教科書輪講

プログラミングコンテスト攻略のための
アルゴリズムとデータ構造

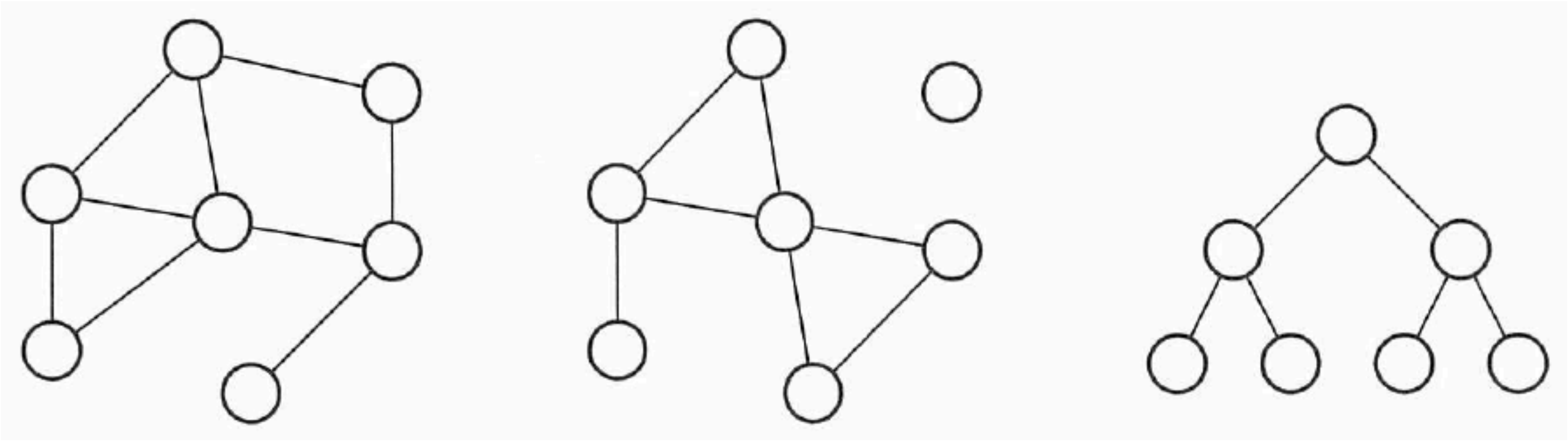
第12章 グラフ

石田研究室 M1
西岡知剛

グラフ

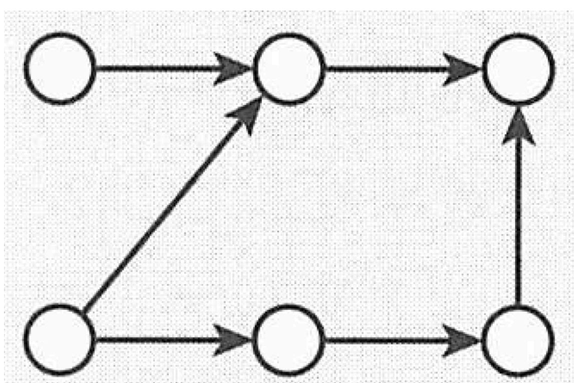
グラフとは・・・

- 「対象の集合とそれらのつながり(関係)の集合」
- 「対象」…ノード、頂点
- 「つながり」…エッジ、辺

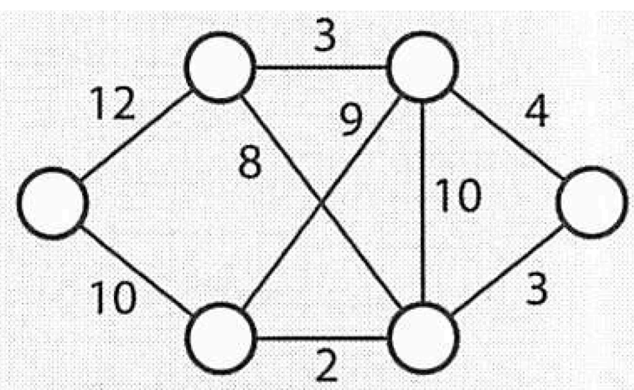


グラフの種類

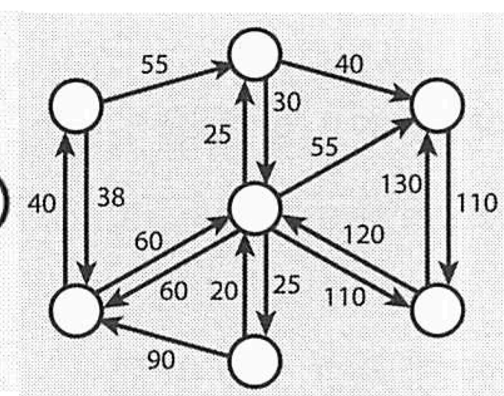
名前	特徴
無向グラフ	エッジに方向がないグラフ
有向グラフ	エッジに方向があるグラフ
重み付き無向グラフ	エッジに重み（値）があり、方向がないグラフ
重み付き有向グラフ	エッジに重み（値）があり、方向があるグラフ



有向グラフ



重み付き無向グラフ



重み付き有向グラフ

グラフの表記と用語(1)

- 頂点集合をV、辺の集合がEであるグラフ $G = (V, E)$
  (頂点の数) $= |V|$
  (辺の数) $= |E|$

- 2つの頂点u, vを結ぶ辺 $\mathrm{e} = (u, v)$
  無向グラフの場合、$(\mathrm{u}, \mathrm{v}) = (v, u)$

- 重み付きグラフの辺(u, v)の重み $w(u, v)$

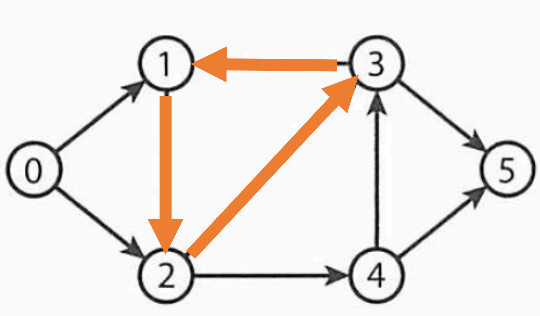
- 辺(u, v)が存在する　⇔　uとvは隣接している(adjacent)

グラフの表記と用語(2)

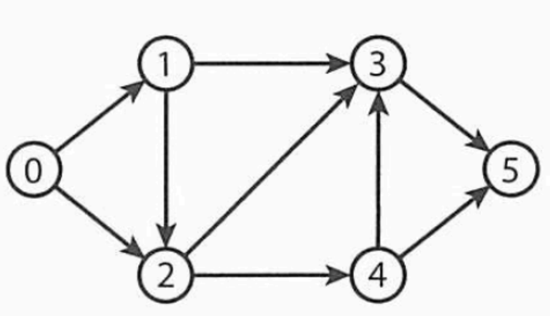
- 経路(パス, path)
  隣接している頂点の列$v_0, v_1, \ldots, v_k$($i = 1,2,3, \ldots, k$で(v_{i-1}, v_i)が存在)
  単純経路…頂点の重複がない経路

- 閉路(cycle)
  始点と終点が同じ単純経路

- Directed Acyclic Graph(DAG)
  閉路のない有向グラフ



DAGでない



DAG

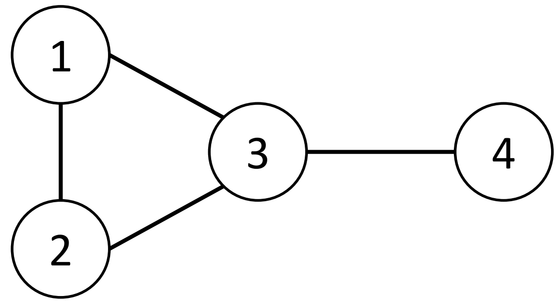
グラフの表記と用語(3)

- 次数(degree)
  頂点uに繋がっている辺の数
  有向グラフの場合、頂点uに入る辺の数を入次数、
  頂点uから出る辺の数を出次数という

- 連結グラフ
  任意の2つの頂点についてパスが存在するグラフ

- 部分グラフ
  2つのグラフ$G = (V, E)$と$G' = (V', E')$について、
  $V \supseteq V'$, $\mathrm{E} \supseteq E'$であるとき、G'をGの部分グラフという

隣接リストと隣接行列



隣接リスト

1	{2, 3}
2	{1, 3}
3	{1, 2, 3}
4	{3}

隣接行列

	1	2	3	4
1	0	1	1	0
2	1	0	1	0
3	1	1	0	1
4	0	0	1	0

メリット: メモリ効率　高
頂点の次数を求める　速
デメリット: 特定の辺を求める O(n)

メリット: 特定の辺を求める O(1)
デメリット: メモリ効率　低
(特に疎行列)
頂点の次数を求める 遅

演習問題(1) | グラフの表現

入力:

最初の行にGの頂点数nが与えられる
続くn行で各頂点uの隣接リスト$Adj[u]$が以下の形式で与えられる

$u\ k\ v_1\ v_2\ \ldots v_k$
（uは頂点の番号、kはuの次数、$v_1\ v_2\ \ldots v_k$はuに隣接する頂点の番号）

出力:

出力例に従い、Gの隣接行列を出力せよ
a_{ij}の間に1つの空白を入れること

制約:

$1 \leq n \leq 100$

演習問題(1) | グラフの表現

入力例:

```
4
1 2 2 4
2 1 4
3 0
4 1 3
```

出力例:

```
0 1 0 1
0 0 0 1
0 0 0 0
0 0 1 0
```

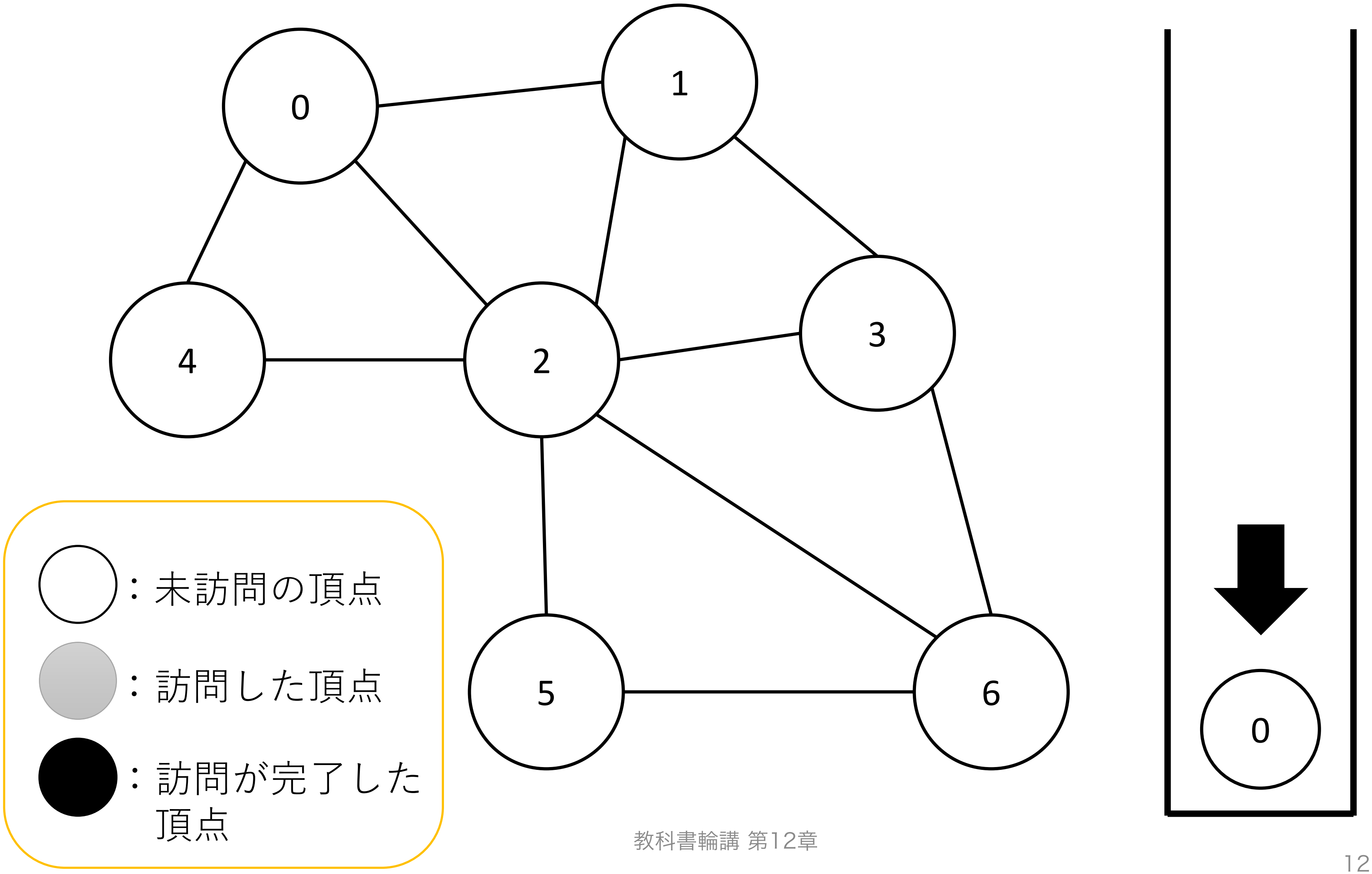
実装例 | グラフの表現

```python
# -*- coding: utf-8 -*-
def main():

    n = int(input())
    adj = [[0 for i in range(n)] for j in range(n)]

    for i in range(n):
        tmp = list(map(int, input().split()))
        u = tmp[0] - 1 #ノード番号を0オリジンに
        k = tmp[1]
        v = tmp[2:]
        for j in range(k):
            adj[u][v[j]-1] = 1

    for i in adj:
        print(*i)


if __name__ == "__main__":
    main()
```

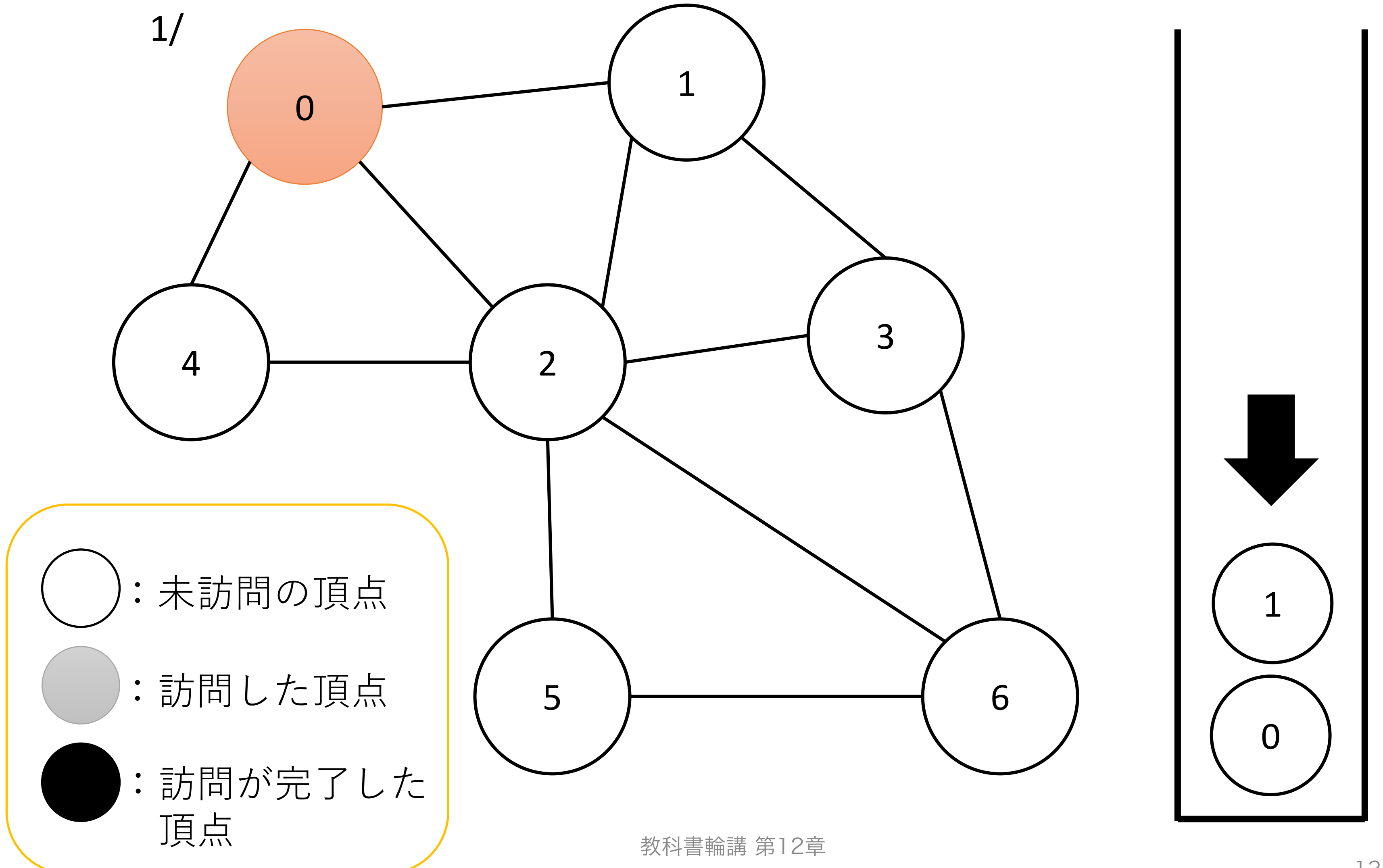
深さ優先探索(Depth First Search)

1. 一番最初に訪問する頂点をスタックに入れておく
2. スタックに頂点が積まれている限り、以下の処理を繰り返す
   - ➢スタックのトップにある頂点uを訪問
   - ➢現在訪問中の頂点uに未訪問の隣接する頂点がなければuをスタックから削除する

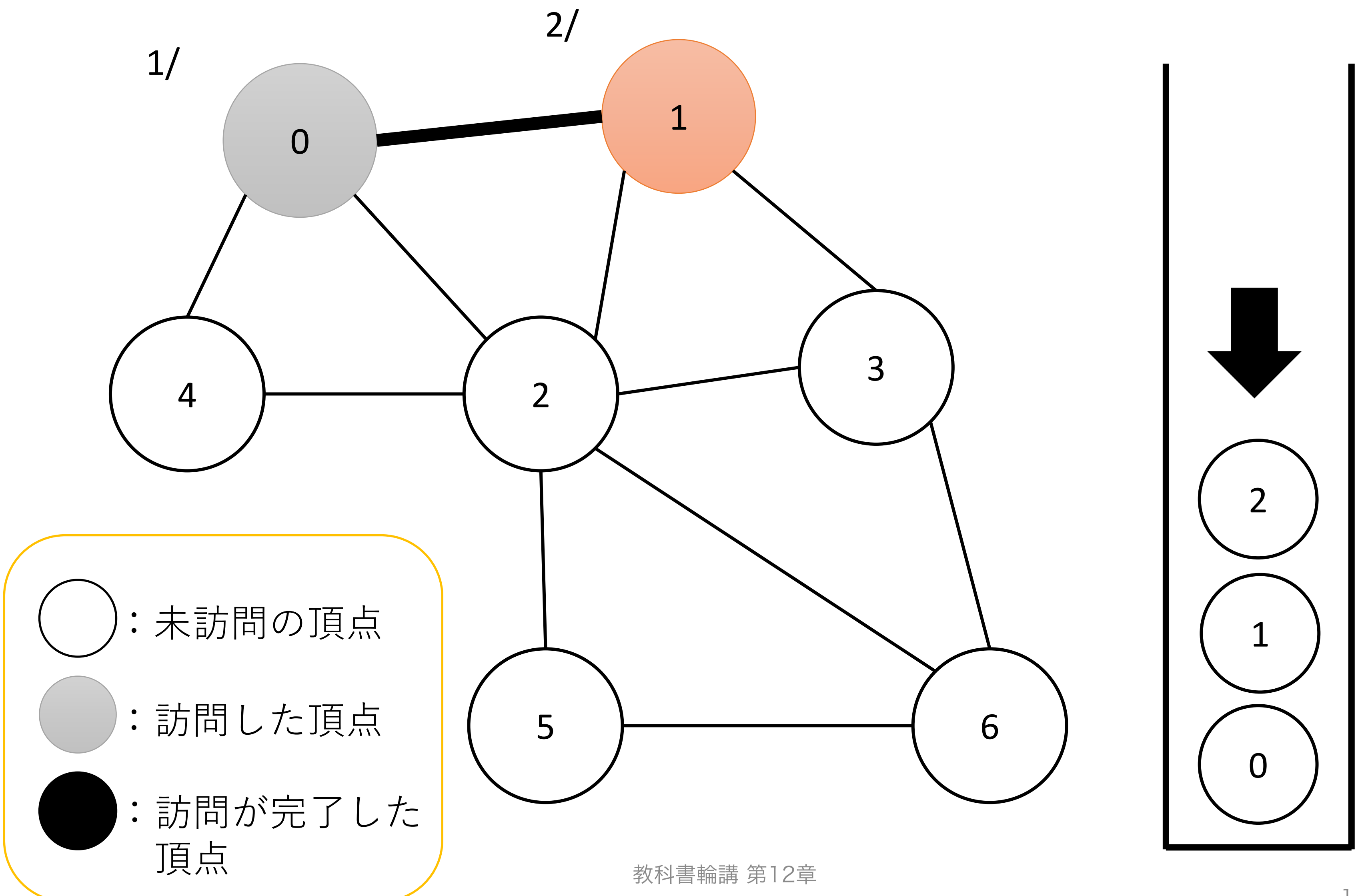
深さ優先探索の例



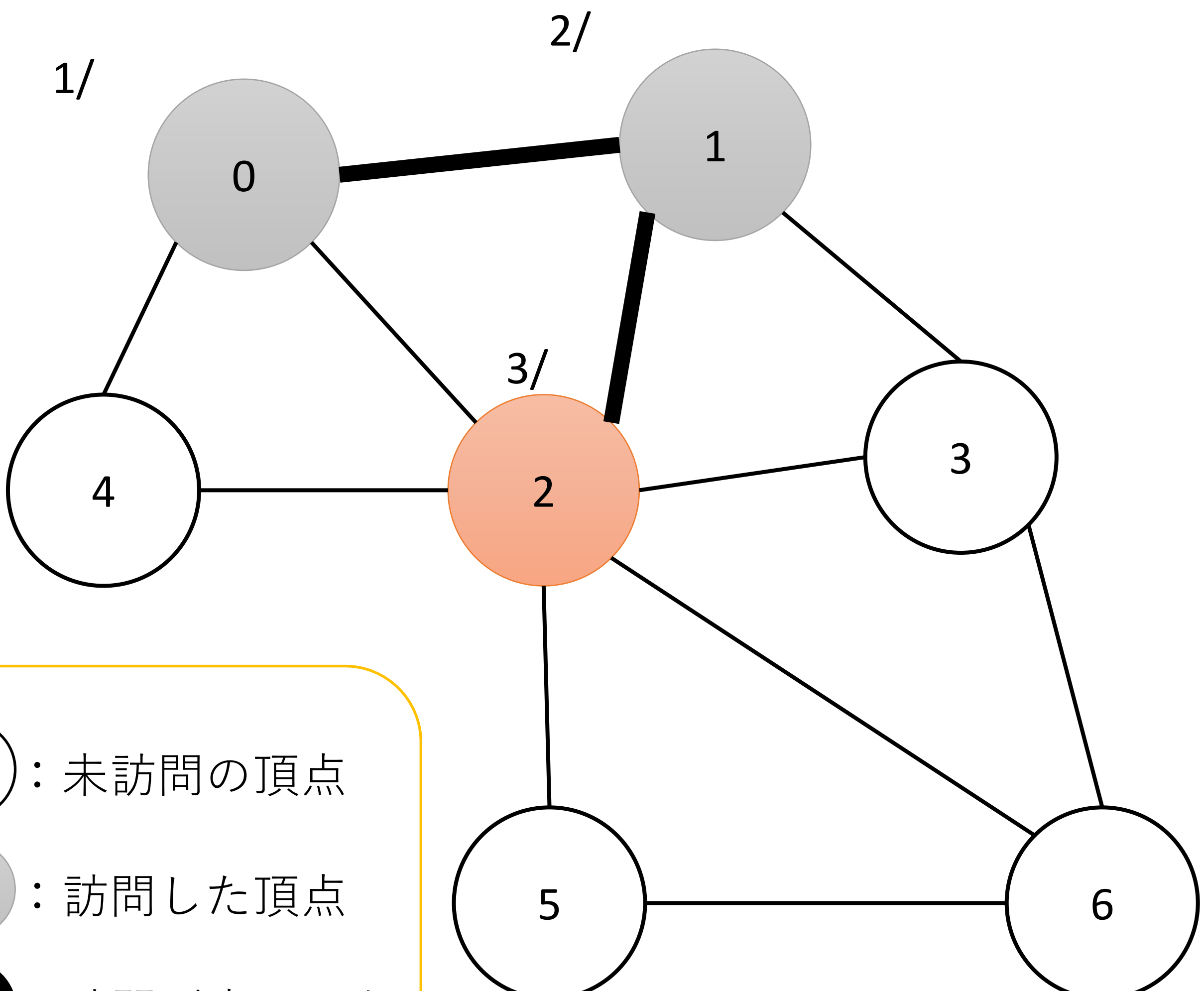
深さ優先探索の例

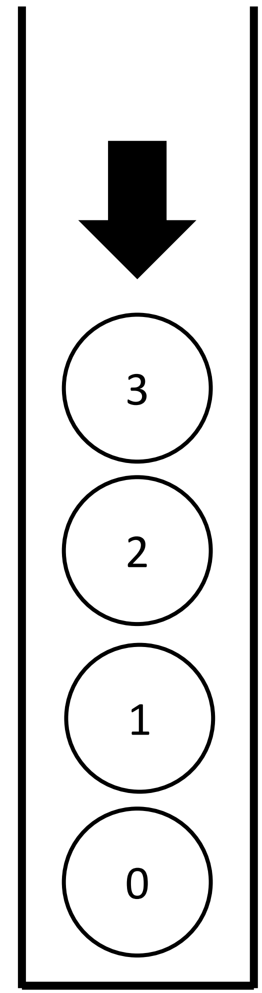


深さ優先探索の例



深さ優先探索の例



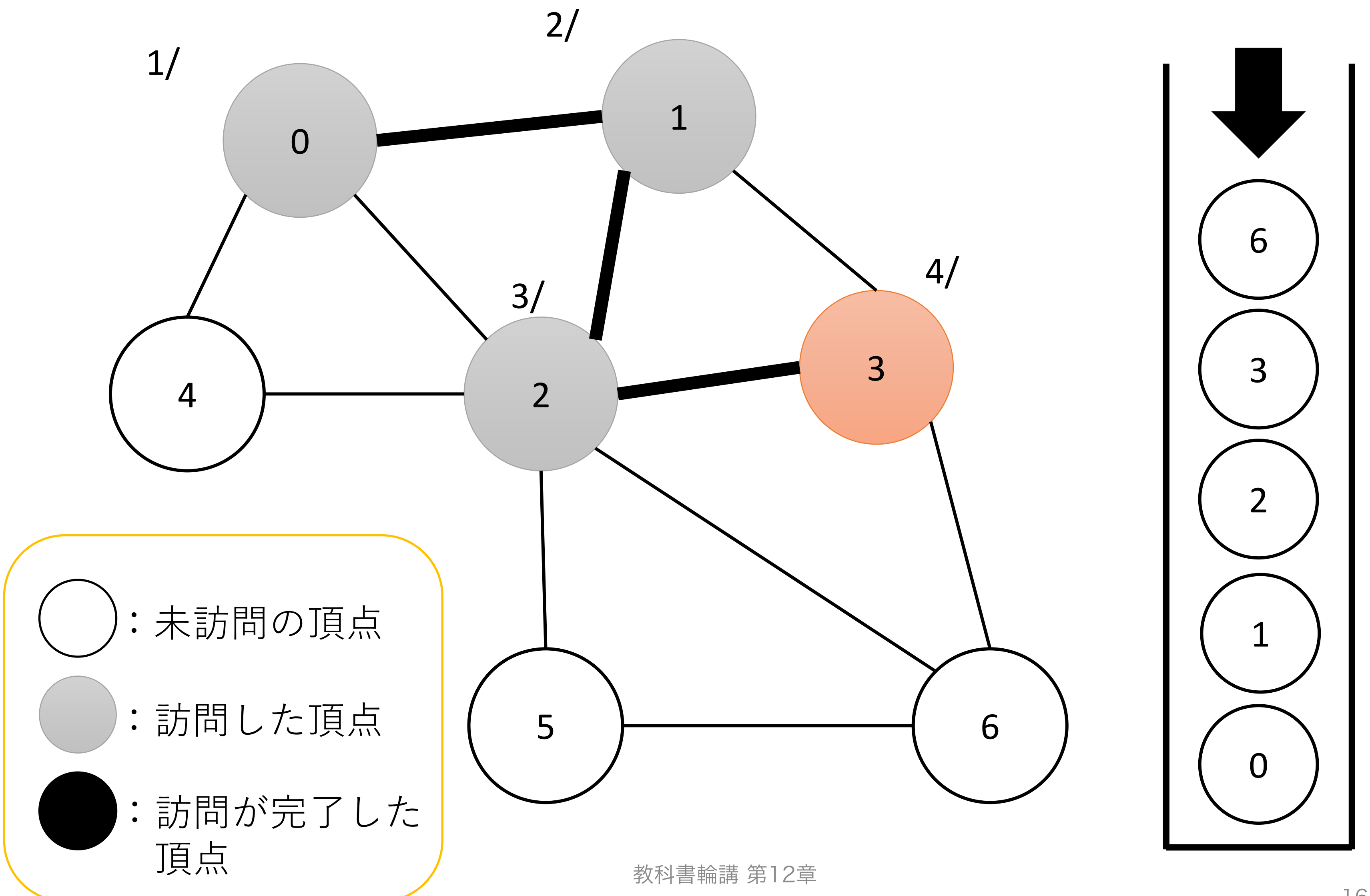


：未訪問の頂点

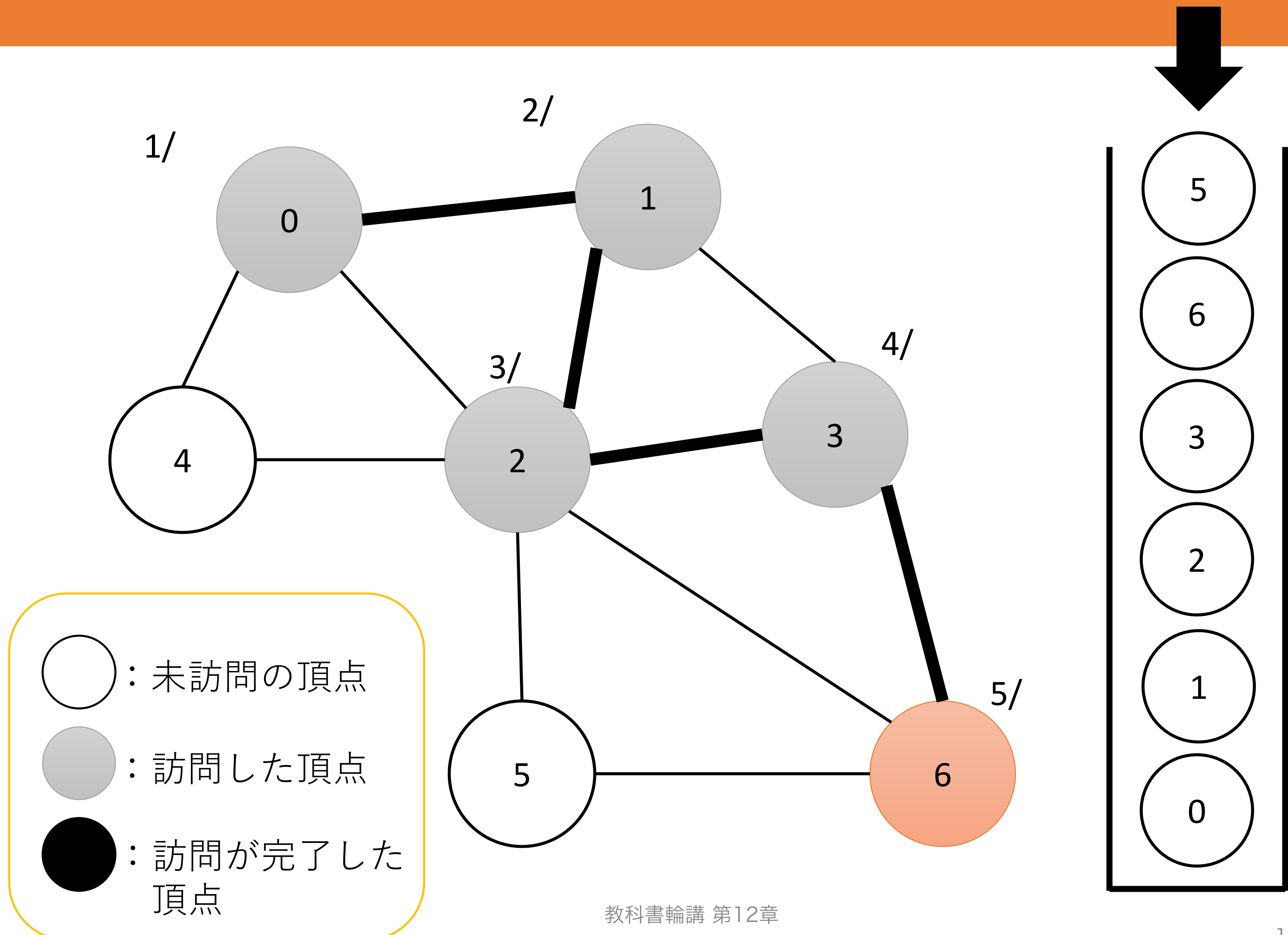
：訪問した頂点

：訪問が完了した頂点

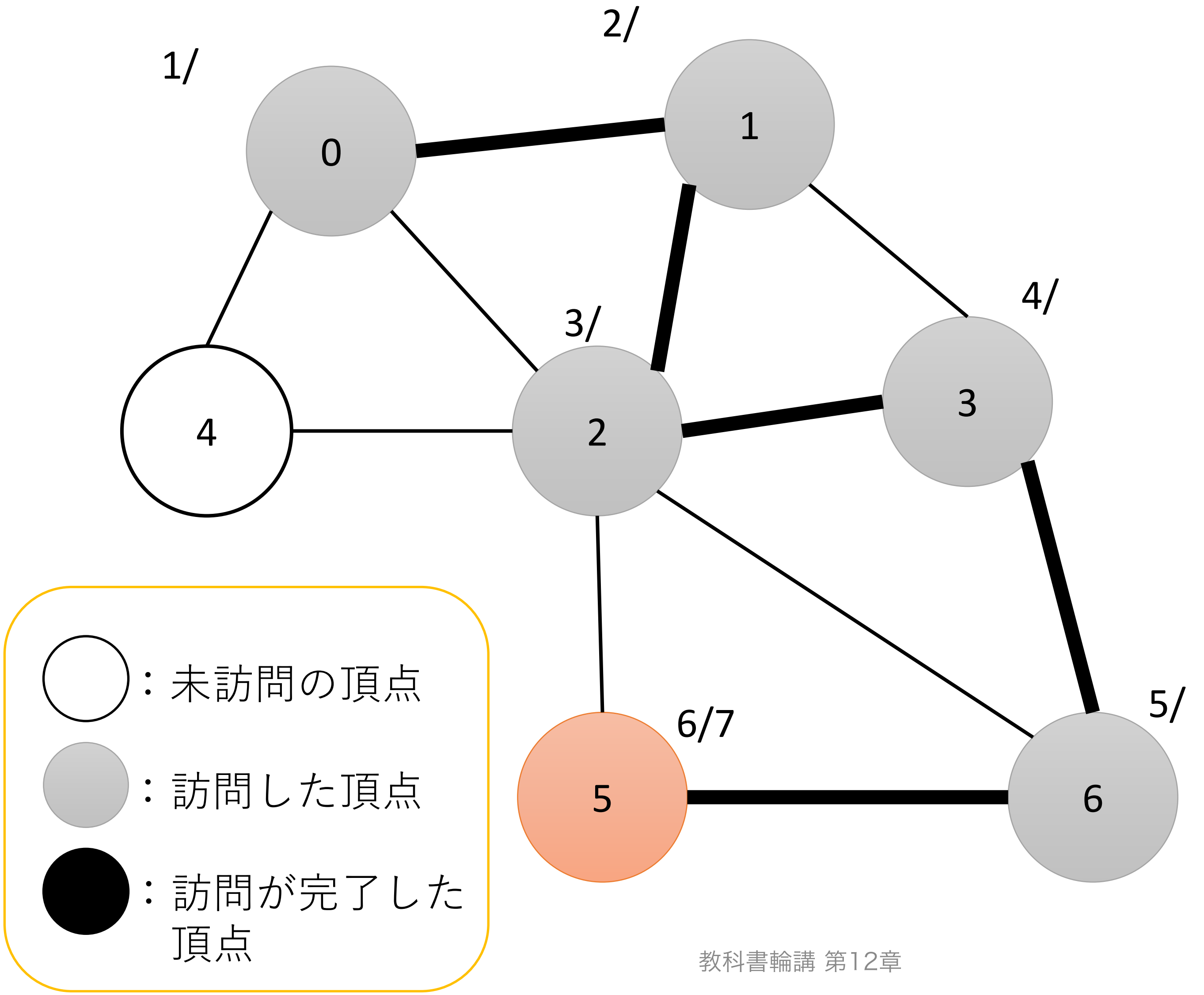
深さ優先探索の例

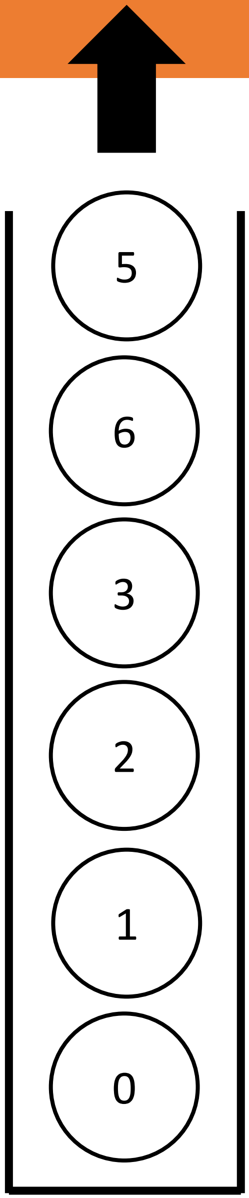


深さ優先探索の例

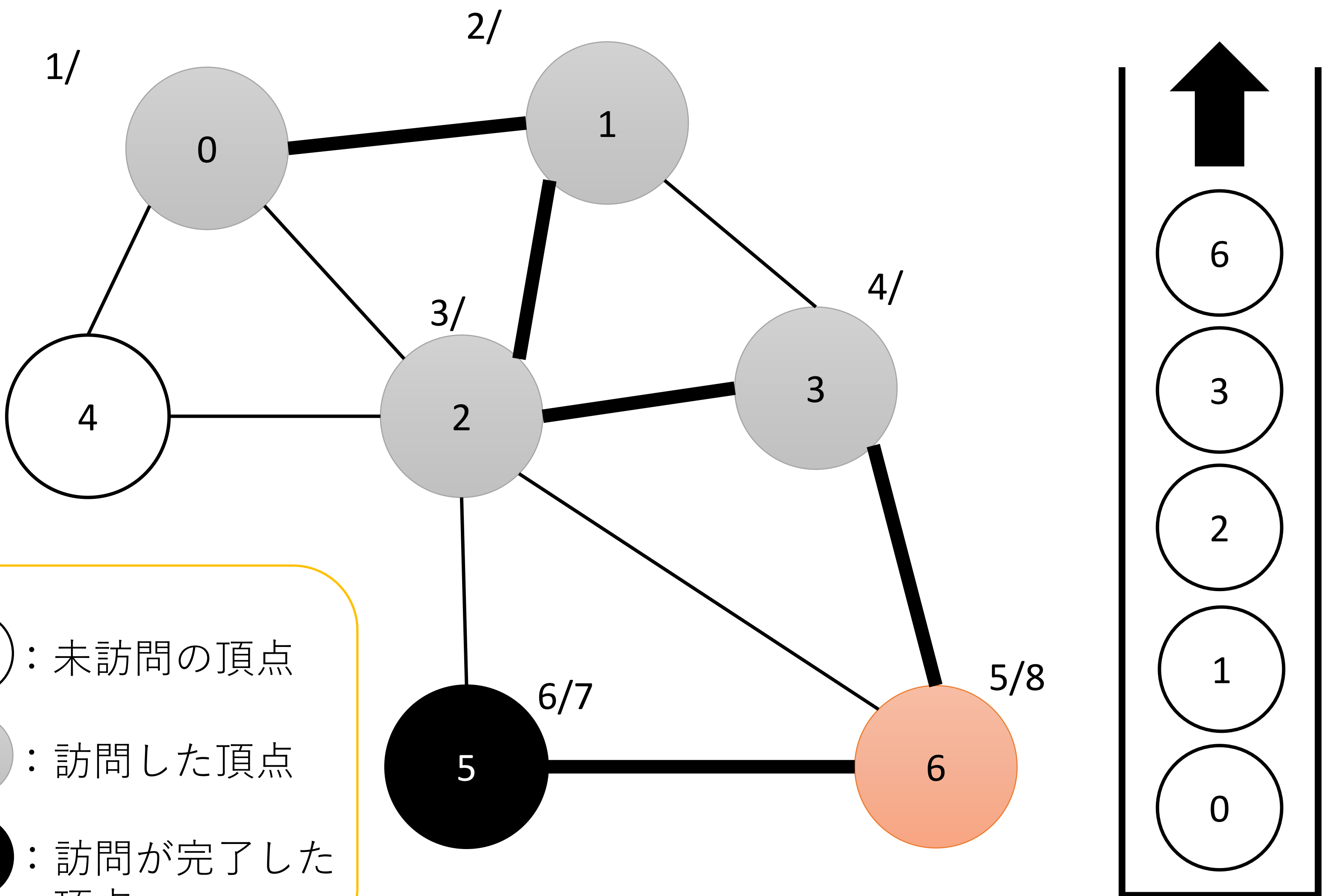


深さ優先探索の例



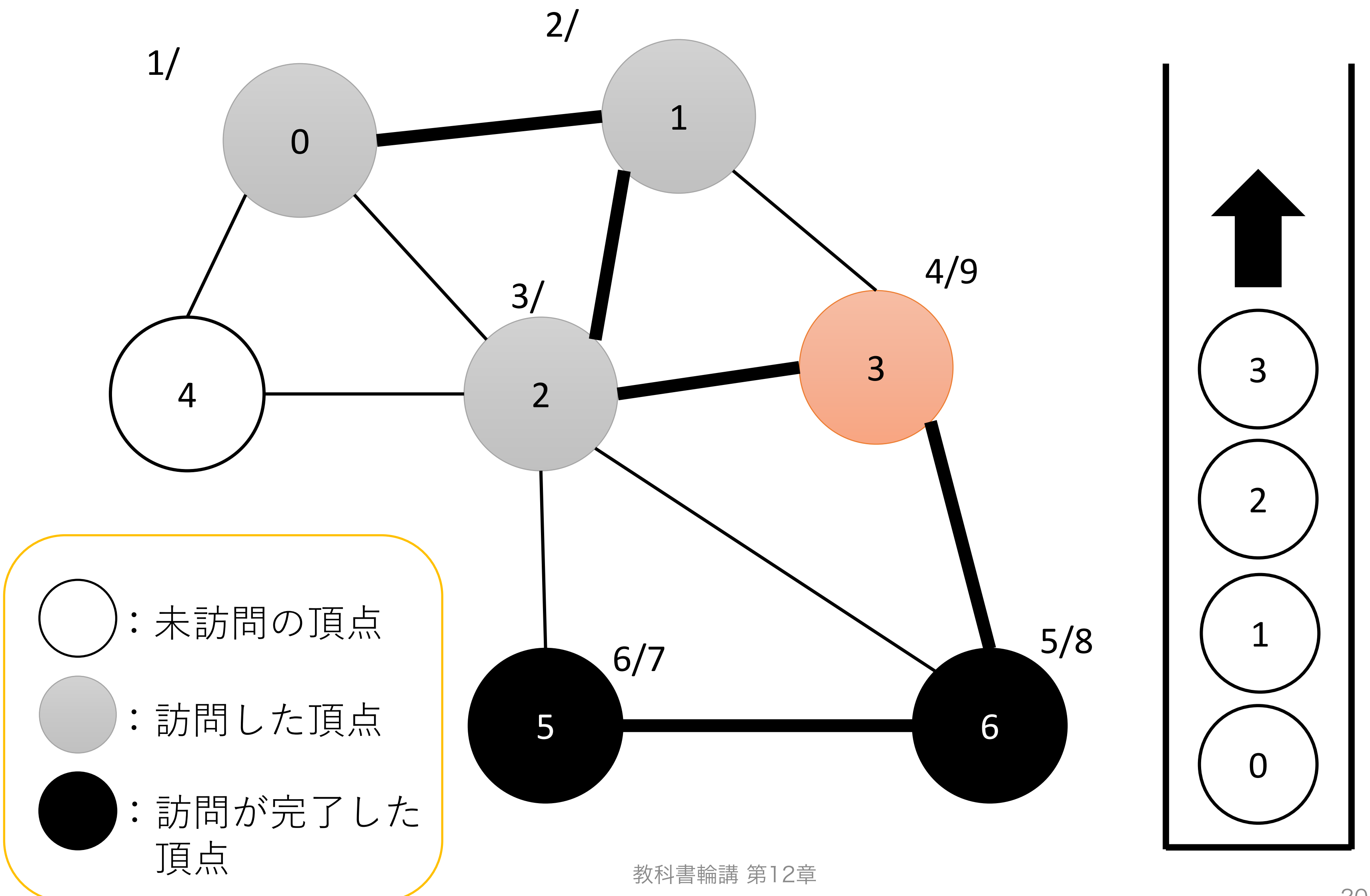


深さ優先探索の例

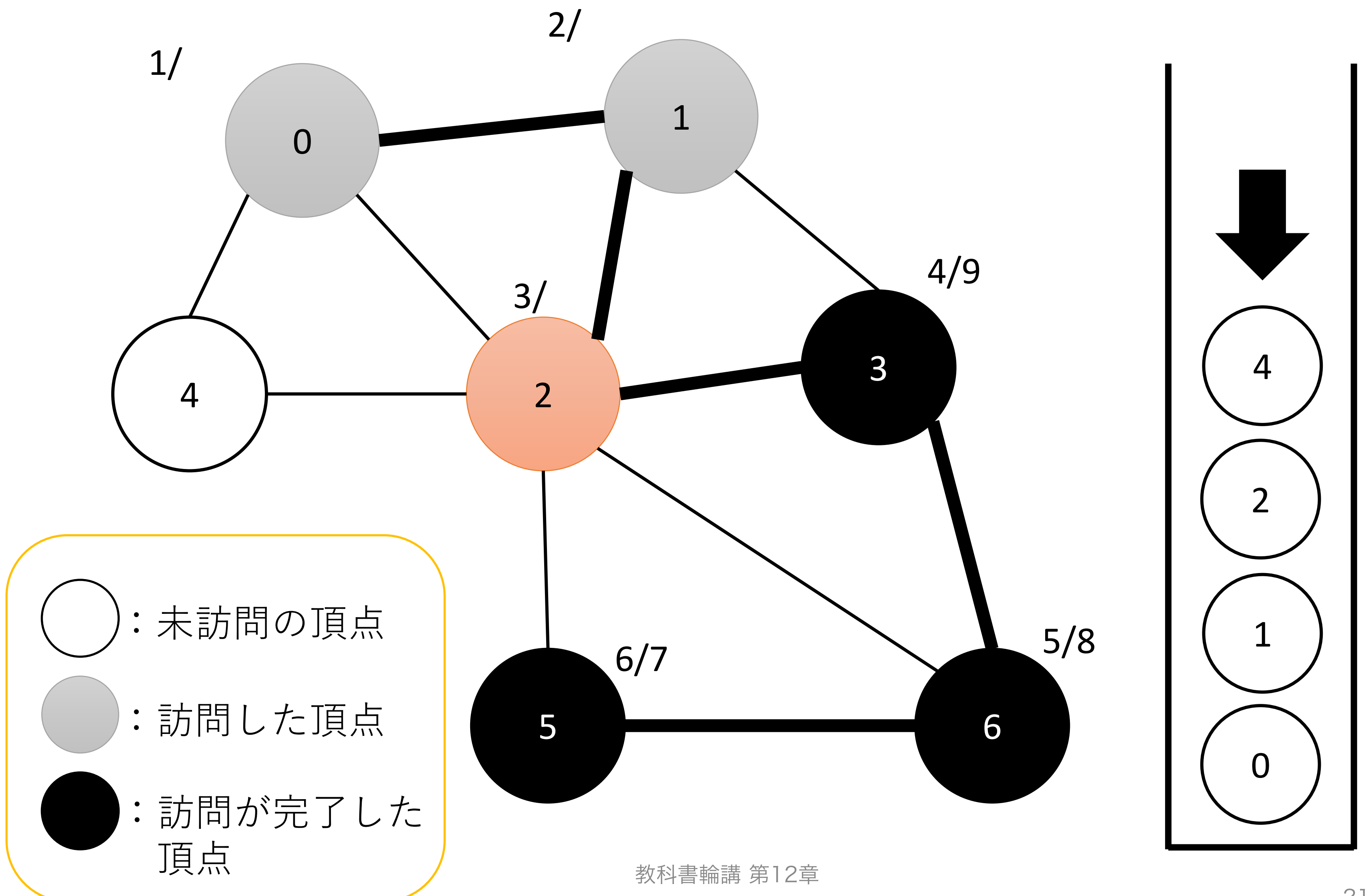


：未訪問の頂点
：訪問した頂点
：訪問が完了した頂点

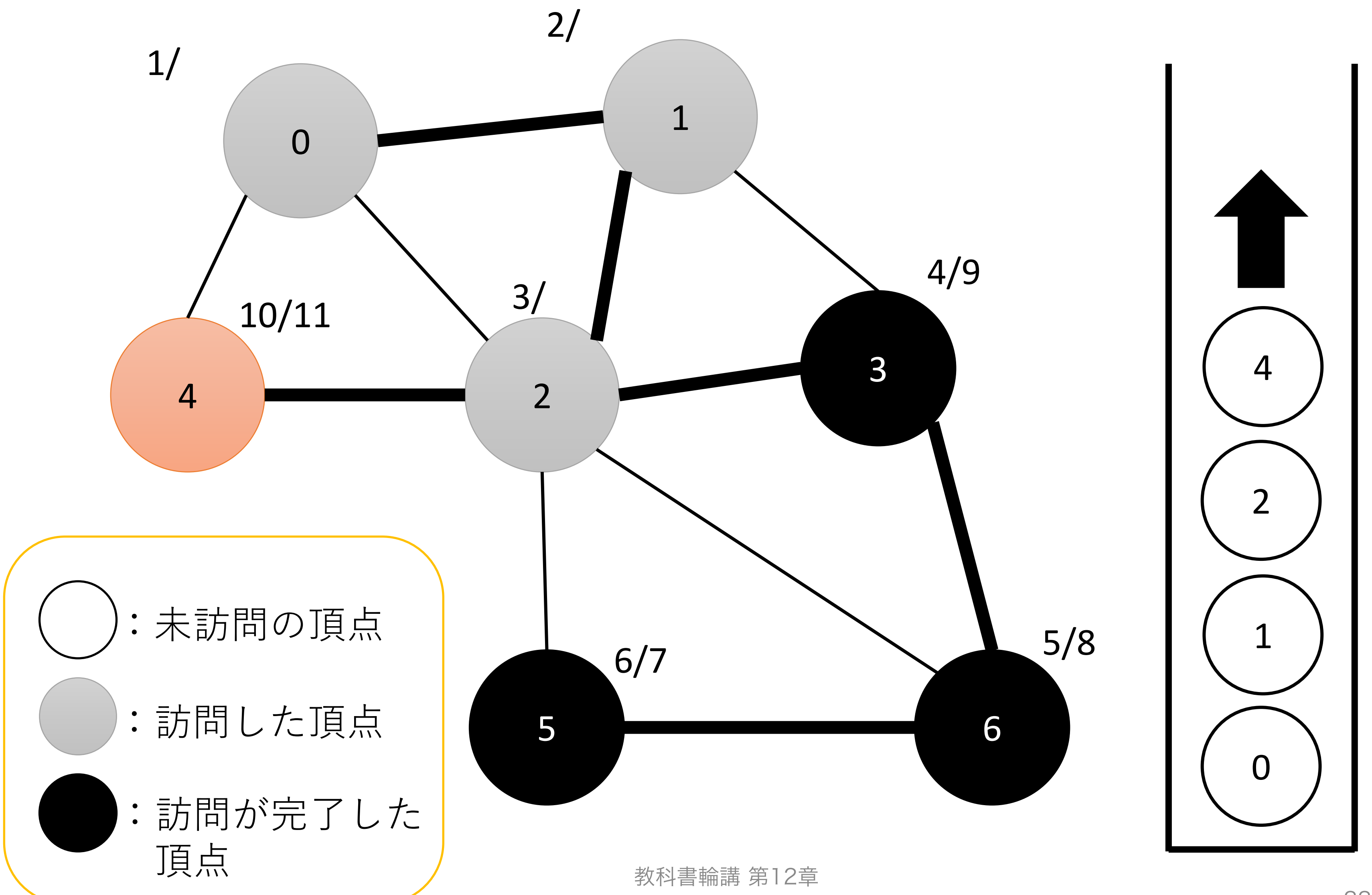
深さ優先探索の例



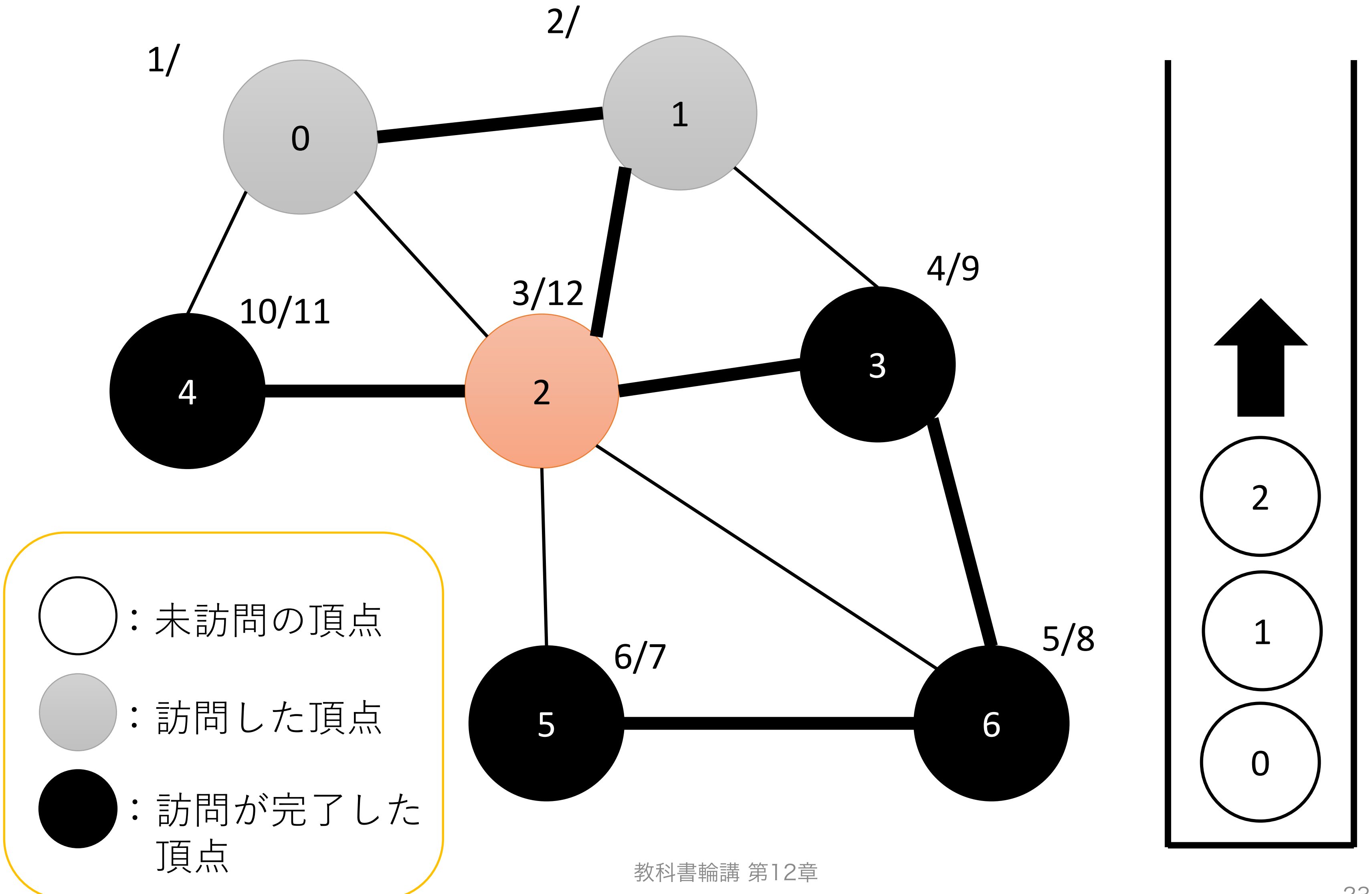
深さ優先探索の例



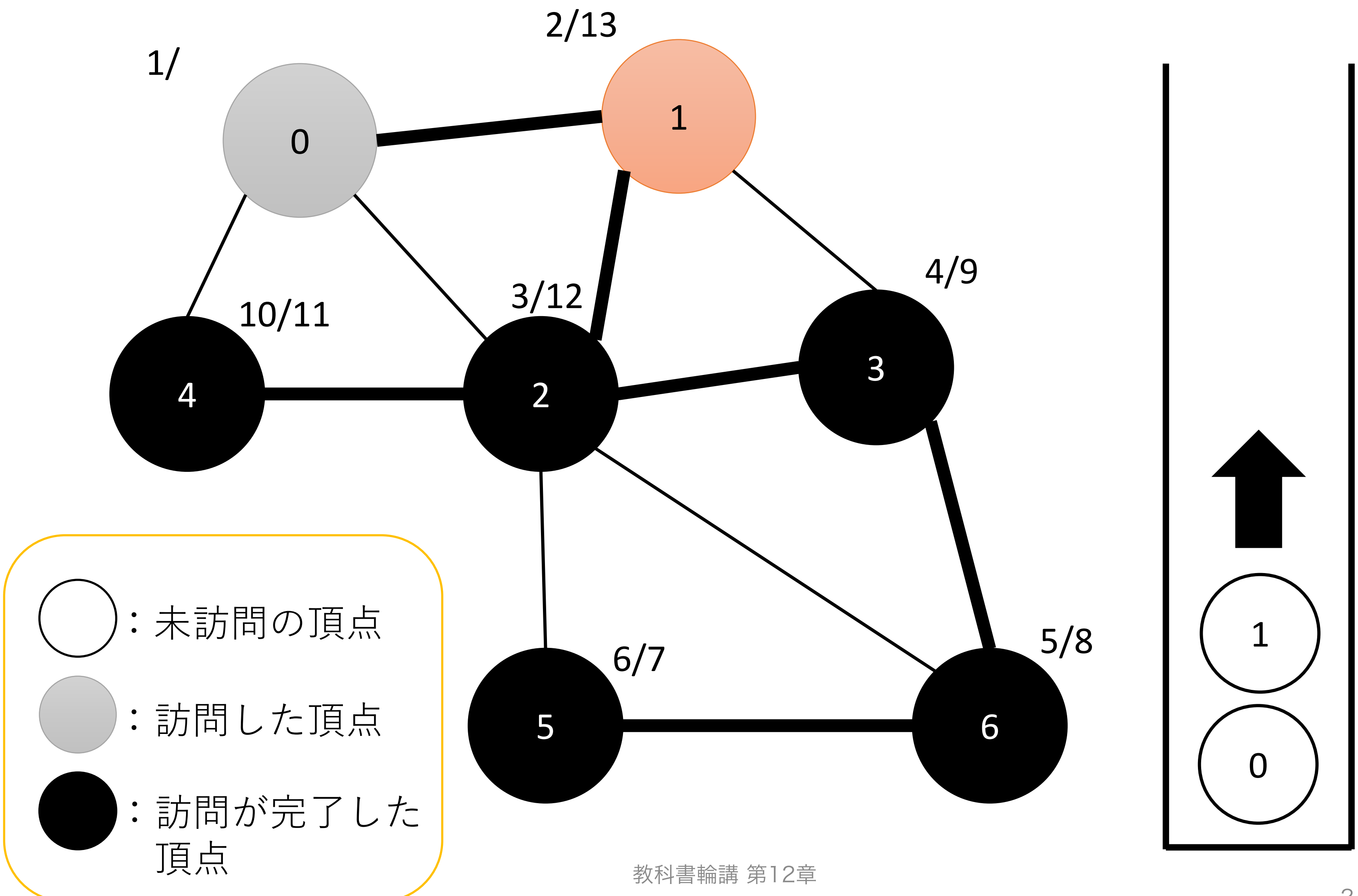
深さ優先探索の例



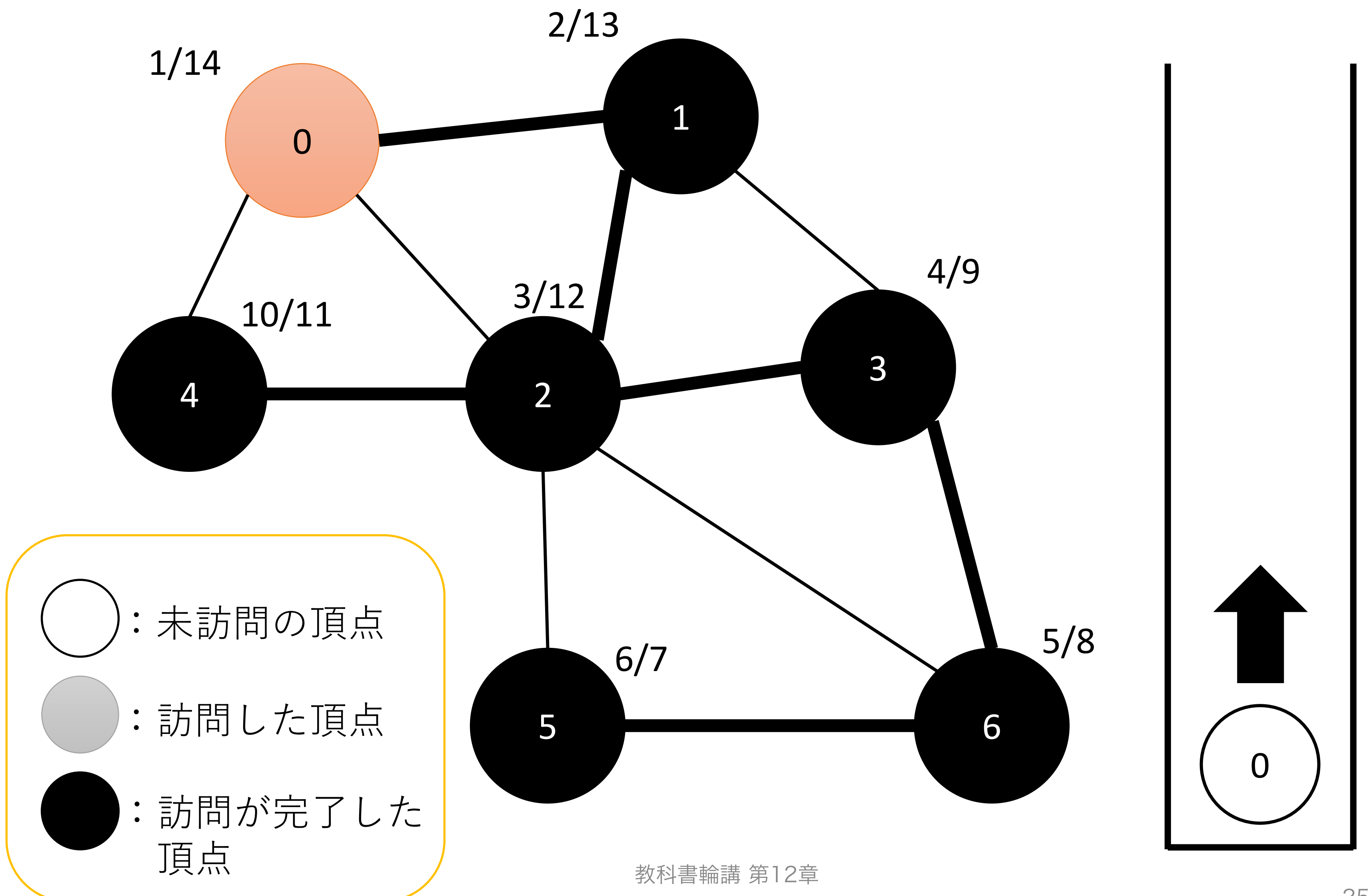
深さ優先探索の例



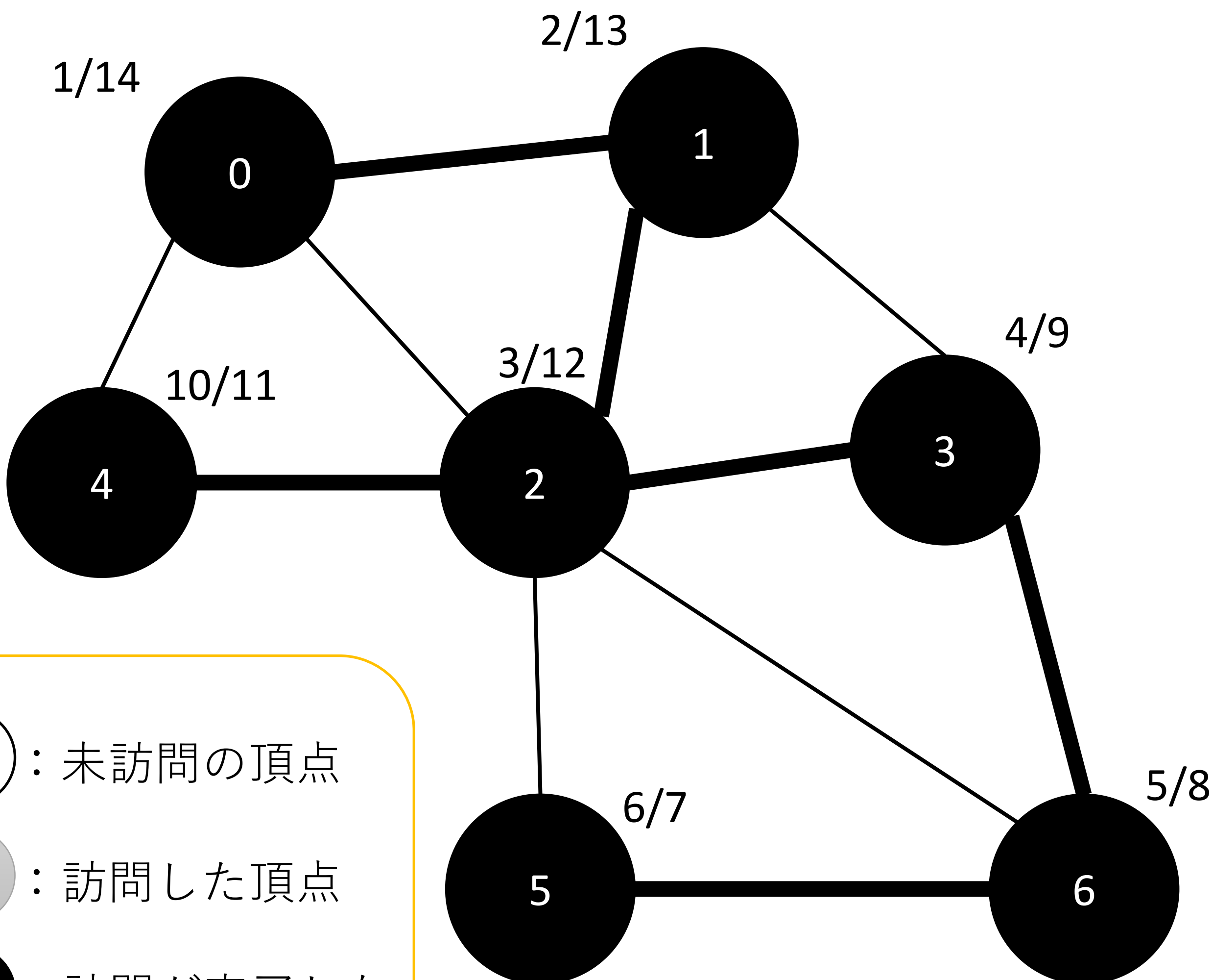
深さ優先探索の例

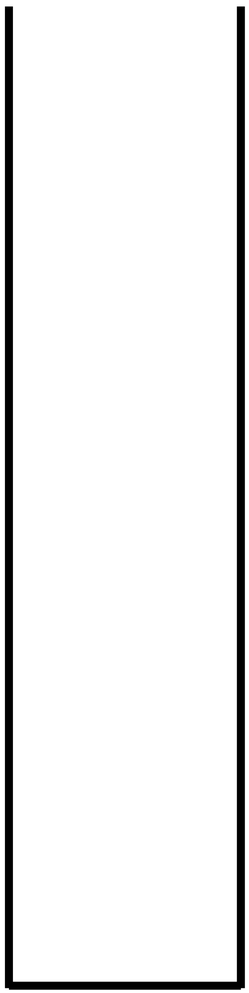


深さ優先探索の例



深さ優先探索の例





：未訪問の頂点

：訪問した頂点

：訪問が完了した頂点

演習問題(2) | 深さ優先探索

入力:

最初の行にGの頂点数nが与えられる
続くn行で各頂点uの隣接リスト$Adj[u]$が以下の形式で与えられる

$u\ k\ v_1\ v_2\ \ldots v_k$
（uは頂点の番号、kはuの次数、$v_1\ v_2\ \ldots v_k$はuに隣接する頂点の番号）

出力:

各頂点についてid, d, fを空白区切で1行に出力せよ
idは頂点の番号、dはその頂点の発見時刻、fはその頂点の完了時刻を表す
頂点の番号順で出力せよ

制約:

$1 \leq n \leq 100$

演習問題(2) | 深さ優先探索

入力例:

```
6
1 2 2 3
2 2 3 4
3 1 5
4 1 6
5 1 6
6 0
```

出力例:

```
1 1 12
2 2 11
3 3 8
4 9 10
5 4 7
6 5 6
```

解説 | 深さ優先探索

スタックを用いた実装

flagが
・未探索
・訪問済み
・訪問完了
の３状態を管理

```python
def DFS_stack(adj, start):
    n = len(adj)
    d = [0] * n
    f = [0] * n
    flag = [0] * n # 0:未訪問, 1:訪問済み, 2:訪問完了
    S = []
    time = 1

    S.append(start)
    flag[start] = 1
    d[start] = time
    time = time + 1

    while flag.count(2) != n: # 全てのノードが探索終了になるまで
        if len(S) != 0:
            u = S.pop()
            v = [i for i, x in enumerate(adj[u]) if (x == 1) and (flag[i] == 0)
                ]
            # v := 隣接ノードのうち未探索のノード番号
            if len(v) != 0:
                S.append(u)
                S.append(v[0])
                flag[v[0]] = 1
                d[v[0]] = time
                time = time + 1
            else:
            #隣接ノードが全て訪問済み
                flag[u] = 2
                f[u] = time
                time = time + 1

        else:
            #スタックが空になった場合未探索のノードから再度探索開始
            u = flag.index(0)
            S.append(u)
            flag[u] = 1
            d[u] = time
            time = time + 1

    return d, f
```

解説 | 深さ優先探索

再帰を用いた実装

```
def DFS(adj, start):

    n = len(adj)
    d = [0] * n
    f = [0] * n
    flag = [0] * n
    time = 1

    def DFS_recursive(u, time):
        #print(u, flag)
        flag[u] = 1
        d[u] = time
        time = time + 1
        v = [i for i, x in enumerate(adj[u]) if x == 1]
        for i in v:
            if flag[i] == 0:
                time = DFS_recursive(i, time)
        flag[u] = 2
        f[u] = time
        time = time + 1
        return time

    time = DFS_recursive(start, time)
    for i in range(n):
        if flag[i] == 0:
            time =  DFS_recursive(i, time)

    return d, f
```

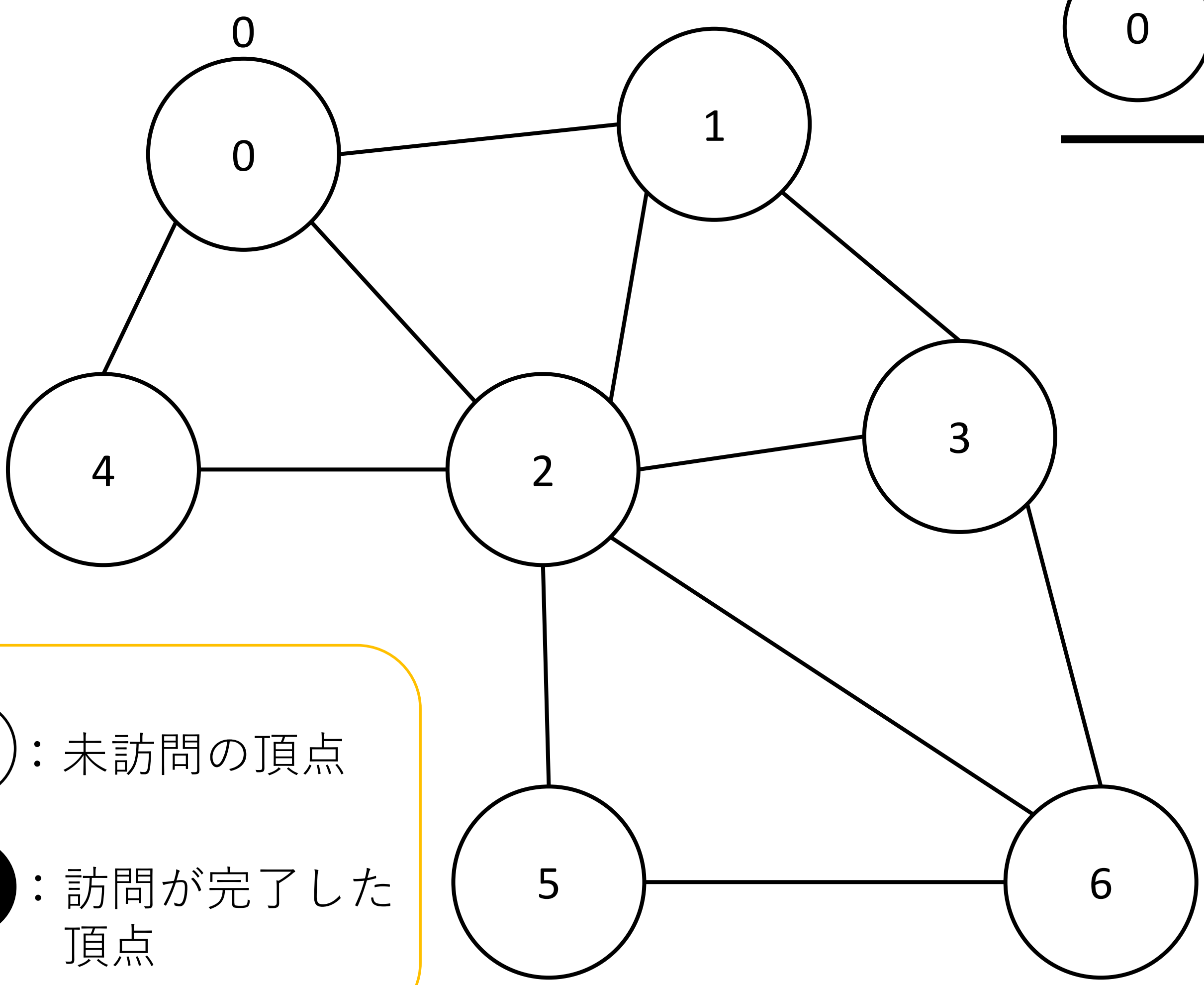
幅優先探索(Breadth First Search)

1. 始点sをキューQに入れる(訪問する)

2. Qが空でない限り、以下の処理を繰り返す

   - Qから頂点uを取り出し訪問する(訪問完了)
   - uに隣接し未訪問の頂点vについて$d[v]$を$d[u]+1$と更新し、vをQに入れる

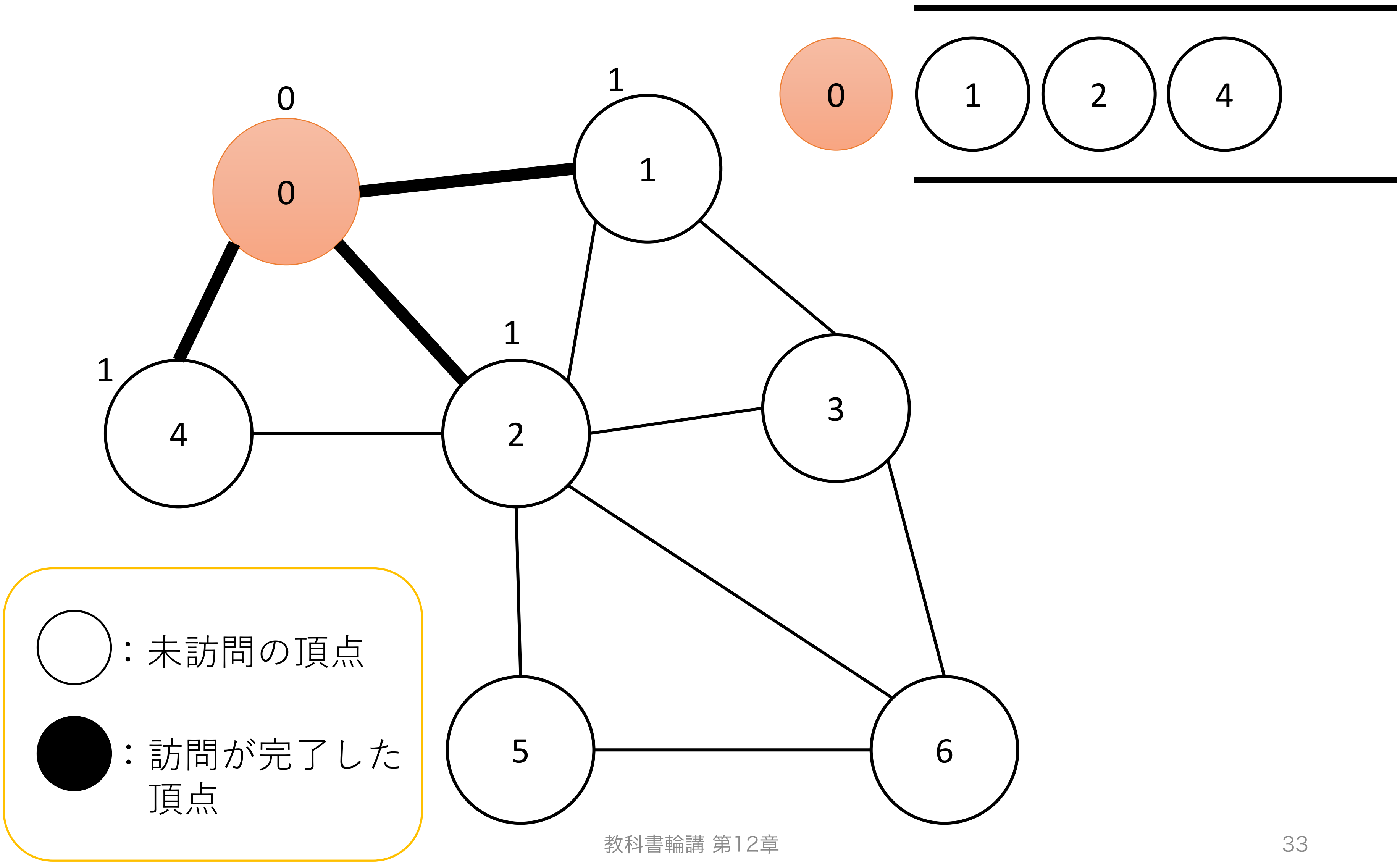
幅優先探索の例



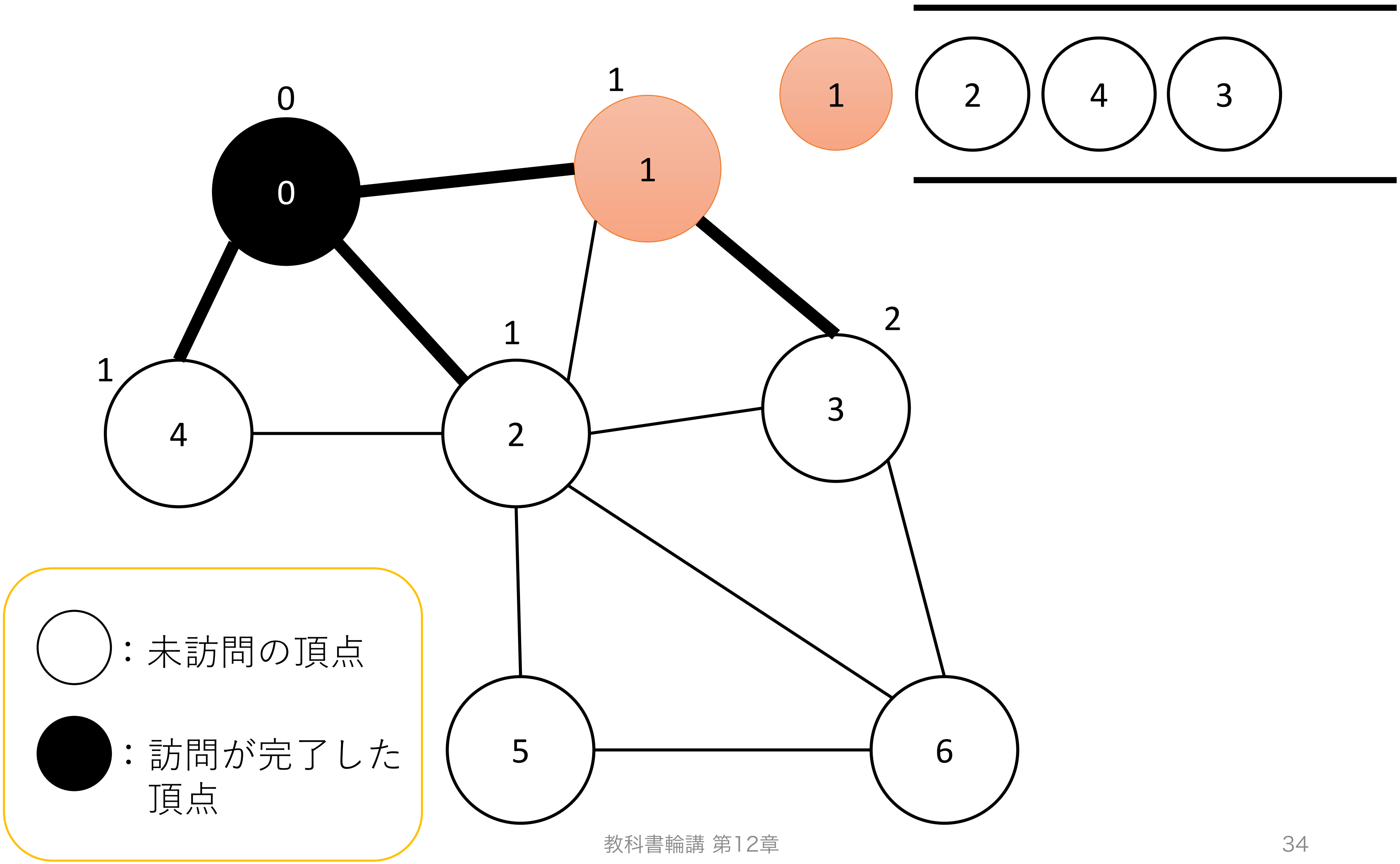
：未訪問の頂点

：訪問が完了した頂点

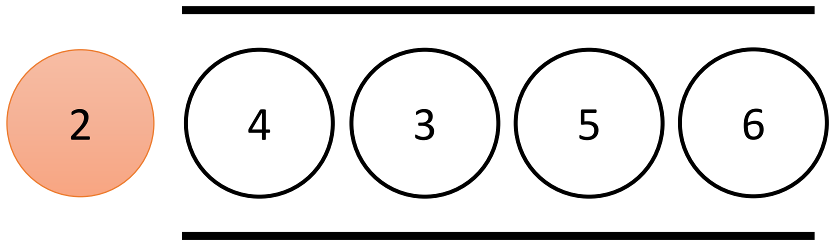
幅優先探索の例

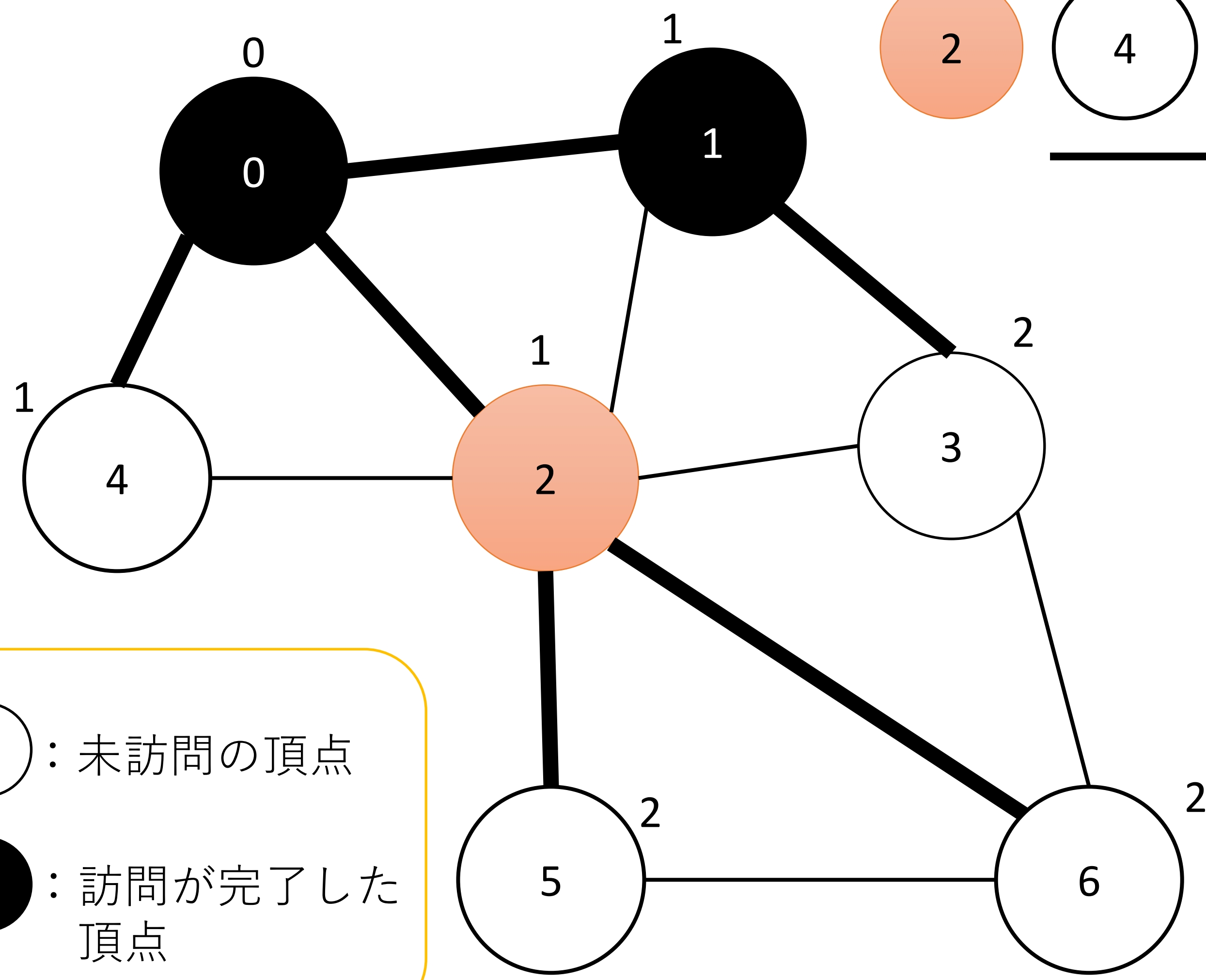


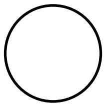
幅優先探索の例

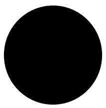


幅優先探索の例

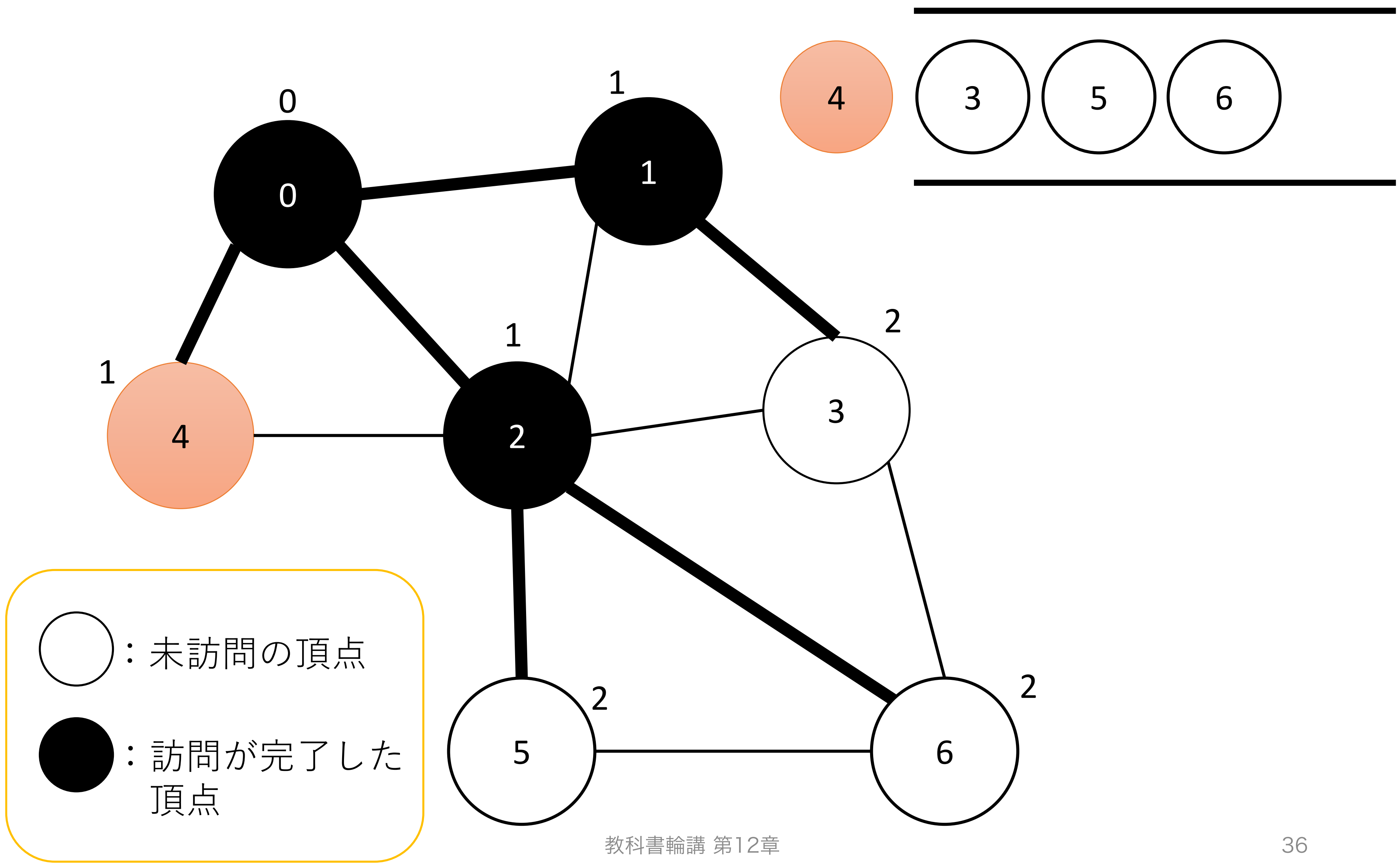




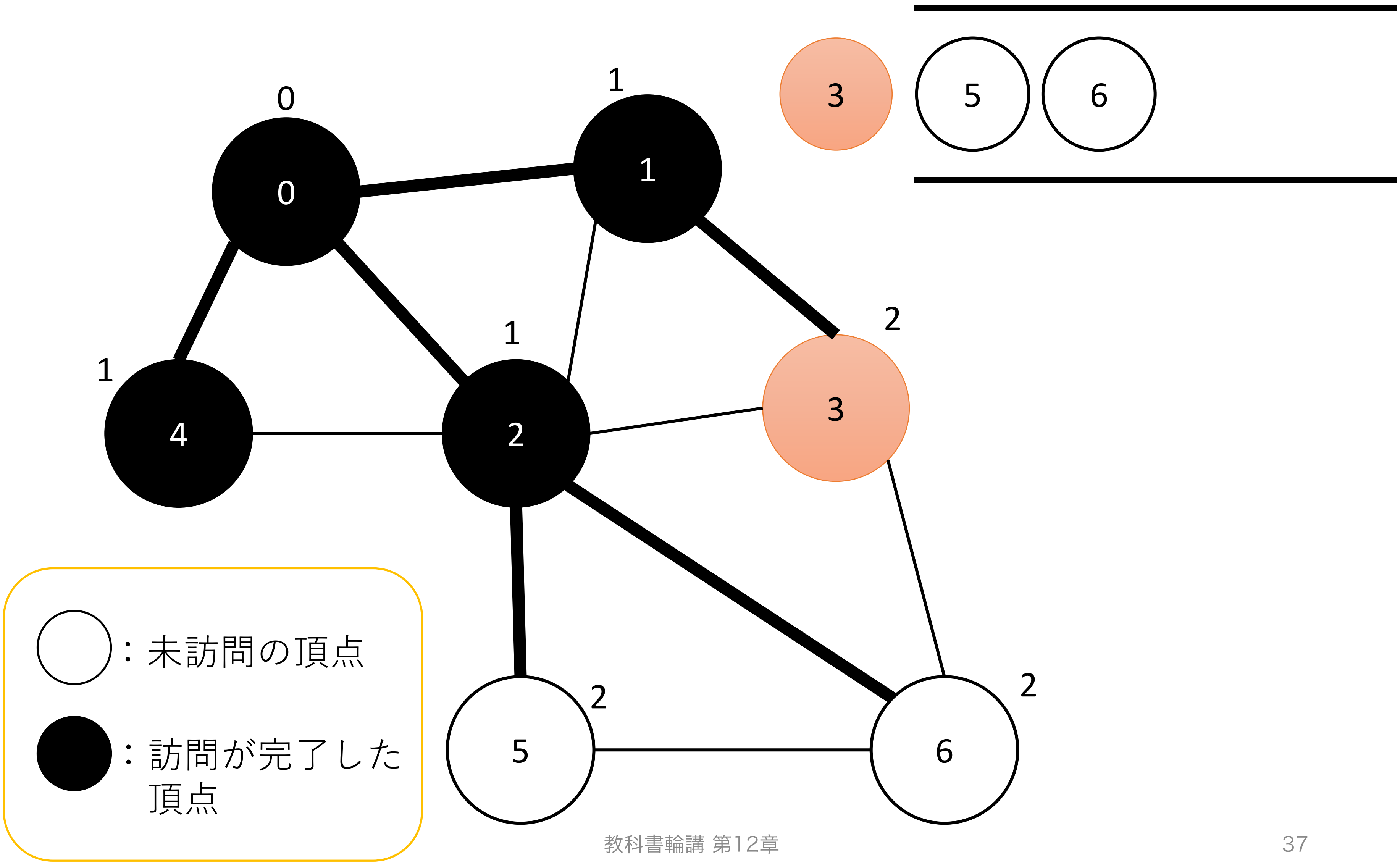

：未訪問の頂点


：訪問が完了した頂点

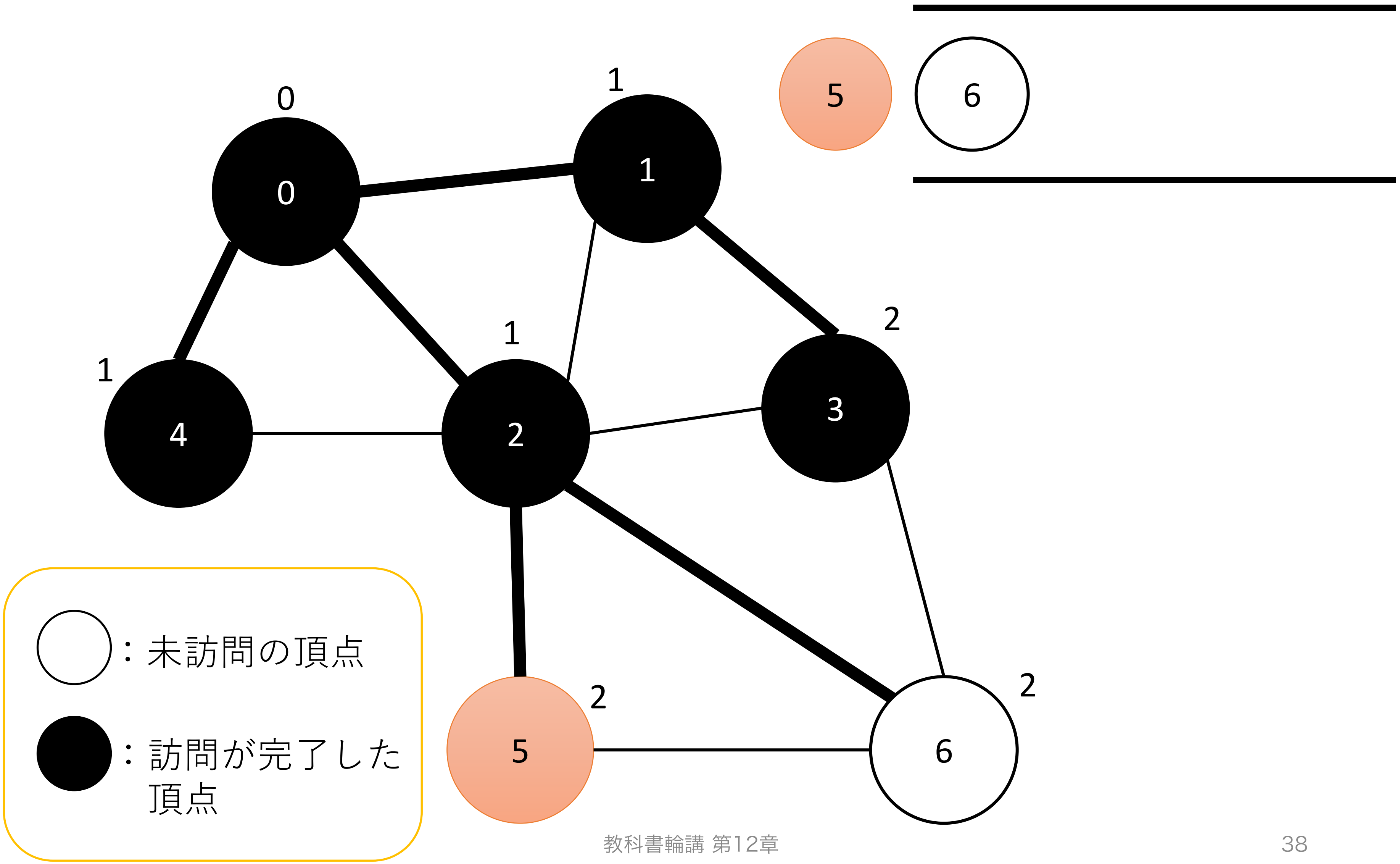
幅優先探索の例



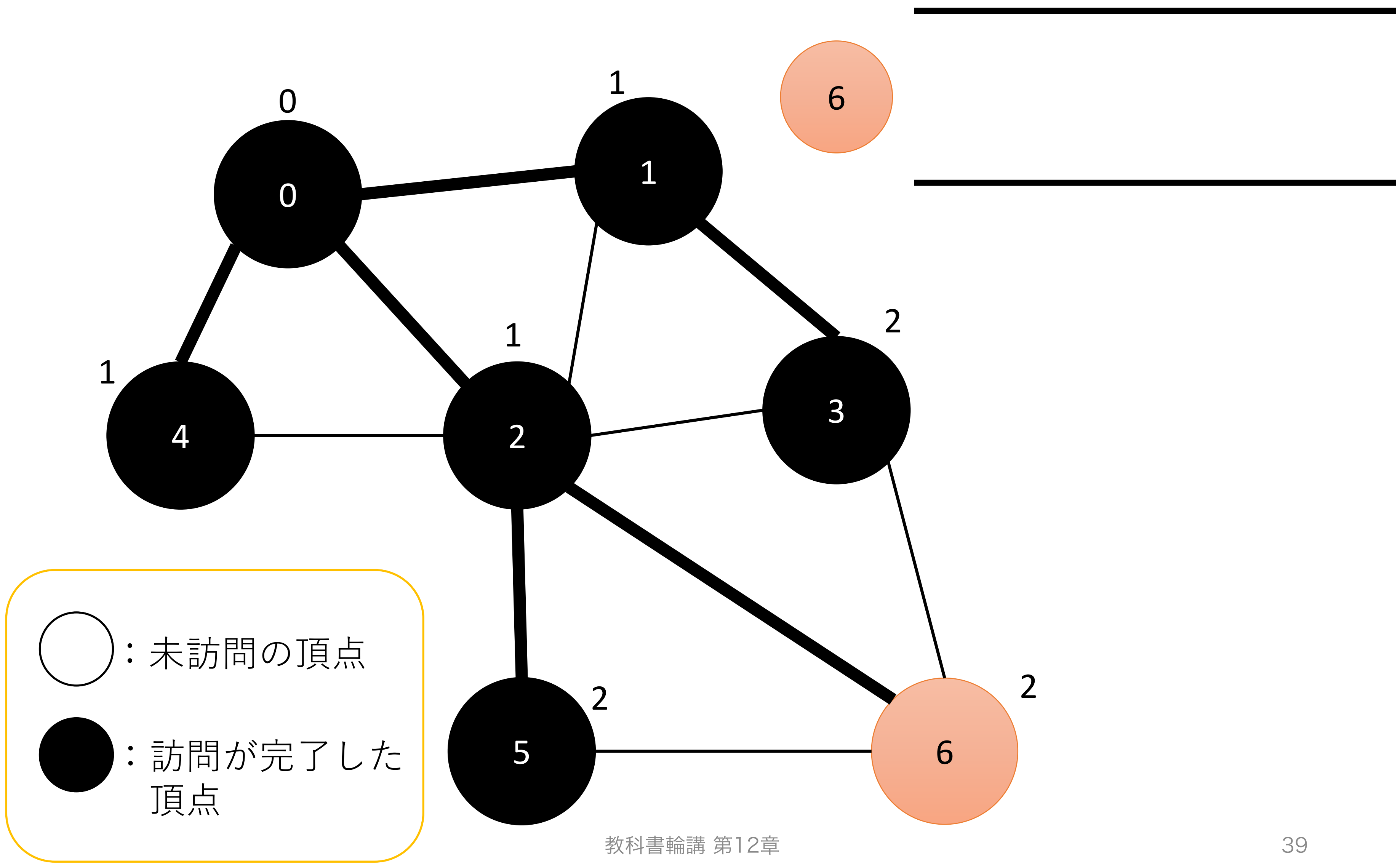
幅優先探索の例



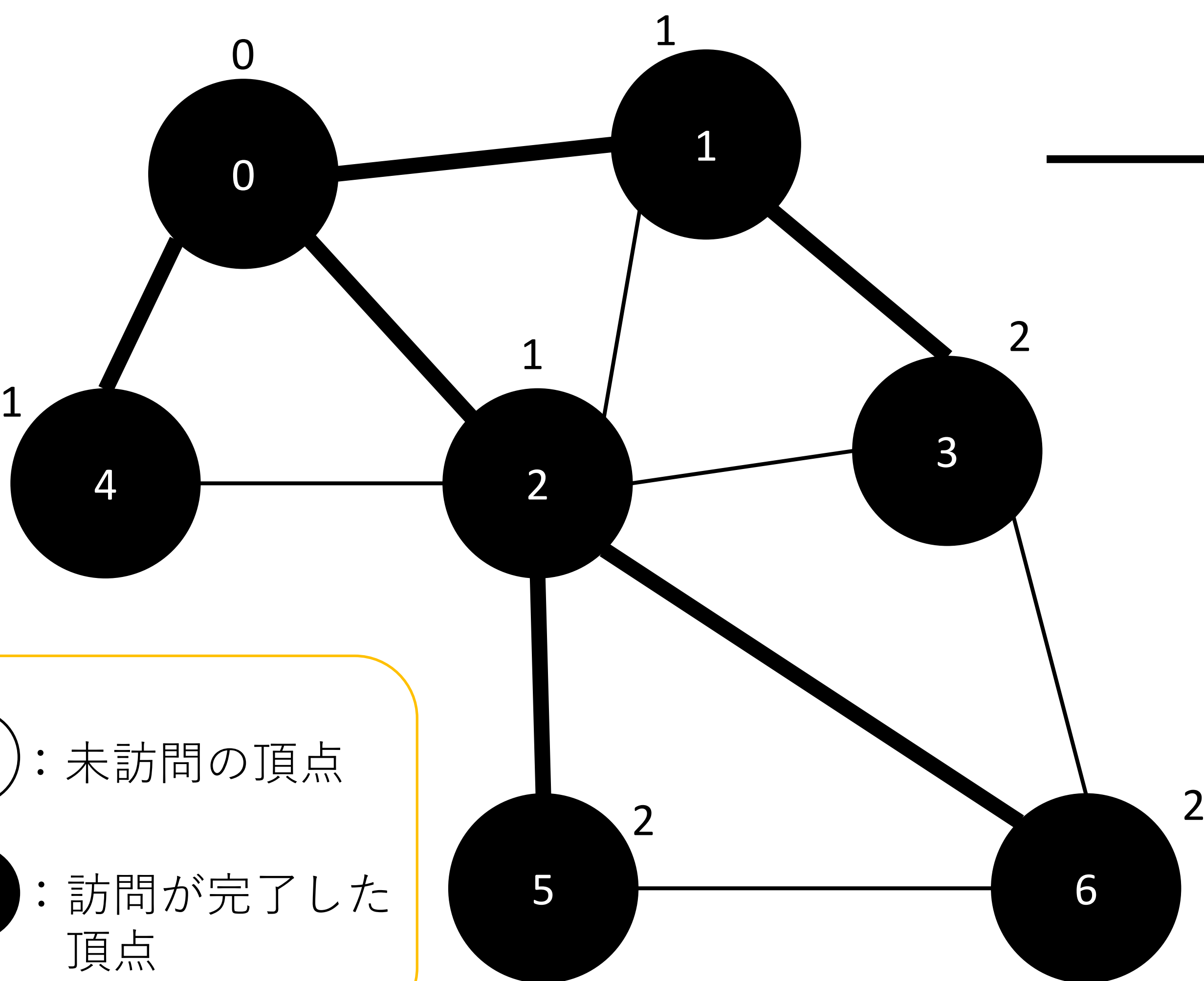
幅優先探索の例



幅優先探索の例



幅優先探索の例



：未訪問の頂点

：訪問が完了した頂点

演習問題(3) | 幅優先探索

入力:

最初の行にGの頂点数nが与えられる
続くn行で各頂点uの隣接リスト$Adj[u]$が以下の形式で与えられる

$u\ k\ v_1\ v_2\ \dots v_k$
（uは頂点の番号、kはuの次数、$v_1\ v_2\ \dots v_k$はuに隣接する頂点の番号）

出力:

各頂点についてid, dを空白区切で1行に出力せよ
idは頂点の番号、dはその頂点の発見時刻を表す
頂点の番号順で出力せよ

制約:

$1 \leq n \leq 100$

演習問題(3) | 幅優先探索

入力例:

```
4
1 2 2 4
2 1 4
3 0
4 1 3
```

出力例:

```
1 0
2 1
3 2
4 1
```

解説｜幅優先探索

キューを用いた実装

```
def BFS(adj, start):
    n = len(adj)
    d = [-1] * n
    flag = [0] * n
    Q = []

    Q.append(start)
    d[start] = 0
    flag[start] = 1

    while len(Q) != 0:
        u = Q.pop(0)
        v = [i for i, x in enumerate(adj[u]) if (x == 1) and (flag[i] == 0)]
        # v := 隣接ノードのうち未探索のノード番号

        for i in v:
            Q.append(i)
            d[i] = d[u] + 1
            flag[i] = 1

    return d
```

練習問題(時間余ったら) | Connected Components

SNSの友達関係を入力し、双方向の友達リンクを経由してある人からある人へたどりつけるかを判定するプログラムを作成せよ

入力:

1行目にSNSユーザー数を表す整数nと友達関係の数mが空白区切りで与えられる.
SNSの各ユーザには0から$n-1$までのIDが割り当てられている.

続くm行に1つの友達関係が各行に与えられる.1つの友達関係は空白で区切られた2つの整数s、tで与えられ、sとtが友達であることを示す.

続く1行に、質問の数qが与えられる. 続くq行に質問が与えられる.
各質問は空白で区切られた2つの整数s、tで与えられ、「sからtへたどり着けるか？」という質問を意味する.

出力:

各質問に対してsからtにたどり着ける場合はyesと、たどり着けない場合はnoと1行に出力せよ.

練習問題(時間余ったら) | Connected Components

制約:

$$0 \leq m \leq 100{,}000$$
$$1 \leq q \leq 10{,}000$$

入力例:

```
10 9
0 1
0 2
3 4
5 7
5 6
6 7
6 8
7 8
8 9
3
0 1
5 9
1 3
```

出力例:

```
yes
yes
no
```