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.

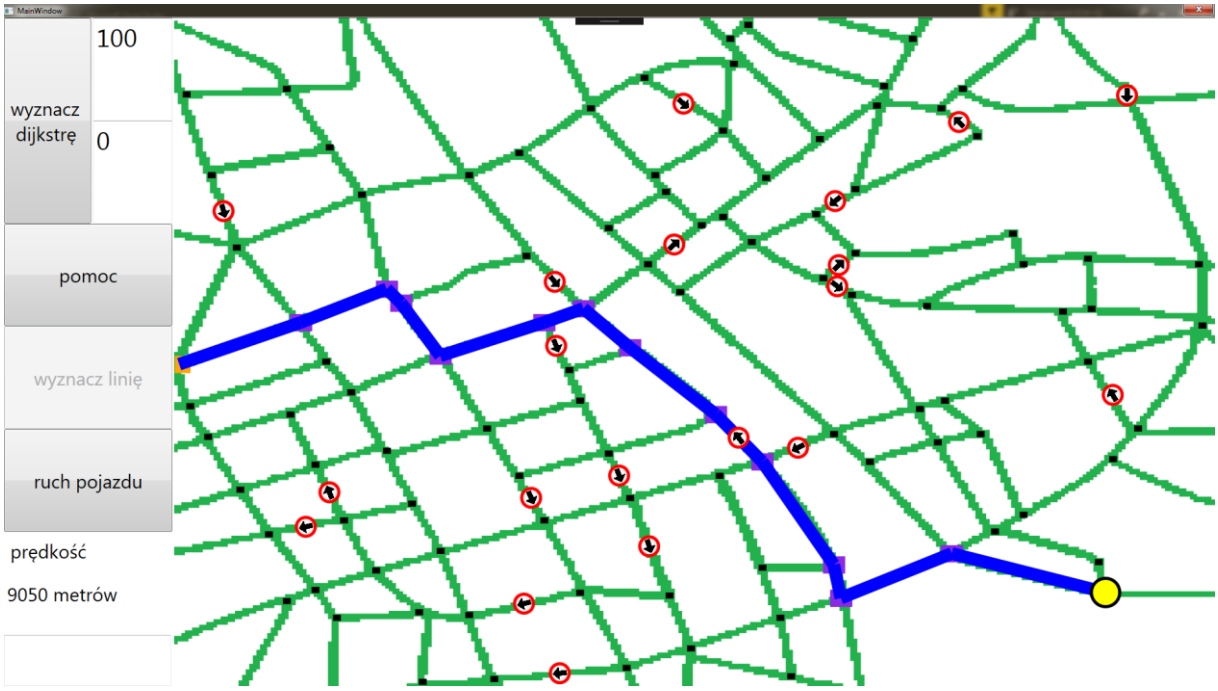
Zagadnienie przejazdu z punktu A do punktu B jest ważnym zagadnieniem optymalizacyjnym ze względów ekonomicznych. W zależności od potrzeb danej osoby lub firmy możemy rozpatrywać optymalizację względem najkrótszej drogi przejazdu, najszybszego czasu przejazdu, czy też najmniejszego spalania paliwa. Niniejsza 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 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 ulice 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.

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.

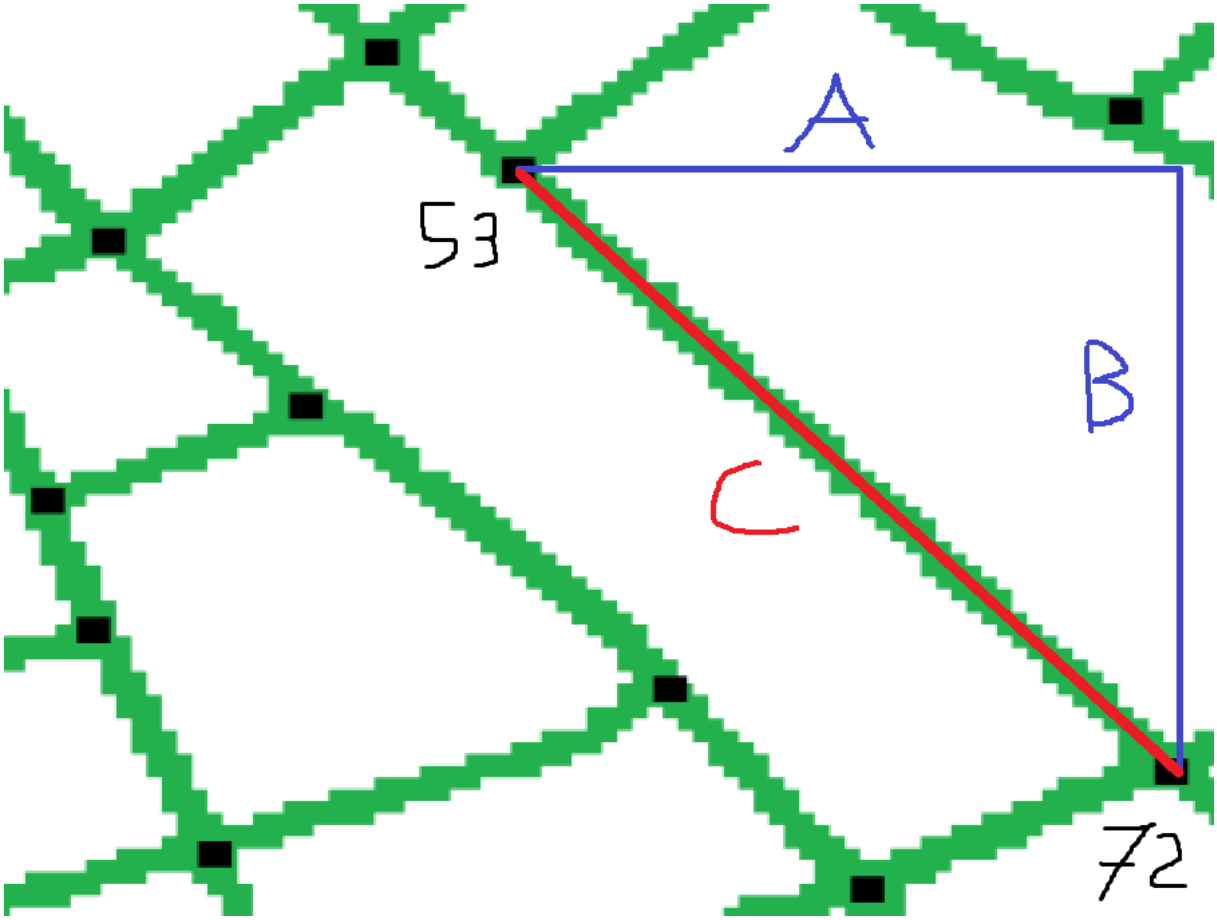
2.2. Schemat krokowy algorytmu Dijkstry.

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 $d(\mathbf{s}, \mathbf{w_i})$, 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 $d(\mathbf{s}, \mathbf{w_k})$ 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 $d(\mathbf{s}, \mathbf{w_i}) > d(\mathbf{s}, \mathbf{w_k}) + d(\mathbf{w_k}, \mathbf{w_j})$.
8.1. Jeśli warunek jest spełniony to $d(\mathbf{s}, \mathbf{w_j}) = d(\mathbf{s}, \mathbf{w_k}) + d(\mathbf{w_k}, \mathbf{w_j})$.
8.2. Jeśli warunek nie jest spełniony to $d(\mathbf{s}, \mathbf{w_j})$ 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.

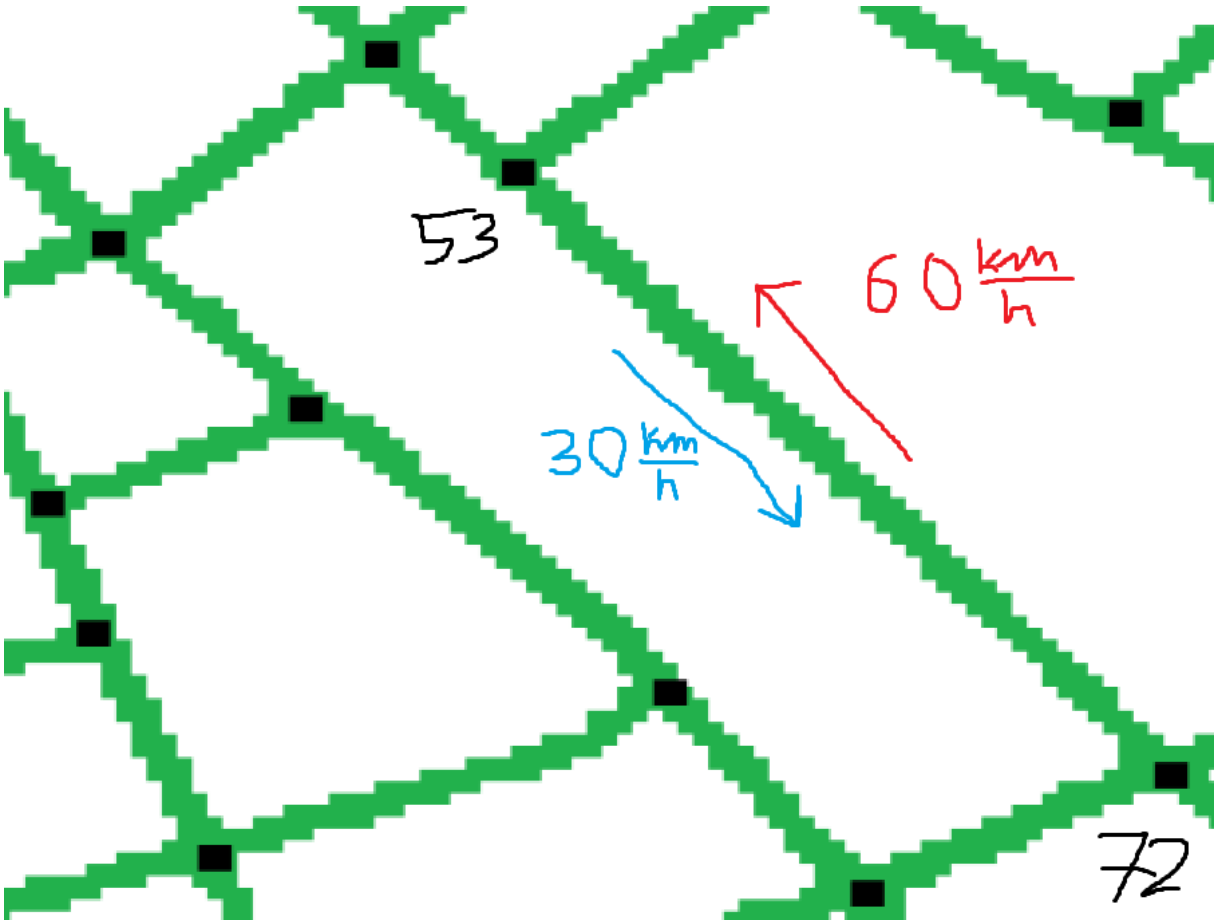
1. Wprowadzamy punkt początkowy i końcowy.
2. Wprowadzamy odległości między skrzyżowaniami, które są obliczane w sposób przybliżony za pomocą twierdzenia Pitagorasa.
3. Oznaczamy na mapie pewne drogi, które są jednokierunkowe.
4. Wprowadzamy losowo stałe ograniczenia prędkości na drogach.
5. Jeśli drogi nie mają jeszcze nadanych ograniczeń prędkości tymczasowych, wprowadzamy w sposób losowy tymczasowe ograniczenia prędkości dla dróg między skrzyżowaniami z uwzględnieniem ograniczeń stałych i jednokierunkowości.
5.5. 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.
6. Wyznaczamy wagi krawędzi grafu, które są czasami przejazdu na odpowiadających im drogach.
7. Zauważmy, że program może wylosować różne ograniczenia prędkości tymczasowej, 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.

Poniższy przykład ilustruje punkt **2.**



Na powyższym obrazie widzimy wycinek mapy programu. Z twierdzenia Pitagorasa wiemy, że długość $C = \sqrt{A^2 + B^2}$, gdzie A to ilość pikseli w poziomie, a B w pionie. Żeby otrzymać długość drogi w kilometrach od 53 do 72, musimy obliczone C pomnożyć przez skalę mapy. Jest to przybliżenie, ale na potrzeby działania programu jest wystarczająco dobre.

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



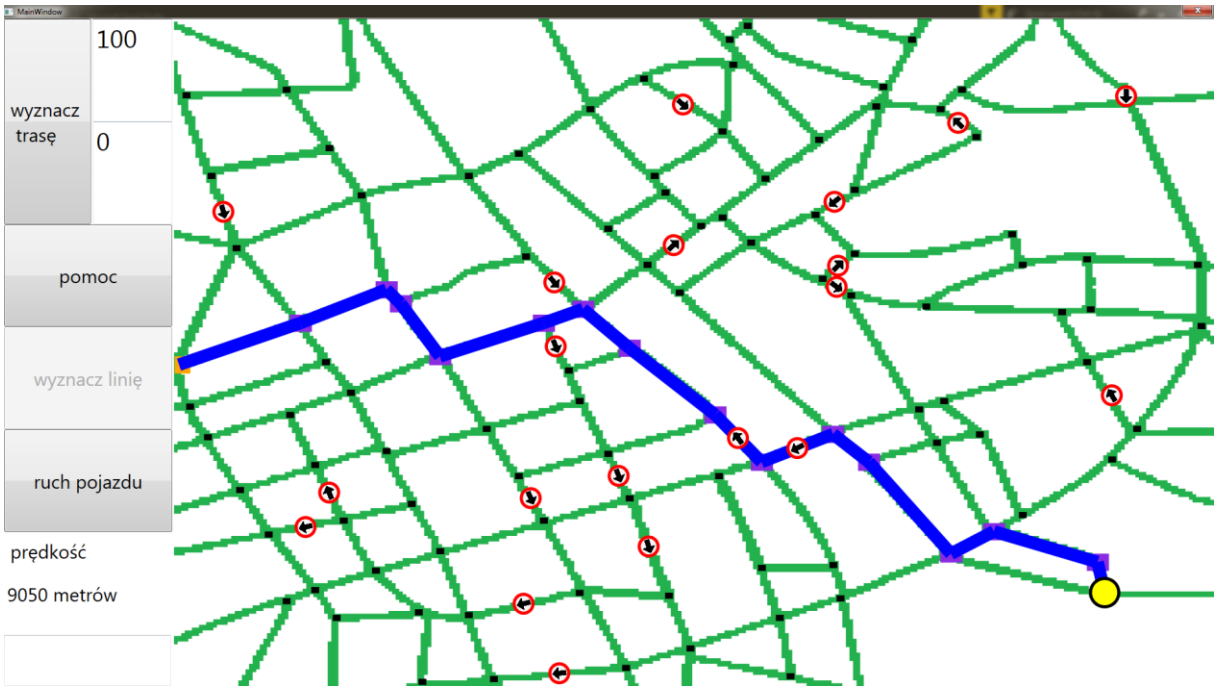
W celu uproszczenia obliczeń przyjmijmy, że odległość 53 od 72 wynosi 1km. Z fizyki wiemy, że czas jest to droga podzielona przez prędkość. W powyższym przypadku prędkość z 53 do 72 wynosi $30\frac{km}{h}$, a z 72 do 53 wynosi $60\frac{km}{h}$. Po podzieleniu drogi przez czas otrzymujemy kolejno dwie minuty i jedną minutę. Wtedy w grafie waga krawędzi z wierzchołka 53 do 72 wynosi 2min, a z 72 do 53 wynosi 1min.

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. Wprowadź ograniczenia długości dróg, ograniczeń prędkości stałych i tymczasowych oraz jednokierunkowości.
5. Zastosuj algorytm wyznaczania trasy na mapie od punktu startowego do końcowego.
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.

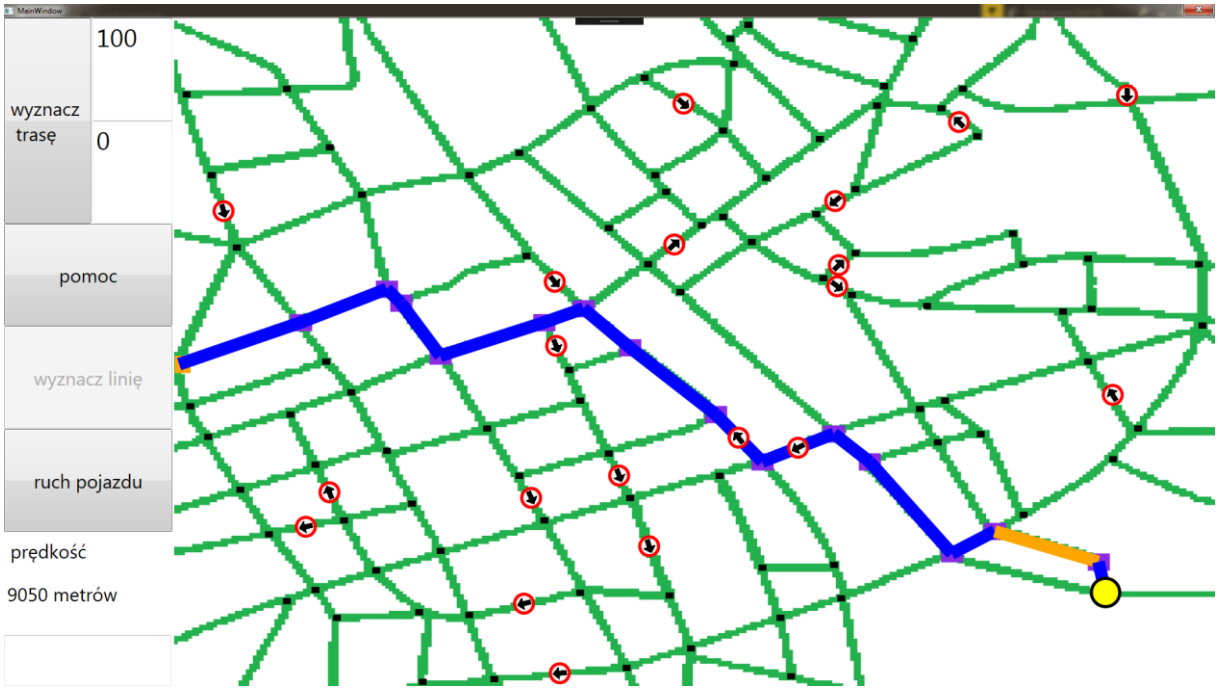
2.4.2. Schemat iteracyjny.

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.

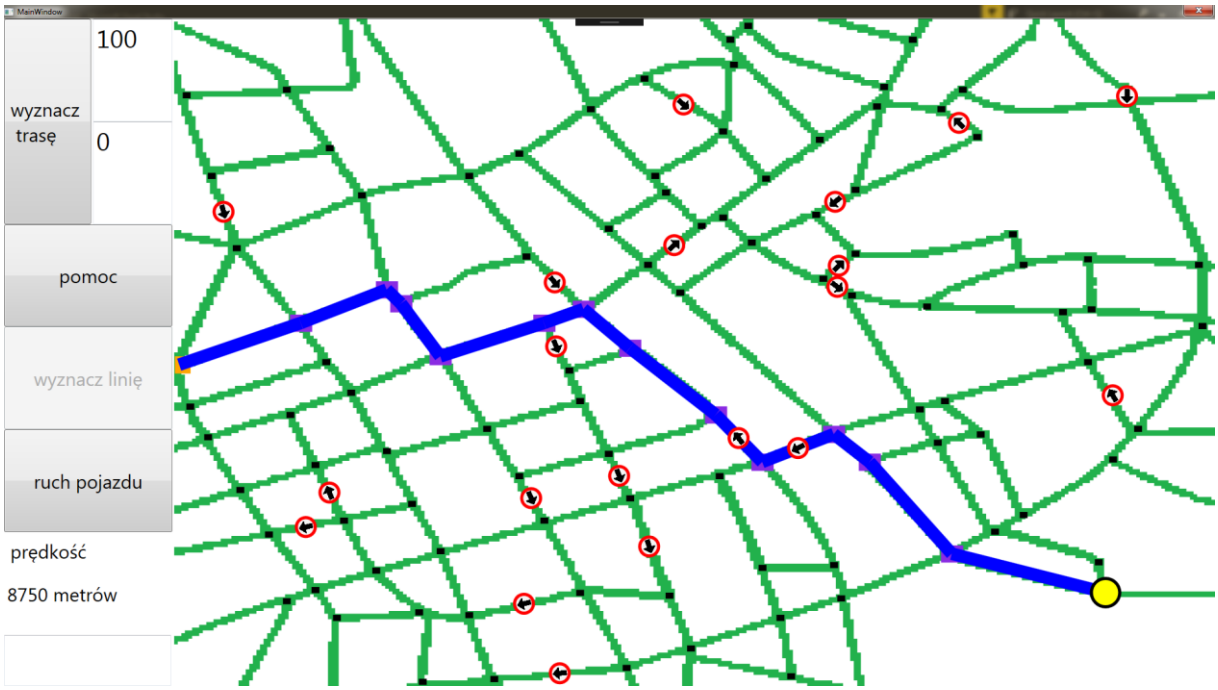
Samochód, w celu czysto wizualizacyjnym, będzie jechał zmieniającą się trasą od skrzyżowania do skrzyżowania. Punkt początkowy od punktu startowego różni się tym, że za każdym razem trasa jest wyznaczana od punktu, w którym aktualnie znajduje się samochód do końcowego, a nie od punktu startowego do punktu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

1. Uruchom plik wykonywalny.
2. Wprowadź mapę do programu razem z ulicami i skrzyżowaniami.
3. Wybierz punkt początkowy i końcowy.
4. Wprowadź ograniczenia długości dróg, ograniczeń prędkości stałych i tymczasowych oraz jednokierunkowości.
5. Zastosuj algorytm wyznaczania trasy na mapie od punktu początkowego do końcowego.
6. Zaznacz trasę niebieską krzywą łamaną.
7. Jeśli nie chcesz dodać korka, przejdź do punktu 8.
7.5. Jeśli chcesz dodać korek, kliknij na wybrany fragment trasy; następnie wróć do punktu 5.
8. Wykonaj ruch samochodu do skrzyżowania.
9. Jeśli samochód nie znalazł się w punkcie końcowym, to punkt początkowy staje się punktem, w którym aktualnie znajduje się samochód, wróć do punktu 4.
10. 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 w momencie wyznaczania trasy nastąpiło losowanie ograniczeń prędkości tymczasowych na drogach.

3. Opis struktury programu.

3.1. Opis struktury.

3.1.1. 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.

3.1.2. 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.

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

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,

- **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

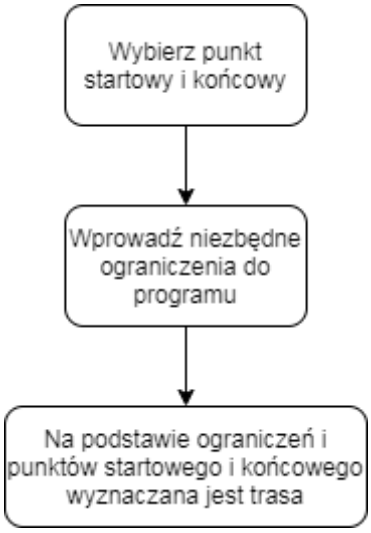
- **Crosses.png** – plik finalny mapy Warszawy, który jest używany w programie,

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

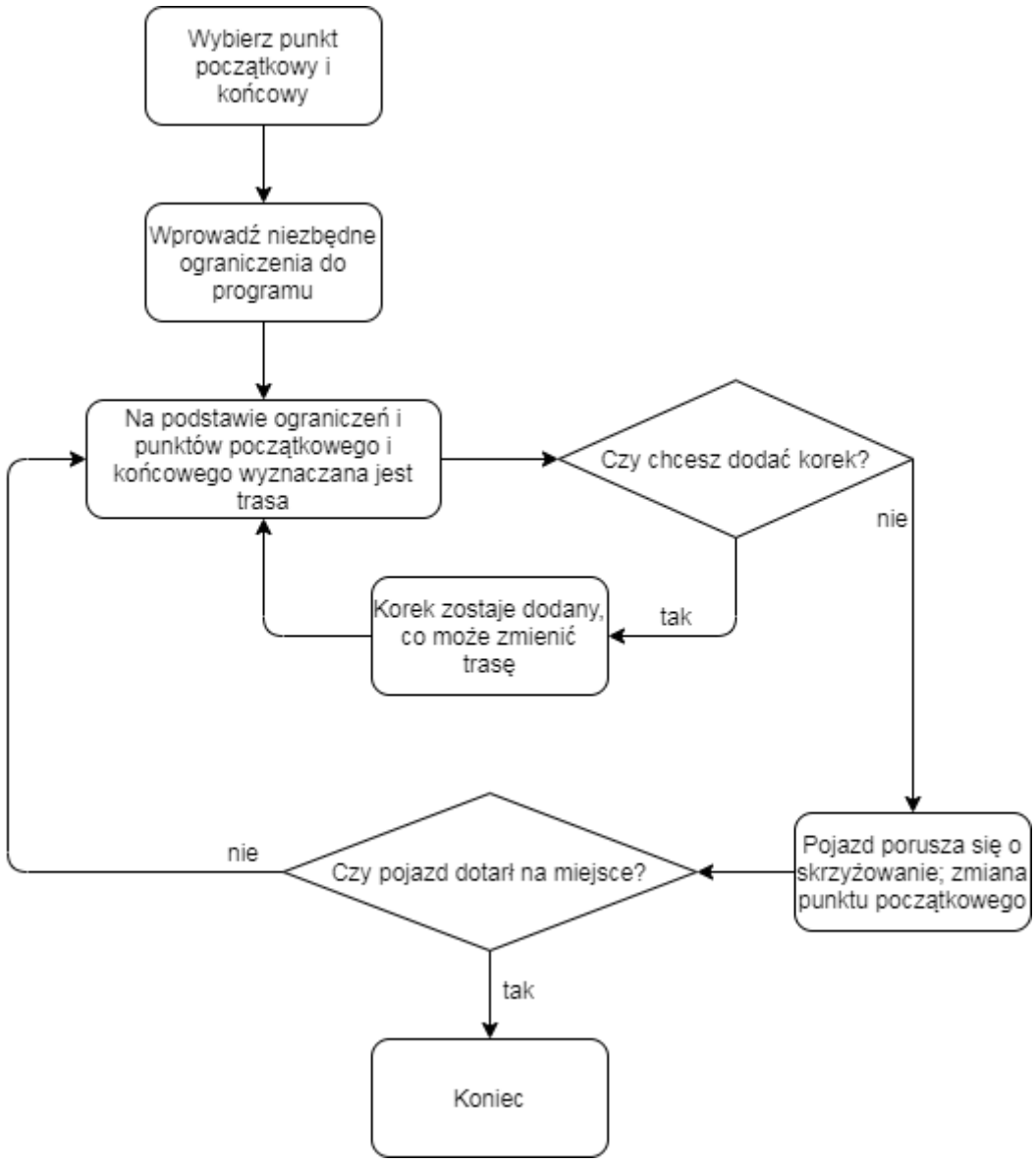
- **left arrow.jpg** – plik, w którym jest zdjęcie strzałki, która jest używana, do zaznaczania ulic jednokierunkowych.

3.2. Schematy blokowe.

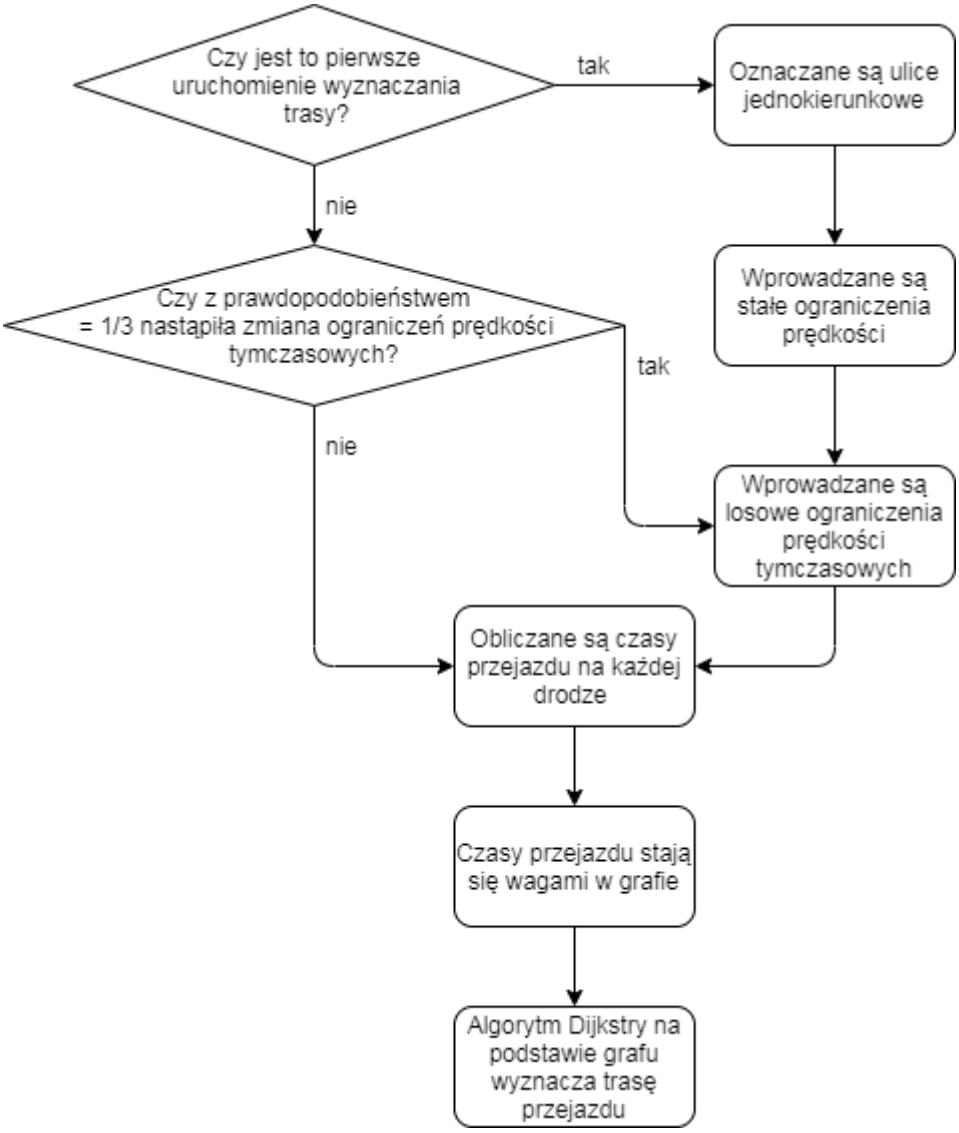
3.2.1. Schemat blokowy działania programu, dla jednorazowego wyznaczenia trasy.



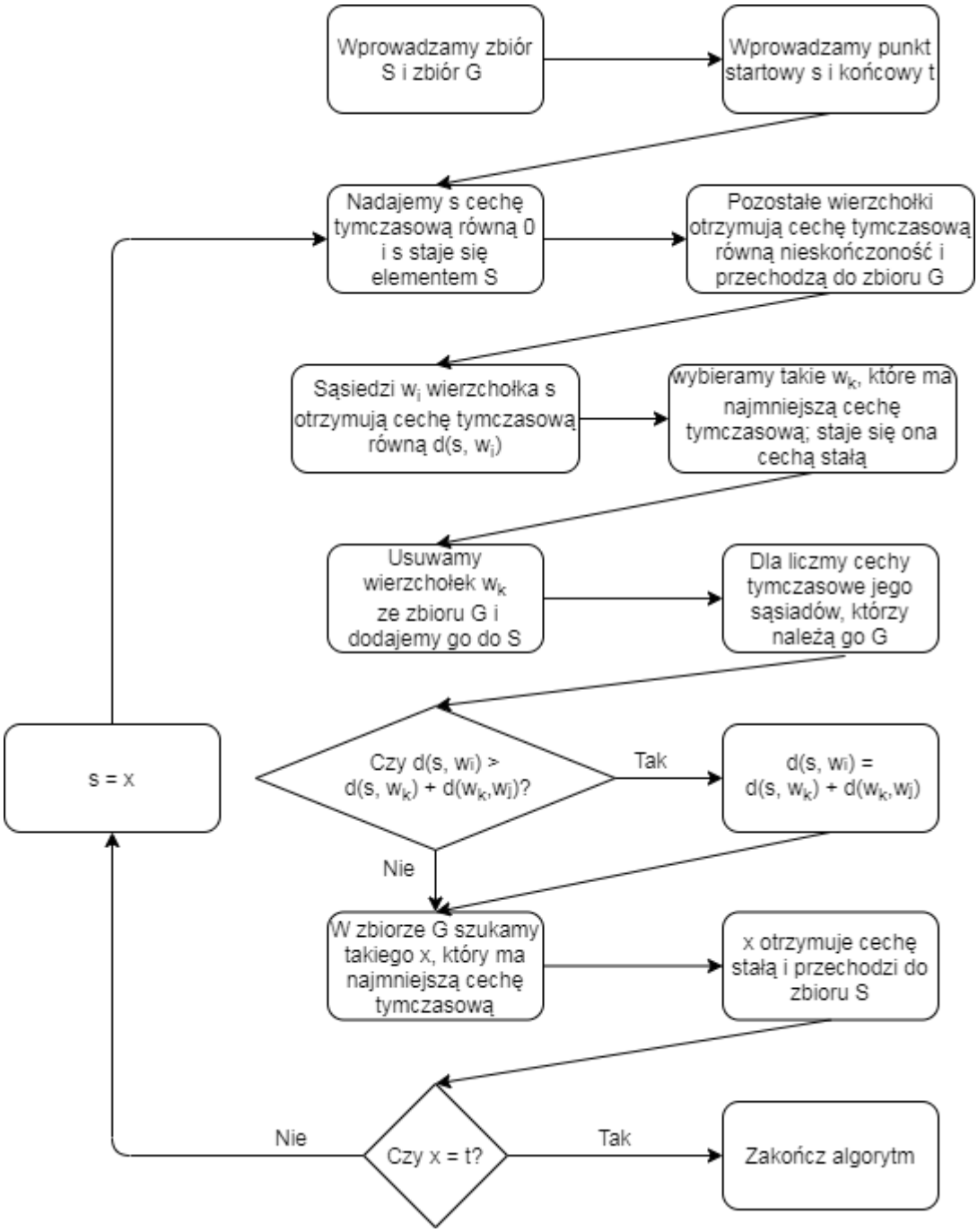
3.2.2. Schemat blokowy działania programu, dla iteracyjnego wyznaczenia trasy.



3.2.3. Schemat blokowy wyznaczania trasy.



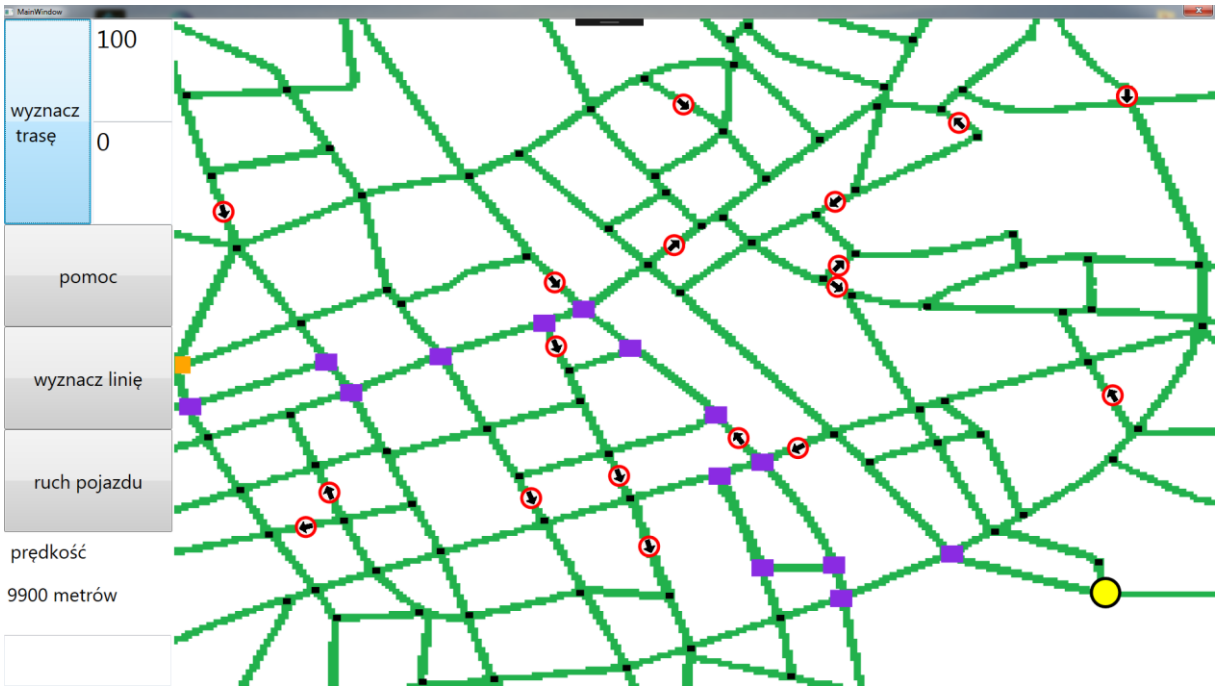
3.2.4. Schemat blokowy algorytmu Dijkstry.



3.3. 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 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. 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 trasę** oraz **pomoc**. Na dole napisy prędkość i pozostała droga, będą wyświetlać wartości prędkości i odległości w trakcie jazdy. Po wciśnięciu **wyznacz trasę** 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.



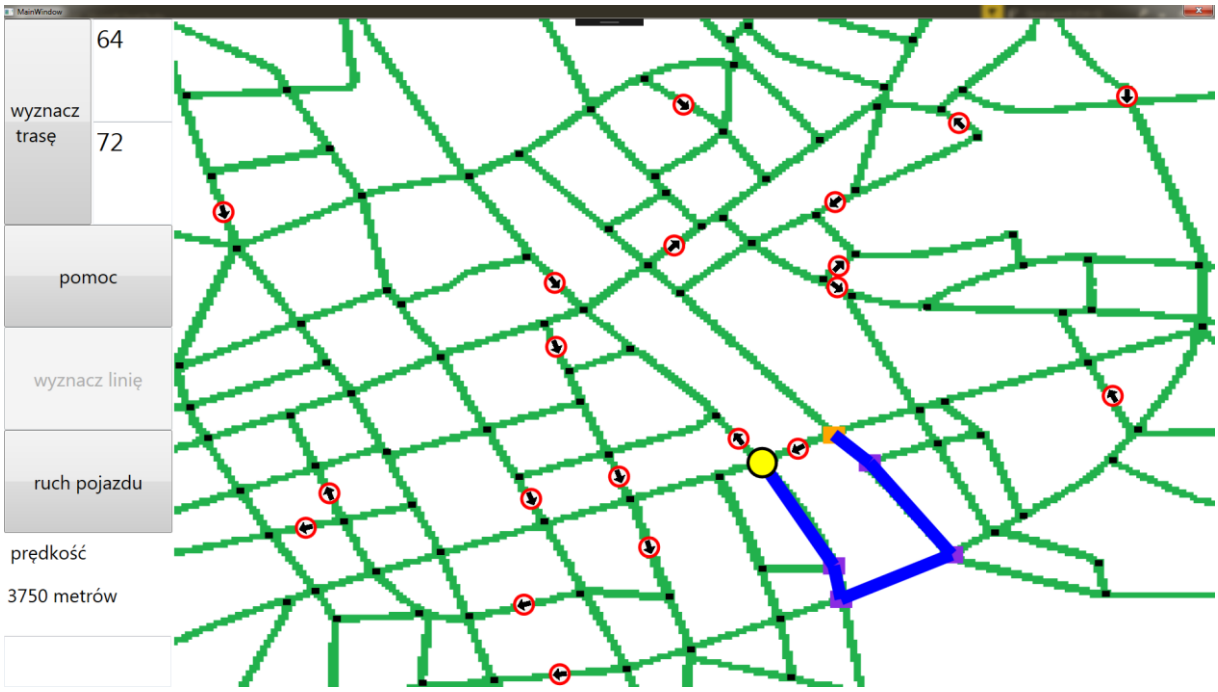
Po kliknięciu **wyznacz linię** go na mapie rysuje się krzywa łamana i przycisk staje się nieaktywny.

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.

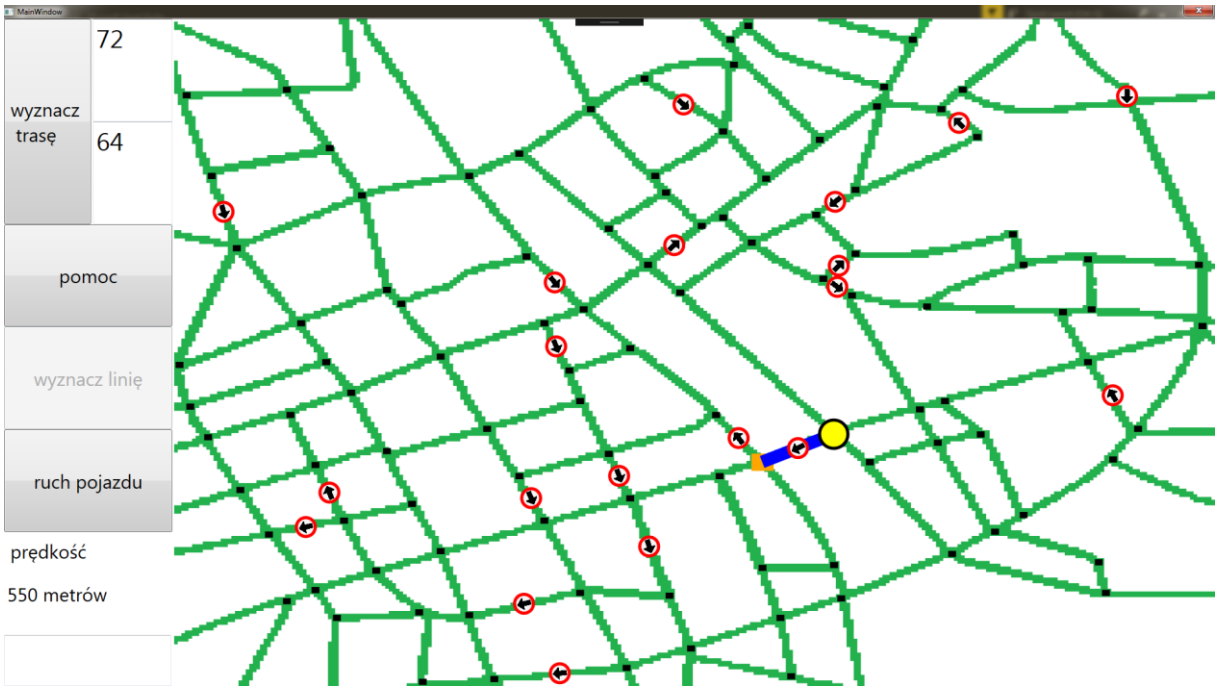
Po dojechaniu do skrzyżowania ponownie możemy kliknąć **wyznacz trasę.** Z prawdopodobieństwem $\frac{1}{3}$ prędkości tymczasowe na drogach się zmienią, co może skutkować zmianą trasy. Żeby mieć pewność, że trasa się zmieni, wciśnijmy prawym przyciskiem myszy na jedną z niebieskich linii. W ten sposób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

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

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.


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

3.4. Wybrane fragmenty kodu.

3.4.1. Ruch samochodu.

Ten fragment kodu odpowiada za poruszanie się samochodem na mapie.

```
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;
523    double x = x1 - x0;
524    double y = y1 - y0;
525    angle = y / x;
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
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 – $\mathbf{x_0,x_1,y_0,y_1}$. Następnie w linijce **525** obliczamy kąt pod jakim pojazd będzie się poruszał po mapie (góra/dół) oraz w linijkach **526-531** kierunek (prawo/lewo). W linijkach **533-537** przesuwamy pojazd o jeden piksel w prawo/lewo, a w linijce **538** przesuwamy o jeden piksel pomnożony przez kąt. 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.

```
265     //losowanie prędkości dróg
266     var rnd = new Random(0);
267     var rnd2 = new Random();
268     if (rnd2.Next(3)==0 || crosses[0].velocity.Count == 0)
269     {
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             {
276                 int x = rnd.Next(0, 14);
277                 if (x == 0)
278                     crosses[i].velocity.Add(0);
279                 else
280                 {
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. Zauważmy, że to są liczby z generatora **rnd**, który ma stałe ziarno, więc drogi jednokierunkowe zawsze będą takie same. 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 prędkość równą ograniczeniu stałemu.

3.4.3. Macierz incydencji.

Ten fragment kodu tworzy macierz incydencji.

```
309     var graph = new double[crosses.Count, crosses.Count];
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             {
316                 if (crosses[i].neighbours[k] == crosses[j].index)
317                 {
318                     graph[i, j] = crosses[i].distance[k] * 1/
                        crosses[i].velocity[k];
319                     break;
320                 }
321                 else
322                     graph[i, j] = 0;
323             }
324         }
325     }
326     return graph;
```

W tej potrójnej pętli dla każdej pary skrzyżowań (**i,j**) ustalamy wagę w grafie równą czasom przejazdu. 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.

3.4.4. Znak drogi jednokierunkowej.

Ten fragment kodu tworzy znaki jednokierunkowe w postaci strzałki.

```
371     //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
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. 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.