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

Warszawa 2021

Spis treści

1. Wstęp.

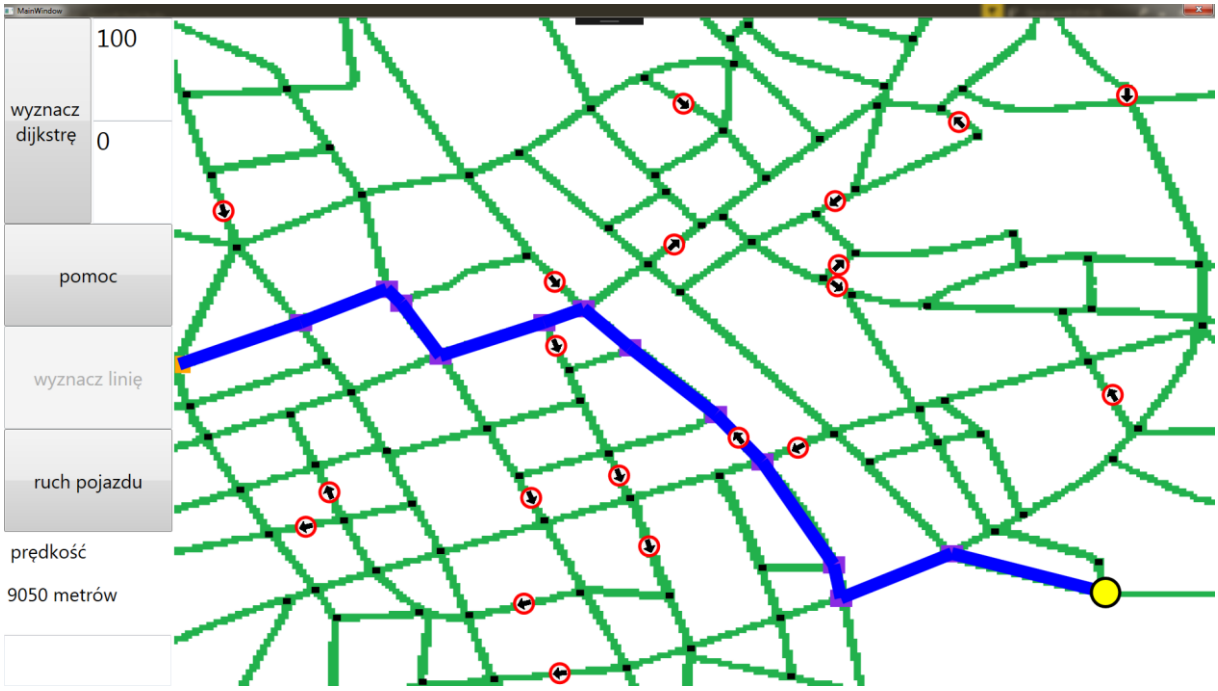
Zagadnienie przejazdu z punktu A do punktu B jest ważnym zagadnieniem optymalizacyjnym ze względów ekonomicznych. W zależności od potrzeb danej osoby lub firmy możemy rozpatrywać optymalizację względem najkrótszej drogi przejazdu, najszybszego czasu przejazdu, czy też najmniejszego spalania paliwa. Niniejsza praca opisuje zagadnienie najkrótszego czasu przejazdu w sieci miejskiej, czyli zmienne w czasie wagi łuków. W tym wypadku wagi łuków, to są aktualne prędkości dróg – czasami drogi są puste, a innym razem zakorkowane. Jest to oczywiście zmienne w czasie oraz względnie losowe. 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 drogę 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 ulice w Warszawie. Ma to zastosowanie praktyczne, ponieważ głównymi ulicami jeździ się szybciej i najczęściej są one wielopasmowe.



Na mapie jest 107 skrzyżowań – małe czarne prostokąty, które połączone są ze sobą zielonymi liniami – drogami. Mapa jest odwzorowana w postać grafu skierowanego, gdzie małe czarne prostokąty to skrzyżowania, które odpowiadają wierzchołkom w grafie, zielone linie to drogi, których odpowiednikami w grafie są krawędzie, a białe koło z czerwonym obwodem i czarną strzałką wskazuje ulicę jednokierunkową, która w grafie odzwierciedla krawędź skierowaną – można jechać tylko zgodnie z kierunkiem wskazanym przez znak.

Dzięki temu możemy stworzyć macierz sąsiedztwa. W tym wypadku będzie to macierz koincydencji, czyli macierz kwadratowa, która ma 1 w miejscu **ij**, jeśli skrzyżowanie **i** oraz **j** można połączyć jedną drogą i 0 w przeciwnym przypadku. Macierz ta jest niezbędna do wyznaczenia optymalnej ścieżki w grafie.

Program wykorzystuje macierz sąsiedztwa, punkt startowy i punkt końcowy do algorytmu Dijkstry z uwzględnieniem odległości (rzeczywistej w kilometrach) skrzyżowań od siebie, prędkości tymczasowych, które zależą od ograniczeń stałych i jednokierunkowości. Algorytm Dijkstry wyznacza optymalną ścieżkę w grafie, na podstawie której tworzona jest trasa na mapie, gdzie trasa jest złożona z dróg, którymi najszybciej przejedziemy z punktu startowego do końcowego.

W programie zostały użyte następujące założenia:

1. **Stałe ograniczenia 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.

2. **Prędkości tymczasowe 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 bierzemy MIN(wylosowana wartość, ograniczenie stałe). Istotne jest to, że ograniczenie stałe jest ważniejsze niż tymczasowe, czyli jeśli droga ma ograniczenie stałe $40\frac{km}{h}$ i zostanie wylosowane $60\frac{km}{h}$ to ta droga będzie miała prędkość $40\frac{km}{h}$. Po dojechaniu auta do skrzyżowania, z prawdopodobieństwem $\frac{1}{3}$ następuje losowanie, czy ograniczenie prędkości tymczasowe dróg się zmieniły. Jeśli prawdopodobieństwo byłoby większe, to wariancja byłaby zbyt duża i trasa przejazdu mogłaby się zmieniać praktycznie co każde skrzyżowanie. W rzeczywistości warunki na drogach zmieniają się w czasie, ale też nie za często.

3. **Jednokierunkowość**. Niektóre drogi (niezgodnie z rzeczywistością) są jednokierunkowe, co oznacza, że jeśli można przejechać z punktu A do punktu B, to nie można przejechać z B do A. W rzeczywistości mogą to być jakieś roboty drogowe, które wyłączą dany pas ruchu i w efekcie stworzy się droga jednokierunkowa.

4. **Korki**. 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.

1. Wybierz punkt startowy **s** i nadaj mu cechę stałą 0 oraz punkt końcowy **t**.
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**.
3. Nadaj pozostałym wierzchołkom, należącym do zbioru **G**, cechę tymczasową równą nieskończoność.
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 $\mathbf{w_i}$.
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.
6. Usuwamy wierzchołek $\mathbf{w_k}$ ze zbioru **G** i dodajemy do zbioru **S**.
7. Dla wierzchołka $\mathbf{w_k}$ liczymy cechy tymczasowe równe d($\mathbf{w_k}$, $\mathbf{w_j}$) jego sąsiadów, którzy należą do zbioru **G**.
8. Sprawdzamy, czy spełniony jest warunek d(**s**, $\mathbf{w_i}$) > d(**s**, $\mathbf{w_k}$) + d($\mathbf{w_k}$, $\mathbf{w_j}$).
8.1. Jeśli warunek jest spełniony to d(**s**, $\mathbf{w_j}$) = d(**s**, $\mathbf{w_k}$) + d($\mathbf{w_k}$, $\mathbf{w_j}$).
8.2. Jeśli warunek nie jest spełniony to d(**s**, $\mathbf{w_j}$) zostaje niezmienione.
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**.
10. Sprawdzamy warunek, czy **x** = t.
10.1. Jeżeli warunek jest prawdziwy, to kończymy algorytm.
10.2. Jeżeli warunek nie jest prawdziwy, to przechodzimy do kroku 5, tylko zamiast **s** bierzemy **x**.

2.3. Schemat krokowy wyznaczania trasy o najkrótszym czasie przejazdu.

1. Oznaczamy na mapie pewne drogi, które są jednokierunkowe.
2. Wprowadzamy losowo ograniczenia stałe na drogach.
3. Wprowadzamy odległości między skrzyżowaniami, które są obliczane w sposób przybliżony za pomocą twierdzenia Pitagorasa.
4. Wprowadzamy punkt początkowy i końcowy.
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.
5.1. 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 prędkości tymczasowe dla dróg między skrzyżowaniami z uwzględnieniem ograniczeń stałych i jednokierunkowości.
6. Wyznaczamy wagi krawędzi grafu, które są czasami przejazdu na odpowiadających im drogach.
7. Zauważmy, że mogą wylosować się różne prędkości tymczasowe, co oznacza, że często w dwóch różnych kierunkach ta sama krawędź będzie mieć inną wagę.
8. Za pomocą algorytmu Dijkstry wyznaczamy ścieżkę o najkrótszym czasie przejazdu na podstawie grafu.
9. Na podstawie wyznaczonej ścieżki w grafie, znajdujemy odpowiadającą jej trasę na mapie.

2.4. Schematy działania programu.

2.4.1. Schemat jednorazowy.

1. Uruchom plik wykonywalny.
2. Wprowadź mapę do programu razem z ulicami i skrzyżowaniami.
3. Wybierz punkt startowy i końcowy.
4. Zastosuj algorytm wyznaczania trasy na mapie od punktu startowego do końcowego.
5. Zaznacz trasę niebieską krzywą łamaną.

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, a nie startowego.

Samochód, w celu czysto wizualizacyjnym, będzie jechał po trasie 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 końcowego. Oznacza to, że po przejechaniu samochodem jednej drogi, mogą się zmienić tymczasowe ograniczenia prędkości na mapie, co w efekcie może zmienić trasę przejazdu.

1. Uruchom plik wykonywalny.
2. Wprowadź mapę do programu razem z ulicami i skrzyżowaniami.
3. Wybierz punkt początkowy i końcowy oraz skrzyżowania.
4. Zastosuj algorytm wyznaczania trasy na mapie od punktu początkowego do końcowego.
5. Zaznacz trasę niebieską krzywą łamaną.
6. (Opcjonalnie) dodaj korek na trasie. Jeśli korek zostanie dodany, wróć do punktu 4.
7. Wykonaj ruch samochodu do skrzyżowania.
8. Jeśli samochód nie znalazł się w punkcie końcowym, to punkt początkowy staje się punktem, w którym aktualnie znajduje się samochód, wróć do punktu 4.
9. Jeśli samochód znalazł się w punkcie końcowym, zakończ program.

3. Opis struktury programu.

3.1. Opis środowiska.

Program jest napisany obiektowo w języku C#. Środowiskiem programistycznym użytym do wykonania programu jest Microsoft Visual Studio 2017. Interfejs programu jest przedstawiony za pomocą WPF – Windows Presentation Foundation. Wersja frameworku wykorzystywana w programie to .NET Framework 4.5.

3.2. 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.

- **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 docelowego 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, zawiera całość interfejsu oraz metody pozwalające wyznaczyć drogę, narysować ją, jeździć po niej i robić korki.

3.3. 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,

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

- **help.txt** – plik tekstowy, który jest wyświetlany, po wciśnięciu przycisku pomoc,

- **Warszawa.png** – plik źródłowy uproszczonej mapy drogowej Warszawy,

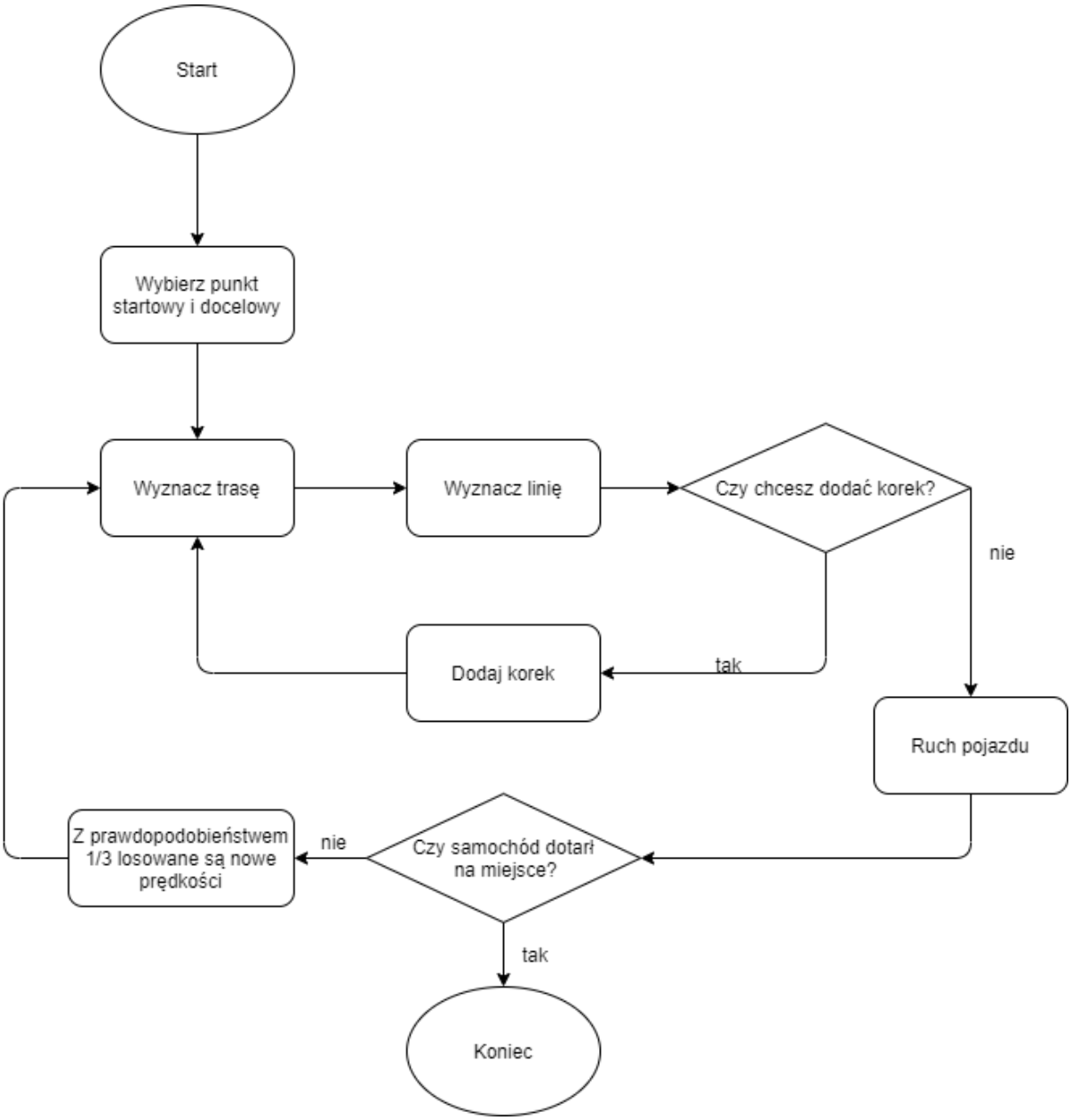
- **Warszawa2.png, Warszawa3.png** – pliki, które zostały wyrenderowane poprzez obróbkę pliku Warszawa.png

- **Crosses.png** – plik finalny mapy Warszawy, który jest używany w programie,

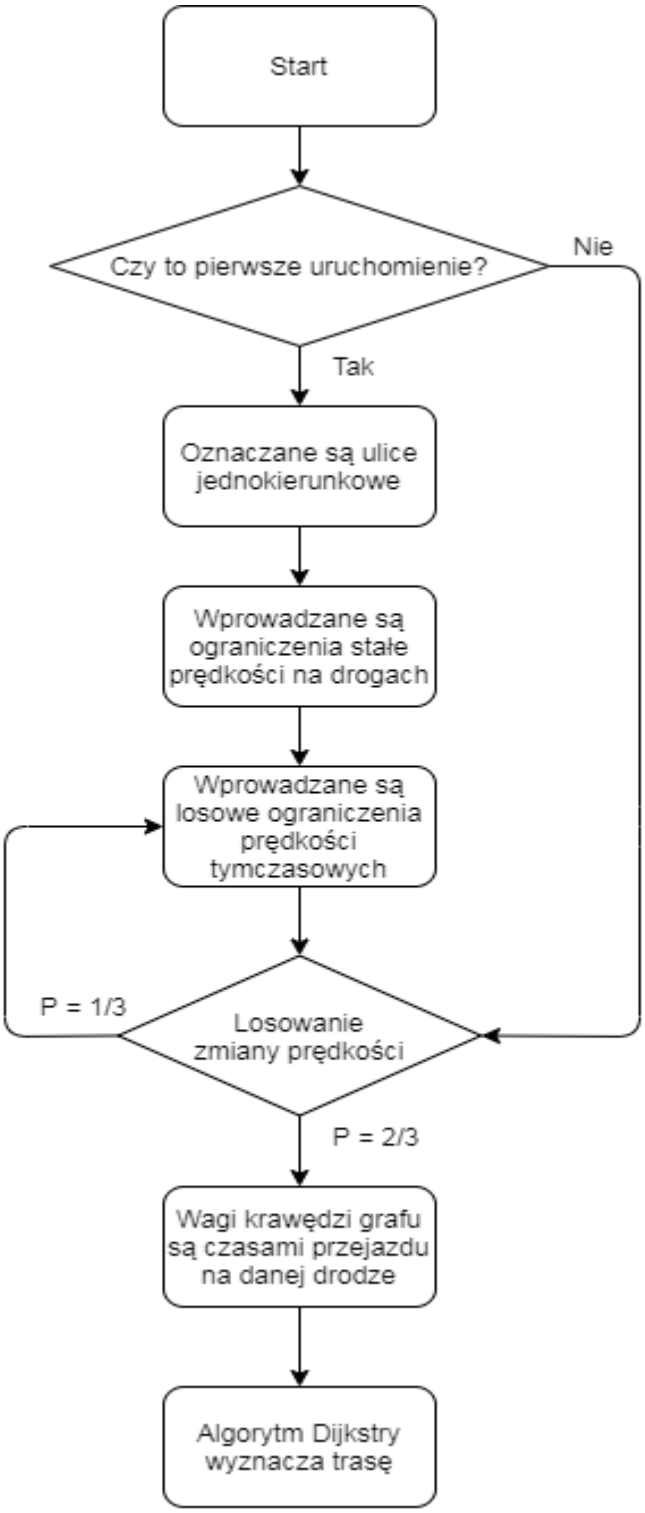
- **Crosses.txt** – plik, w którym zapisane są skrzyżowania z ich sąsiadami; w trakcie uruchamiania program pobiera dane z tego pliku.

- **left arrow.jpg** – plik, w którym jest zdjęcie strzałki, która jest używana, do zaznaczania ulic jednokierunkowych.

3.4. Schemat blokowy działania programu.



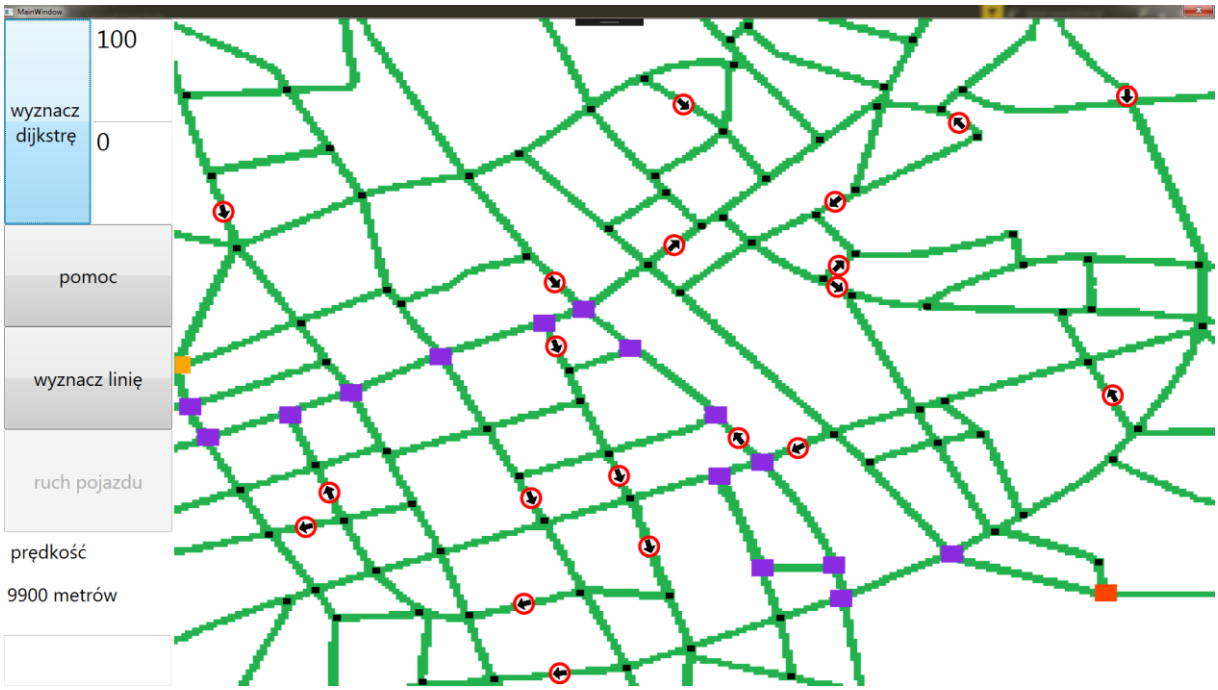
3.5. Schemat blokowy wyznaczania trasy.



3.6. Interfejs.

Jest to ekran startowy. Mapa składa się z zielonych dróg oraz skrzyżowań (małych czarnych prostokątów) między nimi. Na mapie jest 107 skrzyżowań numerowanych od 0 do 106. W lewym dolnym rogu widzimy liczbę 46. Z kolei na mapie widzimy jeden fioletowy prostokąt oraz trzy czerwone. Fioletowym 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. Działa to też w drugą stronę – można w lewym dolnym rogu napisać liczbę od 0 do 106, i wtedy się na mapie podświetli.

W lewym górnym rogu widzimy liczby 100 i 0 – 100 jest skrzyżowaniem startowym, 0 jest skrzyżowaniem końcowym. Aktywnymi przyciskami jest **wyznacz dijkstrę** oraz **pomoc**. Na dole napisy prędkość i pozostała droga, będą wyświetlać wartości prędkości i odległości w trakcie jazdy. Po wciśnięciu **wyznacz dijkstrę** program oblicza najlepszą trasę i odblokowuje się przycisk **wyznacz linię** oraz zaznacza na mapie ulice jednokierunkowe i trasę przejazdu, gdzie prostokąt czerwony to punkt startowy, prostokąt pomarańczowy to punkt końcowy, a prostokąty fioletowe, to skrzyżowania pośrednie. W lewym dolnym rogu pokazała się odległość wyrażona w metrach – 9900m.



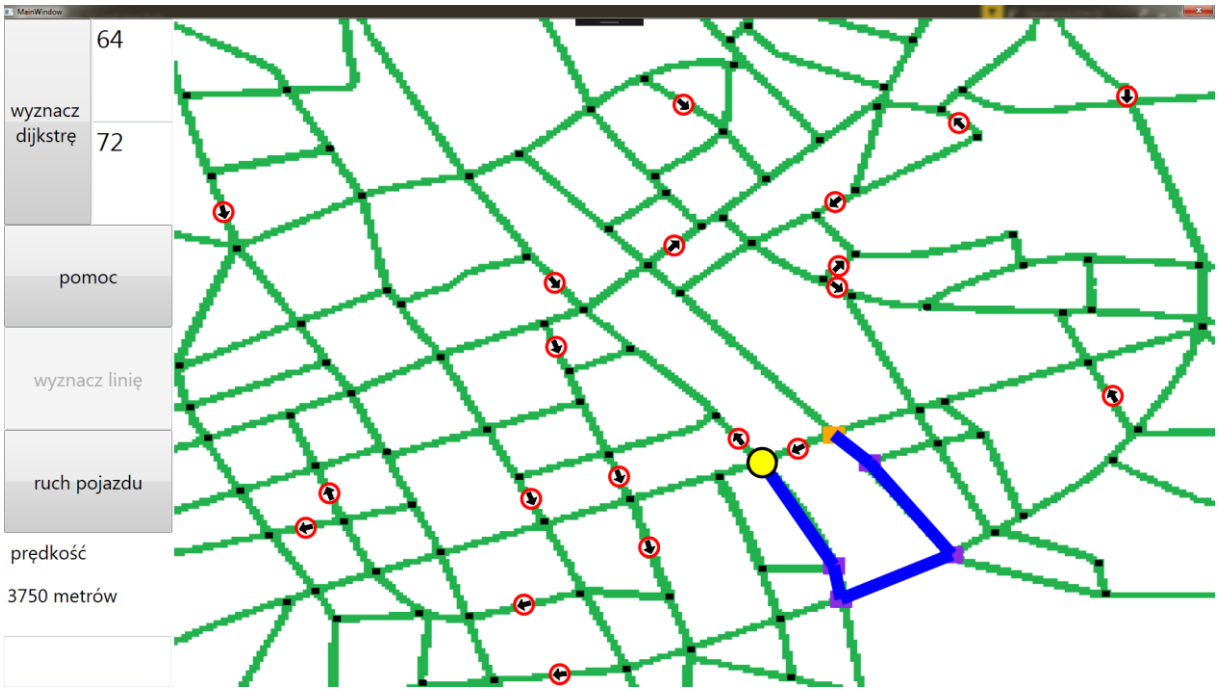
Po kliknięciu **wyznacz linię** go na mapie rysuje się krzywa łamana i przycisk staje się nieaktywny.

Wciśnijmy teraz **ruch pojazdu**. Pojazd – żółte koło z czarnym obwodem, porusza się. W lewym dolnym rogu widać jego prędkość tymczasową 50$\frac{km}{h}$. Odległość zmniejsza się z krokiem 50m.

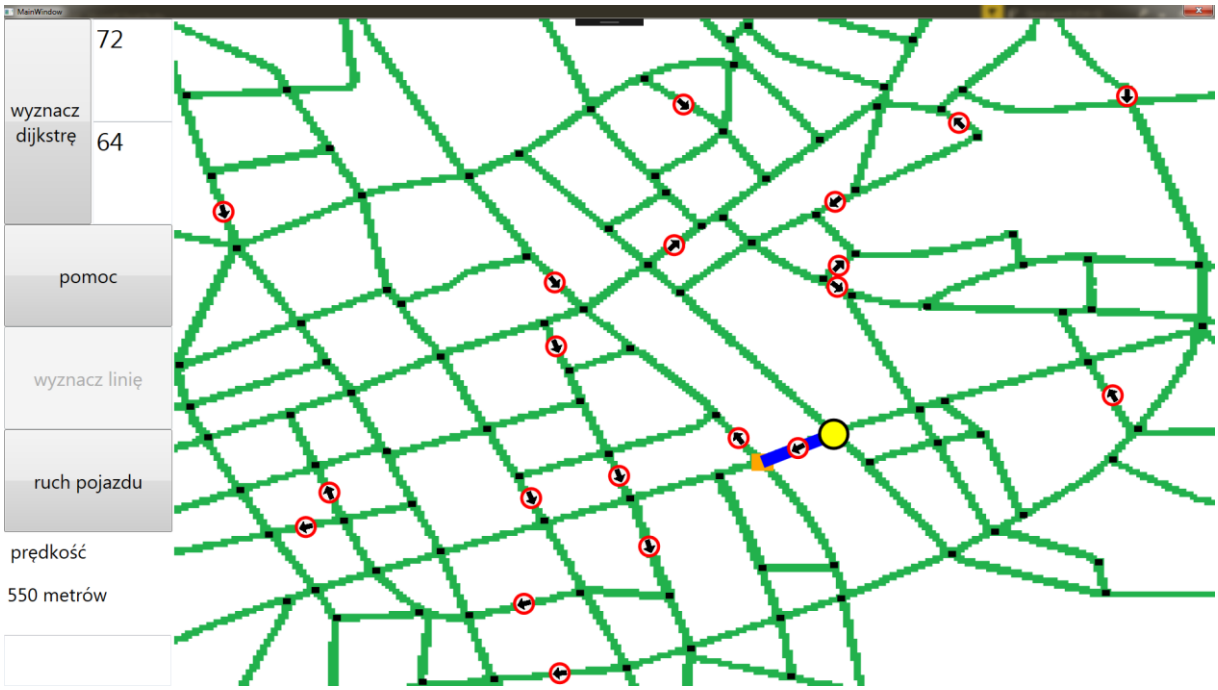
Po dojechaniu do skrzyżowania ponownie możemy kliknąć **wyznacz trasę.** Z prawdopodobieństwem $\frac{1}{3}$ prędkości tymczasowe na drogach się zmienią, co może skutkować zmianą trasy. Żeby mieć pewność, że trasa się zmieni, wciśnijmy prawym przyciskiem myszy na jedną z niebieskich linii. W ten sposób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

Teraz wciśnijmy **wyznacz dijkstrę.** Nowa trasa różni się od starej tak jak i odległość.

Pokażę teraz, że drogi jednokierunkowe rzeczywiście działają. Wyznaczmy trasę dla punktów 64 i 72.

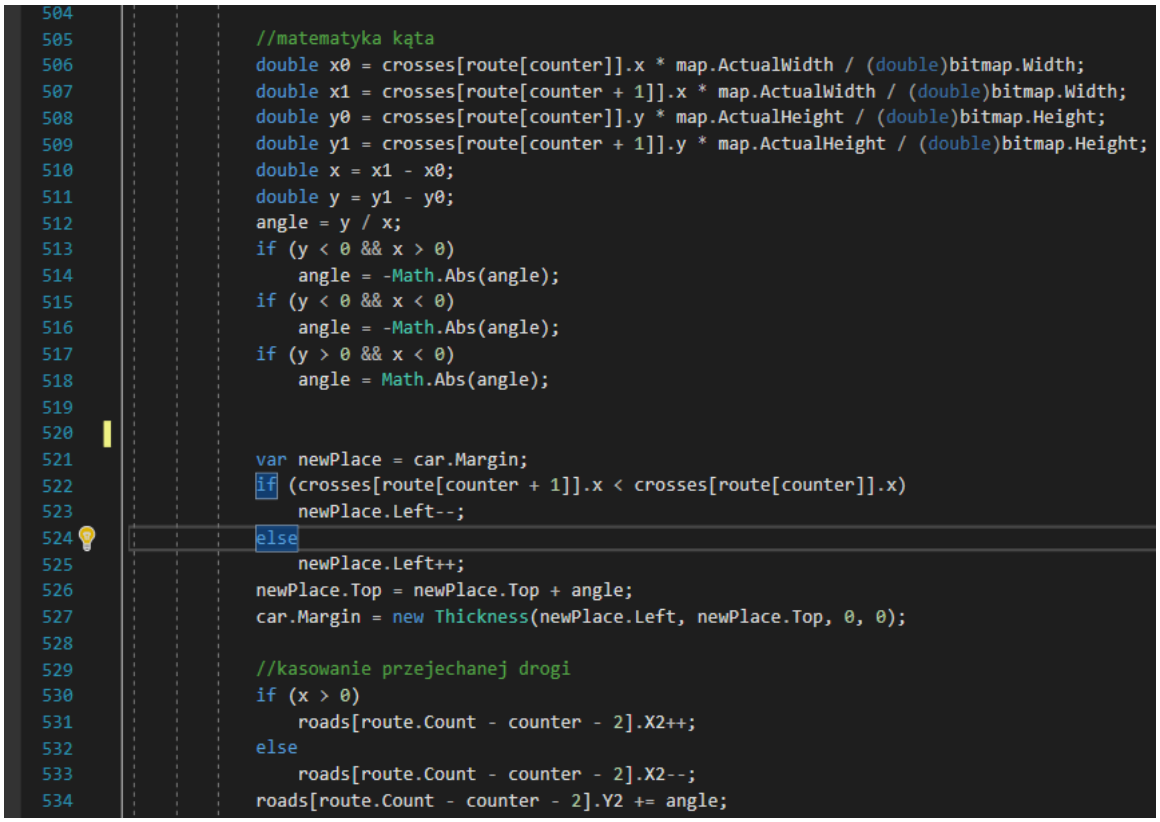


Teraz wyznaczmy trasę dla 72 i 64. Jak widać te trasy istotnie się różnią.


MainWindow
72
wyznacz dijkstrę
64
pomoc
wyznacz linię
ruch pojazdu
prędkość
550 metrów

3.7. Wybrane fragmenty kodu.

3.7.1. Ruch samochodu.



```
504
505             //matematyka kąta
506             double x0 = crosses[route[counter]].x * map.ActualWidth / (double)bitmap.Width;
507             double x1 = crosses[route[counter + 1]].x * map.ActualWidth / (double)bitmap.Width;
508             double y0 = crosses[route[counter]].y * map.ActualHeight / (double)bitmap.Height;
509             double y1 = crosses[route[counter + 1]].y * map.ActualHeight / (double)bitmap.Height;
510             double x = x1 - x0;
511             double y = y1 - y0;
512             angle = y / x;
513             if (y < 0 && x > 0)
514                 angle = -Math.Abs(angle);
515             if (y < 0 && x < 0)
516                 angle = -Math.Abs(angle);
517             if (y > 0 && x < 0)
518                 angle = Math.Abs(angle);
519
520
521             var newPlace = car.Margin;
522             if (crosses[route[counter + 1]].x < crosses[route[counter]].x)
523                 newPlace.Left--;
524             else
525                 newPlace.Left++;
526             newPlace.Top = newPlace.Top + angle;
527             car.Margin = new Thickness(newPlace.Left, newPlace.Top, 0, 0);
528
529             //kasowanie przejechanej drogi
530             if (x > 0)
531                 roads[route.Count - counter - 2].X2++;
532             else
533                 roads[route.Count - counter - 2].X2--;
534             roads[route.Count - counter - 2].Y2 += angle;
```

W linijkach 506-509 obliczamy współrzędne dwóch skrzyżowań trasy. Następnie obliczamy kąt pod jakim pojazd będzie się poruszał po mapie (góra/dół) oraz kierunek (lewo, prawo). W linijce 522 przesuwamy pojazd o jeden pixel w lewo/prawo, a w linijce 526 przesuwamy o 1 pixel pomnożony przez kąt. W linijce 527 samochód przesuwa się na wcześniej obliczone miejsce. Następnie kasowany jest fragment przejechanej drogi.

3.7.2. Losowanie prędkości i jednokierunkowości.

```
262             //losowanie prędkości dróg
263             var rnd = new Random(0);
264             var rnd2 = new Random();
265             if (rnd2.Next(3)==0 || crosses[0].velocity.Count == 0)
266             {
267                 for (int i = 0; i < crosses.Count; i++)
268                     crosses[i].velocity.Clear();
269                 for (int i = 0; i < crosses.Count; i++)
270                 {
271                     for (int j = 0; j < crosses[i].neighbours.Count; j++)
272                     {
273                         int x = rnd.Next(0, 14);
274                         if (x == 0)
275                             crosses[i].velocity.Add(0);
276                         else
277                         {
278                             int vel = rnd2.Next(3);
279                             crosses[i].velocity.Add(30 + 10 * vel);
280                         }
281                     }
282                 }
283             }
284
```

Jest to prosty kod, który na początku używa dwóch generatorów liczb losowych. Na początku mamy warunek, który sprawdza, czy nastąpiło zdarzenie losowania nowych prędkości dla dróg. Jeśli tak się zdarzyło, to usuwane są poprzednie prędkości dróg. W linijce 273 losujemy jednokierunkowość. Zauważmy, że to są liczby z generatora **rnd**, który ma stałe ziarno, więc drogi jednokierunkowe zawsze będą takie same. Następnie losowane są prędkości dla dróg pozostałych. Jeśli wylosowana prędkość byłaby większa niż ograniczenie, to metoda .Add to wychwyci i nada prędkość równą ograniczeniu stałemu.

3.7.3. Macierz incydencji.

```
307             var graph = new double[crosses.Count, crosses.Count];
308             for (int i = 0; i < crosses.Count; i++)
309             {
310                 for (int j = 0; j < crosses.Count; j++)
311                 {
312                     for (int k = 0; k < crosses[i].neighbours.Count; k++)
313                     {
314                         if (crosses[i].neighbours[k] == crosses[j].index)
315                         {
316                             graph[i, j] = crosses[i].distance[k] * 1 / crosses[i].velocity[k];
317                             break;
318                         }
319                         else
320                             graph[i, j] = 0;
321                     }
322                 }
323             }
324 
325             return graph;
326         }
```

W tej potrójnej pętli dla każdej pary skrzyżowań (i,j) ustalamy wagę w grafie równą rzeczywistej odległości podzielonej przez prędkość. Zauważmy, że jeżeli skrzyżowania nie są połączone, to wartość w grafie wynosi 0. Z kolei jeśli połączenie jest, ale jest to droga jednokierunkowa, to jej prędkość wynosi 0. Zatem 1 dzielone przez 0, to jest nieskończoność. Wtedy algorytm Dijkstry nie będzie takiej drogi brał pod uwagę.

4. Bibliografia.

[1] C# od podszewki. Wydanie IV, autor Jon Skeet

[2] https://visualstudio.microsoft.com/pl/

[3] https://docs.microsoft.com/

[4] https://pl.wikipedia.org/wiki/Algorytm_Dijkstry

5. Zawartość nośnika.

6. Podział pracy. Chreścionko, Litner.

7. Oświadczenie.