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 2 月 5 日

第 10 回

1　アルゴリズム compo(G, s) のプログラムを作成しよう

第 6 回で示したアルゴリズム **compo(G, s)** の詳細化およびプログラム化を第 7 回〜第 9 回で行った準備に基づいて説明します。まず、グラフの隣接行列を 2 次元配列に格納すること、基点 s からの距離が k である点の集合 (距離集合) を集合 $N_k\,(k \geq 0)$ として求めて 2 次元配列に格納すること、および s から連結である点のすべてからなる集合 R を 1 次元配列に格納することを説明します。次に、このようなデータの格納方法をベースにして、**compo(G, s)** の Step1〜Step4 の各ステップ、特に Step3 の 3.2〜3.4 をどのようにプログラムとして実現するか (詳細化とプログラム化) を説明します。さらに、**compo(G, s)** には記述していませんが、得られた結果を表示する操作をプログラムに追加します。最後に、これらを 1 つのプログラムとして記述し、図 90 のグラフ G_4 に対する実行結果を示します。また、非連結グラフに対する実行結果も示します。

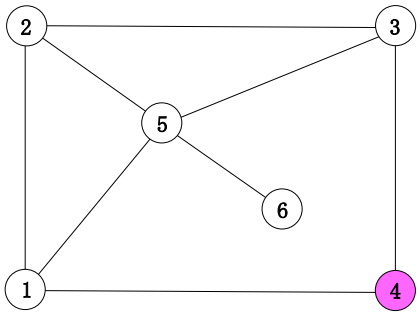

図 90　グラフ $G_4 = (V_4, E_4)$ (点数 6、辺数 8) と基点 $s = 4$

1.1　第 6 回のおさらい

第 6 回を提示してから少し時間が経過していますので、第 6 回の内容の大まかなおさらいをしておきます。

1.1.1　基点 s からの距離と距離集合

グラフ G における基点 s からの距離という数値を導入して、距離 1 の点の集合、距離 2 の点の集合、$\cdots$ 、と点の集合を求めていくことで、グラフ G 上の基点 s から連結な点のすべてを求めるアルゴリズムを考えま

した。

グラフ G 上の異なる 2 点 s,t に対して、s と t を結ぶ 単純なパスのうち辺数の最も少ないものに着目して、その辺数 (すなわち、このパスの長さ) を s と t の**距離** (distance) と呼び、$dist_G(s,t)$ という記号で表すことにします。G がわかっているときは G を省略して単に $dist(s,t)$ と書きます。s と t が非連結なときは $dist(s,t)=\infty$(無限大) とし、$s=t$ のときは $dist(s,t)=0$ とします。なお、この着目した辺数最小のパスを (複数あるときはいずれも) s,t 間の**最短パス** (shortest path) といいます。

G 上で、s からの距離が k の点のすべてからなる集合を

$$A_k=\{v \mid dist(s,v)=k\}$$

と表しますと

$$A_i \cap A_j=\emptyset \ (i \neq j)$$

です。ここでは A_k を (距離 k の) **距離集合**とよぶことにしましょう。

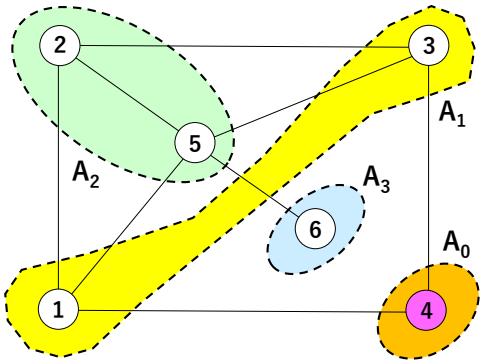

図 91　$G_4=(V_4,E_4)$ における $s=4$ から距離が k である点の集合 A_k $(k=0,1,2,3)$

図 90 のグラフ G_4 について、$s=4$ を基点として A_k ($k\geq 0$ なるすべての整数 k について) を求めると図 91 に示すように

$$A_0=\{4\},\ A_1=\{1,3\},\ A_2=\{2,5\},\ A_3=\{6\},\ A_j=\emptyset \ (j\geq 4)$$

となります。どの辺もある A_k と A_{k+1} を結ぶ形か、またはある A_j 内の 2 点を結ぶ形か、いずれかです。

1.1.2　アルゴリズムのポイント

G 上で s から連結な点すべてを集めた集合を $A_G(s)$ と表しましょう。G がわかっているときは単に $A(s)$ と表すことにします。$s\in A(s)$ です。目標は $A(s)$ を求めるアルゴリズムの設計です。

$A_0=\{s\}$ として、s に隣接する点を集めると A_1 ができます。A_1 の点に隣接する点で A_0, A_1 いずれにも含まれないものを集めれば A_2 ができます。以下同様に $k=3,\cdots$ について、A_{k-1} の点に隣接する点で A_{k-2}, A_{k-1} いずれにも含まれないもの集めれば A_k ができます。$A_0,\cdots,A_m$ と進んで最初に $A_{m+1}=\emptyset$ となった時点で、$A_0,\cdots,A_m$ を集めれば $A(s)$ ができる、ということがアルゴリズムのあらすじです。

A_k の性質として次の命題が成り立ちます。

命題 6.1 A_k $(k\geq 0)$ に対して以下が成り立つ。

(1) $A_k \neq \emptyset$ なる A_k が存在すれば $A_j \neq \emptyset\,(0 \leq j \leq k$ なるすべての j について)
(2) $A_k = \emptyset\,(k \geq 1)$ なる A_k が存在すれば $A_r = \emptyset\,(r \geq k$ なるすべての r について)

さらに、G のすべての点について s からの距離をみたときの最大値 (有限値) を m としますと、$A(s)$ と A_k の関係として以下のことが成り立ちます。

$$A(s) = A_0 \cup \cdots \cup A_m$$

1.1.3 距離集合 A_k $(k \geq 1)$ を構成するアルゴリズム

上述の通り

$$A(s) = A_0 \cup \cdots \cup A_m$$

ですので、$A(s)$ は A_0 から A_1 を求め、A_1 から A_2 を求め、$\cdots$、A_k から A_{k+1} を求め、と続けていき A_m まで求めていけば決まります。そのためのアルゴリズム **compo(G, s)** を以下に示します。

compo(G, s) は、$N_0 \leftarrow \{s\}\,(= A_0)$ と初期設定し、Step3 ((3.1)〜(3.4)) で集合 N_k を構成する操作を $k = 1, 2 \cdots$ と反復します。$k \geq 1$ なる各整数 k について $N_k = A_k$ であり、このようにして A_k が構成できる、というのがアルゴリズムの主張です。その主張が正しいこと (アルゴリズムの正当性) については第 6 回で説明しました。以後、N_k も (距離 k の) 距離集合とよぶことにします。

compo(G, s)

/* G が 1 点 s のみの場合は 1 回目の 3.3 で $N_1 = \emptyset$ となり、2 回目の 3.1 において $N_1 = \emptyset$ で Step4 に進み終了します。したがって、以下では G が 2 点以上を含むとして説明します */

Step1. $k \leftarrow 1$ とする;

Step2. $N_0 \leftarrow \{s\}$, $R \leftarrow N_0$ とする; /* $A_0 = \{s\}$ であり、R は最終的には $A(s)$ を格納します */

Step3.

3.1. $N_{k-1} \neq \emptyset$ である間は、以下の 3.2〜3.4 を実行する; /* $N_{k-1} = \emptyset$ ならば Step4 進む */

3.2. $N_k \leftarrow \emptyset$ とする; /* 各 k について、N_k は A_k に入るべき要素を保持します */

3.3. N_{k-1} のすべての点 $u_1, \cdots, u_r$ に対して、$i = 1, \cdots, r$ の順に、各 u_i について以下の 拡大操作 を行う;

(拡大操作)

- u_i に隣接する点が存在しないときは何もしない;
- u_i に隣接する点が $v_1, \cdots, v_d\,(d \geq 1)$ のとき、$j = 1, \cdots, d$ の順に、各 v_j について以下の (i) または (ii) を実行する;
  - (i) $k = 1$ のとき、v_j を N_k に加える; /* 図 92 参照 */
  - (ii) $k \geq 2$ のとき、$v_j \notin R \cup N_k$ ならば v_j を N_k に加え、$v_j \in R \cup N_k$ ならば何もしない;[*1] /* 図 93 参照 */

3.4. もし $N_k \neq \emptyset$ ならば $R \leftarrow R \cup N_k$ とする。k の値を 1 だけ増やして 3.1 に戻る;

Step4. 終了する; /* $N_{k-1} = \emptyset$ */

[*1] v_j が $R \cup N_k$ に含まれるか否かは、詳細に言えば、v_j が $N_{k-2} \cup N_{k-1} \cup N_k$ に含まれるか否かの判定です。

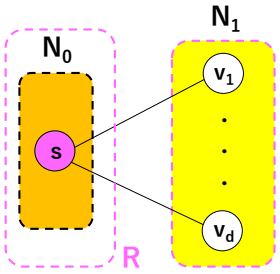

図 92　$N_0 = \{s\}$、および $s\,(= u_1)$ に隣接する点 v_j の集合 N_1；まず $R \leftarrow N_0$ と初期設定し、N_1 が空でない集合として確定したならば $R \leftarrow R \cup N_1$ と更新します

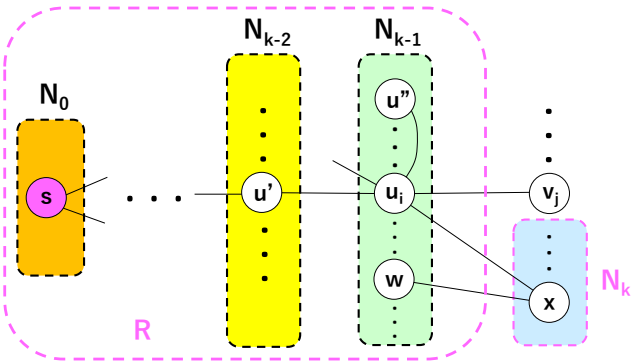

図 93　N_{k-1} の点 u_i に隣接する点 $u' \in N_{k-2} \subset R$, $u\text{”} \in N_{k-1} \subset R$, $x \in N_k$ および $v_j \notin R \cup N_k$ (実質的には、$v_j \notin N_{k-2} \cup N_{k-1} \cup N_k$)；このとき、$N_k \leftarrow N_k \cup \{v_j\}$ と更新し、最終的に N_k が空でない集合として確定したならば $R \leftarrow R \cup N_k$ なる更新もします

N_k, R に着目した **compo(G, s)** の大まかな流れ図を図 94 に示します。これにより、アルゴリズムの流れが明らかになると思います。なお、以下の注意は第 6 回にも示しましたが、再度掲載しておきます。

注意 10.1 (3.1 と 3.4 について) **compo(G, s)** の 3.4 で k の値は 1 だけ増えて 3.1 に戻りますので、3.3 で構成された N_k は 3.1 に戻ったときには N_{k-1} として扱われることに留意してください。このことは特に、3.4 で $N_k \neq \emptyset$ の場合には注意が必要です。このような変数や添字を 1 だけ増やしたり減らしたりする操作はアルゴリズムやプログラムでよく出てきます。記号は同じで持っている値が変化する、ということに少しずつ慣れてほしいと思っています。

1.2　データの配列への格納

グラフの隣接行列 A および距離集合 $N_k\,(k \geq 0)$ を 2 次元配列に格納すること、および s から連結である点のすべてからなる集合 R を 1 次元配列に格納することについて説明します。グラフの点集合を $\{1, \cdots, n\}$ とします。

1.2.1　隣接行列の 2 次元配列への格納

隣接行列 A は $n \times n$ 行列です。これを整数型 2 次元配列 `A[n+1][n+1]` に格納します。以下、`A` の連続する部分 (2 次元配列を行列に例えれば、行 (横方向の並び)) に対して以下のような記号を使います。ここで、

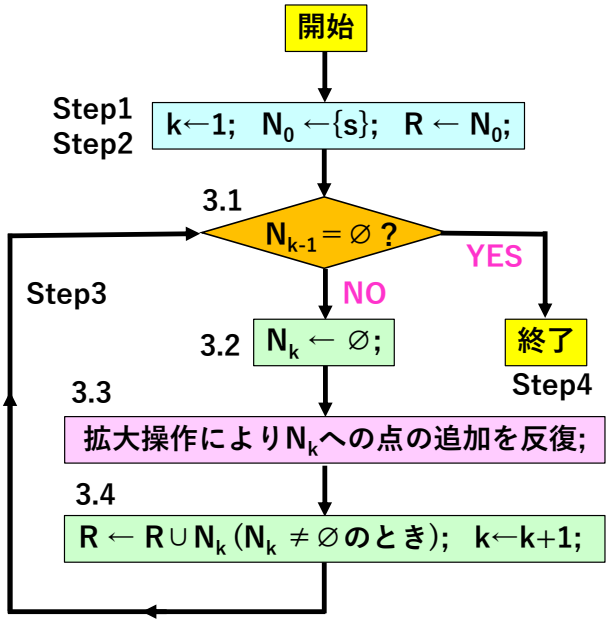

図 94　N_k, R に着目した **compo(G, s)** の流れ図

$0 \le j < k \le n$ です。

$$\texttt{A[i][j..k]} \quad \longleftrightarrow \quad \texttt{A[i][j]}, \cdots, \texttt{A[i][k]}$$

他の配列でも同様の表記をします。

隣接行列は、具体的には図 95 の左側に示しますように、`A[1][1..n]`, $\cdots$, `A[n][1..n]` (図の縦線の右側、横線の下側の部分) に格納します。異なる 2 点 i, j に対して以下の関係があります。

i と j は隣接する $\longleftrightarrow A(i,j) = 1 \longleftrightarrow$ `A[i][j]==1`
i と j は非隣接である $\longleftrightarrow A(i,j) = 0 \longleftrightarrow$ `A[i][j]==0`

配列要素の等号には C 言語の `==` を使っています。さらに `A[i][0]` には点 i の次数 $d(i)$ を格納します ($1 \le i \le n$ なるすべての整数 i について)。なお、`A[0][0..n]` は使いません。図 95 の右側は図 90 のグラフ G_4 の隣接行列に対応しますが、宣言で初期設定したデータを示しています。上端の行と左端の列は初期設定で 0 が入ります。左端の列には点の次数は入っていません。上端の行は 2 次元配列に合わせて用意していますが使用しません。隣接行列がどのように 2 次元配列に格納されるかが具体的に見えると思います。

1.2.2　距離集合の 2 次元配列への格納

基点 s から連結な点の最大距離を $r-1$ とするとき、距離集合は $N_0 \neq \emptyset, \cdots, N_{r-1} \neq \emptyset, N_r = \emptyset$ です。このとき、図 96 の左側に示しますように、距離集合 N_k ($k \ge 0$) を整数型 2 次元配列 `N[r+1][n+1]` に格納します。なお、一般には $r-1$ が初めからわかっているわけではありません。$r-1 \le n-1$ ですので、とりあえず $r = n$ と設定しておけば大丈夫ですが、r が n よりかなり小さいときには不必要に多くのメモリを使ってしまいます。要素の格納については今度は配列の全体を使います。`N[k][1..n]` に距離集合 N_k を格納します。$1 \le i \le n$ について

$$i \in N_k \ \longleftrightarrow\ \texttt{N[k][i]==1} \qquad\qquad i \notin N_k \ \longleftrightarrow\ \texttt{N[k][i]==0}$$

です。N[k][0] には N_k の要素数 $|N_k|$ を格納します ($0 \leq k \leq r-1$ なるすべての整数 k について)。なお、$N_r = \emptyset$ ですので、N[r][0] の値は 0 です。

1.2.3 基点 s から連結な点の 1 次元配列への格納

基点 s から連結である点のすべてからなる集合 R は、整数型 1 次元配列 R[n+1] に格納します。$1 \leq i \leq n$ なるすべての整数 i について

$$i \in R \ \longleftrightarrow\ \texttt{R[i]==1} \qquad\qquad i \notin R \ \longleftrightarrow\ \texttt{R[i]==0}$$

です。R[0] には R の要素数 $|R|$ を格納します。

A[0][0]	A[0][1]	...	A[0][n]
A[1][0]	A[1][1]	...	A[1][n]
⋮	⋮	⋱	⋮
A[i][0]	A[i][1]	...	A[i][n]
⋮	⋮	⋱	⋮
A[n][0]	A[n][1]	...	A[n][n]

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 95　隣接行列の 2 次元配列への格納 (右側は図 90 の G_4 の隣接行列、$n = 6$)

N[0][0]	N[0][1]	...	N[0][n]
N[1][0]	N[1][1]	...	N[1][n]
⋮	⋮	⋱	⋮
N[k][0]	N[k][1]	...	N[k][n]
⋮	⋮	⋱	⋮
N[r][0]	N[r][1]	...	N[r][n]

R[0]	R[1]	...	R[n]

図 96　点 s からの距離が k の距離集合 N_k ($k = 0, \cdots, r$) およびこれらの和集合 R それぞれの 2 次元配列および 1 次元配列への格納

1.3 プログラムの全体の枠組み

まず、**compo(G, s)** のプログラム全体がどのようになるか枠組みを図 97 に示します。**compo(G, s)** のアルゴリズム記述と対比してみればプログラム全体の構造が見えると思います。その後で、各ステップのプログラム化を説明します。

具体的なデータとしては、図 90 のグラフ G_4 の隣接行列、その点数 $n = 6$、最大距離 $r - 1 = 3$、基点 $s = 4$ としています。操作対象のグラフが変われば、この部分を書き換えればいいことになります。なお、宣言はここまでの説明に基づいて記述しています。また、Step1 および Step2 についても記述しています。初期設定といってもいいと思います。Step4 はプログラムの終了です。なお、タイトル表示は無くても差し支えありま

```
// Computing vertices connected to a specified one s in a graph
#include <stdio.h>
#define n 6
#define r 4
#define s 4

int main(void)
{
    int i, j, k, sum;
    int A[n+1][n+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}
    };
    int N[r+1][n+1];
    int R[n+1];

    //タイトル表示

    // 初期化
    for (i=0; i<=r; i++){
        for (j=0; j<=n; j++){
            N[i][j]=0;
        }
    }
    for (i=0; i<=n; i++){
        R[i]=0;
    }

    // *** Step1 ***
    k=1;

    // *** Step2 ***
    N[0][s]=1; N[0][0]=N[0][0]+1;
    R[s]=1; R[0]=R[0]+1;

    // *** Step3  拡大操作の反復 ***
        // 3.1 反復の判定(反復制御)
        // 3.2 N_kの初期化
        // 3.3 拡大操作
        // 3.4 N_kをRに追加；kを1だけ増加

    // ***** 結果の表示 *****
    // 点の次数の計算
    // 隣接行列の表示
    // 距離集合 N_0, ..., N_{r-1}, N_rの表示；
    // 点sと連結な点の集合Rの表示

    // **** Step4 ***
    return 0;
}
```

図 97　**compo(G, s)** のプログラム全体の枠組み

せん。

これから、Step3 以降のコメント文として記述している部分について、具体的にどのようなプログラムとして実現していくかを説明します。「結果の表示」の部分は **compo(G, s)** には記述していませんが、結果をわかりやすく表示することはプログラムには必須と言っていいと思いますので加えています。

1.4 各ステップの詳細化とプログラム化

上述のデータ格納方法に基づいて、**compo(G,s)** における Step3 の 3.2〜3.4 の詳細化とプログラム化、および実行結果のディスプレイへの表示について説明します。

1.4.1 Step3.1 反復の判定 (反復制御)

$N_{k-1} \neq \emptyset$ である間は 3.2〜3.4 の実行を反復する、という枠組みですので、while 文を使います。N_{k-1} の要素は `N[k-1][1..n]` に格納されており、その要素数は `N[k-1][0]` にあります。

$$N_{k-1} \neq \emptyset \quad \longleftrightarrow \quad \texttt{N[k-1][0] > 0}$$

ですので、3.1〜3.4 は以下の形になります。

```
// 3.1 反復の判定（反復制御）
 while (N[k-1][0] > 0){
     // 3.2 N_k の初期化
     // 3.3 拡大操作
     // 3.4 N_k を R に追加；k を 1 だけ増加
 }
```

1.4.2 Step3.2 N_k の初期化

例えば、以下の整数型配列を宣言した場合、

```
int A[n+1][n+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}
};
int N[r+1][n+1];
int R[n+1];
```

`A[n+1][n+1]` の要素は上記の指定した値に初期値が設定されます。一方、`N[r+1][n+1]`、`R[n+1]` のように初期値設定の指定がない整数型の場合にはその配列内のすべての要素は 0 に初期値が設定されるのですが、ここでは以下のように `N` と `R` の要素は明確に初期値をすべて 0 に設定することにします。

```
// 初期化
for (i=0; i<=r; i++){
    for (j=0; j<=n; j++){
        N[i][j]=0;
    }
}
for (i=0; i<=n; i++){
    R[i]=0;
}
```

Step3.2 の $N_k \leftarrow \emptyset$ については、宣言の段階で `N[r+1][n+1]` の値がすべて 0 に設定され、N_k を空集合に初期化することも含んでいます。したがって、Step3.2 については以下の記述をして実質的に何もしません。

```
// 3.2：N[k][0..n] の要素はすべて 0 に初期設定され、空集合に初期化済み
```

1.4.3 Step3.3 拡大操作の反復による N_k の構成

N_{k-1} の点 i について、i に隣接する点 j、あるいは i と非隣接な点 k は隣接行列 A ではそれぞれ $A(i,j)=1$、あるいは $A(i,k)=0$ ですので (図 98 参照)、2 次元配列 `A[n+1][n+1]` 上で見れば以下の関係です。

i が j に隣接するならば `A[i][j]==1`;　i が k に非隣接ならば `A[i][k]==0`

`==` は C 言語の等号です。N_{k-1} の点 i は `N[k-1][i]==1` なる点 i で、`N[k-1][1..n]` の中のこのような点 i すべてについて拡大操作をしますのでまず for 文で次のように書いておきます。

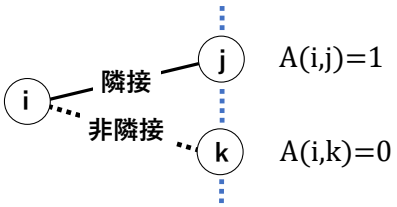

図 98　点 i の隣接点 j と非隣接点 k

```
// 3.3 拡大操作
for (i=1 ;i<=n; i++){
    if (N[k-1][i]==1){
        //点 i に対する拡大操作
    }
}
```

次に、`N[k-1][i]==1` なる点 i に対する拡大操作です。対象となるのは i に隣接する点 j、すなわち `A[i][j]==1` なる点 j です。はじめに、$k=1$ のときはこのような点 j は無条件で N_k に加えます。なお、N_k の要素数を 1 だけ増やす操作も必要です。この要素数は N_k が空集合か否かを迅速に判定するために有用です。したがって、$k=1$ については

```
if ( (k==1 && A[i][j]==1){
    N[k][j]=1;
    N[k][0]=N[k][0]+1;
}
```

となります。一方、$k \geq 2$ のときは `A[i][j]==1` なる点 j を N_k に加えるのはまこの点がまだ $R \cup N_k$ に含まれていない場合です。すなわち、(`(R[j]==0 && N[k][j]==0)`) が成り立つ場合で、この条件を満たす点 j を N_k に加えるという操作になります。したがって、点 j が満たすべき条件を入れて

```
if ( ((k>1 && A[i][j]==1) && (R[j]==0 && N[k][j]==0)) ){
    N[k][j]=1;
    N[k][0]=N[k][0]+1;
}
```

となります。$k=1$, $k>1$ についてどちらかの条件を満たす点 j を N_k に加える操作ですので、これらを「または (C 言語では | なる記号)」で合わせて記述は以下のようになります。

```
if ( (k==1 && A[i][j]==1) | ((k>1 && A[i][j]==1) && (R[j]==0 && N[k][j]==0)) ){
    N[k][j]=1;
    N[k][0]=N[k][0]+1;
}
```

点 i に対する拡大操作は、A[i][1..n] の中で A[i][1] から A[i][n] まで上記の 2 条件のどちらかを満たす点 j を探しながら行います。したがって記述は

```
for (j=1; j<=n; j++){
    if ( (k==1 && A[i][j]==1) | ((k>1 && A[i][j]==1) && (R[j]==0 && N[k][j]==0)) ){
        N[k][j]=1;
        N[k][0]=N[k][0]+1;
    }
}
```

となります。**//点 i に対する拡大操作**の部分にこの記述を入れて拡大操作は以下の記述になります。

```
// 3.3 拡大操作
for (i=1; i<=n; i++){
    if (N[k-1][i]==1){
        for (j=1; j<=n; j++){
            if ((k==1 && A[i][j]==1)|((k>1 && A[i][j]==1)&&(R[j]==0 && N[k][j]==0))){
                N[k][j]=1;
                N[k][0]=N[k][0]+1;
            }
        }
    }
}
```

1.4.4 Step3.4 R の更新と k を 1 だけ増加する操作

N_k が空集合か否かは N[k][0]==0 が成り立つか否かということになります。$N_k \neq \emptyset$ すなわち N[k][0]>0 が成り立つとき、N_k の点 (すなわち N[k][i]==1 なる点 i) を R に加え、同時に R の要素数 R[0] を 1 だけ増やします。これらの操作の反復が終了して k の値を 1 だけ増やします。以上を合わせて以下の記述になります。

```
// 3.4 N_k を R に追加；k を 1 だけ増加
 if (N[k][0] > 0){
     for (i=1; i <=n; i++){
         if (N[k][i]==1){
             R[i]=1;
             R[0]=R[0]+1;
         }
     }
 }
 k=k+1;
```

1.5 実行結果のディスプレイへの表示

compo(G,s) のアルゴリズム記述には含めていませんが、実行結果をわかりやすく表示することはプログラム作成においては重要です。以下では、実行結果の表示形式とそのためのプログラム記述を示します。

1.5.1 点の次数の計算

グラフは単純グラフとしていますので、点 i の次数は隣接点の総数になります。2 次元配列 A[n+1][n+1] で見れば A[i][1..n] の中にある 1 の総数になります。要素は 0 または 1 ですから、次数は A[i][1]〜A[i][n] の和を計算すれば求めることができます。したがって、以下の記述になります。A[i][0] には宣言の初期設定で 0 が入っていることに注意してください。和の計算で A[i][0] の初期値を 0 に設定する記述がないのはそのためです。

```
// 点の次数の計算　（A[i][0] の初期値は 0 です）
 for (i=1; i<=n; i++){
     for (j=1; j<=n; j++){
         A[i][0]=A[i][0]+A[i][j];
     }
 }
```

1.5.2 隣接行列の表示

隣接行列上端の行 A[0][0..n] は表示しません。$i \geq 1$ のとき、A[i][0] には点 i の次数が入っていますので、A[i][0]=(点 i の次数): という表示にしています。その右に続けて A[i][j]==1 なる点 (点 i に隣接する点 j) を表示しています。点 i に隣接する点がすべて表示されると改行し、次の点 $i+1$ に隣接する点の表示に移ります。実際の表示形式は図 106 を参照してください。

表示操作は i, j についての 2 重ループです。j についてのループで「点 i に隣接する点を横方向に表示して最後に改行すること」を行います。これを i についてのループで繰り返しています。具体的な表示操作は以下の記述になります。

```
// 隣接行列の表示
printf("\n***** 隣接行列（左端は各点の次数） *****\n");
for (i=1; i<=n; i++){
    for (j=0; j<=n; j++){
        if (j==0){
            printf("A[%d][%d]=%d: ",i,j,A[i][j]); //次数の表示
        }
        else if (A[i][j]==1){    //隣接点のみ表示
                printf("%d ",j);
        }
    }
    printf("\n");  //i に隣接する点すべてを表示して改行
}
```

1.5.3 距離集合 $N_0, ..., N_{r-1}, N_r$ の表示

$N_i \neq \emptyset (0 \leq i \leq r-1$ なるすべての整数 i について)、および $N_r = \emptyset$ です。$i \geq 0$ に対して、N[i][0] には N_i の要素数が入っていますので、N[i][0]=(要素数): という表示にしています。その右に続けて N[i][j]==1 なる点 (点 N_i に含まれる点 j) を表示しています。点 N_i に含まれる点がすべて表示されると改行し、次の N_{i+1} に含まれる点の表示に移ります。

表示操作は i, j についての 2 重ループです。j についてのループで「N_i に含まれる点を横方向に表示して最後に改行すること」を行います。これを i についてのループで繰り返しています。表示操作は以下の記述になります。なお、N_r は空集合ですので何も表示されません。実際の表示形式は図 106 を参照してください。

```
    // s からの距離が k>=0 の点の集合 N_0, ..., N_{r-1}, N_{r}の表示；
    // N_{r}は空集合です
    printf("\n***** 点 %d からの距離が k>=0 の点の集合 N[0],...,N[%d](左端は要素
数) *****\n", s, r);
    for (i=0; i<=r; i++){
        for (j=0; j<=n; j++){
            if (j==0){
                printf("N[%d][%d]=%d: ", i, j, N[i][j]);  //N_i の点数の表示
            }
            else if (N[i][j]==1){   //N_i に含まれる点の表示
                printf("%d ", j);
            }
        }
        printf("\n");  //N_i に含まれる点すべてを表示して改行
    }
```

1.5.4 点 s と連結な点の集合 R の表示

R[0] には R の要素数が入っていますので、R[0]=(要素数): という表示にしています。その右に続けて R[i]==1 なる点 (点 R に含まれる点 i) のみを表示しています。表示操作は以下の記述になります。実際の表示形式は図 106 を参照してください。

```
// 点 s と連結な点の集合 R の表示
    printf("\n***** 点 %d と連結な点の集合（左端は要素数） *****\n", s);
    for (i=0; i<=n; i++){
        if (i==0){
            printf("R[%d]=%d: ", i, R[i]);  //R の点数の表示
        }
        else if (R[i]==1){  //R の点の表示
            printf("%d ", i);
        }
    }
    printf("\n\n");  //R の点すべてを表示して改行
```

1.6 最終的なプログラムと例題に対する実行結果

ここまで個別に説明してきた記述を統合して、最終的なプログラムの記述および図 90 のグラフ G_4 に対する実行結果を示します。

1.6.1 最終的なプログラム

ここまでの記述を統合した **compo(G, s)** の最終的なプログラム compo.c を図 99〜図 101 に示します。左端の番号は説明用に書いています。プログラムに書くとエラーになります。

```
1   // Computing vertices connected to a specified one s in a graph  compo-1.c
2
3   #include <stdio.h>
4   #define n 6
5   #define r 4
6   #define s 4
7
8   int main(void)
9   {
10      int i, j, k, sum;
11      int A[n+1][n+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      int N[r+1][n+1];
15      int R[n+1];
16
17      printf("\n ***** Computing vertices connected to a specified one
               %d in a graph ******\n",s);
18
```

図 99　textmc**compo(G, s)** のプログラム compo.c (その 1)

1.6.2 図 90 のグラフ G_4 と基点 $s = 4$ に対する距離集合

まず、図 90 のグラフ G_4 と基点 $s = 4$ に対して **compo(G, s)** を実行した場合に、A_0, A_1, A_2, A_3 がそれぞれ N_0, N_1, N_2, N_3 として順次求められていく様子を図 102〜図 105 に示します。図 106 に示すプログラムの実行結果の表示と比較してみるとプログラムが結果をどのように表示しているか見えると思います。

1.6.3 例題に対する実行結果

例題 10.1　図 90 のグラフ G_4 と基点 $s = 4$ に対する **compo(G, s)** の実行結果 (ディスプレイへの表示) が図 106 です。最初の 2 行はコンパイルとリンク、および実行の指示です。それ以下の部分、3 行目以下が実行結果の表示です。

例題 10.2　図 107 に示すグラフ G_6 に対して compo.c を実行してみましょう。compo.c の 2 次元配列 A (隣接行列) の宣言を図 108 に変更するだけですが、別プログラムとして compo-1.c としておきます。$s = 1$

```
19      // 初期化
20      for (i=0; i<=r; i++){
21          for (j=0; j<=n; j++){
22              N[i][j]=0;
23          }
24      }
25      for (i=0; i<=n; i++){
26          R[i]=0;
27      }
28
29      // *** Step1 ***
30      k=1;
31
32      // *** Step2 ***
33      N[0][s]=1; N[0][0]=N[0][0]+1;
34      R[s]=1; R[0]=R[0]+1;
35
36      // *** Step3  拡大操作の反復 ***
37
38      // 3.1  反復の判定（反復制御）
39      while (N[k-1][0] > 0){
40
41          // 3.2：N[k][0..n] の要素はすべて 0 に初期設定され、空集合に初期化済み
42
43          // 3.3 拡大操作
44          for (i=1; i<=n; i++){
45              if (N[k-1][i]==1){
46                  for (j=1; j<=n; j++){
47                      if ( (k==1 && A[i][j]==1) | ((k>1 && A[i][j]==1) &&
                                                    (R[j]==0 && N[k][j]==0)) ){
48                          N[k][j]=1;
49                          N[k][0]=N[k][0]+1;
50                      }
51                  }
52              }
53          }
54
55          // 3.4 N_k を R に追加；k を 1 だけ増加
56          if (N[k][0] > 0){
57              for (i=1; i <= n; i++){
58                  if (R[i]==0 && N[k][i]==1){
59                      R[i]=1;
60                      R[0]=R[0]+1;
61                  }
62              }
63          }
64          k=k+1;
65      }
66
```

図 100　**compo(G, s)** のプログラム compo.c (その 2)

```
67      // ***** 結果の表示 *****
68      // 点の次数の計算
69      for (i=1; i<=n; i++){
70          for (j=1; j<=n; j++){
71              A[i][0]=A[i][0]+A[i][j];
72          }
73      }
74
75      // 隣接行列の表示
76      printf("\n***** 隣接行列（左端は各点の次数） *****\n");
77      for (i=1; i<=n; i++){
78          for (j=0; j<=n; j++){
79              if (j==0){
80                  printf("A[%d][%d]=%d: ",i, j, A[i][j]);  //次数の表示
81              }
82              else if (A[i][j]==1){  //隣接点のみ表示
83                      printf("%d ",j);
84              }
85          }
86          printf("\n");  //i に隣接する点すべてを表示して改行
87      }
88      // s からの距離が k>=0 の点の集合 N_0, ..., N_{r-1}, N_{r}の表示；N_{r}は空集合
90      printf("\n***** 点 %d からの距離が k>=0 の点の集合 N[0],...,N[%d]
                                          （左端は要素数） *****\n", s, r);
91      for (i=0; i<=r; i++){
92          for (j=0; j<=n; j++){
93              if (j==0){
94                  printf("N[%d][%d]=%d: ", i, j, N[i][j]);   //N_i の点数の表示
95              }
96              else if (N[i][j]==1){   //N_i に含まれる点の表示
97                  printf("%d ", j);
98              }
99          }
100         printf("\n");   //N_i に含まれる点すべてを表示して改行
101     }
102
103     // 点 s と連結な点の集合 R の表示
104     printf("\n***** 点 %d と連結な点の集合（左端は要素数） *****\n", s);
105     for (i=0; i<=n; i++){
106         if (i==0){
107             printf("R[%d]=%d: ", i, R[i]);   //R の点数の表示
108         }
109         else if (R[i]==1){   //R に含まれる点の表示
110             printf("%d ", i);
111         }
112     }
113     printf("\n\n");   //R に含まれる点すべてを表示して改行
114
115     // **** Step4 ***
116     return 0;
117 }
```

図 101　**compo(G, s)** のプログラム compo.c (その 3)

図 102　Step25 終了時の状況 (R は省略)

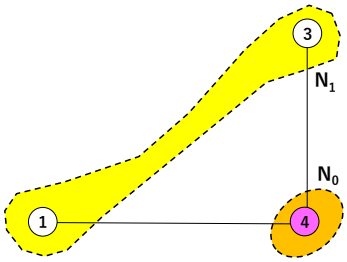

図 103　1 回目の 3.3 終了時の状況 (R は省略)

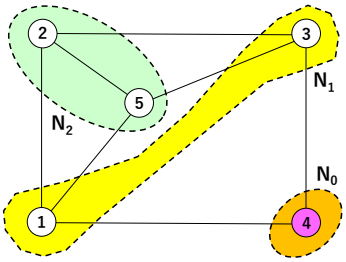

図 104　2 回目の 3.3 終了時の状況 (R は省略)

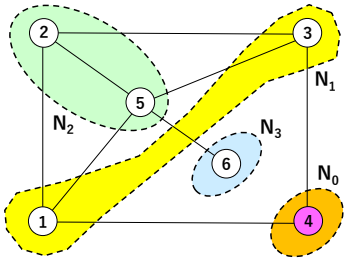

図 105　3 回目の 3.3 終了時の状況 (R は省略)

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

 ***** Computing vertices connected to a specified one 4 in a graph ******

***** 隣接行列（左端は各点の次数） *****
A[1][0]=3: 2 4 5
A[2][0]=3: 1 3 5
A[3][0]=3: 2 4 5
A[4][0]=2: 1 3
A[5][0]=4: 1 2 3 6
A[6][0]=1: 5

***** 点 4 からの距離が k>=0 の点の集合 N[0],...,N[4]（左端は要素数） *****
N[0][0]=1: 4
N[1][0]=2: 1 3
N[2][0]=2: 2 5
N[3][0]=1: 6
N[4][0]=0:

***** 点 4 と連結な点の集合（左端は要素数） *****
R[0]=6: 1 2 3 4 5 6
```

図 106　G_4 と $s = 4$ に対する compo.c の実行結果

の場合および $s = 6$ の場合の compo-1.c の実行結果をそれぞれ図 109 および図 110 に示します。

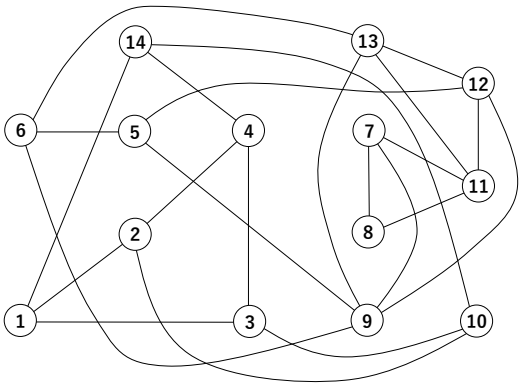

図 107　グラフ G_6 (点数 14、辺数 23)

```
int A[n+1][n+1]={
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0},{0,0,1,1,0,0,0,0,0,0,0,0,0,0,1},
    {0,1,0,0,1,0,0,0,0,0,1,0,0,0,0},{0,1,0,0,1,0,0,0,0,0,1,0,0,0,0},
    {0,0,1,1,0,0,0,0,0,0,0,0,0,0,1},{0,0,0,0,0,0,1,0,0,1,0,0,1,0,0},
    {0,0,0,0,0,1,0,0,0,1,0,0,0,1,0},{0,0,0,0,0,0,0,0,1,1,0,1,0,0,0},
    {0,0,0,0,0,0,0,1,0,0,0,1,0,0,0},{0,0,0,0,0,0,1,1,1,0,0,0,0,1,1,0},
    {0,0,1,1,0,0,0,0,0,0,0,0,0,0,1},{0,0,0,0,0,0,0,0,1,1,0,0,0,1,1,0},
    {0,0,0,0,0,1,0,0,0,1,0,1,0,1,0},{0,0,0,0,0,0,0,1,0,0,1,0,1,1,0,0},
    {0,1,0,0,1,0,0,0,0,0,1,0,0,0,0}
};
```

図 108　compo.c におけるグラフデータ (隣接行列) の変更部分：プログラム名 compo-1.c

図 111 および図 112 からわかりますように、G_6 は連結ではなく、連結な部分グラフ (**連結成分** (connected component) とよばれます)[*2] を 2 つ持ちます。compo-1.c は $s = 1$ を含む連結成分を求めた後で、含まれなかった点を基点 (ここでは $s = 6$) としてプログラムを再実行しています。自動的にすべての連結成分を求めるように compo-1.c を拡張するのは簡単です。皆さんの演習問題にしてください。

1.7　まとめ

第 6 回で示したアルゴリズム **compo(G, s)** の詳細化およびそれに合わせたプログラム化を第 7 回〜第 9 回で行った準備に基づいて説明しました。

具体的には、グラフの隣接行列 A および集合 $N_k\,(k \geq 0)$ として求めた距離集合 A_k それぞれを 2 次元配列に格納すること、および s から連結である点のすべてからなる集合 R を 1 次元配列に格納することをベースにして、**compo(G, s)** の Step1〜Step4 のプログラム化を説明しました。さらに、**compo(G, s)** には記述していませんが、得られた結果を表示する操作をプログラムに追加しました。最後に、これらを 1 つのプログラムとして記述し、図 90 の連結グラフ G_4 に対する実行結果を示しました。別の実例として、非連結グラフ G_6 について $s = 1$ および $= 6$ に対する実行結果も示しました。

[*2] 正確には、連結な点をすべて集めてできる連結グラフです。

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

 ***** Computing vertices connected to a specified one 1 in a graph ******

***** 隣接行列（左端は各点の次数） *****
A[1][0]=3: 2 3 14
A[2][0]=3: 1 4 10
A[3][0]=3: 1 4 10
A[4][0]=3: 2 3 14
A[5][0]=3: 6 9 12
A[6][0]=3: 5 9 13
A[7][0]=3: 8 9 11
A[8][0]=2: 7 11
A[9][0]=5: 5 6 7 12 13
A[10][0]=3: 2 3 14
A[11][0]=4: 7 8 12 13
A[12][0]=4: 5 9 11 13
A[13][0]=4: 6 9 11 12
A[14][0]=3: 1 4 10

***** 点 1 からの距離が k>=0 の点の集合 N[0],...,N[4]（左端は要素数） *****
N[0][0]=1: 1
N[1][0]=3: 2 3 14
N[2][0]=2: 4 10
N[3][0]=0:
N[4][0]=0:

***** 点 1 と連結な点の集合（左端は要素数） *****
R[0]=6: 1 2 3 4 10 14
```

図 109　G_6 と $s = 1$ に対するプログラム compo-1.c の実行結果

これで、第 6 回で提示し、第 7 回〜第 9 回で準備したアルゴリズム **compo(G, s)** のプログラム作成が終了しました。これらの説明を通して、アルゴリズム設計とデータ構造 (データの格納方法)、アルゴリムの詳細化とプログラム化、コーディング、デバッギングと実行、など一連の流れを説明したことになります。

プログラムに関しては、とにかく動くプログラムを作ることに重点を置きました。プログラムとしては、より洗練した形で読みやすく、また改良や拡張などの今後の扱いが容易な形であることが望ましい姿です。あまり専門的な事項に深入りすることなくこれまで説明してきた事項を使ってできる範囲内で、今回作成したプログラムを対象にしてこのことに挑戦してみるつもりです。

今の予定では、関数という仕組みの説明をして、今回のプログラムに関数を取り込むことで全体の筋道がよく見える理解しやすい記述にすることを考えています。

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

 ***** Computing vertices connected to a specified one 6 in a graph ******

***** 隣接行列（左端は各点の次数） *****
A[1][0]=3: 2 3 14
A[2][0]=3: 1 4 10
A[3][0]=3: 1 4 10
A[4][0]=3: 2 3 14
A[5][0]=3: 6 9 12
A[6][0]=3: 5 9 13
A[7][0]=3: 8 9 11
A[8][0]=2: 7 11
A[9][0]=5: 5 6 7 12 13
A[10][0]=3: 2 3 14
A[11][0]=4: 7 8 12 13
A[12][0]=4: 5 9 11 13
A[13][0]=4: 6 9 11 12
A[14][0]=3: 1 4 10

***** 点 6 からの距離が k>=0 の点の集合 N[0],...,N[4]（左端は要素数） *****
N[0][0]=1: 6
N[1][0]=3: 5 9 13
N[2][0]=3: 7 11 12
N[3][0]=1: 8
N[4][0]=0:

***** 点 6 と連結な点の集合（左端は要素数） *****
R[0]=8: 5 6 7 8 9 11 12 13
```

図 110　G_6 と $s = 6$ に対するプログラム compo-1.c の実行結果

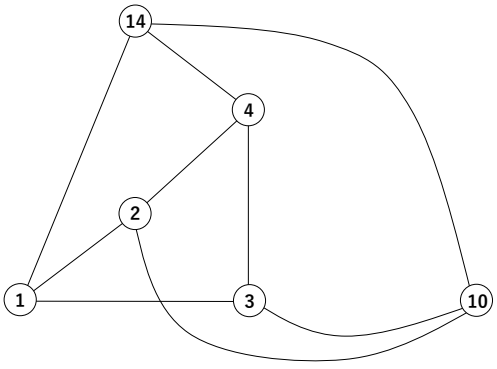

図 111　G_6 における $s = 1$ を含む連結成分

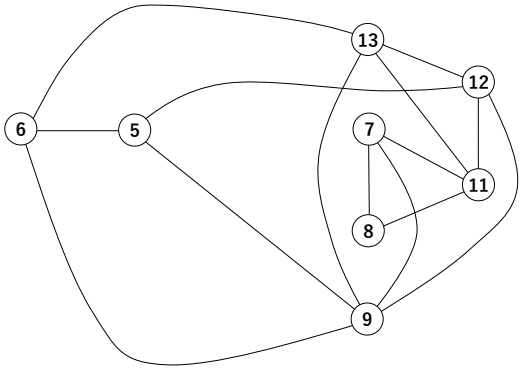

図 112　G_6 における $s = 6$ を含む連結成分