Wykrywanie kości do gry na zdjęciach

Witold Kupś, indeks 127088
Politechnika Poznańska, Wydział Informatyki, 2017

Spis treści

1. Wprowadzenie

Celem projektu jest wytworzenie oprogramowania, którego jedynym zadaniem jest wykrycie kości (dowolny kolor, dowolne tło) na zdjęciu (w dowolnej ilości) wykorzystując jedynie manipulacje na dostarczonych obrazach dowolnie je przetwarzając – krótko, bez zastosowań sztucznej inteligencji lub ingerencji zewnętrznej, jak `template matching`.

Strategia wykrywania jest prosta – początkowo poszukujemy samych kostek, następnie wyniki filtrujemy przez szereg porównań, po czym na odfiltrowanych kandydatach poszukujemy kropek. Warto jednak nadmienić, że w programie nie jest nigdzie zdefiniowane jak wygląda kostka – jest to po prostu obiekt wyróżniający się na tle otoczenia. Podobnie definiujemy kropkę, z tą różnicą, że tutaj przyjmujemy ogólne założenia co do jej rozmiarów w stosunku do kostki.

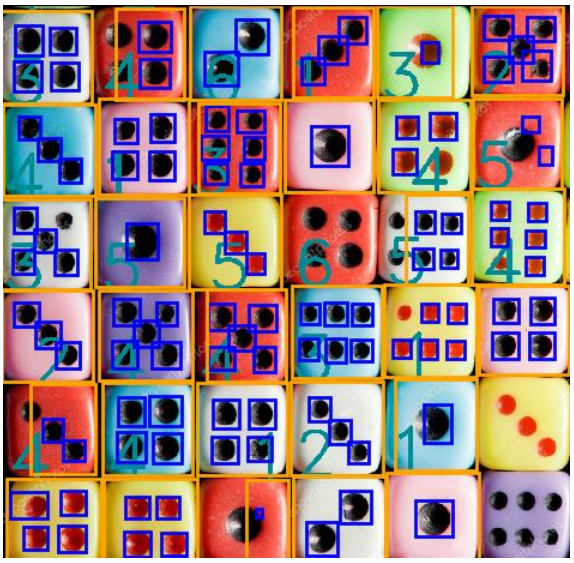
W programie posługujemy się abstrakcją obiektu, a konkretnie jego bounding boxem. Jedyne założenie co do niego jest takie, że ma on kształt prostokąta mniej lub bardziej przypominającego kwadrat (w przypadku poszukiwania kostek przyjmujemy, ze owy bounding box ma stosunek boków maksymalnie 2:1 niezależnie od krawędzi). W przypadku kropek na kostce dochodzi ograniczenie, że może być ich maksymalnie 6, jednak gdy żadna nie zostanie wykryta, takowa kostka nie jest w ogóle brana pod uwagę w wyniku.

Tworząc program próbowano utrzymać tendencje do ignorowania szumów występujących również na obrazie, jak tekst (pospolite w obrazach pobieranych z Internetu) czy odblaski światła. Zabiegi te często jednak mogą wpływać na poprawność programu (częściej na ignorowanie poprawnego wyniku, aniżeli jego zastosowanie).

Niestety, powyższe funkcjonalności częściowo usztywniają możliwości aplikacji ze względu na:

- bliskość kości: w przypadku zbyt dużej bliskości kości mogą zostać zidentyfikowane jako jedna; gdy suma ich oczek jest większa niż 6, są one zawsze błędnie identyfikowane.

- gładkość krawędzi: gdy kości mają obłe krawędzi, może to powodować załamanie algorytmu wykrywającego kontury kości. Istnieje duże prawdopodobieństwo, że początkowo wykryje on ściany [ostre krawędzi], a następnie górną ścianę, która jednak ma zaokrąglone rogi. Kontur otaczający będzie jednak słabo wyraźny ze względu na mały kontrast pomiędzy górną a bocznymi ścianami. Wewnętrzy zaś z tego samego powodu (mały kontrast) zostanie wykryty, jednak możliwe jest, że jego kontury będą na tyle niewyraźne (lub częściowe), że zostaną przysłonięte przez kropki wewnątrz (lub w najlepszym razie zostanie wzięty pod uwagę mniejszy obszar poszukiwania).



Rys. 1: Ogólność dziedziny wspieranych przypadków sprawia trudności w zaspokojeniu wszystkich przypadków testowych

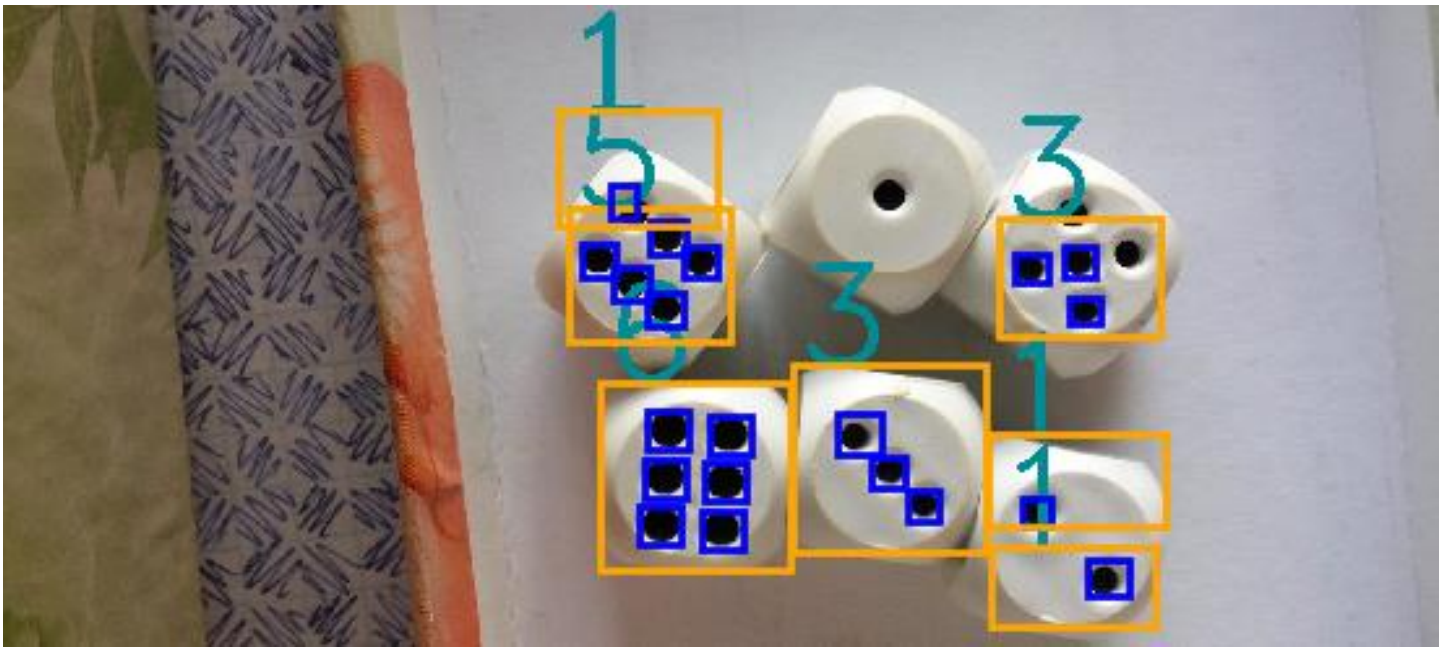
- nierównomierna wielkość: W przypadku, gdy na obrazie znajduje się wiele kości w perspektywie, lub różnej wielkości, ciężko stwierdzić które zostaną wykryte, a które uznane za szum. Algorytm bazuje bowiem na założeniu, że obiekty są podobnej wielkości (usuwane są wszystkie odstające w większym stopniu od zadanego percentyla).

Ponadto dowolność tła oraz samych kości powoduje, że nie mamy żadnego poziomu odniesienia w przypadku wątpliwości – jedynym punktem zaczepienia jest wydobywanie z obrazu wyróżniających się od otoczenia obiektów, przy założeniu (które występuje w programie), że stanowią one co najwyżej jaką część obrazu – mówiąc dosadnie, w przypadku badania jedynie samej góry kostki pokrywającej całe zdjęcie, **nie zostanie ona poprawnie zidentyfikowana**.

2. Zawodność programu

2.1. Pozornie trywialne przypadki

Wszystkie wcześniej wymienione ograniczenia sprawiają, że program był pisany – niestety – metodą prowadzoną przypadkami testowymi. Starano się, by ich pula była jak najbardziej różnorodna, jednak oczywiście pokrycie wszystkich przypadków nie jest w pełni możliwe, stąd istnieje możliwość, że dla, jakby się mogło wydawać, najtrywialniejszych przypadków, program może zwrócić nieprawidłowy lub zniekształcony wynik. Na następnej stronie znajduje się wynik pozornie prostego przypadku, a także charakterystyka rezultatów.

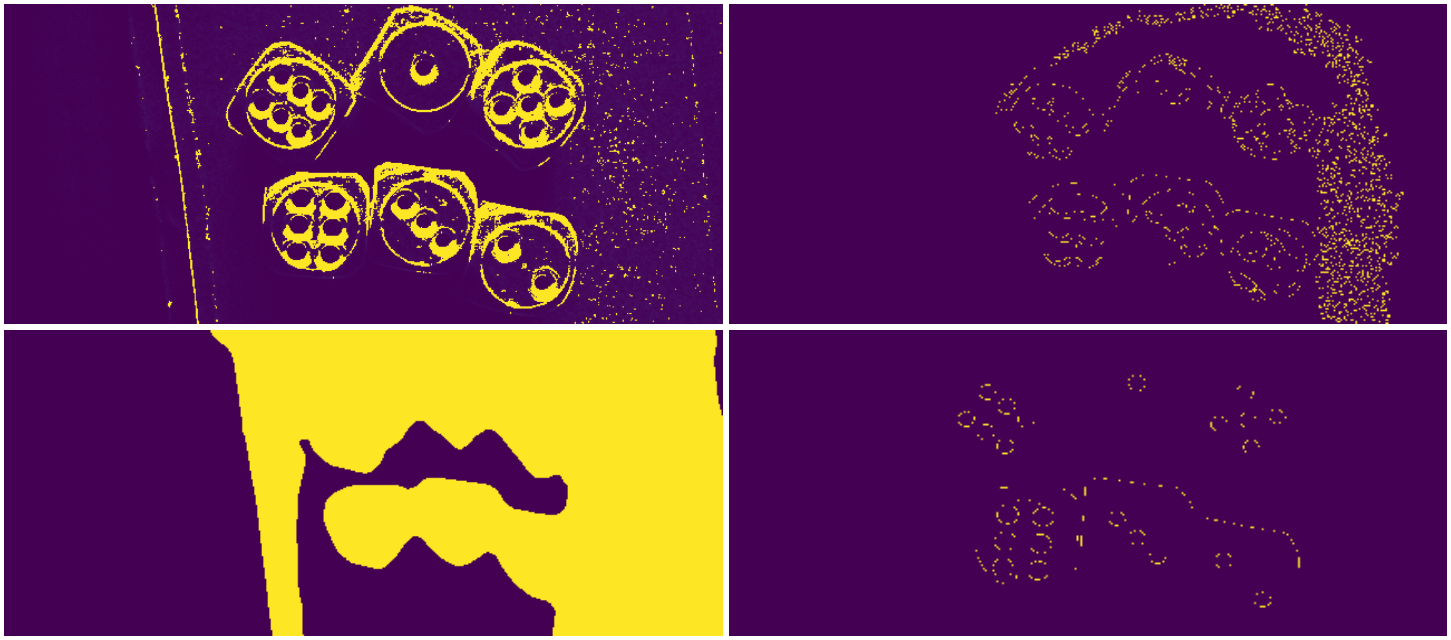


Rys. 2: Wynik działania program.

Wynik może być zaskakujący ze względu na jego niedoskonałość, jednak gdy spojrzymy na obraz zwrócony przez jedną z kilku operacji „wytrawiania” kości z rysunku, powód takiego zachowania wydaje się zrozumiały:

Rys. 3: Rezultat zwrócony przez jeden z wyodrębniaczy kości.

Cześć pozostałych przedstawiono poniżej:

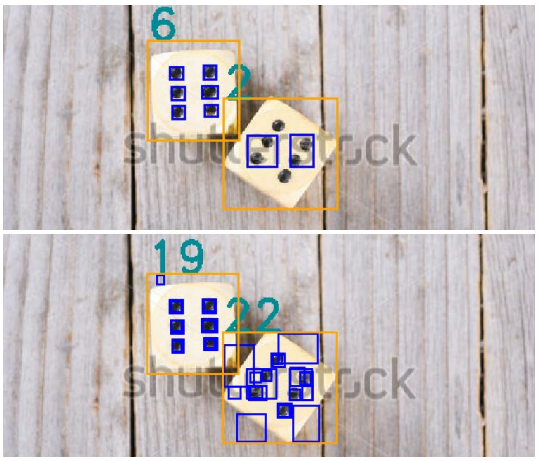


Rys. 4: Część wyników pozostałych wyodrębniaczy.

Jak więc widzimy, rezultaty są z goła odmienne. Gdyby zawarta była sztucna inteligencja, prawdopodobnie udałoby sie z obrazka znajdującego się w prawym dolnym rogu wyodrębnić ilość oczek, lub nawet całą kość na podstawie zagęszczenia znalezionych obiektów. Widzimy jednak, że kółka są nieregularne (przypominają raczej elpse), oraz nieciągłe. Możnaby próbować łączyć owe linie, jednak prawdopodonie wtedy uzyskalibyśmy dużo połączonych kształtów, z czego dla dolnej kostki o 6 kropkach wynikiem byłoby 5. Aby temu zapobiec, wymagane byłby zapewne heurystyki weryfikujące położenie kropek w zależności od wykrytej ich ilości, co mogłoby w tym wypadku dać prawidłowy rezultat, w ogólności jednak byłoby dużo bardziej złożone, oraz prawdopodobnie nieopłacalne pod względem czasowym. Same wyniki również mogłyby być niezadowalające oraz często zawodne (pochylenie obrazu, niedoskonałości). Na uwagę zasługuje również rezultat umieszczony w górnym lewym rogu – należy jednak pamiętać o tym, że musimy wyodrębnić oddzielne regiony – tutaj kwalifikowane jedynie wartościami {0,1}, nie natomiast samym kształtem. W związku z tym wykryte kontury w dużej mierze stanowią jedność

2.2. Obsługa występujących na obrazie napisów – szumów

Wykrywanie kropek jest również utrudnione, gdy na zdjęciu występuje napis – nawet w innym niż kropki kolorze. Opisane zachowanie dobrze obrazuje zamieszczone obok zdjęcie. Na kostce po lewej stronie górnego zdjęcia ilość kropek została wykryta prawidłowo, jednak dla prawej kostki napis spowodował uniemożliwienie poprawnej identyfikacji. Należy tutaj nadmienić, że oczka filtrowane sa według danych statystycznych. Dolne zdjęcie przedstawia natomiast wynik po zlikwidowaniu większosci filtrów. Widzimy na nim, że faktyczne kropki zostały wykryte, jednak znajdują się one w obrysie liter powodując, że zostają one przez nie pochłonięte. W dalszej części są zaś brane pod uwagę parametry wypełnienia konturu w bounding boxie, jego całkowita powierzchnia oraz szereg innych parametrów, na skutek czego te, które zawierają napisy wypierają prawidłowe wyniki.

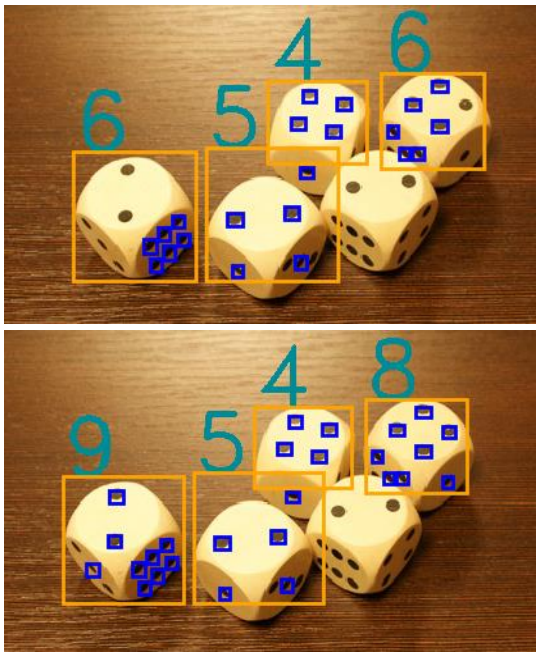


Rys. 5: Filtrowanie zdjęcia zawierającego napis oraz jego możliwe konsekwencje

2.3. Obsługa przypadków testowych z dużym kątem nachylenia

Założenia co do zadanych zdjęć są dosyć swobodne w porównaniu do przyjętego algorytmu – wykrywanie potencjalnych kości, a dopiero w dalszej kolejności znajdujących się na danej kości oczkach. Może to powodować problemy z wykryciem faktycznej liczby oczek na kostkach, co dobrze obrazuje zamieszczony obok wynik działania.

Wiemy, że kostka znajdująca się skrajnie na lewo przedstawia wartość 2. Algorytm jednak dał wynik 6 ze względu na fakt większego zagęszczenia zwracanych w rezultacie kółek. Gdyby wyeliminować to ograniczenie, wynik wyglądałby jak na drugim z rysunków. Oczywiście jest to pewnego rodzaju błąd – pośrednio algorytm mógł wydzielić kość na dwie części, jednak nie jest to wspierane ze względu na zamkniętość świata, jakim jest środowisko w jakim wyszukujemy oczka. Można by również wyznaczyć (lub lepiej wydzielić powierzchnie kości) tak, aby preferować wyżej położoną, jednak tutaj ponownie pojawia się problem braku identyfikacji barw kości oraz samych oczek.



Rys. 6: Nieprawidłowy wynik ze względu na zbyt dużą ilość wykrytych kropek, oraz otrzymany po wyłączeniu ograniczenia na ilość oczek.

2.4. Różne poziomy naświetlenia

Zmiana poziomów natężenia światła jest problematyczna w ogólności, jednak w momencie, gdy nie mamy żadnego poziomu odwołania, bardzo ciężko jest ją zniwelować. W poprzednim podpunkcie analizowane było zdjęcie zawierające zarówno wysokie poziomy oświetlenia, jak i znaczne cienie. Same kości znajdowały się również stosunkowo blisko siebie, czego implikacja została przedstawiona poniżej – jest to wynik uzyskany dzięki jednemu z filtru wyodrębniającego kości z rysunku.

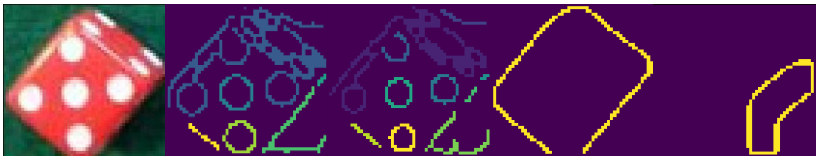
Warto nadmienić, że ostateczne obszary do filtrowania kości są wynikiem złożenia wszystkich filtrów, dzięki czemu filtry w ogólności nie muszą pokrywać całej przestrzeni, jednak istnieje prawdopodobieństwo, że któryś nie pokryje zadanego przypadku – jak w obecnej sytuacji.

Rys. 7: Różne poziomy natężenia swiatła mogą spowodować błędne identyfikacje kości

2.5. Bardzo małe obiekty

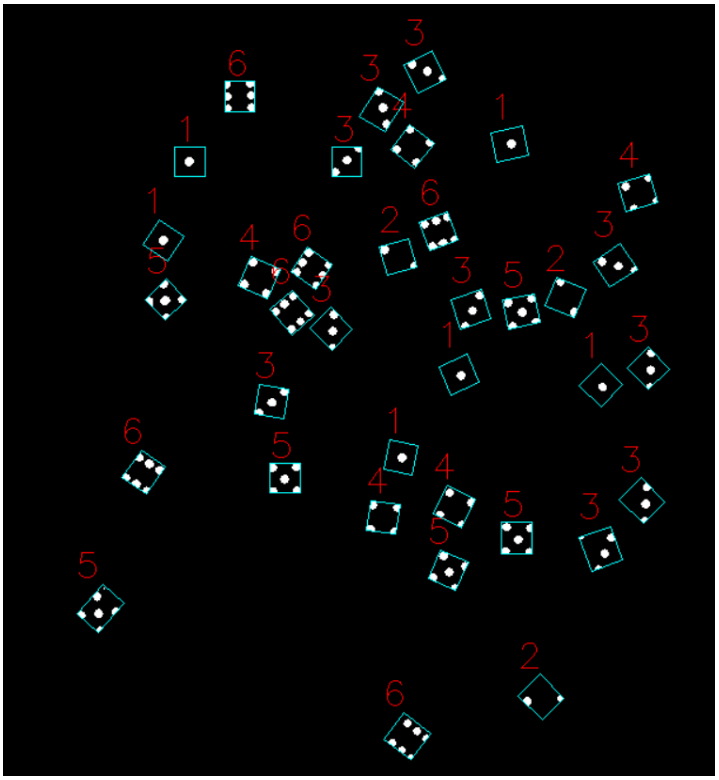
Rys. 8: Zdjęcie z wieloma kostkami tego samego rodzaju, jednak pod różnym nachyleniem oraz naświelteniem

Ze względu na brak wyspecyfikowania ograniczeń dotyczących ilości kości na zdjęciu, samej wielkości zdjęcia czy też minimalnej wielkości kości, wyniki mogą być w ogólności niepoprawne. Przykładowo, rozpatrzmy zamieszczony obok obraz. Zaistniała sytuacja może być bardzo zastanawiająca. Należy jednak wziąć pod uwagę, że rozmiar całkowity zdjęcia wynosi 600 x 600 pikseli, zaś pojedynczej kości (bounding boxa) oscyluje w przedziale 30-50 px. I tak, przykładowo dla jednej z kości otrzymamy wynik zamieszczony na rysunku nr 9 – ciężko stwierdzić, aby były one satysfakcjonujące. Oczywiście, gdyby przyjąć założenie, że kości są czerwone, problem byłby prawie że trywialny. W obecnej sytuacji nie możemy jednak takiego założenia podjąć, w związku z czym jesteśmy zmuszeni do obsługi przypadku niskiej jakości zdjęcia (kości są rozmazane, co widać na rysunku), licznych cieni oraz zróżnicowanych kątów nachylenia. Zdjęcie jest pobrane z jednego z przykładów znalezionych w Internecie (https://github.com/andli/dicecounter), gdzie jest to sztandarowy przykład autora – wyniki znajdują obrazuje rysunek 10



Rys. 9: Surowe rezultaty wykrywania oczek na kości

Widać więc, że program w obecnym stanie ma jeszcze spory obszar do pokrycia oraz pole do rozwoju. Problem jednak wciąż może stanowić duża ogólność, gdyż nie wiemy na jakich wartościach się skupić – należy pamiętać, że oczka wykrywane są poprzez transformacje na obrazie. Jak widać na



Rys. 10:Wyniki możliwe do uzyskania przy założeniach dotyczących barwy (https://github.com/andli/dicecounter)

Rys. 11: Wynik uzyskany dzięki dodaniu kolejnego dostawcy surowych obrazów oczek na kostce

uzyskanych przykładach, te są skrajnie różne. Można by taki uzyskać, jednak skutkowałoby to pogorszeniem rezultatów dla części pozostałych wyników.

Widać więc, że wzrost efektywności jest znikomy. Jest to spowodowane licznymi filtrami, które jednak są konieczne przy podjętych założeniach dotyczących niewrażliwości (lub próbie zaradzenia) na przykładowo wystąpienie tekstu. W przypadku ich wyłączenia otrzymamy rezultat jak ten zamieszczony na rysunku 12. Mimo to jednak widzimy znaczą ilość niepokrytych przypadków – część oczek wciąż pozostaje niewykryta.

Rys. 12: Rezultat uzyskany po wyłączeniu cześci filtrów usuwających elementy odstające

3. Działanie programu

Jak częściowo zostało wspomniane wcześniej, program bazuje na 4 zasadniczych fazach:

a) Przetworzenie dostarczonego obrazu pod kątem wykrycia dużych obiektów znajdujących się na nim, wyróżniających się od otoczenia.
b) Odfiltrowania uzyskanych regionów pod względem trafności, wypełnienia, stosunku długości krawędzi, nakładających się lub zbyt dużych/małych regionów oraz ogólnie statystyki uzyskanych rezultatów.
c) Poszukiwanie potencjalnych oczek w przestrzeni każdej z uzyskanych we wcześniejszym kroku kostek.
d) Filtrowanie oczek znajdujących się w rejonie poszczególnych kostek przez szereg ograniczeń statystycznych oraz jakościowych.

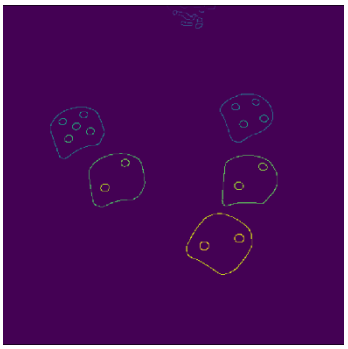
W dalszej części zostaną bardziej szczegółowo omówione poszczególne kroki oraz sposób ich implementacji.

Rys. 13: Analizowany przykład

3.1. Przetwarzanie początkowego obrazu pod kątem wyszukania kości

Jest to prawdopodobnie najważniejsza cześć programu, bowiem jeśli w niej kość nie zostanie wykryta, nie jest brana pod uwagę w dalszym przetwarzaniu. Jest to często spotykane w zdjęciach zróżnicowanych pod względem kontrastu, gdzie cześć obiektów jest mocno skontrastowana, z kolei inna znajdująca się w mniejszości odstaje od nich – większość tych algorytmów bazuje bowiem na pewnego rodzaju statystykach odnośnie jasności czy zagęszczenia kolorów. W obecnej wersji programu wyspecyfikowanych zostało 6 algorytmów obróbki obrazu, których zadaniem jest odnalezienie na dostarczonym obrazie kości. Będą one omawiane na przykładzie zdjęcia zamieszczonego na rysunku 13. Należy jednak wspomnieć, że każdy z tych algorytmów ma inne zadanie detekcji poszczególnych cech, dlatego cześć wyników może wydawać się zbędnych dla danego przypadku.

a) Get_edges – nazwa nie jest zbyt znacząca, jednak jest to podstawowy algorytm – również bardzo prosty. Bazuje on na:
   - zwiększenie kontrastu (sparametryzowany w tym przypadku dla 91 i 90 percentyla – w takiej kolejności[negacja]) na obrazie monochromatycznym.
   - zwiększenie jasności obrazu (transformacja Gamma – parametr == 0.4 przy wartościach [0,1]).
   - zastosowanie algorytmu Canny z sigmą równą 2.7. Parametry dotyczące dolnego oraz górnego poziomu pozostają domyślne – odpowiednio 10 oraz 20 procent maksymalnej wartości.



a) just_canny_and_dilation – jak nazwa mówi, algorytm opiera się na transformacji Canny'ego. Wcześniej jednak obraz poddany jest transformacji monochromatycznej oraz dylatacji. Tym razem jednak niski próg Canny jest ustalony na 0.05, natomiast wysoki na 0.3. Sigma wynosi 4.

Rys. 15: Rezultat dla just_canny_and_dilation przy wyszukiwaniu kości

b) Get_by_hsv_value – schemat bazujący na progowaniu z obrazu o formacie HSV, kolejno:
   - obierany jest format HSV
   - wszystkie wartości v większe od progu (w tym przypadku 0.8) podnoszone są do wartości 1, natomiast wszystkie poniżej sprowadzane do 0.
   - obraz transformowany jest do postaci monochromatycznej
   - przeprowadzenie podwójnej erozji, za pierwszym razem z siatką 2x2, natomiast za drugim 3x3.
   - przeprowadzany jest proces otwarcia (domyślna siatka)

Rys. 16: Rezultat dla get_by_hsv_value przy odnajdywaniu kości

c) splashing_image – poddanie obrazu bardzo dużemu rozmyciu (skuteczne dla małej ilości kości na rysunku, jednocześnie przy zróżnicowanych warunkach otoczenia czy szumów)
   - obraz jest negowany, a następnie transformowany do postaci monochomatycznej
   - wszystkie wartości obrazu mnożone są razy dwa
   - następuje transformacja Gamma (rozjaśnienie) z parametrem 0.5
   - wartości obrazu dzielone są przez 2
   - następuje erozja obrazu z siatką 10x10
   - następuje ponowna negacja obrazu
   - obraz poddany jest filtrowi medianowemu z siatką 25x25
   - następuje progowanie obrazu metodą Otsu
   - obraz ponownie poddawany jest filtrowi medianowemu z siatką 50x50

Rys. 17: Rezultat dla splashing_image przy odnajdywaniu kości

d) sobel_and_scharr_connected – opiera się na algorytmie wykrywania krawędzi Sobel'a oraz scharr'a:
   - obraz transformowany jest do monochromatu
   - następuje przyciemnienie obrazu – transformacja Gamma z parametrem = 5
   - na osobnych kopiach przeprowadzony jest algorytm Sobel'a oraz Scharra. Następnie wyniki są sumowane.
   - Jeśli uzyskane w wyniku sumowania wartości są większe od wartości progowej(0.05), następuje ich wyniesienie do 1.
   - następuje dylatacja z domyślną siatką

Rys. 18: Rezultat dla sobel_and_scharr_connected przy odnajdywaniu kości

e) High_percentile_shrink – opiera się na manipulowaniu kontrastem
   - Transformacja do monochromatu
   - Pociemnienie obrazu z parametrem gamma = 3
   - Następuje zwiększanie kontrastu metodą wyrównania histogramu
   - Jeśli średnia z wartości w obrazie jest większa niż 0.5, następuje jego negacja
   - Zastosowanie progowania obrazu metodą Otsu.

Rys. 19: Rezultat dla high_percentile_shrink przy odnajdywaniu kości

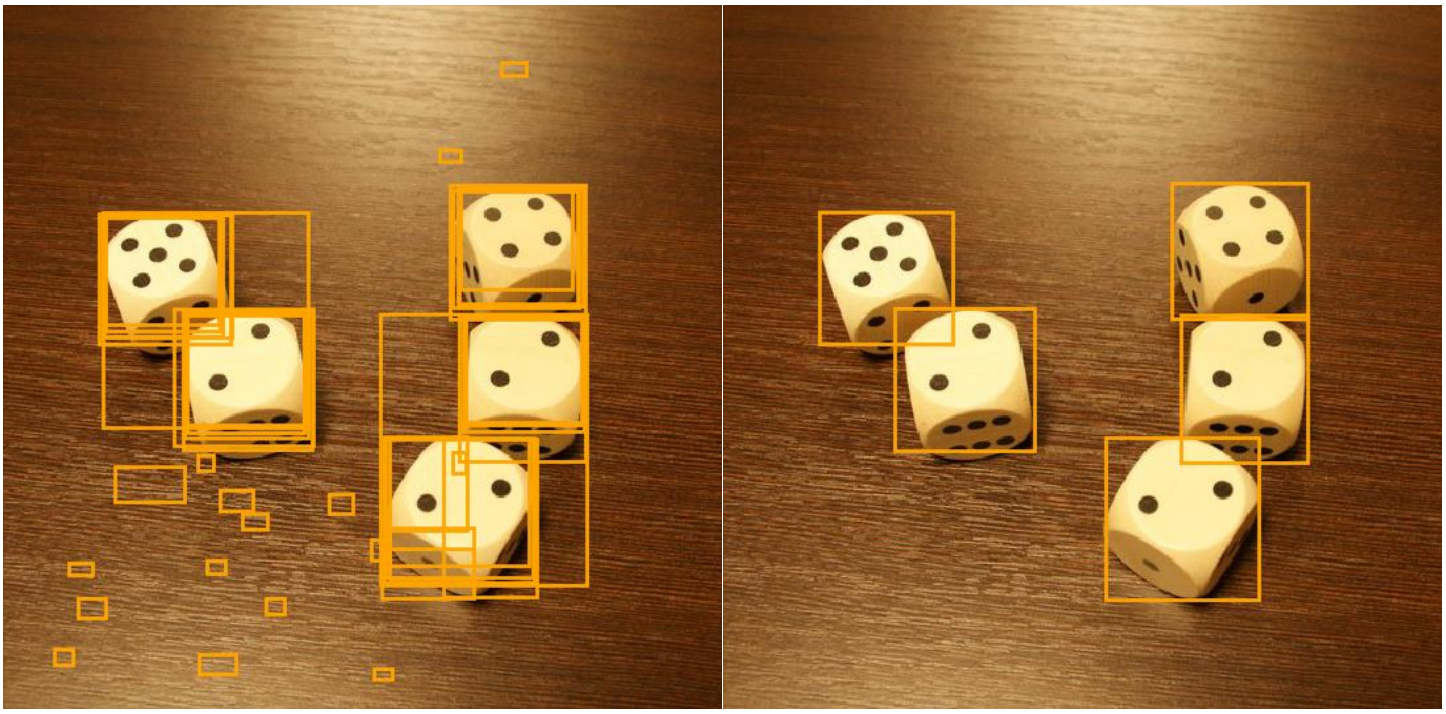
Jak widać na rysunkach 14-19 pokazujących działanie algorytmów wyszukujących w obrazach kości, są one różnokolorowe – jest to skutek oznaczania wykrytych osobliwych regionów, co również daje możliwość do przeprowadzenia następnego kroku – filtrowania.

3.2. Filtrowanie potencjalnych kości

Filtrowanie odbywa się jako jeden ciąg czynności – jeśli dany kandydat nie przejdzie pojedynczego kroku, jest eliminowany.

a) W pierwszym kroku eliminowane są wszystkie oznaczone regiony, których bounding box(szerokość x wysokość) ma powierzchnię mniejszą niż 0.25% całkowitej powierzchni obrazu.
b) Następnie usuwane są powierzchnie w przybliżeniu jednokolorowe, tzn. oznaczające się bardzo niewielkim kontrastem, lub posiadających jedną w pełni dominującą barwę (np. obraz o różnych odcieniach czerwonego w konkretnym przedziale wartości). Opiera się to na:
   - progowaniu wartości rgb (średnio co 50),
   - algorytmie określającym, czy obraz jest skontrastowany, znajdującym się w bibliotece scimage, w module exposure – is_low_contrast
c) następnie kandydaci zostają posortowani malejąco według powierzchni bounding boxa w celu usunięciu kolejno wszystkich odstających pod względem kolejno szerokości, wysokości oraz powierzchni bounding box od pozycji znajdującej się na indeksie 25% z całej stawki. Akceptowani są kandydaci mieszczący się w zadanym promieniu określającym stopień odchyłki od wartości wyroczni.
d) W kolejnym kroku łączeni są wszyscy kandydaci, którzy się pokrywają oraz mają co najmniej 25% wspólnej powierzchni
e) Ponownie usuwani są wszyscy kandydaci którzy odstają pod względem pola bounding boxa od kandydata znajdującego się w na indeksie (0.25 * ilość kandydatów).

Rezultaty działania opisanego powyżej filtru znajdują się na następnej stronie (rysunek 20).



Rys. 20: Rezultaty uzyskane przed oraz po zastosowaniu filtrów opisanych w punkcie 3.2

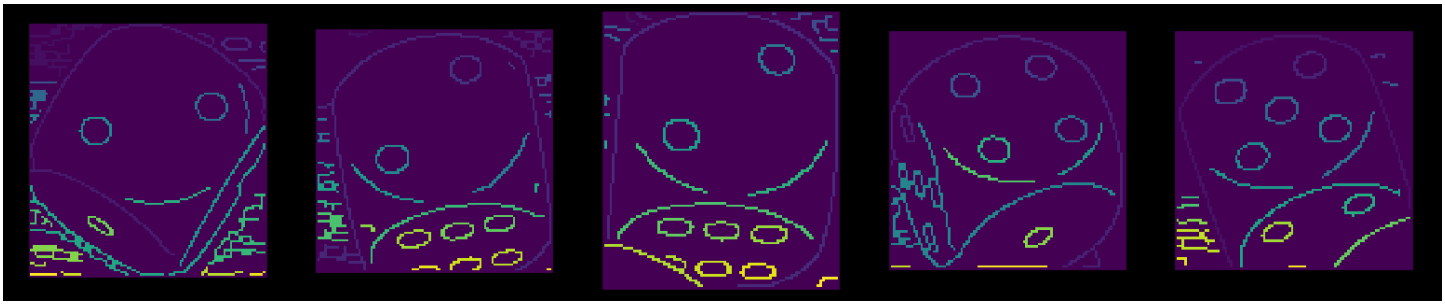
3.3. Odnajdywanie oczek na kościach

W kolejnym kroku zawężamy obszar poszukiwania oczek do wcześniej odfiltrowanych regionów, przyjmując, że zawierają one kości. Ponownie przetwarzamy obrazy wydzielone w regionach znalezionych kości.

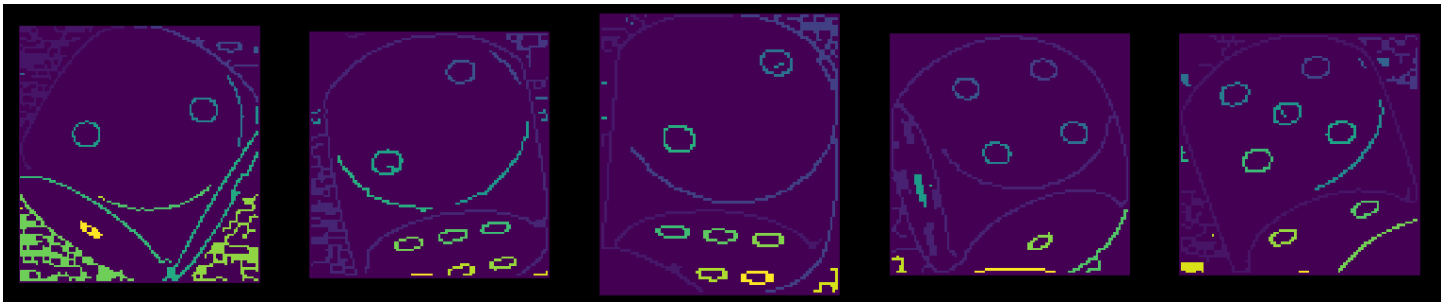
Tym razem operujemy na 4 algorytmach:

a) Get_edges – ponownie, tym razem jednak gamma wynosi 1, sigma 1, bez ograniczeń odnośnie dolnych i górnych percentyli (0,100).
b) Just_canny_and_dilation – z parametrami sigma 0.4, niskim progiem na 0.1 oraz wysokim na 0.3
c) Double_sobel – jest to algorytm specyfikowany do wynajdywania oczek tam, gdzie kontrast jest nikły, np. czarne kropki na ciemno czerwonym tle. Jest to raczej algorytm pomocniczy.
    - Początkowo obraz poddawany jest transformacji monochromatycznej oraz filtrowi medianowemu z siatką 10x10
    - Zastosowane zostaje progowanie metodą Otsu – pamiętamy obraz jako o1
    - Od początkowego monochromatycznego obrazu odejmujemy uzyskany metodą progowania (o1 - niwelując również wartości ujemne)
    - Stosujemy filtr medianowy z siatką 13x13 (dobrane raczej metodą empiryczną)
    - Ponownie stosujemy progowanie Otsu oraz negacje obrazu
    - Odejmujemy od obrazu o1 wynik ostatnich przekształceń, ponownie niwelując wartości ujemne
    - Stosujemy filtr medianowy, ponownie z siatką 13, a następnie algorytm Canny'ego z parametrem sigma = 2
    - Nastepuje progowanie Otsu, dylatacja z siatką 10x10 oraz ponownie filtr Canny'ego, tym razem z sigmą = 0.2

d) Negative_on_v_and_canny – filtr ten z kolei przeznaczony jest do wykrywania oczek na obrazkach o bardzo małej rozdzielczości oraz niskiej wyrazistości.
   - Obraz mapowany jest na 3 współrzędną HSV, jednak zanegowaną
   - Następuje filtrowanie medianowe oraz erozja
   - Następuje pociemnienie obrazu (zaciemnienie oryginału) – gamma = 2
   - Zamknięcie oraz progowanie Otsu
   - Dylatacja z siatką 2x2
   - Canny z sigmą 0.9 oraz domyślnymi progami.

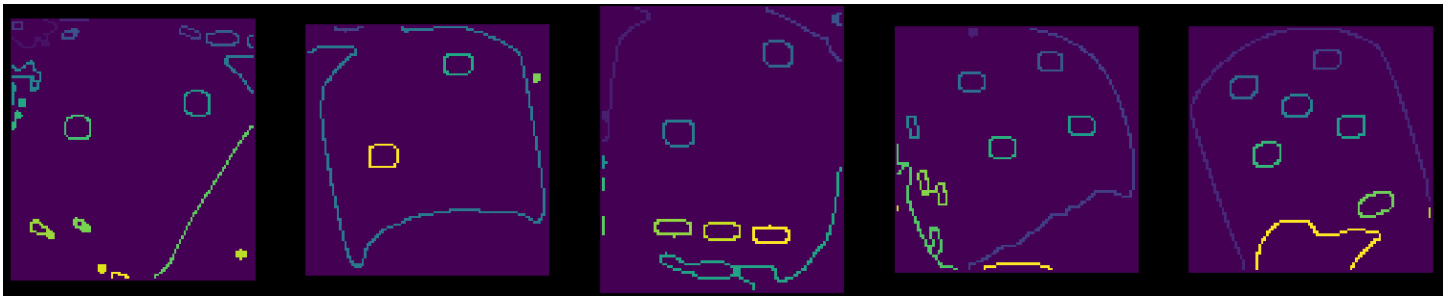


Rys. 21: Rezultat dla get_edges przy wyszukiwaniu oczek



Rys. 22: Rezultat dla just_canny_and_dilation przy wyszukiwaniu oczek

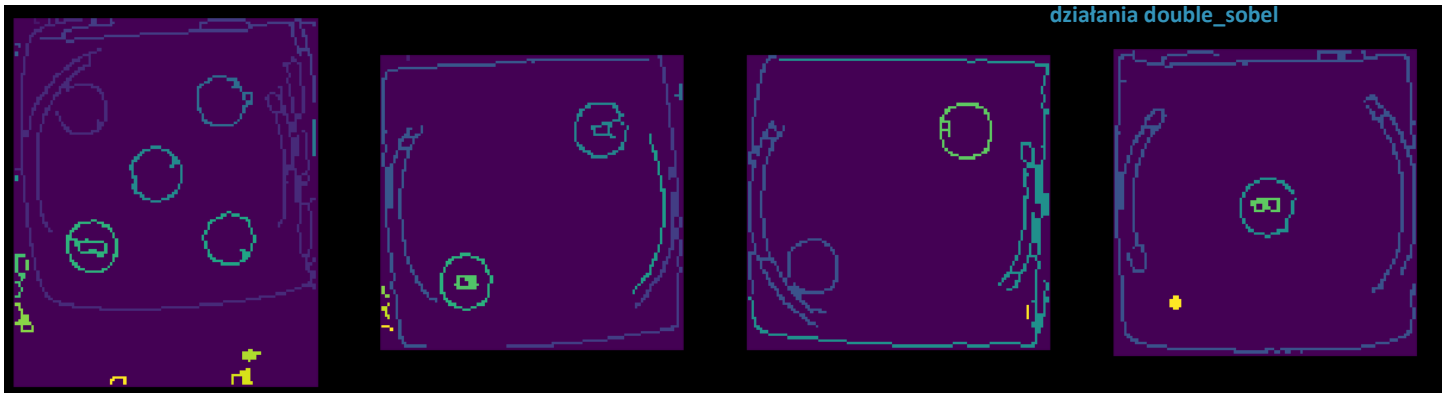
Rys. 23: Rezultat dla double_sobel przy wyszukiwaniu oczek



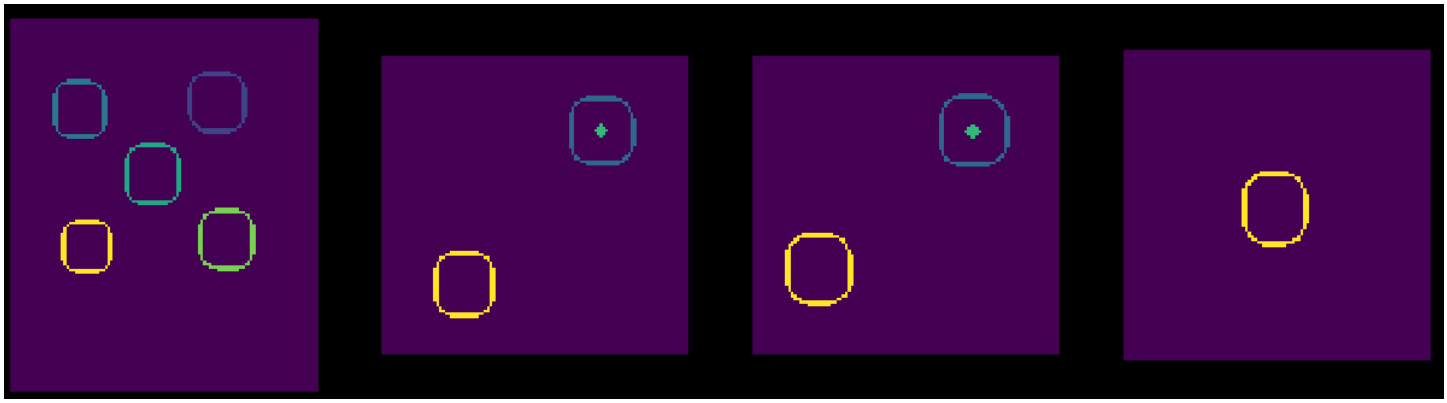
Rys. 24: Rezultat dla negative_on_v_and_canny przy wyszukiwaniu oczek

Algorytm double_sobel może sprawiać wrażenie nadmiarowego, jednakże nie do końca tak jest – ma on swoje specyficzne zastosowania. Przykładowo wynik działania get_edges dla obrazu prezentowanego na rysunku 25 jest następujący zaprezentowany jest na rysunku 26, natomiast po obróbce algorytmem double_sobel wynik jest taki, jak na rysunku 26. Ponadto sprawdza się on również dla oczek niewyraźnych, bowiem zwiększa on ich średnicę zwiększając (jedynie) ich szansę w dalszym procesowaniu.

Rys. 25: przykład dla weryfikacji działania double_sobel



Rys. 26: Wynik przetworzenia przykładowego obrazu przez get_edges



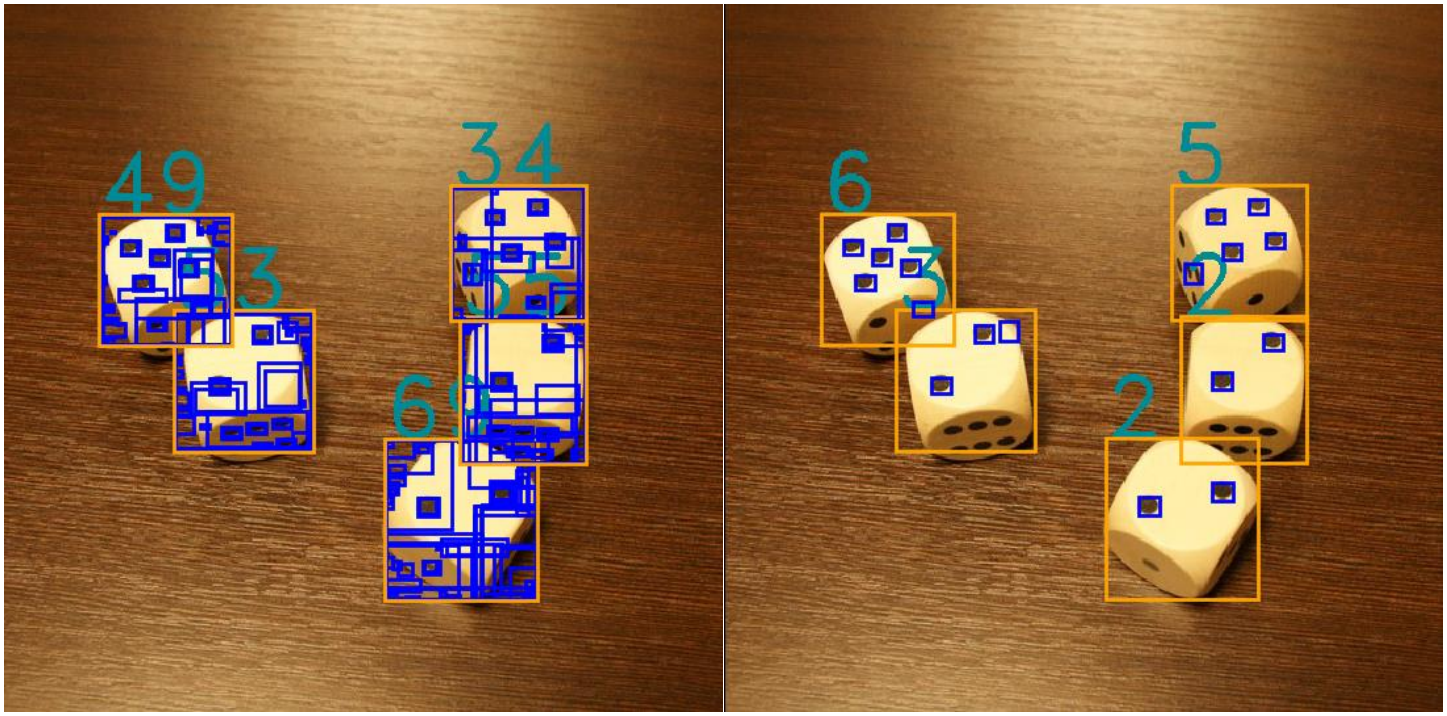
Rys. 27: Wynik przetworzenia przykładowego obrazu przez double_sobel

3.4. Proces filtrowania oczek

Proces filtrowania oczek, podobnie jak filtrowanie kostek, podzielony jest na fazy o charakterze przepływu.

- Usuwane są wszystkie potencjalne oczka, których bounding box jest większy niż 15% lub mniejszy niż 0.5% powierzchni całego badanego fragmentu
- Przez następne sito przechodzą jedynie kandydaci, których krawędzie bounding boxa spełniają wymaganie stosunku 7:5 – w idealnym przypadku byłoby to oczywiście 1:1, jednak musimy brać pod uwagę perspektywę oraz naświetlenie. Niedoskonałości zdjęć również odgrywają tutaj kluczową rolę.
- Kolejnym krokiem jest kandydatów wykrytych w rogach. Istnieje spore prawdopodobieństwo, że są oni jedynie szumem. Usuwani są również Ci, którzy znajdują się na krawędzi badanego obszaru.
- W dalszej części usuwani są kandydaci znacząco odstający od 80 percentyla wartości powierzchni bounding boxów wszystkich kandydatów. Proces jest 3 stopniowy, za każdym razem przepuszczamy przez niego minimum 30% kandydatów, w przeciwnym razie zwiększamy promień akceptacji.
- Z racji kilku algorytmów wykrywania oczek, a także braku specyfikacji ich charakteru, otrzymujemy bardzo dużo szumu oraz kilkukrotnych wykryć. Z tego powodu konieczne jest łączenie uzyskanych rezultatów oraz łączenie częściowych wyników, co następuje w tej fazie.
- Do dalszej fazy przepuszczani są jedynie kandydaci spełniający przynajmniej jeden z 3 warunków dotyczących pola bounding boxa, wypełnienia oraz wielkości wykrytego regionu (faktycznego pola zataczanego przez obrys). Kandydaci porównywani są ponownie wedle 25% indeksu od całkowitej ilości kandydatów. Parametry akceptacyjne nie przekraczają 20% parametrów wyroczni.
- W ostatniej fazie zakładamy, że początkowo został wybrany faktycznie obrys jedynie jednej kości, a więc również, że maksymalnie możemy wykryć 6 oczek. Z tego powodu następuje eliminacja kandydatów najbardziej oddalonych od środka ciężkości.

Wynik całego procesu obrazuje rysunek 28



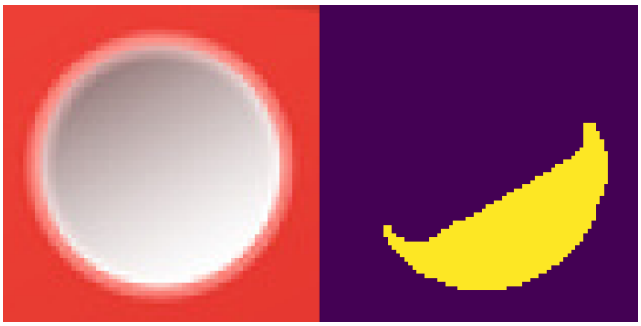
Rys. 28: Skutek działania filtru oczek. Lewy obrazek przedstawia stan przed zastosowaniem filtru, prawy zaś po procesowaniu przez niego.

4. Problemy, rozwiązania, wątpliwości

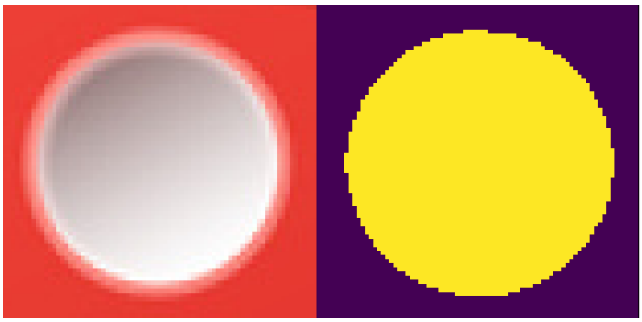
W dalszej części znajduje się rozpatrzenie części problemów które zostały napotkane w trakcie realizacji projektu – jedynie te oparte na przypadkach testowych. Niestety, cześć z rozwiązań usztywniły algorytm, utrudniając jego dalszy rozwój czy elastyczność na przypadki, część rozwiązań przyczyniła się jedynie do rozwiązania danego problemu i jemu pokrewnych – ściśle.

4.1. Problem występowania tekstu na obrazach

Jak wspomniane zostało we wprowadzeniu, algorytm projektowany był z myślą o niwelowaniu szumu w postaci tekstów występujących często na zdjęciach pobranych z Internetu. Problem oczywiście nie występuje w realnych zastosowaniach, a jedynie można by rzec – jako benchmark, stąd jest to pewnego rodzaju cecha dodatkowa. Starano się, by była ona w pełni wspierana, jednak nie zostało to do końca spełnione. Oczywiście można podjąć idee, aby wykrywać jedynie obiekty okrągłe – została podjęta próba wdrożenia takiego algorytmu do programu, jednak pojawiły się problemy z ogólnym wykrywaniem okręgów – dobrze do tego celu spisuje się metoda Hough'a wyspecyfikowania okręgu. W niej jednak pojawia się problem wyróżnienia promienia, jaki ma mieć nasze koło. Ponadto pojawia się problem nierównomierności zdjęć, oświetlenia, nachylenia. Tutaj z pomocą może przyjść odmiana algorytmu pozwalająca na wykrycie elipsoid, jednak ma ona dużo większy koszt obliczeń. Wciąż jednak istnieje problem z właściwym przetworzeniem obrazu tak, by wyeliminować szumy, nie uszkodzić obrazu, przy tym zachowując krągłość obiektu. Przykładowo, występuje problem z rozróżnieniem litery 'o' od kółka. Można oczywiście sprawdzać, czy wykryty obiekt jest wypełniony, co również zostało zaimplementowane poprzez sprawdzanie, czy obszar w pewnym procencie oryginalnego promienia (dostarczonego dzięki wyspecyfikowaniu bounding boxa odkrytego oczka) ma określoną barwę (lub ściśle, czy wykryto tam w znacznej mierze 0 lub 1), oraz czy w większej od niego odległości wykryto w większości barwę przeciwną. Rozwiązanie to jednak okazało się mieć małe zastosowanie na badanych przypadkach ze względu na to, że przyjmujemy dowolny rozmiar zdjęcia, stąd często faktyczne oczka po transformacjach przyjmowały dowolny kształt. Sam pomysł jest oczywiście dobry, jednak jego dopracowanie wymagało sporych zasobów czasowych, których brakowało. Z tego względu postanowiono przyjąć postawę bardziej defensywną, opartą jednak na statystyce – przyjęto, że wykryte powierzchnie mają w większości podobną powierzchnię bounding boxa, ilość elementów konturu(pikseli obrysu) oraz wypełnienie – a przynajmniej powinny. Jako punkt odniesienia przyjęto 25% pozycję względem ciągu posortowanego według powierzchni bounding boxa – malejąco. Aby kandydat został przepuszczony, musi mieścić się w zadanym zakresie opartym na parametrach wyroczni oraz możliwej odchyłce procentowej. Ogólnie założenie jest dobre, jednak gdy weźmiemy pod uwagę fakt, że obraz może być w perspektywie, natomiast kostka stosunkowo duża (lub oczka na tyle małe, że nawet 20% to mniej niż 1 px), algorytm ten często może zawodzić. Przykładowo, rysunek 32 przedstawia wykryte 'oczka' po wcześniejszym

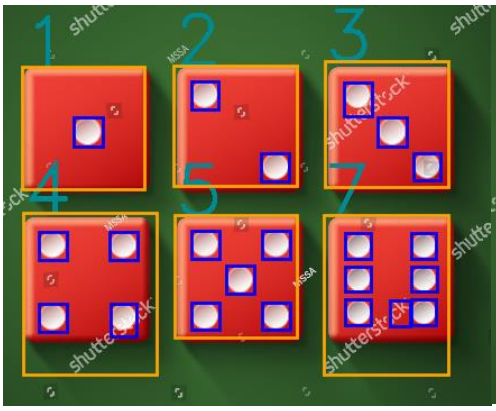


Rys. 29: Próby wykrywania kół na kostkach – 50x50 px

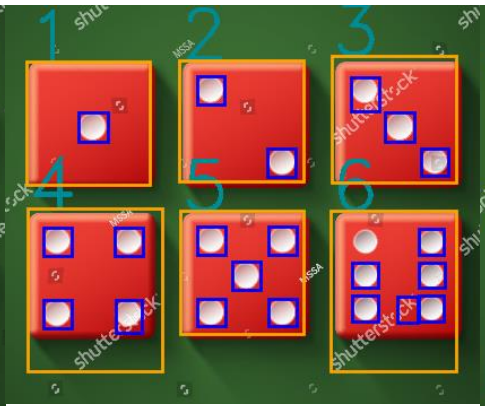


Rys. 30: Wykrywanie kół po modyfikacjach

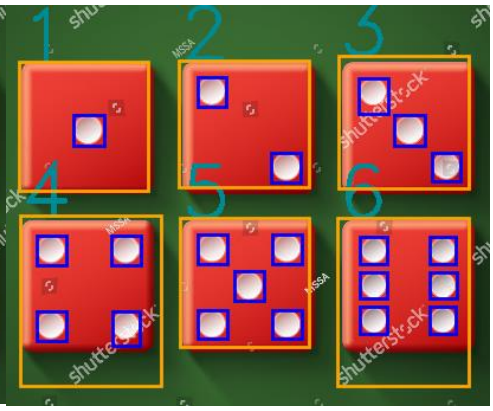
Rys. 31: Próba wykrycia na innym obrazku - 6x6 pikseli – moment przed wyrzuceniem błędu wykonania



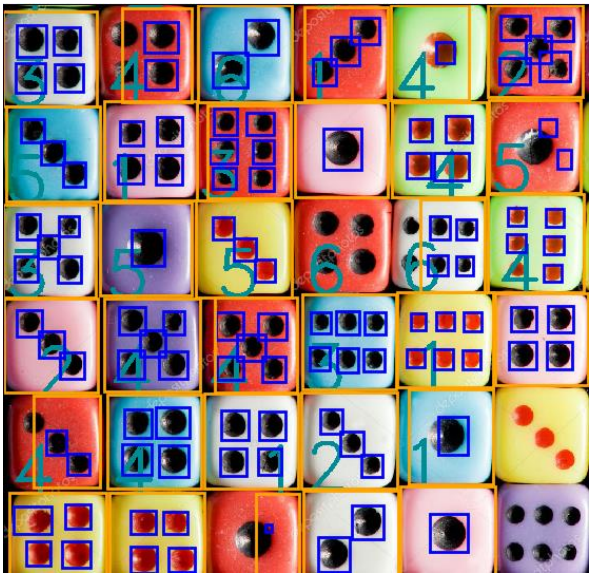
Rys.32: Brak odfiltrowania elementów odstających pod względem rozmiaru i wypełnienia



Rys.33: Wynik działania bez filtru statystycznego po odfiltrowaniu oczek nadmiarowych



Rys. 34: Wynik działania po zastosowaniu filtru statystycznego



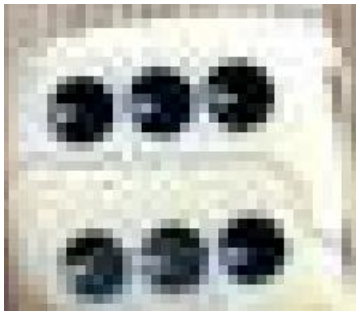
Rys.35: Wynik dla zdjęcia z rysunku 1 po wyłączeniu filtra statystycznego

odfiltrowaniu, jednak przed omawianym obecnie filtrem. Widać na nim, że na kostce usytuowanej w prawym dolnym rogu jako oczko został zakwalifikowany kawałek napisu. Sytuacja jest o tyle niefortunna, że został on wykryty tuż przy innym oczku, ponadto znajduje się on prawie w samym środku wykrytego bounding boxa kostki. W obecnej sytuacji, gdyby nie zastosować omawianego filtru, wynik byłby działania byłby taki, jak ten przedstawiony na rysunku 33 – widać więc, że napis ten musi zostać odseparowany przed naliczaniem ostatecznej ilości oczek. Rysunek 34 przedstawia rezultat uzyskany poprawne rozwiązanie uzyskane po zastosowaniu omawianego filtru.

Ogólnie jednak, gdyby usunąć filtr statystyczny, moglibyśmy dla innych przypadków uzyskać lepsze rezultaty. Na rysunku 35 znajduje się wynik dla zdjęcia przedstawionego na rysunku 1 z wyłączonym filtrem. Widzimy, że część rezultatów uległa poprawie. Sytuacja prawdopodobnie byłaby jeszcze lepsza gdyby dopracować rozwiązanie bazujące na wykrywaniu kółek (dla przykładu na rys 32-34 wersja próbna daje rezultaty dokładnie takie same jak rys 34 nawet bez stosowania filtrów przygotowawczych określających rozmiar minimalny). Ogólnie jednak problem nie jest do końca trywialny przy braku ustalenia wielkości obrazu kostki – próg 90% (który wydaje się rozsądny w stopniu wypełnienia koła poniżej wartości jego promienia) przy obrazie poniżej 8 px na każdą ze ścian może sprawiać problemy, szczególnie jeśli weźmiemy pod uwagę antyaliasing krawędzi (rysunek 31 dobrze oddaje sytuacje).

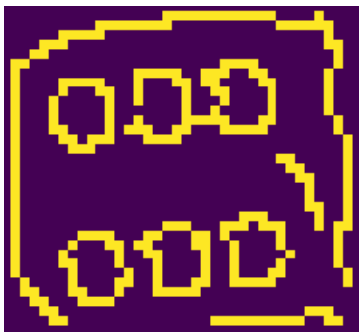
4.2. Duża ilość oczek przy jednocześnie niskiej rozdzielczości obrazu

W ogólnym przypadku, jakość nie powinna stanowić problemu – zdjęcia mogą być niewyraźne, kolor przejściowy. Problematyczna jest jednak sytuacja, gdy mamy do czynienia z kostką 6 oczkową, zdjęcie jest niskiej rozdzielczości, same kostki niezbyt duże oraz pod nachyleniem. Stwarza to problemy opisanie nie tylko w poprzednim punkcie, jednak również z wyspecyfikowaniem poszczególnych oczek po przetworzeniu – filtry o dużym stopniu modyfikacji sprawią, że z takiego fragmentu nie wyniknie dużo informacji. Przykładowo rozpatrzmy kostkę z rysunku 36. Widzimy, że oczka są wyraźne gołym okiem, jednakże pomiędzy nimi znajduje się gradient utrudniający ich identyfikację. W ogólnym przypadku filtr Canny daje zadowalające rezultaty przedstawione na rysunku 37 (choć już



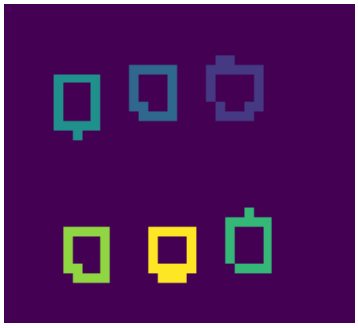
Rys. 36: Przykładowa kostka o niskiej rozdzielczości

tutaj oczka są połączone), jednakże każdy wynik ‘wyodrębniacza’ zostaje procesowany również przez algorytm wyodrębniania oddzielnych kontur – jego rezultat jest przedstawiony na rysunku 38. Konieczne więc stało się wykorzystanie filtru, który będzie w niewielkim stopniu manipulował obrazem – w obecnym stanie rzeczy algorytm nie wykrywał żadnego oczka (zostawały one usuwane na następnych stadiach filtrowania – oddzielnym kolorem zaznaczono wszystkie wykryte obiekty, tzn. zarówno górne jak i dolne oczka wykryte są jako pojedyncze obiekty). W tym celu powstał algorytm ‘negative_on_v_and_canny’, który daje dosyć zadowalające rezultaty przedstawione na rysunku 39. Algorytm ten działa na tyle dobrze, że jest w stanie samodzielnie wyodrębnić z badanego rysunku wszystkie oczka – rezultat przedstawia rysunek 41.

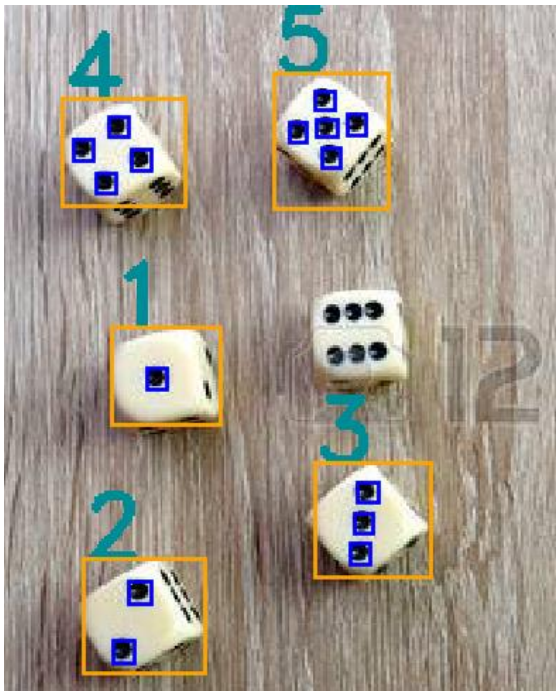


Rys. 37: Wynik procesowania algorytmem get_edges (w dużej mierze alg. Canny'ego)

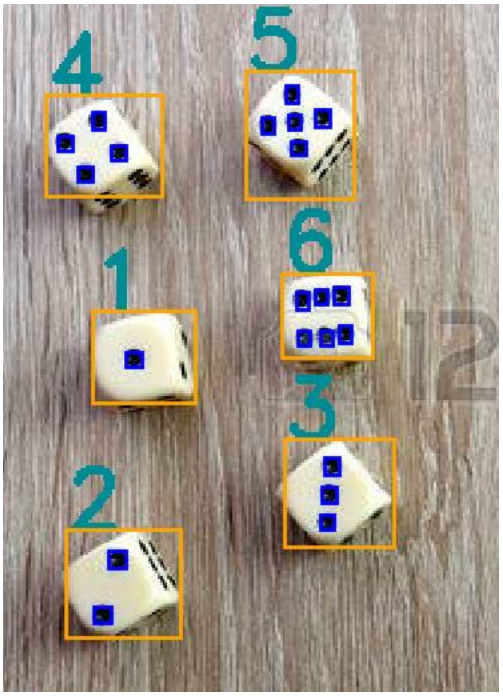
Rys. 38: Wynik procesowania kostki z rys. 37 przez alg. odnajdywania obiektów



Rys. 39: Wynik procesowania badanej kostki przez negative_on_v_and_canny

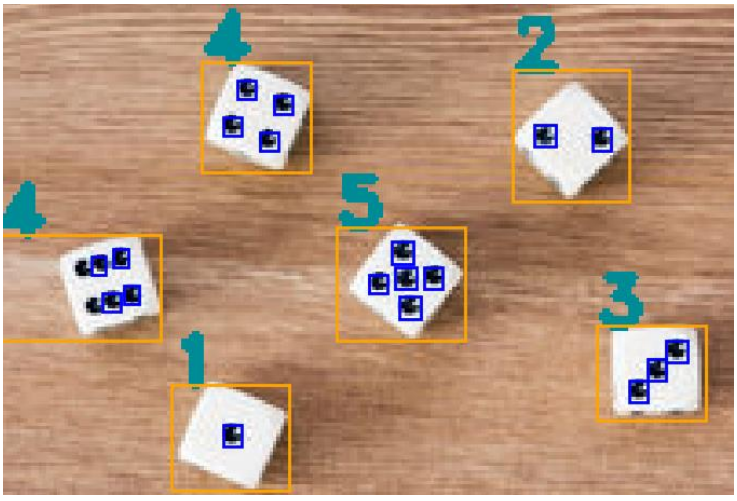


Rys. 40: Rezultat otrzymany przez stosowanie jedynie preprocesora obrazu get_edges



Rys. 41: Rezultat otrzymany przez zastosowanie filtru negative_on_v_and_canny

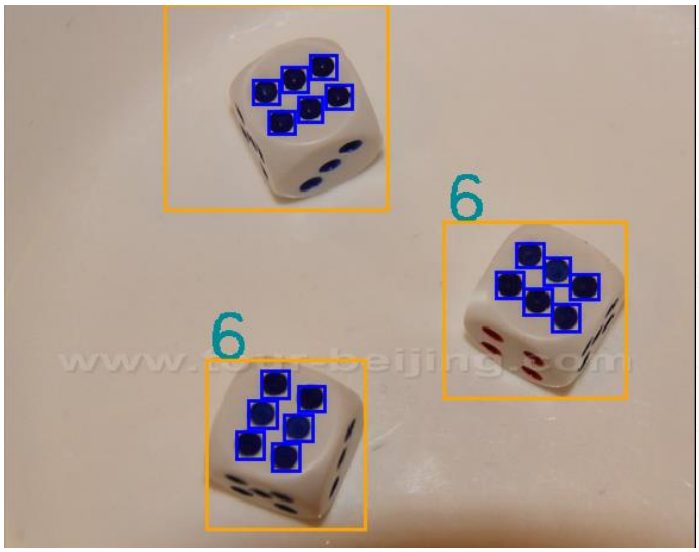
W ogólności jednak dodanie każdego nowego procesora obrazu wejściowego wymusza dodanie kolejnych filtrów lub manipulowaniem istniejących w związku z nasileniem dostarczanych kandydatów. Może to również skutkować pogorszeniem rezultatów dla innych przypadków testowych – przykładowo po dodaniu omawianego preprocesora na obrazie z rysunku 42 przestały być kwalifikowane dwie skrajne lewe kropki na lewej kostce ze względu na pogorszenie ich statystyki w stosunku do reszty stawki. Zwiększenie akceptowanego zasięgu zniwelowało problem, jednak daje również do myślenia – każdy preprocesor z jednej strony zwiększa nasz horyzont rozwiązań, jednak równocześnie wymusza ściślejsza definicje, czym tak naprawdę jest oczko oraz jakie powinno przyjmować parametry.

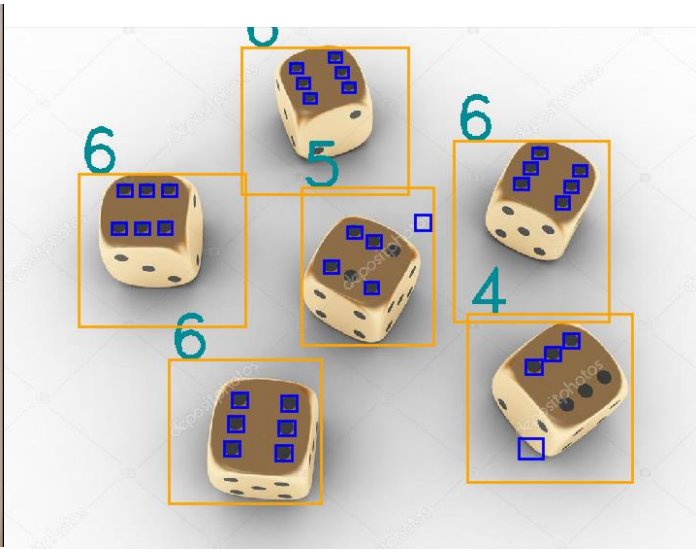


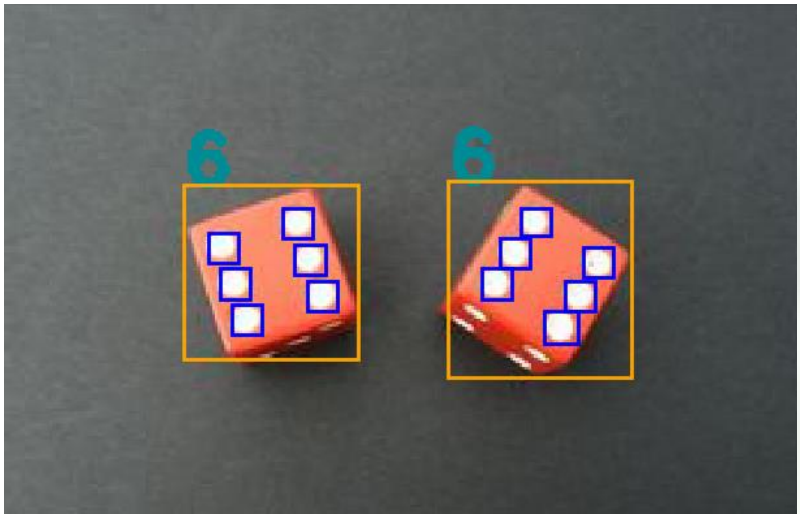
Rys. 42: Pogorszenie rezultatu przez dodanie nowego preprocesora

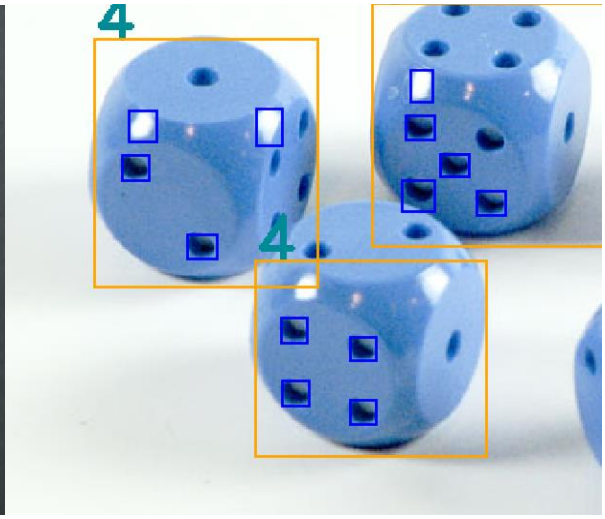
5. Wyniki algorytmu na przykładach testowych

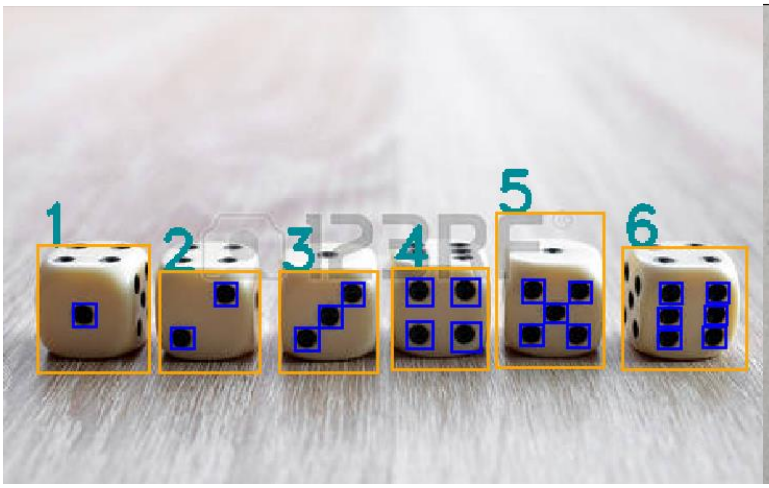
Poniżej znajdują się wcześniej nieuwzględnione przypadki testowe (część zdjęć została przycięta na potrzeby lepszego ulokowania).

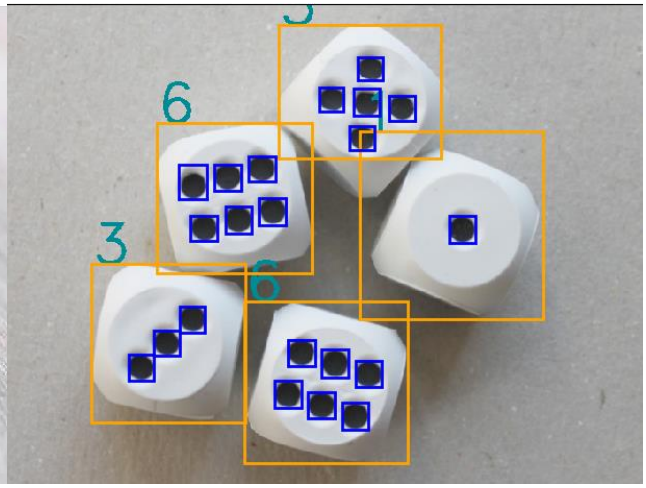


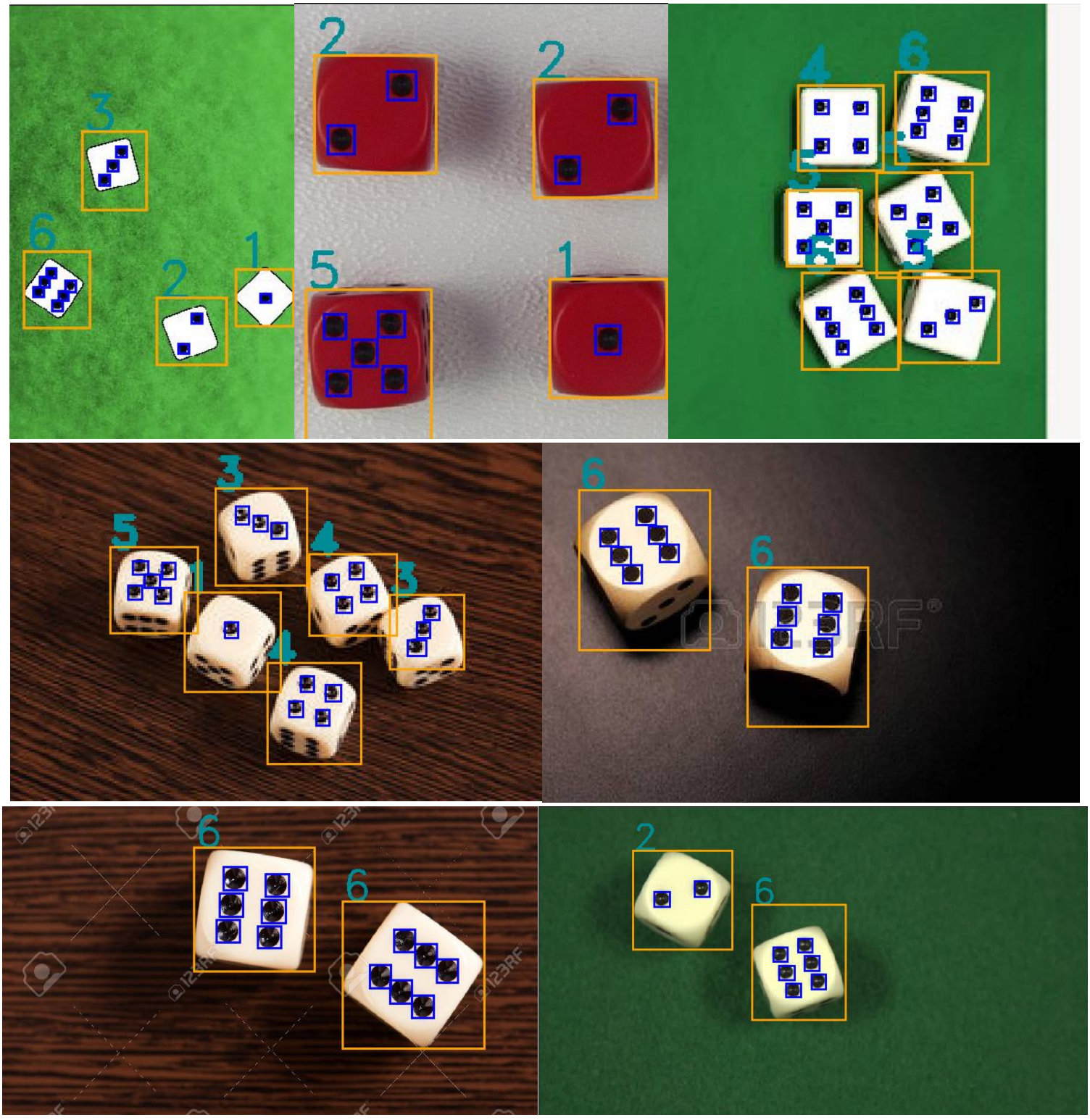











3
6
2
1
2
2
5
1
4
6
6
3
5
3
4
1
4
6
6
6
6
2
6