修 士 論 文

作業領域調節可能アルゴリズムに関する研究

北陸先端科学技術大学院大学
情報科学研究科情報科学専攻

清井　孝裕

2014 年 3 月

修 士 論 文

作業領域調節可能アルゴリズムに関する研究

指導教員　浅野哲夫 教授

審査委員主査　浅野哲夫 教授
審査委員　上原隆平 教授
審査委員　緒方和博 准教授

北陸先端科学技術大学院大学
情報科学研究科情報科学専攻

1110033 清井　孝裕

提出年月: 2014 年 2 月

概 要

本稿では, 具体的な２つの問題に対して作業領域調節可能アルゴリズムを示す．１つは３-SUM 問題の基本問題であり，もう１つは重み付き点集合の最適分割問題である．どちらも $O(n)$ の作業領域を使えれば，効率よく解くことができるアルゴリズムは知られているが，作業領域が調節可能であるようなアルゴリズムは知られていない．

3-SUM 問題は関連する問題が数多くあり，本稿で紹介する方法で多くの問題に対し作業領域調節可能アルゴリズムを設計できることが期待できる．

重み付き点集合の最適分割問題では，直線のアレンジメントを走査することで問題を解く．直線のアレンジメントとは, 平面を多数の直線により区切られた構造のことをいう．アレンジメントを走査するアルゴリズムとしてトポロジカルスィープやトポロジカルウォークが知られている．しかし, これらのアルゴリズムは $O(n)$ の作業領域を必要とする．本稿では, 作業領域のサイズを表すパラメータ s が $\lg n$ から $O(n/\lg n)$ の間で指定されたとき, $O(n^3/s)$ の時間でセルを隣から隣へと訪問し, アレンジメント全体を走査するシンプルなアルゴリズムを与える．トポロジカルスィープでは，直線のアレンジメントのセルを隣から隣へと訪問することで, 多くの問題を効率よく解いているため，提案アルゴリズムを使用することで同様の問題に対し作業領域調節可能アルゴリズムを示すことが期待できる．

目 次

第1章　はじめに

1.1　研究の背景

集積回路技術の急速な進歩により，数年前の一般的なデスクトップPCのメモリ容量が片手で持てる大きさの端末に搭載されている．それに伴い，ソフトウェアがハードウェアに対する性能の要求も大きくなっている．そのため，メモリ容量が増えた端末上であってもプログラム作成にはメモリ容量の制約を考慮に入れなければならないことがある．例の１つとしてビッグデータの活用がある．ビッグデータに対して様々な処理を行いたい場合，入力となるビックデータの内容を書き換えることは他の処理に影響が出るためできない．もちろん，データに対し使用できるメモリ容量が小さいため，限られたメモリ容量で作業を行う必要がある．

また，近年ではデジタルカメラやタブレット等の小型端末上で画像処理や目的場所までの経路を求める処理が当然のようにできるが，高度な処理をするためにメモリ容量を増やすと，端末のサイズが大きくなり，端末のコストも上がる．

もし，処理に必要な作業領域を調節できるアルゴリズムがあれば，設計時やプログラム実行時のメモリの空き状況に柔軟に対応できる．

メモリ容量が限られた環境での効率の良いアルゴリズムは，組込み環境だけでなく，汎用的な環境でも嬉しいことである．

1.2　研究の目的

本研究では，２つの問題に対し作業領域調節可能アルゴリズムを設計する．１つは入力データに対しソートを行うことで効率よく解くことが知られている3-SUM問題の基本問題である．作業領域が限られた環境で既存のアルゴリズム同様の動きを作業領域調節可能アルゴリズムでシミュレートすることを目的の１つとする．これにより，他の3-SUM問題に対して作業領域調節可能アルゴリズムを設計できることが期待できる．

もう一つはトポロジカルスィープで効率よく解ける多くの問題の１である重み付き点集合の最適分割問題である．これは，平面上の点集合を直線で２つに分割し，分割された点集合の重みの総和の差を最小にする直線を見つける問題である．この問題に対し既存アルゴリズムは直線のアレンジメントを走査することで問題を解いている．直線のアレンジメントとは，多数の直線によって平面が区切られた構造のことを言う．既存のアルゴリズムとして，トポロジカルスィープ[1]やトポロジカルウォーク[2]が知られているが，これら

のアルゴリズムは，$O(n)$ の作業領域を用いて $O(n^2)$ 時間で直線のアレンジメント全体を走査する．しかし，$o(n)$ の作業領域で設計されている効率の良いアルゴリズムは知られていない．直線のアレンジメントを走査する際に作業領域が調節可能なアルゴリズムを設計することが目的である．トポロジカルスィープと同様の方法で，数多くの問題が効率よく解かれている．提案アルゴリズムでトポロジカルスィープで効率よく解ける問題に対しての作業領域調節可能アルゴリズムが設計できることが期待できる．

1.3 本論文の流れ

本稿では，2 章で後の章で述べるアルゴリズムで使用する重要なデータ構造について述べる．

3 章では 3-SUM 問題の基本問題について定義を行い，その問題に対する既存アルゴリズムと作業領域調節可能アルゴリズムについて述べる．

4 章では重み付き点集合の最適分割問題について定義を行い，その問題を解く上で重要な双対変換について述べる．その後に，直線のアレンジメントを走査する作業領域調節可能アルゴリズムにより，この問題を解く．

第2章　作業領域調節可能優先度付きキュー

ここでは，3章，4章で説明する問題に対してのアルゴリズムの重要なデータ構造について説明する．そのデータ構造とは，文献 [3] の作業領域が調節可能である優先度付きキューである．ここでは，作業領域調節可能優先度付きキューのことを，単にキューと呼ぶことにする．また，ここでは要素の値が小さいものから取り出すキューとして説明を行うが，その逆も同様にできる．

これを使用することで，作業領域が限られた環境であっても n 個の読み出し専用配列の要素を $O(s)$ の作業領域を用いることで，$O(n^2/s)$ 時間でソートされた結果を出力することができる．ただし，$\lg n \leq s \leq n/\lg n$ である．

2.1 節でデータ構造を説明し，2.2 節でキューの重要な操作である要素の挿入操作について説明し,2.3 節で要素を取り出す操作についてそれぞれ説明する．

2.1　データ構造

キューのデータ構造は二分木のような構造を持つ．ここで説明する二分木はノードごとに番号がついているとし，ルートノードを１とする．ルートノードから，幅優先探索によりノードを訪問した順位番号がついているものとする．例えば，ルートノードの子ノードの番号は２と３，ノード２の子ノードは４と５であるようにつける．また，ノードの高さはルートノードを $\lg s$ とし，葉の部分を高さ１とする．キューのデータ構造を図 2.1 に示す．入力データの数を n とし，作業領域を表すパラメータを s としたとき，配列の要素を $\lceil n/s \rceil$ 個ごとに区切ったものをバケットと呼ぶ．キューから取り出し操作を行う際に，取り出す要素がどのバケットのどのあたりにあるかを二分木のデータ構造を使いキューの状態を管理する．

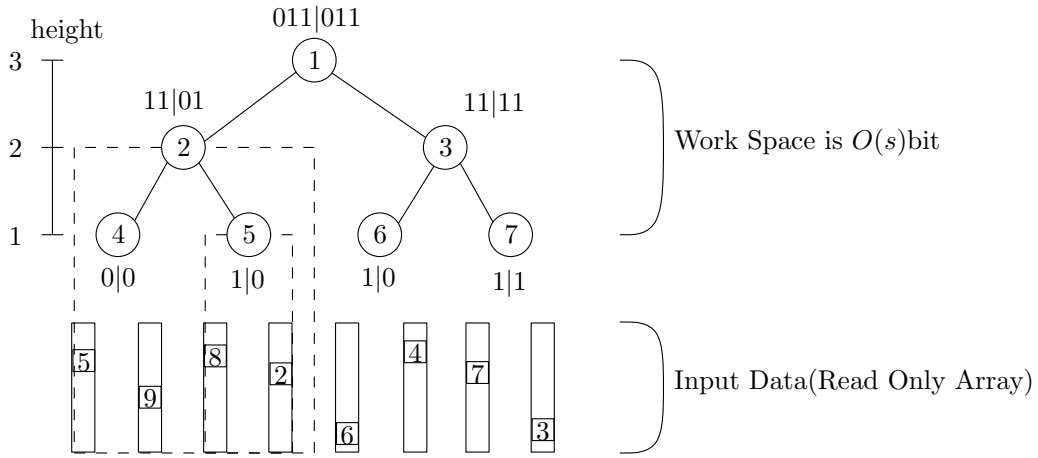


図 2.1: ノードの情報

各ノードはバケットの情報（バケットアドレスと呼ぶ）と，右側はバケット内でどの範囲に取り出しの候補となる要素があるかを示す情報を表している (クォンタイルアドレスと呼ぶ)．各ノードが使う作業領域はノードの高さ $h \times 2$bit である．例えば，高さ１の各ノードが使う作業領域はバケットアドレスとクォンタイルアドレス共に１ビットずつ使用し合計２ビットを各ノードで使用する．高さ２のノードは４ビットずつであるようにノードの高さに応じて作業領域を変えている．

バケットアドレスは，取り出し候補の要素がどのバケットにあるかを表しているもである．また，ノードごとに管理しているバケットが異なる．ノードの高さを h として，バケットに対し 1 から s のインデックス番号が付いているものとすると，ノード i は $(2^h i \mod s) + 1$ から $(2^h i \mod s) + 2^h$ までのバケットを管理している．

例えば，$s = 8$ のときノード 4 の高さは $h = 1$ である．ノード 4 はバケット 1 から 2 までを管理している．つまり，ノード 4 にバケット 1 から 2 に関してのバケットアドレスとクォンタイルアドレスのみの情報が管理されている．ノード 5 はバケット 3 から 4 までをノード 2 はバケット 1 から 4 までを管理しているといったように，親ノードは子ノードが管理しているバケットを管理しているようになっている．

管理しているバケットのインデックスが小さい方から再度インデックス番号を０から割り当て直すと，各ノードのバケットアドレスは割り当て直したインデックス番号に対応する．例えば，ノード５の管理しているバケットは３と４だが，これを０と１のように割り当て直す．ノード５はバケットアドレスを保存できる情報量は１ビットである．０なら３のバケット１なら４のバケットというようにそれぞれのバケットを表す．ノード２なら，バケットの１から４までを管理しているが，これを００から１１のように割り当てる．ノード２のバケットアドレスが保存できる情報量は２ビットである．００ならバケット１を１０ならバケット３を表している．各ノードに保存されているバケットアドレスから１から s のインデックスが割り当てられたバケットを求めるには以下のように求める．ノードの番号を i，ノードの高さを h，作業領域を s，ノードに保存されているバケットの

値をbとするとバケットのインデックス番号は

$$(2^h i \mod s) + b \tag{2.1}$$

である．

クォンタイルアドレスはビット数に応じてその値が示すバケット内のデータ範囲が変化する (図 2.2)．各ノードのクォンタイルアドレスの保持できるビット数はノードの高さの値と同じである．クォンタイルアドレスはその値を保持しているノードの高さをhとすると，2^hでバケットを分割されたものを左から番号付けした範囲を表してる．例えば，高さ１のノードのクォンタイルアドレスは１ビットで表されている．そのクォンタイルアドレスはバケット内を二等分し，０なら左側，１なら右側の範囲に取り出しの候補となる要素があることを表している．高さが２のノードのクォンタイルアドレスは２ビットで表されているためバケット内を四等分し，００なら左から１番目の範囲，０１なら左から２番目の範囲を示すようになる．

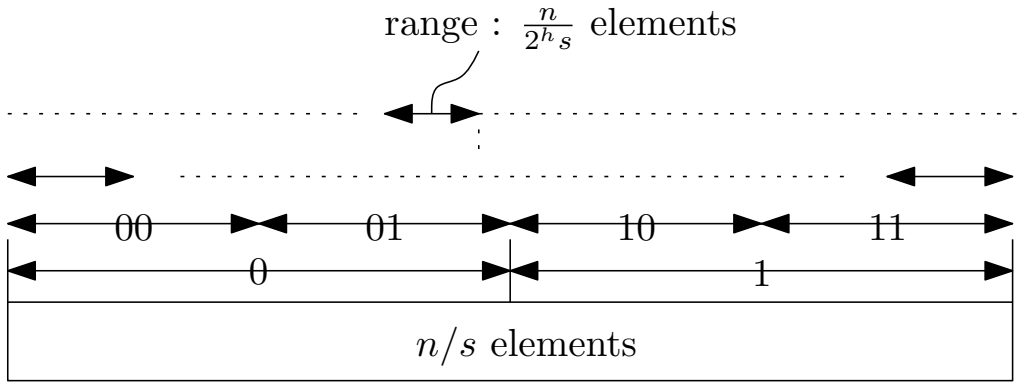


図 2.2: クォンタイルアドレスが示す目的データの範囲．ノードの高さをｈとした時のそのノードのクォンタイルアドレス指す範囲は$n/2^h s$個の要素数

バケットアドレスやクォンタイルアドレスがどのようなものであり，どのようなデータ構造であるかは説明した．各ノードが保持している情報は，そのノードが管理しているバケットの中で次の取り出し候補となる最小の要素があるバケットアドレスとそのバケットの中のどの範囲にその要素があるかのクォンタイルアドレスである．このデータ構造を表現するために必要な作業領域は全体で$O(s)$ビットに収まる (2.2) 式．高さｋのノードのバケットアドレスとクォンタイルアドレスの情報を保持するための作業領域をW_kとすると

作業領域 W_{all} は

$$
\begin{aligned}
W_k &= k\text{bits} \times 2 \times \frac{s}{2^k}\text{nodes} \\
W_{all} &= \sum_{k=1}^{\lg s} W_k \\
&= W_1 + W_2 + \cdots + W_{\lg s} \\
&= 2 \times \frac{s}{2} + 4 \times \frac{s}{4} + \cdots + 2\lg s \times \frac{s}{2^{\lg s}} \\
&= 2s \sum_{k=1}^{\lg s} \frac{k}{2^k} \\
&< 2s \sum_{k=1}^{\infty} \frac{k}{2^k} \\
&= 2s \times \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} \\
&= 4s \\
&= O(s)
\end{aligned}
\tag{2.2}
$$

である．

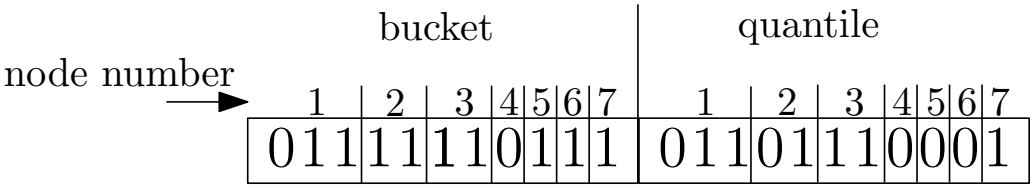


図 2.3: 木全体をビット列で構成した図．左側のビット列は各ノードのバケットアドレス，右は各ノードのクォンタイルアドレス．ビット列の上の数字はノードの番号を表している．

2.2 要素の挿入操作

2.1 節で説明したデータ構造を構築するため，要素すべてをキューに挿入する操作が必要になる．挿入といっても作業領域に要素を追加するわけではなく，入力配列のすべての要素から説明したデータ構造を作ることを行う．

まず，一番下のノードを作る．一番下の各ノードは 2 つのバケットを管理している．各ノードが管理しているバケットの中で最小の要素を見つけノードにバケットアドレスとクォンタイルアドレスを登録する．高さが 2 の各ノードは子ノードから管理している 4 つのバケットから最小の要素を見つけバケットアドレスとクォンタイルアドレスを登録す

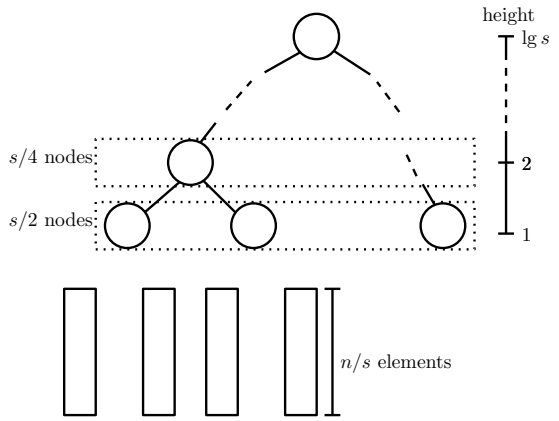
る．このとき，子ノードにはすでに管理しているバケットの中で最小の要素があるバケットとクォンタイルアドレスが登録されているので，その情報を使うと $n/2s \times 2$ 個の要素探すことで４つのバケットの中で最小の要素を見つけることができる．高さが３以降の各ノードも同様に子ノードにすでに登録されているアドレス情報から管理しているバケットの中で最小の要素を見つけ，その要素のバケットアドレスとクォンタイルアドレスを登録する．挿入操作の処理をアルゴリズム 1 に示す．

アルゴリズム 1 挿入操作

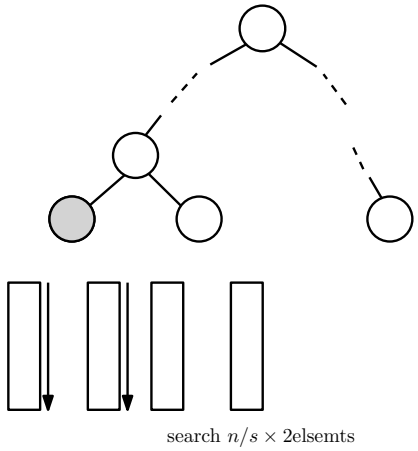
Input: n 個の整数の配列 $a[1..n]$ と作業領域のパラメータ s.
Output: 挿入結果の 2 分木

```
for h = 1 to lg s do
  for ノード i = s/2^h to s/2^{h-1} − 1 do
    if 子ノードなし then
      ノード i の管理しているバケットから最小の要素を見つける.
    else
      ノード i の２つの子ノードの情報から管理してるバケットの中で最小の要素を見つける.
    end if
    見つけた要素のバケットとクォンタイルアドレスをノード i に登録する.
  end for
end for
```

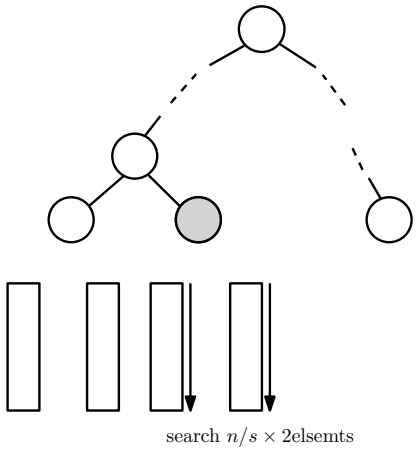
最も下のノードは $s/2$ 個あり，２つのバケットの要素すべてを調べてから登録するため，$T_1 = n/s \times 2 \times s/2$ の時間がかかる．高さが２のノードは $s/4$ 個あり，子の情報を利用することで要素の探索範囲を狭めるため，$T_2 = n/2s \times 2 \times s/4 + T_1$ の時間で高さ２までのノードに登録ができる．すべてのノードにアドレス情報を登録する時間は (2.3) 式により，$O(n)$ 時間で可能である．



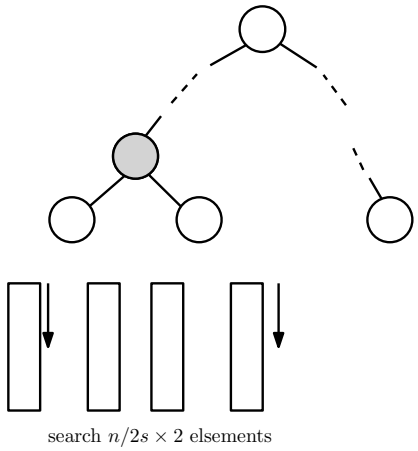
(a) 挿入後の 2 分木の外観．1 つのバケットあたり n/s 個の要素がある．



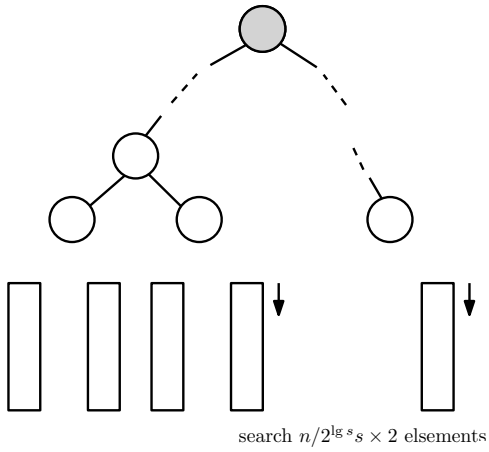
(b) 高さ 1 のノードは管理しているバケットから最小の要素を探し登録する．



(c) 各ノード同様に最小を要素を探し登録する．



(d) 高さ 2 の各ノードはすでに登録されている子ノードの情報を使い最小要素を探し登録する．



(e) 一番上のノードまで繰り返す．

図 2.4: 要素の挿入操作

高さ 1 のノードの更新にかかる時間 $T_1 = n$

高さ k のノードの更新にかかる時間 $T_k = T_{k-1} + \frac{n}{2^{k-1}s} \times 2 \times \frac{n}{2^k}$

$$
\begin{aligned}
T_{\lg s} &= \frac{n}{s} \times 2\frac{s}{2} + \frac{n}{2s} \times 2\frac{s}{4} + \cdots + \frac{n}{2^{\lg s - 1}s} \times 2\frac{s}{2^{\lg s}} \\
&= n + \frac{n}{4} + \frac{n}{8} + \cdots + \frac{n}{4^{\lg s - 1}} \\
&< n + n\sum_{k=0}^{\infty} \frac{1}{2^{2k-1}} \\
&= n + \frac{2}{3}n \\
&= O(n) \qquad (2.3)
\end{aligned}
$$

2.3 要素の取り出し操作

要素の取り出し操作について説明する．2 分木の各ノードはそのノードが管理しているバケットの中で次の取り出し候補になる要素が入っているバケットを指している．また，そのバケット内でどの範囲にデータがあるかを示すクォンタイルアドレスを保持していることは 2.1 節で説明した．

取り出し操作では，挿入操作のような対応するバケットを調べ，対応するノード順に更新する操作をする．まず，取り出した要素のがどのバケットにあったものかを求める．そのバケットを管理している一番下のノードを更新する必要がある．一番下のノードは 2 つのバケットを管理している．そのバケットの中から次の要素の候補を探す．要素の場所からノードの情報を更新する．

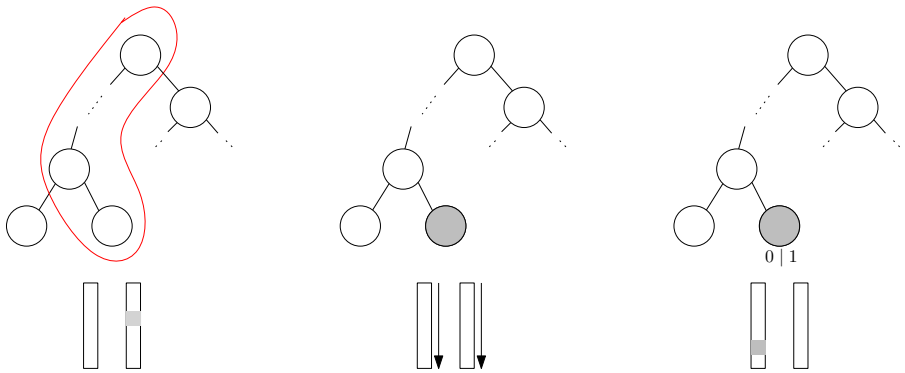
次にそのノードの親ノードを更新する．親ノードは子ノードが持っている情報から次の要素の候補を探す．今，更新された子側で管理されているバケットについては先程の更新でどの要素が次の候補かわかっている．更新されていない子のノードにはその子ノードが管理しているバケットの中での次の候補となる範囲がわかる．その範囲から候補となる要素を選び，更新された子の要素と比較し，ノードを更新する．

そして．次の親へと同じ更新を順に行っていく．取り出し操作の処理をアルゴリズム 2 に示す．

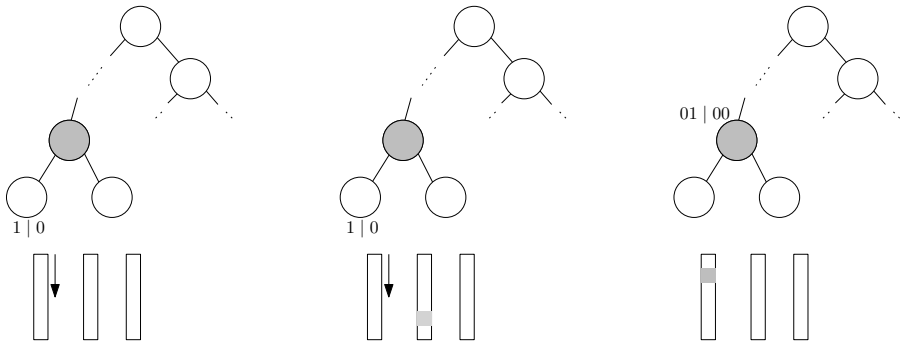
アルゴリズム 2 取り出し操作

取り出した要素の場所からバケットを求め，そのバケットを管理している高さ 1 のノードを求める．そのノードを ノード i とする．ノード i が管理している 2 つのバケットから取り出した要素の値以上で最小の要素を見つける．その要素を $min_{current}$ とする．
見つけた要素のバケットアドレスとクォンタイルアドレスをノード i に登録する
while $i > 1$ **do**
  $i = \lfloor i/2 \rfloor$
  ノード i の更新済みでない子ノードの情報から取り出した要素の値以上で最小の要素を見つける．その要素を min_{search} とする．
  if $min_{search} < min_{current}$ **then**
    $min_{current} = min_{search}$
  end if
  ノード i に $min_{current}$ のバケットアドレスとクォンタイルアドレスを登録する．
end while

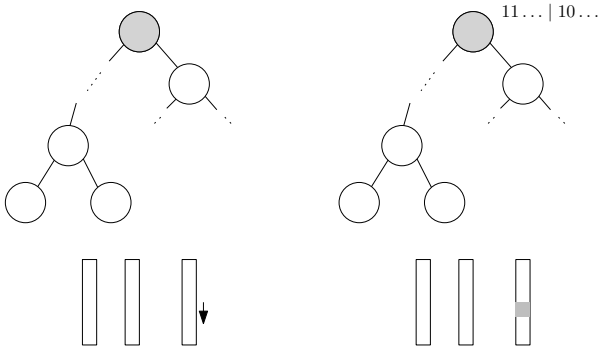
一番下のノードの更新には 2 つのバケットの要素すべてを調べるため $2n/s$ 個の要素を調べる．その親は，一方の子の情報から 1 つのバケットの要素の半分を調べるため $n/(2s)$ 個の要素調べる．また，その親は 1 つのバケットの要素の半分を調べるため $n/(4s)$ 個の要素調べる．と，順にノードを更新するために調べる要素の数は減っていく．更新全体にかかる時間は (2.4) 式から，$O(n/s)$ となる．これが，取り出し操作にかかる時間である．



(a) 更新の対象となるノード (b) 高さが 1 のノードが管理しているバケットから次の要素を探す (c) 要素の場所に応じたアドレスを更新する



(d) 高さ 2 のノードはもう一方の子ノードの情報から次の要素を探す (e) 2 つの子ノードの探した要素を比較する (f) より小さい要素のアドレスでノードを更新する



(g) 繰り返しそれを一番上のノードまで繰り返す (h) 一番上のノードの更新が済めば，要素の取り出し操作が完了する

図 2.5: 要素の取り出し操作

$$
\begin{aligned}
\text{高さ}1\text{のノードの更新時間}\, T_1 &= \frac{n}{s} \times 2 \\
\text{高さ}\, k\, \text{のノードまでの更新時間}\, T_k &= T_{k-1} + n/2^{k-1}s \\
T_{\lg s} &= \frac{n}{s} \times 2 + \frac{n}{2s} + \frac{n}{4s} + \cdots + \frac{n}{2^{\lg s - 1}s} \\
&< \frac{n}{s} \sum_{k=0}^{\infty} \frac{1}{2^k} \\
&= \frac{2n}{s} \\
&= O\left(\frac{n}{s}\right) \qquad (2.4)
\end{aligned}
$$

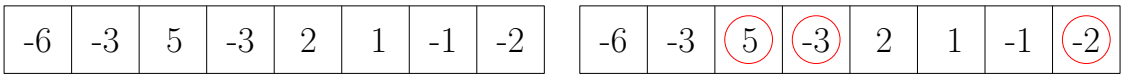
第3章　3-SUM問題

3.1　問題の定義

n 個の整数の集合 S が与えられた時 $a+b+c=0$ となる 3 要素 $a,b,c \in S$ は存在するかどうかを求める，例えば，S の要素がすべて正やすべて負だった場合は，$a+b+c=0$ となる 3 要素は存在しない．もし，$S=-6,-3,5,-3,2,1,-1,-2$ のような整数の集合が与えられた場合は，$a=5,b=-3,c=-2$ と選ぶことで，$a+b+c=0$ となる 3 要素が存在する．

この問題を正式に定義すると次のようになる．

問題 1 (3-SUM 問題) $a[1..n]$ に蓄えられた n 個の整数からなる集合 S が与えられたとき，$a[i]+a[j]+a[k]=0$ となる 3 要素 $a[i],a[j],a[k]$ が存在するか？



(a) 入力　　(b) 出力：True ($5-3-2=0$ となるため)

図 3.1: 3-SUM 問題の例

3.2　既存アルゴリズム

この問題に対する最も効率の良いアルゴリズムとしてアルゴリズム 3 が知られている．このアルゴリズムは，n 個の整数に対して定義された 3-SUM 問題を $O(n^2)$ 時間と $O(n)$ の作業領域で解く．

この問題は 3-SUM 問題クラスの基本問題として知られており，関連した問題は文献 [5] に紹介されている．

アルゴリズム 3 3-SUM 問題に対する $O(n^2)$ 時間のアルゴリズム

Input: n 個の整数の配列 $a[1 \ldots n]$.

Output: $a[i] + a[j] + a[k] = 0$ であるような 3 要素が存在すれば，**true** を返し，そうでなければ **false** を返す.

```
与えられた n 個の整数を昇順にソートする.
//a[1] ≤ a[2] ≤ ··· ≤ a[n] と仮定する.
i = 1, j = 2, k = n
repeat
  if a[i] + a[j] + a[k] = 0 then
    return true
  end if
  if a[i] + a[j] + a[k] > 0 then
    k = k − 1
  else
    //a[i] + a[j] + a[k] < 0
    j = j + 1
  end if
  if j ≥ k then
    i = i + 1; j = i + 1; k = n
  end if
until i > n − 2
return false
```

3.3 作業領域調節可能アルゴリズム

本論文では作業領域を減らすことに関心がある．この 3-SUM 問題に対して作業領域調節可能アルゴリズムを設計できるかどうかを考える．入力で与えられるデータは読み込み専用配列に入っていて，$O(s)$ の作業領域が利用できるとき，どれだけ早く問題を解けるか考える．この問題を正式に定義すると次のようになる.

問題 2 (メモリ制約付 3-SUM 問題) 読み出し専用配列 $a[1..n]$ に蓄えられた n 個の整数からなる集合 S と $O(s)$ の作業領域が与えられたとき，$a[i] + a[j] + a[k] = 0$ となる 3 要素 $a[i], a[j], a[k]$ が存在するか？

アルゴリズム 3 では，入力データをソートしたものを使用することで，効率的に解いている．しかし，この問題では入力データは読み出し専用配列に蓄えられている．ソートされた配列を使用するには $O(n)$ の作業領域が必要となる．$O(n)$ の作業領域を使用できるのであれば，アルゴリズム 3 と同じアルゴリズムで解ける．$o(n)$ の作業領域でアルゴリズム

3 を実行するには少ない作業領域だけで配列要素をソート順に 1 つずつ取り出す必要がある．2 章で説明したデータ構造を使用することで配列要素からソート順に 1 つずつ取り出すことができる．

2 章のキューを用いると，$O(s)$ の作業領域だけでアルゴリズム 3 をシミュレートできる．

アルゴリズム 4 3-SUM 問題に対するメモリ調節可能アルゴリズム

Input: n 個の整数からなる読み出し専用配列 $a[1 \ldots n]$ とサイズ s の作業領域．

Output: $a[i] + a[j] + a[k] = 0$ であるような 3 要素が存在すれば，**true** を返し，そうでなければ **false** を返す．

3 つの優先順位付キューを n 個全ての整数を入れた状態で初期化する．そのうち，2 つは最小値を取り出すタイプで，残る 1 つは最大値を取り出すタイプである．

a=Extract-Min1()

優先順位付きキュー 1 の内容を優先順位付キュー 2 にコピーする．

b=Extract-Min2()

c=Extract-Max()

repeat

  if $a + b + c = 0$ **then**

    return true

  end if

  if $a + b + c > 0$ **then**

    c=Extract-Max()

  else

    b=Extract-Min2()

  end if

  if $b \geq c$ **then**

    a=Extract-Min1()

    優先順位付キュー 1 をキュー 2 にコピーする

    b=Extract-Min2()

    優先順位付キュー 3 を初期化

    c=Extract-Max()

  end if

until a, b, c のどれかが定義されない

return false

このアルゴリズムでは，1 回の取り出し操作は $O(n/s)$ 時間かかり，比較は $O(n^2)$ 回行われるため，全体の計算時間は $O(n^3/s)$ となる．

第4章　重み付き点集合の最適分割問題

4.1　問題の定義

入力として，平面上に重み付きの点集合 $S = \{p_1, p_2, \ldots, p_n\}$ が読み出し専用メモリに与えられる．点 p_i には重み w_i がある．どの点も通らない直線 ℓ により平面を 2 つに分割するとき，直線 ℓ より上に位置する点集合を $S_A(\ell)$，下に位置する点集合を $S_B(\ell)$ とする．$S_A(\ell)$ と $S_B(\ell)$ の重みの総和の差が最小になる直線 ℓ を求める (図 4.1)．定式化すると (4.1) 式になる．

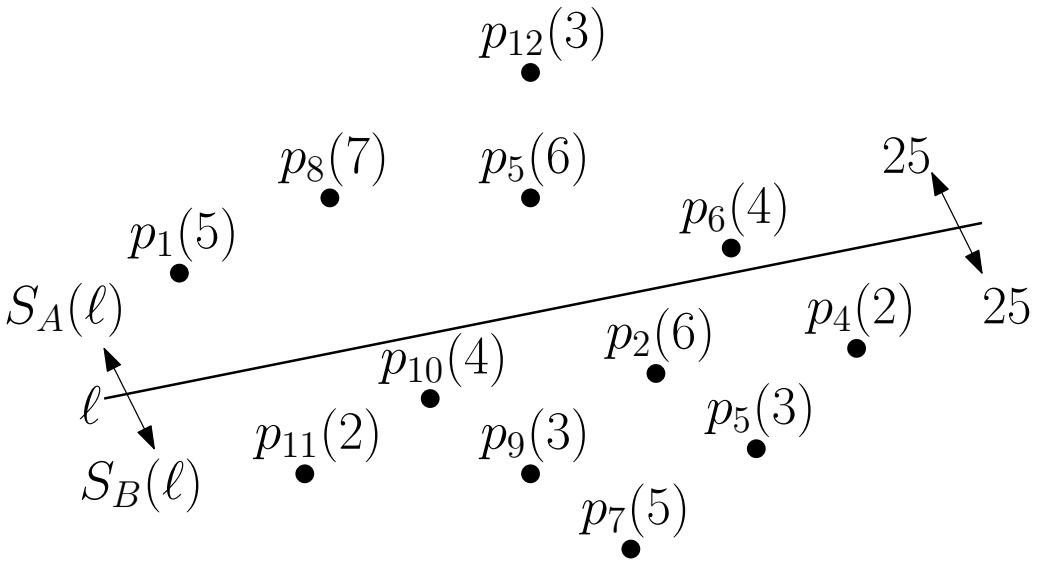


図 4.1: 問題と解の例．両側のそれぞれの重みの総和が 25．点 p_i の括弧内の数は重みを表している．

$$\ell_{min} = \arg\min_{\ell} \left(\left| \sum_{p_i \in S_A(\ell)} w_i - \sum_{p_j \in S_B(\ell)} w_j \right| \right). \tag{4.1}$$

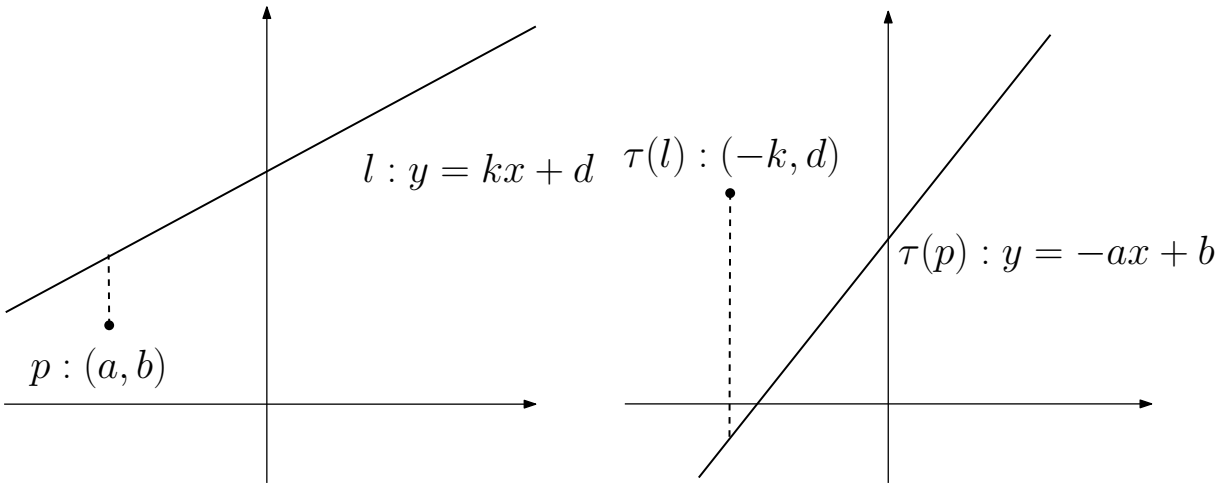
4.2 既存アルゴリズム

単純な方法として，全ての直線について点集合の重みの総和の差を求める方法を思いつくが，直線は無限通りあるためこの方法で解くことはできない．また，ランダムに直線を引くのであれば，直線は違うが分割のパターンが同じになるような直線を何度も調べてしまう．そして，調べていない分割のパターンが総和の差が最小になる直線の可能性もある．この問題では，どのような分割が最適であるかを見つけることに興味がある．異なる直線の方程式でも同じ分割を与える直線を同値として扱うことにすると，全ての分割パターンを調べることで，この問題を解くことができる．もし，具体的な直線の方程式を求めたいのであれば，最適な分割を見つけたあとに計算すればよい．分割パターンを調べる際に点集合に対し双対変換を行う．既存のアルゴリズムは，点集合を双対変換 [4] してできた直線のアレンジメントに対してトポロジカルスィープ [1] やトポロジカルウォーク [2] を行うことで，この問題を解くことができる．トポロジカルスィープやトポロジカルウォークも $O(n)$ の作業領域を使い $O(n^2)$ 時間で解くことができるこの節では，双対変換を説明し，トポロジカルスィープでこの問題を解く方法を説明する．

点集合のままでは見通しが悪いので，双対変換を行う．双対変換とは，点を線に，線を点に写像する変換である（(4.2) 式，(4.3) 式）．また，双対変換は変換後も点と線の上下関係を維持する性質がある（図 4.2）．２つの変換式から二度変換すると元に戻る．この性質を利用することで，点集合のすべての分割パターンを調べることができる．

$$\text{点}\, p : (a, b) \quad \rightarrow \quad \text{直線}\tau(p) : y = -ax + b \tag{4.2}$$

$$\text{直線}\, \ell : y = -kx + d \quad \rightarrow \quad \text{点}\tau(\ell) : (-k, d) \tag{4.3}$$

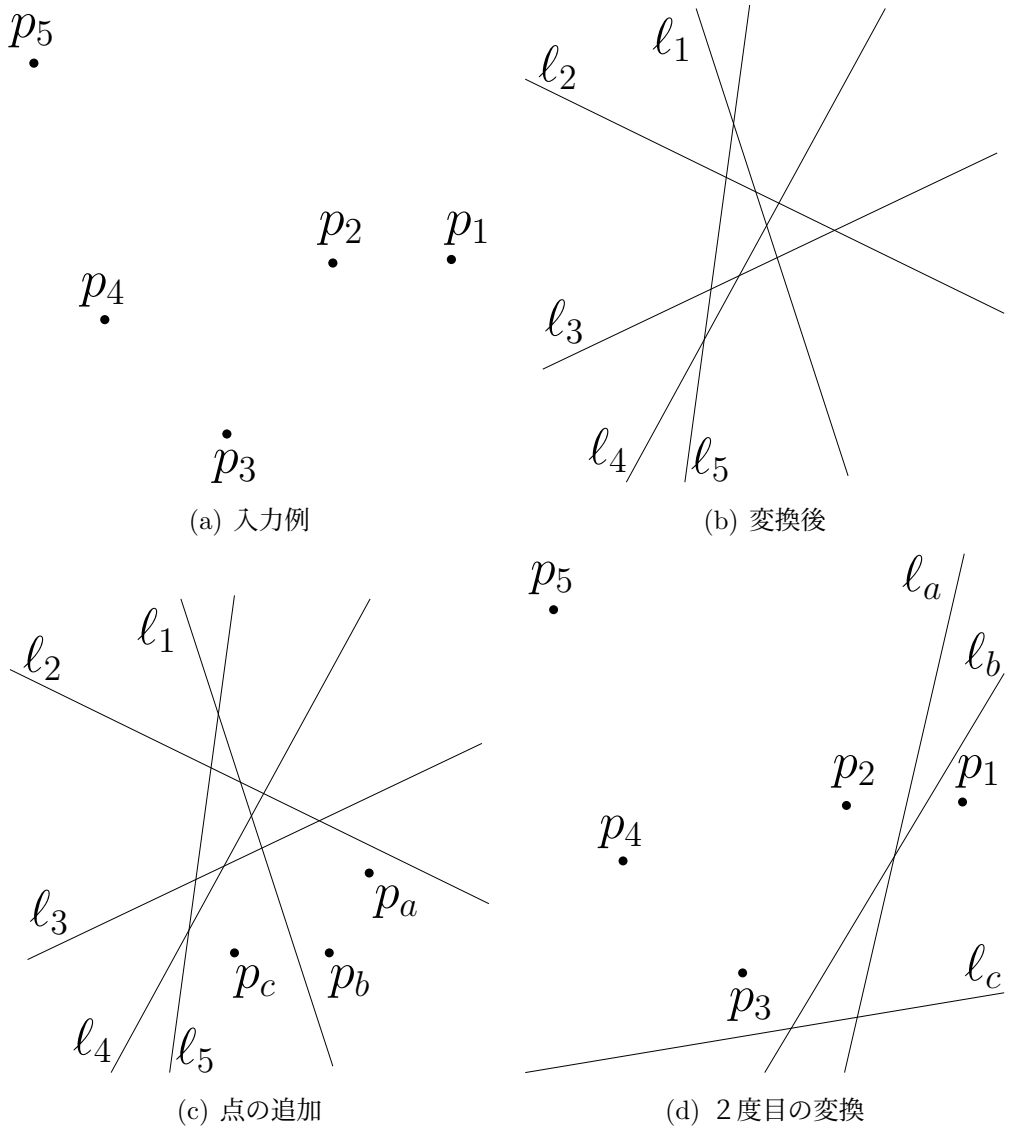


(a) 変換前　　(b) 変換後

図 4.2: 双対変換

５個の点集合に対する双対変換の例を図 4.3 に示す．図 4.3(a) のような入力例が与えたれたとする．双対変換後は図 4.3(b) になる．$p_1, p_2, \ldots, p_5$ は $\ell_1, \ell_2, \ldots, \ell_5$ にそれぞれ変換されている．図 4.3(c) は図 4.3(b) に新たに３点加えた状態である．図 4.3(c) を再び双対変換すると図 4.3(d) になる．p_a, p_b, p_c の変換後の直線は，ℓ_a, ℓ_b, ℓ_c にそれぞれ変換されている．図 4.3(c) の p_a と p_b は $\ell_1, \ldots, \ell_5$ に対して点の位置が $\ell_2, \ell_3, \ell_4, \ell_5$ の下，ℓ_1 の上といった上下関係になっている. 対応する直線 ℓ_a, ℓ_b が分割する点集合のパターンは同じである．

このように，同じセルに属する２点は n 本の直線と上下関係がすべて同じであるため，この２点を双対変換してできる直線の点集合の分割は同じである．すべてのセルについてそれぞれ１点選び，その点に対応する直線による分割を調べれば，すべての分割パターンを調べたことになる．



(a) 入力例　(b) 変換後

(c) 点の追加　(d) ２度目の変換

図 4.3: 双対変換の例

4.3 作業領域調節可能アルゴリズム

ここでは，トポロジカルスィープのように直線のアレンジメント全体を走査する作業領域調節可能アルゴリズムを示す．アルゴリズムはとてもシンプルで，作業領域を示すパラメータ s が与えられた際，n^3/s 時間でアレンジメント全体の走査をすることが可能なアルゴリズムである．提案アルゴリズムではセルを隣から隣へと順に訪問する操作が一番重要なポイントとなる．その際に限られた作業領域で隣のセルがどのセルであるかを調べる操作に優先度付きキューを使用する．作業領域が調節可能な優先度付きキューに関して説明を示したあとに，全体のアルゴリズムを示す．優先度付きキューに関しては，文献 [3] に示されているが，ビットレベルでデータ構造を作り，操作するため複雑である．2 節でビットレベルの操作を考慮した説明をする．

しかし，１つの小領域を調べる際に n 点全てを調べていては時間がかかる．次に，隣合う小領域の特徴から計算時間を短くする方法を説明する．

図 4.3(c) の p_c は直線 ℓ_1 を挟んで隣の小領域に属してる．p_c の属してる小領域は p_a や p_b の小領域と比較すると，ℓ_1 の上下関係だけが違う．このように，直線を挟んだ隣の小領域では挟んだ直線の上下関係が逆になっている．p_c は ℓ_1 の下に位置するので，ℓ_c のようになる．ℓ_a や ℓ_b と比較すると p_1 の上下関係のみ違う．辺によって隣接してる小領域は，その辺を含む直線の上下関係が逆になっている．隣接している小領域について調べる際は，上下関係が逆になる直線に対応する１点について計算するだけでよい．隣接している小領域を順に調べることで，全体の計算時間を小さくすることができる．隣から隣の小領域を調べることで，各小領域について調べる時間を定数時間で行うことができる．

双対変換によって点を直線に写像すると，多数の直線によって平面が区切られた構造ができ，直線によって $O(n^2)$ 個の小領域ができる．この構造を直線のアレンジメントと呼び，小領域のことをセルと呼ぶが，直線のアレンジメントは，情報として直線だけでなく，直線同士の交点と，交点と直線の間の接続関係も含んでいるため，$O(n^2)$ の記憶領域が必要であるため，メモリに制約がある場合，すべての情報を保持することはできない．

入力データの点から直線への双対変換は，$O(1)$ 時間で可能であり，直線同士の交点を求める時間も $O(1)$ で可能であるため，直線や交点の情報はメモリに蓄えない．必要なときに計算により求めることで，作業領域を節約する．

作業領域を節約し，隣接してセルを隣から隣へと訪問するには次のようにする．図 4.4 のように ℓ_i と ℓ_j の交点を c_{ij} とし，その右側の ℓ_i の上のセルの１点を c_{ij}^+ として，下のセルの１点を c_{ij}^- とする．

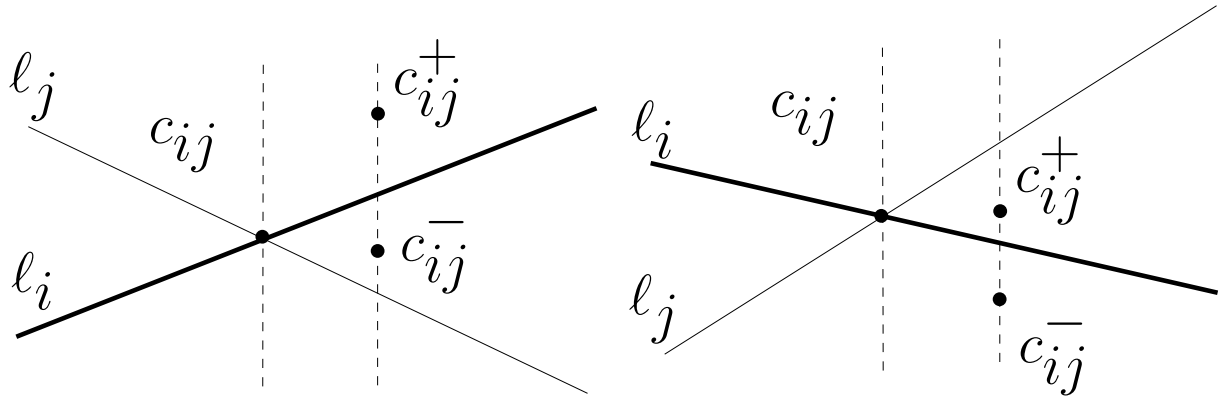


図 4.4: 2 つの交差．c_{ij}^+ と c_{ij}^- の定義

ℓ_i 上の交点が端から順に訪問できるとして，図 4.5 のように ℓ_i の上のセルと下のセルについてそれぞれ端から順に隣接しているセルについて調べる．

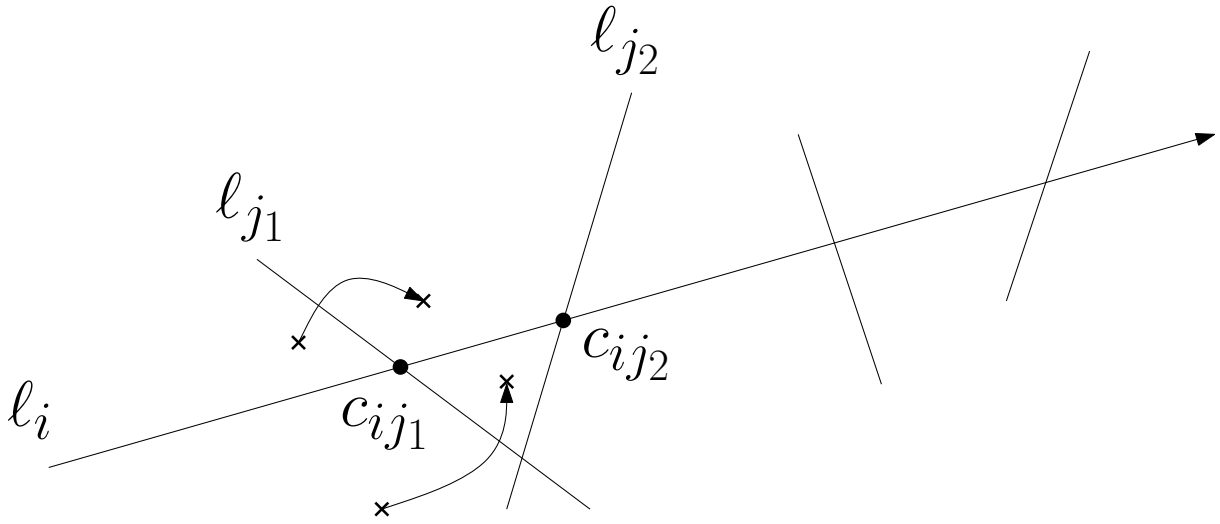
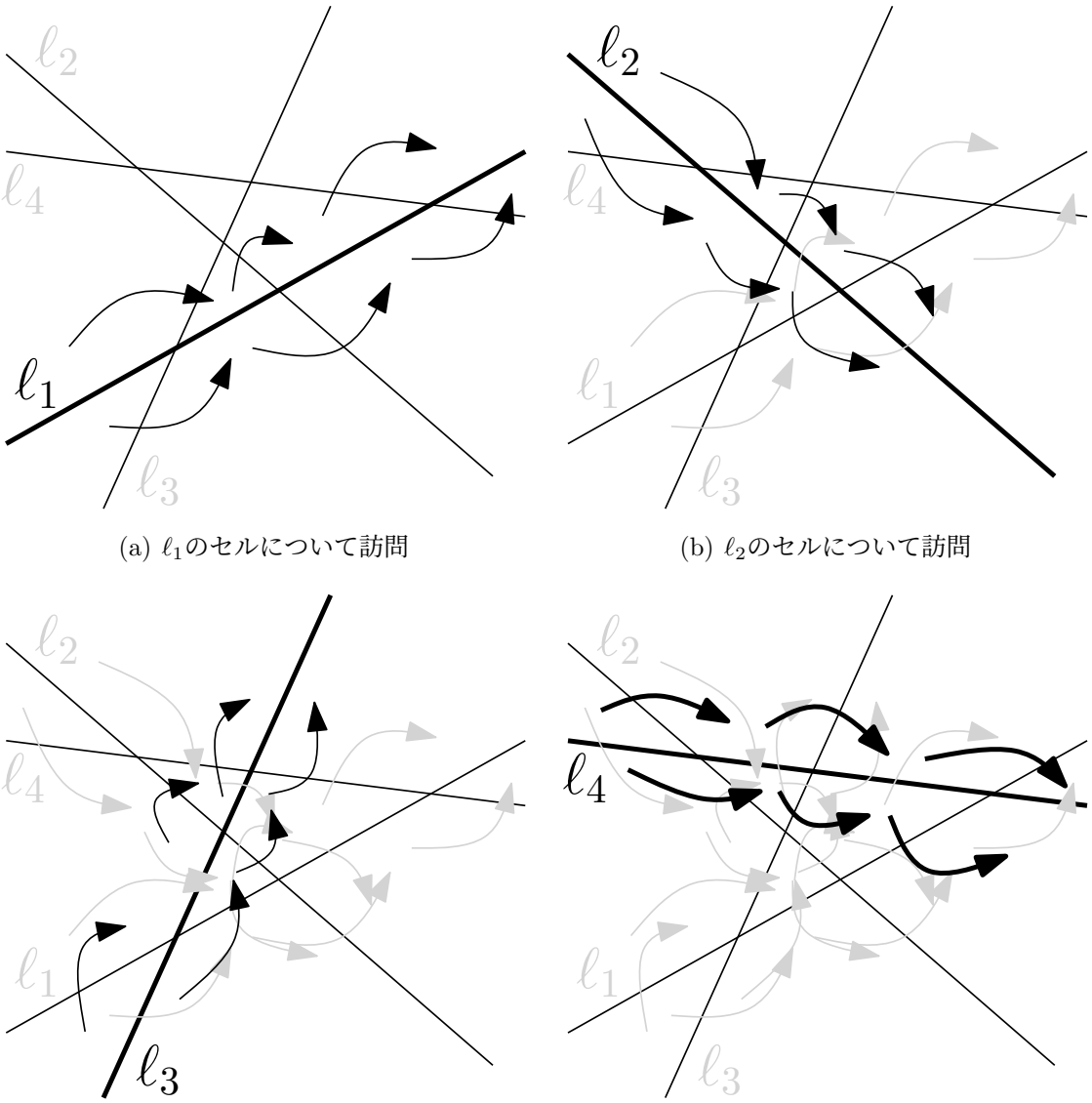


図 4.5: セルの訪問



(a) ℓ_1のセルについて訪問　(b) ℓ_2のセルについて訪問

(c) ℓ_3のセルについて訪問　(d) ℓ_4のセルについて訪問

図 4.6: すべてのセルの訪問の例

端のセルについて調べる際は$O(n)$時間かかるが，それ以降の隣接しているセルについて調べる際は$O(1)$でそれぞれ求められる．

すべての直線に対し同様のことを行えば，すべてのセルについて調べたことになる（図4.6）．アルゴリズム 5 に全体のアルゴリズムを示す．

アルゴリズム 5 重み付き点集合の最適分割問題に対するメモリ調節可能アルゴリズム

Input: n 個の重み付き点集合 $S = \{p_1, p_2, \ldots, p_n\}$, 作業領域のサイズ s.

Output: 両側の点集合の総和の差が最小になるように分割する直線 ℓ

```
min=∞
for all p_i do
  ℓ(p_i) を点 p_i の双対変換後の直線とし，c_ij を ℓ(p_i) と ℓ(p_j) の交点とする．
  ℓ(p_i) を除く ℓ(p_1), ..., ℓ(p_n) のそれぞれの直線と ℓ(p_i) との交点のなかで，最も左にあるものを求める．
  求めた交点の左側で ℓ(p_i) の上のセル内の点 c^+_{ij_0} と下のセル内の点 c^-_{ij_0} について総和の差をそれぞれ求める．
  min より総和の差が小さければ，より小さいほうで min を更新する．総和の差が最小になる点 c_min の情報も更新する．
  for k = 1 to n − 1 do
    次の交点 c_{ij_k} を求める
    c_{ij_k} の右側で ℓ(p_i) の上のセル内の点 c^+_{ij_k} と下のセル内の点 c^-_{ij_k} について総和の差を c^+_{ij_{k-1}} と c^-_{ij_{k-1}} からそれぞれ求める．
    総和の差を計算し，小さければ，min と c_min を更新する．
  end for
  ℓ(c_min) を出力する．
end for
```

アルゴリズム 5 は，作業領域のパラメータ s が $\lg n$ から $n/\lg n$ の間で決められているとき，$O(n^3/s)$ 時間で動作する．

直線上の $n-1$ 個の交点を端から順に訪問する際に文献 [3] を使用することで，$O(n/s)$ 時間で次に訪問する交点を求めることができる．

端のセルについて計算するときは，$O(n)$ 時間かかるが隣接しているセルについては $O(1)$ 時間で計算できる．直線上のすべてのセルについて計算する時間は交点を順に訪問するときにかかる時間と同じく，$O(n^2/s)$ である．n 本の直線について同様のことを行うので，全体の時間は，$O(n^3/s)$ である．

第5章　おわりに

本研究では，入力データが読み込み専用配列に蓄えられていると仮定した問題に対して具体的な問題２つを作業領域が調節可能であるアルゴリズムを示した．

１つは．3 章で 3-SUM 問題を取り上げた．3-SUM 問題クラスとして知られている問題は多い [5]．個々の問題は他の問題への変換が可能であるため，本論文で提案したアルゴリズムを使用することで，他の問題を作業領域調節可能で解けることが期待できる．

もう１つは，4 章で重み付き点集合の最適分割問題を取り上げた．トポロジカルスィープは直線のアレンジメントを走査することで，多くの問題を解いているので，4 章で説明した提案アルゴリズムを使用することで同様に多くの問題を作業領域を調節して解くことが期待できる．しかし，アレンジメントの走査はトポロジカルスィープの走査の仕方とは異なるため，提案アルゴリズムをそのままトポロジカルスィープと置き換えるだけでは解けない．提案アルゴリズムを使用してトポロジカルスィープで解けている問題を作業領域を調節して解くには，検討が必要である．

２つ問題は入力データが読み出し専用配列に蓄えられていると仮定している．$O(n)$ の作業領域を使用せずに要素をソート順にを取り出す方法で作業領域を調節可能にしている点が重要である．また，アルゴリズム 4 ではキューをコピーすることで，何度も同じソート順での取り出すことができているため，ソート結果を保持せずに解いている点がポイントである．そして，アルゴリズム 5 では，入力データの２点を双対変換してできる直線の交点の値が $O(1)$ で可能であること着目し，変換後の値を蓄えずに必要なときに計算により求めることで，作業領域の削減を図るテクニックを使用している．

本論文では，２つの具体的な問題に対し，作業領域調節可能アルゴリズムを示した．その２つの問題に関連している問題は多く，他の作業領域調節可能アルゴリズムを設計する上で有用と思われるテクニックについても述べた．

謝辞

学生生活を送るに**あ**たって様々な人にお世話になりました.

主指導の浅野哲夫教授には，アルゴリズムを一から丁寧に教えて下さったことに感謝して**い**ます．講義内容やゼミでの指導では，知識だけではなく，**ど**のようなことがポイントであり，対象の本質は何かといったことを考え**る**癖が身につきました．**ま**た，人へ限らた時間で的確に物事を伝えることの大切さや難しさ，そして準備の必要さといったところまで学べたことは，とても良かったと感じます.

上原隆平教授には，ゼミでの指導はもちろんのこと，パズル関係で楽しませていただいたことに感謝します．JAIST ギャラリーの運営に関わらせて頂いたことで，パズルへの興味が深まり，数学に対**す**る興味も一層深まりました．また，多くのパズル関係者にお会いでき，楽しく学校生活を送ることができまし**た**.

大舘陽太助教とは研究室のブースが近いこともあり，研究に対する姿勢や情熱が伝わってきて良い刺激になりました.

最後に，同じ時間を共にした友人，家族，全ての方に敬意と感謝の意を表したいと思います.

参考文献

[1] Herbert Edelsbrunner and Leonidas J. Guibas. Topologically sweeping an arrangement. *J. Comput. Syst. Sci.*, 38(1):165–194, 1989.

[2] Tetsuo Asano, Leonidas J. Guibas, and Takeshi Tokuyama. Walking on an arrangement topologically. *Int. J. Comput. Geometry Appl.*, 4(2):123–151, 1994.

[3] Tetsuo Asano, Amr Elmasry, and Jyrki Katajainen. Priority queues and sorting for read-only data. In *TAMC*, pages 32–41, 2013.

[4] 浅野 哲夫. **計算幾何.理論の基礎から実装まで**. 共立出版, 東京, Japan, 2007.

[5] Anka Gajentaan and Mark H. Overmars. On a class of $O(n^2)$ problems in computational geometry. *Comput. Geom.*, 5:165–185, 1995.