データ構造とアルゴリズム(第10回)

グラフのアルゴリズム(1)

第6章　グラフアルゴリズム

- 6.1 グラフの利用
- 6.2 グラフの表現
- 6.3 用語の定義
- 6.4 グラフの探索
- 6.5 最短経路問題
- 6.6 ネットワークフロー

第6章　グラフアルゴリズム

- 6.1 グラフの利用
- 6.2 グラフの表現
- 6.3 用語の定義 → 自習(付録参照)
- 6.4 グラフの探索
- 6.5 最短経路問題
- 6.6 ネットワークフロー

第6章　グラフアルゴリズム

- 6.1 グラフの利用
- 6.2 グラフの表現
- 6.3 用語の定義 → 自習(付録参照)
- 6.4 グラフの探索
- 6.5 最短経路問題
- 6.6 ネットワークフロー

第10週

第6章　グラフアルゴリズム

- 6.1 グラフの利用
- 6.2 グラフの表現
- 6.3 用語の定義 → 自習(付録参照)
- 6.4 グラフの探索
- 6.5 最短経路問題
- 6.6 ネットワークフロー

第10週

第11週

第6章　グラフアルゴリズム

- 6.1 グラフの利用
- 6.2 グラフの表現
- 6.3 用語の定義 → 自習(付録参照)
- 6.4 グラフの探索
- 6.5 最短経路問題
- 6.6 ネットワークフロー

第10週

第11週

第12週

この章の学習目標

- 隣接行列，隣接リストとその特徴を説明できる
- グラフアルゴリズムを実行例を示しながら説明できる
  - 幅優先探索，深さ優先探索，最短経路，最大フロー
- 上記アルゴリズムの(漸近的)計算時間を説明できる

グラフによる表現の例

- グラフは「ものとものとのつながり」を表現する数学的構造
- 道路網・通信路
- 回路
- ソーシャルネットワーク

などなど，応用は山ほど存在

グラフの表現

- グラフ$G=(V,E)$ ($|V|=n$, $|E|=m$とする)を計算機上で表現するには？
- 2つの表現(データ構造)
  - 隣接行列(Adjacency matrix)：2次元配列を利用
  - 隣接リスト(Adjacency list)：リスト配列を利用

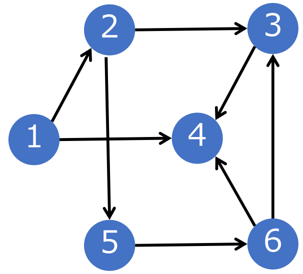
    (リスト配列:各要素がリストであるような配列)

それぞれ詳しく説明

隣接行列（有向グラフ）

- $n\times n$正方行列(=2次元配列)で表現
  - 有向辺$i\to j$が存在 $\Leftrightarrow$ i,j成分が1

$$
\begin{array}{c c}
 & \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} &
\begin{pmatrix}
0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 1 & 0 & 0
\end{pmatrix}
\end{array}
$$

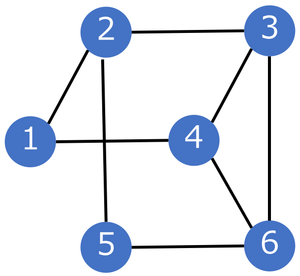


- メモリ使用量$O(n^2)$ビット

隣接行列（無向グラフ）

- 無向グラフのとき
  - 辺(i,j)が存在：有向辺$i\to j$と$j\to i$が両方あると思う

$$
\begin{array}{c c}
 & \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} &
\begin{pmatrix}
0 & 1 & 0 & 1 & 1 & 0 \\
1 & 0 & 1 & 1 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 & 1 \\
1 & 1 & 1 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 & 1 & 0
\end{pmatrix}
\end{array}
$$

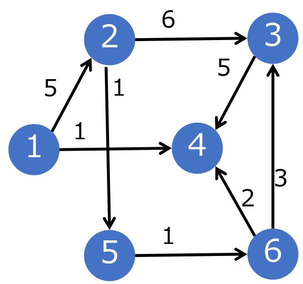


  - 行列は対称になる(メモリ量は同様に$O(n^2)$ビット)
    - 情報量的には半分は無駄だが，あえてこうするのが普通

隣接行列（重み付きグラフ）

- 辺に重みがついているときは，行列の各成分に重みを記載する

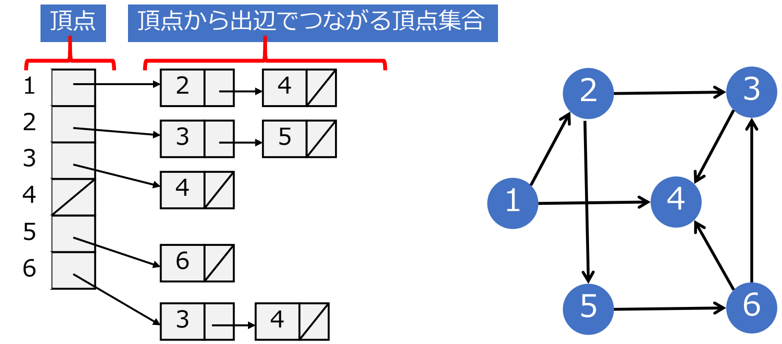
$$
\begin{array}{c c}
 & \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} &
\begin{pmatrix}
0 & 5 & 0 & 1 & 0 & 0 \\
0 & 0 & 6 & 0 & 1 & 0 \\
0 & 0 & 0 & 5 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 3 & 2 & 0 & 0
\end{pmatrix}
\end{array}
$$



- メモリ使用量$O(n^2\log M)$ビット(Mは重みの最大値)
  - Mを定数と思うと$O(n^2)$ビット

隣接リスト（有向グラフ）

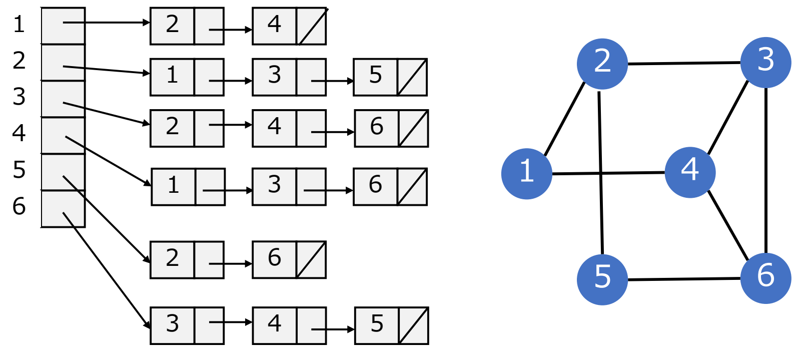
- 各頂点の接続辺のリストからなる配列で表現



- メモリ使用量$O((n+m)\log n)$ビット

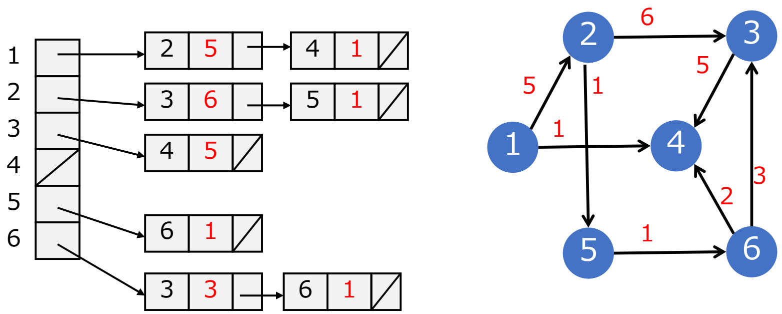
隣接リスト（無向グラフ）

- 隣接行列のときと同様に，双方向辺として扱う



隣接リスト（重み付きグラフ）

- 辺の重みはリストの中に記載する
  (レコードの要素として(相手の頂点，辺重み)の対を記録する)



どちらを使う？

- $m = O(n)$: 疎なグラフ(sparse graph)
  - 通常隣接リストを使う
  - 隣接行列ではスペースの無駄が多すぎる
- $m = \omega(n)$ かつ $o(n^2)$ ：密でないグラフ/疎でないグラフ
- $m = \Omega(n^2)$: 密なグラフ(dense graph)
  - 場合による（必要な操作に応じて使い分ける）

実世界におけるグラフデータは疎なグラフであることが多く
利用頻度でいうと隣接リストのほうが高い

クイズ

- 次の操作はそれぞれ隣接行列，隣接リストで処理するときどの程度時間がかかるか？
  1. 与えられた頂点vの隣接頂点をすべてチェックする
  2. 与えられた辺(i, j)について逆辺(j, i)が存在するかどうかチェックする
  3. 重み付きグラフで，辺(i, j)の重みを変更する
  4. 辺(i, j)を追加する

クイズ

δ_v:頂点vの次数

- 答え
  1. 与えられた頂点vの隣接頂点をすべてチェックする
     - 隣接リスト：$O(\delta_v)$　/　隣接行列：$O(n)$
  2. 与えられた辺(i, j)について逆辺(j, i)が存在するかどうかチェックする
     - 隣接リスト：$O(\delta_j)$ / 隣接行列：$O(1)$
  3. 重み付きグラフで，辺(i, j)の重みを変更する
     - 隣接リスト：$O(\delta_i)$ / 隣接行列：$O(1)$
  4. 辺(i, j)を追加する
     - 隣接リスト：$O(1)$ / 隣接行列：$O(1)$

余談：

- 最近のモダンなプログラミング言語を用いる場合，隣接リストのリスト部分は，可変長配列を利用することが多い
- 可変長配列を用いて，かつリスト中の頂点番号をちゃんとソートすることを前提とすれば，計算量は以下のようになる
  - (i, j)に対して(j, i)の存在チェック: $O(\log \delta_j)$
  - (i, j)の重みの変更: $O(\log \delta_i)$
- その代わり，辺(i, j)の追加/削除は$\Omega(\delta_i)$必要
  - 頻繁にグラフを変形させるときは効率悪い

幅優先探索

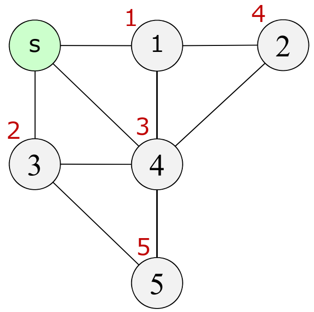
グラフの探索

- グラフ中の頂点を（何らかの規則に従った順序で）順次チェックする

幅優先探索(Breadth First Search: BFS)
- 始点から探索できるものすべて探索(近い方優先)
- キューを使う

深さ優先探索(Depth First Search: DFS)
- 始点からの距離(遠い方) を優先して 探索
- スタックを使う

幅優先探索 (BFS)



探索順序のイメージ

1. s をスタート地点とする
2. s から辿れる頂点をすべて辿る
3. 2. で辿った頂点から一つずつ順番に頂点を選び，それぞれ辿れる頂点を全て辿る.
4. 3. の動作を繰り返す.

探索順序の例

s → 1 → 3 → 4 → 2 → 5

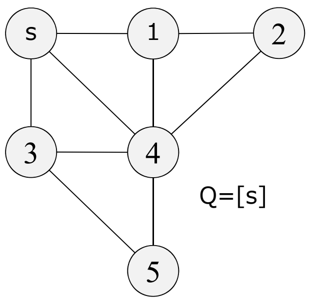
スタート!

s から辿れる頂点 (s の隣接頂点)

1 から辿れる頂点 (1 の隣接頂点のうち未探索のもの)

3 から辿れる頂点 (3 の隣接頂点のうち未探索のもの)

BFSアルゴリズム(1)



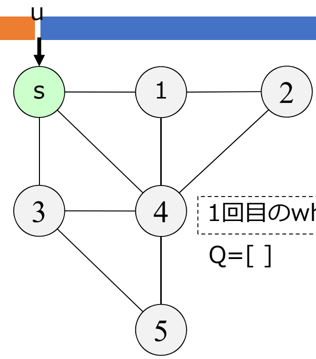
Q=[s]

未訪問

既訪問

```
アルゴリズム 6.1　BFS
入力：G=(V,E)，始点 s∈V
for (各頂点 v∈V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
キュー Q=[s] とする；
while( Qが空でない) {
          Qから頂点 u を取り出す；
          for (v∈adj(u)){
              if (vが未訪問) {
                 v を既訪問にする；
                   v をQに入れる；
              }
          }
}
```

BFSアルゴリズム(2)



1回目のwhile開始

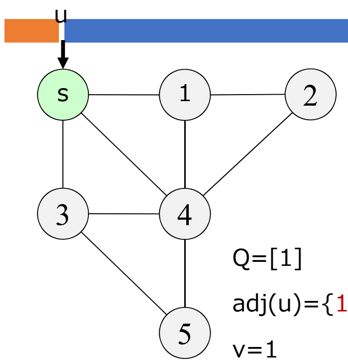
Q=[ ]

未訪問

既訪問

```
アルゴリズム 6.1　BFS
入力：G=(V,E)，始点 s∈V
for (各頂点 v∈V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
キュー Q=[s] とする；
while( Qが空でない) {
          Qから頂点 u を取り出す；
          for (v∈adj(u)){
              if (vが未訪問) {
                 v を既訪問にする；
                   v をQに入れる；
              }
          }
}
```

BFSアルゴリズム(3)



Q=[1]

adj(u)={1, 3, 4}

v=1

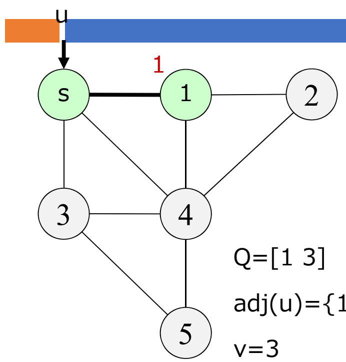
未訪問

既訪問

```
アルゴリズム 6.1　BFS
入力：G=(V,E),  始点 s∈V
for (各頂点 v∈V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
キュー Q=[s] とする；
while( Qが空でない) {
            Qから頂点 u を取り出す；
            for (v∈adj(u)){
                if (vが未訪問) {
                   v を既訪問にする；
                     v をQに入れる；
                }
            }
}
```

u の隣接頂点集合を返す

BFSアルゴリズム(4)



Q=[1 3]

adj(u)={1, 3, 4}

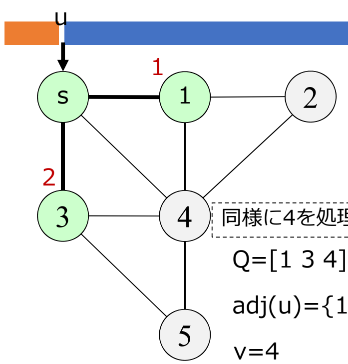
v=3

未訪問

既訪問

```
アルゴリズム 6.1　BFS
入力：G=(V,E),  始点 s∈V
for (各頂点 v∈V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
キュー Q=[s] とする；
while( Qが空でない) {
            Qから頂点 u を取り出す；
            for (v∈adj(u)){
                if (vが未訪問) {
                   v を既訪問にする；
                     v をQに入れる；
                }
            }
}
```

BFSアルゴリズム(5)



同様に4を処理して…

Q=[1 3 4]

adj(u)={1, 3, 4}

v=4

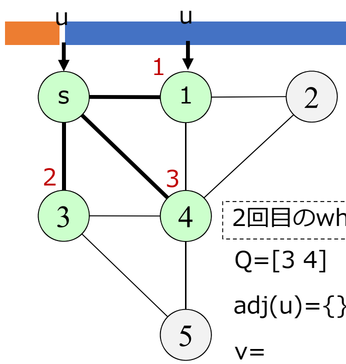
未訪問

既訪問

1回目の
while 終了

```
アルゴリズム 6.1　BFS
入力：G=(V,E),  始点 s∈V
for (各頂点 v∈V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
キュー Q=[s] とする；
while( Qが空でない) {
            Qから頂点 u を取り出す；
            for (v∈adj(u)){
                if (vが未訪問) {
                   v を既訪問にする；
                     v をQに入れる；
                }
            }
}
```

BFSアルゴリズム(6)



2回目のwhile開始

Q=[3 4]

adj(u)={}

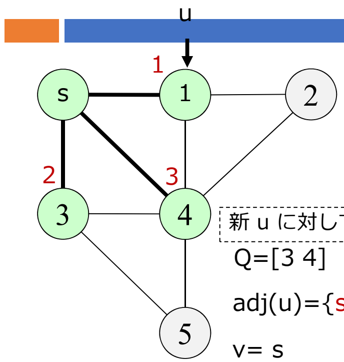
v=

未訪問

既訪問

```
アルゴリズム 6.1　BFS
入力：G=(V,E),  始点 s∈V
for (各頂点 v∈V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
キュー Q=[s] とする；
while( Qが空でない) {
            Qから頂点 u を取り出す；
            for (v∈adj(u)){
                if (vが未訪問) {
                   v を既訪問にする；
                     v をQに入れる；
                }
            }
}
```

BFSアルゴリズム(7)



新 u に対して同じ処理

Q=[3 4]

adj(u)={s, 2, 4}

v= s

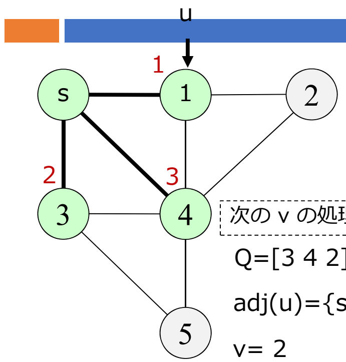
sは既訪問なので
何もしない

未訪問

既訪問

```
アルゴリズム 6.1　BFS
入力：G=(V,E), 始点 s∊V
for (各頂点 v∊V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
キュー Q=[s] とする；
while( Qが空でない) {
            Qから頂点 u を取り出す；
            for (v∊adj(u)){
                if (vが未訪問) {
                   v を既訪問にする；
                     v をQに入れる；
                }
            }
}
```

BFSアルゴリズム(8)



次の v の処理

Q=[3 4 2]

adj(u)={s, 2, 4}

v= 2

v=4 も既訪問なので何もしない

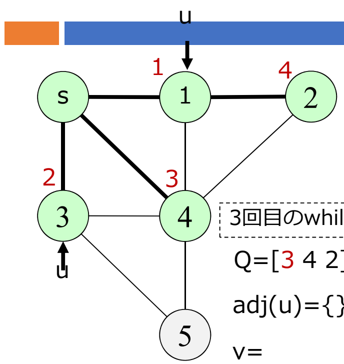
2回目の
while 終了

未訪問

既訪問


```
アルゴリズム 6.1　BFS
入力：G=(V,E), 始点 s∊V
for (各頂点 v∊V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
キュー Q=[s] とする；
while( Qが空でない) {
            Qから頂点 u を取り出す；
            for (v∊adj(u)){
                if (vが未訪問) {
                   v を既訪問にする；
                     v をQに入れる；
                }
            }
}
```


BFSアルゴリズム(9)



3回目のwhile開始

Q=[3 4 2]

adj(u)={}

v=

Qから3が取り出され
u は 3 になる
adj(u)={s, 4, 5} と
なるs, 4 は既訪問で
何もせず
5 は未訪問なので処理
5 を Q に入れる

Q=[4 2 5]

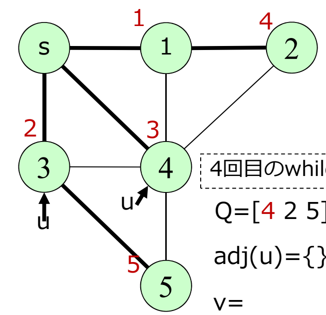
adj(u)={s, 4, 5}

v= 5

3回目のwhile 終了


```
アルゴリズム 6.1　BFS
入力：G=(V,E), 始点 s∊V
for (各頂点 v∊V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
キュー Q=[s] とする；
while( Qが空でない) {
            Qから頂点 u を取り出す；
            for (v∊adj(u)){
                if (vが未訪問) {
                   v を既訪問にする；
                     v をQに入れる；
                }
            }
}
```


BFSアルゴリズム(10)



4回目のwhile開始

Q=[4 2 5]

adj(u)={}

v=

Qから4が取り出され
u は 4 になる
adj(u)={s,1,2,3,5} と
なるs,1,2,3,5 は
全て既訪問
何もせずに終了

Q=[2 5]

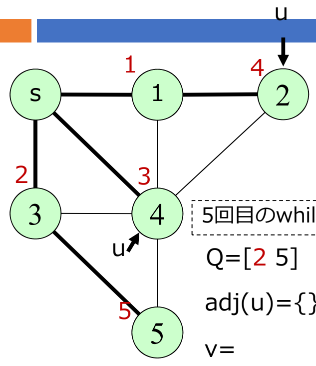
adj(u)={s, 1, 2, 3, 5}

v= 5

4回目のwhile 終了


```
アルゴリズム 6.1　BFS
入力：G=(V,E), 始点 s∊V
for (各頂点 v∊V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
キュー Q=[s] とする；
while( Qが空でない) {
            Qから頂点 u を取り出す；
            for (v∊adj(u)){
                if (vが未訪問) {
                   v を既訪問にする；
                     v をQに入れる；
                }
            }
}
```

BFSアルゴリズム(11)



5回目のwhile開始

Q=[2 5]

adj(u)={}

v=

Q=[5]

adj(u)={1, 4}

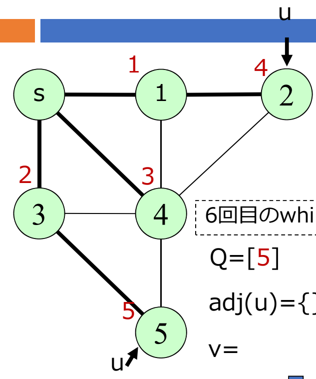
v= 4

```
アルゴリズム 6.1　BFS
入力：G=(V,E)，始点 s∊V
for (各頂点 v∊V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
キュー Q=[s] とする；
while( Qが空でない) {
            Qから頂点 u を取り出す；
            for (v∊adj(u)){
                if (vが未訪問) {
                   v を既訪問にする；
                     v をQに入れる；
                }
            }
}
```

Qから2が取り出され
u は 2 になる
adj(u)={1,4} と
なるが全て既訪問
何もせずに終了

5回目のwhile 終了

BFSアルゴリズム(12)



6回目のwhile開始

Q=[5]

adj(u)={}

v=

Q=[]

adj(u)={3, 4}

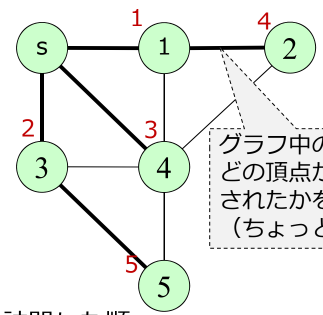
v= 4

```
アルゴリズム 6.1　BFS
入力：G=(V,E)，始点 s∊V
for (各頂点 v∊V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
キュー Q=[s] とする；
while( Qが空でない) {
            Qから頂点 u を取り出す；
            for (v∊adj(u)){
                if (vが未訪問) {
                   v を既訪問にする；
                     v をQに入れる；
                }
            }
}
```

Qから5が取り出され
u は 5 になる
adj(u)={3,4} と
なるが全て既訪問
何もせずに終了

アルゴリズム終了

BFSアルゴリズム結果



グラフ中の太線は，
どの頂点から訪問
されたかを示す
（ちょっと大事）

```
アルゴリズム 6.1　BFS
入力：G=(V,E)，始点 s∊V
for (各頂点 v∊V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
キュー Q=[s] とする；
while( Qが空でない) {
            Qから頂点 u を取り出す；
            for (v∊adj(u)){
                if (vが未訪問) {
                   v を既訪問にする；
                     v をQに入れる；
                }
            }
}
```

訪問した順

s → 1 → 3 → 4 → 2 → 5

注意：幅優先探索による訪問順は一意には
決まらない．adj() からの頂点の取り出し順に依存

BFSアルゴリズムの実行時間

(隣接リストを用いた場合)

```
アルゴリズム 6.1　BFS
入力：G=(V,E)，始点 s∊V
for (各頂点 v∊V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
キュー Q=[s] とする；
while( Qが空でない) {
            Qから頂点 u を取り出す；
            for (v∊adj(u)){
                if (vが未訪問) {
                   v を既訪問にする；
                     v をQに入れる；
                }
            }
}
```

$O(n)$

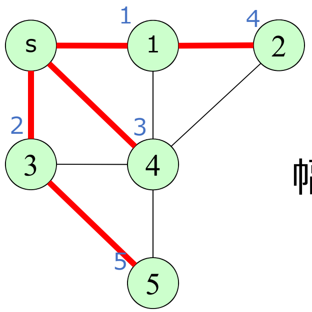
実行全体を通して考えると…

- キューの各操作= $O(1)$
- 内側ループの総回数 =次数の総和=$O(m)$
- 各頂点は Q に1度入れられる (= $O(n)$)

合計 $O(m+n)$

幅優先木

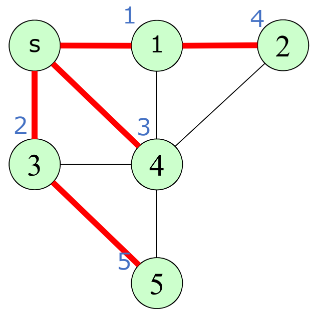
- BFSにおいて(訪問元,訪問先)の辺をすべて集めたものは(sを根とする)全域木になる
  - n-1頂点の連結グラフなので



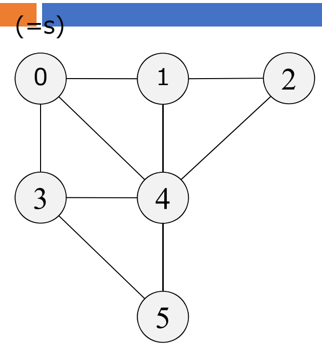
幅優先木(BFS木) と呼ぶ

幅優先探索の応用

- BFS木は「始点から各頂点への最短経路」を与える
  - 重みなしグラフの最短経路発見に使える！



幅優先木の構成と記録



T

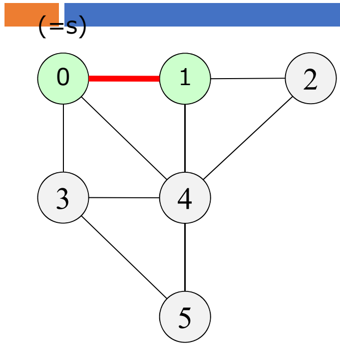
0					

未訪問

既訪問

```
アルゴリズム 6.1(改) BFS木
入力：G=(V,E), 始点 s∊V
T[0, n-1]：幅優先木を記録する配列
T[s] = s;
for (各頂点 v∊V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
キュー Q=[s] とする；
while( Qが空でない) {
          Qから頂点 u を取り出す；
          for (v∊adj(u)){
              if (vが未訪問) {
                     T[v] = u ;
                 v を既訪問にする；
                   v をQに入れる；
              }
          }
}
```

幅優先木の構成と記録



T

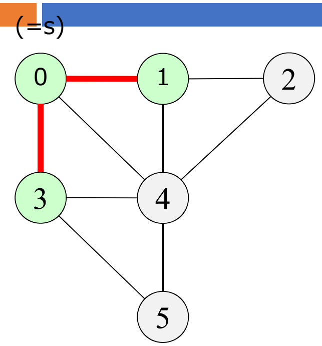
0	**0**				

未訪問

既訪問

```
アルゴリズム 6.1(改) BFS木
入力：G=(V,E), 始点 s∊V
T[0, n-1]：幅優先木を記録する配列
T[s] = s;
for (各頂点 v∊V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
キュー Q=[s] とする；
while( Qが空でない) {
          Qから頂点 u を取り出す；
          for (v∊adj(u)){
              if (vが未訪問) {
                     T[v] = u ;
                 v を既訪問にする；
                   v をQに入れる；
              }
          }
}
```

幅優先木の構成と記録



T

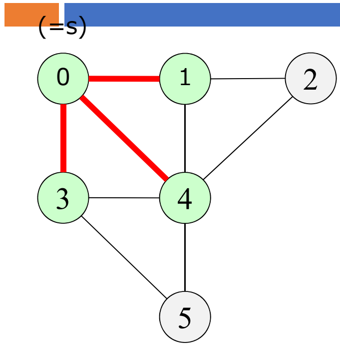
0	0		0		

未訪問

既訪問

```
アルゴリズム 6.1(改) BFS木
入力：G=(V,E), 始点 s∈V
T[0, n-1]：幅優先木を記録する配列
T[s] = s:
for (各頂点 v∈V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
キュー Q=[s] とする；
while( Qが空でない) {
          Qから頂点 u を取り出す；
          for (v∈adj(u)){
              if (vが未訪問) {
                     T[v] = u；
                 v を既訪問にする；
                   v をQに入れる；
               }
          }
}
```

幅優先木の構成と記録



T

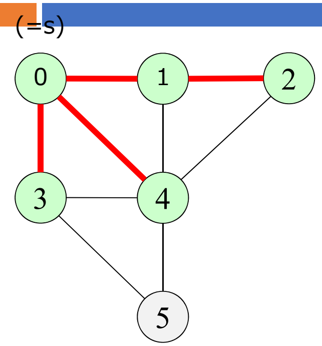
0	0		0	0	

未訪問

既訪問

```
アルゴリズム 6.1(改) BFS木
入力：G=(V,E), 始点 s∈V
T[0, n-1]：幅優先木を記録する配列
T[s] = s:
for (各頂点 v∈V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
キュー Q=[s] とする；
while( Qが空でない) {
          Qから頂点 u を取り出す；
          for (v∈adj(u)){
              if (vが未訪問) {
                     T[v] = u；
                 v を既訪問にする；
                   v をQに入れる；
               }
          }
}
```

幅優先木の構成と記録



T

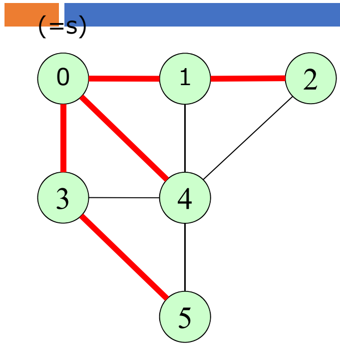
0	0	1	0	0	

未訪問

既訪問

```
アルゴリズム 6.1(改) BFS木
入力：G=(V,E), 始点 s∈V
T[0, n-1]：幅優先木を記録する配列
T[s] = s:
for (各頂点 v∈V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
キュー Q=[s] とする；
while( Qが空でない) {
          Qから頂点 u を取り出す；
          for (v∈adj(u)){
              if (vが未訪問) {
                     T[v] = u；
                 v を既訪問にする；
                   v をQに入れる；
               }
          }
}
```

幅優先木の構成と記録



T

0	0	1	0	0	3

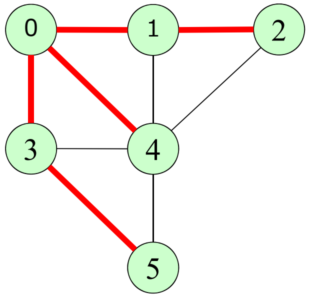
未訪問

既訪問

```
アルゴリズム 6.1(改) BFS木
入力：G=(V,E), 始点 s∈V
T[0, n-1]：幅優先木を記録する配列
T[s] = s:
for (各頂点 v∈V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
キュー Q=[s] とする；
while( Qが空でない) {
          Qから頂点 u を取り出す；
          for (v∈adj(u)){
              if (vが未訪問) {
                     T[v] = u；
                 v を既訪問にする；
                   v をQに入れる；
               }
          }
}
```

経路の復元

(=s)



T

0	0	1	0	0	3

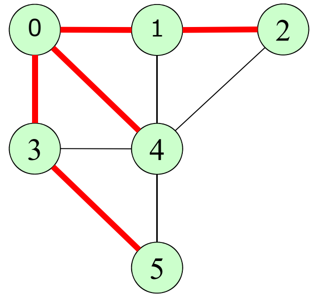
v

```
vから始点sへの経路の復元
(T[0, n-1]：幅優先木を記録する配列)
vを出力
while(v ≠ T[v] ) {
    v = T[v];
    vを出力;
}
```

v=5のとき

経路の復元

(=s)



T

0	0	1	0	0	3

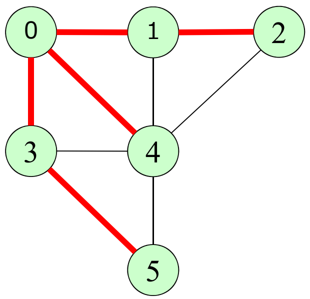
v

```
vから始点sへの経路の復元
(T[0, n-1]：幅優先木を記録する配列)
vを出力
while(v ≠ T[v] ) {
    v = T[v];
    vを出力;
}
```

v=5のとき
<u>5を出力</u>

経路の復元

(=s)



T

0	0	1	0	0	3

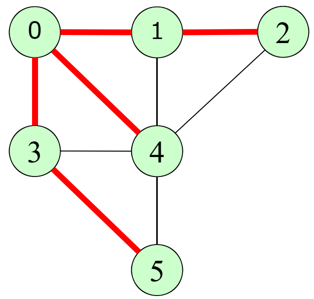
v

```
vから始点sへの経路の復元
(T[0, n-1]：幅優先木を記録する配列)
vを出力
while(v ≠ T[v] ) {
    v = T[v];
    vを出力;
}
```

v=5のとき
<u>5を出力</u>
v = T[v] = T[5] = 3

経路の復元

(=s)



T

0	0	1	0	0	3

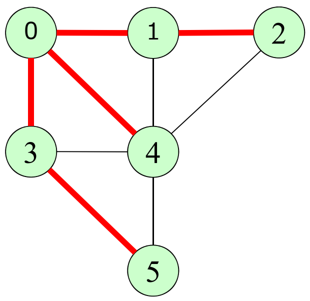
v

```
vから始点sへの経路の復元
(T[0, n-1]：幅優先木を記録する配列)
vを出力
while(v ≠ T[v] ) {
    v = T[v];
    vを出力;
}
```

v=5のとき
<u>5を出力</u>
v = T[v] = T[5] = 3
<u>3を出力</u>

経路の復元

(=s)



T	0	0	1	0	0	3

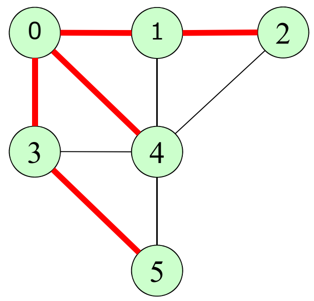
v

```
vから始点sへの経路の復元
(T[0, n-1]：幅優先木を記録する配列)
vを出力
while(v ≠ T[v] ) {
   v = T[v];
   vを出力;
}
```

v=5のとき
5を出力
v = T[v] = T[5] = 3
3を出力
v = T[v] = T[0] = 0

経路の復元

(=s)



T	0	0	1	0	0	3

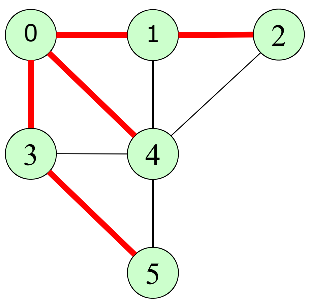
v

```
vから始点sへの経路の復元
(T[0, n-1]：幅優先木を記録する配列)
vを出力
while(v ≠ T[v] ) {
   v = T[v];
   vを出力;
}
```

v=5のとき
5を出力
v = T[v] = T[5] = 3
3を出力
v = T[v] = T[0] = 0
0を出力

経路の復元

(=s)



T	0	0	1	0	0	3

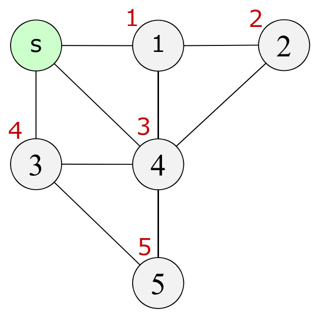
v

```
vから始点sへの経路の復元
(T[0, n-1]：幅優先木を記録する配列)
vを出力
while(v ≠ T[v] ) {
   v = T[v];
   vを出力;
}
```

v=5のとき
5を出力
v = T[v] = T[5] = 3
3を出力
v = T[v] = T[0] = 0
0を出力
v = 0, T[v] = 0なので終了

深さ優先探索

深さ優先探索 (DFS)



探索順序のイメージ

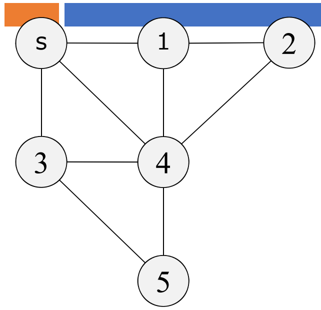
1. s をスタート地点とする
2. s から辿れる頂点を一つ辿る
3. さらにそこから辿れる頂点を辿る
4. 辿れる頂点がなくなったら，辿れる頂点がある頂点まで戻ってそこから同じことの繰り返し

とにかくたどれるだけたどれる

探索順序の例　s → 1 → 2 → 4 → 3 → 5

別の例　s → 4 → 1 → 2 → 3 → 5

DFSアルゴリズム(1)



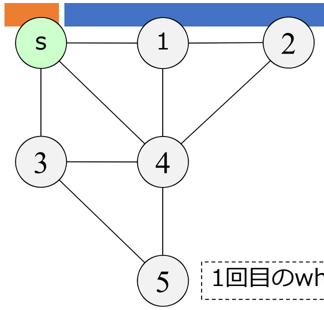
S=[(s,1),(s,3),(s,4)]

未訪問

既訪問

```
アルゴリズム 6.2　DFS
入力：G=(V,E), 始点 s∊V
スタック S を初期化
for (各頂点 v∊V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
for(各辺(s,v)∊E)
   辺 (s,v) をスタックに入れる
while( Sが空でない) {
          Sから辺 (x,y)を取り出す；
          if( yが未訪問) {
             yを既訪問にする;
             for( yに接続する各辺 (y,z)∊E)
                (y,z)をSに入れる；
             }
}
```

DFSアルゴリズム(2)



1回目のwhile開始

S=[(s,3),(s,4)]

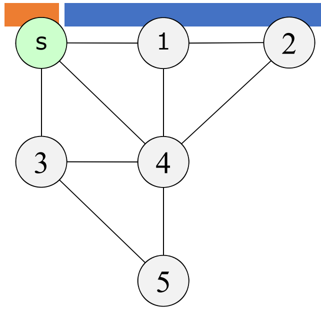
(x,y) = (s,1)

未訪問

既訪問

```
アルゴリズム 6.2　DFS
入力：G=(V,E), 始点 s∊V
スタック S を初期化
for (各頂点 v∊V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
for(各辺(s,v)∊E)
   辺 (s,v) をスタックに入れる
while( Sが空でない) {
          Sから辺 (x,y)を取り出す；
          if( yが未訪問) {
             yを既訪問にする;
             for( yに接続する各辺 (y,z)∊E)
                (y,z)をSに入れる；
             }
}
```

DFSアルゴリズム(3)



S=[(1,s),(1,2),(1,4),(s,3),(s,4)]

(x,y) = (s,1)

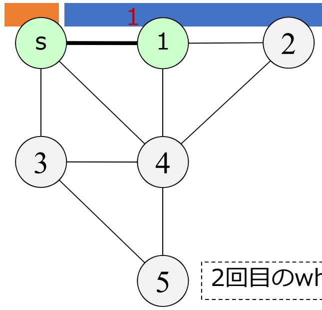
未訪問

既訪問

```
アルゴリズム 6.2　DFS
入力：G=(V,E), 始点 s∊V
スタック S を初期化
for (各頂点 v∊V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
for(各辺(s,v)∊E)
   辺 (s,v) をスタックに入れる
while( Sが空でない) {
          Sから辺 (x,y)を取り出す；
          if( yが未訪問) {
             yを既訪問にする;
             for( yに接続する各辺 (y,z)∊E)
                (y,z)をSに入れる；
             }
}
```

1回目のwhile終了

DFSアルゴリズム(4)



2回目のwhile開始

S=[(1,2),(1,4),(s,3),(s,4)]

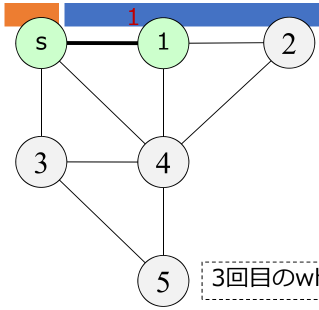
(x,y) = (1,s)

sは既訪問なので何もしない

```
アルゴリズム 6.2　DFS
入力：G=(V,E), 始点 s∈V
スタック S を初期化
for (各頂点 v∈V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
for(各辺(s,v)∈E)
   辺 (s,v) をスタックに入れる
while( Sが空でない) {
          Sから辺 (x,y)を取り出す；
          if( yが未訪問) {
             yを既訪問にする;
             for( yに接続する各辺 (y,z)∈E)
                (y,z)をSに入れる；
             }
}
```

2回目のwhile終了

DFSアルゴリズム(5)



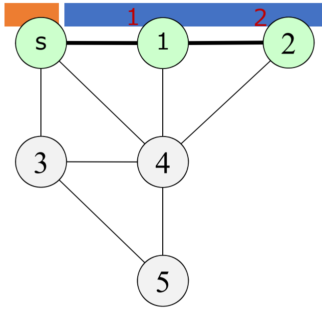
3回目のwhile開始

S=[(1,4),(s,3),(s,4)]

(x,y) = (1,2)

```
アルゴリズム 6.2　DFS
入力：G=(V,E), 始点 s∈V
スタック S を初期化
for (各頂点 v∈V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
for(各辺(s,v)∈E)
   辺 (s,v) をスタックに入れる
while( Sが空でない) {
          Sから辺 (x,y)を取り出す；
          if( yが未訪問) {
             yを既訪問にする;
             for( yに接続する各辺 (y,z)∈E)
                (y,z)をSに入れる；
             }
}
```

DFSアルゴリズム(6)



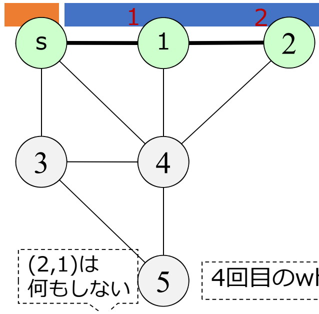
S=[(2,1),(2,4),(1,4),(s,3),(s,4)]

(x,y) = (1,2)

```
アルゴリズム 6.2　DFS
入力：G=(V,E), 始点 s∈V
スタック S を初期化
for (各頂点 v∈V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
for(各辺(s,v)∈E)
   辺 (s,v) をスタックに入れる
while( Sが空でない) {
          Sから辺 (x,y)を取り出す；
          if( yが未訪問) {
             yを既訪問にする;
             for( yに接続する各辺 (y,z)∈E)
                (y,z)をSに入れる；
             }
}
```

3回目のwhile終了

DFSアルゴリズム(7)



(2,1)は
何もしない

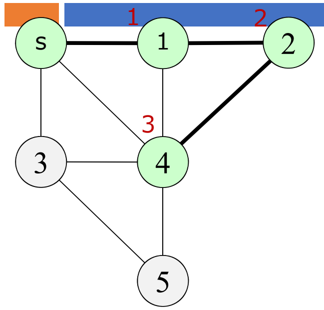
4回目のwhile開始

S=[(1,4),(s,3),(s,4)]

(x,y) = (2,4)

```
アルゴリズム 6.2　DFS
入力：G=(V,E), 始点 s∈V
スタック S を初期化
for (各頂点 v∈V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
for(各辺(s,v)∈E)
   辺 (s,v) をスタックに入れる
while( Sが空でない) {
          Sから辺 (x,y)を取り出す；
          if( yが未訪問) {
             yを既訪問にする;
             for( yに接続する各辺 (y,z)∈E)
                (y,z)をSに入れる；
             }
}
```

DFSアルゴリズム(8)



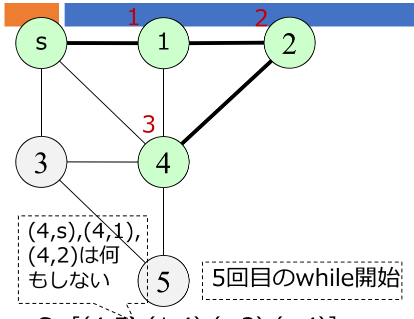
S=[(4,s),(4,1),(4,2),(4,3),(4,5),(1,4),(s,3),(s,4)]

(x,y) = (2,4)

```
アルゴリズム 6.2　DFS
入力：G=(V,E), 始点 s∈V
スタック S を初期化
for (各頂点 v∈V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
for(各辺(s,v)∈E)
   辺 (s,v) をスタックに入れる
while( Sが空でない) {
          Sから辺 (x,y)を取り出す；
          if( yが未訪問) {
             yを既訪問にする;
             for( yに接続する各辺 (y,z)∈E)
                (y,z)をSに入れる；
             }
}
```

4回目のwhile終了

DFSアルゴリズム(9)

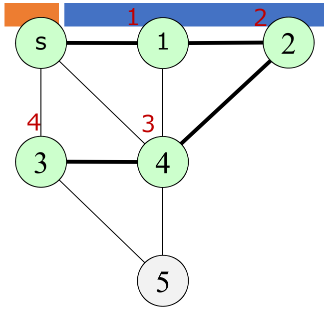


S=[(4,5),(1,4),(s,3),(s,4)]

(x,y) = (4,3)

```
アルゴリズム 6.2　DFS
入力：G=(V,E), 始点 s∈V
スタック S を初期化
for (各頂点 v∈V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
for(各辺(s,v)∈E)
   辺 (s,v) をスタックに入れる
while( Sが空でない) {
          Sから辺 (x,y)を取り出す；
          if( yが未訪問) {
             yを既訪問にする;
             for( yに接続する各辺 (y,z)∈E)
                (y,z)をSに入れる；
             }
}
```

DFSアルゴリズム(10)



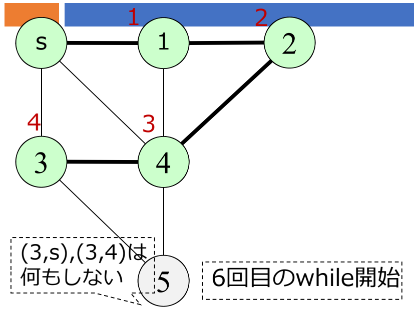
S=[(3,s),(3,4),(3,5),(4,5),(1,4),(s,3),(s,4)]

(x,y) = (4,3)

```
アルゴリズム 6.2　DFS
入力：G=(V,E), 始点 s∈V
スタック S を初期化
for (各頂点 v∈V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
for(各辺(s,v)∈E)
   辺 (s,v) をスタックに入れる
while( Sが空でない) {
          Sから辺 (x,y)を取り出す；
          if( yが未訪問) {
             yを既訪問にする;
             for( yに接続する各辺 (y,z)∈E)
                (y,z)をSに入れる；
             }
}
```

5回目のwhile終了

DFSアルゴリズム(11)

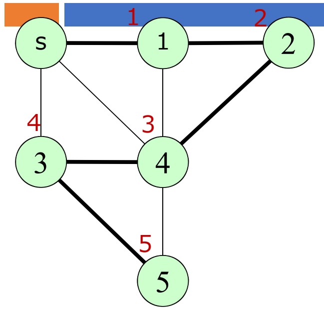


S=[(4,5),(1,4),(s,3),(s,4)]

(x,y) = (3,5)

```
アルゴリズム 6.2　DFS
入力：G=(V,E), 始点 s∈V
スタック S を初期化
for (各頂点 v∈V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
for(各辺(s,v)∈E)
   辺 (s,v) をスタックに入れる
while( Sが空でない) {
          Sから辺 (x,y)を取り出す；
          if( yが未訪問) {
             yを既訪問にする;
             for( yに接続する各辺 (y,z)∈E)
                (y,z)をSに入れる；
             }
}
```

DFSアルゴリズム(12)



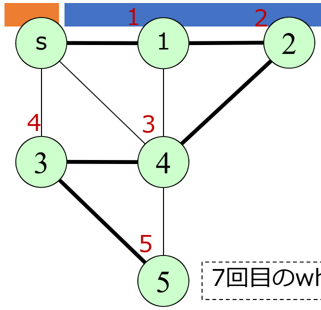
S=[(5,3),(5,4),(4,5),(1,4),(s,3),(s,4)]

(x,y) = (3,5)

```
アルゴリズム 6.2　DFS
入力：G=(V,E), 始点 s∈V
スタック S を初期化
for (各頂点 v∈V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
for(各辺(s,v)∈E)
   辺 (s,v) をスタックに入れる
while( Sが空でない) {
          Sから辺 (x,y)を取り出す；
          if( yが未訪問) {
             yを既訪問にする;
             for( yに接続する各辺 (y,z)∈E)
                (y,z)をSに入れる；
             }
}
```

6回目のwhile終了

DFSアルゴリズム(13)



7回目のwhile開始

S=[ ]

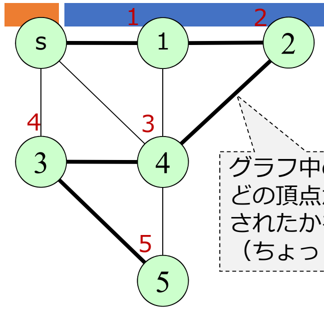
(x,y) = ()

```
アルゴリズム 6.2　DFS
入力：G=(V,E), 始点 s∈V
スタック S を初期化
for (各頂点 v∈V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
for(各辺(s,v)∈E)
   辺 (s,v) をスタックに入れる
while( Sが空でない) {
          Sから辺 (x,y)を取り出す；
          if( yが未訪問) {
             yを既訪問にする;
             for( yに接続する各辺 (y,z)∈E)
                (y,z)をSに入れる；
}
```

すべて既訪問→while文によりスタックから全ての辺を削除

アルゴリズム終了

DFSアルゴリズムの結果



グラフ中の太線は，どの頂点から訪問されたかを示す（ちょっと大事）

訪問した順

s → 1 → 2 → 4 → 3 → 5

注意: 深さ優先探索による訪問順は
スタック S へ辺を push する順序に依存

```
アルゴリズム 6.2　DFS
入力：G=(V,E), 始点 s∈V
スタック S を初期化
for (各頂点 v∈V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
for(各辺(s,v)∈E)
   辺 (s,v) をスタックに入れる
while( Sが空でない) {
          Sから辺 (x,y)を取り出す；
          if( yが未訪問) {
             yを既訪問にする;
             for( yに接続する各辺 (y,z)∈E)
                (y,z)をSに入れる；
             }
}
```

DFSアルゴリズムの実行時間

(隣接リストを用いた場合)

```
アルゴリズム 6.2　DFS
入力：G=(V,E), 始点 s∈V
スタック S を初期化
for (各頂点 v∈V)
   頂点 v に未訪問の印をつける
始点 s に既訪問の印をつける；
for(各辺(s,v)∈E)
   辺 (s,v) をスタックに入れる
while( Sが空でない) {
          Sから辺 (x,y)を取り出す；
          if( yが未訪問) {
             yを既訪問にする;
             for( yに接続する各辺 (y,z)∈E)
                (y,z)をSに入れる；
             }
}
```

$O(n)$

実行全体を通して考えると…

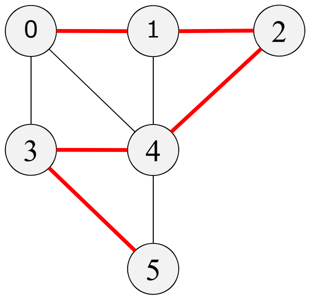
スタックの各操作: $O(1)$

全ての辺が2回ずつpush: $O(m)$

合計 $O(m+n)$

アルゴリズムの実行時間とDFS木

- アルゴリズムの実行時間：幅優先と同じく$O(n+m)$
  (隣接リストのとき)
- 幅優先探索と同様に「DFS木」を定義できる
  - 訪問元→訪問先の辺を取る
  - 格納の仕方も幅優先探索のときと同じ



T

0	0	1	4	2	3

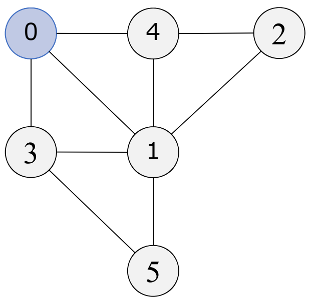
再帰を用いたDFS

- DFSの実現は，再帰呼び出しによりスタックを模倣する実現方法もある
  - 実装はこちらのほうがシンプル

```
アルゴリズム6.3 DFS(再帰版，木の構成付き)
入力：G=(V,E) 始点s∊V
T[0, n-1]：幅優先木を記録する配列
T[s] = s;
s に対する手続きdfs(s)を呼び出す

手続き　dfs(u) {
          頂点uに既訪問の印をつける
          for(頂点uに接続する各辺(u,v)∊E)
            if (頂点 v は未訪問) {
                T[v] = u;
                dfs(v)
}
```

動作例

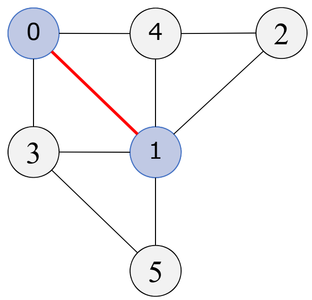


dfs(0)

T

0	0	0	0	0	0

動作例

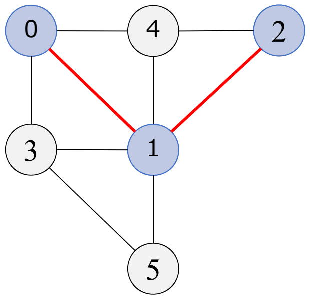


dfs(0)
　dfs(1)

T

0	0	0	0	0	0

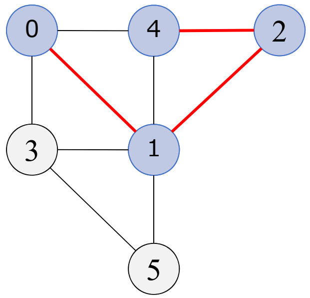
動作例




```
dfs(0)
    dfs(1)
        dfs(2)
```

T	0	0	1	0	0	0

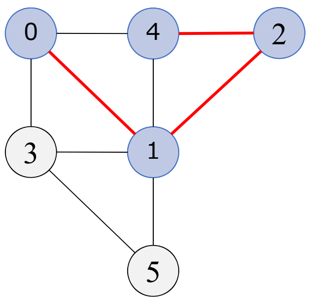
動作例




```
dfs(0)
    dfs(1)
        dfs(2)
            dfs(4)
```

T	0	0	1	0	2	0

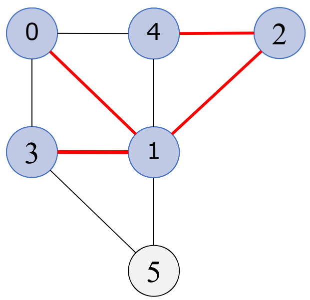
動作例




```
dfs(0)
    dfs(1)
        dfs(2)
            dfs(4)
                return
            return
```

T	0	0	1	0	2	0

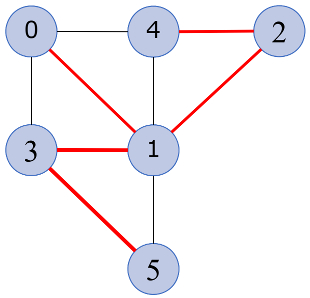
動作例




```
dfs(0)
    dfs(1)
        dfs(2)
            dfs(4)
                return
            return
        dfs(3)
```

T	0	0	1	1	2	0

動作例

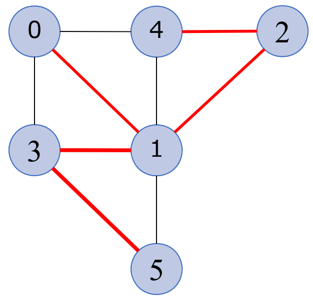



```
dfs(0)
    dfs(1)
        dfs(2)
            dfs(4)
                return
            return
        dfs(3)
            dfs(5)
```

T

0	**0**	**1**	**1**	**2**	**3**

動作例




```
dfs(0)
    dfs(1)
        dfs(2)
            dfs(4)
                return
            return
        dfs(3)
            dfs(5)
                return
            return
        return
    return
return
```

T

0	**0**	**1**	**1**	**2**	**3**

DFS木の性質

- $G = (V, E)$：(有向 or 無向)グラフ
  - $v_1, v_2, \ldots v_n$：DFSの探索順序($s = v_1$)
- T：Gのv_1を根とするDFS木

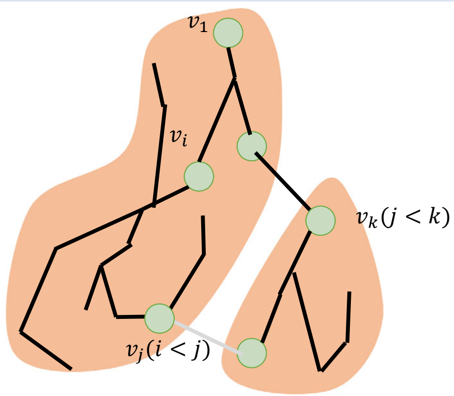
DFS木の性質

v_iを根とするTの部分木は，$G - \{v_1, v_2, \ldots v_{i-1}\}$において$v_i$から到達可能な頂点をすべて含む

DFS木の性質

DFS木の性質

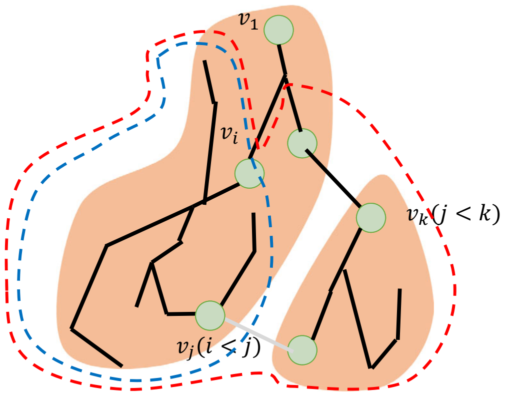
v_iを根とするTの部分木は，$G - \{v_1, v_2, \ldots v_{i-1}\}$において$v_i$から到達可能な頂点をすべて含む



DFS木の性質

DFS木の性質

v_iを根とするTの部分木は，$G-\{v_1, v_2, \ldots v_{i-1}\}$において
v_iから到達可能な頂点をすべて含む

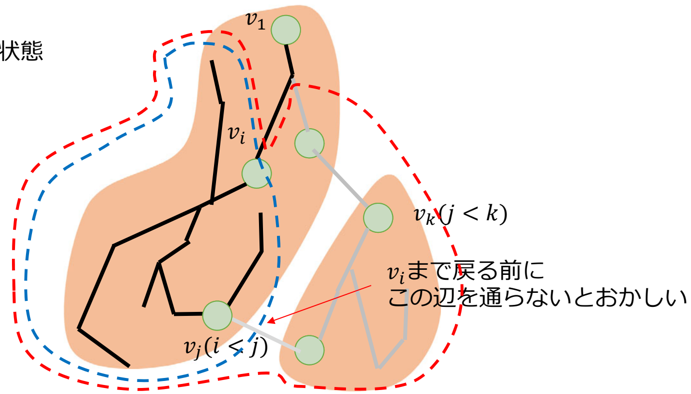


DFS木の性質

DFS木の性質

v_iを根とするTの部分木は，$G-\{v_1, v_2, \ldots v_{i-1}\}$において
v_iから到達可能な頂点をすべて含む

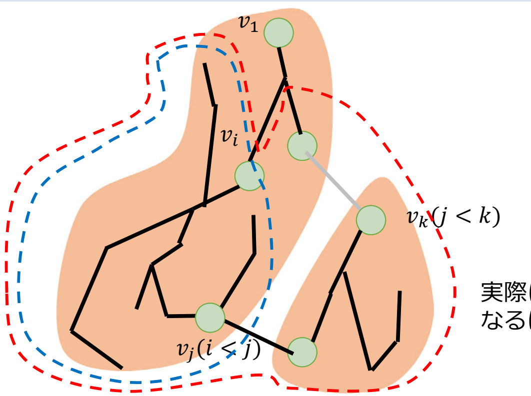
v_j訪問時の状態



DFS木の性質

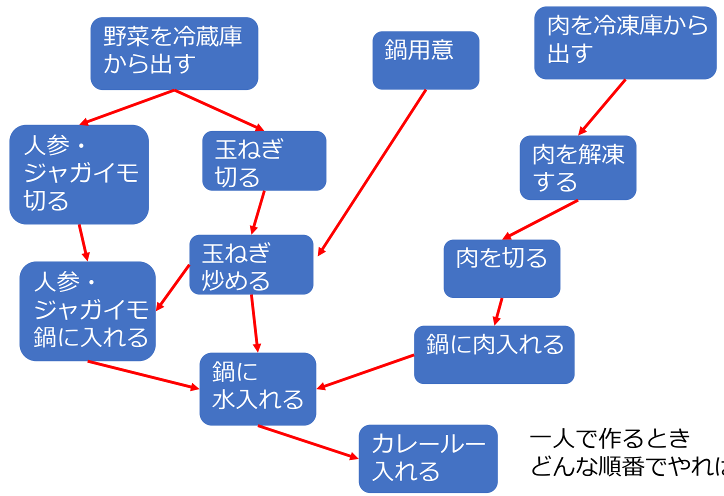
DFS木の性質

v_iを根とするTの部分木は，$G-\{v_1, v_2, \ldots v_{i-1}\}$において
v_iから到達可能な頂点をすべて含む



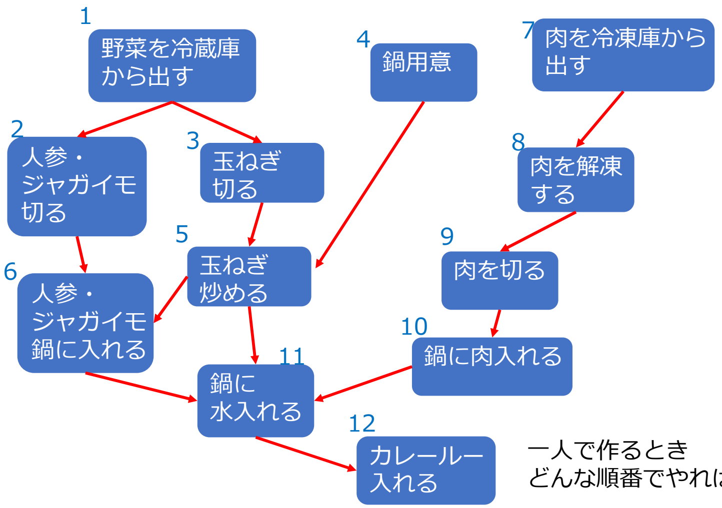
実際にはこんな感じに
なるはず

性質1の応用：トポロジカルソート



一人で作るとき
どんな順番でやればいい?

性質1の応用：トポロジカルソート



一人で作るとき
どんな順番でやればいい?

トポロジカルソート

- 入力
  - $G=(V,E)$：閉路のない有向グラフ(DAGという)
    - 閉路があると問題が成立しないことに注意
- 出力
  - 以下を満たす頂点の並び
    「Gにおいてuからvに到達可能ならば，並びにおいてuはvに先行する」

EをV上の部分順序関係と思ったとき，
順序関係に矛盾しないようにVを並べる問題といってもよい

トポロジカルソート：アルゴリズム

- DFS木の性質1を思い返す

DFS木の性質

v_iを根とするTの部分木は，$G-\{v_1, v_2, \dots v_{i-1}\}$において
v_iから到達可能な頂点をすべて含む

- これと「子は親よりも探索順序において後」という事実から

DFSの探索順の並びはトポロジカルソート列(の一部)

という事実がわかる

トポロジカルソート：アルゴリズム

- 実際には，1つの頂点からDFSを始めても全頂点を訪問できるとは限らない
  →DFSを繰り返す

```
入力：G=(V,E) 始点s∊V
出力：リストL
while (未訪問頂点が存在) {
   sを任意の未訪問頂点とする
   s に対する手続きtsort(s)を呼び出す
}
手続き   tsort(u) {
            頂点uに既訪問の印をつける
            for(頂点uに接続する各辺(u,v)∊E)
              if (頂点 v は未訪問) {
                tsort(v);
              }
              Lの先頭にuを追加
}
```

ここは再帰DFSと同じ

最後に

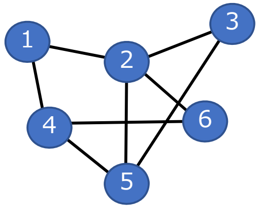
- DFSは他にも色々な応用がある
  - 有向グラフがサイクルを持つか？の判定
  - 強連結成分への分解
  - 関節点，橋の発見
  - 平面性の判定

などなど…

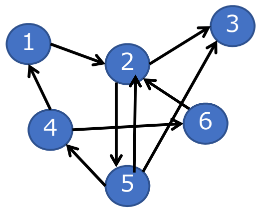
付録：グラフ用語集

グラフとは

- 頂点(vertex)を辺(edge)でつないだもの
  - $G = (V, E)$のような二項組で書く
    - V：頂点集合(集合であればなんでもよい)
    - E：辺集合 ($E \subseteq V \times V$，空もOK)
      - 有向グラフ：辺に向きあり / 無向グラフ：辺に向きなし



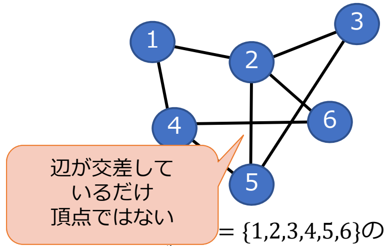
頂点集合$V = \{1,2,3,4,5,6\}$の
無向グラフの例



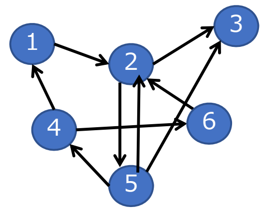
頂点集合$V = \{1,2,3,4,5,6\}$の
有向グラフの例

グラフとは

- 頂点(vertex)を辺(edge)でつないだもの
  - $G = (V, E)$のような二項組で書く
    - V：頂点集合(集合であればなんでもよい)
    - E：辺集合 ($E \subseteq V \times V$)
      - 有向グラフ：辺に向きあり / 無向グラフ：辺に向きなし



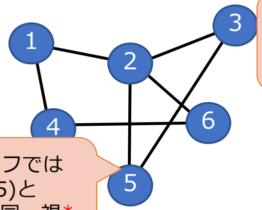
$= \{1,2,3,4,5,6\}$の
無向グラフの図示例



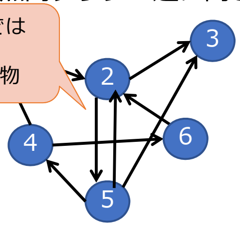
頂点集合$V = \{1,2,3,4,5,6\}$の
有向グラフの図示例

グラフとは

- 頂点(vertex)を辺(edge)でつないだもの
  - $G = (V, E)$のような二項組で書く
    - V：頂点集合(集合であればなんでもよい)
    - E：辺集合 ($E \subseteq V \times V$)
      - 有向グラフ：辺に向きあり / 無向グラフ：辺に向きなし



無向グラフでは
辺(2,5)と
辺(5,2)は同一視*



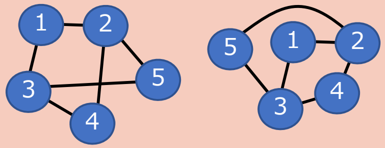
有向グラフでは
辺(2,5)と
辺(5,2)は別物

無向グラフ

(*) 順序が関係ないので，本来は{5,2}と集合で表すべきかもしれないが，慣例的に()で書かれることも多い(この授業もそうやって書く)

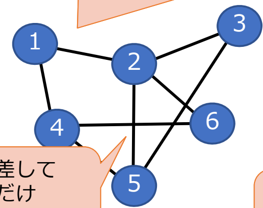
グラフとは

グラフの図示において，頂点の配置や辺の配置は
グラフそのものとは無関係(描き方は自由)



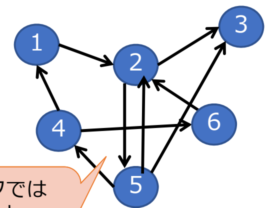
←この２つは同じグラフ

辺に向きあり / 無向グラフ：辺に向きなし



辺が交差して
いるだけ
頂点ではない

= {1,2,3,4,5,6}の
無向グラフの図示例

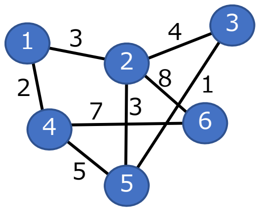


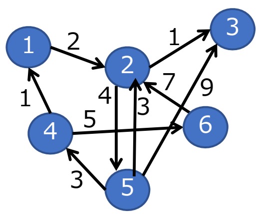
有向グラフでは
辺(2,5)と
辺(5,2)は別物

頂点集合$V = \{1,2,3,4,5,6\}$の
有向グラフの図示例

グラフとは

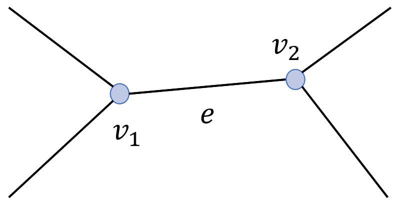
- 重み付きグラフ(weighted graph)
    - V：頂点集合(集合であればなんでもよい)
    - E：辺集合 ($E \subseteq V \times V$)
    - w：重み関数 ($w: E \to \mathbb{N}$) (値域は$\mathbb{R}$とするときもある)





基本的な用語

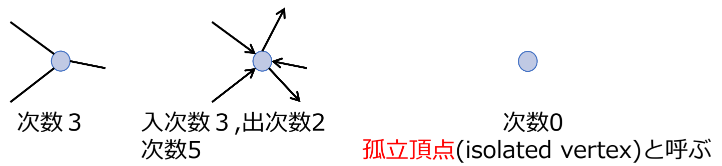
- 頂点v_1が頂点v_2に隣接(adjacent)している
  - 辺(v_1, v_2)がある
- 辺eが頂点v_1に接続（incident）している
  - eの端点のいずれかがv_1（すなわち，$e = (v_1, v_*)$）



基本的な用語

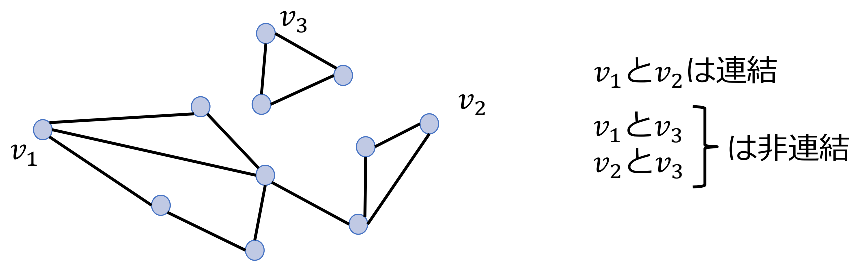
- 頂点v_1の次数(degree)
  - 接続する辺の本数
  - 有向グラフの場合，出次数(out-degree)と入次数(in-degree)の２つがある

(それぞれ「でじすう」「いりじすう」と読む)



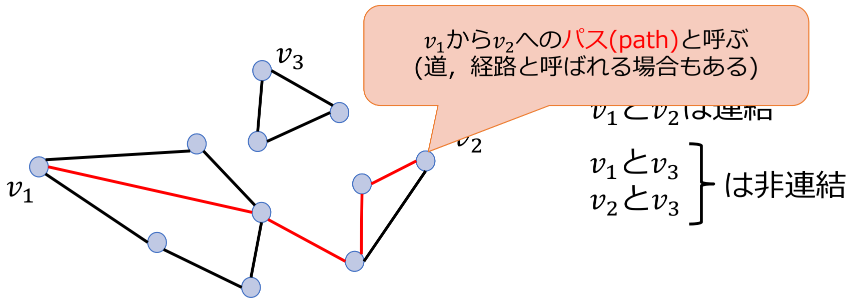
基本的な用語

- 頂点v_1と頂点v_2が連結(connected)している
  - 辺をたどってv_1からv_2に行ける
- 特に，任意の2頂点が連結であるとき，そのグラフは連結であるという



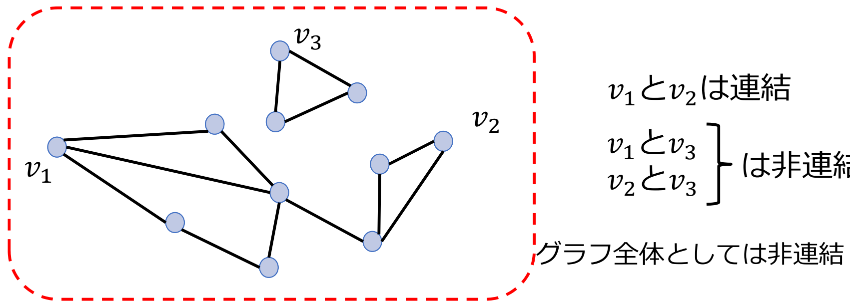
基本的な用語

- 頂点v_1と頂点v_2が連結(connected)している
  - 辺をたどってv_1からv_2に行ける
- 特に，任意の2頂点が連結であるとき，そのグラフは連結であるという



基本的な用語

- 頂点v_1と頂点v_2が連結(connected)している
  - 辺をたどってv_1からv_2に行ける
- 特に，任意の2頂点が連結であるとき，そのグラフは連結であるという

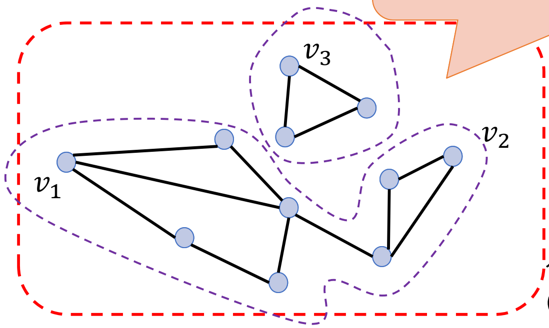


基本的な用語

(*)より厳密には，そのようなもので極大のもの

- 頂点v_1と頂点v_2が連結(connected)している
  - 辺をたどってv_1からv_2に行ける
- 特に，任意の2頂点が
  連結であるという

互いに連結な頂点のグループからなるグラフ（の一部分）を連結成分(connected component)と呼ぶ(*)



v_1とv_2は連結

v_1とv_3
v_2とv_3 } は非連結

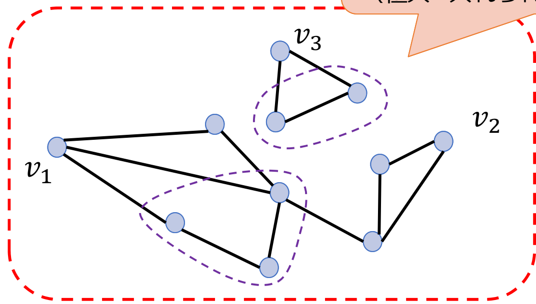
グラフ全体としては非連結
(連結成分数2)

基本的な用語

(*)より厳密には，そのようなもので極大のもの

- 頂点v_1と頂点v_2が連結(connected)している
  - 辺をたどってv_1からv_2に行ける
- 特に，任意の2頂点が
  連結であるという

こういうのは連結だが
連結成分とは言わない
（極大＝入れられるものはすべて入れる）



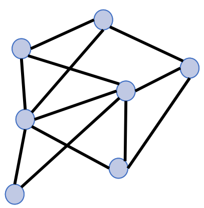
v_1とv_2は連結

v_1とv_3
v_2とv_3 } は非連結

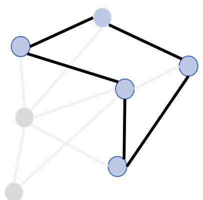
グラフ全体としては非連結
(連結成分数2)

(誘導)部分グラフと全域部分グラフ

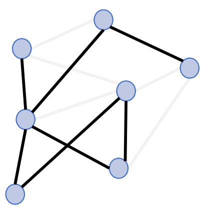
- $G = (V, E)$の部分グラフ(subgraph)$G' = (V', E')$
  - $V' \subseteq V$, $E' \subseteq E$であるようなグラフ
    - ただしE'中の辺の端点はV'に含まれないといけない
- 特に$V = V'$のとき，全域部分グラフ(spanning subgraph)と呼ぶ



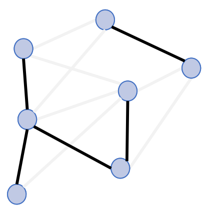
もとのグラフ



部分グラフ



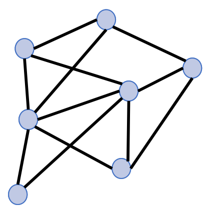
全域部分グラフ



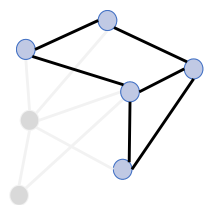
これも全域部分グラフ

(誘導)部分グラフと全域部分グラフ

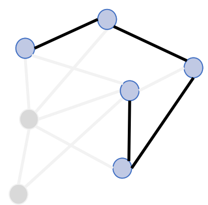
- $G = (V, E)$のV'による誘導部分グラフ ($V' \subseteq V$)
  (subgraph induced by V')
  - $E' = (V' \times V') \cap E$であるような部分グラフ$G' = (V', E')$
    - 両端点がV'に含まれるような辺はすべて含まれないといけない



もとのグラフ



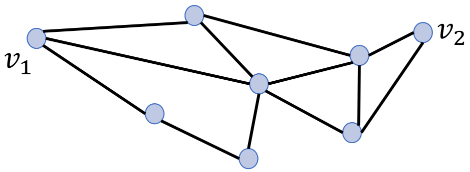
(色のついた頂点集合による)
誘導部分グラフ



部分グラフだが
誘導部分グラフではない

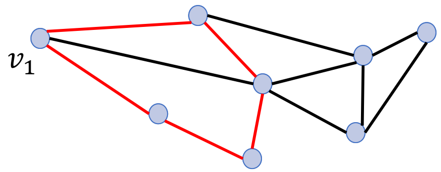
パスに関して補足(1)

- 自分自身から自分自身へのパスは特に閉路(cycle)と呼ぶ



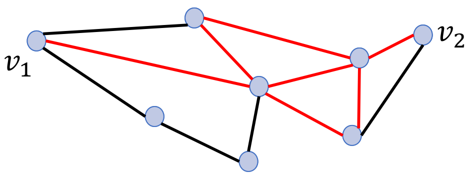
パスに関して補足(1)

- 自分自身から自分自身へのパスは特に閉路(cycle)と呼ぶ



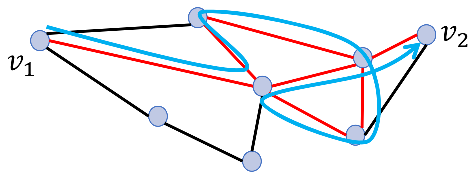
パスに関して補足(2)

- これはv_1からv_2へのパスだろうか？



パスに関して補足(2)

- これはv_1からv_2へのパスだろうか？

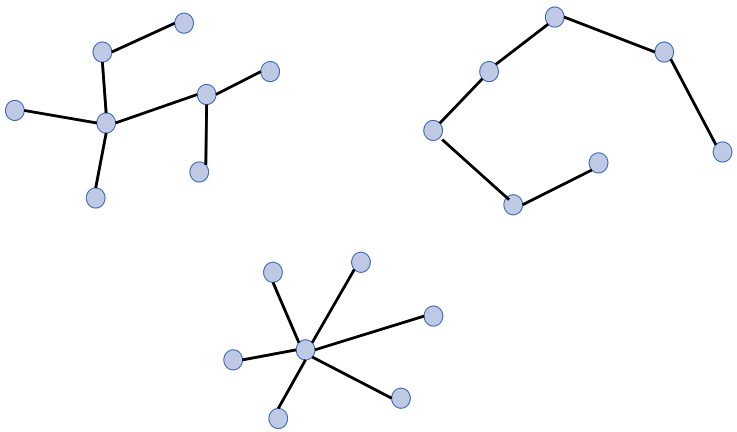


  - 一応パスである
  - 同じ頂点を2回通らないものを「単純(simple)なパス」と呼んで区別する
  - ただし，「パス」といったとき，暗黙に単純なパスを指していることも多い

（同様に閉路も「単純な閉路」と区別する）

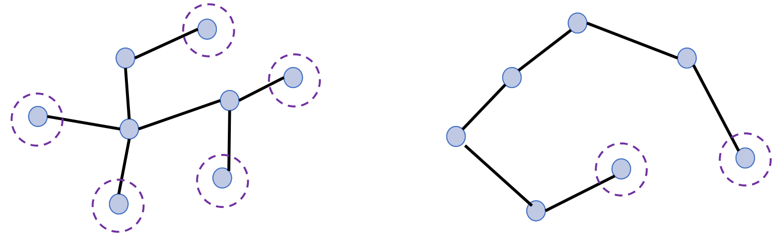
木(Tree)

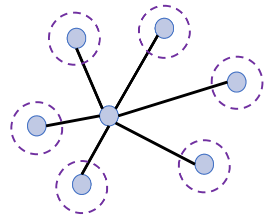
- 定義：閉路を持たない連結グラフ



木(Tree)

- 定義：閉路を持たない連結グラフ

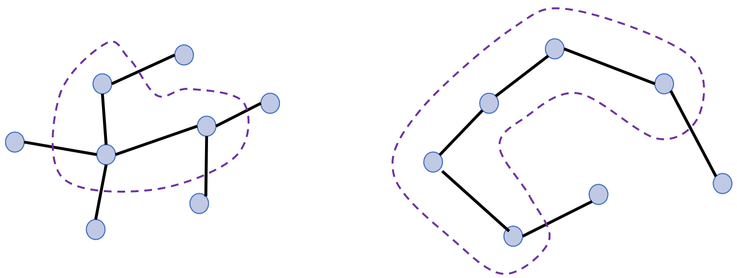


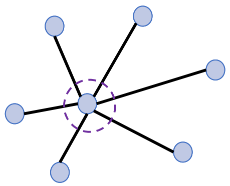


葉(leaf)：次数1の頂点

木(Tree)

- 定義：閉路を持たない連結グラフ

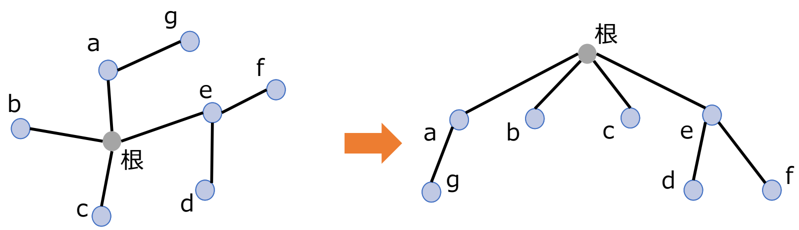




内部頂点(internal vertex)：
葉でない頂点

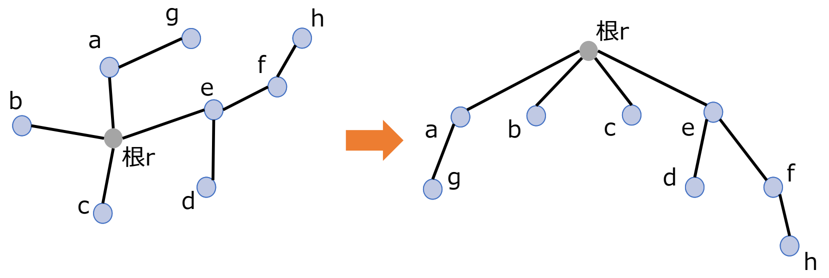
木(Tree)

- 特別な1頂点(根(root)という)を指定して親子関係を定めたものを根つき木(rooted tree)と呼ぶ



木(Tree)

- 特別な1頂点(根(root)という)を指定して親子関係を定めたものを根つき木(rooted tree)と呼ぶ



eはdの親(parent) / dはeの子(child)
hはeの子孫(descendant) / eはhの祖先(ancestor)