# グラフとアルゴリズムとプログラムのやさしいおはなし

渡邉敏正

2022 年 1 月 28 日

**第 9 回**

## 1　グラフと行列

単純グラフや多重グラフをコンピュータで扱う代表的な方法として行列の利用があります。2 種類の行列を紹介します。一つは単純グラフを扱うことができる隣接行列、もう一つは単純グラフと多重グラフいずれも扱うことができる接続行列です。これらの行列を 2 次元配列に格納することでグラフをコンピュータに取り込むことができます。グラフ上の様々な特徴は行列のデータに反映され、それはコンピュータ内の配列にも受け継がれます。グラフの特徴抽出をはじめとするグラフに対する操作は、行列を介してコンピュータ内で配列に対する特徴抽出などの操作として実現されます。

### 1.1　隣接行列と接続行列

グラフをコンピューターで扱うことを考えましょう。図 74 のグラフ $G_4 = (V_4, E_4)$ を例として説明します。$G_4$ は第 6 回で取り上げています。

- $G_4$ は単純グラフです。まず､単純グラフをコンピュータで扱う場合によく使われる**隣接行列** (adjacency matrix) を紹介しましょう。

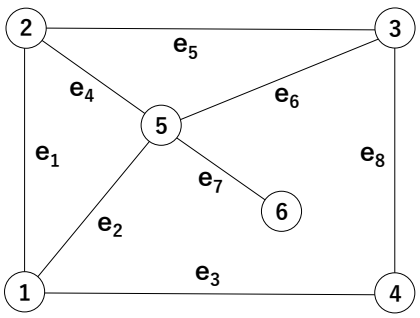


図 74　グラフ $G_4 = (V_4, E_4)$

- 点数 $n$ のグラフについては、通常、点を整数 1〜$n$ で表します。整数の扱いは容易ですし、配列添字にも使用できますし何かと便利です。ここでは、$n$ を 6 として

$V_4 = \{1,2,3,4,5,6\}$,
$E_4 = \{\, e_1 = \{1,2\}, e_2 = \{1,5\}, e_3 = \{1,4\}, e_4 = \{2,,5\}, e_5 = \{2,3\}, e_6 = \{3,5\},$
$e_7 = \{5,6\}, e_8 = \{3,4\}\,\}$
とします。

- このとき、$G_4$ の隣接行列 $A$ は以下のような 0,1 を要素とする 6 行 6 列の行列です (6×6 行列と表記することもあります)。一般的には $A$ は $n \times n$ 正方行列です。

$$A = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

- ここで、$A$ の行 (横方向) は

| | | | | | | |
|---|---|---|---|---|---|---|
| 第 1 行： | 0 | 1 | 0 | 1 | 1 | 0 |
| 第 2 行： | 1 | 0 | 1 | 0 | 1 | 0 |
| 第 3 行： | 0 | 1 | 0 | 1 | 1 | 0 |
| 第 4 行： | 1 | 0 | 1 | 0 | 0 | 0 |
| 第 5 行： | 1 | 1 | 1 | 0 | 0 | 1 |
| 第 6 行： | 0 | 0 | 0 | 0 | 1 | 0 |

であり、列 (縦方向) は

| 第 1 列 | 第 2 列 | 第 3 列 | 第 4 列 | 第 5 列 | 第 6 列 |
|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 1 | 0 |

です。

- $A$ の第 $i$ 行、第 $j$ 列の要素を $A(i,j)$ と表記することにします。例えば、$A(1,2) = 1,\ A(5,4) = 0$ です。
- $A$ を隣接行列とよぶ理由にもなりますが、第 $i$ 行は $G_4$ において点 $i$ に隣接する点の情報を表します。すなわち、

$\{i,j\} \in E_4$ ならば $A(i,j) = 1$;　$\{i,k\} \notin E_4$ ならば $A(i,k) = 0$　(図 75 参照)

です。第 $j$ 列についても全く同様に、点 $j$ に隣接する点を 1、それ以外を 0 として隣接点の情報を表します。したがって、$A(i,j) = A(j,i)$ となりますので $A$ は対称行列です。図 74 と行列 $A$ を見比べることで、隣接行列 $A$ の意味が理解できると思います。

- $G_4$ は単純グラフですので、隣接点と接続する辺は一対一に対応していますから、以下が成り立ちます：

**$i$ 行の 1 の個数は点 $i$ に接続する辺の数に、すなわち点 $i$ の次数に等しい**

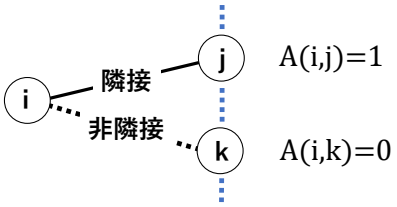


図 75　点 $i$ の隣接点 $j$ と非隣接点 $k$

- 一方、多重グラフでは隣接点との間に辺は複数存在する場合がありますので、隣接点の個数と次数が等しくなるとは限りません。多重グラフを行列で扱う場合には**接続行列** (incidence matrix) と呼ばれる以下のような行列 $C$ を利用することが多くなります。
- 接続行列を配列に格納する場合には通常、$G_4$ の辺 $e_1, \cdots, e_8$ もそれぞれ整数 $1, \cdots, 8$ として扱います。作り方は以下の通りです。
- 点 $i$ が第 $i$ 行に、辺 $j$ が第 $j$ 列に対応し、点 $i$ と辺 $j$ の接続関係および接続行列 $C$ は以下の通りです。

  点 $i$ に辺 $e_j$ が接続しているならば $C(i,j)=1$;
  点 $i$ に辺 $e_j$ が接続していないならば $C(i,j)=0$

$$C=\begin{bmatrix}1&1&1&0&0&0&0&0\\1&0&0&1&1&0&0&0\\0&0&0&0&1&1&0&1\\0&0&1&0&0&0&0&1\\0&1&0&1&0&1&1&0\\0&0&0&0&0&0&1&0\end{bmatrix}$$

- $G_4$ は単純グラフですので以後は隣接行列 $A$ で扱うことにします。

## 1.2　隣接行列の 2 次元配列への格納

行列 $A$ が $n\times m$ 行列のとき、行列の要素を整数型 2 次元配列 `A[n+1][m+1]` に格納することにします。$G_4$ の隣接行列 $A$ に対しては $n=m=6$ で、プログラムの最初に

```
#define n 6
#define m 6
```

と書いておき[*1]、行列のデータは宣言で設定するとして例えば以下のように記述します。

```
// この部分はプログラム本体の最初に記述
int A[n+1][m+1]={
     {0,0,0,0,0,0,0},{0,0,1,0,1,1,0},{0,1,0,1,0,1,0},
     {0,0,1,0,1,1,0},{0,1,0,1,0,0,0},{0,1,1,1,0,0,1},
     {0,0,0,0,0,1,0}
};
```

- まず、2 次元配列の宣言と隣接行列 $A$ について説明します。`int A[n+1][m+1]` が整数型 2 次元配列 `A`

[*1] 第 8 回の注意 8.5 で説明したマクロです。

の宣言です。さらに初期値設定を利用して行列の要素を格納しています。(プログラムでは n, m と共に 6 という値ですが、以下では n, m を使って説明しておきます。) これにより、コンピュータのメモリに整数を格納する領域が $(n+1)\times(m+1)$ 個だけ碁盤格子状に用意されます。一つの領域に一つの整数が入ります。(「碁盤格子状」というのはあくまでも説明用のイメージです。実際の様子はまた別の機会に説明する予定ですので、今はこのようなイメージを持っておいてください。)

- 図 76 を見てください。配列 A の $i$ 行 $j$ 列の要素は A[i][j] です。i,j が配列添字です。int A[n+1][m+1] と宣言すると配列添字 i を $0$〜$n$, j を $0$〜$m$ として以下の左側に示したイメージで $(n+1)\times(m+1)$ 個の整数型配列変数 (整数データを蓄える領域の名前) A[i][j] が用意され、右側に示した実際の値を格納します。$G_4$ については $n=m=6$ です。なお、横線と縦線は説明用に表記しています。
- 図の横線の下側部分、縦線の右側部分に

$$\texttt{A[i][j]} \ \leftarrow \ A(i,j) \quad (i=1,\cdots,n;\ j=1,\cdots,m)$$

として隣接行列 $A$ の要素 $A(i,j)$ を格納します。これで隣接行列の第 $i$ 行第 $j$ 列の要素 $A(i,j)$ と配列の要素 A[i][j] がストレートに対応し、非常にわかりやすくなりますので、C 言語プログラムではこの格納方法がよく採用されます。(この辺りのことは 第 10 回 1.6.2 注意 8.6 を参照して下さい。) なお、横線の上側部分、縦線の左側部分は値を 0 に設定しますが、プログラムでは使用しません。また、データを表示する場合にもこれらの表示は除外することが普通です。

| | | | |
|---|---|---|---|
| A[0][0] | A[0][1] | ... | A[0][m] |
| A[1][0] | A[1][1] | ... | A[1][m] |
| ⋮ | ⋮ | ⋱ | ⋮ |
| A[n][0] | A[n][1] | ... | A[n][m] |

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 |

図 76　隣接行列を格納する 2 次元配列の宣言と実際の行列要素

## 1.3　隣接行列からグラフの点の次数を求める

ここで、隣接行列 $A$ から $G_4$ の点 $1,\cdots,6$ の次数 $d(1),\cdots,d(6)$ を求めるアルゴリズムを考えてみましょう。

$A$ の第 $i$ 行 $A(i,1),\cdots,A(i,6)$ の中で $A(i,j)=1$ は隣接点、$A(i,k)=0$ は非隣接点を表していますので、$A(i,1)+\cdots+A(i,6)$ **の値が点** $i$ **の隣接点の個数**であり、$G_4$ は単純グラフですから、これは**点** $i$ **の次数** $d(i)$ に等しくなります。

この考え方で、$i=1,\cdots,6$ と各点 $i$ の次数 $d(i)$ を計算していきます。

(1) $i=1$ のとき:

(a) $d(1)\leftarrow 0$;

(b) $j=1$ から $j=6$ まで各 $j$ について以下の計算を反復する:
$d(1) \leftarrow d(1)+A(1,j)$;

$i=2$ の場合に進む前に、上記の手順で $d(1)$ が正しく計算できることを確かめておきましょう。なお、以下の=は等号です。

第 1 行： 0　1　0　1　1　0

ですので、

- (開始前) $d(1)=0$;
- $j=1:\ d(1) \leftarrow d(1)+A(1,1)\,(=0+0=0)$; //矢印の右側で $d(1)=0$、左側で $d(1)=0$ です
- $j=2:\ d(1) \leftarrow d(1)+A(1,2)\,(=0+1=1)$; //矢印の右側で $d(1)=0$、左側で $d(1)=1$ です
- $j=3:\ d(1) \leftarrow d(1)+A(1,3)\,(=1+0=1)$; //矢印の右側で $d(1)=1$、左側で $d(1)=1$ です
- $j=4:\ d(1) \leftarrow d(1)+A(1,4)\,(=1+1=2)$; //矢印の右側で $d(1)=1$、左側で $d(1)=2$ です
- $j=5:\ d(1) \leftarrow d(1)+A(1,5)\,(=2+1=3)$; //矢印の右側で $d(1)=2$、左側で $d(1)=3$ です
- $j=6:\ d(1) \leftarrow d(1)+A(1,6)\,(=3+0=3)$; //矢印の右側で $d(1)=3$、左側で $d(1)=3$ です

となって、確かに $d(1)=3$ と正しく計算されています。次に $i=2$ の場合に進みましょう。

(2) $i=2$ のとき:

(a) $d(2) \leftarrow 0$;
(b) $j=1$ から $j=6$ まで各 $j$ について以下の計算を反復する:
$d(2) \leftarrow d(2)+A(2,j)$;

いま

第 2 行： 1　0　1　0　1　0

ですので、

- (開始前) $d(2)=0$;
- $j=1:\ d(2) \leftarrow d(2)+A(2,1)\,(=0+1=1)$;
- $j=2:\ d(2) \leftarrow d(2)+A(2,2)\,(=1+0=1)$;
- $j=3:\ d(2) \leftarrow d(2)+A(2,3)\,(=1+1=2)$;
- $j=4:\ d(2) \leftarrow d(2)+A(2,4)\,(=2+0=2)$;
- $j=5:\ d(2) \leftarrow d(2)+A(2,5)\,(=2+1=3)$;
- $j=6:\ d(2) \leftarrow d(2)+A(2,6)\,(=3+0=3)$;

となって、確かに $d(2)=3$ と正しく計算されています。

以下、同じ形式の計算を $i=1,2,\cdots$ と反復していけば各点の次数を計算できることが理解できると思います。

- $i=1,,\cdots,6$ についての計算をまとめて書くと以下のようになります。

1. $i=1,\cdots,6$ の順に以下の (a), (b) の計算を反復する:

2. (a) $d(i) \leftarrow 0$;
   (b) $j = 1$ から $j = 6$ まで各 $j$ について以下の計算を反復する:
   $d(i) \leftarrow d(i) + A(i, j)$;

- この次数を計算するアルゴリズムは、$i$ に関する 1 重ループの中に、$j$ に関する 1 重ループを含む形になっています。このような反復処理を 2 重ループとよぶことは前回に説明しました。
- 整数型 1 次元配列を `int d[n+1]` と宣言して整数型変数 `i,j` を使ってこの部分を C 言語で書くと、例えば以下のようになります。ここでは `d[0]` は扱っていません。(整数型の場合、初期値を特に設定しなければ通常は宣言によって初期値は自動的に 0 に設定されます。)

```
int d[n+1];
int i, j;   // ここまでの 2 行はプログラム本体の最初に記述
// Computing degrees of vertices
for (i=1; i<=n; i++){
    d[i]=0;
    for (j=1; j<=n; j++){
        d[i]=d[i]+A[i][j];
    }
}
```

- $G_4$ が隣接行列として与えられたときに、各点の次数を計算するための C 言語プログラムとその実行結果をそれぞれ図 77 と図 78 に示します。プログラムおよび実行結果について説明しておきます。
- まず、図 77 についてです。
  - 13 行目までは $n, m$ の数値定義 (オブジェクト形式マクロ)、および宣言と初期値設定を利用した隣接行列のデータ入力です。2 次元配列 `A` の行配列添字 0 の変数に入る要素と列配列添字 0 の変数に入る要素は値をすべて 0 に設定しています。
  - 17 行目〜23 行目: 上述した次数の計算アルゴリズムを C 言語で記述しています。$i = 1, \cdots, n\,(n = 6)$ について、次数 $d(i)$ を 1 次元配列の `d[i]` に格納しています。
  - 24 行目〜31 行目: 2 次元配列 `A` に格納された隣接行列 $A$ をディスプレイに表示しています。
  - 32 行目〜36 行目: 1 次元配列 `d` に格納された次数ベクトル $\boldsymbol{d}$ をディスプレイに表示しています。
- 次に、図 78 です。図 77 のプログラムの実行結果 (ディスプレイの表示) です。
  - 1〜2 行はコンパイルと実行の操作です。
  - その下側には、2 次元配列 `A` の要素を表示しています。
  - さらにその下側には、求めた次数ベクトルを表示しています。

**ちょっと一言** $G_4$ の各点の次数が計算できましたので、もし $G_4$ が連結であるならば、点の次数をチェックしていけば第 4 回の結果からオイラーサイクルがあるかどうかを判定できます。次数がすべて偶数であればオイラーサイクルが存在し、1 つでも奇数の次数があればオイラーサイクルは存在しません。ただ、「グラフが連結であれば」という条件があります。$G_4$ の場合は連結であることはすぐにわかりますが、一般的に言えば、与えられたグラフが連結かどうかを判定することが要求されます。基点 $s$ から連結な点をすべて求めるアルゴリズムはこの連結性判定に直接関係します。このことについては第 10 回で **compo(G, s)** のプログラムを作成してから説明します。

ところで、$G_4$ は連結ですので上記で求めた次数ベクトルを格納している 1 次元配列 `d` の要素を見ていけばオイラーサイクルが存在するか否かを判定できます。ここまでの理解度チェックも兼ねて図 77 のプログラム

```
1  // Computing vertex-degrees of a graph
2
3  #include <stdio.h>
4  #define n 6
5  #define m 6
6
7  int main(void)
8  {
9      int d[n+1];
10     int i, j;
11     int A[n+1][m+1]={
12         {0,0,0,0,0,0,0},{0,0,1,0,1,1,0},{0,1,0,1,0,1,0},{0,0,1,0,1,1,0},
           {0,1,0,1,0,0,0},{0,1,1,1,0,0,1},{0,0,0,0,0,1,0}
13     };
14
15     printf("\n ***** Computing degrees of vertices ******\n");
16
17     // Computing degrees of vertices
18     for (i=1; i<=n; i++){
19         d[i]=0;
20         for (j=1; j<=n; j++){
21             d[i]=d[i]+A[i][j];
22         }
23     }
24     // Showing an adjacency matrix
25     printf("\n***** Adjacency matrix *****\n");
26     for (i=1; i<=n; i++){
27         for (j=1; j<=n; j++){
28             printf("A[%d][%d]=%d  ",i,j,A[i][j]);
29         }
30         printf("\n");
31     }
32     // Showing degrees of vertices
33     printf("***** Degrees of vertices *****\n");
34     for (i=1; i<=n; i++){
35         printf("d[%d]=%d\n",i,d[i]);
36     }
37     printf("\n\n");
38
39     return 0;
40 }
```

図 77　$G_4$ の隣接行列から点の次数を求める C 言語プログラム：左端の行番号は説明用に入れています (プログラムに入れるとエラーになります)

```
Toshimasa-no-MacBook-Pro:hmh-hp watanabe$ gcc -o comp-deg comp-deg.c
Toshimasa-no-MacBook-Pro:hmh-hp watanabe$ ./comp-deg

 ***** Computing degrees of vertices ******

***** Adjacency matrix *****
A[1][1]=0  A[1][2]=1  A[1][3]=0  A[1][4]=1  A[1][5]=1  A[1][6]=0
A[2][1]=1  A[2][2]=0  A[2][3]=1  A[2][4]=0  A[2][5]=1  A[2][6]=0
A[3][1]=0  A[3][2]=1  A[3][3]=0  A[3][4]=1  A[3][5]=1  A[3][6]=0
A[4][1]=1  A[4][2]=0  A[4][3]=1  A[4][4]=0  A[4][5]=0  A[4][6]=0
A[5][1]=1  A[5][2]=1  A[5][3]=1  A[5][4]=0  A[5][5]=0  A[5][6]=1
A[6][1]=0  A[6][2]=0  A[6][3]=0  A[6][4]=0  A[6][5]=1  A[6][6]=0
***** Degrees of vertices *****
d[1]=3
d[2]=3
d[3]=3
d[4]=2
d[5]=4
d[6]=1
```

図 78　$G_4$ の隣接行列から点の次数を求めるプログラムの実行結果 (ディスプレイの表示)

に、この判定操作を付け加えことを考えてみてください。皆さんへの演習問題のつもりです。偶数かどうかの判定や配列 d の要素をチェックする操作は第 8 回に参考になる説明があります。

## 1.4　オイラーサイクルに関するおさらい

オイラーサイクルとは、連結なグラフにおいて、ある 1 点から出発してすべての辺をちょうど 1 度だけ通過して出発点に戻るようなサイクルのことです。

さて皆さん、上述の「ちょっと一言」でお願いした演習問題として、判定プログラムを記述することに挑戦していただいたでしょうか。それはどのような判定操作の記述になったでしょうか。以下では、まずオイラーサイクルの存在性判定を考えてみましょう。

### 1.4.1　連結グラフにオイラーサイクルが存在する否かの判定

- いま、($n = 6$ として) 1 次元配列 d の d[1], $\cdots$, d[n] に次数 $d(1), \cdots, d(n)$ が求められているとしましょう。
- 第 4 回の結果に基づけば、( 連結グラフであれば ) これらの偶奇をチェックしていき、全てが偶数ならばオイラーサイクルは存在し、1 つでも奇数があればオイラーサイクルは存在しません。
- この考え方に基づいて次のような判定操作が考えられます。なお、以下の $=$ は等号で、$d(i)\%2$ は $d(i)$ を 2 で割ったとき (整数商) のあまりを表します。この場合のあまりは 0 ($d(i)$ は偶数) か 1 ($d(i)$ は奇数) です。
  1. 整数値を格納する変数 $F$ を用意し、$F \leftarrow 0$ と初期値設定をする;
  2. $i = 1, \cdots, n$ の順に各 $i$ について以下の操作を行う:　// $n = 6$ です
     もし $d(i)\%2 = 0$ ならば何もしない;
     もし $d(i)\%2 \neq 0$ ならば $F \leftarrow 1$ とし、このループを抜けて 3 に進む;

3. もし $F=0$ ならば「オイラーサイクルは存在する」と表示する;
   もし $F\neq 0$ ならば「オイラーサイクルは存在しない」と表示する;
4. 終了;

- ここで、$d(i)\%2\neq 0$ なる $i$ が出ると $F\leftarrow 1$ なる操作後、break でこのループを抜けて次の if〜else の条件判断文に進みます。上述の操作を C 言語で記述してみると例えば以下となります。

```
int F=0;   // この行はプログラム本体の最初に記述
//Deciding existence of an Euler cycle
for (i=1; i<=n;i++){
    if (d[i]%2 != 0){
        F=1; break;
    }
}
if (F==0){  // すべての点の次数が偶数
    printf("オイラーサイクルは存在します\n\n");
}
else{  // F==1: 次数が奇数の点が存在
    printf("オイラーサイクルは存在しません\n\n");
}
```

- 以上のことを考慮して作成したプログラムが図 79 です。その実行結果が図 80 となります。図 80 は図 78 に「オイラーサイクルは存在しません」という表示が追加されただけです。以下では、図 79 と図 77 との違い (追加した部分) について説明しておきます。図 77 に追加したのは以下の部分です。
  - 9 行目の F=0 : 整数値 0, 1 を格納する変数 $F$ を宣言し、さらに初期値を 0 に設定しています。$F$ の値が 0 ならばすべての次数が偶数であることを表し、$F$ の値が 1 ならば奇数の次数が存在することを表します。
  - 36 行目〜47 行目: 37 行目〜41 行目で 1 次元配列 d の d[1], $\cdots$ , d[n] を順に見ていき、格納されている次数 $d(1),\cdots,d(n)$ の中にもし奇数があれば $F$ を 1 に変更し、すぐその後で break によりこのループを抜けて次の if〜else 文に進みます。(奇数がなければ $F$ の値は 0 のままでループを終了します。)
  - 42 行目〜47 行目にある if$\cdots$ else $\cdots$ が条件判断文です。$F$ の値が 0 か 1 かによってオイラーサイクルが存在するか否かを判断してそのこと表示します。すなわち、(if) 条件 F==0 ($F$ が 0 に等しいこと) が成り立つならば "オイラーサイクルは存在します" とディスプレイに表示し、(else) 条件 F==0 が成り立たない ($F$ が 0 に等しくないこと) ならば "オイラーサイクルは存在しません" とディスプレイに表示します。$F$ が 0 に等しいことはすべての点の次数が偶数であることを意味し、$F$ が 0 でないこと (今の場合は 1 であること) は奇数次数を持つ点が存在することを意味します。第 4 回で示しましたように、連結グラフにおいて、前者ならばオイラーサイクルは存在し、後者ならばオイラーサイクルは存在しません。

**注意 9.2** 図 79 の 39 行目にある break; ですが、これは

**処理の流れを強制的に終了し、そのブロックから抜ける**

という処理を行う文 (命令) です。今の場合、ブロックは 1 重ループのことになりますので、break; でループを抜けて次の if〜else 文の処理に移ることになります。処理を途中で強制的に終了したい場合によく使われます。

- 図 80 に示した結果は、オイラーサイクルを持たないグラフ $G_4$ についてでした。では、図 81 のグラフ

```
1  // Computing vertex-degrees of a connected graph and deciding
existence of an Euler cycle
2  #include <stdio.h>
3  #define n 6
4  #define m 6
5
6  int main(void)7  {
8      int d[n+1];
9      int i, j, F=0;
10     int A[n+1][m+1]={
11          {0,0,0,0,0,0,0},{0,0,1,0,1,1,0},{0,1,0,1,0,1,0},
            {0,0,1,0,1,1,0},{0,1,0,1,0,0,0},{0,1,1,1,0,0,1},
            {0,0,0,0,0,1,0}
12     };
13
14     printf("\n ***** Showing adjacency matrices of graphs
******\n");
15     // Computing degrees of vertices
16     for (i=1; i<=n; i++){
17         d[i]=0;
18         for (j=1; j<=n; j++){
19             d[i]=d[i]+A[i][j];
20         }
21     }
22     // Showing adjacency matrices
23     printf("\n***** Adjacency matrix *****\n");
24     for (i=1; i<=n; i++){
25         for (j=1; j<=n; j++){
26             printf("A[%d][%d]=%d  ",i,j,A[i][j]);
27         }
28         printf("\n");
29     }
30     // Showing degrees of vertices
31     printf("***** Degrees of vertices *****\n");
32     for (i=1; i<=n; i++){
33         printf("d[%d]=%d\n",i,d[i]);
34     }
35     printf("\n\n");
36     //Deciding existence of an Euler cycle
37     for (i=1; i<=n;i++){
38         if (d[i]%2 != 0){
39             F=1; break;
40         }
41     }
42     if (F==0){  //すべての点の次数が偶数
43         printf("オイラーサイクルは存在します\n\n");
44     }
45     else{  // F==1: 次数が奇数の点が存在
46         printf("オイラーサイクルは存在しません\n\n");
47     }
48     return 0;
49 }
```

図 79　連結グラフにオイラーサイクルが存在する否か判定する C 言語プログラム：左端の行番号は説明用に入れています (プログラムに入れるとエラーになります)

```
Toshimasa-no-MacBook-Pro:hmh-hp watanabe$ gcc -o comp-deg-euler
comp-deg-euler.c
Toshimasa-no-MacBook-Pro:hmh-hp watanabe$ ./comp-deg-euler

 ***** Showing adjacency matrices of graphs ******

***** Adjacency matrix *****
A[1][1]=0  A[1][2]=1  A[1][3]=0  A[1][4]=1  A[1][5]=1  A[1][6]=0
A[2][1]=1  A[2][2]=0  A[2][3]=1  A[2][4]=0  A[2][5]=1  A[2][6]=0
A[3][1]=0  A[3][2]=1  A[3][3]=0  A[3][4]=1  A[3][5]=1  A[3][6]=0
A[4][1]=1  A[4][2]=0  A[4][3]=1  A[4][4]=0  A[4][5]=0  A[4][6]=0
A[5][1]=1  A[5][2]=1  A[5][3]=1  A[5][4]=0  A[5][5]=0  A[5][6]=1
A[6][1]=0  A[6][2]=0  A[6][3]=0  A[6][4]=0  A[6][5]=1  A[6][6]=0
***** Degrees of vertices *****
d[1]=3
d[2]=3
d[3]=3
d[4]=2
d[5]=4
d[6]=1


オイラーサイクルは存在しません
```

図 80　連結グラフにオイラーサイクルが存在する否か判定する C 言語プログラムの実行結果

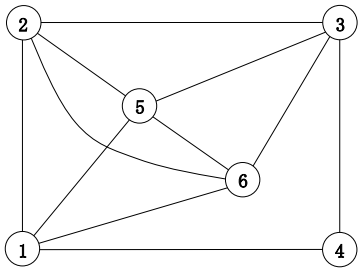


図 81　グラフ $G_5 = (V_5, E_5)$

$G_5$ はどうでしょうか。$G_5$ は連結です。次数ベクトルは $\boldsymbol{d'} = (4, 4, 4, 2, 4, 4,)$ となります。すべての点の次数は偶数です。したがって、$G_5$ はオイラーサイクルを持ちます。このグラフに対して図 79 のプログラムは正しく判定するでしょうか。

- 確かめてみましょう。$G_5$ の隣接行列 $A'$ は以下のようになります。

$$A' = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \end{bmatrix}$$

- 図 79 のプログラムの 10 行目〜12 行目を以下のように書き換えればプログラムはでき上がりです。(行添字 0 と列添字 0 の配列要素はすべて 0 としていることに注意してください。) プログラムの掲載は省略します。

```
// この部分は全てプログラム本体の最初に記述
int A[n+1][m+1]={
     {0,0,0,0,0,0,0},{0,0,1,0,1,1,1},{0,1,0,1,0,1,1},
     {0,0,1,0,1,1,1},{0,1,0,1,0,0,0},{0,1,1,1,0,0,1},
     {0,1,1,1,0,1,0}
};
```

- このプログラムを実行すれば図 82 のような正しい結果が出ます。以上、$G_4$ と $G_5$ に対する検証結果から図 79 のプログラムの正当性を確認したということにしておきます。

```
Toshimasa-no-MacBook-Pro:hmh-hp watanabe$ gcc -o comp-deg-euler-1
comp-deg-euler-1.c
Toshimasa-no-MacBook-Pro:hmh-hp watanabe$ ./comp-deg-euler-1

 ***** Showing adjacency matrices of graphs ******

***** Adjacency matrix *****
A[1][1]=0  A[1][2]=1  A[1][3]=0  A[1][4]=1  A[1][5]=1  A[1][6]=1
A[2][1]=1  A[2][2]=0  A[2][3]=1  A[2][4]=0  A[2][5]=1  A[2][6]=1
A[3][1]=0  A[3][2]=1  A[3][3]=0  A[3][4]=1  A[3][5]=1  A[3][6]=1
A[4][1]=1  A[4][2]=0  A[4][3]=1  A[4][4]=0  A[4][5]=0  A[4][6]=0
A[5][1]=1  A[5][2]=1  A[5][3]=1  A[5][4]=0  A[5][5]=0  A[5][6]=1
A[6][1]=1  A[6][2]=1  A[6][3]=1  A[6][4]=0  A[6][5]=1  A[6][6]=0
***** Degrees of vertices *****
d[1]=4
d[2]=4
d[3]=4
d[4]=2
d[5]=4
d[6]=4


オイラーサイクルは存在します
```

図 82　グラフ $G_5$ に対する図 79 のプログラムの実行結果 (初期データを変更しましたのでプログラム名を comp-deg-euler-result-1.c に変更しています)

### 1.4.2　オイラーサイクル自体を求めること

図 81 の $G_5$ がオイラーサイクルを持つことはわかりました。では、$G_5$ では実際にオイラーサイクルはどのようにして求めるのでしょうか。

- 連結なグラフのオイラーサイクルを求めるアルゴリズムはすでに知られています。ただ残念ながら現段階では皆さんに説明するには少し準備が足りません。点とその接続辺を順に辿ってサイクルを求めて行く効率的な方法があるのですが、ここでその説明をすることは諦めます。
- かわりに、第 4 回で説明したサイクルの拡大を繰り返してオイラーサイクルを求める手法を $G_5$ に適用して説明します。
- 図 81 および図 83〜図 89 を見てください。これらに沿って構成手順を以下に示します。

( **オイラーサイクルの構成手順** )

1. 図 81 の $G_5$ において、1 点 $v$ を選び、$v$ を含むサイクルを求めます。ここでは点 4 を含むサイクル

を求めます。それを図 83 に示すように $C_0$ としましょう。図では説明の都合で有向グラフとして表し、点 4 から有向辺に沿って番号をつけています。あとでサイクル拡大でどのようにサイクルが組み込まれたかを示すためです。以下、有向グラフと無向グラフを厳密に区別しないで扱うとにします。

2. $G_5$ から $C_0$ の辺を削除したグラフ (これを $G_5 - C_0$ と表記) に着目し (図 84 参照)、この中で 2 点以上を持つ連結グラフで $C_0$ 上の点を含むものを選びます。なお、$G_5 - C_0$ の 1 点だけからなる連結グラフは $C_0$ に含まれており、サイクル拡大には無関係ですから対象外としています。ポイントは選んだ (2 点以上を含む) この連結グラフはどれも $C_0$ と共有点を少なくとも 1 つは持つことです。この場合は共有点 1 に着目し、図 84 に示す $H_1$ を選びます。$H_1$ において点 1 を含み $1 \to 5 \to 3 \to 6$ なるサイクル $C_1$ を求めます。図 85 に有向グラフとして辺に番号を付けています。
3. 図 83 の $C_0$ と図 85 の $C_1$ を共有点 1 で結合します (図 86 参照)。オイラーサイクルの番号付けとしては、同図に示しますように、$C_0$ の点 1 でサイクル $C_1$ に入って一周してから再び点 1 から $C_0$ の周回動作を継続する番号付けに変更します。
4. $H_1$ から $C_1$ の辺を削除したグラフ $H_1 - C_1$ において 2 点以上を持つ連結グラフに着目します。このような連結グラフはいずれも $C_1$ と共有点を持ちます。ここでは共有点 5 を含む連結グラフ $H_2$ を選びます (図 87 参照)。
5. 図 87 の $H_2$ において、点 5 を含むサイクル $C_2$ を選びます。ここでは $C_2$ は $H_2$ そのものになりますが、図 88 にサイクル $C_2$ として取り出しています。($H_2 - C_2$ は空グラフですのでサイクル拡大はここが最終ステップです。)
6. 今度は、図 86 のグラフ $C_0 + C_1$ とこの $C_2$ を共有点 5 で結合します (図 89 参照)。$C_0 + C_1 + C_2$ おけるオイラーサイクルの番号付けとしては、同図に示しますように、$C_0 + C_1$ の点 5 でサイクル $C_2$ に入って一周してから再び点 5 から $C_0 + C_1$ の周回動作を継続する番号付けに変更します。
7. 図 89 からわかるように、$G_5$ のオイラーグラフが構成されています。

- 以上が、第 4 回で説明した「( 連結グラフについて ) すべての点の次数が偶数ならばオイラーサイクルが存在する」ことの証明で述べた構成法を $G_5$ に適用した場合の説明です。皆さんの理解を深めることあるいはおさらいに役立つことを期待しています。

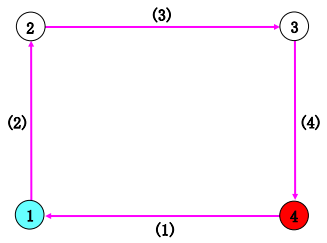


図 83　$G_5$ における点 4 を含むサイクル $C_0$: 説明用に辺の横に番号を付け、有向サイクルとして示しています

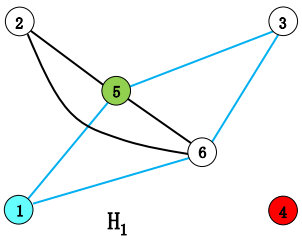


図 84　グラフ $G_5 - C_0$ ($G_5$ から $C_0$ の辺を除去して得るグラフ) における 2 点以上を含む連結グラフ $H_1$: 点 4 は対象外です。

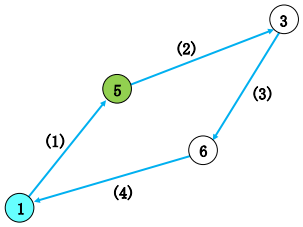


図 85　$C_0$ の点 1 について、$H_1$ における点 1 を含むサイクル $C_1$(有向サイクルとして示しています)

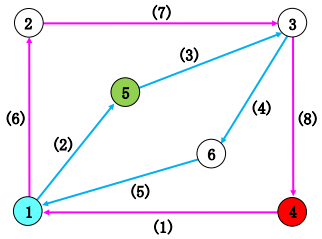


図 86　グラフ $C_0 + C_1$ ($C_0$ と $C_1$ を合わせたグラフ): サイクルの拡大の様子が見えるように辺に番号を付けて有向グラフとして示しています

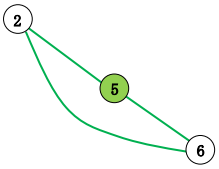


図 87　グラフ $H_2 = H_1 - C_1$

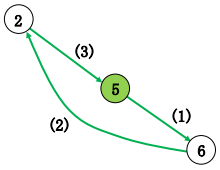


図 88　$C_1$ の点 5 について、$H_2$ において点 5 を含むサイクル $C_2$(実質的に $H_2$ と同じ)

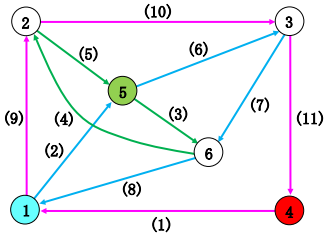


図 89　グラフ $C_0 + C_1 + C_2$: 同様に、サイクル拡大の様子を表すために辺に番号を付けて有向グラフとして示しています

## 1.5　まとめ

第 6 回において作成した、グラフの基点 $s$ から連結な点をすべて求めるアルゴリズム **compo**$(G, s)$ をプログラムとして実現することを目指しています。そのためにはプログラミングについての準備が必要で、第 7 回は

- プログラミングの基礎のそのまた基礎として、コンピュータの仕組、コンピュータ内部でのプログラムの処理過程など

についての大まかな説明をしました。続いて第 8 回ではプログラミングの直接的な基礎となる

- ベクトルや行列をコンピュータのメモリに配列として格納すること
- これらに関連して繰り返し処理を C 言語の for 文で記述すること
- 条件によって操作を選択して実行する処理を C 言語の条件判断文 (if〜else 文) で記述すること

を中心に説明しました。今回はグラフをコンピュータでどのようにして扱うかに焦点を当てました。具体的には

- グラフを隣接行列として表し、隣接行列を整数型 2 次元配列に格納する、という一連の操作

を説明しました。グラフが持つ様々な特徴や情報が 2 次元配列上に引き継がれます。グラフ上で求めたい情報や実行したい操作は 2 次元配列上で情報として求めたり操作として実行することになります。これによりグラフをコンピュータで扱う一つの方法を説明したことになります。

また、これまで説明してきたプログラミングの作り方の演習を兼ねて、第 4 回で取り上げた

- 連結グラフにオイラーサイクルが存在するか否かを判定する問題に対して、存在判定アルゴリズムを C 言語プログラムとして記述すること

にも挑戦してみました。加えて、皆さんの理解を深めることあるいはおさらいの助けになることを期待して、第 4 回の

- 「( 連結グラフについて ) すべての点の次数が偶数ならばオイラーサイクルが存在する」に対する証明で述べたオイラーサイクルの構成法を $G_5$ に適用して構成手順を具体例で示しました。

引き続き、アルゴリズム $\mathbf{compo}(G, s)$ のプログラム作成に進むことも考えましたが、すでに原稿が長くなっていることから次回に移すことにしました。ここまで準備を十分にしてきましたので、次回はスムーズに本題に入っていくことができると思っています。2 点が隣接しているかどうかの判定、基点 $s$ から連結な点を求めること、などに取り組みながら、目標のプログラム作成に入っていくことを考えています。