Wykrywanie kości do gry na zdjęciach

Witold Kupś

Politechnika Poznańska, Wydział Informatyki, 2017

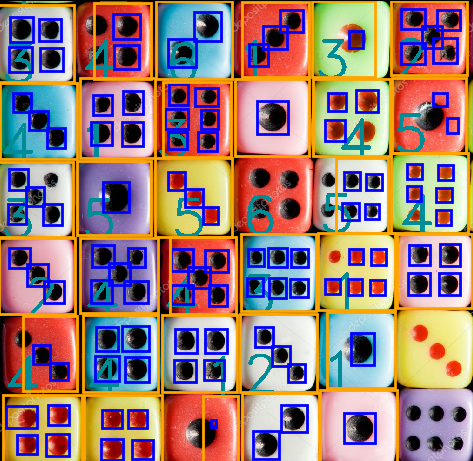
# Wprowadzenie

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

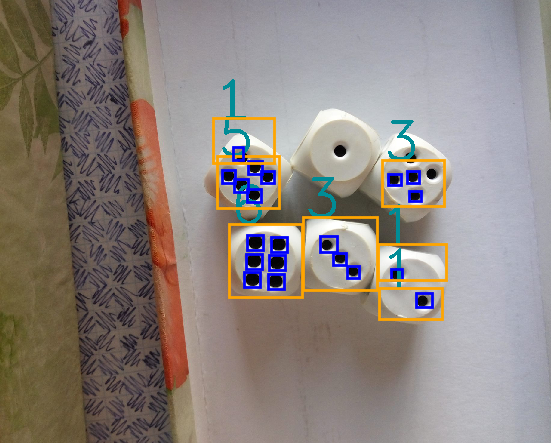
- nierównomierna wielkość: W przypadku, gdy na obrazie znajduje się wiele kości w perspektywie, lub różnej wielkości, ciężko stwierdzić które zostaną wykryte, a które uznane za szum. Algorytm bazuje bowiem na założeniu, że obiekty są podobnej wielkości (usuwane są wszystkie odstające w większym stopniu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

# Zawodność programu

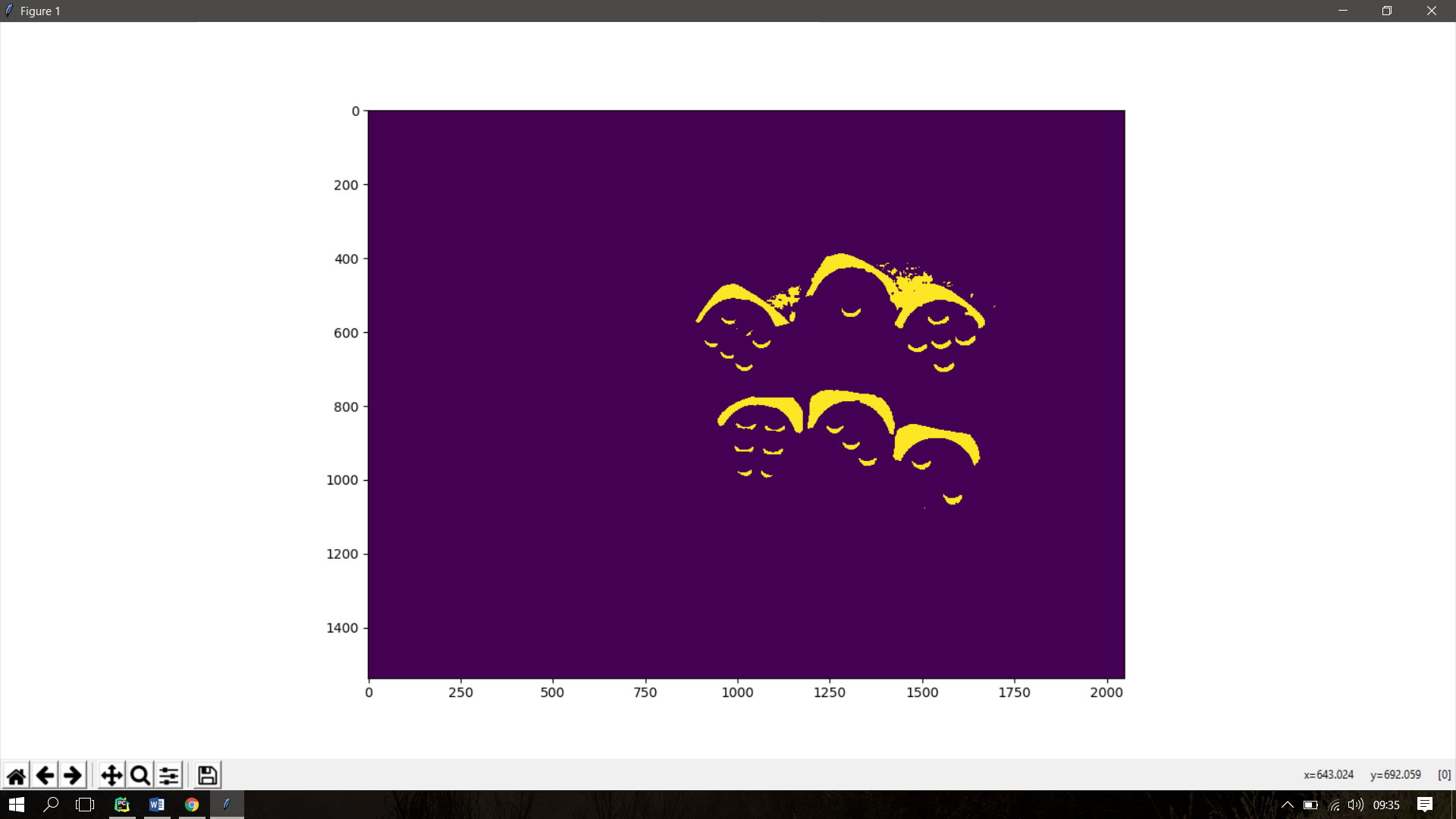
## Pozornie trywialne przypadki

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.

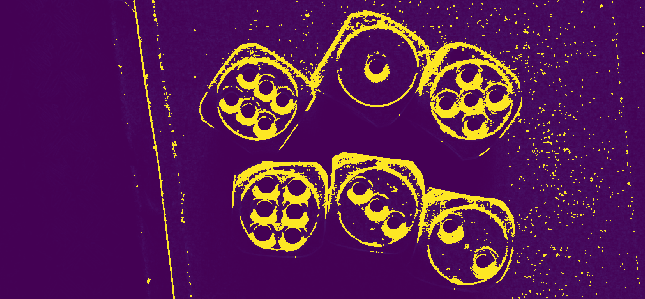
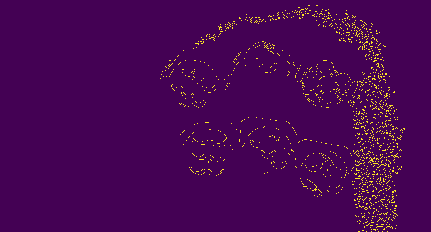
Wynik może być zaskakujący ze względu na jego niedoskonałość, jednak gdy spojrzymy na obraz zwrócony przez jedną z kilku operacji „wytrawiania” kości z rysunku, powód takiego zachowania wydaje się zrozumiały:



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

Cześć pozostałych przedstawiono poniżej:

Rys. 4: Część wyników pozostałych wyodrębniaczy.

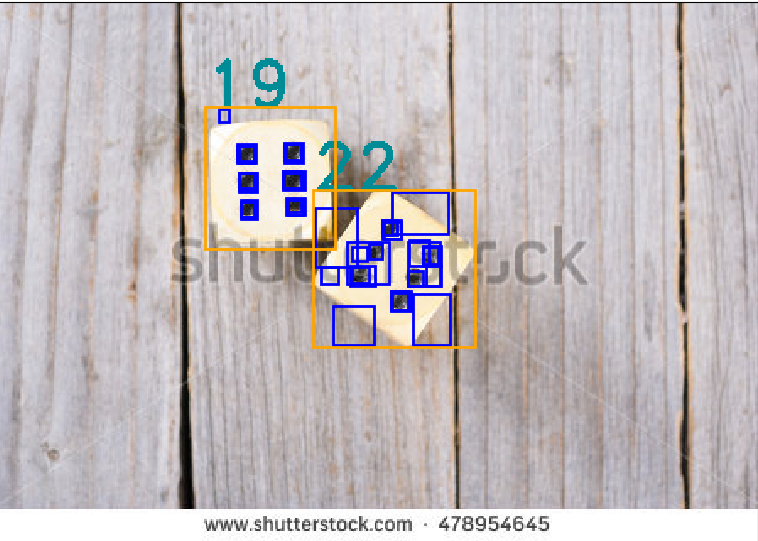
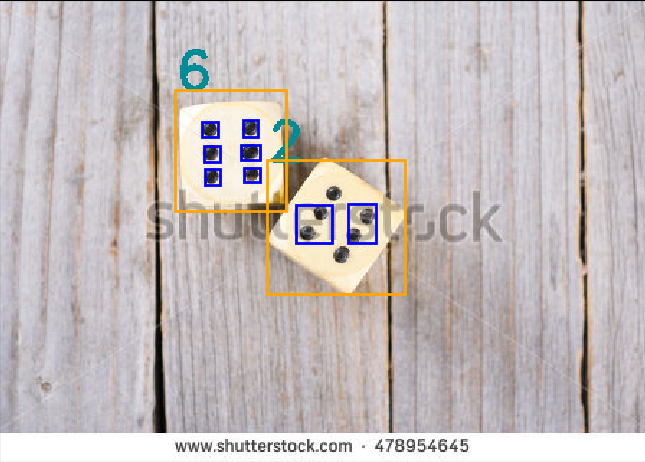
  

Jak więc widzimy, rezultaty są z goła odmienne. Gdyby zawarta była sztucna inteligencja, prawdopodobnie udałoby sie z obrazka znajdującego się w prawym dolnym rogu wyodrębnić ilość oczek, lub nawet całą kość na podstawie zagęszczenia znalezionych obiektów. Widzimy jednak, że kółka są nieregularne (przypominają 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

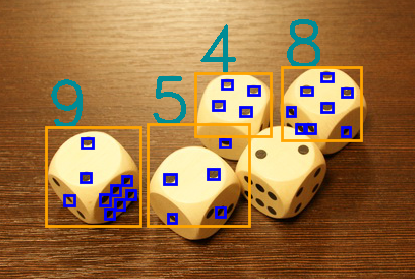
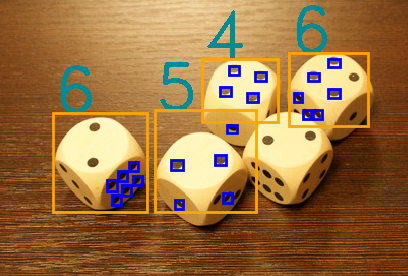
## Obsługa występujących na obrazie napisów – szumów

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



## 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.

## Różne poziomy naświetlenia

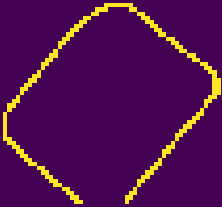
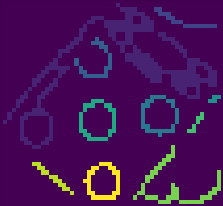
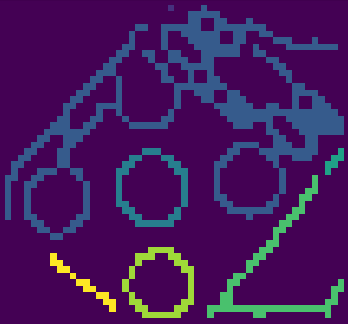
Zmiana poziomów natężenia światła jest problematyczna w ogólności, jednak w momencie, gdy nie mamy żadnego poziomu odwołania, bardzo ciężko jest ją zniwelować. W poprzednim podpunkcie analizowane było zdjęcie zawierające zarówno wysokie poziomy oświetlenia, jak i znaczne cienie. Same kości znajdowały 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.

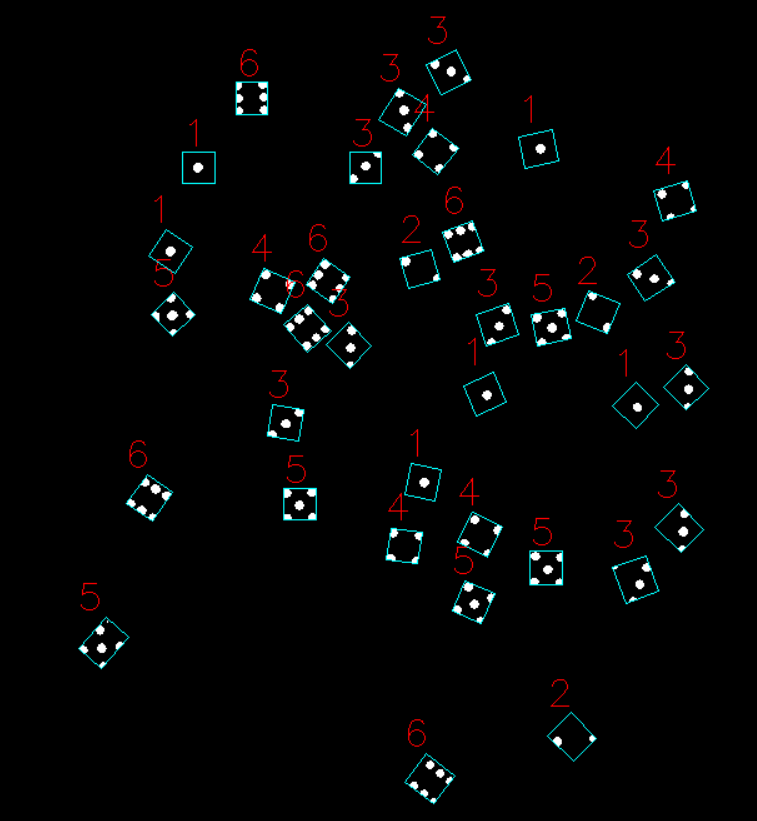
Rys. 7: Różne poziomy natężenia swiatła mogą spowodować błędne identyfikacje kości

## Bardzo małe obiekty

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

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

Rys. 9: surowe rezultaty wykrywania oczek na kości

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

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

# Działanie programu

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

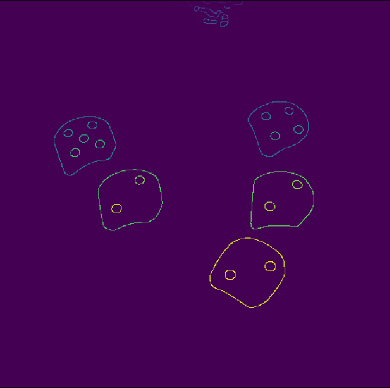
1. Przetworzenie dostarczonego obrazu pod kątem wykrycia dużych obiektów znajdujących się na nim, wyróżniających się od otoczenia.
2. Odfiltrowania uzyskanych regionów pod względem trafności, wypełnienia, stosunku długości krawędzi, nakładających się lub zbyt dużych/małych regionów oraz ogólnie statystyki uzyskanych rezultatów.
3. Poszukiwanie potencjalnych oczek w przestrzeni każdej z uzyskanych we wcześniejszym kroku kostek.
4. 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.

## 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.

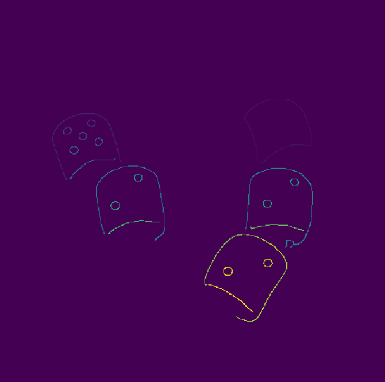
Rys. 13: Analizowany przykład



1. 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.

Rys. 14: Rezultat dla get\_edges przy wyszukiwaniu kości

1. 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. 16: Rezultat dla get\_by\_hsv\_value przy odnajdywaniu kości

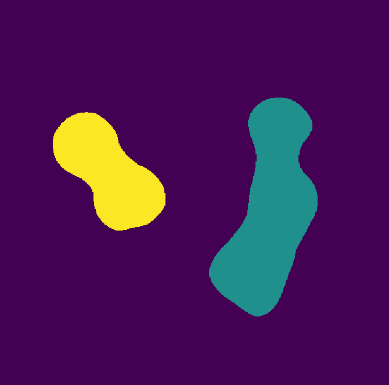
Rys. 15: Rezultat dla just\_canny\_and\_dilation przy wyszukiwaniu kości

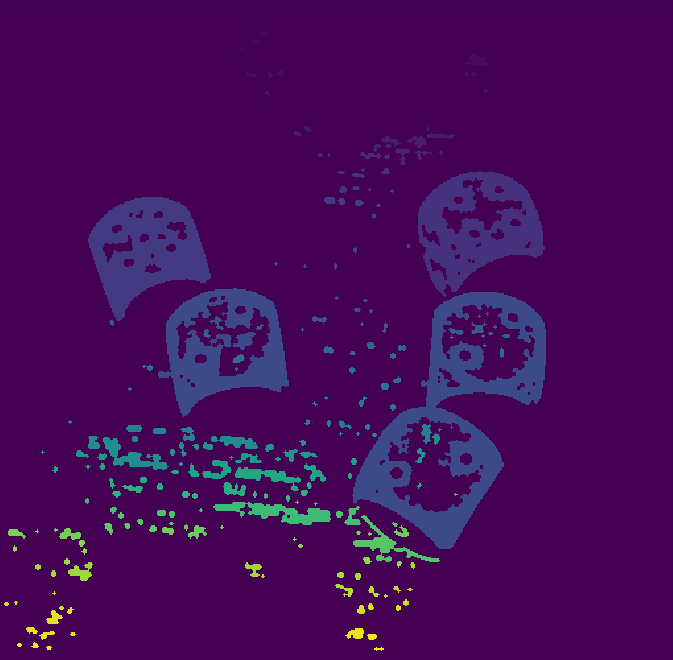
Rys. 17: Rezultat dla splashing\_image przy odnajdywaniu kości

1. 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)

1. 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

1. sobel\_and\_scharr\_connected – opiera się na algorytmie wykrywania krawędzi Sobel’a oraz scharr’a:

* obraz transformowany jest do monochromatu

Rys. 19: Rezultat dla high\_percentile\_shrink przy odnajdywaniu kości

* 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

1. 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.

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.

## Filtrowanie potencjalnych kości

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

1. W pierwszym kroku eliminowane są wszystkie oznaczone regiony, których bounding box(szerokość x wysokość) ma powierzchnię mniejszą niż 0.25% całkowitej powierzchni obrazu.
2. 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:

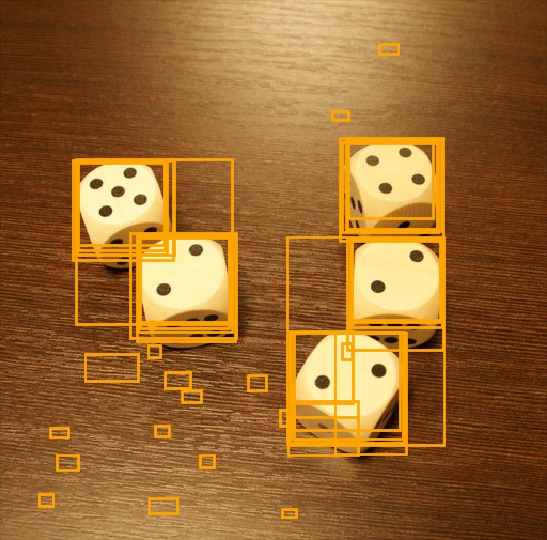
* progowaniu wartości rgb (średnio co 50),
* algorytmie określającym, czy obraz jest skontrastowany, znajdującym się w bibliotece scimage, w module exposure – is\_low\_contrast

1. 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.
2. W kolejnym kroku łączeni są wszyscy kandydaci, którzy się pokrywają oraz mają co najmniej 25% wspólnej powierzchni
3. 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).

## Odnajdywanie oczek na kościach

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



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

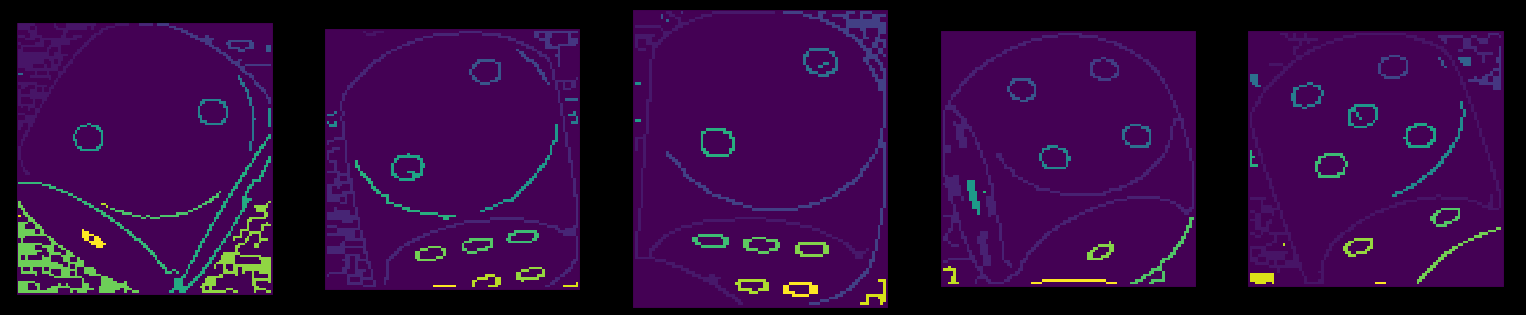
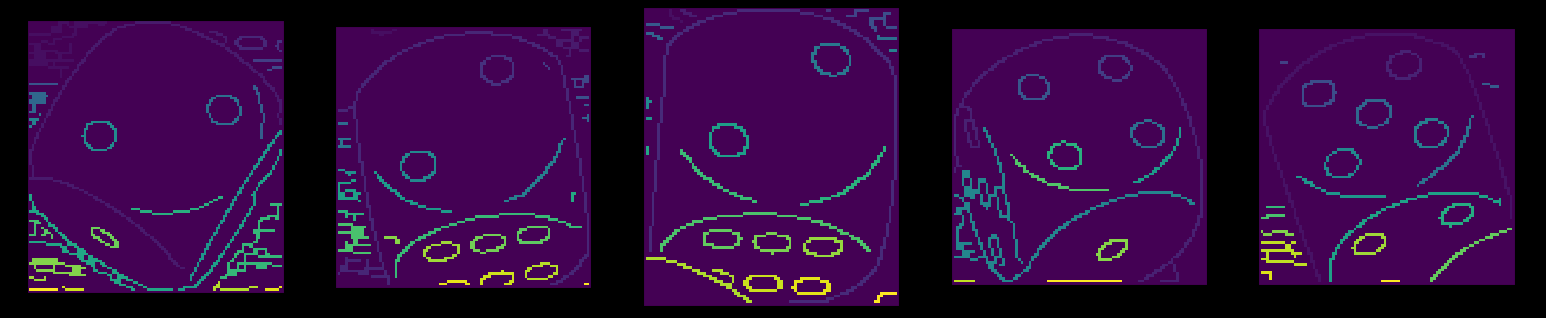
Tym razem operujemy na 4 algorytmach:

1. Get\_edges – ponownie, tym razem jednak gamma wynosi 1, sigma 1, bez ograniczeń odnośnie dolnych i górnych percentyli (0,100).
2. Just\_canny\_and\_dilation – z parametrami sigma 0.4, niskim progiem na 0.1 oraz wysokim na 0.3
3. 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

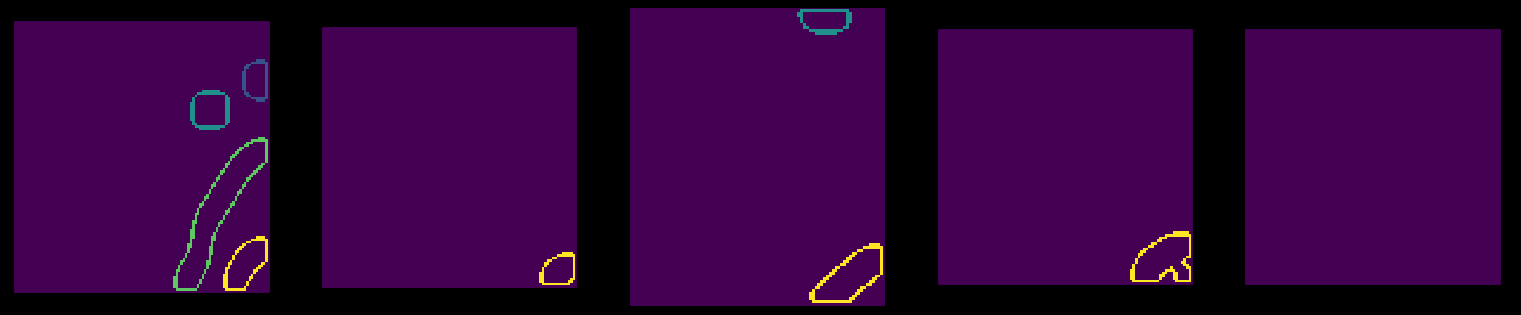
1. Negative\_on\_v\_and\_canny – filtr ten z kolei przeznaczony jest do wykrywania oczek na obrazkach o bardzo małej rozdzielczości oraz niskiej wyrazistości.

* 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.

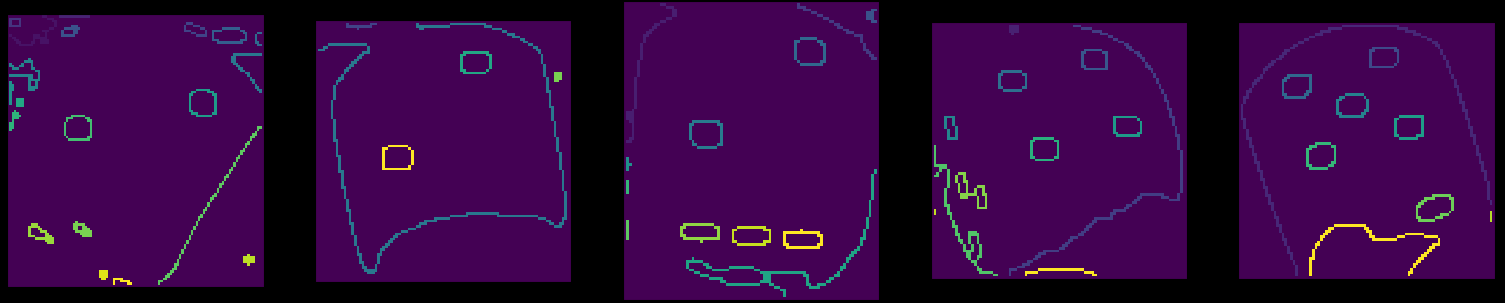


Rys. 21: Rezultat dla get\_edges przy wyszukiwaniu oczek

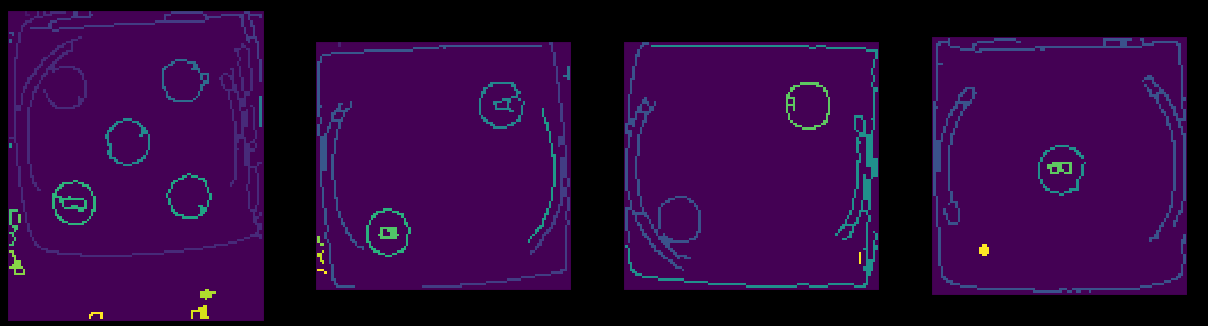
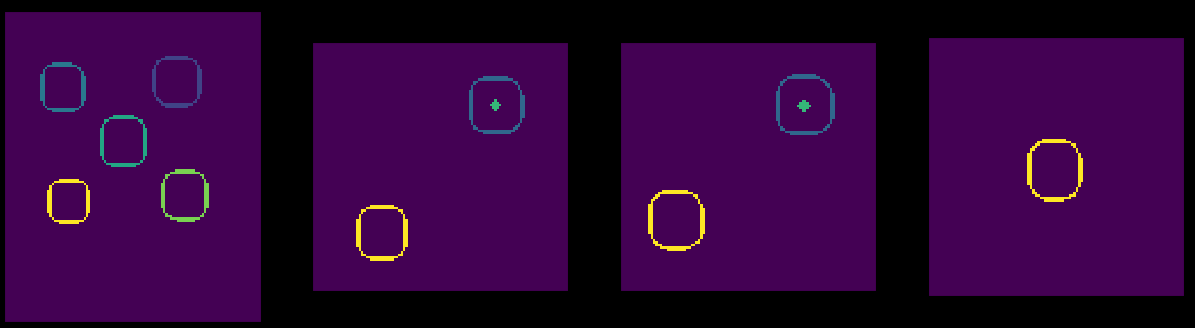
Rys. 22: Rezultat dla just\_canny\_and\_dilation przy wyszukiwaniu oczek



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. 27: Wynik przetworzenia przykładowego obrazu przez double\_sobel

Rys. 26: Wynik przetworzenia przykładowego obrazu przez get\_edges

Rys. 25: przykład dla weryfikacji działania double\_sobel

## Proces filtrowania oczek