

Algorithms (2021 Summer)

#11 : グラフアルゴリズム2

矢谷 浩司

期末試験

日時：7/21 13:00集合， 13:10開始， 14:40頃解散予定

試験時間：70分を予定

会場：工学部2号館内教室

(後日，座席を指定しますので詳細をお待ち下さい。)
ただし，特別な事情があり，工学部2号館での受験
ができない人は，オンラインでの受験が認められる
ことがあります。

【重要】 期末試験受験者調査

期末試験を受験する予定の人は全員，以下のアンケートで回答をしてください。

<https://forms.gle/p7AcNuBxWdzkT73C6>

回答期限：7/5 12:00（正午）

このアンケートに回答していない場合，期末試験の受験を認めないことがあります。

オンラインでの受験

後日、先のアンケートとは別に、学科としてオンライン受験を希望する方の調査を行いますので、そちらにも必ず回答をお願いします。

EEIC以外の学部・学科の方も、この学科が管理するアンケートにお答えください。

このアンケートに答えて、承認された方のみ、オンラインでの受験が許可されることとなります。

サンプル問題

<https://hackmd.io/@yatani/H1rvBati>

このサンプル問題は、本講義の期末試験で出題される問題のフォーマットを例題を通じて示すものです。どのような形式で本講義で学んだことを問われるか、の参考にしてください。

このサンプル問題は、大問・小問の数、出題範囲、問題の難易度を規定するものではないことに留意してください。

成績評価の一部変更

期末試験が7/21となったため、13回目のExtra課題を中止といたします。これに伴い、Extra・レポート課題による採点を以下のように変更いたします。

Extra課題：33点満点で採点（13回目はなし）

レポート課題：36点満点で採点し、33点満点に換算

不足する3点：榎崎様の特別講義の感想提出

7/14 特別講演！

特別ゲスト：

SOMPOホールディングス株式会社 CDO 榎崎浩一様

SOMPOホールディングス株式会社が
目指すデジタル戦略におけるデータ活用，
そしてアルゴリズムなどコンピュータ科学
の知識がビジネスの世界でどのように
生かされるかを，ご自身の経験とともに
お話しいただきます。



7/14 特別講義！

時間：14:00～15:00


場所：工学部2号館246号室， およびオンライン

**Zoom webinarは授業のものとは違うURLになりますので、
ご注意ください！**

7/14 特別講義 聴講申込み

<https://iis-lab.org/dls/koichinarasaki/>

**本講義受講者も別途登録が必要ですので、ぜひ今よろしく
お願いいたします！**

また、本学教職員，学生さん全ての方が参加できますので、
ご友人やお知り合いの方もお誘いください！

7/14の授業に関して

本講義受講者に対しては，特別講義終了後，皆さんの感想を伺うアンケートを流します。

感想は匿名化の上，ご講演者の方に共有される予定です。

この感想提出が3点分となります。

前回のアナウンスと違い，追加のボーナス点ではなくなりましたので，ご注意ください。

提出期限は当日24:00の予定です。

7/14の授業に関して

本特別講義に合わせて7/14の授業は以下のように変更します。ご承知おきください。

13回目の講義（矢谷担当分＋特別講義）

→矢谷担当分は事前に録画し，7/12に公開予定。

→13回目の講義までに見ておいてください。

13回目のコードチャレンジ

→~~Extra1問のみ。特別講義終了後，配信予定。~~

7/14の授業に関して

ただし以下のような場合，減点，不採点の対象となります。

- 十分な時間ご講演に出席していない。
 - zoomのログ，教室での出席で確認。
- 意味のある感想でない。
- その他，誠実な出席や感想が確認できないケース。

レポート課題提出

UTAS登録者にはコードチャレンジで使用しているメールアドレスでTurnitinへの登録を行いました。

スパムフォルダ等も確認してください。

https://www.turnitin.com/login_page.asp?lang=en_us

レポート課題の提出はTurnitinより行ってください。それ以外の提出方法は認められません。

講義のホームページに提出方法の詳細を公開していますので、参考にしてください。

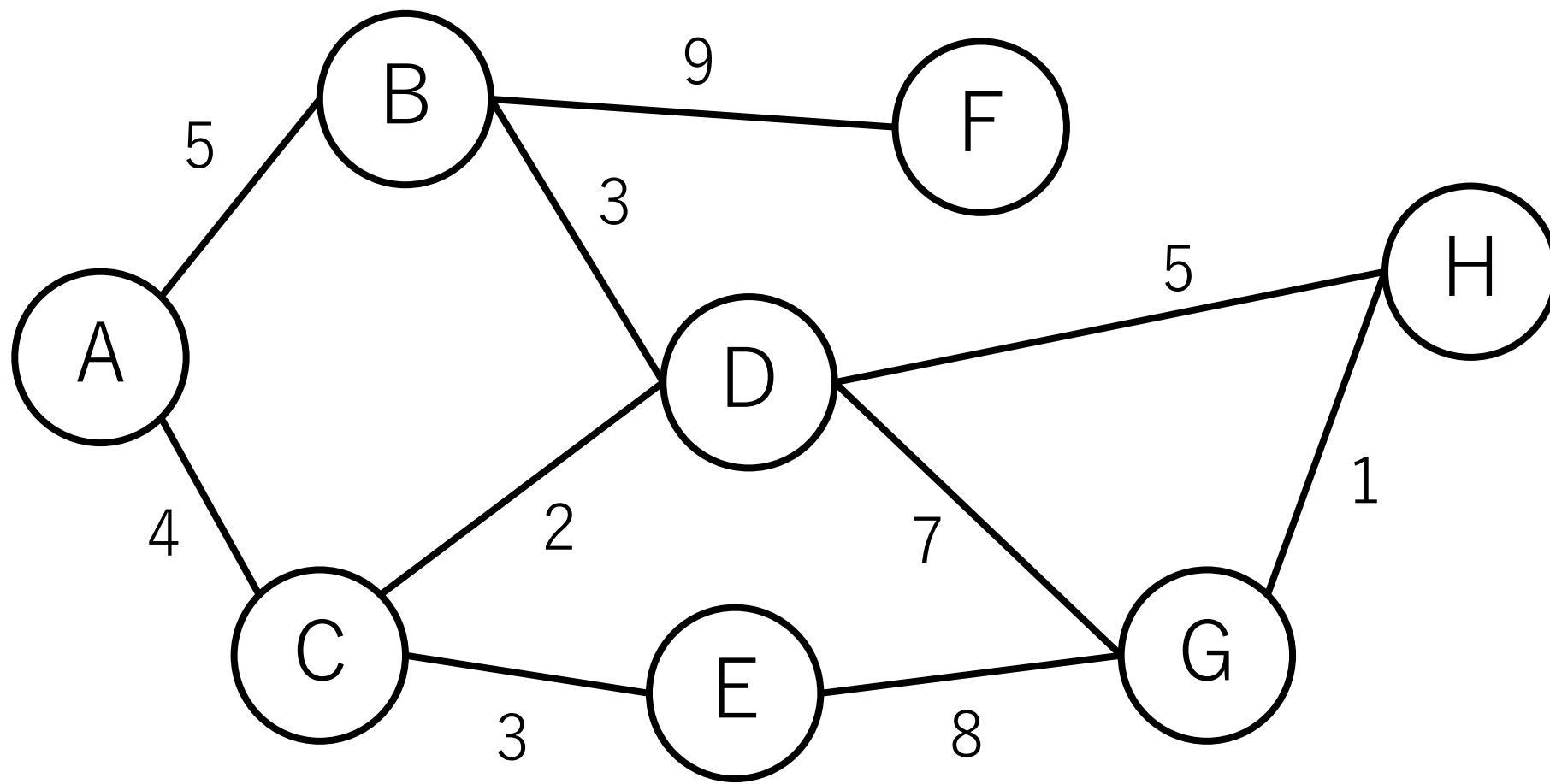
今日のテーマ

最小全域木

トポロジカルソート

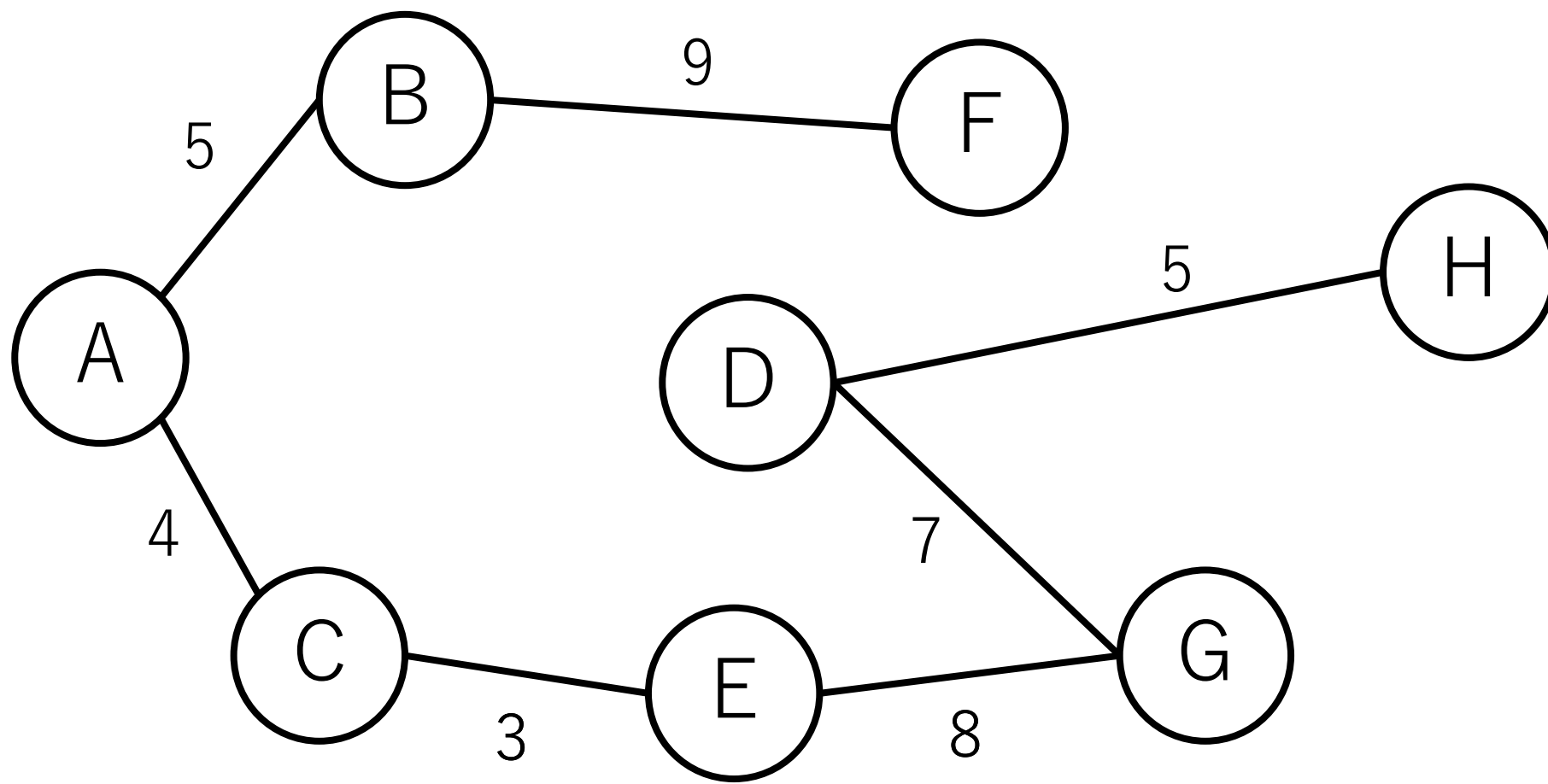
全域木 (spanning tree)

グラフにおいて、すべての頂点がつながっている木（閉路を持たない連結グラフ）。



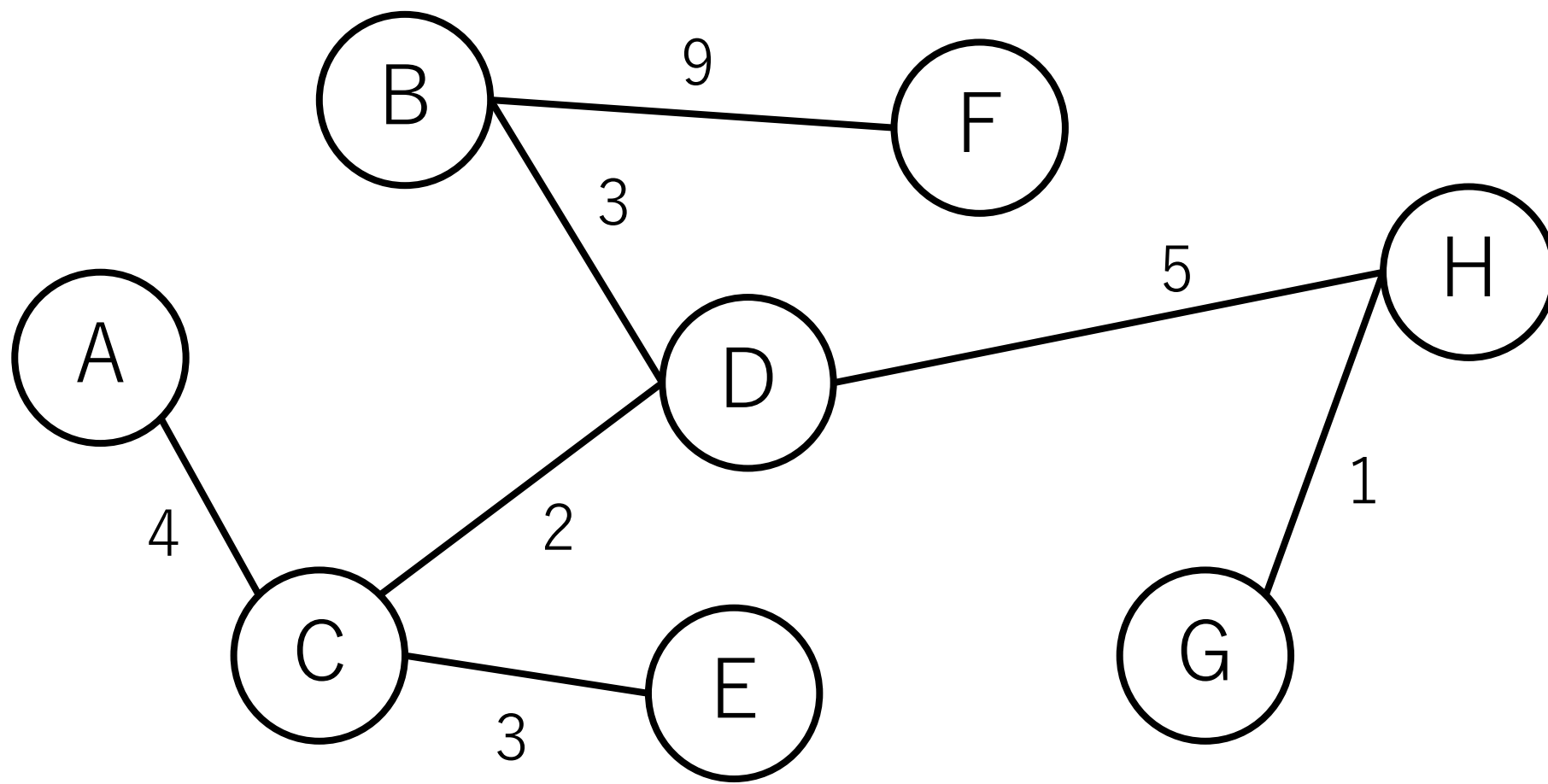
全域木

全域木の1例.



最小全域木 (minimum spanning tree)

全域木の中で辺の距離（コスト）の総和が最小になるもの。



最小全域木がわかると何が嬉しい？

「複数の建物を有線のネットワークで接続する時、コストを最小にするように線を引きたい。」

全体のコストを最小にしつつ、ノード間がつながっていることが保証されないといけない。

最小全域木の求め方

辺ベースのアプローチ

存在する辺を距離の短い順に並べて順に入れていき、閉路が出来ないことが確認できた場合は追加し、全部の辺をチェックしたら終了。

ノードベースのアプローチ

すでに到達した頂点の集合からまだ到達していない頂点の集合への辺のうち、距離が最短のものを追加し、全ノードつながったら終了。

最小全域木のアルゴリズム

辺ベースのアプローチ：クラスカル法

存在する辺を距離の短い順に並べて順に入れていき、閉路が出来ないことが確認できた場合は追加し、全部の辺をチェックしたら終了。

ノードベースのアプローチ：プリム法

すでに到達した頂点の集合からまだ到達していない頂点の集合への辺のうち、距離が最短のものを追加し、全ノードつながったら終了。

最小全域木のアルゴリズム

辺ベースのアプローチ：クラスカル法

存在する辺を距離の短い順に並べて順に入れていき、閉路が出来ないことが確認できた場合は追加し、全部の辺をチェックしたら終了。

ノードベースのアプローチ：プリム法

すでに到達した頂点の集合からまだ到達していない頂点の集合への辺のうち、距離が最短のものを追加し、全ノードつながったら終了。

クラスカル法 (Kruskal)

#1 全ての辺を距離の短い順にソート.

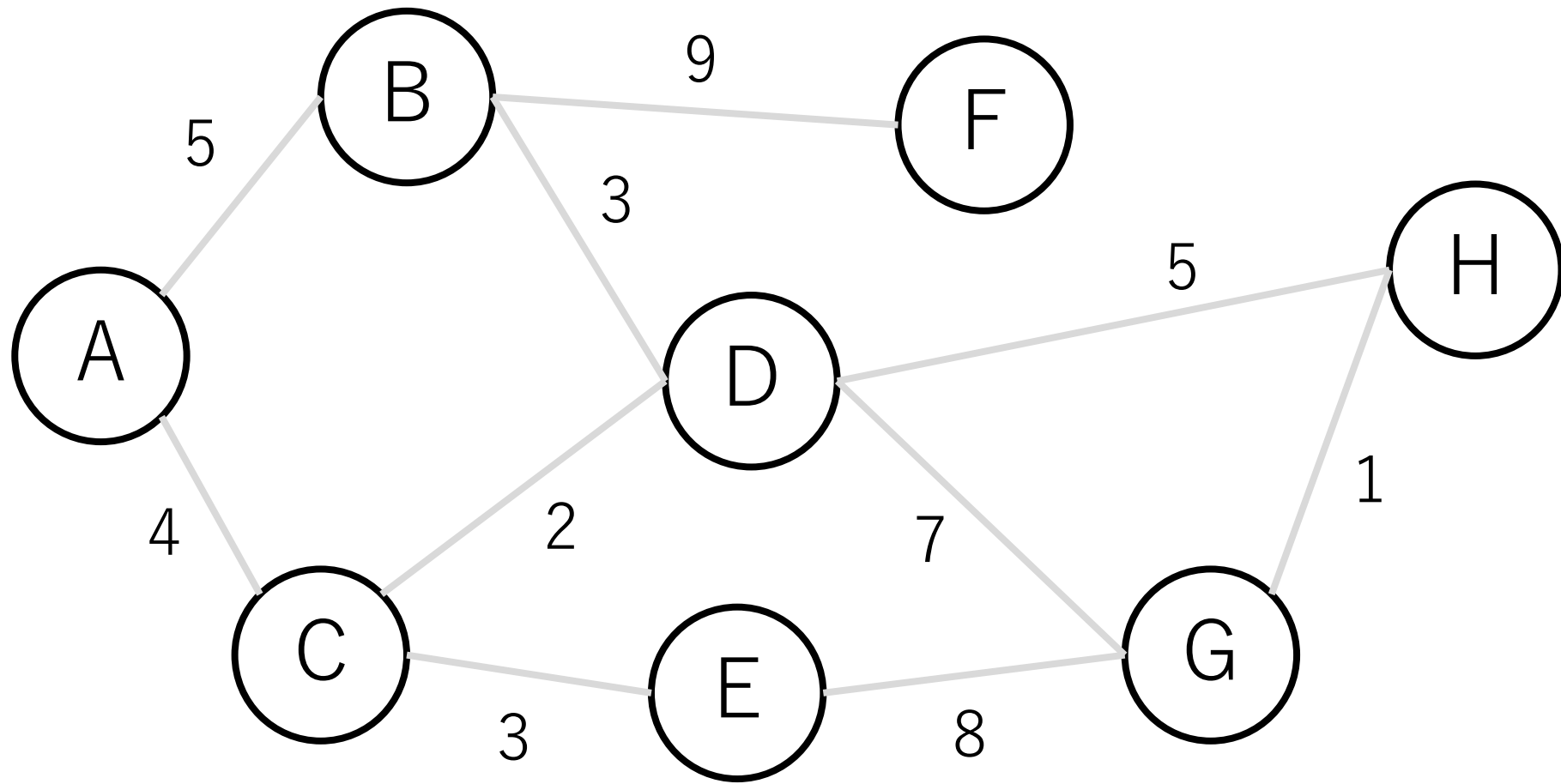
#2 一番距離の短い辺からスタート.

#3 今までに出来た木に辺を追加した時, 閉路が新しく出来ないことを確認する. 出来ない場合, この辺を最小全域木に追加.

#4 以降, 全ての辺をチェックするまで#3を繰り返す.

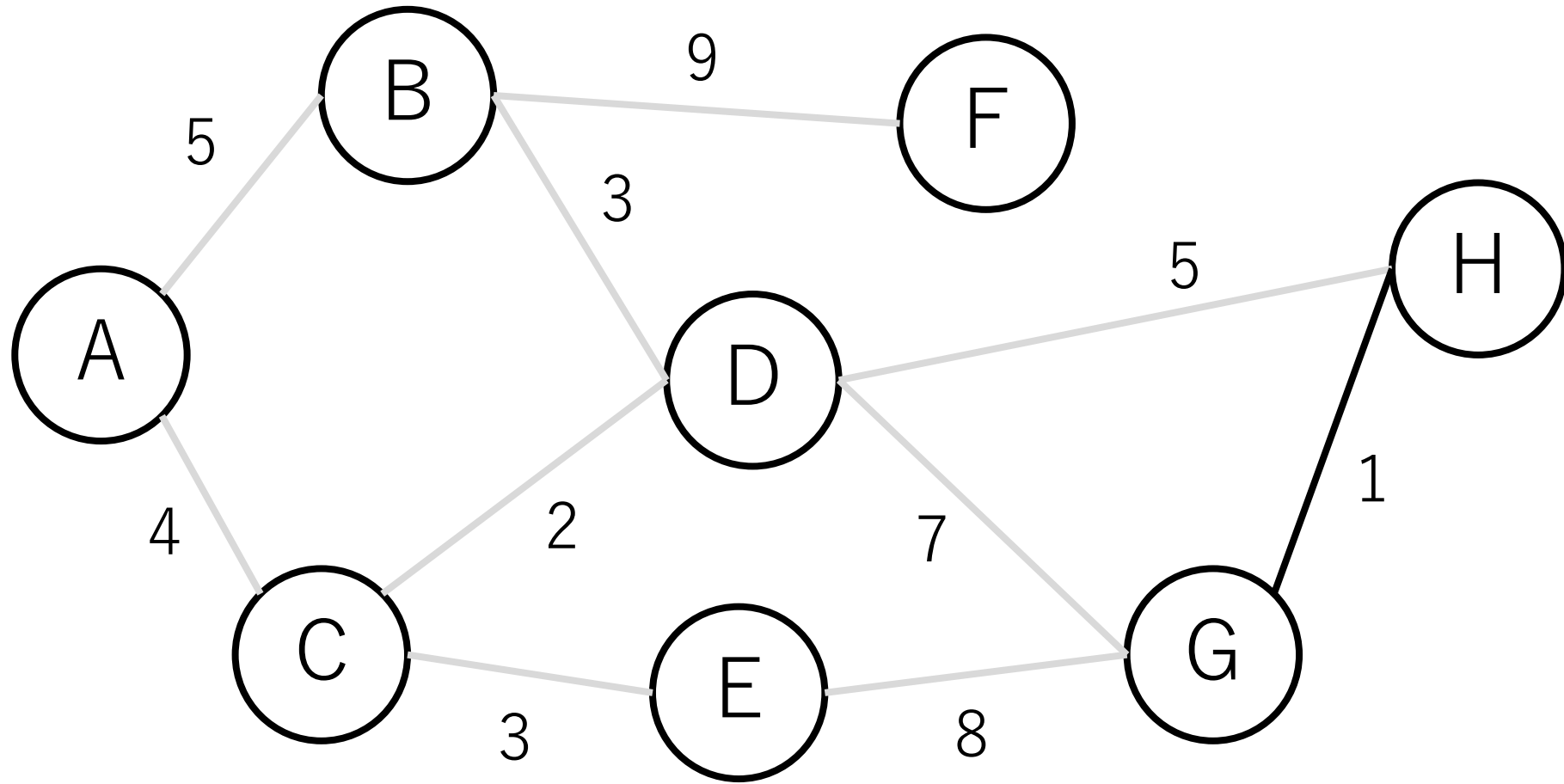
クラスカル法の例

すべての辺を距離の短い順にソート。



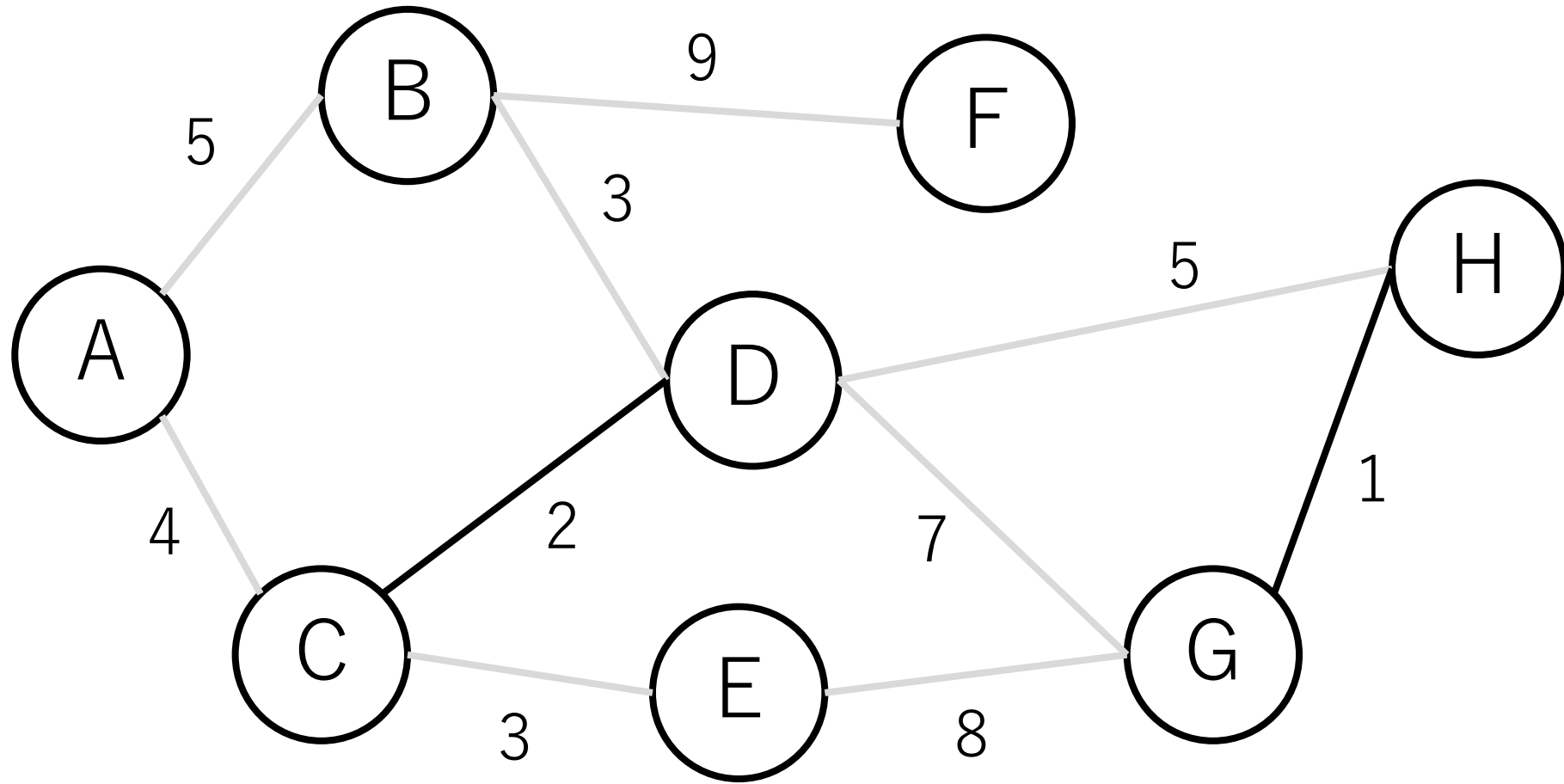
クラスカル法の例

一番距離の短い辺からスタート. 閉路にならないので入れる.



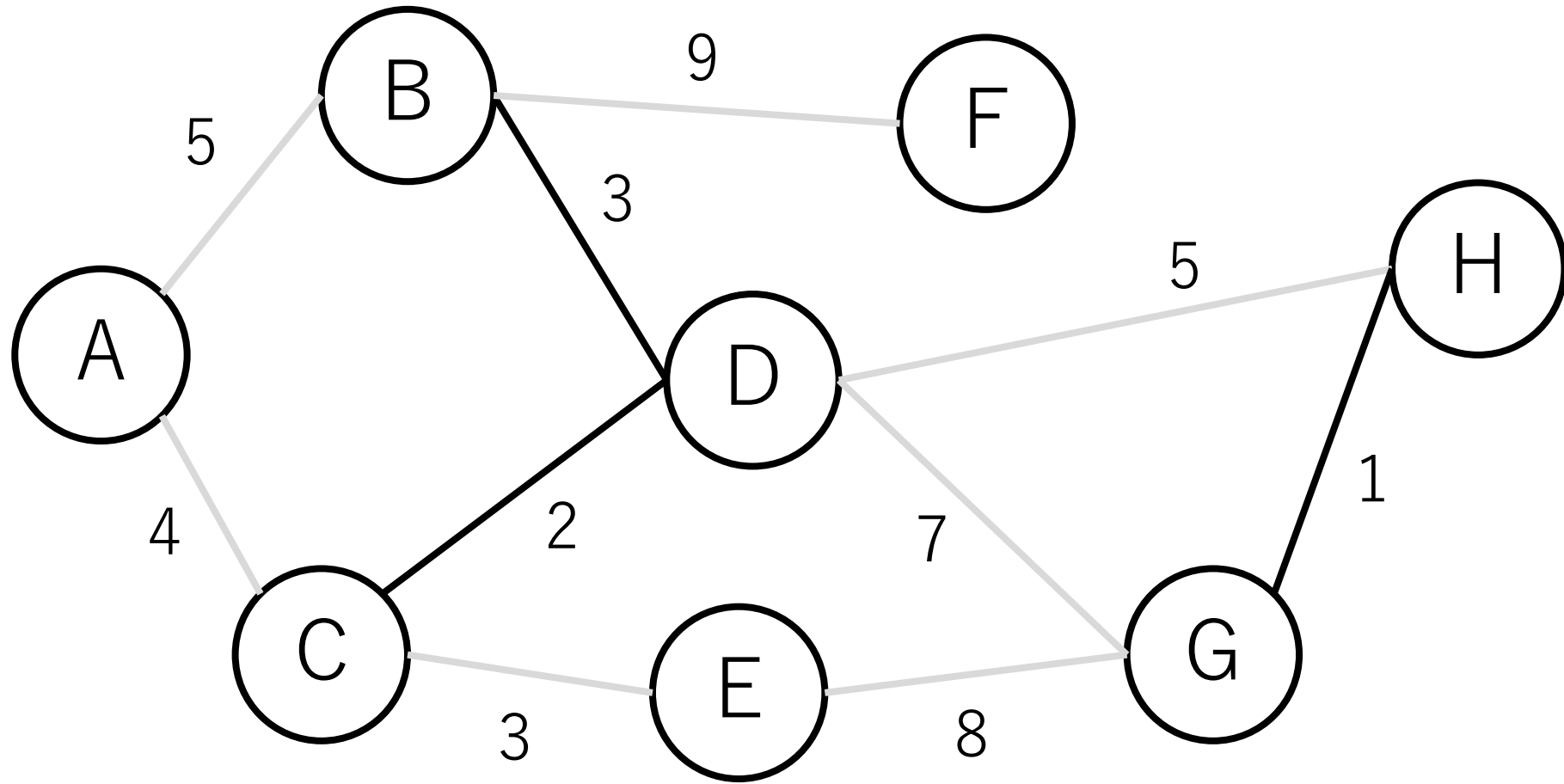
クラスカル法の例

C-Dも同じ.



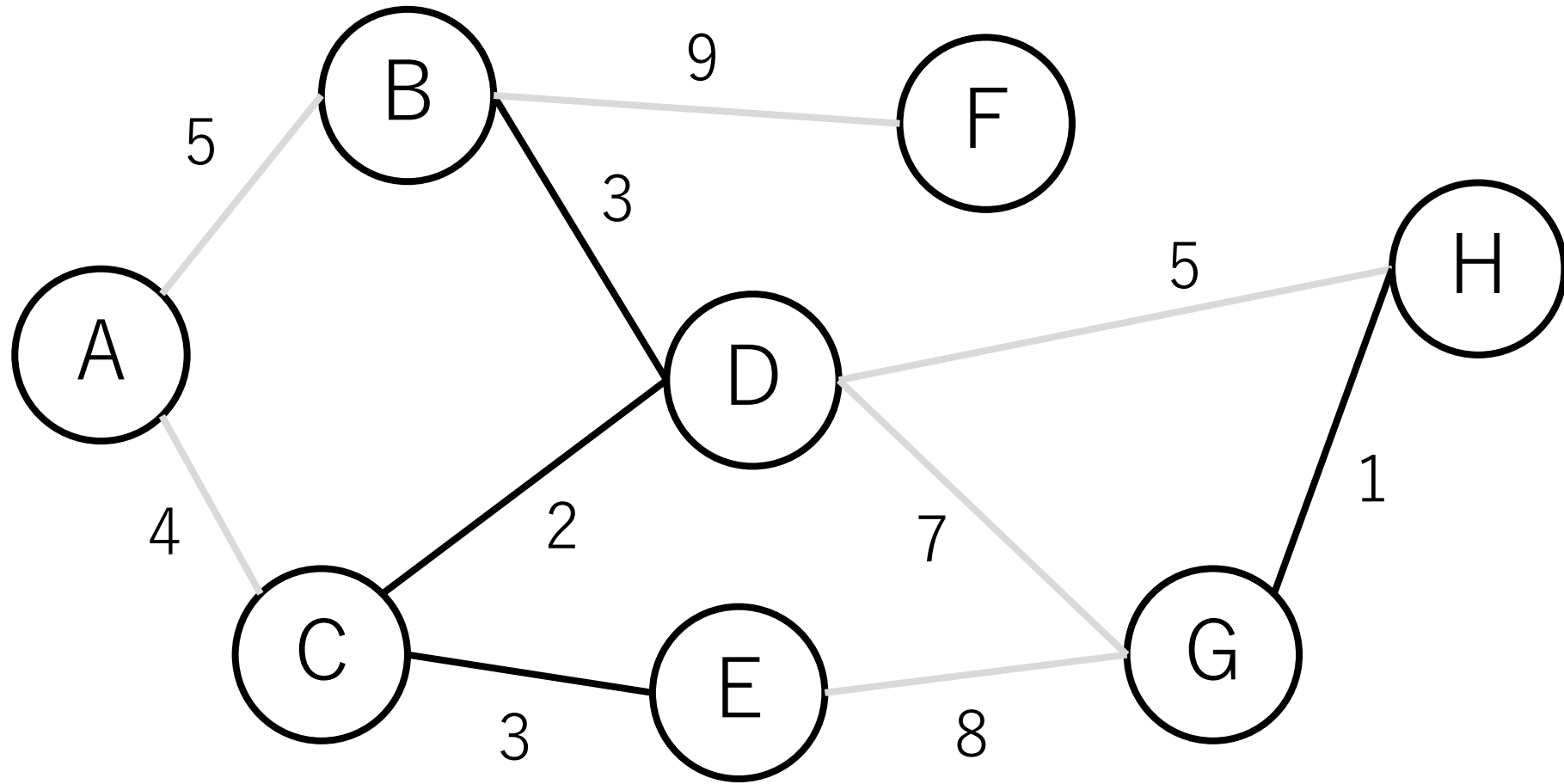
クラスカル法の例

B-Dも同じ.



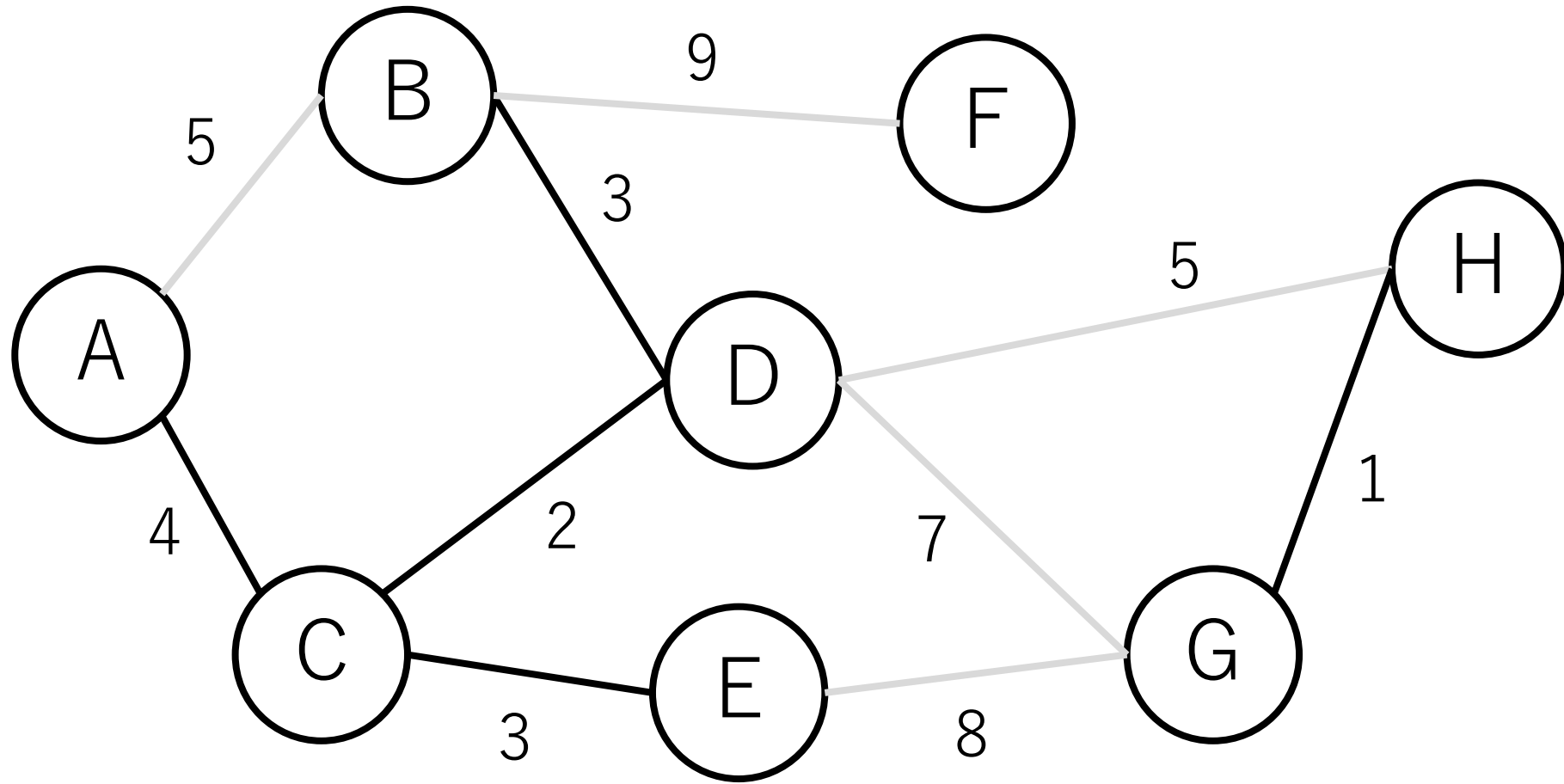
クラスカル法の例

C-Eも同じ.



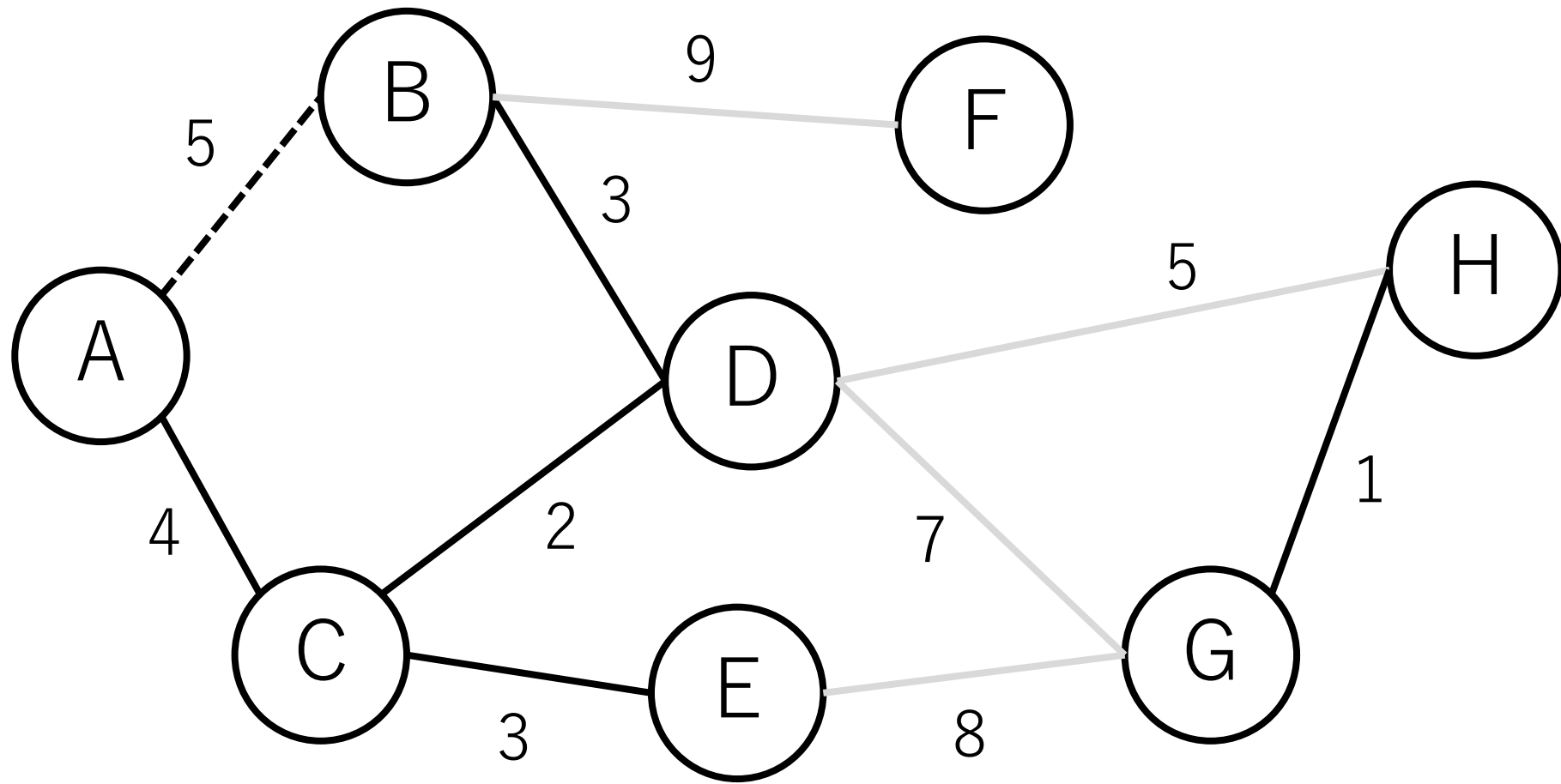
クラスカル法の例

A-Cも同じ.



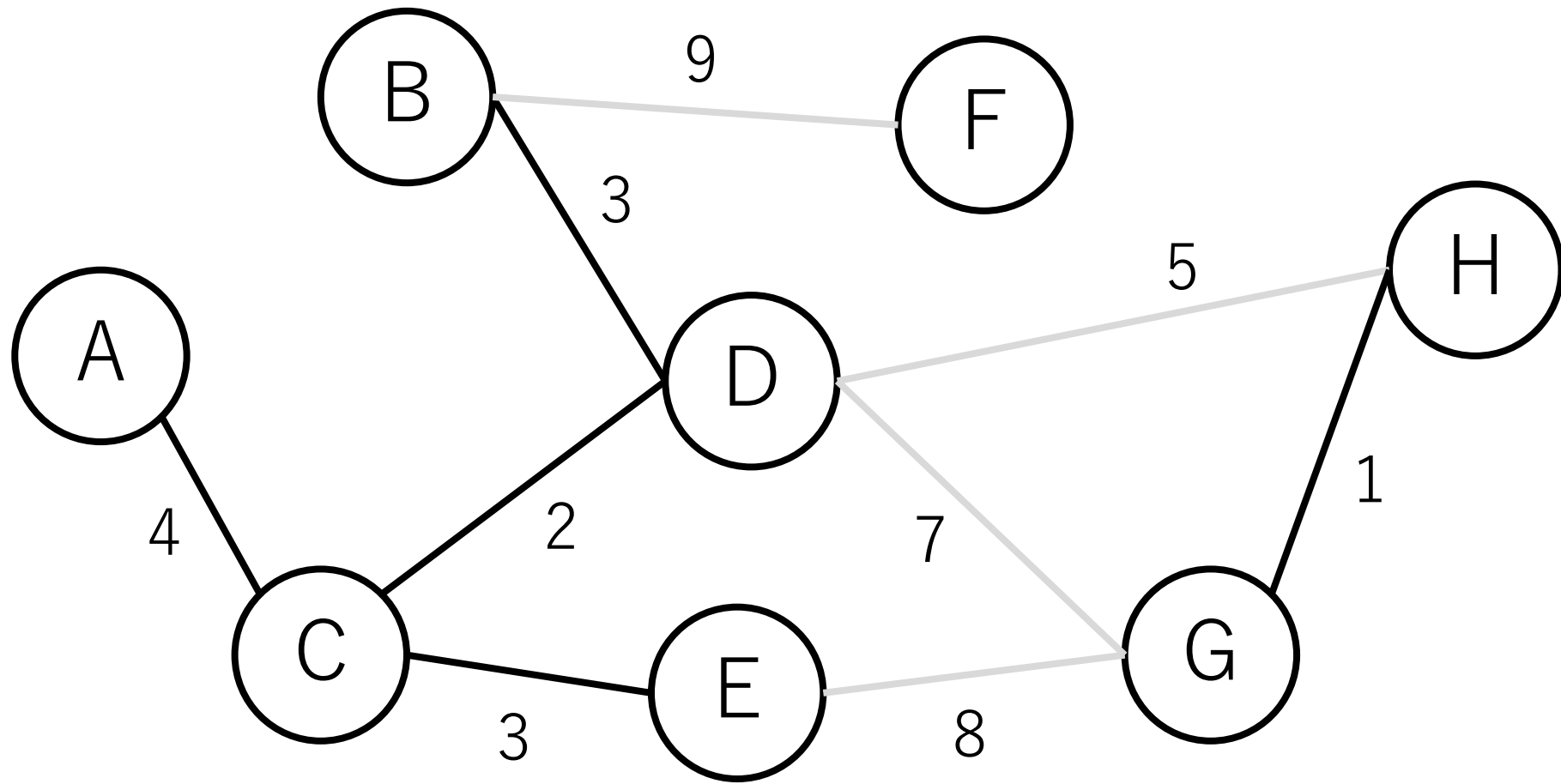
クラスカル法の例

A-Bをつないでしまうと閉路ができてしまう。



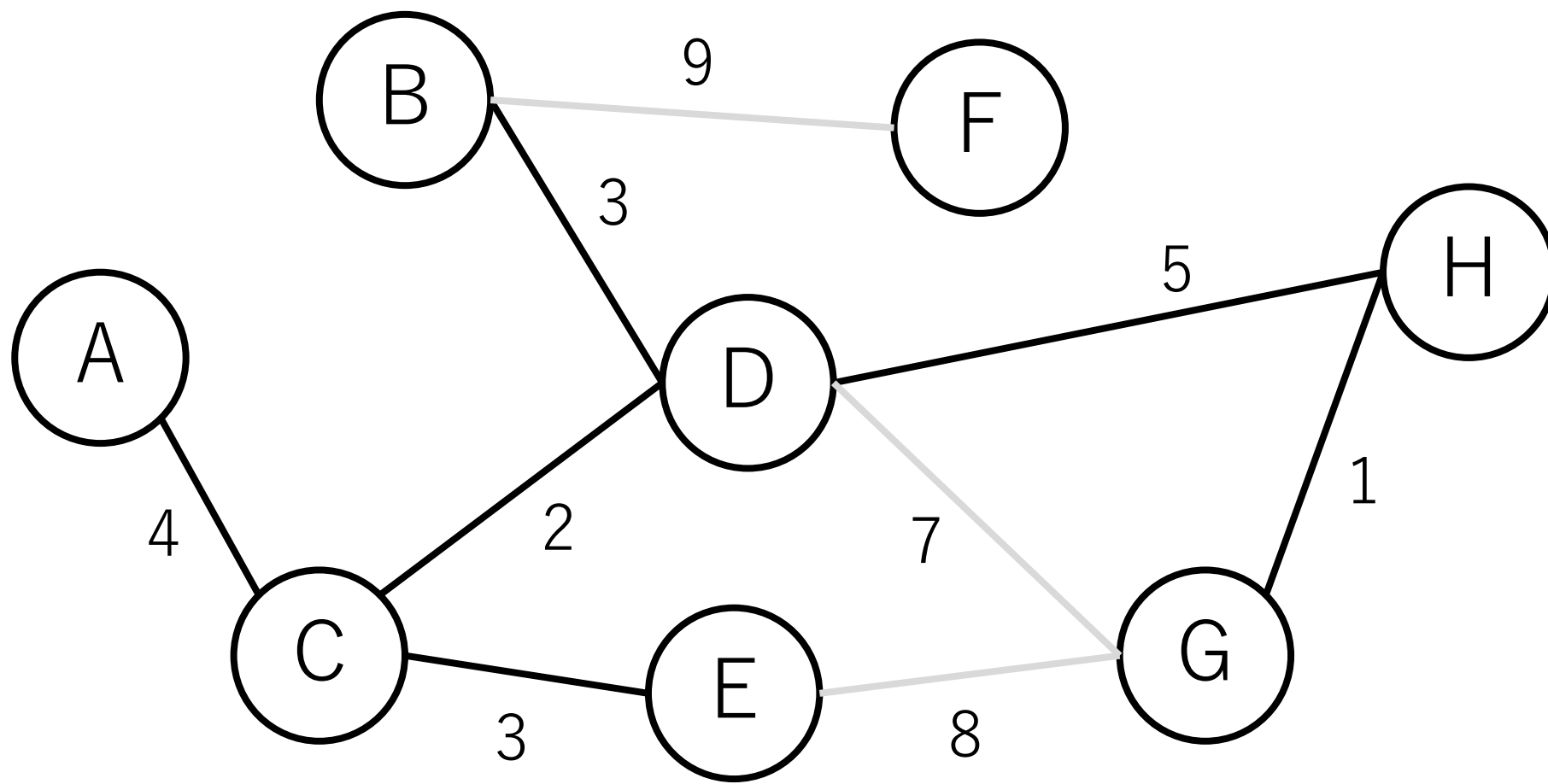
クラスカル法の例

よって、A-Bは入れない。



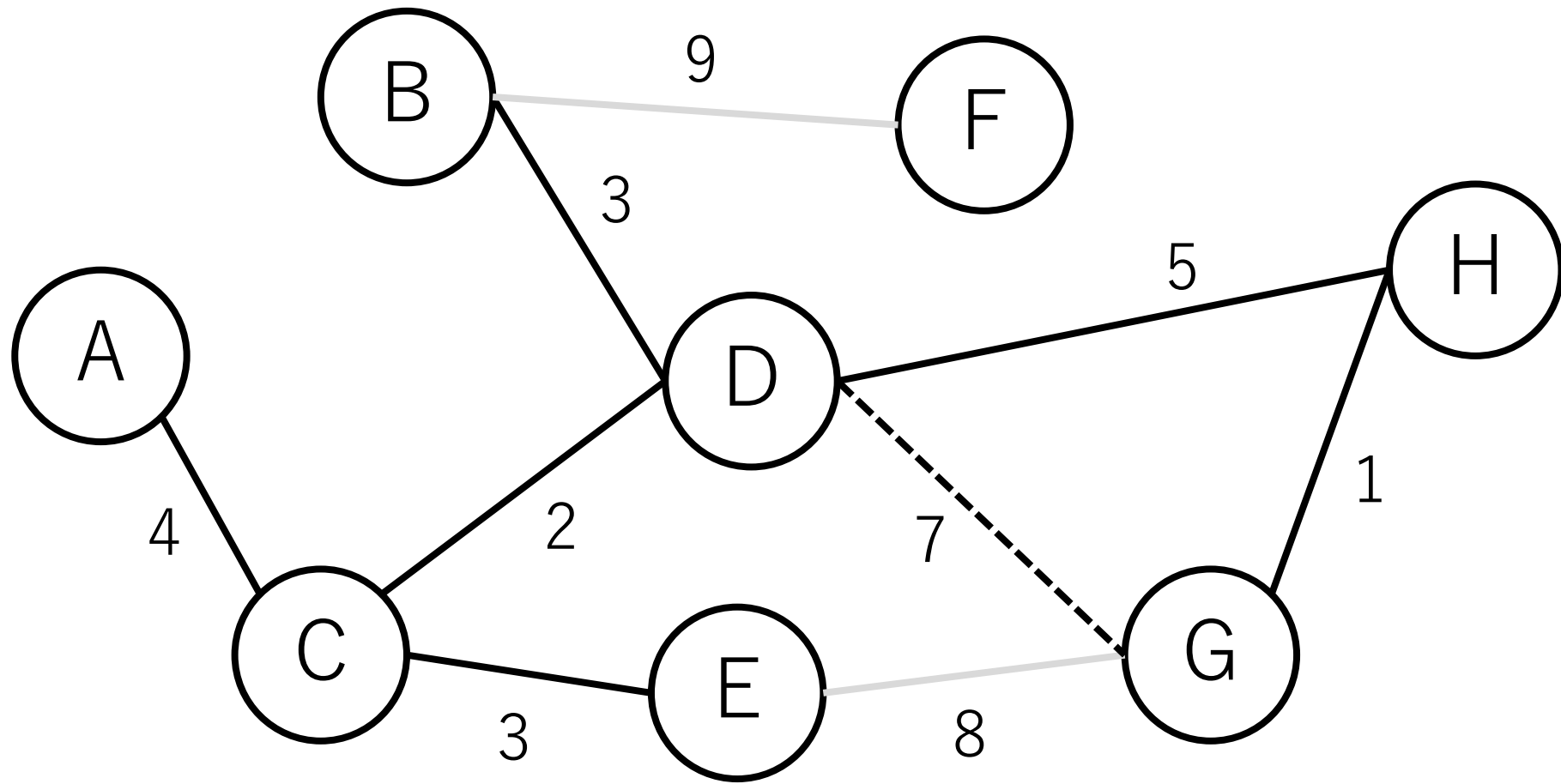
クラスカル法の例

D-Hは入れる。



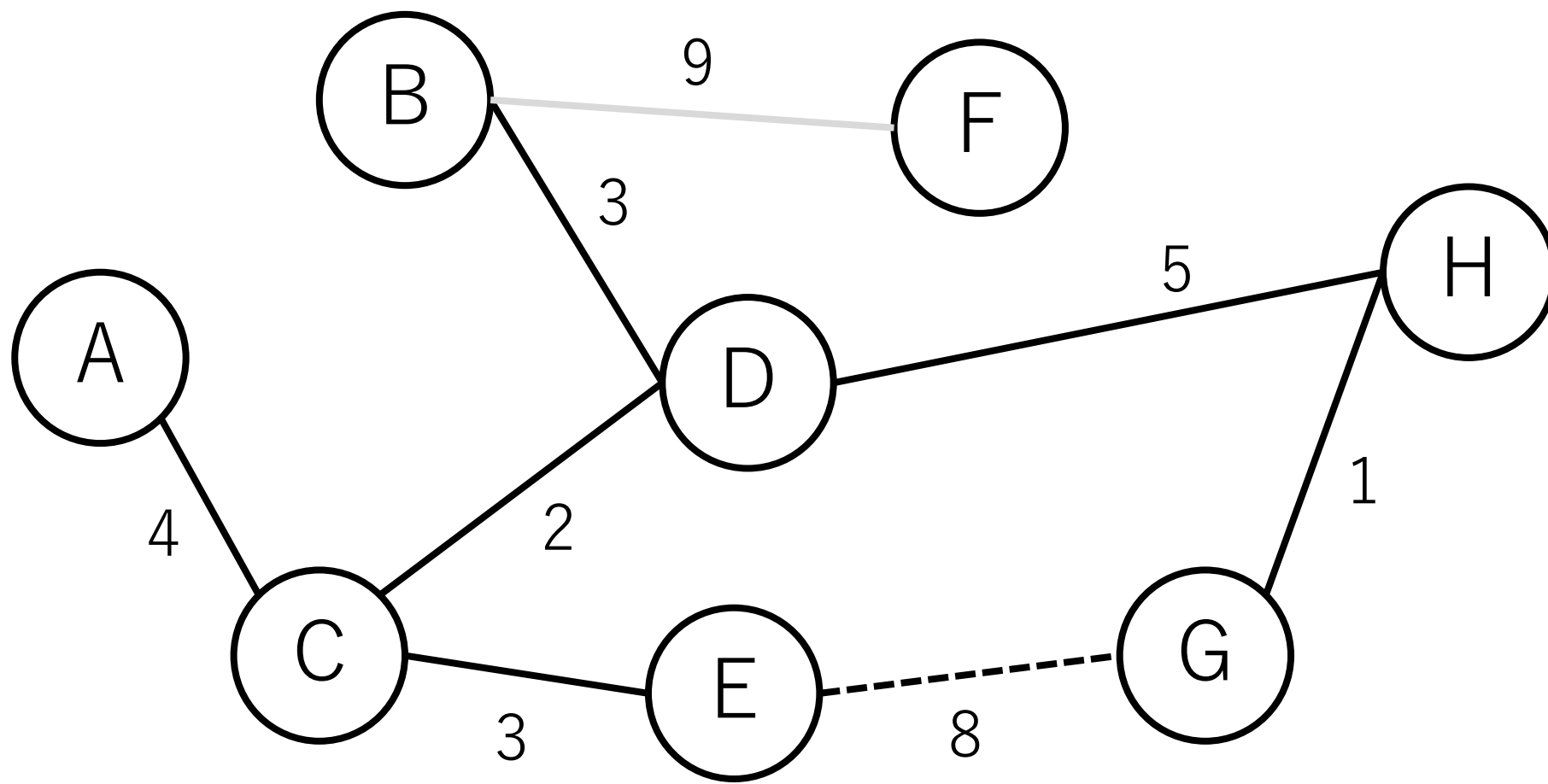
クラスカル法の例

D-Gは入れない。



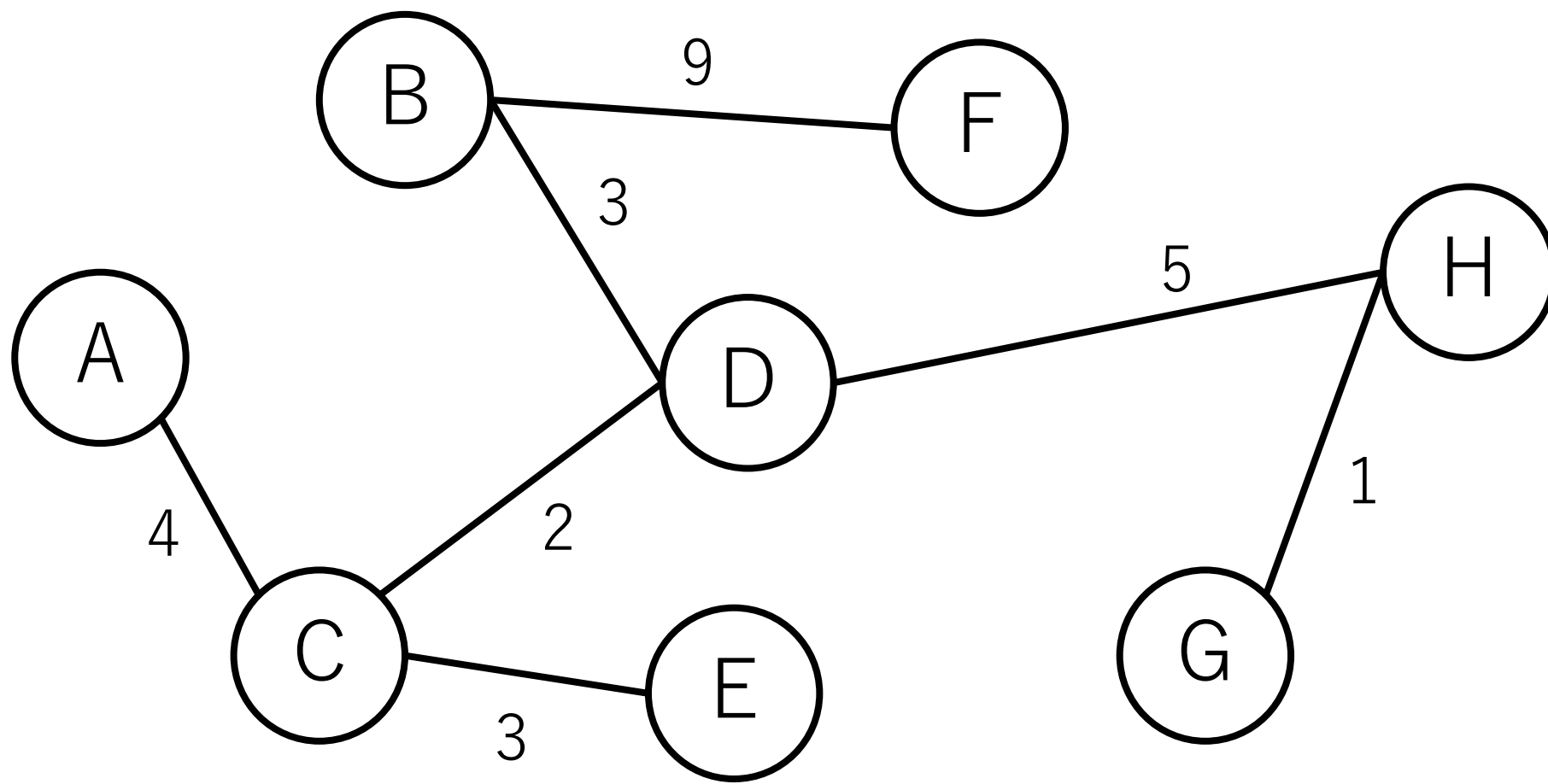
クラスカル法の例

E-Gは入れない。



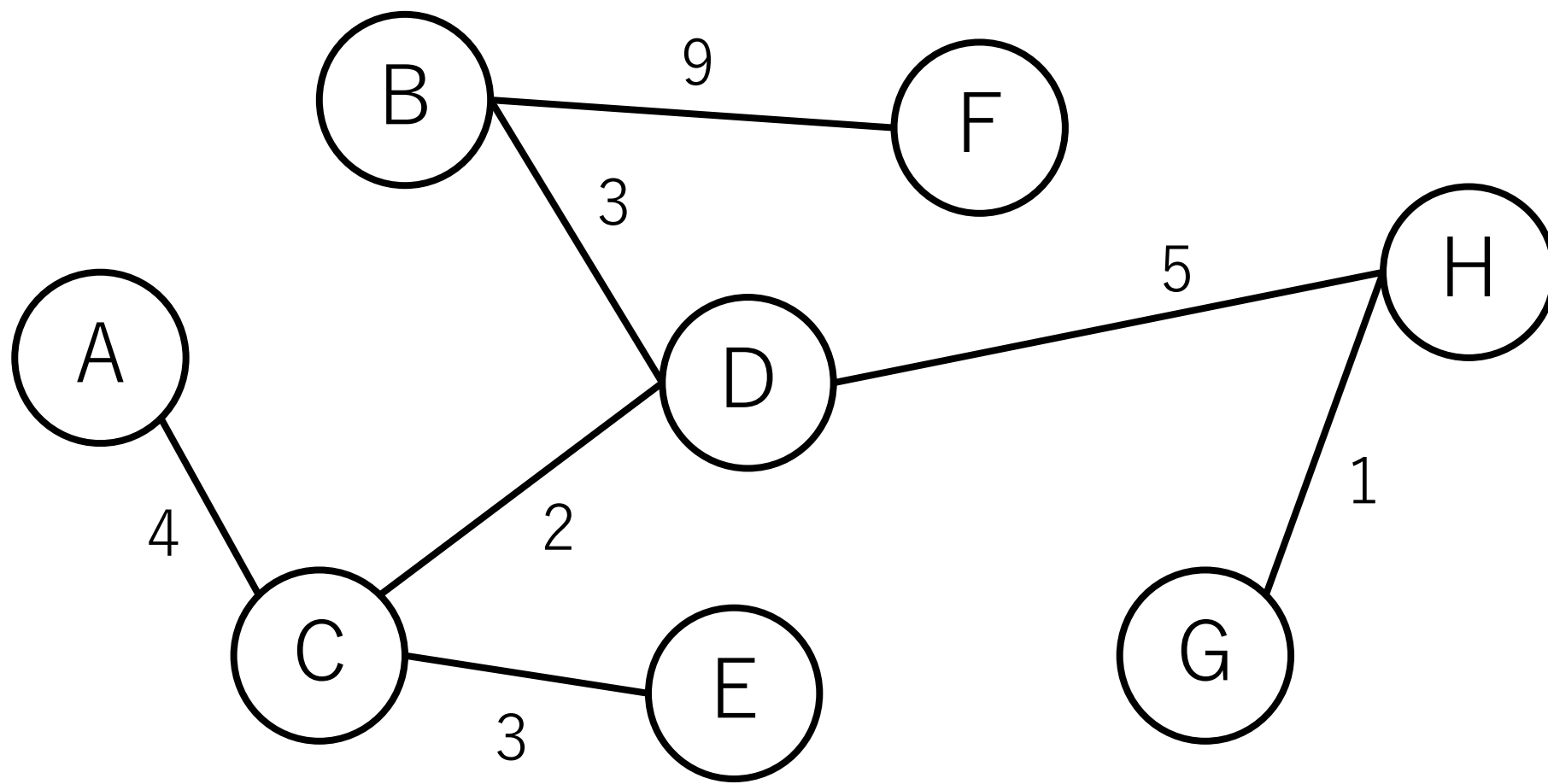
クラスカル法の例

B-Fは入れる。



クラスカル法の例

すべての辺が終わり，終了。



クラスカル法の実装

重要なポイントは2つある。

「存在する辺を距離の短い順に並べて順に入れていき」
これはソートすればよいだけ。

「閉路が出来ないことが確認できた場合は追加し」

これはどうやれば効率的に実現できる？

辺を足すごとに毎回グラフをたどるのは非現実的。

素集合データ構造 (Union-Find木)

要素を素集合 (互いに重ならない集合) に分割して管理するデータ構造. このデータ構造には2つの操作がある.

Union : 2つの集合をマージする.

Find : ある要素がどの集合にいるかを見つける.

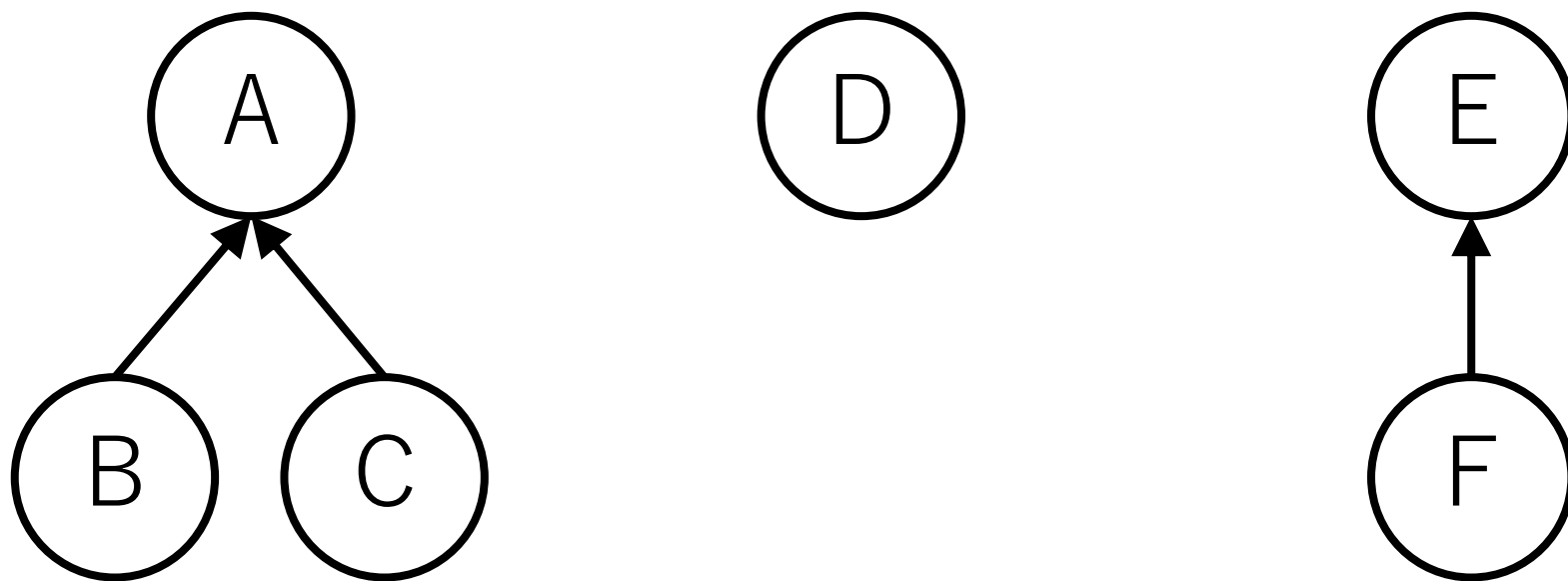
集合を元にroll backする操作はここでは扱わない.

興味のある方は, Undo可能Union-Find, 永続Union-Findとかチェックしてみてください.

Union-Find木の例

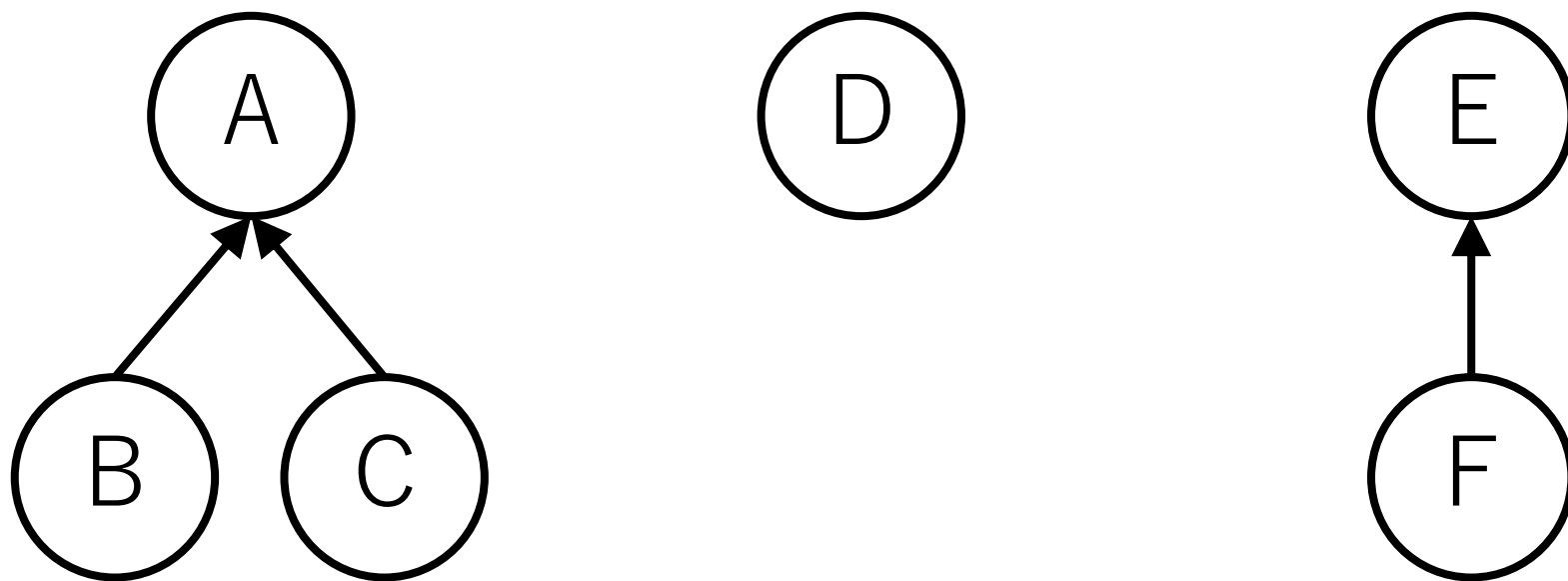
素集合

木が3つある（「森になっている」と表現することもある）。



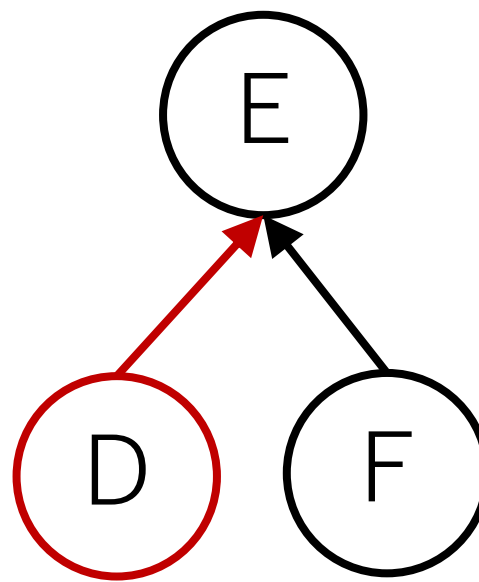
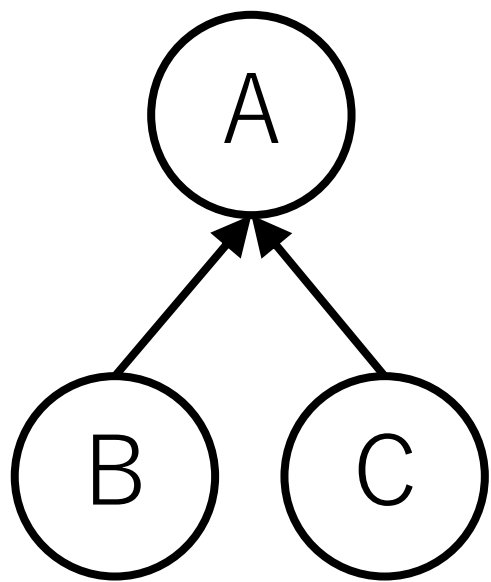
Union-Find木の例

Unite : 片方の根からもう片方の根につなぐ.



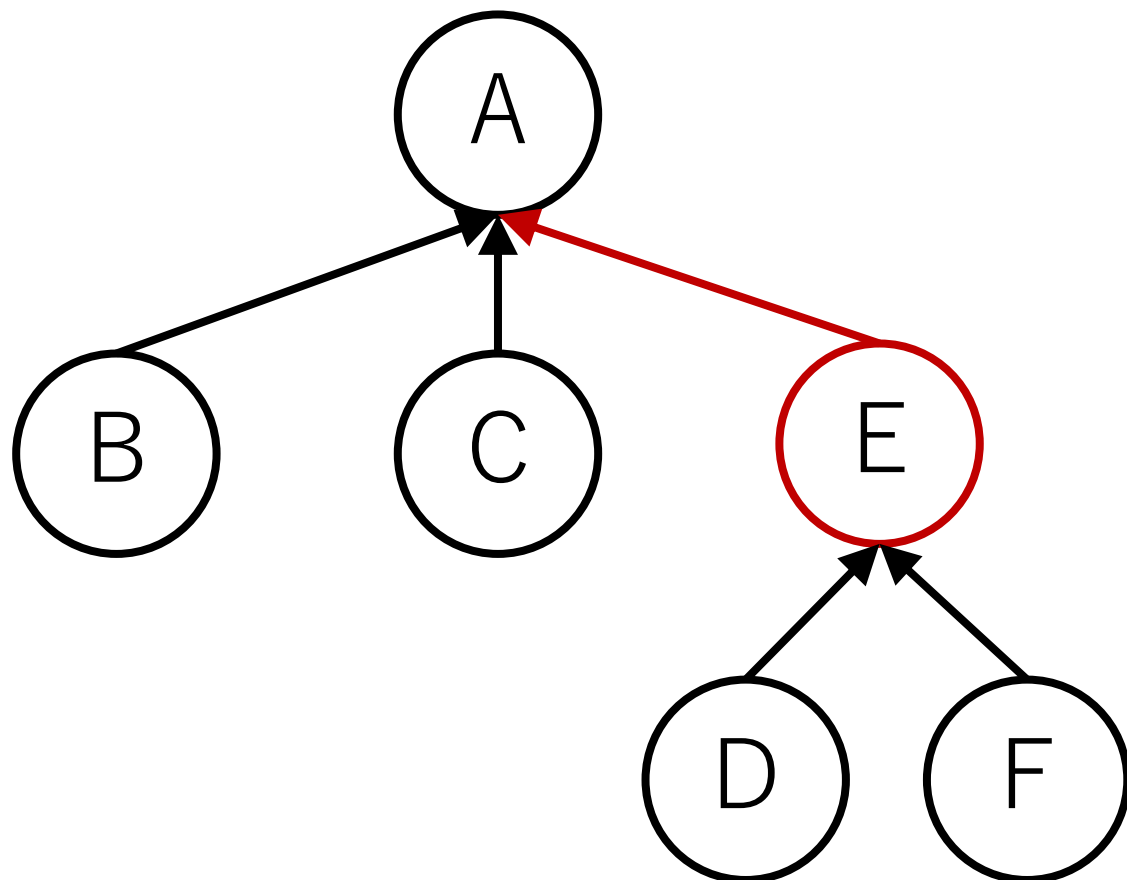
Union-Find木の例

Unite : 片方の根からもう片方の根につなぐ。
Dと[E, F]をマージ。



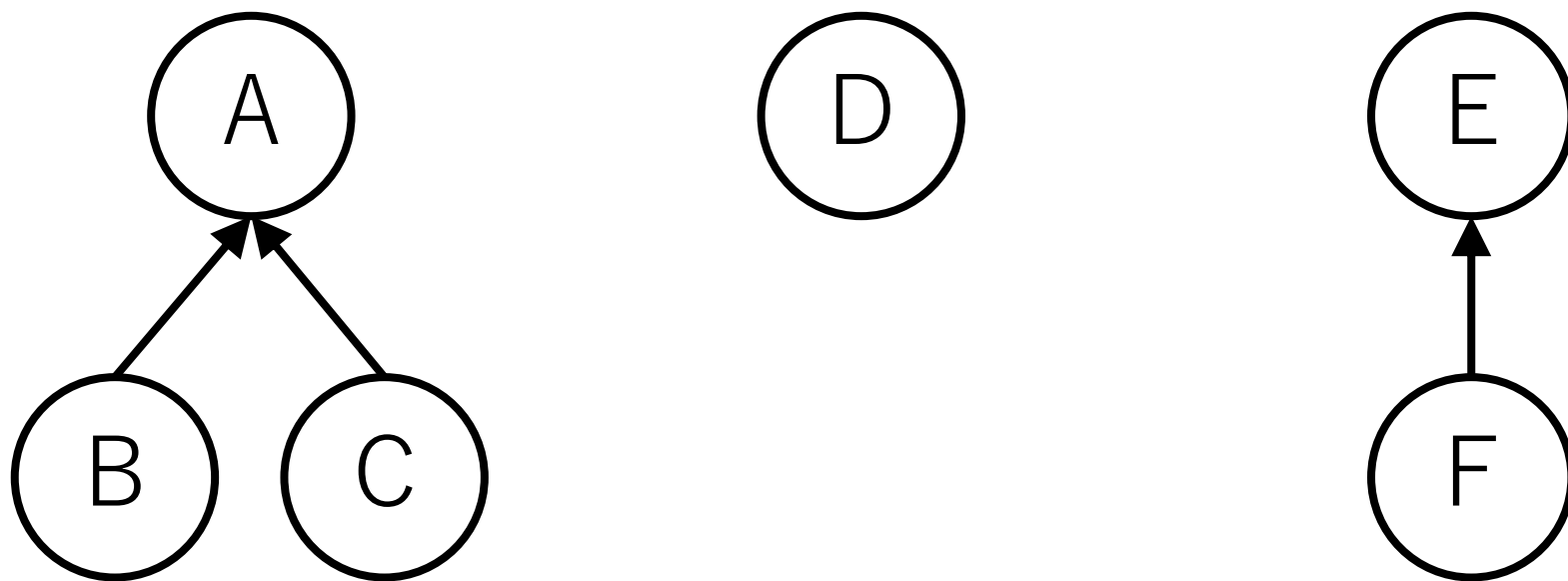
Union-Find木の例

Unite : 片方の根からもう片方の根につなぐ。
残り2つの集合をマージ。



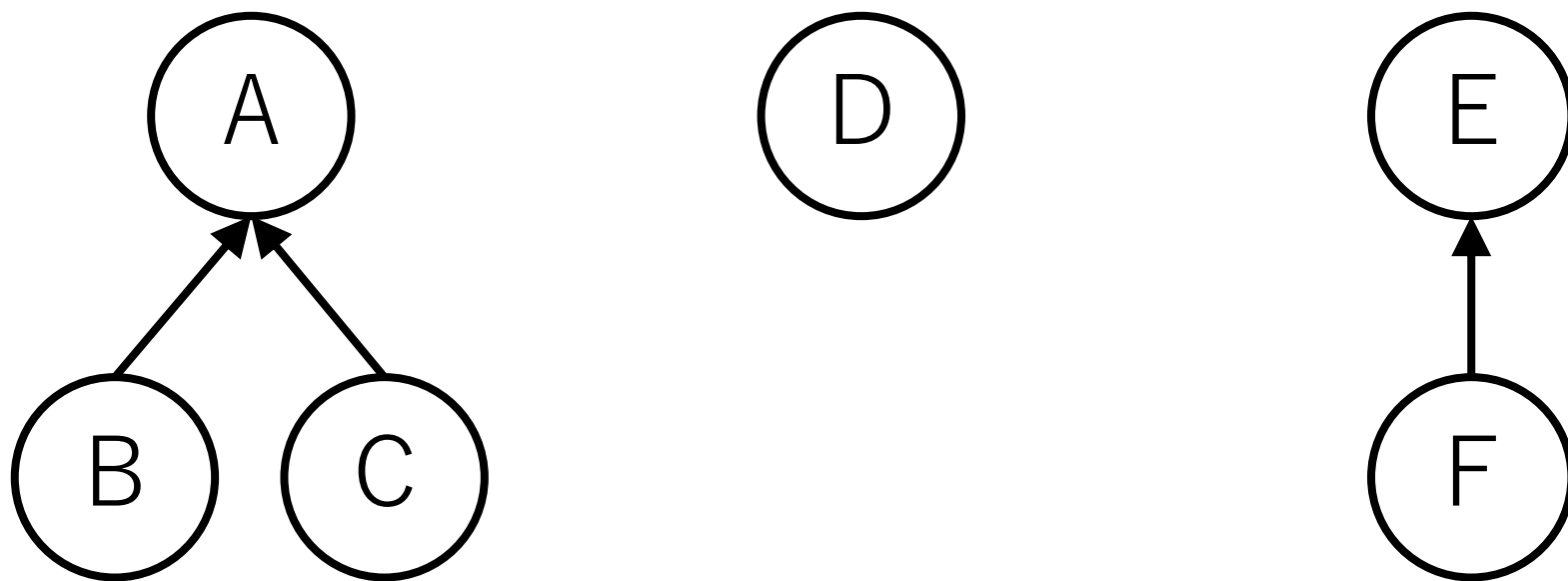
Union-Find木の例

Find : 「同じグループである」 = 「同じ根である」なので、
根ノードを返す。



Union-Find木の例

2つの要素が同じグループか：根ノードが同じかどうかをチェックする。



Union-Find木の実装

N個の要素がある時，長さNの配列を用意．

この配列には親ノードのindexを入れる．

自分が根ノードの場合は自分自身のindexを入れる．

この値をたどっていけば最終的に根ノードに行き着く．

最初の時点では自分自身しかグループに属していないので，自分自身が根ノードになる

Union-Find木の効率化

できる限り根ノードに速くたどり着けるように構造を更新する.

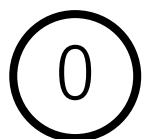
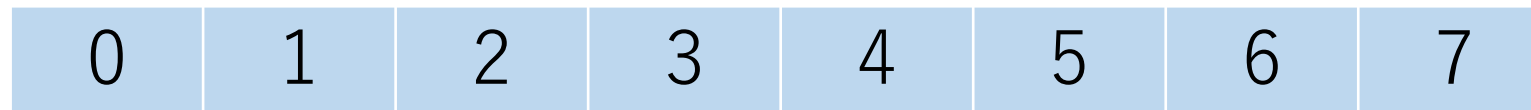
#1 Unite時に木の高さが高い方にマージ.

こうすることで、マージのときに出来る限り木を高くしない.

#2 根を調べたときに、直接根につながるようにつなぎ替える. (経路圧縮)

Union-Find木の例

初期化：



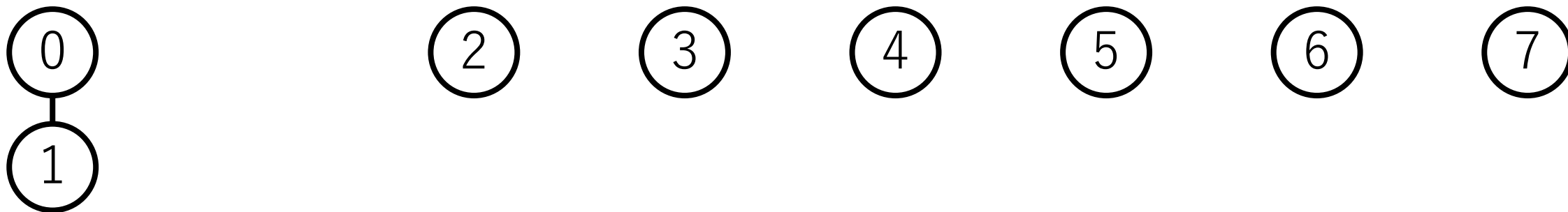
Union-Find木の例

初期化：

0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

0と1をunite：

0	0	2	3	4	5	6	7
---	----------	---	---	---	---	---	---



Union-Find木の例

初期化：

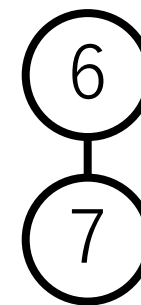
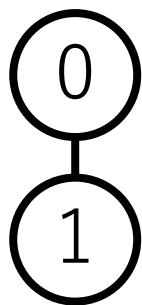
0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

0と1をunite：

0	0	2	3	4	5	6	7
---	---	---	---	---	---	---	---

6と7をunite：

0	0	2	3	4	5	6	6
---	---	---	---	---	---	---	----------



Union-Find木の例

初期化：

0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

0と1をunite：

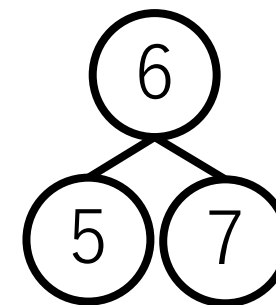
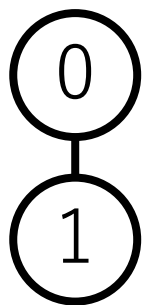
0	0	2	3	4	5	6	7
---	---	---	---	---	---	---	---

6と7をunite：

0	0	2	3	4	5	6	6
---	---	---	---	---	---	---	---

5と7をunite：

0	0	2	3	4	6	6	6
---	---	---	---	---	----------	---	---



Union-Find木の例

初期化：

0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

0と1をunite：

0	0	2	3	4	5	6	7
---	---	---	---	---	---	---	---

6と7をunite：

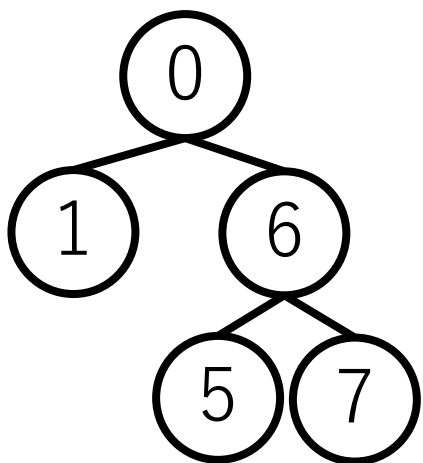
0	0	2	3	4	5	6	6
---	---	---	---	---	---	---	---

5と7をunite：

0	0	2	3	4	6	6	6
---	---	---	---	---	---	---	---

1と7をunite：

0	0	2	3	4	6	0	6
---	---	---	---	---	---	----------	---



Union-Find木の例

初期化 :

0と1をunite :

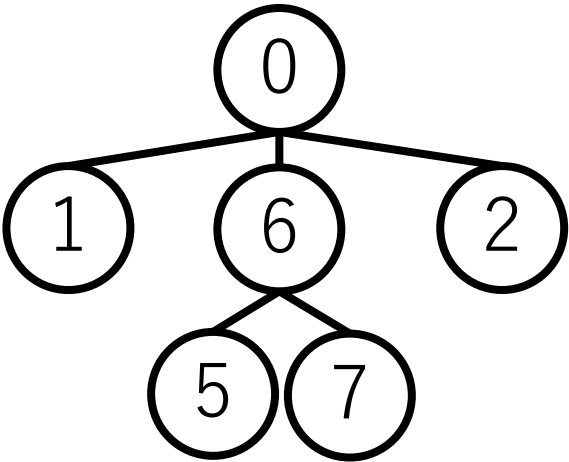
6と7をunite :

5と7をunite :

1と7をunite :

2と5をunite :

0	1	2	3	4	5	6	7
0	0	2	3	4	5	6	7
0	0	2	3	4	5	6	6
0	0	2	3	4	6	6	6
0	0	2	3	4	6	0	6
0	0	0	3	4	6	0	6



Union-Find木の例

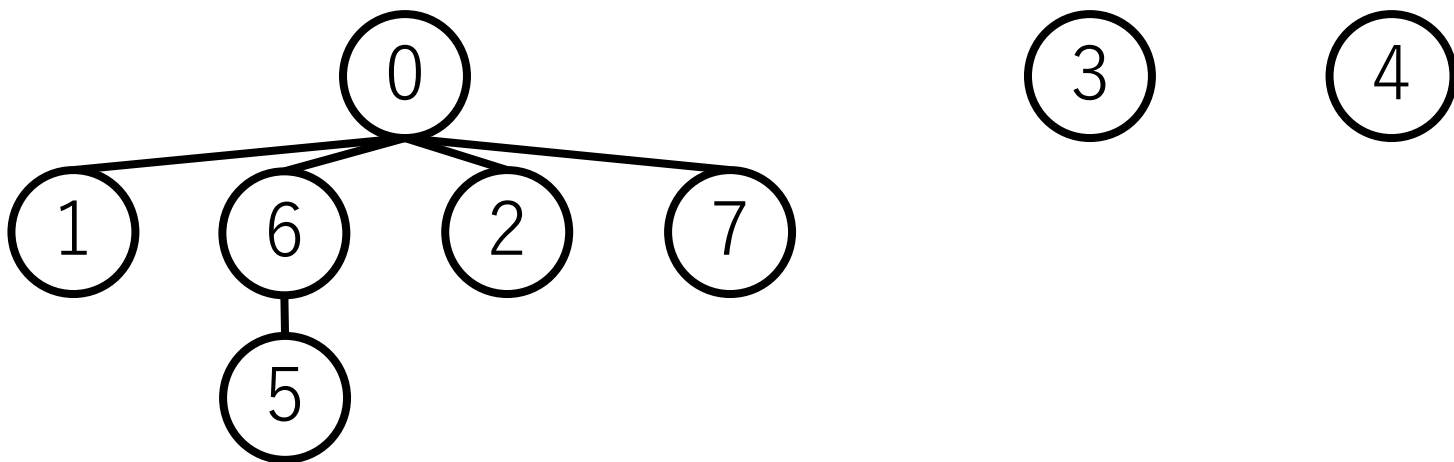
現時点：

0	0	0	3	4	6	0	6
---	---	---	---	---	---	---	---

7の根ノード or 属するグループをチェック

チェック完了後：

0	0	0	3	4	6	0	0
---	---	---	---	---	---	---	----------



Union-Find木の実装例

```
class UnionFind:  
    def __init__(self, n):  
        self.parent = [i for i in range(n)]  
        self.height = [0 for _ in range(n)] # 各木の高さ
```

Union-Find木の実装例

```
def get_root(self, i):  
    if self.parent[i] == i: # 自分が根ノードの場合  
        return i  
    else: # 経路圧縮しながら根ノードを探す  
        self.parent[i] = self.get_root(self.parent[i])  
        return self.parent[i]
```

Union-Find木の実装例

```
def unite(self, i, j):
    root_i = self.get_root(i)
    root_j = self.get_root(j)
    if root_i != root_j:    # より高い方にマージ
        if self.height[root_i] < self.height[root_j]:
            self.parent[root_i] = root_j
        else:
            self.parent[root_j] = root_i
            if self.height[root_i] == self.height[root_j]:
                self.height[root_i] += 1
```

Union-Find木の実装例

```
def is_in_group(self, i, j):  
    if self.get_root(i) == self.get_root(j):  
        return True  
    else:  
        return False
```


Union-Find木の計算量

正確にはアッカーマン関数 $A(n, n)$ の逆関数 $\alpha(n)$ になる.

$$A(0, 0) = 1, A(1, 1) = 3, A(2, 2) = 7, A(3, 3) = 61,$$

$$A(4, 4) = 2^{2^{2^{65536}}} - 3 \text{となる.}$$

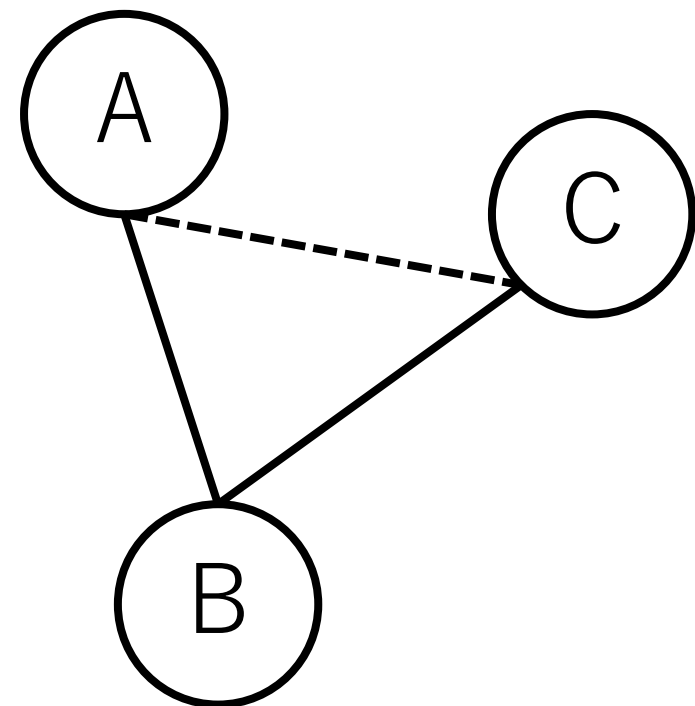
これはlogよりもさらに増加しない関数であり, 定数倍とみなして扱われることもある.

Union-Find木による閉路の判定

ある辺が与えられた時，その2つのノードが同じグループに属している場合，その辺を加えると閉路が生まれることになる。

よって，2つのノードが同じグループに属していないことをチェックすれば良い。

→Union-Find木なら速攻できる！



クラスカル法の実装例

```
# 引数：ノードの総数, 隣接リスト
```

```
def kruskal(V, e_list):
```

```
    e_cost_sorted = [] # 距離で整列された辺
```

```
    # ソートのために先頭の要素を距離にする
```

```
    for e in e_list:
```

```
        e_cost_sorted.append([e[2], e[0], e[1]])
```

```
    e_cost_sorted.sort()
```

クラスカル法の実装例

```
def kruskal(V, e_list):
```

```
    ...
```

```
    # Union-Find木を使う
```

```
    uf_tree = UnionFind(V)
```

```
    # 最小全域木の辺を保持するリスト
```

```
    mst = []
```

クラスカル法の実装例

```
def kruskal(V, e_list):
```

```
    ...
```

```
    [距離の小さい辺から順に全部見ていく]:
```

```
        [e[1], e[2]が同じグループでないならば]:
```

```
            [e[1], e[2]を同じグループにする]
```

```
            # 最小全域木に追加
```

```
            mst.append([e[1], e[2]])
```

クラスカル法の実装例

```
def kruskal(V, e_list):
```

```
    ...
```

```
    # ソートして表示
```

```
    mst.sort()
```

```
    print(mst)
```

クラスカル法の実行例

```
edges_list = [[0, 1, 5], [0, 2, 4], [1, 0, 5], [1, 3, 3], [1, 5, 9],  
[2, 0, 4], [2, 3, 2], [2, 4, 3], [3, 1, 3], [3, 2, 2], [3, 6, 7],  
[3, 7, 5], [4, 2, 3], [4, 6, 8], [5, 1, 9], [6, 3, 7], [6, 4, 8],  
[6, 7, 1], [7, 3, 5], [7, 6, 1]]
```

```
kruskal(8, edges_list)
```

=== 実行結果 ===

```
[[0, 2], [1, 3], [1, 5], [2, 3], [2, 4], [3, 7], [6, 7]]
```

クラスカル法の計算量

隣接リストの場合，辺の数を $|E|$ として，辺のソートに $O(|E| \log |E|)$ かかる．

隣接行列の場合はすべての辺を取り出すために追加で $O(|V|^2)$ かかる．（ $|V|$ はノードの数）

各辺を入れるかどうかの判断はUnion-Find木を使うと， $O(\alpha(|V|))$ となり，これを $O(|E|)$ 回やるので， $O(|E| \alpha(|V|))$ ．

よって，アルゴリズム全体では $O(|E| \log |E|)$ ．

最小全域木のアルゴリズム

辺ベースのアプローチ：クラスカル法

存在する辺を距離の短い順に並べて順に入れていき、閉路が出来ないことが確認できた場合は追加し、全部の辺をチェックしたら終了。

ノードベースのアプローチ：プリム法

すでに到達した頂点の集合からまだ到達していない頂点の集合への辺のうち、距離が最短のものを追加し、全ノードつながったら終了。

プリム法 (Prim)

- #1 最初のノードを1つ選び（どれでも可），訪問済にする．
- #2 そのノードに繋がっている全ての辺を取り，最小全域木の候補の辺に入れる．

プリム法 (Prim)

#3 最小全域木の候補の辺の中から，接続先のノードが未訪問である最短の距離の辺を選ぶ。（接続先のノードが訪問済の場合は無視して次候補に移る。）

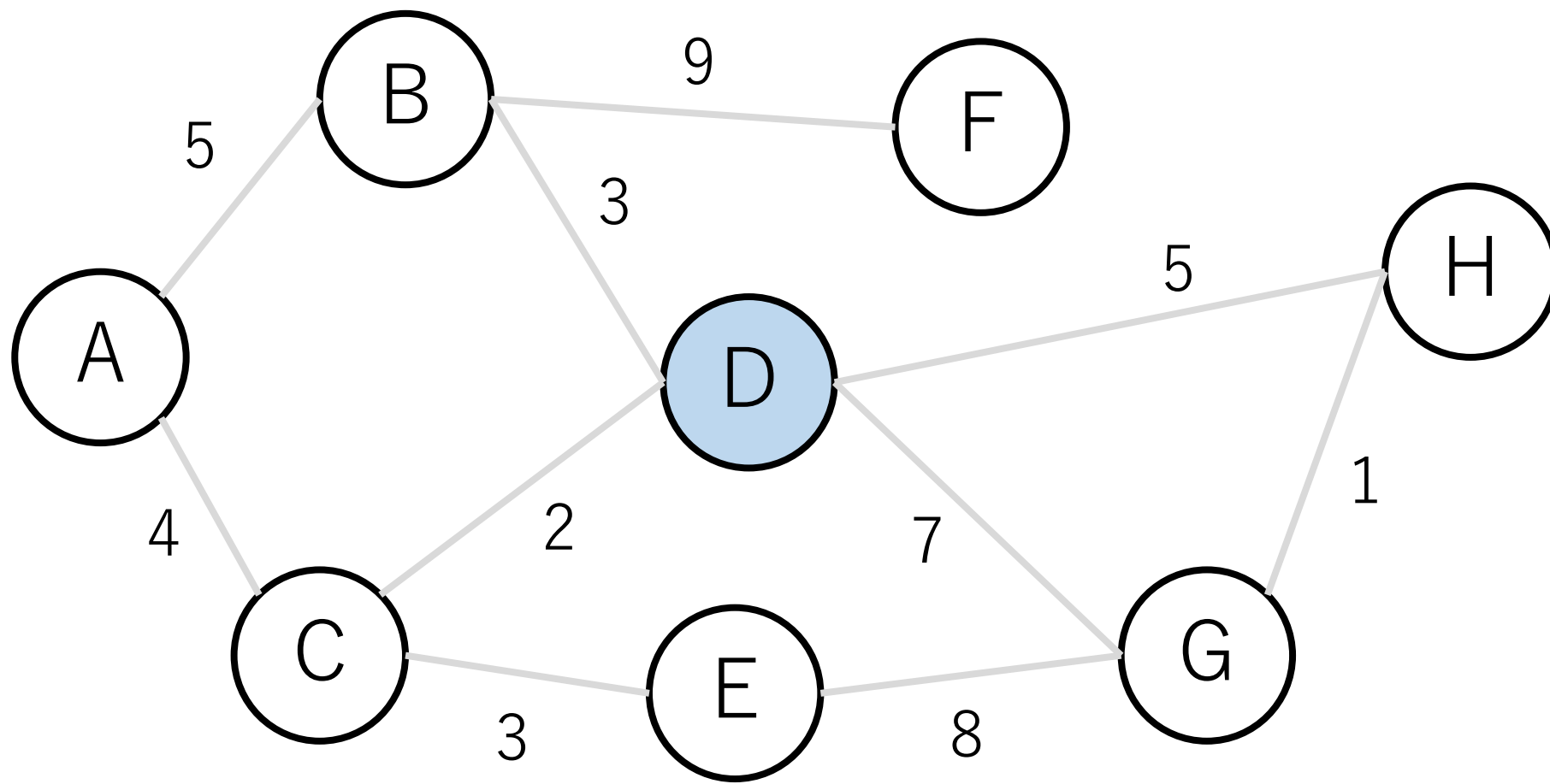
#4 選んだ辺を最小全域木に入れ，その接続先にあるノードを訪問済にする。

#5 #4で新しく訪問したノードから，更にその先につながっている辺のうち，接続先のノードが未訪問の全ての辺を最小全域木の候補に入れる。

#6 以降，全ノードが訪問済になるまで#2～#4を繰り返す。

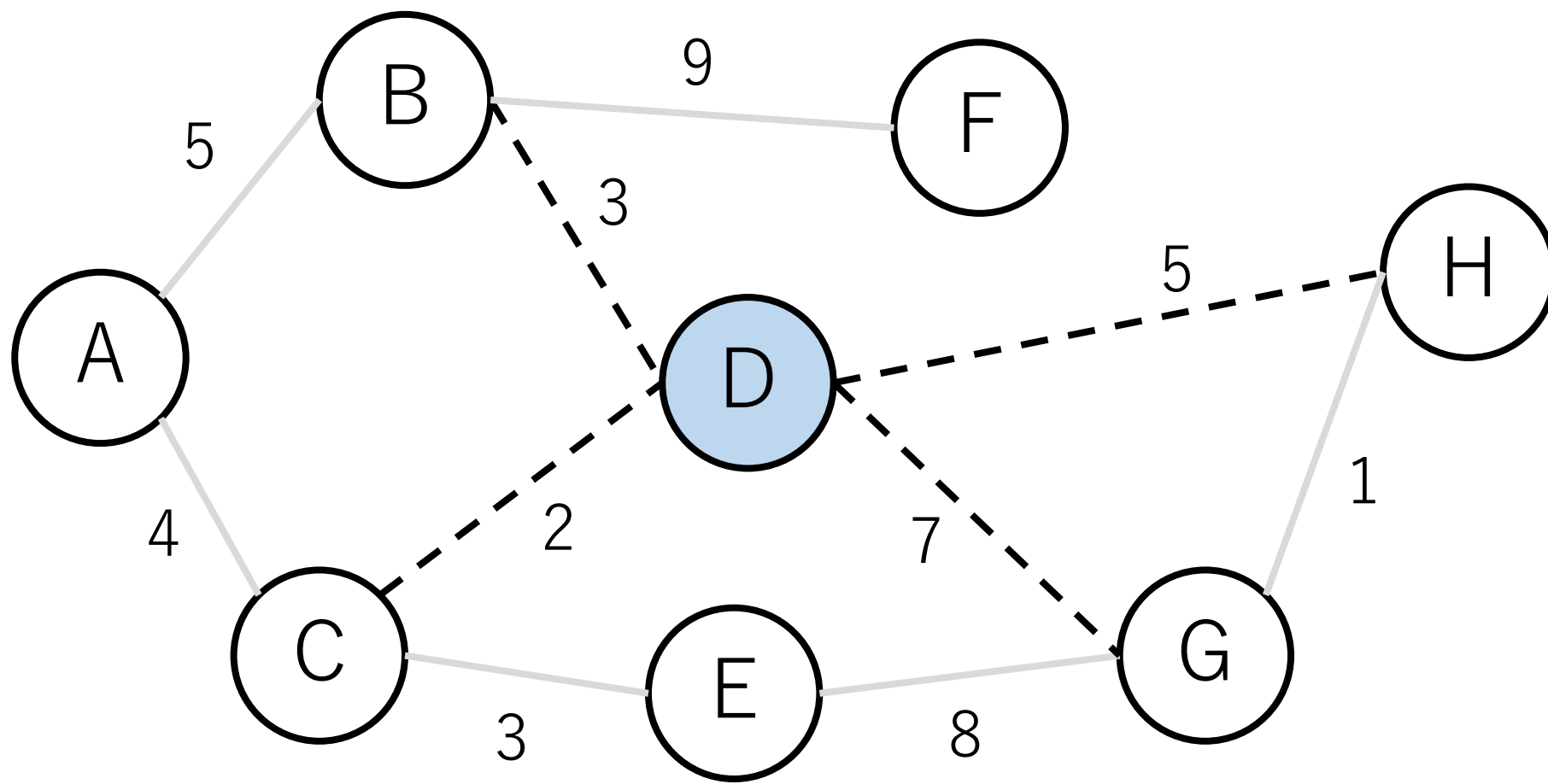
プリム法の例

Dからスタート. (どのノードから始めても良い)



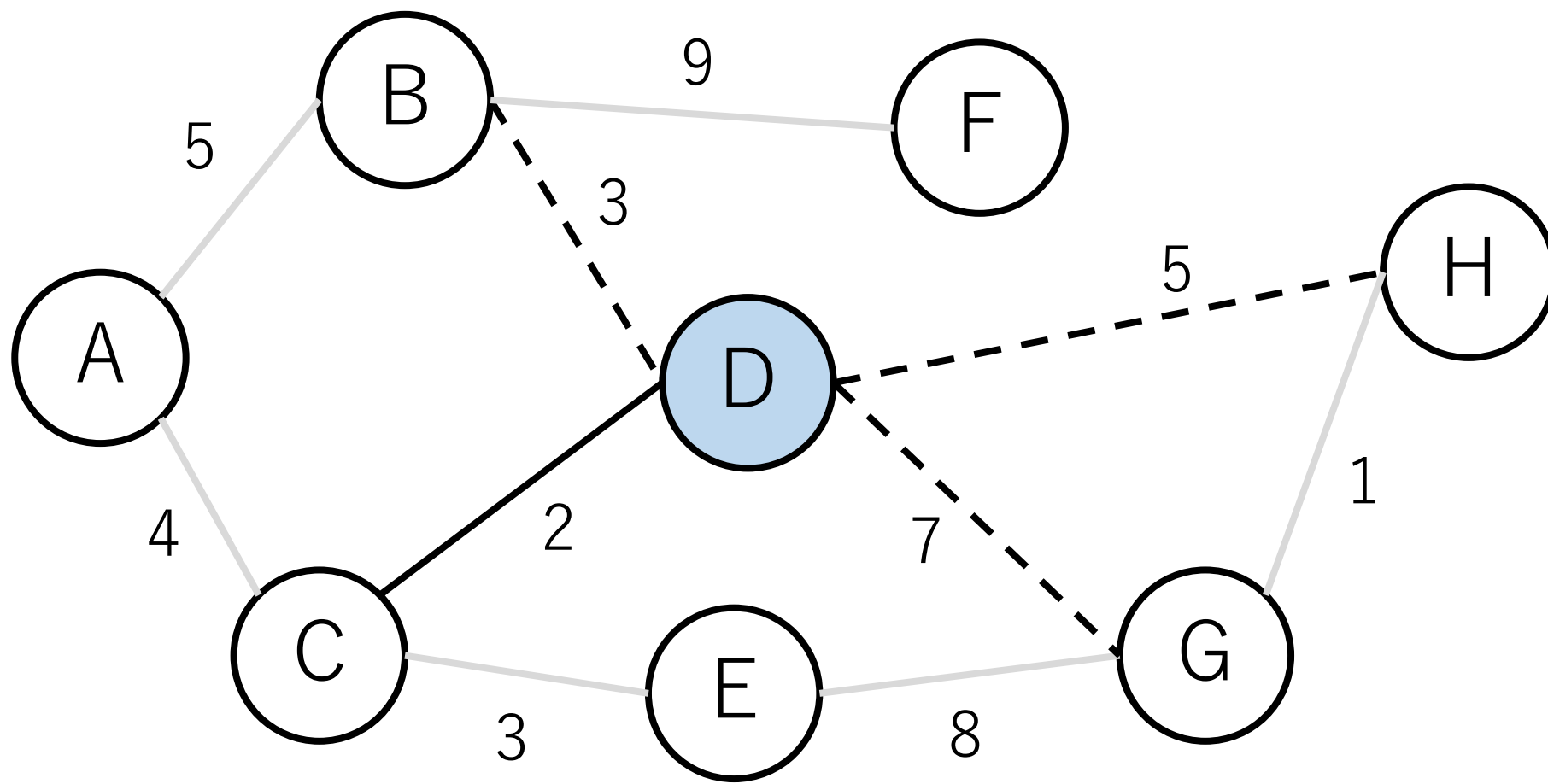
プリム法の例

最小全域木の候補の辺は点線で示すもの。



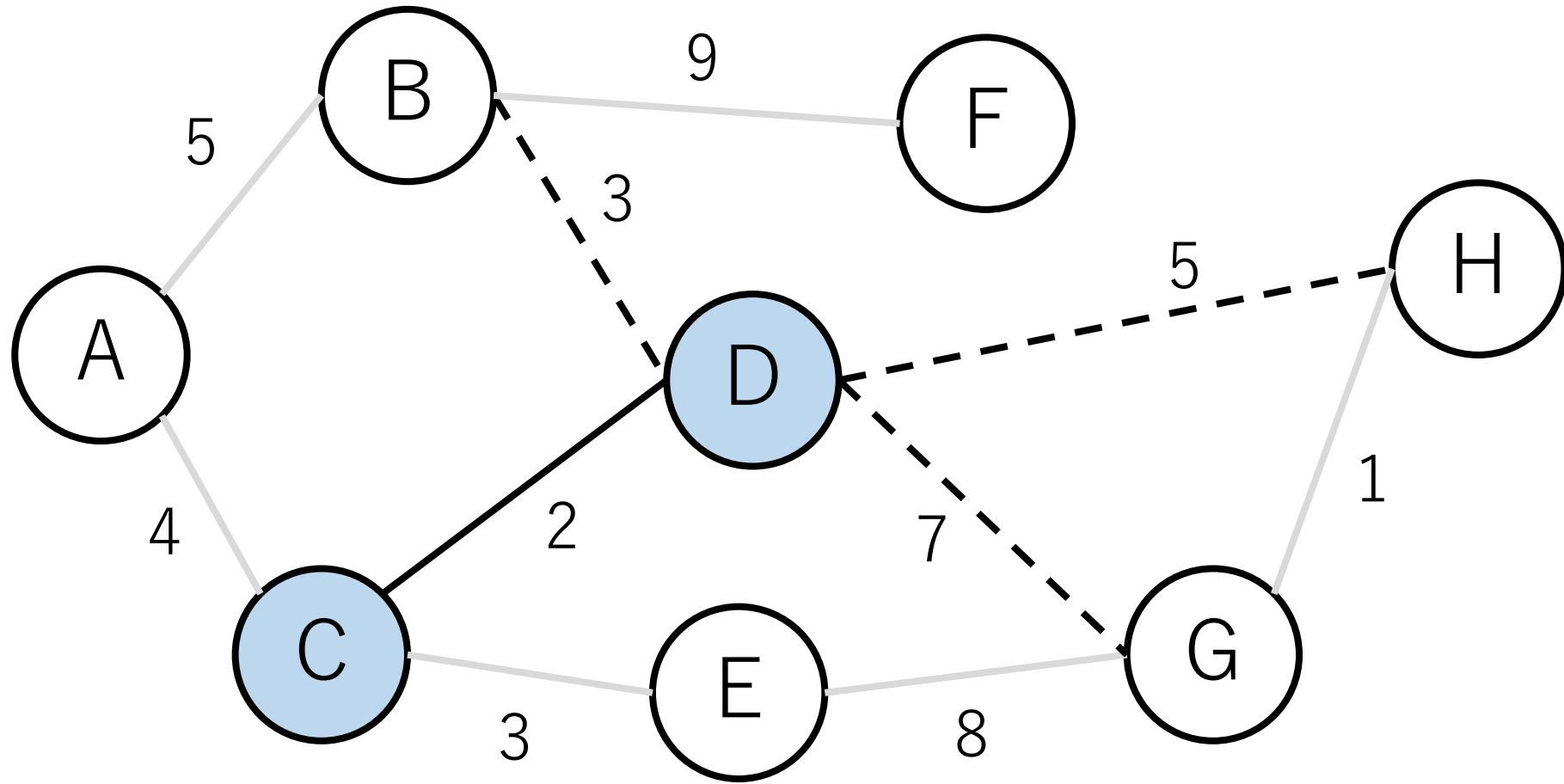
プリム法の例

C-Dの辺が最短なので、この辺を入れてCとつなぐ。



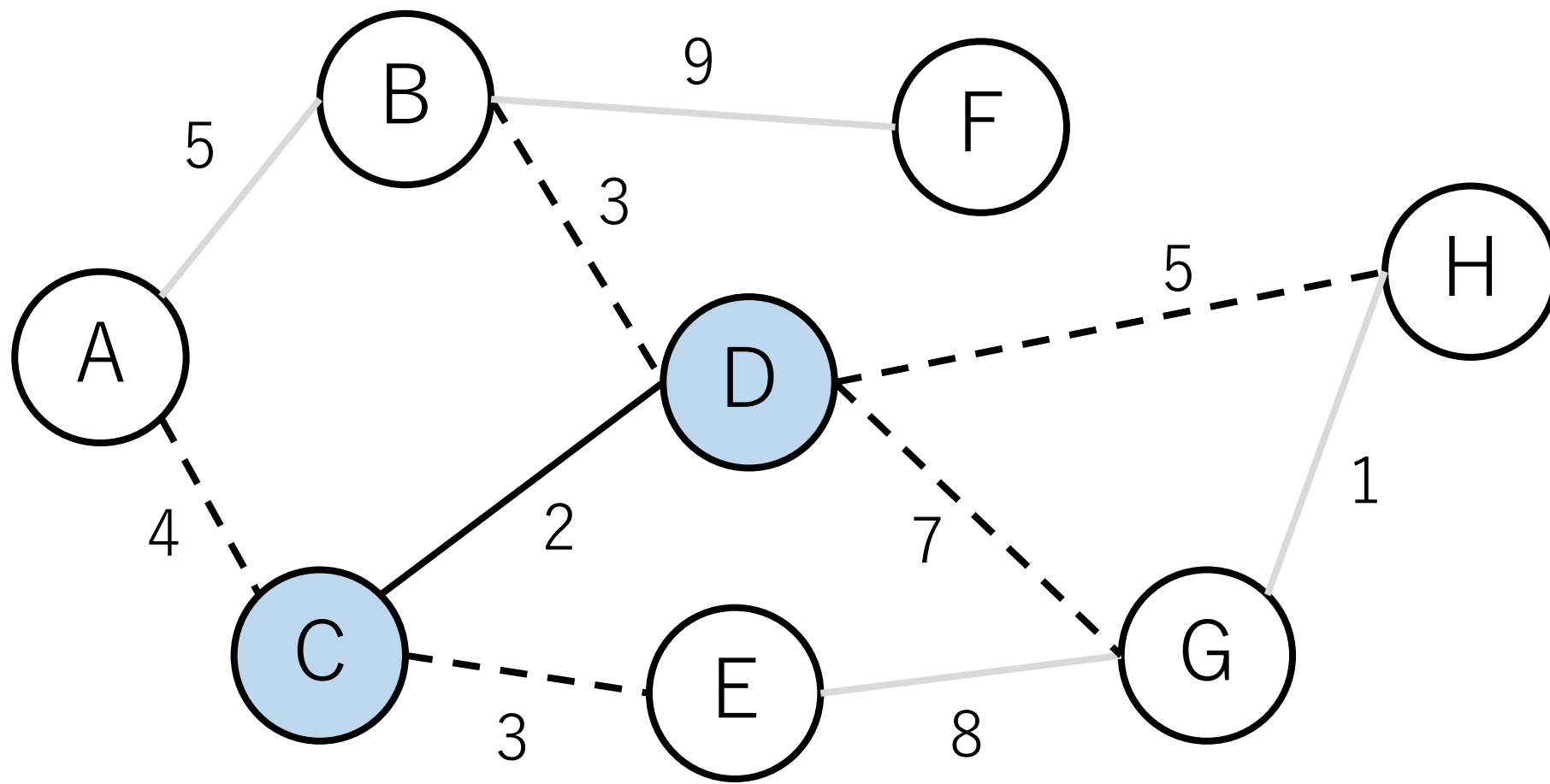
プリム法の例

CとDが「訪問済みグループ」になる。



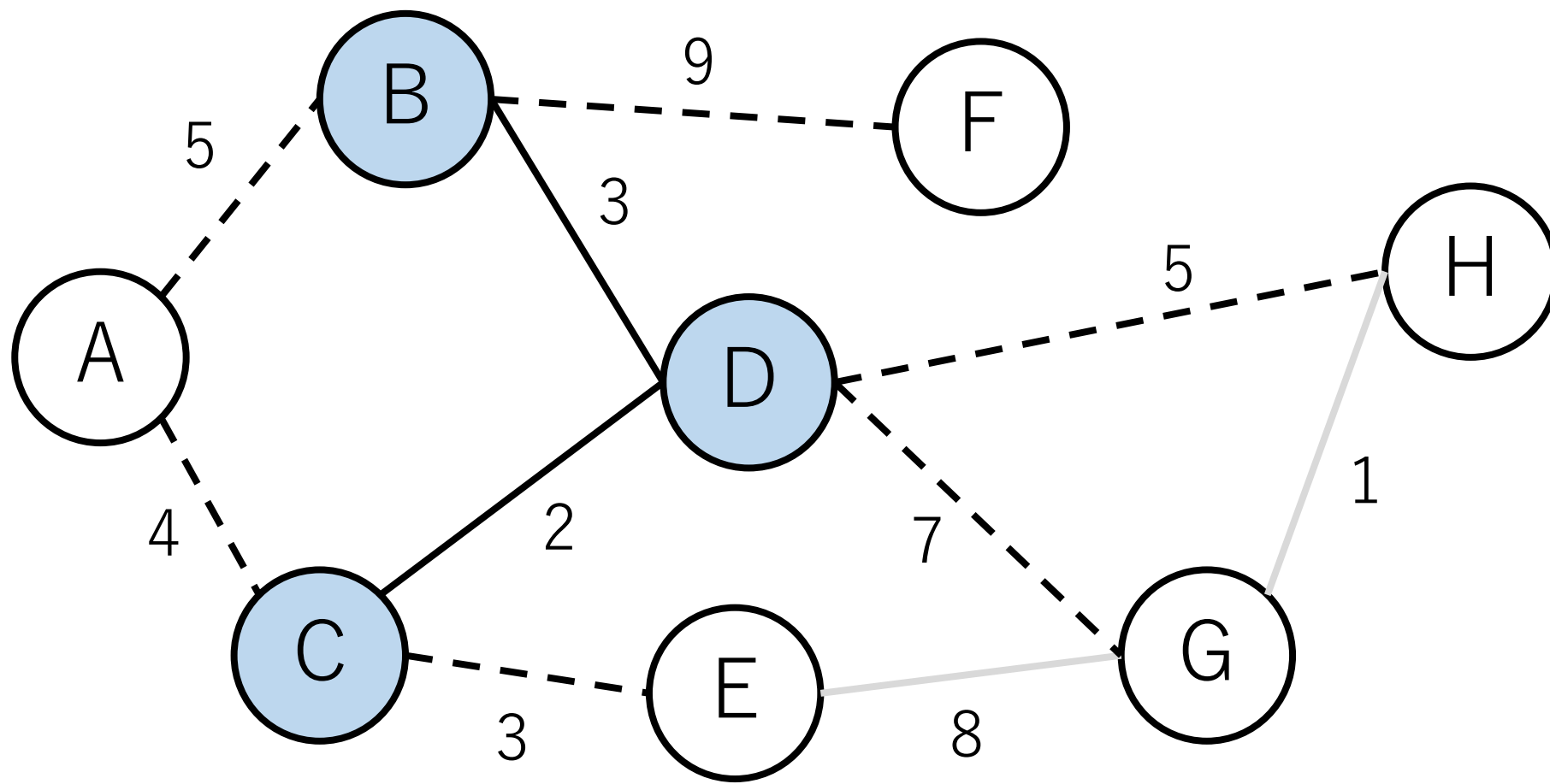
プリム法の例

次に青色ノードから白色ノードにつながっている辺で最短の距離のものを探す。



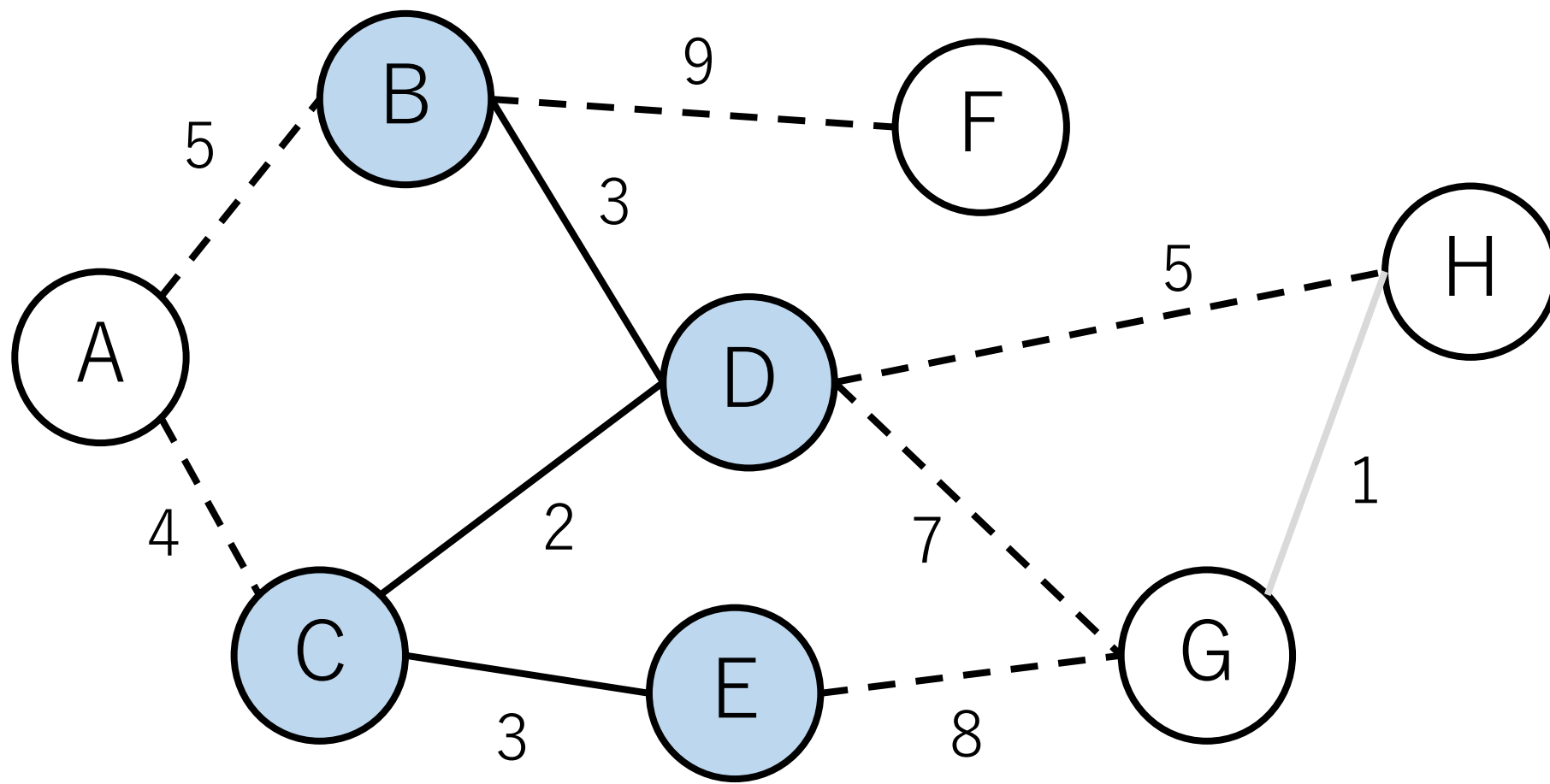
プリム法の例

最短距離は3 (B->DとC->E) . ここではB-Dをつなぐ.



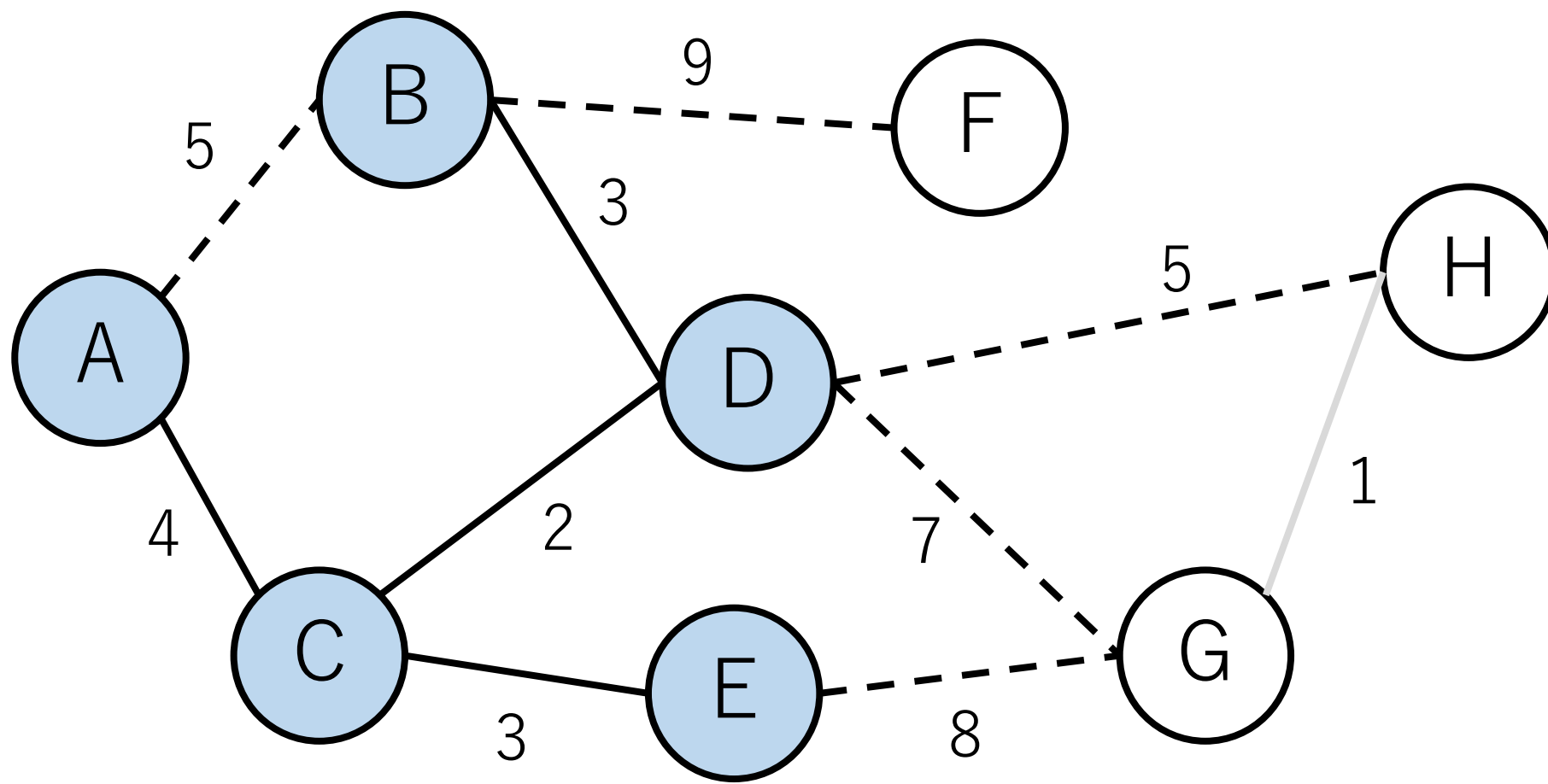
プリム法の例

次に最短距離のものはC-E.



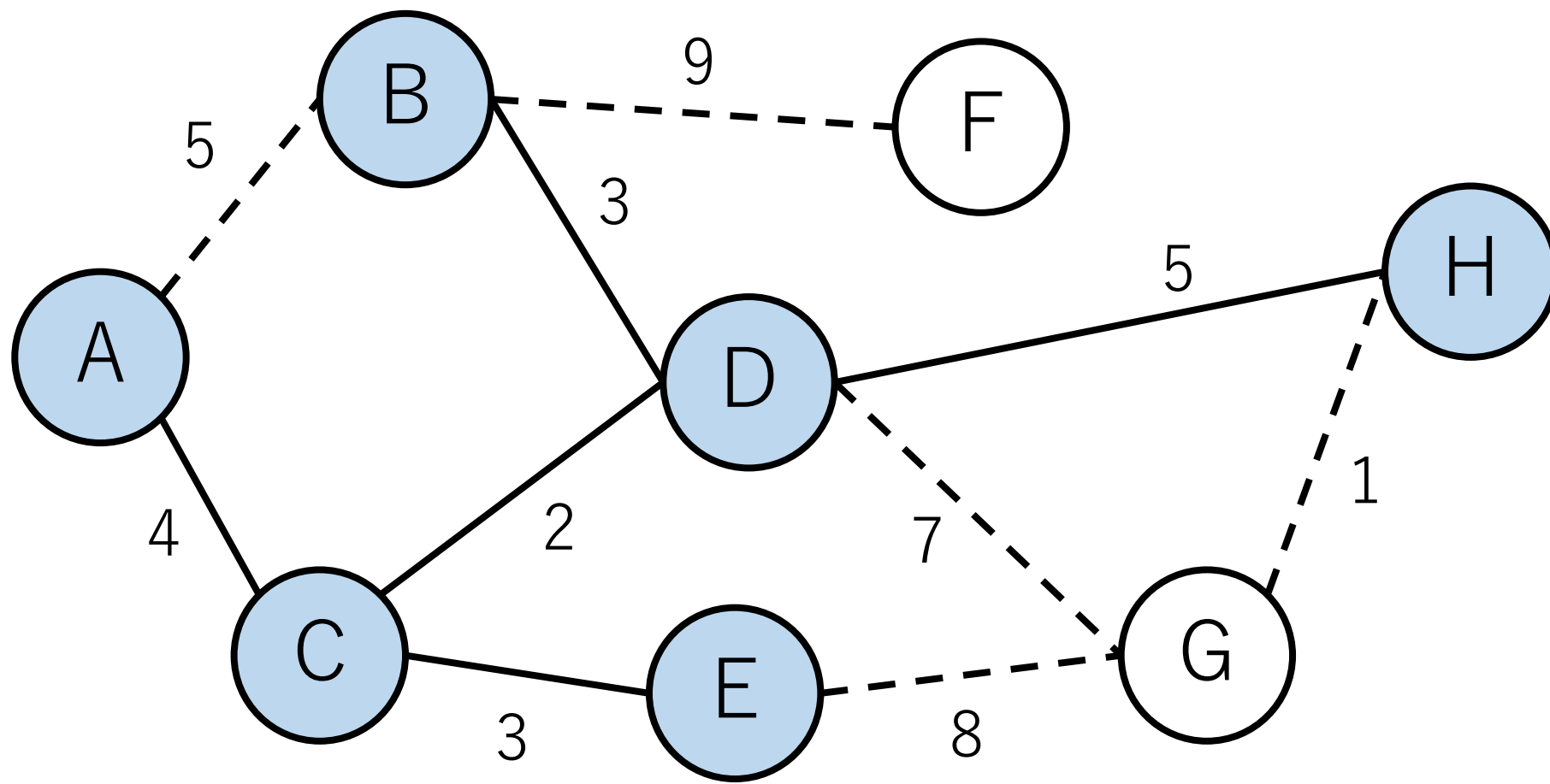
プリム法の例

その次は, A-C.



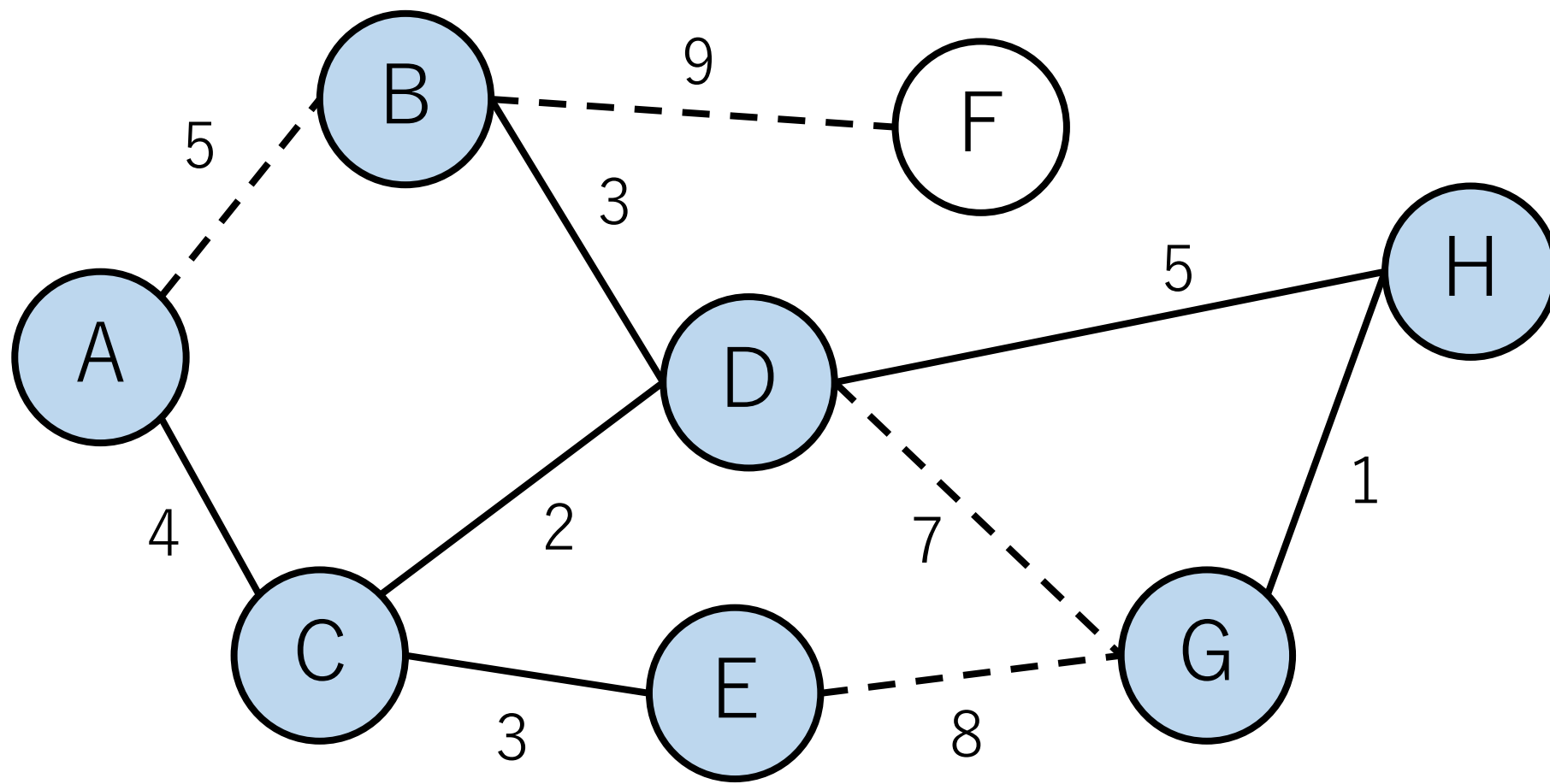
プリム法の例

その次は, D-H.



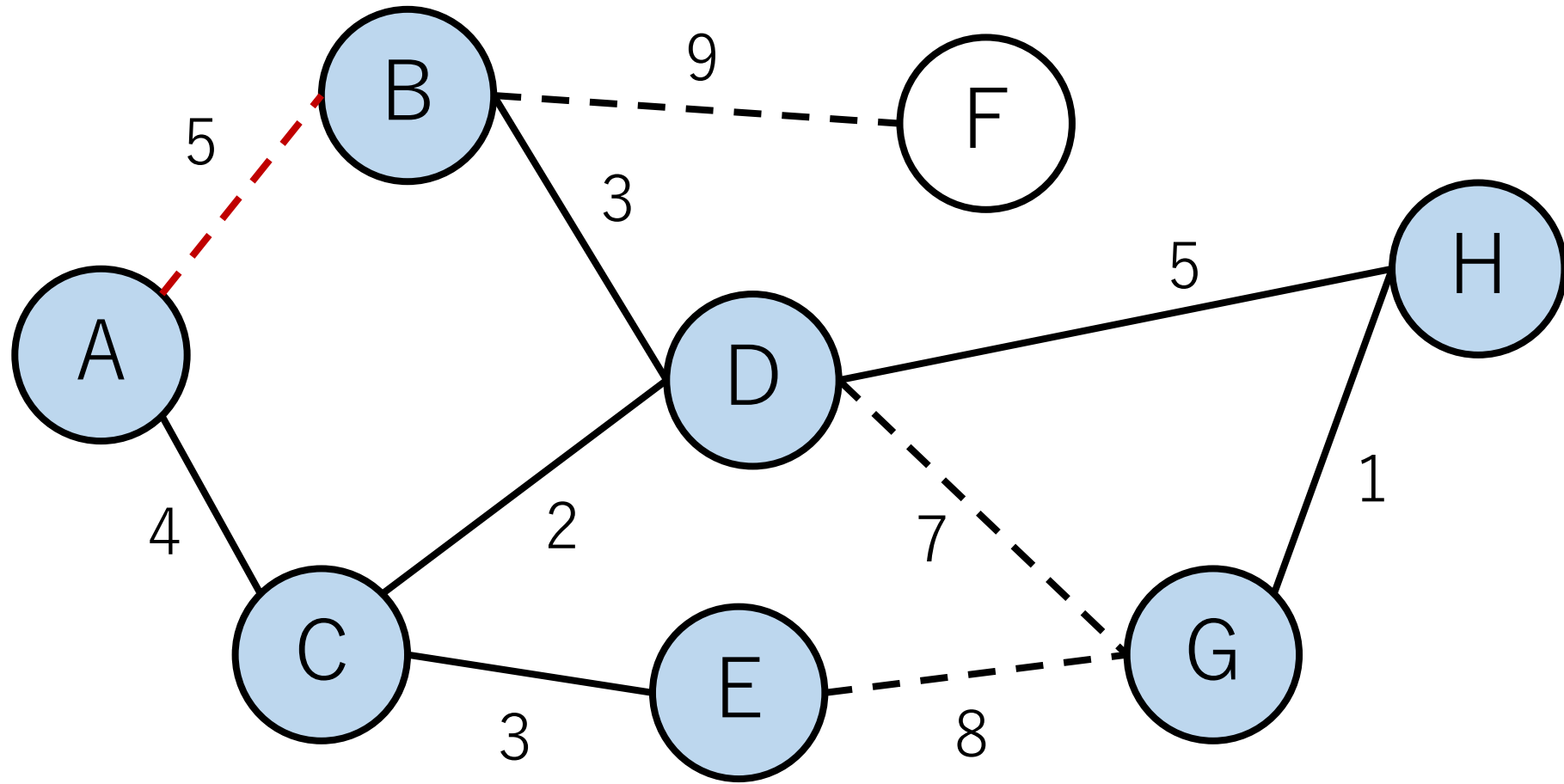
プリム法の例

その次は, G-H.



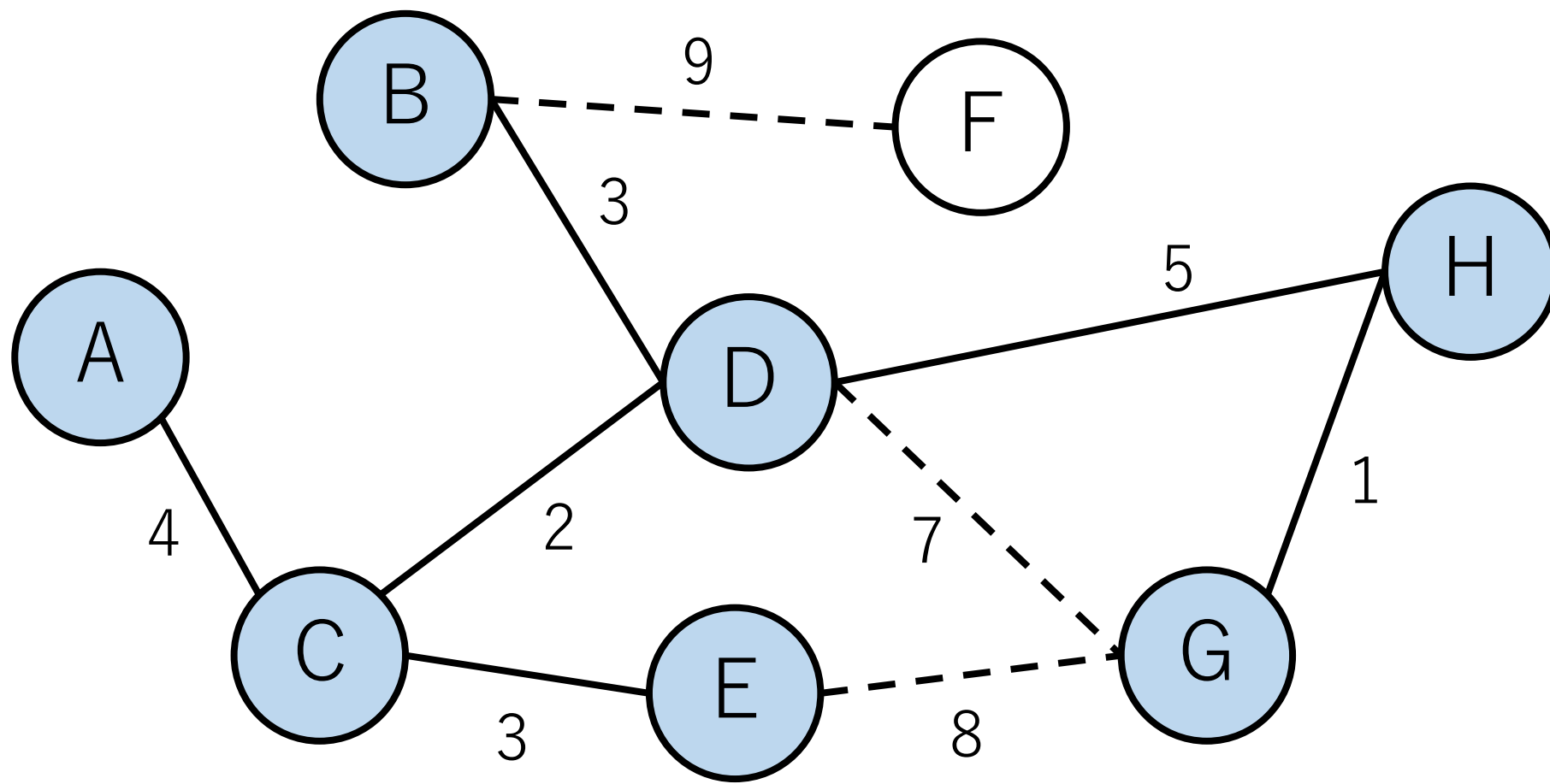
プリム法の例

その次は、A-Bだが、どちらのノードもすでに訪問済なのでこの辺はスキップする。



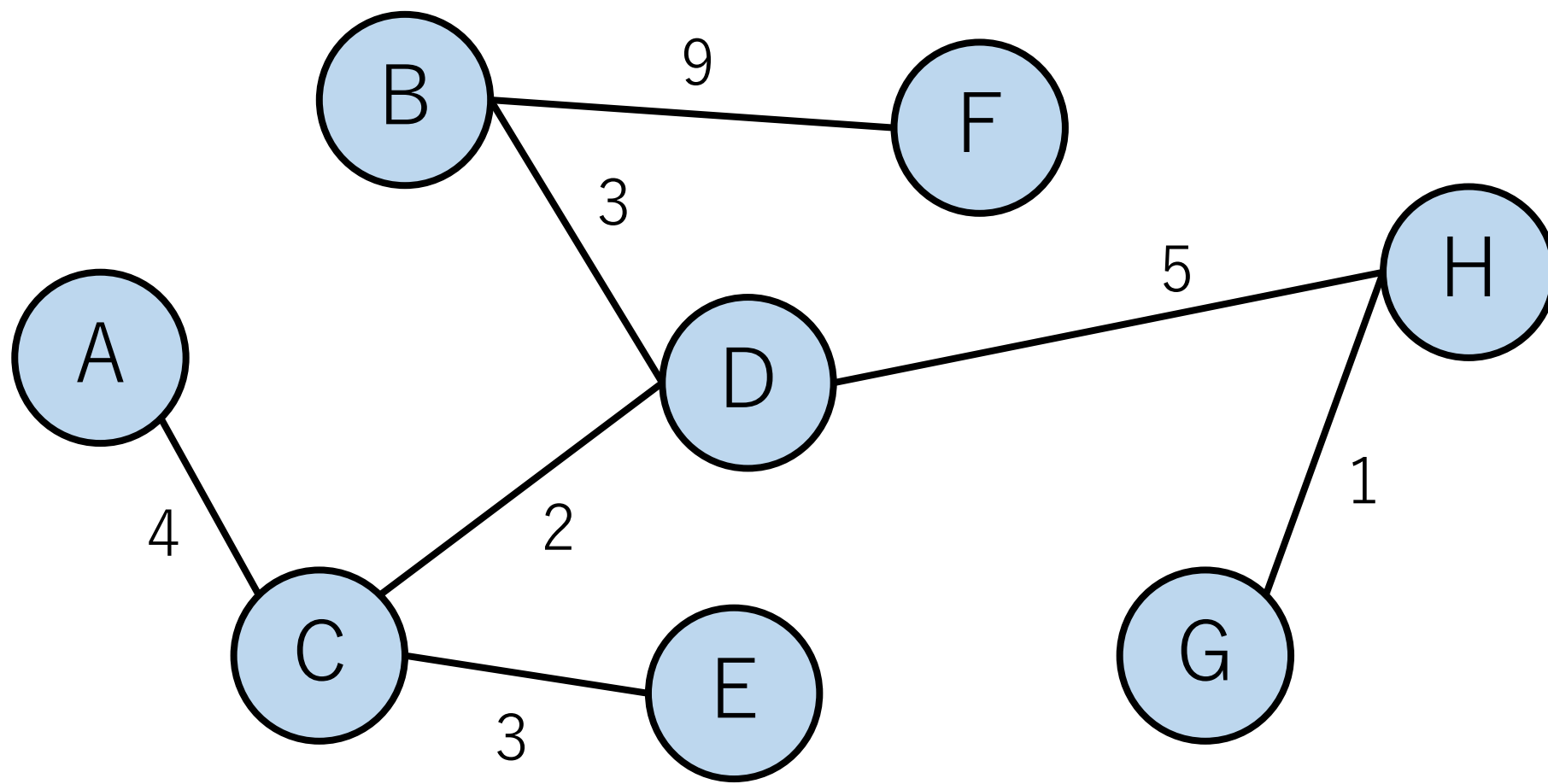
プリム法の例

D-G, E-Gについても同様にスキップ。



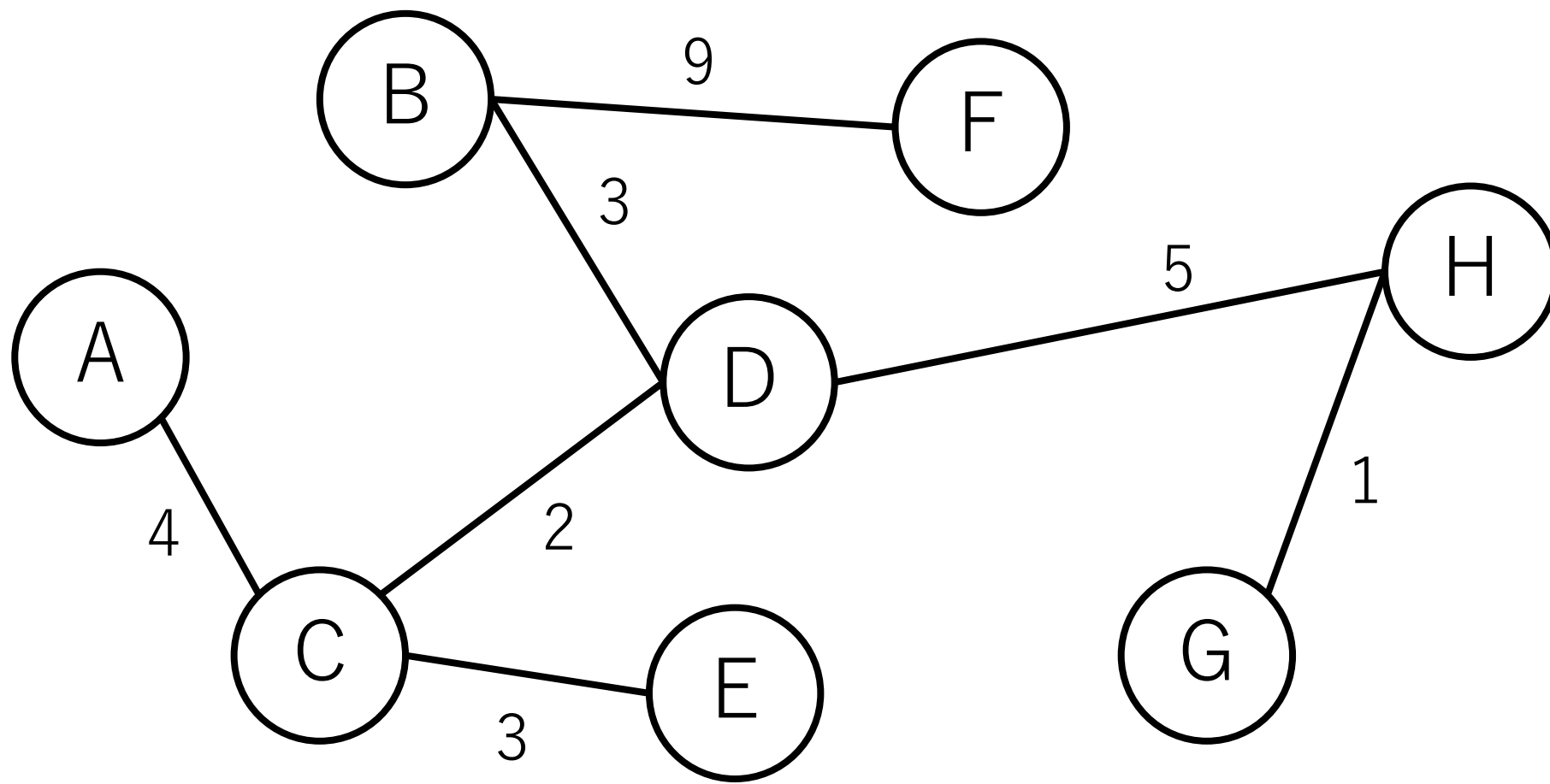
プリム法の例

B-FはFが未訪問だったので、最小全域木に入れる。



プリム法の例

これで終了.



プリム法の計算量

ノードの数を $|V|$ として,

隣接行列 + 最短の辺の単純な探索： $O(|V|^3)$

最短の辺の探索に $O(|V|^2)$, それを $O(|V|)$ 回.

隣接リスト + 最短の辺の単純な探索： $O(|E||V|)$

プリム法の計算量

ただし，ダイクストラのときのようにデータ構造を工夫することで高速化できる．

以下の実装例では，隣接リスト + 優先度付きキューを使っている．

プリム法の実装例

```
import heapq
```

```
def prim(V, e_list):
```

```
    # edges_from[i]はノードiからのすべての辺を格納
```

```
    edges_from = [[] for _ in range(V)]
```

```
    # ヒープでソートされるために距離を最初の要素にする
```

```
    for e in e_list:
```

```
        edges_from[e[0]].append([e[2], e[0], e[1]])
```

プリム法の実装例

```
def prim(V, e_list):  
    ...  
    e_heapq = []          # ヒープ  
    mst = []             # 最小全域木  
    # ノードが最小全域木に入ったかどうかのフラグ  
    included = [False]*V
```

プリム法の実装例

```
def prim(V, e_list):
```

```
    ...
```

```
    # ノードを1つ選ぶ. 何でも良いがこの実装では
```

```
    # ノード0を選ぶことにする.
```

```
    included[0] = True
```

```
    [ノード0に接続する辺を全てヒープに入れる]
```

プリム法の実装例

```
def prim(V, e_list):  
    ...  
    # ノード0に接続する辺を全てヒープに入れる。  
    for e in edges_from[0]:  
        heapq.heappush(e_heapq, e)
```

プリム法の実装例

```
def prim(V, e_list):
    ...
    while len(e_heapq):
        min_edge = heapq.heappop(e_heapq)
        #その辺の到達先（ノードj）が未訪問なら追加
        if not included[min_edge[2]]:
            included[min_edge[2]] = True
            mst.append([min_edge[1], min_edge[2]])
```


プリム法の実装例

```
def prim(V, e_list):
```

```
    ...
```

```
    while len(e_heapq):
```

```
        ...
```

```
            #ノードjから伸びる辺をe_heapqに入れる
```

```
            for e in edges_from[min_edge[2]]:
```

```
                if not included[e[2]]:
```

```
                    heapq.heappush(e_heapq, e)
```

プリム法の実装例

```
def prim(V, e_list):
```

```
    ...
```

```
    # ソートして表示
```

```
    mst.sort()
```

```
    print(mst)
```

プリム法の実行例

```
edges_list = [[0, 1, 5], [0, 2, 4], [1, 0, 5], [1, 3, 3], [1, 5, 9],  
[2, 0, 4], [2, 3, 2], [2, 4, 3], [3, 1, 3], [3, 2, 2], [3, 6, 7],  
[3, 7, 5], [4, 2, 3], [4, 6, 8], [5, 1, 9], [6, 3, 7], [6, 4, 8],  
[6, 7, 1], [7, 3, 5], [7, 6, 1]]
```

```
prim(8, edges_list)
```

=== 実行結果 ===

```
[[0, 2], [1, 5], [2, 3], [2, 4], [3, 1], [3, 7], [7, 6]]
```

プリム法の計算量（ヒープを使う場合）

上記の実装では，ヒープに入る要素の数は辺の総数になるので， $O(|E|)$.

よって，追加，削除にかかる計算量は $O(\log|E|)$.

ヒープへの追加も取り出しも $O(|E|)$ 回あるので，全体では $O(|E| \log|E|)$ となる。

（ダイクストラ法のとくに説明したとおり， $O(\log|E|)$ は $O(\log|V|)$ と等価であるとも考えられるので， $O(|E| \log|V|)$ と説明される場合もある。）

プリム法の計算量

ダイクストラ法と同じように、フィボナッチヒープを使うことにより、 $O(|E| + |V| \log |V|)$ に落とせることが知られている。

今日のテーマ

最小全域木

トポロジカルソート

トポロジカルソート

閉路の無い有向グラフを「ソート」する。

全ての有向辺が1つの向きになるようにノードを並び替える。

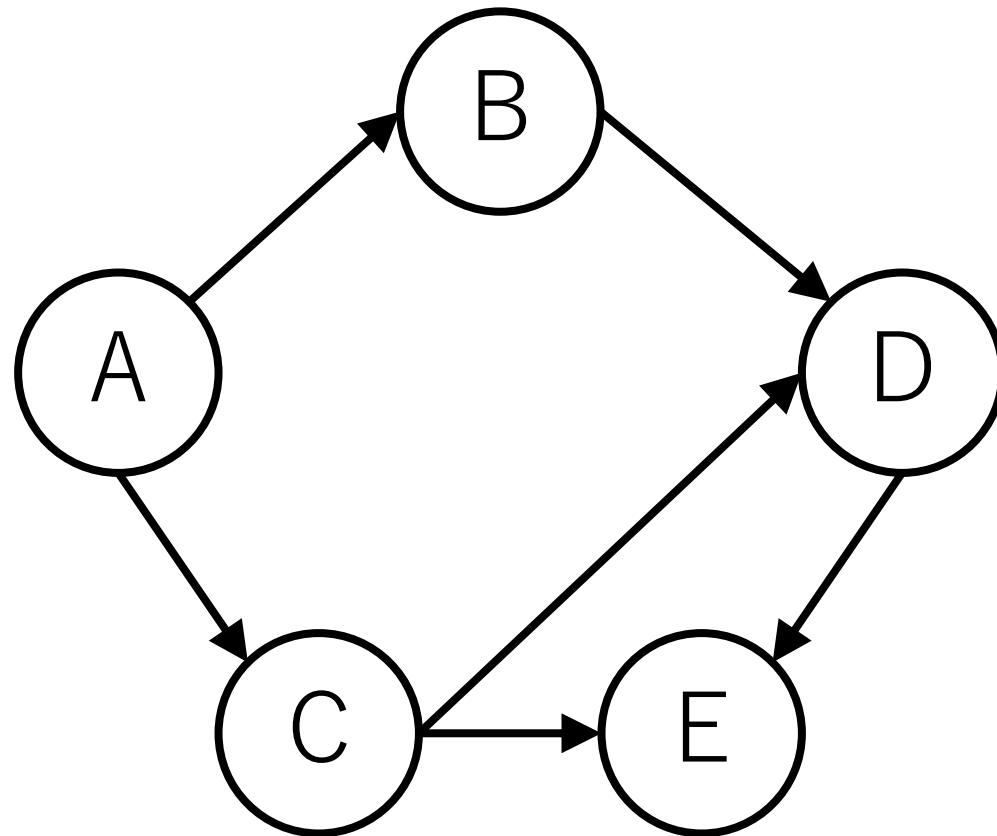
このようなグラフを有向非巡回グラフと呼ぶ。

英語ではDirected Acyclic Graph (DAG) .

また、与えられるDAGには多重辺がないとする。

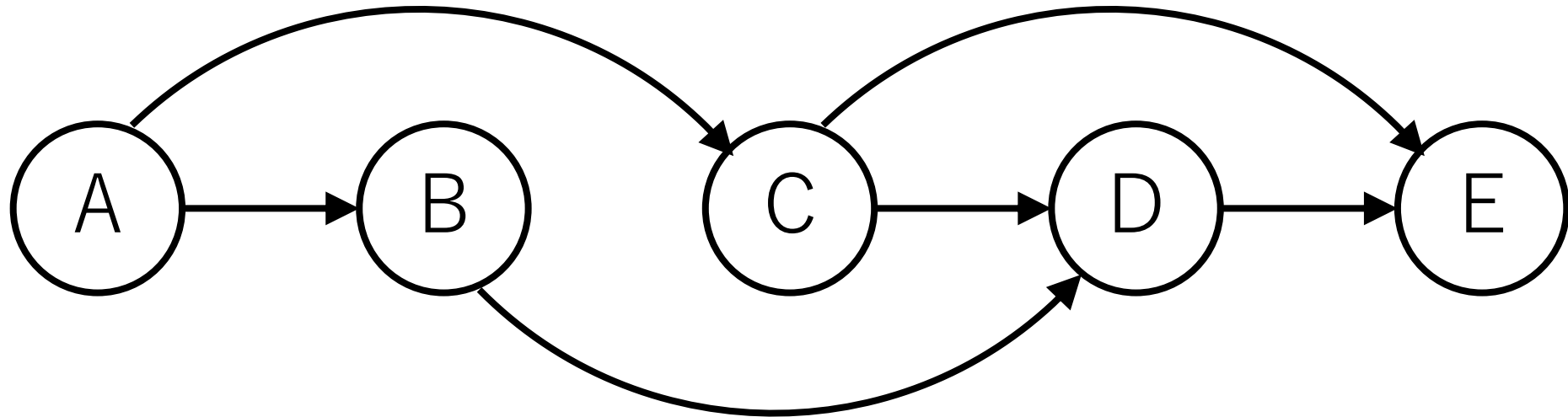
DAG

DAGの例. 巡回できる経路は存在しない.



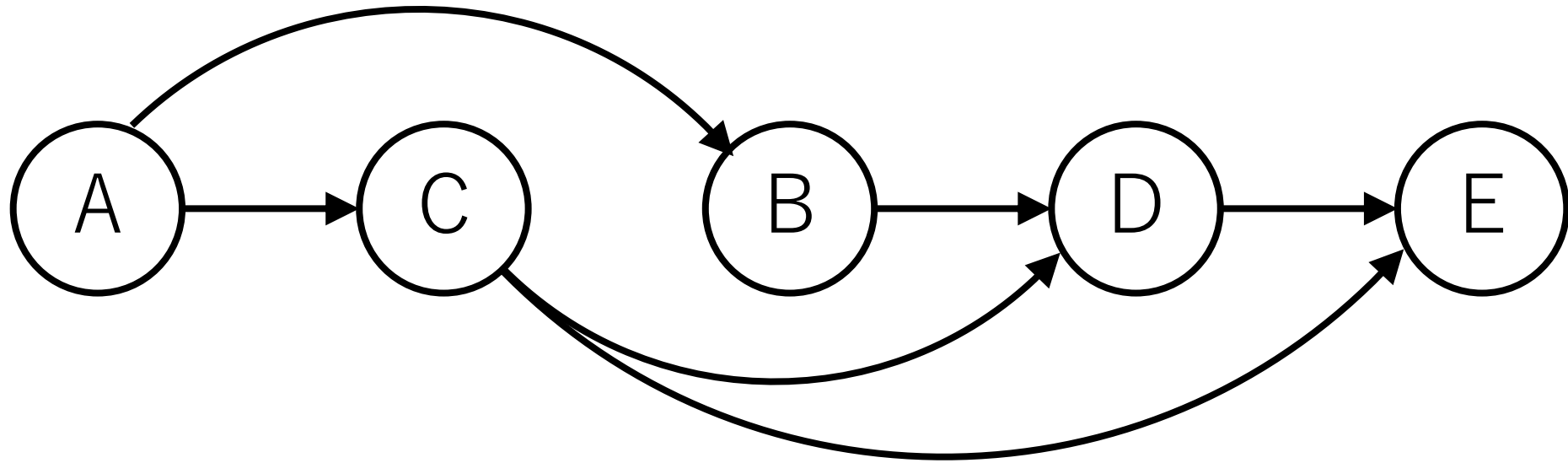
トポロジカルソート例

すべての辺が右方向に向くようにノードを並べ替えることができる。



トポロジカルソート例

トポロジカルソートの結果は1つとは限らない。



次数, 入次数

次数 (degree) : あるノードにつながっている辺の総数.

入次数 (indegree) : あるノードに入ってくる辺の総数.

言葉としては, 出次数 (outdegree) もある.

あるノードから出ていく辺の総数.

入次数, 出次数はそれぞれ「いりじすう」, 「でじすう」と読むのが正式だそうです.

<http://dopal.cs.uec.ac.jp/okamotoy/lect/2014/gn/term01.pdf>

次数, 入次数

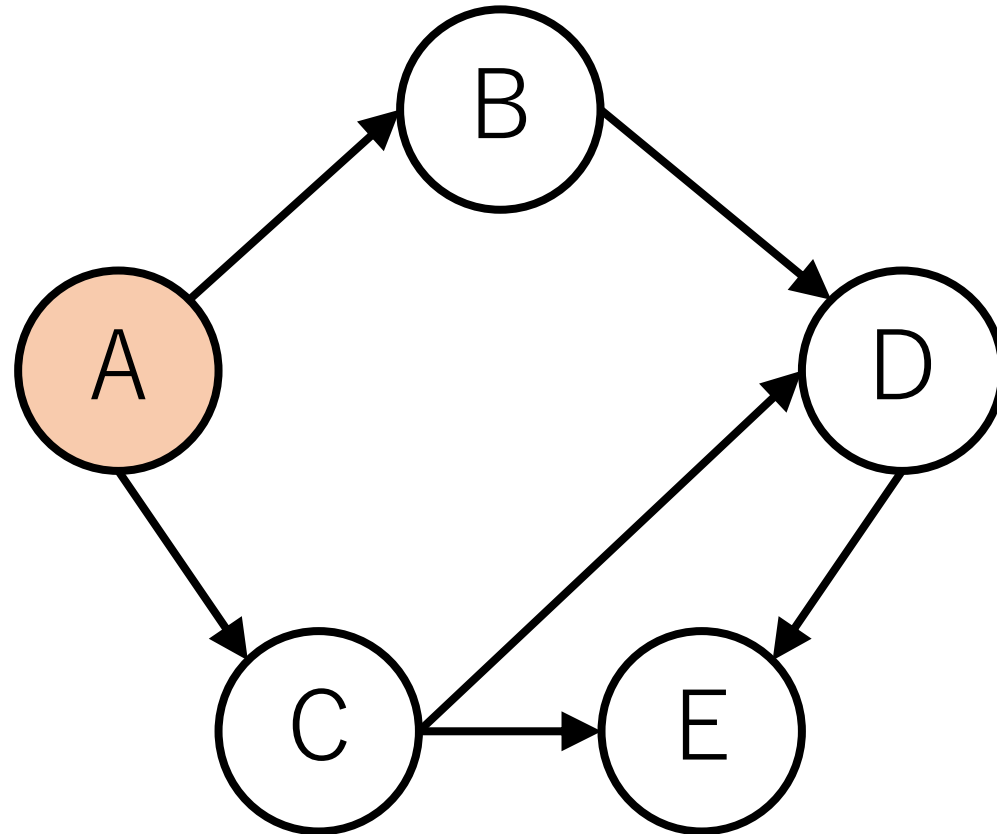
$$[\text{次数}] = [\text{入次数}] + [\text{出次数}]$$

自分自身へのループは入次数, 出次数ともに1ずつカウントされる.

よって, 自分自身へのループ1つに対して, 次数は2つ増える.

DAG

DAGには必ず、入次数0のノードが最低1つ存在する。
存在しなければ閉路が存在し、DAGにならない。



トポロジカルソート

代表的なものは2つ.

Kahnさんが提案したもの.

DFSをベースにしたもの.

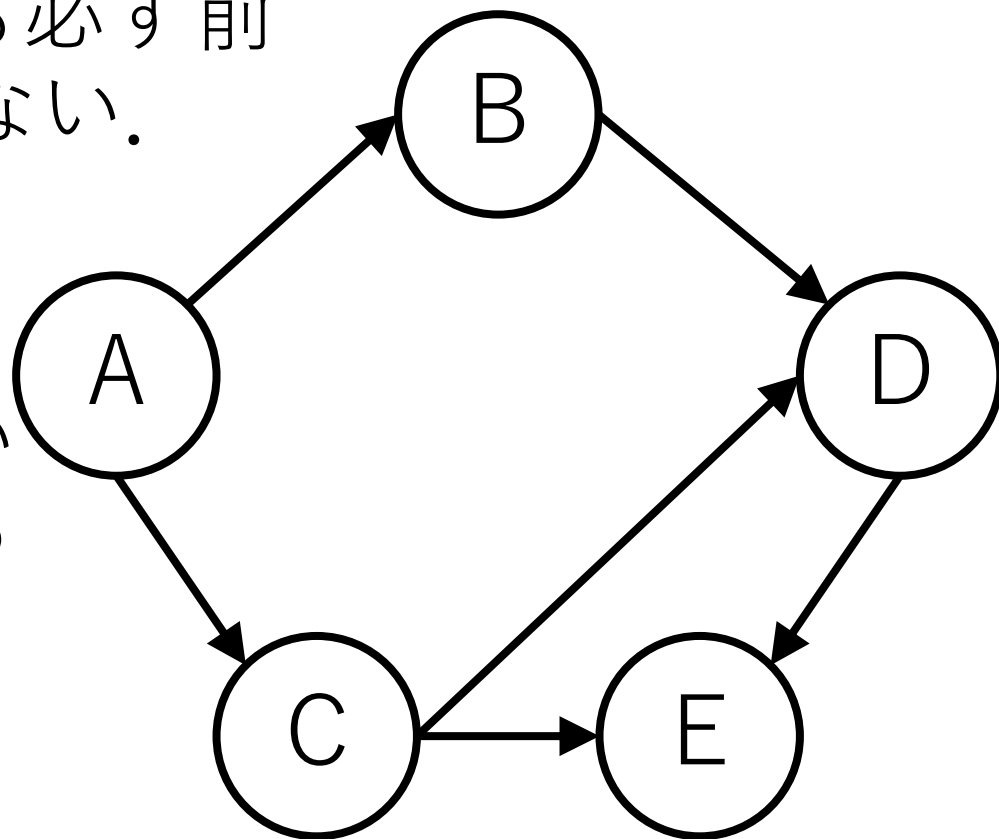
今日はこの2つを順に紹介をしていきます.

Khanのトポロジカルソートの方針

入次数0のノードを見つけ出し，それをグラフから取り除き，ソート済の場所に入れていく。

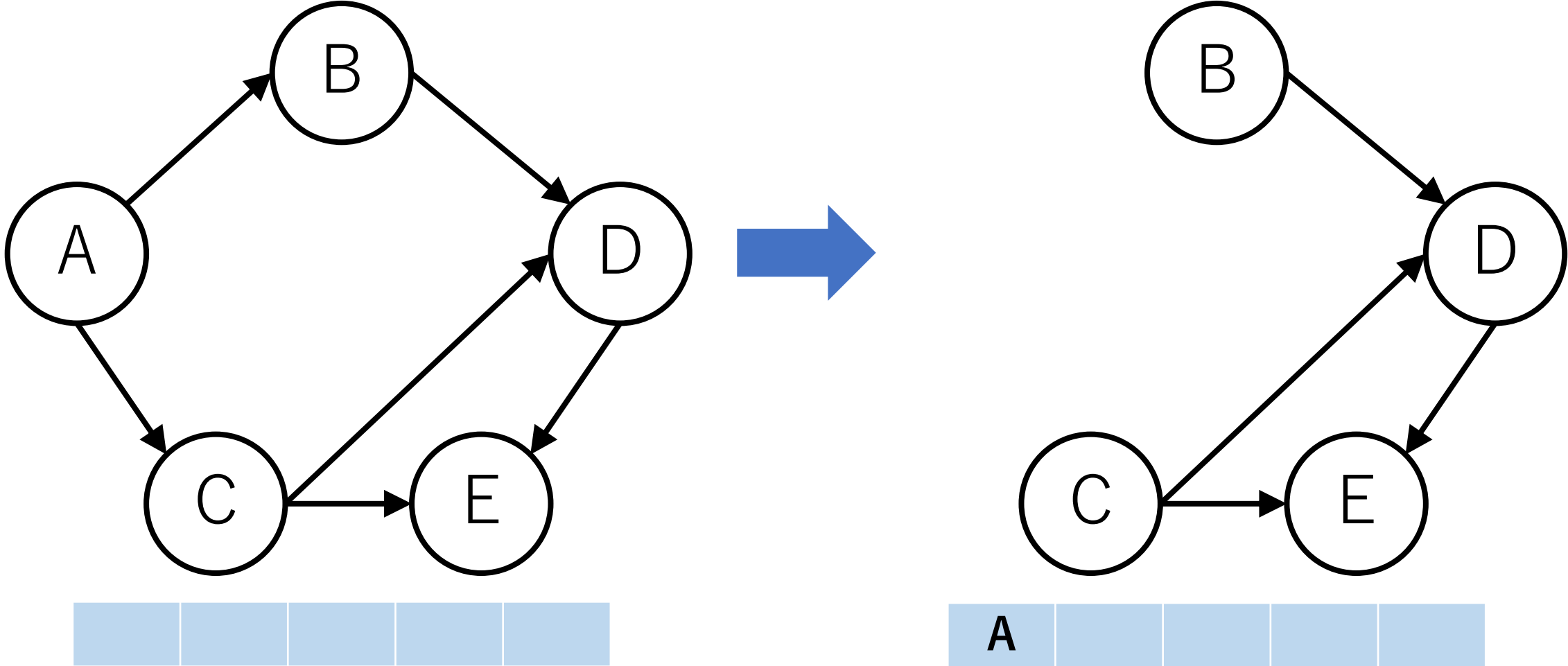
入次数が0 = このノードよりも必ず前（左側）に並ぶべきノードはない。

例えば，右のグラフではノードAは，その前段につながるものはないので，ソートした時一番最初に来ることになる。



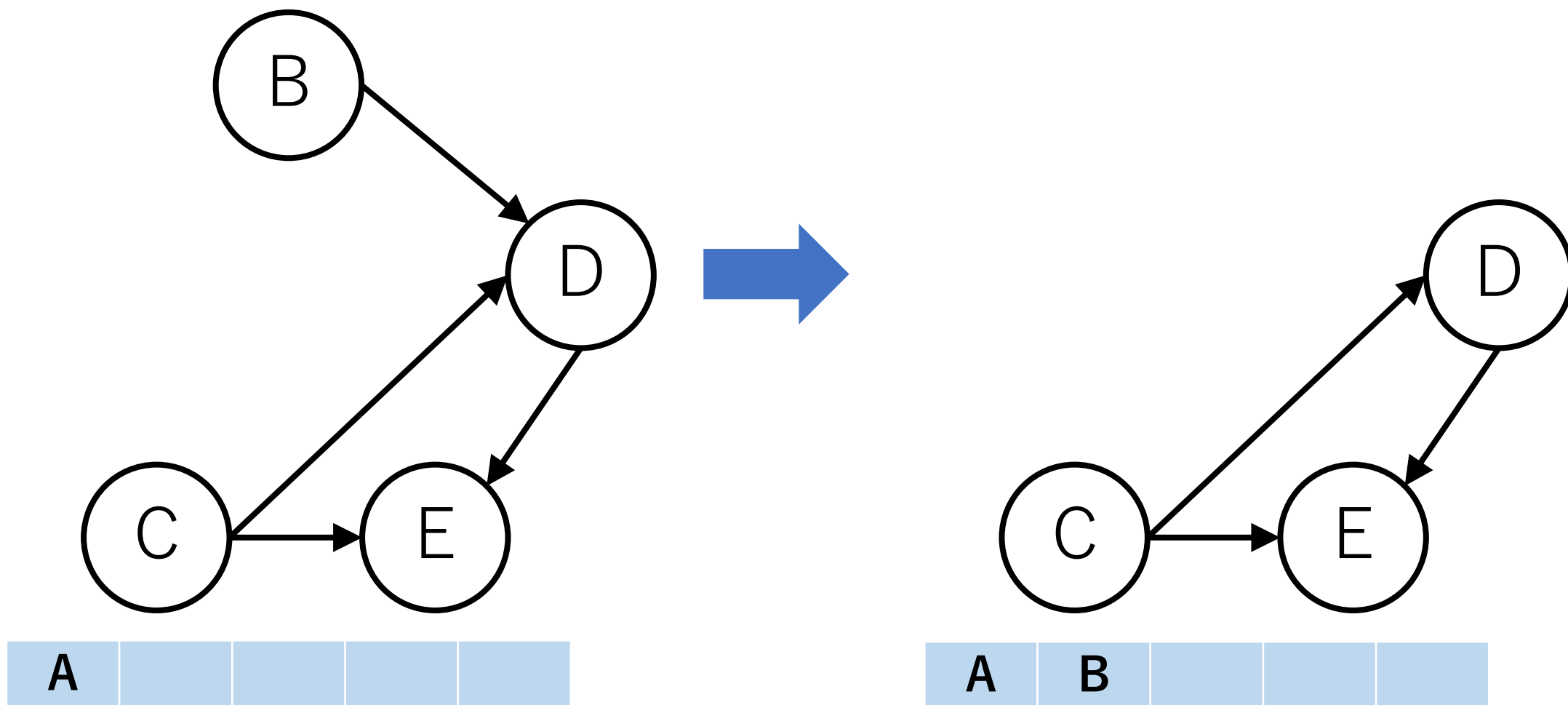
Khanのトポロジカルソートの実行例

まずノードAを取り出してソート済とする。



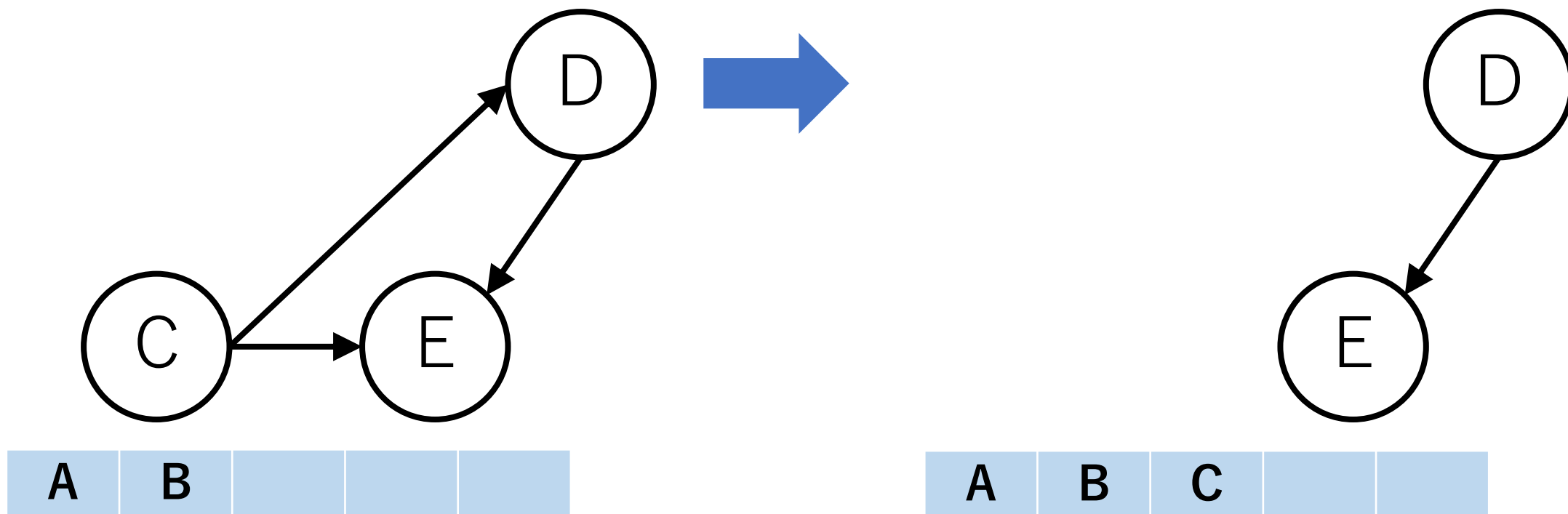
Khanのトポロジカルソートの実行例

次に入次数が0のノードBを取り出す（ノードCでもよい）。



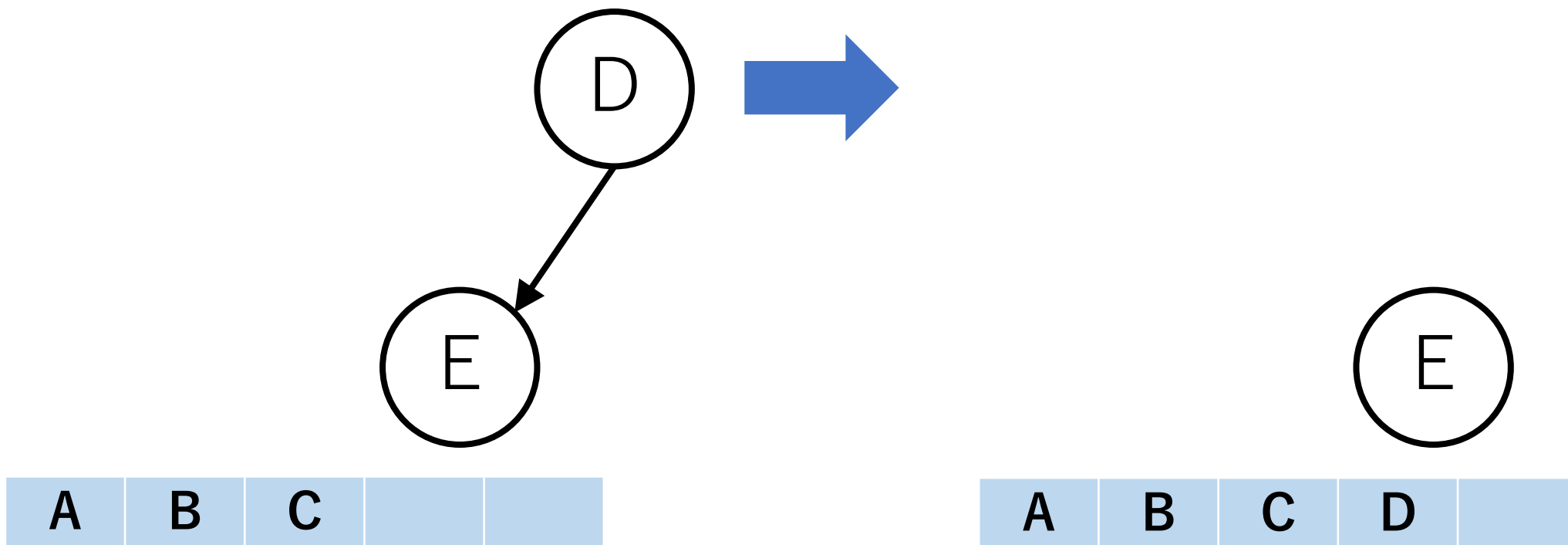
Khanのトポロジカルソートの実行例

次に入次数が0のノードCを取り出す。



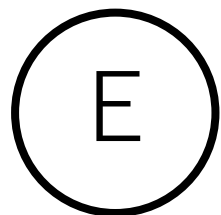
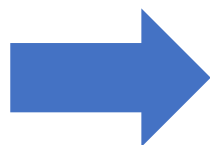
Khanのトポロジカルソートの実行例

次に入次数が0のノードDを取り出す。



Khanのトポロジカルソートの実行例

以降、入次数が0となるノードがなくなるまで繰り返す。

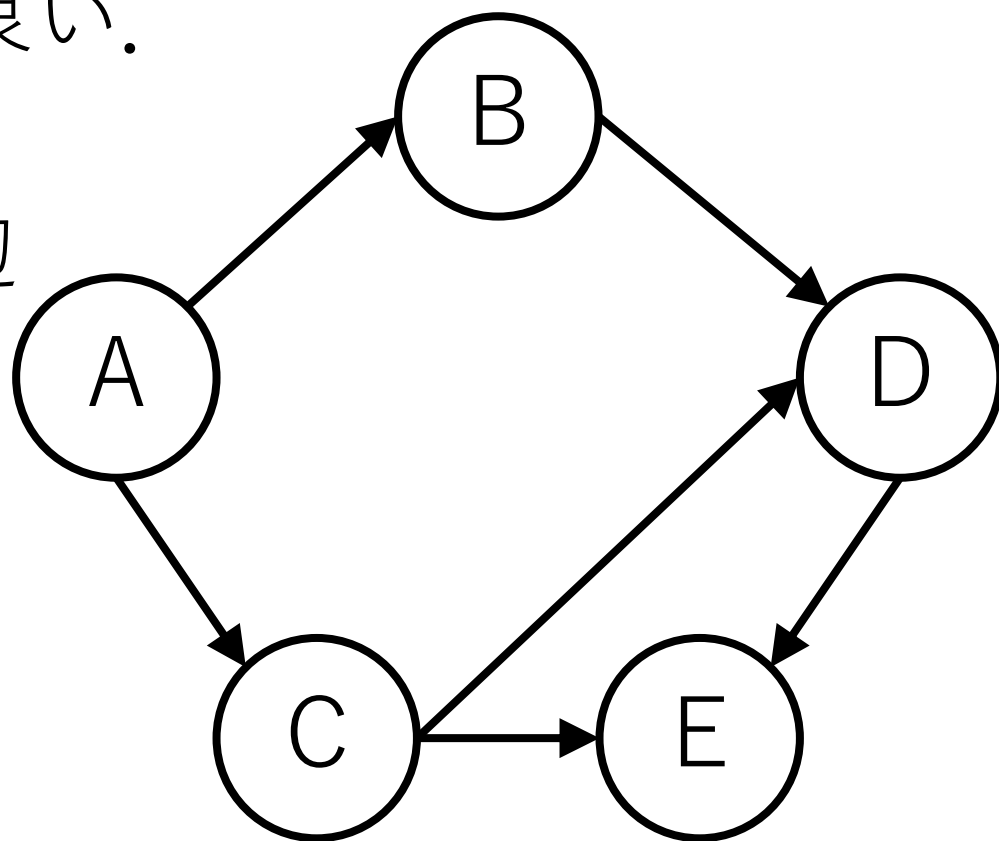


Khanのトポロジカルソートの実装方針

具体的な実装としては、各ノードの入次数を予め記録しておき、ノードを取り出すたびに入次数の値を更新する。

更新するときには、 -1 すれば良い。

この時、取り除いたノードの出力辺の接続先ノードの入次数のみ更新すれば良い。



Khanのトポロジカルソート

whileループを抜けたあと、まだ残っている辺がある場合、DAGになっていないので、エラーを返す。

Khanのトポロジカルソートの実装例

$V = 5$ # ノードの総数

$E = 6$ # 辺の総数

有向辺の配列

`edges = [[0, 1], [0, 2], [1, 3], [2, 3], [2, 4], [3, 4]]`

`print(topoSort(V, E, edges))`

Khanのトポロジカルソートの実装例

```
from collections import deque
```

```
def topoSort(V, E, edges):
```

```
    indeg = [0]*V # 入次数を格納する配列
```

```
    # 出力辺を保持する配列
```

```
    outedge = [[] for _ in range (V)]
```


Khanのトポロジカルソートの実装例

```
def topoSort(V, E, edges):  
    ...  
    # 入次数と出力辺の情報を整理する  
    for v_from, v_to in edges:  
        indeg[v_to] += 1  
        outedge[v_from].append(v_to)
```

Khanのトポロジカルソートの実装例

```
def topoSort(V, E, edges):
```

```
    ...
```

```
    # ソート済のノードを格納する配列
```

```
    # 最初に入次数0のものを入れておく
```

```
    sorted_g = list(v for v in range(V) if indeg[v]==0)
```

```
    # 入次数0のノードを処理するためのdeque
```

```
    deq = deque(sorted_g)
```

Khanのトポロジカルソートの実装例

```
def topoSort(V, E, edges):
```

```
    ...
```

```
    while deq: # 入次数0のノードがある限り繰り返す
```

```
        v = deq.popleft()    # deq.pop()でもよい
```

```
        [for vからつながるすべてのノードu]:
```

```
            [E, uの入次数を1減らす]
```

```
            if [uの入次数が0]:
```

```
                [uをdeqとsorted_gに入れる]
```

Khanのトポロジカルソートの実装例

```
def topoSort(V, E, edges):
```

```
    ...
```

```
    if E != 0:
```

```
        [DAGになっておらず, エラーを返す]
```

```
    return sorted_g
```

Khanのトポロジカルソートの実行結果例

$V = 5$

$E = 6$

$edges = [[0, 1], [0, 2], [1, 3], [2, 3], [2, 4], [3, 4]]$

```
print(topoSort(V, E, edges))
```

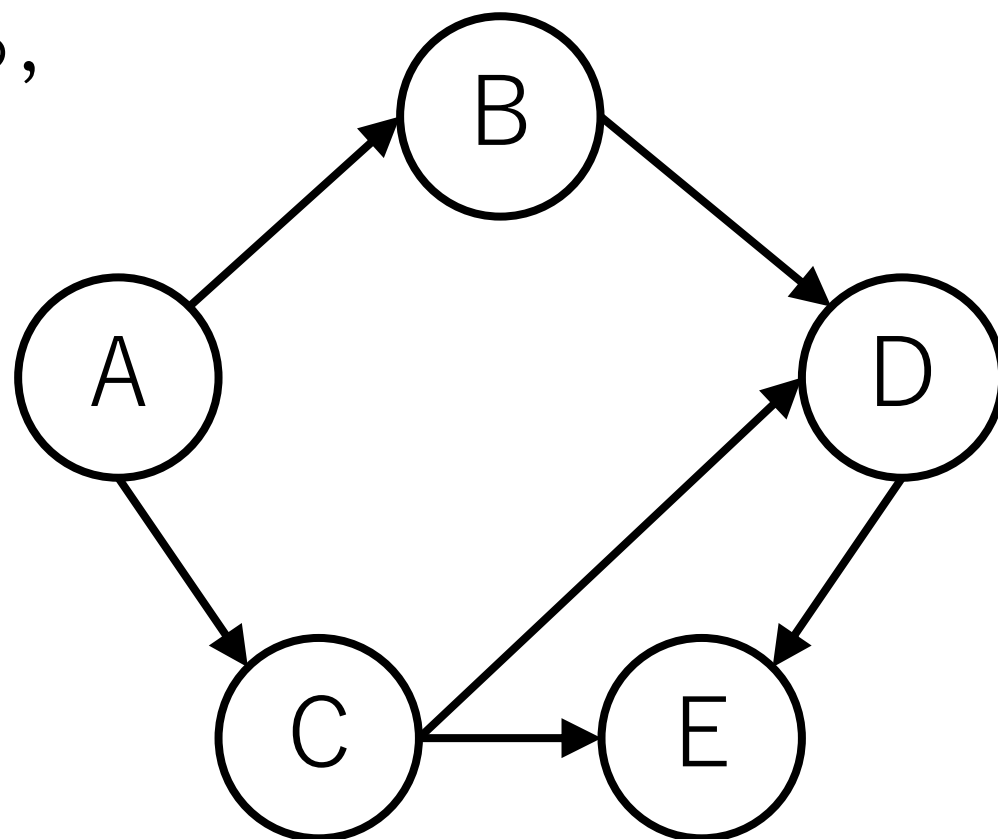
```
[0, 1, 2, 3, 4]
```

DFSを使うトポロジカルソートの実装方針

ノードを1つ選び，DFSでたどっていく。

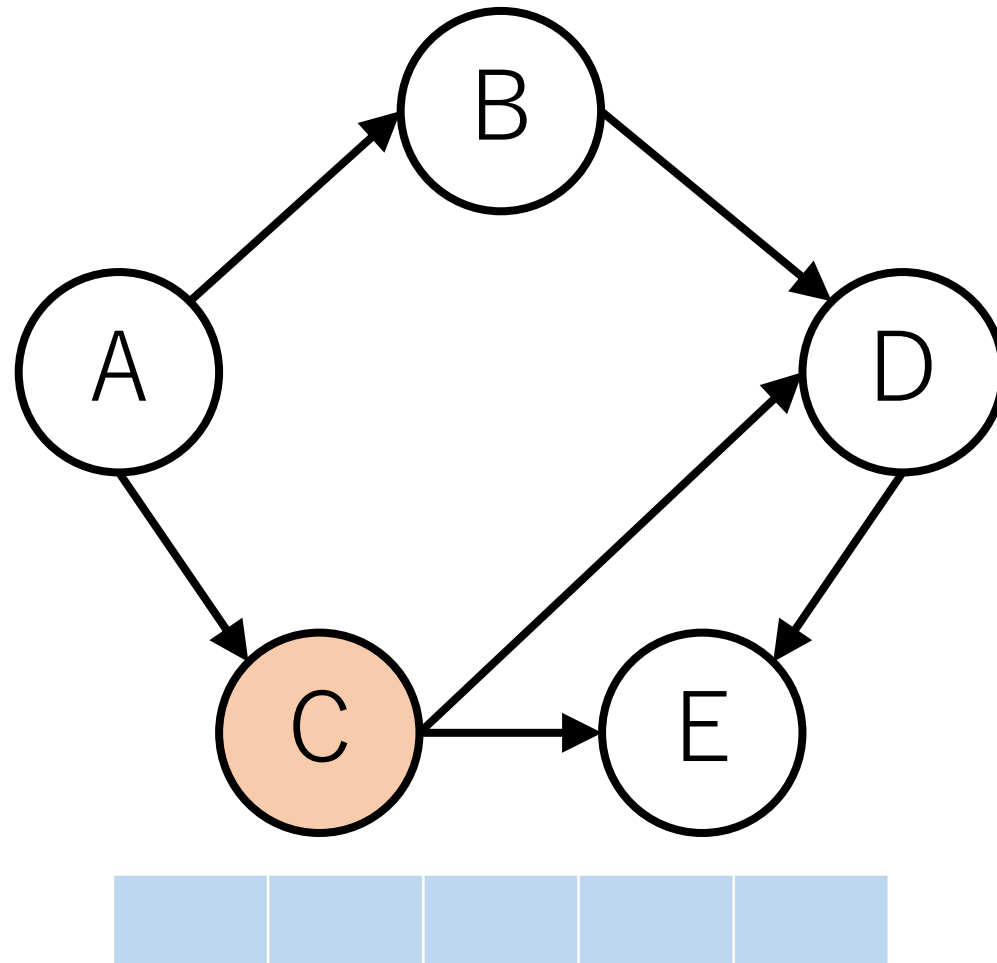
先に進めないところまで到達したら，
後戻りしながらソート済の場所に
先頭から順に入れていく。

これを全てのノードがチェック
されるまで繰り返す。



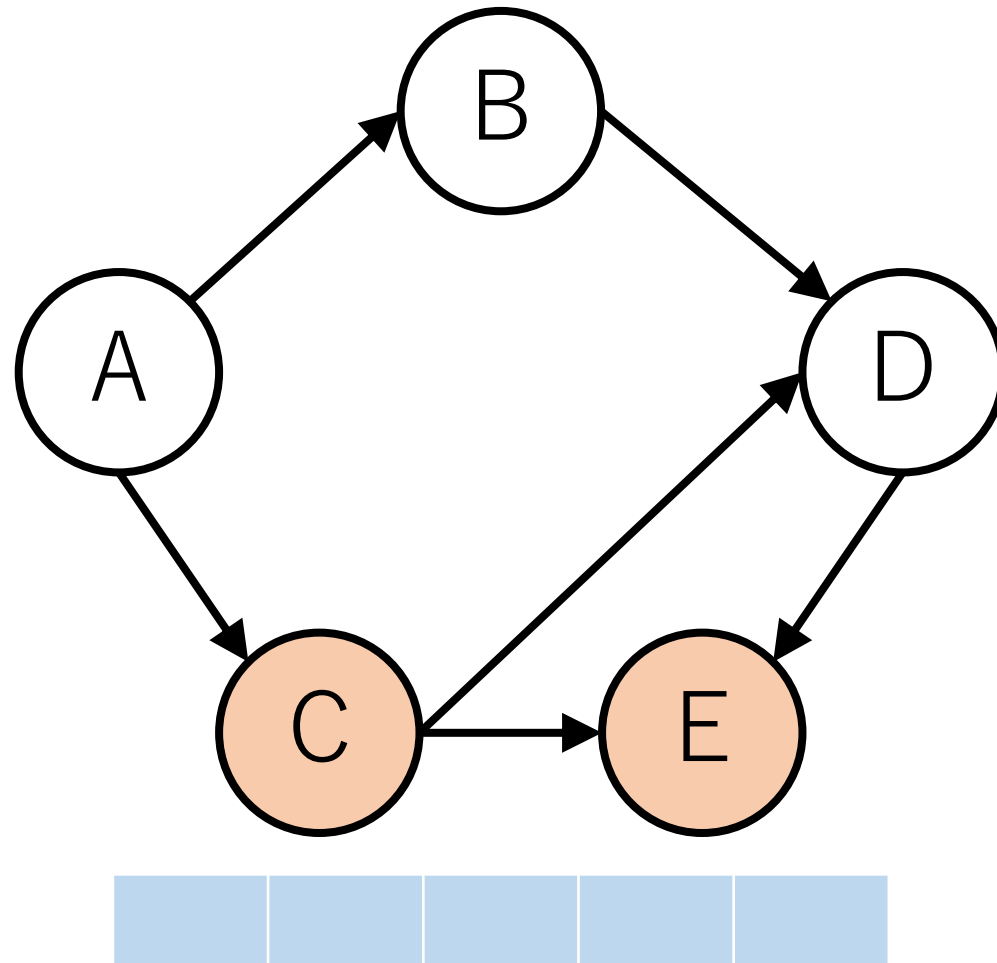
トポロジカルソート (DFS版) の実行例

ノードCからスタートする (どこからスタートしても良い) .



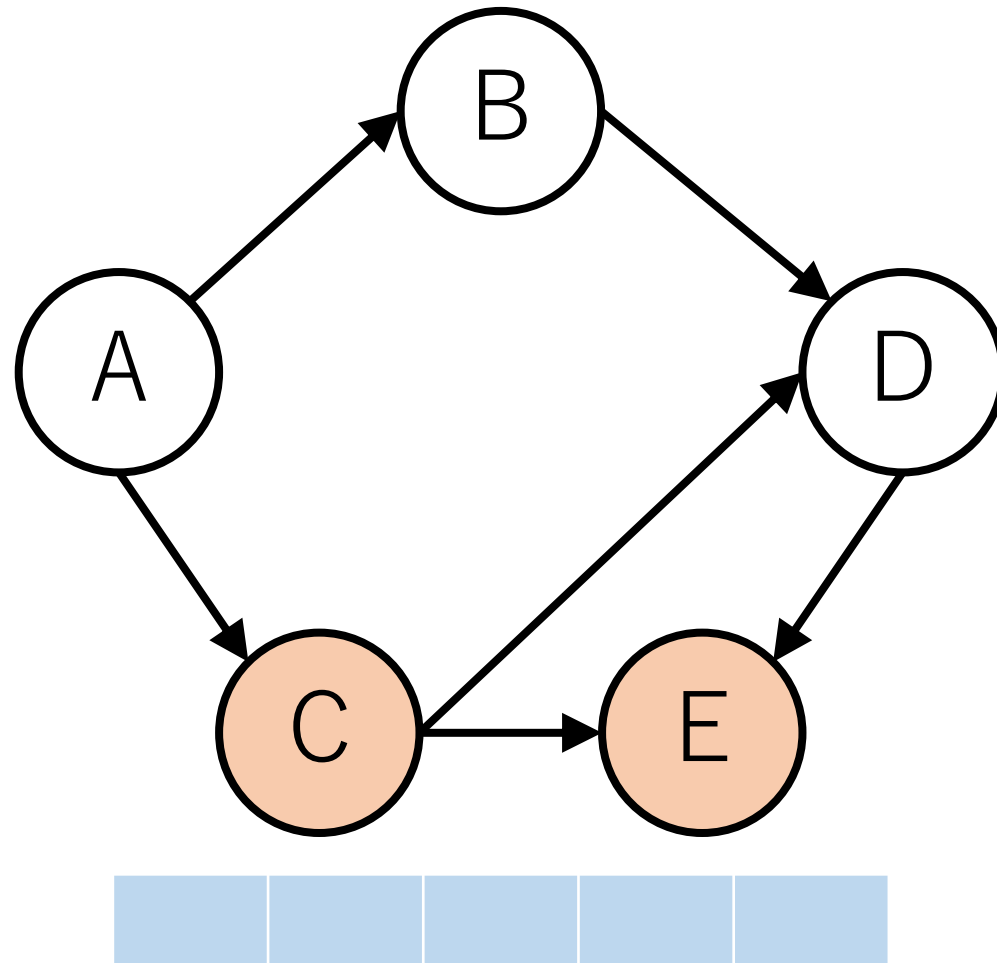
トポロジカルソート (DFS版) の実行例

DFSで行けるところまで行く。今回はC->Eと行ったとする。



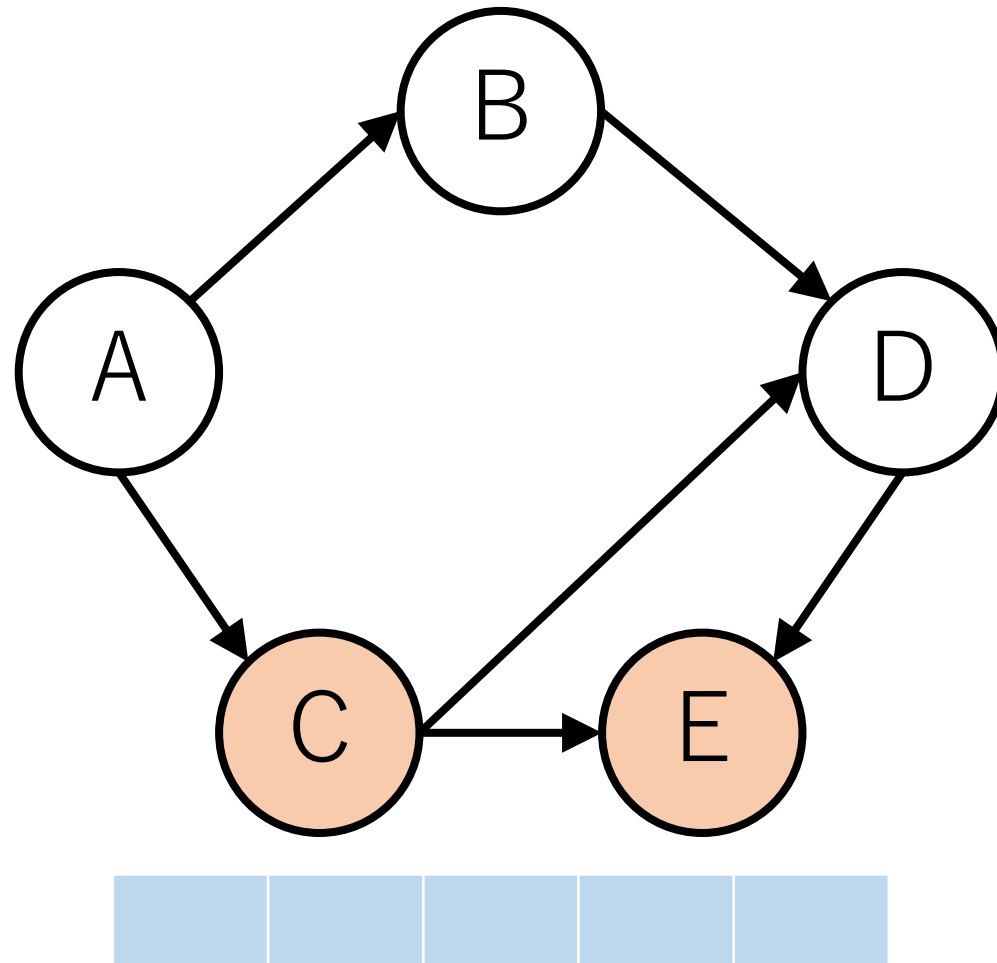
トポロジカルソート (DFS版) の実行例

進めなくなったら、逆戻りし、ソート済に入れる。



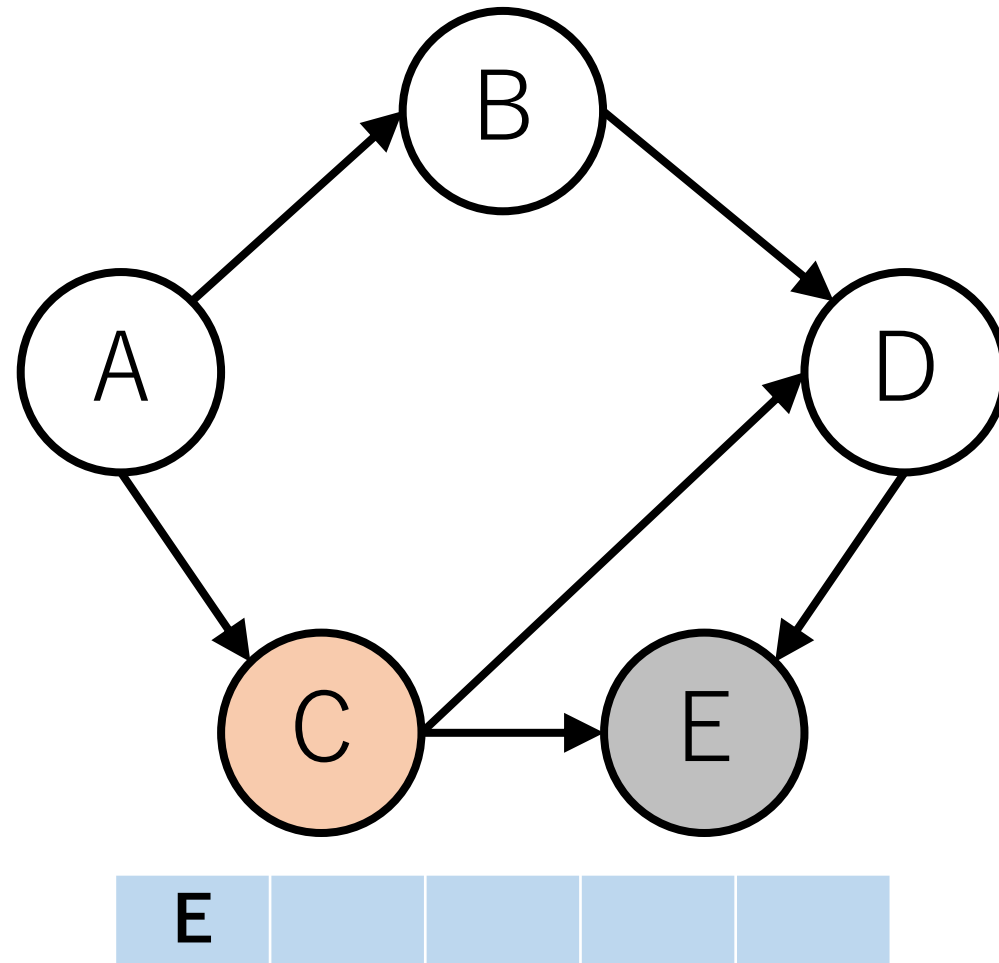
トポロジカルソート (DFS版) の実行例

ただし、**先頭**に追加していく。



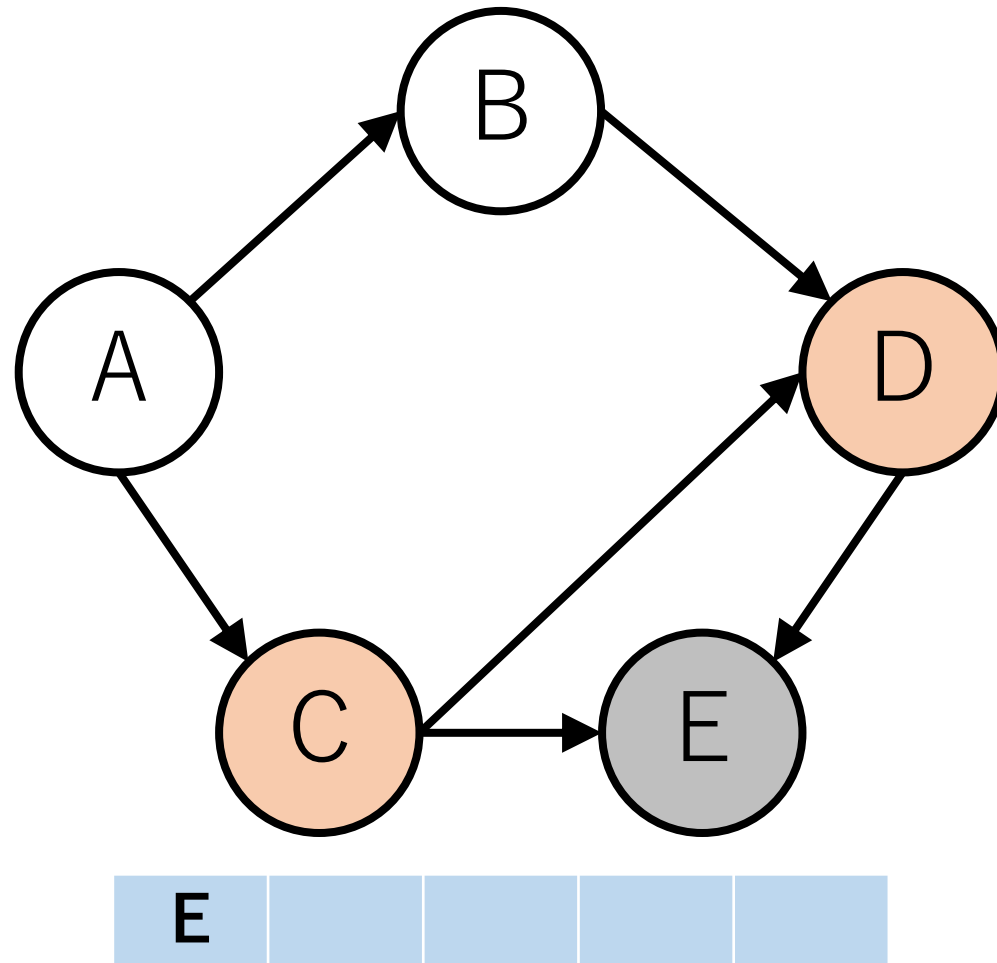
トポロジカルソート (DFS版) の実行例

また、一度でも訪問したノードは訪問済にする。



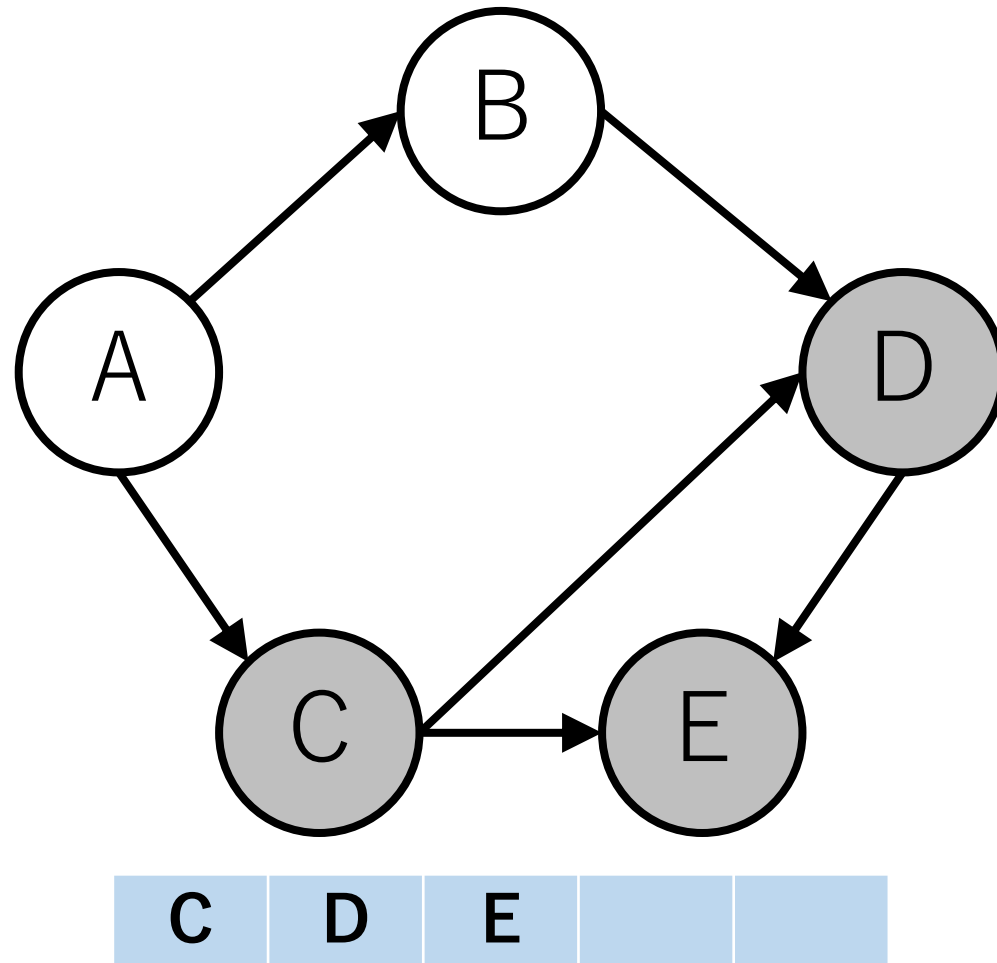
トポロジカルソート (DFS版) の実行例

今回はもう1つDFSで辿ることができるルートがある。



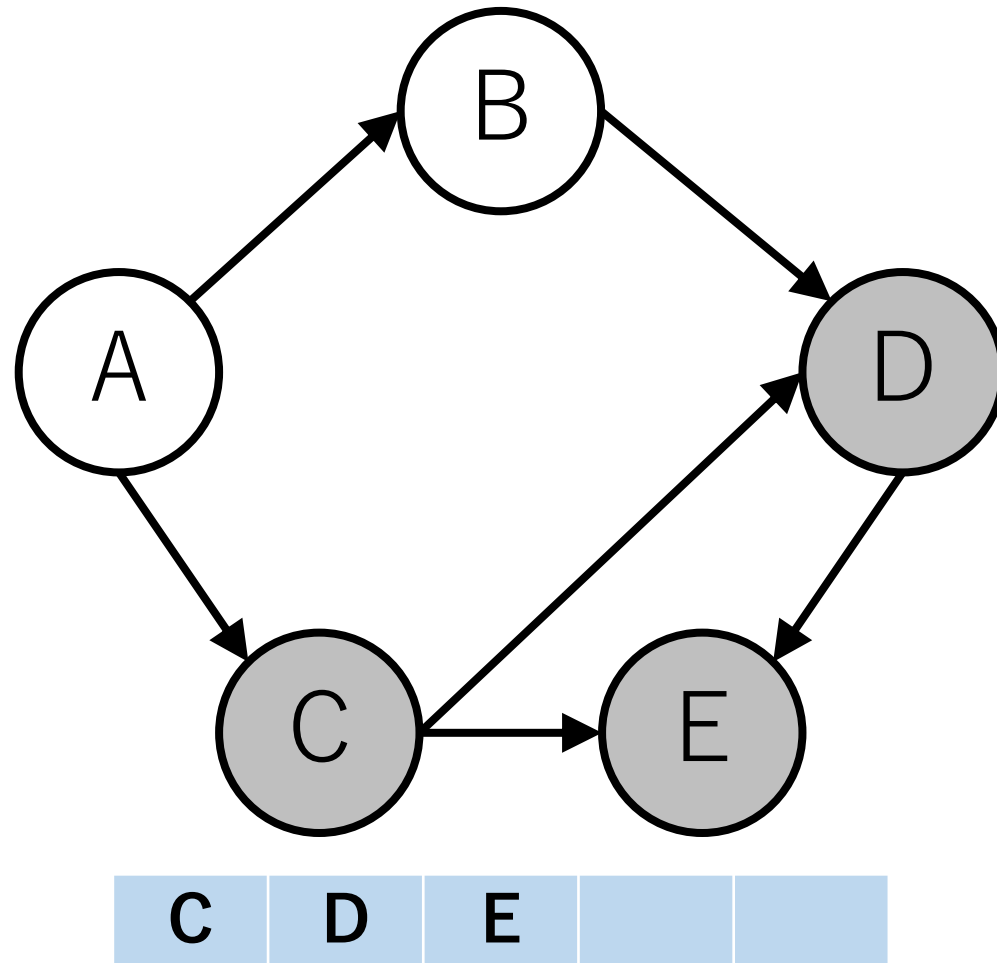
トポロジカルソート (DFS版) の実行例

後戻りしながら、**先頭**に追加していく。



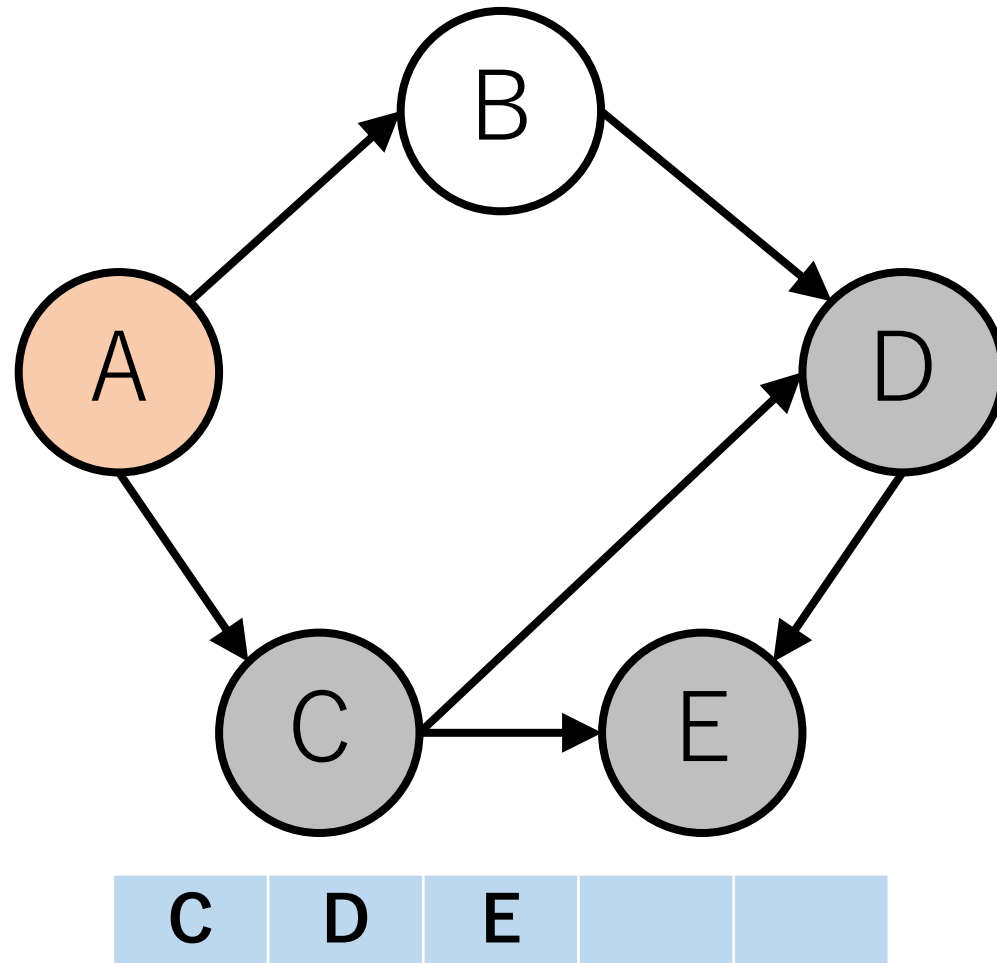
トポロジカルソート (DFS版) の実行例

これ以上戻れないので、未訪問のノードの1つへ移動する。



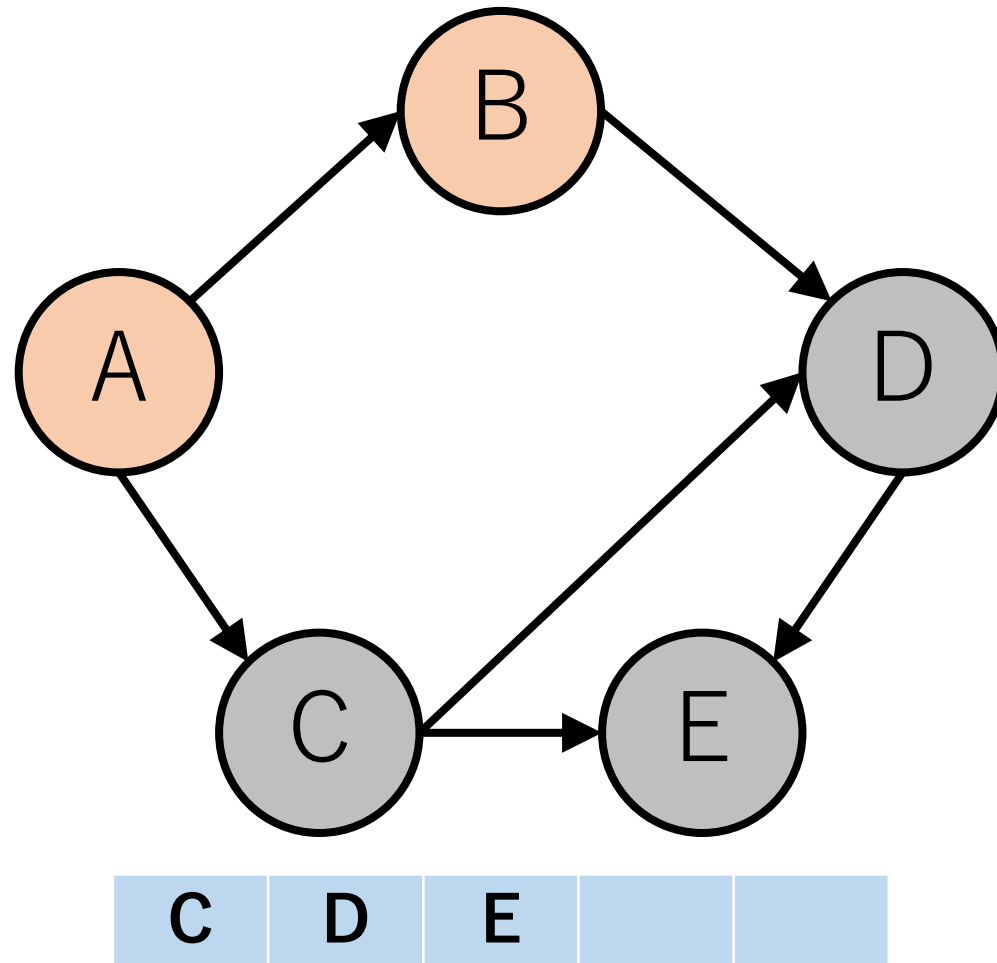
トポロジカルソート (DFS版) の実行例

今回の場合は, ノードAに移動する.



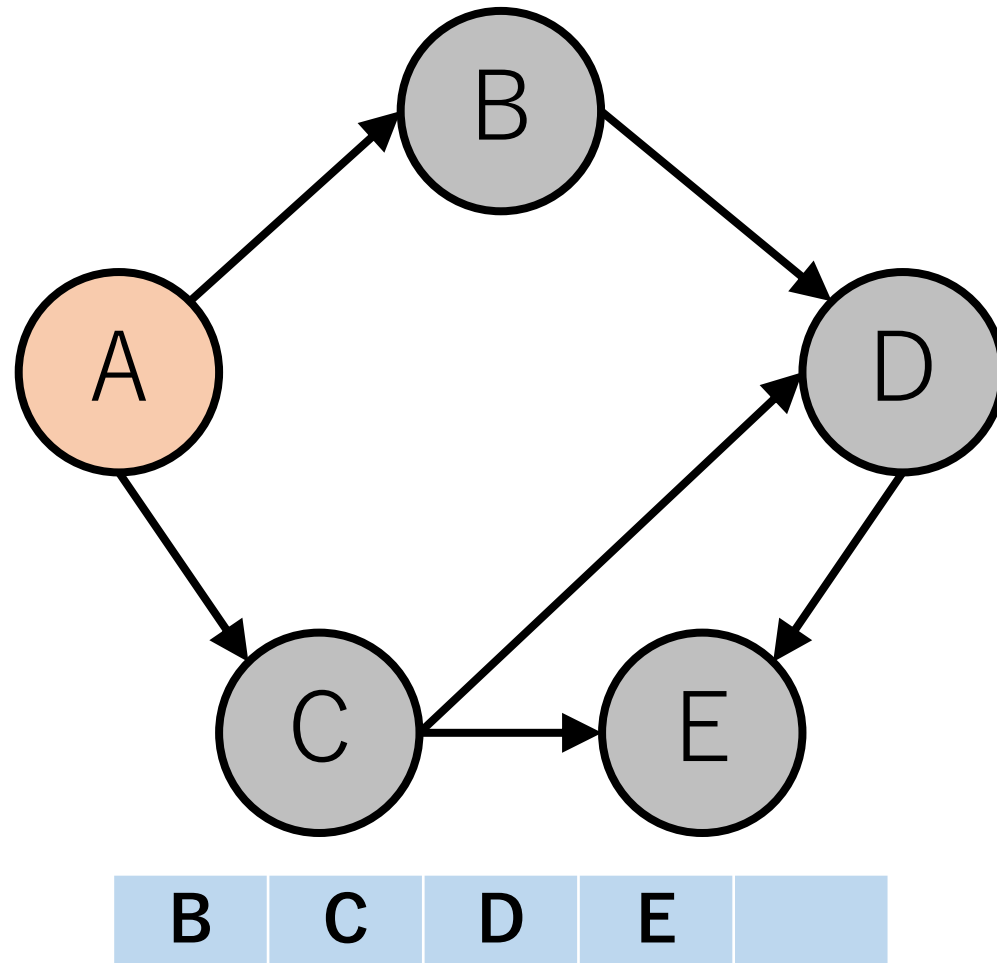
トポロジカルソート (DFS版) の実行例

同様にDFSし，ソート済みに追加していく。



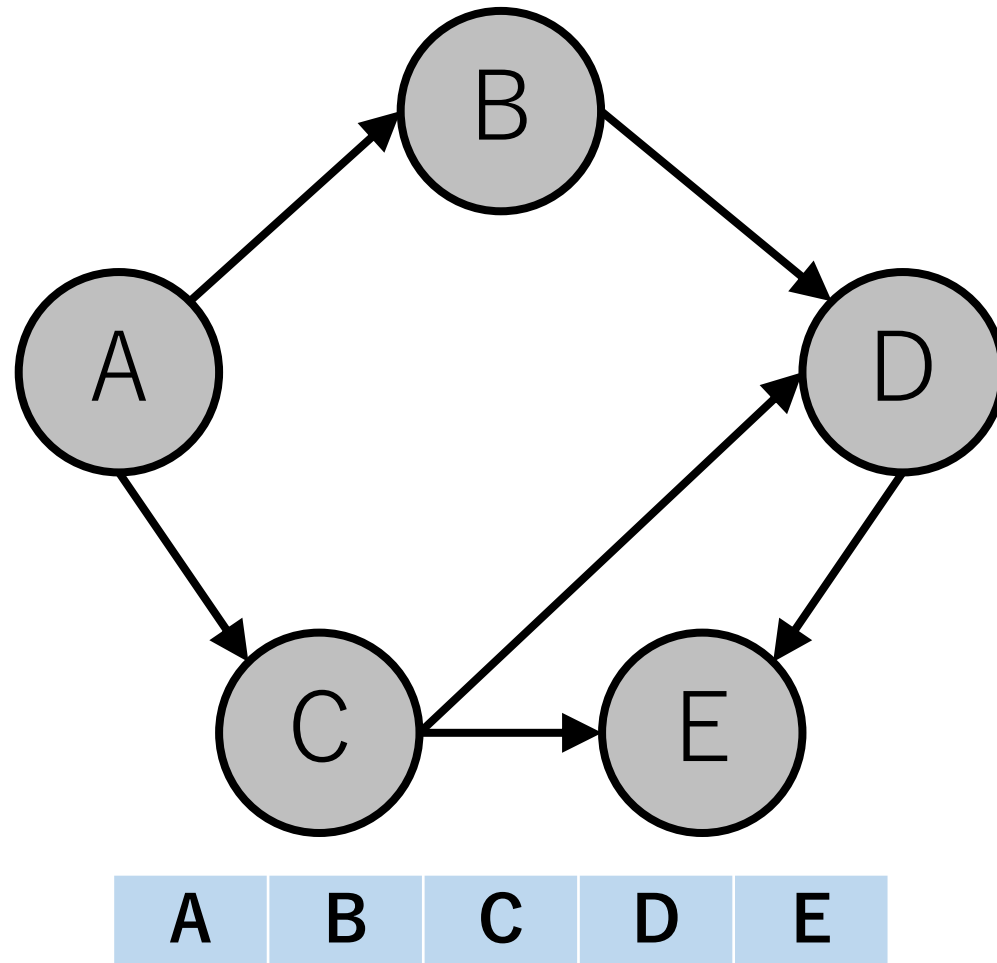
トポロジカルソート (DFS版) の実行例

同様にDFSし，ソート済に追加していく。



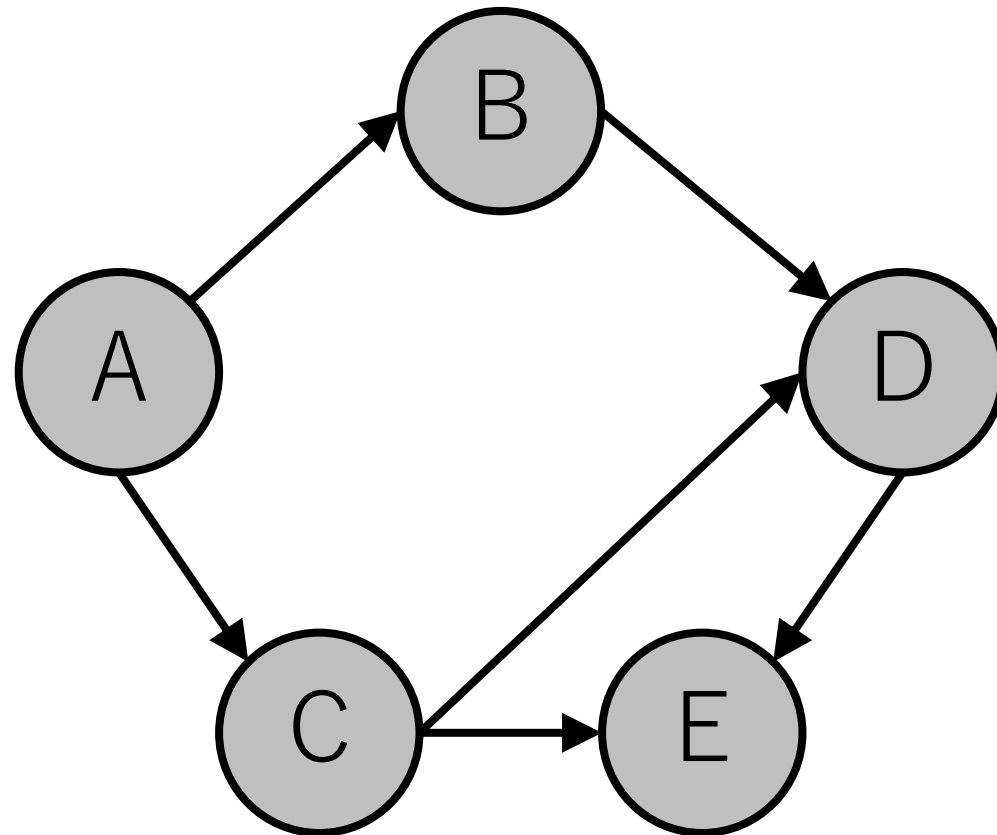
トポロジカルソート (DFS版) の実行例

同様にDFSし，ソート済みに追加していく。



トポロジカルソート (DFS版) の実行例

これを繰り返し、全てノードが訪問済になるまでやる。



トポロジカルソート (DFS版)

DFSを行っている途中で「処理中」のノードを再度訪れることがあった。

現在進行中の探索においてすでに訪れているノードを、再度訪れていることを意味しており、これは閉路があることを示唆している。

DAGになっていないので、エラーを返す。

トポロジカルソートの実装例 (DFS版)

```
def topoSortDFS(V, edges):
```

```
    def check(v):
```

```
        [再帰で呼び出す関数. 後で実装.]
```

```
    # ノードをすでに見たかどうかを格納する配列
```

```
    # 0 : 未訪問, 1 : 処理待ち, 2 : 処理済
```

```
    visited = [0]*V
```

```
    outedge = [[] for _ in range (V)]
```

トポロジカルソートの実装例 (DFS版)

```
def topoSortDFS(V, edges):  
    ...  
    sorted_g = deque()  
  
    # 全てのノードをチェックする  
    for i in range(V):  
        check(i)  
  
    return sorted_g
```

トポロジカルソートの実装例 (DFS版)

```
def check(v):
```

```
    if visited[v] == 1:
```

```
        [DAGになっていないので, エラーを返す]
```

トポロジカルソートの実装例 (DFS版)

```
def check(v):
```

```
    ...
```

```
    elif visited[v] == 0:
```

```
        visited[v] = 1    # 処理待ちにする
```

```
        for to_v in outedge[v]:
```

```
            check(to_v)    # 再帰で呼び出す
```

```
        visited[v] = 2    # 処理済にする
```

```
        sorted_g.appendleft(v)    # ソート済の先頭に追加
```


トポロジカルソート (DFS版) の実行結果例

$V = 5$

edges = [[0, 1], [0, 2], [1, 3], [2, 3], [2, 4], [3, 4]]

```
print(topoSortDFS(V, edges))
```

[0, 2, 1, 3, 4]

トポロジカルソートの計算量

ソートの本質的な部分はKhanのアルゴリズムの場合whileループ、DFS版の場合再帰部分になる。

入力されたグラフがDAGである場合、全てのノードと辺は高々1回しかチェックされない。

よって、 $O(|V| + |E|)$.

トポロジカルソートの応用例

依存関係を調べて整理することに使われる。

プロジェクト内の実行タスク順序

ビルドにおけるライブラリの依存関係

今日のまとめ

最小全域木

クラスカル法, Union Find木
プリム法

トポロジカルソート

Khanのアルゴリズム
DFSベースのアルゴリズム

コードチャレンジ：基本課題#11-a [1.5点]

スライドで説明したUnion-Find木を使って、
クラスカル法を実装してください。

Union-Find木を使っていない実装は認められない
ので、注意してください。

コードチャレンジ：基本課題#11-b [1.5点]

Khanのトポロジカルソートを実装してください。結果は辞書順最小になるようにしてください。

DFSを用いるトポロジカルソートは認められません。

deque, heapqを使用しても構いません。

コードチャレンジ：Extra課題#11 [3点]

本日勉強したアルゴリズムに関する問題.