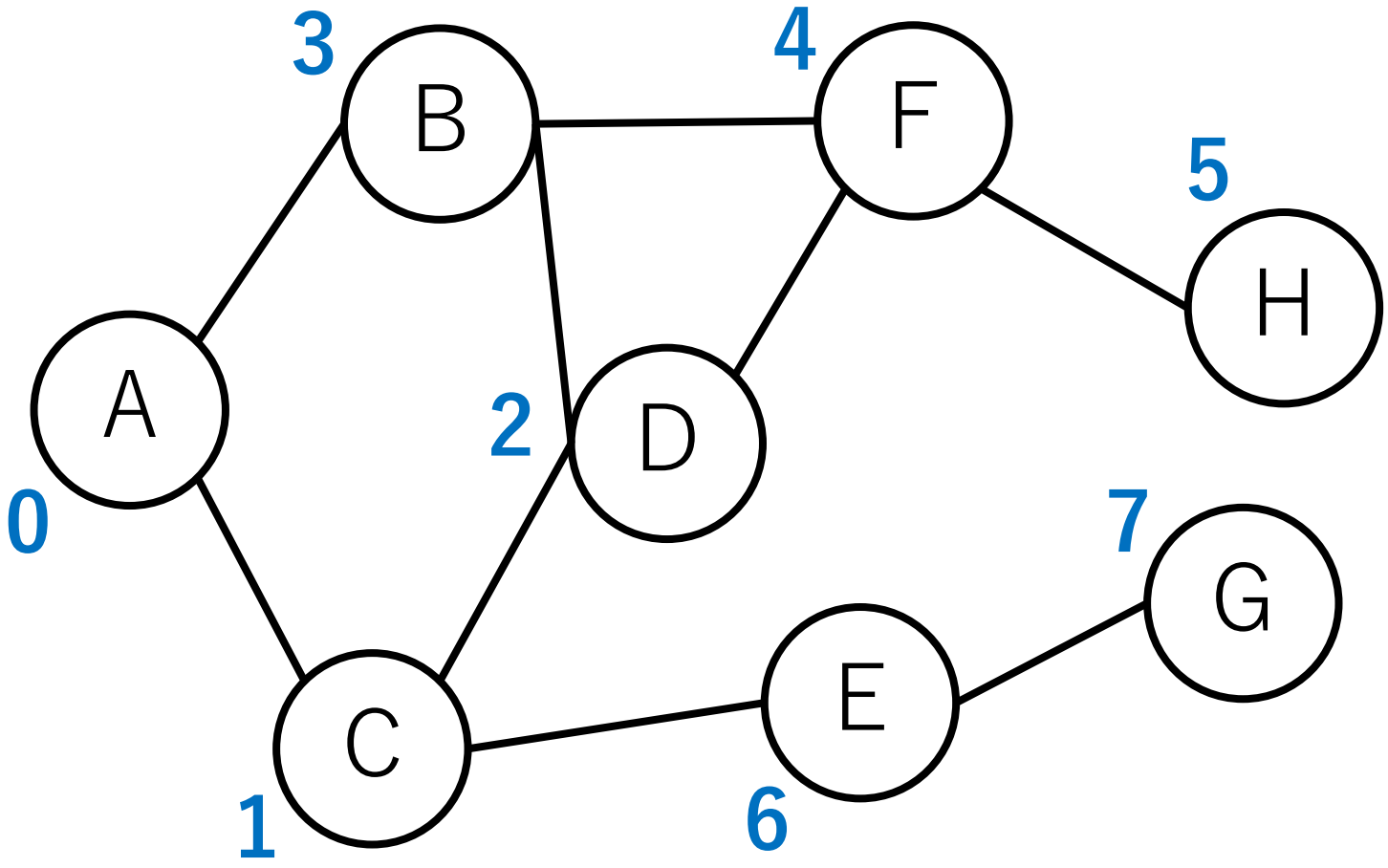
BFS木，DFS木

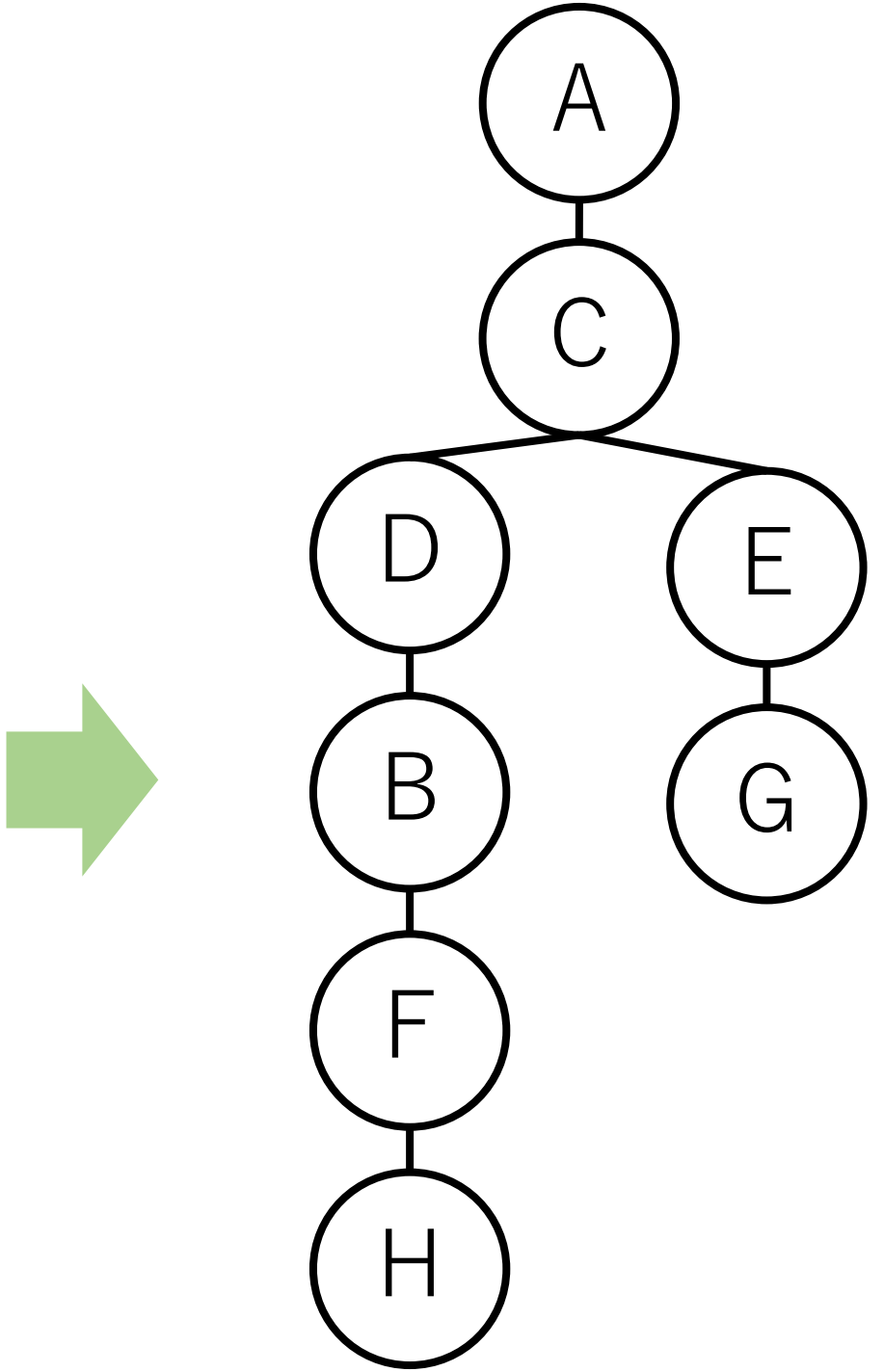
BFS，DFSを行うと，通った辺の順番を基に木が構成できる．

BFS木



DFS木





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，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が確認できない．

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、空きマスを辿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。

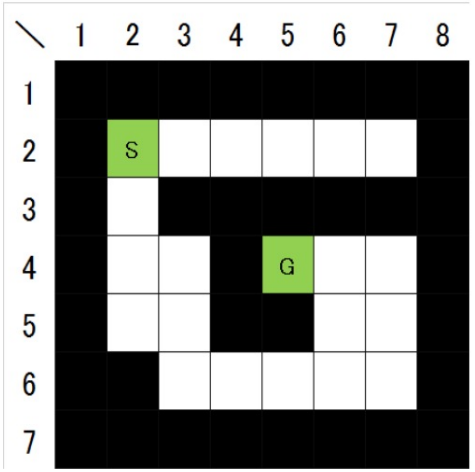


図1. 説明に用いる盤面

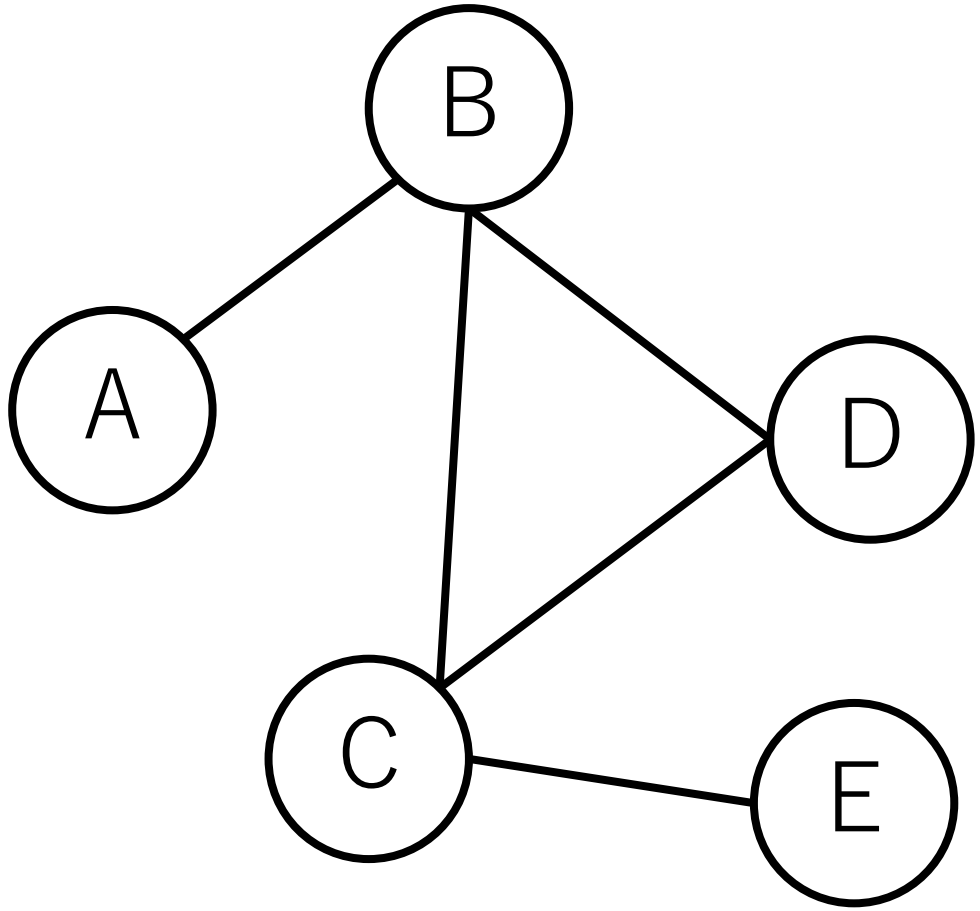
https://atcoder.jp/contests/abc007/tasks/abc007_3

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.

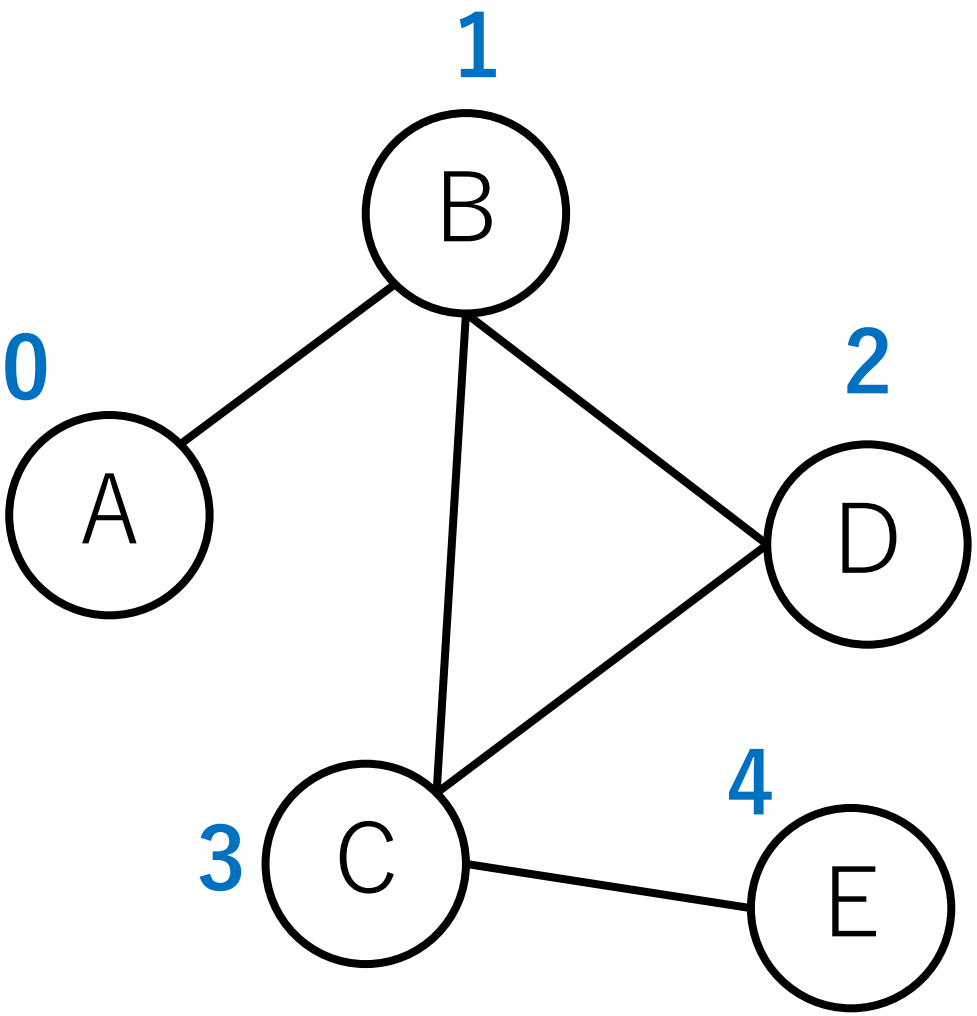
右のグラフではA-BとC-Eの辺.



橋の検出

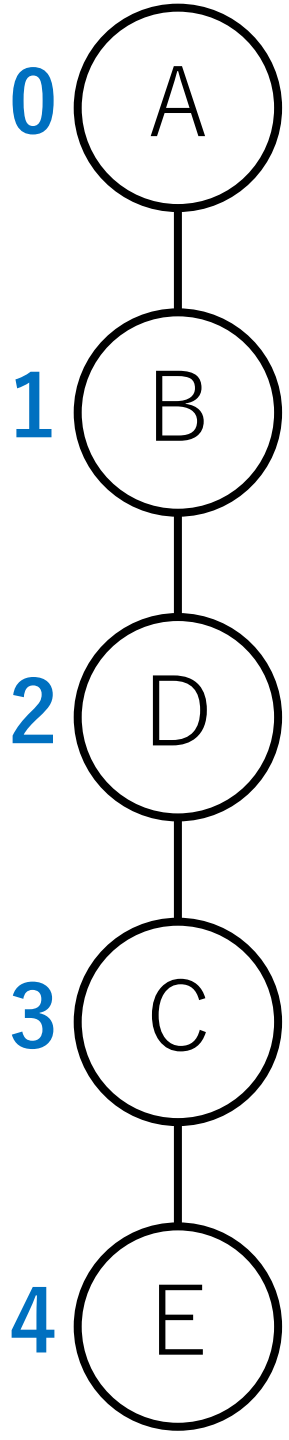
DFSでグラフを探索.

これによって訪問順が出る.



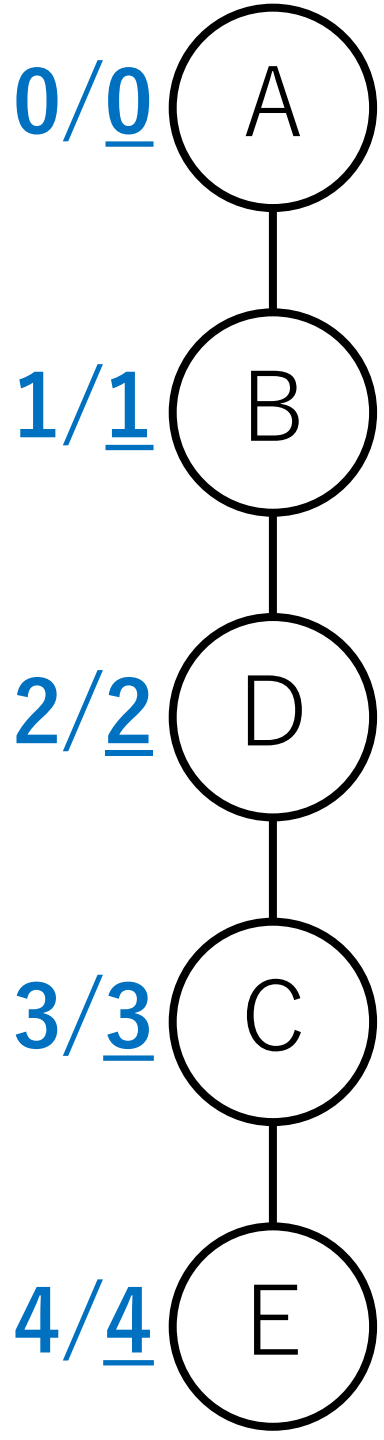
橋の検出

この訪問順はDFS木が生成される
（今回の場合はたまたま右のように
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.

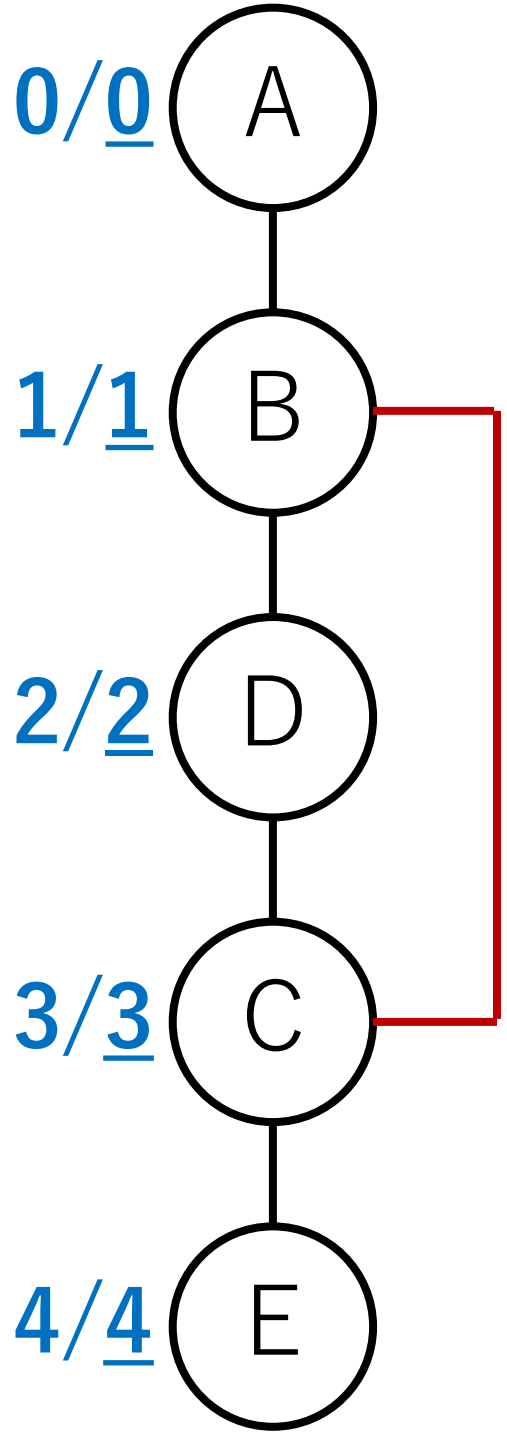


橋の検出

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．

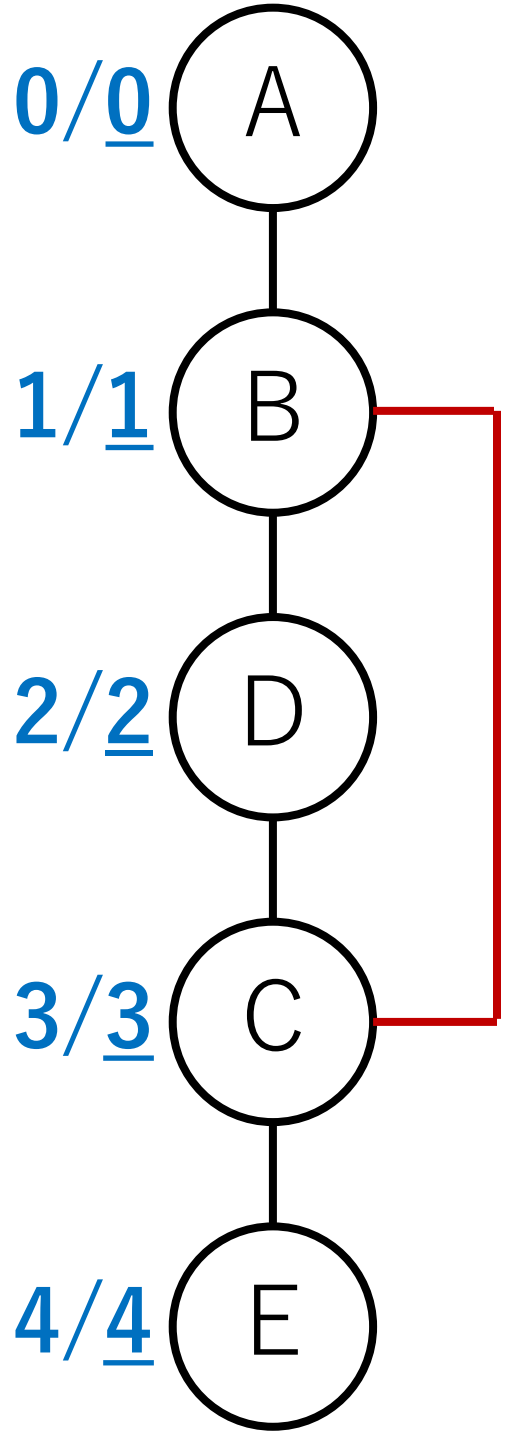
訪問順序がより前のノードに接続する，探索では使わなかった辺．



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．

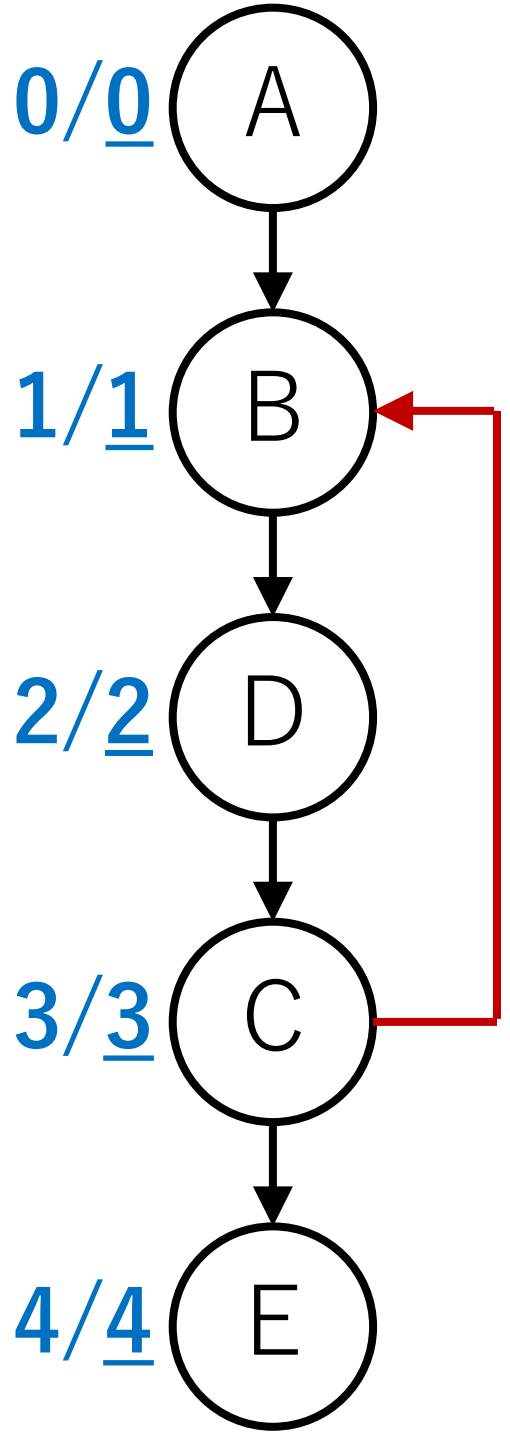


橋の検出

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.

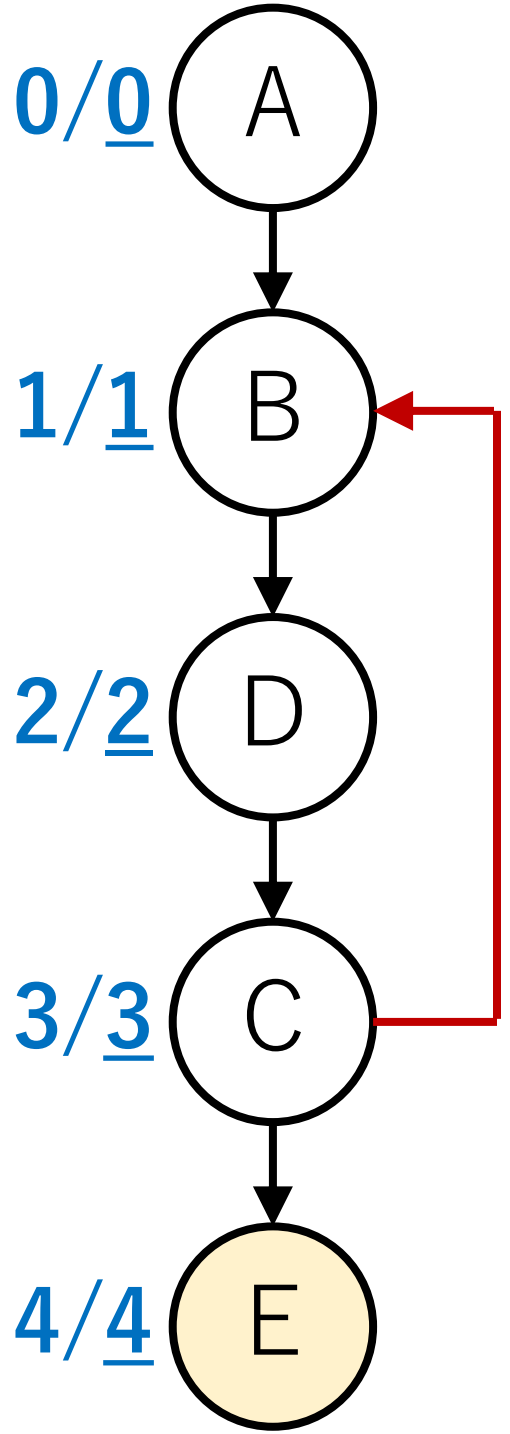
ただし，後退辺は1回しか使えない.



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．

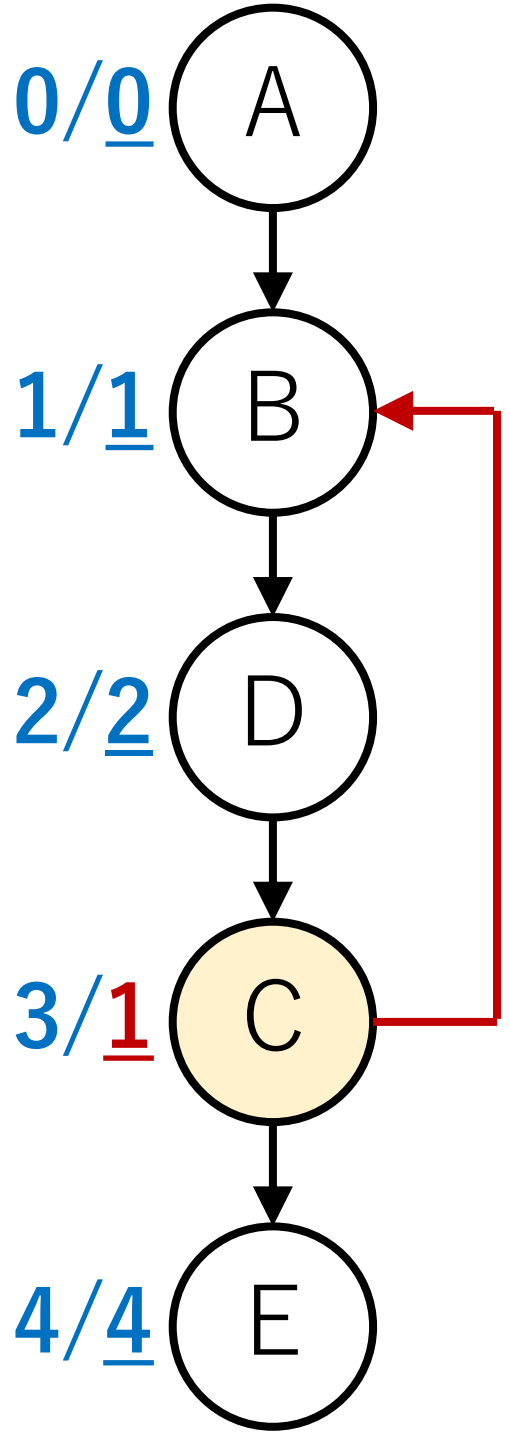
よって，ここは4そのまま．



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．

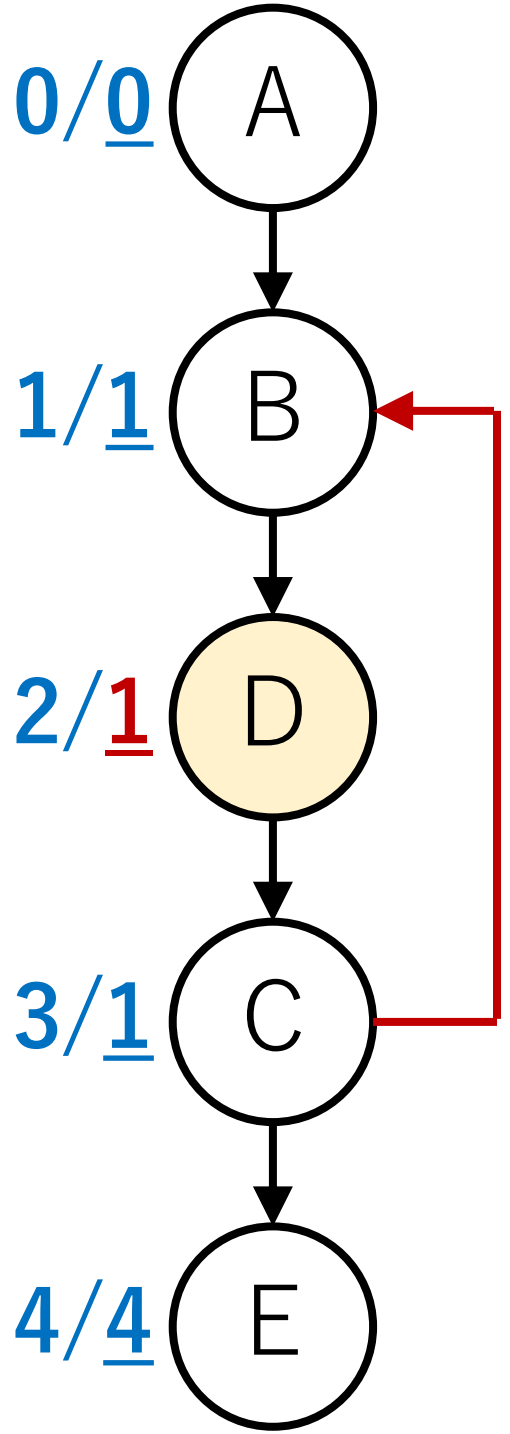
よって，ここは1に更新．



橋の検出

Dの場合，D->C->Bと辿ることが可能.

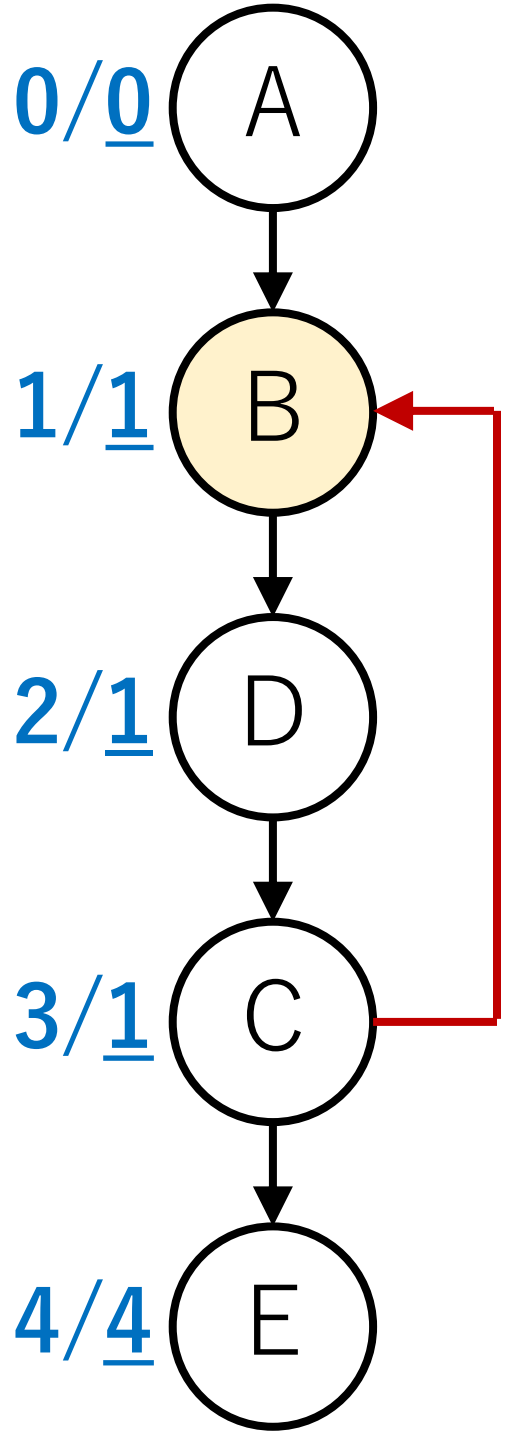
よって，ここも1に更新.



橋の検出

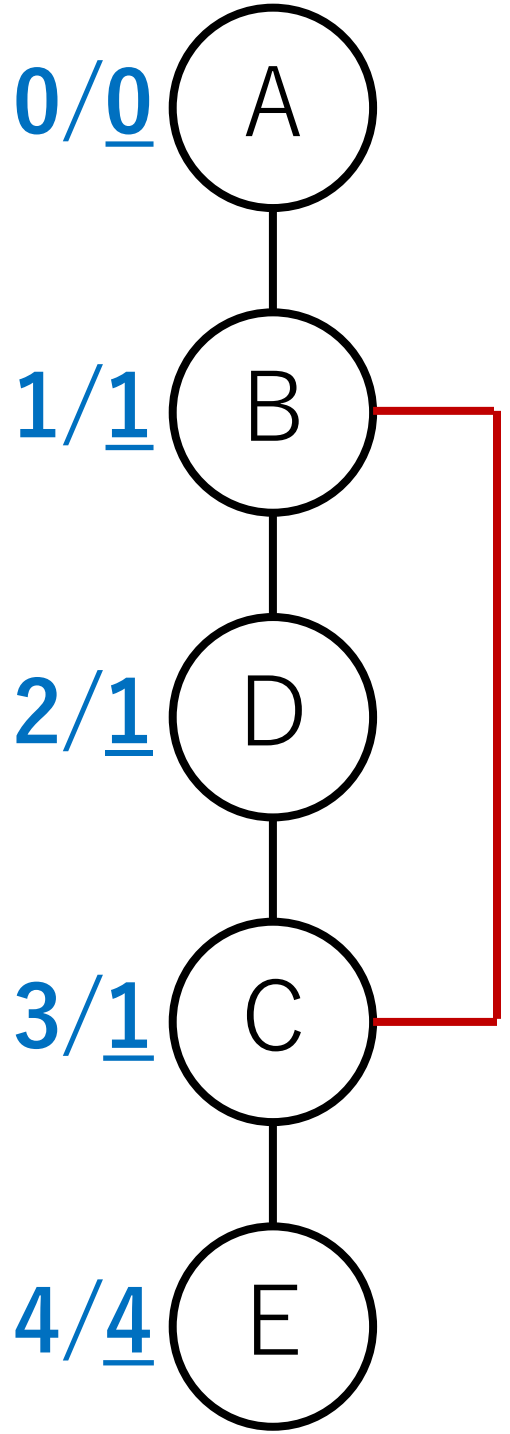
Bの場合，現在のlowlinkよりも小さな訪問順序を持つノードにはたどり着けない．

よって，ここは1のまま．



橋の検出

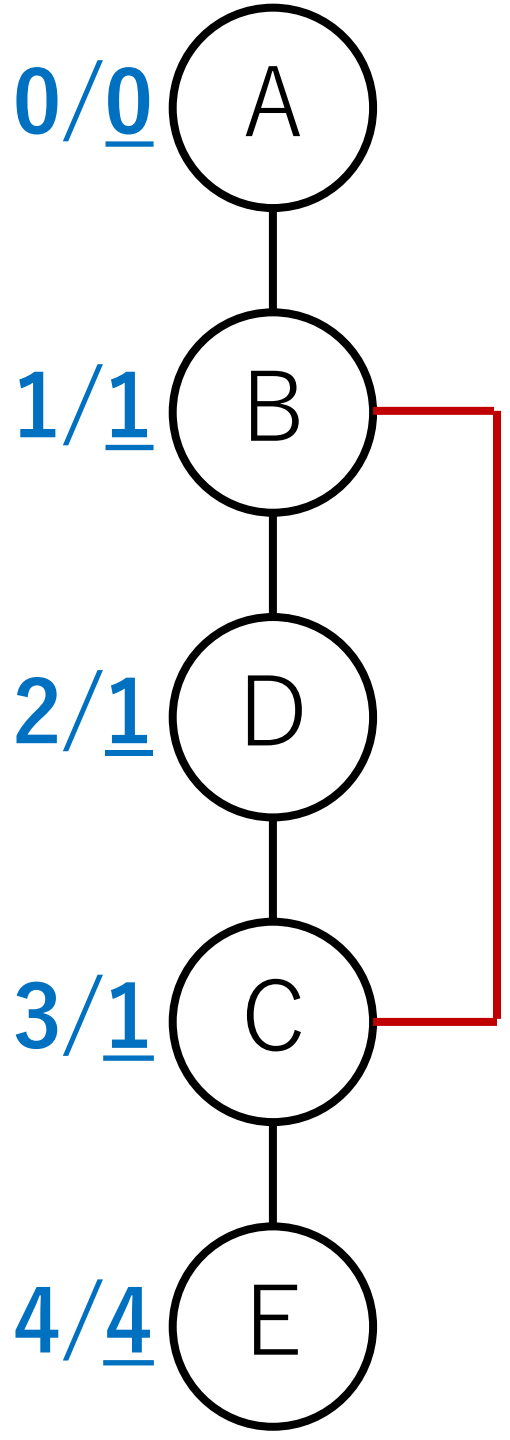
lowlinkは最終的には右のような値になる.



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．

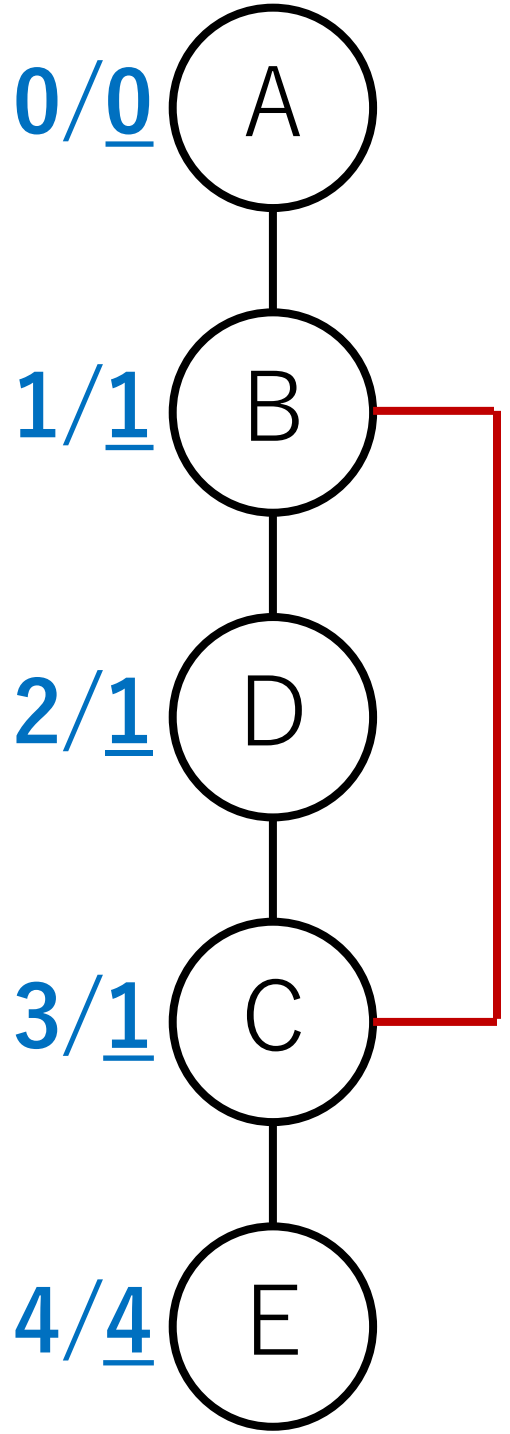


橋の検出

「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]

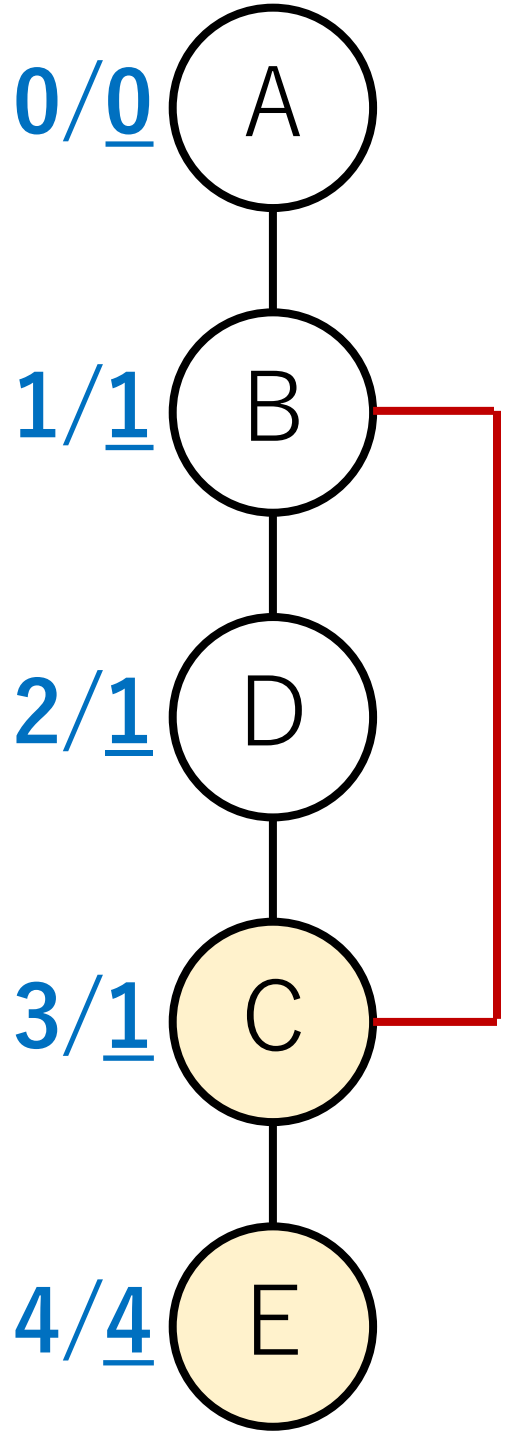


橋の検出

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．

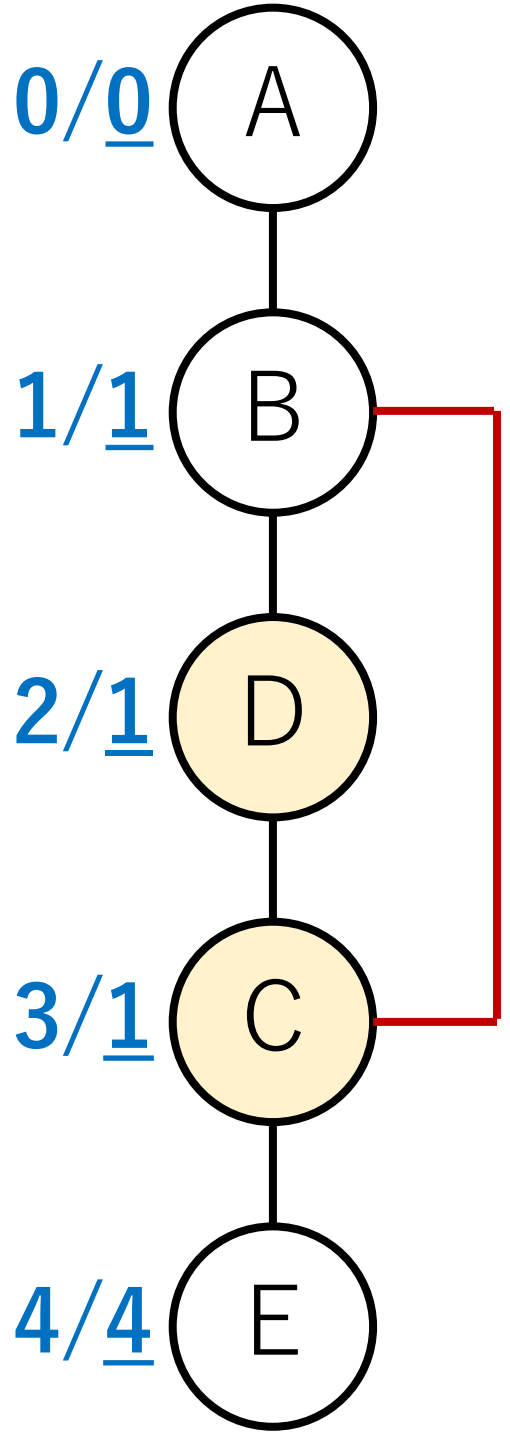


橋の検出

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．

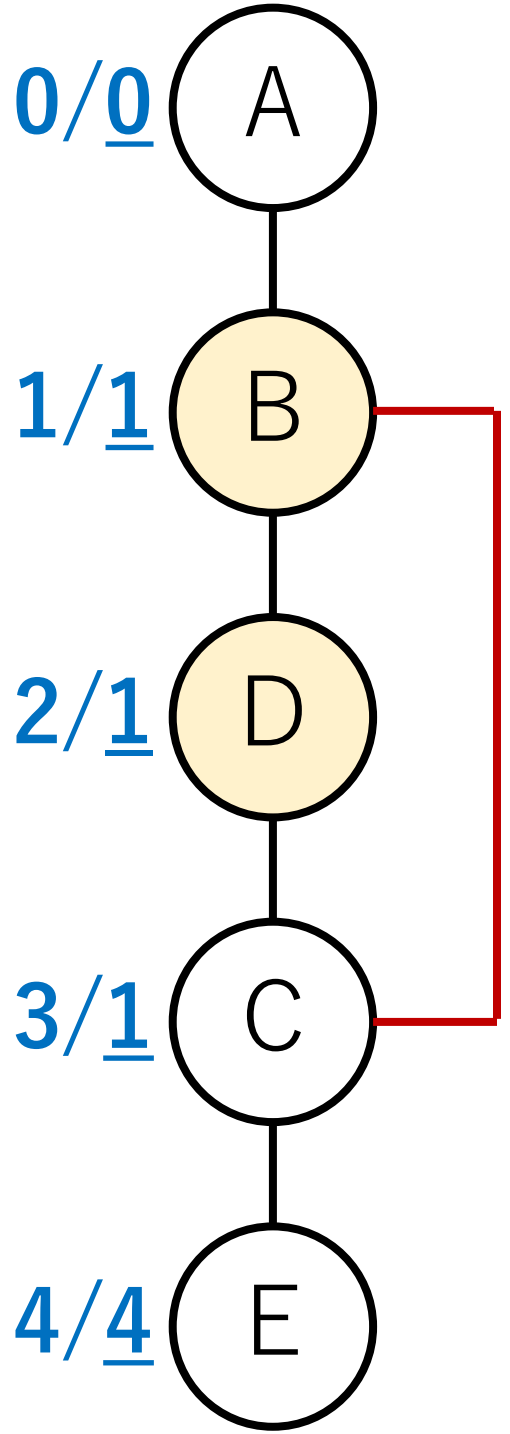


橋の検出

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．

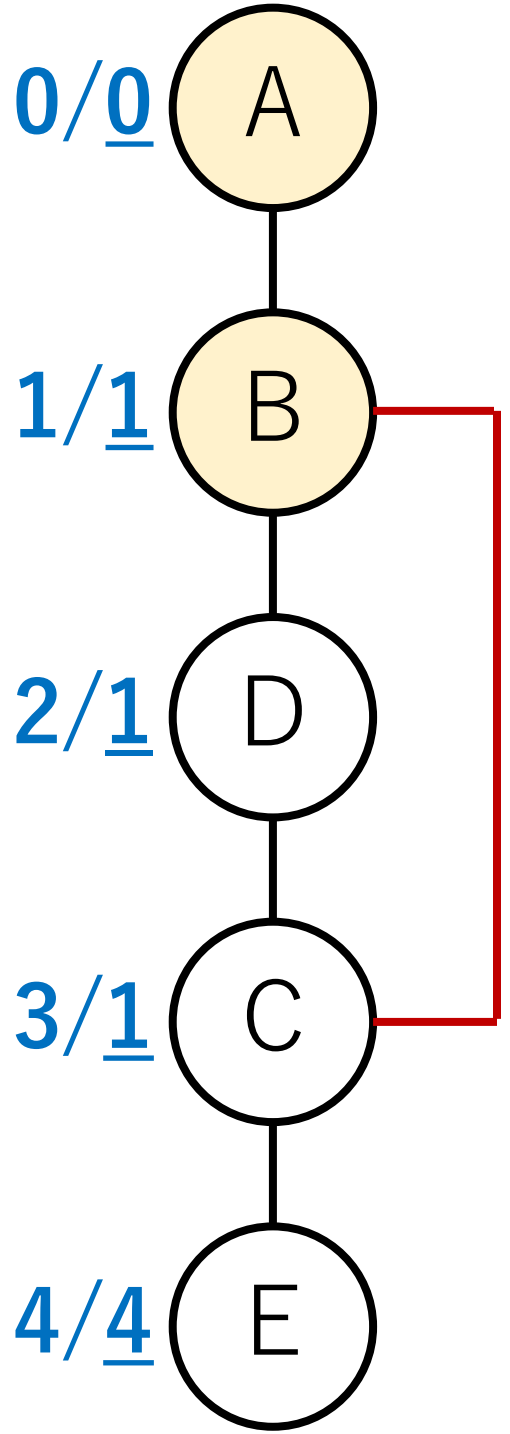


橋の検出

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DFSで探索すると同時に，lowlinkを定数回で更新できる．

このため，計算量はDFSのときと変わらない．

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，，，

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，少しづつ理解しながら，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

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は使えませんが，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 [2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.