# PREVIEW

- 사람 뇌의 영상 분할

(a) 원래 영상 (b) 분할된 영상

그림 5-1 맨 왼쪽 영상을 보고 장면을 서술해 보자.

  - "상자 위에 쌓여있는 형형색색의 파프리카와 가격표"라고 해석하는 과정에서 상자, 파프리카, 가격표, 글자와 같이 인지 과정에서 의미 있는 영역으로 분할
  - 분할한 후 학습을 통해 알고 있는 물체와 매칭을 통한 인식이 동시에 일어남

- 컴퓨터 비전
  - 고품질의 영상 분할 작업이 필요 (영상 검색, 물체 추적, 얼굴 인식, 증강 현실, 동작 인식 등)
  - 현재는 분할 후 인식하는 순차 처리(동시 수행을 추구하는 연구도 있으나 초보 단계)
  - 가장 어려운 문제 중 하나

# 차례

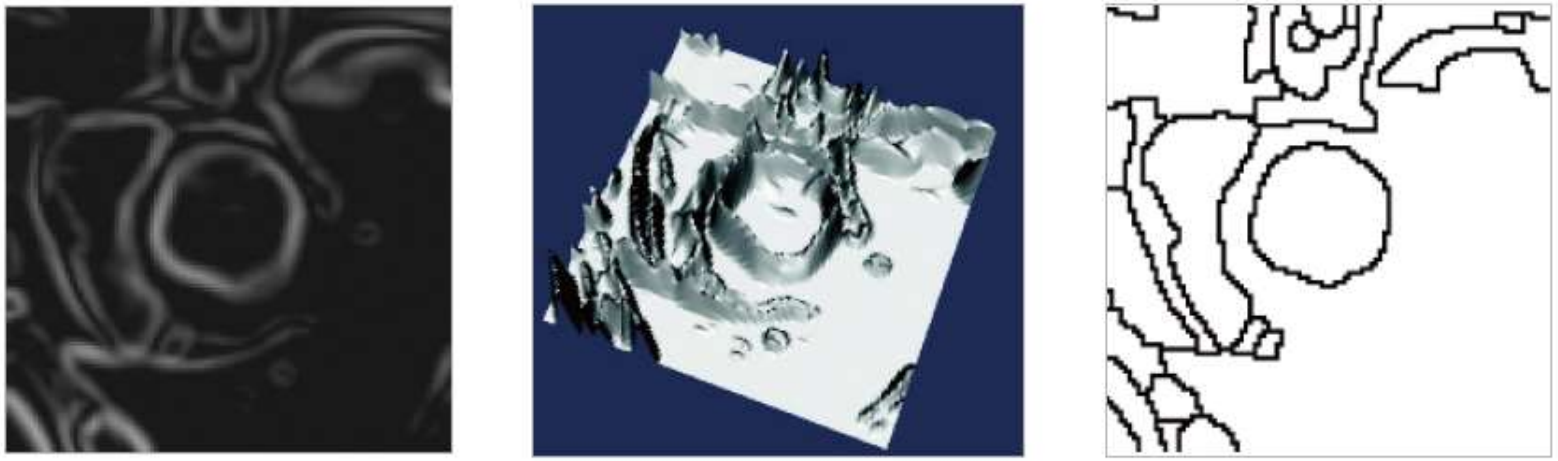
# 4.4.1 배경이 단순한 영상의 영역 분할

## ■ 단순한 영상의 예

- 스캔한 책 영상 또는 컨베이어 벨트 위를 흐르는 물체의 영상

## ■ 영역 분할 알고리즘

- 이진화 알고리즘 (여러 임계값을 사용하는 오츄 알고리즘, adaptive 이진화, 군집화 알고리즘 등) 적용
- 워터셰드 알고리즘 적용



(a) 에지 강도 맵 (b) 지형으로 간주 (c) 워터셰드

그림 4-14 워터셰드 분할 알고리즘[Cousty2007]

# Otsu 삼진화

- 이진화를 이용한 영역 분할
  - 문서 영상의 경우 오츄 이진화(2.3.1절)는 훌륭한 영상 분할 알고리즘
  - 하지만 명암 단계가 둘 이상인 경우는 오작동

- 삼진화로 확장
  - 이중 임계값 사용

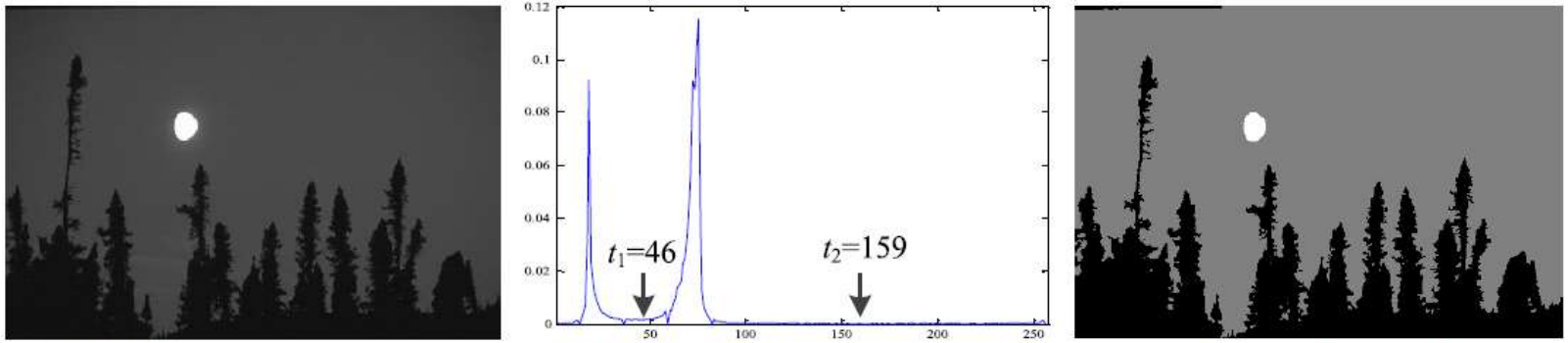
$$v_{between}(t_1, t_2) = w_0(\mu_0 - \mu_g)^2 + w_1(\mu_1 - \mu_g)^2 + w_2(\mu_2 - \mu_g)^2$$

이때, $\mu_0 = \frac{1}{w_0}\sum_{i=0}^{t_1} i\hat{h}_i$, $\mu_1 = \frac{1}{w_1}\sum_{i=t_1+1}^{t_2} i\hat{h}_i$, $\mu_2 = \frac{1}{w_2}\sum_{i=t_2+1}^{L-1} i\hat{h}_i$ (5.2)

$$w_0 = \sum_{i=0}^{t_1} \hat{h}_i\,, \quad w_1 = \sum_{i=t_1+1}^{t_2} \hat{h}_i\,, \quad w_2 = \sum_{i=t_2+1}^{L-1} \hat{h}_i$$

$$(\dot{t}_1, \dot{t}_2) = \underset{0<t_1<t_2<L-1}{\operatorname{argmax}}\, v_{between}(t_1, t_2) \tag{5.3}$$

# Otsu 삼진화




(a) 원래 영상 (b) 명암 히스토그램 (c) 이중 임계값으로 분할한 영상

그림 5-3 이중 임계값 오츄 알고리즘에 의한 영상 분할

**알고리즘 5-1 이중 임계값 오츄 알고리즘**

**입력 :** 영상 $f(j,i), 0 \le j \le M-1, 0 \le i \le N-1$

**출력 :** 삼진영상 $g(j,i), 0 \le j \le M-1, 0 \le i \le N-1$ // 0, 1, 2 세 가지 값을 가진 영상

1 [알고리즘 2-1]을 이용하여 $f$의 정규 히스토그램 $\hat{h}$을 만든다.
2 for($t_1$=1 to $L$-3)
3 for($t_2$=$t_1$+1 to $L$-2)
4 식 (5.2)를 이용하여 $v_{between}(t_1, t_2)$를 계산한다.
5 2~4행에서 가장 큰 $v_{between}$을 생성한 $(t_1, t_2)$를 임계값 $(\dot{t}_1, \dot{t}_2)$로 취한다.
6 $(\dot{t}_1, \dot{t}_2)$로 $f$를 삼진화하여 $g$를 만든다.

# 임계화를 이용한 영역 분할

- 적응적 임계화
  - 하나 또는 두 개의 임계값을 영상 전체에 적용하는 전역 방법의 한계
    - 지역적으로 명암 분포가 다른 경우 낮은 분할 품질


(a) 원래 영상 (b) [알고리즘 2-4]를 적용한 결과 (c) [알고리즘 5-1]을 적용한 결과

그림 5-4 효모 영상과 임계화한 영상

이진화 실패한 영상

  - 적응적 임계화로 해결: 지역에 따라 적응적으로 임계값 결정
    - 각 화소의 이웃을 사용하는 지역 임계값 사용
    - $t(j,i)$를 어떻게 결정할까? 지역 블록을 설정

$$b(j,i) = \begin{cases} 1, & f(j,i) \geq t(j,i) \\ 0, & \text{그렇지 않으면} \end{cases} \qquad (5.4)$$

```
import cv2 as cv
#%%
gray=cv.imread("book.bmp",0); print(gray.shape)
cv.imshow("Gray", gray); cv.waitKey()

t, binaryFixed=cv.threshold(gray, 70, 255, cv.THRESH_BINARY)
cv.imshow("BinaryFixed", binaryFixed), cv.waitKey()

t, binaryOtsu=cv.threshold(gray, 70, 255, cv.THRESH_BINARY +
cv.THRESH_OTSU); print(t)
cv.imshow("binaryOtsu", binaryOtsu), cv.waitKey()

blockSize=21; C=10
binaryAdaptive=cv.adaptiveThreshold(gray, 255,
cv.ADAPTIVE_THRESH_MEAN_C, cv.THRESH_BINARY, blockSize, C)
cv.imshow("binaryAdaptive", binaryAdaptive), cv.waitKey()
cv.destroyAllWindows()
```

- void adaptiveThreshold(InputArray **src**, OutputArray **dst**, double **maxValue**, int **adaptiveMethod**, int **thresholdType**, int **blockSize**, double **C**)
  - **src** – Source 8-bit single-channel image.
  - **dst** – Destination image of the same size and the same type as src .
  - **maxValue** – Non-zero value assigned to the pixels for which the condition is satisfied. See the details below.
  - **adaptiveMethod** – Adaptive thresholding algorithm to use, ADAPTIVE_THRESH_MEAN_C or ADAPTIVE_THRESH_GAUSSIAN_C .
    - ADAPTIVE_THRESH_MEAN_C , the threshold value is a mean of the neighborhood of minus C .
    - ADAPTIVE_THRESH_GAUSSIAN_C , the threshold value is a weighted sum (cross-correlation with a Gaussian window) of the neighborhood of minus C
  - **thresholdType** – Thresholding type that must be either THRESH_BINARY or THRESH_BINARY_INV .
  - **blockSize** – Size of a pixel neighborhood that is used to calculate a threshold value for the pixel: 3, 5, 7, and so on.
  - **C** – Constant subtracted from the mean or weighted mean (see the details below). Normally, it is positive but may be zero or negative as well.
  - (예제) cv::adaptiveThreshold(image, binaryAdaptive, 255, cv::ADAPTIVE_THRESH_MEAN_C, cv::THRESH_BINARY, blockSize, threshold);

**Python:**

cv.adaptiveThreshold( src, maxValue, adaptiveMethod, thresholdType, blockSize, C[, dst] ) -> dst

# 워터셰드

- 워터셰드란?
  - 워터셰드 (분수계): 빗물이 안쪽과 바깥쪽 중 하나로 흐를 수 있는 점(붉은선)
    - 영상에서는 에지 강도가 상대적으로 큰 지점
  - 유역(골짜기): 같은 호수로 빗물이 모이는 점들의 집합

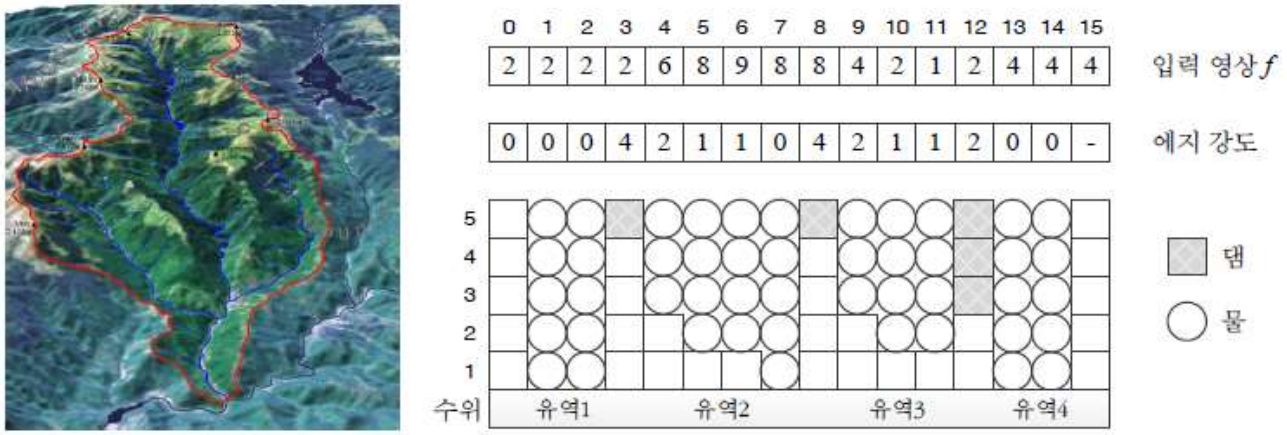


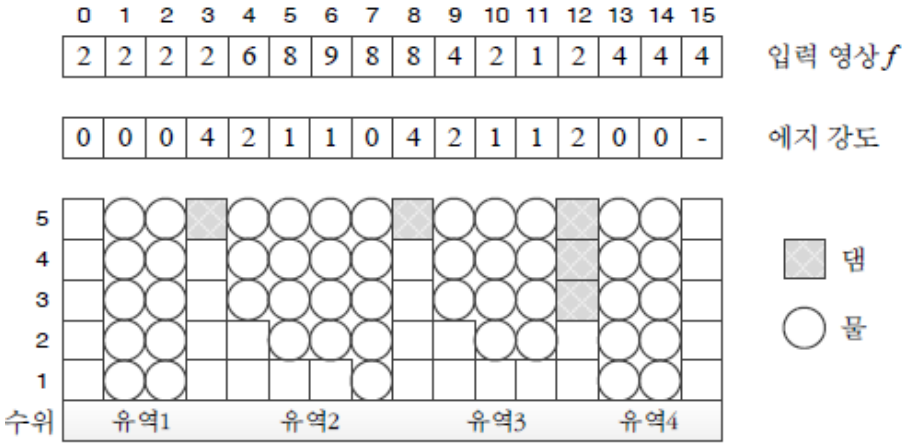

그림 5-16 워터셰드를 이용한 영상 분할의 원리

  - 위치 3, 8, 12는 워터셰드, 1~2, 4~7, 9~11, 13~14는 유역

- 영상 분할에 적용
  - 에지 강도 맵의 워터셰드는 두 영역을 가르는 경계에 해당
  - 에지 강도 맵에서 워터셰드를 어떻게 찾을 것인가?

# 워터셰드

- 댐 건설 방법 [그림 5-16]
  - 영상 분할 = 에지 강도 맵에서 워터셰드를 찾는 과정 → 댐 건설 방법
  - 댐 = 워터셰드 = 영상 분할 경계면




  - [그림 5-16]에서 물을 주입하면 수위1에 물이 고인다 → 유역 1, 2, 4에 호수 생성
  - 물을 더 넣어 수위2까지 채우면 → 유역 3에 호수 생성
  - 수위가 3이 되면 → 유역3과 4가 범람해 합쳐짐. → 댐이 필요!

....

  - 수위가 5가 되면 → 유역 1과 2, 유역 2와 3, 유역 3과 4가 범람 → 각각 댐이 필요!
  - 그리고 최고 수위가 되므로 알고리즘 멈춤

- 워터쉐드는 영상을 과도하게 분할하는 단점
- OpenCV 워터쉐드 알고리즘
  - 미리 정의된 마커를 사용하여 특정 영역에 소속된 화소를 표시(초기 레이블링)
    - 예)전경: 255, 배경: 128(0과 255가 아닌 임의의 값), 레이블링 과정에서 알수 없는 화소: 0
  - 마커 영상을 그레이레벨 영상으로 생성
  - 정수형 영상으로 변형(OpenCV 함수가 워터쉐드에 -1을 할당)

step1: 입력 영상 이진화
step2: 중요한 객체 속하는 화소를 유지하도록 과도한 침식 : fg
객체화소: 255 객체가 아닌 화소 : 0
step3: 배경에 속하는 화소를 유지하도록 과도한 팽창 : bg
배경 화소: 128, 어디 소속인지 불 명확한 다른 화소: 0
step4: 마커 영상 생성 : markers = fg + bg;
step5: 정수 영상으로 변환 후 cv::watershed(image, marker); 사용

cv2.getStructuringElement(shape, ksize, anchor=None) -> retval

- shape: 구조 요소 kernel 모양 (cv2.MORPH_RECT, cv2.MORPH_ELIPSE, cv2.MORPH_CROSS)
- ksize: kernel 크기
- anchor: cv2.MORPH_CROSS 에만 적용, 구조 요소의 기준점(default: 중심)

# OpenCV 함수

- void watershed(InputArray **image**, InputOutputArray **markers**)
  - **image** – Input 8-bit 3-channel image.
  - **markers** – Input/output 32-bit single-channel image (map) of markers. It should have the same size as image
    - The markers are "seeds" of the future image regions. All the other pixels in markers , whose relation to the outlined regions is not known and should be defined by the algorithm, should be set to 0's. In the function output, each pixel in markers is set to a value of the "seed" components or to -1 at boundaries between the regions.

**Python:**

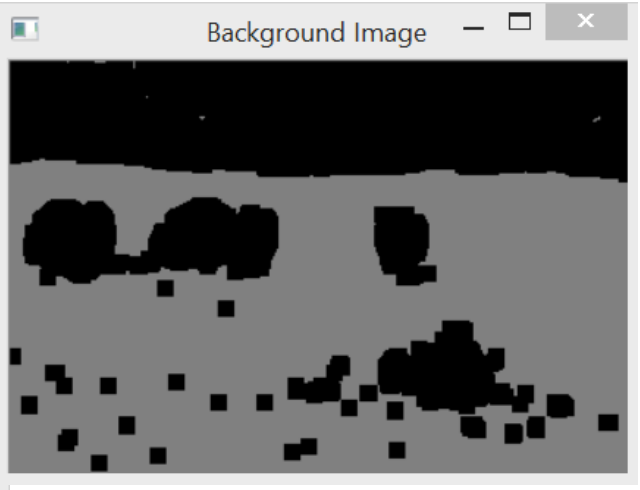
```
cv.watershed( image, markers ) -> markers
```

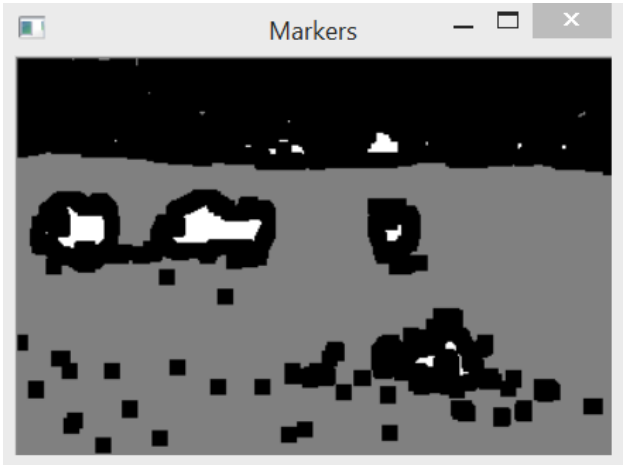
```
import cv2 as cv
import numpy as np
#%%
img=cv.imread("group.jpg")
gray=cv.cvtColor(img, cv.COLOR_BGR2GRAY)
t, binary=cv.threshold(gray, 0,255, cv.THRESH_BINARY_INV + cv.THRESH_OTSU) ; print(t)

cv.imshow("Color", img); cv.imshow("Gray", gray); cv.imshow("Binary", binary)
cv.waitKey(); cv.destroyAllWindows()

kernel=cv.getStructuringElement(cv.MORPH_RECT, (3,3))
fg=cv.erode(binary,kernel, iterations= 6)
cv.imshow("Foreground", fg); cv.waitKey()
bg=cv.dilate(binary, kernel, iterations=6)
t, bg=cv.threshold(bg, 1, 128, cv.THRESH_BINARY_INV)
cv.imshow("Background", bg); cv.waitKey()
```

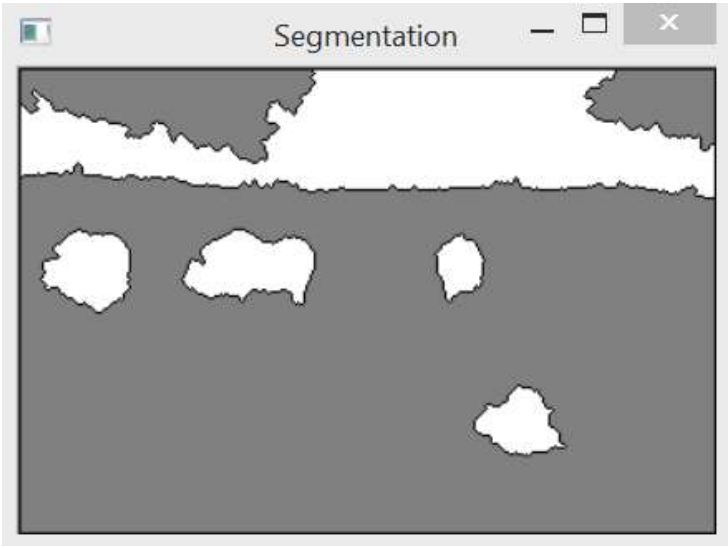
```
markers=fg+bg
cv.imshow("Markers", markers); cv.waitKey()

markers=np.int32(markers)
markers=cv.watershed(img, markers)
np.max(markers); np.min(markers)

dst=np.ones(img.shape, np.uint8)*255
dst[markers==-1] = 0
cv.imshow("Watershed", dst)
img[markers==-1]=[255,0,0]
cv.imshow("Color&Boundry", img); cv.waitKey()

markers=np.uint8(np.clip(markers, 0, 255))
cv.imshow("Segmentation", markers);cv.waitKey()

cv.destroyAllWindows()
```

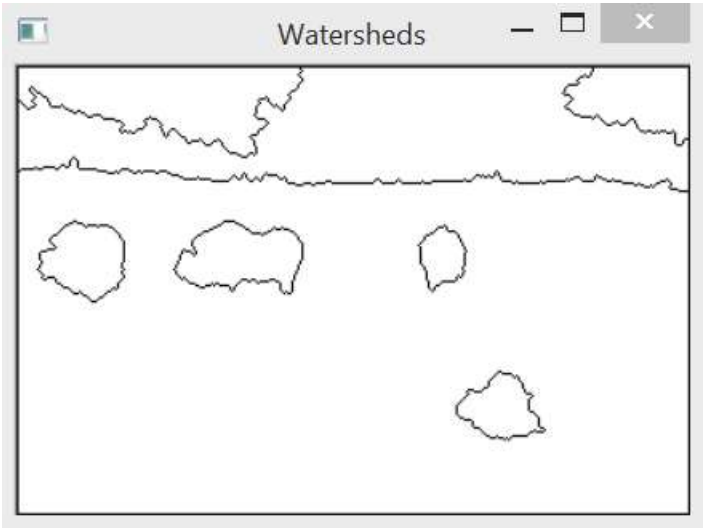
# 워터쉐드


Foreground Image


Background Image


Markers


Segmentation


Watersheds

# 과제

- 워터쉐드 알고리즘을 이용하여 hand_sample2.jpg에서 손의 윤곽을 찾아라

# 4.4.2 슈퍼 화소 분할

- **슈퍼 화소는 화소보다 크지만 물체보다 작은 자잘한 영역으로 과잉 분할**
  - A superpixel can be defined as a group of pixels that share common characteristics (like pixel intensity ).
  - Superpixels are becoming useful in many Computer Vision and Image processing algorithms like Image Segmentation, Semantic labeling, Object detection and tracking etc
- **SLIC**$_{\text{simple linear iterative clustering}}$ **알고리즘**
  - k-평균 군집화와 비슷하게 동작 (화소 할당 단계와 군집 중심 갱신하는 단계를 반복: 군집 주임의 이동량의 평균이 임계치 보다 작으면 수렴했다고 판단함)
  - SLIC은 (R,G,B,x,y) 5차원 벡터를 사용하여 군집화 수행
  - 예 : k=8인 경우

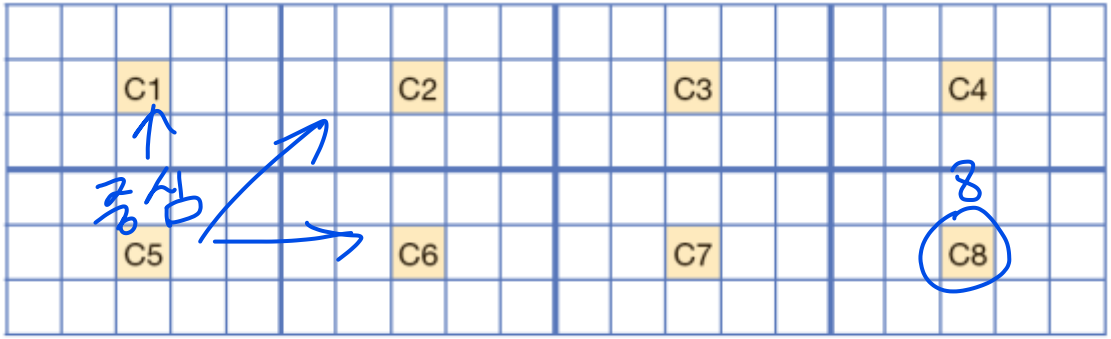



그림 4-15 SLIC 알고리즘의 초기 군집 중심

# 4.4.2 슈퍼 화소 분할

**프로그램 4-5** SLIC 알고리즘으로 입력 영상을 슈퍼 화소 분할하기

```
01  import skimage
02  import numpy as np
03  import cv2 as cv
04
05  img=skimage.data.coffee()
06  cv.imshow('Coffee image',cv.cvtColor(img,cv.COLOR_RGB2BGR))
07
08  slic1=skimage.segmentation.slic(img,compactness=20,n_segments=600)
09  sp_img1=skimage.segmentation.mark_boundaries(img,slic1)
10  sp_img1=np.uint8(sp_img1*255.0)
11
12  slic2=skimage.segmentation.slic(img,compactness=40,n_segments=600)
13  sp_img2=skimage.segmentation.mark_boundaries(img,slic2)      # 0~1 사이 값을 갖음
14  sp_img2=np.uint8(sp_img2*255.0)
15
16  cv.imshow('Super pixels (compact 20)',cv.cvtColor(sp_img1,cv.COLOR_RGB2BGR))
17  cv.imshow('Super pixels (compact 40)',cv.cvtColor(sp_img2,cv.COLOR_RGB2BGR))
18
19  cv.waitKey()
20  cv.destroyAllWindows()
```

# 4.4.2 슈퍼 화소 분할


Coffee Image
Super pixels (compact 20)
Super pixels (compact 40)
compactness=20
compactness=40

```
skimage.segmentation.slic(image, n_segments=100, compactness=10.0,
max_num_iter=10, sigma=0, spacing=None, convert2lab=None,
enforce_connectivity=True, min_size_factor=0.5, max_size_factor=3,
slic_zero=False, start_label=1, mask=None, *, channel_axis=-1)
```
[source]

Segments image using k-means clustering in Color-(x,y,z) space.

**Parameters:**

**image** : *2D, 3D or 4D ndarray*

Input image, which can be 2D or 3D, and grayscale or multichannel (see *channel_axis* parameter). Input image must either be NaN-free or the NaN's must be masked out

**n_segments** : *int, optional*

The (approximate) number of labels in the segmented output image.

**compactness** : *float, optional*

Balances color proximity and space proximity. Higher values give more weight to space proximity, making superpixel shapes more square/cubic. In SLICO mode, this is the initial compactness. This parameter depends strongly on image contrast and on the shapes of objects in the image. We recommend exploring possible values on a log scale, e.g., 0.01, 0.1, 1, 10, 100, before refining around a chosen value.

**max_num_iter** : *int, optional*

Maximum number of iterations of k-means.

# scikit-image

scikit-image는 이미지처리에 특화된 Python 이미지 라이브러리이며 Numpy배열로 이미지 객체를 네이티브하게 다룹니다.
pip 또는 Anaconda를 통해 설치를 진행합니다.

```
pip install scikit-image
conda install -c conda-forge scikit-image
```

```
skimage.segmentation.mark_boundaries(image, label_img, color=(1, 1, 0),
outline_color=None, mode='outer', background_label=0)                [source
```

Return image with boundaries between labeled regions highlighted.

If you have an object embedded in some background, "inner" puts the boundary on the border pixels of that object, while "outer" puts the boundary around the object in the background.

**Parameters:**

**image** : *(M, N[, 3]) array*

Grayscale or RGB image.

**label_img** : *(M, N) array of int*

Label array where regions are marked by different integer values.

**color** : *length-3 sequence, optional*

RGB color of boundaries in the output image.

**outline_color** : *length-3 sequence, optional*

RGB color surrounding boundaries in the output image. If None, no outline is drawn.

**mode** : *string in {'thick', 'inner', 'outer', 'subpixel'}, optional*

The mode for finding boundaries.

# 4.4.3 최적화 분할

- **지금까지 분할 알고리즘은 지역적 명암 변화만 살피기 때문에 한계**
  - 예) 양말의 색이 배경과 비슷하면 양말이 배경 영역에 섞임
- **전역적 정보를 고려하여 문제 해결**
  - 지역적으로 색상 변화가 약하지만 전역적으로 유리하다면 물체 경계로 간주
  - 영상을 그래프로 표현하고 최적화 알고리즘으로 분할 문제를 품

# 4.4.3 최적화 분할

## ■ 영상의 그래프 표현

- 화소 또는 슈퍼 화소를 노드로 취함
- 두 노드의 유사도를 식 (4.8)로 계산하여 에지에 부여
  - $f(v)$는 $v$에 해당하는 화소의 색상(r,g,b)와 위치 (x,y)를 결합한 5차원 벡터
  - $v$가 슈퍼 화소인 경우 화소 평균을 사용

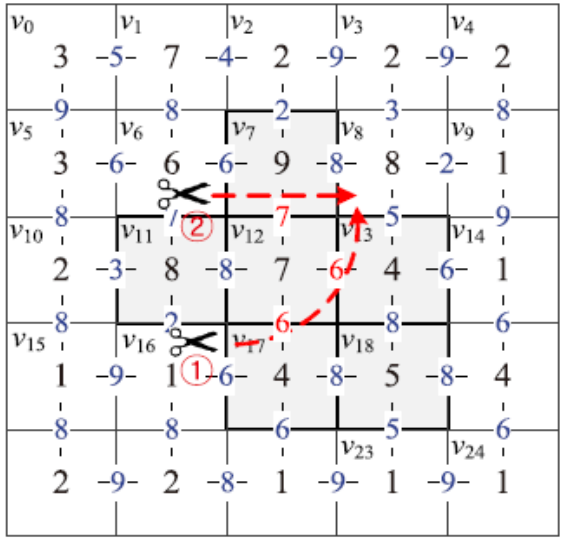
$$\text{거리} \begin{cases} d_{pq} = \left\| f(v_p) - f(v_q) \right\|, \text{ 만일 } v_q \in neighbor(v_p) \\ \infty, \text{ 그렇지 않으면} \end{cases}$$

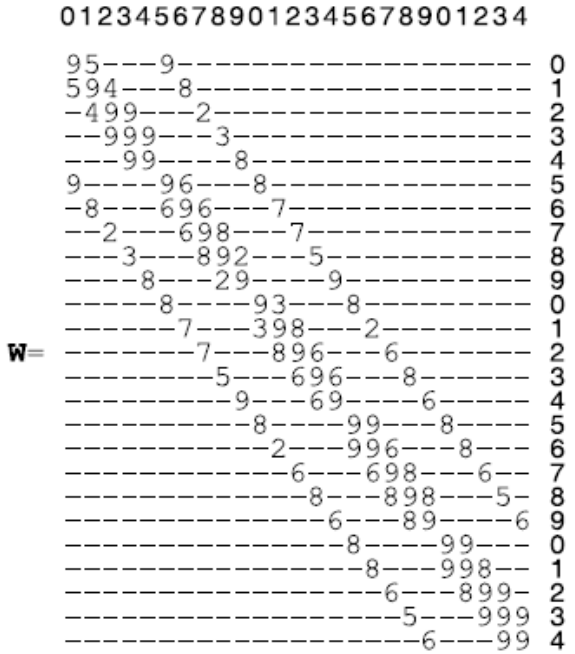
$$\text{유사도} \begin{cases} s_{pq} = D - d_{pq} \text{ 또는 } \dfrac{1}{e^{d_{pq}}}, \text{ 만일 } v_q \in neighbor(v_p) \\ 0, \text{ 그렇지 않으면} \end{cases} \quad (4.8)$$

- 그래프: $G=(V,E)$
  - $V$: 노드 집합 $V=\{v_1, v_2, \ldots, v_n\}$ , n: 화소 또는 슈퍼 화소 개수
  - $E$: 에지(간선) 집합 , 이웃 노드 간에 에지(간선) 설정

# 4.4.3 최적화 분할

- 유사도 그래프와 Wu의 분할 품질 척도
  - 예) 간선에 유사도 표시: 거리가 $d$ 일 때 9-$d$ 로 유사도 계산




왼쪽 그림과 같이 5x5 영상을 그래프로 표현하고 연결요소 C를 2개의 물체로 $C_1$, $C_2$로 분할하고자 한다.

(1)방법 분할: 유사도인 cut값이 12
(2)방법 분할: 유사도인 cut값이 7
cut 값이 적을 수록 유리한 분할

그러나=>(1)이 더 좋은 분할임에도 작은 크기의 연결요소로 분할되는 경향

그림 5–11 유사도 그래프와 인접 행렬

- Wu의 분할 품질 척도 cut [Wu93]
  - $cut(C_1, C_2) = \sum_{v_p \in C_1, v_q \in C_2} s_{pq}$
    - $v_p$와 $v_q$를 연결하는 에지는 가중치 $s_{pq}$
    - 가중치 $s_{pq}$ 는 유사도

# 5.3.2 정규화 절단

- Shi의 정규화 절단 **normalized cut (ncut)** 알고리즘 [Shi2000]
  - **cut**의 문제점
    - 원래 연결요소를 작은 것과 큰 것으로 분할하는 경향
    - 두 연결요소 간의 에지 개수가 적으므로 **cut**이 작아짐
    - 예) 분할 (1)의 cut은 12, 분할 (2)는 7 → 하지만 직관적으로 분할 (1)이 우수
  - Shi는 정규화 절단 **ncut**으로 확장하여 문제 해결
    - **ncut**은 **cut**을 정규화하여 영역의 크기에 중립이 되게 해 줌

$$ncut(C_1, C_2) = \frac{cut(C_1, C_2)}{cut(C_1, C)} + \frac{cut(C_1, C_2)}{cut(C_2, C)} \qquad (4.10)$$

$$C = C_1 \cup C_2$$

  - **ncut**이 최소가 되는 분할을 찾는 문제는 NP-complete 근사적

# 4.4.3 최적화 분할

프로그램 4-6 정규화 절단 알고리즘으로 영역 분할하기

```
01  import skimage
02  import numpy as np
03  import cv2 as cv
04  import time
05
06  coffee=skimage.data.coffee()
07
08  start=time.time()
09  slic=skimage.segmentation.slic(coffee,compactness=20,n_segments=600,start_
                                   label=1)
10  g=skimage.graph.rag_mean_color(img, slic, mode='similarity')
11  ncut=skimage.graph.cut_normalized(slic, g)  # 정규화 절단
12  print(coffee.shape,' Coffee 영상을 분할하는 데 ',time.time()-start,'초 소요')
13
14  marking=skimage.segmentation.mark_boundaries(coffee,ncut)
15  ncut_coffee=np.uint8(marking*255.0)
16
17  cv.imshow('Normalized cut',cv.cvtColor(ncut_coffee,cv.COLOR_RGB2BGR))
18
19  cv.waitKey()
20  cv.destroyAllWindows()
```

# 4.4.3 최적화 분할

```
(400, 600, 3) Coffee 영상을 분할하는 데 6.43808341026306l5초 소요
```


- **고전 영역 분할 알고리즘은 색상 정보에만 의존하므로 의미 분할 불가능**
  - 사람은 물체의 3차원 모델과 2차원 겉모습 모델$_{\text{appearance model}}$을 동시에 사용하여 의미 분할을 수행함
  - 9장에서는 딥러닝을 이용한 의미 분할을 다룸

skimage.graph.**rag_mean_color**(*image, labels, connectivity=2, mode='distance', sigma=255.0*) [source]

Compute the Region Adjacency Graph using mean colors.

Given an image and its initial segmentation, this method constructs the corresponding Region Adjacency Graph (RAG). Each node in the RAG represents a set of pixels within *image* with the same label in *labels*. The weight between two adjacent regions represents how similar or dissimilar two regions are depending on the *mode* parameter.

**Parameters:**

**image : *ndarray, shape(M, N, [..., P,] 3)***

Input image.

**labels : *ndarray, shape(M, N, [..., P])***

The labelled image. This should have one dimension less than *image*. If *image* has dimensions *(M, N, 3)* *labels* should have dimensions *(M, N)*.

**connectivity : *int, optional***

Pixels with a squared distance less than *connectivity* from each other are considered adjacent. It can range from 1 to *labels.ndim*. Its behavior is the same as *connectivity* parameter in `scipy.ndimage.generate_binary_structure`.

**mode : *{'distance', 'similarity'}, optional***

The strategy to assign edge weights.

> 'distance' : The weight between two adjacent regions is the $|c_1 - c_2|$, where $c_1$ and $c_2$ are the mean colors of the two regions. It represents the Euclidean distance in their average color.
> 'similarity' : The weight between two adjacent is $e^{-d^2/sigma}$ where $d = |c_1 - c_2|$, where $c_1$ and $c_2$ are the mean colors of the two regions. It represents how similar two regions are.

skimage.graph.cut_normalized(*labels*, *rag*, *thresh=0.001*, *num_cuts=10*, *in_place=True*, *max_edge=1.0*, ***, *rng=None*) [source]

Perform Normalized Graph cut on the Region Adjacency Graph.

Given an image's labels and its similarity RAG, recursively perform a 2-way normalized cut on it. All nodes belonging to a subgraph that cannot be cut further are assigned a unique label in the output.

**Parameters:**

**labels** : *ndarray*

The array of labels.

**rag** : *RAG*

The region adjacency graph.

# 4.5 대화식 분할

- 앞에서 다룬 분할 알고리즘은 영상 전체를 여러 개의 영역으로 분할
- 때로 한 물체의 분할에만 관심이 있는 응용이 있음
  - 예) 특정 물체를 오려내고 다른 물체로 대치
  - 사용자가 초기 정보를 입력하면 반자동 분할해주는 여러 알고리즘이 있음

# 4.5.1 능동 외곽선

- 원리
  - 초기 곡선에서 시작하여, 최적 상태(안정적인 에너지 상태)를 능동적으로 찾아감
- 외곽선 표현(매개변수 곡선, parametric curve)
  - 연속 공간에서는 $\mathbf{g}(s)=(y(s),x(s))$, $0 \le s \le 1$
  - 디지털 공간에서는 $\mathbf{g}(s)=(y(s),x(s))$, $s=0,1,2,\ldots,n$

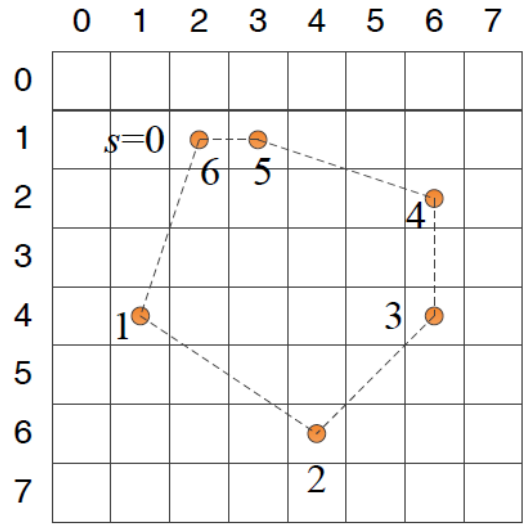
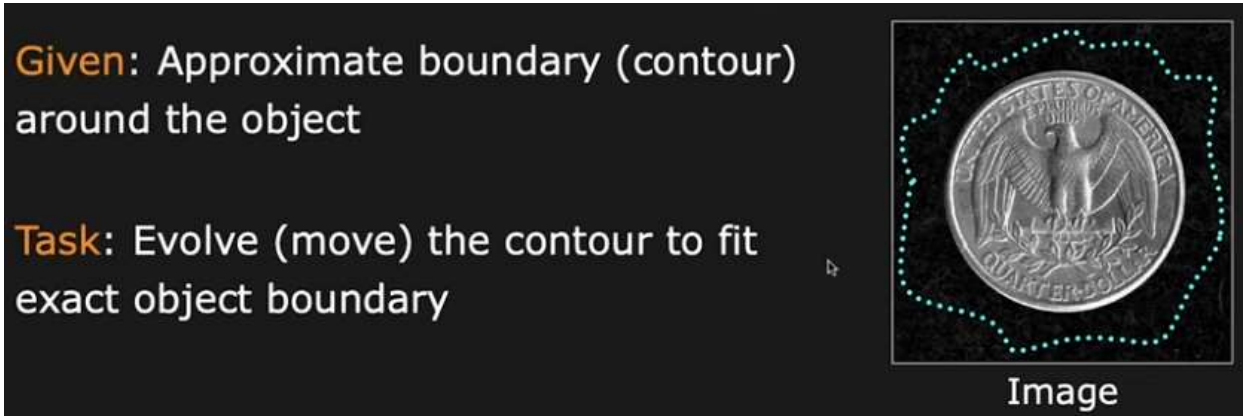



그림 5-18 스네이크를 표현하는 폐곡선 g(s)

$\mathbf{g}(0)=(1,2)$
$\mathbf{g}(1)=(4,1)$
$\mathbf{g}(2)=(6,4)$
$\mathbf{g}(3)=(4,6)$
$\mathbf{g}(4)=(2,6)$
$\mathbf{g}(5)=(1,3)$
$\mathbf{g}(6)=(1,2)$



# 4.5.1 능동 외곽선

계속 변형

- 스네이크 [Kass87]
  - 최초의 능동 외곽선 기법
  - 외곽선의 총 에너지 $E^*$

Iteratively "deform" the initial contour so that:

- It is near pixels with high gradient (edges)
- It is smooth

$$E^*(\mathbf{g}(s)) = \int_0^1 E(\mathbf{g}(s))\,ds = \int_0^1 E_{internal}(\mathbf{g}(s)) + E_{image}(\mathbf{g}(s)) + E_{constraint}(\mathbf{g}(s))\,ds \qquad (5.21)$$

- $E_{image}$: 영상의 명암에 반응하는 항
  - 찾고자 하는 곳은 물체 경계이므로 에지 강도가 클수록 작은 값을 가짐
  - 잡음에 민감
- $E_{internal}$: 곡선의 내부 에너지
  - 물체의 경계는 매끄럽다는 전제하에, 곡률이 클수록 큰 값 가짐
- $E_{constraint}$: 사용자가 지정한 모양을 반영 실제로는 잘 안씀
  - pernalize deviation from prior model of shape

- $E^*$가 최소가 되는 곡선을 찾는 최적화 문제

$$\check{\mathbf{g}}(s) = \underset{\mathbf{g}(s)}{\operatorname{argmin}} E^*(\mathbf{g}(s))$$

# 4.5.1 능동 외곽선

- 외곽선의 에너지 총합 $E^*(g(s))$의 계산
  - Sum of Gradient magnitude is small

$$E_{image}(\mathbf{g}(s)) = -\|\nabla f(\mathbf{g}(s))\|^2$$

  - The change of the curve is small and the rate of change is small

$$E_{internal}(\mathbf{g}(s)) = \frac{\alpha(s)\|\mathbf{g}_s(s)\|^2 + \beta(s)\|\mathbf{g}_{ss}(s)\|^2}{2}$$

$$\|\mathbf{g}_s(s)\|^2 \cong \|\mathbf{g}(s) - \mathbf{g}(s-1)\|^2 = (y(s) - y(s-1))^2 + (x(s) - x(s-1))^2$$

$$\|\mathbf{g}_{ss}(s)\|^2 \cong \|\mathbf{g}(s-1) - 2\mathbf{g}(s) + \mathbf{g}(s+1)\|^2$$

$$= (y(s-1) - 2y(s) + y(s+1))^2 + (x(s-1) - 2x(s) + x(s+1))^2$$

$$E_{internal}(g(s)) = \alpha E_{elastic} + \beta E_{smooth}$$

$$E^*(\mathbf{g}(s)) = \sum_{s=0}^{n} (E_{image}(\mathbf{g}(s)) + E_{internal}(\mathbf{g}(s)))$$

# 4.5.1 능동 외곽선

- 최적해 탐색 알고리즘
  - 초기 곡선 $\mathbf{g}_0(s)$에서 시작하여, 수렴할 때까지 $\mathbf{g}_0(s) \rightarrow \mathbf{g}_1(s) \rightarrow \mathbf{g}_2(s) \rightarrow \ldots$ 반복
  - 동적 프로그래밍 [Amini88]
  - 탐욕 알고리즘 [Williams92]

- Williams 알고리즘
  - 개선된 에너지 수식 사용

$$E^*(\mathbf{g}(s)) = \sum_{s=0}^{n} (\alpha(s)E_{continuity}(\mathbf{g}(s)) + \beta(s)E_{curvature}(\mathbf{g}(s)) + \gamma(s)E_{image}(\mathbf{g}(s)))$$

이때 $E_{continuity}(\mathbf{g}(s)) = \bar{d} - \|\mathbf{g}(s) - \mathbf{g}(s-1)\|$ ($\bar{d}$는 평균 거리)

$$E_{curvature}(\mathbf{g}(s)) = \|\mathbf{g}(s-1) - 2\mathbf{g}(s) + \mathbf{g}(s+1)\|^2 \qquad (5.26)$$

$$E_{image}(\mathbf{g}(s)) = \frac{min - mag}{max - min}$$

스네이크 알고리즘 참조
https://www.youtube.com/watch?v=y3sVTrcxXwc
https://www.youtube.com/watch?v=FROJUMk9P3Y

# 4.5.1 능동 외곽선: deformable contours



Boundary Tracking: Use the boundary from the current image as initial boundary for the next image.

# 4.5.2 그래프 절단

- 네트워크 흐름 문제 [Ahuja93]
  - 교통 흐름, 전기 흐름 등 많은 응용
  - *S*에서 출발하여 *T*로 흐르는 최대 용량은?
    - 병목으로 인해 어떤 에지는 최대 용량 발휘 못함
    - 아래 그래프의 최대 흐름은 6

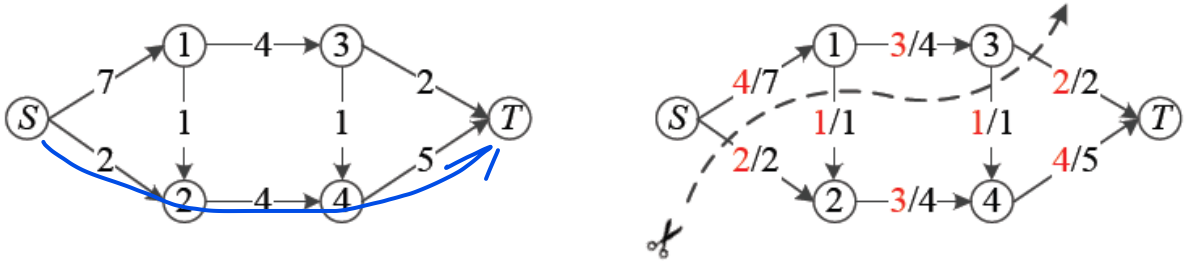



그림 5-20 네트워크 흐름과 그래프 절단

# 4.5.2 그래프 절단

- 영상 분할과 어떻게 관련되나?
  - 영상을 그래프로 변환
    - 영상의 화소를 노드로 간주하고, 유사도를 에지 가중치로 설정
    - 예) [그림 5-20]의 절단은 최소 그래프 절단임
      - {v1, v3}과 {v2, v4}로 분할
      - 이 절단은 최적의 영상 분할임: 그룹내 유사도가 높다
  - 최소 절단을 어떻게 찾을 것인가?
    - Ford의 정리[Ford62]에 따르면, 네트워크 최대 흐름은 그래프의 최소 절단과 같음
    - 최대 흐름 찾는 효율적인 알고리즘 존재(최대 흐름 상태에서 포화 상태의 에지를 절단)

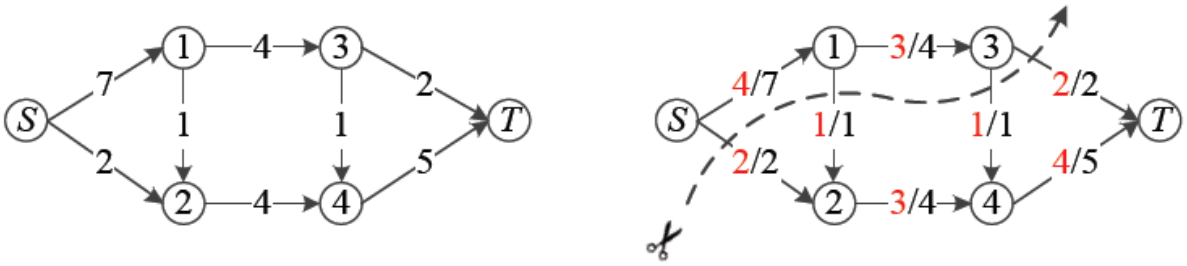



그림 5-20 네트워크 흐름과 그래프 절단

# 4.5.2 그래프 절단

## 영상 분할 알고리즘

- 최대 흐름을 찾으면, Ford의 정리에 따라 그것이 바로 최소 절단이다.
- 최소 절단은 최적의 영상 분할 정보를 가진다.

**알고리즘 5-11 그래프 절단을 이용한 영상 분할**

**입력 :** 명암 영상 $f(j,i)$, $0 \le j \le M-1$, $0 \le i \le N-1$

**출력 :** 물체 $\{C_{object1}, C_{object2}, \cdots\}$와 배경 $\{C_{background1}, C_{background2}, \cdots\}$ // $C$는 연결요소

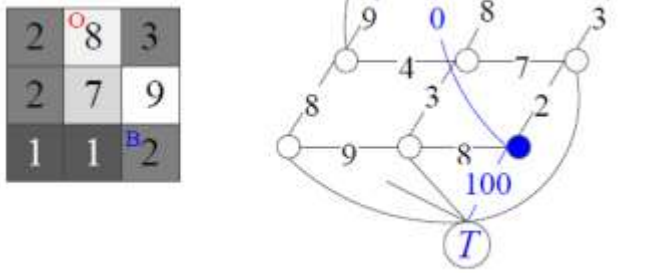
| | |
|---|---|
| 1 | $f$로부터 그래프 $G$를 구축한다. |
| 2 | $G$로부터 최소 절단을 찾는다. |
| 3 | 최소 절단을 영상 분할 결과인 $C_{object}$와 $C_{background}$로 변환한다. |

# 4.5.2 그래프 절단

- 그래프 $G=(V+\{S,T\},E)$를 구축하는 방법 (알고리즘 5-11의 1행)
  - 에지 가중치(유사도)는 아래 수식 사용

$$w_{pq} = e^{-\frac{(f(v_p)-f(v_q))^2}{2\sigma^2}} d(v_p, v_q) \qquad (5.27)$$

영상에 O(물체) 와 B(배경)을 지정한다.



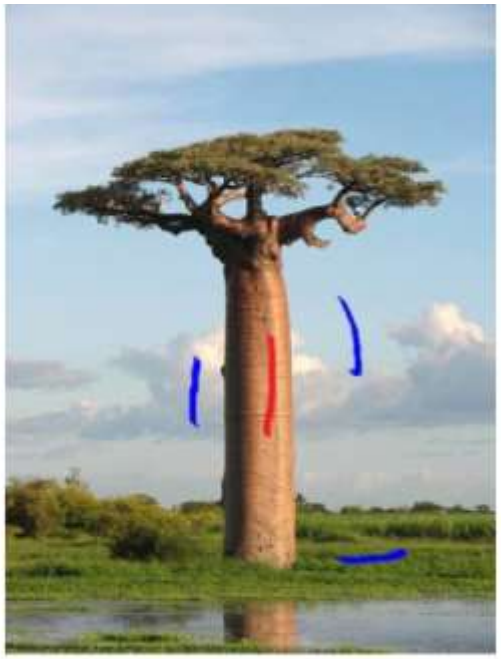



그림 5-21 영상 $f$와 그래프 표현(에지 가중치는 [예제 5-2] 참고)

이 그림에서는 가중치는 $9-|f(v_p)-f(v_q)|$ 로 계산
O와 S, B와 T는 절단되지 않도록 큰 값을 주고, S와 B, T와 O는 그 반대로 0을 준다

# 4.5.2 그래프 절단

- 사용자가 지정한 물체(O)와 배경(B) 정보 반영
  - O로 지정된 화소는 노드 $S$와 큰 값, 노드 $T$와 0
  - B로 지정된 화소는 노드 $S$와 0, 노드 $T$와 큰 값
  - 나머지 화소 $p$는

에지 $(p,S)$의 가중치 $w_{pS} = \lambda \cdot (-\ln prob(f(v_p) \mid \text{'background'}))$ (5.28)

에지 $(p,T)$의 가중치 $w_{pT} = \lambda \cdot (-\ln prob(f(v_p) \mid \text{'object'}))$ (5.29)

# 4.5.2 그랩컷 알고리즘으로 영상 분할

- 사용자 입력이 최소화 된 전경 추출 알고리즘
  - Input: foreground, background, and unknown part of the image. Unknown part can be either foreground or background.
    - Select rectangle around the object of interest.
      - region of inside : unknown
      - outside of rectangle: known background.
    - initial image segmentation:
      - inside rectangle=>foreground, outside rectangle=> background
    - compute similarity between pixels(에지 가중치, 픽셀 p,q의 가중치).
      - d는 거리, f()는 픽셀 색상값

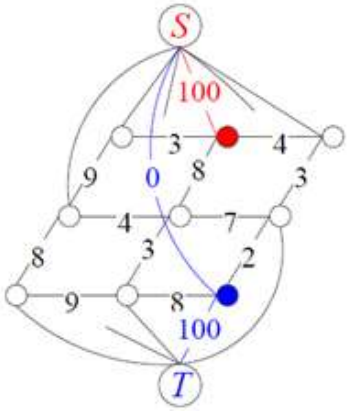
$$w_{pq} = e^{-\frac{(f(v_p)-f(v_q))^2}{2\sigma^2}} d(v_p, v_q) \qquad (5.27)$$

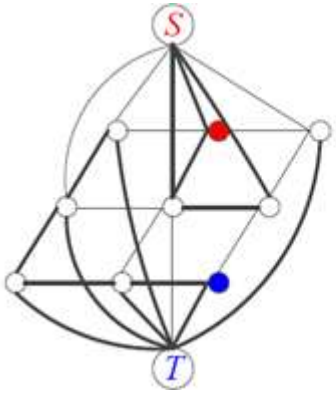
      - 배경(T) 혹은 전경(S)과 현재 픽셀과 에지 연결강도

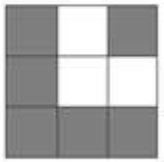
에지 $(p,S)$의 가중치 $w_{pS} = \lambda \cdot (-\ln prob(f(v_p) \mid \text{'background'}))$ (5.28)

에지 $(p,T)$의 가중치 $w_{pT} = \lambda \cdot (-\ln prob(f(v_p) \mid \text{'object'}))$ (5.29)

# 4.5.2 그랩컷 알고리즘으로 영상 분할

Graph cut

GrabCut

# 4.5.2 GrabCut

프로그램 4-7 GrabCut을 이용해 물체 분할하기

```
01  import cv2 as cv
02  import numpy as np
03
04  img=cv.imread('soccer.jpg')                    # 영상 읽기
05  img_show=np.copy(img)                          # 붓칠을 디스플레이할 목적의 영상
06
07  mask=np.zeros((img.shape[0],img.shape[1]),np.uint8)
08  mask[:,:]=cv.GC_PR_BGD                         # 모든 화소를 배경일 것 같음으로 초기화
09
10  BrushSiz=9                                     # 붓의 크기
11  LColor,RColor=(255,0,0),(0,0,255)              # 파란색(물체)과 빨간색(배경)
12
13  def painting(event,x,y,flags,param):
14      if event==cv.EVENT_LBUTTONDOWN:
15          cv.circle(img_show,(x,y),BrushSiz,LColor,-1) # 왼쪽 버튼 클릭하면 파란색
16          cv.circle(mask,(x,y),BrushSiz,cv.GC_FGD,-1)
17      elif event==cv.EVENT_RBUTTONDOWN:
18          cv.circle(img_show,(x,y),BrushSiz,RColor,-1) # 오른쪽 버튼 클릭하면 빨간색
19          cv.circle(mask,(x,y),BrushSiz,cv.GC_BGD,-1)
20      elif event==cv.EVENT_MOUSEMOVE and flags==cv.EVENT_FLAG_LBUTTON:
21          cv.circle(img_show,(x,y),BrushSiz,LColor,-1)
                                                   # 왼쪽 버튼 클릭하고 이동하면 파란색
22          cv.circle(mask,(x,y),BrushSiz,cv.GC_FGD,-1)
23      elif event==cv.EVENT_MOUSEMOVE and flags==cv.EVENT_FLAG_RBUTTON:
24          cv.circle(img_show,(x,y),BrushSiz,RColor,-1)
                                                   # 오른쪽 버튼 클릭하고 이동하면 빨간색
25          cv.circle(mask,(x,y),BrushSiz,cv.GC_BGD,-1)
26
27      cv.imshow('Painting',img_show)
```

cv.GC_FGD: 확실히 물체
cv.GC_BGD: 확실히 배경
cv.GC_PR_FGD: 물체일 것 같음
cv.GC_PR_BGD: 배경일 것 같음

# 4.5.2 GrabCut

```
28
29  cv.namedWindow('Painting')
30  cv.setMouseCallback('Painting',painting)
31
32  while(True):                                # 붓칠을 끝내려면 'q' 키를 누름
33      if cv.waitKey(1)==ord('q'):
34          break
35
36  # 여기부터 GrabCut 적용하는 코드
37  background=np.zeros((1,65),np.float64)   # 배경 히스토그램 0으로 초기화
38  foreground=np.zeros((1,65),np.float64)   # 물체 히스토그램 0으로 초기화
39
40  cv.grabCut(img,mask,None,background,foreground,5,cv.GC_INIT_WITH_MASK)
41  mask2=np.where((mask==cv.GC_BGD)|(mask==cv.GC_PR_BGD),0,1).astype('uint8')
42  grab=img*mask2[:,:,np.newaxis]
43  cv.imshow('Grab cut image',grab)
44
45  cv.waitKey()
46  cv.destroyAllWindows()
```

# 4.5.2 GrabCut


잘 분할한 곳　　분할 오류

np.where(condition, [x,y, ])

condition을 만족하면 x를, 만족하지않으면 y를 반환한다.

- grabCut(InputArray **img**, InputOutputArray **mask**, Rect **rect**, InputOutputArray **bgdModel**, InputOutputArray **fgdModel**, int **iterCount**, int **mode**=GC_EVAL )
  - **img**: input 8-bit 3channel image
  - **mask**: input/output 8-bit single channel mask. The mask is initialized by the function when mode is set to GC_INIT_WITH_RECT. Its elements may have one of following values:
    - **GC_BGD** defines an obvious background pixels. (0)
    - **GC_FGD** defines an obvious foreground (object) pixel. (1)
    - **GC_PR_BGD** defines a possible background pixel. (2)
    - **GC_PR_FGD** defines a possible foreground pixel. (3)

**rect** ROI containing a segmented object. The pixels outside of the ROI are marked as "obvious background". The parameter is only used when mode==**GC_INIT_WITH_RECT** .

**bgdModel** Temporary array for the background model. Do not modify it while you are processing the same image.

**fgdModel** Temporary arrays for the foreground model. Do not modify it while you are processing the same image.

**iterCount** Number of iterations the algorithm should make before returning the result. Note that the result can be refined with further calls with mode==**GC_INIT_WITH_MASK** or mode==GC_EVAL .

**mode** Operation mode that could be one of the **GrabCutModes**

**Python:**

cv.grabCut( img, mask, rect, bgdModel, fgdModel, iterCount[, mode] ) -> mask, bgdModel, fgdModel