AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
WYDZIAŁ INFORMATYKI, ELEKTRONIKI I TELEKOMUNIKACJI
INSTYTUT INFORMATYKI

Projekt dyplomowy

Środowisko do modelowania i optymalizacji ruchu drogowego

Environment for modeling and optimalisation of traffic

Autorzy: Wiktor Kamiński, Grzegorz Poręba, Miłosz Galas
Kierunek studiów: Informatyka
Opiekun pracy: prof. AGH, dr hab. inż. Rafał Dreżewski

Kraków, 2022

Spis treści

1. Cel prac i wizja produktu

1.1. Wizja

W dzisiejszych czasach transport lądowy jest wszechobecny w ludzkiej rzeczywistości. Jego zakres rozciąga się od samochodów i ciężarówek dostawczych, będących ostatnim elementem globalnego łańcucha dostaw, po prywatne samochody, których używamy w codziennych czynnościach, aby dojechać do pracy, czy na zakupy. Podczas epidemii COVID-19 byliśmy świadkami dynamicznego rozwoju rynku e-commerce [52] oraz zmiany modelu funkcjonowania restauracji na praktycznie w całości oparty o zamówienia robione na wynos [50]. Rozwój tych sektorów spowodował zwiększone zapotrzebowanie usługi kurierów, którzy poruszają się głównie za pomocą samochodów osobowych i dostawczych. Jeszcze przed pandemią mogliśmy obserwować rewolucję na rynku taksówkarskim, wywołaną powstaniem takich firm jak Uber, Lyft, czy Bolt. Przytoczone przykłady świadczą o wystąpieniu drastycznych zmian w strukturze ruchu drogowego. Wymagają one ulepszenia infrastruktury drogowej, by była ona w stanie sprostać nowym wymaganiom. Skutki źle zaprojektowanych dróg odczuwamy każdego dnia: hałas, zanieczyszczenie powietrza, korki, wypadki itd. Do skutecznego projektowania systemów drogowych potrzebne są narzędzia, które umożliwią symulowanie ruchu na podstawie projektu sieci drogowej. Naszym celem jest stworzenie rozszerzalnego oraz konfigurowalnego środowiska, które umożliwi przeprowadzanie symulacji ruchu drogowego za pomocą rozmaitych modeli oraz monitorowanie jego parametrów i jakości. System ma również pozwalać na: porównywanie różnych modeli ruchu oraz sterowania sygnalizacją świetlną, swobodną zmianę owych modeli, ocenę stanu ruchu za pomocą parametrów takich jak gęstość ruchu, czy średnia prędkość.

1.2. Produkt

Kraksim [51] to aplikacja desktopowa, będąca rozszerzalnym środowiskiem symulacyjnym do modelowania ruchu drogowego. Składa się ona z modeli ruchu (makroskopowe, mikroskopowe), algorytmów sterowania sygnalizacją świetlną (rozwiązania heurystyczne oraz oparte na uczeniu maszynowym), metod opisu jakości ruchu (parametry typu średnia prędkość, gęstość, płynność) oraz interfejsu graficznego pozwalającego na obserwację, oraz zmianę parametrów symulacji. Była ona używana przez studentów naszej uczelni do badań i analizy ruchu drogowego; rozwijana na przestrzeni lat. Z powodu długu technologicznego rozwijanie oraz używanie jej jest coraz trudniejsze, dlatego nasza praca będzie miała na celu stworzenie współczesnej wersji analogicznej aplikacji w formie aplikacji webowej, co zapewni jej większą dostępność. Ważne jest dla nas to, żeby powstała aplikacja mogła być w łatwy sposób dalej rozwijana przez innych studentów oraz pracowników wydziału.

2. Zakres funkcjonalności

Zakres funkcjonalności produktu podzieliliśmy na 4 podstawowe sekcje:

1. Modele ruchu.
2. Sterowanie sygnalizacją świetlną.
3. Parametry opisu jakości ruchu.

4. Interfejs graficzny.

Stanowią one rozłączne segmenty logiczne aplikacji, które powinny być ze sobą jak najmniej związane, aby umożliwić wysoką konfigurowalność systemu.

2.1. Modele ruchu

Modele ruchu stanowią fundament środowiska symulacyjnego. Określają, według jakich reguł funkcjonuje ruch samochodów podczas symulacji. Jako że poszczególne modele lepiej symulują daną sytuację na drodze (np. korki, czy ruchu na autostradzie), to do skutecznego działania naszej aplikacji konieczna jest implementacja kilku modeli ruchu oraz umożliwienie użytkownikowi wyboru, z którego w danej chwili chce skorzystać.

2.1.1. Makroskopowe

Modele makroskopowe opisują stan ruchu za pomocą pewnych globalnych wartości, takich jak średnia prędkość, gęstość ruchu, czy przepływ. W ich przypadku nie jest konieczne operowanie na stanie pojedynczego pojazdu – całość ruchu traktujemy jak strumień. Dzięki temu modele makroskopowe są zdecydowanie mniej wymagające obliczeniowo niż modele mikroskopowe.

Greenshielda

Model Greenshielda [45] jest prostym modelem makroskopowym, za pomocą którego można otrzymać akceptowalne rezultaty. Głównym założeniem modelu jest liniowa zależność między średnią prędkością a gęstością:

$$v = v_f - \left[\frac{v_f}{k_j}\right] \cdot k$$

gdzie v i k to obecna średnia prędkość i gęstość, v_f prędkość wolnego przepływu, k_j maksymalna gęstość korka. Wraz z dążeniem gęstości do 0, prędkość dąży do v_f. Mając przepływ dany wzorem

$$q = vk$$

możemy wyprowadzić zależność zarówno między przepływem a gęstością, podstawiając prędkość do poprzedniego równania:

$$q = v_f k - \left[\frac{v_f}{k_j}\right] \cdot k^2$$

jak i analogicznie relacje między przepływem a średnią prędkością:

$$q = k_j v - \left[\frac{k_j}{v_f}\right] \cdot v^2$$

Jako że obydwa równania mają postać paraboliczną, to w łatwy sposób możemy wyliczyć wartości ekstremalne, prędkość i gęstość przy maksymalnym przepływie:

$$v_0 = \frac{v_f}{2}$$

$$k_0 = \frac{k_j}{2}$$

Z powodu swojej prostoty model Greenshielda jest świetnym kandydatem do zaimplementowania w pierwszej kolejności i przetestowania, jak modele makroskopowe działają w porównaniu z mikroskopowymi opartymi na modelu Nagela-Schreckenberga.

2.1.2. Mikroskopowe

Modele mikroskopowe, w odróżnieniu od makroskopowych, biorą pod uwagę każdy element symulowanego systemu. W naszym przypadku są to samochody, z których każdy ma własny stan. Samochód na postawie własnego stanu, otaczających go pojazdów, a także elementów losowych, podejmuje akcję ruchu. Zależność ta powoduje, że zmiana zachodząca w jednym z elementów systemu może znacząco wpłynąć na całościowy stan ruchu. Ze względu na tę właściwość modele te umożliwiają bardzo dokładne badanie własności ruchu, również na małych, lokalnych fragmentach tras. Jednak z powodu konieczności symulowania każdego elementu osobno, wymagają one dużej mocy obliczeniowej. Większość modeli mikroskopowych zaimplementowanych w naszej aplikacji przyjmie postać automatu komórkowego – droga będzie składała się z pól (komórek), w których przebywać będą auta, aktualizujące co turę swoją pozycję w zależności od danego modelu.

Nagel-Schreckenberg

Najbardziej znanym modelem do symulacji ruchu jest model Nagela-Schreckenberga [47]. Ma on postać prostego automatu komórkowego, który składa się z faz odpowiadających danym zachowaniom kierowcy podczas prowadzenia pojazdu. Model przewiduje następujące fazy:

1. Przyśpieszenie

   Wszystkie samochody, które jadą poniżej prędkości maksymalnej, zwiększają swoją prędkość o 1.

2. Hamowanie

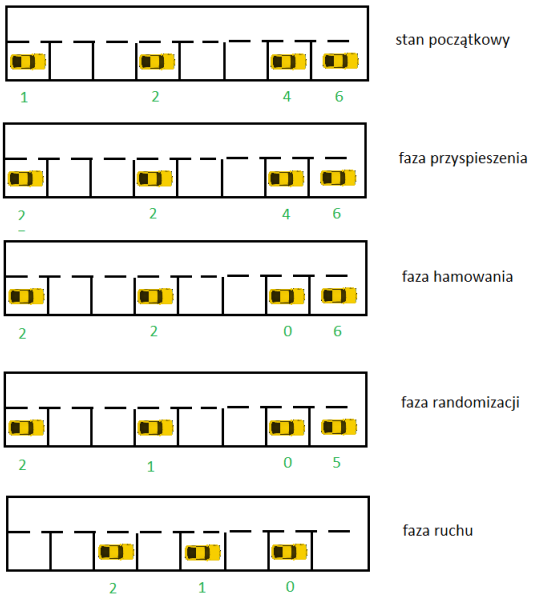
   Jeżeli dystans od auta naprzeciwko jest mniejszy niż obecna prędkość, to wartość prędkości jest ustawiana na równą tej odległości, w celu uniknięcia kolizji.

3. Randomizacja

   Prędkość wszystkich aut, które się poruszają, jest zmniejszana o 1 z prawdopodobieństwem p.

4. Ruch samochodu

   Wszystkie auta przemieszczają się o tyle komórek do przodu, ile wynosi wartość ich prędkości.



Rysunek 1: Wizualizacja faz modelu Nagela-Schreckenberga.

Velocity Dependent Randomisation

Model Velocity Dependent Randomisation [43] jest modyfikacją modelu Nagela-Schreckenberga. Wprowadza ona zasadę powolnego startu. W tym modelu prawdopodobieństwo zmniejszenia prędkości samochodu w fazie randomizacji jest zależne od jego aktualnej prędkości. Jeżeli samochód stoi, to będzie przyśpieszał wolniej, niż gdyby się poruszał. Zachowanie to umożliwia bardziej realistyczne symulowanie tzw. szerokich korków. Prawdopodobieństwo zmniejszenia prędkości w fazie randomizacji dane jest następującym wzorem:

$$p(v) = \begin{cases} p_0 & \text{kiedy } v = 0 \\ p & \text{kiedy } v > 0 \end{cases}$$

Podany wzór powoduje pojawienie się reguły powolnego startu wtedy, kiedy $p_0 > p$

Brake Light

Model Brake Light również jest modyfikacją modelu Nagela-Schreckenberga. W modelu tym kierowcy dostosowują swoją prędkość według obserwacji zachowania auta przed nimi. W tym celu wprowadzona jest dodatkowa faza, która jest wykonywana na początku każdej tury. W fazie tej:

- przy dużej odległości pojazdy poruszają się z maksymalną prędkością,
- przy średniej odległości pojazdy reagują na zmiany prędkości poprzedzających ich pojazdów (tj. do świateł stopu),
- przy małych odległościach pojazdy dostosowują swoją prędkość tak, aby bezpieczna jazda była możliwa,
- dodatkowo jeśli pojazd rusza lub zostało wymuszone jego hamowanie, przyśpieszenie pojazdu jest opóźnione.

$$p = p(v_n(t), b_{n+1}(t)) = \begin{cases} p_b & \text{jeśli } b_{n+1} = 1 \text{ oraz } t_h < t_s \\ p_0 & \text{jeśli } v_n = 0 \\ p_d & \text{w przeciwnym wypadku} \end{cases}$$

b_{n+1}

Wielopasmowy Nagel-Schreckenberg

Podstawowy model Nagel-Schreckenberg dobrze symuluje ruch dla jednopasmowej drogi, na której ruch odbywa się w jednym kierunku. Jednakże w rzeczywistości częściej mamy do czynienia z drogami wielopasmowymi. Aby uwzględnić ten rodzaj drogi, stworzono wielopasmowy model Nagela-Schreckenberga. Dodaje on wiele pasów wchodzących w skład jednej drogi, które mogą zaczynać się i kończyć w ustalonych miejscach na drodze (na przykład pas do skrętu w lewo przed skrzyżowaniem). Definiuje również zasady zmiany pasów tak, żeby auta nie wymuszały pierwszeństwa.

Duży wkład w stworzenie tego modelu mają autorzy systemu TRANSIMS [48]. Przed wykonaniem faz podstawowego modelu dodawana jest dodatkowa faza odpowiadająca za zmianę pasa. W fazie tej auta są oznaczane jako gotowe do zmiany pasa na podstawie prędkości i pozycji aut na obecnym i docelowym pasie, Następnie wszystkie auta przesuwane są równocześnie w bok, na inny pas. Za ruch tych aut w przód odpowiada faza ruchu z podstawowego modelu.

Rozróżniane są 2 główne powody zmiany pasa przez auto:

- Zmiana pasa w celu wyprzedzenia auta

  Decyzja o zmianie jest zależna od algorytmu 1.

Algorytm 1 Zmiana pasa w celu wyprzedzenia [49]

if pozycja $x_0(i)$ na sąsiednim pasie jest wolna **then**
  $gap(i) \leftarrow$ odległość od następnego auta na obecnym pasie
  $gap_f(i) \leftarrow$ odległość od następnego auta na pasie docelowym
  $gap_b(i) \leftarrow$ odległość od poprzedniego auta na pasie docelowym
  if $gap(i) < v(i)$ & $gap_f(i) > gap(i)$ **then**
    $weight1 \leftarrow 1$
  else
    $weight1 \leftarrow 0$
  end if
  $weight2 \leftarrow v(i) - gap_f(i)$
  $weight3 \leftarrow v_{max}(i) - gap_b(i)$
  if $weight1 > weight2$ & $weight1 > weight3$ **then**
    zaznacz pojazd do zmiany pasa
  end if
end if

- Zmiana pasa w celu umożliwienia skrętu na docelową drogę

  Auta podążają wzdłuż zaplanowanej trasy, co oznacza, że zbliżając się do skrzyżowania, muszą zająć odpowiedni pas, który prowadzi do docelowej drogi. W celu osiągnięcia tego zachowania do algorytmu 1 dodana została modyfikacja, która wraz ze zbliżaniem się auta do skrzyżowania, chętniej oznacza je do zmiany na wymagany pas. Zachowanie to przedstawione jest w algorytmie 2.

Algorytm 2 Zmiana pasa w celu ustawienia przed skrzyżowaniem [49]

if pozycja $x_0(i)$ na sąsiednim pasie jest wolna **then**
  $gap(i) \leftarrow$ odległość od następnego auta na obecnym pasie
  $gap_f(i) \leftarrow$ odległość od następnego auta na pasie docelowym
  $gap_b(i) \leftarrow$ odległość od poprzedniego auta na pasie docelowym
  if $gap(i) < v(i)$ & $gap_f(i) > gap(i)$ **then**
    $weight1 \leftarrow 1$
  else
    $weight1 \leftarrow 0$
  end if
  $weight2 \leftarrow v(i) - gap_f(i)$
  $weight3 \leftarrow v_{max}(i) - gap_b(i)$
  $weight4 \leftarrow \max\left((d^{ch} - d)/V_{max}, 0\right)$
  if $weight1 + weight4 > weight2$ & $weight1 + weight4 > weight3$ **then**
    zaznacz pojazd do zmiany pasa
  end if
end if

Przy 3 i więcej pasach zmiany pasów mogą powodować kolizje. Przykładowo, na trzypasmowej drodze 2 auta ustawione kolejno na lewym i prawym pasie mogą zdecydować się na zajęcie tego samego miejsca na środkowym pasie. Aby zapobiec kolizjom, wprowadzony jest warunek definiujący możliwość zmiany w zależności od tury: tj. jeżeli numer tury jest parzysty, to auta w fazie zmiany mogą zmienić pas tylko na prawy, a w przypadku tury nieparzystej, tylko na lewy.

2.2. Sterowanie sygnalizacją świetlną

Sygnalizacja świetlna pełni kluczową rolę w ruchu drogowym. Dzięki doborowi odpowiednich algorytmów oraz ich parametrów można poprawić jakość ruchu, a także znacząco zwiększyć komfort pieszych i kierowców, ograniczając czas oczekiwania na czerwonym świetle. Konieczne jest również zaimplementowanie kilku algorytmów, aby użytkownicy systemu mogli porównywać ich skuteczność oraz wybrać odpowiedni dla danej sytuacji.

2.2.1. Algorytm stałych faz

Algorytm stałych faz to najprostszy możliwy algorytm, często spotykany w życiu codziennym. Czas trwania każdego światła jest ściśle zdefiniowany, a zmiany następują w nieskończonym cyklu. Mimo swojej prostoty może być użyteczny do symulowania rzeczywistego ruchu z pewnego odcinka sieci drogowej lub porównywania działania innych, bardziej skomplikowanych algorytmów

2.2.2. Algorytm SOTL

Algorytm SOTL (Self Organising Traffic Light) [44] jest algorytmem, który zmienia światła w zależności od aktualnego stanu ruchu na drodze. W tym algorytmie światło pozostaje czerwone, dopóki spełniony jest warunek:

$$c \cdot t < \Phi$$

Gdzie c to liczba pojazdów, które znajdują się na pasie z obecnie rozważanym światłem, t to czas palenia się czerwonego światła, a Φ to ustalona wartość. Ustawiamy czas trwania zielonego światła tak, aby każdy samochód czekający na poprzednim czerwonym świetle zdążył przejechać. W celu uniknięcia zbyt częstej zmiany świateł czas trwania jednej fazy będzie wynosić minimalnie Φ_{min}.

W celu uniknięcia znaczących zaburzeń płynności wprowadza się jeszcze dwa warunki [46]:

1. Jeżeli na prostopadłej ulicy istnieje co najmniej jeden pojazd w odległości mniejszej bądź równej ω_{min} od skrzyżowania, to omijamy zmianę światła na zielone,

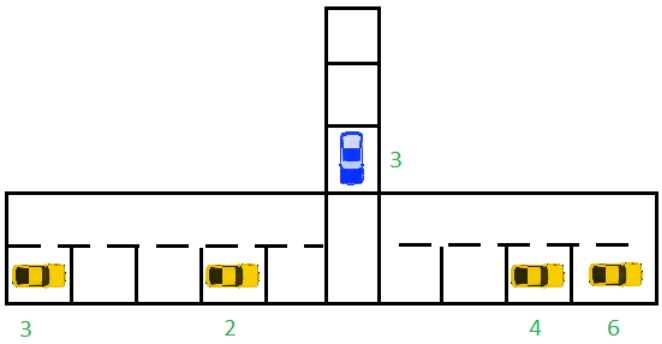
2. Jeżeli do światła na prostopadłej drodze zbliża się więcej niż μ samochodów, to pomijamy warunek 1) i zmieniamy światło.

Wartości ω_{min} oraz μ są ustalane przez osobę przeprowadzającą symulację.

2.3. Nasze modyfikacje

Nagel-Schreckenberg

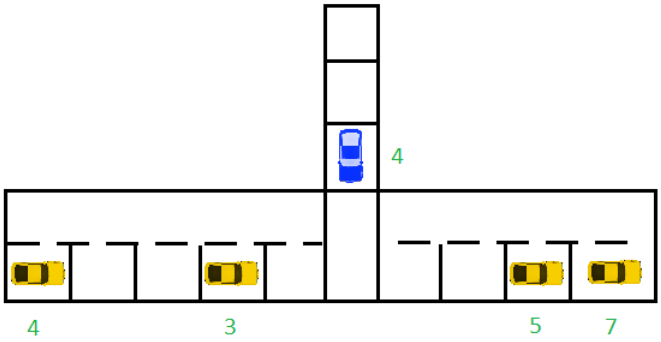
Model ten wskazuje nam zachowanie aut na drodze, jednakże nie przewiduje zachowania związanego ze skrzyżowaniami. W tym celu zastosowaliśmy autorską modyfikację, która zmienia zachowanie fazy 'Hamowania', 'Ruchu samochodu' i dodaje nową fazę końcową o nazwie 'Rozwiązanie skrzyżowania'. W tych fazach, auta dzielimy na dwie grupy: auto najbliżej skrzyżowania oraz pozostałe samochody. Finalnie otrzymujemy następujące fazy:



Rysunek 2: Wizualizacja stanu wyjściowego. Wartości przy autach symbolizują prędkości.

1. Przyśpieszenie

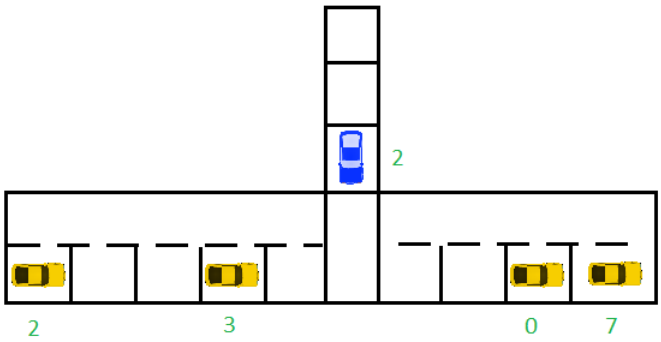
   Wszystkie samochody, które jadą poniżej prędkości maksymalnej, zwiększają swoją prędkość o 1.



Rysunek 3: Wizualizacja stanu po przyspieszeniu. Wartości przy autach symbolizują prędkości.

2. Hamowanie

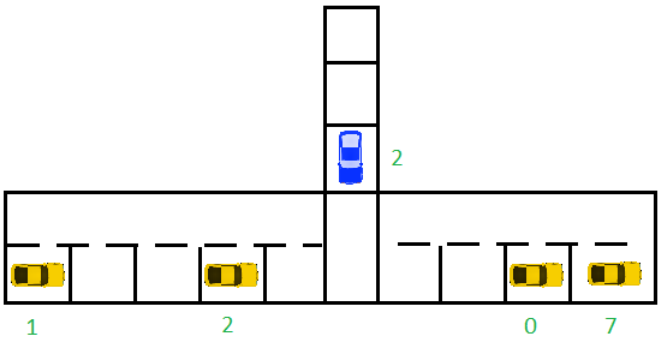
   Jeżeli dystans od auta naprzeciwko jest mniejszy niż obecna prędkość, to wartość prędkości jest ustawiana na równą tej odległości, w celu uniknięcia kolizji. W przypadku auta najbliżej skrzyżowania brana jest odległość od skrzyżowania, jeżeli jest na tym pasie czerwone światło, natomiast w przypadku zielonego, suma odległości od skrzyżowania i wolnego miejsca na docelowym pasie.



Rysunek 4: Wizualizacja stanu po hamowaniu. Wartości przy autach symbolizują prędkości.

3. Randomizacja

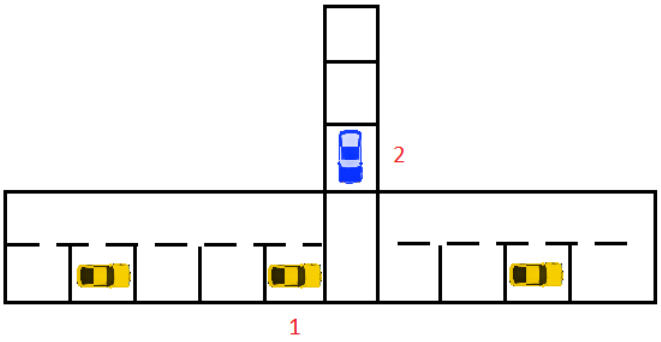
   Prędkość wszystkich aut, które się poruszają, jest zmniejszana o 1 z prawdopodobieństwem p.



Rysunek 5: Wizualizacja stanu po randomizacji. Wartości przy autach symbolizują prędkości.

4. Ruch samochodu

   Wszystkie auta przemieszczają się o tyle komórek do przodu, ile wynosi wartość ich prędkości. W przypadku auta najbliżej skrzyżowania przemieszczenie następuje maksymalnie do komórki zaraz przed skrzyżowaniem, a pozostała różnica prędkości i tego dystansu jest zapamiętywana dla fazy skrzyżowań.

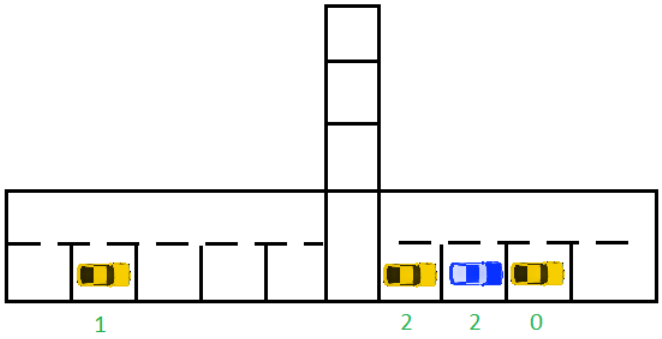


Rysunek 6: Wizualizacja stanu po ruchach samochodów. Wartości przy autach symbolizują dystans pozostały do pokonania.

5. Rozwiązanie skrzyżowania

   Auta są grupowane według pasa, na który chcą zjechać. Wcześniej zapisany dystans pozostały do przejechania określa priorytet wjazdu na pas, auta ustawiane są kolejno na możliwych pozycjach. W momencie zajęcia pierwszej komórki pozostałe auta nie wykonują przejazdu przez skrzyżowanie.

Większość mikroskopowych modeli zaimplementowanych w naszej aplikacji będzie się opierać na tej wersji modelu Nagela-Schreckenberga.



Rysunek 7: Wizualizacja stanu po wszystkich fazach. Wartości przy autach symbolizują prędkości.

2.4. Parametry opisu jakości ruchu

Ponieważ głównym celem naszego środowiska symulacyjnego jest ocenianie jakości ruchu oraz porównywanie różnych modeli i algorytmów, to konieczne jest wybranie wielkości, na podstawie których dokonamy oceny. Będą one przedstawiane w dwóch wariantach: turowym (statystyki ze stanu obecnej tury) oraz globalnym (średnia wyników na przestrzeni całej symulacji). Pod uwagę będą brane następujące parametry:

- Średnia arytmetyczna prędkości samochodów na całej mapie,
- Średnia arytmetyczna prędkości samochodów z podziałem na drogę,
- Gęstość ruchu (liczba samochodów na pole drogi) z podziałem na drogę,
- Płynność ruchu (stosunek średniej arytmetycznej prędkości na danej drodze do prędkości oczekiwanej – ustalanej przez użytkownika (np. prędkość swobodnego przepływu)).

2.5. Interfejs graficzny

Do wygodnej obsługi aplikacji konieczny będzie interfejs graficzny. Jako że środowisko symulacyjne ma mieć formę aplikacji webowej, to jej klient musi być dostępny przez przeglądarkę internetową. Interfejs musi pozwalać na: tworzenie trasy na pustym płótnie lub mapie, konfigurowanie parametrów symulacji, obserwację aktualnego stanu symulacji oraz sterowanie nią, przedstawianie statystyk obecnego stanu symulacji i jej całości, porównywanie ze sobą różnych symulacji (głównie przeprowadzanych w ramach tej samej trasy, ale ze zmienionym modelem ruchu, innym algorytmem sterującym sygnalizacją świetlną itd.).

2.5.1. Tworzenie siatki drogi

Tworzenie trasy dla symulacji będzie polegać na określeniu kształtu siatki drogowej. Jej składowe to: droga, pas, skrzyżowanie oraz brama.

Brama

Brama jest jednym z rodzajów wierzchołków tworzących graf siatki symulacji. Może stanowić zarówno początek, jak i koniec drogi. Każdy wierzchołek, który posiada drogi jednego rodzaju, musi być bramą (wyjazdową lub wjazdową). Bramy określają granicę obszaru symulacji oraz służą jako generatory ruchu (z nich samochody wyjeżdżają na trasę oraz z niej zjeżdżają). Do stworzenia bramy konieczne jest podanie jej współrzędnych na płótnie lub mapie.

Skrzyżowanie

Skrzyżowanie to drugi rodzaj wierzchołka składającego się na graf trasy. Stanowi on miejsce przecięcia się kilku dróg; zawiera także drogi wchodzące i wychodzące. Mogą znajdować się na nim sygnalizacje świetlne, które określają, z jakich pasów pojazdy mogą przejechać przez skrzyżowanie. Do określenia skrzyżowania potrzebne są: współrzędne na mapie, drogi wchodzące oraz wychodzące, światła drogowe przyporządkowane do odpowiednich pasów dróg wchodzących wraz ze stanem początkowym, i pasy wchodzące przyporządkowane do odpowiednich dróg wychodzących. Wymienione przyporządkowanie pasów wchodzących służy określeniu miejsca zjazdu z danego pasa, co umożliwia m.in. tworzenie pasów do skrętów w prawo.

Droga

Droga jest strukturą łączącą skrzyżowanie z innym skrzyżowaniem lub bramą wjazdową. Do jej określenia konieczne jest wybranie początku, końca oraz określenie parametrów pasów, które zawiera.

Pas

Pas jest podstawową składową drogi, to po nim poruszają się samochody. Żeby określić pas, potrzebna jest informacja, na którym metrze zaczyna się i kończy (pasy nie muszą być tej samej długości co droga, może istnieć np. lewy pas obecny tylko przez część długości danej drogi). Do określenia pasa konieczna jest również znajomość drogi, do której należy, przy czym każda droga musi zawierać takie pasy, żeby możliwy był przejazd od początku do końca.

2.5.2. Konfigurowanie parametrów symulacji

Parametry globalne symulacji (dotyczące każdego jej fragmentu, np. modelu ruchu), określane będą przy pomocy głównego panelu znajdującego się po lewej stronie interfejsu. Jako że poszczególne modele różnią się między sobą wymaganymi parametrami, to w zależności od wybranego modelu pojawiać się będą odpowiednie pola do zaznaczenia. Parametry lokalne (np. ilość i częstotliwość samochodów wyjeżdżających z bramy, stan początkowy świateł, algorytm nimi sterujący, prędkość oczekiwana na danej drodze (konieczna do obliczenia płynności ruchu)), będą one ustawiane dla danego elementu w menu kontekstowym, pojawiającym się po kliknięciu danego elementu trasy.

2.5.3. Sterowanie symulacją oraz jej obserwacja

Do prezentowania obecnego stanu symulacji konieczna jest wizualizacja samochodów z ich obecnymi położeniami na drodze oraz obecnym stanem sygnalizacji świetlnej.Kiedy symulacja jest rozpoczęta, po kliknięciu obiektu, w menu kontekstowym wyświetlać się będzie obecny stan danego obiektu. Do sterowania symulacją będzie służył panel znajdujący się na dole strony. Za jego pomocą użytkownik będzie mógł przesuwać się w przód i w tył o ustaloną liczbę tur. Z

panelu możliwe będzie zatrzymanie oraz uruchomienie trybu ciągłego (tj. symulacja następujących po sobie tur).

2.5.4. Prezentacja statystyk

Statystyki zebrane na przestrzeni symulacji będą prezentowane w zakładce śtatystyki". Będą się one składać z wykresów każdego parametru z wyróżnioną na przestrzeni symulacji wartością minimalną i maksymalną.

2.5.5. Porównywanie symulacji

Porównywanie symulacji będzie się odbywać w zakładce "porównaj z". W niej użytkownik będzie musiał wybrać już zapisaną symulację, z którą chce porównać obecną. Najlepsze wyniki otrzyma, porównując symulacje o tej samej trasie, ale wybór symulacji pozostawiony jest użytkownikowi. Porównanie odbywać się będzie na zasadzie przytoczenia statystyk dla danego przedziału turowego. Porównaniu towarzyszyć będzie komentarz pokazujący, która statystyka miała lepsze wyniki i w jakim stopniu. Użytkownik będzie miał również możliwość zobaczyć porównanie dla każdej tury z osobna.

3. Wybrane aspekty realizacji

3.1. Architektura serwera

Za wszystkie obliczenia symulacyjne, persystencję oraz przetwarzanie zapytań odpowiada serwer REST API. Jest on uruchomiony w chmurze, dzięki czemu może przetwarzać zapytania od wielu użytkowników jednocześnie, niezależnie od ich lokalizacji czy wykorzystania klienta.

3.1.1. Stos technologiczny

Główne narzędzia

Do napisania części serwerowej wybrany został Spring Boot [7], ponieważ jest to obecnie jeden z najpopularniejszych frameworków na rynku, ponadto posiadamy doświadczenie w budowaniu za jego pomocą aplikacji, udostępnia on również ogrom funkcjonalności w formie mniejszych bibliotek, które dołączane są do zależności projektu. W ich skład wchodzą biblioteki takie jak Spring Boot Starter Web [9], Spring Boot Starter Validation [8], Spring Boot Starter Data JPA [10], które upraszają zbudowanie serwera i łączność z bazą danych. Najczęstszym językiem, z którym się łączy ten framework, jest Java [11], w której także został napisany. Mimo to finalnie zdecydowaliśmy się na wykorzystanie języka Kotlin [12]. Kotlin, to język oparty na JVM [14], który kompiluje się do kodu binarnego Javy. Dzięki temu mamy nadal możliwość uruchamiania kodu pisanego w Javie, ale za to mamy udogodnienia specyfiki języka w tworzeniu aplikacji takie jak: null safety [15], function extensions [16] oraz eliminacje dużej ilości powtarzalnego kodu potrzebnego do podstawowych funkcjonalności [17].

Przetwarzanie zapytań

Za otrzymywanie i przetwarzanie zapytań odpowiada biblioteka wchodząca w skład frameworku Spring Boot. Zapytania odbierane są przy użyciu metod znajdujących się w klasie nazywanej kontrolerem z adnotacją `@RequestMapping()`. Adnotacja ta może przyjmować jako

argument ścieżkę, która jest dodawana jako prefiks w ścieżce URL przed docelowym węzłem końcowym. Poszczególne metody adnotowane są w zależności od metod zapytań HTTP, które dana funkcja będzie obsługiwać. Biblioteka dodatkowo zapewnia nam wygodną obsługę przesyłania danych w formie ciała załączonego do zapytania. Dodając adnotację `@RequestBody` przed argumentem w funkcji Spring biblioteka mapuje ciało zapytania w formacie JSON na klasę w języku Kotlin.

```kotlin
@Validated
@RequestMapping( ...value: "/map")
@RestController
class MapController(
    val mapper: MapMapper,
    val service: MapService
) {

    @PostMapping( ...value: "/create")
    fun createMap(@Valid @RequestBody createMapRequest: CreateMapRequest): ResponseEntity<MapDTO> {
        val result = mapper.convertToDto(service.createMap(createMapRequest))
        return ResponseEntity.ok(result)
    }
```

Rysunek 8: Przykładowa metoda przetwarzająca zapytania. W tym przypadku jest to zapytanie o stworzenie mapy pod adres `https://kraksim.herokuapp.com/map/create` z wykorzystaniem danych sparsowanych do klasy `CreateMapRequest`.

Walidacja

Walidacja otrzymanych danych odbywa się na dwa sposoby. Ręcznie sprawdzane są bardziej skomplikowane zależności, na przykład założenie, że nazwy dróg powinny być unikalne na całej mapie. Natomiast łatwiejsze realizowane są poprzez mechanizm, który wspiera framework z wykorzystaniem Spring Boot Starter Validation [8]. Biblioteka ta operuje głównie na adnotacjach nad polami w przyjmowanych obiektach, które definiują m.in. zakres, w jakim może znajdować się zmienna. Walidacja odbywa się, gdy zapytanie dotrze do węzła końcowego, ale tylko pod warunkiem, że nad danym kontrolerem znajduje się adnotacja `@Validated`, a przed argumentem, który ma być walidowany znajduje adnotacja `@Valid`.

```
class CreateLightPhaseStrategyRequest {
    lateinit var algorithm: AlgorithmType
    @field:Positive
    var turnLength: Int? = null
    @field:Positive @field:Max( value: 1)
    var phiFactor: Double? = null
    @field:Positive
    var minPhaseLength: Int? = null

    @field:Valid
    @field:NotEmpty
    lateinit var intersections: List<IntersectionId>
}
```

Rysunek 9: Przykładowa metoda zawierająca dane o strategii sterowania światłami. Adnotacja `@field:Positive` wymusza, żeby argument był liczbą większą od zera. Adnotacja `@field:Max(1)` wymaga argumentu mniejszego lub równego jeden. `@field:NotEmpty` waliduje, czy argument w postaci listy nie jest pusty. Warto zaznaczyć, że niektóre argumenty mogą przyjmować wartość null, co nie narusza wcześniejszych ograniczeń.

Persystencja

Wszelakie dane w postaci map, symulacji (w tym także każdy stan symulacji w danej turze) czy statystyk zapisywane są w bazie danych PostgreSQL [18]. W sprawnym zapisywaniu danych pomaga nam moduł Spring Starter Boot Data JPA [10]. Podobnie jak w technologii Hibernate [19] tworzymy klasy encji zawierające pola, które chcemy zapisać, i za pomocą adnotacji definiujemy relacje między obiektami oraz inne właściwości rekordu. Biblioteka w momencie uruchomienia programu tworzy odpowiednie tabele, a w momencie zapisu mapuje odpowiednio klasę na rekordy w bazie danych. W celu wykonania wspomnianych operacji na bazie tworzymy interfejsy repozytoriów, które rozszerzają interfejs `JpaRepository`. Interfejs ten posiada podstawowe metody, takie jak `getById`, które pozwala na podstawie Id wyciągnąć odpowiednią encję czy `save`, odpowiadające za zapis do bazy. W celu tworzenia własnego, nieprzewidzianego w tej podstawowej puli zapytania, dodajemy do tego interfejsu metodę z odpowiednią nazwą, spełniającą wymogi języka JPA query language [20]. Dzięki adnotacji `@Repository` framework w momencie uruchomienia dostarcza nam implementacje wszystkich metod, m.in. połączenie do bazy danych oraz otwieranie sesji i transakcji. Dane potrzebne do połączenia definiujemy w pliku `aplication.yaml`

```
@Repository
interface StatisticsRepository : JpaRepository<StatisticsEntity, Long> {
    fun findAllBySimulationEntityId(simulationId: Long): List<StatisticsEntity>
}
```

Rysunek 10: Przykładowy interfejs z dodaną metodą. Za jej pomocą możemy wyciągnąć wszystkie statystyki należące do symulacji o id podanym jako argument.

Jednym z ciekawszych z zastosowanych rozwiązań, jest skorzystanie z adnotacji `@Convert(converter)` dodawanej nad polem w encji, która jako argument przyjmuje klasę odpowiedzialną za konwersję. Bez jej użycia w celu przechowywania listy lub mapy typów prostych, biblioteka stworzyłaby dodatkową tabelę, aby stworzyć relacje wiele do jednego.

Dzięki użyciu tej adnotacji możemy sami zdefiniować zachowanie, według którego dana lista lub mapa zostanie zredukowana do jednego pola w bazie danych.

```
class LongArrayToStringConverter : AttributeConverter<List<Long>, String> {
    override fun convertToDatabaseColumn(attribute: List<Long>?): String? {
        return attribute?.joinToString( separator: ",")
    }

    override fun convertToEntityAttribute(dbData: String?): List<Long> =
        when (dbData) {
            null, "" -> emptyList()
            else -> dbData.split( ...delimiters: ",").map { it.toLong() }
        }
}
```

Rysunek 11: Przykładowy konwerter redukujący listę liczb typu `Long` do ciągu znaków, rozdzielając liczby znakiem przecinka.

Mapowanie zapytań

Ponieważ mamy dwa rodzaje adnotacji występujące bezpośrednio nad polami (adnotacje związane z walidacją oraz adnotacje związane z persystencją), łatwo jest doprowadzić do dezorganizacji w klasach. Aby tego uniknąć, w ramach dobrej praktyki, dokonany został podział, w wyniku którego oba rozwiązania operują na osobnych klasach. Podział ten wytworzył jednak kolejny problem, jakim jest mapowanie zawartości jednej klasy na drugą. Większość pól w obu klasach pokrywa się ze sobą zarówno nazwą, jak i zawartością. W celu ograniczenia ilości zbędnego kodu używamy biblioteki MapStruct [23]. Bibliotekę dołączamy w zależnościach, a także dodajemy ją jako zależność do kapt [21], czyli pluginu do przetwarzania adnotacji dla języka Kotlin, za pomocą którego możemy odwoływać się do wygenerowanego kodu z poziomu Kotlina. Ponownie definiujemy interfejs lub klasę abstrakcyjną, w której dodajemy metody, które jako argument przyjmują obiekt jednej klasy, a jako typ zwracany inną klasę, na którą ten obiekt będzie sparsowany. Biblioteka w momencie kompilacji automatycznie generuje implementację interfejsu, która przekopiowuje elementy z jednej klasy do drugiej. Jeżeli występują jakieś różnice (np. chcemy, żeby pole w klasie encji nazywało się inaczej niż to w zwracane), możemy za pomocą adnotacji definiować te różnice oraz sposób zachowania danej biblioteki.

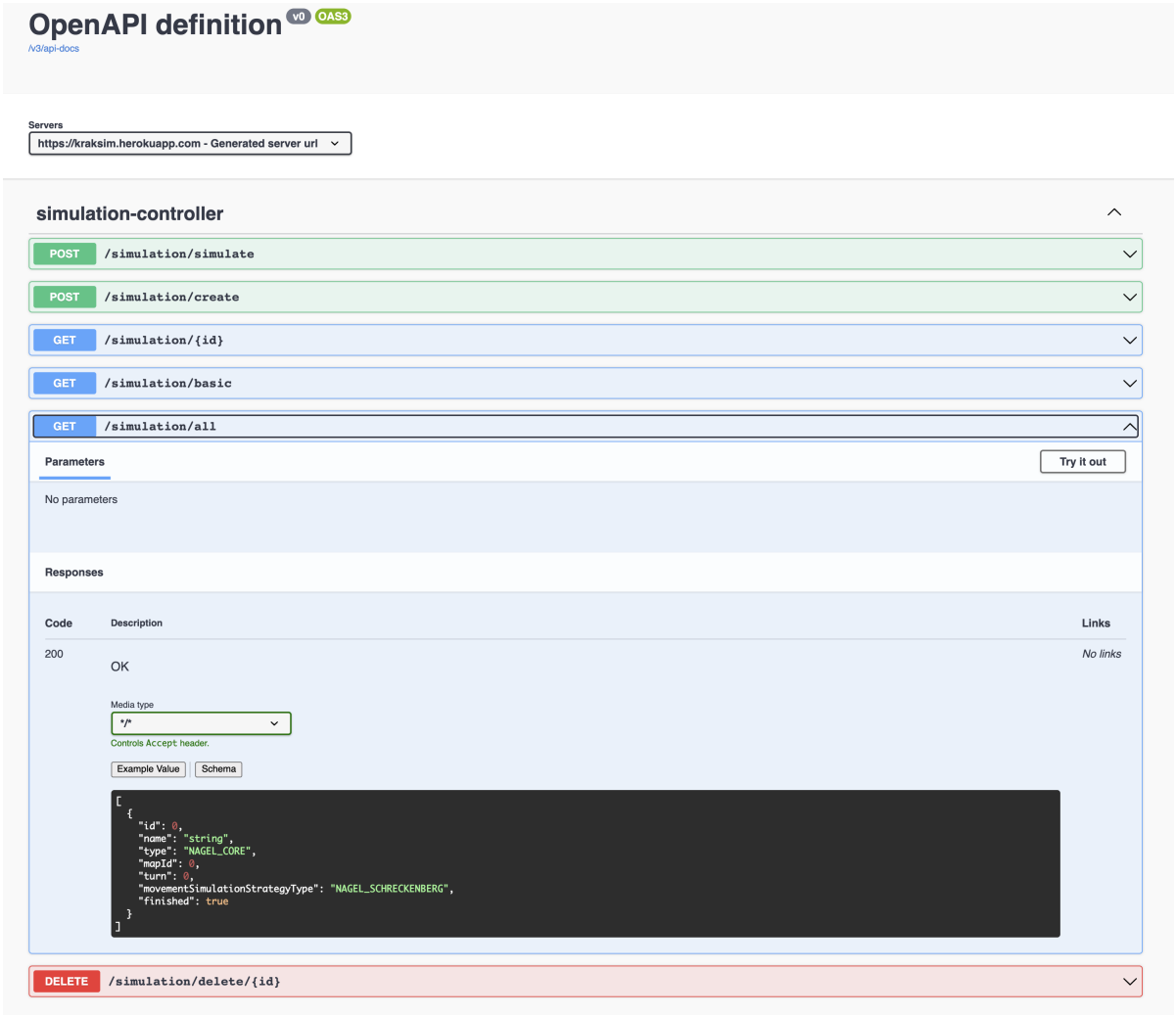
Rozwiązanie to można stosować także do klas zagnieżdżonych, wystarczy zdefiniować, z jakich innych mapperów ma skorzystać dany mapper. Wspomniana biblioteka obsługuje również cykliczne dependencje. Kiedy podklasa w mapowanym obiekcie ma referencje do rodzica, wystarczy dodać jako kolejny argument, w funkcji odpowiedzialnej za mapowanie tego obiektu, obiekt typu `CycleAvoidingMappingContext`, który przechowuje obiekty, dla których mapowanie zostało już dokonane. Dostarczeniem instancji tego obiektu zajmuje się już biblioteka.

```
@Mapper(uses = [SimulationStateMapper::class,
        MapMapper::class, MovementSimulationStrategyMapper::class,
        LightStateMapper::class])
interface SimulationMapper {

    /**
     * we add CycleAvoidingMappingContext, which is a hashmap
     * that before it maps checks if it has saved instance mapped earlier,
     * it helps with circular dependency
     */
    @Mapping(source = "mapEntity", target = "mapDTO")
    fun convertToDTO(
        simulationEntity: SimulationEntity,
        @Context context: CycleAvoidingMappingContext = CycleAvoidingMappingContext()
    ): SimulationDTO
}
```

Rysunek 12: Przykładowy mapper, który mapuje dane obiektu encji bazodanowej na obiekt służący do transferu danych – zwracany jako wynik zapytania do węzła końcowego. Można tutaj zauważyć dodatkowy mechanizm zapobiegania cyklicznej dependencji.

Swagger

Do projektu została dodana także dokumentacja Swagger [32]. Jest ona automatycznie generowana przez bibliotekę springdoc-openapi-ui, dlatego mamy zawsze pewność, że jest aktualna. Można ją znaleźć, wchodząc na adres serwera z końcówką ‘/swagger-ui.html‘. Swagger w przejrzysty sposób wyświetla nam dostępne węzły końcowe wraz z parametrami i ciałem zapytania, które przyjmują, a także typ obiektów, których możemy spodziewać się jako odpowiedź.



Rysunek 13: Przykładowa dokumentacja dla kontrolera symulacji. Rozwinięta dokumentacja węzła końcowego '/simulation/all' wskazuje, że węzeł ten nie przyjmuje żadnych parametrów, oraz że jako odpowiedź otrzymujemy listę uproszczonych obiektów symulacji. Aktualną dokumentację można znaleźć pod linkiem `https://kraksim.herokuapp.com/swagger-ui.html`.

3.1.2. Serwis map

Odpowiedzialnością serwisu map jest tworzenie, pobieranie oraz walidacja map tras, na których mogą odbywać się symulacje. Udostępnia on następujące węzły końcowe:

1. **POST /map/validate** – sprawdza poprawność utworzonej mapy,
2. **POST /map/create** – tworzy daną mapę pod warunkiem, że jest skonstruowana poprawnie,
3. **GET /map/{id}**– zwraca mapę o podanym id,
4. **GET /map/basic/{id}** – zwraca uproszczoną postać mapy, używaną przy jej wizualizacji,
5. **GET /map/all** – zwraca wszystkie mapy w postaci uproszczonej.

Serwis składa się z elementów widocznych na rysunku 14.

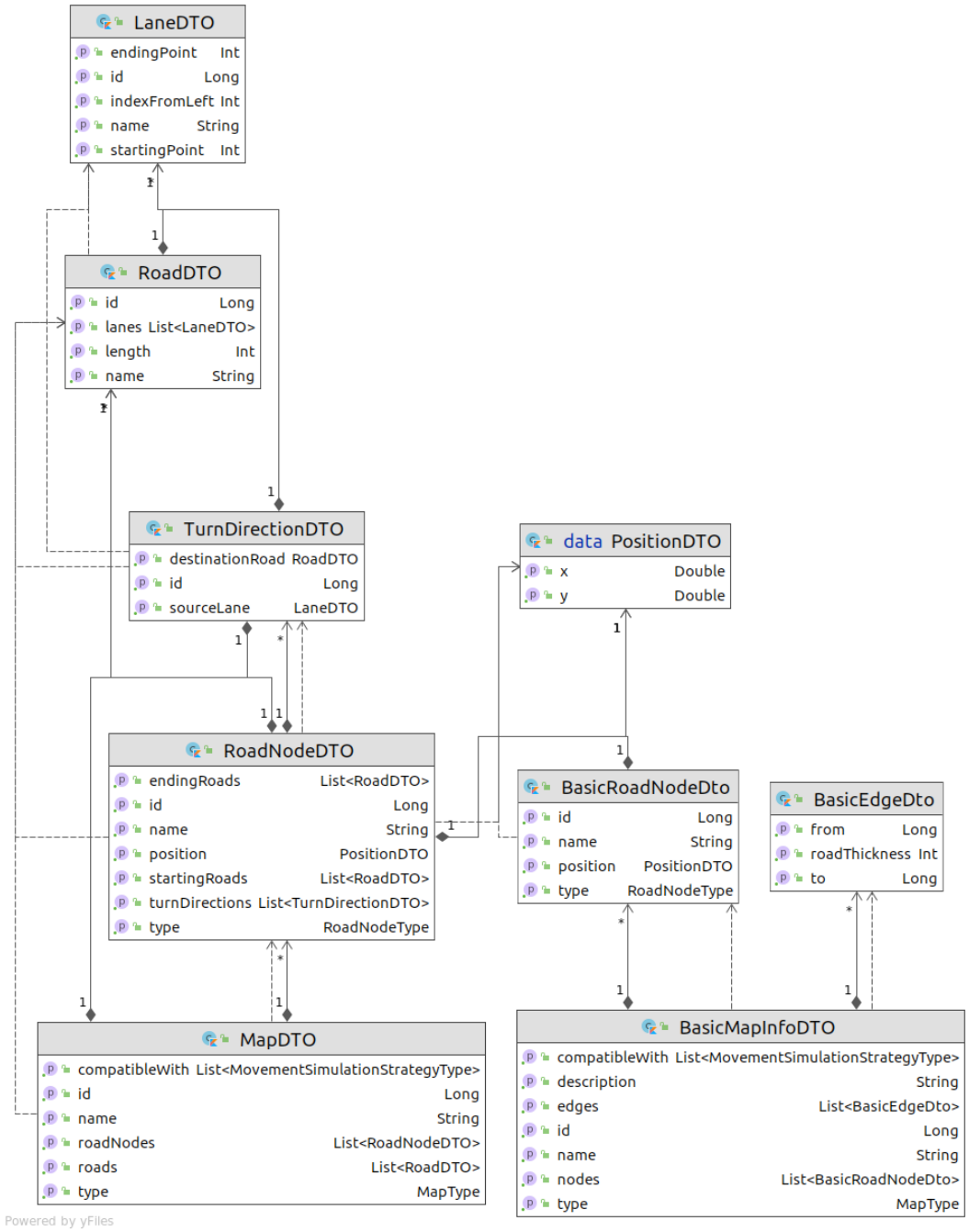
Rysunek 14: Diagram UML serwisu map.

Każdy z komponentów ma określoną rolę:

- **MapController** – Klasa wystawiająca węzły końcowe zdefiniowane wcześniej. Na podstawie otrzymanego zapytania komunikuje się z `MapService` i otrzymuje od niego potrzebne dane, które potem w razie konieczności parsuje za pomocą klasy `MapMapper`. Odbywa się w niej wstępna walidacja poprawności zapytań.
- **MapService** – Służy do kontaktu z repozytorium, odbywa się w nim dokładniejsza walidacja poprawności zapytania (między innymi czy stworzona mapa ma odpowiednio określone skrzyżowania i bramy), a także dostosowuje strukturę danych otrzymanych z repozytorium do wymaganej dla danego węzła końcowego.
- **MapMapper** – Interfejs określający klasę, która jest generowana za pomocą biblioteki MapStruct. Służy do parsowania danych z encji na postać zwracaną przez zapytanie i odwrotnie. Interfejsy, od których jest zależny – `RoadNodeMapper` oraz `RoadMapper` spełniają analogiczną rolę, ale służą do parsowania konkretnych składowych mapy.
- **MapRepository** – Repozytorium danych związanych z mapą, odpowiada za kontakt z bazą danych. Implementację tego obiektu dostarcza Spring.

Zwracane dane

Węzły końcowe serwisu zwracają dwa rodzaje obiektów – mapę oraz mapę w postaci uproszczonej. Są one wykorzystywane przez aplikacje wizualizującą głównie w dwóch kontekstach – przy formularzu do tworzenia symulacji (wersja pełna) oraz wizualizacji mapy (wersja uproszczona). Węzeł służący do walidacji oprócz uproszczonej wersji mapy zwraca także błędy walidacji, jakie napotkał. Zostało to zaprojektowane w ten sposób, ponieważ podczas tworzenia mapy chcemy możliwie narysować to, co użytkownik wprowadził, informując go jednocześnie, co musi poprawić, żeby mapa mogła zostać utworzona. Strukturę klas będących składowymi tych obiektów możemy zobaczyć na rysunku 15.

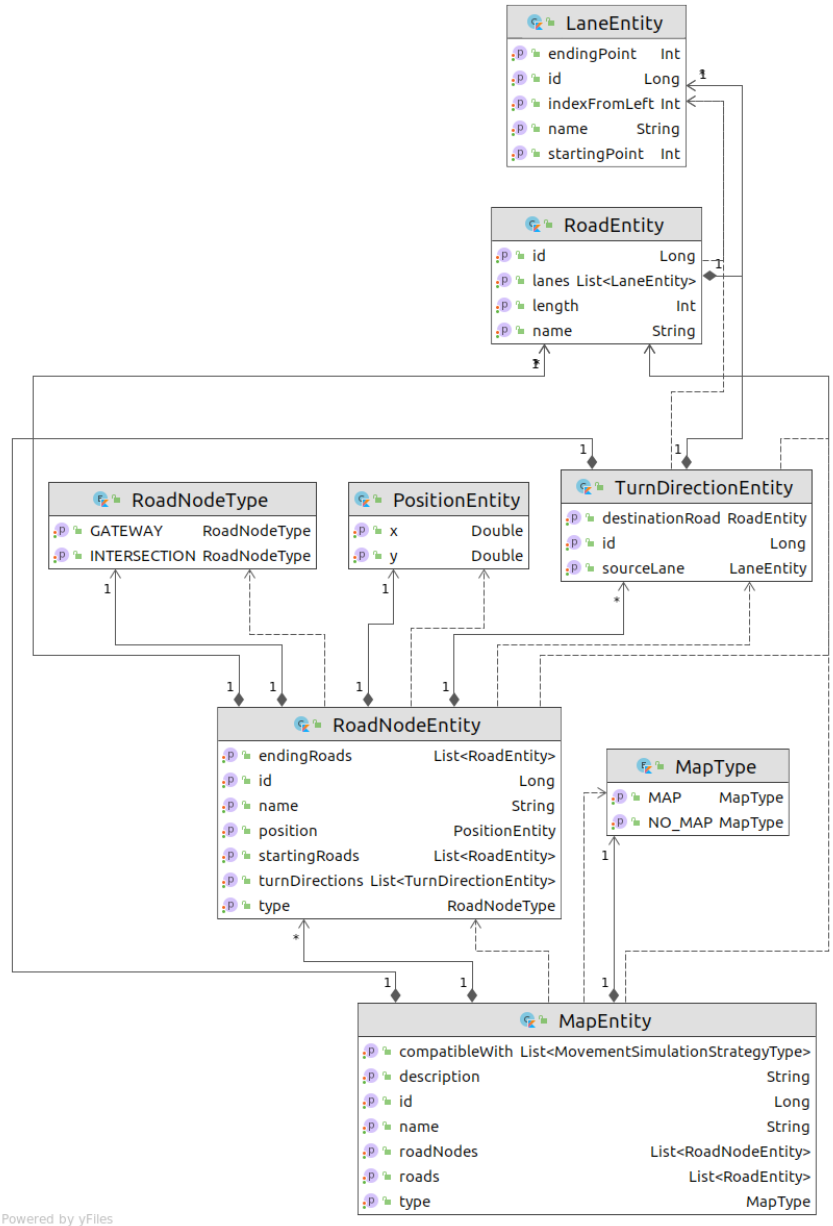


Rysunek 15: Diagram klas składowych mapy oraz jej uproszczonej wersji.

Jak można zauważyć na rysunku 15, uproszczona wersja mapy zawiera podstawowe informacje o jej właściwościach oraz o grafie ją tworzącym (lista wierzchołków oraz lista krawędzi). Dokładny opis pól mapy znajduje się w następnym rozdziale.

Persystencja

Serwis persystuje jeden rodzaj obiektu – mapę. Później, w razie potrzeby, to na jej podstawie generowane są uproszczone obiekty. Nie przewidujemy na ten moment możliwości usuwania map. Wynika to z faktu, że na podstawie map tworzone są symulacje. Usunięcie mapy pociągałoby więc ze sobą konieczność usunięcia wszystkich symulacji odbywających się na danej mapie. Niemniej jednak w razie konieczności ta funkcjonalność może zostać łatwo zaimplementowana. Encje składające się na mapę można zobaczyć na rysunku 16.



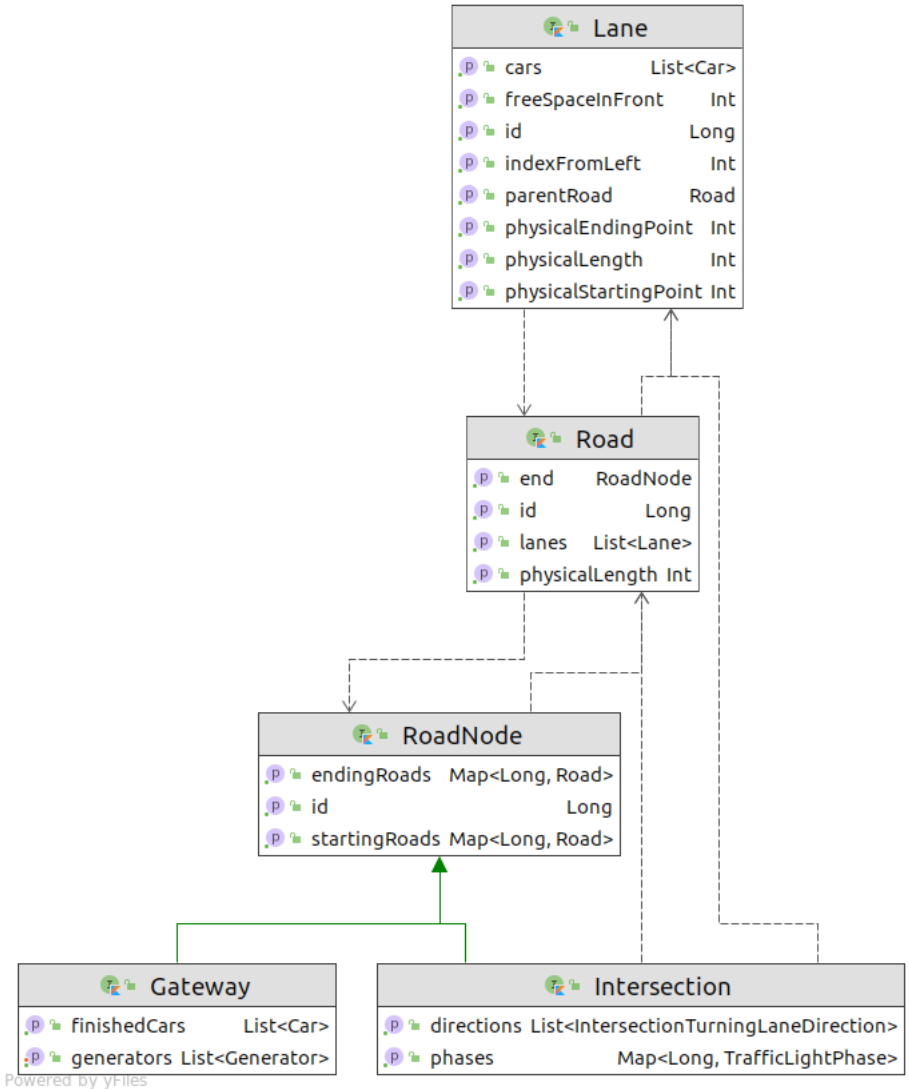
Rysunek 16: Diagram encji składowych persystowanej mapy.

Struktura mapy

Ponieważ chcieliśmy zapewnić jak największą niezależność pomiędzy logiką modelu ruchu, mapą, a logiką symulacji, konieczne było wydzielenie pewnych podstawowych komponentów mapy. Są one zdefiniowane jako interfejsy, które będą implementowane przez odpowiednie klasy, stanowiące odwzorowanie danej składowej mapy. Klasy te będą zależeć od zdefiniowanego dla mapy modelu ruchu. W przypadku różnych modeli może być konieczne np. użycie prywatnych metod pomocniczych specyficznych tylko dla tego modelu. Te podstawowe komponenty zostały zdefiniowane za pomocą interfejsów:

- `RoadNode`, rozszerzany przez `Gateway` i `Intersection`,
- `Road`,
- `Lane`.

Relacje pomiędzy tymi interfejsami przedstawia rysunek 17.



Rysunek 17: Diagram interfejsów składowych mapy.

Węzły

Interfejs RoadNode reprezentuje węzeł, czyli punkt, w którym zaczynają i kończą się drogi. W wizualizacji węzły są przedstawione jako wierzchołki grafu skierowanego. Posiada on następujące pola:

- **id** – Identyfikator węzła,
- **endingRoads** – Mapa zawierająca drogi, które kończą się w tym węźle. Kluczem jest tutaj identyfikator tej drogi,
- **startingRoads** – Analogicznie do poprzedniego pola, mapa zawierająca drogi zaczynające się w tym węźle.

Stanowi on część wspólną bramy i skrzyżowania

Bramy

Bramy są jednym z rodzajów węzłów, ich rolą jest określenie granic mapy. Podczas symulacji wyjeżdżają z nich oraz zjeżdżają do nich samochody (tylko brama może zostać określona jako cel podróży samochodu). Węzeł, który posiada drogi jednego rodzaju (zaczynające lub kończące się), musi być bramą. Bramy są reprezentowane przez interfejs Gateway, który oprócz pól RoadNode posiada dodatkowo:

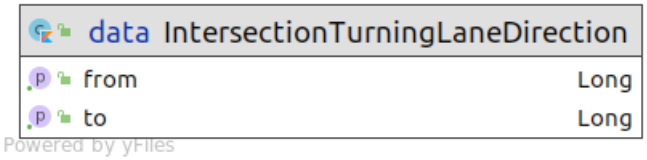
- **finishedCars** – lista aut, które zakończyły symulację i zjechały do tej bramy,
- **generators** – lista generatorów tworzących auta (o ich działaniu można przeczytać w rozdziale serwis symulacji).

Skrzyżowanie

Skrzyżowania to drugi rodzaj węzłów, będących miejscem przecięcia dróg. Samochody mogą je pokonywać w celu dotarcia do wyznaczonej im bramy. Posiadają one światła na każdym pasie dróg, które kończą się na danym skrzyżowaniu – mogą być więc przyczyną korków. Interfejs `Intersection`, który je reprezentuje, posiada następujące dodatkowe pola:

1. **phases** – Pole zawierające fazy świateł dla danego pasa,
2. **directions** – Lista obiektów określających, z którego pasa można skręcić na jaką drogę.

Projektując skrzyżowania, zależało nam, aby możliwe było wyznaczanie specjalnych pasów, z których skręty będą dopuszczalne tylko w konkretne drogi (np. pas do skrętu w lewo). Taka możliwość daje zdecydowanie większą różnorodność w obrębie możliwych do stworzenia map i symulacji. Funkcjonalność ta została zaimplementowana za pomocą listy directions. Obiekty przez nią przechowywane mają postać widoczną na rysunku 18.



Rysunek 18: Klasa IntersectionTurningLaneDirection.

Niestety, to rozwiązanie ma też wady. Z powodu konieczności określenia dozwolonych dróg dla każdego pasa, żądanie tworzenia mapy już dla kilku skrzyżowań potrafi osiągnąć ogromne rozmiary. W większości przypadków użytkownik będzie zainteresowany zdefiniowaniem specjalnych pasów skrętu, tylko w kontekście ograniczonej liczby skrzyżowań. Umożliwiamy więc ominięcie definiowania wszystkich dozwolonych skrętów za pomocą flagi `overrideTurnDirectionsTurnEverywhere` – ustawienie jej wartości na `true` w zapytaniu spowoduje wygenerowanie wszystkich możliwych kierunków skrętu (rysunek 19).

```
private fun getTurnDirections(
    request: CreateRoadNodeRequest,
    lanes: Map<Long, LaneEntity>,
    roads: Map<Long, RoadEntity>
): List<TurnDirectionEntity> {
    if (request.type == RoadNodeType.GATEWAY) {
        return emptyList()
    }
    if (request.overrideTurnDirectionsTurnEverywhere) {
        return generateTurnDirectionEverywhere(roads, request)
    }
    return request.turnDirections!!.map { createTurnDirectionRequest ->
        TurnDirectionEntity(
            sourceLane = lanes[createTurnDirectionRequest.sourceLaneId]!!,
            destinationRoad = roads[createTurnDirectionRequest.destinationRoadId]!!
        )
    }
}
```

Rysunek 19: Fragment kodu pokazujący zastosowanie flagi `overrideTurnDirectionsTurnEverywhere`.

Drogi

Interfejs Road przedstawia drogi na mapie. Droga w naszym przypadków składa się z pasów biegnących w tym samym kierunku (co jest w pewien sposób nieintuicyjne, gdyż w życiu codziennym spotykamy drogi z pasami przeciwnie zwróconymi). Każda droga posiada początek i koniec w postaci węzła oraz taką konfigurację pasów, która pozwala na przejechanie z jednego końca na drugi. Między dwoma węzłami mogą przebiegać dwie drogi, po jednej w każdym kierunku. Interfejs posiada następujące pola:

- **end** – Węzeł, na którym droga się kończy (w jego stronę jadą samochody na danej drodze),
- **id** – Identyfikator drogi,
- **lanes** – Lista pasów zawartych w tej drodze,
- **physicalLength** – Fizyczna długość drogi.

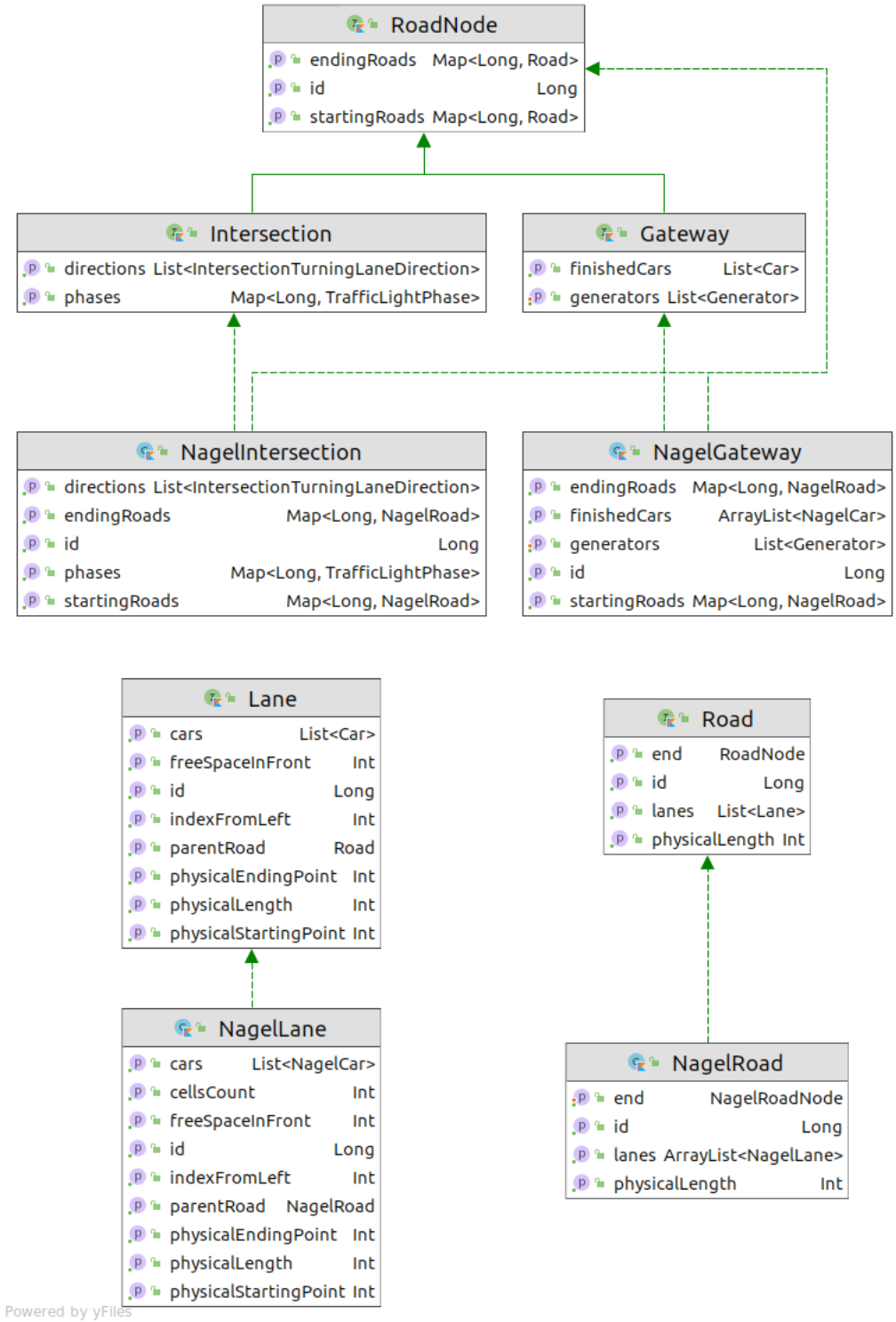
Pasy

Pasy są składowymi każdej drogi, to po nich poruszają się samochody. Pasy nie muszą mieć długości równej drodze, mogą zaczynać i kończyć się przed jej końcem. Dla każdej z dróg

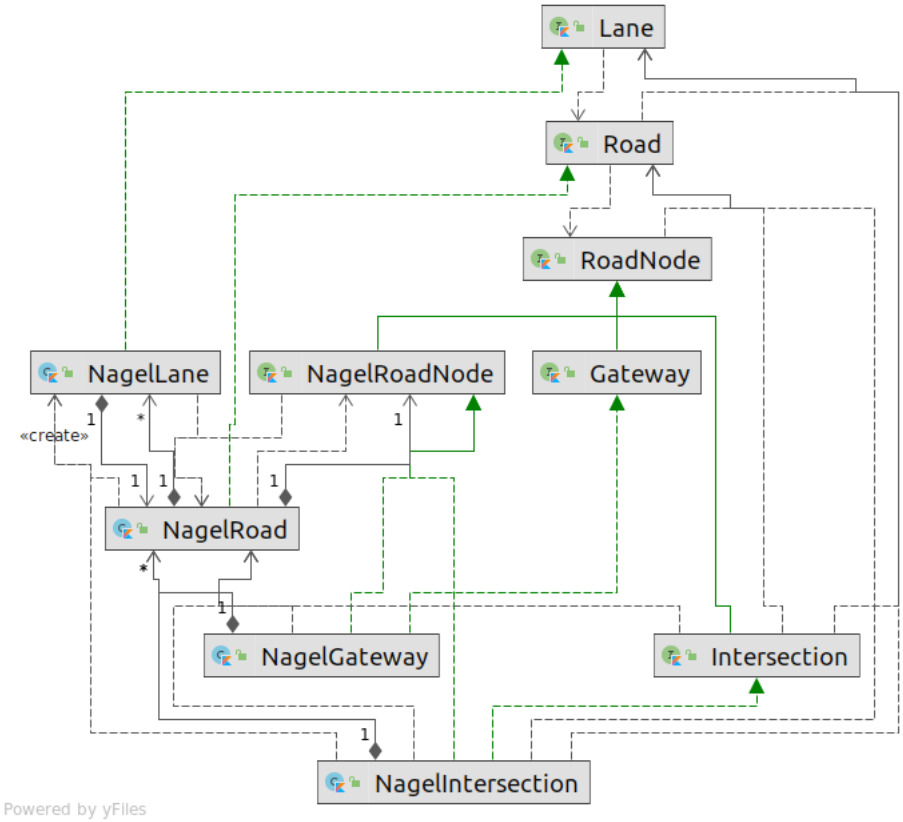
muszą być zdefiniowane tak, aby możliwy był przejazd. Każdy pas, którego droga kończy się na skrzyżowaniu, ma na końcu światło drogowe (czerwone lub zielone). Pas jest reprezentowany przez interfejs Lane o polach:

- **cars** – Samochody znajdujące się na danym pasie,
- **freeSpaceInFront** – Odległość pomiędzy pierwszym samochodem na pasie a jego końcem, wartość wyliczana na bieżąco potrzebna w kontekście zmiany pasa,
- **id** – Identyfikator pasa,
- **indexFromLeft** – Numer pasa, licząc od lewej,
- **parentRoad** – Droga, do której należy pas,
- **physicalEndingPoint** – Punkt, w którym pas się kończy,
- **physicalStartingPoint** – Punkt, w którym pas się zaczyna,
- **physicalLength** – Długość pasa, obliczana na podstawie dwóch powyższych wartości.

Przy dodawaniu do aplikacji możliwości stosowania konkretnego modelu ruchu, tworzy się odpowiedniki powyższych obiektów. Odpowiedniki te implementują wymienione w tym rozdziale interfejsy. Przykład dla algorytmu Nagela-Schreckenberga przedstawia rysunek 20.



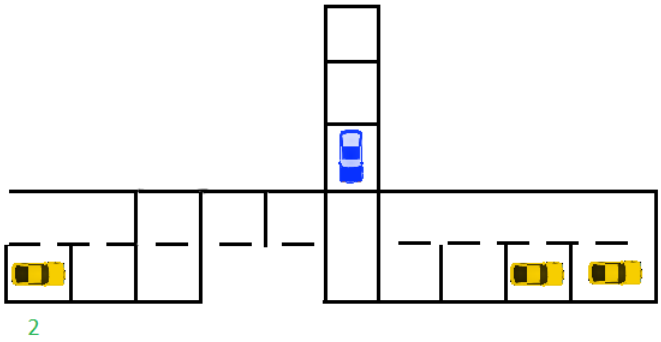
Rysunek 20: Diagram implementacji podstawowych interfejsów mapy dla modelu Nagela-Schreckenberga.



Rysunek 21: Pełny diagram UML implementacji mapy dla modelu Nagela-Schreckenberga.

Kompatybilność

Pomimo naszych starań, utrzymanie każdej mapy kompatybilnej dla wszystkich rodzajów symulacji okazało się niemożliwe. W przypadku zaimplementowanych przez nas modeli ruchu przeszkodą okazało się zachowanie przy drogach z pasami. W modelach, które wykluczają zmianę pasa przez samochód, po początkowym wjechaniu na urwany pas (przy wyjeżdżaniu z bramy lub po pokonaniu skrzyżowania) samochód nie może już go opuścić – utknie na końcu pasa i przez to symulacja nie będzie mogła zostać zakończona. Żeby temu zaradzić, wprowadziliśmy do mapy pole `compatibleWith` – tablicę, która zawiera wszystkie kompatybilne symulacje dla danej mapy. Dzięki temu mamy pewność, że każda utworzona symulacja dla danej mapy będzie się mogła zakończyć. Unikniemy też sytuacji, w której do bazy zapisywane jest wiele nadmiarowych rekordów z symulacji, w której zostały już tylko samochody niebędące w stanie ukończyć trasy.



Rysunek 22: Droga z dwoma niepełnymi pasami – w przypadku symulacji nieprzewidującej możliwości zmiany pasa samochód nr 2 nie będzie w stanie przejechać przez skrzyżowanie.

Walidacja

W celu upewnienia się, że na danej mapie da się przeprowadzić symulacje, przed jej stworzeniem i umieszczeniem w bazie przeprowadzamy sprawdzenie jej poprawności. Sprawdzenie to wymaga następujących obiektów składowych mapy:

- Węzły

    1. Istnieją co najmniej dwa węzły będące bramami – przy definiowaniu generatora konieczne jest podanie bramy początkowej i końcowej. W obecnej implementacji nie może być to ta sama brama (nie zawsze da się zawrócić w to samo miejsce), więc do utworzenia jakiegokolwiek ruchu samochodów konieczna jest obecność przynajmniej 2 bram.
    2. Każdy węzeł ma unikalną nazwę, aby rozróżnianie węzłów było łatwiejsze, szczególnie przy interakcji z wizualizacją mapy oraz wypełnianiu formularza stworzenia nowej symulacji, wymuszamy na użytkowniku podanie unikatowej nazwy.
    3. Każde skrzyżowanie ma co najmniej jedną drogę wychodzącą – węzeł, który ma tylko drogi wchodzące naturalnie stanowi granicę mapy. W naszym modelu przyjęliśmy, ze taką rolę pełnią bramy, więc z tego powodu nie dopuszczamy możliwości skrzyżowań będących punktem końcowym trasy.
    4. Wszystkie skrzyżowania muszą mieć zdefiniowane skręty – w przypadku map kompatybilnych z modelami jednopasmowymi każdy pas na skrzyżowaniu musi mieć możliwość zjazdu na każdą drogę wychodzącą. Jest to spowodowane ograniczeniami GPSa dla modeli jednopasmowych. W przypadku mapy dla modeli wielopasmowych ten warunek jest znacznie luźniejszy – wystarczy jeden skręt zdefiniowany dla jednego skrzyżowania. Może to budzić pewne wątpliwości – w takim zestawieniu mogą istnieć drogi, z których można wprawdzie dojechać do skrzyżowania, ale nie można go już pokonać. Stwierdziliśmy jednak, że może to być dobra metoda symulowania np. remontów drogi lub ich zamknięcia z powodu np. demonstracji.
    5. Skręty muszą być poprawne – skręty zdefiniowane dla skrzyżowań muszą być ustawiane z pasów drogi wchodzącej do drogi wychodzącej.

- Drogi

    1. Mapa zawiera co najmniej jedną drogę – aby ruch samochodów się odbywał, potrzebna jest przynajmniej jedna droga, po której mogą jechać.
    2. Identyfikatory oraz nazwy dróg muszą być unikatowe – analogicznie jak dla węzłów, umożliwia to ich komfortowe odróżnienie.
    3. Dla każdego pasa zaczynającego się od węzła musi istnieć możliwość dojazdu do końca drogi. Jako że samochody mogą wjechać na każdy pas, który ma bezpośrednią styczność z węzłem (przy wkraczaniu na mapę z bramy lub po pokonaniu skrzyżowania), to, aby uniknąć utknięcia pojazdu, musi istnieć możliwość pokonania za jego pomocą (lub innych, przy algorytmie pozwalającym na zmianę pasa) drogi do końca.

- Pasy

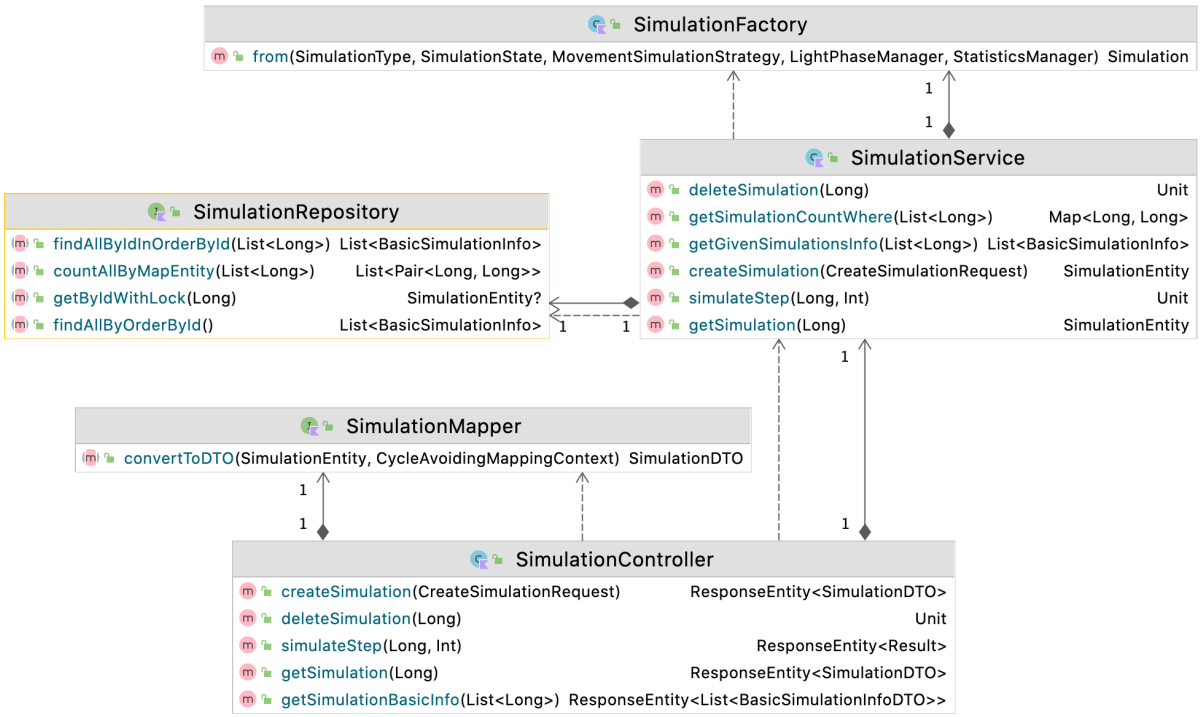
    1. Identyfikatory pasów są unikalne – na tej samej zasadzie co identyfikatory dróg.
    2. Punkt startowy i końcowy pasa musi znajdować się w obrębie drogi – nie może wystąpić sytuacja, gdzie pas będący częścią drogi jest od niej dłuższy.
    3. Pasy poprawnie uporządkowane od lewej do prawej – indeksy zaczynają się od 0 dla najbardziej lewego pasa i rosną z każdym miejscem w prawo o jeden. Muszą też być uporządkowane rosnąco.

3.1.3. Serwis symulacji

Serwis symulacji jest odpowiedzialny za tworzenie, pobieranie, usuwanie i uruchamianie symulacji. Udostępnia on następujące węzły końcowe:

1. **GET /simulation/{id}**– zwraca symulację o podanym id,
2. **GET /simulation/all** – zwraca listę podstawowych informacji o wszystkich symulacjach,
3. **POST /simulation/simulate** – uruchamia pożądaną liczbę kroków symulacji,
4. **POST /simulation/create** – tworzy nową symulację,
5. **DELETE /simulation/delete/{id}** – usuwa symulację o podanym id,

Serwis składa się z elementów widocznych na rysunku 23.



Rysunek 23: Diagram UML serwisu symulacji.

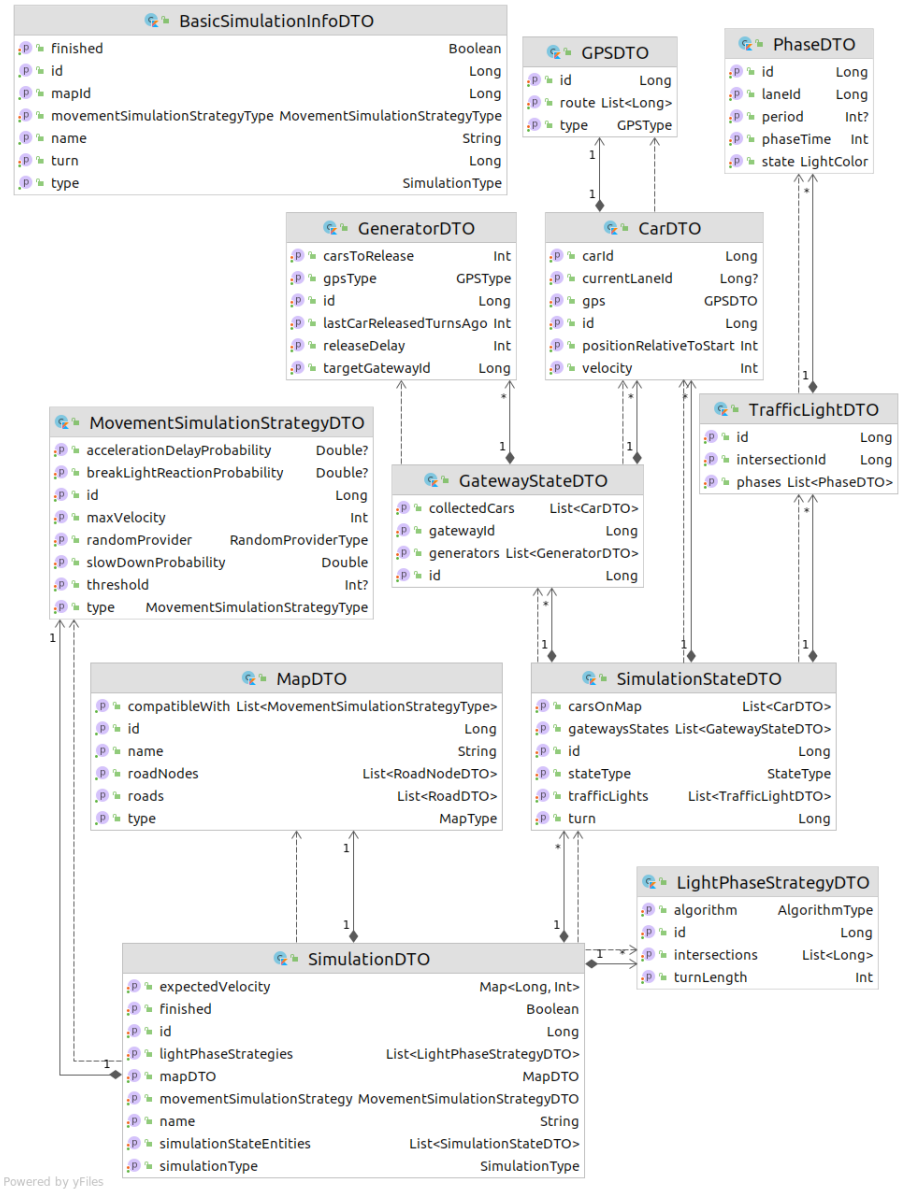
Każdy z komponentów ma określoną rolę:

- **SimulationController** – Klasa, która wystawia węzły końcowe wymienione wyżej. Przekazuje żądania do SimulationService oraz, w razie potrzeby, zwraca odpowiedź przekształconą przy pomocy klasy SimulationMapper.
- **SimulationService** – Serwis, którego zadaniem jest tworzenie symulacji (przy pomocy SimulationFactory), uruchamianie jej oraz kontakt z repozytorium. Podczas tworzenia dodatkowo waliduje on poprawność otrzymanych danych (sprawdza m.in. czy zostały podane wszystkie parametry potrzebne do stworzenia strategii poruszania pojazdów, oraz czy mają one właściwe wartości).
- **SimulationMapper** – Interfejs określający klasę, która jest generowana za pomocą biblioteki MapStruct. Służy do parsowania danych z encji na postać zwracaną przez zapytanie.
- **SimulationRepository** – Repozytorium danych związanych z symulacją, odpowiada za kontakt z bazą danych. Implementację tego interfejsu dostarcza spring.
- **SimulationFactory** – Służy do stworzenia symulacji odpowiedniego typu.

Zwracane dane

Ponownie jak w przypadku mapy, węzły końcowe statystyk zwracają dwa rodzaje obiektów – cały obiekt symulacji oraz podstawowe informacje o symulacji. Drugi typ obiektów wykorzystywany jest w widoku listy kilku symulacji, do której potrzebujemy tylko podstawowych informacji wyświetlonych w formie tabelki. Pierwszy typ obiektów jest zwracany, ale nie jest obecnie wykorzystywany, ponieważ nie mamy funkcjonalności odpowiadającej za rysowanie

symulacji wraz z autami, pojedynczymi pasami oraz światłami. Struktura klasy będących składowymi tych obiektów jest przedstawiona na rysunku 24.



Rysunek 24: Diagram klas składowych symulacji oraz uproszczonej wersji.

Symulowanie może trwać nawet do pięciu minut, dlatego został dodany mechanizm symulowania asynchronicznego. Węzeł końcowy przetwarzając zapytanie, uruchamia symulacje na osobnym wątku. Następnie, jeżeli w ciągu piętnastu sekund symulacja zakończy się, zwraca informacje o ukończonej symulacji. W przeciwnym przypadku zwrócona zostaje informacja o tym, że symulacja nadal się przetwarza.

Rozszerzalność

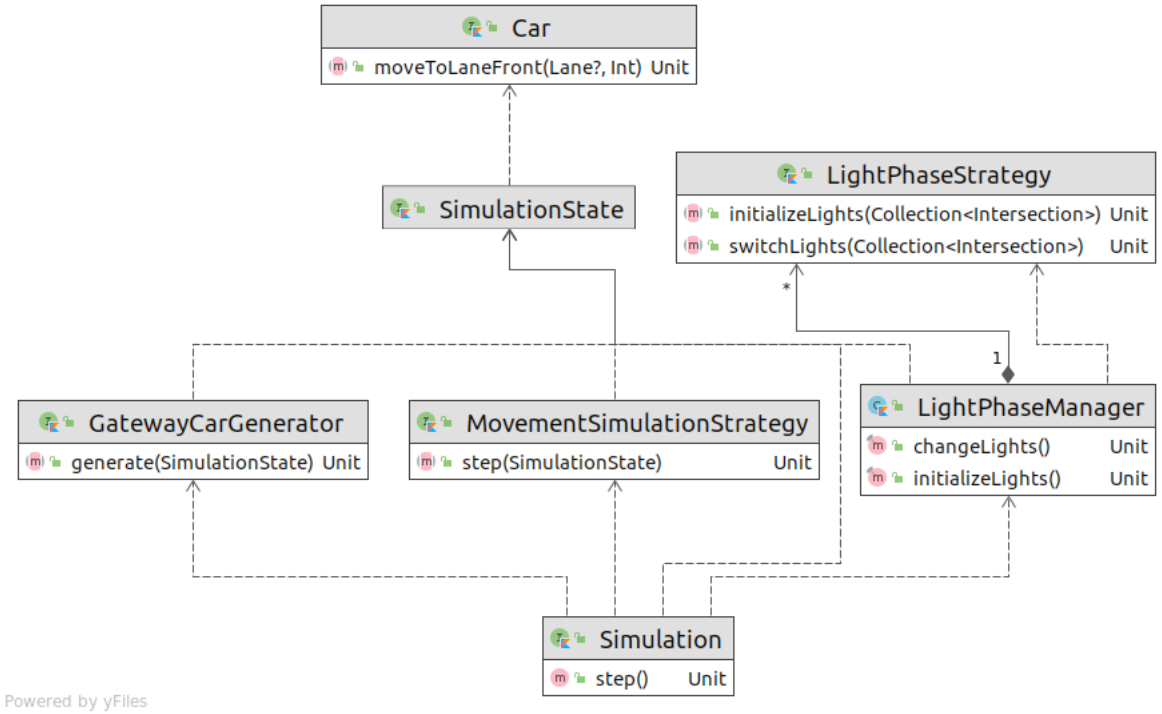
Model systemu, który stworzyliśmy, został w pełni stworzony z myślą o przyszłym rozwoju. Elementy takie jak: model symulacji ruchu, model sterowania sygnalizacją świetlną, czy typ całej symulacji, zostały zdefiniowane przy pomocy interfejsów, które określają wymagane metody i atrybuty niezbędne do umieszczenia ich w symulacji i obsłużenia.

Po dodaniu implementacji możemy w prosty sposób dodać tworzenie strategii do odpowiednich fabryk (`SimulationStrategyFactory`, `LightPhaseManagerFactory`, `SimulationFactory`)

Podstawowe interfejsy

Wspomniane wcześniej interfejsy, które umożliwiają nam rozszerzalność, to:

- **MovementSimulationStrategy** – implementując ten interfejs, definiujemy, w jaki sposób przetwarzany będzie krok symulacji,
- **SimulationState** – opisuje aktualny stan symulacji. Na stan symulacji składa się: stan mapy, stan pojazdów oraz informacje o turze. Posiada on następujące pola:
  - **id** – identyfikator stanu,
  - **turn** – numer tury,
  - **roads** – lista dróg w mapie,
  - **gateways** – lista bram w mapie,
  - **intersections** – lista skrzyżowań w mapie,
  - **lanes** – lista pasów w mapie,
  - **cars** – lista aut w symulacji,
  - **type** – typ symulacji,
  - **finished** – flaga informująca, czy symulacja się zakończyła.
- **Car** – opisuje pojazdy w symulacji. Każdy pojazd musi mieć prędkość oraz pozycję na pasie. Posiada następujące pola:
  - **id** – identyfikator pojazdu,
  - **velocity** – prędkość pojazdu,
  - **positionRelativeToStart** – pozycja pojazdu względem początku pasa,
  - **gps** – obiekt informujący pojazd gdzie ma jechać.
- **LightPhaseStrategy** – opisuje metody, które musi posiadać strategia świateł. Każda strategia musi posiadać metodę inicjalizującą światła drogowe oraz taką, która w każdej turze aktualizuje stan świateł.
- **GatewayCarGenerator** – posiada jedną metodę, którą muszą implementować generatory. Jest nią metoda `generate`, która jest wywoływana w każdej turze. Odpowiednio tworzy ona pojazdy w generatorach wewnątrz bram, jeśli z zaimplementowanego algorytmu tak wynika.



Rysunek 25: Podstawowe interfejsy tworzące serwis symulacji.

Przykładowa implementacja

Przykładowy schemat zaimplementowanych interfejsów na przykładzie modelu ruchu Nagela-Schreckenberga, świateł turowych oraz standardowego generatora pojazdów

Rysunek 26: Przykładowa implementacja elementów symulacji.

Generowanie samochodów

Pojazdy w symulacjach tworzone są przy pomocy generatorów znajdujących się w bramach. Obiekty klas implementujące interfejs `Gateway` posiadają listy zawierające obiekty klasy `Generator`. Klasa generator posiada następujące pola:

- **lastCarReleasedTurnsAgo** – informuje nas o tym, ile tur temu wypuszczony był poprzedni pojazd,
- **releaseDelay** – określa, w jakich odstępach wypuszczane mają być samochody,
- **carsToRelease** – przechowuje informację o tym, ile pozostało aut do wypuszczenia,

- **targetGatewayId** – identyfikator bramy, do której mają dojechać pojazdy stworzone przez ten generator,
- **gpsType** – typ nawigacji, jaką będą się kierować utworzone pojazdy,
- **id** – identyfikator generatora.

Obiekty klas implementujących interfejs `GatewayCarGenerator` w każdej turze przechodzą przez wszystkie bramy. Jeśli dana brama posiada generatory, w których czas od wypuszczenia pojazdu jest większy bądź równy parametrowi `releaseDelay`, próbuje ona stworzyć auto i umieścić je na początku pasa. Jeśli na początku pasów nie ma miejsca, brama w tym kroku nie robi nic. Następnie w każdym generatorze brama zwiększa wartość parametru `lastCarReleasedTurnsAgo`. Na koniec brama usuwa generatory, które nie mają już pojazdów do wypuszczenia.

```
override fun generate(state: SimulationState) {
    state.gateways.values.forEach { gateway: Gateway ->
        gateway.generateCars(state)
    }
}

private fun Gateway.generateCars(state: SimulationState) {
    removeEmptyGenerators() // to be sure that e.g. inserted by user data doesnt contain 0

    val releaseGenerators = getGeneratorsToReleaseNow()
    val generatorsThatReleased = releaseNewCarsIfSpace(releaseGenerators, state)

    increaseTurnData(generatorsThatReleased)
    removeEmptyGenerators()
}

private fun Gateway.removeEmptyGenerators() {
    generators = generators.filter { it.carsToRelease != 0 }
}
```

Rysunek 27: Fragment przykładowej implementacji `GatewayCarGenerator`.

Zaimplementowane modele symulacji ruchu

- **Model Nagela-Schreckenberga** – Podstawowy model symulacji ruchu, oryginalnie składający się z 4 faz: przyspieszenia, hamowania, randomizacji i ruchu. Dodatkowo w naszej implementacji dodaliśmy fazę rozwiązywania ruchu na skrzyżowaniu. Został zaimplementowany w klasie NagelMovementSimulationStrategy.
- **Model Wielopasmowy** – Jest to rozszerzenie modelu Nagela-Schreckenberga, które dodaje fazę zmiany pasa na początku tury. Implementacja tego modelu znajduje się w klasie MultiLaneNagelMovementSimulationStrategy, która dziedziczy po NagelMovementSimulationStrategy.

- **Model Brake Light** – Jest to kolejne rozszerzenie modelu Nagela-Schreckenberga. Podobnie jak w przypadku modelu wielopasmowego, realizujemy go poprzez dziedziczenie z klasy `NagelMovementSimulationStrategy` i dodanie fazy na początku kroku. Faza ta dla każdego pojazdu wylicza i modyfikuje wartość parametru odpowiedzialnego za spowolnienie w fazie randomizacji, biorąc pod uwagę stan pojazdu poprzedzającego. Na potrzeby tego modelu, do klasy `NagelCar` zostało dodane pole `brakeLightOn`, dzięki któremu pojazdy mogą reagować na hamowanie samochodów przed nimi.

Pokonywanie skrzyżowań

W modelach symulacji ruchu musieliśmy dodać dodatkową fazę rozwiązywania sytuacji na skrzyżowaniach, aby rozwiązać problem, który pojawiał się, gdy kilka aut chciało wjechać na ten sam pas w tej samej turze. Auta mogą chcieć wjechać na to samo pole, więc stworzyliśmy fazę zapobiegającą temu i organizującą ruch na skrzyżowaniu. W poprzednich fazach pojazdy, które mają przejechać przez skrzyżowanie, dojeżdżają do niego i zapisują, ile zostało im do przejechania. W dodanej przez nas turze dla każdego pasa zbieramy wszystkie pojazdy, które chcą na niego wjechać, po czym sortujemy je według dystansu pozostałego do przejechania. Dopóki na początku pasa pozostaje miejsce dla aut, umieszczamy je na odpowiednich pozycjach.

```
fun resolveIntersections(state: NagelSimulationState) {
    getCarsToResolve(state.roads.values).forEach { (destinationLane, cars) ->
        var spaceLeft = destinationLane.getFreeSpaceInFront()

        cars.sortedByDescending { car -> car.distanceLeftToMove }
            .takeEachWhile({ spaceLeft > 0 }) { car ->
                val newPosition = min(spaceLeft, car.distanceLeftToMove) - 1

                car.moveToLaneFront(
                    destinationLane,
                    newPosition
                )
                car.gps.popNext()
                spaceLeft = newPosition
            }
    }
}

private fun getCarsToResolve(roads: Collection<NagelRoad>): Map<NagelLane, List<NagelCar>> {
    return roads.filter { it.end is NagelIntersection }
        .flatMap { it.lanes }
        .mapNotNull { it.cars.lastOrNull() }
        .filter { it.hasDistanceLeftToMove() }
        .groupByTo(hashMapOf()) { it: NagelCar
            val targetLane = getTargetLane(it)
            it.gps.getNext()
            targetLane ^groupByTo
        }
}
```

Rysunek 28: Metody Rozwiązujące ruch na skrzyżowaniach.

Światła drogowe

Nasza implementacja modelu aktualnie przewiduje tylko skrzyżowania posiadające światła. Ponieważ większość spotykanych na co dzień skrzyżowań posiada osobne światła do np. skrętu

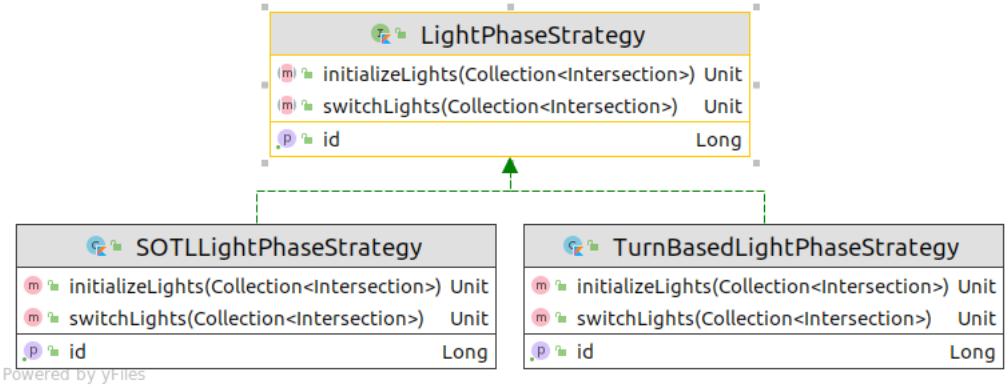
w lewo bądź w prawo, każdy z pasów posiada swój osobny sygnalizator świetlny. Reprezentuje go klasa `TrafficLightPhase`, w której skład wchodzi stan sygnalizatora (zielone lub czerwone światło, aktualna ilość tur od poprzedniej zmiany stanu oraz opcjonalny parametr, który może wykorzystać dana strategia (aktualnie wykorzystuje go algorytm turowy)). Relacja pomiędzy pasem a sygnalizatorem jest przedstawiona w formie mapy, gdzie do id pasa przypisujemy sygnalizator. Mapa ta znajduję się w `Intersection`.

```
class TrafficLightPhase(
    var phaseTime: Int = 0,
    var state: LightColor = LightColor.RED,
    var period: Int? = null
) {

    fun switchLight(period: Int) {
        this.period = period
        phaseTime = 0
        state = when (state) {
            LightColor.RED -> LightColor.GREEN
            LightColor.GREEN -> LightColor.RED
        }
    }

    enum class LightColor {
        RED, GREEN
    }
}
```

Rysunek 29: Klasa reprezentująca pojedynczy sygnalizator drogowy.

Za zmianę świateł odpowiada klasa `LightPhaseManager`. Nasz system umożliwia grupowanie kilku skrzyżowań podlegających pod dany algorytm sterowania światłami. Ma to między innymi na celu umożliwienie definiowania strategii, które wykorzystują grupowanie do tworzenia płynniejszego ruchu na przykład poprzez tworzenie kilku zielonych świateł z rzędu na głównych drogach oraz do uczenia maszynowego optymalizującego konkretną grupę skrzyżowań. Menadżer posiada dwie metody. Jedna z metod służy do inicjalizacji początkowego stanu świateł na mapie; jest uruchamiana przed wykonaniem pierwszej tury symulacji. Druga metoda wykorzystywana jest do zmiany świateł. Obie z metod uruchamiają dla każdego algorytmu analogiczne metody.

Algorytm sterowania światłami reprezentuje interfejs `LightPhaseStrategy`



Rysunek 30: Diagram UML zaimplementowanych algorytmów.

Zaimplementowane algorytmy sterowania światłami

- Algorytm turowy – Użytkownik definiuje długość jednej fazy światła w parametrze `turnLength`. Światła są grupowane po drogach, do których należą odpowiadające im pasy. Następnie w procesie inicjalizacji wybierana jest jedna grupa, która będzie miała zielone światło. Parametr `period` jest wykorzystywany do ustawiania czasu trwania danej fazy. W przypadku zielonego światła jest to jedna wartość `period`, zaś dla świateł czerwonych każda kolejna grupa dróg ma ustawianą wartość `turnLength` przemnożoną przez indeks określający kolejność grupy. Dzięki temu w fazie zmiany światła zwiększamy tylko obecny czas trwania tury, a w momencie, w którym osiągnie on wartość `period` światła są zmieniane. W przypadku tylko jednej wchodzącej drogi światło jest zawsze zielone.
- Algorytm SOTL – Użytkownik podaje dwa parametry: minimalną długość trwania tury oraz parametr Φ_{min}. Podczas inicjalizacji każde światło ustawiane jest na czerwone. Następnie podczas fazy zmiany świateł realizowany jest algorytm widoczny na rysunku 31.

Jak można zauważyć pominięte zostało wykorzystanie parametrów ω_{min} oraz μ opisanych w dziale teoretycznym, głównie ze względu na problem ze zdefiniowaniem kiedy inna droga jest prostopadła do obecnej.

```
private fun switchLight(
    phase: TrafficLightPhase,
    lane: Lane?
) {
    requireNotNull(lane) { "Invalid road setup, expected not null road" }
    when (phase.state) {
        RED -> {
            val duration = phase.phaseTime
            val carsCount = lane.cars.count()
            if (duration * carsCount >= phiFactor && minPhaseLength <= duration) {
                phase.state = GREEN
                // calculate green light length
                val lastCarPosition = lane.cars
                    .minByOrNull { it.positionRelativeToStart }?.positionRelativeToStart ?: 0
                val lengthToEnd = lane.physicalLength - lastCarPosition
                // all cars should go through this green light, so we can limit this by 1 * length to beat
                phase.phaseTime = 0
                phase.period = max(minPhaseLength, lengthToEnd)
            }
        }
        GREEN -> {
            if (phase.phaseTime == phase.period) {
                phase.state = RED
                phase.phaseTime = 0
            }
        }
    }
}
```

Rysunek 31: Algorytm SOTL zmiany świateł.

Walidacja symulacji

Aby upewnić się, że symulacja będzie działać prawidłowo, oraz że będzie możliwe jej uruchomienie, przed jej stworzeniem sprawdzamy jej poprawność. W tym celu sprawdzamy następujące właściwości elementów symulacji:

- Każde skrzyżowanie musi posiadać strategię sterowania światłami. Przy definiowaniu symulacji dla każdego skrzyżowania musimy zdefiniować jaki algorytm będzie sterował światłami oraz parametry używane przez dany algorytm.
- Walidacja generatorów – przy generatorach musimy sprawdzić, czy:
  - Węzeł źródłowy jest różny od węzła docelowego – w przeciwnym przypadku nie możemy wyznaczyć żadnej trasy dla generowanych pojazdów,
  - Z węzła źródłowego da się dojechać do węzła docelowego. W stworzonej siatce dróg może się zdarzyć, że przy konkretnej parze węzłów nie będzie się dało dotrzeć ze źródłowego do docelowego.
- Model Brake Light potrzebuje więcej parametrów niż podstawowe – sprawdzamy, czy jeśli wybrany został model Brake Light to zostały także podane dodatkowe parametry potrzebne do działania tego modelu.

3.1.4. Serwis statystyk

Rolą serwisu statystyk jest zbieranie oraz udostępnianie informacji o jakości ruchu przeprowadzanych symulacji. Dane agregowane są po zasymulowaniu każdej tury, dzięki czemu można z

nich otrzymać informacje na temat trendu przebiegu symulacji. Jego węzły końcowe wyglądają następująco:

1. **GET /statistics/{id}** – Zwraca statystyki o podanym identyfikatorze,

2. **GET /statistics/simulation/{simulationId}** – Zwraca wszystkie statystyki z danej symulacji,

3. **GET /statistics/simulation/{simulationId}/turnRange** – Zwraca wszystkie statystyki z danej symulacji mieszczące się w podanym przedziale turowym,

Poszczególne składowe serwisu odpowiadają za różne funkcjonalności:

- **StatisticsController** – Klasa wystawiająca węzły końcowe serwisu. Korzysta z `StatisticsService` w celu komunikacji z repozytorium danych oraz z `StatisticsMapper` do mapowania encji na obiekty udostępniane przez węzły końcowe.

- **StatisticsMapper** – Interfejs, którego implementację dostarcza biblioteka MapStruct. Służy do parsowania encji do obiektów transportujących dane.

- **StatisticsService** – Klasa służąca do komunikacji z repozytorium statystyk, z którego pozyskuje potrzebne, w zależności od podanej metody, encje.

- **StatisticsRepository** – Interfejs, który jest implementowany przez Springa. Jego rolą jest bezpośrednia komunikacja z bazą danych.

- **StatisticsManaager** – Klasa zbierająca wyniki z każdej tury symulacji. Jej instancja jest dodawana do przeprowadzanej symulacji na etapie parsowania jej z encji. Do tworzenia statystyk wykorzystuje instancję stanu symulacji.

- **StatisticsFactory** – Fabryka odpowiadająca za przygotowanie menadżerów statystyk dla tworzonych instancji symulacji oraz za parsowania wyników zebranych statystyk do odpowiednich encji.

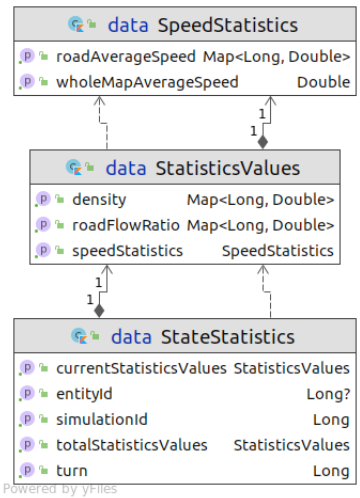
- **SimulationService** – Klasa stanowiąca część serwisu symulacji, ale to w niej dokonuje się zapis statystyk z wykonanej symulacji do bazy danych.

Rysunek 32: Klasy składające się na serwis statystyk.

Jak widać na rysunku 32, schemat serwisu składa się z dwóch rozłącznych diagramów. Podział ten wynika z tego, że w przypadku tego serwisu zapisywanie statystyk po wykonaniu symulacji jest rozłączne z pozyskiwaniem ich z bazy danych po wykonaniu zapytania z klienta. W pierwszym przypadku zapisanie odbywa się poprzez serwis symulacji (encja `SimulationEntity` posiada pole `statisticsEntities` będące listą statystyk z każdej tury symulacji), a w drugim odczyt danych wykonywany jest bezpośrednio z `SimulationRepository`.

Zbierane statystyki

Diagram obiektu odpowiedzialnego za przechowywanie statystyk można znaleźć na rysunku 33.



Rysunek 33: Klasy składające się na serwis statystyk.

- **simulationId** – Identyfikator symulacji, z której pochodzą dane,
- **entityId** – Identyfikator encji statystyk,
- **currentStatisticsValues** – Statystyki z obecnej tury,
- **totalStatisticsValues** – Statystyki na przestrzeni całej symulacji,
- **turn** – Tura, z której statystyki zostały zebrane.

Wartości statystyk z obecnej tury oraz te z całej symulacji mają analogiczną strukturę. Posiadają one dane na temat: globalnej średniej prędkości oraz gęstości; przepływu i średniej prędkości z podziałem na drogę:

- **density** – Gęstość drogi będąca ilorazem liczby samochodów na drodze przez liczbę kratek, z których składa się droga,
- **roadFlowRatio** – Wartość przepływu (ilorazu oczekiwanej wartości prędkości na drodze przez średniej prędkości aut poruszających się po pasach tej drogi),
- **speedStatistics** – Zebrane w jeden obiekt wartości średniej prędkości na każdym pasie oraz średnia prędkość na całej mapie.

W przypadku statystyk na przestrzeni całej symulacji operujemy na średniej arytmetycznej podanych wartości, zamiast na danych z wybranej tury.

Rozszerzalność

Ponieważ menadżer statystyk posiada dostęp do stanów symulacji w każdej wykonanej turze podczas obliczania wartości, to zawartość zapisywanych statystyk może być w łatwy sposób rozszerzona. Zaproponowane przez nas parametry stanowią bazę możliwych do zebrania danych, które mają służyć za przykład podczas dalszego rozwoju serwisu. Największe ograniczenie obecnego mechanizmu spowodowane jest przez założenie produktu (chcieliśmy, aby wszystkie symulacje umieszczone na jednej mapie były między sobą porównywalne). Ograniczeniem tym jest konieczność zbierania dokładnie tych samych parametrów dla każdego rodzaju symulacji.

```
class StatisticsManager(
    var states: List<StateStatistics> = ArrayList(),
    var expectedVelocity: Map<RoadId, Velocity> = HashMap()
) {

    lateinit var latestState: StateStatistics
```

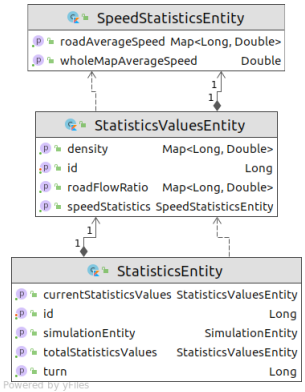
Rysunek 34: Pola, do których ma dostęp `StatisticsManager`. Dzięki temu, że znajduje się tam lista poprzednich stanów, mamy możliwość wyliczania statystyk na przestrzeni całej symulacji.

```
fun createStatistics(state: SimulationState) {
    val roadsSpeed = getRoadsSpeed(state)
    val roadData = getRoadData(state)

    val speedStatistics = speedStatistics(roadsSpeed)
    val density = roadData.associate { it.id to it.carsNumber.toDouble() / it.surface }
    val roadFlowRatio = roadsSpeed.filter { expectedVelocity.containsKey(it.key) }
        .map { (id, carSpeeds) ->
            id to carSpeeds.averageOr( default: 0.0) / expectedVelocity[id]!!
        }.toMap()

    val currentStatisticsValues = StatisticsValues(
        speedStatistics,
        density,
        roadFlowRatio
    )

    val totalStatisticsValues = createTotalStatisticsValues(currentStatisticsValues)
    val currentState = StateStatistics(state.id, state.turn, currentStatisticsValues, totalStatisticsValues)
    states += currentState
    latestState = currentState
}
```

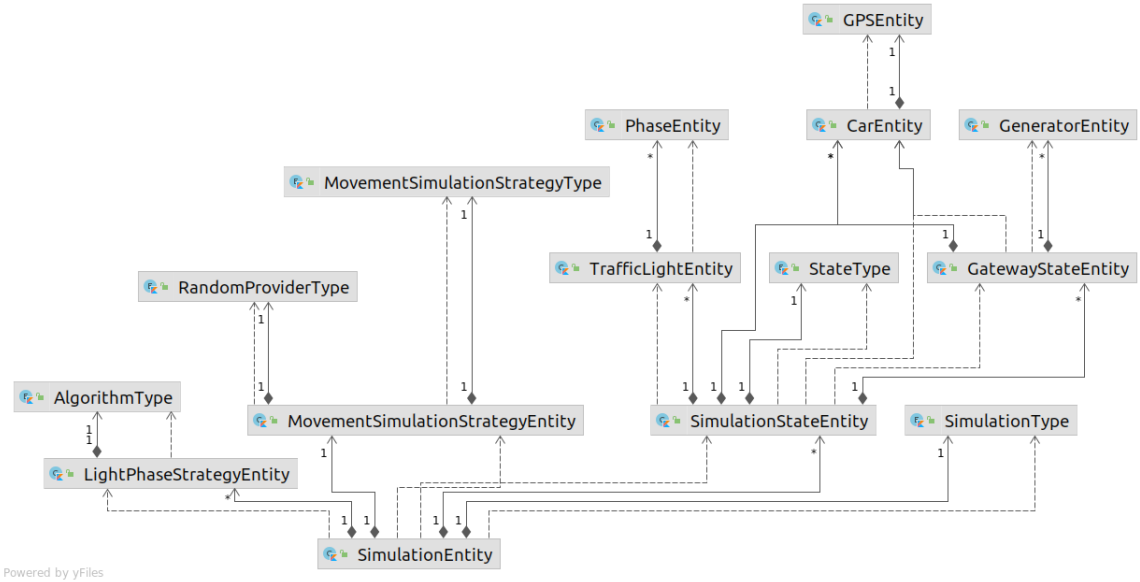
Rysunek 35: Metoda klasy `StatisticsManager` odpowiadająca za tworzenie statystyk. W celu zbierania nowych danych należy zaimplementować w niej pożądaną funkcjonalność.

3.1.5. Warstwa persystencji

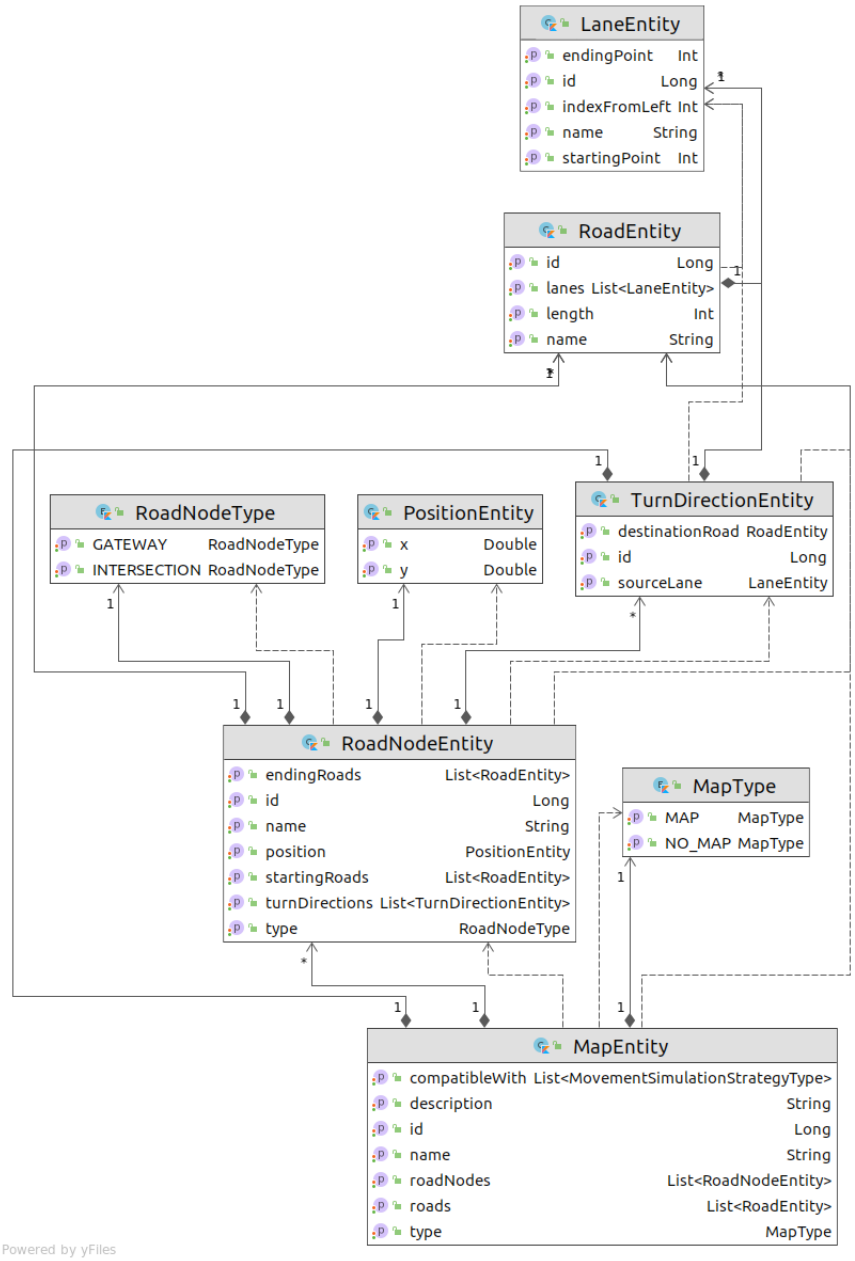
W kontekście warstwy persystencji nasz model wyróżnia 3 obiekty: mapę, symulację oraz statystykę. Dzięki zdefiniowaniu odpowiednich interfejsów dla każdego rodzaju symulacji oraz mapy nie jest konieczne stosowanie dziedziczenia encji – co zdecydowanie upraszcza i tak skomplikowany schemat bazy danych. Jak było to wspomniane w rozdziale Architektura Serwera, w paru miejscach używamy własnego konwertera danych między obiektem a encją w bazie danych. Jest to spowodowane tym, że aplikacja szybko produkuje dużą ilość rekordów, a darmowe rozwiązania umożliwiające hosting często ograniczają ilość rekordów. Z powodu dużego zagnieżdżenia obiektów w przypadku symulacji oraz mapy, przy modelowaniu relacji jeden do wielu zawsze używaliśmy adnotacji `@OneToMany` z opcją `cascade = [CascadeType.ALL]`. Umożliwiło nam to wygodne umieszczanie oraz usuwanie rozbudowanych obiektów.



Rysunek 36: Diagram UML obiektów reprezentujących encje statystyk.



Rysunek 37: Diagram UML obiektów reprezentujących encje symulacji.



Rysunek 38: Diagram UML obiektów reprezentujących encje mapy.

3.1.6. Testy

Na działającą symulację składa się bardzo dużo elementów. Aby zapewnić poprawność implementacji, wraz z rozwojem aplikacji do każdej nowej większej funkcjonalności dodawane były testy. Znaczną część z nich stanowią testy jednostkowe, sprawdzające takie funkcjonalności jak:

- **Strategie poruszania** – Podstawowa strategia jednopasmowa Nagela-Schreckenberga została dokładnie przetestowana, sprawdzając pod względem poprawności wyniku każdego kroku z osobna. Sprawdzone zostało również wielopasmowe rozszerzenie oraz strategia Brake Light,
- **Generator samochodów** – Sprawdzona została poprawność generowania samochodów przez jeden lub kilka generatorów,

- **GPS** – Testy klasy odpowiadającej za obliczanie optymalnej trasy, w tym przypadku najkrótszej pod względem długości drogi,

- **Strategie zmiany świateł** – Testy różnych przypadków zachowań dla strategii turowej oraz SOTL,

- **Manager statystyk** – Na podstawie stanu symulacji sprawdzana była poprawność wartości wygenerowanych statystyk.

Adnotacja `@Test` zaznacza funkcję, która powinna być uruchomiona w formie testu. Funkcja ta ma spełniać warunku asercji podane (najczęściej) na końcu. Adnotacja `@SpringBootTest` mogąca się znaleźć nad klasą, oznacza, że zostaje ona uruchomiona w kontekście frameworku Spring. Klasy, których instancje powinien zapewnić mechanizm wstrzykiwania zależności, podane są w kwadratowych nawiasach.

```
@SpringBootTest(classes = [DelayGatewayCarGenerator::class, GpsFactory::class, RoadLengthGPS::class])
internal class DelayGatewayCarGeneratorTest {

    @Autowired
    lateinit var generator: DelayGatewayCarGenerator

    @Test
    fun `Given gateway with generator, when generate put correct car`() {
        // given
        val state = getOneRoadSimulationState()
        val gateway = state.gateway( id: 0)
        gateway.generators = listOf(
            Generator(
                lastCarReleasedTurnsAgo = 0,
                releaseDelay = 0,
                carsToRelease = 1,
                targetGatewayId = 1,
                gpsType = GPSType.DIJKSTRA_ROAD_LENGTH
            )
        )

        // when
        generator.generate(state)

        // then
        val cars = state.road( id: 0).lanes[0].cars
        assertThat(cars.size).isEqualTo(1)
        cars[0].assertVelocity(0)
            .assertPositionRelativeToStart(0)
            .assertGpsRouteIsEmpty()
            .assertGpsType(GPSType.DIJKSTRA_ROAD_LENGTH)
    }
```

Rysunek 39: Test generatora samochodów.

Oprócz testów jednostkowych znajdują się także testy integracyjne. W przeciwieństwie do testów jednostkowych sprawdzają one większe zależności, które angażują kilka klas. Aby jak najlepiej zasymulować prawdziwe działanie aplikacji, wykorzystana została biblioteka Testcontainers, która na czas testów zapewnia nam kontenery dockerowe [22] z uruchomioną instacją bazy danych wygenerowaną przez Spring. Na czas każdego z testów baza jest stawiana na nowo – eliminuje to przypadki, kiedy na wynik testów mogłyby wpływać zapisane przez poprzednie testy dane. W ten sposób zostały przetestowane następujące funkcjonalności:

- **Fabryka mapy** – Test polega na: przygotowaniu odpowiednich encji bazodanowych związanych z mapą, zapisaniu ich, dokonaniu odczytu z bazy danych, sparsowaniu danych przez fabrykę i sprawdzeniu ich poprawności,

- **Fabryka stanu symulacji** – W tym teście na początku przygotowujemy przykładową symulację i zapisujemy ją do bazy danych, następnie odczytujemy encje i sprawdzamy poprawność parsowania obiektu encji przez fabrykę stanu symulacji,

- **Serwis symulacji** W przypadku tego testu również przygotowujemy przykładową symulację i zapisujemy ją do bazy danych, następnie korzystając z serwisu symulacji, przeprowadzamy symulację jednej tury, po jej wykonaniu sprawdzamy poprawność zapisanych do bazy encji,

- **Test mappera** – Test polega na: pobraniu zapisanych wcześniej do bazy encji oraz sprawdzeniu poprawności parsowania ich na obiekty transferu danych przez implementację interfejsu mappera dostarczoną przez bibliotekę MapStruct [23].

```kotlin
@Testcontainers
@SpringBootTest
@Transactional
@EnableAutoConfiguration
@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)
internal class SimulationServiceTest @Autowired constructor(
    val simulationRepository: SimulationRepository,
    val simulationService: SimulationService,
    val mapRepository: MapRepository,
    val carRepository: CarRepository,
) {

    companion object {
        @Container
        private val postgreSQLContainer = PostgreSQLContainer<Nothing>( dockerImageName: "postgres:latest")

        @DynamicPropertySource
        @JvmStatic
        fun registerDynamicProperties(registry: DynamicPropertyRegistry) {
            registry.add( name: "spring.datasource.url", postgreSQLContainer::getJdbcUrl)
            registry.add( name: "spring.datasource.username", postgreSQLContainer::getUsername)
            registry.add( name: "spring.datasource.password", postgreSQLContainer::getPassword)
        }
    }

    fun createTestSimulation(): Long {...}

    @Test
    fun `Given amount of turns in a simulation, check if amount of CarEntities representing one car is equal to it`() {
        // given
        val simulationId = createTestSimulation()
        // when
        simulationService.simulateStep(simulationId)
        // then
        val count = carRepository.findCarEntitiesByCarId( carId: 1).count()
        assertThat(count).isEqualTo(2)
    }
```

Rysunek 40: Test serwisu symulacji. Zawartość obiektu towarzyszącego to utworzenie instancji kontenera z bazą danych oraz ustawienie połączenia w ustawieniach aplikacji. Funkcja tworząca encje została ukryta ze względu na jej rozmiar.

3.2. Architektura klienta

Interfejs graficzny służący do obsługi środowiska symulacyjnego został zrealizowany w formie aplikacji webowej. Dzięki temu użytkownik nie musi przechodzić przez proces instalacji przy pierwszym uruchamianiu. Ponadto, aplikacja webowa daje możliwość łatwego wprowadzania zmian przez deweloperów.

3.2.1. Stos technologiczny

Główne narzędzia

Ta część naszego projektu została zbudowana przy użyciu biblioteki React [1] w połączeniu z językiem TypeScript [13]. Podjęliśmy taką decyzję, ponieważ React jest przodującym rozwiązaniem na rynku, a my mieliśmy już doświadczenie w budowaniu aplikacji z pomocą tej biblioteki. Podczas wyboru języka programowania, początkowo, planowaliśmy użyć samego JavaScriptu, ale z uwagi na to, że model danych koniecznych do wizualizacji jest skomplikowany, dodaliśmy do projektu TypeScript, co dało nam możliwość zauważenia części błędów na poziomie kompilacji. Aplikacja została stworzona za pomocą narzędzia create-react-app [40] a następnie, poprzez komendę `eject`, została zamieniona na konfigurację webpackową [36]. Architektura w ten sposób stworzonego klienta realizuje wzorzec Single Page Application [33].

Szata graficzna

Szata graficzna aplikacji jest stworzona za pomocą komponentów będących częścią biblioteki Material UI [3] (w tym wypadku również zdecydowaliśmy się na znane nam już rozwiązanie). Stylowanie komponentów wykonujemy za pomocą biblioteki emotion [41], służącej do definiowania arkuszy stylu z poziomu JavaScriptu.

Zarządzanie danymi

Ponieważ na różnych widokach aplikacja operuje na tych samych danych, to zdecydowaliśmy się zastosować scentralizowany model ich przechowywania.Wykorzystaliśmy do tego bibliotekę Redux [4] oraz zestaw dodatkowych narzędzi do jej obsługi – Redux-Toolkit [24].

Wizualizacja danych

Mapy, na których odbywa się symulacja, można przedstawić w uproszczonej postaci jako grafy z krawędziami skierowanymi o dwóch rodzajach wierzchołków. Do rysowania takich grafów użyliśmy biblioteki vis.js [39]. Dane ze statystyk zdecydowaliśmy się przedstawiać na standardowych wykresach – głównie słupkowych i liniowych. Do ich stworzenia wykorzystaliśmy bibliotekę react-vis [38]

Formularze

Podczas rozwijania aplikacji bardzo uciążliwa okazała się praca z rozbudowanymi formularzami (w szczególności tym odpowiadającym za utworzenie nowej symulacji). W tym przypadku pomogła nam biblioteka Formik [2], która ułatwia śledzenie wprowadzanych danych oraz zatwierdzanie formularza. Działa ona na zasadzie kontekstu [6], co powodowało problemy z wydajnością (zbyt częste rendery). Udało nam się jednak zneutralizować ten efekt, stosując memoryzację komponentów składowych formularzy.

Nawigacja

Do nawigowania pomiędzy stronami wykorzystujemy bibliotekę react-router [42]. Jest to najpopularniejsza biblioteka w ekosystemie Reacta do tego zadania, więc ponownie zdecydowaliśmy się użyć sprawdzonej technologii

3.2.2. Zarządzanie stanem danych z serwera

Pobieranie, aktualizowanie oraz synchronizowanie danych przechowywanych na kliencie z danymi z serwera było kluczowym zagadnieniem, ponieważ to na danych opierała się cała aplikacja. Dzięki zestawowi narzędzi z biblioteki Redux-Toolkit udało się zrealizować to w bardzo praktyczny sposób bez konieczności pisania powtarzalnego kodu. Najbardziej przydatna okazała się funkcja `createApi` pozwalająca zdefiniować (na podstawie węzła końcowego oraz systemu tagów), jak i kiedy należy pobierać, aktualizować, oraz zapisywać dane. Za jej pomocą zdefiniowaliśmy 2 kluczowe zbiory danych: `simulationApi` oraz `mapApi`.

```
export const mapApi = createApi({
  reducerPath: 'mapApi',
  baseQuery: fetchBaseQuery( {baseUrl, prepareHeaders, fetchFn, ...baseFetchOptions}: {
    baseUrl: process.env.REACT_APP_API_URL || 'http://localhost:8080/',
  }),
  tagTypes: ['Map', 'BasicMap'],
  endpoints: (builder :EndpointBuilder<BaseQuery, TagTypes, ReducerPath> ) => ({
    getMapById: builder.query<SimulationMap, number>( definition: {
      query: (id :QueryArg ) => ({ url: `map/${id}` }),
      providesTags: (result :ResultType | undefined ) =>
        result ? [{ type: 'Map', id: result.id }, 'Map'] : ['Map'],
    }),
    getAllMapsBasicInfo: builder.query<BasicMapInfo[], void>({...}),
    createMap: builder.mutation<SimulationMap, CreateMapRequest>( definition: {
      query: (request :QueryArg ) => ({
        url: 'map/create',
        method: 'POST',
        body: request,
      }),
      invalidatesTags: (result :ResultType | undefined ) =>
        result ? [{ type: 'Map', id: result.id }, 'Map'] : ['Map'],
    }),
    getBasicMapById: builder.query<BasicMapInfo, number>({...}),
    validateMap: builder.mutation<ErrorWrapper<BasicMapInfo>, CreateMapRequest>({...}),
  }),
});
```

Rysunek 41: Obiekt `mapApi` utworzony za pomocą funkcji `createApi`. Odpowiada za dane związane z mapami.

W powyższych interfejsach można wyróżnić dwa rodzaje metod: zapytanie oraz mutacja. Każde zapytanie dostarcza dane o tagach określonych w parametrze `providesTags`, a każda mutacja dezaktualizuje dane o tagach określonych w `invalidatesTag`. Na tej podstawie biblioteka decyduje, kiedy należy uaktualnić dane, dzięki czemu uzyskujemy automatyczną aktualizację. Żeby uzyskać dostęp do danych w komponencie, należy jedynie wywołać odpowiedniego hooka [5] udostępnianego przez wynik wywołania funkcji.

```
const { data, isLoading, error } =
  useGetStatisticsFromSimulationQuery(selectedSimulationId);
```

Rysunek 42: Wywołanie hooka, który dostarcza dane statystyk z danej symulacji. Proces ten działa na zasadzie subskrypcji, która rozwiązuje problem zdezaktualizowania danych. Hook zapewnia również pomocnicze zmienne określające, czy zapytanie jest w trakcie realizacji oraz, czy jego wykonanie się powiodło.

4. Organizacja pracy

4.1. Charakterystyka zadania

Jednym z wymagań naszego klienta było zaprojektowanie łatwej do rozszerzania aplikacji. Projekt ten jest nową implementacją istniejącego już systemu Kraksim, rozwijanego przez poprzednie kilkanaście lat. Ponieważ jego największym problemem była skomplikowana architektura, początkowe etapy projektu realizowaliśmy wspólnie, z dużym naciskiem na dobre rozplanowanie zależności. Aplikacja przeznaczona jest do dalszego rozwijania w przyszłości

4.2. Osoby związane z projektem oraz podział obowiązków

Naszym Klientem oraz opiekunem pracy był dr hab. inż. Jarosław Koźlak, a po Jego śmierci rolę tę przejął dr hab. inż. Rafał Dreżewski.

Jak już wcześniej wspomniano, część prac realizowana była wspólnie (dotyczy to zwłaszcza początkowych etapów projektu). Zaimplementowane funkcjonalności, którym trudno przypisać jednego twórcę, zostały ujęte osobno (jako wspólna praca). Podział wygląda następująco:

Wspólna praca

- Analiza poprzedniego systemu,
- Projektowanie schematu,
- Wstępna implementacja abstrakcji strategi ruchu,
- Stworzenie podstaw symulacji Nagela-Schreckenberga
- Wstępna implementacja modułu zbierania statystyk.

Wiktor Kamiński

- Finalizacja modułu statystyk,
- Przygotowanie projektu aplikacji klienta + CI/CD,
- Algorytm sterowania światłami SOTL,
- Testy integracyjne serwisu symulacji,
- Wizualizacja map,

- Widok wszystkich symulacji,
- Widok listy map wraz z wizualizacją,
- Formularz tworzenia symulacji,
- Widok statystyk,
- Widok tworzenia mapy,
- Poprawki wizualne.

Grzegorz Poręba

- Przygotowanie projektu serwera,
- Menedżer zarządzania światłami + algorytm turowy,
- Moduł generatora aut oraz GPS (obliczanie trasy auta algorytmem Dijkstry),
- Serwis symulacji + zapisywanie danych do bazy danych,
- Mechanizm obsługi błędów i walidacja zapytań,
- Widok wszystkich symulacji,
- Widok tworzenia mapy,
- Wielopasmowa strategia ruchu Nagela-Schreckenberga
- Przygotowanie API,
- Poprawki wizualne.

Miłosz Galas

- Finalizacja modułu statystyk,
- Poprawki w podstawowej strategi ruchu
- Serwis symulacji wraz z mapowaniem encji na obiekty transferu danych,
- Strategia ruchu Brake Light,
- Poprawki w strategiach sterowania światłami.

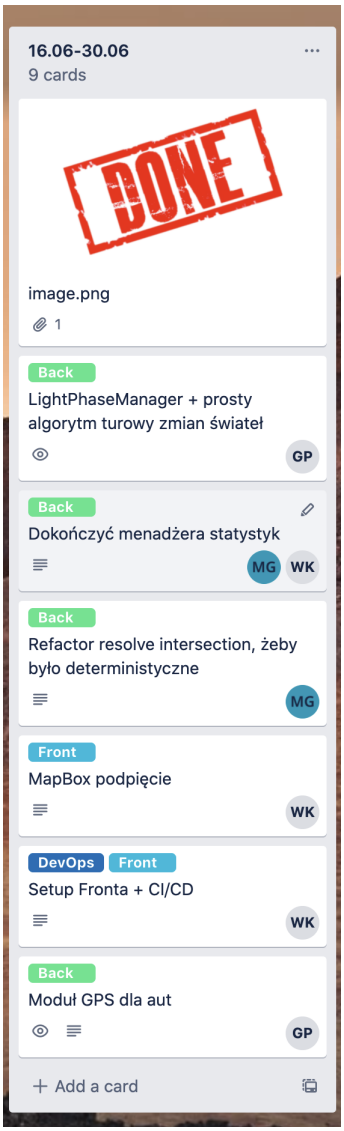
4.3. Zastosowane techniki pracy w grupie

Początkowo, przygotowując wizję pracy, korzystaliśmy ze strony hackmd.io, gdzie w szybki sposób można wspólnie notować i pracować nad wspólnym tekstem. Później wykonana praca została przeniesiona do narzędzia do grupowego edytowania na platformie overleaf.com [25], już w formacie pracy inżynierskiej w LaTexie [26].

Narzędzie do organizacji pracy

Do efektywnej organizacji i podziału pracy wykorzystaliśmy aplikację internetową Trello. Rozpisany został tam plan prac z podziałem na poszczególne miesiące. Dla każdego miesiąca dodana została kolumna z zadaniami, które chcemy w nim zrealizować. Następnie utworzone zostały 3 podstawowe kolumny, kolejno odpowiadające stanowi zadań: ‘do zrobienia‘, ‘w trakcie‘ oraz ‘ukończone‘. Stosując uproszczone techniki metodologii SCRUM przy wykorzystaniu tablicy Kanban, podzieliliśmy naszą pracę na 2 tygodniowe sprinty. Na początku każdego sprintu spotykaliśmy się, aby dodać zadania do pierwszej kolumny i przypisać osobę do realizacji każdego z nich. W celu zachowania historii przebiegu całego procesu, po każdym sprincie przenosiliśmy skończone zadania do kolumny z przedziałem dat, w którym był on realizowany. Każde z zadań posiadało także jedną lub więcej etykiet odpowiadających następującym dziedzinom:

- ‘Front‘ – Klient,
- ‘Back‘ – Serwer,
- ‘DevOps‘ – Zadania związane z automatyzacją procesów testowania kodu oraz wystawianiem aplikacji na środowiska testowe,
- ‘Non-tech‘ – Dokumentacja pracy oraz jej postępu (pisanie pracy lub raportów na pracownię projektową).



Rysunek 43: Przykładowy sprint wraz z zadaniami.

4.4. Narzędzia wykorzystane podczas procesu

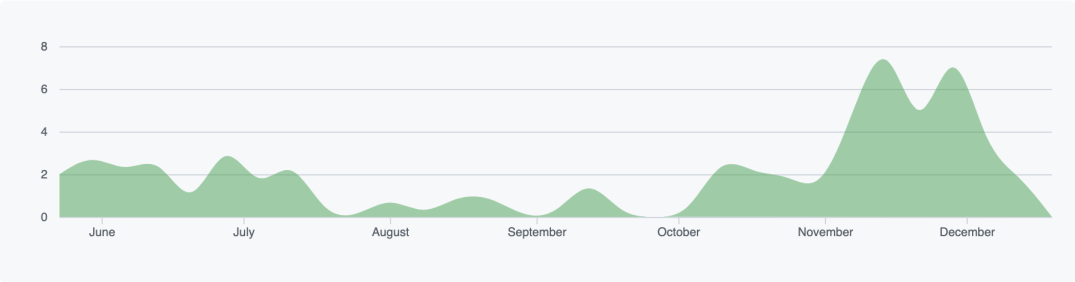
Komunikacja

Głównym narzędziem służącym nam do porozumiewania się, był komunikator internetowy Signal. Tam wymienialiśmy się codziennymi informacjami, planowaliśmy spotkania i podział prac. Materiały, do których chcieliśmy mieć szybki dostęp (prace magisterskie realizowane na bazie starego systemu, schematy utworzone przez nas, linki do artykułów na temat algorytmów do symulowania ruchu oraz używanych przez nas bibliotek), przechowywaliśmy na platformie Discord [27]. Utworzony na niej dedykowany serwer pozwalał nam także na wspólne spotkania. Z promotorem kontaktowaliśmy się mailowo lub poprzez spotkania na platformie Teams.

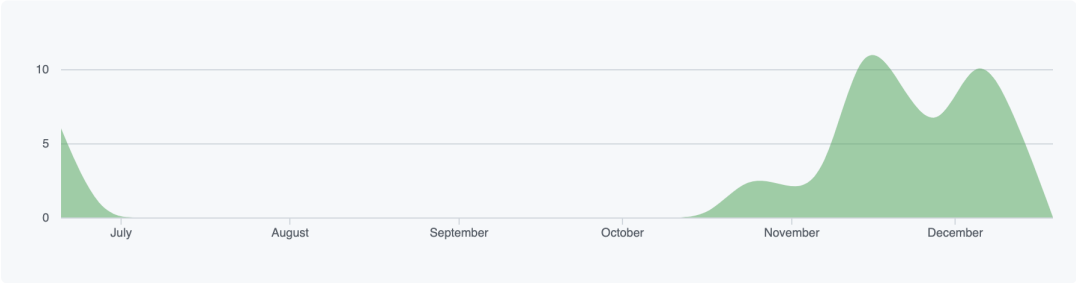
Repozytorium kodu źródłowego

Na platformie GitHub znajduje się organizacja Kraksim, do której dodano trzy repozytoria. Pierwsze repozytorium zawiera kod starego systemu. Jest on łatwo dostępny w razie potrzeby

referencji. W drugim, właściwym repozytorium, znajduje się kod serwera [35]. Trzecie repozytorium mieści w sobie kod klienta [37].



Rysunek 44: Wykres pokazujący częstotliwość dodawania kodu do głównej gałęzi w projekcie serwera.

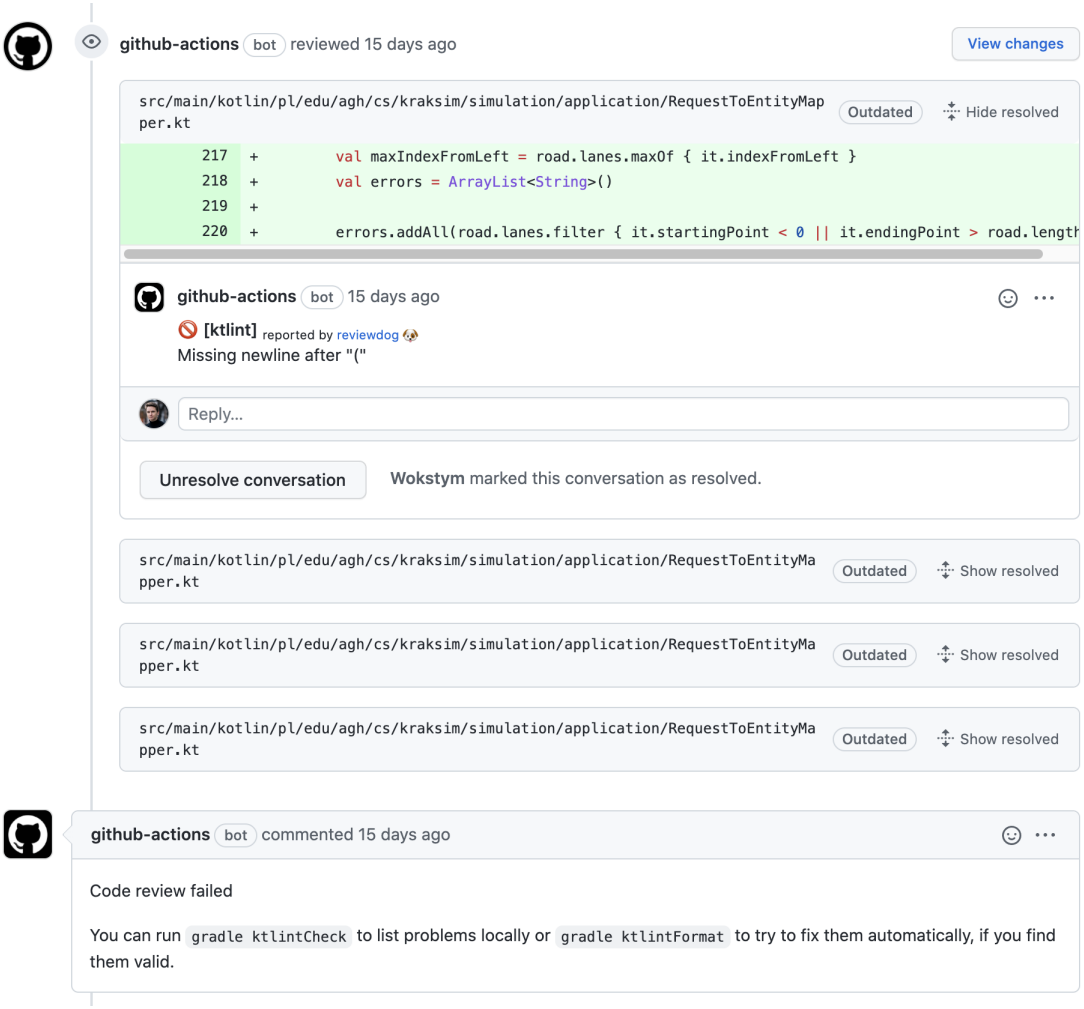


Rysunek 45: Wykres pokazujący częstotliwość dodawania kodu do głównej gałęzi w projekcie klienta.

CI/CD

Aby utrzymać dobrą jakość kodu, zostały dodane Github Actions, czyli skrypty uruchamiające się przy określonych działaniach.

Dla repozytorium serwera przy każdym utworzeniu zapytania o scalenie kodu oraz po scaleniu kodu do głównej gałęzi uruchamiana jest kompilacja oraz testy znajdujące się w projekcie. Drugi skrypt skanuje kod i, korzystając z biblioteki ktlint [28], wykrywa miejsca w kodzie, które są źle sformatowane, bądź mają oczywiste niepotrzebne fragmenty (nieużywany import, średniki). Skrypt ten komentuje też miejsca, które wymagają poprawy. W przypadku wystąpienia problemów w którymś ze skryptów wyświetlana jest informacja o błędzie. Traktujemy ją jako wskazówkę i zezwalamy dalej na dodanie zmian do głównej gałęzi repozytorium.

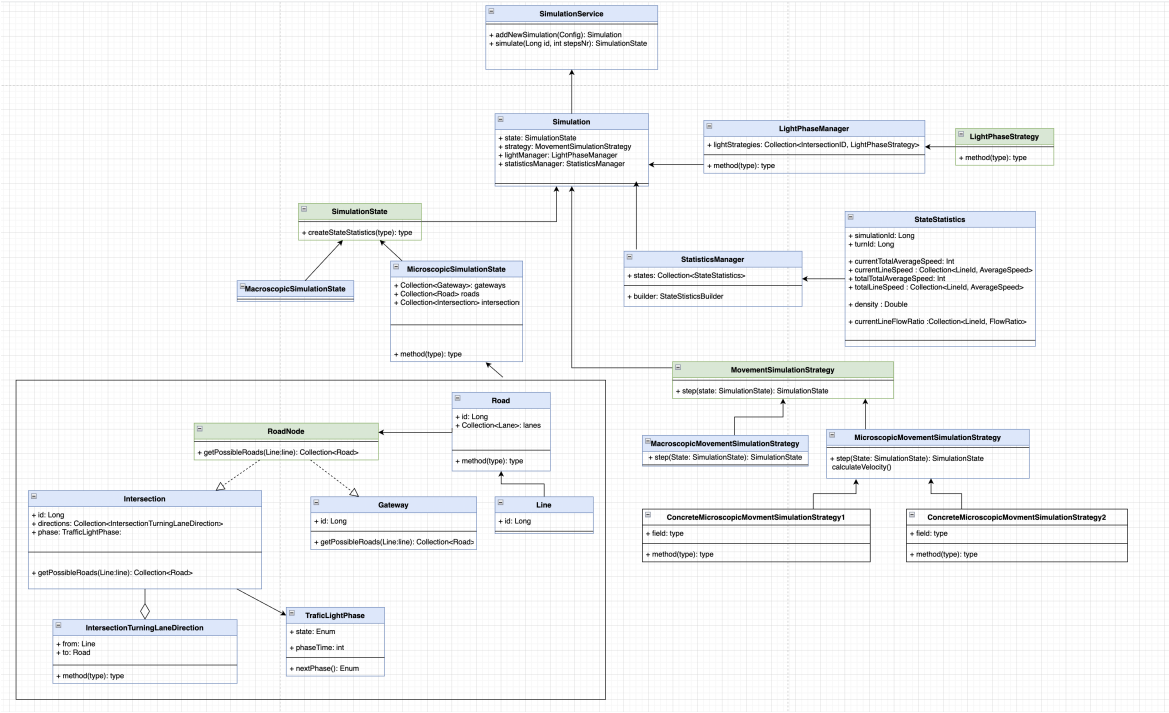


Rysunek 46: Przykładowy komentarz zostawiony przez bota pokazujący błędy w formatowaniu.

Oba repozytoria posiadają również akcje, które po wykonaniu scalenia kodu do głównej gałęzi, wdrażają kod na produkcję. W przypadku serwera kod jest hostowany na platformie heroku.com, natomiast kod klienta uruchomiony jest na platformie vercel.com.

Rysunki koncepcyjne/diagramy

Fazę projektowania architektury serwera rozpoczęliśmy od rysunków ogólnej struktury UML. Wykorzystaliśmy do tego aplikację internetową draw.io [29], gdzie przygotowywaliśmy diagramy przez pierwsze tygodnie. Diagramy UML znajdujące się w tej pracy są wygenerowane przez narzędzie wbudowane w zintegrowane środowisko programistyczne Intellij IDEA [30]. Uproszczone rysunki obrazujące sytuacje spotykające auta na drodze lub skrzyżowaniach były rysowane w programie Paint [31].



Rysunek 47: Diagram serwera narysowany przy użyciu draw.io. Na jego podstawie implementowany był serwer.

5. Wyniki projektu

5.1. Podsumowanie zaimplementowanych funkcjonalności

Algorytmy

Udało nam się zaimplementować trzy strategie symulowania ruchu: podstawowego Nagela-Schreckenberga oraz dwie jego modyfikacje: Brake Light i model wielopasmowy. W przypadku sterowania sygnalizacją świetlną użytkownik ma na ten moment do wyboru 2 algorytmy: turowy oraz SOTL. Ta liczba algorytmów, w połączeniu z innymi możliwymi do konfiguracji parametrami symulacji, umożliwia tworzenie różnorodnych scenariuszy z widocznymi różnicami w przebiegu.

Interfejs graficzny

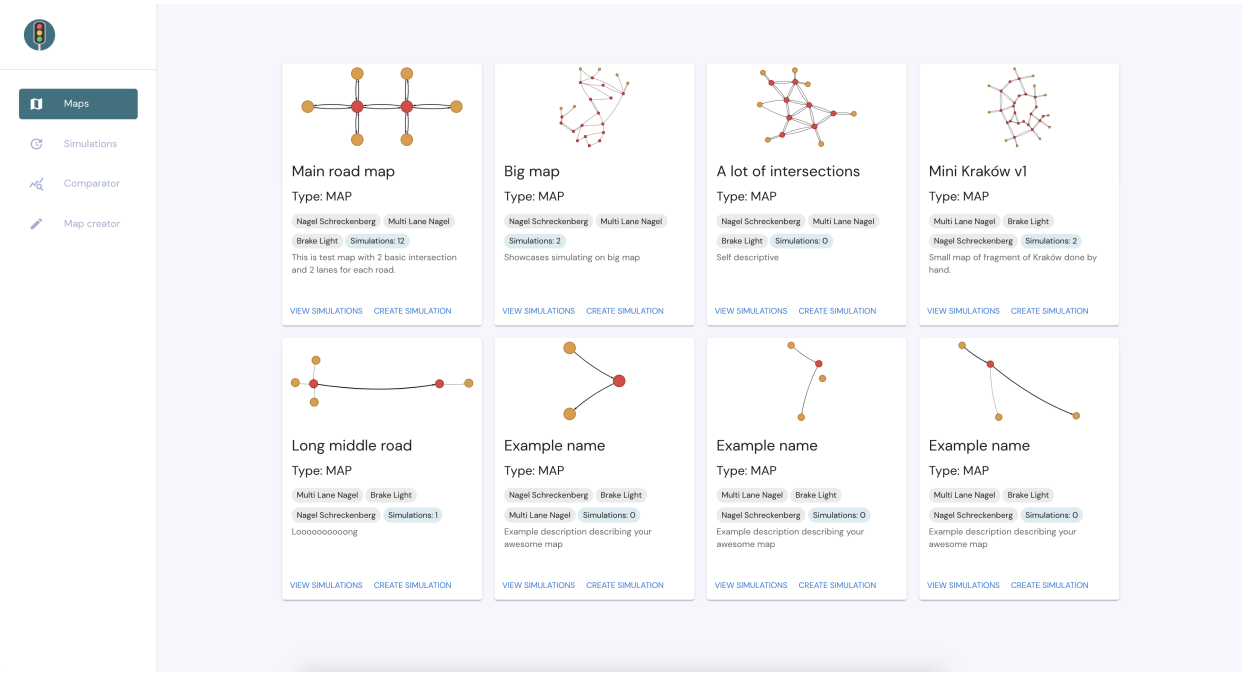
W aplikacji klienta użytkownik ma możliwość: tworzenia map, tworzenia symulacji, przeprowadzania symulacji, oglądania jej wyników i porównywania ze sobą symulacji przeprowadzonych na tej samej mapie. Ten zakres funkcjonalności pozwala na skorzystanie z każdego elementu aplikacji w wygodny dla użytkownika sposób.

5.1.1. Przegląd interfejsu graficznego i scenariuszy działania

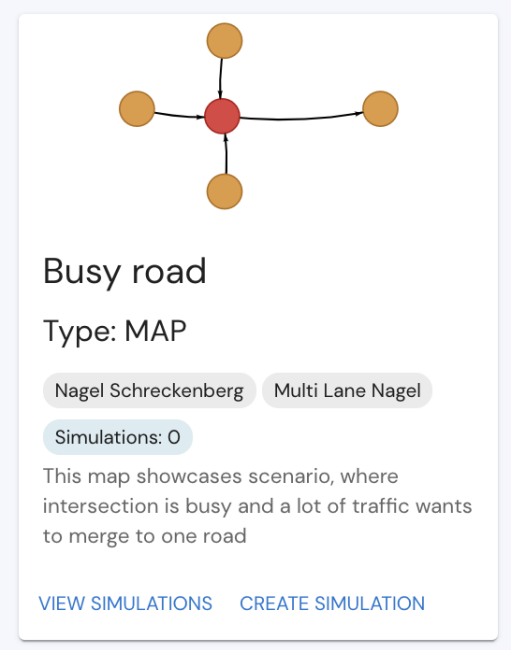
Widok map

Widok map składa się z listy kart reprezentujących mapy. Każda karta zawiera miniaturowy obraz struktury mapy, nazwę, opis, liczbę symulacji utworzonych na mapie oraz nazwy

kompatybilnych algorytmów sterowania ruchem drogowym. Każda karta udostępnia możliwość dwóch akcji: utworzenia symulacji na danej mapie oraz przejście na widok listy symulacji odbywających się na wybranej mapie.



Rysunek 48: Widok listy map dla przykładowych danych.

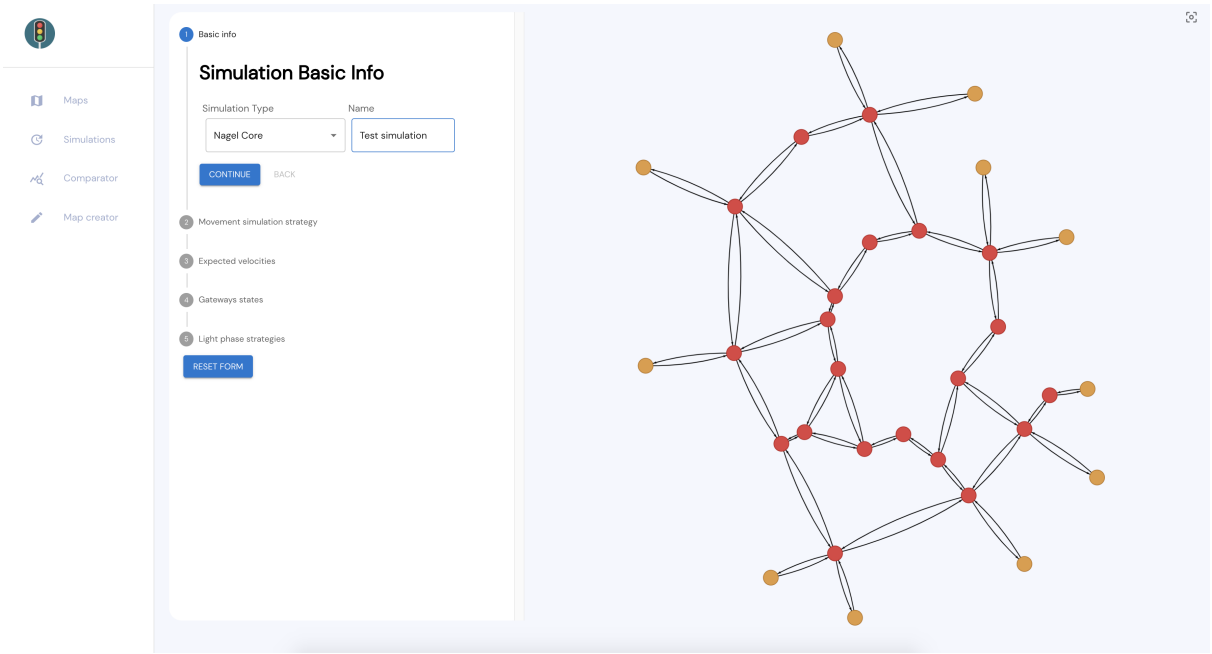


Rysunek 49: Przykładowa karta reprezentująca mapę. Składa się z uproszczonej wizualizacji, tytułu, opisu, listy kompatybilnych algorytmów do sterowania ruchem, liczby stworzonych symulacji oraz dwóch przycisków: `View simulations` – przycisk przenoszący na widok listy symulacji dla wybranej mapy, `Create simuilation` – przycisk przenoszący na widok tworzenia nowej symulacji.

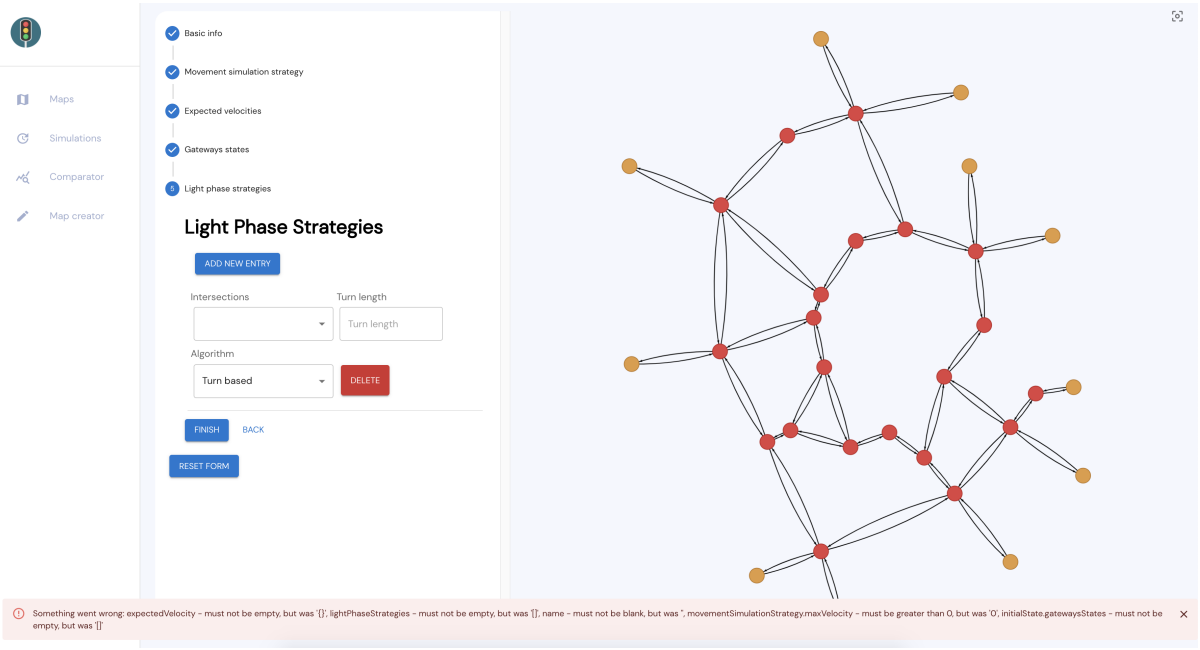
Widok tworzenia symulacji

Głównym komponentem widoku tworzenia symulacji jest formularz składający się z 5 kroków oraz rysunku wybranej mapy. W celu utworzenia symulacji użytkownik musi wypełnić każdy z kroków formularza następującymi danymi:

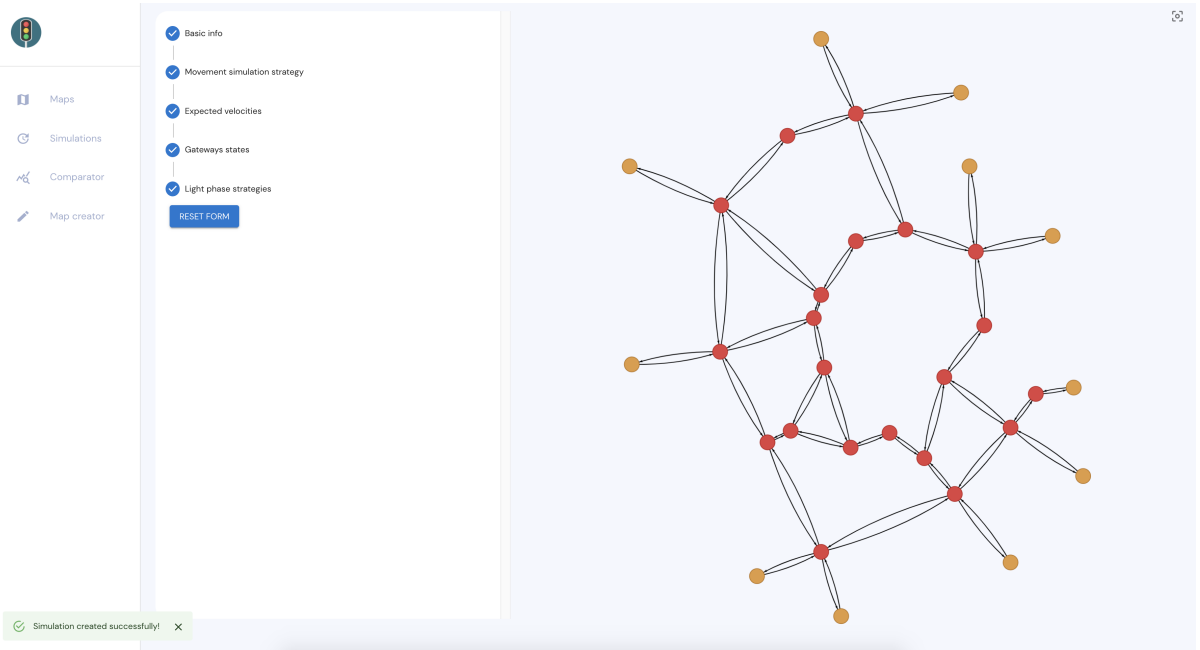
- `Basic Info` – Podstawowe informacje o symulacji: nazwa oraz jej rodzaj,
- `Movement simulation strategy` – Typ strategii symulowania ruchu oraz potrzebne dla niej dodatkowe parametry (np. dla algorytmu Nagel-Schreckenberg ustawiamy dodatkowo maksymalną prędkość samochodu oraz prawdopodobieństwo zwolnienia),
- `Expected velocities` – Oczekiwana prędkość na danej drodze. Służy do wyliczenia wartości przepływu podczas tworzenia obiektów statystyk. W celu ograniczenia ilości dodawanych rekordów jedną wartość prędkości można przypisać do kilku dróg,
- `Gateways states` – Stan początkowy bram. W tym formularzu określamy, z jakimi generatorami zaczyna każda brama. Dla każdej z nich możemy dodać nieograniczoną ilość generatorów. Każdy z nich składa się z ilości samochodów do wypuszczenia, bramy docelowej, opóźnienia w wypuszczaniu samochodów oraz rodzaju GPSa,
- `Light phase strategies` – Strategie kierowania światłami. Dla wybranej grupy skrzyżowań wybieramy algorytm sterowania światłami oraz powiązane z nim dodatkowe parametry (np. długość tury dla podstawowej strategii turowej).



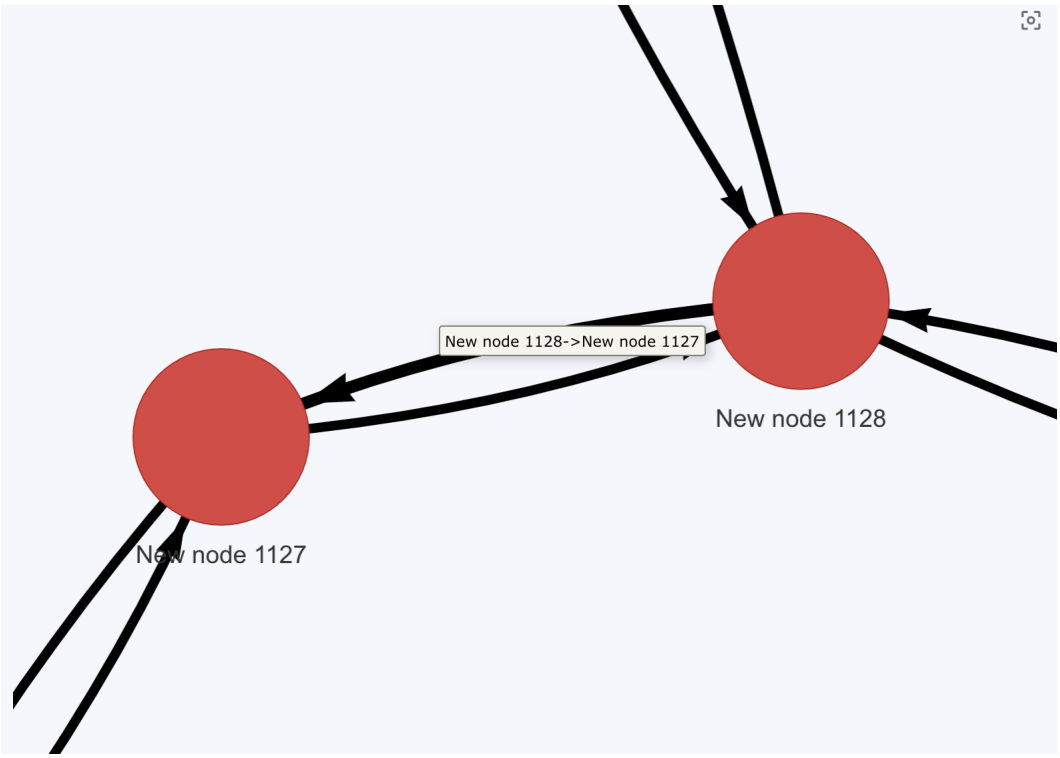
Rysunek 50: Widok tworzenia symulacji. Formularz pozwalający stworzyć symulację składa się z pięciu kroków. Po wypełnieniu danych otrzymamy wiadomość o udanym bądź nieudanym stworzeniu symulacji wraz z ewentualną listą błędów. W celu łatwiejszej orientacji dodaliśmy podgląd mapy po prawej stronie.



Rysunek 51: Widok tworzenia symulacji po błędnym wypełnieniu formularza. Przykład wiadomości zwracającej błędy – w tym wypadku został wysłany pusty formularz.



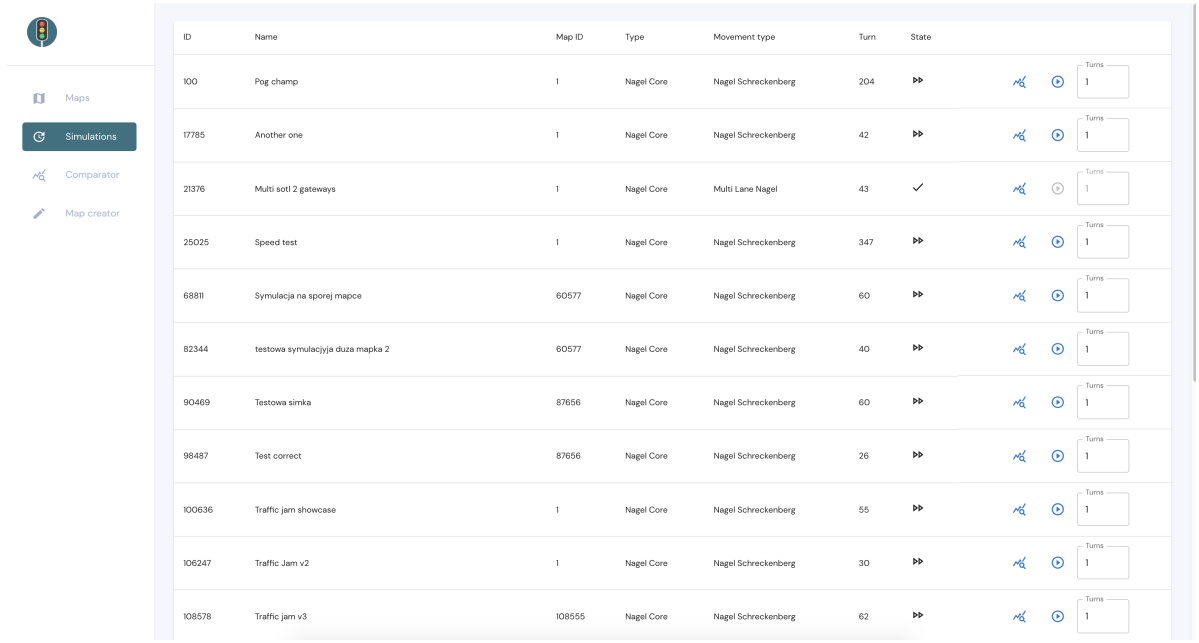
Rysunek 52: Widok tworzenia symulacji po wysłaniu poprawnego zapytania.



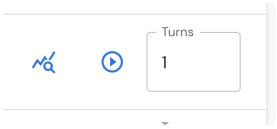
Rysunek 53: Przybliżona mapa – po najechaniu na jej krawędź możemy zobaczyć nazwę drogi przez nią reprezentowanej, co ułatwia proces wypełniania formularza. Za pomocą przycisku w prawym górnym rogu możemy wycentrować mapę.

Widok listy symulacji

Widok listy symulacji to tabela zawierająca podstawowe informacje o każdej z symulacji. Możemy tam znaleźć: identyfikator symulacji, nazwę, identyfikator mapy, typ symulacji, rodzaj strategi ruchu, obecną turę oraz stan ukończenia symulacji. Każda z symulacji posiada także dwa przyciski oraz pole wejściowe. Pierwszy z nich przenosi użytkownika na widok statystyk dla tej symulacji. Drugi przycisk oraz pole wejściowe służą do przeprowadzania symulacji. Do pola należy wpisać, ile tur ma się wykonać, a następnie kliknąć przycisk w celu wysłania zapytania. Po udanym przeprowadzeniu symulacji otrzymamy wiadomość potwierdzającą. Dodatkowo istnieje drugi wariant tego widoku. Kiedy zostanie wybrany przycisk `View simulations`, na widoku map zostanie wyświetlona tabela zawierająca tylko symulacje odbywające się na danej mapie.



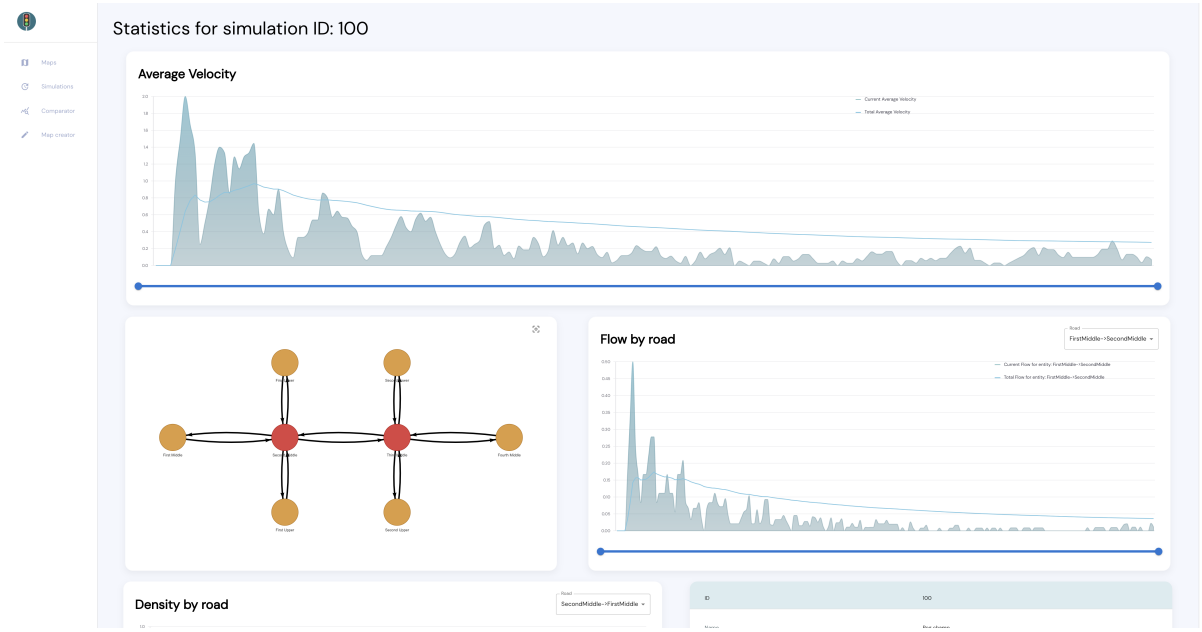
Rysunek 54: Widok listy map dla przykładowych danych.



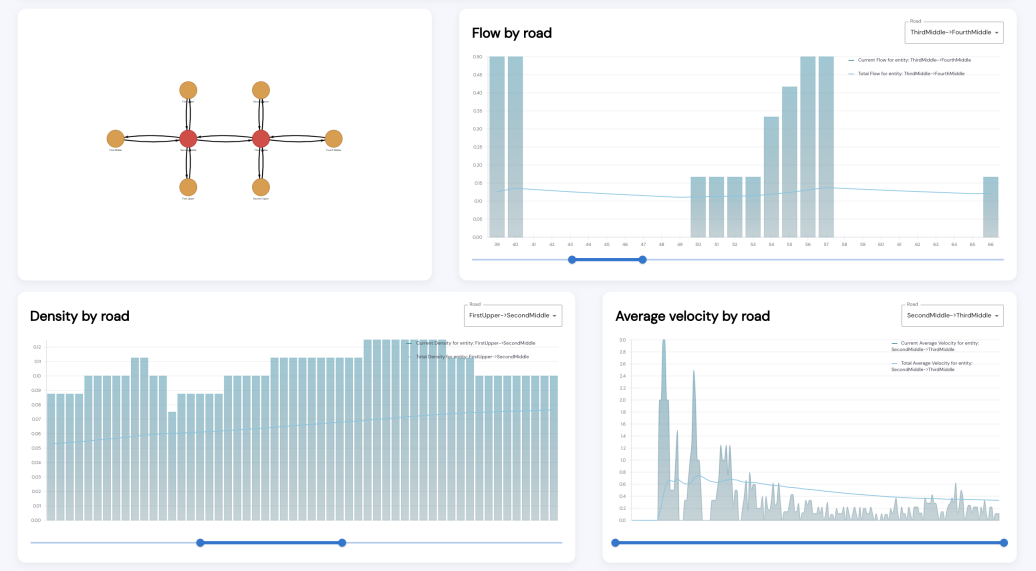
Rysunek 55: Panel akcji przy rekordzie w tabeli symulacji. Znajdują się na nim od lewej: przycisk przenoszący do widoku statystyk, przycisk symulowania oraz pole numeryczne, w którym użytkownik określa, jaką liczbę tur chce zasymulować.

Widok statystyk

Widok statystyk prezentuje wszystkie dane zbierane na przestrzeni symulacji na wykresach liniowo-słupkowych. Dla obecnie gromadzonych danych daje to jeden wykres ogólny: średniej prędkości, oraz 3 wykresy z podziałem na drogę: przepływu, gęstości i średniej prędkości. W celu ustawienia pożądanej drogi należy wybrać odpowiadającą jej nazwę z menu znajdującego się w prawym górnym rogu każdej planszy z wykresem. W celu ułatwienia znalezienia właściwej nazwy drogi dodana została również plansza z mapą, na której odbywała się symulacja. Użytkownik może kontrolować ilość wyświetlanych danych – służy do tego suwak, który znajduje się pod każdym wykresem. Wykres jest responsywny – będzie dostosowywał swój kształt w zależności od wybranego przedziału.



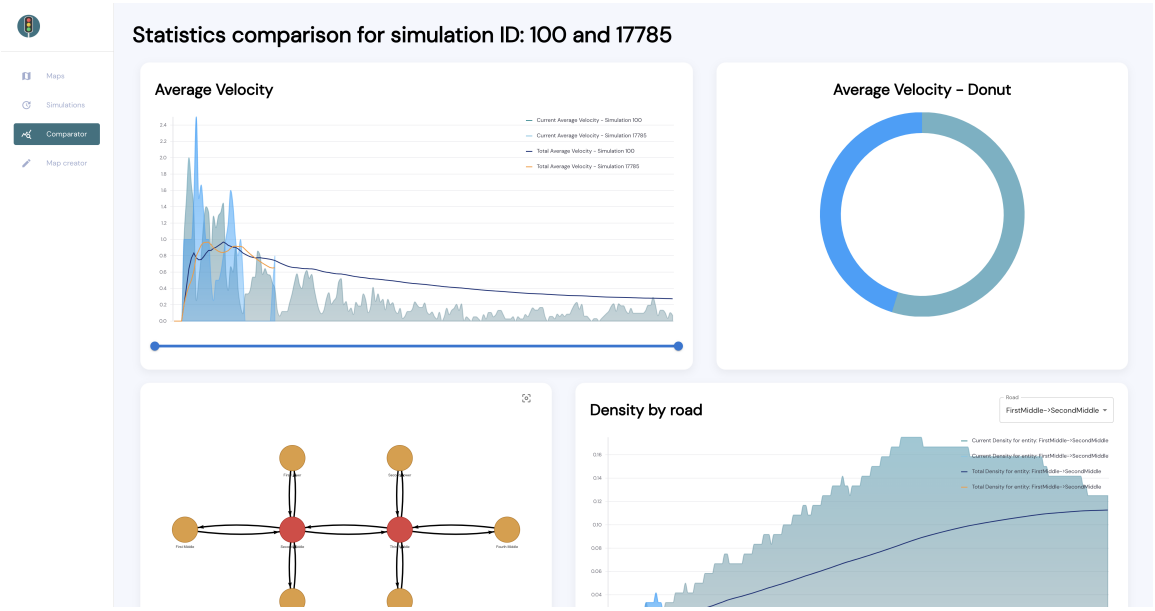
Rysunek 56: Widok statystyk. Fragment pokazujący wykres globalnej średniej prędkości, mapę oraz przepływ z podziałem na drogę. Suwaki są ustawione tak, aby obejmować cały zakres symulacji, przez co wykres zmienia się ze słupkowego na obszarowy.



Rysunek 57: Fragment widoku statystyk pokazujący wykresy z podziałem na drogi. W tym przypadku suwaki są ustawione na mniejszy zakres, więc wyświetlają się wykresy słupkowe.

Widok porównania statystyk

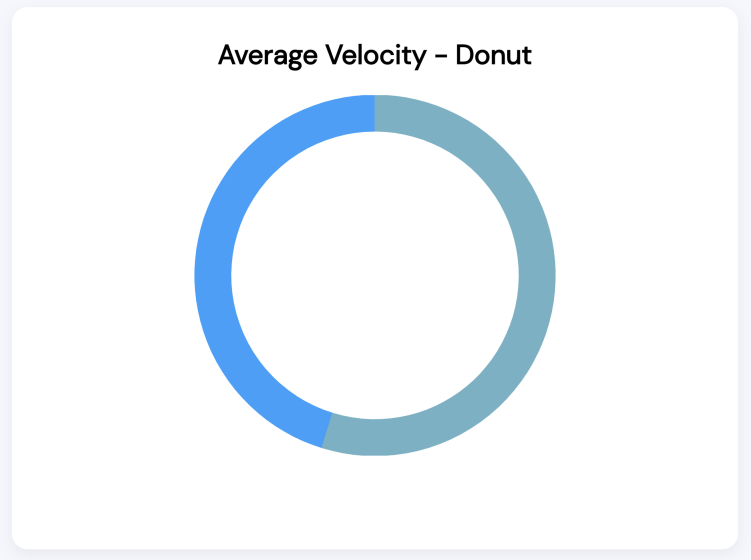
Widok porównania statystyk prezentuje ten sam rodzaj danych z tą różnicą, że na wykresach możemy zobaczyć statystyki dla obu symulacji. Dodatkowo, w widoku tym pojawia się nowa plansza z wykresem kołowym pokazującym, jaki procent tur jednej symulacji miał większą średnia prędkość od tur drugiej symulacji. W widoku porównywanych statystyk pojawia się również tabela pokazująca podstawowe dane porównywanych symulacji (te same dane znajdują się w widoku symulacji).



Rysunek 58: Fragment widoku porównania statystyk pokazujący wykresy z podziałem na drogi.

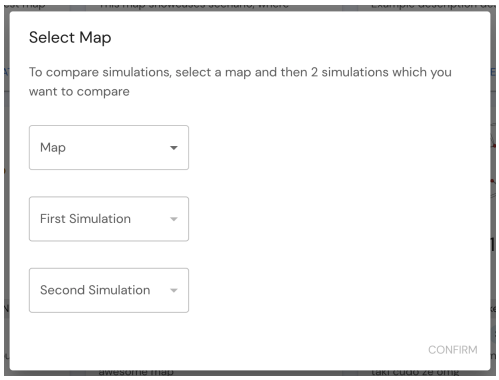
ID	100	17785
Name	Pog champ	Another one
Map ID	1	1
Type	Nagel Core	Nagel Core
Movement type	Nagel Schreckenberg	Nagel Schreckenberg
Turn	204	42
State	▶▶	▶▶

Rysunek 59: Plansza z tabelką porównania danych symulacji.

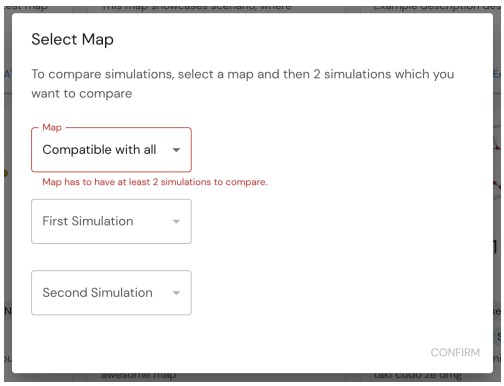


Rysunek 60: Plansza z wykresem kołowym.

W celu przejścia do tego widoku należy wybrać w bocznym panelu wybrać opcje `Compare simulations`,następnie w wyskakującym okienku, które się pojawiło wybrać mapę, z której chcemy porównać statystyki, wybrać nazwy dwóch symulacji i ostatecznie nacisnąć przycisk `Confirm`.



Rysunek 61: Wyskakujące okienko pozwalające wybrać symulacje, które chcemy porównać.



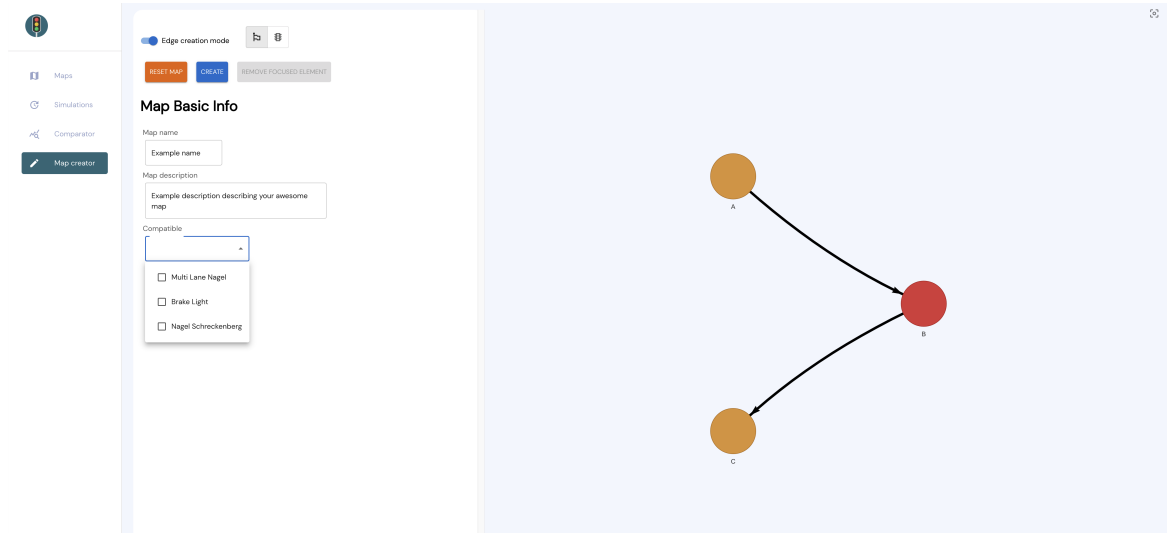
Rysunek 62: Komunikat z błędem, pojawiający się, kiedy wybrana mapa ma mniej niż 2 symulacje do porównania.

Widok kreatora map

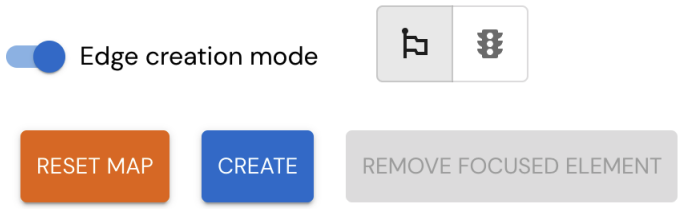
Widok kreatora map składa się z interaktywnej mapy oraz planszy, która dynamicznie wyświetla formularze na podstawie działań na mapie wraz z przyciskami. Zdefiniowane są następujące typy formularzy:

- `Map Basic Info` – Podstawowe informacje o mapie: nazwa, opis oraz typy symulacji, z którymi ta mapa jest kompatybilna,
- `Gateway Basic Info` – Pozycja oraz nazwa węzła, pojawia się po naciśnięciu na daną bramę,
- `Intersection Basic Info` – Pozycja, nazwa węzła, przełącznik ustawiający automatyczną generację kierunków skrętu oraz pola wejściowe pozwalające te skręty zdefiniować ręcznie, pojawia się po naciśnięciu na dane skrzyżowanie,
- `Road Basic Info` – Nazwa drogi oraz pola wejściowe pozwalające zdefiniować pasy, pojawia się po naciśnięciu na daną drogę.

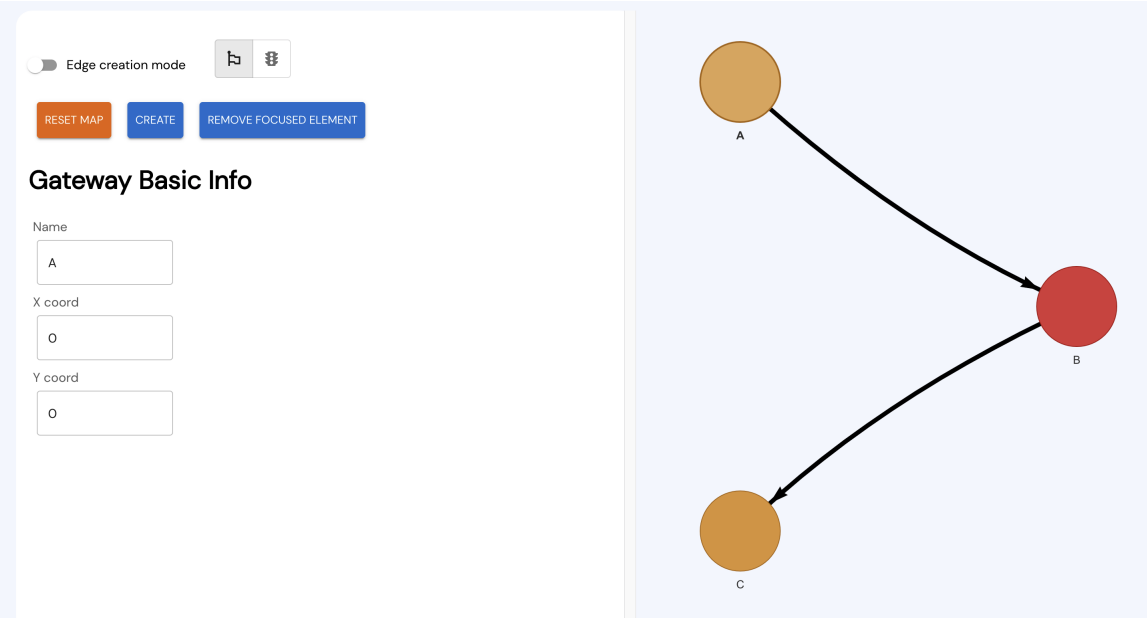
Naciskając na planszy podwójnym przyciskiem myszy w miejscu kursora, zostaje dodany węzeł. Rodzaj dodanego węzła zależy od przełącznika widocznego na rysunku 64. Aby utworzyć drogę, należy włączyć tryb tworzenia krawędzi również widoczny na rysunku 64, a następnie wybrać dwa węzły, między którymi ta droga ma się pojawić.



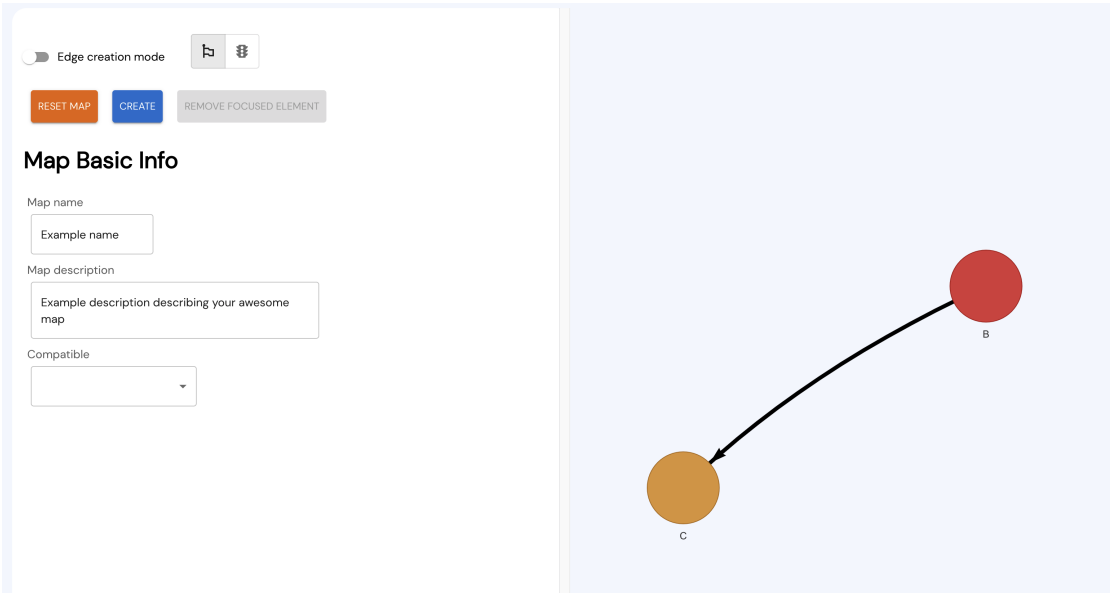
Rysunek 63: Widok tworzenia mapy w początkowym stanie. W momencie kiedy kiedy żaden z elementów na mapie nie jest wybrany, formularz wyświetla podstawowe dane mapy.



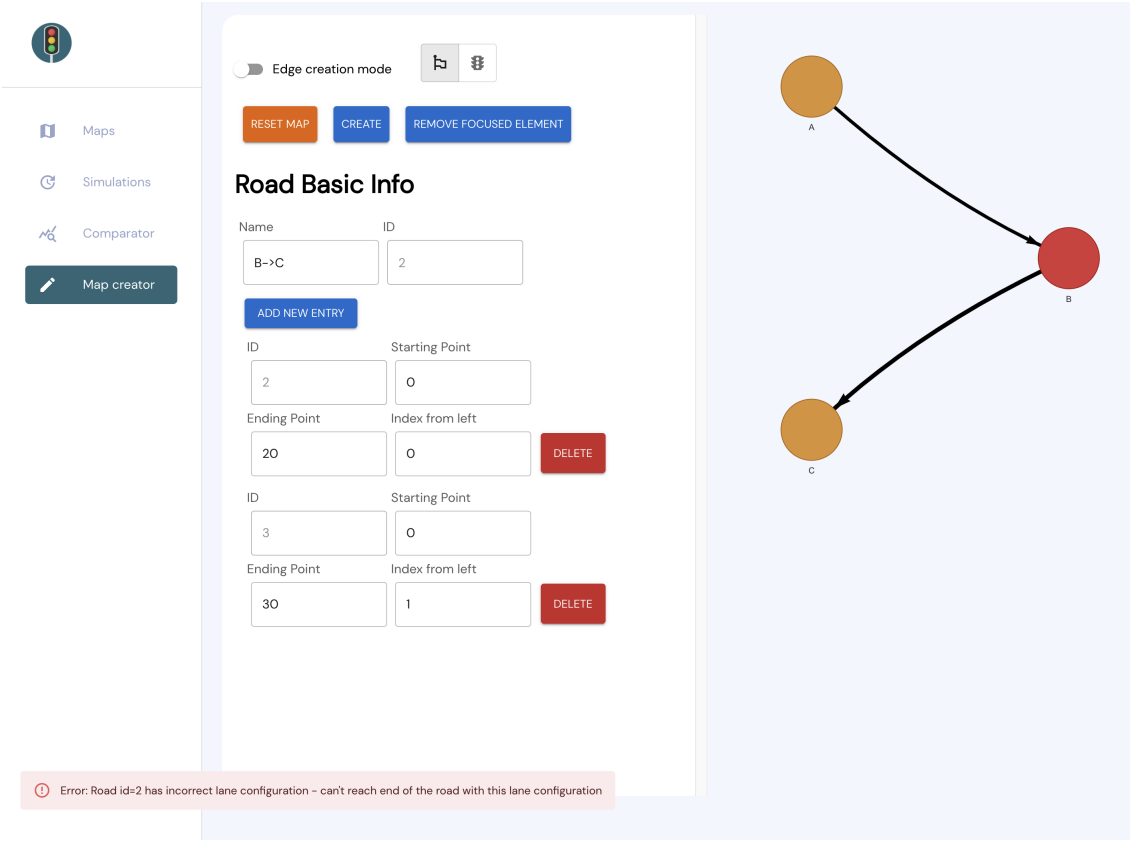
Rysunek 64: Przyciski pomagające w tworzeniu mapy. W lewym górnym rogu znajduje się włącznik trybu tworzenia krawędzi, w prawym górnym rogu przełącznik, jakiego typu nowo utworzone są węzły (pierwsza ikona oznacza tworzenie bram, druga skrzyżowań), a poniżej kolejno: przycisk resetowania mapy (przywraca stan mapy do początkowego ustawienia); przycisk wysyłający zapytanie o utworzenie mapy; przycisk usuwający element aktualnie naciśnięty.



Rysunek 65: Widok tworzenia mapy z naciśniętą bramą A. W lewym panelu wyświetlają się podstawowe informacje, które można zmienić w bramie.



Rysunek 66: Widok tworzenia mapy po naciśnięciu przycisku `Remove focused element` w momencie, kiedy była naciśnięta brama A. Spowodowało to usunięcie tej bramy oraz wszystkich wychodzących i przychodzących dróg (w tym przypadku droga biegnąca do skrzyżowania B).



Rysunek 67: Widok tworzenia mapy po próbie utworzenia mapy z błędną konfiguracją. W tym przypadku utworzona konfiguracja pasów nie pozwala na przejazd samochodu na drodze ze skrzyżowania B do bramy A. Błędna droga pokazana jest na formularzu, jeden pas kończy się po dwudziestu metrach, a drugi pas zaczyna od trzydziestego metra.

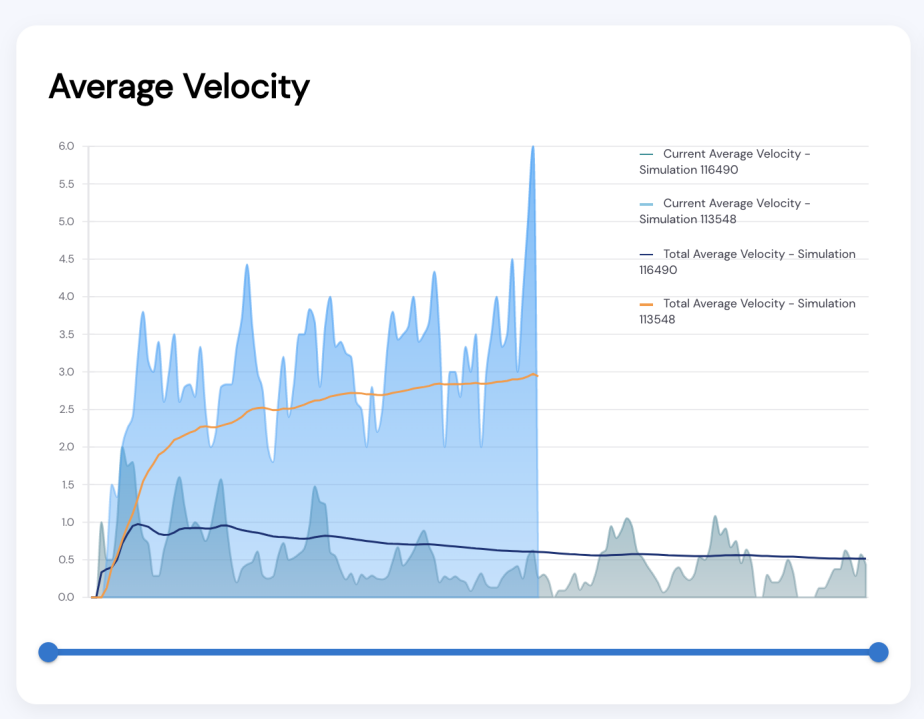
5.2. Wybrane wyniki symulacji

W celu objaśnienia różnic pomiędzy poszczególnymi parametrami symulacji zdecydowaliśmy się przytoczyć oraz objaśnić kilka przykładowych wyników. Zostały one wykonane na bardzo prostych mapach, ponieważ zależało nam na tym, żeby różnice w wynikach wynikały z inaczej zdefiniowanych symulacji, a nie na przykład ze stopnia skomplikowania trasy.

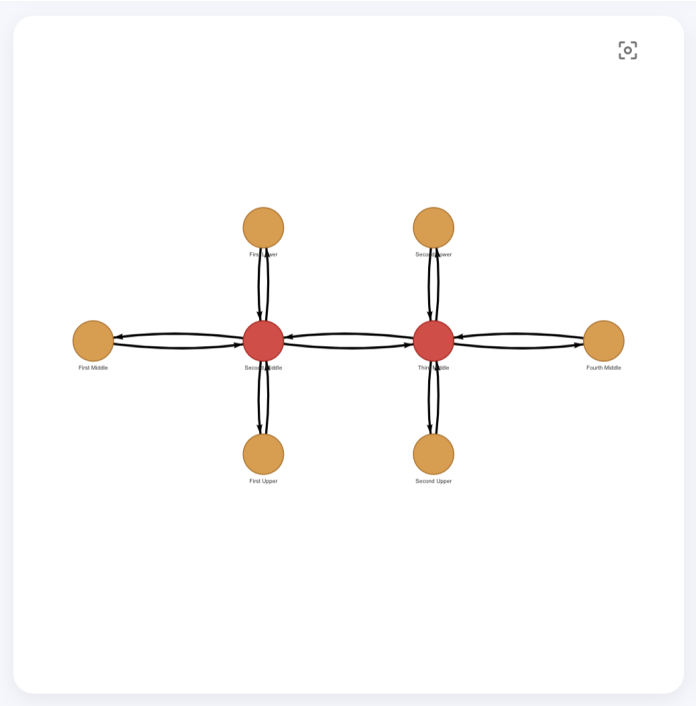
5.2.1. Porównanie algorytmów

SOTL oraz algorytm turowy

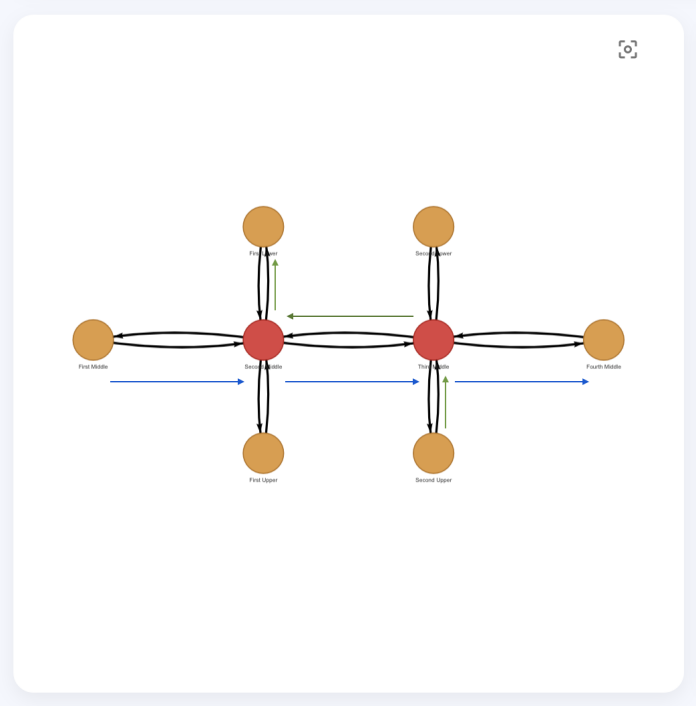
Wybór algorytmu sterowania światłami wpływa na to, jak często samochody stoją na czerwonym świetle, oraz jaka ich ilość zdąży przejechać podczas jednej zmiany. Ma to znaczący wpływ na średnią prędkość ruchu, ponieważ stojące na światłach samochody – takie, których prędkość wynosi 0, znacząco ją obniżają, W przypadku zestawienia obok siebie tych dwóch algorytmów, różnica jest znacząca – SOTL upewnia się, że każde auto, które czekało na czerwonym świetle, będzie w stanie przejechać drogę przy następnej okazji. Skutkuje to znacząco lepszymi wynikami dla algorytmu SOTL.



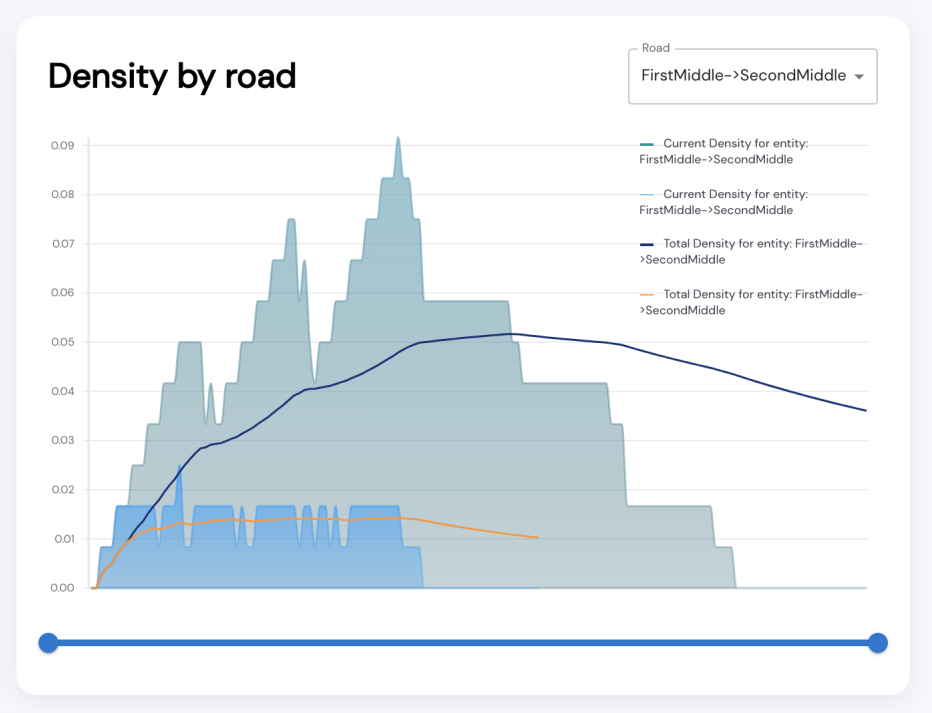
Rysunek 68: Wykres globalnej średniej prędkości dla algorytmów SOTL (niebieski) oraz turowego (seledynowy). Widać na nim dużych rozmiarów przewagę pierwszego algorytmu na przestrzeni całej symulacji. Momentami prędkość średnia w symulacji z algorytmem SOTL była nawet 6 razy większa niż przy wersji turowej.



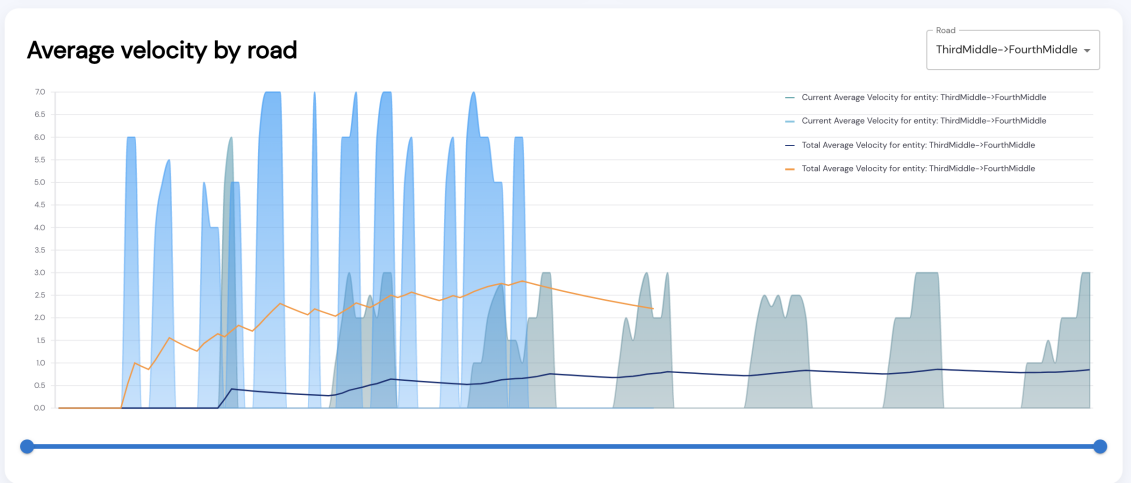
Rysunek 69: Mapa, na której przeprowadzone zostały symulacje.



Rysunek 70: Mapa z zaznaczonymi trasami zdefiniowanych generatorów.



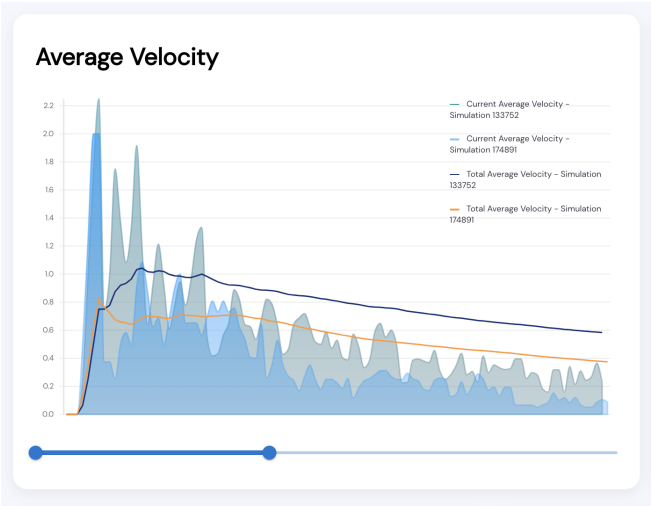
Rysunek 71: Wykres gęstości ruchu na drodze od trzeciego od lewej skrzyżowania na środkowym rzędzie do ostatniej bramy w środkowym rzędzie. W tym przypadku z tego, że momentami gęstość spada do 0, wynika, że wszystkie czekające auta są w stanie przejechać przez skrzyżowanie. W przypadku algorytmu turowego te spadki występują co 10-11 tur, a w przypadku SOTL odstępy między przerwami wahają się między 2 a 4 turami.



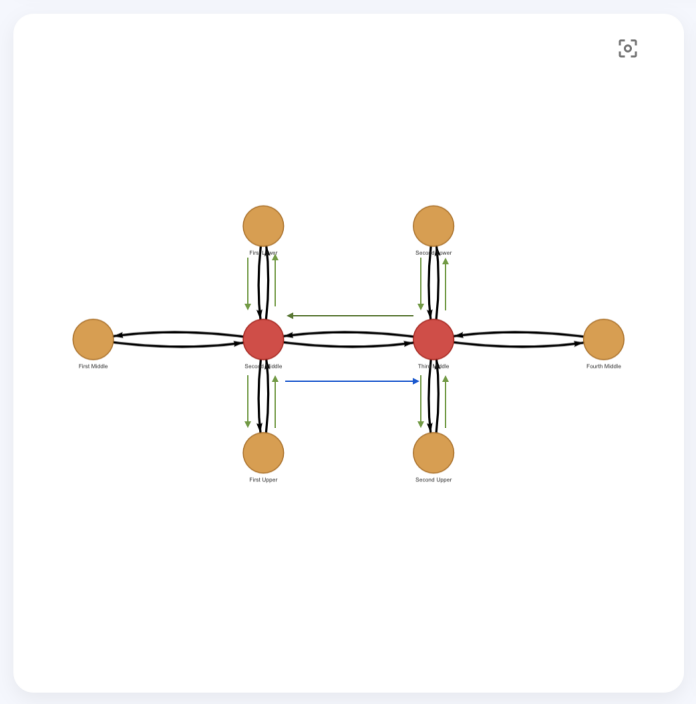
Rysunek 72: Wykres średniej prędkości ruchu na drodze od trzeciego od lewej skrzyżowania na środkowym rzędzie do ostatniej bramy w środkowym rzędzie. W tym przykładzie algorytmy różnią się tym, że samochody pokonujące skrzyżowanie w przypadku SOTL utrzymują większą prędkość, niż w algorytmie turowym. Wynika to z tego, że przy algorytmie turowym więcej samochodów na raz chce przejechać przez skrzyżowanie i doprowadzają do korka.

Algorytmy symulowania ruchu

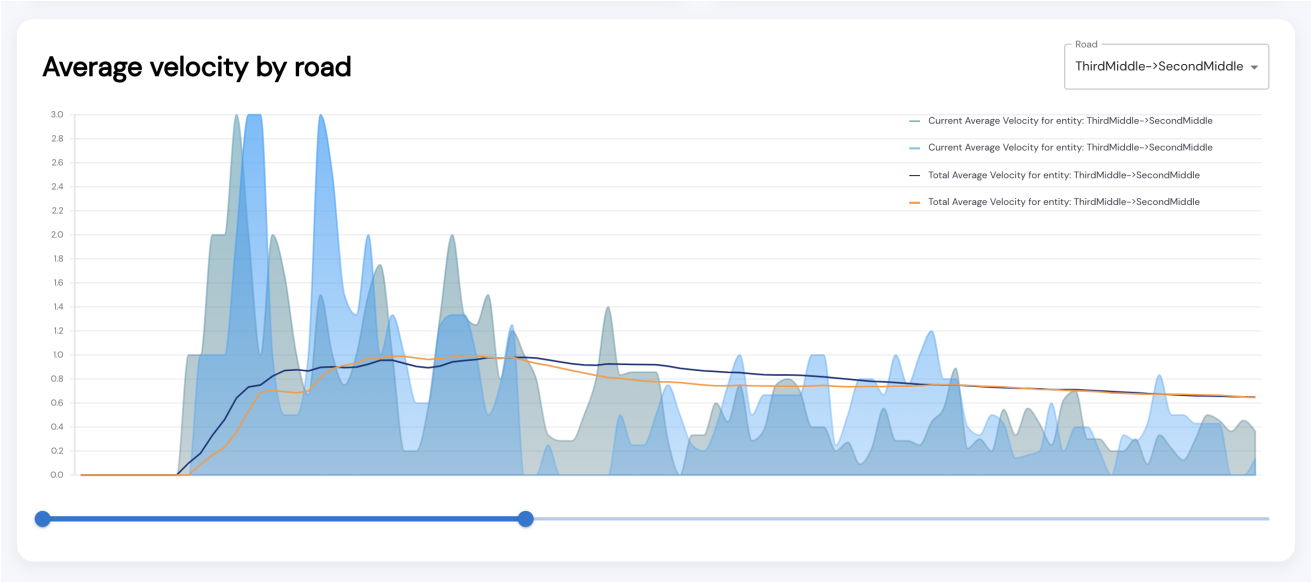
W naszej pracy zaimplementowaliśmy trzy algorytmy symulowania ruchu: podstawowy Nagel-Schreckenberg, modyfikację wielopasmową oraz modyfikację Brake Light. Jako że każdy z tych algorytmów jest oparty na pierwszym z nich, podczas przeprowadzania symulacji różnica nie jest znacząca. Małe wahania w wynikach wynikają też z tego, że ciężko jest porównywać algorytm wielopasmowy do jednopasmowego — działają na innej zasadzie.



Rysunek 73: Wykres globalnej średniej prędkości przy porównaniu algorytmu Nagela-Schreckenberga i Brake Light. Jako że drugi model bierze pod uwagę zachowanie świateł stopu samochodów, i na ich podstawie decyduje o hamowaniu, to średnia prędkość w tym przypadku jest znacznie niższa. Mimo to drugi algorytm jest lepszy w odwzorowaniu rzeczywistego zachowania kierowców.



Rysunek 74: Mapa z zaznaczonymi przebiegami tras wygenerowanych samochodów. To na niej przeprowadzane były symulacje sprawdzające zachowanie algorytmów.



Rysunek 75: W przypadku niektórych dróg różnica między zachowaniem algorytmów jest marginalna.

Przy porównaniu modeli Nagela-Schreckenberga oraz jego wielopasmowej wersji otrzymaliśmy praktycznie nierozróżnialne wyniki. Wynika to z tego, że kształt wybranej przez nas mapy (głównie jej liczba pasów na każdej z dróg) był zbyt prosty do uchwycenia różnicy. Na mapie nie był również obecny żaden pas do skrętu w konkretnym kierunku — Po zdefiniowaniu takiego samochody w modelu wielopasmowym musiałyby zmienić pas, co na pewno ujawniłoby różnice. Jednakże wtedy mapa nie byłaby kompatybilna z podstawową wersją algorytmu Nagela-Schreckenberga, więc niemożliwe byłoby ich porównanie. Z powodu nieznacznej różnicy wyników, nie zamieszczamy wykresu dla tego rodzaju symulacji.

5.3. Dalszy rozwój projektu

5.3.1. Wizualizacja symulacji

Z powodu niedostatecznej ilości czasu nie udało nam się zaimplementować żadnej formy wizualizacji symulacji. Użytkownik może jedynie zobaczyć statystyki zebrane podczas jej trwania w formie wykresów. Dane potrzebne do prezentacji każdej tury są zbierane w bazie danych, żeby uzyskać funkcjonalność wizualizacji, wystarczy zaimplementować odpowiedni widok po stronie klienta.

5.3.2. Tworzenie mapy na podstawie prawdziwej trasy

W trakcie planowania przebiegu pracy postanowiliśmy stworzyć kreator map na podstawie biblioteki umożliwiającej wizualizację prawdziwego elementu Ziemi (Open Street Map lub Google Maps). Z uwagi na to, że nasz model pozwala użytkownikowi definiować nieograniczoną ilość pasów na drodze, rysowanie siatki mapy było bardzo uciążliwe. Możliwość tworzenia mapy na podstawie prawdziwej trasy nie miała wysokiego priorytetu, dlatego zamieniliśmy ją na prostszą formę wizualizacji.

5.3.3. Porównywanie statystyk z różnych map

Obecnie aplikacja nie pozwala na porównanie symulacji z różnych map – mimo że mogą mieć one część wspólną (w skrajnym przypadku mogą się różnić tylko jedną drogą). Dodanie takiej możliwości pozwoliłoby lepiej badać zależność kształtu mapy od jakości ruchu. Porównanie symulacji z różnych map umożliwiłoby np. zbadanie struktury ruchu po zablokowaniu lub zwężeniu głównej ulicy.

5.3.4. Powiadomienie o ukończeniu asynchronicznego symulowania

W przypadku gdy przetwarzanie oraz zapis symulacji trwa dłużej niż 15 sekund, aplikacja klienta wyświetli informacje o tym, że symulacja nadal jest przetwarzana. Aby sprawdzić, czy symulacja została ukończona, użytkownik jest zmuszony odświeżać widok i porównywać liczbę tur danej symulacji. W celu uniknięcia powyższej niedogodności należy dodać możliwość powiadomienia osoby, która stworzyła symulację, o jej poprawnym zakończeniu i zapisaniu do bazy lub o wystąpieniu błędu. Można to zrealizować poprzez wysłanie wiadomości e-mail na podany przez użytkownika adres lub poprzez zastosowanie powiadomień typu web-push.

5.3.5. Stronicowanie odpowiedzi serwera

W nowoczesnych aplikacjach webowych operujących na dużych zbiorach danych standardem jest wprowadzenie stronicowania – dzięki niemu klient nie będzie musiał pobrać danych, których użytkownik może nie zobaczyć. Pozwoli to również uniknąć przekroczenia limitu czasu połączenia przy dużej ilości zdefiniowanych map, symulacji lub statystyk. Używana przez nas biblioteka Spring posiada narzędzia do realizacji stronicowania, więc jego implementacja nie powinna stanowić problemu. Zmiany dotkną głównie aplikację klienta, gdyż będzie trzeba wprowadzić ograniczenie ilości rekordów symulacji i map mogących pojawić się na jednej stronie. W przypadku widoku statystyk wprowadzenie stronicowania będzie trudniejsze, ponieważ wymaga ono uzależnienia pobierania danych od przesuwania suwaka przez użytkownika

5.3.6. Usprawnienie warstwy persystencji

Naszą początkową decyzją było wykorzystanie relacyjnej bazy danych, głównie ze względu na naszą znajomość tego rodzaju baz danych. Jednakże, kiedy nasz system był już na tyle zaawansowany, że mogliśmy testować symulacje z wieloma samochodami na większych mapach, okazało się, że nawet niewielka liczba tur powoduje powstanie ogromnej ilości rekordów w tabeli (rzędu 10 tysięcy), co przekłada się na długi czas zapisu. Pozostała ilość czasu na realizacje projektu nie pozwoliła nam na zmianę decyzji. Aby usprawnić nasz system, należałoby zastosować dokumentową bazę danych. Muszą się w niej znaleźć trzy kolekcje, przechowujące dane map, symulacji i statystyk. Z powodu małej ilości relacji pomiędzy tymi kolekcjami nie ma konieczności wykonywania złożeń pomiędzy nimi, więc każdą z encji można przechowywać osobno.

5.3.7. Moduł GPS na podstawie zebranych statystyk

Kolejną ciekawą funkcjonalnością, wartą dodania, jest moduł generujący optymalną trasę na podstawie statystyk, które zostały zebrane w poprzednich turach. Przygotowana abstrakcja modułu pozwala na łatwą implementację własnej strategii. Wystarczy rozszerzyć klasę `DijkstraBasedGPS` i nadpisać metodę `calculate`, podając funkcję anonimową jako argument funkcji `calculate` z klasy nadrzędnej `DijkstraBasedGPS`, która na podstawie

danej drogi zwraca wagę wykorzystaną w algorytmie Dijkstry. Statystyki zawierają informację o gęstości ruchu na każdej drodze, więc taki algorytm mógłby wybierać bardziej przejezdną trasę.

```
@Component
class RoadLengthGPS : DijkstraBasedGPS() {

    fun calculate(source: Gateway, target: Gateway, state: SimulationState): GPS =
        super.calculate(
            source, target, state,
            { road ->
                road.physicalLength.toDouble()
            },
            GPSType.DIJKSTRA_ROAD_LENGTH
        )
}
```

Rysunek 76: Implementacja klasy `RoadLengthGPS`. Wywołana funkcja z klasy nadrzędnej jako czwarty argument przyjmuje lambdę, która zwraca fizyczną długość trasy.

5.3.8. Implementacja kolejnych strategii

Z powodu niedostatecznej ilości czasu nie udało się nam również zaimplementować wszystkich strategii opisanych we wstępie teoretycznym. Pominięty został makroskopowy model ruchu Greenshielda oraz mikroskopowy model ruchu VDR. Dodatkowo w poprzednim systemie Kraksim istniały inne modyfikacje, o których nie wspominaliśmy w tej pracy. Jedną z nich jest strategia sterowania światłami na podstawie modelu uczenia maszynowego. Model ten, wykorzystując możliwości grupowania kilku skrzyżowań do jednej strategii, mógłby optymalizować przepływ samochodów.

5.4. Ocena końcowa

Nie udało nam się zrealizować wszystkich założonych funkcjonalności, ale mimo to jesteśmy zadowoleni z produktu końcowego. Aplikacja spełnia swoją rolę w podstawowym zakresie, jest dopracowana wizualnie (m.in. odpowiednie stylowanie aplikacji klienta, zastosowanie indykatorów ładowania) oraz jest gotowa do wdrożenia na produkcję. Za pomocą narzędzia SLOC [34] wygenerowaliśmy statystyki kodu źródłowego. Aplikacja serwera posiada 6641 znaczących linijek kodu, a aplikacja klienta 6108. Uważamy, że taki rozmiar w pełni spełnia wymagania pracy inżynierskiej.

```
gporeba@C02F20RWMD6V Kraksim % sloc ./ -e '.idea'

---------- Result ------------

            Physical :  8094
              Source :  6641
             Comment :  368
 Single-line comment :  228
       Block comment :  140
               Mixed :  6
 Empty block comment :  2
               Empty :  1093
               To Do :  0

Number of files read :  155

---------------------------
```

Rysunek 77: Wyniki analizy znaczących linii kodu dla projektu klienckiego.

```
gporeba@C02F20RWMD6V KraksimGUI % sloc ./ -e 'node_modules|.idea'

---------- Result ------------

            Physical :  7269
              Source :  6108
             Comment :  520
 Single-line comment :  455
       Block comment :  65
               Mixed :  17
 Empty block comment :  3
               Empty :  661
               To Do :  2

Number of files read :  77

---------------------------
```

Rysunek 78: Wyniki analizy znaczących linii kodu dla projektu serwera.

Materiały źródłowe

[1] React. https://reactjs.org/.

[2] Formik. https://formik.org/.

[3] Material UI. https://mui.com/.

[4] Redux. https://redux.js.org/.

[5] Hooks at a Glance. https://reactjs.org/docs/hooks-overview.html.

[6] React context. https://reactjs.org/docs/context.html.

[7] Spring Boot. https://spring.io/projects/spring-boot.

[8] Spring Boot Starter Validation. https://spring.io/guides/gs/validating-form-input/.

[9] Spring Boot Starter Web. https://spring.io/guides/gs/spring-boot/.

[10] Spring Boot Data JPA. https://spring.io/projects/spring-data-jpa.

[11] Java. https://www.java.com/en/download/help/whatis_java.html.

[12] Kotlin. https://kotlinlang.org/.

[13] TypeScript. https://www.typescriptlang.org/.

[14] JVM. https://en.wikipedia.org/wiki/Java_virtual_machine.

[15] Kotlin null safety. https://kotlinlang.org/docs/null-safety.html.

[16] Kotlin function extension. https://kotlinlang.org/docs/extensions.html.

[17] Kotlin properties. https://kotlinlang.org/docs/properties.html#overriding-properties.

[18] PostgreSQL. https://www.postgresql.org/.

[19] Hibernate. https://hibernate.org/.

[20] JPA query language. https://docs.spring.io/spring-data/data-jpa/docs/current/reference/html/#jpa.query-methods.

[21] Kapt. https://kotlinlang.org/docs/kapt.html.

[22] Docker. https://www.docker.com/.

[23] Mapstruct. https://mapstruct.org/.

[24] Redux-Toolkit. https://redux-toolkit.js.org/.

[25] Overleaf. https://www.overleaf.com/.

[26] LaTeX. https://www.latex-project.org/.

[27] Discord. https://discord.com/.

[28] Ktlint. https://ktlint.github.io/.

[29] Draw.io. https://app.diagrams.net/.

[30] Intellij IDEA. https://www.jetbrains.com/idea/.

[31] Paint. https://pl.wikipedia.org/wiki/Microsoft_Paint.

[32] Swagger. https://swagger.io/.

[33] Single Page Application. https://en.wikipedia.org/wiki/Single-page_application.

[34] SLOC. https://github.com/flosse/sloc.

[35] Repozytorium kodu serwera. https://github.com/Kraksim/Kraksim.

[36] webpack. https://webpack.js.org/.

[37] Repozytorium kodu klienta. https://github.com/Kraksim/KraksimGUI.

[38] react-vis. https://uber.github.io/react-vis/.

[39] vis.js. https://visjs.org/.

[40] create-react-app. https://create-react-app.dev/.

[41] emotion. https://emotion.sh/docs/introduction.

[42] react-router. https://reactrouter.com/.

[43] R. Barlovic, L. Santen, A. Schadschneider, i M. Schreckenberg. Metastable States in CA models for Traffic Flow. *The European Physical Journal B-Condensed Matter and Complex Systems, vol. 5(3), pp. 793–800*, 1998.

[44] S.-B. Cools, C. Gershenson, i B. D'Hooghe. Self-organizing traffic lights: A realistic simulation. 2006.

[45] T. V. Mathew. Transportation systems engineering. *IIT Bombay*, 2014.

[46] Matyjewicz, Rybacki, Adamski, Pierzchała, Dziewoński, Zalewski, Kot, Skałka, Doliński, Skowron, Legień, Bibro, Bober, Liput, Haręza, Ostaszewski, Dygoń, Kawula, Ćwik, Kruczek, Grabiański, i Królikowski. Dokumentacja programisty systemu kraksim. *AGH Kraków*, 2016.

[47] K. Nagel i M. Schreckenberg. A cellular automaton model for freeway traffic. *Journal de Physique*, 1992.

[48] K. Nagel, P. E. Stretz, M. Pieck, i S. Leckey. TRANSIMS traffic flow characteristics. 1998.

[49] T. Nizio. Optymalizacja zachowań kierowców w ruchu drogowym. *Praca magisterska, IET AGH, Kraków*, 2020.

[50] S. A. O'Brien. The pandemic boosted food delivery companies. soon they may face a reality check. *CNN Business*, 2021.

[51] B. Rybacki. Modelowanie i optymalizacja ruchu miejskiego przy użyciu wybranych technik. *Praca magisterska, EAIiE AGH, Kraków*, 2008.

[52] J. Smith. Logistics Providers See E-Commerce Momentum Continuing Post-Pandemic. *Wall Street Journal*, 2021.