

第7章

グラフアルゴリズム

本章では、グラフ (graph) と呼ばれる抽象型のデータ構造と、グラフに対するアルゴリズムとして、幅優先探索や深さ優先探索、ダイクストラ法について解説します。

グラフは、情報サービスを提供するアプリケーションで使用されるデータ構造です。たとえば、ソーシャルネットワークやコンピュータネットワークなどを抽象化してグラフとして表現します。また、一般道路や鉄道会社、航空会社などの交通サービスでは、交通網や路線、空路などを抽象化してグラフ構造にします。さらに入力されたキーワードの意味を解析するために、知識グラフと呼ばれるグラフ構造が Google の検索エンジンで用いられています。Google の例は、本書の執筆時点の話です。

7.1 グラフの用途

グラフは頂点 (vertex) と辺 (edge) で構成されます。各頂点と辺がデータ構造内の要素を表します。視覚化すると頂点は円形で辺は線です。木構造も円形と線で表すので、木構造はグラフの特殊な例と言えます。

それではソーシャルネットワークをグラフ構造で表現してみます。Alice と Bob、Chris、David、Elizabeth という 5 人の学生がいたとします。グラフ構造ではこれらの学生を抽象化し、頂点として表します。そしてソーシャルネットワークにおける関係をグラフの辺として表します。関係に関しては、さまざまな意味で定義できますが、ここでは友達関係を辺として表します。図 7.1 に例を示します。

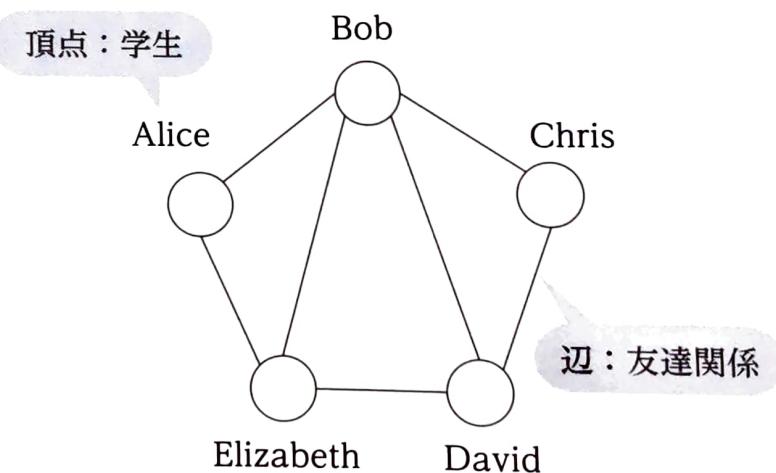


図 7.1 ソーシャルネットワークをグラフ構造で表した例

辺で接続されている Alice と Bob、Alice と Elizabeth、Bob と Chris、Bob と David、Bob と Elizabeth、Chris と David、David と Elizabeth がそれぞれ友達同士です。

同様にコンピュータネットワークもグラフ構造に抽象化することができます。インターネットにアクセスするパソコンやスマートフォン、アクセスポイントやルータと呼ばれる中継機がグラフ内の頂点となります。これらの通信機器が有線ケーブル、または無線で直接接続されれば、頂点を辺で結びます。

また、路線や空路のグラフ化はわかりやすい例です。路線であれば、駅を頂点で表し、隣接する駅を辺で接続します。空路であれば、空港を頂点で表し、直行便が飛んでいる空港同士を辺で接続します。一方、道路の場合は少し違う方法でグラフ構造にします。道路の交差点を頂点とし、道路で繋がっている隣接する交差点を辺で結びます。たとえば、図 7.2 に示す道路を抽象化してグラフ構造にすると、図 7.3 のようなグラフになります。

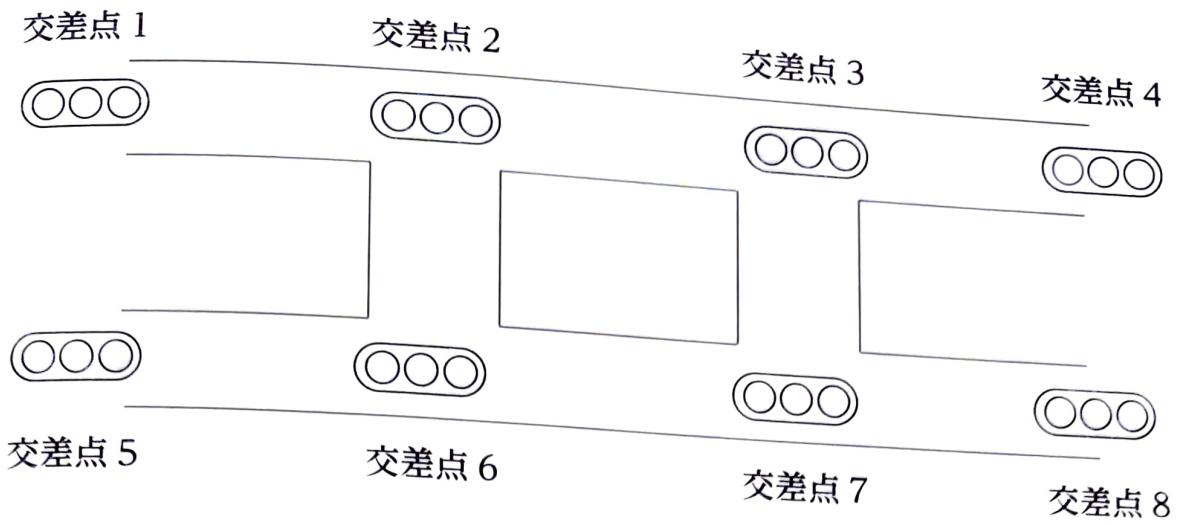


図 7.2 道路の例

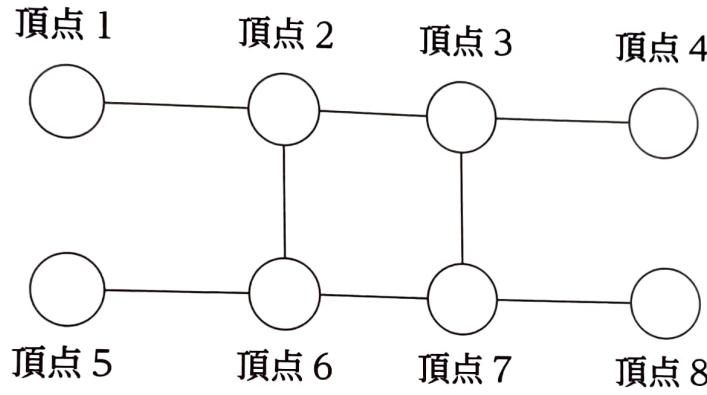


図 7.3 図 7.2 に示す道路を抽象化し、グラフ構造で表した例

グラフアルゴリズムでは、与えられたグラフ内で探索などを行います。幅優先探索や深さ優先探索は、ある頂点を始点としてグラフ内の到達可能なすべての頂点を探査します。その結果、ある頂点から他の頂点への経路などを明らかにできます。ソーシャルネットワークであれば、Alice が Chris と友達を介して繋がっているかどうか調べることができます。図 7.1 では、Alice は Bob を介して Chris と繋がっています。Alice を表す頂点を始点として、探索アルゴリズムを適応することによって、これらの関係が明らかになります。

また、グラフ上のある頂点から他の頂点への最短経路を求めるためのアルゴリズムとして、7.7 節で解説するダイクストラ法があります。路線グラフにおいて、ある駅から他の駅への最短経路を探査するときなどに用いられます。最短経路の「最短」という意味ですが、これもさまざまな意味として定義できます。たとえば、所要時間が一番短い経路や運賃が一番安い経路などです。辺の重みを定量化して、最短経路の意味を定義することができます。

このようにグラフ構造は、さまざまな情報サービスを実現するために利用されるデータ構造です。

7.2

グラフ構造の基本

グラフ構造は、頂点へ向かう方向もつか、もたないか（方向性の有無）によって、無向グラフと有向グラフの2種類に分けられます。また、グラフ構造をプログラミング上で実現する方法として、マトリクスを用いた方法とクラスを用いた方法の2種類があります。本節と第7.3節、第7.4節では、これらの違いを確認しながら、グラフを用いたプログラムの説明をします。

7.2.1 無向グラフと有向グラフ

グラフには、無向グラフ（undirected graph）と有向グラフ（directed graph）があります。無向グラフは前節の例で解説したような、辺に方向性がないグラフのことです。

一方、有向グラフは辺に方向性があるグラフのことです。たとえば、道路をグラフ構造で表したいとします。交通量が多い都市部や十分な幅員が確保できな場所などでは、片側通行の道路を見かけると思います。このような場合は有向グラフで道路網を抽象化することができます。

たとえば、前節で説明した道路の図7.2を見てください。交差点2と交差点6を結ぶ道路は一方通行と仮定し、交差点6から交差点2にしか進めないとします。同様に交差点3から交差点7の道も一方通行とします。その他の道路は往復方向に進むことができる道路とします。この道路を有効グラフで表すと図7.4のようになります。片側通行の道路は、辺で結ばれた頂点の片方からもう片方へしか進めないことが表現できていることが確認できます。

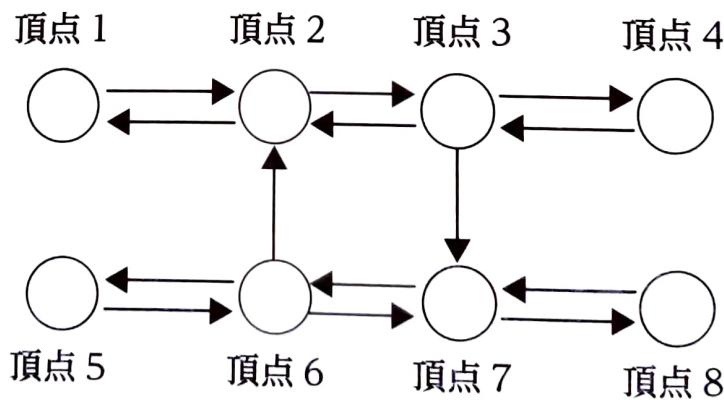


図7.4 有向グラフの例

実は、無向グラフは有向グラフの特殊な例です。無向グラフ内で頂点*i*から頂点*j*の辺があれば、方向性をもつ頂点*i*から頂点*j*への辺と頂点*j*から頂点*i*への辺存在すると見なせば、すべての無向グラフは有向グラフで表すことができるからです。

7.3

隣接行列を用いたグラフの表現

プログラム内でグラフ構造を扱うには、頂点を表す変数と辺を表す変数を定義する必要があります。ここでは、まず頂点と辺を行列に見立てた**隣接行列 (adjacent matrix)**を用いた方法を説明します。また、単純にマトリクスと呼ぶ場合もあります。一般に、グラフ理論などでは隣接行列を用います。

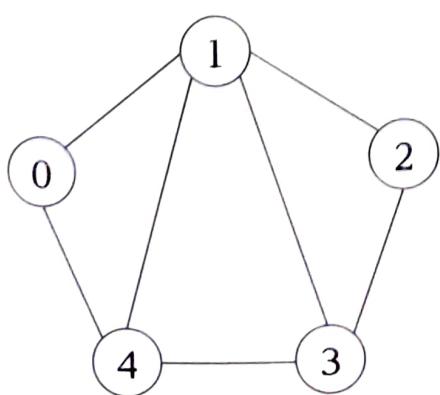
複数の行と列から構成される行列の各要素は行と列のインデックスを指定して識別します。たとえば、 i 番目の行、 j 番目の列に位置する要素は、 (i, j) と記述します。この点では2次元配列と似ています。

グラフ内の頂点の数を n とした場合、グラフの構造を表す隣接行列は、 n 行と n 列の行列になります。各行と各列のインデックスをグラフ内の各頂点に割り当てます。簡略化のために、グラフ内の頂点の識別子は 0 から始まる整数値とし、頂点 i がインデックス i に対応するものとします。行列の (i, j) の値は 0 または 1 のいずれかです。もし、頂点 i と頂点 j が辺で結ばれていたら、 (i, j) の値が 1 となります。そうでなければ 0 となります。

7.3.1 隣接行列を用いた無向グラフ

たとえば、図 7.5 に、識別子 0 ~ 4 をもつ 5 つの頂点が辺で接続されているグラフを示します。このグラフを行列で表すと同図の右側の行列になります。頂点 0 は頂点 1 と頂点 4 と繋がっていますので、行列の $(0, 1)$ と $(0, 4)$ の値が 1 に設定されます。その他の頂点とは接続されていないので、 $(0, 2)$ と $(0, 3)$ は 0 となっています。また、 $(0, 0)$ は 0 です。本書の例では、ある頂点から出て同じ頂点に戻ってくる辺は存在しないので、 (i, i) は 0 です。

他の頂点に関しても同じルールで (i, j) の値が設定されています。また、無向グラフの場合は (i, j) が 1 であれば、 (j, i) も 1 です。辺に方向性が無いので、頂点 i から頂点 j への辺が存在すれば、頂点 j から頂点 i への辺も存在します。



頂点間の関係を
行列で表現

		列				
		0	1	2	3	4
行	0	0	1	0	0	1
	1	1	0	1	1	1
2	0	1	0	1	0	
3	0	1	1	0	1	
4	1	1	0	1	0	

図 7.5 頂点間の関係を行列で表現（有向グラフの場合は列から行、行から列への方向がある。辺があるときに 1 が入る）

第7章 グラフアルゴリズム

辺の追加と削除は簡単です。新たに辺を追加したい2つの頂点を i と j とします。隣接行列の (i, j) と (j, i) の値を 1 にすれば良いだけです。逆に辺を削除したい場合は (i, j) と (j, i) の値を 0 にするだけです。

頂点の追加と削除は要注意です。新たな頂点を追加、または既存の頂点を削除すると、隣接行列の大きさが変わります。そのため、再度、隣接行列を作り直す必要があります。そのため、隣接行列を用いたグラフの表現は、動的なグラフ構造に不向きです。

■ 隣接行列を用いた無向グラフの実装例

隣接行列を用いた方法では、2次元の配列でグラフ構造を定義することができます。ソースコード 7.1 (undir_graph.py) に隣接行列を用いた無向グラフの例を示します。無向グラフを生成して、隣接行列を表示するプログラムです。

ソースコード 7.1 マトリクスを用いた無向グラフの実装

`~/ohm/ch7/undir_graph.py`

ソースコードの概要

2 行目～4 行目	2つの頂点を接続するための <code>connect</code> 関数の定義
7 行目～17 行目	隣接行列を表示する <code>pretty_print</code> 関数の定義
19 行目～36 行目	<code>main</code> 関数の定義

■ main 関数の説明

19行目～36行目で main 関数を定義しています。21行目でグラフ内の頂点の数を定義するため、変数 N を宣言して整数値 5 で初期化します。24行目で隣接行列を表す 2 次元配列 graph を宣言し、すべての要素を整数値 0 で初期化します。配列の大きさは $N \times N$ です。

初期化状態では、どの頂点も接続されていない状態なので、27行目～33行目の命令で connect 関数を呼び出しながら頂点を接続します。グラフの状態は図 7.5 と同じです。5つの頂点と 7 つの辺で構成されます。

36行目で pretty_print 関数を呼び出して、隣接行列である変数 graph の中身を表示します。

■ connect 関数 (2 つの頂点の接続) の説明

2行目～4行目で 2 つの頂点を接続するための connect 関数を定義しています。引数として、グラフを表す 2 次元配列を変数 graph、2 つのインデックスを変数 i と j で受け取ります。頂点 i と j を接続する処理なので、3行目と 4 行目でそれぞれ $graph[i][j] = 1$ 、 $graph[j][i] = 1$ という命令を実行しています。

■ pretty_print 関数 (隣接行列の中身の表示) の説明

データ構造とは関係ありませんが、7行目～17行目で pretty_print 関数を定義しています。関数内では、隣接行列の中身をインデックス付きで表示します。単純に数字を文字列として並べるだけではなく、行列の要素がわかりやすく表示されるようにしています。このようにきれい (pretty) に表

示 (print) する処理を一般的に pretty print と呼びます。

図 7.6 に示すように、筆者の環境では行と列の各要素がきれいに表示されることを確認しましたが、環境によっては若干異なるかもしれません。

```
macbook:ch7 sakai$ python3 undir_graph.py
    0  1  2  3  4
0  0  1  0  0  1
1  1  0  1  1  1
2  0  1  0  1  0
3  0  1  1  0  1
4  1  1  0  1  0
```

図 7.6 pretty_print 関数の実行結果

■ 隣接行列を用いた無向グラフの例題プログラムの実行

ソースコード 7.1 (undir_graph.py) を実行した結果をログ 7.2 に示します。図 7.5 で示した行列と同じであることが確認できます。なお、2 行目の整数値 0 ~ 4 は隣接行列のインデックスで、3 行目 ~ 7 行目の 1 つ目の数値である 0 ~ 4 も隣接行列のインデックスを表します。

ログ 7.2 undir_graph.py プログラムの実行

```
01 $ python3 undir_graph.py
02     0  1  2  3  4
03 0  0  1  0  0  1
04 1  1  0  1  1  1
05 2  0  1  0  1  0
06 3  0  1  1  0  1
07 4  1  1  0  1  0
```

■ 7.3.2 隣接行列を用いた有向グラフ

有向グラフの場合も無向グラフと同じように隣接行列を定義します。ただし無向グラフと異なり、辺に方向性があるので (i, j) の値が 1 であっても (j, i) の値が 1 とは限りません。有向グラフでは、頂点 i から頂点 j に辺が存在する場合にだけ (i, j) の値が 1 となります。

たとえば、図 7.7 に、識別子 0 ~ 4 をもつ 5 つの頂点が方向性をもつ辺で接続されているグラフを示します。このグラフを行列で表すと同図の右側の行列になります。行のインデックスは辺の始点を指し、列のインデックスは辺の終点を指します。頂点 0 から頂点 1 と頂点 4 に繋がっています

ので、行列の(0, 1)と(0, 4)の値が1に設定されます。ただし、頂点1から頂点0、頂点4から頂点0への辺は存在しないので、(1, 0)と(4, 0)の値は0です。他の頂点に関しても同じルールで(i, j)の値が設定されています。

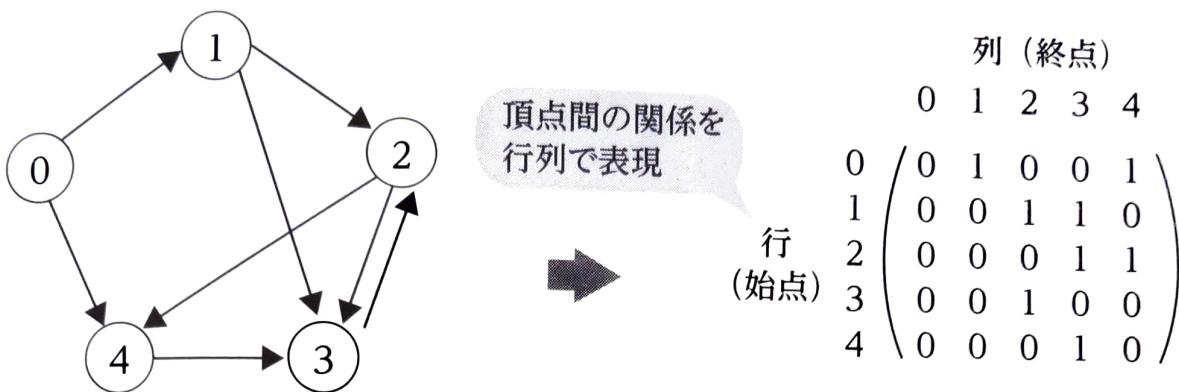


図 7.7 隣接行列を用いた有向グラフの例

有向グラフにおける辺の追加と削除は、方向性があるため、無向グラフの操作とは少し異なります。たとえば、頂点*i*から頂点*j*へ向かう辺を新たに追加したい場合は、隣接行列の(i, j)の値を1にします。方向性があるので、(j, i)の値は変更しません。また、頂点*i*から頂点*j*にある既存の辺を削除する場合は、隣接行列の(i, j)の値を0にします。同様に(i, j)の値は変更しません。

頂点の追加と削除は、無向グラフと同様に隣接行列の大きさが変わるので、隣接行列自体を再計算する必要があります。

■ 隣接行列を用いた有効グラフの実装例

ソースコード 7.3 (dir_graph.py) に隣接行列を用いた有効グラフの例を示します。有効グラフを生成して、隣接行列を表示するプログラムです。

ソースコード 7.3 マトリクスを用いた有向グラフの実装

~/ohm/ch7/dir_graph.py

ソースコードの概要

2行目～3行目	2つの頂点を接続するための connect 関数の定義
6行目～16行目	隣接行列を表示する pretty_print 関数の定義
18行目～36行目	main 関数の定義

2行目～3行目の connect 関数は無向グラフのソースコード 7.1 (undir_graph.py) とは少し異なります。方向性があるので、頂点*i*から頂点*j*への辺を加えますが、その反対方向へは辺を加えません。また、6行目～16行目の pretty_print 関数はソースコード 7.1 (undir_graph.py) と同じです。

```

05 # グラフ情報の表示
06 def pretty_print(graph, n):
07     print(" ", end = "")
08     for i in range(0, n):
09         print(i, " ", end = "")
10     print("")
11
12     for i in range(0, n):
13         print(i, " ", end = "")
14         for j in range(0, n):
15             print(graph[i][j], " ", end = "")
16         print("")
17
18 if __name__ == "__main__":
19     # 頂点の数
20     N = 5
21
22     # 隣接行列の生成と初期化
23     graph = [[0 for i in range(0, N)] for j in range(0, N)]
24
25     # 辺の設定
26     connect(graph, 0, 1)
27     connect(graph, 0, 4)
28     connect(graph, 1, 2)
29     connect(graph, 1, 3)
30     connect(graph, 2, 4)
31     connect(graph, 2, 3)
32     connect(graph, 3, 2)
33     connect(graph, 4, 3)
34
35     # グラフの情報を表示
36     pretty_print(graph, N)

```

■ main 関数の説明

18行目～36行目でmain関数を定義しています。20行目の隣接行列の大きさ初期化と23行目の2次元配列の初期化は前項と同じです。変数名をそれぞれ、Nとgraphとします。

26行目～33行目で8つの辺をグラフに追加しています。たとえば、26行目のconnect(graph, 0, 1)では、頂点0から頂点1への辺を変数graphに追加します。36行目で隣接行列である変数graphの中身を表示します。**図7.7**と同じ隣接行列が生成されます。

■ 隣接行列を用いた有効グラフの例題プログラムの実行

ソースコード7.3 (dir_graph.py) を実行した結果を**ログ7.4**に示します。2行目～7行目に隣接行列のインデックスと各要素の値が表示されています。**図7.7**と同じであることが確認できます。

ログ7.4 dir_graph.py プログラムの実行

```
01 $ python3 dir_graph.py
02   0  1  2  3  4
03 0  0  1  0  0  1
04 1  0  0  1  1  0
05 2  0  0  0  1  1
06 3  0  0  1  0  0
07 4  0  0  0  1  0
```

隣接行列を用いたグラフでは、頂点の関係を保存するために2次元配列を用いますが、接続されていない頂点同士も0という値を記録する必要があります。頂点数をnとした場合、ある頂点に隣接する頂点を走査するのに $O(n)$ の時間がかかります。そのため、隣接行列を用いた手法は、頂点の数が億を超えるような大規模なグラフ構造の表現には向いていません。そこで頂点を表すクラスを定義し、各頂点のオブジェクトが隣接する頂点の情報だけをもつようにすれば、大規模なグラフを扱うことができます。

7.4

クラスを用いた無向グラフの表現

本節では**クラスを用いた無向グラフ**の表現を解説します。隣接行列と同様に有向グラフの場合もほぼ同じですので、クラスを用いた有向グラフの実装例の説明は割愛します。

クラスを用いたグラフの表現では、頂点を表すクラスとして以下のようないMyVertexを定義します。クラスメンバは、識別子を表すidと隣接頂点(adjacent vertices)の識別子を表すadjです。引数として、頂点の識別子を変数idで受け取り、3行目のself.id = idで識別子を初期化します。隣接頂点の情報は、インスタンス生成時に空で随时追加します。隣接頂点の情報は、インスタンス生成時には空で随时追加します。

```
class MyVertex:
    def __init__(self, id):
        self.id = id
        self.adj = set()
```

クラスメンバ adj がセット (set) で初期化されています。Python には標準ライブラリとして、集合を扱うデータ型である set 型が用意されています。set 型は重複しない要素の集合を含み、和集合や積集合などの集合演算が可能になります。また、連結リストと異なり、セットの要素に順番はありません。このようにデータの集合を扱う機能を一般的にコレクション (collection) と呼びます。

上記のクラスの定義の 4 行目で self.adj = set() と記述すると、空のセットが生成されます。要素をセットに追加するときには、変数名 .add (引数)、という書式で add メソッド実行します。本節の例では、self.adj が含む値は頂点の識別子を表す整数値となります。

set 型に関しては、必要に応じて使用方法を説明しますが、詳しい情報を知りたい読者は、**Python の公式 API ドキュメント¹** を参照してください。

たとえば、頂点 0 と 1 を生成して、それらを辺で接続したい場合は以下のように記述します。ここで変数 v と u は、それぞれ頂点 0 と 1 を表す MyVertex オブジェクトです。

```
v = MyVertex(0)
u = MyVertex(1)
v.adj.add(1)
u.adj.add(0)
```

頂点の追加は、新たな MyVertex オブジェクトを生成するだけです。辺の追加と削除は該当する頂点の self.adj を変更することによって行います。たとえば、上記の頂点 0 と 1 からなるグラフに、頂点 2 を追加し、頂点 0 と 2 を追加したいとします。以下のようないくつもになります。

```
v = MyVertex(0)
u = MyVertex(1)
v.adj.add(1)
u.adj.add(0)
# ここで頂点2を生成し、頂点0と接続する
w = MyVertex(2)
v.adj.add(2)
w.adj.add(0)
```

頂点 2 を表す MyVertex オブジェクトの変数を w として宣言します。この時点で頂点 2 がグラフに存在する状態ですが、他の頂点と接続されていません。頂点 0 を表す MyVertex オブジェクトは変数 v なので、v.adj.add(2) と w.adj.add(0) を実行し、それぞれのオブジェクトの変数 adj を更新します。

¹ Python3.8.1 ドキュメント, <https://docs.python.org/ja/3/>

頂点と辺の削除は、隣接する頂点の self.adj から当該頂点の識別子を削除します。前述の頂点 0 と 1 と 2 からなるグラフから、頂点 2 を削除したいとします。頂点 2 は頂点 0 と辺で接続されています。頂点 2 と 0 の self.adj から当該識別子を削除します。v.adj.remove(2) と w.adj.remove(0) を実行するだけです。頂点 2 を表す変数 w は、グラフのどの頂点からも接続されていない状態になります。

7.4.1 クラスを用いた無向グラフの実装例

ソースコード 7.5 (my_graph.py) にクラスを用いた無向グラフの実装例を示します。隣接行列を用いた無向グラフで使用した図 7.5 と同じグラフを生成します。

ソースコード 7.5 クラスを用いた無向グラフの実装

~/ohm/ch7/my_graph.py

ソースコードの概要

2 行目～9 行目	頂点を表す MyVertex クラスの定義
12 行目～14 行目	2 つの頂点 i と j を辺で接続するための connect 関数の定義
16 行目～37 行目	main 関数の定義

```

01 # 頂点
02 class MyVertex:
03     def __init__(self, id):
04         self.id = id
05         self.adj = set()
06
07     # 頂点の情報を表示
08     def to_string(self):
09         return str(self.id) + ", adj = " + str(self.adj)
10
11 # 2つの頂点を互いに接続
12 def connect(vertices, i, j):
13     vertices[i].adj.add(j)
14     vertices[j].adj.add(i)
15
16 if __name__ == "__main__":
17     # 頂点の数
18     N = 5
19
20     # 頂点の生成
21     vertices = []
22     for i in range(0, N):
23         v = MyVertex(i)
24         vertices.append(v)
25
26     # 接続
27     connect(vertices, 0, 1)
28     connect(vertices, 0, 2)
29     connect(vertices, 1, 2)
30
31     # 削除
32     v = vertices[2]
33     v.adj.remove(2)
34     v.adj.remove(0)
35
36     print(v.to_string())

```

```

24
25    # 辺の設定
26    connect(vertices, 0, 1)
27    connect(vertices, 0, 4)
28    connect(vertices, 1, 2)
29    connect(vertices, 1, 3)
30    connect(vertices, 2, 4)
31    connect(vertices, 2, 3)
32    connect(vertices, 3, 2)
33    connect(vertices, 4, 3)
34
35    # 各頂点の表示
36    for i in range(0, N):
37        print(vertices[i].to_string())

```

■ MyVertex クラスの説明

2行目～9行目で頂点を表す MyVertex クラスを定義しています。コンストラクタに関しては、すでに説明したとおりです。8行目～9行目の `to_string` メソッドは、MyVertex オブジェクトがもつ `id` と `adj` の値を文字列に変換します。頂点の情報を見たいときに使用します。

■ connect 関数 (2つの頂点の接続) の説明

12行目～14行目で2つの頂点を接続するための `connect` 関数を定義しています。引数として、頂点を含む配列を変数 `vertices`、2つのインデックスを変数 `i` と `j` で受け取ります。頂点 `i` と `j` を接続する処理なので、13行目と14行目でそれぞれ `vertices[i].adj.add(j)` と `vertices[j].adj.add(i)` を実行します。`vertices[i]` は `MyVetex` オブジェクトなので、`vertices[i].adj` と記述すると、`MyVertex` クラス内で定義したクラスメンバの `adj` にアクセスできます。そして `set` 型のメソッドである `add` を呼び出し、`vertices[i].adj` に引数で指定した識別子を追加します。

※有向グラフの場合であれば、片方向にだけ辺で接続するので、14行目の命令は必要ありません。

■ main 関数の説明

18行目で頂点の数として変数 `N` を宣言し、整数値 5 で初期化します。21行目～23行目で頂点を生成します。変数 `vertices` を宣言して、空のリスト（配列として用いる）で初期化します。そして `for` ループを用いて、識別子 0～4 をもつ頂点として `MyVertex` のオブジェクトを生成します。

26行目～33行目で辺を追加します。`connect` 関数を呼び出し、引数として頂点を含む配列 `vertices` と2つの頂点の識別子を渡します。36行目と37行目の処理では、`for` ループを用いて `vertices[0]～vertices[4]` の情報を `to_string` メソッドで表示します。

■ クラスを用いた無向グラフの例題プログラムの実行

ソースコード 7.5 を実行した結果をログ 7.6 に示します。2 行目～6 行目に各頂点の識別子である id の値と隣接する頂点の識別子の集合である adj の中身が表示されています。adj の情報をもとに頂点を線で結ぶと、図 7.5 に示すグラフと同じグラフができると思います。

ログ 7.6 my_graph.py プログラムの実行

```
01 $ python3 my_graph.py
02 0, adj = {1, 4}
03 1, adj = {0, 2, 3}
04 2, adj = {1, 3, 4}
05 3, adj = {1, 2, 4}
06 4, adj = {0, 2, 3}
```

これまでにデータ構造に対する基本的な操作として、要素の取得 (get) や挿入 (insert)、削除 (delete) を解説しました。グラフ構造においては、頂点と辺が要素になるので、変数 vertices に新しい MyVertex オブジェクトを追加する処理や MyVertex オブジェクトのクラスメンバ adj に隣接頂点の識別子を追加する処理が挿入にあたります。グラフ構造は抽象型であり、実装方法によって具体的な処理内容は変わってきます。

以降のグラフアルゴリズムの解説では、クラスを用いた方法を使用します。

7.5 幅優先探索

幅優先探索 (BFS:breath first search) は、ある頂点を始点 (source) として、そこから到達可能なすべての頂点への経路 (path) を探索するアルゴリズムです。始点から辺をたどり各頂点を訪れます、その順番が始点から近い順番になります。そのため、“幅優先”と呼びます。

前述の交通網の例であれば、ある地点から他の地点への経路が存在するかどうかなどを効率的に調べることができます。

7.5.1 幅優先探索と深さ優先探索

次節で解説する“深さ優先”という似たような単語がありますので、まず幅優先と深さ優先の違いについて説明します。図 7.8 に示すグラフを見てください。説明のしやすさのために木構造を用いていますが、実は木構造もグラフの一種です。

頂点 10 を始点として、到達可能な頂点をすべて探索するとします。ここで頂点 10 から各頂点へ

の**距離** (distance) を定義します。ここではすべての辺が同じ重みをもつと仮定します。すなわち、頂点 10 から頂点 5 と 14 への距離は 1、頂点 10 から頂点 2 と 12 への距離は 2、頂点 10 から頂点 8 への距離は 3 となります。つまり、各頂点へ移動するときに経由する辺の数が距離となります。

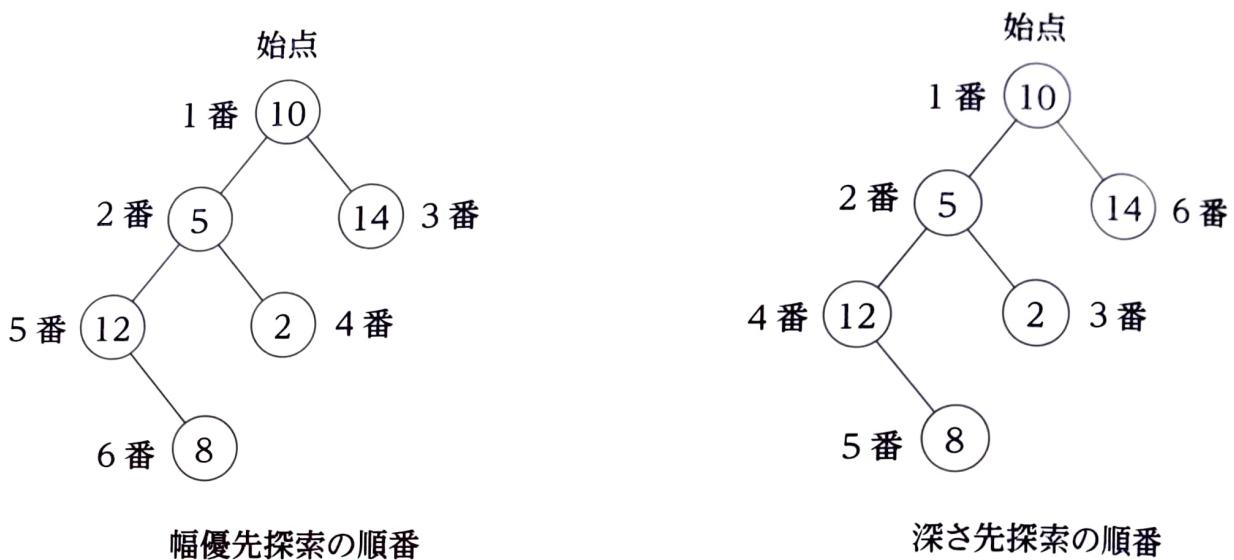


図 7.8 幅優先探索と深さ優先探索の違い

幅優先探索は距離が近い頂点から探索します。図 7.8 の左側のグラフに頂点 10 から各頂点を訪れる順番を示します。また、距離が同じであれば、頂点の識別子が小さい方を優先します。この場合、10、5、14、2、12、8 という順番に探索します。

深さ優先探索の場合は、始点から見て距離が遠い頂点へできるだけ移動し探索をします。これ以上深く移動できなくなれば、すでに訪れた頂点へいったん戻ってきます。同様に、距離が同じであれば、本書の例では識別子が小さい頂点を優先します。図 7.8 の右側のグラフに深さ優先探索の例を示します。頂点 10 から頂点 5 へ移動し、次は頂点 2 に移動します。ここで頂点 2 は頂点 5 以外に隣接する頂点をもたないので、これ以上深く移動できなくなります。いったん頂点 5 へ戻ってきて、頂点 12 と頂点 8 を探索します。行き止まりになると、頂点 10 へ戻ってきて、最後に頂点 14 を探索します。そのため、探索の順番は 10、5、2、12、8、14 となります。

■ 7.5.2 幅優先探索の概要

それでは、幅優先探索をプログラム内で実装する方法の概要を説明します。前節で解説したクラスを用いたグラフの表現と同様に MyVertex クラスを定義します。クラスメンバとして識別子を表す変数 id と隣接頂点の識別子の集合である adj を定義します。

■ キューを用いた頂点の走査

各頂点を幅優先で訪れるためには、第 3.3 節で解説したキューを用います。以下に示す擬似コー

ドを見てください。具体的な処理はもう少し複雑になるので、概要を説明するために擬似的なコードで記述しています。1行目で変数 `q` を宣言して、リスト（キューとして用いる）で初期化します。始点となる頂点を指す `MyVertex` オブジェクトを `src` とします。始点なので、図 7.9 では 10 が入ります。キューは初期化したときに、`src` だけを含んでいる状態になります。

```
q = [src]
while len(q) != 0:
    q からデキューした要素を頂点 u とする
    u の識別子を表示
    u.adj 内で未だ訪れていない頂点の識別子をqにエンキュー
```

2行目～5行目の while ループで、キューの要素をデキューして変数 `u` に格納します。頂点 `u` の識別子を表示し、頂点 `u` の隣接頂点の集合 `u.adj` の中で未だ訪れていない頂点の識別子を変数 `q` にエンキューします。これをキューが空になるまで繰り返します。

図 7.8 で示したグラフで頂点 10 から上記の擬似コードを実行した場合のキューの中身の変化を図 7.9 に示します。while ループ開始前と各イテレーション終了時のキューの状態と訪れた頂点の識別子を示しています。

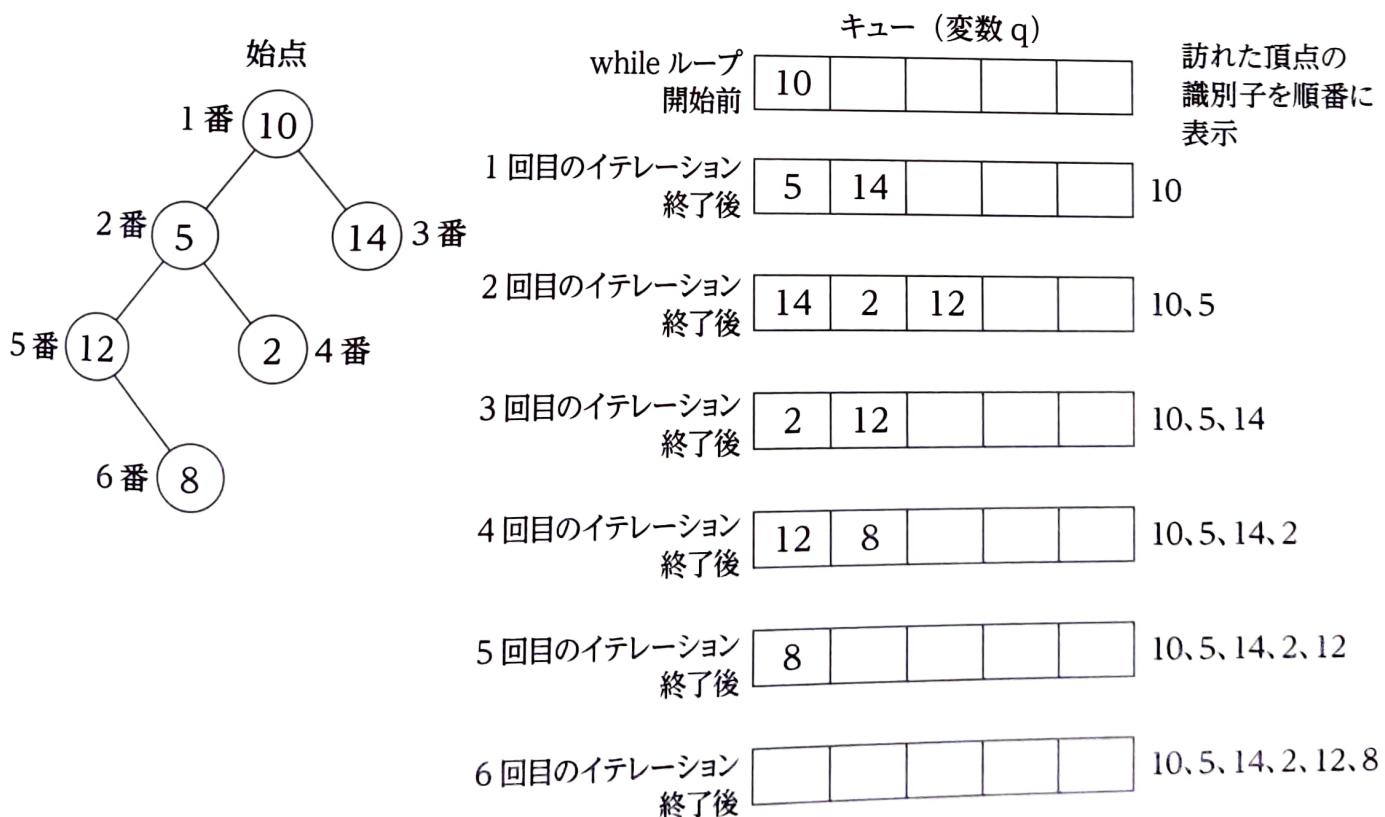


図 7.9 キューを用いた幅優先探索

まず、`q` は 10 だけを含みます。while ループに入り、キューの要素をデキューします。頂点 10 が

変数 u に格納されます。整数値 10 を表示して、10 の隣接端末の識別子をキューにエンキューします。この時点で q の中身は 5 と 14 になります。

次のイテレーションでは、キューの先頭にある頂点 5 がデキューされます。頂点 5 の識別子を表示して、未だ訪れていない頂点 5 の隣接頂点の情報をキューにエンキューします。頂点 10 はすでに訪れています。そのため、頂点 2 と 12 を q にエンキューします。なお、識別子の小さい方からエンキューしています。この時点で q の中身が $[14, 2, 12]$ となります。

次のループでは頂点 14 がデキューされます。先入れ先出しの性質をもつキューを用いているので、頂点 10 の隣接頂点である頂点 14 が先に処理されます。すなわち、頂点 10 から見て距離が 1 の頂点を先に処理してから、距離が 2 である頂点 2 と 12 を処理します。

図 7.9 に示すように、同様の処理を進めていくと、頂点を訪れた順番が 10、5、14、2、12、8 になります。この順番と **図 7.8** の左側の木構造のグラフで探索する順番を比べると、同じであることが確認できると思います。

■ すでに走査した頂点と距離と経路の管理

幅優先探索では、識別子 id と隣接する頂点の識別子の集合 adj に加えて、3 つのクラスメンバを定義します。以下の MyVertex クラスの定義を見てください。5 行目～7 行目で追加したクラスメンバを 1 つひとつ解説していきます。

```
class MyVertex:
    def __init__(self, id):
        self.id = id
        self.adj = set()
        self.color = WHITE
        self.dist = INFTY
        self.pred = None
```

キューからデキューした頂点に隣接する頂点をキューにエンキューするときに、すでに訪れた頂点かどうかを確認する仕組みが必要です。このために、各頂点に 3 つの状態を識別するクラスメンバを用意します。クラスメンバ名を $color$ として宣言し、値は WHITE、GRAY、BLACK のいずれかとします。なお、WHITE と GRAY と BLACK は、以下のように別途グローバル変数をソースコードの冒頭で定義します。

```
WHITE = 0
GRAY = 1
BLACK = 2
```

なぜ $color$ という名前をつけるのかというと、コンピュータサイエンスの分野では、状態を示す識別子に $color$ という名前をつけることが多いからです。実際、図で説明するときに、白色と

灰色と黒色で図形を塗りつぶすことによって、状態の違いを視覚化できます。3つの状態ですが、WHITEは未だ訪れていない状態(一度もキューにエンキューしたことがない状態)、GRAYはキューに入っているが未処理の状態、BLACKはキューからデキューされて隣接する頂点をすべて走査し終えた状態を表します。

最終的に幅優先探索を行なうことは、始点から到達できるすべての頂点の距離と経路を調べることです。そのため、クラスメンバとして始点からの距離を記録する変数 dist (distance を省略して dist と名付ける) を定義します。初期値として 1 を表す変数を設定します。上記の定義では、self.dist = INFTY となっていますが、INFTY という名前のグローバル変数を別途ソースコードの冒頭で定義します。距離の計算は、隣接頂点をキューにエンキューするときに設定します。

一方、経路を調べるためにには、各頂点が始点へ繋がる頂点の識別子を記録する必要があります。これを先行頂点 (predecessor) と呼びます。変数 pred (predecessor を省略して pred と名付ける) をクラスメンバとして定義します。再度、図 7.8 を見てください。木構造になっているので、始点である頂点 10 から探索を始めると、かならず親から子へ頂点をたどります。そのため、各頂点は親となる頂点を pred に保存しておけば、各頂点から始点への経路がわかります。変数 pred の設定は距離と同様に、キューに入れるとときに行います。

7.5.3 幅優先探索の実装例

ソースコード 7.7 (my_bfs.py) に幅優先探索のプログラムを示します。8つの頂点と10個の辺から構成されるグラフを生成し、ある頂点から到達可能なすべての頂点への経路を探索するプログラムです。

ソースコード 7.7 幅優先探索アルゴリズム

[~/ohm/ch7/my_bfs.py](#)

ソースコードの概要

2 行目～4 行目	頂点の探索状態を表すグローバル変数を定義
9 行目～23 行目	グラフ内の頂点を表す MyVertex クラスの定義
26 行目～43 行目	幅優先探索を実行する bfs 関数の定義
46 行目～53 行目	2つの頂点間の経路を表示する print_path 関数の定義
56 行目～58 行目	2つの頂点を辺で接続する connect 関数の定義
60 行目～91 行目	main 関数の定義

63 行目～65 行目で定義している connect 関数は、前節と同じです。

```
01 # 状態の種類を定義
02 WHITE = 0
03 GRAY = 1
```

■ main 関数の説明

60 行目～91 行目で main 関数を定義しています。62 行目で頂点の数を宣言し、65 行目～67 行目で頂点 0～7 を表す MyVertex クラスのインスタンスを生成します。70 行目～79 行目でグラフに辺を追加します。生成したグラフは図 7.10 のようになります。

82 行目の bfs(vertices, vertices[3]) という命令で、頂点 3 から幅優先探索を実行します。引数として、グラフ内の頂点の集合である vertices と、始点となる頂点の MyVertex オブジェクトである vertices[3] を指定します。bfs 関数の処理が終了すると、頂点 3 から到達可能なすべての頂点の dist と pred が設定されます。

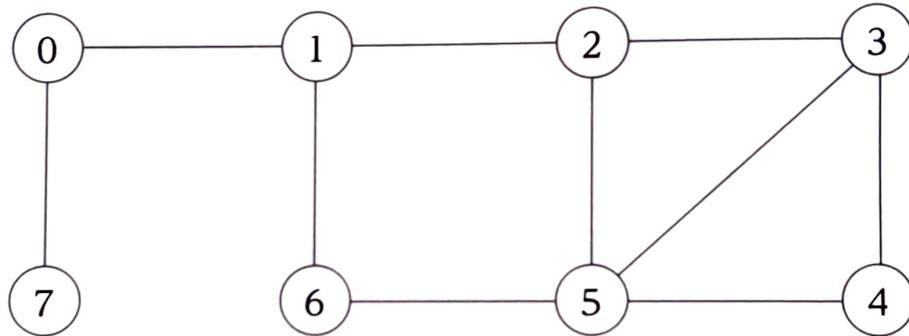


図 7.10 ソースコード 7.7 で生成するグラフ

85 行目～87 行目では、print_path 関数を用いて頂点 3 から頂点 7 の経路を表示します。引数として、vertices と 2 つの頂点の MyVertex オブジェクトである vertices[3] と vertices[7] を渡します。今回は頂点 3 から幅優先探索を行ったので、第 2 引数は vertices[3] である必要があります。第 3 引数は頂点 0～頂点 2、頂点 4～頂点 7 のどれでも構いません。本書の例では vertices[7] を引数として指定します。

90 行目と 91 行目では、グラフ内の各頂点の識別子 id と隣接頂点 adj、探索状態 color、距離 dist、先行頂点 pred の情報を表示します。これは bfs 関数の実行後、各頂点に正しい値が設定されているかを確認するためのものです。

■ bfs 関数（幅優先探索の実行）の説明

26 行目～43 行目で幅優先探索を実行する bfs 関数を定義しています。引数として、頂点の集合を変数 vertices、始点となる頂点の MyVertex オブジェクトを変数 src で受け取ります。28 行目～

30行目で頂点 src の情報を更新します。頂点 src 自身がキューにエンキューされるので、src.color の値を GLAY にします。距離は当然 0 なので、src.dist = 0 という命令を実行します。そして頂点 src は探索の始点なので src.pred は None になります。

33行目～43行目ですが、前項の概要で解説したキューを用いた幅優先探索の骨格を具体化したもので、33行目で変数 q を宣言して、要素として src だけを含むリストを生成します。このリストをキューとして用います。34行目の while ループをキューが空になるまで繰り返します。

35行目の u = q.pop(0) では、変数 u にキューである変数 q の先頭要素をデキューします。なお、Python の標準ライブラリで提供している pop メソッドは、インデックスを指定して要素を取り出します。取り出した要素はキューから削除されます。なお、プログラミング言語によってメソッドの名前と動作が微妙に異なるので、気をつけてください。ここでは先頭の要素を取り出すので、インデックスに 0 を指定します。

36行目～42行目の for ループで、キューから取り出した頂点 u の隣接頂点である u.adj を 1つひとつ調べていきます。ループカウンタの変数 i が u.adj に含まれる頂点の識別子です。37行目で v = vertices[i] を実行して、頂点 u に隣接している頂点 v の MyVertex オブジェクトを取り出します。37行目はなくても構いませんが、ループ内の記述を簡略化するために実行します。また、set 型の場合、u.adj に含まれる要素が昇順に走査されて、ループカウンタ i に値が入ります。

キューに入れる頂点の識別子は、未だ訪れていない頂点だけです。そのため、38行目の if 文で v.color が WHITE かどうかを判定します。WHITE であれば、if ブロックの中に入り、頂点 v をキューにエンキューする処理をします。39行目で、頂点 v の color を GLAY にします。頂点 v が 2 回以上キューにエンキューされるのを防ぐためです。40行目で頂点 src から頂点 v までの距離を設定します。辺を 1 つ経由する毎に距離が 1 増えるので、u.dist+1 が v.dist の値となります。41行目で、先行頂点となる v.pred の値を u.id に設定します。そして、42行目で v を変数 q にエンキューします。

for ループを抜けると、頂点 u に隣接するすべての頂点をキューにエンキューした状態になります。すなわち、頂点 u を探索し終えたことを意味します。そのため、43行目の u.color = BLACK という命令で、頂点 u の探索状態を更新します。

キューが空になると、34行目の while ループの判定式が False になり、ループを抜けます。ループを抜けた時点で、頂点 src から到達することができるすべての頂点の color の値が BLACK になります。また、dist は 1 以上かつ ∞ 以下の整数値、pred は頂点の識別子になります。

本書での例は、すべての頂点が間接的に辺で接続されていますが、もしグラフが 2 つの部分グラフに分割されている場合は、始点から到達できない頂点の dist は ∞ のままでです。到達できないから距離が無限大なのです。また、pred も初期値のままになるので値は None です。

具体例を示すために、ソースコード内で生成したグラフの状態がどのように変化するかを図解します。各頂点の変数 color と dist と pred の値、探索時に用いるキュー（33行目で宣言した変数 q）の中身を確認しながらトレースしてみてください。

まず、図 7.11 に 34 行目の while ループに入る前の状態を示します。キューには頂点 3 が含まれており、vertices[3].color が GLAY、vertices[3].dist が 0、vertices[3].pred は None です。color に関しては、頂点を表す円形を白色、灰色、黒色に色分けして状態を示します。

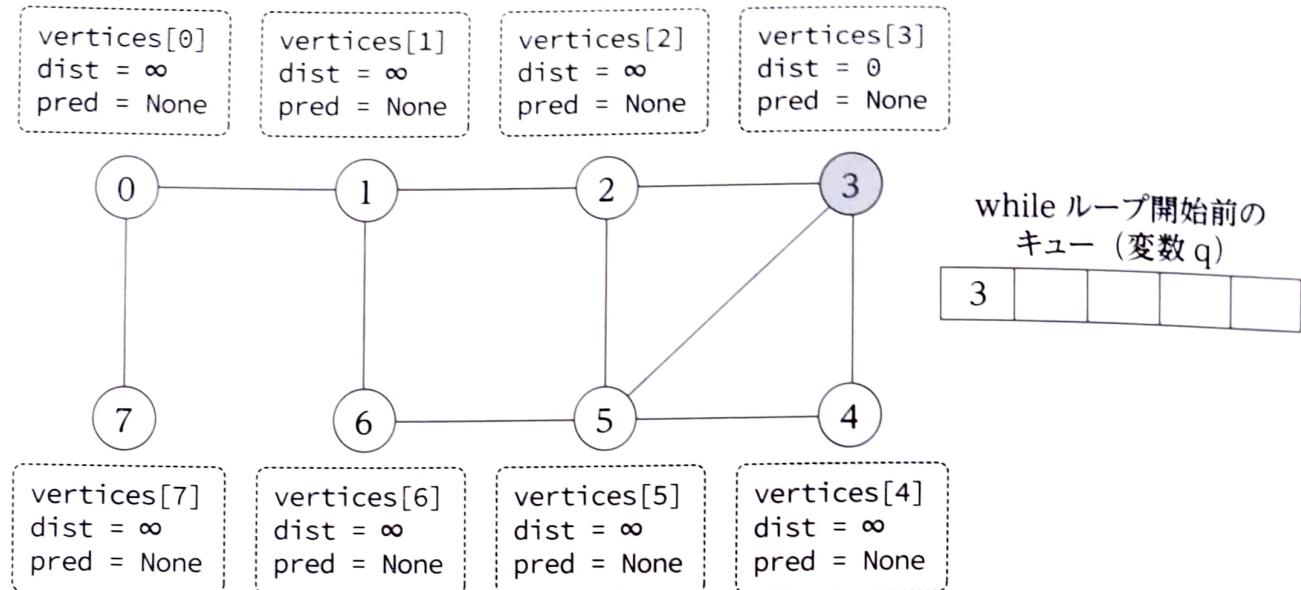


図 7.11 while ループ開始前の状態

while ループに入ると変数 q の先頭要素がデキューされ、頂点 3 の隣接頂点を調べます。color が WHITE である隣接頂点がキューにエンキューされるため、頂点 2 と 4、5 がエンキューされます。図 7.12 に 1 回目のイテレーションが終了した時点での状態を示します。vertices[2] と vertices[4] と vertices[5] の color と dist と pred が更新されています。

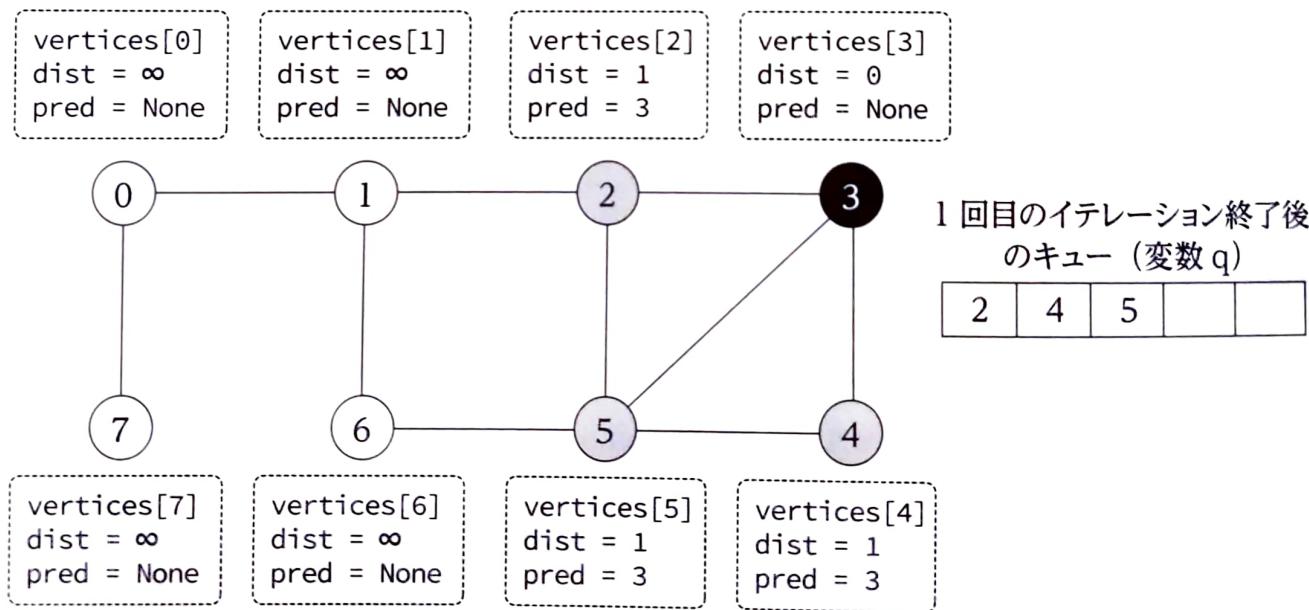


図 7.12 while ループの 1 回目のイテレーション終了後の状態

2回目のイテレーションでは、 q の先頭要素である頂点 2 がポップされます。頂点 2 は頂点 1 と 5 に隣接しています。頂点 5 の探索状態がすでに GRAY なので、無視します。そのため、頂点 1 だけがキューにエンキューされます。2回目のイテレーションが終了したときの状態を図 7.13 に示します。vertices[2].color と vertices[1] の各変数が更新されます。

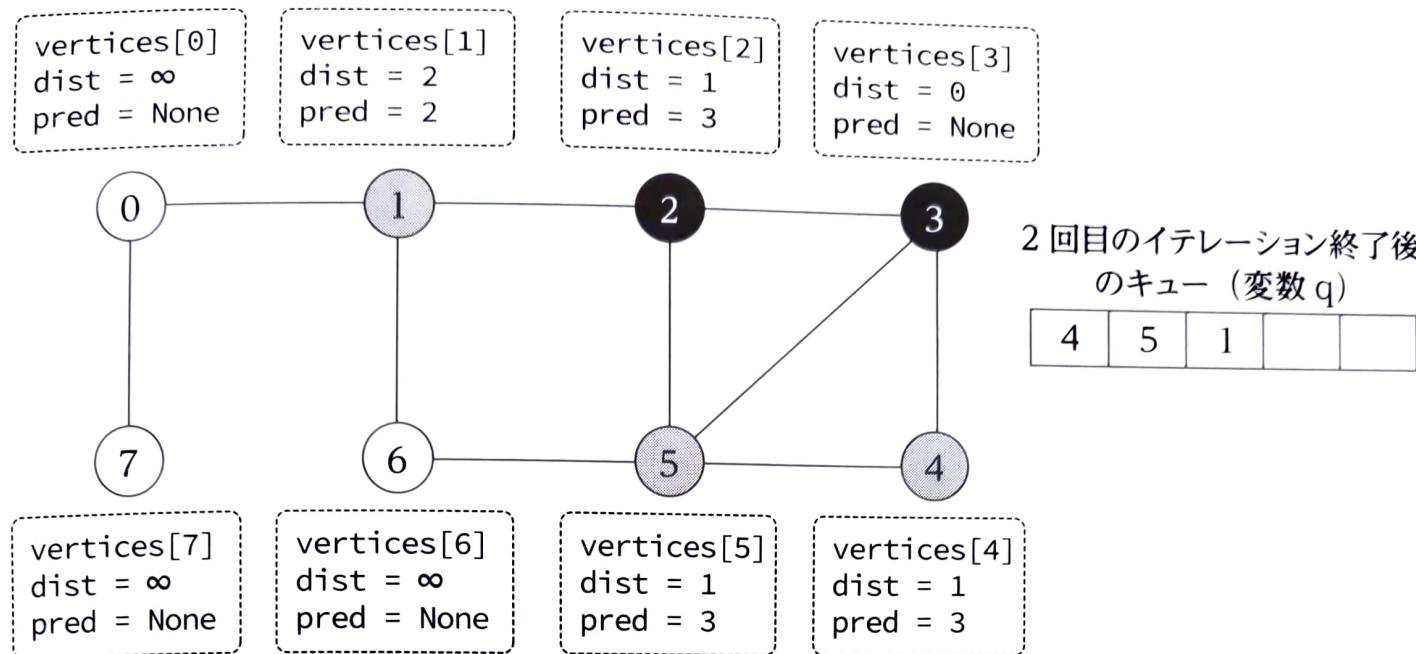


図 7.13 while ループの 2 回目のイテレーション終了後の状態

3回目のイテレーションでは、頂点4がデキューされますが、頂点4に隣接する頂点の中で探索状態がWHITEの頂点はありません。そのため、vertices[4].colorがBLACKに更新されますが、それ以外の変化はありません。3回目のイテレーション終了時の状態を図 7.14 に示します。

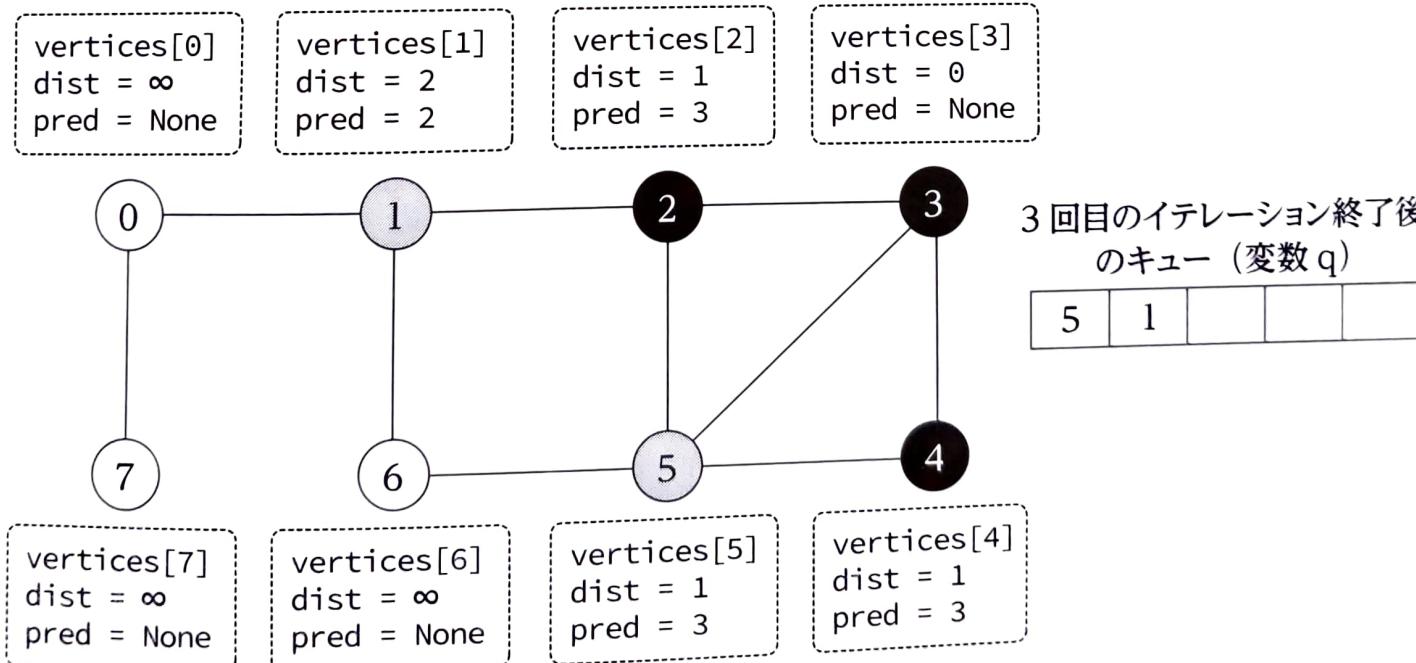


図 7.14 while ループの 3 回目のイテレーション終了後の状態

4回目のイテレーションでは頂点5がデキューされます。隣接頂点で、ある頂点6の探索状況がWHITEなので、頂点6をキューに入れて変数の値を更新します。図 7.15 に4回目のイテレーションが終了したときの状態を示します。

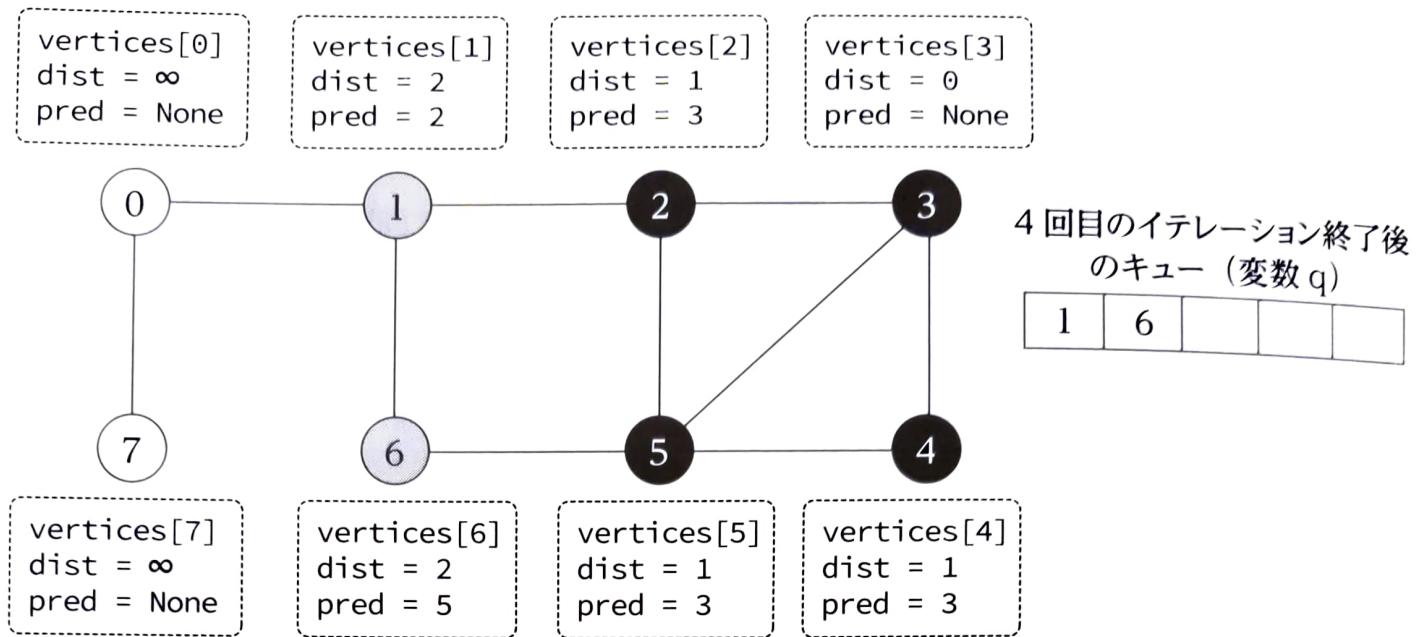


図 7.15 while ループの 4 回目のイテレーション終了後の状態

5回目のイテレーションも同様に処理します。頂点1がデキューされ、頂点0がエンキューされます。各頂点の状態は図 7.16 に示すとおりです。

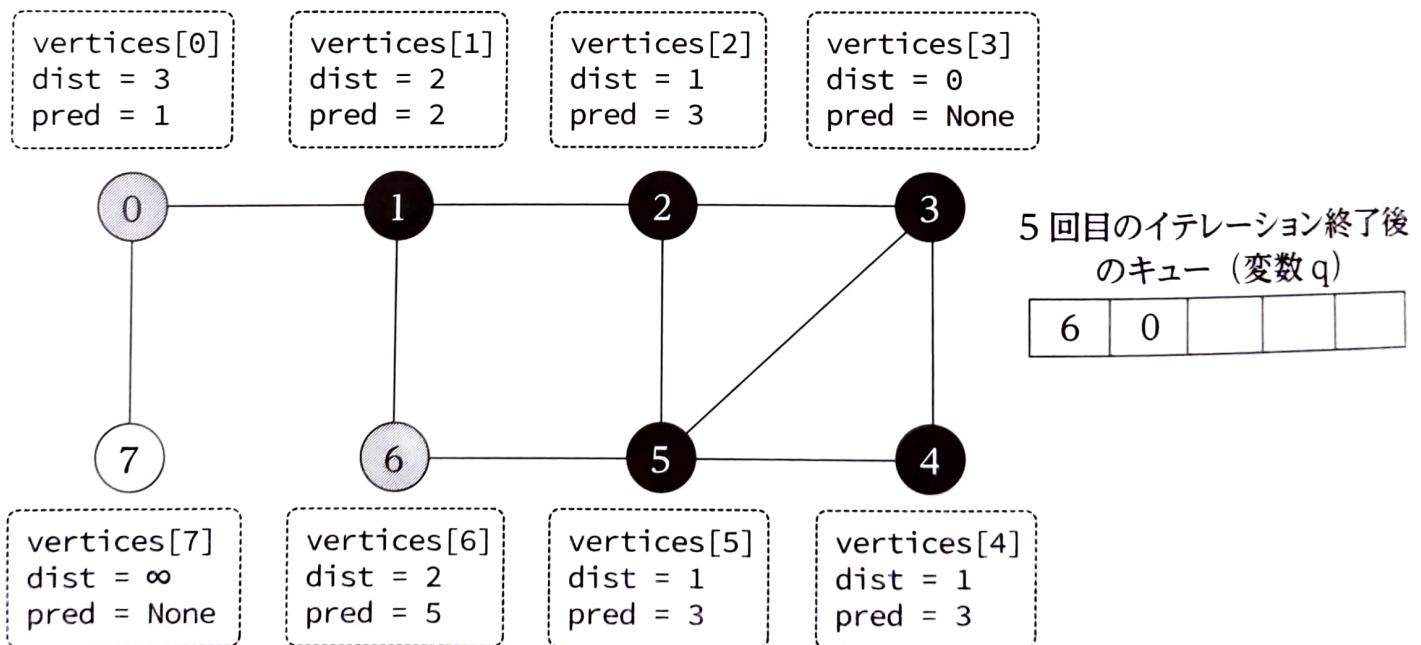


図 7.16 while ループの 5 回目のイテレーション終了後の状態

6回目のイテレーションでは、頂点6がデキューされますが、キューにエンキューする頂点が存在しないので、vertices[6].colorをBLACKに変更するだけです。6回目のイテレーション終了時の状態を図7.17に示します。

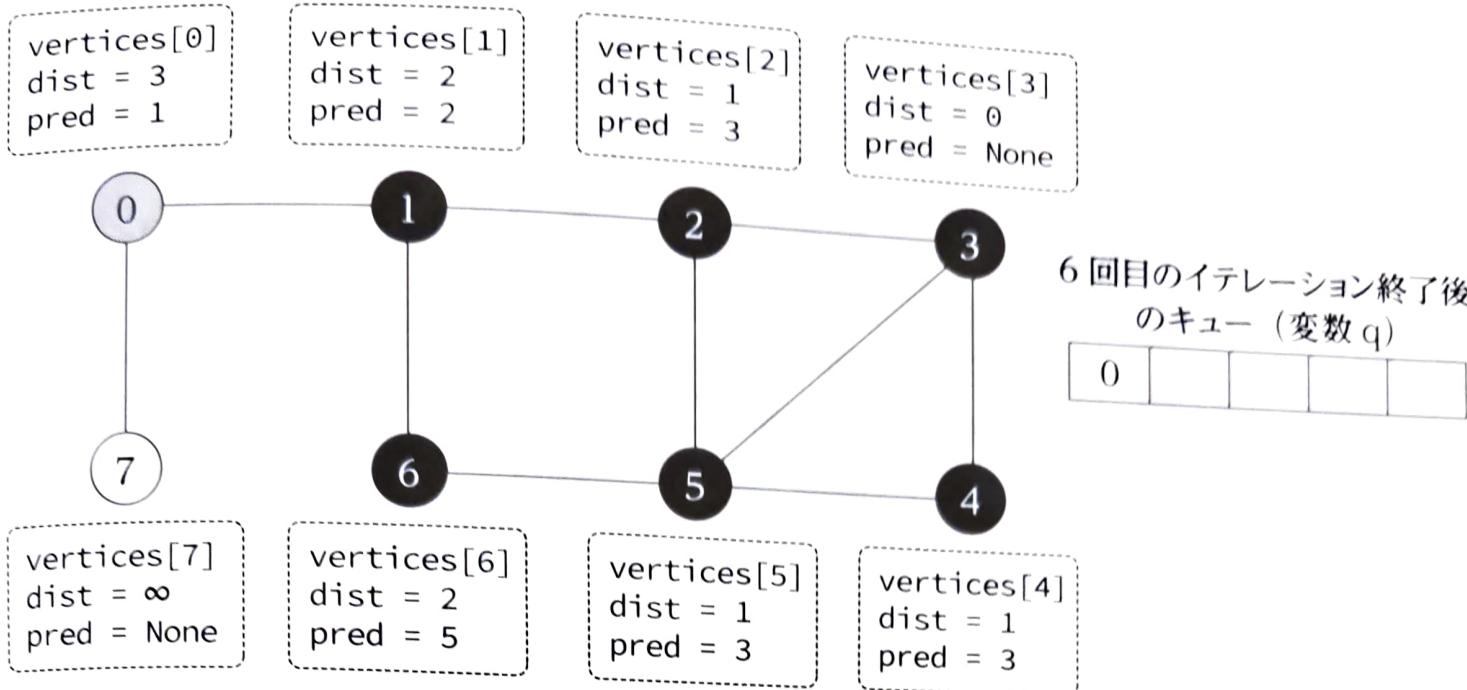


図 7.17 while ループの 6 回目のイテレーション終了後の状態

7回目のイテレーションも同様です。最後に頂点7がエンキューされます。図 7.17 に7回目のイテレーションが終了したときの状態を示します。

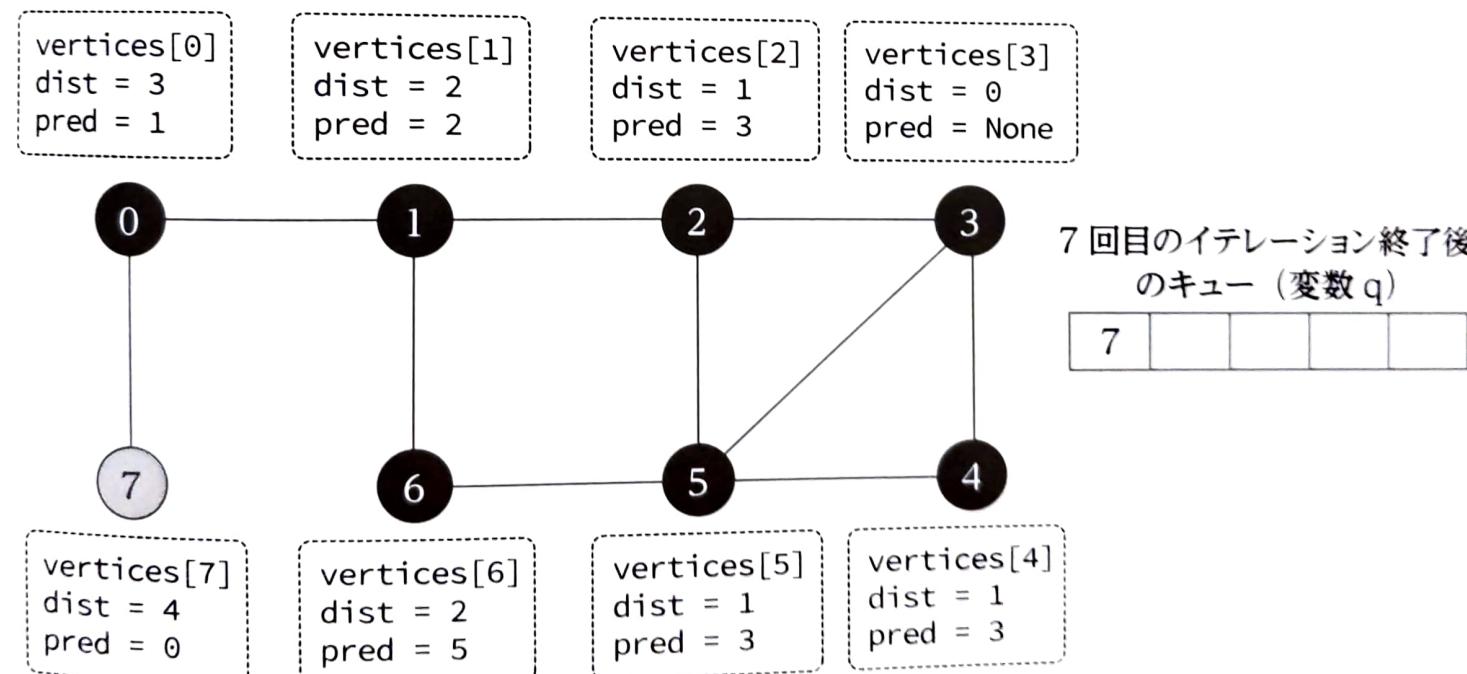


図 7.18 while ループの 7 回目のイテレーション終了後の状態

8回目のイテレーションで、頂点7がキューから取り出されますが、キューに入れる頂点が存在しません。そのため、vertices[7].color の値を BLACK に変更するだけです。ここでキューである変数 q が空になります。そのため、34 行目の $\text{len}(q) > 0$ が False になり、while ループを抜けます。ループ終了後の状態を図 7.19 に示します。