

A G H

AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
WYDZIAŁ INFORMATYKI ELEKTRONIKI I TELEKOMUNIKACJI

INSTYTUT TELEKOMUNIKACJI

Projekt dyplomowy

Aplikacja mobilna wspierająca gracza w grze Scrabble
A mobile application supporting the player in the Scrabble game

Autor: Krzysztof Stanisław Pałka
Kierunek studiów: Teleinformatyka
Opiekun pracy: dr inż. Andrzej Matiolański

Kraków, 2022

Spis treści

1 Wstęp	3
1.1 Cel i motywacje pracy	3
1.2 Zasady gier scrabble	3
1.3 Skanowanie planszy	4
2 Powiązane prace	5
2.1 Programy wspomagające gracza	5
2.1.1 Słowniki	5
2.1.2 Wyszukiwarki słów	5
2.1.3 Scrabble solver	5
2.1.4 Scrabble solver z rozpoznawaniem obrazu	6
2.2 Przegląd rozwiązań algorytmicznych	6
2.2.1 Skanowanie planszy	6
2.2.2 Rozwiązywanie scrabbli	7
3 Implementacja	8
3.1 Architektura i dobór technologii	8
3.1.1 Mobilna część aplikacji	8
3.1.2 Serwerowa część aplikacji	9
3.2 Algorytm skanowania planszy	9
3.2.1 Pozycjonowanie obszaru roboczego	9
3.2.2 Klasyfikacja pól	13
3.3 Algorytm rozwiązywania scrabbli	15
3.3.1 Wyszukiwanie zorientowane na bloki liter	15
3.3.2 Budowa słownika	15
3.3.3 Wyszukiwanie potencjalnych słów	16
3.3.4 Filtrowanie możliwych do ułożenia słów	16
3.3.5 Pusta plansza	17
3.3.6 Braki algorytmu	17
3.3.7 Wydajność	17
3.4 Integracja części serwerowej	19
3.4.1 Kontroler	19
3.4.2 <i>ImageProcessingService</i>	20
3.4.3 <i>SolvingService</i>	20
3.4.4 Testy	20
3.4.5 Tworzenie logów	21
3.4.6 Konteneryzacja	21
3.5 Aplikacja mobilna	22
3.5.1 Struktura aplikacji	22
3.5.2 Strony	22
3.5.3 Interfejs użytkownika	24
3.5.4 Uruchamianie aplikacji	26
4 Porównanie	26
4.1 Funkcjonalności	26
4.2 Interfejs graficzny	27
4.3 Interakcje z użytkownikiem	27
4.4 Algorytmy	27
5 Podsumowanie	28
Bibliografia	30

1 Wstęp

Scrabble® jest znakiem towarowym popularnej gry słownej. Posiada 38 oficjalnych wersji językowych oraz drugie tyle nieoficjalnych wersji [25]. Istnieje wiele podróbek Scrabble®, wydawanych pod różnymi nazwami. Zgodnie ze słownikiem języka polskiego PWN, wszystkie te gry nazywane są powszechnie polskim słowem scrabble. Jest to jedna z najpopularniejszych gier słownych na świecie. Dodatkowo istnieje wiele gier podobnych do scrabble, zmieniając delikatnie zasady, ułożenie pól bonusowych, dozwolone słowa itp. (np. Literaki, Words With Friends). Jednak praca skupia się na grach z rodziną scrabble. Cały napisany kod oraz historia zmian wykonanego rozwiązania znajdują się w repozytorium git [20] na platformie GitHub.

1.1 Cel i motywacje pracy

Celem pracy jest zaprojektowanie i implementacja aplikacji mobilnej, pozwalającej graczowi na szybkie znalezienie dobrze punktowanych słów podczas realnej rozgrywki w grę scrabble. Opracowanie programu ma na celu pomóc graczom w znajdowaniu różnych słów, co pozwoli mu lepiej planować swoje ruchy i zwiększyć szanse na zwycięstwo. Aplikacja będzie przydatna dla początkujących graczy, którzy chcą nauczyć się gry w scrabble i poprawić swoje umiejętności. Sprawi, że gra w scrabble stanie się bardziej atrakcyjna dla szerszej grupy graczy, ponieważ ułatwi ona uczenie się nowych słów i strategii.

Aplikacja pozwoli użytkownikowi na zrobienie zdjęcia planszy oraz zaproponuje listę słów, które można ułożyć przy użyciu posiadanych liter. Wykonanie zdjęcia za pomocą aplikacji umożliwi skanowanie liter znajdujących się na planszy, aby zautomatyzować wprowadzanie informacji o aktualnym jej stanie. Aplikacja będzie działała z dowolną wersją planszy do gry. Zeskanowane dane zostaną przekazane algorytmowi, który na ich podstawie wyliczy zbiór możliwych ruchów. Dla każdego z nich zostanie obliczona wartość punktowa ruchu, a cała lista zostanie przedstawiona graczowi. Po wybraniu jednej z możliwości aplikacja przedstawi wizualnie gdzie na planszy ułożyć dane słowo oraz jakich liter użyć. Proces powinien być zautomatyzowany i przebiegać możliwie szybko, aby nie marnować czasu podczas rozgrywki. Powinna istnieć możliwość wyboru języka polskiego lub angielskiego oraz łatwy sposób dodania nowych języków z poziomu kodu aplikacji, w celu zwiększenia grupy docelowej użytkowników. Wymagania postawione przed aplikacją zostały zebrane w postaci czterech punktów:

- wybór jednego z dostępnych języków,
- skanowanie dowolnej planszy do gry z wykorzystaniem aparatu w telefonie,
- obliczenie listy słów, które można ułożyć,
- wyświetlenie wybranego słowa na planszy oraz zaznaczenie liter, których należy użyć do jego ułożenia.

1.2 Zasady gier scrabble

Gra polega na układaniu słów na planszy z dostępnych liter (przykład na rysunku 1). Każdy gracz otrzymuje na początku określona liczbę płyt z literami (zwykle siedem), które trzyma na swoim stojaku. Plansza składa się z siatki pól o wymiarach 15×15 , niektóre z nich zawierają bonusy (podwójna/potrójna premia literowa/słowna). Każda płytką zawiera jedną literę oraz przypisaną do niej wartość punktową. Wartość punktowa każdej z liter jest różna dla różnych języków i zależy odwrotnie proporcjonalnie od częstotliwości występowania tej litery w języku. Liczba płyt z daną literą zależy natomiast wprost proporcjonalnie od jej występowania. Dodatkowo istnieje określona liczba tak zwanych blanków, płytek, które mogą być traktowane jako dowolna litera. Gracze naprzemiennie układają po jednym słowie na planszy w każdej turze. Każde dokładane słowo musi być ułożone poziomo od lewej do prawej lub pionowo z góry na dół. Pierwsze słowo musi zaczynać się od środkowego pola, a każde następne musi być powiązane z już istniejącymi na planszy. Według oficjalnych zasad dla języka polskiego [23] istnieje pięć sposobów na dołożenie nowego słowa:

- dołożenie jednej lub kilku płytek na początku, lub na końcu słowa już znajdującego się na planszy
- ułożenie słowa pod kątem prostym do słowa znajdującego się na planszy, tak aby się przecinały
- ułożenie całego słowa równolegle do słowa już istniejącego w taki sposób, by stykające się płytki także tworzyły małe słowa
- nowe słowo może także dodać literę do istniejącego słowa
- most między co najmniej dwiema literami

Na planszy można układać tylko tak zwane legalne słowa. Są to wszystkie słowa znajdujące się w słowniku danego języka oraz ich poprawne formy gramatyczne, z wyjątkiem tych, które są skrótkami, przedrostkami lub przyrostkami albo zaczynają się od wielkiej litery, albo wymagają użycia apostrofu lub łącznika. Po zakończonym ruchu płytki na planszy nie mogą być zmieniane, a gracz dobiera z puli tyle płytek, ile wykorzystał. Jeśli gracz nie widzi słowa możliwego do ułożenia z posiadanych liter, może wymienić swoje płytki z literami na nowe z puli dostępnych płytek.

Gracze otrzymują punkty za ułożone słowa, a ich wartość zależy od wartości punktowej poszczególnych liter oraz od pól bonusowych na planszy, na których słowo jest ułożone. Jeżeli dołożone litery tworzą więcej niż jedno słowo, gracz otrzymuje punkty za każde z nich. Wszystkie punkty są sumowane. Gra kończy się, gdy wyczerpie się pula płytek lub gdy na planszy nie ma już miejsca na ułożenie kolejnych słów. Wygrywa gracz z największą liczbą punktów.



Rysunek 1: Przykładowa plansza scrabble w języku Polskim

1.3 Skanowanie planszy

Ważnym elementem pracy jest skanowanie planszy, czyli odczytywanie informacji ze zdjęcia fizycznej wersji gry. W tym celu wykorzystano przetwarzanie obrazu, czyli proces modyfikowania lub analizowania obrazu w celu zmiany jego wyglądu, lub uzyskania informacji. By realizować funkcjonalności z tym związane, został wykorzystany język Python oraz biblioteki przetwarzania obrazu i rozpoznawania tekstu.

OpenCV (Open Computer Vision) jest powszechnie używaną biblioteką do przetwarzania obrazu, ponieważ oferuje szeroki zakres narzędzi i funkcji. Oferuje proste funkcjonalności jak wczytywanie, zapis i wyświetlanie obrazu na dysku, oraz zaawansowane jak na przykład wykrywanie krawędzi, filtrowanie obrazu, detekcję twarzy i innych obiektów, segmentację obrazu itp. OpenCV jest biblioteką otwarto-źródłową (ang. open-source), co oznacza, że jest bezpłatna do użytku oraz może być

modyfikowana i rozpowszechniana zgodnie z warunkami licencji. OpenCV ma rozbudowaną i czytelną dokumentację oraz jest bardzo popularną i szeroko stosowana, przez co ma dużą społeczność użytkowników. Dostęp do społeczności jest pomocny przy rozwiązywaniu problemów.

OCR (Optical Character Recognition) to technologia pozwalająca na rozpoznawanie tekstu z obrazu. Dostarczony obraz przygotowywany, segmentowany na pojedyncze znaki lub grupy znaków, następnie są one rozpoznawane i umieszczone w odpowiedniej kolejności, tworząc tekst, który może być dalej przetwarzany lub edytowany. W pracy testowano dwie popularne biblioteki OCR dla języka Python: EasyOCR oraz PyTesseract.

2 Powiązane prace

Zważając na popularność gry, powstało wiele publikacji, artykułów naukowych i aplikacji o tematyce scrabble. W pierwszej części pracy podjęta została próba zebrania i uporządkowania informacji o istniejących aplikacjach, bibliotekach, algorytmach i artykułach mających związek ze wspomaganiem gracza w scrabbach.

2.1 Programy wspomagające gracza

Istnieje wiele programów mających cel podobny do założonego w ramach tej pracy - wspomóc gracza w realnej rozgrywce lub w nauce gry. Rozwiązania zostały zebrane i podzielone na cztery kategorie ze względu na stopień pomocy graczowi oraz, co za tym idzie złożoność systemu. Rozwiązania zostały uszeregowane od najprostszych do najbardziej skomplikowanych.

2.1.1 Słowniki

Słowniki są pasywnymi wspomagaczami pozwalającymi jedynie dokonać weryfikacji. Gracz musi sam odnaleźć jakie słowo może ułożyć oraz gdzie je wstawić na planszy, słownik pozwala jedynie zweryfikować czy dane słowo jest poprawne. Istnieje wiele aplikacji webowych oraz mobilnych tego typu [26, 29]. Organizacji, które publikują własne listy legalnych słów (np. Polska Federacja Scrabble), często udostępniają je w formie tego typu aplikacji.

2.1.2 Wyszukiwarki słów

Są to programy o małym stopniu skomplikowania, będące aktywnymi wspomagaczami. Polegają na przeszukiwaniu słownika pod kątem podanych kryteriów. Przyjmują jako parametr dostępne litery, często również prefix, infix, sufix oraz długość. W wyniku podają użytkownikowi słowa spełniające zadane kryteria. Narzędzia te nie uwzględniają stanu planszy, więc gracz sam musi znaleźć miejsce, w którym można ułożyć płytki z literami. W tej kategorii również istnieje wiele aplikacji webowych oraz mobilnych [2, 27, 28, 33]. Większość znalezionych rozwiązań wspiera jeden lub kilka z języków: angielski, francuski, niemiecki, hiszpański i włoski, rzadziej język polski.

2.1.3 Scrabble solver

Rozwiązania tego typu biorą pod uwagę stan planszy oraz litery dostępne na stojaku. Gracz sam uzupełnia zawartość planszy w systemie, dopisując kolejne pojawiające się na planszy słowa. Aplikacja musi implementować algorytm, który pozwala na wyznaczenie najlepiej punktowanego możliwego do ułożenia słowa - algorytm rozwiązujący scrabble. Zostały znalezione jedynie dwa rozwiązania tego typu:

- kod źródłowy aplikacji desktopowej napisanej we frameworku .NET, przez co możliwy do uruchomienia jedynie w systemie Windows [3].
- otwarto źródłowa strona internetową, udostępnioną dla użytkowników pod adresem scrabblesolver.org [17]. Pozwala na łatwe edytowanie planszy oraz stojaka i szybkie wyszukiwanie słów możliwych do ułożenia. Wspiera sześć wersji językowych. Strona jest wykonana bardzo przejrzysta i czytelna oraz intuicyjna w obsłudze. Nie jest responsywna, co utrudnia korzystanie z niej na urządzeniu mobilnym.

2.1.4 Scrabble solver z rozpoznawaniem obrazu

Jest to najbardziej zaawansowana kategoria, rozszerza poprzednią o algorytmy przetwarzania obrazu. Dzięki temu użytkownik nie musi wpisywać liter ręcznie - wystarczy wprowadzenie do aplikacji plik graficzny z planszą. Rozwiązania te można podzielić na dwie kategorie, według pochodzenia dostarczanego pliku graficznego.

Pierwszym sposobem jest dostarczenie **zrzutu ekranu z aplikacji**. Ma zastosowanie, gdy gracz uczestniczy w rozrywce w aplikacji mobilnej, takiej jak *Scrabble Go* lub *Words With Friends*. Znalezionych zostało kilka aplikacji tego typu, najlepszym przykładem jest *WordFinder by YourDictionary* [34] który wspiera 26 różnych gier mobilnych podobnych do scrabble. Wykrywanie planszy jest tutaj stosunkowo proste i bazuje na stałej pozycji planszy na ekranie. Wynika to z prostego eksperymentu, po drobnym przycięciu poprawnego zrzutu ekranu aplikacja informowała o błędzie.

Drugim możliwym rodzajem dostarczanego pliku graficznego jest **zdjęcie z aparatu**. Pozwala na używanie aplikacji podczas rozgrywki na fizycznej planszy do gry. Znaleziono dwa rozwiązania tego typu:

- kod źródłowy strony internetowej [22], która umożliwia przesłanie zdjęcia z urządzenia, na którym jest uruchomiona - sama w sobie nie pozwala jednak na wykonanie zdjęcia. Jest to jednak podejście mało przyjazne użytkownikowi, ponieważ musi on się przełączać między różnymi aplikacjami.
- aplikacja mobilna udostępniona w sklepie Google Play [14]. Jest to jedyne znalezione rozwiązanie, spełniające założenia wykonywanego projektu. Pozwala na zrobienie zdjęcia planszy i sama rozpoznaje płytki z literami, które się na niej znajdują. Następnie aplikacja rozwiązuje scrabble i wyświetla najlepiej punktowane słowa graczowi. Może działać w trybie offline, co znaczy, że wszystkie obliczenia prowadzone są na urządzeniu mobilnym.

2.2 Przegląd rozwiązań algorytmicznych

Aplikacje z ostatniej kategorii, aby realizować funkcjonalności, muszą posiadać algorytmy do odczytywania informacji o planszy ze zdjęcia, oraz wyznaczania najlepszego możliwego do ułożenia słowa. Każdy z nich jest osobnym, niełatwym problemem inżynierskim. Wykonany został przegląd prac i repozytoriów z kodem, które podejmują próbę rozwiązania zadanych problemów.

2.2.1 Skanowanie planszy

Skanowanie jest rozumiane jako proces składający się z wykrycia elementu na zdjęciu oraz odczytanie z niego informacji. Celem jest zeskanowanie ułożonych liter na fizycznej planszy do gry. Nie jest to temat popularny, znaleziono tylko cztery prace poruszające tę problematykę. Każdy z przeanalizowanych algorytmów składa się z kilku stałych elementów: plansza jest pozycjonowana oraz nanoszona jest siatka na granice między polami - dzięki temu można uzyskać dostęp do każdego z pól. Następnie każde pole jest klasyfikowane jako literę, blank lub puste miejsce. Różnice występują w sposobie pozycjonowania obszaru roboczego planszy i klasyfikowania liter. Należy podkreślić duży wpływ błędów pozycjonowania na klasyfikator. Przesunięcie planszy o zaledwie 3% jej długości powoduje przesunięcie każdego pola o 50% i ucięcie połowy litery. W konsekwencji poprawna klasyfikacja praktycznie nie jest możliwa. Dlatego też bardzo ważna jest jak największa precyzja pierwszej części algorytmu.

Praca Davida Koeplingera [16] dobrze definiuje cel i trudności skanowania planszy oraz proponuje ogólne kroki algorytmu. Pozycjonowanie obszaru roboczego miałoby zostać zrealizowane za pomocą wyrycia prostych linii pomiędzy polami na zdjęciu, a klasyfikowanie za pomocą metody porównywania do płytEK wzorcowych. Brak w niej jednak implementacji oraz weryfikacji działania rozwiązania.

W pracy Zach Witzel i Xiluo He [35] autorzy rozpoznają stan planszy w dwóch grach: scrabble oraz set. Do pozycjonowania wykorzystano wykrywanie krawędzi *Canny*, transformację linii *Hough*, oraz klasteryzację planszy. Do klasyfikacji płytEK użyto splotowej sieci neuronowej. W testach całosciowych (pozycjonowanie obszaru roboczego oraz klasyfikacja znaków) uzyskują 93.10% skuteczności. Zbiór uczący i testowy zostały wygenerowane komputerowo za pomocą oprogramowania Blender

[9]. Mimo użycia różnych kątów kamery oraz zmiennego oświetlenia renderowane grafiki są dalekie od rzeczywistych zdjęć, a tło pod planszą jest jednolite. Autorzy zdają sobie sprawę, że w realnych warunkach ich rozwiązywanie może mieć dużo gorszą skuteczność. Wiele elementów (na przykład nie-istotne linie w tle) może zaburzyć kadrowania planszy. Zauważają również, że drobne rozbieżności w pierwszym etapie mają duży wpływ na późniejsze rozpoznawanie znaków, co przekłada się na niższą ostateczną skuteczność.

David Hirschberg w swojej pracy [15] jako cel przyjmuje stworzenie algorytmu do użycia w asystentach scrabble. Do pozycjonowania zostało wykorzystane wykrywanie krawędzi Canny, operacje morfologiczne na wykrytych krawędziach [19], oraz wyodrębnienie największego jednolitego obiektu. Następnie za pomocą algorytmu k najbliższych sąsiadów ze wzorcem każdej z możliwości w dobrej rozdzielczości została przeprowadzona klasyfikacja płytek. Otrzymano 94% skuteczności w pozycjonowaniu obszaru roboczego oraz 67% skuteczności w testach całościowych. Testy zostały przeprowadzone jedynie na siedmiu zdjęciach zawierających całą planszę z delikatnie zmienną perspektywą i z brakiem rotacji. Z tego powodu otrzymane wartości nie są wierzytelne.

Ostatnie rozważane rozwiązywanie [24] zostało przedstawione w formie 3-częściowej filmowej demonstracji. Twórca stworzył algorytmy pozwalające na skanowanie planszy oraz stojaka. Do pozycjonowania obszaru roboczego planszy została użyta metoda detekcji i opisu cech oraz dopasowanie do wzorca [18]. Następnie znaleziono homografię i skorygowano perspektywę, uzyskując obraz, na którym znajduje się tylko obszar roboczy planszy. Do klasyfikacji płytek użyto splotowej sieci neuronowej. Sieć neuronowa osiągnęła ponad 99.9% skuteczności na 400 zdjęciach po 26 liter na każdym. Brakuje testu całościowego, jednak podczas prezentacji widać, że pozycjonowanie obszaru roboczego działa bardzo dobrze. Zasadniczą wadą tego rozwiązania jest współpraca jedynie z jednym rodzajem planszy, przez bezpośrednie dopasowywanie do wzorca.

2.2.2 Rozwiązywanie scrabbli

Celem gry jest zdobycie jak największej ilości punktów, więc gracze chcą zwykle znaleźć jak najlepiej punktowane słowo. Nie jest to jednak optymalna taktyka w każdym przypadku. Istotne jest również długoterminowe myślenie strategiczne pomiędzy turami, zarządzanie posiadanymi literami, blokowanie przeciwników czy śledzenie jakie litery są na planszy, w worku i u przeciwnika. Algorytm grający w scrabble ma więc dwa dające się wyróżnić elementy. W pierwszej kolejności algorytm wyznacza słowa możliwe do ułożenia przy danym stanie planszy i stojaka, następnie dokonuje decyzji o wyborze słowa lub wymianie liter. Dokonanie wyboru jest często poruszane w pracach dotyczących scrabble, na przykład przystępnej publikacji o programie MAVEN [30]. Ten temat wykracza jednak poza zakres pracy i nie będzie rozwijany. Ta część pracy skupia się na rozwiązywaniu scrabbli, czyli znalezieniu możliwych do ułożenia słów.

The World's Fastest Scrabble Program [1] to artykuł naukowy z 1988 roku, szeroko cytowany w ramach tej tematyki. Autory wskazują, że efektywny generator wszystkich legalnych ruchów sam w sobie jest nietykalny do zaprogramowania i jest przydatny jako podstawa dla automatu grającego w scrabble. Stworzyli oni algorytm oraz zaimplementowali program w języku C rozwiązuje ten problem. W artykule znajdują się fragmenty pseudokodu i dokładny opis działania. W porównaniu do istniejących w tamtym czasie rozwiązań zyskali oni dużą wydajność, dzięki użyciu drzewa przedziałowego do reprezentacji słownika. Algorytm wyszukuje na planszy "kotwice", czyli miejsca, do których można dołożyć nowe słowo, aby nie przeszukiwać całej planszy. Redukuje to problem trójwymiarowy, do dwuwymiarowego, co znaczco optymalizuje czas działania. Działając na słowniku z 94240 słowniami, czas wyznaczenia wszystkich ruchów wynosił w ich przypadku 2-3 sekundy.

Znaleziono dwie **biblioteki** pozwalające na rozwiązywanie scrabbli. Obydwie zostały napisane w języku C++ oraz ich kod jest dostępny na platformie GitHub.

- Words with Bots [21] to biblioteka z interfejsem użytkownika w linii komend. Pozwala zarządzać stanem planszy oraz stojaka przechowywanych w pliku, oraz rozwiązywać scrabble.
- Unofficial Scrabble Library [13] jest bardzo wydajną biblioteką. Autorzy podają, że średni czas przetwarzania dla planszy stojaka bez blanków oraz planszy z pięcioma słowami wynosi około

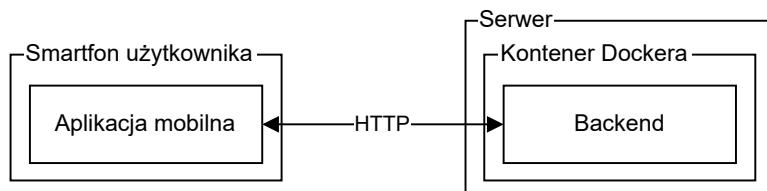
0.18 sekundy. Program również można uruchomić z linii komend. Jest dobrze opisana i zadbaana, przez co wydaje się najlepszą opcją do użycia jako element innego programu.

3 Implementacja

Celem pracy było zbudowanie przenośnej aplikacji mobilnej wspomagającej gracza w fizycznej wersji scrabble. Użytkownik robi za pomocą aplikacji zdjęcie planszy oraz wpisuje listę liter ze stojaka. Następnie otrzymuje listę możliwych do ułożenia słów, posortowanych według kryteriów: najlepiej punktowane lub najdłuższe. Domyślnie dostępne są dwa języki (polski i angielski), można także używać innego języka po wgraniu słownika. W ten sposób postawiony problem jest bardzo skomplikowany, podzielono go więc na dające się wyodrębnić mniejsze elementy: aplikacja mobilna, algorytm skanowania planszy, algorytm rozwiązywania scrabble oraz łącząca je w działającą całość aplikacja serwerowa.

3.1 Architektura i dobór technologii

Pierwszą decyzją, jaką podjęto, był **wybór architektury aplikacji**. Obliczenia, takie jak przetwarzanie obrazu czy rozwiązywanie scrabbli, mogą być wykonywane na telefonie użytkownika, lub w aplikacji serwerowej. Urządzenie mobilne jest mniej wydajne niż urządzenia serwerowe. Obliczenia po stronie klienta, będą prowadzić do zużycia baterii oraz zmniejszenia wydajności innych programów działających w tle na sprzęcie użytkownika i samej aplikacji. Dodatkowo uruchamianie różnego rodzaju bibliotek (np. OpenCV) lub skryptów w innych językach niż sama aplikacja (np. Python, C++) na sprzęcie mobilnym często stanowi problem inżynierski. Natomiast przy aplikacji wykonującej obliczenia po stronie serwera, użytkownik musi mieć zawsze połączenie z internetem, a przesyłanie danych jest opatrzone pewnym opóźnieniem. Dodatkowo należy zaimplementować i utrzymywać aplikację serwerową. Rozważając powyższe za i przeciw wybrano architekturę z obliczeniami po stronie serwera ze względu na prędkość wykonywania oraz prostotę uruchamiania skryptów. Architektura została przedstawiona na rysunku 2. Kolejnym krokiem jest wybór technologii, które zostaną użyte podczas implementacji rozwiązania.



Rysunek 2: Diagram architektury aplikacji

3.1.1 Mobilna część aplikacji

Aplikację mobilną składającą się na rozwiązanie problemu można budować, opierając się na jednej z trzech architektur [12]:

- aplikacja natywna jest tworzona dla konkretnego systemu operacyjnego, w języku Java dla systemu Android, w Objective-C lub Swift dla IOS. Kod napisany dla jednego systemu, nie będzie działał w drugim.
- mobilna aplikacja webowa działa jak miniaturowa przeglądarka internetowa, wyświetlająca kod pisany w technologiach webowych. Wadą tego rozwiązania jest dużo mniejsza wydajność i ograniczenia sprzętowe.
- architektura hybrydowa pozwala tworzyć jeden kod dla wielu platform, przy jednoczesnej dobrej wydajności. Dzięki natywnym interfejsom można w łatwy sposób używać na przykład kamery urządzenia.

Do projektu została wybrana ostatnia kategoria ze względu na wieloplatformowość. Dwa najpopularniejsze frameworki hybrydowe to React Native oraz Flutter [31]. Używają odpowiednio języka JavaScript oraz Dart. Jak pokazują badania, odznaczają się podobną wydajnością, oraz łatwością wytwarzania oprogramowania [32]. Wybrany został React Native ze względu na wykorzystanie powszechnie używanego języka JavaScript oraz popularnej biblioteki React.

3.1.2 Serwerowa część aplikacji

Komunikacja między telefonem użytkownika a serwerem została zrealizowana za pomocą REST API, jako że jest to najczęściej stosowany sposób komunikacji między elementami aplikacji. Część serwerowa (ang. backend) została zaimplementowana w języku Java 17 za pomocą frameworka Spring Boot w wersji drugiej, ze względu na jego popularność oraz prostotę tworzenia REST API. Do funkcjonalności skanowania planszy użyto biblioteki OpenCV, zważając na powszechnie użycie w tego typu rozwiązaniach oraz dobrą dokumentację. Jest napisana w C/C++ jednocześnie oferując interfejsy dla innych języków programowania (Python, Java, JavaScript). Z testów wynika, że komunikacja z biblioteką za pomocą interfejsu ma marginalny wpływ na wydajność w porównaniu do używania jej bezpośrednio w C/C++ [10]. Z tego względu skrypty przetwarzające obraz zostały napisane w języku Python, ze względu na prostotę prototypowania oraz testowania kodu. Funkcjonalność rozwiązywania scrabbli została zaimplementowana w Javie, ze względu użycie tego języka do implementacji backendu.

Zaplecze serwerowe zostało skonteneryzowane, oznacza to umieszczenie całego środowiska potrzebnego do jej uruchomienia w pojedynczym, izolowanym kontenerze. Dzięki temu aplikacja może być łatwo przenoszona i uruchamiana w różnych środowiskach, bez konieczności dostosowywania jej do konkretnych systemów operacyjnych czy innych zależności. Konteneryzacja zapewnia więc większą elastyczność i niezawodność aplikacji oraz ułatwia zarządzanie nią. W tym celu zostało użyte narzędzie Docker.

3.2 Algorytm skanowania planszy

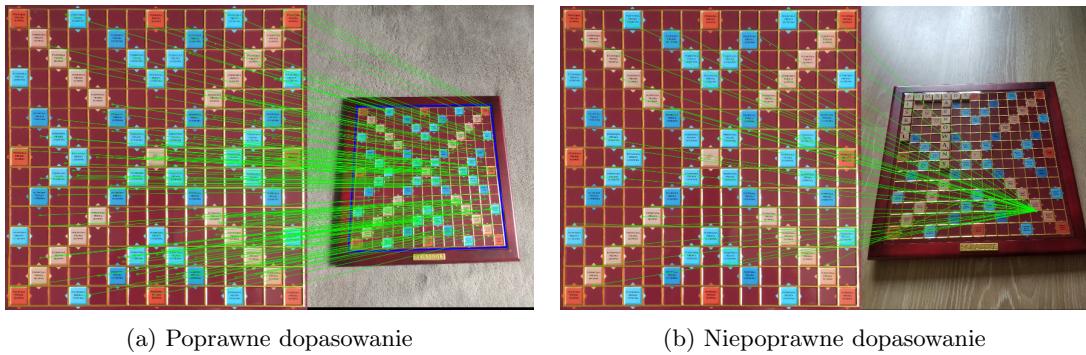
Skanowanie planszy składa się z dwóch etapów: znalezienia planszy na zdjęciu oraz odczytania informacji o literach lub ich braku. Jest to zadanie trudne do wykonania, ze względu na dużą różnorodność istniejących plansz do gry. Plansza składa się z obszaru roboczego (siatka pól o wymiarach 15x15) oraz dodatkowych elementów dookoła jak logo, napisy i grafiki. Gra scrabble występuje w wielu wariantach kolorystycznych oraz używa różnych czcionek. Może być wykonana z papieru, drewna, plastiku, metalu lub ich kombinacji - niektóre wersje powodują dużo refleksów świetlnych na zdjęciu. Płytki mogą mieć od milimetra do 7 milimetrów wysokości i być wczepiane do obszaru roboczego lub swobodnie układane - przez co mieć zmienną pozycję. Algorytm został zrealizowany w języku Python, przy użyciu biblioteki OpenCV [7]. Pochodzą z niej wszystkie opisywane poniżej funkcje, metody i klasy.

3.2.1 Pozycjonowanie obszaru roboczego

Celem tej części algorytmu jest wyodrębnienie ze zdjęcia jedynie obszaru roboczego, tak aby pola z potrójną premią słowną znajdowały się dokładnie w rogach wyciętego zdjęcia.

Pierwszym podejściem do problemu był algorytm dopasowywania do wzorca - użyty z sukcesem w czwartej pracy z przeglądu algorytmów [24]. Metoda wymaga posiadania wzorca obszaru roboczego lub całej planszy, co jest jej dużą wadą, ponieważ pozwala na rozpoznawanie tylko znanych rodzajów wersji gry. Na wykonanym zdjęciu oraz wzorcu wyszukiwane są deskryptory, następnie używając klasy *BMatcher* i metody *knnMatch* wyszukiwane są dopasowania między deskryptorami. Na podstawie najlepszych dopasowań funkcja *findHomography* szuka najlepiej pasującego przekształcenia, które jest używane w funkcji *warpPerspective* w celu wyodrębnienia obszaru roboczego. Przy bardzo zbliżonym oświetleniu i pustej planszy, algorytm działa dobrze. Jednak przy różnicy w oświetleniu lub występowaniu płytka na planszy, ilość poprawnych dopasowań drastycznie spadała i uniemożliwiała na prawidłowe wykrycie obszaru roboczego. Problemem okazało się też wielokrotne powtarzanie się wzorów na obszarze roboczym, co utrudniało znajdowanie dopasowań pomiędzy deskryptorami. Drobne błędy między dopasowaniami powodowały duże błędy w znajdowaniu przekształcenia,

w wyniku obraz nie nadawał się do dalszego przetwarzania. Na rysunku 3 przedstawiono poprawne dopasowanie oraz niepoprawne dopasowanie spowodowane powtórzeniem wzorca.



Rysunek 3: Dopasowanie do wzorca za pomocą wykrywania i dopasowywania cech

Ostatecznie zaimplementowano metodę bazującą na pracy Davida Hirschberga z przeglądu algorytmów [15]. Zdjęcia dostarczane z aparatu mogą być różnej rozdzielczości, więc pierwszym krokiem jest normalizacja wymiarów w taki sposób, aby mniejszy był zawsze równy 1000 pikseli - współczynnik skalowania jest zapisywany. Na rysunkach 4 i 5 przedstawiono procesy znajdowania wierzchołków zakończone odpowiednio sukcesem i niepowodzeniem. Elementy na rysunku odpowiadają kolejnym, wymienionym poniżej etapom. Pierwszą częścią algorytmu jest **wyznaczanie maski obszaru roboczego**:

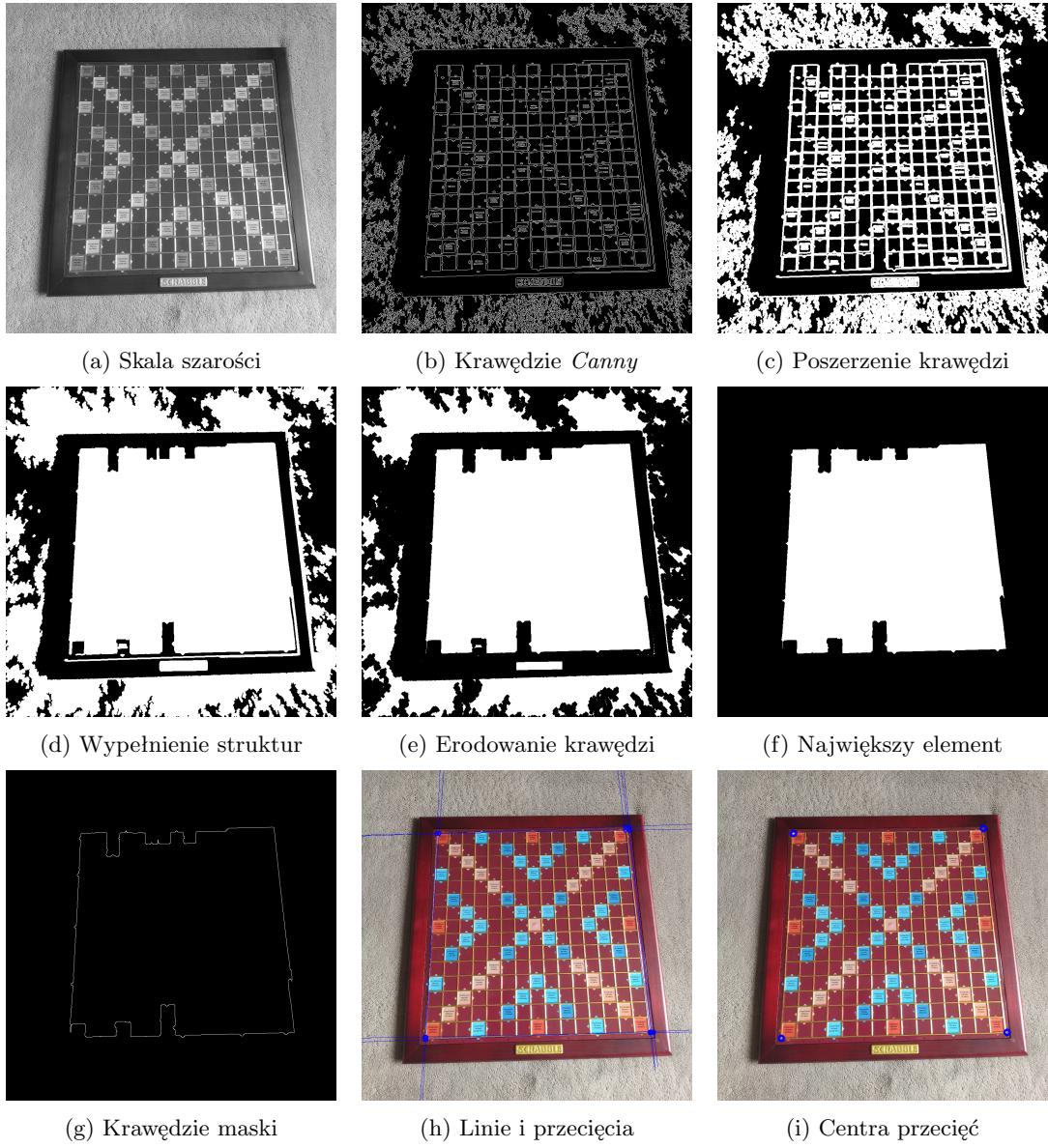
- (a) Konwersja obrazu na skalę szarości
- (b) Rozmycie drobnych nierówności za pomocą *GaussianBlur* i wykrywanie krawędzi funkcją *Canny*. Dzięki rozmyciu wykrywane są tylko wyraźne krawędzie. W dalszym przetwarzaniu obraz jest de facto macierzą o wartościach binarnych
- (c) Poszerzanie krawędzi, powoduje połączenie znajdujących się blisko siebie linii
- (d) Wypełnianie wnętrza wszystkich zamkniętych struktur za pomocą funkcji *findContours* oraz *drawContours* (z parametrem *thickness*=-1)
- (e) Erodowanie krawędzi jądrem o większym promieniu niż operacja poszerzenia. Po tej operacji na obrazie zostaną tylko struktury, które były zamkniętymi lub prawie zamkniętymi obiektami
- (f) Ponowne wykrycie konturów za pomocą *findContours*, wybranie jednego o największym polu i wypełnienie go w środku powoduje powstanie obszaru roboczego

Następnie odbywa się **określanie pozycji wierzchołków** na podstawie wyznaczonej maski:

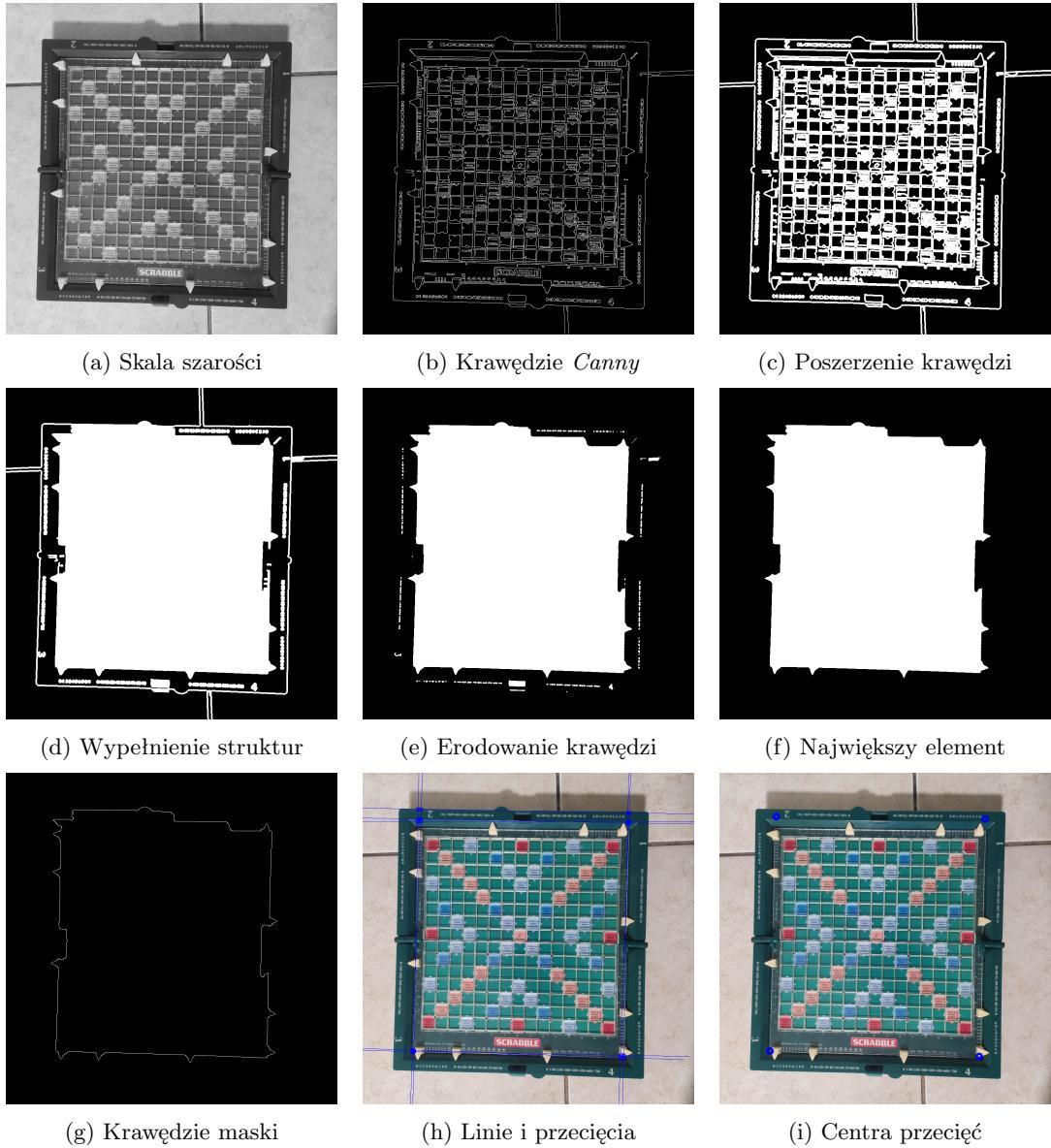
- (g) Zamiana maski na jej krawędzie przy użyciu funkcji *Canny*
- (h) Wykrycie prostych za pomocą transformacji *Hougha*. W idealnych warunkach powinny zostać wykryte dokładnie cztery proste. Następnie wyznaczane są współrzędne przecięć prostych
- (i) Wyznaczone współrzędne są rozdzielane na cztery grupy, ze względu na odległość od rogów grafiki. Dla każdej z grup liczona jest średnia obydwu współrzędnych

Wyznaczone współrzędne są skalowane do rozmiaru oryginalnego zdjęcia za pomocą zapisanego wcześniej współczynnika. Funkcja *minAreaRect* wylicza optymalną rozdzielcość wynikowej grafiki - taką aby zminimalizować konieczność skalowania. Następnie wyliczana jest macierz transformacji i za pomocą metody *warpPerspective* wycinany jest obszar roboczy z oryginalnego zdjęcia. W efekcie powstaje kwadratowa grafika w całości zajmowana przez obszar roboczy planszy.

Przeprowadzono testy metody na 40 zdjęciach łatwiej do wykrycia planszy wykonanych pod różnym kątem, rotacją i odlegością od obiektu, oraz stałym tłem. Z każdego ze zdjęć zostały precyjnie



Rysunek 4: Proces poprawnego wykrycia wierzchołków obszaru roboczego



Rysunek 5: Proces niepoprawnego wykrycia wierzchołków obszaru roboczego

pobrane współrzędne rogów obszaru roboczego. Następnie mierzono odległości między pobranymi danymi a wartościami wyznaczonymi przez algorytm. Wyniki przedstawiono w tabeli 1. Średnia wartość błędu wyniosła 25 pikseli. Natomiast średnia długość krawędzi obszaru roboczego wyniosła 2072 piksele. Tak więc średnio błąd wyznaczania wierzchołka wyniósł 1.20% długości krawędzi.

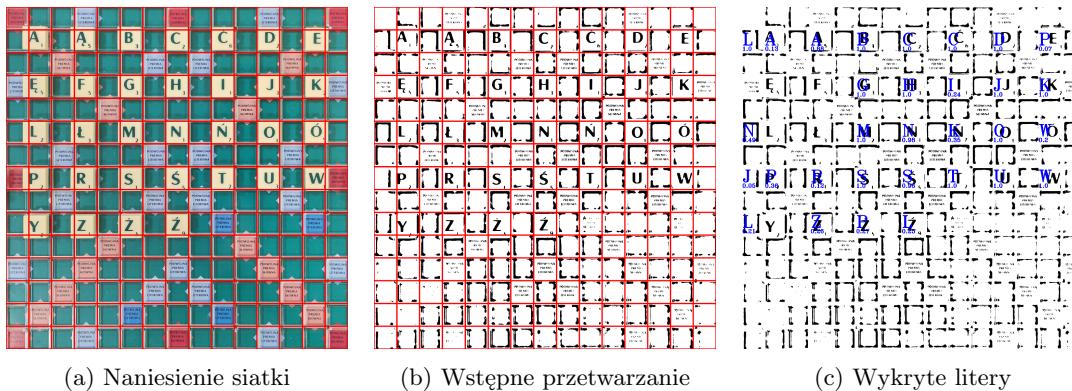
Tabela 1: Test poprawności pozycjonowania obszaru roboczego

	Wartość błędu	Długość planszy
Średnia wartość	25	2072
Odcylenie standardowe	27	217
Minimalna wartość	1	1653
Maksymalna wartość	221	2557

Na zdjęciach testowych metoda miała dużą dokładność, jednak przy testach w praktyce na trudniejszych wersjach planszy, zmiennym otoczeniu oraz oświetleniu, błędy były na nieakceptowalnym poziomie. Działo się tak z przynajmniej dwóch powodów. Najczęściej maska nie była czworokątem i miała zbyt dużo artefaktów w postaci ubytków lub przyrostów, przez co transformacja *Hougha* zwracała dużo błędnych linii. Sposobem na poprawienie wyników może być inna metoda wyznaczania wierzchołków maski, mniej wrażliwa na artefakty. W niektórych sytuacjach wykrywana była cała plansza do gry, zamiast obszaru roboczego. Jest to duża wada wynikająca z samego działania metody, co powoduje, że nie jest ona uniwersalna. Słabe wyniki algorytmu w realnych warunkach wymagały dodania w aplikacji możliwości poprawienia pozycji wierzchołków.

3.2.2 Klasyfikacja pól

Podczas tego etapu, każde z pól powinno zostać zakwalifikowane jako puste lub jedna z liter. Obszar roboczy został wycięty, więc każde z pól może zostać wyodrębnione ze zdjęcia za pomocą znanych współrzędnych. Na tej samej zasadzie można nanieść na zdjęcie siatkę, co przedstawiono w podpunkcie (a) na rysunku 6. W idealnych warunkach naniesione linie powinny dokładnie pokrywać się z granicami pól na zdjęciu, a litery powinny być dokładnie w centrum. Jednak przesunięcie rogu obszaru roboczego o zaledwie 3% jego długości, oznacza przesunięcie najbliższych pól o 50% ich długości. Powoduje to ucięcie połowy litery lub przesunięcie artefaktów w miejsce, w którym powinna być litera. W takiej sytuacji prawidłowe zakwalifikowanie jest bardzo trudne. Problemem jest też wysokość płytek w niektórych wersjach planszy oraz efekt "rybiego oka" często występujący w aparacie. Perspektywa powoduje, że nadrukowana litera jest na granicy lub poza polem nawet w przypadku prawidłowego pozycjonowania obszaru roboczego. Widać to na przykład przy literze "A" na rysunku 6.



Rysunek 6: Przykład klasyfikacji pól na poprawnie pozycjonowanej planszy

Każde pole należy zakwalifikować jako puste lub jedną z liter alfabetu. Może występować również blank, jednak w ogólnym przypadku dowolnej planszy, jest on bardzo trudny do wykrycia. Z tego względu z założenia blanki muszą zostać uzupełnione przez użytkownika w aplikacji mobilnej. Na początku wykonywana jest **wstępna kwalifikacja**. Każde pole kwalifikowane jest do jednego z dwóch stanów: na pewno nie litera lub prawdopodobnie litera. Zdjęcie jest wstępnie przetwarzane poprzez konwersję do skali szarości, filtr bilateralny, progowanie adaptacyjne oraz erodowanie pozostacych konturów. W idealnym przypadku po tych operacjach puste pola powinny być zupełnie białe, a niepuste powinny zawierać jedynie literę. W praktyce napisy na polach bonusowych, boki płytek i granice pól pozostawiają sporo artefaktów. Następnie zliczana jest ilość czarnych pikseli w centrum pola. Jeżeli jest mniejsze niż pewien próg, pole zaliczane jest jako puste. Jeżeli większe, prawdopodobnie znajduje się na nim płytki.

Następnie uruchamiany jest **algorytm OCR**. Każde pole rozpoznane jako "prawdopodobnie litera" musi zostać ostatecznie zakwalifikowane. Do tego celu zostały użyte dwie, otwarto źródłowe biblioteki: PyTesseract [8] oraz EasyOCR [6]. Każda z nich może rozpoznawać litery z wielu różnych alfabetów. Dla obydwu bibliotek można skonfigurowano język oraz listę dozwolonych znaków (tylko duże litery danego alfabetu). W PyTesseract dodatkowo ustawiono typ segmentacji na 10 - rozpoznawanie pojedynczych znaków. W obu przypadkach zwracana jest lista rozpoznanych znaków z poziomem pewności dla każdego z nich - wybierany jest znak z najwyższym poziomem pewności. Natomiast pole kwalifikowane jest jako puste, jeżeli na liście nie ma żadnego znaku. Uzależnienie oznaczenia jako puste od konkretnego progu poziomu ufności mija się z celem. Jak widać w podpunkcie (c) na rysunku 6, niektóre z poprawnie rozpoznanych liter mają niską pewność, a niektóre niepoprawnie rozpoznane bardzo wysoką.

Przeprowadzono testy kwalifikacji pól, w celu weryfikacji skuteczności oraz porównania obydwu silników. Przygotowano planszę z wszystkimi literami języka polskiego o znanych pozycjach jak na rysunku 6. Zrobiono 40 zdjęć pod różnymi kątami nachylenia, różnej odległości, przy realistycznym oświetleniu (pozostawiającym cienie i refleksy) - otrzymane zdjęcia należą do trudnych do rozpoznania. Dla każdego ze zdjęć dokonano ręcznego pozycjonowania obszaru roboczego oraz wykonywano algorytm klasyfikacji pól. Zliczano wyniki wstępnej kwalifikacji (niezależnej od silnika OCR) oraz ilość pól rozpoznanych poprawnie i błędnie, oraz różne podzbiory tych kategorii. Wyniki przedstawiono jako procent wszystkich przetworzonych pól. Na podstawie zebranych danych obliczono, ile procent liter algorytmy OCR rozpoznały poprawnie. Dodatkowo zmierzono średni czas trwania całej kwalifikacji. Wyniki przedstawiono w tabeli 2.

Tabela 2: Test poprawności wstępnej klasyfikacji oraz porównanie poprawności klasyfikacji przy użyciu EasyOCR i PyTesseract

Poprawność wstępnej kwalifikacji	96.344 %	
Nieoczekiwane puste	1.2 %	
Nieoczekiwana litera	2.456 %	
Silnik OCR	EasyOCR	PyTesseract
Poprawnie:	90.344 %	90.489 %
- Poprawnie litera	6.822 %	6.967 %
- Poprawnie puste	83.522 %	83.522 %
Niepoprawnie:	9.656 %	9.511 %
- Niepoprawnie rozpoznane litery	6.044 %	5.222 %
- Nieoczekiwane puste	1.356 %	2.033 %
- Nieoczekiwana litera	2.256 %	2.256 %
Ogólnie poprawnie sklasyfikowanych liter	53.02 %	57.16 %
Średni czas przetwarzania jednego zdjęcia [s]	4.446	17.377
Odcchylenie standardowe czasu [s]	0.37	1.084

Wstępna kwalifikacja płytek jest poprawna w 96.3%. W 2.4% pól algorytm zakwalifikował puste pole jako literę, a dla 1.2% przypadków literę jako puste pole. Obydwie biblioteki OCR mają zbliżone

statystyki. Dla każdej z nich średnio na planszy około 90% pól będzie miało poprawne wartości, natomiast ponad 50% z występujących liter będzie rozpoznanych poprawnie. PyTesseract o 0.7 punktu procentowego częściej niż EasyOCR oznaczał pole zawierające literę jako puste. Natomiast EasyOCR o podobną wartość częściej źle rozpoznawał literę. Silniki mylą się podobnie często, jednak popełniają błędy innego typu. Występuje duża różnica czasu przetwarzania pomiędzy algorytmami OCR. EasyOCR jedną planszę średnio przetwarza ponad 4 sekundy, natomiast PyTesseract ponad 17 sekund. Różnica wynika z czasochłonnego wstępnie przygotowywania modelu, które w bibliotece EasyOCR, dzięki przygotowaniu obiektu, można wyciągnąć przed pętle iterujące po polach. W bibliotece PyTesseract rozpoznawanie jest wywoływanie za pomocą pojedynczej funkcji, przez co wstępne przygotowywanie modelu musi się wykonywać przy każdym polu od nowa.

Oba algorytmy mają bardzo zbliżoną poprawność klasyfikacji, różnice występują w sposobie, w jaki się mylą. Z tego względu na płaszczyźnie rezultatów nie można powiedzieć, że jeden z algorytmów działa lepiej lub gorzej. Jednak ze względu na czas przetwarzania, EasyOCR jest lepszą opcją do użycia w algorytmach, w których rozpoznawanie jest wywoływanie wielokrotnie. Najważniejszym wnioskiem z porównania jest jednak fakt, że silniki OCR służące do rozpoznawania tekstu na obrazach lub skanowania dokumentów, mają niską skuteczność przy rozpoznawaniu pojedynczych liter. Szczerze gólnie gdy dookoła występuje wiele artefaktów mogących zaburzać rozpoznawanie. Potencjalnym kierunkiem dalszego rozwoju algorytmu może być użycie do kwalifikacji splotowej sieci neuronowej, wytrenowanej na dużej ilości danych. Mogliby ignorować nieistotne elementy i rozpoznawać litery również korzystając z informacji o wartości punktowej litery znajdującej się na zdjęciu. Minusem tego rozwiązania jest jednak konieczność wyuczenia sieci osobno dla każdego z obsługiwanych języków, jeżeli zawierają specyficzne dla siebie znaki diakrytyczne.

3.3 Algorytm rozwiązywania scrabbli

Celem algorytmu jest znalezienie jak największej ilości słów, możliwych do ułożenia przy użyciu posiadanych liter. Gracz w jednym ruchu może układać słowa pionowo (z góry na dół) lub poziomo (od lewej do prawej) w jednej kolumnie lub jednym rzędzie. Wyszukiwanie słów jest więc prowadzone poprzez iterowanie po każdym rzędzie i kolumnie. Następnie znalezione słowa są łączone w jedną listę. Zaimplementowane jest jedynie przetwarzanie dla kolumn, dla rzędów obliczenia są prowadzone dzięki transpozycji planszy. Redukuje to problem dwuwymiarowy, do jednowymiarowego w obrębie pojedynczej kolumny.

3.3.1 Wyszukiwanie zorientowane na bloki liter

Zdecydowaną większość legalnych ruchów można wykonać, dokładając litery do istniejącego słowa, wydłużając je lub przecinając pod kątem prostym. W każdej kolumnie wyszukiwane są więc bloki, czyli nieprzerwane ciągi ułożonych liter. Blok jest miejscem zaczepienia nowego słowa. Długość bloku jest z zakresu od pojedynczej litery (dla słowa przecinanego pod kątem prostym) do długości najdłuższego słowa w słowniku (dla słowa ułożonego w danej kolumnie). Dla każdego znalezionej bloku powstaje zbiór liter do wykorzystania, czyli suma zbiorów liter z bloku i liter posiadanych przez gracza - przechowywany jako posortowana alfabetycznie lista liter w zmiennej *lettersToUse*. Na bazie tych liter wyszukiwane są słowa potencjalne, czyli takie, które można z nich ułożyć. Da przykładu, potencjalnymi słowami dla liter [M, M, A, A] są [MAMA, MAM, MAA, AM, MA, AA]. Problem znalezienia słów potencjalnych sprowadza się do iterowania po słowniku ze wszystkimi poprawnymi słowami oraz zakwalifikowaniu każdego z nich jako możliwego do ułożenia (podzbioru posiadanych liter) lub niemożliwego. Ten element jest wykonywany wielokrotnie podczas jednego uruchomienia algorytmu, z tego względu został odpowiednio zoptymalizowany.

3.3.2 Budowa słownika

Słownik bazuje na liście wszystkich legalnych słów w danym języku. Przed uruchomieniem rozwiązywania scrabble, budowany jest obiekt klasy *Dictionary*, zawierający wstępnie przetworzone dane słownika. Słowniki budowane są dla każdego języka oddzielnie, na podstawie listy słów z pliku tekstowego. Słownik składa się z następujących struktur danych:

- *requiredLettersToWordsMap* - mapa gdzie kluczem są posortowane alfabetycznie litery, a wartością lista słów, które się z nich składają. Na przykład: "AŁMOS" → ["MASŁO", "SŁOMA", "SMOŁA"]
- *sortedRequiredLetters* - lista posortowanych liter z *requiredLettersToWordsMap*, posortowana po długości zbioru
- *sizeToWordsIndex* - mapa gdzie kluczem jest długości zbioru liter, a wartością indeks w *sortedRequiredLetters*, na którym występuje pierwszy element o danej długości

3.3.3 Wyszukiwanie potencjalnych słów

Celem jest znalezienie wszystkich słów, możliwych do ułożenia z liter z listy *lettersToUse*. Aby to osiągnąć, wystarczy znaleźć wszystkie klucze *requiredLettersToWordsMap*, które są podzbiorami *lettersToUse*. W tym celu iterowana jest lista *sortedRequiredLetters*, od długości bloku liter + 1 (nie można utworzyć słowa krótszego lub równego długości bloku, dla którego jest prowadzone wyszukiwanie), do długości *lettersToUse* (nie można utworzyć słowa dłuższego niż wszystkie dostępne litery). Indeksy dla iterowania "od-do" pobierane są ze struktury *sizeToWordsIndex*. Dla każdego z elementów iterowania wykonywana jest funkcja, sprawdzająca czy klucz jest podziobrem zbioru *lettersToUse*. Działa szybko, ponieważ uwzględnia, że oba zbiory są posortowane alfabetycznie - przy dowolnej niezgodności przetwarzanie jest przerwane.

```
protected static boolean isSubsetOf(char[] A, char[] B) {
    int pointerB = 0;
    int pointerA = 0;

    while (pointerA < A.length) {
        if (pointerB >= B.length) return false;
        while (A[pointerA] > B[pointerB]) {
            pointerB++;
            if (pointerB >= B.length) return false;
        }
        if (A[pointerA] == B[pointerB]) {
            pointerA++;
            pointerB++;
        } else return false;
    }
    return true;
}
```

Jeżeli funkcja zwróci prawdę, to znaczy, że klucz jest podziobrem liter do użycia, wtedy za pomocą klucza pobierana jest lista słów z mapy. Wszystkie otrzymane listy są łączone w jedną.

3.3.4 Filtrowanie możliwych do ułożenia słów

Otrzymane w poprzednim kroku słowa są możliwe do ułożenia ze względu na litery składowe. Nie są jednak umiejscowione na planszy i nie wiadomo czy na pewno da się je fizycznie ułożyć. W tym celu słowo jest sprawdzane pod kątem pasowania do bloku. Dla każdego z potencjalnych słów wyszukiwane są w nim wystąpienia zawartości bloku. Wystąpień może być kilka - dla każdego z nich zwracane jest osobne słowo możliwe do ułożenia. Na przykład do jednoliterowego bloku "M" słowo "MAMA" można dołożyć na dwa sposoby. Jeżeli nie znaleziono wystąpienia, słowa nie da się ułożyć. Na przykład dla bloku o zawartości "SOK" i posiadanej litery "Y" potencjalnym słowem będzie "KOSY" - nie da się go jednak ułożyć. Zwrócone słowa będą już obiekty zawierającymi ciąg znaków, współrzędne na planszy i kierunek układania. Następnie sprawdzane jest dopasowanie do otoczenia na planszy. Jeżeli słowo nachodzi na inną strukturę, weryfikowana jest zgodność nowych i istniejących liter na konkretnych pozycjach. Jeżeli styka się z innymi wyrażeniami (poza całą długością bloku),

sprawdzane jest, czy nowo powstały ciąg znaków tworzy legalne słowo. Jeżeli tak, takie dodatkowe słowo jest zapisywane do obiektu w celu doliczenia za nie punktów.

W ten sposób otrzymywana jest lista możliwych do ułożenia słów. Dla każdego z nich obliczana jest wartość punktowa, uwzględniająca pola punktowe, ich zajętość, dodatkowe ułożone słowa i bonus za wykorzystanie wszystkich liter. Następnie lista jest sortowana według kryteriów.

3.3.5 Pusta plansza

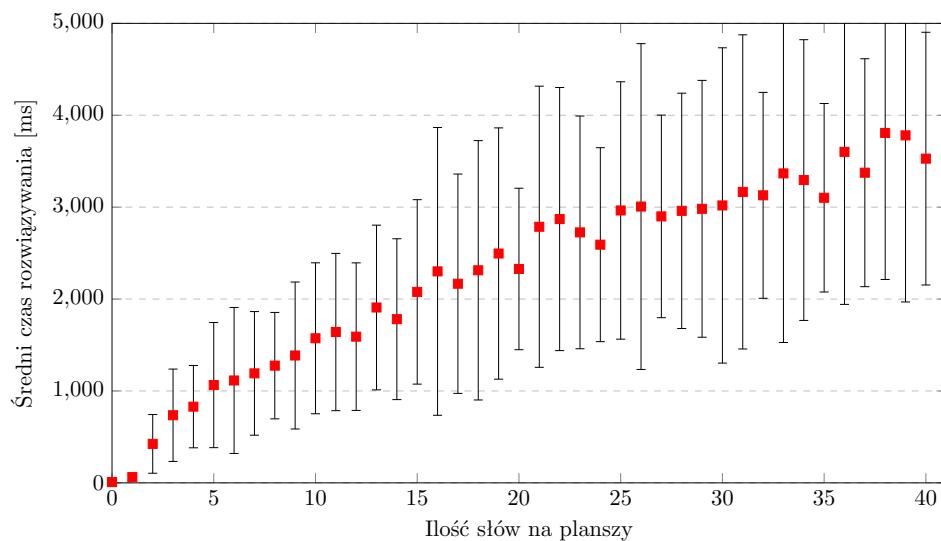
Tak opisany algorytm nie zadziała dla przypadku brzegowego, jakim jest **pusta plansza**. W tym celu został dodany wyjątek, który w takiej sytuacji wykorzysta kod generujący potencjalne słowa, bazując jedynie na literach posiadanych przez użytkownika. Wszystkie wygenerowane słowa ustawiane są pionowo w centrum planszy, zaczynając od środkowego pola. W ten sposób przypadek zostaje obsłużony.

3.3.6 Braki algorytmu

Tak zaimplementowany algorytm ma jednak kilka braków. Przede wszystkim nie pozwala na używanie **blanków** - struktura algorytmu oparta na wyszukiwaniu potencjalnych słów wymagałaby znacznych zmian i utraty wydajności, aby zapewnić wsparcie dla tego elementu. Dodatkowo nie wykrywa słów możliwych do dołożenia równolegle do już istniejącego słowa oraz mostów. Są to dwa z pięciu możliwych sposobów na ułożenie słowa według zasad scrabble [23], w realnej rozgrywce występują dosyć rzadko. Mimo to mogą to być najlepiej punktowane słowa w danym scenariuszu.

3.3.7 Wydajność

Czas wykonywania pojedynczego rozwiązywania jest bardzo zróżnicowany, dla pustej planszy jest niemal natychmiastowy, dla zapełnionej wykonuje się nawet kilkanaście sekund. Przeprowadzono symulację polegającą na wylosowaniu zawartości stojaka, rozwiązaniu scrabbli, dodaniu najlepszego znalezionejgo słowa do planszy. Zaczęto od pustej planszy i powtórzono proces 40 razy, mierząc czas każdego kolejnego rozwiązywania. Symulację powtórzono 50 razy. Średni czas przetwarzania kolejnych słów został zobrazowany na rysunku 7. Czas rozwiązywania zależy od wielu czynników takich jak długość słów na planszy, ich rozmieszczenie, występujące litery (dla tych częściej pojawiających się w języku przetwarzanie będzie trwało dłużej). Z tego względu dla różnych plansz o takiej samej liczbie słów, odchylenie standardowe czasu przetwarzania jest bardzo duże.



Rysunek 7: Uśredniony czas rozwiązywania scrabbli dla 50 symulacji wraz z odchyleniem standardowym, w zależności od ilości słów znajdujących się na planszy

Aby móc oceniać i optymalizować wydajność, skonstruowany został test wydajności. Polega on na wielokrotnym uruchamianiu obliczania najlepszych słów, dla różnych zestawów danych (planszy i stojaka), aby możliwie wszechstronnie przetestować elementy algorytmu. Jednak ze względu na wpływ użytych liter na czas obliczania, aby wyniki były wiarygodne, zawartości stojaka nie może być losowana. Kolejne kroki muszą być quasi-losowe, jednak zawsze takie same. Metoda spełniająca podane warunki jest następująca: test rozpoczyna się od pustej planszy. Wybierany jest zestaw wszystkich liter alfabetu o dowolnej, stałej kolejności. Zawartość stojaka ustawiana jest na pierwsze 7 z tych liter. W pętli obliczane jest najlepsze słowo, dodawane do planszy, a zawartość stojaka jest wymieniana na następne 7 liter. Dzieje się tak przez wybraną ilość iteracji. Gdy pobieranie następnych liter wyjdzie poza zakres długości alfabetu, następne litery brane są z początku. W ten sposób zawsze wyznaczane są te same słowa w tej samej kolejności. Mierzony jest tylko czas wykonywania obliczania najlepszych słów, następnie jest dzielony przez ilość obliczonych słów. W trakcie Maszyną testową był laptop zainstalowanym systemem Linux Fedora 37. Posiada 8-rdzeniowy procesor o taktowaniu 1.6 GHz oraz 16 GB pamięci RAM. Podczas testów na urządzeniu nie były aktywnie wykonywane żadne inne czynności. Procesy systemowe zajmujące stale około 2% mocy obliczeniowej procesora. Testy przeprowadzono, obliczając 21 kolejnych słów, powtarzając proces 26 razy. Dla ostatecznej wersji kodu (ze wszystkimi optymalizacjami), średni czas przetwarzania wszystkich słów wyniósł 33.8 sekund, z odchyleniem standardowym 0.38 sekundy, jedno słowo średnio liczyło się 1.6 sekundy. Wyniki wszystkich pomiarów przedstawiono w tabeli 3.

Tabela 3: Porównanie wydajności różnych wersji algorytmu

	Średnia czas iteracji testu [s]	Odchylenie standardowe [s]	Średni czas jednego słowa [s]
Ostateczny kod	33.775	0.381	1.6
Wersja bez zrównoleglenia	113.060	0.623	5.38
Wersja z JCF	33.879	0.436	1.6

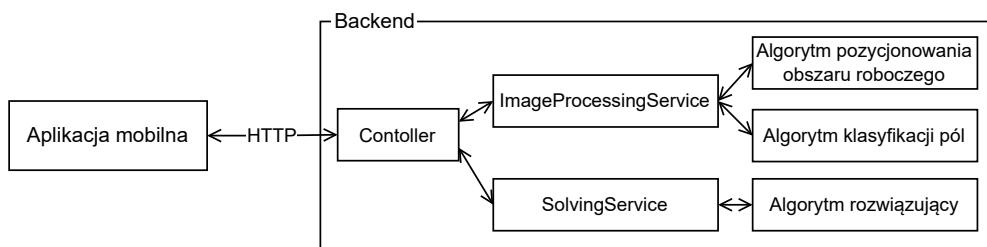
Zrównoleglenie programu polega na wykorzystaniu kilku procesorów lub rdzeni procesora do jednoczesnego wykonania poszczególnych części kodu. Dzięki temu można zmniejszyć czas jego wykonania. Aby zrównoleglenie było możliwe, program musi być odpowiednio zaprojektowany tak, aby możliwe było podzielenie jego wykonania na niezależne części. Domyslnie program napisany w Javie wykonuje się w jednym wątku. Począwszy od Java 8, można w łatwy sposób uzyskać bezpieczne zrównoleglenie, używając Java Stream API. Należy w tym celu zamienić pętlę iterującą po elementach kolekcji na obiekt Stream oraz użyć metody *parallel*. Operacje wykonywane na kolejnych elementach kolekcji, są wtedy uruchamiane w kolejnych wątkach. Jednak wywoływanie interfejsu Stream API wiąże się ze znacznym obciążeniem czasu wykonywania [4]. Z tego względu Stream API zostało użyte w miejscach kodu, które są wywoływane rzadko, jednocześnie mając długim czas przetwarzania. Jest to przetwarzanie każdej kolumny i rzędu, obliczanie potencjalnych słów, oraz filtrowanie słów możliwych do ułożenia. Dla wersji kodu bez zrównoleglenia oraz Stream API, ale ze wszystkimi innymi optymalizacjami, średni czas przetwarzania wszystkich słów wyniósł 113.1 sekundy, z odchyleniem standardowym 0.62 sekundy, jedno słowo średnio liczyło się 5.38 sekundy. Przetwarzanie trwało więc ponad trzykrotnie dłużej. Pokazuje to, że zrównoleglenie potrafi bardzo przyśpieszyć działanie napisanego kodu.

Kolekcje w Javie domyślnie implementowane są przez JCF (Java Collection Framework). Istnieją też inne wersje kolekcji, odznaczające się w niektórych przypadkach większą wydajnością. Przeanalizowano wyniki analizy porównawczej JCF oraz popularnych bibliotek [11] pod kątem możliwych optymalizacji. W wyniku tego domyślna implementacja listy *ArrayList* została zastąpiona przez wersję z biblioteki Eclipse Collections. Przez *FastList* oraz prymitywną kolekcję *IntArrayList* ze względu na zwiększenie wydajności niektórych operacji. Używana w kodzie *HashMapa* nie została zastąpiona, ponieważ w innych implementacjach zaobserwowano spadki wydajności. Przeprowadzono pomiary dla wersji kodu z użytym JCF. Wyznaczono 21 kolejnych słów, test powtórzono 26 razy. Średni czas przetwarzania wszystkich słów wyniósł 33.9 sekundy, z odchyleniem standardowym 0.44 sekundy.

Jest to nieznaczna różnica, mniejsza niż odchylenie standardowe obydwu pomiarów, przez co pomijalna. Wnosząc zmianę domyślnych implementacji bibliotek w przypadku tego algorytmu nie poprawiło wydajności.

3.4 Integracja części serwerowej

Aby użytkownik aplikacji mobilnej mógł korzystać z algorytmów, konieczne jest udostępnienie API, które przyjmie dane, przetworzy je i zwróci wynik. W tym celu zbudowano część serwerową aplikacji, integrującą wszystkie komponenty. Backend został zbudowany zgodnie ze wzorcem Controller-Service-Repository. Wprowadza on podział struktury aplikacji na trzy elementy: kontrolery, serwis, repozytoria. Kontrolery odpowiadają za udostępnienie funkcjonalności zewnętrznym podmiotom, tworząc interfejsy REST API do logiki biznesowej. Serwisy są implementacją logiki biznesowej, w tym przypadku uruchamiają kod odpowiedzialny za prowadzenie obliczeń. Natomiast repozytoria pozwalają na dostęp do szeroko pojętych baz danych - ten element nie występuje w aplikacji, ponieważ nie przechowuje ona żadnych danych, jedynie przeprowadza obliczenia i zwraca wynik. Taka struktura aplikacji pozwala na podział funkcjonalności, łatwiejsze testowanie kodu i wspiera stosowanie zasad SOLID. Diagram architektury aplikacji przedstawiono na rysunku 8.



Rysunek 8: Diagram architektury backendu

3.4.1 Kontroler

Zaimplementowany został kontroler, wystawiający cztery punkty końcowe (ang. endpoint) na porcie 8080. Zapytania przeprowadzane są za pomocą protokołu HTTP, a przesyłane dane używają do zapisu notacji JSON. Zdjęcia przesyłane są w formacie base64 - reprezentacji danych binarnych za pomocą znaków ASCII, dzięki czemu jako tekst mogą zostać dodane do obiektu JSON. Endpointy przyjmujące zdjęcia jako parametr, są zabezpieczone przed przesaniem zbyt dużego pliku. Maksymalny rozmiar zdjęcia wynosi 10 MB. Aplikacja wystawia następujące endpointy:

- *find-corners* - metoda POST, jako treść zapytania przyjmuje zdjęcie. Wykonywany jest algorytm pozycjonowania obszaru roboczego planszy, zwracane są współrzędne wykrytych rogów. Jeżeli nie znaleziono planszy, zwracany jest kod błędu 500.
- *crop-and-recognize* - metoda POST, jako treść zapytania przyjmuje obiekt składający się ze zdjęcia, oraz listę współrzędnych. Jako parametr zapytania można przesyłać język, domyślnie angielski. Wykonywane jest przycinanie i klasyfikacja obszaru roboczego. Zwraca dwuwymiarową tablicę reprezentującą zawartość planszy scrabble.
- *solve* - metoda POST, jako treść zapytania przyjmuje obiekt zawierający dwuwymiarową tablicę z zawartością planszy, oraz jednowymiarową tablicę liter na stojaku. Dodatkowo przyjmuje parametry: język, sposób sortowania wyniku, ilość elementów do zwrócenia. Domyślne wartości to odpowiednio: angielski, po ilości punktów, 10. Wykonywany jest algorytm rozwiązywania scrabble, zwracana jest lista obiektów możliwych rozwiązań. Obiekt zawiera słowo, kierunek układania, współrzędne startu, wartość punktową, oraz zużyte litery. Może zwrócić pustą listę.
- *info* - metoda GET. Zwraca obiekt z konfiguracjami. Obiekt zawiera wymiary planszy, maksymalny rozmiar stojaka, symbol reprezentujący puste pole, listę dostępnych sposobów sortowania

wyniku, listę dostępnych języków, dwuwymiarową tablicę bonusów znajdujących się na planszy, oraz listę obiektów wartości punktowych słów dla każdego języka. Endpoint pełni dla aplikacji dwie funkcje: sprawdza, czy backend jest uruchomiony pod danym adresem oraz przesyła stałe wartości, aby nie musiały być definiowane w dwóch miejscach.

3.4.2 *ImageProcessingService*

Serwis ten jest odpowiedzialny za przetwarzanie obrazu. Uruchamia skrypty napisane w języku Python, za pomocą klasy *PythonRunner*. Pozwala ona na utworzenie nowego procesu systemowego poprzez wywołanie komendy zawierającej: wybrany interpreter Pythona, ścieżkę do pliku ze skryptem oraz dodatkowe parametry (na przykład obiekty w formacie JSON). Komenda uruchamiająca interpreter oraz ścieżka do katalogu ze skryptami są pobierane z pliku konfiguracyjnego *application.properties*. Klasa *PythonRunner* odbiera następnie od skryptu strumień wyjściowy i strumień błędów. Za pomocą wyróżnienia regularnego wykrywa i wyodrębnia ze strumienia wyjścia obiekt JSON będący wynikiem działania algorytmu, następnie zwraca go do serwisu. Jeżeli w skrypcie wystąpił błąd, w strumieniu błędów pojawi się słowo kluczowe "Traceback", a klasa rzuci wyjątek.

Zdjęcie otrzymane od kontrolera jest w tekstowym formacie base64. Nie można go jednak przesłać jako dodatkowy parametr w tej formie poprzez linię komend, ponieważ ma ona limit ilości znaków. Z tego względu zdjęcie jest konwertowane do formy binarnej i tworzony jest obiekt klasy *ImageTemp*. Zapisuje on dane w pliku na dysku i zwraca ścieżkę do pliku. Unikalność nazw plików jest zapewniana dzięki hashowi zdjęcia. Ścieżka jest przekazywana jako dodatkowy parametr do skryptu, który odczyta i przetworzy zdjęcie. Po zakończeniu przetwarzania, na obiekcie wywoływana jest metoda usuwania, w ten sposób pliki tymczasowe są czyszczone po każdym użyciu.

3.4.3 *SolvingService*

Odpowiada za uruchamianie algorytmu przetwarzania scrabbli. W pierwszej kolejności konwertuje otrzymane dane do postaci używanej przez algorytm oraz weryfikuje ich poprawność. Następnie bezpośrednio wywołuje główną klasę algorytmu.

Na poziomie serwisu zdefiniowane jest klasa *ScrabbleResources* która daje dostęp do stałych wartości: wielkości planszy, pojemności uchwytu, listę bonusów na planszy, listę wspieranych języków oraz słowników i alfabetów. Dla każdego z języków przechowywane są załadowane obiekty klas *Alphabet* i *Dictionary*, w mapach gdzie kluczem jest dwuliterowy kod języka a wartością obiekt klasy. Obiekt klasy *Alphabet* zawiera listę liter dostępnych w danym języku, ich wartości punktowe oraz statyczny pusty symbol. Klasa *Dictionary* została dokładnie opisana w rozdziale 3.3.2. Dane ładowane do klas pobierane są z pliku tekstuowego.

Docelowo aplikacja może wspierać dowolną ilość języków, stworzony więc został mechanizm leniwego ładowania alfabetu i słownika. Obiekty alfabetów przechowywane są w mapie *loadedAlphabets* i udostępniane za pomocą metody *getAlphabet*, przyjmującej jako argument kod języka. Jeżeli alfabet dla danego języka był już pobierany, w mapie istnieje wpis, którego wartość jest zwracana. Jeżeli nie, wczytywany jest plik tekstowy, dane są przetwarzane i powstaje nowy obiekt, który dodawany jest do mapy i zwracany. Dla słownika działanie jest analogiczne. Mechanizm ten przypomina tworzenie obiektu we wzorcu projektowym singleton.

Kluczową rolę w ładowaniu zasobów odgrywa uruchamiany przy starcie aplikacji konstruktor serwisu. Pobiera ścieżkę do plików tekstowych z pliku konfiguracyjnego *application.properties* i przekazuje do klasy *ScrabbleResources* - jest to jedyny sposób na zdefiniowanie ścieżki w pliku, a nie statycznie w kodzie (cel tej operacji jest wytłumaczony w rozdziale 3.4.6).

3.4.4 Testy

Testy jednostkowe i integracyjne są bardzo istotnym elementem każdej aplikacji. Pozwalają na proste wykrycie błędów w funkcjonalnościach. Przetestowanych zostało wiele pojedynczych funkcji algorytmu, aby upewnić się, że zwracają prawidłowe rezultaty. Algorytm został też przetestowany integracyjnie pod kątem zwracania oczekiwanych rezultatów dla wyizolowanej części planszy. Jednostkowo zostały przetestowane klasy *ImageTemp* i *PythonRunner* oraz integracyjnie klasa *ScrabbleResources*.

3.4.5 Tworzenie logów

Logi w aplikacji polega na rejestrowaniu zdarzeń, które wydarzyły się w systemie. Pomagają namięścić błędy oraz zbierać statystyki. W aplikacji logowanie zostało zaimplementowane przy użyciu podejścia aspektowego. Pozwala na wywołanie funkcjonalności przed i/lub po działaniu zdefiniowanych funkcji, bez implementowania funkcjonalności dla każdej z nich osobno. Pozwala to zaoszczędzić powtarzającego się kodu. Do tego celu wykorzystano bibliotekę AspectJ [5]. Zaimplementowano system logujący informację o wywoływanych endpointach: który z nich został wywołany, ile trwało wykonywanie, jakie były parametry wejściowe oraz wyjściowe.

3.4.6 Konteneryzacja

Konteneryzacja aplikacji serwerowej to proces umieszczenia jej w wirtualnym, odizolowanym środowisku. Należy w tym celu zdefiniować niezbędne konfiguracje oraz zależności. Pozwala to na łatwe uruchamianie aplikacji w dowolnym systemie operacyjnym, bez konieczności dbania o zgodność, dodatkowe pakiety czy chociażby zajętość używanych portów.

Do konteneryzacji aplikacji wykorzystano narzędzie Docker. Istnieją tu dwa ważne pojęcia: obraz oraz kontener. Relację między nimi można porównać do związku między klasą a obiektem. Obraz jest przepisem na kontener, a kontener jest uruchomioną instancją obrazu. Pierwszym etapem jest więc zbudowanie obrazu, na którego podstawie następnie zostanie uruchomiony kontener.

Sposób budowania obrazu zawarty jest w pliku **Dockerfile**. Składa się z etapów, a każdy z nich jest zbiorem instrukcji, które będą się wykonywać (na przykład kopiowanie plików, uruchamianie komend, ustawianie zmiennych środowiskowych). Opisywany scenariusz zawiera dwa etapy, *build* oraz *production*. Do wykonania etapu **build** tworzony jest nowy kontener, w którego wnętrzu będzie komplikowany kod. Kontener powstaje na podstawie gotowego obrazu *maven:3.8.3-openjdk-17*, który zapewnia środowisko do komplikacji za pomocą narzędzia Maven. W pierwszej kolejności do kontenera budującego kopowane są pliki źródłowe kodu i pliki konfiguracyjne:

- *src* - katalog główny z plikami źródłowymi
- *pom.xml* - plik konfigurujący zawierający informacje o projekcie Java, zawiera informacje, które będą użyte przez Mavena do zbudowania projektu.
- *scrabble_resources* - katalog ze słownikami i alfabetami w postaci plików tekstowych
- *python_scripts* - katalog ze skryptami do przetwarzania obrazu w języku Python
- *application.properties* - zawiera konfigurację aplikacji Spring Boot. Domyślny plik *application.properties* zawiera ustawienia, które odpowiadają środowisku lokalnemu. W kontenerze ścieżki do zasobów, polecenia linii komend oraz inne konfiguracje będą zupełnie inne. W tym celu stworzono dodatkowy plik *application-docker.properties* definiujący ustawienia dla kontenera, następnie nadpisuje domyślny *application.properties*.

Następnie uruchamiane jest narzędzie Maven do budowania projektów na platformę Java za pomocą polecenia:

```
mvn -f /home/app/pom.xml package
```

Flagi *-f* wskazują plik konfigurujący projektu, natomiast *package* oznacza komplikowanie kodu i pakowanie go do formatu dystrybucji. W efekcie powstaje plik *app.jar*.

Etap **production** polega na zestawieniu środowiska produkcyjnego w docelowym kontenerze. Jest tworzony na podstawie obrazu *ubuntu:latest*. Z poprzedniego kontenera kopowane są foldery *scrabble_resources*, *python_scripts* i plik *app.jar*. Następnie instalowane są pakiety: *python3*, *pip* i *openjdk-17-jre*; oraz biblioteki do Pythona: *numpy*, *opencv-python*, *easyocr*. Eksponowany (wystawiany na zewnątrz) jest port 8080, na którym nasłuchuje aplikacja. Ustawiany jest punkt wejściowy, czyli komenda uruchamiana przy starcie kontenera, tak aby uruchomił się plik *app.jar*.

W ten sposób powstał obraz, który następnie należy uruchomić. Do tego celu potrzebne są konfiguracje łączące wnętrze kontenera, ze środowiskiem hosta. Przykładem może być mapowanie portów, dzięki temu aplikacja w kontenerze działająca na porcie 8080, na zewnątrz może być widoczna na

porcie 5000. Uruchomienie kontenera może zostać wykonane ręcznie, poprzez napisanie komendy z konfiguracjami. Dobrą praktyką jest jednak stworzenie pliku *doker-compose.yml*, który może zawiera parametry uruchamiające wiele kontenerów za jednym razem. W tym przypadku plik jest bardzo prosty i zawiera nazwę uruchamianego kontenera oraz mapowanie portów. Po uruchomieniu komendy

```
docker compose up
```

aplikacja zostanie automatyczne zbudowana i uruchomiona.

3.5 Aplikacja mobilna

Aplikacja mobilna została zaimplementowana w bibliotece React Native, pozwalającej na budowanie wieloplatformowych projektów używając języka JavaScript. Aplikacja została osadzona na platformie Expo, która automatyzuje i ułatwia tworzenie, uruchamianie i rozwijanie aplikacji oraz umożliwia programiście zupełnie nie używać natywnego kodu. Całość bazuje na bibliotece React, służącej do tworzenia interaktywnych interfejsów użytkownika przy użyciu komponentów. Komponent to izolowana część kodu w postaci funkcji, zawierająca logikę działania i zwracającą wygląd w formacie jsx. Komponenty mogą być w sobie dowolnie zagnieżdżone. Działanie komponentów bazuje na dwóch mechanizmach: state oraz props. Mechanizm state oznacza stan komponentu, który jest specjalną zmienną. Po zmianie jej wartości wszystkie elementy korzystające z niej zostaną wyrenderowane od nowa. Komponent może zawierać wiele stanów, jest to mechanizm pozwalający na dynamiczną zmianę wyświetlanej zawartości. Mechanizm *props* pozwala na przekazanie informacji od komponentu nadzawanego do podzwanego. Mogą w ten sposób być przekazane konkretne wartości, ale również metody, które następnie mogą być wywołane w podzwanym komponencie. Drugi z elementów jest nazywany mechanizmem *callback* i pozwala na wykonanie czynności z komponentu podzwanego w nadzwanym.

3.5.1 Struktura aplikacji

Struktura aplikacji bazuje na głównym komponencie, który stanowi korzeń dla wszystkich innych elementów. Jest renderowany przez cały czas, gdy aplikacja jest otwarta, nie ma jednak widocznych elementów. Zwraca on komponent jednej ze stron, czyli osobnych paneli widocznych dla użytkownika, zmieniających się w zależności od wykonywanych czynności. Komponent główny zawiera zmienną stanu *page*, która jako wartość zawiera unikalny kod strony. Gdy zmienna *page* jest ustawiana na inną, React automatycznie renderuje kolejną wybraną stronę.

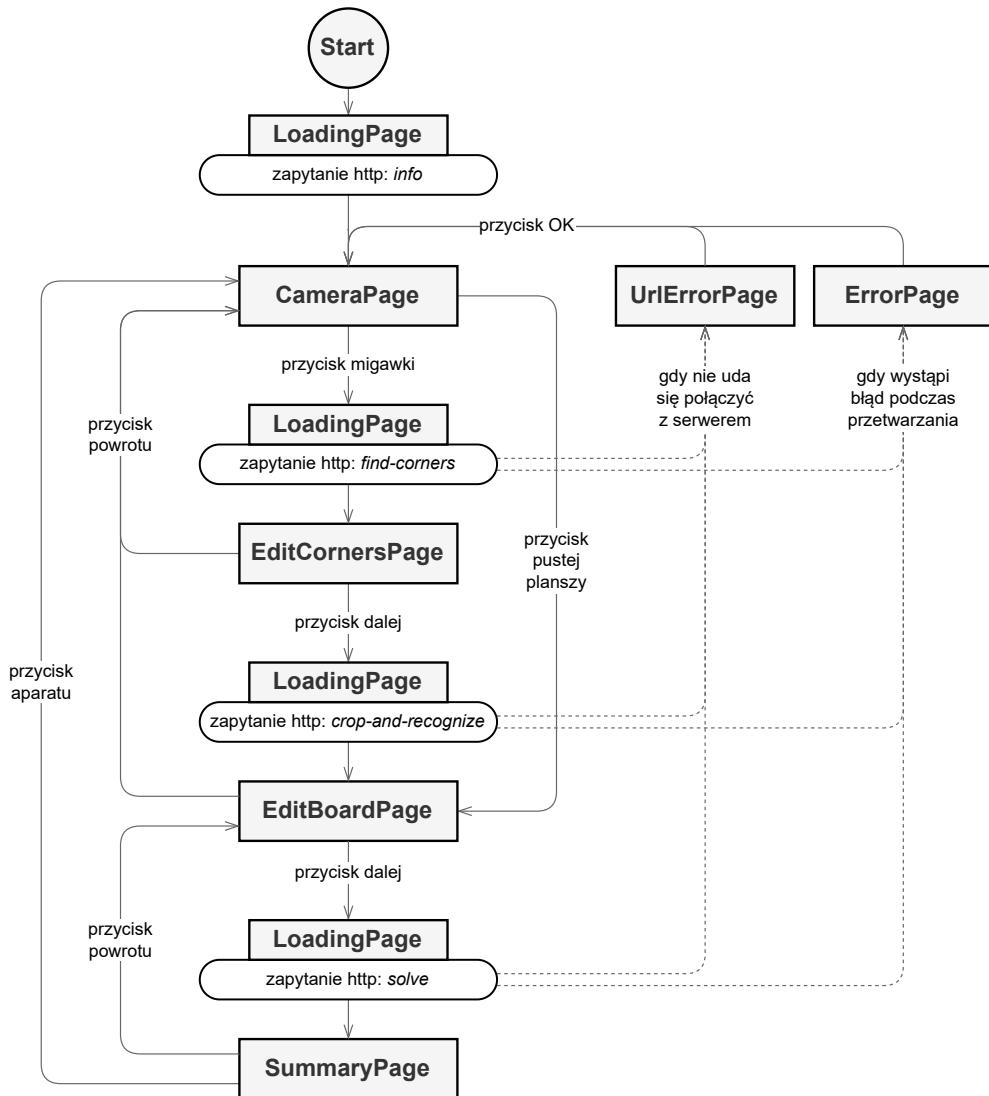
Każdy komponent strony zajmuje pełną powierzchnię ekranu i zawiera inne komponenty jak na przykład napisy, przyciski, zdjęcia. Każda strona zawiera elementy nawigacyjne, pozwalające poruszać się pomiędzy nimi. W tym celu główny komponent za pomocą mechanizmu *props* przekazuje stronie referencję (*callback*) do funkcji, która zmieni stan *page*. Po jej wywołaniu wewnątrz komponentu strony, zostanie wyrenderowana inna strona.

3.5.2 Strony

Strony zawierają przyciski nawigacyjne, zmieniające aktualnie wyświetlany panel. Dodatkowo na każdej ze stron użytkownik może kliknąć systemowy przycisk wstecz, która w domyśle przenosi na "poprzednią" stronę. Interakcje z użytkownikiem (ang. userflow), czyli możliwe przejścia pomiędzy stronami i wykonujące się wtedy zapytania HTTP, zostały zaprezentowane na rysunku 9.

Gdy aplikacja jest uruchamiana, na początku wyświetlana jest strona *LoadingPage*, w tym czasie wykonywane jest zapytanie do endpointa *info*, pobieraj podstawowe informacje, na podstawie domyślnego adresu IP. Jeżeli nie uda się połączyć z backendem, użytkownik zostanie przeniesiony na stronę *UrlErrorPage*, aby zmienił adres backendu. Jeżeli wszystko pojedzie poprawnie, strona zmieni się na *CameraPage*.

1. **CameraPage** to strona startowa aplikacji, głównym elementem jest wyświetlany widok z aparatu, oraz suwak wyboru języka gry. Poniżej znajduje się panel nawigacyjny z trzema przyciskami. Przycisk pomocy, wyświetlający漂wające okno z podstawowymi informacjami o aplikacji. Przycisk migawki powoduje zrobienie zdjęcia, odpytanie endpointu *find-corners* oraz zmianę



Rysunek 9: Userflow aplikacji, oraz zapytania HTTP do backendu

strony na *EditCornersPage*. Przycisk pustej planszy powoduje zmianę strony na *EditBoardPage*, z pustą zawartością planszy.

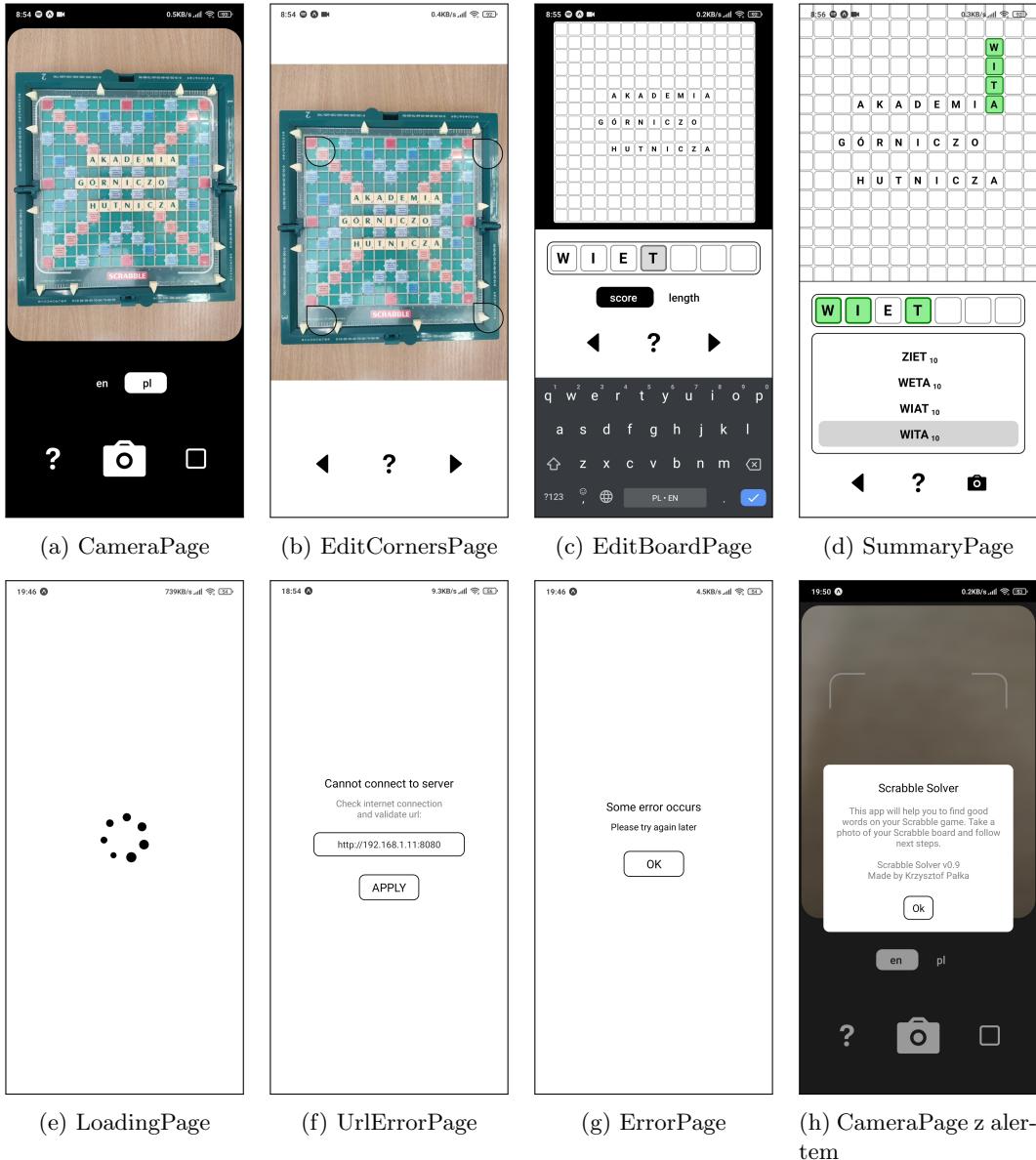
2. ***EditCornersPage*** pozwala edytować współrzędne rogów planszy wykrytych przez algorytm. Głównym elementem jest zrobione wcześniej zdjęcie. Na nim znajdują się cztery wskaźniki, których pozycję można zmieniać za pomocą przeciągania. Ich pozycje ustawiane są na punkty zwrócone przez backend lub wartości domyślne w przypadku błędu. Użytkownik powinien je ustawić na rogi planszy, jeżeli nie zostały prawidłowo wykryte. Panel nawigacyjny pozwala na powrót do *CameraPage*, otrzymanie okna z pomocą, oraz przejście dalej. Ostatnie element powoduje odpytanie endpointu *crop-and-recognize*, a następnie przejście do strony *EditBoardPage*.
3. ***EditBoardPage*** zawiera możliwą do przybliżania i oddalania planszę oraz stojak, składające się z edytowalnych pól. Po kliknięciu dowolnego z pól pojawi się klawiatura, która umożliwia wprowadzenie lub usunięcie litery (tylko prawidłowe wartości danego języka są akceptowane). Poniżej jest suwak umożliwiający wybór sposobu sortowania słów oraz panel nawigacyjny. Przycisk wstecz spowoduje powrót do *CameraPage*, przycisk pomocy wyświetli okno z pomocą, a przycisk dalej odpyta endpoint *solver* oraz zmieni stronę na *SummaryPage*.

4. **SummaryPage** zawiera planszę i stojak jak w poprzednim widoku, tym razem jednak bez możliwości edycji. Poniżej znajduje się lista słów znalezionych przez algorytm, z wartościami punktowymi. Po kliknięciu słowa zostanie wyświetcone na planszy na zielono, a na stojaku zostaną zaznaczone użyte płytki. Panel nawigacyjny umożliwia powrót do *EditBoardPage*, otrzymanie pomocy, oraz bezpośredni powrót do *CameraPage*. Powrót do *EditBoardPage* zachowuje stan planszy, więc można w ten sposób edytować ją bez konieczności powtórnego skanowania.
5. **LoadingPage** jest wyświetlana na początku przy ładowaniu aplikacji oraz w trakcie każdego zapytania do backendu. Na środku znajduje się animowany element ładowania. Jest to jedyna strona, która nie zawiera elementów nawigacyjnych.
6. **UrlErrorPage** jest stroną, na którą użytkownik zostanie przekierowany, jeżeli wystąpi błąd połączenia z backendem. Zawiera informację, pole edycyjne z adresem IP i portem, oraz przycisk zatwierdzania. Użytkownik powinien podać poprawny adres serwera oraz zatwierdzić. Aplikacja spróbuje wtedy się połączyć z backendem. Jeżeli to się uda, użytkownik zostanie przekierowany do *CameraPage*.
7. **ErrorPage** wyświetla się, jeżeli wystąpi nieobsługiwany wyjątek. Zawiera komunikat o błędzie oraz przycisk przenoszący użytkownika do *CameraPage*.

3.5.3 Interfejs użytkownika

Interfejs użytkownika został zaprojektowany w minimalistycznej, czarno-białej formie, z dodatkowym kolorem zielonym jako akcent. Wszystkie elementy bazują na prostokątach o zaokrąglonych brzegach. Zrzut ekranu każdej ze stron zaprezentowano na rysunku 10. Każda strona jest złożona z różnych komponentów React, które odpowiadają za poszczególne części interfejsu użytkownika. Komponent to część, dająca się wydzielić ze względu na wygląd i funkcjonalność. Każdy z nich bazuje na podstawowych elementach interfejsu jak tekst, przyciski, pola dotykowe, wyświetlanie grafiki itp. Najważniejsze i najczęściej używane zbudowane komponenty aplikacji to:

1. **Button** (przycisk) to podstawowa część interfejsu. Została oparta na wbudowanym polu dotykowym oraz odpowiednio ostylowana. Jako parametry przyjmuje wyświetlany tekst oraz metodę (*callback*) wykonującą się podczas kliknięcia.
2. **VectorIcon** (ikony wektorowe) pozwalają na wyświetlenie jednej z dostępnych ikon. W aplikacji służą do stworzenia czytelniejszych przycisków. Zostały użyte gotowe komponenty z pakietu *@expo/vector-icons*. Jako parametr przyjmuje nazwę ikony i wyświetlony rozmiar.
3. **SegmentedControl** umożliwia użytkownikowi wybór jednej opcji. Składa się z kilku elementów, które reprezentują dostępne opcje. Pozwala użytkownikowi na wybranie tylko jednej z nich. Wyświetla animację podczas zmiany wartości. Główne parametry to lista elementów do wyświetlenia, indeks aktualnego elementu, i czynność wykonywana przy zmianie.
4. **Alert** to komponent służący do wyświetlania informacji użytkownikowi, który jest widoczny na wierzchu innych elementów interfejsu. Wyświetla płynące okno z tekstem i przyciskiem zamykania. Został zbudowany na bazie pakietu *react-native-awesome-alerts*. Jako parametr przyjmuje tekst do wyświetlenia.
5. **Camera** umożliwia aplikacji dostęp do aparatu urządzenia mobilnego i jego funkcji, takich jak wyświetlanie podglądu, czy robienie zdjęcia. Jest częścią biblioteki Expo, zawartą w pakiecie *expo-camera*. Najważniejsze parametry to proporcje zdjęcia (należy wybrać jeden ze wspieranych przez telefon, domyślnie wszystkie wspierają proporcje 4:3) oraz typ aparatu (przedni lub tylny). Do korzystania z aparatu, aplikacja potrzebuje uprawnień systemowych, które musi przyznać użytkownik podczas uruchomienia aplikacji po raz pierwszy.



Rysunek 10: Zrzuty ekranu stron w aplikacji mobilnej

6. **Field** to komponent obrazujący pojedyncze pole na planszy. Składa się z tekstu i ramki, może być wypełniony literą lub pusty. Ma możliwość włączenia edycji - po kliknięciu przez użytkownika otwiera się klawiatura, na której należy wpisać nową wartość. Litera jest sprawdzana pod kątem poprawności z aktualnie wybranym językiem. Gdy edytowanie jest wyłączone, pole reaguje na dotyk jedynie animacją. Gdy pole zawiera jakąś wartość, automatycznie wyświetla wartość punktową litery.

Edycja zawartości została zoptymalizowana, aby możliwe było wyświetlenie w jednym czasie całej planszy (225 elementów). Najbardziej zasobnym elementem jest pole wprowadzania tekstu, z tego względu jest wyświetlany zwykły tekst bez możliwości edycji. Aby umożliwić edycję, po kliknięciu pole zostaje utworzony ukryty komponent pola wprowadzania tekstu. Służy tylko do uruchomienia klawiatury systemowej i pobrania z niej wartości. Po przejściu do innego pola jest automatycznie usuwane.

7. **Board** oraz **Rack** to niestandardowe elementy, mające w aplikacji obrazować planszę scrabble

oraz stojak użytkownika. Obydwa są tablicami komponentów *Field*. Umożliwiają ustawienie wartości wszystkich podrzędnych pól za pomocą tablicy liter oraz ustawiają możliwość edycji. Komponent *Board* ma wbudowaną możliwość dowolnego przybliżania, oddalania oraz przesuwania.

3.5.4 Uruchamianie aplikacji

Aplikacja oparta na platformie Expo może zostać uruchomiona na fizycznym telefonie z Androidem w trybie deweloperskim. Dzięki niemu każda zmiana wykonana w kodzie zostaje automatycznie wyświetlona. W tym celu na urządzeniu należy zainstalować aplikację ExpoGo ze Sklepu Play. Konieczna jest również instalacja pakietu expo-cli na komputerze za pomocą komendy:

```
npm install -g expo-cli
```

Następnie za pomocą komendy:

```
expo start
```

uruchamiany jest na komputerze serwer udostępniający deweloperską wersję aplikacji. Aby się do niego podłączyć, urządzenia muszą być w tej samej sieci lokalnej, oraz należy za pomocą aplikacji ExpoGo zeskanować kod QR wyświetlony w terminalu. Aby uruchomić aplikację na telefonie z systemem IOS, niezbędne jest posiadanie laptopa i telefonu z tym systemem. Proces przebiega podobnie.

Aby umożliwić zainstalowanie aplikacji na dowolnym urządzeniu, musi zostać zbudowana do pliku APK. W katalogu projektu należy uruchomić polecenie:

```
expo build:android
```

Program budujący zada kilka pytań, między innymi o nazwę docelowego pakietu, oraz typ pliku. Proces budowania będzie się odbywał na serwerze Expo. Po zakończeniu procesu budowania Expo zapewni dostęp do pobrania pliku APK, który następnie można zainstalować na dowolnym urządzeniu. Dla IOS proces przebiega analogicznie, jednak tak wygenerowanego pliku z rozszerzeniem IPA nie da się uruchomić na dowolnym telefonie, bez akceptacji firmy Apple.

4 Porównanie

Zbudowany system został porównany aplikacją mobilną ScrabbleSolver [14]. Została ona wybrana, ponieważ jest to jedyne znalezione rozwiązanie realizujące te same funkcjonalności co autorska aplikacja - skanowanie fizycznej planszy do gry oraz proponowanie użytkownikowi możliwych rozwiązań. Jest to aplikacja udostępniona użytkownikom za pomocą sklepu Google Play. Ma ona średnią ocenę użytkowników w wysokości 3.4 gwiazdek na 5 możliwych, gdzie ponad 25% opinii jest wystawionych z najniższą możliwą punktacją - sugeruje to, że aplikacja nie jest doskonała. Porównanie zostało dokonane na kilku płaszczyznach.

4.1 Funkcjonalności

Aplikacje są bardzo zbliżone funkcjonalnościami: robienie zdjęcia, rozpoznawanie planszy, wyliczanie najlepszych słów. Konkurencyjne rozwiązanie zawiera jednak kilka dodatkowych elementów:

- pozwala na zrobienie zdjęcia aparatem, ale również wybór zdjęcia z galerii
- umożliwia wybór różnych typów planszy dla różnych gier scrabble-podobnych (na przykład dla gry Words With Friends)
- może zasugerować podpowiedź, a nie tylko gotowe rozwiązania
- pozwala na zapis stanu planszy na przyszłość

Nie zawiera jednak wbudowanego systemu pomocy użytkownikowi, jedynym sposobem na zapoznanie się z zasadą działania jest odnalezienie przycisku prowadzącego do poradnika w serwisie YouTube.

4.2 Interfejs graficzny

Konkurencyjne rozwiązanie ma przestarzały interfejs użytkownika. Zawiera wiele niepasujących do siebie kolorów (niebieski, czerwony, żółty, jasnoniebieski, biały), co nie odpowiada nowoczesnym zasadom projektowania interfejsu graficznego. Na każdej stronie znajduje się duże, rozpraszające logo. Ułożenie elementów oraz ich wymiary są niespójne i chaotyczne między stronami. Na przykład wysokości przycisków przyjmują wartości 3, 6, 8, 9, 11, 12 lub 22 milimetry (pomiary dokonano ręcznie, linijką na ekranie telefonu), co jest brakiem spójności. Powoduje to zamieszanie i dezorientację użytkownika. Natomiast interfejs użytkownika w autorskim rozwiązaniu jest prosty i przejrzysty, bazujący na kolorze białym i czarnym.

4.3 Interakcje z użytkownikiem

W obydwu aplikacjach interakcja z użytkownikiem ma takie same filary - należy zrobić zdjęcie, poprawić wykryte rogi plansz oraz poprawić rozpoznane litery. Jednak w inny sposób zostały zrealizowane bazowe funkcjonalności. **Edycja wierzchołków** w ScrabbleSolver odbywa się poprzez trzy nieintuicyjne kliknięcia - pierwsze wybiera najbliższy punkt do edycji, drugie przybliża planszę w miejscu kliknięcia, trzecie zapisuje punkt w miejscu kliknięcia i oddala planszę. Ta metoda nie pozwala na precyzyjne zaznaczenie wybranego punktu. Zbudowana aplikacja pozwala na prostą edycję za pomocą przeciągania i upuszczania elementu wskazującego, co jest dużo wygodniejszym sposobem.

Aby **edytować planszę** w ScrabbleSolver należy kliknąć punkt na planszy, który zostanie przybliżony, następnie wybrać jedno z pól. Otworzy się okienko dialogowe, które pozwala na wpisanie ciągu znaków pionowo lub poziomo, dowolnej długości, które zostaną umieszczone na planszy. W konsekwencji edycja pojedynczej litery wymaga aż czterech kliknięć, a dodanie słowa długości n wymaga $2+n$ kliknięć. Ten sposób wymaga bardzo dużej ilości kliknięć podczas edytowania pojedynczych źle rozpoznanych liter w różnych miejscach planszy, ale pozwala na szybkie dodanie bardzo długiego słowa. W zbudowanym rozwiązaniu edycja pojedynczej litery wymaga jedynie dwóch kliknięć, ale edycja n liter wymaga $2*n$ kliknięć, ponieważ za każdy razem trzeba zmieniać aktywne pole. Powoduje to, że edycja błędów jest szybka, ale dodanie długiego słowa wymaga wielu kliknięć. Do aplikacji bardzo prosto można zaaplikować zalety obydwu podejść, poprzez automatyczne zmianianie aktywnego pola po kliknięciu klawiaturze. Zmiana może dokonywać się pionowo lub poziomo, do wyboru przez użytkownika.

4.4 Algorytmy

Porównywane aplikacje działają w bardzo podobny sposób i zawierają te same kroki rozpoznawania planszy - w konsekwencji muszą używać zarówno algorytmu pozycjonującego i kwalifikującego. Wszystkie obliczenia w aplikacji ScrabbleSolver są uruchamiane lokalnie na urządzeniu, ponieważ aplikacja może działać bez dostępu do internetu. **Skanowanie planszy** zostało manualnie przetestowane na trudnej do rozpoznawania planszy, z ułożonymi 23 różnymi literami z języka angielskiego. Każdą z aplikacji wykonano dziesięciokrotnie skanowanie planszy, robiąc zdjęcie od góry i przy dobrym oświetleniu. Zapisywano ilość poprawnie wykrytych wierzchołków, poprawnie rozpoznanych liter, oraz liter rozpoznanych w miejscach, w których nie powinny wystąpić (nieoczekiwanych liter). Otrzymane wyniki przedstawiono w tabeli 4.

Na 40 oczekiwanych poprawnie wykrytych wierzchołków, ScrabbleSolver wykrył jeden, a zbudowana aplikacja trzy. Są to jednak wartości pomijalne, sugerujące, że obydwie aplikacje nie potrafią pozycjonować planszy, na której były testowane. Aplikacja ScrabbleSolver ma około 63% poprawnie rozpoznanych liter. Ma jednak bardzo duże odchylenie standardowe, ponieważ część plansz została rozpoznana poprawnie praktycznie w całości, a część zawierała prawie wyłącznie błędy. Zbudowane rozwiązanie prawidłowo rozpoznało około 57% liter. Ma jednak niskie odchylenie standardowe, co oznacza, że ilość poprawnych rozpoznań była względnie stała. Jest to wartość zbliżona do 52% poprawnie rozpoznanych, otrzymanych podczas porównania silników OCR. W tamtym przypadku test był wykonywany w języku Polski, stąd może wynikać występująca różnica 5 punktów procentowych. Natomiast nieoczekiwane rozpoznane litery, występują dwukrotnie częściej w zbudowanym rozwiązaniu - na poziomie około czterech na planszę. Odchylenie standardowe w obydwu przypadkach jest prawie równe ilości nieoczekiwane rozpoznanych.

Niestabilność ilości poprawnie rozpoznanych pól w konkurencyjnej aplikacji może mieć związek z trudnym do przeprowadzenia w sposób precyzyjny ustaleniem pozycji wierzchołków. Był to jedyny zmienny czynnik przy testach, spowodowany ludzką niedokładnością. Jeżeli teza ta jest prawdziwa, oznacza to, że najważniejszym elementem poza algorytmami jest dostarczenie użytkownikowi dobrego narzędzia, aby mógł możliwie precyzyjnie dokonać pozycjonowania. Wnioskując, konkurencyjna aplikacja posiada lepszy algorytm kwalifikujący, ponieważ istnieją sytuacje, gdy większość znaków jest rozpoznana poprawnie. Natomiast zbudowany system zawiera lepsze narzędzie do ręcznego pozycjonowania, co zapewnia stabilną ilość poprawnych rozpoznań.

Tabela 4: Porównanie poprawności pozycjonowania i kwalifikacji płytka dla ScrabbleSolver oraz zbudowanej aplikacji, dla każdej z aplikacji test przeprowadzono 10 razy

	ScrabbleSolver	Autorska aplikacja
Wszystkie wierzchołki do wykrycia	40	40
Poprawne wykryte wierzchołki	1	3
Procent poprawnie wykrytych	2.5 %	7.5 %
Ilość liter na planszy	23	23
Średnio poprawnie rozpoznanych liter	14.5	13.2
Procent poprawnie rozpoznanych	63.0 %	57.4 %
Odczytanie standardowe	8.48	1.40
Ilość pól na planszy	225	225
Średnio nieoczekiwane rozpoznane liter	2.2	4.1
Procent niepoprawnie rozpoznanych	0.98 %	1.82 %
Odczytanie standardowe	2.2	4.25

Algorytm rozwiązujący w aplikacji ScrabbleSolver działa bardzo szybko oraz wspiera blanki. Zbudowany system działa znacznie wolniej oraz nie wspiera blanków. Przetestowano obydwie aplikacje na czterech scenariuszach planszy dla języka angielskiego i dla każdego z nich najlepsze zwrócone słowa były takie same. Jednak porównując wszystkie możliwe rozwiązania, aplikacja ze sklepu zawierała więcej słów. W zbudowanym systemie brakowało elementów ze względu na znane braki. Prostym rozwiązaniem długiego czasu działania oraz braków może być porzucenie własnego rozwiązania algorytmu, na rzecz jednego z gotowych implementacji otwarto-źródłowych, wymienionych w przeglądzie powiązanych prac w rozdziale 2.

Podsumowując, aplikacja ScrabbleSolver jest dojrzałym rozwiązaniem, udostępnionym użytkownikom za pomocą platformy Google Play. Ma ono więcej funkcjonalności oraz lepiej działające algorytmy, jednak odstaje przejrzystością i prostotą interfejsu użytkownika. Część wad zbudowanego rozwiązania może zostać prosto poprawiona. Spowoduje to, że zbudowane rozwiązanie może być konkurencją dla istniejącej aplikacji.

5 Podsumowanie

Zaprojektowana i zbudowana aplikacja spełnia postawione założenia. Program pozwala na wybór jednego z dwóch dostępnych języków, aby system obsługiwał więcej języków, wystarczy dodać plik ze słownikiem oraz wartościami punktowymi liter do katalogu. Umożliwia skanowanie planszy przy pomocy aparatu w telefonie i na szybkie poprawienie wyników działania algorytmów. Następnie oblicza listę słów, które mogą zostać ułożone i prezentuje je użytkownikowi w przejrzystej postaci interaktywnej listy. Zbudowany system jest kompletnym rozwiązaniem możliwym do użycia podczas prawdziwej rozgrywki. Jest oparta na algorytmach, które w ramach pracy zostały zaprojektowane i wykonane od zera, bez używania zewnętrznego kodu. Każdy z nich sam w sobie jest niełatwym do rozwiązania problemem inżynierskim i może zostać ulepszony (przyspieszony, poprawiona dokładność i niezawodność).

W pracy wykazano trudność zbudowania uniwersalnego **algorytmu do pozycjonowania** obszaru roboczego planszy scrabbli. Przetestowano dwa sposoby rozwiązania problemu, z których żaden nie

dał oczekiwanych rezultatów dla dowolnej z testowanych plansz. W konsekwencji dodano możliwość ręcznej edycji, przerzucając część odpowiedzialności na użytkownika. Trudności tego problemu dowodzi fakt, że jedyna istniejąca na rynku aplikacja o podobnym zastosowaniu również daje użytkownikowi taką możliwość. Ta tematyka wymaga jednak dalszej pracy w celu znalezienia niezawodnego rozwiązania. Potencjalnym kierunkiem może być rezygnacja z wykrywania wszystkich plansz, na rzecz automatycznego wykrywania tylko niektórych, ale za to z większą skutecznością. Przykładem tego typu metody może być odnajdywanie skrajnych pól z potrójną premią słowną za pomocą filtrowania odpowiedniego koloru.

Następnie przetestowano dwie biblioteki OCR (PyTesseract oraz EasyOCR) do **kwalifikowania pól** na planszy scrabble jako jedna z liter. Udowodniono, że silniki te mają niską skuteczność podczas rozpoznawania pojedynczych znaków, szczególnie gdy w pobliżu znajdują się problematyczne artefakty. Jednym z kierunków dalszych badań w tym temacie może być splotowa sieć neuronowa, która nauczy się ignorować nieistotne elementy. Dzięki tej metodzie litery mogłyby być w dowolnej orientacji przestrzennej. Jednak istotną wadą jest konieczność zdobycia dużego zbioru danych uczących dla każdego z obsługiwanych języków. W gotowych rozwiązańach OCR użycie praktycznie dowolnego języka jest możliwe out-of-the-box.

Ostatnim dużym komponentem aplikacji jest **algorytm rozwiązyujący scrabble**. Został zaprojektowany i zaimplementowany bez znajomości innych rozwiązań. Wyznacza wysoko punktowane słowa w średnim czasie około 2-4 sekund. Nie znajduje jednak wszystkich możliwych słów, nie wspiera blanków, a jego czas działania nie jest na najwyższym poziomie. Poprawienie tych aspektów może być trudne do realizacji, dlatego lepszym rozwiązaniem na poprawę działania aplikacji jest zastąpienie własnej implementacji, za pomocą otwarto źródłowej biblioteki (na przykład Unofficial Scrabble C++ Library [13]). W ten sposób wszystkie wymienione wady zostaną rozwiązane niskim nakładem pracy.

Stworzona **aplikacja mobilna** jest przejrzysta, przyjazna dla użytkownika i prosta w obsłudze. Pozwala na szybką nawigację i oferuje użytkownikowi pomoc na każdym z etapów. Jest zintegrowana z **backendem**, który umożliwia prostą edycję używanych algorytmów i bibliotek, oraz może zostać uruchomiony na serwerze produkcyjnym za pomocą jednej komendy. Mimo niedoskonałości algorytmów projektowany system stanowi bazę do dalszego rozwoju. Aplikacja może być konkurencyjna dla jedynej aplikacji na rynku oferującej podpowiadanie słów dla fizycznej wersji gry scrabble.

Bibliografia

- [1] Andrew W. Appel and Guy J. Jacobson. “The World’s Fastest Scrabble Program”. In: *Commun. ACM* 31.5 (May 1988), pp. 572–578. ISSN: 0001-0782. DOI: 10.1145/42411.42420. URL: <https://doi.org/10.1145/42411.42420>.
- [2] Gabriel Araújo. *Mobile Scrabble Word Finder*. URL: <https://github.com/gabriaraujo/scrabble>. (czas dostępu: 30.10.2022).
- [3] Xavier Averbouch. *Sharp Scrabble Solver*. URL: <https://github.com/xavave/Sharp-Scrabble-Solver>. (czas dostępu: 30.10.2022).
- [4] Matteo Basso et al. “Optimizing Parallel Java Streams”. In: *2022 26th International Conference on Engineering of Complex Computer Systems (ICECCS)*. 2022, pp. 23–32. DOI: 10.1109/ICECCS54210.2022.00012.
- [5] *Biblioteka AspectJ dla języka Java*. URL: <https://github.com/eclipse/org.aspectj>. (czas dostępu: 01.12.2022).
- [6] *Biblioteka EasyOCR dla języka Python*. URL: <https://pypi.org/project/easyocr/>. (czas dostępu: 26.11.2022).
- [7] *Biblioteka OpenCV dla języka Python*. URL: <https://pypi.org/project/opencv-python/>. (czas dostępu: 26.11.2022).
- [8] *Biblioteka Pytesseract dla języka Python*. URL: <https://pypi.org/project/pytesseract/>. (czas dostępu: 26.11.2022).
- [9] *Blender*. URL: <https://www.blender.org/>. (czas dostępu: 14.11.2022).
- [10] Andrés Bustamante et al. “Video Processing from a Virtual Unmanned Aerial Vehicle: Comparing Two Approaches to Using OpenCV in Unity”. In: *Applied Sciences* 12.12 (2022). ISSN: 2076-3417. DOI: 10.3390/app12125958. URL: <https://www.mdpi.com/2076-3417/12/12/5958>.
- [11] Diego Costa et al. “Empirical Study of Usage and Performance of Java Collections”. In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering. ICPE ’17*. L’Aquila, Italy: Association for Computing Machinery, 2017, pp. 389–400. ISBN: 9781450344043. DOI: 10.1145/3030207.3030221. URL: <https://doi.org/10.1145/3030207.3030221>.
- [12] *Difference Between Web vs Hybrid vs Native Apps*. URL: <https://www.lambdatest.com/blog/web-vs-hybrid-vs-native-apps/>. (czas dostępu: 08.11.2022).
- [13] Adam Escobedo. *Unofficial Scrabble C++ Library (scl)*. URL: <https://github.com/adamcesco/Unofficial-Scrabble-Library>. (czas dostępu: 08.11.2022).
- [14] Jordi Hautot. *Scrabboard Solver - Scrabble H*. URL: <https://play.google.com/store/apps/details?id=com.scrabble.jhautot>. (czas dostępu: 30.10.2022).
- [15] David Hirschberg. *Scrabble Board Automatic Detector for Third Party Applications*. URL: http://rasdasd.com/projects/Scrabble_Detector/Scrabble_Paper.pdf. (czas dostępu: 03.11.2022).
- [16] David Koeplinger. *EE368 Project Proposal - Scrabble Assistant*. URL: <https://5y1.org/download/ef9b15f978fbf691fb445c4afdf591254.pdf>. (czas dostępu: 03.11.2022).
- [17] Kamil Mielnik. *Scrabble Solver 2*. URL: <https://github.com/kamilmielnik/scrabble-solver>. (czas dostępu: 30.10.2022).
- [18] *OpenCV - Feature Matching*. URL: https://docs.opencv.org/4.x/dc/dc3/tutorial_py_matcher.html. (czas dostępu: 14.11.2022).
- [19] *OpenCV documentation - Morphological Transformations*. URL: https://docs.opencv.org/4.x/d9/d61/tutorial_py_morphological_ops.html. (czas dostępu: 14.11.2022).
- [20] Krzysztof Pałka. *Repozytorium zawierające kod projektu*. URL: <https://github.com/kristopalka/scrabble-solver>. (czas dostępu: 18.12.2022).

- [21] Michael Pang. *Words-with-Bots*. URL: <https://github.com/Akababa/Words-with-Bots>. (czas dostępu: 08.11.2022).
- [22] Saurabh Singh Parihar. *Enigma*. URL: <https://github.com/ssp5zone/enigma>. (czas dostępu: 30.10.2022).
- [23] *Reguły Gry Polskiej Federacji Scrabble*. URL: <http://www.pfs.org.pl/regulaminy.php>. (czas dostępu: 26.10.2022).
- [24] *Scrabble Assistant With Computer Vision*. URL: <https://www.youtube.com/watch?v=I3ALe-qEwzM>. (czas dostępu: 03.11.2022).
- [25] *Scrabble letter distributions*. URL: https://en.wikipedia.org/wiki/Scrabble_letter_distributions. (czas dostępu: 26.10.2022).
- [26] *Scrabble tools - dictionary*. URL: <https://scrabble.hasbro.com/en-us/tools>. (czas dostępu: 30.10.2022).
- [27] *Scrabble Word Finder*. URL: <https://scrabblewordfinder.org/solver>. (czas dostępu: 30.10.2022).
- [28] *Scrabble Word Finder - Online Scrabble Solver*. URL: <https://playscrabble.com/word-finder>. (czas dostępu: 30.10.2022).
- [29] *ScrabbleMania - Słownik Scrabble, Literaki*. URL: <https://scrabblemania.pl/>. (czas dostępu: 30.10.2022).
- [30] Brian Sheppard. “World-championship-caliber SCRABBLE”. In: *Artificial Intelligence* 134.1 (2002), pp. 241–275. ISSN: 0004-3702. DOI: [https://doi.org/10.1016/S0004-3702\(01\)00166-7](https://doi.org/10.1016/S0004-3702(01)00166-7). URL: <https://www.sciencedirect.com/science/article/pii/S0004370201001667>.
- [31] *Stack Overflow Survey - most popular technologies*. URL: <https://insights.stackoverflow.com/survey/2021#most-popular-technologies-misc-tech>. (czas dostępu: 08.11.2022).
- [32] Simon Stender and Hampus Åkesson. *Cross-platform Framework Comparison : Flutter & React Native*. 2020. URL: <http://urn.kb.se/resolve?urn=urn:nbn:se:bth-19749>.
- [33] *Word Finder & Scrabble Cheat*. URL: <https://wordfinder.yourdictionary.com/>. (czas dostępu: 30.10.2022).
- [34] *WordFinder by YourDictionary*. URL: <https://play.google.com/store/apps/details?id=com.lovetoknow.wordfinder>. (czas dostępu: 30.10.2022).
- [35] Zach Witzel Xiluo He. *Computer Move Suggestions from Physical Game Images*. URL: <http://cs231n.stanford.edu/reports/2022/pdfs/52.pdf>. (czas dostępu: 03.11.2022).