グラフのアルゴリズム

神戸高専 AE1 r209211
野瀬田 裕樹

2009 年 12 月 1 日 (火)

ケーニヒスベルクの橋問題

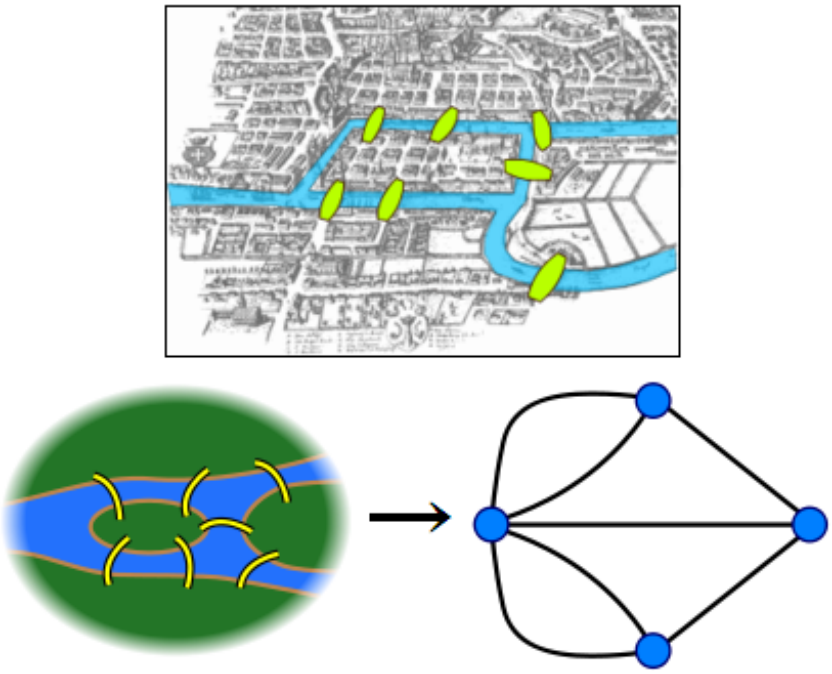

図 1　ケーニヒスベルクの橋問題

「このプレーゲル川に架かっている 7 つの橋を 2 度通らずに，全て渡って，元の所に帰ってくることができるか．ただし，どこから出発してもよい」

レオンハルト・オイラーはこの問題をグラフに置き換えて一筆書きの問題として解いた．

目次

1　グラフとは

1.1　グラフの定義

グラフ (graph) とは，一般的には「複数の数量や関数の関係を図形に示したもの」という意味で用いられる．しかし，ここで用いるグラフとは，いわゆる**グラフ理論** (graph theory) [*1]で使われる用語としてのグラフ[*2]であり，その定義は以下に示すとおりである．

> グラフ G は二つの集合 $V(G)$，$E(G)$ と 1 つの関数 ϕ_G の組 $(V(G), E(G), \phi_G)$ をいう．

ここで，$V(G)$ は空集合ではなく，その元は G の**頂点** (vertex, node, 節点) と呼ばれる．また，$E(G)$ は $V(G)$ とは素な集合[*3]で，その元を G の**辺** (edge, arc, branch, 枝) と呼ぶ．ϕ_G は**接続関数**といわれ，G の各辺に G の頂点の対を対応させる．

$e \in E$ で，u と v が $\phi_G(e) = uv$ である頂点であれば，「e は u と v を結ぶ」といい，u と v を e の**端点**と呼ぶ．

グラフの名の由来はそれが図形として表示できることにある．頂点は点により，辺はその両端点を結ぶ線によって示される．以下にその例を一つ挙げる．

$$G = (V(G), E(G), \phi_G) \tag{1.1}$$

$$\begin{cases} V(G) = v_1, v_2, v_3, v_4 \\ E(G) = e_1, e_2, e_3, e_4 \\ \phi_G(e_1) = v_1v_2, \quad \phi_G(e_2) = v_2v_3 \\ \phi_G(e_3) = v_1v_3, \quad \phi_G(e_4) = v_3v_4 \end{cases}$$

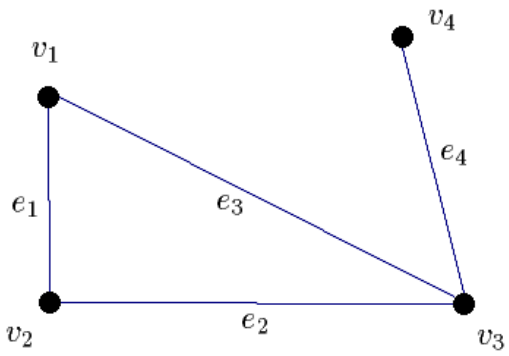

図 2　グラフ G の表示

例では，頂点を小さな黒丸で表した．頂点を表す点と辺を表す線の相対的位置は意味がないため，グラフを描く一意的な方法はない．

辺の端点はその辺に，また辺はその端点に**接続**しているという．1 つの辺の両端点は**隣接**しているといい，また同一の頂点に接続している 2 つの辺は隣接しているという．両端点が一致する辺を**ループ** (loop)，両端点が異なる辺を**リンク** (link) という．ループの存在を許すとき，そのグラフを**一般グラフ** (general graph) という．2 つの辺の端点が異なるとき，その 2 つの辺は**互いに素である**という．

[*1] グラフ理論とは，いくつかの点とこれらを結ぶ線分からなる図形の位相幾何学的性質を解析する数学理論である．

[*2] グラフとは，物事の結び付きを抽象化した概念であり，教科書にその具体的な問題の例が載っている．

[*3] 互いに素な集合であるとは，それらが共通の元 (げん) を持たぬことをいう．

1.2 グラフの種類

頂点集合と辺集合が共に有限集合であるグラフを**有限**であるという．頂点集合と辺集合の両方，あるいはいずれか一方が無限集合であるグラフを**無限**であるという．

ただ 1 つの頂点をもつグラフを**自明**，他はすべて**非自明**なグラフという．

グラフには，今まで示してきたような辺に方向がない**無向グラフ** (undirected graph) の他に，各辺に 1 つの向き付けを与えた**有向グラフ** (directed graph) と呼ばれるグラフが存在する．ここで，有向グラフの定義を示す．

有向グラフ D は二つの集合 $V(D)$，$A(D)$ と 1 つの関数 ϕ_D の組 $(V(D), A(D), \phi_D)$ をいう．

ここで，$V(D)$ は空集合ではなく，その元は**点**と呼ばれる．$A(D)$ は $V(D)$ と互いに素な集合で，その元を**孤**と呼ぶ．ϕ_D は**接続関数**といわれ，$A(D)$ の各元に $V(D)$ の元の順序対を対応させる関数である．

グラフ G の各辺 e に 1 つの実数 $\omega(e)$ を対応させて e の**重み** (weight) と呼び，各辺に重みを付けたグラフを**重み付きグラフ** (weighted graph) という．

グラフ G における**歩道** (walk) とは，有限個の点と辺の交互列 $W = v_0e_1v_1e_2v_2\cdots e_kv_k$ をいう．ここで $e_i(1 \leq i \leq k)$ の両端点は v_{i-1}，v_i である．v_0, v_k をそれぞれ W の**始点** (starting vertex)，**終点** (terminating vertex)，$v_1, \cdots, v_k$ を**内点**，整数 k を W の**長さ** (length) という．

歩道 W は，その辺 $e_1, e_2, \cdots, e_k$ がすべて異なるとき，**小径** (trail) と呼ばれる．さらに，$v_0, v_1, \cdots, v_k$ もすべて異なるとき，W を**道** (path) という．

歩道は，長さが正で，その始点と終点とが一致するとき，**閉**という．始点も内点もすべて異なる閉小径を**閉路** (cycle) という．道のうち閉路を含まないものを**単純路** (simple path) と呼ぶ．

グラフ G の点 v_i から点 v_j への道が存在するとき，点 v_i と点 v_j は**連結している** (connected) という．G のどの 2 点も連結しているとき，G を**連結グラフ**といい，そうでないときは**非連結グラフ**という．

単純グラフ[*4]においてそのすべての点対がちょうど 1 つの辺で結ばれているとき，そのグラフは**完全グラフ** (complete graph) といわれる．

1.3 グラフアルゴリズムの計算量

グラフに関するアルゴリズムの計算量の評価は，グラフを構成する頂点の数と辺の数に関するオーダーを求めることによって行う．

辺の本数が多いグラフを**密なグラフ**といい，逆に辺の本数が比較的少ないグラフを**疎なグラフ**という．

[*4] ループがなく，頂点のどの対も 1 つのリンクで結ばれているグラフのことを単純グラフ (simple graph) という．

2 グラフの表現方法

2.1 行列表現 (matrix representation)

任意の無向グラフ G の点集合，辺集合をそれぞれ

$$V(G) = v_1, v_2, \cdots, v_n,\ E(G) = e_1, e_2, \cdots, e_m \tag{2.1}$$

とする．点 v_i が辺 e_j に接続する数を $m_{ij}(0, 1$ または 2 である) とすれば，$n \times m$ 行列 $\mathbf{M}(G) = (m_{ij})$ ができる．これを G の**接続行列**と呼ぶ．また v_i と v_j を結ぶ辺の数を a_{ij} とし，a_{ij} を要素とする $n \times n$ 行列 $\mathbf{A}(G) = (a_{ij})$ を G の**隣接行列**という．

任意の有向グラフ D の場合も無向グラフの場合と同様であり，D の点集合 $V(D)$ が

$$V(D) = v_1, v_2, \cdots, v_n \tag{2.2}$$

であるとき，a_{ij} を要素とする $n \times n$ 行列 $\mathbf{A}(D) = (a_{ij})$ を D の**隣接行列**という．なお，a_{ij} は v_i から v_j への孤の数を表す．

無向グラフの場合は隣接行列が対称行列となり，有向グラフの場合はそうはならない．

辺に重みが定義されているときは，隣接行列の要素を重みとする行列 $\mathbf{W}$ を用いる．この行列 $\mathbf{W}$ は重み行列と呼ばれる．辺が存在しない場合は特別な値を用いることが多い．

2.2 リスト表現 (list representation)

隣接リスト L とは，グラフの各点 v_i について，点 v_i に隣接するすべての点を任意の順番でつないで得られるリストである．以下に，隣接リスト L の型の例を示す．

```
#define N 100          /* グラフの点の数を最大 100 個までとする */
struct LIST {          /* 構造体 LIST の宣言 */
  int v_num;           /* 点の番号 */
  // int weight;       /* 重みの値を格納する変数 */
  struct LIST *next;   /* 次の構造体を指すポインタ */
}
struct LIST *vertex[N+1];  /* LIST 型のポインタ配列の宣言 */
```

2.3 二つの表現法の比較

① 行列表現
・頂点数 n に対して $O(n^2)$ の記憶容量が必要
・任意の二頂点を結ぶ辺の有無を調べる計算量は $O(1)$
・辺の始点から終点，終点から始点のいずれの向きでも調べられる
② リスト表現
・頂点数 n および辺の数 m に対して $O(m+n)$ の記憶容量が必要
・任意の二頂点を結ぶ辺の有無を調べる計算量は $O(n)$
・終点から始点を調べるのは難しい

3 グラフの探索

グラフの最も基本的なアルゴリズムは，すべての節点や枝を通って処理を行うことである．このとき 2 回以上同じ節点や枝を通らないようにしながら，1 回は必ず通るように，系統だった手順が必要になる．このように系統立ててグラフを調べることをグラフの**探索** (search) という．また，探索の過程でそれぞれの頂点を調べにいくことをその頂点の**訪問** (visit) と呼ぶ．

グラフを探索する際には，そのグラフが**連結グラフ** (connected graph) [*5]であるかどうかに注意する必要がある．連結グラフの場合には，任意の頂点からの一度の探索で全ての頂点を訪問することができるが，非連結グラフの場合には，その頂点と連結した頂点しか訪問できない．

グラフの探索アルゴリズムの代表的なものとして，**深さ優先探索** (depth first search. 縦型探索ともいう) [*6]と**幅優先探索** (breadth first search, 横型探索ともいう) [*7]がある．

3.1 深さ優先探索

深さ優先探索のアルゴリズムを以下に示す．

(1) 任意の頂点を訪問する．
(2) 頂点を訪問した後，以下のようにして次に訪問する頂点を探す．
(2-1) その頂点に隣接した未訪問の頂点があれば，その頂点のどれかを訪問し，(2) へ戻る．
(2-2) 未訪問の頂点がなければ，今訪問している頂点の前の頂点に引き返して，(2) の続きを行う．
(3) すべての頂点とそれらから出る辺を調べ終わったら探索は終了する．

このアルゴリズムを実装するプログラムの雛形は以下のようになる．

```
void depthFirstSearch( int currentNode ){
  currentNode を訪問済みとし，currentNode に対する処理を行う
  ( currentNode に隣接しているノード nextNode を探す ){
    調査中の辺に対して処理を行う
    nextNode が未訪問ならば depthFirstSearch( nextNode );
  }
}
```

深さ優先探索では，途中でたどった辺は木を構成する．この木を構成する辺を**木の辺** (tree edge)，それ以外の辺を**逆辺** (back edge) と呼ぶ．

無向グラフの場合は連結であるかどうかの判断が容易であるが，有向グラフの場合はそれほど容易ではない．具体的な内容は教科書を参考に解説を行う．

＜無向グラフの場合＞
・深さ優先探索を一度行った後，未訪問の頂点が残っていればそのグラフは連結でない．
・深さ優先探索の再帰呼出しの際に，訪問済みの頂点が存在すればそのグラフには閉路が存在する．

[*5] 連結グラフとはグラフ上の任意の 2 頂点間に道が存在するグラフのこと．
[*6] 深さ優先探索は，グラフを深い方へと進み，行き止まったら後戻りしてまた深い方へ進みながら頂点を探索していく方法である．
[*7] 幅優先探索はある階層をすべて調べ，それが終わると一つ深い階層を全て調べるということを繰り返す方法である．

3.2 幅優先探索

幅優先探索のアルゴリズムを以下に示す.

(1) 任意の頂点を訪問する.
(2) その頂点に隣接した頂点を全て訪問する.
(3) 全て訪問した後，訪問した頂点全てについて (2) の処理を行う.
(4) (3) の処理に戻る.

このアルゴリズムを実現するには，待ち行列 (キュー) を用いて次のように訪問すればよい.
(1) 最初に訪問する頂点を決め，その番号をキューに入れる.
(2) キューの先頭からノード番号を取り出し，その頂点を訪問する.
(3) その頂点に隣接した未訪問である全ての頂点の番号をキューに入れる.
(4) キューが空になるまで (2), (3) を繰り返す.

このアルゴリズムを実装するプログラムの雛形は以下のようになる.

```
void breadthFirstSearch(){
  最初に訪問するノードをキューに入れる
  while ( キューが空でない間 ){
    キューの先頭からノード currentNode を取り出す
    currentNode に対する処理を行う
    ( currentNode に隣接しているノード nextNode を探す ){
      調査中の辺に対して処理を行う
      nextNode が未訪問ならば nextNode を訪問済みとし，キューに入れる
    }
  }
}
```

＜計算量＞
深さ優先探索：リスト表現であれば $O(m+n)$，行列表現であれば $O(n^2)$
幅優先探索：リスト表現であれば $O(m+n)$，行列表現であれば $O(n^2)$

＜記憶域＞
深さ優先探索：$O(n)$ の記憶域が必要
幅優先探索：$O(n)$ の記憶域が必要

3.3 一般的な探索アルゴリズム

深さ優先探索は，前線[*8]の中から深さ最大の頂点，すなわち出発点から最も遠い頂点を選んで訪問する．逆に，幅優先探索は，前線の中から出発点に最も近い頂点を選んで訪問する.

前線上から頂点を選ぶときの基準をこれらとは違ったものにすると，別の種類の探索アルゴリズムが得られる．これをここでは一般的な探索アルゴリズムと呼ぶ.

一般的な探索アルゴリズムを実現するには，前線上の頂点を優先順位付き待ち行列で管理すればよい．優先順位付き待ち行列に対する挿入や取出しには $O(\log n)$ の計算量を要するから，このアルゴリズムの計算量は $O(n \log n + m)$ となる.

[*8] 訪問済みの頂点から到達できて，まだ訪問していない頂点の集合を前線とする.

簡単なまとめ

グラフの概念

・ネットワーク状の形をしたものを抽象化した概念
・現実の問題を解くためのモデルとして有効
・一般にいくつかの「もの」とそれらの間の「接続」ないし「関係」が与えられているとき，グラフで記述できる

用語の簡単な説明

重要な用語を分かりやすく図にすると，以下のようになる．

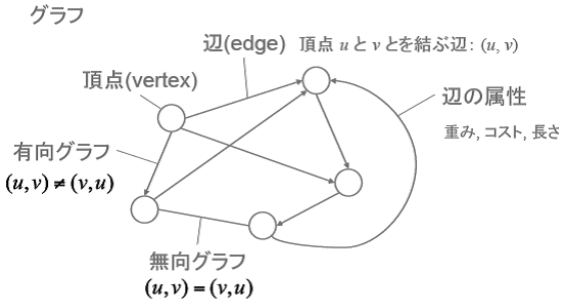

図 3　簡単な用語の説明

グラフと木の関係

グラフと木の関係を図にすると，以下のようになる．

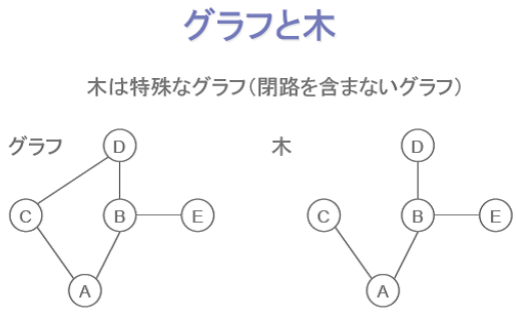

図 4　グラフと木の関係

グラフの種類

グラフには無向グラフと有向グラフがある．
無向グラフ：辺に向きがないグラフ
有向グラフ：辺に向きがあるグラフ

計算量

グラフを構成する頂点数を n，辺の数を m とする．

・無向グラフの場合：m は最大 $\frac{n(n-1)}{2}$
・有向グラフの場合：m は最大 $n(n-1)$　　⇒ 辺の数が最大のグラフを完全グラフという．

また，辺の数が多いグラフを密 (dense) なグラフ，少ないグラフを疎な (sparse) グラフと呼ぶ．

グラフに関するアルゴリズムの計算量の評価は，n と m に関するオーダーを求めることによって行う．

グラフの表現方法

グラフの表現方法には行列によるものとリストによるものがある．
・行列表現：隣接行列に頂点と頂点の接続関係を入れる
・リスト表現：各頂点を始点とする辺のリストを，頂点ごとに作る

グラフの探索

グラフの探索アルゴリズムの代表的なものとして，**深さ優先探索**と**幅優先探索**がある．

深さ優先探索は，グラフを深い方へと進み，行き止まったら後戻りしてまた深い方へ進みながら頂点を探索していく方法で，幅優先探索はある階層をすべて調べ，それが終わると一つ深い階層を全て調べるということを繰り返す方法である．

深さ優先探索の計算量

① リスト表現の場合
・各頂点について 1 回ずつ手続きを呼ぶ $O(n)$
・各辺を一回ずつ調べる $O(m)$
・深さ優先探索に必要な計算量は $O(n+m)$

② 行列表現の場合
・各頂点について 1 回ずつ手続きを呼ぶ $O(n)$
・辺は各頂点あたりに調べる $O(n)$
・深さ優先探索に必要な計算量は $O(n^2)$

幅優先探索の計算量と記憶域の大きさ

① リスト表現の場合
・各頂点について 1 回ずつ手続きを呼ぶ $O(n)$
・各辺を一回ずつ調べる $O(m)$
・探索に必要な計算量は $O(n+m)$
・記憶域の大きさは待ち行列の実現に必要な $O(n)$

② 深さ優先探索と幅優先探索の比較
・比較的枝分かれが少ない場合に深さ優先探索の記憶域の大きさは最悪
・枝分かれが多い場合に幅優先探索の記憶域の大きさは最悪

参考文献

[1] 石畑 清：アルゴリズムとデータ構造, 岩波書店 (2008)
[2] 佐藤 公男：グラフ理論入門, 日刊工業出版プロダクション (1999)
[3] J.A.Bondy & U.S.R.Murty：グラフ理論への入門, 共立出版株式会社 (1991)
[4] R.J. ウィルソン：グラフ理論入門, 近代科学社 (2004)