情報科学 2011　久野クラス　＃11

久野　靖*

2011.1.13

あけましておめでとうございます。予告通りこの科目で出席を課すのは今回が最後で、あと 2 回ありますがそれらは出席自由とします。今回は動的計画法でちょっと頭を使うと思いますが、そういうわけで最後にひと頑張りしてください。なお、残りの内容ですが、次回#12 では Java 言語でプログラムを書いていただきますので、Ruby のついでに Java もちょっと書けるようになりたいという方はぜひいらしてください。#13 は予告したように、昨年度の試験問題の解説をしますが、今回の内容が済んだら試験範囲は全部カバーされていますので、**必ず昨年度の問題を全部自力でやってみてから**来るようにしてください。何も準備しないて来ても何も得るものはないと思います (本当に)。

ところで、本題の内容のウォームアップとして、次の問題を考えてみてください (紙にます目を描いて考えるのがよいです)。回答は本題部分で。

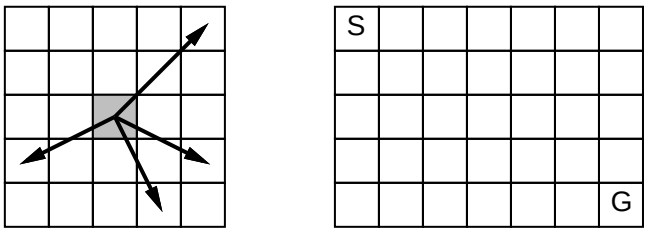

図 1: ロボットの動きと盤面

図 1 右のような盤面の S の位置にロボットがいて、G の位置まで移動したい。ただし、ロボットは「1 歩」で図 1 左のような動きしかできないものとする (そして盤面を外れる動きはできない)。G まで最短で何歩かかるか。

1　前回の演習問題の解説

1.1　演習 1-2 — 連結リストによるスタックとキュー

スタックとキューを単リストで実装するのは慣れれば簡単です:

```
Cell = Struct.new(:data, :next)

class Stack2
  def initialize
    @top = nil
  end
  def isempty
    return @top == nil
  end
  def push(x)
    @top = Cell.new(x, @top)
  end
```

*筑波大学大学院経営システム科学専攻

```
  def pop
    x = @top.data; @top = @top.next; return x
  end
end

class Queue2
  def initialize
    @top = @last = nil
  end
  def isempty
    return @top == nil
  end
  def enq(x)
    if isempty
      @top = @last = Cell.new(x, nil)
    else
      @last.next = Cell.new(x, nil); @last = @last.next
    end
  end
  def deq
    if @top == @last
      x = @top.data; @top = @last = nil; return x
    else
      x = @top.data; @top = @top.next; return x
    end
  end
end
```

キューの場合は「空っぽ」を特別扱いするのでちょっとだけ長くなりますね。

1.2 演習 3 — グラフをたどる

路線図のデータをどのように表現するかをまず考えましょう。ここでは、1 つの駅 (ノード) を次の 3 つのフィールドを持つレコードとします:

- `name` — 駅の名前
- `arr` — 隣接する駅 (ノード) を並べたもの
- `dist` — 出発駅からの距離 (不明なら-1)

そして、駅名からノード検索するためのハッシュ表をグローバル変数として用意します。路線図を図 2 に再掲しておきます。

この路線図のデータ構造を構築するメソッドを示します:

```
def prepare
  $graph = Hash.new()
  cn("赤羽", "池袋"); cn("赤羽", "田端"); cn("池袋", "田端")
  cn("八王子", "立川"); cn("立川", "新宿"); cn("池袋", "新宿")
  cn("新宿", "お茶の水"); cn("お茶の水", "秋葉原");
  cn("田端", "秋葉原"); cn("お茶の水", "東京")
  cn("秋葉原", "東京"); cn("東京", "品川"); cn("新宿", "大崎");
  cn("大崎", "品川"); cn("品川", "川崎"); cn("立川", "川崎");
```

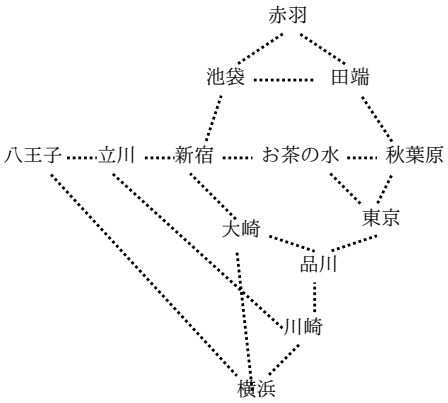

図 2: とある鉄道路線図

```
    cn("大崎", "横浜"); cn("川崎", "横浜"); cn("八王子", "横浜")
  end

  Node = Struct.new(:name, :arr, :dist)
  def cn(name1, name2)
    if $graph[name1] == nil then $graph[name1] = Node.new(name1, [], -1) end
    if $graph[name2] == nil then $graph[name2] = Node.new(name2, [], -1) end
    $graph[name1].arr.push($graph[name2])
    $graph[name2].arr.push($graph[name1])
  end
```

スタックを使う (深さ優先の) グラフの「たどり」は次のとおりです:

```
  def traverse1(start, goal)
    s = Stack1.new; n = $graph[start]; n.dist = 0; s.push(n)
    puts("START: #{start}")
    while !s.isempty do
      n = s.pop
      n.arr.length.times do |i|
        n1 = n.arr[i]
        if n1.dist < 0
          n1.dist = n.dist + 1
          puts("#{n1.dist}: #{n1.name}")
          if n1.name == goal then return else s.push(n1) end
        end
      end
    end
  end
```

これを幅優先にするには、キューを使うように取り換えるだけです (push は enq、pop は deq になります)。これらを動かすメソッドも用意しました:

```
  def test1
    prepare; traverse1('横浜', '池袋')
    puts('------')
    prepare; traverse2('横浜', '池袋')
  end
```

実行例を示します:

```
irb> test1
START: 横浜
1: 大崎
1: 川崎
1: 八王子
2: 立川
3: 新宿
4: 池袋
------
START: 横浜
1: 大崎
1: 川崎
1: 八王子
2: 新宿
2: 品川
2: 立川
3: 池袋
=> nil
```

確かに、深さ優先のほうが「さっさと」池袋に到達しますが、幅優先のほうが一番短い経路を発見できることが分かります。

演習 5 — オートマトンを解読する

これは解答だけ示しますので確認してみてください:

a. 「a が 0 個以上あり、b が 1 回あり、その後 a が 0 個以上ある」

b. 「まず a があり、その後 aa と ba が任意個、任意の順序で続いたもの」

c. 「a、abb、abbaa、abbaabb、abbaabbaa、…」言葉で言うなら、「まず a があり、その後 b2 個と a2 個が交互に現れる。a2 個や b2 個の途中以外ならどこで終わってもよい」

演習 6 — オートマトンを組み立てる

これは図 3 にオートマトンを示し、対応する`$atm` の値を以下に示します:

a. 「a が 1 個以上連続し」なので、初期状態からまず a が 1 個来た時の状態に進む必要がある。ここからは次は b でもよいし、a が何個あってもこの状態を続ければよい。b が来たら最終状態で、そこから先は遷移はない。

```
$atm = [{'a' => 1},
        {'a' => 1, 'b' => 2},
        {:final => true}]
```

b. 「a の連続は 2 個まで」なので、初期状態 (a がない状態)、a が 1 個の状態、a が 2 個連続の状態の 3 つを区別する必要がある。どこでも b が来たら初期状態 (a がない状態) に戻る。a が 2 個来た状態ではさらに a があってはいけないので a の遷移がない。すべての状態は最終状態であってよい。

```
$atm = [{'a' => 1, 'b' => 0, :final => true},
        {'a' => 2, 'b' => 0, :final => true},
        {'b' => 0, :final => true}]
```

c. これが実は一番簡単で、a が偶数個 (0 も含む) と奇数個の 2 つの状態だけあればよい。もちろん偶数個の状態 (初期状態) が最終状態。

```
$atm = [{'a' => 1, 'b' => 0, :final => true},
        {'a' => 0, 'b' => 1}]
```

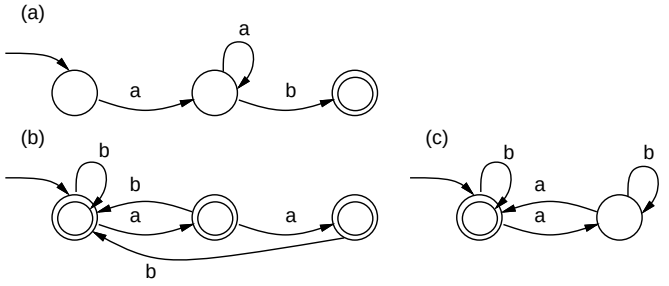

図 3: 演習 6 の有限オートマトン

2 動的計画法

2.1 イントロ: 冒頭の問題の解法

冒頭のロボット問題をやってみましたか。このような問題はどのようにしたら系統的に解くことができるでしょうか。枝分かれのある探索問題なので、前回と同じようにスタックやキューを用いて探索することもできますが、ややこしいですね。今回は少し違う次のやり方を考えてみます。

(1) まず、S から 1 歩で行けるところに「1」を記入。

(2) すべてのます目について検討し、そのます目に「N」が記入されていたら、そこから 1 歩で行けるところに「$N+1$」を新たに記入。ただし、その場所に $N+1$ 以下の値が既に記入されていたら記入はしない。

(3) 上記 (2) を繰り返して、それ以上新たなます目の記入が起こらなくなったら、おわり。

(4) G の場所に記入されている数値が答えの歩数。

図 4 にこの解法が進んで行く様子を示します。

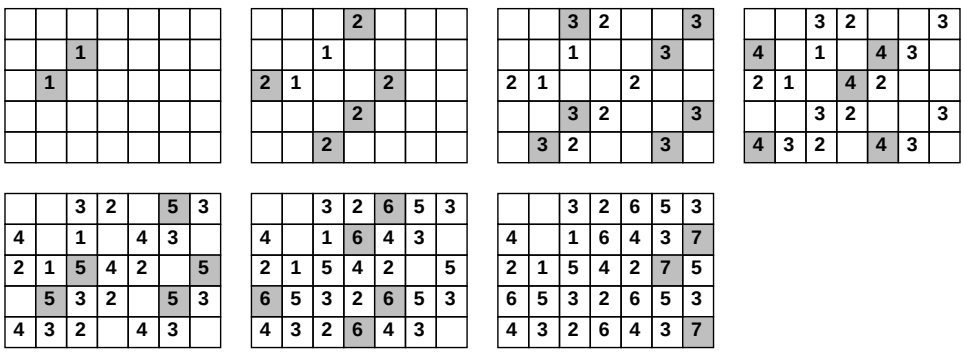

図 4: ます目を埋めていくロボット問題の解法

では、これを Ruby プログラムにしてみます。

```
def robot
  a = Array.new(5) do Array.new(7) do 0 end end
```

```
  a[2][1] = a[1][2] = 1
  done = false
  while !done do
    done = true
    5.times do |i|
      7.times do |j|
        [[2,1],[1,2],[-2,2],[1,-2]].each do |d|
          if move(a, i, j, i+d[0], j+d[1]) then done = false end
        end
      end
    end
    a.each do |b| p(b) end; puts "------"
  end
  return a[4][6]
end
def move(a, i, j, i1, j1)
  if a[i][j]==0 || i1<0 || i1>=5 || j1<0 || j1>=7 then return false end
  if a[i1][j1] > 0 && a[i1][j1] <= a[i][j] + 1 then return false end
  a[i1][j1] = a[i][j] + 1
  return true
end
```

最初にすべての要素が 0 の 2 次元配列を作り、1 歩目で行けるところに 1 を入れます。その後は、バブルソートでやったのと同じに、フラグを使って「これ以上変化がなくなるまで」繰り返し記入操作を行います。

記入操作の中では、配列の各要素について、そこから 4 通りの動きを検討します。4 通りの動きは、縦と横の動きをペアで入れた配列の配列で表し、ここから 1 つずつ動き (2 要素の配列) を取り出して、動きを検討する下請けメソッド move を呼びます。move では、「(i, j) に値が入っていないか、動き先が盤面外なら無視」「動き先に $N+1$ 以下の値が入っていれば無視」で、これらのチェックをパスしたら動き先に $N+1$ を入れて true を返します。メイン側では true が戻って来たら変化があったということなので旗を降ろします。

これを動かす様子を見てみましょう。図 4 よりも少ない回数で記入が終わっていますが、これはたとえば 2 のチェックの途中で 2 から 3 ができたら、その先で 3 から 4 もすぐ記入してしまうためです。

```
irb> robot
[0, 0, 3, 2, 0, 0, 3]
[0, 0, 1, 0, 4, 3, 0]
[2, 1, 5, 4, 2, 0, 5]
[0, 0, 3, 2, 0, 0, 3]
[4, 3, 2, 0, 4, 3, 0]
------
[0, 0, 3, 2, 6, 5, 3]
[4, 3, 1, 6, 4, 3, 7]
[2, 1, 5, 4, 2, 7, 5]
[6, 5, 3, 2, 6, 5, 3]
[4, 3, 2, 6, 4, 3, 7]
------
[0, 0, 3, 2, 6, 5, 3]
[4, 3, 1, 6, 4, 3, 7]
[2, 1, 5, 4, 2, 7, 5]
[6, 5, 3, 2, 6, 5, 3]
[4, 3, 2, 6, 4, 3, 7]
```

```
------
=> 7
irb>
```

このように、状態が変化しなくなるまで状態を更新していくことで解を求める、というのは最適化や求解のプログラムでよく出て来るやり方ですが、それをこのような問題でも使えるわけです。

ところで、このプログラムで何回も配列を走査していたのは、ロボットが右だけでなく左にも動くからだということに注意してください。もしもロボットが右方向にだけしか動かないなら、左から順にます目を処理していって、一番右まで来たらそれですべての処理は完了しているはずです。そうしたら、プログラムはずっと短くなり、効率もとても良いものになるわけです。これのようなプログラムについて、以下で取り上げています。

2.2 動的計画法とは

先に**フィボナッチ数**の計算を取り上げた時、次のような再帰的定義を示し、それをそのまま再帰関数にしたのでは遅すぎる、という説明をしました。遅すぎる理由は、この定義どおりだと 1 段階の再帰ごとに自分自身を 2 回呼び出し、同じパラメタに対する値を何回も重複して実行してしまうためでした。

$$fib(n) = \begin{cases} 1 & (n = 0 \; or \; n = 1) \\ fib(n-1) + fib(n-2) & (otherwise) \end{cases}$$

遅くなる理由である再計算を防ぐために、たとえば配列 `fib[`i`]` を用意して一度計算した値はそこに蓄え、2 回目からは計算しないでそれを持ってくる、という方法があります。一般に、関数を計算する時に、一度計算した結果を引数と一緒に覚えておいて、同じ引数に対しては覚えておいた値を返すようにすることを**メモ化** (memoization) と呼びます。

しかしそもそも、配列を使うのだったら、いちいち計算する代わりに、最大 30 番目までのフィボナッチ数だったら最初に順番に計算してしまい、それを参照するだけの方が分かりやすいはずです:

```
fib = Array.new(31); fib[0] = fib[1] = 1
2.step(30) do |i| fib[i] = fib[i-1] + fib[i-2] end
```

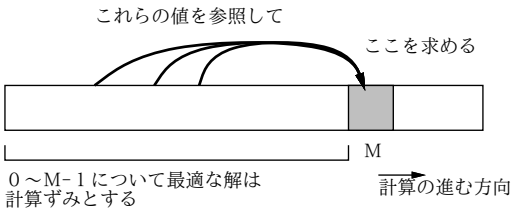

図 5: 動的計画法の考え方

これは図 5 のように、「値 $0 \sim M-1$ までについて解が求まっていれば値 M についての解もすぐ求まる」問題に対し、配列を用いて 0 から順に答えを埋めていくことで値 M に対する答えを求めていると言えます。一般にある問題に対して、その問題だけを解く代わりに小さい問題から順に全ての問題を答えを記録しつつ解くことで必要な解を求める手法のことを、**動的計画法** (dynamic programming) と呼びます。なお、これは単なる 1 つの手法であり、特別に動的でも特別にプログラミングでも何でもありません。この手法を考案した人がそういう名前をつけた、というだけです。他の方法では計算量が多すぎて扱えない問題が動的計画法によって効率よく扱える場合も多くあります。

2.3 部屋割り問題

動的計画法の適用例として、次のような問題を考えてみます。

> 合宿で 1 泊料金が「1 人部屋:5,000 円、3 人部屋:12,000 円、7 人部屋:20,000 円」というホテルに泊まることになった。[1] 合計宿泊人数 n 人に対し、最も安い宿泊金額総計を求めよ。

[1] 参加者は全員同性とします。各部屋には収容人数より少ない人数で泊まってもかまいません。また、どの部屋も数は十分あるものとします。

この問題では、7 人部屋が非常に割安なので、7 人より少ない人数で泊まっても 7 人部屋を選んだほうがよい場合があり、最適な割り当てを求めるのは簡単ではありません。この問題に限って言えば、できるだけ多く 7 人部屋を使って、残った 1〜6 人の場合について全部の場合を検討すれば済みますが、17 人部屋とか 31 人部屋とかもあったとすると大変すぎます。

そこで動的計画法を用いる準備として、人数 n に対して最も安い値段を計算する関数 $roomprice$ を次のように定義します。

$$roomprice(n) = minimumof \begin{cases} roomprice(n-1) + 5000 \\ roomprice(n-3) + 12000 \\ roomprice(n-7) + 20000 \end{cases}$$

$minimumof$ というのは聞いたことがないと思いますが (今発明したものなので当然です)、右側の選択肢のうち一番小さい値を取る、という意味のつもりです。なお、$roomprice(n)$ は $n \leq 0$ のときは 0 であるものとします (泊まる人数が 0 以下ならお金は掛かりませんから)。

なぜこれでいいかというと、n 人で泊まる時の最も安い方法は、「$n-1$ 人で泊まる時の最も安い場合に 1 人部屋を追加する」「$n-3$ 人で泊まる時の最も安い場合に 3 人部屋を追加する」「$n-7$ 人で泊まる時の最も安い場合に 7 人部屋を追加する」のうちのどれか 1 つに決まっているからです。

では、これを Ruby プログラムにしてみましょう。(添字 n まで使うため) 大きさ $n+1$ の配列 `roomprice` を作り、1〜n までの `i` について順次、3 つの場合の最小値を求めて入れて行きます。

```
def room(n)
  roomprice = Array.new(n+1, 0)
  1.step(n) do |i|
    min = roomprice[i-1]+5000;
    if min > roomprice[i-3]+12000 then min = roomprice[i-3]+12000 end
    if min > roomprice[i-7]+20000 then min = roomprice[i-7]+20000 end
    roomprice[i] = min
  end
  return roomprice[n]
end
```

ところで、このコードは「`n < 0` の場合は 0」をとくに扱っていませんが、いいのでしょうか? 実は Ruby では、配列の添字が −1、−2、…の場合は、配列の「一番最後」「最後から 2 番目」…の要素をアクセスするので、これらには (`n` が 7 以上なら) 初期値の 0 が入った状態なので値も 0 になります。このようなトリックはあまり褒められたものではないのですが、ここではあまりにコードが簡潔になるのでこのようにしてみました。では動かしてみましょう。

```
irb> room 10
=> 32000
irb> room 11
=> 37000
irb> room 12
=> 40000
```

なるほど、12 人だと 7 人部屋が 2 つの方が安いわけです。しかし、何人部屋がいくつなのかも知りたいですよね? そのためには、次の定義による値 $roomsel(n)$ も一緒に計算すればよいのです。

$$roomsel(n) = \begin{cases} 1 & (roomprice(n-1) + 5000 \ is \ the \ smallest) \\ 3 & (roomprice(n-3) + 12000 \ is \ the \ smallest) \\ 7 & (roomprice(n-7) + 20000 \ is \ the \ smallest) \end{cases}$$

この関数は、n 人のときに「最後に選んだ最適な部屋人数」を返します。選んだ部屋のリストを得るには、たとえば $roomsel(10)$ が 3 だったら、さらに $roomsel(7)$ を調べ、というふうに次々に「逆向きに」たどって行く必要があります。このため、こちらの情報のことを「トレースバック情報」と呼びます。では、先のメソッドを改造してトレースバックを記録し、金額に続いて部屋のリストを (1 つの配列として) 並べて返すようにしてみます。

```
def room1(n)
  roomprice = Array.new(n+1, 0)
  roomsel = Array.new(n+1, 0)
  1.step(n) do |i|
    min = roomprice[i-1]+5000; s=1
    if min > roomprice[i-3]+12000 then min = roomprice[i-3]+12000; s=3 end
    if min > roomprice[i-7]+20000 then min = roomprice[i-7]+20000; s=7 end
    roomprice[i] = min; roomsel[i] = s
  end
  a = [roomprice[n]]
  while n > 0 do a.push(roomsel[n]); n -= roomsel[n] end
  return a
end
```

これを動かすと、今度はちゃんと部屋の選択が分かります。

```
irb> room1 10
=> [32000, 3, 7]
irb> room1 11
=> [37000, 1, 3, 7]
irb> room1 12
=> [40000, 7, 7]
```

演習 1 上の例題を打ち込んでそのまま動かしなさい (最初はトレースバック無しの簡単な方を動かし、動いてからトレースバックを追加した方が楽だと思います)。動いたら、「13 人部屋 3 万円」「17 人部屋 4 万円」の選択肢を追加して動かしてみなさい。

演習 2 「最長増加部分列 (LIS, Longest Incerasing Sequence)」問題とは、長さ N の数列 $s(s_i > 0)$ が与えられた時、その (とびとびの) 部分列で、単調増加 $(i < j \rightarrow s_i < s_j)$ となるものの最大の長さを求める問題です。ここで、s の n 番目を最後の要素として必ず含むような LIS 長を与える関数 $longest(s, n)$ は次のように定義できます (これを元に $longest(s, 0)$〜$longest(s, N-1)$ のうち最大のものに相当する列を取り出せばそれが全体の LIS となります)。

$$longest(s,n) = maximumof \begin{cases} longest(s,i)+1 & (0 \le i < n \wedge s_i < s_n) \\ 1 & (no\ constraint) \end{cases}$$

そのココロは「$i < n \wedge s_i < s_n$ なる s_i で終わる最長の列の後に s_n をくっつけたもの (長さはプラス 1) が s_n で終わる最大列、ただしそのような i が見つからないときは s_n 単独で長さ 1 の列とする」ということです。この定義を用いて与えられた数列の LIS 長を求めるプログラムを作成しなさい。図 6 に「1,5,7,3,2,6,8,9,4」に対する計算の例を示しました (longlen はトレースバック情報の持ち方の例を示しています。これを参照すると、この例では最大増加部分列は「1,5,7,8,9」となります)。

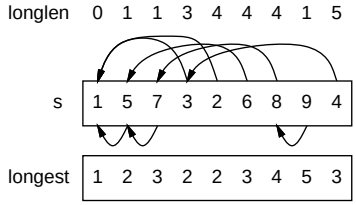

図 6: 最長増加部分列

3　パターン認識

3.1　パターン認識とは

パターン認識 (pattern recognition) とは、さまざまなデータの中にあるパターンを同定することを言い、次のものが代表的です: [2]

- 音声認識 — 「音」から「何を喋っているか」を抽出
- 画像認識 — 「画像」から「誰の写真か」「どこに何がある」などを抽出
- 文字認識 — 「文字の画像」や「書いている様子のデータ」から「何が書かれているか」を抽出[3]

パターン認識でいうパターンは、1 つのデータではなく、似通ったデータの集まりを意味しています。したがって、パターンへの「あてはまり」も YES/NO ではなく「どれくらい似ているか」を判断することが普通です。パターン認識が難しい理由として、次のようなものが挙げられます:

- あるパターンに属するデータに大きな多様性がある。「あ」という文字でも実にいろいろな書き方がありえる。
- パターン認識に用いるデータにはノイズが含まれていることがある。紙に汚れがついていたりした状態で文字を読み取るなど。
- パターンをどのような枠組みで捉えたらよいかの判断が難しい。[4]

人間がパターン認識を得意とするのは、多数の神経細胞がつながった回路に**学習** (learning) による情報が蓄積されているからとされています。そこでこれを真似して、人間の神経回路のような構造をプログラム上のデータ構造として構築し、それに学習を行わせてパターンを認識させる**ニューラルネットワーク** (neural networks) などの研究が多く行われています。以下では学習に基づくものではなく、アルゴリズム的なパターン認識を扱います。

3.2　文字列の編集距離

「どれくらい似ているか」を調べる例として、文字列の**編集距離** (edit distance) ないし**レーベンシュタイン距離** (Levenshtein distance)(レーベンシュタインはこの概念を提唱した人の名前です) と呼ばれる値の計算を取り上げましょう。これは、次のような問題だと言えます:

> 文字列 s を、エディタで編集して文字列 t になるように変更する時、文字の挿入、削除、置換 (書き換え) の回数は最少でいくつか。

この「最少書き換え回数」が少なければ文字列 s と t は「より似ている」と考えることができるので、これを編集距離と呼ぶわけです。

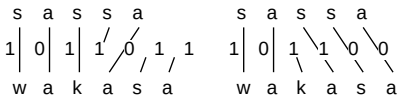

図 7: 編集距離の計算例

編集距離では各場面で「対応/置換」「挿入」「削除」の 3 とおりがあります。たとえば図 7 では「sassa」と「wakasa」の対応させ方を 2 とおり示していますが、左の場合は置換/挿入/削除が 4 箇所必要なのに対し、右の場合は 3 箇所で済みます。

[2] パターン認識の中には、人間には簡単だけれどプログラムにやらせるのは難しいことが多数あります。これらはその代表例でもあります。

[3] 「書いている様子のデータ」のことを**ストロークデータ** (stroke data) と呼びます。これは書き順や書く速さなども含んだデータなので、単なる書かれたものの画像よりも多くの情報が含まれています。

[4] たとえば音声認識であれば「音素」→「音節」→「単語」→「文」のような構造があり、一番下のパターン認識にも上の構造が参照されます。「きょうの・てんきは・□れ・ですね。」では「天気」という話題の範囲が分かっていれば「□れ」は「はれ」だろうと予想できます。

では、動的計画法を用いて編集距離を求めるための、長さ m の文字列 s と長さ n の文字列 t の編集距離 $editdist(s[0..m], t[0..n])$ の定義を示します。

$$editdist(s[0..m], t[0..n]) = minimumof \begin{cases} editdist(s[0..m-1], t[0..n-1]) & (s[m] = t[n]) \\ editdist(s[0..m-1], t[0..n-1]) + 1 & (s[m] \neq t[n]) \\ editdist(s[0..m-1], t[0..n]) + 1 & (no\ constraint) \\ editdist(s[0..m], t[0..n-1]) + 1 & (no\ constraint) \end{cases}$$

ただしこれに加えて、$editdist(空列, t[0..n]) = n$、$editdist(s[0..m], 空列) = m$ とします。これは次のように考えればよいでしょう。

- s を「左から順に」編集して t を作り出す場合、その一番最後の操作を考えると、s と t の最後の 1 文字ずつを対応させるか、s の最後の文字を削除するか、t の一番最後の文字を挿入するかのいずれかしかない。
- 対応させる場合は、編集回数は最後の 1 文字が同じならそれらの文字を除いた前の部分どうしの編集距離と等しく、同じでないならそれプラス 1 である。
- s の最後を削除するなら、s の最後を除いた部分と t との編集距離プラス 1 が編集回数となる。
- t の最後を挿入するなら、s と t の最後を除いた部分との編集距離プラス 1 が編集回数となる。
- 以上 3 つのうち最も小さいものを編集距離として採用する。
- 片方が空列なら、全部削除か全部挿入しかないので、編集距離は他方の文字列の長さに等しい。

今回はパラメタが m、n の 2 つあるので、これを動的計画法で計算する場合、2 次元の配列が必要になります (図 8)。2 つの次元はそれぞれの文字列の長さ+1 になるように取ります。

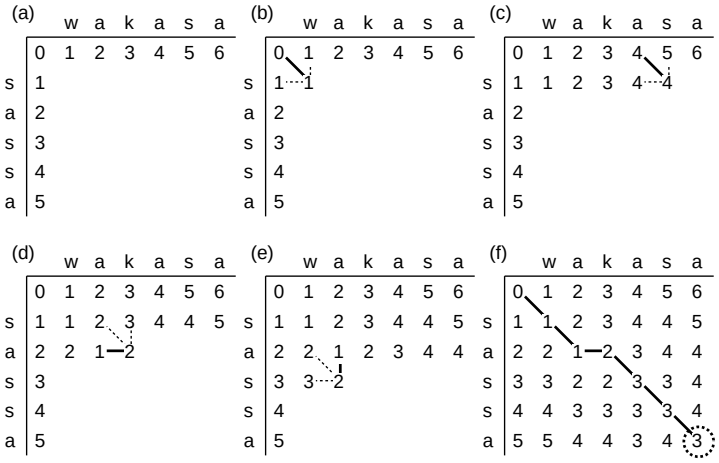

図 8: 動的計画法によって編集距離を求める

そしてこの配列の各要素を、`d[i][j]` が文字列 `s` の先頭から `i` 文字、`t` の先頭から `j` 文字までの編集距離であるように書き込んでいきます。それが完成したら、文字列全体の編集距離は `d[s.length][t.length]` を取り出せば分かるわけです。

まず片方が空列である場合の処理として、上端と左端の列を 0、1、2…で初期化します。続いて、各 `i`、`j` について、前述の 3 つの場合 (対応、挿入、削除) に基づいて編集距離を計算し、`d[i][j]` に入れます。最終的に一番端の `d[m][n]` の値が 2 つの文字列全体の編集距離となります。コードを示しましょう。

```
def editdist(s, t)
  d = Array.new(s.length+1) do |i| Array.new(t.length+1) do|j| i+j end end
  1.step(s.length) do |i|
    1.step(t.length) do |j|
      x = d[i-1][j-1]; if s[i-1] != t[j-1] then  x += 1 end
      if x > d[i][j-1]+1 then x = d[i][j-1]+1 end
```

```
        if x > d[i-1][j]+1 then x = d[i-1][j]+1 end
        d[i][j] = x
      end
    end
#   p d
    return d[s.length][t.length]
  end
```

2 次元配列の初期化のとき、各次元の添字 i、j を取得して i+j で初期化していますが、これは両端の部分を 0〜m,n で初期化するのにこれが簡単だからです。中央部分はどうせこれから書き換えるので、何が入っていても関係ありません。動かしてみましょう:

```
irb>  editdist 'wakasa', 'sassa'
=> 3
```

このコードでは 2 つの文字列をどのように対応させたかは出力されませんが、トレースバックを行えば対応づけが分かります。具体的には、各段階で選択肢 (対応、挿入、削除) のどれを選んだか記録しておいて、結果から逆向きにたどっていきます。

ここまでは文字列の対応づけとしていましたが、生物の**遺伝子** (gene) を構成する A/T/G/C の塩基の配列について、2 つの遺伝子の対応を求める**塩基列のアラインメント** (base sequence alignment) にもこれと同じ動的計画法が使用されます。この場合、「対応」「置換」「挿入/削除」それぞれにプラス/マイナスのコストを割り当て、コストが最小となるように対応を計算するという形になります。

演習 3 文字列の編集距離を計算するメソッドにバックトレースを表示する機能を追加せよ。表示方法はたとえば次のように、2 つの文字列を上と下に表示し、その間に (1) 同じ文字が対応するなら「|」、(2) 違う文字が対応する (書き換え) なら「:」、(3) 上の文字だけがある (削除) なら「u」、(4) 下の文字だけがある (挿入) なら「n」を表示する方法が考えられる。

```
w a k a s a
: | : u | |
s a s   s a
```

これ以外の表示方法を工夫してもよい。

演習 4 動的計画法を使わずに文字列の編集距離を求めるプログラムを作成し、動的計画法を使った場合との処理時間の比較と計算量の検討を行え。

演習 5 ファイルを編集した時、古いファイルと新しいファイルを見比べて「行単位で」どことどこを直したか[5]を調べたいことはよくある。そのようなプログラムを作れ。表示方法も工夫してほしい。[6]

3.3 隠れマルコフモデル

マルコフ過程 (Markov process) とは、系がいくつかの状態の間を遷移していく際、次の状態が現在の状態のみに依存する (過去の履歴には依存しない) ようなモデルです。たとえば図 9 は、最初状態 A から始まり、状態 A の次は確率 0.9 で状態 A、0.1 で状態 B、状態 B の次は確率 0.7 で状態 B、0.3 で状態 A になるというマルコフ過程を示しています。これからたとえば次のような状態の系列が得られるわけです:

```
AAAAABBBBBBBBBAAAAAAAAAAAAAAAAAAAAAABBAAAAAAAAAABBBBBBBBBBB
```

[5] どこに行を挿入し、どこの行を削除したかということです。

[6] 単純に 2 次元配列を使うと 2 つの列の長さが M、N とした時に領域計算量が $O(M \times N)$ になり、大きなファイルの比較はできません。しかしよく考えると、ある行 (や列) を計算する時にはその隣の行 (や列) だけあれば済むわけなので、領域はもっと節約できます。できれば、そのような工夫もチャレンジしてみてください。ただしトレースバック情報の持ち方も工夫する必要があります。

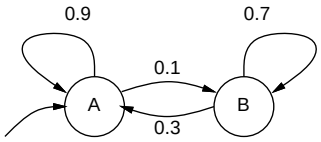

図 9: マルコフ過程の例

現実世界でマルコフ過程によってモデル化できることがらはいろいろありますが、そこでは「どの状態」という情報がそのまま観測できるのではなく、その状態によって「影響を受けた」情報が観測されるのが普通です。ここでたとえば、状態ごとに確率的に出力が決まるものとしたモデルを**隠れマルコフモデル** (hidden Markov model) と呼びます。なぜ「隠れ」かというと、状態そのものは隠されていて観測できないためです。

「いかさまカジノ」という例題を取り上げてみます。これは、たとえば上記のマルコフ過程において、状態 A では普通のサイコロ (1〜6 が各 $\frac{1}{6}$ の確率で出現)、状態 B ではいかさまサイコロ (6 が $\frac{1}{2}$ の確率、1〜5 が $\frac{1}{10}$ の確率で出現) が使われる、というモデルです。ここで観測されるのは現在選ばれているサイコロを振って出た出目の系列の情報で、各状態の出力が確率的に決まるため、隠れマルコフモデルになります。

「観測された出目」から、「何回目はどちらの状態 (どちらのサイコロ) が使われたか」を求める、というのがパターン認識の問題に相当します。たとえば手書き文字や発話の認識では、観測されるストロークや音は大きな多様整がありますが、これを「何という文字を書いている」「何と発音している」というモデルから確率的に現れたものと考えて、元となっている状態すなわち文字や発音内容を求めるわけです。

ここでデータにつける名前を少し整理しておきましょう:

- `input`[t] — t 番目の入力 (観測されたデータ)
- `init`[i] — 状態 i が最初の状態である確率
- `out`[i][c] — 状態 i で c を出力する確率
- `trans`[i][j] — 状態 i から状態 j へ遷移する確率

`init`、`out`、`trans` がマルコフモデルを記述していて、`input` は観測データであり、これらを用いて状態列を推定する、という問題設定とします。

なお、これとは別の問題として**パラメタ推定問題** (parameter inference problem)、つまりいくつかの出力列を与えてモデルのパラメタ (`init`、`out`、`trans`) を推定する、というのも重要な問題ですが、ここでは扱いません。

3.4 評価問題

状態列の推定の前に、類似の (しかしもう少し分かりやすい) 問題として、マルコフモデル M を前提として、ある観測データの列 w が与えられた時に、その列が出現する確率 $P(w|M)$ を求める、という問題を考えます。これを**評価問題** (evaluation problem) と呼びます。

ここでも動的計画法を使用し、配列 `a`[t][i] を「観測列 w の最初の $t-1$ 個を出力して状態 i に到達し、状態 i において t 番目のものを出力した確率」と考えます。最初にそれぞれの状態にいる確率は `init`[i] なので、そこで最初の文字を出力する確率は (各状態 i について) 次のようになります:

`a`[0][i] = `init`[i] × `out`[i][`input`[0]]

次の文字からは、それぞれの状態 j にいる確率 `a`[$t-1$][j] と状態 j からから状態 i に遷移する確率 `trans`[j][i] を掛けて合計することで状態 i にいる確率が求まるので、これを `init`[i] の代わりに使います:

`a`[t][i] = ($\sum_j$ `a`[$t-1$][j] × `trans`[j][i]) × `out`[i][`input`[t]]

一番最後に確率 $P(w|M)$ を求めるところでは、最後に各状態にいる確率の和を求めます (tt はデータの個数とします):

r = $\sum_j$ `a`[$tt-1$][i]

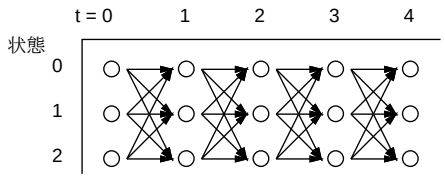

図 10: 前向きアルゴリズム

このとおり順番に計算するだけですが (図 10)、このアルゴリズムは**前向きアルゴリズム** (forward algorithm) と呼ばれます。Ruby コードを示します:

```
$ns = 2
$init = [1.0, 0.0]
$trans = [[0.9, 0.1], [0.3, 0.7]]
$out = [[1.0/6.0,1.0/6.0,1.0/6.0,1.0/6.0,1.0/6.0,1.0/6.0],
        [0.1,0.1,0.1,0.1,0.1,0.5]]
def calc(str)
  input = Array.new(str.length)
  input.length.times do |t| input[t] = str[t] - '1'[0] end
  a = Array.new(input.length) do Array.new($ns) end
  $ns.times do |i| a[0][i] = $init[i]*$out[i][input[0]] end
  1.step(input.length-1) do |t|
    $ns.times do |i|
      s = 0.0
      $ns.times do |j| s = s + a[t-1][j]*$trans[j][i] end
      a[t][i] = s*$out[i][input[t]]
    end
  end
  r = 0.0
  $ns.times do |i| r = r + a[input.length-1][i] end
  return r
end
```

3.5 復号化問題

いよいよ隠れマルコフの状態を推定する問題に進みましょう。これは「元の状態列→確率的データ→元の状態列の復元」という形になるので、たとえばノイズのあるチャネルで情報を送信し、受け取った側が元の情報を復元することに相当することから**復号化問題** (decode problem) とも呼ばれます。

その考え方は簡単で、先の評価問題では各状態から来る確率の合計を取っていたのに対し、こんどは「最も確率が大きい」1 つの状態を選んでそこから来たものと考えるだけです (図 11)。これは発案者の名前を取って**ビタビアルゴリズム** (Viterbi algorithm) と呼ばれています。これは実際に携帯電話の通信などノイズのある伝送路での通信に使われています。

こんどは計算用の配列名を d[t][i] とし、最初の値は先と同じです:

$$\texttt{d[0][}i\texttt{]} = \texttt{init[}i\texttt{]} \times \texttt{out[}i\texttt{][input[0]]}$$

2 番目以降は上述のように合計ではなく最大を選びます:

$$\texttt{d[}t\texttt{][}i\texttt{]} = (\ max_j\ \texttt{d[}t-1\texttt{][}j\texttt{]} \times \texttt{trans[}j\texttt{][}i\texttt{]}\ ) \times \texttt{out[}i\texttt{][input[}t\texttt{]]}$$

最後の値も最大を選びます:

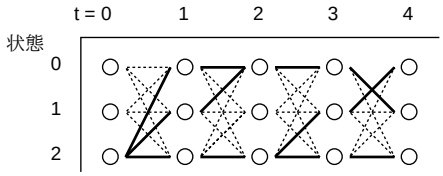

図 11: Viterbi のアルゴリズム

r = max_j d[$tt-1$][i]

実際には状態列が知りたいので、配列 prev[t][i] を用意しておき、各状態ごとにトレースバック用配列に「最大値を取った j の値」を記録していきます:

```
def viterbi(str)
  input = Array.new(str.length)
  input.length.times do |t| input[t] = str[t] - '0'[0] end
  d = Array.new(input.length) do Array.new($ns) end
  prev = Array.new(input.length) do Array.new($ns) end
  $ns.times do |i| d[0][i] = $init[i]*$out[i][input[0]] end
  1.step(input.length-1) do |t|
    max = 0.0; k = 0
    $ns.times do |i|
      $ns.times do |j|
        if d[t-1][j] * $trans[j][i] > max
          max = d[t-1][j] * $trans[j][i]; k = j
        end
      end
      d[t][i] = max * $out[i][input[t]]; prev[t][i] = k
    end
  end
  max = 0.0; k = 0
  $ns.times do |i|
    if d[input.length-1][i] > max
      max = d[input.length-1][i]; k = i
    end
  end
  out = ''
  (input.length-1).step(0, -1) do |i|
    out = '0123456789'[k..k] + out; k = prev[i][k]
  end
  return out
end
```

出力はトレースバック用配列をたどりながら文字列を「右から左に」連結して組み立てています。これを動かした例を見てみましょう (番号を 0 から始まるようにしたので、0 がダイスの「1」、5 がダイスの「6」に対応しています)。

```
irb> viterbi '04123151553552513421'
=> "00000011111111100000"
```

確かに「6」が多いあたりは「いかさまダイス」状態と推定されるようです。

演習 6 「いかさまダイス」のデータを乱数で生成し、それをビタビアルゴリズムで復号して、どれくらい正しく状態 (いかさまダイスか否か) が推定できるか検討せよ。

演習 7 図 12 のような「-」「0」「1」という 3 つの状態を持つマルコフモデルを考える。0 と 1 の並んだ列を与え、先頭

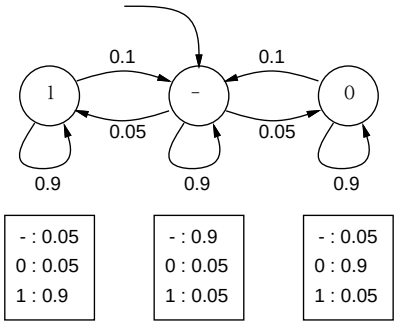

図 12: 練習問題のマルコフモデル

と末尾と各文字の間に「-」を挿入し、それぞれの状態を 9 回繰り返したら次に行くようにしてデータを生成した後、確率 r でランダムに文を別の文字に取り換える (つまりノイズを入れる)。その結果をビタビアルゴリズムで復号してから、連続しない「- 」「0」「1」の列を削除した上で連続した「0」「1」の列をそれぞれ 1 個の「0」「1」にすることで、元の 0 と 1 の列が正しく復元できるか試してみよ。[7]

A 本日の課題 11A

演習問題から 1 つ選び、動かしたプログラムを含む小レポートを今日中に久野までメールで送ってください。

1. Subject: は「Report 11A」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 動かしたプログラムどれか 1 つのソース。
4. 以下のアンケートの回答 (Ruby 終了なので今回限定でない質問です)。

   Q1. 「動的計画法」の考え方は分かりましたか? 今後自力で使えそう?

   Q2. この講義は「情報科学」の学習を目的とし、その確認手段として「Ruby によるプログラミング」を演習して頂いて来ました。本クラスにおけるその両者の位置づけやバランス、難易度、授業全体としてのよしあしについて考えをお聞かせください。

   Q3. 講義開始時の自分を振り返り、今回までの講義+演習で自分がどんなことを学んだか簡単にまとめて見てください (全体的な感想も含めてどうぞ)。なお、参考までにこれまで各回の主要な内容を挙げておきます。

1	アルゴリズムとプログラム、数値の表現、浮動小数点と誤差
2	制御構造、枝分かれ、繰り返し、数値積分、計数ループ
3	制御構造の組合せ、$f(x) = 0$ の求解、データ型、配列
4	手続きと抽象化、整列、再帰関数、レコード型と画像
5	様々な整列アルゴリズム、時間計算量、各種アルゴリズムの別バージョン
6	連立方程式の数値解法、消去法、反復法、常微分方程式の数値解法
7	オブジェクト指向、クラス定義、乱数、ランダムアルゴリズム
8	動的データ構造、連結リスト、表と探索、二分探索木、連想配列
9	式木、抽象構文木、継承、言語処理系、字句解析、BNF、構文解析
10	スタック、キュー、構造のたどり、オートマトン、状態空間の探索
11	動的計画法、パターン認識、隠れマルコフモデル、ビタビアルゴリズム

[7] どの程度までノイズがあっても復元できるかも検討してください。