アルゴリズムとデータ構造

- 第13回講義トピック:グラフ
  - グラフの定義　Definition of graph
  - グラフの表現　Representation of graph
  - 深さ優先探索　Depth-first search
  - 幅優先探索　Breadth-first search

グラフとは

- グラフはツリーの拡張である。
- 形式的には、グラフは

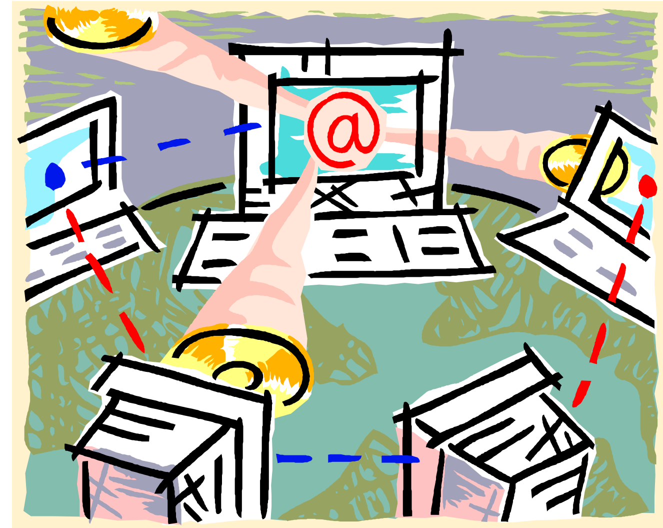
$$G=(V,E)$$

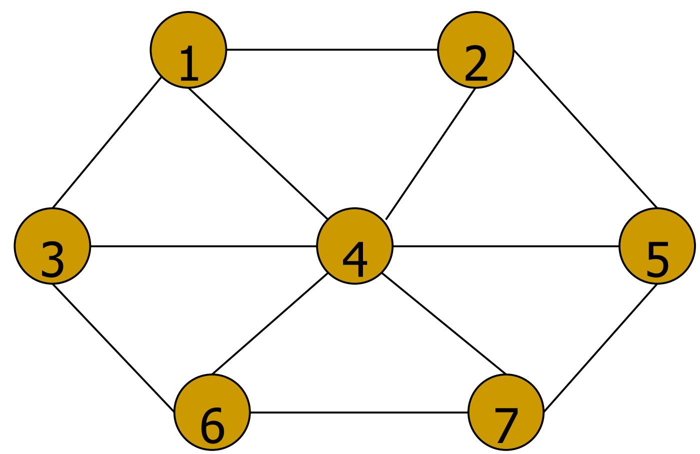
のように、VとEの2つ組（2-tuple, 或は double)で定義される。ここで、
- Vは頂点（vertex, vertices）或はノード(node)の集合
- Eはエッジ(edge)或は連結(connection)の集合である。エッジは頂点のペアからなる。無向グラフではエッジの頂点に順序関係はない。有向グラフでは、頂点の順序関係があり、エッジ(u,v)はuからvへのエッジを示す。この場合、vはuに隣接しているという。

グラフの例

- コンピュータネットワーク
- 日本の航空路線図
- 会津若松の上下水道図
- 電気回路

- グラフはこれらのシステムを表すためのもの。
- グラフにおいて、ノードの位置は、通常意味を持たない。
- ノード間の繋がりが重要である。



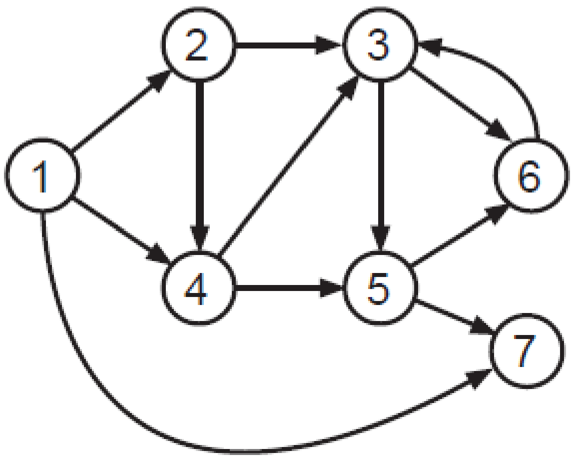


関連用語(1)

- Path(パス):ノードAとBの間を繋げる複数の連結。
- Simple path(シンプルパス): ノードAとBを繋げるパスにおいて、同じノードが2回以上現れない。
- Cycle(サイクル): Simple pathで、始点と終点が同じである。
- Connected graph(連結グラフ):すべてのノードの間にパスがある。
- Connected component(連結成分): 部分グラフで、それ自体が連結グラフになっている。
- Directed graph(有向グラフ): 連結に方向性が(一方通行)あるグラフ。
- Undirected graph(無向グラフ):連結に方法性がないグラフ。
- Weighted graph: 連結に重みが付いているグラフ。
- Tree: サイクルがないグラフ。
- Spanning tree: グラフのすべてのノードを持つツリー(部分グラフ)
- Strongly connected(強連結):有効グラフにおいて任意の頂点u, v間で双方向に道がある。強連結な部分グラフ(strongly connected component)は、強連結成分という。
- 有向グラフGに対応する無向グラフが連結であれば、Gは連結である。

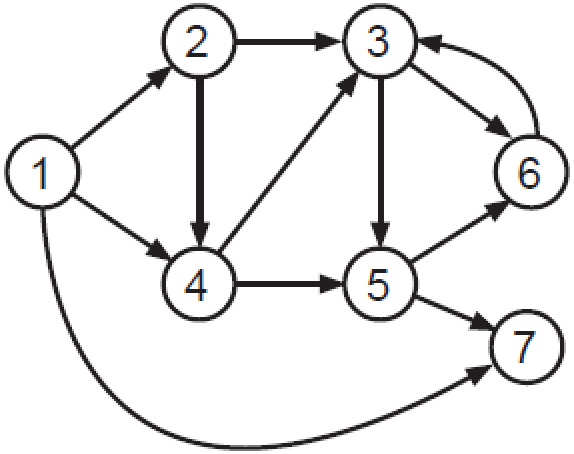
クイズ(1)

- 右のグラフについて、以下の問いを答えよ:
  - ノード1とノード7の間のパスを1つだけ示せ。
  - このグラフにあるサイクルを一つ示せ。



クイズ(2)

- 右のグラフについて、以下の文が正しいか否か:
  - このグラフは連結グラフである。(　　)
  - このグラフは無向グラフである。(　　)
  - このグラフは重み付きグラフである。(　　)
- このグラフのスパニングツリー(spanning tree)を一つ示せ。



グラフの表現その１
隣接リスト法(Adjacency-list representation)

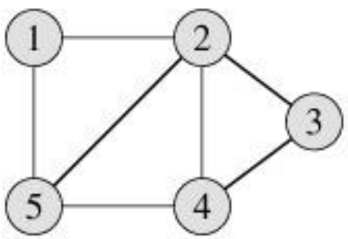
- N=|V|：ノード数
- N個のリストAdj[0],Adj[1],…,Adj[N-1]を定義する。
- Adj[i]は、i番目のノードに対応するリストで、このノードに隣接するすべてのノードを含む。
  即ち、Adj[i]に含まれている任意のノードjに対して、(i,j)はEに属する連結である。

グラフの表現その２
隣接行列法(Adjacency-matrix representation)

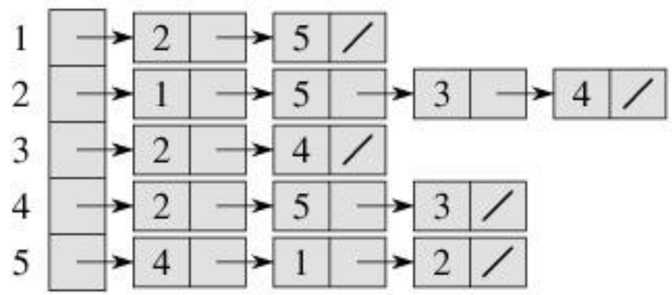
- N=|V|: ノード数
- ノードに番号を振る（任意）
- NxNの行列Aを定義する。
- Aのi行目、j列目の要素は以下のように定義される:

$$a_{ij} = \begin{cases} 1 & if\,(i, j) \in E \\ 0 & otherwise \end{cases}$$

グラフ表現の例：無向グラフ



(a)

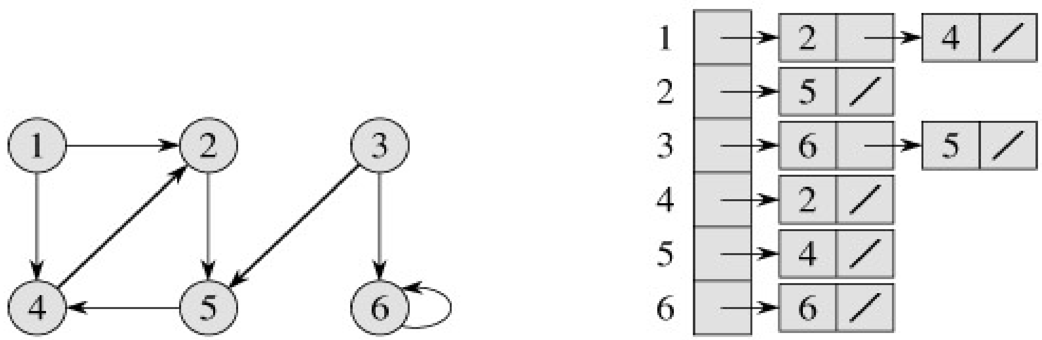


(b)

$$
\begin{array}{c|ccccc|}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 1 & 0 & 0 & 1 \\
2 & 1 & 0 & 1 & 1 & 1 \\
3 & 0 & 1 & 0 & 1 & 0 \\
4 & 0 & 1 & 1 & 0 & 1 \\
5 & 1 & 1 & 0 & 1 & 0 \\
\hline
\end{array}
$$

(c)

グラフ表現の例：有向グラフ



(a) (b)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

深さ優先探索（Depth-first search)

- 深さ優先探索(DFS)は、文字通り、できるだけ深いところ、或いは先の先を探索（訪問）する。
- 即ち、現在ノードに未探索の子ノードがある限り、その子ノードを再帰的に探索する。
- 現在ノードの子ノードが全部探索済みの場合では、探索は“backtrack”し、現在ノードの親ノードの場所に戻る。
- このように、すべてのノードが訪問されるまで（或は、所望のノードを見つかるまで）、以上のことを繰り返す。
- 以上の操作が終わっても、未探索のノードがある場合（連結グラフではない場合）、その中から一つ選び、以上のことを繰り返す。

深さ優先探索の擬似コード（1）

DFS(G)

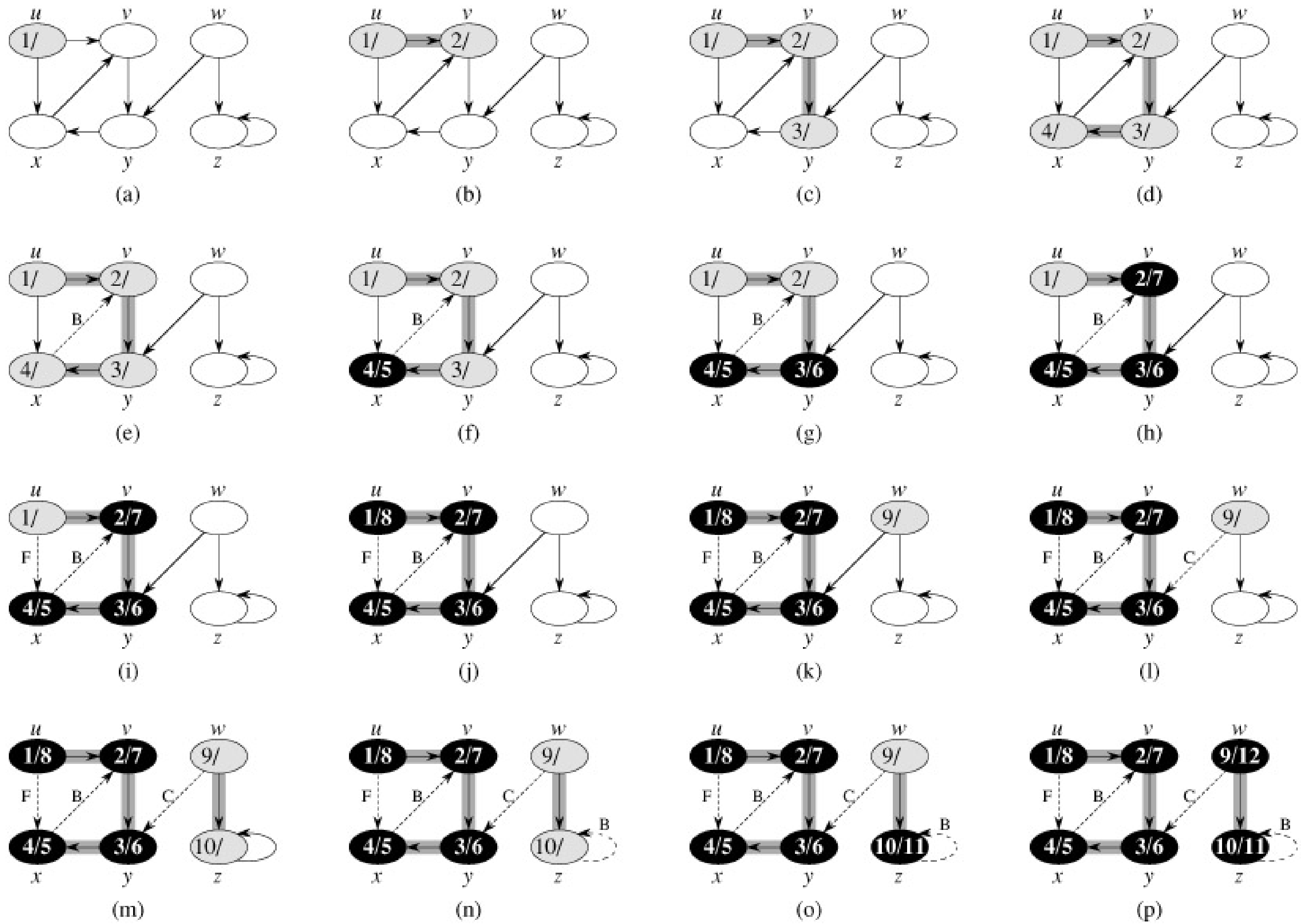
```
for each vertex u ∈ V[G]
     do color[u] ← WHITE
time ← 0
for each vertex u ∈ V[G]
     do if color[u] = WHITE
          then DFS-Visit(u)
```

深さ優先探索の擬似コード（2）

DFS-Visit(u)

```
color[u] ← GRAY   // just been discovered
time ← time+1
d[u] ← time  // discovered time
for each v in Adj[u]  // explore edge (u,v)
     do if color[v] = WHITE
               then DFS-Visit(v)
color[u] ← BLACK    // finished
f[u] ← time ← time+1  // finished time
```

深さ優先探索の例


u v w
1/
x y z
(a)
u v w
1/ 2/
x y z
(b)
u v w
1/ 2/
3/
x y z
(c)
u v w
1/ 2/
4/ 3/
x y z
(d)
u v w
1/ 2/
B
4/ 3/
x y z
(e)
u v w
1/ 2/
B
4/5 3/
x y z
(f)
u v w
1/ 2/
B
4/5 3/6
x y z
(g)
u v w
1/ 2/7
B
4/5 3/6
x y z
(h)
u v w
1/ 2/7
F B
4/5 3/6
x y z
(i)
u v w
1/8 2/7
F B
4/5 3/6
x y z
(j)
u v w
1/8 2/7 9/
F B
4/5 3/6
x y z
(k)
u v w
1/8 2/7 9/
F B C
4/5 3/6
x y z
(l)
u v w
1/8 2/7 9/
F B C
4/5 3/6 10/
x y z
(m)
u v w
1/8 2/7 9/
F B C B
4/5 3/6 10/
x y z
(n)
u v w
1/8 2/7 9/
F B C B
4/5 3/6 10/11
x y z
(o)
u v w
1/8 2/7 9/12
F B C B
4/5 3/6 10/11
x y z
(p)

コメント

- DFS timestamps each vertex.
- Each vertex v has two timestamps:
  - the first timestamp d[v] records when v is first discovered, and
  - the second timestamp f[v] records when the search finishes examining v's adjacency list.
- These timestamps are used in many graph algorithms and are usually helpful in reasoning about the behavior of DFS.

幅優先探索(Breadth-first search)

- 深さ優先探索(DFS)と同じように、幅優先探索(BFS)も、与えられたグラフG = (V, E)の任意のノードsから、すべてのノードを、系統的に探索する手法である。
- DFSと違ってBFSは、現在ノードのすべての子ノードを先に探索する。すべての子ノードが探索終わった場合のみ、孫ノードを探索する。
- 深さを制御するために、
  - sからの距離を計算する。ノードuの距離はd[u]で表す。
  - キューQを使用してノードを管理する。

幅優先探索の擬似コード(1)

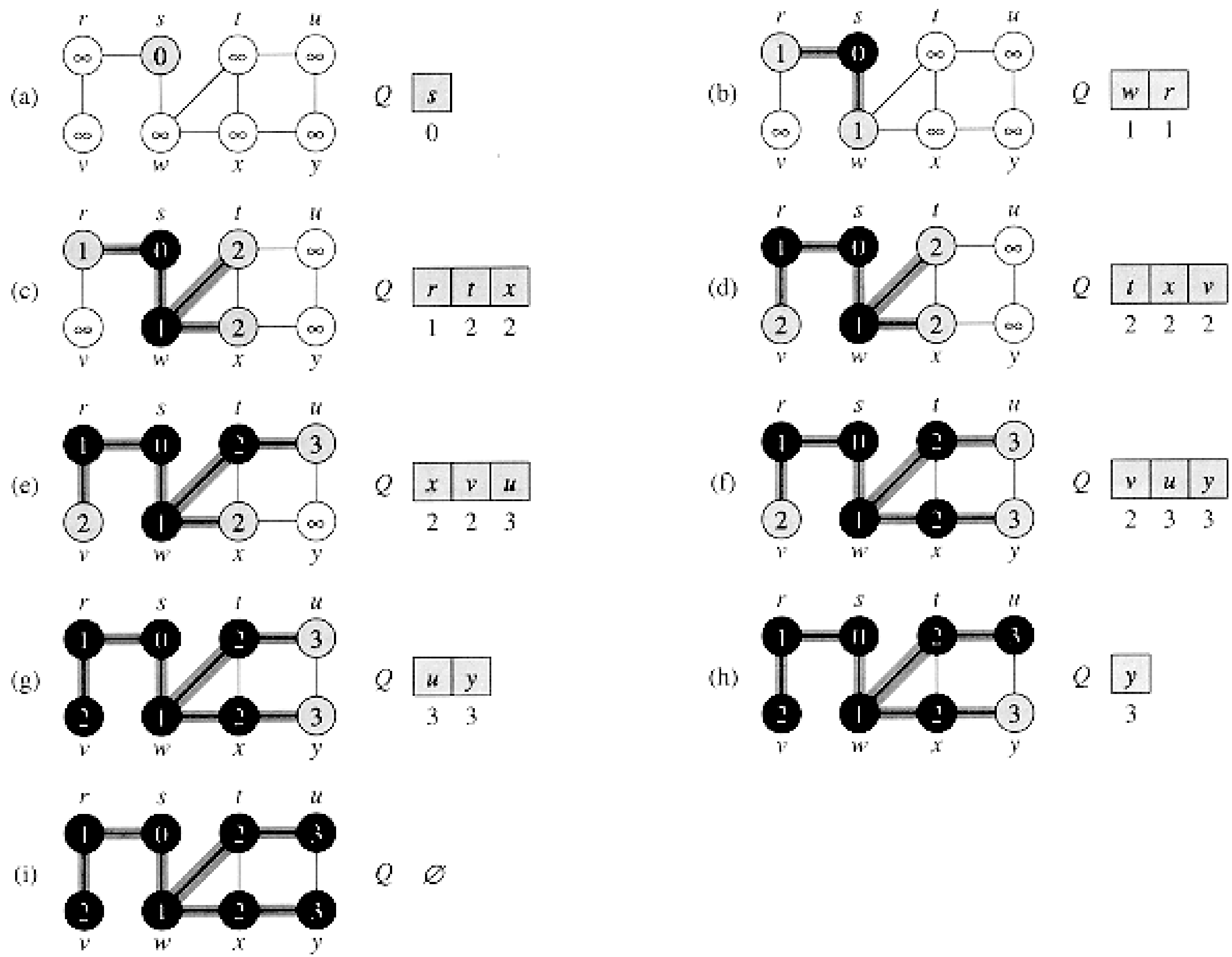
BFS(G, s)

```
for each vertex u ∈ V[G] -{s}
    do    color[u] ← WHITE
          d[u] ← ∞
color[s] ← GRAY
d[s] ← 0
Q ← φ
Enqueue(Q, s)
```

幅優先探索の擬似コード(2)

```
while Q ≠φ
    do u ← Dequeue(Q)
         for each v ∈ Adj[u]
              do     if color[v] = WHITE
                     then   color[v] ← GRAY
                            d[v] ← d[u] + 1
                            Enqueue(v)
         color[u] ← BLACK
```

幅優先探索の例


(a)
Q s
0
(b)
Q w r
1 1
(c)
Q r t x
1 2 2
(d)
Q t x v
2 2 2
(e)
Q x v u
2 2 3
(f)
Q v u y
2 3 3
(g)
Q u y
3 3
(h)
Q y
3
(i)
Q ∅