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 ulice 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 ulic 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 Warszawy. Ulice, które są na mapie to główne drogi w Warszawie. Ma to zastosowanie praktyczne, ponieważ głównymi drogami jeździ się szybciej i najczęściej są one niejednopasmowe. 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$\frac{km}{h}$} – 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. Losowanie prędkości danej drogi w danej chwili. One też są losowane ze zbioru {30$\frac{km}{h}$, 40$\frac{km}{h}$, 50$\frac{km}{h}$}, ale potem bierzemy MIN(wylosowana wartość, ograniczenie stałe). Istotne jest to, że ograniczenie stałe jest ważniejsze niż chwilowe, czyli jeśli droga ma ograniczenie stałe 40$\frac{km}{h}$ i zostanie wylosowane 50$\frac{km}{h}$ to ta droga będzie miała prędkość 40$\frac{km}{h}$. Po dojechaniu auta do skrzyżowania, z prawdopodobieństwem $\frac{1}{3}$ następuje losowanie, czy prędkości chwilowe dróg się zmieniły. Jeśli prawdopodobieństwo byłoby większa, 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. Niektóre ulice (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 zrobią ulicę jednokierunkową.

4. W celu testowania istnieje możliwość w sposób deterministyczny sprowadzić korek. Można zasymulować korek - 10$\frac{km}{h}$ na drodze wygenerowanej trasy, co w efekcie w większości przypadków wymusi na programie znalezienie innej drogi.

5. Na mapie jest 107 skrzyżowań numerowanych od 0 do 106. Program korzysta z algorytmu Dijkstry, z uwzględnieniem odległości skrzyżowań od siebie, prędkości chwilowych i jednokierunkowości.

2.2. Schemat krokowy działania programu.

1. Uruchom plik wykonywalny.
2. Wybierz punkt startowy i docelowy.
3. Wyznacz trasę.
4. (Opcjonalnie) Dodaj korki.
5. Wykonaj ruch samochodu do skrzyżowania.
6. Jeśli samochód nie znalazł się w punkcie docelowym, wróć do punktu 3.
7. Jeśli samochód znalazł się w punkcie docelowym, wróć do 2.

2.3. Schemat wyznaczania trasy.

1. Na początku programu pewne drogi stają się jednokierunkowe.
2. Potem są losowane ograniczenia stałe.
3. Odległości faktyczne obliczane są za pomocą twierdzenia Pitagorasa.
4. Drogi otrzymują losowe prędkości tymczasowe z uwzględnieniem ograniczeń stałych i jednokierunkowości.
5. Odległości „do algorytmu Dijkstry" obliczane są z wcześniej obliczonych prędkości podzielonych przez prędkość – im większa prędkość, tym „mniejsza" odległość.
6. Algorytm Dijkstry wyznacza trasę.
7. Można dodać korki.
8. Po przejechaniu do skrzyżowania następuje losowanie, gdzie z prawdopodobieństwem $\frac{1}{3}$ wracamy do punktu 4 oraz z prawdopodobieństwem $\frac{2}{3}$ wracamy do punktu 5.

2.4. Algorytm Dijkstry.

1. Wybierz punkt startowy S i nadaj mu wartość stałą 0.
2. Nadaj pozostałym wierzchołkom wartość tymczasową równą nieskończoność.
3. Wszyscy sąsiedzi W_x punktu S otrzymują wartość tymczasową równą d(S, W_x), gdzie d jest odległością z uwzględnieniem prędkości.
4. Następnie spośród wszystkich W_x wybieramy taki W_k, który ma najmniejszą wartość tymczasową i otrzymuje on wartość stałą.
5. Teraz powtarzamy czynność dla W_k, ale pomijamy sąsiadów o cechach stałych.
6. Po pewnej ilości iteracji punkt końcowy T będzie mieć cechę stałą, co kończy algorytm.

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,

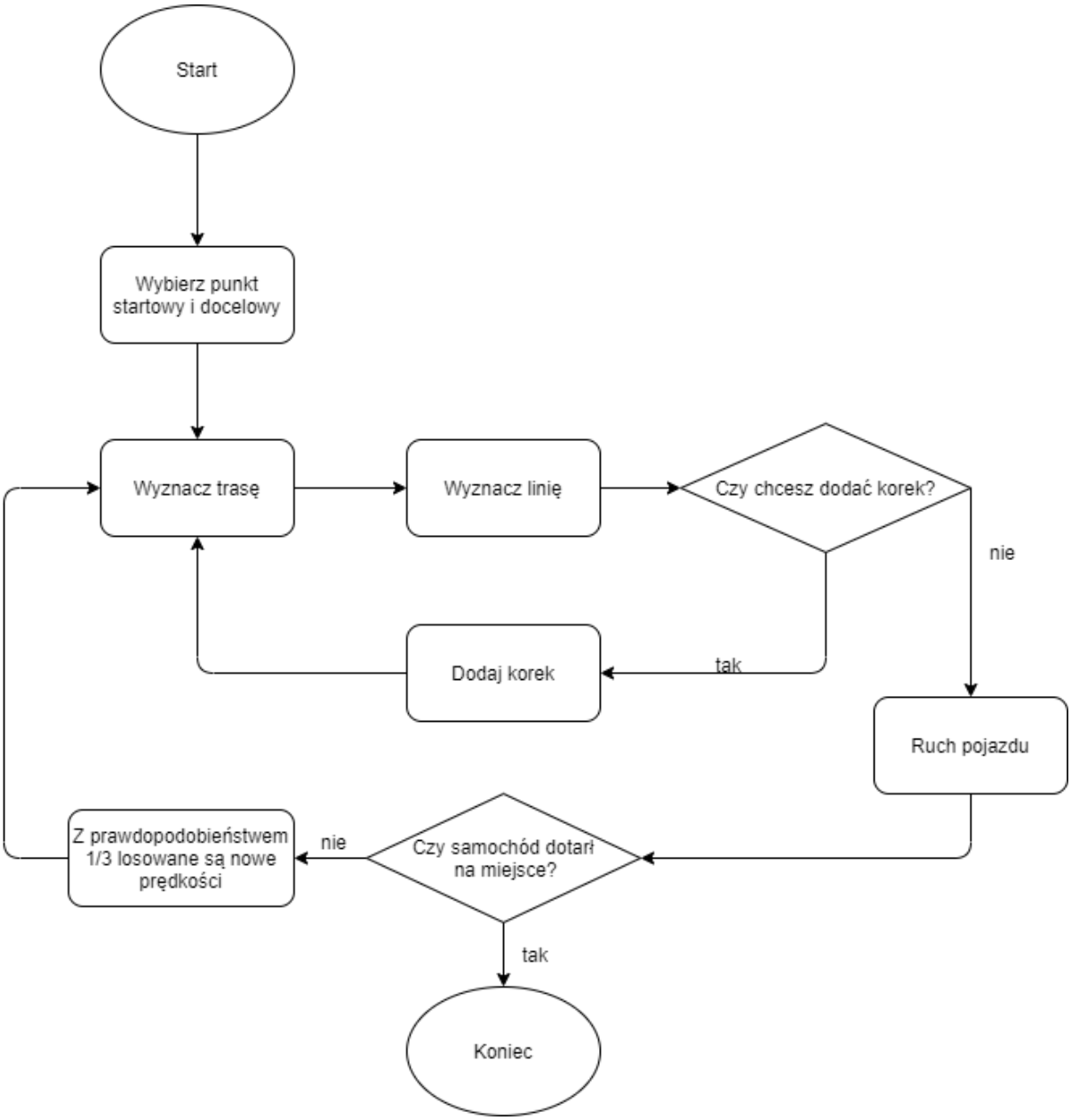
- **Warszawa.png** – plik źródłowy uproszczonej mapy drogowej Warszawy,

- **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.

3.4. Schemat blokowy działania programu.



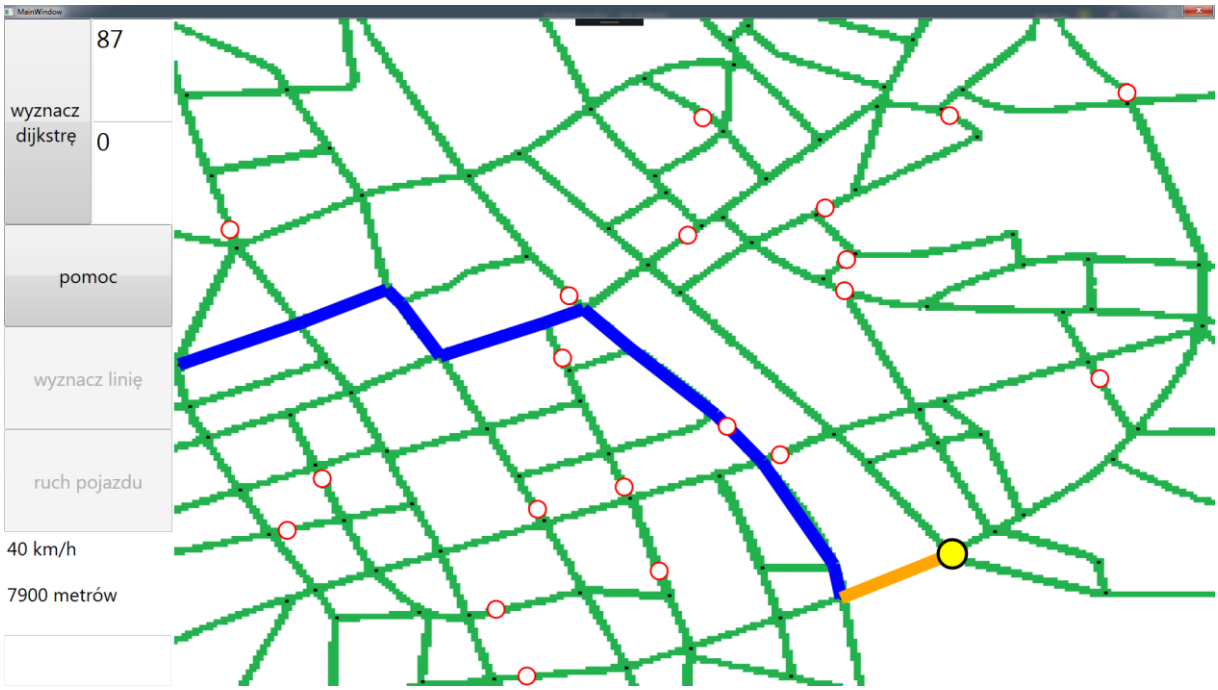
3.5. Interfejs.

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**. Po kliknięciu go na mapie rysuje się krzywa łamana, przycisk staje się nieaktywny, a w polu, gdzie była pozostała droga, pojawia się odległość wyrażona w metrach z krokiem 50m.

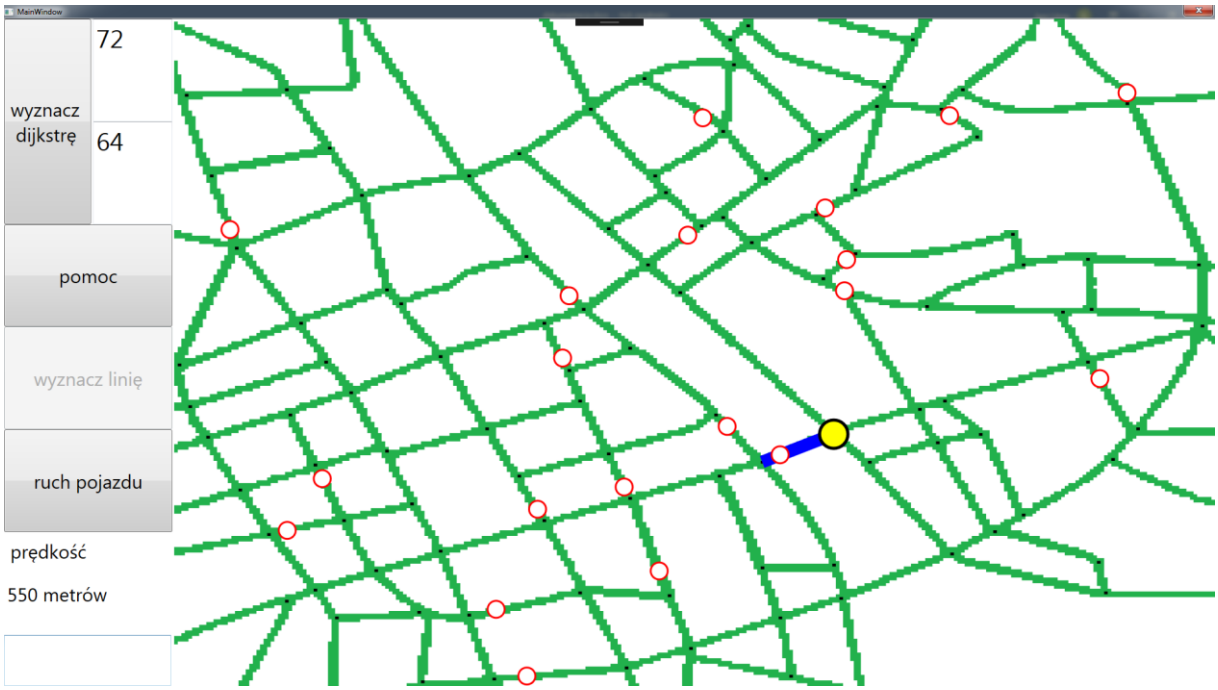
Zauważmy, że na mapie pojawiły się białe kółka z czerwoną obwódką – znak B1 w ruchu drogowym. Ten znak będzie oznaczał ulice jednokierunkowe – od strony bliższego skrzyżowania do tego znaku, nie będzie można w danym kierunku jechać. Teraz pozostaje nam przycisk **ruch pojazdu.**

Pojazd porusza się z prędkością $40\frac{km}{h}$, a droga do końca zmniejsza się.



Po dojechaniu możemy kliknąć na następny odcinek drogi – staje się on pomarańczowy. Oznacza to, że tam teraz jest korek, czyli prędkość na tym odcinku wynosi $10\frac{km}{h}$. Przyciski **wyznacz linię** i **ruch pojazdu** są wyłączone, więc pozostaje nam przycisk **wyznacz dijkstrę**. Trasa zostanie wyznaczona ponownie z uwzględnieniem korków. Z prawdopodobieństwem $\frac{1}{3}$ zmienią się prędkości chwilowe na drogach. Niemniej jednak ograniczenia stałe, korki i ulice jednokierunkowe mają większe znaczenie w wyznaczaniu trasy za pomocą algorytmu Dijkstry.

Nowa trasa różni się od starej tak jak i odległość. W każdym momencie w lewym górnym rogu można zmienić punkt startowy i końcowy, co przeniesie samochód na nowy punkt startowy. Na koniec jeszcze pokażę, że jednokierunkowość faktycznie działa na przykładzie skrzyżowań 64 i 72.


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


MainWindow
wyznacz dijkstrę
64
72
pomoc
wyznacz linię
ruch pojazdu
prędkość
3750 metrów

3.6. Wybrane fragmenty kodu.

3.6.1. 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.

3.6.2. 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 ulice 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.

3.6.3. 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.