Algorithms (2020 Summer)
#10：グラフアルゴリズム1

矢谷 浩司

今日の問題：最短経路問題

与えられたグラフの辺には一定値でないコスト（距離など）が予め設定されている.

- コストが一定値ならばBFSでよい.
- コストは負であることもある.

グラフのあるノードから別のノードへの繋がるパスにおいて，コストが最小（距離が最短など）になるようなパスを選ぶ.

最短経路問題の種類

2頂点対最短経路問題

特定の2つのノード間の最短経路を求める.

単一始点最短経路問題

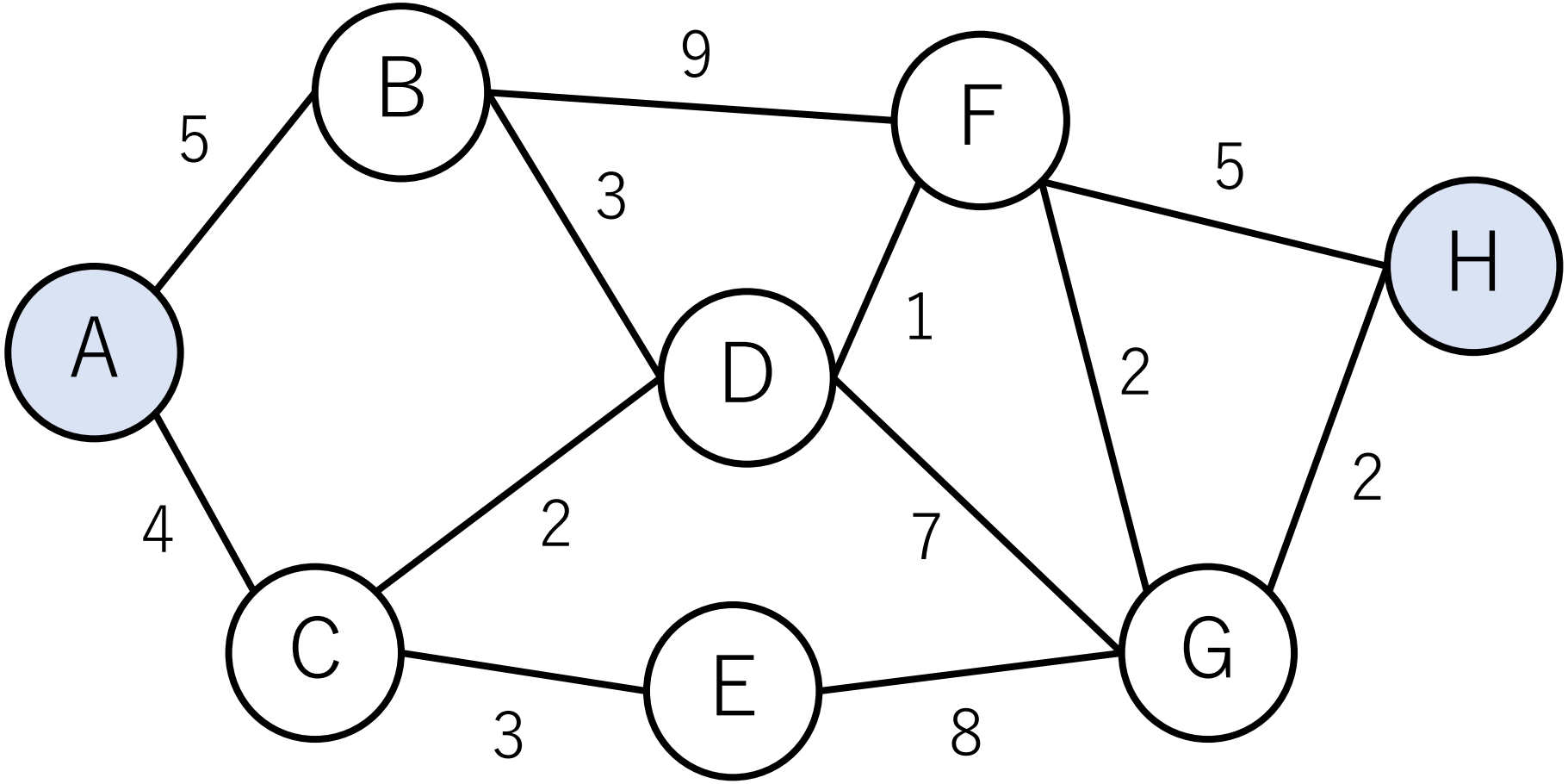
ある始点ノードから他の全部のノードへの最短経路を求める.

全点対最短経路問題

すべての2ノード間の最短経路を求める.

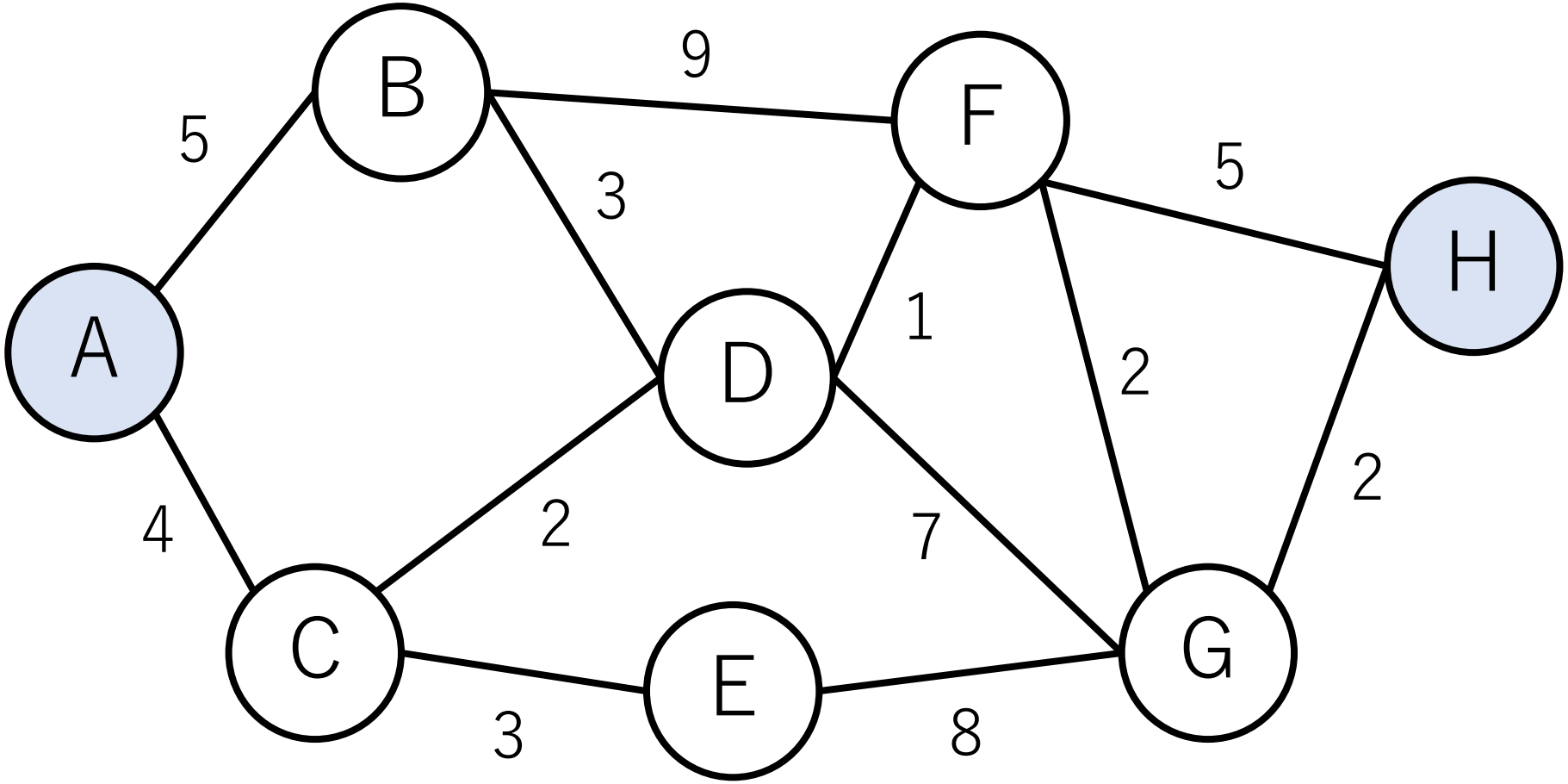
最短経路問題の例（2頂点対最短経路問題）

辺にその距離（コスト）が紐付けされている.



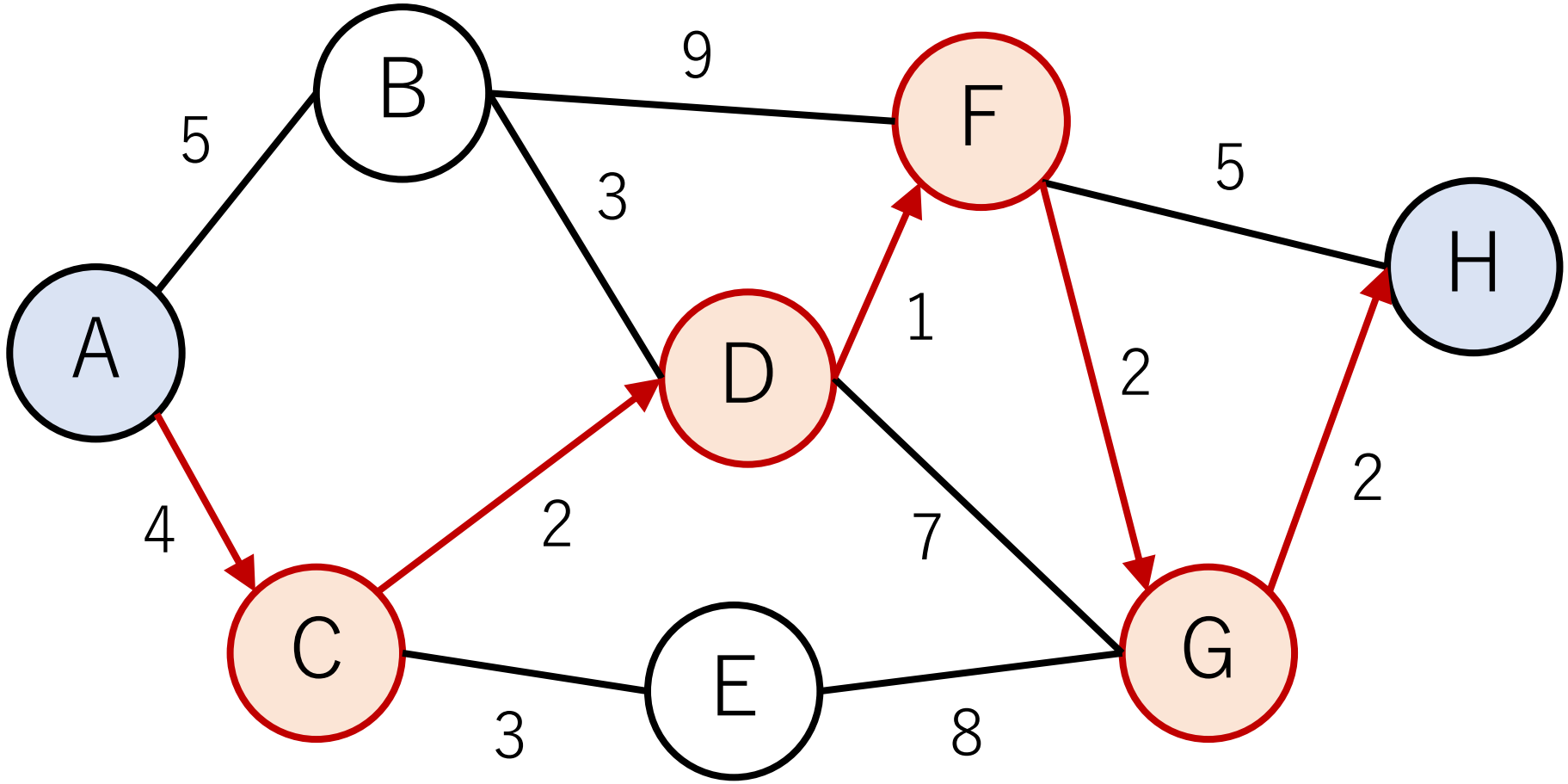
最短経路問題の例（2頂点対最短経路問題）

このグラフにおけるAからHへの最短経路は？



最短経路問題の例（2頂点対最短経路問題）

赤色になっているパスで，距離は11.



最短経路問題の例（2頂点対最短経路問題）

距離がバラバラなのでBFSは使えない．．．

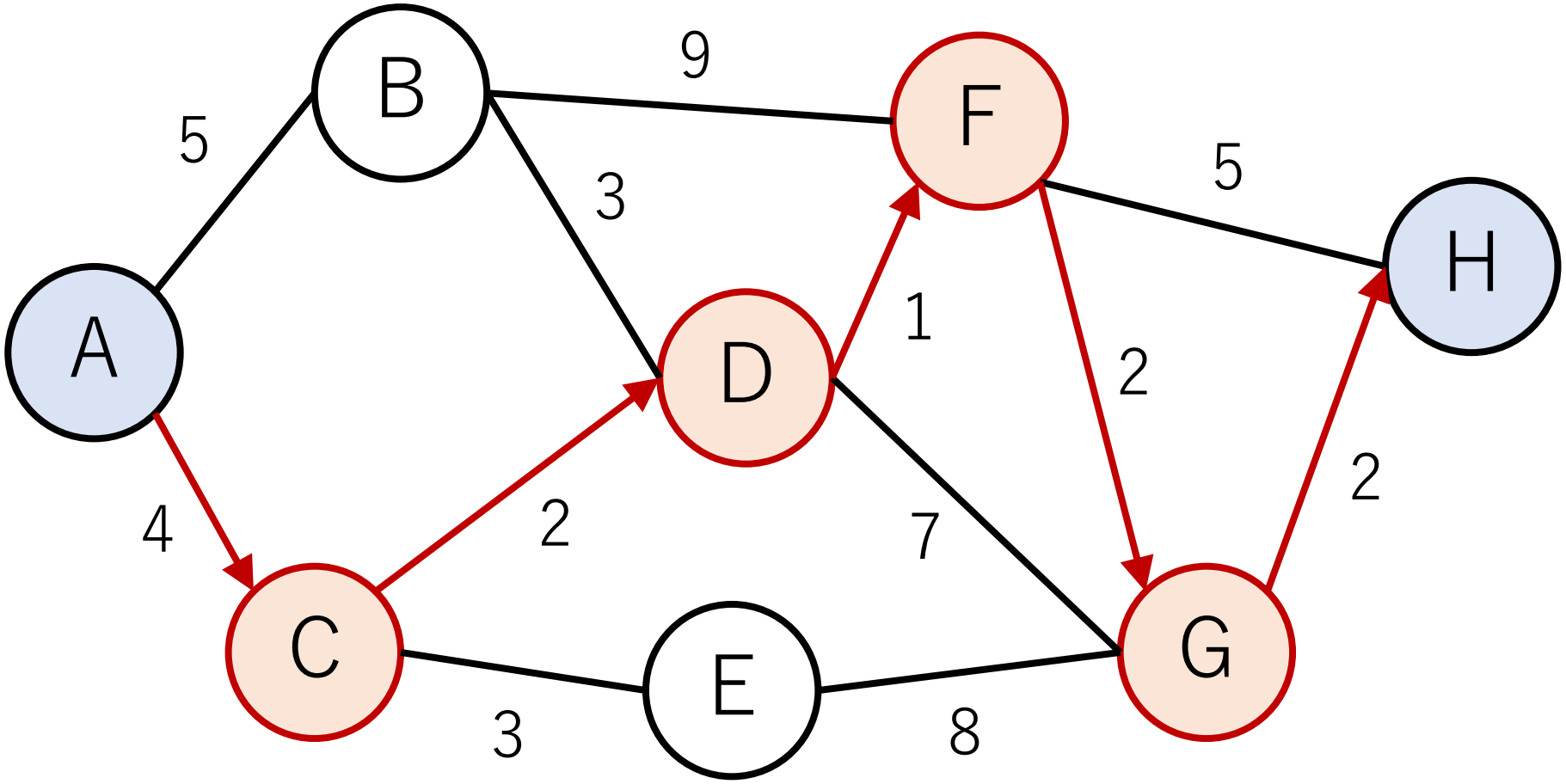
どのノードを通るかを総当りでチェックすると$O(|V|!)$になりシャレにならない．．．

2頂点対最短経路問題と単一始点最短経路問題

2頂点対最短経路問題は，単一始点最短経路問題の
アルゴリズムを利用して解くのが一般的.

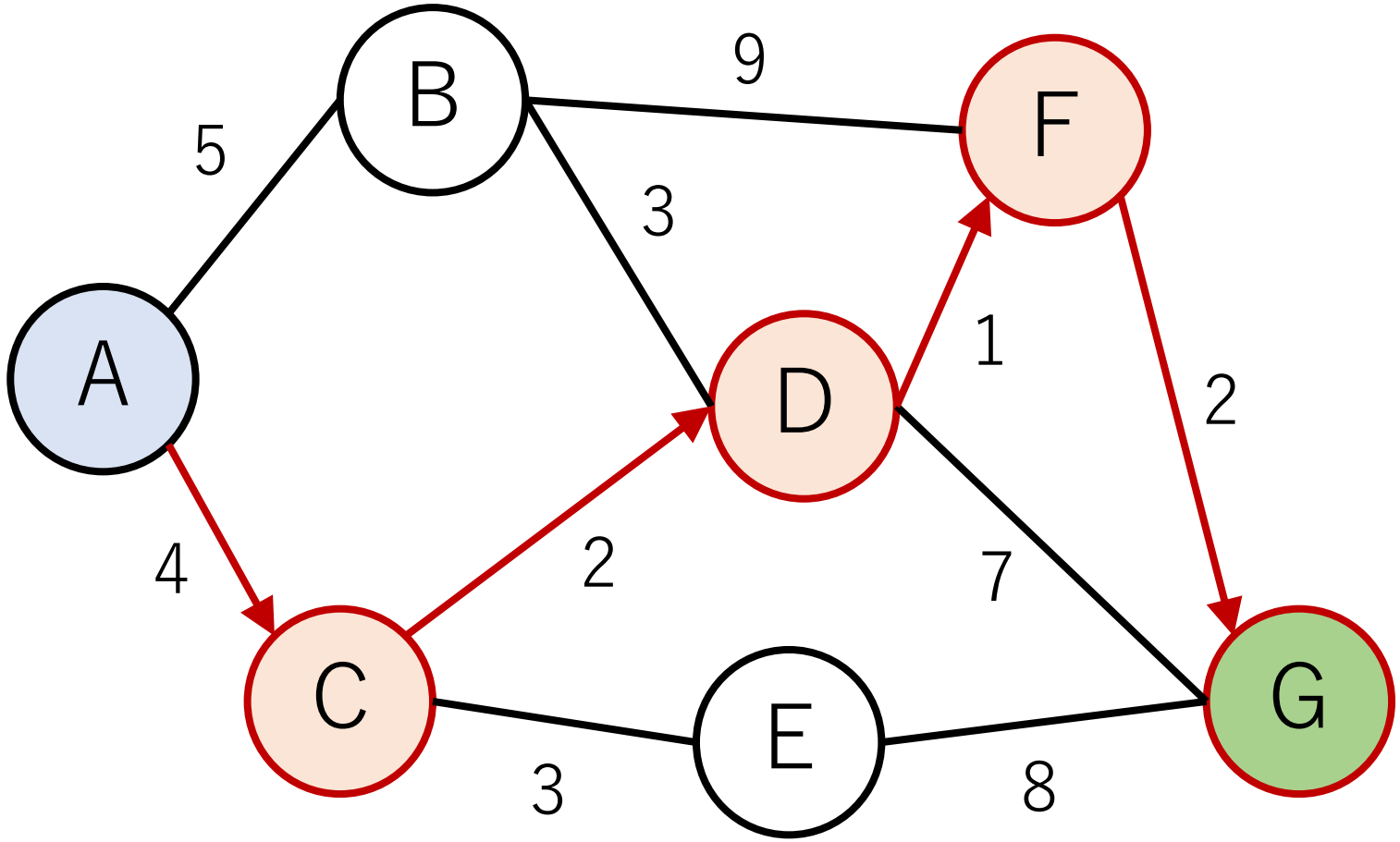
まずは，単一始点最短経路問題を解くアルゴリズムに
ついて見てきましょう！

最短経路をよく見てみよう



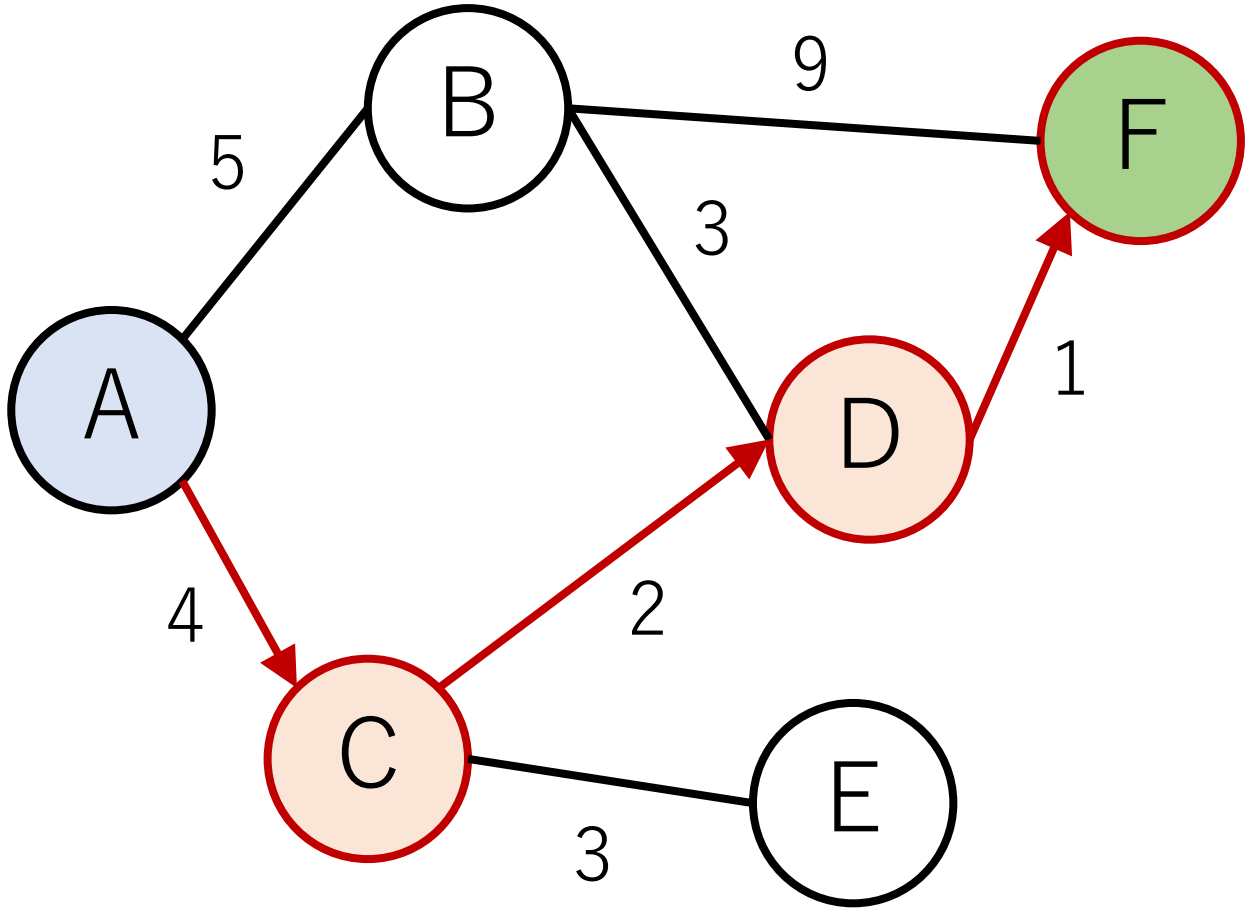
最短経路をよく見てみよう

もしHではなく，Gがゴールだとすると？



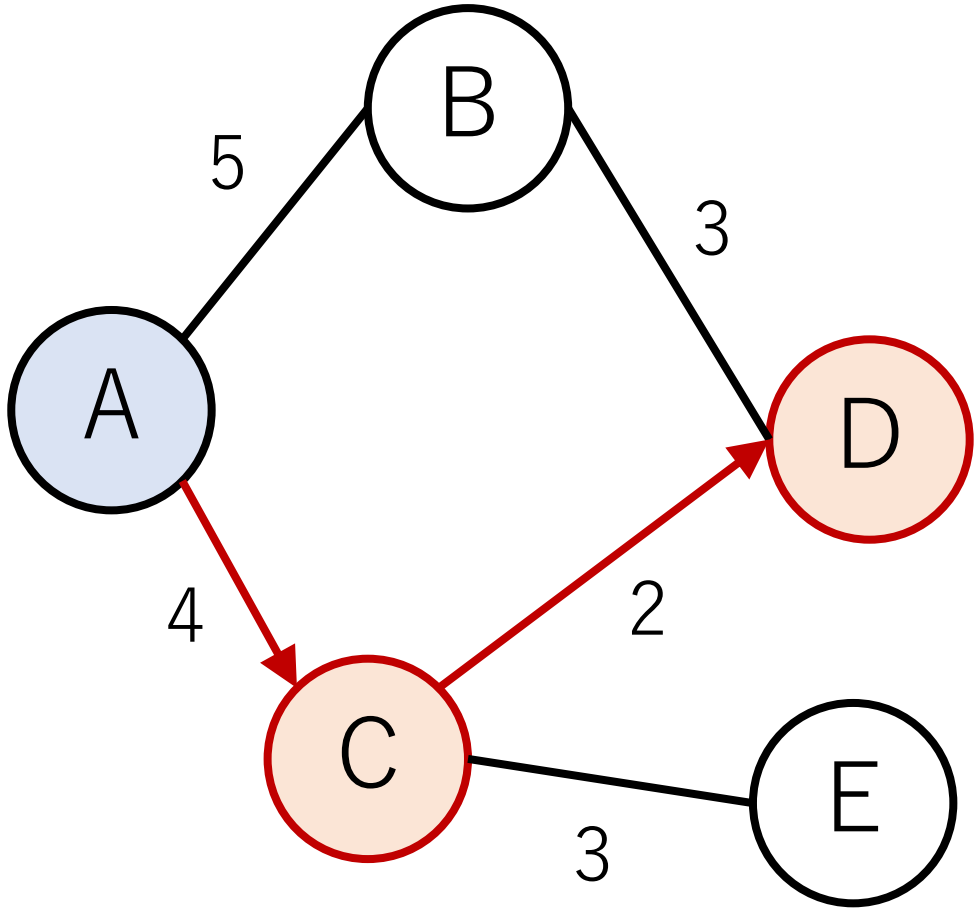
最短経路をよく見てみよう

もしFがゴールだとすると？



最短経路をよく見てみよう

もしDがゴールだとすると？



最短経路をよく見てみよう

元々の最短経路の手前のノードで止めた時でも，その時点での最短経路になっている．

重要な性質：「最短経路の部分経路も最短経路」

Pを最短経路として，その部分経路をP'とする．もし，P'より短い経路Q'があるとすると，それを使った全体の最短経路Qを考えることができる．しかしQが存在するなら，Pは最短経路ではなくなるので，矛盾する．

解き方の手がかり

あるノードまでの最短経路は，その1つ前までの（複数の）ノードのうち，最短で繋がるものだけ取り出せば良い．

つまり，各ノードに今までの最短経路の情報を保持しておけばよい！

ダイクストラ（Dijkstra）法

まだ距離が完全に確定していないノードのうち，
最短の距離になっているノードiを選ぶ．

ノードiにつながっているノードの距離を更新する．

更新が終わるとノードiを確定済とする．この時点で
ノードiまでの最短距離は確定となる．

これを全ての頂点の最短距離が確定するまで行う．

ダイクストラ（Dijkstra）法

変数

edge[“X->Y”]: ノードXからYに行く距離

dist[“X”]: ノードXの現在までの最短距離

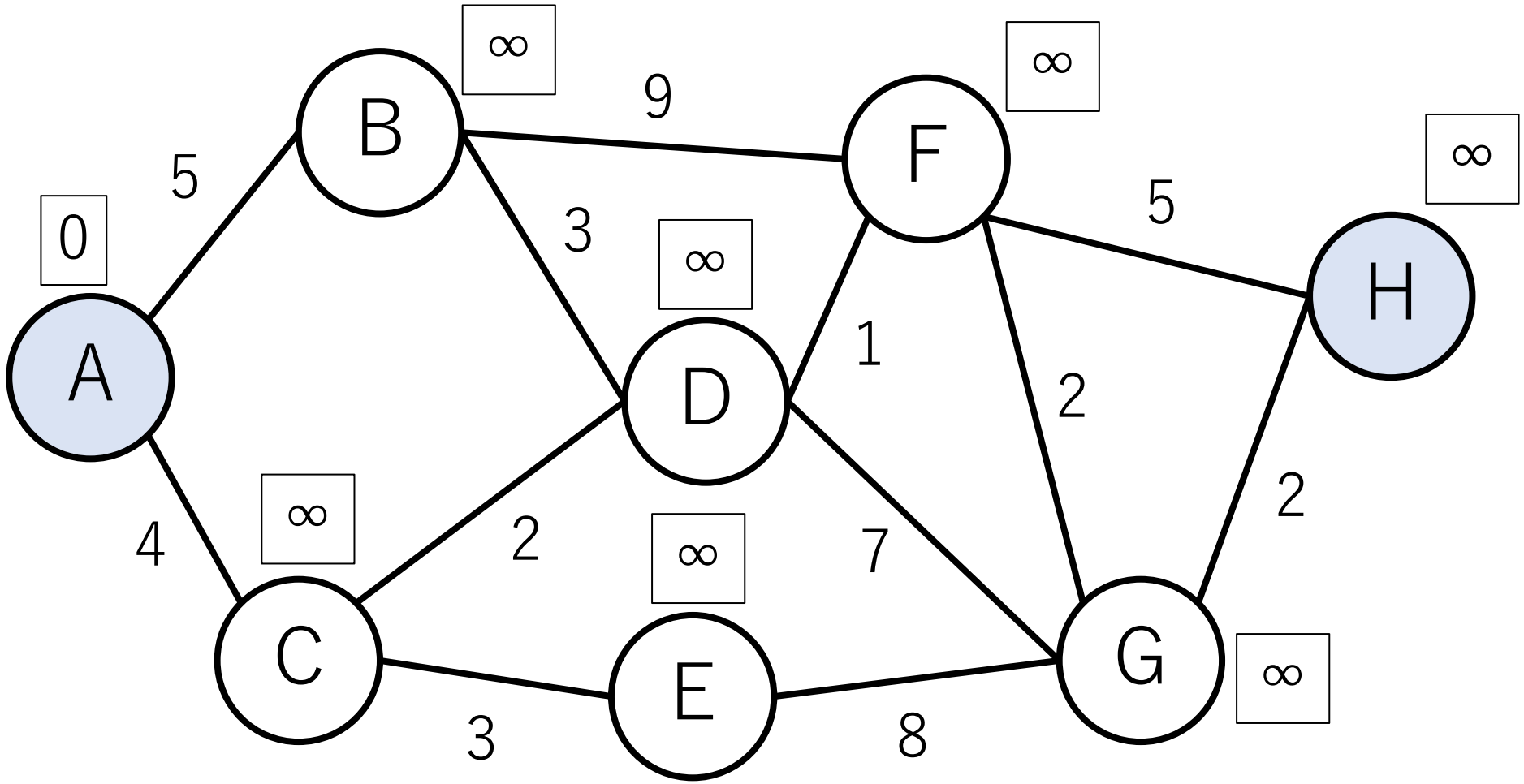
初期化

dist[“開始ノード”] <- 0

dist[“それ以外”] <- ∞

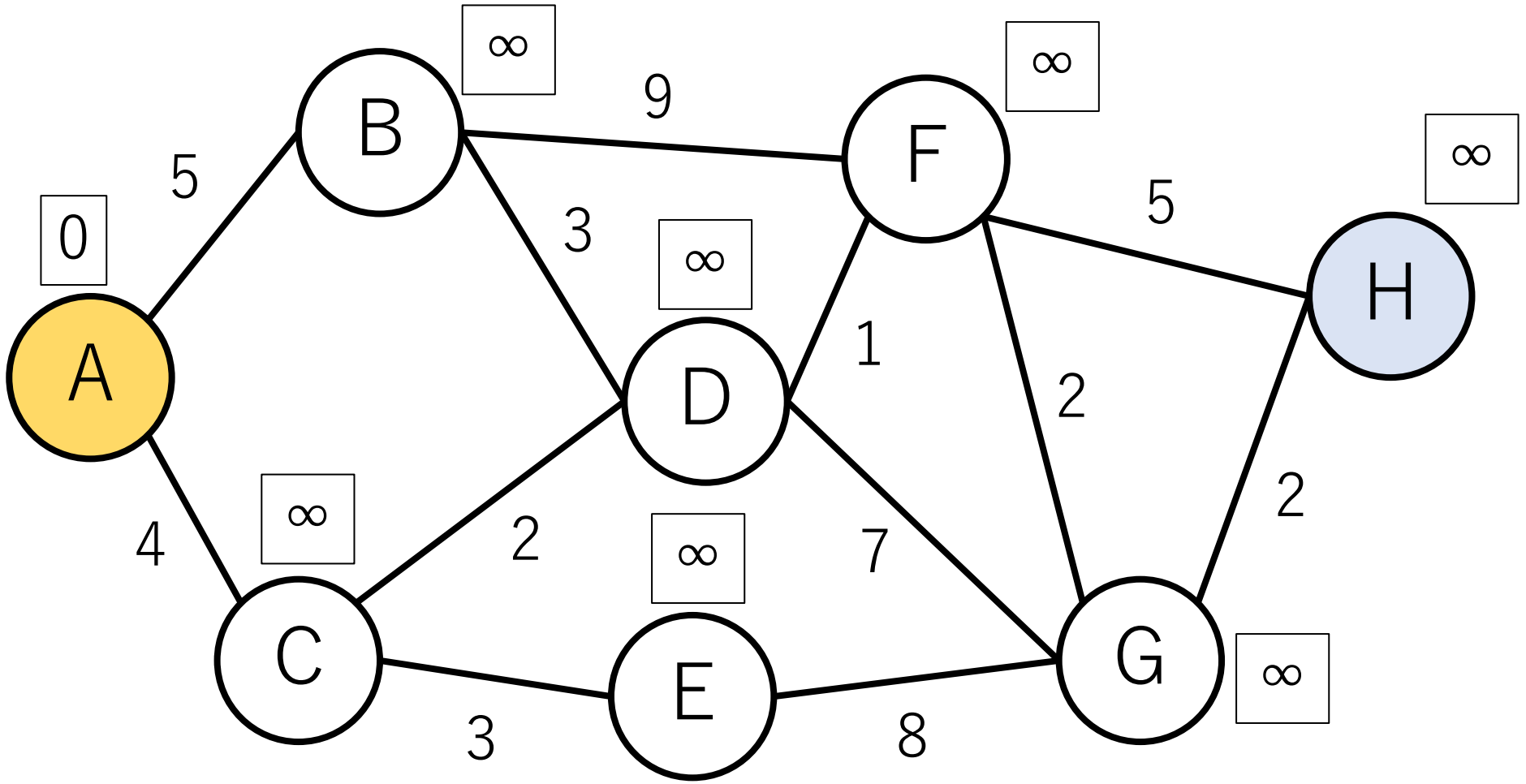
ダイクストラ法の例

初期化後の状態.



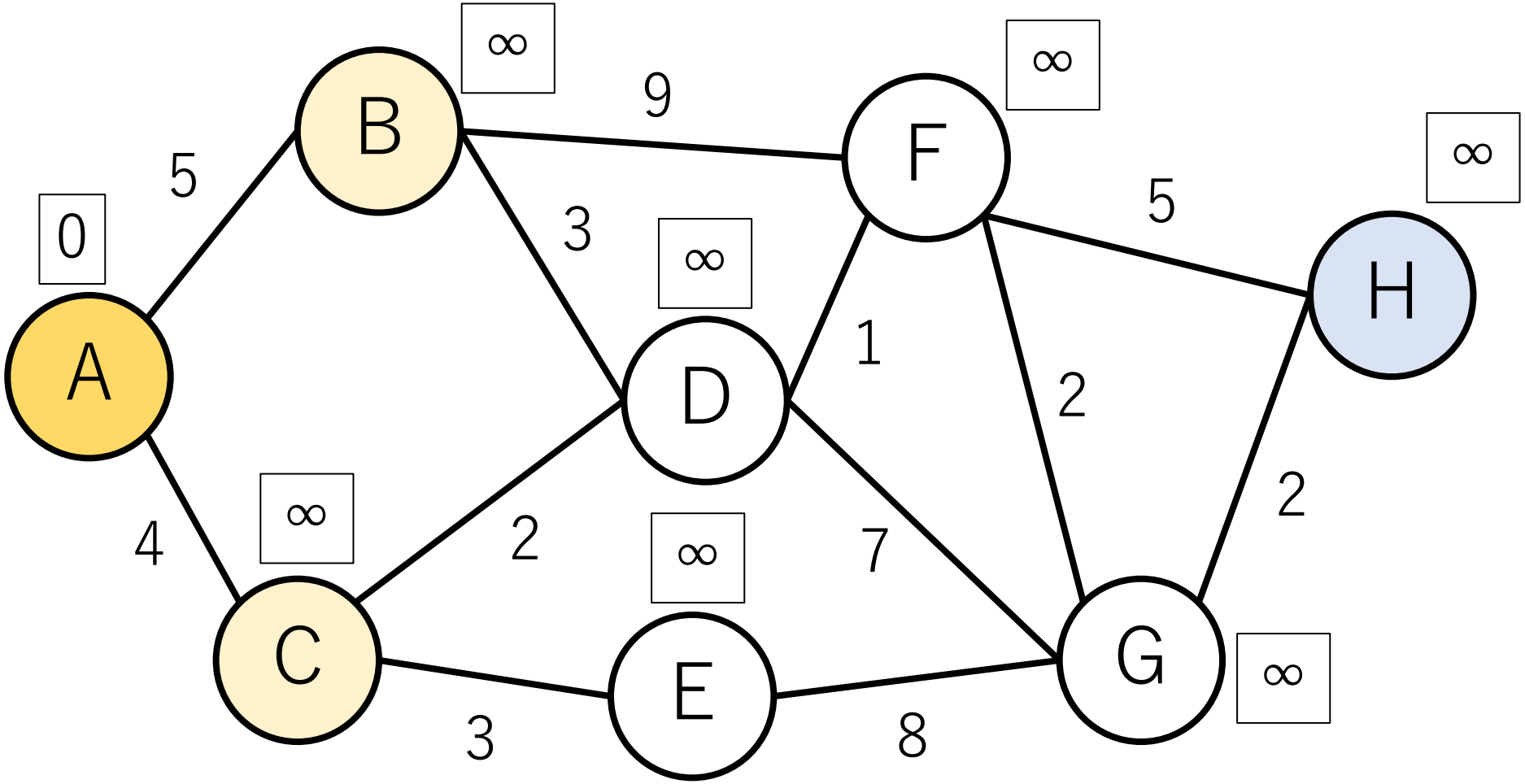
ダイクストラ法の例

Aからスタート.



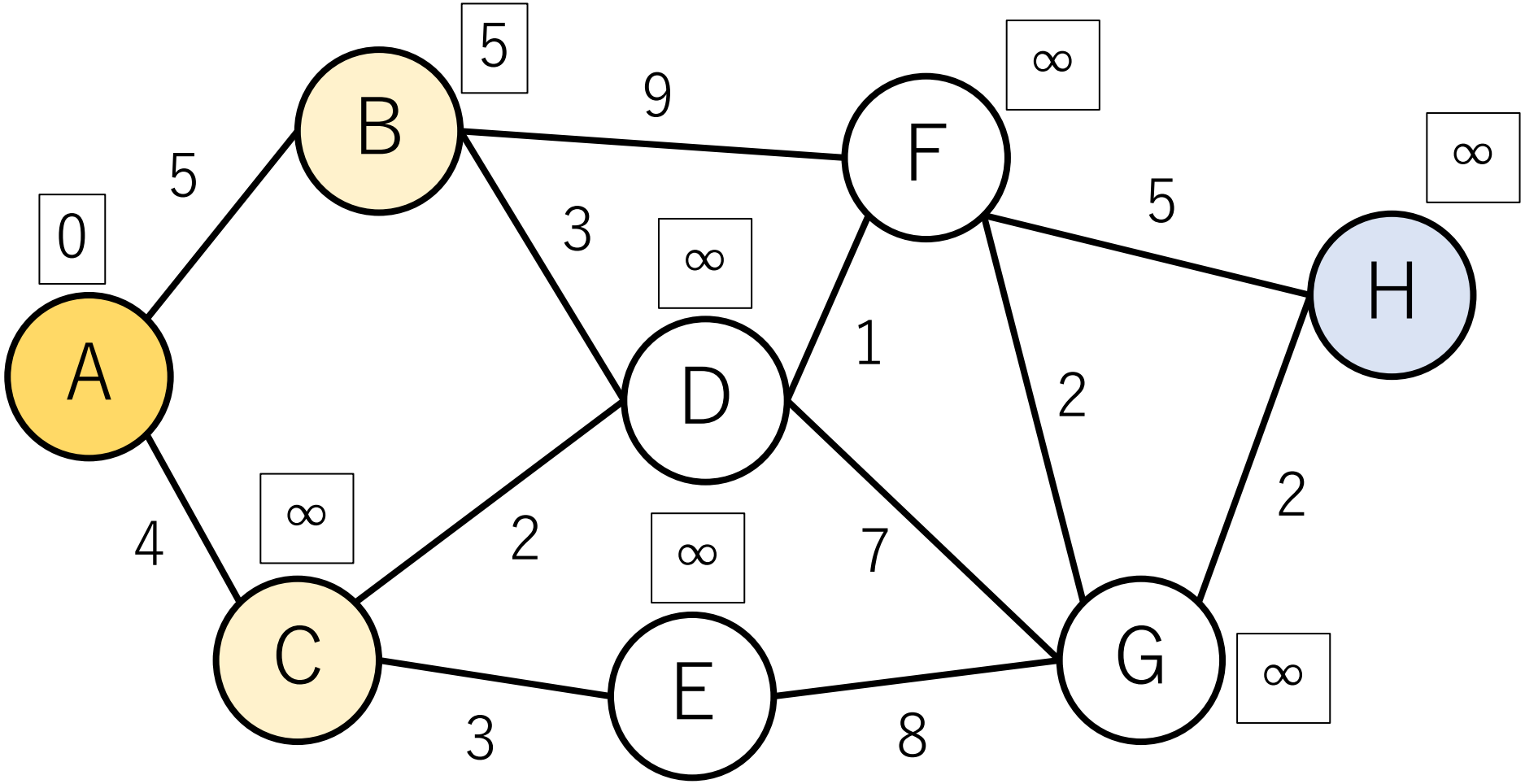
ダイクストラ法の例

繋がっているノードに対して，最短距離を更新.



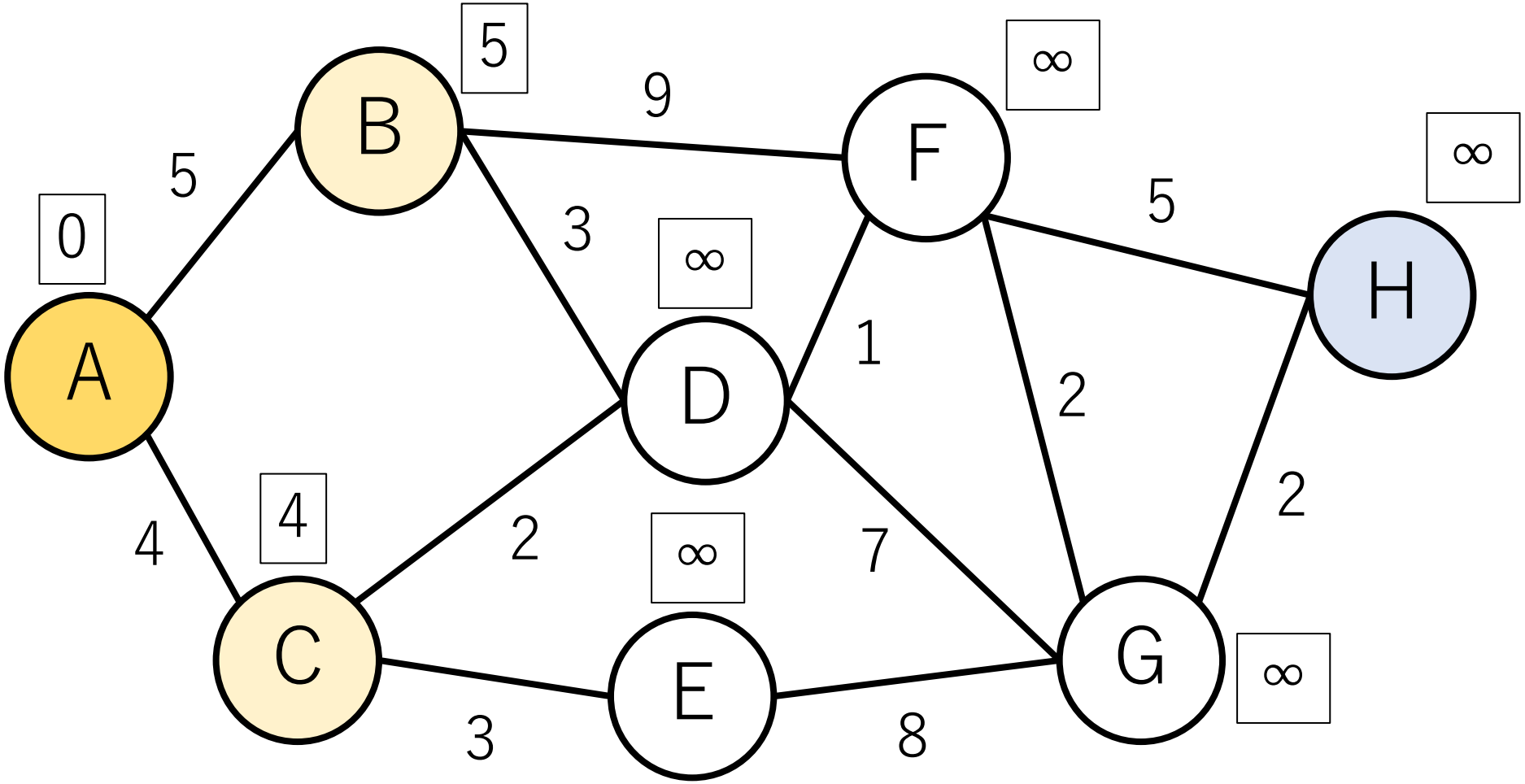
ダイクストラ法の例

dist[“B”] <- dist[“A”] + edge[“A->B”]



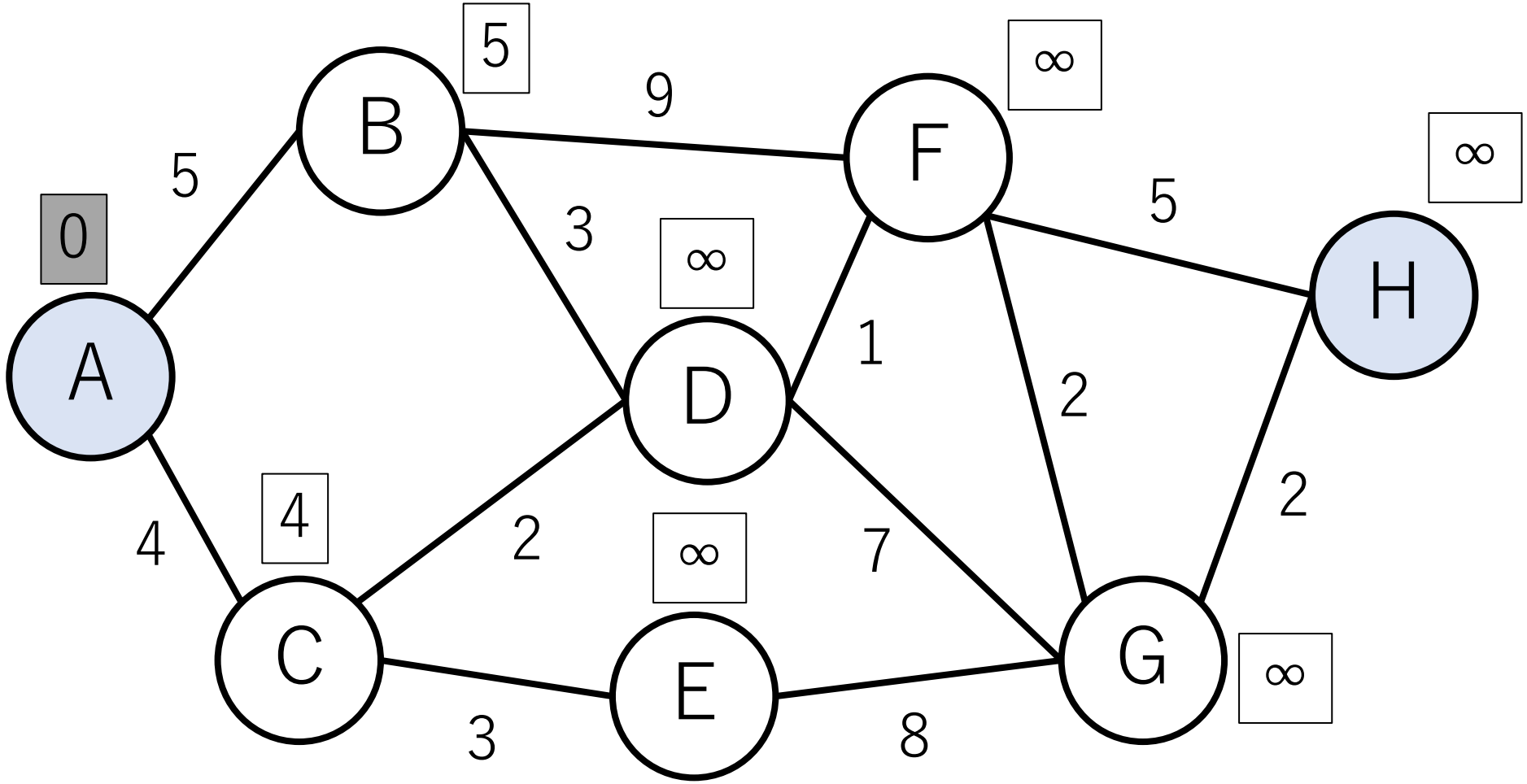
ダイクストラ法の例

dist[“C”] <- dist[“A”] + edge[“A->C”]



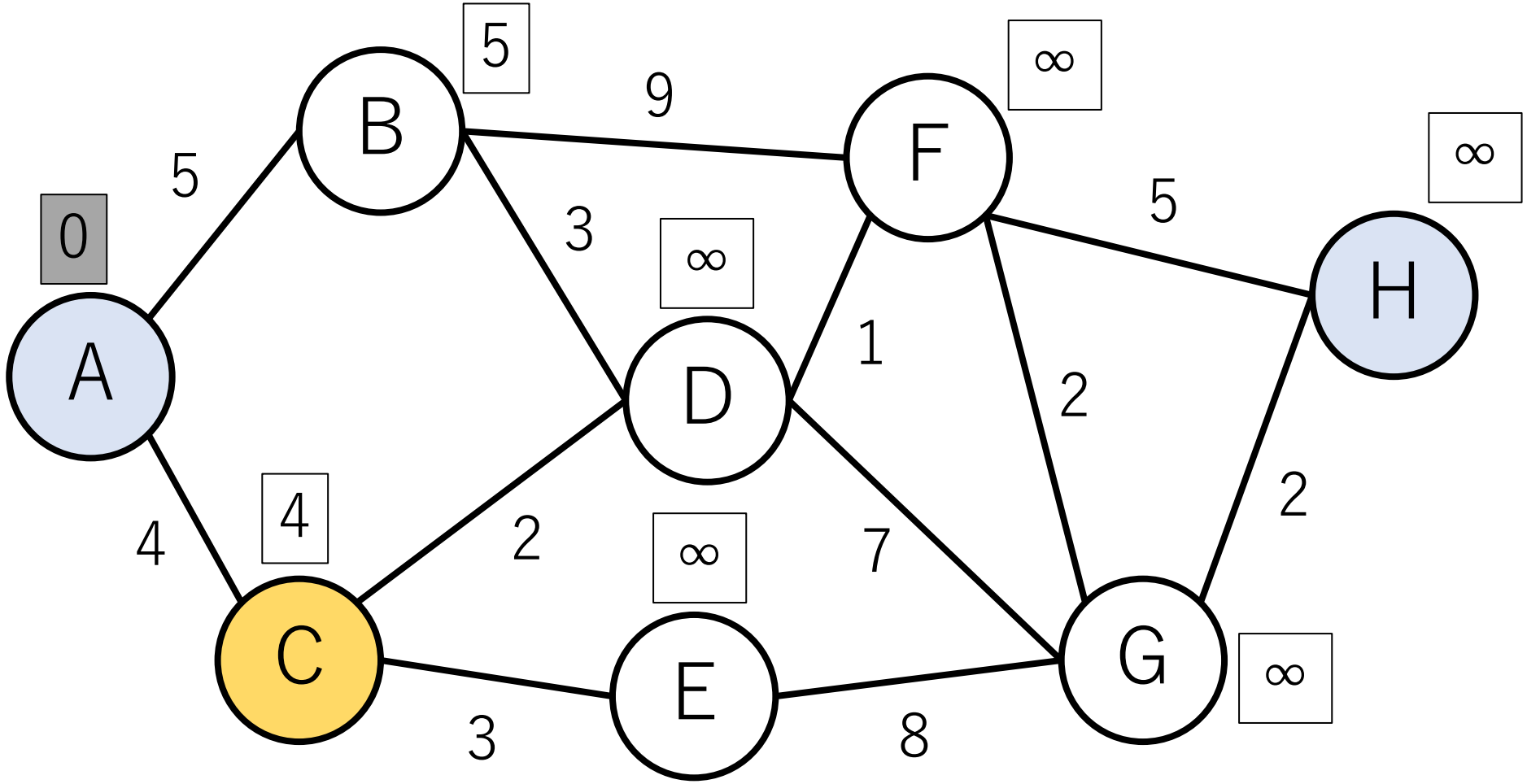
ダイクストラ法の例

ノードAから伸びるパスは全部見たので，Aは終了.



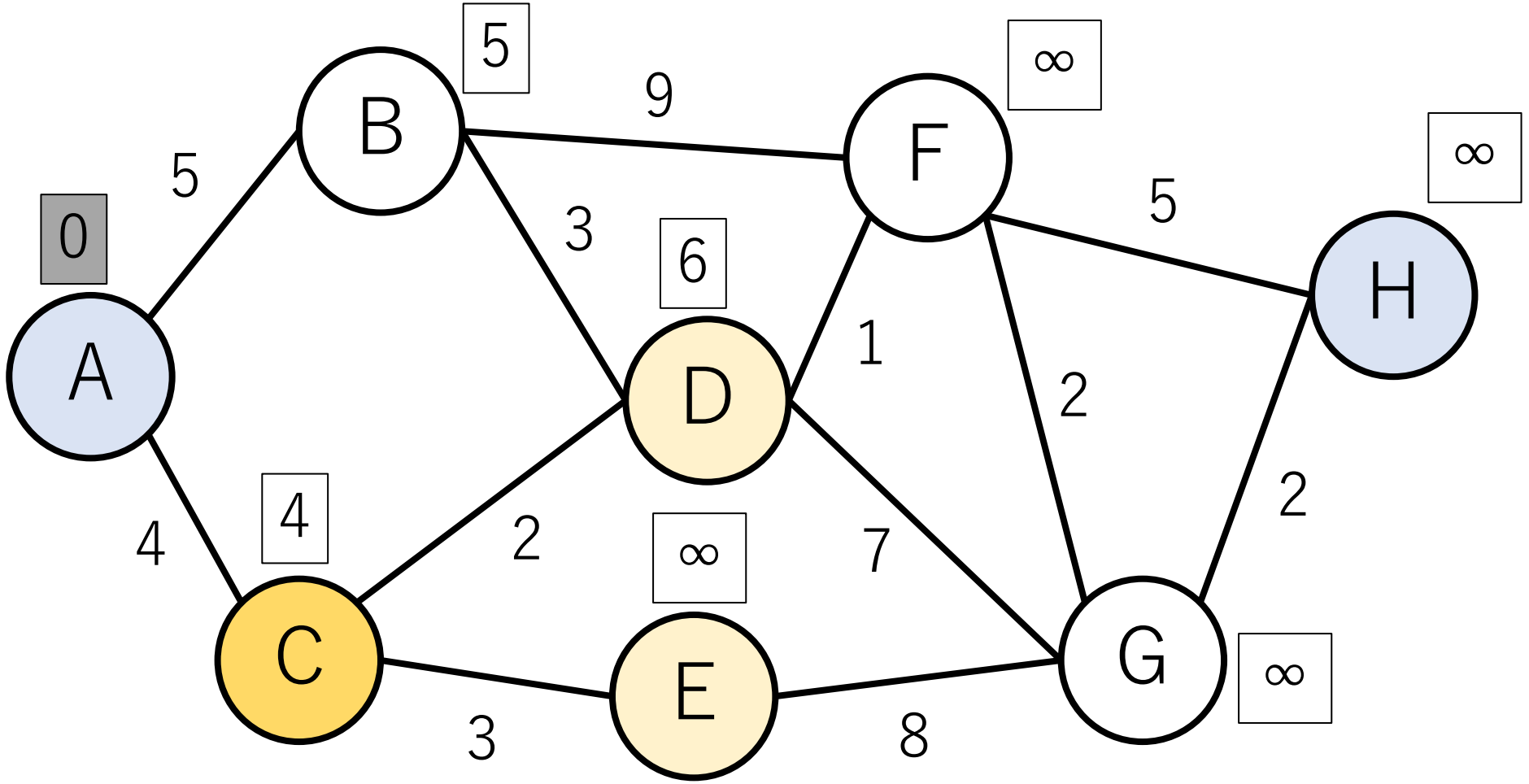
ダイクストラ法の例

現在の最短距離のノードはCなので，Cをみる．



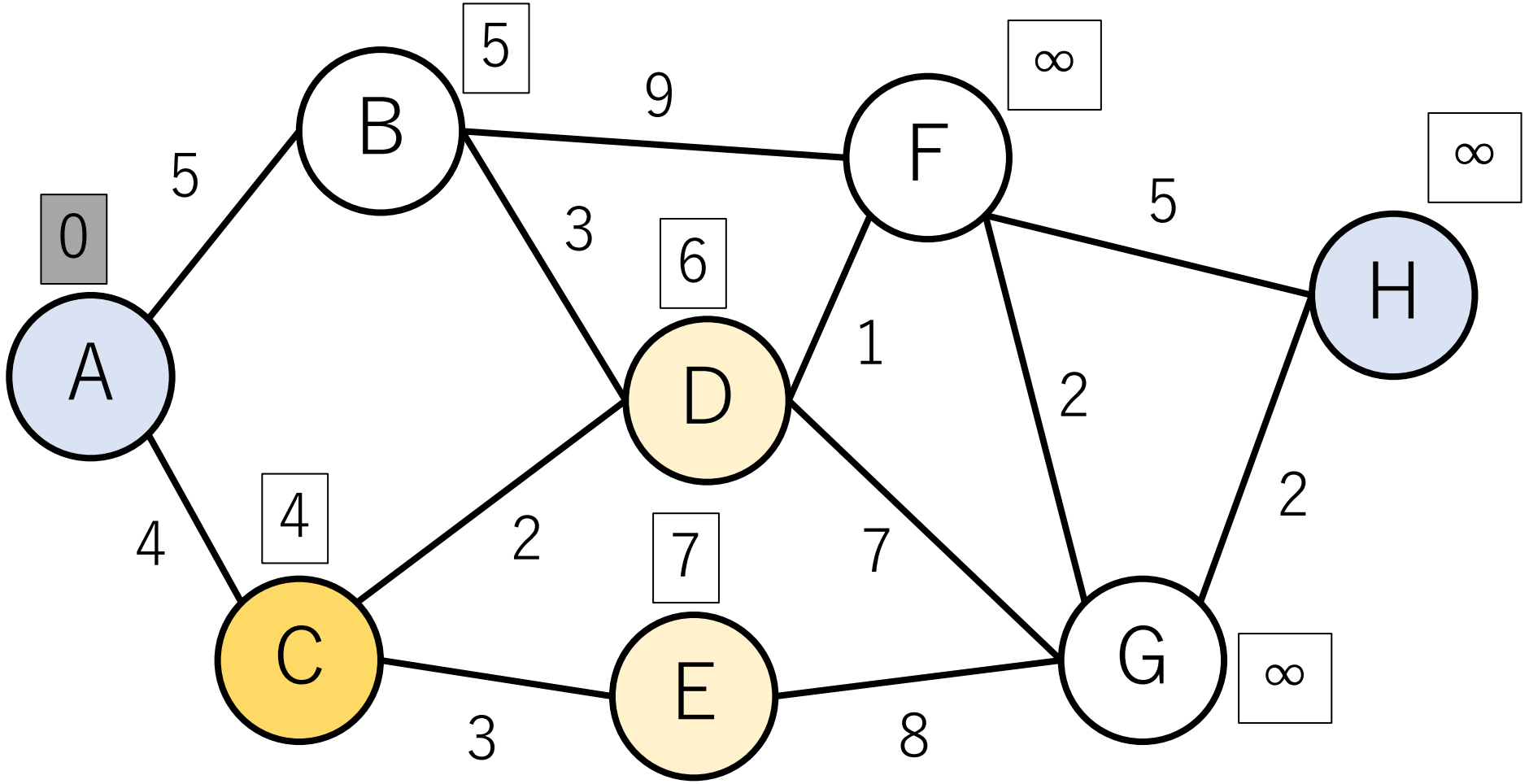
ダイクストラ法の例

dist[“D”] <- dist[“C”] + edge[“C->D”]



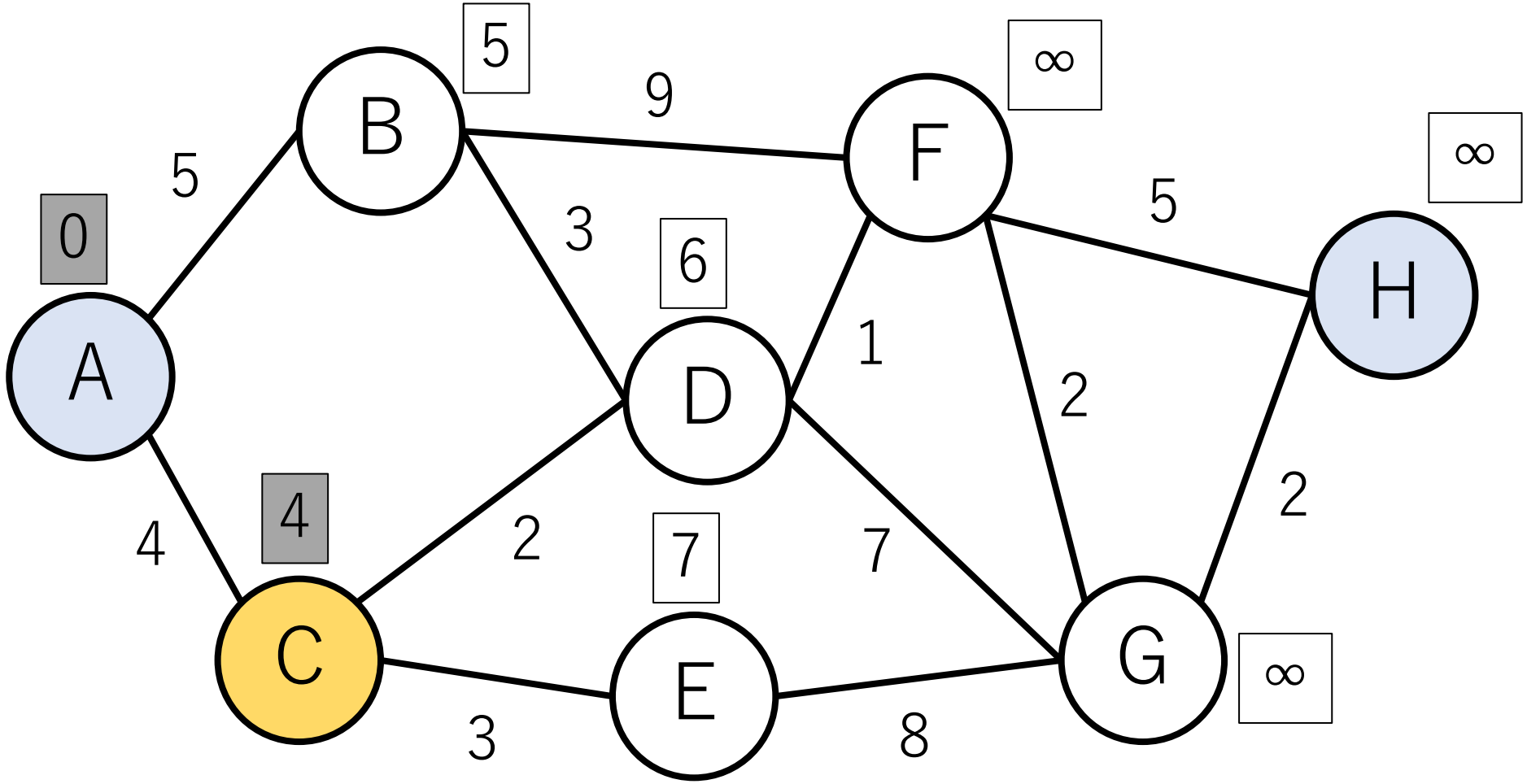
ダイクストラ法の例

dist[“E”] <- dist[“C”] + edge[“C->E”]



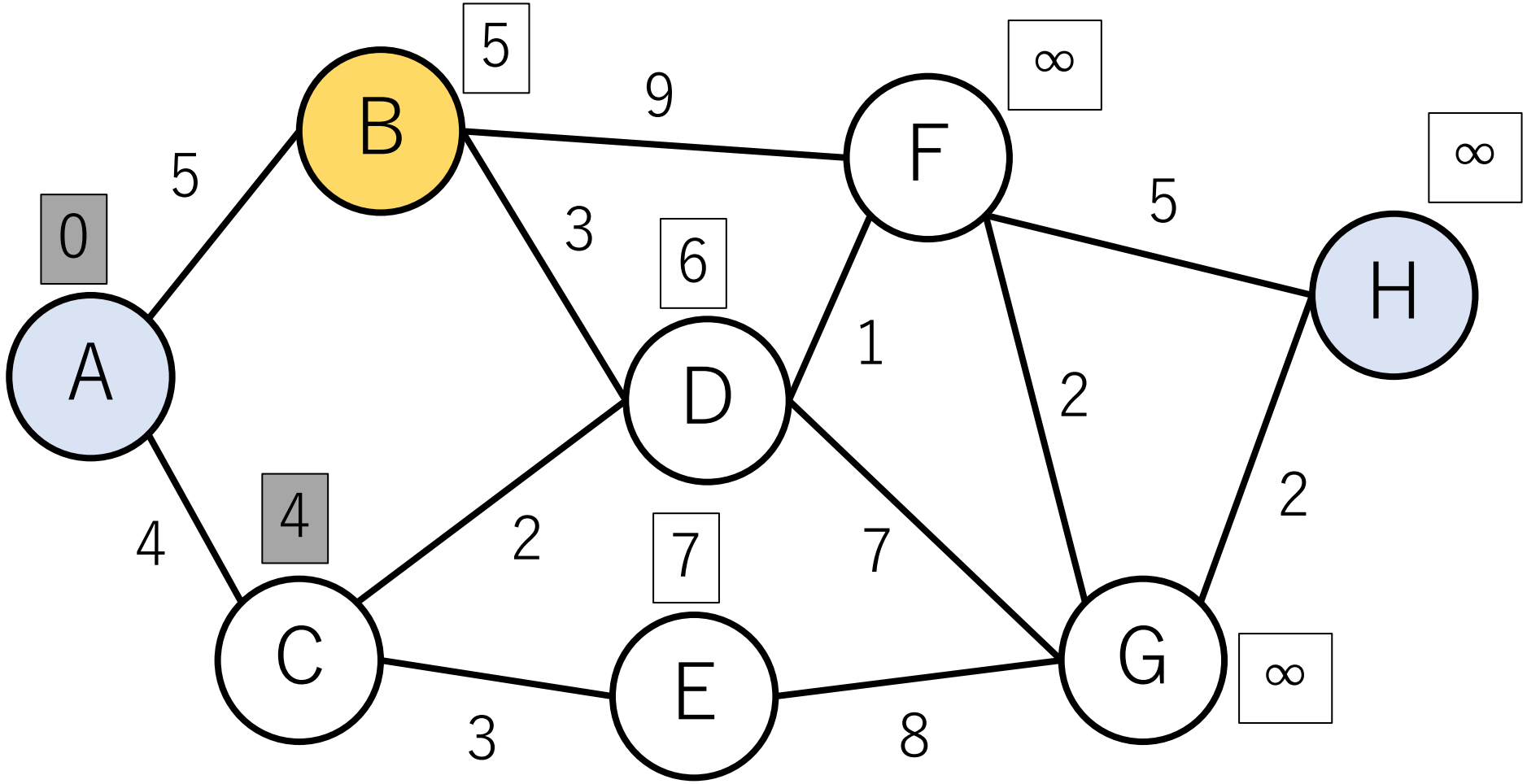
ダイクストラ法の例

C終わり.



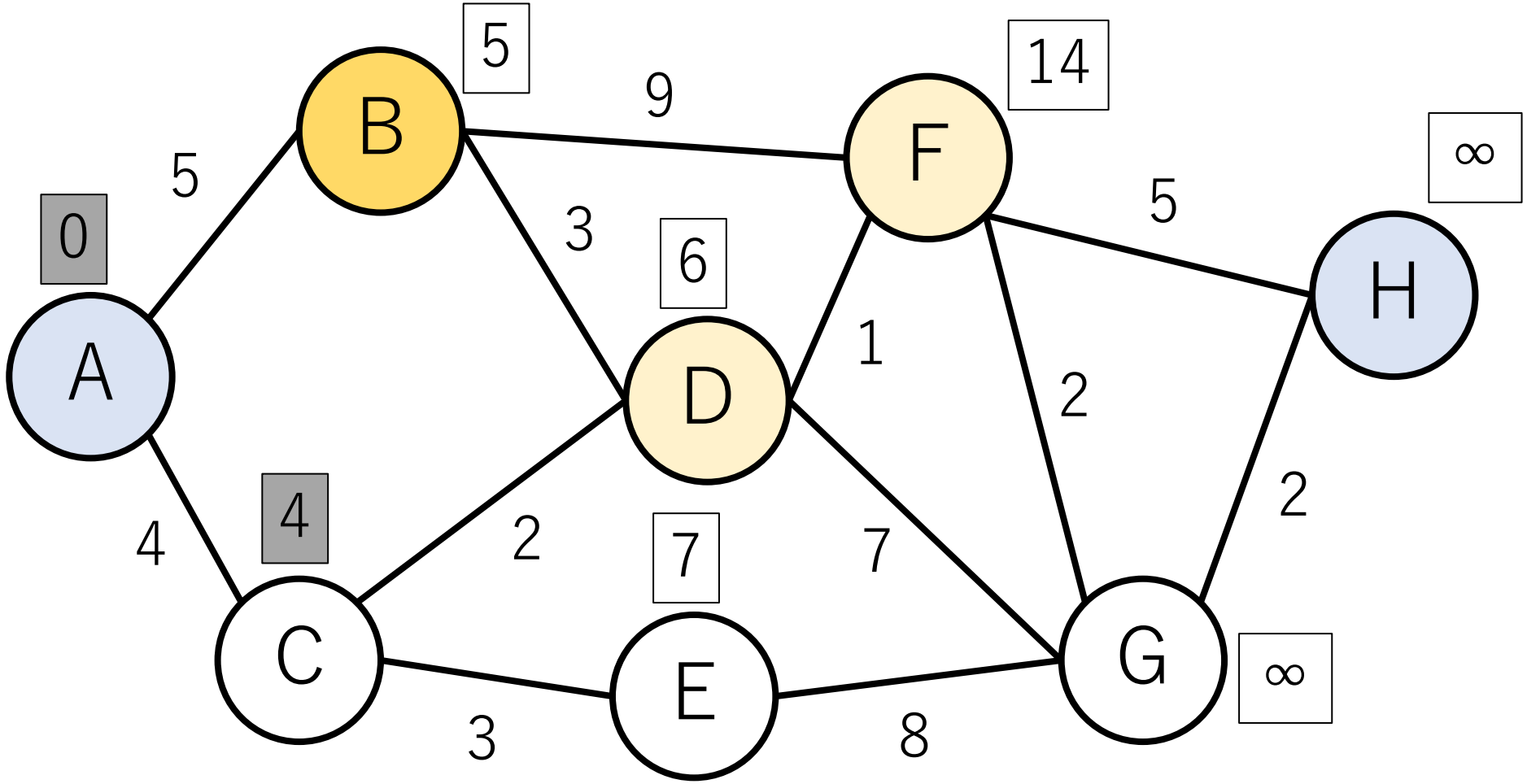
ダイクストラ法の例

最短距離になっているのはB.



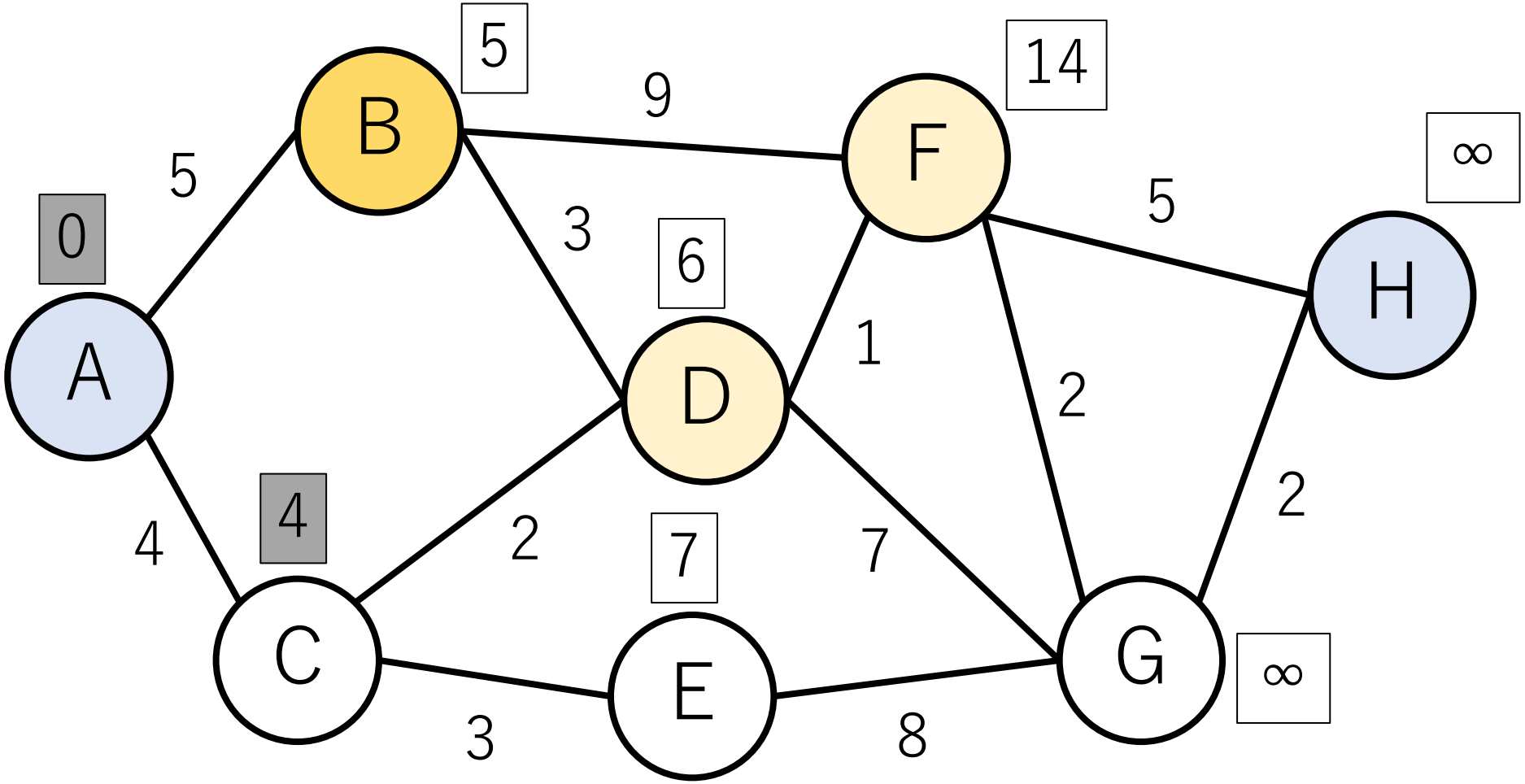
ダイクストラ法の例

dist[“F”] <- dist[“B”] + edge[“B->F”]



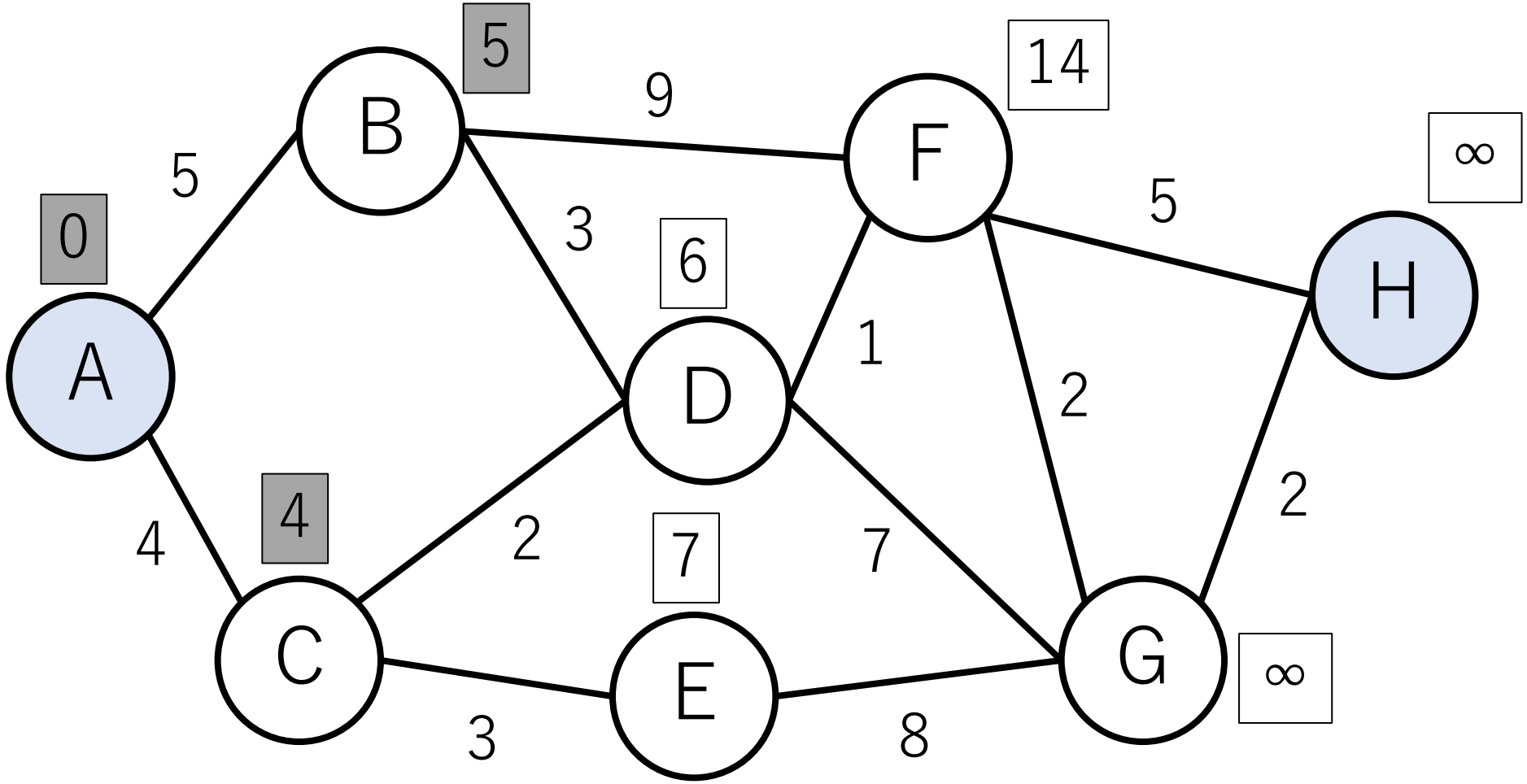
ダイクストラ法の例

dist[“B”] + edge[“B->D”]は8でdist[“D”]より大きい．　-> 無視



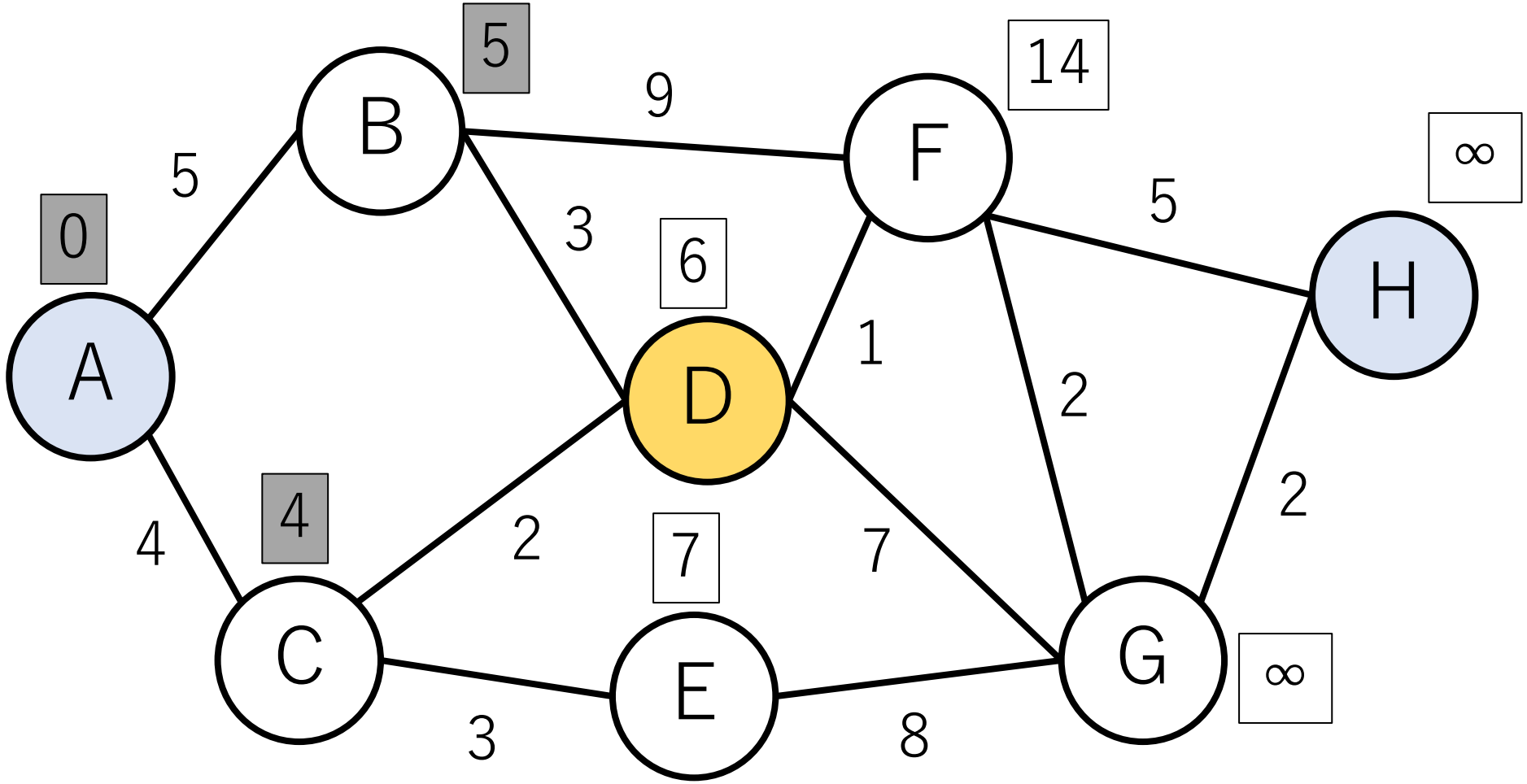
ダイクストラ法の例

C終わり.



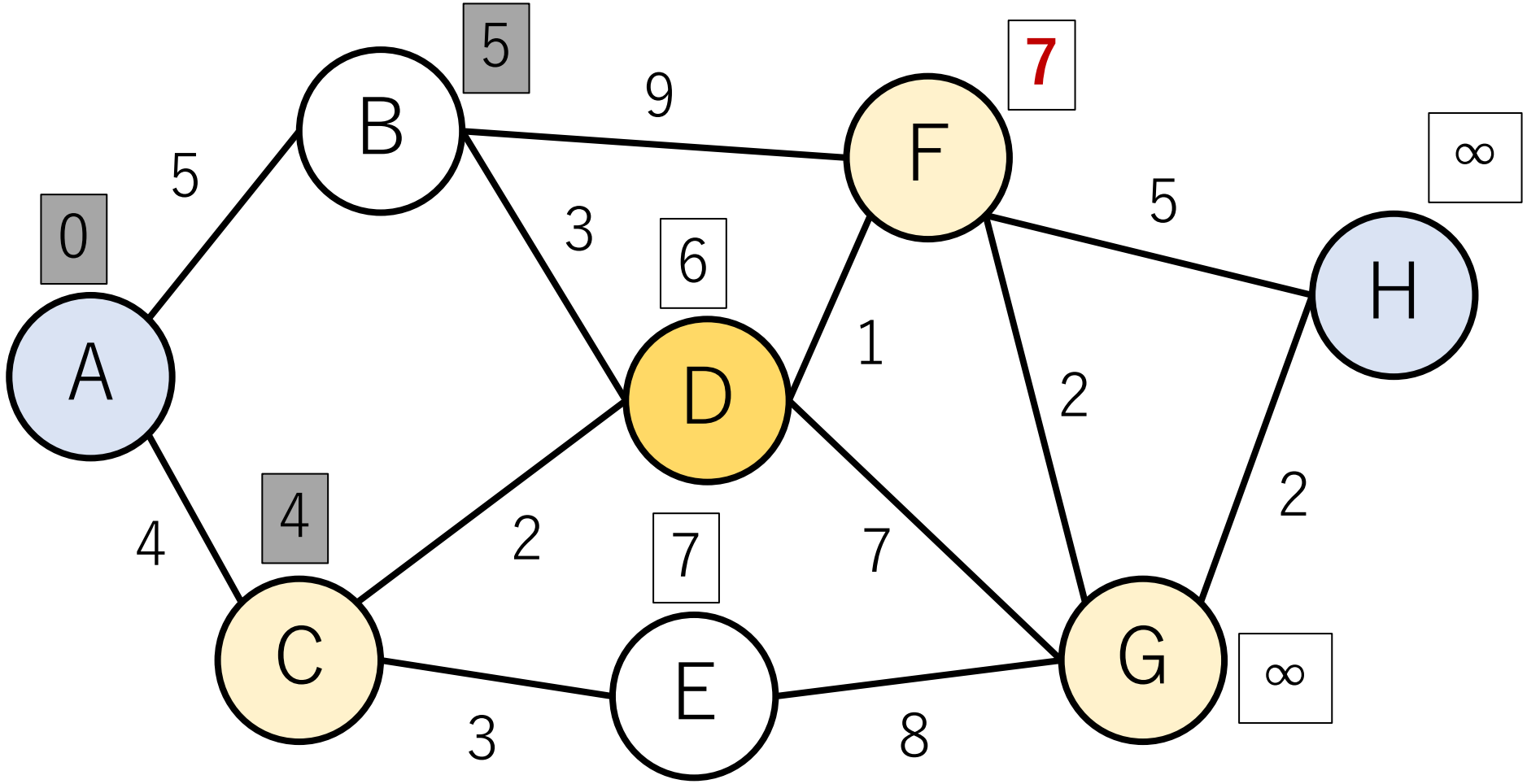
ダイクストラ法の例

最短距離になっているのはD.



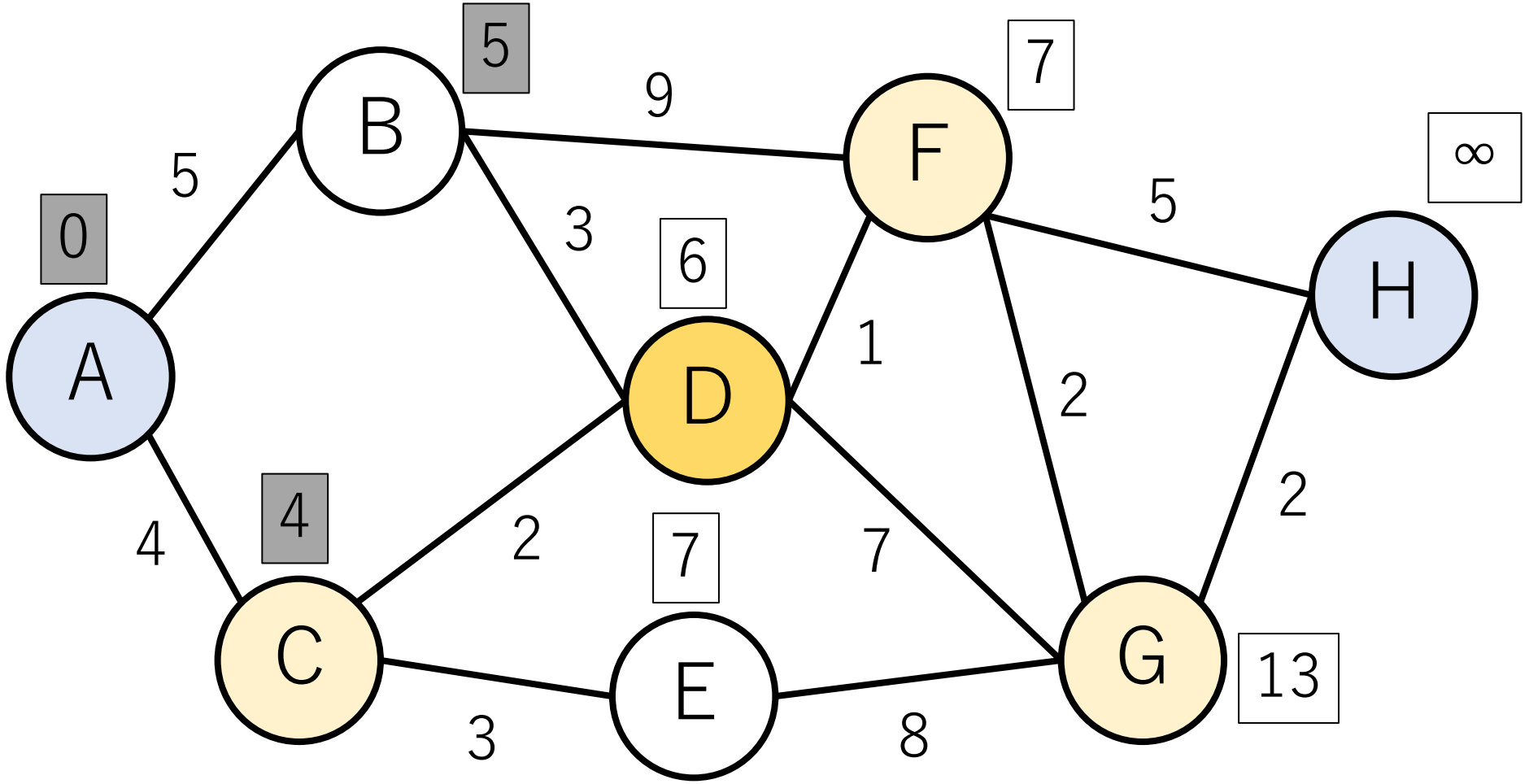
ダイクストラ法の例

dist[“D”] + edge[“D->F”]は7でdist[“F”]より小さい．　-> 更新



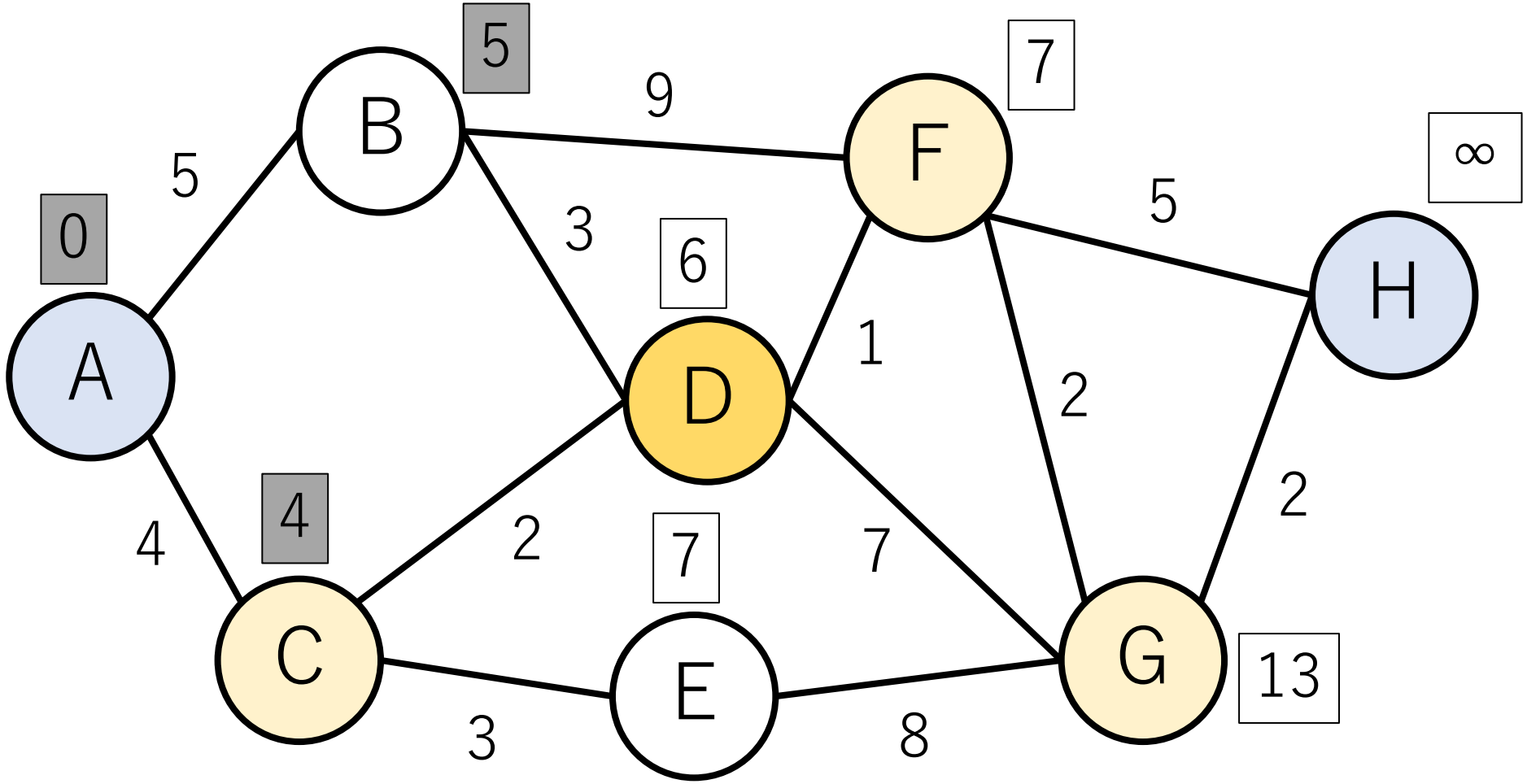
ダイクストラ法の例

dist[“G”] <- dist[“D”] + edge[“D->G”]



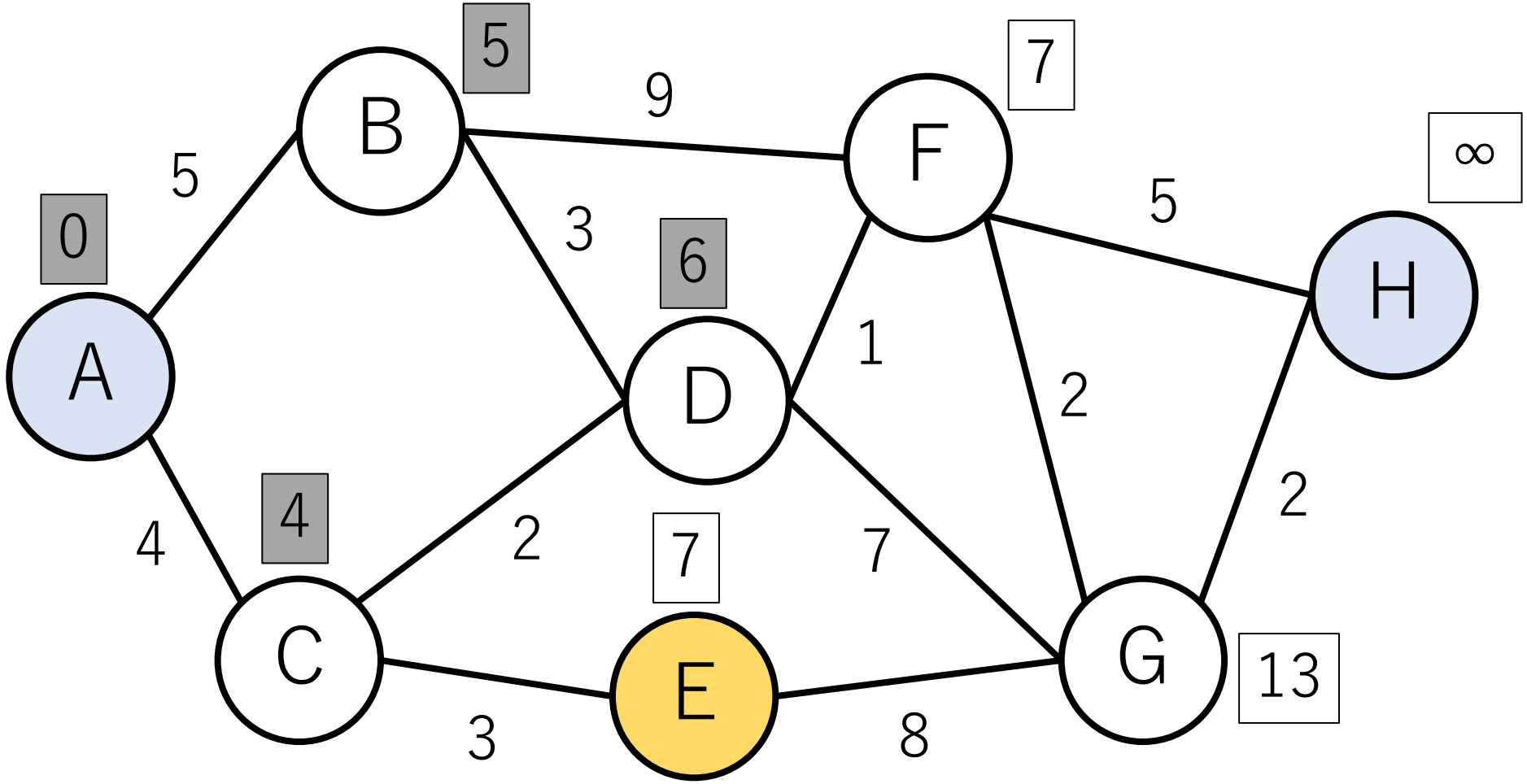
ダイクストラ法の例

Cはもう終わった（灰色になっている）ので，スルー．



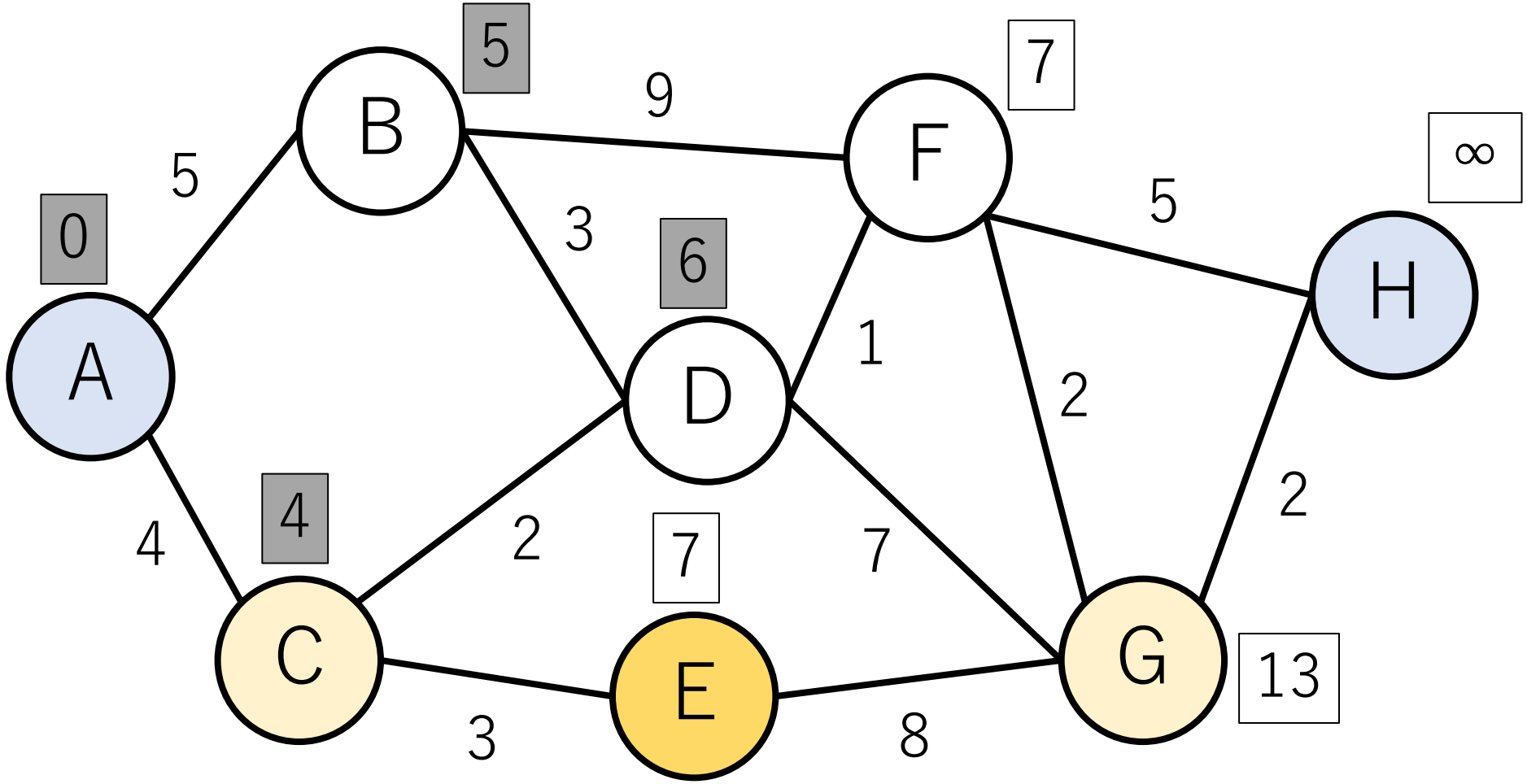
ダイクストラ法の例

D終了．Eに移る．



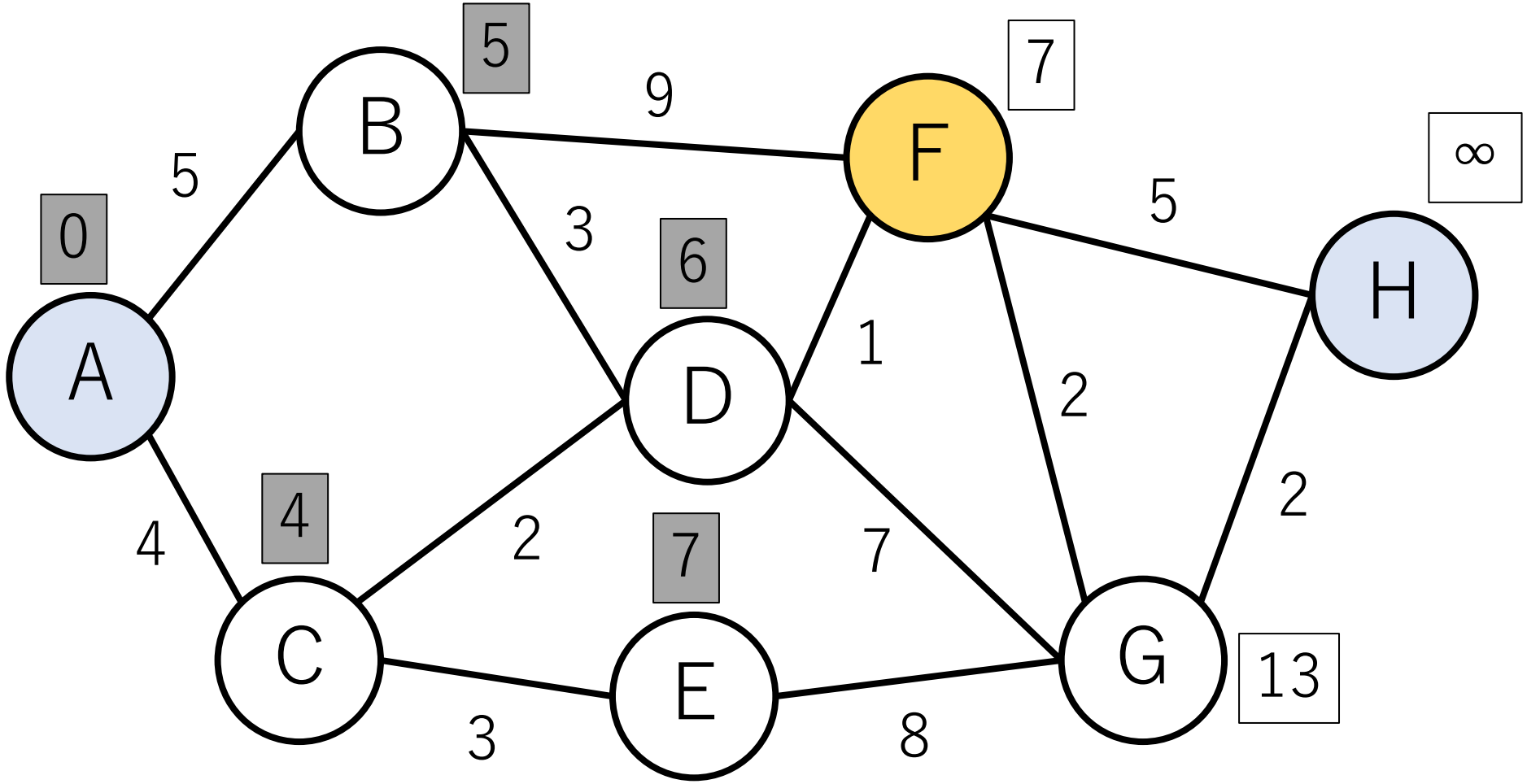
ダイクストラ法の例

EからはCとGがつながっているが，どちらも更新不要.



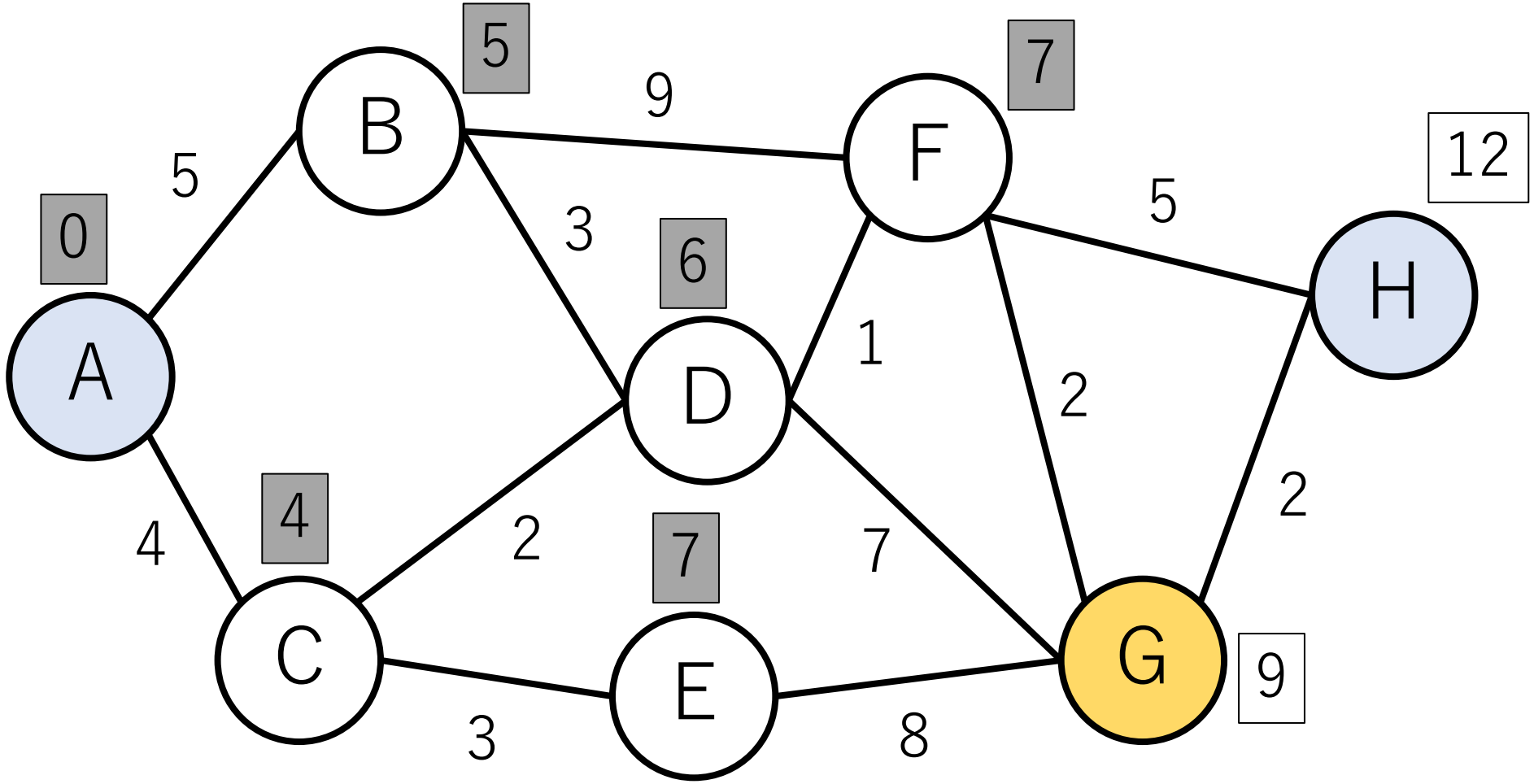
ダイクストラ法の例

更新なしでE終了．Fに移る．



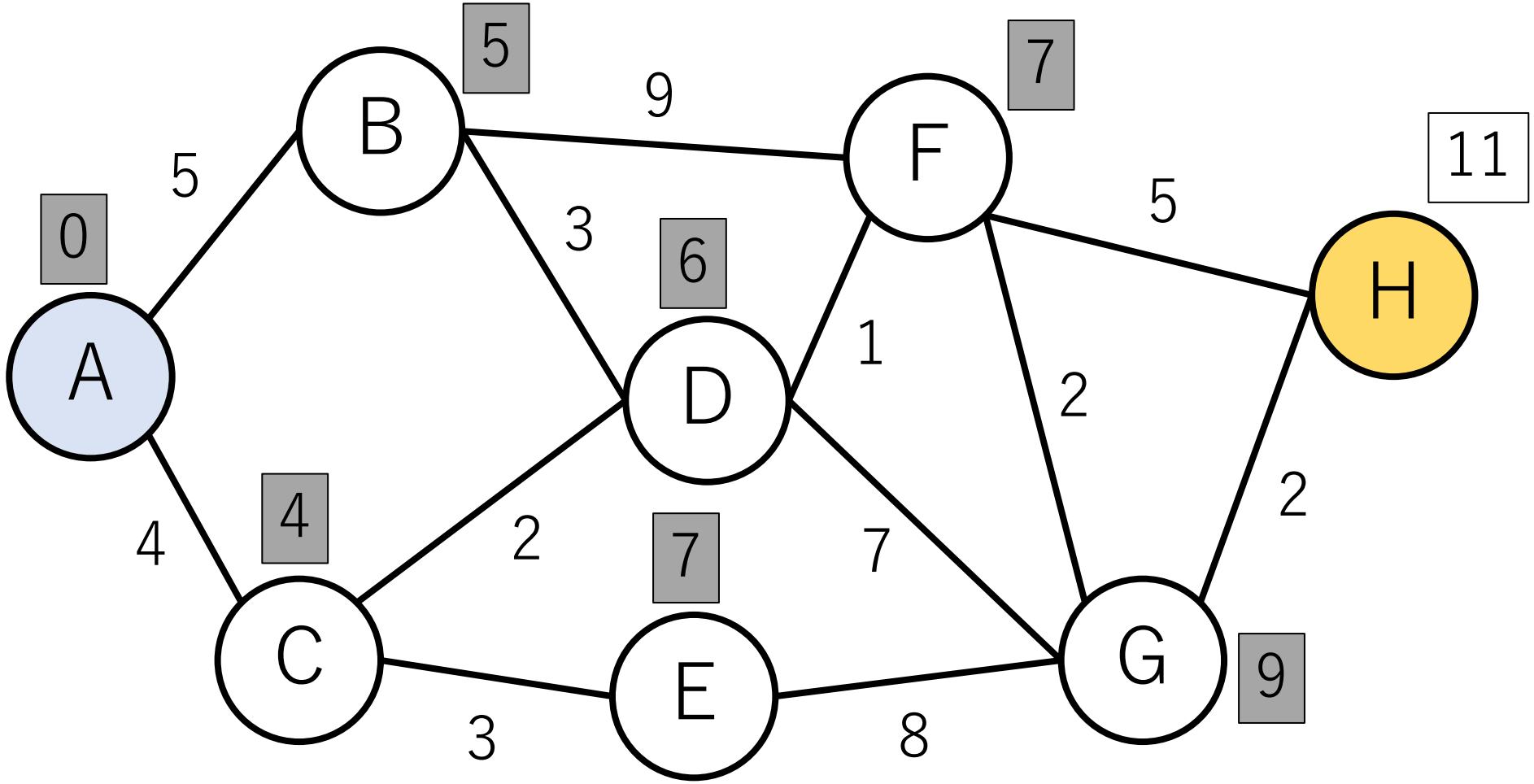
ダイクストラ法の例

dist[“G”]，dist[“H”]を更新して，F終了．Gに移る．



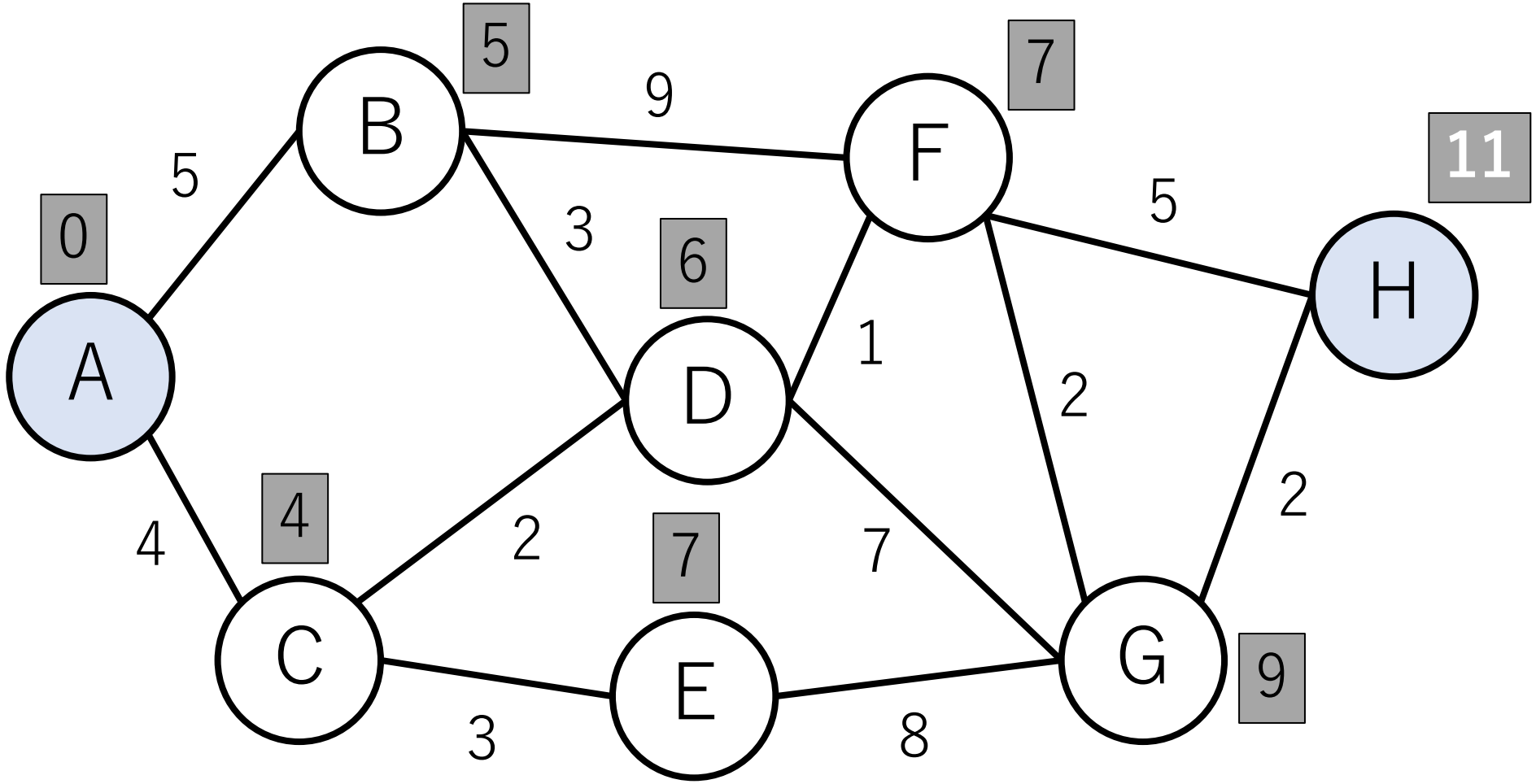
ダイクストラ法の例

Gから繋がるノードを更新して，終了．Hに移る．



ダイクストラ法の例

Hはゴールなので，ここでストップ．11が答え．



ダイクストラ（Dijkstra）法

#1 各ノードの最短距離を表す変数を（十分に大きい数字で）初期化する.

#2 開始ノードからスタート．ここは距離0.

#3 直接繋がっているノードに対して，接続する辺の距離を参照し，そのノードの現時点での最短距離を記録.

#4 開始ノードは処理が終わったので確定とする.

ダイクストラ（Dijkstra）法

#5 次に，最短距離が初期化されたときは違う値になっていて，かつ最短距離が確定されていないノードのうち，現時点で最短距離が最も小さいものを選び出す．

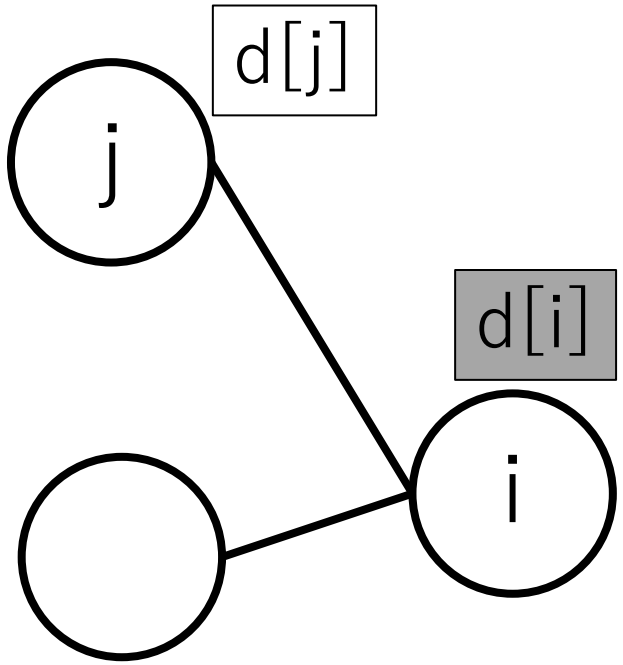
#6 直接繋がっているノードに対して，接続する辺の距離を参照し，その辺を使うことでそのノードの現時点での最短距離を更新できる場合は，更新する．

#7 このノードを確定とする．以降，全てのノードが確定するまで#5，#6を繰り返す．

ダイクストラ（Dijkstra）法

最短のノードから順に確定させていくことで,
後戻りしなくても済むようにしている.

例えば右の図のようにノードiの最短距離を確定させた時を考える．この時ノードiに繋がるノードjはまだ確定していないとする.

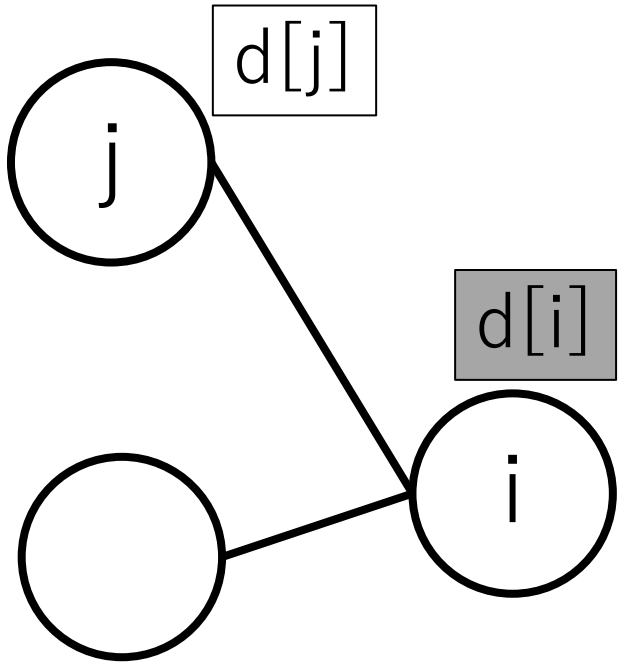


ダイクストラ（Dijkstra）法

ノードiは今までで未確定のノードのうち，d（上の説明ではdist）が最も小さいために選ばれ，確定したノードである．

つまり，$d[i] \leq d[j]$であり，かつ，ノードjからノードiの辺は非負なので，今確定した$d[i]$より大きくなることはない．

このため確定したノードはもう振り返る必要がない．



ダイクストラ法の実装

edge, distの他に，各ノードにおいて最短距離が確定されたかどうかを記録する．

done[]

False: このノードまでの最短距離が未確定．（白色）

True: このノードまでの最短距離が確定．（灰色）

ダイクストラ法の実装（隣接リスト）

```
#一番上が開始ノード
edges_list = [[[1, 5], [2, 4]],      #ノードA
[[0, 5], [3, 3], [5, 9]],            #ノードB
[[0, 4], [3, 2], [4, 3]],            #ノードC
[[1, 3], [2, 2], [5, 1], [6, 7]],    #ノードD
[[2, 3], [6, 8]],                    #ノードE
[[1, 9], [3, 1], [6, 2], [7, 5]],    #ノードF
[[3, 7], [4, 8], [5, 2], [7, 2]],    #ノードG
[[5, 5], [6, 2]]]                    #ノードH
```

ダイクストラ法の実装（初期化）

```
# Vはノードの数，e_listは隣接リスト
def dijkstra(V, e_list):
    inf = 10**9
    done = [False]*V
    dist = [inf]*V          # とても大きな値で初期化
    dist[0] = 0             # ノード0が開始ノード
```

ダイクストラ法の実装（探索部分）

```
def dijkstra(V, e_list):
        …
        while 1:
                # 現在までで最短距離を持つ未確定のノードを取り出す.
                tmp_min_dist = inf
                cur_node = -1
                for i in range(V):
                        if (not done[i]) and (tmp_min_dist > dist[i]):
                                tmp_min_dist = dist[i]
                                cur_node = i

                if cur_node == -1: break# 全部終わったらループ脱出.
```

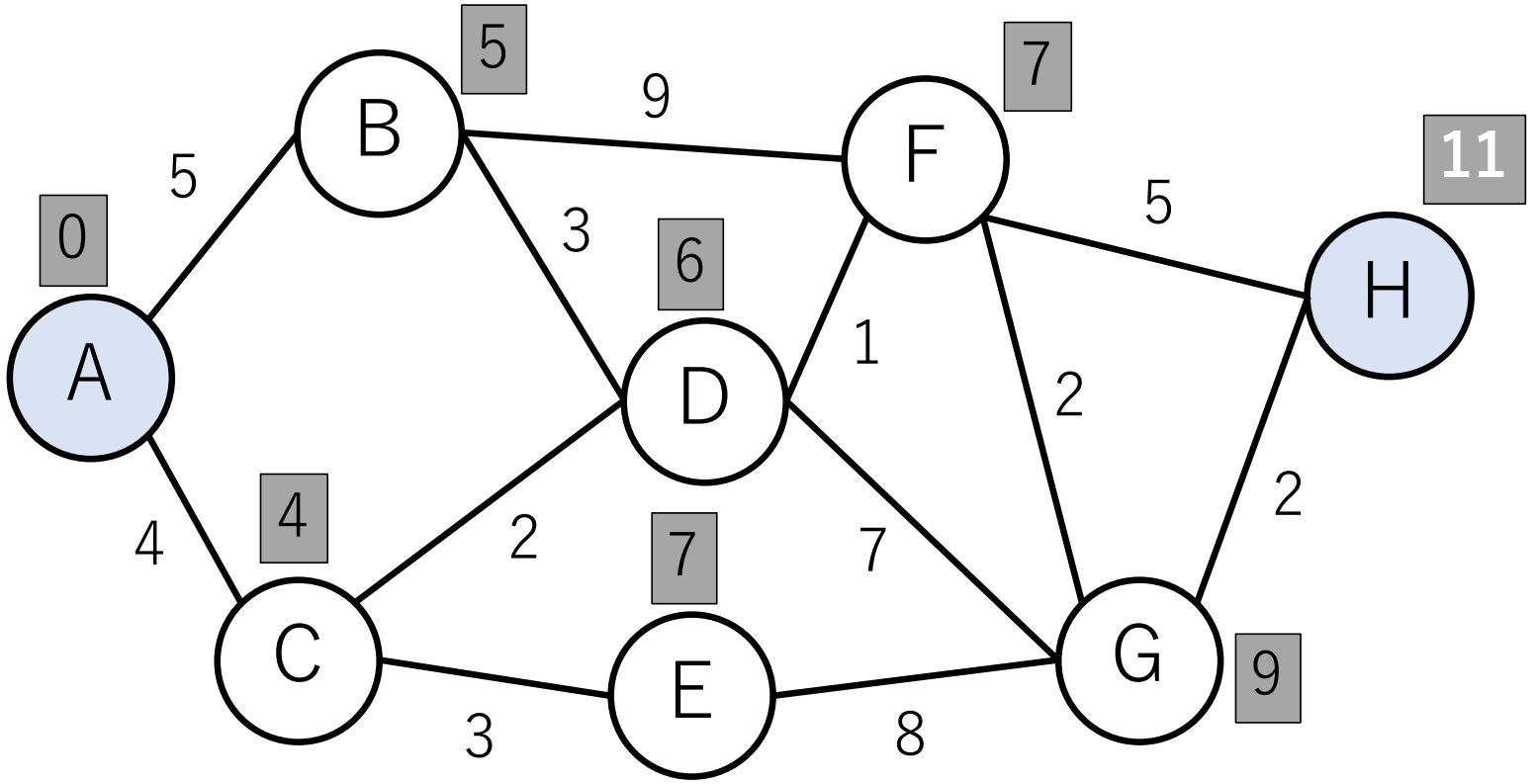
ダイクストラ法の実装（更新部分）

```
def dijkstra(V, e_list):
        …
            for e in e_list[cur_node]:
                # ノードcur_nodeから接続している辺を使う
                # ほうが距離を短く出来る場合は更新.
                if dist[e[0]] > dist[cur_node] + e[1]:
                    dist[e[0]] = dist[cur_node] + e[1]

            done[cur_node] = True # このノードは終わり.

        print(dist)
```

ダイクストラ法の実行例

dijkstra(8, edges_list)

-----実行結果-----
[0, 5, 4, 6, 7, 7, 9, 11]



ダイクストラ法の計算量

whileループは全てのノードを見るまで回り続けるので，$O(|V|)$.

forループは現在までで最短距離を持つ未確定のノードを取り出す部分と，繋がっているノードを更新する部分で2つ存在.

ダイクストラ法の計算量

最初のforループは常に$O(|V|)$で，whileループの分と合わせると，$O(|V|^2)$.

2つ目のforループでは選んだノードに接続した辺を全部見る．whileループと合わせると，全部の辺を見ることになるので，$O(|E|)$.

よって，上記の実装例では$O(|V|^2 + |E|)$.

ダイクストラ法の改良

「現在までで最短距離を持つ未確定のノードを取り出す」というところで毎回探索が必要となり，ここがボトルネックになっている．．．

あらかじめ最短距離を持つ未確定のノードがすぐわかるようなデータ構造で記録できない？

→ヒープを使おう！

ヒープを使うダイクストラ法

#1 ヒープから，現在までで最短距離を持つ未確定のノードを取り出す（ヒープが更新される）．

#2 distを更新する．distの更新があればヒープの更新（要素の追加）を行う．

#3 全ノード終わるまで，#1に戻る．

ヒープを使うダイクストラ法の実装例

```
import heapq

def dijkstra_heap(V, e_list):
        inf = 10**9
        done = [False]*V
        dist = [inf]*V
        dist[0] = 0
        node_heap = []        # ヒープ
```

ヒープを使うダイクストラ法の実装例

```
def dijkstra_heap(V, e_list):
    ...

    # ヒープには[[現在までの最短距離], [ノード]]で格納.
    # これで現在までの最短距離で優先度が付けられる.
    heapq.heappush(node_heap, [dist[0], 0])
```

ヒープを使うダイクストラ法の実装例

```
def dijkstra_heap(V, e_list):
    ...
    # ヒープに要素がある限りループを回す.
    while node_heap:
        # 未確定で最短距離のノードを取り出す.
        tmp = heapq.heappop(node_heap)
        cur_node = tmp[1]
```

ヒープを使うダイクストラ法の実装例

```
def dijkstra_heap(V, e_list):
        …
                if not done[cur_node]:  # 未訪問ならば処理する
                        for e in e_list[cur_node]:
                                if dist[e[0]] > dist[cur_node] + e[1]:
                                        dist[e[0]] = dist[cur_node] + e[1]
                                        # 更新した場合ヒープに入れる.
                                        # iが重複してもより距離の短いほうの
                                        # 情報が先に使われるので問題ない.
                                        heapq.heappush(node_heap,
                                                       [dist[e[0]], e[0]])
```

ヒープを使うダイクストラ法の実装例

```
def dijkstra_heap(V, e_list):
        …
        while node_heap:
                …
                # すべて終わったら，このノードを処理済にする
                done[cur_node] = True

        print(dist)
```

ヒープを使うダイクストラ法の計算量

上記の実装では，ヒープに入る要素の数は$O(|E|)$となる．
　　重複して入るノードが存在するため．

distの更新に伴う要素の追加
　　→辺の数$O(|E|)$分発生するので，$O(|E|\log|E|)$.
最短距離のノードを取り出す
　　→ヒープの要素の数だけ発生するので，$O(|E|\log|E|)$.

よって，全体でも$O(|E|\log|E|)$.

ヒープを使うダイクストラ法の計算量

もし，重複するノードをヒープに追加せず，直接更新出来る場合，ヒープの大きさはノードの数$O(|V|)$になり，更新にかかる計算量は$O(\log|V|)$となる.

distの更新に伴う要素の更新

→辺の数$O(|E|)$分発生するので，$O(|E|\log|V|)$.

最短距離のノードを取り出す

→ノードの数$O(|V|)$分発生するので，$O(|V|\log|V|)$.

ヒープを使うダイクストラ法の計算量

よって，全体としては$O((|V| + |E|) \log |V|)$.

連結グラフ（任意の2ノード間にパスが存在するグラフ）では$O(|V|) \leq O(|E|)$なので，$O(|E| \log |V|)$として説明されることもある.

ただし，ある場合には$O(|V|^2)$より悪くなりえる.

それはどんな場合？その場合の計算量は？

ヒープを使うダイクストラ法の計算量

なお，$O(\log|E|)$は$O(\log|V|)$と等価であるともいえる．

完全グラフ（全ノードがお互いに接続されているグラフ）を考えても$|E|$は高々$|V|^2$なので，$O(\log|E|) = O(\log|V|^2) \rightarrow O(\log|V|)$となるため．

さらに

フィボナッチヒープという特殊なヒープを使うと，
$O(|E| + |V| \log |V|)$に出来ることが知られている．

フィボナッチヒープでは要素の追加が$O(1)$，最小値の取り出し（＆削除）が$O(\log |V|)$とみなせる．

この場合，先程の最悪なケースでも$O(|V|^2 + |V| \log |V|)$となり，少しはまし．

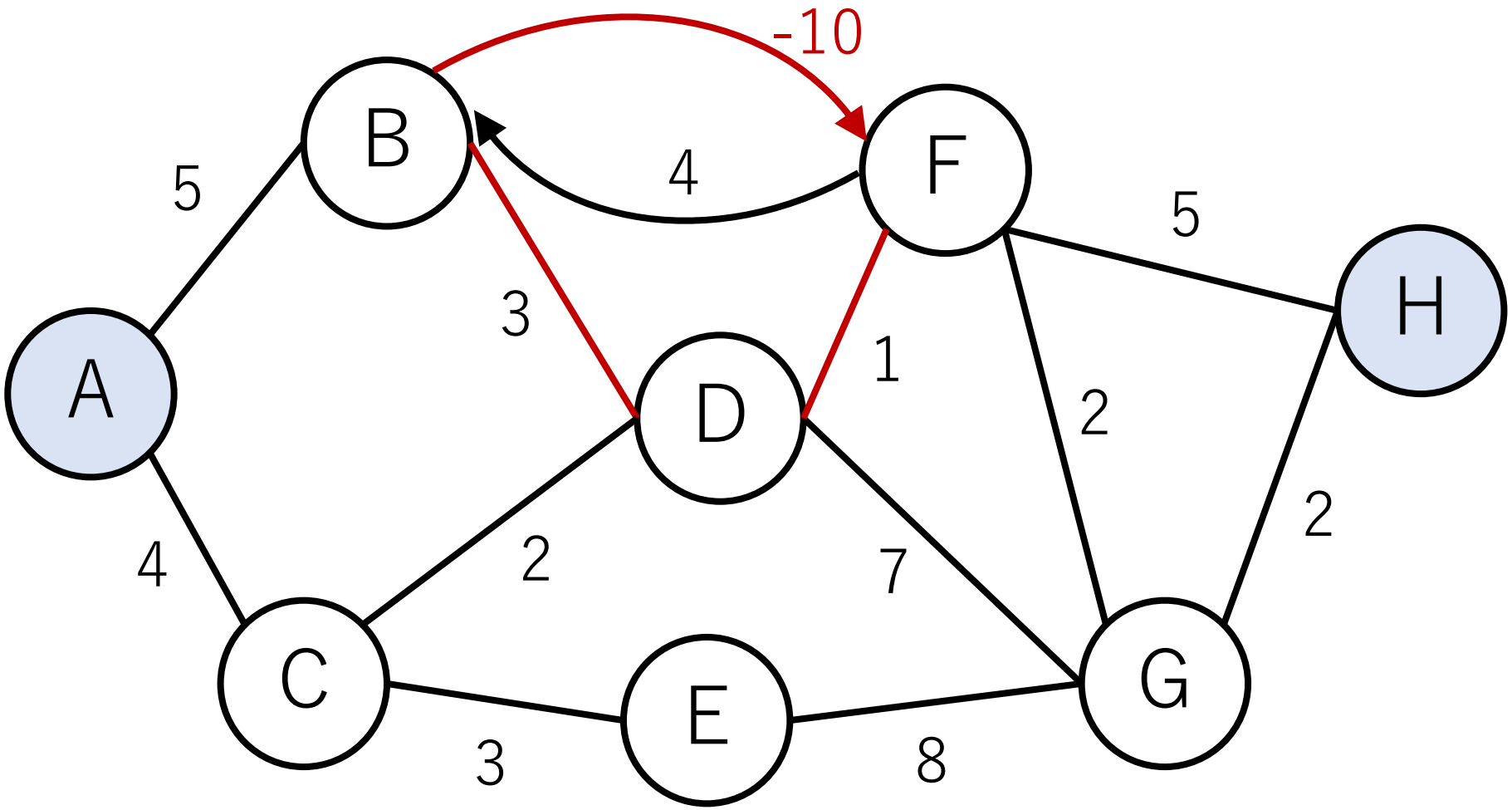
ダイクストラ法を使う前提

距離はすべて非負.

距離の値はバラバラ. （距離がすべて一定ならBFSでよい）

最短距離が計算できない時

負の距離が存在し，負になる閉路が存在する．



ベルマン・フォード（Bellman-Ford）法

構造はダイクストラと同じ.

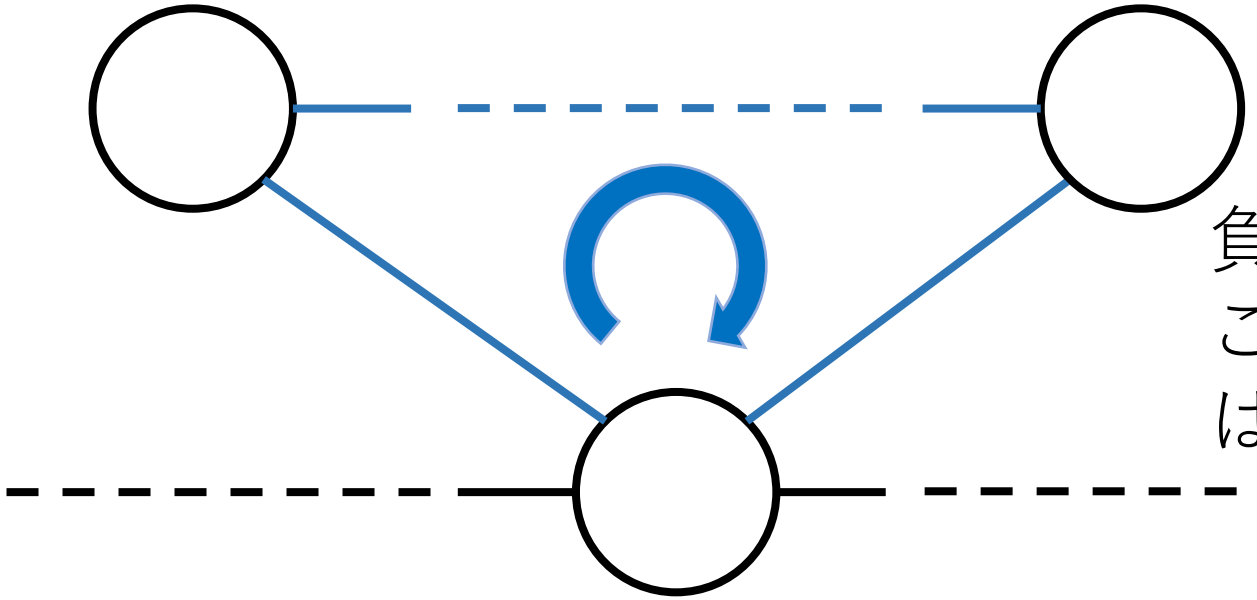
ダイクストラと違い，最短距離の選択を行わず，更新毎に全ての辺に対しての計算を毎回行う.

負の経路があっても計算可能.

負の閉路があってもそれを検知可能.

負の閉路の検知

負の閉路がなければ，最短路が同じノードを通ることはない．二度以上通れば，それは通らない場合と比較して距離が大きくなるはず．



負の閉路がなければ，この青色経路の総距離は必ず正．

負の閉路の検知

負の閉路がなければ，最短路が同じノードを通ることはない．二度以上通れば，それは通らない場合と比較して距離が大きくなるはず．

つまり，負の閉路がない場合，最短距離になり得るパスの最大の長さ（距離・コストの総和ではない）は $|V| - 1$.

ダイクストラ法でいうところのwhileループの実行回数．（ノードの総数 – 開始ノード）

負の閉路の検知

もし，このループの$|V|$回以上実行した時，あるノードの最短距離の更新があったとすると，それは負の閉路があることになるサインとなる.

よって，$|V|$回目のループを実行したときに更新があるかどうかをチェックすれば良い！

ベルマン・フォード法の実装例

```
# リスト表現だが先ほどと形式が違うことに注意.
# 始点，終点，距離の順.
edges_list2 = [[0, 1, 5], [0, 2, 4], [1, 0, 5], [1, 3, 9], [1, 5, 9],
[2, 0, 4], [2, 3, 2], [2, 4, 3], [3, 1, 9], [3, 2, 2], [3, 5, 1],
[3, 6, 7], [4, 2, 3], [4, 6, 8], [5, 1, 9], [5, 3, 1], [5, 6, 2],
[5, 7, 5], [6, 3, 7], [6, 4, 8], [6, 5, 2], [6, 7, 2], [7, 5, 5],
[7, 6, 2]]
```

ベルマン・フォード法の実装例

```
def BellmanFord(V, e_list):
    inf = 10**9
    dist = [inf]*V
    dist[0] = 0
```

ベルマン・フォード法の実装例

```
def BellmanFord(V, e_list):
    ...

    # whileではなく，|V|回のforループにする．
    # もしj=V-1で更新があれば，それは負の閉路の
    # 存在を表す．
    for j in range(V):
```

ベルマン・フォード法の実装例

```
def BellmanFord(V, e_list):
    ...
    for j in range(V):
        for e in edges_list:
            if dist[e[1]] > e[2] + dist[e[0]]:
                dist[e[1]] = e[2] + dist[e[0]]
                # 負の閉路の検知
                if j==V-1: return -1
    print(dist)
```

ベルマン・フォード法の実行例

edges_list2 = [[0, 1, 5], [0, 2, 4], [1, 0, 5], [1, 3, 9], [1, 5, 9], [2, 0, 4], [2, 3, 2], [2, 4, 3], [3, 1, 9], [3, 2, 2], [3, 5, 1], [3, 6, 7], [4, 2, 3], [4, 6, 8], [5, 1, 9], [5, 3, 1], [5, 6, 2], [5, 7, 5], [6, 3, 7], [6, 4, 8], [6, 5, 2], [6, 7, 2], [7, 5, 5], [7, 6, 2]]

BellmanFord(8, edges_list2)

-----実行結果-----

[0, 5, 4, 6, 7, 7, 9, 11]

ベルマン・フォード法の実行例

edges_list2 = [[0, 1, 5], [0, 2, 4], [1, 0, 5], [1, 3, 9], [1, 5, -10], [2, 0, 4], [2, 3, 2], [2, 4, 3], [3, 1, 9], [3, 2, 2], [3, 5, 1], [3, 6, 7], [4, 2, 3], [4, 6, 8], [5, 1, 9], [5, 3, 1], [5, 6, 2], [5, 7, 5], [6, 3, 7], [6, 4, 8], [6, 5, 2], [6, 7, 2], [7, 5, 5], [7, 6, 2]]

BellmanFord(8, edges_list2)

-----実行結果-----

-1

ベルマン・フォード法の計算量

2重ループの1つ目は，$O(|V|)$.

2つ目は毎回すべての辺をチェックするので，$O(|E|)$.

よって，全体では$O(|V||E|)$.

もし，隣接行列で実装すると2つ目のループは$O(|V|^2)$
になってしまうので，全体では$O(|V|^3)$.

Shortest Path Faster Algorithm (SPFA)

基本の考えはベルマン・フォードに同じだが，毎回全部の辺をチェックすることを避けることで高速化を図る．

Shortest Path Faster Algorithm (SPFA)

基本の考えはベルマン・フォードに同じだが，毎回全部の辺をチェックすることを避けることで高速化を図る.

ノードiのdist[i]に更新がなければ，そこから直接つながっているノードに対するdistの更新は必要ない.

つまり，dist[i]に更新が起きた時のみ，そこに接続するノードも更新する必要がある，として順次処理をする.

Shortest Path Faster Algorithm (SPFA)

実装においては，更新が必要なノードが出てきたら，
それをキューに入れる．

そのキューが空になるまでループを回す．

SPFAの実装例

```
edges_list = [          # ダイクストラのときと同じ形式
[[1, 5], [2, 4]],
[[0, 5], [3, 3], [5, 9]],
[[0, 4], [3, 2], [4, 3]],
[[1, 3], [2, 2], [5, 1], [6, 7]],
[[2, 3], [6, 8]],
[[1, 9], [3, 1], [6, 2], [7, 5]],
[[3, 7], [4, 8], [5, 2], [7, 2]],
[[5, 5], [6, 2]]]
```

SPFAの実装例（負の経路がないと仮定）

```
from collections import deque

# 引数：ノード数，隣接リスト
def spfa(V, e_list):
        inf = 10**9
        dist = [inf]*V
        dist[0] = 0
```

SPFAの実装例（負の経路がないと仮定）

```
def spfa(V, e_list):
    ...
    # チェックが必要なノードを格納するキュー
    node_to_check = deque()
     # キューに入っているかどうかのフラグ
    in_queue = [False]*V
```

SPFAの実装例（負の経路がないと仮定）

```
def spfa(V, e_list):
    ...
    # 開始ノード（index：0）をキューに入れる
    node_to_check.append(0)
    in_queue[0] = True

    # 辺をチェックした数をカウント（本来は必要なし）
    count = 0
```

SPFAの実装例（負の経路がないと仮定）

```
def spfa(V, e_list):
    ...
    # キューにノードがある限りループを回す.
    while node_to_check:

        # キューから取り出し，このノードをチェック.
        cur_node = node_to_check.popleft()
        in_queue[cur_node] = False
```

SPFAの実装例（負の経路がないと仮定）

```
def spfa(V, e_list):
    ...
    while node_to_check:
    ...
        # cur_nodeからつながっている辺をチェック.
        for e in e_list[cur_node]:
            count += 1
            if dist[e[0]] > dist[cur_node] + e[1]:
                dist[e[0]] = dist[cur_node] + e[1]
```

SPFAの実装例（負の経路がないと仮定）

```
def spfa(V, e_list):
        ...
                if dist[e[0]] > dist[cur_node] + e[1]:
                    ...
                    # 更新したらキューに入れる
                    if not in_queue[e[0]]:
                        in_queue[e[0]] = True
                        node_to_check.append(e[0])
```

SPFAの実装例（負の経路がないと仮定）

```
def spfa(V, e_list):
    …

    # 開始ノードから各ノードまでの最短距離と
    # 辺をチェックした回数を出力.
    print(dist)
    print(count)
```

SPFAの実行例

spfa(8, edges_list)

-----実行結果-----
[0, 5, 4, 6, 7, 7, 9, 11]
24

ベルマン・フォードを使った場合，辺のチェック回数は，192回になります．

SPFAの計算量

最悪のケースでは毎回全ての辺を調べることになり，
ベルマン・フォードと等価になるので，$O(|V||E|)$.

負の辺がないランダムなケースでは，実験的には$O(|E|)$
くらいになることが知られている.

厳密な証明はまだされていないらしい.

負の閉路を検知するのは，ベルマン・フォードと同様の
考え方に立ち，あるノードがキューに$|V|$回以上入って
しまうことをチェックすれば良い.

SPFAの計算量

キューの代わりにスタックでも実装可能．ただし，辺の比較回数はかなり増える．

キューを使うとBFS的にノードを辿ることになる．負辺がない場合，最小ステップ数で行ける経路が最短の経路になる事が多いと期待できるので，できうる限り少ないステップ数でノードに到達するようにチェックしていくことが有利に働いていると考えられる．

（あくまで雑な考え方として理解してください．．．）

パフォーマンス比較例 [msec]

負辺なし無向グラフ，ランダムケースで比較.

ノード数	**1,000**	**1,000**	**2,000**	**2,000**
辺の数	**3,000**	**6,000**	**3,000**	**6,000**
ダイクストラ（単純探索） $O(\|V\|^2 + \|E\|)$	**52**	**53**	**200**	**198**
ダイクストラ（ヒープ） $O(\|E\| \log \|E\|$ の実装)	**2.1**	**3.4**	**2.8**	**4.2**
ベルマン・フォード $O(\|V\|\|E\|)$	**976** (6,000,000)	**1,779** (12,000,000)	**1,712** (12,000,000)	**3,587** (24,000,000)
SPFA $O(\|V\|\|E\|)$（**実験的には** $O(\|E\|)$）	**2.7** (11,709)	**4.8** (28,469)	**3.0** (9,336)	**5.6** (25,898)

ベルマン・フォードとSPFAのカッコ内の数字は，辺の総チェック回数を表す.

最短経路問題の種類

2頂点対最短経路問題

特定の2つのノード間の最短経路を求める.

単一始点最短経路問題

ある始点ノードから他の全部のノードへの最短経路を求める.

全点対最短経路問題

すべての2ノード間の最短経路を求める.

ワーシャル・フロイド（Warshall-Floyd）法

2つのノードの全ての組み合わせに対して，最短経路（全点対最短経路）を導出するアルゴリズム．

2ノード間（i ->‥-> j）のパスにおいて，ノードkを経由（i ->‥-> k ->‥-> j）した方が距離が短くなるかどうかを，全てのノードに対してチェックする．

非常に簡潔にコードが書ける！

ワーシャル・フロイド法の考え方

$V_k = \{1,2,\cdots,k\}$とし，V_kのみを経由するノードiとjの間のパスの最短距離を$d_{i,j}^k$とする．

もし，すべてのi，jで$d_{i,j}^{k-1}$（V_{k-1}を経由するパス）がわかっているとした時，これを使って$d_{i,j}^k$が求めることを考えよう．

ワーシャル・フロイド法の考え方

ノードkを加えたV_kを経由するパスにおける$d_{i,j}^k$の候補は以下の2通り.

1) $d_{i,j}^k$がノードkを経由しないパスである場合
2) $d_{i,j}^k$がノードkを経由するパスである場合

ワーシャル・フロイド法の考え方

1) $d_{i,j}^{k}$がノードkを経由しないパスである場合

この場合は，V_{k-1}を経由するパスと同じになる．

つまり，$d_{i,j}^{k} = d_{i,j}^{k-1}$.

2) $d_{i,j}^{k}$がノードkを経由するパスである場合

この場合，どこかでノードkを通ることになる．

すなわち，i ->…-> k ->…-> jとなる．

この場合，$d_{i,j}^{k} = d_{i,k}^{k-1} + d_{k,j}^{k-1}$.

ワーシャル・フロイド法の考え方

以上より，すべてのi，jのペアに対して，

$$d_{i,j}^{k} = \min(d_{i,j}^{k-1}, d_{i,k}^{k-1} + d_{i,k}^{k-1})$$

として，順次計算できる．

さらに，$V_0 = \{\}$の場合，$d_{i,j}^{0}$はノードiとjを直接つながっている辺の距離になる（つながってない場合は，無限大）ので，明らか．

これによって，$d_{i,j}^{k}$を求めることができる！

ワーシャル・フロイド法の実装例

```
# 引数：ノードの総数，隣接行列
def WarshallFloyd(V, e_matrix):

    # dist[i][j]：ノードiからノードjまで最短距離を保持する．
    # 隣接行列を保持しておきたいならdeepcopyにする．
    dist = e_matrix
```

ワーシャル・フロイド法の実装例

```
def WarshallFloyd(V, e_matrix):
    …
    [全てのi，j，kの組み合わせで以下を行う]:
        [dist[i][k]とdist[k][j]の両方がinfでないならば]:
            [ノードi からノードj の距離に関して，経由
            ノードkを経由する場合としない場合を比較し，
            より短い方をdist[i][j]に入れる．]

    print(dist)
```

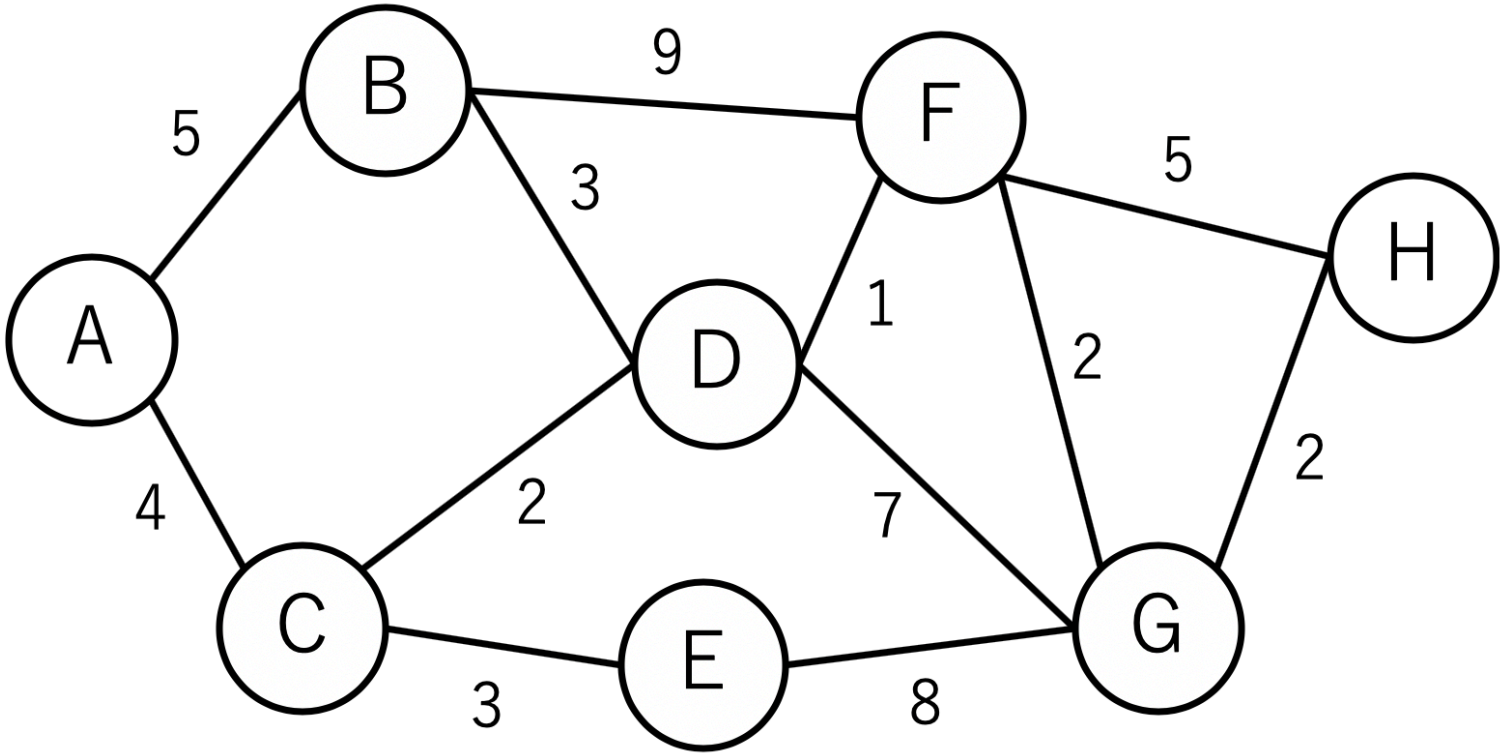
ただし，ここに注意！

[全てのi，j，kの組み合わせで以下を行う]

どの順番でループを回さないといけないか，上のスライドの説明をよく見て，考えてみてください．

ワーシャル・フロイド法の実行例

[[0, 5, 4, 6, 7, 7, 9, 11],
[5, 0, 5, 3, 8, 4, 6, 8],
[4, 5, 0, 2, 3, 3, 5, 7],
[6, 3, 2, 0, 5, 1, 3, 5],
[7, 8, 3, 5, 0, 6, 8, 10],
[7, 4, 3, 1, 6, 0, 2, 4],
[9, 6, 5, 3, 8, 2, 0, 2],
[11, 8, 7, 5, 10, 4, 2, 0]]



ワーシャル・フロイド（Warshall-Floyd）法

負の経路があっても使える.

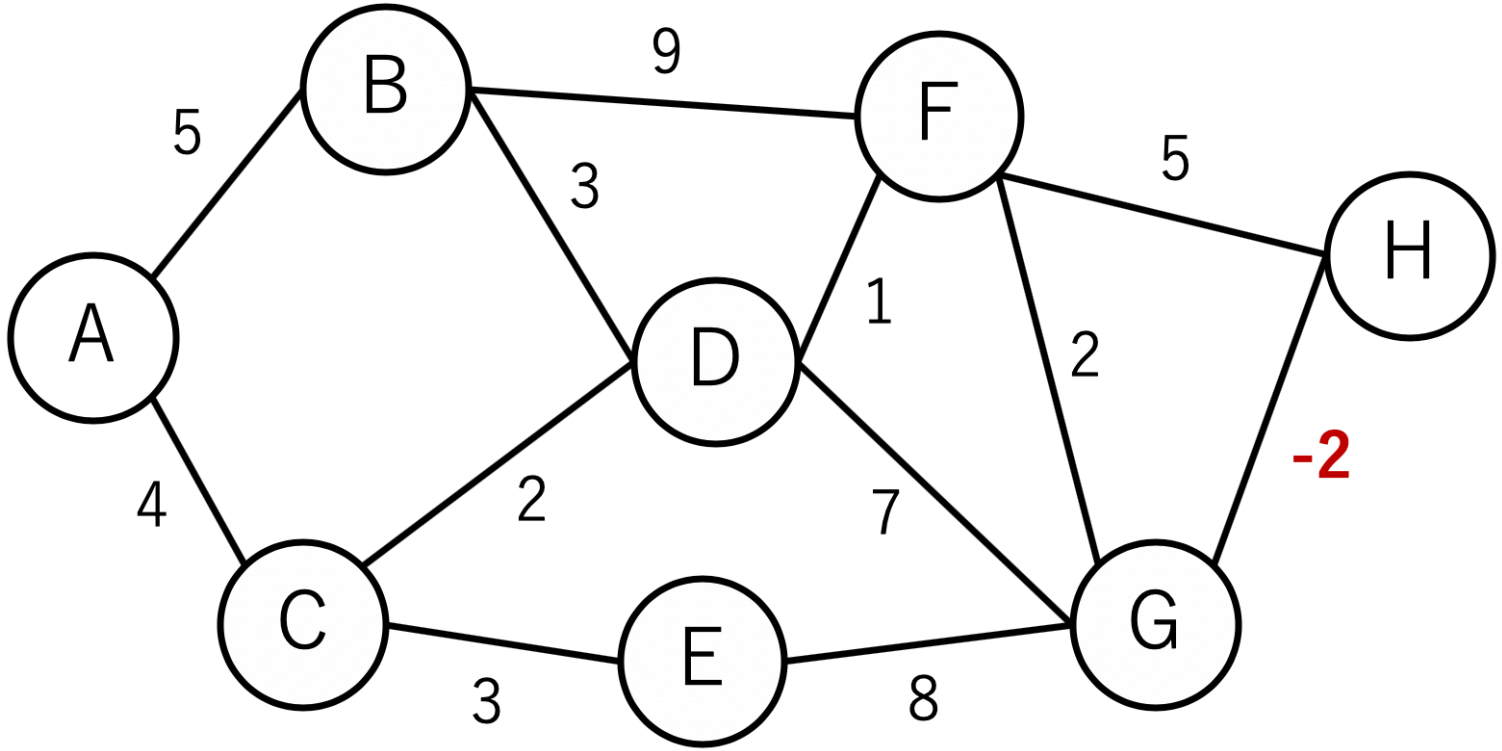
負の閉路が存在する場合，dist[i][i]が負の値になる.

本来なら0だが負の閉路があるためにどこかを回って来たほうが短くなってしまう.

dist[i][i]が負の値になった場合，得られた最短距離は正しくないので注意.（負の閉路を何度も回ればいくらでも小さくできるため）

ワーシャル・フロイド法の実行例

[[0, 5, 4, 6, 7, 7, 9, 7],
[5, 0, 5, 3, 8, 4, 6, 4],
[4, 5, 0, 2, 3, 3, 5, 3],
[6, 3, 2, 0, 5, 1, 3, 1],
[7, 8, 3, 5, 0, 6, 8, 6],
[7, 4, 3, 1, 6, 0, 2, 0],
[9, 6, 5, 3, 8, 2, -4, -6],
[7, 4, 3, 1, 6, 0, -6, -24]]



ワーシャル・フロイド法の計算量

3重ループが存在しており，$O(|V|^3)$.

ノードの数が増えるとけっこう大変．．．

まとめ

最短経路問題に対するアルゴリズム

単一始点最短経路問題

- ダイクストラ
- ベルマン・フォード
- SPFA

全点対最短経路問題

- ワーシャル・フロイド

ダイクストラ，ベルマン・フォード，SPFA

全部以下のような構造を持つ．

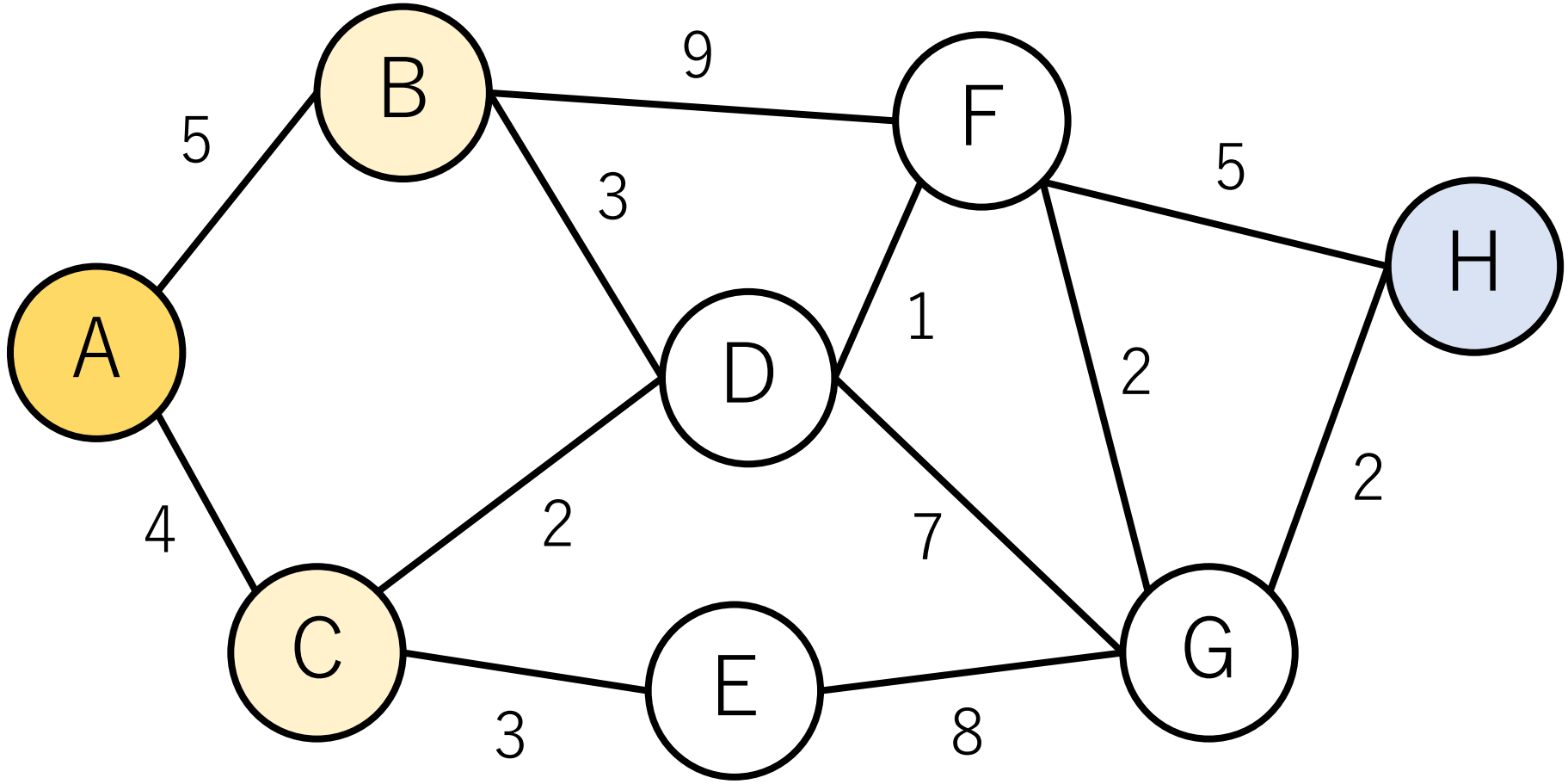
```
if [ノードjのdist] > [ノードiのdist] + [i-jの距離]:
    [ノードjのdist] <- [ノードiのdist] + [i-jの距離]
```

DPのときと同じ！（ちなみに，ベルマン・フォードの「ベルマン」は動的計画法を考案したRichard Bellmanさんです．）

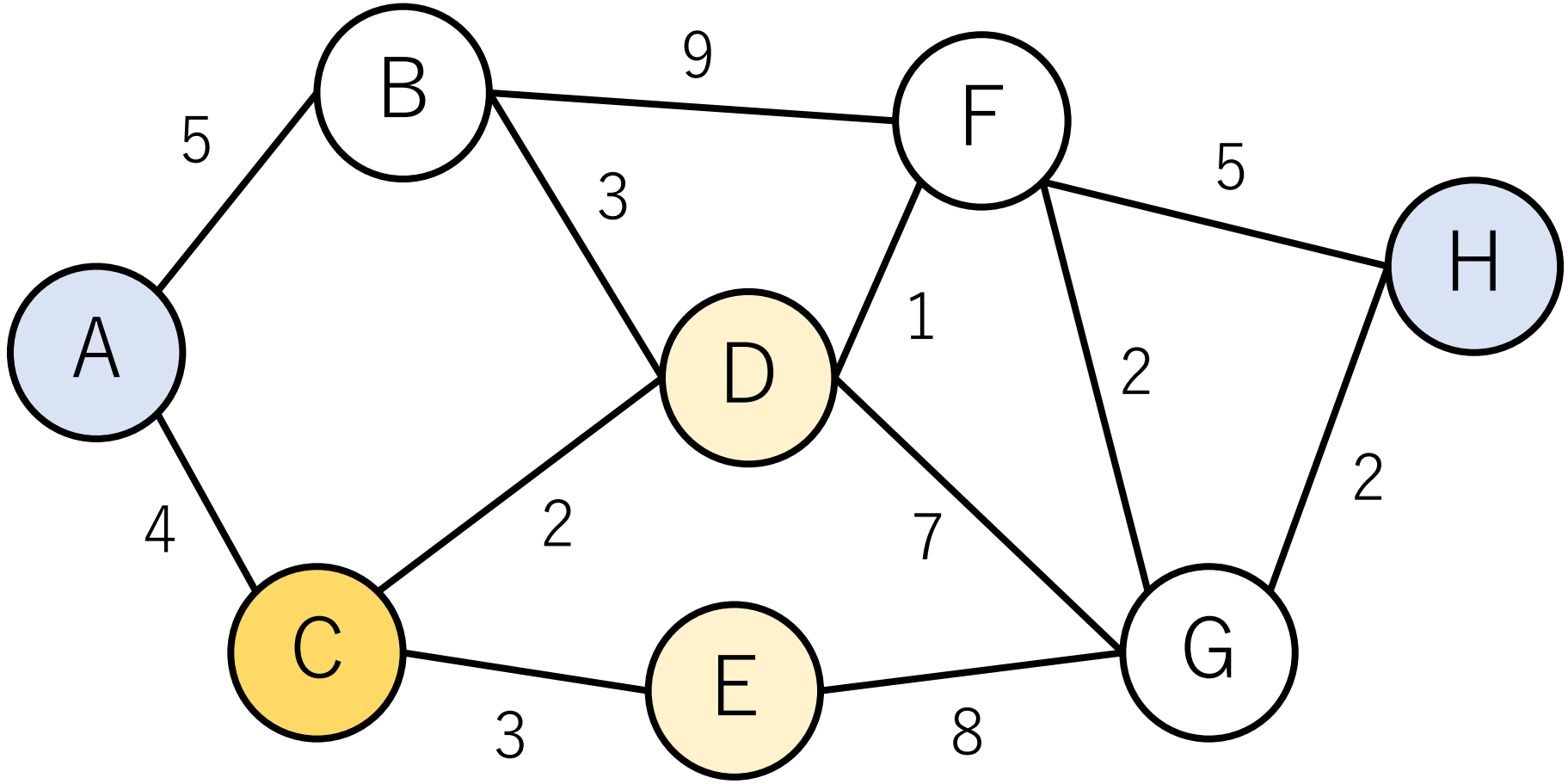
ダイクストラ，ベルマン・フォード，SPFA

上の3つは配るDPになっているとも考えることができる．

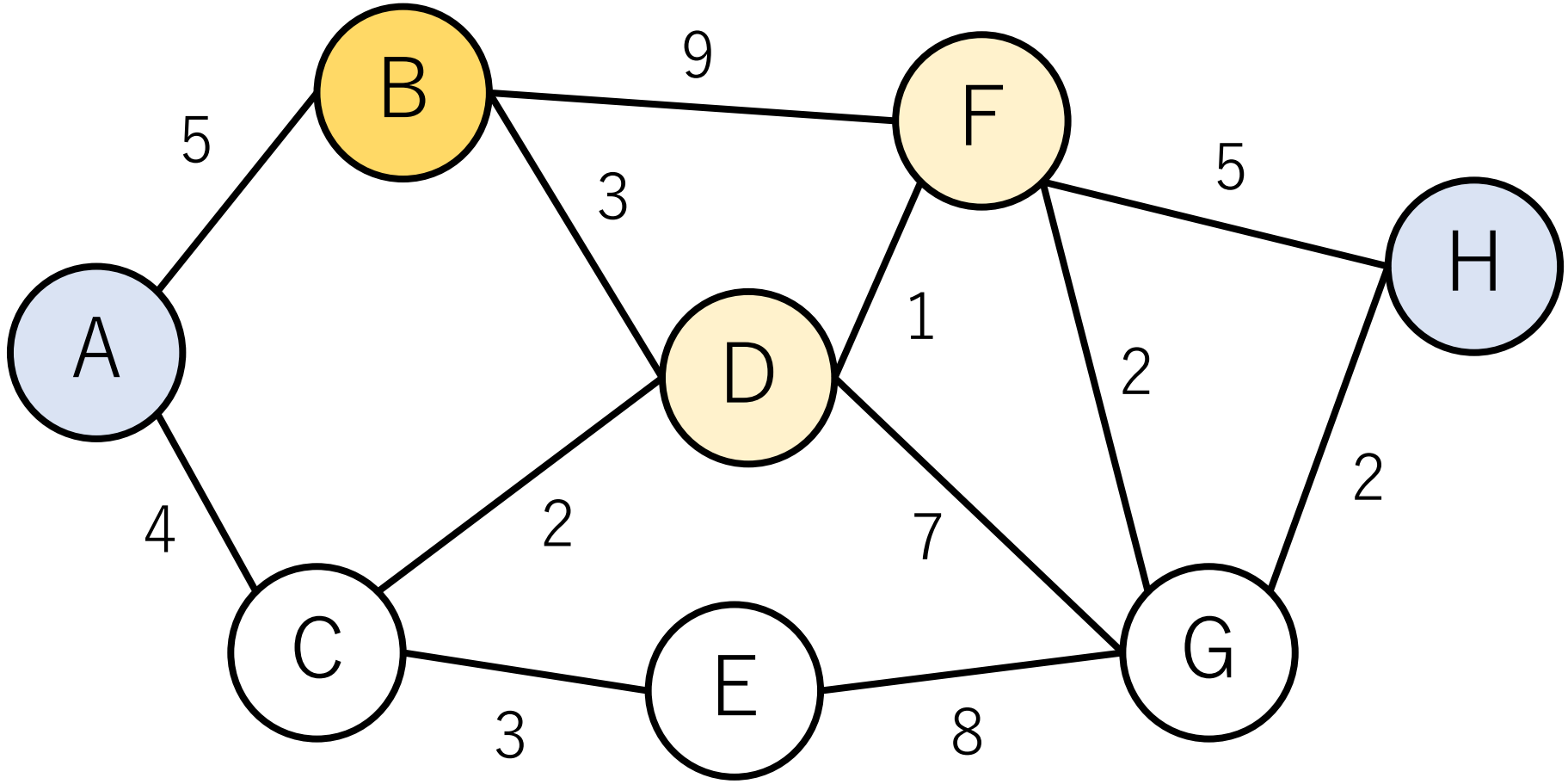
以下では，ダイクストラにおける「DPテーブル」を見てみましょう．



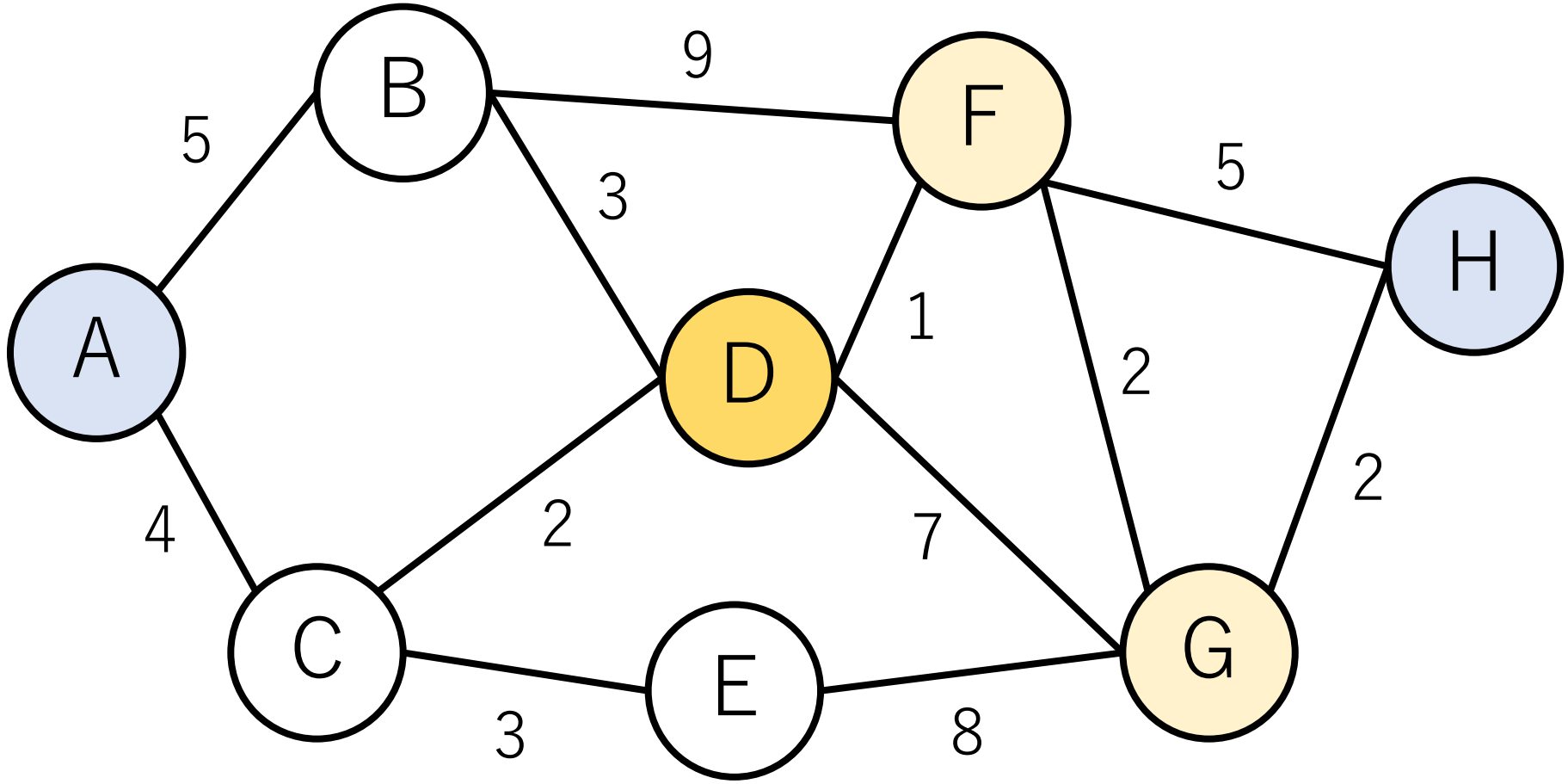
A	B	C	D	E	F	G	H
0	5	4					
	0+5	0+4					



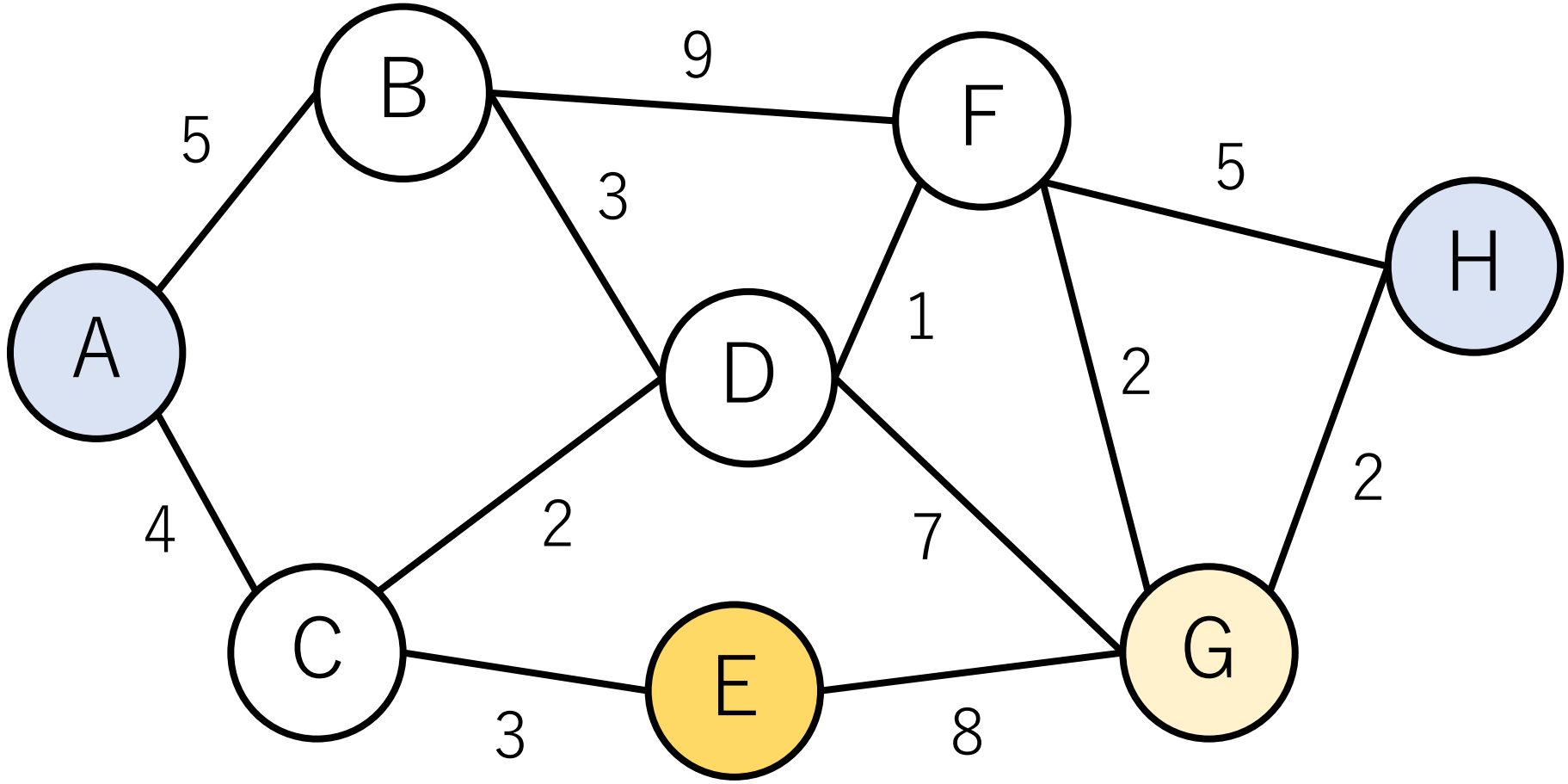
A	B	C	D	E	F	G	H
0	5	4	**6**	**7**			
			4+2	4+3			



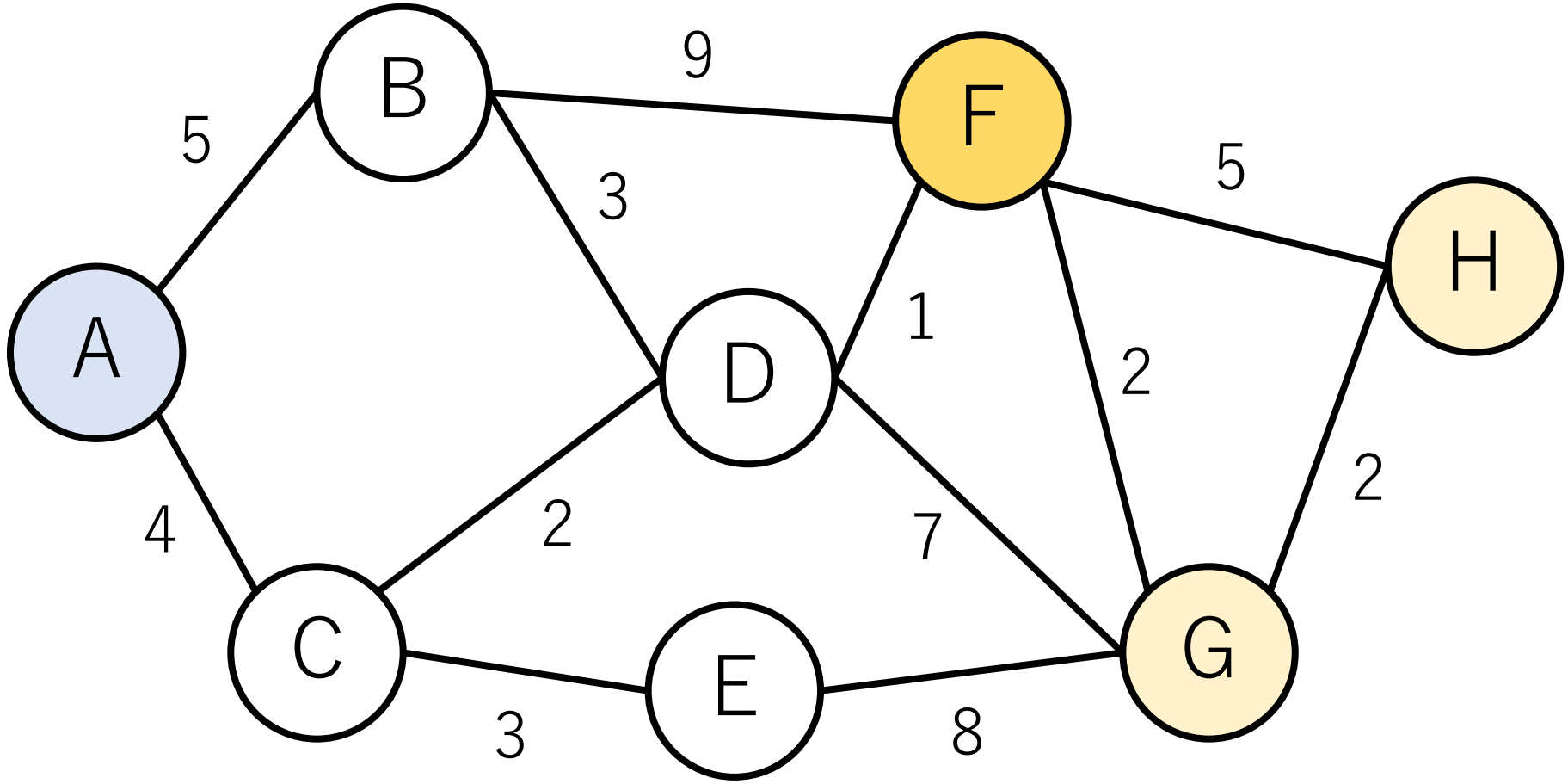
A	B	C	D	E	F	G	H
0	5	4	6	7	**14**		
			(5+3)		5+9		



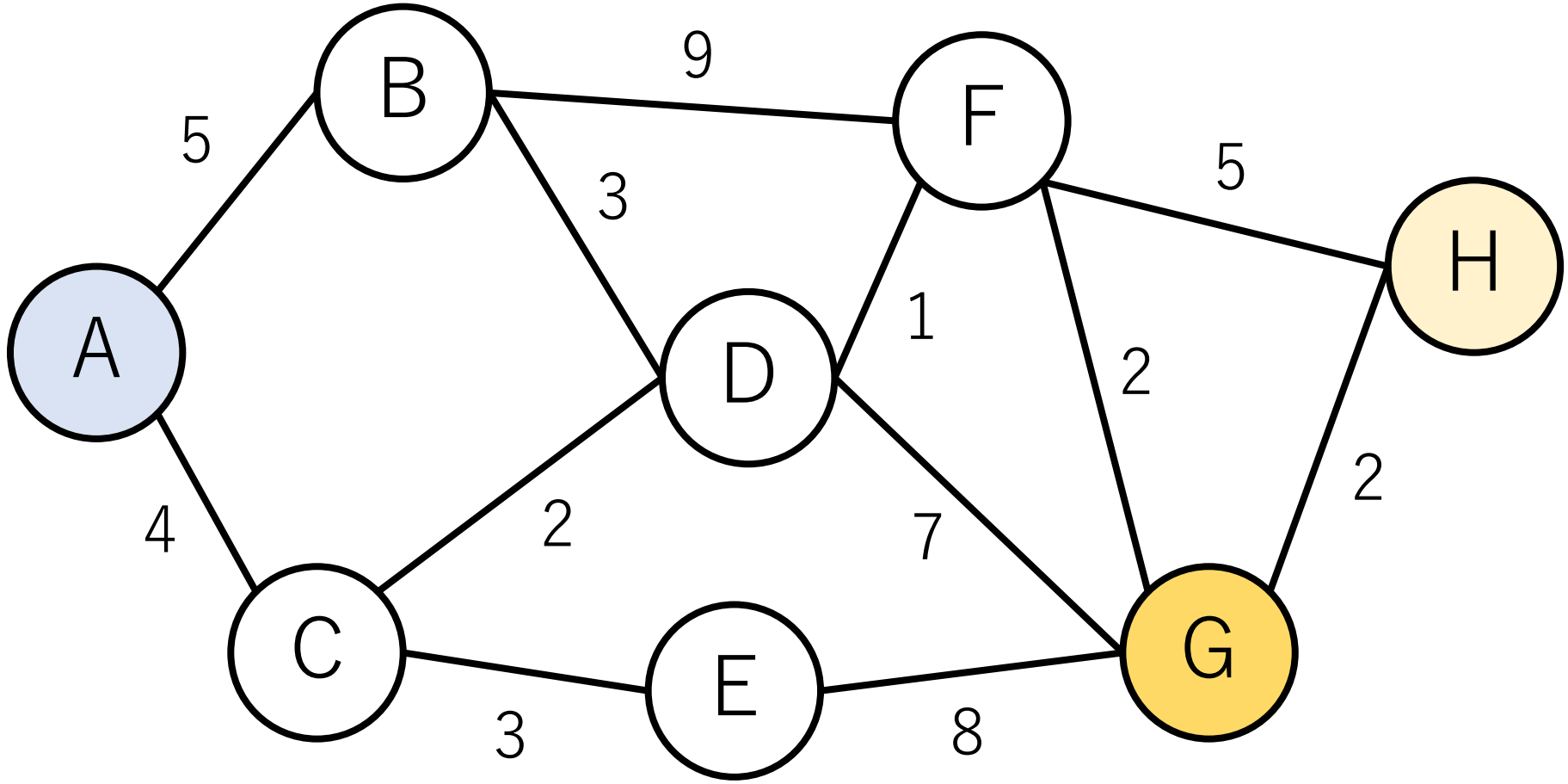
A	B	C	D	E	F	G	H
0	5	4	6	7	**7**	**13**	
					6+1	6+7	



A	B	C	D	E	F	G	H
0	5	4	6	7	7	13	
						(7+8)	

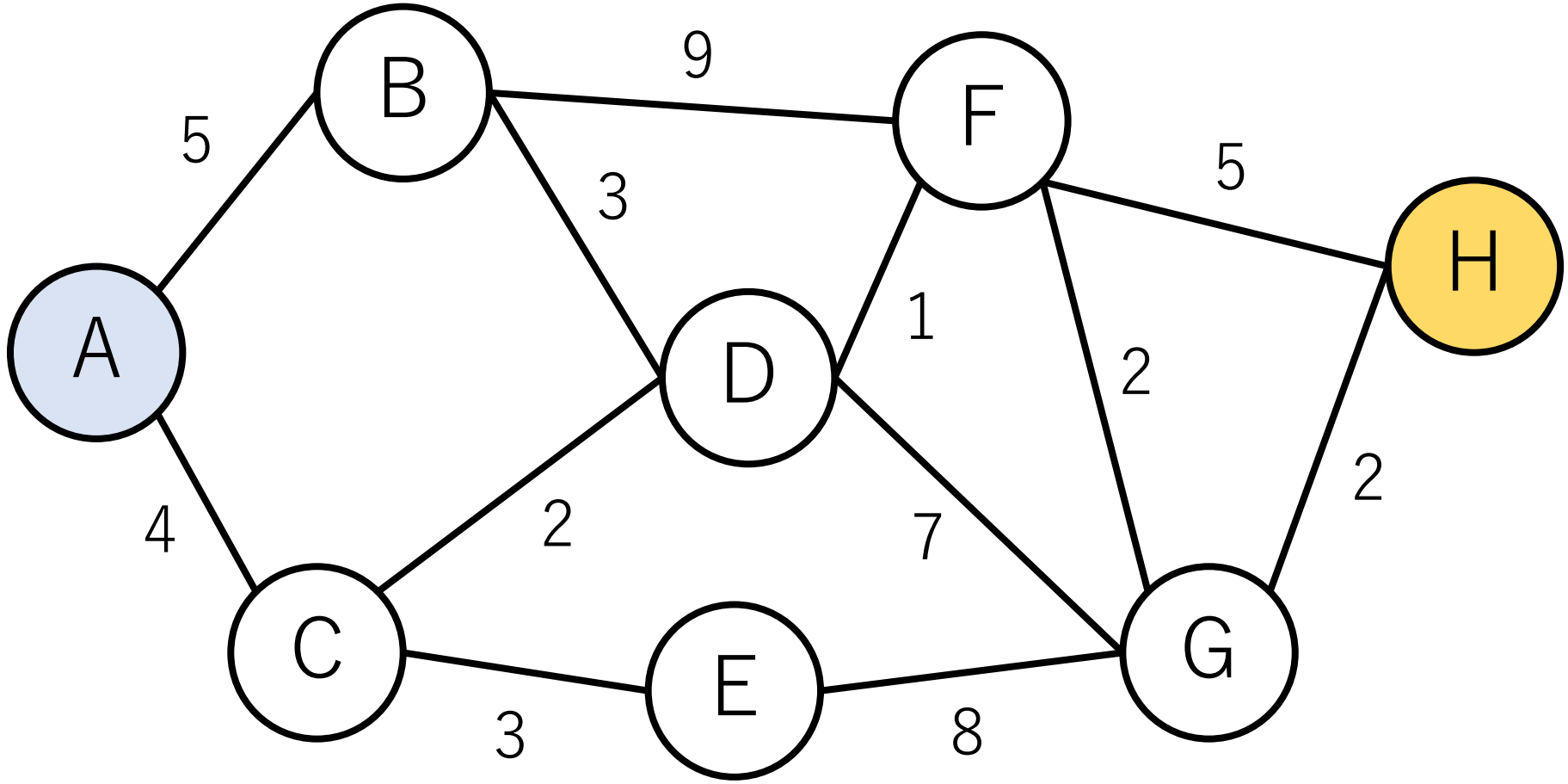


A	B	C	D	E	F	G	H
0	5	4	6	7	7	**9**	**12**
						7+2	7+5



A	B	C	D	E	F	G	H
0	5	4	6	7	7	9	**11**

9+2



A	B	C	D	E	F	G	H
0	5	4	6	7	7	9	11

コードチャレンジ：基本課題#10-a [1.5点]

ダイクストラ法において，開始ノードからその他の
全てのノードの最短経路を返すプログラムを書いて
ください．

開始ノードは一番最初のノードとは限らないことに
注意してください．

この実装では優先度付きキューを使う必要は必ずしも
ありません．（もちろん使ってもらっても良いです）

コードチャレンジ：基本課題#10-b [1.5点]

スライドで紹介した実装例に従って，ワーシャル・フロイド法を実装してください．

コードチャレンジ：Extra課題#10 [3点]

最短経路問題に対するアルゴリズムを応用する問題.