Rozpoznawanie odcisków palców - szkieletyzacja i detekcja minucji

Autorzy: Gabriela Majstrak, Jan Opala

May 2025

Spis treści

1 Wstęp **4**

2 Implementacja **4**
2.1 Aspekty techniczne 4
2.2 Wczytywanie i przetwarzanie wstępne 4
2.2.1 Funkcje podstawowe 4

3 Operacje morfologiczne **5**
3.1 Implementacja podstawowych operacji 5
3.1.1 Erozja 5
3.1.2 Dylatacja 5
3.1.3 Otwarcie i zamknięcie 6

4 Binaryzacja **6**
4.1 Metoda Otsu 6
4.2 Binaryzacja z parametrem delta 8
4.3 Binaryzacja adaptacyjna 8

5 Szkieletyzacja morfologiczna **8**
5.1 Klasyczna szkieletyzacja morfologiczna 8

6 Algorytm K3M **9**
6.1 Implementacja algorytmu K3M 9
6.2 Fazy algorytmu K3M 10
6.3 Warunki K3M 10
6.4 Liczenie przejść 11
6.5 Warunki fazowe 11

7 Detekcja minucji - szczegółowe wyjaśnienia **12**
7.1 Wprowadzenie do minucji 12
7.2 Klasa MinutiaeDetector 12
7.3 Detekcja granic odcisku 13
7.4 Wykrywanie zakończeń i bifurkacji 13
7.5 Crossing Number - kluczowy algorytm klasyfikacji 15

8 Wykrywanie rdzeni i delt **16**
8.1 Pole orientacji 16
8.2 Indeks Poincaré 17

9 Filtrowanie i kontrola jakości **18**
9.1 Kontrola jakości szkieletu 18
9.2 Usuwanie fałszywych minucji 19

10 Wizualizacja i analiza **20**
10.1 Główna funkcja analizy 20

11 Ewaluacja i wnioski **22**
11.1 Porównanie algorytmów szkieletyzacji 22
11.2 Ograniczenia w wykrywaniu niektórych minucji 24
11.3 Wpływ jakości skanów 25
11.4 Obserwacje dotyczące powtarzalności 26
11.5 Mechanizmy filtrowania 27
11.6 Wnioski końcowe 28

1 Wstęp

W tym projekcie opracowana została metoda rozpoznawania odcisków palców w oparciu o szkieletyzację obrazu oraz detekcję charakterystycznych punktów (minucji). Implementacja obejmuje dwa główne algorytmy szkieletyzacji: morfologiczną szkieletyzację oraz algorytm K3M, a następnie zaawansowaną detekcję różnych typów minucji.

2 Implementacja

2.1 Aspekty techniczne

Projekt został napisany w języku `Python` w notatniku Jupyter notebook (rozszerzenie `.ipynb`). Wykorzystane zostały następujące biblioteki:

- `NumPy` – w celu operacji na pikselach jako macierzach
- `PIL` – biblioteka wymagana do otwierania plików będącymi zdjęciami
- `Matplotlib` – w celu wyświetlania obrazów
- `cv2` - biblioteka OpenCV służąca do zaawansowanych operacji na obrazach

2.2 Wczytywanie i przetwarzanie wstępne

2.2.1 Funkcje podstawowe

Funkcja zamiany obrazu na tablicę NumPy:

```
def load_image(image_path):
    image = np.array(Image.open(image_path))
    return image
```

Funkcja wyświetlania obrazu:

```
def display_image(image_array):
    if image_array.ndim == 2:
        plt.imshow(image_array, cmap='gray')
    else:
        plt.imshow(image_array)
    plt.axis('off')
    plt.show()
```

Konwersja do skali szarości:

```
def to_grey_scale(image_array):
    if image_array.ndim == 3 and image_array.shape[2] >= 3:
        r, g, b = image_array[..., 0], image_array[..., 1],
            image_array[..., 2]
        greyscale = 0.299 * r + 0.587 * g + 0.114 * b
        return greyscale.astype(np.uint8)
    else:
        return image_array
```

3 Operacje morfologiczne

3.1 Implementacja podstawowych operacji

3.1.1 Erozja

Erozja jako niekstensywna transformacja, która "ściera" macierz o jądro:

$$I \ominus K(i,j) = \begin{cases} 1, & \text{jeśli } \forall (m,n) \in K,\ I(i+m, j+n) = 1 \\ 0, & \text{w przeciwnym razie} \end{cases}$$

gdzie I to obraz, a K to jądro została zaimplementowana:

```
def erosion(image, kernel):
    eroded_image = np.zeros_like(image)
    h, w = image.shape
    kh, kw = kernel.shape
    pad_h, pad_w = kh // 2, kw // 2

    for i in range(pad_h, h - pad_h):
        for j in range(pad_w, w - pad_w):
            region = image[i - pad_h:i + pad_h + 1, j -
                pad_w:j + pad_w + 1]
            if np.all(region[kernel == 1] == 0):
                eroded_image[i, j] = 0
            else:
                eroded_image[i, j] = 1
    return eroded_image
```

3.1.2 Dylatacja

Dylatacja jako operacja rozszerzenia, która służy do "wypełniania dziur i zatok", która może zostać scharakteryzowane wzorem:

$$I \oplus K(i,j) = \begin{cases} 1, & \text{jeśli } \exists (m,n) \in K:\ I(i+m, j+n) = 1 \\ 0, & \text{w przeciwnym razie} \end{cases}$$

analogicznie do erozji, I to obraz, a K to jądro została zaimplementowana:

```
def dilation(image, kernel):
    dilated_image = np.zeros_like(image)
    h, w = image.shape
    kh, kw = kernel.shape
    pad_h, pad_w = kh // 2, kw // 2
    for i in range(pad_h, h - pad_h):
        for j in range(pad_w, w - pad_w):
            region = image[i - pad_h:i + pad_h + 1, j -
                pad_w:j + pad_w + 1]
            if np.any(region[kernel == 1] == 0):
                dilated_image[i, j] = 0
            else:
                dilated_image[i, j] = 1
    return dilated_image
```

3.1.3 Otwarcie i zamknięcie

Pozostałe operacje morfologiczne zostały zaimplementowane jako kombinacje pozostałych:

Otwarcie (erozja + dylatacja):

```
def opening(image, kernel):
    eroded = erosion(image, kernel)
    return dilation(eroded, kernel)
```

Zamknięcie (dylatacja + erozja):

```
def closing(image, kernel):
    dilated = dilation(image, kernel)
    return erosion(dilated, kernel)
```

4 Binaryzacja

4.1 Metoda Otsu

W celu wyznaczenia optymalnego progu binaryzacji stosujemy algorytm Otsu:

1. Funkcja oblicza histogram intensywności pikseli dla całego obrazu w zakresie 0-255
2. Wyznaczana jest całkowita liczba pikseli oraz suma ważona wszystkich intensywności
3. Algorytm iteruje przez wszystkie możliwe wartości progowe (0-255)

4. Dla każdego progu obliczane są:
    - wagi klas tła i pierwszego planu (liczba pikseli w każdej klasie)
    - średnie intensywności dla obu klas
    - wariancja międzyklasowa jako iloczyn wag klas i kwadratu różnicy średnich
5. Jako optymalny próg wybierana jest wartość maksymalizująca wariancję międzyklasową
6. Funkcja zwraca wartość progową zapewniającą najlepsze rozdzielenie obiektów od tła

Innymi słowy algorytm Otsu automatycznie znajduje próg, który minimalizuje wariancję wewnątrzklasową (lub równoważnie maksymalizuje wariancję międzyklasową), co prowadzi do optymalnej separacji dwóch głównych grup pikseli w obrazie. Zosatało to zaimplementowane:

```
def otsu_threshold(image_array):
    hist, bins = np.histogram(image_array.flatten(), bins
        =256, range=(0, 256))
    total = image_array.size
    current_max, threshold = 0, 0
    sum_total, sum_foreground = 0, 0
    weight_background, weight_foreground = 0, 0

    for i in range(256):
        sum_total += i * hist[i]

    for i in range(256):
        weight_background += hist[i]
        if weight_background == 0:
            continue
        weight_foreground = total - weight_background
        if weight_foreground == 0:
            break
        sum_foreground += i * hist[i]
        mean_background = sum_foreground / weight_background
        mean_foreground = (sum_total - sum_foreground) /
            weight_foreground
        between_class_variance = weight_background *
            weight_foreground * (mean_background -
            mean_foreground) ** 2
        if between_class_variance > current_max:
            current_max = between_class_variance
            threshold = i
    return threshold
```

4.2 Binaryzacja z parametrem delta

W celu przeprowadzenia binaryzacji obrazu stosujemy następujący algorytm:

1. Funkcja wykonuje wstępne przetwarzanie obrazu wejściowego
2. Wyznaczany jest optymalny próg binaryzacji za pomocą metody Otsu
3. Obliczany jest skorygowany próg poprzez przemnożenie wartości Otsu przez współczynnik delta (domyślnie 1.1) z ograniczeniem do maksymalnej wartości 255
4. Wykonywana jest binaryzacja obrazu: piksele o intensywności większej lub równej skorygowanemu progowi otrzymują wartość 255 (biały), pozostałe otrzymują wartość 0 (czarny)
5. Funkcja zwraca obraz binarny jako tablicę 8-bitowych liczb całkowitych bez znaku

Implementacja:

```
def binarize(image_array, delta=1.1):
    pre = preprocess(image_array)
    otsu_thresh = otsu_threshold(pre)
    adjusted_thresh = min(255, otsu_thresh * delta)
    binary = np.where(pre >= adjusted_thresh, 255, 0).astype
        (np.uint8)
    return binary
```

4.3 Binaryzacja adaptacyjna

Binaryzacja adaptacyjna z wykorzystaniem OpenCV:

```
def binarize_adaptive(image_array):
    return cv2.adaptiveThreshold(
        image_array,
        255,
        cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
        cv2.THRESH_BINARY,
        blockSize=11,
        C=2
    )
```

5 Szkieletyzacja morfologiczna

5.1 Klasyczna szkieletyzacja morfologiczna

Implementacja algorytmu szkieletyzacji opartego na operacjach erozji i otwarcia:

```
class MorphologicalSkeleton:
    def __init__(self):
        self.name = "Morfologiczna szkieletyzacja (klasyczna
            )"
        self.structuring_element = np.array([[0, 1, 0],
                                             [1, 1, 1],
                                             [0, 1, 0]],
                                                dtype=np.
                                                uint8)

    def thin(self, binary_image):
        img = binary_image.copy().astype(bool)
        skeleton = np.zeros_like(img, dtype=bool)
        eroded = img.copy()

        while True:
            opened = binary_opening(eroded, structure=self.
                structuring_element)
            temp = eroded & ~opened
            skeleton |= temp
            eroded = binary_erosion(eroded, structure=self.
                structuring_element)

            if not eroded.any():
                break

        return skeleton.astype(np.uint8)
```

Algorytm działa w następujący sposób:

1. Wykonuje iteracyjną erozję obrazu
2. Dla każdej iteracji oblicza otwarcie obrazu po erozji
3. Różnica między obrazem po erozji a jego otwarciem stanowi warstwę szkieletu
4. Proces kontynuuje się aż do całkowitego zerodowania obrazu

6 Algorytm K3M

6.1 Implementacja algorytmu K3M

Algorytm K3M to zaawansowana metoda szkieletyzacji, która wykorzystuje czterofazowy proces iteracyjnego usuwania pikseli brzegowych:

```
class K3MThinning:
def init(self):
self.name = "K3M"
```

```
def thin(self, binary_image):
    """Główna funkcja algorytmu K3M"""
    img = binary_image.copy().astype(np.uint8)

    if np.max(img) > 1:
        img = (img > 128).astype(np.uint8)

    changed = True
    iteration = 0

    while changed and iteration < 100:
        changed = False
        for phase in range(4):
            phase_changed = self._k3m_phase(img, phase)
            if phase_changed:
                changed = True
        iteration += 1

    return img
```

6.2 Fazy algorytmu K3M

Algorytm K3M składa się z czterech faz, każda skupiająca się na usuwaniu pikseli z różnych kierunków. Każda faza wykonuje pełne przeskanowanie obrazu i usuwa odpowiednie piksele:

```
def _k3m_phase(self, img, phase):

changed = False
to_remove = []
rows, cols = img.shape
for i in range(1, rows - 1):
    for j in range(1, cols - 1):
        if img[i, j] == 1:
            if self._k3m_condition(img, i, j, phase):
                to_remove.append((i, j))
                changed = True

for i, j in to_remove:
    img[i, j] = 0

return changed
```

6.3 Warunki K3M

Algorytm K3M sprawdza trzy główne warunki przed usunięciem piksela:

1. **Liczba sąsiadów** - musi mieścić się w przedziale [2, 6]

2. **Liczba przejść 0→1** - w 8-sąsiedztwie musi wynosić dokładnie 1

3. **Warunki fazowe** - specyficzne dla każdej z czterech faz

```
def _k3m_condition(self, img, i, j, phase):
"""Sprawdza warunki K3M dla danego piksela i fazy"""
region = img[i-1:i+2, j-1:j+2]

neighbors = np.sum(region) - 1
if not (2 <= neighbors <= 6):
    return False

transitions = self._count_transitions_k3m(region)
if transitions != 1:
    return False

return self._phase_specific_condition(region, phase)
```

6.4 Liczenie przejść

Funkcja licząca przejścia 0→1 w 8-sąsiedztwie piksela:

```
def _count_transitions_k3m(self, region):
"""Liczy przejŻcia 0 1   wok  Ć piksela"""
neighbors = [
region[0, 1], region[0, 2], region[1, 2], region[2, 2],
region[2, 1], region[2, 0], region[1, 0], region[0, 0]
]
neighbors.append(neighbors[0])
transitions = 0
for k in range(8):
    if neighbors[k] == 0 and neighbors[k + 1] == 1:
        transitions += 1

return transitions
```

6.5 Warunki fazowe

Każda z czterech faz ma swoje specyficzne warunki oparte na wartościach sąsiadujących pikseli:

```
def _phase_specific_condition(self, region, phase):
"""Warunki fazowe algorytmu K3M"""
p2 = region[0, 1]
p4 = region[1, 2]
p6 = region[2, 1]
p8 = region[1, 0]
if phase == 0:
```

```
    return (p2 * p4 * p8) == 0 and (p2 * p6 * p8) == 0
elif phase == 1:
    return (p2 * p4 * p6) == 0 and (p4 * p6 * p8) == 0
elif phase == 2:
    return (p2 * p4 * p8) == 0 and (p2 * p6 * p8) == 0
else:  # phase == 3
    return (p2 * p4 * p6) == 0 and (p4 * p6 * p8) == 0
```

7 Detekcja minucji - szczegółowe wyjaśnienia

7.1 Wprowadzenie do minucji

Minucje to charakterystyczne punkty w odcisku palca, gdzie linie papilarne ulegają przerwaniu lub rozgałęzieniu. Wyróżnia się dwa podstawowe typy:

- **Zakończenia (endpoints)** - punkty gdzie linia papilarna kończy się
- **Bifurkacje (bifurcations)** - punkty gdzie jedna linia rozdziela się na dwie

7.2 Klasa MinutiaeDetector

Główna klasa odpowiedzialna za wykrywanie minucji z konfigurowalnymi parametrami filtrowania:

```
class MinutiaeDetector:
    def __init__(self, border_margin=15,
        min_distance_from_edge=20):
        # 8-s siedztwo w kolejno Żci zgodnej z ruchem
            wskaz wek zegara
        # Rozpoczyna od lewego g rnego rogu (-1,-1)
        self.neighbors_8 = [(-1,-1), (-1,0), (-1,1), (0,1),
                            (1,1), (1,0), (1,-1), (0,-1)]
        self.border_margin = border_margin  # Margines od
            kraw Ździ obrazu
        self.min_distance_from_edge = min_distance_from_edge
              # Min. odleg Co Ż  od granic odcisku
```

Parametry inicjalizacji:

- `neighbors_8`: Tablica definiująca 8-sąsiedztwo piksela. Kolejność jest kluczowa dla crossing number.
- `border_margin`: Eliminuje artefakty powstające na brzegach obrazu podczas przetwarzania.
- `min_distance_from_edge`: Odrzuca minucje zbyt blisko granic rzeczywistego odcisku palca.

7.3 Detekcja granic odcisku

Funkcja wykrywająca rzeczywiste granice odcisku na podstawie analizy wariancji lokalnej:

```
def detect_fingerprint_boundary(self, image, threshold_ratio
    =0.1):
    # Krok 1: Wyg Ćadzenie gaussowskie - redukcja szumu z
        zachowaniem struktur
    blurred = cv2.GaussianBlur(image.astype(np.float32),
        (15, 15), 0)

    # Krok 2: Obliczanie  Żredniej  lokalnej w oknach 16x16
    kernel = np.ones((16, 16), np.float32) / 256  #
        Znormalizowany kernel
    mean_img = cv2.filter2D(blurred, -1, kernel)

    # Krok 3: Obliczanie wariancji lokalnej u ywaj c wzoru
         Var(X) = E[X ] - (E[X])
    sqr_img = blurred * blurred  # Kwadrat ka dego piksela
    sqr_mean = cv2.filter2D(sqr_img, -1, kernel)  # E[X ]
    variance = sqr_mean - mean_img * mean_img  # Var(X)

    # Krok 4: Progowanie adaptacyjne
    threshold = np.mean(variance) * threshold_ratio
    mask = (variance > threshold).astype(np.uint8)

    # Krok 5: Operacje morfologiczne dla oczyszczenia maski
    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE,
        (15, 15))
    mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel)
        #   Ączy   bliskie regiony
    mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)
        # Usuwa ma Će artefakty

    return mask
```

Zasada działania:

1. Obszary z liniami papilarnymi mają wysoką wariancję lokalną (tekstura)
2. Obszary tła są gładkie, więc mają niską wariancję
3. Próg adaptacyjny dostosowuje się do ogólnej jakości obrazu
4. Operacje morfologiczne wygładzają i oczyszczają wynikową maskę

7.4 Wykrywanie zakończeń i bifurkacji

Główna funkcja klasyfikująca piksele szkieletu jako potencjalne minucje:

```
def detect_endpoints_and_bifurcations(self, skeleton,
    original_image=None):
    endpoints = []
    bifurcations = []

    # Opcjonalne wykrywanie granic odcisku dla filtrowania
        brzegowego
    boundary_mask = None
    if original_image is not None:
        boundary_mask = self.detect_fingerprint_boundary(
            original_image)
    else:
        boundary_mask = np.ones_like(skeleton)

    # Pobranie wszystkich punkt w szkieletu (niezerowe
        piksele)
    skeleton_points = np.where(skeleton > 0)

    for i in range(len(skeleton_points[0])):
        x, y = skeleton_points[0][i], skeleton_points[1][i]

        # Filtr 1: Sprawdzenie odleg Ćo Żci od granic
            odcisku
        if self.is_near_edge(x, y, boundary_mask, self.
            min_distance_from_edge):
            continue

        # Filtr 2: Kontrola jako Żci lokalnej struktury
            szkieletu
        if not self.check_skeleton_quality(skeleton, x, y):
            continue

        # G Ć wna klasyfikacja na podstawie liczby
            s siad w
        neighbors_count = self.get_neighbors_count(skeleton,
            x, y)

        if neighbors_count == 1:
            # Punkt z jednym s siadem = zako czenie linii
            endpoints.append((x, y))
        elif neighbors_count >= 3:
            # Punkt z  3  s siadami mo e by  bifurkacj
            # Dodatkowa weryfikacja przez crossing number
            crossing = self.get_crossing_number(skeleton, x,
                y)
            if crossing >= 3:
                bifurcations.append((x, y))

    return endpoints, bifurcations
```

Algorytm klasyfikacji minucji:

1. **Filtrowanie przestrzenne**: Eliminacja punktów zbyt blisko brzegów
2. **Kontrola jakości**: Sprawdzenie czy lokalny szkielet nie zawiera artefaktów
3. **Liczenie sąsiadów**: Podstawowa klasyfikacja na podstawie stopnia węzła
4. **Crossing number**: Dodatkowa weryfikacja dla bifurkacji

7.5 Crossing Number - kluczowy algorytm klasyfikacji

Metoda obliczająca liczbę przejść z 0 na 1 w 8-sąsiedztwie:

```
def get_crossing_number(self, image, x, y):
    h, w = image.shape
    values = []

    # Pobranie wartoŻci wszystkich 8 s siad w w ustalonej
        kolejnoŻci
    for dx, dy in self.neighbors_8:
        nx, ny = x + dx, y + dy
        if 0 <= nx < h and 0 <= ny < w:
            values.append(1 if image[nx, ny] > 0 else 0)
        else:
            values.append(0)  # Piksele poza granicami = 0

    # Liczenie przej Ż  z 0 na 1 w cyklu
    crossing = 0
    for i in range(8):
        # Sprawdzenie przejŻcia z values[i] na values[(i+1)
            % 8]
        if values[i] == 0 and values[(i+1) % 8] == 1:
            crossing += 1

    return crossing
```

Interpretacja crossing number:

- **CN = 1**: Zakończenie linii (endpoint)
- **CN = 2**: Punkt normalny na linii (nie jest minucją)
- **CN = 3**: Bifurkacja (punkt rozgałęzienia)
- **CN 4**: Skomplikowane przecięcie lub artefakt

8 Wykrywanie rdzeni i delt

8.1 Pole orientacji

Obliczanie lokalnej orientacji linii papilarnych metodą gradientową:

```
def compute_orientation_field(self, image, block_size=16,
    smooth_sigma=1.0):
    # Wyg Ćadzenie pocz tkowe
    image_smooth = cv2.GaussianBlur(image.astype(np.float32)
        , (5, 5), smooth_sigma)
    h, w = image_smooth.shape
    orientations = np.zeros((h // block_size, w //
        block_size))

    # Obliczanie gradient w w kierunkach x i y
    sobelx = cv2.Sobel(image_smooth, cv2.CV_32F, 1, 0, ksize
        =3)
    sobely = cv2.Sobel(image_smooth, cv2.CV_32F, 0, 1, ksize
        =3)

    # Przetwarzanie w blokach
    for i in range(0, h - block_size, block_size):
        for j in range(0, w - block_size, block_size):
            gx_block = sobelx[i:i+block_size, j:j+block_size
                ]
            gy_block = sobely[i:i+block_size, j:j+block_size
                ]

            # Sk Ćadowe tensora struktury
            gxx = np.sum(gx_block * gx_block)      #   ( ĆI
                / Ćx )
            gyy = np.sum(gy_block * gy_block)      #   ( ĆI
                / Ćy )
            gxy = np.sum(gx_block * gy_block)      #   ( ĆI
                / Ćx )( ĆI / Ćy )

            # Sprawdzenie czy gradient jest wystarczaj co
                silny
            gradient_magnitude = gxx + gyy
            if gradient_magnitude > 100:
                # Orientacja jako po Ćowa k ta g Ć wnej
                    osi tensora
                orientation = 0.5 * np.arctan2(2 * gxy, gxx
                    - gyy)
                orientations[i // block_size, j //
                    block_size] = orientation

    return self.smooth_orientation_field(orientations)
```

Matematyczne podstawy:

- Tensor struktury: $J = \begin{bmatrix} G_x^2 & G_x G_y \\ G_x G_y & G_y^2 \end{bmatrix}$
- Orientacja: $\theta = \frac{1}{2} \arctan\left(\frac{2\sum G_x G_y}{\sum G_x^2 - \sum G_y^2}\right)$
- Dzielenie przez 2 wynika z symetrii linii papilarnych (+)

8.2 Indeks Poincaré

Wykrywanie punktów osobliwych (rdzeni i delt) na podstawie cyrkulacji pola orientacji:

```
def detect_cores_and_deltas(self, image, orientations,
    block_size=16):
    cores = []
    deltas = []
    h, w = orientations.shape

    # Przeszukiwanie ka dego punktu wewn Ątrznego pola
        orientacji
    for i in range(1, h-1):
        for j in range(1, w-1):
            poincare_index = 0

            # Definicja 8 punkt w wok  Ć centralnego
                punktu (i,j)
            points = [(i-1,j-1), (i-1,j), (i-1,j+1), (i,j+1)
                ,
                      (i+1,j+1), (i+1,j), (i+1,j-1), (i,j-1)]

            # Obliczanie cyrkulacji wok  Ć zamkni Ątej
                p Ątli
            for k in range(8):
                p1 = points[k]
                p2 = points[(k+1) % 8]

                angle1 = orientations[p1]
                angle2 = orientations[p2]

                # Podwojenie k t w dla w Ća Żciwej
                    cyrkulacji (       2  )
                double_angle1 = 2 * angle1
                double_angle2 = 2 * angle2
                angle_diff = double_angle2 - double_angle1

                # Normalizacja r  nicy k t w do
                    przedzia Ću [-  ,  ]
                while angle_diff > np.pi:
                    angle_diff -= 2 * np.pi
```

```
                while angle_diff < -np.pi:
                    angle_diff += 2 * np.pi

                poincare_index += angle_diff

            # Normalizacja indeksu przez 2
            poincare_index = poincare_index / (2 * np.pi)

            # Klasyfikacja punkt w osobliwych
            if abs(poincare_index - 0.5) < 0.3:  # Rdze  (
                core)
                real_x = i * block_size + block_size // 2
                real_y = j * block_size + block_size // 2
                cores.append((real_x, real_y))
            elif abs(poincare_index + 0.5) < 0.3:  # Delta
                real_x = i * block_size + block_size // 2
                real_y = j * block_size + block_size // 2
                deltas.append((real_x, real_y))

    return cores, deltas
```

Interpretacja indeksu Poincaré:

- **PI +0.5**: Rdzeń (core) - linie tworzą zamkniętą pętlę
- **PI -0.5**: Delta - linie rozchodzą się na trzy strony
- **PI 0**: Punkt regularny - brak osobliwości
- Podwojenie kątów (2) jest konieczne ze względu na symetrię linii

9 Filtrowanie i kontrola jakości

9.1 Kontrola jakości szkieletu

Funkcja eliminująca artefakty i zapewniająca wiarygodność wykrytych minucji:

```
def check_skeleton_quality(self, skeleton, x, y, window_size
    =11):
    h, w = skeleton.shape
    half_win = window_size // 2

    # Sprawdzenie czy okno mie Żci si Ź w granicach obrazu
    if (x - half_win < 0 or x + half_win >= h or
        y - half_win < 0 or y + half_win >= w):
        return False

    # Wyodr Źbnienie lokalnego okna wok  Ć kandydata na
        minucj Ź
    window = skeleton[x-half_win:x+half_win+1, y-half_win:y+
        half_win+1]
```

```
    # Test 1: Sprawdzenie g Ćsto Żci pikseli szkieletu
    density = np.sum(window > 0) / (window_size *
        window_size)
    if density > 0.3:  # Zbyt g Ćsty szkielet mo e
        wskazywa  na artefakt
        return False

    # Test 2: Liczenie zako cze  w lokalnym oknie
    endpoint_count = 0
    for i in range(1, window_size-1):
        for j in range(1, window_size-1):
            if window[i, j] > 0:
                neighbors = self.get_neighbors_count(window,
                    i, j)
                if neighbors == 1:  # Zako czenie
                    endpoint_count += 1

    # Zbyt wiele zako cze  w ma Ćym obszarze =
        prawdopodobny artefakt
    if endpoint_count > 3:
        return False

    return True
```

Kryteria jakości:

1. **Gęstość szkieletu**: Maksymalnie 30% pikseli w oknie może należeć do szkieletu
2. **Liczba zakończeń**: Maksymalnie 3 zakończenia w oknie 11×11
3. **Lokalizacja**: Kandydat musi być wystarczająco daleko od brzegów

9.2 Usuwanie fałszywych minucji

Filtrowanie na podstawie odległości przestrzennej między minucjami:

```
def remove_spurious_minutiae(self, endpoints, bifurcations,
    min_distance=10):
    def distance(p1, p2):
        return np.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])
            **2)

    # Filtrowanie zako cze
    filtered_endpoints = []
    for i, ep1 in enumerate(endpoints):
        too_close = False
        for j, ep2 in enumerate(endpoints):
            if i != j and distance(ep1, ep2) < min_distance:
                too_close = True
```

```
                break
        if not too_close:
            filtered_endpoints.append(ep1)

    # Filtrowanie bifurkacji
    filtered_bifurcations = []
    for i, bf1 in enumerate(bifurcations):
        too_close = False
        for j, bf2 in enumerate(bifurcations):
            if i != j and distance(bf1, bf2) < min_distance:
                too_close = True
                break
        if not too_close:
            filtered_bifurcations.append(bf1)

    return filtered_endpoints, filtered_bifurcations
```

Uzasadnienie filtrowania odległościowego:

- Prawdziwe minucje rzadko występują bardzo blisko siebie
- Skupiska minucji często wynikają z artefaktów szkieletyzacji
- Minimalna odległość 10 pikseli jest kompromisem między czułością a specyficznością

10 Wizualizacja i analiza

10.1 Główna funkcja analizy

Kompletny pipeline przetwarzania i porównywania różnych metod szkieletyzacji:

```
def process_and_visualize_arrays(images, delta=0.35):
    # Inicjalizacja klas przetwarzania
    detector = MinutiaeDetector()
    ms = MorphologicalSkeleton()  # Szkieletyzacja
        morfologiczna
    k3m = K3MThinning()           # Algorytm K3M

    # Kernel strukturalny dla operacji morfologicznych
    kernel = np.array([
        [0, 0, 0],
        [1, 1, 1],  # Poziomy element strukturalny
        [0, 0, 0]
    ], dtype=np.uint8)

    num_images = len(images)
    fig, axes = plt.subplots(num_images, 5, figsize=(20, 4 *
         num_images))
```

```
    for row, image in enumerate(images):
        # Krok 1: Binaryzacja adaptacyjna
        bw = binarize(image, delta)

        # Krok 2: Operacja domknięcia morfologicznego
        clean = closing(bw, kernel)

        # Krok 3: Szkieletyzacja dwoma metodami
        skel_ms = ms.thin(clean.copy())     # Metoda
            morfologiczna
        skel_k3m = k3m.thin(clean.copy())  # Metoda K3M

        # Krok 4: Detekcja minucji dla obu szkieletów
        min_ms = detector.detect_minutiae(skel_ms, clean)
        min_k3m = detector.detect_minutiae(skel_k3m, clean)

        # Krok 5: Wizualizacja wyników w 5 kolumnach
        axes[row, 0].imshow(image, cmap='gray')
        axes[row, 0].set_title('Obraz oryginalny')

        axes[row, 1].imshow(skel_ms, cmap='gray')
        axes[row, 1].set_title('Szkielet morfologiczny')

        detector.visualize_minutiae_on_ax(axes[row, 2],
            skel_ms, min_ms)
        axes[row, 2].set_title('Minucje MS')

        axes[row, 3].imshow(skel_k3m, cmap='gray')
        axes[row, 3].set_title('Szkielet K3M')

        detector.visualize_minutiae_on_ax(axes[row, 4],
            skel_k3m, min_k3m)
        axes[row, 4].set_title('Minucje K3M')

    plt.tight_layout()
    plt.show()
```

Pipeline przetwarzania:

1. **Binaryzacja**: Konwersja odcisku do obrazu binarnego z progiem
2. **Oczyszczanie**: Operacja closing wypełnia małe przerwy w liniach
3. **Szkieletyzacja**: Porównanie dwóch algorytmów (morfologiczny vs K3M)
4. **Detekcja minucji**: Aplikacja jednolitego detektora do obu szkieletów
5. **Wizualizacja**: Systematyczne porównanie wyników obu metod

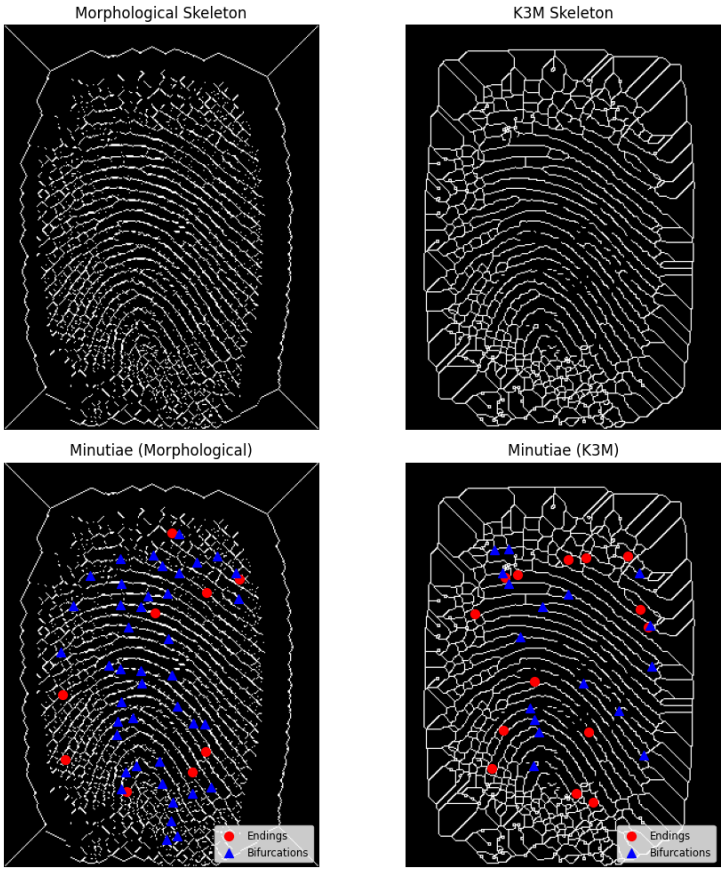
11 Ewaluacja i wnioski

11.1 Porównanie algorytmów szkieletyzacji

Algorytm K3M generalnie produkuje lepsze szkielety niż klasyczna szkieletyzacja morfologiczna, co przekłada się na dokładniejszą detekcję minucji. Przeprowadzone eksperymenty wykazały, że K3M radzi sobie znacznie lepiej niż szkieletyzacja morfologiczna w kontekście analizy odcisków palców.

Szkieletyzacja morfologiczna często pozostawia na obrazie różnego rodzaju artefakty strukturalne. Z racji tego, że przy wykrywaniu minucji analizujemy sąsiedztwo danego piksela, takie artefakty mogą znacznie fałszować wyniki i prowadzić do wykrywania minucji, które wcale nie są prawdziwymi minucjami. Problem ten jest szczególnie widoczny podczas analizy crossing number, gdzie dodatkowe elementy strukturalne wprowadzają błędne klasyfikacje punktów jako bifurkacje lub zakończenia.

W przeciwieństwie do tego, algorytm K3M produkuje znacznie czystsze szkielety, co bezpośrednio przekłada się na wyższą jakość detekcji minucji i mniejszą liczbę fałszywych pozytywów.

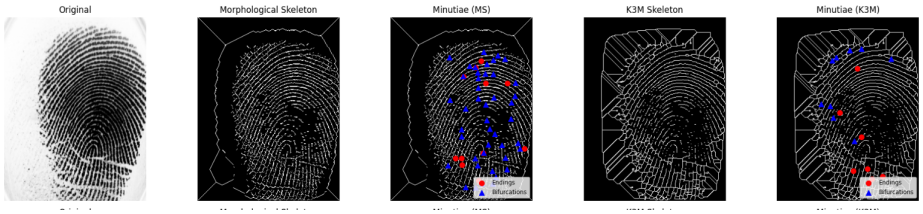


Rysunek 1: Porównanie szkiletyzacji morfologicznej i k3m

11.2 Ograniczenia w wykrywaniu niektórych minucji

Mimo licznych prób, nie udało się napisać skutecznego algorytmu pozwalającego na wykrywanie rdzeni i delt. Algorytm jest w stanie jedynie wykrywać zakończenia i bifurkacje, co stanowi znaczące ograniczenie funkcjonalności systemu. Implementacja metody indeksu Poincaré, choć teoretycznie poprawna, nie dawała zadowalających rezultatów praktycznych w przypadku analizowanych obrazów odcisków palców.

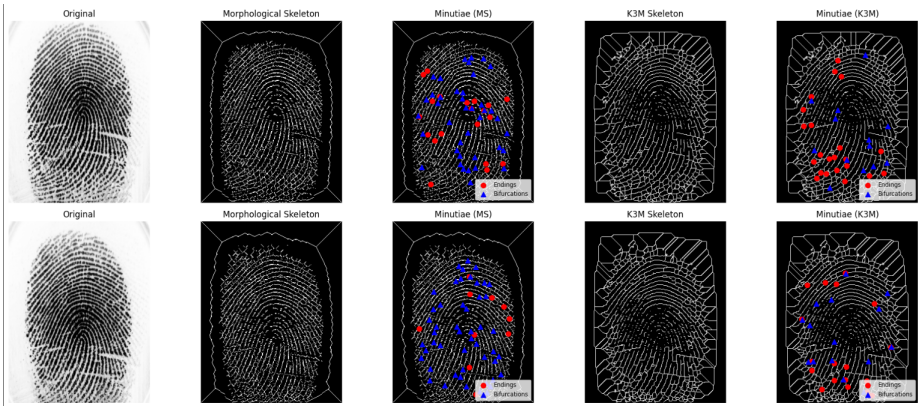
W związku z tym, że nie udało się otrzymać prawidłowej detekcji delt i rdzeni, niemożliwe było przeprowadzenie porównywania dwóch odcisków tego samego palca w oparciu o pełny zestaw cech charakterystycznych, co uniemożliwiło próbę rozpoznawania odcisku palca w sposób kompletny.



Rysunek 2: przykład z za ciemnym skanem

11.3 Wpływ jakości skanów

Przeszkodą w projekcie była nie zawsze zadowalająca jakość skanów odcisków palców. Problemy z rozdzielczością, kontrastem oraz obecność szumu znacząco wpływały na jakość szkieletyzacji i późniejszą detekcję minucji. Szczególnym problemem była za duża lub za mała jasność niektórych skanów.



Rysunek 3: Caption

11.4 Obserwacje dotyczące powtarzalności

Jednak w przypadku porównywania dwóch różnych skanów tego samego palca można było gołym okiem zauważyć, że część wykrytych minucji na obu skanach się pokrywa.

Natomiast patrząc na wyniki szkieletyzacji morfologicznej można było stwierdzić chaos - brak wyraźnej korelacji między pozycjami minucji w różnych skanach tego samego palca, co czyni tę metodę praktycznie niezdatną do zastosowań biometrycznych wymagających powtarzalności wyników.

11.5 Mechanizmy filtrowania

System implementuje zaawansowane mechanizmy filtrowania w celu eliminacji artefaktów i fałszywych minucji powstających na brzegach obrazu lub w obszarach o niskiej jakości. Zastosowane filtry obejmują:

- Kontrolę jakości szkieletu w lokalnym otoczeniu kandydata na minucję
- Filtrowanie brzegowe eliminujące minucje zbyt blisko granic odcisku
- Usuwanie skupisk minucji na podstawie minimalnej odległości przestrzennej
- Weryfikację topologiczną przy użyciu crossing number

11.6 Wnioski końcowe

Przeprowadzone badania jednoznacznie wskazują na przewagę algorytmu K3M nad tradycyjną szkieletyzacją morfologiczną w kontekście analizy odcisków palców. Główne zalety K3M to:

- Znacznie lepsza jakość produkowanych szkieletów
- Mniejsza liczba artefaktów strukturalnych
- Wyższa stabilność i powtarzalność wyników
- Lepsza przydatność dla praktycznych zastosowań biometrycznych

Pomimo ograniczeń związanych z niewykrywaniem punktów osobliwych (rdzeni i delt) oraz wrażliwości na jakość skanów, system wykazuje potencjał dla zastosowań wymagających analizy podstawowych typów minucji. Kluczowym czynnikiem sukcesu pozostaje odpowiedni dobór parametrów przetwarzania oraz zapewnienie wysokiej jakości obrazów wejściowych.

Projekt potwierdził fundamentalne znaczenie wyboru właściwego algorytmu szkieletyzacji - różnice w jakości szkieletów bezpośrednio przekładają się na użyteczność całego systemu analizy biometrycznej.