修 士 論 文

定数作業領域だけを用いた 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処理学専攻

田中 宏

2009 年 3 月

修 士 論 文

定数作業領域だけを用いたアルゴリズムに関する研究

指導教官　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 准教授
審査委員　宮地充子 教授

北陸先端科学技術大学院大学
情報科学研究科情報処理学専攻

0710044 田中 宏

提出年月: 2009 年 2 月

概 要

本研究では定数作業領域で実現出来るアルゴリズムの開発を目的とする．特に，使用可能な作業領域が厳しく制限されるハードウェア上で行う問題に対し，メモリ効率の良いアルゴリズムを提案する．本論文では，連結成分ラベル付け問題とユークリッド距離変換問題という，画像処理の基礎的な操作としてよく利用される二つの問題を取り上げ，定数作業領域かつ線形時間で解くことのできるアルゴリズムを提案する．

目 次

List of Algorithms

第1章　はじめに

1.1　研究の背景

情報の記憶はコンピュータにとって欠かせない機能であり，記憶装置によって行われる．記憶装置にはいくつか種類があり，速度や容量，価格帯などが異なる．その結果，現在では記憶装置に階層構造を用いるのが一般的になっている．具体的には，演算装置内の高速小容量のキャッシュメモリ，中速中容量の主記憶装置（メインメモリ），低速大容量の外部記憶装置（ハードディスク等）に分けられる．さらに低速となるが，ネットワーク越しの記憶装置を利用する事も出来る．このような記憶装置の特徴を上手く使い分ける事で，素早い処理の実行と，大きなデータの取り扱いと言う，二つの機能の実現を可能にしている．しかし，すべてのコンピュータがそのような環境を利用できる訳ではない．現在，コンピュータは身の周りの至る所に存在している．炊飯器やエアコンなどの家電製品，テレビやゲーム機，携帯電話などのデジタル家電，自動車や飛行機などの輸送機器，エレベータや自動販売機などの機械設備などである．これらの機器は一般に組込みシステムと呼ばれる．組込みシステムは年々知的で高度になり，組込みソフトの需要が増大している．パソコンのような汎用的なシステムと異なり，組込みシステムでは普通，上記のような環境は利用できない．そのため，一般的なソフトウェアと異なり，組み込みソフトではローカルメモリ内での処理を強いられる．例えば，スキャナやデジタルカメラでは，いくつかの画像処理アルゴリズムをハードウェア上で行わなければならない．しかし，多くの場合，ソフトウェアから利用できる総作業領域は厳しく制限されている．一方，取り扱うデータ量はハードウェアの性能向上に伴い，どんどん増加している．最新のスキャナでは，光学解像度 9600dpi×9600dpi，6 色光源 各色 16bit 階調で読み取る事が出来る．仮に A4 サイズの文書を読み込んだとすると，約 99.6GB のメモリが必要になる．実際には，そのような大きなサイズのメモリは使えないため，高解像度の時には読み取れる領域を制限する事で画像の容量を一定以下（例えば 2GB）に抑えている．この上に補正や加工などの画像処理を行おうとした場合，その作業の為にさらに多くのメモリが必要になる．

扱うべきデータが巨大である場合には，メモリを効率良く使う事がなおさら重要になってくる．そのため，メモリ効率のよいアルゴリズム，特に入力サイズに依存しない定数作業領域で実現出来るアルゴリズムが求められる．定数作業領域で実現出来るアルゴリズムが増えれば，必要となるメモリは最小限で済み，同じメモリサイズでより大きなデータを扱えるようになったり，ハードウェア上でより多くの処理を行えるようになる．また，記憶装置の階層構造の中で，低速な記憶装置が不要になる事があるため，メモリ効率のよいアルゴリズムは組込みシステムに限らず汎用的なシステムにとっても嬉しい事である．

1.2 研究の目的

本研究は，定数作業領域で実現できるメモリ効率の良いアルゴリズムの開発が目的である．特に，ハードウェア上でよく行われる処理を対象にアルゴリズムを開発することで，限られた資源であるメモリをより有効に使うことができる．

それらの処理の一つとして連結成分ラベル付け問題がある．0 と 1 からなる 2 値画像が与えられたとき，繋がった 1 のピクセル部分を連結成分みなし，連結成分毎のラベルを決め，連結成分に属する各ピクセルにラベルを付ける問題である．画像処理を行う上で，基本となる操作である．しかし，現在利用されている FILL アルゴリズムや，Rosenfeld-Pfaltz のアルゴリズム [1] では線形時間で処理できるが，入力サイズに比例した作業領域が必要となる．

また別の問題として，ユークリッド距離変換問題がある．この問題も 0 と 1 からなる 2 値画像に対する処理であり，各 1 のピクセルに対して，直線距離で最も近い 0 のピクセルまでの距離を求める問題である．現在利用されている Kirkpatrick ら [2] や Hirata[3] のアルゴリズムは線形時間で処理できるが，入力サイズに依存した作業領域が必要となる．

本稿では，これらの問題に対し，既存のアルゴリズムと同じ線形時間でありながら，入力サイズに依存しない定数作業領域で実現出来るメモリ効率の良いアルゴリズムを提案する．

1.3 本論文の流れ

本稿では，第 2 章で連結成分ラベル付け問題について述べ，第 3 章でユークリッド距離変換問題について述べる．どちらも，問題の定義，既存のアルゴリズムの紹介，提案アルゴリズムの説明，提案アルゴリズムの拡張という流れになっている．

第2章　連結成分ラベル付け問題

この章では，画像処理の中でも基本的な問題の一つである連結成分ラベル付け問題に関して述べる．この問題は2値画像を各成分毎にラベルを付けて成分毎の繋がりを明確にする問題であり，例えば画像認識の基礎的な処理として利用される．既存のアルゴリズムでは画像の面積に比例した作業領域が必要であるが，ここでは定数作業領域で実現できるアルゴリズムを提案する．

2.1　問題の定義

$n \times n$ の画像（もしくは2次元配列）G_n があり，各要素 $G(x,y)$ は 0 か 1 の2値で構成される．[1]この時，0 の要素を **0-pixel**，1 の要素を **1-pixel** と呼ぶ．また要素 $p(x,y)$ の**隣接**を以下のように定義する．

$$N(x,y) = \{(x+1,y),(x-1,y),(x,y+1),(x,y-1)\}$$

二つの 1-pixel p と q が**連結**であるとは，G_n に，全ての i について $p_{i+1} \in N(p_i)$ となるような 1-pixel の列 $(p = p_0, p_1, ..., p_k = q)$ が存在することである．

連結成分とは，G_n の中で任意の 1-pixel の要素を含む集合のうち，どの2点も連結であるような 1-pixel の極大集合である．

以上の定義を図 2.1 に示す．

連結成分ラベル付け問題は，0 と 1 からなる2値画像 G_n が与えられた時に，各 1-pixel に対し，その要素が属する連結成分のインデックスでラベル付ける問題である．

例えば図 2.2 のような2値画像の場合，連結成分が4つある事が分かる．各連結成分のラベル 1〜4 を付けた配列が出力となる．この例ではラベルとして 1〜4 を用いたが，必ずしも1から連続した整数をラベルとして使う必要はない．

[1]この論文内では，簡単の為，入力サイズを $n \times n$ の正方形に固定して論じるが，長方形であっても本質な違いはない．

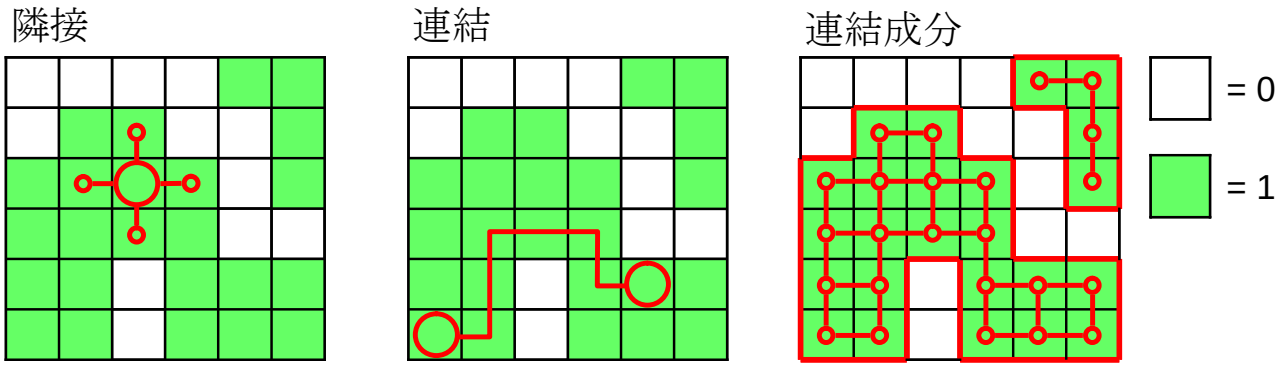

図 2.1: 隣接，連結，連結成分の図示．隣接は縦横に繋がった要素．連結は 2 つの 1-pixel 間に 1-pixel のみで構成される経路があること．連結成分は 1-pixel のみで構成される各まとまりのことで，この例では 2 つの連結成分からなる．

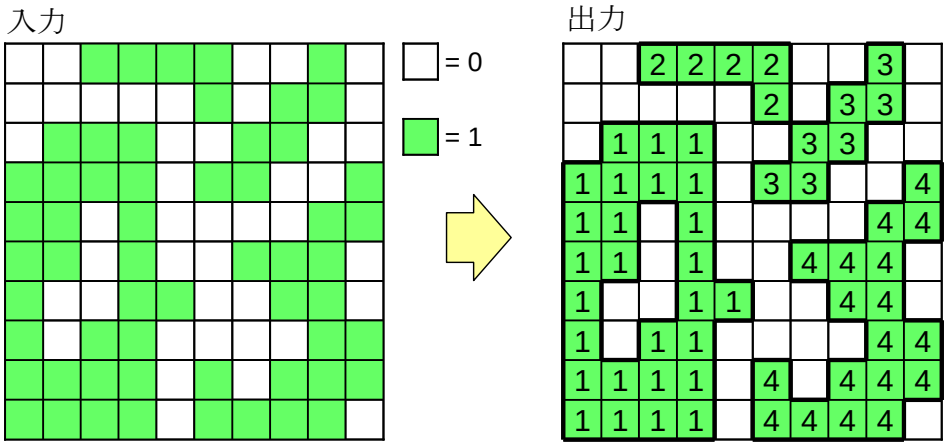

図 2.2: 連結成分ラベル付け問題の例．入力として 2 値画像を与えたとき，連結成分毎にラベルを付けた行列を出力する．この例では 4 つの成分に分けられる．

2.2 既存のアルゴリズム

現在利用されているアルゴリズムとして，Klette と Rosenfeld の本 [1] から二つ紹介する．

2.2.1 FILL アルゴリズム

一つ目は FILL アルゴリズムと呼ばれる簡単なアルゴリズムである．スタックを用いて周囲の隣接要素にラベルを伝播させていく方法で実現される．(Algorithm 1)

この方法での処理時間は $O(n^2)$ 時間であり，入力サイズに比例した線形時間で処理できる．しかし，図 2.3 のような例では，スタックに入力サイズに比例したサイズの領域が必要となるため，作業領域も $O(n^2)$ となる．

スタックの代わりにキューを用いた場合の作業領域はどうなるだろうか．多くの場合，スタックを用いた時よりも良い結果を出す．しかし，残念なことにキューを用いた場合でも $\Theta(n^2)$ かかる例を作る事ができる．

Algorithm 1: FILL アルゴリズム

Input: $n \times n$ の 2 値行列 G

```
L = 1
for each (i, j) do
    if G(i, j) = 1 then
        G(i, j) = L
        スタックに (i, j) を PUSH
        L = L + 1
        while スタックが空でない do
            スタックから (x, y) に POP
            for each N(x, y) do
                if N(x, y) = 1 then
                    N(x, y) = L
                    スタックに N(x, y) を PUSH
```

まず，配列の全要素を 0-pixel に初期化する．その後，配列の下中央から配列中央に向かって長さ $n/2$ の 1-pixel の経路を作る．次に中央から 5 方向に長さ $n/4$ の 1-pixel の経路を延ばす．前，右，左の 3 方向には直線で経路を延ばし，後方には最初に作った経路を避けるように 1 要素分のスペースを空けて，右後方，左後方の 2 方向に経路を延ばす．(図 2.4) この操作で作られた経路の末端を，前者の 3 方向では **F-endpoints**，後者の 2 方向では **P-endpoints** と呼ぶ．その後，各末端からさらに $n/8$ の経路を延ばす．F-endpoint からは先ほどと同じやり方で 5 方向に延ばし，3 つの F-endpoints と 2 つの P-endpoints を作る．P-endpoint からは 3 方向に経路を延ばす，前方，後方の 2 つに P-endponts を作り，残りの直交方向（右もしくは左）に 1 つの F-endpoint を作る．(図 2.5) この方法を延長経路の長さが十分小さくなるまで繰り返す．延長経路は毎回半分の長さになるため，反復回数は $O(\log n)$ 回となる．

図 2.6 はこの方法で作成した配列である．

k 回拡張した時の F-endpoints の数を F_k 個，P-endpoints の数を P_k 個とする．1 つの F-endpoints からは 3 つの F-endpoints と 2 つの P-endpoints が生成される．1 つの P-endpoints からは 1 つの F-endpoint と 2 つの P-endpoints が生成される．したがって次の式を得る．

$$F_{k+1} = 3F_k + P_k,\ P_{k+1} = 2F_k + 2P_k.$$

常に $F_k \geq P_k$ であることに気付く．

$$F_k - P_k = 3F_{k-1} + P_{k-1} - (2F_{k-1} + 2P_{k-1}) = F_{k-1} - P_{k-1} = \cdots = F_1 - P_1 = 1.$$

反復回数は $\Theta(\log n)$ であるため，次式を得る．

$$n^2 \geq F_k \geq P_k = \Omega(n).$$

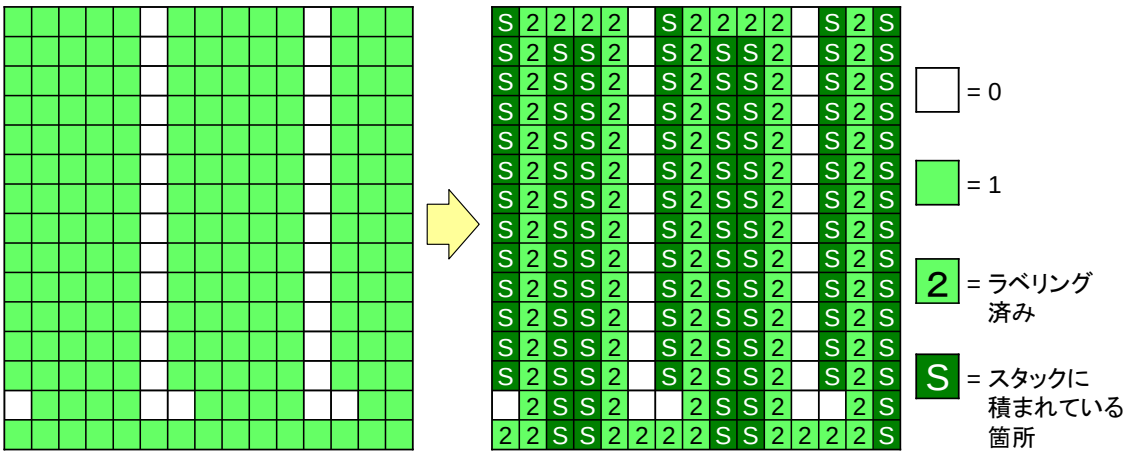

図 2.3: FILL アルゴリズムでスタックを用いた場合に $\Theta(n^2)$ 領域となるケース．左下から開始し，隣接要素を左下右上の順に探索しスタックに PUSH した場合，右上まで来た時点では全要素の約半数がスタックに積まれている．

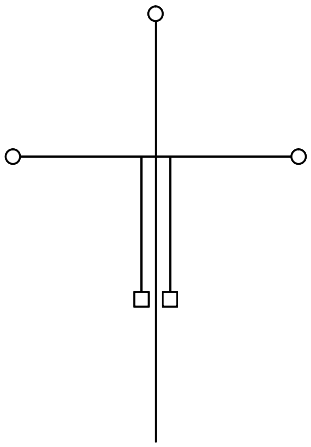

図 2.4: 中央からの 1 回目の拡張．中央から等距離に 3 つの F-endpoints（○）と 2 つの P-endpoints（□）を生成する．

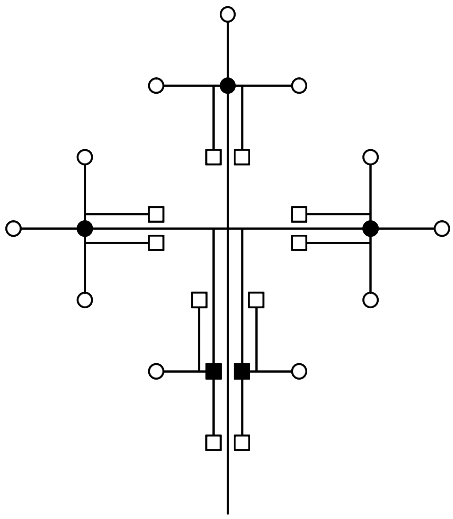

図 2.5: 2 回目の拡張．●と■は 1 回目の拡張の末端．

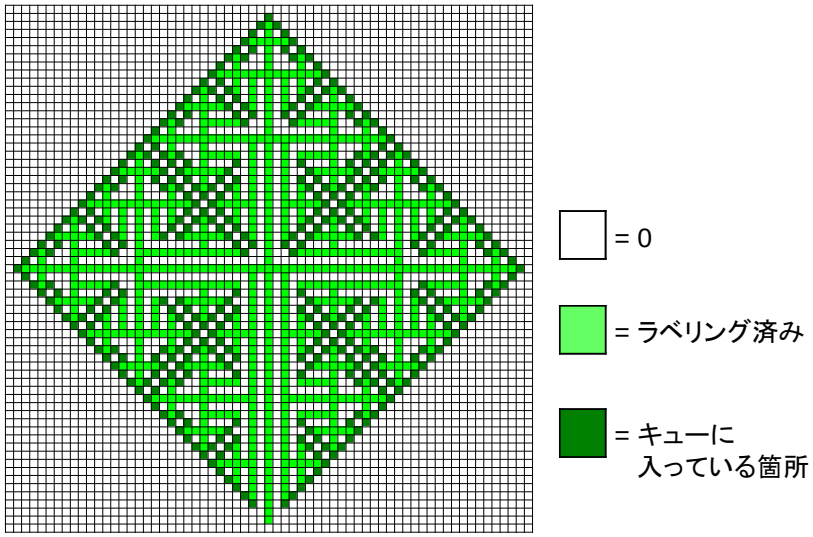

図 2.6: FILL アルゴリズムでキューを用いた場合に $\Theta(n^2)$ 領域となるケース．下から開始し，末端まで来た時点で開始点以外の末端要素が全てキューに入っている．

2.2.2 Rosenfeld-Pfaltz アルゴリズム

二つ目は Rosenfeld-Pfaltz アルゴリズムで，集合ユニオンファインド木を使った方法である．このアルゴリズムは 3 つのステップに分かれる．STEP 1 ではラスタースキャンを行い，各 1-pixel に仮ラベルを付ける．その際，異なる仮ラベルが隣接したとき，同値ラベル表に記録する．STEP 2 では，同値ラベル表から集合ユニオンファインド木を用いて仮ラベルを集合に分け，各集合の代表ラベルを決める．STEP 3 でラスタースキャンをしながら，仮ラベルが属する集合の代表ラベルで 1-pixel の仮ラベル置き換える．という方法である．(Algorithm 2，図 2.7)

Algorithm 2: Rosenfeld-Pfaltz アルゴリズム

Input: $n \times n$ の 2 値行列 G

```
//— STEP 1 —//
L = 1
for each (i, j) do
    if G(i, j) = 1 then
        if G(i, j) の隣接がラベリング済み then
            if G(i, j) の隣接の仮ラベルが全て同じ then
                G(i, j) = G(i, j) の隣接の仮ラベル
            else
                G(i, j) = min(G(i, j) の隣接の仮ラベル)
                同値ラベル表にそれらのラベルが同値である事を記録
        else
            L = L + 1
            G(i, j) = L
//— STEP 2 —//
STEP 1 で作成した同値ラベル表から同値類を決定し，
各同値類から代表ラベルを一つ選ぶ
//— STEP 3 —//
for each (i, j) do
    if G(i, j) > 1 then
        G(i, j) = G(i, j) の仮ラベが属する同値類の代表ラベル
```

このアルゴリズムのワーストケースは，図 2.8 に示すような市松模様である．このような場合，ラベルの数は $n^2/2$ となる．同値類や代表ラベルを決定は上手く行えば線形時間である $O(n^2)$ 時間で済むが，同値ラベル表のサイズや同値類と代表ラベルを決定するための集合ユニオンファインド木のサイズはラベル数に依存するため，作業領域も，$O(n^2)$ となる．

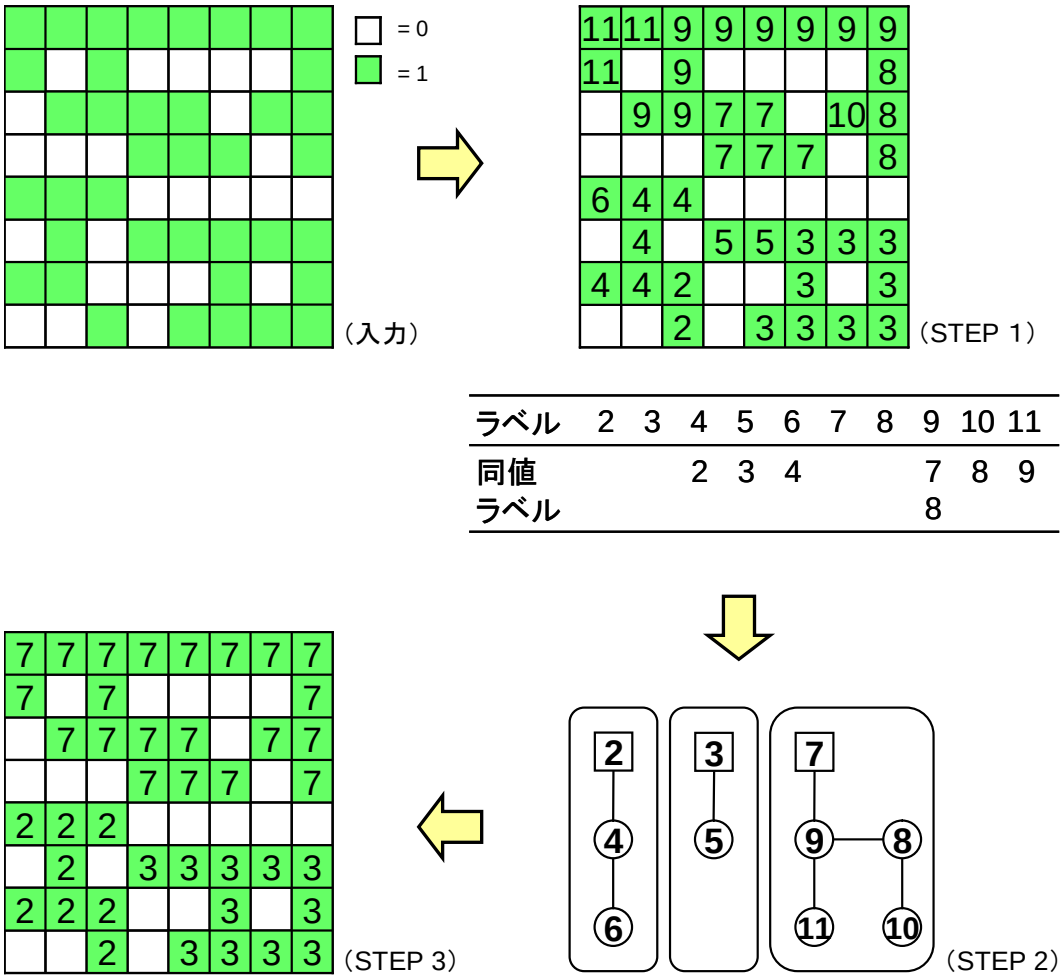

図 2.7: Rosenfeld-Pfaltz アルゴリズムの例．STEP 1 で仮ラベルを付け，同値ラベル表を作る．STEP 2 で表から同値類に分け，代表ラベル（この例では四角）を決定する．STEP 3 で各仮ラベルを代表ラベルに置き換える．

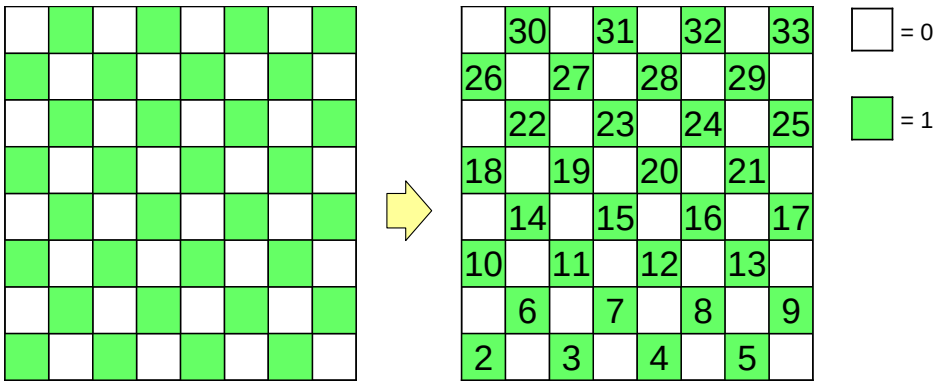

図 2.8: Rosenfeld-Pfaltz アルゴリズムのワーストケース．連結成分の数に比例して，ラベル管理に必要な領域が増える．

2.3 提案するアルゴリズム

既存のアルゴリズムより作業領域の点で優れた，入力に依存しない定数作業領域だけを用いたアルゴリズムを提案する．ここで提案するアルゴリズムは既存のアルゴリズムとは大きく異なる方法を用いる．基本的なアイデアは，各成分の境界をぐるりと辿ながら何らかの処理をするという方法である．簡単な方法ではあるが，成分の中に 0-pixel からなる穴が複数空いていたり，連結成分が入れ子になっていたりする事も考慮する必要があり，単純に辿るだけでは難しい．

2.3.1 準備

簡単のために G_n の周囲は 0-pixel で囲まれているとする．つまり，$G(i,0)=0, G(i,n-1)=0, G(0,i)=0, G(n-1,i)=0 (i=0 \text{ から } n-1)$ である．

G_n 上にある各要素を以下のようにラスター順に順序付ける．

$$(x,y) < (x',y') \text{ if } y < y' \text{ or } (y = y' \text{ and } x < x').$$

各連結成分に含まれる要素のうち，順序が最小となる要素を**最小要素**と呼ぶ．各連結成分の最小要素の順序を比較する事で，連結成分にも自然な順序付けが出来る．各連結成分を最小要素の小さい順に $C_1, C_2, \ldots, C_k$ とする．

配列の各要素は，隣りの要素と辺で区切られていると考える．特に 0-pixel と 1-pixel の間のにある辺を**境界辺**と呼ぶ．この論文中では境界辺の向きは 1-pixel 側から見て常に左向きであると決めて議論する．したがって，1-pixel の左側に境界辺がある場合，それらの向きは常に下向きに固定される．任意の境界辺から，その境界辺を含む完全な境界を辺の向きに従って辿る事は簡単に出来る．図 2.9 のように局所的な情報で境界の辿り方は決まる．このように境界を辿る操作を**境界追跡**と呼ぶ．

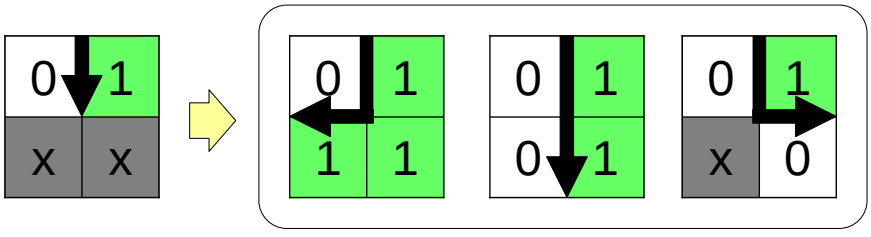

図 2.9: 下方向に辿ったあとの辿り方．他の方向についても回転させるだけで辿り方は同じ．

境界は，連結成分の外側を囲む**外部境界**と，連結成分の内側にある 0-pixel からなる穴とを区切る**内部境界**に分けられる．一般に各連結成分はただ一つの外部境界といくつかの内部境界を持つ．境界追跡の方法から，外部境界は反時計回りに進み，内部境界は時計回りに進む事になる．連結成分と同様，境界についても順序付けを行う事が出来る．各境界中の下向き辺の中でラスター順が最も早いものを**最小辺**と呼び，境界 B_i の最小辺を $e_s(B_i)$ と書く．各境界の最小辺の順序を比較する事で境界，及び最小辺に順序付けを行う．

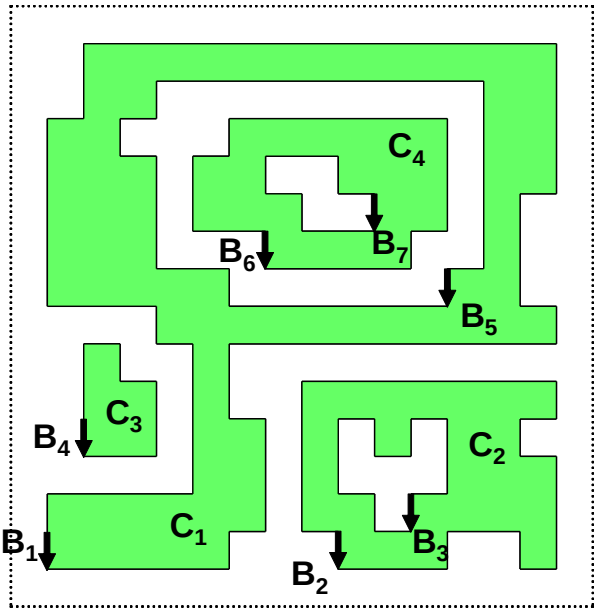

図 2.10: 2 値画像の例．4 つの連結成分と 7 つの境界からなる．

図 2.10 の例では，4 つの連結成分 C_1, C_2, C_3, C_4 があり，C_2 と C_4 は入れ子の関係になっている．各連結成分は 1 つの外部境界があり，C_1, C_2, C_4 にはそれぞれ，B_3, B_5, B_7 の内部境界からなる穴があり，全部で 7 つの境界がある．矢印で描かれた部分が各境界の最小辺はであり，順序は $e_s(B_1), e_s(B_2), \ldots, e_s(B_7)$ となる．各境界の順序は $B_1, B_2, \ldots, B_7$ となる．

補題 1 連結成分 C_k の内部境界を B_i とする．この時，B_i は連結成分 C_k の外部境界 B_j に囲まれ，B_j の最小辺は B_i の最小辺よりも前にある．

証明： 図 *2.11* 参照．内部境界 B_i が外部境界 B_j に囲まれているのは自明である．このとき，もし内部境界 B_i の最小辺 $e_s(B_i)$ を左方向に水平移動すると，必ず外部境界 B_j の下向き辺 e と衝突する．順序の定義より，$e < e_s(B_i)$．外部境界 B_j の最小辺 $e_s(B_j)$ は B_j 上のどの辺よりも前にあるため，$e_s(B_j) \leq e$．したがって，$e_s(B_j) < e_s(B_i)$．

補題 2 連結成分 C_k の内部境界の一つを B_i とする．この時， B_i の最小辺 $e_s(B_i)$ から左方向に水平移動した時に最初に見つかる下向き辺は C_k の境界辺である．

証明： 要素を水平に見た時，どんな場合でも，*0-pixel* の区間と *1-pixel* の区間が交互に現れる．したがって，最小辺から水平移動した時に見つかる境界辺の向きは，必ず上向き，下向きの順に交互に現れる．(図 *2.12*)

よって，$e_s(B_i)$ と ，$e_s(B_i)$ から左に水平移動した時に最初に見つかる下向き辺の間には一つの上向き辺が存在する．この時の上向き辺を e_u とし，最初に見つかる下向き辺を e_d とする．e_d と e_u の間は *1-pixel* の区間であるため，e_d と e_u は明らかに同一の連結成分の境界辺である．この事から，e_u が C_k の境界辺である事が言えれば良い．

内部境界は向きが時計回りであるため，下向き辺の左方には必ず同一内部境界の上向き辺が存在する．ここで二つのケースが考えられる．

ケース *1*：e_u が B_i の辺である場合．図 *2.13*（*a*），(*b*)．e_u が C_k の境界辺である事は明らかである．さらに，e_d と e_u は同一の連結成分の境界辺であることから，e_d も C_k の境界辺となる．

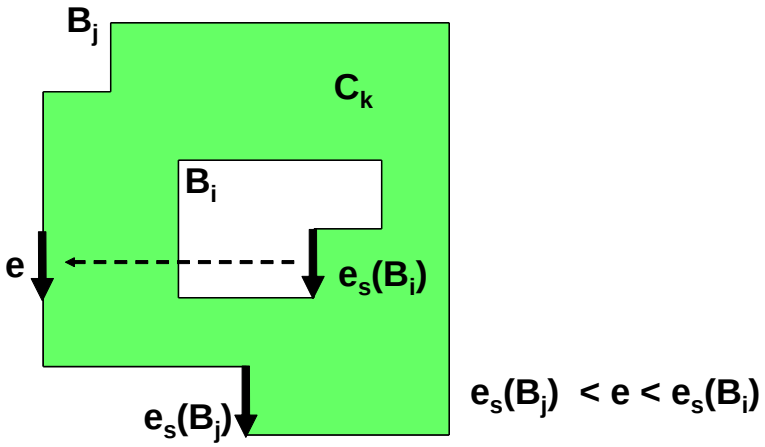

図 2.11: 外部境界の最小辺は内部境界の最小辺よりも先にある．

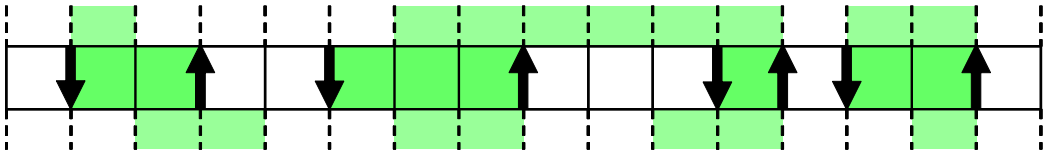

図 2.12: 水平に要素を見た時，境界辺は上向きと下向きが交互に並ぶ．

ケース *2*：e_u が B_i の辺でない場合．図 *2.13* (*c*)．$e_s(B_i)$ と B_i の上向き辺との間に B_i とは異なるの境界 B_x が存在することになる．B_x が存在するためには，B_i の中に C_k とは異なる連結成分 C_x が入れ子状態になっている場合以外にはない．C_x の周囲は *0-pixel* で囲まれているはずである．$e_s(B_i)$ の水平位置に C_x が存在するならば，周囲の *0-pixel* は $e_s(B_i)$ より下にも存在する事になる．すると，B_i には $e_s(B_i)$ よりも小さな辺が存在する事になり，最小辺の定義と矛盾する．よって，ケース *2* になる事はなく，e_u は必ず B_i の辺であり，e_d は C_k の境界辺となる．

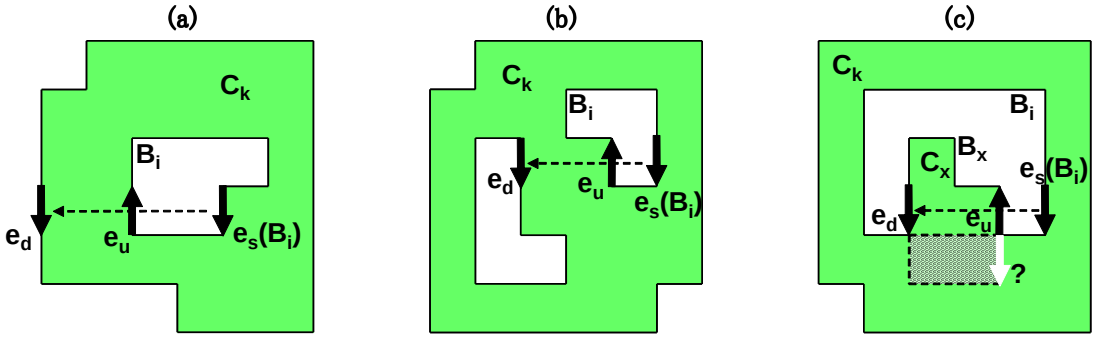

図 2.13: 最初に見つけた下向き辺 e_d のパターン．(a) では e_d が外部境界上にあり，(b) では e_d が内部境界上にある．(c) は e_u が B_i とは別の境界 B_x 上にあると仮定した場合．B_x が存在するためには点線内が 0-pixel でなければならないが，その場合，白矢印の部分が $e_s(B_i)$ でなければならず，矛盾している．

2.3.2 提案アルゴリズム

提案アルゴリズムは，2 つのステップに分けられる．STEP 1 では各連結成分の下向き辺の右の要素にラベルを付ける．STEP 2 では STEP 1 で付けたラベルをまだラベルが付いていない 1-pixel に伝播させる．

各ステップの説明の前に，提案するアルゴリズムで重要な処理となる**境界ラベリング (boundary labeling)** を定義する．

Boundary-Labeling(e, λ)
辺 e から境界追跡を行う．
下向き辺を見つけたとき，
辺の右側（進行方向左側）の 1-pixel をラベル λ で置き換える．

擬似コードでは Algorithm 3 のようになる．

STEP 1 では，G_n 上でラスタースキャンを行う．0-pixel ，1-pixel の順に並んだ要素つまり 2 要素間の下向き辺を探索する．そのとき，連結成分 C_1 の境界 B_1 の最小辺 $e_s(B_1)$ を最初に見つけることになる．補題 1 より，B_1 は必ず外部境界である．次に，見つけた最小辺 $e_s(B_1)$ と置き換えラベル 2 をパラメータとして Boundary-Labeling$(e_s(B_1), 2)$ を実行する．この操作により， B_i の下向き辺の右側の要素全てにラベル 2 がが付く．ラスタースキャンを再開し，次の 0-pixel ，1-pixel の間の下向き辺を探索する．この時，B_1 の下向き辺の右側の 1-pixel は先の境界ラベリングにより，全てラベル 2 に置き換わっているため，0,1 の並びにはならず，無視される．そのため，下向き辺の探索は次の最小辺を探索する操作になる．

次の最小辺となるのは 2 つのケースが考えられる．1 つはさっきとは別の連結成分 C_2 の外部境界の最小辺．もう一つはさっきと同じ連結成分 C_1 の内部境界の最小辺である．前者のような外部境界の場合は新しいラベルをパラメータとして Boundary-Labeling$(e_s(B_2), 3)$ のように実行し，後者のような内部境界の場合はその境界を含む連結成分と同じラベルをパラメータとして Boundary-Labeling$(e_s(B_2), 2)$ のように実行する．

以下，同様に繰り返す事で，G_n 中の全ての下向き辺の右側の要素にラベルを付けることができる．

STEP 2 では，再び G_n 上でラスタースキャンを行いまだラベルの付いていない 1-pixel を見つけたら，直前の要素のラベルで置き換える．

Algorithm 3: Boundary-Labeling 関数

Input: e = 境界辺， λ = ラベル

Result: 辺 e が属する境界の下向き辺の右要素に λ が付く

```
c = e の次の辺
while c ≠ e do
    if c の向き = 下方向 then
        辺 c の右（進行方向左）の要素 = λ
    c = c の次の辺
```

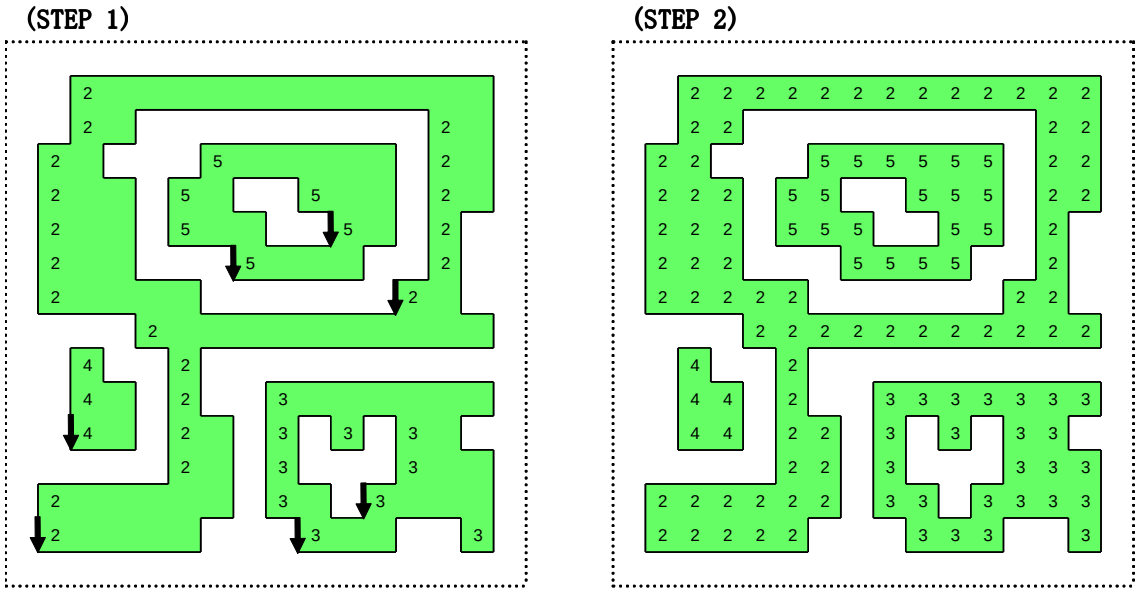

図 2.14: 各ステップの終了時の状態．STEP 1 で各連結成分の下向き辺の右側の要素にラベルを付け，STEP 2 でラベルを右に伝播する．

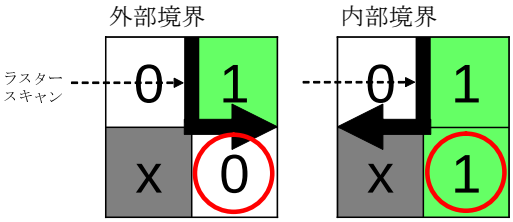

図 2.15: 外部境界と内部境界の判別方法．右下の要素が 0 の時は外部境界．1 の時は内部境界．

各ステップの終了時の状態を図 2.14 に示す.

STEP 1 では最小辺の属する境界が内部境界か外部境界かによって，処理を変えなければならない．そのためにはまず，見つけた最小辺が内部境界と外部境界のどちらに属するのか判断する必要がある．その判断は驚くほど簡単にできる．外部境界は反時計回り，内部境界は時計回りであり，最小辺の定義と辺の辿り方から，最小辺の直後には必ず左右のどちらかに曲がる事になる．曲がる方向はどちらの境界かによって決まっており，図 2.15 のように最小辺の右下の要素を調べるだけでどちらの境界か判断できる．

Boundary-Labeling のパラメータについて，外部境界の場合は単純に新しいラベルを用いればよいが，内部境界の場合はその境界が属する連結成分に使われたラベルを見つける必要がある．一見大変そうに思えるが，これも簡単な方法で実現できる．補題 2 より，内部境界 B_i の最小辺 $e_s(B_i)$ の左にある最初の下向き辺 e_d は，その内部境界と同一の連結成分に属している．さらに，境界の順序から，辺 e_d の右の要素にはすでにラベルが付けられている．また，その要素と最小辺 $e_s(B_i)$ の間には他の連結成分も存在しないため，ラスタースキャン中に最後に見たラベル，つまり辺 e_d の右の要素のラベルが Boundary-Labeling のパラメータとなる．

本研究で提案するアルゴリズムを擬似言語で表現すると Algorithm 4 のようになる．

Algorithm 4 の 2 つのステップ，境界ラベリングとラベルの伝播は一度のラスタースキャンで同時に行う事ができる．改良したアルゴリズムを Algorithm 5 に示す.

定理 1 *$n \times n$ の 2 値画像が与えられた時，Algorithm5 では連結成分のラベル付けを $O(n^2)$ 時間，定数作業領域で行える.*

証明： すでにアルゴリズムの正当性は示した．*Algorithm5* では各要素のアクセスは高々 *4* 回である．内訳はラスタースキャンで *1* 度の読み取り，境界を辿るために最大で *1* 要素あたり *2* 回の読み込み，ラベル付けに *1* 度の書き込みである．作業領域に関しても，境界ラベリング，及び，ラベルの伝播操作は定数個の変数があれば実現できる.

Algorithm 4: 提案アルゴリズム（連結成分ラベル付け問題）

Input: $n \times n$ の 2 値行列 G

```
//— STEP 1 —//
C = 1
for y = 1 to n − 2 do
    for x = 0 to n − 2 do
        if G(x, y) = 0 and G(x + 1, y) = 1 then
            e =2 要素間の境界辺
            if G(x + 1, y − 1) = 0 then
                C = C + 1
                Boundary-Labeling(e, C)
            else
                Boundary-Labeling(e, L)
        if G(x, y) > 1 then
            L = G(x, y)
//— STEP 2 —//
for y = 1 to n − 2 do
    for x = 0 to n − 2 do
        if G(x, y) > 1 then
            L = G(x, y)
        else if G(x, y) = 1 then
            G(x, y) = L
```

Algorithm 5: 提案アルゴリズム（連結成分ラベル付け問題）改良版

Input: $n \times n$ の 2 値行列 G

```
C = 1
for y = 1 to n − 2 do
    for x = 0 to n − 2 do
        if G(x, y) = 0 and G(x + 1, y) = 1 then
            e =2 要素間の境界辺
            if G(x + 1, y − 1) = 0 then
                C = C + 1
                Boundary-Labeling(e, C)
            else
                Boundary-Labeling(e, L)
        if G(x, y) > 1 then
            L = G(x, y)
        else if G(x, y) = 1 then
            G(x, y) = L
```

2.4 比較・検証

既存のアルゴリズムと提案アルゴリズムとを比較すると表 2.1 のようになる．計算時間は既存のアルゴリズムと同じ線形時間である．作業領域に関しては，入力サイズに比例した線形領域から，入力に依存しない定数作業領域に改善された事が分かる．

表 2.1: 各アルゴリズムの比較

	計算時間	作業領域
FILL アルゴリズム	$O(n^2)$	$O(n^2)$
Rosenfeld-Pfalz アルゴリズム	$O(n^2)$	$O(n^2)$
提案アルゴリズム	$O(n^2)$	$O(1)$

既存のアルゴリズムと提案アルゴリズムについて計算時間の比較を行った．

計算時間には入力画像の面積が大きく影響する．それ以外の要因はアルゴリズムによって影響力が異なる．FILL アルゴリズムでは 1-pixel の総数，Rosenfeld-Pfaltz アルゴリズムでは仮ラベルの総数，提案アルゴリズムでは境界の総長が計算時間に大きな影響を与える．よって，提案アルゴリズムの計算時間のワーストケースとベストケースはそれぞれ，図 2.16，2.17 のようになる．

入力画像へのアクセス回数により計算時間を比較する．FILL アルゴリズムでは，ラスタースキャンで 1 度の読み取り，スタックに積むために各 1-pixel の周囲の読み取りが 1-pixel につき 4 回，1-pixel のラベル上書きが 1 回である．Rosenfeld-Pfaltz アルゴリズムでは，ラスタースキャンで 1 度の読み取り，各 1-pixel の周囲の読み取りが 1-pixel につき 1 回（左と下の要素を見るが左は直前の要素なので値を保持できる），2 回目のラスタースキャンで 1 度読み取り，1-pixel についている仮ラベル上書きが 1 回である．提案アルゴリズムでは，ラスタースキャンで 1 度の読み取り，境界追跡で，直進，右折時に 2 回，左折時に 1 回の読み取り，1-pixel のラベル上書きが 1 回である．境界追跡時の方向を考えず単に 2 回としてまとめると以下のようになる

$$\begin{aligned} \text{FILL アルゴリズム} &: \text{面積} \times 1 + \text{1-pixel 数} \times 5, \\ \text{Rosenfeld-Pfaltz アルゴリズム} &: \text{面積} \times 2 + \text{1-pixel 数} \times 2, \\ \text{提案アルゴリズム} &: \text{面積} \times 1 + \text{1-pixel 数} \times 1 + \text{境界辺数} \times 2. \end{aligned}$$

境界追跡で 1-pixel からのアクセス回数の最大は 4 回であるため，提案アルゴリズムの最悪のアクセス回数は（面積 $\times 1 +$ 1-pixel 数 $\times 5$）となり，最悪の場合でも FILL アルゴリズムと同等であると言える．計算時間には，入力画像へのアクセス回数以外の要因も影響する．FILL アルゴリズムではスタックに関する処理，Rosenfeld-Pfaltz アルゴリズムでは仮ラベルの管理と代表ラベルの決定処理，提案アルゴリズムでは境界追跡の処理が，計算時間に大きく影響する．

実際に計算機上での実験結果を表 2.2，及び図 2.19 に示す．入力サイズは 8192×8192 である．全て 0-pixel，1-pixel のパターンと，連結成分の形が市松模様，四角形，斜め（右下がり），円形のパターンで，それぞれ連結成分の大きさを変えながら測定した．パターンのイメージを図 2.18 に示す．表 2.2 と図 2.19 を見るときは，CPU キャッシュの利用率を考慮すべきである．つまり，水平方向へのアクセスはメモリ上で連続配置されているため高速であるが，垂直方向へのアクセスでは，メモリアドレスが大きく離れるため，CPU キャッシュの性能が活かせない．また，スタック等で多くのメモリを使う場合も，画像データがキャッシュから追い出されやすくなり，速度が落ちる．これらの点が誤差として現れているので注意する．

オール 0-pixel では，各アルゴリズムのラスタースキャンでの読み取り時間が反映される．Rosenfeld-Pfaltz アルゴリズムでは 2 回，FILL アルゴリズム及び，提案アルゴリズムでは 1 回のラスタースキャンを行う．ただし，FILL アルゴリズムでは，1 要素を見ればよいのに対し，提案アルゴリズムでは境界辺を探すため，前後関係も見ているため，FILL アルゴリズムよりも時間がかかる．オール 1-pixel は，提案アルゴリズムが最も得意とするパターンである．実際，既存のアルゴリズムの半分以下の時間で済む．

Rosenfeld-Pfaltz アルゴリズムは，速度が比較的に一定しているが，仮ラベルの種類が多く，また同値類の数が多くなる斜めのパターンのような場合，ラベルの管理に時間がかかる．

FILL アルゴリズムは基本的に要素数に応じて実行時間が変化している．ただ，垂直方向へのアクセスが多いようなパターンでは CPU キャッシュが活かせず時間がかかっている．

提案アルゴリズムでは，小さな連結成分では他のアルゴリズムに劣るが，ある程度大きな連結成分になると他のアルゴリズムよりも速くなる．

各アルゴリズムの最悪な実行時間を比較すると，少なくとも今回示したパターンの中では，提案アルゴリズムが最も良い結果を出している．定数作業領域であり，メモリを効率よく使えることが速度にも影響していると考えられる．以上の点から計算機上の速度においても既存のアルゴリズムと同等かそれ以上の性能であるといえる．

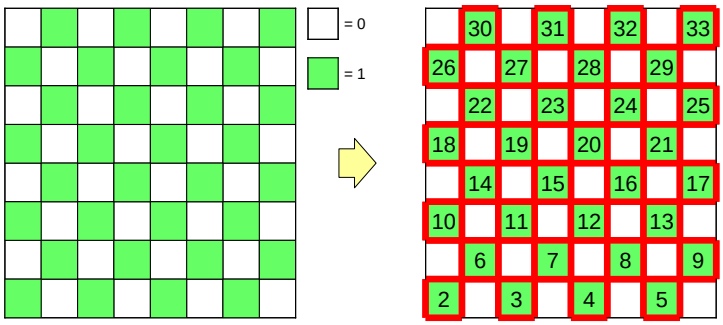

図 2.16: 提案アルゴリズムの計算時間のワーストケース．1-pixel 数に対する境界辺の総長が最長になる．

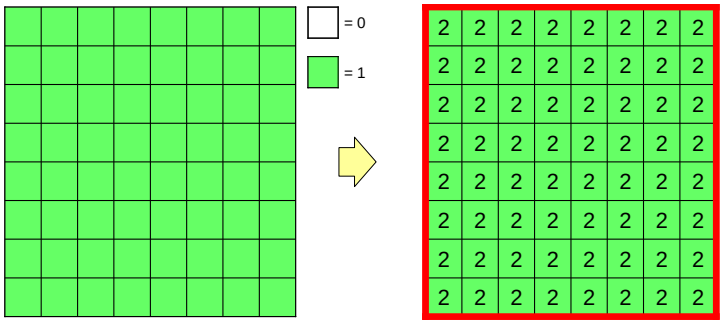

図 2.17: 提案アルゴリズムの計算時間のベストケース．1-pixel 数に対する境界辺の総長が最短になる．

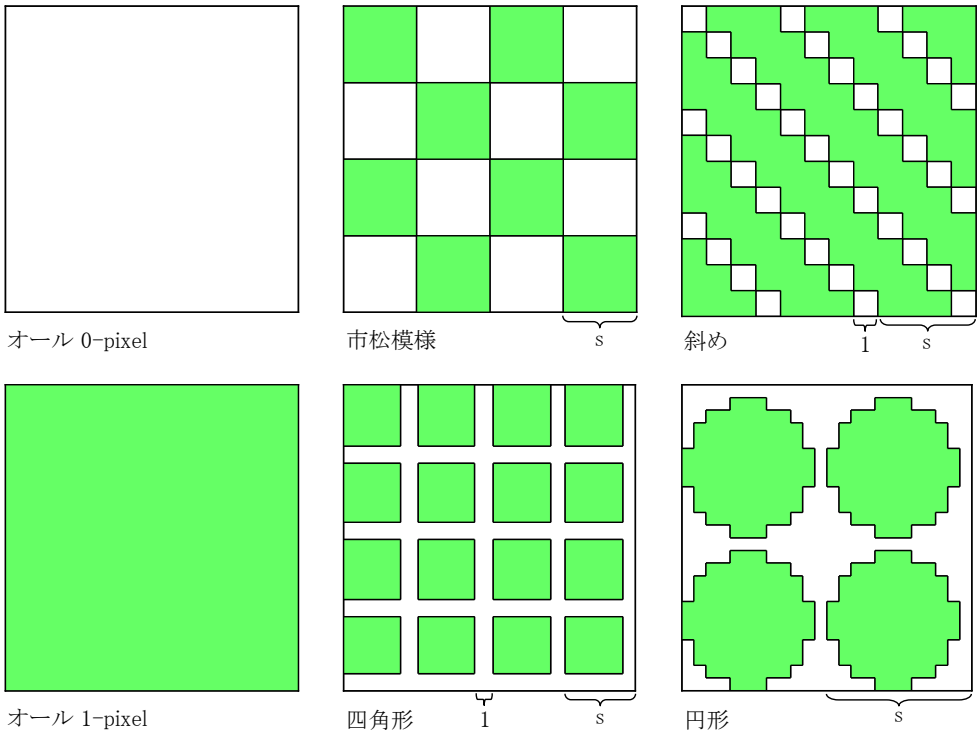

図 2.18: 計算機実験でのテストパターン.

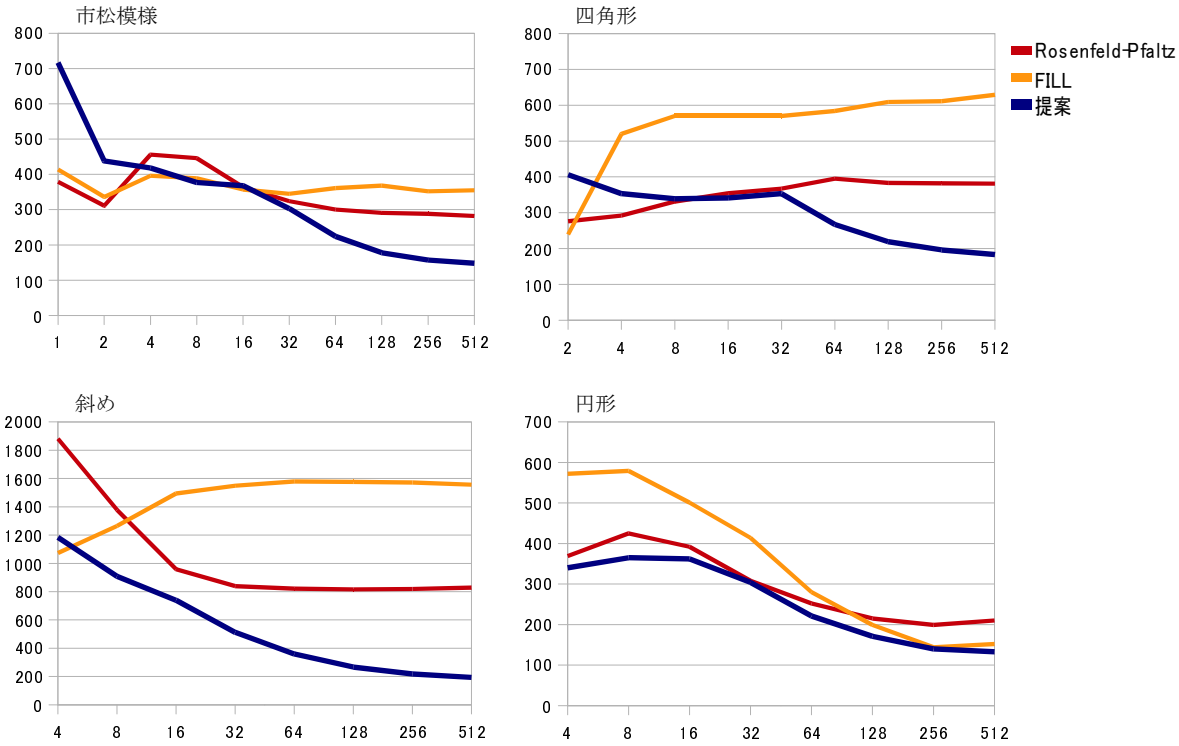

図 2.19: 各パターンの実験結果のグラフ.

表 2.2: 実行速度の比較．$n = 8192$．$s =$ 連結成分の大きさ．単位はミリ秒．

	s	Rosenfend-Pfaltz	FILL	提案
オール 0-pixel		173	78	105
オール 1-pixel		381	634	172
市松模様	1	379	414	717
	2	311	336	438
	4	456	396	418
	8	446	389	377
	16	364	357	368
	32	324	345	303
	64	300	361	224
	128	291	368	178
	256	288	352	157
	512	282	355	148
四角形	2	276	239	406
	4	292	520	353
	8	331	570	339
	16	354	570	341
	32	367	570	353
	64	395	584	267
	128	383	609	219
	256	382	611	196
	512	381	629	183
斜め	4	1881	1073	1184
	8	1378	1265	908
	16	958	1494	739
	32	839	1549	512
	64	821	1579	359
	128	816	1576	267
	256	819	1571	218
	512	828	1556	194
円形	4	369	572	340
	8	425	579	365
	16	392	501	362
	32	308	414	304
	64	252	280	221
	128	215	199	171
	256	199	144	140
	512	210	152	133

2.5 アルゴリズムの拡張

これまでは連結成分ラベル付け問題の中でも基本的な問題にのみ触れてきた．この章では，2 値画像の連結成分ラベル付け問題に類似した別の問題に対しても，提案アルゴリズムを適用する方法について述べる．

2.5.1 連結成分数え上げ問題

連結成分数え上げ問題は，2 値画像が入力として与えられたとき，1-pixel の連結成分の数を求める問題である．連結成分ラベル付け問題と異なり，各成分にラベルを付ける必要はない．

2.3.1 で述べたように，各連結成分には外部境界が必ず一つだけ存在する．つまり，連結成分数を数えることは，外部境界の数を数えることと同義である．前述の Algorithm 5 では，外部境界を見つける度に，変数 C の値をインクリメントし，新しいラベルとして用いた．よって，Algorithm 5 の変数 C を用いて連結成分数を簡単に数えられる．また，ラベルを付ける必要もないため，境界線のダブルカウントを防ぐ意味で目印となるラベルを一種類利用するだけで済む．

Algorithm 5 を連結成分数え上げ問題用に変更したアルゴリズムを Algorithm 6 に示す．目印のラベルとして 2 を用いている．

Algorithm 6: 提案アルゴリズム（連結成分数え上げ問題）

Input: $n \times n$ の 2 値行列 G

```
C = 0
for y = 1 to n − 2 do
    for x = 0 to n − 2 do
        if G(x, y) = 0 and G(x + 1, y) = 1 then
            e =2 要素間の境界辺
            if G(x + 1, y − 1) = 0 then
                C = C + 1;
            Boundary-Labeling(e, 2)
```

Output: C

2.5.2　8 連結への拡張

これまで連結成分は，上下左右の 4 方向に繋がった 4 連結であると暗黙にみなしてきた．当然，斜め方向も含めた 8 連結の問題も定義できる．8 連結での連結成分ラベル付け問題は，2.1 の問題の隣接の定義を以下のように変更したものである．

隣接.

$$N(x,y) = \{ \quad (x+1,y+1),(x,y+1),(x-1,y+1),(x+1,y),(x-1,y), \\ (x+1,y-1),(x,y-1),(x-1,y-1) \quad \}$$

8 連結の場合でも 4 連結に用いた提案アルゴリズムを少し変更するだけで適用できる．その変更は境界の辿り方と境界の種類の判別方法を図 2.20，2.21 のように変更するだけである．

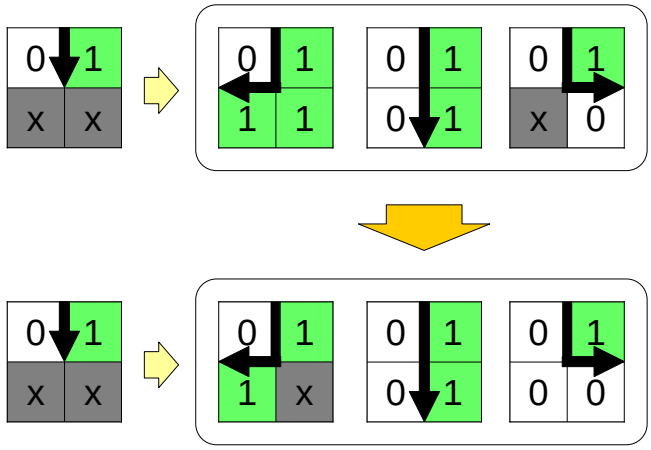

図 2.20: 辿り方の変更点．下方向に辿ったあとの辿り方．上は 4 連結，下は 8 連結．

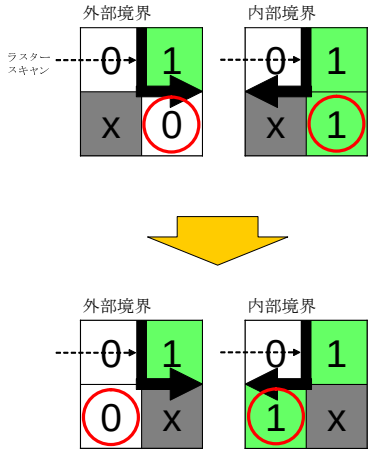

図 2.21: 外部境界と内部境界の判別方法の変更．上は 4 連結，下は 8 連結．8 連結では，左下の要素が 0 の時は外部境界．1 の時は内部境界．

ちなみに，1-pixel が 4 連結のとき，0-pixel は 8 連結になっており，1-pixel が 8 連結のとき，0-pixel は 4 連結になっている．

2.5.3 多値画像への拡張

連結成分ラベル付け問題は 2 値画像の一方の要素（本稿では 1-pixel）に対するラベル付けを行う問題である．この問題を拡張し，2 値画像の両方の要素に対するラベル付けや数え上げも効率良くできないか，また，2 値に限らずもっと多くの値の場合にも効率良くできないか，ということを考えるのは自然な発想である．ここで，多値画像に対する連結成分ラベル付け問題，及び，連結成分数え上げ問題について述べ，提案アルゴリズムを応用し，効率よく求める方法について述べる．

まず，多値画像に対する連結成分ラベル付け問題（4 連結）とは，図 2.22 のように，0 以上の整数値の入った画像を入力とし，全ての要素に対し，連結成分毎に異なるラベルを付ける問題である．

$O(n^2)$ の作業領域が許される場合，FILL アルゴリズムを 1-pixel に限らず全ての要素に対して行うだけで，線形時間で実行する事が出来る．

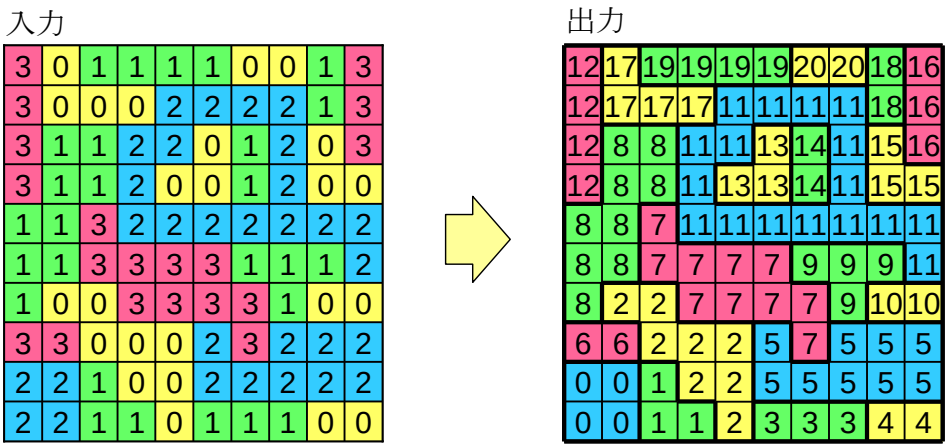

図 2.22: 多値画像に対する連結成分ラベル付け問題（4 連結）の例．入力として多値画像が与えられたとき，全ての要素に連結成分毎にラベルを付けた行列を出力する．この例では 21 の成分に分けられる．

定数作業領域で処理する場合を考える．自然な発想では，提案アルゴリズムを入力画像に用いられている数値の種類数分だけ何度も行えばよい．しかし，数値の種類は最大で n^2 種類となり，全体では $O(n^4)$ 時間かかってしまう．(図 2.23)

別の発想として，提案アルゴリズムと一度だけ走らせ，全ての要素に対してラベル付けを行えないか考える．単純に提案アルゴリズムを適用しただけでは，上手く行かない．例えば，図 2.24 のような，(a)（b）二種類の入力を考える．提案アルゴリズムを適用すると，まず，ラスタースキャンで最初に見つかる境界辺に対し境界ラベリングを行い，ラベル 4 を付ける．その次に見つかった 1 の境界ではラベル 5 をつけ，右に伝播させる．ラスタースキャンは次の行に移る．最初にある 4 はラベルなのでそのまま通りすぎ，次に見つかった 2 で問題が起こる．入力（a）（b）のこの時点の内容は同じである．仮に（a）が入力されていた場合，手前のラベルである 4 を伝播させなければならない．逆に（b）が入力されていた場合，新しくラベル 6 で 2 の境界ラベリングを行わなければならない．定数作業領域で実現するためには，(c）の状態からどうすべきかの判断材料である，ラベル 4 の元の値を記録しておく訳にもいかず，上手くいかない．各ラベルの元の値を記録するには，ラベルの種類数に比例する $O(n^2)$ 領域必要となる．

提案アルゴリズムを多値画像に対しても，定数作業領域，線形時間で上手く動くように拡張する，アプローチの異なる 2 つの方法を紹介する．

一つ目は，境界情報に着目し，最初に数値情報を境界情報に変換する方法である．図 2.24 のケースが上手くいかなかったのは境界情報が失われてしまったのが一つの原因である．実は提案アルゴリズムでは，元の数値情報は境界の判断とラベルを付けるべき要素の判断にしか用いられない．よって，全ての要素にラベルを付ける場合，境界さえ分かればよく，元の数値情報は必要ない．まず，境界情報の表現方法について述べる．境界情報は各要素の下位 4bit で四方に境界があるかどうかを示す．例えば，図 2.25 のように，四方の各境界を bit に対応させ，bit が 1 の時には，その位置に境界辺があるとみなす．入力例（a）に対しては（b）のような境界情報を表現する．

次に数値情報から境界情報に変換する方法について述べる．変換は 2 回のラスタースキャンによって実現する．1 回目のスキャンでは，左下から始め，各要素について，上の要素と右の要素とを比較し，境界の有無を判断しながら，上側の辺と右側の辺だけからなる境界情報に変換する．2 回目のスキャンは右上から逆方向に始め，各要素について，下の要素と左の要素の境界情報から境界の有無を判断し，下側の辺と左側の辺を追加した完全な境界情報に変換する．(図 2.26)

27	35	54	76	17	99	46	26	61	83
7	84	25	34	53	5	75	16	47	20
95	55	11	65	85	43	39	74	6	91
28	59	23	90	0	60	33	82	40	62
12	4	77	44	19	64	52	10	89	41
92	49	36	66	42	1	56	73	32	93
29	58	3	50	38	81	31	57	86	63
70	13	78	24	2	45	94	18	21	97
14	69	15	67	51	98	71	8	87	22
79	37	68	9	80	30	96	72	88	48

図 2.23: 提案アルゴリズムを数値の種類分何度も行う方法で $O(n^4)$ 時間かかるケース.

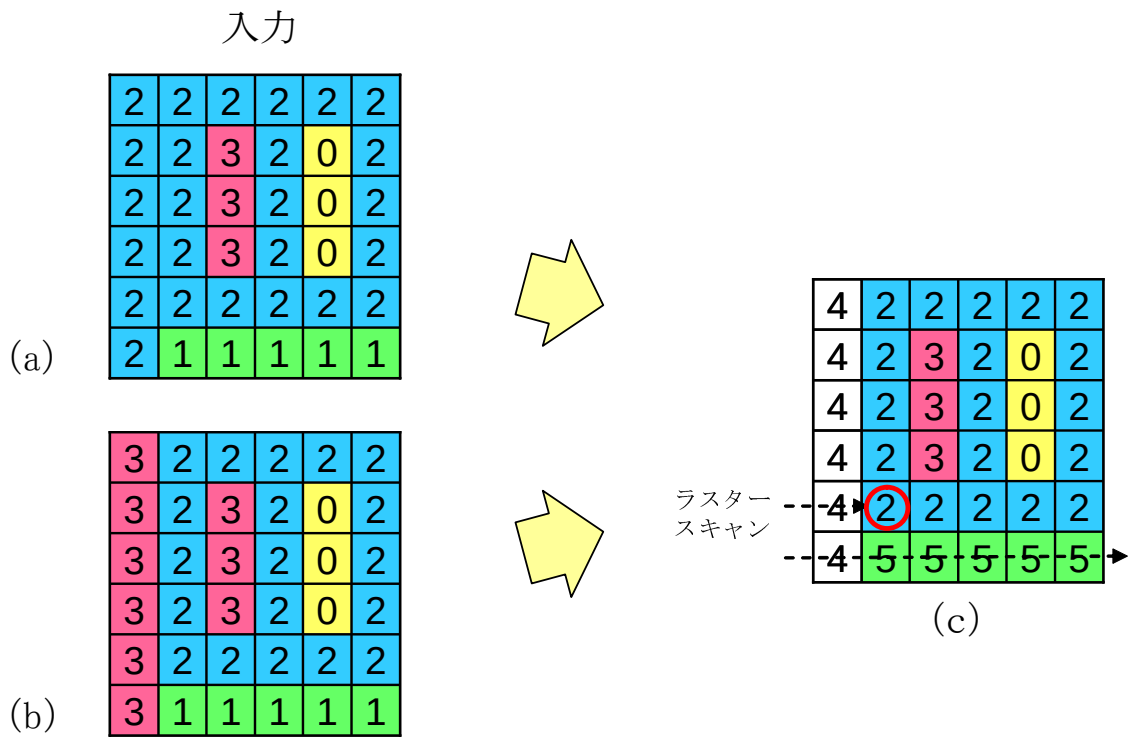

図 2.24: 提案アルゴリズムを一度のスキャンで単純に全ての要素に適用した場合. (a) (b) に対する処理は (c) の時点で同じであり，丸印の要素をどのように処理すべきか判断できない.

境界情報を求められたら，あとは，提案アルゴリズムを実行するだけである．まだラベルが付いていない要素に来たとき，左側に境界がなければ左の要素のラベルをコピーする．左側に境界があれば下側にも境界があるかどうかで外部境界か内部境界かを判断する．下側に境界のある外部境界の場合は，新しいラベルを用いて，境界をたどり，ラベルを付ける．下側に境界のない内部境界の場合は，下の要素のラベルを用いて，境界をたどり，ラベルを付ける．この方法であれば，図 2.24 のケースであっても，図 2.27 のように，要素の境界情報から，処理方法を正しく判断できる．この方法では，ラベルと境界を合わせて $n^2 + 16$ 種類の値を使える必要がある．また，8 連結では 8bit 必要になるため $n^2 + 256$ 種類の値を使える必要がある．

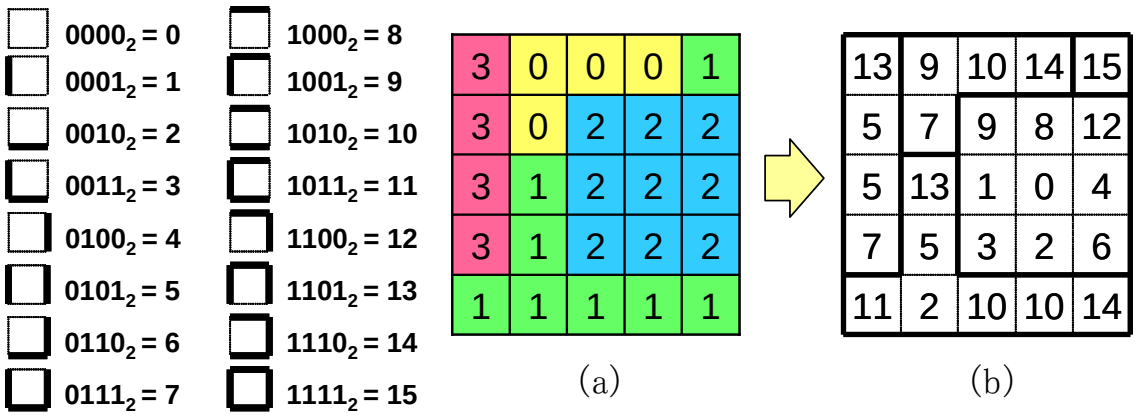

図 2.25: 境界情報の表現方法．四方の境界辺を 4bit で表現する．入力例（a）に対しては境界情報として（b）が得られる．

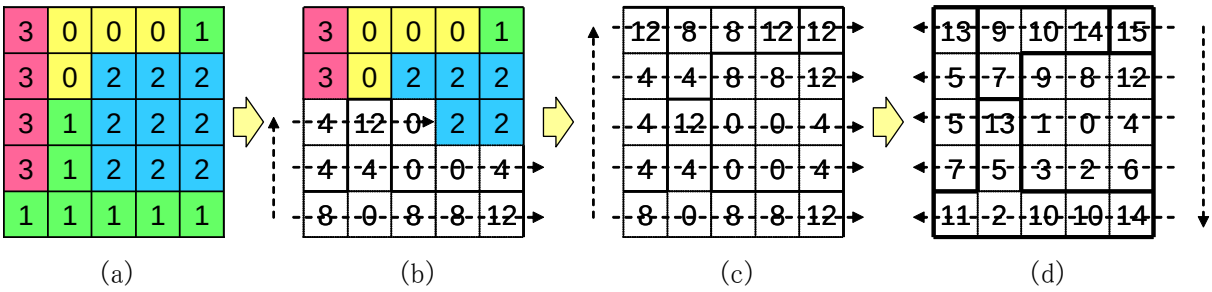

図 2.26: 数値情報から境界情報への変換方法． 入力として（a）が与えられたとき，1 回目のスキャンは，左下から（b）のように上と右の境界情報を作成していき，(c）のようになる．2 回目は右上からスキャンし，左と下の要素の境界情報をコピーして（d）のようになる．

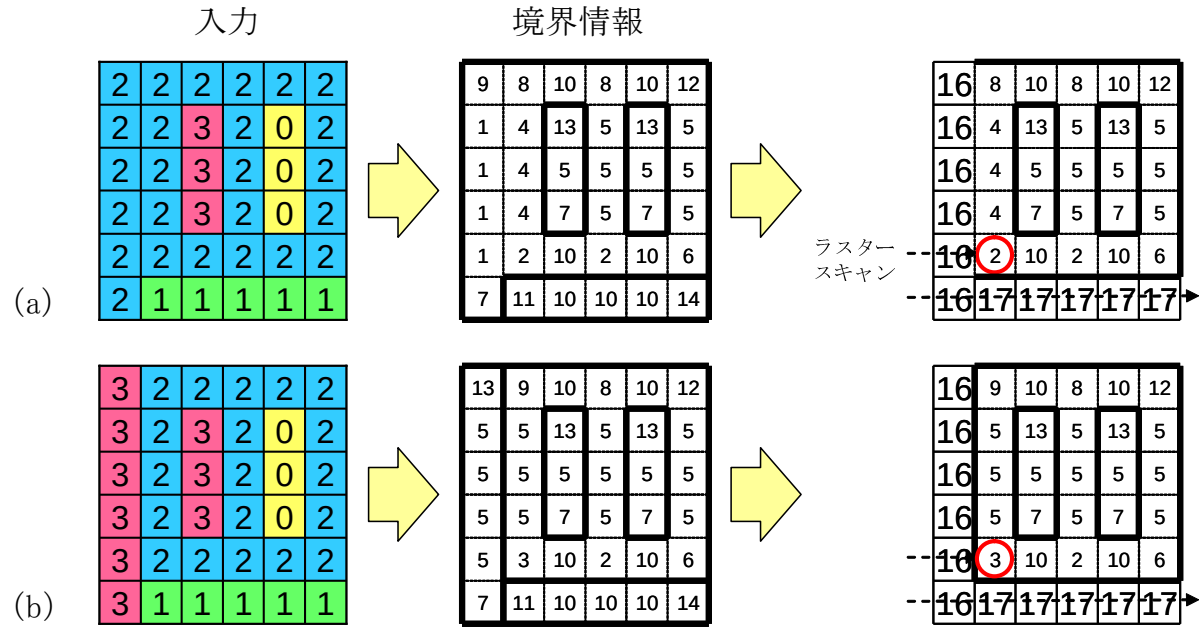

図 2.27: 図 2.24 のケースに境界情報を用いた場合．境界情報に 0〜15 を使っているため，ラベルは 16 から始める．丸印まで来ても，残っている境界情報から，処理方法を正しく判断できる．

二つ目の方法は，新しい連結成分に出会った時に，その連結成分に一気にラベルを付ける方法である．図 2.24 のようなケースでは，境界ラベリングで端にしかラベルが付かないため，同一連結成分中にラベルが付いている要素と付いていない要素が混在した状態が発生する．その状態のままラスタースキャンを行い，再びラベルに出会ったときにラベルの元の値が分からないのが上手くいかない一つの原因であった．もし，同一連結成分中にラベルが付いている要素と付いていない要素が混在した状態がラスタースキャンのタイミングで発生しなければ，つまり，ラベルを付け始めたら同一連結成分の全要素にラベルを付け，その後にラスタースキャンを再開するようにできればラベルの元の値を知る必要はなくなる．このようなことを実際に行う方法について述べる．

ラスタースキャンで新しく見つけた，ラベルのまだ付いていない要素の値を c とし，値が c である要素を c-pixel と呼び，c-pixel 以外の要素を o-pixel と呼ぶことにする．また，このときの連結成分に付けるべきラベルを L とする．

まず，c-pixel からなる連結成分の中に o-pixel で出来た穴（内部境界）がない場合を考える．この場合は連結成分の要素数に対する線形時間で実現出来る．図 2.28 のように境界追跡を 2 回行う．1 周目は提案アルゴリズムと同様に，下向き辺の右の要素にラベルを付ける．この時に用いるラベルは終点を意味する定数ラベル E である．2 周目は，上向き辺に出会った時，その位置から左方向にラベル L を付けながら進む．終点ラベル E に出会ったとき，左方向の進行は中止する．このときの終点ラベル E も ラベル L で上書きする．右方向にある元の上向き辺に戻り，境界追跡を再開する．以降，上向き辺に出会う度に同じ事を繰り返す．境界を辿り終えたとき，この連結成分に対するラベル付けは終了している．

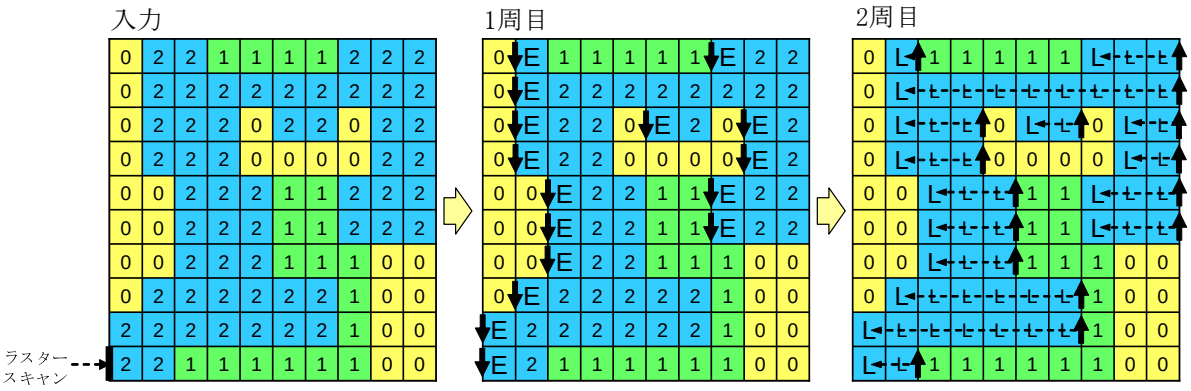

図 2.28: 連結成分に穴（内部境界）がない場合に一度に連結成分全体にラベル付ける例. 境界追跡を 2 回行い，1 周目は下向き辺の右に終点ラベル E を付ける．2 周目は上向き辺に出会う度に，終点ラベル E に出会うまで左方向に進みながらこの連結成分用のラベル L を付ける.

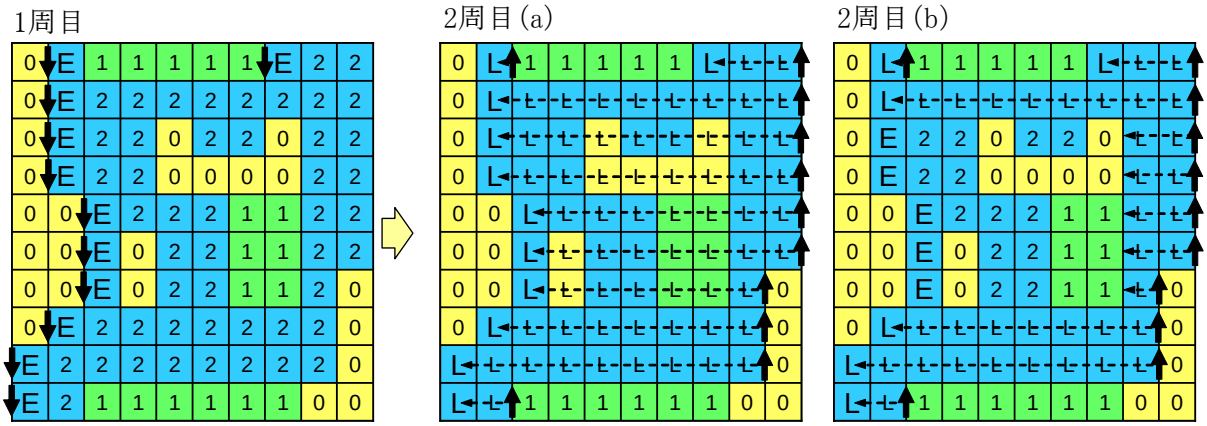

図 2.29: 連結成分に穴（内部境界）があり，上手くいかない例. (a) 終点ラベル E に出会うまでラベルを付ける場合. (b) 内部境界に出会った場合もラベル付けを止める場合．どちらも正しくラベル付けが出来ない.

穴（内部境界）がある場合を考える．前述の内容だけでは，外部境界の周囲にしかラベル E がついておらず，図 2.29 のように境界の 2 周目の処理が上手くいかない．(a）のように終点ラベル E に出会うまで左方向へのラベル付けを行う場合，内部境界内の要素にまでラベルを付けてしまう．また，(b）のように内部境界に出会った場合も左方向へのラベル付けを止める場合，内部境界より左の隠れた部分にラベルを付けられなくなってしまう．このような事態を避けるため，内部境界に出会った時になんらかの処理をする必要がある．

内部境界に対しても外部境界に対する処理と同じ処理を行う事で，この問題はほぼ解決する．すなわち，左方向に進んでいるときに内部境界（o-pixel）を見つけた場合，外部境界と同じようにその内部境界に対し，を 2 回境界追跡を行う．1 周目は下向き辺を見つけたとき，その右の要素に終点ラベル E を付ける．2 周目は上向き辺を見つけたとき，左方向に進みながら，終点ラベル E を見つけるか，境界（o-pixel）を見つけるまでラベルを付ける．境界を見つけた場合は，同様の方法を繰り返す．この方法では，境界の 2 周目の処理の途中で別の境界の処理が割り込む，再帰的な処理をが行われる．そのため，境界の辿り始めの位置や，別の境界が終わった時に戻る位置など，割り込み処理から戻った時に処理を再開するための情報が必要となる．それらの位置を仮に変数として記憶しておくとすると，境界毎にそれらの位置を記憶する必要がある．再帰処理の深さは図 2.30 のように最大で $O(n^2)$ であるため，作業領域として $O(n^2)$ 必要となる．よって，定数作業領域で処理するためには，再開に必要な情報を記憶することなく判断する必要がある．

任意の境界 B_c の処理が終わり，前の境界 B_p の処理を再開することを考える．境界 B_c の開始位置が分かれば，その位置から右に進み前の境界 B_p を見つけることができる．また，そのときの位置が B_p の処理を再開すべき位置である．どの境界の処理でも，1 周目の処理時には別の境界の処理が割り込むことはないため，1 周目に関しては，境界の開始位置，すなわちその境界で最初に見つけられた辺を記憶したまま処理する事が出来る．2 周目の処理では他の境界の処理が割り込むため，開始位置を記録しておくことは出来ないが，簡単に知る方法がある．1 周目の境界追跡時に開始位置には終点ラベル E を付けず，開始点を意味するラベル S を付ける．よって，2 週目では，下向き辺の右側の要素が S になっている所を探せば，開始位置を見つけ出すことができる．開始位置から前の境界 B_p に戻る前に ラベル S を L に上書きしておく．

以上の方法により，連結成分に穴（内部境界）が存在する場合でも，定数作業領域，線形時間で実行できる．具体例を図 2.31 に示す．

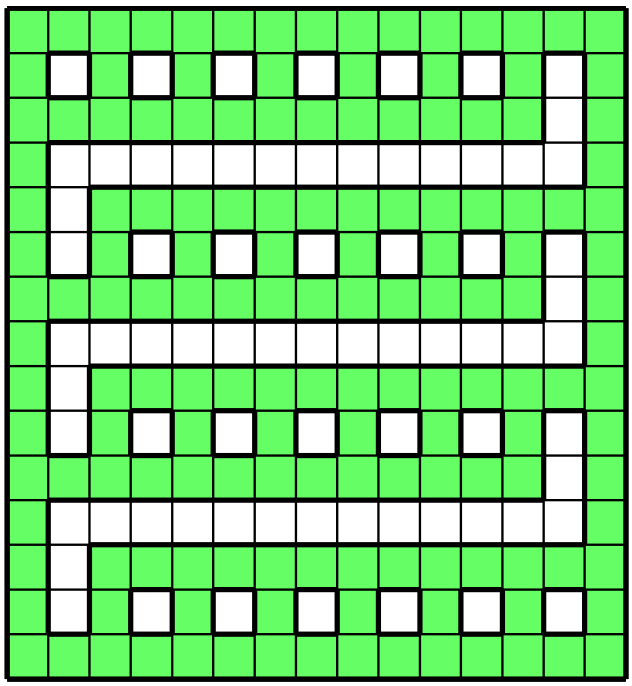

図 2.30: 再帰処理の深さと作業領域が $O(n^2)$ となる例.

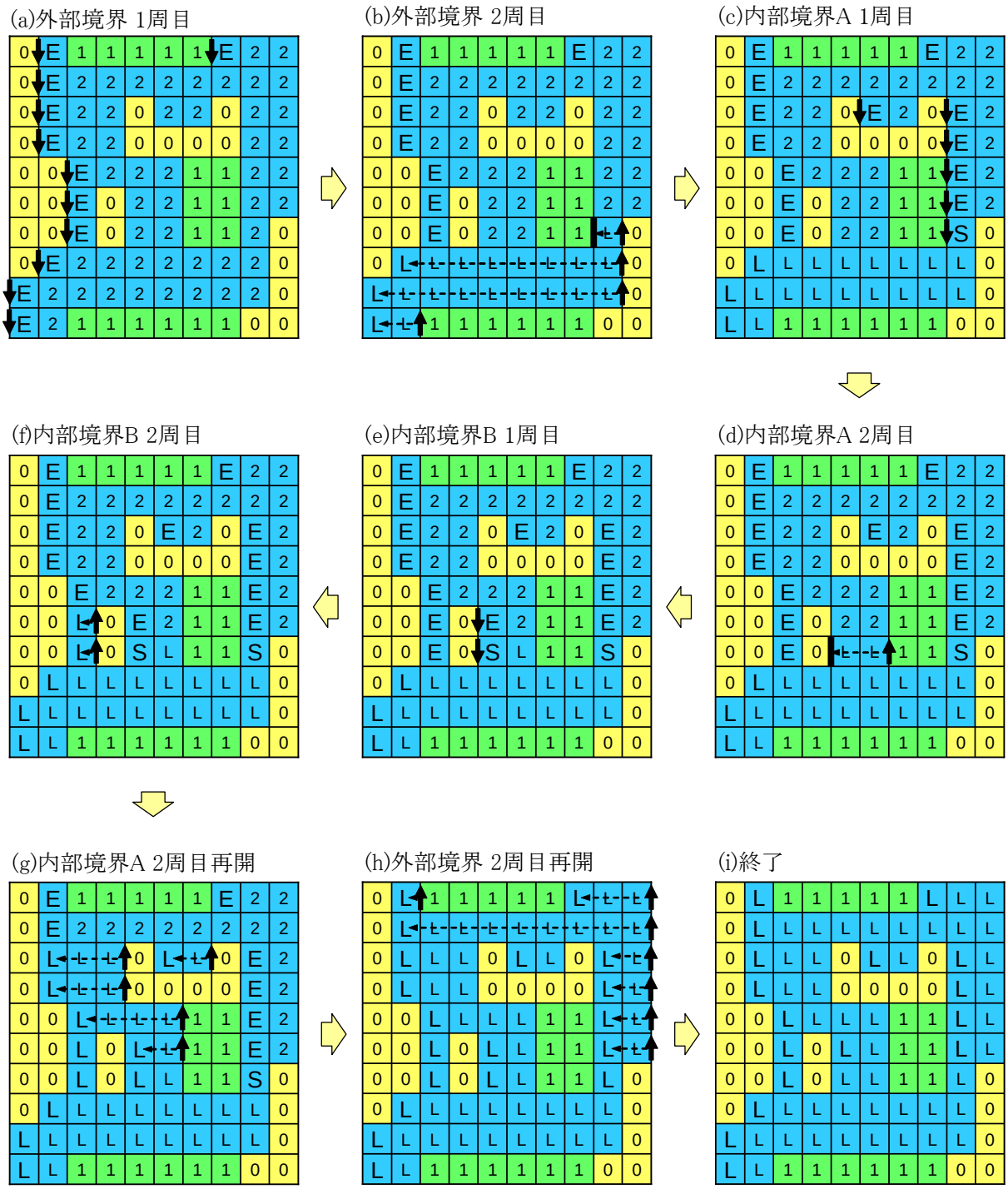

図 2.31: 連結成分の全ての要素に一度にラベルを付ける例．2 の要素全てにラベル L を付ける．

2.5.4 書き込み専用メモリでの実現

連結成分ラベル付け問題，及び連結成分数え上げ問題では，入力画像の配列に直接ラベルを書き込み，そのラベルを参照する事で問題を解くことが出来た．もし，書き込んだラベルを参照出来ないとき，これらの問題は解けるだろうか．もちろん定数作業領域での話である．

このような状況は，例えば連結成分数え上げ問題であれば，連結成分数を求めたあとに，まだ元のデータが必要な場合や，各要素が 1bit の 2 値画像で目印となるラベルを書き込めむ余地がない場合などが考えられる．連結成分ラベル付け問題であれば，多少強引な例ではあるが，元のデータはあとで使うので消してはいけないが，一方向にしか通信できない別の装置がラベル付けされた行列を必要としている場合が考えられる．さらに，これらはメモリ内で複製する余裕がないないほど大きな入力データであった場合であり，定数作業領域での処理を強いられるとする．

このような場合，入力画像には何も書き込む事が出来ず，定数作業領域のみで結果を出力しなければならない．連結成分数の出力は単に個数である．連結成分ラベル付け問題の出力は，座標とラベルをセットとし，要素の順序は問わないものとする．例えば，(X 座標，Y 座標，ラベル）という形で 1 要素ずつ出力する．同じ座標の要素を出力した場合，出力を受けた側で前のラベルは上書きするものとする．以上の条件で問題を解く方法を考える．

提案アルゴリズムでのラベルの機能は，同じ境界に対する多重処理を防止する働きが主であった．つまり，ラベルを使わずに各境界に対して，一度だけ処理する方法があれば，ラベルがなくても提案アルゴリズムを適用できる．よって，これ以降は，ラベルを使わずに各境界に対して一度だけ処理を行う方法について述べる．

まず，各境界で一度だけ処理を行う場合，境界のどの位置から開始するのが最適かを考える．恐らくそれは 2.3.1 で定義した最小辺である．最小辺は各境界に一つだけ存在し，一意に求めることができる．また最小辺の周囲の要素から，その境界が内部境界か外部境界か判断することもできる．そして，提案アルゴリズムの変更が最も少なくて済む．以上の点から最小辺を見つけた場合のみ境界に対する処理を行うことにする．

あとは，ラスタースキャンで見つけた境界辺が，最小辺かどうかだけ判断できれば，少なくとも連結成分数え上げ問題は解くことが出来る．最小辺かどうかは，境界追跡で判断できる．最小辺の定義は，境界中の下向き辺の中でラスター順が最も早いものである．つまり，最小辺より下に同一境界上の下向き辺は存在せず，また，同一行で最小辺より左にも同一境界上の下向き辺も存在しない．いま，ラスタースキャンで境界 B_i の下向き辺 $e_c(Bi)$ を見つけたとする．そのとき，この下向き辺から境界 B_i を追跡する．追跡の途中で，別の下向き辺 e_o を見つけたとき，e_c と e_o の座標を比較する．e_c と e_o の座標をそれぞれ，(x_c, y_c) ，(x_o, y_o) とすると $y_c > y_o$ or $(y_c = y_o$ and $x_c > x_o)$ であるとき，e_c は最小辺ではないと判断できる．最小辺でないと分かれば，境界を辿るのをやめ，ラスタースキャンに戻る．境界を辿るのが中断されないまま辺 e_c まで戻ってきた時， e_c は最小辺であるので，境界に対する処理を行う．

上のアルゴリズムは上手く動くが，図 2.32 のようなワーストケースでは $O(n^4)$ 時間かかってしまう．そこで，少し工夫し，境界追跡を，辺の向きに合わせた一方向ではなく，逆方向にも進むことにする．つまり，境界を双方向に一歩ずつ辿る．この場合のワーストケースは図 2.33 であり，$O(n^2 \log n)$ 時間で済む．この操作は，辿っている境界辺の位置と向きなどの定数作業領域で実現できる．

連結成分数え上げ問題に対しては，上のアルゴリズムを元に提案アルゴリズムを修正することで，元の入力データに書き込みを行う事無く，定数作業領域，$O(n^2 \log n)$ 時間で解く事ができる．

連結成分ラベル付け問題に対しては，上のアルゴリズムを適用しただけでは内部境界を見つけたときに用いるべきラベルを知る事ができず，上手く動かない．しかし，2.5.3 で紹介した二つ目のアルゴリズム，同一連結成分の全要素に一度にラベルを付ける方法を応用すれば，ラベルを知る必要はなくなる．このアルゴリズムでは，ラベルを付ける方向が左右逆になるため，内部境界に対する最小辺の定義も左右逆にし，境界の下向き辺のうち，最下段の一番右の辺に変更する．内部境界の処理に移るのは内部境界の最小辺にぶつかったときのみである．以上の点さえ気を付ければ，提案アルゴリズムの修正は問題なく行えるはずである．よって，連結成分ラベル付け問題についても，元の入力データに書き込みを行うことなく，定数作業領域，$O(n^2 \log n)$ 時間で解く事ができる．

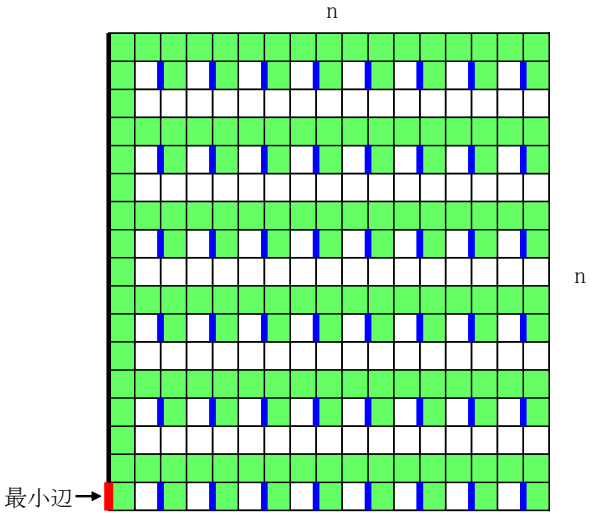

図 2.32: 一方向に境界追跡する最小辺判断方法でのワーストケース．太い辺が下向き辺．各下向き辺が最小辺でないと判断できるのは一番左の下向き辺に対してのみ．下向き辺から一番左の下向き辺までの道のりの平均は面積に比例した $O(n^2)$ で，下向き辺の数も面積に比例した $O(n^2)$ である．そのため，全体では $O(n^4)$ 時間となる．

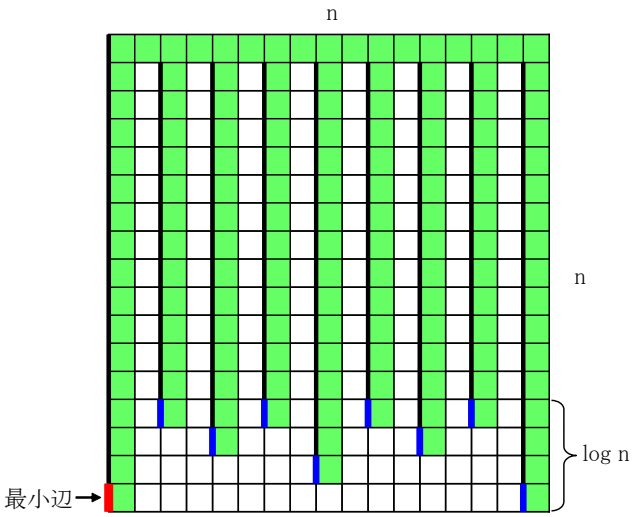

図 2.33: 双方向に境界追跡する最小辺判断方法でのワーストケース．太い辺が下向き辺．縦に連なった要素の先の各下向き辺からは双方向にたどっても，時間がかかる．そのような下向き辺の数は $O(n)$ であり，それらの Y 座標でレベル分けすると，各レベルの距離の合計は $O(n^2)$ となる．レベルは $O(\log n)$ 段階であるため，全体では $O(n^2 \log n)$ 時間となる．

2.5.5 形状特徴パラメータの同時抽出

連結成分ラベル付けは画像解析によく使われる．例えば，解析対象画像 → 2 値化 → 連結成分ラベル付け → 連結成分毎の形状特徴パラメータの計算，と言う順序で解析を進める．つまり，連結成分ラベル付けは，連結成分毎の形状特徴パラメータを計算するための下準備であることが多い．提案アルゴリズムでは，境界を辿るという独特な手法を用いていることから，FILL アルゴリズムなどの他の連結成分ラベル付けアルゴリズムと比較し，連結成分ラベル付けと平行して求められるパラメータが多く，形状特徴パラメータを求める工程の多くを削減する事が出来る．

形状特徴の主要パラメータには以下のようなものがある．(図 2.34 参照)

外接矩形
: 連結成分に接する最小の長方形．

穴数 連結成分にある穴の数．つまり内部境界の数．

面積 連結成分の要素数．外部境界に囲まれた領域の面積（全面積），内部境界に囲まれた領域の面積（穴面積），全面積から穴面積を除いた面積（純面積）がある．

周囲長
: 連結成分の周囲の長さ．外周囲長と内周囲長があり，内周囲長は穴に対する周囲長の総和．

重心 連結成分の各要素に等しい重さがあると仮定したときの重心の座標．

凸包 連結成分の全ての要素を含む最小の凸多角形．

半径 外周上の任意の要素と重心との距離のうち最大のものを最大半径，最小のものを最小半径と呼ぶ．

絶対最大長
: 外周上の任意の 2 要素間の距離の最大長．

慣性モーメント角度
: 慣性モーメントの角度．連結成分の傾き．

モーメント射影距離
: 連結成分を慣性モーメント角度で見た時の縦横の長さ．

他にも様々なパラメータがあるが，それらの多くは上記のパラメータから算出できる．

連結成分ラベル付けと平行して求められるパラメータとその効率の良い方法について述べる．その際ラベル付けには 2.5.3 で紹介した二つ目のアルゴリズム，同一連結成分の全要素に一度にラベルを付ける方法を用いると仮定する．

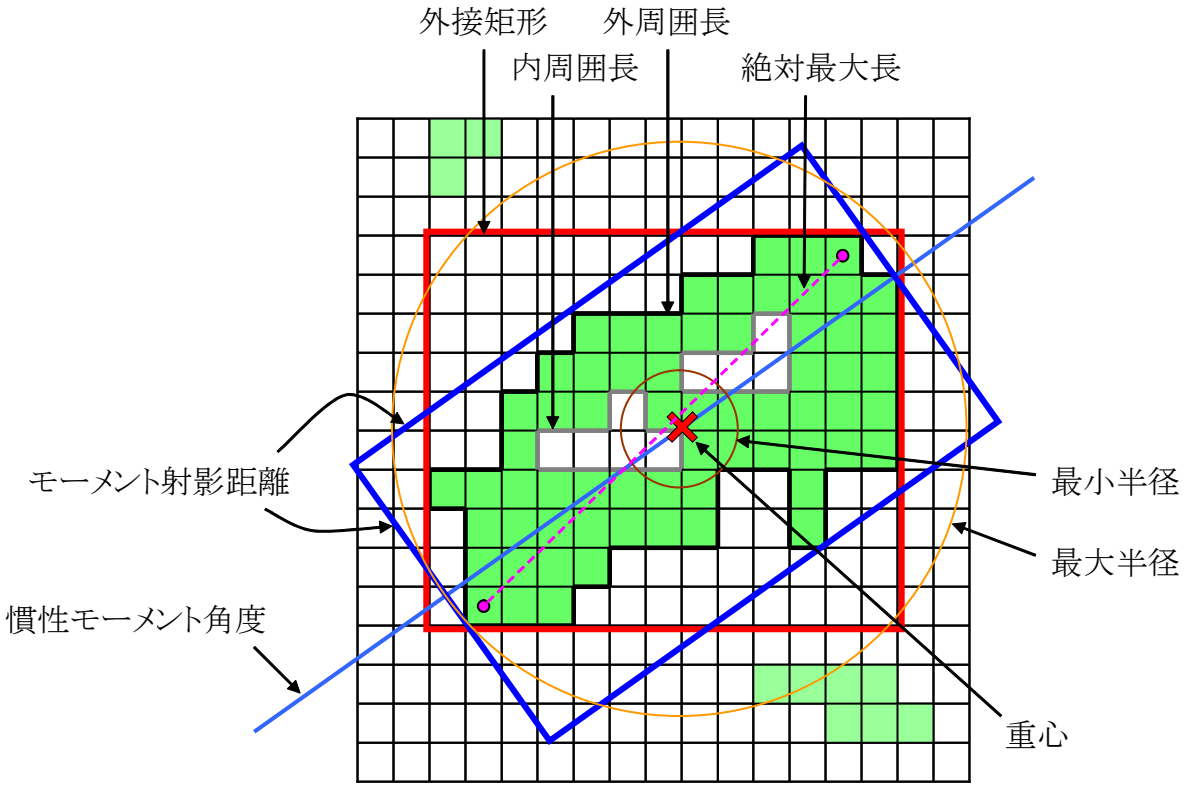

図 2.34: 連結成分の主要パラメータ.

外接矩形はX軸の最大値と最小値，Y軸の最大値と最小値を記録する4つの変数があれば線形時間で実現できる．外部境界を辿りながら，要素の座標と上記の4つの変数とを比較しながら，最大値最小値を得ればよい．領域内の全要素と比較する場合に比べ，外部境界に接している要素だけを比較するので効率が良い．さらに，最小辺から始めるため，最初からY軸の最小値は確定する．また，辺の向きを考えると，毎回更新される可能性のある値は一つだけなので，一度の比較で済む．つまり，右に進んだときはX軸の最大値，上に進んだときはY軸の最大値，左に進んだときはX軸の最小値と比較するだけでよく，下に進んだときは何もする必要はない．

穴数は，単純に内部境界の最小辺をカウントするだけでよい．

面積の求め方は2通り考えられる．一つは各要素にラベルを付けた回数をカウントしていく方法である．簡単であるが，この方法では純面積しか求められない．もう一つの方法は，効率よく，全面積，穴面積，純面積を求められる．境界 B_i を辿りながら，変数 a に B_i 内の面積を求めるとする．まず，a を0に初期化する．境界辺が上向きであったとき，辺のX座標を a に加算する．境界辺が下向きであったときは辺のX座標を a から減算する．これだけである．B_i が外部境界の場合，a は正の値になり，B_i が内部境界の場合，a は負の値になる．それぞれの絶対値が境界 B_i 内の面積である．(図2.35) 外部境界からのみ計算した結果が全面積であり，内部境界から計算した結果の合計の絶対値が穴面積，全ての境界から計算した結果が純面積となる．ラベルを付ける毎に計算する必要がなく，また，全面積であれば外部境界を一度辿るだけで効率的に求められる．

周囲長は境界追跡を行いながら，簡単に求められるので省略する．

重心を求めるには各要素の座標の平均値をとる．連結成分 C_i に対する重心 G_i は $p \in C_i$ に対して，$(\frac{(\sum p.x)}{|C_i|}, \frac{(\sum p.y)}{|C_i|})$ として与えられる．重心の計算も面積の場合と同様，2通りの方法が考えられる．一つは各要素にラベルを付けるタイミングで，各座標を合計値に加算し，ラベルが付け終わった時に面積（=要素数）で割り，平均を求める方法である．もう一つの方法も面積を求めた時と似た方法を取る．境界 B_i を追跡しながら，変数 c_y に B_i 重心のY座標を求めるとする．まず，c_y を0に初期化する．境界辺が上向きであったとき，c_y に (辺の Y 座標 $\times$ 辺の X 座標) を加算する．境界辺が下向きであったときは逆に c_y から (辺の Y 座標 $\times$ 辺の X 座標) を減算する．この方法で境界を辿り終えたとき，c_y には各要素のY座標の合計が入っている．よって，c_y を面積（=要素数）で割ってやれば，重心のY座標を求める事ができる．重心のX座標 c_x を求める場合も同様で，左向き辺のとき，c_y に (辺の X 座標 $\times$ 辺の Y 座標) を加算する．右向き辺のとき，c_y から (辺の X 座標 $\times$ 辺の Y 座標) を減算する．ラベルを付ける毎に計算する必要がなく，また，内部境界を考慮しないのであれば外部境界を一度辿るだけで効率的に求められる．

凸包は，外部境界を辿りながら，境界上の要素に対し，スタックを用いて簡単に計算できる．一般的な凸包の計算と異なり，外部境界上の要素の順序関係は凸包でも保存されるため，外周囲長に対する線形時間で計算できる．外周囲長は $O(n^2)$ である．また，凸包の要素となりえるのは，外部境界の追跡時に左折した所の要素のみであるのでさらに計算を省くことができる．凸多角形を構成する要素は上下の無限遠方から見たときに各列に一つずつあるのが最大であるため高々$2n$ である．よって，スタックの領域は $O(n)$ である．スタックに最後まで残った要素が凸多角形を構成する要素であり，スタックの内容がそのままパラメータとなる．凸包は他の計算の高速化にも使える．

半径は，外部境界を追跡しながら重心との距離を求めて行けば，外周囲長に対する線形時間で求める事ができる．特に最大半径の場合であれば，凸包に含まれる要素に対して重心との距離を調べるだけでよく， $O(n)$ 時間で求められる．

絶対最大長は外周囲上の 2 点間の最大距離である．最大長となる 2 点になりえるのは，凸包に含まれる要素のみであるため，絶対最大長を求める問題は凸包の直径を求める問題に還元できる．凸包の直径を求める問題は要素数に対する線形時間でできるキャリパー法というアルゴリズムが知られている．

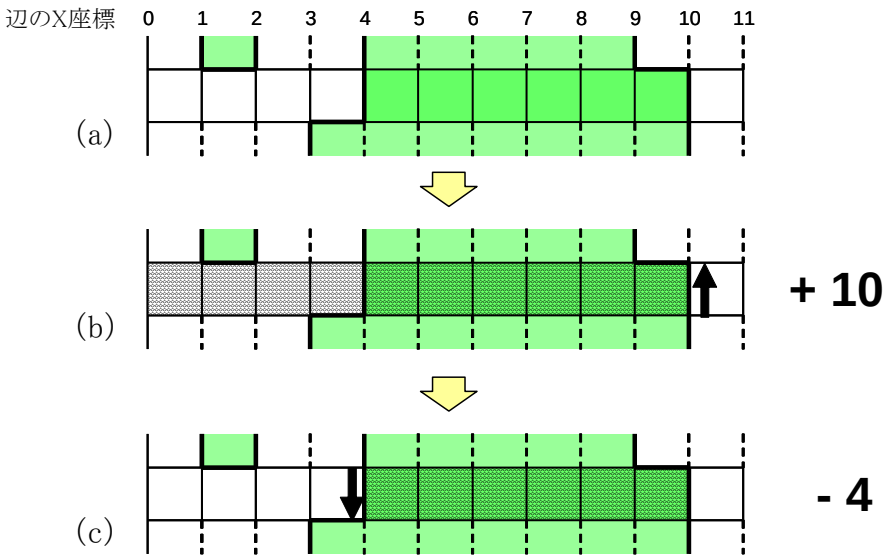

図 2.35: 面積の計算方法.（a）のような外部境界の一部分があったとする．境界を辿ると，（b）で上向き辺に出会い，辺の X 座標を面積に加算する．このとき，網掛け部分が面積として余分な部分まで加算されていることになるが,（c）のように反対側にある下向き辺に出会ったとき，辺の X 座標を面積から減算するため，正しい面積が求められる．内部境界のときは，辺の関係が逆になるため，負の値となる．

慣性モーメント角度を求めるには，分散と共分散が必要となる．連結成分 C_i の慣性モーメント角度を求める時，C_i の重心を G_i とすると $p \in C_i$ に対して，

$$\begin{aligned} \text{分散}.x &= \frac{\sum p.x^2}{|C_i|} - G_i.x^2 \\ \text{分散}.y &= \frac{\sum p.y^2}{|C_i|} - G_i.y^2 \\ \text{共分散} &= \sum((p.x - G_i.x) * (p.y - G_i.y)) \\ \text{慣性モーメント角度} &= \arctan \frac{\text{共分散}}{\text{分散}.x - \text{分散}.y} \end{aligned}$$

となる．分散は，二乗に変わる事にさえ気を付ければ重心の場合と同じ方法で求められる．共分散に対しては，重心が分かっていない段階でも $\sum(p.x \times p.y)$ ，$\sum p.x$ ，$\sum p.y$ の 3 つを調べておけば，それを元に，重心が分かった時に計算可能である．$\sum p.x$ 及び $\sum p.y$ については，重心を求める際の面積（=要素数）で割る前の値と同じである．$\sum(p.x \times p.y)$ に大しても，境界を辿りながら，上向き辺の時は $(p.x \times p.y/2)$ を加算し，下向き辺の時は $(p.x \times p.y/2)$ を減算すればよい，よって，共分散も境界を一度辿るだけで求めらる．慣性モーメント角度は，重心の計算と同時に行えば，境界を一度辿るだけで求める事ができる．

モーメント射影距離は慣性モーメント角度と凸包が分かれば，凸包の要素数に対する線形時間で求められる．

これら全てのパラメータは線形時間で計算でき，パラメータとしての領域を除けば，定数作業領域で実現できる．半径，絶対最大長，モーメント射影距離には別のパラメータ，凸包，重心，慣性モーメントを用いる事で，連結成分へのアクセスは，ラベル付け時の 1 回だけでこれらのパラメータを求めることができる．また，この方法は，ラベルを付けない場合にもその成分のパラメータを求める事ができる．成分にラベルを付ける事が出来ない場合は，2.5.4 を述べた方法を用いて $O(n^2 \log n)$ 時間で計算できる．ラベルは不要だが，一次的にラベルを付ける事が許される場合は，目印となるラベルを用いて計算したあと，目印を元に戻す方法で線形時間（$O(n^2)$ 時間）で計算する事ができる．

第3章　ユークリッド距離変換問題

この章では，画像処理の中でも基本的な問題の一つであるユークリッド距離変換問題に関して述べる．この問題は2値画像の各1の要素から，直線距離でもっとも近い0の要素までの距離を求める問題である．例えば画像認識の基礎的な処理として利用される．既存のアルゴリズムでは画像の面積に依存した作業領域が必要である．ここでは既存のアルゴリズムを改良した，定数作業領域で実現できるアルゴリズムを提案する．

3.1　問題の定義

$n \times n$ の画像（もしくは2次元配列）G_n があり，各要素 $G(x,y)$ は 0 か 1 の2値で構成される．[1]この時，0の要素を **0-pixel**，1の要素を **1-pixel** と呼ぶ．

ユークリッド距離とは，平面上の直線距離であり，任意の二つの要素 $(x,y),(x',y')$ の距離 $d((x,y),(x',y')$ は以下のように書ける．

$$d((x,y),(x',y')) = \sqrt{(x-x')^2 + (y-y')^2}$$

ユークリッド距離変換問題は，0 と 1 からなる2値画像 G_n が与えられた時に，各1-pixelに対し，その要素から最も近い 0-pixel までの距離（の2乗）で置き換えた2次元配列 D_n を出力する問題である．

例えば図3.1では，(a）のような2値画像が入力として与えられた時，各 1-pixel から矢印で表されるような最も近い 0-pixel までの距離を求める．ユークリッド距離の定義では距離は（b）のようになるが，平方根付きのデータは扱い難いため，一般的に（c）のように，距離を2乗して整数化した値を扱う．つまり二つの要素 $(x,y),(x',y')$ の距離はデータ上では $(x-x')^2+(y-y')^2$ で表される．整数化しても大小関係は変わらず，平方根の計算も不要になるという利点がある．

[1]この論文内では，簡単の為，この問題に対しても入力サイズを $n \times n$ の正方形に固定して論じるが，長方形であっても本質な違いはない．

図 3.1: ユークリッド距離変換の例．入力（a）の各 1-pixel から最も近い 0-pixel（矢印）までの距離を求める．実際の距離は（b）であるが，計算機上では扱い易さのため，2 乗して平方根を取った値を出力とする（c）．

3.2 既存のアルゴリズム

ユークリッド距離変換問題は画像処理において重要な問題であったが，線形時間, すなわち要素数に比例した $O(n^2)$ 時間で計算可能かどうかは長い間未解決であった．しかし，1995 年に Kirkpartrick と Brew[2] が，1996 年に Hirata[3] が線形時間のアルゴリズムを開発した．それぞれ異なる方法で実現しており，Kirkpartrick らの方法ではボロノイ図の概念を用い，Hirata の方法では放物線の下側包絡線を用いている．

ここではまず，自然な手法を紹介したあと，提案するアルゴリズムと深い関係のある Hirata のアルゴリズムを紹介する．

3.2.1 自然な手法

ユークリッド距離変換を行う最も素朴な方法は，各 1-pixel から範囲を波紋のように広げながら幅優先探索を行い，0-pixel を探す方法である．実現は簡単にできるが，図 3.2 のような画像の場合，各 1-pixel からほぼ全ての要素を調べることになる．要素数は n^2 なので $O(n^4)$ 時間かかることになる．この方法では定数作業領域で実現する事ができ，さらに入力画像が読み込み専用であったとしても各要素に対応する距離を求めることができる．

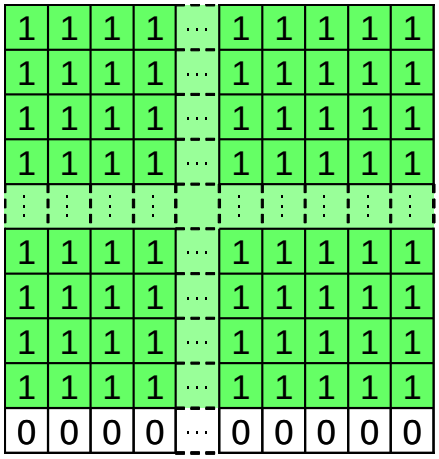

図 3.2: $O(n^4)$ 時間かかる入力画像の例.

3.2.2 Hirata のアルゴリズム

Hirata のアルゴリズムは放物線の下側包絡線を利用して実現される．アルゴリズムは 3 つのステップに分けられる．STEP 1 では各要素から垂直方向に最も近い 0-pixel までの距離を求める．STEP 2 では，STEP 1 で求めた距離を放物線ととらえ，スタックを用いて各行の下側包絡線を求める．STEP 3 では，STEP 2 で求めた下側包絡線から各要素のユークリッド距離を求める．STEP 2 と 3 は行毎に繰り返す．アルゴリズムを Algorithm7 に示す．

STEP 1 では，各列を上下方向に 2 回走査し，各要素から垂直方向で最も近い 0-pixel までの距離の行列 D を作る．走査開始前に最も近い 0-pixel までの距離 d を ∞ と初期化する．走査中の要素 $G(x_c, y_c)$ が 0-pixel の場合，距離 d を 0 にし．1-pixel の場合は d に 1 ずつ加算する．逆方向にも同じ走査を行い，2 回の走査で得られた数値のうち小さいほうが，求める垂直距離となっている．(図 3.3)

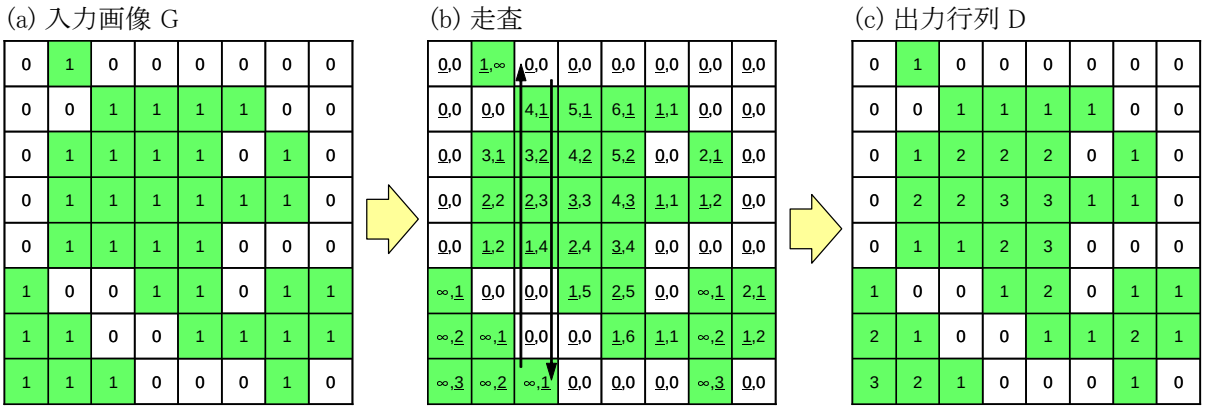

図 3.3: Hirata のアルゴリズム STEP 1. (a) を入力とした時, (b) のように各列を上下方向に走査し，各要素から垂直方向で最も近い 0-pixel までの距離（下線付き数字）を求め，(c) のような行列 D を出力する．

STEP 2 と 3 では，STEP 1 で求めた行列 D を元に，各行において各要素から最も近い 0-pixel までのユークリッド距離を求める．

1-pixel p_{ij} に最も近い 0-pixel を p_{kl} とすると，その距離は $d_{ij} = \sqrt{(i-k)^2 + (j-l)^2}$ で与えられる．定義より，要素 (i, j) を中心に半径 d_{ij} の円を描けばそれより内側の要素はすべて 1-pixel でなければならない．p_{ij} と同じ行にあり，p_{kl} と同じ列にある要素 p_{il} も円内にあるので必ず 1-pixel となる．(図 3.4) また， p_{kl} は p_{il} から垂直方向で最も近い 1-pixel である．そうでなければ，円の内部に 0-pixel が存在する事になり，定義と矛盾する．したがって，要素 p_{ij} に最も近い 0-pixel は，p_{ij} と同じ行の各要素から垂直方向に最も近い 0-pixel の集合に含まれる．つまり， 要素 p_{ij} から最も近い 0-pixel ユークリッド距離は，STEP 1 で求めた行列 D の i 行を調べるだけで求める事が出来る．

要素 p_{ij} の垂直距離が d のとき，最も近い 0-pixel は $p_{i-d,j}$ または $p_{i+d,j}$ である．その要素から i 行の他の要素 p_{ik} までの距離は図 3.5（a）に示すように $\sqrt{(k-j)^2+d^2}$ となり，距離を y 軸にとると，i 行の各要素までの距離は $y=\sqrt{(x-j)^2+d^2}$ となる曲線で表される．簡単のため，図 3.5（b）のように平方根をとりのぞき $y=(x-j)^2+d^2$ となる放物線に変換する．平方根は単調増加関数であるから大小関係は保たれる．よって，i 行の全ての要素について，最も近い 0-pixel までの距離を用いて上述の放物線を描き，各要素からそれらの放物線までの垂直距離の最小値を求めれば，それが求める距離変換の値となっている．この時必要となるのは距離の最小値であるため，放物線の集合の下側包絡線が分かれば良い．

図 3.6 は行の処理の例を示したものである．STEP 1 で求めた垂直方向の距離から放物線を定め，その下側包絡線までの垂直距離を求めればよい．

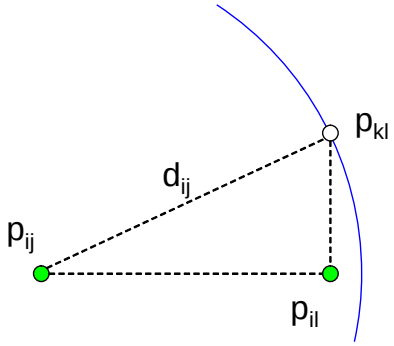

図 3.4: 1-pixel P_{ij} に最も近い 0-pixel P_{kl}．円内の全ての要素は 1-pixel であり，P_{kl} は P_{il} から垂直方向で最も近い 0-pixel である．

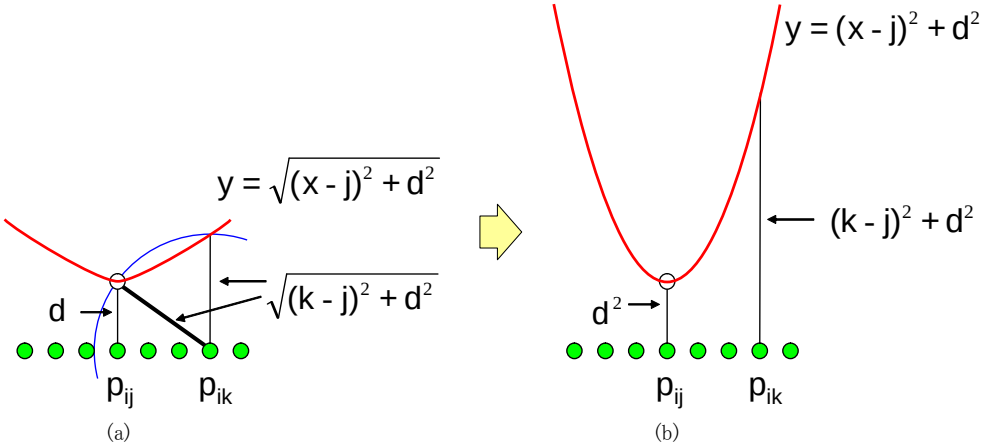

図 3.5: 要素 P_{ij} に最も近い 0-pixel と i 行の他の要素 p_{ik} との距離の関係．(a) は実際の距離と，距離を y 軸に取った時の変化．(b) は平方根を取り除いた距離の変化．必ず $y=x^2$ の放物線と同形になる．

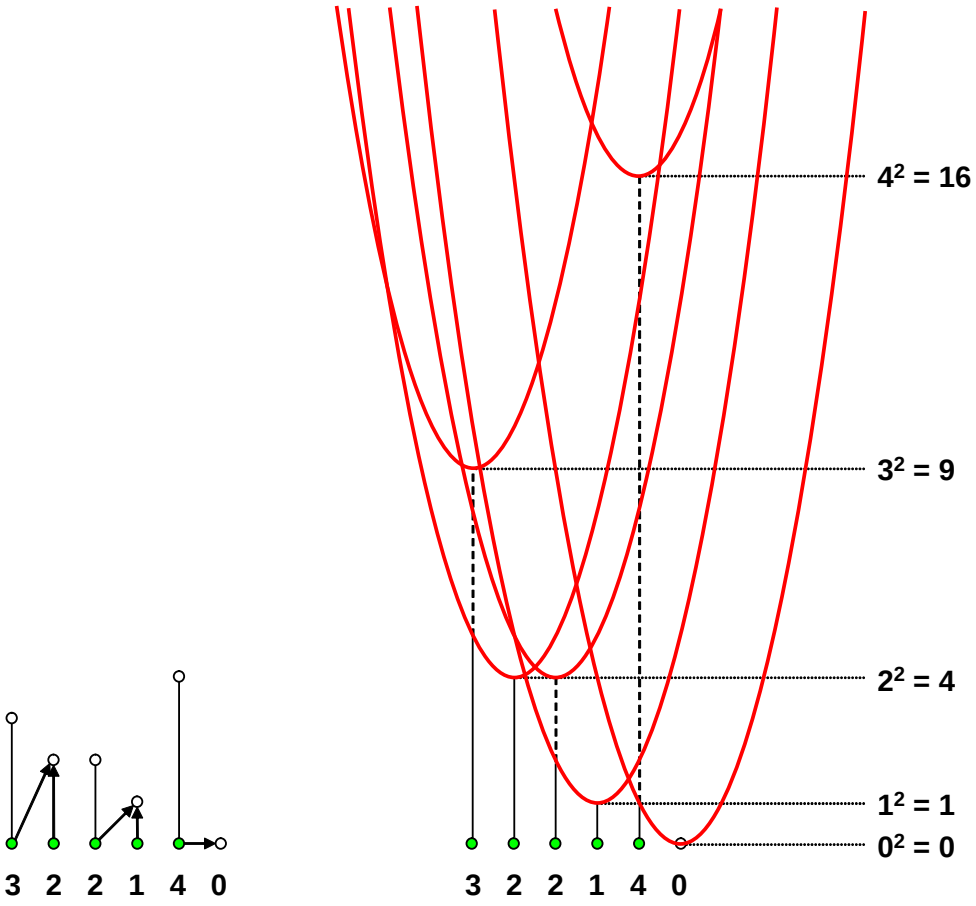

図 3.6: 行の処理の例．左は縦軸に垂直方向の 0-pixel までの距離をとり，矢印で各要素から最も近い 0-pixel を表す．右は放物線に変換したもの．各要素から垂直に見たとき，最も近い 0-pixel の放物線が一番下に来る．

STEP 2 の処理である，放物線の下側包絡線を線形時間で求める方法について述べる．**下側包絡線**とは，図 3.7 のような，放物線の集合に対して下方無限遠から見える部分の事である．

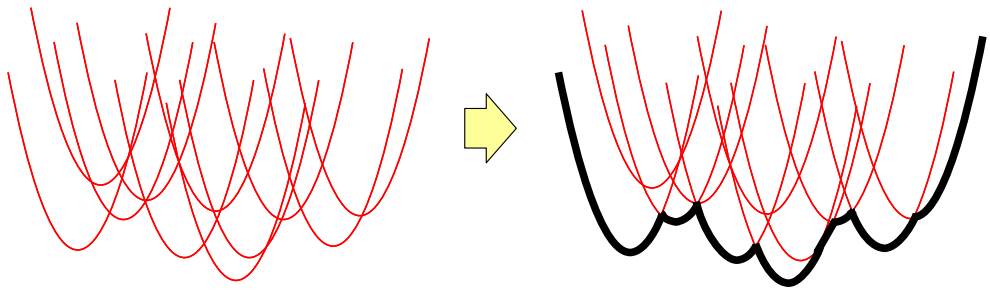

図 3.7: 放物線の集合に対する下側包絡線．

放物線 P_i は $P_i : y = (x - x_i)^2 + y_i$ で定義され，行列 D の各行から得られる放物線の集合 $P_1, P_2, \ldots, P_n$ は x_i の順にソートされている．すべての放物線は形が同じであるので，どの二つの放物線 P_i, P_j をとっても下方無限遠から見ると，放物線 $P_i \rightarrow P_i$ と P_j の交点 $\rightarrow$ 放物線 P_j ，の順に見える．つまり，交点は一つだけ存在し，一つの放物線が下側包絡線に複数部分に現れる事はない．したがって，下側包絡線は放物線の名前の系列として完全に表現できることになる．P_i までの放物線に対する下側包絡線を表す系列を $(P'_1, P'_2, \ldots, P'_{m_i})$ とし，二つの放物線 P_i, P_j の交点を $c(P_i, P_j)$ とする．

放物線を左から順に処理し，下側包絡線を表す系列を構成する．最初の放物線 P_1 の時点では下側包絡線は (P_1) である．いま，P_i までの放物線に対する下側包絡線 $(P'_1, P'_2, \ldots, P'_{m_i})$ が求まっているとして，これに次の放物線 P_{i+1} を加えることを考える．P_{i+1} は最も右にあるので必ず下側包絡線に含まれる．問題は，今までの下側包絡線上にあった放物線の中で，P_{i+1} によって隠されてしまうものがあるときであり，その場合，それらの放物線は下側包絡線の系列から削除しなければならない．一つの放物線が下側包絡線の複数部分に現れる事がなく，P_{i+1} が必ず下側包絡線に含まれることから，下側包絡線上の右の放物線から削除される．よって，下側包絡線上の一番右の放物線 P'_{m_i} が P_{i+1} によって隠されるかどうかを判定できれば，隠されない放物線が見つかるまで処理を繰り返して P_{i+1} の追加処理を正しく行える．

判定には交点 $c(P'_{m_i-1}, P'_{m_i})$ と追加する放物線 P_{i+1} を比較する．放物線 P_{i+1} が交点 $c(P'_{m_i-1}, P'_{m_i})$ より上を通るとき， P'_{m_i} は下側包絡線上に残る．放物線 P_{i+1} が交点 $c(P'_{m_i-1}, P'_{m_i})$ 以下を通るとき， P'_{m_i} は削除される．(図 3.8)

この操作には複雑なデータ構造は不要で，単なるスタックを下側包絡線を表す放物線の系列として用いることで実現できる．計算時間に関してはスタックの操作回数で評価できる．PUSH は各放物線に対して一度しか実行しないし，一度 POP して削除された放物線は 2 度と削除されることはない．よって，放物線の数を n とすると，スタックの操作回数は $2n$ 回以下になる．すなわち，ソート済みの放物線の集合から下側包絡線を求めるのには，放物線の数に対して線形時間で実現できる．作業領域に関しては，下側包絡線の系列を表すスタックの領域が必要となる．下側包絡線上の放物線が一つも削除されないケースがあるため，スタックの領域も放物線の数に対して線形である．

STEP 3 では，STEP 2 で求めた下側包絡線を用いて，各要素から一番近い 0-pixel へのユークリッド距離を求める．STEP 2 では下側包絡線を示す放物線の系列をスタックを用いて左から順に求めた．つまり，スタックの一番上の要素は系列中で最も右にある放物線である．STEP 3 では線形時間で処理するため，不要になった放物線を削除（POP）する必要がある．よって，スタックのデータ構造上，今度は右の要素から順に処理を行う．

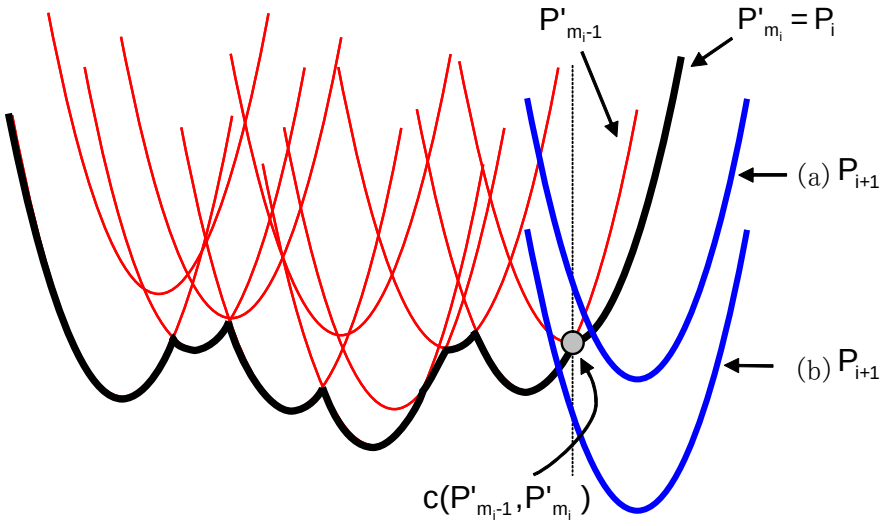

図 3.8: 現在までの下側包絡線の系列に次の包絡線 P_{i+1} を加える時の状況．(a) では $c(P'_{m_i-1}, P'_{m_i})$ の上を通っているので， P'_{m_i} は下側包絡線上に残る．(b) では $c(P'_{m_i-1}, P'_{m_i})$ の下を通っているので， P'_{m_i} は削除され，下側包絡線上には残らない．

図 3.9 のような下側包絡線の放物線系列を右端から順に見ていくとする.（a）の位置では下側包絡線を表す放物線は P_j であり，交点 $c(P_i, P_j)$ は（a）より左側にある．このまま左に進み,（b）の位置まで来た時，下側包絡線を表す放物線が P_j から P_i にかわる．同形放物線の下側包絡線では，同一の放物線が複数部分で現れることはないことから，これより先に P_{j} が現れることなく，P_{j} は不要である．つまり,（b）や（c）のように交点 $c(P_i, P_j)$ が同じか右に来た時，$c(P_i, P_j)$ より右にある放物線は削除する．

計算時間に関して，スタックの操作で評価できる．PUSH は行われる事はなく，POP して削除された放物線は 2 度と削除されることはない．よって，下側包絡線の系列数分だけスタック操作行われる．下側包絡線の系列数はその行の要素数以下なので，その行の要素数に対して線形時間で処理できる．

以上の操作により，各要素から最も近い 0-pixel までのユークリッド距離を全て求められた．この方法では，すでに示したように，全ての処理が線形時間 $O(n^2)$ で実行できる．作業領域は，行列 D に $O(n^2)$ 領域とスタックに $O(n)$ 領域必要となるため，全体の作業領域は $O(n^2)$ 領域となる．

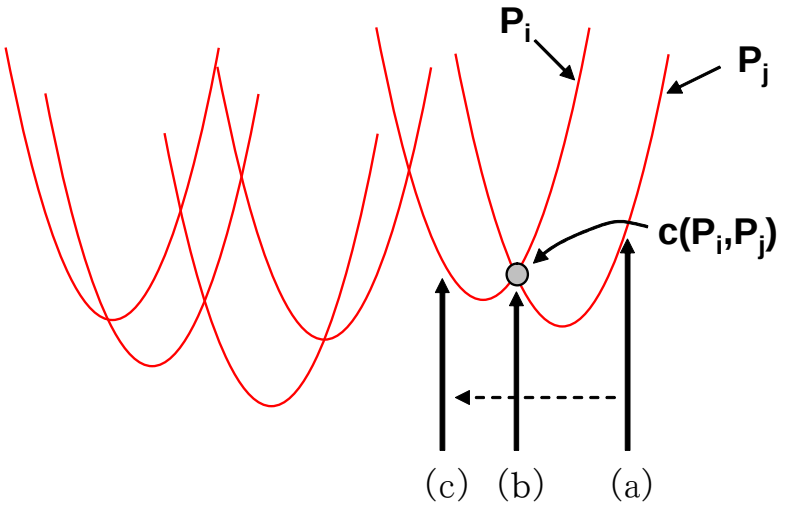

図 3.9: 下側包絡線上の放物線の系列例.（a）の位置からは交点 $c(P_i, P_j)$ は左側にあるが,（b）や（c）の位置からは交点 $c(P_i, P_j)$ は同じか右にある.（b）以降では下側包絡線上に放物線 P_j は現れないため，交点 $c(P_i, P_j)$ を越えた時に P_j をスタック上から削除する．

Algorithm 7: Hirata のアルゴリズム

Input: $n \times n$ の 2 値行列 G

```
//— STEP 1 —//
for x_c = 0 to n − 1 do
    d = ∞
    for y_c = 0 to n − 1 do
        if G(x_c, y_c) > 0 then
            d = d + 1
        else
            d = 0
        D(x_c, y_c) = d
    d = ∞
    for y_c = n − 1 to 0 do
        if G(x_c, y_c) > 0 then
            d = d + 1
        else
            d = 0
        D(x_c, y_c) = min(D(x_c, y_c), d)

for y_c = 0 to n − 1 do
    //— STEP 2 —//
    スタックの初期化
    スタックに (0, D(0, y_c)) と (1, D(1, y_c)) を PUSH
    for x_c = 2 to n − 1 do
        repeat
            (X_s, Y_s) = スタックの上二つの要素の放物線の交点
            if Y_s ≥ (X_s − x_c)^2 + D(x_c, y_c)^2 then
                スタックを POP しての一番上の要素を削除
        until Y_s < (X_s − x_c)^2 + D(x_c, y_c)^2
        スタックに (x_c, D(x_c, y_c)) を PUSH
    //— STEP 3 —//
    for x = n − 1 to 0 do
        repeat
            (X_s, Y_s) = スタックの上二つの要素の放物線の交点
            if X_s ≥ x_c then
                スタックを POP しての一番上の要素を削除
        until X_s < x_c or スタックの要素数 = 1
        (x_t, g_t) = スタックの一番上の要素
        D(x_c, y_c) = (x_c − x_t)^2 + g_t^2
```

3.3 提案するアルゴリズム

3.2 でアルゴリズムを二つ紹介した．一つは $O(n^4)$ 時間，$O(1)$ 領域の自然なアルゴリズム．もう一つは $O(n^2)$ 時間，$O(n^2)$ 領域の Hirata のアルゴリズムである．これら二つの欠点を補い， $O(n^2)$ 時間，$O(1)$ 領域で実現できる Hirata のアルゴリズムを提案する．

提案するアルゴリズムは Hirata のアルゴリズムを改良したものとなっている．そのため，大まかな流れは同じであり，作業領域に影響が出る箇所に変更を加え，定数作業領域で実行可能なアルゴリズムを実現している．

まず，Hirata のアルゴリズムで用いられる作業領域，配列 D について考える．配列 D は最終的な出力結果としても用いられる．つまり，最終出力の時点で D の各要素には，各要素から最も近い 0-pixel までのユークリッド距離が入っている．ユークリッド距離の取りえる範囲は，0 から対角線の最大距離である $\sqrt{2(n-1)^2}$ であり，配列 D のデータ上では 0 から $2(n-1)^2$ の範囲となる．(図 3.10)

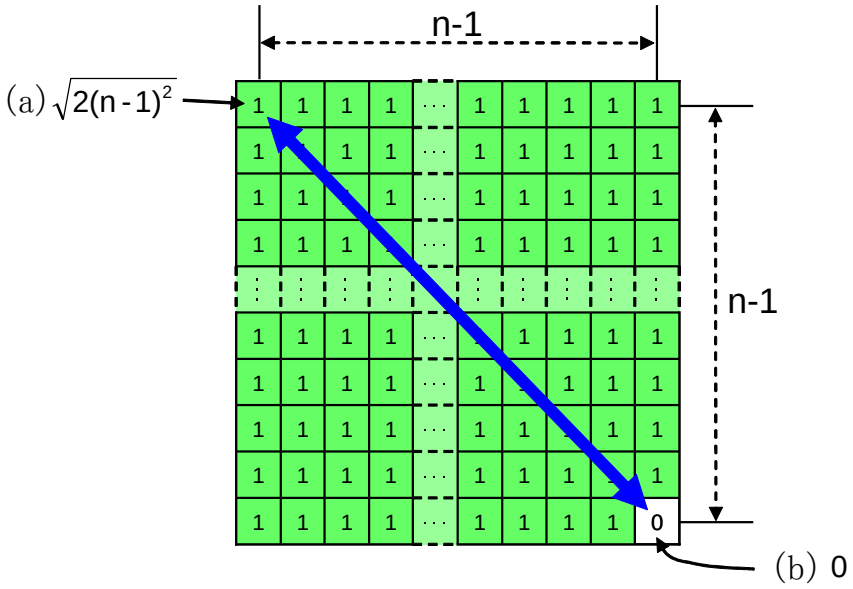

図 3.10: ユークリッド距離の取りえる範囲. (b) の要素からはそれ自身が最も近い 0-pixel なので 0. (a) から最も近い 0-pixel は (b) なので，ユークリッド距離は $\sqrt{2(n-1)^2}$ であり，配列 D のデータ上では 0 から $2(n-1)^2$ の範囲となる．

仮に，入力 G の各要素に 0 から $2(n-1)^2$ のまでの範囲を入れる余裕があれば Hirata のアルゴリズムの STEP 1 で，D の代わりに G をそのまま使う事が出来る．実現は問題なく行える．0-pixel であった要素はそのまま 0 であり，1-pixel は 1 以上の値になる．そのため，0 かどうかを見れば入力時点で 0-pixel であったか 1-pixel であったかを判断できる．

このように入力 G をそのまま出力にも用いることで，作業領域の行列 D を削減できる．問題となるのはスタックの領域である．提案アルゴリズムでは，スタックを G の領域に埋め込む事でスタックの領域を削減し，定数作業領域を実現する．当然，単純な方法では埋め込むことはできずいくつかの問題をクリアしなければならない．

G の領域をスタックとして利用するにあたり，各要素には距離の最大値である $2(n-1)^2$ より大きな値が入ることは期待すべきではない．逆に言えば， $2(n-1)^2$ 以下の値であれば入れる事が出来る．Hirata のアルゴリズムでは，スタック上に下側包絡線を表す放物線の系列を置いた．全ての放物線は同形であり，横軸の位置 x_s と，垂直方向に一番近い 0-pixel までの距離 d_s が分かれば各放物線を求める事ができる．そのため，スタックにはその二つの値 x_s, d_s を格納した．G の領域をスタックと使うため， x_s, d_s を $2(n-1)^2$ 以下の値で表現する方法について述べる．つまり， G の一つの要素に x_s, d_s の二つの値を埋め込む．d_s の取り得る値，すなわち，垂直方向の距離は 0 から $n-1$ の n 種類であり，また，x_s の取り得る値も 0 から $n-1$ の n 種類である．よって，次の計算式で G の要素 $G(x_g, y_g)$ に x_s, d_s を埋め込むことができる．

$$G(x_g, y_g) = x_s + n \times d_s.$$

また，次の計算式で $G(x_g, y_g)$ から x_s, d_s 取り出すことができる．

$$\begin{aligned} x_s &= G(x_g, y_g) \bmod n, \\ d_s &= \lfloor G(x_g, y_g)/n \rfloor. \end{aligned}$$

このとき，要素 $G(x_g, y_g)$ が最大になるのは $x_s = n-1, d_s = n-1$ のときであり，$(n-1) + n(n-1) = (n+1)(n-1)$ となる．$n \geq 3$ のとき $(n+1)(n-1) \leq 2(n-1)^2$ なので，入力として与えられた 2 値画像の一辺の要素数が 3 以上のとき，出力結果の要素の最大値である $2(n-1)^2$ を超える事はない．一般的な画像の一辺の要素数は 3 以上であるので，$n \geq 3$ と仮定してよい．このような方法により，スタックの一つの要素を G 上の一つの要素に対応付けることができる．スタックとして使うには，G の任意の行の左端からスタックの要素として用い，その行のどの要素までスタックが積まれているかを示す変数 t を用意すれば，G 上でスタックを構成することができる．(図 3.11)

以上の方法によって，G 上の要素を用いてスタックを実現する事が出来る．しかし，当然ながら G 上の要素をスタックとして用いたとき，その要素に前に入っていた値は上書きされてしまう．同様に，スタックとして使用している要素に別の値，例えばその要素の求めるユークリッド距離を上書きした場合，スタックのデータが壊れてしまう．単純に Hirata のアルゴリズムのスタックを G 上で実現した場合，上記の問題が発生し，正しく動かない．

Algorithm 7 STEP 2 の処理である下側包絡線の系列を求める操作以降，下側包絡線に含まれない放物線は使われない．したがって，この操作後には，下側包絡線に含まれる放物線が分かればよく，使われない放物線は消えてしまっても構わない．下側包絡線に含まれる放物線は G 上のスタックとして保存される．いま，y 行の左から下側包絡線を形成し，任意の要素 $G(x_g, y)$ まで処理が終わり，スタックとして $G(x_s, y)$ の要素まで使われているとする．この時，$x_s \leq x_g$ であり，まだ処理が終えていない放物線が消えることはない．

問題は Algorithm 7 STEP 3 の下側包絡線からユークリッド距離を求める操作である．スタックとして使われていない右端からユークリッド距離を求め，行列 G 上の対応する要素に上書きしていく．この時，求める要素がスタックに追いついてしまい，スタックのデータが上書きされて壊れることがある．スタックが壊れるパターンは二つあり，図 3.12 のような，要素 $G(0, y)$ よりも左に交点が出来るような場合と，図 3.13 のような，1 区間に交点 $c(P_{i-1}, P_i)$ と $c(P_i, P_{i+1})$ が両方とも現れるような放物線 P_i が存在する場合である． どちらの場合も，各要素の垂直方向に下側包絡線として現れない放物線，つまり使われる事のない余分な放物線がスタック上に含まれるためである．スタック上の全ての放物線が使用されるのであれば，求める要素がスタックに追いついてしまうことはない．なぜなら，余分な放物線がないにも関わらず求める要素がスタックに追いついたと仮定すると，残りの要素数 < スタックのサイズ となるが，一つの要素が下側包絡線として使う放物線は一つだけあるため，全ての要素が異なる放物線を使ったとしても，スタックのサイズ − 残りの要素数 > 0 となり，使われない放物線が残り，矛盾するからである．ちなみに，要素 $G(n-1, y)$ よりも右に交点が出来るような場合も，各要素の垂直方向に下側包絡線として現れない放物線が存在する事になるが，Hirata のアルゴリズムで右の要素からユークリッド距離を求めたとき，余分な放物線は真っ先に削除されるので問題にはならない．

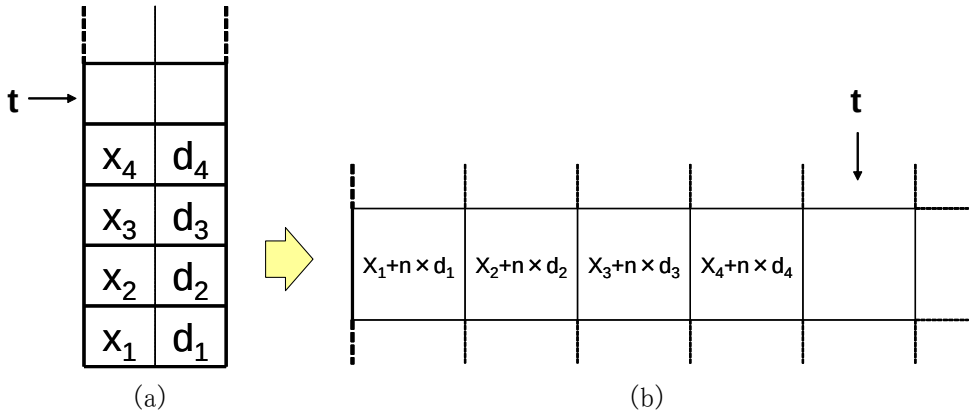

図 3.11: （a）Hirata のアルゴリズムのスタックと，(b） G 上のスタックの対応．t は使用されている要素の次の要素を指す．

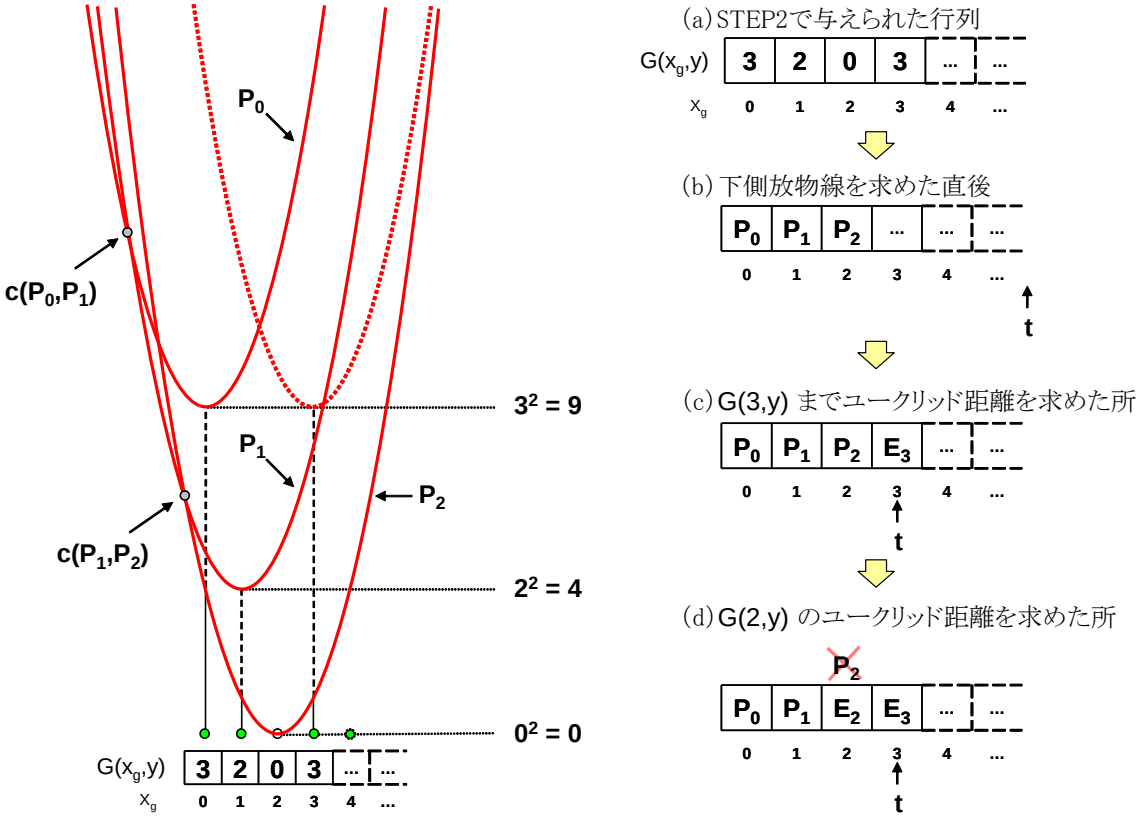

図 3.12: スタックのデータが壊れる例 1. (d) の時点でスタック上の放物線 P_2 が $G(2, y)$ の求めたユークリッド距離 E_2 によって上書きされ，スタックのデータが壊れてしまう．実際には使われない余分な放物線 P_0, P_1 の分，スタックが大きくなるためである．

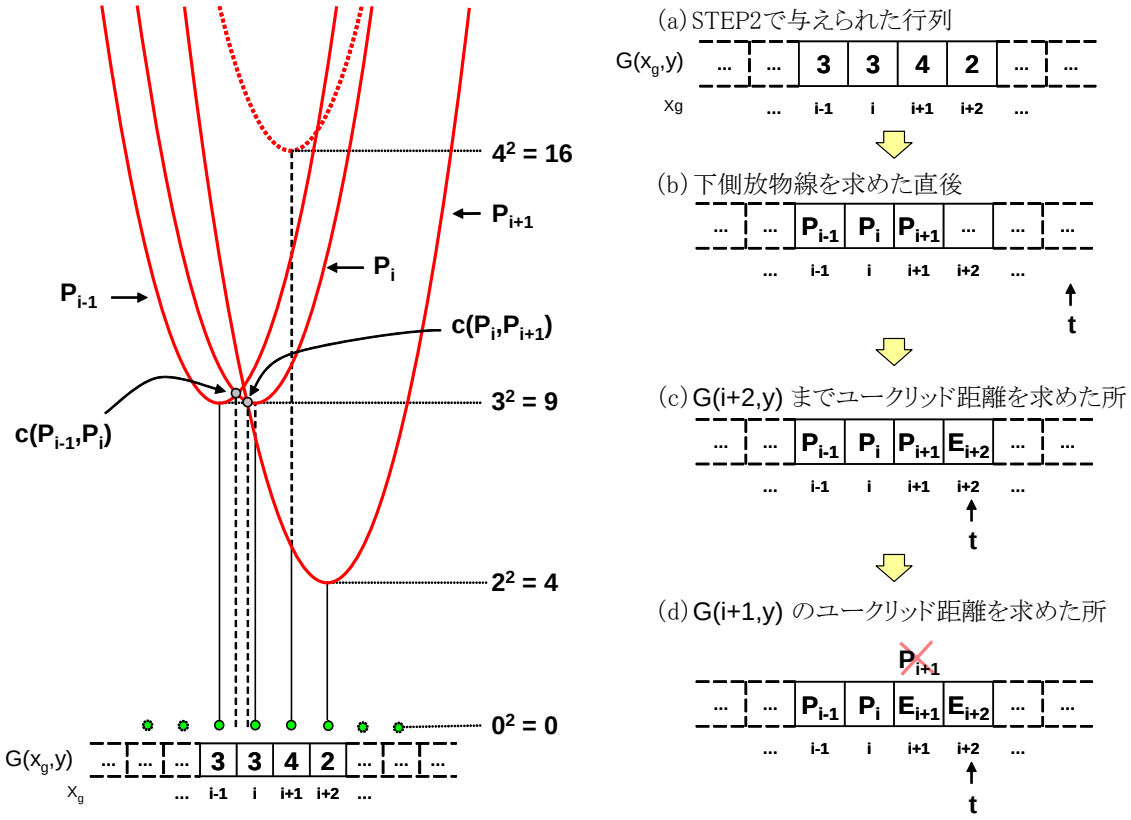

図 3.13: スタックのデータが壊れる例 2．$G(0,y)$ から $G(i,y)$ までの放物線が，全て下側包絡線に含まれ，スタック上にあるとする．(d) の時点でスタック上の放物線 P_{i+1} が $G(i+1,y)$ の求めたユークリッド距離 E_{i+1} によって上書きされ，スタックのデータが壊れてしまう．実際には使われない余分な放物線 P_i の分，スタックが大きくなるためである．

以上の事から，Algorithm 7 STEP 3 の下側包絡線からユークリッド距離を求める操作の前に，スタック上の下側包絡線の系列から，上述の余分な放物線を取り除く操作を行えばよいことがわかる．余計な放物線の判断は簡単に行える．図 3.12 のような，要素 $G(0, y)$ よりも左に交点が出来るような場合は，単に交点の x 座標が 0 以下であるか調べることで判断でき，図 3.13 のような，1 区間に交点 $c(P_{i-1}, P_i)$ と $c(P_i, P_{i+1})$ が両方とも現れるような放物線 P_i が存在する場合は，条件式 $\lfloor c(P_{i-1}, P_i)$ の x 座標 $\rfloor = \lfloor c(P_i, P_{i+1})$ の x 座標 $\rfloor$ が成り立てば放物線 P_i は余分であると判断できる．

これで，全ての問題をクリア出来た．提案するアルゴリズムを Algorithm 8 及び Algorithm 9 に示す．このアルゴリズムでは，スタックが不要になり，定数作業領域で実現できる．また，新しく追加された操作である，余分な放物線を取り除く操作は，左から順に不要な放物線でないか確認するだけなので線形時間でできる．

Algorithm 8: 提案アルゴリズム（ユークリッド距離変換問題）STEP 1

Input: $n \times n$ の 2 値行列 G

```
//— STEP 1 —//
for x_c = 0 to n − 1 do
    d = ∞
    for y_c = 0 to n − 1 do
        if G(x_c, y_c) > 0 then
            d = d + 1
        else
            d = 0
        G(x_c, y_c) = d
    d = ∞
    for y_c = n − 1 to 0 do
        if G(x_c, y_c) > 0 then
            d = d + 1
        else
            d = 0
        G(x_c, y_c) = min(G(x_c, y_c), d)
```

Algorithm 9: 提案アルゴリズム（ユークリッド距離変換問題）STEP 2 - 3

Input: $n \times n$ の 2 値行列 G

for $y_c = 0$ *to* $n-1$ **do**
  //— STEP 2 —//
  $G(0, y_c) = 0 + n \times G(0, y_c)$
  $G(1, y_c) = 1 + n \times G(1, y_c)$
  $t = 2$ // スタックに二つの要素が入っている
  for $x_c = 2$ *to* $n-1$ **do**
    repeat
      $(X_{t-2}, Y_{t-2}) =$ スタックの上二つの要素の放物線の交点
      if $Y_{t-2} \geq (X_{t-2} - x_c)^2 + G(x_c, y_c)^2$ **then**
        $t = t - 1$ // スタックの一番上の要素を削除
    until $Y_{t-2} < (X_{t-2} - x_c)^2 + G(x_c, y_c)^2$
    $G(t, y_c) = x_c + n \times G(x_c, y_c), t = t + 1$ // スタックに $(x_c, D(x_c, y_c))$ を PUSH
  //— STEP 2.5 —//
  下側包絡線から以下の余分な放物線を削除し左に詰める.
  ・$c(P_i, P_{i+1})$ の x 座標 ≤ 0 である交点の左側の放物線 P_i
  ・$\lfloor c(P_{i-1}, P_i)$ の x 座標 $\rfloor = \lfloor c(P_i, P_{i+1})$ の x 座標 $\rfloor$ である放物線 P_i
  //— STEP 3 —//
  for $x = n-1$ *to* 0 **do**
    repeat
      $(X_{t-2}, Y_{t-2}) =$ スタックの上二つの要素の放物線の交点
      if $X_{t-2} \geq x_c$ **then**
        スタックを POP しての一番上の要素を削除
    until $X_{t-2} < x_c$ *or* $t = 1$
    $x_t = G(t-1, y_c) \bmod n$
    $g_t = \lfloor G(t-1, y_c)/n \rfloor$
    $G(x_c, y_c) = (x_c - x_t)^2 + g_t^2$

3.4 アルゴリズムの拡張 - 双方向ユークリッド距離変換問題

これまでの問題では，1-pixel から最も近い 0-pixel までの距離のみを求めていた．今度は 2 値画像の 1-pixel と 0-pixel の両方にユークリッド距離を求める事を考える．つまり，各 1-pixel から最も近い 0-pixel までの距離と，各 0-pixel から最も近い 1-pixel までの距離，その両方を求める事を考える．ただし，元のデータが 0-pixel か 1-pixel を判断するため，1bit の目印を使う事を認めるとする．要するに，1-pixel から 0-pixel への距離は正の値を用い，0-pixel から 1-pixel への距離は負の値を用いる．(図 3.14)

図 3.14: 双方向ユークリッド距離変換の例. (a) 最も近い異なる要素. (b) の各距離を 2 乗し，0-pixel にはマイナスを付けた値を出力とする (c) .

線形時間かつ $O(n)$ 作業領域での実現は簡単である．例えば Hirata のアルゴリズムを少し変更し，STEP1 の各列，STEP2 の各行の処理に $O(n)$ サイズの配列を利用する．1-pixel の処理時には 0 以下の要素を 0-pixel とみなし，0-pixel の処理時には 1 以上の要素を 1-pixel とみなし，通常通り処理をする．0-pixel に対する結果と 1-pixel の結果を一時的に $O(n)$ サイズの別配列に保存し，各行または各列の処理が終わるたびに 2 つの結果を合併し，出力行列に入れる．というような単純な変更で実現できる．

では，線形時間かつ定数作業領域で実現する事は出来るだろうか．3.3 で提案したアルゴリズムを利用する事を考える．提案したアルゴリズムでは，スタックを G 上に再現することで定数作業領域での実現が可能になった．同様の方法を用いて，一つの要素にさらに情報を追加出来ないか考える．目印として正負が使えるようになったので，各要素に使ってよい値は，$-2(n-1)^2$ から $2(n-1)^2$ である．このとき，スタックとして使用される値は符号情報を加えた $-(n+1)(n-1)$ から $(n+1)(n-1)$ の範囲である．仮に 1bit の情報を追加するとなると，範囲は 2 倍必要になる．$2(n-1)^2 < 2(n+1)(n-1)$ なので，同じ方法で，さらに情報を追加することは出来ない．よって，スタックとして G を使用すると，この問題でさらに必要になった情報，各要素が 0-pixel であったか 1-pixel であったかを保持することが出来ず，実現できない．

しかし，別の方法で線形時間かつ定数作業領域で計算可能なアルゴリズムを実現できる．提案アルゴリズムの STEP2 について，現在は各行全体を一度に下側包絡線に変換し，ユークリッド距離を求める方法をとっている．ここで，各行に注目し，1-pixel が連続した各区間について考える．図 3.15 のように $G(l,y)$ から $G(r,y)$ の連続した 1-pixel の区間があるとする．このとき，$G(l-1,y)$ 及び $G(r+1,y)$ は（行列の範囲外でなければ）必ず 0-pixel である．つまり， $G(l,y)$ から $G(r,y)$ までの求める最も近い 0-pixel は，最悪でも，それら 0-pixel のうち近い方となる．それよりも外の範囲の要素がなんであろうと，$G(l,y)$ から $G(r,y)$ に影響することはない．0-pixel に対する 1-pixel についても同様であり，各行の同じ要素が連続した区間はそれぞれ独立していることが分かる．このことから，行全体を一度に変換する必要はなく，0-pixel が連続した区間，1-pixel が連続した区間に分けて，それぞれで処理をしても問題はないといえる．

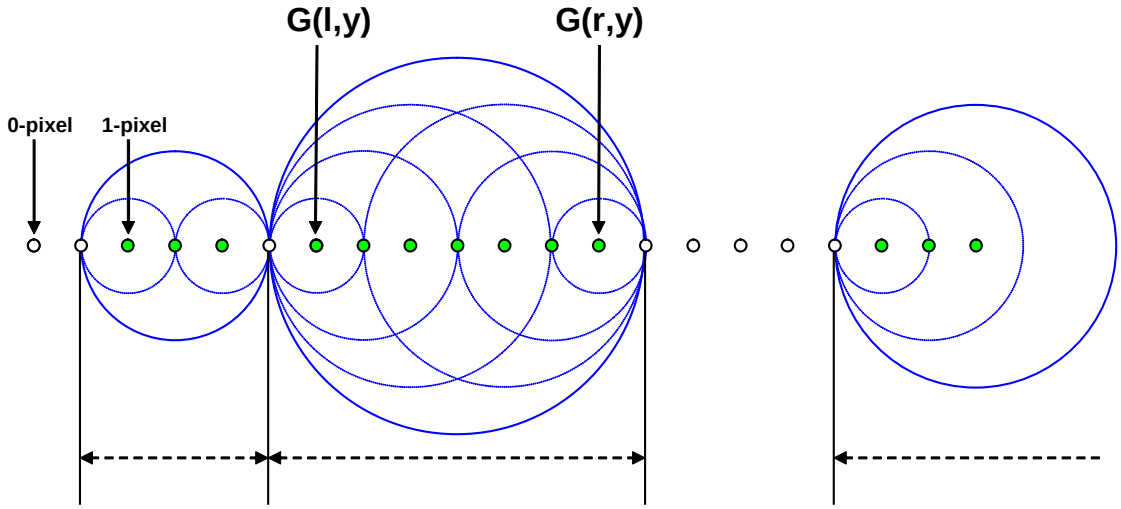

図 3.15: 行をみた時，各 1-pixel が取りえる距離の範囲．1-pixel の区間毎に独立しているのが分かる．0-pixel についても同様である．

$G(l,y)$ から $G(r,y)$ の 1-pixel の区間に対してユークリッド距離を求めるためには，$G(l-1,y)$ から $G(r+1,y)$ の放物線を用いて下側包絡線を求める必要がある．$G(l,y)$ から $G(r,y)$ の範囲のみをスタックとして用いて下側包絡線を作るには要素数が足りない．そこでその両サイドの要素，$G(l-1,y)$ と $G(r+1,y)$ もスタックとして用いることにする．当然，$G(l-1,y)$ ，$G(r+1,y)$ の要素は破壊されるため，スタックとして用いる前に 2 つの変数にそれぞれの値を移しておき，$G(l-1,y)$ から $G(r+1,y)$ の区間の処理が終われば値を戻す．以上の方法により，高々2 個の変数を追加するだけで線形時間，定数作業領域でこの問題を解くことが出来る．

双方向ユークリッド距離変換問題に対する提案するアルゴリズムを Algorithm 10 及び Algorithm 11 に示す．

Algorithm 10: 提案アルゴリズム（双方向ユークリッド距離変換問題） STEP 1

Input: $n \times n$ の 2 値行列 G

```
//— STEP 1 —//
for x_c = 0 to n − 1 do
    d_0 = −∞
    d_1 = ∞
    for y_c = 0 to n − 1 do
        if G(x_c, y_c) > 0 then
            d_0 = 0
            d_1 = d_1 + 1
            G(x_c, y_c) = d_1
        else
            d_0 = d_0 − 1
            d_1 = 0
            G(x_c, y_c) = d_0
    d_0 = −∞
    d_1 = ∞
    for y_c = n − 1 to 0 do
        if G(x_c, y_c) > 0 then
            d_0 = 0
            d_1 = d_1 + 1
            G(x_c, y_c) = min(G(x_c, y_c), d_1)
        else
            d_0 = d_0 − 1
            d_1 = 0
            G(x_c, y_c) = max(G(x_c, y_c), d_0)
```

Algorithm 11: 提案アルゴリズム（双方向ユークリッド距離変換問題） STEP 2 - 3

```
for y_c = 0 to n − 1 do
    //— STEP 2-3 —//
    // 各行の処理
    // 連続区間を探し，区間毎にユークリッド距離を求める
    // 区間の両サイドの要素，G(l − 1, y_c) と G(r + 1, y_c) が行列外でないかの判定
    // 及びその場合の例外処理は省略する
    // また，下側包絡線，及びユークリッド距離を求めるアルゴリズムは
    // Algorithm9 と同様なので省略する
    l = 0
    for x_i = 0 to n − 1 do
        if x_i = n − 1 or G(x_i, y_c) の符号! = G(x_i + 1, y_c) の符号 then
            r = x_i
            temp_l = G(l − 1, y_c)
            temp_r = G(r + 1, y_c)
            G(l − 1, y_c) = G(r + 1, y_c) = 0
            s = G(x_i, y_c) の符号
            G(l − 1, y_c) から G(r + 1, y_c) の範囲のユークリッド距離を求める（省略）
            G(l, y_c) から G(l, y_c) の符号を s に変換
            G(l − 1, y_c) = temp_l
            G(r + 1, y_c) = temp_r
            l = r + 1
```

第4章　おわりに

本論文では，連結成分ラベル付け問題，及びユークリッド距離変換問題に対して，線形時間定数作業領域で実現するアルゴリズムを提案した．

連結成分ラベル付け問題では，境界を辿る方法を用いることで定数作業領域を実現した．大きな成分に対して効率的なラベル付けを行えるため，計算時間に置いても優れている．計算機実験において，各得意なパターンで既存のアルゴリズムよりも良い結果を出せた．また，各アルゴリズムの最悪時の実行時間を比較した場合でも，既存のアルゴリズムと同等かそれよりも良い結果を出せた．拡張問題として5つ提示し，提案アルゴリズムを応用する方法について述べた．特に形状特徴パラメータの同時抽出は，同時に求めることが可能なパラメータの種類が他のアルゴリズムよりも圧倒的多く，さらにラベル付けをせずに各パラメータを抽出することができる利点がある．今後の課題として，並列化が挙げられる．Rosenfeld-Pfaltz アルゴリズムでは，複数のブロックに分け，同時に処理する事が容易にできる．しかし，提案アルゴリズムでは，効率的な並列化は簡単ではない．

ユークリッド距離変換問題では，既存のアルゴリズムで必要となる一時的な情報を出力行列中に埋め込むことで定数作業領域を実現した．拡張問題として，双方向ユークリッド距離変換問題を提示し，定数作業領域で実現する方法について述べた．

謝辞

浅野哲夫教授，上原隆平准教授，元木光雄准教授，清見礼助教をはじめ，情報棟 3 棟 6 階，浅野・上原研究室の皆様，及び，よく出入りされていた皆様方には公私に渡り様々な面でお世話になりました．この場を借りてお礼申し上げます．特に指導教官の浅野哲夫教授には日頃から懇切丁寧なご指導を頂きましたことを深く感謝致します．最後に，同じ時間を共にした全ての方に心からの敬意と感謝の意を表します．

参考文献

[1] A.Rosenfeld and J.L.Pfaltsz, *"Sequential operations in digital picture processing"* J.ACM, pp.471-494, 1966.

[2] H.Breu, J.Gil, D.Kirkpatrick, and M.Werman, *"Linear Time Euclidean Distance Algorithms"* IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol.17, No.5, pp.529-533, 1995.

[3] T.Hirata, *"A unified linear-time algorithm for computing distance maps"* Information Processing Letters, Vol.58, No.3, pp.129-133, 1996.