計算機ソフトウェア

目次

第 1 章

データ構造

本章はデータ構造を扱う．

1.1　配列

配列とは計算機内で最も基本的なデータ構造だ．データを格納する容器のようなもの自体に番号が振ってあるもの．このようなデータ構造において新たなデータを今までの配列の間に割り込む場合，容器そのものに番号があるので，割り込む場所以降のすべてのデータを移動させる必要がある．データ格納の容器自体のサイズによっては大掛かりな処理になってしまい，無駄が多いといえる．

1.2　リスト構造

リスト構造は配列の難点を解消するデータ構造だ．ポインタの発想を使いデータを管理している．リスト構造で管理されるデータを表としてあらわしてみる (表 1.1).

表 1.1　リスト構造の例

ID	data1	data2	next
1	A		2
2	B		5
3	C		9
⋮	⋮	⋮	⋮
n	Ω		-

このデータ構造では next とある部分に次のデータの ID(ポインタ) が格納されている．そのためデータを格納する容器のサイズによらず，データの並び替えは next の内容を変更することでなされる．

そのため，例えば i 番目の次に新たなデータを割り込むように格納する手順は次のようになる．

1. データの格納する容器を作成し，ID はデータ数 n に 1 を加えたもの，next は空白としてデータを格納する．
2. next[$n+1$] に対して next[i] を代入する．
3. next[i] の値を $n+1$ とする．

視覚的に表すと図 1.1 のようになる．データの格納する容器をセルと呼ぶ．セルはいちいち作るのではなく，未使用のものが用意されている．間に割り込ませるという動きをそのまま実現していることが図 1.1 からわかる．

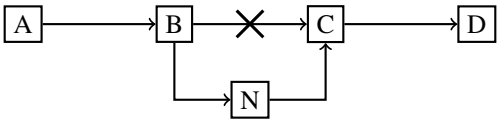

図 1.1　リスト構造での挿入

次の例はデータの削除だ．ID が i であるデータを削除する．これはもっと容易だ．

1. i を指している next[j] を持つ j を探す．
2. next[j] に next[i] を代入する．

これは不要データの読み飛ばしを実現しているのだ．j を探すのには全データの確認が必要となってしまう．そこで，次のデータのポインタである next のほかに前のデータのポインタである prev を持たせるデータ構造もある．

さて，読み飛ばしによるデータの削除では，読み飛ばされ意味を持たないのに使い道のないゴミと呼ばれるセルが発生してしまう．図 1.2 のように不要となったセルをクリアしたのち未使用セルの先頭に回しておくという方法がある．

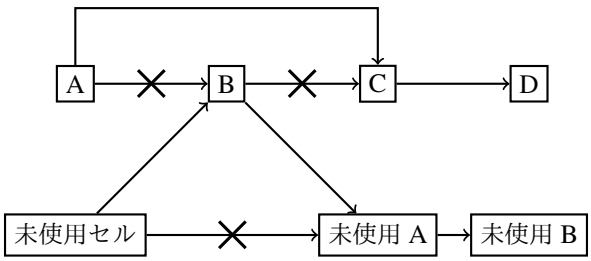

図 1.2　リスト構造での挿入

1.3　根付き木

1.3.1　導入

データ構造として根付き木というものを新たに導入する．図 1.3 のようなものだ．トーナメント表のようなものだが，下に来る個数に指定はない．

頂点としてる部分がデータの格納されている部分だ．そして各データは他の一部データと辺によって結ばれている．根と呼ばれる頂点を 1 つ定めると，他の頂点には図 1.3 のように親子関係，兄弟関係が定義できる．

親子関係とは 1 つの辺を共有している頂点の組の関係を表している．そして，根に近い頂点を親，他方を子という．また，同じ親を持つ頂点同士の関係を兄弟関係という．

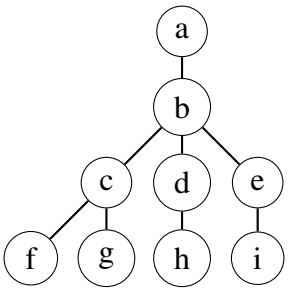

図 1.3　根付き木

1.3.2　頂点の指示

頂点はデータを格納する部分だが，残したいデータとともに，親や子といった関係のある頂点へのポインタを格納しておくことで根付き木としてのデータ構造を形成する．しかし，ここで気になるのは，子の数に指定のない根付き木の場合，頂点にはポインタ格納用の領域をどれほど確保する必要があるのかという問題だ．

これに対する答えは 3 つだ．格納すべきポインタは頂点の親へのポインタ，次の兄弟へのポインタ，自らの第一の子へのポインタだ．これによってすべての頂点を次の兄弟へ進む回数や第 1 の子へ進む回数として表現することができる．

例えばある頂点 v の子をすべて列挙することを考える．そのアルゴリズムは次のようになる．

```
x = FirstChild[v]
do until x = NULL
    report x, x = NextBrother[x]
```

他にも頂点 v の第 1 の子に新たな頂点 w を追加するアルゴリズムは次のようになる．

```
Perent[w] = v
NextChild[w] = FirstChild[v]
FirstChild[v] = w
```

順序が大事で，間違えるとうまく頂点の更新ができない．

第 2 章

整列

順序関係が定義できるデータをその順番で並べることを整列 (ソート) という．

2.1 最も単純なソート (バカソート)

バブルソートではない．アルゴリズムを次に示す．

```
for i = 0 to n-2
    for j = i+1 to n-1
        if x[i] > x[j] then
            swap(x[i], x[j])
return x
```

このアルゴリズムの計算量は $O(n^2)$ となっていて，かなり悪いアルゴリズムとなっている．最も効率の良いソートの計算量は $O(n \log n)$ である．そこで計算量が $O(n^2)$ のソートは効率の悪いソートアルゴリズム，$O(n \log n)$ のソートは効率の良いソートアルゴリズムであるという視点でこれからのアルゴリズムを見てゆく．

2.2 バブルソート

聞いたことがある人もいるかもしれない．プログラムを勉強するとかなり早い段階で書こうとするソートの 1 つではないだろうか．以下にアルゴリズムを示す．

```
for i = 0 to n-1
    for j = 0 to n-i-2
        if x[j] > x[j+1]
            swap(x[j], x[j+1])
return x
```

ほとんどバカソートと同じアルゴリズムに見えるが，2 段目のループの繰り返し回数が確実に減るため同じ計算量 $O(n^2)$ ではあるが効率がいいほうであるといえる．しかし，これも良いアルゴリズムだというには物足り

ないソートとなっている．

アルゴリズムを読むと，1 段目のループを一回実行すると最も大きな値が必ず最後尾に回り，次のループでは最後を無視したデータ列で同じことを繰り返す．ソートの完了していないデータの集合から必ず最大値を拾うことでソートを実行している．

2.3 挿入法

挿入ソートと呼ばれるアルゴリズムだ．早速アルゴリズムを示す．

```
for i = 1 to n-1
    y = x[i];
    while(a[j]>y && j>=0){
        a[j+1] = a[j];
        j--;
    }
    a[j+1] = y;
return x
```

平均した計算量は $O(n^2)$ となるが，これはバブルソートよりさらに効率のいいソートとなっている．また，すでにある程度の整列されているデータに対しては $O(n)$ ほどの計算量でソートを完了することもある．

アルゴリズムの感覚的な理解としては，データ列の先頭から順番にソートが完了しているデータ列を伸ばしてゆくものだ．

2.4 クイックソート

2.4.1 一般的な理解

クイックソートは計算量 $O(n \log n)$ を実現するソートの一つだ．その発想は次の通りだ．

1. 配列データの中から中央[*1]にある値を基準値とする．
2. 基準値に対して，それ以下のものを基準値より添字を小さく，以上のものを基準値より添字を大きくする．視覚的に言うと，基準値の左右により分ける．
3. 基準値に対するより分けが終わると，基準値を境界に配列データを 2 つの配列データに分ける．このとき，基準値はどちらにも属さない．
4. 分けたデータ配列に対してこれまでの操作を繰り返す．

以下に c 言語によるクイックソートの実現例を示す．

```
void quicksort(int a[], int first, int last){
    int i, j;
```

[*1] 配列の添字で数えて中央と考えている．

```
    int x, t;
    x = a[(first+last)/2];
    i=first;    j=last;
    while(ture){
        while(a[i] < x) i++;
        while(a[j] > x) j--;
        if(i >= j)  break;
        t = a[i];
        a[i] = a[j];
        a[j] = t;
        i++;    j--;
    }
    if(first < i-1) quicksort(a, first, i-1);
    if(j+1 < last)  quicksort(a, j+1, last);
}
```

基準値に対してその他のデータを左右により分ける作業を入れ替えによって実現している．i,j によって配列を両端から走査するが，i,j の示す領域の外側はより分けが完了するようになってる．

2.4.2　2 進木・2 分木による理解

根付き木のうち子の数が最大で 2 つとなるようにしたものを 2 進木，2 分木という．これを用いてクイックソートを理解することができる．例として次の配列データのソートを 2 分木によって実現する．

$$\{8, 5, 7, 3, 10, 9, 6, 1, 20\}$$

この配列に対して次のように 2 分木を作成する．

1. 配列の先頭から値を確認する．
2. 先頭の場合はそのまま根する．
3. 2 つ目以降の要素に対しては既存の 2 分木の根からその頂点との値の比較をおこなう．
4. 値を比較し，頂点より値が小さいときは左側の辺に，頂点より値が大きいときは右の辺に進む．
5. 辺を進み頂点が現れたら，その頂点との値の比較をおこない同様に進む．
6. 進む辺がない場合，新たな辺を作ったうえでその要素を頂点として追加する．

上の手順で作成した 2 分木は図 2.1 の通りだ．

これは子を持つ頂点のどれに注目しても，その頂点を基準値として左右に要素をより分けた状態となっている．2 分木が完成したら次は値の読み出しだ．次の手順で実行される．

1. 読み出しの基本はある頂点を親としてとらえ，それに対して左側の子，自分，右側の子の順でおこなう．
2. 読みだそうした頂点に子がある場合，その頂点を親として 1 と同様の操作をおこなう．
3. 上の操作を繰りかえし，葉に到達したとき，その葉の値を読み出したうえで，親に戻り読み出しを続

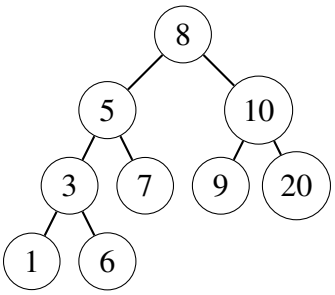

図 2.1　クイックソートの理解

ける．

この操作を繰り返すことで例の配列の昇順ソートが得られる．

2.5　ヒープソート

2.5.1　ヒープとは

ヒープは 2 分木の一種で以下の条件を満たすものだ．

- 高さ k のヒープは，高さ $k-1$ までの頂点をすべて使い，高さ k では頂点は左から使われている．
- 親の値は子の値以上である．

この条件を使いクイックソートの例で挙げた配列をヒープにする．

$$\{8, 5, 7, 3, 10, 9, 6, 1, 20\}$$

実際の手順としては 1 つ目の条件を満たすように単調に頂点を並べてゆく．すると，10 を頂点として追加したときに 2 つ目の条件を満たさないことがわかる (図 2.2).

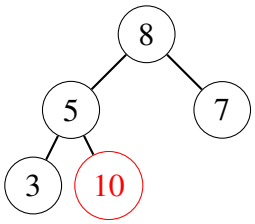

図 2.2　ヒープ作成過程 1

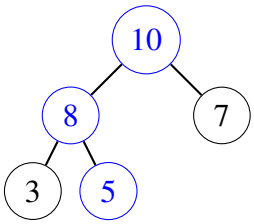

図 2.3　ヒープ作成過程 2

2 つ目の条件を満たさない場合にのみ，次の操作をおこなう．

1. 2 つ目の条件を満たさない頂点 v と，親の入れ替えをおこなう．
2. 入れ替えをした後，v と新たな親との大小比較をおこなう．
3. 条件を満たさない場合は再度親と入れ替える．
4. 条件の満たすまで v とその親との入れ替えを続ける．

要約すると，条件を満たすようになるで頂点を根に近づけてくのだ．すると図 2.2 の状態は図 2.3 の状態へ変

わり，ヒープとなる．

1 つ目の条件を満たすように頂点を加えながら，親との入れ替えなどを繰り返すと次の 2 分木が完成する．

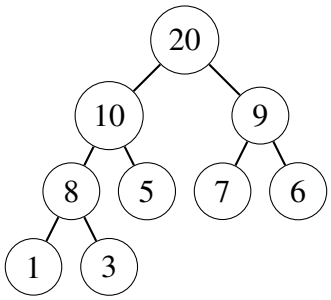

図 2.4　クイックソートの理解

これでヒープが完成する．

2.5.2　値の取り出し

ヒープを作成する部分は理解できた，次は値を昇順ソートとなるように値を取り出す．この手順が面白い．ヒープの性質として，2 分木の根がデータの最大値であることを利用する．

1. ヒープの根の値を最大値として取り出し，ヒープの最も高く[*2]最も右となる葉と交換する．そして，頂点から交換された頂点は以降の操作から無視する[*3]．
2. 新たに根に移動した頂点 v は 2 つ目の条件を満たさない．そこで，v と左の子，右の子で大小比較をおこなう．
3. 大小比較の結果，子の方が大きいとわかった場合，そのうちでも左右の子の大きいほうと v を交換する．
4. v が移動できなくなるまで 2，3 の操作を続ける．
5. 操作が完了すると，2 分木は新たにヒープを形成しているので 1 に戻る．

ヒープの根を取り出し，ヒープを作り直すという操作を繰り返すのだ．

2.6　2 分木の有用性

2 分木等のはプログラム作成上，とても扱いやすい構造であるといえる．これは 2 分木に対して図 2.5 のように番号を振ることで見えてくる．

親と子に振られた番号を見ると，だいたい 2 倍の関係が見える．親から左の子へ進むときは 2 倍して 1 を足す．左側の子に進むには 2 倍して 2 を足す．子から親へ進むときは 1 を引いて 2 で割ればいい[*4]．

一定の計算によって親子の移動が実現できるのが有用な点だ．これによってヒープソートも簡潔に記述できる．

[*2] ヒープにおいて高いとは 2 分木で見ると最も下という意味である．

[*3] 2 分木の中で最も大きな値がヒープの最も端に移動され無視される．すると残りの 2 分木は形状としてヒープを維持する．

[*4] プログラムでは商は整数となることを利用している．

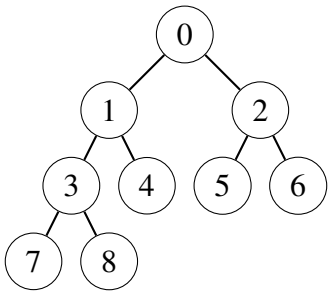

図 2.5　番号を振ったヒープ

2.6.1　c 言語による実現

効率的なヒープの作成はヒープを下から作成することが必要だ．以下にヒープソートの実現例を示す．

```
void heapsort(int n,int a[]){
    int i,j;
    for(i= (n-2)/2;i>=0;i--){
        int k = i;
        int v = a[k];
        while(1){
            j=2*k +1;
            if(j>n-1)
                break;
            else if(j != n-1){
                if(a[j+1]>a[j])
                    j++;
            }
            if(v>=a[j])
                break;
            a[k]=a[j];
            k=j;
        }
        a[k]=v;
    }

    while(n>0){
        int x = a[n-1];
        a[n-1]=a[0];  n--;
        i=0;
        while((j=2*i+1)<=n-1){
```

```
            if(j<n-1 && a[j]<a[j+1])
                j++;
            if(a[j]<x)
                break;
            a[i]=a[j];
            i=j;
        }
        a[i]=x;
    }
}
```

前半の for 文がヒープの作成，後半の while 文が値の取り出しとなっている．ヒープの作成で効率化をすることで，同じ計算量 $O(n \log n)$ でも効率のいいほうであるといえる．また，ヒープの作成の後，並び替えを実行するのではなく，大きいものから値をいくつかとるなどの応用ができる．

第 3 章

ハッシュ

データの集合から任意のデータを取り出すことを考える．どのように探せば早くデータを取り出せるのか，どのようにデータを整理しておけば早くデータを探し出せるかということを扱う．

3.1　探査・検索

前提としてこの章で考えるデータの状態を考える．データは key と呼ばれる固有の値と，record と呼ぶデータの 1 組を単位として保存されているとする (図 3.1).

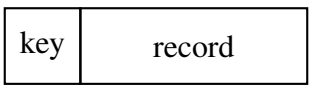

図 3.1　ハッシュを考えるときのデータ単位

データがたくさんあり，所望のデータの格納先を探すとき，単純に思いつくのは線形探索だ．単純にデータが格納されている部分を先頭から参照して，探している内容と一致するのかを判定するというものだ．計算回数はデータの個数に比例するので $O(n)$ ということになる．

格納されているデータが整列されているという条件があれば二分探査と呼ばれる方法でデータを探すことが

できる．これはちょうどデータの列の中央にあるデータを参照し，そのデータに対して順序として前の方にあるのか後ろの方にあるのかを確認することで毎回操作する範囲を半分にしてゆく方法だ．これであれば計算量は $O(\log n)$ となる．

3.2 ハッシュ法

線形探査や 2 分探査というのは，データが配列として漠然と格納されていることを想定している．せいぜい整列がなされている状態である．しかし，格納する先そのものを何らかの規則によって決定して入れば，1 度の計算で格納先を決定できる．

このような発想を実現するのがハッシュ法だ．具体的には格納するデータ自体から格納先となる key を生成することで，データと格納先を対応させる．

3.2.1 ハッシュ表

ハッシュ表とは図 3.1 のデータ単位をたくさんまとめたものだ．そこには key が 0 から番号が振ってある．最初は record の部分は空白[*1]が格納されている．

ハッシュ表を埋める作業がハッシュ法の核をなす部分だ．方法は保存したいデータ x に対して関数 h を作用させることで key の範囲に入る数 $h(x)$ を獲得する．そしてハッシュ表の key が $h(x)$ である部分に record として x を格納する．

これを繰り返すことで，ハッシュ表は一部がデータが格納され，一部は空白のままという状態になる．これでデータは格納されたことになる．データを探査するときは探したいデータ y から $h(y)$ を得ることで，key が $h(y)$ である部分の record を参照すればよいことになる．計算量は $O(1)$ となる．

3.2.2 ハッシュ関数と衝突

ハッシュ表を埋めるとき，格納したいデータから key を算出した関数 h をハッシュ関数という．データから整数を返す関数だ．

このハッシュ関数というのは，例えば割り算のあまりなど，データから key の範囲に収まる整数を得られれば良い関数である．そのため，時として異なるデータ x, y に対して同じ関数値 $h(x) = h(y)$ を与える場合がある．これを衝突と呼ぶ．ハッシュ表において 1 つの key に対して record は 1 つしか格納できないので，あとから来たデータに対しては何らかの方法で別の格納先を用意する必要がある．

衝突の解決法の 1 つがチェイン法だ．データの単位にポインタを加えることで解決する (図 3.2).

図 3.2　ハッシュを考えるときのデータ単位

これによって衝突を図 3.3 のように解決することができる．同じ key に数珠つなぎでデータを格納する．探査する場合は key の決定の後，key の中のデータ群に対しては線形走査をおこなう．同じ key にデータが集中

[*1] 空白を意味する何か.

すると効率は落ちるようになる．

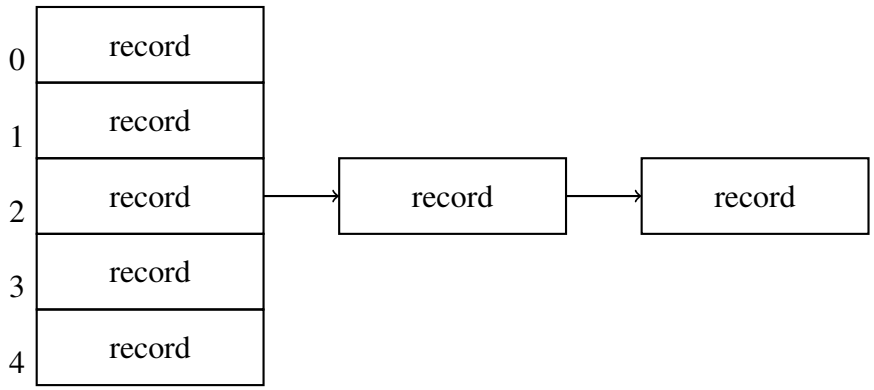

図 3.3　チェイン法

チェイン法はもともと少ない key の数に対してポインタによって格納の領域を増やすというイメージで衝突を回避している．チェイン法とは異なり，十分に大きなハッシュ表を用意できている場合はデータ値からハッシュ関数によって計算される key の値そのものを規則に従い変更することで衝突を回避する方法がある．これを開番地法という．

開番地法の発想は複数のハッシュ関数を用意することで衝突が発生したときにハッシュ値を計算しなおすことにある．しかし，この複数のハッシュ関数というのはでたらめにつくるわけではない．いくつかの手法が存在する．

線形走査法　基本となるハッシュ関数 $h_1(x)$ を

$$h_1(x) = h(x)$$

としておく．そして衝突が発生したときに衝突が解消されるまで次のようなハッシュ関数を使う．

$$\begin{aligned} h_2 &= h(x) + 1 \\ h_3 &= h(x) + 2 \\ &\vdots \end{aligned}$$

ハッシュ表から眺めると，もしハッシュ値がかぶりデータの格納ができない場合，1 つ下の key を確認し空白であれば格納．そうでなければさらに下を確認という操作を続けることになる．

平方捜査法　ハッシュ関数やデータの集合によっては，ハッシュ値が似通ってしまい，線形走査ではいつまでたってもデータを格納できないという状況[*2]がまれに存在する．これを解消するために 1 つ下ばかりを見るのではなく，平方数を用いて飛び飛びに確認することで効率を向上させるという手法がある．

[*2] ハッシュ表に生じる連続した key にデータが格納されている部分をクラスタと呼び，クラスタが生まれ効率が落ちる現象をクラスタリング現象という．

$$
\begin{aligned}
h_2 &= h(x) + 1^2 \\
h_3 &= h(x) + 2^2 \\
&\vdots
\end{aligned}
$$

2 重ハッシュ法　基本となるハッシュ関数 $h(x)$ のほかに，衝突解消用のハッシュ関数 $g(x)$ を用意する．この 2 つのハッシュ関数を使うことで

$$
\begin{aligned}
h_1 &= h(x) \\
h_2 &= h(x) + g(x) \\
h_3 &= h(x) + 2g(x) \\
&\vdots
\end{aligned}
$$

とする．これで 1 つのデータの格納に対して，複数回の衝突回避をすることは少なくなる．

開番地法で格納されたデータの探索はハッシュによる key の決定と，そのあとの線形探査で達成される．衝突する回数が少なく済めばほぼ 1 回の計算で所望のデータに到達できる．当然計算量は $O(1)$ である．

また，開番地法を使うときはハッシュ表の大きさが重要になる．これは単純に衝突回数が増えてしまうためだ．基本的に格納したいデータの個数の 2 倍のサイズのハッシュ表を用意できることが理想的であるとされている．もちろん，データの個数より大きくさえあれば理論的には問題ないが恩恵を受けられなくなる．

3.3　バケットソート

データの数値としての桁数が一致している場合，それらのソートはバケットソートと呼ばれるソートによって計算量 $O(1)$ を実現することができる．

具体例として，3 桁の整数のソートを実行する．

$$
\{312, 111, 213, 321, 132\}
$$

今回は 1 から 3 の数字しか現れてはいないが，一般に 0 から 9 までで問題ない．まず，現れる整数 1 から 3 に対してバケットと呼ぶラベル付きの容器を用意する．1 から 3 の番号が振られたバケットに対し，1 の位とラベルが一致するようにデータを振り分ける．

$$
\left\{\begin{matrix}111\\321\end{matrix}\right\}_1, \left\{\begin{matrix}312\\132\end{matrix}\right\}_2, \{213\}_3
$$

これを 1 番のバケットから順に数をとる．

$$
111, 321, 312, 132, 213
$$

となり，これで 1 段階目が終了する．次は十の位とラベルを一致させる．

$$
\left\{\begin{matrix}111\\312\\213\end{matrix}\right\}_1, \{321\}_2, \{132\}_3
$$

再度，数を取り出す．

$$111, 312, 213, 321, 132$$

最後に百の位とラベルが一致するようにバケットに移す．

$$\begin{Bmatrix}111\\132\end{Bmatrix}_1, \{213\}_2, \begin{Bmatrix}312\\321\end{Bmatrix}_3$$

取り出すとソートは完了する．

$$111, 132, 213, 312, 321$$

第 4 章

再帰呼び出し・分割統治・縮小法

4.1　再帰呼び出し

手続きの中で自分自身を呼び出すことを再帰呼び出しという．

4.1.1　階乗

具体的な再帰呼び出しの例をいくつか挙げる．まずは階乗だ．

```
int factrial(n){
    if (n=0) return 1;
    else return factrial(n-1)*n;
}
```

このように 1 つ小さな数の階乗関数を呼び出している．計算量は $O(n)$ である．

4.1.2　ハノイの塔

もう 1 つ代表的な例としてハノイの塔を挙げる．問題設定としては A,B,C の 3 つの支柱のうち，A に n 段の塔がある状態から C へ移動させるというものだ．これをアルゴリズムで表現すると次のようになる．

```
MOVE(n,A,B,C){
    if(n=1){
        A -> C
```

```
    }
    else{
        MOVE(n-1,A,C,B)
        A -> C
        MOVE(n-1,B,A,C)
    }
}
```

となる．計算量は $O(2^n)$ となる．これはアルゴリズムが悪いのではなく，ハノイの塔という問題自体が複雑なのだ．MOVE 関数について説明する．これは，第 2 引数が移動させる前の塔の位置，第 1 引数は移動させる塔の段数，第 3 引数が目的地ではないほうの支柱，第 4 引数が目的の支柱となっている．当然，呼び出されるタイミングで支柱の関係は変わる．

4.2 分割統治法

大きな個数のデータに対する整列や探索などの操作はどうしても計算量が増えてしまう．しかし，問題の中には個数を分割し小さくすることで，小さくした以上に計算量が小さくなるものがある．この性質を利用したものが分割統治法だ．

4.2.1 マージソート

早速，アルゴリズムを示す．

```
Merge_Sort(s){
    if(s.records == 2){
        return [min(s),max(s)];
    }
    else{
        [s_1,s_2] = s;
        Merge_Sort(s_1);
        Merge_Sort(s_2);
        return [s_1 s_2];   \\小さいもの順で合わせる.
    }
}
```

これを具体例から確認する．

$$\{3, 8, 6, 1, 9, 2, 4, 5\}$$

これをマージソートでソートする．まずは分割する．

$$\{3,8,6,1\},\{9,2,4,5\}$$
$$\{3,8\},\{6,1\},\{9,2\},\{4,5\}$$

2 つの組になったら，その大小で並び替える．

$$\{3,8\},\{1,6\},\{2,9\},\{4,5\}$$

ここから合体させる．

$$\{1,3,6,8\},\{2,4,5,9\}$$
$$\{1,2,3,4,5,6,8,9\}$$

計算量は $O(n\log n)$ であり，効率の良いソートの 1 つである．

4.2.2 分割統治の計算量

マージソートというのは個数 n のソートを $n/2$ のソート 2 つ分にするという変換を行うことで計算量を小さくしている．さらに，分けた問題の解決結果の統合には計算量として cn が必要である．半分という分け方を使ったが，これを一般化し，大きさ n の問題を大きさ n/b の問題が a 個あるという風に分割したときの計算量を考える．

計算量を $f(n)$ であらわすと，分割は次のように表現される．

$$f(n)=\begin{cases}c, & n=1\\ af(n/b)+cn, & n\geq 2\end{cases}$$

式を見ても b に対して a が小さければ計算量が小さくなることが予想できる．これをまとめたのが次の式だ．

$$f(n)=\begin{cases}O(n), & a<b\\ O(n\log n), & a=b\\ O(n^{\log_b a}), & a>b\end{cases}$$

ハノイの塔は再帰呼び出ししても問題のサイズが 1 しか小さくならない．したがって計算量が大きくて当然なのだ．

また $a=2,b=2$ となる例がマージソートであったが，そのほかの値となる例を 1 つ挙げる．2 進数の掛け算だ．偶数の n ビット整数の x,y の掛け算を分割統治で実行する．

$$x=p\cdot 2^{n/2}+q,\quad y=r\cdot 2^{n/2}+s$$

このように前半のビットと，後半のビットで分けると p,q,r,s は $n/2$ ビットの正数となる．ここから計算をする．

$$\begin{aligned}x\cdot y&=(p\cdot 2^{n/2}+q)\cdot(r\cdot 2^{n/2}+s)\\ &=pr\cdot 2^{n}+\{(p+q)(r+s)-pr-qs\}2^{n/2}+qs\end{aligned}$$

この計算では $n/2$ ビットの掛け算が $pr,qs,(p+q)(r+s)$ の 3 つが現れる．したがって

$$a=2,b=3\Rightarrow f(n)=O(n^{\log_b a})$$

4.3　フィボナッチ数列

フィボナッチ数列の第 n 項を求めるアルゴリズムには計算量が多いものからすくなものまでいくつか存在する．本章のまとめとして，そのアルゴリズムを紹介する．

4.3.1　再帰呼び出し

再帰呼び出しを使うアルゴリズムは次の通りだ．

```
Fibonacci_1(n){
    if(n>=2){
        return Fibonacci_1(n-1)+Fibonacci_1(n-2);
    }
    else{
        return 1;
    }
}
```

このアルゴリズムの計算量は少々面倒な計算の結果

$$f(n) = O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$$

となる．

4.3.2　ループ

次は数列の計算を愚直にループで回す方法だ．

```
Fibonacci_2(n){
    if(n>=2){
        for(i=2;i<n;i++){
            z=x+y;
            x=y;
            y=z;
        }
    }
    else{
        return 1;
    }
    return z;
}
```

このようになる．計算量は $O(n)$ である．これが自然な実装ではないだろうか．しかし，これより効率の高い方法が存在する．それも無理数の含まれるフィボナッチ数列の一般項を使うわけではない．

4.3.3 行列の利用

フィボナッチ数列は以下の 3 項間漸化式で表現できる．

$$a_{n+2} = a_{n+1} + a_n, \quad a_0 = a_1 = 1$$

これを行列で表現する．添字はうまくずらす．

$$\begin{pmatrix} a_n \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} a_{n-1} \\ a_{n-2} \end{pmatrix}$$

繰り返しこの行列での漸化式を使うと

$$\begin{pmatrix} a_n \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \begin{pmatrix} a_1 \\ a_0 \end{pmatrix}$$

となる．つまり，

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n} (= A(n))$$

を直ちに計算できればフィボナッチ数列の一般項は計算できる．

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = X, \quad \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} = Y$$

としてアルゴリズムを示す．

```
Matrix_A(n){
    if(n=1) return X;
    else{
        if(n%2==0){
            B=A(n/2);
            return B*B;
        }
        else{
            B=A((n-1)/2);
            return B*B*Y;
        }
    }
}
```

分割統治において $a = 1, b = 2$ の状態となる．計算量を計算する．

$$
\begin{aligned}
f(n) &= f(n/2) + c \\
&= f(n/2^k) + kc \\
&= f(1) + c\log n \qquad (n = 2^k)
\end{aligned}
$$

となる．ここからフィボナッチ数列の一般項導出の計算量は $O(\log n)$ と結論付けられる．

4.4 縮小法

分割統治では大きさ n の問題を a 個の問題に分け，それぞれの問題の大きさが n/b となる場合を考えた．このときの計算量は a, b の大小関係によって次のように計算された．

$$
f(n) = \begin{cases} O(n), & a < b \\ O(n\log n), & a = b \\ O(n^{\log_b a}), & a > b \end{cases}
$$

分割統治法の利点は扱う問題のサイズを小さくすることで効率の良いアルゴリズムを作ることができる点である．そのため分割後の問題の総量 an/b は元の n より大きくても仕方なかった．しかし，分けるだけで問題の総量自体を小さくすることができる場合がある．上の計算式では $a < b$ の状態であり，その場合の計算量は $O(n)$ となる．

縮小法ではこの分割の大きさを一般のものとして考え，分けることで問題の総量を小さくできる場合を考える．

4.4.1 計算量

サイズ n の問題 P を同じ形式の問題 $P_1, P_2, \cdots$ に分解する．分解された問題のサイズは $s_1 n, s_2 n, \cdots$ である．問題を分割し，それが元の問題の総量より小さいので

$$
0 < s_1, s_2, \cdots < 1, \quad s_1 + s_2 + \cdots < 1
$$

となる．

問題の分割は再帰的におこなうとして，ある程度のサイズ (n_0) になると分割はやめ，直接問題を解くとする．このときの計算量を c とする．分割した問題の結果を合わせるのにかかる時間がもとの問題のサイズ n に比例するものとすると，問題の計算量は次のように計算される．

$$
f(n) = \begin{cases} c, & n \le n_0 \\ f(s_1 n) + f(s_2 n) + \cdots + cn, & n > n_0 \end{cases}
$$

関数 f は問題 P と同形式の問題の計算量を導く関数である．ここから

$$
f(n) \le \frac{cn}{s_1 + s_2 + \cdots} \qquad \therefore f(n) = O(n)
$$

となる．この式は数学的帰納法より証明される．

4.4.2　定順位要素の抽出

要素数 n の集合から k 番目に大きな要素を取り出すことを考える．このとき，ソートしてから取り出すことを考えると，ソートのアルゴリズムによるが計算量は $O(n\log n), O(n+k\log n)$ となる．しかし，必要なのは k 番目に大きな要素であるので，ソートでは少々余計なことをしている印象となる．

以下に縮小法を利用したアルゴリズム ORDER(S,k) を示す．

1. $|S|<100$ ならば，単純に S をソートして k 番目の要素を取り出す．
2. $|S|\geq 100$ ならば，
   （a）S を 5 個ずつのグループに分け，各グループの 3 番目の要素を取り出し，それらで集合 T を作る．
   （b）m = `ORDER(T,[n/10])` とする．`[]` は中の数字を超えない最大の正数である．
   （c）m より大きな S の要素を S_1 に，等しいものを S_2 に，小さなものを S_3 に分割する．
   （d）$k\leq|S_1|$ ならば $y=\mathrm{ORDER}(S_1,k)$，$|S_1|<k\leq|S_1|+|S_2|$ ならば $y=m$，$|S_1|+|S_2|<k$ ならば $y=\mathrm{ORDER}(S_3,k-|S_1|-|S_2|)$ とする．
   （e）y を出力とする．

m の決定は集合 T の中央値をとっていることに注意してほしい．中央値を 2 回とった格好になっている m はこのアルゴリズムの中では”とてもいい数”となっている．これは図 4.1 から確認できる．中央の赤丸が m を示している．これは T について降順ソートがなされている．左の方が大きく，右のほうが小さい．この状態であれば，m に対して左上の位置にある要素はすべて m より大きく，右下の位置にある要素はすべて m より小さいことになる．ここから，m は S の中に自らより小さい数を $|S|/4$ 個，自らより大きい数を $|S|/4$ 個持つことがわかる．

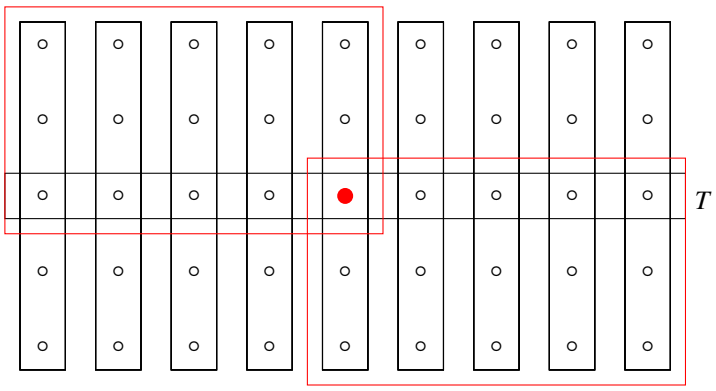

図 4.1　m の位置

このような性質から，計算量の計算は次のようになる．

$$f(n)=f\left(\frac{1}{5}\cdot n\right)+f\left(\frac{3}{4}\cdot n\right)+cn$$

第 1 項は集合 T から m を計算するときの計算量．第 2 項は最悪の分割の場合の残りの計算量を表している．

これは

$$s_1=\frac{1}{5},\ s_2=\frac{3}{4}\ \Rightarrow\ s_1+s_2<1$$

となるので計算量は $O(n)$ となる．

第 5 章

グラフ入門

5.1　グラフの定義

グラフとは有限との頂点と頂点同士をつなぐ辺の集合をいう．$G(V, E)$ と表記する．図 5.1 のようなものがその例である．

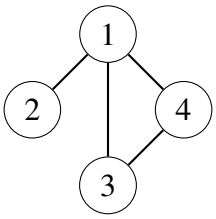

図 5.1　グラフの例

5.1.1　隣接行列

各頂点が辺によって結ばれているのか行列によって表現したものを隣接行列という．図 5.1 のグラフに対する隣接行列は次のようになる．

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

行と列の番号を見て，辺があるところに 1，ないところに 0 となるように行列を作っている．

5.2　2 分木の探索

5.2.1　問題設定

しばらく探索で扱うグラフは 2 分木だ．図 5.2 の 2 分木のグラフを 2 つの方法で探索する．ここでの探索とは根と隣接の関係[*1]しか知らない状態でグラフ全体の頂点を把握することをいう．

[*1] 2 つの子の存在.

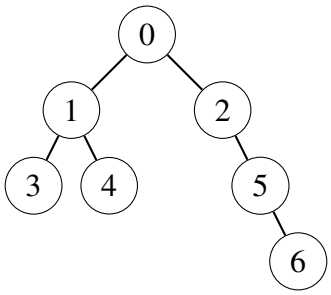

図 5.2　探索の対象のグラフ

探し方の方針は 2 つ存在する．幅優先探索と，深さ優先探索の 2 つである．この言葉の説明は後にするとして，実際に探査を実行する．

5.2.2　基本的なアルゴリズム

アルゴリズムを示す．

1. 頂点を記録しておく領域を A，B と 2 つ作る．
2. 2 分グラフの根 v_0 を A，B に入れる．
3. A に記録されている頂点から 1 つを取り出す．取り出された頂点と隣接する頂点を確認し，B に記録されていないものを，A，B に記録する．そして最初に取り出したものは A から削除しておく．
4. A に記録されている頂点がなくなるまで 3. の動作を繰り返す．
5. B を出力とする．

考え方は単純で，隣接関係で現れてくるすべての頂点を 1 つずつ B に記録していくということだ．B に関しては前章で扱ったハッシュなどを使うと効率が良い．これは B にある頂点か否かの判定を少ない計算量で実行するためだ．

さて，ここで気になるのは A から頂点を取り出すときの規則だ．アルゴリズムであるから，無作為というわけにはいかない．さらには，取り出し方の規則から出力 B の作成過程に変化が生まれるかもしれない．そこで，以降は A の部分の工夫について扱う．

5.2.3　キュー待ち行列

キューとは，FIFO(First In First Out) という原則で A を扱う手法である．A に記録された頂点の中から，最も早い段階で記録された頂点を取り出すというものだ．このキューを実現することを考えると，A のデータ構造は配列またはリストであるといい．

配列によってキューの説明をする．A 配列に対して，データが格納されている先頭を指すポインタ head とデータの最後尾を指すポインタ tail を与える．A に新たな記録が格納されるときは tail をインクリメントしたうえで，tail の指す場所に頂点を格納すればいい．そして A から取り出すときは head が指す頂点を取り出し head をインクリメントすればいい (図 5.3).

このようにすると配列でキューを実現できる．しかし，配列では一度使った領域がゴミになるので説明で使える程度だ．実際にはリストを用いて実現する．これに関しては各自の学習に任せる．

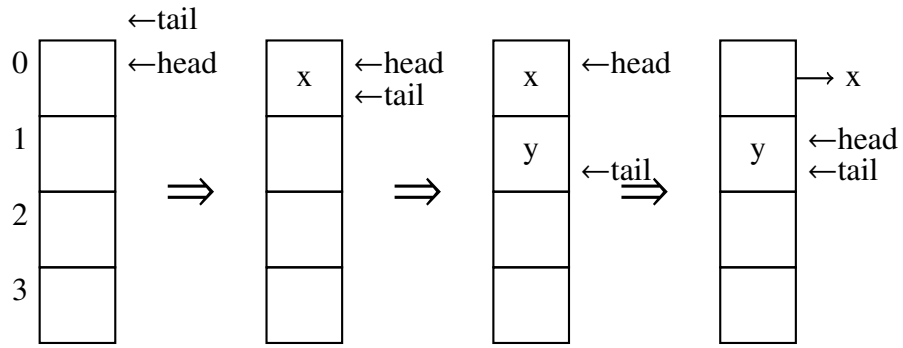

図 5.3　キューの実現

では，実際に図 5.2 の探索をおこなう．手順を進める中での A と B の中に格納される頂点を表 5.1 で示す．

表 5.1　キューを用いた探索

A	B
0	0
1,2	0,1,2
2,3,4	0,1,2,3,4
3,4,5	0,1,2,3,4,5
4,5	0,1,2,3,4,5
5	0,1,2,3,4,5
6	0,1,2,3,4,5,6
-	0,1,2,3,4,5,6

B に記録される頂点は同じ高さのものからであることがわかる (図 5.4)．これが幅優先探索だ．

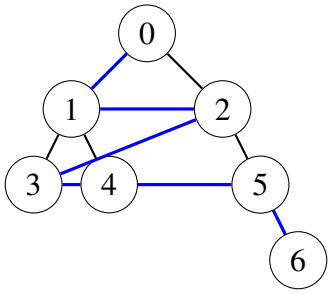

図 5.4　幅優先探索で B に記録される順番

5.2.4　スタック

スタックとは，LIFO(Last In First Out) という原則で A を扱う手法である．A に記録された頂点の中から，最も遅い段階で記録された頂点を取り出すというものだ．これを実現する A データ構造は配列かリストであるといい．

配列によってスタックを説明する．A 配列に対して最後に格納されたデータを指すポインタを top として与える．新たな頂点が格納されるときは top をインクリメントし，top の指す場所に頂点を格納する．頂点を取り出すときは top の指す頂点を取り出したうえで，top をディクリメントとする (図 5.5).

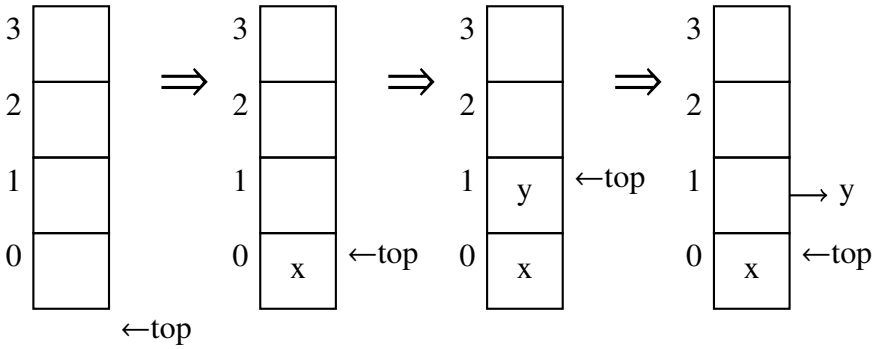

図 5.5　スタックの実現

これで図 5.2 の探索を実行してみる．表 5.2 にその様子を示す.

表 5.2　キューを用いた探索

A	B
0	0
1,2	0,1,2
1,5	0,1,2,5
1,6	0,1,2,5,6
1	0,1,2,5,6
3,4	0,1,2,5,6,3,4
3	0,1,2,5,6,3,4
-	0,1,2,5,6,3,4

頂点の現れる順番を示すと次のようになる．このような探索を深さ優先探索という.

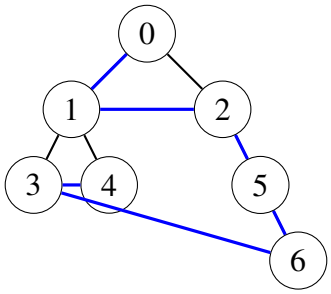

図 5.6　深さ優先探索で B に記録される順番

第 6 章

グラフを利用した諸問題

6.1　最短路探索問題

6.1.1　アルゴリズム

グラフの辺に非負の整数を与えるとその整数は辺を通過するためのコストを表すことになる．任意の頂点の組の間を移動することを考える．最も移動にかかるコストが小さくなるときの経路を探すのが最短路問題だ．

次の図 6.1 のようなグラフを考える．

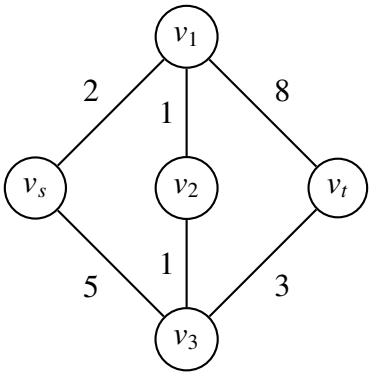

図 6.1　最短路問題の例題

このようなグラフで v_s から v_t へ向かう最短路を求めるアルゴリズムを示す．頂点集合を V，頂点 v_s から $v \in V$ へ向かうコストを $d(v)$ で表すとする．

1. $d(v_s) = 0$ とする．
2. $v \in V, v \neq v_s$ に対して $d(v) = \infty$ とする．
3. 集合 A に頂点 v_s を記録する．
4. 集合 A に記録されている頂点のうち $d(v)$ が最小となる v を取り出す．
5. 先の操作で取り出された頂点が v_t であるとき，$d(v_t)$ を出力としてアルゴリズムは終了する．
6. 取り出した v に隣接する各頂点 w に対して，

```
if (d(w)= nan){
    w を A に追加
```

```
    d(w)=d(v) + l(v,w)
}
else if (d(w)>d(v) + l(v,w)){
    d(w) = d(v) + l(v,w)
}
```

7. 4. に戻る．

操作 4. において集合 A から頂点を取り出すとき，$d(v)$ が最小となる v を選んでいるが，これを評価値優先という．この A からの頂点のとり方が最短路を保証している．これは頂点 v を取り出したときの $d(v)$ がその v に到達するまでの最短路となっていることが理由だ．最短路を作り続けて v_t を目指しているとこのアルゴリズムは解釈できる．

操作 4. で頂点集合 A で最も v_s に近いもの v が取り出されるが，もしそのときの経路よりさらに小さい経路があることを仮定すると，それはすでに頂点集合にある別の頂点 w を経由して v に到達する場合と，まだ A に記録されていない w' を経由して v に到達する 2 通りの場合が考えられる．

前者はまず起き得ない事態だ．これは A から最小の $d(v)$ をとるという 4. の操作に矛盾する[*1]からだ．また，後者の場合は誤った v が取り出される前に w' が取り出される局面を迎えるので問題がない．以上によってアルゴリズムの正当性がわかる．

6.1.2 計算量

4. の評価値優先を実現するのは A のデータをヒープで管理することで行われる．ヒープの作成の計算量は頂点 1 つあたり $\log n$ となる．基本的に根を取り出し，頂点の再構築をおこなうので $\log n$ が何回出てくるのかが計算量になる．

4. の操作は頂点を取り出すので頂点数 n を最大とする．ことがわかる．6. では辺の長さが変更されてヒープを作り直すが，これは辺の数 m に比例し $m \log n$ となる．頂点と辺の数は明らかに辺の方が多いので計算量は $m \log n$ である．

6.2 最大流問題

6.2.1 問題設定

最大流問題を扱う上でのグラフは有向グラフである．始点と終点の頂点，そのほかの頂点，辺とその容量の集合 V が問題の対象である．

$$V = (V, \vec{B}, C, v_s, v_t)$$

具体的な例として図 6.2 のネットワーク[*2]を考える．

このネットワークにおいて v_s, v_t の間を流せる最大の量を求めるのが最大流問題だ．

[*1] $d(w) < d(v)$ が明らか．

[*2] V の 1 セットをネットワークという．

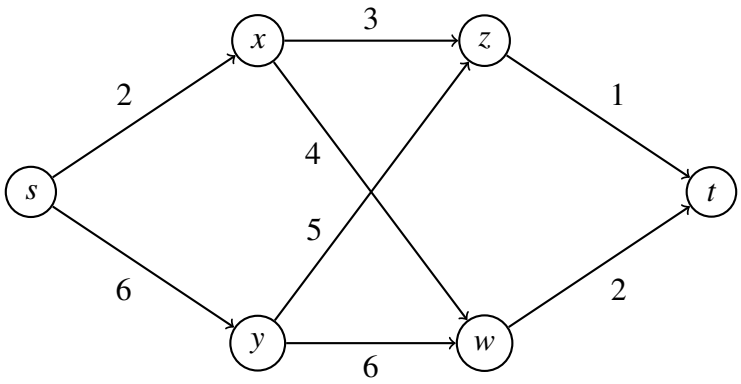

図 6.2　ネットワークの例

6.2.2　フローの定義

問題を解く上で重要な概念を定義する．頂点 x, y の間を流す情報の量を $f(x, y)$ と表現し，フローと呼ぶ．以下の 2 式を満たす．

$$0 \leq f(x, y) \leq C(x, y)$$
$$\sum_{x} f(x, v) = \sum_{y} f(y, v)$$

第 1 式は辺の容量を上回らないことを示していて，第 2 式は頂点へのフローの流出入は等しいことを示している．

このフローというものを各辺に対して適切に決定してゆくことが解決の糸口となる．

6.2.3　最大流の逐次構成法

方法は単純で，余裕のある辺に対してできるだけ多くのフローを与えるというものである．この”余裕のある”というものは次のように考えられる．

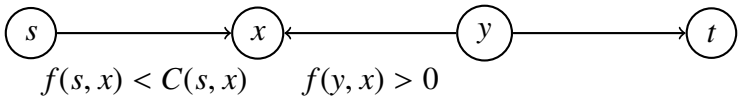

図 6.3　”余裕”とは

図 6.3 において，s から x に進む方向，x から y に進む方向には余裕があるという．フローを与えるルールは次のようになっているからだ．

- 辺の持つ方向と同じ方向にフローを与えるときは，既存のものに加算する．
- 辺の持つ方向と逆の方向にフローを与えるときは，既存のものに減算する．

この規則を守り，実際に最大流を求める．方法は実際に経路にフローをできるだけ大きく設定していくとい

う地道な方法である．また．フローの定義を崩さないように，t から s への辺を用意し無限大のフローを与えておく．

最初は s, x, z, t の経路を考える．図 6.4 から s, x, z, t の経路は 1 の容量を流すことができることがわかる．

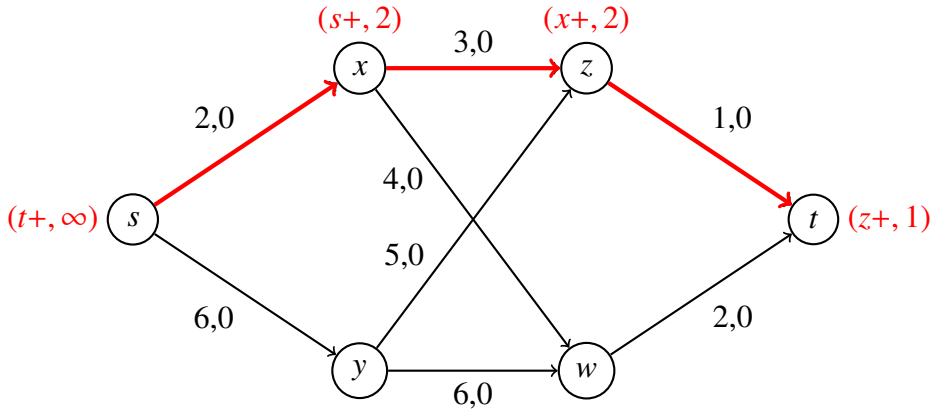

図 6.4　最大流問題 1

次も $s \to x$ で始まる経路を考える．x 以降は”余裕”のある経路を選択する．すでに $f(s,x) = 1$ なので新たに 1 をフローとして加算できる．ここから z へ進むと $f(z,t) = 1, f(y,z) = 0$ より，余裕がないので，x から w へ進む．あとは t へ進める．したがって，s, x, w, t の経路に容量 1 が流せることがわかる (図 6.5).

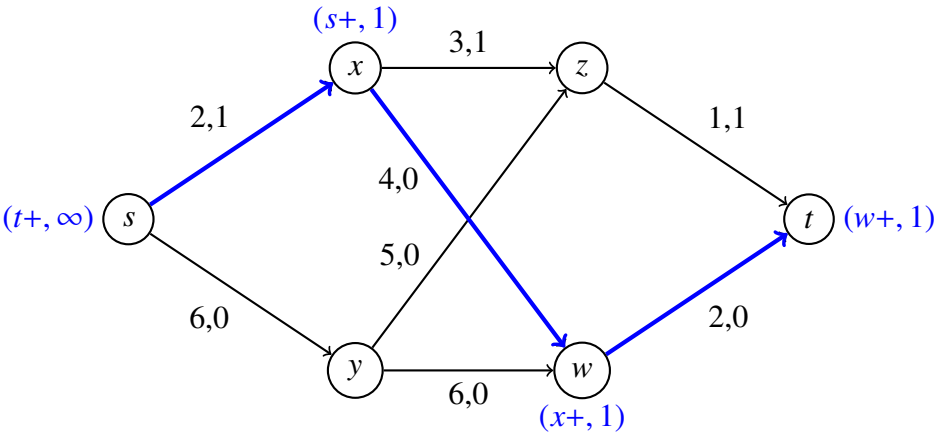

図 6.5　最大流問題 2

この段階で $f(s,x) = C(s,x)$ となるので $s \to x$ の経路は使えなくなる．そこで $s \to y$ から始まる．すると”余裕”のある経路を選択してゆくと $z \to x$ では逆走する場面のありながら容量 1 が伝わることがわかる (図 6.6).

ここで t へ流入する経路 $z \to t, w \to t$ がともに容量いっぱいとなるのでこれ以上は流せない．この操作を繰り返すと結論としては図 6.7 となる．

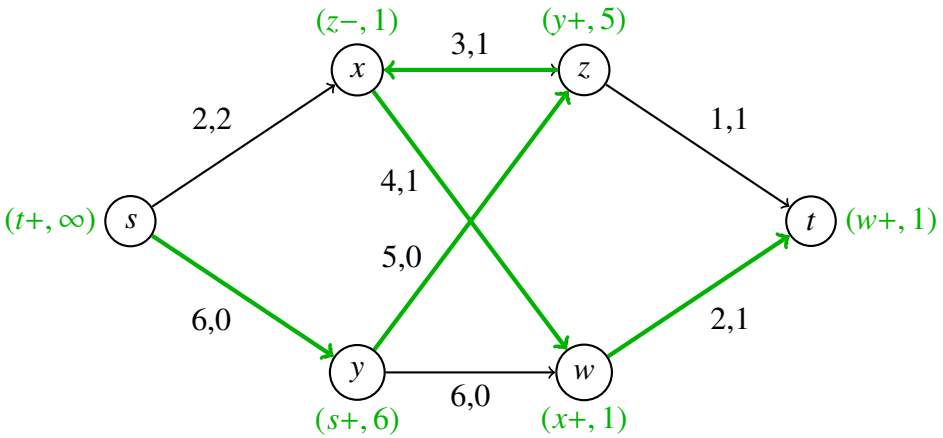

図 6.6　最大流問題 3

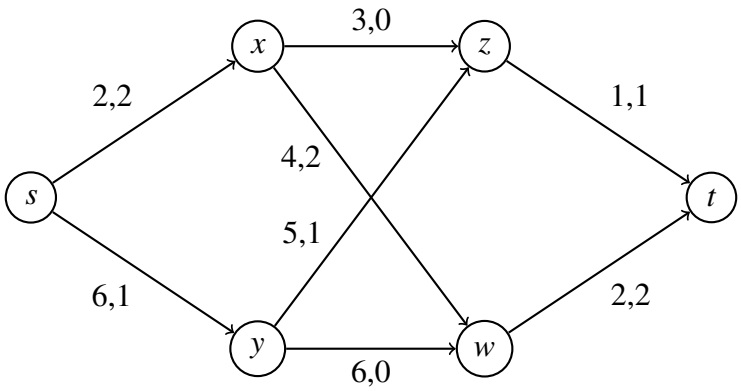

図 6.7　最大流問題 4

6.3　最大マッチング

6.3.1　2 部グラフ

グラフのうち，頂点を 2 つの集合 U, V に分けることができ，辺を共有する頂点の組は必ず U から 1 つ，V から 1 つ頂点をとったものとなるものを 2 部グラフという (図 6.8).

6.3.2　最大マッチングの解決

最大マッチング問題の例として挙げられるのが集団お見合いの問題だ．男女が集合 U, V にあたる．良いと思った人へ辺を作りグラフを作成すると 2 部グラフになる[*3]．このうち，最大でいくつのカップルができると考えらるかを考えるのが最大マッチング問題だ．解き方は容易で次のようなグラフを作る (6.9)．そして，各辺の容量を 1 として $s \to t$ の最大流問題を解けばいい．

[*3] 同性の人が気にいった場合は面倒なので考えていない．

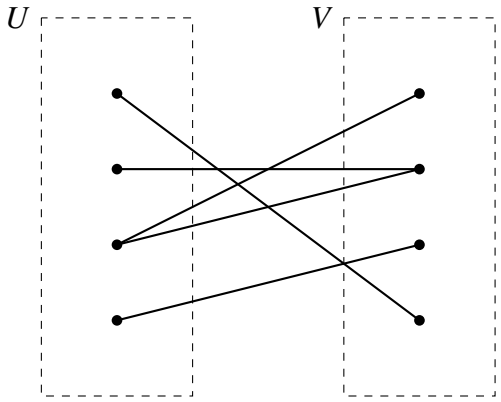

図 6.8　2 部グラフ

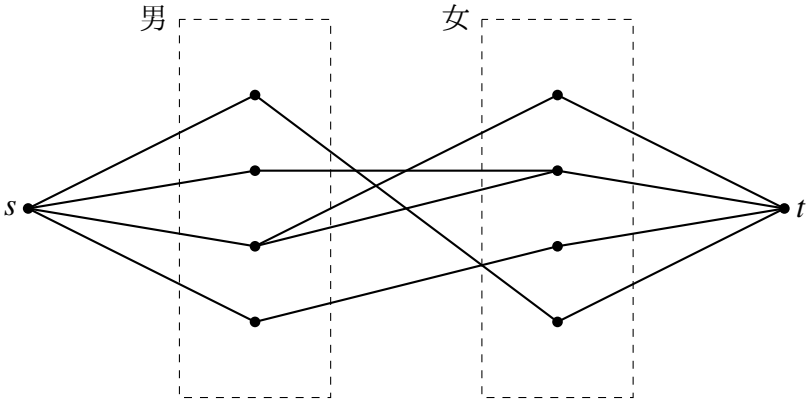

図 6.9　最大マッチング

6.3.3　割り当て問題

最大マッチングでは男女のカップルのように 1 対 1 の対応を考える場合を考えてた．それを拡張する．

問題の例は学生のゼミ所属である．設定として，学生は 2 つのゼミにできるということ，1 つのゼミは 3 名までの生徒を受け入れることができる．このような問題を割り当て問題という．生徒の集合とゼミの集合を頂点集合として問題を解く．これは，頂点 s からから生徒集合までの辺の容量を 2 に，ゼミ集合から頂点 v までの辺の容量を 3 に，生徒・ゼミ間の辺の容量を 1 として最大流問題を解くことで解決できる．

割り当て問題はさらに複雑化できるが，基本的には始点と集合 U までの辺，集合 V から終点までの辺の 2 種の辺の容量に問題の制限となる値を代入し，その他の容量を 1 としてグラフを作成したら，あとは最大流問

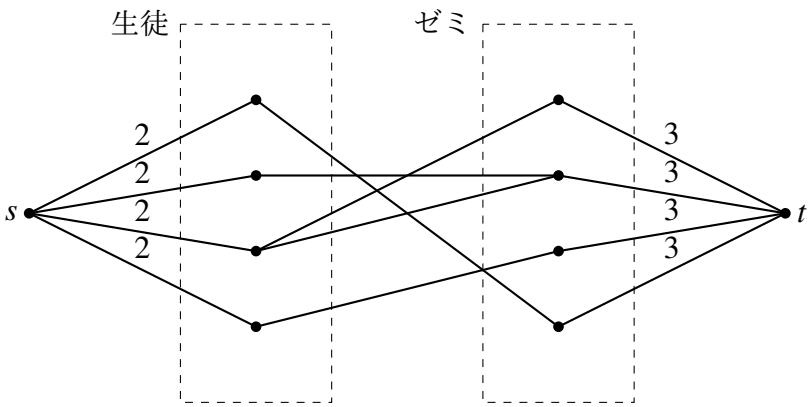

図 6.10　割り当て問題

題として解くことができる．

第 7 章

動的計画法

7.1　最適性の原理

7.1.1　最適性の原理と動的計画法

無数の頂点集合から始点と終点をつなぐ最短路を得たとする．このとき，最短路中にある 2 頂点をとると，その頂点間の最短路は初めに得た最短路の部分路となっている．最短路とは部分的な最短路の集合といえる．

この例が示すのは，大問題の最適解はその大問題を構成する小問題の最適解の積み重ねで与えることができるということだ．そして，この考え方に沿い大問題を解こうとする手法を動的計画法という

7.1.2　最短経路

図 7.1 のグラフの最短経路を考える．ここではどこか始点と終点をおいてその最短経路を求めるのではなく，すべての頂点の組に対して最短距離を与えることを目的としている．

アルゴリズム All Sortest Path は以下の通りとなる．

```
for i = 1 to n-1 do
    for j = i to n do
        d_0(i,j) = l(i,j)
```

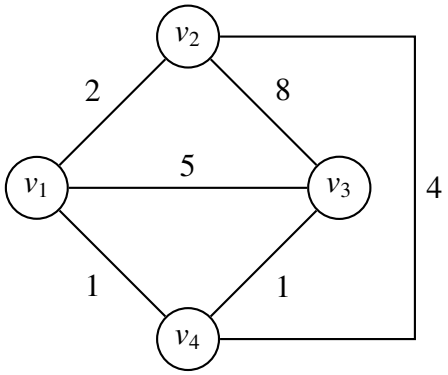

図 7.1　最短経路

```
for k = 1 to n do
    for i = 1 to n-1 do
        for j = i+1 to n do
            d_k(i,j) = min{d_{k-1}(i,j), d_{k-1}(i,k)+d_{k-1}(k,j)}
```

このアルゴリズムの計算量は最後の 3 重ループによって決定し $O(n^3)$ となる．流れとしては，初めに共有した辺を通っただけの距離を格納した `d_0` を作成する．そのあとの 3 重ループで `k` の値が示しているのは経由を許可された頂点である．`k=1` ならば，頂点 v_1 を経由した距離とそれまでの距離を比較してより良いほうを保存するのだ．すべての頂点が通過を許可されたときの，すべての組み合わせが吟味されたことになり，全ての頂点の組の最短経路が得られたことになる．

実際に図 7.1 においてアルゴリズムの実行してみる．結果は表にまとめる．

表 7.1　アルゴリズム：All Shortest Path

k	(1,2)	(1,3)	(1,4)	(2,3)	(2,4)	(3,4)
0	2	5	1	8	4	1
1	-	-	-	7	3	-
2	-	-	-	-	-	-
3	-	-	-	-	-	-
4	-	2	-	4	-	-

7.2　弾性マッチング

7.2.1　弾性マッチングと最小コスト

弾性マッチングとは以下の条件を満たす 2 列の要素の対の集合 T

- 各要素は少なくとも 1 つの対に含まれる．

- 交差しない．

2 つ目の交差しないというのはグラフにして表したときに，辺が交差しないということだ．図 7.2 のようなグラフになる．

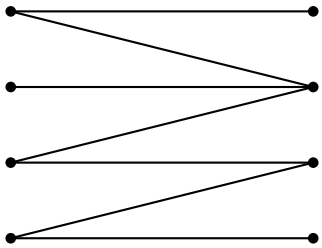

図 7.2　弾性マッチング

弾性マッチングを格子点で表現する．すると図 7.3 のようになる．単調増加のグラフのようにも見える図形が生まれる．点で結んだ関係を a_1, b_1 の点から a_m, b_n の方向になぞることを考えると，その線の向きは 3 つしかない．コスト最小の弾性マッチングを求めるには，この 3 方からのコストのみを考えて最小を選べばよい．これは計算量の削減に大きく役立つ性質である．

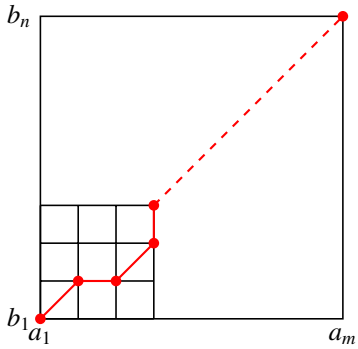

図 7.3　格子点で表現した弾性マッチング

コスト最小の弾性マッチングを求めるアルゴリズム ELASTIC-MATCHING を以下に示す．

```
begin
    d(1,1) = c(1,1);
    for j = 2 to n do{
        d(1,j) = d(1,j-1) + c(1,j);
    }
    for i = 2 to m do
        for j = 2 to n do
            d(i,j) = min{d(i-1,j-1), d(i,j-1), d(i-1,j)} + c(i,j);
```

計算量は 2 重の繰り返しにより $O(mn)$ となる．このアルゴリズムの流れとしては，まず 1 つ目の繰り返しによって，(1,1) から (1,n) までのコストをすべて計算する．ここから経路を右に広げてゆく．経路は常に下から

完成するので，3 方の最も良いものの選択が可能になっている．もちろん，途中で不要となるデータが存在するが，これは計算のコストと考えるべきだ．

ここでアルゴリズムを具体的に実行してみる．表 7.2 は要素間の辺のコストを示している．

表 7.2　最小弾性マッチングの例

	a_1	a_2	a_3	a_4
b_1	2	5	8	3
b_2	1	6	2	6
b_3	7	1	5	4

各要素を座標とするような格子を作図し，格子点に表 7.2 の値を記す．そこから 3 方向の移動によってコスト最小となるもののみを選んで経路を作成する．すると図 7.4 のようになる．

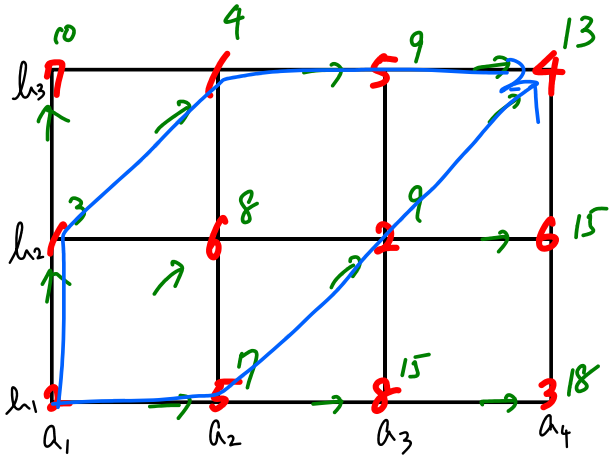

図 7.4　弾性マッチングのアルゴリズム実行例

7.2.2　編集距離

弾性マッチングの活用法の 1 つに編集距離がある．例えば，google の検索などで検索ワードに誤りがあった場合でも検索エンジンが真の単語を推測した検索結果を表示することがある．これは誤った検索ワードと真の単語の間の編集距離を計算し，その編集距離が小さいことを根拠に入力が誤りであると推測しているのだ．

具体例として単語”MILER”を”HILLER”と誤った場合を考える．ここでも図 7.3 ようなものを作成する．編集距離は 2 と分かる．

この図 7.5 の作成は先に示した弾性マッチング作成のアルゴリズムを少し変えたものによって作成される．まず文字が格子点の座標を示してはいないことに注意したい．コストは格子点の間の移動に当てられている．これは文字列を比較するときに以下の 3 つの誤りの訂正方法が考えられる．

- 正しい文字と入れ替える (sustitution)
- 余計な文字を消す (deletion)
- 必要な文字を挿入する (insertion)

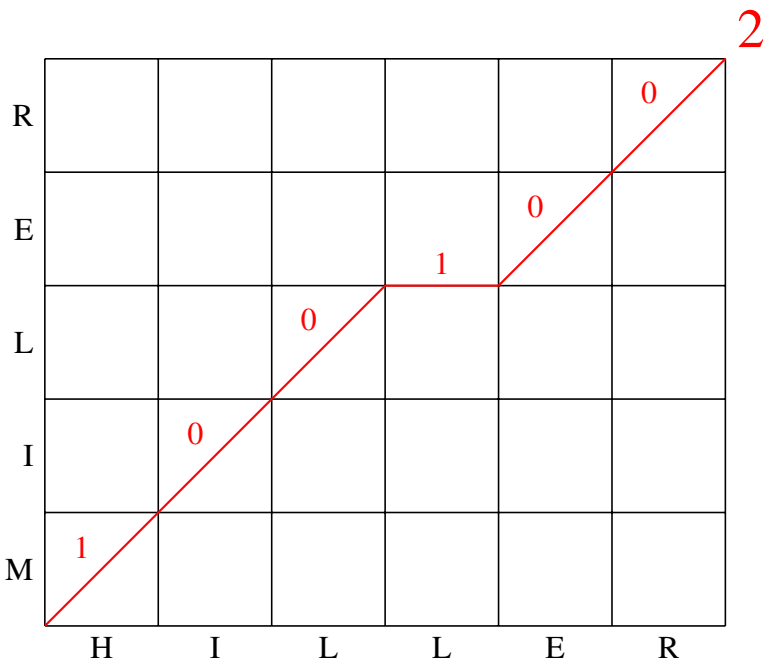

図 7.5　編集距離の例

この 3 つの誤り方を図 7.5 で表現すると辺の方向がそれぞれ違う．例えば 1 つ目の誤り方は斜めの方向となるし，2 つ目では横，3 つ目では上に移動する．ここで誤りに対してすべてコストを 1 と設定すれば ELASTIC-MATCHING で次のようにすれば編集距離を求めることができる．

```
d(i,j) = min{d(i-1,j) + 1, d(i,j-1) + 1, d(i-1,j-1) + c(i,j)}
```

移動方向に対してコストを与えるので，最小の比較の前にコストの加算がなされている．c(i,j) は文字の誤りを判定していてる誤り文字列の i 番目の文字と真の文字列の j 番目の文字が等しければ 0，異なれば 1 となる．

誤りに対するコストの設定によって，どのような文字列が編集距離が小さいとなるかが変わる．

第 8 章

自然言語処理

8.1 文字列マッチング

本節では，長い文字列から単一のパターンが発見されるのかという文字列マッチングを扱う．そのアルゴリズムはいくつか存在する．

8.1.1 簡単な文字列マッチング

簡単に思いつく文字列マッチングをいかに示す．文章の長さは n，マッチングするパターン長は m である．

```
i = 0, j = 0;
while(i<n){
    while(j < m and text[i+j] == pattern[j])
        j++;
    if(j == m)
        return i;
    else
        i++, j = 0;
}
```

単純にパターンを文字列と照合し，合わなければ照合の開始位置を 1 つずらすというアルゴリズムだ．計算量は $O(n \times m)$ である．

どうしてもパターン長 n の分の一致の確認は避けられないが，一致しなかったときに 1 つずつシフトするのではどうしても無駄な計算が多いように考えられる．これは，照合させる文字列パターンの特徴を一切考慮しないアルゴリズムであるからだ．前もって与えられる情報を有効に使う必要がある．

8.1.2 KMP

文字列マッチングの有効なアルゴリズムの 1 つに KMP(Knuth-Morris-Pratt) というものがある．これは前もってマッチングさせるパターンに注目することで skip 配列を作成する．これは文字列とパターンを照合させるときに不一致が判明したときの位置に対応して次の照合の開始位置までいくつパターンをシフトさせたら

良いのかという値が格納されている．

具体的にパターン”abcd”に対して skip 配列を作成する．

1. パターンの’a’ で不一致が判明したら，そのほかにわかることはないので skip[0] は 1 とする．
2. パターンの’b’ で不一致が判明したら，照合開始位置から 1 つシフトしたところが’b’ ではないとわかる．この場合は’a’ であるかもしれないので skip[1] は 1 とする．
3. パターンの’c’ で不一致が判明したら，照合開始位置が’a’，照合開始位置から 1 つシフトしたところが’b’ であることがわかる．つまり，1 つシフトさせると確実にパターンの’a’ で不一致が判明する．したがって skip[2] は 2 とする．
4. パターンの’d’ で不一致が判明したら，照合開始位置から 2 つシフトまでは’a’ がないことが明らかなので skip[3]=3 となる．

このように skip 配列が完成すると KMP アルゴリズムは以下のようになる．

```
KMP(text, pattern){
    i = 0, j = 0;
    while(i < n){
        while(j < m and text[i+j] == pattern[j])
            j++;
        if (j == m)
            return i;
        else
            i += skip[j], j = max{j-skip[j], 0};
    }
}
```

j の更新に違和感を覚えた場合，実際にアルゴリズムを実行してみるとわかる．j をいちいち 0 にしていたのでは，無駄があるのでシフトさせる分だけ減らすので良いのはわかる．しかし，”abcd”において j=0 すなわち，’a’ で不一致を確認したとき j=0 にする必要がある．これは j-skip[0] では-1 となってうまくいかない．

”abcd”では skip=$\{1, 1, 2, 3\}$ となったが，”aaab”では skip=$\{1, 2, 3, 1\}$ となる．このとき，パターンの’b’ で不一致が確認された場合は，1 つシフトする．これは’b’ ではない部分が’a’ であるのかを確認する必要があるからである．しかし，3 つの’a’ を確認する必要はない．すでに 2 つ目までの’a’ までは一致していることがわかっている．そのため j-skip[j] が必要になるのだ．

8.1.3 BM

これは後ろから一致を確認し，不一致が確認されたときのテキスト側の文字からシフトの幅を決定する方法である．例えば文字列”never”に対して後ろから数えた文字数を与える (表 8.1).

この表を参考に不一致が確認されたときにいくつシフトするのかを決定する (表 8.2)．これはパターンに現れなかった文字に対しては文字列長を，パターンに含まれる文字に対してはずらし表の値で小さいほうを採用

表 8.1　BM 法のずらし表 1

n	e	v	e	r
4	3	2	1	0

する[*1].

表 8.2　BM 法のずらし表 2

n	v	e	r	その他
4	2	1	5	5

これを skip 配列[*2]として扱い以下のアルゴリズムを実行するのが BM 方だ.

```
BM(text,pattern){
    i = 0;
    while(i < text.length - pattern.length){
        j = m-1;
        while(j >= 0 and text[i+j] == pattern[j])
            j--;
        if(j == -1)
            return i;
        else
            i -= max{skip[text[i+j]]-(m-1-j), 1}
    }
}
```

このアルゴリズムの発想は誤った文字がパターンに含まれるとき，確実にそれが次は一致するようにシフトするということだ．パターンに含まれない文字の場合は大きなシフトが可能になる．

しかし，パターンに含まれる文字の場合はシフトの幅が小さくなることがある．その場合は後ろから照合することを前提に作成した KMP 法の skip 配列を併用することで，シフトの幅を維持することができる．

8.2　自然言語処理

40 年代に生まれたコンピュータにより機械翻訳という発想が生まれる．57 年にスプートニクショックが起きコンピュータの進歩は加速した．コンピュータネットワーク (web) は 90 年代以降に進歩し，現在ではコーパスなどのビッグデータを扱うようになる．

[*1] 同じ文字が複数個現れた場合，同じ文字の間の文字数を考える方が正しいように思われるが，その辺りはよくわからない．

[*2] r は 0 以外に値がない．この場合はパターン長 5 を採用する．

8.2.1　あいまい性

同じような形でも文脈によって正確な理解ができる．また，有名な例として”Time flies like an arrow.”という英文を与える．主語は”Time”なのか”Time flies”なのかというのは，もともとの意味を知っていいれば明らかであるが，コンピュータが解釈できるのかとなると難しい話ではないだろうか．

8.2.2　マルコフモデル

マルコフモデルというものは，すでにほかの講義においても顔を出している．出力を決定する確率が過去の出力に依存するものをマルコフモデルという．過去 m 回の出力に依存する場合，m 階マルコフモデルという．

マルコフモデルの中には隠れマルコフモデルと呼ばれるものがある．これは状態の遷移を出力によってのみうかがい知ることができるマルコフモデルのことだ．しかし，これは今まで扱っていたマルコフモデルと何ら変わらない．通常のマルコフモデルとの差は，状態が直接確認できるか否かのみである．

8.2.3　動的計画法による品詞の推定

先ほど登場した例文”Time flies like an arrow.”を単語に分解し，その品詞を判別することを試みる．これは動的計画法のうちでもビタビアルゴリズムを使う．

品詞に分解して考えられる組み合わせを図 8.1 に表す．

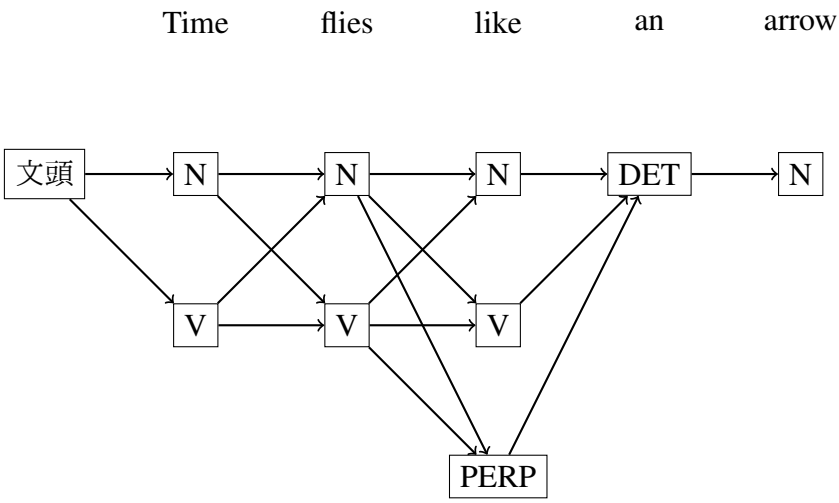

図 8.1　構文解析で考えられるすべての組み合わせ

この図 8.1 は有向グラフになっていて，コーパスから辺と頂点に以下のルールでコストを与える．

- コーパスではある品詞に対して，次にどの品詞が来るのかという確率がわかっている．辺にはその確率をコストとして与える．
- コーパスではある品詞と，その品詞をとることができる単語とに対して，品詞に対する単語の出現確

率[*3]がわかっている．頂点にはその確率をコストとして与える．

以上のようにコストを与え，そのコストを掛け算しながら最も大きくなる経路を探す．これによって与えられた経路の品詞が最もらしい品詞の組み合わせとなる．

8.2.4 構文解析

名詞と前置詞によって名詞句が作られ，さらに名詞句と前置詞句が合わさることで大きな名詞句ができて，という風にあらかじめ品詞がどのように複合してゆくかということをわかっている．具体的にはチョムスキー標準形という形で与えられている．

品詞を合わせ大きな句を作り，最終的には文をなすことを考える．この過程で現れる句などの枠組みは品詞の従属関係を表していることがわかる．これによって構文解析が可能となる．具体的な例として以下の英文の構文解析を試みる．

I saw a girl with a telescope.

一見すると”望遠鏡を持つ女の子を見た．”となるのか，”望遠鏡で女の子を見た．”となるのかの判別はできないが，コンピュータでもこの段階までは来れるのかというのがここでも問題だ．

人の手でこの解析をおこなると図 8.2 のようになる．

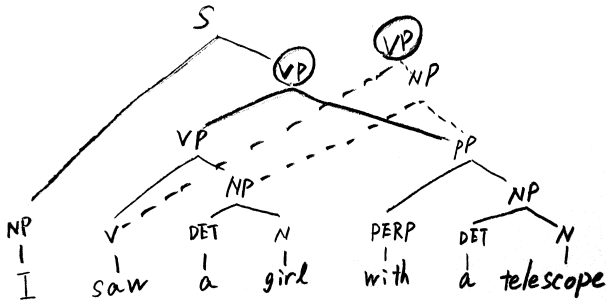

図 8.2　人の手による構文解析

これを計算機により実現するのが CKY 法である．やっていることは図 8.2 と同じだ．これを単一の動作により実現することでアルゴリズムをなしている．このアルゴリズムは以下の図のマス目を埋めることを目的としている．あるマス目を埋めるには，そのマスと底辺で構成される三角形に注目し，底辺を 2 分割して得られる 2 つの三角形の頂点のマスにある要素とチョムスキー標準形との比較でおこなう (図 8.3).

[*3] 例えば名詞の中で”Time”が現れる確率というものがわかっている．

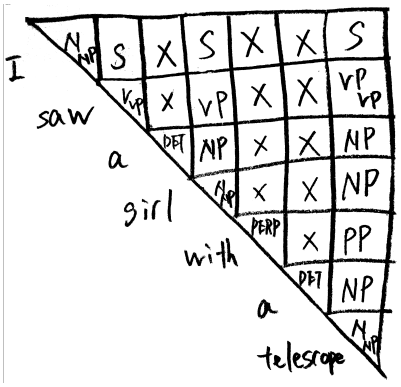

図 8.3　CKY 法の実行例

第 9 章

図形

9.1　凸図形・凸包

図形中の任意の 2 点をとり，その点よる線分を考える．この線分上のすべての点が元の図形に含まれるとき，その図形は凸であるという (図 9.1)．これは立体においても同様で，凸である図形を凸図形という．

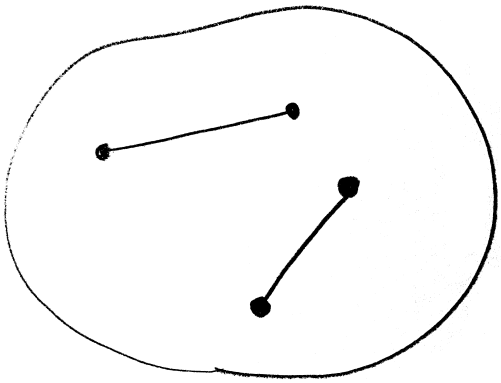

図 9.1　凸図形

凸ではない図形を与える．単純にはへこみのある図形のことだ．この図形を包含して，同時に凸である最小の図形を考える．これはちょうどへこみを埋めたような図形となる．このような図形を凸包という (図 9.2).

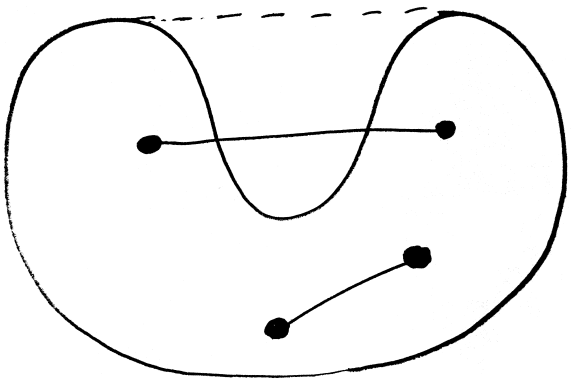

図 9.2　凸図形

9.2　ボロノイ図

9.2.1　ボロノイ領域

空間上に母点 $S = \{p_1, p_2, \cdots, p_n\}$ を与える．母点 1 つ 1 つに対してボロノイ領域というものを考えることができる．p_i のボロノイ領域とは，その領域内の任意の点に最も近い母点が p_i となる領域のことだ．これを図示したものをボロノイ図という．具体例として図 9.3 にボロノイ図を示す．ボロノイ領域を仕切る線をボロノイ辺と呼び，ボロノイ辺の交点をボロノイ点と呼ぶ．

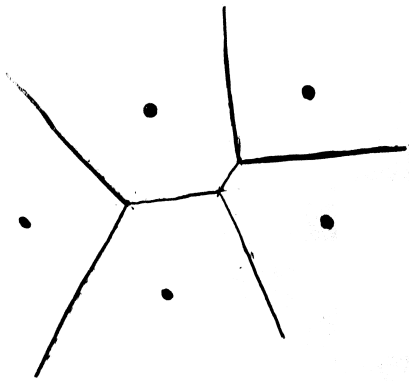

図 9.3　ボロノイ図の例

ボロノイ点は 3 点以上の母点を含む円の中心となっている．そして，その円の内部には他は母点が含まれないという性質を持つ．これを空円性と呼ぶ．図 9.3 から確認できる内容である．

9.2.2　ボロノイ図の性質

ボロノイ図を作成する元となるのは母点だが，この母点の個数によってボロノイ辺とボロノイ点の個数を考えることができる．母点の数を n，ボロノイ点の数を v，ボロノイ辺の数を e として計算する．

まず，ボロノイ点に接続する辺の延べ数を A として計算する．ボロノイ図の性質上，ボロノイ点には 3 本

以上の辺が接続するので

$$3v \leq A$$

と分かる．

次にボロノイ辺の延べ数は辺の両端の数だけカウントされると考えることができるので単純には $2e$ となる．しかし，ボロノイ辺の中には無限遠まで延び，端点にボロノイ点がないもののあるので実際はそれより少なくなる．つまり，

$$A < 2e \qquad \therefore 3v < 2e$$

を得る．

最後にオイラーの関係式を使う．

$$V - E + F = 2 \qquad (V : \text{頂点の数}, E : \text{辺の数}, F : \text{面の数})$$

オイラーの関係式を適応するためにボロノイ図に対して十分に大きな円を加えることで，無限遠に伸びているボロノイ辺に対しても端点を与える．この操作によって増える頂点の個数 $(V - v)$ と辺の個数 $(E - e)$ は等しいので，

$$V - E = v - e$$

を得る．また面の数は円の内部の面の数と，その外の領域の面 1 つ分を足したものになる．そして円内部においては母点がボロノイ領域として 1 つの面を作るので

$$F = n + 1$$

である．以上から

$$v - e + n = 1$$

を得る．

$$\begin{cases} 3v < 2e \\ v - e + n = 1 \end{cases} \Rightarrow \begin{cases} v < 2n - 2 \\ e < 3n - 3 \end{cases}$$

この結果から

$$\frac{2e}{n} < 6 - \frac{6}{n}$$

と式変形ができるが，左辺が示すのは 1 つのボロノイ領域の持つ辺の数の平均だ．n が大きくなることを考えると，ボロノイ領域は平均して六角形であることがわかる．

9.3 ドロネー図

ドロネー図はボロノイ図から導くことができる図形のである．

9.3.1 ドロネー多角形

ドロネー図はボロノイ図において，ボロノイ領域がとなりあう母点同士をつなぐことで得られる図形だ．図 9.4 にその例を示す．

ドロネー図は母点の集合 S の内部を凸多角形で分割した図形である．内部の凸多角形をドロネー多角形といい．その多角形がすべて三角形である場合，ドロネー図をドロネー三角形分割という．

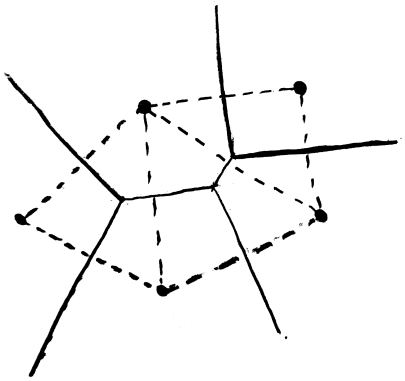

図 9.4　ドロネー図

9.3.2　ドロネー三角形分割の条件

ボロノイ図のときに空円性の話をしたが，さらに話を詳しくする．空円性では 3 つ以上の母点を通る円の中心がボロノイ点となり，その円の中には他の母点が存在しないと述べた．この 3 つ以上の母点というのを厳密に 3 つの母点であるときとそれ以外で区別する．

全てのボロノイ点と最寄りの母点による円が 3 つの母点を含むとき，母点集合 S は非退化であると表現する．それ以外は退化しているという．このような区別をするとドロネー三角形分割ができる条件は，母点集合 S が非退化であるとすることができる．そしてドロネー三角形は S のうち，異なる 3 つを選びできる三角形のうち，その外接円の内部に他の母点を含まないものということができる．

9.3.3　ドロネー辺

全域木を考える．これは頂点集合 S が与えられたとき，S のすべての要素を含む木のことをいう．ここでは辺の長さが最小となる全域木を与える方法を与える．一般的なアルゴリズム MINIMUM-SPANNING-TREE は以下で与えられる．

1. S のすべての頂点対の距離を計算し，ソートする．
2. 辺のないグラフ G を用意する．この G に辺を書き加えながら全域木の作成をする．
3. 頂点対のうち距離の最小のものから取り出し，その頂点対による辺を G に加えても G にサイクルができない場合は G に加える．
4. 辺が加えられなくなるまで繰り返す．

これは考えられる点対をすべて考えている．しかし，ここで与えられる全域木の辺は必ずドロネー辺になっているという性質がある．そこでアルゴリズムは次のように書き換えられる．

1. S のすべてのドロネー辺の距離を計算し，ソートする．
2. 辺のないグラフ G を用意する．この G に辺を書き加えながら全域木の作成をする．
3. ドロネー辺のうち距離の最小のものから取り出し，そのドロネー辺による辺を G に加えても G にサイクルができない場合は G に加える．

4. 辺が加えられなくなるまで繰り返す.

全ての点対のソートは点対の数が $O(n^2)$ であることから $O(n^2 \log n)$ となることがわかる．これに対してドロネー辺の作成の計算量は $O(n \log n)$ であり，ドロネー辺の数も $O(n)$ であることから，辺のソートも $O(n \log n)$ となる．計算量は少なくなる．

9.4 3 次元凸包とドロネー図

9.4.1 図形的な力技

これは原理に近い話だ．最後の計算量を確認して読み飛ばしてよい.

3 次元空間に与えられた頂点集合 S[*1]の凸包を $C(S)$ とする．この $C(S)$ を求めることを考える.

これは分割統治によって考えることができる．例えば x 座標によって頂点集合を S_1, S_2 の 2 等分にする．各々は $C(S_1), C(S_2)$ という凸包を求めることができる．この 2 つの凸包を合わせる作業が問題とある.

これは出来上がった 2 つの凸包に対して，その 2 つの凸包に接するように十分大きな平面をぶつけるという形で求めることができる．平面の角度次第では平面と 2 つの凸包の接点がともに頂点 1 つずつという状態になる．このとき，その 2 つの頂点を辺で結ぶ．この辺を最初の辺として e とする

e を軸に平面を回転させると，2 つの凸包のいずれかの頂点と平面が接触する．すると，平面には 3 つの頂点が含まれる状態となる．この三角形を橋渡し三角形と呼び，2 つの凸包を合わせる際に生まれる面である．そして新たに生まれた辺を e' としてまた平面を回転させる.

以上の操作を繰り返すと，2 つの凸包の間にいくつかの橋渡し三角形を生み，初めの e に平面は帰ってくる．生まれた橋渡し三角形で凸包を接続することで $C(S)$ を求めるに至る．2 つの凸包を合わせる作業の計算量は $O(n)$ である．ここから，3 次元凸包を求めるアルゴリズムの計算量は

$$f(n) = 2f\left(\frac{n}{2}\right) + cn \;\Rightarrow\; f(n) = O(n \log n)$$

となる.

9.4.2 ドロネー図の構成算法

平面上の点集合 $S = \{P_1, P_2, \cdots, P_n\}$ でドロネー図を構成することを考える．過程として S は非退化であるとしておく．S から 3 点をとり，その外接円の内部にほかの点がないことがドロネー三角形の条件である．そこで一般の座標 $P = (x, y)$ と 3 点 P_i, P_j, P_k によって F を与える.

$$F(P_i, P_j, P_k, P) = \begin{vmatrix} 1 & x_i & y_i & x_i^2 + y_i^2 \\ 1 & x_j & y_j & x_j^2 + y_j^2 \\ 1 & x_k & {}_k y & x_k^2 + y_k^2 \\ 1 & x & y & x^2 + y^2 \end{vmatrix}$$

[*1] S の条件として，S に含まれるどの 4 点も同一平面上にないものとする.

$f = 0$ は 3 点 P_i, P_j, P_k を通る円の方程式となっている．ここで 3 点 P_i, P_j, P_k を反時計回りでとったものと考えると，P_i, P_j, P_k がドロネー三角形となる条件は次のように与えられる．

$$F(P_i, P_j, P_k, P_l) > 0 \quad {}^\forall P_l \in S = \{P_i, P_j, P_k\}$$

これを三次元に広げる．座標を $P^* = (x, y, x^2 + y^2)$ とする．F は三次元に拡張すると以下の G で表現される．

$$F(P_i^*, P_j^*, P_k^*, P^*) = \begin{vmatrix} 1 & x_i & y_i & z_i \\ 1 & x_j & y_j & z_j \\ 1 & x_k & y_k & z_k \\ 1 & x & y & z \end{vmatrix}$$

$G = 0$ は 3 点 P_i^*, P_j^*, P_k^* を通る平面の方程式となる．そしてこれがドロネー三角形となる条件は

$$G(P_i^*, P_j^*, P_k^*, P_l^*) > 0 \quad {}^\forall P_l^* \in S = \{P_i^*, P_j^*, P_k^*\}$$

となる．

第 10 章

難問

10.1　解けない問題

理論的に解くことができない問題．答えが存在しない問題というものがある．

10.1.1　停止性判定問題

プログラム p とその p の引数の x とを引数に，p が停止するのか否かを判定する関数 halt(p, x) が存在するとする．

$$\text{halt}(p, x) = \begin{cases} 1, & p \text{ が引数 } x \text{ で停止する．} \\ 0, & \text{停止しない．} \end{cases}$$

結論としてこのような関数 halt は存在しない．この事実は以下の疑似プログラムを考えるとわかる．

```
A(string u){
    if (halt(A,u)==1)
```

```
            while(true);
        else
            reeturn 0;
    }
```

この関数の動ぎは必ず halt 関数の逆となる．つまり，矛盾が生じることになるのだ．

10.1.2 プログラムの等価性判定問題

2 つのプログラム p, q が等価であるか否かを判定する関数 equiv(p, q) を考える．これも実際には存在しない関数である．

これはあるプログラム p を部分的に書き換えたプログラム q を考えると答えられる．書き換え個所は

- p の引数を x に固定する．
- return 文はすべて return 0 とする．

そして r 関数をただ 0 を返すものとして定義する．ここで

```
hoge(p,x){
    if(equiv(q,r)==1)
        return 1;
    else
        return 0;
}
```

という関数を作ると，これは存在しないはずの halt 関数になる．もし元の p が x で停止する関数であれば，書き換え後の q は x の影響を全く受けることなくただ 0 を返す関数といえる．これは r と同じ関数であるといえる．

これを認めると equiv 関数により halt 関数が作ることになる．したがって equiv 関数は存在しない．

10.2 難しい問題

難しいとは多項式オーダーの計算量のアルゴリズムが存在しない問題をいう．

具体例としてハミルトン閉路問題を挙げる．頂点集合に対してすべての頂点を 1 回ずつ通る閉路が存在するのかを考える問題である．計算量は $n! > 2^n$ となることから指数オーダーである．似た問題でオイラー閉路問題[*1]があるがこちらの計算量は $O(n)$ である．

[*1] 全ての辺を通る経路が存在するかを問う問題.

10.3　問題のクラス

10.3.1　クラス P

問題の大きさが n の多項式オーダーのアルゴリズムが存在するとき，その問題をクラス P という．例えば要素数 n の配列が既にソートされているのかを確認するという問題は計算量が $O(n)$ であるのでクラス P である．

10.3.2　クラス NP

ここで考えるのは判定問題だ．そして計算する計算機は今まで想定していたものとは少し異なる．今まで暗に考えてきた計算機は計算の手順を 1 つ 1 つ実行するタイプのものであった．並列計算機であっても，並列の数だけの計算量の変化でありオーダーの変化はなかった．

ここで考える計算機は処理の分岐のとき，任意の枝を選択できる計算機になっている．つまり，最も都合の良い枝で計算を進めるのだ．このような計算機を非決定的計算機と呼ぶ．

クラス NP は非決定的計算機によって与えることができる．クラス NP とは判定問題において，その答えが YES の場合，非決定的計算機での計算時間が n の多項式時間であるものをいう．ハミルトン閉路問題はクラス NP に分類される．

非決定的計算機を並列計算機で表現すると，独立した並列計算機を無数に持っていると表現することができる．そして，無数の並列計算機のうちいずれか 1 つが YES との答えを出せば処理は終了し，その計算時間が n の多項式になるのだ．

10.3.3　クラス NP 完全

2 つのクラスのクラス P とクラス NP の包含関係に関して，いまのところクラス NP はクラス P を覆っていることになっている[*2]．しかし，依然としてクラス NP の問題で多項式時間のアルゴリズムが見つかっていないものも多く，実際のところはクラス NP とクラス P は異なるものと考えられている．このような状況において，NP 完全という概念が定義される．判定問題 Q がクラス NP 完全とは問題が以下の条件を満たすものをいう．

- Q はクラス NP に属する．
- Q がクラス P に属することを示せば，クラス NP に属するすべての問題がクラス P に属する．

この定義の示すところは，クラス NP の問題をすべて Q に帰着することができるということだ．そして，Q はクラス NP の中で最も複雑な問題であることも示している．このクラス NP 完全にはハミルトン閉路問題や，のちに紹介するクリーク問題が属している．

[*2] クラス NP に属していながら，クラス P に属さない問題が見つかっていない．これは，確認ができていないということであり，すべてクラス P に属していることを確認したわけではない．

10.3.4　クラス NP 困難

ここで考える問題は判定問題ではなく，一般の問題だ．クラス NP 困難とは問題の内部にクラス NP 完全の問題を含むものである．つまり，問題の解決の前提として必ずクラス NP 完全の問題を解くというものだ．

具体的な例として挙げられる問題は巡回セールスマン問題という問題だ．頂点を都市，辺を都市の間の移動距離に見立てて，最短ですべての都市を回る経路を考える問題だ．これは言い換えると全ての頂点 (都市) を通過する経路の中から最も最短の経路を選ぶということだ．すべての頂点を通過する経路というのがクラス NP 完全に属するハミルトン閉路問題になっている．

したがって，巡回セールスマン問題を解くということは，ハミルトン閉路問題を解くということを含んでいる．

10.4　問題の具体例

10.4.1　クリーク問題

$G(V,E)$ に対して，その部分グラフ $G'(S,E')$ を考える．G' のすべての頂点対が辺で結ばれているとき，S をクリークと表現する．図 10.1 はクリークの例だ．

クリーク問題とは，グラフ G と自然数 k が与えられたとき，G が頂点数 k のクリークを持つのかという問題である．

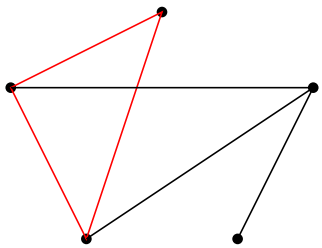

図 10.1　グラフ $G(V,E)$ におけるクリーク

10.4.2　頂点被覆問題

$G(V,E)$ に対して，部分頂点集合 $S \in V$ を考える．G のすべての辺において，その少なくとも一方の端点が S に含まれているとき，S を G の頂点被覆と呼ぶ．図 10.1 は集合の大きさが 2 の頂点被覆の例である．グラフは図 10.1 における辺の関係を反転させたグラフ $\bar{G}(V,\bar{E})$ である．

頂点被覆問題とはグラフ G と自然数 k が与えられたとき，G が集合の大きさ k の頂点被覆を持つのかという問題である．

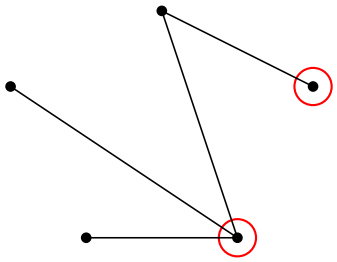

図 10.2　グラフ $\bar{G}(V,\bar{E})$ における頂点被覆

10.4.3　集合被覆問題

n 個の集合の族[*3] $S_1, S_2, \cdots, S_n$ から k 個の集合を取り出す．取り出した集合の要素によって，n 個の集合のすべての要素を網羅しているとき，取り出した k 個の集合を集合被覆という．

集合被覆問題とは，n 個の集合の族と自然数 k が与えられたとき，集合族の大きさが k の集合被覆が存在するかを判定する問題だ．

10.4.4　クラス NP 完全における問題の帰着

集合被覆問題において考える集合族を以下のように与える．

$$v_1 = \{1,2\},\quad v_2 = \{3\},\quad v_3 = \{4\},\quad v_4 = \{2,3,4\},\quad v_5 = \{1\}$$

これをグラフにおける頂点と辺で表現する (図 10.3)．すると，集合被覆問題は頂点被覆問題に置き換えることができる．図 10.3 から k が 2 以上であれば，集合被覆が存在することがわかる．

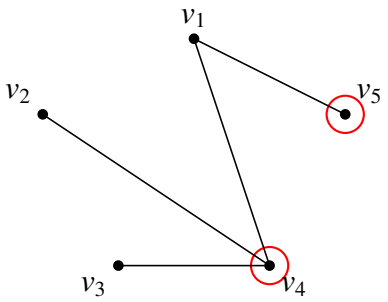

図 10.3　集合被覆問題を頂点被覆問題に帰着

頂点被覆問題はクリーク問題に帰着することができる．これは暗に示しているが図 10.3 の辺の有無を反転させると良い．頂点被覆問題とクリーク問題には以下の関係が成立する．

$$\{G,k\} \text{ のクリーク問題} = \{\bar{G}, |V| - k\} \text{ の頂点被覆問題}$$

クリーク問題はクラス NP 完全であることがわかっている．ここから，頂点被覆問題も集合被覆問題のクラス NP 完全であることがわかる．

[*3] 複数の集合の集まりを集合の族，集合族と呼ぶ．

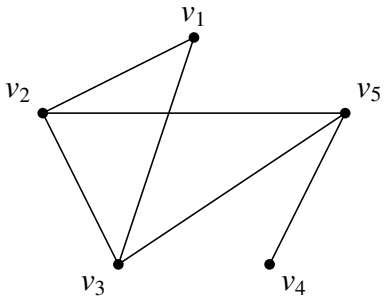

図 10.4　頂点被覆問題をクリーク問題に帰着

10.5　問題の緩和

10.5.1　難問となる条件の排除

難問の対処法の 1 つに問題の緩和という手法がある．巡回セールスマン問題を例にその手法を扱う．まずは巡回セールスマン問題についてのべる．この問題の目的は $\sum c(v_i, v_j)x_{ij}$ の最小化にある．$c(v_i, v_j)$ は頂点間のコストで非負整数であり，x_{ij} は頂点間の辺を通過するかを表している．

$$x_{ij} = \begin{cases} 1, & v_i, v_j\text{を通過する．} \\ 0, & v_i, v_j\text{を通過しない．} \end{cases}$$

この問題の条件は $x_{ij} = 1$ なる辺を全体がハミルトン閉路となることである．

この問題の難点はハミルトン閉路の導出が困難である点だ．これでは x_{ij} を与えるところで時間がかかってしまう．そこでとられる手法が問題の緩和だ．ここではハミルトン閉路の必要条件[*4]をハミルトン閉路にかわる x_{ij} の条件として問題を与える．

ハミルトン閉路は辺の数が n であるので x_{ij} の条件は以下のようになる．

$$x_{ij} = \begin{cases} 1 \\ 0 \end{cases}$$
$$\sum x_{ij} = n$$

10.5.2　分枝限定法

図 10.5 のグラフで巡回セールスマン問題を考える．緩和問題の解決は極めて簡単で，与えられた辺のうちコストが小さいものから $n = 5$ 個を選択すればいい．図 10.6 が解決の結果となる．

図 10.6 は明らかにハミルトン閉路ではない．ハミルトン閉路ではないのはコストが 1，2，3 の辺でループを形成しているからだ．つまり，求められる答えにおいて，ループをなす辺のうち，少なくとも 1 つは使用されないことがわかる．ループ解消を考えると次の 3 通りを考えるとすべてを尽くしたことになる．破線は使用しない辺を示し，青線は必ず通らなければならない辺を示している．

[*4] 問題の緩和で考えるのは，問題を難化させる条件の部分的な条件 (必要条件) である．

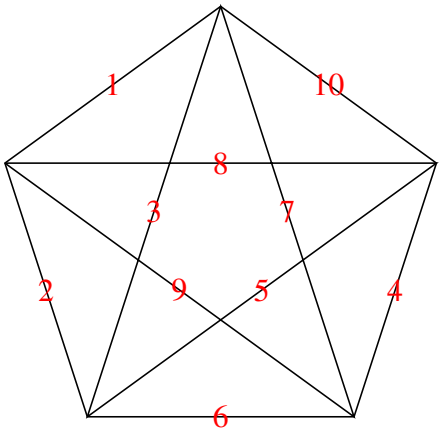

図 10.5　巡回セールスマン問題

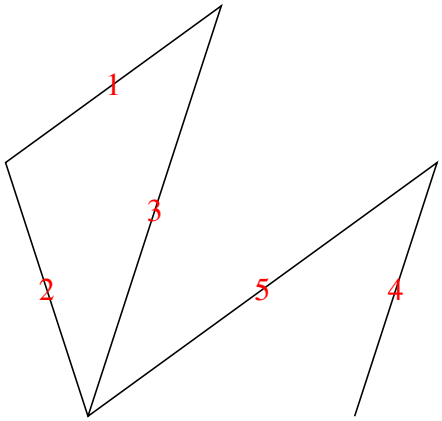

図 10.6　緩和問題の解決

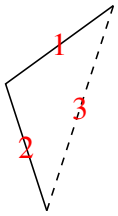

図 10.7　分枝 1

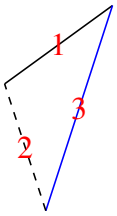

図 10.8　分枝 2

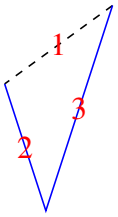

図 10.9　分枝 3

3 つの分枝から選ぶのは，まず分枝 1 だ．3 つの辺のうち最もコストが大きい辺を省いているので答えに近づくと考えられる[*5]．

すると新たに以下のグラフで緩和問題を考えることになる．すると，また答えはハミルトン閉路ではない．ループが生まれているのでそれを解消する分枝を考える．コスト 6 の辺を省く．

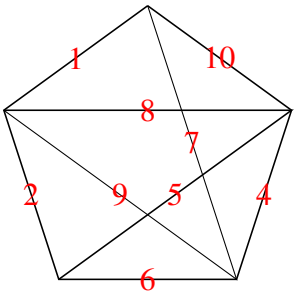

図 10.10　分枝の操作後の緩和問題

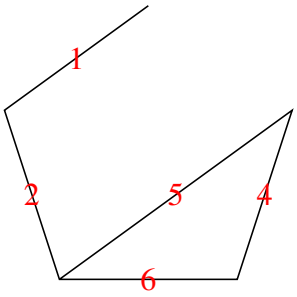

図 10.11　左図の解決

最終的には以下の答えが現れる．最小コストの近似解は 19 であるとわかる．あとは，一旦飛ばした他の分枝に対しても同様の操作をして近似解を求める．ただし，別の分枝を考えているときにコストが 19 を超えることがわかったら直ちにその分枝の以降の処理を中止し，別の分枝に移る．処理の順序は深さ優先探索と同じであり，最初に得られた近似解を基準に処理の進行を限定する．これが分枝限定法といわれる処理だ．そして

[*5] 正しい選択であるのかはわからない．あくまで選択の基準である．

この分枝限定法を最後まで実行すると 19 という解が得られる．

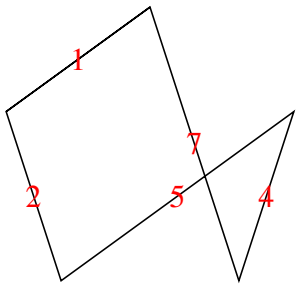

図 10.12　近似解

分枝限定法では深さ優先の処理により，とにかく 1 つの解を得るというのがスタートになる．あとは無視した他の分枝に戻り，時間が許す限りより良い近似解を探すのだ．

10.5.3　局所改良

近似解が得られたら，その近傍のさらに良い解が存在するかを考えることができる．これによりより良い解を得る手法を局所改良という．グラフにおける近傍とは，2 組の辺で結ばれた点対を取り出し，その頂点の接続の関係を入れ替えて得られるグラフのことだ．当然，連結グラフであることを維持するように入れ替える．

10.6　難問の利用：公開鍵暗号

10.6.1　ナップザック問題

ナップザック問題とは NP 困難に属する問題である．正の実数 $a_1, a_2, \cdots, a_n$ と実数 S によって与えられる．

$$\sum_{a_i \in X} \leq S$$

左辺が最大となる実数の集合 X を求めるのがその問題だ．

しかし，

$$\sum_{k=1}^{i-1} a_k < a_i$$

を満たすとき，計算量は $O(n)$ となる．この NP 困難の問題が条件次第で計算量 $O(n)$ となるところに暗号化のヒントがある．

10.6.2　ナップザック問題を利用した公開鍵暗号

公開鍵の作成手順を示す．数はすべて自然数と考える．

1. 十分に大きい n を用意する．
2. $\{a_n\}$ を以下の式が成立するように用意する．

$$\sum_{k=1}^{i-1} a_k < a_i$$

3. m を以下の数として用意する．

$$\sum a_n < m$$

4. m と互いに素な数 w を用意する．
5. 以下の式を成立させる v を用意する．

$$v \cdot w \equiv 1 \pmod{m}$$

6. $\{b_n\}$ を次のように用意する．

$$b_i \equiv w \cdot a_i \pmod{m}, 0 \leq b_i < m$$

7. $n, \{b_n\}$ を公開鍵，$m, v, \{a_n\}$ をプライベート鍵とする．

n は一度に暗号化できるビット数を表している．暗号化は以下のように暗号文 c を計算するだけである．

$$c = \sum x_i b_i$$

x_i は 0 または 1 をとるビットで，公開鍵の数列の線形和を計算する．

復号は受信された c は

$$S \equiv v \cdot c \pmod{m}$$

なる S に対して $\{a_n\}$ とのナップザック問題を解くことでビット列を得ることができる．

10.6.3 運用

実際にナップザック問題を利用した公開鍵暗号を運用する．$n = 4$ とする．

$$\{a_n\} = \{2, 5, 10, 20\}, \quad m = 38, \quad w = 13$$

とする．すると

$$3 \cdot 13 \equiv 1 \pmod{m = 38}$$

より $v = 3$ が決定する．これから

$$\{b_n\} = \{26, 27, 16, 32\}$$

を得る．

$(1, 0, 1, 0)$ を暗号化することを考える．$c = 42$ は直ちに得られる．さて，この $c = 42$ と $\{b_n\}$ によるナップザック問題を考えると，これは NP 困難の問題であるがゆえに極めて多くの時間を要する[*6]

復号は $3 \cdot 42 \equiv 12 \pmod{38}$ と $\{a_n\}$ より

$$12 = 1a_1 + 0a_2 + 1a_2 + 0a_3$$

が計算時間 $O(n)$ で計算され元のビット列 $(1, 0, 1, 0)$ を得る．

[*6] n が大きければという話である．