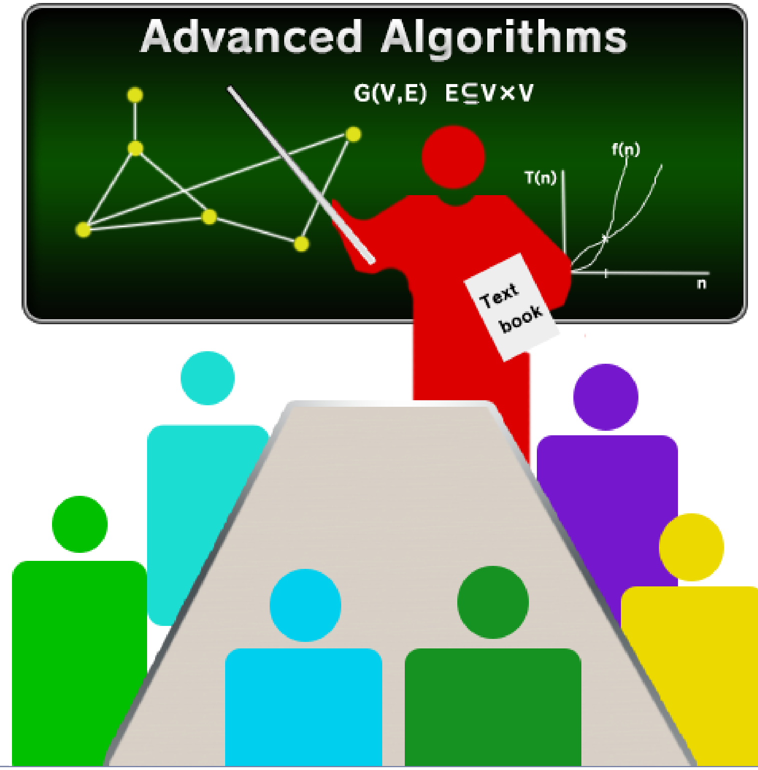


アルゴリズム特論 [AA201X] Advanced Algorithms

Lecture06. Algebraic Path Problem 〜 Transitive Closure

Exercise 06 のために

Algebraic Path Problem

様々なアルゴリズムを代数的な性質に基づいて**抽象化**

具体例：

- Warshall's Algorithm　（推移閉包）
- Floyd's Algorithm　（全頂点対最短経路）
- Maggs-Plotkin Algorithm　（最小全域木）

抽象化すると，**様々な解を求める異なるアルゴリズムが全部同じものに見える**（見なせる）ようになる

Algebraic Path Problem (APP)

今日のアルゴリズムは全て以下の形で書ける

```
for( k=0; k < n; k++){
    for( i=0; i < n; i++){
        for( j=0; j < n; j++){
            C[i][j] = C[i][j] ▪ C[i][k] ▪’ C[k][j] ;
        }
    }
}
```

C[ ][ ] は何者？ 演算 ▪ と ▪’ は何？

Algebraic Path Problem (APP)

APP ＝ 代数的経路問題

APPは，代数系「半環」(semi-ring) の定義に基づいた設計

・ Warshall's Algorithm　（推移閉包）
・ Floyd's Algorithm　（全頂点対最短経路）
・ Maggs-Plotkin Algorithm　（最小全域木）

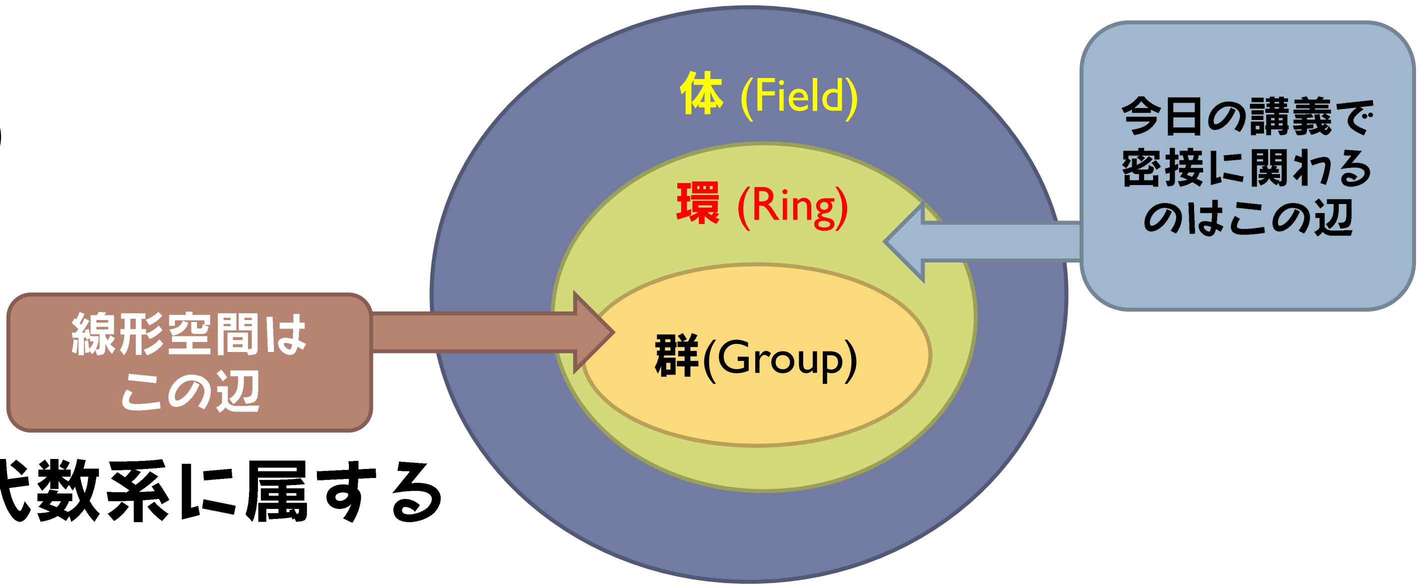
はいずれも APP に落とし込める！

代数系

- 数の集合と、ある演算が与えられるような問題
- 数学，物理学など科学全般　（小学校の算数すらも含む）
- コンピュータ，アルゴリズム

は、その性質に注目すると、大きく分けて

・群 (Group)
・環 (Ring)
・体 (Field)



と呼ばれる代数系に属する

代数系

- 普段なんとなく計算に利用しているかもしれない性質

・結合法則　$(1+2)+3 = 1+(2+3)$

・分配法則　$a*(b+c) = a*b + a*c$

・交換法則　$a + b = b + a$

・「逆数」の概念　$3 * 1/3 = 1$

などなど…は、便利な公式などではなく、その扱っている問題が

暗黙のうちに、それらの使用を許すような代数系の上で議論されている（その代数系における問題である以上、それらの性質を「満たさなければならない」と定義されている）から使ってもよいものである。

例）行列積は一般に、交換してはならない

代数系

- Warshall's Algorithm （推移閉包）
- Floyd's Algorithm （全頂点対最短経路）
- Maggs-Plotkin Algorithm （最小全域木）

はいずれも、「半環」の定義を満たしているゆえに
APPでまとめることができる。

環の定義（半環の前提）

集合 R と 第１算法・，第２算法・’ の３つ組で表される

(R, ・, ・’) が環であるとは…

・集合R が　第１算法に対して　可換群　をなす

・集合Rが　第２算法に対して　半群　をなす

・また、第１算法と第２算法の間で、分配律

a,b,c ∈ R, a・’(b ・c) = a・’b ・ a・’b を満たす

群 とは… 環よりもっと条件の緩い、基本的な代数系

群の定義（環の前提）

- 集合G，演算・に対して (G, ・)が群であるとは

1. 集合Gが演算・に対して閉じている

   $\forall a,b \in G,\ \ a \cdot b \in G$

2. 結合律を満たす

   $\forall a,b,c \in G,\ \ (a \cdot b) \cdot c = a \cdot (b \cdot c)$

3. 単位元が存在する

   $\forall a \in G, \exists e \in G \ \ \text{s.t.} \ \ a \cdot e = e \cdot a = a$

4. 各元に対して逆元が存在する

   $\forall a \in G, \exists x \in G \ \ \text{s.t.} \ \ a \cdot x = x \cdot a = e$

演算が閉じている（群が満たす公理１）

2つの要素を取り出して演算した結果は、必ず取り出した要素が属する集合に含まれている

例）

整数＋整数　＝整数

・<u>整数と整数の和は、小数には絶対にならない</u> <u>2+3=5など</u>

整数×整数　＝整数

複素数×複素数　＝複素数

有理数と有理数の２数の最大値　＝有理数

２つのベクトルの外積　＝ベクトル

２つのベクトルの内積　…×（スカラーになる）

結合律を満たす（群が満たす公理２）

- 小学校で習う、いわゆる「結合法則」
- 計算途中のカッコは、**カッコの位置をズラすだけなら自由にやってもよい**

(a ・ b) ・ c ＝ a ・ (b ・ c)

交換はダメ！！

(a・b) ・ c ＝ (b ・ a)・ c

単位元が存在する（群が満たす公理３）

- **演算しても結果を全く変えない要素**

整数Zの集合，加法＋の場合：　単位元は　０

3 + 0 = 3

2 + 0 = 2

-1 + 0 = -1

有理数Q（から0を除いた）の集合，乗法×の場合：　単位元は　1

1/2 × 1 = 1/2

3 × 1 = 3

-5/3 × 1 = -5/3

正方行列の場合：

・**加法の単位元　＝　零行列**

・**乗法の単位元　＝　単位行列**

各元に対して逆元が存在（群が満たす公理４）

- **ある元に演算をすると単位元になる元**

整数Z、加法+の場合：

3 + (-3) = 0,　　5 + (-5) = 0,　　-10 + 10 = 0

実数R（から0をのぞいたもの）、乗法×の場合：

4 × 1/4 = 1,　　√2 × 1/√2 = 1,　　2/5 × 5/2 = 1

可換群

- 「群」の定義の下では、「演算順序の交換」が不能

- 群にもう１個条件をプラスする
- 交換律を満たす　　$\forall a,b \in G,\ a \cdot b = b \cdot a$

- このとき、「可換群」と呼ばれる

- n次の正則行列たちを集合として、
- 乗法を演算とすると

群にはなるが、可換群にはならない

抽象化のポイント

- イメージする際には、具体的な演算（加法や乗法）などを考えてよい
- ただし、抽象化するときには「定義を満たすなら」集合や演算は何を設定してもよいのである

・最終的に、抽象的な定義に戻ってこなければ抽象化を勉強したことにはならない
（これがやろうとしない人＝これが理解できない人）

「ａとｂの２人がデートをする」という演算を ♡ として、a ♡ b　を定義して考察してもよいのである（代数系の定義を実際に満たせるかどうかは定義のしかた次第だが…）

環に戻すと…

・集合R が　第１算法に対して　**可換群**　をなす
・集合Rが　第２算法に対して　**半群**　をなす
・また、第１算法と第２算法の間で、**分配律**を満たす

「半群」は群の定義のうち、１・２のみを満たすもの
（**単位元・逆元は無くてもよい**、つまり、実際に活用する場合には、あるかどうかの保証がないということになる）

群は、小学校でやるような１つの演算だけを扱った話。
環（体）は中学校以上に出てくる複雑に演算が混合した話。

一応「体」の定義

・集合R が　第１算法に対して　**可換群**　をなす
・集合R が　第２算法に対して　**可換群**　をなす
・また、第１算法と第２算法の間で、**分配律**を満たす

つまり、実質**なんでもＯＫ**という状態。

一般的な数学の授業で扱う多くの問題はこれが前提。
暗号やチェックサムなど、情報理論・セキュリティの分野でも頻出。

APPのための「半環」の定義

・集合R が　第１算法に対して　可換 モノイド　をなす
・集合R が　第２算法に対して　モノイド　をなす
・また、第１算法と第２算法の間で、分配律を満たす

APP はこの上での議論である。
・ Warshall's Algorithm　（推移閉包）
・Floyd's Algorithm　（全頂点対最短経路）
・Maggs-Plotkin Algorithm　（最小全域木）
などは、これを満たすので全てまとめてしまえる

APP

APPにおけるアルゴリズムの場合、

数の集合（以下のコードでいえば二次元配列 C[][]）は アルゴリズムの動作に直接関わる値が入る。グラフ問題では「辺の重み」がくる。

あとは、半環の定義をクリアするような２つの演算を・と・'を適切に定めてやればよいだけである。

（どう定めればよいか、具体例は講義スライドを参照せよ）

```
for( k=0; k < n; k++){
    for( i=0; i < n; i++){
        for( j=0; j < n; j++){
            C[i][j] = C[i][j] ・ C[i][k] ・' C[k][j] ;
        }
    }
}
```

Transitive Closure

推移閉包

⇒（全ての頂点において、推移律 a〜b, b〜c ⇒ a〜c を満たすようにする）

APPに集合・演算を適用することで簡単に作れる。

集合⇒ 0か1 （隣接行列だから）

演算⇒ 第１：論理和、第２：論理積

```
for( k=0; k < n; k++){
    for( i=0; i < n; i++){
        for( j=0; j < n; j++){
            C[i][j] = C[i][j] |  C[i][k] & C[i][j];
    }
}
```

Transitive Closure

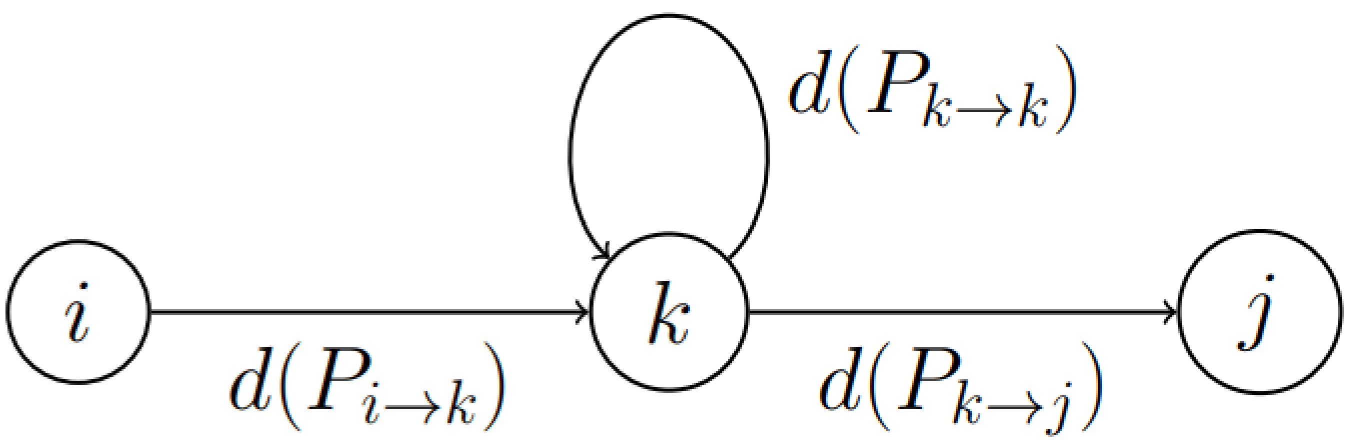
この講義でいうところの
推移閉包はどんな頂点も
「１つでも他の頂点を経由してから、
自分のところに戻って来れる」

理屈：

頂点 i から 頂点 j に向かって

・直接行けるか？

・他のいずれかの頂点 k を経由することで (i, k) → (k, j) と進む方法があるか？　⇒　あるなら「直接行ける扱い」にする



Transitive Closure

この講義でいうところの
推移閉包はどんな頂点も
「１つでも他の頂点を経由してから、
自分のところに戻って来れる」

Let $G(V, E)$ be a directed graph. $G^n(V, E^n)$ is defined by

$$E^n = \{(i, j) | \exists k \in V.[(i, k) \in E^{n-1} \wedge (k, j) \in E]\}$$

where $E^0 = \{(i, i) | i \in V\}$.

- transitive closure:

$$G^+ = G \cup G^2 \cup \cdots \cup G^n$$

そして、全ての辺 (i, j) について（全パターン）いけるかどうか検索する。

⇒結局、コードはすべての頂点に対しての３重ループとなる。

APPのコード全般

全てのｉとｊのペアに対して、

- (i, j)へ直接向かう
- 任意の k を経由して間接的に (i, k), (k, j) 向かう

- どちらをどのように採用するかは定義した演算しだい

本質的には、3重ループ＋１行の命令で簡単に書ける。

⇒ 汎用性が非常に高いため、簡単にいろいろなものを表せるが、処理の性質上、オーダーは 3乗 になる

⇒ 特定の問題を解くことに特化したアルゴリズムには性能は絶対に負ける

(例：最小全域木を求めることにおいて、Maggs-PlotkinではKruskalには勝てない)