DOKUMENTACJA TECHNICZNA PROJEKTU GEOKODOWANIA

18 stycznia 2025

Spis treści

Rozdział 1

Opis techniczny projektu

1.1 Wprowadzenie

1.1.1 Cel aplikacji

Celem aplikacji jest przetwarzanie, geokodowanie i analiza danych adresowych. Program wczytuje dane z pliku CSV, przetwarza informacje o adresach, wzbogaca je o współrzędne geograficzne (szerokość i długość geograficzną) oraz dodatkowe dane, takie jak nazwa miejsca, miasto, kod pocztowy i ulica. Wyniki są zapisywane w pliku CSV, który może być wykorzystany do dalszej analizy.

1.1.2 Zakres dokumentacji

Dokumentacja opisuje:

- Funkcjonalności programu,
- Struktury danych i algorytmy użyte w aplikacji,
- Szczegóły implementacyjne, takie jak wielowątkowość, obsługa sieci i bazy danych,
- Format danych wejściowych i wyjściowych.

1.1.3 Technologie i wymagania

Program został napisany w języku C++ i korzysta z następujących technologii:

- **PostgreSQL z PostGIS** – do obsługi danych geograficznych,
- **Nominatim API** – do geokodowania adresów,
- **Wielowątkowość** – do równoległego przetwarzania danych,
- **Asynchroniczne połączenia sieciowe** – do obsługi zapytań HTTP.

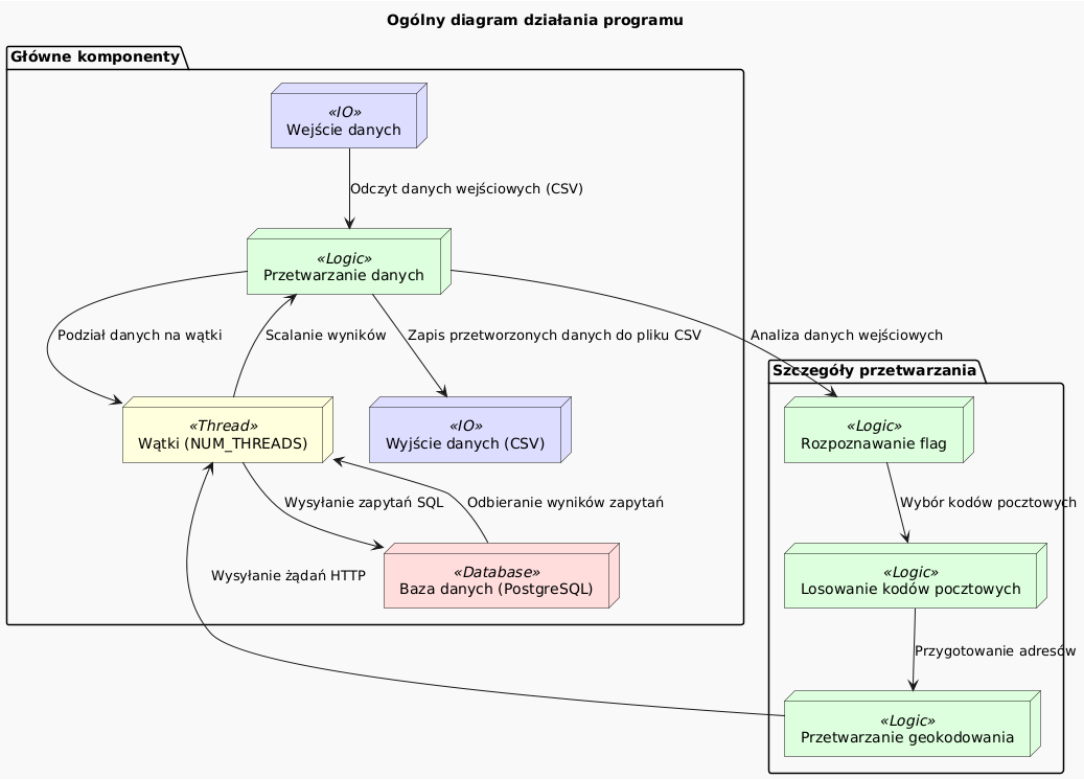
Wymagania systemowe:

- System operacyjny Linux,
- Kompilator obsługujący standard C++17 lub nowszy,
- Zainstalowana baza danych PostgreSQL z PostGIS,
- Biblioteki: `pqxx`, `nlohmann::json`, `csvstream`.

1.2 Funkcjonalności aplikacji

1.2.1 Opis działania programu na podstawie diagramu

Diagram przedstawia strukturę i przepływ danych w aplikacji. Składa się z dwóch głównych sekcji: komponentów głównych, które odpowiadają za ogólny przepływ danych w systemie, oraz szczegółowych etapów przetwarzania, które opisują wewnętrzne operacje wykonywane na danych.



Rysunek 1.1: Diagram przedstawiający przepływ danych w aplikacji.

Jak pokazano na rysunku 1.1, aplikacja składa się z kilku głównych komponentów, które współpracują w celu przetwarzania danych adresowych. Poniżej opisano każdy z tych komponentów.

Główne komponenty systemu

Diagram zawiera sekcję zatytułowaną "Główne komponenty", w której przedstawiono podstawowe elementy programu oraz sposób, w jaki dane przepływają między nimi:

- **Wejście danych**: Komponent "Wejście danych"reprezentuje proces wczytywania danych wejściowych z pliku CSV. Dane te zawierają informacje o adresach, które zostaną przetworzone przez system. Proces odczytu danych jest pierwszym krokiem w przepływie, który dostarcza informacje do kolejnego komponentu, czyli "Przetwarzania danych".

- **Przetwarzanie danych**: Komponent "Przetwarzanie danych"jest odpowiedzialny za główną logikę programu. Dane wczytane z pliku CSV są analizowane, przetwarzane i dzielone na mniejsze podzbiory, które następnie są przekazywane do wątków w celu równoległego przetwarzania. Ten etap obejmuje operacje takie jak analiza poprawności danych, przypisywanie flag oraz przygotowanie danych do dalszych operacji.

- **Wątki (NUM_THREADS)**: Komponent "Wątki"reprezentuje wielowątkowe przetwarzanie danych, które pozwala na równoległe wykonywanie operacji na różnych fragmentach zbioru danych. Każdy wątek komunikuje się z bazą danych PostgreSQL, wysyłając zapytania SQL w celu uzyskania brakujących informacji lub weryfikacji istniejących danych.

- **Baza danych (PostgreSQL)**: Komponent "Baza danych"odpowiada za przechowywanie informacji geograficznych, takich jak współrzędne geograficzne (szerokość i długość geograficzna) dla adresów. Wątki wysyłają zapytania SQL do bazy danych, a wyniki tych zapytań są zwracane do odpowiednich wątków w celu dalszego przetwarzania.

- **Wyjście danych (CSV)**: Komponent "Wyjście danych"reprezentuje proces zapisywania przetworzonych danych do pliku CSV. Po zakończeniu wszystkich operacji dane są scalane i zapisane w uporządkowanym formacie, który zawiera zarówno oryginalne informacje, jak i uzupełnione dane, takie jak współrzędne geograficzne.

Szczegóły przetwarzania danych

W diagramie znajduje się również sekcja "Szczegóły przetwarzania", która opisuje bardziej szczegółowe etapy przetwarzania danych, wykonywane wewnątrz komponentu "Przetwarzanie danych":

- **Rozpoznawanie flag**: W pierwszym kroku szczegółowego przetwarzania dane wejściowe są analizowane w celu określenia, które informacje są obecne, a które brakuje. Na podstawie tej analizy każdemu rekordowi przypisywana jest flaga, która wskazuje, jakie operacje należy wykonać. Na przykład, jeśli brakuje kodu pocztowego, ale miasto i ulica są dostępne, rekord otrzymuje flagę oznaczającą konieczność uzupełnienia brakującego kodu pocztowego.

- **Losowanie kodów pocztowych**: W przypadku brakujących kodów pocztowych program przeprowadza losowanie na podstawie rozkładu prawdopodobieństwa obliczonego z dostępnych danych. Rozkład ten jest tworzony na podstawie liczby wystąpień poszczególnych kodów pocztowych w danych wejściowych. Wynik losowania jest przypisywany do odpowiednich rekordów, aby uzupełnić brakujące informacje.

- **Przetwarzanie geokodowania**: Kolejnym krokiem jest przygotowanie adresów do wysyłania żądań HTTP do serwera geokodowania. Adresy są kodowane w odpowiednim formacie, a następnie wysyłane w celu uzyskania współrzędnych geograficznych, takich jak szerokość i długość geograficzna. Wyniki są odbierane przez wątki i przypisywane do odpowiednich rekordów.

- **Komunikacja z wątkami**: Po zakończeniu przetwarzania geokodowania dane są przekazywane do wątków, które realizują równoległe operacje, takie jak wysyłanie zapytań SQL do bazy danych PostgreSQL. Wątki odbierają wyniki zapytań i integrują je z danymi przetwarzanymi w systemie.

Podsumowanie

Diagram przedstawia kompleksowy przepływ danych w programie, który łączy w sobie operacje wejścia/wyjścia, wielowątkowość, komunikację z bazą danych oraz obsługę żądań HTTP. Dzięki logicznemu podziałowi na główne komponenty oraz szczegółowe etapy przetwarzania, program jest w stanie efektywnie przetwarzać duże zbiory danych adresowych, uzupełniać brakujące informacje i zapisywać wyniki w uporządkowanej formie. Kluczowym elementem jest wykorzystanie flag do klasyfikacji danych oraz losowanie kodów pocztowych w przypadku brakujących informacji, co pozwala na automatyczne uzupełnianie danych wejściowych.

1.2.2 Wczytywanie danych wejściowych

Program wczytuje dane z pliku CSV. Każdy rekord zawiera informacje o adresie (ulica, kod pocztowy, miasto, województwo, kraj) oraz dodatkowe dane, takie jak numer umowy, data rozpoczęcia i zakończenia.

Wczytane dane są następnie przetwarzane w celu przygotowania ich do geokodowania:

- Adresy są analizowane i klasyfikowane na podstawie dostępnych danych (np. brak ulicy, brak kodu pocztowego).
- Rekordy są oznaczane flagami (`flaga1`, `flaga2`), które określają, jakie dane są dostępne i jakie przetwarzanie będzie wymagane.
- W przypadku brakujących danych, rekordy są dodawane do wektora `rozklad`, gdzie później zostaną uzupełnione.

1.2.3 Geokodowanie adresów

Aplikacja przetwarza dane adresowe i wzbogaca je o:

- Szerokość i długość geograficzną,
- Nazwę miejsca (np. pełny adres w formacie tekstowym).

W przypadku brakujących danych program korzysta z losowego uzupełniania na podstawie rozkładu prawdopodobieństwa.

1.2.4 Przetwarzanie danych adresowych

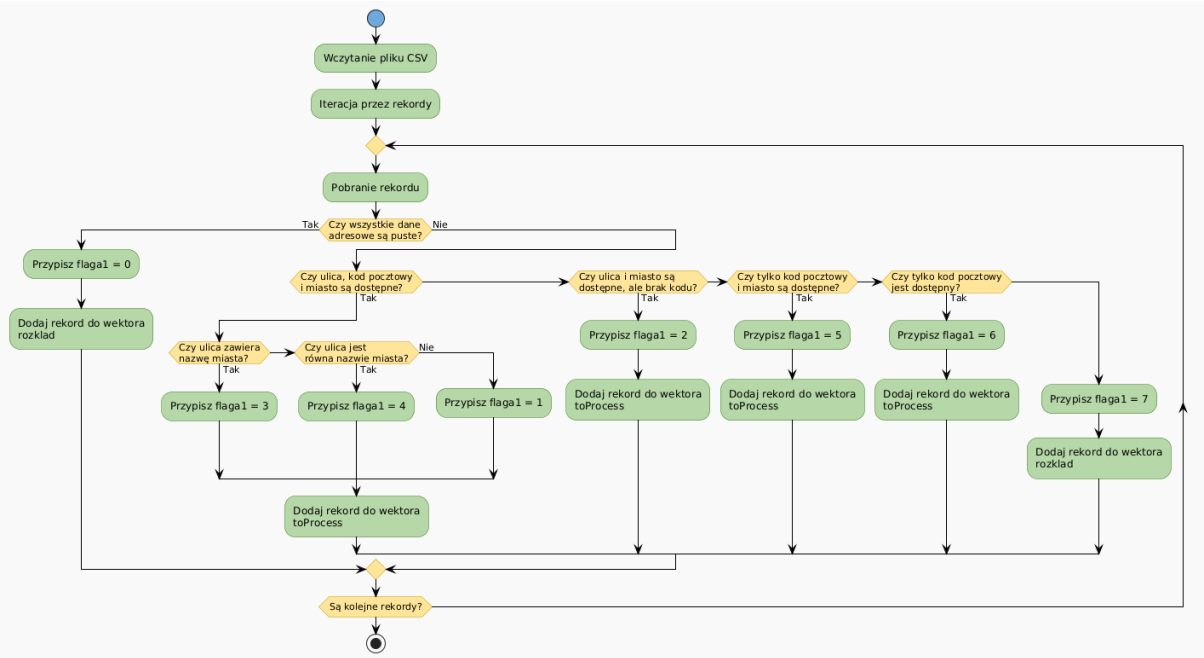
Proces przetwarzania danych adresowych składa się z następujących kroków:

1. **Inicjalizacja zmiennych**: Przygotowanie struktur danych i wektorów na potrzeby przetwarzania.
2. **Wczytanie pliku CSV**: Odczyt danych wejściowych z pliku.
3. **Iteracja przez rekordy**: Pobranie rekordu z pliku i jego przetwarzanie krok po kroku.
4. **Przypisywanie flag**: Każdy rekord jest analizowany na podstawie dostępności danych adresowych (ulica, kod pocztowy, miasto), a następnie przypisywana jest odpowiednia wartość flagi `flaga1`. Opis poszczególnych wartości flag:
   - `flaga1 = 0`: Brak wszystkich kluczowych danych adresowych (ulica, kod pocztowy, miasto są puste).
   - `flaga1 = 1`: Dane są częściowo dostępne, ale nie pasują do żadnego z pozostałych warunków.
   - `flaga1 = 2`: Ulica i miasto są dostępne, ale kod pocztowy jest pusty, a ulica nie jest taka sama jak miasto.
   - `flaga1 = 3`: Wszystkie kluczowe dane są dostępne, ale ulica zawiera nazwę miasta.
   - `flaga1 = 4`: Wszystkie kluczowe dane są dostępne, ale ulica jest równa nazwie miasta.
   - `flaga1 = 5`: Kod pocztowy i miasto są dostępne, ale ulica jest pusta.
   - `flaga1 = 6`: Kod pocztowy jest dostępny, ale zarówno ulica, jak i miasto są puste.
   - `flaga1 = 7`: Wszystkie dane adresowe są puste (inny przypadek niż `flaga1 = 0`).
5. **Przypisywanie flagi** `flaga2`: Flaga `flaga2` jest domyślnie ustawiana na wartość `0`. Jest używana do oznaczania rekordów gotowych do dalszego przetwarzania.

6. **Klasyfikacja rekordów**: Na podstawie wartości flagi `flaga1`, rekordy są klasyfikowane i dodawane do odpowiednich struktur danych:
   - Rekordy o wartościach `flaga1 = 0` lub `flaga1 = 7` są dodawane do wektora `rozklad`.
   - Rekordy z innymi wartościami `flaga1` są dodawane do wektora `toProcess`.

1.2.5 Diagram przepływu przetwarzania danych

Na Rysunku 1.2 przedstawiono diagram przepływu przetwarzania danych wejściowych. Diagram ilustruje proces wczytywania danych z pliku CSV, klasyfikacji rekordów na podstawie dostępnych danych oraz przygotowania ich do dalszego przetwarzania.



Rysunek 1.2: Diagram przepływu przetwarzania danych wejściowych.

Opis diagramu Diagram przedstawia proces przetwarzania rekordów adresowych z pliku CSV. Kroki są następujące:

1. **Inicjalizacja zmiennych**: Przygotowanie struktur danych i wektorów (`rozklad` oraz `toProcess`) na potrzeby przetwarzania.
2. **Wczytanie pliku CSV**: Odczytanie danych wejściowych z pliku CSV.
3. **Iteracja przez rekordy**: Przeglądanie każdego rekordu w pliku CSV w pętli.
4. **Sprawdzenie, czy wszystkie dane adresowe są puste**:
   - Jeśli tak, rekord jest klasyfikowany jako niekompletny (`flaga1 = 0`) i dodawany do wektora `rozklad`.
   - W przeciwnym razie, przechodzi do dalszego przetwarzania.
5. **Sprawdzenie, czy dostępne są wszystkie dane adresowe (ulica, kod pocztowy, miasto)**:
   - Jeśli tak, następuje dodatkowa analiza:

- Jeśli ulica zawiera nazwę miasta, rekord otrzymuje `flaga1 = 3`.
- Jeśli ulica jest równa nazwie miasta, rekord otrzymuje `flaga1 = 4`.
- W przeciwnym razie, rekord otrzymuje `flaga1 = 1`.

Rekord jest następnie dodawany do wektora `toProcess`.

6. **Sprawdzenie, czy dostępne są ulica i miasto, ale brak kodu pocztowego**:
   - Jeśli tak, rekord otrzymuje `flaga1 = 2` i jest dodawany do wektora `toProcess`.
7. **Sprawdzenie, czy dostępne są tylko kod pocztowy i miasto**:
   - Jeśli tak, rekord otrzymuje `flaga1 = 5` i jest dodawany do wektora `toProcess`.
8. **Sprawdzenie, czy dostępny jest tylko kod pocztowy**:
   - Jeśli tak, rekord otrzymuje `flaga1 = 6` i jest dodawany do wektora `toProcess`.
9. **Pozostałe przypadki**: Jeśli żadne z powyższych kryteriów nie zostało spełnione, rekord otrzymuje `flaga1 = 7` i jest dodawany do wektora `rozklad`.
10. **Zakończenie iteracji**: Proces powtarza się dla każdego rekordu w pliku CSV, aż wszystkie zostaną przetworzone.

1.2.6 Przetwarzanie wielowątkowe

Program wykorzystuje przetwarzanie wielowątkowe w celu zwiększenia wydajności operacji na dużych zbiorach danych. Domyślnie uruchamiane są 32 wątki, które równolegle wykonują różne zadania, takie jak przetwarzanie danych wejściowych, wysyłanie żądań HTTP czy zapis wyników.

Główne elementy przetwarzania wielowątkowego:

1. **Inicjalizacja danych wejściowych:**
   - Dane są wczytywane z pliku CSV, a następnie klasyfikowane na podstawie ich zawartości (np. brakujące dane adresowe, kompletne dane, itp.).
   - Każdy rekord jest przypisywany do odpowiedniego wektora (`toProcess`, `rozklad`, `flaga1`, `flaga2`, itd.) w zależności od jego klasyfikacji.
   - Wektory są także podzielone na części odpowiadające liczbie wątków (`all`, `all2`, `all3`, `all4`) w celu równoległego przetwarzania.
2. **Podział pracy:**
   - Dane do przetwarzania są dzielone na fragmenty odpowiadające liczbie wątków (`NUM_THREADS = 32`).
   - Każdy wątek otrzymuje zakres rekordów do przetworzenia, co pozwala na równoległe operacje.
3. **Równoległe przetwarzanie danych:**
   - Każdy wątek wykonuje funkcję `perform_requests`, która obsługuje przetwarzanie rekordów w zakresie przypisanym do danego wątku.
   - Wątki komunikują się z serwerem za pomocą asynchronicznych żądań HTTP, wykorzystując gniazda w trybie nieblokującym oraz mechanizm `epoll`.

- Odpowiedzi serwera są analizowane, a wyniki (np. współrzędne geograficzne, nazwa miejsca) są przypisywane do odpowiednich rekordów.

4. **Obsługa brakujących danych:**

   - Rekordy z brakującymi danymi są przetwarzane w dodatkowych iteracjach, aż do uzyskania wymaganych informacji (np. kodu pocztowego lub współrzędnych geograficznych).
   - W przypadku braku danych, rekordy są klasyfikowane i przypisywane do odpowiednich wektorów (np. `flaga2`, `flaga3`).

5. **Zapis wyników:**

   - Po zakończeniu przetwarzania dane są zapisywane w pliku CSV za pomocą funkcji `saveToCSV`.
   - Plik wynikowy zawiera dodatkowe kolumny, takie jak współrzędne geograficzne (`Szerokosc`, `Dlugosc`), flaga przetwarzania (`Flaga_1`, `Flaga_2`) oraz numer województwa (`Nr_Wojewodztwa`).

Szczegóły implementacyjne:

- Program korzysta z bibliotek standardowych C++ (`<thread>`, `<mutex>`, `<vector>`, `<atomic>`) do obsługi wątków i synchronizacji.
- Mechanizm `epoll` jest wykorzystywany do obsługi asynchronicznych żądań HTTP w trybie nieblokującym.
- Dane wejściowe są wczytywane z pliku CSV za pomocą biblioteki `csvstream.hpp`.
- Wyniki są analizowane za pomocą biblioteki `nlohmann::json` do parsowania odpowiedzi w formacie JSON.
- Synchronizacja dostępu do współdzielonych struktur danych (np. `occurrences`, `workers_data`) jest zapewniona za pomocą mutexów (`std::mutex`).

Podział pracy na wątki: Dane są dzielone na fragmenty odpowiadające liczbie wątków (`NUM_THREADS = 32`). Zakresy danych są obliczane na podstawie liczby rekordów i liczby wątków:

```
int M = NUM_THREADS;       // liczba wątków
int N = toProcess.size();  // liczba rekordów do przetworzenia
std::vector<std::pair<int, int>> ranges;
int chunkSize = N / M;

for (int i = 0; i < M; ++i) {
    int start = i * chunkSize;
    int end = (i == M - 1) ? N : start + chunkSize;
    ranges.push_back({start, end});
}
```

Każdy wątek przetwarza dane w zakresie `[start, end)`.

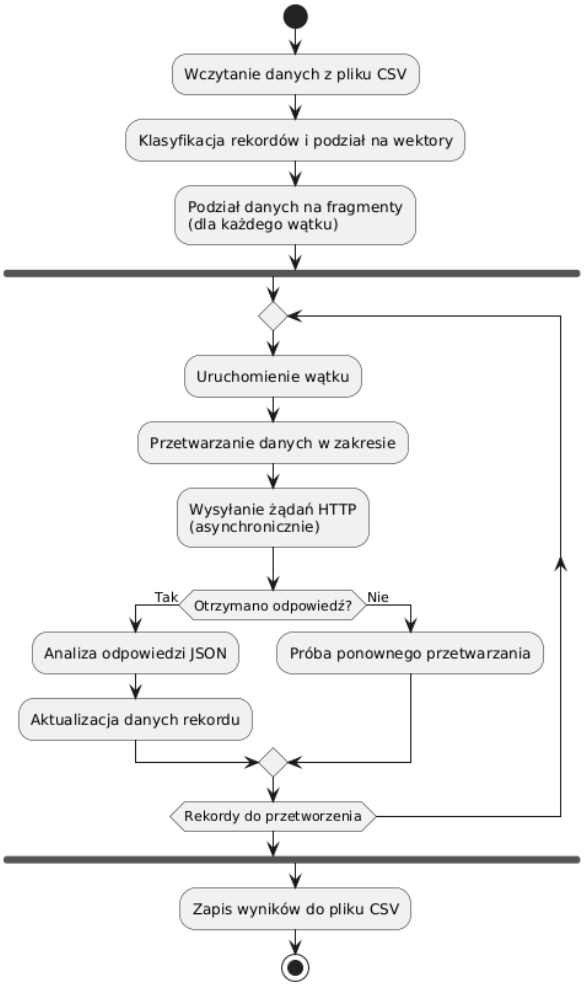
Mechanizm przetwarzania żądań HTTP: Wątki wysyłają żądania HTTP do serwera za pomocą funkcji `create_nonblocking_socket` i `epoll`. Żądania są wysyłane w formacie:

```
GET /search.php?q=<adres>&format=json&limit=1 HTTP/1.1
Host: 127.0.0.1:8080
Connection: keep-alive
```

Odpowiedzi w formacie JSON są parsowane za pomocą biblioteki `nlohmann::json`, a wyniki (np. współrzędne geograficzne) są przypisywane do odpowiednich rekordów.

Zapis wyników: Dane są zapisywane w pliku CSV w formacie:

```
Lp;DataPoczatku;DataKonca;SumaUbezpieczenia;Odnowienia;
Ulica;KodPocztowy;Miasto;Wojewodztwo;Kraj;ReasekuracjaO;
ReasekuracjaF;Szerokosc;Dlugosc;Flaga_1;Flaga_2;Nr_Wojewodztwa
```



Rysunek 1.3: Proces przetwarzania wielowątkowego

1.2.7 Zapis wyników

Wyniki przetwarzania są zapisywane w pliku CSV z dodatkowymi kolumnami, takimi jak współrzędne geograficzne, nazwa miejsca oraz flaga przetwarzania. Funkcja `saveToCSV` odpowiada za iterację przez przetworzone dane i zapisanie ich w odpowiednim formacie do pliku.

Opis działania funkcji:

1. **Otwarcie pliku**: Funkcja próbuje otworzyć plik o nazwie podanej w parametrze `filename`. Jeśli plik nie może zostać otwarty, zwracany jest komunikat o błędzie i funkcja kończy swoje działanie.

2. **Zapis nagłówka**: Do pliku zapisywany jest nagłówek zawierający nazwy kolumn, takie jak: `Lp`, `DataPoczatku`, `DataKonca`, `Ulica`, `KodPocztowy`, `Miasto`, `Wojewodztwo`, `Kraj`, `Szerokosc`, `Dlugosc`, `Flaga_1`, `Flaga_2`, `Nr_Wojewodztwa`.

3. **Iteracja przez dane**: Funkcja iteruje przez wektor `dataToCSV`, który zawiera przetworzone rekordy adresowe.

4. **Przypisanie województwa i numeru województwa**: Dla każdego rekordu wywoływana jest funkcja `getWojewodztwoMapa`, która na podstawie kodu pocztowego przypisuje nazwę województwa oraz jego numer.

5. **Zapis danych do pliku**: Każdy rekord jest zapisywany w formacie CSV, gdzie poszczególne pola są oddzielone średnikami. Zawartość rekordu obejmuje dane adresowe, współrzędne geograficzne, flagi przetwarzania oraz numer województwa.

6. **Zamknięcie pliku**: Po zapisaniu wszystkich rekordów plik zostaje zamknięty.

Nagłówek pliku CSV:

```
Lp;DataPoczatku;DataKonca;SumaUbezpieczenia;Odnowienia;Ulica;KodPocztowy;Miasto;
Wojewodztwo;Kraj;ReasekuracjaO;ReasekuracjaF;Szerokosc;Dlugosc;Flaga_1;Flaga_2;Nr_Wojewodztwa
```

Pola danych: Każdy zapisany rekord zawiera następujące pola:

- `Lp` – liczba porządkowa rekordu.
- `DataPoczatku`, `DataKonca` – daty związane z okresem obowiązywania danych.
- `SumaUbezpieczenia`, `Odnowienia` – dane finansowe.
- `Ulica`, `KodPocztowy`, `Miasto`, `Wojewodztwo`, `Kraj` – dane adresowe.
- `ReasekuracjaO`, `ReasekuracjaF` – informacje o reasekuracji.
- `Szerokosc`, `Dlugosc` – współrzędne geograficzne.
- `Flaga_1`, `Flaga_2` – flagi przetwarzania.
- `Nr_Wojewodztwa` – numer województwa.

1.2.8 Statystyki i analiza

Program analizuje częstość występowania kodów pocztowych w danych wejściowych i wykorzystuje te statystyki do losowego uzupełniania brakujących informacji. Proces ten jest realizowany w następujących krokach:

Analiza kodów pocztowych:

- Na podstawie danych wejściowych zliczana jest liczba wystąpień każdego kodu pocztowego.
- Częstości występowania są normalizowane, aby stworzyć rozkład prawdopodobieństwa dla każdego kodu pocztowego. Rozkład ten jest przechowywany w wektorze `probabilities`, a odpowiadające mu kody w wektorze `codes`.
- Rozkład prawdopodobieństwa jest wykorzystywany do uzupełniania brakujących kodów pocztowych w rekordach.

Losowe uzupełnianie brakujących informacji:

- Dla każdego rekordu z brakującymi danymi (np. kodem pocztowym lub miastem) program analizuje jego flagę (`flaga1`) i podejmuje odpowiednie działania:
  - Jeśli rekord ma brakujący kod pocztowy (`flaga1 == 0`), jest on losowany na podstawie rozkładu prawdopodobieństwa.
  - Jeśli rekord ma brakujące miasto (`flaga1 == 5`) lub inne brakujące dane, podejmowane są dodatkowe próby uzupełnienia danych.
- Program śledzi liczbę prób (`proby`) dla każdego rekordu, aby uniknąć nieskończonych iteracji.
- W przypadku braku możliwości uzupełnienia danych rekord jest oznaczany odpowiednią flagą i przechowywany w specjalnym wektorze.

Przetwarzanie danych w pętli:

- Program działa w pętli, w której rekordy są przetwarzane iteracyjnie, aż do wyczerpania danych do przetworzenia.
- Dla każdego rekordu:
  1. Wyciągane są numery pozycji (`vecPosNumbers`) z zapytania SQL.
  2. Jeśli rekordy zawierają brakujące dane, są one przetwarzane za pomocą funkcji losującej kod pocztowy. Losowanie odbywa się z wykorzystaniem klasy `std::discrete_distribution`, która korzysta z wektora `probabilities`.
  3. Wyniki są zapisywane w strukturze `toProcess`, a dodatkowe wystąpienia są rejestrowane w mapie `temp_occurrences`.
- Jeśli dane nadal są niekompletne, rekord jest ponownie przetwarzany w kolejnych iteracjach.

Przykład działania algorytmu: Przetwarzanie rekordu z brakującym kodem pocztowym (`flaga1 == 0`):

```
if (toProcess[num].flaga1 == 0) {
    std::discrete_distribution<> dist(probabilities.begin(), probabilities.end());
    std::string pCode = codes[dist(gen)];
    toProcess[num].kodPocztowy = pCode;
    addOccurrenceTemp(temp_occurrences, "", pCode, 0, num);
}
```

Przetwarzanie rekordu z brakującym miastem (`flaga1 == 5`):

```
if (toProcess[num].flaga1 == 5 && toProcess[num].proby == 0) {
    toProcess[num].proby += 1;
    addOccurrenceTemp(temp_occurrences, toProcess[num].miasto, "", 5, num);
}
```

Zalety podejścia:

- Wykorzystanie statystyk kodów pocztowych pozwala na bardziej realistyczne uzupełnianie brakujących danych.
- Iteracyjne przetwarzanie i śledzenie liczby prób dla każdego rekordu minimalizuje ryzyko błędów i nieskończonych pętli.
- Program jest skalowalny i może obsługiwać duże zbiory danych dzięki wykorzystaniu wielowątkowości i mechanizmu `std::discrete_distribution`.

1.3 Struktura kodu

1.3.1 Główne komponenty

Kod aplikacji jest podzielony na kilka głównych komponentów:

- **Struktury danych:**
  - `Address` – struktura przechowująca dane adresowe, takie jak ulica, kod pocztowy, miasto, województwo, współrzędne geograficzne, flagi przetwarzania oraz inne szczegóły,
  - `ThreadSafeDeque` – bezpieczna wątkowo kolejka FIFO, używana do przechowywania i przetwarzania zapytań SQL w środowisku wielowątkowym.
- **Funkcje:**
  - `addOccurrence` – dodaje informacje o wystąpieniu adresu do mapy `occurrences`,
  - `getWojewodztwoMapa` – określa województwo na podstawie kodu pocztowego,
  - `url_encode` – koduje ciąg znaków w formacie URL, aby przygotować go do wysyłania w zapytaniach HTTP,
  - `perform_requests` – obsługuje zapytania HTTP w celu pobrania brakujących danych adresowych,
  - `perform_random` – iteracyjnie przetwarza dane z kolejki `workers_data`, uzupełniając brakujące informacje na podstawie statystyk i zapytań SQL,
  - `saveToCSV` – zapisuje przetworzone dane do pliku CSV w określonym formacie,
  - `removeWord`, `removeWordsWithDot`, `removeAfterSlash` – funkcje pomocnicze do czyszczenia i przetwarzania danych tekstowych,
  - `create_nonblocking_socket` – tworzy nieblokujące gniazdo sieciowe dla zapytań HTTP,
  - `addOccurrenceTemp` – dodaje tymczasowe wystąpienie adresu do lokalnej mapy w trakcie przetwarzania.
- **Zmienne globalne:**
  - `toProcess` – wektor przechowujący dane do przetworzenia,
  - `postalCodeCount` – mapa przechowująca liczbę wystąpień kodów pocztowych w danych wejściowych,
  - `occurrences` – mapa przechowująca dane o wystąpieniach adresów do dalszego przetwarzania,
  - `codes` – wektor przechowujący unikalne kody pocztowe,
  - `probabilities` – wektor przechowujący prawdopodobieństwa wystąpień kodów pocztowych, obliczone na podstawie danych wejściowych,
  - `workers_data` – wątkowo bezpieczna kolejka, zawierająca zapytania SQL do przetwarzania przez wątki,
  - `dataToCSV` – wektor przechowujący dane gotowe do zapisu w pliku CSV.
- **Wątki:**
  - Aplikacja wykorzystuje `NUM_THREADS` równoległych wątków do przetwarzania danych, co pozwala na efektywne wykorzystanie zasobów obliczeniowych.

1.3.2 Przebieg programu

1. Inicjalizacja:

- Program rozpoczyna od zainicjalizowania zmiennych globalnych, takich jak `postalCodeCount`, `occurrences`, `toProcess` oraz innych struktur danych.
- Dane wejściowe są wczytywane z pliku CSV (`Szkody2.csv`) za pomocą klasy `csvstream`.
- Każdy rekord jest analizowany i klasyfikowany na podstawie kompletności danych (np. brak kodu pocztowego, brak miasta).

2. Analiza kodów pocztowych:

- Program zlicza wystąpienia kodów pocztowych i oblicza ich prawdopodobieństwa, które są przechowywane w wektorze `probabilities`.
- Na podstawie tych prawdopodobieństw brakujące kody pocztowe są losowane przy użyciu klasy `std::discrete_distribution`.

3. Przetwarzanie danych:

- Dane są podzielone na zakresy i rozdzielane pomiędzy wątki.
- Każdy wątek wykonuje zapytania HTTP lub SQL w celu uzupełnienia brakujących informacji, takich jak współrzędne geograficzne, kod pocztowy czy miasto.
- W przypadku niepowodzenia dane są ponownie przetwarzane w kolejnych iteracjach.

4. Zapis wyników:

- Po zakończeniu przetwarzania dane są zapisywane do pliku CSV w określonym formacie, zawierającym m.in. kod pocztowy, miasto, województwo, współrzędne geograficzne i inne szczegóły.

1.3.3 Kluczowe fragmenty kodu

Struktura `Address`:

```
struct Address
{
    std::string lp;
    std::string ulica;
    std::string kodPocztowy;
    std::string miasto;
    std::string wojewodztwo;
    std::string kraj;
    std::string lot;
    std::string lat;
    int flaga1;
    int flaga2;
    std::string sklejone;
    std::string numerUmowy;
    std::string dataPoczatku;
```

```
    std::string dataKonca;
    std::string sumaUbezpieczenia;
    std::string odnowienia;
    std::string reasekuracjaO;
    std::string reasekuracjaF;
    std::string adresujedn;
    int proby = 0;
    bool dbProcess = false;
};
```

Funkcja `addOccurrence`**:**

```
void addOccurrence(const std::string &city, const std::string &postcode,
                   int flag, int vecPos)
{
    if (!city.empty() || !postcode.empty())
    {
        std::lock_guard<std::mutex> lock(occurrencesMutex);
        auto &entry = occurrences[{city, postcode, flag}];
        entry.first++;
        entry.second += std::to_string(vecPos) + " ";
    }
}
```

Funkcja `perform_requests`**:**

```
void perform_requests(int thread_id,
                      std::shared_ptr<std::vector<std::string>> &data,
                      std::pair<int, int> range)
{
    int NUM_REQUESTS = range.second - range.first;
    int epoll_fd = epoll_create1(0);
    if (epoll_fd == -1)
    {
        perror("epoll_create1");
        exit(1);
    }
    int sockfd = create_nonblocking_socket(HOST, PORT);
    if (sockfd == -1)
    {
        exit(1);
    }
    epoll_event ev, events[NUM_REQUESTS];
    ev.events = EPOLLOUT | EPOLLET;
    ev.data.fd = sockfd;
    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, sockfd, &ev) == -1)
    {
        perror("epoll_ctl: EPOLLOUT");
```

```
        close(sockfd);
        exit(1);
    }
    // Obsługa zapytań HTTP i przetwarzania odpowiedzi...
}
```

1.3.4 Efektywność i skalowalność

- Program jest zoptymalizowany pod kątem wielowątkowości, co pozwala na równoległe przetwarzanie dużych zbiorów danych.
- Wykorzystanie asynchronicznych zapytań HTTP i SQL minimalizuje czas oczekiwania na odpowiedzi.
- Struktury takie jak `ThreadSafeDeque` oraz mechanizmy synchronizacji (np. `std::mutex`) zapewniają bezpieczeństwo wątkowe.

1.3.5 Główne klasy i struktury

Struktura `Address`

```
struct Address {
    std::string lp;
    std::string ulica;
    std::string kodPocztowy;
    std::string miasto;
    std::string wojewodztwo;
    std::string kraj;
    std::string lot;
    std::string lat;
    int flaga1;
    int flaga2;
    std::string sklejone;
    std::string numerUmowy;
    std::string dataPoczatku;
    std::string dataKonca;
    std::string sumaUbezpieczenia;
    std::string odnowienia;
    std::string reasekuracjaO;
    std::string reasekuracjaF;
    std::string adresujedn;
    int proby = 0;
    bool dbProcess = false;
};
```

Klasa `ThreadSafeDeque`

Kolejka FIFO bezpieczna wątkowo, używana do przechowywania zapytań SQL.

```
class ThreadSafeDeque {
public:
    std::stringstream pop_front();
    void push_back(const std::stringstream &value);
    size_t size() const;

private:
```

```
8       std::deque<std::string> data;
9       std::mutex mtx;
10  };
```

1.4 Obsługa sieci

1.4.1 Tworzenie asynchronicznego gniazda

Funkcja `create_nonblocking_socket` tworzy gniazdo TCP/IP w trybie nieblokującym, używając biblioteki `<sys/socket.h>` oraz `<fcntl.h>`. Jest to kluczowe dla obsługi wielu połączeń jednocześnie, co pozwala na efektywne przetwarzanie zapytań HTTP w środowisku wielowątkowym.

Listing 1.1: Funkcja tworząca asynchroniczne gniazdo

```c
int create_nonblocking_socket(const char *host, const char *port) {
    struct addrinfo hints, *res;
    int sockfd;

    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;

    if (getaddrinfo(host, port, &hints, &res) != 0) {
        perror("getaddrinfo");
        return -1;
    }

    sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
    if (sockfd == -1) {
        perror("socket");
        freeaddrinfo(res);
        return -1;
    }

    int flags = fcntl(sockfd, F_GETFL, 0);
    if (fcntl(sockfd, F_SETFL, flags | O_NONBLOCK) == -1) {
        perror("fcntl");
        close(sockfd);
        freeaddrinfo(res);
        return -1;
    }

    if (connect(sockfd, res->ai_addr, res->ai_addrlen) == -1) {
        if (errno != EINPROGRESS) {
            perror("connect");
            close(sockfd);
            freeaddrinfo(res);
            return -1;
        }
    }

    freeaddrinfo(res);
    return sockfd;
}
```

Opis działania:

- Funkcja korzysta z `getaddrinfo`, aby uzyskać informacje o adresie serwera na podstawie parametrów `host` i `port`.
- Gniazdo jest tworzone za pomocą funkcji `socket`, a następnie ustawiane w tryb nieblokujący przy użyciu `fcntl`.

- Połączenie jest inicjowane za pomocą funkcji `connect`. Jeśli połączenie nie może zostać ustanowione natychmiast, funkcja zwraca kod błędu `EINPROGRESS`, co jest zgodne z trybem nieblokującym.

1.4.2 Obsługa zapytań HTTP

Funkcja `perform_requests` obsługuje wysyłanie zapytań HTTP do serwera geokodowania przy użyciu mechanizmu `epoll`. Umożliwia to obsługę wielu zapytań jednocześnie w sposób asynchroniczny.

Listing 1.2: Funkcja obsługująca zapytania HTTP

```
void perform_requests(int thread_id,
                      std::shared_ptr<std::vector<std::string>> &data,
                      std::pair<int, int> range) {

    int NUM_REQUESTS = range.second - range.first;
    int epoll_fd = epoll_create1(0);
    if (epoll_fd == -1) {
        perror("epoll_create1");
        exit(1);
    }

    int sockfd = create_nonblocking_socket(HOST, PORT);
    if (sockfd == -1) {
        exit(1);
    }

    epoll_event ev, events[NUM_REQUESTS];
    ev.events = EPOLLOUT | EPOLLET;
    ev.data.fd = sockfd;

    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, sockfd, &ev) == -1) {
        perror("epoll_ctl: EPOLLOUT");
        close(sockfd);
        exit(1);
    }

    int requests_sent = 0;
    bool keep_going = true;

    while (keep_going) {
        int nfds = epoll_wait(epoll_fd, events, NUM_REQUESTS, -1);
        if (nfds == -1) {
            perror("epoll_wait");
            exit(1);
        }

        for (int n = 0; n < nfds; ++n) {
            if (events[n].events & EPOLLOUT && requests_sent < NUM_REQUESTS) {
                // Wysyłanie zapytania HTTP
            } else if (events[n].events & EPOLLIN) {
                // Odbieranie odpowiedzi
            }
        }
    }

    close(sockfd);
    close(epoll_fd);
}
```

Opis działania:

- Funkcja korzysta z mechanizmu `epoll`, aby monitorować zdarzenia związane z gniazdem.
- Zdarzenie `EPOLLOUT` oznacza gotowość do wysyłania danych (zapytania HTTP).
- Zdarzenie `EPOLLIN` oznacza dostępność danych do odczytu (odpowiedź HTTP).
- Mechanizm `epoll` pozwala na obsługę wielu zapytań jednocześnie w jednym wątku, co zwiększa wydajność.

1.4.3 Kodowanie adresów URL

Funkcja `url_encode` konwertuje ciąg znaków na format zgodny z URL, zamieniając niedozwolone znaki na ich reprezentację procentową (`%XX`).

Listing 1.3: Funkcja kodująca adres URL

```
std::string url_encode(const std::string &sSrc) {
    const char DEC2HEX[16 + 1] = "0123456789ABCDEF";
    const unsigned char *pSrc = (const unsigned char *)sSrc.c_str();
    const int SRC_LEN = sSrc.length();
    unsigned char *const pStart = new unsigned char[SRC_LEN * 3];
    unsigned char *pEnd = pStart;
    const unsigned char *const SRC_END = pSrc + SRC_LEN;

    for (; pSrc < SRC_END; ++pSrc) {
        if (SAFE[*pSrc])
            *pEnd++ = *pSrc;
        else {
            *pEnd++ = '%';
            *pEnd++ = DEC2HEX[*pSrc >> 4];
            *pEnd++ = DEC2HEX[*pSrc & 0x0F];
        }
    }

    std::string sResult((char *)pStart, (char *)pEnd);
    delete[] pStart;
    return sResult;
}
```

Opis działania:

- Funkcja iteruje po znakach ciągu wejściowego `sSrc`.
- Jeśli znak jest zgodny z formatem URL (np. litery, cyfry), jest kopiowany bez zmian.
- Niedozwolone znaki są zamieniane na ich reprezentację procentową (`%XX`), gdzie `XX` to wartość szesnastkowa znaku.

1.4.4 Efektywność i zalety

- Mechanizm `epoll` pozwala na asynchroniczne przetwarzanie wielu zapytań HTTP w jednym wątku, co minimalizuje opóźnienia wynikające z oczekiwania na odpowiedzi sieciowe.
- Kodowanie URL zapewnia poprawność zapytań HTTP, umożliwiając wysyłanie danych zawierających znaki specjalne.

- Ustawienie gniazda w trybie nieblokującym pozwala na wykonywanie innych operacji w trakcie oczekiwania na zdarzenia sieciowe.

1.5 Format danych

1.5.1 Dane wejściowe

Plik CSV wejściowy powinien zawierać dane adresowe oraz dodatkowe informacje, które są przetwarzane w aplikacji. Struktura pliku wejściowego powinna być zgodna z poniższymi kolumnami:

- `Lp` – numer porządkowy rekordu,
- `NumerUmowy` – numer identyfikujący umowę,
- `DataPoczatku` – data rozpoczęcia umowy,
- `DataKonca` – data zakończenia umowy,
- `SumaUbezpieczenia` – wartość sumy ubezpieczenia,
- `Odnowienia` – liczba odnowień umowy,
- `Ulica` – nazwa ulicy w adresie,
- `KodPocztowy` – kod pocztowy,
- `Miasto` – nazwa miejscowości,
- `Wojewodztwo` – województwo, w którym znajduje się adres,
- `Kraj` – nazwa kraju,
- `ReasekuracjaO` – informacje o reasekuracji obligatoryjnej,
- `ReasekuracjaF` – informacje o reasekuracji fakultatywnej.

Przykład zawartości pliku wejściowego:

Listing 1.4: Przykładowy plik CSV wejściowy

```
Lp;NumerUmowy;DataPoczatku;DataKonca;SumaUbezpieczenia;Odnowienia;Ulica;KodPocztowy;
    ↪ Miasto;Wojewodztwo;Kraj;ReasekuracjaO;ReasekuracjaF
1;12345;2023-01-01;2023-12-31;100000;1;Kwiatowa 10;00-001;Warszawa;Mazowieckie;Polska;
    ↪ Tak;Nie
2;12346;2023-02-01;2023-12-31;150000;0;Lipowa 5;30-001;Kraków;Małopolskie;Polska;Nie;Tak
```

1.5.2 Dane wyjściowe

Po przetworzeniu przez aplikację, dane wejściowe są wzbogacane o dodatkowe informacje uzyskane w wyniku geokodowania oraz przetwarzania. Plik wynikowy zawiera następujące kolumny:

- `Lp` – numer porządkowy rekordu (zgodny z danymi wejściowymi),
- `DataPoczatku`, `DataKonca`, `SumaUbezpieczenia`, `Odnowienia`, `Ulica`, `KodPocztowy`, `Miasto`, `Wojewodztwo`, `Kraj`, `ReasekuracjaO`, `ReasekuracjaF` – dane wejściowe bez zmian,
- `Szerokosc` – szerokość geograficzna uzyskana w wyniku geokodowania,
- `Dlugosc` – długość geograficzna uzyskana w wyniku geokodowania,
- `Adresujedn` – pełny adres w formacie tekstowym (np. `Kwiatowa 10, Warszawa, Polska`),

- `Flaga_1` – flaga wskazująca typ przetwarzania (np. brak danych, brak kodu pocztowego, brak miasta itp.),
- `Flaga_2` – flaga wskazująca liczbę prób przetwarzania (np. ilość zapytań do serwera geokodowania),
- `Nr_Wojewodztwa` – numer województwa przypisany na podstawie kodu pocztowego.

Listing 1.5: Przykładowy plik CSV wynikowy

```
Lp;DataPoczatku;DataKonca;SumaUbezpieczenia;Odnowienia;Ulica;KodPocztowy;Miasto;
    ↪ Wojewodztwo;Kraj;ReasekuracjaO;ReasekuracjaF;Szerokosc;Dlugosc;Flaga_1;Flaga_2;
    ↪ Nr_Wojewodztwa
1;2023-01-01;2023-12-31;100000;1;Kwiatowa 10;00-001;Warszawa;Mazowieckie;Polska;Tak;Nie
    ↪ ;52.2297;21.0122;1;0;7
2;2023-02-01;2023-12-31;150000;0;Lipowa 5;30-001;Kraków;Małopolskie;Polska;Nie;Tak
    ↪ ;50.0647;19.9450;1;0;6
```

1.5.3 Flagi przetwarzania

W aplikacji wykorzystano dwa typy flag, które są zapisywane w danych wynikowych:

- `Flaga_1` – określa typ brakujących danych lub metodę przetwarzania:
  - `0` – brak danych adresowych (ulica, kod pocztowy, miasto),
  - `1` – pełne dane adresowe (ulica, kod pocztowy, miasto),
  - `2` – brak kodu pocztowego, ale dostępne miasto i ulica,
  - `3` – brak miasta, ale dostępne kod pocztowy i ulica,
  - `4` – ulica i miasto są takie same,
  - `5` – brak ulicy, ale dostępne miasto i kod pocztowy,
  - `6` – brak miasta, dostępny tylko kod pocztowy.
- `Flaga_2` – liczba prób przetwarzania rekordu w przypadku błędów lub braków w danych.

1.5.4 Numer województwa

Numer województwa (`Nr_Wojewodztwa`) jest przypisywany na podstawie kodu pocztowego zgodnie z mapą województw:

- `Mazowieckie` – kody od 00-001 do 09-999 (`Nr_Wojewodztwa = 1`),
- `Małopolskie` – kody od 30-001 do 34-999 (`Nr_Wojewodztwa = 2`),
- `Śląskie` – kody od 40-001 do 44-999 (`Nr_Wojewodztwa = 3`),
- ...

Pełna mapa województw została zaimplementowana w funkcji `getWojewodztwoMapa`.