Politechnika Krakowska
im. Tadeusza Kościuszki

Wydział Inżynierii Elektrycznej i Komputerowej

Aplikacja usprawniająca pracę kuriera – problem komiwojażera w praktyce

Aleksander Krzeszowski

Promotor:
dr inż. Damian Grela

Kraków, 4 marca 2017

Panu dr inż. Damianowi Greli
składam serdeczne podziękowania
za poświęcony czas i pomoc
przy realizacji niniejszej pracy.

Spis treści

Wstęp

W literaturze można odnaleźć liczne prace skupiające się na analizie wydajności i optymalizacji różnych algorytmów rozwiązujących problem komiwojażera. Celem poniższej pracy jest stworzenie łatwej w obsłudze aplikacji, umożliwiającej odnalezienie optymalnej trasy łączącej wprowadzone miejsca docelowe. Założenie dostępności dla zwykłego użytkownika nakłada pewne wymagania na aplikację: interfejs musi być prosty w obsłudze, a aplikacja powinna być dostępna na różnych urządzeniach (komputery, tablety, urządzenia przenośne).

Wymagania te spełnia aplikacja internetowa – dostępna przez przeglądarkę. Taki rodzaj aplikacji pozwala na realizację architektury klient-serwer. W tym modelu obliczenia są wykonywane po stronie serwera, nie obciążając klienta, który wyświetla tylko interfejs aplikacji.

Kolejną konsekwencją przeznaczenia aplikacji dla zwykłych użytkowników jest konieczność zapewnienia odpowiedniej wydajności. Przetwarzanie danych powinno przebiegać w możliwie krótkim czasie, tak by użytkownik nie musiał czekać na zakończenie obliczeń. Znalezienie dokładnego rozwiązania problemu komiwojażera w czasie wielomianowym jest niemożliwe [8].

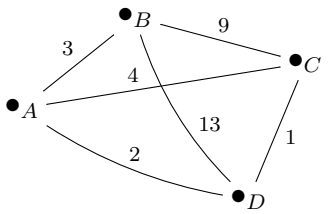
Poszukiwanie optymalnego rozwiązania już dla małej liczby miejsc docelowych wymagałoby znacznego czasu, nawet na wydajnym komputerze. Z tego powodu aplikacja powinna wykorzystywać algorytm sub-optymalny, który pozwala odnaleźć rozwiązanie w stosunkowo krótkim czasie, a odnalezione trasy są wystarczająco optymalne dla praktycznych zastosowań.

Rozdział 1

Definicja i rozwiązanie problemu

1.1 Określenie problemu

Problem komiwojażera polega na wyznaczeniu trasy łączącej wybrane punkty[1], przy dodatkowych warunkach: każdy punkt może zostać odwiedzony wyłącznie raz, poza wybranym punktem będącym początkiem i końcem trasy. Można więc powiedzieć, że rozwiązanie stanowi permutacja n punktów, a optymalnym rozwiązaniem jest permutacja o minimalnej sumie odległości między punktami[6].



Rysunek 1.1: Przykład symetryczny: Reprezentacja grafowa

Przykładowy problem został przedstawiony na grafie 1.1. Przyjmując B za punkt startowy, optymalną trasą dla takiego zbioru punktów jest na przykład: $B \to A \to D \to C \to B$ o długości 15.

Punkty pośrednie stanowią wierzchołki grafu, a trasy je łączące to krawędzie o wagach równych odległościom między punktami. Powyższy prosty przykład to wariant symetryczny problemu komiwojażera – odległości między dwoma punktami są identyczne w każdym kierunku. Dla takiego grafu wystarczy odnaleźć wagi dla $\frac{n(n-1)}{2}$ krawędzi, ponieważ są nieskierowane.

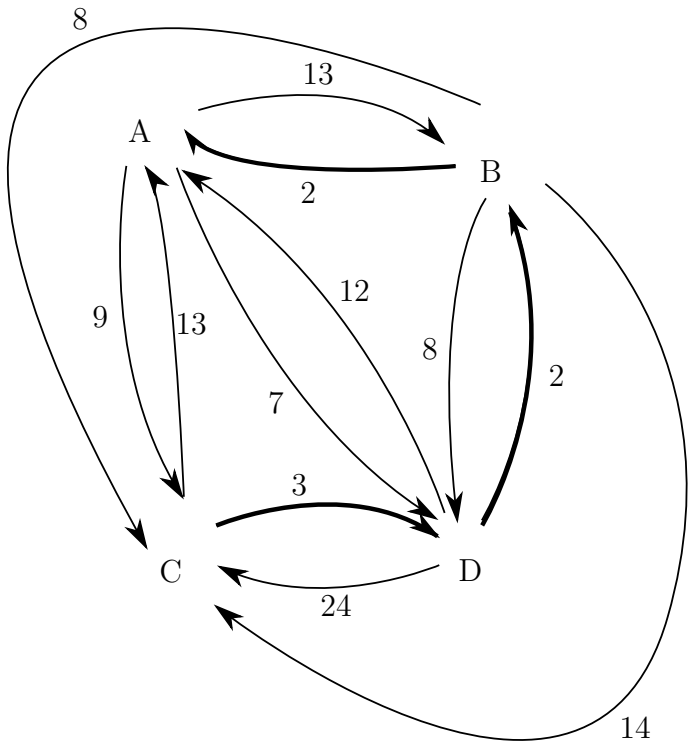
[1] Pierwotnie problem dotyczył tras między miastami, przez co w opracowaniach lub algorytmach punkty pośrednie często nazywa się miastami

	A	B	C	D
A	0	3	4	2
B	3	0	9	13
C	4	9	0	1
D	2	13	1	0

Tabela 1.1: Przykład symetryczny: Macierz sąsiedztwa

Źródło: Opracowanie własne

Jednak w rzeczywistym zastosowaniu (a także w zrealizowanej aplikacji) mamy do czynienia z asymetryczną wersją problemu, a więc ze skierowanym grafem. Jest to spowodowane tym, że co prawda z dowolnego punktu możemy dotrzeć do innego, jednak trasy między dwoma punktami mogą być inne (na przykład ulice jednokierunkowe). W rezultacie konieczne jest odnalezienie $n^2 - n$ odległości między punktami.



Rysunek 1.2: Przykład asymetryczny: Reprezentacja grafowa

	A	B	C	D
A	0	13	9	7
B	2	0	8	8
C	13	14	0	3
D	12	2	24	0

Tabela 1.2: Przykład asymetryczny: Macierz sąsiedztwa

Źródło: Opracowanie własne

Przykład problemu asymetrycznego znajduje się na grafie 1.2. Najkrótszą trasą rozpoczynającą się w punkcie C jest $C \rightarrow D \rightarrow B \rightarrow A \rightarrow C$ o długości 16.

Warto zauważyć że dla algorytmów nie ma znaczenia w jakiej jednostce jest wyrażona waga – można więc tym samym algorytmem optymalizować zarówno odległość, jak i czas przejazdu.

1.2 Algorytmy genetyczne

Algorytm genetyczny jest rodzajem algorytmu ewolucyjnego, zainspirowanego biologicznymi procesami ewolucji[6]. Do procesów tych należy dziedziczenie cech osobników, selekcja najsilniejszych (najlepiej przystosowanych) organizmów, których cechy zostaną połączone przez krzyżowanie, kończąc na mutacji będącej wprowadzeniem losowych zmian w genotypie. Do opisu powyższych procesów w algorytmice korzysta się z tych samych pojęć co w biologii.

Częstym zastosowaniem algorytmów genetycznych jest rozwiązywanie problemów optymalizacyjnych, takich jak wytyczanie trasy, projektowanie obwodów elektrycznych czy opisywany problem komiwojażera.

Ogólny przebieg programu ewolucyjnego (a zarazem genetycznego) został przedstawiony w algorytmie 1. W kolejnych podrozdziałach zostaną omówione poszczególne etapy zaimplementowanego programu.

Algorytm 1 Program ewolucyjny

```
1:  procedure Program Ewolucyjny
2:      t ← 0
3:      ustal początkowe P(t)                          ▷ Inicjalizacja
4:      while not warunek zakończenia do
5:          t ← t + 1
6:          wybierz P(t) z P(t − 1)                    ▷ Selekcja
7:          zmień P(t)                                 ▷ Krzyżowanie i mutacja
8:          oceń P(t)                                  ▷ Funkcja oceny
9:      end while
10: end procedure
```

Źródło: [6]

Ponieważ wyniki ewolucji dla standardowego schematu algorytmu genetycznego nie zawsze były zadowalające, została dodana pewna modyfikacja do podstawowego algorytmu – najlepszy osobnik z populacji jest przenoszony do następnego pokolenia, z pominięciem krzyżowania i mutacji. Ten rodzaj modyfikacji algorytmu nazywa się elitaryzmem.

Po dodaniu powyższej modyfikacji odnajdowane trasy były krótsze a najlepszy wynik był odnajdowany przy mniejszej liczbie pokoleń (program mógł działać krócej).

1.2.1 Inicjalizacja

Inicjalizacja polega na wygenerowaniu populacji złożonej z losowych osobników. W przypadku problemu komiwojażera populację stanowi zbiór tras, składających się z punktów pośrednich, dlatego należy spełnić warunek, że każdy element musi wystąpić dokładnie raz w wylosowanej trasie.

Najprostszym sposobem na spełnienie powyższego warunku jest przetasowanie zbioru wszystkich punktów. Opracowany program korzysta ze współczesnej wersji algorytmu Fishera-Yatesa[2]. Ma on złożoność $O(n)$ względem oryginalnego algorytmu o złożoności $O(n^2)$. Algorytm polega na wykonaniu n zamian między n-tym elementem zbioru, a losowym elementem, gdzie n jest równe liczbie elementów zbioru.

Wybór większego rozmiaru populacji zapewnia większą różnorodność, a w rezultacie zwiększa prawdopodobieństwo odnalezienia lepszego rozwiązania. Jednak zwiększanie tego parametru wydłuża czas działania algorytmu, co może być niepożądane przez użytkownika. Dlatego rozmiar populacji jest jednym z parametrów dostępnych do modyfikacji przez użytkownika – może on samodzielnie dobrać rozmiar odpowiadający wymaganej poprawności i czasowi przetwarzania.

1.2.2 Funkcja oceny

Ocena rozwiązania polega na przypisaniu wartości liczbowej danemu rozwiązaniu, tak aby dało się wybrać bardziej optymalne przy porównywaniu. Dla trasy jest to jej całkowita długość, czyli suma długości odcinków od wybranego punktu, między wszystkimi punktami pośrednimi i od ostatniego punktu pośredniego do punktu początkowego.

Punkty są przechowywane w aplikacji w postaci grafowej, więc odnalezienie długości trasy jest proste. Wszystkie punkty zawierają zbiór odległości do każdego innego punktu. Odległości są przechowywane w standardowej kolekcji `Dictionary<T>`, która jest zaimplementowana w postaci tablicy z haszowaniem, umożliwiającej odnalezienie elementu w średnim czasie $O(1)$. Dlatego obliczanie całkowitej odległości ma złożoność bliską liniowej.

Aplikacja umożliwia także wyszukiwanie tras bez określonego punktu początkowego. Przy tej samej liczbie wszystkich punktów skorzystanie z tej opcji zwiększa liczbę możliwych tras, co utrudnia odnalezienie tej najkrótszej.

Ponieważ kurierzy po dostarczeniu wszystkich przesyłek zazwyczaj nie muszą wracać do sortowni, dodana została również opcja wyłączenia powrotu do punktu początkowego.

1.2.3 Selekcja

Selekcja kandydatów do krzyżowania przebiega metodą turniejową[5]. Polega na wybraniu losowych osobników z populacji, a następnie zapisaniu najlepszego (według kryterium funkcji oceny). Identycznie dokonuje się wyboru drugiego osobnika do krzyżowania.

Często turniej przeprowadza się dla dwóch losowych osobników – wtedy nazywa się turniejem binarnym. Można również użyć tej metody dla większej liczby osobników. Rozmiar turnieju jest konfigurowalny w aplikacji, a domyślnie jest równy 5.

Liczba operacji w selekcji turniejowej jest zależna od rozmiaru turnieju – złożoność tego algorytmu to $O(n)$.

1.2.4 Krzyżowanie

Krzyżowanie to proces łączenia cech dwóch osobników, prowadzący do stworzenia potomka przez wymianę odcinków chromosomów rodziców[6].

Klasyczne metody krzyżowania używane dla innych problemów, w przypadku problemu komiwojażera okazują się niewydajne. Przy prostym krzyżowaniu (np. dwupunktowym) uzyskany osobnik mógłby zawierać duplikaty, które sprawiają że rozwiązanie jest błędne. Można zlikwidować nieprawidłowe elementy przez dodatkowe „naprawianie", jednak lepszą metodą jest użycie algorytmu wyspecjalizowanego do danego problemu.

Jako metodę krzyżowania w aplikacji został wybrany algorytm OX[1] autorstwa L. Davisa[1]. Wynikiem krzyżowania tym algorytmem jest poprawna trasa (metoda gwarantuje brak duplikatów), dlatego jest dobrym wyborem do rozwiązywania problemu komiwojażera.

Przykład: Dla rodziców P_1 i P_2 wybrano dwa losowe punkty: o indeksach 2 i 4 (licząc od zera). Należy przepisać elementy między wybranymi punktami krzyżowania z pierwszego rodzica do potomka, równocześnie usuwając je z drugiego chromosomu.

[1] Ordered Crossover

P_1:	1	2	3	4	5	6	7	8
P_2:	3	7	2	1	8	6	4	5

O:			3	4	5			

Następnie trzeba wypełnić brakujące indeksy potomka pozostałymi elementami drugiego rodzica zachowując kolejność.

P_1:	1	2	3	4	5	6	7	8
P_2:	~~3~~	7	2	1	8	6	~~4~~	~~5~~

O:	7	2	3	4	5	1	8	6

Źródło: Opracowanie własne na podstawie [1]

Krzyżowanie zostało zakończone – potomek O zawiera cechy chromosomów obu rodziców.

1.2.5 Mutacja

Mutacja to etap wprowadzania losowych zmian do chromosomu w celu zachowania różnorodności populacji. Pozwala zmniejszyć szanse wystąpienia sytuacji, w której krzyżowanie nadmiernie przystosowanych osobników nie poprawi wyniku w kolejnych pokoleniach.

Podczas mutacji istnieje identyczny problem jak przy etapie krzyżowania – jej wynikiem musi być poprawna trasa. Prostym sposobem jej przeprowadzenia jest losowanie dwóch punktów, a następnie zamiana ich kolejności.

Przykład: Dla osobnika O wybrano dwa losowe punkty: o indeksach 1 i 5 (licząc od zera). Elementy znajdujące się pod tymi indeksami zostały zamienione w osobniku M.

O:	1	2	3	4	5	6	7	8

M:	1	6	3	4	5	2	7	8

Źródło: Opracowanie własne

O ilości powyższych zamian decyduje współczynnik mutacji - konfigurowalny parametr, określający jak liczna część wszystkich punktów powinna zostać zamieniona. Współczynnik ten nie powinien być zbyt wysoki, gdyż za duża liczba losowych modyfikacji może uszkodzić najlepsze osobniki. Domyślna wartość tego parametru w aplikacji to 0,1.

Rozdział 2

Implementacja

2.1 Struktura projektu

Aplikacja została zrealizowana w języku C#. Jest to zorientowany obiektowo język korzystający z .NET Framework [7]. Do tworzenia aplikacji w tej technologii można skorzystać z darmowego środowiska Visual Studio. Język ten został wybrany głównie ze względu na możliwość stworzenia w nim aplikacji internetowej w technologii ASP.NET oraz dobrą wydajność pozwalającą na szybkie dokonanie obliczeń.

C# jest kompilowany do kodu pośredniego (CIL), przez co jego szybkość jest nieco niższa od języków kompilowanych do języka maszynowego, jak np. C, C++. Jednak skompilowany kod pośredni jest mocno zoptymalizowany, co pozwala na użycie go także w czasochłonnych obliczeniach, a niższa wydajność względem kodu natywnego jest praktycznie niezauważalna.

Praca w Visual Studio polega na stworzeniu *solucji*, czyli zbioru powiązanych projektów tworzących po skompilowaniu gotową aplikację. Podział na projekty umożliwia skorzystanie z różnych języków i kompilatorów w jednej solucji, a także logiczny podział komponentów. Powiązania między projektami są określane przez *referencje*. Po określeniu które projekty są zależne od innych, kompilator potrafi automatycznie ustalić kolejność ich budowania.

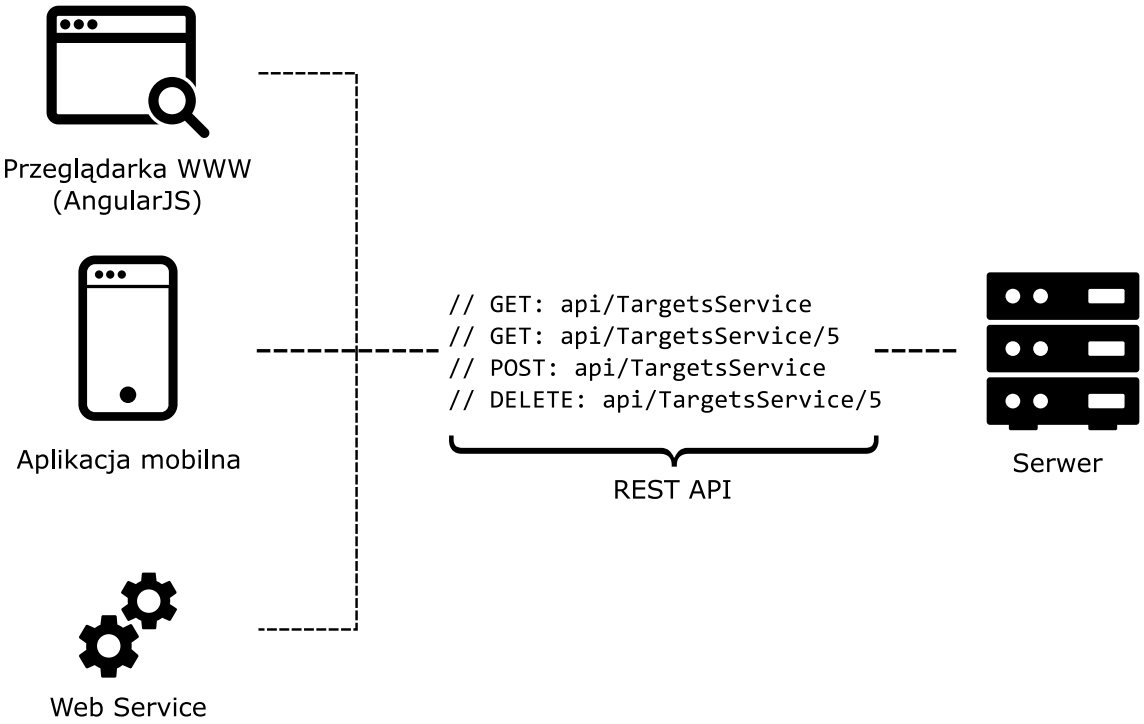
Solucja opisywanej aplikacji składa się z trzech projektów.

TSP

Przeglądarkowy interfejs użytkownika, napisany w AngularJS, który korzysta z API[1] stworzonego w ASP.NET. API zostało zrealizowane jako *webservice* w konwencji REST[2]: udostępnia wszystkie funkcjonalności przy użyciu standardowych metod

[1] Application Programming Interface

[2] Representional State Transfer



Rysunek 2.1: Schemat komunikacji między serwerem aplikacji oraz potencjalnymi klientami usługi sieciowej

Źródło: Opracowanie własne

HTTP, co pozwala łatwo napisać alternatywny interfejs, na przykład w formie aplikacji mobilnej. Schemat komunikacji z API został przedstawiony na rysunku 2.1. Klient może pobrać listę punktów pośrednich na mapie, wysyłając żądanie typu `GET`, lub podając identyfikator punktu wybrać pojedynczy obiekt. Zarówno odpowiedzi serwisu jak i żądania klienta składają się z obiektów zserializowanych do tekstowego formatu JSON[1], który stanowi alternatywę dla XML o czytelniejszej składni. Pobierając wybrany punkt z serwisu, odpowiedź zostanie zwrócona w formacie:

```
{
    "Id":"fff9a6bc-d36e-4c30-a49b-aa7022bfa352",
    "Name":"Basztowa 1, 33-332 Kraków, Polska",
    "Location":
    {
        "Latitude":50.066285384750863,
        "Longitude":19.935379028320312
    }
}
```

[1]JavaScript Object Notation

Dodawanie punktów jest realizowane przez żądanie `POST`, a usuwanie przez operację typu `DELETE`. Próba pobrania lub usunięcia nieistniejącego punktu spowoduje zwrócenie standardowego błędu HTTP 404.

Aby umożliwić jednoznaczną identyfikację punktów, każdy punkt przy dodawaniu otrzymuje indywidualny, losowy identyfikator GUID[1], który wygląda następująco: `5B7665CB-0B67-4D14-85F6-CDE6C5ACA7C8`. Składa się on z 32 znaków heksadecymalnych, przez co prawdopodobieństwo wylosowania identycznego identyfikatora jest znikome.

W zrealizowanej aplikacji z API serwisu korzysta strona napisana w HTML oraz JavaScript z biblioteką AngularJS. Dzięki skorzystaniu z możliwości JavaScript, strona nie musi być przeładowywana przy wykonywaniu żądania, przez co przypomina natywną aplikację okienkową pod względem wygody i szybkości obsługi.

Solver

Silnik odnajdujący optymalne trasy na podstawie zbioru punktów. Utworzenie silnika w osobnym projekcie sprawia że jest kompilowany do osobnego pliku DLL. Jest on zupełnie niezależny od interfejsu użytkownika (nie posiada referencji do projektu TSP). Umożliwia to skorzystanie z jego możliwości w innych aplikacjach, oraz implementację alternatywnych interfejsów, np. w formie zwykłej aplikacji okienkowej na system Windows.

Opis wykorzystanych algorytmów znajduje się w rozdziale 1.

Solver.Tests

Testy jednostkowe, sprawdzające poprawność działania silnika. Zostały napisane z pomocą otwartoźródłowej biblioteki NUnit, umożliwiającej tworzenie testów jednostkowych w .NET Framework. Najważniejsze testy zostaną opisane w rozdziale 3.2.

2.2 Wzorce projektowe

Wzorce projektowe to proste rozwiązania problemów spotykanych w programowaniu zorientowanym obiektowo[4]. Programowanie z użyciem wzorców wymaga większych nakładów pracy przy tworzeniu programu, jednak stworzony kod jest bardziej elastyczny na potencjalne modyfikacje.

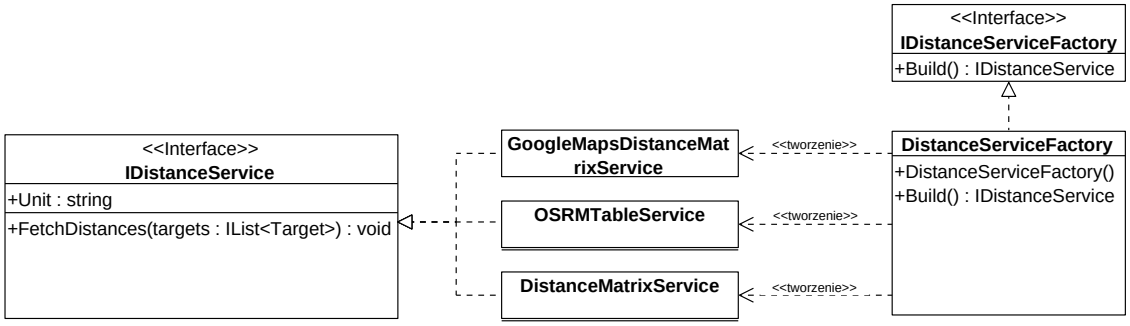
[1] Globally Unique Identifier

Fabryka

Ten wzorzec konstrukcyjny został użyty do budowania klas pobierających dane o odległościach z serwisów zewnętrznych. Poszczególne ich rodzaje powstały jako osobne klasy implementujące wspólny interfejs `IDistanceService`.

Wydzielenie tworzenia klas do klasy fabrycznej umożliwia łatwe dodanie kolejnych źródeł danych. Ponieważ obiekty są tworzone w jednej metodzie o nazwie `DistanceServiceFactory.Build()`, dodanie obsługi kolejnej klasy nie będzie wymagało modyfikacji każdego miejsca w kodzie korzystającego z pobierania danych.

Do klasy `DistanceServiceFactory` jest wstrzykiwany obiekt zawierający konfigurację aplikacji, na podstawie której klasa konstruuje odpowiedni typ.



Rysunek 2.2: Diagram klas fabryki konstruującej klasy pobierające dane zewnętrzne

Źródło: Opracowanie własne na podstawie [4]

MVC

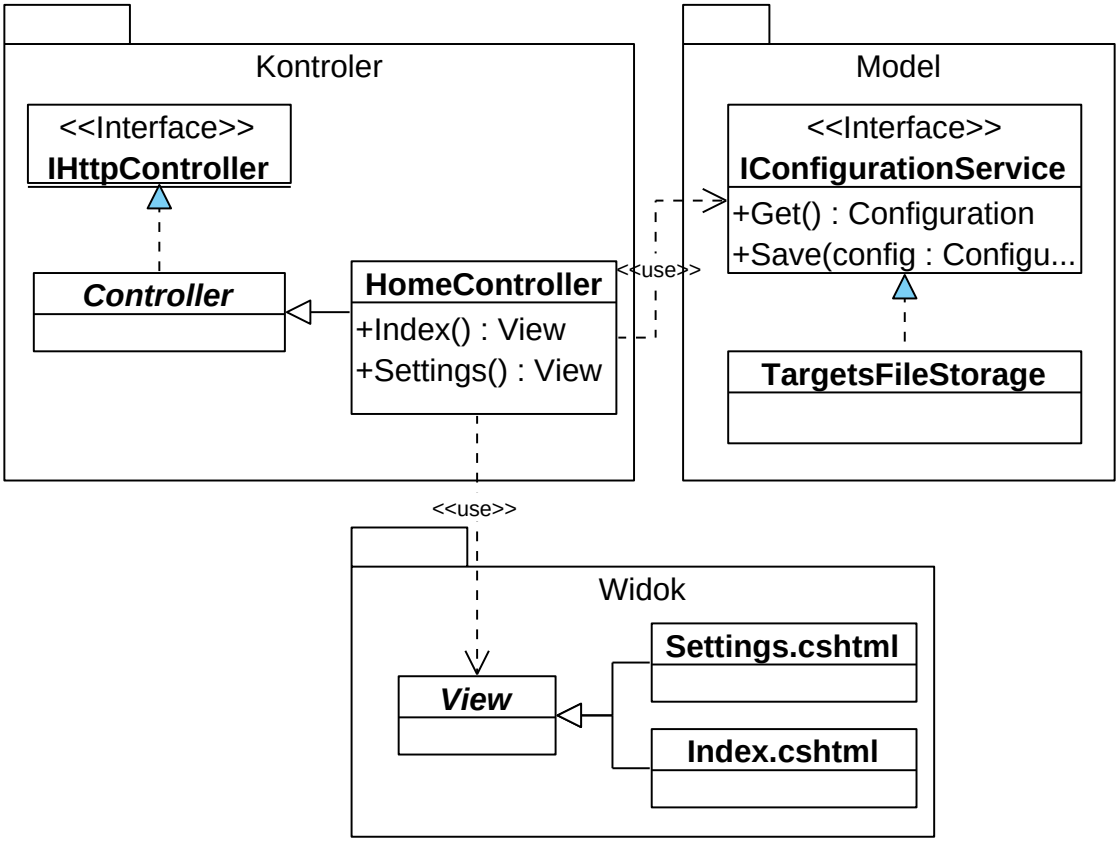
Wzorzec MVC składa się z trzech typów obiektów: Modelu (*Model*), Widoku (*View*) i Kontrolera (*Controller*). MVC pozwala na oddzielenie widoku użytkownika od warstwy przetwarzania.

W aplikacji ten wzorzec został użyty do wyświetlenia strony głównej i ustawień (schemat na rysunku 2.3). Całym mechanizmem zarządza kontroler `HomeController`. Dziedziczy po standardowej w ASP.NET klasie abstrakcyjnej `Controller`.

`HomeController` ma dwie publiczne metody: `Index()` i `Settings()`, które służą odpowiednio do wyświetlenia strony głównej oraz podstrony zarządzania konfiguracją. Metody te wywołują konstruktory odpowiednich widoków, które zostały stworzone w HTML z pomocą składni Razor - języka służącego do tworzenia szablonów w ASP.NET.

Kontroler korzysta z modelu opisanego przez interfejs `IConfigurationService`, który zawiera dwie metody: `Get()` i `Save(Configuration config)`. Służy on do

pobierania i zapisywania danych konfiguracyjnych. Interfejs ten został zaimplementowany przez klasę `TargetsFileStorage`. Klasa ta przechowuje zserializowane dane konfiguracyjne na dysku w postaci tekstowego pliku JSON. Kontroler nie tworzy obiektu tej klasy, lecz jest on wstrzykiwany jako zależność. Umożliwia to zamianę implementacji interfejsu na inny, przykładowo korzystający z bazy danych.



Rysunek 2.3: Diagram wzorca MVC w części wyświetlającej strony: główną i ustawień

Źródło: Opracowanie własne

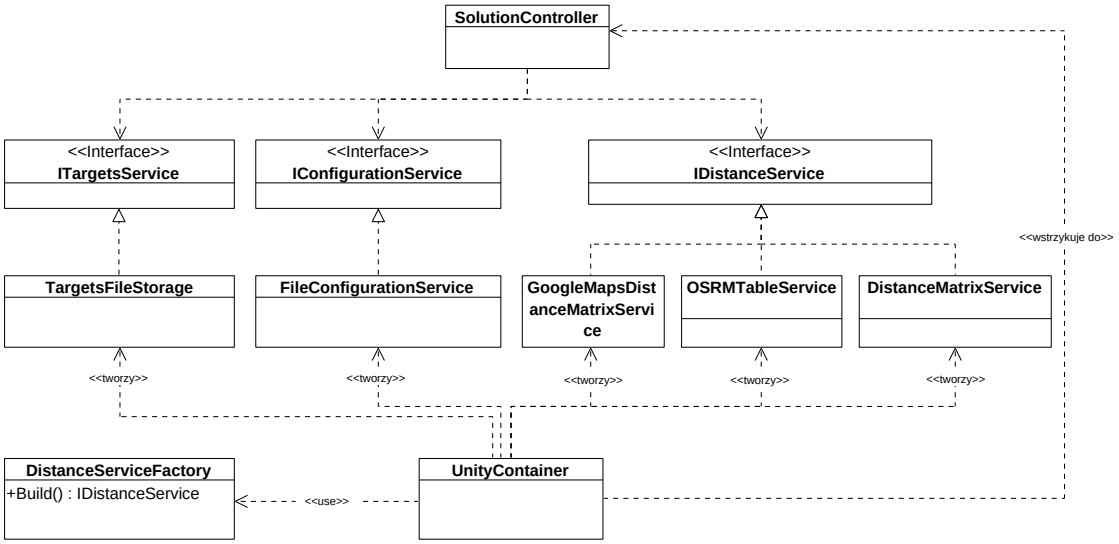
Wstrzykiwanie zależności

Wstrzykiwanie zależności (*Dependency Injection*) jest wzorcem, w którym obiekty wymagane przez klasę do działania są dostarczane przez inny obiekt – *injector*. Pozwala na usunięcie bezpośredniego powiązania między klasami[3]. Dzięki temu klasa korzystająca z zależności nie musi znać implementacji konkretnego obiektu, a operować na interfejsach. Pozwala to na bezproblemową wymianę implementacji na inną.

Klasę przechowującą obiekty i przeprowadzającą wstrzykiwanie nazywa się kontenerem wstrzykiwania zależności (*Dependency Injection Container*). Ponieważ kontener najczęściej zajmuje się stworzeniem obiektów, to w nim konfiguruje się czas ich życia, np. tworzenie instancji na jedno żądanie HTTP lub jedna instancja na całą aplikację (podobnie jak we wzorcu Singleton).

Mechanizm wstrzykiwania jest wspierany w ASP.NET po odpowiedniej konfiguracji. Aplikacja korzysta z gotowego kontenera o nazwie Unity. Pozwala on na wstrzykiwanie zależności do parametrów konstruktora.

Wszystkie kontrolery w opisywanej aplikacji mają zależności wstrzykiwane przez kontener.



Rysunek 2.4: Diagram wstrzykiwania zależności kontrolera

Źródło: Opracowanie własne na podstawie [3]

Na rysunku 2.4 został przedstawiony schemat wstrzykiwania zależności klasy `SolutionController`. Konstruktor tego kontrolera przyjmuje trzy parametry jako interfejsy. Kontener `UnityContainer` rozwiązuje zależności wyszukując obiekt implementujący dany interfejs. Ponieważ wszystkim klasom wystarczy jedna instancja na czas działania programu, obiekty zostały uprzednio utworzone przez kontener podczas startu aplikacji.

Klasy implementujące interfejs `IDistanceService` są tworzone przez wcześniej opisaną fabrykę, natomiast pozostałe zależności są konstruowane bezpośrednio przez kontener.

2.3 Integracja systemów zewnętrznych

Ponieważ aplikacja operuje na rzeczywistych mapach, konieczna była integracja z odpowiednimi źródłami danych geograficznych. Program pobiera z systemów zewnętrznych:

- odległości między punktami (czas przejazdu lub dystans),
- przybliżony adres na podstawie współrzędnych,
- graficzne mapy, przeznaczone do wyświetlania.

2.3.1 Google Maps

Usługa Google Maps udostępnia poprzez API dane geograficzne i algorytmy, np. znajdowania trasy, lokalizacji adresu na podstawie współrzędnych. Konieczna jest rejestracja na stronie firmy Google, gdzie możliwe jest utworzenie klucza identyfikującego aplikację na serwerze. Wygenerowany klucz musi być przesyłany z każdym żądaniem do serwera, w przeciwnym wypadku zostanie odesłana odpowiedź o braku dostępu.

Interfejs Google jest dostępny dla wielu języków i technologii. Program korzysta z dwóch: **Maps JavaScript API**, działającego po stronie przeglądarki, oraz **Web Services API**, używanego przez napisany w C# serwer.

Widoczna na interfejsie użytkownika interaktywna mapa jest utworzona przy pomocy API JavaScript. Umożliwia ono dodanie do strony HTML odpowiedniego widoku graficznego, oraz wykonywanie na nim różnych akcji z poziomu kodu JavaScript, np. zaznaczenia punktów i narysowania odcinków pomiędzy nimi. Możliwa jest także obsługa zdarzeń pochodzących z mapy, np. kliknięcia lub zmiany powiększenia. Tę obsługę realizuje się poprzez tzw. *callbacki*, czyli własne funkcje obsługujące zdarzenie, napisane w JavaScript i przekazane do obiektów API.

Pozostałe dane są pobierane przez serwer z Web Services API. Jest to przede wszystkim macierz odległości między punktami. **Google Maps Distance Matrix API** umożliwia wysłanie listy współrzędnych w pojedynczym zapytaniu i otrzymanie odpowiedzi w formacie JSON, zawierającej wszystkie odległości i czasy.

Poważnym ograniczeniem tego API jest wymiar macierzy odległości. Jest to tylko 25 punktów na zapytanie. Ponadto istnieją ograniczenia na dobową i sekundową liczbę zapytań.

2.3.2 OSRM: Open Source Routing Machine

Aby móc rozwiązać problem dla większej niż 25 liczby punktów, należy w ustawieniach aplikacji wybrać silnik **OpenStreetMap**. OpenStreetMap to otwartoźródłowy projekt, mający na celu dostarczyć darmową alternatywę dla map komercyjnych.

Na bazie OpenStreetMap powstał projekt **OSRM**[1]. Jest to konsolowy program, możliwy do uruchomienia na własnym komputerze bez dostępu do internetu. Przez udostępnione API HTTP umożliwia m.in. odnajdowanie tras między podanymi współrzędnymi.

Podobnie jak z Google Maps możliwe jest pobranie macierzy w jednym zapytaniu. Zaletą OSRM jest brak ograniczenia liczby punktów (w praktyce ograniczeniem są możliwości komputera na którym uruchomiono program). Niestety zwracana z API macierz zawiera wyłącznie informacje o szacowanych czasach, a nie dystansach. W czasie pisania poniższej pracy nie została ukończona wersja OSRM zwracająca dystanse w macierzy.

W celu skorzystania z OSRM do optymalizacji odległości, aplikacja wysyła wielokrotne zapytania do API - wymaga to wysłania jednego żądania HTTP na jedną odległość. Ponieważ złożoność tego rodzaju pobierania danych to $O(n^2)$, a nawiązanie połączenia HTTP jest czasochłonne, to jego prędkość jest bardzo wolna. Aby nieco przyspieszyć działanie, aplikacja wysyła zapytania równolegle, z wielu wątków. Jednak ta metoda wciąż pozostaje najwolniejsza - porównanie jej szybkości z innymi znajduje się w rozdziale 3.3.

Aby uruchomić OSRM lokalnie, można skorzystać z Dockera. Skompilowany projekt jest dostarczony w postaci kontenera, który należy wcześniej skonfigurować. W tym celu trzeba pobrać plik map ze strony OpenStreetMap, następnie udostępnić go Dockerowi i wewnątrz kontenera skompilować go do użycia przez OSRM.

Ponieważ OSRM nie ma potrzeby komunikacji z żadnym zewnętrznym serwisem przez internet, nie ma ograniczeń na liczbę zapytań.

[1] Open Source Routing Machine

Rozdział 3

Testowanie aplikacji

3.1 Testy manualne

Aplikacja może być uruchomiona w dowolnej współczesnej przeglądarce. Najważniejszym elementem interfejsu, widocznego na rysunku 3.1 jest mapa. Przesyłki oznaczone są czerwonymi znacznikami, natomiast punkt początkowy zielonym znacznikiem z ikoną ciężarówki.

Dodanie przesyłek jest możliwe na dwa sposoby:

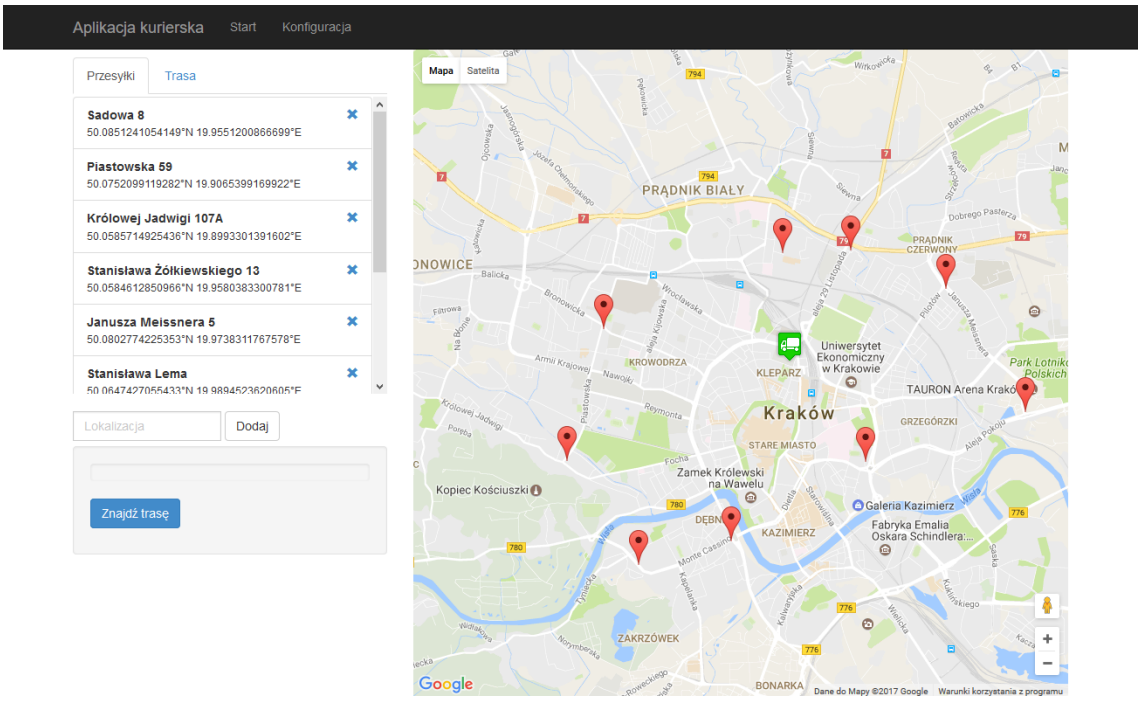
- kliknięcie na mapie – w tym wypadku przybliżony adres zostanie automatycznie znaleziony na podstawie współrzędnych,
- wpisanie adresu i naciśnięcie przycisku „Dodaj" – współrzędne zostaną odnalezione przez wyszukanie adresu.

Rezultatem w obu przypadkach jest dodanie przesyłki do listy oraz odpowiedniego punktu na mapie.

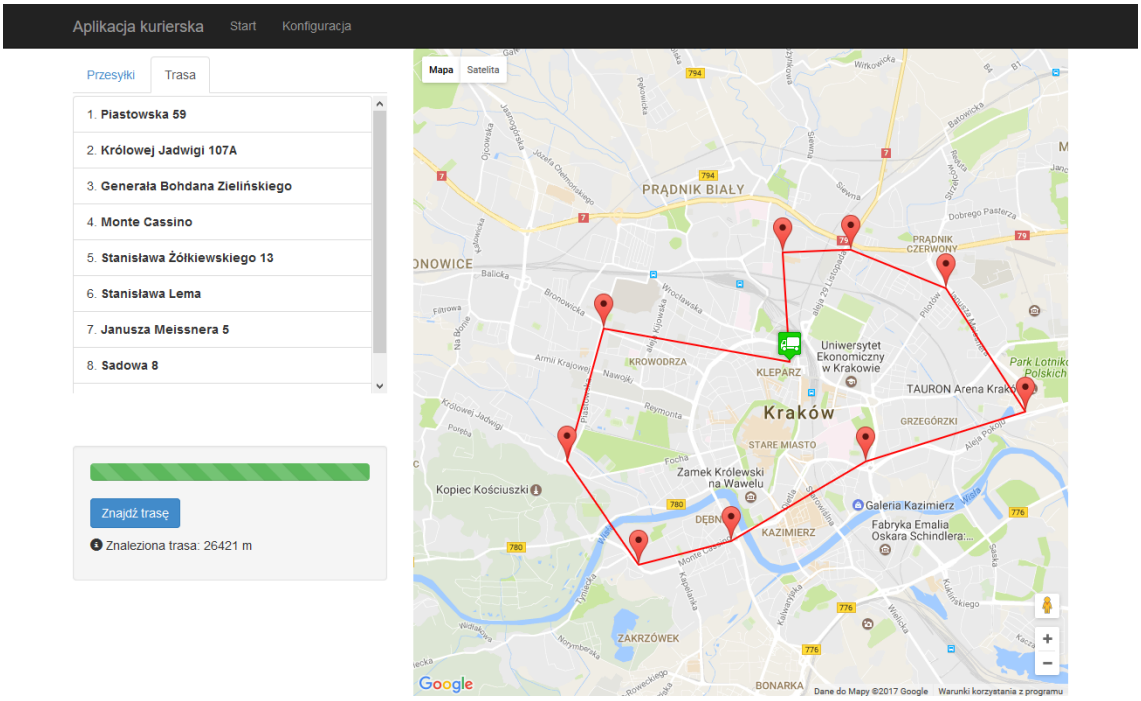
Po wprowadzeniu wszystkich punktów, należy nacisnąć przycisk „Znajdź trasę". Uruchamia to proces wyszukiwania trasy na serwerze, a użytkownik widzi w tym czasie animację paska postępu. W przypadku wystąpienia błędu, na przykład z połączeniem, zostanie wyświetlony komunikat z opisem błędu.

Gdy serwer zwróci poprawną odpowiedź (rysunek 3.2), na mapie pokazują się połączenia między punktami trasy. Kolor paska postępu zmienia się na zielony, a poniżej można odczytać łączną długość lub czas potrzebny do przebycia trasy (w zależności od wybranego przez użytkownika sposobu optymalizacji).

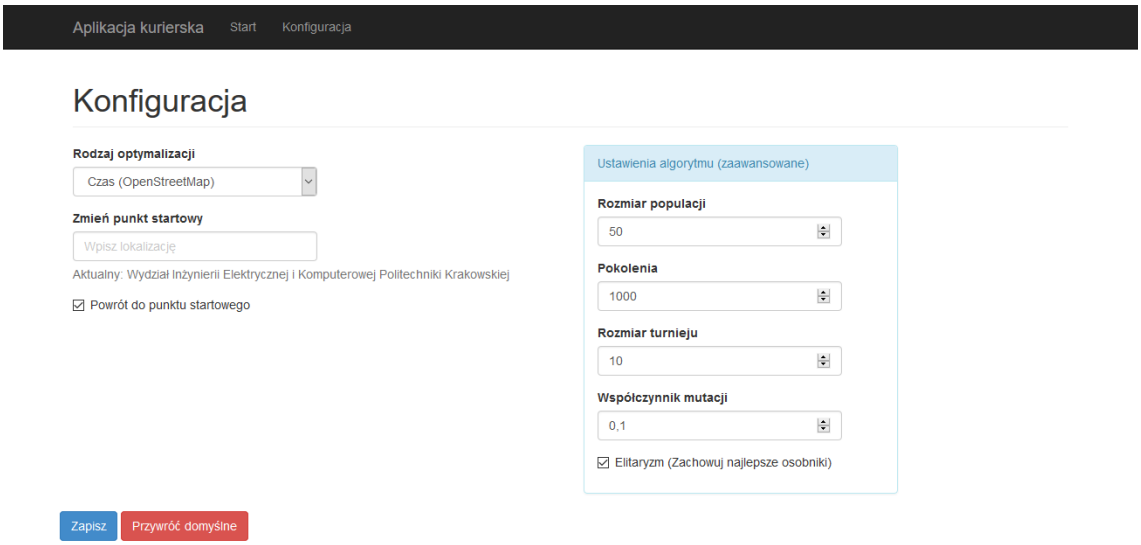
Po wybraniu z menu odnośnika „Konfiguracja" użytkownik ma możliwość zmiany ustawień aplikacji (rysunek 3.3a). Okno zostało podzielone na opcje podstawowe, oraz ustawienia silnika przeznaczone dla zaawansowanych użytkowników.



Rysunek 3.1: Widok aplikacji po dodaniu kilku przesyłek

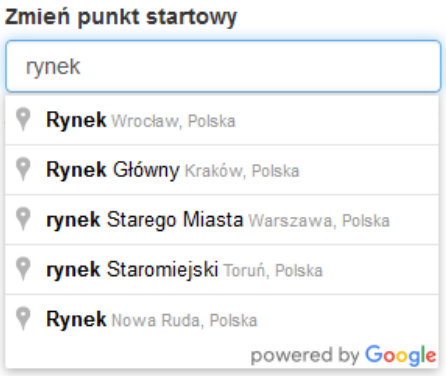


Rysunek 3.2: Mapa z widoczną trasą oraz lista kolejnych celów



(a) wszystkie dostępne ustawienia

(b) zmiana rodzaju optymalizacji



(c) wybór punktu początkowego

Rysunek 3.3: Ekran konfiguracji

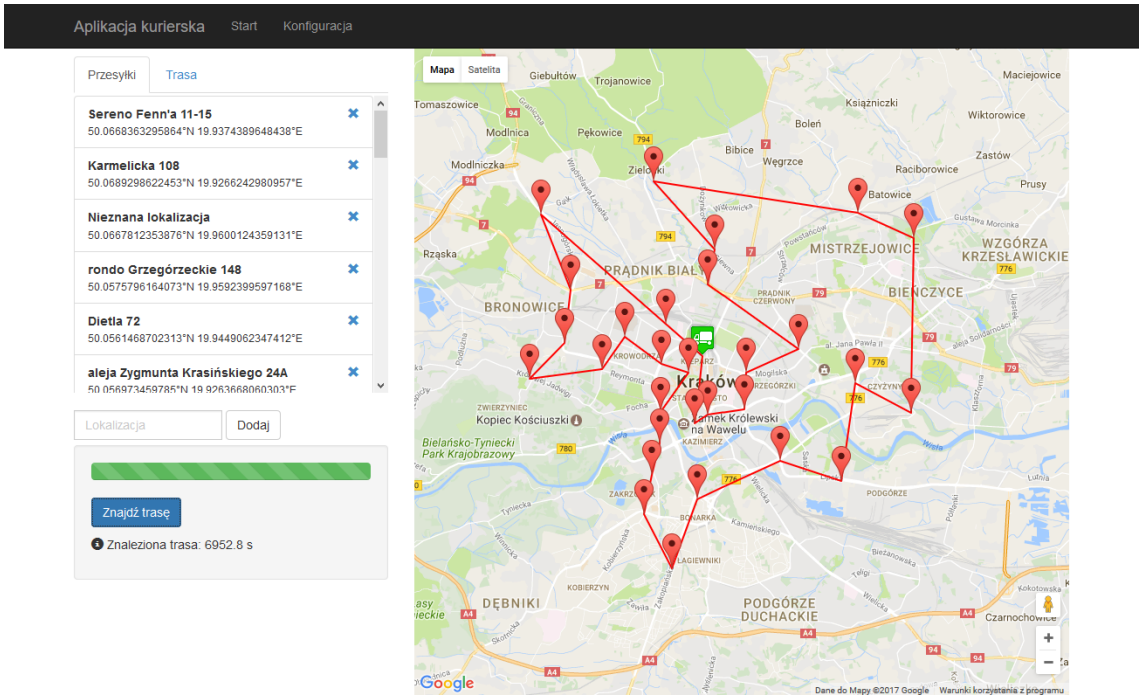
Naciśnięcie przycisku „Przywróć domyślne” resetuje wszystkie ustawienia do stanu początkowego.

Konfiguracja jest zapisywana na dysku serwera w pliku JSON, przez co wartości mogą być odczytane po restarcie serwera.

Do podstawowych opcji należy wybór jednego z trzech rodzajów optymalizacji. Użytkownik ma możliwość wybrania punktu początkowego, czyli siedziby firmy kurierskiej. Wyboru tej lokalizacji dokonuje się poprzez wpisanie adresu lub nazwy miejsca. Aplikacja na bieżąco podpowiada wpisywane adresy.

W zaawansowanych ustawieniach można dokonać konfiguracji parametrów algorytmu genetycznego. Pozwala to dostosować aplikację do wymagań konkretnego użytkownika, który może preferować większą dokładność optymalizacji lub jej czas.

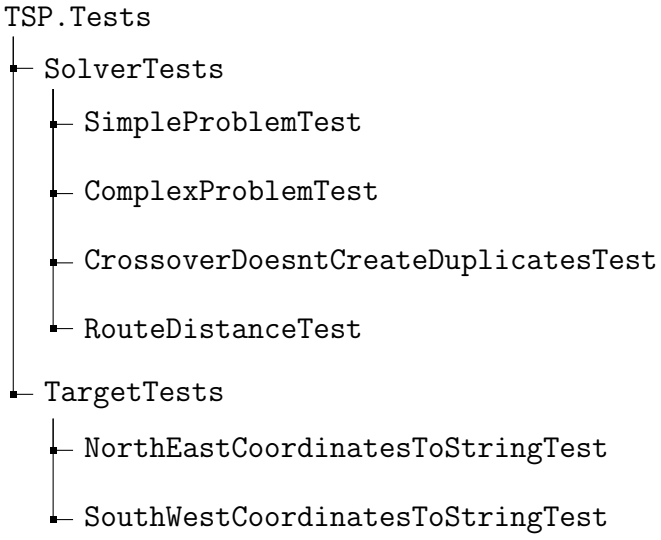
Odpowiednie dostrojenie parametrów algorytmu umożliwia znalezienie trasy dla większej liczby przesyłek. Na rysunku 3.4 aplikacja znalazła optymalną trasę dla 30 punktów.



Rysunek 3.4: Znaleziona trasa dla 30 przesyłek

3.2 Testy jednostkowe

Projekt zawiera kilka testów jednostkowych, które sprawdzają poprawność poszczególnych komponentów programu. Testy jednostkowe są wykonywane automatycznie, przez co po modyfikacji kodu programu można łatwo upewnić się, czy wprowadzone zmiany są zgodne z testami. Struktura projektu testowego wygląda następująco:



Dwa testy (`SimpleProblemTest` i `ComplexProblemTest`) sprawdzają działanie całego silnika, przy domyślnych ustawieniach. Drugi z testów dostarcza na wejście przykładowy problem przedstawiony we wprowadzeniu.

Test `CrossoverDoesntCreateDuplicatesTest` gwarantuje że wynik krzyżowania jest poprawny: nie zawiera duplikatów i składa się z tej samej liczby punktów co źródłowe trasy.

`RouteDistanceTest` zapewnia poprawność obliczania całkowitej długości trasy.

Metody klasy `TargetTests` testują zamianę przykładowych współrzędnych geograficznych na tekst. Jest to istotne, ponieważ w aplikacji występują trzy sposoby zapisu współrzędnych. Ze względu na wykorzystanie dwóch systemów zewnętrznych wymagających różnych formatów, oraz wyświetlanie współrzędnych użytkownikowi, potrzebne są wszystkie trzy:

- ”14.123°N 13.345°E” - format przeznaczony do wyświetlania,
- ”14.123,13.345” - format Google Maps,
- ”13.345,14.123” - format OSRM.

3.3 Badanie wydajności

Wydajność pobierania danych geograficznych

Pierwszym etapem poszukiwania rozwiązania jest pobranie danych z serwisu. Aplikacja obsługuje trzy źródła:

- odległości z Google Maps (serwis internetowy, pojedyncze zapytanie),
- odległości z OpenStreetMap (serwis lokalny, wiele zapytań równoległych),
- czas przejazdu z OpenStreetMap (serwis lokalny, pojedyncze zapytanie).

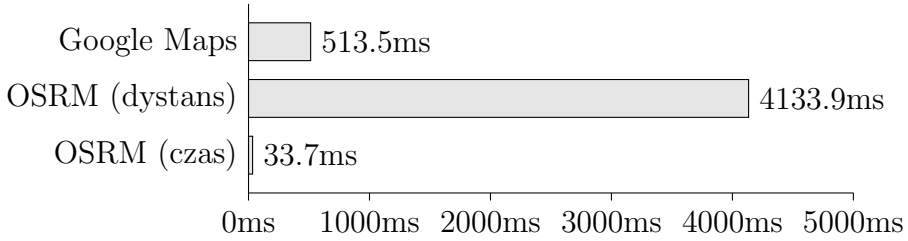
Wyniki pomiarów wydajności dla 10 punktów zostały przedstawione w tabeli 3.1 oraz na wykresie 3.5.

Tabela 3.1: Czas pobierania danych dla różnych metod [ms]

	Google Maps	**OSRM (dystans)**	**OSRM (czas)**
	601	7797	62
	447	1074	31
	446	6747	26
	454	1046	45
	423	1228	34
	437	7176	29
	492	960	21
	948	1073	40
	472	6859	19
	415	7379	30
Średnia	**513.5**	**4133.9**	**33.7**
Maksimum	948	7797	62
Minimum	415	960	19

Źródło: Opracowanie własne

Rysunek 3.5: Czas pobierania danych dla różnych metod: wykres



Źródło: Opracowanie własne

Po wykonaniu powyższych pomiarów okazało się że najszybszą metodą jest pobieranie macierzy czasów przejazdów z lokalnego serwisu OSRM. Jest to zgodne z przewidywaniami, gdyż cała macierz jest pobierana w jednym zapytaniu HTTP oraz nie występują opóźnienia spowodowane transmisją sieciową.

Wydajność algorytmu genetycznego

Ze względu na dużą ilość danych do zebrania, pomiary wykonano przy pomocy testów automatycznych. Każdy test polega na ustawieniu odpowiednich parametrów, stworzenia obiektu `System.Diagnostics.Stopwatch` (klasa w .NET Framework pozwalająca dokładnie mierzyć czas) i uruchomieniu algorytmu genetycznego. Zwrócone wyniki (najkrótsze trasy) oraz czas jego pracy są wypisywane w konsoli.

Testy wydajnościowe znajdują się w klasie `PerformanceTests` należącej do projektu `Solver.Tests`.

Ponieważ algorytm korzysta z generatora liczb pseudolosowych, wyniki są różne przy każdym uruchomieniu. Ponadto czas działania jest dodatkowo zależny od szybkości komputera na którym działa program. Aby uzyskać dokładniejsze wyniki, dla każdej konfiguracji powtórzono test 10 razy.

Sprawdzono trzy parametry, mające największy wpływ na optymalizację i czas działania, tj. liczbę pokoleń, elitaryzm oraz rozmiar populacji.

Dane o czasach przejazdów między punktami zostały uprzednio pobrane z OSRM i zapisane do pliku.

Tabela 3.2: Czas działania algorytmu genetycznego [ms]

Pokolenia	**100**						**1000**					
Elitaryzm	**NIE**			**TAK**			**NIE**			**TAK**		
Rozmiar populacji	**50**	**100**	**1000**	**50**	**100**	**1000**	**50**	**100**	**1000**	**50**	**100**	**1000**
1	156	325	3436	176	328	3478	1681	3157	34421	1672	3330	34770
2	155	319	3483	238	331	3429	1544	3109	33009	1616	3283	34325
3	152	359	3415	208	337	3478	1542	3135	32183	1629	3344	34546
4	152	320	3445	165	325	3482	1569	3090	32067	1637	3426	34595
5	155	318	3524	161	332	3462	1571	3155	32037	1607	3280	34564
6	152	317	3340	166	332	3429	1528	3163	32028	1619	3322	34643
7	153	313	3299	163	328	3459	1549	3106	32397	1665	3326	34462
8	151	309	3338	164	335	3602	1571	3141	32401	1619	3266	34469
9	151	316	3344	180	331	3445	1541	3177	33075	1617	3308	34532
10	151	309	3288	173	326	3470	1533	3341	32679	1672	3318	34433
Suma	1528	3205	33912	1794	3305	34734	15629	31574	326297	16353	33203	345339
Średnia	**152.80**	**320.50**	**3391.20**	**179.40**	**330.50**	**3473.40**	**1562.90**	**3157.40**	**32629.70**	**1635.30**	**3320.30**	**34533.90**
Max	156	359	3524	238	337	3602	1681	3341	34421	1672	3426	34770
Min	151	309	3288	161	325	3429	1528	3090	32028	1607	3266	34325

Źródło: Opracowanie własne

Najlepszy znaleziony czas przejazdu to 3,96 h. Został on wyszukany przy następujących opcjach: rozmiar populacji 50, 1000 pokoleń, elitaryzm włączony. Także dla tej konfiguracji uzyskano najlepsze średnie wyniki. Można więc zauważyć że zwiększanie rozmiaru populacji nie zawsze poprawia optymalizację, za to znacząco zwiększa czas działania.

Tabela 3.3: Wyniki optymalizacji czasu przejazdu trasy [h]

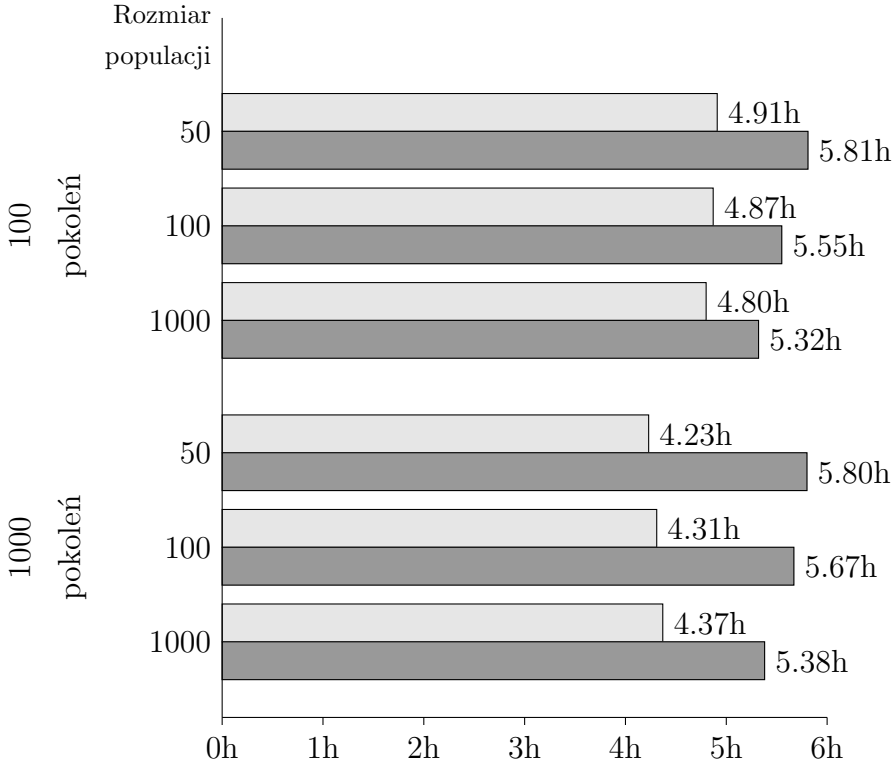
Pokolenia	**100**						**1000**					
Elitaryzm	**NIE**			**TAK**			**NIE**			**TAK**		
Rozmiar populacji	**50**	**100**	**1000**	**50**	**100**	**1000**	**50**	**100**	**1000**	**50**	**100**	**1000**
1	5.62	5.34	5.28	4.89	4.98	4.62	5.85	5.50	5.51	4.25	4.47	4.14
2	5.98	5.71	5.44	4.91	5.13	4.90	6.01	5.69	5.40	4.43	4.45	4.47
3	5.91	5.55	5.53	4.78	4.80	4.97	5.96	5.83	5.31	4.59	4.21	4.02
4	5.62	5.27	5.20	5.02	4.93	4.78	5.58	5.72	5.23	4.19	4.13	4.39
5	5.69	5.73	5.13	5.27	4.75	4.88	5.96	5.49	5.39	4.09	4.04	4.55
6	5.86	5.53	5.28	5.05	4.73	4.48	5.75	5.43	5.26	3.96	4.42	4.19
7	5.89	5.80	5.32	4.80	4.82	4.87	5.66	5.82	5.51	4.17	4.22	4.73
8	5.62	5.31	5.14	4.98	5.09	4.67	5.70	5.63	5.44	4.60	4.56	4.60
9	6.04	5.28	5.32	4.75	4.81	4.96	5.70	5.81	5.32	4.07	4.18	4.30
10	5.89	5.94	5.56	4.66	4.62	4.88	5.82	5.77	5.42	3.96	4.46	4.35
Suma	58.12	55.47	53.20	49.12	48.68	48.00	57.99	56.71	53.80	42.30	43.13	43.74
Średnia	**5.81**	**5.55**	**5.32**	**4.91**	**4.87**	**4.80**	**5.80**	**5.67**	**5.38**	**4.23**	**4.31**	**4.37**
Maksimimum	6.04	5.94	5.56	5.27	5.13	4.97	6.01	5.83	5.51	4.60	4.56	4.73
Minimum	5.62	5.27	5.13	4.66	4.62	4.48	5.58	5.43	5.23	3.96	4.04	4.02

Źródło: Opracowanie własne

Włączenie elitaryzmu znacznie poprawiło wynik dla każdego przypadku, a w dodatku ma minimalny wpływ na czas obliczeń.

Przez zasadę działania algorytmu genetycznego, przy zbyt niskim rozmiarze populacji lub liczby pokoleń wyniki są losowe. Nawet drobne zwiększenie tych parametrów poprawi optymalizację, a czas oczekiwania na wyniki wciąż będzie niezauważalny dla użytkownika (do paru sekund).

Rysunek 3.6: Średnie wyniki optymalizacji czasu przejazdu trasy: wykres



Źródło: Opracowanie własne

Podsumowanie

Efektem niniejszej pracy jest działająca aplikacja, realizująca zgodnie z założeniami postawione zadanie jakim jest odnajdowanie optymalnych tras dla problemu komiwojażera. Program może być wykorzystywany przez kurierów, listonoszy ale też inne osoby, których praca wymaga szybkiego poruszania się po obszarze miasta, na przykład przedstawicieli handlowych, pośredników nieruchomości.

Wybór algorytmu genetycznego pozwolił na osiągnięcie odpowiedniej optymalizacji odnajdowanych tras, mimo że należy do algorytmów przybliżonych.

Aplikacja może być łatwo rozwijana w przyszłości. Język C#, w jakim powstał kod umożliwia integrację z C++, co pozwala na uzyskanie jeszcze lepszej wydajności (przy zachowaniu pełnej funkcjonalności) po przepisaniu części obliczeniowej na język kompilowany do kodu natywnego. Dzięki oparciu API o standard HTTP możliwe jest łatwe stworzenie dodatkowych interfejsów użytkownika, w formie aplikacji mobilnej lub okienkowej. Elastyczna konstrukcja silnika, zbudowanego z użyciem wzorców projektowych pozwala na dodanie i rozbudowę jego poszczególnych części, takich jak algorytmy krzyżowania i selekcji lub całkowitą zamianę typu algorytmu na inny niż genetyczny.

Bibliografia

[1] Lawrence Davis. Applying adaptive algorithms to epistatic domains. *IJCAI*, 85:162–164, 1985.

[2] Richard Durstenfeld. Algorithm 235: Random permutation. *ACM*, 7(7):420–, 1964.

[3] Martin Fowler. Inversion of Control containers and the Dependency Injection pattern. `https://martinfowler.com/articles/injection.html`. [dostęp 26.02.2017].

[4] Erich Gamma [i in]. Design patterns: Elements of reusable object-oriented software. *Addison-Wesley*, 1995.

[5] Kalyanmoy Goldberg, David E. i Deb. A comparative analysis of selection schemes used in genetic algorithms. *Foundations of genetic algorithms*, 1:69–93, 1991.

[6] Zbigniew Michalewicz. *Algorytmy genetyczne + struktury danych = programy ewolucyjne*. Wydaw. Naukowo-Techniczne, Warszawa, 2003.

[7] Microsoft. Microsoft Developer Network: C#. `https://msdn.microsoft.com/pl-pl/library/kx37x362.aspx`. [dostęp 26.02.2017].

[8] Christos H. Papadimitriou. The Euclidean travelling salesman problem is NP-complete. *Theoretical Computer Science*, 4(3):237, 1977.