Uniwersytet Kardynała Stefana Wyszyńskiego
Wydział Matematyczno-Przyrodniczy
SZKOŁA NAUK ŚCISŁYCH

ZAGADNIENIE NAJKRÓTSZEGO CZASU PRZEJAZDU W SIECI MIEJSKIEJ (ZMIENNE W CZASIE PARAMETRY RUCHU DROGOWEGO)

Igor Litner
Bartosz Chreścionko
Paweł Jeleń

Warszawa 2021

Spis treści

1. Wstęp.

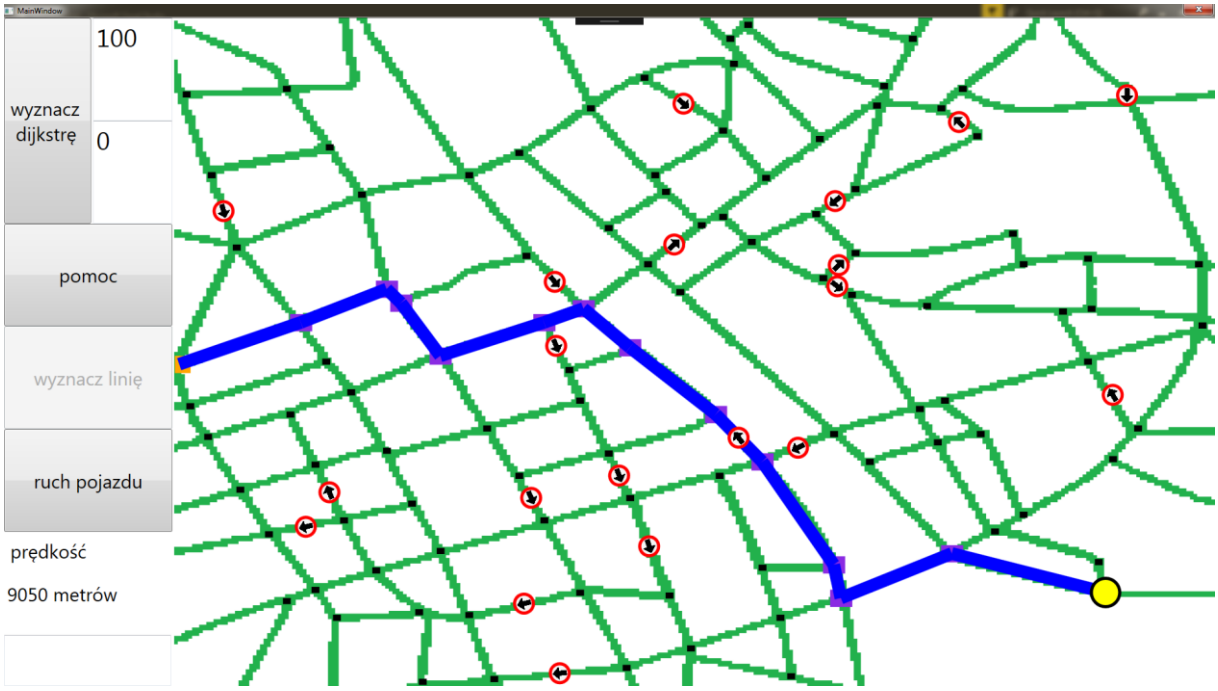
Zagadnienie przejazdu z punktu A do punktu B jest ważnym zagadnieniem optymalizacyjnym ze względów ekonomicznych. W zależności od potrzeb danej osoby lub firmy możemy rozpatrywać optymalizację względem najkrótszej drogi przejazdu, najszybszego czasu przejazdu, czy też najmniejszego spalania paliwa. Niniejsza praca opisuje zagadnienie najkrótszego czasu przejazdu w sieci miejskiej, co w tym wypadku oznacza, że w pewnym momencie dana droga będzie pusta, przez co będzie można nią przejechać szybko, a po chwili ta droga może się zakorkować, co spowolni przejazd nią. Jest to oczywiście zmienne w czasie oraz względnie losowe. W celu lepszej wizualizacji problemu, będzie można zasymulować korek w sposób deterministyczny. Miasto posiada też drogi jednokierunkowe, co często będzie oznaczało w praktyce, że droga z A do B będzie inna niż droga z B do A.

Program odwzorowuje to zagadnienie, dla środkowej części Warszawy, dla głównych dróg w Warszawie. Został on napisany w języku zorientowanym obiektowo – C# [1] w środowisku programistycznym Microsoft Visual Studio 2017 [2]. Interfejs użytkownika bazuje na WPF, czyli Windows Presentation Foundation [3] - nazwa silnika graficznego i API bazującego na .NET 3. Algorytmem liczącym najkrótszą (najszybszą) ścieżkę w grafie utworzonym na podstawie mapy miasta jest algorytm Dijkstry [4].

2. Opis modelu.

2.1. Założenia.

Na potrzeby programu została wycięta środkowa część mapy Warszawy. Ulice, które są na mapie, to główne arterie w Warszawie. Ma to zastosowanie praktyczne, ponieważ jeździ się nimi szybciej. One są najczęściej wielopasmowe.



Na mapie jest 107 skrzyżowań – małe czarne prostokąty, które połączone są ze sobą zielonymi liniami – drogami. Mapa jest odwzorowana w postaci grafu skierowanego, gdzie małe czarne prostokąty są skrzyżowaniami, odpowiadającymi wierzchołkom w grafie. Drogi stanowią zielone linie, których odpowiednikami w grafie są krawędzie. Natomiast białym kołem z czerwonym obwodem i czarną strzałką zaznaczono drogi z jednym kierunkiem jazdy, które w grafie są krawędziami skierowanymi. Oznacza to, że można jechać tylko zgodnie z kierunkiem wskazanym przez znak. Trasę na mapie tworzy ciąg dróg i skrzyżowań. Odpowiednikiem trasy na mapie jest ścieżka w grafie.

Dzięki temu możemy stworzyć macierz sąsiedztwa. W tym wypadku będzie to macierz koincydencji, czyli macierz kwadratowa, która w miejscu **ij** zawiera czas przejazdu ze skrzyżowania **i** do skrzyżowania **j** dla takich **i**, **j**, które można połączyć jedną drogą oraz 0 w przeciwnym przypadku. Macierz ta jest niezbędna do wyznaczenia optymalnej ścieżki w grafie.

Program wykorzystuje macierz sąsiedztwa do znalezienia wag w grafie. Ponadto, wierzchołki startowy i końcowy odpowiadają punktom startowemu i końcowemu na mapie. Zarówno macierz jak i punkty są niezbędne do algorytmu Dijkstry, z uwzględnieniem odległości (rzeczywistej w kilometrach) skrzyżowań od siebie, jednokierunkowości i ograniczeń prędkości tymczasowych, które są zależne od stałych ograniczeń prędkości. Algorytm Dijkstry wyznacza optymalną ścieżkę w grafie; na jej podstawie na mapie tworzona jest trasa. Trasa z kolei jest złożona z dróg, którymi najszybciej przejedziemy z punktu startowego do końcowego.

W programie zastosowano następujące ograniczenia:

1. Ograniczenia długości dróg. Każda droga ma swoją długość wyrażoną w kilometrach.

2. Stałe ograniczenia prędkości na drogach. W rzeczywistości działają one jak znak drogowy ograniczający prędkość. Ograniczenia stałe losowane są ze zbioru $\{30\frac{km}{h}, 40\frac{km}{h}, 50\frac{km}{h}, 60\frac{km}{h}, 70\frac{km}{h}\}$ – są to realne prędkości na drogach w Warszawie w godzinach szczytu. Oznacza to, że na danej drodze nie można jechać szybciej niż wynosi jej ograniczenie stałe.

3. Tymczasowe ograniczenia prędkości na drogach. One też są losowane ze zbioru $\{30\frac{km}{h}, 40\frac{km}{h}, 50\frac{km}{h}, 60\frac{km}{h}, 70\frac{km}{h}\}$, ale potem wybieramy MIN(wylosowana wartość, ograniczenie stałe). Istotne jest, że stałe ograniczenie prędkości jest ważniejsze niż tymczasowe, czyli jeśli droga ma stałe ograniczenie prędkości $40\frac{km}{h}$ i zostanie wylosowane tymczasowe ograniczenie prędkości $60\frac{km}{h}$ to ta droga będzie miała ograniczenie prędkości $40\frac{km}{h}$. Po dojechaniu auta do skrzyżowania, z prawdopodobieństwem $\frac{1}{3}$ następuje losowanie, czy tymczasowe ograniczenia prędkości dróg się zmieniły. Jeśli prawdopodobieństwo byłoby większe, to wariancja byłaby zbyt duża. Wtedy trasa przejazdu mogłaby się zmieniać praktycznie co każde skrzyżowanie. W rzeczywistości warunki na drogach ulegają zmianie, lecz nie za często.

4. Ograniczenia wynikające z jednokierunkowości. Niektóre drogi (niezgodnie z rzeczywistością) są jednokierunkowe, co oznacza, że jeśli można przejechać z punktu A do punktu B, to nie można odwrotnie - mogą to być roboty drogowe lub inne utrudnienia, które wyłączają dany pas ruchu i w efekcie stworzy się droga jednokierunkowa.

5. Ograniczenia spowodowane korkami na drogach. Istnieje możliwość wprowadzenia „korka” w sposób deterministyczny. Można zasymulować korek, poprzez wygenerowanie ograniczenia prędkości tymczasowej równej $10\frac{km}{h}$ na drodze wygenerowanej trasy, co w efekcie w większości przypadków wymusi na programie znalezienie innej trasy.

2.2. Schemat krokowy algorytmu Dijkstry.

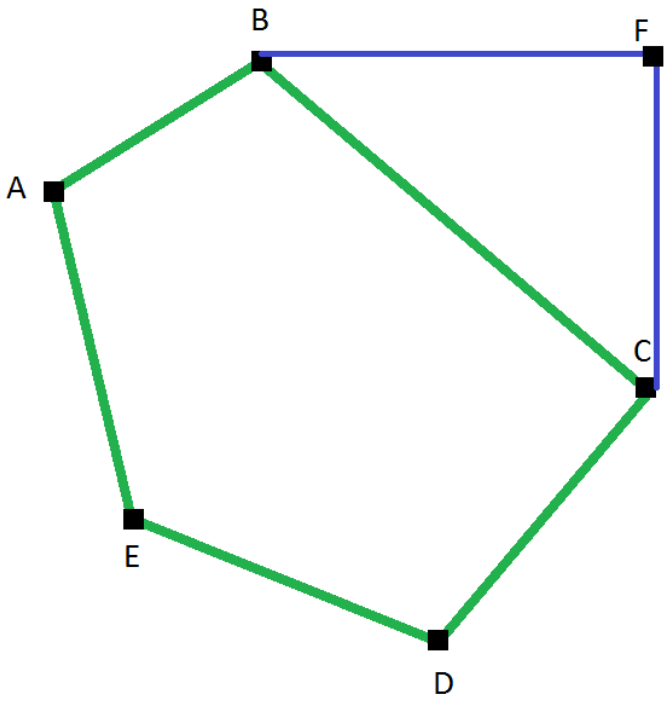
Krok 1. Wybierz punkt startowy **s** i nadaj mu cechę stałą 0 oraz punkt końcowy **t**.
Krok 2. Wprowadzamy dwa zbiory **S** – zbiór wierzchołków z cechami stałymi oraz **G** – zbiór wierzchołków z cechami tymczasowymi, **s** należy do **S**, inne wierzchołki należą do **G**.
Krok 3. Nadaj pozostałym wierzchołkom, należącym do zbioru **G**, cechę tymczasową równą nieskończoność.
Krok 4. Wszystkie wierzchołki sąsiednie $\mathbf{w_i}$ wierzchołka **s** otrzymują cechę tymczasową równą d(**s**, $\mathbf{w_i}$), gdzie d jest czasem przejazdu z wierzchołka **s** do $\mathbf{w_i}$.
Krok 5. Spośród wszystkich sąsiadów **s** wybieramy taki $\mathbf{w_k}$, który ma najmniejszą cechę tymczasową d(**s**, $\mathbf{w_k}$) i jego cecha tymczasowa staje się cechą stałą.
Krok 6. Usuwamy wierzchołek $\mathbf{w_k}$ ze zbioru **G** i dodajemy do zbioru **S**.
Krok 7. Dla wierzchołka $\mathbf{w_k}$ liczymy cechy tymczasowe równe d($\mathbf{w_k}$, $\mathbf{w_j}$) jego sąsiadów, którzy należą do zbioru **G**.
Krok 8. Sprawdzamy, czy spełniony jest warunek d(**s**, $\mathbf{w_i}$) > d(**s**, $\mathbf{w_k}$) + d($\mathbf{w_k}$, $\mathbf{w_j}$).
Krok 8.1. Jeśli warunek jest spełniony to d(**s**, $\mathbf{w_j}$) = d(**s**, $\mathbf{w_k}$) + d($\mathbf{w_k}$, $\mathbf{w_j}$).
Krok 8.2. Jeśli warunek nie jest spełniony to d(**s**, $\mathbf{w_j}$) zostaje niezmienione.
Krok 9. W zbiorze **G** szukamy takiego **x**, który ma najmniejszą cechę tymczasową. Otrzymuje on cechę stałą oraz zostaje usunięty ze zbioru **G** i dodany do zbioru **S**.
Krok 10. Sprawdzamy warunek, czy **x** = t.
Krok 10.1. Jeżeli warunek jest prawdziwy, to kończymy algorytm.
Krok 10.2. Jeżeli warunek nie jest prawdziwy, to przechodzimy do kroku 5, tylko zamiast **s** bierzemy **x**.

2.3. Schemat krokowy algorytmu wyznaczania trasy o najkrótszym czasie przejazdu.

Krok 1. Wprowadzamy punkt początkowy i końcowy.
Krok 2. Wprowadzamy odległości między skrzyżowaniami, które są obliczane w sposób przybliżony za pomocą twierdzenia Pitagorasa.
Krok 3. Oznaczamy na mapie pewne drogi, które są jednokierunkowe.
Krok 4. Wprowadzamy losowo stałe ograniczenia prędkości na drogach.
Krok 5. Jeśli drogi nie mają jeszcze nadanych ograniczeń prędkości tymczasowych, wprowadzamy w sposób losowy tymczasowe ograniczenia prędkości dla dróg między skrzyżowaniami z uwzględnieniem ograniczeń stałych i jednokierunkowości.
Krok 6. Jeśli drogi mają nadane tymczasowe ograniczenia prędkości, to z prawdopodobieństwem $\frac{2}{3}$ te prędkości zostają jakie były, a z prawdopodobieństwem $\frac{1}{3}$ wprowadzamy w sposób losowy ograniczenia prędkości tymczasowych dla dróg między skrzyżowaniami z uwzględnieniem ograniczeń stałych i jednokierunkowości.
Krok 7. Wyznaczamy wagi krawędzi grafu, które są czasami przejazdu na odpowiadających im drogach.
Krok 8. Zauważmy, że program może wylosować różne ograniczenia prędkości tymczasowej, co oznacza, że często w dwóch różnych kierunkach ta sama krawędź będzie mieć inną wagę.
Krok 9. Za pomocą algorytmu Dijkstry wyznaczamy ścieżkę o najkrótszym czasie przejazdu na podstawie grafu.

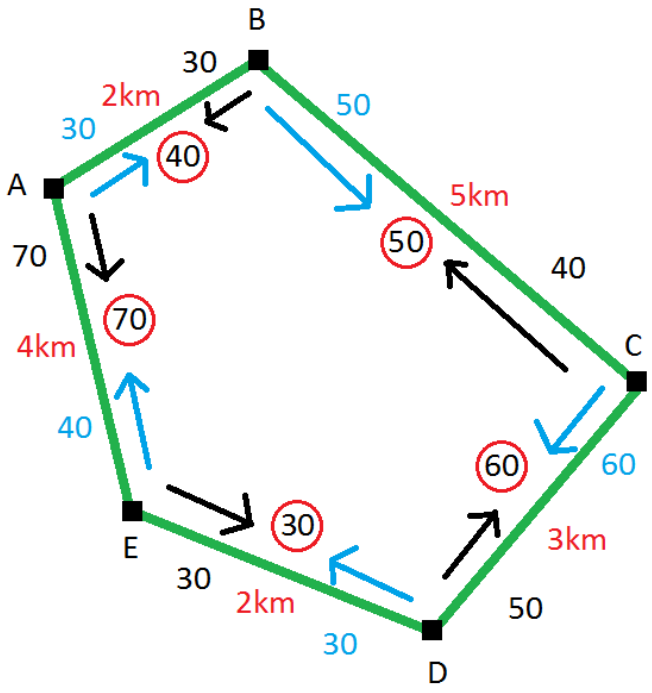
Krok 10. Na podstawie wyznaczonej ścieżki w grafie, znajdujemy odpowiadającą jej trasę na mapie.

Poniższy przykład ilustruje **krok 2.**



Na powyższym obrazie widzimy przykładowy wycinek mapy programu. Z twierdzenia Pitagorasa wiemy, że długość $BC = \sqrt{CF^2 + BF^2}$, gdzie BF to ilość pikseli w poziomie, a CF w pionie. Żeby otrzymać długość drogi w kilometrach, musimy obliczone BC pomnożyć przez skalę mapy. Jest to przybliżenie, ale na potrzeby działania programu jest wystarczająco dobre.

Poniższy przykład ilustruje **krok 6.** i **7.**



Objaśnienia:

- liczby w obwodzie symbolizują ograniczenia prędkości stałej w $\frac{km}{h}$,

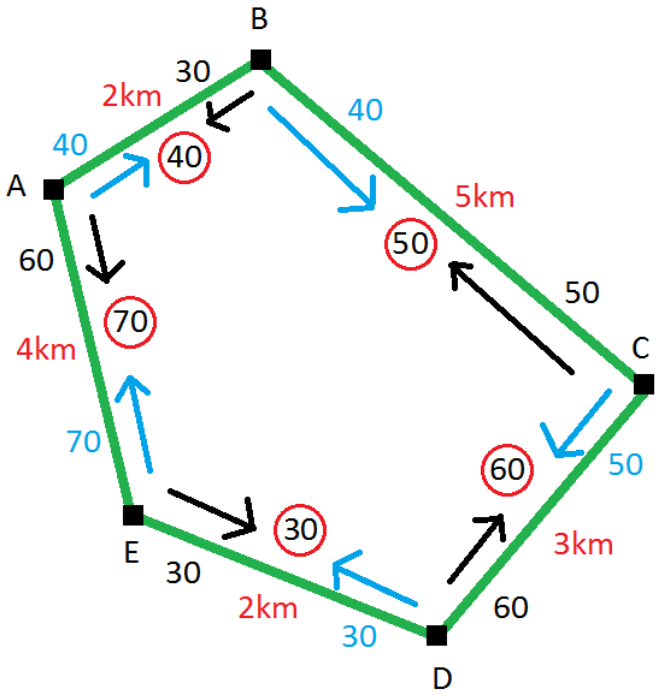
- liczby w kolorze czerwonym oznaczają długości dróg w kilometrach,

- liczby niebieskie są ograniczeniami prędkości tymczasowych na drogach z kierunkiem strzałki niebieskiej,

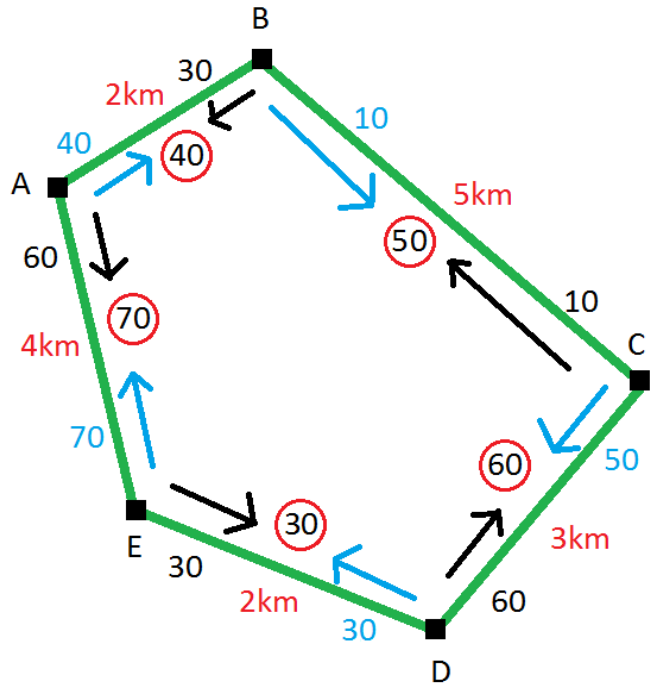
- liczby czarne są ograniczeniami prędkości tymczasowych na drogach z kierunkiem strzałki czarnej.

Załóżmy, że chcemy jechać z D do B. Czas trasy DCB jest to suma czasów DC i CB, czyli $\frac{3}{50}h + \frac{5}{40}h$ = 11min 6s. Z kolei trasa DEAB wynosi $\frac{2}{30}h + \frac{4}{40}h + \frac{2}{30}h$ = 14min 0s. Algorytm wybierze trasę DCB.

Po przejechaniu do punktu C z prawdopodobieństwem $\frac{2}{3}$ ograniczenia prędkości tymczasowych się nie zmienią, a z prawdopodobieństwem $\frac{1}{3}$ nastąpi ponowne losowanie. Załóżmy, że doszło do ponownego losowania.



W tej sytuacji dojechanie z C do B zajmie $\frac{5}{50}h$ = 6min 0s. Jeśli jednak nie zmieniłyby się ograniczenia prędkości tymczasowych, to zajęłoby to $\frac{5}{40}h$ = 7min 30s. Załóżmy teraz, że dojechaliśmy do B i chcemy pojechać do C, ale pojawił się tam korek.



W tej sytuacji załóżmy, że chcemy przejechać z B do C. Trasą BC zajmie to $\frac{5}{10}h$ = 30min 0s. Z kolei trasa BAEDC wyniesie $\frac{2}{30}h + \frac{4}{60}h + \frac{2}{30}h + \frac{3}{60}h$ = 15min 0s. Algorytm wybierze trasę BAEDC.

2.4. Schematy działania programu.

2.4.1. Schemat jednorazowy.

Krok 1. Uruchom plik wykonywalny.
Krok 2. Wprowadź mapę do programu razem z drogami i skrzyżowaniami.
Krok 3. Wybierz punkt startowy i końcowy.
Krok 4. Wprowadź ograniczenia długości dróg, ograniczeń prędkości stałych i tymczasowych oraz jednokierunkowości.
Krok 5. Zastosuj algorytm wyznaczania trasy na mapie od punktu startowego do końcowego.
Krok 6. Zaznacz trasę niebieską krzywą łamaną.

Poniżej zilustrowany został przykład schematu jednorazowego.

W tym schemacie na mapie trasa jest zaznaczona niebieską krzywą łamaną. Nie występują korki oraz warunki na drodze nie zmieniają się w czasie. Oznacza to, że raz wyznaczona trasa jest niezmienna.

2.4.2. Schemat iteracyjny.

W schemacie iteracyjnym warunki na drodze mogą się zmienić, co każde przejechane skrzyżowanie. Możliwe jest też wprowadzenie korka, co wymusi na programie znalezienie innej trasy. W celu lepszej wizualizacji rozwiązania problemu wprowadzamy pojęcie samochodu oraz punkt początkowego, który niekoniecznie musi być punktem startowym.

Samochód, w celu czysto wizualizacyjnym, będzie jechał zmieniającą się trasą od skrzyżowania do skrzyżowania. Punkt początkowy od punktu startowego różni się tym, że za każdym razem trasa jest wyznaczana od punktu, w którym aktualnie znajduje się samochód do końcowego, a nie od punktu startowego do punktu końcowego. Oznacza to, że po przejechaniu samochodem jednej drogi, mogą się zmienić tymczasowe ograniczenia prędkości na mapie, co w efekcie może zmienić trasę przejazdu.

Krok 1. Uruchom plik wykonywalny.
Krok 2. Wprowadź mapę do programu razem z drogami i skrzyżowaniami.
Krok 3. Wybierz punkt początkowy i końcowy.
Krok 4. Wprowadź ograniczenia długości dróg, ograniczeń prędkości stałych i tymczasowych oraz jednokierunkowości.
Krok 5. Następuje losowanie czy ograniczenia prędkości tymczasowych się zmienią. Z prawdopodobieństwem $\frac{1}{3}$ nastąpi zmiana tych ograniczeń, w przeciwnym wypadku nie zmienią się.
Krok 6. Zastosuj algorytm wyznaczania trasy na mapie od punktu początkowego do końcowego.
Krok 7. Zaznacz trasę niebieską krzywą łamaną.
Krok 8. Jeśli nie chcesz dodać korka, przejdź do punktu 9.
Krok 9. Jeśli chcesz dodać korek, kliknij na wybrany fragment trasy; następnie wróć do punktu 5.
Krok 10. Wykonaj ruch samochodu do skrzyżowania.
Krok 11. Jeśli samochód nie znalazł się w punkcie końcowym, to punkt początkowy staje się punktem, w którym aktualnie znajduje się samochód, wróć do punktu 4.
Krok 12. Jeśli samochód znalazł się w punkcie końcowym, zakończ program.

Poniżej zilustrowany został przykład schematu iteracyjnego.

W tym schemacie możemy dodać korek na rysunku zaznaczony na pomarańczowo. W efekcie wyznaczy nam to inną trasę.

Trasa może też się zmienić losowo. Na powyższym przykładzie przejedźmy jedno skrzyżowanie.

Trasa zmieniła się bez ingerencji użytkownika. Oznacza to, że nastąpiła zmiana ograniczeń prędkości tymczasowych na drogach.

3. Opis struktury programu.

3.1. Opis struktury.

3.1.1. Opis środowiska.

Program został napisany w języku C#[1] w środowisku Microsoft Visual Studio 2017[2] z uwzględnieniem kompatybilności z wersją Microsoft Visual Studio 2013. W programie występują trzy klasy – MainWindow, Dijkstra oraz Cross. W programie jest wiele plików; między innymi zdjęcia Warszawy, lista skrzyżowań, instrukcja obsługi. Interfejs programu jest przedstawiony za pomocą WPF – Windows Presentation Foundation[3]. Wersja frameworku wykorzystywana w programie to .NET Framework 4.5.

Zalecane parametry sprzętowe potrzebne do uruchomienia to:

- System operacyjny Windows XP / Vista / 2003 / 7/ 8 / 10,

- Procesor 1,6 GHz lub szybszy,

- 1 GB pamięci RAM (1,5 GB w przypadku uruchamiania na maszynie wirtualnej),

- 1 GB dostępnego miejsca na dysku twardym,

- Dysk twardy o szybkości 5,400 obrotów na minutę,

- Karta wideo obsługująca technologię DirectX 9 i rozdzielczość ekranu 1024 x 768 lub wyższą,

- Środowisko Microsoft Visual Studio 2013 lub nowsze,

- Framework .Net 4.5 lub nowszy.

3.1.2. Opis klas.

W programie znajdują się następujące klasy:

- **Cross** – klasa, która reprezentuje skrzyżowanie; ta klasa zawiera w sobie tylko definicję skrzyżowania, czyli: indeks, współrzędne x i y, listę sąsiadów, listę odległości od sąsiadów oraz listę prędkości do danych sąsiadów. Klasa ta nie posiada żadnych metod; można powiedzieć, że jest to zwykła struktura;

- **Dijkstra** – klasa, w której zaimplementowany jest algorytm Dijkstry; główną metodą tej klasy jest metoda int[] Dijkstra(double[,] matrix, int Start), która zwraca listę, która dla każdego indeksu ma wartość jego „rodzica"(rodzicem punktu startowego jest -1, rodzicem sąsiadów punktu startowego jest punkt startowy itd.) w taki sposób, że droga od punktu końcowego do startowego będzie najszybsza; klasa ma też metody wypisujące drogę do konsoli oraz metodę zwracającą konkretną ścieżkę dla punktu startowego i końcowego;

- **MainWindow** – klasa główna; dzieli się ona

    - **MainWindow.xaml** – ta część klasy odpowiada za GUI, czyli wizualizację programu, układ przycisków, pól to wpisywania itp.;

    - **MainWindow.cs** – ta część klasy odpowiada za kod wszystkich przycisków, pól do wpisywania itp. W niej wielokrotnie do różnych operacji jest używana lista wszystkich 107 skrzyżowań. W jednym z guzików wykorzystuje się tą listę do wyznaczenia optymalnej trasy za pomocą klasy **Dijkstra**.

3.1.3. Opis plików.

Program posiada następujące pliki:

- **Cross.cs** – plik zawierający klasę **Cross**;

- **Dijkstra.cs** – plik zawierający klasę **Dijkstra**;

- **MainWindow.xaml** – plik, w którym jest interfejs w języku XAML – Extensible Aplication Markup Language; odpowiada on za układ przycisków, pól tekstowych itp.;

- **MainWindow.cs** – plik, w którym jest kod odpowiedzialny za całość działania programu, czyli przyciski, pola tekstowe, samochód itp.;

- **help.txt** – plik tekstowy, który jest wyświetlany, po wciśnięciu przycisku pomoc; zapisana jest w nim instrukcja krok po kroku, co należy robić, żeby poprawnie korzystać z programu;

- **Warszawa.png** – plik źródłowy uproszczonej mapy drogowej Warszawy; mapa ta zawiera główne drogi w Warszawie;

- **Warszawa2.png, Warszawa3.png** – pliki, które zostały wyrenderowane poprzez obróbkę pliku Warszawa.png, pierwszy z nich zaznaczył drogi na czerwono, a drugi do białego obrazka skopiował tylko czerwone drogi;

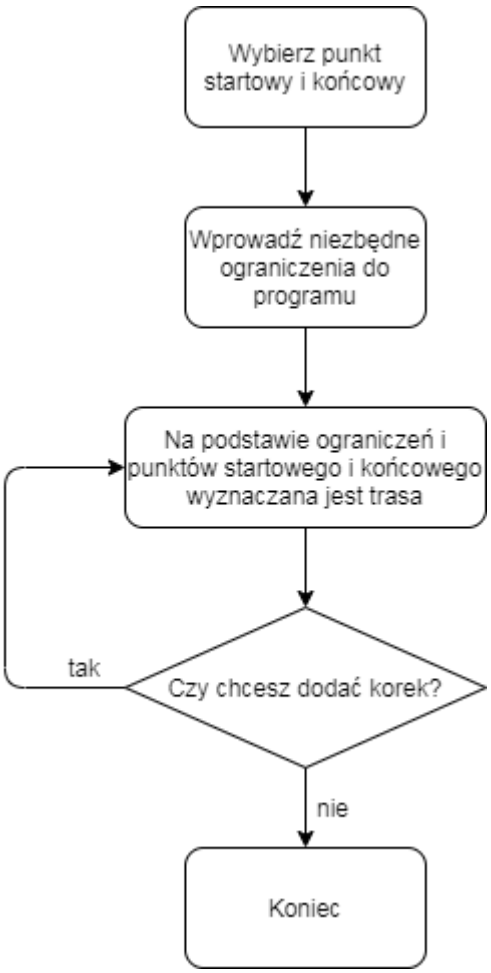
- **Crosses.png** – plik finalny mapy Warszawy, który jest używany w programie; z pliku **Warszawa3.png** wycięto środkowy fragment Warszawy, drogi pokolorowano na zielono, a skrzyżowania oznaczono czarnymi prostokątami;

- **Crosses.txt** – plik, w którym zapisane są skrzyżowania z ich sąsiadami; w trakcie uruchamiania program pobiera dane z tego pliku;

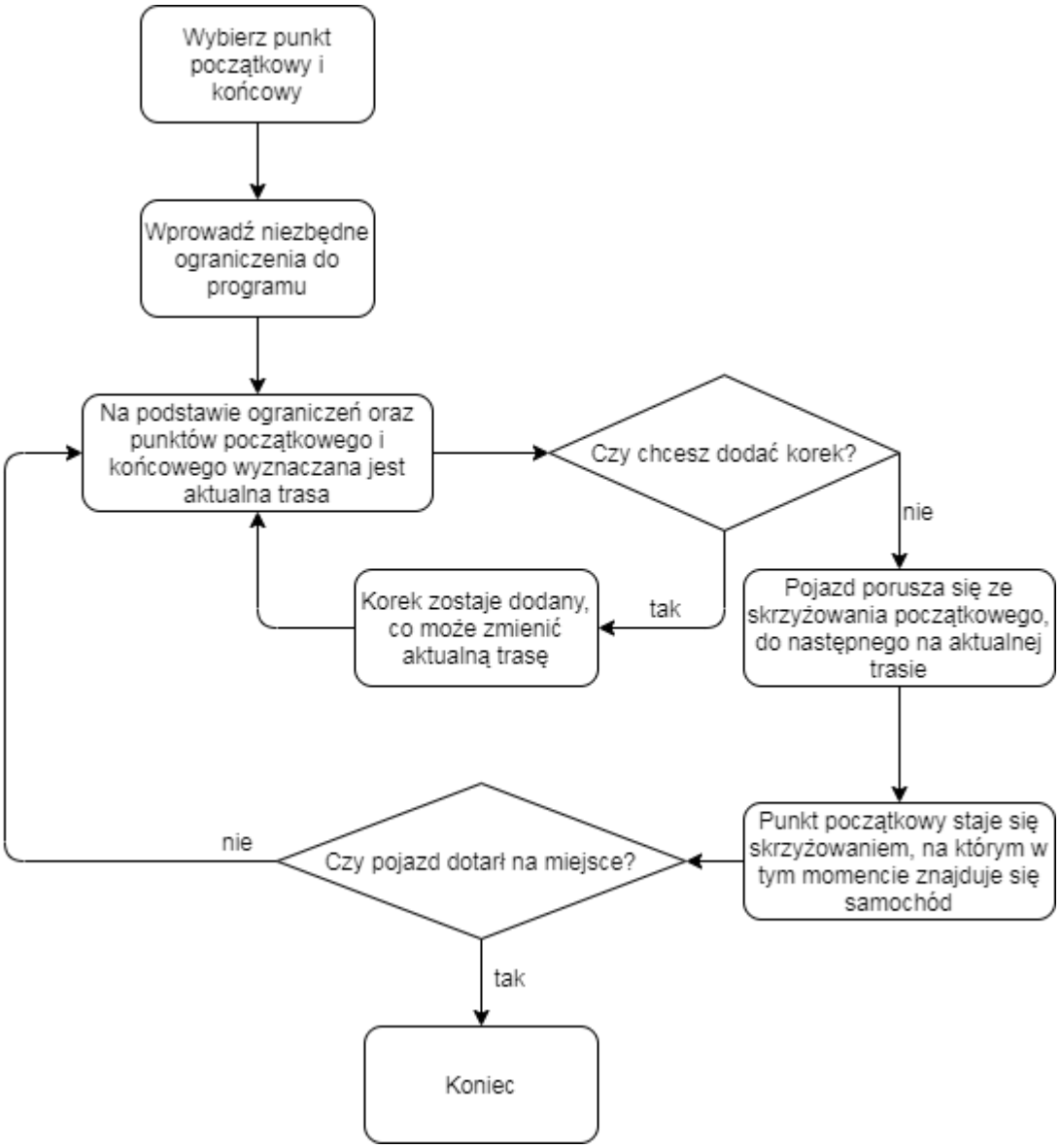
- **left arrow.jpg** – plik, w którym jest zdjęcie lewej strzałki, która jest używana, do zaznaczania dróg jednokierunkowych; w programie jest ona obracana o odpowiedni kąt.

3.2. Schematy blokowe.

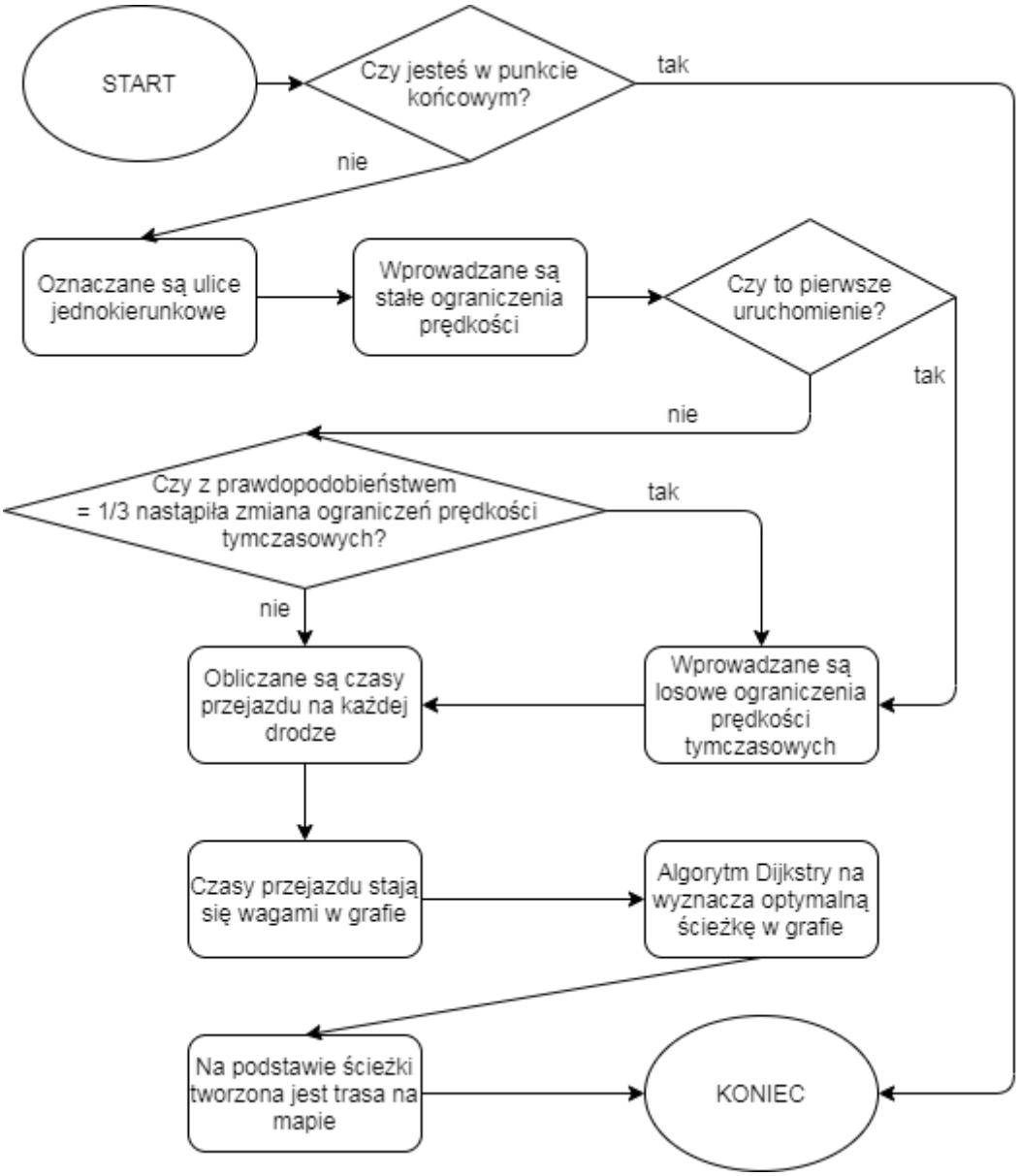
3.2.1. Schemat blokowy działania programu, dla jednorazowego wyznaczenia trasy.



3.2.2. Schemat blokowy działania programu, dla iteracyjnego wyznaczenia trasy.

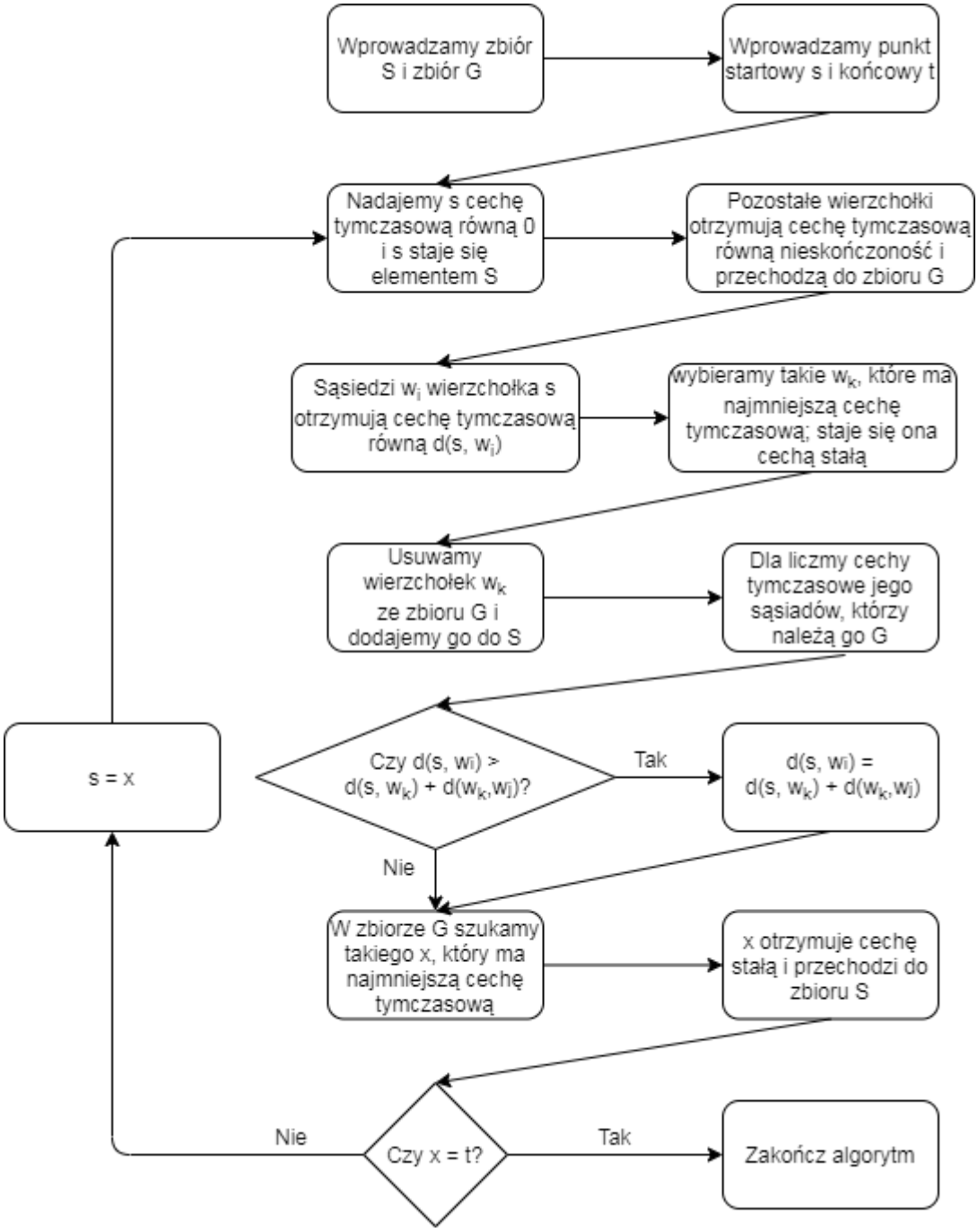


3.2.3. Schemat blokowy wyznaczania trasy.



3.2.4. Schemat blokowy algorytmu Dijkstry.

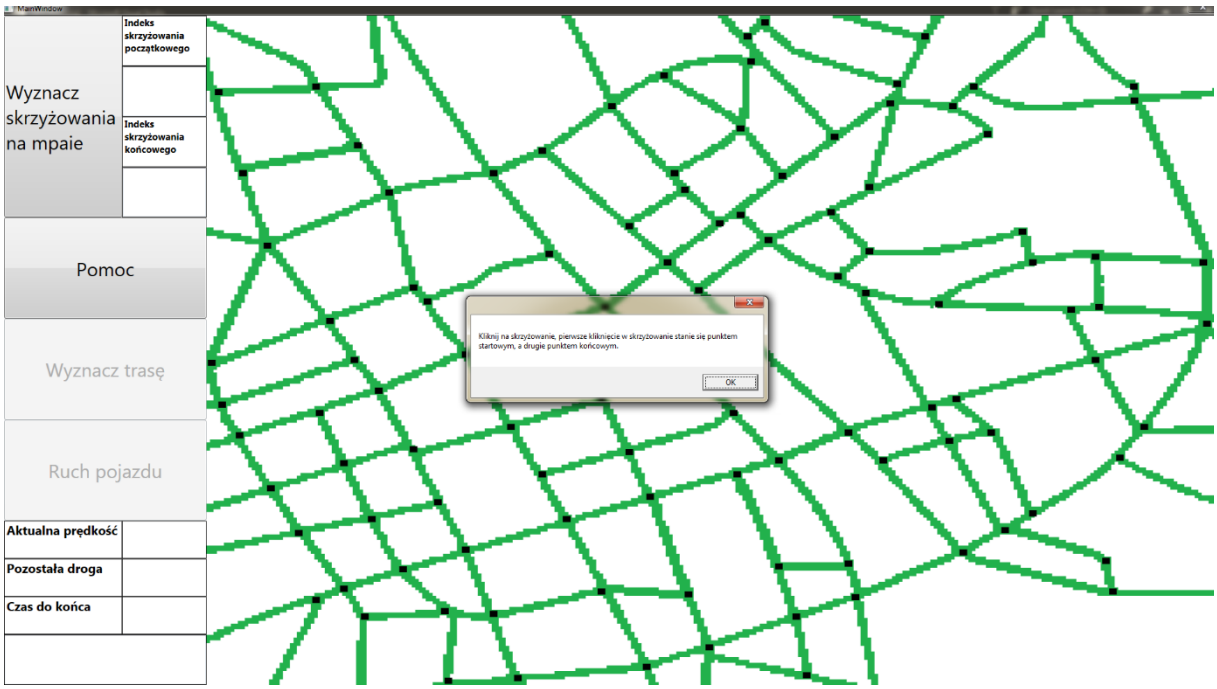
Zbiór **S** – zbiór wierzchołków z cechami stałymi
Zbiór **G** – zbiór wierzchołków z cechami tymczasowymi
d(a,b) – waga krawędzi łączącej wierzchołek a z wierzchołkiem b na grafie



3.3. Interfejs.

Interfejs opracowano w języku C#[1] przy wykorzystaniu frameworku WPF – Windows[2] Presentation Foundation z użyciem XAML - Extensible Application Markup Language[3].

Poniżej znajduje się ekran startowy.



Pojawia się komunikat, o wybraniu skrzyżowania startowego i końcowego. W lewym górnym rogu widzimy **Indeks skrzyżowania początkowego** oraz **Indeks skrzyżowania końcowego**. Po wybraniu skrzyżowań, pojawią się w ich miejsce indeksy. Aktywnymi przyciskami są **Wyznacz skrzyżowania na mapie** oraz **pomoc**. **Wyznacz skrzyżowania na trasie** zaznacza na mapie skrzyżowania, którymi będzie poruszał się samochód. **Pomoc** wyświetla podstawowe informacje i instrukcje. Na dole napisy **Aktualna prędkość**, **pozostała droga** i **Czas do końca**, będą wyświetlać kolejno aktualną prędkość samochodu, odległości między samochodem i skrzyżowaniem końcowym w trakcie jazdy oraz czas docelowy. Przycisk **Wyznacz trasę** rysuje niebieską krzywą łamaną na mapie, wzdłuż której będzie poruszał się samochód. Kliknięcie na niebieską linię zmieni jej kolor na pomarańczowy, co jest równoznaczne z korkiem na danej drodze. Po narysowaniu trasy **Ruch pojazdu** przesuwa samochód do następnego skrzyżowania. Na dole jest pole, w którym można wpisać liczby od 0 do 106, co podświetli na mapie skrzyżowanie z jego sąsiadami.

3.4. Wybrane fragmenty kodu.

3.4.1. Ruch samochodu.

Ten fragment kodu odpowiada za poruszanie się samochodem na mapie.

```
        //obliczanie współrzędnych x0,x1,y0,y1
519     double x0 = crosses[route[counter]].x * map.ActualWidth / (double)bitmap.Width;
520     double x1 = crosses[route[counter + 1]].x * map.ActualWidth /
        (double)bitmap.Width;
521     double y0 = crosses[route[counter]].y * map.ActualHeight /
        (double)bitmap.Height;
522     double y1 = crosses[route[counter + 1]].y * map.ActualHeight /
        (double)bitmap.Height;

        //wyznaczenie tangensa kąta nachylenia
523     double x = x1 - x0;
524     double y = y1 - y0;
525     angle = y / x;

        //warunki sprawdzające, czy kąt należy do (0,90), czy (90,180) stopni
526     if (y < 0 && x > 0)
527           angle = -Math.Abs(angle);
528     if (y < 0 && x < 0)
529           angle = -Math.Abs(angle);
530     if (y > 0 && x < 0)
531           angle = Math.Abs(angle);
532
        //obliczanie nowego miejsca samochodu
533     var newPlace = car.Margin;
534     if (crosses[route[counter + 1]].x < crosses[route[counter]].x)
535           newPlace.Left--;
536     else
537           newPlace.Left++;
538     newPlace.Top = newPlace.Top + angle;
539     car.Margin = new Thickness(newPlace.Left, newPlace.Top, 0, 0);
540
541     //kasowanie przejechanej drogi
542     if (x > 0)
543           roads[route.Count - counter - 2].X2++;
544     else
545           roads[route.Count - counter - 2].X2--;
546     roads[route.Count - counter - 2].Y2 += angle;
```

W linijkach **519-522** obliczamy współrzędne dwóch skrzyżowań trasy – ($\mathbf{x_0}$,$\mathbf{y_0}$) oraz ($\mathbf{x_1}$,$\mathbf{y_1}$). Następnie w linijce **525** obliczamy nachylenie, pod jakim pojazd będzie się poruszał po mapie (do góry lub do dołu) oraz w linijkach **526-531** kierunek (w prawo lub w lewo). W linijkach **533-537** przesuwamy pojazd o jeden piksel w prawo lub w lewo, a w linijce **538** przesuwamy o jeden piksel pomnożony przez wartość nachylenia. W linijce **539** samochód przesuwa się na wcześniej obliczone miejsce. Następnie kasowany jest fragment przejechanej drogi.

3.4.2. Losowanie prędkości i jednokierunkowości.

Ten fragment kodu odpowiada za tymczasowe ograniczenia prędkości oraz jednokierunkowość.

```
        //inicjalizacja generatorów liczb losowych
266     var rnd = new Random(0);
267     var rnd2 = new Random();
        //sprawdzenie, czy nastąpiła zmiana ograniczeń prędkości tymczasowych
268     if (rnd2.Next(3)==0 || crosses[0].velocity.Count == 0)
269     {
            //usuwanie starych ograniczeń
270         for (int i = 0; i < crosses.Count; i++)
271             crosses[i].velocity.Clear();
272         for (int i = 0; i < crosses.Count; i++)
273         {
274             for (int j = 0; j < crosses[i].neighbours.Count; j++)
275             {
                    //inicjalizacja dróg jednokierunkowych
276                 int x = rnd.Next(0, 14);
277                 if (x == 0)
278                     crosses[i].velocity.Add(0);
279                 else
280                 {
                        //nadawanie prędkości 30-70
281                     int vel = rnd2.Next(5);
282                     crosses[i].velocity.Add(30 + 10 * vel);
283                 }
284             }
285         }
286     }
```

Jest to prosty kod, który na początku używa dwóch generatorów liczb losowych. Następnie mamy warunek, który sprawdza, czy nastąpiło zdarzenie losowania nowych ograniczeń prędkości tymczasowych dla dróg. Jeśli tak się zdarzyło, to w linijkach **270-271** usuwane są poprzednie prędkości dróg. W linijkach **276 – 278** losujemy jednokierunkowość. Dzięki stałemu ziarnu generatora, oznaczenie drogi jednokierunkowej pozostaje bez zmian. Następnie losowane są ograniczenia prędkości tymczasowych dla dróg pozostałych. Jeśli to ograniczenie byłoby większe niż ograniczenie prędkości stałej, to metoda .Add to wychwyci i nada ograniczenie stałej prędkości równą ograniczeniu stałemu.

3.4.3. Macierz incydencji.

Ten fragment kodu tworzy macierz incydencji.

```
309     var graph = new double[crosses.Count, crosses.Count];
        //pętla potrójna, dla każdego elementu o współrzędnych i,j
310     for (int i = 0; i < crosses.Count; i++)
311     {
312         for (int j = 0; j < crosses.Count; j++)
313         {
314             for (int k = 0; k < crosses[i].neighbours.Count; k++)
315             {
                    //sprawdzamy, czy dla połączenia i,j k-ty sąsiad jest równy j
316                 if (crosses[i].neighbours[k] == crosses[j].index)
317                 {
                        //wartość dla i,j w grafie to czas z i do j
318                     graph[i, j] = crosses[i].distance[k] * 1/
                        crosses[i].velocity[k];
319                     break;
320                 }
321                 else
                        //jeśli nie ma połączenia, to w macierzy wstawiamy odpowiednik
nieskończoności
322                     graph[i, j] = double.PositiveInfinity;
323             }
324         }
325     }
326     return graph;
```

W tej potrójnej pętli dla każdej pary skrzyżowań (**i,j**) ustalamy wagę w grafie równą czasowi przejazdu. Zauważmy, że jeżeli skrzyżowania nie są połączone, to wartość w grafie wynosi `double.PositiveInfinity`, czyli plus nieskończoność. W poprzednim podrozdziale nadaliśmy ograniczenie prędkości stałej drogom jednokierunkowym równą 0 dla kierunku pod prąd. W linijce **318** obliczamy czas przejazdu. Czas jest to droga podzielona przez prędkość. Droga jest zawsze dodatnia; prędkość w tym wypadku wynosi 0. Zatem dla dróg jednokierunkowych także otrzymamy plus nieskończoność. Algorytm Dijkstry nie bierze pod uwagę połączeń o wartościach nieskończonych.

3.4.4. Znak drogi jednokierunkowej.

Ten fragment kodu tworzy znaki jednokierunkowe w postaci strzałki.

```
        //strzałka jako obraz
372     ImageBrush left_arrow = new ImageBrush();
373     var uri_left = new
Uri(Directory.GetParent(Directory.GetCurrentDirectory()).Parent.FullName +
@"\Resources\left arrow.jpg");
374     left_arrow.ImageSource = new BitmapImage(uri_left);
375
376     //obracanie obrazu
377     RotateTransform leftTransform = new RotateTransform();
378     double X = crosses[crosses[i].neighbours[j]].x - crosses[i].x;
379     double Y = crosses[crosses[i].neighbours[j]].y - crosses[i].y;
380     angle = Math.Atan(Y / X) / Math.PI * 180;
381     leftTransform.CenterX = 0.5;
382     leftTransform.CenterY = 0.5;
383     leftTransform.Angle = angle;
384     left_arrow.RelativeTransform = leftTransform;
385
        //warunek obrotu strzałki o dodatkowe 180 stopni
386     if (crosses[i].x < crosses[crosses[i].neighbours[j]].x)
387     {
388         stops[count].Fill = left_arrow;
389         stops[count].StrokeThickness = 5;
390         stops[count].Stroke = System.Windows.Media.Brushes.Red;
391     }
392     else
393     {
394         leftTransform.Angle = angle + 180;
395         left_arrow.RelativeTransform = leftTransform;
396         stops[count].Fill = left_arrow;
397         stops[count].StrokeThickness = 5;
398         stops[count].Stroke = System.Windows.Media.Brushes.Red;
399     }
```

W linijkach **371-374** pobieramy obraz strzałki. W linijkach **377-384** obliczamy kąt, o jaki trzeba obrócić strzałkę. W linijkach **386-398** wypełniamy znak obróconą strzałką oraz nadajemy znakowi czerwony obwód. W **ifie** sprawdzamy, czy strzałkę należy dodatkowo obrócić o 180 stopni.

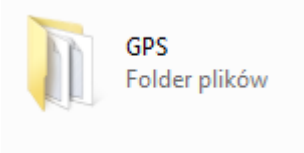
4. Instrukcja obsługi.

4.1. Wymagania sprzętowe i systemowe.

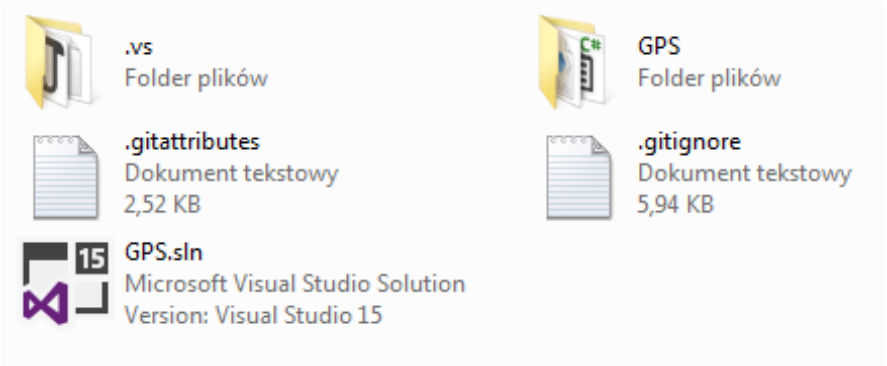
Zalecane parametry sprzętowe potrzebne do uruchomienia to:

- System operacyjny Windows XP / Vista / 2003 / 7/ 8 / 10,
- Procesor 1,6 GHz lub szybszy,
- 1 GB pamięci RAM (1,5 GB w przypadku uruchamiania na maszynie wirtualnej),
- 1 GB dostępnego miejsca na dysku twardym,
- Dysk twardy o szybkości 5,400 obrotów na minutę,
- Karta wideo obsługująca technologię DirectX 9 i rozdzielczość ekranu 1024 x 768 lub wyższą,
- Środowisko Microsoft Visual Studio 2013 lub nowsze,
- Framework .Net 4.5 lub nowszy.

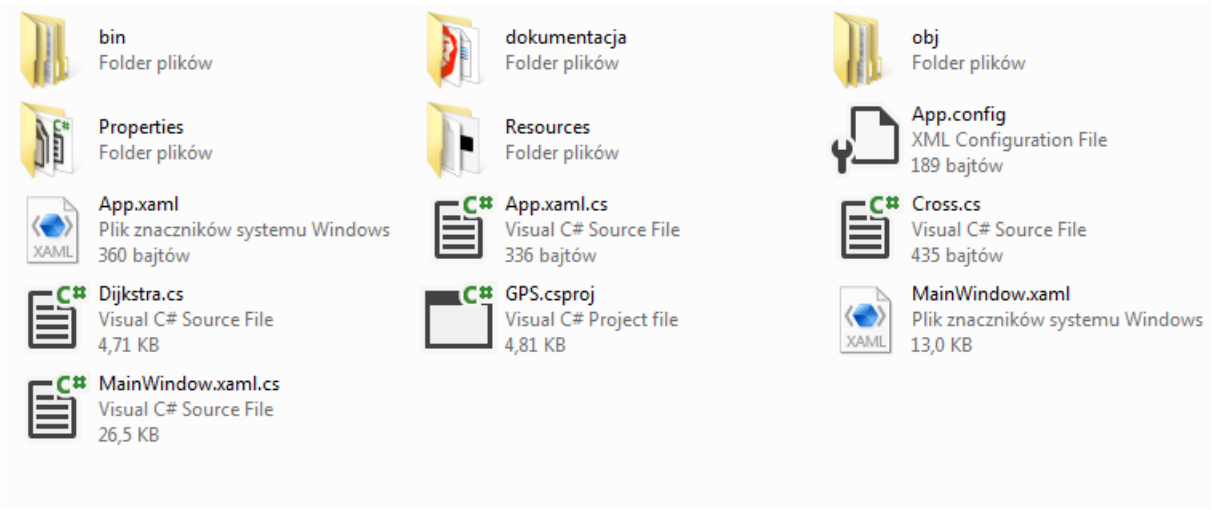
4.2. Zawartość nośnika



Jest to główny katalog. W nim znajdują się poniższe pliki.



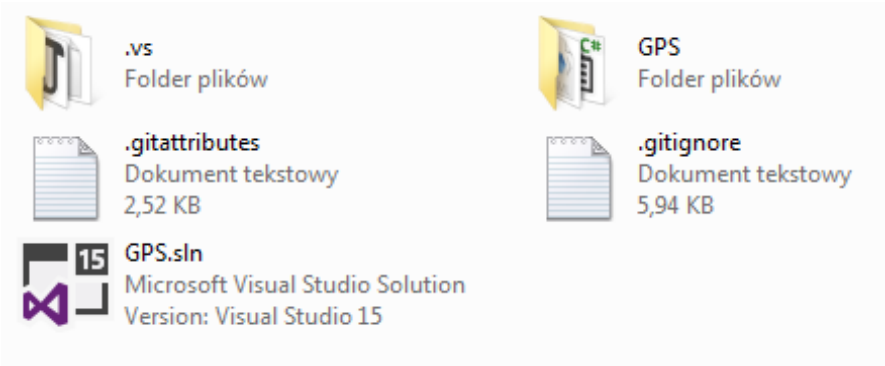
W katalogu .vs znajdują się pliki związane z Microsoft Visual Studio. Pliki .git są związane z githubem. Plik **GPS.sln** jest plikiem projektu do otwarcia za pomocą Microsoft Visual Studio. Poniżej znajduje się zawartość katalogu **GPS**.



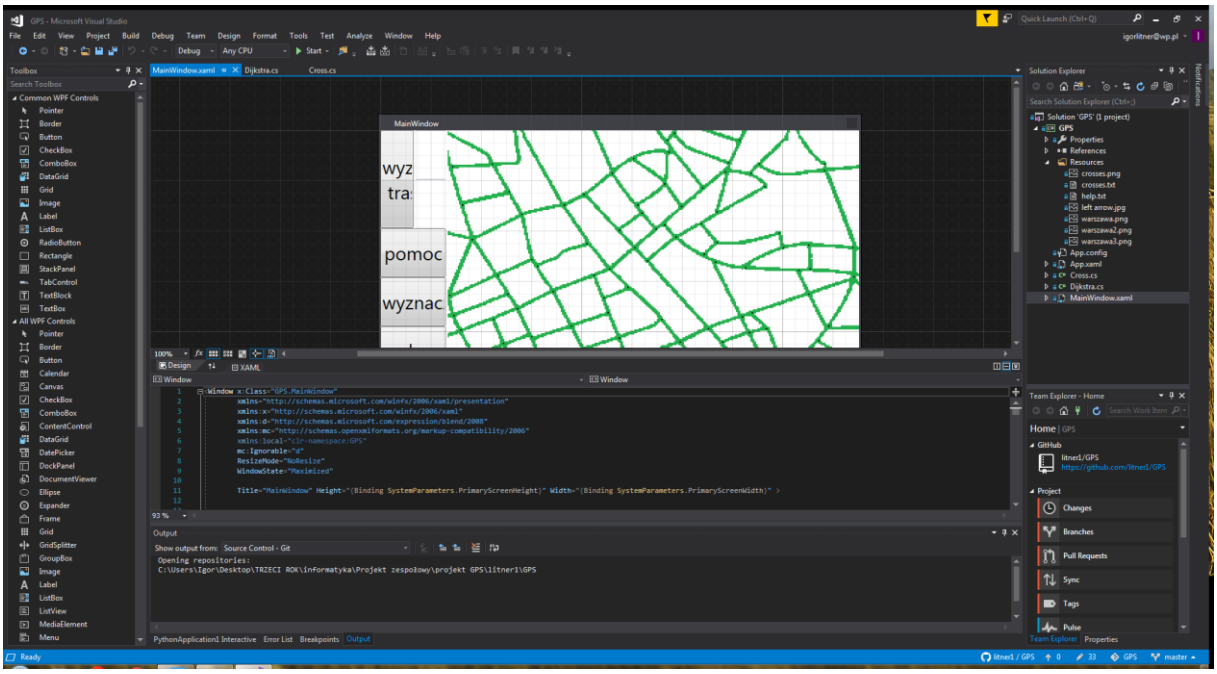
W katalogu **bin** znajdują się pliki wykonywalne, dzięki którym można uruchomić program. W katalogu **dokumentacja** znajduje się dokumentacja oraz wszystkie z nią niezbędne rzeczy związane. Katalog **obj** jest plikiem tworzonym przez Microsoft Visual Studio. Katalog **properties** zawiera wszystkie pliki związane z właściwościami projektu. Katalog **resources** zawiera pliki źródłowe, między innymi mapę Warszawy. Pozostałe pliki to pliki odpowiedzialne za klasy i za GUI.

4.3. Kompilacja.

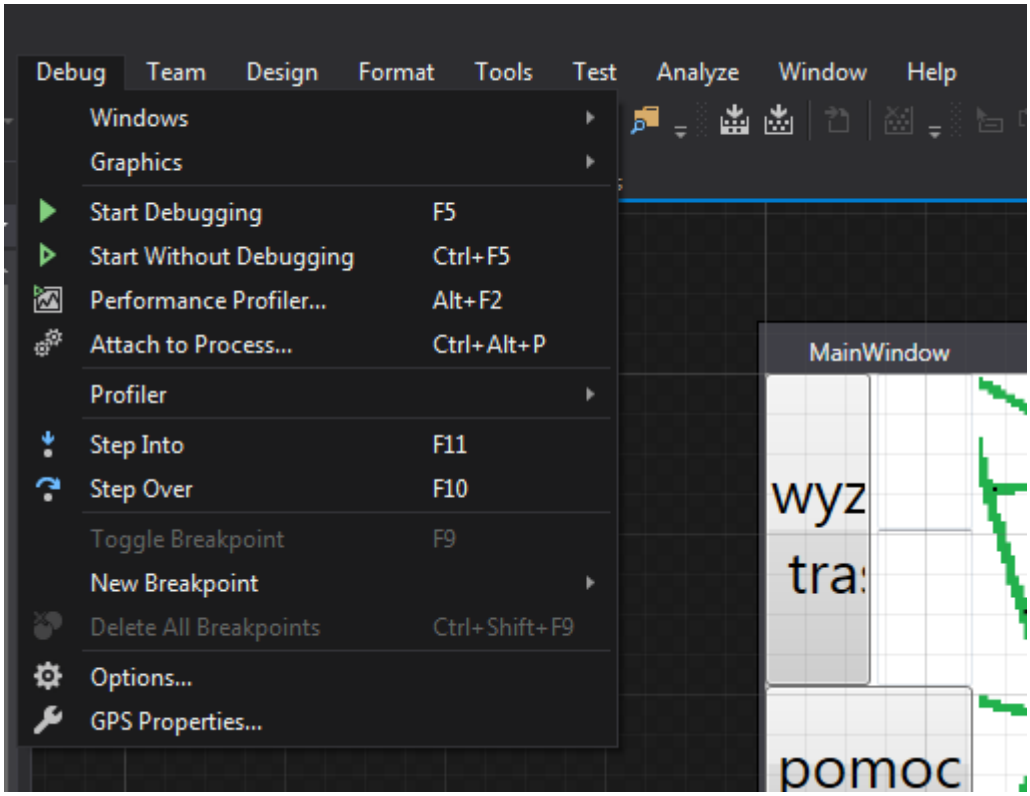
W celu skompilowania projektu należy uruchomić plik **GPS.sln**.



Pojawi się wtedy następujące okno.

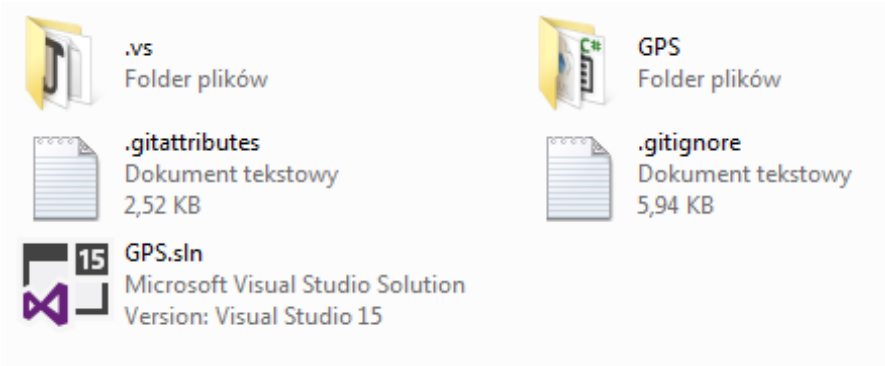


Należy teraz rozwinąć **Debug** i wybrać opcje **Start Debugging** lub **Start Without Debugging**.

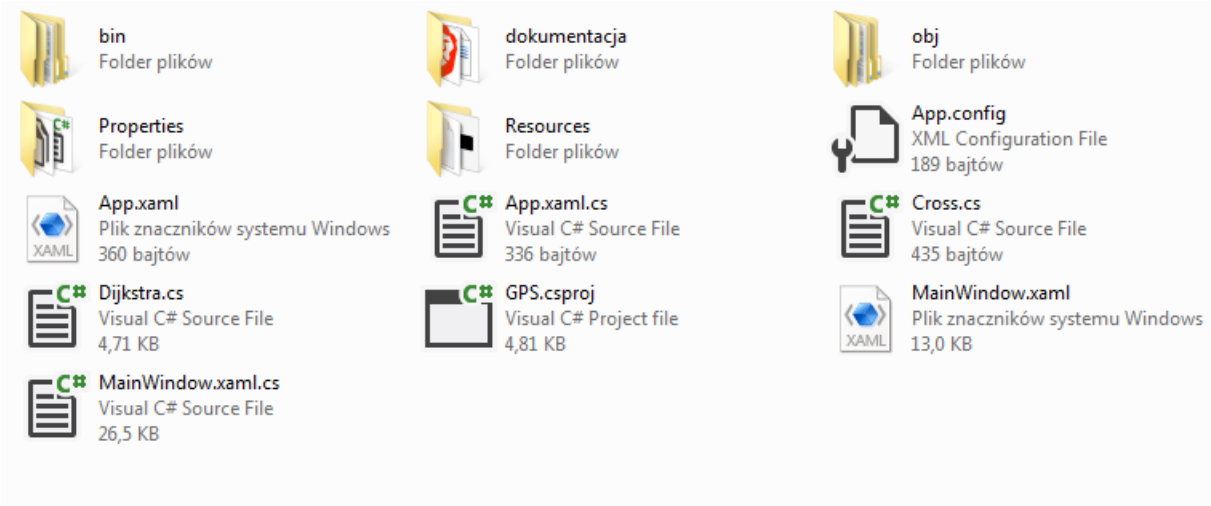


4.4. Uruchomienie programu.

W celu uruchomienia programu, należy wejść w katalog **GPS**.

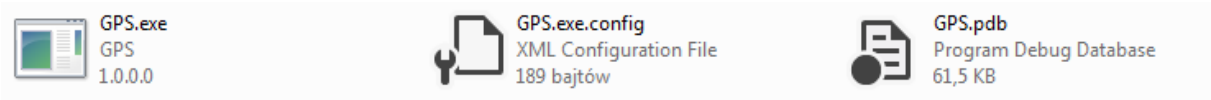


Należy teraz wejść w katalog **bin**.



Pojawia się wtedy następujące okno.

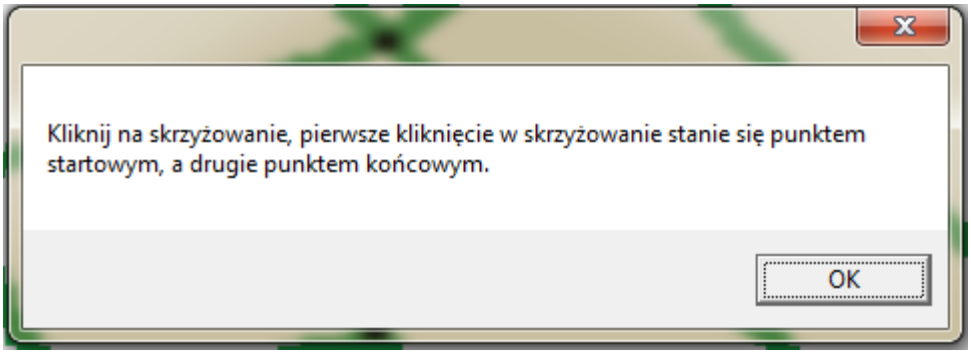
Katalogi **Debug** i **Release** mają identyczną strukturę. Wejdźmy w **Debug**.



Teraz należy otworzyć **GPS.exe**.

4.5. Przykład obsługi programu.

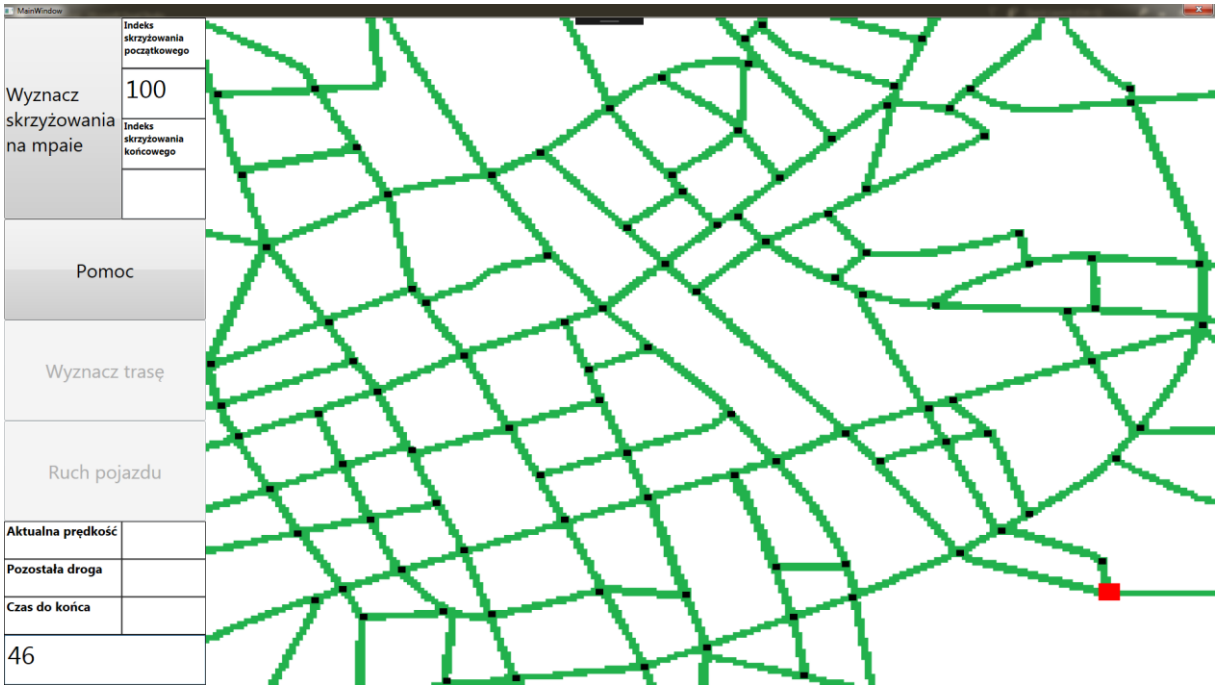
Po uruchomieniu programu pojawia się następujący komunikat.



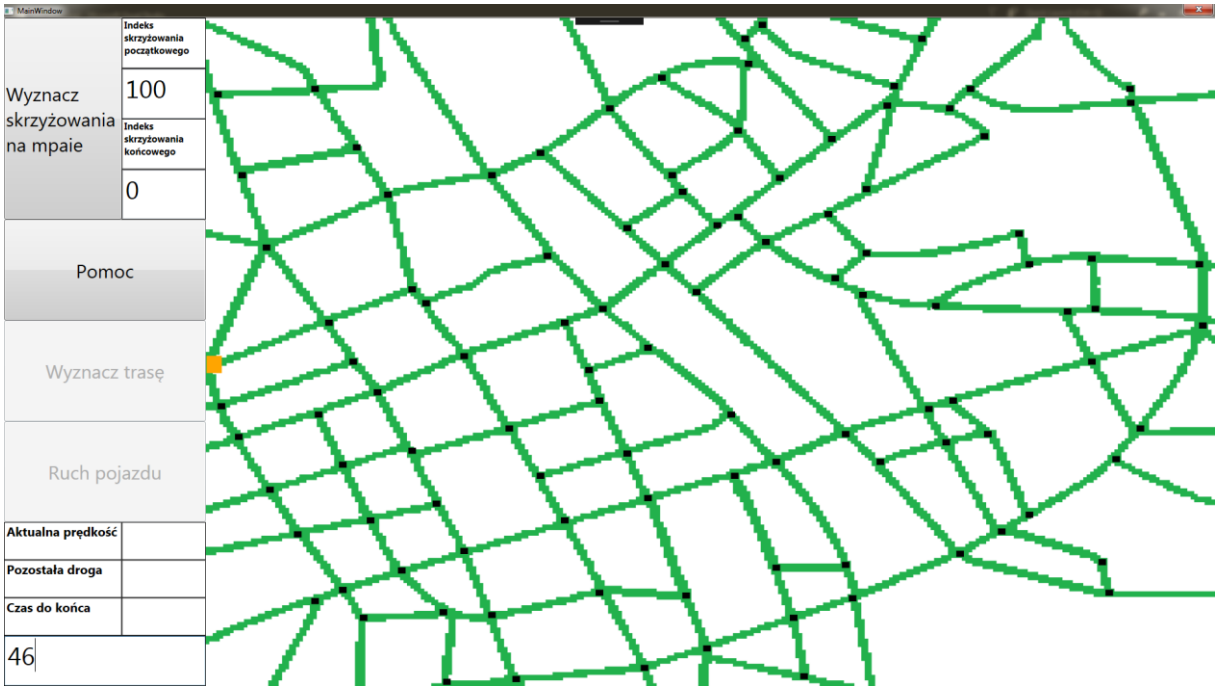
Po zamknięciu tego okienka, widzimy następujące okno.

Mapa składa się z zielonych dróg oraz skrzyżowań (małych czarnych prostokątów) między nimi. Na mapie jest 107 skrzyżowań numerowanych od 0 do 106. Wpiszmy w lewym dolnym rogu liczbę 46.

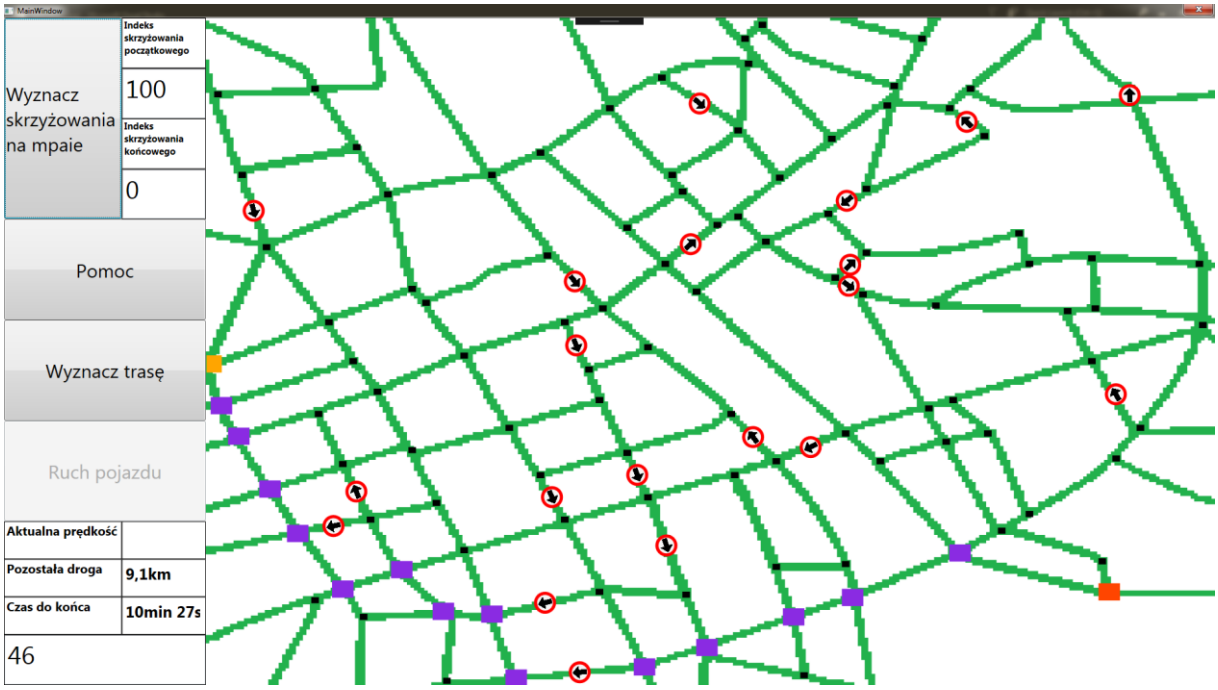
Po wpisaniu 46, na mapie widzimy jeden granatowy prostokąt oraz trzy czerwone. Granatowym kolorem jest oznaczone skrzyżowanie jest 46, a czerwone, to są jego sąsiedzi. Kliknięciem w skrzyżowanie podświetlamy je i jego sąsiadów na wyżej wymienione kolory. Wtedy w lewym dolnym rogu pojawia się numer skrzyżowania (poza pierwszymi dwoma kliknięciami). Działa to też w drugą stronę – można w lewym dolnym rogu napisać liczbę od 0 do 106 i wtedy się na mapie podświetli. Kliknijmy teraz na jedno ze skrzyżowań.



Na mapie widzimy czerwony prostokąt. Symbolizuje on skrzyżowanie 100. W lewym górnym rogu liczba 100 pojawiła się w miejsce **Indeksu skrzyżowania początkowego**. Kliknijmy na kolejne skrzyżowanie.



Na mapie widzimy teraz pomarańczowy prostokąt symbolizujący skrzyżowanie końcowe. W miejsce **Indeksu skrzyżowania końcowego** pojawiło się 0. Aktywnymi przyciskami są **Wyznacz skrzyżowania na mpaie** oraz **pomoc**. Na dole napisy prędkość i pozostała droga, będą wyświetlać wartości prędkości i odległości w trakcie jazdy.



Po wciśnięciu **Wyznacz skrzyżowania na mapie** program oblicza najlepszą trasę i odblokowuje się przycisk **wyznacz linię** oraz zaznacza na mapie drogi jednokierunkowe i skrzyżowania, po których

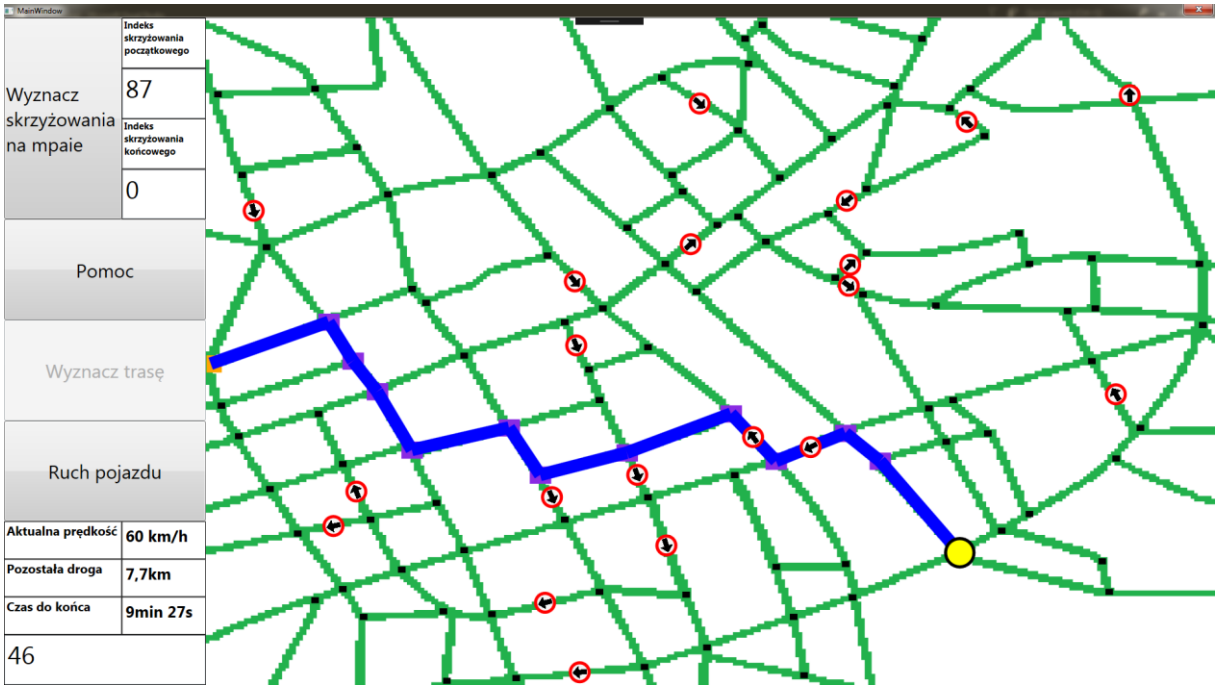
będzie się poruszał pojazd, gdzie prostokąt czerwony to punkt początkowy, prostokąt pomarańczowy to punkt końcowy, a prostokąty fioletowe, to skrzyżowania pośrednie. W lewym dolnym rogu widzimy, że **Pozostała droga** wynosi 9,1km, a **Czas do końca** wynosi 10 minut 27 sekund. Wciśnijmy teraz **Wyznacz trasę**.

Wyznacz trasę narysowało niebieską krzywą łamaną oraz pojawiło się żółte koło z aczrnym obwodem – samochód. Wciśnijmy teraz **ruch pojazdu**. Pojazd, porusza się. W lewym dolnym rogu widać jego prędkość tymczasową $60\frac{km}{h}$. Odległość zmniejsza się z krokiem 0,1km, a czas zmienia się o sekundę.

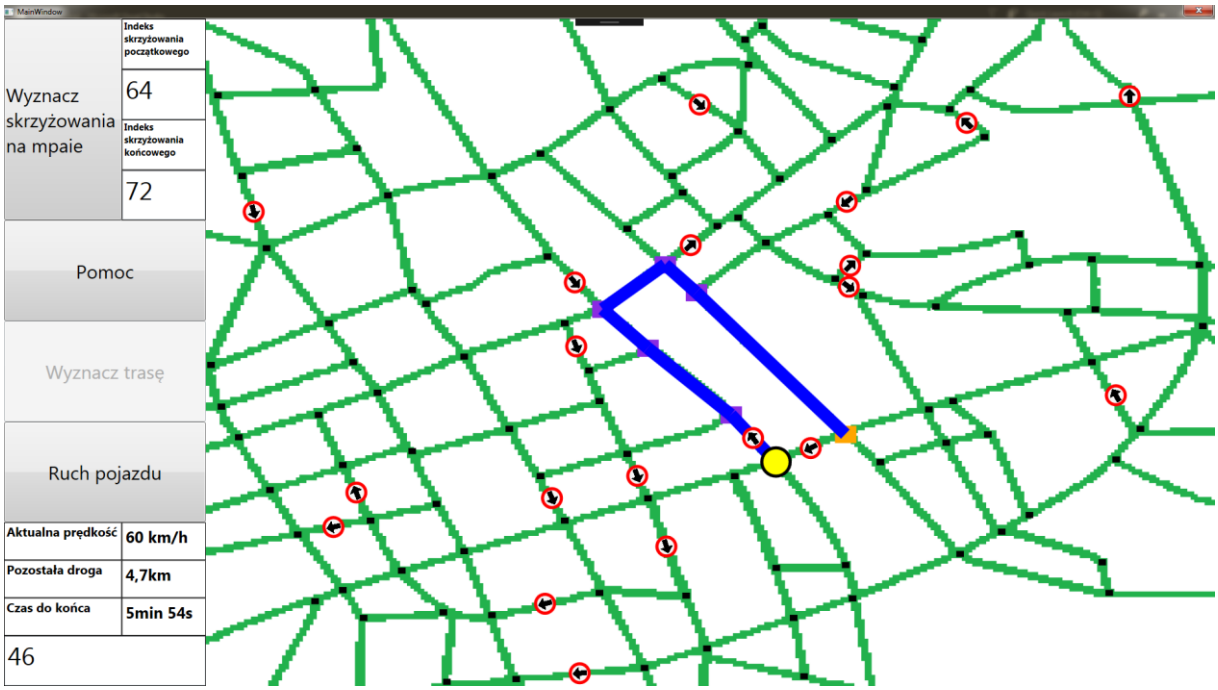
Po dojechaniu do skrzyżowania ponownie możemy kliknąć **Wyznacz skrzyżowania na mapie.** Z prawdopodobieństwem $\frac{1}{3}$ prędkości tymczasowe na drogach się zmienią, co może skutkować zmianą trasy. Żeby mieć pewność, że trasa się zmieni, wciśnijmy prawym przyciskiem myszy na jedną z niebieskich linii. W ten sposób w tym miejscu generujemy korek, czyli na danej drodze można jechać tylko $10\frac{km}{h}$.

Po kliknięciu na jedną z linii otrzymujemy korek.

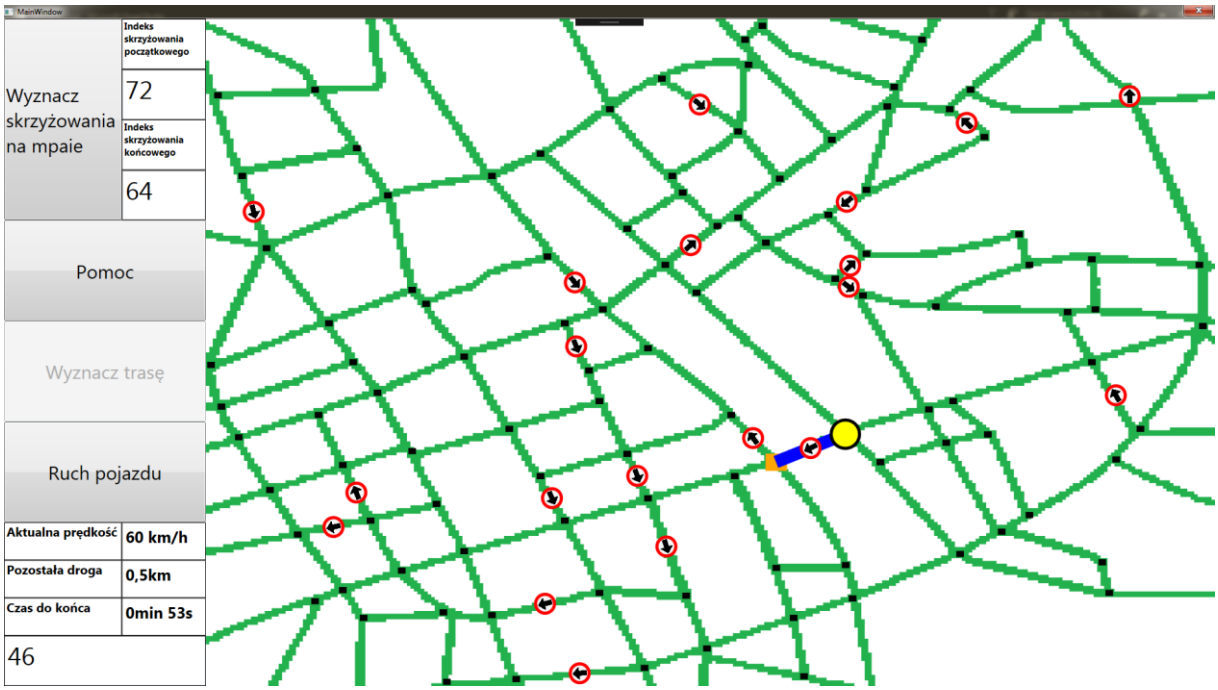
Teraz wciśnijmy **Wyznacz skrzyżowania na mapie** i **Wyznacz trasę.** Nowa trasa różni się od starej tak jak i **Pozostała droga** oraz **Czas do końca**.



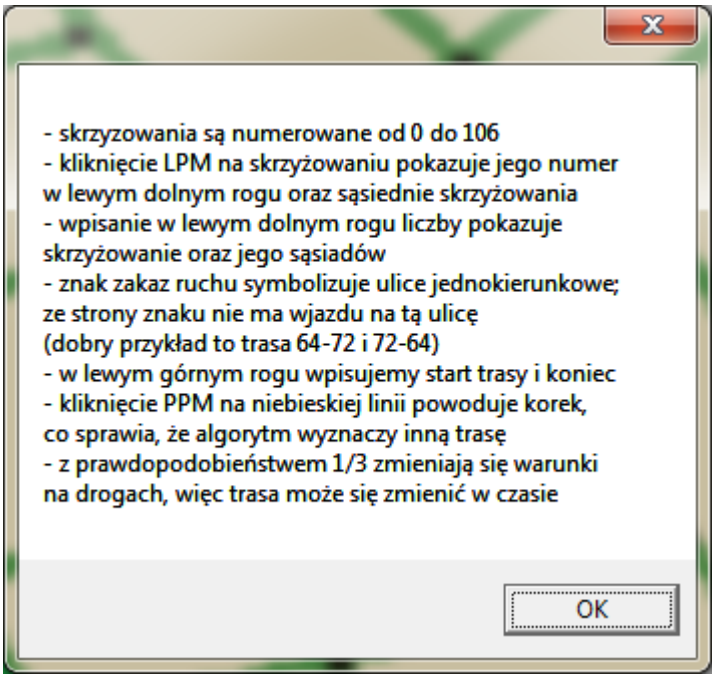
Pokażemy teraz, że drogi jednokierunkowe pozwalają poruszać się tylko w jednym kierunku. Wyznaczmy trasę dla punktów 64 i 72.



Teraz wyznaczmy trasę dla 72 i 64. Jak widać te trasy istotnie się różnią.



Wciśnięcie przycisku **Pomoc** otwiera następujące okno.



5. Bibliografia.

[1] J. Skeet, ‘’C# od podszewki’’, wydawnictwo Helion, Gliwice, wydanie IV, rok 2020

[2] https://visualstudio.microsoft.com/pl/ - Microsoft Visual Studio 2017

[3] https://docs.microsoft.com/pl-pl/visualstudio/get-started/csharp/tutorial-wpf?view=vs-2019 - Windows Presentation Foundation

[4] M. Sysło, N. Deo, J. Kowalik, ‘’Algorytmy Optymalizacji Dyskretnej”, wydawnictwo naukowe PWN, Warszawa, wydanie III, 1999 – Algorytm Dijkstry

Podział pracy.

1. Algorytm Djikstry – Igor Litner 50%, Bartosz Chreścionko 50%
2. Interfejs – Igor Litner 50 %, Paweł Jeleń 50%
3. Obecne położenie pojazdu – Igor Litner 100%
4. Implementacja korków – Igor Litner 70%, Bartosz Chreścionko 30%
5. Drogi jednokierunkowe – Igor Litner 90%, Bartosz Chreścionko 10%
6. Ruch pojazdu – Igor Litner 100%
7. Rysowanie bieżącej trasy – Igor Litner 80 %, Paweł Jeleń 20%
8. Zaznaczanie pozycji startowej oraz końcowej – Igor Litner 100%
9. Dokumentacja – Igor Litner 80 %, Paweł Jeleń 20%
10. Utworzenie repozytorium GitHub – Paweł Jeleń 100%

Oświadczenie.