

図 7.44 頂点 u より先に頂点 v の隣接頂点を走査したときに起こる不具合 1

次に頂点 u の隣接頂点を走査します。そのときの状態を図 7.45 に示します。ここでリラックス処理によって、頂点 v の dist の値が 10 から 8 に更新されます。頂点 $s \rightarrow v$ という経路より、 $s \rightarrow u \rightarrow v$ という経路のほうが距離が短いからです。

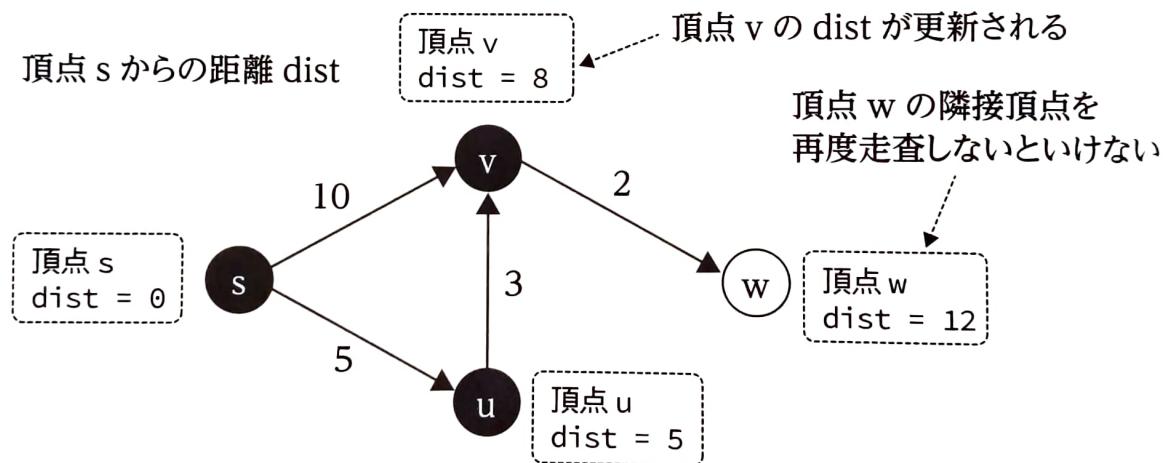


図 7.45 頂点 u より先に頂点 v の隣接頂点を走査したときに起こる不具合 2

当然、 v . dist の値が更新されると、 w . dist も更新しなければなりません。頂点 w への最短経路は、 $s \rightarrow u \rightarrow v \rightarrow w$ なので、 w . dist は 10 となるはずです。しかし、頂点 v はすでに走査済みです。正しい距離を設定しようとするならば、走査済みの頂点を再度走査しなければなりません。このような問題を回避するために優先度付きのキューを使用します。

各頂点の dist の小さい順から隣接頂点を走査した場合は上手く処理できます。図 7.46 に、 dist の値が一番小さい頂点 u を先に走査したときの状態を示します。 v . dist の値が 8 に更新されます。そして v . dist と w . dist の値を比べると、頂点 v の優先度のほうが高いので、頂点 v を表す円形を灰色に塗りつぶしています。

次に頂点 v の隣接頂点を走査したときの状態を図 7.47 に示します。 w . dist の値が 10 に更新されます。各頂点を操作する回数が 1 回で、かつすべての頂点の dist が正しく設定されています。

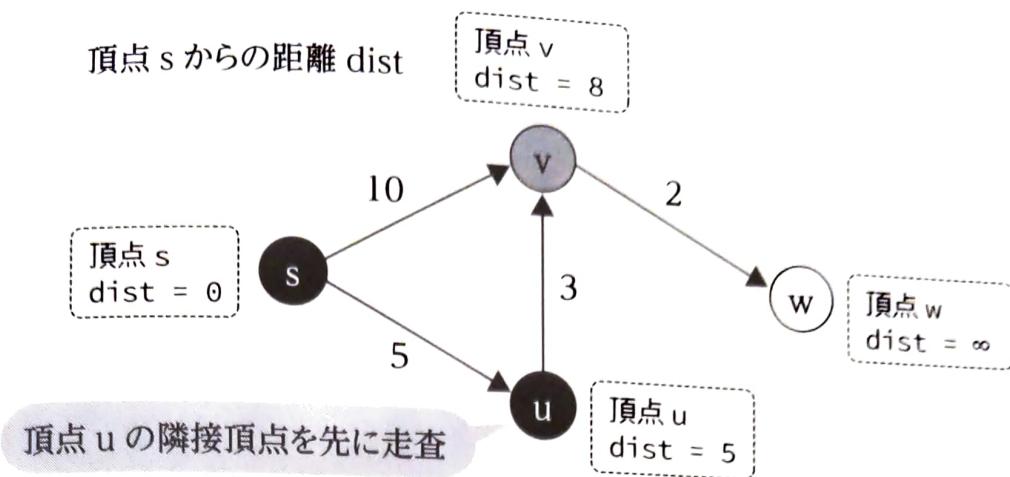


図 7.46 各頂点の dist を基にして優先度を用いたときの処理 1

次に頂点 v の隣接頂点を走査したときの状態を図 7.47 に示します。w.dist の値が 10 に更新されます。各頂点を操作する回数が 1 回で、かつすべての頂点の dist が正しく設定されています。

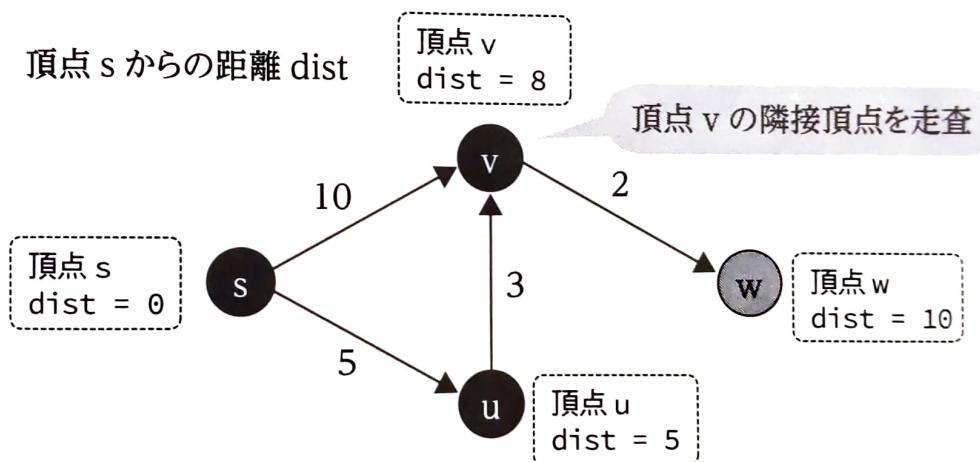


図 7.47 各頂点の dist を基にして優先度を用いたときの処理 2

■ Python が提供するヒープ

本書の例では、優先度付きキューとしてヒープを用います。第 6.4 節で解説したソースコード 6.3 (my_heap.py) では、大きな値をもつ要素の優先度を高くしましたが、ダイクストラ法では dist の値が小さい要素の優先度を高くします。そのため、max_heapify 関数や create_max_heap 関数を変更して、min_heapify 関数や create_min_heap 関数を定義する必要があります。これらを修正して適応しても良いですが、ソースコードが長くなるので、ここでは Python が提供するライブラリを使用します。

まず、ヒープが定義されているモジュール名と関数の概要を以下に示します。「変数名」は、ヒープ化したいリスト（配列）の変数の名前です。

インポート : import heapq

エンキュー関数：heapq.heappush (変数名, 新しい要素)

デキュー関数：heapq.heappop (変数名)

ヒープ化関数：heapq.heapify (変数名)

ヒープモジュールの使い方は簡単なので、具体例はダイクストラ法の実装例を見ながら解説します。

■ クラスの比較演算

ヒープを適応するためには、要素が比較可能でなければなりません。そうでなければ優先度が定義できないからです。キューの中には MyVertex オブジェクトの集合が格納されていて、それらの優先度は各オブジェクトがもつ dist の値を基準に優先度を決定します。すなわち、クラスのオブジェクト同士の比較演算を定義しないといけないです。

以下の MyVertex の定義の抜粋を見てください。4 行目で eq メソッドを実装しています。なお、eq は equal (等しい) の略です。

これは Python の拡張比較 (rich comparison) と呼ばれるもので、Python 言語そのものの解説は本筋から逸れるので、各自で公式 API ドキュメントを参照してください。このように記述すれば、あるクラスのオブジェクト同士の比較演算ができると考えてください。

引数の v は比較対象となる MyVertex オブジェクトです。2 つの頂点を表す MyVertex オブジェクト self と v がもつ self.dist と v.dist を同値であるかどうか、といった判定式を返します。

```
class MyVertex:
    def __eq__(self, v):
        return self.dist == v.dist
    # 以下、同様の処理を!=、<、<=、>、>=についても記述
```

同様に ne(not equal)、lt(less than)、le(less equal)、gt(greater than)、ge(greater equal)についても定義しますが、似たような処理になります。これらのメソッドは、ダイクストラ法の実装ソースコードで列挙します。本当は、例外処理 (引数 v が MyVertex オブジェクトかどうかなど) をしたほうが良いのですが、簡略化のために省いています。

■ 7.7.3 ダイクストラ法の実装例

ソースコード 7.11 (my_dijkstra.py) にダイクストラ法の実装例を示します。無向グラフで例を示すとステップ数が多くなるため、今回は有向グラフを用いています。

ソースコード 7.11

ダイクストラ法の実装

~/ohm/ch7/my_dijkstra.py

ソースコードの概要

6 行目～43 行目	頂点を表す MyVertex クラスの定義
46 行目～52 行目	リラックス処理を行う relax 関数の定義
55 行目～68 行目	ダイクストラ法を実行する dijkstra 関数の定義
71 行目～78 行目	始点から他の頂点への経路を表示する print_path 関数の定義
81 行目～82 行目	2 つの頂点を方向性をもつ辺で接続する connect 関数の定義
85 行目～89 行目	ヒープの中身を表示する print_heap 関数の定義
91 行目～122 行目	main 関数の定義

1 行目で heapq モジュールをインポートします。4 行目は無限大の定義です。71 行目～78 行目の print_path 関数はこれまでと同じ内容なので説明は省きます。85 行目～89 行目の print_heap 関数は、優先度付きキューとして使用しているリストの中身を表示する処理です。アルゴリズムに直接関係ないので解説は省きます。

```

01 import heapq
02
03 # 無限大の定義
04 INFTY = 2**31 - 1
05
06 class MyVertex:
07     def __init__(self, id):
08         self.id = id
09         self.adj = {} # dict型
10         self.dist = INFTY
11         self.pred = None
12
13     # == の定義
14     def __eq__(self, v):
15         return self.dist == v.dist
16
17     # != の定義
18     def __ne__(self, v):
19         return self.dist != v.dist

```

```

108     connect(vertices, 4, 1, 4)
109     connect(vertices, 4, 2, 11)
110     connect(vertices, 4, 3, 2)
111
112     # 頂点0を始点として探索
113     dijkstra(vertices, vertices[0])
114
115     # 頂点0から頂点2の経路を表示
116     print("頂点0から頂点2への経路: ", end = "")
117     print_path(vertices, vertices[0], vertices[2])
118     print("")
119
120     # 各頂点の表示
121     for i in range(0, N):
122         print(vertices[i].to_string())

```

■ MyVertex クラス (頂点を表す) の説明

6 行目～43 行目で頂点を表す MyVertex クラスを定義しています。クラスメンバに関しては、解説したとおりです。14 行目～35 行目で 6 種類 (=、!=、<、<=、>、>=) の比較演算を定義しています。すべて距離 dist をもとに同値、または大小を比較します。

また、隣接する頂点の識別子と辺の重みを表す変数 self.adj は dict 型なので、81 行目～82 行目の connect 関数で辺を追加するときの処理がこれまでとは若干異なります。connect 関数は、vertices[i] から vertices[j] へ向かう辺を追加し、その辺の重みである weight を設定します。vertices[i].adj[j] = weight と記述して、辺を追加します。

■ main 関数の説明

91 行目～122 行目で main 関数を定義しています。行っていることは幅優先探索や深さ優先探索とほぼ同じです。今回は 5 つの頂点から構成される有向グラフを用います。93 行目で変数 N を整数値の 5 で初期化し、96 行目～98 行目で MyVertex オブジェクトを生成します。101 行目～110 行目でグラフに辺を追加しています。それぞれの辺は重みがあるので、第 2 と第 3 引数が頂点の識別子、第 4 引数が辺の重みです。

図 7.48 にソースコード 7.11 で生成するグラフを示します。

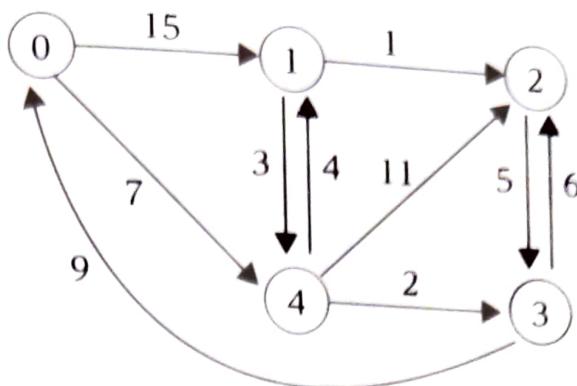


図 7.48 ソースコード 7.11 で用いる有向グラフ

113行目で `dijkstra` 関数を呼び出し、頂点 0 から到達可能なすべての頂点への最短経路を求めます。116行目～118行目で、頂点 0 から頂点 2 への経路を表示します。117行目の `print_path` 関数の第 2 引数は始点である `vertices[0]` を指定する必要があります。第 3 引数の頂点は `vertices[1] ~ vertices[4]` のいずれでも構いません。

最後に、121行目～122行目で各頂点の情報として、識別子 `id`、隣接頂点と辺の重み `adj`、始点からの距離 `dist`、先行頂点 `pred` を表示します。

■ `relax` 関数（リラックス処理）の説明

46行目～52行目でリラックス処理を行う `relax` 関数を定義しています。前項で解説した擬似コードとほぼ同じです。2つの頂点の `MyVertex` オブジェクトを変数 `u` と変数 `v` で受け取ります。47行目の `if` 文では、すでに発見された頂点 `v` へ到達する経路より、頂点 `u` を経由したほうが距離が短くなるかどうかを判定します。なお、頂点 `u` から `v` への辺の重みである `w(u, v)` ですが、変数 `adj` は `dict` 型なのでソースコード内では `u.adj[v.id]` となります。

`True` であれば、48行目と49行目で `v.dist` と `v.pred` の値を更新します。値を更新した場合は、戻り値として `True` を返します。条件判定の結果が `False` であれば、なにもせずに、戻り値として `False` を返します。戻り値を返す理由は、`v.dict` の値が変更されると、優先度付きキュー内の頂点 `v` の優先度も変化するからです。そのため、呼び出し元の `dijkstra` 関数内で、優先度付きキューを再度ヒープ化する必要があるかどうかを判断するために、`v.dist` の値を変更したかどうかを `True`、または `False` で返します。

■ `dijkstra` 関数（ダイクストラ法の実行）の説明

55行目～68行目でダイクストラ法を実行する `dijkstra` 関数を定義しています。引数として、頂点を表す `MyVertex` 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。

57行目～60行目は初期化処理です。57行目で始点となる頂点の `src.dist` の値を 0 にします。その他の頂点の `dist` は初期値の ∞ のままでです。58行目で変数 `q` を宣言し、空のリストで初期化します。

このリストを優先度付きキューとして使用します。59行目と60行目で変数 q に vertices のすべての要素をエンキューします。エンキューは、heapq モジュールの heappush 関数を使用します。関数名に push という単語が含まれていますが、キューなのでエンキューと呼びます。この関数を用いてエンキューするとヒープの性質を維持することができます。heapq.heappush(q, i) の第1引数はキュー、第2引数はエンキューする要素です。

変数 q の要素は MyVertex オブジェクトです。MyVertex クラスの定義で、比較演算子を定義したので、距離 dist をもとに優先度が決まります。heapq モジュールのヒープでは、小さい値が優先されます。すなわち、デキュー操作をすると、dist の値が一番小さい値が取り出されます。ちなみに heapq モジュールは大きい値の優先度を高くするといった機能は提供されていません。幸い本書の例では、dist の値が小さい頂点の優先度を高くするので、heapq モジュールをそのまま使用することができます。

63行目～68行目で探索を行います。63行目の while ループの開始時点で、変数 q にはすべての頂点の MyVertex オブジェクトが格納されています。各イテレーションで1つずつ取り出し、`len(q) > 0` という判定式のとおり、キューが空になるまで繰り返します。

64行目で優先度付きキューである変数 q の中身を表示するために print_heap 関数を呼び出します。アルゴリズムに関係ないのでコメントアウトしていますが、どのようにアルゴリズムが動作するかを確認したい読者はこの箇所をコメントインしてください。

65行目の `u = heapq.heappop(q)` という命令は、変数 q から一番 dist の値が小さい頂点をデキューし、変数 u に格納します。66行目の for ループで、頂点 u の隣接頂点を走査します。`u.adj` の要素はキーが頂点の識別子、値が辺の重みです。そのため、`for i in u.adj.keys():` と記述するとループカウンタ i にキーの値である隣接頂点の識別子が格納されます。

67行目で、頂点 u と隣接頂点の `vertices[i]` を指定して、relax 関数を呼び出します。もし、relax 関数内で、`vertices[i].dist` の値が更新されれば、優先度付きキューである変数 q 内での `vertices[i]` の優先度が変わります。relax 関数の戻り値が True であれば、`vertices[i].dist` の値が更新されたことを意味するので、68行目で変数 q を再度ヒープ化します。

具体例として、図 7.49 に、63行目の while ループ開始前の状態を示します。`vertices[0].dist` の値は 0 ですが、その他の頂点の dist と pred の値は初期値のままでです。図の右側に優先度付きキューである変数 q の中身を示します。左側がキューの先頭を表し、一番優先度が高い (dist の値が一番小さい) 要素が格納されています。

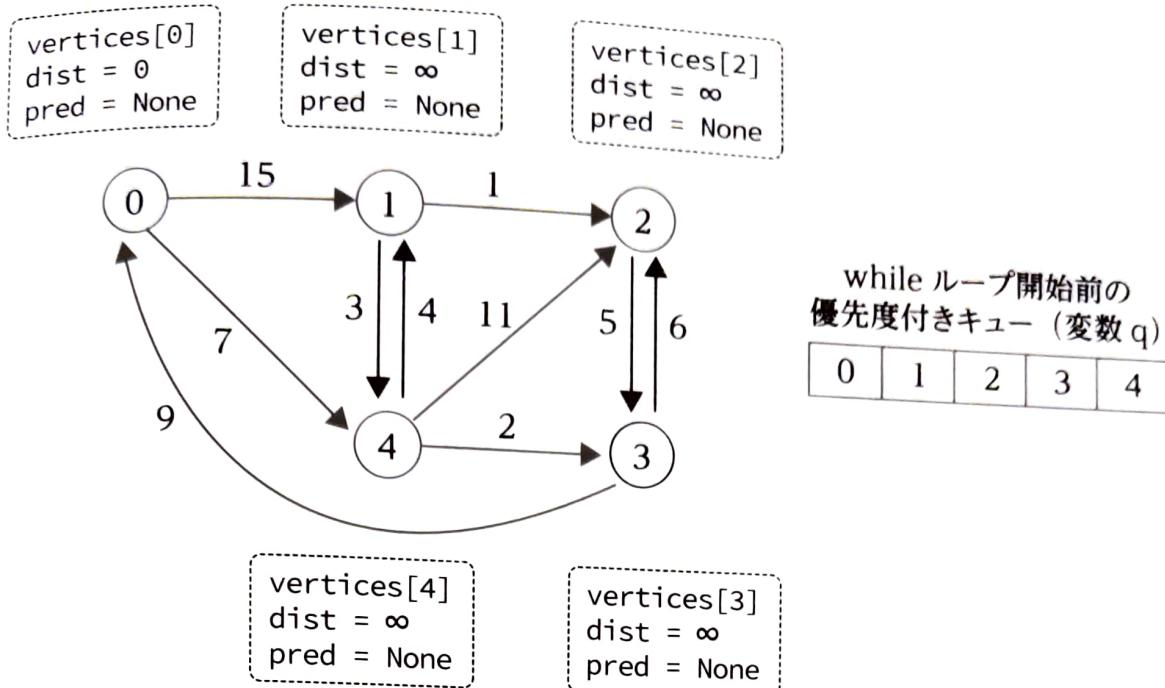


図 7.49 while ループ開始前の状態

while ループのイテレーションを進めて、頂点 0 を変数 q からデキューし、隣接頂点を走査します。図 7.50 に 1 回目のイテレーションが終了した時点でのグラフの状態を示します。頂点 1 の dist と pred、頂点 4 の dist と pred が更新されていることが確認できます。また、頂点 0 を指す円形は走査が終了したので、黒色で塗りつぶしています。一方、頂点 4 を指す円形は、変数 q に含まれる頂点の中で一番 dist の値が小さいので、灰色で塗りつぶしています。リラックス処理で、頂点 1 と 4 の値を更新したので、変数 q の中身も変わります。そのため、頂点 4 がキューの先頭に移動します。

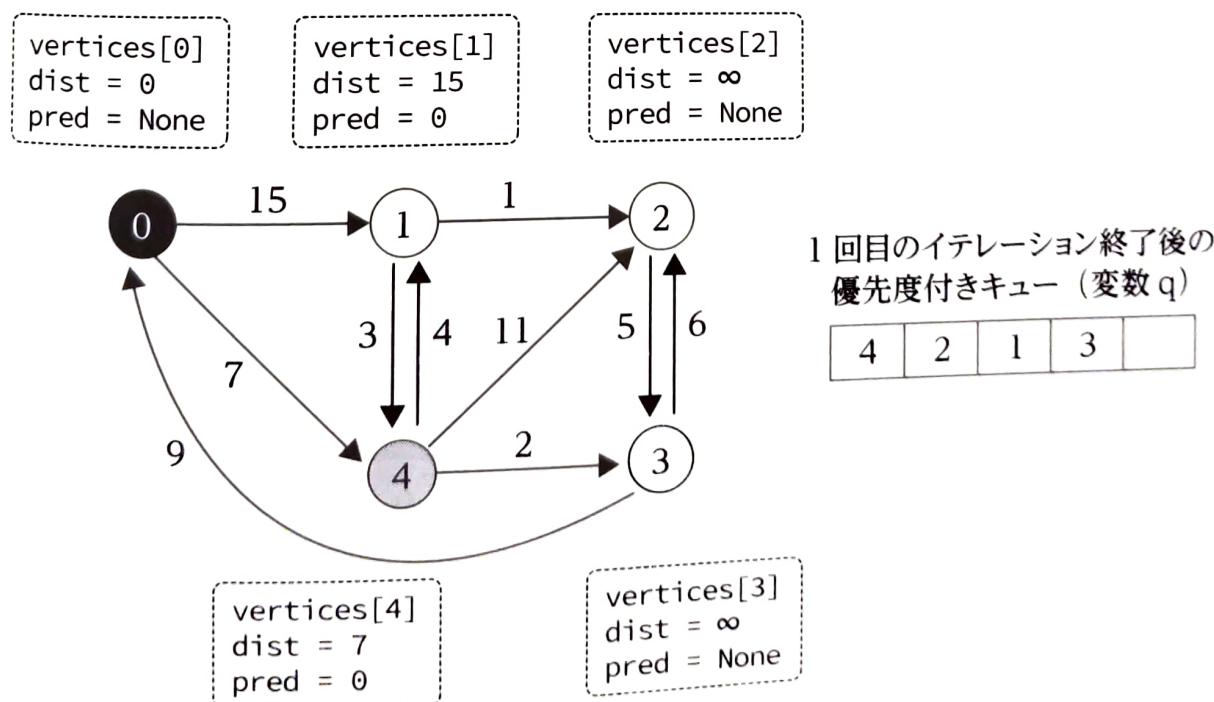


図 7.50 1 回目のイテレーション終了後の状態

2回目のイテレーション終了後の状態を図 7.51 に示します。頂点 4 をキューからデキューして隣接頂点を走査するので、頂点 1 と 2 と 3 の情報が更新されます。頂点 $0 \rightarrow 1$ という経路より、頂点 $0 \rightarrow 4 \rightarrow 1$ という経路のほうが距離が短くなるので、vertices[1].dist と vertices[1].pred が更新されています。頂点 2 と 3 は、初めて訪れる頂点なので、当然ながら変数が更新されます。キューに含まれる頂点の dist の値を見ると、頂点 3 が一番小さい dist の値をもつことがわかります。

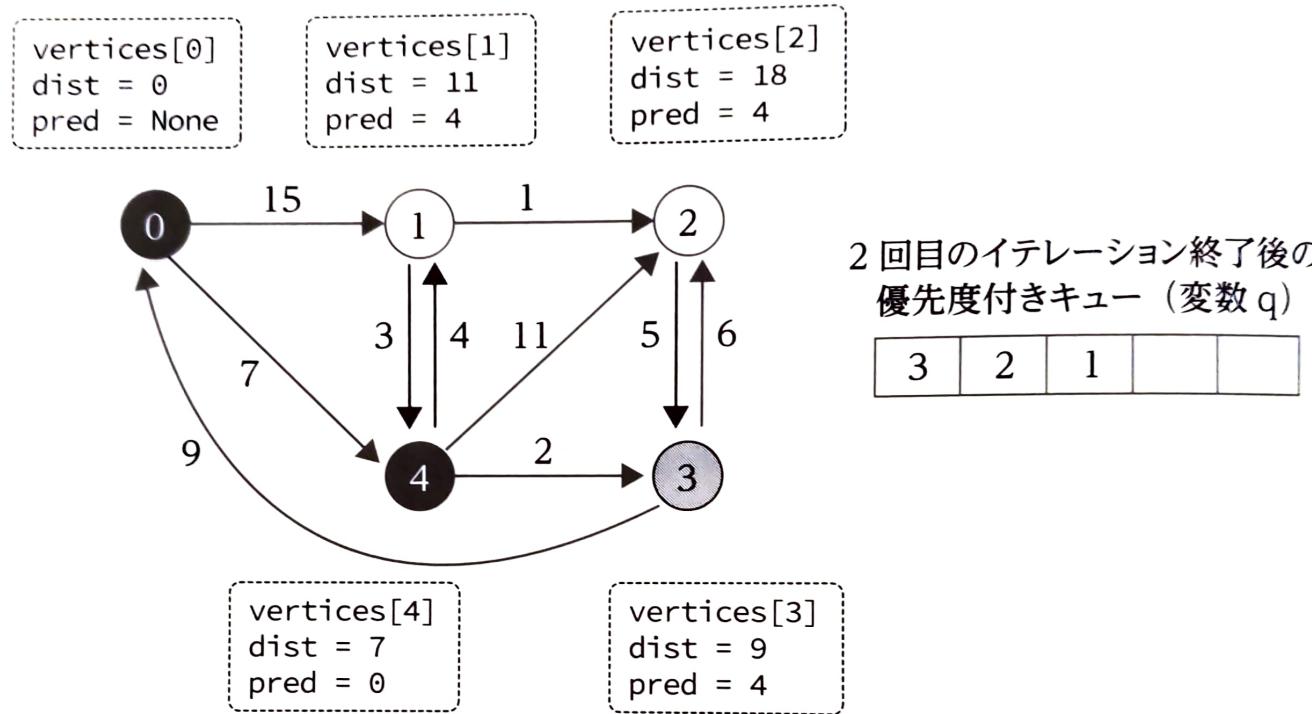


図 7.51 2回目のイテレーション終了後の状態

3回目のイテレーションでは、頂点 3 を変数 q からデキューし、隣接頂点を走査します。頂点 3 は頂点 0 と 2 に接続されていますが、リラックス処理で変数を更新するのは頂点 2 だけです。イテレーション終了後の状態を図 7.52 に示します。

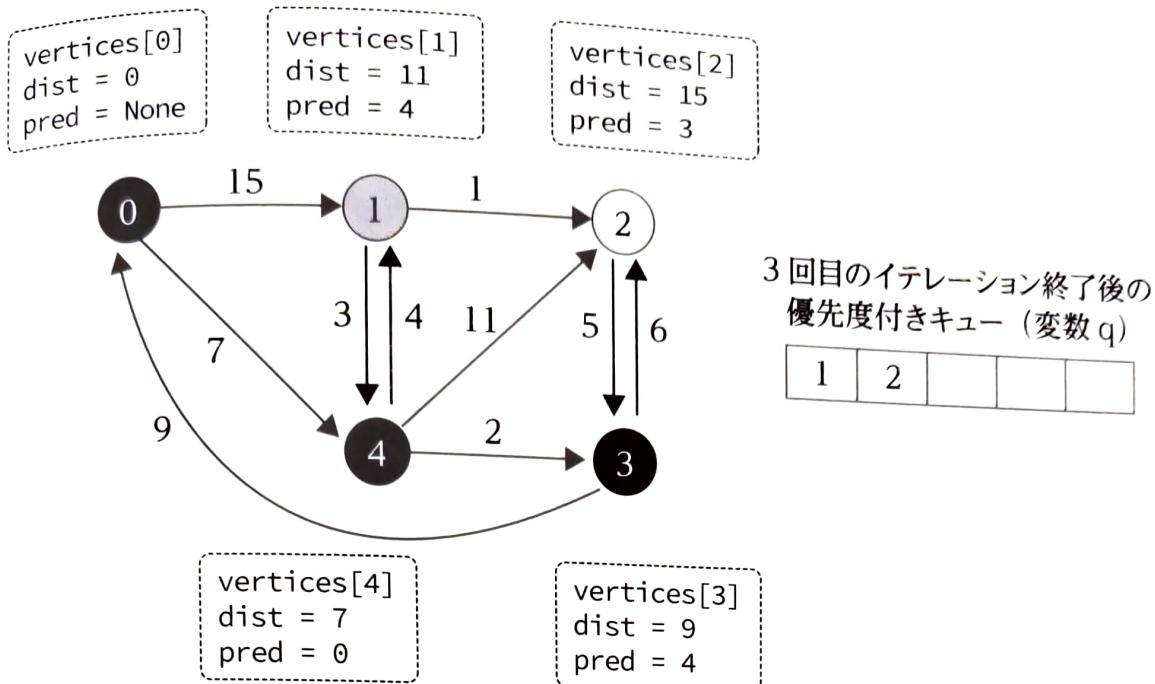


図 7.52 3回目のイテレーション終了後の状態

4回目のイテレーションでは、頂点 1 を変数 q からデキューし、隣接頂点を走査します。これまで発見された頂点 2 への最短経路は、頂点 $0 \rightarrow 4 \rightarrow 3 \rightarrow 2$ ですが、頂点 $0 \rightarrow 4 \rightarrow 1 \rightarrow 2$ の経路のほうが距離が短くなるので、vertices[2].dist と vertices[2].pred の値を更新します。4回目のイテレーション終了後の状態を図 7.53 に示します。

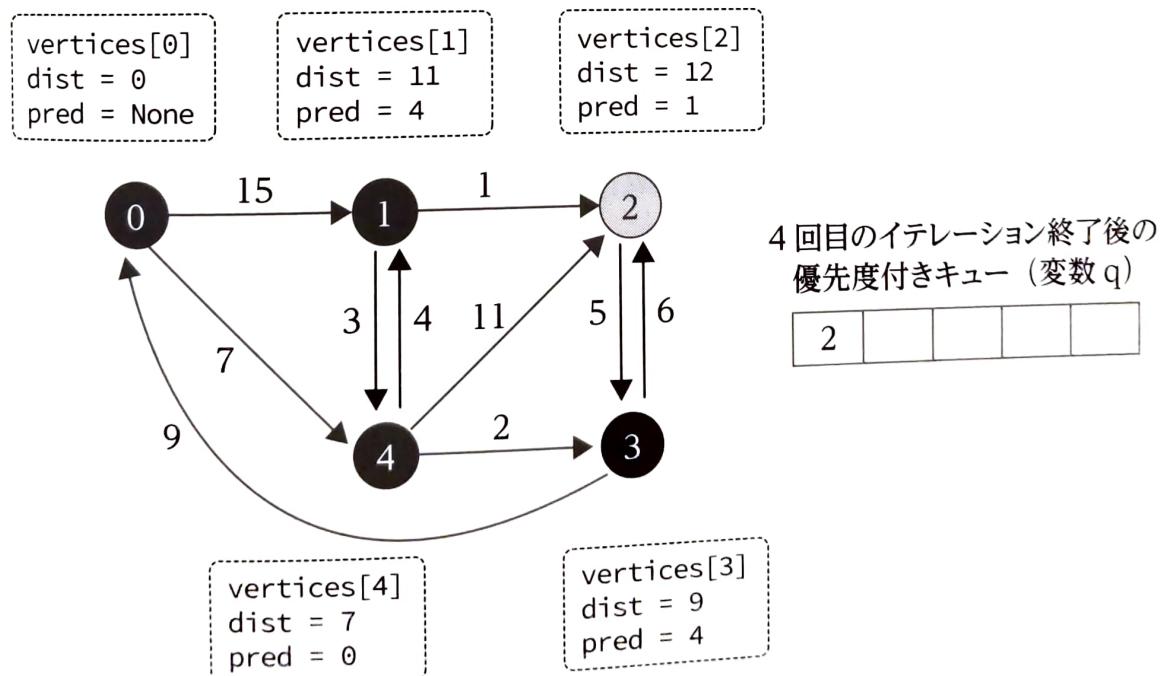


図 7.53 4回目のイテレーション終了後の状態

最後に頂点 2 を変数 q から取り出し、隣接頂点の走査をします。すべての隣接頂点は探索が完了している状態なので、リラックス処理で変数を更新する頂点はありません。5回目のイテレーション終了後、頂点 2 の dist 値が 12 に更新されます。

了後の状態を図 7.54 に示します。ここで変数 q が空になるので、while ループを抜けて処理を終了します。

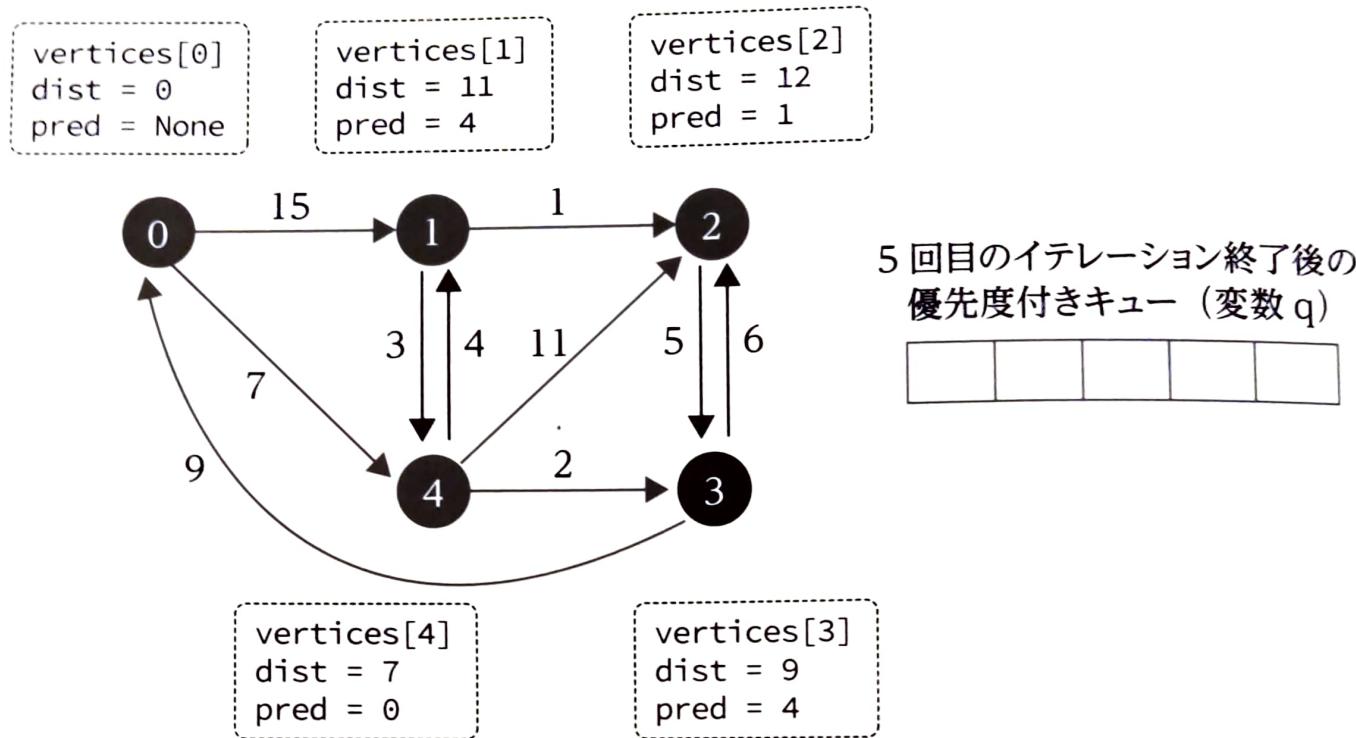


図 7.54 5回目のイテレーション終了後の状態

以上で最短経路の探索は終了です。頂点 0 から到達可能なすべての頂点の dist の値が設定されます。頂点の数が少ないので、最短経路を目で確認できると思いますが、すべての頂点の dist の値が最短経路の辺の重みの合計値になっています。また、pred の値も設定されているので、これをもとに経路調べることができます。

■ ダイクストラ法の実装プログラムの実行

ソースコード 7.11 (my_dijkstra.py) を実行した結果をログ 7.12 に示します。2 行目に頂点 0 から頂点 2 への最短経路が表示されています。図 7.54 からも確認できますが、経由する辺の重みの合計が最小の経路になっています。3 行目以降は、各頂点の変数の値を表示しています。図 7.54 で示したものと同じ値になっています。

ログ 7.12 my_dijkstra.py プログラムの実行

```
01 $ python3 my_dijkstra.py
```

7.7.4 ダイクストラ法の計算量

ダイクストラ法の計算量は、ヒープの実装方法によって異なります。63行目の while ループと 66 行目の for ループで、各頂点と各辺を 1 度ずつ走査します。合計で $V+E$ 回となります。さらに for ループの中で、リラックス処理をしたときにヒープ化を行っています。公式 API ドキュメントによれば、heapq モジュールの heapify 関数は線形時間の計算量がかかるため $O(V)$ です。そのため、本書の例では計算量が $O((V+E)V)$ です。

これは Python が提供している heapq モジュールの機能が悪いだけです。第 6.4 節で解説したヒープ実装のソースコード 6.3 (my_heap.py) で実装した max_heapify メソッドは $O(\log V)$ で動作します。最小ヒープをサポートする min_heapify メソッドを定義すれば、ダイクストラ法の計算量は $O((V+E)\log V)$ となります。

また、フィボナッチヒープ (Fibonacci heap) と呼ばれる高性能なヒープを使用すれば、 $O(E+V \log V)$ の計算量でダイクストラ法を実装できます。