Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Wydział Elektrotechniki, Automatyki, Informatyki i Inżynierii Biomedycznej

KATEDRA INFORMATYKI STOSOWANEJ

SYMULACJA DYSKRETNA SYSTEMÓW ZŁOŻONYCH

DAMIAN JURKIEWICZ
MATEUSZ WIĘCŁAWEK

SYMULACJA RUCHU SAMOCHODÓW NA I OBWODNICY KRAKOWA

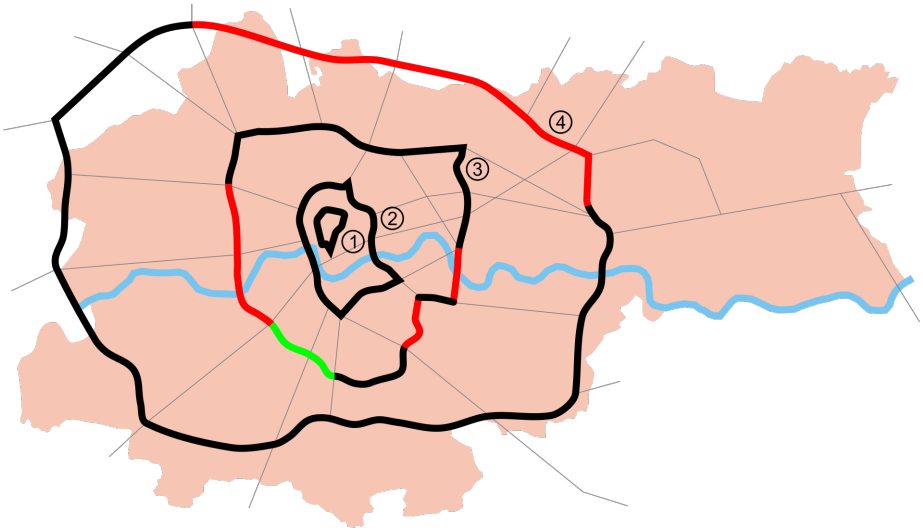
Kraków 2020

Spis treści

1. Wprowadzenie

1.1. Opis problemu

Naszym problemem jest zamodelowanie i zasymulowanie ruchu samochodów osobowych oraz ciężarowych na I obwodnicy Krakowa.



Rysunek 1.1: Obwodnice Krakowa, I obwodnica oznaczona numerem 1.

Mianem pierwszej obwodnicy określa się w Krakowie ciąg ulic otaczających Stare Miasto wzdłuż plant. Na wykaz ulic wchodzących w skład pierwszej obwodnicy wchodzą:

- ul. św. Idziego
- ul. Podzamcze
- ul. F. Straszewskiego
- ul. Podwale
- ul. J. Dunajewskiego
- ul. Basztowa
- ul. Westerplatte
- ul. św. Gertrudy

Rysunek 1.2: I obwodnica Krakowa.

Na większości wymienionych ulic obowiązują ograniczenia oraz ruch jednokierunkowy - zgodny z ruchem wskazówek zegara na rysunku 1.2.

1.2. Cel

Tworząc model skupimy się na odwzorowaniu ruchu samochodów osobowych oraz celów ich podróży. W modelu uwzględnimy wszystkie wjazdy i zjazdy z dróg składających się na I obwodnicę oraz wszelkie skrzyżowania na niej występujące wraz z ich przepustowością. Nasz model powinien uwzględniać podejście mikroskopowe w modelowaniu, czyli uwzględniać specyfikę pojedyńczego pojazdu poruszającego się po drogach jednopasmowych i wielopasmowych. Model ma uwzględniać specyfikę zmiany pasa ruchu, a także brać pod uwagę charakterystyki przyspieszenia oraz hamowania pojazdu.

2. Przegląd literatury

2.1. Model Nagela- Schreckenberga

Model Nagela- Schreckenberga jest teoretycznym modelem ruchu samochodowego opracowanym w latach 90 XX wieku przez niemieckich fizyków: Kai Nagela i Michael Schreckenberga. Zakłada on:

– przyspieszenie:

$$v(t+1) = min(v(t)+1, v_{max}),$$

– hamowanie:

$$v(t+1) = min(v(t), g(t)-1)$$

gdzie g(t)- liczba pustych komórek między pojazdami,

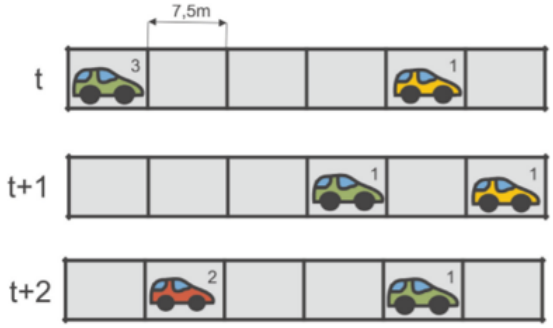
– przypadki losowe

$$v(t+1) = min(v(n)-1, 0),$$

– ruch samochodu

$$x(t+1) = x(t)+v(t).$$

Mówi też, że pojedyncza komórka powinna być wielkości 7,5 metra Modelując ruch drogowy będziemy musieli skupić się m.in. na dynamice pojazdu czyli przyspieszeniu oraz hamowaniu a także gabarytach pojazdu w zależności czy będzie to samochód osobowy czy ciężarowy. Użyjemy do tego celu modelu Nagela-–Schreckenberga, który dotyczy ruchu samochodów na autostradzie, lecz z jest także punktem odniesienia do modelowania ruchu miejskiego.



Rysunek 2.1: Model Na-Sch

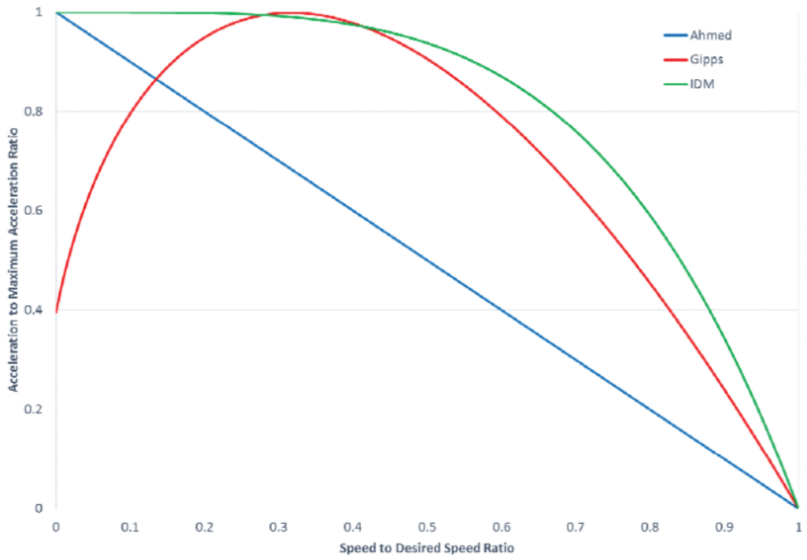
2.2. Przyspieszenie Gippsa

Jedną z głównych wad modelu Na–Sch jest fakt, iż dynamika pojedyńczego samochodu nie jest dobrze odwzorowana. Przyspiesznie pojazdu może przyjmować zbyt duże wartości, które odbiegają znacznie od rzeczywistości, a drugi krok modelu opisujący hamowanie służy tylko zapobieganiu kolizjom i zachowaniu odpowiedniego odstępu między pojazdami, a nie odwzorowuje rzeczywistego zachowania podczas zwalniania. [ii09]

Jedną z modyfikacji powyższego modelu jaką zamierzamy wprowadzić jest użycie modelu przyspieszenia Gippsa, która wprowadza większy realizm ruchu pojazdów w modelu - zastępując linearne przyspieszenie, przyspiszeniem zmiennym zależnym od prędkości. [MM18]

Ogólny wzór opisujący przyspieszenie Gippsa:

$$a_n(t+\tau) = l_n \frac{[v_{n-1}(t) - v_n(t)]^k}{[x_{n-1}(t) - x_n(t)]^m}$$



Rysunek 2.2: Wykres przyspieszenia od predkości dla poszczególnych modeli przyspieszenia.

Kolejną propozycją jest pozwolenie aby pojazd zajował więcej niż jedną komórkę. Dzięki czemu pojedyńcza komórka nie będzie musiała odpowiadać przyjętej długości $7.5m$ co pozwoli na osiągnięcie lepszego odwzorowania rzeczywistej prędkości niż w klasycznym modelu Na–Sch. Opcja ta pozwili również na wprowadzenie dróg wielopasmowych i skrzyżowań ze światłami co akurat przy modelowaniu I obowdnicy nie będzie miało zastosowania.

Należy wspomnieć, że w modelowaniu drogi będziemy posługiwać się grafem skierowanym, wierzchołki będą reprezentowały skrzyżowania na których ruch nie będzie modelowany.

3. Proponowany model

3.1. Założenia

Proponowany przez nas model zakłada utworzenie siatki automatu, gdzie pojedyncza komórka jest wielkości 5m. Takie założenie bardziej odpowiada rzeczywistej odległości w ruchu miejskim powmiędzy pojazdami. Jako że prędkość w naszym modelu jest wartością dyskretną, zmiena wielkości komórki dała większe pole manewru prędkością pojazdów. W oryginalnym modelu Na-Sch

$$1komorka/sekunde = 27km/h$$

natomiast nasza propozycja zakłada

$$1komorka/sekunde = 18km/h$$

W taki sposób prędkosc rzeczywista ma się do modelowanej:

Prędkość rzeczywista	Prędkość w naszym modelu	
$20km/h$	$18km/h$	$1kratka/s$
$30km/h$	$36km/h$	$2kratki/s$
$40km/h$	$36km/h$	$2kratki/s$
$50km/h$	$54km/h$	$3kratki/s$
$60km/h$	$54km/h$	$3kratki/s$
$70km/h$	$72km/h$	$4kratki/s$

W ten sposób różnica prędkości wynosi maksymalnie $6km/h$ co jest do przyjęcia.

Nasza symulacja opiera się na istotych założeniach:

– nie występuje zmiana pasa ruchu, na całej długości I obwodnicy jest tylko jeden pas, momentami dozwolone jest poruszanie się po torowisku, jedynie w celu skrętu w lewo, bądź jako pas rozbiegowy służący do włączenia się w ciąg dróg obwodnicy,

– sygnalizacja w ciągu ulic jest wyłączona (była sukcesywnie wyłączana od 2015 roku),

– pierwszeństwo występuje w ciągu ulic obwodnicy, wyjątkiem są skrzyżowania, gdzie tramwaj jadąc obwodnicą z niej zjeżdża. Jednakże jako, że modelowanie ruchu tramwajów wzdłóż obwodnicy nie jest tematem pracy, fakt ten został pokazany jako dodatkowe ograniczenie prędkości przed skrzyżowaniem,

– samochody dążą do maksymalnej dozwolonej prędkości na danych odcinku z założoną wcześniej tolerancją,

– wzdłóż obwodnicy nie ma rond.

3.2. Założenia algorytmów

Została zastosowana koncepcja automatów komórkowych, z których każdy może posiadać dwa stany

1. zajęty,
2. niezajęty.

```
public class Cell {
    public static double measure = 5;
    private int number;
    private int maxVelocity;
    private boolean occupied = false;
    private Pair<Long,Long> lane_id;
    private Car car;

    Cell(int maxVelocity,long begin, long end, int number){
        this.maxVelocity = maxVelocity;
        this.lane_id = new Pair<Long,Long> (begin,end);
        this.number = number;
    }

    public void setOccupied() { this.occupied=true; }

    public void setNotOccupied() { this.occupied=false; }

    public void setCar(Car car) { this.car = car; }

    public boolean getOccupied() { return occupied; }

    public void occupyCell( Car car)  {...}
    void freeCell()
    {...}

    public Car getCar() { return car; }

    public int getMaxVelocity() { return maxVelocity; }

    public void setMaxVelocity(int maxVelocity) { this.maxVelocity = maxVelocity; }
```

Rysunek 3.1: Klasa Cell

Reprezentacją jest klasa Cell która agreguje w sobie obiekt car klasy Car. Reszta informacji jest zapisywana z wykorzystaniem OpenStreetMap.

Głównym algorytmem, który realizuje ruch po siatce jest metoda simulate() klasy Lane. Jest ona wywoływana dla każdej zajętej komórki. Działanie tej metody wyraża się następująco:

1. Przeszkuje się od tyłu listy wszystkie komórki i jeżeli jest zajęta przez pojazd to:
   (a) zapisuje się do pojazdu prędkość maksymalną z komórki,
   (b) zwiększa się lub zmniejsza się prędkość pojazdu o 18 km/h jeżeli była ona różna od maksymalnej dla komórki
   (c) oblicza się ilość komórek do zmniany, która jest równa prędkości w komórkach/sekundę
   (d) jeżeli ilość jest mniejsza od 1, przerywa się pętlę i szuka następnej zajętej komórki,
   (e) jeżeli ilość komórek do przejechania jest większa niż ilość komórek pozostalych w danym odcinku, to:
      i. zmniejsza się ilość sekcji do przejechania dla pojazdu o 1,
      ii. jeżli ilość sekcji do przejechania jest równa 0, zwalnia się komórkę, a pojazd nie kontynuuje jazdy,
      iii. w przeciwnym wypadku pojazd dodaje się do kolejki wyjeżdżających ze skrzyżowania,
   (f) w przeciwnym wypadku, jeżli komórka do której chcemy jechać jest zajęta, zmniejszamy prędkość o 18km/h i wracamy do punktu c.
   (g) jeżeli komórka do której chcemy jechać jest wolna, to zapisujemy w niej pojazd, a dotychczasową komórkę zwalniamy.
2. Sprawdzana jest zawartość kolejki pojazdów chcących wjechać na skrzyżowanie, jeżeli jest niepusta, to jeżli pierwsza komórka jest wolna, pojazd ją zajmuje

Ta metoda jest wykonywana dla wszystkich odcinków.

4. Technologie, środowisko pracy, plan pracy

4.1. Technologie i środowisko pracy

Zasadniczno byliśmy zgodni co do poziomu języka. Uznaliśmy, że model nie musi być na tyle szczegółowy aby używać języka niższego poziomu. Paradygmat programowania obiektowego również wydawał nam się odpowiednim do tego projektu. Stąd rozpatrywaliśmy wykorzystanie jednego z dwóch języków programowania, Java lub Python. Zdecydowaliśmy się jednak na użycie **Javy**, głównie ze względów technicznych oraz z uwagi na fakt, że ten język jest najlepiej przez nas znanym i najczęściej używanym. Nie znaleźliśmy na tyle istotnych argumentów za innymi językami aby je wdrożyć. Z tego też względu IDE na którym pracowaliśmy to **IntelliJ IDEA** produkt od JetBrains.

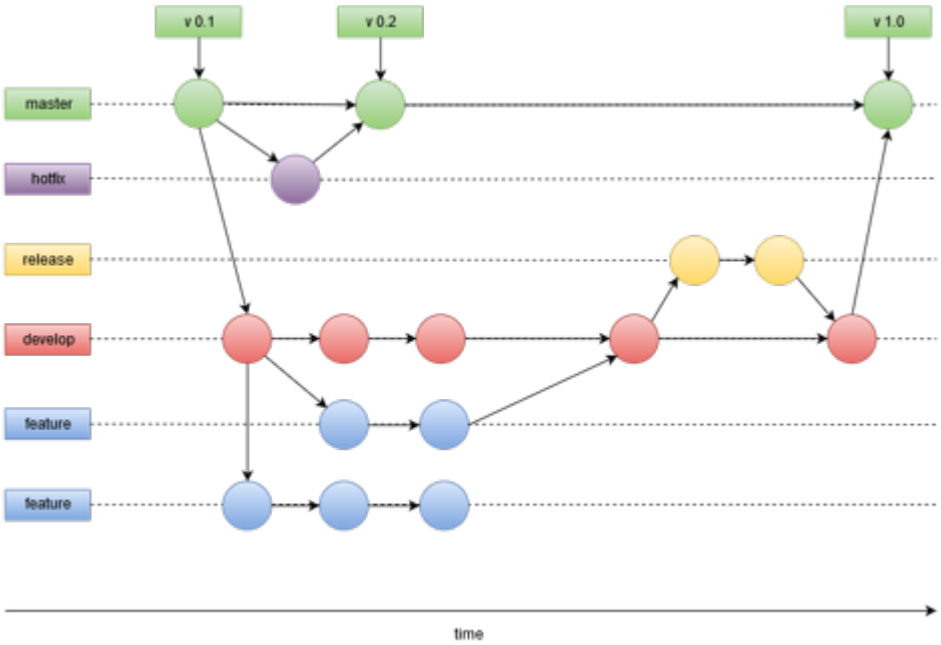
Kolejnym istotnym wyborem był wybór biblioteki graficznej. W tym przypadku zastanawialiśmy się pomiędzy JavaFX, a AWT/Swing. Postanowiliśmy wykorzystać **AWT/Swing**. Powodem jest fakt, że mimo iż GUI napisane przy użyciu JavaFX jest zwykle bardziej estetyczne i ładne, to w tym przypadku uznaliśmy to za drugorzędną sprawę. Fakt, że AWT/Swing jest prostsze w obsłudze, GUI tworzy się łatwiej i szybciej oraz jest bardziej intuicyjne, zadecydował o wyborze.

```
private SimulatePanel simpanel = new SimulatePanel( height: 700,  width: 630);

void buildGui() {
    JPanel panel = new JPanel();
    panel.setLayout(new BorderLayout());
    JPanel opt = new JPanel();
    JButton start = new JButton( text: "Start");
    JButton stop = new JButton( text: "Stop");
    JButton plus = new JButton( text: "Plus");
    JButton next=new JButton( text: "Next Section");
    JButton previous=new JButton( text: "Previous Section");
    next.addActionListener(p->{simpanel.onNext();});
    previous.addActionListener(p->{simpanel.onPrevious();});
    opt.add(next);
    opt.add(previous);
    plus.addActionListener(p-> {
        try {
            simpanel.onPlus();
        } catch (Exception e) {
            e.printStackTrace();
        }
    });
    opt.add(plus);


    start.addActionListener(p -> simpanel.onStart());


    start.addActionListener(p -> { start.setEnabled(false);stop.setEnabled(true); });
```

Rysunek 4.1: Metoda buildGui() klasy SimulateFrame

4.2. Plan pracy

Repozytorium z projektem znajduje się na GitHubie pod adresem github.com/MrRedonis/DSCS-Project Przy realizacji projektu wykorzystalismy model pracy Gitflow Workflow. Prace były prowadzone



Rysunek 4.2: Model pracy Gitflow Workflow

w gałęzi Develop, a część na dev/Damian, dev2. Ostateczna działająca wersja została umieszczona w gałęzi master.

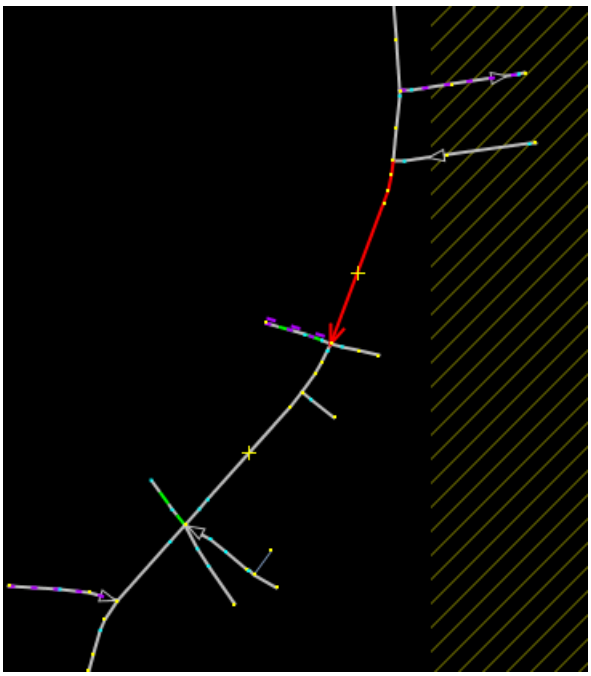
5. Implementacja

5.1. Pierwsza wersja

5.1.1. Model

LaneSection

Pierwotna wersja modelu wyglądała następująco. Poza wspomnianą wcześniej klasą Cell, klasa LaneSection była klasą, której obiektem był odcinek od skrzyżowania do skrzyżowania. Takie podejście daje gwarancję, że pojazd musi przejechać cały odcinek, ponieważ po drodze nie napotka żadnego skrzyżowania. Klasa agregowała komórki automatu w ArrayLiscie lane. Klasa zawierała też dwie kolejki



Rysunek 5.1: Przykład odcinka który może być obiektem klasy LaneSection

FIFO:

- waitingCar- pojazdy które chcą wjechać do sekcji,
- outOfSection- pojazdy które wyjechały z sekcji.

```
public class LaneSection {
    ArrayList<Cell> lane;
    Boolean islimited;
    String label;
    Queue<Car> waitingCar;
    Queue<Car> outOfSection;

    public ArrayList<Cell> getLane() { return lane; }


    public LaneSection(int from, int to, String label, boolean islimited) {
        this.lane = new ArrayList<>();
        this.waitingCar = new ArrayDeque<>();
        this.outOfSection=new ArrayDeque<>();
        this.islimited = islimited;
        this.label = label;
        for (int i = from; i < to; i++) {
            //lane.add(new Cell(i));
            lane.add(new Cell(i));
        }
    }
    public LaneSection(ArrayList<Cell> cells,String label, boolean islimited)
    {
        this.lane=cells;
        this.waitingCar=new ArrayDeque<>();
        this.outOfSection=new ArrayDeque<>();
        this.islimited = islimited;
        this.label = label;
    }
```

```
    public void simulate() throws Exception {
        Boolean cond = true;

        for (int i = lane.size() - 1; i >= 0; i--) //dla każdej komórki
        {
            if (lane.get(i).getOccupied()) {//jeżeli jest zajęta
                if(lane.get(i).car.setmax)
                lane.get(i).car.maxvelocity=lane.get(i).getMaxVelocity();
                lane.get(i).car.setmax=true;

                cond = true;
                toMaxVelocity(i);
               // if(lane.get(i).car.getDistanceToNextCarInFront()<3)lane.get(i).car.decreaseVelocity(18);
                while (cond) {
                    double plus = (lane.get(i).car.getVelocity()) / 18;// zmiana predkosci z km/h na kratki/s
                    if (plus < 1)//jeżeli jest mniejsza niż 1 nie zmieniaj pozycji pojazdu; zakładamy minimalną predkosc 18 km/h
                        break;
                    if(i+(int)plus>(lane.size()-1)) { //pojazd wyjeżdża z sekcji
                        lane.get(i).car.decreaseSection();
                        if(lane.get(i).car.getNumberOfSectionToPass()==0)
                        {
                        lane.get(i).freeCell();
                       cond=false;
                        }
                        else {
                        if(lane.get(i).car.getNumberOfSectionToPass()!=0)
                            outOfSection.add(lane.get(i).car);
                            lane.get(i).freeCell();
                            cond = false;
                        }
                    }
```

```
                else if (lane.get(i + (int) plus).getOccupied()) {//jeżeli komorka do ktroej chce pojechac jest zajeta
                    lane.get(i).car.decreaseVelocity( velocityChange: 18); //zmniejsz predkosc o 1 komorke/s

                } else {//zmien komorke

                    lane.get(i + (int) plus).occupyCell(lane.get(i).car); //zajmij komorke
                    lane.get(i).freeCell();// zwolnij poprzednia
                    lane.get(i+(int)plus).car.setDistanceToNextCarInFront(toNextCar( index: i+(int)plus));

                    lane.get(i+(int)plus).car.maxvelocity=lane.get(i+(int)plus).getMaxVelocity();
                    cond = false;//nie powtarzaj
                }

            }

        }
    }

    if (!waitingCar.isEmpty()) {
        if (!lane.get(0).getOccupied()) {
            addCar(waitingCar.poll());
        }
    }
}
```

Rysunek 5.2: Klasa LaneSection

Lane

Obiekt klasy Lane był zamkniętą jednokierunkową jezdnią. Składał się z odcinków LaneSection, które były przechowywane w jednowymiarowej tablicy. I obwodnicy Krakowa nie da się objechać w żadnym kierunku bez zjeżdżania z niej, stąd też, gdy jeden z odcinków nie będzie mieć kontynuacji, w tablicy pojawi się null. Pole isInterconnected mówi, czy po ostatnim odcinku jest null, czy pierwszy odcinek, innymi słowy, czy jezdnia jest domknięta.

Rysunek 5.3: Klasa Lane

Symulacja

SimulateFrame

Klasa SimulateFrame dziedziczy po JFrame. Umożliwia ona utworzenie okienka, składają się z paneli. Panel symulacujny zostanie utworzony jako obiekt klasy SimulatePanel dziedziczący po JPanel.

SimulatePanel

Ta klasa tworzy panel z symulacją. Metoda rysująca panel nazywa się PaintComponent()

Klasa zawiera też klase wewnętrzną AnimationThread, która dziedziczy po Thread. Ta klasa jest odpowiedzialna za wątek związany z animacją. Metoda run() w nieskończonej pętli wywołuje:

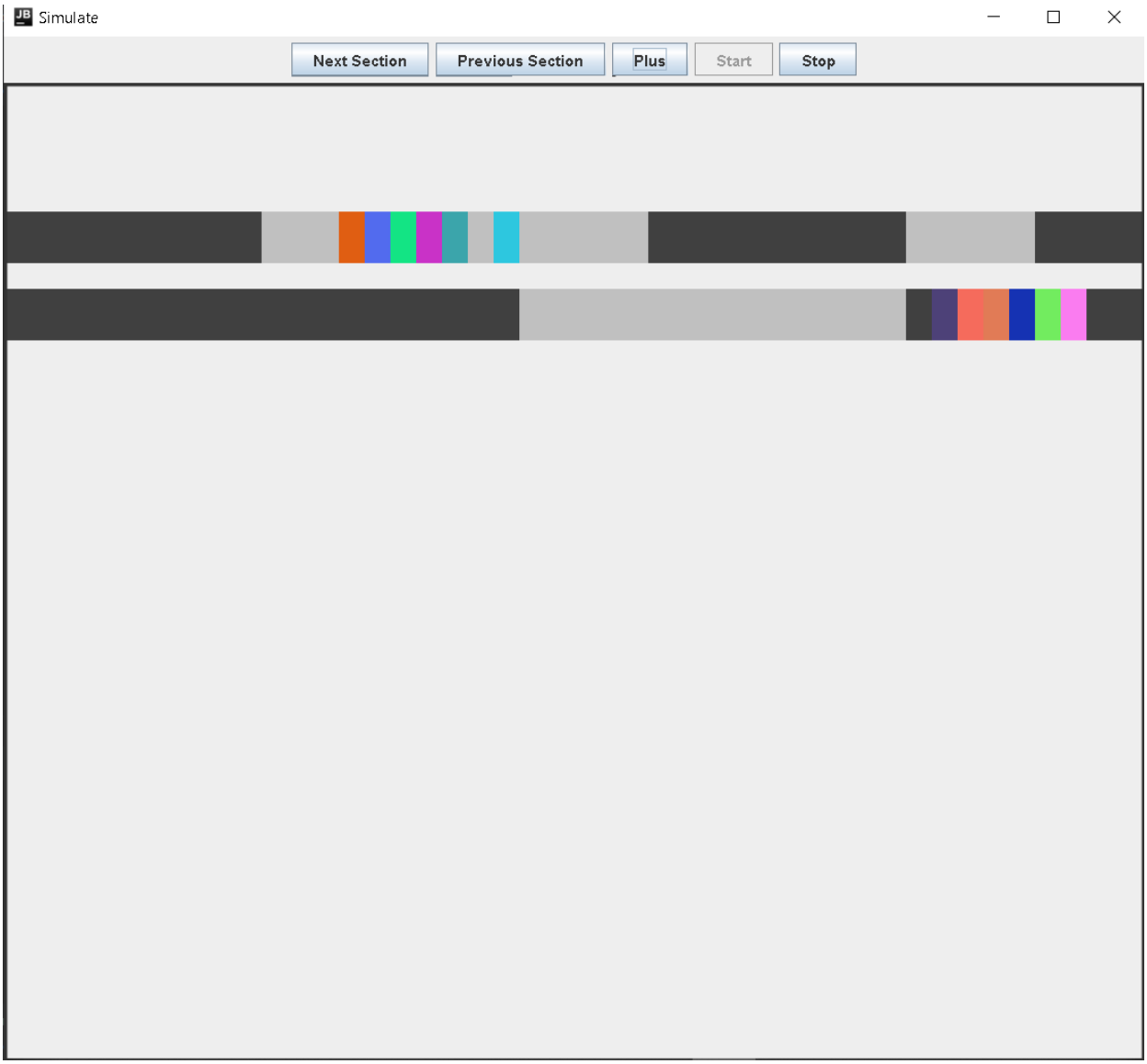
– metode simulate(),

– metodę PaintComponent(),

– próbuje uśpić wątek na 1 sekundę

Rysunek 5.4: PaintComponent()

Pierwsza wersja programu, w celu weryfikacji modelu wyglądała następująco: Odcienie szarego oznaczają niezajętą komórkę automatu, natomiast kolorowy zajętą. Kolor jest przechowywany w obiekcie klasy Car.



Rysunek 5.5: Pierwsza wersja

5.1.2. Mapa

Aby jak najwierniej odwzorować parametry dróg takie jak długość, prędkość maksymalna czy nazwa, wykorzystaliśmy Polski Format Mapy. W tym celu niezbędne było utworzenie klasy ParseOSM

ParseOSM

Klasa ta otwiera w konstruktorze plik .osm i zapisuje do następujących struktur:

KnotList kn_lst	listę węzłów
WayList w_lst	listę dróg
LaneList l_lst	listę odcinków

Komórki automatu są zapisane natomiast w klasie Lane jako lista macierzowa, a obiekty tej klasy są agregowane w klasie LaneList.

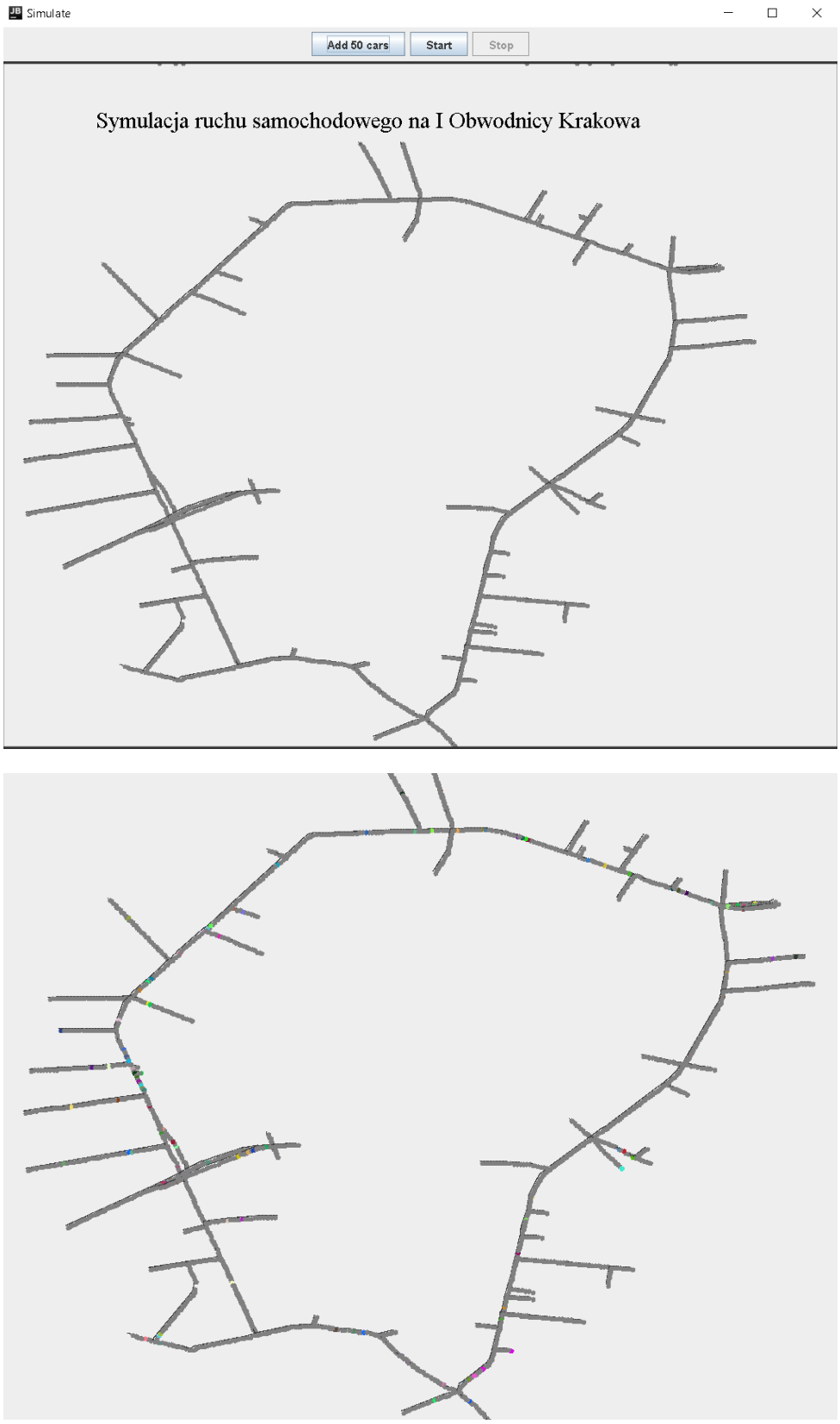
Rysunek 5.6: map.osm

5.2. Zmiany wprowadzone w ostatecznej wersji

Po udanym zrzutowaniu mapy na siatkę automatów komórkowych należało zmodyfikować metodę PaintComponent() klasy SimulatePanel tak, aby uwzględniała ona położenie siatki na ekranie, co w poprzedniej wersji nie miało miejsca. Zmianie uległo też założenie klas Lane, Way oraz Knot. Ostateczna ramka symulacji wyglądała tak:

```
public void paintComponent(Graphics g) {
    System.out.println("Repaint");
    super.paintComponent(g);
    Random rand = new Random();

    Graphics2D g2 = (Graphics2D) g;

    g2.setColor(Color.BLACK);
    g2.setFont(new Font( name: "Serif", Font.PLAIN,  size: 24));
    g2.drawString( s: "Symulacja ruchu samochodowego na I Obwodnicy Krakowa", i: 100, i1: 70);
    for (Map.Entry<Pair<Long, Long>, Lane> entry : data.getLaneList().getLanes().entrySet()) {
        g2.setColor(Color.darkGray);
        Point begin = new Point((int)((entry.getValue().getBegin().getLon()*size-lon*size)),(int)((entry.getValue().getBegin().getLat()*size-lat*size)));
        Point end = new Point((int)((entry.getValue().getEnd().getLon()*size-lon*size)),(int)((entry.getValue().getEnd().getLat()*size-lat*size)));
        g2.drawLine( i: begin.x+odstep, i1: height-begin.y, i2: end.x+odstep, i3: height-end.y);
        double retA = retA(begin,end);
        double linLen = lineLength(entry.getValue())*size;
        double ss = linLen/entry.getValue().getCells().size();
        for (int i = 0; i < entry.getValue().getCells().size(); i++) {
            if (entry.getValue().getCells().get(i).getOccupied()){
                g2.setColor(entry.getValue().getCells().get(i).getCar().color);
                g2.fillOval((int)(odstep+begin.x-ss*i),(int)(height-(begin.y-retA*i*ss)), i2: 5, i3: 5);
            }
            else {
                g2.setColor(Color.gray);
                g2.fillOval((int)(odstep+begin.x-ss*i),(int)(height-(begin.y-retA*i*ss)), i2: 5, i3: 5);
            }
        }
    }
```

Rysunek 5.7: PaintComponent()



Rysunek 5.8: Okienko symulacji

6. Wnioski

Opracowany przez nas model, jak i symulacja dostarczają zadowalający efekt. Podczas tworzenia modelu i wprowadzaniu kolejnych założeń, można było zauważyć znaczną zmianę w płynności ruchu pojazdów. Płynności, mając na myśli mniejszą tendecję do powstawania zatorów drogowych. Jednym z głównych powodów ich powstawania, jest założenie, że pojazdy co jakiś czas muszą zmiejszyć prędkość, czyli wprowadzenie do modelu sytuacji losowych. Takimi sytuacjami mogą być np. wtargnięcie pieszego lub nieuwaga kierowcy. Przed wprowadzeniem tej funkcjonalności, pojazdy zmieniały swoje położenie bardzo ściywno", opierając się jedynie na swojej prędkości. Po wprowadzeniu natomiast, można było zauważyć małą tendencję do korkowania się obwodnicy. Model nie uwzględnia w stopniu wystarczającym czynnika ludzkiego, czyli czasu reakcji, czy niedoskonałości w podejmowaniu szybkich decyzji, a to jest kluczowe w tego typu symulacji.

Bibliografia

[ii09] Jarosław Wąs i inni. *Problematyka modelowania ruchu miejskiego z wykorzystaniem automatów komórkowych*. AGH, Krakow, 2009.

[MM18] Georgios Fontaras Michail Makridis, Tomer Toledo. Capability of current car-following models to reproduce vehicle free-flow acceleration dynamics. *IEEE*, 19:3594 – 3603, 2018.