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, 19 lutego 2017

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 [5].

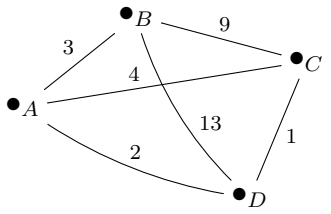
Poszukiwanie optymalnego rozwiązania już dla małej liczby miejsc docelowych wymagałoby znacznego czasu, nawet na wydajnym komputerze. Z tego powodu aplikacja powinna wykorzystywać algorytm sub-optymalny, który pozwala odnaleźć rozwiązanie w stosunkowo krótkim czasie, a odnalezione trasy są wystarczająco 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[3].



Rysunek 1.1: Przykład symetryczny: Reprezentacja grafowa

[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

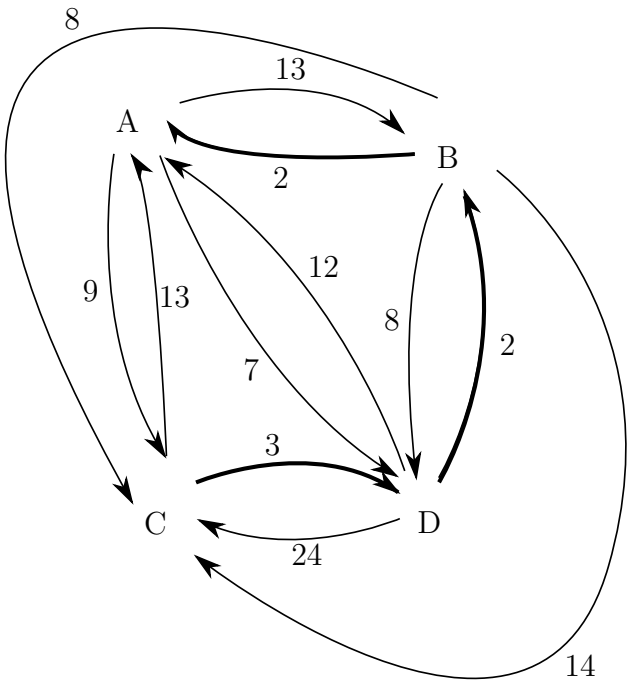
Tabela 1.1: Przykład symetryczny: Macierz sąsiedztwa

Źródło: Opracowanie własne

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.

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 symetryczny: Macierz sąsiedztwa

Źródło: Opracowanie własne

Przykład problemu asymetrycznego znajduje się na grafie 1.2. Najkrótszą trasą rozpoczynającą się w punkcie C jest $C \to D \to B \to A \to C$ 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[3]. 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: [3]

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 zawiera zbiór odległości do każdego innego punktu. Odległości są przechowywane we 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

1.2.4 Krzyżowanie

Krzyżowanie to proces łączenia cech dwóch chromosomów, prowadzący do stworzenia potomka przez wymianę odcinków chromosomów rodziców[3].

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].

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.

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

O:			3	4	5			

[1]Ordered Crossover

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

Rozdział 2

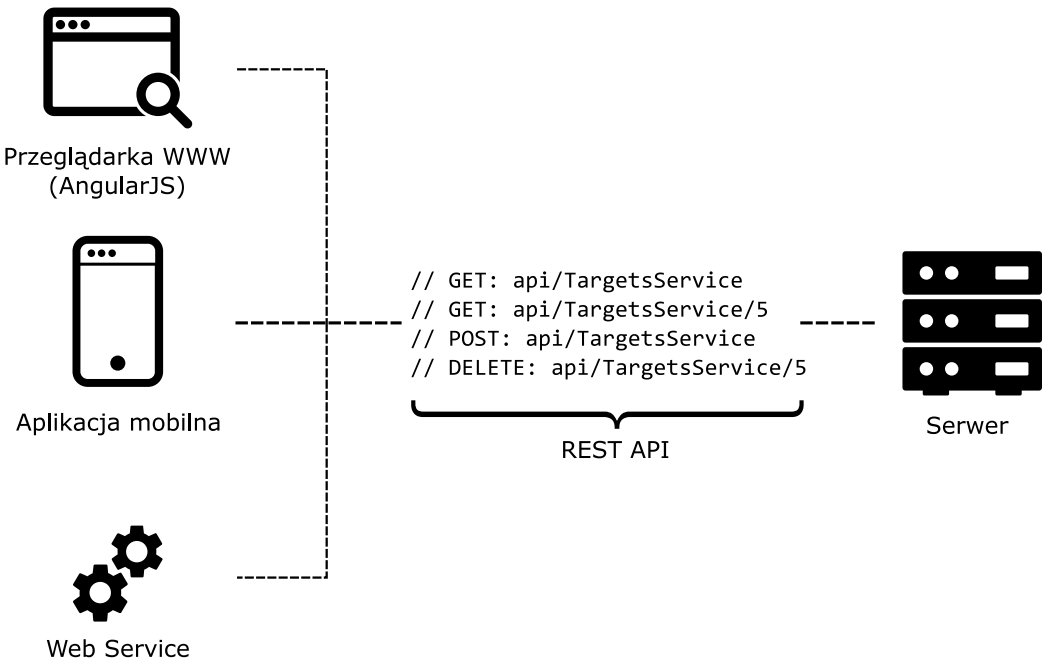
Implementacja

2.1 Struktura projektu

(Opis najważniejszych projektów i klas, architektura aplikacji, schematy. Wzorce projektowe w osobnym podrozdziale? MVVM, Factory, IoC,) Aplikacja została zrealizowana w języku C#. Jest to zorientowany obiektowo język korzystający z .NET Framework [4]. 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



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

Źródło: Opracowanie własne

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 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 punkt. Zarówno odpowiedzi serwisu jak i żąda-

[1] Application Programming Interface

[2] Representional State Transfer

nia klienta składają się z obiektów zserializowanych do tekstowego formatu JSON[1], który stanowi alternatywę dla XML. Pobierając wybrany punkt z serwisu, odpowiedź zostanie zwrócona w poniższym formacie:

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

Dodawanie punktów jest realizowane przez żądanie `POST`, a usuwanie przez `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[2], 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

[1] JavaScript Object Notation

[2] Globally Unique Identifier

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 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.

2.2.1 Google Maps

Usługa Google Maps udostępnia poprzez API różne usługi za darmo. 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.

2.2.2 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 szacowa-

[1] Open Source Routing Machine

nych 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 $O(n^2)$, 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.

Rozdział 3

Testowanie aplikacji

3.1 Testy manualne

(Prezentacja działania aplikacji (screenshoty), uruchomienie dla dużej liczby punktów, przy różnej konfiguracji)

3.2 Testy jednostkowe

(Opis testów jednostkowych sprawdzających poprawność komponentów)

3.3 Badanie wydajności

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)

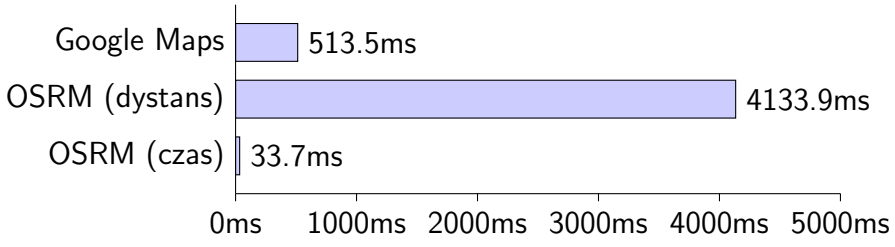
Wyniki pomiarów wydajności dla 10 punktów zostały przedstawione w tabeli 3.1 oraz na wykresie 3.1.

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 3.1: 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.

Podsumowanie

(Podsumowanie pracy, potencjalne możliwości rozwoju aplikacji.) Język C# umożliwia integrację z kodem 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] Zbigniew Michalewicz. *Algorytmy genetyczne + struktury danych = programy ewolucyjne*. Wydaw. Naukowo-Techniczne, Warszawa, 2003.

[4] Microsoft. Microsoft Developer Network: C#. `https://msdn.microsoft.com/pl-pl/library/kx37x362.aspx`. [dostęp 9.02.2017].

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