Sztuczna inteligencja i inżynieria wiedzy
K01-49i, czwartek 11:15

prowadzący
mgr inż. Katarzyna Fojcik



Politechnika
Wrocławska

Rozwiązywanie problemów przez przeszukiwanie

Lista 1

Tomasz Chojnacki
260365@student.pwr.edu.pl

WROCŁAW 23 marca 2023

Spis treści

1 Wstęp **2**
1.1 Tematyka sprawozdania 2
1.2 Analiza danych 2
1.3 Wybór języka 3

2 Przetwarzanie danych **4**
2.1 Transformacja grafu 4
2.2 Przetwarzanie pozycji 6
2.3 Reprezentacja grafu 8

3 Zadania **8**
3.1 Zadanie 1 8
3.1.1 Algorytm Dijkstry 9
3.1.2 Algorytm A* 12
3.2 Zadanie 2 18

4 Podsumowanie **24**

1 Wstęp

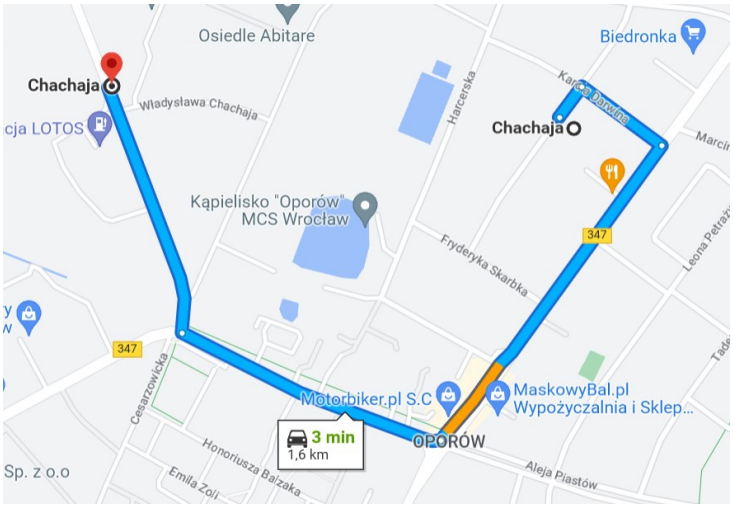
1.1 Tematyka sprawozdania

Tematem sprawozdania jest rozwiązywanie problemów przez przeszukiwanie. Celem ćwiczenia jest praktyczne przećwiczenie omawianych na wykładzie metod rozwiązywania problemów optymalizacyjnych [1]. Do listy zadań został dołączony plik `connection_graph.csv`, zawierający dane dotyczące przejazdów komunikacji miejskiej we Wrocławiu, który jest wykorzystywany w obu zadaniach.

1.2 Analiza danych

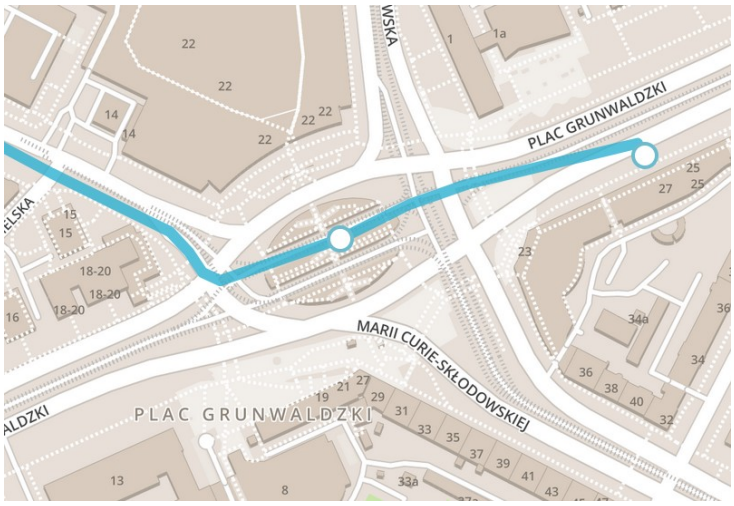
Przed wykonaniem zadań, zacząłem od wstępnej analizy pliku za pomocą Pythona. Celem rozeznania było lepsze poznanie struktury danych i wykrycie ewentualnych problemów.

Plik zawiera listę krawędzi grafu, gdzie wierzchołki są przystankami komunikacji miejskiej, a krawędzie modelują przejazdy autobusowe występujące pomiędzy nimi. Pierwszą istotną obserwacją jest to, że **dane tworzą multigraf**, ponieważ między dwoma przystankami występuje nawet kilkaset połączeń dziennie. Plik **nie zawiera linii nocnych**. Skutkuje to tym, że przy wyborze późnej godziny startowej, należy rozważyć też opcję czekania na kolejny autobus do rana. Kolejnym wnioskiem jest to, że przystanki o tej samej nazwie niekoniecznie znajdują się w tym samym miejscu. W skrajnych przypadkach, **przystanki o tej samej nazwie znajdują się nawet ponad 500m od siebie**. Rys. 1 przedstawia parę najbardziej oddalonych przystanków o tej samej nazwie – *Chachaja*, które znajdują się 713 metrów od siebie w linii prostej oraz aż 1,6 km trasą komunikacji miejskiej.



Rysunek 1: Położenie przystanków *Chachaja*.

W dodatku, trasa autobusu może **przechodzić przez różne przystanki o tej samej nazwie**. Przykładowo, linia C ma na swojej trasie dwa przystanki o nazwie *PL. GRUNWALDZKI*, co zostało przedstawione na Rys. 2. Plac Grunwaldzki jest prawdopodobnie najbardziej problematycznym przystankiem, ponieważ każdy jego peron ma inną lokalizację.



Rysunek 2: Trasa autobusu C w okolicach przystanku *PL. GRUNWALDZKI*.

Najdłuższą trasą (w linii prostej) w zbiorze danych jest *Kątna–Lubiatów*. Przystanki te są **oddalone o ponad 50 km** od siebie, transport między nimi zajmuje ponad 5 godzin i wymaga czterech przesiadek. Trasa ta została przedstawiona na Rys. 3.



Rysunek 3: Trasa *Kątna – Lubiatów*.

1.3 Wybór języka

Po przetwarzaniu postanowiłem wybrać język programowania do implementacji zadań. Zależało mi, aby był to język kompilowalny, z uwagi na rozmiar zestawu danych. Już przy wstępnym rozpoznaniu Python okazał się być momentami zbyt wolny. Drugą pożądaną przeze mnie cechą wybranego języka jest statyczna typizacja (ze wsparciem algebraicznych typów danych), którą preferuję przy większych projektach.

W związku z powyższym, postanowiłem wykonać implementację zadań w języku **Rust**. Jest to wieloparadygmatowy język kompilowalny, stanowiący alternatywę dla C/C++ [3]. Jedną z kluczowych cech Rusta jest zapewnienie bezpieczeństwa pamięci bez jednoczesnego wykorzystania garbage collectora, za pomocą wzorca RAII (podobnego do działania typów kontenerowych i wskaźników inteligentnych z C++). W związku z tym, między innymi Google zdecydował się na pisanie wszystkich nowych niskopoziomowych modułów Androida w tym języku. Ponadto, język ten wspiera wzorce funkcyjne lepiej niż większość niskopoziomowych języków, co jest przydatne, biorąc pod uwagę, że w zadaniu 1 pracujemy na różnych anonimowych funkcjach (funkcje kosztu, heurystyka, itd.).

W programie wykorzystałem pakiety: `colored` – do kolorowania treści wypisywanej w terminalu, `smol_str` – do optymalizacji małych ciągów tekstowych, `rand` – do losowania liczb. Żaden z nich nie wpływał istotnie na implementację.

2 Przetwarzanie danych

2.1 Transformacja grafu

W początkowym wariancie rozwiązania traktowałem każdy przystanek (biorąc pod uwagę, że dwa różne przystanki mogą mieć tę samą nazwę) jako osobny wierzchołek grafu, który był połączony z innymi wierzchołkami potencjalnie wielokrotnymi krawędziami zawierającymi numer linii oraz czas odjazdu.

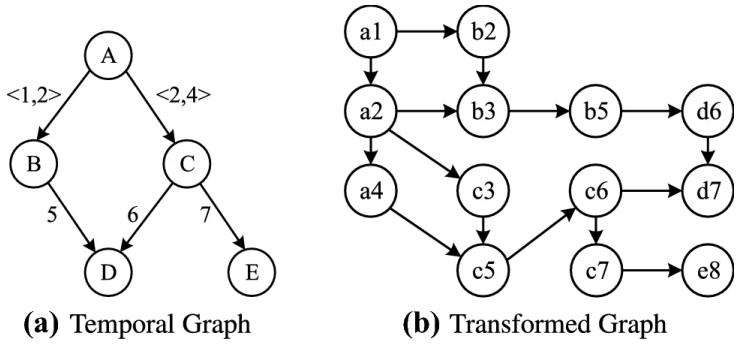
Jednakże to rozwiązanie skutkowało wieloma problemami implementacyjnymi:

- Operujemy na multigrafie, co ogranicza możliwości reprezentacji grafu w kodzie. Ponadto, część algorytmów grafowych nie działa na multigrafach bez dokonania modyfikacji. W szczególności, algorytm Dijkstry, a co za tym idzie A*, **nie działa w niektórych przypadkach dla multigrafów** [4].
- **Wagi krawędzi grafu są zmienne w czasie**, tzn. np. jeżeli między wierzchołkami A i B występuje przejazd o godzinie 11:15, który zajmuje 5 min, to o godzinie 11:05 krawędź ta będzie miała wagę 15, natomiast o 11:10 już tylko 10.
- Wymaga ono ręcznego uwzględniania konceptu upływu czasu w algorytmie Dijkstry oraz A*, co komplikuje implementację tych algorytmów oraz utrudnia generalizację algorytmu dla różnych rodzajów grafów.
- Przesiadki na trasie są trudne do wykrycia. Aby wykryć czy nastąpiła właśnie przesiadka czy nie musimy trzymać i analizować historię obecnie analizowanej ścieżki trzy przystanki (czyli dwa przejazdy – aby porównać linie) wstecz. Co więcej przesiadki pomiędzy autobusami tej samej linii (np. wysiadamy na przystanku i wsiadamy w autobus skierowany w przeciwnym kierunku) są praktycznie niewykrywalne.

Wszystkie te problemy wynikają z tego, że między dwoma przystankami występują jedynie przejazdy w konkretnych, dyskretnych momentach w czasie, w przeciwieństwie do prostszego i bardziej popularnego modelu, w którym lokalizacje są oddzielone trasami zajmującymi t czasu, ale dostępnymi w dowolnym momencie (taki model sprawdziłby się przykładowo dla przejazdu samochodem – gdzie prowadzimy my i możemy odjechać w dowolnej chwili, a nie jedziemy według rozkładu).

Najprostszym rozwiązaniem tego problemu jest uproszczenie grafu do opisanego powyżej rodzaju, jednakże sprawia to, że wyniki programu będą znacznie mniej dokładne, np. nie będą uwzględniały, że niektóre autobusy jeżdżą tylko raz dziennie, oraz że autobusy jeżdżą z różną częstotliwością w zależności od pory dnia.

Metody rozwiązania tego problemu bez utraty dokładności wyniku nie są szeroko opisywane w literaturze, natomiast jedno potencjalne rozwiązanie zostało przedstawione w artykule „Time-Dependent Graphs: Definitions, Applications, and Algorithms” [5]. Polega ono na transformacji multigrafu do klasycznego grafu rozbijając każdy wierzchołek na n kolejnych postaci `(przystanek, czas)`. Rozwiązuje to trzy z czterech powyższych problemów. Wagi krawędzi nie są już zmienne w czasie (przywołując przykład z góry, niezależnie od obecnej godziny mamy `(A,11:15)-[5min]->(B,11:20)`). Nie musimy przechowywać w trakcie działania algorytmu obecnego czasu, ponieważ numer wierzchołka identyfikuje jednoznacznie obecną godzinę. Akcja czekania na przystanku jest teraz modelowana zwykłym przejściem między wierzchołkami, np. `(A,15:11)-[6min]->(A,15:17)`. Ideę tej metody łatwiej zrozumieć wizualnie, dlatego Rys. 4 przedstawia przykładową transformację, zaczerpniętą wprost ze wspomnianego artykułu. Okazuje się, że ostatni problem – trudnego wykrywania przesiadek – można rozwiązać w analogiczny sposób, dodając (opcjonalną) linię do właściwości wierzchołka [6].



(a) Temporal Graph **(b)** Transformed Graph

Rysunek 4: Przykład transformacji grafu z artykułu [5].

Finalnie, każdy wierzchołek jest opisany przez krotkę `(przystanek, godzina, linia)`, natomiast krawędzie grafu nie przechowują żadnych danych. Analizowany graf przestaje być grafem przystanków, a staje się grafem stanów pasażera. Przejście krawędzią między wierzchołkami nie oznacza literalnego przejścia między przystankami, a zmianę stanu pasażera (np. oczekiwanie na przystanku, jazda autobusem, wsiadanie/wysiadanie z autobusu). Każdemu z takich przejść można przyporządkować trywialnie odległość, ale także czas i ilość przesiadek, przez co implementacja wszystkich algorytmów dla wariantu czasowego i przesiadkowego może być identyczna za wyjątkiem obranej funkcji kosztu. Przesiadka do innego autobusu tej samej linii jest liczona identycznie jak każda inna.

Takie podejście opiera się na licznych, wymienionych w bibliografii pracach, jak i na własnych doświadczeniach, ale przede wszystkim implementuje opisany w bardzo ciekawym filmie „Theseus and the Minotaur — Exploring State Space" koncept *przestrzeni stanów*, opierający się na modelowaniu wierzchołkami grafu stanów, a nie pozycji [7].

Główną wadą tego rozwiązania jest wzrost rozmiaru grafu – każdy przystanek tworzy kilkanaście do kilkuset wierzchołków. Jednakże nawet przy wstępnej, naiwnej implementacji, nie spowodowało to problemów z pamięcią ani czasem wykonywania programu.

Reprezentacja wierzchołków (czyli stanu pasażera) w kodzie została przedstawiona na Lis. 1. Implementacja uwzględnia to, że przystanki o różnych nazwach mogą mieć różne położenia.

```
struct Node { stop: Stop, time: Time, line: Option<SmolStr> }

struct Stop { name: Rc<str>, pos: Pos }
```

Listing 1: Reprezentacja wierzchołków w kodzie.

Listing ukazuje finalną budowę tych struktur, po optymalizacji dokonanej w ostatnim podpunkcie zadania pierwszego. W jej wyniku, nazwa przystanku jest trzymana w strukturze zliczającej referencje `Rc<str>`, aby uniknąć duplikatowych alokacji – każdy przystanek znajduje się w wielu wierzchołkach, a jako że nazwa przystanku jest stała, każdy z nich może korzystać z referencji do jednego łańcucha (zaimplementowałem w tym celu własną strukturę, realizującą uproszczone internowanie łańcuchów [8]). Natomiast numer linii jest trzymany w strukturze `SmolStr`, która jest zoptymalizowana pod względem przechowywania tekstu krótszego niż 24 bajty (domyślny rozmiar struktury `String` na stosie), a numer linii jest taki zawsze.

Można podzielić krawędzie grafu na cztery intuicyjne kategorie w zależności od parametrów wierzchołków końcowych. Podział ten został przedstawiony na Lis. 2. Warto zaznaczyć, że struktura ta **nie jest** przechowywana w reprezentacji grafu. Jest jedynie tworzona w poszczególnych algorytmach, aby zwiększyć czytelność kodu i w większości przypadków jest to całkowicie wyoptymalizowane przez kompilator.

```
enum Edge<'a> {
    Wait { // czekanie na przystanku
        at_stop_name: &'a str, from_time: Time, to_time: Time,
        // przejście między przystankami o tej samej nazwie
        distance_km: f32,
    },
    Ride { // jazda autobusem pomiędzy przystankami
        on_line: &'a SmolStr,
        from_stop: &'a Stop, from_time: Time,
        to_stop: &'a Stop, to_time: Time,
    },
    Enter { // wsiadanie do autobusu
        line: &'a SmolStr, at_stop: &'a Stop, at_time: Time,
    },
    Leave { // wysiadanie z autobusu
        line: &'a SmolStr, at_stop: &'a Stop, at_time: Time,
    },
}
```

Listing 2: Reprezentacja krawędzi w kodzie.

2.2 Przetwarzanie pozycji

Domyślna, znajdująca się w pliku reprezentacja położenia przystanku jest problematyczna. Są to koordynaty w formie WGS-84, zawierające szerokość i długość geograficzną danego punktu. Niektóre z wad takiej reprezentacji na cele analizy ruchu komunikacji miejskiej to:

- Nietrywialne liczenie odległości między punktami. Odjęcie od siebie wartości kątowych odpowiednich współrzędnych **nie daje** miary odległości między tymi punktami. Należy wykorzystać wzór na odległość (haversine), którego obliczenia mogą być kosztowne.
- Przedstawienie punktu w przestrzeni trójwymiarowej, gdy na nasze potrzeby można założyć, że obszar Wrocławia jest płaski. Skutkuje to dodatkową złożonością implementacji.
- Trudność odczytu wartości na cele testowania, debugowania i prezentacji wyników. Poszczególne wartości różnią się dopiero dalekimi miejscami po przecinku. Analiza faktycznego znaczenia wartości położenia wymaga użycia zewnętrznych narzędzi (np. mapy).
- Zmienna wartość błędu. Błąd wynikający z odjęcia poszczególnych współrzędnych kątowych nie jest stały, zmienia się wraz z faktyczną odległością między tymi punktami.
- Duże zużycie pamięci. Naiwny sposób przechowywania współrzędnych może zająć aż do 64 bajtów na przystanek.

W związku z tym postanowiliśmy, we współpracy z kolegą siedzącym obok na zajęciach laboratoryjnych, przekształcić ten zestaw współrzędnych do dwuwymiarowego układu kartezjańskiego, wyśrodkowanego w centrum Wrocławia, z kilometrem pełniącym rolę jednostki odległości. Proces ten składa się z poniższych kroków:

1. Przetłumaczenie współrzędnych geograficznych na punkt 3D na sferze powierzchni Ziemi.
2. Rzut punktu 3D na płaszczyznę 2D styczną do powierzchni Ziemi w centrum Wrocławia.

Pierwszy krok procesu jest prosty i wymaga jedynie przemnożenia promienia Ziemi przez odpowiednie wartości funkcji trygonometrycznych długości i szerokości. Wykorzystany wzór wymaga zamiany wartości kątowych ze stopni na radiany. Przybliżoną wartość promienia ziemi w okolicy Wrocławia obliczyłem za pomocą kalkulatora internetowego [9]. Wzór opisujący tę transformację znajduje się w równaniu (1).

$$\begin{aligned} x &= R \cdot \cos(\phi) \cdot \cos(\lambda) \\ y &= R \cdot \cos(\phi) \cdot \sin(\lambda) \\ z &= R \cdot \sin(\phi) \end{aligned} \tag{1}$$

Drugi krok wymaga najpierw wyznaczenia konkretnego układu współrzędnych. Na ten cel potrzebujemy wektora normalnego płaszczyzny (prostopadłego do płaszczyzny, najprościej go obliczyć biorąc wektor od jądra Ziemi do lokalizacji Wrocławia) oraz wektorów baz nowego układu (wektorów jednostkowych, prostopadłych, leżących na płaszczyźnie). Obie te wartości są stałe i mogą zostać obliczone tylko raz dla wszystkich punktów (i wszystkich uruchomień programu). Za wektor normalny płaszczyzny przyjmujemy wektor prowadzący z pozycji $(0, 0, 0)$ (środka Ziemi) do pozycji Wrocławskiego Rynku. Jest to lokalizacja wybrana symbolicznie. Każdy inny punkt znajdujący się w okolicy Wrocławia również by zadziałał. Wektory bazowe możemy obliczyć na podstawie arbitralnie wybranego wektora „ziarna” [10]. Proces ten opisuje równanie (2).

$$\begin{aligned} e_1 &= n \times v_0 \\ e_2 &= n \times e_1 \end{aligned} \tag{2}$$

Finalnie, tworzymy rzut punktu 3D z części pierwszej na nowo utworzoną płaszczyznę. Po uproszczeniach wzoru [11], otrzymujemy równanie (3) na końcowe współrzędne punktu.

$$\begin{aligned} k &= -n \cdot v \\ v' &= v + kn \\ s_1 &= e_1 \cdot v' \\ s_2 &= e_2 \cdot v' \end{aligned} \tag{3}$$

W zakresie implementacji, stworzyłem strukturę `PosConverter` przedstawioną w Lis. 3, która oblicza leniwie stałe układu współrzędnych i wykorzystuje je przy kolejnych konwersjach, przyspieszając proces wczytywania danych. Do obliczeń wykorzystałem napisaną na cele listy małą bibliotekę do obliczeń wektorowych, zawierającą takie obliczenia jak iloczyn skalarny, iloczyn wektorowy, normalizacja wektora i obliczanie długości.

```
struct PosConverter {
    normal: Vec3,
    e1: Vec3,
    e2: Vec3,
}

impl PosConverter {
    fn initialize() -> Self { /* ... */ }
    fn wgs84_to_pos(&self, lat: &str, lon: &str) -> Pos { /* ... */ }
}
```

Listing 3: Struktura przyspieszająca konwersję współrzędnych.

Finalnie, pozycja przystanku jest przechowywana w strukturze `Pos`, będącej abstrakcją nad parą liczb zmiennoprzecinkowych pojedynczej precyzji (`f32`), tworzonej przez obiekt `PosConverter`.

Takie wartości pozbywają się wszystkich wad współrzędnych geograficznych: są tanie w przechowaniu (cała struktura zajmuje jedynie 8 bajtów), obliczanie odległości jest łatwe i tanie (możemy zastosować zwykłe wzory na odległość Euklidesową), są łatwe w odczycie (wartość `Pos(1, 0)` znajduje się dokładnie 1 km od Rynku). Błąd odległości tego rozwiązania, w porównaniu z miernikiem odległości wbudowanym Google Maps, nie przekracza 2 cm, co jest wartością pomijalną, biorąc pod uwagę, że jest kilkuset razy krótsza od długości wiaty przystankowej.

2.3 Reprezentacja grafu

Jako reprezentację grafu w pamięci wybrałem listę sąsiedztwa. Wariant z macierzą sąsiedztwa miałby w tym przypadku większe zużycie pamięci (ponieważ graf nie jest w ogóle gęsty). Dodatkowo, reprezentacja ta zapewnia możliwość bardzo szybkiego znalezienia sąsiadów danego wierzchołka, co jest najczęściej wykonywaną akcją w implementowanych algorytmach. Metadane wierzchołków przechowuję w osobnej strukturze, aby utrzymać mały rozmiar listy sąsiedztwa. Ostatnim elementem reprezentacji grafu jest mapa tłumacząca nazwy przystanków na numery odpowiadających im wierzchołków, utworzona w celu przyspieszenia początkowej fazy algorytmu Dijkstry oraz A*. Pełna reprezentacja grafu w pamięci została przestawiona na Lis. 4.

```
type NodeIndex = usize;

type AdjList = Vec<Vec<NodeIndex>>;

struct BusNetwork {
    adj_list: AdjList,
    nodes: Vec<Node>,
    name_lookup: HashMap<Rc<str>, Vec<NodeIndex>>,
}
```

Listing 4: Reprezentacja grafu w pamięci programu.

3 Zadania

3.1 Zadanie 1

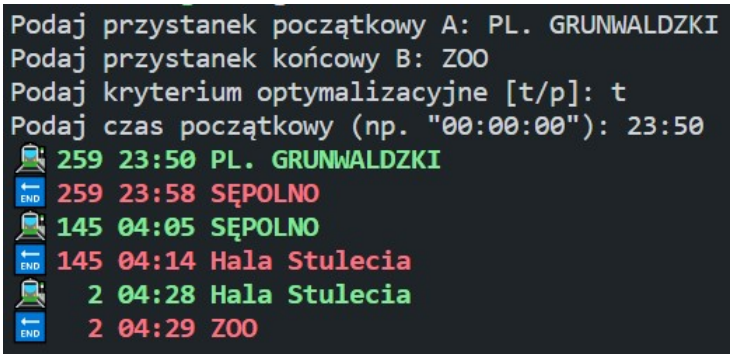
Zadanie 1 polegało na opracowaniu programu do znajdywania najbardziej optymalnych (pod względem czasu podróży oraz ilości przesiadek) tras pomiędzy dwoma dowolnymi przystankami. Program powinien dostać na wejściu: przystanek początkowy A, przystanek końcowy B, kryterium optymalizacyjne (t – czas, p – ilość przesiadek) oraz czas pojawienia się na przystanku początkowym [1].

Na potrzeby zadania założyłem, że jako przystanek początkowy i końcowy mogą służyć **dowolne przystanki o danej nazwie**. Oznacza to przykładowo, że jeżeli podamy na wejściu *PL. GRUNWALDZKI* oraz *DWORZEC GŁÓWNY*, program znajdzie najkrótszą trasę między dowolnymi dwoma pasującymi przystankami. W tym podejściu, przystanki o tej samej nazwie ale innej lokalizacji są traktowane jako oddzielne. Najbardziej kompleksowym rozwiązaniem byłoby pokazanie użytkownikowi po wpisaniu nazwy przystanku wszystkich kandydatów (wraz ze współrzędnymi i liniami odjeżdżającymi z każdego z nich), jednakże uważam, że jest to rozwiązanie wykraczające poza zakres zadania.

Kolejną konsekwencją podziału przystanków było dodanie możliwości **przechodzenia między przystankami o tej samej nazwie**. Bez takiego rozwiązania czasy podróży rosną nierealistycznie, ponieważ niedozwolona jest np. zmiana peronu na Placu Grunwaldzkim. Na potrzeby zadania założyłem, że podróż między dwoma przystankami o tej samej nazwie zajmuje dokładnie minutę. Jest to kompromis pomiędzy w pełni uproszczonym rozwiązaniem zakładającym

teleportację między przystankami a realistyczną symulacją. Lepszym rozwiązaniem byłoby dodanie możliwości przejścia między dowolnymi dwoma przystankami, z uwzględnieniem prędkości chodu, jednakże jest to również rozwiązanie wykraczające poza zakres polecenia.

Ciekawym przypadkiem brzegowym jest akcja **czekania na kolejny dzień**. W przypadku wyboru bardzo późnej godziny początkowej, program jest zmuszony poczekać aż do czasu kursowania porannych autobusów. Możliwa jest też sytuacja, w której przejedziemy część trasy, a następnie dokończymy podróż rano, przedstawiona na Rys. 5. O dowolnej wcześniejszej godzinie opłacalne jest wzięcie tramwaju linii 2 bezpośrednio między przystankami *PL. GRUNWALDZKI* a *ZOO*, jednakże nie kursuje on już o godzinie 23:50. Dlatego bierzemy autobus 259, który zabiera nas w inne miejsce, ale takie, z którego podróż rano jest szybsza niż z Placu Grunwaldzkiego.



Rysunek 5: Przykładowa podróż zawierająca oczekiwanie na kolejny dzień.

3.1.1 Algorytm Dijkstry

Pierwszy podpunkt zadania polegał na wykorzystaniu algorytmu Dijkstry do znajdywania drogi zajmującej najmniej czasu pomiędzy dwoma przystankami.

Algorytm Dijkstry jest algorytmem znajdowania najkrótszych ścieżek w grafie o nieujemnych wagach z jednym źródłem. Algorytm opiera się na utrzymywaniu zbioru wierzchołków o najkrótszej odległości od źródła oraz aktualizacji tego zbioru wraz z odwiedzaniem kolejnych wierzchołków [1]. Obliczana jest odległość od wierzchołka źródłowego do *wszystkich* innych wierzchołków, jednakże wykonywanie algorytmu można zakończyć przedwcześnie, jeżeli zależy nam na tylko na odległości do kilku (lub jednego) wierzchołków. Algorytm Dijkstry działa w grafach nieskierowanych jak i skierowanych. Algorytm ten znajduje **zawsze optymalne** rozwiązanie problemu (o ile wagi krawędzi są nieujemne) [12].

Główną trudnością tego zadania był prawdopodobnie fakt, że, jak zostało wspomniane wyżej, algorytm Dijkstry nie działa na multigrafach [4] i wymagana byłaby modyfikacja. Na szczęście odpowiedni dobór reprezentacji grafu opisany we wcześniejszych sekcjach mnie przed tym uchronił. Implementację zacząłem od przetłumaczenia zapewnionego przez prowadzącą kodu z Pythona na język Rust. Następnie wprowadziłem minimalne zmiany takie jak:

- Wybór wierzchołka początkowego i końcowego na podstawie nazwy a nie indeksu.
- Uwzględnienie czasu pomiędzy początkiem poszukiwań a pierwszym odjazdem autobusu z przystanku początkowego w wyniku.
- Pomiar czasu wykonywania algorytmu, wymagany w poleceniu.

Największym wyzwaniem przy realizacji podpunktu było naprawienie problemów z działaniem kolejki priorytetowej. Struktura kopca binarnego, takiego jak zapewniony przez Pythona `heapq` istnieje w bibliotece standardowej Rusta, natomiast jest to kopiec typu *max*, a nie *min* [13].

Skutkuje to wyciąganiem z kolejki w pierwszej kolejności obiektów o najwyższej a nie najmniejszej wartości dystansu. Rozwiązanie tego problemu znalazłem bezpośrednio w dokumentacji tej struktury, gdzie szczęśliwym trafem jako przykład zastosowania modułu jest przedstawiony algorytm Dijkstry. Sposób rozwiązania błędu polega na zdefiniowaniu struktury o odwrotnej kolejności sortowania, co jest możliwe dzięki bogatemu systemowi typów tego języka.

Poszedłem nawet o krok dalej, tworząc rozwiązanie generyczne ponad wszystkimi typami, jakie można użyć jako koszt krawędzi. Jest to możliwe dzięki systemowi „cech" wbudowanemu w język, zainspirowanemu takimi językami jak Haskell. Zapewnia on szereg cech, o których można myśleć jak o bardziej potężnych interfejsach, które opisują możliwości danych typów (np. możliwość dodawania, odejmowania czy zamieniania w łańcuch tekstowy). Typ kosztu jest wtedy opisany w sposób przedstawiony na Lis. 5, co można przeczytać jako „typ `Cost` to dowolny typ implementujący jednocześnie cechy: `Sized` (ma określony, stały rozmiar w pamięci), `Copy` (może być tanio klonowany – poprzez skopiowanie reprezentacji bitowej wartości), `Default` (ma domyślną wartość – „zero"), `Add<Output = Self>` (może być dodany do innych wartości swojego typu), `PartialEq` (ma relację równości z niektórymi wartościami swojego typu), `PartialOrd` (ma relację porządku – mniejszy/większy – z niektórymi wartościami swojego typu), `Display` (typ może być zmieniony w `String`)". Istotne jest wykorzystanie cech `PartialEq`/`PartialOrd`, a nie `Eq`/`Ord`, ponieważ typy zmiennoprzecinkowe nie definiują równości na całym zbiorze wartości (w szczególności, `NaN != NaN`) [14].

```rust
impl<T> Cost for T where
    T: Sized + Copy + Default +
    Add<Output = Self> + PartialEq +
    PartialOrd + Display {}
```

Listing 5: Definicja cechy kosztu.

Taka implementacja pozwala wykorzystać wszystkie typy numeryczne jako koszt krawędzi, ale też szereg innych typów, jednocześnie raportując ewentualne błędy na etapie kompilacji, a nie działania programu. Wymaga ona też od programisty obsłużenia przypadków skrajnych, np. kompilator wymusił na mnie potwierdzenie, że wartości `NaN` nie mogą być kosztem.

Pełna implementacja algorytmu Dijkstry znajduje się na Lis. 7. Kod odpowiedzialny za rekonstrukcję ścieżki został wydzielony do osobnej metody, ponieważ jest też używany w implementacji algorytmu A* oraz Tabu. Został on przedstawiony na Lis. 6.

```rust
fn reconstruct_edges<'s>(
    &'s self, parents: &HashMap<NodeIndex, NodeIndex>, to: NodeIndex,
) -> Vec<Edge<'s>> {
    let mut current = to;
    let mut path = VecDeque::from([to]);
    while let Some(&parent) = parents.get(&current) {
        path.push_front(parent);
        current = parent;
    }
    path.make_contiguous().windows(2)
        .map(|e| Edge::from(&self.nodes[e[0]], &self.nodes[e[1]]))
        .collect::<Vec<_>>()
}
```

Listing 6: Kod odpowiedzialny za rekonstrukcję ścieżki.

```rust
fn dijkstra_time<'bn>(
    bn: &'bn BusNetwork,
    start_name: &str,
    start_time: Time,
    end_name: &str,
) -> Option<Path<'bn, u32>> {
    let instant = Instant::now();
    let start = bn.find_node_index(start_name, start_time)?;
    let mut costs = HashMap::with_capacity(bn.order());
    let mut parents = HashMap::with_capacity(bn.order());
    let mut queue = BinaryHeap::new();
    costs.insert(start, 0);
    queue.push(State { cost: 0, node: start });
    while let Some(cur) = queue.pop() {
        if bn.is_valid_stop(cur.node, end_name) {
            return Some(Path {
                edges: bn.reconstruct_edges(&parents, cur.node),
                cost: cur.cost,
                runtime: instant.elapsed(),
            });
        } else if Some(&cur.cost) > costs.get(&cur.node) { continue; }
        for neighbour in bn.neighbours(cur.node) {
            let edge = Edge::from(bn.node(cur.node), bn.node(neighbour));
            let new_cost = cur.cost + edge.time_min();
            if !costs.contains_key(&neighbour) || new_cost < costs[&neighbour] {
                costs.insert(neighbour, new_cost);
                parents.insert(neighbour, cur.node);
                queue.push(State { cost: new_cost, node: neighbour });
            }
        }
    }
    None
}
```

Listing 7: Kod algorytmu Dijkstry.

Polecenie nie opisuje, w jakim miejscu programu wynikowego ma znajdować się wywołanie algorytmu Dijkstry, a wręcz sugeruje, że nie powinno się tam znajdować. Jednakże na cele testowania jak i demonstracji wykonanego polecenia, do programu pierwszego dodałem nowe kryteria, jednocześnie podnosząc opcję wyboru kryterium z `[t/p]` do:

- `t` - optymalizacja czasu algorytmem A*
- `p` - optymalizacja przesiadek algorytmem A*
- `d` - optymalizacja dystansu algorytmem A*
- `dt` - optymalizacja czasu algorytmem Dijkstry
- `dp` - optymalizacja przesiadek algorytmem A*
- `dd` - optymalizacja dystansu algorytmem A*

3.1.2 Algorytm A*

Podpunkty (b) i (c) zadania 1 polegały na implementacji algorytmu A* do znalezienia tras o najkrótszym czasie przejazdu bądź najmniejszej ilości przesiadek.

Algorytm A* jest heurystycznym algorytmem służącym do znajdowania najkrótszej ścieżki w grafie. Jest rozbudowaniem algorytmu Dijkstry o estymację kosztu ścieżki od celu. Algorytm ten przy nieprawidłowo zaprojektowanej heurystyce może osiągnąć nieoptymalne wyniki [1].

Istotną częścią algorytmu A* jest heurystyka $h(x)$, czyli funkcja, która przewiduje koszt trasy od wierzchołka x do celu. Heurystykę nazywamy **dopuszczalną**, jeżeli jej wartość jest zawsze mniejsza bądź równa od rzeczywistej odległości do węzła końcowego (czyli daje optymistyczne przybliżenia odległości do węzła końcowego) [15].

A* zawsze znajduje **optymalne rozwiązanie, jeżeli** [2]:

- rozwiązanie istnieje,
- każdy węzeł ma skończoną liczbę sąsiadów,
- wagi krawędzi są nieujemne,
- heurystyka jest dopuszczalna.

Jeżeli funkcja heurystyki jest zawsze równa zero ($h(x) = 0$), to algorytm degeneruje do algorytmu Dijkstry. Każda wyższa wartość przyspiesza działanie algorytmu kierunkując go w stronę wierzchołka docelowego. Dzieje się tak aż do momentu, kiedy heurystyka jest równa rzeczywistemu kosztowi od wierzchołka x do końca – w tym momencie algorytm działa najlepiej, odwiedza kolejno wierzchołki ścieżki wynikowej nie próbując skorzystać z żadnych innych ścieżek. Niestety jest to wariant nieosiągalny w praktycznym użytku – aby heurystyka wskazywała wartość idealną musimy tą odległość obliczyć... algorytmem Dijkstry lub A*, w którym wykorzystujemy tę heurystykę, którą należy policzyć. Przekraczając punkt tej równości heurystyka przestaje być dopuszczalna i tracimy gwarancję poprawnego wyniku. Każda wyższa wartość heurystyki wpływa pozytywnie na czas działania algorytmu ale negatywnie na jego wynik.

Implementacja algorytmu A* na przetransformowanym grafie okazała się trywialna. Kod udostępniony nam przez platformę MS Teams działał po bezpośrednim przetłumaczeniu na inny język programowania. Łatwość tego zadania wynika też z tego, że implementacja algorytmu A* różni się od Dijkstry zaledwie kilkoma linijkami.

Moja implementacja, przedstawiona na Lis. 8 jest generyczna nad wszystkimi funkcjami kosztu oraz heurystykami. Pozwala to na eksperymentowanie z różnym doborem kosztu i heurystyki w różnych celach. Dzięki temu rozwiązanie podpunktu (b) i (c) jest praktycznie takie samo. Jedyna różnica polega na wyborze funkcji kosztu liczącej czas lub liczącej autobusy oraz odpowiedniej heurystyki.

Funkcja wykorzystuje w pełni generyczne wartości kosztu, przez co zera zostały zastąpione wywołaniami `C::default()`, zwracającymi domyślną wartość typu kosztu. Wywołanie tej funkcji jest przez kompilator wyoptymalizowane w procesie monomorfizacji, przez co rozwiązanie generyczne jest równie szybkie co wykorzystanie konkretnych typów.

Przeniesione są tu również wszystkie szczegóły implementacyjne z rozwiązania podpunktu (a), tzn. przekształcenie kopca *max* w kopiec *min*, mierzenie czasu wykonywania programu i rekonstrukcja ścieżki. Dodatkowo, została wykorzystująca funkcja pomocnicza wyszukująca przystanek końcowy na podstawie jego nazwy, która była zbędna w algorytmie Dijkstry (w którym nie musimy znać indeksu przystanku końcowego aż do uzyskania wyniku).

```rust
fn astar<'bn, C, CF, HF>(
    bn: &'bn BusNetwork,
    start_name: &str, start_time: Time,
    end_name: &str,
    cost_fn: CF, heuristic_fn: HF,
) -> Option<Path<'bn, C>>
    where C: Cost, CF: Fn(&Edge) -> C, HF: Fn(&Node, &Stop) -> C
{
    let instant = Instant::now();
    let start = bn.find_node_index(start_name, start_time)?;
    let end_stop = bn.find_stop(end_name)?;
    let mut costs = HashMap::with_capacity(bn.order());
    let mut parents = HashMap::with_capacity(bn.order());
    let mut queue = BinaryHeap::new();
    costs.insert(start, C::default());
    queue.push(State { cost: C::default(), node: start });

    while let Some(State { node, .. }) = queue.pop() {
        if bn.is_valid_stop(node, end_name) {
            return Some(Path {
                edges: bn.reconstruct_edges(&parents, node),
                cost: costs[&node],
                runtime: instant.elapsed(),
            });
        }

        for neighbour in bn.neighbours(node) {
            let edge = Edge::from(bn.node(node), bn.node(neighbour));
            let new_cost = costs[&node] + cost_fn(&edge);

            if !costs.contains_key(&neighbour) || new_cost < costs[&neighbour] {
                let priority = new_cost + heuristic_fn(bn.node(neighbour), end_stop);
                costs.insert(neighbour, new_cost);
                parents.insert(neighbour, node);
                queue.push(State { cost: priority, node: neighbour });
            }
        }
    }
    None
}
```

Listing 8: Kod algorytmu A*.

W przypadku czasu **można utworzyć heurystykę dopuszczalną**. Potrzebujemy optymistycznego przybliżenia czasu podróży z dowolnego punktu do dowolnego innego punktu. Jest nim czas trwania ruchu jednostajnego prostoliniowego między tymi punktami z prędkością równą maksymalnej dopuszczalnej prędkości. Rozwiązanie zakłada, że autobusy i tramwaje nie przekraczają prędkości 80 km/h w mieście (co jest maksymalną prędkością, do której można podnieść domyślne ograniczenie prędkości w obszarze zabudowanym [16]), co zostało sprawdzone również na zbiorze danych drogą eksperymentalną. Będziemy wtedy mieli pewność, że algorytm znajduje poprawne wyniki, ponieważ pozostałe trzy warunki są spełnione.

Jest to jedno z miejsc, w których dokonana transformacja współrzędnych znacznie ułatwia problem. Odjęcie wartości kątowych nie zapewniałoby tu poprawności rozwiązania. Natomiast aby uzyskać dystans obecnego punktu od punktu końcowego w przekształconym układzie współrzędnych wystarczy policzyć ich odległość Euklidesową. Następnie zamieniamy wartość w kilometrach na czas trwania takiej trasy w minutach. Bezpośrednie dodanie dwóch wartości o różnych jednostkach może sprawić, że algorytm przestanie być dopuszczalny. Konwersji dokonujemy następująco:

```
next.stop.pos.distance_km(end.pos) / (expected_speed_kmph / 60.) // t = s/v
```

Wartość 60 w powyższym równaniu oznacza ilość minut w godzinie. Wykorzystując podaną wyżej heurystykę uzyskujemy wyniki takie same jak w przypadku algorytmu Dijkstry, jednakże obliczone znacznie szybciej. W skrajnych przypadkach czas obliczeń spada z około 50 ms do 1 ms. Generalnie czas wykonywania algorytmów jest bardzo niski z uwagi na wykorzystanie języka kompilowalnego.

W przypadku liczby przesiadek zdecydowałem się zmienić minimalnie funkcję kosztu. Polecenie wskazuje, że należy liczyć liczbę przesiadek, jednakże takie rozwiązanie cechuje jedna wada występująca w przypadkach brzegowych. Liczba przesiadek nie odróżnia wariantu jazdy jednym autobusem od braku jazdy jakąkolwiek linią. To znaczy, jeżeli punkt początkowy jest równy końcowemu, rozwiązanie $START \rightarrow STOP$ ma taką samą wartość funkcji kosztu co $START \rightarrow$ wejście w linię A $\rightarrow$ okrążenie całego Wrocławia $\rightarrow$ wyjście z linii A $\rightarrow STOP$. Zmieniając funkcję kosztu z *ilość przesiadek* na ***ilość wykorzystanych autobusów*** rozwiązujemy ten problem, a jednocześnie otrzymujemy takie samo optymalne rozwiązanie w sytuacjach ogólnych. Ponieważ $changes = max(0, buses - 1)$, minimalizując ilość autobusów minimalizujemy jednocześnie ilość przesiadek.

O wiele trudniejsze jest utworzenie tutaj dopuszczalnej heurystyki. Wymagałoby to znalezienia funkcji dającej optymistyczne zaokrąglenie liczby przesiadek między dowolnymi dwoma punktami. Jedyną sensowną funkcją działającą w czasie stałym jest $h(x) = 0$, co jest dopuszczalną heurystyką, ale jednocześnie degeneruje A* do algorytmu Dijkstry, co prawdopodobnie nie było zamiarem autorów listy.

Zdecydowałem się zatem na implementację zestawu kilku niedopuszczalnych heurystyk i wybranie najlepszej z nich. Utworzone i przetestowane przeze mnie heurystyki to:

- `Disabled` – heurystyka $h(x) = 0$, degenerująca algorytm do Dijkstry. Używana jako przypadek bazowy – jako że mamy do czynienia z algorytmem Dijkstry wynik będzie na pewno poprawny, natomiast czas wykonywania będzie dość długi. Jeżeli inna heurystyka zapewnia wynik o takim samym koszcie, ale dłuższym czasie wykonywania, to nie warto z niej korzystać. Podobnie, jeżeli wynik ma znacznie wyższy koszt niż ta, należy z niej najprawdopodobniej zrezygnować.

- `Distance { changes_per_km: f32 }` – ilość przesiadek skalowana liniowo z pozostałym dystansem. Opiera się na uproszczeniu, że przystanki znajdują się w stałych odległościach od siebie i że przesiadki pasażera występują równomiernie na całej trasie.

- `StopNodes { weight: f32 }` – preferuj wybieranie przystanków, które mają wiele połączeń. Pasażer częściej (regulowane parametrem) wysiada na przystankach, z których wychodzi wiele połączeń. Metoda opiera się na założeniu, że przesiadki częściej występują na wielkich przystankach.

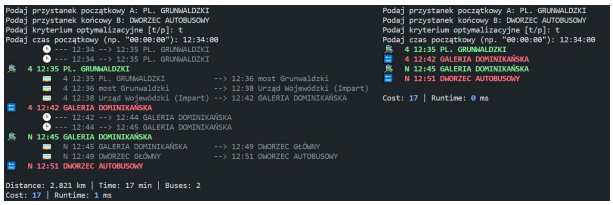
- `PreferMajorStops { penalty: u32 }` – preferuj wybieranie przystanków nazwanych wielkimi literami. Niektóre przystanki mają nazwy napisane samymi wielkimi literami. Są to zazwyczaj kluczowe przystanki, z których wyjeżdża wiele autobusów lub przystanki końcowe. Heurystyka dolicza karę za wysiadanie na przystankach innych niż kluczowe, bazując na tym samym założeniu co poprzednia metoda.

- `AvoidExpressLines { penalty: u32 }` – unika linii przyspieszonych. Linie przyspieszone nazwane są literami a nie numerami. Linie te zazwyczaj jeżdżą taką samą trasą co inne linie, ale odwiedzają mniej przystanków po drodze. Unikając przesiadek opłacalne może być zatem unikanie tych linii i zamiast tego wybieranie alternatyw odwiedzających więcej przystanków. Każde skorzystanie z linii przyspieszonej wiąże się z parametryzowaną karą.

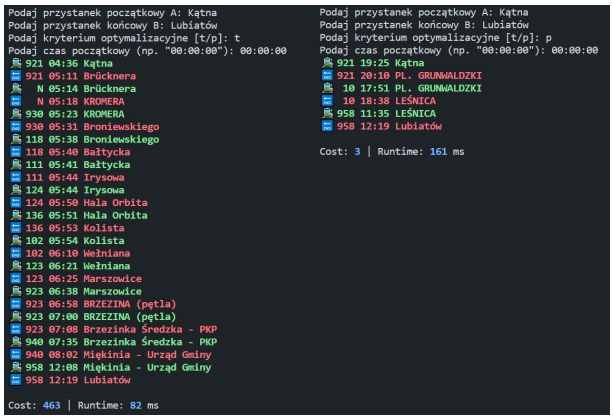
Wszystkie heurystyki (przy odpowiednim wyborze parametrów) dawały wyniki zbliżone do optymalnych. Jednakże czas wykonywania algorytmu wahał się w zależności od wybranego punktu początkowego i końcowego trasy. Heurystyka `Distance` była jedyną, która konsekwentnie dawała wyniki lepsze od przypadku bazowego. W związku z tym, w finalnej implementacji algorytmu szukającego tras z najmniejszą ilością przesiadek wykorzystałem tę heurystykę z parametrem 0.1 przesiadek na kilometr.

Warto zauważyć, że obranie za funkcję kosztu jedynie ilości przesiadek może dawać bardzo nieintuicyjne (ale poprawne) wyniki. Przykładowo, w przypadku połączenia bezpośredniego, wejście w pierwszy autobus i pojechanie do celu ma identyczny koszt co poczekanie na przystanku trzech dni po czym wejście do tego samego autobusu. Rozwiązaniem tego problemu jest doliczanie w koszcie również czasu (przykładowo każda minuta podróży to koszt jeden, a każda przesiadka ma koszt 60 – wtedy czekamy maksymalnie godzinę na przystanku jeżeli miałoby to zapobiec przesiadce). Jeszcze lepszym rozwiązaniem, na które pozwala generyczna cecha `Cost` jest użycie jako kosztu krotki `(przesiadki, czas)`, wtedy w pierwszej kolejności jest optymalizowana bezwzględnie ilość przesiadek, a następnie wybieramy trasę trwającą najkrócej z dostępnych. Taki wariant musiałby zostać wykorzystany w prawdziwej aplikacji, jednakże w wersji zademonstrowanej w raporcie wykorzystałem wariant najprostszy, ponieważ pokazuje on ciekawą zależność, a jednocześnie jest najbardziej zgodny z poleceniem.

Przykładowe wywołania programu znajdują się poniżej. W zależności od tego, czy program jest kompilowany w trybie debugowania czy zoptymalizowanym, wyświetla on pełną ścieżkę (wszystkie odwiedzone przystanki, w tym przystanki, na których pasażer nie podejmuje żadnej decyzji), bądź jedynie podsumowanie podróży (przystanki, gdzie pasażer wsiada lub wysiada, zgodnie z poleceniem zadania). Porównanie tych trybów znajduje się na Rys. 6.



Rysunek 6: Porównanie trybów wypisu wyniku.



Rysunek 7: Porównanie optymalizacji pod kątem czasu i ilości przesiadek.



Rysunek 8: Długa trasa przedstawiona w widoku szczegółowym.

Celem podpunktu (d) było zmodyfikowanie części (b) lub (c) tak, aby uzyskiwała szybsze, bądź bardziej dokładne wyniki. Zdecydowałem się na optymalizację wariantu czasowego, ponieważ łatwiej jest dla niego sprawdzić poprawność wyników. Ponieważ trudnym zadaniem jest przyspieszenie algorytmu wykonującego się w kilka milisekund, zacząłem od optymalizacji pamięciowej, której szczegóły implementacyjne opisałem w rozdziale o tranformacji grafu. Pozwoliło to zmniejszyć wykorzystaną przez program pamięć trzykrotnie i przyspieszyło fazę transformacji grafu, natomiast nie miało znacznego wpływu na samo wyszukiwanie.

Najprostszą optymalizacją pod względem czasu wykonywania programu jest pomijanie części wierzchołków/krawędzi, jednakże takie rozwiązanie może zmniejszyć jakość wyników lub w skrajnych przypadkach nawet nie pozwolić na znalezienie rozwiązania. Zamiast tego postanowiłem zoptymalizować heurystykę.

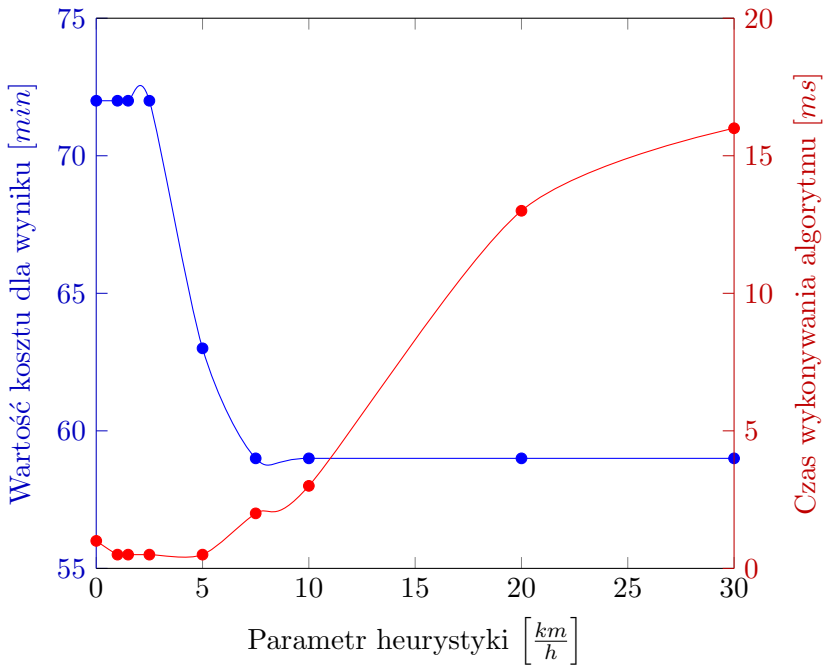
Idea opiera się na parametryzacji heurystyki odległościowej różnymi prędkościami. Podanie wysokiej prędkości referencyjnej (tj. 50-80 km/h) gwarantuje poprawność wyniku ale też zajmuje więcej czasu. Natomiast coraz niższe prędkości zmniejszają szansę na uzyskanie poprawnej odpowiedzi, ale przyspieszają obliczenia. W szczególności, przy pewnej prędkości granicznej, algorytm przestanie być dopuszczalny. Finalna reprezentacja algorytmu A* liczącego czas podróży, po modyfikacji, została pokazana na Lis. 9.

W celu sprawdzenia powyższej metody, obrałem przykładową, nieoczywistą trasę i wykonałem program dla różnych wartości oczekiwanej prędkości. Ustawienia podane w ramach testu do programu to: *Ołtaszyn*, *Pola*, *t*, *12:00:00*. Pomiaru dla każdego parametru heurystyki dokonałem trzykrotnie i obliczyłem średnią, aby zminimalizować wpływ czynników zewnętrznych na wynik. Wykres czasu wykonywania algorytmu oraz całkowitego kosztu trasy (jakości algorytmu) względem obranego parametru „przewidywanej prędkości" heurystyki przedstawia Rys. 9.

```rust
pub fn astar_time<'bn>(
    bn: &'bn BusNetwork,
    start_name: &str,
    start_time: Time,
    end_name: &str,
    expected_speed_kmph: f32,
) -> Option<Path<'bn, u32>> {
    let cost_fn = |edge: &Edge| edge.time_min();
    let heuristic_fn = |next: &Node, end: &Stop| {
        // v = s/t => t = s/v
        (next.stop.pos.distance_km(end.pos) / (expected_speed_kmph / 60.)) as u32
    };

    astar(bn, start_name, start_time, end_name, cost_fn, heuristic_fn)
}
```

Listing 9: Modyfikacja algorytmu A* minimalizująca czas podróży.



Rysunek 9: Wpływ parametru heurystyki na koszt trasy oraz czas wykonywania algorytmu.

Jak widać, w okolicach wartości 10 km/h znajduje się punkt krytyczny, gdzie po jednej stronie mamy wyniki otrzymane bardzo szybko, ale niekoniecznie poprawne, a po drugiej wyniki pewne, ale otrzymane wolniej. W finalnej implementacji programu pozostawiłem wartość 20 km/h, gdyż wtedy heurystyka najprawdopodobniej jest nadal dopuszczalna, a obliczenia nadal są dość szybkie.

Warto też zauważyć, że nawet najwolniejszy wariant znalazł trasę pomiędzy punktami po dwóch przeciwnych stronach Wrocławia w 15 milisekund. Generalnie algorytm jest na tyle szybki, że nie widzę sensu dalszej optymalizacji. Przy heurystyce bazującej na prędkości 5 km/h możemy wykonać takie przeszukanie ponad tysiąc razy na sekundę na wątek procesora na moim niezbyt drogim laptopie.

3.2 Zadanie 2

Zadanie drugie polegało na rozwiązaniu problemu komiwojażera algorytmem Knoxa (za pomocą przeszukania Tabu). Metoda ta polega na iteracyjnym przeszukiwaniu sąsiedztwa aktualnego rozwiązania, z uwzględnieniem ruchów, które są niedozwolone. Pozwala uniknąć wady przeszukania lokalnego, którą jest utknięcie w lokalnym optimum funkcji celu [17].

Należało utworzyć program, który przyjmuje punkt początkowy oraz listę przystanków, a zwraca najoptymalniejszą (**trwającą najkrócej, mającą najmniej przesiadek**, lub w wariancie uproszczonym najkrótszą) trasę zaczynającą się w punkcie startowym, przechodzącą przez wszystkie przystanki z listy (w dowolnej kolejności) i wracającą do przystanku początkowego [1]. W tym przypadku modyfikujemy jedynie kolejność na liście przystanków (sąsiadem danego rozwiązania jest rozwiązanie z zamienionymi miejscami dwoma elementami listy), natomiast przy koszcie trasy bierzemy również pod uwagę przystanek początkowy/końcowy.

W wersji uproszczonej, można wykorzystać odległość Euklidesowską lokalizacji przystanków od siebie. Odpowiednie dystanse zapisujemy w macierzy i w razie potrzeby (obliczenia kosztu nowego rozwiązania) sumujemy. Jednakże taki model nie rozwiązuje w pełni zadania 2, ponieważ nie uwzględnia aspektów takich jak: zmienna częstotliwość przejazdu autobusów w trakcie dnia oraz konieczność oczekiwania na przystankach. Oczywiście, rozwiązanie to jest też bezużyteczne w zakresie liczenia przesiadek.

Lepszym sposobem na obliczanie kosztu rozwiązania jest wykorzystanie w tym celu algorytmu Dijkstry lub A* zaimplementowanego w zadaniu 1. Wtedy, wykorzystujemy algorytm do obliczenia kolejno kosztu tras $A \to A_2$, $A_2 \to A_3$, ..., $A_n \to A$ i sumujemy poszczególne wartości kosztu. Jest to podejście mniej optymalne czasowo, ale za to zwracające o wiele bardziej realistyczne wyniki. Dodatkową zaletą tej metody jest prostota przy implementacji rozwiązania generycznego – algorytm Tabu liczący liczbę przesiadek różni się od tego liczącego czas podróży jedynie wykorzystaną funkcją kosztu.

Podpunkt (a) zadania wymagał implementacji podstawowej wersji przeszukiwania Tabu. Bazowałem ją głównie na implementacji w Pythonie udostępnionej przez platformę MS Teams. Działanie algorytmu zaczyna się od sprawdzenia poprawności podanych przez użytkownika przystanków oraz ustanowienia stałych takich jak maksymalna liczba iteracji oraz próg wykonanych iteracji bez poprawy wyniku. Następnie rozpoczyna się iteracja, kończąca się po dojściu do górnego limitu liczby iteracji lub przekroczeniu progu iteracji bez poprawy optimum. W każdym kroku pętli przeszukiwane jest sąsiedztwo rozwiązania celem znalezienia najbardziej optymalnego sąsiada. Ustalany jest również „kandydat" Tabu, dodawany na koniec listy w ostatniej fazie pętli. Jeżeli program jest uruchomiony w trybie debugowania, wypisywany jest koszt najlepszego znalezionego do tej pory rozwiązania. Po zakończeniu iteracji, odbudowana i zwracana jest pełna ścieżka uzyskanego rozwiązania.

Podpunkt (b) polegał na dodaniu do poprzedniego rozwiązania ograniczenia na maksymalny rozmiar tablicy T. Całość modyfikacji polegała na zmianie kodu dodającego kandydata do tablicy tak, aby wyglądał jak na Lis. 10. Usuwanie nadmiarowych elementów przed, a nie po dodaniu nowych pozwala uniknąć dodatkowych alokacji. Wprowadzona modyfikacja nie poprawiła znacząco kosztu finalnego rozwiązania, ale pozwoliła na zmniejszenie jego złożoności pamięciowej.

```
if tabu_list.len() >= stops.len() {
    tabu_list.pop_front();
}
tabu_list.push_back(tabu_candidate);
```

Listing 10: Ograniczenie długości tablicy T.

Celem podpunktu (c) było dodanie do algorytmu mechanizmu aspiracji. Wartość aspiracji A określa, że jeśli rozwiązanie jest wystarczająco dobre, to mimo obecności w tablicy T może być ono wybrane. Podobnie jak w przypadku kodu zademonstrowanego przez prowadzącą, ustawiam tę wartość na pewien procent sumy wszystkich odległości między przystankami. Lepszym rozwiązaniem byłoby dynamiczne dostosowanie kryterium, ale uważam, że wybiega to poza zakres polecenia, szczególnie, że ten podpunkt warty jest jedynie 5 pkt. Kluczowy fragment modyfikacji przedstawia Lis. 11.

```
if (!tabu_list.contains(&(i, j)) || neighbour.cost() < aspiration_criteria)
    && neighbour.cost() < best_neighbour.cost()
{
    best_neighbour = neighbour.clone();
    tabu_candidate = (i, j);
}
```

Listing 11: Dodanie mechanizmu aspiracji do przeszukiwania Tabu.

W podpunkcie (d) należało dodać do algorytmu strategię próbkowania sąsiedztwa. Wszystkie zaawansowane strategie jakie przetestowałem znacznie zwiększały czas wykonywania algorytmu, dlatego zdecydowałem się wykorzystać najprostszy wariant – próbkowanie losowe. Metoda ta polega na sprawdzeniu jedynie $x\%$ losowych sąsiadów obecnego rozwiązania. Została ona przedstawiona na Lis. 12.

```
if thread_rng().gen_bool(0.25) {
    continue;
}
```

Listing 12: Najprostsza metoda próbkowania sąsiedztwa.

Zgodnie z instrukcją, metoda próbkowania powinna być deterministyczna. Nie zmienia to i tak algorytmu w deterministyczny, ponieważ pierwszym krokiem nadal pozostaje wymieszanie listy przystanków, więc osobiście nie jestem w stanie znaleźć zalet takiego rozwiązania. Natomiast w razie potrzeby, zamiana losowania $x\%$ sąsiedztwa, na deterministyczny wybór $x\%$ sąsiedztwa jest możliwa za pomocą funkcji mieszającej. W takim wariancie, produkujemy hash indeksu sąsiada i przyrównujemy jego podzielność przez obraną stałą do zera. Metoda ta została przedstawiona na Lis. 13. W finalnej wersji programu pozostawiłem jednak próbkowanie losowe.

```
let mut hasher = DefaultHasher::new();
(i, j).hash(&mut hasher);
let hash = hasher.finish();
if hash % 4 == 0 {
    continue;
}
```

Listing 13: Deterministyczne próbkowanie sąsiedztwa.

Wersja finalna algorytmu wykorzystanego w programie drugim, w formie skróconej została przedstawiona na Lis. 14. Poniżej zaprezentowane zostały wyniki przykładowych wywołań programu, z różnymi długościami tras i kryteriami.

```rust
fn tabu<'bn, C, CF>(
    bn: &'bn BusNetwork,
    start_name: &'bn str, start_time: Time,
    stops: &[&'bn str], cost_fn: CF,
) -> Option<Path<'bn, C>> where C: Cost, CF: Fn(&Edge) -> C {
    let instant = Instant::now();
    if is_invalid(start_name, bn) || stops.iter().any(|s| is_invalid(s, bn)) {
        return None;
    }
    let context = SolutionContext::new(bn, start_name, start_time, cost_fn);
    let max_iterations = stops.len().pow(2);
    let improve_threshold = (max_iterations as f64).sqrt().floor() as usize;
    let aspiration_criteria = context.aspiration_criteria(stops);
    let mut tabu_list = VecDeque::new();
    let mut turns_since_improve = 0;
    let mut current_solution = context.initial_solution(stops);
    let mut best_solution = current_solution.clone();

    for iteration in 0..max_iterations {
        if turns_since_improve > improve_threshold { break; }
        let mut best_neighbour = current_solution.clone();
        let mut tabu_candidate = (0, 0);
        for i in 0..stops.len() {
            for j in i + 1..stops.len() {
                if thread_rng().gen_bool(0.25) { continue; }
                let neighbour = current_solution.mutate(&context, i, j);
                if (!tabu_list.contains(&(i, j))
                    || neighbour.cost() < aspiration_criteria)
                    && neighbour.cost() < best_neighbour.cost()
                {
                    best_neighbour = neighbour.clone();
                    tabu_candidate = (i, j);
                }
            }
        }
        current_solution = best_neighbour.clone();
        if tabu_list.len() >= stops.len() { tabu_list.pop_front(); }
        tabu_list.push_back(tabu_candidate);
        if best_neighbour.cost() < best_solution.cost() {
            best_solution = best_neighbour.clone();
            turns_since_improve = 0;
        } else { turns_since_improve += 1; }
    }

    Some(Path {
        edges: context.reconstruct_edges(&best_solution),
        cost: best_solution.cost(), runtime: instant.elapsed(),
    })
}
```

Listing 14: Wersja finalna programu rozwiązującego problem komiwojażera.

```
Podaj przystanek początkowy A: Dubois
Podaj przystanki do odwiedzenia (oddzielone średnikiem): Rynek;GALERIA DOMINIKAŃSKA;Hala Targowa
Podaj kryterium optymalizacyjne [t/p/d]: t
Podaj czas początkowy (np. "00:00:00"): 08:00:00
Iteration #000 cost: 30
Iteration #001 cost: 23
Iteration #002 cost: 23
Iteration #003 cost: 23
Iteration #004 cost: 23
Iteration #005 cost: 23
       --- 08:00 --> 08:01 Dubois
       --- 08:00 --> 08:01 Dubois
128 08:01 Dubois
       128 08:01 Dubois                    --> 08:03 pl. Bema
128 08:03 pl. Bema
       --- 08:03 --> 08:04 pl. Bema
       --- 08:04 --> 08:04 pl. Bema
 11 08:04 pl. Bema
        11 08:04 pl. Bema                  --> 08:06 Hala Targowa
 11 08:06 Hala Targowa
 11 08:06 Hala Targowa
        11 08:06 Hala Targowa              --> 08:07 pl. Nowy Targ
        11 08:07 pl. Nowy Targ             --> 08:09 GALERIA DOMINIKAŃSKA
 11 08:09 GALERIA DOMINIKAŃSKA
       --- 08:09 --> 08:09 GALERIA DOMINIKAŃSKA
                  ...
       --- 08:11 --> 08:12 GALERIA DOMINIKAŃSKA
 10 08:12 GALERIA DOMINIKAŃSKA
        10 08:12 GALERIA DOMINIKAŃSKA      --> 08:14 Świdnicka
        10 08:14 Świdnicka                 --> 08:15 Zamkowa
        10 08:15 Zamkowa                   --> 08:17 Rynek
 10 08:17 Rynek
       --- 08:17 --> 08:17 Rynek
                  ...
       --- 08:19 --> 08:19 Rynek
132 08:19 Rynek
       132 08:19 Rynek                     --> 08:23 Dubois
132 08:23 Dubois

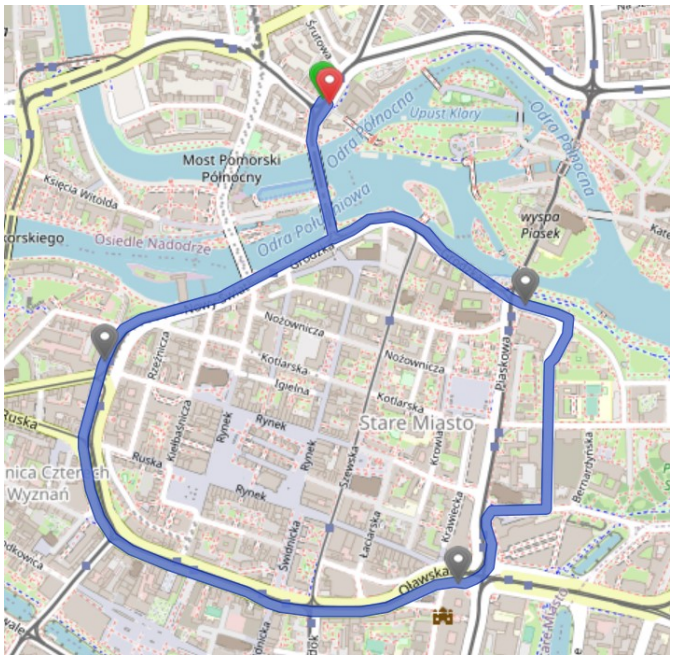
Distance: 5.155 km | Time: 23 min | Buses: 5
Cost: 23 | Runtime: 110 ms
```

Rysunek 10: Trasa dookoła Rynku z minimalnym czasem podróży w trybie debugowania.

```
Podaj przystanek początkowy A: Dubois
Podaj przystanki do odwiedzenia (oddzielone średnikiem): Rynek;GALERIA DOMINIKAŃSKA;Hala Targowa
Podaj kryterium optymalizacyjne [t/p/d]: p
Podaj czas początkowy (np. "00:00:00"): 08:00:00
  7 06:08 Dubois
  7 06:13 Rynek
 10 15:04 Rynek
 10 15:10 GALERIA DOMINIKAŃSKA
 23 22:56 GALERIA DOMINIKAŃSKA
 23 22:58 Hala Targowa
242 23:54 Hala Targowa
242 23:56 Dubois

Cost: 4 | Runtime: 1773 ms
```

Rysunek 11: Ta sama trasa z minimalną ilością przesiadek.

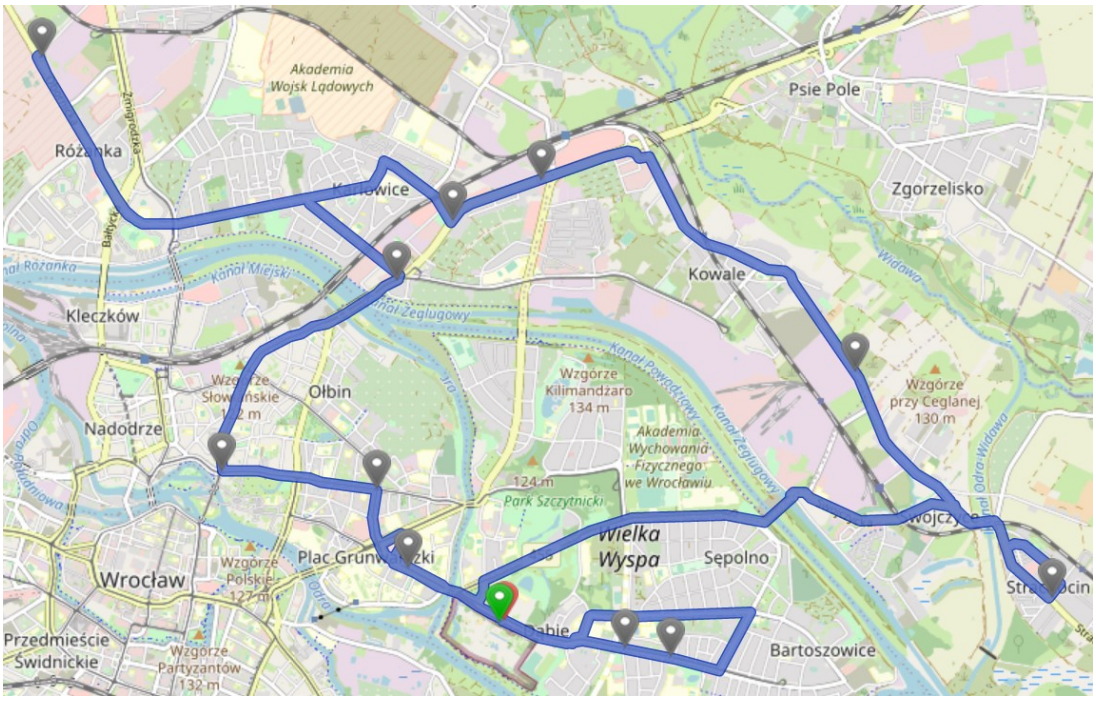


Rysunek 12: Mapa powyższej trasy.

```
Podaj przystanek początkowy A: ZOO
Podaj przystanki do odwiedzenia (oddzielone średnikiem): Kowalska;Chełmońskiego;pl. Bema;Kromera (Czajkowskiego)
Podaj kryterium optymalizacyjne [t/p/d]: t
Podaj czas początkowy (np. "00:00:00"): 23:00:00
  4 23:04 ZOO
  4 23:06 Chełmońskiego
  4 23:06 Chełmońskiego
  4 23:07 Piramowicza
  4 23:08 Piramowicza
  4 23:19 Piastowska
  9 23:23 Piastowska
  9 23:27 pl. Bema
 11 23:29 pl. Bema
 11 23:36 KROMERA
128 23:38 KROMERA
128 23:49 Zajezdnia Obornicka
130 04:00 Zajezdnia Obornicka
130 04:19 Kromera (Czajkowskiego)
130 04:19 Kromera (Czajkowskiego)
130 04:21 Brücknera
118 04:37 Brücknera
118 04:44 Kowalska
118 04:44 Kowalska
118 04:51 Zagrodnicza
115 04:52 Zagrodnicza
115 05:05 Kliniki - Politechnika Wrocławska
  2 05:07 Kliniki - Politechnika Wrocławska
  2 05:10 ZOO

Cost: 368 | Runtime: 594 ms
```

Rysunek 13: Dłuższa trasa zoptymalizowana pod względem czasu.



Rysunek 14: Mapa powyższej trasy.

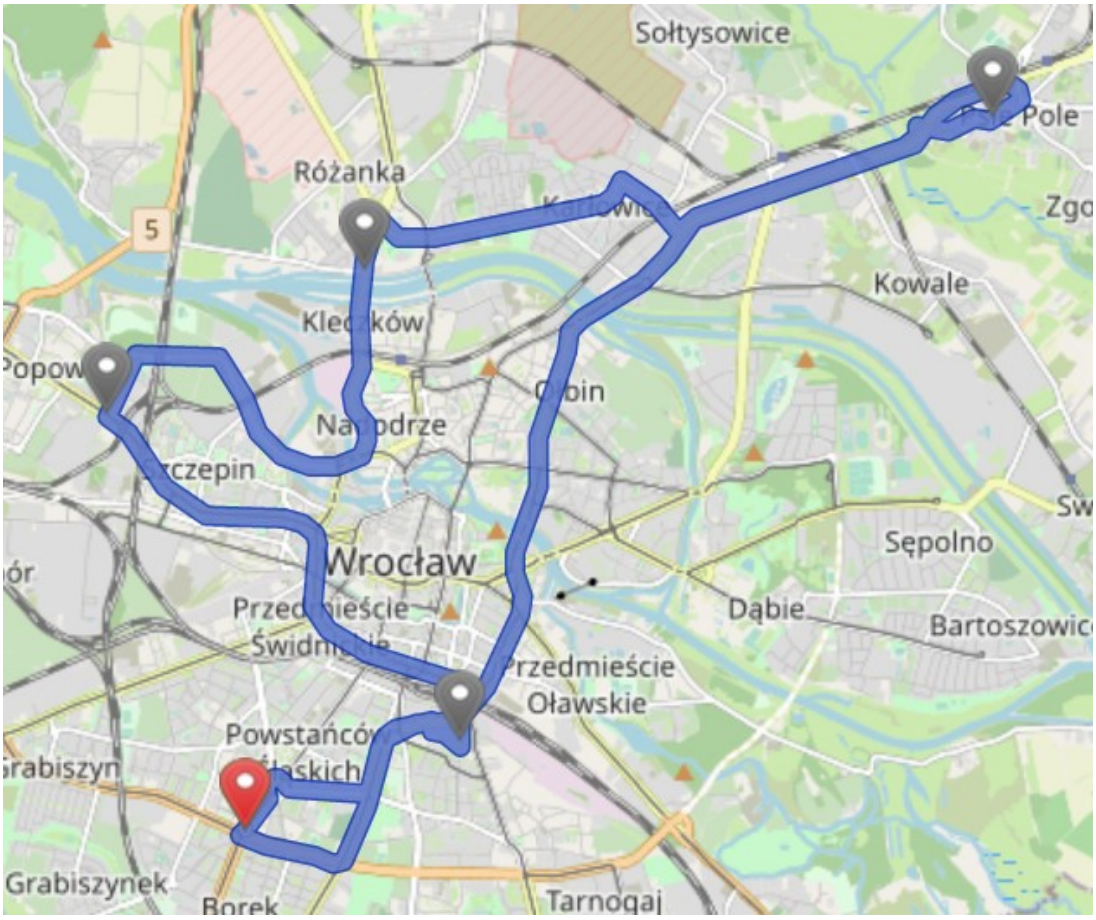
Warto zwrócić uwagę na fragment na północnym-zachodzie na Rys. 14, gdzie pasażer trafia na przystanek *Zajezdnia Obornicka*. Odwiedzenie tego przystanku nie było wymagane przez użytkownika i zdaje się być zbędnym objazdem. Jednakże warto zauważyć, że zgodnie z Rys. 13, pasażer trafia na zajezdnię o godzinie 23:49, a wyjeżdża z niej o 04:00.

Przy dość późnym (23:00) terminie początkowym, na trasie musiało nastąpić oczekiwanie do kolejnego dnia. Algorytm postanowił, że lepiej przed tym okresem dostać się do zajezdni, skąd rano najwcześniej zaczynają kursować kolejne autobusy.

```
Podaj przystanek początkowy A: Hallera
Podaj przystanki do odwiedzenia (oddzielone średnikiem): Niedźwiedzia;Bałtycka;Psie Pole;Gajowa
Podaj kryterium optymalizacyjne [t/p/d]: p
Podaj czas początkowy (np. "00:00:00"): 05:00:00
   D 08:53 Hallera
   D 09:28 Psie Pole
111 17:27 Psie Pole
111 18:06 Bałtycka
102 06:03 Bałtycka
102 06:22 Niedźwiedzia
 32 14:21 Niedźwiedzia
 32 14:43 Gajowa
 31 23:43 Gajowa
 31 23:56 Hallera

Cost: 5 | Runtime: 7070 ms
```

Rysunek 15: Trasa dookoła Wrocławia optymalizowana pod względem ilości przesiadek.



Rysunek 16: Mapa powyższej trasy.

W przypadku trasy z Rys. 15, algorytmowi udało się znaleźć połączenia bezpośrednie między wszystkimi punktami podróży. Czas trwania podróży jest natomiast bardzo nieoptymalny, ponieważ nie stanowił on żadnej wagi w kryterium kosztu.

4 Podsumowanie

Zadania z listy na pierwszy rzut oka wydawały się być bardzo łatwe, ale bynajmniej takie nie były. Natomiast główna trudność nie polegała na samej implementacji algorytmów, a bardziej na dostosowaniu ich do otrzymanego wycinka świata rzeczywistego. Dane były niewygodnie ustrukturyzowane, miejscami zawierały błędy, co spowolniło proces implementacji.

Głównym problemem było zaadaptowanie algorytmów (lub reprezentacji grafu) tak, aby brać pod uwagę ustalone czasy odjazdu z przystanków, które sprawiały, że dane formowały multigraf. Zignorowanie niektórych sytuacji brzegowych (takich jak problemy algorytmu Dijkstry przy pracy z multigrafami, położenie przystanków o tej samej nazwie w różnych lokalizacjach, błąd powiązany z odejmowaniem współrzędnych geograficznych od siebie) prowadziło do prostszej, ale mniej poprawnej implementacji, dlatego zdecydowałem się wziąć pod uwagę wszystkie takie przypadki.

Nie natrafiłem na znaczne problemy implementacyjne, większość algorytmów działała za pierwszym razem, a pozostałe wymagały jedynie małych poprawek. Wybór języka okazał się być dość dobry, ponieważ napisane programy wykonują się kilkaset, a momentami kilka tysięcy razy szybciej niż w Pythonie.

Istotny jest odpowiedni wybór funkcji kosztu, algorytmy wyszukiwania ścieżki bezwzględnie minimalizują wartość kosztu ignorując pozostałe właściwości ścieżki. Przykładowo, optymalizując wyłącznie pod kątem przesiadek możemy uzyskać trwające komicznie długo trasy (ale posiadające skrajnie mało przesiadek), co raczej nigdy nie jest zachowaniem pożądanym przez użytkownika. Istotny jest też wybór odpowiedniej heurystyki dla algorytmu A*. Przy nieodpowiedniej heurystyce wynik algorytmu może być daleki od optymalnego.

Wszystkie struktury pomocnicze pokryłem testami jednostkowymi, co pozwoliło wykryć kilka błędów we wczesnych fazach implementacji. Wyniki testów zostały przedstawione na Rys. 17.

```
running 18 tests
test graph::state::tests::state_has_reversed_order ... ok
test structs::pos::tests::bases_are_perpendicular_to_normal ... ok
test structs::pos::tests::bases_are_orthogonal ... ok
test structs::pos::tests::bases_are_unit ... ok
test graph::state::tests::state_turns_max_heap_into_min_heap ... ok
test structs::pos::tests::distance_to_self_is_zero ... ok
test structs::pos::tests::cartesian_coords_are_placed_on_sphere ... ok
test structs::pos::tests::distance_satisfies_the_triangle_inequality ... ok
test structs::pos::tests::market_square_is_placed_at_origin ... ok
test structs::time::tests::invalid_time_cant_be_constructed - should panic ... ok
test structs::time::tests::time_is_parsed_and_formatted_correctly ... ok
test structs::time::tests::time_is_subtracted_correctly ... ok
test util::string_pool::tests::returns_different_buffers_for_different_strings ... ok
test util::string_pool::tests::returns_same_buffers_for_same_strings ... ok
test util::vec3::tests::cross_product_returns_perpendicular_vector ... ok
test util::vec3::tests::cross_product_uses_the_valid_formula ... ok
test util::vec3::tests::len_and_len_2_return_correct_result ... ok
test util::vec3::tests::normalized_vector_is_a_unit_vector ... ok
```

Rysunek 17: Testy jednostkowe

Do wariantów optymalizujących czas podróży oraz ilość przesiadek z zadania pierwszego i drugiego dopisałem optymalizację dystansu, dostępną z poziomu interfejsu użytkownika jako kryterium „d” (i „dd”). Taki wariant jest najłatwiej manualnie przetestować, a różni się od dwóch pozostałych kryteriów jedynie funkcją kosztu, więc może służyć jako test działania logiki algorytmów.

Literatura

[1] Instrukcja do laboratorium 1 – M. Karol, P. Syga.

[2] Wykład 1: Rozwiązywanie problemów przez przeszukiwanie – M. Piasecki.

[3] Rust – Wikipedia [dostęp: 18.03.2023],
`https://pl.wikipedia.org/wiki/Rust_(j%C4%99zyk_programowania)`

[4] Generalization of Dijkstra's Algorithm for Extraction of Shortest Paths in Directed Multigraphs – S.S. Bitwas et al. [dostęp: 16.03.2023],
`https://thescipub.com/pdf/jcssp.2013.377.382.pdf`

[5] Time-Dependent Graphs: Definitions, Applications, and Algorithms – Y. Wang et al. – SpringerLink [dostęp: 12.03.2023],
`https://link.springer.com/article/10.1007/s41019-019-00105-0`

[6] Least turns in a maze – Stack Overflow [dostęp: 18.03.2023],
`https://stackoverflow.com/a/28160195`

[7] Theseus and the Minotaur — Exploring State Space – Tomáš Sláma [dostep: 14.03.2023],
`https://www.youtube.com/watch?v=umszOeerdsU`

[8] Internowanie łańcuchów – Wikipedia [dostęp: 24.03.2023],
`https://pl.wikipedia.org/wiki/Internowanie_%C5%82a%C5%84cuch%C3%B3w`

[9] Earth Radius by Latitude Calculator – Rechneronline [dostęp: 12.03.2023],
`https://rechneronline.de/earth-radius/`

[10] Finding Orthogonal Vectors in a Plane – Mathematics Stack Exchange [dostęp: 12.03.2023],
`https://math.stackexchange.com/a/2450750`

[11] How to Project a 3D Point onto a 2D Plane? – Baeldung [dostęp: 12.03.2023],
`https://www.baeldung.com/cs/3d-point-2d-plane`

[12] Algorytm Dijkstry – Wikipedia [dostęp: 16.03.2023],
`https://pl.wikipedia.org/wiki/Algorytm_Dijkstry`

[13] `std::collections::binary_heap` – Rust [dostęp: 12.03.2023],
`https://doc.rust-lang.org/std/collections/binary_heap/index.html`

[14] Why does Rust not implement total ordering via the `Ord` trait for `f64` and `f32`? – Stack Overflow [dostęp: 18.03.2023],
`https://stackoverflow.com/a/26490185`

[15] Algorytm A* – Wikipedia [dostęp: 19.03.2023],
`https://pl.wikipedia.org/wiki/Algorytm_A*`

[16] Zarządzenie nr 8 generalnego dyrektora dróg krajowych i autostrad z dnia 7 lutego 2013 r. w sprawie zasad ustanawiania prędkości dopuszczalnych [dostęp: 20.03.2023]
`https://www.archiwum.gddkia.gov.pl/userfiles/articles/...`

[17] Tabu search performance on the symmetric traveling salesman problem – J. Knox – Sci-Hub [dostęp: 21.03.2023],
`https://sci-hub.se/10.1016/0305-0548(94)90016-7`

Sztuczna inteligencja i inżynieria wiedzy
K01-49i, czwartek 11:15

prowadzący
mgr inż. Katarzyna Fojcik



Politechnika
Wrocławska

Algorytmy grające w gry logiczne
Lista 2

Tomasz Chojnacki
260365@student.pwr.edu.pl

WROCŁAW 11 maja 2023

Spis treści

1 Wstęp **2**
1.1 Tematyka sprawozdania . . . 2
1.2 Gra planszowa Reversi . . . 2
1.2.1 Warianty zasad gry . . . 2
1.2.2 Othello . . . 3
1.2.3 Przyjęte założenia . . . 3
1.3 Wybór technologii . . . 4
1.3.1 Wykorzystane biblioteki . . . 5

2 Implementacja logiki gry **5**
2.1 Reprezentacja stanu gry w pamięci [Z1] . . . 5
2.2 Funkcja generująca możliwe ruchy [Z1] . . . 6
2.3 Obliczenie ogólnych statystyk gry . . . 9
2.4 Walidacja stanu wejściowego . . . 11
2.5 Implementacja abstrakcyjnej strategii . . . 13
2.6 Testy jednostkowe i integracyjne . . . 13
2.7 Opytmalizacja z użyciem bitboardów [Z1] . . . 15

3 Metoda Minimax **19**
3.1 Implementacja algorytmu [Z3] . . . 19
3.2 Cięcie alfa–beta [Z4] . . . 22
3.3 Dobór heurystyk [Z2] . . . 23
3.3.1 Przewaga pionów (`MaximumDisc`, `MinimumDisc`) . . . 23
3.3.2 Ważone pola (`Weighted`) . . . 23
3.3.3 Posiadane narożniki (`CornersOwned`) . . . 23
3.3.4 Bliskość narożników (`CornerCloseness`) . . . 23
3.3.5 Obecna mobilność (`CurrentMobility`) . . . 23
3.3.6 Potencjalna mobilność (`PotentialMobility`) . . . 24
3.3.7 Stabilność (`Stability`, `InternalStability`, `EdgeStability`) . . . 24
3.3.8 Kombinacja Kormana (`Korman`) . . . 24
3.4 Ocena skuteczności heurystyk . . . 24
3.5 Implementacja interfejsu tekstowego . . . 27

4 Zadanie dodatkowe **28**
4.1 Modyfikacja programu [Z5] . . . 28
4.2 Heurystyki adaptacyjne [Z5] . . . 29

5 Podsumowanie **32**
5.1 Dalsze badania . . . 32
5.2 Wnioski . . . 33

1 Wstęp

1.1 Tematyka sprawozdania

Poniższe sprawozdanie stanowi raport z realizacji drugiej listy zadań z kursu *Sztuczna inteligencja i inżynieria wiedzy*. Tematyką tej listy były algorytmy grające w gry dwuosobowe (w szczególności algorytm Minimax oraz jego optymalizacja z zastosowaniem cięć alfa-beta), a głównym celem konstrukcja programu grającego w grę planszową Reversi [1].

Sprawozdanie zostało ustrukturyzowane według głównych wykonanych etapów projektowych, a nie zgodnie z kolejnością zadań przedstawioną na liście, ponieważ bywała ona miejscami nielogiczna (np. zadania wymagają doboru listy heurystyk przed implementacją algorytmu Minimax, przez co nie można w żaden sposób testować tworzonych funkcji). Aby ułatwić sprawdzanie realizacji zadań, sekcje, w których były one realizowane zostały oznaczone na spisie treści oraz w nagłówkach czerwonym symbolem (np. [Z1]).

Repozytorium zawierające pełny kod źródłowy rozwiązania zostało umieszczone w bibliografii, jako pozycja [2]. Niemniej, sprawozdanie zawiera bloki kodu opisujące najistotniejsze części rozwiązania. Na końcu sprawozdania został umieszczony słownik pojęć.

1.2 Gra planszowa Reversi

Reversi jest grą planszową dla dwóch graczy, rozgrywaną na planszy o wymiarach 8×8. Gracze na przemian (o ile to możliwe) stawiają na pustych polach piony swojego koloru, tak, aby utworzyć linię składającą się z pionów przeciwnika flankowanych (otoczonych) z obu stron pionami obecnego gracza. Utworzenie takiej linii odwraca wszystkie piony gracza przeciwnego zamieniając je w piony obecnego gracza. Gracz posiadający większą liczbę pionów na planszy wygrywa [1]. Gra została wynaleziona w XIX wieku, ale zyskała drugie życie w latach 80. za sprawą opatentowania komercyjnego wariantu gry zwanego **Othello** [4].

1.2.1 Warianty zasad gry

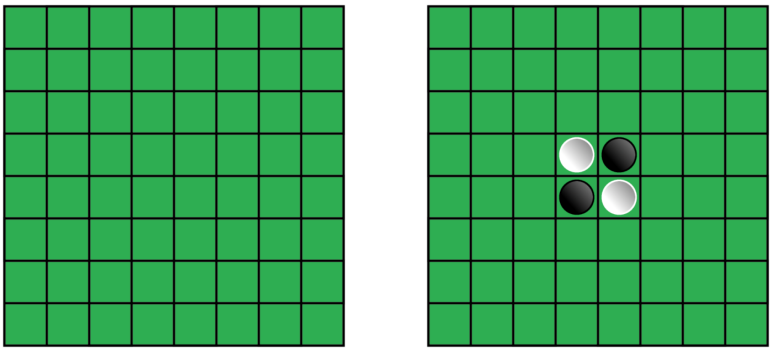
Niestety, z uwagi na długowieczność gry, z czasem wykształciła się znaczna liczba wariantów gry, pomiędzy którymi występują sprzeczności w kwestii niektórych zasad gry. Również w instrukcji do zadania pojawiły się nieścisłości – niektóre z przedstawionych na niej reguł nie obowiązują w żadnym spośród znalezionych przeze mnie wariantów gry (nawet tych podanych w bibliografii listy zadań). W związku z tym istotna jest identyfikacja dostępnych możliwości i wybór jednolitych zasad wykorzystanych przy późniejszej implementacji programu. Główne spory dotyczące reguł Reversi zostały przedstawione poniżej:

- **Początkowy stan planszy**: W historycznej wersji gry plansza początkowa była pusta i w pierwszych czterech turach gry gracze musieli stawiać naprzemiennie piony na czterech centralnych polach planszy, prowadząc do powstania jednego z dwóch wzorów: równoległych linii pionów lub pionów ułożonych na krzyż (w tych turach poluzowany jest wymóg przejęcia piona w każdym ruchu, ponieważ takie przejęcie jest niemożliwe). W większości współczesnych zastosowań gra rozpoczyna się już z ułożonym na planszy wzorem krzyża. Zgodnie z konwencją piony czarne (gracza pierwszego) znajdują się na północnym wschodzie i południowym zachodzie planszy [4]. Zastosowanie początkowego układu planszy wynika z tego, że wzór równoległych linii pionów prowadzi do mniej interesującej rozgrywki [5][7]. Lista zadań wskazuje, że gra zaczyna się z pustą planszą, ale nie wskazuje zasady dotyczącej stawiania pierwszych czterech ruchów na polach centralnych. Opisane powyżej różnice zostały przedstawione na Rys. 1.

- **Wykonywanie ruchów**: Najpopularniejszym obecnie wariantem jest taki, w którym każdy ruch gracza musi flankować przeciwnika, a jeżeli nie jest to możliwe, tura jest przekazywana przeciwnikowi bez żadnych dodatkowych konsekwencji. Dwa alternatywne rozwią-

zania to: zakończenie gry porażką obecnego gracza w przypadku braku dostępnych ruchów lub zezwolenie na stawianie pionów, które nie flankują przeciwnika. W przypadku braku legalnych ruchów dla obu graczy, gra kończy się zwycięstwem gracza posiadającego więcej pionów na planszy (we wszystkich powyższych wariantach) [7].

- **Rozstrzyganie remisów**: W przypadku remisu (gry zakończonej z identyczną liczbą pionów czarnych i białych na planszy – zazwyczaj, ale nie zawsze, wynikiem 32–32) stosowane są różnorakie metody rozstrzygnięcia zwycięzcy. Najpopularniejsze z nich to: wybór gracza z mniejszym wykorzystanym czasem, wygrana gracza z wyższym współczynnikiem Brightwella [4], powtórzenie meczu lub przyznanie połowicznych punktów obu graczom.
- **Liczenie wyniku**: W większości wariantów gry finalnym wynikiem jest liczba pionów gracza na planszy, natomiast na niektórych zawodach pola puste doliczane są do wyniku gracza, który wygrał grę (wtedy, przykładowo, zamiast 13–0 wynikiem meczu jest 64–0) [8].



Rysunek 1: Porównanie plansz początkowych zgodnych z historycznymi (po lewej) oraz współczesnymi (po prawej) zasadami Reversi [6].

1.2.2 Othello

Othello jest ustandaryzowaną i opatentowaną wersją Reversi, wykorzystywaną na wszystkich turniejach, w tym, w szczególności na mistrzostwach świata – World Othello Championship. W tym wariancie gry, plansza zawiera początkowo wzór krzyża (prawa strona Rys. 1). Grę rozpoczyna gracz korzystający z czarnych pionów i ruchy wykonywane są naprzemiennie w turach ponumerowanych od 1 do 60 włącznie. Każdy ruch gracza musi flankować przeciwnika, jeżeli nie jest to możliwe, ruch jest oddany graczowi przeciwnemu. Jeżeli żaden z graczy nie jest w stanie wykonać ruchu, gra kończy się. Remisy są rozstrzygane zgodnie ze współczynnikiem Brightwella, a puste pola są liczone w wyniku końcowym na konto zwycięskiego gracza [8].

1.2.3 Przyjęte założenia

Głównymi celami przy wyborze spójnego zestawu reguł do dalszej implementacji były: najwyższa możliwie zgodność z zasadami Othello (co pozwala wykorzystywać do testowania i trenowania algorytmów oficjalnych meczów, których archiwa są dostępne w internecie), najwyższa możliwie zgodność z poleceniem oraz prostota rozwiązania.

W związku z tym, jako ostateczny zestaw reguł wybrałem **ustandaryzowane reguły Othello z następującymi różnicami**:

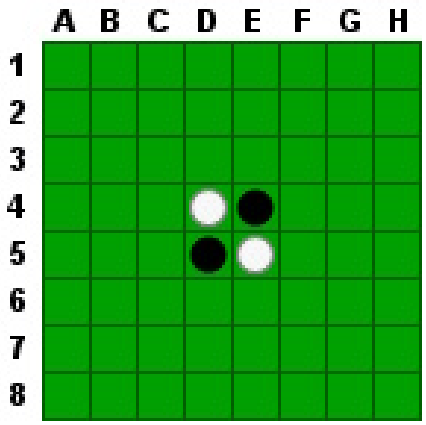
- Finalny wynik rozgrywki jest zawsze **równy ilości pionów czarnych i białych na planszy**. Puste pola nie są doliczane do wyniku żadnego z graczy i istnieją mecze zakończone

wynikiem, który nie sumuje się do 64 (np. 13–0). Wynik jest zawsze przedstawiony w formacie `BLACK-WHITE`, niezależnie od tego, który z graczy posiada więcej pionów na planszy. **Remisy nie są rozstrzygane** – nie liczą się jako wygrana, ani przegrana dla żadnego spośród uczestników. Niezależnie od stanu gry, osiągnięcie remisu jest zawsze bardziej opłacalne od przegranej oraz mniej opłacalne od zwycięstwa.

- Wszystkie algorytmy powinny, o ile jest to możliwe i niesprzeczne z pozostałymi regułami, **działać również dla pozycji nieosiągalnych w Othello**, ale osiągalnych w historycznej wersji Reversi (tj. dla tur od -3 do 0 włącznie). Czyli, gdy na planszy znajdują się mniej niż 4 piony, dozwolone jest tylko stawianie pionów na polach centralnych.

Gwarantuje to 100% zgodności programu z zapisami oficjalnych meczów, ale także pozwala na analizę działania programu w historycznym (i bardziej zgodnym z poleceniem) wariancie gry.

Ponadto, aby ułatwić wyróżnianie konkretnych pól na planszy wprowadzona jest notacja `A`–`H` dla kolumn oraz `1`–`8` dla rzędów planszy. Do konkretnego pola na planszy można się zatem odwołać za pomocą kombinacji liter i cyfr od `A1` do `H8`. Warto zauważyć, że rzędy są oznaczone odwrotnie niż w szachach. Wizualizacja tej notacji znajduje się na Rys. 2.



Rysunek 2: Notacje pól na planszy [9].

1.3 Wybór technologii

Po wstępnym rozeznaniu tematu, doszedłem do wniosku, że istotne w dalszej implementacji jest, aby wykorzystany język programowania miał dobre wsparcie dla rekurencji, posiadał rozbudowany system testowania jednostkowego i pozwalał na szczegółowe dostosowanie reprezentacji przetwarzanych danych.

Następnie, zbadałem w jakich językach zostały napisane poprzednie popularne i osiągające wysokie wyniki programy grające w Reversi. Spośród programów wymienianych w publikacjach jako istotne czy też przełomowe znajdują się programy napisane w następujących językach: BASIC, Fortran, Assembler, C, C++, Rust [10][11]. Łatwo zauważyć, że każde z powyższych to języki dość niskopoziomowe, pozwalające na szczegółową optymalizację i operacje bitowe. Częściowo wynika to na pewno z tego, że największe postępy w badaniach nad Reversi zostały poczynione w latach 80. i 90. Natomiast analizując repozytoria na GitHubie rozwiązujące ten problem, aby uzyskać bardziej współczesny rozkład, można zauważyć, że najpopularniejsze są te napisane w C++, C oraz Javie (pomimo, że obecnie języki takie jak Python czy JavaScript są znacznie popularniejsze w skali całej platformy), co potwierdza wcześniejsze wnioski.

W związku z tym ponownie zdecydowałem się na implementację programu w języku Rust, ponieważ spośród języków, w których czuję się komfortowo pozwala on na programowanie na najniższym poziomie.

1.3.1 Wykorzystane biblioteki

Starałem się wykorzystać jak najmniejszą liczbę bibliotek, zamiast tego implementując całą funkcjonalność ręcznie. Jednakże wykorzystywałem pakiety zewnętrzne, gdy uznawałem, że problem, który rozwiązują nie ma zbyt wielkiej wartości dydaktycznej w kontekście obecnej listy zadań. Były to:

- `clap` – parser argumentów wiersza poleceń [12]
- `colored` – ułatwia kolorowanie tekstu wypisywanego w konsoli [13]
- `once_cell` – ułatwia bezpieczną deklarację ewaluowanych leniwie danych globalnych [14]
- `rand` – służy do generacji liczb losowych [15]

2 Implementacja logiki gry

> *„1. poprawne zdefiniowanie stanu gry i funkcji generującej możliwe ruchy dla danego stanu i gracza”* [1]

2.1 Reprezentacja stanu gry w pamięci [Z1]

Oczywistym początkowym podejściem jest reprezentacja planszy jako tablica dwuwymiarowa 8×8 reprezentująca poszczególne pola. Zamiast tego, można przechowywać pola w tablicy jednowymiarowej o długości 64, co pozwala uniknąć jednego dodatkowego skoku przez wskaźnik oraz gwarantuje liniowość pamięci. Należy wtedy dobrać odpowiednią funkcję tłumaczącą notację na indeks w tablicy. Zdecydowałem się na przyporządkowanie indeksów zaczynające od 0 w polu `A1` i rosnące do 63 w polu `H8`. Przejście o jedno pole w prawo zwiększa indeks o 1, natomiast przejście o jedno pole w dół zwiększa o 8. Pozycje na planszy reprezentuje struktura `Position`, zajmująca dokładnie jeden bajt pamięci i przedstawiona na Lis. 1. W celu zwiększenia czytelności pominięto nieciekawą implementację parsowania pozycji z ciągu tekstowego.

```rust
#[must_use]
#[derive(Clone, Copy, Debug, PartialEq, Eq, Hash)]
pub struct Position(u8);

impl Position {
    #[must_use]
    pub fn from(notation: &str) -> Option<Self> { /* ... */ }

    #[must_use]
    pub const fn index(&self) -> usize { self.0 as usize }
}
```

Listing 1: Struktura opisująca pozycję na planszy.

Istotne jest, że publiczny interfejs struktury nie zezwala na konstrukcję pozycji reprezentujących niepoprawne pola (np. wartość 70 mieści się w zakresie bajtu, ale nie może być przechowywana w strukturze `Position`).

Danymi przechowywanymi w ww. tablicy są natomiast struktury reprezentujące pola planszy – `Square` zaimplementowane jako rekordy z wariantami [16] i korzystające z kolejnej struktury pomocniczej o nazwie `Player`. Cały system został przedstawiony na Lis. 2.

```rust
#[must_use]
#[derive(Clone, Copy, Debug, PartialEq, Eq, Hash)]
pub enum Square {
    Empty,
    Placed(Player),
}

#[must_use]
#[repr(u8)]
#[derive(Clone, Copy, Debug, PartialEq, Eq, Hash)]
pub enum Player {
    Black = 1,
    White = 2,
}
```

Listing 2: Rekordy z wariantami `Square` oraz `Player`.

Dzięki metodyce *Zero Cost Abstractions* (znanej w kontekście C++ pod nazwą *zero–overhead principle*) [17], kompilator jest w stanie zagwarantować, że obie te struktury będą miały rozmiar dokładnie jednego bajtu. Co więcej ich reprezentacja w pamięci będzie równa dokładnie 0, 1 i 2 dla odpowiednio `Empty`, `Placed(Black)` i `Placed(White)`.

Takie rozwiązanie pozwala zapobiec niektórym błędom, które łatwo popełnić przy reprezentacji za pomocą zwykłych liczb całkowitych. Przykładowo, niemożliwe jest wykonywanie działań arytmetycznych na tych strukturach albo przechowanie w pamięci błędnej wartości. Natomiast podział na dwa typy pozwala wykorzystać strukturę `Player` również w innych kontekstach (np. do reprezentacji zwycięzcy rozgrywki).

Początkowa wersja pełnej reprezentacji stanu gry znajduje się na Lis. 3. Oprócz planszy należy przechować, który gracz wykonuje obecną turę, co zostanie uzasadnione w późniejszych rozdziałach.

```rust
pub const BOARD_SIDE: usize = 8;
pub const BOARD_SQUARES: usize = BOARD_SIDE.pow(2);

#[derive(Clone)]
pub struct GameState {
    turn: Player,
    board: [Square; BOARD_SQUARES],
}
```

Listing 3: Początkowa wersja reprezentacji stanu gry.

2.2 Funkcja generująca możliwe ruchy [Z1]

Następnie opracowana została funkcja generująca możliwe ruchy oraz funkcja wykonująca ruch i odwracająca piony. Najprostszym podejściem jest zdefiniowanie funkcji sprawdzającej czy podany ruch jest poprawny i wywołanie jej 64 razy w pętli, ale jest to rozwiązanie mniej optymalne niż generacja wszystkich możliwych ruchów jednocześnie. Implementacja algorytmu znajduje się na Lis. 4, a poniżej zawarte jest objaśnienie wykorzystanej metodyki.

```rust
pub fn moves(&self) -> impl Iterator<Item = Position> + '_ {
    let mut result = HashSet::new();

    if self.occupied_squares().count() < 4 { // Reversi earlygame
        result.extend(
            Position::CENTER_SQUARES
                .into_iter()
                .filter(|&p| self.at(p) == Square::Empty));
        return result.into_iter();
    }

    for position in self.discs_of(self.turn) {
        for dir in DIRECTIONS {
            if let Some(mut coord) = position.offset(dir) {
                while self.at(coord) == Square::Placed(self.turn.opponent()) {
                    if let Some(next) = coord.offset(dir) {
                        coord = next;
                        if self.at(coord) == Square::Empty {
                            result.insert(coord);
                        }
                    } else { break; }
                }
            }
        }
    }

    result.into_iter()
}
```

Listing 4: Początkowa wersja funkcji generującej możliwe ruchy.

Powyższy kod można podsumować następująco:

1. Jeżeli mniej niż cztery pola są zajęte, zwróć iterator po pustych polach centralnych.
2. Z każdego pola, na którym znajduje się pion obecnego gracza pójdź w każdym z ośmiu kierunków, poprzez co najmniej jeden pion przeciwnika aż do natrafienia na koniec planszy lub na puste pole. Jeżeli istnieje takie pole, dodaj je do zbioru wynikowego.
3. Zwróć iterator po zbiorze wynikowym.

Czas wykonywania takiego rozwiązania jest proporcjonalny do liczby zajętych przez obecnego gracza pól, więc w wielkim uproszczeniu kod wykonuje się wolniej w późniejszych fazach gry. Niestety kod wymaga zastosowania wielu pętli i instrukcji warunkowych, a sprawdzanie warunku flankowania nie jest łatwe.

Logika wykonywania ruchu jest łatwiejsza do zrozumienia: z danej pozycji idziemy w każdym kierunku, zbierając kolejkę napotkanych po drodze pozycji z pionami przeciwnika. Jeżeli na końcu iteracji w danym kierunku napotkamy pion obecnego gracza obracamy wszystkie piony z kolejki, jeżeli nie, to czyścimy kolejkę. Na końcu, jeżeli obecny gracz nie może wykonać ruchu, przekazujemy turę przeciwnikowi (`next_state.pass_if_required()`). Sprawdzamy też, czy wykonywany ruch jest legalny, aby zachować poprawność danych. Implementacja została przedstawiona na Lis. 5.

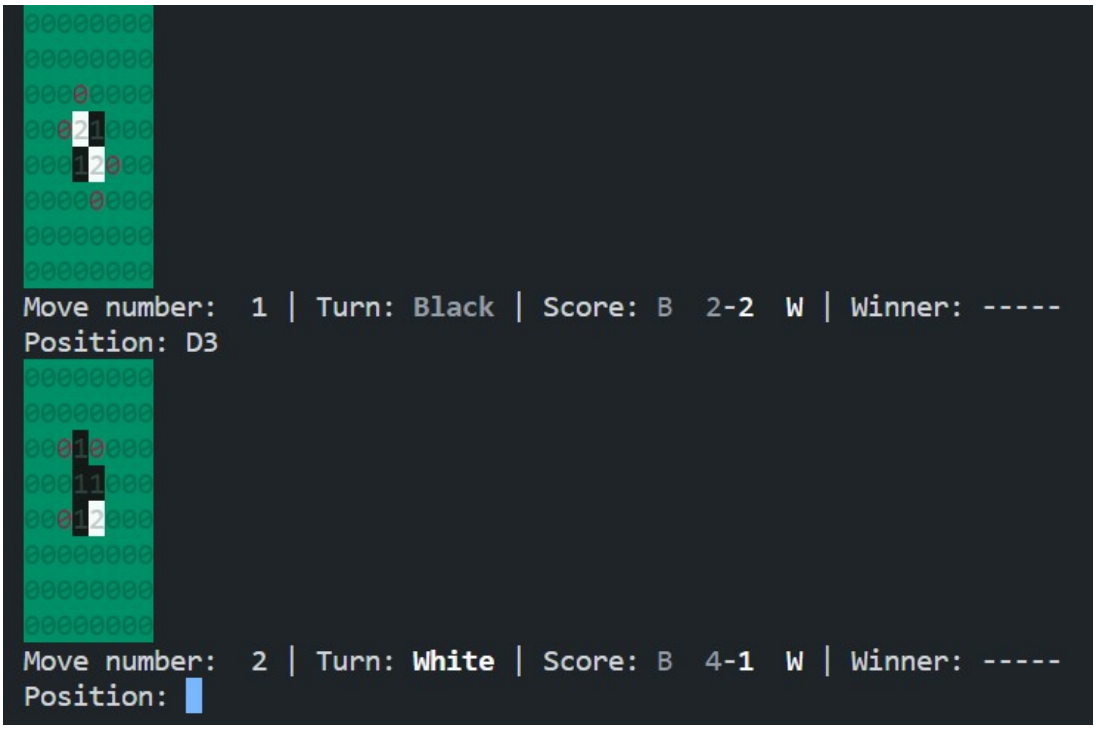
```rust
pub fn make_move(&self, position: Position) -> GameState {
    if !self.is_valid(position) { panic!("Invalid move!"); }
    let mut next_state = (*self).clone();
    next_state.board[position.index()] = Square::Placed(self.turn);
    for dir in DIRECTIONS {
        let mut current = position;
        let mut flip_queue = Vec::new();
        while let Some(next) = current.offset(dir) {
            current = next;
            match self.at(current) {
                Square::Placed(p) if p == self.turn.opponent() =>
                    flip_queue.push(current),
                Square::Placed(_) => {
                    flip_queue
                        .iter()
                        .for_each(|p| next_state.board[p.index()] =
                            Square::Placed(self.turn));
                    break;
                }
                Square::Empty => break,
            }
        }
    }
    next_state.turn = state.turn.opponent();
    next_state.pass_if_required();
    next_state
}
```

Listing 5: Początkowa wersja funkcji wykonującej ruch.

Metoda produkuje nową instancję struktury zamiast modyfikować obecnej. Takie rozwiązanie ułatwi później przechodzenie po drzewie stanów (nie będzie potrzeby tworzenia funkcji cofającej ruch) kosztem spowolnienia algorytmu o (praktycznie pomijalny) czas kopiowania stanu.

W tym momencie utworzyłem też pomocnicze metody, m.in. konstruktory stanu gry, wyświetlanie planszy w konsoli, obliczanie numeru ruchu, sprawdzanie warunków końcowych meczu. W celu przetestowania zaimplementowanych do tej pory funkcji utworzyłem na bazie tworzonej biblioteki program `pvp`, który pozwala rozegrać mecz między dwoma ludzkimi graczami korzystającymi ze wspólnego wiersza poleceń.

Na Rys. 3 został przedstawiony fragment rozgrywki z ww. programu. Po każdym ruchu wyświetlany jest stan gry, tj. plansza w formacie zgodnym z poleceniem, ale pokolorowanym pakietem `colored`, numer ruchu, obecny gracz, wynik w opisanym w poprzednim rozdziale formacie i ewentualny zwycięzca. Gracze stawiają naprzemiennie piony wpisując w wiersz poleceń notacje pól. Dostępne obecnie ruchy są wyróżnione spośród wszystkich pustych pól czerwoną czcionką.



Rysunek 3: Fragment rozgrywki w programie `pvp`.

2.3 Obliczenie ogólnych statystyk gry

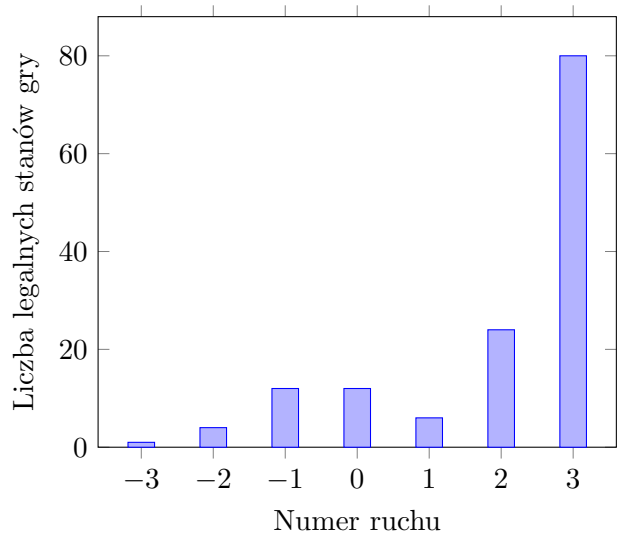
W poleceniu znajduje się założenie, że gdy *„liczba 1 oraz 2 jest równa, następuje tura gracza 1”*. Wydaje mi się ono niezbyt poprawne i w celu potwierdzenia moich obaw postanowiłem obliczyć dla jakiego procentu wszystkich stanów gry to stwierdzenie jest poprawne oraz znaleźć ewentualną alternatywę. Przy okazji obliczyłem kilka innych, ciekawych zależności.

Proces polegał na rozgrywaniu gier, w których obaj gracze wykonują losowe ruchy i dodawaniu wszystkich napotkanych stanów gry do zbioru, aż do momentu znalezienia dziesięciu milionów różnych stanów. Następnie dla zebranych stanów porównałem zapisanego w zbiorze obecnego gracza z graczem wywnioskowanym za pomocą podanej na liście zależności. Wymieniona zasada była poprawna **tylko dla 67%** badanych stanów gry. Oznacza to, że jedna na trzy podane przez użytkownika plansze byłaby rozpatrywana błędnie.

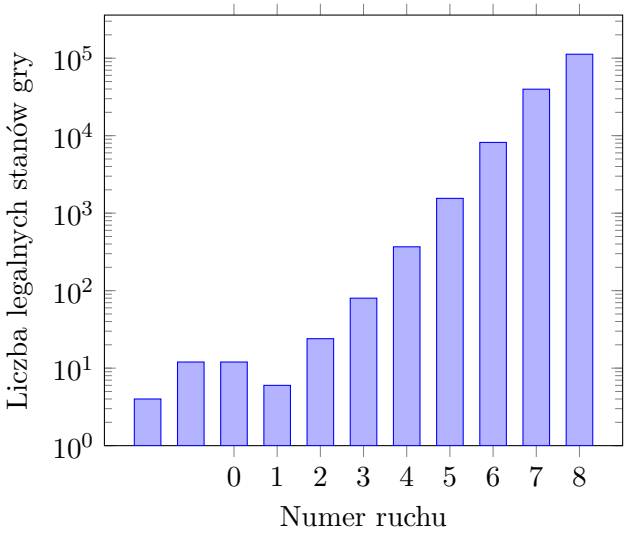
Alternatywną regułą, która moim zdaniem sprawdziłaby się lepiej jest: jeżeli liczba pól zajętych na planszy jest parzysta następuje tura gracza 1, w przeciwnym wypadku następuje tura gracza 2. Sprawdziłem również poprawność tej zasady z danymi, tym razem osiągając wynik 98.27%. Na logikę, ta reguła zostaje złamana tylko, jeżeli w grze nastąpi (nieparzysta liczba) pominiętych tur, a pomijanie tur następuje w Reversi dość rzadko. **W dalszych zadaniach wykorzystałem tę regułę zamiast poprzedniej pomimo niezgodności z poleceniem.**

Następnie policzyłem w ilu grach następuje pominięcie tury osiągając wynik rzędu 1.5%, co jest zgodne z poprzednimi wnioskami. Pomimo tak niskiego wpływu mechaniki pomijania tur na rozgrywkę, uważam, że warto ją uwzględnić i dobrze przetestować.

Następnie zmierzyłem liczbę osiągalnych, legalnych stanów gry dla każdego numeru ruchu. Na wykresach zawarto jedynie stany z niskimi numerami ruchu. Na Rys. 4 widzimy anomalie przy najniższych numerach ruchu, wynikające z symetrii planszy i powodujące wprowadzenie w Othello zasady dotyczącej układu początkowego planszy (dzięki któremu gra zaczyna się na numerze ruchu równym 1). Natomiast Rys. 5 zawiera liczbę stanów gry dla numerów ruchu od -2 do 8 włącznie w skali logarytmicznej. Można zauważyć, że od ruchu pierwszego, ilość legalnych stanów gry rośnie wykładniczo.



Rysunek 4: Liczba stanów dla ruchów od -3 do 3.



Rysunek 5: Liczba stanów (**w skali logarytmicznej**) dla ruchów od -2 do 8.

Następnie obliczyłem liczbę stanów końcowych (czyli takich, gdzie żaden z graczy nie ma legalnego ruchu i jeden z nich wygrywa bądź dochodzi do remisu) w zależności od numeru ruchu. **Ponad 99%** rozgrywek kończy się po ruchu 60, czyli dochodzi do zapełnienia całej planszy. Natomiast możliwe jest zakończenie gry przed tym momentem. Program znalazł ponad 1500 końcowych stanów gry z numerem ruchu 59 (czyli niemożliwe jest postawienie dopiero ostatniego piona). Możliwość wyczerpania wszystkich ruchów jest bardzo rzadka, ale występuje dla około połowy numerów ruchu. Warto zauważyć, że znajdowane jest wtedy zazwyczaj 2, 4 lub 8 stanów gry, co wynika z symetrii planszy. Najkrótsza możliwa gra, rozpoczynająca się od wzoru krzyża składa się z 9 ruchów. Istnieje kilka takich stanów, ale wszystkie są w zasadzie symetrią jednego i kończą się zwycięstwem gracza pierwszego z wynikiem 13–0. Jako pierwszy, konfigurację tę znalazł Manubu Maruo [18]. Obraz takiej planszy jest przedstawiony na stronie zawartej w bibliografii na pozycji [18], ale ten stan gry zostanie omówiony również później wraz z innymi ciekawymi rezultatami rozgrywek.

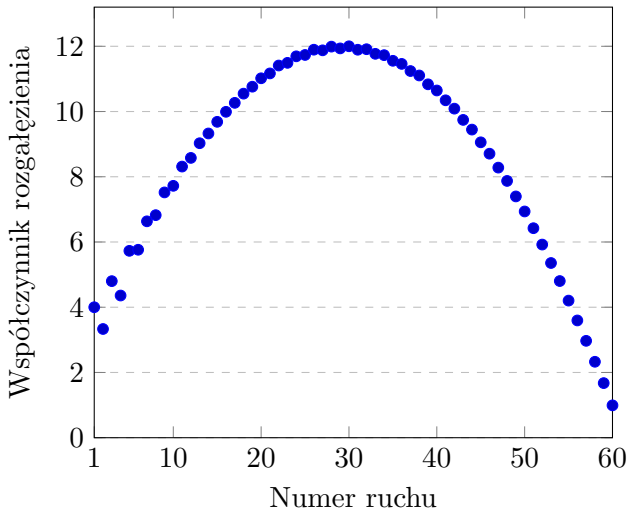
Ostatnią obliczoną zależnością jest tzw. *branching factor*, który od tej pory będzie nazywany *czynnikiem rozgałęzienia*, ponieważ nie byłem w stanie znaleźć w żadnej pozycji utartego polskiego odpowiednika tego pojęcia. Oznacza on liczbę legalnych ruchów dla danego stanu gry. Możemy mówić o minimalnym, maksymalnym, średnim i oczekiwanym współczynniku [19].

Najprostszym do policzenia i jednocześnie najmniej przydatnym jest współczynnik minimalny. Oznacza on najniższą liczbę możliwych ruchów dla danego stanu gry. Nawet bez wykonywania żadnych obliczeń, widzimy, że wyniesie on 0, ponieważ w stanach końcowych żaden z graczy nie może wykonać żadnego ruchu. Wraz z brakiem cykli w drzewie gry (a nie występują one, ponieważ każdy ruch dodaje dokładnie jeden pion na planszę, natomiast nie istnieje możliwość zabrania pionów z planszy), gwarantuje nam to skończoność drzewa gry. Gdyby nie to, przeszukiwanie drzewa mogłoby zakończyć się nieskończoną rekurencją.

Kolejną, ciekawszą liczbą jest maksymalny współczynnik rozgałęzienia, który mówi o tym, jaki jest najbardziej pesymistyczny przypadek dla algorytmów przeszukujących drzewo, tzn. jaka jest największa liczba ruchów dla dowolnego stanu gry. Wśród analizowanych dziesięciu milionów stanów, maksymalny współczynnik rozgałęzienia wyniósł **26**. Oznacza to, że w skrajnych przypadkach gracz musi rozważyć ponad 1/3 całej planszy jako możliwe ruchy.

Dwie ostatnie metryki to średni i oczekiwany współczynnik rozgałęzienia. Pierwszy z nich to średnia po wszystkich odkrytych stanach, natomiast drugi uwzględnia też numer ruchu. Wartość oczekiwaną otrzymujemy licząc średnią wśród stanów o danym numerze ruchu, a następnie biorąc średnią z tych średnich. Stanów późniejszych jest znacznie więcej niż stanów wczesnych, więc dominują one średni współczynnik rozgałęzienia, ale oczekiwanego już nie. Wśród analizowanych danych, średni współczynnik wyniósł 8.84, a oczekiwany 8.03. Oznacza to, że w trakcie analizy drzewa będziemy musieli zazwyczaj sprawdzić **aż 8 dzieci** dla jednego węzła. Wynika stąd, że zaimplementowane później algorytmy przeszukwiania drzewa będą musiały przejrzeć 8^n wierzchołków, żeby sprawdzić n ruchów w głąb, więc prawdopodobnie nie będą w stanie działać zbyt głęboko.

Poniżej, na Rys. 6 zostały przedstawione wartości średnie współczynników rozgałęzienia dla stanów o poszczególnych numerach ruchu. Łatwo zauważyć, że najtrudniejsze będzie wyznaczanie ruchu gracza w środkowej fazie gry.



Rysunek 6: Współczynniki rozgałęzienia dla stanów o różnych numerach ruchu.

2.4 Walidacja stanu wejściowego

Warto zauważyć, że nie każde ułożenie pionów na planszy jest osiągalne za pomocą legalnych ruchów z położenia początkowego. Przykładowo niemożliwe jest osiągnięcie stanu z dwoma (lub więcej) oddzielonymi od siebie „grupami" pionów. W tradycyjnych programach grających

w Reversi nie jest to problemem, ponieważ zawsze zaczynamy od planszy początkowej. W zadaniu należy natomiast podać stan gry z wiersza poleceń, co pozwala użytkownikowi wprowadzić niepoprawne dane wejściowe.

W celu zminimalizowania szansy na pomyłkę staramy się weryfikować poprawność każdej planszy podanej przez użytkownika. *Niezweryfikowanym* stanem gry nazwiemy taki, który mógł potencjalnie zostać osiągnięty wykonując legalne ruchy z pozycji początkowej, ale nie jest to udowodnione. Analogicznie, *zweryfikowany* stan gry to taki, co do którego mamy pewność, że mógł zostać osiągnięty w normalnej rozgrywce.

Problem ten z powodów określonych w pierwszym akapicie nie jest dobrze zbadany. Nie był poruszany w żadnej ze sprawdzonych przeze mnie prac naukowych, a jedynie na forach tematycznych [20][21]. Najlepszym opracowanym do tej pory sposobem jest przeszukanie całego drzewa gry od korzenia z odpowiednim obcinaniem nieprzydatnych stanów.

Biorąc pod uwagę, że możemy jedynie stawiać nowe piony, a niemożliwe jest usuwanie bądź przesuwanie poprzednich, pierwszym oczywistym cięciem jest odrzucenie wszystkich ruchów, które stawiają pion na niezajętym w planszy wynikowej polu. Kolejną optymalizacją jest trzymanie listy pionów *pierwotnych*, czyli takich, które nie mogły zostać nigdy odwrócone. Są to piony, które na każdej przekątnej mają maksymalnie jednego niepierwotnego sąsiada. Warto zauważyć, że jest to reguła rekurencyjna. Znalezienie pierwotnych pionów polega zatem na iteracyjnym oznaczaniu pionów spełniających warunki reguły i powtarzaniu procesu aż do momentu kiedy iteracja nie znajduje żadnych nowych pierwotnych pionów. Ostatnie wykorzystane przeze mnie cięcie odrzuca stany z nieodpowiednim polem `gs.turn`, czyli graczem rozgrywającym obecną turę. Odwiedzone stany są zapamiętywane w zbiorze zapobiegając duplikatowemu przetwarzaniu tego samego węzła.

Proces weryfikacji działa w czasie wykładniczym, jednakże dla niektórych stanów bardzo szybko potwierdza lub obala ich osiągalność. Dlatego jako wyniki weryfikacji przyjmujemy potwierdzenie lub obalenie poprawności stanu ale też przekroczenie limitu czasu. Zamysł algorytmu, bez szczegółów implementacyjnych pokazuje Lis. 6.

```
impl GameState {
    pub fn from_board_str_unverified(board_str: &str) -> Option<Self> {
        /* Parse game state from input string... */
    }

    pub fn verify_reachability(&self, timeout: Duration) -> Option<bool> {
        let start_time = Instant::now();
        /* Calculate occupied and original discs, initialize visited set... */
        let mut stack = Vec::from([GameState::othello_initial()]);
        while let Some(current) = stack.pop() {
            if current == *self { return Some(true); }
            if Instant::now() - start_time >= timeout { return None; }
            for pos in current.moves() {
                /* Prune based on occupied squares, original
                   squares and the visited set... */
                stack.push(current.make_move(pos));
            }
        }
        Some(false)
    }
}
```

Listing 6: Zarys ogólny logiki weryfikacji stanu gry.

2.5 Implementacja abstrakcyjnej strategii

Kolejnym krokiem było zauważenie, że wszystkie strategie gracza komputerowego jak i akcje ludzkiego gracza można zgeneralizować do wspólnego interfejsu, który zapewnia funkcję wyboru ruchu $f : GameState \to Position$. Pożądaną cechą późniejszych programów jest wybór strategii, czy też heurystyki przez użytkownika, zatem należy zastosować polimorfizm dynamiczny ad-hoc. W języku Rust rolę tę realizują cechy [22], występujące też np. w Scali i PHP, które można rozumieć jako bardziej zaawansowana wersja interfejsów z języków obiektowych. Pełną definicję cech realizujących abstrakcyjną strategię zawiera Lis. 7.

```
pub trait Strategy: Display + Sync {
    fn decide(&self, gs: &GameState) -> Position;
}

pub trait TreeVisitingStrategy: Strategy {
    #[must_use]
    fn visited(&self) -> u32;
}
```

Listing 7: Implementacja abstrakcyjnej strategii.

Powyższy kod można rozumieć następująco: cecha `Strategy` wymaga, aby struktury, które ją implementują zawierały metodę `decide` o sygnaturze opisanej na początku sekcji, oraz implementowały też cechy `Display` (mogły być zamieniane na ciąg tekstowy, np. na cele wypisu w konsoli) i `Sync` (mogły być bezpiecznie przesyłane między wątkami). Definiowana jest również cecha `TreeVisitingStrategy`, która pomoże później w implementacji wypisywania ilości odwiedzonych przez algorytm wierzchołków drzewa gry.

Pierwszą implementacją cechy jest typ `PlayerInput`, który po wywołaniu metody `decide` za pomocą wiersza poleceń pyta użytkownika o podanie ruchu. W celach późniejszego testowania algorytmów zdefiniowano też szereg „naiwnych" strategii:

- `FirstMove` – wybiera dostępny ruch z najniższym indeksem
- `RandomMove` – wybiera losowy dostępny ruch
- `ScoreGreedy` – wybiera ruch odwracający najwięcej pionów przeciwnika
- `CornersGreedy` – zajmuje losowe dostępne pole narożne, jeżeli takie nie istnieją, zachowuje się zgodnie z logiką `RandomMove`

Strategie te zostaną porównane ze strategią Minimax, jeżeli jakaś heurystyka nie jest w stanie ich pokonać, to nie została dobrze zaprojektowana. Implementacja tych typów została pominięta, ponieważ ich logika jest bardzo prosta.

2.6 Testy jednostkowe i integracyjne

W tym momencie realizacji listy została zaimplementowana cała logika gry Reversi, możliwość wypisywania planszy na konsolę jak i wczytywania ruchów od użytkownika. Stany gry podane przez użytkownika są dodatkowo weryfikowane. Powstały też proste strategie, które są w stanie konkurować z graczem.

Aby sprawdzić, czy cała logika gry, która zawiera zaskakująco dużą ilość przypadków szczególnych działa poprawnie postanowiłem pokryć kod testami jednostkowymi i integracyjnymi. Jest to również rekomendowane przez twórców czołowych programów do gry w Othello [23].

Zaimplementowane testy uwzględniają: potwierdzenie rozmiaru reprezentacji struktur (typy `Player` i `Square` mają mieć zawsze rozmiar jednego bajtu), sprawdzenie poprawności dostępnych

ruchów dla początkowych faz gry, potwierdzenie, że funkcje zwracają błędy po podaniu błędnych wartości.

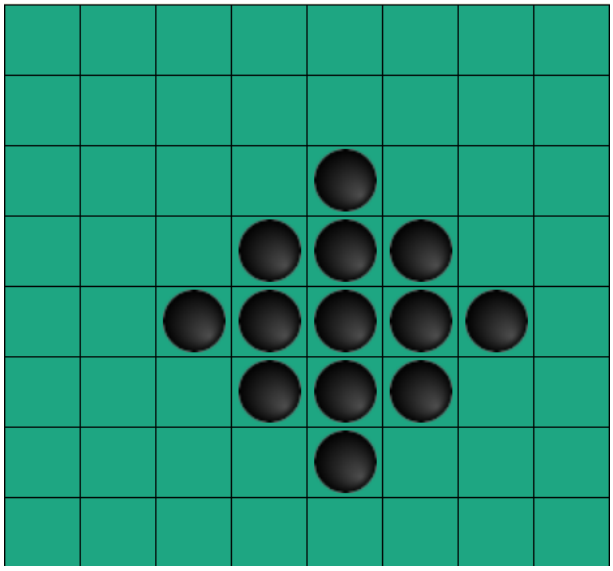
Zastosowałem też metodę zwaną w zależności od źródła *property testing* lub *hypothesis testing* [24] polegającą na weryfikacji niezmienników programu, podając do jego funkcji ogromną ilość pseudolosowych (ale deterministycznych, z powodu wykorzystania ziarna) danych. Możemy w ten sposób sprawdzić np. zależności dotyczące struktury `Position`, a dokładniej odwrotność funkcji `from_index` i `index` oraz `from` i `to_string`. Tego typu testy są w języku Rust realizowane pakietem `quicktest` [25]. Wspomniane testy znajdują się na Lis. 8. Biorąc pod uwagę, że możliwych wartości typu `Position` jest jedynie 64, zadeklarowane funkcje zostaną wywołane dla każdej z nich, zapewniając, że nie został pominięty żaden przypadek graniczny. Podobnego typu testy zostały napisane m.in. do weryfikacji zawarcia numeru ruchu, czy też ilości dostępnych ruchów w odpowiednich przedziałach dla wszystkich stanów gry.

```
#[quickcheck]
fn to_string_and_from_are_inverses(position: Position) -> bool {
    Position::from(&position.to_string()) == Some(position)
}

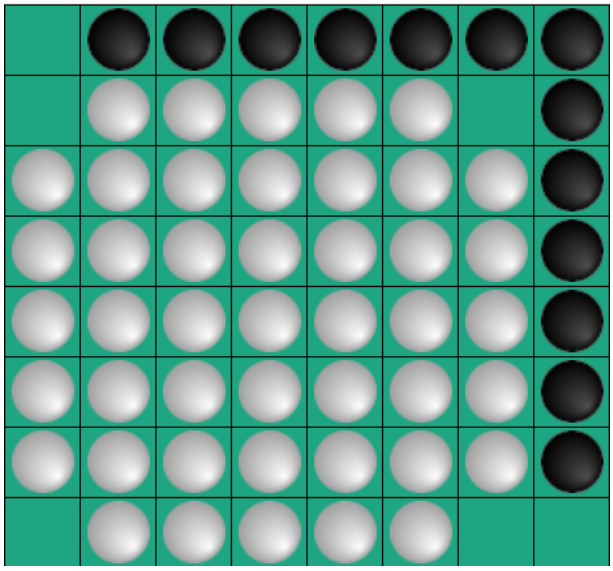
#[quickcheck]
fn index_and_from_index_are_inverses(position: Position) -> bool {
    Position::from_index(position.index()) == position
}
```

Listing 8: Testowanie inwariantów struktury `Position`.

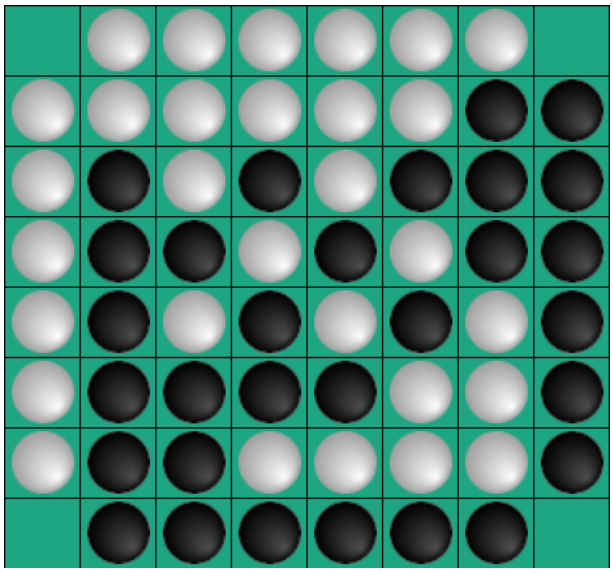
Testy integracyjne polegały natomiast na rozgrywaniu przez program gier z transkryptów oficjalnych rozgrywek [26]. Na każdym etapie sprawdzana jest integralność stanu wewnętrznego gry z zapiskami historycznymi. Dobrane zostały celowo rozgrywki zawierające sytuacje nietypowe, to znaczy: remisy (przed i po 60 ruchu), najkrótsze możliwe rozgrywki, oddawanie tury przeciwnikowi (w tym kilka razy w trakcie meczu), zakończenie meczu poprzez brak możliwych ruchów dla obu graczy i tym podobne. Najciekawsze znalezione przeze mnie stany końcowe gry znajdują się na rysunkach poniżej.



Rysunek 7: Najkrótsza możliwa gra, Manubu Maruo 1975.



Rysunek 8: Gra zakończona bez wypełnienia planszy, Vecchi vs. Nicolas 2017.



Rysunek 9: Remis bez wypełnienia planszy, Othello Quest 2017.

2.7 Opytmalizacja z użyciem bitboardów [Z1]

Po dokładnym pokryciu programu testami, postanowiłem zoptymalizować funkcję zwracającą dostępne ruchy oraz wykonującą ruch, bez obaw o wprowadzenie regresji. Logika ta wymaga przejścia przez kilkukrotnie zagnieżdżone pętle, a jest najczęściej wykonywaną akcją w utworzonych do tej pory programach. Później, dla algorytmu Minimax byłoby to największym ograniczeniem szybkości przejścia po drzewie. Obecną wersję tych funkcji, przedstawioną w sekcji „Funkcja generująca możliwe ruchy" nazwiemy `V1`.

Pierwszym podejściem było zapisywanie listy dostępnych ruchów w stanie gry przy jego tworzeniu. Optymalizacja ta opiera się na fakcie, że lista dostępnych ruchów jest zawsze obliczana co najmniej raz dla danego stanu gry. Każda operacja wykonania ruchu sprawdza też listę ruchów, aby mieć pewność, że dany ruch jest legalny. W przykładowej sytuacji, załóżmy, że dla danego stanu gry, chcemy sprawdzić listę dostępnych ruchów, a następnie wykonać każdy z nich zwracając nowe stany gry. Wywołujemy funkcję `moves`, która zwraca liczbę n. Następnie,

przy wykonywaniu n ruchów musimy ponownie n razy wywołać funkcję `moves`, aby zweryfikować poprawność tych ruchów. Oznacza to, że dla każdego odwiedzonego algorytmem Minimax węzła drzewa **musimy obliczyć** $n+1$ **razy liczbę dostępnych stanów** lub pogodzić się z możliwością wykonania niepoprawnego ruchu i usunąć zabezpieczenia z metody `make_move`.

Obliczenie tej wartości raz i zapisanie pozwala skrócić $n+1$ razy czas przetwarzania danego węzła decyzyjnego, ale oczywiście nie skraca samego w sobie generowania listy ruchów. W tym wariancie struktura `GameState` zyskuje dodatkowe pole `moves: Box<[Position]>`, gdzie `Box` oznacza inteligentny wskaźnik, zapisujący listę pozycji na stosie. Tę wersję metod nazwiemy `V2`.

Przełomową optymalizacją okazało się jednak zastosowanie tzw. *bitboardów*, czyli enkodowanie planszy za pomocą liczb binarnych i operowanie na polach planszy za pomocą operacji bitowych [27]. Pomysł ten jest znany głównie z szachów, ale jest też bardzo często wykorzystywany w Reversi [23]. Szczęśliwym trafem, plansza Reversi składa się z 64 pól, co oznacza, że możemy ją zakodować za pomocą dwóch liczb 64 bitowych – jednej dla pionów czarnych i jednej dla białych. Oczywistą zaletą, na ten stopień implementacji jest spadek zużycia pamięci. Wykorzystana na zapis pozycji pionów pamięć **spada z 64 bajtów na 16 bajtów**. Natomiast sporą wadą jest powstanie możliwości istnienia stanu z dwoma pionami na jednym polu. Aby zapobiec ewentualnej pomyłce, każda metoda zwracająca instancje `GameState` zawiera teraz na końcu klauzulę `assert_eq!(black & white, EMPTY)`. Implementacja struktury `GameState` za pomocą bitboardów została przedstawiona na Lis. 9. Warto zauważyć, że dostępne ruchy nie są już zapisywane.

```rust
pub type Bitboard = u64;

#[must_use]
#[derive(Clone, PartialEq, Eq, Hash, Debug)]
pub struct GameState {
    turn: Player,
    black: Bitboard,
    white: Bitboard,
}
```

Listing 9: Reprezentacja stanu gry wykorzystująca bitboardy.

Idea działania algorytmu zwracającego dostępne ruchy oraz wykonującego ruch jest identyczna jak poprzednio, tylko że za pomocą operacji bitowych jesteśmy w stanie „zrównoleglić" kroki wykonywane poprzednio na pojedynczych polach na wszystkie pola planszy. Daje to (czysto teoretycznie) 64–krotne przyspieszenie obu tych algorytmów. Jak się okazuje, logika sprawdzania ruchów w Reversi jest bardzo podobna do sprawdzania ruchów królowej w szachach, co pozwoliło zaadaptować algorytm znany jako Dumb7Fill [28] na cel poszukiwania legalnych ruchów. Pierwsza wersja takiej implementacji, która stanowi w zasadzie przetłumaczenie przedstawionego na Chess Programming Wiki algorytmu z języka C++ na Rust, bez wykorzystania optymalizacji specyficznych dla Reversi ani Rust, będzie zwana `V3`. Kolejny krok stanowiło uproszczenie procedur tak, aby nie obejmowały przypadków występujących szachach, ale nieistniejących w Othello oraz sprawienie, żeby kod był bardziej idiomatyczny dla wykorzystywanego przeze mnie języka programowania. Wersję po tych poprawkach nazwiemy `V4`. Nowy system sprawdzania ruchów działa następująco:

1. Enkodujemy kierunki świata jako liczby, odpowiadające przesunięciu indeksu z obecnego pola do sąsiada znajdującego się w tym kierunku.

2. Implementujmy operacje `shift` oraz `fill`. `shift` przesuwa maskę bitów w danym kierunku świata unikając przepełnienia, natomiast `fill` (zaimplementowany jako Dumb7Fill) kopiuje i przesuwa maskę w danym kierunku dopóki znajdują się w nim tzw. bity propagatora.

Działanie wypełniania możemy porównać do narzędzia „Wypełnij kolorem” z programu Paint (które zostało zaimplementowane właśnie tym algorytmem lub jego odpowiednikiem), tyle że działającego jedynie w określonym kierunku.

3. Implementujemy specyficzne przypadki operacji, po jednym w każdym kierunku, tak aby później nie musieć dbać o zgodność masek z kierunkami propagacji.

4. Z powstałych prymitywów składamy funkcje `attack_fill` oraz `all_flipped` zwracające odpowiednio maski dostępnych ruchów oraz odwróconych przy wykonaniu ruchu krążków.

Wybrane części algorytmu przedstawia Lis. 10.

```
/* 1. */
const EAST: i32 = 1; /* etc. for other directions */

/* 2. */
const fn shift(bitboard: Bitboard, by: i32, mask: Bitboard) -> Bitboard {
    mask & if by >= 0 { bitboard >> by } else { bitboard << -by } }

const fn fill(mut gen: Bitboard, mut pro: Bitboard,
              dir: i32, mask: Bitboard) -> Bitboard {
    /* Implementation from https://chessprogramming.org/Dumb7Fill */ }

/* 3. */
pub const fn shift_east(bb: Bitboard) -> Bitboard {
    shift(bb, EAST, NOT_H_FILE) } /* etc. for other directions */

pub const fn fill_east(gen: Bitboard, pro: Bitboard) -> Bitboard {
    fill(gen, pro, EAST, NOT_H_FILE) } /* etc. for other directions */

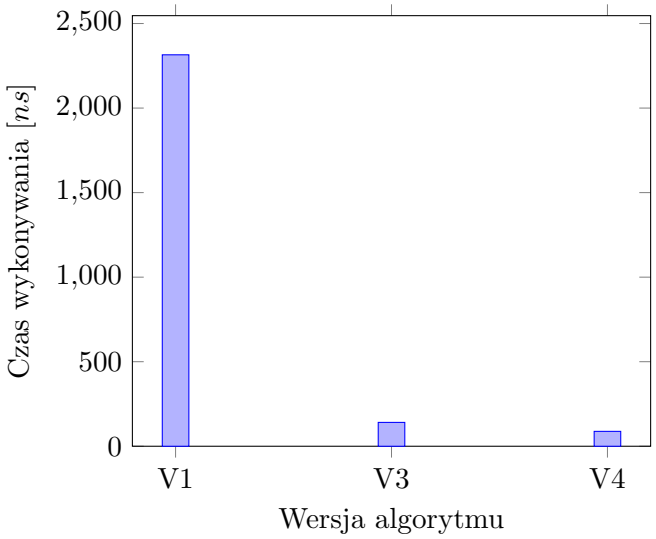
/* 4. */
const fn all_flipped(position: Bitboard, current: Bitboard, opponent: Bitboard) -> Bitboard
    fill_nort(position, opponent) & fill_sout(current, opponent)
        | fill_noea(position, opponent) & fill_sowe(current, opponent)
        | fill_east(position, opponent) & fill_west(current, opponent)
        | fill_soea(position, opponent) & fill_nowe(current, opponent)
        | /* etc. */
}

const fn attack_fill(current: Bitboard, opponent: Bitboard) -> Bitboard {
    !(current | opponent)
        & (shift_nort(fill_nort(current, opponent))
            | shift_noea(fill_noea(current, opponent))
            | shift_east(fill_east(current, opponent))
            | shift_soea(fill_soea(current, opponent))
            | /* etc. */)
}
```

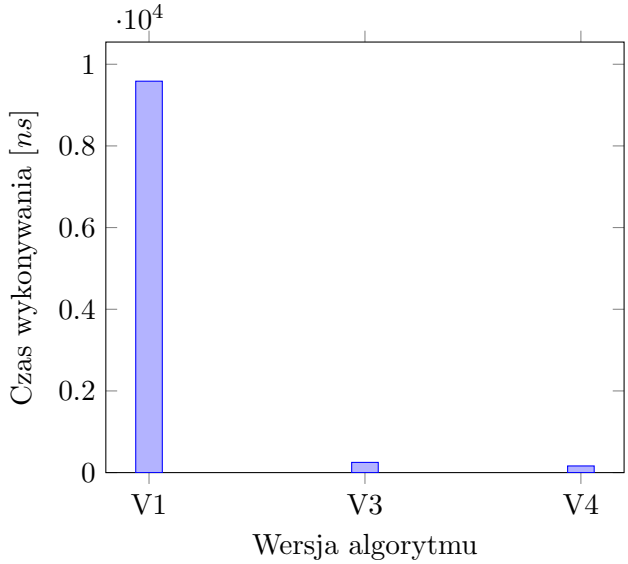
Listing 10: Implementacja wersji `V4` algorytmów.

Warto zauważyć, że wszystkie wykorzystane funkcje mają adnotację `const`, czyli mogą być ewaluowane w pełni w czasie kompilacji programu (analogicznie do `constexpr` z C++), kosztem korzystania jedynie z podzbioru funkcjonalności języka. Dzięki temu, wyrażenia typu `GameState::othello_initial().moves()` są optymalizowane przez kompilator do zwykłej liczby.

Interfejs publiczny struktury `GameState` nie zmienił się w żadnym stopniu. Jedynie oprócz metody `moves` została dodana metoda `move_bb`, która zwraca bitboard dostępnych ruchów zamiast listy, ponieważ będzie to przydane przy optymalizacji niektórych heurystyk. Wyniki optymalizacji zostały zaprezentowane na wykresach 10 i 11 poniżej. Wersja `V2` nie została zawarta w zestawieniu, ponieważ nie różniła się od wersji `V1` czasem wykonania, a jedynie gwarantowała, że ten czas będzie spędzony jedynie raz dla danego stanu gry. Wszystkie pomiary zostały dokonane uruchamiając metodę w izolacji kilka tysięcy razy, dla różnych, losowych stanów gry.



Rysunek 10: Czas wykonywania różnych wersji metody `make_move`.



Rysunek 11: Czas wykonywania różnych wersji metody `moves`.

Jak widać nie jest już raczej konieczne zapamiętywanie lokalnie listy dostępnych ruchów. Optymalizacja przyspieszyła obie metody mniej więcej 20–krotnie. Wszystkie testy jednostkowe i integracyjne poprawnie przechodzą przy nowej implementacji.

3 Metoda Minimax

3.1 Implementacja algorytmu [Z3]

„3. implementacja metody Minimax z punktu widzenia gracza 1”

W grach dwuosobowych, **algorytmem Minimax** nazywamy taką taktykę, która zakłada, że gracze mają przeciwny cel, aby przewidzieć jakie ruchy wykonają gracze (zakładając, że obaj grają optymalnie) i na podstawie tego wybiera odpowiedni ruch [29]. Przechodzimy po drzewie gry w głąb do momentu dojścia do węzła końcowego lub osiągnięcia limitu głębokości. Zastosowanie tego limitu jest niezbędne, ponieważ dla Reversi przegląd całego drzewa jest niewykonalny na współczesnym sprzęcie. Wybieramy gracza maksymalizującego i minimalizującego. W przypadku dojścia do węzła końcowego, przypisywana jest mu wartość wartość wyniku gry – ujemna jeżeli wygrał gracz minimalizujący, dodatnia jeżeli wygrał gracz maksymalizujący i równa zero w przypadku remisu. Dla węzłów osiągniętych poprzez dojście do limitu rekurencji należy zamiast tego użyć heurystyki, która estymuje jak korzystna dana pozycja jest dla graczy. Algorytm zakłada, że obaj gracze grają optymalnie, więc na poziomach gdzie ruch wykonuje gracz minimalizujący bierzemy minimum z wartości kolejnych ruchów, a przy ruchach gracza maksymalizującego liczymy maksimum.

Chcemy, żeby algorytm zawsze wybierał wariant gwarantujący zwycięstwo, a w przypadku zwracania wyniku gry w węzłach końcowych, może dojść do tego, że algorytm wybierze stan gry niegwarantujący zwycięstwa ale posiadający korzystniejszą wartość heurystyczną niż sąsiedni stan końcowy. Innymi słowy, jeżeli dowolna heurystyka jest zwraca wartości wyższe od 64 lub niższe od -64 (które są najbardziej skrajnymi przypadkami zwycięstwa), algorytm może „przegapić” sytuację gwarantującą mu pewne zwycięstwo. Aby temu zapobiec, dla węzłów końcowych zwracamy zamiast wyniku gry odpowiednio $+\infty$ lub $-\infty$ w zależności od tego, który z graczy wygrał. Jest to również poprawny wariant algorytmu Minimax [30].

Początkowa implementacja algorytmu została zademonstrowana na Lis. 11. Jest to praktycznie książkowa implementacja tego algorytmu, zainspirowana pseudokodem z filmu „Algorithms Explained – minimax and alpha-beta pruning” [31].

Istotną różnicą implementacji z Lis. 11 względem typowego algorytmu, jest pominięcie argumentu wskazującego czy na danym poziomie należy maksymalizować czy minimalizować wynik. Zastosowanie go tutaj (bez modyfikacji) **byłoby błędne, ponieważ gracze nie zawsze wykonują tury naprzemiennie**. W przypadku pominięcia tury z powodu braku dostępnego ruchu, jeden gracz wykonuje ruch dwa razy pod rząd. Chcemy wtedy maksymalizować bądź minimalizować dwa razy pod rząd, co nie jest zachowaniem klasycznej implementacji tego algorytmu.

Zamiast tego, o tym, czy maksymalizujemy czy minimalizujemy wynik w obecnym wywołaniu funkcji decyduje pole `turn` struktury `GameState`, które zazwyczaj (ale nie zawsze) zamienia się na przeciwne przy wykonaniu ruchu.

Następnie zmodyfikowano algorytm w celu zredukowania duplikacji kodu między dwoma gałęziami funkcji. Finalną implementację algorytmu Minimax, wraz z opakowującą go strukturą logiką doboru heurystyki oraz liczeniem odwiedzonych węzłów drzewa demonstruje Lis. 12. Do liczenia odwiedzonych węzłów zostały wykorzystane zmienne atomowe, ponieważ pożądaną cechą strategii jest możliwość przesyłania jej między wątkami. Użyty podzbiór metod tego typu danych wykonuje się w jednej instrukcji assemblera, więc nie jest to zaimplementowane kosztem wydajności.

```rust
pub const MAX_PLAYER: Player = Player::Black;
pub const MIN_PLAYER: Player = Player::White;

#[must_use]
fn minimax(&self, gs: &GameState, depth: u32) -> (f64, Option<Position>) {
    if let Some(outcome) = gs.outcome() {
        return (
            match outcome {
                Outcome::Winner(MAX_PLAYER) => f64::INFINITY,
                Outcome::Winner(MIN_PLAYER) => f64::NEG_INFINITY,
                Outcome::Draw => 0.,
            },
            None,
        );
    }

    if depth == 0 {
        return (self.heuristic.evaluate(gs), None);
    }

    if gs.turn() == MAX_PLAYER {
        let (mut max_eval, mut max_pos) = (f64::NEG_INFINITY, None);
        for position in gs.moves() {
            let (eval, _) = self.minimax(&gs.make_move(position), depth - 1);
            if eval >= max_eval {
                max_eval = eval;
                max_pos = Some(position);
            }
        }
        (max_eval, max_pos)
    } else {
        let (mut min_eval, mut min_pos) = (f64::INFINITY, None);
        for position in gs.moves() {
            let (eval, _) = self.minimax(&gs.make_move(position), depth - 1);
            if eval <= min_eval {
                min_eval = eval;
                min_pos = Some(position);
            }
        }
        (min_eval, min_pos)
    }
}
```

Listing 11: Początkowa implementacja algorytmu Minimax.

```rust
#[must_use]
pub struct Minimax { heuristic: Heuristic, max_depth: u32, visited: AtomicU32 }

impl Minimax {
    pub const fn new(heuristic: Heuristic, max_depth: u32) -> Self {
        Self { heuristic, max_depth, visited: AtomicU32::new(0) }
    }

    #[must_use]
    fn minimax(&self, gs: &GameState, depth: u32) -> (f64, Option<Position>) {
        self.visited.fetch_add(1, atomic::Ordering::Relaxed);

        if let Some(outcome) = gs.outcome() {
            return (outcome.evaluate(), None);
        }

        if depth == 0 {
            return (self.heuristic.evaluate(gs), None);
        }

        let mut moves = gs.moves();
        let mut best_pos = moves.pop().unwrap();
        let (mut best_eval, _) = self.minimax(&gs.make_move(best_pos), depth - 1);
        for position in moves {
            let (eval, _) = self.minimax(&gs.make_move(position), depth - 1);
            if matches!(
                (gs.turn(), eval.partial_cmp(&best_eval).unwrap()),
                (MAX_PLAYER, Ordering::Greater) | (MIN_PLAYER, Ordering::Less),
            ) {
                best_eval = eval;
                best_pos = position;
            }
        }
        (best_eval, Some(best_pos))
    }
}

impl Strategy for Minimax {
    fn decide(&self, gs: &GameState) -> Position {
        let (_, pos) = self.minimax(gs, self.max_depth);
        pos.unwrap()
    }
}
```

Listing 12: Finalna implementacja algorytmu Minimax, wraz z jego strukturą.

3.2 Cięcie alfa–beta [Z4]

„4. implementacja alfa-beta cięcia z punktu widzenia gracza 1”

Obcinanie alfa–beta jest rozszerzeniem algorytmu Minimax, które go przyspiesza, ale nie zwraca jakości zwracanych przez niego rozwiązań (a przy odpowiedniej implementacji zwraca zawsze ten sam wynik). Optymalizacja opiera się na założeniu, że nie ma sensu przeszukiwać gałęzi, które nie mają szansy na poprawienie ogólnego wyniku dla analizowanego poddrzewa [3]. W tym celu zapamiętujemy dwie zmienne: α – największa wartość maksymalna oraz β – najmniejsza wartość minimalna. Wprowadzenie tej optymalizacji jest dość łatwe i sprowadza się zazwyczaj do dopisania kilku linijek do implementacji algorytmu Minimax.

W przypadku mojej wersji funkcji, wprowadzenie cięcia alfa–beta sprowadziło się do:

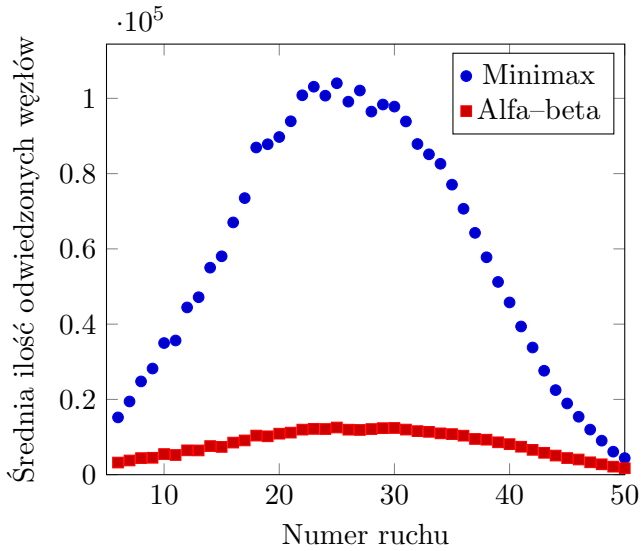
1. Dodania argumentów `mut alpha: f64, mut beta: f64` do sygnatury metody.
2. Dodania na końcu głównej pętli w funkcji bloku kodu zademonstrowanego w Lis. 13.

```
// Alpha-beta pruning
match gs.turn() {
    MAX_PLAYER => alpha = alpha.max(eval),
    MIN_PLAYER => beta = beta.min(eval),
}
if beta <= alpha {
    break;
}
```

Listing 13: Modyfikacja algorytmu Minimax o cięcie alfa–beta.

Zazwyczaj przypadki zawarte w poleceniu `match` są rozważane w osobnych fragmentach kodu, ale w tym przypadku udało się je ujednolicić. Zaimplementowany wariant (dzięki wykorzystaniu silnej nierówności zamiast słabej) gwarantuje zwracanie z tej funkcji identycznych wartości co z algorytmu Minimax.

Rys. 12 przedstawia średnią ilość odwiedzonych węzłów decyzyjnych w różnych fazach gry dla algorytmu Minimax z i bez cięć alfa–beta w różnych fazach gry.



Rysunek 12: Średnia ilość odwiedzonych węzłów decyzyjnych z i bez cięć.

3.3 Dobór heurystyk [Z2]

„2. zbudowanie zbioru heurystyk oceny stanu gry dla każdego z graczy, każdy z graczy powinien mieć przynajmniej 3 różne strategie"

3.3.1 Przewaga pionów (`MaximumDisc`, `MinimumDisc`)

Przewaga pionów (znana również jako *coin party*, *piece difference*) zwraca stosunek między pionami gracza maksymalizującego i minimalizującego. Opiera się na założeniu, że skoro grę wygrywa gracz posiadający większą ilość pionów na planszy, to istotne jest utrzymywanie ich dużej ilości [32]. Jest to generalizacja opisanej poprzednio strategii `ScoreGreedy` (która jest równoznaczna wywołaniu algorytmu z głębokością równą jeden i heurystyką `MaximumDisc`).

W rzeczywistości nie jest to dobra heurystyka, ponieważ w początkowych fazach gry pożądane jest wręcz przeciwne zachowanie – minimalizujemy ilość swoich pionów, aby gracz przeciwny miał jak najmniej dostępnych ruchów. Cytując Rosenblooma, *„One of the intriguing aspects of Othello is that this obvious strategy is a miserable failure"*. Z tego powodu czasami stosowana jest wręcz inna, pozornie bezsensowna heurystyka `MinimumDisc`, która minimalizuje przewagę pionów gracza obecnego [32][33].

3.3.2 Ważone pola (`Weighted`)

Strategia **ważonych pól** (znana również jako *disc squares*, *weighted square*, *static heuristic*) polega na nadaniu każdemu polu planszy dodatniej lub ujemnej wagi. Jeżeli gracz maksymalizujący ma na tym polu piona, dodajemy wartość wagi do akumulatora, jeżeli ma go gracz minimalizujący – odejmujemy wagę. Wartość akumulatora po przejściu przez wszystkie pola stanowi wynik heurystyki [34].

Jest to heurystyka osiągająca dość dobre wyniki i prosta w implementacji. Jej główną wadą jest trudność w dobraniu odpowiednich wag. Oryginalna macierz zaproponowana przez Maggsa w 1979 [34] działa o wiele gorzej niż warianty zaproponowane przez Vaishnaviego i Muthukaruppana [35], czy też Kormana [36]. W programie uwzględniłem powyższe trzy macierze.

3.3.3 Posiadane narożniki (`CornersOwned`)

`Posiadane narożniki` (znane również jako *corner occupancy*) zwracają stosunek zajętych przez obu graczy pól narożnych. Strategia bazuje na tym, że pionów postawionych na polach narożnych nie można nigdy obrócić, tzn. są *stabilne* i mają wpływ również na stabilizację innych pól [36]. Są to zdecydowanie najważniejsze pola na planszy.

3.3.4 Bliskość narożników (`CornerCloseness`)

Bliskość narożników (znana również jako *corner proximity*) nadaje ujemne wagi polom sąsiadującym z polami narożnymi, pod warunkiem, że te są puste [36]. Opiera się na tym, że stawienie piona na polu sąsiadującym z narożnikiem zazwyczaj pozwala przeciwnikowi stawić tam swojego piona, a jak zostało wspomniane powyżej są to kluczowe miejsca na planszy.

3.3.5 Obecna mobilność (`CurrentMobility`)

Obecna mobilność (znana również jako *actual mobility*) to stosunek ilości dostępnych ruchów gracza maksymalizującego do tych gracza minimalizującego [32]. Jest to heurystyka kluczowa w środkowej fazie gry, gdzie warto maksymalizować ilość ruchów gracza obecnego i minimalizować ilość ruchów przeciwnika. Brak uwzględnienia mobilności w heurystykach był jednym z głównych problemów wczesnych algorytmów do gry w Reversi.

3.3.6 Potencjalna mobilność (`PotentialMobility`)

Potencjalna mobilność (znana również jako *frontier discs*) to stosunek ilości pustych pól sąsiadujących z dowolnym pionem przeciwnika [32]. Idea jest prosta – tylko na takich polach mogą być stawiane piony obecnego gracza. Każdy ruch jest też potencjalnym ruchem, ale nie każdy potencjalny ruch jest legalny.

3.3.7 Stabilność (`Stability`, `InternalStability`, `EdgeStability`)

Heurystyka `stabilności` polega na obliczeniu stosunku stabilnych pionów gracza maksymalizującego i minimalizującego. *Stabilnymi* pionami nazywamy takie, które nie mogą zostać odwrócone aż do końca rozgrywki [32]. Jest to swego rodzaju odwrotność wspomnianych wcześniej pionów pierwotnych, które nie mogły być odwrócone od momentu stawienia.

Najprostszym przypadkiem są narożniki, które zawsze są stabilne, ponieważ na każdej przekątnej mają jedynie jednego sąsiada, więc nie mogą być flankowane. Pozostałe pola stabilne znajdujemy algorytmem iteracyjnym, podobnym do tego do obliczania pól pierwotnych. Wykorzystane struktury danych to zbiór nieprzetworzonych pól i bitboard stabilnych pól. Kroki algorytmu są następujące [32]:

1. Inicjalizujemy pusty zbiór nieprzetworzonych pól i pusty bitboard stabilnych pól.
2. Każde zajęte pole narożne oznaczamy jako stabilne i dodajemy do zbioru.
3. Dopóki zbiór nie jest pusty, usuwamy element i przetwarzamy go: dla każdego sąsiedniego pola, jeżeli jest ono stabilne i nieoznaczone, oznaczamy je i dodajemy do zbioru.
4. Zwracamy bitboard stabilnych pól.

Stabilność dzielimy na *wewnętrzną* – liczbę stabilnych pionów na polach wewnętrznych i *brzegową* – liczbę stabilnych pionów na polach brzegowych. Skoro konfiguracji pól brzegowych (dla konkretnego brzegu) jest jedynie kilkaset, w oryginalnej wersji algorytmu Rosenblooma stabilność była liczona dla każdej z tych konfiguracji i zapisywana w statycznej tablicy. Później obliczenie stabilności danego brzegu sprowadzało się jedynie do podejrzenia wartości z tablicy. Przy współczesnych prędkościach procesorów nie jest problemem obliczenie tej wartości za każdym razem algorytmem, więc na taki wariant się zdecydowałem.

3.3.8 Kombinacja Kormana (`Korman`)

Ostatnią analizowaną heurystyką bazową jest średnia ważona pozostałych heurystyk znaleziona przez Michaela Kormana w 2004 roku [36]. Aby taka kombinacja miała sens, wszystkie heurystyki składowe muszą być znormalizowane do wspólnego przedziału. W oryginalnej pracy Kormana był to przedział $[-100; 100]$, ale ja zdecydowałem się na, moim zdaniem, naturalniejsze $[-1; 1]$. Przynależność wartości każdej heurystyki do tego przedziału została również potwierdzona testami jednostkowymi. Wagi znalezione przez Kormana to, w zaokrągleniu:

$$802 \cdot \text{CornersOwned} + 382 \cdot \text{CornerCloseness} + 79 \cdot \text{CurrentMobility} +$$
$$10 \cdot \text{MaximumDisc} + 26 \cdot \text{Weighted} + 74 \cdot \text{PotentialMobility} + 100 \cdot \text{Stability}$$

3.4 Ocena skuteczności heurystyk

W celu oceny skuteczności heurystyk, zorganizowano aproksymację pozycji rankingowej metodą Monte Carlo. Idea metody jest następująca: rozgrywając mecze pomiędzy graczami korzystającymi z dwóch heurystyk, na mocy centralnego twierdzenia granicznego, z czasem obliczony przez nas wynik będzie się zbliżał do rzeczywistego.

Do oceny jakości heurystyki wykorzystałem punktację Elo, wykorzystywaną kiedyś do obliczania relatywnej siły gry szachistów [37]. Od tej pory została ona zastąpiona rankingiem Glicko, ale na potrzeby zadania uproszczona wersja powinna być wystarczająca. Taki system posiada wiele zalet względem zwykłego obliczania procentu wygranych. Bierze on pod uwagę ranking obu uczestników meczu, przyznawane jest wiele punktów za wygraną przeciw mocniejszemu od nas przeciwnikowi, a mało za wygraną przeciw znacznie słabszemu oponentowi. Analogicznie, przegrana przeciw słabszemu przeciwnikowi znacznie obniża nasz ranking. Wadą systemu jest to, że stanowi on miarę relatywną, a nie absolutną, więc o danej wartości MMR ma sens mówić jedynie w kontekście jednej grupy graczy.

W tym celu powstała metoda `run_tournament`, która rozgrywa mecze pomiędzy graczami z puli na wielu wątkach i wysyła poprzez kanał ich wyniki, aż do minięcia określonego czasu. Wątek główny natomiast zbiera napływające z innych rezultaty meczy i odpowiednio aktualizuje procenty zwycięstw i wartości MMR.

Wyniki oceny skuteczności algorytmów znajdują się poniżej wraz z odpowiednimi wnioskami.

```
NAIVE STRATEGIES
Played 160245 games!
 1.       CornersGreedy       1074 MMR, 71.9% WR
 2.         FirstMove         1041 MMR, 44.6% WR
 3.        ScoreGreedy         985 MMR, 48.5% WR
 4.        RandomMove          900 MMR, 35.0% WR
```

Rysunek 13: Porównanie naiwnych strategii.

Wśród naiwnych strategii, przedstawionych na Rys. 13, zdecydowanie najlepszą jest zachłanne przejmowanie pól narożnych. Tłumaczy to również wysokie wagi komponentów związanych z polami narożnymi w średniej wykorzystanej przez Kormana. Najgorsze wyniki osiągnęła strategia wykonująca losowe ruchy, natomiast bardzo słabo wypadło również zachłanne wykonywanie ruchów przejmujących najwięcej pionów.

```
MINIMAX VS ALPHA-BETA
Played 12039 games!
 1.        MM(MaxD, 3)        1068 MMR, 50.2% WR
 2.        αβ(MaxD, 3)         932 MMR, 49.8% WR
```

Rysunek 14: Wpływ cięcia alfa–beta na siłę algorytmu.

Rys.14 przedstawia turniej, którego celem było udowodnienie, że dodanie cięcia alfa–beta nie zmienia wyników algorytmu, a jedynie go przyspiesza. Procent wygranych obu strategii jest praktycznie identyczny, w granicy błędu pomiarowego. W związku z tym, w kolejnych zestawieniach każdy gracz będzie wykorzystywał cięcia alfa–beta.

```
MAX DEPTH COMPARISON
Played 126 games!
 1.       αβ(KORMAN, 5)       1272 MMR, 97.8% WR
 2.       αβ(KORMAN, 4)       1159 MMR, 78.4% WR
 3.       αβ(KORMAN, 3)        941 MMR, 40.9% WR
 4.       αβ(KORMAN, 2)        921 MMR, 36.2% WR
 5.       αβ(KORMAN, 1)        707 MMR,  3.8% WR
```

Rysunek 15: Wpływ głębokości na siłę algorytmu.

Rys. 15 pokazuje wpływ maksymalnej głębokości rekurencji na osiągi algorytmu. Jak widać każdy kolejny przetworzony poziom ogromnie zwiększa siłę algorytmu.

```
WEIGHT MATRIX COMPARISON
Played 1107 games!
 1.      αβ(W(KORMAN), 4)      1083 MMR, 69.8% WR
 2.  αβ(W(VAISHU&MUTHU), 4)    1056 MMR, 55.3% WR
 3.       αβ(W(MAGGS), 4)       861 MMR, 24.6% WR
```

Rysunek 16: Porównanie różnych macierzy wag.

Najlepszego doboru wag w heurystyce ważonych pól dokonał zgodnie z Rys. 16 Michael Korman, a najlepsze wyniki osiągnęła najstarsza macierz, ta z 1979 roku.

```
BASIC HEURISTICS
Played 521 games!
 1.        αβ(CrCls, 3)        1183 MMR, 69.5% WR
 2.         αβ(Stab, 3)        1155 MMR, 74.5% WR
 3.        αβ(CurMob, 3)       1134 MMR, 54.1% WR
 4.        αβ(EdStab, 3)       1111 MMR, 67.5% WR
 5.        αβ(CrOwn, 3)        1046 MMR, 59.8% WR
 6.        αβ(PotMob, 3)        910 MMR, 42.5% WR
 7.        αβ(InStab, 3)        890 MMR, 38.1% WR
 8.         αβ(MaxD, 3)         854 MMR, 34.8% WR
 9.         αβ(MinD, 3)         717 MMR, 12.5% WR
```

Rysunek 17: Porównanie podstawowych heurystyk.

Zestawienie wszystkich elementarnych heurystyk przedstawia Rys. 17. Potwierdza on, że maksymalizowanie ilości przejętych pionów jest bardzo słabą strategią, pokonującą jedynie przeciwną sobie `MinimumDisc`. Najlepsze wyniki osiągają heurystyki związane ze stabilnością oraz polami narożnymi.

```
FULL TOURNAMENT
Played 4474 games!
 1.        αβ(KORMAN, 4)       1491 MMR, 94.6% WR
 2.     αβ(W(KORMAN), 4)       1256 MMR, 74.5% WR
 3.         αβ(Stab, 4)        1063 MMR, 57.0% WR
 4.        αβ(CrCls, 4)        1009 MMR, 50.3% WR
 5.        αβ(CurMob, 4)        964 MMR, 48.6% WR
 6.        CornersGreedy        664 MMR, 19.5% WR
 7.         RandomMove          553 MMR,  7.2% WR
```

Rysunek 18: Wyniki pełnego turnieju.

Rys. 18 przedstawia wyniki turnieju rozegranego między najlepszymi graczami z poszczególnych poprzednich grup. Konkurencję zdominowała heurystyka stanowiąca średnią ważoną, która wygrała prawie 95% swoich meczy. Zgodnie z oczekiwaniami, każda strategia pokonała losowe stawianie ruchów. Gdybyśmy byli zmuszeni do wykorzystanie tylko jednej heurystyki bazowej, najlepszym wyborem jest metoda ważonych pól.

3.5 Implementacja interfejsu tekstowego

Ostatnim krokiem podstawowej wersji programu było zaimplementowanie interfejsu wiersza poleceń do rozpatrywania podanej przez użytkownika pozycji.

Program rozpoczyna od wczytania ze standardowego wejścia planszy podanej przez gracza, oraz konstrukcji obiektów abstrakcyjnej strategii na podstawie argumentów przy wywołaniu programu. Stan gry jest weryfikowany, o ile jest to możliwe. Następnie, gracze wykonują ruchy w pętli, korzystając ze swoich strategii. Czas wykonywania całej rozgrywki jest rejestrowany i na końcu wypisany użytkownikowi.

Na standardowym wyjściu prezentowany jest stan końcowy gry, a na wyjściu błędów ilość odwiedzonych węzłów drzewa i czas obliczeń. Działanie programu można dostosować za pomocą argumentów przedstawionych na Rys. 20, a przykładowe wywołanie programu pokazuje Rys. 19.



Rysunek 19: Przykład wywołania programu `solve`.

```
Usage: solve.exe [OPTIONS]

Options:
  -i, --no-initial       Don't print the initial info
  -v, --no-verification  Don't verify the given game state
  -h, --help             Print help

Player 1:
      --bh <BLACK_HEURISTIC>  Heuristic for player 1 [default: korman]
      --bd <BLACK_DEPTH>      Max recursion depth for player 1 in range 1..=10 [default: 5]
      --bm                    Disable alpha-beta pruning for player 1 (use pure Minmax)

Player 2:
      --wh <WHITE_HEURISTIC>  Heuristic for player 2 [default: korman]
      --wd <WHITE_DEPTH>      Max recursion depth for player 2 in range 1..=10 [default: 5]
      --wm                    Disable alpha-beta pruning for player 2 (use pure Minmax)

Available heuristics:
- max-disc                    - min-disc                    - w-maggs
- w-sannid                    - w-korman                    - corn-own
- corn-close                  - cur-mob                     - pot-mob
- int-stab                    - edge-stab                   - stab
- iago                        - korman                      - le051
- le064                       - le148                       - le162
- le215
```

Rysunek 20: Argumenty dostępne w programie `solve`.

4 Zadanie dodatkowe

„5. modyfikacja programu tak, by wykonywał tylko jeden ruch (z punktu widzenia gracza, którego jest kolej) co umożliwi rozegranie partii pomiędzy dwoma programami (np. dwoma wywołaniami tej samej aplikacji) oraz zastosowanie heurystyk adaptacyjnie zmieniających strategię gracza w celu osiągnięcia zwycięstwa”

4.1 Modyfikacja programu [Z5]

Modyfikacja programu była trywialna, polegała na usunięciu połowy argumentów programu (tak, aby odnosił się tylko do jednego gracza), natomiast dodanie dodatkowego argumentu wskazującego, jako który gracz ma wykonywać ruchy gracz. Argumenty nowego programu pokazuje Rys. 21.

```
Usage: step.exe [OPTIONS] --player <PLAYER>

Options:
  -p, --player <PLAYER>        Player, for which turns should be played [possible values: p1, p2]
  -h, --heuristic <HEURISTIC>  Heuristic function for the player [default: korman]
  -d, --depth <DEPTH>          Max recursion depth for the player [default: 5]
  -n, --no-pruning             Disable alpha-beta pruning for the player
      --help                   Print help

Available heuristics:
- max-disc                    - min-disc                    - w-maggs
- w-sannid                    - w-korman                    - corn-own
- corn-close                  - cur-mob                     - pot-mob
- int-stab                    - edge-stab                   - stab
- iago                        - korman                      - le051
- le064                       - le148                       - le162
- le215
```

Rysunek 21: Argumenty dostępne w programie `step`.

Program komunikuje się z drugim (lub człowiekiem) za pomocą strumieni standardowego wejścia i wyjścia. Ruchy, które stawia komputer są wypisywane na standardowym wyjściu, co pozwala drugiej instancji wczytać je i zsynchronizować swoją lokalną wersję obiektu `GameState`. Plansza jest wypisywana na standardowe wyjście błędów, aby pokazać obserwatorowi stan rozgrywki jednocześnie bez zakłócania komunikacji.

Największym wyzwaniem było uruchomienie dwóch programów tak, aby sprzężyć ich standardowe wejścia i wyjścia razem w systemie Windows. Finalnie rozwiązanie wykorzystuje tymczasowe pliki, służące do komunikacji obustronnej. Przygotowanie takiej komunikacji przedstawia Lis. 14. Oczywiście w przypadku gry komputera ze zwykłym graczem wystarczy uruchomić program `step.exe`.

```
cargo build --release -p game-theory

cd target/release
copy nul 1to2.txt >nul
copy nul 2to1.txt >nul
cls
step.exe -p p2 <1to2.txt >>2to1.txt | step.exe -p p1 <2to1.txt >>1to2.txt
del 1to2.txt 2to1.txt
cd ../..
```

Listing 14: Uruchomienie dwóch sprzężonych instancji programu.

Fragment rozgrywki między dwoma komputerami przedstawia Rys. 22. Niestety system Windows usuwa kolorowanie tekstu z przekierowanych wyjść.

```
22210222
22222222
21222212
22122212
22212212
21211212
21122112
22222222
Move number: 60 | Turn: White | Score: B 15-48 W | Winner: -----

22222222
22222222
21222212
22122212
22212212
21211212
21122112
22222222
Move number: -- | Turn: ----- | Score: B 14-50 W | Winner: White
```

Rysunek 22: Fragment komunikacji dwóch programów `step`.

4.2 Heurystyki adaptacyjne [Z5]

Zwyczajne heurystyki zwracają liczbę, będącą ewaluacją stanu gry na podstawie instancji stanu gry. Tzn. mają sygnaturę `Fn(GameState) -> f64`. Heurystyki adaptacyjne korzystają z różnych funkcji adaptacyjnych w zależności od stanu gry, można je rozumieć zatem jako funkcje o sygnaturze `Fn(GameState) -> (Fn(GameState) -> f64)`. W najprostszym wariancie, możemy uzależnić wagi poszczególnych heurystyk składowych w zależności od numeru ruchu. Jedną z takich heurystyk wykorzystuje program Iago, stworzony przez Rosenblooma w 1982 roku.

Jego funkcja ewaluacyjna ma postać [32]:

$$\begin{aligned}\mathrm{ESAC}(n) \cdot \mathrm{EdgeStability} + 36 \cdot \mathrm{InternalStability} + \\ \mathrm{CMAC}(n) \cdot \mathrm{CurrentMobility} + 99 \cdot \mathrm{PotentialMobility}\end{aligned}$$

Gdzie funkcje `ESAC` i `CMAC` są funkcjami rosnącymi zależnymi od numeru ruchu. Implementacja tej heurystyki była trywialna, biorąc pod uwagę, że jej komponenty były już zadeklarowane.

Pomysł ten można zgeneralizować tworząc heurystykę stanowiącą średnią ważoną podstawowych heurystyk, gdzie każda waga jest funkcją, dla uproszczenia załóżmy, że liniową ewaluowaną numerem ruchu. Jako heurystyki bazowe wybrałem w tym przypadku `MaximumDisc`, `CornersOwned`, `CornerCloseness`, `CurrentMobility`, `PotentialMobility`, `InternalStability`, `EdgeStability`. Nie ma sensu uwzględnianie `MinimumDisc` oraz `Stability`, ponieważ są one liniowo zależne od odpowiednio `MaximumDisc` oraz `InternalStability` i `EdgeStability`. Ważenie pól zostało pominięte, ponieważ wtedy pojawia się dodatkowo problem wyznaczenia macierzy wag. Definicja takiej heurystyki znajduje się na Lis. 15.

Każdej instancji takiej heurystyki nadajemy trzycyfrową liczbę, stanowiącą skrót jej hashu, w celu ułatwienia identyfikacji poszczególnych tablic wag. Nie można tej liczby stosować jako miary podobieństwa dwóch tablic, ponieważ są one efektywnie pseudolosowe.

```
pub const LINEAR_WEIGHT_LEN: usize = 14;
const LINEAR_COMPONENTS: [Heuristic; LINEAR_WEIGHT_LEN / 2] = [
    Heuristic::MaximumDisc,
    Heuristic::CornersOwned,
    Heuristic::CornerCloseness,
    Heuristic::CurrentMobility,
    Heuristic::PotentialMobility,
    Heuristic::InternalStability,
    Heuristic::EdgeStability,
];

pub enum Heuristic {
    /* Other heuristics... */

    LinearEquations(Box<[f64; LINEAR_WEIGHT_LEN]>)
}
```

Listing 15: Definicja heurystyki adaptacyjnej.

Ostatnim problemem pozostaje dobranie odpowiedniego zestawu wag. W tym celu, inspirując się rozwiązaniem zawartym na wykładzie [3], został zastosowany algorytm genetyczny.

Jako chromosom wybieramy tablicę 14 wag, początkowo wylosowanych z zakresu od -1 do 1. W każdej generacji rozgrywamy trwający minutę turniej, celem wyznaczenia punktów rankingowych Elo wśród populacji. Zostają one wykorzystane jako wartość *fitness* danych chromosomów. Kolejną generację tworzymy z trzech części: 25% stanowi 25% najlepszych chromosomów z poprzedniej generacji na mocy zasady elitaryzmu, 10% jest dobierane losowo, aby testować także wydajność algorytmu przeciw słabym przeciwnikom, oraz aby zapobiec utykaniu optymalizacji w ekstremach lokalnych. Pozostałe 65% populacji stanowią osobniki utworzone poprzez krzyżowanie i mutację dwóch rodziców wybranych drogą selekcji proporcjonalnej. Każde dziecko bierze losową część z każdego z rodziców, a następnie modyfikuje wartość genu o ± 0.1. Na końcu każdej generacji zostaje wypisanie pięć najlepszych osobników, wraz z ich rankingiem i tablicą wag.

W rezultacie godzinnego treningu (60 generacji) uzyskano osobniki przedstawione na Lis. 16.

```
1. LinEq(064), 1155 MMR
   = [-0.088,-1.444,0.029,0.857,1.546,1.019,0.088,0.985,0.064,0.307,...
2. LinEq(215), 1150 MMR
   = [-0.192,0.677,0.495,0.147,1.286,-0.197,0.545,0.307,0.128,0.666,...
3. LinEq(162), 1118 MMR
   = [-0.250,-1.468,1.044,0.828,1.556,-1.389,0.625,0.700,0.023,0.347,...
4. LinEq(051), 1117 MMR
   = [-0.064,-0.065,0.106,0.746,0.928,0.101,0.183,0.183,0.427,1.047,...
5. LinEq(148), 1115 MMR
   = [-0.198,0.553,0.755,0.602,0.345,-1.263,0.445,0.253,0.305,0.659,...
```

Listing 16: Najlepsze osobniki po godzinie treningu algorytmu genetycznego.

Wartości punktów rankingowych są pozornie niskie, ale trzeba zauważyć, że system Elo, jak zostało wcześniej wspomniane ocenia relatywną, a nie absolutną siłę graczy. W związku z tym, pokazane wartości są rankingiem tych chromosomów wśród całej (długo trenowanej) populacji 60, a nie wśród wszystkich graczy.

Uzyskane tablice wag wykorzystujemy definiując odpowiednio heurystyki `lineq051`, `lineq064`, `lineq148`, `lineq162`, `lineq215`. Zostały one też dodane do programów `solve` oraz `step`.

Jako finalną ewaluację siły heurystyk został rozegrany ostateczny turniej, z udziałem Iago oraz `lineqXXX`. Wyniki zostały przedstawione na Rys. 23.

```
FULL TOURNAMENT
Played 1624 games!
 1.     αβ(LinEq(118), 4)     1393 MMR, 76.7% WR
 2.     αβ(LinEq(064), 4)     1302 MMR, 74.2% WR
 3.     αβ(LinEq(164), 4)     1239 MMR, 74.2% WR
 4.       αβ(KORMAN, 4)       1220 MMR, 71.8% WR
 5.     αβ(LinEq(165), 4)     1206 MMR, 73.7% WR
 6.        αβ(IAGO, 4)        1189 MMR, 70.9% WR
 7.     αβ(LinEq(223), 4)     1175 MMR, 74.4% WR
 8.     αβ(W(KORMAN), 4)       990 MMR, 46.0% WR
 9.        αβ(Stab, 4)         820 MMR, 30.6% WR
10.       αβ(CurMob, 4)        753 MMR, 25.3% WR
11.       αβ(CrCls, 4)         693 MMR, 21.8% WR
12.       CornersGreedy        548 MMR, 10.7% WR
13.        RandomMove          472 MMR,  4.1% WR
```

Rysunek 23: Wyniki ostatecznego turnieju.

Trzy z pięciu algorytmów adaptacyjnych pokonały całą konkurencję, cztery z pięciu były lepsze od Iago, a każdy z nich był lepszy od podstawowych heurystyk. Co ciekawe, heurystyka Kormana osiągnęła lepsze wyniki od Iago, pomimo że była statyczna, a nie adaptacyjna. Jednakże powstała prawie 20 lat później, więc nie jest to też szokujące.

5 Podsumowanie

5.1 Dalsze badania

Omawiany temat jest bardzo szeroki i istnieje wiele możliwości rozwoju projektu, które wykraczały poza zakres zadania lub były zbyt czasochłonne, są to między innymi:

- Zastosowanie tablicy transpozycyjnej, zapamiętującej wartości heurystyk dla stanów w celu ich ponownego użycia przy natrafieniu na ten sam stan. Niestety w przeciwieństwie do szachów lub warcabów, dokonanie ruchów $A \to B$ nie daje takiego samego stanu co $B \to A$, przez co duplikatowe stany występują znacznie rzadziej. Według źródeł taka optymalizacja może zaoszczędzić około 20% czasu wykonywania algorytmu [23], w moich próbach była to wartość bliższa 10%.

- Zastosowanie tzw. *opening book*, biorąc pod uwagę jak mało istnieje stanów dla wczesnych numerów ruchów, można obliczyć raz, które z nich są korzystne a które nie i odwoływać się do tej „księgi" przy wykonywaniu ruchów początkowych. Jest to metoda wykorzystywana w każdym współczesnym programie do gry w Othello, ale jej implementacja ma dość małą wartość dydaktyczną.

- Zastosowanie *endgame table* – zamysł jest identyczny co opening book, tylko dotyczy stanów końcowych, a nie początkowych.

- Optymalizacja weryfikacji stanów wykorzystując podejście „meet in the middle", co skróciłoby czas jej wykonywania dwukrotnie. Niestety czas wykonywania weryfikacji rośnie wykładniczo, więc skrócenie go o połowę, wbrew pozorom, nie byłoby znacznie pomocne.

- Zastosowanie iteracyjnego pogłębiania – zamiast schodzić do stałej głębokości, zaczynając od głębokości 1 próbujemy schodzić coraz głębiej aż do osiągnięcia limitu czasu. Biorąc pod uwagę, że czas wykonywania algorytmu rośnie wykładniczo z poziomem drzewa, suma czasów przeszukiwań poziomów $0..n-1$ nie powinna przekroczyć czasu przeszukiwania poziomu n, więc nie tracimy w ten sposób zbyt wiele czasu.

- Sortowanie ruchów – jest to metoda znacznie przyspieszająca algorytm alfa–beta, ale nie zaimplementowałem jej, ponieważ jest bardzo specyficzna. Optymalizując prędkość jednej heurystyki moglibyśmy spowolnić inną (oczywistym przykładem jest `MaximumDisc` i `MinimumDisc`, sortując ruchy uzyskalibyśmy najbardziej optymalną prędkość wykonywania dla jednego, a jednocześnie najbardziej pesymistyczną dla drugiego). Nam natomiast zależy na zachowaniu jak największej generalizacji, chociażby dlatego, że wybór heurystyki zostawiamy użytkownikowi w przeciwieństwie do tradycyjnych programów grających w Othello.

- Wykorzystanie lepszych algorytmów adaptacyjnych, w tym między innymi:

  - Wykorzystanie innych współczynników krzyżowania i mutacji dla części liniowych i wyrazów wolnych – zmiana wyrazu wolnego o 1 ma o wiele większy wpływ na formę funkcji niż zmiana nachylenia funkcji o 1.
  - Użycie bardziej zaawansowanej funkcji niż liniowa – mobilność teoretycznie jest najistotniejsza w środkowych fazach gry, nie da się tego obecnie zareprezentować formą `lineqXXX`, można tylko przedstawić wzrost, spadek, lub stała istotność parametru w czasie.
  - Bazowanie funkcji ewaluacyjnej na większej części stanu początkowego, a nie tylko numerze ruchu, np. wykorzystując sieci neuronowe przyjmujące na wejściu stany każdego pola i zwracające wagi heurystyk dla takiej sytuacji.

5.2 Wnioski

Problem konstrukcji programu grającego w Reversi jest pozornie łatwy, ale wymaga analizy wielu skrajnych i rzadko występujących przypadków. W zapewnieniu odpowiedniej obsługi sytuacji nietypowych pomagają testy jednostkowe i integracyjne.

Zadania wykorzystują wiele idei poruszonych na poprzedniej liście. Drzewo gry, które przeglądamy jest specjalnym przypadkiem grafu, a algorytm Minimax zmodyfikowanym przeglądem DFS. Obrazuje to istotność teorii grafów w tworzeniu algorytmów sztucznej inteligencji.

Cięcie alfa–beta jest bardzo dobrą optymalizacją algorytmu Minimax, a jego implementacja jest na tyle prosta, że nie jestem w stanie zauważyć żadnej sytuacji, w której wartoby z niej nie skorzystać. W przypadku Reversi, cięcia okazały się najbardziej pomocne w środkowych fazach rozgrywki, ponieważ to wtedy przeszukujemy najwięcej węzłów drzewa gry.

Korzystając z algorytmów genetycznych można w krótkim czasie wytrenować heurystyki silniejsze od programów uznawanych kilkanaście lat temu za najmocniejsze. Pokazuje to, jak szybko zostały poczynione postępy w dziedzinie sztucznej inteligencji, w tym szczególności w rozwiązywaniu gier logicznych. Współczesne implementacje algorytmów sięgających lat 80. są w stanie działać wiele razy szybciej, a także są łatwiejsze w implementacji, na przykład dzięki możliwości skorzystania z liczb zmiennoprzecinkowych, co w tamtych czasach było niewystarczająco optymalne i często niedostępne.

Słownik pojęć

W związku z wysoką ilością występujących w pracach naukowych dotyczących Reversi definicji, często kolidujących ze sobą oraz nieposiadających oficjalnego polskiego tłumaczenia, poniżej znajduje się spis wybranych, jednolitych i użytych w implementacji pojęć, w wariancie polskim (wykorzystanym w sprawozdaniu) oraz angielskim (wykorzystanym w kodzie):

brzegowe (`edge`) **pola** wszystkie pola, leżące w kolumnach `A` i `H` lub rzędach `1` i `8`

centralne (`center`) **pola** cztery pola znajdujące się na środku planszy (D4, D5, E4, E5)

flankowanie (`outflanking`) otoczenie pionów gracza przeciwnego ułożonych w linii poziomej, pionowej lub ukośniej z obu stron pionami obecnego gracza, powoduje odwrócenie pionów gracza przeciwnego

narożne (`corner`) **pole** pola znajdujące się w rogach planszy `A1`, `A8`, `H1`, `H8`

niezweryfikowany (`unverified`) **stan gry** stan, który mógł potencjalnie być osiągnięty w zwykłej rozgrywce, ale nie mamy co do tego pewności

notacja (`notation`) odwołanie do konkretnego pola za pomocą symboli od `A1` do `H8`

numer ruchu (`move number`) liczba pomiędzy 1 a 60 włącznie, określająca ile tur zostało rozegranych na planszy, zawsze o 3 mniejsza od liczby pionów na planszy

odwrócenie (`flip`) zamiana koloru danego piona na przeciwny

pierwotny (`original`) **pion** pion, który nie miał prawa się obrócić od czasu jego postawienia

pion (`disc`) stawiany przez gracza na planszę przedmiot, nazywany czasami krążkiem

plansza (`board`) zbiór pól w układzie 8×8, na których rozgrywana jest gra

pole (`square`) jedno z 64 miejsc na planszy, na których stawiane są piony

puste (`empty`) **pole** pole, na którym nie znajduje się żaden pion

stabilny (`stable`) **pion** pion, który nie może zostać już obrócony do końca gry

stan gry (`game state`) krotka (plansza, gracz) identyfikująca stan gry

transkrypt (`transcript`) zapis notacji kolejnych wykonanych w trakcie gry ruchów

wewnętrzne (`internal`) **pola** pola, które nie są brzegowe

zajęte (`occupied`) **pole** pole, na którym znajduje się dowolny pion

zweryfikowany (`verified`) **stan gry** stan, do którego na pewno można dotrzeć wykonując same legalne ruchy od pozycji początkowej

Literatura

[1] Instrukcja do laboratorium 2 – Jakubik, J., Syga, P.

[2] Rozwiązania list zadań – tchojnacki [dostęp: 11.05.2023], `https://github.com/tchojnacki/uni-sem-6-ai`

[3] Wykład 3: Gry logiczne – Kwaśnicka, H., Piasecki, M.

[4] Reversi – Wikipedia [dostęp: 03.05.2023], `https://en.wikipedia.org/wiki/Reversi`

[5] Rules and Instructions for Reversi – Masters Traditional Games [dostęp: 04.05.2023], `https://www.mastersofgames.com/rules/reversi-othello-rules.htm`

[6] Reversi and Othello – two different games. – Ludo, B. [dostęp: 04.05.2023], `https://bonaludo.com/2016/02/18/reversi-and-othello-...`

[7] Reversi versus Othello – World Othello Federation [dostęp: 04.05.2023], `https://www.worldothello.org/news/47`

[8] WOC rules – World Othello Federation [dostęp: 04.05.2023], `https://www.worldothello.org/about/tournaments/...`

[9] Reversi Rules – Reversi Documentation [dostęp: 04.05.2023], `https://documentation.help/Reversi-Rules/rules.htm`

[10] Computer Othello – Wikipedia [dostęp: 05.05.2023], `https://en.wikipedia.org/wiki/Computer_Othello`

[11] The evolution of strong Othello programs – Buro, M. [dostęp: 05.05.2023], `https://skatgame.net/mburo/ps/compoth.pdf`

[12] `clap` – crates.io [dostęp: 28.04.2023], `https://crates.io/crates/clap`

[13] `colored` – crates.io [dostęp: 28.04.2023], `https://crates.io/crates/colored`

[14] `once_cell` – crates.io [dostęp: 28.04.2023], `https://crates.io/crates/once_cell`

[15] `rand` – crates.io [dostęp: 28.04.2023], `https://crates.io/crates/rand`

[16] Rekord z wariantami – Wikipedia [dostęp: 28.04.2023], `https://pl.wikipedia.org/wiki/Rekord_z_wariantami`

[17] Zero Cost Abstractions – The Embedded Rust Book [dostęp: 28.04.2023], `https://doc.rust-lang.org/beta/embedded-book/static-guarantees/...`

[18] Problem of the Month (November 2012) – Math Magic [dostęp: 28.04.2023], `https://erich-friedman.github.io/mathmagic/1112.html`

[19] Branching factor – AI For Anyone [dostęp: 04.05.2023], `https://www.aiforanyone.org/glossary/branching-factor`

[20] Reversi/Othello – Given a certain game state ([...]) is it possible to find a set of moves that would satisfy it? – Stack Overflow [dostęp: 04.05.2023], https://stackoverflow.com/questions/75639842

[21] Othello/Reversi: Given the board state of an ongoing game, is it possible to programatically determine the moves the players have made? – Stack Exchange [dostęp: 04.05.2023], https://boardgames.stackexchange.com/questions/58476

[22] Cecha (programowanie obiektowe) – Wikipedia [dostęp: 05.05.2023], https://pl.wikipedia.org/wiki/Cecha_(programowanie_obiektowe)

[23] Writing an Othello program – Andersson, G. [dostęp: 05.05.2023], http://www.radagast.se/othello/howto.html

[24] Property–based Testing With QuickCheck – Typeable [dostęp: 05.05.2023], https://typeable.io/blog/2021-08-09-pbt.html

[25] quicktest – crates.io [dostęp: 05.05.2023], https://crates.io/crates/quicktest

[26] The game Reversi – Thesaurus [dostęp: 06.05.2023], https://thesaurus.altervista.org/revers-game

[27] Bitboards – Chess Programming Wiki [dostęp: 06.05.2023], https://chessprogramming.org/Bitboards

[28] Dumb7Fill – Chess Programming Wiki [dostęp: 06.05.2023], https://chessprogramming.org/Dumb7Fill

[29] Minimax – Roberts, E. [dostęp: 06.05.2023], https://cs.stanford.edu/people/eroberts/courses/soco/...

[30] Minimax – Wikipedia [dostęp: 06.05.2023], https://en.wikipedia.org/wiki/Minimax

[31] Minimax and alpha-beta pruning – Lague, S. [dostęp: 06.05.2023], https://www.youtube.com/watch?v=l-hh51ncgDI

[32] A World Championship–Level Othello Program – Rosenbloom, P. [dostęp: 24.04.2023], https://stacks.stanford.edu/file/druid:wk764yw7162/wk764yw7162.pdf

[33] Fundamental Othello misconceptions: Disc–counting strategies, Stringham, G.

[34] Programming Strategies in the Game of Reversi – Maggs, P. [dostęp: 25.04.2023], https://archive.org/details/byte-magazine-1979-11/page/n71/mode/2up

[35] Analysis of Heuristics in Othello – Vaishnavi, Muthukaruppan [dostęp: 26.04.2023], https://courses.cs.washington.edu/courses/cse573/04au/Project/...

[36] Playing Othello with Artificial Intelligence – Korman, M. [dostęp: 26.04.2023], https://www.semanticscholar.org/paper/Playing-Othello-with-Artificial-...

[37] System rankingowy Elo – Chess.com [dostęp: 05.05.2023], https://www.chess.com/pl/terms/system-rankingowy-elo

Sztuczna inteligencja i inżynieria wiedzy
K01-49i, czwartek 11:15

prowadzący
mgr inż. Katarzyna Fojcik



Politechnika
Wrocławska

Wnioskowanie i symboliczna reprezentacja wiedzy

Lista 3

Tomasz Chojnacki
260365@student.pwr.edu.pl

WROCŁAW 25 maja 2023

Spis treści

1 Wstęp **2**

2 Sieć semantyczna **2**

3 Przykłady unifikacji i wnioskowania **4**

4 Definicje reguł diagnostycznych **5**

5 Interfejs tekstowy **6**

6 Rozwiązywanie problemów **7**

7 Przykładowe wywołania programu **8**

8 Wnioski **9**

1 Wstęp

Celem listy trzeciej było zapoznanie z wnioskowaniem i symboliczną reprezentacją wiedzy w środowisku Prolog. W tym celu należało zbudować bazę wiedzy opisującą wybrane urządzenie oraz na jej podstawie zbudować prosty program wspierający użytkownika rozwiązywaniu problemów [3].

Najistotniejsze części implementacji zostały zawarte w sprawozdaniu, wraz z opisami, natomiast pełna implementacja jest dostępna w repozytorium [2]. W zadaniu wykorzystałem (najpopularniejszą) dystrybucję SWI Prolog [4], może ona zawierać wbudowane predykaty, które są niedostępne w innych wersjach środowiska.

2 Sieć semantyczna

Urządzenie, które wybrałem do analizy to drukarka HP Photosmart 420, którego instrukcja obsługi jest dostępna w internecie [5]. Instrukcja ta zawiera dość długą i szczegółową sekcję *troubleshooting*.

Niestety, większość metod rozwiązywania problemów jest bardzo prosta, zaskakujące było dla mnie, jak wiele problemów producent proponuje rozwiązać poprzez wyłączenie i włączenie urządzenia. Po przejrzeniu innych instrukcji doszedłem do wniosku, że drukarka ta nie stanowi wyjątku – sekcje rozwiązywania problemów nie są skomplikowane i pozwalają wykluczyć tylko oczywiste błędy, a przy poważniejszych awariach zalecany jest zazwyczaj telefon na infolinię. W związku z tym zdecydowałem się na rozbudowanie pozostałych sekcji programu, wiedząc z góry, że logika rozwiązywania problemów nie będzie zaawansowana.

Pierwszym krokiem było zdefiniowanie sieci semantycznej urządzenia, czyli jego części i powiązania między nimi. Części definiuję w kodzie jako atomy spełniające predykat `def_part`. Atrybutami części są: atom, który ją identyfikuje (`Part`), czytelny opis części w formie łańcucha tekstowego (`Label`), listę tagów urządzenia, określające np. z jakim rodzajem części mamy do czynienia i co można z nią zrobić, np. wcisnąć, zapalić, itd. (`Tags`) oraz opcjonalny „rodzic" części, czyli moduł, w którym jest umiejscowiona definiowana część (`Parent`). Przykładowe definicje tego typu pokazuje Lis. 1.

```
def_part(printer, "Printer", [], null).
def_part(internal_battery, "Internal Battery", [battery], printer).
def_part(camera, "Camera", [], printer).
def_part(camera_battery, "Camera Battery", [battery], camera).
def_part(control_panel, "Control Panel", [panel], printer).
def_part(save_button, "Save Button", [button], control_panel).
def_part(control_button, "Control Button", [button], control_panel).
def_part(rear_panel, "Rear Panel", [panel], printer).
def_part(power_cord, "Power Cord", [port], rear_panel).
def_part(top_panel, "Top Panel", [panel], printer).
def_part(on_light, "On Light", [light], top_panel).
```

Listing 1: Definicja części drukarki.

Możemy zdefiniować ciekawsze predykaty, operujące na `def_part`. Są to:

```
% Predykat sprawdzający czy dany atom jest częścią
is_part(Part) :- def_part(Part, _, _, _).

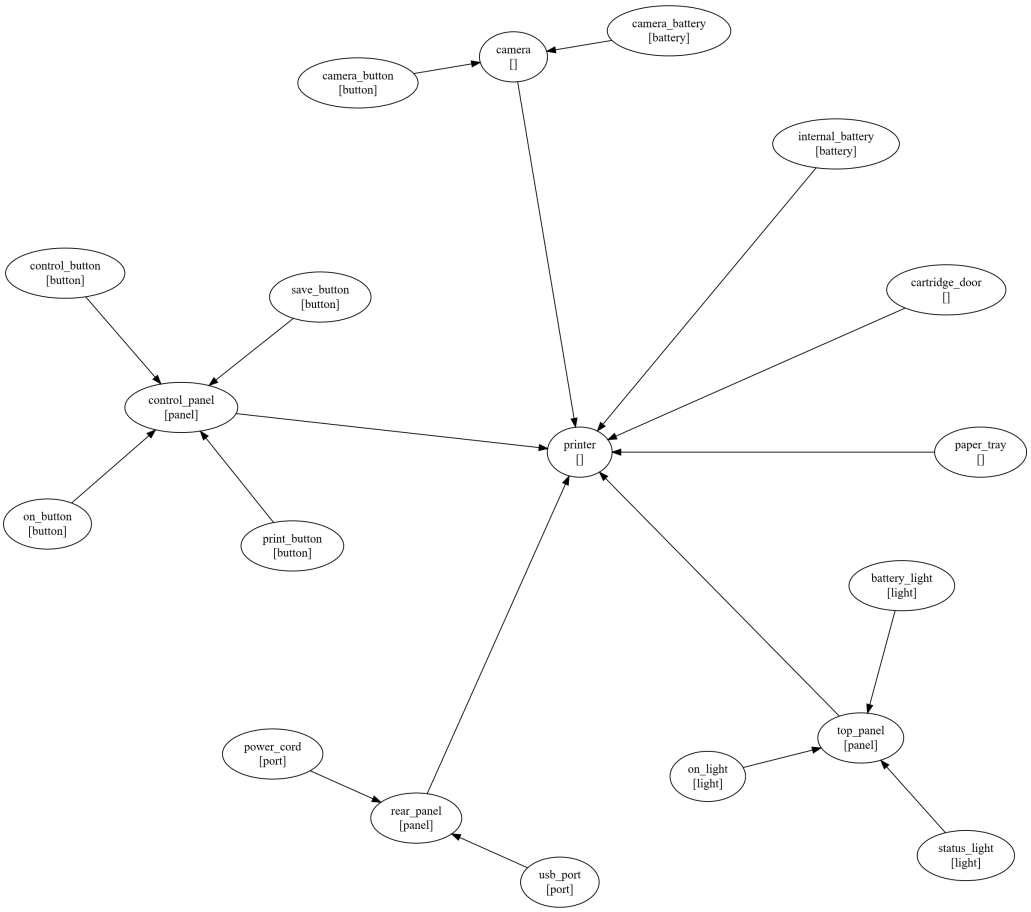
% Wyszukanie rodzica (lub dziecka) części
part_parent(Part, Parent) :-
    is_part(Parent),
    def_part(Part, _, _, Parent).

% Wyszukanie tagów części (lub w drugą stronę)
part_tag(Part, Tag) :-
    def_part(Part, _, Tags, _),
    member(Tag, Tags).

% Rekurencyjne wyszukiwanie części
descendant_part(Part, Part).
descendant_part(Descendant, Part) :-
    part_parent(Descendant, Parent),
    descendant_part(Parent, Part).
```

Listing 2: Predykaty pomocnicze dla części drukarki.

Pełną zdefiniowaną sieć semantyczną pokazuje Rys. 1.



Rysunek 1: Sieć semantyczna drukarki.

3 Przykłady unifikacji i wnioskowania

Na podstawie predykatów zdefiniowanych w poprzednim rozdziale, możemy zrozumieć możliwości środowiska Prolog. Przykładowo, wpisując `is_part(X).`, gdzie `X` stanowi niewiadomą, możemy uzyskać wszystkie wartości, które pasują w tym kontekście (sprawiają że predykat jest prawdziwy), czyli wszystkie części:

```
?- is_part(X).
X = printer ;
X = paper_tray ;
X = cartridge_door ;
X = internal_battery ;
X = camera ;
...
```

Dodatkowe możliwości pokazuje nam predykat `part_parent` – możemy szukać zarówno rodzica części jak i dziecka części, bez definiowania dwóch osobnych funkcji do tych celów:

```
?- part_parent(X, printer).
X = paper_tray ;
X = cartridge_door ;
X = internal_battery ;
X = camera ;
X = control_panel ;
X = rear_panel ;
X = top_panel.

?- part_parent(camera, X).
X = printer ;
```

Analogiczne działanie można pokazać dla `part_tag`. Ciekawszy jest `descendant_part`, który działa rekurencyjnie. Bazowym warunkiem jest `descendant_part(Part, Part).`, czyli każda część jest w zbiorze należących do samego siebie. W wywołaniu rekurencyjnym stwierdzamy, że jeżeli `Descendant` ma rodzica `Parent`, a `Parent` jest potomkiem `Part`, to `Descendant` też jest potomkiem. Kroki wykonywane przez silnik można podejrzeć wykonując przed komendą predykat `trace` [6]. Uproszczone wyjście z takiego wywołania to:

```
?- trace.
true.

[trace] ?- descendant_part(camera, printer).
   Call: descendant_part(camera, printer) ?
   Call: part_parent(camera, _1494) ?
   Call: is_part(_1494) ?
   Call: def_part(_1494, _3186, _3188, _3190) ?
   Call: def_part(camera, _5636, _5638, printer) ?
   Call: descendant_part(printer, printer) ?
true.
```

Prolog najpierw sprawdza, czy warunek bazowy, który nie jest prawdziwy, więc przechodzi do sprawdzania rodzica kamery, w tym celu wywołuje `is_part`, po czym podgląda rodzica z definicji `def_part` i dochodzi do wniosku, że jest nim drukarka. Wtedy wypełnia się warunek bazowy `descendant_part(printer, printer).` i predykat jest spełniony.

4 Definicje reguł diagnostycznych

Na początku definiujemy kategorie problemów, zgodne z tymi zawartymi w instrukcji obsługi. Zostało to dokonane analogicznie do definicji części. Pełna definicja kategorii znajduje się na Lis. 3.

```
def_category(hardware, "Hardware Problems").
def_category(printing, "Printing Problems").
def_category(bluetooth, "Bluetooth Problems").
def_category(errors, "Error Message Shown").

is_category(Category) :- def_category(Category, _).
```

Listing 3: Definicje kategorii problemów.

Podobnie jak w przypadku części definiujemy też predykat `is_category`, który jest spełniony dla atomów będących częściami. Możemy sprawdzić poprawność jego definicji wywołując go z REPL Prologa:

```
?- is_category(X).
X = hardware ;
X = printing ;
X = bluetooth ;
X = errors.
```

Następnie, zgodnie z instrukcją obsługi zdefiniowałem potencjalne problemy, za pomocą predykatu `def_problem`. Przyjmuje on: atom (lub predykat!) identyfikujący problem (`Problem`), czytelny opis problemu (`Label`) oraz kategorię (`Category`). Przykładowe definicje problemów, o różnych stopniach zaawansowania pokazuje Lis. 4.

```
def_problem(no_page_out, "No page came out of the printer.", printing).
def_problem(paper_jammed, "The paper jammed while printing.", printing).

def_problem(flashing(status_light), "The Status light is flashing.", hardware).
def_problem(flashing(on_light), "The On light flashed briefly.", hardware).

def_problem(not_responding(Button), Label, hardware) :-
    part_tag(Button, button),
    label(Button, L),
    string_concat(L, " on the control panel does not respond.", Label).
def_problem(not_charging(Battery), Label, hardware) :-
    part_tag(Battery, battery),
    label(Battery, L),
    string_concat(L, " will not charge.", Label).
```

Listing 4: Przykładowe definicje problemów.

Dwie pierwsze definicje problemu są najprostszym wariantem – wszystkie ich części są statyczne. Następne dwie odwołują się do poszczególnych części. Możemy przykładowo wyszukać wszystkie problemy, w których błyska dowolna kontrolka wpisując `is_problem(flashing(X))`..

Ostatnie dwie definicje są w pełni dynamiczne. Definiują problem niereagującego przycisku i niełądującej się baterii, które będą dostępne **dla każdego** przycisku i baterii, wraz z dynamicznie generowanymi opisami problemu.

Dla problemu definiujemy podobne do poprzednich predykaty pomocnicze:

```
is_problem(Problem) :- def_problem(Problem, _, _).

problem_category(Problem, Category) :-
    is_category(Category),
    def_problem(Problem, _, Category).
```

Listing 5: Predykaty pomocnicze dla problemów.

Podobnie zdefiniowano pozostałe wymienione w instrukcji obsługi problemy.

5 Interfejs tekstowy

Aby pozwolić użytkownikowi na rozwiązanie problemu należało zdefiniować jakiś rodzaj interfejsu do komunikacji z nim. Zdecydowałem się na interfejs tekstowy, w którym użytkownik może przekazywać programowi informacje za pomocą wyboru z listy (w danym zakresie) oraz pytań tak/nie.

Gotowy i dość zaawansowany kod interfejsu można było znaleźć pod jednym z linków umieszczonych w bibliografii listy zadań [7]. Jednakże, uważam, że większą wartość dydaktyczną ma zrobienie własnego, prostszego menu, niż skorzystanie z gotowego rozwiązania. Implementacja mojego interfejsu do komunikacji z użytkownikiem znajduje się na Lis. 6.

```
label(Part, Label) :- def_part(Part, Label, _, _).
label(Category, Label) :- def_category(Category, Label).
label(Problem, Label) :- def_problem(Problem, Label, _).

print_list(List) :- print_list(List, 1).
print_list([], _).
print_list([Head | Tail], N) :-
    write(N), write(". "), write(Head), nl,
    N1 is N + 1, print_list(Tail, N1).

select_menu(Name, List, Result) :-
    write("Choose "), write(Name), write(":"), nl,
    maplist(label, List, Labels), print_list(Labels),
    write("Number: "), read(Number),
    nth1(Number, List, Result), label(Result, ResultLabel),
    write("Selected "), write(Name), write(": "), write(ResultLabel), nl.

ask(Question) :- write(Question), write(" [y/n]: "), read(Ans), is_truthy(Ans).

main :-
    prompt(_, ''),
    findall(C, is_category(C), Categories),
    select_menu("problem category", Categories, Category),
    findall(P, problem_category(P, Category), Problems),
    select_menu("problem", Problems, Problem),
    fix(Problem), nl.
```

Listing 6: Kod odpowiedzialny za interakcję z użytkownikiem.

Implementacja wykorzystuje m.in. operacje na listach, mechanizm dopasowania do wzorca, wbudowane funkcje do operacji na tablicach, predykaty wyższego rzędu, operator `is`, funkcje `read` i `write` i zmienne anonimowe.

6 Rozwiązywanie problemów

Finalnie, możemy zdefiniować sposoby rozwiązywania poszczególnych problemów, korzystając z zdefiniowanego wcześniej predykatu `ask` do uzupełniania bazy informacji w trakcie wykonywania programu. Przykładowe rozwiązania problemów, zarówno proste jak i skomplikowane zostały przedstawione na Lis. 7.

```
fix(error_camera_connected) :-
    write("Only one camera can be connected at a time."), nl,
    write("Unplug the previous camera before plugging in a new one.").

fix(flashing(status_light)) :-
    ask("Is the camera connected?"),
    write("Check camera screen for instructions.").
fix(flashing(status_light)) :-
    ask("Is the printer connected to a computer?"),
    write("Check the computer monitor.").
fix(flashing(status_light)) :- step(turn_off).

fix(not_charging(Battery)) :-
    part_parent(Battery, Parent),
    label(Battery, BLabel),
    label(Parent, PLabel),
    write("- Open the "), write(BLabel), write(" compartment in the "),
    write(PLabel), write("."), nl,
    write("- Remove the "), write(BLabel), write("."), nl,
    write("- Wait about 10 seconds."), nl,
    write("- Reinstall the battery.").

fix(not_responding(Button)) :-
    part_parent(Button, Parent),
    label(Parent, Label),
    write("An error has occured with the "), write(Label), write("."), nl,
    write("- Undock and redock the camera."), nl,
    step(turn_off).

% Fallback fix
fix(_) :- write("No suitable fix found :(").
```

Listing 7: Przykłady rozwiązywania problemów.

7 Przykładowe wywołania programu

```
2 ?- main.
Choose problem category:
1. Hardware Problems
2. Printing Problems
3. Bluetooth Problems
4. Error Message Shown
Number: 1.
Selected problem category: Hardware Problems
Choose problem:
1. The Status light is flashing red.
2. The On light flashed briefly after I turned the printer off.
3. The printer does not find and display the images.
4. The printer will not turn on.
5. The printer makes noises.
6. Camera Button on the control panel does not respond.
7. Save Button on the control panel does not respond.
8. Control Button on the control panel does not respond.
9. On Button on the control panel does not respond.
10. Print Button on the control panel does not respond.
11. Internal Battery will not charge.
12. Camera Battery will not charge.
Number: 1.
Selected problem: The Status light is flashing red.
Is the camera connected? [y/n]: n.
Is the printer connected to a computer? [y/n]: n.
Is the internal battery installed in the printer? [y/n]: y.
- Remove the battery.
- Wait about 10 seconds.
- Reinstall the battery.
```

Rysunek 2: Przykładowe wywołanie nr 1.

```
3 ?- main.
Choose problem category:
1. Hardware Problems
2. Printing Problems
3. Bluetooth Problems
4. Error Message Shown
Number: 2.
Selected problem category: Printing Problems
Choose problem:
1. Printer does not feed into the printer correctly.
2. The image is printed at an angle or is off-center.
3. No page came out of the printer.
4. The paper jammed while printing.
Number: 4.
Selected problem: The paper jammed while printing.
Did the paper come part way through the front? [y/n]: n.
Does the paper partly stick out of the tray? [y/n]: y.
Try removing the paper from the back of the printer.
```

Rysunek 3: Przykładowe wywołanie nr 2.

```
3 ?- main.
Choose problem category:
1. Hardware Problems
2. Printing Problems
3. Bluetooth Problems
4. Error Message Shown
Number: 3.
Selected problem category: Bluetooth Problems
Choose problem:
1. My Bluetooth device cannot find the printer.
2. The image printed with borders.
Number: 1.
Selected problem: My Bluetooth device cannot find the printer.
Does the Bluetooth adapter flash? [y/n]: y.
You may be too far from the printer.
Maximum recommended distance is 10 meters.
```

Rysunek 4: Przykładowe wywołanie nr 3.

8 Wnioski

Lista trzecia była niewątpliwie najprostszą ze wszystkich, ale też najmniej doprecyzowaną, przez co momentami nie było wiadome co jest oczekiwane.

Prolog jest bardzo potężnym narzędziem, ale też bardzo specyficznym. Całe jego działanie opiera się na przeszukiwaniu w głąb, backtrackingu i unifikacji, a dość mała część problemów wymaga kombinacji tych funkcjonalności do rozwiązywania. Nauka tego języka nie jest bardzo skomplikowana, szczególnie jeżeli ma się doświadczenie z funkcyjnymi językami programowania. W szczególności widzę podobieństwa do języka Erlang, który, jak się okazuje był początkowo zaimplementowany w Prologu [8]. Istnieją również różnego rodzaju biblioteki, które pozwalają na wykorzystanie podzbioru programowania opartego o klauzule hornowskie bez jednoczesnego narzutu nowej i w dodatku dość egzotycznej składni.

Literatura

[1] Instrukcja do laboratorium 3 – Piasecki, M.

[2] Rozwiązania list zadań – tchojnacki [dostęp: 25.05.2023], `https://github.com/tchojnacki/uni-sem-6-ai`

[3] Wykład 3: Wnioskowanie i symboliczna reprezentacja wiedzy – Piasecki, M.

[4] SWI–Prolog [dostęp: 25.05.2023], `https://www.swi-prolog.org/`

[5] HP Photosmart 420 series – Printer User's Manual [dostęp: 25.05.2023], `http://h10032.www1.hp.com/ctg/Manual/c00446517.pdf`

[6] Debugging and Tracing Programs – SWI–Prolog Manual [dostęp: 25.05.2023], `https://www.swi-prolog.org/pldoc/man?section=debugger`

[7] Assignment #2: Knowledge Intensive Processing – University of Zurich [dostęp: 25.05.2023], `https://files.ifi.uzh.ch/ddis/oldweb/ddis/teaching/...`

[8] Erlang – Historia – Wikipedia [dostęp: 25.05.2023], `https://pl.wikipedia.org/wiki/Erlang_(j%C4%99zyk_programowania)#Historia`

Sztuczna inteligencja i inżynieria wiedzy
K01-49i, czwartek 11:15

prowadzący
mgr inż. Katarzyna Fojcik



Politechnika
Wrocławska

Wprowadzenie do uczenia maszynowego

Lista 4

Tomasz Chojnacki
260365@student.pwr.edu.pl

WROCŁAW 15 czerwca 2023

Spis treści

1 Wstęp **2**
1.1 Wykorzystane biblioteki 2

2 Eksploracja danych **2**
2.1 Wstępne wnioski 6

3 Przygotowanie danych **7**

4 Klasyfikacja **10**

5 Ocena wykorzystanych metod **12**
5.1 Metody uzupełniania brakujących danych 13
5.2 Sposoby transformacji danych 13
5.3 Klasyfikatory 14

6 Ostateczny trening i ewaluacja **15**

7 Wnioski **18**

1 Wstęp

Celem czwartej listy zadaniowej było zapoznanie z podstawowymi algorytmami uczenia maszynowego [1]. Należało dokonać analizy zbioru danych UCI Glass Identification [2], a następnie wytrenować estymator klasyfikujący szkło do jednego z typów na podstawie zawartości poszczególnych tlenków w jego składzie. Pełny kod rozwiązania znajduje się w repozytorium [3].

1.1 Wykorzystane biblioteki

Cały eksperyment został wykonany w języku Python 3.11, w notatniku Jupyter. Do realizacji zostały wykorzystane następujące biblioteki:

- `scikit-learn` – biblioteka zapewniająca gotowe algorytmy uczenia maszynowego dla Pythona, to na zawartych w niej rozwiązaniach bazuje większość implementacji [5]
- `pandas` – biblioteka do wczytywania, obróbki i analizy danych, wykorzystana do wczytania zbioru danych, wstępnej obróbki i jego eksploracji [6]
- `matplotlib` – biblioteka do tworzenia wykresów prosto z poziomu języka Python [7]
- `numpy` – biblioteka oferująca optymalną obsługę wielowymiarowych macierzy [8]
- `seaborn` – biblioteka opakowująca `matplotlib` i oferująca łatwiejsze tworzenie estetycznych wykresów [9]

2 Eksploracja danych

Pierwszym krokiem eksperymentu była eksploracja danych, celem znalezienia wstępnych zależności, wykrycia brakujących czy też podejrzanych danych, zbadania rozkładu poszczególnych parametrów, itp.

Pierwszą, po wczytaniu i nadaniu nazw kolumnom, sprawdzoną wartością był kształt zbioru danych `dataset.shape`, który wyniósł `(214, 11)`, co oznacza 214 rekordów, każdy z 11 atrybutami. Następnie, za pomocą `dataset.head(10)` można podejrzeć kilka pierwszych rekordów w zbiorze. Wyniki zostały przedstawione w Tab. 1.

id	RI	Na	Mg	Al	Si	K	Ca	Ba	Fe	class
1	1.52101	13.64	4.49	1.10	71.78	0.06	8.75	0.0	0.00	1
2	1.51761	13.89	3.60	1.36	72.73	0.48	7.83	0.0	0.00	1
3	1.51618	13.53	3.55	1.54	72.99	0.39	7.78	0.0	0.00	1
4	1.51766	13.21	3.69	1.29	72.61	0.57	8.22	0.0	0.00	1
5	1.51742	13.27	3.62	1.24	73.08	0.55	8.07	0.0	0.00	1
6	1.51596	12.79	3.61	1.62	72.97	0.64	8.07	0.0	0.26	1
7	1.51743	13.30	3.60	1.14	73.09	0.58	8.17	0.0	0.00	1
8	1.51756	13.15	3.61	1.05	73.24	0.57	8.24	0.0	0.00	1
9	1.51918	14.04	3.58	1.37	72.08	0.56	8.30	0.0	0.00	1
10	1.51755	13.00	3.60	1.36	72.99	0.57	8.40	0.0	0.11	1

Tabela 1: Pierwsze 10 rekordów w zbiorze danych.

Możemy zauważyć w mniej więcej jakich przedziałach znajdują się dane. Rekordy są najprawdopodobniej posortowane po klasach, ponieważ widoczne są same jedynki. Zbędna jest kolumna id, którą należy usunąć aby model nie próbował wyuczyć się zależności klasy od `id`. Kolumna `RI` oznacza współczynnik załamania, a każda kolejna kolumna przed klasą oznacza zawartość poszczególnych tlenków w składzie szkła (np. `Ca` - zawartość tlenku wapnia w szkle).

Kolejną komendą jest `dataset.info()`, która zwraca typy kolumn i liczbę brakujących wartości. Zwrócony wynik został przedstawiony na Lis. 1. Widać, że w zbiorze nie ma brakujących wartości. Wszystkie cechy są liczbami zmiennoprzecinkowymi, a klasa jest liczbą całkowitą. Zbiór zajmuje raptem 18KB pamięci.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 214 entries, 0 to 213
Data columns (total 11 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   id      214 non-null    int64
 1   RI      214 non-null    float64
 2   Na      214 non-null    float64
 3   Mg      214 non-null    float64
 4   Al      214 non-null    float64
 5   Si      214 non-null    float64
 6   K       214 non-null    float64
 7   Ca      214 non-null    float64
 8   Ba      214 non-null    float64
 9   Fe      214 non-null    float64
 10  class   214 non-null    int64
dtypes: float64(9), int64(2)
memory usage: 18.5 KB
```

Listing 1: Informacje o zbiorze danych.

Następnie można wywołać `dataset.describe()`, które zwraca statystyki wszystkich atrybutów zbioru. Wyniki (zaokrąglone do jednego miejsca po przecinku) przedstawia Tab. 2.

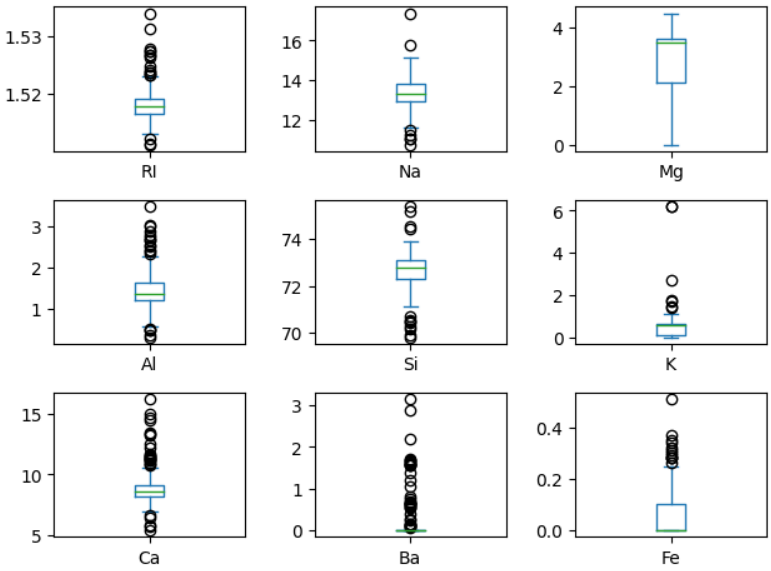
	id	**RI**	**Na**	**Mg**	**Al**	**Si**	**K**	**Ca**	**Ba**	**Fe**	**class**
count	214.0	214.0	214.0	214.0	214.0	214.0	214.0	214.0	214.0	214.0	214.0
mean	107.5	1.5	13.4	2.6	1.4	72.6	0.4	8.9	0.1	0.0	2.7
std	61.9	0.0	0.8	1.4	0.4	0.7	0.6	1.4	0.4	0.0	2.1
min	1.0	1.5	10.7	0.0	0.2	69.8	0.0	5.4	0.0	0.0	1.0
25%	54.2	1.5	12.9	2.1	1.1	72.2	0.1	8.2	0.0	0.0	1.0
50%	107.5	1.5	13.3	3.4	1.3	72.7	0.5	8.6	0.0	0.0	2.0
75%	160.7	1.5	13.8	3.6	1.6	73.0	0.6	9.1	0.0	0.1	3.0
max	214.0	1.5	17.3	4.4	3.5	75.4	6.2	16.1	3.1	0.5	7.0

Tabela 2: Statystyki wszystkich atrybutów zbioru.

Oczywiście, `id` oraz `class` nie powinny być traktowane jako dane ilościowe, a raczej nominalne i branie z nich średniej czy mediany nie ma sensu. W późniejszym etapie nie będzie to problemem, ponieważ estymatory będą pracować tylko na cechach (kolumny `RI` do `FE`). Dokumentacja zawiera nazwy poszczególnych klas, ale są one dość długie, a nie zawierają istotnych informacji, więc zamiast tego dalej będą wykorzystywane ich identyfikatory numeryczne.

Wywołanie `dataset[target_col].value_counts()` zwraca liczbę rekordów posiadających poszczególne klasy. Warto zauważyć, że brakuje w zbiorze klasy 4, co zostało z resztą opisane w dokumentacji. Liczności grup są bardzo różne, najwięcej rekordów ma klasa 2 (76 pozycji), a najmniej klasa 6 (tylko 9 pozycji). Istnieje spore ryzyko, że żaden osobnik z klasy 6 nie trafi do zbioru testowego. W związku z tym, przy późniejszym podziale wykorzystany zostanie parametr `stratify`, który zapobiega takim sytuacjom.

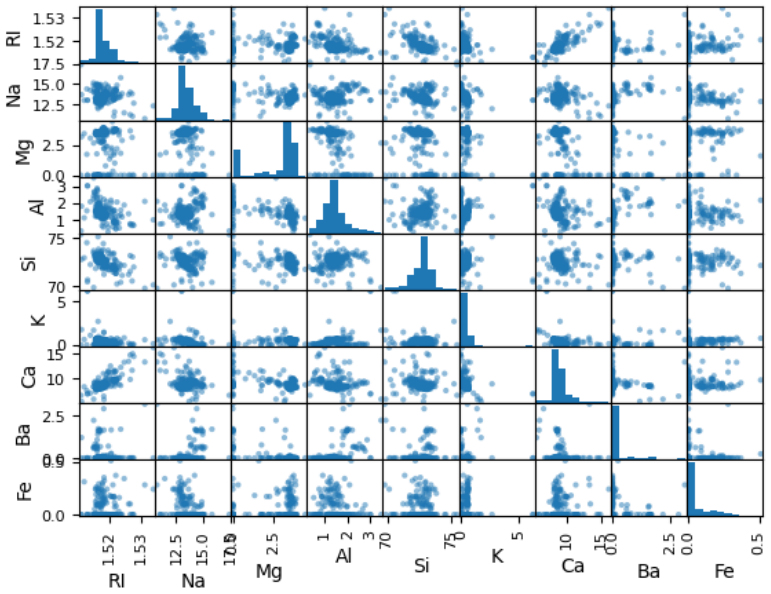
Następnie możemy narysować wykresy skrzynkowe dla wszystkich cech. Pozwala to łatwiej zauważyć dystrybucję atrybutów niż w formie tabeli. Obrazuje to Rys. 1.



Rysunek 1: Wykres skrzynkowy dla każdego atrybutu.

Widzimy, że każdy atrybut ma sporo odstępstw (*outlier*), co może utrudnić klasyfikację danych. Trudno stwierdzić, czy wartości te wynikają z błędów pomiarowych, czy też faktycznie niektóre rodzaje szkła mają bardzo nietypowe składy. Wartości poszczególnych atrybutów są bardzo skoncentrowane wokół mediany i raczej nie występują duże odchylenia. Przydatne może być zatem zastosowanie transformatora, który dobrze radzi sobie z odstępstwami [10].

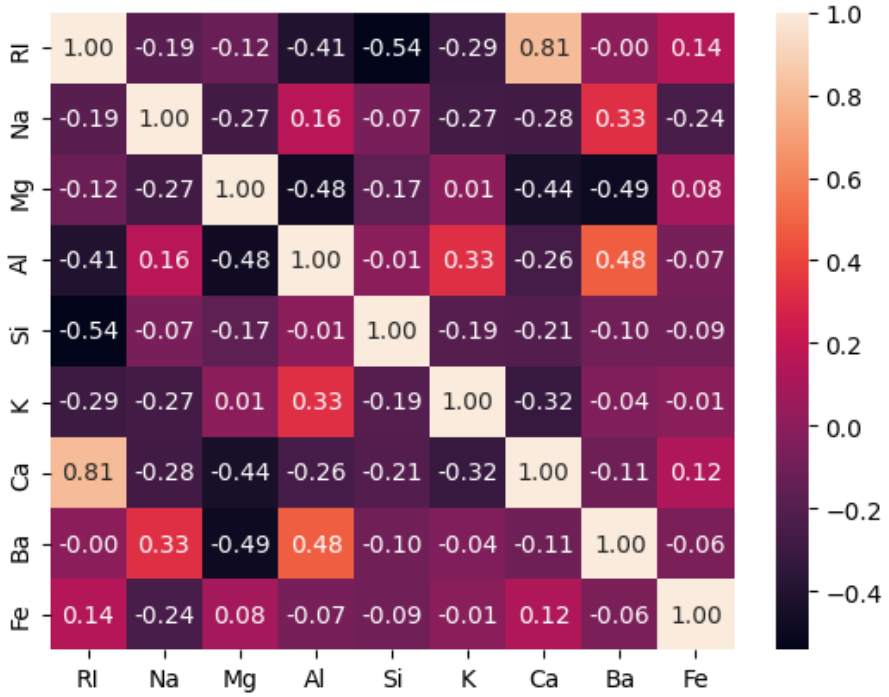
Ostatecznie tworzymy `scatter_matrix`, który przedstawia na przekątnej histogramy poszczególnych cech, a w pozostałych polach wykresy punktowe zależności między poszczególnymi parami cech. Wykres jest przedstawiony na Rys. 2.



Rysunek 2: Zależności między cechami.

Niektóre parametry mają między sobą widoczne zależności, przykładowo `RI` zdaje się być prawie liniowo zależny od `Ca`. Może to później wpłynąć negatywnie na działanie naiwnego klasyfikatora Bayesa, którego założeniem jest niezależność wszystkich cech. Widoczne są też zależności zauważone na wykresach skrzynkowych.

Następnie, w celu potwierdzenia podejrzeń zauważonych w macierzy wykresów punktowych, za pomocą biblioteki `seaborn` wyświetlamy macierz korelacji między cechami. Jest ona przedstawiona na Rys. 3. Oczywiście ignorujemy przekątną, która wyświetla korelację cechy z nią samą.

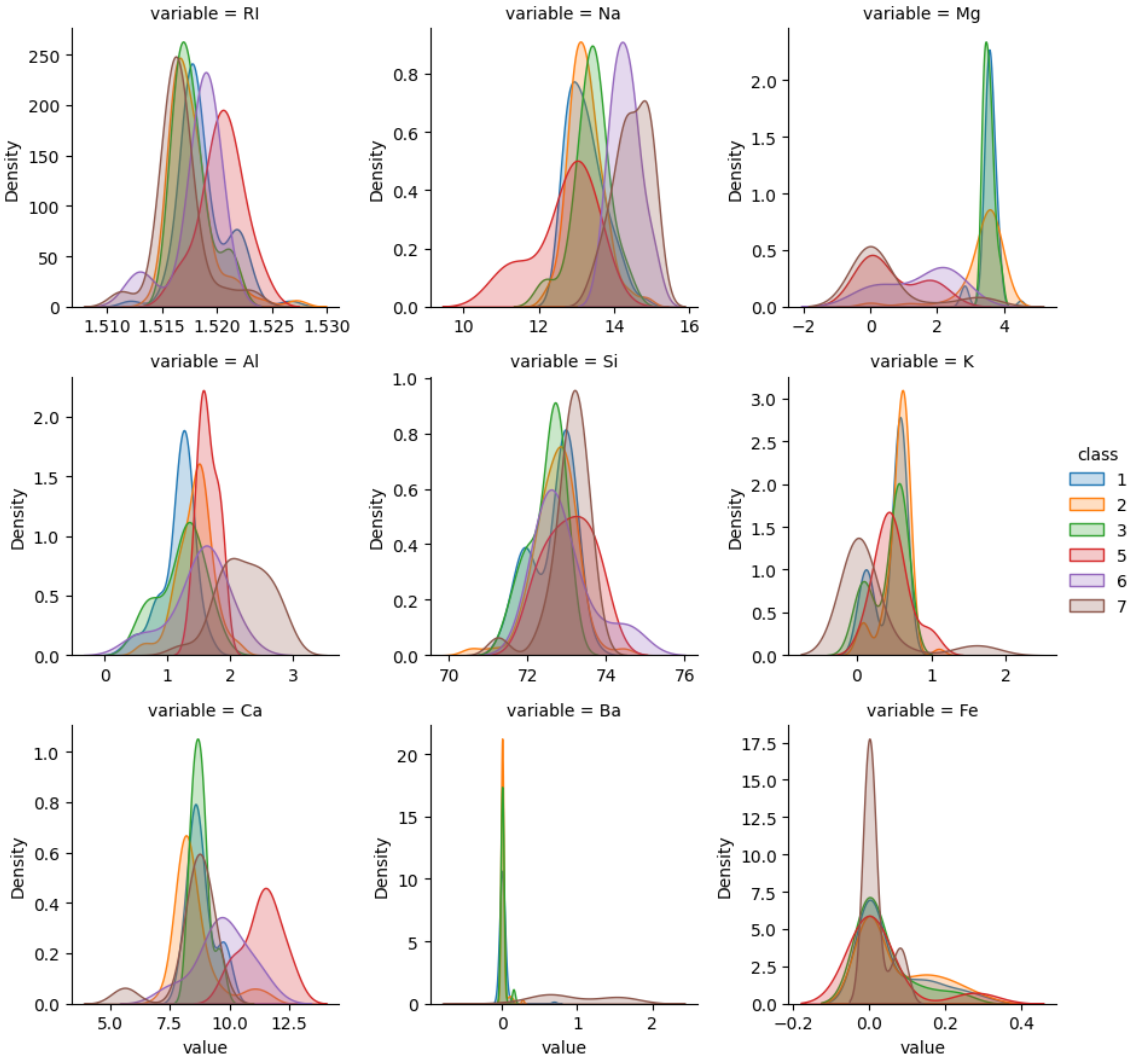


Rysunek 3: Macierz korelacji między cechami.

Można zauważyć opisaną wcześniej zależność `Ca` i `RI`, między którymi występuje korelacja równa $0,81$, ale niektóre cechy mają też dość wysoką (choć niższą niż ta) korelację.

Ostatnim, badanym w fazie eksploracji danych zagadnieniem jest rozkład (estymator jądrowy gęstości [11]) poszczególnych cech z podziałem na klasy. Wykres jest przedstawiony na Rys. 4.

Widoczne jest, że trudno będzie znaleźć cechę oczywiście odróżniającą klasy. Zawartość tlenku wapnia (lewy dolny róg) dość dobrze odseparowuje klasę 5, ale rozkład pokrywa się też częściowo z klasą 6. Z większości wykresów trudno cokolwiek wywnioskować (np. `Al` i `Ba`).



Rysunek 4: Rozkład cech z podziałem na klasy.

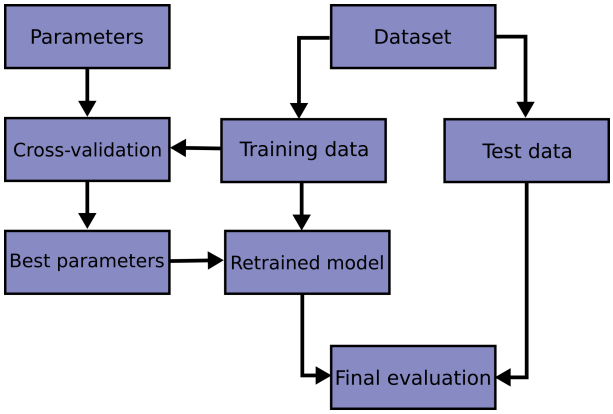
2.1 Wstępne wnioski

- Klasa rekordu jest przedstawiona jako liczba naturalna od 1 do 7.
- W zbiorze danych brakuje klasy 4.
- Rekordy zbioru są posortowane według klasy.
- Liczności rekordów są różne dla różnych klas.
- W zbiorze nie ma brakujących wartości.
- W zbiorze jest dość mało wierszy, jedynie około 200.
- Cechy zbioru są mocno skoncentrowane, ale występuje dużo odstępstw.
- Pomiędzy niektórymi parami cech występuje wysoka korelacja.
- Żaden parametr nie rozdziela klas w sposób trywialny.

3 Przygotowanie danych

Przed przygotowaniem danych ustawiono w notatniku generator liczb losowych ze stałym ziarnem, aby zapewnić reprodukowalność wyników. Rekomendowane jest przez twórców biblioteki, aby utworzyć jeden generator `numpy` ze stałym źródłem i przekazywać go do każdej metody, zamiast przekazywania zwykłej liczby [12]. Zbiór danych jest na tyle mały, że zmieniając ziarno, wyniki klasyfikacji zmieniają się w zakresie nawet do $\pm 0,05$.

Następnie cechy są odseparowane od etykiety, oraz zgodnie z bonusowym poleceniem usunięte zostaje 5% wartości w zbiorze. Dane dzielone są na podzbiór treningowy i testowy. Zgodnie z procesem zalecanym przez `scikit-learn` (Rys. 5), zbiór ten będzie wykorzystany w finalnej ewaluacji, natomiast trenowanie i optymalizacja hiperparametrów następuje drogą walidacji krzyżowej zbioru treningowego. Kod wszystkich opisanych powyżej kroków pokazuje Lis. 2.



Rysunek 5: Standardowy przepływ pracy z modelem w `scikit-learn` [13].

```
data_cols = ["RI", "Na", "Mg", "Al", "Si", "K", "Ca", "Ba", "Fe"]
target_col = "class"

rng = np.random.RandomState(1)

X_full = dataset[data_cols]
Y = dataset[target_col]

missing_ratio = 0.05
X = X_full.mask(rng.random(X_full.shape) < missing_ratio)

test_size = 0.3
X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
                                                    test_size=test_size,
                                                    stratify=Y,
                                                    random_state=rng)
```

Listing 2: Pierwsze kroki przetwarzania danych.

Został wykonany podział ze stratyfikacją, aby zapobiec sytuacji brakujących klas w zbiorze testowym. Warto zauważyć, że funkcja `train_test_split` automatycznie losuje kolejność rekordów w zbiorze przed podziałem, co zapobiega problemom wynikającym z początkowego posortowania danych według klasy. Istotne jest też, że na tym kroku została usunięta już kolumna `id`, a macierz `X` nie zawiera etykiet.

Główny tok przetwarzania będzie składać się z dwóch etapów: wypełniania brakujących danych i ewentualnej transformacji zbioru. Przetestowane zostały następujące strategie wypełniania luk:

- `KNNImputer(n_neighbors=1)` – `KNN-1`
- `KNNImputer(n_neighbors=3)` – `KNN-3`
- `SimpleImputer(strategy="mean")` – `MEAN`
- `SimpleImputer(strategy="median")` – `MEDIAN`

Strategie KNN wypełniają brakujące dane wykorzystując średnią z `n` najbliższych (pod względem niebrakujących cech) sąsiadów rekordu. W szczególności, dla $n = 1$, kopiowana jest wartość z najbliższego rekordu. Strategia `SimpleImputer` wypełnia korzystając ze statystyki obliczonej wśród wszystkich rekordów – odpowiednio średniej i mediany.

Następny krok stanowi transformacja, wybrana jako jedna z poniższych [10]:

- `"passthrough"` – brak transformacji, specjalny łańcuch tekstowy, traktowany przez bibliotekę jako funkcja tożsamościowa $f(x) = x$.
- `StandardScaler` – zeruje średnią zbioru i skaluje go tak, żeby miał wariancję równą jeden. Sporo algorytmów estymacji wymaga, bądź działa lepiej, jeżeli otrzymuje takie dane. Algorytm nie gwarantuje zbalansowanego rozkładu, jeżeli dane wejściowe zawierają odstępstwa.
- `Normalizer` – normalizuje wektor cech, tzn. skaluje je tak, żeby miały długość równą jeden. W przypadku dwóch wymiarów oznacza to umieszczenie punktów na okręgu, a gdy wszystkie dane są nieujemne, na jego pierwszej ćwiartce.
- `KBinsDiscretizer` – dyskretyzuje dane do kilku kubełków zawierających odpowiednie przedziały. Hiperparametr `n_bins` decyduje o ilości przedziałów. Wykorzystana została strategia `quantile`, która jest domyślną wartością. Zapewnia ona, że każdy kubełek ma równą liczność próbek (w przeciwieństwie do strategii `uniform`, która zapewnia równą szerokości kubełków).
- `VarianceThreshold` – odrzuca cechy, które nie przekraczają progu (`threshold`) wariancji.
- `SelectKBest` – pozostawia tylko `k` najlepszych cech (bazując na testach statystycznych).
- `PCA` – analiza głównych składowych. Obraca układ współrzędnych tak, aby zmaksymalizować wariancję pierwszej współrzędnej, później aby zmaksymalizować wariancję drugiej (prostopadłej) współrzędnej, itd. Domyślnie automatycznie dobiera ilość komponentów do pozostawienia.
- `RobustScaler` – skalowanie odporne na odstępstwa. Centruje medianę, a nie średnią, a następnie skaluje bazując na przedziale między pierwszym a trzecim kwartylem. W przypadkach, kiedy występuje wiele odstępstw, może dać lepsze rezultaty niż skalowanie.
- `PowerTransformer` – transformuje dane z wykorzystaniem algorytmu Yeo-Johnsona (nieliniowo), co sprawia, że mają zerową średnią i jednostkową wariancję, ale uwzględnia odstępstwa. Kształt rozkładu zmienia się za sprawą tego na przypominający krzywą Gaussa.
- `QuantileTransformer` – nakłada na dane nieliniową transformację taką, że wszystkie dane (wraz z odstępstwami) są mapowane do rozkładu jednostajnego. Tak samo jak `RobustScaler` i `PowerTransformer` działa dobrze z odstępstwami, ale w przeciwieństwie do nich ogranicza wszystkie dane do przedziału (0, 1). Możliwe jest też użycie parametru `output_distribution` równego `normal`, co dokonuje tej samej transformacji, ale centruje dane zgodnie z krzywą normalną.

Za definicję imputerów i transformatorów odpowiedzialny jest kod przedstawiony na Lis. 3. Pominięty został tutaj parametr `random_state` wykorzystany w rzeczywistej implementacji.

```
imputers = {
    "KNN-1": KNNImputer(n_neighbors=1),
    "KNN-3": KNNImputer(n_neighbors=3),
    "MEAN": SimpleImputer(strategy="mean"),
    "MEDIAN": SimpleImputer(strategy="median")
}

transformers = {
    "PASSTHROUGH": "passthrough",
    "SCALE": StandardScaler(),
    "NORMALIZE": Normalizer(),
    "DISCRETIZE-3": KBinsDiscretizer(n_bins=3, encode="onehot-dense"),
    "VAR-THRESHOLD-0.5": VarianceThreshold(threshold=0.5),
    "SELECT-3-BEST": SelectKBest(k=3),
    "PCA": PCA(svd_solver="auto"),
    "ROBUST": RobustScaler(),
    "POWER": PowerTransformer(),
    "QUANTILE-UNIFORM": QuantileTransformer(output_distribution="uniform"),
    "QUANTILE-NORMAL": QuantileTransformer(output_distribution="normal")
}
```

Listing 3: Przetestowane strategie wypełniania i transformowania danych.

Zastosowanie skalowania przed wykonaniem PCA jest w stanie polepszyć osiągi estymatora [14]. Jednakże na tym konkretnym zbiorze danych nie było możliwe zauważenie tej zależności, więc taka kombinacja nie została umieszczona w ostatecznym zestawieniu. Warto też pamiętać, że o ile PCA jest w stanie poprawić jakość dokonywanych predykcji, jego główną zaletą jest zredukowanie wymiarowości danych, co pozwala skrócić rozmiar oraz czas przetwarzania zbioru danych. Zbiór GLASS jest na tyle mały, że nie jest to konieczne.

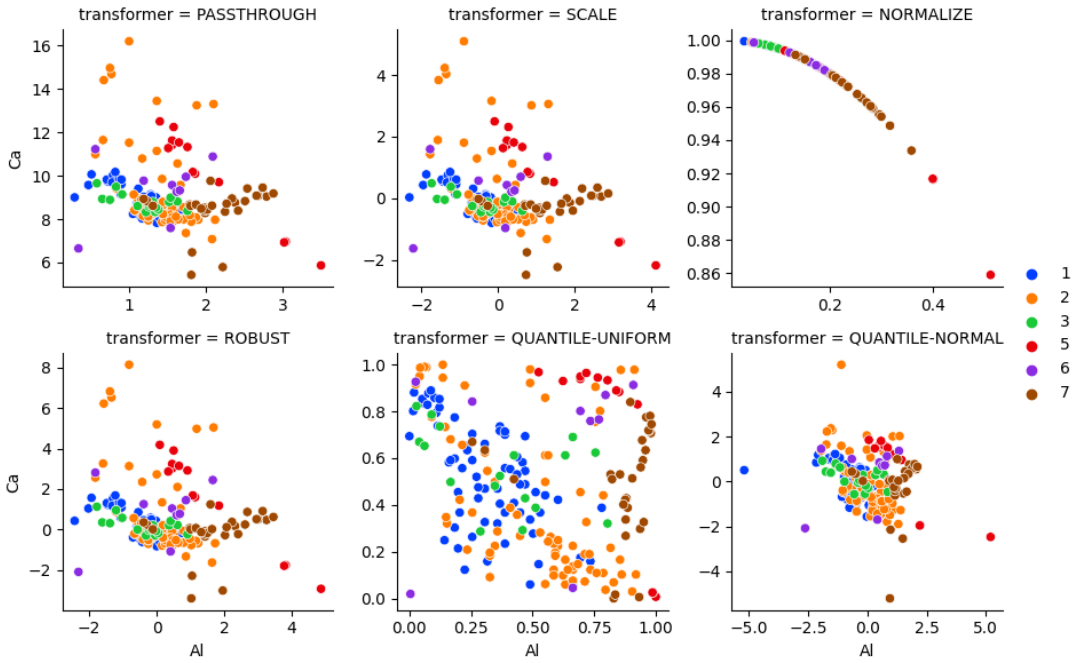
Transformatory `RobustScaler` i `QuantileTransformer` są wskazane przez twórców biblioteki jako najskuteczniejsze dla danych zawierających wiele odstępstw [10], dlatego można podejrzewać, że osiągną dobre wyniki na zbiorze GLASS.

Na Rys. 6 zostały przedstawione rozkłady arbitralnie wybranych dwóch cech ze zbioru przed transformacją (`PASSTHROUGH`) oraz po kilku wybranych transformacjach.

Wyniki transformacji `SCALE` oraz `ROBUST` zdają się być identyczne jak rozkład sprzed transformacji, ale warto zauważyć, że wykresy nie współdzielą skali i wartości na osiach liczbowych w każdym z trzech przypadków są różne. Transformacja `NORMALIZE`, w związku z tym, że wszystkie dane są nieujemne, tworzy ćwiartkę okręgu w pierwszej ćwiartce układu współrzędnych.

Żadna z transformacji nie potrafiła bardzo dobrze odseparować osobników poszczególnych klas między sobą, ale wnioskując po wykresach najlepiej poradził sobie `QUANTILE-UNIFORM`, który prawie całkowicie odseparował klasy 1 (niebieska) i 7 (brązowa). Wszystkie rekordy z klasy 5 (czerwona) również skumulowały się w jednym miejscu, oprócz dwóch osobników, w przypadku których mógł nastąpić jakiś błąd pomiarowy. Najgorzej odseparowane są osobniki klasy 2 (pomarańczowa), które są rozrzucone po całym przedziale.

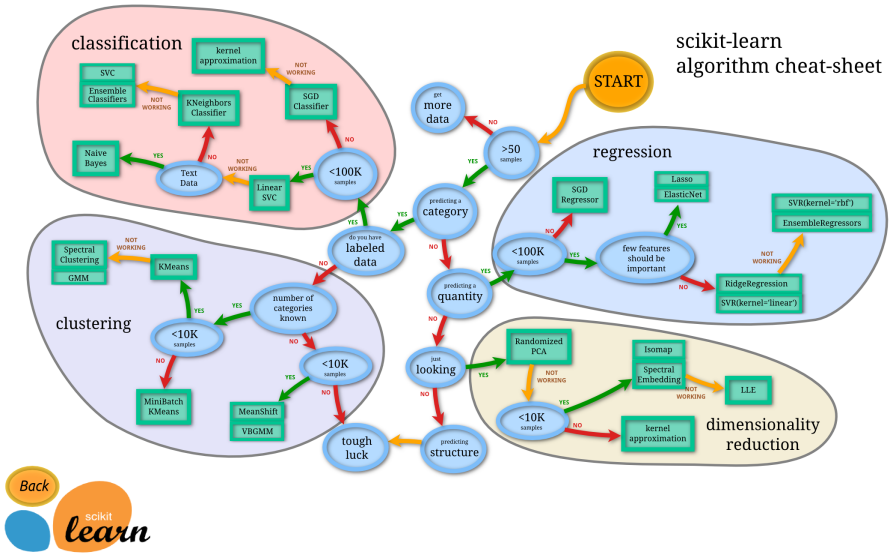
Porównanie wyników osiąganych przez poszczególne metody uzupełniania brakujących danych oraz transformatorów znajduje się w sekcji „Ocena wykorzystanych metod".



Rysunek 6: Cechy `Al` i `Ca` przed i po transformacjach.

4 Klasyfikacja

Naiwny klasyfikator Bayesa oraz drzewo decyzyjne były klasyfikatorami obowiązkowymi, wymienionymi w instrukcji. Oprócz tego, w ramach polecenia dodatkowego należało przetestować klasyfikatory oparte na SVM oraz lasy losowe. Przetestowane zostały wszystkie powyżej wymienione klasyfikatory oraz klasyfikator K najbliższych sąsiadów, który jest rekomendowany przez twórców `scikit-learn` do tego typu zadań (Rys. 7).



Rysunek 7: Flowchart opisujący wybór odpowiedniego estymatora do zadania [15].

Kod odpowiedzialny za deklarację klasyfikatorów znajduje się na Lis. 4. Pominięty został parametr `random_state`, który w rzeczywistej implementacji został podany do każdego klasyfikatora, który go wspiera.

```
classifiers = {
    "NAIVE-BAYES-1e-9": GaussianNB(var_smoothing=1e-9),
    "NAIVE-BAYES-1e-6": GaussianNB(var_smoothing=1e-6),
    "NAIVE-BAYES-1e-3": GaussianNB(var_smoothing=1e-3),
    "DECISION-TREE-2-GINI": DecisionTreeClassifier(
                                    max_depth=2,
                                    criterion="gini"),
    "DECISION-TREE-10-GINI": DecisionTreeClassifier(
                                    max_depth=10,
                                    criterion="gini"),
    "DECISION-TREE-2-ENTROPY": DecisionTreeClassifier(
                                    max_depth=2,
                                    criterion="entropy"),
    "DECISION-TREE-10-ENTROPY": DecisionTreeClassifier(
                                    max_depth=10,
                                    criterion="entropy"),
    "KNN-3": KNeighborsClassifier(n_neighbors=3),
    "SVC-LINEAR": LinearSVC(),
    "SVC-RBF": SVC(gamma="auto"),
    "RANDOM-FOREST": RandomForestClassifier()
}
```

Listing 4: Wykorzystane w eksperymencie klasyfikatory.

W przypadku naiwnego klasyfikatora Bayesa, hiperparametr `var_smoothing` jest odpowiedzialny za zwiększenie stabilności obliczeń poprzez dodanie małej części największej wariancji w zbiorze do wszystkich pozostałych wariancji. Generalnie, klasyfikator ten nie jest najlepszym narzędziem do zadania wykonywanego na tym zbiorze, ponieważ jednym z jego założeń jest niezależność danych (która nie zawsze zachodzi w zbiorze GLASS co wynika z analizowanej wcześniej macierzy korelacji).

Model drzewa decyzyjnego przyjmuje łącznie 12 hiperparametrów, ale większość z nich można zostawić w domyślnej konfiguracji. Pierwszym istotnym hiperparametrem jest `criterion`, który określa kryterium wykorzystane do oceny jakości podziału (a co za tym idzie wyboru podziału do dokonania). Dostępne opcje to `gini` (1) oraz `entropy` (2), gdzie k to indeks wyboru w obecnym węźle drzewa.

$$H(Q) = \sum_k p_k(1 - p_k) \tag{1}$$

$$H(Q) = -\sum_k p_k \log(p_k) \tag{2}$$

W zależności od wartości hiperparametru `splitter` wybrany jest najlepszy (`best`) lub losowy spośród najlepszych (`random`) podział. Natomiast `max_depth` decyduje o maksymalnej głębokości drzewa. Domyślnie, drzewo jest dzielone, dopóki w każdej nowo powstałej sekcji będą znajdowały się co najmniej dwa rekordy.

Klasyfikator KNN jest najprostszy – rekord jest klasyfikowany według „głosowania” wśród jego K najbliższych sąsiadów w przestrzeni wielowymiarowej. Domyślnie, każdy sąsiad ma taką samą wagę głosu, można zamiast tego ustawić, np. priorytetyzację najbliższych sąsiadów.

Klasyfikator SVM (maszyny wektorów nośnych) wyznacza hiperpłaszczyznę która rozdziela klasy z maksymalnym „marginesem" opierając się o osobniki ze zbioru danych nazywane wektorami nośnymi. Można uzyskać podział nieliniowy, stosując nieliniowe jądro (np. `rbf`, które jest domyślną wartością). Klasa `LinearSVC` jest wariantem `SVC` zoptymalizowanym specjalnie dla liniowego jądra (`kernel="linear"`). Hiperparametr `gamma` określa współczynnik jądra, dla `auto` (wartość domyślna) wynosi $1/N$.

Las drzew losowych trenuje `n_estimators` (domyślnie 100) drzew decyzyjnych na podzbiorach zbioru danych, a następnie kwalifikuje korzystając ze średniej z wyników przewidywanych przez poszczególne drzewa.

Pełne przetwarzanie danych w estymatorze można opakować w obiekt zwany `Pipeline`, który skraca kod, ułatwia optymalizację hiperparametrów oraz zapobiega wyciekom danych testowych do treningowych. Wykorzystany „rurociąg" operacji został przedstawiony na Lis. 5. Podane początkowo wartości poszczególnych kroków (drugi element krotki) nie są istotne, ponieważ metody optymalizacji będą w ich miejsce wstawiać faktyczne estymatory i transformacje.

Następnie szukamy najlepszej kombinacji tych trzech kroków metodą siatki (zwaną też metodą kratownicy), czyli dopasowując wszystkie istniejące kombinacje.

```
search_pipeline = Pipeline([
    ("imputer", "passthrough"),
    ("transformer", "passthrough"),
    ("classifier", next(iter(classifiers.values())))
])

grid_search = GridSearchCV(search_pipeline, {
    "imputer": list(imputers.values()),
    "transformer": list(transformers.values()),
    "classifier": list(classifiers.values())
}, n_jobs=-1)

grid_search.fit(X_train, Y_train)

print(grid_search.best_score_)
best_pipeline = grid_search.best_estimator_
best_pipeline
```

Listing 5: Wykorzystana struktura `Pipeline` i poszukiwania metodą siatki.

Najlepszą kombinacją okazał się model `KNN-1`, `QUANTILE` i `RANDOM-FOREST`, który osiągnął trafność $0,751$ w walidacji krzyżowej na zbiorze treningowym. Na podstawie pozostałych wyników z przeszukiwania siatki zostało zbudowane porównanie metod przedstawione w kolejnym rozdziale.

5 Ocena wykorzystanych metod

Niemożliwe jest ocenianie transformacji, czy też estymatorów w izolacji, raczej powinno się je rozważać jako całość. Przykładowo, drzewa decyzyjne (w tym ich las) nie są wrażliwe na wariancję i `StandardScaler` nie ma na nich prawie żadnego wpływu, co nie znaczy, że ta transformacja nie mogłaby poprawić wyników innego klasyfikatora (np. KNN). W związku z tym, ocena klasyfikatorów i transformatorów dokonana została na podstawie najbardziej optymalnej kombinacji, w której występują.

Istotne jest też podkreślenie, że wykorzystane metody zostały ocenione tylko i wyłącznie pod względem zadania, którym jest klasyfikacja rekordów zbioru danych GLASS i struktura tego zbioru danych ma spory wpływ na jakość klasyfikacji. Przykładowo, naiwny klasyfikator Bayesa prawdopodobnie uplasował by się wyżej, gdyby problemem była klasyfikacja tekstu.

Wszystkie poniższe wykresy przedstawiają średnią dokładność, osiągniętą w walidacji krzyżowej na zbiorze testowym najoptymalniejszej konfiguracji, w której występuje dana metoda, wraz z odchyleniem standardowym.

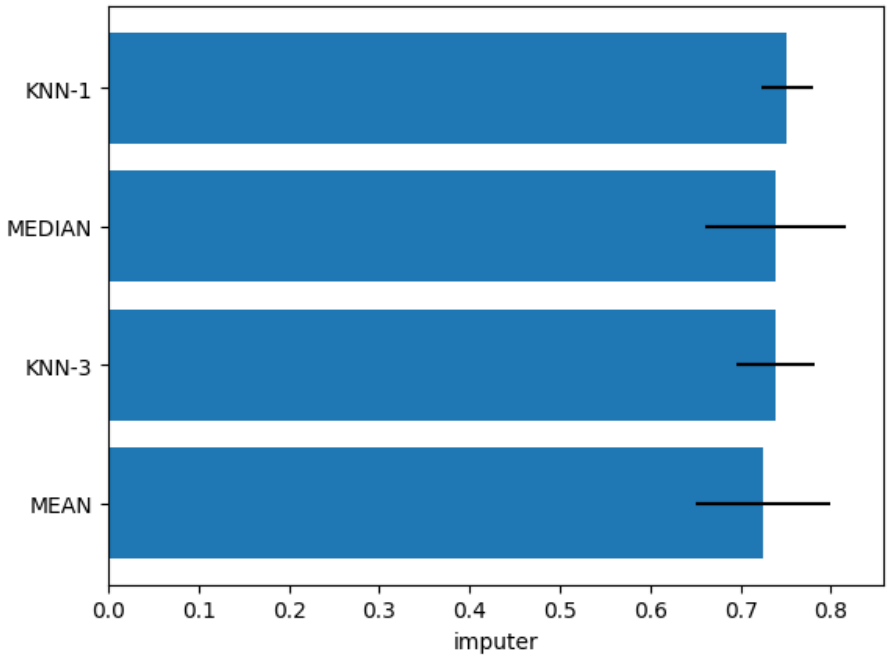
5.1 Metody uzupełniania brakujących danych

Wśród metod uzupełniania danych, najlepsza okazała się `KNN-1`, przy czym wyniki osiągnięte przez wszystkie metody są porównywalne i leżą w granicy błędu statystycznego. Przy wykorzystaniu innego źródła często wygrywała zamiast tego metoda `MEDIAN`.

W tym przypadku, wybór metody wypełniania danych okazał się dość nieistotny. Być może różnica byłaby bardziej zauważalna, gdyby brakowało większej ilości danych.

Dodatkowe porównanie ze zbiorem, w którym nie brakuje danych znajduje się w sekcji opisującej ostateczną ewaluację.

Wyniki są przedstawione na Rys. 8.



Rysunek 8: Trafność poszczególnych metod uzupełniania brakujących danych.

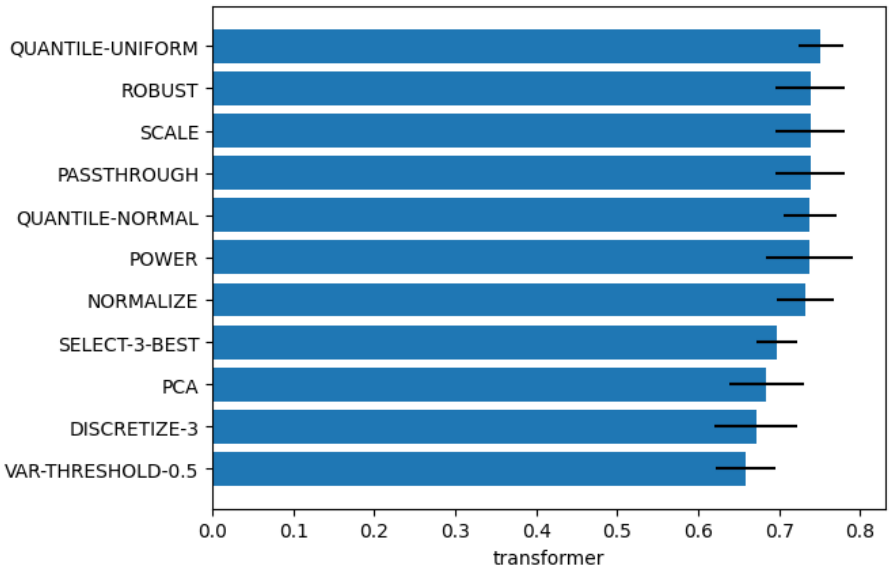
5.2 Sposoby transformacji danych

W przeciwieństwie do poprzedniej kategorii, wśród sposobów transformacji danych można zauważyć większe różnice w osiągniętych wynikach. Zgodnie z oczekiwaniami, metoda `QUANTILE`, czyli ta, która jest odporna na odstępstwa i najlepiej wizualnie dzieliła klasy, poradziła sobie najlepiej.

W następnej kolejności znalazły metody `ROBUST` i `SCALE`, które poradziły sobie lepiej niż źródłowy zbiór danych. Najgorsze wyniki osiągnęły metody selekcji, dyskretyzacji i progu wariancji. Na ich słabe osiągi spory wpływ mógł mieć nieodpowiedni dobór hiperparametrów, które zostały wybrane arbitralnie.

Rozczarowujący wynik osiągnęła również metoda PCA, ale warto podkreślić, że pomniejszyła ona zbiór z 11 do 9 cech i w przypadku większego zbioru danych tego typu redukcja mogłaby mieć ogromny wpływ na rozmiar zbioru w pamięci i czas treningu. Przy odpowiednim zbiorze danych, PCA potrafi zredukować wymiarowość danych nawet kilkukrotnie jednocześnie zachowując ponad 95% informacji zawartych w danych.

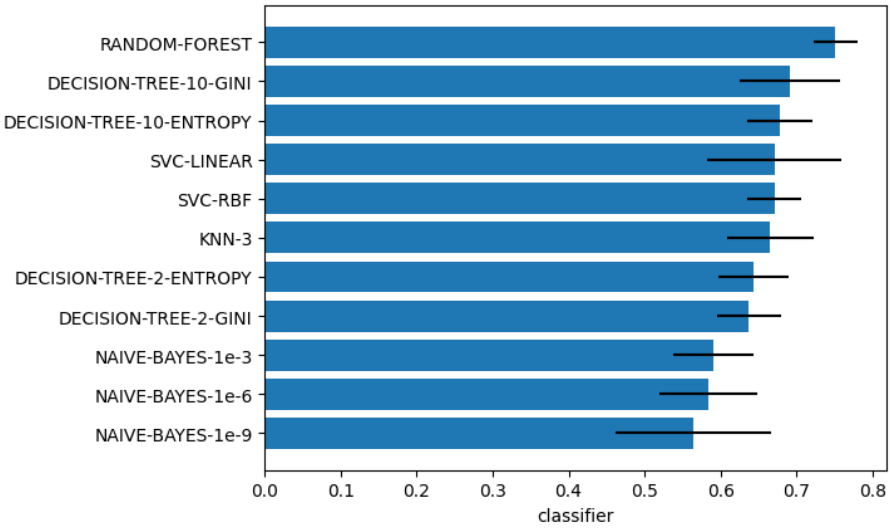
Wyniki są przedstawione na Rys. 9.



Rysunek 9: Trafność poszczególnych metod transformacji danych.

5.3 Klasyfikatory

Wyniki są przedstawione na Rys. 10.



Rysunek 10: Trafność poszczególnych klasyfikatorów.

Dobór klasyfikatora miał największy wpływ na ostateczny wynik. Najlepszy okazał się klasyfikator lasu drzew losowych, a bardzo dobre wyniki osiągnęły również tradycyjne drzewa losowe, metoda KNN i maszyna wektorów nośnych. Metoda naiwnego klasyfikatora Bayesa osiągnęła

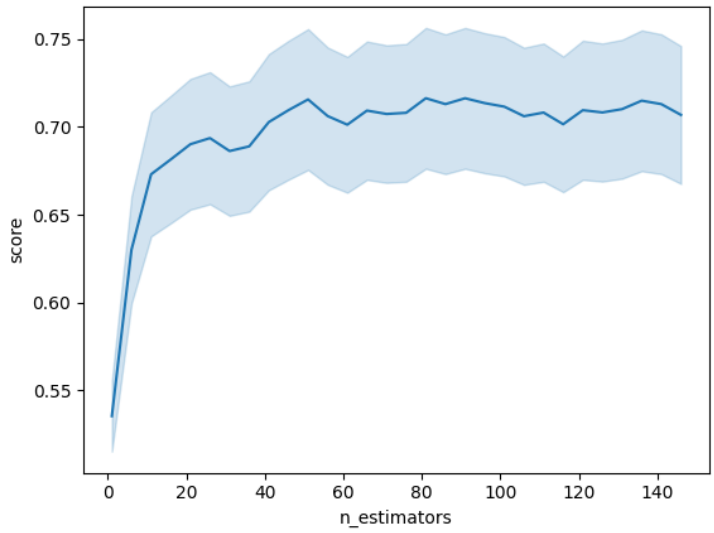
dużo gorszą trafność, w najlepszym przypadku rzędu $0,5 - 0,6$. Mogła mieć na to wpływ zależność między cechami jak i słabe dostosowanie algorytmu do tego typu zadań.

6 Ostateczny trening i ewaluacja

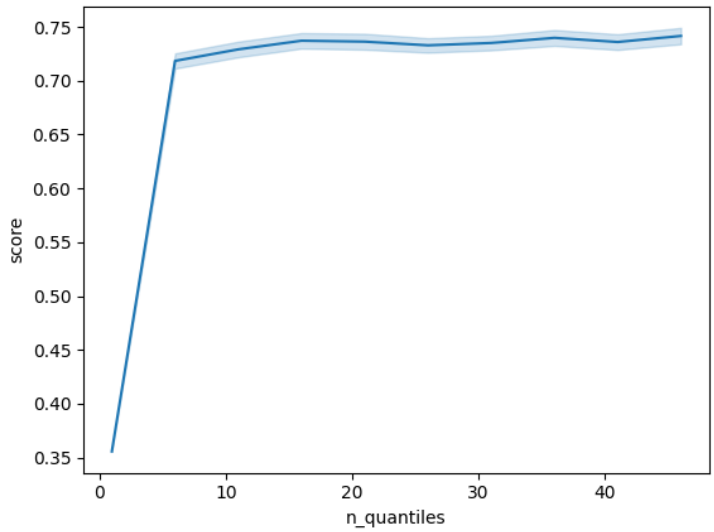
Przed ostateczną ewaluacją na zbiorze testowym, została dokonana optymalizacja hiperparametrów:

- `quantiletransformer__n_quantiles` – od 1 do 50
- `randomforestclassifier__n_estimators` – od 1 do 150

Wpływ liczby estymatorów w lesie oraz kwantyli w transformatorze na trafność modelu został przedstawiony na wykresach na Rys. 11 i 12.



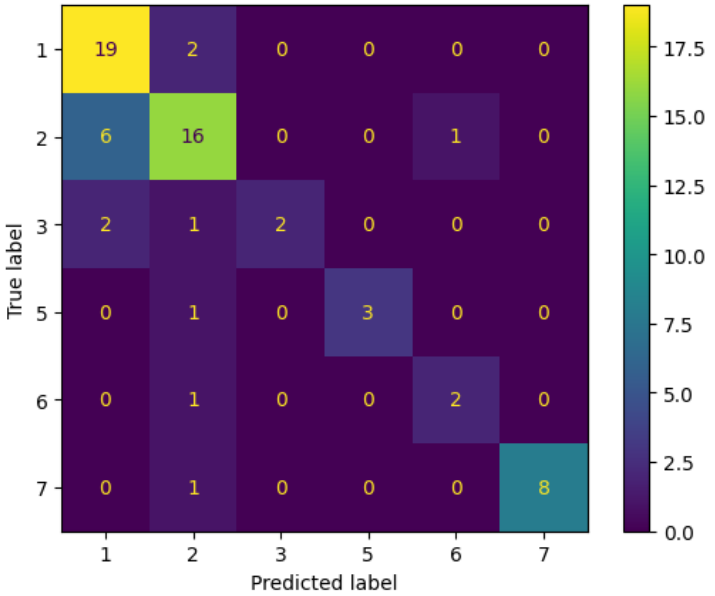
Rysunek 11: Wpływ liczby drzew w lesie na trafność modelu.



Rysunek 12: Wpływ liczby kwantyli transformatora na trafność modelu.

Następnie, pipeline został ponownie wytrenowany na zbiorze treningowym i po raz pierwszy przetestowany na wydzielonym na samym początku eksperymentu zbiorze testowym. Chodzi oczywiście o wersję zbioru z usuniętymi 5% danych.

Macierz konfuzji i wyniki znajdują się odpowiednio na Rys. 13 oraz Lis. 6.



Rysunek 13: Macierz konfuzji (95% danych).

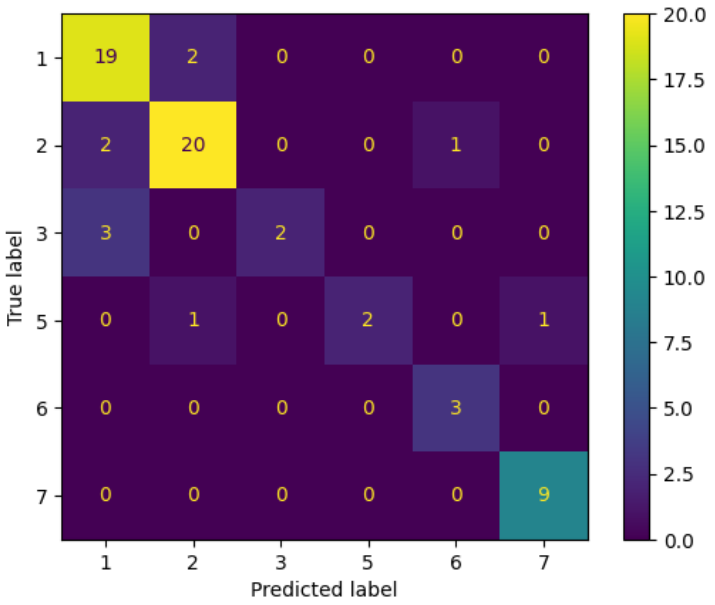
	precision	recall	f1-score	support
1	0.70	0.90	0.79	21
2	0.73	0.70	0.71	23
3	1.00	0.40	0.57	5
5	1.00	0.75	0.86	4
6	0.67	0.67	0.67	3
7	1.00	0.89	0.94	9
accuracy			0.77	65
macro avg	0.85	0.72	0.76	65
weighted avg	0.79	0.77	0.77	65

Listing 6: Raport klasyfikacji (95% danych).

Model osiągnął trafność $0,77$, w zależności od doboru ziarna generatora losowego wartości wahają się w przedziale $0,75 - 0,80$.

Model został również przetestowany na pełnym zbiorze testowym (z dokonaniem takiego samego podziału na dane treningowe i testowe).

Macierz konfuzji i wyniki znajdują się odpowiednio na Rys. 14 oraz Lis. 7.



Rysunek 14: Macierz konfuzji (100% danych).

	precision	recall	f1-score	support
1	0.79	0.90	0.84	21
2	0.87	0.87	0.87	23
3	1.00	0.40	0.57	5
5	1.00	0.50	0.67	4
6	0.75	1.00	0.86	3
7	0.90	1.00	0.95	9
accuracy			0.85	65
macro avg	0.89	0.78	0.79	65
weighted avg	0.86	0.85	0.84	65

Listing 7: Raport klasyfikacji (100% danych).

Model osiągnął trafność $0,84$, w zależności od doboru ziarna generatora losowego wartości wahają się w przedziale $0,82 - 0,87$.

Wykorzystane metryki można rozumieć następująco [1]:

- `accuracy` – jaka część danych została zakwalifikowana poprawnie
- `precision` – jaka część danych predykowanych pozytywnie jest rzeczywiście pozytywna
- `recall` – jaka część wszystkich przypadków pozytywnych została poprawnie predykowana
- `f1-score` – średnia harmoniczna z `precision` i `recall`
- `support` – liczba rekordów każdej klasy w zbiorze testowym

To z której metryki korzystamy zależy od potrzeby biznesowej. Przykładowo, czasami bardzo istotne jest, aby ograniczyć liczbę próbek błędnie zaklasyfikowanych pozytywnie do minimum, a czasami lepiej, aby występowały próbki fałszywie ocenione jako pozytywne, ale za to żeby każde faktycznie pozytywne próbki zostały wykryte. Jako że wykonane zadanie jest czysto dydaktyczne, trudno wybrać odpowiednią metrykę, bo nie wiemy do czego jest wykorzystana zaimplementowana klasyfikacja szkła.

7 Wnioski

Tradycyjne klasyfikatory uczenia maszynowego radzą sobie dobrze z problemem identyfikacji rodzajów szkła ze zbioru UCI GLASS, choć jest to w związku z małą ilością próbek oraz sporą liczbą odstępstw dość trudny do klasyfikacji zbiór danych. Na współczesne zastosowania odpowiednimi algorytmami do tego zadania są: klasyfikator wektorów nośnych, las drzew losowych i metoda K najbliższych sąsiadów.

Eksploracja danych pozwala zaobserwować związki w zbiorze danych, które później można wykorzystać do poprawnej klasyfikacji zbioru. Dogłębna analiza danych pozwoliła domyślić się, które metody transformacji i klasyfikacji odniosą najlepsze wyniki jeszcze przed dokonaniem testów.

Podział zbioru danych na treningowy i testowy oraz walidacja krzyżowa pozwalają dobrać odpowiednie estymatory i ich hiperparametry, jednocześnie zapobiegając nadmiernemu dopasowaniu danych.

Dokumentacja biblioteki `scikit-learn` jest bardzo dobrze wykonana. Oprócz opisu klasyfikatorów i ich hiperparametrów zawiera sekcje zawierające opisy teoretyczne, oraz wymienia dobre praktyki i metody uniknięcia częstych problemów.

Literatura

[1] Instrukcja do laboratorium 4 – Fojcik, K., Szołomicka, J.

[2] Glass Identification – UCI Machine Learning Repository [dostęp: 10.06.2023], https://archive.ics.uci.edu/dataset/42/glass+identification

[3] Rozwiązania list zadań – tchojnacki [dostęp: 15.06.2023], https://github.com/tchojnacki/uni-sem-6-ai

[4] Wykład 5: Wprowadzenie do uczenia maszynowego – Kwaśnicka H., Piasecki, M.

[5] Scikit-learn: Machine Learning in Python – Pedregosa, F. et al. [dostęp: 11.06.2023], https://jmlr.csail.mit.edu/papers/v12/pedregosa11a.html

[6] Data Structures for Statistical Computing in Python – McKinney, W. [dostęp: 12.06.2023], https://conference.scipy.org/proceedings/scipy2010/pdfs/mckinney.pdf

[7] Matplotlib: A 2D graphics environment – Hunter, J. D. [dostęp: 12.06.2023], https://ieeexplore.ieee.org/document/4160265

[8] Array programming with NumPy – Harris, C.R. et al. [dostęp: 12.06.2023], https://www.nature.com/articles/s41586-020-2649-2

[9] seaborn: statistical data visualization – Waskom, M. L. [dostęp: 12.06.2023], https://doi.org/10.21105/joss.03021

[10] Compare the effect of scalers on data with outliers – scikit-learn [dostęp: 14.06.2023], https://scikit-learn.org/.../plot_all_scaling.html

[11] Estymacja jądrowa gęstości – Meyer, S. [dostęp: 15.06.2023], http://www.math.uni.wroc.pl/~p-wyk4/mag2021a/refs/Estymatory_jadrowe.pdf

[12] Common pitfalls and recommended practices – scikit-learn [dostęp: 12.06.2023], https://scikit-learn.org/stable/common_pitfalls.html

[13] Cross–validation: evaluating estimator performance – scikit-learn [dostęp: 12.06.2023], https://scikit-learn.org/stable/modules/cross_validation.html

[14] Importance of Feature Scaling – scikit-learn [dostęp: 12.06.2023], https://scikit-learn.org/.../plot_scaling_importance.html

[15] Choosing the right estimator – scikit-learn [dostęp: 13.06.2023], https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html