■6 群（基礎理論とハードウェア）- 3 編（アルゴリズムとデータ構造）

3 章 主要なアルゴリズムとデータ構造

（執筆者：滝本宗宏・久野　靖）[2012 年 7 月 受領]

■概要■

1 章でアルゴリズム一般について，また 2 章でデータ構造について説明してきたが，本章ではいよいよ具体的な個別のアルゴリズムについて説明する．

アルゴリズムはそれが扱うデータ構造と密接に結び付いているので，その提示に当たっては対象とするデータ構造に着目することが一つ考えられる．また，別の分類方法としては，アルゴリズムの主要な用途に基づいて分類することも考えられる．本章ではこの両方の視点による分類を併用する．

ただし，各種のアルゴリズムについて網羅するとなると，それだけで「アルゴリズム事典」のような大著になってしまうので，ここでは代表的なデータ構造及び代表的なアルゴリズムに絞って述べることにする．

また，実際のアプリケーションの構築に当たっては，そのアプリケーションの目的に応じてアルゴリズムやデータ構造を設計していくことが中心となっていく．その一般的な方法については 7 群「ソフトウェア」で扱われるが，本章では具体的な実例として「コンパイラ内部のコード最適化」を取り上げ，これを扱うためのデータ構造とアルゴリズムについて 1 節を設けて解説している．

【本章の構成】

本章では，データ構造やデータがもつ固有の性質に着目しながら，それを対象としたアルゴリズムを紹介する．最も古くからある基本的なデータ構造は配列であることから，3-1 節では配列をデータ構造として用いるアルゴリズムを取り上げる．続いて 3-2 節では動的なデータ構造の一般的な形であるグラフをデータ構造として用いるアルゴリズム，3-3 節ではグラフの制約された形でありコンピュータサイエンスで広く使われるデータ構造である木を対象としたアルゴリズムについて説明する．

3-4 節以降ではアルゴリズムの用途を中心とした分類に進み，3-4 節では文字列に対する探索とパターンマッチングのアルゴリズム，3-5 節では状態空間及びゲーム局面の探索に関するアルゴリズムを解説する．最後に 3-6 節では，プログラムコードを扱うという問題を取り上げ，そのためにデータ構造，その性質と解析法，及び解析結果の応用としてのコード最適化アルゴリズムの例を示す．

■6 群 - 3 編 - 3 章

3-1 配列アルゴリズム

（執筆者：山口文彦 · 久野　靖）[2012 年 7 月 受領]

3-1-1 配列とそれに対するアルゴリズム

配列は同一型の要素が連続して並んでいるデータ構造であり，添字（何番目かを示す番号）を与えることでその要素にアクセスできる．このとき，添字の値にかかわらず一定時間で（$O(1)$ の時間計算量で）要素にアクセスできることが配列の特徴であり，アルゴリズムもこれを前提に組み立てられる．

配列を使って実現されるデータ構造の代表的なものに，**表（table）** がある．表とは，**鍵（key）**と**値（value）**の対を格納したものであり，鍵を指定して対応する値を効率よく取り出すことが目標となる．これを**探索（searching）**と呼ぶ．鍵の種類が大きくない非負整数であれば，配列自体がそのまま（鍵を添字として指定することで）表として機能する．しかしこのような都合のよい場面ばかりではないので，配列の上に効率のよい表の格納と探索を行うアルゴリズムを構築することが目標となる．

配列に対する別の種類のアルゴリズムとして，**整列（sorting）**がある．整列とは，配列に連続して格納されている値（ないし鍵と値の組）を値や鍵の昇順（または降順）に並べることをいう，整列に対しては様々なアルゴリズムが考案されている．

このほか，配列の配列などにより，2 次元行列，3 次元行列などを扱うことも広く行われる．この場合，線形代数の問題（連立方程式の解法，連立不等式の解法）を扱うことになる．これらについては本章では紙面の都合上扱わない，

3-1-2 線形探索と 2 分探索

探索の問題を最も単純化して扱うため，配列に一群の要素が並んで格納されていて，そのなかから特定の値 x を探し出す（x が格納されている配列の添字を知る）ことを考えることにする．（すなわち，ここでは鍵と値が同一のものである場合を考えている．一般に鍵と値が別の場合は，鍵と値（複数あってよい）それぞれに同一サイズの配列を用意し，鍵と同じ添字位置に対応する値を格納しておけばよい．または，レコード型の配列などを用い，各レコードに鍵と値を組にして格納してもよい．）

単純に，配列の先頭から順に要素と x を比較していく方法が考えられる．これを**線形探索（linear search）**と呼ぶ（図 3･1）．この方法は配列の要素がどのような順番に並んでいても使用可能であるが，要素数を n とすると，配列に含まれている x を見つけるまでに平均して $\frac{n}{2}$

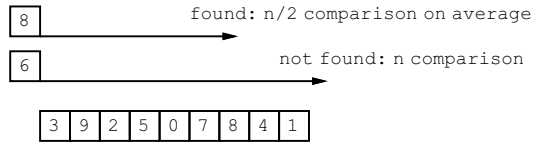


図 3･1　線形探索

回の比較を必要とする．また，x が配列に含まれていない場合には，含まれていないと分かるまでに n 回の比較を必要とする．すなわち，線形探索は探索に $O(n)$ の時間計算量を要する．値を格納するときも，まず x が含まれているかどうか調べる必要があるので，同じである．

鍵が全順序をもつデータであり（整数，実数，文字列など），かつ配列の内容を鍵の昇順ないし降順に並べておける場合は，**二分探索（binary search）**を用いることができる．これは，図 3･2 に示すように，x と探索範囲の中央にある値を比較し，等しくなければ探索範囲を約半分にすることを繰り返すアルゴリズムである（値が全順序をもつことから，比較した値が x より大きいか小さいかによって，どちらかの半分は調べなくてよいと分かる）．したがって，二分探索は x を見つけるか，または含まれていないと分かるまでに $O(\log n)$ 回の比較で済む．一方，新しい鍵を挿入したり，既存の鍵を削除する場合には，その鍵の位置以降の要素（平均して $\frac{n}{2}$ 要素）をずらす必要があるため，$O(n)$ の時間を要する．

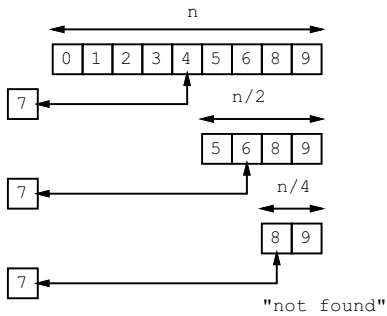


図 3･2　2 分探索

図 3･3 に，配列 a から要素 x を探すプログラム binarySearch を示す．ここで a の要素はあらかじめ昇順に並べられているものとする．x が a に含まれていれば，x を値とする a の添字の値を返し，もし含まれていなければ –1 を返す．図 3･3 で，変数 min と max は探索する範囲を表す変数で，配列の添字が min 以上 max 未満の範囲から x を探そうとする．

3-1-3 ハッシュ表

前述のように，配列は鍵が大きくない値の非負整数であるときに $O(1)$ でアクセスできる表であると考えることができる．そこで，鍵が上記の条件を満たさないときでも，ある関数 $H : Dom(key) \rightarrow 0..N$ を定義し，これを用いて要素の格納位置を決めることが考えられる．この関数 H を**ハッシュ関数（hash function）**と呼ぶ．ハッシュ関数は，等価なオブジェクトに対して異なるハッシュ値を返してはならない．例えば，文字列に対するハッシュ値は，その文字列を構成する文字から計算されるべきで，文字列を格納しているメモリのアドレスから計算されるべきではない．そしてハッシュ関数を用いて格納位置を決定することに基づく表のことを**ハッシュ表（hash table）**と呼ぶ．

いくつかのプログラミング言語では，整数以外の値を添字として用いることができる配列が組込みで用意されており，**連想配列（assosiative array）**などと呼ばれる．この機能はハッシュ表で実現されていることが多い．

```
def binarySearch(x,a)
  min=0; max=a.size
  while(min < max)
    i=(min+max)/2;
    if (x < a[i]) then
      max=i
    elsif (x > a[i]) then
      min=i+1
    else
      return i
    end
  end
  return -1
end
```

図 3·3　二分探索のコード

もし「$key_1 \neq key_2$ であるなら常に $H(key_1) \neq H(key_2)$」なら，H を**完全ハッシュ関数（perfect hash function）**と呼び，これを使って一意的に格納位置を計算でき，アクセス時間 $O(1)$ の表が構成できる．例えばプログラミング言語の予約語のように，あらかじめ鍵の集合が与えられている場合には，完全ハッシュ関数を構成することも実用的に行われる．

実際にはハッシュ関数において $H(key_1) = H(key_2)$ が起きることは避けられない．これを**衝突（collision）**と呼ぶ．衝突は，ハッシュ関数で計算した位置に格納されているキーが検索に用いたキーと等しくないことで検出できる（図 3·4 中）．

衝突に対処する方法としては，各配列位置を連結リストとして構成し，衝突したものをリストに連結して格納する方法（連鎖法）と，衝突したときにはもう一つ別のハッシュ関数 h により $h(key_1)$，$h(key_2)$ を計算し，最初の位置からその値だけずれた位置に値を格納する方法（開アドレス法）がある（図 3·4 右，そこがふさがっている場合は更に $h(key_1)$，$h(key_2)$ ずつずれた位置を探して行き，表の末尾に来たら先頭に戻る）．

ハッシュ表では，ハッシュ関数をできるだけ 0..N の間にランダムに値が分布するように構成することで，表の大きさに対する格納されている値の数（充填率）が高くないときには平

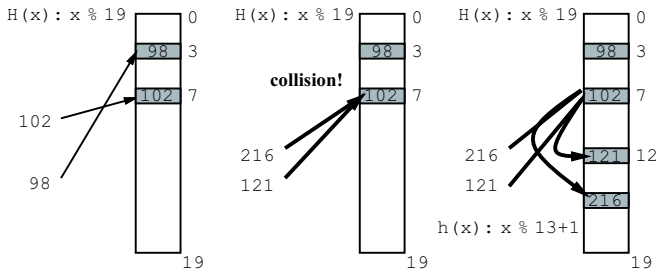


図 3·4　ハッシュ法 (開アドレス法)

均して $O(1)$ の時間でアクセスを行うことができる．充填率が高くなってきた場合には，サイズの大きな配列を新たに割り当て，格納されている値をすべて新たな表に移し換えることで充填率を低くできる．これを**再ハッシュ（rehashing）**と呼ぶ．

3-1-4 バブルソート・選択ソート，挿入ソート

与えられた配列の要素を，ある全順序にしたがって昇順に並べかえることを，**整列**ないし**ソート（sorting）**と呼ぶ．**バブルソート（bubble sort）**はソートアルゴリズムの一つであり，配列の位置を順にスキャンしながら各位置についてそれと隣接する要素と比較し，もし順序が逆であれば交換する操作を繰り返す方法である．例えば降順に並べるとして，1 回目に先頭から末尾までスキャンすると（$n-1$ 回の比較・交換）最大要素が最後の位置にくる（図 3･5），このように，比較交換によって大きい要素が水の泡のように立ち昇ってくるところからこの名前がある．

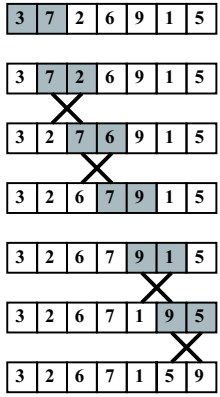


図 3･5　バブルソート

2 回目のスキャンでは，2 番目に大きい要素が最後から 2 番目の位置にくるが，このときは最後の要素との比較は不要である．このように，スキャンするごとにその範囲は一つずつ小さくしていける．このコードを図 3･6 に示す．比較・交換の回数は $(n-1)+(n-2)+\cdots+2+1=\frac{n(n-1)}{2}$ 回となるので，バブルソートの時間計算量は $O(n^2)$ に属する．また，バブルソートには「比較交換が 1 回も起きなくなるまで繰り返しスキャンを繰り返す」というバリエーションも可能であるが，計算量は同じである．バブルソートは，元の配列にデータが入った状態で（余分の領域を使用せずに）整列が行える．

選択ソート（selection sort）は，（昇順に並べる場合）まず全範囲から最小の要素を選んでそれを先頭に置き，次に残った要素から最小のものを選んで 2 番目に置き，というふうに選択を繰り返す方法である．選んだ要素を目指す位置に置くとき，その目指す位置にある要素と交換することで，もとの配列以外の領域を使用せずに整列を行える（図 3･7）．選択ソートは，最初に n 要素の最小，次に $n-1$ 要素の最小，…を求める繰り返しとなるため，比較回数は $(n-1)+(n-2)+\cdots+2+1$ となり，時間計算量はバブルソートと同じく $O(n^2)$ に属する．

```
def bubbleSort(a)
  for i in 0..(a.length-2)
    for j in 1..(a.length-i-1)
      if a[j-1] > a[j] then
        t=a[j]; a[j]=a[j-1]; a[j-1]=t;
      end
    end
  end
end
```

図 3･6　バブルソートのコード

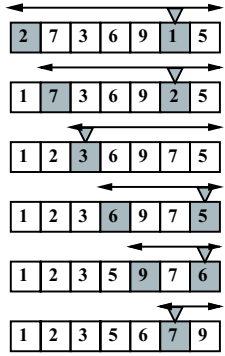


図 3･7　選択ソート

挿入ソート（insertion sort）は，まず 0 要素の列に最初の要素を追加し，次に 1 要素の列に 2 番目の要素を追加し，次に 2 要素の列に 3 番目の列を追加し，というふうに要素を追加していくが，それぞれの追加に際して適切な位置に追加要素を挿入することで最終的に整列された列を構成する（図 3･8）．この方法ももとの配列以外の領域を必要としない．挿入ソートの時間計算量は，各回の挿入に際して平均して現在整列ずみの列のおよそ半分の要素をずらすことになるので，ずらす回数が $\frac{1}{2}(1+2+\cdots+n-1)$ となり，やはり $O(n^2)$ に属する．

バブルソート，選択ソート，挿入ソートはいずれも分かりやすいがナイーブな方法であり，時間計算量が $O(n^2)$ に属するため，ごく少量のデータに適用する場合を除いては実用的ではない．

3-1-5 マージソート・クイックソート

マージソート（mergesort）は「配列を二つのほぼ同じ大きさの配列に分割し，それぞれを整列したのち，整列済みの二つの配列を一つに併合（マージ）する」という再帰的な記述で説明できる（図 3･10）．その具体的な進行は，まず配列を長さ 1 以下になるまで繰り返し分割し，そこから二つずつ併合していく，というかたちになる（図 3･9）．

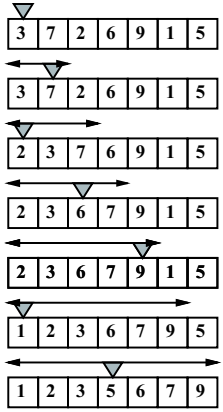


図 3･8　挿入ソート

整列済みの配列を併合するには，両配列の先頭要素を比べて小さい方を取り除き結果の配列に追加することを繰り返せばよい．分割の各段ごとに，分割の操作も併合の操作も n 個の要素を順次操作するので $O(n)$ の時間がかかり，段数は $\log_2 n$ であるので，全体としてマージソートの計算量は $O(n \log n)$ となる．マージソートはマージ操作の際にもとの（マージされる）列とマージ結果の列を両方扱う必要があるため，一つの配列上ではうまく実現できない．

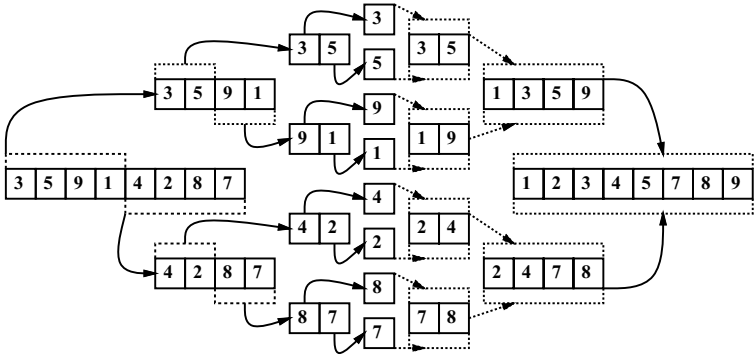


図 3･9　マージソート

クイックソート（quicksort）は，「整列対象のなかから要素を一つ選び（これをピボットと呼ぶ）．残りの配列をピボットよりも大きなものと小さなものに分割してからそれぞれを整列し，それらを連結する」という再帰的な記述で説明できる．ただし図 3･11 にあるように，もとの配列をピボットより小さい部分配列と大きい部分配列に分けるように工夫することで，一つの配列だけでクイックソートを行うことは可能である．

```
def mergeSort(a)
  if a.length < 2 then return; end
  l=[]; r=[]
  until a.empty?
    l << a.shift
    if a.empty? then break; end
    r << a.shift
  end
  mergeSort(l); mergeSort(r)
  until (l.empty? and r.empty?)
    if (! l.empty?) and (r.empty? or l[0]<r[0]) then
      a << l.shift
    else
      a << r.shift
    end
  end
end
```

図 3･10　マージソートのコード

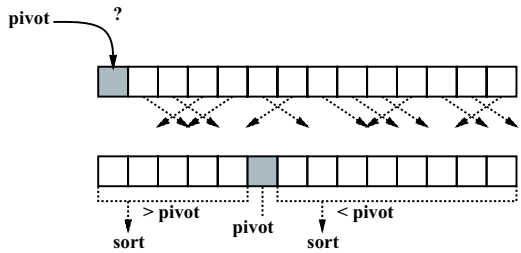


図 3･11　クイックソート

図 3･12 の上側に示したクイックソートのコードでは分かりやすさのため，先頭要素をピボットに選んだ後，二つの列に振り分けてそれぞれ自分自身を再帰的に呼ぶことで整列し，連結している．一方，図 3･12 の下側では，qsort を配列 a の第 l 要素から第 r 要素を整列する手続きとして定義することで，新たな配列を割り当てることなしに整列を行っている．

クイックソートでは，ピボットによる分割ごとに各要素を振り分けるので，分割 1 段について合計で $O(n)$ の計算量となる．そして分割の段数は，毎回整列対象の配列をおおむね半分にできるなら，$\log_2 n$ となるので，全体としての時間計算量は $O(n \log n)$ となる．

しかし，ピボットとして選んだ要素が配列中の最小値または最大値であった場合には，配列が事実上分割されない．このように配列が偏って分割される場合には，計算コストがかかる．例えば，必ず配列の先頭の要素をピボットとして選ぶ場合の最悪ケースの一つは最初からソートされている場合であり，この場合には $O(n^2)$ の計算量となる．

クイックソートでは，このようにピボットの選び方によって計算コストが変わる．配列を

できるだけ等分に分割したいので，ピボットよりも大きな要素の個数と小さな要素の個数がほぼ同じであることが望ましい．平均的にそのような性質を満たす方法として，配列の要素中からランダムに一つ選んでピボットとする方法が挙げられる．すなわち，乱択アルゴリズムを用いることで，計算量が最悪ケースとなる可能性を非常に小さくすることができる．このように，最悪ケースの計算コストではマージソートに劣るが，平均的にはクイックソートの方がデータの操作量が少ないため，多くの場合マージソートよりもクイックソートの方が高速だとされる．

```
def quickSort(a)
  if a.length < 2 then
    return a
  end
  l=[]; r=[]; x = a.shift
  until a.empty?
    if a[0] < x then
      l << a.shift
    else
      r << a.shift
    end
  end
  a.concat(quickSort(l))
  a << x
  a.concat(quickSort(r))
end

def qsort(a,l,r)
  i=l; j=r; x=a[(l+r)/2]
  x=a[l]
  until (i >= j)
    until ( a[i] >= x ); i=i+1; end
    until ( x >= a[j] ); j=j-1; end
    if (i <= j) then
      tmp=a[i]; a[i]=a[j]; a[j]=tmp
      i=i+1; j=j-1
    end
  end
  if (l < j) then qsort(a,l,j); end
  if (i < r) then qsort(a,i,r); end
end
```

図 3·12　クイックソートのコード

3-1-6 2 分木の配列表現とヒープソート

各ノードがデータとたかだか二つの子をもつ木構造を 2 分木と呼ぶ．バランスしている（どのノードも両方の子の高さがたかだか 1 しか違わない）木のうちで，葉ができるだけ左に寄っているものを考えると，このような木を配列上に表現することができる．具体的には，根の添字を 1 として，添字が i の位置に格納されているノードの子を $2i+1$ と $2i+2$ の位置に格納すればよい．例えば，図 3･13 で左の配列は論理的には右の木構造に対応している．これを **2 分木の配列表現（array representation of binary tree）**と呼ぶ．

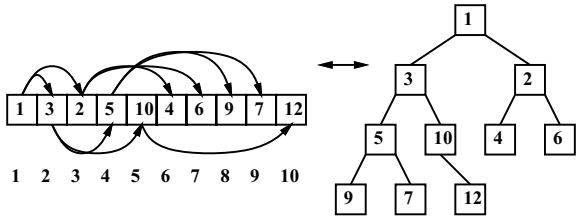


図 3･13　2 分木の配列表現とヒープ

一般に，木構造のうちで「どのノードについても，子が持つ値が親よりも大きいか等しい」という条件を満たすものをヒープと呼ぶ．2 分木の配列表現の上にもヒープを実現できる．図 3･13 に格納されているデータはこの条件を満たしている．ヒープの先頭には最小値が格納されているので，順序付きキューなどを実現するのに都合がよい．

ヒープの先頭要素を取り出して削除するときは，ヒープの終端のデータを暫定的に先頭に置き，この値を左右の子のうちで小さい方と交換する操作を，ヒープの条件（子があるなら自分より大きい値）が満たされる状態になるまで繰り返す（図 3･14 上）．これをシフトダウン操作と呼ぶ．逆にヒープに新たな値を追加するときには，その値を暫定的にヒープの終端に置き，ヒープの条件が満たされるまで親との間で交換することを繰り返す（図 3･14 下）．これをシフトアップ操作と呼ぶ，いずれの操作も，木の高さの回数までの比較交換で終了するため，$O(\log n)$ で終了する．図 3･16 にヒープのシフト操作のコードを示す．

まず 1 要素のヒープを作成し，そこに配列の各要素をすべて挿入し，続いて最小値（ないし最大値）を次々に取り出し並べることで整列を実現できる．これを**ヒープソート（heap sort）**と呼ぶ．図 3･15 のように，一つの配列の先頭部分にヒープを実現し，残りの部分に配列のデータを保持することで，一つの配列だけでヒープソートを実現することができる（この場合は昇順に並べたければ根が最大になるようなヒープを使うとよい）．ヒープソートは，n 個の値を挿入し取り出すので，計算量は $O(n \log n)$ に属する．

ヒープソートはクイックソートと異なり最悪の場合でも $O(n \log n)$ であるため，時間の上限保証が必要であり，かつ記憶領域に余裕がない場合に使われることがある．

3-1-7 2 次記憶ソート

ここまでは配列アルゴリズムを扱ってきたが，整列について扱った関係上，ここで 2 次記憶ソートについても説明しておく．

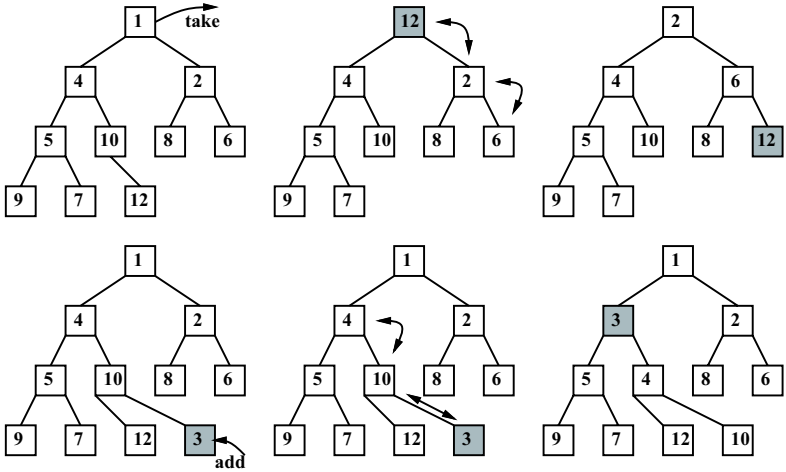


図 3·14　ヒープのシフト操作

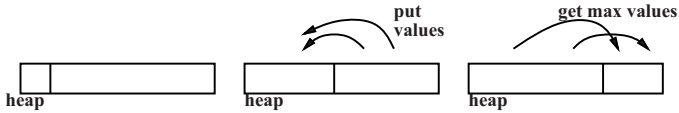


図 3·15　ヒープソート

メモリに収まらないような巨大なデータの整列を行う場合には，ファイルシステムなどの 2 次記憶装置を用いることになる．整列されるデータや整列結果だけでなく，計算の途中で利用する記憶装置にも 2 次記憶装置を用いる場合を考える．このようなソートを **2 次記憶ソート**または**外部ソート（external sort）**と呼ぶ．これと対比して，主記憶上でのソートを**内部ソート（internal sort）**と呼ぶこともある．

ファイルなどの 2 次記憶装置は順次の読み書きに向いているため，マージソートで用いた分割と併合の操作が扱いやすい．しかし，マージソートにおける分割された列のそれぞれをファイルに記憶しようとすると，計算の途中に用いるファイルが膨大な個数になってしまい，現実的ではない．このような条件の下でソートを行う方法として，作業用のファイルを二つ用いる次のようなアルゴリズムが挙げられる．

ソートしたいデータの列が格納されているファイルを F とし，作業用に A，B の二つのファイルを用意する．まず，F 中のデータが部分的に昇順に並んでいると考える．昇順に並んでいる部分をひとまとめにし，そのまとまりを A と B に交互に書き込んでいくことで，F を分割する．分割したら，新たに F として A と B を併合したファイルをつくる．併合の際

```
class Heap
  def initialize()
    @tail=-1; @a=[]
  end
  def put(n)
    @tail=@tail+1; @a[@tail]=n
    shiftUp(@tail)
  end
  def shiftUp(i)
    until i <= 0
      j = (i-1)/2
      if @a[j] < @a[i] then return; end
      n=@a[i]; @a[i]=@a[j]; @a[j]=n
      i=j
    end
  end
  def get()
    if @tail < 0 then return nil; end
    n = @a[0]; @a[0]=@a[@tail]; @a[@tail]=nil; @tail=@tail-1
    shiftDown(0)
    return n
  end
  def shiftDown(i)
    until false
      j=2*i+1; k=2*i+2
      if @a[j].nil? then return; end
      if @a[k].nil? or @a[j] < @a[k] then
        if @a[i] <= @a[j] then
          return
        end
        n=@a[i]; @a[i]=@a[j]; @a[j]=n
        i=j
      else
        if @a[i] <= @a[k] then
          return
        end
        n=@a[k]; @a[k]=@a[i]; @a[i]=n
        i=k
      end
    end
  end
end
```

図 3·16　ヒープの挿入・削除のコード

は，A，B を先頭から見ていき，小さい方を読み進めて F に追加する．マージソートと違い，A，B が整列されているわけではないので，新たに得られる F も整列済みとは限らないが，F 中で部分的に昇順に並んでいるまとまりの一つひとつは，分割する前よりも大きくなっている．そこで，この分割と併合の操作を F がソート済みになるまで繰り返す．図 3･17 は，このようなソートの様子を表す，F の列の下に付けられた弧は，部分的に昇順に並んでいる部分を示している．

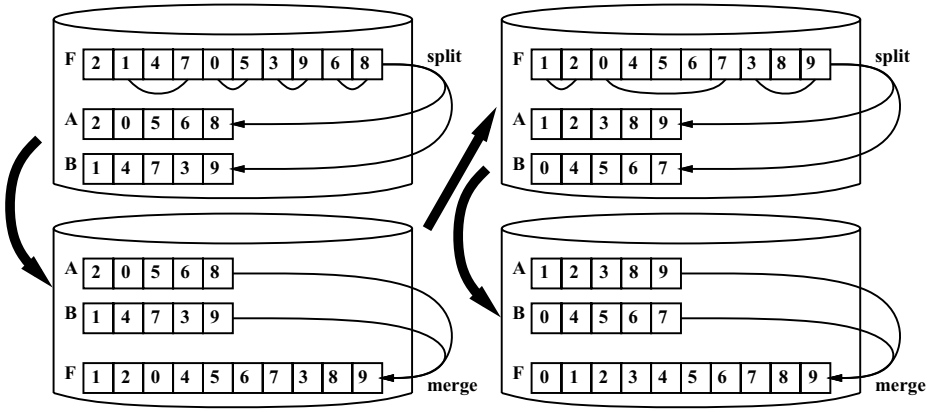


図 3･17　2 次記憶ソートの様子

ここでは，2 次記憶のみを用いた整列の様子を示したが，実際には，ある程度の大きさのソート済みの部分列を，より高速な主記憶を利用して作成し，その後でここで示したアルゴリズムを使用することが多い．

■参考文献

1) 五十嵐健夫, “データ構造とアルゴリズム,” 新・情報/通信システム工学=TKC-3, pp.45–59, 数理工学社発行, サイエンス社発売, 2007.
2) Jan van Leeuwen 編, 廣瀬　健, 野崎昭弘, 小林孝次郎 監訳, “アルゴリズムと複雑さ（原題: Algorithm and Complexity）,” コンピュータ基礎理論ハンドブック I, pp.460–472, 丸善, 1990.
3) K. メールホルン, P. サンダース 著, 浅野哲夫 訳, “アルゴリズムとデータ構造 基礎のツールボックス,” シュプリンガー・ジャパン, pp.119–198, 2009.
4) 紀平拓男, 春日伸弥, “プログラミングの宝箱 アルゴリズムとデータ構造 第 2 版,” pp.2–16, 36–45, ソフトバンククリエイティブ, 2003.
5) 杉原厚吉, “データ構造とアルゴリズム,” pp.29–34, 35–39, 101–106, 共立出版, 2001.
6) 玉木久夫, “乱択アルゴリズム,” アルゴリズム・サイエンス シリーズ 4, 数理技法編, pp.29–37, 共立出版, 2008.

■6 群 - 3 編 - 3 章

3-2 グラフアルゴリズム

（執筆者：山口文彦）［2012 年 7 月 受領］

3-2-1 グラフとその表現

コンピュータサイエンスにおける**グラフ（graph）**とは，**頂点（vertex）**ないし**ノード（node）**を**辺（edge）**で結んでつくられる構造であり，頂点で表されるものどうしの「つながり方」を表現するために用いられる．グラフは，頂点の集合 V と辺の集合 E を用いて (V, E) で表される．各辺は二つの頂点を結んでおり，辺ごとにその辺を通るときにかかる**コスト（cost）**ないし**重み（weight）**が付随している場合がある．一般に，辺の向きを考慮するグラフを**有向グラフ（directed graph）**，考慮しないグラフを**無向グラフ（undirected graph）**と呼ぶ．

グラフを表現するデータ構造として簡単なものの一つに，頂点 i と頂点 j を結ぶ辺の有無やコストなどの情報を 2 次元配列の (i, j) 要素に格納する方法が挙げられる（図 3･18）．この配列のことを，頂点の隣接関係を保持していることから，**隣接行列（adjacency matrix）**と呼ぶ．図 3･19 に隣接行列表現でグラフを表現するクラスのコードを示す．

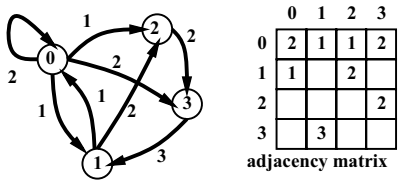


図 3･18　重み付き有向グラフとその隣接行列表現

```
class Graph
  def initialize(o)
    @order = o
    @edge = Array.new(o)
    @edge.each_index{|x|
      @edge[x] = Array.new(o,nil)
    }
  end
  def setEdge(x,y,w); @edge[x][y]=w; end
  def order; @order; end
  def edge(x,y); @edge[x][y]; end
end
```

図 3･19　隣接行列でグラフを表すクラス

この方法は簡便だが，無向グラフでは配列用に確保したメモリ領域の約半分以上が使われないことや，疎な（辺が少ない）グラフでは配列のほとんどの要素が辺がないことを表す値（例えば `nil`）になることから，記憶領域を無駄に使ってしまうことが多い．頂点数の 2 乗に

比べて辺の個数が少ない場合などにおいては，頂点ごとにその頂点から伸びる辺の情報をリスト構造などを用いて記憶しておく方法も考えられる．

一つの辺で直接結ばれた二つの頂点は隣接しているという．また，辺の両端が同じ頂点であるとき，この辺を自己辺といい，同じ二つの頂点を結ぶ辺が複数あるとき，これらの辺を多重辺という．多重辺がある場合には隣接行列表現は使用できない．

3-2-2 最短経路

グラフの各辺に重みが設定されているとき，2 頂点を結ぶ経路のうちで，辺の重みの和が最小となるものを見つける問題を**最短経路問題（shortest path problem）**という．

ダイクストラ法（Dijkstra Algoritm）は，各辺の重みが非負の場合に 2 頂点間の最短経路を求める代表的なアルゴリズムであり，始点からの最短距離が判明した頂点について，隣接する頂点の暫定的な最短距離を更新することを繰り返す方法である．初期状態では，始点だけが始点からの最短距離が 0 であると判明しており，始点以外の頂点までの暫定最短距離は $+\infty$ を表す値としておく．また，既に訪れた頂点の集合を用意し，初期状態では空集合としておく．

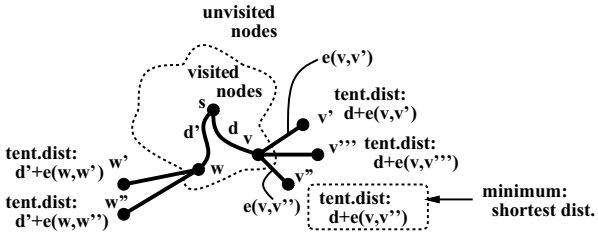


図 3･20　ダイクストラ法による最短路

図 3･21 に，ダイクストラ法のコードを示す，ここでは，順序付きソートを実現するために，図 3･16 に示したヒープを用いている，

頂点 v までの最短距離が d であると判明したとき，v に隣接するまだ訪れていない頂点 v' について，始点から v' に至る経路の長さはたかだか $d + e(v, v')$ となる．そこで，v' の暫定最短距離が，この値よりも大きければ更新する．v に隣接する頂点の暫定最短距離をすべて更新したら，v を既に訪れた頂点の集合に加える．各辺の重みが非負なので，v' よりも s に近い頂点 w を通って v' に至る経路がある場合にのみ，v' までの暫定最短距離が短くなる可能性があり，そのような頂点 w はまだ訪れていない頂点に含まれる．したがって，まだ訪れていない頂点のうちで暫定最短距離が最も短い頂点は，（暫定ではなく）最短距離が求められたことになるので，この頂点を新たに訪れたものとして上と同様の動作を繰り返す．

このアルゴリズムでは，暫定最短距離が最も短い頂点から順に訪れるために，頂点の集合を順序付きのキューに入れておく．順序付きのキューは，ヒープなどを用いて実装される．ここで，頂点 v の暫定最短距離がこれまで知られていた値よりも短くなった場合には，順序付きキューのなかでの v の位置を修正する必要がある．これを，終点が既訪頂点となるまで

```
require 'Heap.rb'

class Dijkstra
  class Node
    def initialize(d,ident)
      @ident = ident; @d = d
      @prev = nil
    end
    def ident; @ident; end
    def <(x);  @d.cost[@ident] <  @d.cost[x.ident]; end
    def <=(x); @d.cost[@ident] <= @d.cost[x.ident]; end
    def ==(x); @d.cost[@ident] == @d.cost[x.ident]; end
  end
  def initialize(graph)
    @graph = graph
    @visited = Array.new(graph.order,false)
    @cost = Array.new(graph.order,nil)
    @prev = Array.new(graph.order,nil)
  end
  def cost; @cost; end
```

図 3·21　ダイクストラ法（図 3·22 につづく）

繰り返す．

なお，頂点 v に至る暫定最短距離を更新する際に，そのような暫定最短距離を与える経路における v の一つ手前の頂点が分かっている．そこで，更新のたびに，各頂点の一つ手前がどの頂点だったかを記録しておけば，最終的に最短距離が求まったときに，終点から逆順に最短経路を取り出すことができる．

ベルマン・フォード法（Bellman-Ford algoritm）は，負の重みをもつ辺が含まれる場合に，与えられた二頂点間の最短経路を求めるアルゴリズムであり，すべての辺 (i, j) について，頂点 j の暫定最短距離を i の暫定最短距離と辺 (i, j) の重みの和と比較し，より小さい方に更新することを，$|V| - 1$ 回繰り返す方法である．暫定最短距離の初期値は，始点が 0，始点以外の頂点は $+\infty$ とする．重みが負になる閉路が存在しない限り，各頂点は最短経路中にたかだか 1 回しか含まれないので，頂点数（ただし始点は最短距離が 0 と分かっているので除く）だけ繰り返せば，真の値への更新がグラフ全体に伝播する．

重みが負になる閉路が存在する場合には最短距離を決めることができないが，すべての辺の終端頂点の暫定最短距離の更新を $|V| - 1$ 回行った後で，頂点 j の暫定最短距離を減らすような辺 (i, j) が存在すれば，重みが負になる閉路が存在することを検知できる．すべての辺についての比較・更新を $|V| - 1$ 回繰り返すので，ベルマン・フォード法の計算量は $O(|V| \cdot |E|)$ である．

ワーシャル・フロイド法（Warshall-Floyd algorithm）は，グラフ内の二つの頂点間の最短経路をすべて求めるアルゴリズムであり，二つの頂点 i, j の間の暫定最短経路を $p(i, j)$ とするとき，$p(i, j)$ を $p(i, k)$ と $p(k, j)$ を連結した経路と比較し，より短い方に更新すること

```
  def solve(s,g)
    queue = Heap.new()
    node = Array.new(@graph.order)
    (0..@graph.order-1).each{|i|
      node[i] = Node.new(self,i)
    }
    @cost[s] = 0;
    @visited[s] = true;
    queue.put(node[s])
    while(!@visited[g] && !(n=queue.get()).nil?)
      (0..@graph.order-1).each {|j|
        if !@visited[j] then
          if !(c=@graph.edge(n.ident,j)).nil? then
            tmpCost = @cost[n.ident] + c
            if @cost[j].nil? then
              @cost[j] = tmpCost
              @prev[j] = n.ident
              queue.put(Node.new(self,j));
            else
              if @cost[j] > tmpCost then
                tmp = []
                while(!(x=queue.get()).nil?)
                  tmp << x
                  if @cost[x.ident] > @cost[j] then break; end
                end
                @cost[j] = tmpCost
                @prev[j] = n.ident
                tmp.each{|x| queue.put(x) }
              end
            end
          end
        end
      }
      @visited[n.ident] = true
    end
    if @visited[g] then
      p=g
      while(p != s)
        print "#p "
        p = @prev[p]
      end
      print "#s\n"
    end
    return @cost[node[g].ident]
  end
end
```

図 3·22　ダイクストラ法（つづき）

を，すべての頂点 k について繰り返す方法である．$p(i, j)$ の初期値は，辺 (i, j) が存在するときに，経路 $[i, j]$，辺が存在しないときに「なし」とする．暫定最短経路が「なし」であるときの経路長（暫定最短距離）を $+\infty$ と定めれば，経路が見つかった際に更新される．すべての頂点対についての更新を，頂点数だけ繰り返すので，ワーシャル・フロイド法の計算量は $O(|V|^3)$ である，これはダイクストラ法を繰り返し用いるよりも高速である．

3-2-3 最大流問題

前節では辺の重みを 2 頂点間の道のりないし距離としてとらえていたが，別の問題として，辺の重みを 2 頂点間の「パイプの太さ」ないし流れの容量（流量）としてとらえるものがある．

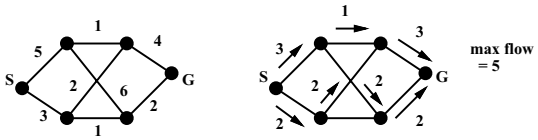


図 3･23　最大流問題

そこで最短経路問題の流れ版として，有向グラフ上の 2 頂点間の流れについて，流量の最大値を求める問題を考える（図 3･23）．辺 (i, j) の流量を $f(i, j)$ と書くとき，これは i から j に向けた流量を示しており，逆向きの流れは正負が逆になると考える．すなわち，すべての i, j について，$f(i, j) = -f(j, i)$ であるとする．また流量には，キルヒホッフの法則が成り立つとする．すなわち，始点と終点以外のすべてのノード i について，$\sum_{k \in V} f(i, k) = \sum_{k \in V} f(k, i) = 0$ である．

グラフの各辺に容量（非負の値とする）が設定されており，各辺の流量が容量を越えてはならないとする．このとき，始点からに流すことのできる流れの最大量を求める問題を最大流問題と呼ぶ．簡単のため，自己辺や多重辺は考えないものとする．

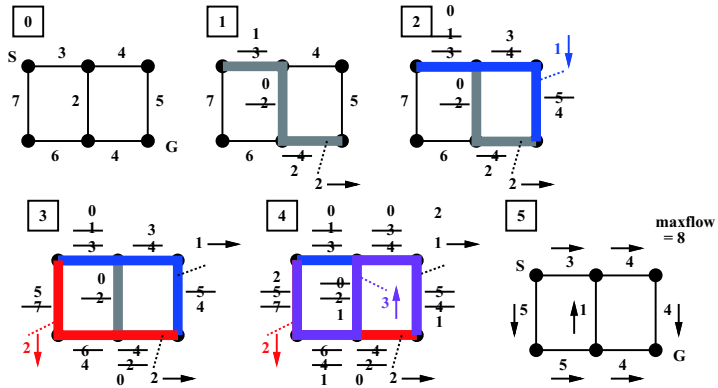


図 3･24　フォード・ファルカーソン法による最大流の計算例

フォード・ファルカーソン法（Ford-Fulkerson algoritm）は，最大流問題を解く代表的なアルゴリズムであり，容量に余裕があるか，または逆向きの流量のある辺だけをたどって，始点から終点まで至る経路（増大道（augumenting path）と呼ぶ）を探し，増大道に沿って流量を（始点から終点に向かう向きに）加算することを，増大道が見つからなくなるまで繰り返す方法である．

辺 (i, j) について i から j に向けた流量を a 増やせるのは，i から j に向けた容量 $c(i, j)$ が流量 $f(i, j)$ よりも a 以上大きい場合と，j から i に a 以上の流量 $f(j, i)$ がある場合である．そこで，このような辺だけをたどる経路（増大道）を探し，増大道で増やせる流量の最小値 a を求め，増大道上の各辺の流量に（始点から終点に向かう向きに）加算する．例えば図 3･24 では，増大道 (1)，(2)，(3) までを設定した時点で中央の辺は残容量が 0 であるが，ここに逆向きの流れを考えることで増大道 (4) を設定できている．

フォード・ファルカーソン法では，増大道を深さ優先探索などによって求める．なお，辺 (i, j) が存在しない場合の容量 $c(i, j)$ を 0 とすると，逆向きの流量 $f(j, i)$ が存在することと，容量に余裕があることを，どちらも $c(i, j) - f(i, j) > 0$ で判断することができる．図 3･25 に，フォード・ファルカーソン法のコードを示す．

```
class FordFulkerson
  def initialize(graph)
    @graph = graph
    @flow = Array.new(graph.order)
    @flow.each_index{|x| @flow[x] = Array.new(graph.order,0) }
  end
  def capacity(x,y)
    e=@graph.edge(x,y); return (e.nil?)?0:e
  end
  def flow(x,y)
    if x<y then @flow[x][y]; else -@flow[y][x]; end
  end
  def setFlow(x,y,f)
    if x<y then @flow[x][y] = f; else @flow[y][x] = -f; end
  end
  def depthFirst(n,goal,path,incCost)
    if n == goal then return incCost; end
    if !path.index(n).nil? then return nil; end
    (0..@graph.order-1).each {|x|
      aug = capacity(n,x)-flow(n,x)
      if aug > 0 then
        a = depthFirst(x,goal,path+[n],(aug<incCost)?aug:incCost)
        if (!a.nil?) then
          setFlow(n,x,flow(n,x)+a)
          return a
        end
      end
    }
    return nil
  end
  def solve(start,goal)
    until( depthFirst(start,goal,[],1000000).nil? ); end
    (0..@graph.order-1).each{|x|
      (0..@graph.order-1).each{|y|
        if (!@graph.edge(x,y).nil?) then
          print" (#x-#y) : #flow(x,y)\n"
        end
      }
    }
  end
end
```

図 3·25　フォード・ファルカーソン法のコード

3-2-4 最小全域木問題

グラフ G に対し，G の頂点をすべて含んだ G の部分グラフのうち木であるようなものを，**全域木（spanning tree）**と呼ぶ．全域木は，ネットワーク上の配送経路を求めるときなどに用いられる．更に，G の各辺に重みが設定されているとき，全域木のうちで，辺の重みの総和が最小であるようなものを**最小全域木（minimal spanning tree, MST）**と呼ぶ．

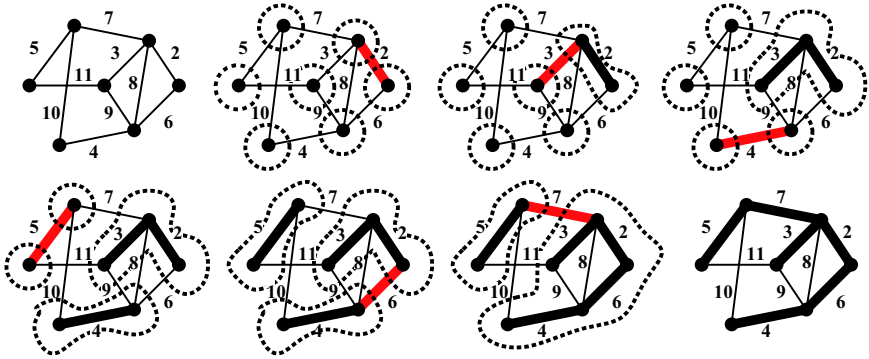


図 3･26　クラスカル法による最小全域木の計算例

クラスカル法（Kruskal Algorithm）は最小全域木を求める代表的なアルゴリズムであり，すべての辺について重みの小さいものから順に，閉路をつくらない限り，解となるグラフに加えていくことで木を構成する方法である（図 3･26）．閉路をつくらないことを判定するために，各頂点が属するクラスを考える．初期状態として，各頂点は，すべて異なるクラスに属する（すなわちすべてのクラスは頂点を一つだけ含む）とする．また，解グラフ T は，$(V, \emptyset)$ を初期値とする．辺を重みの小さい順に一つずつ取り出し，取り出した辺の両端の頂点が異なるクラス c_1, c_2 に属している場合は，その辺を T に加え，c_1 と c_2 を併合する．取り出した辺の両端が，同じクラスに属する頂点である場合には，何もしない．これをすべての辺について行うことで，最小な全域木 T が求められる．

図 3･27 にクラスカル法のコードを示す．`Kruskal` の初期化では，図 3･19 の `Graph` のインスタンスを与える．また，辺を重みの小さい順に取り出すために，図 3･16 に示したヒープを用いている．

```
class Kruskal
  def initialize(g)
    @graph = g
    @cluster = Array.new(g.order);
    @cluster.each_index{|i| @cluster[i] = i }
  end
  class Edge
    def initialize(u,v,c); @u=u; @v=v; @cost=c; end
    def ==(x); cost == x.cost; end
    def <=(x); cost <= x.cost; end
    def <(x);  cost <  x.cost; end
    def cost; @cost; end
    def u; @u; end
    def v; @v; end
  end
  def mergeCluster(c1,c2)
    @cluster.each_index{|i|
      if @cluster[i]==c1 then @cluster[i]=c2; end
    }
  end
  def solve
    heap = Heap.new()
    (0..@graph.order-1).each{|i|
      (0..@graph.order-1).each{|j|
        if !(e=@graph.edge(i,j)).nil? then
          heap.put(Edge.new(i,j,e))
        end
      }
    }
    edges = []
    until((e=heap.get()).nil?)
      if @cluster[e.u] != @cluster[e.v] then
        mergeCluster(@cluster[e.u],@cluster[e.v])
        edges << e
      end
    end
    edges.each{|e|
      print "(#{e.u},#{e.v})\n"
    }
  end
end
```

図 3･27　クラスカル法のコード

■参考文献

1) R. ディーステル 著, 根上生也, 太田克弘 訳, “グラフ理論（原題: Graph Theory）,” pp.157–163, シュプリンガー・フェアラーク東京, 2000.
2) 五十嵐健夫, “データ構造とアルゴリズム,” 新・情報/通信システム工学=TKC-3, pp.63–68, 数理工学社発行, サイエンス社発売, 2007.
3) Jan van Leeuwen 編, 廣瀬　健, 野崎昭弘, 小林孝次郎 監訳, “アルゴリズムと複雑さ（原題: Algorithm and Complexity）,” コンピュータ基礎理論ハンドブック I, pp.586–599, 丸善, 1990.
4) K. メールホルン, P. サンダース 著, 浅野哲夫 訳, “アルゴリズムとデータ構造 基礎のツールボックス,” pp.237–243, シュプリンガー・ジャパン, 2009.
5) 西原清一, “データ構造,” 新コンピュータサイエンス講座, pp.94–99, オーム社, 1993.
6) 紀平拓男, 春日伸弥, “プログラミングの宝箱 アルゴリズムとデータ構造 第 2 版,” pp.358–375, ソフトバンククリエイティブ, 2003.

■6 群 - 3 編 - 3 章

3-3 木のアルゴリズム

（執筆者：山口文彦 · 久野　靖）[2012 年 7 月 受領]

3-3-1 2 分探索木

全順序をつけられるデータとたかだか二つの子をもつノードからなる木（2 分木）を考える．簡単のため，木に記録されるデータに重複はないものとする．木の任意のノード n について，そこに記録されているデータを n_{val}，n の左の子である木に記録されたデータの最大値を $L_{\max}$，n の右の子である木に記録されたデータの最小値を $R_{\min}$ とするとき，$L_{\max} < n_{\mathrm{val}} < R_{\min}$ であるような木を **2 分探索木（binary search tree, BST）**と呼ぶ，図 3･28 に 2 分探索木の例を示す．

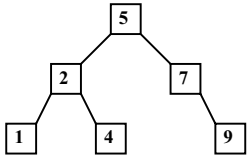


図 3･28　2 分探索木

このような木からあるデータ x が記録されているノードを探すことを考える．探索の手順は次のようになる．最初 n を根ノードとする．もし，$n_{\mathrm{val}} = x$ ならば n が目的とするノードである．$x < n_{\mathrm{val}}$ ならば左の子を，$n_{\mathrm{val}} < x$ ならば右の子を新しく n として，この手順を繰り返す．もし，子が存在しなければ，x をデータとするノードは存在しない．このアルゴリズムでは，比較を行う回数は木の高さに等しい．

このアルゴリズムは配列における 2 分探索とまったく同様である．ただし，配列の場合はデータの追加や削除のためにはその箇所より後ろの要素をすべてずらす必要があり，コストがかかる．これに対し，2 分探索木ではすぐ後に述べるアルゴリズムによって，やはり木の高さに等しいコストで挿入・削除が行える（図 3･29）．

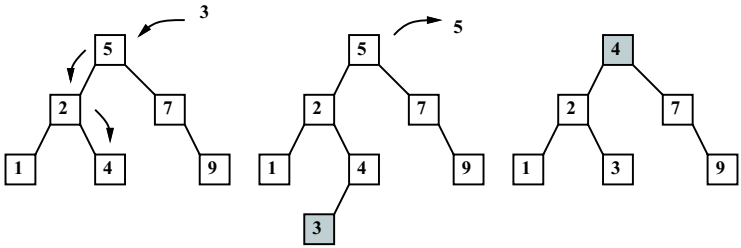


図 3･29　2 分探索木に対する挿入・削除

2 分探索木に新たなデータ x を登録することを考える．x が木に含まれていないならば，上記の探索手順にしたがって x を探索すると，少なくとも一方の子が存在しないノードに至る，そこで，そのノードの子として，x をデータとする新しい葉を追加すればよい．

2 分探索木からデータ x をもつノード n を削除することを考える．もし n が葉であれば，それを単に削除できる．n がたかだか一つしか子をもたないノードであれば，n の親に n の子をつなげればよい．問題は n が二つの子をもつ場合である．このときは，n の左の子である木のうちで最大の値を $l_{\max}$ として，この値を n に書き込めば，$L_{\max} < n_{\mathrm{val}} < R_{\min}$ が維持される．その代わりに，これまでの $l_{\max}$ を削除する必要があるが，これまで $l_{\max}$ を保持していたノードは (1) 葉であるか，(2) 左側の子だけをもつかのいずれかである（もし右の子があれば，その内容の方が大きいはずだから），(1)，(2) いずれでも，上記の方法で削除できる．

2 分探索木に対して，ランダムに n 個のデータを挿入した場合は，木の高さは平均して $O(\log n)$ 程度である．このため，平均では 2 分探索木は $O(\log n)$ で検索・挿入・削除ができ，効率がよい．しかし，値の昇順（ないし降順）にデータを挿入するような場合は偏った木ができてしまい，その高さが $O(n)$ となる．このような偏りの可能性があることが 2 分探索木の弱点である．

3-3-2 木のバランスと AVL 木

2 分探索木の探索における比較の回数は，根から目的の値をもつノードまでの距離に比例する．そこで木の高さを小さくし，木をバランスさせる手法が複数提案されている．ここではその代表的なものである，**AVL 木（AVL tree）**を紹介する．

AVL 木では，任意のノード n を根とする部分木について，「右側の部分木と左側の部分木の高さがたかだか 1 しか違わない (*)」という性質をもつ．根しかない木については，この性質は明らかに成り立つ．そこで，木に次々に要素を追加していく際に，(*) が維持されるようにすればよい．そのためには，前節で説明した 2 分探索木の挿入・削除を行った後で，(*) が成り立たなくなっていたら木を変形する．

これを行うため，すべてのノードについて，「左右の高さの差」の情報を保持し，挿入／削除によって木の高さが変化したときはそれに基づいてこの値が ± 1 を越えたことを検知する．あるノード R について，高さの差が 2 となったとき，例えば左側の方が 2 高くなったとする（左右対称なので右が高い場合も同様）．このとき，左側のノードの更に「左の子が高くなった場合（図 3･30 左）」と「右の子が高くなった場合（図 3･30 右）」の 2 通りの場合がある．ここで P，Q，R，及び部分木 A，B，C，D に含まれる値について，$A < P < B < Q < C < R < D$ であることに注意すると，いずれの場合も図 3･30 中のように構造を変更することでバランスした木となり，(*) の条件を維持できる．

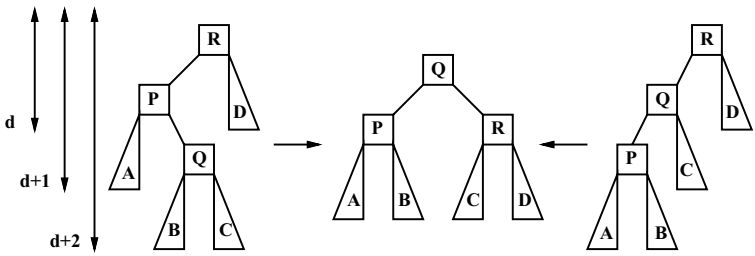


図 3･30　AVL 木のための木の変形

AVL 木は 2 分探索木のアンバランスの問題を解消しているが，「左右の高さの差が最大 1」という厳しい条件のため，変形の回数が多くなる傾向にある．これに対し，より緩い条件を採用した**赤黒木（red-black tree）**や，確率的アルゴリズムによりバランスを実現する**スプレー木（spray tree）**などの手法もよく知られている．

3-3-3 スキップリスト

木ではないが，2 分探索木の代替として使われるデータ構造に**スキップリスト（skip list）**がある．スキップリストは鍵の値の昇順にセルが並んだリストであるが，図 3･31 のように単連結リストが多段になっている．そして，一番下の段（レベル 0）はすべてのセルを順に結んでいるが，レベルが上がるごとに途中のセルをスキップする度合いが高くなり，ただしレベル i でスキップされたセルはレベル $i+1$ 以上でもやはりスキップされるようになっている．レベル $i+1$ 以上でスキップされるセルのことをレベル i のセルである，という．先頭と末尾にはすべてのキー値より小さい／大きい最大レベルのセルが常にあるものとしている．

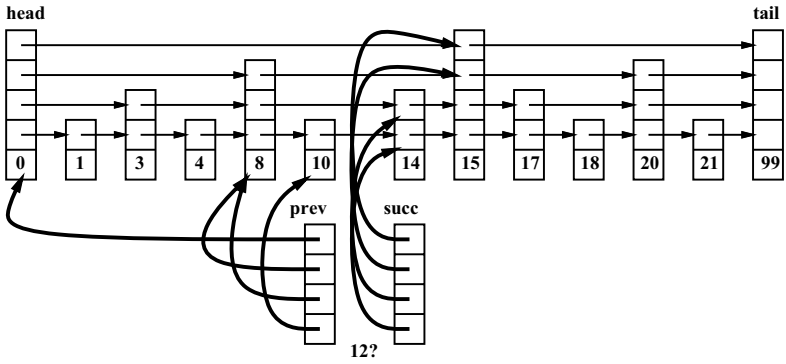


図 3･31　スキップリスト

スキップリストで鍵 k を探すときは，一番上のレベルで先頭から k 以上の値が現れるセルまでたどり，その直前とともに記録する．次に一つ下のレベルで，先に記録した直前から k 以上の現れるセルまでたどり，その直前とともに記録する．これをレベル 0 まで繰り返すことで，k を持つセルが見つかるか，そのようなセルがないことが分かる．

レベル $i+1$ のノードとレベル i のノードの比率（スキップ率 p）が仮に $\frac{1}{2}$ であり，レベル数が $\log_2 n$ 以上あるなら，各レベルごとにおよそ 1 個のノードを探せば済むので，探索の時間計算量は $O(\log n)$ に属する（段数を節約するために p をもっと小さくすることもでき，例えば $\frac{1}{4}$ なども使われる）．

鍵 k の挿入や削除は，まず k を上と同様にして探し，各レベルの「次」「前」のセルを記録することから始める．削除の場合は，単に削除するセルのレベル以下の部分において，「前」のセルの次が削除セルの「次」のセルになるようにリンクを変更すればよい．

挿入の場合は続いて乱数を用いて挿入するセルのレベルを決定する（$p = \frac{1}{2}$ であれば，50%の

確率でレベル 0，25%の確率でレベル 1，等々となる）．そして，そのレベルのセルを挿入し，それがもつ次へのリンクは先に記録した「次」のセルにより，「前」のセルの次へのリンクについては，挿入したセルのレベル以下の部分のみについて，挿入したセルに変更する．例えば図 3･31 において「12」のセルを挿入するとき，レベルがいくつであっても記録してある「前」「次」からそのレベル数分を用いて作業することでスキップリストの条件が維持できることが分かる．

図 3･32，図 3･33 にスキップリストのコードを示す．図 3･33 のメソッド `compLevel` は，スキップ率 0.5 を用いて乱数でレベルを決定する．またクラス `Cell` はセルオブジェクトのクラスで，レベル，鍵，値，及び次のセルへのリンクを並べた配列をもつ．各インスタンス変数へのアクセサを自動生成しており，ほぼレコードとして使っている．リンクの配列は簡単のため，レベルにかかわらず全レベル分の大きさとする．

図 3･32 が本体で，初期設定時にレベル数，最大／最小の値を受け取り記録するとともに，先頭と最後のセルを割り当て，また「前」「次」を記録する配列も用意しておく．`search` は検索を行うが，その際に「前」「次」の記録も行い，`add` と `delete` はまず `search` を呼んで「前」「次」を適切に設定してもらってから，これに基づいて挿入と削除を行っている．簡単のため，既にある鍵を挿入したり，存在しない鍵を削除することの検査は省いている．また，挿入時には `serach` が用意した「次」の配列をそのまま新しいセルに与え，自分用に新たな配列を作成している．

スキップリストは挿入時に乱数によってレベルを割り当てるので，偏りが起きる心配がない（削除の結果偏りが生ずることはあり得るが，その場合は全セルのレベルを再度乱数で割り当て直せばバランスさせられる）．また，挿入や削除を局所的に行えることから，並列アルゴリズムとの相性がよい．

```
class SkipList
  def initialize(mLevel = 8, min = 0, max = 9999)
    @mLevel = mLevel; @min = min; @max = max
    @tail = Cell.new(mLevel-1, max, 0, Array.new(mLevel, nil))
    @head = Cell.new(mLevel-1, min, 0, Array.new(mLevel, @tail))
    @succ = Array.new(mLevel); @prev = Array.new(mLevel)
  end
  def search(key)
    pre = @head; cur = nil; found = false
    (@mLevel-1).step(0, -1) do |level|
      cur = pre.succ[level]
      while key > cur.key
        pre = cur; cur = pre.succ[level]
      end
      if key == cur.key
        found = true
      end
      @prev[level] = pre; @succ[level] = cur
    end
    return found ? cur : nil
  end
  def add(key, val)
    search(key)
    level = compLevel(@mLevel)
    node = Cell.new(level, key, val, @succ)
    for l in 0..level
      @prev[l].succ[l] = node
    end
    @succ = Array.new(@mLevel, nil)
  end
  def delete(key)
    cur = search(key)
    cur.level.step(0, -1) do |level|
      @prev[level].succ[level] = cur.succ[level]
    end
  end
end
```

図 3･32　スキップリストのコード（図 3･33 につづく）

```
def compLevel(mLevel)
  for level in 0..mLevel-1
    if rand < 0.5 || level == mLevel-1
      return level
    end
  end
end

class Cell
  attr_reader :level, :key, :succ
  attr_accessor :val
  def initialize(level, key, val, succ)
    @level = level; @key = key; @val = val; @succ = succ
  end
end
```

図 3･33　スキップリストのコード（つづき）

3-3-4 B 木と B+木

探索木は 2 次記憶に配置することもあるが，その場合は 2 次記憶のアクセス回数をできる限り減らしたいので，木の高さを小さくしたい．このためには，2 分木よりも次数の高い多分木に基づくデータ構造が望ましい．

B 木（とその変形である B+木）は，(1) 根からすべての葉までの高さが等しい（バランスしている），(2) 挿入・削除に際して変更されるノード数が少ない，(3) 次数を任意に大きくできる（ディスクブロックに収まるように決められる），という性質をもつため，データベースをはじめ 2 次記憶上のデータ構造として広く使われている．一方で，各ノードに未使用領域ができることがあるが，空き領域の比率はノードの容量のおよそ半分を越えることはない（2 次記憶上のデータ構造では，容量には余裕があることが普通なので，使用率の下限が 50%であることは問題にならないことが多い）．

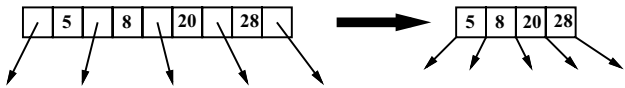


図 3･34　B 木のノード

次数 m の **B 木（B-tree）**とは，各内部ノードが最大 m 本の枝をもつ木構造であり，それぞれのノードは図 3･34 左のように枝に「挟まれた」かたちで $m-1$ 個の鍵（とデータ，図では省略）をもつことができる（枝の本数が m より少ない場合も同じで，常に鍵の数は枝の数より 1 少ない）．以下では図を簡潔にするため，図 3･34 右のように鍵と鍵の境目からリンクが出ているように描く．

次数 m の B 木は，次の性質をもつ．B 木における要素の挿入／削除に際しては，この条件を壊さないようにする必要がある．

- ルートは葉であるか，または少なくとも 2 つの子をもつ．

- ルート以外のノードは，$\lceil \frac{m}{2} \rceil$ 以上 m 以下の子をもつ．
- すべての葉は同じレベルにある．

図 3･35 に次数 3 の B 木を示す．要素の検索については，各ノードに記録されている鍵と見比べながら下へたどっていくことで，鍵が見つかるか，または葉まできてその鍵が含まれていないことが分かる．

次に，左側の木に 18 を挿入したいとする，挿入する位置は上と同様にして検索することでみつける（挿入は常に葉の位置で起きる）．今の場合は「20，21」のノードに挿入するが，このノードは既に満杯なので「18，20，21」の中央値 20 を決め，それより小さい 18，それより大きい 21 をそれぞれ一つずつのノードに割り当てる．中央値 20 は，一つ上のノードに格納する必要がある．しかしここでは，一つ上のノード「6，15」も満杯である．このため，「6，15，20」の中央値で分け，6 のノードと 20 のノードをつくり，残った中央値は新しいルートとなる（ルートは個数の制約はないことに注意，次数が高い B 木でも 1 個しか値をもたないルートはあり得る）．

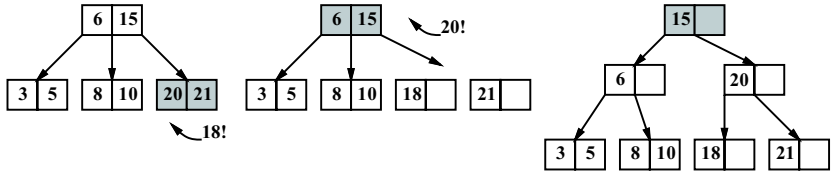


図 3･35　次数 3 の B 木

このように，B 木では要素の挿入によってノードの分割が起き，結果としてルートが満杯になったときに高さが増す．逆に削除の場合はノードの併合が行われ，ルートの直下での併合の結果木が低くなることがある．

ここまでは，途中ノードも葉ノードも同じ構造であるものとしてきたが，実際のデータベースなどでは葉の部分にデータを格納し，中間ノード以上はインデックスのみとして利用することが普通である．このように，葉のみにデータを置くように B 木を変更したものが **B+木（B+-tree）**である．B+木の場合は，中間ノードの値はあくまでもインデックスなので，途中ノードで一致するキーが見つかっても葉までたどっていく必要がある．

また，B+木では，同一レベルのノードどうしを双方向リンクでつなげることにより，葉ではデータを順番に取り出すことができ，また中間ノードでは挿入や削除に際して空き領域を融通して分割・併合の頻度を減らすようにしている．

■参考文献

1) 五十嵐健夫, “データ構造とアルゴリズム,” 新・情報/通信システム工学=TKC-3, pp.23–33, 数理工学社発行, サイエンス社発売, 2007.
2) 紀平拓男, 春日伸弥, “プログラミングの宝箱 アルゴリズムとデータ構造 第 2 版,” pp.282–302, ソフトバンククリエイティブ, 2003.
3) C. シェーファー 著, 久野禎子, 久野　靖 訳, “Java によるデータ構造とアルゴリズム解析入門,” pp.328-343, 349–353, ピアソンエデュケーション, 2000.

■6 群 - 3 編 - 3 章

3-4 文字列探索

（執筆者：延澤志保）［2012 年 7 月 受領］

3-4-1 文字列探索アルゴリズム

(1) 文字列探索の概要

文字列探索（string searching）とは，文字列中のある部分文字列を機械的に発見する処理であり，何らかの方法で文字列の内容を**走査（scan）**し，探索対象文字列を含む箇所を特定することを目的としている．

探索対象の文字列を**パターン（pattern）**と呼ぶ．例えば，図 3･36 の文書中の文字列 **all** を探すことを考えるとき，この **all** がパターンである．それに対して，探索対象となる文字列は文書全体を一つの文字列とみなしたものであり，これを**テキスト（text）**と呼ぶ．

As soft as silk, as white as milk,
As bitter as gall, a strong wall,
And a green coat covers me all.

図 3･36　検索対象とする文書の例（英語）

ある二つの文字列が何らかの基準に照らして一致するとみなせるかを調べる処理を**照合（match）**という．すなわち，文字列探索とは，テキストの各部分文字列に対してパターンとの照合を行い，一致する箇所を見つけ出す処理である．テキストは 1 次元の文字の並びであり，これを何らかの順に走査しながら照合を行う．

文字列探索は，その目的によって，パターンがテキストの中に含まれているかを調べる場合と，テキスト中のすべての出現箇所を調べるものとに分けられる．前者は，パターンと一致する文字列が 1 個みつかればよい．それに対して後者は，パターンと一致する文字列がテキスト中に出現するすべての箇所を特定する処理である．また，探索の成否については，テキストの部分文字列とパターンが完全に一致する場合のみ一致とする場合（exact match）と，類似した文字列についても探索する場合とがある．

文字列探索の基本的な考え方では，完全一致とはパターンとテキストの部分文字列とが一致する場合を指す（例えば図 3･36 の **wall** の後ろ 3 文字に一致する場合も含む）．自然言語処理の観点では，**all** と **wall** は別の単語であり，**all** の探索結果に **wall** が含まれることは望ましくないため，単語の区切りを意識し，単語単位で一致する場合のみ一致とみなすことが多い．しかし，派生語（**walk** に対して **walked** など）や表記の揺れ（**color** と **colour** など）を含めた探索が必要となる場合も多く，これは**近似的文字列照合（approximate string match）**と呼ばれている．本節は文字列探索のアルゴリズムとしての説明のため，単語としての意味にこだわらず，単純な文字列の照合を扱う．

(2) 単純な文字列探索アルゴリズム

最も単純な探索アルゴリズムは，テキストの先頭から順にテキストの部分文字列を走査し，パターンとの照合を行う力任せ法である（Brute-force Algorithm，Naïve String Search などと呼ばれる）．

簡単な例として，**みちにまよったみちる**とのテキストに含まれるパターン**みちる**を探す場合を考える（図 3･37）．なお，この節では，パターン中の二重下線が現在走査中の文字（失敗箇所），パターン及びテキストの下線は次に走査する文字を示す，またテキスト中の斜め線付きの文字はその文字以外の文字であることを示す．例えば，$\underline{\not{\mathbf{A}}}$ は **A** 以外の文字を意味する．テキスト及びパターンの太字の文字は一致が確認された文字を示す．

この例では，テキストの頭から順に 1 文字ずつ照合を行っている，3 文字すべてマッチすれば照合成功で，そのときの 1 文字目の場所（この例であれば，「先頭から 8 文字目」）を出力として返す．すなわち，検索は二重ループとなっており，文字列照合の開始文字位置 i とパターンの何文字目を照合しているかを示す位置 j との組（i , j）について，i を 1 文字ずつずらしながら文字列をチェックしていく．これをテキストの末尾まで繰り返すことで，該当箇所をすべて探し出すことができる．

	み	ち	に	ま	よ	っ	た	み	ち	る
(1,1)	み									
(1,2)	み	ち								
(1,3)	み	ち	に							
(2,1)		み								
(3,1)			み							
(4,1)				み						
(5,1)					み					
(6,1)						み				
(7,1)							み			
(8,1)								み		
(8,2)								み	ち	
(8,3)								み	ち	る

図 3･37　検索対象とする文章の例（日本語）

テキストの 1 文字目とパターンの 1 文字目はともに**み**で一致するので，テキストの 1 文字目から連なる文字列がパターンと一致する可能性があると考え，i を 1 文字目に固定したまま，照合文字 j を 2 文字目に進める．これも一致するため照合文字を 3 文字目に進めるが，ここでテキストの 3 文字目は**に**，パターンの 3 文字目は**る**と不一致が起こるので，$i = 1$ でのマッチに失敗する．$i = 1$ についての照合はここで打ち切り，i を 1 進めて同様の照合処理を続ける．$i = 8$ のとき，テキストの 8 文字目以降の 3 文字がパターンの 3 文字と一致するため，このときの i の値を検索結果として出力する．

この例では，パターンが 3 文字とも異なる文字であるため，例えばテキストの 2 文字目がパターンの 2 文字目と一致するのであれば，テキストの 2 文字目がパターンの 1 文字目と一致することはない．したがって，(2, 1) の照合は不一致となることが明らかであり，この照合を行うことは冗長である．この点を考慮しない単純なアルゴリズムでは，最悪の場合 $O(m \times n)$ となる．ただしこれはテキスト上のそれぞれの文字に対してパターン全体を比較しないといけないような場合である．具体的には，テキストとパターンがともに同じ 1 種類の文字の羅列（a^n）の場合，比較回数は最大となる．一般に，テキストに含まれる文字の種類が十分に多い場合には計算量は $O(n)$ 程度となる．以下では，検索を効率よく行う手法として，基本的な KMP 法，BM 法を紹介する．

3-4-2 KMP 法

(1) KMP 法の概要

KMP 法（Knuth-Morris-Pratt Algorithm）[2] は，パターンの各文字位置に対して，その文字位置で照合に失敗した場合に走査を再開する場所を事前に調べておくことで探索を効率化する手法である．走査を再開する場所の情報はシフトテーブルに保管し，冗長な照合を回避する．

図 3･38 に，パターン**ちるちるみちる**の検索について，照合に失敗した箇所と次の操作開始箇所の関係を示す．対象のテキストについては，既に走査の済んだ文字はひらがなで，まだ走査していない文字は X で示している．これには，次の走査の対象となる文字も含むものとする．（走査済み及び未走査の表記については次節以降でも同様とする．）

	照合失敗位置	シフト幅	1	2	3	4	5	6	7	8	9	10	11	12	13
1	テキスト		ち	X	X	X	X	X	X	X	X	X	X	X	X
	パターン（現）		ち	る	ち	る	み	ち	る						
	パターン（次）	1		ち	る	ち	る	み	ち	る					
2	テキスト		ち	る	X	X	X	X	X	X	X	X	X	X	X
	パターン（現）		ち	る	ち	る	み	ち	る						
	パターン（次）	1		ち	る	ち	る	み	ち	る					
3	テキスト		ち	る	ち	X	X	X	X	X	X	X	X	X	X
	パターン（現）		ち	る	ち	る	み	ち	る						
	パターン（次）	3				ち	る	ち	る	み	ち	る			
4	テキスト		ち	る	ち	る	X	X	X	X	X	X	X	X	X
	パターン（現）		ち	る	ち	る	み	ち	る						
	パターン（次）	3				ち	る	ち	る	み	ち	る			
5	テキスト		ち	る	ち	る	み	X	X	X	X	X	X	X	X
	パターン（現）		ち	る	ち	る	み	ち	る						
	パターン（次）	2			ち	る	ち	る	み	ち	る				
6	テキスト		ち	る	ち	る	み	ち	X	X	X	X	X	X	X
	パターン（現）		ち	る	ち	る	み	ち	る						
	パターン（次）	6							ち	る	ち	る	み	ち	る
7	テキスト		ち	る	ち	る	み	ち	る	X	X	X	X	X	X
	パターン（現）		ち	る	ち	る	み	ち	る						
	パターン（次）	6							ち	る	ち	る	み	ち	る

図 3･38　照合失敗箇所と走査開始箇所の関係

1 文字目で照合に失敗した場合には，パターンを 1 文字ずらして走査を再開する．2 文字目以降で照合に失敗した場合には，基本的には，テキスト上の失敗した文字位置までパターンをずらして走査を再開すればよい．ここでは，2 文字目，4 文字目及び 7 文字目で失敗した例がこれに当たる．この例では，テキストのある文字とパターンのある文字が一致するか否かの判定のみを行っているものとしてシフト幅を示している．照合する際にはテキスト側の文字を走査するので，照合に失敗した際この文字がパターンの 1 文字目と一致するか否かを調べることによって，シフト幅を 1 多くとることもできる．これは結局，1 文字目で失敗した場合の処理と同じなので，ここでは 2 文字目以降で失敗した場合はテキスト側の文字にかかわらず失敗箇所へパターンをずらして再開することとする．

3 文字目，6 文字目での失敗例のように，パターンの失敗箇所の文字とパターンの 1 文字目とが同じ文字の場合には，失敗箇所の次の文字までずらすことができる．それに対して 5 文字目で失敗した例では，パターン中に複数含まれる文字がテキスト中照合に成功した箇所（1–4 文字目）に複数回含まれているため，その場所へパターンをずらして走査を再開する必要がある．このシフト幅はテキストに依存しないので，与えられたパターンに対して事前にシフトテーブルを作成することで，冗長な照合を回避することが可能である．

（2）KMP 法の効率

KMP 法はバックトラックをしないため，検索対象テキストのサイズを n とすると，その探索効率は $O(n)$ で表せる．KMP 法では，事前のシフトテーブルの作成プログラムと本体となる検索プログラムの 2 種類のプログラムを必要とする．検索パターンの長さを m とするとシフトテーブル作成プログラムは $O(m)$ であり，全体として，最悪でも $O(m+n)$ である．

3-4-3 BM 法

（1）BM 法の概要

BM 法（Boyer-Moore Algorithm）[1] は，KMP 法と同様，事前にシフトテーブルを用意することで無駄な走査を減らして探索の効率化を図る手法である．KMP 法との違いは，パターンの末尾から走査を行うことと，文字の種類ごとのシフトテーブルも作成することである．照合に失敗した箇所のテキストの文字に基づくシフト幅（bad character heuristics）と，照合に失敗するまでに一致していた文字列のパターン内での出現箇所に基づくシフト幅（good suffix heuristics）の 2 種類の尺度でシフト幅を考慮することが，BM 法の特徴である．

図 3･39 に，パターン**ちるちるみちる**の検索について，照合に失敗した箇所と次の操作開始箇所の関係を示す．

まず，テキストの i 文字目とパターンの 1 文字目を合わせ，パターン末尾である 7 文字目とテキストの $i+7-1$ 文字目を照合する．例 7 では，**あ**はパターンには出現しない文字なので，パターンを 1–6 文字ずらしても一致する箇所はないことが明らかである．したがって，この例のようにパターンに含まれない文字の出現により照合に失敗した場合には，パターンの長さの分パターンをずらすことができる．それに対して，例 7′ などのように失敗した箇所の文字がパターン中に含まれる場合には，その文字に一致する箇所までパターンをずらすことで，無駄な走査を回避できる．例 7″ 及び例 5 では，同じ文字列がパターン中に複数回出現するため，複数のシフト幅が考えられる（次 1，次 2 など）．

（2）BM 法のアルゴリズム

テキスト中の文字に基づくシフトテーブルの作成（bad character heuristics）では，パターン中で各文字が最も末尾に近い位置で出現する場合について，末尾からの距離を記録すればよい（表 3･1）．

パターン中で出現する文字以外の文字で失敗した場合には，現時点でパターンに対応させているテキスト部分が照合に成功することはないので，パターンの長さ分だけパターンをずらして走査を再開する．

照合の済んだ文字列に基づくシフトテーブルの作成（good suffix heuristics）では，パターンのすべての接尾辞について，パターン内での出現箇所を調べる．ここでの接尾辞（suffix）は文字列の任意の位置から末尾までの部分文字列を指し，文字列そのものはこれに含めない．

	照合失敗位置	シフト幅	1	2	3	4	5	6	7	8	9	10	11	12	13	14
7	テキスト		X	X	X	X	X	X	あ	X	X	X	X	X	X	X
	パターン（現）		ち	る	ち	る	み	ち	る							
	パターン（次）	7								ち	る	ち	る	み	ち	る
7′	テキスト		X	X	X	X	X	X	み	X	X	X	X	X	X	X
	パターン（現）		ち	る	ち	る	み	ち	る							
	パターン（次）	2			ち	る	ち	る	み	ち	る					
7″	テキスト		X	X	X	X	X	X	ち	X	X	X	X	X	X	X
	パターン（現）		ち	る	ち	る	み	ち	る							
	パターン（次 1）	1		ち	る	ち	る	み	ち	る						
	パターン（次 2）	4					ち	る	ち	る	み	ち	る			
	パターン（次 3）	6							ち	る	ち	る	み	ち	る	
6	テキスト		X	X	X	X	X	み	る	X	X	X	X	X	X	X
	パターン（現）		ち	る	ち	る	み	ち	る							
	パターン（次）	7								ち	る	ち	る	み	ち	る
5	テキスト		X	X	X	X	る	ち	る	X	X	X	X	X	X	X
	パターン（現）		ち	る	ち	る	み	ち	る							
	パターン（次 1）	3				ち	る	ち	る	み	ち	る				
	パターン（次 2）	5						ち	る	ち	る	み	ち	る		
4	テキスト		X	X	X	ち	み	ち	る	X	X	X	X	X	X	X
	パターン（現）		ち	る	ち	る	み	ち	る							
	パターン（次）	5						ち	る	ち	る	み	ち	る		
3	テキスト		X	X	る	る	み	ち	る	X	X	X	X	X	X	X
	パターン（現）		ち	る	ち	る	み	ち	る							
	パターン（次）	5						ち	る	ち	る	み	ち	る		
2	テキスト		X	ち	ち	る	み	ち	る	X	X	X	X	X	X	X
	パターン（現）		ち	る	ち	る	み	ち	る							
	パターン（次）	5						ち	る	ち	る	み	ち	る		
1	テキスト		る	る	ち	る	み	ち	る	X	X	X	X	X	X	X
	パターン（現）		ち	る	ち	る	み	ち	る							
	パターン（次）	5						ち	る	ち	る	み	ち	る		

図 3･39　BM 法における照合失敗箇所と走査開始箇所の関係

その際，各接尾辞の前に付く文字を考慮する（パターンの冒頭部分については，冒頭部分に一致する部分のみ考慮すればよい）．例えば，パターン末尾 1 文字の接尾辞（**る**）はパターン 2 文字目及び 4 文字目にも出現するが，3 箇所すべてにおいて直前の文字は同じ**ち**であり，7 文字目で照合に失敗した場合には 2 文字目及び 4 文字目でも同じように失敗することが明らかである．それに対して，パターン末尾 2 文字の接尾辞（**ちる**）はパターン 1–2 文字目及び 2–3 文字目にも出現するが，その直前の文字はそれぞれ異なっており，6–7 文字目の**ちる**が 5 文字目で照合に失敗した場合でも，ほか 2 箇所の**ちる**をテキストの同じ位置に合わせた場合にも照合に失敗するとは限らない．

照合に失敗した位置に基づくシフトテーブルの作成では，パターン中で各文字が最も末尾に近い位置で出現する場合について，末尾からの距離を記録すればよい（表 3･1）．

末尾の 1 文字での照合で失敗した場合には，まだ一致した文字列はないので，単純に 1 文字分パターンをずらして走査を再開すればよい．一致した文字列がある場合には，これの部

表 3･1　照合失敗文字に基づくシフトテーブルの例

文字	シフト幅
ち	1
み	2
る	0
その他	7

分文字列がパターン中に複数存在し，かつその直前の文字が失敗した文字に一致しない場合のみ，次に末尾に近い出現箇所までパターンをずらす．それ以外の場合には，現時点でパターンと対応させているテキスト部分が照合に成功することはないため，パターンの長さ分だけパターンをずらして再開する．

このように，BM 法では，照合に失敗した箇所のテキストの文字に基づくシフトテーブル（表 3･1）と照合に失敗した箇所のパターン中での位置に基づくシフトテーブル（表 3･2）の二つのシフトテーブルを利用し，当てはまる条件のうち大きい方のシフト幅を採用する．したがって図 3･39 の例 7″ では 1 文字シフト（次 1）し，また例 5 では 3 文字シフト（次 1）する．

表 3･2　照合失敗位置に基づくシフトテーブルの例

失敗位置	文字列	シフト幅
7	**る**	1
6	**ち る**	7
5	**み ちる**	3
4	**る みちる**	5
3	**ち るみちる**	5
2	**る ちるみちる**	5
1	**ち るちるみちる**	5

（3）BM 法の効率

BM 法では，パターン中に現れない文字がテキスト中に出現した場合にはパターン 1 個分シフトすることができるため，テキスト中にこのような文字が多い場合に効果的である．逆に，テキスト及びパターンに出現する文字の種類が少ない場合にはシフト幅が小さくなり，効率が悪くなる．最悪の場合は $O(m \times n)$ となるが，平均的には $O(n/m)$ であり，KMP 法よりも効率がよい．

BM 法は効率のよいアルゴリズムだが，テキストに含まれるすべての文字についてシフトテーブルを作成する必要がある点で，日本語のように文字の数の多い言語では効果的な利用が難しいものと考えられる．

（4）BM 法の応用

BM 法ではシフトテーブルの作成がオーバヘッドとなるため，二つのシフトテーブルのうち片方のみを利用するような方法も多く用いられる．テキスト中の照合失敗位置の文字に基

づくシフトテーブルのみを用いる場合，パターンによっては，テキスト中の照合失敗位置の文字に基づくシフトテーブルのシフト幅がマイナスになることがある（照合に失敗した時点のパターン中の位置よりもパターン中後方に同じ文字が出現する場合）．このような場合バックトラックを起こし，最悪の場合には無限ループに陥るので，こういった場合にはシフト幅を適切に調整する（例えば，バックトラックを起こす場合にはシフト幅を 1 とする，など）必要がある．

BM 法の改良アルゴリズムとしては，Horspool 法[3]，Sunday 法[4]など様々なものがある．

3-4-4 正規表現

(1) 正規表現による探索の概要

KMP 法，BM 法はともに，パターンと完全に一致する文字列の探索を対象としたアルゴリズムである．これらは，パターンの前後にほかの文字列が付着した文字列（例えば，パターン**パターン**に対して**パターン群**や**サブパターン**など）を検出することは可能だが，例えば**パタン**のようにパターンの一部分が変化したものについては検出ができない．このような曖昧性をもつパターンの探索では，正規表現が有効である．

正規表現とは，文字列の表現方法の一つであり，類似した文字列をグループ化して扱うことができる表現方法である．例えば，**abc** のような曖昧性のない文字列は，正規表現でもそのまま **abc** と表記する．それに対して，例えば **abc** と似た文字列として **abccc** など末尾の **c** が複数回連続して出現する文字列や **adc** など 2 文字目が異なる文字列なども同じように扱いたい場合には，これらを $\mathbf{a.c}^{+}$ などのようにメタ文字を使って表現することで両方を一度に表現することができる．また正規表現の照合過程は，決定性有限オートマトンで表現することができる．

正規表現では，文字またはメタ文字の連結，論理和，及びその繰り返しの記述で文字列を表現する．例えば文字 **a** と **b** がこの順に並んで出現する文字列 **ab** を **a** と **b** の連結と表現する．そのうえで，文字の選択（論理和）及び文字の繰り返しを表すメタ文字を利用して曖昧な文字列を表現する（表 3･3）．

表 3･3　正規表現の表記法

表記法	意味	利用例	利用例に対応する文字列の例
.	**任意の 1 文字 (改行コードを除く)**	**a.c**	**aac,abc,acc,a1c,･･･**
|	**論理和**	**a | b**	**a,b**
[ ]	**文字クラス内の任意の 1 文字**	**[abc]**	**a,b,c**
		[0-9]	**任意の半角数字 1 文字**
		[a-zA-Z]	**任意の半角英字 1 文字**
[^]	**文字クラスの否定**	**[^0-9]**	**半角数字以外の任意の文字**
*****	**0 回以上の繰り返し**	$\mathbf{abc}^{*}$	**ab,abc,abcc,abccc,･･･**
+	**1 回以上の繰り返し**	$\mathbf{abc}^{+}$	**abc,abcc,abccc,･･･**
?	**0 回または 1 回の出現**	**abc?**	**ab,abc**
( )	**グループ化**	$\mathbf{a(bc)}^{+}$	**abc,abcbc,abcbcbc,･･･**
^	**行頭**	**^abc**	**行頭に現れる abc**
$	**行末**	**abc$**	**行末に現れる abc**

(2) 正規表現の定義

正規表現では文字列の連結は「・」で表すが，これは明らかな場合には省略される．例えば，**a**・**bc** は文字列 **a** と文字列 **bc** の連結を示し，**abc** とも書く．s^n は文字列 s の n 回の繰り返し（$ss\cdots s$）を示し，s^0 は空文字列（ε）である．空文字列 ε との連結はもとの文字列と等価である（$\varepsilon \cdot t = t \cdot \varepsilon = t$）．

ある言語 L の文字集合（アルファベット）を $\Sigma = \{a_1, a_2, \cdots, a_n\}$ とすると，$L(a_i) = \{a_i\}$ は言語 Σ 上の正規表現である．空文字列 $L(\varepsilon) = \{\varepsilon\}$ も Σ 上の正規表現である．$L(r)$ を Σ 上の正規表現とすると，$r_1 \mid r_2$ も正規表現であり $L(r_1) \cup L(r_2)$ を表す．$r_1 \cdot r_2$ も正規表現であり，$L(r_1) \cdot L(r_2)$ を表す．r^* も正規表現であり，$L(r)^*$ を表す．これらの正規表現の組合せで表現できるものもまた正規表現であり，また，これ以外は正規表現ではない．

ある言語 L の文字列の集合 A に対して，A の閉包 A^* は A の要素の任意の個数の連結を示す（$A^* = A^0 \cup A^1 \cup A^2 \cup \cdots = \cup_{k=0}^{\infty} A^k$）．特に，$A$ の要素の 1 回以上の連結を正の閉包と呼び，A^+ で表す（$A^+ = A^1 \cup A^2 \cup \cdots = \cup_{k=1}^{\infty} A^k$，ゆえに $A^* = \{\varepsilon\} \cup A^+$）（例えば，$A = \{\mathbf{a}\}$ の場合，$A^+ = \{\mathbf{a}, \mathbf{aa}, \mathbf{aaa}, \cdots\}$，$A^* = \{\varepsilon, \mathbf{a}, \mathbf{aa}, \mathbf{aaa}, \cdots\}$ である）．

(3) 正規表現を利用した文字列探索

正規表現を利用した文字列探索は，perl，AWK などのスクリプト言語などで利用可能である．文字のグループ化（文字列をひとかたまりとして扱う表現）や行頭，行末を示すメタ文字など，文字列探索のためのパターンの表現能力を高めるようなメタ文字も定義されている（表 3･4）．

表 3･4　正規表現で利用可能なメタ文字

メタ文字	使い方	利用可能な範囲
\	エスケープ	パターン内
.	任意の 1 文字	パターン内（[ ] 内を除く）
+	パターンの 1 回以上の繰り返し	パターン内（[ ] 内を除く）
*	パターンの 0 回以上の繰り返し	パターン内（[ ] 内を除く）
(	グループ化	パターン内（[ ] 内を除く）
)	グループ化	パターン内（[ ] 内を除く）
[	文字の集合	パターン内（[ ] 内を除く）
]	文字の集合	パターン内
{	繰り返し回数の指定	パターン内（[ ] 内を除く）
}	繰り返し回数の指定	パターン内（[ ] 内を除く）
?	パターンの 0 回または 1 回の出現	パターン内（[ ] 内を除く）
^	否定	[の直後
	パターン先頭	パターン先頭（[ ] 内を除く）
,	繰り返し回数の指定	{ } 内
|	論理和	パターン内（[ ] 内を除く）
$	パターン末尾	パターン末尾（[ ] 内を除く）
-	文字クラス範囲の指定	[ ] 内

メタ文字は正規表現中では特殊な意味をもつため，これらの文字を探索対象とする場合には，直前にエスケープ文字 \（半角）を置く（例えば，*という文字そのものを探索対象とし

たい場合には *と記述する）．なお，**[ ]** で囲まれた範囲では，^，-，[，] の 4 種類以外のメタ文字は，メタ文字としてではなく一般の文字として扱われる（例えば，**[*+?]** は*****，**+**，**?**のいずれか 1 文字を示す）．

正規表現ではこのほか，改行コードを示す**\n** など，エスケープシークエンス（エスケープシーケンス）と呼ばれる特殊な表記が利用可能である（表 3･5）．エスケープシークエンスには，perl，AWK など正規表現を利用した文字列処理を行うスクリプト言語独自のものが定義される場合もある（表 3･5 後半）．

表 3･5　正規表現で利用可能なエスケープシークエンス

エスケープシークエンス	使い方	等価な正規表現
\a	0x07（Alart）	
\b	0x08（Back Space）	
\e	**エスケープコード**	
\f	0x0c（Form Feed）	
\n	0x0a（New Line）	
\r	0x0d（Carriage Return）	
\t	0x09（Horizontal Tab）	
\v	0x0b（Vertical Tab）	
\ooo	8 **進数文字コード（***ooo* **部に** 1〜3 **桁で指定）**	
\xhh,*\Xhh*	16 **進数文字コード（***hh* **部に** 1〜2 **桁で指定）**	
\d	**半角数字**	**[0-9]**
\D	**半角数字以外**	**[^0-9]**
\s	**空白文字**	**[\t\r\n\f]**
\S	**空白文字以外**	**[^ \t\r\n\f]**
\w	**半角英数字およびアンダースコア**	**[0-9a-zA-Z_]**
\W	**半角英数字およびアンダースコア以外**	**[^0-9a-zA-Z_]**
\z	EOF	

（4）表現可能性

正規表現は正規言語の実現であり，これを用いた文字列探索は種々のスクリプト言語をはじめとする様々なプログラミング言語で対応している．

正規言語はチョムスキー階層の 4 レベルの形式言語のうち最も制約が厳しく，数の概念の扱いが難しいなど，複雑なパターンが表現できない場合がある．そのため，自然言語処理の分野では，文法規則等の複雑なパターンの記述など，更に表現可能性の高い形式言語である文脈自由言語を利用する場面も多い．

■参考文献

1) R.S. Boyer and J.S. Moore, “A Fast String Searching Algorithm,” Communications of the Association for Computing Machinery, vol.20, no.10, pp.262–272, 1977.
2) D.E. Knuth, J.H. Morris, Jr, and V.R. Pratt., “Fast Pattern Matching in Strings,” SIAM Journal on Computing, vol.6, no.1, pp.323–350, 1977.

3) R.N. Horspool, “Practical Fast Searching in Strings,” Software-Practice and Experience, vol.10, no.6, pp.501–506, 1980.
4) D.M. Sunday, “A Very Fast Substring Search Algorithm,” Communications of the Association for Computing Machinery, vol.33, no.8, pp.132–142, 1990.
5) 長尾　真 編, “自然言語処理,” 岩波講座ソフトウェア科学 15, 岩波書店, 1996.
6) 長尾　真, 宇津呂武仁, 島津　明, 匂坂芳典, 井口征士, 片寄晴弘, “文字と音の情報処理,” 岩波講座マルチメディア情報学 4, 岩波書店, 2000.
7) 高橋恒介, “テキスト検索プロセッサ,” 電子情報通信学会, 1991.

■6 群 - 3 編 - 3 章

3-5 状態空間とゲーム木の探索

（執筆者：山口文彦）［2012 年 7 月 受領］

3-5-1 状態空間と探索

状態の遷移（state transition）で表現される問題がある．例えば，パズルのなかには（図 3･40 に示した 9 パズルのように），有限個の選択肢のなかから次の動作を選ぶ（すなわち，次の状態を選ぶ）ことを繰り返して，解に至る手順を問うものがある．このような問題では，**始状態（initai state）**と**目標状態（goal state）**が与えられ，始状態から目標状態にいたる遷移が存在するか，存在する場合は最短の（ないし確実に目標状態に到達できる）手順はどのようなものか，を求めることになる．

このような問題では，状態を節点とし，遷移を枝とする木を考えることができる．始状態を根とし，各状態は，次の状態をすべて子とする．遷移可能な次の状態が存在しない状態は葉となる．このような木を**探索木（search tree）**と呼ぶ（図 3･40）．ある状態から異なる遷移を経て同じ状態に至ることがある場合には，木ではなく無閉路有向グラフを考えることもある．

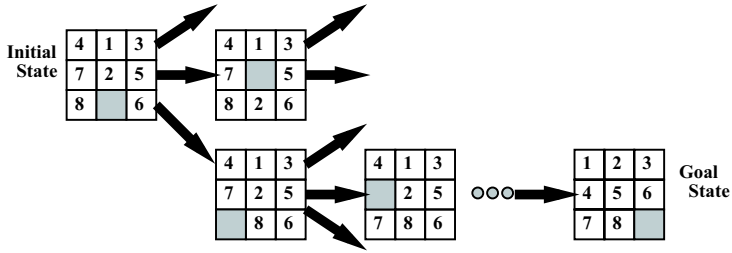


図 3･40　状態空間と探索木

状態の遷移で表現される問題は，原理的には，探索木の上で根から目標状態までの経路を求めることによって解くことができる．しかし，可能な状態の総数が多い場合，あらかじめ探索木をつくっておくことは現実的ではない．そこで，探索木のなかで既知の部分を徐々に広げながら，経路を求める方法が用いられる．このようなアルゴリズムは**状態空間探索（state space search）**と呼ばれる．

各状態に対して，それが目標状態であるか否かを調べることと，次の状態の集合を返すこと，及び二つの状態が等しいか否かを調べることができるものとする．ある状態 s から，次の状態の集合を得て，その要素を探索の候補に加えることを s の展開と呼ぶ．状態を展開する順序の違いによって，様々な探索のアルゴリズムが考えられる．

なお，状態を表すデータ構造に，その状態の前の状態を記録しておけば，目標状態が得られたときに，初期状態から目標状態までの経路を逆順に得ることができる．

3-5-2 深さ優先探索

状態 s から目標状態に至る遷移があるか否かを調べるには，s の次の状態として選べる状態のそれぞれについて，そこから始めて目標状態にいたる遷移があるか否かを調べればよい．もちろん，与えられた s が目標状態であれば探索を終える．

状態 s が目標状態でなく，状態 s の次の状態として $s_1, s_2, \cdots, s_n$ が選択肢として挙げられるとする．このとき，上の考え方で探索を行うとすると，s の次は s の次状態のうちの一つ，例えば s_1 から目標状態に到る遷移があるか否かを調べることになる．同様に s_1 が目標状態でなく，s_1 の次の状態として $s_{11}, s_{12}, \cdots, s_{1m}$ が選択肢として挙げられるとき，このうちの一つ，例えば s_{11} から目標状態に到る遷移があるか否かを調べることになる．このとき，s_2 よりも先に s_{11} を展開することになり，探索木のより深い（初期状態すなわち根ノードから遠い）ノードを先に展開することになる．このような探索を**深さ優先探索（depth first search, DFS）**または縦型探索と呼ぶ（図 3･41）．

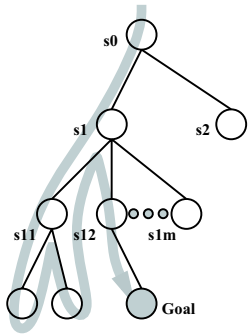


図 3･41　深さ優先探索

深さ優先探索では，既に訪れて展開した状態を再び展開しない（再訪抑制する）ことが必要となる．状態 s を展開しようとするとき，もし始状態から s（の一つ手前）までの経路のなかに s が含まれていれば，経路がループしていることになる．問題を解くには，始状態から目標状態に至る経路が存在するか否かを考えればよいので，ループを含まない経路のみを考えれば十分である．逆にループを排除しないと，同じ状態の展開を繰り返してしまい，目標状態までの経路が存在するにもかかわらず，アルゴリズムが停止しないことがある（図 3･42 左）．なお，始状態から現在の状態までの経路上にある状態だけでなく，これまでに展開したすべての状態を記録しておくことで，計算の無駄を減らすことができる．すなわち，状態 s から目標状態に至る経路がないことが分かっているときは（これが分かるのは既に s を展開して，s を経由する経路をすべて探索済みである場合である），s を展開する必要がない（図 3･42 右）．

図 3･43 は，深さ優先探索を行うプログラムを二つ示している．ここで，状態を表すクラスには，目標状態であるか否かを判定する `goal?` メソッド，次状態のリストを返す `nextStates` メソッド，及び，二つの状態が等しいか否かを判定するメソッドが適宜定義されているものとする．

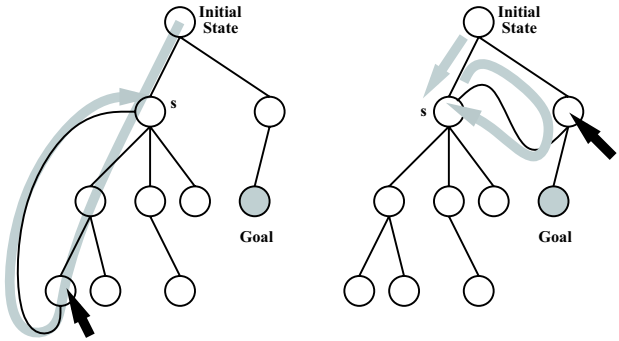


図 3･42　再訪抑制の必要性

図 3･43 上（DepthFirst1）は，再帰を用いて深さ優先探索を行うプログラムである．search の第 1 引数は現在訪れている状態，第 2 引数は初期状態からこれまでに訪れた状態のリストである．これに初期状態と空リストを与えて呼び出す．目標状態に至る遷移が存在すれば目標状態を返し，存在しなければ nil を返す．このプログラムでは，目標状態が見つかったとき，search の第 2 引数に目標状態（の一つ手前の状態）から始状態までの経路が格納されていることになる．

一方，未展開の状態をスタックで管理することで，深さ優先の探索が実現できる．すなわち，スタックの先頭から順に展開すべき状態を取り出し，状態を展開して得られる次状態をスタックに格納する．こうすることで，最近の展開によって得られた状態から順に展開される．図 3･43 下（DepthFirst2）は，スタックを用いて深さ優先探索を行うプログラムを示している．search の引数は初期状態である．ここで stack は，これから調べるべき状態を入れたスタックを表すリストであり，その初期値は初期状態だけを格納したリストである．またこちらのプログラムでは，ループだけでなく，これまでに訪れた状態すべてを記録して再訪抑制をしている．状態の記録にはハッシュ表を用いているため，状態を表すクラスにはハッシュ値を計算する hash メソッドを実装しておく必要がある．

```
class DepthFirst1
  def search(s,lst)
    if !lst.index(s).nil? then
      return nil
    end
    if s.goal? then
      return s
    end
    lst << s
    nxt = s.nextStates
    while( nxt.length > 0 )
      ns = nxt.pop
      a = search(ns,lst)
      if !a.nil? then
        return a
      end
    end
    return nil
  end
end

class DepthFirst2
  def initialize
    @ht = Hash.new(nil)
  end
  def search(ini)
    stack = [ini]
    while( stack.length > 0 )
      s = stack.shift
      next if !@ht[s].nil?
      if s.goal? then
        return s
      end
      @ht.store(s,true)
      stack = s.nextStates + stack
    end
    return nil
  end
end
```

図 3･43　深さ優先探索

3-5-3 幅優先探索

始状態から目標状態までの遷移の経路の長さ（手数）は，各状態における次状態の選び方に依存するが，深さ優先探索は，最短手数の解を返すとは限らない．一方，未展開の状態のうちで，始状態に近い状態から順に展開すれば，最初に見つかった目標状態までの経路が，最短手数であることが保証できる．このような探索を**幅優先探索（breadth first search, BFS）**または横型探索と呼ぶ（図 3･44）．

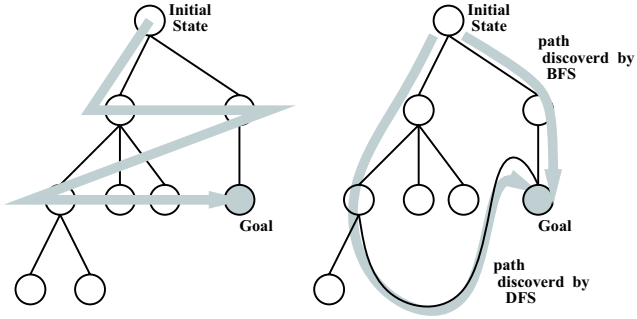


図 3･44　幅優先探索

始状態に近い状態ほど，展開によって生成された時期が古いと考えると，未展開の状態のうちで古いものから順に展開すれば，幅優先の探索が実現できる．そのために，未展開のノードをキューで管理する．すなわち，キューの先頭から次に展開すべき状態を取り出し，状態を展開して得られる次状態をすべてキューに格納する．こうすることで，古い状態から順に展開される．

幅優先探索では，解が存在するならば，原理的には再訪抑制をしなくても解を求めることができる．しかし，同じ状態がキューに格納されることを防ぐことで，メモリも実行時間も節約できるので，やはり再訪抑制をした方がよい．図 3･45 は，幅優先探索のプログラムである．未展開の状態をスタックに保存するかキューに保存するかの違いを除いて，図 3･43 右と同じプログラムとなっている．

3-5-4 最良優先探索

未展開の状態のうち，目標状態までのコスト（例えば遷移の回数）が最小のものを選んで展開すれば，最小コストの経路が得られる．しかし多くの場合，探索の途中で目標状態までのコストの実際の値を知ることはできない．そこで問題に依存した**発見的評価値（heuristics）**を用いて，未展開の状態から目標状態までのコストを見つもることで，深さ優先探索や幅優先探索よりも高速に質の良い解を得ようとする方法が考えられる．このような探索を**最良優先探索（best first search）**と呼ぶ．最良優先探索の性能（解の質や求解までの計算時間など）は，発見的評価値の性能に依存する．

```
class BreadthFirst
  def initialize
    @ht = Hash.new(nil)
  end
  def search(ini)
    queue = [ini]
    while( queue.length > 0 )
      s = queue.shift
      next if !@ht[s].nil?
      if s.goal? then
        return s
      end
      @ht.store(s,true)
      queue = queue + s.nextStates
    end
    return nil
  end
end
```

図 3･45　幅優先探索のコード

3-5-5 A*アルゴリズム

未展開の状態のうち，始状態からのコストと目標状態までのコストの合計が最小のものを選んで展開すれば，最小コストの経路が得られると期待できる．ここで，目標状態までのコストは，実際の値を探索中に知ることが難しいので，発見的評価値によって見つもる．このような探索アルゴリズムを **A アルゴリズム（A-algorithm）**と呼ぶ（図 3･46）．

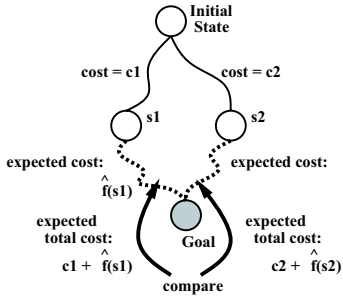


図 3･46　A アルゴリズム

最良優先探索との違いは，既知である始状態からのコストを用いることで，目標状態に至った段階で正確なコストが得られることにある．ある状態 s から目標状態までの真のコストを $f(s)$，発見的評価値を $\hat{f}(s)$ とする．もちろん $\hat{f} = f$ であれば，最初から最小コストの経路を

選ぶことができるが，そのような発見的評価値が求められると期待することは現実的ではない．$\hat{f}(s) > f(s)$ となる s が存在する場合には，実際は s を選ぶ経路のコストが小さいにもかかわらず，s が選ばれない可能性がある．このことから実際には最小コストである経路が，よりコストが大きいと見積もられることによって選ばれない可能性がある．一方，$\hat{f} \leq f$ である場合には，実際にはコストの高い経路につながる状態を選ぶ可能性がある．しかし，その経路をたどるうちに始状態からのコストが大きくなる．見積りコストは実際のコスト以下なので，より見積りコストが小さい経路を選んでいるうちに，実際に最小コストの経路が選ばれる．

このように，A アルゴリズムのうち，$\hat{f} \leq f$ を満たすものは，最小コストの経路を選ぶことが保証される．このようなアルゴリズムを **A*アルゴリズム（A*-algorithm）**と呼ぶ．

図 3･47 のプログラムでは，各状態について，目標状態までの距離の発見的評価を得る eval．初期状態からの距離を返す depth を仮定している．ここでは，queue を毎回ソートしているが，ヒープなどを用いた順序付きのキューを用いるとよい．

```
class A
  def initialize
    @ht = Hash.new(nil)
  end
  def search(ini)
    queue = [ini]
    while( queue.length > 0 )
      s = queue.shift
      next if !@ht[s].nil?
      if s.goal? then
        return s
      end
      @ht.store(s,true)
      queue = queue + s.nextStates
      queue.sort!{|x,y| x.eval+x.depth <=> y.eval+y.depth}
    end
    return nil
  end
end
```

図 3･47　A(または A*) アルゴリズムのコード

3-5-6 ゲーム木の探索

チェスや将棋などのように，二人で行うゲームを考えよう．ある局面で手番が回ってきたプレイヤは，有限な選択肢のうちから手を選ぶ．すなわち，次の局面を選ぶ．二人のプレイヤは交互に手を選び，一方の勝ち局面に至ったときにゲームが終了し，他方は負けとなる．どちらのプレイヤも，ある局面で相手が選ぶことのできる手を知ることができる．このようなゲームでは，初期局面や自分の番の局面を始状態として探索を行うことで，ゲームの先読みをすることができる．

対象とするゲームにおいて，局面の種類の総数が十分に小さければ，初期局面からゲームが終了する局面まで探索することができる．探索木と同様に，局面を節点とし，それぞれの局面が可能な次局面を子とするような木を考えることができる．葉はゲームが終了する局面であり，勝ち，負け，または引き分けの局面である．このようなゲームについての探索木を特にゲーム木と呼ぶ．

自分に与えられた局面を始状態，自分の勝ちとなる局面を目標状態として，勝ちに至る遷移があるか否かを調べることで，勝ち局面に至る手順があるか，及び，勝てるとしたらどの手を選べばよいかを知ることができる．この探索は状態空間探索に似ているが，相手が（積極的に）非協力的である点が異なる．すなわち，相手がどのような手を選んでも必ず勝つ手順を求めたい．このような探索をゲーム木探索と呼ぶ．

3-5-7 ミニ・マックス法

多くのゲームでは局面の総数が多く，勝敗が決するまでゲーム木を探索することは現実的ではない．そこで，何手か先までを読むことにし，その局面が有利か否かを発見的に評価する．評価値は局面が自分に有利な場合に大きな値を，不利な場合に小さな値をとるものとする．勝敗が決する場合の評価値は，極端に大きな値または極端に小さな値となる．

自分は（自分が）有利になるように，相手は（自分が）不利になるように手を選択するので，自分の手番では最も評価値が高い局面に至るように次の局面を選択し，相手の手番では最も評価値が低い局面に至るように次の局面を選択する．

このようなゲーム木探索のアルゴリズムを**ミニ・マックス法（minimax algorithm）**と呼ぶ．ミニ・マックス法では，初期局面からの距離を定めて，ゲーム木の一部を探索する．例えば，図 3･48 に示す例は，3 手先までの局面を表している．葉である局面に付された数値は，3 手先の局面の評価値を表している．葉以外のノードに付された数値は，子である局面の評価のうちから，自分の手番であれば最大値を，相手の手番であれば最小値を選んだものである．この図では相手の手番に弧を付している．なお，A,B,C,D,E は以下の説明のために局面に付けた名前である．

この例の場合，初期局面の評価値は 10 で，それは現在局面の三つの次の局面のうちから A を選ぶことによって得られる．相手が局面 A においてどの局面を選んだとしても，自分は 10 以上の評価値を得られるのである．このように，自分の手番では選択可能な評価値のうち

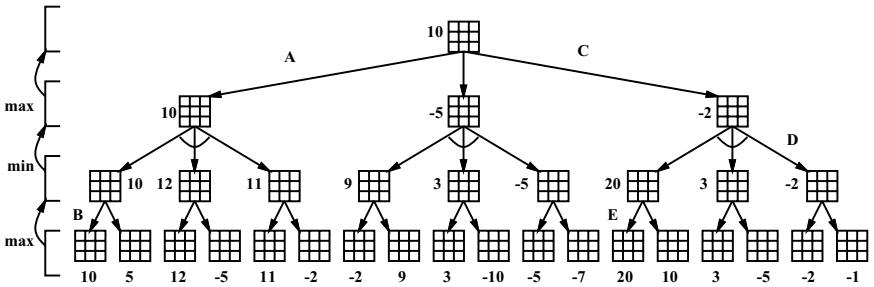


図 3･48　ゲーム木とミニ・マックス法で得られる局面の評価値

最大のもの（マックス）を選択できる．一方，3 手先の最も評価値の高い局面は E の 20 だが，E に至る局面として C を選んだとしても，相手は評価値が -2 となる局面を目指して D を選ぶだろう．すなわち相手の手番では可能なうちで最小のもの（ミニ）が選択されることを想定すべきである，このように，ミニ・マックス法では各局面がどちらの手番であるかに応じて，選択肢のうちで最大ないし最小の値をその局面の評価値とすることでゲーム木に評価を割り当てていく．

3-5-8 α-β 法

探索木のすべてのノードを調べなくても同じ（または似た）結果が得られる場合がある．不要なノードの探索をしないことを探索の枝刈り（カット）と呼ぶ．枝刈りの方法を考えることは，計算に要する時間や資源を節約するうえで重要である．

ミニ・マックス法では，最大値と最小値を交互に計算することから，次のような枝刈りの方法が知られている．まず，最小値を得るノードにおいて暫定最小値 β が得られているとき，ほかの枝がそれより大きい最小値をもつと分かった時点で，その枝の残りの部分は見なくてもよいと分かる．例えば，図 3･49 の左側の枝では暫定最小値「10」が得られているが，これはその下のノードから繰り上がってきたものである．次にほかの枝を調べにいくとき，一つ下のレベルでは複数の枝の最大値を取るので，例えば「11」や「12」が見つかったらこれらの枝からは先の「10」より小さい値は得られない．このため，「11」や「12」の兄弟に当たるノード以下は見なくてよい．これを β-カットと呼ぶ．

最大値を求めるノードでも同様に，暫定最大値 α が得られている状態で，ほかの枝がそれより小さい最大値をもつことが分かったら，その枝の残りの部分は見なくてよいと分かる．例えば，図 3･49 のトップレベルでは左側の枝から暫定最大値「10」が得られている．そこで，中央と右の枝において，最小値が「-5」や「-2」であると分かった時点で，残りの部分については見なくてよい．これを α-カットと呼ぶ．このような枝刈りを行うゲーム木探索のアルゴリズムを α-β **法（α-β algorithm）**と呼ぶ．

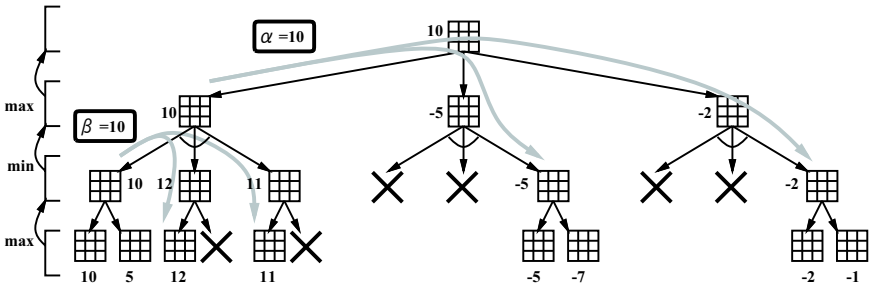


図 3･49　α-β 法による枝刈り

3-5-9 ゲーム木探索の様々な工夫

ここまで，ゲームの局面の探索順序を考えるに当たりゲーム木を考えたが，一般には局面間の関連は **DAG（Directed Acyclic Graph）**であり，同じ局面から互いに異なる複数の手を

通って同一の局面が得られる場合がある．そこで，局面をキー，評価値を値とするようなハッシュ表を用いるなどの方法で，同じ局面の評価をしないように工夫することが考えられる．

α-β 法では，ノードを調べる順序によって，カットされるノードの個数が変わる．カットされるノードを増やすには，自分の手番（max ノード）では評価値の大きな子ノードを先に，相手の手番（min ノード）では評価値の小さな子ノードを先に，それぞれ調べればよい．しかし，ノードの評価値は葉まで調べるまで分からないので，何らかの予測をする必要がある．そこで，読みの深さ（探索木の深さ）を増やしながら，α-β 法による探索を繰り返し，過去の（最大の深さが少ない）探索で得られた評価値（ハッシュ表などに記録されている）を，そのノードの評価値の予測値として用いることが考えられる．

別の方法として，深さの上限値を指定した深さ優先探索を行い，上限値を一つずつ増やしながら繰り返す方法があり，**反復深化（iterative deeping）**と呼ばれる．上限を一つずつ増やしているので，ゴールまでの経路が見つかったときには，その経路は最短であることが分かる．その一方で，基本的なアルゴリズムは深さ優先探索なので，比較的効率がよい．反復深化では探索を繰り返すことがオーバーヘッドとはなるが，同じ深さまでの探索にかかる計算時間は，単純な α-β 法よりも短くなることが多いといわれている．

■参考文献

1) 新谷虎松, “改訂 Java による知能プログラミング入門,” pp.22–44, コロナ社, 2002.
2) 菅原研次, “人工知能 [第 2 版],” 情報工学入門シリーズ 20, pp.13–55, 森北出版, 1997.
3) E. Rich 著, 廣田　薫, 宮村　勳 訳, “人工知能 I（原題: Artificial Intelligence）,” マグロウヒルブック, pp.31–38, 94–107, 137–152, 1984.
4) Stuart Russell, Peter Norvig 著, 古川康一 監訳, “エージェントアプローチ人工知能 第 2 版（原題: Artificial Intelligence: A Modern Approach 2 ed.）,” 共立出版, pp.59–79, 94–110, 162–172, 2008.

■6 群 - 3 編 - 3 章

3-6 コード情報のデータ構造とアルゴリズム

（執筆者：滝本宗宏）[2012 年 7 月 受領]

本節では，プログラムコードを表現するデータ構造とその作成法，及びその応用について述べる．まず，制御の流れを表現した**制御フローグラフ（control flow graph）**[1]について述べ，通常プログラムの制御フローグラフへの変換法と，制御フローグラフから通常プログラムへの逆変換法を示す．次に，制御フローグラフの節点どうしの関係の例として，**支配（dominace）**[1]を挙げ，支配関係を木構造で表現した**支配木（dominator tree）**[1]に触れる．最後に，制御フローグラフと支配木を用いた解析の例として，ループを検出する方法を説明し，そのループ内の不変コードをループ外へ移動するコード最適化を紹介する．

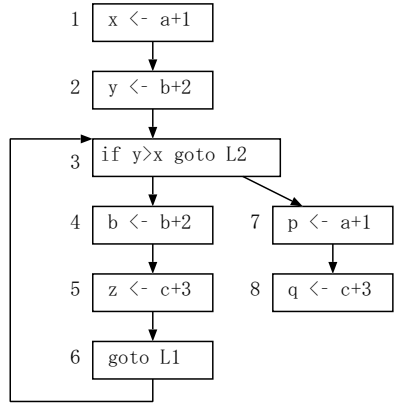
3-6-1 制御フローグラフ

プログラムの性質を調べる際には，プログラムがどのような順番で実行されるかという制御の流れの情報が重要である．この制御の流れの情報は，**制御フロー（control flow）**[1]と呼ばれる．

例えば，図 3･50(a) のプログラムを考えよう．ここで，各行を**文（statement）**と呼ぶことにする．図 3･50(a) のようなプログラムでは，`goto` *Li* のような goto 文と，*Li* : のような飛び先ラベルが各所に現れ，制御フローが見にくいので，制御フローを明示したグラフ表現がよく用いられる．各文を**節点**（node）として，s_1，s_2 の順序で続けて実行される文について，それぞれの節点を**辺**（edge）で繋ぐ．このようにして得られるグラフ表現は，制御フローグラフ（control flow graph）と呼ばれる．以降，制御フローグラフは，CFG と呼ぶことにする．

```
1:          x ← a+1
2:          y ← b+2
3: L1:      if y > x goto L2
4:          b ← b + 2
5:          z ← c + 3
6:          goto L1
7: L2:      p ← a + 1
8:          q ← c+3
```

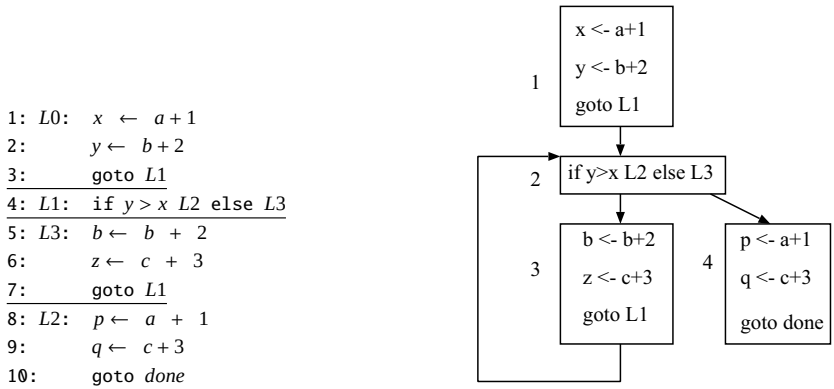
(a) **プログラム**



(b) **制御フローグラフ**

図 3･50　プログラムと制御フローグラフ

図 3･50(b) は，図 3･50(a) の CFG を表している．CFG の各節点の左には，対応する文の行番号を記述してある．節点 1 と節点 8 は，それぞれプログラムの開始と終了を意味しているので，特に，**開始節（start node）**と**終了節（end node）**と呼ばれる[1]．節点 3 のような 2 本の辺が出ている if 文の分岐は，真になる場合と，偽になる場合とで，どちらに分岐すればよいか示されていない．一般に，if 文の条件の真偽は，実行してみないと分からないので，CFG を実行前の**静的**（static）な解析に用いる場合には，これで十分である．すなわち，CFG は，プログラムに代わる表現ではなく，プログラムの性質を解析するために用いる，抽象化された表現である．



(a) **分割した基本ブロック**　　(b) **基本ブロックを節点とする** CFG

図 3･51　基本ブロック

（1）基本ブロック

CFG を，制御フローという側面だけから見ると，更に抽象化を進めることができる．制御フローで，プログラムの性質を調べる際に重要になるのは，分岐と，飛び先で現れる**結合点（join point）**[1] である．この 2 点だけに着目すると，個々の文の操作や，分岐をせずに直線的に実行される文の列は，無視することができる．直線的に実行される文の列を一まとめにしたものは，**基本ブロック（basic block）**[1]と呼ばれる．図 3･51(a) は，図 3･50(a) を基本ブロックに分割した結果を示している．CFG の節点として，基本ブロックを採用すると，節点の数が減るので，解析効率を向上させられる場合がある．基本ブロックは，もとのプログラムを次の手順で分割することによって生成する．

1. ラベルが見つかると，新しい基本ブロックを開始する（以前のブロックは終了）．
2. goto あるいは if 文が見つかると，現在のブロックを終了し，新しいブロックを開始する．
3. goto あるいは if 文なしでブロックを終了する場合は，次のブロックのラベルへの goto 文を，最後に付加する．
4. ブロックのはじめにラベルがなければ，新しいラベルを付加する．

上記の手順で生成した基本ブロックは，次の基準を満たすことが分かる．

1. 最初の文には，ラベルが付いている．
2. 最後の文は，if 文であるか goto 文である．
3. その他の場所に，ラベル，if 文，goto 文のいずれも存在しない．

基本ブロックのこの性質を利用すると，各基本ブロックに対して，節点を生成し，各節点と，その節点の最後の if 文あるいは goto 文の飛び先レベルをもつ節点との間に辺を生成することによって，より粗い CFG を作成することができる．図 3･51(b) に，(a) の基本ブロックを節点にした CFG を示す．

3-6-2 制御フローグラフの逆変換

CFG 上で，解析やプログラム変形を行ったあとは，通常のプログラムに変換しなければならない．その際，まず連続して実行される可能性がある CFG 節の並びを作成する．この並びは，**トレース（trace）**と呼ばれる．適切にトレースを作成すると，無用な goto 文を減らすことができる．最終的に，すべてのトレースを一列に並べると，全体のプログラムを復活できる．

（1）トレース

上記で述べた基本ブロックを節点とする CFG を対象とすると，トレースは，次の手順で生成できる．

1. いずれかの基本ブロックを選ぶ．訪問済みブロックには印をつける．
2. 訪問中の基本ブロックをトレースに加える．
3. 訪問中の基本ブロックの最後が，if 文なら，条件が偽のときの飛び先ラベル，goto 文なら，その飛び先ラベルをもつ基本ブロックを探して，訪問する．
4. 印が付いたブロックを訪問したときは，トレースを終了させ，次のトレースを開始するために，印のついていない基本ブロックを選ぶ．

例えば，図 3･51(b) のトレースを計算すると，節点 1 から始めて，{1，2，3} と {4} のトレースが得られる．

（2）最終処理

最終的に，トレースを一列に並べて，一つのプログラムにする．この際，次の手順に従って，各基本ブロックの分岐を調整する必要がある．

- goto 文の直後に，その飛び先ラベルがきている場合は，goto 文を除去する．
- if 文の条件が真のときの飛び先が，直後にきている場合は，if 文の真の飛び先ラベルと偽の飛び先ラベルを交換し，条件を反転させる．
- if 文 `if` *cond* `goto` *L*1 `else` *L*2 の直後に，真の飛び先ラベルも，偽の飛び先ラベルも

続かない場合は，新しい偽の飛び先ラベル *L2′* を用意し，*L2′* が直後に続くように if 文を，次の二つの文に書き換える．

```
1:      if cond L1 else L2'
2: L2': goto  L2
```

(3) 最適トレース

基本的に，トレース数が少なくなるようにトレースを構成すると，goto 文の数が減って，実行効率の良いコードが得られる．しかしながら，この直観が良い結果を生じない場合がある．

例えば，前節の例で示した図 3･51 のトレースをコードで示すと，図 3･52(a) のようになる．ここで，横線はトレースの境界を示しており，トレース数が 2 であることを示している．一方，図 3･52(b) のようにトレースを構成すると，トレース数は 3 になる．このとき，除去できる goto 文の数は同じであるが，ループ内だけに限って見ると，(b) の方が (a) より goto 文の数が少なくなる．一般的に，ループの本体は，複数回実行されるので，(b) のコードの実行効率が高くなると考えられる．

このように，多くの場合，トレース数を少なくするようにトレースを構成する方が有利であるが，かならずしも，実行効率を最適にするとは限らないことを，理解しておかなければならない．

```
1: L0:  x ← a+1
2:      y ← b+2
3:      goto L1
4: L1:  if y > x L2 else L3
5: L3:  b ← b + 2
6:      z ← c + 3
7:      goto L1
8: L2:  p ← a + 1
9:      q ← c+3
10:     goto done
```

(a) **最少トレース**

```
1: L0:  x ← a+1
2:      y ← b+2
3:      goto L1
5: L3:  b ← b + 2
6:      z ← c + 3
7:      goto L1
4: L1:  if y > x L2 else L3
8: L2:  p ← a + 1
9:      q ← c+3
10:     goto done
```

(b) **最適トレース**

図 3･52　トレースの選択

3-6-3 支配木

前節では，プログラムの制御の流れを表現するデータ構造として，制御フローグラフを紹介した．この制御フローについて，そのフローの仕方を，特定野性質に着目して表現したデータ構造が，多く提案されている．これらのデータ構造を用いると，コード最適化やコード並列化を効率よく行えるようになる．

この節では，最も応用範囲の広い，**支配木（dominator tree）**を紹介する．支配木を利用すると，あるプログラム点に制御が移るまでに，必ず実行されるプログラム点を知ることができる．まず，**支配節（dominator）**について定義を与え，支配木の簡単な構成法を示す．支配木の応用については，次の節で触れることにする．

(1) 支配節

CFG 上で，開始節 **s** から節点 *n* に到るすべての実行経路が，節点 *d* を通過するならば，*d* は *n* を**支配（dominate）**[1] するという．ここで，すべての節点は，自分自身を支配する．

s から *n* に到達するためには，必ず *n* の先行節を通過しなければならないことは明らかなので，*d* が *n* を支配するなら，*d* の先行節も支配することになる．この事実から，*n* の支配節の集合 *Dom*(*n*) は，次の方程式を満たすことが分かる．

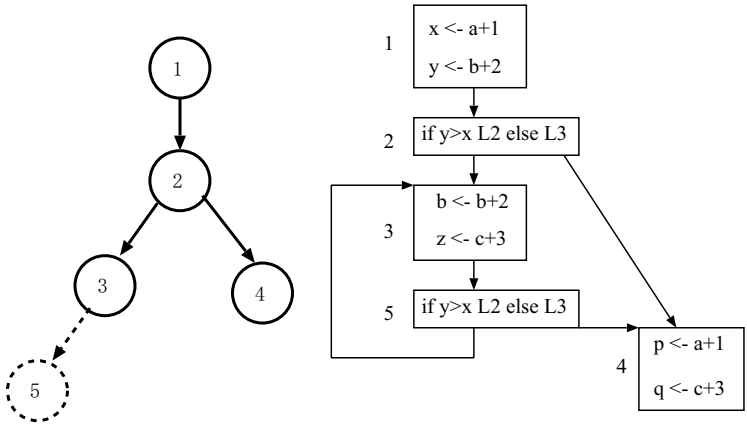
$$Dom(n) = \begin{cases} \text{もし，} n = \mathbf{s} \text{ なら，} & \{\mathbf{s}\} \\ \text{さもなければ，} & \{n\} \cup (\bigcap_{p \in pred(n)} Dom(p)) \end{cases} \quad (3\cdot1)$$

方程式 3･1 の解は，**s** 以外の *n* について *Dom*(*n*) をすべての節点で初期化し，= を代入操作に読みかえてすべての *Dom*(*n*) が変化しなくなるまで繰り返し計算すると得られる．

(2) 直接支配節と支配木

節点 *d* と *e* が，節点 *n* を支配するとき，*d* が *e* を支配するか，*e* が *d* を支配するかのいずれかであることが証明できる．すなわち，*n* は，ほかのすべての *n* の支配節に支配される支配節 *d′* をもつことが分かる．この *d′* は，**直接支配節（immediate dominator）**[1] と呼ばれる．

各節点に，その直接支配節から辺を引くと，開始節を根とする木構造が得られる．この木は，**支配木（dominator tree）**[1] と呼ばれる．例えば，図 3･51(b) の支配木は，図 3･53(a) の実線の図のようになる．支配木は，その祖先の節点が子孫の節点を支配することを表している．



（a）支配木　　（b）while**ループの変形**

図 3･53　支配木とループ変形

3-6-4 制御フローグラフと支配木の応用

CFG と支配木を用いると，プログラムの制御フロー上の性質を調べたり，その性質を利用したコード最適化を容易に実現したりできる．この節では，プログラム中のループの検出と，ループ不変コードのループ外への移動の例を示す．

（1）ループ検出

構造化されたプログラムの CFG 上で，辺 $n \to h$ のうち節点 h が節点 n を支配するものは，**戻辺**（back edge）[2] と呼ばれる．戻辺が見つかると，ループは，h によって支配され，h を通過せずに n に到達できる節点の集合として定義できる．例えば，図 3･51(b) において，戻辺 $3 \to 2$ が見つかる．節点 2 から辺を逆向きにたどって，節点 3 に到達できるまでに訪問する節点集合 {2,3} がループとして検出できる．

（2）ループ不変コード

ループ中で計算結果や状態が変わらない命令は，**ループ不変コード（loop invariant code）**[1] と呼ばれる．ループ不変コードは，ループの外に移動することによって，プログラムの実行効率を向上させることができる．

ループ不変コードをループの外へ移動させるためには，対象を式 t と代入先変数 x からなるループ中の代入文 $x \leftarrow t$ と仮定すると，次の 2 条件を満たす必要がある．

1. x のような代入先や，式 t に含まれるオペランドがループ中で変更されない．
2. 代入文の CFG 節点が，すべてのループ出口を支配する．

例えば，図 3･51(b) で，ループ中で変更される変数をもたない代入文は，節点 3 の $z \leftarrow c + 3$ だけであるが，節点 3 はループの出口節点 2 を支配しないので，図 3･51(b) には，ループの外に移動できるループ不変コードが存在しないことになる．

一方，図 3･51(b) のような while ループを，図 3･53(b) のような同じ条件を二つもつ if 文と do-while 文の組合せに変形すると，節点 3 は，図 3･53(a) の点線部分で示すようにループの出口節点 5 を支配するようになる．ここで，簡略化するために，goto 文は省略してある．

この変形した CFG 上では，$z \leftarrow c + 3$ を節点 3 の直前に移動することができる．

■参考文献

1) Andrew W. Apple 著, 神林　靖, 滝本宗宏 訳, “最新コンパイラ構成技法,” 翔泳社, 2009.
2) Alfred V. Aho, Jeffery D. Ullman, Ravi Sethi, Monica S. Lam 著，原田賢一 訳, “コンパイラ-原理・技法・ツール 第 2 版,” サイエンス社, 2009.