졸업논문 청구논문

# 근사적인 Sequence Matching을 이용한 다중 조각 영상 재합성 알고리즘 개발

## Multiple Image Stitching Algorithm using Approximate Sequence Matching


경 기 과 학 고 등 학 교

수 학 및 정 보 분 야

김 준 기


2004 년 11월 27일

졸업논문 청구논문

# 근사적인 Sequence Matching을 이용한 다중 조각 영상 재합성 알고리즘 개발

## Multiple Image Stitching Algorithm using Approximate Sequence Matching

이 논문을 김준기의 졸업 논문으로 인준함

지도교사 김 순 근 (인)

과학부장 박 정 행 (인)

2004 년 11월 27일

경 기 과 학 고 등 학 교

# < 차 례 >

그림 목차 ········ 2
국문 초록 ········ 3

**I.** 연구 동기 및 목적 ········ 4
**II.** 선행 연구 및 이론적 배경 ········ 4
**1.** 현재의 토기·도기 복원 과정 ········ 4
**2.** 용어의 뜻과 정의 ········ 5
**3.** 기존의 Image Stitching 알고리즘 ········ 6
**4.** 기존 알고리즘의 문제점 ········ 6
**5.** Approximate Sequence Matching ········ 6
**6.** 현재까지의 Sequence Matching 알고리즘의 활용 ········ 7
**III.** 연구 내용 및 결과 ········ 8
**1.** 제한 및 가정 ········ 8
**2.** 알고리즘의 목표 ········ 8
**3.** Sequence Matching 도입 ········ 9
**4.** System Parameters ········ 9
**5.** 알고리즘의 설계 및 구현 ········ 10
**5.1.** 전처리 과정 - 조각 Image의 Sequence 표현 ········ 11
**5.2.** Approximate Sequence Matching ········ 15
**5.3.** 결과 출력 ········ 18
**IV.** 결론 및 제언 ········ 19
**1.** 연구 의의 ········ 19
**2.** 앞으로의 연구 방향 ········ 19
**V.** 참고 문헌 ········ 20
**VI.** 부록 ········ 20
**1. Bresenham's Line Algorithm** ········ 21

## < 그림 목차 >

**Fig.1** 일부 복원된 한 선사시대 토기의 모습 ······ 4
**Fig.2** 조각 Image 경계선의 Sampling 예 ······ 9
**Fig.3** 실제 Image에서의 Sampling Box Size ······ 10
**Fig.4** 경계선 추적 과정 ······ 13
**Fig.5** 실제 Sampling 작업 중 정보를 읽게 되는 한 Box의 픽셀 영역 예 ······ 14
**Fig.6** 두 sequence를 비교할 때 서로 뒤집혀 있어야 하는 이유 ······ 16
**Fig.7** A와 B의 shifting 과정 모식도 ······ 16

# < 초 록 >

본 연구는 원래 한 물체였다가 부서진 조각들을 Digital Image로 스캔하고 이를 서로 짜 맞추어 원래 형태를 알아내는 Multiple Image Stitching 알고리즘을 제안한다. 이 연구는 유적에서 발굴된 토기 및 도기의 복원 작업이 현재도 전 과정이 사람에 의해 이루어지고 있어 그 중에서 부서진 조각들을 원래대로 합쳐 주는 과정을 자동화하면 더 빠르고 정확하게 복원을 할 수 있을 것이라는 생각에서 출발하였다. 이러한 Digital Image 처리 과정을 Image Stitching이라 하는데, 본인은 Image Stitching 알고리즘에 Approximate Sequence Matching 기법을 도입함으로써 기존의 알고리즘을 그대로 적용하였을 때 발생할 수 있는 문제점들을 알아보고 그 해결 방안을 탐색하였다. 이와 함께 앞으로의 연구 방향을 제시하고자 한다.

Keywords : Approximate Sequence Matching, String Matching, Image Processing, Image Stitching, Multiple Image Stitching, Stitching Algorithm for 2D fragment images

## I. 연구 동기 및 목적

현대에는 정보과학의 발달로 일상 생활에서부터 전문적인 연구 분야에 이르기까지 많은 부분이 자동화되었고, 더욱 정확하면서 빠른 시간에 원하는 결과를 출력할 수 있게 되었다. 하지만, 아직까지도 깨진 도·토기들의 복원 작업에서는 이러한 자동화의 혜택을 거의 받고 있지 못하다는 사실을 발견하였고, 그 중에서 정보과학의 응용 가능성을 탐색해 보니, 부서진 조각들을 원래대로 합치는 과정이 가장 어렵고 오래 걸리는 일이라는 것을 알게 되었다. 따라서 그 문제를 해결하기 위한 Image Stitching 알고리즘을 적용하고, 조금 더 효율적인 설계를 위해 Sequence Matching 기법을 도입하였다.

## II. 사전 연구 및 이론적 배경

### 1. 현재의 토기·도기 복원 과정

국립중앙박물관 고고학부를 찾아가 본 결과 현재도 깨진 도·토기들의 복원 작업은 모두 직접 사람이 하고 있음을 확인할 수 있었다. 실제로는 깨진 조각이 모두 발견되는 경우보다는 그렇지 않은 경우가 더 많으며, 문양이나 시대적 배경을 알 수 있는 경우는 보다 쉽게 맞출 수 있다고 하였다.

Fig.1 일부 복원된 한 선사시대 토기의 모습

박물관의 설명을 들어보니 이런 복원 작업을 실제 자동화하려는 시도가 있었으나 끝을 보지 못하고 실패했다고 하며, 그것은 3차원으로 스캔하여 이미지를 맞추

려는 작업이었다고 한다. 당시 고려해야 할 변수가 지나치게 많았던 것이 실패의 원인이라고 하였다. 그래서 이 연구에서는 약간 다른 방식으로 접근하고자 한다.

## 2. 용어의 뜻과 정의

이 논문에서 사용하는 용어의 뜻과 정의는 아래에서 설명하는 바와 같다.

(1) Image(이미지, 영상) : 디지털로 입력되어 있는 사진이나 그림 정보를 말하며, 영상 처리 알고리즘에서 다룰 때는 임의의 자연수 $m$, $n$에 대하여 $0<x<n$, $0<y<m$인 $(x, y)$원소에 색상 정보를 가지는 $m$ *by* $n$ 행렬로 표현된다. [1]

(2) Image Stitching : 둘 이상의 Image가 있을 때, 서로 공통되는 가장자리 영역을 찾아내어 하나로 합치는 과정을 말한다.

(3) Sequence와 String : Sequence는 각 원소에 순서가 매겨져 있는 유한 집합을 말하는 것으로[2], 보통 List와 같은 뜻으로 사용된다. String은 보통 알파벳, 숫자, 기호 등으로 구성된, 순서가 정해진 문자열을 뜻하지만 본 논문에서는 연속적인 순서를 가지고 임의의 한 집합에 포함되는 원소들이 연속적으로 붙어있는 Sequence로 간주한다. Subsequence와 Substring은 String의 부분집합이다. 마찬가지로, Subsequence는 반드시 연속으로 원소를 포함하지 않아도 되지만 Substring은 반드시 연속적으로 인접한 원소들이어야 한다.

(4) Sequence Matching : 임의의 Sequence 또는 String에서 Subsequence를 찾아내는 것으로 본 논문에서는 불완전한 Approximate Matching도 포함한다.

(5) Color Difference : Color Difference는 두 가지의 색이 얼마나 차이가 나는지 나타내는 지표이다. 보통은 Lab 색상 모델을 이용한 Delta E로 계산되지만, 여기서는 RGB Color Difference($D$)로 정의하여 사용한다.

$$D = \sqrt{\Delta R^2 + \Delta G^2 + \Delta B^2}$$

Eq.1 RGB Color Difference

$$\Delta E = \sqrt{\left(\frac{\Delta L}{K_L S_L}\right)^2 + \left(\frac{\Delta C}{K_C S_C}\right)^2 + \left(\frac{\Delta H}{K_H S_H}\right)^2}$$

where

$$\Delta C = C_1 - C_2,\ C_1 = \sqrt{a_1^{\ 2} + b_1^{\ 2}},\ C_2 = \sqrt{a_2^{\ 2} + b_2^{\ 2}}$$

$$\Delta H = \sqrt{\Delta a^2 + \Delta b^2 - \Delta C^2}$$

$$K_L = K_C = K_H = 1,\ K_1 = 0.045,\ K_2 = 0.015$$

$$S_L = 1,\ S_C = 1 + K_1 C_1,\ S_H = 1 + K_2 C_2$$

Eq.2 Delta E Color Difference (CIE 1994)

### 3. 기존의 Image Stitching 알고리즘

현재 Image Stitching은 의료 영상 분야, 전자 현미경의 영상 처리, 디지털 카메라의 파노라마 기능 등에 사용되고 있다. 이러한 Image Stitching 알고리즘은 대다수가 두 개의 Image를 비교하여 하나로 합치는 문제에 관한 것이다. 이들은 두 Image의 좌표끼리 mapping을 하거나, Image에 나타나는 외곽선의 (연속적) 형태를 비교하는 등의 방법을 이용하고 있다.

### 4. 기존 알고리즘의 문제점

기존의 Image Stitching 알고리즘들은 간단히 두 개의 Image를 합치는 것이지만, 유물 복원 작업에서는 다각형 또는 부분적 곡선 형태를 가진 여러 개의 Image들을 합치는 과정으로, 단순한 좌표의 mapping을 통한 비교로는 해결하기 어려운 문제들이 있다. 첫째로, 유물 조각들의 경우 원래 물체가 금만 가서 그대로 쪼개져 있는 경우는 거의 없고 여러 곳에서 흩어져 발견되는 경우가 대부분이다. 이때 각 조각들을 스캔한 Image는 특정 방향에 대하여 임의의 각도로 회전되어 있을 것이고, 단순히 특정 좌표끼리 색상이나 밝기를 비교하는 방법은 모든 가능한 경우에 대해 회전 변환을 적용하여야 비교해야 하므로 연산 과정에서 효율이 떨어지게 된다. 둘째로, 조각 Image들의 외곽선이 일정하지 않다는 데 있다. 어떤 것은 딱딱한 직선형의 다각형 형태일 수도 있고, 어떤 것은 부드러운 곡선 형태를 포함할 수도 있다. 이들의 형태를 분석하여 비교하는 것은 이들 외곽선들을 충분히 작은

각도 단위로 회전시켜가면서 역시 모든 경우에 대해 비교를 해 봐야 하므로 비효율적이다.

### 5. Approximate Sequence Matching

Approximate Sequence Matching은 subsequence가 정확히 맞아떨어지지 않더라도 어느 정도의 차이는 인정하여 matching을 수행하는 방법이다. 가장 기본적으로는 Edit Distance, 즉 한 문자열에서 다른 문자열로 바꿀 때 얼마만큼의 문자를 추가하고, 삭제하고, 삽입하는 과정을 거쳐야 하는지 나타내는 지표를 사용하여 그 유사도를 표현한다.[3]

### 6. 현재까지의 Sequence Matching 알고리즘 활용

생명공학의 발달로 인간의 유전자 지도가 밝혀지고, 각종 생물들의 유전자 지도를 만들어 이를 질병 치료에 이용하고자 하는 수요가 크게 증가하고 있다. 이 과정에서 필수적인 것이 바로 DNA Sequence의 분석이다. DNA Sequence는 T, C, G, A의 네 문자로 이루어지는 긴 String이라 할 수 있다. 이 방대한 String 데이터에서 어떤 유전자가 있는지, 특정 아미노산이나 단백질을 암호화하는 부분이 있는지 알아보는 것이 현대 생명공학의 주 관심사이다. 자연히, 정보 과학에서 오래 전부터 일반 사용자들에게는 워드프로세서의 “문자열 찾기”로 알려진 String Matching 알고리즘을 응용하게 되었고, 여러 개의 DNA Sequence가 있을 때 더 효율적으로 하기 위한 Multiple Sequence Matching, 또 부분적으로 불일치하더라도 match로 인정하는 Approximate Sequence Matching 등의 방법이 개발되었다. 최근의 연구 방향은 생물정보학(Bioinformatics) 분야에서의 빠르고 정확한 유전자 분석을 목표로 하고 있다. (아직도 Multiple Sequence Matching은 The Holy Grail, 즉 최후의 성배라고 불릴 만큼 그 가능성이 많이 남아 있는 분야다)[3]

## III. 연구 결과

### 1. 제한 및 가정

실제 발굴되는 도·토기들은 대부분 불완전한 수의 조각만 발견되며, 겉의 문양이나 색이 심하게 변질되어 있는 경우가 많다. 본 연구에서는 Sequence Matching을 도입하여 Image Stitching 알고리즘을 구현하는 것이 주된 목적이므로, 위와 같은 현실적 사항들에 대해서는 다음과 같이 가정하여 무시하겠다.

(1) 맞추려는 조각들은 반드시 하나의 완성된 모양을 가진다.

(2) 조각들은 부분에 따른 변색 차이 등 보정해야 할 요소가 없다. (이를 다르게 표현하면 이미 동일하게 보정이 이루어진 이미지들을 사용한다고도 할 수 있다)

(3) 모든 조각들의 Image는 같은 조건(카메라의 위치 및 배율, 단색 배경, 조명)에서 촬영된 것이다.

(4) 3차원적 조각이 커짐으로써 발생하는 2차원 Image에서의 왜곡은 무시한다. 즉, 외곽선을 따라 sampling한 sequence들이 가지는 임의의 subsequence에 대해 서로 scale이 동일하다.

(5) 촬영 조건 중에서 배경은 조각이 가지고 있지 않은 전혀 다른 색을 띠고 있으며 촬영된 Image에서 가장자리 테두리를 모두 둘러싸고 있다.

(6) 조각은 서로 뒤집어진 것이 없다. (전체가 그릇 모양이라고 할 때 한 조각은 안쪽을 촬영하고 다른 한 조각은 바깥쪽을 촬영하거나 하는 일 없이 모두 안쪽이거나 모두 바깥쪽이다)

### 2. 알고리즘의 목표

II.1에서의 제한 사항을 바탕으로, 본 알고리즘이 수행해야 하는 기능은 다음과 같으며, 알고리즘의 수행 단계도 이와 같다.

(1) 각 조각 Image들의 외곽선을 구하고, 이를 바탕으로 색상 및 형태에 관한 정보를 sampling해야 한다.

(2) Sampling 과정에서 발생할 수 있는 차이를 흡수할 수 있을 만큼의 Approximate Sequence Matching이 이루어져야 한다.

(3) Sequence Matching 결과를 다시 원래의 Image에 표현할 수 있어야 한다.

여기서 형태에 관한 정보가 추가된 것은 선사시대 토기들의 경우 무늬나 색상 정보에 의한 것보다는 형태가 더 중요하게 작용하기 때문이며, 문양이 있는 후시대의 것들이라 할지라도 무늬가 없는 빈 부분이 더 많기 때문이다.

### 3. Sequence Matching의 도입

II장에서 말한 기존 알고리즘의 문제점을 해결하기 위해 Sequence Matching 도입을 제안한다. 이를 도입하여 생기는 장점은 회전과 기하학적 변환을 전혀 신경 쓰지 않아도 된다는 것이다. 외곽선을 따라 외곽선 바로 안쪽의 내용을 sampling 하여 만드는 sequence이기 때문에, 외곽선의 형태에 구애받지 않게 되며 다만 각 sequence의 길이는 각 조각의 외곽선 길이에 비례한다.

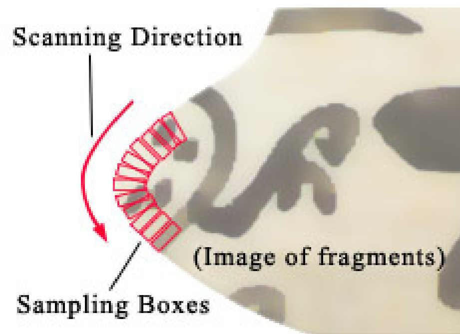


Fig.2 조각 Image 경계선의 Sampling 예

여기서 사용할 Approximate Sequence Matching에서는 2장에서 설명했던 Edit Distance의 개념을 사용하지 않고, 대신 각 matching 가능한 요소 사이에 최대 몇 개까지 matching되지 않는 요소가 포함될 수 있는지 나타내는 Maximum Gap의 개념을 사용하였다.

### 4. System Parameters

이 알고리즘에는 다음과 같은 기본 상수들이 필요하다. 이들은 어떤 조건의 Image를 가져오느냐에 따라 (결과를 확인하여) 알고리즘을 수행하기 전에 재조정될 수 있다. 이 논문에서는 Test Image에 가장 적절한 값들을 선택하였다.

(1) Color Difference Offset ($d$) : 두 요소 사이의 색 차이가 얼마나 되어야 서로

다른 색으로 인정할 것인지를 나타내는 실수값이다. 본 논문에서는 Color Difference로만 정의했지만, 다른 추가적인 형태 정보가 포함될 수도 있다.

(2) Sampling Box Size ($w$, $h$) : 외곽선을 따라 sampling을 할 때 외곽선에서 안쪽으로 얼마만한 크기의 직사각형 영역(width, height)을 이용할 것인지 나타내는 값이다. 여기서 width는 외곽선 방향의 길이이고, height는 외곽선에 수직한 방향의 길이를 나타내는 것으로 $h$가 클수록 더 안쪽 영역까지 검사하겠다는 의미가 된다. 따라서 깨진 가장자리 부분의 변형이 있을 경우는 $h$가 커야 보다 올바른 결과를 얻을 수 있다. $w \times h$의 값이 클수록 sampling 시간이 오래 걸리며, 지나치게 크면 이미지의 detail을 뭉개버리므로 정확도가 떨어지게 된다. (단위 : 픽셀)

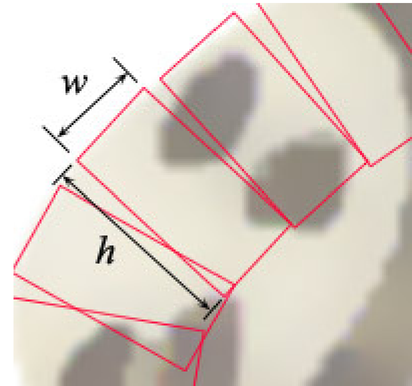


Fig.3 실제 Image에서의 Sampling Box Size

(3) Maximum Gap ($g$) : subsequence끼리 matching을 할 때 matching 되는 요소 다음에 다시 matching되는 요소가 최대 몇 개의 요소를 사이에 둘 수 있는지 결정한다. 이 값이 클수록 더 띄엄띄엄 match되더라도 subsequence로 인정될 수 있다.

(4) Shift Amount ($s$) : 두 개의 sequence를 비교할 때 긴 sequence를 고정시켜 놓고 짧은 sequence를 얼마만큼씩 뒤로 밀며 matching을 할 것인지 나타낸다. 이 값이 작을수록 보다 정밀한 결과를 얻을 수 있지만 그만큼 시간이 오래 걸린다.

(5) Minimum Matching Length ($m$) : 어떤 subsequence가 matching 조건을 만족하여 발견되었을 때 그것이 가져야 하는 최소 길이를 나타낸다. 이 값이 너무 작으면 매우 짧은 subsequence들이 무수히 많이 matching 결과에 포함될 수 있으므로 적절하게 설정해야 한다.

## 5. 알고리즘의 설계 및 구현

본 알고리즘은 다음과 같이 수행되며, 각 함수에 대한 자세한 설명은 각 처리 과정별로 구분하여 5.1에서 5.3에 걸쳐 다룰 것이다.

여기서 사용하는 모든 의사 코드(pseudo-code)에서 배열은 0-based로 가정하였다.

배열의 첫 원소의 index는 0이며, 마지막 원소의 index는 length - 1이다. “배열변수.ubound”는 length - 1을 축약한다. count는 문맥상 쉽게 읽히도록 사용한 것으로 length와 같은 값을 가진다. 또, a mod b 연산은 a를 b로 나눈 나머지로 생각한다.

```
1. begin
2.     images[] ← new Image Array[]
3.     outlines[] ← new Outline Arrray[]
4.     sequences[] ← new Sequence Array[]
5.     { input filenames and system parameters from the user }
6.     for i ← 0 to CountOfImages - 1
7.          LoadImage(filenames[i], images[i])
8.          outlines[i] ← GetOutline(images[i], GetWidth(images[i]),
 GetHeight(images[i]))
9.          sequences[i] ← OutlineToSequence(images[i], outlines[i])
10.    result ← DoMatch(sequences)
11.    for i ← 0 to CountOfImages - 1
12.         OutputResult(images[i], outlines[i], result)
13. end
```

### 5.1 전처리 과정 - 조각 Image의 Sequence 표현

일반적으로 Digital Image 스캔을 하면 직사각 행렬 형태로 정보가 입력된다. 따라서 이 중에서 원하는 부분, 즉 유물 조각에 대한 정보를 골라내기 위한 전처리 과정을 거쳐야 한다. 각각의 조각 Image에서 외곽선을 얻어낸 후 그 외곽선을 따라 작은 간격으로 외곽선 바로 안쪽의 색상 정보와 형태 정보를 sampling 한다.

조각 Image의 외곽선을 얻는 것은 다음의 의사 코드와 같다. [4] (여기서, 조각 이미지는 반드시 조각이 포함하지 않는 색상의 배경을 가지면서 이미지의 가장자리 부분은 모두 배경이 테두리를 이루고 있어야 한다)

```
1. function GetOutline(image, width, height)
2.     //// 배경 부분만 골라내어 true/false로 표시함 /////
3.     bgColor ← image[0, 0] // 배경색 기준점
```

```
4.     bgArea ← new bool Array[width, height] // 각 점이 배경인지 아닌지 저장
5.     for each curPoint in image
6.          if IsSimilarColor(image[curPoint], bgColor) = true then
7.               bgArea[curPoint] ← true
8.
9.     //// 각 점이 배경 부분인지 아닌지를 이용하여 외곽선 검출 ////
10.    near8Points[8] = { relative coordinates of 8 pixels around }
11.    visited ← new bool Array[width, height]
12.    outline ← new Point Array[]
13.    for each curPoint in image
14.         if (not bgArea[curPoint]) and (not visited[curPoint]) then
15.              origPoint ← curPoint
16.              n ← 4 // 관심점(curPoint) 주위의 4번 점에서 시작함
17.              repeat
18.                   for i ← 0 to 8
19.                        tempPoint ← curPoint + near8Points[n]
20.                        if not bgArea(tempPoint) then exit for
21.                        n ← (n + 1) mod 8
22.                   visited[curPoint] ← true
23.                   outline.add(curPoint)
24.                   curPoint ← curPoint + near8p[n] // 다음 관심점 지정
25.                   n  (n + 5) mod 8
26.              while(curPoint = origPoint)
27.              exit for
28.    return outline
29.
30. function IsSimilarColor(c1, c2)
31.    if RGBDifference(c1, c2) < 400 then return true else return false
32.
33. function RGBDifference(c1, c2)
34.    return Sqrt((c1.Red - c2.Red) ^ 2 + (c1.Green - c2.Green) ^ 2 +
 (c1.Blue - c2.Blue) ^ 2)
```

위 코드 중 특히 25행이 외곽선 추적에서 중요한데, 5를 더하는 이유는 어떠한 형태의 외곽선이더라도(1 pixel 단위로 구불구불해도) 다음번 루프에서 올바른 위치의 경계점을 찾아낼 수 있도록 하기 위함이다.

16행에서 *n*=4로 두고 시작하는 것은 for each 부분을 어떻게 구현하느냐에 따라 달라질 수 있다. for x, for y 순서로 구현을 하면 *n*=4가 되어야 하고 for y, for x 순서가 되면 *n*=6이 되어야 한다. (이것은 near8Points 배열을 어떻게 놓았느냐에 따라 맞춰진 것이다)

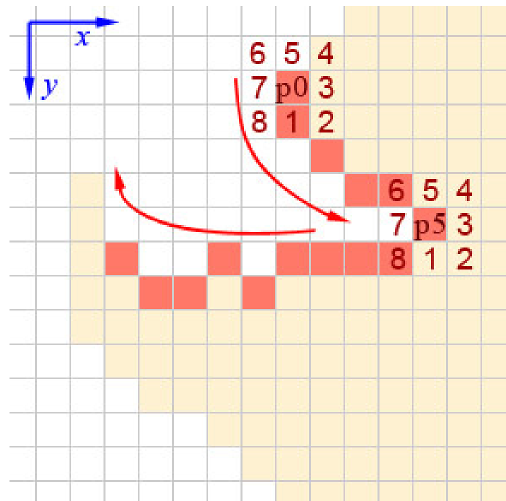


Fig.4 경계선 추적 과정. 점 주변의 8개 점을 반시계 방향으로 검사하면서 p0부터 시작해 차례로 찾아나간다.

Fig.4를 보면 p0부터 외곽선을 따라 순서대로 p5까지 찾아가는 것을 볼 수 있는데, p0에서 처음 시작했다고 보면, 처음 검색점은 4부터 시작하여 반시계방향으로 돌게 되므로 *n*=1에서 경계선으로 판정되고 p0 바로 아래의 p1(그림에는 라벨이 표시되지 않음)으로 가서 *n*=6부터 검색한다. 그러면 *n*=2의 위치가 p2의 위치가 되고 다시 *n*=7부터 검색하여 *n*=2의 위치가 p3, *n*=7부터 검색하여 *n=3*의 위차가 p4, *n*=8부터 검색하여 *n*=2의 위치가 p5가 되는 것이다. 이런 식으로 쭉 따라가면 어떤 형태의 외곽선에서도 검출이 이루어질 수 있다는 것을 알 수 있다. 물론 반대 방향으로 가는 경우에도 성립한다.

실제로는 점이 바깥으로 떨어져 나와 1개뿐인 경우도 검출이 되지만(그런 경우는 26행으로 넘어갔을 때 *i=8*이다) 이 알고리즘에서는 제한 및 가정에 따라 배경

색이 조각의 색과 확연히 다르며 조각은 하나의 외곽선을 갖는 파편이므로 이를 무시하였다.

이제 위의 GetOutline 함수를 수행하여 얻은 점(Point) 배열을 쭉 따라가면서 Sampling하여야 하는데, 여기서 색상 정보는 각각의 Sampling Box 내부의 모든 점들에 대한 RGB 평균값을 사용하며 형태 정보는 주변 외곽선의 곡률 정보를 이용한다. 여기서 Sampling Box 내의 모든 점들을 얻는 원리는 Fig.5와 같다.

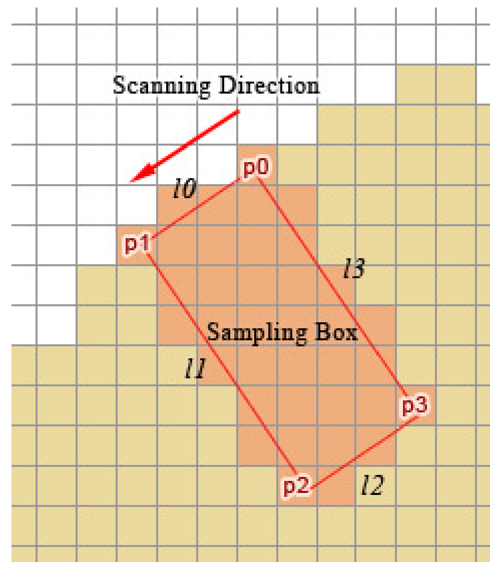


Fig.5 실제 Sampling 작업 중 정보를 읽게 되는 한 Box의 픽셀 영역 예

Fig.5에서 p0 ~ p3까지가 사각형의 각 꼭지점이고, *l0* ~ *l3*가 각 변이다. 2차원 정수 좌표 상에서 선을 그리는 가장 빠른 알고리즘 중 하나인 Bresenham's Line Algorithm[5]을 이용하여 *l0* ~ *l3* 위의 점들을 표시한 다음 좌표축과 평행하고 네 꼭지점들을 포함하는 최소 직사각형 영역을 이중 for 루프로 돌며 flag 변수를 이용해 처음 *N* 번째 변(*l*)에 해당하는 픽셀이 나오면 flag를 설정하여 쭉 읽다가 다시 *N*+2 *mod* 4 번째 변(*l*)의 점을 만나면 flag를 해제하고 그냥 넘어가는 것이다.

실제 구현할 때는 위처럼 하면 시간이 매우 오래 걸리므로, *l0* 위의 점들과 *l2* 위의 점들만 구해서 바로 직선으로 대응시켜 sampling하는 최적화 기법을 사용할 수 있다. (다만 이때는 중간에 몇몇 픽셀들이 빠지게 되나, 최종 결과에는 그다지 영향을 미치지 않을 정도이므로 무시할 수 있다)

최적화된 Sampling 작업을 의사 코드로 나타내면 다음과 같다.

```
1.  global w, h    // system parameters
2.  function OutlineToSequence(image, outline)
3.      sequence ← new Element Array[]
4.      for i ← 0  to outline.length - 1
5.          { get the 4 vertex points of the sampling box (p0 ~ p3), w x h }
6.          line0 ← GetPointsOfLine(p0, p1)
7.          line2 ← GetPointsOfLine(p3, p2)
8.          for j ← 0 to Min(line0.length - 1, line2.length - 1)
9.              templine ← GetPointsOfLine(line0[j], line2[j])
10.             for k ← 0 to templine.length - 1
11.                 { collect mean RGB values of the pixels to element }
12.         sequence.add(element)
13.     if (direction of outline sampling is reversed) then { reverse sequence }
14.     return sequence
15.
16. function GetPointsOfLine(pt1, pt2)
17.     { perform Bresenham's Line Algorithm, putting points into ptArray - 부록 1 }
18.     return ptArray
```

각 이미지에 대하여 GetOutline 함수와 OutlineToSequence 함수를 수행하여 얻은 여러 개의 sequence들이 바로 다음 단계에서 approximate matching의 대상이 된다.

14행에서는 Sequence Matching 알고리즘을 간단하게 하기 위해 모두 한 방향으로만 맞춰 준다.

### 5.2 Approximate Sequence Matching

전처리 과정에서 얻어진 sequence 데이터를 바탕으로 Multiple Approximate Sequence Matching을 수행한다.

Sequence들을 길이 순서대로 배열하여 두 개씩 비교해 나가는데 항상 긴 것(앞으로 B라고 표현)을 짧은 것(앞으로 A라고 표현)에 대해 $s$만큼 움직여가면서

(shifting) 읽으며, 비교할 때 두 sequence는 서로 방향이 뒤집혀 있어야 한다. 이유는 Fig.6을 참고한다.

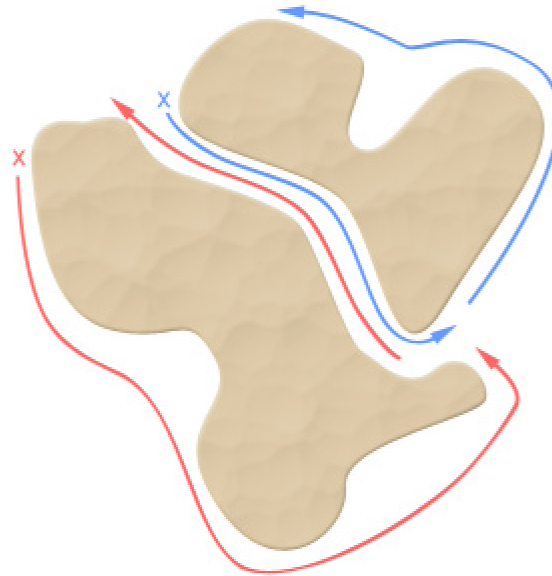

Fig.6 두 sequence를 비교할 때 서로 뒤집혀 있어야 하는 이유

B를 A에 대해 움직이는 것(shifting)은 Fig.7과 같은 원리이다.

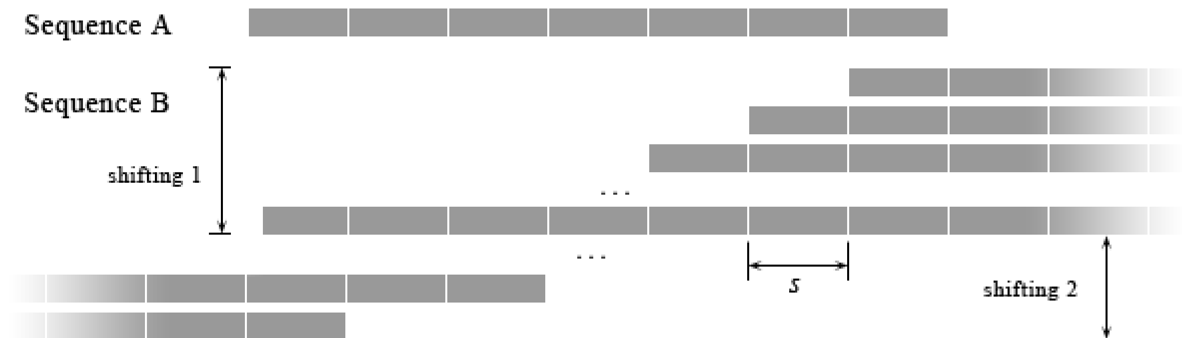


Fig.7 A와 B의 shifting 과정 모식도

일단 이에 대한 알고리즘을 의사 코드로 기술하면 다음과 같다.

```
1.  global s, g, m, d    // system parameters
2.  function DoMatch(sequences[])
3.      { sort sequences by length }
4.      for i ← 0 to sequences.count - 1
```

```
5.          for j ← i + 1 to sequences.count
6.               Match2Seq(sequences[i], sequences[j])
7.     return result
8.
9. function Match2Seq(seq1, seq2)
10.    { reverse seq1 temporarily }   // seq1이 짧은 쪽이므로 seq1을 뒤집는다
11.    //// shifting 1 ////
12.    c ← 1
13.    for i ← 0 to seq1.ubound step s
14.         a ← seq1.ubound - s * c
15.         b ← 0
16.         la ← 0, lb ← 0
17.         starta ← a, startb ← b
18.         for j ← 0 to seq1.ubound step 1    // each elements
19.              { seq1[a]와 seq2[b]를 탐색하면서 서로 match되는 element가 g+1
 이내에 있으면 subsequence로 인정하여 연장하고, 그보다 벗어나면 subsequence 연장을
 종료하며, subsequence 길이가 m 이상일 때만 result로 추가한다. : 1차 제출 - 미완성 }
20.              if a > seq1.ubound then a ← 0
21.         c ← c + 1
22.    //// shifting 2 ////
23.    c ← 1
24.    for i ← 0 to seq1.ubound step s
25.         b ← s * c
26.         starta ← 0, startb ← b
27.         for a ← 0 to seq1.ubound step 1   // each elements
28.              { do same as shifting1 with seq1[a] and seq2[b] }
29.              if b > seq2.ubound then b ← 0
30.         c ← c + 1
31.    return result
32.
33. function IsMatched(e1, e2)
34.    if (the difference of e1 and e2 is less than d) then
35.         return true
36.    else
```

```
37.         return false
```

DoMatch 함수에서는 짧은 것과 긴 것을 비교할 수 있는 모든 경우의 수를 for 루프로 돌며 Match2Seq 함수를 호출하며, 이 함수는 실제 가장 중요한 Maximum Gap을 고려한 Approximate Sequence Matching을 하게 된다.

### 5.3 결과 출력

Sequence Matching 결과를 바탕으로, 다시 원래의 외곽선 정보에 이 결과를 표현하여 실제 사용자가 어디와 어디를 맞추어야 할지 원래 Image와 함께 화면에 출력한다. 다음은 하나의 outline과 그걸 토대로 만들어졌던 sequence에 대한 결과 출력 알고리즘이다.

```
1. function OutputResult(image, outline, sequence, result)
2.     subseq ← { find all subsequences from result within this sequence }
3.     for i ← 0 to subseq.count - 1
4.         for j ← subseq[i].firstIndex + 1 to subseq[i].lastIndex
5.             DrawLine(image, _
               outlines[subseq[i].elements[j-1].PointIndex], _
               outlines[subseq[i].elements[j].PointIndex])
```

## IV. 결론 및 제언

### 1. 연구의 의의

이 연구를 통해 Sequence Matching 기법을 생물정보학 분야가 아닌 영상 처리 분야로 그 응용 범위를 확대시켰다. 또한 영상 처리 분야의 특성에 맞게 다시 설계된 Sequence Matching 알고리즘이 기존의 영상 처리 방법들에 비해 어떤 장점을 가지는지 살펴보았다. Image의 외곽선을 따라 sampling하여 만든 sequence를 이용하면 이미지의 기하학적 변환을 고려할 필요가 없어져 보다 간단한 구현이 가능해진다는 장점이 있다. Sequence Matching 알고리즘을 크게 최적화한다면 성능도 향상될 것이라 기대된다.

### 2. 앞으로의 연구 방향

본 연구에서는 이미지의 색상 정보만을 이용하였지만 실제 고대 토기들은 무늬가 거의 없어 색상 정보보다는 외곽선 형태 정보에 의해서만 맞출 수 있는 경우가 많지 않아 그런 경우에는 적용하기 어렵다. 그 이후 시대의 것이더라도 무늬가 없이 빈 부분은 적용할 수 없다. 따라서 조각 외곽선 형태 정보(부분적 곡률 정보 등)를 sequence에 포함시켜야 할 것이다.

추가로, 전처리 과정을 최적화하여 성능을 개선하고, Sequence Matching 알고리즘을 더욱 효율적으로 설계하여 더욱 빠른 속도로 정확하게 결합 부위를 찾아낼 수 있도록 해야 할 것이다. 그리고 실제 박물관 등에서 응용하려면 항상 최대의 %로 matching되는 것이 아닌 것들도 List로 나열하여 실제 사용자가 프로그램 상에서 비교해 보고 틀린 것은 수동으로 선택할 수 있게 하는 방법도 포함되어야 할 것이며 인공지능 방식을 도입한 Expert System 형태로 시대별 특징이나 문양 등도 고려하여 matching하는 것이 필요하다.

## V. 참고 문헌

[1] Rafael C. Gonzalez, Richard E. Woods (하영호 외 3명 공역), *디지털 영상 처리 (2nd Edition)*, 도서출판 그린, 2003, pp 52-56

[2] Thomas H. Cormen 외 3명, *Introduction To Algorithms (2nd Edition)*, The MIT Press, 2001, pp 350-351

[3] Dan Gusfield, *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, 1999, pp 215-216, pp 332-336

[4] 강동중, 하종은, *Visual C++을 이용한 디지털 영상처리*, SciTech Media, 2003, pp 261-269

[5] Jack Bresenham, "*Algorithm for computer control of a digital plotter*", IBM Systems Journal, vol.4 no.1, pp 25-30, 1965

## VI. 부록

1. Bresenham's Line Algorithm

< 부록 1 - Bresenham's Line Alogrithm >

Bresenham's Line Alogrithm은 픽셀 좌표계에서 정수 계산만으로 직선을 그리는 것으로 1초 수천 개의 Random 직선을 그을 수 있을 정도의 성능을 가진다.

원리는 x1부터 x2까지 가면서 y값을 1을 더해야 할지 말아야 할지 결정하는 two_dx_error를 정수로만 계산하는 것이 핵심이다.

본문에서 생략했던 GetPointsOfLine 함수의 내용은 다음과 같다.

```
1. function GetPointsOfLine(pt1, pt2)
2.     ptArray ← new Point Array[]
3.     dx ← pt1.x - pt2.x, dy ← pt1.y - pt2.y
4.     two_dx ← 2 * dx,    two_dy ← 2 * dy
5.     currentX ← pt1.x,   currentY ← pt1.y
6.     xinc ← 1,           yinc ← 1
7.     if dx < 0 then xinc ← -1, dx ← -dx
8.     if dy < 0 then yinc ← -1, dy ← -dy
9.     ptArray.add(currentX, currentY)
10.    if dx ≠ 0 or dy ≠ 0 then  // 선 길이가 0이 아니라면
11.        if dy ≤ dx then
12.            two_dx_error ← 0
13.            for currentX ← pt1.x + 1 to pt2.x
14.                two_dx_error ← two_dx_error + 1
15.                two_dy ← two_dy + 1
16.                if two_dx_error > dx then
17.                    currentY ← currentY + 1
18.                    yinc ← yinc + 1
19.                ptArray.add(currentX, currentY)
20.        else
21.            { do same as above, swapping x and y except 'ptArray.add' }
22.    return ptArray
```