Systemy agentowe
Generowanie ruchu w systemie Kraksim na podstawie danych z ZDKiA

Tomasz Adamski
Paweł Pierzchała

1. Cel

Celem projektu jest aktualizacja modelu Krakowa wykorzystywanego w programie Kraksim, tak aby ruch w nim generowany odpowiadał rzeczywistości.

2. Mapa

Z eksperymentów poprzednio przeprowadzonych wynika, że znaczna część ruchy odbywała się poza głównymi drogami. Aby tego uniknąć należy:

- Poprawić informację o drogach jednokierunkowych
- Ustawić limity prędkości na drogach w centrum (w rzeczywistości duża liczba zaparkowanych samochodów zmniejsza ich przepustowość)

3. Schemat ruchu

Z danych dostarczonych przez ZDiK należy wygenerować graf przepływu pojazdów. Na jego podstawie możemy oszacować rozkłady ruchu między skrzyżowaniami. Tam gdzie liczba wjeżdżających oraz wyjeżdżających będzie się znacząco różniła należy wstawić bramy:

- no obrzeżach będą źródłami ruchu

- w centrum parkingami

4. Proponowane rozwiązania

Do tej pory rozważaliśmy głównie skonstruowanie układu równań, w którym poszczególne równania odpowiadałyby skrzyżowaniom. Niestety mamy wątpliwości czy taki sposób rozwiązania zadania jest poprawny. Ruch na ulicach jest ciągły, a więc każdy z pojazdów może włączyć się do ruchu w dowolnym punkcie mapy, a wobec tego wszystkie niezmienniki jakie moglibyśmy sobie założyć podczas konstrukcji układu równań (np. Mówiące, że liczba samochodów wjeżdżająca w ulicę jest równa liczbie samochodów wyjeżdżających z tej ulicy jest identyczna) są fałszywe. Uważamy, że do rozwiązania tego problemu należy użyć heurystycznej metody, która zminimalizuje różnice błędu między danymi ZDKiA a danymi wynikowymi algorytmu.

Zdecydowaliśmy się na rozwiązanie tego problemu poprzez skorzystanie z algorytmów genetycznych. Osobniki w populacji rozwiązań będą odpowiada różnym możliwym schematom ruchu, a funkcja przystosowania danego osobnika będzie tym większa im ruch ten będzie przypominał dane ZDTiK. Przyjrzyjmy się szczegółom tego rozwiązania. Pierwszym krokiem było wybranie sposobu zapisu ruchu w chromosomach osobnika w taki sposób, żeby wygodnie poddawał się on wszystkim operatorom genetycznym, a ponadto, żeby na jego podstawie można było w łatwy sposób (bez żadnych dodatkowych obliczeń i przekształceń) generować ruch. Zdecydowaliśmy się na sposób zapisu powiązany ze sposobem generowania ruchu przez Kraksim. Przypomnijmy, że Kraksim wypuszcza z określoną częstotliwością samochody z bramy i każdemu z nich nadaje kierunek (bramę docelową). W naszej reprezentacji każdy osobnik posiada tyle genów ile jest uporządkowanych par bram w całym grafie, a każdy gen oznacza ilość samochodów wypuszczonych z bramy stanowiącej pierwszy element pary do bramy stanowiącej drugi element pary. Funkcja przystosowania obliczana jest w nastepujący sposób: dla każdej z uporządkowanych par znana jest najkrótsza ścieżka między pierwszym a drugim wierzchołkiem z pary. Ścieżki te są obliczane na początku programu za pomocą algorytmu Dijkstry. Ruch na każdej krawędzi na początku obliczania funkcji przystosowania wynosi

zero (jest każdorazowo resetowany przed rozpoczęciem obliczania). Znając najkrótszą ścieżkę między elementami pary oraz ruch generowany przez tą parę dodajemy ten ruch do ruchu na każdej krawędzi z tej najkrótszej ścieżki. Operacje taką wykonujemy dla wszystkich par (a więc, inaczej mówiąc, dla wszystkich genów danego osobnika). Następnie obliczamy różnicę między średnią samochodów wjeżdzających i wyjeżdzających dla danej krawędzi (taki format mają dane z ZDTiK). Jest to podejście heurystyczne, jednak pozwala ono na uniknięcie komplikacji związanych z ewentualnym korygowaniem niespójnych danych. Wartości bezwględne tak obliczonych różnic dla wszystkich krawędzi sumujemy. Ostatecznym wynikiem jest iloraz liczby 1 i otrzymanej sumy. Po wykonaniu takich obliczeń otrzymujemy funkcję przystosowania o pożadanych własnościach: rosnącą wraz ze zbliżaniem się ruchu generowanego przez rozwiązanie do danych z ZDTiK oraz omijającą nieścisłości w danych. Całość algorytmu jest realizacją standardowego algorytmu genetycznego: algortm wykonuje się dla zadanej liczności populacji i zadanej liczby kroków. Selekcja przebiega zgodnie z metodą turnieju binarnego, krzyżowania losuje wartość genu podobną do genów rodziców. Mutacja zmienia z małym prawdopodobieństwem geny rozwiązania. Metody te zostaną opisane dokładnie w opisie matematycznym.

Projektując rozwiązanie w taki sposób chcieliśmy zaimplementować uniwersalną metodę generowania ruchu w systemie Kraksim dla zadego obciążenia na poszczególnych ulicach. Rozwiązanie takie, jeżeli okaże się skuteczne, odcina nas od konieczności doprowadzenia danych o ruchu drogowym do matematycznej dokładności. Dane o ruchu drogowym są z natury niespójne gdyż ilość możliwych włączania się do ruchu miejsc nie jest dyskretna, a pomiar niedokładny. Jak wspomniano wyżej może to stwarzać problemy: układ równań może być sprzeczny (bardzo prawdopodobne przy niespójnych danych). Ale nawet gdyby udało się «wygładzić» dane, to nie mielibyśmy pewności, czy układ nie okaże się nieoznaczony. Taka sytuacja jest z kolei prawdopodobna przy zbyt dużej ilości bram. Metoda heurystyczna powinna natomiast zawsze znaleźć rozwiązanie, które można skalibrować rozsądnym doborem bram.

Matematyczny opis rozwiązania

Graf

Dany jest graf skierowany G(V,E) reprezentujący mapę drogową. Wierzchołki grafu reprezentują skrzyżowania oraz wjazdy do miasta, a krawędzie grafu reprezentują pasy ruchu pomiędzy skrzyżowaniami. Dodatkowo wyróżniamy podzbiór wierzchołków S (v1,v2,...,vn) reprezentujących bramy, z których mogą wyjeżdzać i do których mogą wjeżdzać auta. Jeżeli zakładamy liczbę bram n, to liczba par bram wynosi m=n*n. Zbiór par oznaczamy przez P. Wybieramy arbitralnie uporządkowanie p zbioru par, będące funkcją różnowartościową ze zbioru 1..m w zbiór P.

Osobnik

Osobnik jest uporządkowaną m-ką liczb. Liczba o numerze i w tej m-ce będzie poniżej nazywana i-tym genem danego osobnika. I-ty gen odpowiada i-tej parze ze zbioru P zgodnie z uporządkowaniem p.

Funkcja przystosowania

W celu obliczenia funkcji przystosowania podejmujemy następujące kroki: Dla każdego z genów obliczmy przyczynek ruchu generowany przed odpowiadającą mu parę bram. Dla każdej pary (s,d) bram znamy najkrótszą ścieżkę: k1,k2,...,kl, gdzie ai należy do E dla każdego i (te najkrótsze ścieżki są obliczane na początku jednorazowo na początku programu zgodnie z algorytmem Dijkstry). Oznaczmy ponadto przez x ilość aut które wyjeżdzają z bramy s do bramy d w rozważanej jednostce czasu (jeżeli brama ta odpowiada i-temu genowi osobnika, to wartość tego genu wynosi x). Przy takich oznaczeniach możemy założyć, że para (s,d) zwiększa obciążenie na każdej krawędzi ze zbioru k1,k2,..,kl o x. Przeprowadzając identyczną analizę jesteśmy w stanie wyznaczyć dla każdej z krawędzi

sumę obciążeń generowanych przez wszystkie pary: oznaczmy przez p1,p2,...,pl zbior tych par, których najkrótsza trasa przechodzi przez krawędź a. Przy takim oznaczeniu obciążenie generowane na krawędzi a będzie równe sumie obciążeń generowanych przez wszystkie te pary. Po wykonaniu tych obliczeń sumujemy błąd e na wszystkich krawędziach: Dla każdej z krawędzi k oznaczmy jej błąd przez ek, sumę aut wjeżdzających przez x_k_wj, a wyjeżdzających przez x_k_wj, a sumę obciążenia przez x_k. Wtedy zachodzi ek=| (x_k_wj+x_k_wyj)/2 — x_k |. Po zsumowaniu wartości e_k wszystkich krawędzi otrzymujemy błąd e. Wartością funkcji przystosowania danego osobnika jest odwrotność tego błędu 1/e.

Selekcja

Selekcja odbywa się za pomocą metody turnieju binarnego: Oznaczmy przez s liczność populacji. W ramach selekcji wykonujemy s-krotnie następujący krok: Losujemy dwóch osobników z populacji, wybieramy tego z większą wartością funkcji przystosowania i wrzucamy ją do puli rodzicielskiej. Dwóch wylosowanych osobników nie zostanie usuniętych z populacji.

Krzyżowanie

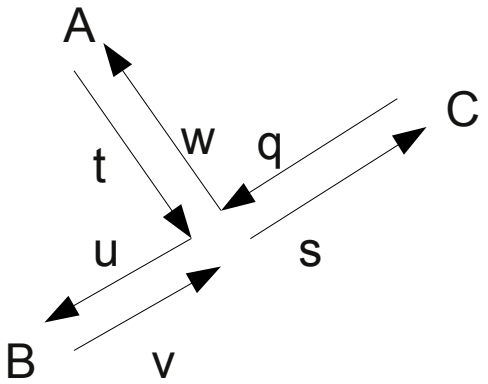
Podczas krzyżowania losujemy s/2 krotnie parę osobników z puli rodzicielskiej. Po każdym losowaniu usuwamy oba wylosowane osobniki z puli rodzicielskiem. Każda z takich par generuje dwóch potomków, którzy są umieszczani w nowej populacji. Podczas operacji krzyżowania dwóch osobników wykonujemy następujące czynności: dla każdego z genów dwóch krzyżowanych osobników: Oznaczmy przez g_a wartość i-genu osobnika a i g_b wartość i-tego genu osobnika b. Oznaczmy przez delta wartość |g_a-g_b|, i przez avg wartość (g_a+g_b)/2. Wartość i-tego genu osobnika potomnego jest wówczas losowana z przedziału (avg-delta,avg+delta). Taka metoda zapewnia, że osobniki zachowują podobieństwo do swoich rodziców, ale ich cechy mogą zmieniać się w dowolnym kierunku względem cech rodziców.

Mutacja

Mutacja polega na zmianie z małym prawdopodobieństwem losowych genów członków nowej populacji.

Po wykonaniu operacji selekcji, krzyżowania i mutacji nowa populacji zastępuję starą.

Przykład:



Rozważmy parę wierzchołków A i B — najkrótsza trasa między nimi to t,u. Jeżeli obciążenie generowane przez parę AB oznaczymy przez xAB, to wiemy, że xAB będzie częścią obciążenia krawędzi t i u. Rozważając w podobny sposób wszystkie pary i oznaczając przez xt obciążenie docelowe na krawędzi t otrzymamy następujący sposób obliczania obciążenia na wszystkich krawędziach:

xAB=xt xBA+xBC=xv xCA+xCB=xq

xAB+xCB=xu xBC+xAC=xs xCA+BA=xw

W związku z tym, że graf posiada sześć uporządkowanych par bram, to osobniki w populacji składają się z sześciu genów. Wybierzmy następujące uporządkowanie par: AB,AC,BA,BC,CA,CB i rozważmy osobnika [3,5,4,1,7,3]. Wtedy mamy następujące obciążenie na krawędziach:

xt=3 xv=4+1=5 xq=7+3=10

xu=3+3=6 xs=5+1=6 xw=7+4=11

Załóżmy teraz, że zadany ruch wynosi dla poszczególnych krawędzi (podane liczby to ruch wjazdowy i wyjazdowy:

t->4,5 v->7,6 q->12,14

u->5,7 s->6,8 w=10,9

Obliczmy zatem wartość funkcji przystosowania f:

e=|4.5-3|+|6.5-5|+|13-10|+|6-6|+|6-7|+|11-9.5|=1.5+1.5+3+0+1+1.5=8.5

f=1/8.5=2/17

Przyjrzyjmy się jeszcze operacji krzyżowania:

Załóżmy, że drugi osobnik jest postaci [1,7,10,3,11,5]. Przypomnijmy, że pierwszy osobnik to: [3,5,4,1,13,3]. Wtedy dla poszczególnych genów:

1 → avg=2 delta=2 przedział: (0,2)

2 → avg=6 delta=2 przedział: (4,8)

3 → avg=7 delta=6 przedzial: (1,13)

4 → avg=3 delta=2 przedzial: (1,5)

5 → avg=12 delta=2 przedzial: (10,14)

6 → avg=4 delta=2 przedział: (2,4)

Opis implementacji

Algorytm został zaimplementowany w języku Java. Rozwiązanie składa się z dwóch pakietów: graph oraz algorithm. W pakiecie graph znajdują się wszystkie klasy związane z grafem reprezentującym ruch w mieście, a w klasie algorithm wszystkie klasy związane z algorytmem genetycznym.

Graph.Graph

Klasa reprezentuje graf. Enkapsuluje ona listę wierzchołków i pozwala na wykonywanie takich operacji jak: dodawanie wierzchołków, pobieranie wierzchołków i listy bram, resetowanie obciążenia na krawędziach, drukowania grafu w celach debugu. Jest ważną częścią jest statyczna funkcja parseGraph, która pozwala na wczytanie pliku z grafu w następującym formacie:

n

g_1 … g_n

m

k_ij k_oj k_lj k_icj k_ocj

...

gdzie: n- ilość krawędzi, gi — informuje czy wierzchołek i jest bramą (0 — nie , 1 — tak), m - ilość krawędzi, ki — numer wierzchołka wejściowego krawędzi j, numer wierzchołka wyjściowego krawędzi j, kl — długość krawędzi j, kic — ilość samochodów wjeżdzających do krawędzi j, ilość samochodów wyjeżdzających z krawędzi j

Graph.Vertex

Klasa reprezentująca wierzchołek grafu. Enkapsulująca takie właśności jak numer, listę krawędzi i informacje czy wierzchołek jest bramą. Dodatkowo posiada pola potrzebne podczas obliczania algorytmu Dijkstry.

Graph.Edge

Klasa reprezentująca krawędź grafu. Enkapsulująca takie informacje jak ruch na tej krawędzi (dla analizowanego w danej chwili osobnika), długość krawędzi, liczbę samochodów wjeżdzających i wyjeżdzających.

Graph.Path

Reprezentuje ścieżkę rozpatrywaną przez algorytm Dijkstry — implementacyjne: lista krawędzi.

Algorithm.Individual

Reprezentuje osobnika w populacji. Enkapsuluje jego geny oraz fitness.

Algorithm.GatePair

Klasa pomocnicza reprezentująca parę bram i używana przez klasę główną Algorithm.

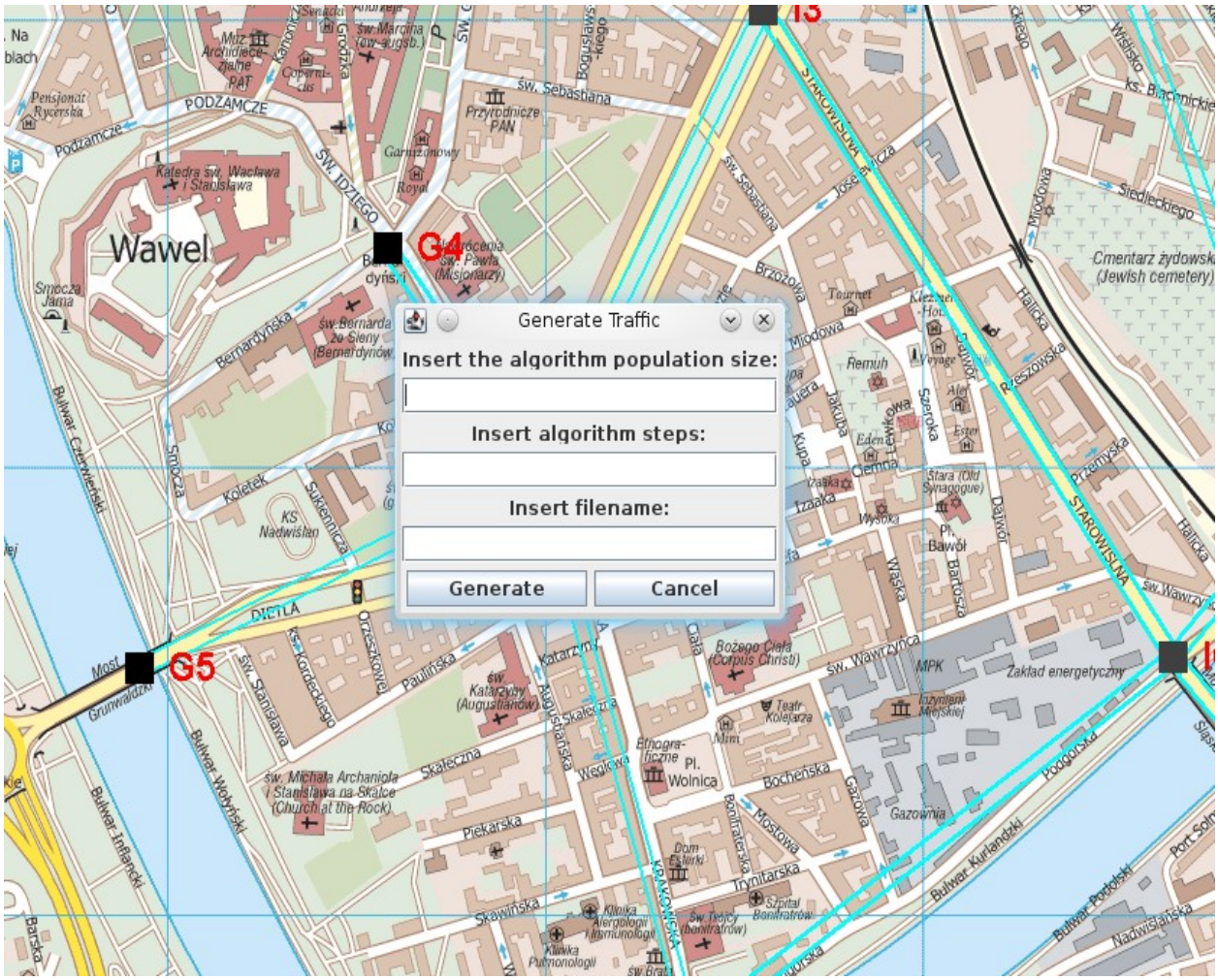
Algorithm.Algorithm

Klasa główna reprezentująca cały algorytm, enkapsuluje ona wszystkie funkcje związane z algorytmem genetycznym takie jak: przygotowanie populacji, selekcje, krzyżowanie i mutacje. Funkcje te są prywatne a obiekt klasy Algorithm umożliwia uruchomienie eksperymentu poprzez wywołanie funkcji perform.

Modyfikacje programu KraksimCityDesinger

Generowanie ruchu na podstawie danych

Po stworzeniu modelu ruchu, oraz przypisaniu każdemu ze skrzyżowań pliku danych opisującego ruch na tym skrzyżowaniu możemy przystąpić do generowania obciążenia ruchu. W tym celu klikamy w menu «Traffic» a następnie w «GenerateTraffic». Pokazuje się wówczas następujące okno:

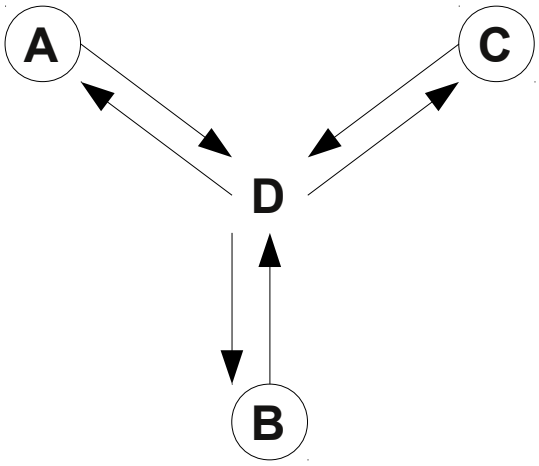


Wprowadzamy do niego dane dla algorytmu ewolucyjnego. Pierwszy z parameterów ro rozmiar populacji, a drugi ilość kroków algorytmu. Po kliknięciu przycisku «Generate» rozpoczyna się obliczanie schematu ruchu. Paremetr trzeci, to ścieżka do pliku, w którym ma być zapisany wygenerowany schemat ruchu. Jest ono stosunkowo długotrwałe i powoduje zablokowanie pracy KraksimDesignera na czas obliczeń. Postęp algorytmu można śledzić w oknie konsoli. Po zakończeniu obliczeń zostanie wygenerowany plik xmlowy opisujący ruch w systemie, którego można użyć do generowanie ruchu w symulacji programu Kraksim.

Przykłady

Trójkąt

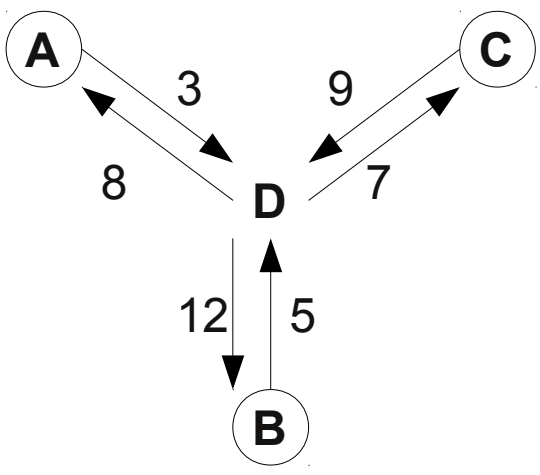
Celem przykładu jest sprawdzenie poprawności algorytm na bardzo prostym zestawie danych. Wejściem do algorytmu jest graf składający się z czterech wierzchołków i przedstawiony na rysunku poniżej:



Wierzchołki są oznaczone dużymi literami, a krawędzie strzałkami, bramy zostały zaznaczone okręgiem. W celu wygenerowania ruchu na krawędziach założyliśmy losowe obciążenie między poszczególnymi parami wierzchołków. Tabela tych obciążeń przedstawiona jest poniżej:

Brama	**Obciążenie**
AB	3
AC	5
BA	2
BC	7
CA	1
CB	4

Po przeliczeniu całkowitego obciążenia, na krawedziach, jakie generują te bramy otrzymaliśmy następujący graf (przy każdej krawędzi, jest tylko jedna liczba reprezentująca obciążenie, gdyż dane są spójne i obciążenie wejściowe krawędzi jest zawsze równe jej obciążeniu wyjściowemu):



A więc plik wejsciowy będzie miał następującą postać:

```
4

1 1 1 0
6
0 3 1 8 8
1 3 1 9 9
2 3 1 5 5
3 0 1 3 3
3 1 1 7 7
3 2 1 12 12
```

Po uruchomieniu algorytmu dla takich danych wejściowych przy zadanej liczbie kroków 500 i populacji 200 otrzymaliśmy następujące wyniki (wyście debugu):

```
 NAJLEPSZY JEST TAKI GOSC:
[3.523305950740986, 4.476694049259014, 1.4766940492590135,
7.523305950740987, 1.5233059507409865, 3.476694049259014]
A JEGO FITNESS TO:
Wierzcholek 0
      z 0 do 3 traffic 8.0 input 8 output 8
Wierzcholek 1
      z 1 do 3 traffic 9.0 input 9 output 9
Wierzcholek 2
      z 2 do 3 traffic 5.0 input 5 output 5
Wierzcholek 3
      z 3 do 0 traffic 3.0 input 3 output 3
      z 3 do 1 traffic 7.0 input 7 output 7
      z 3 do 2 traffic 12.0 input 12 output 12
```

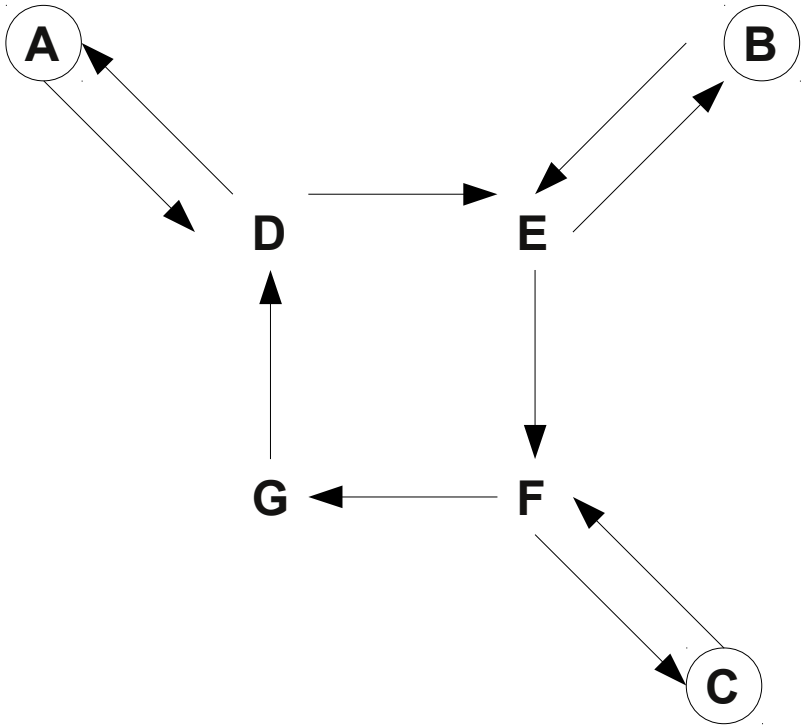
Jak widać dane zwrócone przez algorytm nie są identyczne jak dane użyte do generowania grafu. Mimo tego, jak pokazuje wyjście, obciążenie wygenerowane na krawędziach zgadza się z założonym obciążeniem. Po uruchomieniu algorytmu jeszcze raz:

```
 NAJLEPSZY JEST TAKI GOSC:
[3.3598735217606928, 4.640126478239307, 1.6401264782393072,
7.359873521760693, 1.3598735217606928, 3.6401264782393072]
A JEGO FITNESS TO:
Wierzcholek 0
      z 0 do 3 traffic 8.0 input 8 output 8
Wierzcholek 1
      z 1 do 3 traffic 9.0 input 9 output 9
Wierzcholek 2
      z 2 do 3 traffic 5.0 input 5 output 5
Wierzcholek 3
      z 3 do 0 traffic 3.0 input 3 output 3
      z 3 do 1 traffic 7.0 input 7 output 7
      z 3 do 2 traffic 12.0 input 12 output 12
```

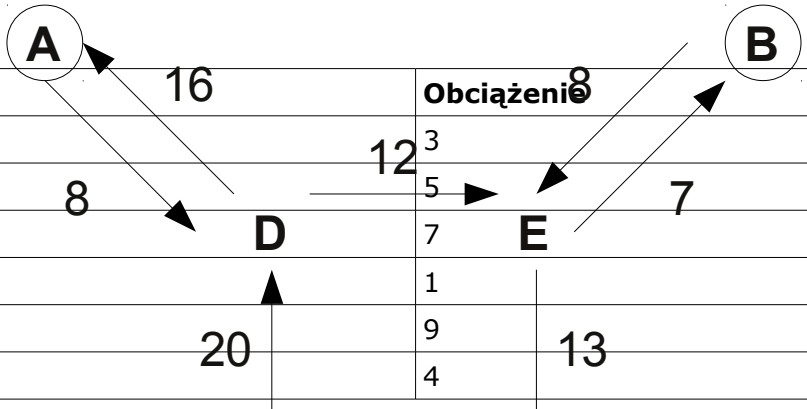
Otrzymujemy zupełnie innego osobnika, który jednak również generuje ruch w sposób idealny. Okazało się więc, że mimo idealnie wygładzonych danych układ równań odpowiadający temu problemowi nie jest oznaczony. Nie występuje jednak sprzeczność, a nieoznaczoność. Algorytm heurystyczny ma tutaj przewagę nad metodą układu równań, gdyż radzi sobie zarówno z niespójnymi danymi jak i nieoznaczonością bez podejmowania dodatkowych działań.

Rondo

Celem przykładu jest sprawdzenie działania algorytmu na bardziej skomplikowanym grafie:



Obciążenie generowane przez bramy wygląda następująco:



Brama	**Obciążenie**
AB	3
AC	5
BA	7
BC	1
CA	9
CB	4

Po przeliczeniu generowanego obciążenia graf wygląda następująco:

Plik wejściowy jest następujący:

```
7
1 1 1 0 0 0 0
10
0 3 1 8 8
3 0 1 16 16
1 4 1 8 8
4 1 1 7 7
2 5 1 13 13
5 2 1 6 6
3 4 1 12 12
4 5 1 13 13
5 6 1 20 20
6 3 1 20 20
```

Po uruchomieniu algorytmu dla takiego pliku wejściowego dla liczby kroków równej 1500 i populacji równej 200 otrzymaliśmy następujące wyniki:

```
 NAJLEPSZY JEST TAKI GOSC:
[3.075563939238358, 4.9816377046416065, 6.9494645637430414,
1.0505354362569588, 9.126099375495317, 3.924436060761642]
A JEGO FITNESS TO:
Wierzcholek 0
       z 0 do 3 traffic 8.057201643879964 input 8 output 8
Wierzcholek 1
       z 1 do 4 traffic 8.0 input 8 output 8
Wierzcholek 2
       z 2 do 5 traffic 13.050535436256958 input 13 output 13
Wierzcholek 3
       z 3 do 0 traffic 16.075563939238357 input 16 output 16
       z 3 do 4 traffic 11.981637704641606 input 12 output 12
Wierzcholek 4
```

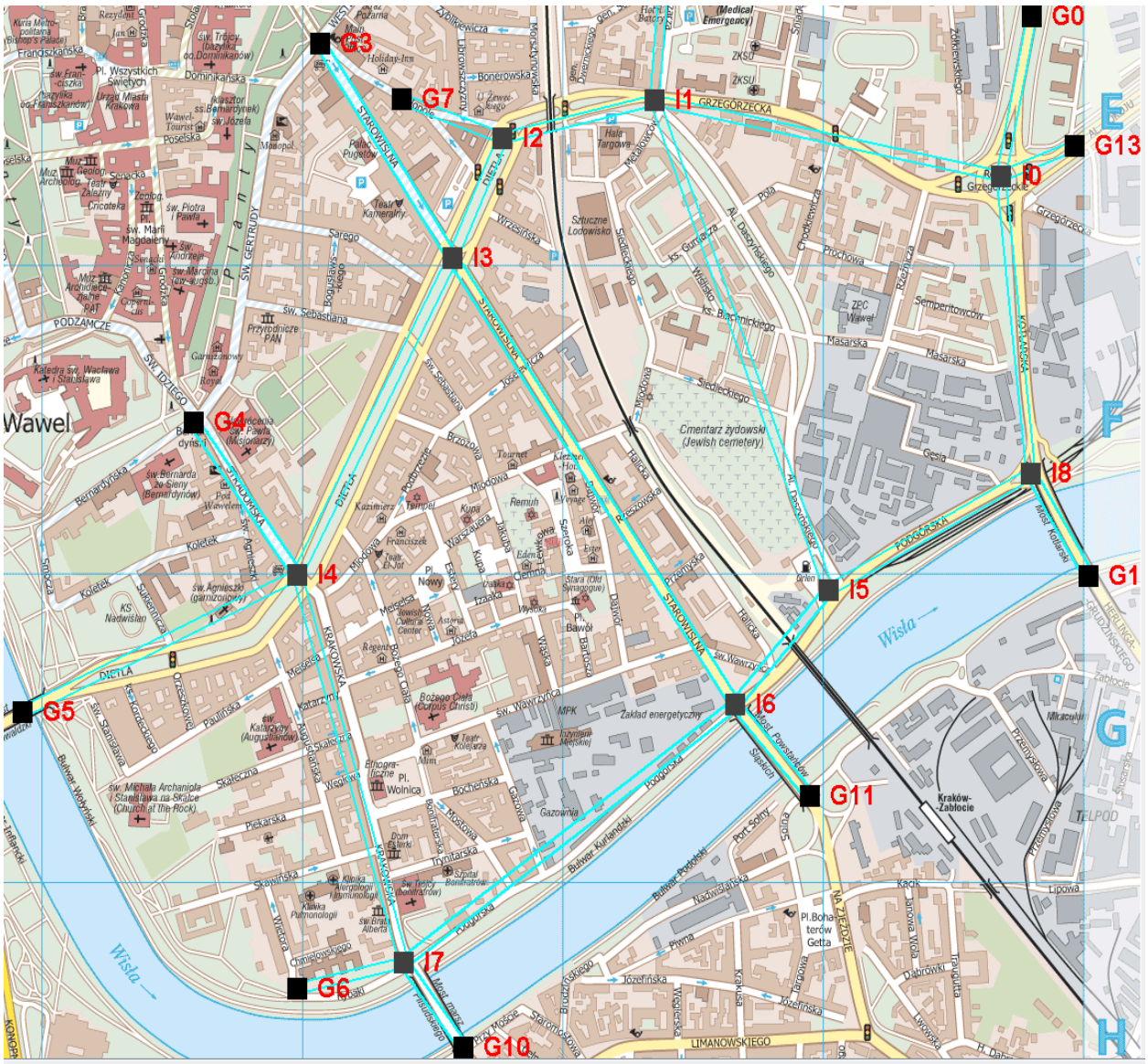
```
      z 4 do 1 traffic 7.0 input 7 output 7
      z 4 do 5 traffic 12.981637704641608 input 13 output 13
Wierzcholek 5
      z 5 do 2 traffic 6.032173140898565 input 6 output 6
      z 5 do 6 traffic 20.0 input 20 output 20
Wierzcholek 6
      z 6 do 3 traffic 20.0 input 20 output 20
```

Jak widać wyniki nie są idealne, ale są wystarczająco dokładne, aby uznać je za zadowalające.

W powyższych przykładach generowaliśmy graf na podstawie założonego obciążenia bram, które generowało spójny ruch na wszystkich krawędziach grafu. Przykłady te pokazały, że algorytm działa poprawnie jednak nie było w stanie przetestować zasadniczej własności, której oczekujemy od algorytmu heurystycznego: radzenia sobie z niespójnymi danymi bez konieczności ich wstępnego przetwarzania. W celu przetestowania tej cechy algorytmu zdecydowaliśmy się na użycie fragmentu mapy Krakowa i danych uzyskanych z ZDKiA. Pozwoli nam to na przeanalizowanie algorytmu na realnych danych, których niedokładności są charakterystyczne dla całości danych ZDKiA.

Kazimierz

Zdecydowaliśmy się na stworzenie przykładu, w którym ruch opiera się na danych z ZDKiA, dotyczącego części miasta, tak aby powstały graf był średnich (20 — 30 wierzchołków) rozmiarów. Jako część miasta, która będzie w tym przykładzie opisana wybraliśmy Kazimierz ze względu dla korzystny dla nas rozmiar dzielnicy i układ ulic. Mapa, na podstawie której został stworzony graf, została wprowadzona do systemu za pomocą modułu KraksimCityDesigner. Poniżej przedstawiamy stworzony schemat:



W powyższym schemacie bramy zostały oznaczone literami G, a skrzyżowania literami I. Połączenia między tymi elementami odzwierciedlone w naszym grafie zostały oznaczone podwójnymi niebieskimi liniami. Jak wspomniano wyżej, dane w przykładzie Kazimierzowskim nie są już wygenerowane przez nas, tylko pochodzą one z ZDKiA. W związku z tym graf ruchu jest automatycznie generowany przez program. Aby program poprawnie wygenerował taki schemat musimy przypisać każdemu ze skrzyżowań odpowiedni plik opisujący ruch. Zostało to dokładnie opisane w części dokumentacji opisującej zmiany interfejsu KraksimDesingera. Na podstawie danych wprowadzonych do systemu uzyskaliśmy następujący graf:

```
Wierzcholek 0
	z 0 do 18 traffic 0.0 input 1991 output 1991 avg 1991 dst 98
Wierzcholek 1
	z 1 do 17 traffic 0.0 input 2559 output 2559 avg 2559 dst 112
Wierzcholek 2
	z 2 do 11 traffic 0.0 input 4607 output 4607 avg 4607 dst 79
Wierzcholek 3
	z 3 do 15 traffic 0.0 input 4190 output 4190 avg 4190 dst 303
Wierzcholek 4
	z 4 do 11 traffic 0.0 input 4017 output 4017 avg 4017 dst 150
Wierzcholek 5
	z 5 do 19 traffic 0.0 input 0 output 0 avg 0 dst 110
Wierzcholek 6
```

```
	z 6 do 12 traffic 0.0 input 548 output 548 avg 548 dst 108
Wierzcholek 7
	z 7 do 14 traffic 0.0 input 1444 output 1444 avg 1444 dst 237
Wierzcholek 8
	z 8 do 15 traffic 0.0 input 792 output 792 avg 792 dst 174
Wierzcholek 9
	z 9 do 18 traffic 0.0 input 388 output 388 avg 388 dst 109
Wierzcholek 10
	z 10 do 13 traffic 0.0 input 0 output 0 avg 0 dst 107
Wierzcholek 11
	z 11 do 19 traffic 0.0 input 3909 output 3909 avg 3909 dst 274
	z 11 do 2 traffic 0.0 input 3999 output 3999 avg 3999 dst 79
	z 11 do 4 traffic 0.0 input 4656 output 4656 avg 4656 dst 150
	z 11 do 12 traffic 0.0 input 4091 output 3262 avg 3676 dst 354
Wierzcholek 12
	z 12 do 6 traffic 0.0 input 686 output 686 avg 686 dst 108
	z 12 do 13 traffic 0.0 input 3454 output 3157 avg 3305 dst 156
	z 12 do 11 traffic 0.0 input 2916 output 4153 avg 3534 dst 354
	z 12 do 16 traffic 0.0 input 336 output 336 avg 336 dst 482
Wierzcholek 13
	z 13 do 14 traffic 0.0 input 861 output 3276 avg 2068 dst 120
	z 13 do 10 traffic 0.0 input 2296 output 2296 avg 2296 dst 107
	z 13 do 12 traffic 0.0 input 4595 output 2864 avg 3729 dst 156
Wierzcholek 14
	z 14 do 7 traffic 0.0 input 886 output 886 avg 886 dst 237
	z 14 do 13 traffic 0.0 input 2604 output 4595 avg 3599 dst 120
	z 14 do 15 traffic 0.0 input 4390 output 2933 avg 3661 dst 330
	z 14 do 17 traffic 0.0 input 802 output 1550 avg 1176 dst 498
Wierzcholek 15
	z 15 do 18 traffic 0.0 input 850 output 1544 avg 1197 dst 371
	z 15 do 3 traffic 0.0 input 2992 output 2992 avg 2992 dst 303
	z 15 do 8 traffic 0.0 input 1281 output 1281 avg 1281 dst 174
	z 15 do 14 traffic 0.0 input 3715 output 2622 avg 3168 dst 330
Wierzcholek 16
	z 16 do 17 traffic 0.0 input 1480 output 1480 avg 1480 dst 140
	z 16 do 19 traffic 0.0 input 0 output 0 avg 0 dst 229
	z 16 do 12 traffic 0.0 input 718 output 718 avg 718 dst 482
Wierzcholek 17
	z 17 do 1 traffic 0.0 input 1473 output 1473 avg 1473 dst 112
	z 17 do 18 traffic 0.0 input 1782 output 666 avg 1224 dst 408
	z 17 do 14 traffic 0.0 input 2362 output 1340 avg 1851 dst 498
	z 17 do 16 traffic 0.0 input 1044 output 1044 avg 1044 dst 140
Wierzcholek 18
	z 18 do 0 traffic 0.0 input 1577 output 1577 avg 1577 dst 98
	z 18 do 15 traffic 0.0 input 2050 output 923 avg 1486 dst 371
	z 18 do 17 traffic 0.0 input 155 output 1072 avg 613 dst 408
	z 18 do 9 traffic 0.0 input 807 output 807 avg 807 dst 109
Wierzcholek 19
	z 19 do 11 traffic 0.0 input 3878 output 3878 avg 3878 dst 274
	z 19 do 5 traffic 0.0 input 0 output 0 avg 0 dst 110
	z 19 do 16 traffic 0.0 input 0 output 0 avg 0 dst 229
```

Pola w powyższych opisach oznaczają:

- traffic — ruch na krawędzi (jest on wszędzie równy zero gdyż jest to stan grafu przed rozpoczęciem obliczeń)
- input — ilość samochodów wjeżdzających do danej krawędzi
- output — ilość samochodów wyjeżdzających z danej krawędzi

- avg — średnia artmetyczna input i output (algorytm ewolucyjny stara się dopasować traffic do takiej wartości)
- dst — odległość między krawędziami na mapie

Po sprawdzeniu danych, okazało się, że program prawidłowo odczytał dane zawarte w plikach opisujących ruch, jednak niestety dane okazały się bardzo niespójne. Zobaczmy jak w takich warunkach spisał się nasz algorytm:

```
Wierzcholek 0
	z 0 do 18 traffic 1933 input 1991 output 1991 avg 1991 dst 98
Wierzcholek 1
	z 1 do 17 traffic 2559 input 2559 output 2559 avg 2559 dst 112
Wierzcholek 2
	z 2 do 11 traffic 4607 input 4607 output 4607 avg 4607 dst 79
Wierzcholek 3
	z 3 do 15 traffic 4174 input 4190 output 4190 avg 4190 dst 303
Wierzcholek 4
	z 4 do 11 traffic 4017 input 4017 output 4017 avg 4017 dst 150
Wierzcholek 5
	z 5 do 19 traffic 2702 input 0 output 0 avg 0 dst 110
Wierzcholek 6
	z 6 do 12 traffic 548 input 548 output 548 avg 548 dst 108
Wierzcholek 7
	z 7 do 14 traffic 1444 input 1444 output 1444 avg 1444 dst 237
Wierzcholek 8
	z 8 do 15 traffic 723 input 792 output 792 avg 792 dst 174
Wierzcholek 9
	z 9 do 18 traffic 336 input 388 output 388 avg 388 dst 109
Wierzcholek 10
	z 10 do 13 traffic 1159 input 0 output 0 avg 0 dst 107
Wierzcholek 11
	z 11 do 19 traffic 3909 input 3909 output 3909 avg 3909 dst 274
	z 11 do 4 traffic 4656 input 4656 output 4656 avg 4656 dst 150
	z 11 do 12 traffic 3665 input 4091 output 3262 avg 3676 dst 354
	z 11 do 2 traffic 3999 input 3999 output 3999 avg 3999 dst 79
Wierzcholek 12
	z 12 do 16 traffic 168 input 336 output 336 avg 336 dst 482
	z 12 do 6 traffic 711 input 686 output 686 avg 686 dst 108
	z 12 do 11 traffic 3727 input 2916 output 4153 avg 3534 dst 354
	z 12 do 13 traffic 3305 input 3454 output 3157 avg 3305 dst 156
Wierzcholek 13
	z 13 do 10 traffic 2296 input 2296 output 2296 avg 2296 dst 107
	z 13 do 14 traffic 2068 input 861 output 3276 avg 2068 dst 120
	z 13 do 12 traffic 3700 input 4595 output 2864 avg 3729 dst 156
Wierzcholek 14
	z 14 do 7 traffic 886 input 886 output 886 avg 886 dst 237
	z 14 do 17 traffic 1006 input 802 output 1550 avg 1176 dst 498
	z 14 do 15 traffic 2460 input 4390 output 2933 avg 3661 dst 330
	z 14 do 13 traffic 3599 input 2604 output 4595 avg 3599 dst 120
Wierzcholek 15
	z 15 do 8 traffic 1420 input 1281 output 1281 avg 1281 dst 174
	z 15 do 18 traffic 1197 input 850 output 1544 avg 1197 dst 371
	z 15 do 3 traffic 3058 input 2992 output 2992 avg 2992 dst 303
	z 15 do 14 traffic 3168 input 3715 output 2622 avg 3168 dst 330
Wierzcholek 16
	z 16 do 12 traffic 0 input 718 output 718 avg 718 dst 482
	z 16 do 17 traffic 1479 input 1480 output 1480 avg 1480 dst 140
	z 16 do 19 traffic 1818 input 0 output 0 avg 0 dst 229
Wierzcholek 17
```

```
      z 17 do 18 traffic 1096 input 1782 output 666 avg 1224 dst 408
      z 17 do 14 traffic 1271 input 2362 output 1340 avg 1851 dst 498
      z 17 do 16 traffic 1818 input 1044 output 1044 avg 1044 dst 140
      z 17 do 1 traffic 1473 input 1473 output 1473 avg 1473 dst 112
Wierzcholek 18
      z 18 do 17 traffic 613 input 155 output 1072 avg 613 dst 408
      z 18 do 15 traffic 1486 input 2050 output 923 avg 1486 dst 371
      z 18 do 9 traffic 865 input 807 output 807 avg 807 dst 109
      z 18 do 0 traffic 1597 input 1577 output 1577 avg 1577 dst 98
Wierzcholek 19
      z 19 do 5 traffic 3240 input 0 output 0 avg 0 dst 110
      z 19 do 11 traffic 3878 input 3878 output 3878 avg 3878 dst 274
      z 19 do 16 traffic 1311 input 0 output 0 avg 0 dst 229
ŚREDNI ERROR WYSZEDŁ:
0.09489745178755196
```

Jak widać otrzymane wyniki są zadowalające. W większości przypadków algorytm dobrze aproksymuję oczekiwaną liczbę samochodów na trasie. Dla niektórych krawędzi wyniki są niezadowalające: przykładem są tu krawędzi, które nie leża na żadnej najkrótszej ścieżce. W realnym ruchu są one oczywiście wybierane przez kierowców, ale algorytm generujący ruch ich nie uwględnia. Rozwiązaniem tego problemu mogą być próby innego rozmieszczenia bram. Na podstawie otrzymanych wyników Kraksim wygenerował schemat obciążenia ruchu odpowiadający najlepszemu osobnikowi w populacji. Opis ten znajduje się w wybranych pliku xml i posłuży on nam do ostatecznej oceny jakości wygenerowanego schematu ruchu poprzez ocenę symulacji w Kraksimie.

Eksperymenty w Kraksimie

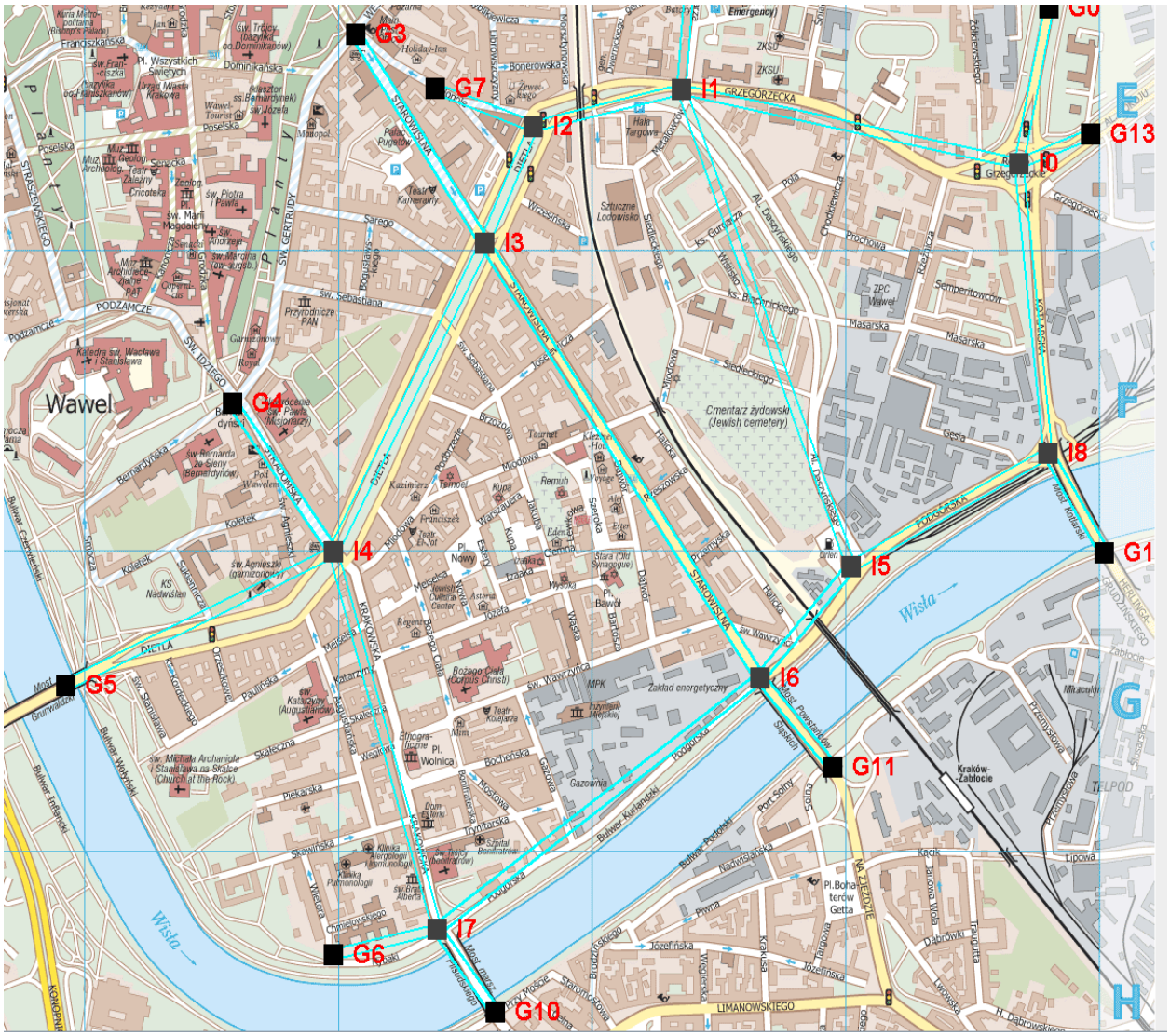
Po uzyskaniu opisanych wyżej danych doświadczalnych uruchomiliśmy wyjściowy plik (opisujący ruch) w środowisku Kraksim. Niestety okazało się, że w symulacji pojawia się więcej samochodów niż symulator jest w stanie obsłużyć i w krótkim czasie osiągamy stan, w którym Kraksim traci responsywność a symulacja nie dobiega końca. Przyczyny takiego stanu mogą być dwie. Pierwsza z nich to nieprawidłowo wygenerowane przez nasz algorytm dane. Drugą możliwością są niedokładności w symulacji, które nie pozwalają na przeprowadzanie symulacji z ilością pojazdów, jaka w rzeczywistości znajduje się na drogach.

W celu sprawdzenia przyczyn niepowodzenia przeprowadzanego eksperymentu przeprowadziliśmy kolejną próbę, w której zmniejszyliśmy wygenerowane dane dziesięciokrotnie. Jeżeli okazałoby się, że obciążenia na poszczególnych ulicach są dziesięciokrotnie mniejsze niż dane z ZDKiA. Kiedy przeprowadziliśmy taki eksperyment okazało się, że Kraksim bardzo dobrze poradził sobie z symulacją. Po porównaniu wyników okazało się, że ilość samochodów jaka przejechała poszczególnymi ulicami satysfakcjonująco przypomina dane statystyczne. Wyniki te przedstawia poniższa tabela (uwględniają one podzielenie ruchu przez 10, a wszystkie błędy zostały zaokrąglone do jedności):

Ulica	**Ruch rzeczywisty**	**Ruch w symulacji**	**Błąd względny**
I0 → I1	368	246	33,00%
I1 → I0	353	425	17,00%
I1 → I2	331	335	1,00%
I2 → I1	373	359	4,00%
I2 → I3	207	197	5,00%
I3 → I2	360	349	3,00%
I3 → I6	118	25	79,00%
I6 → I3	185	68	63,00%

I3 → I4	367	323	12,00%
I4 → I3	317	355	12,00%
I4 → I7	112	111	1,00%
I7 → I4	149	136	9,00%
I6 → I7	122	97	20,00%
I7 → I6	61	71	16,00%
		Średni błąd względny:	20,00%

Poniżej zamieszczona jest mapa oznaczona zgodnie z symbolami skrzyżowań z powyższej tabeli:



Wyniki są satysfakcjonujące ale aby móc generować wiarygodne obciążenie należy jeszcze rozwiązać kilka problemów. Okazuje się, że symulacja trwa dłużej niż piętnaście minut. Może to doprowadzić do sytuacji, że po jakimś czasie mapa ponownie będzie przepełniona, ze względu na nieuwzględnienie w algorytmie zależności między kolejnymi odcinkami czasowymi, między którymi były dokonywane pomiary.
Reasumując, nasz algorytm dał satysfakcjonujące wyniki dla przewidywania obciążenia między poszczególnymi bramami, co znalazło potwierdzenie w symulacji, jednak aby móc generować wiernie ruch na podstawie danych statystycznych należy jeszcze zmodyfikować go tak, aby uwzględniał on zależności pomiędzy kolejnymi odcinkami, na których zachodził

pomiar ruchu. Osobnym aspektem jest działanie samego Kraksima, który, jak się okazuje, nie pozwala na przeprowadzanie symulacji z realnie występującą liczbą samochodów.