情報科学 2007　久野クラス　＃10

久野　靖*

2007.12.21

はじめに

前回の内容のうち、文法と解析器はちょっとヘビーだったようで、おつかれ様でした。今回はもうちょっと身近に親しめそうな題材を取り上げますので。

- スタックとキュー — 汎用的な抽象データ型
- データ構造のたどり、状態空間の探索

グラフの探索は路線図みたいなものを扱えますし、状態空間の探索ではパズルみたいなものが扱えます。なお、前回課題の回答例は期限が次回なので次回に掲載しますね。

1　スタックとキュー

1.1　スタック

スタック (stack) とはコンピュータのアルゴリズムで多く使われる汎用の抽象データ型であり、「**LIFO**(last-in, first-out) の記憶領域」とも呼ばれる。つまり、スタックには次々にデータを入れて置けるが、取り出す時には (まだ取り出されてしまってないもののうち) 一番最近に入れたものが取り出されてくる。これはちょうど、ものを上に積んでいって取り出す時の動作を同じなのでスタック (積む) と呼ばれる。スタックの入れる/取る動作は伝統的に push/pop と呼ばれる (図 1 左)。

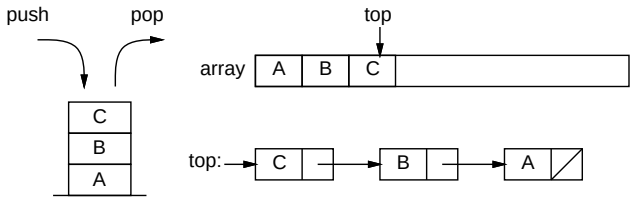

図 1: スタック

スタックの実現方法の 1 つは、配列を用意して、そこに順番に push された要素を詰めていくことである。先頭の位置は変数 `ptr` に覚えておく (図 1 右上)。この方法はシンプルだが、多くのプログラミング言語では配列のサイズは最初に作る時に指定してその大きさが変化しないため、この場合最大いくつまで要素を格納するか予め決めておく必要がある。Ruby では…配列にそもそも `push` とか `pop` とかメソッドがあって何も作る必要がないのだが:-)、まあ練習なのでここでは固定サイズの配列でスタックを実現してみる。

スタックのもう 1 つの実現方法は、連結リストを用いて要素を覚えておく方法である (図 1 右下)。この方法では最大要素数を指定する必要は特にない。

配列を使ったスタックの実現と、それを使って文字列を記憶してみるサンプルを示す。

*筑波大学大学院経営システム科学専攻

```
class Stack1
  def initialize
    @arr = Array.new(100); @ptr = -1
  end
  def isempty
    return @ptr < 0
  end
  def push(x)
    @ptr = @ptr + 1; @arr[@ptr] = x
  end
  def pop
    x = @arr[@ptr]; @ptr = @ptr - 1; return x
  end
end
def test1(stk)
  while true do
    printf('> ')
    s = gets.to_s[0..-2]
    if s == 'q' then return
    elsif s == '' then puts(stk.pop)
    else               stk.push(s)
    end
  end
end
```

動かす様子を見てみよう。

```
irb(main):005:0> test1(Stack1.new)
> A  ←入れる
> B  ←入れる
> C  ←入れる
>    ←リターンで
C    ←取り出し表示
>
B    ← B まで取り出し
> D  ← 3
> E  ←つ
> F  ←積む
>    ←全部取り出し
F
>
E
>
D
>
A     ←最初の「A」が最後に出る
> q   ←「q」で終わり
=> nil
```

演習 1 これを打ち込んで動かせ。動いたら、連結リストを使ったスタックの実現 Stack2 を作り、上の例題で同じに動作することを確認せよ。

1.2 キュー

キュー (queue) もスタックと類似の汎用抽象データ型だが、違うのは取り出すときに、もっとも始めに近く入ったもの (でまだ取り出されていないもの) が取り出されることである。これを「**FIFO**(fist-in, first-out) のデータ構造」とも呼ぶ。キューとはおなじみ「行列」を意味する英語である (もちろん最初に並んだ人が最初にサービスしてもらえなかったら怒りますよね…)。キューの入れる操作/取り出す操作は伝統的に enq/deq と呼ばれる (2 左)。[1]

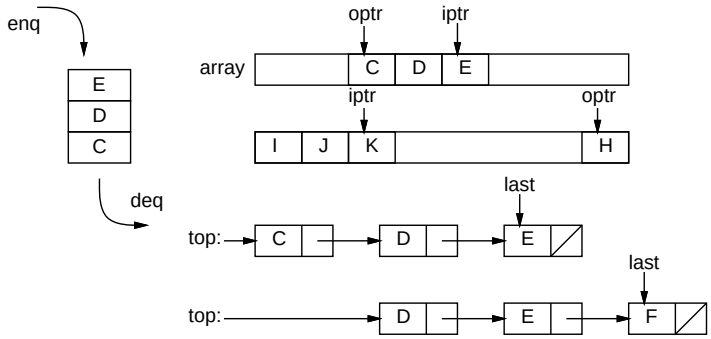

図 2: キュー

キューの実現も配列版と連結リスト版が考えられる。配列版では、`iptr` と `optr` という 2 つの変数で「入れる場所」と「取り出す場所」を覚えておく。そうすると、ずっと入れて出してしていくと入っている場所が配列の端まで来るので、ぐるっと回って次は先頭に戻るようにする必要がある (図 2 右上)。満杯になった/空っぽになったことを知るために、現在いくつ入っているかを別の変数で覚えておくのが普通 (別の方法として、`ipt` が `opt` に追い付きそうになったら満杯だと判断する方法もあるが、その場合は 1 箇所は空けて置かないと空っぽの状態と区別がつかなくなる)。

連結リスト版は、リストの先端を `last` で指しておいて、そこに新しい要素を追加するようにする (図 2 右下)。ただし、リストが空っぽ (`top` が `null` の場合は `last` は意味を持たない (セルが 1 個もないのだから) ので入れる時に特別扱いになる。

以下には配列版のキューのプログラムを示しておく (`test2` の使い方は先の例と同じ)。

```
class Queue1
  def initialize
    @arr = Array.new(100); @count = 0; @iptr = -1; @optr = 0
  end
  def isempty
    return @count <= 0
  end
  def enq(x)
    if @count + 1 >= @arr.length then return end
    @iptr = (@iptr+1)%@arr.length; @arr[@iptr] = x;
    @count = @count + 1
  end
  def deq
    if @count <= 0 then return end
    x = @arr[@optr]; @count = @count - 1;
    @optr = (@optr+1)%@arr.length; return x
  end
end
def test2(que)
  while true do
```

[1] 実はフルスペルは enqueue と dequeue だが長いし発音が同じなので短く書くのが普通。

```
    printf('> ')
    s = gets.to_s[0..-2]
    if s == 'q' then return
    elsif s == '' then puts(que.deq)
    else               que.enq(s)
    end
  end
end
```

演習 2 これを打ち込んで動かせ。動いたら、連結リストを使ったキューの実現 Queue2 を作り、上の例題で同じに動作することを確認せよ。

1.3 スタックとキューを使った構造のたどり

スタックやキューはどういう役に立つのだろう。それは、何かを入れて (覚えて) おいて、後で取り出して使うから。当り前だと思うかもしれないが、こういうことは結構ある。そして、単独の変数に覚えておくのだと、1 つしか覚えておけない (2 つ目を入れると前に入っていたものは上書きされて失われる)。配列だと、「どこに」入れてあるかを覚えておかないと役に立たない。ところが、スタックやキューでは「とにかく入れて」「とにかく取り出す」ことができるので簡単であり、そして入れたものは必ずいつか出て来ることが保証される。

たとえば、前回の式木では exec や to_s で再帰を使ってたどりながら計算や文字列化を行っていたが、これらを使わずにスタックやキューで式木をたどってみよう。以下の式木の構造は前回と同じ (exec や to_s は使わないが削るのも面倒なのでそのまま入れてある)。

```
class Node
  def initialize(l=nil, r=nil) @left = l; @right = r; @op = '?' end
  def to_s() return '(' + @left.to_s + @op.to_s + @right.to_s + ')' end
  def getleft() return @left end
  def getright() return @right end
  def getop() return @op end
end
class Add < Node
  def initialize(l, r) super; @op = '+' end
  def exec() return @left.exec + @right.exec end
end
class Mul < Node
  def initialize(l, r) super; @op = '*' end
  def exec() return @left.exec * @right.exec end
end
class Lit < Node
  def initialize(v) super; @op = '#' end
  def exec() return @left end
end
class Var < Node
  def initialize(v) super; @op = '$' end
  def exec() return $vars[@left] end
end
```

スタックを使ってたどる例。最初に根のノードをスタックに入れ、あとは「取り出して、処理し、子供があれば子供を積む」。

```
def test3
  e = Add.new(Mul.new(Lit.new(3), Var.new('x')), Var.new('y'))
  stk = Stack1.new; stk.push(e)
  while !stk.isempty do
    node = stk.pop
    if node.getop == '$' || node.getop == '#'
      puts(node.getleft)
    else
      puts(node.getop)
      stk.push(node.getleft); stk.push(node.getright)
    end
  end
end
```

なお、左と右の子をスタックに入れるとき、右を先に入れるのは、取り出す時左から取り出された方が見やすいから。実行のようすを見てみよう。

```
irb(main):002:0> test3
+
*
3
x
y
=> nil
```

まず演算が表示され、続いてその演算の子供が左、右の順で表示される。では、キューだとどうだろうか。プログラムは入れるものがキューになるだけ。

```
def test4
  e = Add.new(Mul.new(Lit.new(3), Var.new('x')), Var.new('y'))
  que = Queue1.new; que.enq(e)
  while !que.isempty do
    node = que.deq
    if node.getop == '$' || node.getop == '#'
      puts(node.getleft)
    else
      puts(node.getop)
      que.enq(node.getleft); que.enq(node.getright)
    end
  end
end
```

このコードを実行すると次のようになる。

```
irb(main):009:0> test4
+
*
y
3
x
=> nil
```

これはどういう順だろう? 実は、スタックや再帰関数を使ったたどりでは「まず枝をどんどん深い方にたどり、行き止まりになったら一番最近の枝分かれまで戻って来て次の枝をどんどん深い方にたどり、…」という順番になる。これを**深さ優先のたどり** (depth-first traversal) という。これに対し、キューを使うと「まず 1 レベル目を全部順番にたどり、次に 2 レベル目を全部順番にたどり、…」という形になる。これを**幅優先のたどり** (breadth-first traversal) と呼ぶ (図 3)。幅優先のたどりを使うと「木の中にある何か」を探すことで、その「何か」のうちで木の一番浅いところにあるものが見つけられるという特徴がある。

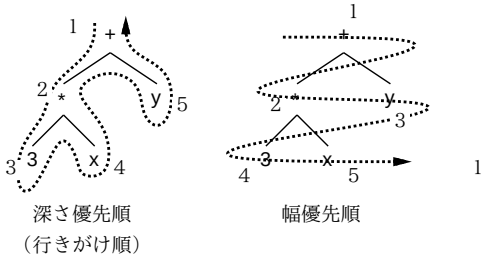

図 3: 深さ優先と幅優先

演習 3 図 4 はとある地域の鉄道路線図のようなものである (架空のものであり現実の場所とは関係ありません)。これを表すデータ構造を設計して作れ。またそのデータをたどって「東京」から「八王子」、または「横浜」から「赤羽」の経路を出力するプログラムを作れ。スタック版とキュー版で比較すること。(ヒント: ノード (分岐駅) ごとにレコードを作り、隣接ノードの情報を格納する。最初に出発ノードをスタック (キュー) に入れ、以後 1 つ取り出してそれがゴールでないならその隣接駅をすべてスタック (キュー) 入れる。ただし同じ場所を堂々巡りにならないために、「訪問ずみ」を表すフィールドを設け、最初にスタック (キュー) に入れる時に `true` にして、以後到達したノードでこれが `true` のものは処理ずみなのでスタック (キュー) に入れないようにする。)

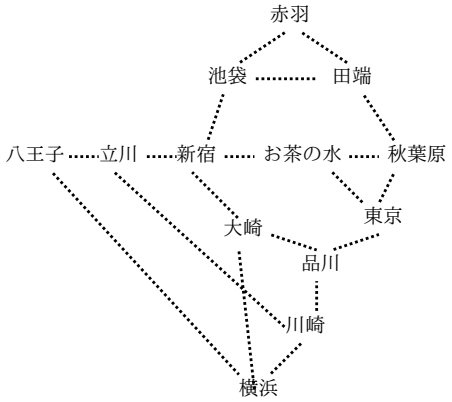

図 4: 鉄道路線図

演習 4 エディタバッファのデータ構造として、次のようなものも考えられる。

- 2 つのスタック A、B に「現在位置より上の行」「現在位置以降の行」を入れるものとする。
- スタック B の一番上が現在行と考える。
- 1 つ下に行くには、スタック B から 1 つ pop してそれをスタック A に push。
- 1 つ上に行くには、スタック A から 1 つ pop してそれをスタック B に push。
- 現在行を消すには、スタック B から 1 つ pop。
- 新しい行を挿入するには、スタック A に 1 つ push。

この方針によるエディタを作ってみよ。コマンドや機能の設計は各自に任せる。

2 状態空間の探索

2.1 状態と状態遷移

我々の日常生活において、ある同じ行動をしても結果が異なることはよくある。たとえば大学のキャンパスで友人と談笑するのは構わないが、授業中の講義室ではまずい。つまり談笑という同じ行動を取っても、文脈 (context) が違うと結果が違うわけである。ここで、同等と見なせる文脈の集合を**状態** (state) と呼ぶことにする。つまり講義中という状態、映画館で映画を見ている状態、普段の状態、などの状態があって、「どの状態である」かに応じて「行動の結果」が変化すると考えるわけである。

実はコンピュータ上の世界は日常生活よりもいっそうはっきりと「状態」の違いを意識する必要がある。同じコンピュータの前に座っていて同じようにキーボードで「ls[ENTER]」と打っても、フォーカスがターミナル (コマンド窓) にあるのかテキストエディタにあるのかで全然結果が違うでしょう? そして、プログラムの中もまた同様である。たとえば `x + 1` という式の値は、変数 `x` に入っている値が 4 であれば 5 になるし、10 であれば 11 になる。だから、変数の値を書き換えつつ計算が進行していく手続き型言語の場合、状態とは「すべての変数の値の集合」ということになる。

状態について考えるときは、ある状態から別の状態への移行、つまり**状態遷移** (state transition) も一緒に考える必要がある。これは、どのような場合にどの状態からどの状態へ移る、という定式化がなければ、現在いる状態はどれなのかも分からないことになるからである。たとえば、先の演習にあった路線図の問題の場合、「現在どの駅にいるか」が状態であり、「ある駅から (電車に乗って) 別の駅へ移動する」ことが状態遷移ということになる。

2.2 有限オートマトン

状態と状態遷移をモデル化する手法の 1 つに**有限状態オートマトン** (finite state automata) がある。以下では、「ある文字列がある規則にあてはまっているかどうか調べる」という問題を題材として有限オートマトンを紹介しよう。まずは規則として「文字列は a から始まり、a と b が交互に現れ、最後は a で終わる」というのを例に取る。このような規則へのあてはまりを調べるのに、5 のようなグラフ (有限オートマトン) を使うことができる。

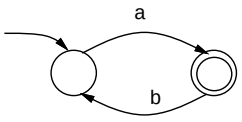

図 5: 有限オートマトンの例

ここで、グラフは○と◎の 2 種類のノードからできている。各ノードは 1 つの状態を表す。最初は、外から矢線が来ている状態にいるものとする (初期状態)。そしてここから、入力を 1 文字ずつ取り出し、その文字に対応するラベルを持った矢線をたどって次の状態へ遷移することを繰り返す。その結果、最後の文字まで終わった時に◎の状態にいれば「OK」であることを意味する。この「OK」であることをオートマトンの用語では「受理」、◎の状態のことを受理状態という。◎以外の状態にいたり、途中で文字に対応する矢線が無かったりした場合は「OK でない」ものとする。図 5 の場合、ちょっと考えてみれば「a から始まり、a と b が交互に現れ、最後は a」という規則にあてはまった文字列のみを受理し、受理した場合は必ず文字列は規則にあてはまっていると納得できますね?

ではこれを Ruby プログラムで表すことを考えてみよう。それぞれの状態を整数で表し、初期状態は 0 番であるものとする。次の状態への矢印を表すため、各状態は 1 つのハッシュとし、ラベルの文字に対して次の状態番号を返すようにする。たとえば、図 5 の場合、0 番の状態は (a が来たら 1 番へ行くわけだから) 次のようなハッシュで表せる。

```
{'a' => 1}
```

さらに、各状態が最終状態か否かも記録する必要があるが、これも一緒のハッシュに入れることにして、`:final` という記号値に対応する値が `true` の場合は最終状態だということにする。すると、図 5 の場合、1 番の状態は次のようになる。

```
{'b' => 0, :final => true}
```

これらを配列としてまとめたものを、広域変数$atm に入れておくことにしよう。文字列を与えてオートマトンが受理するか否かを調べるメソッド accept と一緒に示す (短いでしょ?)。

```
$atm = [{'a' => 1},
        {'b' => 0, :final => true}]
def accept(s)
  cur = 0
  s.length.times do |i|
    k = $atm[cur][s[i..i]];
    if k == nil then return false else cur = k end
  end
  return $atm[cur][:final] == true
end
```

accept では cur に現在の状態番号を保持するようにしたので、最初は 0 から始める。そして文字列の 0 番目、1 番目、… の文字について順に、「オートマトンの現在の状態のハッシュに対してその文字を検索」した結果を変数 k に入れる。これが nil であれば「矢線がない文字」だったのでただちに不受理 (false) を返す。そうでなければ状態を k にしてさらに続ける。最後まで終わったときに現在状態が最終状態であれば受理 (true)、そうでなければ不受理を返す。ではやってみよう。

```
irb(main):013:0> accept('aba')
=> true
irb(main):014:0> accept('ababa')
=> true
irb(main):015:0> accept('ab')
=> false
irb(main):016:0> accept('abba')
=> false
```

確かに先の規則にあてはまるものだけが受理できている。

演習 5 上のコードをそのまま打ち込んで動かせ。動いたら、図 6 の (a)〜(c) について、まずこれらのオートマトンがどのような文字を受理するかを考えて紙に書き、続いて上のコードのオートマトンのデータを変更して動かし確認せよ。

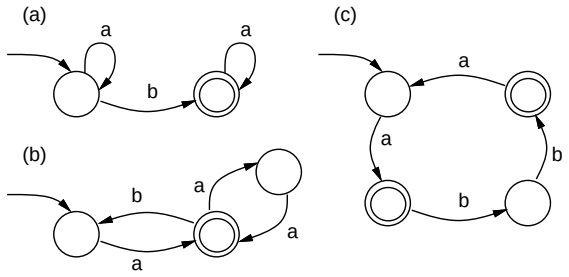

図 6: 演習 5 の有限オートマトン

演習 6 次のような文字列「のみ」を受理するオートマトンを考え、まず紙に描いてみよ。続いて、上のコードのオートマトンのデータを変更して動かし確認せよ。

a. a が 1 個以上連続し、最後に 1 個だけ b がある。

b. a と b がさまざまに混ざって現れるが、a の連続は最大 2 個まで。

c. a と b がさまざまに混ざって現れるが、a の個数は偶数個。

2.3 ゲーム・パズルと状態空間

カードや駒などを使う多くのゲームやパズルは、個々の場面を 1 つの状態と考え、プレーヤが選択する「手」を状態遷移と考えることで定式化できる。たとえばソリティア (トランプの一人遊び) では場のカードの内容 (どれが表/裏かも含む) と山のカードの内容全部を合わせたものが 1 つの状態であり、プレーヤが場のカードを移動したり次のカードをめくってどこかに置いたりするのが状態遷移となる。

このとき、有限オートマトンと違うのは、有限オートマトンでは比較的少数の状態が予め分かっていてそれを全部用意して扱っていたのに対し、ゲームやパズルでは可能な状態の数が非常に多く、それを全部生成しておくのは非現実的だという点である。そうではなく、「手」を打つごとに次の状態をその場で生成して行き、目指す状態 (自分が勝利したり、パズルが解けたり) が見つかったらそこでおしまい、という処理が必要である。これを**状態空間の探索**という。

簡単なゲームの例として、図 7 にマルバツの状態空間の始めの方を描いてみた (×が先手であるものとした)。マス目が 9 個あり、それぞれに (1) ○が入る、(2) ×が入る、(3) まだどちらも入っていない、の 3 つがあるのだから、最大で $3^9 = 19683$ 通りの状態があることになる。が、実際には途中でどちらかが勝ったらおしまいだし、上下/左右/ 点対称のものは分けなくてもいいわけだから、扱う状態はずっと少なくて済むわけだ。

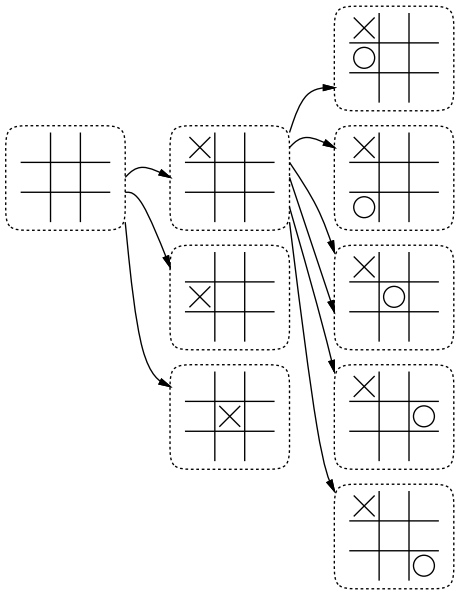

図 7: マルバツの状態空間 (一部)

全部扱えてしまえば、その中で「勝ち筋」をたどっていくだけで勝てるようになる (または勝ち筋がなければ引き分けをめざすが、途中で相手が失策したら勝ち筋に乗れるかも知れない)。一方、囲碁とか将棋とかだと、状態の数は本当にすごく多くなるので、これをコンピュータで扱うのはまだまだ大変な状態である。

スタックを用いた深さ優先探索の場合、状態空間探索の基本的なアルゴリズムは次の形になる。

```
s = 初期状態; mark(s); stack.push(s)
while true do
  if stack.isempty then 全状態を生成したがゴールに到達できなかった! end
  s = stack.pop
  (s に隣接ずる状態を生成) do |s1|
    if s1.isgoal then ゴールに到達した!
    elseif !ismarked(s1) then mark(s1); stack.push(s1)
    end
  end
```

ここでスタックをキューに取り替えると幅優先探索となる。

2.4 例題: 箱入り娘

「箱入り娘」というパズルをご存じだろうか。これは図 8 のような枠と駒から成るパズルで、初期状態から駒をスライドさせていって、最終的には「箱入り娘」が出口から出られる状態にする、というものである。どんなものか分からない人はこれをやってみてください: http://hp.vector.co.jp/authors/VA006860/hako/sample5.html

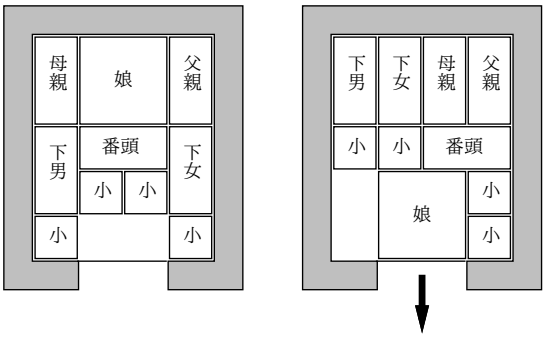

図 8: 箱入り娘の初期状態と最終状態 (の 1 つ)

では、これを解くプログラムを説明しよう。まず、縦横のマス目の数、「目標」となる娘の位置、駒 (0〜9 で表す) の縦と横のサイズ、および駒の名前を表す 1 文字 (表示文字列生成用) を広域変数に入れておく。

```
$hmax = 3; $vmax = 4; $goalx = 1; $goaly = 3
$hsize = [2, 1, 1, 1, 1, 2, 1, 1, 1, 1]
$vsize = [2, 2, 2, 2, 2, 1, 1, 1, 1, 1]
$names = 'Mvvvvwssss.'
```

次に、9 個の駒の位置 (X 座標、Y 座標をそれぞれ要素数 9 の配列に入れてあるものとする) を受け取り、盤面を表現した 2 次元配列を作るメソッド makeboard。まず最初に全要素が-1(何も置いてない) の 2 次元配列を作る (2 次元配列の作り方を忘れた場合は画像の回の資料参照)。そして各駒について、その駒の置かれている全範囲に駒の番号を入れる。

```
def makeboard(x, y)
  b = Array.new($vmax+1) do Array.new($hmax+1, -1) end
  $hsize.length.times do |i|
    $hsize[i].times do |j|
      $vsize[i].times do |k| b[y[i]+k][x[i]+j] = i end
    end
  end
  return b
end
```

さて次に、盤面 b を受け取り、駒 i が現座 x,y にあるものとして、これを dx,dy だけ動かすことが可能かどうか調べるメソッド movable を見てみよう。まず、動かした後が盤面からはみ出るなら NO を返す。そうでないなら、動かした後の自分の置かれる範囲のマスすべてについて、そこが-1(何も置いてない) でもこれまで自分が置いてあったマスでもないなら直ちに NO を返す。最後まで NO でなければ OK を返す。

```
def movable(b, i, x, y, dx, dy)
  if x+dx < 0 || x+$hsize[i]-1+dx > $hmax ||
     y+dy < 0 || y+$vsize[i]-1+dy > $vmax then return false end
  $hsize[i].times do |j|
    $vsize[i].times do |k|
      p = b[y+dy+k][x+dx+j]; if p != -1 && p != i then return false end
    end
```

```
    end
    return true
  end
```

盤面の 1 つの状態を表すのにはオブジェクトを使うことにしたので、State というクラスを作った。インスタンス変数としては、9 個の駒の XY 座標の配列、および「1 つ前の状態」を保持する。なお、配列は initailze で渡されて来たものをそのまま持っていると書き換えられてしまうので、.dup でコピーを作ってそれを保持する。

getx、gety は各駒の座標値の配列をそのまま返す。isgoal は駒 0 が目標位置にあるか否かを返す。move は自分の状態から駒 i を dx,dy だけ動かした新しい状態を作って返す。この時一時的に配列中の駒の位置を動かして新しい状態を作り (self というのは「このオブジェクト」を渡すという意味)、終わったら戻すので、上記.dup が必要なのだった。

```
  class State
    def initialize(x, y, p) @x = x.dup; @y = y.dup; @prev = p end
    def getx() return @x end
    def gety() return @y end
    def isgoal() return @x[0] == $goalx && @y[0] == $goaly end
    def move(i, dx, dy)
      @x[i] = @x[i] + dx; @y[i] = @y[i] + dy; s = State.new(@x, @y, self)
      @x[i] = @x[i] - dx; @y[i] = @y[i] - dy; return s
    end
    def to_s()
      s = ''
      makeboard(@x, @y).each do |a|
        a.each do |i| s = s + $names[i..i] end; s = s + "\n"
      end
      return s
    end
    def output(f)
      if @prev != nil then @prev.output(f) end
      f.puts(to_s); f.puts('------')
    end
  end
```

to_s は makeboard で盤面の配列を作ったあと、それを 1 つの文字列にしているが、そのとき駒の番号ではなく各駒を表す文字 1 文字にしているのに注意。これは、縦長の駒や小僧の駒はどれでも同じことなので、これらを同じ名前にすることで状態数を少なくできるため。output は最後に目的状態に到達できた時使うメソッドで、ファイル f を渡されて最初に近い状態から順にファイルに状態を書き出す (@prev は逆向きのリストなので、先頭から表示するため再帰を使っている)。

では最後に main。今回は速度が問題なので自前のスタックではなく Ruby の配列をスタックに使うことにした。x と y は各駒の初期配置で、これらを指定して最初の状態を作り、スタックに積む。また、ハッシュvisited は状態の文字列表現を入れることでその状態を既に処理ずみであることを表すのに使うので、まずは最初の状態を「訪問ずみ」とした。その後が処理本体で、まずスタックが空ならもう調べる状態がないので失敗。そうでなければスタックから 1 つ取り出し (進捗を見るためここで画面に表示してもいい)、取り出したのが目的状態ならループを抜け出す。そうでない場合は、処理する状態の各駒の座標を取り出し、全ての駒について上下左右に移動できるか調べ、できるのならその新しい状態を生成し、それが既に処理ずみ (な状態と同等) でないなら、スタックに積んで処理ずみとして登録する。

```
  def main
    x = [1, 0, 3, 0, 3, 1, 0, 1, 2, 3]; y = [0, 0, 0, 2, 2, 2, 4, 3, 3, 4]
    stack = [State.new(x,y)]; visited = {stack[0].to_s => true}
    while true do
      if stack.length == 0 then puts("impossible."); return false end
```

```
    s = stack.pop # or s = stack.shift
    # puts(s.to_s); puts('----')
    if s.isgoal then break end
    x = s.getx; y = s.gety; b = makeboard(x, y)
    $hsize.length.times do |i|
      [[1,0],[-1,0],[0,1],[0,-1]].each do |a|
        if movable(b, i, x[i], y[i], a[0], a[1])
          s1 = s.move(i, a[0], a[1]); k1 = s1.to_s
          if visited[k1] == nil then stack.push(s1); visited[k1] = true end
        end
      end
    end
  end
  File.open('out.data', 'w') do |f| s.output(f) end; return true
end
```

成功してループを抜けた時にはファイル out.data に初期状態から目的状態までの各状態を出力しておしまい。ふー長かったですね、おつかれ様でした。では動かしてみよう。

```
irb(main):002:0> main
vMMv
vMMv
vwwv
vssv
s..s
----
vMMv
vMMv
vwwv
vssv
s.s.
----
vMMv
vMMv
vwwv
vssv
ss..
----
(途中略)
vvvv
vvvv
ssww
MM.s
MM.s
----
vvvv
vvvv
ssww
.MMs
.MMs
```

```
----
=> true
```

解けたようだ。で、実行時の表示には「行き詰まった」枝も全部含まれるが、ファイルには見た目はこれと同様でも、正しく解ける経路だけが記録されている。

ちなみに、スタックを使うと探索が深さ優先なので解けるまでの時間は短いが得られた解が最短とは限らない。幅優先探索にすれば得られた解は最短手数のものとなる。それには `stack.pop` の代わりに配列の先頭から取り除くメソッド `stack.shift` を使えばよい (もはやスタックでないのに `stack` という名前なのはまずいですけど)。

演習 7 上のプログラムを打ち込み、動作を確認せよ。成功したら、駒の大きさや配置を変更して正しく動作するか確認せよ。できれば、幅優先と深さ優先の比較もやってみてほしい。

演習 8 自分の好きなゲームないしパズルを扱う状態空間探索プログラムを作れ。

演習 9 その他、ここまでにこの授業で学んだ内容を活かした面白いプログラムを作れ。何が面白いかの定義は各自に任されます。

A　本日の課題 10A

「演習 1」「演習 2」「演習 5」「演習 6」で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 10A」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 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。

   Q1. スタック、キューについて理解しましたか。

   Q2. 状態、状態遷移、オートマトンとか納得しましたか。

   Q3. 本日の全体的な感想と今後の要望をお書きください。

B　1/18 までの課題 10B

次回までの課題は「演習 1」〜「演習 9」の (小) 課題から 2 つ以上選んで報告することです (10A で提出したものは除く)。

各課題のために作成したプログラムは複数ある場合もすべてレポートに掲載してください。レポートは 1/18 の授業開始時刻の **10 分前** までに久野までメールで送付してください。

1. Subject: は「Report 10B」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 1 つ目のプログラムのソース。
4. その説明と分析/考察。
5. 2 つ目のプログラムのソース。
6. その説明と分析/考察。
7. 下記のアンケートの回答。

   Q1. 状態空間探索の例題プログラムは読めましたか。

   Q2. 例題プログラムで各種のデータ構造を駆使して状態を表現していることが読み取れましたか。

   Q3. 課題に対する感想と今後の要望をお書きください。