

Uniwersytet Jagielloński
Wydział Matematyki i Informatyki
Zespół Katedr i Zakładów Informatyki Matematycznej
kierunek: Informatyka Analityczna
studia stacjonarne

Nr albumu: 1074871

Zygmunt Łenyk

Rozpoznawanie i wykrywanie struktur grafowych na zdjęciach

Promotor pracy magisterskiej
dr Grzegorz Matecki

Opracowano zgodnie z Ustawą o prawie autorskim i prawach pokrewnych z dnia 4 lutego 1994 r. (Dz.U. 1994 nr 24 poz. 83) wraz z nowelizacją z dnia 25 lipca 2003 r. (Dz.U. 2003 nr 166 poz. 1610) oraz z dnia 1 kwietnia 2004 r. (Dz.U. 2004 nr 91 poz. 869)

Kraków 2017

Streszczenie

Praca zajmuje się problemem rozpoznawania odręcznie narysowanych grafów. Zaprezentowany w niej algorytm, dostając obraz grafu, zwraca jego reprezentację w postaci listy krawędzi. Konwencja rysowania wierzchołków i krawędzi jest narzucona, lecz w jej ramach algorytm wykazuje dużą elastyczność dla różnych stylów. Za pomocą technik widzenia komputerowego graf jest najpierw oddzielany od podłoża, a następnie wykrywane są kolejno wierzchołki i połączenia między nimi. Algorytm jest ewaluowany na zestawie grafów rysowanych odręcznie przez różne osoby.

Spis treści

Streszczenie	1
1 Wstęp	3
1.1 Plan pracy	4
1.2 Uwagi i ograniczenia pracy	5
2 Diagramy grafów	7
2.1 Normalizacja rysunków	8
2.1.1 Zmiana przestrzeni kolorów	9
2.1.2 Usuwanie szumów	10
2.1.3 Zamiany	11
3 Rozpoznawanie obiektów	12
3.1 Rozpoznawanie wierzchołków	12
3.1.1 Wykrywanie grubości linii	13
3.1.2 Detekcja skupisk	15
3.1.3 Konwolucja z okrągłym jądrem	16
3.1.4 Wykrywanie grubych krawędzi	18
3.1.5 Podsumowanie	23
3.2 Rozpoznawanie krawędzi	24
3.2.1 Grafy planarne	27
3.2.2 Grafy nieplanarne	28
4 Uwagi i implementacja	32
4.1 Rozpoznawanie wierzchołków	32
4.2 Rozpoznawanie krawędzi	33
5 Ewaluacja metody	36
5.1 Wyniki testów	36
5.1.1 Ewaluacja wykrywania wierzchołków	37
5.1.2 Ewaluacja wykrywania krawędzi	37
5.1.3 Liczba błędów na rysunku	38
5.2 Błędy algorytmu	39
5.2.1 Niewykryty wierzchołek	39
5.2.2 Błędnie połączone krawędzie	40
5.2.3 Wykryty nadmiarowy wierzchołek	41
5.2.4 Zbyt krótka krawędź	41
6 Podsumowanie	43

Rozdział 1

Wstęp

Rozpoznanie narysowanego grafu nie przysparza człowiekowi kłopotu. W większości przypadków jesteśmy w stanie bez problemu zaznaczyć poprawnie wierzchołki i powiedzieć, które z nich są połączone. Kwestią nieco bardziej trudną jest zamienienie jego reprezentacji z graficznej na przyjazną komputerowi – listę krawędzi lub tablicę sąsiedztwa. Wciąż jest to zadanie, które każdy zainteresowany matematyką powinien bez problemu wykonać, lecz już czas jego realizacji rośnie bardzo szybko wraz z rosnącą liczbą wierzchołków i krawędzi. Zdecydowanie jest to przyrost szybszy od liniowego. Kolejnym stopniem zaawansowania jest wykonanie algorytmu na narysowanym grafie lub określenie jego nietrywialnych cech (jak na przykład liczba chromatyczna). Jest to proces czasochłonny i bardzo często trudny do wykonania ręcznie. Z drugiej strony narysowanie i modyfikowanie grafu na kartce papieru lub tablicy jest najszybszym sposobem na przelanie pomysłu na konkretny graf i jego wizualizację. W rezultacie osoba badająca właściwości grafu ma dwie możliwości:

- zapisać graf w postaci przyjaznej dla komputera i następnie korzystać z programów implementujących algorytmy,
- narysować graf i analizować go ręcznie.

Oba przypadki są, na którymś ze swoich etapów, bardzo czasochłonne. Odpowiedzią na to jest proces zautomatyzowanego zamianiania reprezentacji grafu z graficznej na cyfrową. Zwłaszcza grafów narysowanych ręcznie. Można zatem podać bardzo konkretny problem, który stara się rozwiązać zaprezentowany w niej algorytm:

Wykrywanie struktur grafowych na zdjęciach

Wejście odręcznie narysowany graf

Wyjście cyfrowa reprezentacja grafu – format DOT^a

^a<http://www.graphviz.org/Documentation.php>

Taki algorytm ma więc szeroki zakres zastosowań, od naukowych po czysto dydaktyczne. Zadaniem użytkownika pozostaje jedynie dostarczenie pliku z obrazem. Jest to czynność wielokrotnie szybsza od wymienionych powyżej.

Temat ten został poruszony w [1]. Algorytm zaprezentowany przez autorów jej osiąga bardzo dobre wyniki rozpoznawania grafów ($\sim 93\%$), jednak był testo-

wany na grafach narysowanych komputerowo, w związku z czym jest nastawiony na jednakową wielkość i kształt wierzchołków oraz niezmienną grubość krawędzi. Prezentowana tutaj metoda powstawała niezależnie od [1]. Jest nastawiona bardziej na rysunku ręczne oraz wszelkie wynikające z tego niedokładności. Założenia poczynione dotyczące kształtu i wielkości wierzchołków oraz krawędzi są dużo luźniejsze.

Podkreślane słowo „odręczny” nie eliminuje oczywiście grafów wygenerowanych komputerowo – można na nie spojrzeć jak na szczególny przypadek idealnego rysowania ręcznego.

Cały proces bardzo zależy od instancji wejściowej – graf narysowany starannie i niepozostawiający wątpliwości co do odróżnienia wierzchołków od krawędzi będzie dużo łatwiejszy do przetworzenia od rysunku niestandardowego. Dlatego też ocena jakości musi być inna. Najczęściej podczas ewaluacji nowego algorytmu stawia się użytkownika jako jego przeciwnika i rozgrywa „grę”, w której jedna strona stara się za pomocą instancji wejściowej „oszukać” drugą. Takie podejście w przypadku analizy obrazów jest skazane na porażkę. Bez większego problemu można by takim sposobem wykazać zerową skuteczność algorytmów z tej dziedziny. Stosowaną metodą jest wzięcie losowych rysunków – w tym przypadku oznacza to narysowanych przez różne osoby.

Algorytm jest oceniany na kilka sposobów. Pierwszy z nich mierzy iloraz liczby grafów rozpoznanych poprawnie do liczby wszystkich grafów. Skuteczność, jaką udało się uzyskać w tej pracy, wynosi $\sim 65\%$ perfekcyjnie wygenerowanych grafów. W wielu przypadkach, chociaż wynik algorytmu nie jest idealnym odwzorowaniem instancji wejściowej, różni się od niej bardzo niewiele. Takie rezultaty też są cenne, ponieważ pozwalają użytkownikowi na wprowadzenie bardzo małych zmian na swoim rysunku, które sprawią, że zostanie on rozpoznany poprawnie. W związku z tym mierzona jest także skuteczność z dopuszczoną niewielką ilością błędów. Algorytm osiąga wyniki: 79% grafów wygenerowanych z maksymalnie jednym błędem oraz 86% z maksymalnie dwoma błędami.

1.1 Plan pracy

Na początku zostanie omówiona charakterystyka zdjęć grafów. Ich struktura jest z jednej strony prosta, ale z drugiej strony delikatna. Oznacza to, że podczas ich obrabiania należy zwrócić uwagę na fakt, że w niektórych przypadkach niewielka zmiana wyglądu obrazka może doprowadzić do wykrycia nadprogramowej lub braku wykrycia poprawnej krawędzi/wierzchołka.

Następnie omówiony będzie proces normalizacji (preprocessing) zdjęć tak, aby były łatwiejsze do dalszej analizy. Główne cele to sprowadzenie obrazka do postaci czarno-białej, usunięcie szumów tła oraz drobnych niedokładności podczas rysowania odręcznego, które mogą mieć duży wpływ na wykrycie obiektu. Do tego zostaną użyte operacje progowania adaptynego (adaptive thresholding), usuwanie obiektów o małym polu oraz operacje morfologiczne – dylatacja i erozja.

Tak przygotowany obraz będzie poddany wykrywaniu wierzchołków (rozdział 3.1) oraz krawędzi (rozdział 3.2). Analiza krawędzi opiera się na uprzednim,

poprawnym wykryciu wszystkich wierzchołków grafu oraz korzysta z niektórych metod użytych w tym procesie. Jest więc niezbędne, żeby te dwa kroki zostały wykonane w tej właśnie kolejności. Czytelnik, który będzie chciał ominąć rozdział o wierzchołkach, może mieć problemy z nieznajomością pojęć zdefiniowanych wcześniej.

Wykrywanie wierzchołków jest serią kilku metod, z których pierwsza – analiza skupisk (Laplacian of Gaussian [2]) – jest najbardziej ogólna i ma za zadanie wskazać szeroki zbiór kandydatów na wierzchołki. Kolejne metody korygują jej wynik poprzez odrzucanie przypadków, które bardzo często generują błędne wskazanie. Każda więc jest traktowana jako filtr, który przepuszcza coraz mniej wyników. Tymi filtrami są:

- analiza jednorodności skupiska – za pomocą specjalnego jądra konwolucji sprawdzane jest, czy kandydat na wierzchołek ma odpowiednią wielkość oraz jednorodność koloru,
- analiza cechy regionu obrazu – każdy piksel jest klasyfikowany, jako należący do jednej z trzech klas: krawędzi, tła oraz rogu obrazu. Zbyt duża liczba pikseli krawędziowych powoduje odrzucenie kandydata.

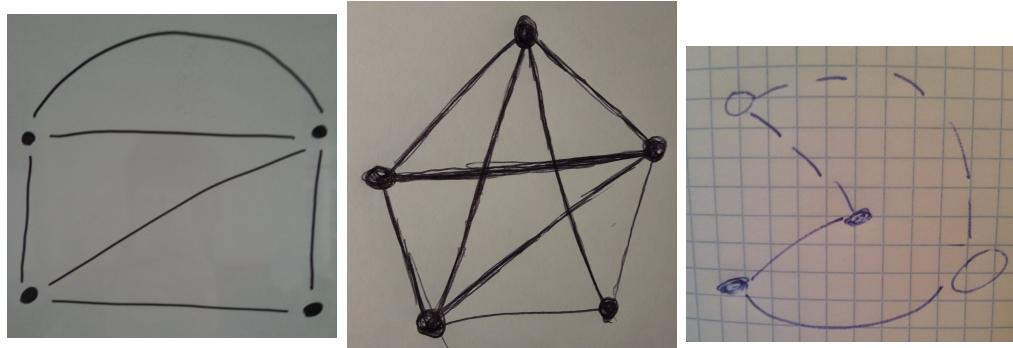
Wszystkie te metody są zależne od skali obrazu, niezbędne więc będzie pozyskanie informacji na temat wielkości rozpoznawanych obiektów. Będzie to wykonane na samym początku rozpoznawania wierzchołków, za pomocą histogramu wartości grubości obiektów.

Wykrywanie krawędzi jest podzielone na dwa przypadki: planarny i nieplanarny. W łatwiejszym do rozwiązania przypadku planarnym głównym problemem jest ocena, do którego wierzchołka prowadzi każda krawędź. W przypadku nieplanarnym dodatkowym utrudnieniem są przecięcia krawędzi. Aby rozwiązać oba te problemy wprowadzone są kryteria, według których oceniane są pary krawędzi–wierzchołek i krawędź–krawędź. Analiza spełnienia każdego z tych kryteriów prowadzi do oceny, które wierzchołki są ze sobą połączone.

Ostatni rozdział jest poświęcony testom oraz ewaluacji algorytmu. Prezentuje on szczegółowe wyniki skuteczności i czasu jego działania. Pokazane są też przypadki, które zostały błędnie rozpoznane. Część z nich wynika z niestandardowego rysowania, ale wiele innych pokazuje słabsze strony algorytmu i wskazuje pola do dalszej pracy.

1.2 Uwagi i ograniczenia pracy

Jak wiadomo, różne osoby rysują grafy na różne sposoby. Praca stara się wyłowić jak najwięcej wspólnych cech różnych stylów, a następnie rozpoznawać wierzchołki oraz krawędzie na podstawie tych wysokopoziomowych cech. Niektóre decyzje oraz parametry są wyznaczane na podstawie danych z obrazu i będą się zmieniać dla każdego przykładu, jednak nie ma prób implementacji rozpoznania stylu rysującego, a potem na tej podstawie wykrywania obiektów. Nacisk jest położony na niezależność od drobnych różnic występujących pomiędzy obrazami – zabrudzeń powierzchni, zmiany oświetlenia, nieidealne narysowane obiekty.



(a) perfekcyjny obraz dla algorytmu (b) rysunek niedbały, ale akceptowalny (c) styl nieakceptowalny

Rysunek 1.1: Różne style rysowania grafów

W związku z tym istnieje potrzeba zdefiniowania, do pewnego stopnia, typów rysunków, jakie będą poprawnie obsługiwane przez program. Styl rysowania, z jakim najlepiej będzie sobie radzić zaprezentowany algorytm, wygląda następująco:

- wierzchołki są wyraźnie grubsze od krawędzi, wypełnione,
- krawędzie są liniami ciągłymi.

W przypadku innego rysowania, jak na przykład niezamalowane wierzchołki lub przerywane krawędzie, algorytm będzie miał dużo mniejszą dokładność.

Kolejną kwestią jest podłoże. Nie może ono zawierać linii lub innych wyraźnych obiektów. Oznacza to, że grafy nie mogą być rysowane na papierze w linie lub w kratę. Obecność tego typu dodatkowych artefaktów bardzo utrudnia analizę elementów narysowanych przez użytkownika. Jeżeli jednak obraz wejściowy będzie zawierać tak wyraźnie odróżniające się od podłoża obiekty, mogą one zostać łatwo potraktowane jako krawędzie i źle wpływać na działanie reszty algorytmu. Najlepszym przykładem czystego podłoża jest zwykajna, biurowa tablica lub czysta kartka papieru.

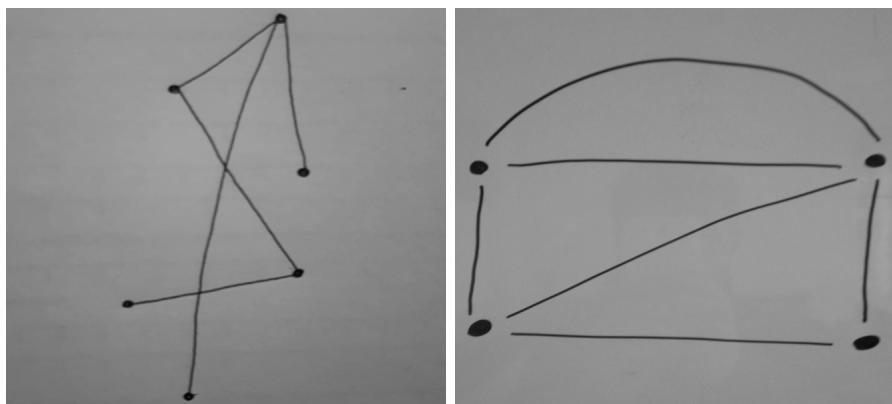
Rysunek 1.1 przedstawia różne typy obrazów. Pierwszy przykład jest idealny dla algorytmu – wyraźne krawędzie i wierzchołki. Drugi jest trudniejszy – obiekty są narysowane bardzo niedbale, wierzchołki nie są do końca zamalone, krawędzie mają różną grubość i są wieloma liniami. Algorytm jest jednak nastawiony na analizę jak najbardziej niestarannych rysunków (oczywiście w rozsądny stopniu). Trzeci rysunek jest przykładem stylu, który nie jest dobrze obsługiwany przez algorytm. Zostały popełnione wszystkie możliwe błędy: papier w kratę, przerywane krawędzie, puste wierzchołki. Zachowanie algorytmu dla takiej instancji jest niezdefiniowane. Nie oznacza to, że mogą wystąpić zapętlenia lub wycieki pamięci, lecz wynik nie musi w żadnym stopniu odzwierciedlać stanu prawdziwego.

Rozdział 2

Diagramy grafów

Ten rozdział ma na celu uświadomienie użytkownikowi złożoność problemu rozpoznawania grafów i pokazać wyzwania stojące przed algorytmem. Zadanie to jest z reguły bardzo proste dla człowieka, ale wiele jego aspektów jest zupełnie nieoczywiste podczas automatyzacji tego procesu. Następnie będzie pokazany proces normalizacji, który jest wykonywany jednakowo dla każdej instancji.

Podstawową różnicą pomiędzy grafami rysowanymi odręcznie, a komputerowo jest mnogość stylów. Tego typu różnice podczas rysowania krawędzi oraz wierzchołków występują nie tylko pomiędzy różnymi obrazkami, ale często także w obrębie jednego obrazka. W efekcie wydaje się niemożliwe stworzenie algorytmu, który bezbłędnie rozpoznałby grafy narysowane przez dowolną osobę. Wynika to też z faktu, że nawet ludzka skuteczność opisywania nie swoich rysunków nie jest perfekcyjna. Rozdział 5 prezentuje przykłady grafów, w których intencje autora nie są do końca klarowne. Problem ten nie jest dostrzegany w normalnych warunkach, ponieważ autor zna swoje intencje. Gdy obserwator ma wątpliwości, zadaje szybkie pytanie. A zdjęcia grafów, które są wysyłane drogą elektroniczną, są zazwyczaj narysowane bardzo starannie, właśnie dlatego, żeby zapobiec możliwej pomyłce.



(a) krawędzie połączone z wierzchołkami
(b) krawędzie nie połączone z wierzchołkami

Rysunek 2.1: Różne style rysowania grafów

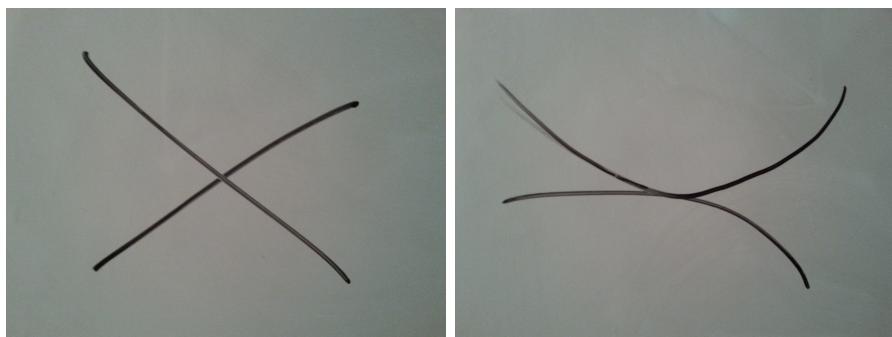
Tak więc, pomimo ujednolicenia stylu, pozostaje kwestia dużych różnic po-

między obiektami rysowanymi przez różne osoby. W pierwszej kolejności zanaliżowane będą cechy wierzchołków. Niewątpliwie mogą się one różnić:

- kształtem,
- wielkością,
- stopniem wypełnienia.

Jednak, ponieważ wymuszony jest styl z wypełnionymi wierzchołkami, można się spodziewać, że większość wnętrza będzie zamalowana. I takie też jest przyjęte założenie. Algorytm, który będzie je rozpoznawał, musi pozostać niezależny od dokładnego kształtu i wielkości. Można zauważać podobieństwo do znanej struktury skupisk (blobs). Takie spojrzenie, a także sposoby ich znajdowania, będą bazą do szukania wierzchołków na zdjęciu.

Rysunek 2.1 prezentuje dwa przykładowe style rysowania grafu. Jeden ma wierzchołki mniej wyraźne i krawędzie połączone z nimi. Na drugim wierzchołki są bardzo wyraźne, a krawędzie kończą się, nie dotykając ich. Tego typu różnice, choć oczywiste dla człowieka, muszą być brane pod uwagę podczas projektowania algorytmu.



Rysunek 2.2: Przecięcia krawędzi

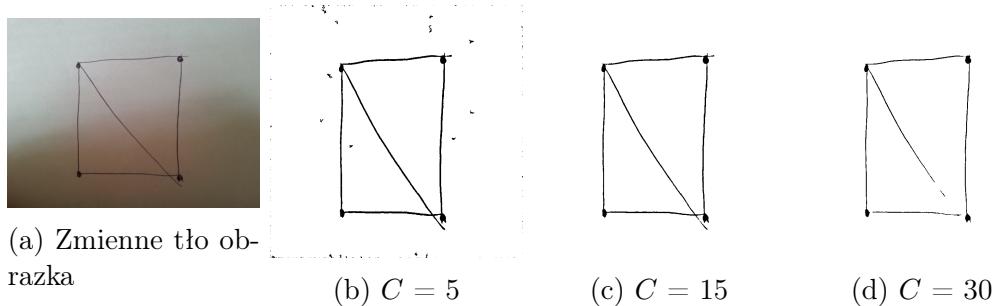
Kolejnym zagadnieniem jest przecinanie się krawędzi. Coś, co nie stanowi problemu dla człowieka, nie jest tak jednoznaczne dla komputera. Pytaniem jest: którą krawędzią podążyć podczas przecięcia? Do rozwiązania tego problemu zostanie wprowadzona ocena podobieństwa krawędzi. Przykłady przecięć krawędzi prezentuje rysunek 2.2. Pierwszy przykład jest zupełnie oczywisty. Drugi jest zapewne niejasny nawet dla czytelnika. Ten problem został opisany w [3], gdzie pokazano związek pomiędzy łatwością śledzenia przecinających się krawędzi, a kątem ich przecięcia.

2.1 Normalizacja rysunków

Początkowym krokiem w kierunku rozpoznania grafów jest maksymalne upodobnienie różnych obrazów do siebie. Zestaw kilku metod, które sprawią, że dalsza analiza będzie łatwiejsza, nazwany jest tutaj normalizacją. Jej główne cele to: ujednolicenie ubarwienia, redukcja szumów, usunięcie niedokładności rysunku.

Dla ułatwienia przyjmujemy, że obrazki są kwadratowe. Niekiedy wymuszenie ich kwadratowości lekko je znieksztalca, ale nigdy w sposób zmieniający strukturę grafu. Grafy są strukturami, przy których nie potrzebujemy wysokiej jakości obrazu. Rozsądne pogorszenie jakości krawędzi lub wierzchołków nie wpływa na możliwość odczytu ani wierzchołków, ani krawędzi, a znaczaco poprawia wydajność niektórych metod. Można więc wypośrodkować te dwie cechy, dużo wyżej ceniąc jednak wynik ewaluacji. W tej pracy wymiary obrazów wynoszą $600px \times 600px$ - wartość wyznaczona eksperymentalnie, zapewnia ona dobry balans pomiędzy skutecznością a czasem działania algorytmu.

2.1.1 Zmiana przestrzeni kolorów



Rysunek 2.3: Progowanie adaptywne z różnymi wartościami

Pierwszym elementem normalizacji obrazu jest zamiana przestrzeni kolorów z RGB na binarną. Odbywa się to w dwóch etapach: najpierw konwersja RGB na szary obraz, a następnie na obraz binarny.

Konwersja przestrzeni kolorów z trzech kanałów (RGB) na jeden kanał (szary obraz) jest przeprowadzona za pomocą algorytmu ze standardu CCIR 601¹:

$$Y = 0.299R + 0.587G + 0.114B$$

Zamiana na wersję binarną ma na celu wydobycie diagramu z tła. Temat ten jest dobrze znany i jest wiele metod jego realizacji [4, str. 82]. Przy wyborze metody należy wziąć pod uwagę cechy opisywanych diagramów:

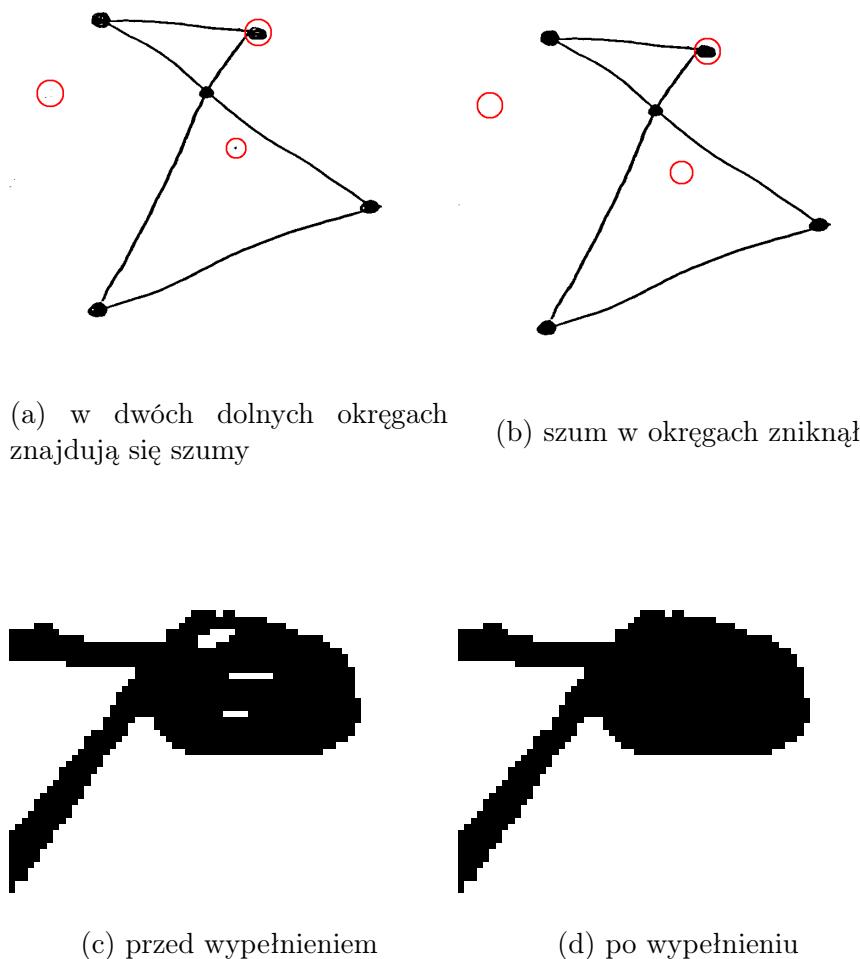
- może występować nierównomierne oświetlenie obrazu (zmiana koloru tła),
- narysowane obiekty nie pozostawiają cienia, więc zmiana koloru tła nie jest zbyt gwałtowna.

Te dwie cechy dają dobre pole do wykorzystania metody progowania adaptynego (adaptive thresholding) [4, str. 88]. Jest ona odporna na zmieniające się tło dzięki liczeniu średniego, lokalnego oświetlenia. Metoda przyjmuje dwa parametry: wielkość lokalnego okna (w) oraz odchylenie od średniej (C), jakie musi mieć piksel, żeby został uznany za zamalowany. Taktyka wybierania wielkości

¹<http://www.itu.int/rec/R-REC-BT.601/>

okna jest następująca: najmniejsze, w którym zawarty będzie fragment tła, niezależnie, w którym miejscu na obrazie będzie przyłożone. Zbyt małe okno spowoduje odrzucenie jednolitego, zamalowanego obszaru, a zbyt duże będzie sobie źle radziło ze zmieniającym się tłem. Podobnie wygląda wybór stałej C . Zbyt mała spowoduje zaakceptowanie wielu szumów, a zbyt duża odrzuci mnóstwo wyraźnych fragmentów rysunku. Wartości te zostały eksperymentalnie wyznaczone na $w = 0.05 \cdot \text{rozmiar_obrazka} = 0.05 \cdot 600 = 30$ oraz $C = 15$. Rysunek 2.3 przedstawia rezultaty użycia różnej stałej podczas progowania adaptynego.

2.1.2 Usuwanie szumów



Rysunek 2.4: Usunięcie szumów oraz wypełnienie wierzchołków

Następnym elementem jest usuwanie szumów obrazu. Popularnymi technikami są wygładzanie gaussowskie [4, str. 40], erozja i dylatacja [4, str. 186]. Jednak w tym przypadku należy wziąć pod uwagę, że krawędzie grafu są "delikatne". Czasami są cienkie, zwłaszcza przy końcach. Sprawia to, że wykonując operacje, które normalnie sprawdzają się przy usuwaniu szumów, a w taki lub inny sposób

powodują uśrednienie pikseli z ich otoczeniem, mogą łatwo zniszczyć strukturę grafu lub sprawić, że będzie ona dużo cięższa do odczytania.

Krawędzie są obiektami ciągłymi. Jest to jedna, podłużna linia. W przeciwieństwie do szumów, które najczęściej są w postaci "pieprzu i soli" (salt and pepper noise, [4, str. 31]), czyli małych obszarów, których jasność nie odpowiada wartości jego sąsiadów. W związku z tym bardzo dobrze sprawdza się prosta metoda usunięcia przedmiotów, których pole nie jest dostatecznie duże. W tym celu wyznaczone są kontury [5, str. 32], a następnie policzone jest pole każdego obiektu [6]. Po tych operacjach, obiekty o małych polach $0.0001 \cdot \text{pole_obrazu}$ są usunięte. Ta operacja wykonana jest "w obu kierunkach", czyli zarówno do małych białych obiektów na czarnym tle, jak i czarnych na tle białym. Rysunek 2.4 przedstawia przykład działania powyższej metody. Można zauważyć, że zniknęły drobne zanieczyszczenia po lewej stronie i w środku grafu, a także wierzchołki grafu stały się bardziej jednolite.

2.1.3 Zamykanie



Rysunek 2.5: Ciąg trzech iteracji operacji zamykania na przykładzie wierzchołka grafu

Ostatnią operacją przygotowującą jest zamykanie (closing) [7, str. 104]. Polega ona na kolejnym wykonaniu: dylatacji, erozji. Ważna jest kolejność tych operacji. Efektem zamykania jest jedna iteracja wypełnienia niezamalowanych części obrazu, sąsiadujących z dużą liczbą pikseli zamalowanych (w praktyce oznacza to prawie całkowite otoczenie takimi pikselami). Można zauważyć podobieństwo tego przekształcenia do usuwania szumów. Ma jednak ważne, inne zastosowanie. Usuwanie szumów nie będzie dobrze działało w momencie, gdy obszar, który powinien być wypełniony jest otwarty – nie jest całkowicie otoczony pikselami czarnymi. Wtedy poprzednia metoda dołączy jego pole do pola całego tła i go nie zmieni. Wtedy lepiej sprawdza się zamykanie. Przykład jego działania przedstawia rysunek 2.5. Widać również problem związany z tą operacją: wielokrotne jej użycie doprowadza do stopniowego wypełniania także powierzchni pomiędzy krawędziami grafu. W efekcie może to doprowadzić do sytuacji, w której przecięcia krawędzi będą nieroróżnicjalne z wierzchołkami. Najlepsze wyniki w tej pracy dało użycie jednej iteracji zamykania. Nie zawsze efekt działania jednej iteracji jest klarownie widoczny na obrazie, jednak jej wpływ na algorytm jest bardzo pozytywny.

Rozdział 3

Rozpoznawanie obiektów

Rozdział ten jest poświęcony głównej części pracy, czyli rozpoznawaniu wierzchołków oraz połączeń pomiędzy nimi. Wiele z opisanych algorytmów jest parametryzowanych. Wartości parametrów zostały wyznaczone za pomocą własnych testów oraz podpowiedzi z literatury. Każdy podrozdział będzie zawierał dokładny opis algorytmu, jaki prezentuje wraz z taktyką doboru parametrów. Dokładne wartości parametrów oraz inne niewpływające na ideę algorytmu szczegóły implementacji zostały zebrane i omówione w rozdziale 4.

W celu ułatwienia śledzenia kolejnych algorytmów, wyniki ich działania będą ilustrowane przykładami na różnych grafach. Natomiast zapis algorytmiczny będzie operować na abstrakcyjnym – *prepared_image*, binarnym obrazie będącym rezultatem normalizacji zdjęcia (zobacz rozdział 2.1).

3.1 Rozpoznawanie wierzchołków

Podczas projektowania algorytmu służącemu rozpoznawaniu wysokopoziomowych cech obiektu (a za taką na pewno należy uznać pytanie „Czy obiekt jest wierzchołkiem?”), dobrze jest odpowiedzieć sobie na pytanie: „W jaki sposób człowiek odróżnia ten obiekt od innych?”. Takie podejście okazuje się zaskakująco skuteczne podczas projektowania algorytmów. Intuicja, jaka posłużyła jako podstawa do opisu wierzchołków, brzmi:

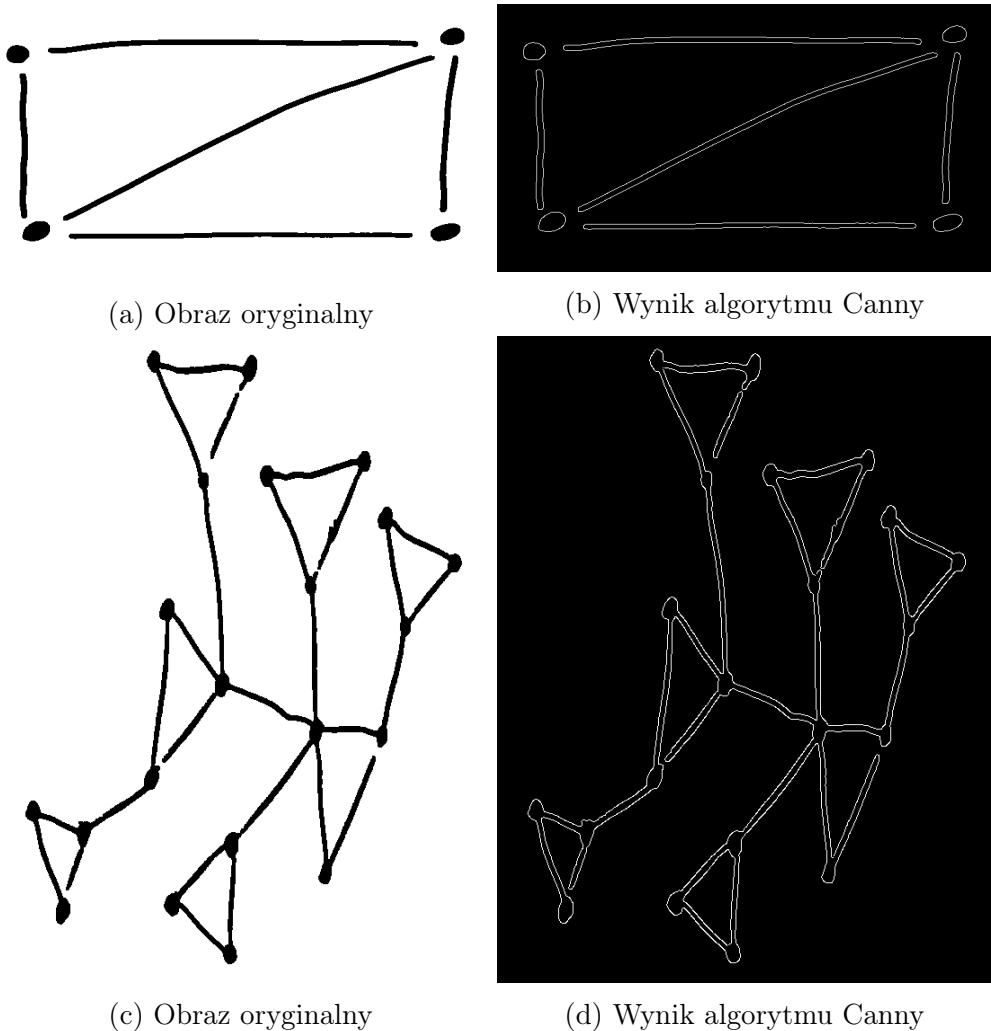
- jest fragmentem obrazu wyraźnie ciemniejszym od otoczenia,
- ma elipsoidalny kształt,
- jest rogiem obrazu¹,
- jest wyraźnie grubszy od krawędzi.

Każda z tych obserwacji jest mocno powiązana ze skalą, z jaką patrzymy na obraz. Coś, co może wydawać się wierzchołkiem w małej skali, wcale nie musi nim być, gdy popatrzymy „z daleka”. Pomimo że istnieją metody wykrywania

¹Nawiązanie do metody wykrywania wierzchołków opisanej w [8]. Rogiem obrazka nazywamy obszar, który zmienia się przy przesunięciu o w każdą ze stron. Przykładowo krawędzie pozostają takie same, gdy poruszamy się wzduż nich.

cech obrazka niezależne od jego skali [9], to nie dają one dostatecznie wiele informacji, żeby przeprowadzić wiarygodne wykrywanie wierzchołków. Niezbędne jest zatem poznanie podstawowej cechy analizowanego obrazu – grubości linii, jaką narysowany jest graf.

3.1.1 Wykrywanie grubości linii



Rysunek 3.1: Działanie algorytmu Canny

Pierwszym elementem jest operacja wykrywania krawędzi obrazu (nie mylić z krawędziami grafu). Jest to znany i dobrze opracowany temat. Dodatkowo sprawę ułatwia fakt, że obraz jest czarno-biały (po zastosowanej normalizacji). W tej pracy użyty jest najpopularniejszy algorytm – Canny [10]. Zwraca on obraz, w którym zaznaczone są jedynie fragmenty będące krawędziami obrazu. Przykład jego działania prezentuje Rysunek 3.1. Wynik działania algorytmu Canny:

$$\text{canny_edges} = \text{Canny}(\text{prepared_image})$$

Grubość linii użyta do narysowania obrazu jest odległością pomiędzy

krawędziami linii, co bardzo dobrze widać na Rysunku 3.1b. Ponieważ każda krawędź grafu jest linią rysowaną długopisem lub markerem, jej grubość pozostaje mniej więcej niezmienna na całym obrazku (poza wierzchołkami)².

Algorytm Line Width Detection

Wejście *canny_edges*

Wyjście *width_list* – tablica grubości linii we wszystkich punktach krawędzi

width_list := lista wartości grubości linii, początkowo pusta

for każdy punkt *p* należący do krawędzi *canny_edges* **do**

neighbours(p) := sąsiedztwo *p* w *canny_edges*

neighbours(p) jest kawałkiem krawędzi o długości *window_size*

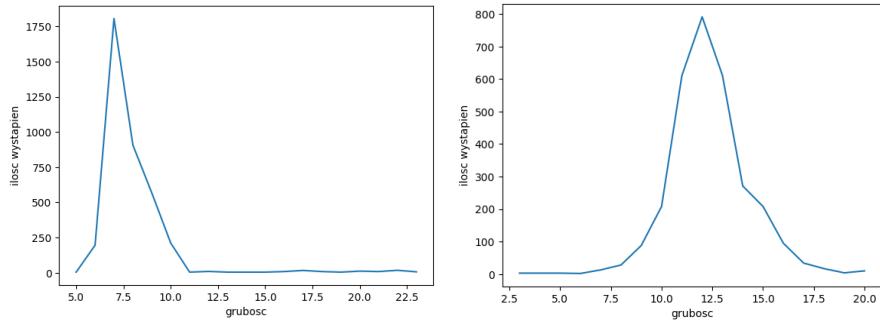
l := prosta prostopadła do *neighbours(p)*, przechodząca przez *p*

p' := punkt przecięcia *l* z krawędzią *canny_edges*, różny od *p*

 dodaj odległość pomiędzy *p* a *p'* do *width_list*

end for

return *width_list*



(a) dla rysunku 3.1b

(b) dla rysunku 3.1d

Rysunek 3.2: Histogramy grubości linii

Algorytm **Line Width Detection** prezentuje sposób wyznaczenia grubości linii. Zmienna *window_size* jest parametrem algorytmu. Sposób wyboru tego parametru jest opisany w rozdziale 4.1. Wynikiem działania algorytmu Line Width Detection jest tablica, która dla każdego punktu na krawędzi zawiera informację, o tym jak daleko jest najbliższa krawędź pod kątem prostym. Informację tę można też zinterpretować jako grubość linii. Analiza histogramu częstości występowania zadanej grubości linii (rysunek 3.2) podpowiada, którą wartość przyjąć za poprawną grubość (wartość modalną). Ponieważ linie tworzące krawędzie grafu, w zdecydowanej większości przypadków, mają sumarycznie większą długość niż linie tworzące wierzchołki, to modalna wartość odległości jest też modalną grubością linii na rysunku. Nazwijmy i zapamiętajmy wartość modalną wyznaczoną tablicy.

$$\text{line_width} = \text{mode}(A(\text{canny_edges}))$$

² Jednakowość grubości linii na obrazku jest jednym z założeń, które jest uproszczeniem głównego problemu. Istnieją przypadki, gdzie powoduje ono błędne rozpoznanie wierzchołka. Takie przypadki będą przedstawione w rozdziale 5

Wartość ta posłuży do rozpoznawania wierzchołków – wyraźne odchylenia od niej będą kandydatami na wierzchołek.

3.1.2 Detekcja skupisk

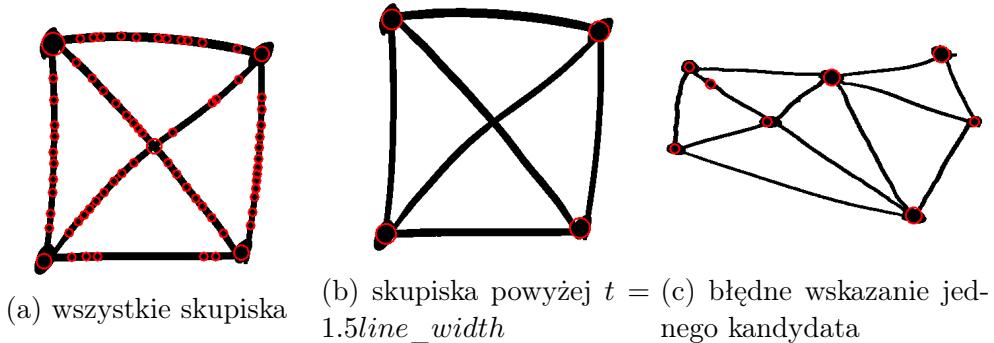
Pierwszą metodą znalezienia wierzchołków będzie algorytm wykrywania skupisk (blobów) - Laplacian of Gaussian (LoG) [2]

Metoda ta najpierw aplikuje filtr Gaussa o zadanej skali t , a następnie operator Laplace'a [7, str. 137]. W celu wykrywania skupisk o różnej wielkości, do dwuwymiarowego obrazka, dokładany jest trzeci wymiar – t , czyli różne wartości parametru skali użytej przy filtrze Gaussa. Następnie skupiska wykrywane są poprzez wyszukanie ekstremów wynikowej funkcji w trójwymiarowej przestrzeni (x, y, t) .

Algorytm wskazuje miejsca, które wyróżniają się od lokalnego otoczenia oraz podaje skalę t , w jakiej dane ekstremum zostało znalezione. Skala jest przybliżeniem wielkości obiektu. Bloby znalezione w skali t mają wielkość około $t\sqrt{2}$. Jego wynikiem jest lista trójkę:

$$\text{Laplacian_of_Gaussian(prepared_image)} = [(x_1, y_1, t_1), (x_2, y_2, t_2), \dots]$$

gdzie x_i, y_i oznaczają położenie skupiska na obrazie a t_i jest skalą, w jakiej skupisko zostało znalezione.



Rysunek 3.3: Skupiska znalezione algorytmem Laplacian of Gaussian

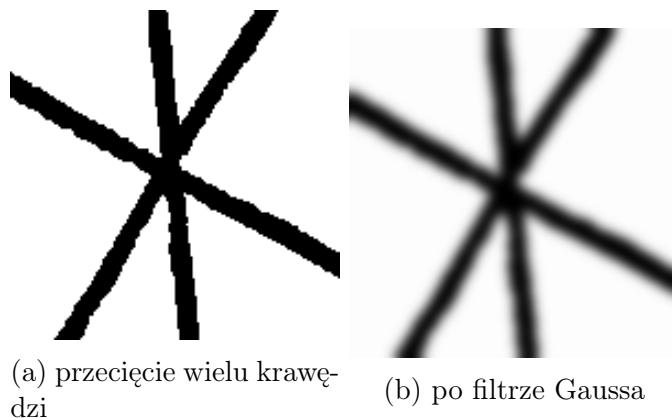
Należy jednak zwrócić uwagę, że w odpowiedniej skali, bardzo podobne do skupisk są części krawędzi. Efektem tego jest znalezienie bardzo wielu wyników w małej skali, które leżą na krawędziach. Można to wyeliminować w łatwy sposób, odrzucając skupiska, które mają niewielką skalę. Do określenia dolnego limitu rozmiaru, od którego blob jest akceptowany, użyta jest wartość *line_width*. Bardzo skuteczną metodą jest przemnożenie jej przez niewielką (ale większą od 1) stałą i ustalenie tej wartości jako dolny limit skali.

Rysunek 3.3 przedstawia działanie algorytmu. Znalezione skupiska są zaznaczone. Pierwszy przykład pokazuje wynik algorytmu bez odrzucenia jakichkolwiek wyników. Zaakceptowanych zostało wiele błędnych, małych rezultatów, które są częścią krawędzi. Można również zauważyc, że skupisko znalezione na przecięciu krawędzi jest większe od tych znalezionych na krawędziach. Sugeruje

to, że będzie wiele przypadków, gdy takie miejsca będą trudne do odróżnienia od prawdziwych wierzchołków.

Kolejne dwa rysunki pokazują odrzucone małe skupiska. Wtedy algorytm dość dokładnie znajduje położenie i wielkość prawdziwych wierzchołków. Na rysunku 3.3c widać także błędnie zaznaczony fragment krawędzi, który jest grubszy od pozostałych. Takie wskazania będą odrzucane kolejnymi metodami.

3.1.3 Konwolucja z okrągłym jądrem



Rysunek 3.4: Obraz problematyczny dla algorytmu Laplacian of Gaussian

Głównym zadaniem kolejnej metody jest wyeliminowanie jednego z najczęstszych błędów, który jest generowany przez poprzednią metodę: przecięcia krawędzi, zwłaszcza wielu krawędzi. Często istnieją regiony obrazka, które są „gęste”, czyli z jednej strony mają wiele pikseli czarnych, ale też są dla użytkownika łatwe do odróżnienia od wierzchołka. Przykład takiego miejsca jest na rysunku 3.4a. Filtr Gaussa powoduje ważone uśrednienie leżących niedaleko pikseli. W efekcie taki region może być trudniejszy do odróżnienia od wierzchołka. Algorytm Laplacian of Gaussian często zwraca takie fragmenty jako poprawne boby o sporej skali. Potrzeba więc algorytmu, który będzie bardziej surowy dla takich regionów i nie będzie operował na rozmazanym obrazie.

Opieramy się na obserwacjach³, że wierzchołki są grubsze oraz mają elipoidalny kształt. Te dwie cechy da się sprawdzić jednocześnie najpopularniejszą metodą w rozpoznawaniu obrazów: konwolucją o odpowiednim jądrze. Dla każdego wierzchołka, zwróconego przez poprzedni algorytm, chcemy sprawdzić, czy posiada obie te cechy. Taki efekt da zastosowanie jądra w kształcie koła. Wielkość koła jest wyznaczana na podstawie grubości linii obrazu. Całą procedurę prezentuje algorytm **Circular Kernel Convolution**. W algorytmie pojawiają się dodatkowe parametry: *kernel_size_ratio* oraz *acceptance_ratio*. Zmienna *kernel_size_ratio* opisuje o ile większy ma być zamalowany obszar od grubości linii obrazka. Wybór zbyt małej wartości spowoduje bezużyteczność całej metody.

³ Obserwacje poczynione na początku rozdziału stają się założeniami. Sprawdzają się one w zdecydowanej większości przypadków, więc dla uproszczenia programu zakładamy ich prawdziwość we wszystkich przypadkach.

Algorytm Circular Kernel Convolution

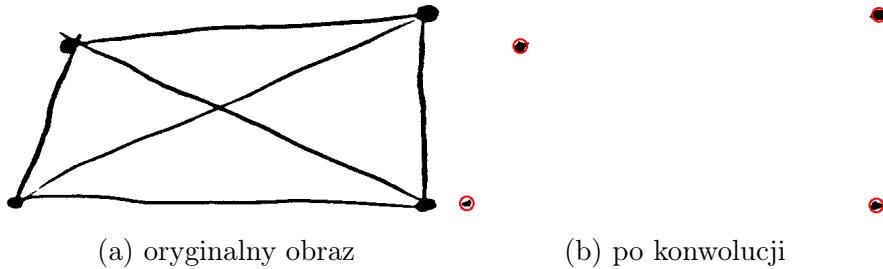
Wejście *prepared_image*, *Laplacian_of_Gaussian(prepared_image)*

Wyjście *vertices* - okrojona lista wierzchołków

```
vertices := Laplacian_of_Gaussian(prepared_image)
kernel := array[prepared_image.height][prepared_image.width]
r := line_width · kernel_size_ratio
kernel[x, y] = [1 for x · x + y · y < r · r, otherwise 0]
conv = convolve(prepared_image, kernel)
for v należący do vertices do
    x, y = v
    if conv[x, y] < πr² · acceptance_ratio then
        usuń v z vertices
    end if
end for
return vertices
```

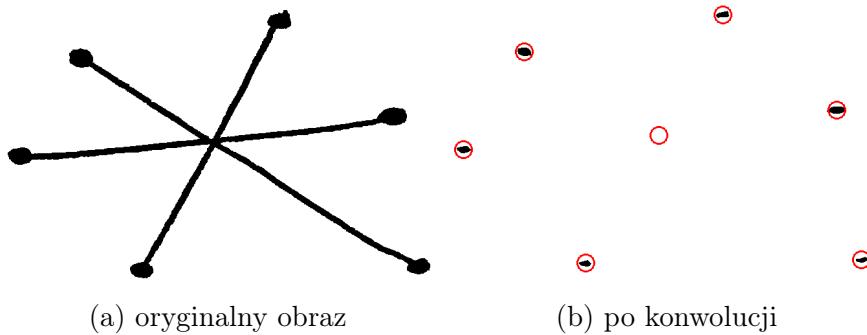
Jeżeli wartość będzie zbyt duża, to wiele poprawnych wierzchołków zostanie odrzuconych. Parametr *acceptance_ratio* określa, jaki procent pikseli musi być czarnych, żeby zaakceptować wierzchołek. Ustawienie go na 1 będzie oznaczało najbardziej surową wersję algorytmu. Ze względu na fakt, że nie zawsze wszystkie szумy zostają usunięte w procesie przygotowania obrazu, ta wartość powinna być mniejsza. Szczegóły dotyczące doboru obu wartości są opisane w rozdziale 4.

Przykład działania algorytmu przedstawia rysunek 3.5. Czarne piksele ozaczają miejsca, które spełniły warunki postawione w algorytmie. Dodatkowo, podobnie jak wcześniej, zaznaczone są wierzchołki wykryte metodą Laplacian of Gaussian. W pierwszym przypadku – rysunek 3.5 widzimy potwierdzenie tych samych miejsc. Ciekawszy jest drugi przypadek. Jest to pełny graf, którego fragment jest na rysunku 3.4 i był użyty jako przykład błędego działania algorytmu LoG. Widać, że LoG zaznaczył wierzchołek w spodziewanym, lecz błędnym miejscu przecięcia się wielu krawędzi. Natomiast ten fragment nie spełnił warunku obecnego algorytmu. Brak rozmażania spowodował, że nie było koła o średnicy znacznie przekraczającej grubość linii, które byłoby prawie w całości czarne. W efekcie ten wierzchołek został odrzucony przez algorytm Circular Kernel Convolution.



Rysunek 3.5: Działanie algorytmu Circular Kernel Convolution

Pozostała jeszcze do wyjaśnienia kwestia relacji zbioru wyników algorytmu Circular Kernel Convolution do zbioru wyników algorytmu LoG. Ponieważ skoro,



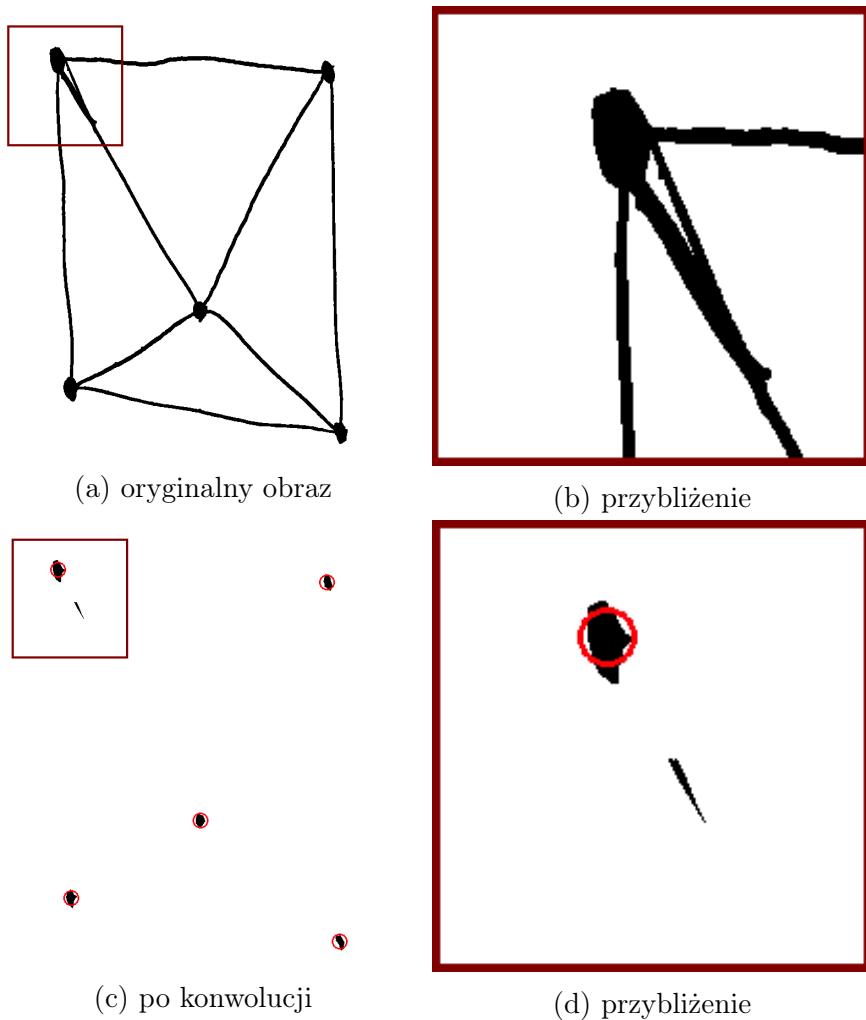
Rysunek 3.6: Działanie algorytmu na przykładzie z rysunku 3.4

obecny algorytm zdaje się weryfikować wszystkie wyniki poprzedniego i nie produkuje żadnych dodatkowych, to dlaczego należy robić analizę skupisk? Powodów jest kilka. Pierwszy jest taki, że wynik konwolucji nie daje bezpośrednio żadnej informacji o ilości różnych skupisk. Zaznacza jedynie, że dane miejsce spełnia warunki narzucone przez algorytm. To jednak można dość łatwo obejść. Patrząc na efekt działania widać, że piksele spełniające warunek oraz należące do jednego wierzchołka, należą też do mniejszego blobu, stworzonego przez konwolucję. Łącząc piksele w spójne składowe, dostajemy podział na wierzchołki. Istnieje jednak typ obrazów, przy których ten algorytm wprowadza w błąd. Są to regiony, w których występuje spore zgrubienie krawędzi tuż przed wierzchołkiem. Wtedy kształt wyniku konwolucji jest bardzo mylący. Algorytm wykrywania blobów, dzięki analizie ekstremów funkcji gęstości, często poprawnie znajduje środek i wielkość takiego wierzchołka. Przykład na rysunku 3.7. Czerwonym okręgiem zostały zaznaczone wierzchołki znalezione wykrywaniem skupisk. Jest widoczne, że zbiór wyników algorytmu Circular Kernel Convolution niekoniecznie jest podzbiorem algorytmu LoG. Innym powodem jest bardzo dobra jakość znalezionych skupisk przez algorytm LoG. Patrząc na wiele wyników działania tej metody, można zauważać, że jej wyniki nie zawsze są tylko tymi, które są rzeczywistymi wierzchołkami, ale każdy z poprawnie zaznaczonych wierzchołków jest bardzo dokładnie umiejscowiony w środku ciężkości skupiska oraz jego wielkość jest wyliczona dokładnie.

3.1.4 Wykrywanie grubych krawędzi

Mając na uwadze rysunek 3.3a, na którym zostało wykryte wiele wierzchołków na krawędziach z powodu braku ograniczenia dolnego na wielkość skupisk oraz widząc duże ograniczenia, w tym samym przypadku, algorytmu Circular Kernel Convolution, widać potrzebę odrzucania kolejnego typu błędnych wskazań – przypadków, w których wartość *line_width* – grubości krawędzi nie pokrywa się ze wszystkimi krawędziami lub ich fragmentami. Jeżeli istnieje krawędź znacznie grubsza, niż wartość modalna grubości krawędzi to poprzednie metody mają dużą szansę zwrócić takie miejsce jako poprawny wierzchołek.

Kolejny algorytm odrzucania niepoprawnych wskazań wykorzystuje metodę opublikowaną w [8] i jest znany jako algorytm Harris. Opisuje ona każdy piksel obrazu za pomocą funkcji z wartości macierzy sąsiedztwa danego miej-



Rysunek 3.7: Przykład regionu poprawnie wykrywanego przez LoG oraz źle przez konwolucję

sca⁴. Następnie wartość funkcji jest interpretowana, zaliczając każdy piksel do jednej z trzech klas: tło, krawędź lub róg. Metoda ta sprawdza się świetnie w wykrywaniu rogów obrazu⁵. Rezultat tego algorytmu zostanie użyty dwukrotnie w tej pracy: do eliminowania niepoprawnych wierzchołków oraz w rozdziale 3.2 do wykrywania przecięć krawędzi. Zarys użycia metody Harrisa:

- obliczenie macierzy wartości własnych obrazu,
- obliczenie funkcji Harrisa–Stephensa,
- zaklasyfikowanie każdego piksela jako krawędź, tło lub róg,

⁴ Macierzą sąsiedztwa nazywamy tutaj kawałek $n \times m$ obrazu, który otacza dany piksel. Aby piksel był w środku, obie wartości muszą być nieparzyste. Ponieważ obraz jest szary (ma tylko jeden kanał koloru), ta macierz jest dwuwymiarowa

⁵ Wprowadzenie do niej oraz przykłady działania można zobaczyć na http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_features_harris/py_features_harris.html

- użycie metody podobnej do Circular Kernel Convolution, lecz zamiast białych pikseli, odrzucamy wierzchołki piksele zaklasyfikowane jako krawędzie.

Początek jest niestandardowy – lekkie rozmazanie obrazu. Użycie konkretnej metody nie ma znaczenia, w implementacji został użyty filtr Gaussa. Pomimo argumentów za szkodliwością tej operacji w poprzednich rozdziałach, w tym miejscu bardzo poprawia ona rezultaty. Dzieje się tak z powodu niedoskonałości krawędzi występujących na rysunkach. Są one często lekko poszarpane i nieregularne, przez co algorytm, który ma za zadanie wykrywać również rogi obrazu, jest dużo mniej pewny czy w danym miejscu jest krawędź czy róg. Rozmazanie obrazu „wygładza” takie miejsca, dzięki czemu są dużo bardziej podobne do krawędzi. Aby lepiej zrozumieć ten proces, należy zagłębić się w działanie algorytmu Harrisa.

Pierwowzór metody został przedstawiony przez Moravec'a [11]. Rozważa on zmianę intensywności obrazu podczas przesuwania lokalnego prostokątnego okna po obrazie i mierzy zmianę intensywności obrazu wraz z przesuwaniem okna o niewielką wartość w każdą ze stron. Można wtedy podzielić wyniki na 3 grupy:

- intensywność obrazu nie zmienia się wraz z przesunięciem w żadną ze stron – wtedy taki region uznawany jest za "płaski",
- intensywność jest niezmienna w jednym z kierunków (ale na obu zwrotach) i bardzo zmienna prostopadle do tego kierunku – tym charakteryzuje się krawędź,
- intensywność zmienia się we wszystkich kierunkach przesuwania okna – oznacza to, że region jest rogiem.

Harris oraz Stephens zaproponowali funkcję do mierzenia tej zmiany:

$$E_{x,y} = \sum_{u,v} w_{u,v} |I_{x+u,y+v} - I_{u,v}|^2$$

Gdzie (x, y) jest zbiorem przesunięć obrazu $\{(1, 0), (1, 1), (0, 1), (-1, 1)\}$, I mierzy intensywność w danym pikselu, a w jest macierzą $n \times m$ – lokalnym oknem obrazu. Macierz w może być jednostkowa lub aplikować filtr Gaussa⁶.

Korzystając z rozwinięcia Taylora, można przekształcić powyższe równanie:

$$E_{x,y} = [x \quad y] M(x, y)^T$$

gdzie

$$M = \sum_{u,v} w_{u,v} \begin{bmatrix} I_u I_u & I_u I_v \\ I_u I_v & I_v I_v \end{bmatrix}$$

I_u oraz I_v są pochodnymi obrazu w kierunkach u i v . Niech α, β będą wartościami własnymi macierzy M . Wtedy również zachodzą 3 przypadki:

- α i β są obie bliskie 0 – wtedy obraz nie zmienia intensywności
- $\alpha \ll \beta$ lub $\beta \ll \alpha$ – obraz zmienia intensywność w jednym kierunku

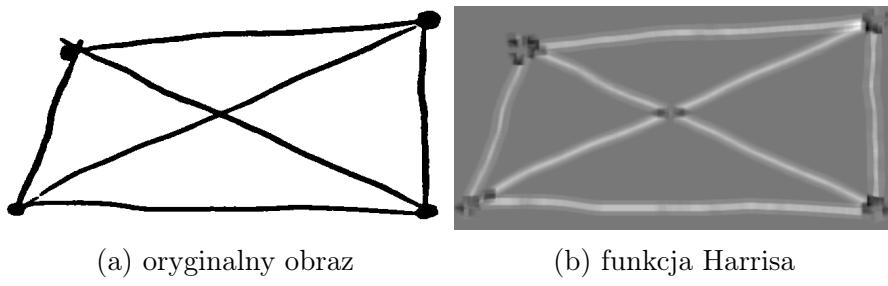
⁶ Dla lepszego zrozumienia można w przedstawić jako macierz o rozmiarach obrazu, która jest jednostkowa na analizowanym regionie (lokalnym oknie) i zerowa wszędzie indziej.

- $\alpha \sim \beta$ oraz $\alpha, \beta \gg 0$ – obraz zmienia intensywność w każdym kierunku

Można wtedy policzyć funkcję:

$$R = \alpha\beta - k(\alpha + \beta)^2$$

- gdy $R < 0$, region jest krawędzią
- gdy $R \sim 0$ region jest płaski
- gdy $R \gg 0$ region jest rogiem

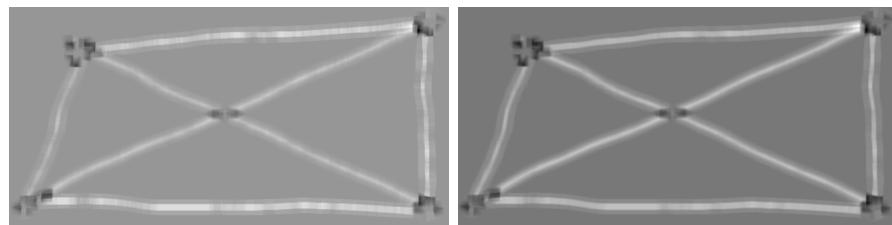


Rysunek 3.8: Wynik działania funkcji Harrisa na obrazie

Daje to elegancki sposób na wyznaczenie rogów i krawędzi obrazu. Po znormalizowaniu wyników funkcji do przedziału $[0, 255]$ oraz ich odwróceniu, wynik funkcji dla rysunku grafu przedstawia się tak jak na rysunku 3.8. Krawędzie grafu są wyraźnie jaśniejsze od reszty obrazu. Są też miejsca wyraźnie ciemne – zostały one zaklasyfikowane jako rogi. Można też zauważyc, że wnętrza wierzchołków są zaklasyfikowane tak samo, jak tło. Dzieje się tak w przypadku kiedy wierzchołek jest wyraźnie większy od wielkości okna użytego w algorytmie. Wtedy intensywność nie zmienia się, gdy okno przesuwa się wewnątrz wierzchołka.

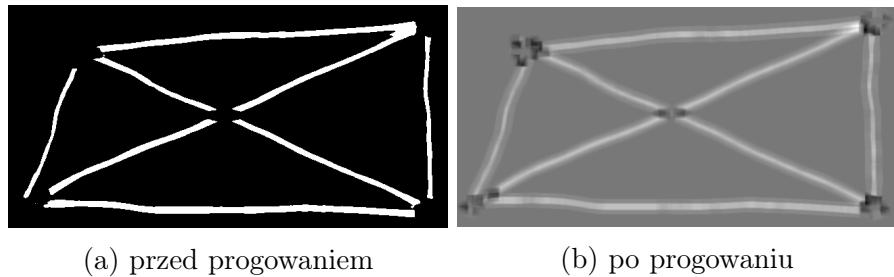


Rysunek 3.9: przybliżona krawędź grafu



Rysunek 3.10: Działanie metody Harrisa na wygładzonym obrazie

Teraz można lepiej zrozumieć, że krawędzie, które nie są gładkie, a przy rysowaniu odręcznym większość jest taka (co pokazuje rysunek 3.9), mogą mocno wpływać na wynik funkcji R i w efekcie krawędzie będą trudniejsze do odróżnienia od tła. Przykład działania na obrazie oryginalnym i wygładzonym jest na rysunku 3.10. Różnica w jasności krawędzi może być trudna do zauważenia (jest dużo wyraźniejsza na skośnych krawędziach), ale za to widać, że tło na wygładzonym obrazie jest ciemniejsze, co wskazuje, że krawędzie są łatwiejsze do odróżnienia w drugim przypadku.



Rysunek 3.11: Progowanie adaptywne po algorytmie Harrisa

Rysunek 3.8 sugeruje także metodę odróżnienia krawędzi od reszty obrazu. Jest to zwyczajne progowanie. Dla minimalnie większej skuteczności można użyć progowania adaptywnego, żeby uniezależnić się od koloru tła. Oczywiście należy ustalić wartość progu, ale fragmenty, które mają zostać zaakceptowane (krawędzie) bardzo mocno się wyróżniają, więc nie sprawia to problemu. Wynik progowania adaptywnego zaaplikowanego do obrazu po algorytmie Harrisa jest przedstawiony na rysunku 3.11. Wynik tej operacji:

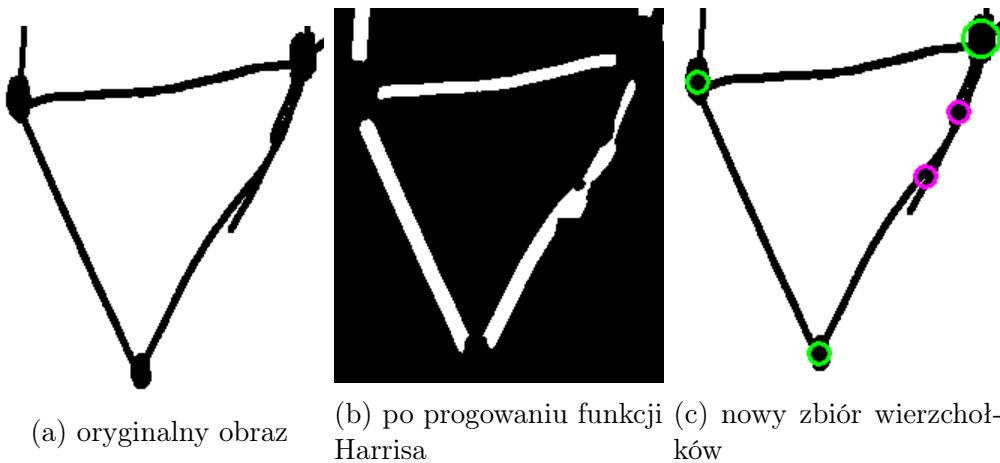
$$\text{graph_edges} = \text{adaptive_threshold}(R)$$

Algorytm Circular Kernel Convolution for edges

Wejście $\text{prepared_image}, \text{graph_edges}, \text{vertices}$ – lista obecnie znalezionej wierzchołków

Wyjście vertices - okrojona lista wierzchołków
for v należący do vertices **do**
 $x, y, r = v$ - dzięki metodzie Laplacian of Gaussian mamy dostęp także do promienia skupiska
 $\text{kernel}[x, y] = [1 \text{ for } x \cdot x + y \cdot y < r \cdot r, \text{otherwise } 0]$
 $\text{conv} = \text{convolve}(\text{prepared_image}, \text{kernel})$
if $\text{conv}[x, y] > \pi r^2 \cdot \text{edge_acceptance_ratio}$ **then**
 usuń v z vertices
end if
end for
return vertices

Zaznaczone są więc krawędzie grafu. Warto zwrócić uwagę, że zostaną zaznaczone nawet grubsze krawędzie lub przynajmniej ich fragmenty. Pozwala to



Rysunek 3.12: Przykład odrzucenia wierzchołków metodą Harrisa

na użycie lekko zmodyfikowanej metody Circular Kernel Convolution – **Circular Kernel Convolution for edges**. Tym razem obraz skanowany jest w poszukiwaniu pikseli należących do krawędzi. Jeżeli obszar, który został wytypowany jako potencjalny wierzchołek, ma ich zbyt dużo (próg jest określony przez parametr *edge_acceptance_ratio*), jest on odrzucany. Można zauważać niewielkie zmiany w stosunku do algorytmu Circular Kernel Convolution. Przede wszystkim nie jest używana taka sama wartość promienia dla wszystkich wierzchołków. Dzięki użyciu metody Laplacian of Gaussian znany jest przybliżony promień skupiska. Pojawia się też nowy parametr *edge_acceptance_ratio* – spełnia on taką samą funkcję jak *acceptance_ratio* w poprzednim algorytmie, lecz jego wartość niekoniecznie musi być taka sama. Działanie algorytmu, które zaowocowało odrzuceniem błędnego wierzchołka prezentuje rysunek 3.12. Widzimy na nim interesujący kawałek grafu. Na rysunku 3.12c zostały zaznaczone wierzchołki znalezione przez dwa poprzednie algorytmy. Kolorem zielonym zaznaczone są te, które przeszły weryfikację nieznajdowania się na grubej krawędzi. Różowe wierzchołki zostały odrzucone. Rysunek 3.12b nie pozostawia wątpliwości dlaczego. Algorytm Harrisa wykrył grubszą krawędź i zaznaczył wiele pikseli leżących pod tymi wierzchołkami jako należące do niej.

3.1.5 Podsumowanie

Na tym kończy się przesiewanie kandydatów na wierzchołki grafu. Wierzchołek, który przeszedł przez trzy opisane algorytmy:

- Laplacian of Gaussian,
- Circular Kernel convolution,
- Circular Kernel Convolution for edges,

jest uznawany już do końca za poprawnie wyznaczony. Jest to bardzo istotne, ponieważ lista wierzchołków nie będzie już modyfikowana, a pozycja i rozmiar każdego wierzchołka będzie potem wykorzystany do badania istnienia krawędzi

pomiędzy nimi. Oznacza to, że jakikolwiek błąd popełniony na tym etapie algorytmu nie ma szans być poprawiony i dalsza analiza jest skazana na porażkę. Skuteczność tych metod jest opisana w rozdziale 5.

W rozdziale 4 są opisane uwagi implementacyjne, głównie wartości parametrów opisywanych przy algorytmach. Ustalanie różnych konfiguracji tych wartości może prowadzić do różnych taktyk odrzucania wierzchołków. Przykładowo, użytkownik może zdecydować się przepuścić więcej kandydatów przez pierwsze sító, i zatrzymać ich w drugim. Różne podejścia mogą być też zastosowane wewnątrz algorytmów. Przykładem może być trzeci algorytm – można zdecydować się na obniżenie wartości progu, co spowoduje zaznaczenie większej liczby pikseli na krawędziach i jednocześnie zwiększenie tolerancji na zaznaczone piksele wewnątrz sprawdzanych wierzchołków.

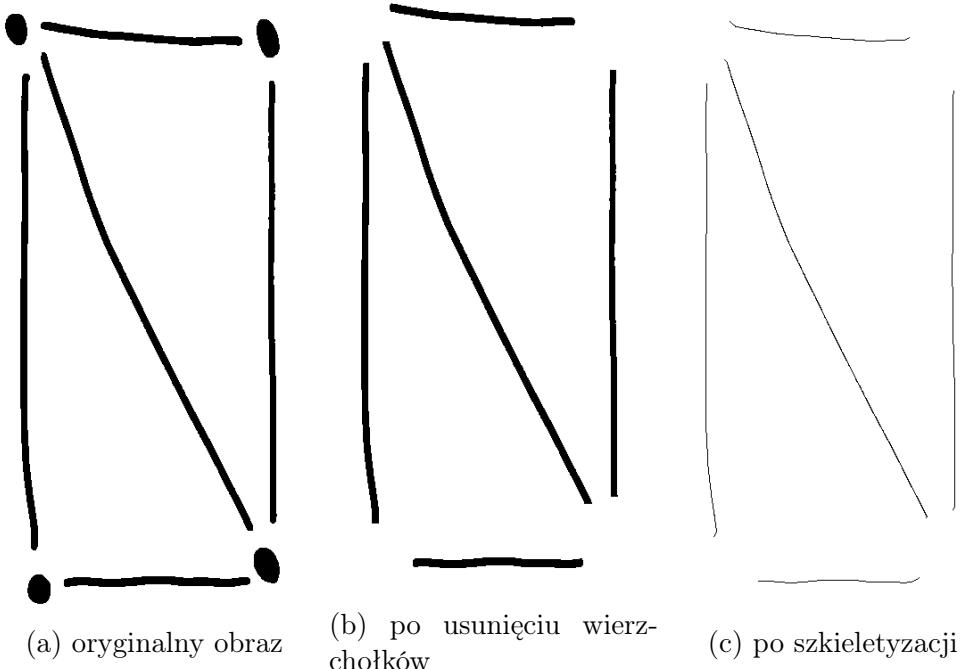
3.2 Rozpoznawanie krawędzi

Poprzedni rozdział dostarczył bardzo dobrych narzędzi do odróżnienia krawędzi od reszty obrazu. Metoda Harrisa i przefiltrowanie jej wyników zaznaczyły je. Nie można jednak polegać na jej wyniku, ponieważ jest on bardzo podatny na jakikolwiek pomyłki w rysowaniu odręcznym. Wystarczy dodatkowe pociągnięcie długopisu lub pisaka, które wprowadza nieregularność, żeby algorytm zaznaczył w takim miejscu róg. Należy pamiętać, że sprawdza on jedynie czy intensywność obrazu zmienia się wzdułż krawędzi. Krawędzie będą zatem odczytywane bezpośrednio z *prepared_image* – obrazu zwróconego przez proces normalizacji. Na tym etapie algorytmu przyjęte jest, że wszystkie wierzchołki są zaznaczone poprawnie. Jeżeli zostaną odjęte od oryginalnego obrazu, w rezultacie powinny zostać tylko krawędzie. Nie jest to nigdy w całości pewne, ponieważ może się zdarzyć, że wierzchołek będzie mieć bardzo nieregularny kształt i nie jest znane dokładne miejsce, w którym się kończy, a zaczyna krawędź wychodząca z niego. Ten fakt nie przeszkodzi jednak w poprawnym wykrywaniu połączeń pomiędzy nimi.

Pierwszym krokiem jest więc odjęcie wierzchołków od obrazu. Wierzchołkami nazywamy zbiory (x, y, r) , a więc w założeniu są one kołami. Ponieważ kształty wierzchołków często nie są idealnym kołem, dobrze jest dla precyzji wyczyścić również niewielką powierzchnię wokół wierzchołków.

Krawędzie grafu nie są cienkimi liniami, które jednoznacznie wyznaczają kierunek, w którym się poruszają. Są czasami obiektami o dwucyfrowej szerokości (biorąc za jednostkę piksele). Bardzo dobrze pokazuje to rysunek 3.1b, gdzie każda krawędź grafu ma swoje własne krawędzie. Taka sytuacja utrudnia analizę ich długości oraz kierunku. Człowiek nie dostrzega tego problemu, ponieważ automatycznie śledzi wzrokiem środek krawędzi. Komputer natomiast dostaje zestaw pikseli i brak wiedzy na temat krzywizny uniemożliwia poprawne poruszanie się po niej. Taki problem nie występuje, gdy krawędź jest grubości 1 – jak na przykład rezultat algorytmu Canny. Wtedy znane są sposoby na ocenę kierunku oraz zakrzywienia krawędzi, jak na przykład wynik operatoru Sobela [7, str. 122]. Kolejnym problemem jest ocena wzajemnego położenia krawędzi i wierzchołka. Jest to niezbędne do ustalenia, które wierzchołki są połączone. Bez wątpienia

staje się to łatwiejsze w momencie, gdy każda krawędź jest reprezentowana przez cienką linię.



Rysunek 3.13: Proces usunięcia wierzchołków oraz szkieletyzacji obrazu

Widać więc potrzebę „odchudzenia” lub inaczej „szkieletyzacji” krawędzi, czyli przekształcenia jej do postaci, w której będzie miała grubość 1, ale zachowana zostanie jej krzywizna. Istnieją różne podejścia do tego zagadnienia [4, str. 244]. W tej pracy będzie wykorzystany algorytm opublikowany przez T.Y. Zhang oraz C.Y. Suen [12] ze względu na jego szybkość (nazwany dalej *thinning*), wysoką precyżję oraz możliwość bezpośredniego zrównoleglenia. Wynikiem tego algorytmu są linie, które są środkami odchudzanego kształtu. Ten fakt zachodzi dzięki skutecznemu wymazywaniu brzegów figur.

Wykonane zostały więc dwie operacje: usunięcie wierzchołków oraz szkieletyzacja obrazu. Przykład ich działania pokazuje rysunek 3.13.

$$\text{edges} := \text{prepared_image} - \text{vertices}$$

$$\text{thin}_e\text{dges} := \text{thinning}(\text{edges})$$

Kolejną niezbędną operacją jest wyznaczanie wektora kierunkowego końca krawędzi. Pomysł jest następujący: jest to wektor, który zawiera informację, w którym kierunku jest zwrócony koniec krawędzi (dla każdej krawędzi będą istniały zatem dwa takie wektory).

Definicja 3.2.1. *edge_direction_vector* Wektor przyłożony do końca krawędzi, mierzący kierunek jej nachylenia

$$\begin{aligned} \text{edge_direction_vector}(\text{edge}, \text{edge_end}) := \\ \text{edge}[\text{edge_end}] - \text{edge}[\text{edge_end} + \text{edge_end} \cdot \text{vector_size}] \end{aligned}$$

gdzie $edge_end$ jest równe 1, gdy sprawdzamy początek krawędzi i -1, gdy koniec.

Ze względu na specyficzny rezultat algorytmu *thinning* implementacja tego wektora posiada pewne pułapki. Warto zobaczyć rozdział 4.2 w celu poznania szczegółów.

Należy teraz odpowiedzieć na pytanie: które wierzchołki są ze sobą połączone? Ten problem można rozbić na dwa zagadnienia:

- przy którym wierzchołku kończy się (lub zaczyna) dana krawędź
- w przypadku przecięcia krawędzi, które krawędzie się ze sobą łączą, a które nie

Nawet bez rozwiązania drugiego punktu, całość problemu będzie rozwiązana dla klasy grafów narysowanych planarnie. Dla ułatwienia, na tym etapie, czynione jest więc założenie planarności grafu.

Algorytm Create Edges List

Wejście $thin_edges$ – wynik algorytmu szkieletyzacji

Wyjście $edges_list$ – lista krawędzi

$kernel = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 10 & 1 \\ 1 & 1 & 1 \end{bmatrix}$

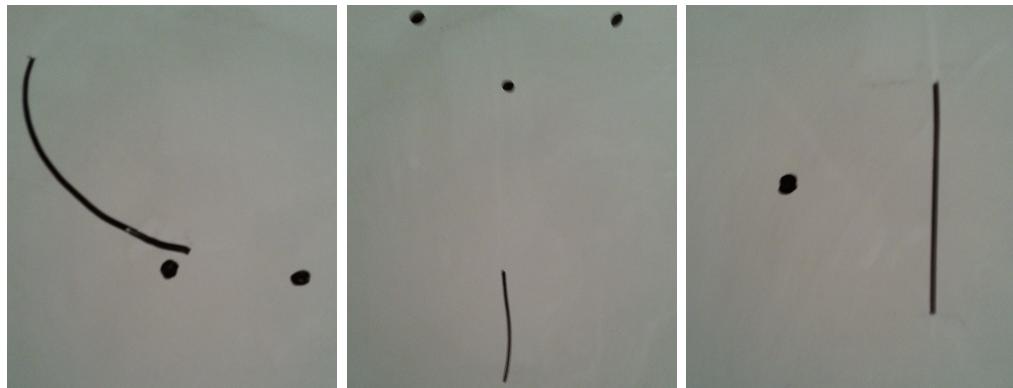
$neighbours_count = convolve(thin_edges, kernel)$
 $edges_ends = (x, y) : neighbours_count[x, y] == 11$ – piksele, które mają 1 sąsiada są końcówkami krawędzi
 $visited = []$
 $edges_list = []$
for (x, y) w $edges_ends$ **do**
 $edge = [(x, y)]$
 while (x, y) ma sąsiada $(u, v) : (u, v)$ nie ma w $visited$ **do**
 dodaj (u, v) do $visited$
 dodaj (u, v) do $edge$
 $(x, y) = (u, v)$
 end while
 dodaj $edge$ do $edges_list$
end for
return $edges_list$

Przed rozwiązaniem ich obu należy wykonać pewne przygotowanie: podział pikseli na krawędzie. Dla człowieka jest on naturalny, lecz w celu wygodnego operowania na krawędziach, potrzebna jest struktura danych trzymająca współrzędne pikseli na każdej krawędzi. Algorytm jest ideologicznie bardzo prosty, lecz posiada pewne pułapki techniczne. Zostanie więc tylko podany jego pseudokod (**Create Edges List**), a szczegóły opisane w rozdziale 4.

Podrozdziały 3.2.1 i 3.2.2 prowadzą do dwóch algorytmów, które zwieńczają działanie całej metody opisanej w pracy – algorytmów zwracających listy krawędzi grafu. Pierwszy z nich działa na grafach planarnych. Może być użyty także

na grafie nieplanarnym, lecz zwróci on tylko wierzchołki połączone przez krawędzie, które nie przecinają się z żadną inną. Drugi, opisany w podrozdziale 3.2.2, działa także na grafach nieplanarnych. Jest rozszerzeniem algorytmu planarnego. Oznacza to, że wystarcza on całkowicie do rozwiązania całego problemu. Jednak dla klarowności i łatwiejszego zrozumienia, przypadki te zostały rozdzielone.

3.2.1 Grafy planarne



(a) krawędź nie jest naj- (b) krawędź jest daleko od (c) krawędź nie prowadzi
bliżej wierzchołka wierzchołka do wierzchołka

Rysunek 3.14: Przykłady krawędzi niepasujących do wierzchołków

Posiadając struktury: *edges_list* oraz *vertices* można przystąpić do analizy połączeń pomiędzy wierzchołkami. Jest ona przeprowadzona w następujący sposób: dla każdego końca krawędzi jest ustalany wierzchołek, do którego ten koniec prowadzi. Jest to oceniane za pomocą trzech kryteriów:

- a) czy dany wierzchołek znajduje się najbliżej końca krawędzi
- b) czy koniec krawędzi jest dostatecznie blisko
- c) czy krawędź prowadzi w stronę wierzchołka

Pierwszy i drugi warunek nie są tożsame – jeżeli jakaś krawędź znajduje się daleko od wszystkich wierzchołków, to nie jest przypisana do żadnego, nawet do najbliższego. Rysunek 3.14 prezentuje przykłady konieczności istnienia każdego z tych warunków. W żadnym z przypadków nie było intencją rysującego przypisanie krawędzi do wierzchołka, który spełniał dwa z tych warunków, a nie spełniał trzeciego. Jak pokazują testy (rozdział 5), skuteczność kryteriów a, b, c jest bardzo wysoka. Można więc wyodrębnić procedurę **Edge Match Vertex**, która dla krawędzi zwraca wierzchołek, do którego ta krawędź prowadzi lub *None*, jeżeli krawędź nie prowadzi do żadnego wierzchołka. Ze względu na bardzo wysoko poziomowy zapis tego algorytmu warto zobaczyć w rozdziale 4 jego szczegóły implementacyjne.

Są zatem przygotowane wszystkie narzędzia potrzebne do zapisania algorytmu **Connected Vertices**, zwracającego listę krawędzi grafu planarnego. Zwraca on strukturę danych powszechnie znaną jako lista krawędzi.

Algorytm Edge Match Vertex

Wejście $vertices, edge$ – element listy, $edges_list, end - 1$: czy sprawdzany jest początek krawędzi lub -1: koniec krawędzi

Wyjście $vertex$ – wierzchołek, do którego prowadzi dany koniec krawędzi

```
if  $end == 0$  then
     $edge\_end :=$  początek krawędzi
else
     $edge\_end :=$  koniec krawędzi
end if
for  $v$  należący do  $vertices$  do
    if  $v$  nie jest najbliższym wierzchołkiem  $edge\_end$  then
        continue
    else if  $v$  i  $edge\_end$  nie znajdują się dostatecznie blisko siebie then
        continue
    else if  $edge\_direction\_vector(edge, edge\_end)$  nie prowadzi w stronę wierzchołka  $then$ 
        continue
    end if
    return  $v$ 
end for
return  $None$ 
```

3.2.2 Grafy nieplanarne

Problem przecięć jest rozwiązywany bardzo podobnie do połączeń krawędzi z wierzchołkami. W pierwszym kroku, miejsca przecięć krawędzi zostają wymazane z obrazu. Następnie dla każdej krawędzi (która wcześniej nie połączyla dwóch wierzchołków) wyznaczane jest jej przedłużenie, czyli inna krawędź, która najbardziej do niej pasuje. Jest więc potrzebna miara, która będzie oceniać jak bardzo dwie krawędzie są do siebie podobne. Jest ona wyznaczana na podstawie bardzo podobnych kryteriów, które wcześniej zostały użyte do wierzchołków:

- odległość końców krawędzi,

Algorytm Connected Vertices

Wejście $vertices, edges_list$

Wyjście $connected_vertices$ – lista połączonych ze sobą wierzchołków

```
connected_vertices = []
for  $edge$  należąca do  $edges\_list$  do
     $v := edge\_match\_vertex(vertices, edge, -1)$ 
     $v' := edge\_match\_vertex(vertices, edge, 1)$ 
    if  $v \neq None$  oraz  $v' \neq None$  then
        dodaj  $(v, v')$  do  $connected\_vertices$ 
    end if
end for
return  $connected\_vertices$ 
```

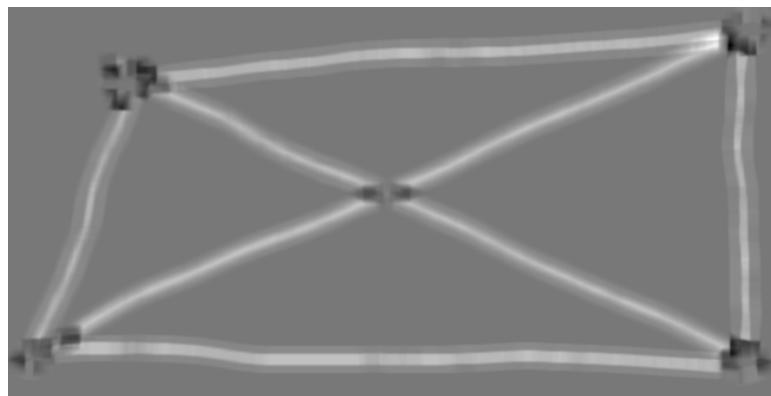
- kierunek końców krawędzi (czy są zwrócone w tę samą stronę).

Widoczna jest więc narzucona odgórnie, reguła podążania przecinającymi się krawędziami: ten sam kierunek krawędzi.

Wchodząc bardziej w szczegóły, można wyodrębnić fazy tego algorytmu:

1. Lokalizacja przecięć

Zaskakująco, ten element został już rozwiązyany wcześniej. Algorytm Harrisa oceniał i lokalizował wszystkie miejsca będące rogami obrazu. Okazuje się, że ta charakterystyka idealnie pasuje do przecięć krawędzi. Aby się o tym przekonać, należy jeszcze raz zobaczyć rysunek 3.8 z rozdziału 3.1.4. Cecha zmieniania się obrazu w każdym kierunku przesuwania okna jest



Rysunek 3.15: funkcja Harrisa

bardzo dobrym opisem szukanych regionów i funkcja Harrisa wyszukuje je z ogromną skutecznością. Dla przypomnienia – rogi obrazu miały najwyższą wartość funkcji R , zdecydowanie wyższą od tła i krawędzi. W celu ich wyznaczenia wystarczy więc zastosować metodę progowania. Szczegółły metody i parametry są opisane w rozdziale 4.2.

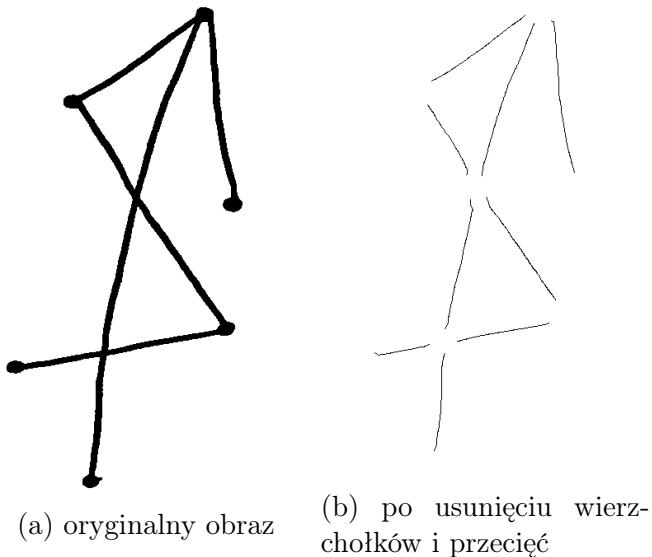
2. Wymazanie przecięć

Miejsca zaznaczone w poprzednim punkcie są odejmowane od *thin_edges* – obrazu po szkieletyzacji. Wynik tej operacji przedstawia rysunek 3.16 Poza wierzchołkami, zostały wymazane także przecięcia krawędzi.

3. Połączenie pasujących do siebie krawędzi

Dla każdej krawędzi, która nie łączy dwóch wierzchołków, szukana jest jej para, która może być jej przedłużeniem. W tym celu wyznaczana jest funkcja, która każdej parze przypisuje skalę ich podobieństwa:

$$\begin{aligned} \text{edges_match_score}(\text{edge}_1, \text{end}_1, \text{edge}_2, \text{end}_2) := \\ \text{distance}(\text{edge}_1[\text{end}_1], \text{edge}_2[\text{end}_2]) - \\ \text{distance}(\text{edge}_1[\text{end}_1] + \text{edge_direction_vector}(\text{edge}_1, \text{end}_1), \\ \text{edge}_2[\text{end}_2] + \text{edge_direction_vector}(\text{edge}_2, \text{end}_2)) \end{aligned}$$



Rysunek 3.16: Usunięcie wszystkich rogów obrazu

Wyznacza ona, jak szybko zbliżają się do siebie dwie krawędzie, gdy przedłużane są ich końce. Tak samo, jak w przypadku wierzchołków, eliminowane są pary, które leżą bardzo daleko od siebie. Każda para krawędzi (e_i, e_j) jest łączona ze sobą pod warunkiem, że $\text{edges_match_score}(\text{edge}_i, \text{end}_i, \text{edge}_j, \text{end}_j)$ jest najwyższym wynikiem dla i oraz $\text{edges_match_score}(\text{edge}_j, \text{end}_j, \text{edge}_i, \text{end}_i)$ jest najwyższym wynikiem dla j . Liczby $\text{end}_i, \text{end}_j$ są oznaczeniami, które końce krawędzi są aktualnie brane pod uwagę. Ich konkretne wartości wynikają z przebiegu algorytmu.

Algorytm **Connected Edges** jest ostatnim krokiem opisywanym w tej pracy. Na wejściu bierze strukturę `edges_list` wyznaczoną algorytmem **Create Edges List**. Należy pamiętać, że tym razem jest ona wyznaczona na obrazie z odjętymi rogami. Algorytm Connected Edges zwraca listę krawędzi dla każdego grafu. Podąża on łączącymi się krawędziami. Gdy w końcu któraś z nich będzie przylegała do wierzchołka, zapisze połączenie. Dodanie do siebie wyników tego algorytmu oraz `connected_vertices`) jest zwracaną przez algorytm listą krawędzi. Zgodnie ze specyfikacją określoną na początku, wyjściowym formatem jest DOT. Konwersja listy krawędzi na plik DOT jest czynnością czysto techniczną.

Algorytm Connected Edges

Wejście $vertices, edges_list$

Wyjście $connected_vertices$ – lista połączonych ze sobą wierzchołków

```
connected_vertices = []
for edge należąca do edges_list nie wykorzystana wcześniej do
    v, edge_end := edge_match_vertex(vertices, edge, -1), -1
    if v == None then
        v, edge_end := edge_match_vertex(vertices, edge, 1), 1
    end if
    if v == None then
        continue
    end if
    while True do
```

$$edge_1, edge_end_1 = \operatorname{argmax}_{e \neq edge, end \in \{-1, 1\}} \\ (\operatorname{edges_match_score}(edge, (-1) \cdot edge_end, e, end))$$

```
v' = edge_match_vertex(vertices, edge, (-1) \cdot edge_end)
if v' \neq None then
    dodaj (v, v') do connected_vertices
    break
end if
```

```
edge_2, _ = \operatorname{argmax}_{e \neq edge_1, end \in \{-1, 1\}} \\ (\operatorname{edges\_match\_score}(edge_1, edge\_end_1, e, end))

if edge_2 == edge then
    edge = edge_1
    edge_end = (-1) \cdot edge_end_1
else
    break
end if
end while
end for
return connected_vertices
```

Rozdział 4

Uwagi i implementacja

Ten rozdział jest poświęcony osobom, które interesują się implementacją przedstawionej metody lub jej doskonaleniem. Są omawiane dokładne wartości, jakie przyjmują różne parametry algorytmu oraz szczegóły implementacji, które mogą wydawać się zaskakujące podczas czytania kodu źródłowego, ale z różnych względów nie zostały uwzględnione w opisie algorytmu. Głównym powodem był ich brak wpływu na ideę rozumowania, ale jednak są na tyle ważne, że bez nich skuteczność rozpoznawania grafów byłaby dużo niższa.

Należy zaznaczyć, że w wielu przypadkach wartości parametrów zostały wyznaczone eksperymentalnie. Może to wzbudzić wątpliwości co do generalizacji metody. W celu uniknięcia tego problemu praca odbywała się podejściem znanym z uczenia maszynowego – dane zostały podzielone na zbiór treningowy i testowy. Eksperymenty były przeprowadzane tylko na zbiorze treningowym. Wyniki testów są rezultatami uzyskanymi na zbiorze testowym.Więcej na ten temat znajduje się w rozdziale 5.

4.1 Rozpoznawanie wierzchołków

Algorytm Canny przyjmuje kilka parametrów: dolny i górny zakres progowania oraz wielkość jądra konwolucji używanej przez operator Sobela. Operator Sobela standardowo działa z jądrem wielkości 3 i także z takim sprawdza się w tym przypadku. Natomiast wartości progowania są płynne. Zależnie od tego, jak bardzo mocne lub słabe krawędzie chcemy wykrywać, możemy zmieniać obie wartości. Przy ich wyznaczaniu można się posilić metodą progowania Otsu [13]. Canny rekomenduje ustawienie wartości dolnej jako $1/3\max_threshold$. W przypadku tej pracy algorytm jest wykonywany na obrazie binarnym, więc każda krawędź jest idealnie zaznaczona. Wartości używane w implementacji wynoszą: 100 i 200.

Podczas liczenia grubości linii na obrazie ważnym elementem jest długość wektora (sąsiedztwa danego piksela wzdłuż krawędzi), od którego liczymy wektor prostopadły. Z jednej strony ta wartość nie może być duża, żeby liczyć rzeczywisty, lokalny kierunek tego wektora. Z drugiej wzięcie zbyt małej wartości naraża na bycie podatnym na niedoskonałości obrazu oraz normalizacji. Używaną wartością jest 5, czyli po dwóch sąsiadów branych z przodu i z tyłu. Należy również pamiętać, że będzie wiele punktów, które nie będą miały, z różnych przyczyn,

sąsiedniej krawędzi. Szukanie należy więc przerwać po pewnej liczbie iteracji.

Metoda Laplacian of Gaussian wyznacza ekstremum funkcji. Daje ona wysokie wartości funkcji Laplace'a dla skupisk o dużej intensywności oraz niskie, ale wciąż będące lokalnym maksimum, dla skupisk o mniejszej intensywności. Można regulować akceptowalną intensywność poprzez parametr progu (threshold). Gdy do rysowania obrazu użytą jest duża grubość linii, a wierzchołki nie są mocno wyróżniające się, wynik może zawierać wiele fałszywych blobów, znalezionych na krawędziach grafu. W łatwy sposób można odrzucić wiele z takich przypadków, używając większego progu. Konkretne wartości uzależnione są od implementacji algorytmu Laplacian of Gaussian.

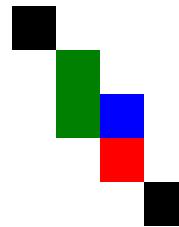
Algorytm **Circular Kernel Convolution** jest algorytmem prostym ideologicznie: akceptuje miejsca, które mają wokół siebie dużo czarnych pikseli, odrzuca inne. Pozostaje kwestia ustalenia parametrów: wielkości koła, jakie sprawdzamy i tolerancji na liczbę białych pikseli. Jak widać na implementacji, pierwsza wartość jest uzależniona od grubości linii na rysunku. Należy więc odpowiedzieć na pytanie: o ile grubsze powinny być wierzchołki od krawędzi? Do testów końcowych został użyty parametr $kernel_size_ratio = 2.5$. Podczas prób z różnymi konfiguracjami, rozsądne wyniki dawały różne wartości pomiędzy 2 a 3. Kolejny parametr jest wyznaczony na $acceptance_ratio = 0.98$. Obrazuje on mniejszą tolerancję na niedoskonałość obrazu. Co ciekawe, algorytm bardzo źle działa, gdy jest on pozostawiony na 1, a zmiany od 0.9 do 0.99 powodują niewielkie różnice w działaniu całego algorytmu. Tak jak w przypadku wielu innych parametrów, te wartości zostały wyznaczone na podstawie eksperymentów.

Metoda Harrisa nie jest niezależna od skali obrazu. To, co w małej skali wydaje się krawędzią, w dużej może wydawać się rogiem. Ważne jest w takim razie ustalenie skali, co w przypadku tego algorytmu jest reprezentowane przez wielkość sąsiedztwa każdego piksela – w . Tutaj znów z pomocą przychodzi ustalona wcześniej grubość linii obrazu – $line_width$. Idealną sytuacją byłoby, gdyby cała grubość krawędzi grafu była objęta przez okno sąsiedztwa. Konsekwencją wybrania zbyt małego rozmiaru okna, będzie zaznaczenie wielu krawędzi grafu na wierzchołkach. Miejsca styku wierzchołka z tłem (krawędzie wierzchołków) będą traktowane jak krawędzie grafu – będzie istniał niewielki lokalny wektor, wzduż którego intensywność obrazu nie będzie się zmieniać. Z drugiej strony wybór zbyt dużego okna może doprowadzić do sytuacji, w której krawędź, która nie będzie prosta, tylko zakrzywiona, będzie uznana za róg. Rozmiar okna został więc wyznaczony na $w = 2 \cdot line_width + 3$. Ustalenie go na niewiele ponad $2 \cdot line_width$ daje bardzo dobre rezultaty w przypadku, gdy odfiltrowane zostały wcześniej wierzchołki mniejsze niż $2 \cdot line_width$. Pozostaje jeszcze parametr $edge_acceptance_ratio$. Jego przeznaczenie jest bliższe do $acceptance_ratio$ z poprzedniego algorytmu. Jego wartość wynosi 0.09.

4.2 Rozpoznawanie krawędzi

Algorytm budowania listy krawędzi `create_edges_list` można podsumować:

- znajdź końce krawędzi



Rysunek 4.1: przybliżona krawędź grafu

- podążaj kolejnymi sąsiadami pikseli dopóki są

Bardzo często jednak zdarza się, że piksel ma więcej niż jednego nieodwiedzonego sąsiada. Przykład na rysunku 4.1, piksel zaznaczony na niebiesko ma jednego sąsiada odwiedzonego – czerwonego, i dwóch nieodwiedzonych – zielonych. Taka sytuacja jest dodatkowo niefortunna, gdy jest na końcu krawędzi. Wtedy należy także zmienić działanie algorytmu znajdowania końców. Można sobie poradzić z tym na dwa sposoby:

- po wykonaniu algorytmu szkieletyzacji, zrobić jeszcze jedną iterację, która wyszukuje dokładnie takie sytuacje i usunąć te piksele (takie usunięcie nie rozspójnia krawędzi)
- podczas szukania sąsiadów, najpierw szukać piksele w linii prostej, a w drugiej kolejności po skosie

Wyznaczanie wektora kierunkowego – Kierunek tego wektora ma odwzorowywać, w którą stronę skierowana jest końówka krawędzi. Jest on więc wyznaczany jako różnica dwóch punktów na krawędzi: jednego tuż przy końcu i drugiego, który znajduje się niewiele wcześniej na krawędzi. Uwaga, pierwszy z tych punktów nie może być końówką krawędzi. Wynika to ze specyficznego działania algorytmu *thinning*. Na rysunku 3.13 można zauważyc, że końówki cienkich krawędzi nie obrazują ich kierunku. Iteracyjne odejmowanie kolejnych pikseli prowadzi do sytuacji, że końce podłużnych kształtów są zakrzywione. Niedobędne jest więc pozostawienie pewnej liczby punktów na krawędzi i mierzenie jej kierunku od tego miejsca. Ostatecznie, wyznaczanie omawianego wektora wygląda w następujący sposób:

$$\text{direction_vector} := \text{edge}[\text{spare_points}] - \text{edge}[\text{spare_points} + \text{vector_size}]$$

Parametr *spare_points* określa, ile punktów na krawędzi należy pominąć, żeby specyficzne działanie algorytmu *thinning* nie wpłynęło błędnie na ocenę kierunku. Pozostawienie dziesięciu punktów w zupełności wystarcza. Wartość *vector_size* specyfikuje, na jakim odcinku krawędzi ma zostać zmierzony kierunek. Ta wartość musi zbalansować lokalność pomiaru (ponieważ osoba rysująca może inaczej zakrzywić końówkę krawędzi niż resztę) oraz niedokładność wynikającą z podziału obrazu na piksele (wzięcie bardzo małej liczby – na przykład 2 spowoduje zmierzenie względnego położenia tylko 2 pikseli, co bardzo zawęzi wiarygodność pomiaru). W implementacji została także użyta wartość 10.

Algorytm Edge Match Vertex składa się z 3 kroków:

- sprawdzenie, czy dany wierzchołek znajduje się najbliżej końca krawędzi
- sprawdzenie, czy wierzchołek i koniec krawędzi są blisko siebie
- sprawdzenie, czy krawędź prowadzi w stronę wierzchołka

Implementacja pierwszego z tych punktów jest trywialna. Przy drugim punkcie jest niewątpliwa potrzeba parametru, który określałby, jak daleko od siebie mogą być obiekty, żeby zostały uznane za sąsiadujące ze sobą. Trzeba przyznać, że jest to cecha bardzo mocno zależna od stylu rysowania. Bardzo łatwo jest więc zdyskredytować poprawnie narysowane połączenie zbyt niską wartością takiego parametru. Z tego powodu ten punkt ma na celu eliminację przypadków bardzo zdegenerowanych, gdzie krawędź znajduje się naprawdę daleko. Jeżeli na obrazie nie ma obiektów narysowanych celowo błędnie dla przetestowania algorytmu lub szumów, które nie zostały usunięte w poprzednich etapach, ten punkt bardzo rzadko odrzuca jakikolwiek wierzchołek. Sprawdzenie skierowania krawędzi w stronę wierzchołka zostało częściowo opisane w algorytmie – wektor kierunkowy danego końca krawędzi jest przykładowy do jej końca. Następnie jest sprawdzane, czy koniec tego wektora znajduje się bliżej wierzchołka niż koniec krawędzi. Sprawdzenie, czy krawędź prowadzi w stronę wierzchołka, staje się sprawdzeniem nierówności:

$$distance(v, edge[0]) > distance(v, edge[0] + direction_vector)$$

Rozdział 5

Ewaluacja metody

Algorytm jest ewaluowany na 70-ciu grafach rysowanych odręcznie przez różne osoby. Sumarycznie liczba wierzchołków w tych grafach wynosi 409, a krawędzi 422. Wszystkie zostały narysowane zgodnie z kryteriami zaprezentowanymi w rozdziale 1.2, t.j.:

- wierzchołki są wyraźnie grubsze od krawędzi, wypełnione,
- krawędzie są liniami ciągłymi.

Program w żadnym momencie nie używa losowych wartości. Oznacza to, że wyniki testów nie będą się w ogóle różniły przy kolejnych wywołaniach.

Poza wynikami testów zaprezentowane są też przypadki, które z różnych powodów zostały rozpoznane błędnie. Część z nich jest „świadomie” uznana za błędne, co oznacza, że autor brał pod uwagę, że obecna budowa algorytmu będzie prowadziła do niepoprawnego rozpoznania. Taka decyzja była podejmowana najczęściej z powodu bardzo dobrej skuteczności w wielu innych przypadkach. Istnieją jednak przykłady, które są zaklasyfikowane błędnie pomimo starań, żeby dany styl był rozpoznany poprawnie. Obie te sytuacje pokazują potrzebę dalszego rozwijania algorytmu.

Dla ułatwienia śledzenia wyników, poza formatem DOT, program generuje graficzną reprezentację rozpoznanego grafu. Są na niej zaznaczone wierzchołki na pozycjach, na których zostały rozpoznane oraz linie proste pomiędzy tymi, które zostały uznane za połączone przez krawędź.

5.1 Wyniki testów

rozmiar boku	skuteczność	średni czas działania
400px	47%	5.6s
600px	65%	8.8s
800px	61%	13.5s

Tabela 5.1

Najprostszą miarą skuteczności, jaką można przyjąć do ewaluacji, jest procent grafów rozpoznanych idealnie. Oznacza to, że graf wygenerowany przez program jest izomorficzny z grafem użyтыm do testu. Rozmiar obrazów jest zmieniany na samym początku działania programu. Tabela 5.1 prezentuje procent poprawnie rozpoznanych grafów oraz średni czas działania algorytmu dla różnych użytych rozmiarów boku obrazu. Niekoniecznie wyższa rozdzielcość obrazu wpływa pozytywnie na jego rozpoznanie. Zbyt duża szczegółowość prowadzi do mniej skutecznego wykrywania krawędzi oraz bardziej skomplikowanego kształtu wierzchołków.

Do dalszych testów został użyty rozmiar $600px \times 600px$.

5.1.1 Ewaluacja wykrywania wierzchołków

	błąd ujemny	błąd dodatni	liczba wierzchołków
grafy planarne	21	0	250
grafy nieplanarne	1	28	187
suma	22	28	437

Tabela 5.2: Skuteczność wykrywania wierzchołków

Zestaw testowy zawiera w sumie 437 wierzchołków. Błędne rozpoznanie któregokolwiek wierzchołka w grafie pociąga za sobą brak izomorficzności. Nie ma tutaj różnicy, czy algorytm pomylił się przy jednym z nich, czy przy wszystkich. Takie podejście nie daje pełnego obrazu działania algorytmu. W procesie powstawania algorytmu należy zwrócić uwagę, ile konkretnie obiektów jest poprawnie rozpoznanych. Występują dwa typy błędów:

- wierzchołek nierożpoznany – błąd ujemny
- wierzchołek rozpoznany w błędnych miejscu (w którym oryginalnie wierzchołka nie ma) – błąd dodatni.

Oba błędy są ze sobą mocno powiązane, ponieważ na różnych etapach algorytm balansuje pomiędzy zaakceptowaniem zbyt dużej lub zbyt małej ilości wierzchołków. Nie da się więc liczby wystąpień jednego z nich bez patrzenia na liczbę wystąpień drugiego.

Istnieje bardzo zauważalna różnica w liczebności obu błędów pomiędzy klasą grafów planarnych i nieplanarnych. Jest to spowodowane przecięciami krawędzi, które bardzo często generują nadmiarowy wierzchołek. Liczbę wystąpień błędów w zestawie testowym przedstawia tabela 5.2. Obrazuje ona dwie informacje: $\frac{22}{437} \sim 5\%$ wierzchołków nie zostało wykrytych oraz $\frac{28}{437-22+28} \sim 6\%$ wskazań było błędnych.

5.1.2 Ewaluacja wykrywania krawędzi

Wykrycie krawędzi musi być poprzedzone poprawnym wykryciem obu wierzchołków, które łączy. W przeciwnym wypadku krawędź nie jest oceniana, ponieważ

	mała krawędź	błędne połączenie	liczba krawędzi
grafy planarne	6	0	262
grafy nieplanarne	1	13	151
suma	7	13	413

Tabela 5.3: Skuteczność wykrywania krawędzi

jej połączenie z wierzchołkiem jest elementem jej poprawnego wykrycia. Typy błędów w rozpoznawaniu krawędzi różnią się od błędów podczas wykrywania wierzchołków. Nie ma problemów z wykrywaniem nadmiarowych krawędzi. Brak wykrycia istnieje tylko w jednym przypadku – gdy krawędź jest zbyt krótka. Głównym błędem jest natomiast niepoprawne połączenie krawędzi z innymi krawędziami i wierzchołkami.

Znowu istnieje wyraźny podział na grafy planarne i nieplanarne. W klasie grafów planarnych krawędzie są wykrywane prawie idealnie. Jedynym przypadkiem błędu jest przypadek opisany w rozdziale 5.2.4 – niedostateczna długość krawędzi. W ten sposób ominiętych zostało 6 krawędzi, co daje skuteczność ich wykrywania na poziomie $\frac{256}{262} \sim 98\%$. Należy zwrócić uwagę, że nie występuje w ogóle problem braku połączenia pomiędzy krawędzią a wierzchołkiem, co oznacza, że kryteria ich połączenia (rozdział 3.2.1) są bardzo skuteczne.

W klasie grafów nieplanarnych dodatkowo dochodzi problem przecięć, a więc poprawnego podążania krawędzią. Pełne wyniki występowania obu błędów prezentuje tabela 5.3. Sumarycznie w tej klasie krawędzie są wykrywane ze skutecznością $\sim 91\%$.

5.1.3 Liczba błędów na rysunku

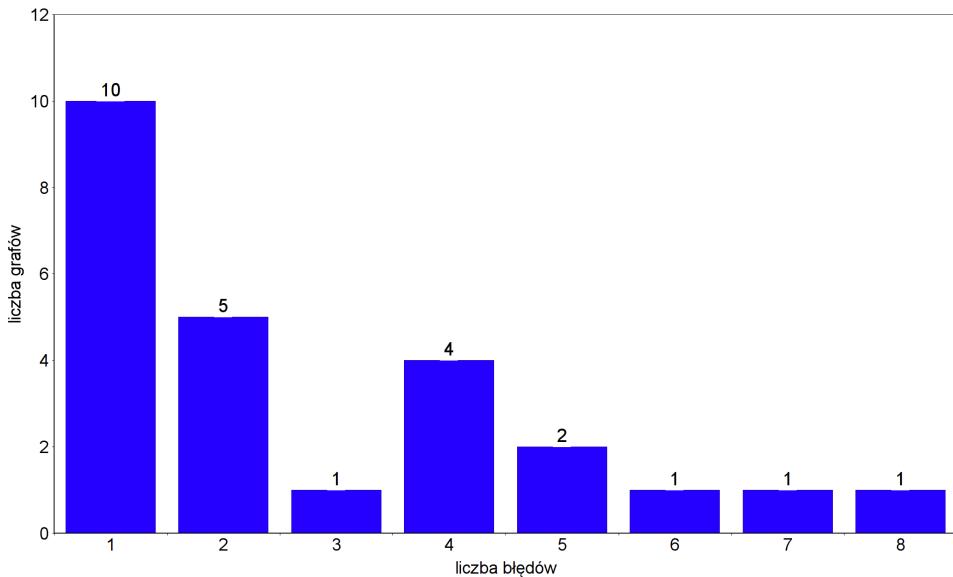


Diagram 5.4: Liczba przykładów z daną ilością błędów

Rozdziały 5.1.1 oraz 5.1.2 klasyfikują każdy błąd oraz zliczają sumaryczną liczbę ich wystąpień. Wobec tego interesującą informacją jest, ile błędów zawierał każdy z wygenerowanych grafów. Diagram 5.4 przedstawia, ile jest przykładów testowych, które wygenerowały daną ilość pomyłek. Wyraźnie widać, że bardzo wiele niepoprawnie wygenerowanych grafów zawiera zaledwie jeden lub dwa błędy. Jest to istotne z punktu widzenia użytkownika. Widząc rezultat działania algorytmu, który bardzo niewiele różni się od oryginału, może on dokonać szybkiej poprawki swojego rysunku, która sprawi, że graf zostanie wykryty poprawnie. Natomiast w sytuacji, w której graf wygenerowany nie przypomina w ogóle intencji użytkownika, szybko zniechęci się on do dalszego korzystania z programu.

maksymalna liczba błędów	skuteczność
0	65%
1	79%
2	86%
3	87%
4	93%
5	96%
6	97%
7	98%
8	100%

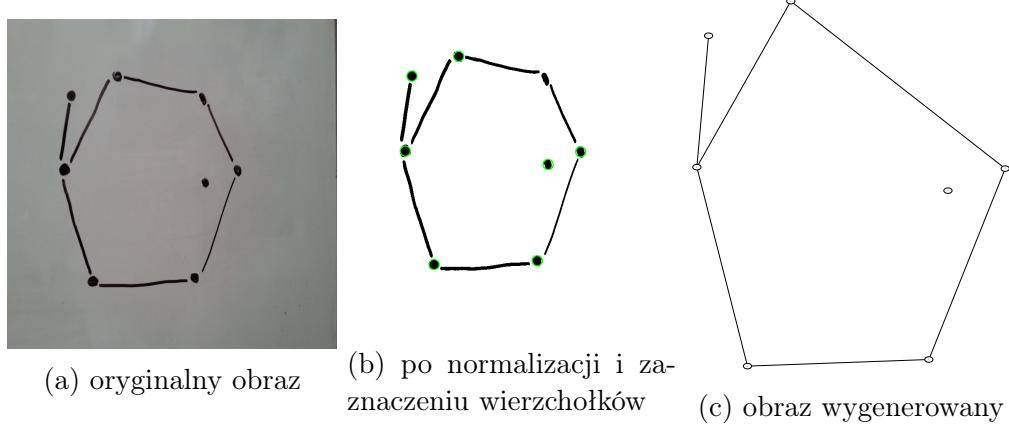
Tabela 5.5

Tabela 5.5 podsumowuje informacje z diagramu 5.4. Przedstawia jaką skuteczność na zbiorze testowym osiąga algorytm, gdy dopuszczona jest różna liczba błędów.

5.2 Błędy algorytmu

5.2.1 Niewykryty wierzchołek

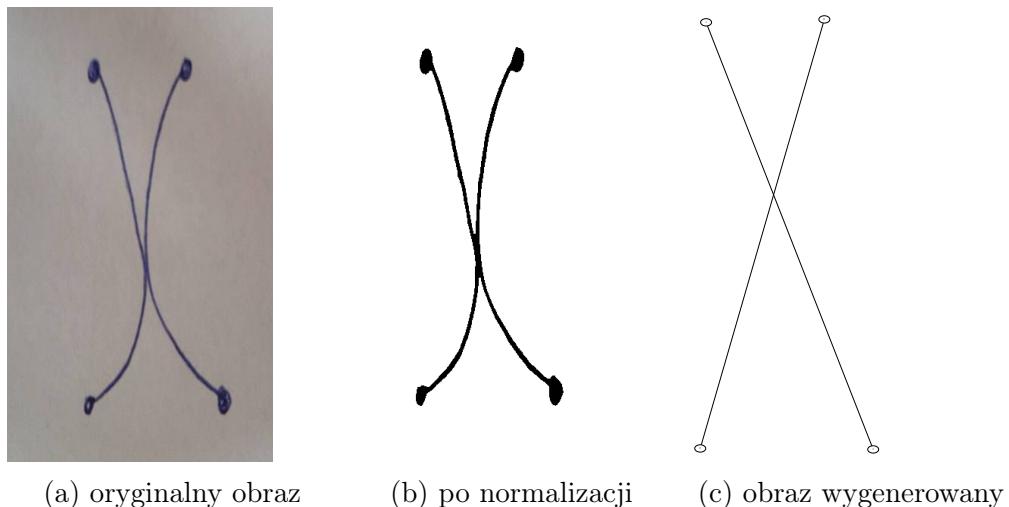
Przykład jednego z najczęściej spotykanego błędu algorytmu – jeden z wierzchołków nie został wykryty. Rysunek 5.6 przedstawia oryginalny obraz wraz z przekształceniami i ostatecznym wygenerowanym grafem. Wierzchołek, który na rysunku 5.6b nie jest zaznaczony zielonym okręgiem, nie został wykryty. Jest to błąd, który wynika z niewielkiej elastyczności metody. Wierzchołek został odrzucony, gdy został uznany za zbyt mały podczas użycia algorytmu Circular Kernel Convolution. Najprostszym rozwiązaniem dla tego konkretnego przypadku byłoby obniżenie stałej decydującej, która w tym algorytmie odpowiada za odrzucanie wierzchołków. To jednak spowodowałoby wiele nadmiarowych wskazań dla innych przykładów. Typ błędu, w którym jakiś obiekt nie zostaje wykryty, jest też łatwiejszy do poprawienia. Użytkownik, który jako rezultat dostaje grafy wygenerowany tak jak na rysunku 5.6c oraz widzi rysunek pomocniczy 5.6b, jest w



Rysunek 5.6: Niewykryty wierzchołek

stanie natychmiast poprawić swój rysunek, tak aby wszystko zostało poprawnie wykryte.

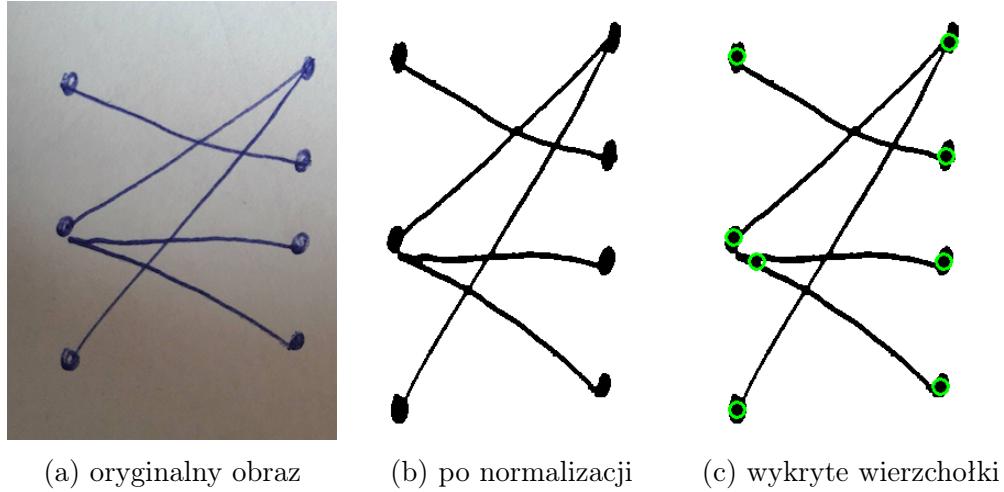
5.2.2 Błędnie połączone krawędzie



Rysunek 5.7: Błędnie połączone krawędzie

Bardzo ciekawym przykładem jest rysunek 5.7. Pokazuje on graf, w którym jego autor chciał połączyć wierzchołki zakrzywionymi krawędziami. Program jednak uznał, że krawędzie powinny się łączyć na wprost. Nie jest to zaskoczeniem, ponieważ tak zostało ustalone jedno z kryteriów połączeń krawędzi w miejscach przecięć (rozdział 3.2.2). Aby mieć możliwość rozpoznać poprawnie tego typu połączenia potrzebna jest wiedza na temat zakrzywienia krawędzi oraz zmiana powyższego kryterium. Ciekawym spostrzeżeniem jest, że ten graf jest zaliczany automatycznie jako poprawny. Wystarczy zwrócić uwagę, że jego niepoprawnie wygenerowana wersja jest izomorficzna z grafem narysowanym przez użytkownika.

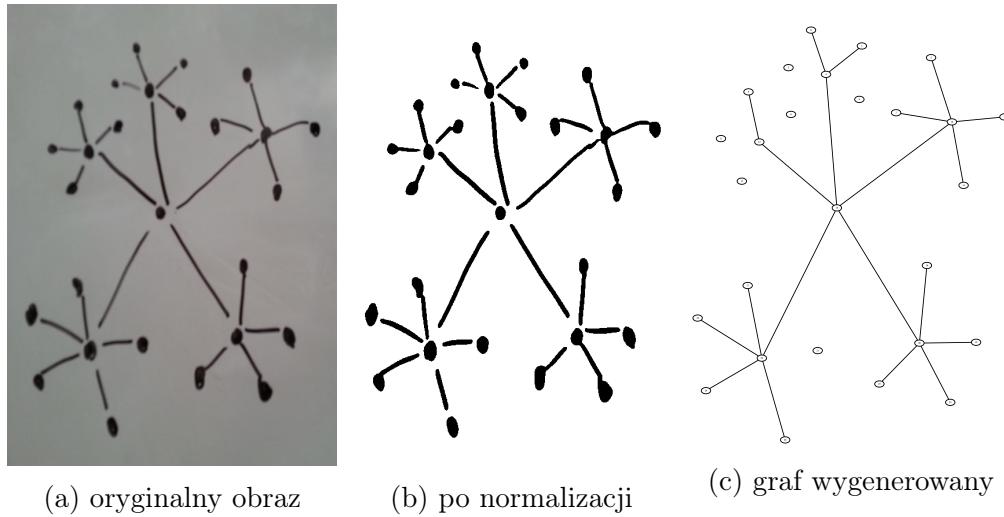
5.2.3 Wykryty nadmiarowy wierzchołek



Rysunek 5.8: Wykryty nadmiarowy wierzchołek

Rysunek 5.8 przedstawia przykład kolejnego uchybienia algorytmu. Został wykryty wierzchołek w miejscu, w którym go nie ma. Krawędź, na której leży jest grubsza od pozostałych. Takie przypadki są jednak eliminowane przez algorytm Circular Kernel Convolution for edges (rozdział 3.1.4). W tym miejscu jednak sprawę utrudnia fakt, że kandydat na wierzchołek znajduje się u zbiegu dwóch krawędzi. To powoduje, że ten region zaklasyfikowany jest jako róg obrazu (zresztą poprawnie). Nie jest więc uznany za krawędź i niepoprawny wierzchołek przedostał się przez ten filtr.

5.2.4 Zbyt krótka krawędź



Rysunek 5.9: Nie wykryte krótkie krawędzie

Ostatni przykład prezentuje kolejną wadę prezentowanego algorytmu. Na rysunku 5.9 można zauważyc, że nie zostało znalezionych kilka krawędzi w lewym

górny rogu grafu. Powodem tego jest jej długość. W rozdziale 3.2 jest opisany algorytm Edge Match Vertex, który między innymi wylicza wektory kierunkowe na obu końcach krawędzi. Jednak, żeby to zrobić, musiał najpierw zostawić ustaloną liczbę pikseli na końcu krawędzi (ze względu na działanie algorytmu *thinning* – patrz rozdział 4.2) i dopiero potem zmierzyć kierunek wektora. W efekcie powstaje ograniczenie dolne na długość krawędzi (jego konkretna wartość wynika ze stałych dobranych przy implementacji). Krawędzie, które nie zostały znalezione, nie spełniły tego ograniczenia.

Rozdział 6

Podsumowanie

Jak pokazują wyniki testów, przedstawiony algorytm jest narzędziem, które może być z powodzeniem wykorzystywane w programach przetwarzających odręczne narysowane grafy. Skuteczność na poziomie 65% dla grafów rozpoznanych idealnie oraz 86% dla maksymalnie dwóch błędów gwarantuje wygodę jego użytkowania. Wciąż niektóre elementy algorytmu wymagają dalszych usprawnień, zwłaszcza w przypadkach opisanych w rozdziale 5.2. Ogólnym kierunkiem, w jakim powinna podążyć dalsza praca nad zagadnieniem cyfryzacji grafów, jest lepsze dopasowanie do stylu rysującego. Aby móc dokładniej przetestować tego typu algorytmy, jest także potrzeba stworzenia bazy wielu ręcznie narysowanych grafów wraz z ich opisem.

Bibliografia

- [1] Christopher Auer, Christian Bachmaier, Franz J. Brandenburg, Andreas Gleissner, and Josef Reishuber. Optical graph recognition. In *Proceedings of the 20th International Conference on Graph Drawing*, GD'12, pages 529–540, Berlin, Heidelberg, 2013. Springer-Verlag.
- [2] T. Lindeberg. Feature detection with automatic scale selection. *Int. J. Comput. Vision*, 30(2):79–116, November 1998.
- [3] Weidong Huang, Seok-Hee Hong, and Peter Eades. Effects of crossing angles. In *IEEE VGTC Pacific Visualization Symposium 2008, PacificVis 2008, Kyoto, Japan, March 5-7, 2008*, pages 41–46, 2008.
- [4] E. R. Davies. *Computer and Machine Vision, Fourth Edition: Theory, Algorithms, Practicalities*. Academic Press, 4th edition, 2012.
- [5] S. Suzuki and K. Abe. *Topological structural analysis of digitized binary images by border following*. 1985.
- [6] G. Green. *An Essay on the Application of Mathematical Analysis to the Theories of Electricity and Magnetism*. author, 1828.
- [7] M. Nixon and A. S. Aguado. *Feature Extraction & Image Processing, Second Edition*. Academic Press, 2nd edition, 2008.
- [8] C. Harris and M. Stephens. A combined corner and edge detector. In *In Proc. of Fourth Alvey Vision Conference*, pages 147–151, 1988.
- [9] David G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, November 2004.
- [10] J Canny. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 8(6):679–698, June 1986.
- [11] H. P. Moravec. *Obstacle Avoidance and Navigation in the Real World by a Seeing Robot Rover*. PhD thesis, Stanford University, Stanford, CA, USA, 1980. AAI8024717.
- [12] T. Y. Zhang and C. Y. Suen. A fast parallel algorithm for thinning digital patterns. *Commun. ACM*, 27(3):236–239, March 1984.

- [13] M. Fang, G. Yue, and Q. Yu. The study on an application of otsu method in canny operator. In *In Proc. of the 2009 International Symposium on Information Processing*, 2009.